JAWAHARLAL COLLEGE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(NBA Accredited)

COURSE MATERIAL

CST 201 DATA STRUCTURES

VISION OF THE INSTITUTION

Emerge as a centre of excellence for professional education to produce high quality engineers and entrepreneurs for the development of the region and the Nation.

MISSION OF THE INSTITUTION

- To become an ultimate destination for acquiring latest and advanced knowledge in the multidisciplinary domains.
- To provide high quality education in engineering and technology through innovative teaching-learning practices, research and consultancy, embedded with professional ethics.

- To promote intellectual curiosity and thirst for acquiring knowledge through outcome based education.
- To have partnership with industry and reputed institutions to enhance the employability skills of the students and pedagogical pursuits.
- To leverage technologies to solve the real life societal problems through community services.

ABOUT THE DEPARTMENT

- Established in: 2008
- Courses offered: B.Tech in Computer Science and Engineering
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To produce competent professionals with research and innovative skills, by providing them with the most conducive environment for quality academic and research oriented undergraduate education along with moral values committed to build a vibrant nation.

DEPARTMENT MISSION

- Provide a learning environment to develop creativity and problem solving skills in a professional manner.
- Expose to latest technologies and tools used in the field of computer science.
- Provide a platform to explore the industries to understand the work culture and expectation of an organization.
- Enhance Industry Institute Interaction program to develop the entrepreneurship skills.
- Develop research interest among students which will impart a better life for the society and the nation.

PROGRAMME EDUCATIONAL OBJECTIVES

Graduates will be able to

- Provide high-quality knowledge in computer science and engineering required for a computer professional to identify and solve problems in various application domains.
- Persist with the ability in innovative ideas in computer support systems and transmit the knowledge and skills for research and advanced learning.
- Manifest the motivational capabilities, and turn on a social and economic commitment to community services.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COURSE OUTCOMES

SUBJECT CODE: C202	
COURSE OUTCOMES	
C202.1	**Design** an algorithm for a computational task and calculate and analyze the time/space complexities of that algorithm.
C202.2	**Identify and Use** appropriate data structures like arrays, linked list, stacks and queues to solve computational problems efficiently.
C202.3	**Categorize** different memory management techniques and the implementations of linear data structures.
C202.4	**Represent** and manipulate data using nonlinear data structures like trees and graphs to design algorithms for various applications.
C202.5	**Illustrate and understand** various techniques for searching ,sorting and hashing algorithms
C202.6	**Design** an algorithm for a computational task and calculate and analyze the time/space complexities of that algorithm.

PROGRAM SPECIFIC OUTCOMES (PSO)

The students will be able to

- Use fundamental knowledge of mathematics to solve problems using suitable analysis methods, data structure and algorithms.
- Interpret the basic concepts and methods of computer systems and technical specifications to provide accurate solutions.
- Apply theoretical and practical proficiency with a wide area of programming knowledge, design new ideas and innovations towards research.

CO PO MAPPING

Note: H-Highly correlated=3, M-Medium correlated=2,L-Less correlated=1

CO'S	P	P	P	P	P	P	P	P	P	PO 10	PO 11	PO 12
C202.1	3	3	2	2	-	1	-	-	-	-	-	2
C202.2	2	2	3	2	-	1	-	-	-	-	-	1
C202.3	3	2	3	2	-	2	-	-	-	-	-	1
C202.4	3	1	2	3	-	1	-	-	-	-	-	2
C202.5	3	2	3	3	-	1	-	-	-	-	-	2

C202.6	3	2	3	2	-	2	-	-	-	-	-	2
C202	2	2	2.	2	-	2						2

CO PSO MAPPING

CO'S	PSO1	PSO2	PSO3
C202.1	2	3	1
C202.2	2	2	-
C202.3	1	2	-
C202.4	2	2	2
C202.5	1	2	-
C202.6	3	3	-
C202	2	2.3	1

GAPS IN THE SYLLABUS

S:NO	TOPIC
1	TOWER OF HANOI PROBLEM
2	

Reference Materials

Module 1 Basic Concepts of Data Structures

System Life Cycle, Algorithms, Performance Analysis, Space Complexity, Time Complexity, Asymptotic Notation, Complexity Calculation of Simple Algorithms

SYSTEM LIFE CYCLE (SLC)

- Good programmers regard large scale computer programs as systems that

contain manycomplex interacting parts. (Systems: Large Scale Computer Programs.)

- As systems, these programs undergo a development process called System life cycle.(SLC : Development Process of Programs)

Different Phases of System Life Cycle

1. Requirements
2. Analysis
3. Design
4. Refinement and coding
5. Verification

1. Requirement Phase:

- All programming projects begin with a **set of specifications** that defines the purpose ofthat program.
- Requirements describe the information that the programmers are given (**input**) and theresults (**output**) that must be produced.
- Frequently the initial specifications are defined vaguely and we must develop rigorousinput and output descriptions that include all cases.

2. Analysis Phase

- In this phase the problem is break down into manageable pieces.
- There are two approaches to analysis:-**bottom up and top down**.
- Bottom up approach is an older, **unstructured** strategy that places an early emphasis on coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many **loosely connected, error ridden segments**.
- Top down approach is a structured approach divide the program into manageable segments.

- This phase generates **diagrams** that are used to design the system.
- Several alternate solutions to the programming problem are developed and comparedduring this phase

3. Design Phase

- This phase continues the work done in the analysis phase.
- The designer approaches the system from the perspectives of both **data objects** that theprogram needs and the **operations** performed on them.
- The first perspective leads to the **creation of abstract data types** while the second requiresthe **specification of algorithms** and a consideration of algorithm design strategies.

Ex: Designing a scheduling system for universityData objects: Students, courses, professors etc Operations: insert, remove search etc

ie. We might add a course to the list of university courses, search for the courses taughtby some professor etc.

- Since abstract data types and algorithm specifications are language independent.
- We must specify the information required for each data object and ignore coding details.Ex: Student object should include name, phone number, social security number etc.

4. Refinement and Coding Phase

- In this phase we choose representations for data objects and write algorithms for each operation on them.
- Data objects representation can determine the efficiency of the algorithm related to it. So we should write algorithms that are independent of data objects first.
- Frequently we realize that we could have created a much better system. (May be we realize that one of our alternate design is superior than this). If our original design is

good, it can absorb changes easily.

5. **Verification Phase**

- This phase consists of
  - developing correctness proofs for the program

- Testing the program with a variety of input data.
- Removing errors.

Correctness of Proofs

- Programs can be proven correct using proofs.(like mathematics theorem)
- Proofs are very time consuming and difficult to develop for large projects.
- Scheduling constraints prevent the development of complete sets of proofs for a larger system.
- However, selecting algorithm that have been proven correct can reduce the numberof errors.

Testing

- Testing can be done only after coding.
- Testing requires working code and set of test data.
- Test data should be chosen carefully so that it includes all possible scenarios.
- Good test data should verify that every piece of code runs correctly.
- For example if our program contains a *switch* statement, our test data should be chosen so that we can check each *case* within *switch* statement.

Error Removal

- If done properly, the correctness of proofs and system test will indicate erroneous code.
- Removal of errors depends on the design and code.
- While debugging large undocumented program written in 'spaghetti' code, each corrected error possibly generates several new errors.
- Debugging a well-documented program that is divided into autonomous units that interact through parameters is far easier. This especially true if each unit is tested separately and then integrated into system.

ALGORITHMS

Definition: An ***algorithm*** is a finite set of instructions to accomplish a particular task. In addition,all algorithms must satisfy the following criteria:

(1) ***Input***. There are zero or more quantities that are externally supplied.

(2) ***Output***. At least one quantity is produced.

(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); italso must be feasible.

We can describe algorithm in many ways

1. We can use a natural language like English
2. Graphical Representation called flow chart, but they work well only if the algorithm issmall and simple.

Translating a Problem into an Algorithm

Example [Selection sort]: Suppose we must devise an algorithm that sorts a collection of n > 1elements of arbitrary type. A simple solution is given by the following

[Selection Sort: In each pass of the selection sort, the smallest element is selected from theunsorted list and exchanged with the elements at the beginning of the unsorted list]

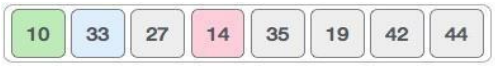
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first positionwhere 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in thelist, appears in the first position of the sorted list

For the second position, where 33 is residing, we start scanning the rest of the list in a linearmanner.

We find that 14 is the second lowest value in the list and it should appear at the second place. Weswap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.Following is a pictorial depiction of the entire sorting process −

- From those elements that are currently unsorted, find the smallest and place it next in

sortedlist

- We assume that the elements are stored in an array ‘list’, such that the i^{th} integer is stored inthe i^{th} Position list[i], 0 <= i <n
- Algorithm 1.1 is our first attempt to deriving a solution

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

1.1 Selection sort algorithm

- We are written this partially in C and partially in English
- To turn the program 1.1 into a real C program, two clearly defined sub tasks are remain: **finding the smallest integer** and **interchanging it with list[i].**
- We can solve this by using a function

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;   /* declares temp as an int and assigns
                       to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

1.2 Swap Function

- To swap their values one could call swap(&a, &b)
- We can solve the first subtask by assuming that the minimum is the list[i]. Checking list[i] with list[i+1], list[i+2]……,list[n-1]. Whenever we find a smaller number we make it as the minimum. We reach list[n-1] we are finished.

```
#include <stdio.h>int
    main()
{
int a[100], n, i, j, position, swap; printf("Enter
    number of elements");scanf("%d", &n);
```

```
printf("Enter %d Numbersn", n);for (i = 0; i < n;
     i++) scanf("%d", &a[i]);

for(i = 0; i < n - 1; i++)

{

position=i;
```

```
for(j = i + 1; j < n; j++)
{
if(a[position] > a[j])position=j;
}
if(position != i)
{
swap=a[i]; a[i]=a[position]; a[position]=swap;
}
}
printf("Sorted Array:n");for(i = 0; i < n;
    i++) printf("%dn", a[i]); return 0;
}
```

- **Correctness Proof**

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1:n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1:q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n - 1$ without damaging the correctness of the algorithm. □

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, $list[0] \leq list[1] \leq \cdots \leq list[n-1]$. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that $list[i] = searchnum$. If *searchnum* is not present, we should return -1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, $left = 0$ and $right = n-1$. Let $middle = (left+right)/2$ be the middle position in the list. If we compare $list[middle]$ with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle]**. In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle]**. In this case, we return *middle*.

(3) **searchnum > list[middle]**. In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and $n - 1$. So, we set *left* to *middle* + 1.

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to $list[middle]$.

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
           return middle;
        else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
  int  middle;
  while (left <= right)  {
     middle = (left + right)/2;
     switch (COMPARE(list[middle], searchnum)) {
        case -1: left = middle + 1;
                 break;
        case 0 : return middle;
        case 1 : right = middle - 1;
     }
  }
  return -1;
}
```

Program 1.7: Searching an ordered list

Recursive Algorithm

- An algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive.

- to be indirect recursive if it calls another algorithm which in turn calls A.

- These recursive mechanisms are extremely powerful, but even more importantly; manytimes they can express an otherwise complex process very clearly.

```
nt binsearch(int list[], int searchnum, int left,
                                        int right)
/* search list[0] <= list[1] <= ... <= list[n-1] for
   searchnum. Return its position if found. Otherwise
   return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
```

Program 1.8: Recursive implementation of binary search

PERFORMANCE ANALYSIS

An algorithm is said to be efficient and fast, if it takes **less time to execute** & **consume lessmemory space**

Performance is analyzed based on 2 criteria

1. Space Complexity
2. Time Complexity

1. Space Complexity

- Analysis of **space complexity of an algorithm** or program is the **amount of memory itneeds to run to completion**.
- The space needed by a program consists of following components.
- **Fixed space requirements:** Independent on the number and size of the programs input and output. It include

➢ Instruction Space (Space needed to store the code)

- Space for simple variable
- Space for constants

- **Variable space requirements:** This component consists of
  - Space needed by structured variable whose size depends on the particularinstance *I* of the problem being solved
  - Space required when a function uses recursion

- Total Space Complexity **S(P)** of a program is

S(P) = C + Sp(*I*)

Here **Sp(*I*)** is Variable space requirements of program P working on an instance *I*.

C is a constant representing the fixed space requirements

- Example :

1. int sum(int A[], int n)

{

int sum=0, i; for(i=0;i<n;i++)

{

Sum=sum+A[i];return sum;

}

}

Here Space needed for variable n = 1 byteSum = 1 byte

i = 1 byte

Array A[i] = n byte

Total Space complexity = [n+3] byte

2. void main()

{

```
int x,y,z,sum; printf(“Enter 3 numbers”);

scanf(“%d%d%d”,&x,&y,&z);sum = x+y+z;
```

printf(“The sum = %d”,sum);

}

Here Space needed for variable x = 1 bytey = 1 byte

z = 1 byte sum = 1 byte

Total Space complexity = 4 byte

3. sum (a,n)

{

int s=0; for(i=0;i<n;i++)

for(j=0;j<m;j++)

s=s+a[i][j];

return s;

}

Here Space needed for variable n = 1 byte m = 1 byte

s = 1 bytei = 1 bytej = 1 byte

Array a[i][j] = nm byte

Total Space complexity = nm+5 byte

2. Time Complexity

- The **time complexity of an algorithm** or a program is the **amount of time it needs to runto completion**.
- **T(P)=C +TP**

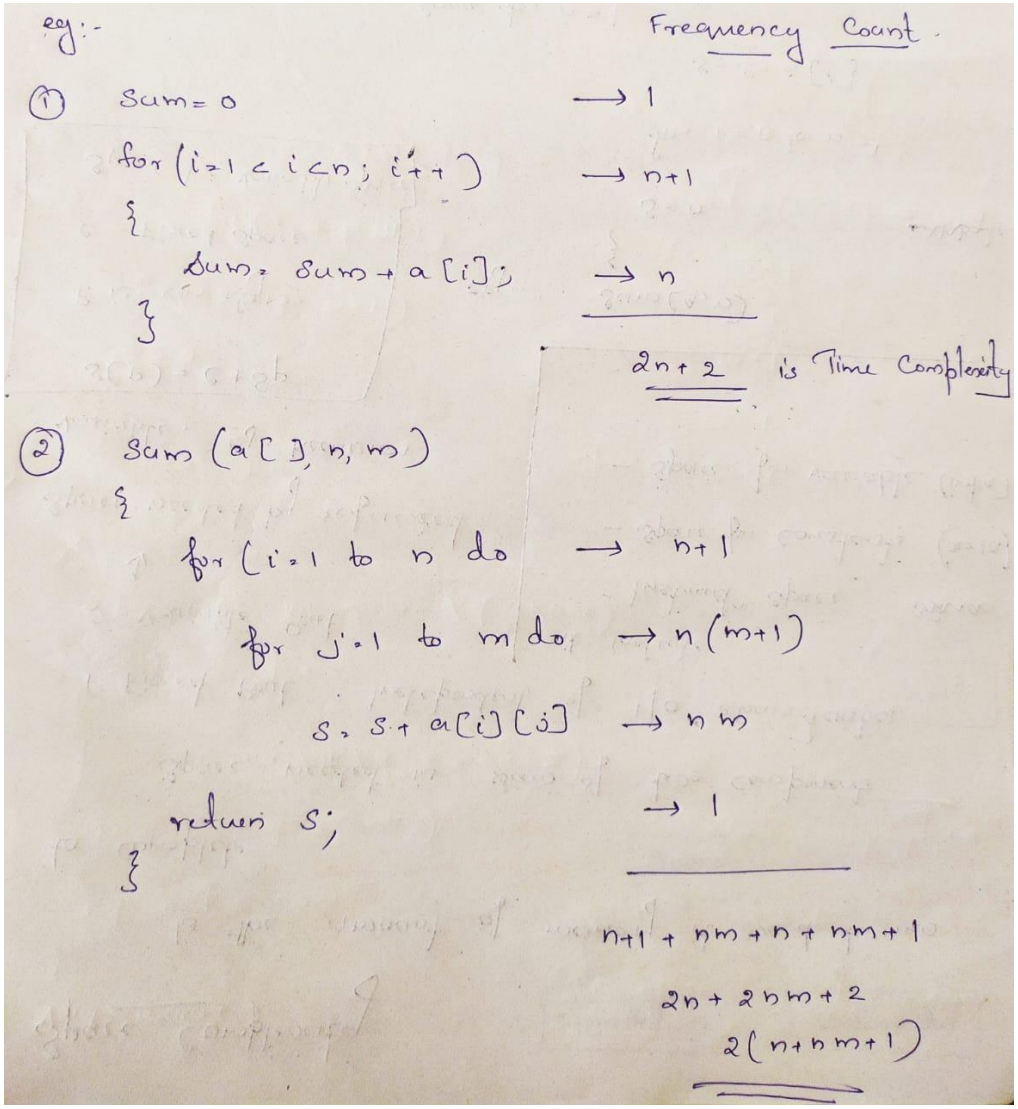
Here **C** is compile time

Tp is Runtime

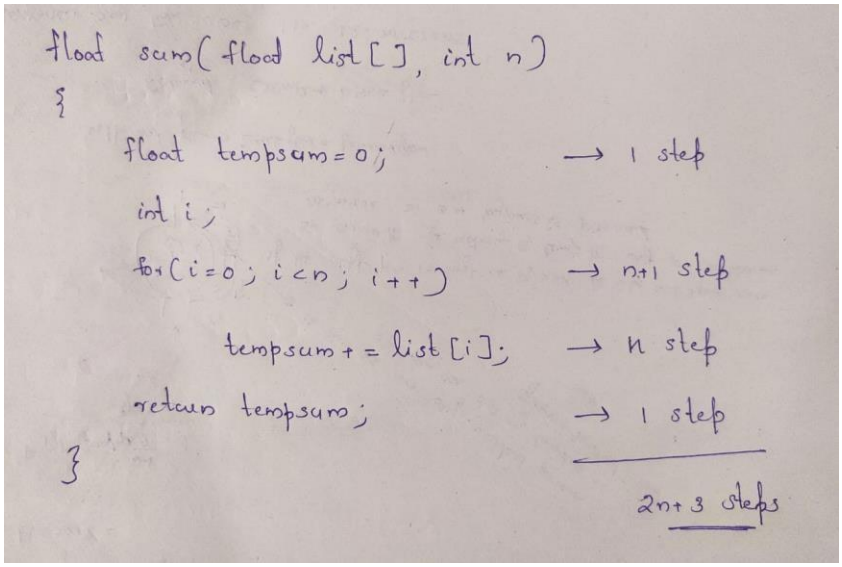
- For calculating the time complexity, we use a method called **Frequency Count** ie, countingthe number of steps

- Comments – 0 step
- Assignment statement – 1 Step
- Conditional statement – 1 Step
- Loop condition for ‘n’ numbers – n+1 Step
- Body of the loop – n step
- Return statement – 1 Step

- Examples:



3. Iterative function for summing a list of numbers

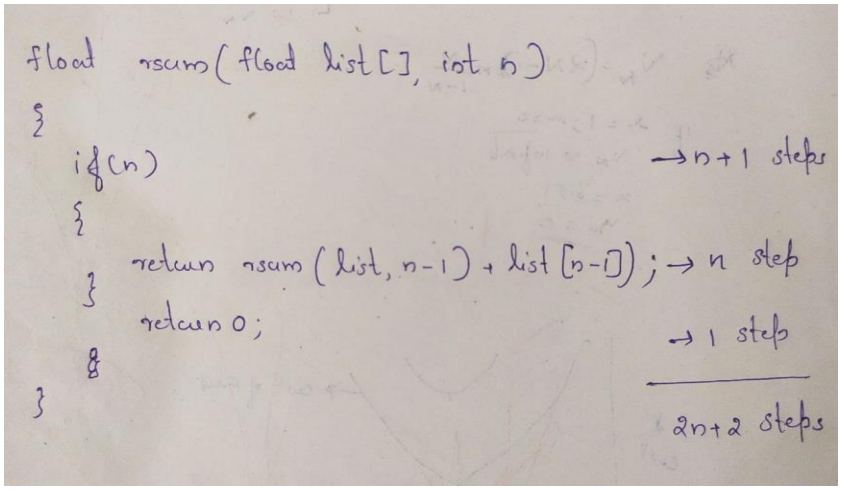


Tabular Method

Statement	s	Frequency	Total steps
float sum(float list[ ],	0	0	0
int n)	0	0	0
{	1	1	1
float tempsum =	0	0	0
0;int i;	1	n+1	n+1n
for(i=0; i <n; i++)	1	n	1
tempsum +=	1	1	0
list[i];	0	0	
return tempsum;			
}			
Total			2n+3

s/e =steps/execution

4. Recursive summing of a list of numbers



Tabular Method

Statement	s	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1n
return rsum(list, n-1)+list[n-1];return list[0];	1	n	1
	1	1	0
}	0	0	
Total			2n+2

- When we analyze an algorithm it depends on the input data, there are three cases :

a. **Best case:** The best case is the minimum number of steps that can be executed for the given parameters.

b. **Average case:** The average case is the average number of steps executed on instances with the given parameters.

c. **Worst case**: In the worst case, is the maximum number of steps that can be executedfor the given parameters

ASYMPTOTIC NOTATION

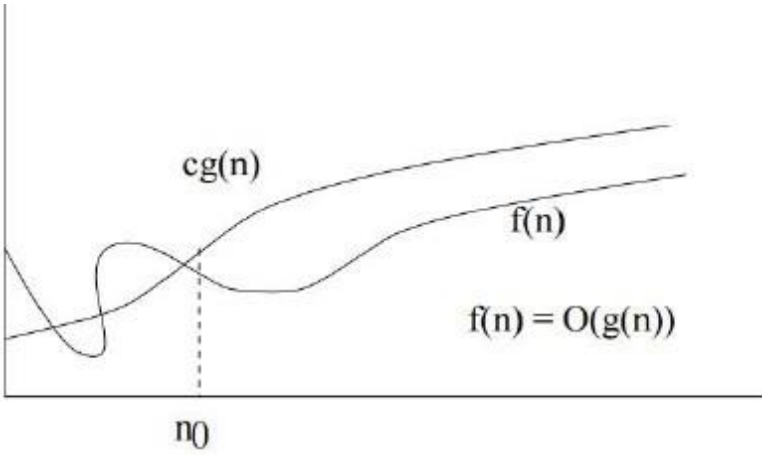
- Complexity of an algorithm is usually a function of n.

- Behavior of this function is usually expressed in terms of one or more standard functions.
- Expressing the complexity function with reference to other known functions is called **asymptotic complexity**.
- Three basic notations are used to express the asymptotic complexity

1. **Big – Oh notation O :** Upper bound of the algorithm
2. **Big – Omega notation Ω :** Lower bound of the algorithm
3. **Big – Theta notation Θ :** Average bound of the algorithm

1. **Big – Oh notation O**

- Formal method of expressing the upper bound of an algorithm's running time.
- i.e. it is a measure of longest amount of time it could possibly take for an algorithm to complete.
- It is used to represent the **worst case** complexity.
- **f(n) = O(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≤ c g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is big-O of g(n)"**.



- Examples:

1. Derive the Big – Oh notation for f(n) = 2n + 3Ans:

2n + 3 <= 2n+3n

2n+3 <= 5n for all n>=1Here c = 5

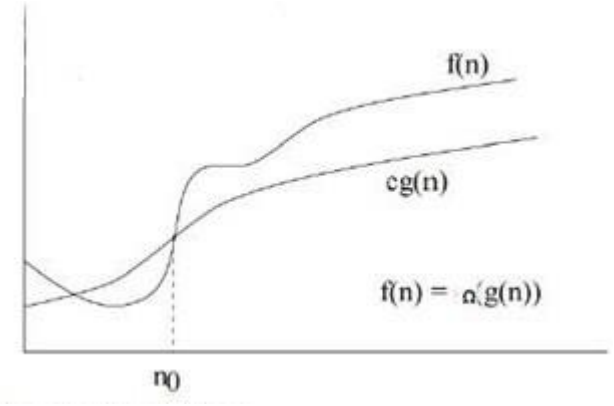
g(n) = nso, f(n) = O(n)

2. **Big – Omega notation Ω**

- **f(n) = Ω(g(n))** if and only if there are two positive constants **c** and **n0** such that

f(n) ≥ c g(n) for all **n ≥ n0.**

- Then we say that **"f(n) is omega of g(n)"**.



- Examples:

Derive the Big – Omega notation for f(n) = 2n + 3Ans:

2n + 3 >= 1n for all n>=1Here c = 1

g(n) = n so, f(n) = Ω (n)

3. Big – Theta notation Θ

- **f(n) = Θ(g(n))** if and only if there are three positive constants c1, c2 and n0 such that **c1 g(n) ≤ f(n) ≤ c2 g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is theta of g(n)"**.
- Examples:

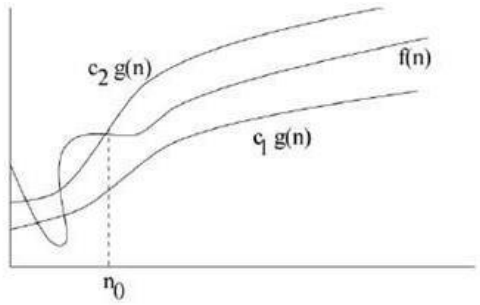
Derive the Big – Theta notation for f(n) = 2n + 3Ans:

1n <= 2n + 3 <= 5n for all

n>=1Here c1 = 1

C2 = 5

g1(n) and g2(n) = nso, f(n) = **Θ** (n)



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

  $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

  Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, c1 = 1, and c2 = 13.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)

Examples

1. $f(n) = 2n^2 + 3n + 4$

$\Rightarrow 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$2n^2 + 3n + 4 \leq 9n^2 \quad n \geq 1$

$\downarrow \quad \downarrow$

$c \quad g(n)$

$f(n) = O(g(n))$

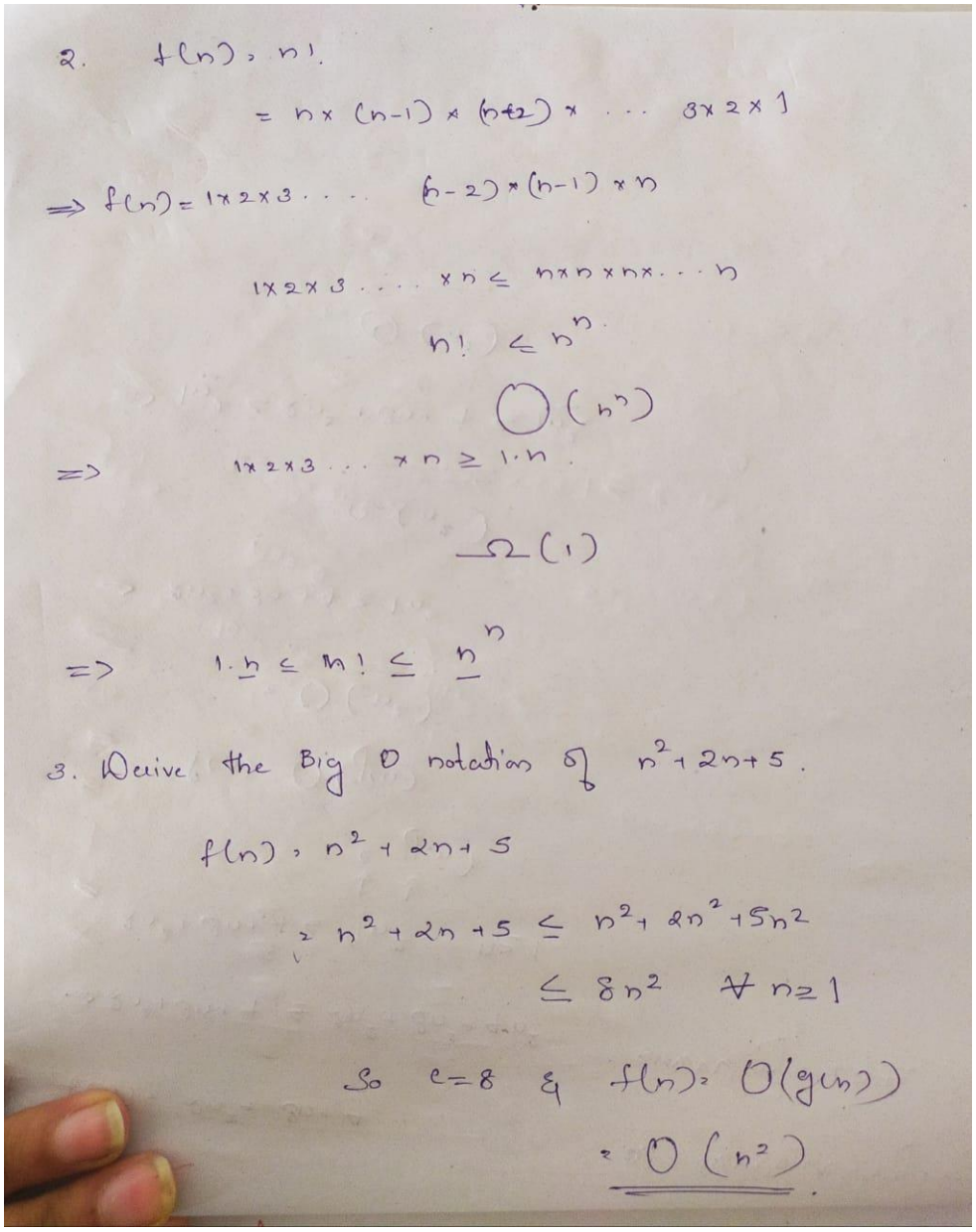
$= O(n^2)$

$\Rightarrow 2n^2 + 3n + 4 \geq 1n^2$

$\Omega(n^2)$

$\Rightarrow 1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

$\Theta(n^2)$



TIME COMPLEXITY OF LINEAR SEARCH

- Any algorithm is analyzed based on the unit of computation it performs. For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Best Case	Worst Case	Average Case
1	n	n / 2

TIME COMPLEXITY OF BINARY SEARCH

- In Binary search algorithm, the target key is examined in a sorted sequence and thisalgorithm starts searching with the middle item of the sorted sequence.

a. If the middle item is the target value, then the search item is found and it returns True.

b. If the target **item < middle** item, then search for the target value in the first halfof the list.

c. If the target **item > middle** item, then search for the target value in the second halfof the list.

- In binary search as the list is ordered, so we can eliminate half of the values in the list in each iteration.
- Consider an example, suppose we want to search 10 in a sorted array of elements, then we first determine 15 the middle element of the array. As the middle item contains 18, which is greater than the target value 10, so can discard the second half of the list and repeat the process to first half of the array. This process is repeated until the desired target item is located in the list. If the item is found then it returns True, otherwise False.
- In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about ***n/2*** items will be eliminated after first comparison. After second comparison, ***n/4*** items of the list will be eliminated. If this process is repeated for several times, then there will be just one item left in the list. Thenumber of comparisons required to reach to this point is $n/2^i = 1$. If we solve for ***i***, then it gives us $i = log2\ n$. The maximum number is comparison is logarithmic in nature, hence the time complexity of binary search is ***O(log n).***

Best Case	Worst Case	Average Case
1	O(log n)	O(log n)

MODULE 2 - ARRAYS AND SEARCHING

Polynomial representation using Arrays, Sparse matrix, Stacks, Queues - Circular Queues, PriorityQueues, Double Ended Queues, Evaluation of Expressions, Linear Search and Binary Search

DATA STRUCTURE

It is a representation of logical relationship between individual elements of data. It is also defined as a mathematical model of particular organization of data items. It is also called building block of a program.

Classification of data structure

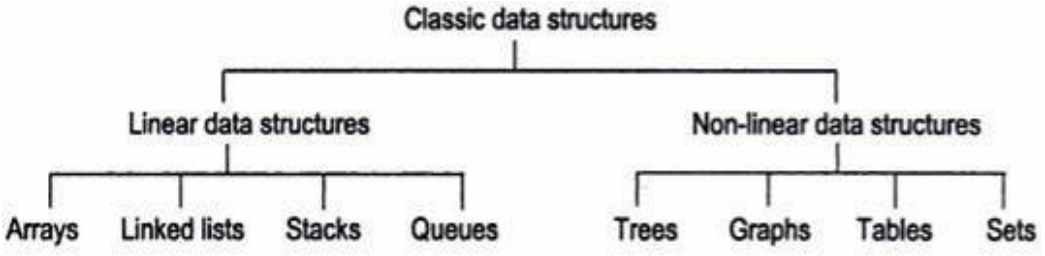
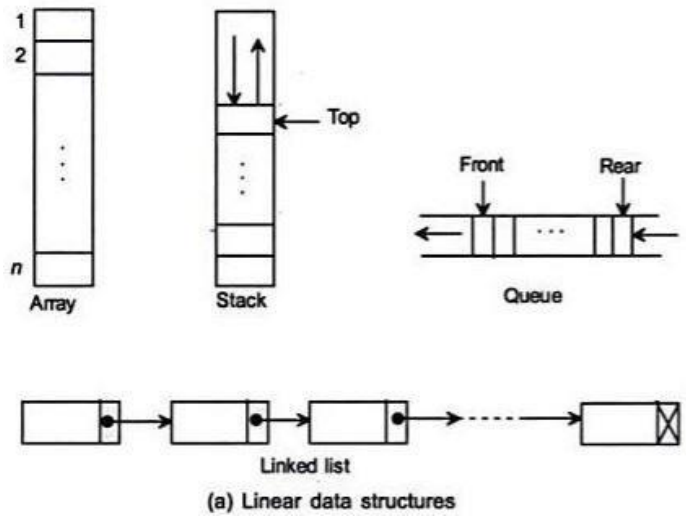


Fig. 1.2 Classification of classic data structures.

1. Linear data structure

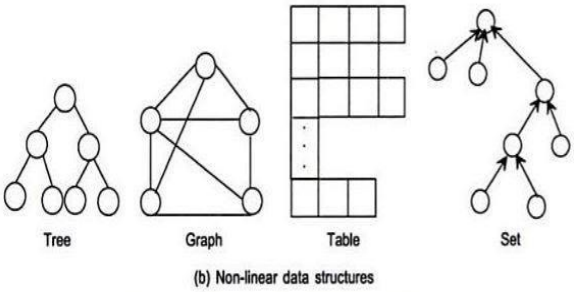
- All the elements form a sequence or maintain a linear ordering.



(a) Linear data structures

2. Non linear data structure

- Elements are distributed over a plane.



(b) Non-linear data structures

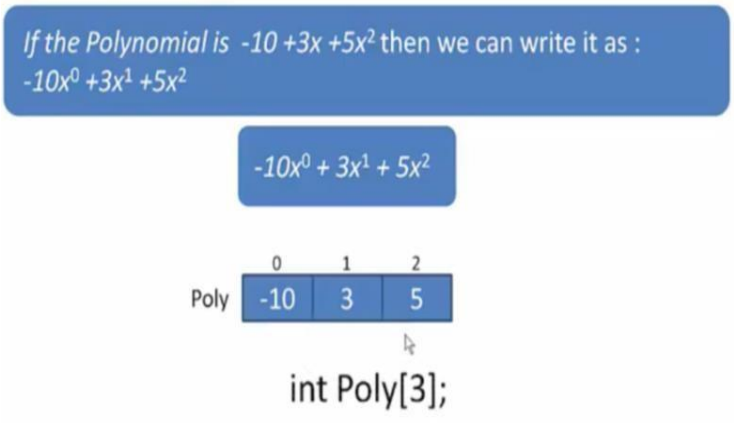
1. POLYNOMIAL REPRESENTATION USING ARRAYS

- A polynomial is a sum of terms where each term has the form $\mathbf{ax^e}$,Where x is the variable, a is the coefficient and e is the exponent.

A general polynomial $A(x)$ can be written as

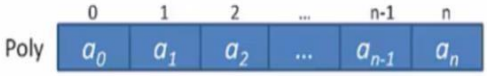
$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where $a_n \neq 0$ and we say that the degree of A is n.

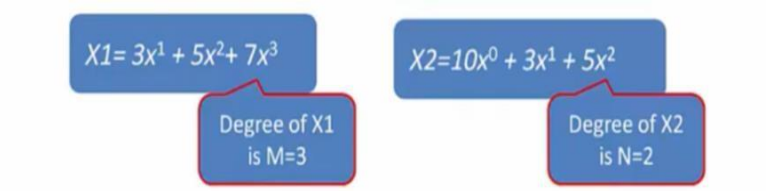


Polynomial representation using Arrays

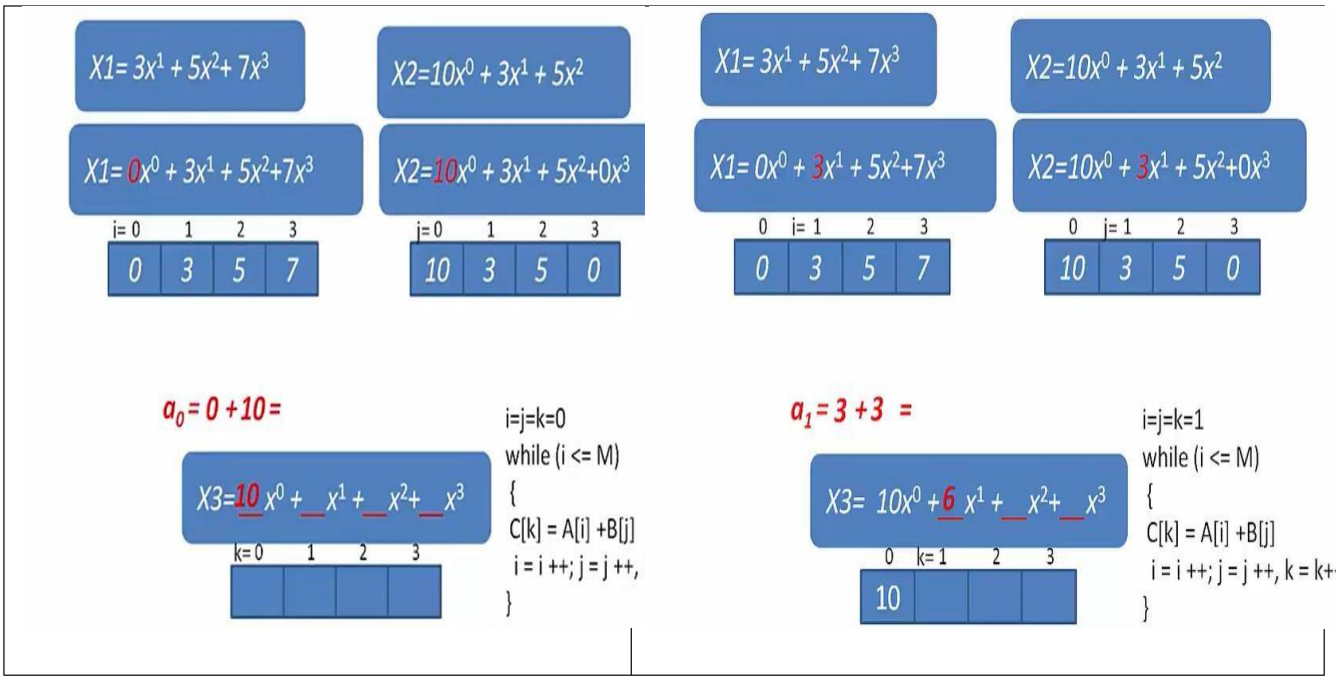
A polynomial of a single variable $A(x)$ can be written as $a_0 + a_1X + a_2X^2 + \ldots + a_nX^n$ where $a_n \neq 0$ and degree of $A(X)$ is n.



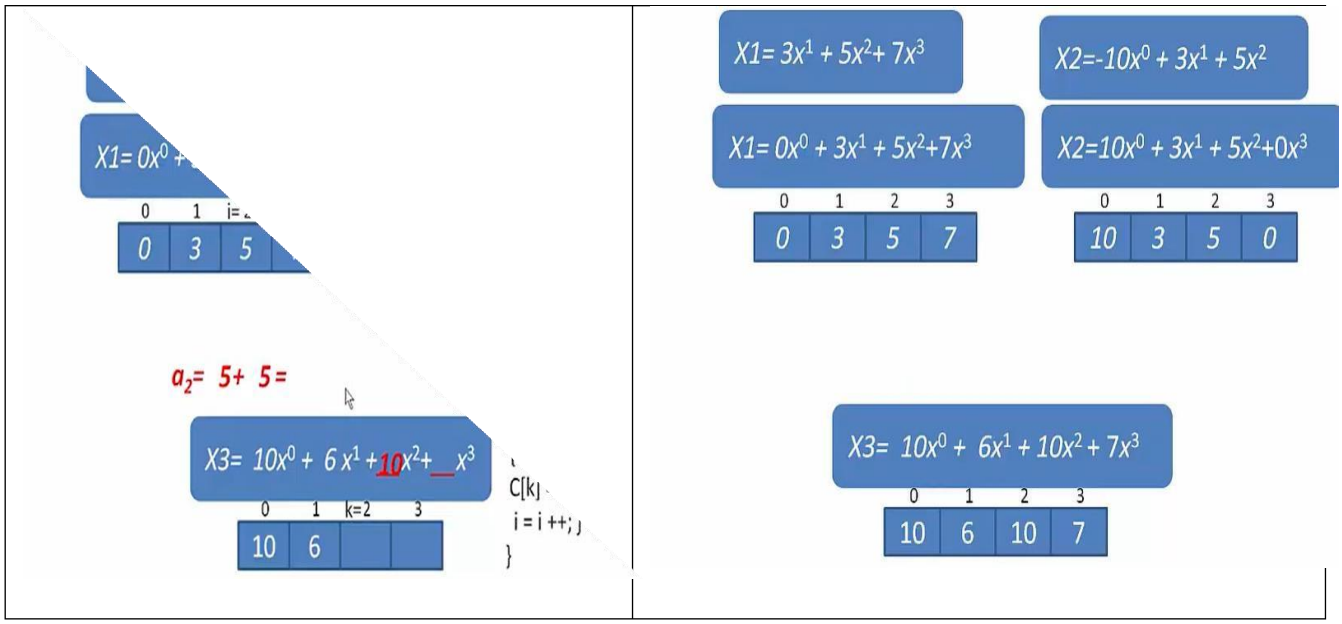
For a polynomial of degree n, n+1 terms are required



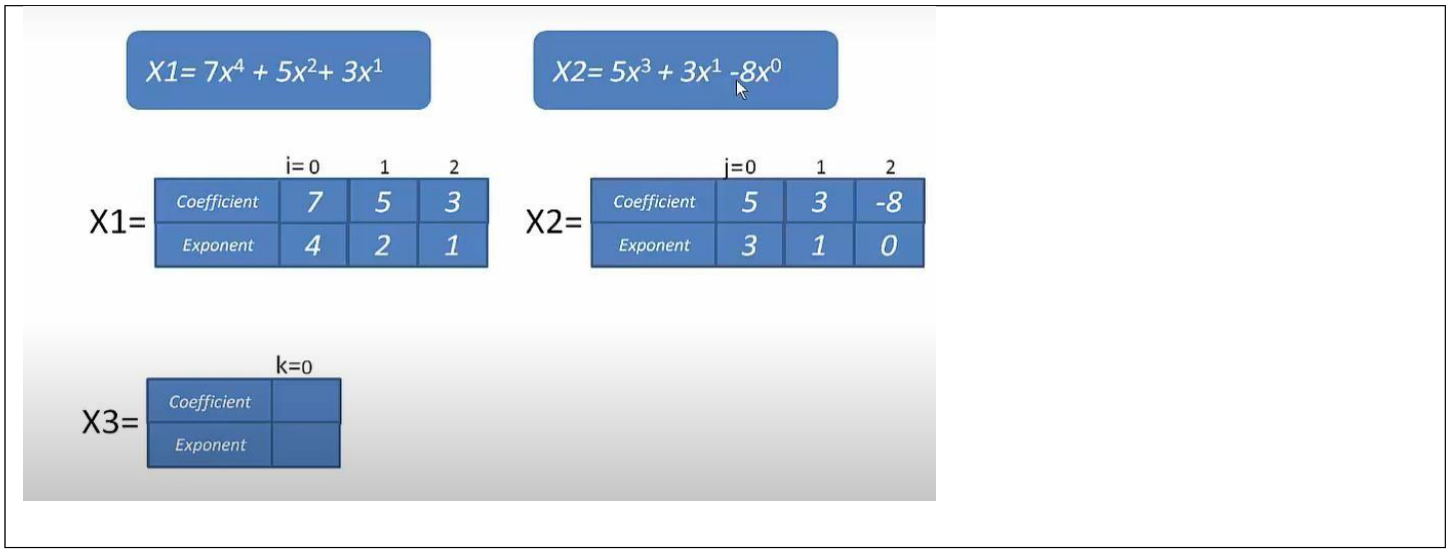
- Identify the value of Highest degree polynomial.
- Write polynomial X3 with degree Max(degree of X1 and degree of X2).



Polynomial Addition Example

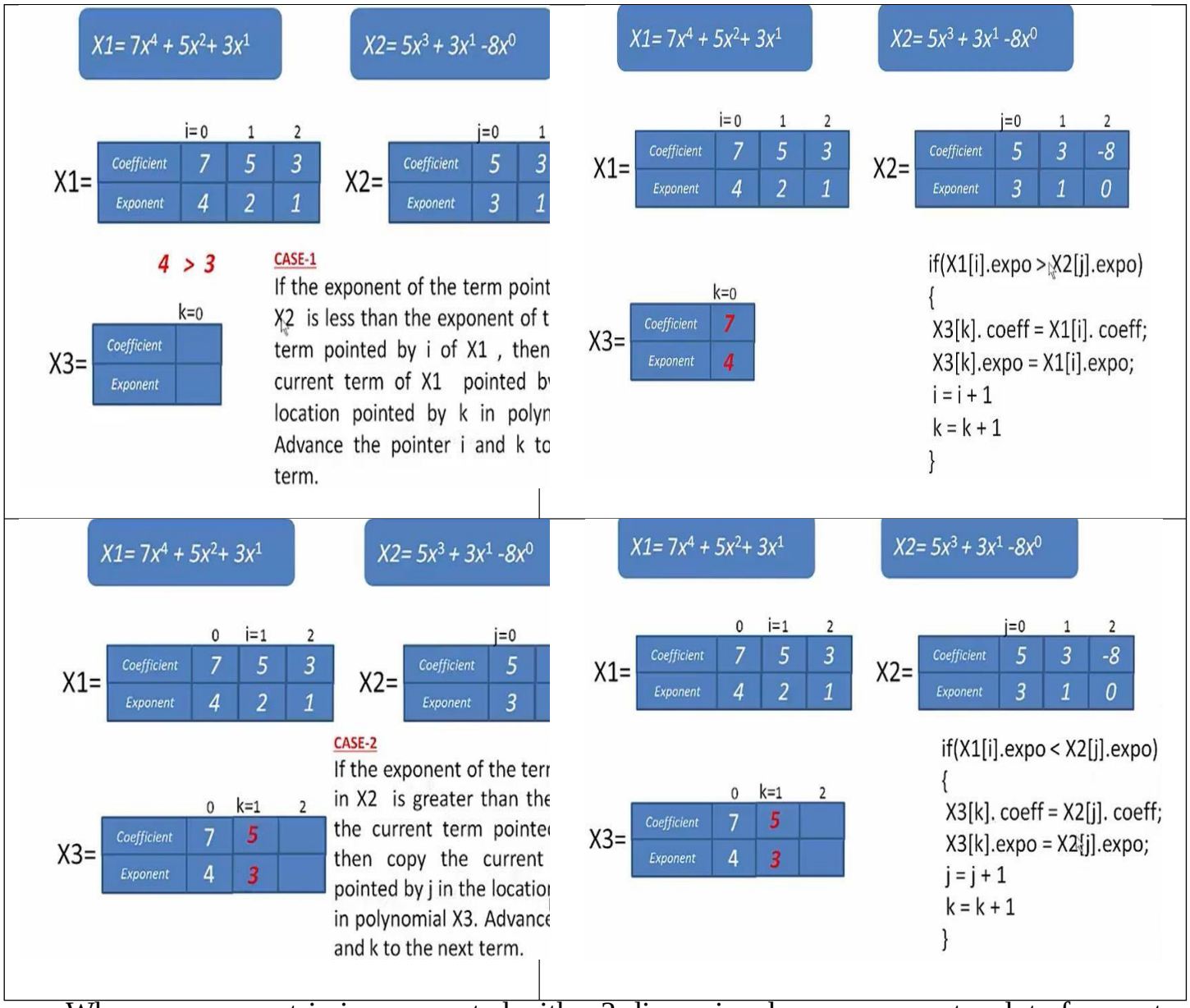


Steps of Polynomial Addition



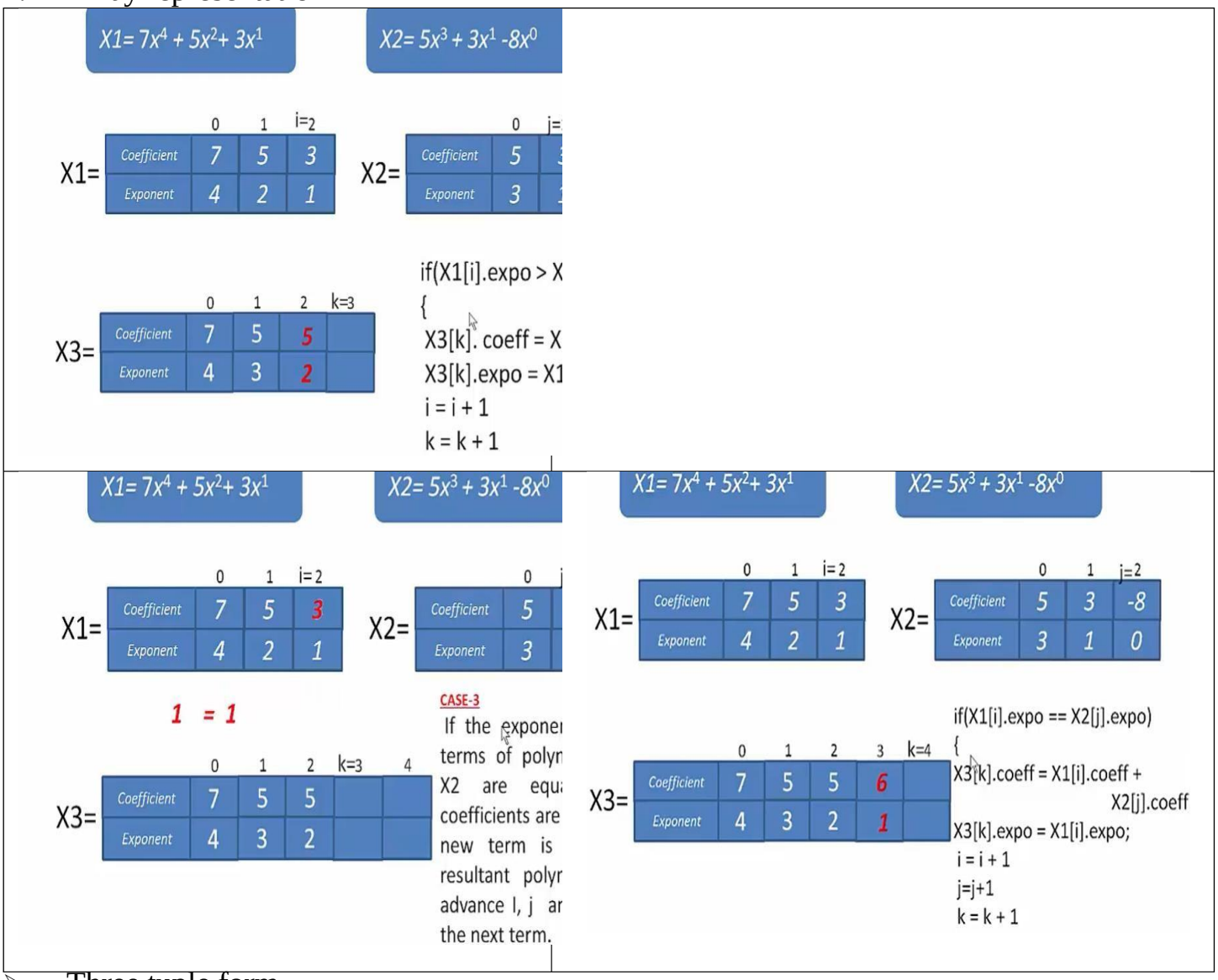
2. SPARSE MATRIX

- A matrix is a two-dimensional data object made of ‘m’ rows and ‘n’ columns, therefore having total m x n values. If most of the elements of the matrix have 0 values, then it is called a **sparse matrix**.
- **Sparse matrix is a matrix which contains very few non-zero elements.**



- When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix.
- Consider a matrix of size **100 X 100** containing only **10 non-zero** elements. In this matrix, **only 10 spaces are filled** with non-zero values and remaining spaces of the matrix are filledwith zero. Totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. To access these 10 non-zero elements we have to make scanning for 10000 times.
- Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation



- Three tuple form

2. Linked list representation

- 2D array is used to represent a sparse matrix in which there are three columns named as
- **Row:**Index of row, where non-zero element is located
- **Column:**Index of column, where non-zero element is located
- **Value:**Value of the non zero element located at index –(row,column)

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0
Coefficient	5
Exponent	3

No more element in i

CASE-3

If there is no r
in X1 and th
elements remain
copy rest of the
to X3 and advar
track to the next

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	-8
Exponent	4	3	2	1	0

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	
Exponent	4	3	2	1	

```
while (j < n) do
{
 X3[k].coeff = X2[j].coeff;
 X3[k].expo = X2[j].expo;
 j = j + 1
 k = k + 1
}
```

X1=

Coefficient	
Exponent	4

X3=

	0	1	2	3	k
Coefficient	7	5	5	6	
Exponent	4	3	2	1	

k =
}

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

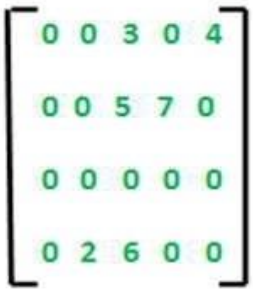
	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	-8
Exponent	4	3	2	1	0



Row	Column	Value
0	2	3
0	4	4
1	2	5
1	3	7
3	1	2
3	2	6

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Triplets

(0,2,3)

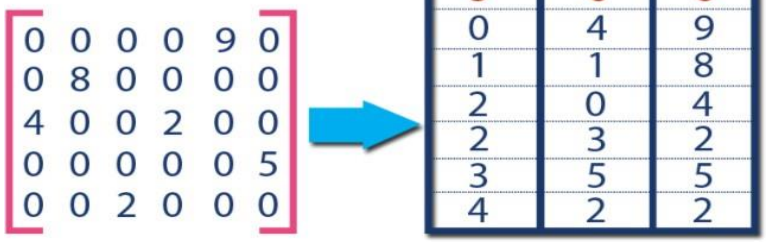
(0,4,4)

(1,2,5)

(1,3,7)

(3,1,2)

(3,2,6)



Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can beused to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data

structuretraversing only non-zero elements.

3. STACK

- It is a linear data structure in which elements are placed one above another.
- A stack is an ordered collection of homogeneous data elements where the insertion anddeletion operations take place only at one end called **Top** of the stack.
- **LIFO -** In stack elements are arranged in **Last-In-First-Out** manner (**LIFO**). So it is alsocalled LIFO lists.
- Anything added to the stack goes on the "**top**" of the stack.
- Anything removed from the stack is taken from the "**top**" of the stack.
- Things are removed in the reverse order from that in which they were inserted

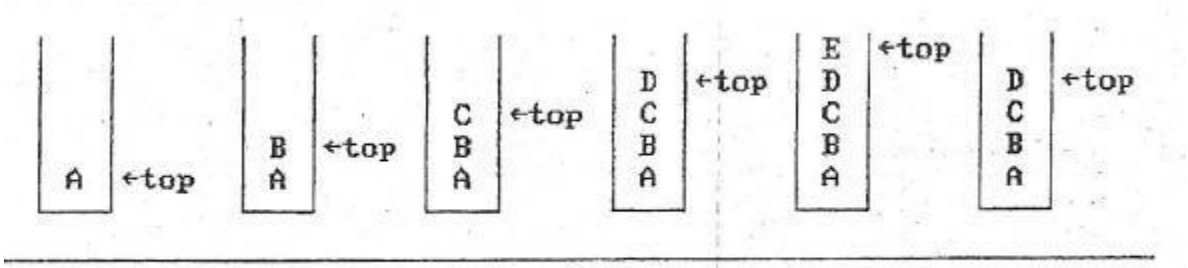
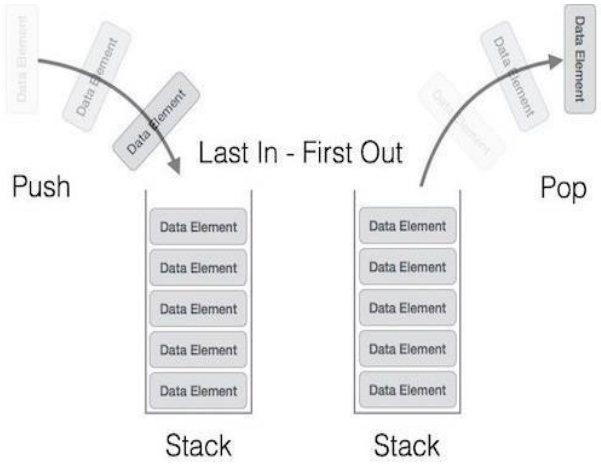


Figure 3.1: Inserting and deleting elements in a stack

Operations of Stack

- Two basic operations of stack:
  - **PUSH :** Insert an element at the top of stack
  - **POP:** Delete an element from the top of stack



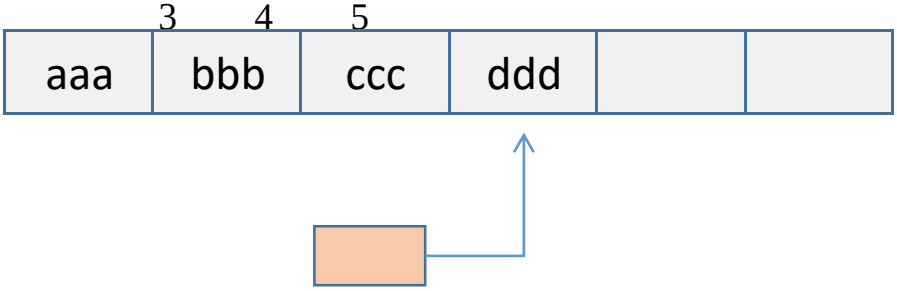
- An element in the stack is termed as **ITEM.**

- Initially top is set to -1, to indicate an **empty stack**. **(Top = -1)**
- The maximum no. of elements that a stack can accommodate is termed **MAX_SIZE**.
- If stack is full **Top = MAX_SIZE - 1**

Array representation of stack

- Stack can be represented using a linear array.
- There is a pointer called TOP to indicate the top of the stack

0 12 3 4 5



top

- **Overflow:** If we try to insert a new element in the stack top (push) which is already **full**,then the situation is called stack overflow.
- **Underflow:** If we try to delete an element (pop) from an **empty** stack, the situation is calledstack underflow.

Basic Operations

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.
- **peek()** − get the top data element of the stack, without removing it.

```
int peek() {
return stack[top];
}
```

- **isFull()** − check if stack is full.bool

```
isfull() {
if (top == MAX_SIZE)
return true;
```

else

return false;

}

- **isEmpty()** − check if stack is empty.bool

isempty() {

if(top == -1)

return true;

else

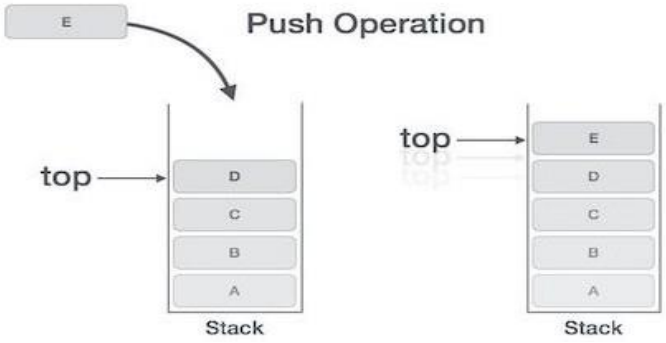
return false;

}

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



Algorithm: PUSH()

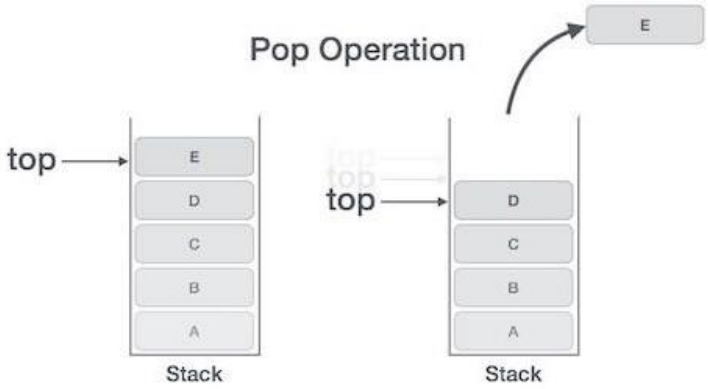
- Let A be an array with Maximum size as MAX_SIZE. Initially, top= -1

POP Oper

1. Start
2. if top < MAX_SIZE – 1
3. set top=top+1
4. Set A[top]=item
5. else
6. print “OVERFLOW”
7. exit

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm: POP()

1. Start
2. if top= -1 then
3. print “UNDERFLOW”
4. else
5. set item=A[top]
6. Set top=top-1
7. exit

Applications of stack

- Reversing an array
  - A B C D
  - Pushing to stack A B C D

- Popping from stack D C B A
- Undo operations
- Infix to prefix, infix to postfix conversion
- Tree Traversal
- Evaluation of postfix expressions

4. QUEUES

- A queue is an ordered collection of homogeneous data elements. In which insertion is doneat one end called **REAR** and deletion is done at another end called **FRONT**.
- **FIFO -** In queue elements are arranged in **First-In-First-Out** manner (**FIFO**).
- First inserted element is removed first

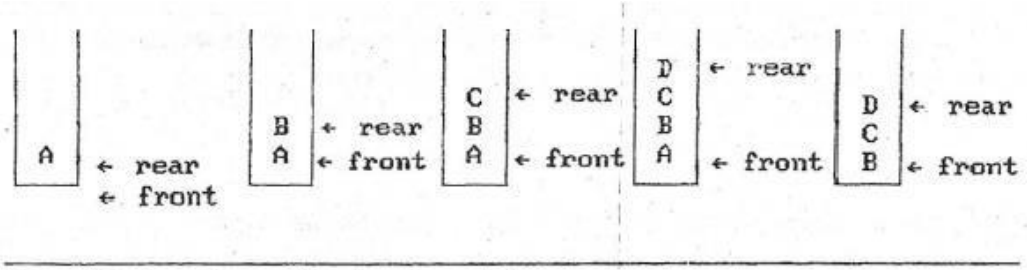
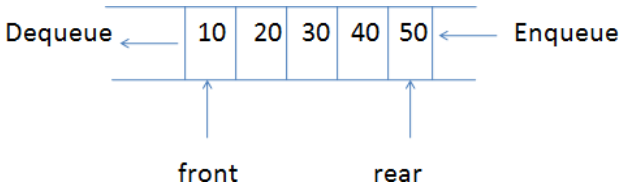


Figure 3.4: Inserting and deleting elements in a queue

- Two basic operations of queue:

1. **Enqueue ->** Insert an element at the **rear** end of queue.
2. **Dequeue->** Delete an element from the **front** end of queue.



- Initial case **rear = -1** and **front = 0**, **MAX SIZE** is the size of the queue.
- If **rear = front** then queue contains only a **single element**
- If **rear < front** then queue is **empty**
- **Queue full :** rear = n-1 and front =0
- Whenever an element is deleted from the queue, the value of FRONT is increased by 1.

- i.e. FRONT=FRONT+1
- Similarly, whenever an element is added to the queue, the REAR is incremented by 1 as,
- REAR=REAR+1

Array Representation of Queue

A one-dimensional array, say $Q[1 \ldots N]$, can be used to represent a queue. Figure 5.3 shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

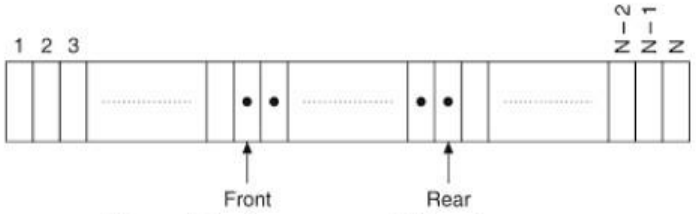


Figure 5.3 Array representation of a queue.

Basic Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **peek()** − Gets the element at the front of the queue without removing it.

```
int peek()
{
return queue[front];
}
```

- **isfull()** − Checks if the queue is full

```
bool isfull()
{
If (rear == MAXSIZE - 1)
                    return true;
```

```
                else
                        return false;
        }
```

- **isempty()** − Checks if the queue is empty.bool

```
isempty()
{

if(front < 0 || front > rear)return true;
```

else

}

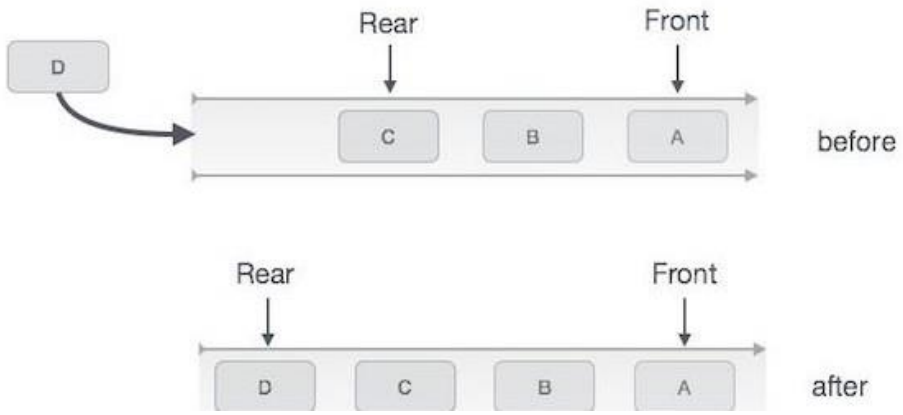
return false;

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



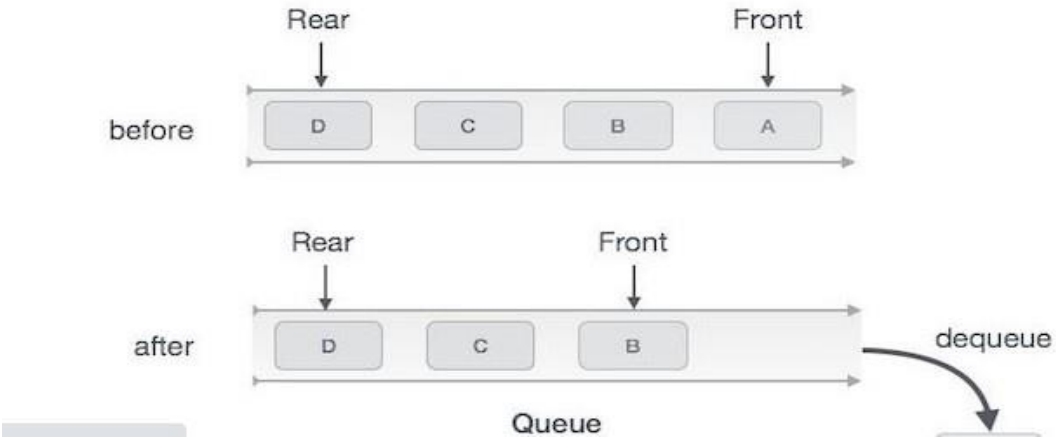
Algorithm : Enqueue

1. Start
2. if rear = MAX_SIZE – 1 then
3. print “OVERFLOW”
4. else
5. set rear = rear + 1
6. Set A[rear]=item
7. exit

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Algorithm : Dequeue

```
1. Start
2. if rear < front then
3.         print “UNDER FLOW”
4. else
5.         set  item = A[front}
6.         set front = front + 1
7. exit
```

Type of Queues

- Circular Queue
- Priority Queue
- Doubly ended Queue

5. CIRCULAR QUEUE

- To utilize space properly, circular queue is derived.
- In this queue the elements are inserted in circular manner.
- So that no space is wasted at all.
- Circular queue empty:

FRONT= -1

REAR= -1

- Circular queue full:

(rear + 1) % max_size = Front

- It is a modification of simple queue in which the rear pointer is set to the initial location,whenever it reaches the location max_size – 1.

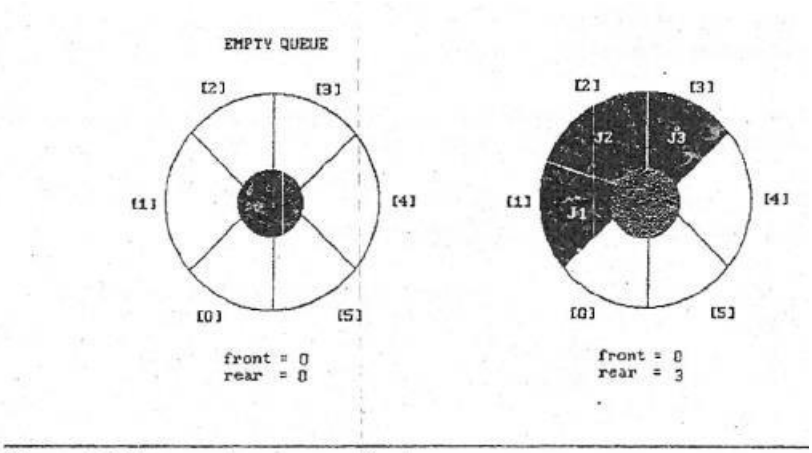


Figure 3.6: Empty and nonempty circular queues

Insertion Algorithm (ENQUEUE)

```
1. if (front == -1 & rear == -1)
2.         set front =0 and rear = 0
3.         Set a[rear]=item
4. else if (front = (rear+1) % max_size) then
5.          Print over flow
6. else
7.         set rear = (rear + 1)% max_size
8.          Set a[rear] = item
9. Exit
```

Deletion Algorithm (DEQUEUE)

```
1. if front = -1 and rear = -1 then
2.         print underflow and exit
3. else if front = rear
4.          set item= a[front]
5.          set front = -1 and rear = -1
6. else
7.         set item= a[front]
8.         set front = (front + 1) % max_size
9. Exit
```

7. PRIORITY QUEUE

- Regular queue follows a First In First Out (FIFO) order to insert and remove an item.Whatever goes in first, comes out first.
- In a priority queue, an item with the highest priority comes out first.

- Therefore, the FIFO pattern is no longer valid.
- Every item in the priority queue is associated with a priority.
- It does not matter in which order we insert the items in the queue
- The item with higher priority must be removed before the item with the lower priority.
- If two elements have the same priority, they are served according to their order in the queue.

Operations on a priority queue

1. **EnQueue**: EnQueue operation inserts an item into the queue. The item can be inserted at the end of the queue or at the front of the queue or at the middle. The item must havea priority.
2. **DeQueue**: DeQueue operation removes the item with the highest priority from the queue.
3. **Peek**: Peek operation reads the item with the highest priority.

1. Enqueue Operation

1. IF((Front == 0)&&(Rear == N-1))
2. PRINT “Overflow Condition”
3. Else IF(Front == -1& rear == -1)
4. Front = Rear =0
5. Queue[Rear] = Data
6. Priority[Rear] = Priority
7. ELSE IF(Rear ==N-1)
8. FOR (i=Front;i<=Rear;i++)
9. FOR(i=Front;i<=Rear;i++)
10. Q[i-Front] =Q[i]
11. Pr[i-Front] = Pr[i]
12. Rear = Rear-Front
13. Front = 0
14. FOR(i = r;i>f;i–)
15. IF(p>Pr[i])
16. Q[i+1] = Q[i] Pr[i+1] = Pr[i]
17. ELSE
18. Q[i+1] = data Pr[i+1] = p
19. Rear++.

2. Dequeue operation

1. IF(Front == -1)
2. PRINT “Queue Under flow condition”
3. ELSE
4. PRINT”Q[f],Pr[f]”
5. IF(Front==Rear)

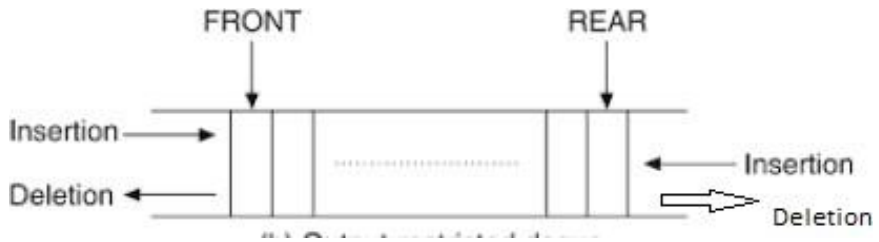
6. Front = Rear = -1
7. ELSE
8. FRONT++

Applications of Priority Queue

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum SpanningTree, etc
3. All queue applications where priority is involved.
4. For load balancing and interrupt handling in an operating system

8. DOUBLY ENDED QUEUE

It is a list of elements in which insertion and deletion are perform at both ends



- It has 4 operations

1. Insertion at rear end
2. Insertion at front end
3. Deletion at rear end
4. Deletion at front end

1. Algorithm : Insertion at rear end

```
1. Start
2. if rear = MAX_SIZE – 1 then
3.       print “OVERFLOW”
4. Else
5.       set rear = rear + 1
6.        Set A[rear]=item
7. exit
```

2. Insertion at front end

```
1. Start
2. if front = 0 then
3.       print “OVERFLOW” and exit
4. Else
5.       set front = front - 1
6.       Set A[front]=item
7. exit
```

3. Deletion at front end

```
1. Start
2. if front = 0 and rear = -1 then
3.       print “UNDER FLOW” and exit
4. set item = A[front]
5. if front = rear then
6.       set front = 0 and rear = -1
7. Else set front = front + 1
8. exit
```

4. Deletion at rear end

1. Start
2. if front = 0 and rear = -1 then
3. print “UNDER FLOW” and exit
4. set item = A[rear]
5. if front = rear then
6. set front = 0 and rear = -1
7. Else set rear = rear - 1
8. exit

9. CONVERSION & EVALUATION OF EXPRESSIONS

- **Infix Expression:** The operator occurs between the operands

<operand> <operator> <operand>

Eg: a+b

- **Prefix Expression (Polish notation):** The operators occurs before the operand

<operator> <operand> <operand>

Eg : +ab

- **Postfix Expression (Reverse Polish notation):** The operators occurs after the operand

<operand> <operand> <operator>

Eg : ab+

Token	Operator	Precedence[1]	Associativity
() [] -> . .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

Infix to prefix

? a + b * c

Ans:- a + b * c [Two operands + & * consider the highest one]
(* is highest)

a + * b c

+ a * b c

? A + (B * C - D / E) - F

Ans:- A + (B * C - D / E) - F

[Consider the expression inside bracket]

A + (* B C - / D E) - F

A + - * B C / D E - F

+ A - * B C / D E - F

- + A - * B C / D E F

Infix to postfix

? a + b * c

Ans:- a + b * c

a + b c *

a b c * +

? A + (B * C - D / E) - F

Ans:- A + (B * C - D / E) - F

A + (B C * - D E /) - F

A + (B C * D E / -) - F

A B C * D E / - + - F

A B C * D E / - + F -

Convert the following expression into postfix.

1. (A + (B * C - (D / E ↑ F) * G) * H)

First consider the inner bracket, that contain / & ↑ operands. ↑ has the highest degree.

(A + (B * C - (D / E ↑ F) * G) * H)

(A + (B * C - (D / EF ↑) * G) * H)

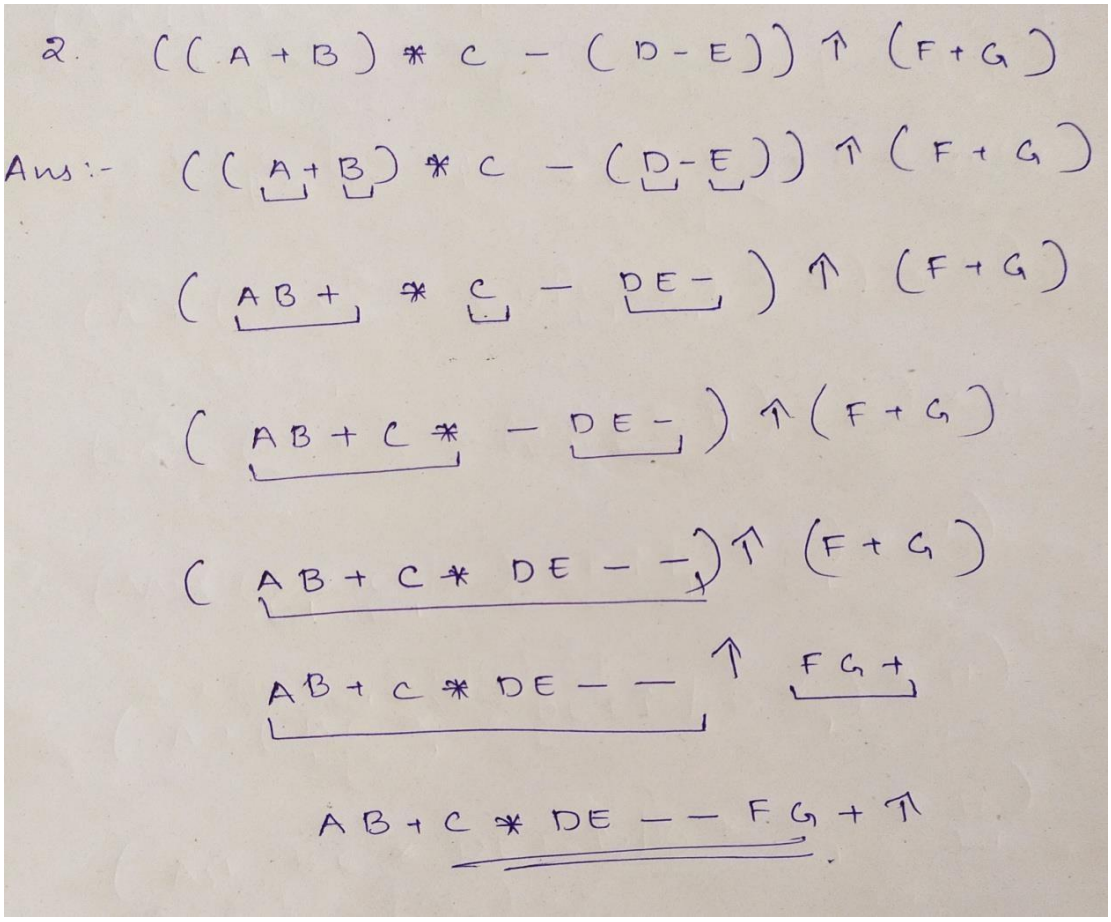
(A + (B * C - DEF ↑ / * G) * H)

(A + (BC * - DEF ↑ / G *) * H)

(A + BC * DEF ↑ / G * - * H)

(A + BC * DEF ↑ / G * - H *)

ABC * DEF ↑ / G * - H * +



A. Postfix Expression Evaluation

Given P is the postfix expression, the following algorithm uses a stack to hold operands. It finds the value of the arithmetic expression P, Written in postfix notation.

Algorithm:

Step 1: Add **") "** at the end of P

Step 2: Scan P from left – right & repeat the steps 3 & 4

Step 3: If an operand occurs, PUSH it to stack.

Step 4: If an operator ⊗ occurs, then

A: Remove the top elements of the stack.

When A is the top element and B is the next top element B: Evaluate

B ⊗ A

C: Place the result of step B back to stack

Step 5: Set the value equals to TOP element of the stack.

1. **Evaluate the expression 5 * (6 + 2) – 12 / 4Ans**

: Convert to postfix notation

5 * 6 2 + - 12 / 4

5 6 2 + * - 12 4 /

= 5 6 2 + * 12 4 / -

Add **“) “** at the end of P

P = 5 6 2 + * 12 4 / -)

Scanned Symbol	**Stack**
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

2. Evaluate the expression (6 +2) / (4 – 2 * 1)

Ans: Convert to postfix notation

6 2 + / (4 – 2 1 *)

6 2 + / 4 2 1 * -

6 2 + 4 2 1 * - /

P = 6 2 + 4 2 1 * - / **)**

Scanned Symbol	Stack

stack which holds the left parenthesis and operators. We renthesis to stack and adding a right parenthesis at the end of Q.

Algorithm

Step 1: PUSH left parenthesis **"("** into stack and add right parenthesis **") "** at the end of Q.

Step 2: Scan the expression Q from Left – Right and repeat the step 3 to 6 for each element of Q until this stack is empty.

Step 3: If an operand occurs add it to P.

Step 4: If a Left parenthesis occurs then PUSH it to stack

Step 5: If an operator $\otimes$ occurs then

A: Repeatedly POP the stack and add to P, each operator which has same or higher precedence than $\otimes$

B: add $\otimes$ to stack

Step 6: If a Right parenthesis occurs then

A: Repeatedly POP from stack and add to P each operator until a left parenthesisoccurs.

B: Remove the left parenthesis

Step 7: Exit

1. Q = A + (B * C - (D / E ^ F) * G) * H

Ans : Add right parenthesis at the end of the expressionQ = A

+ (B * C - (D / E ^ F) * G) * H **)**

Symbol Scanned	**Stack**	**p**
	(	
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	AB
*	(+ (*	AB
C	(+ (*	ABC
-	(+ (-	ABC*
(	(+ (- (	ABC*
D	(+ (- (	ABC*D
/	(+ (- (/	ABC*D
E	(+ (- (/	ABC*DE

^	(+ (- (/ ^	ABC*DE
F	(+ (- (/ ^	ABC*DEF
)	(+ (-	ABC*DEF ^ /
*	(+ (- *	ABC*DEF ^ /
G	(+ (- *	ABC*DEF ^ /G
)	(+	ABC*DEF ^ /G * -
*	(+ *	ABC*DEF ^ /G * -
H	(+ *	ABC*DEF ^ /G * - H
)		ABC*DEF ^ /G * - H * +

2. Q = ((A + B) * C – (D – E)) ^ (F + G)

Ans:

Q = ((A + B) * C – (D – E)) ^ (F + G) **)**

Symbol Scanned	**Stack**	**p**
0	(	
(	((	
(	(((	
A	(((	A
+	(((+	A

B	(((+	AB
)	((	AB+
*	((*	AB+
C	((*	AB+C
-	((-	AB+C*
(	((- (	AB+C*
D	((- (	AB+C*D
-	((- (-	AB+C*D
E	((- (-	AB+C*DE
)	((-	AB+C*DE-
)	(	AB+C*DE--
^	(^	AB+C*DE--
(	(^ (	AB+C*DE--
F	(^ (	AB+C*DE--F
+	(^ (+	AB+C*DE--F
G	(^ (+	AB+C*DE--FG
)	(^	AB+C*DE--FG+
)		AB+C*DE—FG+^

3. Q = (A + B) * C / D + E ^ F / GAns :

Q = (A + B) * C / D + E ^ F / G)

Symbol Scanned	Stack	p
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
/	(/	AB+C*
D	(/	AB+C*D
+	(+	AB+C*D/
E	(+	AB+C*D/E
^	(+ ^	AB+C*D/E
F	(+ ^	AB+C*D/EF
/	(+ /	AB+C*D/EF^
G	(+ /	AB+C*D/EF^G
)		AB+C*D/EF^G/+

10. LINEAR SEARCH AND BINARY SEARCH

1. **Linear search:** Small & unsorted arrays
2. **Binary search :** Large arrays & sorted arrays

1. Linear Search

- It means looking at each element of the array, in turn, until you find the target value.

Algorithm

```
1. Start
2. Read the ITEM to be searched
3. Set flag=0
5.        if  A[i]= =ITEM
6.                print “item
7.                flag=1
8. If flag= =0
9.        print “item not
```

- In the **best case**, the target value is in the first element of the array. So the search takessome tiny, and constant, amount of time. Computer scientists denote this **O(1)** In reallife, we don’t care about the best case, because it so rarely actually happens.
- In the **worst case**, the target value is in the last element of the array. So the search takes an amount of time proportional to the length of the array. Computer scientists denote this **O(*n*)**
- In the **average case**, the target value is somewhere in the array. So on average, the target value will be in the middle of the array. So the search takes an amount of time proportional to half the length of the array – also proportional to the length of the array

– **O(*n*)** again**Binary Search**

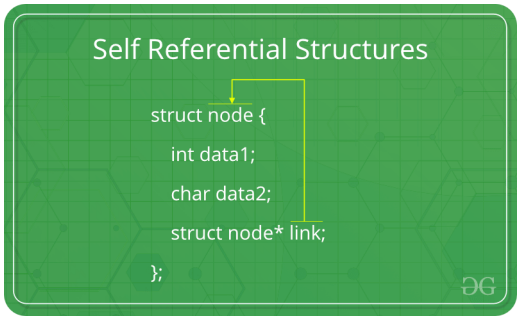
- The general term for a smart search through sorted data is a *binary search*.

1. The initial search region is the whole array.
2. Look at the data value in the middle of the search region.
3. If you've found your target, stop.
4. If your target is less than the middle data value, the new search region is the lower half of the data.
5. If your target is greater than the middle data value, the new search region is the higher half of the data.
6. Continue from Step 2.

Module 3 Linked List and Memory Management

Self Referential Structures, Dynamic Memory Allocation, Singly Linked List-Operations on Linked List. Doubly Linked List, Circular Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List Memory allocation and de-allocation-First-fit, Best-fit and Worst-fit allocation schemes

Self-Referential structures are those structures that have one or morepointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self-referential in nature.

Example: struct node

{

int data1;

char data2;

struct node* link;

};

int main()

{

struct node ob;return 0;

}

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as thereferencing pointer.

An important point to consider is that the pointer should be initializedproperly before accessing,

as by default it contains garbage value.

Types of Self Referential Structures

1 Self Referential Structure with Single Link

2 Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will showus how to connect the objects of a self-referential structure with the singlelink and access the corresponding data members. The connection formed is shown in the following figure.

Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Manycomplicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.

The connections made in the above example can be understood using thefollowing figure.

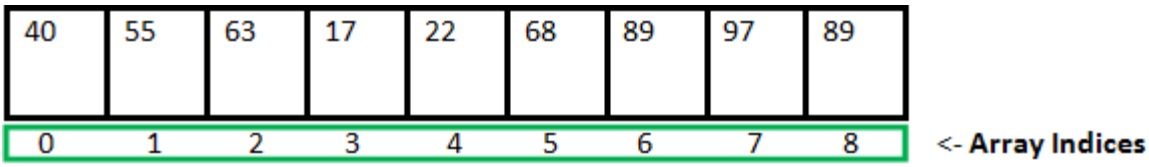
struct node {int data;

struct node* prev_link;struct node*
next_link;

};

DYNAMIC MEMEORY ALLOCATION

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An arrayis collection of items stored at continuous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. Butwhat if there is a requirement to change this length (size). For Example,

1. If there is a situation where only 5 elements are needed to be enteredin this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

2. Take another situation. In this, there is an array of 9 elements withall 9 indices filled. But there is a need to enter 3 more elements inthis array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

C **Dynamic Memory Allocation** can be defined as a procedure in whichthe size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitatedynamic memory allocation in C programming. They are:

(a) malloc()

(b) calloc()

(c) free()

(d) realloc()

malloc() method

“malloc” or **“memory allocation”** method in C is used to dynamicallyallocate a single large block of memory with the specified size. It returnsa pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

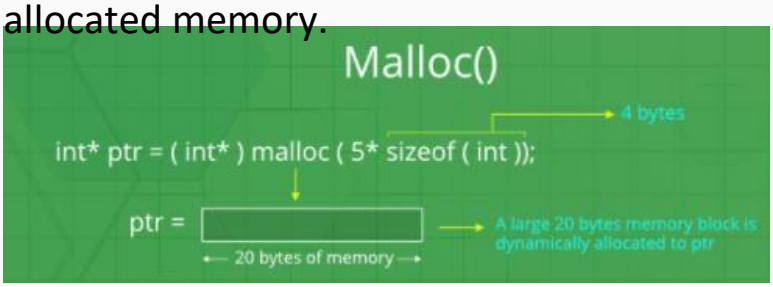
Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.




```
#include <stdio.h> #include
    <stdlib.h>

int main()
{

// This pointer will hold the
// base address of the block createdint* ptr;
int n, i;

// Get the number of elements for the arrayn = 5;
printf("Enter number of elements: %d\n", n);
```

```
// Dynamically allocate memory using malloc()ptr =
    (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or notif (ptr ==
    NULL) {
printf("Memory not allocated.\n");exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully
    allocated using malloc.\n");

// Get the elements of the arrayfor (i = 0; i < n;
    ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array printf("The elements of
    the array are: ");for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}
```

calloc() method

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

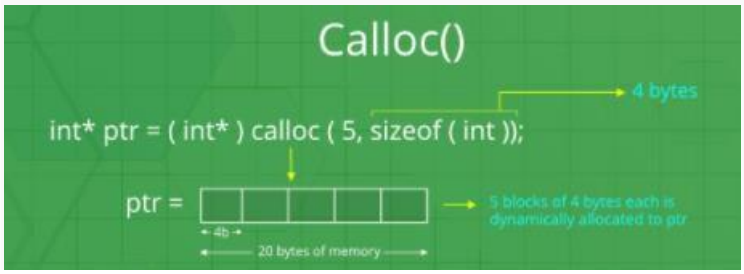
Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.




```
#include <stdio.h> #include
    <stdlib.h>


int main()
{


// This pointer will hold the
// base address of the block createdint* ptr;
int n, i;


// Get the number of elements for the arrayn = 5;
printf("Enter number of elements: %d\n", n);


// Dynamically allocate memory using calloc()ptr =
    (int*)calloc(n, sizeof(int));
```

```
// Check if the memory has been successfully
// allocated by calloc or notif (ptr ==
    NULL) {
printf("Memory not allocated.\n");
exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully
    allocated using calloc.\n");

// Get the elements of the arrayfor (i = 0; i < n;
    ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array printf("The elements of
    the array are: ");for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}
```

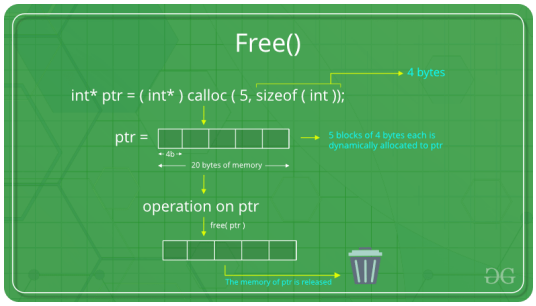
free() method

“free” method in C is used to dynamically **de-allocate** the memory. Thememory allocated using functions malloc() and calloc() is not de- allocated on their own. Hence the

free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);



Example:

#include <stdio.h> #include

<stdlib.h>int main()

{

// This pointer will hold the

// base address of the block createdint *ptr, *ptr1;

int n, i;

// Get the number of elements for the arrayn = 5;

printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using malloc()

```
ptr = (int*)malloc(n * sizeof(int));

// Dynamically allocate memory using calloc()ptr1 =
    (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) {
    printf("Memory not allocated.\n");exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully
    allocated using malloc.\n");

// Free the memoryfree(ptr);
printf("Malloc Memory successfully freed.\n");
```

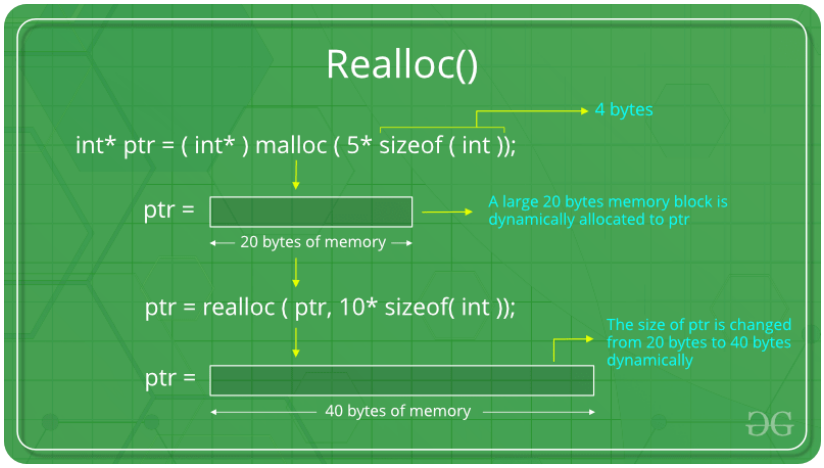
```
// Memory has been successfully allocated printf("\nMemory successfully
    allocated using calloc.\n");


// Free the memoryfree(ptr1);

printf("Calloc Memory successfully freed.\n");
}



return 0;
}
```

realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words,if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blockswill be initialized with default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);


where ptr is reallocated with new size 'newSize'
```



If space is insufficient, allocation fails and returns a NULL pointer.#include <stdio.h>

#include <stdlib.h>

int main()

{

// This pointer will hold the

// base address of the block createdint* ptr;

int n, i;

// Get the number of elements for the arrayn = 5;

printf("Enter number of elements: %d\n", n);

```
// Dynamically allocate memory using calloc()ptr =
    (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or notif (ptr ==
    NULL) {
printf("Memory not allocated.\n");exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully
    allocated using calloc.\n");

// Get the elements of the arrayfor (i = 0; i < n;
    ++i) {
ptr[i] = i + 1;

}
```

```
// Print the elements of the array printf("The elements of
    the array are: ");for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}

// Get the new size for the arrayn = 10;
printf("\n\nEnter the new size of the array: %d\n", n);

// Dynamically re-allocate memory using realloc()ptr = realloc(ptr, n
    * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the arrayfor (i = 5; i < n;
    ++i) {
ptr[i] = i + 1;
}
```

```
// Print the elements of the array printf("The elements of
    the array are: ");for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);

}



free(ptr);
}

return 0;
}
```

LINKED LISTS

A linked list, in simple terms, is a linear collection of data elements.These data elements are called *nodes*.

Linked list is a data structure which in turn can be used to implementother data structures.

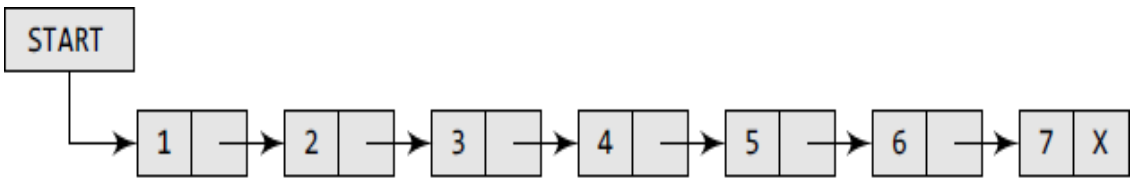


Figure 6.1 Simple linked list

In Fig, we can see a linked list in which every node contains twoparts, an integer and a pointer to the next node.

The left part of the node which contains data may include a simple data type, an array, or a structure.

The right part of the node contains a pointer to the next node (or address of the next node in sequence).

The last node will have no next node connected to it, so it will store a special value called NULL. In Fig, the NULL pointer is represented by X.

While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.

Linked lists contain a pointer variable START that stores the address of the first node in the list.

We can traverse the entire list using START which contains theaddress of the first node; the next part of the first node in turn stores the address of its succeeding node.

Using this technique, the individual nodes of the list will form achain of nodes.

If START = NULL, then the linked list is empty and contains nonodes.

In C, we can implement a linked list using the following code:struct node

{

int data;

struct node *next;

};

Let us see how a linked list is maintained in the memory.

(a) **In order to form a linked list, we need a structure called *node* which has two fields, DATA and NEXT.**

(b) DATA will store the information part and NEXT will store theaddress of the next node in sequence. Consider Fig. 6.2.

(c) In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START= 1, so the first data is stored at address 1, which is H.

(d) The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.

(e) The second data element obtained from address 4 is E.

(f) Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data.

(g) We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the endof the linked list.

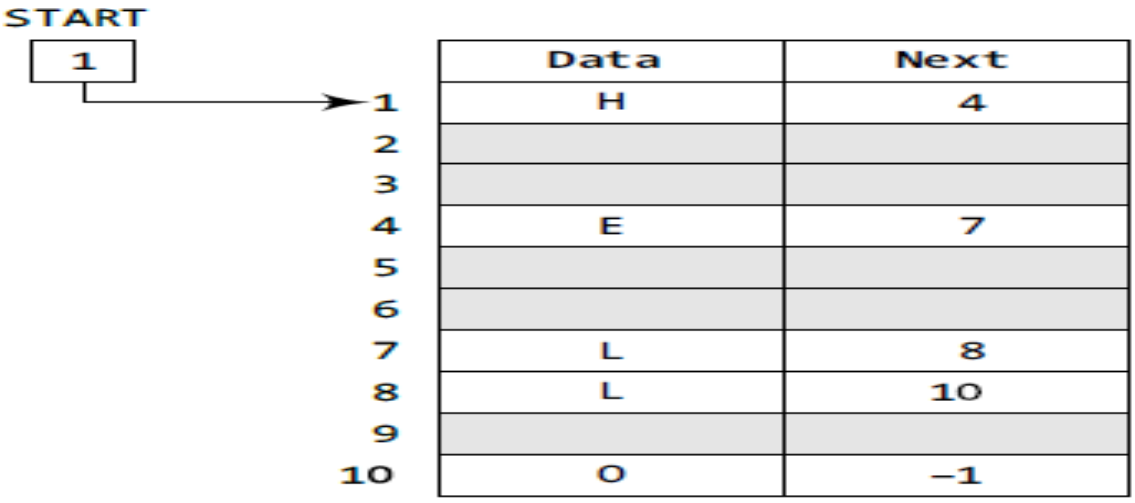


Figure 6.2 START pointing to the first element

Advantages

Linked list have many advantages. Some of the very importantadvantages are:

- ***Linked Lists are dynamic data structure***: That is, they cangrow or shrink during the execution of a program.
- ***Efficient memory utilization:*** Here, memory is not pre- allocated. Memory is allocated whenever it is required. Andit is deallocated when it is no longer needed.
- ***Insertion and deletions are easier and efficient:*** Linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- ***Many complex applications can be easily carried out withlinked lists.***

Disadvantages

- ***More Memory***: If the numbers of fields are more, thenmore memory space is needed.
- Access to an arbitrary data item is little bit cumbersome andalso time consuming.

Types of Linked List

Following are the various flavours of linked list.

- Simple Linked List − Item Navigation is forward only.
- Doubly Linked List − Items can be navigated forward andbackward way.
- Circular Linked List − Last item contains link of the firstelement as next and and first element has link to last element as prev.

Basic Operations

- Insertion − add an element at the beginning of the list.
- Display − displaying complete list.
- Search − search an element using given key.
- Delete − delete an element using given key

SINGLY LINKED Lists

(h) A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node ofthe same data type.

(i) A singly linked list allows traversal of data only in one way. Figure 6.7 shows a singly linked list

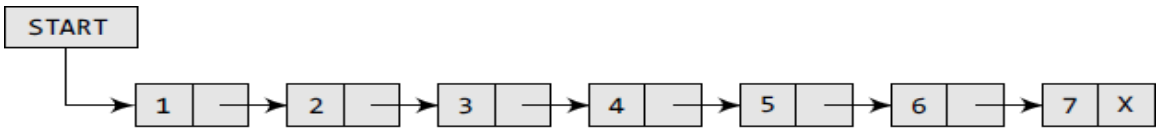


Figure 6.7 Singly linked list

LINKED LIST OPERATIONS

Traversing a Linked List

(j) Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.

(k) a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node.

(l) For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currentlybeing accessed.

(m) The algorithm to traverse a linked list is shown in Fig. 6.8.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR->DATA
Step 4:            SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: EXIT
```

Figure 6.8 Algorithm for traversing a linked list

(n) In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list.

(i) **Then in Step 2, a while loop is executed which is repeated tillPTR processes the last node, that is until it encounters NULL.**

(ii) **In Step 3, we apply the process (e.g., print) to the current node,that is, the node pointed by PTR.**

(o) In Step 4, we move to the next node by making the PTR

variable point to the node whose address is storedin the NEXT field.

Searching for a Value in a Linked List

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR->DATA
                  SET POS = PTR
                  Go To Step 5
             ELSE
                  SET PTR = PTR->NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

Consider the linked list shown in Fig. 6.11. If we have VAL = 4,then the flow of the algorithm can be explained as shown in thefigure.

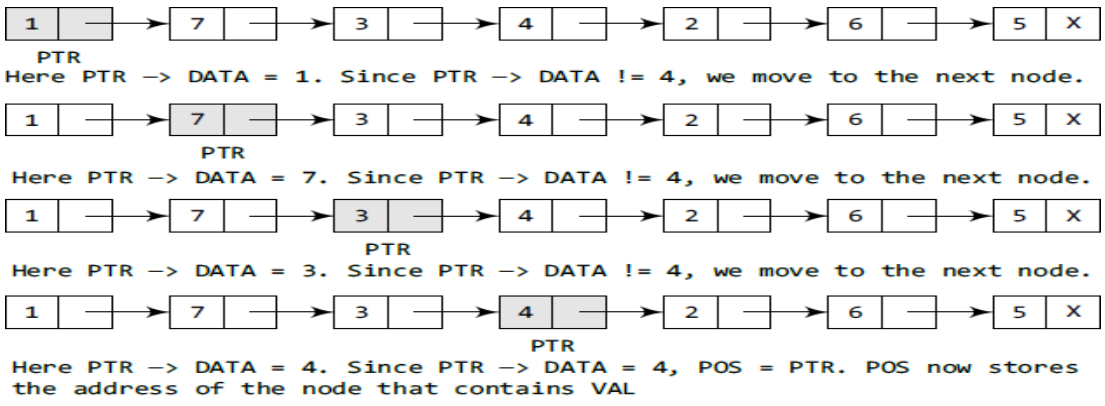


Figure 6.11 Searching a linked list

Steps to create a linked list

Step 1 : Include alloc.h Header

File#include<alloc.h>

1. We don‟t know, how many nodes user is going to createonce he execute the program.
2. In this case we are going to allocate memory usingDynamic Memory Allocation functions malloc.
3. Dynamic memory allocation functions are included inalloc.h

Step 2 : Define Node Structure

We are now defining the new global node which can beaccessible through any of the function.

```
struct node
{  int data;
struct node *next;
}*start=NULL;
```

Step 3 : Create Node using Dynamic Memory Allocation .Now weare creating one node dynamically using malloc function.We don‟thave prior knowledge about number of nodes , so we are calling malloc function to create node at run time.

```
new_node=(struct node *)malloc(sizeof(struct node));
```

Fill Information in newly Created Node ,Now we are accepting value from the user using scanf. Accepted Integer value is stored inthe data field. Whenever we create new node , Make its Next Field

as NULL. printf("Enter the data : "); scanf("%d",&new_node->data);

Step 4 : if(start==NULL) then new_node -> next = NULL;start = new_node;

otherwise ptr=start; while(ptr->next!=NULL)

ptr=ptr->next;

ptr->next = new_node; new_node->next=NULL;

step 5:continue this process till while(num!=-1)

Inserting a New Node in a Linked List

(p) Case 1: The new node is inserted at the beginning.

(q) Case 2: The new node is inserted at the end.

(r) Case 3: The new node is inserted after a given node.

CASE 1: Inserting a Node at the Beginning of a Linked List

(s) Consider the linked list shown in Fig. 6.12. Suppose we want toadd a new node with data 9 and

(t) add it as the first node of the list. Then the following changeswill be done in the linked list.

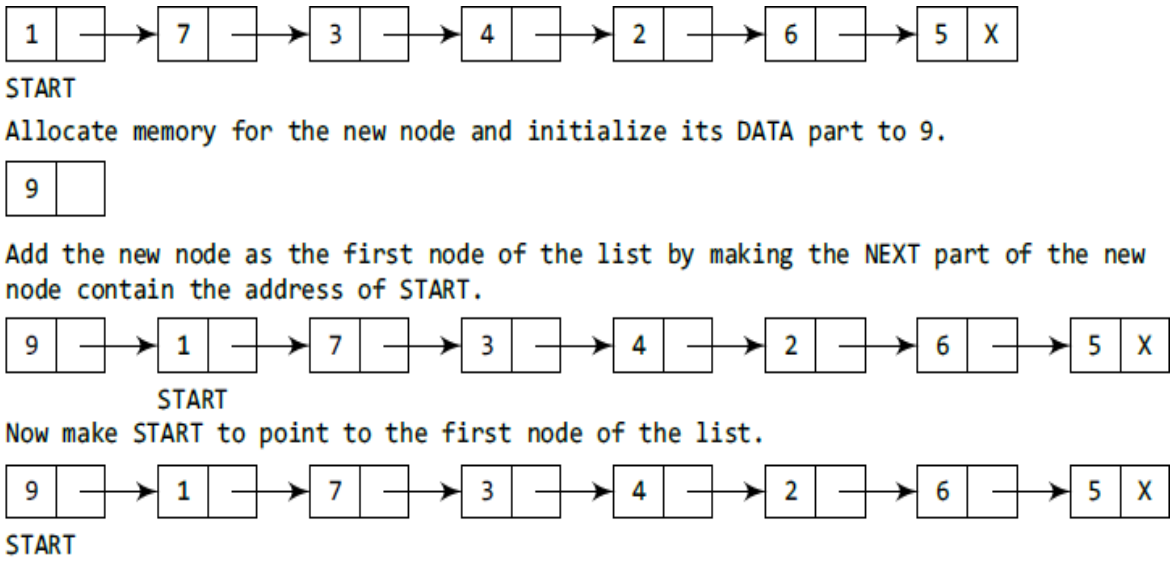


Figure 6.12 Inserting an element at the beginning of a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 6.13 Algorithm to insert a new node at the beginning

(u) In Step 1, we first check whether memory is available for thenew node. If the free memory has exhausted, then an OVERFLOW message is printed.

(v) Otherwise, if a free memory cell is available, then we allocatespace for the new node.

(w) Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which isstored in START.

(x) Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.

Note the following two steps:

(y) Step 2: SET NEW_NODE = AVAIL

(z) Step 3: SET AVAIL = AVAIL -> NEXT

(aa) These steps allocate memory for the new node.

Program

```
node *insert_beg(node *start)
{
node *new_node;int num;
printf("\n Enter the data : ");scanf("%d",
    &num);
new_node = (node *)malloc(sizeof(node));new_node -> data
    = num;
new_node -> next = start;start =
    new_node;
return start;
}
```

CASE 2: Inserting a Node at the End of a Linked List

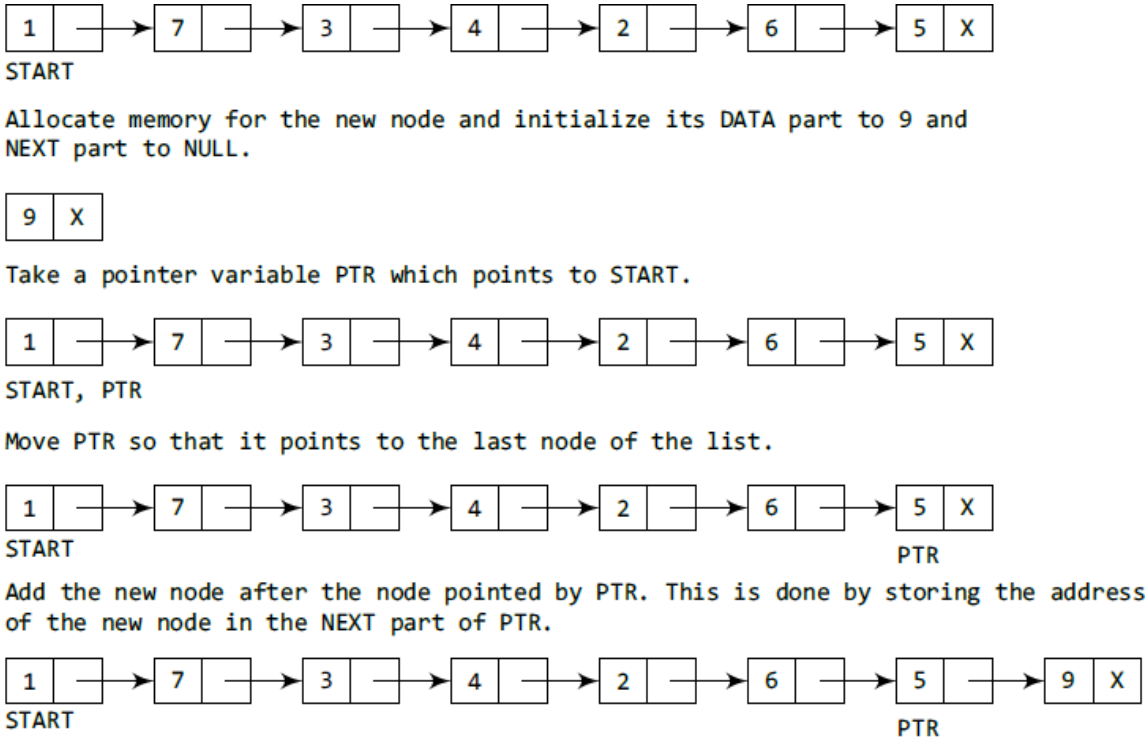


Figure 6.14 Inserting an element at the end of a linked list

(bb) Suppose we want to add a new node with data 9 as the last node ofthe list. Then the following changes will be done in the linked list.

```
Step 1:  IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
         [END OF IF]
Step 2:  SET NEW_NODE = AVAIL
Step 3:  SET AVAIL = AVAIL -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET NEW_NODE -> NEXT = NULL
Step 6:  SET PTR = START
Step 7:  Repeat Step 8 while PTR -> NEXT != NULL
Step 8:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 9:  SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.15 Algorithm to insert a new node at the end

1. In Step 6, we take a pointer variable PTR and initialize it withSTART. That is, PTR now points to the first node of the linkedlist.

2. In the while loop, we traverse through the linked list to reachthe last node.

2 Once we reach the last node, in Step 9, we change the NEXT pointerof the last node to store the address of the new node.

program

```
struct node *insert_end(struct node *start)
{
struct node *ptr, *new_node;int num;
printf("\n Enter the data : ");scanf("%d",
    &num);
new_node = (node *)malloc(sizeof(node));new_node -> data
    = num;
new_node -> next = NULL;ptr = start;
while(ptr -> next != NULL)ptr = ptr -> next;
ptr -> next = new_node;return start;
}
```

3 CASE 3: Inserting a Node After a Given Node in a Linked List

(cc) Consider the linked list shown in Fig. 6.17. Suppose we want to adda new node with value 9 after the node containing data 3

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
        != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Figure 6.16 Algorithm to insert a new node after a node that has value NUM

(dd) In Step 5, we take a pointer variable PTR and initialize it withSTART. That is, PTR now points to the first node of the linked list.

(ee) Then we take another pointer variable PREPTR which will be usedto store the address of the node preceding PTR.

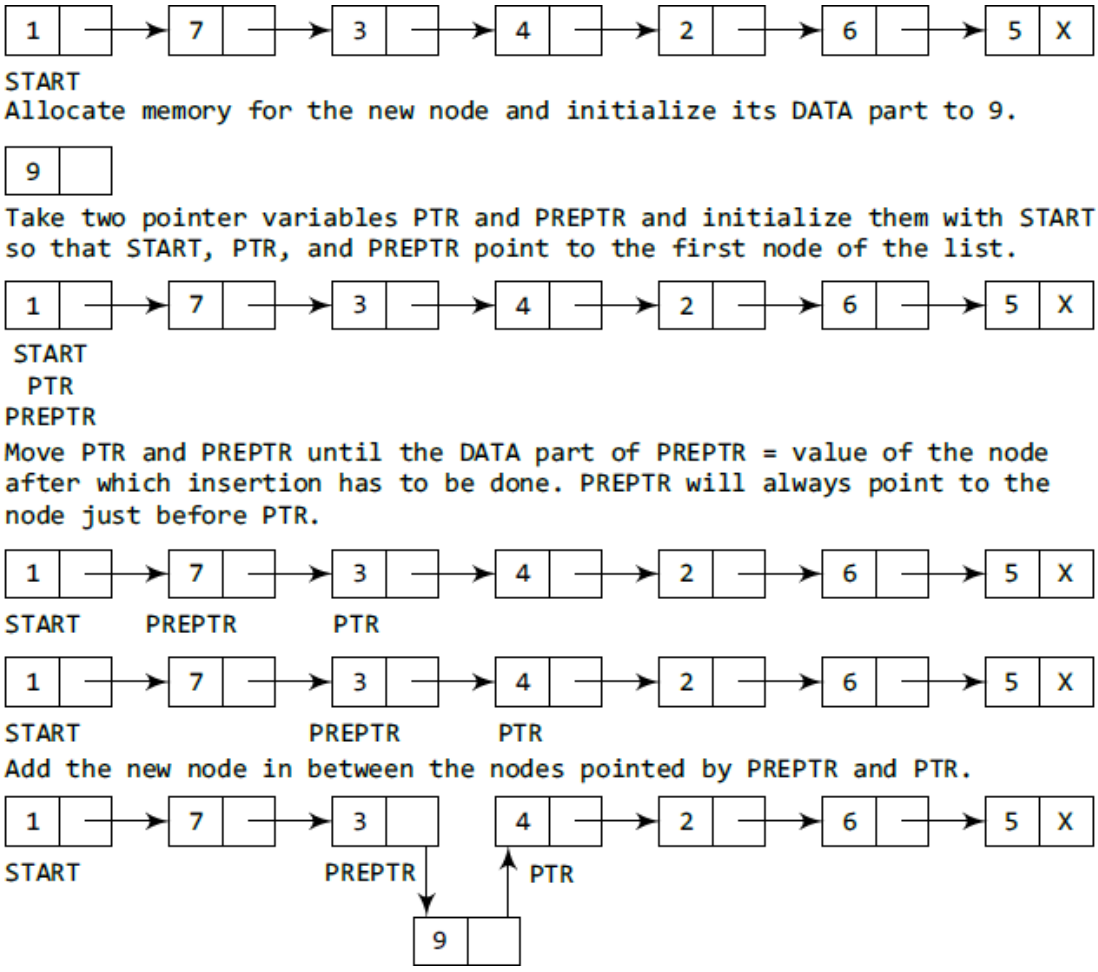
(ff) Initially, PREPTR is initialized to PTR.

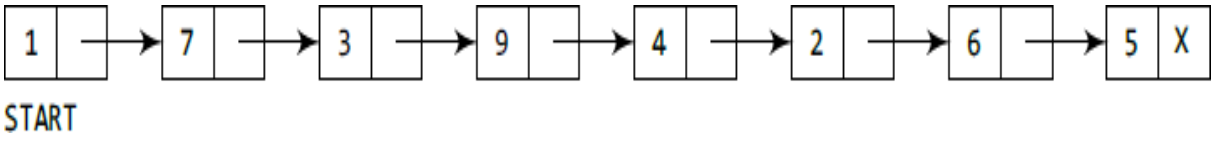
(gg) So now, PTR,PREPTR, and START are all pointing to the first nodeof the linked list.

(hh) In the while loop, we traverse through the linked list to reach thenode that has its value equal to NUM.

(ii) We need to reach this node because the new node will be insertedafter this node.

(jj) Once we reach this node, in Steps 10 and 11, we change the NEXTpointers in such a way that new node is inserted after the desired node.





Program

```
node *insert_after(node *start)
{
```

```
node *new_node, *ptr, *preptr;int num, val;

printf("\n Enter the data : ");scanf("%d",
    &num);

printf("\n Enter the value after which the data has to be inserted : ");scanf("%d", &val);

new_node = (node *)malloc(sizeof(node));new_node -> data
    = num;

ptr = start; preptr = ptr;

while(preptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next=new_node;new_node ->
    next = ptr; return start;
}
```

Deleting a Node from a Linked List

(kk) We will consider three cases and then see how deletion is done ineach case.

(ll) Case 1: The first node is deleted.

(mm) Case 2: The last node is deleted.

(nn) Case 3: The node after a given node is deleted.

CASE 1:Deleting the First Node from a Linked List

(oo) Before we describe the algorithms in all these three cases, let usfirst discuss an important term called UNDERFLOW.

(pp) Underflow is a condition that occurs when we try to delete a nodefrom a linked list that is empty.

(qq) This happens when START = NULL or when there are no morenodes to delete.

(rr) Note that when we delete a node from a linked list, we actually haveto free the memory occupied by that node.

(ss) The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, wealways change the AVAIL pointer so that it points to the address that has been recently vacated.

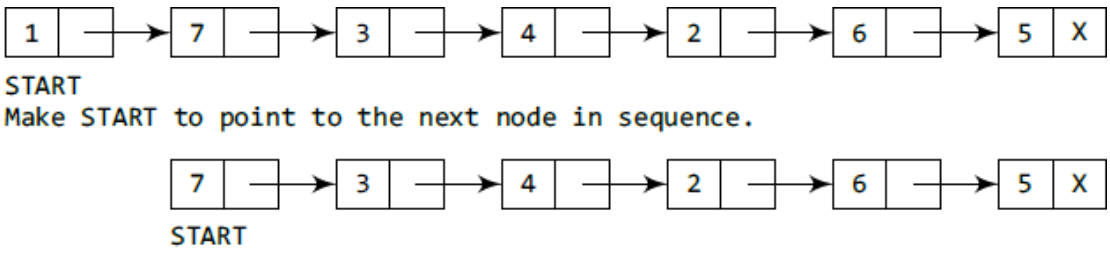


Figure 6.20 Deleting the first node of a linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

(tt) If START = NULL, then it signifies that there are no nodes in thelist and the control is transferred to the last statement of the algorithm.

(uu) if there are nodes in the linked list, then we use a pointer variablePTR that is set to point to the first node of the list.

(vv) For this, we initialize PTR with START that stores the address ofthe first node of the list.

(ww) In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the freepool.

Program

```
struct node *delete_beg(node *start)

{

struct node *ptr;ptr = start;


start = start -> next;
```

free(ptr); return start;

}

CASE2: Deleting the Last Node from a Linked List

(xx) Consider the linked list shown in Fig. 6.22. Suppose we want to delete the last node from the linked list, then the following changeswill be done in the linked list.

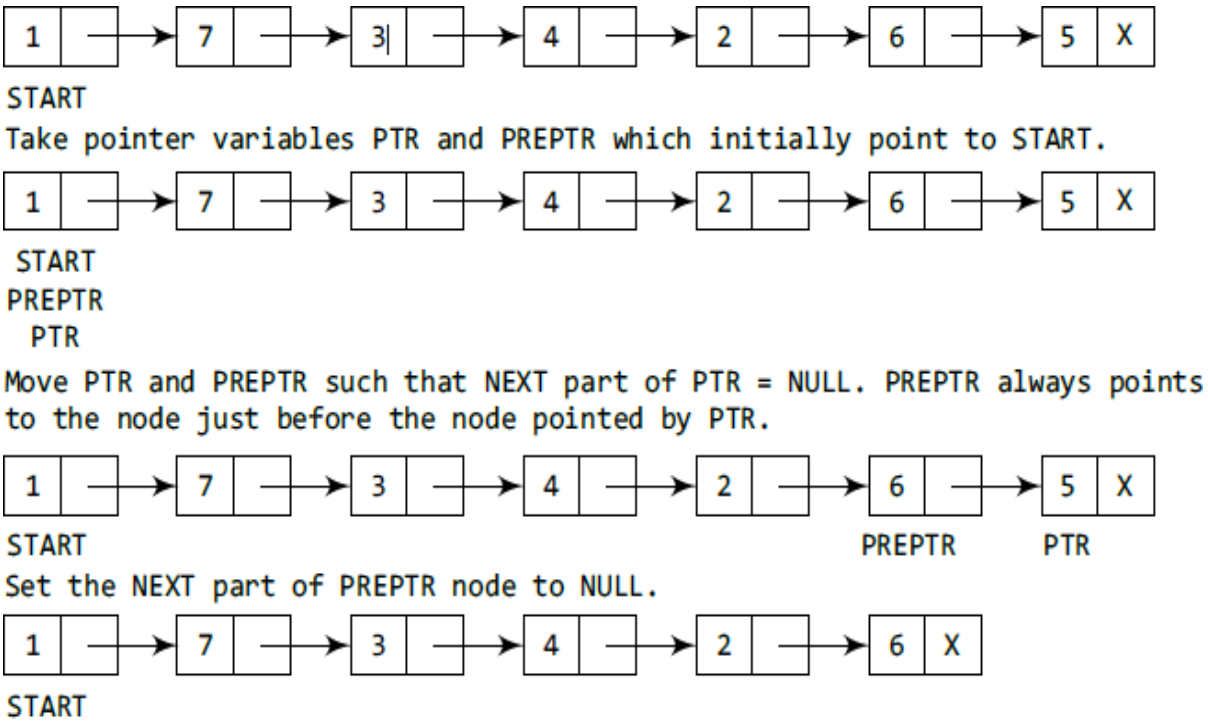


Figure 6.22 Deleting the last node of a linked list

(yy) Figure 6.23 shows the algorithm to delete the last node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START.

(zz) That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.

(aaa) Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.

(bbb) The memory of the previous last node is freed and returned back to the free pool

---

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.23 Algorithm to delete the last node

Program

```
node *delete_end(node *start)

{

node *ptr, *preptr;ptr =

    start;

while(ptr -> next != NULL)
```

{

preptr = ptr;

ptr = ptr -> next;

}preptr -> next = NULL;free(ptr);

return start;}

CASE 3:Deleting the Node After a Given Node in a Linked List

(ccc) In Step 2, we take a pointer variable PTR and initialize it withSTART. That is, PTR now points to the first node of the linked list.

(ddd) In the while loop, we take another pointer variable PREPTR suchthat it always points to one node before the PTR.

(eee) Once we reach the node containing VAL and the node

(fff) succeeding it, we set the next pointer of the node containing VAL tothe address contained in next field of the node succeeding it.

(ggg) The memory of the node succeeding the given node is freed andreturned back to the free pool.

```
Step 1:  IF START = NULL
              Write UNDERFLOW
              Go to Step 10
         [END OF IF]
Step 2:  SET PTR = START
Step 3:  SET PREPTR = PTR
Step 4:  Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:       SET PREPTR = PTR
Step 6:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 7:  SET TEMP = PTR
Step 8:  SET PREPTR -> NEXT = PTR -> NEXT
Step 9:  FREE TEMP
Step 10: EXIT
```

Figure 6.25 Algorithm to delete the node after a given node

(hhh) Consider the linked list shown in Fig. 6.24.

(iii) Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list

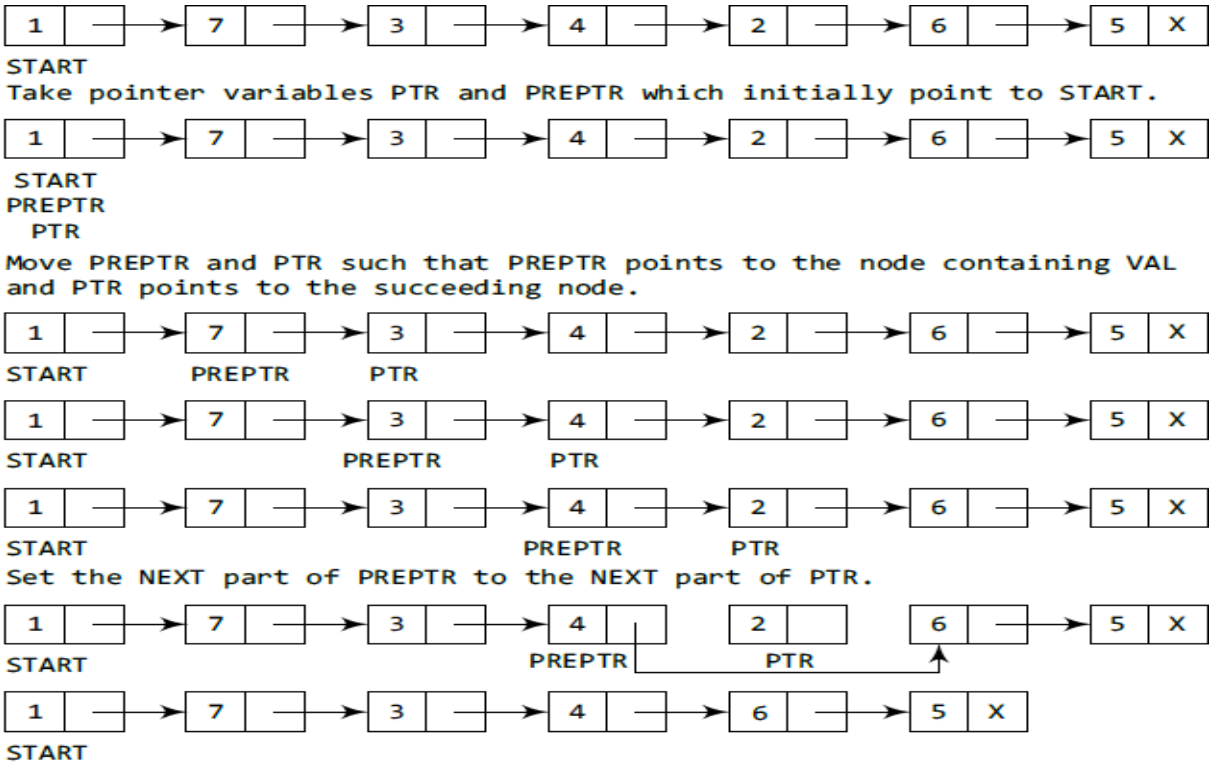


Figure 6.24 Deleting the node after a given node in a linked list

•

Program

```
node *delete_
{
node *ptr, *preptr;int val;

printf("\n Enter the value after which the node has to deleted : ");
```

```
scanf("%d", &val);ptr =
    start;
preptr = ptr;
while(preptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next=ptr -> next;free(ptr);
return start;
}
```

CIRCULAR LINKED LISTs

(jjj) In a circular linked list, the last node contains a pointer to the firstnode of the list.

(kkk) We can have a **circular singly linked list** as well as a **circulardoubly linked list**.

(lll) While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until wereach the same node where we started.

(mmm) Thus, a circular linked list has no beginning and no ending. Figure 6.26 shows a circular linked list.

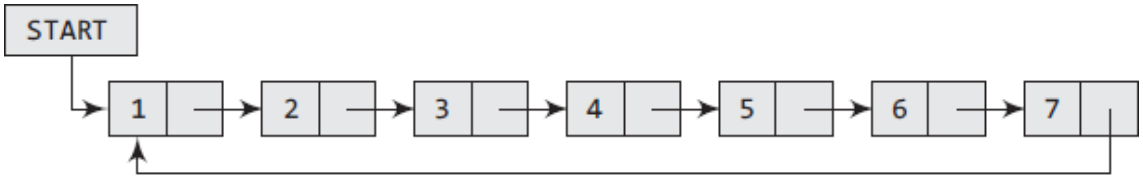


Figure 6.26 Circular linked list

Inserting a New Node in a Circular Linked List

(nnn) Case 1: The new node is inserted at the beginning of the circularlinked list.

(ooo) Case 2: The new node is inserted at the end of the circular linkedlist.

CASE 1:Inserting a Node at the Beginning of a Circular LinkedList

(ppp) Consider the linked list shown in Fig. 6.29. Suppose we want to adda new node with data 9 as the first node of the list.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Figure 6.30 Algorithm to insert a new node at the beginning

(iii) Figure 6.30 shows the algorithm to insert a new node at thebeginning of a linked list. In Step 1, we first check whether memory

is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.

(iv) Otherwise, if free memory cell is available, then we allocate space for the new node.

(v) Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.

(vi) Now, since the new node is added as the first node of the list, it willnow be known as the START node,that is, the START pointer variable will now hold the address of the NEW_NODE.

(vii) While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list.

(viii) Because the last node contains a pointer to START, its NEXT fieldis updated so that after insertion it points to the new node which willbe now known as START

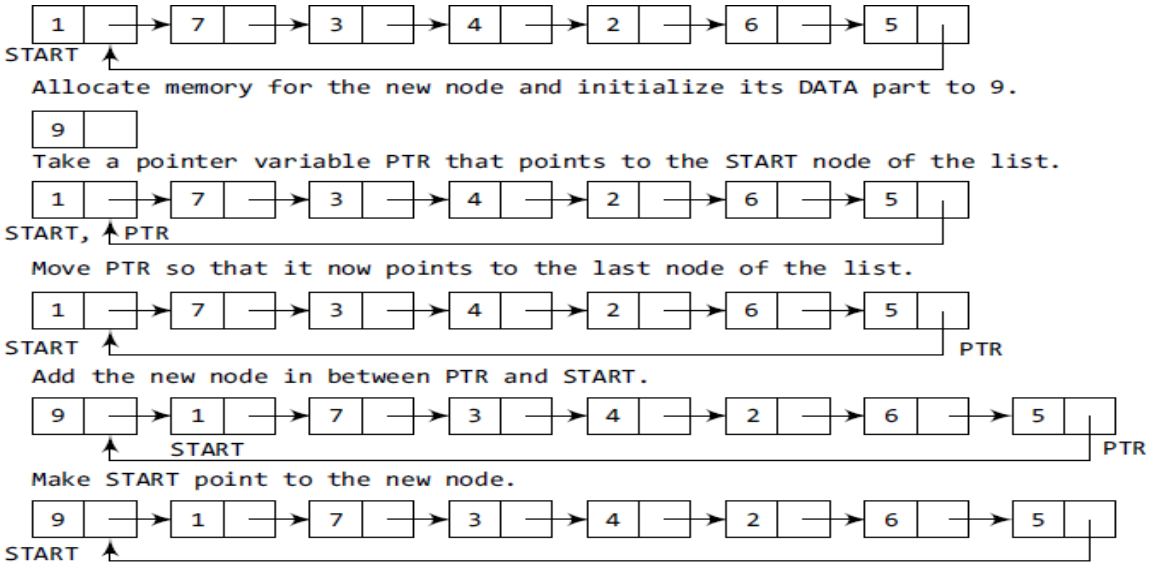


Figure 6.29 Inserting a new node at the beginning of a circular linked list

Program insert new node at beginning

```
struct node *insert_beg(struct node *start)
{
struct node *new_node, *ptr;int num;
printf("\n Enter the data : ");
    scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));new_node ->
    data = num;
ptr = start;
while(ptr -> next != start)ptr = ptr -
    > next;
ptr -> next = new_node; new_node
    -> next = start;start =
    new_node;
return start;
}
```

CASE 2: Inserting a Node at the End of a Circular Linked List

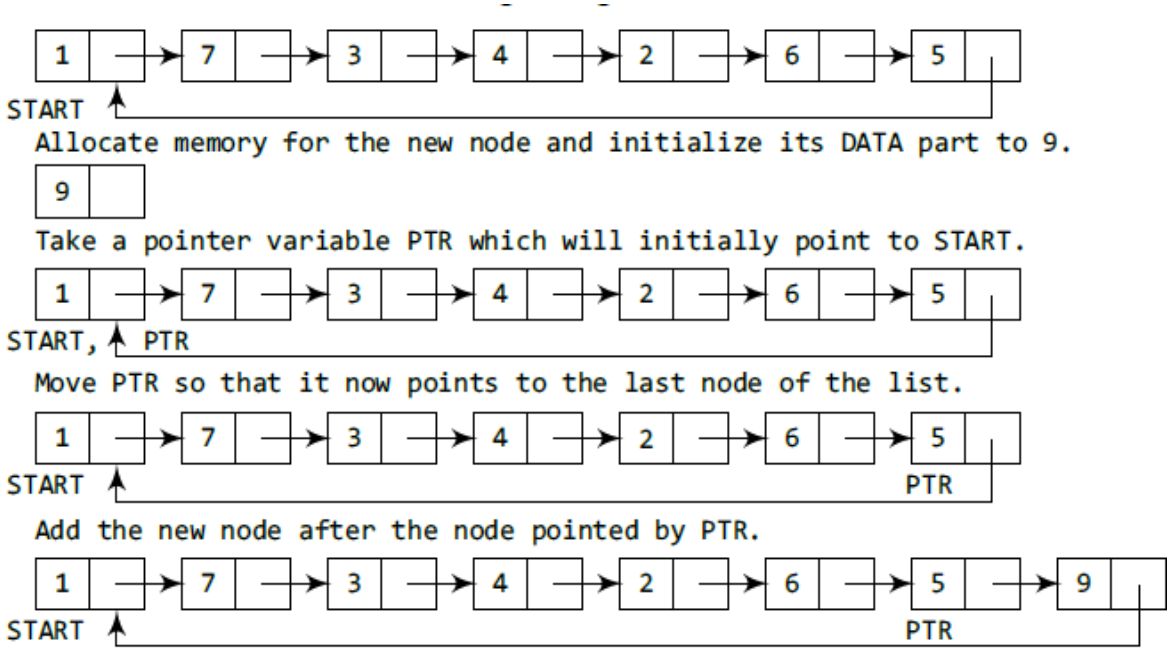


Figure 6.31 Inserting a new node at the end of a circular linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.32 Algorithm to insert a new node at the end

(qqq) Figure 6.32 shows the algorithm to insert a new node at the end of acircular linked list.

(rrr) In Step 6, we take a pointer variable PTR and initialize it withSTART.

(sss) That is, PTR now points to the first node of the linked list.

(ttt) In the while loop, we traverse through the linked list to reach the lastnode.

(uuu) Once we reach the last node, in Step 9, we change the NEXT pointerof the last node to store the address of the new node.

(vvv) Remember that the NEXT field of the new node contains the addressof the first node which is denoted by START.

Program

struct node *insert_end(struct node *start)

```
{
struct node *ptr, *new_node;int num;
printf("\n Enter the data : ");
    scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));new_node ->
    data = num;
ptr = start;
while(ptr -> next != start)ptr = ptr -
    > next;
ptr -> next = new_node; new_node
    -> next = start;return start;}
```

Deleting a Node from a Circular Linked List

(www) Case 1: The first node is deleted.

(xxx) Case 2: The last node is deleted**.**

CASE 1: Deleting the First Node from a Circular Linked List

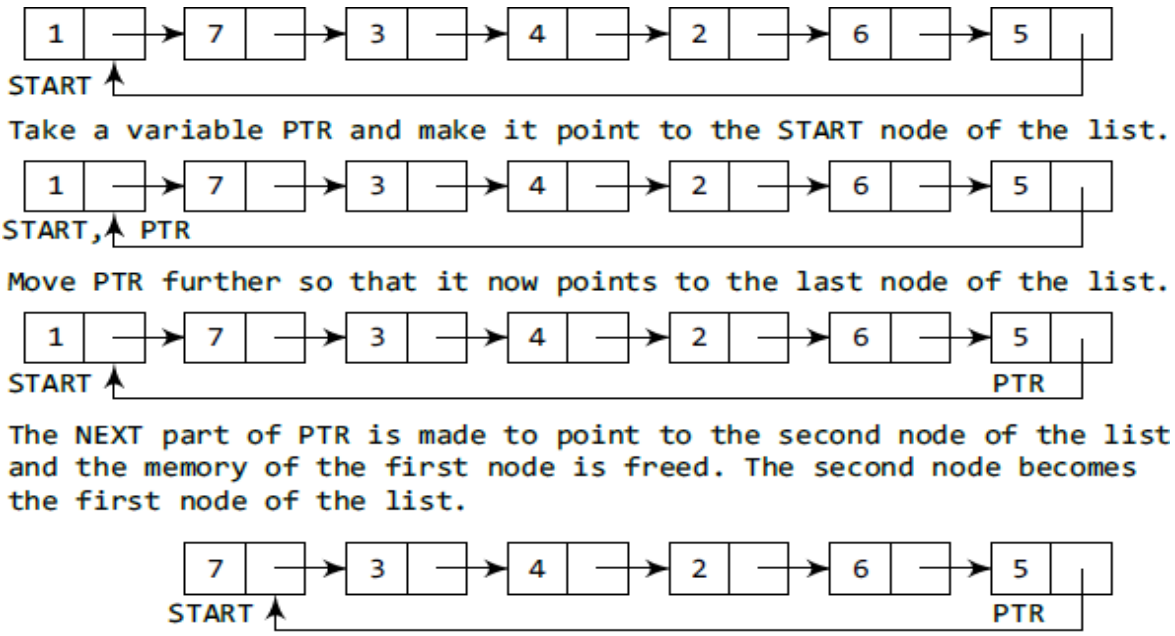


Figure 6.33 Deleting the first node from a circular linked list

(yyy) In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.

(zzz) However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.

(aaaa) In Step 5, we change the next pointer of the last node to point to thesecond node of the circular linked list.

(bbbb) In Step 6, the memory occupied by the first node is freed.

(cccc) Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:         SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

Figure 6.34 Algorithm to delete the first node

Program

```
struct node *delete_beg(struct node *start)
{
```

```
        struct node *ptr;
            ptr = start;

while(ptr -> next != start)ptr = ptr -
    > next;
ptr -> next = start -> next;
    free(start);

start = ptr -> next;return
    start;

}
```

CASE 2: Deleting the Last Node from a Circular Linked List

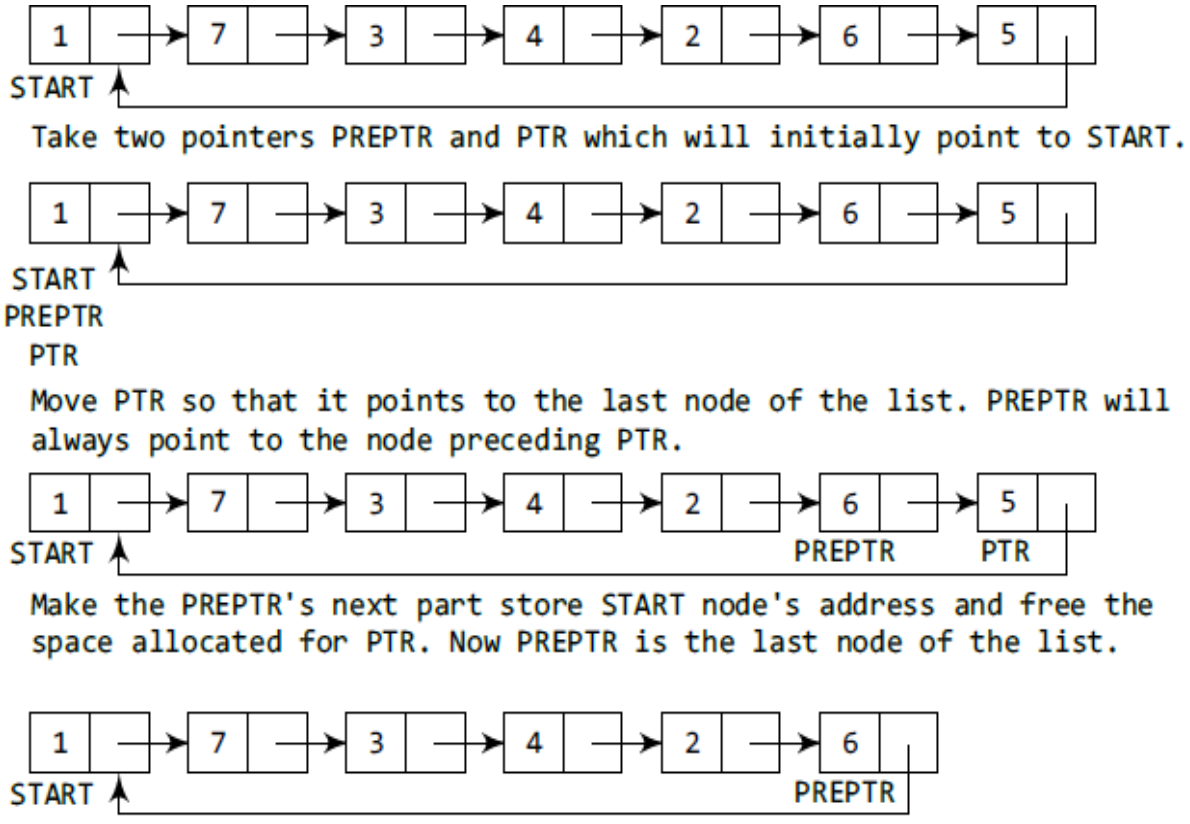


Figure 6.35 Deleting the last node from a circular linked list

(dddd) In Step 2, we take a pointer variable PTR and initialize it withSTART.

(eeee) That is,PTR now points to the first node of the linked list.

(ffff) In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.

(gggg) Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list.

(hhhh) The memory of the previous last node is freed and returned to the free pool.

program

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.36 Algorithm to delete the last node

program

```
struct node *delete_end(struct node *start)
{
```

```
struct node *ptr, *preptr;ptr = start;
while(ptr -> next != start)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next = ptr -> next;free(ptr);
return start;}
```

DOUBLY LINKED LISTS

(iiii) A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well asthe previous node in the sequence.

(jjjj) Therefore, it consists of three parts—data, a pointer to the nextnode, and a pointer to the previous node .

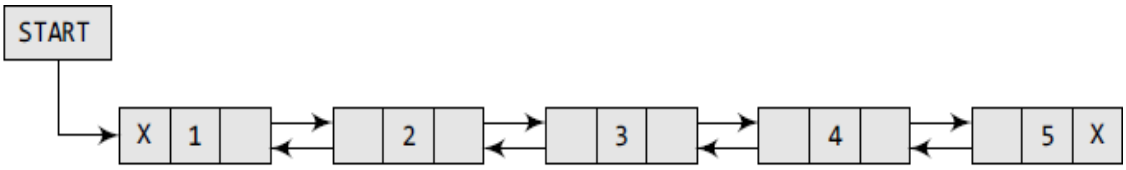


Figure 6.37 Doubly linked list

ı C, the structure of a doubly linked list can be given as,

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

(kkkk) The PREV field of the first node and the NEXT field of the last node will contain NULL.

(llll) The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

(mmmm) Thus, we see that a doubly linked list calls for more space per nodeand more expensive basic operations.

(nnnn) However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).

(oooo) The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Inserting a New Node in a Doubly Linked List

(pppp) Case 1: The new node is inserted at the beginning.

(qqqq) Case 2: The new node is inserted at the end.

(rrrr) Case 3: The new node is inserted after a given node.

CASE 1: Inserting a Node at the Beginning of a Doubly LinkedList

=

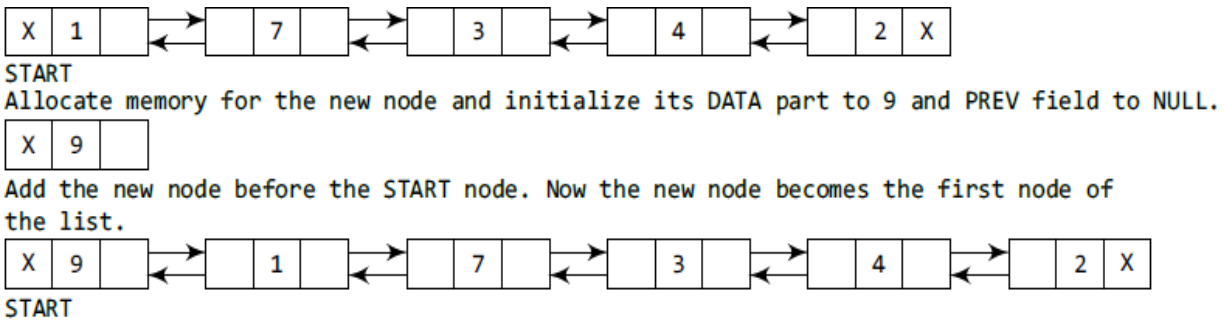


Figure 6.39 Inserting a new node at the beginning of a doubly linked list

(ssss) In Step 1, we first check whether memory is available for the new node.

(tttt) If the free memory has exhausted, then an OVERFLOW message is printed.

(uuuu) Otherwise, if free memory cell is available, then we allocate space for the new node.

(vvvv) Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.

(wwww) Now, since the new node is added as the first node of the list, it willnow be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Figure 6.40 Algorithm to insert a new node at the beginning

program

```
struct node *insert_beg(struct node *start)
{
struct node *new_node;
```

```
int num;
printf("\n Enter the data : ");
    scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));new_node ->
    data = num;
start -> prev = new_node;
new_node -> next = start;
new_node -> prev = NULL;start =
    new_node;
return start;
}
```

CASE 2: Inserting a Node at the End end of a Doubly Linked List

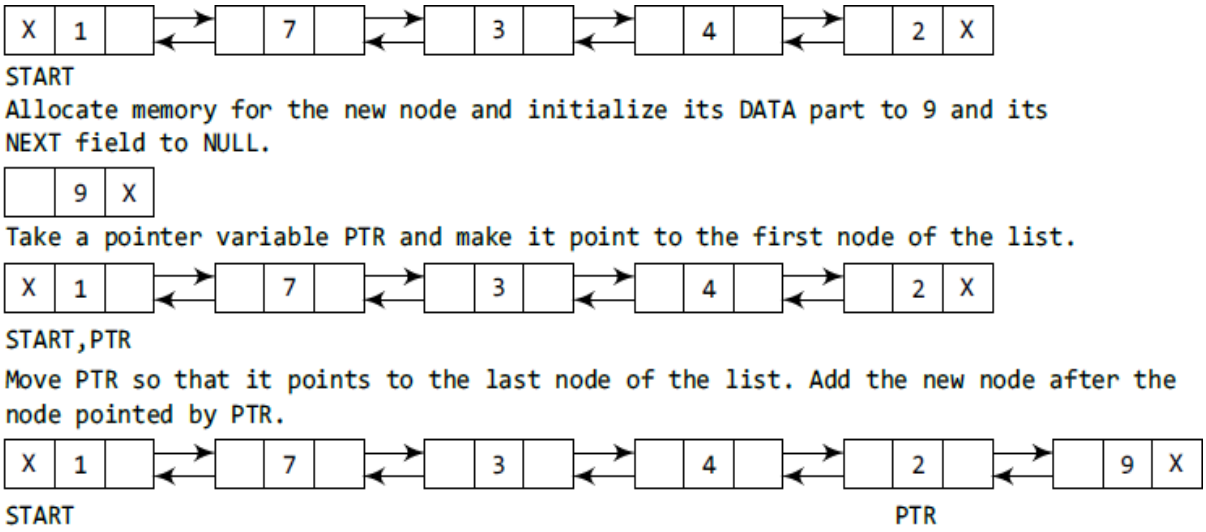


Figure 6.41 Inserting a new node at the end of a doubly linked list

(xxxx) Figure 6.42 shows the algorithm to insert a new node at the end of adoubly linked list. In Step 6, we take a pointer variable PTR and initialize it with START.

(yyyy) In the while loop, we traverse through the linked list to reach the last node.

(zzzz) Once we reach the last node, in Step 9, we change the NEXT pointerof the last node to store the address of the new node. Remember thatthe NEXT field of the

(aaaaa) new node contains NULL which signifies the end of the linked list.

(bbbbb) The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Figure 6.42 Algorithm to insert a new node at the end

Program

```
struct node *insert_end(struct node *start)
{
struct node *ptr, *new_node;int num;
```

```
printf("\n Enter the data : ");
    scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node));new_node ->
    data = num;
ptr=start;
while(ptr -> next != NULL)ptr = ptr ->
    next;
ptr -> next = new_node; new_node ->
    prev = ptr; new_node -> next =
    NULL;return start;
}
```

CASE 3: Inserting a Node After a Given Node in a Doubly Linked List

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 6.43 Algorithm to insert a new node after a given node

Figure 6.43 shows the algorithm to insert a new node after a given nodein a doubly linked list.

In Step 5, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.

We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

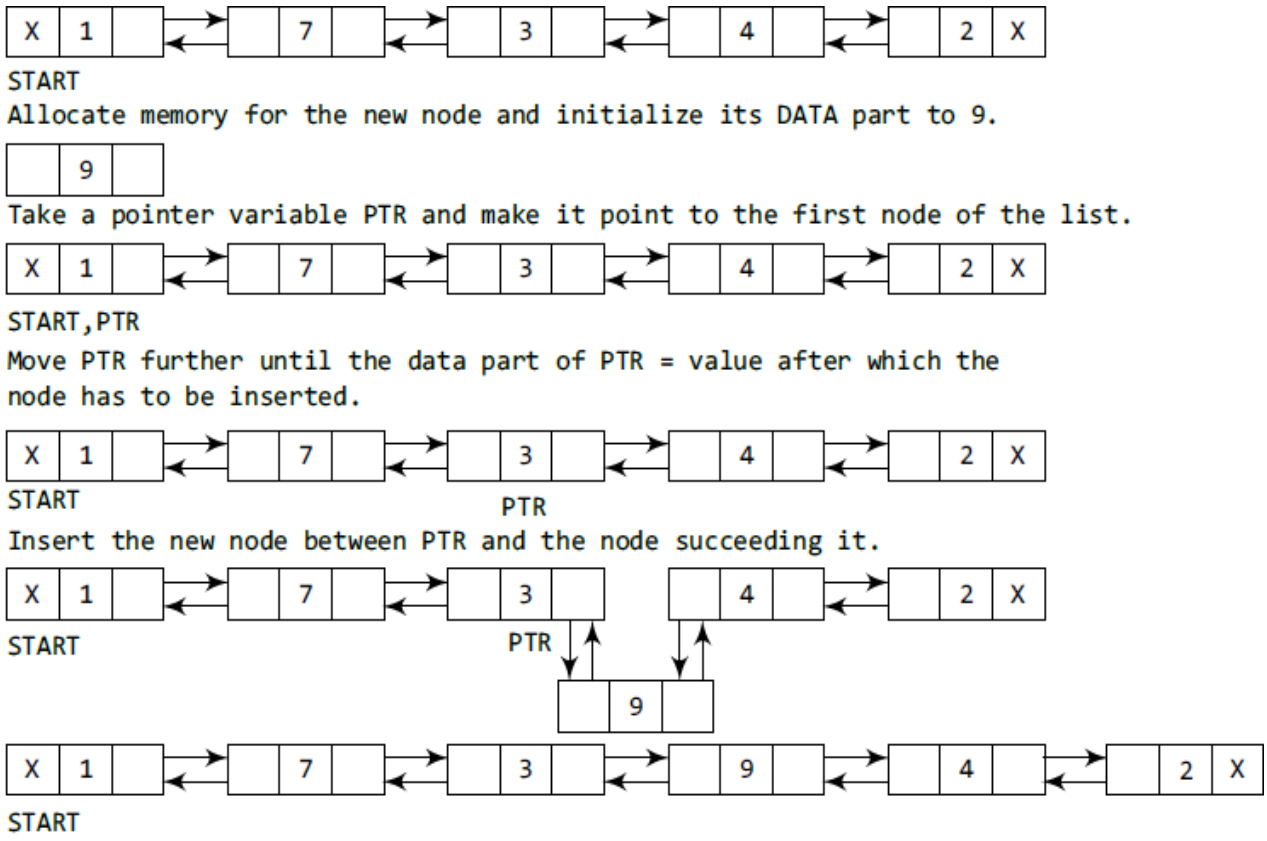


Figure 6.44 Inserting a new node after a given node in a doubly linked list

Program

```
struct node *insert_after(struct node *start)
{
```

```
struct node *new_node, *ptr;int num,
    val;

printf("\n Enter the data : ");
    scanf("%d", &num);

printf("\n Enter the value after which the data has to be inserted : ");scanf("%d", &val);

new_node = (struct node *)malloc(sizeof(struct node));new_node ->
    data = num;

ptr = start;

while(ptr -> data != val)ptr = ptr -
    > next; new_node -> prev =
    ptr;

new_node -> next = ptr -> next;ptr -> next
    -> prev = new_node; ptr -> next =
    new_node;

return start;

}
```

Deleting a Node from a Doubly Linked List

(ccccc) **Case 1: The first node is deleted.**

(ddddd) **Case 2: The last node is deleted.**

(eeeee) **Case 3: The node after a given node is deleted.**

CASE 1: Deleting the First Node from a Doubly Linked List

(fffff) When we want to delete a node from the beginning of the list, thenthe following changes will be done in the linked list.

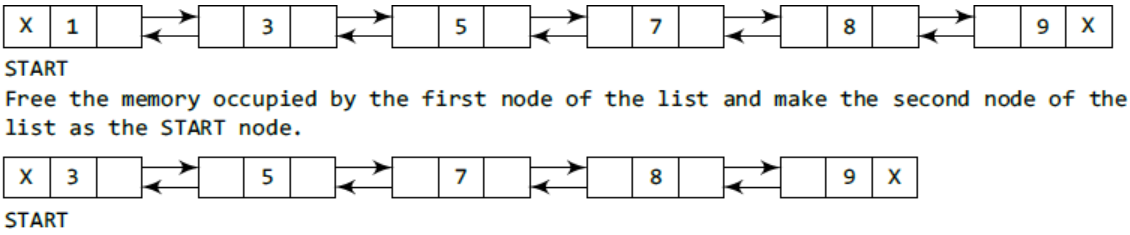


Figure 6.47 Deleting the first node from a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Figure 6.48 Algorithm to delete the first node

(ggggg) Figure 6.48 shows the algorithm to delete the first node of a doubly linked list.

(hhhhh) In Step 1 of the algorithm, we check if the linked list exists or not. If START =NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.

(iiiii) However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list.

(jjjjj) For this, we initialize PTR with START that stores the address of the first node of the list.

(kkkkk) In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

Program

```
struct node *delete_beg(struct node *start)

{

struct node *ptr;ptr =
    start;

start = start -> next;
```

start -> prev = NULL;free(ptr);

return start;

}

CASE 2: Deleting the Last Node from a Doubly Linked List

(IIIII) Suppose we want to delete the last node from the linked list, thenthe following changes will be done in the linked list.

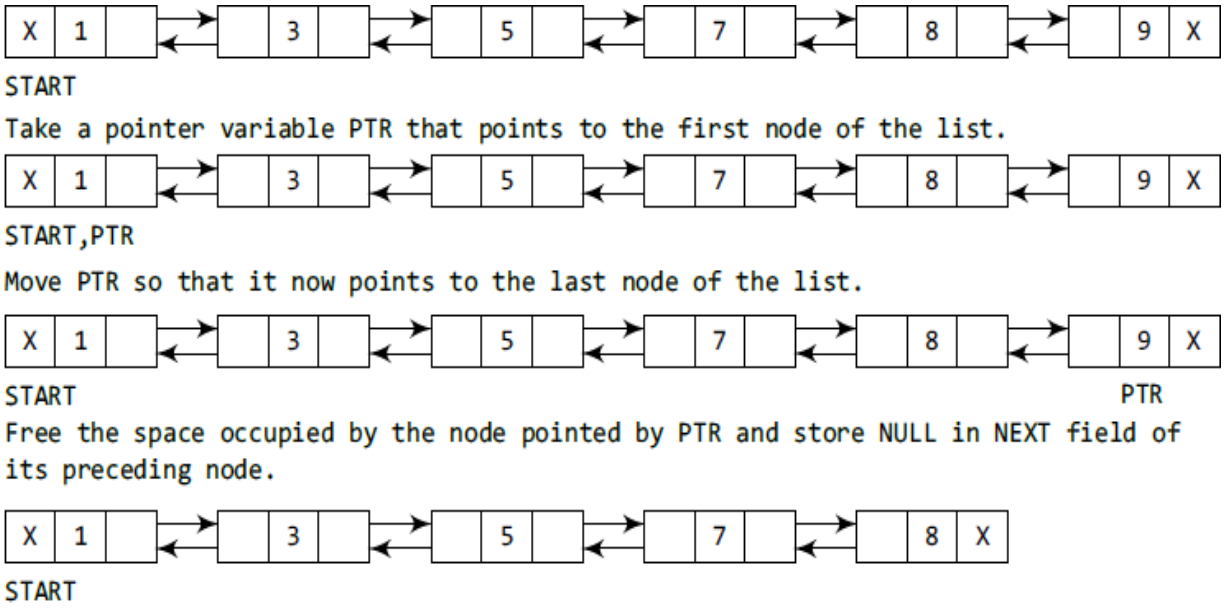


Figure 6.49 Deleting the last node from a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Figure 6.50 Algorithm to delete the last node

(mmmmm) Figure 6.50 shows the algorithm to delete the last node of a doublylinked list.

(nnnnn) In Step 2, we take a pointer variable PTR and initialize it with START.

(ooooo) That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.

(ppppp) Once we reach the last node, we can also access the second last nodeby taking its address from the PREV field of the last node.

(qqqqq) To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list.

(rrrrr) The memory of the previous last node is freed and returned to the free pool.

Program

```
struct node *delete_end(struct node *start)
{
struct node *ptr;ptr =
    start;
while(ptr -> next != NULL)ptr = ptr ->
    next;
ptr -> prev -> next = NULL;free(ptr);
return start;
}
```

CASE 3 : Deleting the Node After a Given Node in a Doubly Linked List

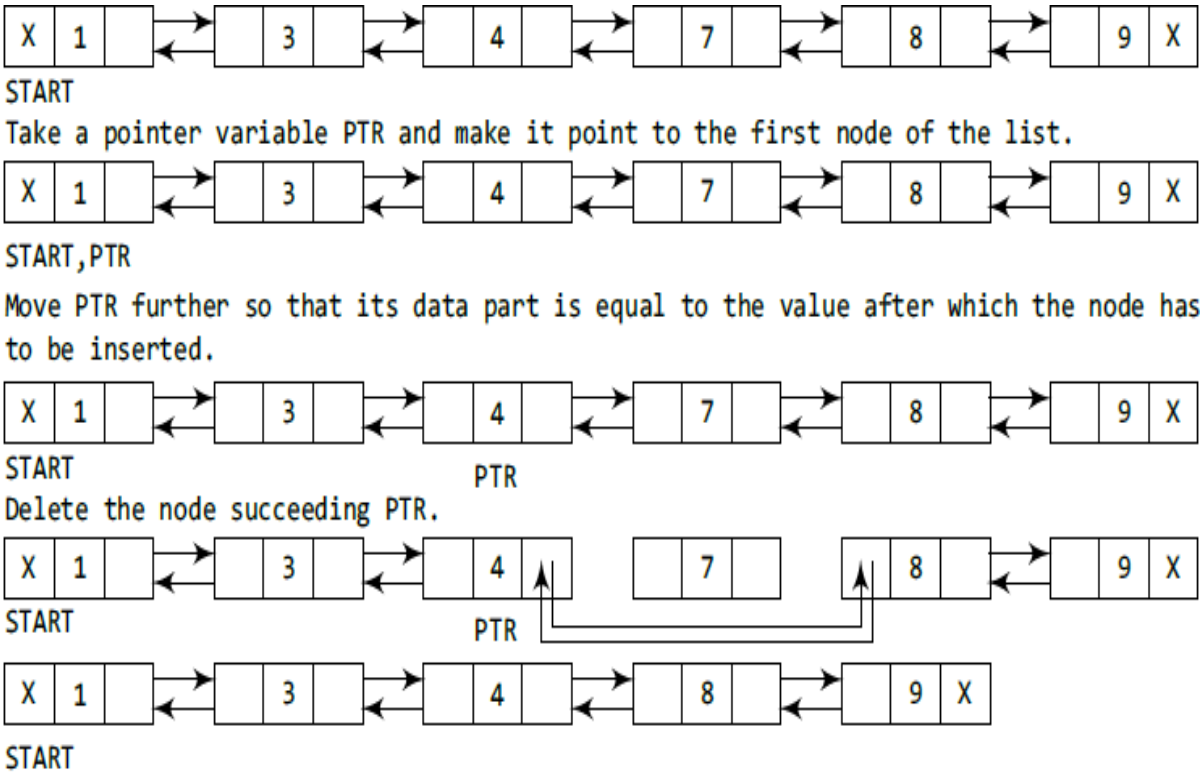


Figure 6.51 Deleting the node after a given node in a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Figure 6.52 Algorithm to delete a node after a given node

In Step 2, we take a pointer variable PTR and initialize it with

START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node.

(sssss) Once we reach the node containing VAL, the node

(ttttt) succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node.

(uuuuu) Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Program

```
struct node *delete_after(struct node *start)
{
struct node *ptr, *temp;int val;
printf("\n Enter the value after which the node has to deleted : ");scanf("%d", &val);
ptr = start;
while(ptr -> data != val)ptr = ptr -
    > next;
temp = ptr -> next;
ptr -> next = temp -> next; temp ->
    next -> prev = ptr;free(temp);
return start;
}
```

Header Linked Lists

(vvvvv) A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list, START will not point to the first node of the list but START will contain the address of the header node.

(wwwww) The following are the two variants of a header linked list:

(xxxxx) *Grounded header linked list* which stores NULL in the next field ofthe last node.

(yyyyy) *Circular header linked list* which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list.

Look at Fig. 6.65 which shows both the types of header linked lists.

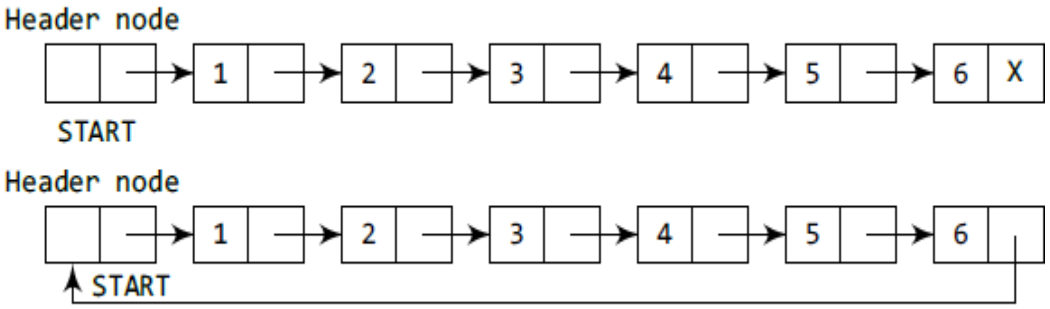


Figure 6.65 Header linked list

Application of linked list-Polynomial

Polynomial Addition

Linked list are widely used to represent and manipulate polynomials. Polynomials are the expressions containing number of terms with nonzero coefficient and exponents.In the linked representation of polynomials, each term is considered as a node.And such a node contains three fields

- Coefficient field

. Exponent field

- Link field

The coefficient field holds the value of the coefficient of aterm and the exponent field contains the exponent value of the term. And the link field contains the address of the next term inthe polynomial. The polynomial node structure is

Coefficient(coeff)	Exponent(expo)	Address of the next node(next)

Algorithm

Two polynomials can be added. And the steps involved inadding two polynomials are given below

1. Read the number of terms in the first polynomial P
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial Q
4. Read the coefficient and exponent of the secondpolynomial
5. Set the temporary pointers p and q to travers the twopolynomials respectively
6. Compare the exponents of two polynomials starting fromthe first nodes
    1. If both exponents are equal then add the coefficientand store it in the resultant linked list

2. If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointerq to point to the next node in the second polynomial Q.

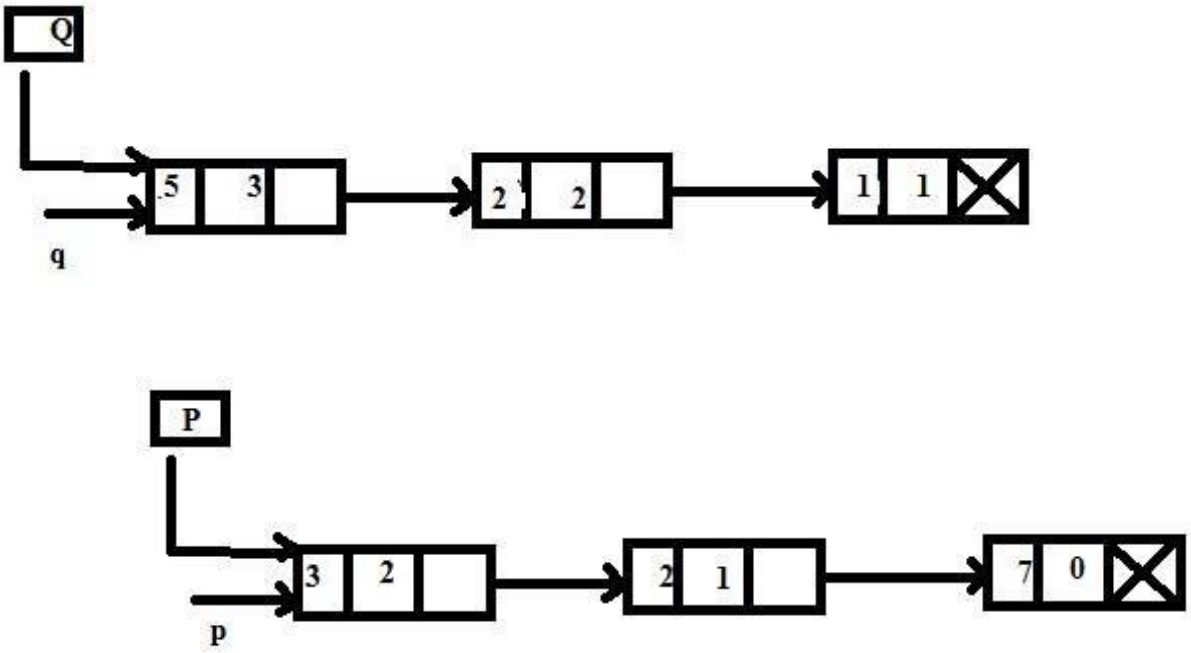
3. If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.

4. Append the remaining nodes of either of the polynomials to the resultant linked list.

Let us illustrate the way the two polynomials are added. Let p and qbe two polynomials having three terms each.

P=3x2+2x+7 Q=$5x^3+2x^2+x$

These two polynomial can be represented as



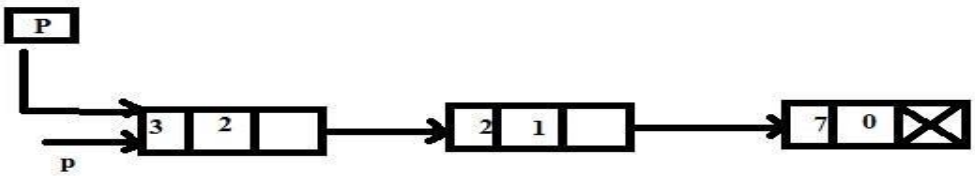
Step 1. Compare the exponent of p and the corresponding

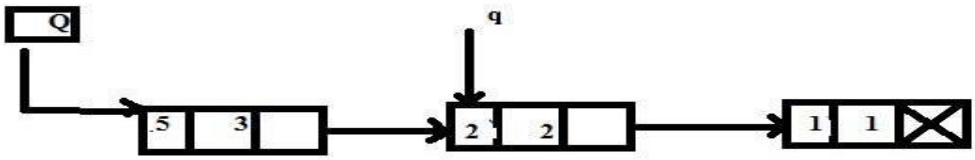
exponent of q. Here,

expo(p)<expo(q)

So, add the terms pointed to by q to the resultant list. And nowadvance the q pointer.

Step 2.



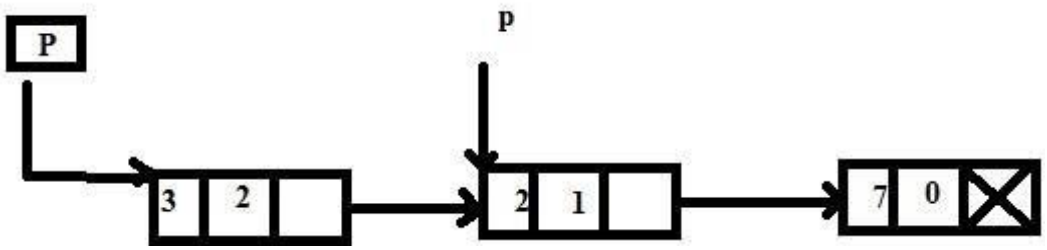

Q
q
.5
3
2
2
1
1

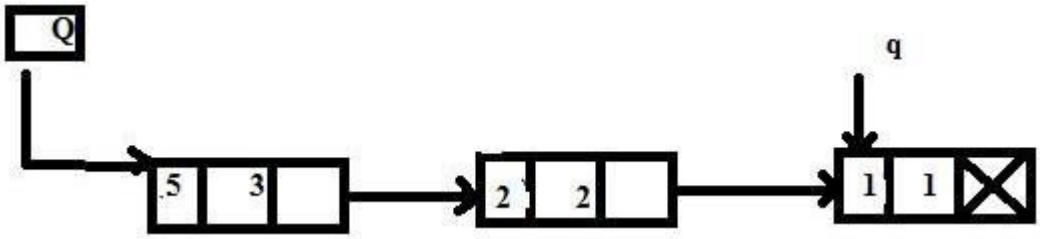
Compare the exponent of the current terms. Here,

expo(p)=expo(q)

So, add the coefficients of these two terms and link this to theresultant list. And,

advance the pointers p and q to their next nodes.



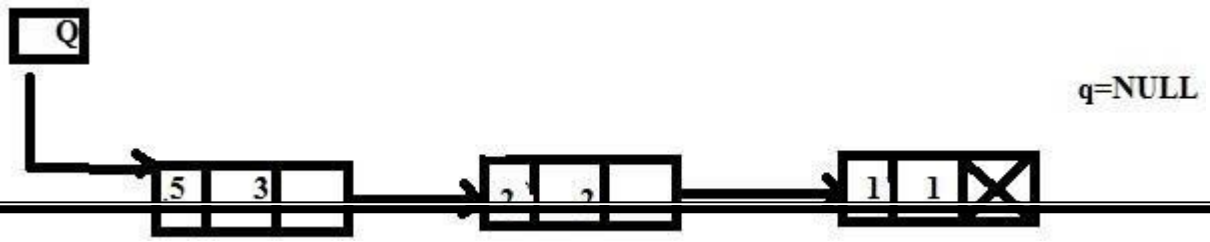


Compare the exponents of the current terms again

expo(p)=expo(q)

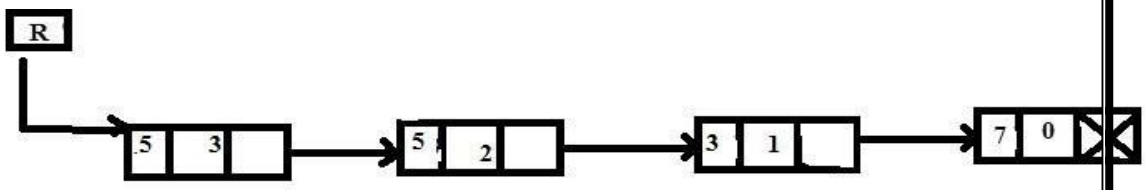
So, add the coefficients of these two terms and link this tothe resultant linked list.

And, advance the pointers to their next nodes. Q reaches the NULL and p points the last node.



There is no node in the second polynomial to compare with. So, the last node in the first polynomial is added to the end of theresultant linked list.

Step 5. Display the resultant linked list. The resultant linkedlist is pointed to by the pointer R



Algorithm for Polynomial Multiplication

1. Read the number of terms in the first polynomial
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial
4. Read the coefficient and exponent of the secondpolynomial
5. if one of the list is empty then the nonempty linked list isadded to the resultant linked list Otherwise goto step 6.
6. for each term of the first list

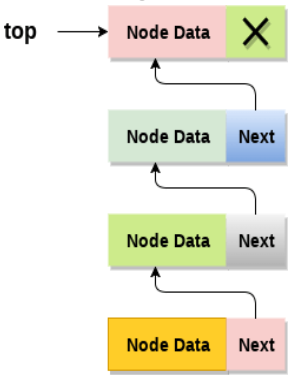
1. multiply each term of the second linked listwith a term of the first linked list
2. add the new term to the resultant polynomial
3. reposition the pointer to the starting of thesecond linked list
4. go to the next node
5. adds a term to the polynomial in thedescending order of the exponent

7. Display the resultant linked list

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push,pop and peek.

In linked list implementation of stack, the nodes are maintained non- contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown ifthe space left in the memory heap is not enough to create a node.



Stack

The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked listimplementation of stack.

Adding a node to the stack (Push operation)

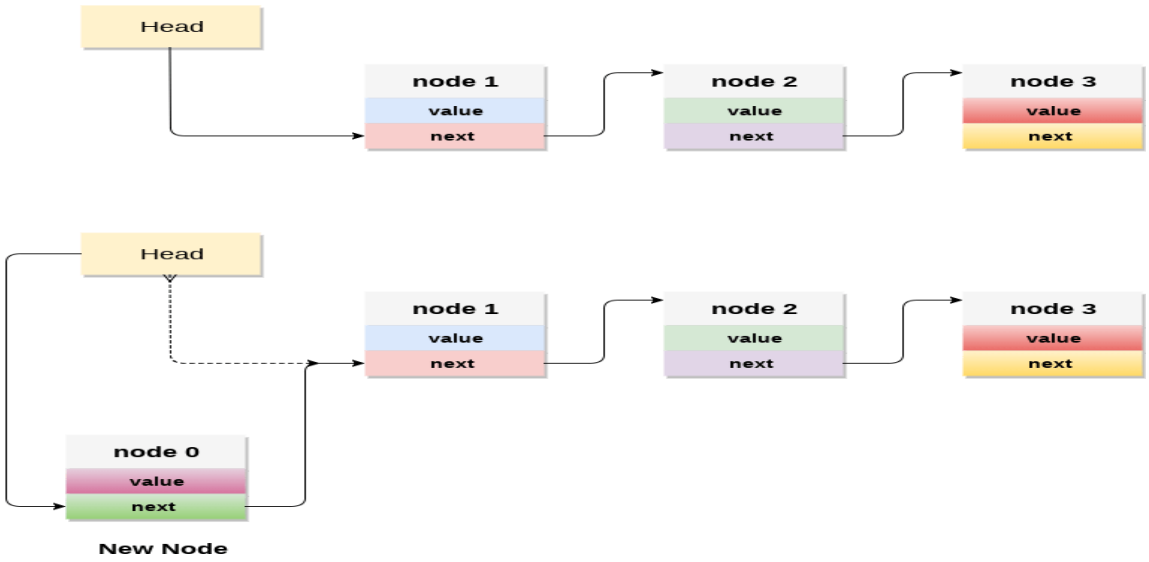
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that ofan array implementation. In order to push an element onto the stack, thefollowing steps are involved.

(a) Create a node first and allocate memory to it.

(b) If the list is empty then the item is to be pushed as the start node ofthe list. This includes assigning value to the data part of the node and assign null to the address part of the node.

(c) If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the propertyof the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)



C implementation :

```
void push ()
{
int val;
struct node *ptr =(struct node*)malloc(sizeof(struct node));if(ptr == NULL)
{
printf("not able to push the element");
}
else
{
printf("Enter the value");scanf("%d",&val);
    if(head==NULL)
{
ptr->val = val;
ptr -> next = NULL;head=ptr;
}
else
{
ptr->val = val; ptr->next = head;
    head=ptr;

}
printf("Item pushed");

}
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list

implementation of stack is different from that in the array implementation. In order to pop an element from the stack, weneed to follow the following steps :

- **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already emptystack. The stack will be empty if the head pointer of the listpoints to null.
- **Adjust the head pointer accordingly:** In stack, the elementsare popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
void pop()
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val;ptr = head;
head = head->next;free(ptr);
printf("Item popped");

}
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes ofthe linked list organized in the form of stack. For this purpose, we need to follow the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the listand print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
void display()
{
int i;
struct node *ptr;ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n");while(ptr!=NULL)
{
printf("%d\n",ptr->val);ptr = ptr->next;
}
}
}
```

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation cannot be used for the large scale applicationswhere the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

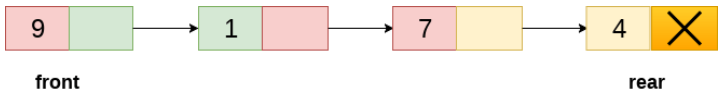
The storage requirement of linked representation of a queue with nelements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediatenext element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e.front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively.If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linkedqueues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the endof the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the followingstatement.

Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into thelinked queue.

In the first scenario, we insert element into an empty queue. In this case,the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item; if(front ==
    NULL)
{
front = ptr;rear = ptr;
front -> next = NULL;rear -> next =
    NULL;
}
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point tothe new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also needto make the next pointer of rear point to NULL.

```
rear -> next = ptr;rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and theC implementation is given as follows.

Algorithm

1. **Step 1:** Allocate the space for the new node PTR
2. **Step 2:** SET PTR -> DATA = VAL
3. **Step 3:** IF FRONT = NULL SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULLELSE

SET REAR -> NEXT = PTRSET REAR = PTR

SET REAR -> NEXT = NULL[END OF IF]

4. **Step 4:** END

C Function

```
void insert(struct node *ptr, int item; )

{


ptr = (struct node *) malloc (sizeof(struct node));if(ptr ==
    NULL)

{

printf("\nOVERFLOW\n"); return;

}

else

{

ptr -> data = item;if(front ==
    NULL)

{

front = ptr;rear = ptr;
```

```
front -> next = NULL;
```

```
rear -> next = NULL;
}
else
{
rear -> next = ptr;rear = ptr;
rear->next = NULL;
}
}
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer intothe pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
front = front -> next;free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

5. **Step 1:** IF FRONT = NULLWrite "
   Underflow "

Go to Step 5 [END OF IF]

6. **Step 2:** SET PTR = FRONT

7. **Step 3:** SET FRONT = FRONT -> NEXT
8. **Step 4:** FREE PTR
9. **Step 5:** END

C Function

```
void delete (struct node *ptr)
{
if(front == NULL)
{
printf("\nUNDERFLOW\n"); return;
}
else
{
ptr = front;
front = front -> next;free(ptr);
}
}
```

Memory management

Basic task of any program is to manipulate data.These data shouldbe stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:

- **Static storage management**
- **Dynamic storage management**
- **Static storage management** In case of static storage management scheme,the net amount of memory required for various data for a program is allocated before the starting of the execution of the program.Once memory is allocated, it neither can be extended nor can be returned to the memory bank for the use of other programs at the same time.

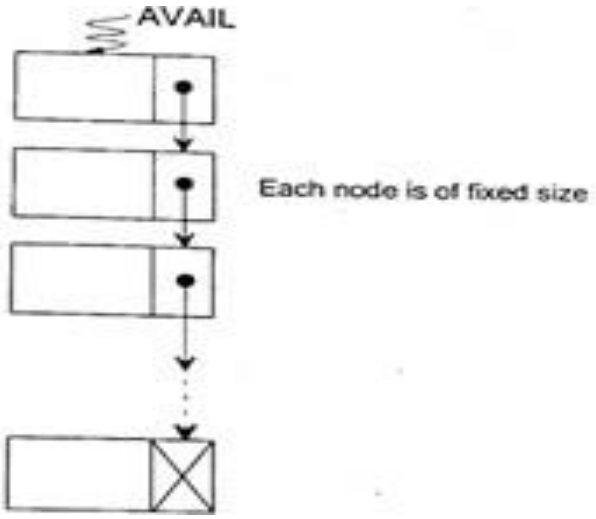
- **Dynamic storage management**

- Dynamic storage management scheme allows the user to allocate and deallocate as per the necessity during the execution of programs.
- This dynamic memory management scheme is suitable in multiprogramming as well as single-user environment where generallymore than one program reside in the memory and their memory requirement can be known only during their execution.
- Operating systems (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is linkedlist.
- Various principles on which the dynamic memory management scheme isbased on
  - **Allocation schemes**: here we discuss how a request for a memory block will be serviced.
    - **Fixed block allocation**
    - **Variable block allocation.**
    - ***First fit and its variant, (ii) Next fit , (iii) Best fit, (iv) Worst fit***
  - **Deallocation schemes**: here we discuss how to return a memory block to the memory bank whenever it is no more required.
    - **Random deallocation**
    - **Ordered deallocation.**
  - **Garbage collection**: to maintain a memory bank so that it can be utilized efficiently.

2. Allocation schemes

- Memory bank or pool of free storages is often a collection of non-contiguous blocks of memory.

- Their linearity can be maintained by means of pointers between one blockto another or in other words, memory bank is a linked list where links are to maintain the adjacency of blocks.



- Regarding the size of the blocks there are two practices: fixed block storageand variable block storage. Let us discuss each of them individually.
- **2.1 Fixed Block Storage**:

Here each block is of the same size.

- The size is determined by the system manager (user).
- Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks.
- A user program communicates with the memory manager by meansof two functions GETNODE(NODE) and RETURNNODE(ptr)

- **Procedure GETNODE(NODE)**

Steps

- **Start**
- **If (AVAIL = NULL) then**

Print "The memory is insufficient"

- **Else**

ptr = AVAIL

AVAIL = AVAIL->LINK

Return(ptr)

- **Endlf**
- **Stop**

- Whenever a memory block is no more required, it can be returned tothe memory bank through a procedure RETURNNODE( )

- **Procedure RETURNNODE(PTR)**
- Steps
- **Start**
- **ptrl = AVAIL**
- **While (ptrl->LINK # NULL) doptrl = ptr1->LINK**
- **EndWhile**
- **ptr1->LINK= PTR**
- **ptr->LINK= NULL**
- **Stop**

- Fixed block allocation is the simplest strategy. But the main drawbackof this strategy is the wastage of space.
- For example, suppose each memory block is of size 1 K (1024 bytes);now for a request of memory block, say, of size 1.1 K we have to avail2 blocks (that is 2 K memory space) thus wasting 0.9 K memory space.
- **Variable Block Storage**
- Here, the memory manager (a program of OS) maintains a pointerAVAIL which points a list of non-contiguous memory blocks.

- To overcome the disadvantages of fixed block storage, we can maintainblocks of variable sizes, instead of fixed size blocks.
- **Procedure GETNODE(NODE)**

Steps

- **Start**
- **If (AVAIL = NULL) then**
- **Print "Memory bank is insufficient"**
- **Exit**
- **Endlf**
- **ptr = AVAIL**
- **While (ptr->LINK ≠NULL) and (ptr->SIZE < SIZEOF(NODE))do**
- **ptrl = ptr**
- **ptr = ptr->LINK**
- **EndWhile**
- **If (ptr->LINK = NULL) and (ptr->SIZE < SIZEOF(NODE))then**
- **Print "Memory request is too large: Unable to serve"**
- **Else**
- **ptrl->LINK = ptr->LINK**
- **2. Return(ptr)**
- **Endlf10.Stop**
- This procedure assumes that blocks of memory are stored in ascending orderof their sizes.
- **Procedure RETURNNODE(PTR)**

Steps

- **Start**
- **ptrl = AVAIL**
- **While (ptrl->SIZE < ptr->SIZE) and (ptrl->LINK # NULL)) do**
- **ptr2 = ptrl**
- **ptrl = ptrl->LINK**
- **EndWhile**
- **ptr2->LINK = PTR**
- **PTR->LINK = ptrl**

* **Stop**

- The dynamic memory management system should provide thefollowing services:

o Searching the memory for a block of requested size and servicingthe request (allocation)

o Handling a free block when it is returned to the memory manager.

- Coalescing the smaller free blocks into larger block(s) (garbagecollection and compaction).

- **Storage allocation strategies**

- (a) First-fit allocation, (b) Best-fit allocation, (c) Worst-fitallocation, (d) Next-fit allocation

- **First-fit storage allocation:**

This is the simplest storage allocation strategy.

- Here the list of available storages will be searched and as soon as a free storage block of size N will be found pointer of that block will besent to the calling program after retaining the residue space.

- For example, for a block of size 2 K, if the first-fit strategy found a block of 7 K, then after retaining a storage block of size 5 K, 2 K memory will be sent to the caller.

- Leads to fast allocation of memory space.

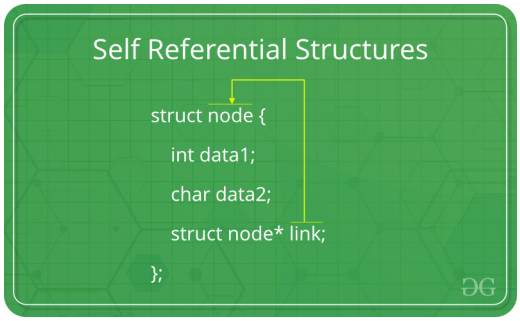
- Leads to memory waste

- **Best-fit storage allocation**
- This strategy will not allocate a block of size > N, as it is found in first-fit method, instead will continue searching to find a suitable block so that the block size is closer to the block size of request.
- Goal: find the smallest memory block into which the job will fit
- Results in least wasted space.
- Slower in making allocation
- For example, for a request of 2 K, if the list contains the blocks of sizes,1 K, 3 K, 7 K, 2.5 K, 5 K, then it will find the block of size 2.5 K as suitable block for allocation. From this block after retaining 0.5 K, pointer for 2 K block will be returned.
- **Worst-fit storage allocation**
- Slower in making allocation
- Allocates the largest free available block to the new job
- Opposite of best-fit
- Best-fit finds a block which is small and nearest to the block size as 'requested, whereas, worst-fit strategy is a reverse of it. It allocates thelargest block available in the available storage list.
- The idea behind the worst-fit is to reduce the rate of production of smallblocks which are quite common when best-fit strategy is used for memory allocation.
- **Next-fit storage allocation**
- The idea behind the worst-fit is to reduce the rate of production of small blocks which are quite common when best-fit strategy is used formemory allocation.
- Next-fit allocation strategy is a modification of first-fit strategy.

- Starts searching from last allocated block, for the next available block when a new job arrives.
- In case of first-fit strategy, searching will always occur from beginning of the free list whereas in next-fit strategy, search begins where the lastallocation has been done; in this strategy, pointer to the free list is savedfollowing an allocation and is used to begin for the subsequent request.

The idea of this strategy is to reduce the search by avoiding examinationof smaller blocks that, in long run, tends to be created at the beginningof the free list as it happens in case of first-f**SELF REFRENTIAL STRUCTURES**

Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self- referential in nature.

Example: struct node {

int data1;

char data2;

struct node* link;

};

int main()

{

struct node ob; return 0;

}

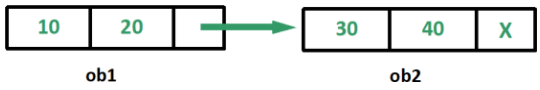
In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

Types of Self Referential Structures

1. Self Referential Structure with Single Link
2. Self Referential Structure with Multiple Links

Self Referential Structure with Single Link: These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



Self Referential Structure with Multiple Links: Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.

The connections made in the above example can be understood using the following figure.

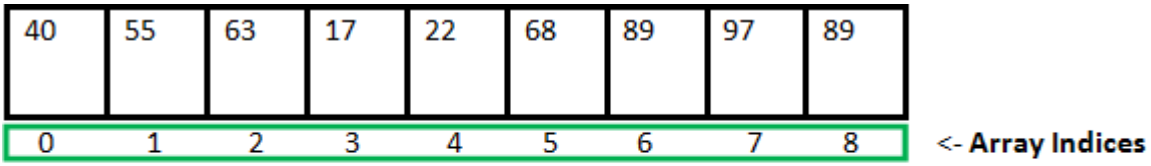
struct node { int data;

struct node* prev_link; struct node* next_link;

};

DYNAMIC MEMEORY ALLOCATION

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() method

“malloc” or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

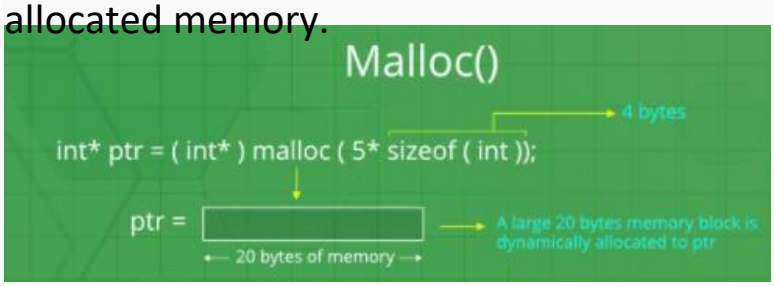
Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



#include <stdio.h> #include <stdlib.h>

int main()

{

// This pointer will hold the

// base address of the block created int* ptr;

int n, i;

// Get the number of elements for the array n = 5;

printf("Enter number of elements: %d\n", n);

```
// Dynamically allocate memory using malloc() ptr = (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not if (ptr == NULL) {
printf("Memory not allocated.\n"); exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully allocated using malloc.\n");

// Get the elements of the array for (i = 0; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array printf("The elements of the array are: "); for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}
```

calloc() method

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

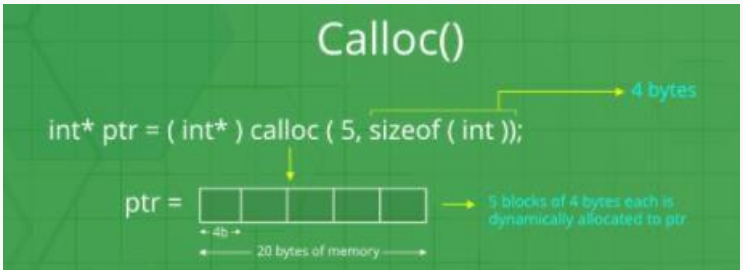
Syntax:

ptr = (cast-type*)calloc(n, element-size);

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



#include <stdio.h> #include <stdlib.h>

int main()

{

// This pointer will hold the

// base address of the block created int* ptr;

int n, i;

// Get the number of elements for the array n = 5;

printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc() ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully

// allocated by calloc or not if (ptr == NULL) {

printf("Memory not allocated.\n");

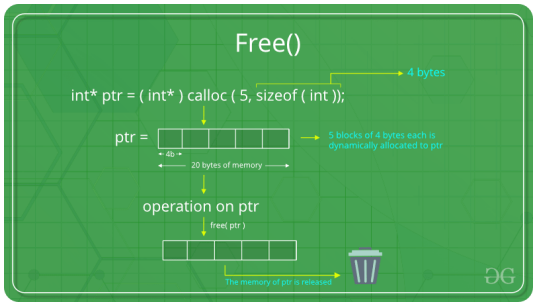
```
exit(0);
}
else {

// Memory has been successfully allocated printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array for (i = 0; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array printf("The elements of the array are: "); for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}
```

free() method

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de- allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



Example:

#include <stdio.h> #include <stdlib.h> int main()

{

// This pointer will hold the

// base address of the block created int *ptr, *ptr1;

int n, i;

// Get the number of elements for the array n = 5;

printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using malloc()

```
ptr = (int*)malloc(n * sizeof(int));



// Dynamically allocate memory using calloc() ptr1 = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) { printf("Memory not allocated.\n"); exit(0);
}
else {



// Memory has been successfully allocated printf("Memory successfully allocated using malloc.\n");

// Free the memory free(ptr);
printf("Malloc Memory successfully freed.\n");
```

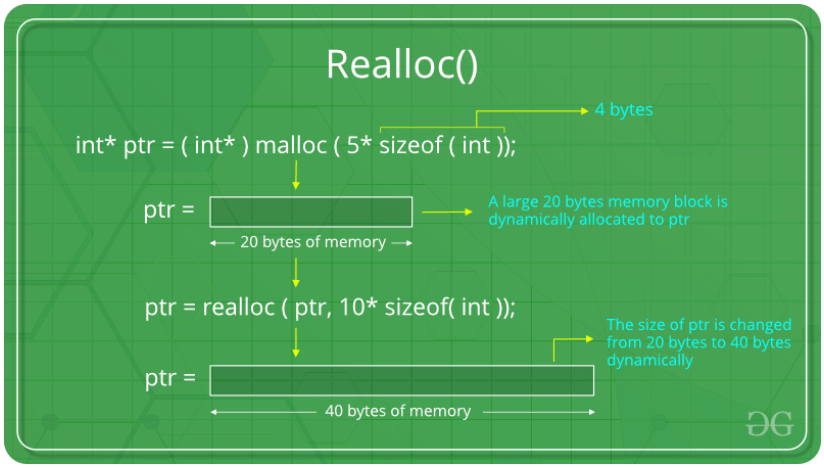
```
// Memory has been successfully allocated printf("\nMemory successfully allocated using calloc.\n");



// Free the memory free(ptr1);

printf("Calloc Memory successfully freed.\n");

}




return 0;

}
```

realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);


where ptr is reallocated with new size 'newSize'
```



If space is insufficient, allocation fails and returns a NULL pointer. #include <stdio.h>

#include <stdlib.h>

int main()

{

// This pointer will hold the

// base address of the block created int* ptr;

int n, i;

// Get the number of elements for the array n = 5;

printf("Enter number of elements: %d\n", n);

```
// Dynamically allocate memory using calloc() ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not if (ptr == NULL) {
printf("Memory not allocated.\n"); exit(0);
}
else {


// Memory has been successfully allocated printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array for (i = 0; i < n; ++i) {
ptr[i] = i + 1;

}
```

```
// Print the elements of the array printf("The elements of the array are: "); for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);

}


// Get the new size for the array n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);


// Dynamically re-allocate memory using realloc() ptr = realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");


// Get the new elements of the array for (i = 5; i < n; ++i) {
ptr[i] = i + 1;

}
```

```
// Print the elements of the array printf("The elements of the array are: "); for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);

}



free(ptr);
}

return 0;
}
```

LINKED LISTS

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*.

Linked list is a data structure which in turn can be used to implement other data structures.

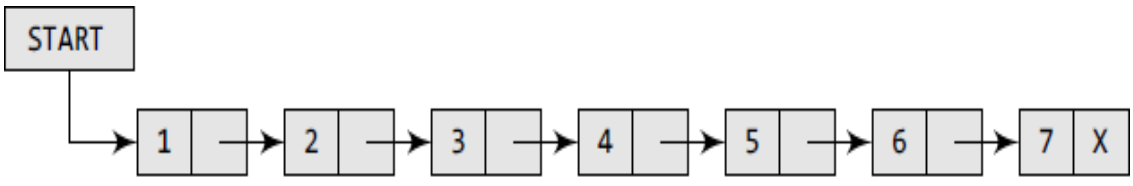


Figure 6.1 Simple linked list

In Fig, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.

The left part of the node which contains data may include a simple data type, an array, or a structure.

The right part of the node contains a pointer to the next node (or address of the next node in sequence).

The last node will have no next node connected to it, so it will store a special value called NULL. In Fig, the NULL pointer is represented by X.

While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.

Linked lists contain a pointer variable START that stores the address of the first node in the list.

We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node.

Using this technique, the individual nodes of the list will form a chain of nodes.

If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code: struct node

{

int data;

struct node *next;

};

Let us see how a linked list is maintained in the memory.

- **In order to form a linked list, we need a structure called *node***

which has two fields, DATA and NEXT.

- **DATA will store the information part and NEXT will store the address of the next node in sequence. Consider Fig. 6.2.**
- **In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START= 1, so the first data is stored at address 1, which is H.**
- **The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.**
- **The second data element obtained from address 4 is E.**
- **Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data.**
- **We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.**

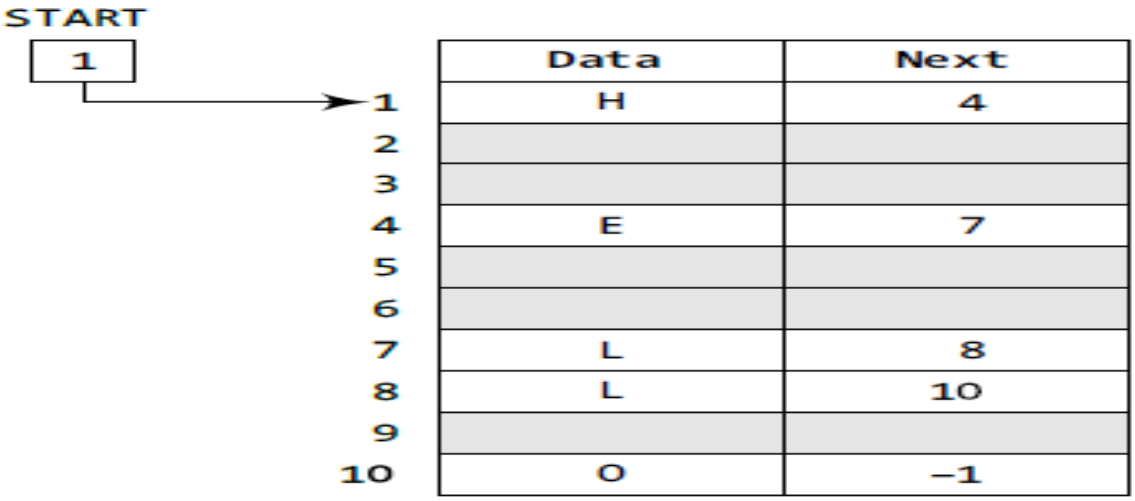


Figure 6.2 START pointing to the first element

Advantages

Linked list have many advantages. Some of the very important advantages are:

- ***Linked Lists are dynamic data structure***: That is, they can grow or shrink during the execution of a program.
- ***Efficient memory utilization:*** Here, memory is not pre- allocated. Memory is allocated whenever it is required. And it is deallocated when it is no longer needed.
- ***Insertion and deletions are easier and efficient:*** Linked lists provide flexibility in inserting data item at a specified position and deletion of a data item from the given position.
- ***Many complex applications can be easily carried out with linked lists.***

Disadvantages

- ***More Memory***: If the numbers of fields are more, then more memory space is needed.
- Access to an arbitrary data item is little bit cumbersome and also time consuming.

<u>Types of Linked List</u>

Following are the various flavours of linked list.

- Simple Linked List − Item Navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward way.
- Circular Linked List − Last item contains link of the first element as next and and first element has link to last element as prev.

Basic Operations

- Insertion − add an element at the beginning of the list.
- Display − displaying complete list.
- Search − search an element using given key.
- Delete − delete an element using given key

SINGLY LINKED Lists

- **A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.**
- **A singly linked list allows traversal of data only in one way. Figure 6.7 shows a singly linked list**

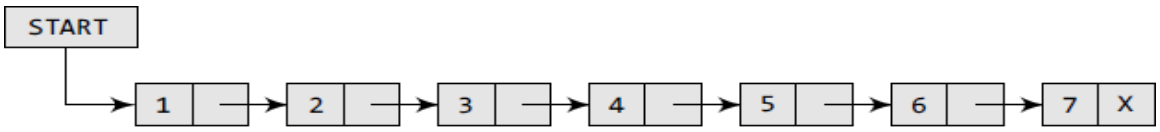


Figure 6.7 Singly linked list

LINKED LIST OPERATIONS

Traversing a Linked List

- **Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.**
- **a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node.**
- **For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.**
- **The algorithm to traverse a linked list is shown in Fig. 6.8.**

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR->DATA
Step 4:            SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: EXIT
```

Figure 6.8 Algorithm for traversing a linked list

- **In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list.**

- **Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL.**
- **In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR.**
- **In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.**

Searching for a Value in a Linked List

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR->DATA
                  SET POS = PTR
                  Go To Step 5
             ELSE
                  SET PTR = PTR->NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Figure 6.10 Algorithm to search a linked list

Consider the linked list shown in Fig. 6.11. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

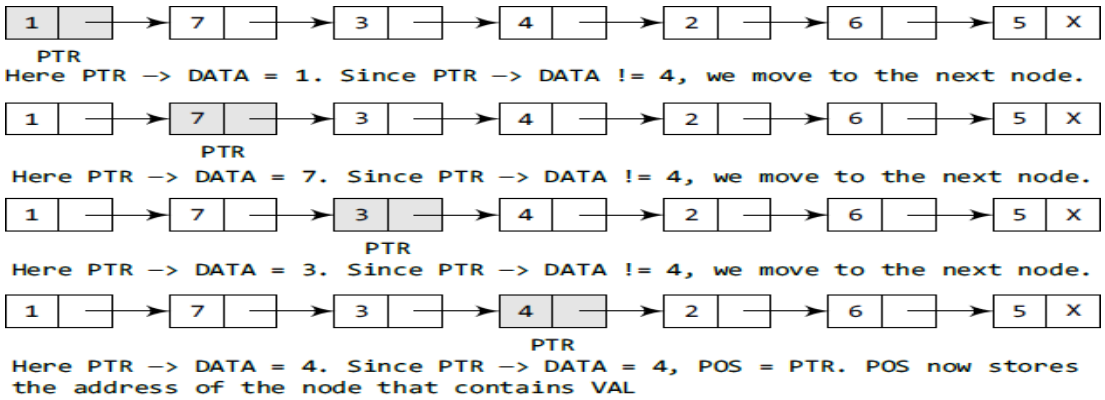


Figure 6.11 Searching a linked list

Steps to create a linked list

Step 1 : Include alloc.h Header File #include<alloc.h>

1. We don‟t know, how many nodes user is going to create once he execute the program.
2. In this case we are going to allocate memory using Dynamic Memory Allocation functions malloc.
3. Dynamic memory allocation functions are included in alloc.h

Step 2 : Define Node Structure

We are now defining the new global node which can be accessible through any of the function.

struct node

{ **int data;**

struct node *next;

}*start=NULL;

Step 3 : Create Node using Dynamic Memory Allocation .Now we are creating one node dynamically using malloc function.We don‟t have prior knowledge about number of nodes , so we are calling malloc function to create node at run time.

new_node=(struct node *)malloc(sizeof(struct node));

Fill Information in newly Created Node ,Now we are accepting value from the user using scanf. Accepted Integer value is stored in the data field. Whenever we create new node , Make its Next Field

as NULL. printf("Enter the data : "); scanf("%d",&new_node-

>data);

Step 4 : if(start==NULL) then new_node -> next = NULL; start = new_node;

otherwise ptr=start; while(ptr->next!=NULL) ptr=ptr->next;

ptr->next = new_node; new_node->next=NULL;

step 5:continue this process till while(num!=-1)

Inserting a New Node in a Linked List

- **Case 1: The new node is inserted at the beginning.**
- **Case 2: The new node is inserted at the end.**
- **Case 3: The new node is inserted after a given node.**

CASE 1: Inserting a Node at the Beginning of a Linked List

- **Consider the linked list shown in Fig. 6.12. Suppose we want to add a new node with data 9 and**
- **add it as the first node of the list. Then the following changes will be done in the linked list.**

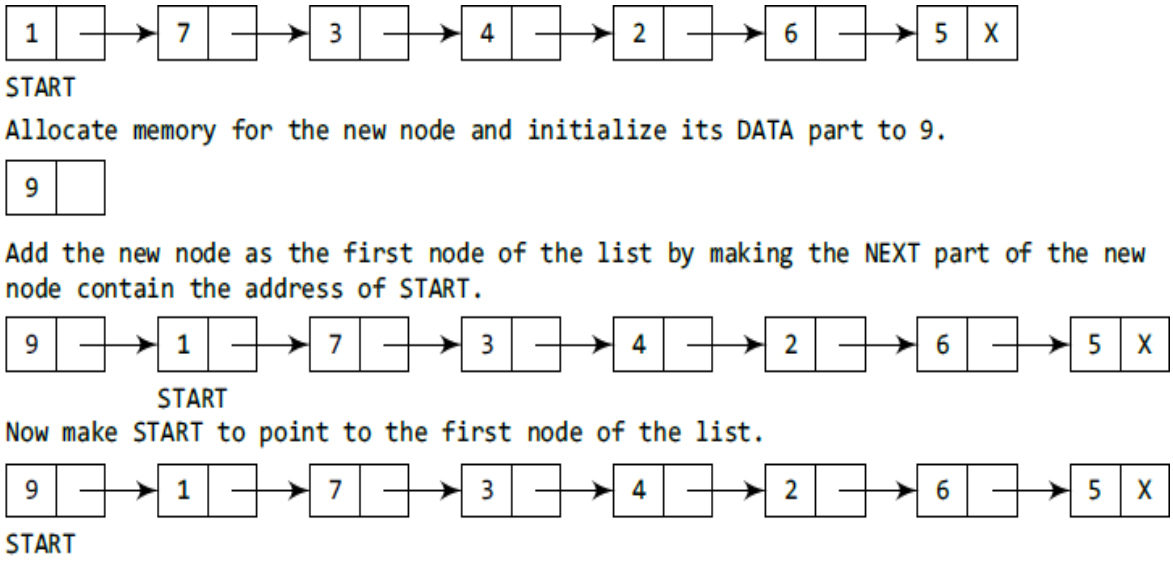


Figure 6.12 Inserting an element at the beginning of a linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Figure 6.13 Algorithm to insert a new node at the beginning

- **In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.**
- **Otherwise, if a free memory cell is available, then we allocate space for the new node.**
- **Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START.**

- **Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE.**

Note the following two steps:

- **Step 2: SET NEW_NODE = AVAIL**
- **Step 3: SET AVAIL = AVAIL -> NEXT**
- **These steps allocate memory for the new node.**

Program

```
node *insert_beg(node *start)
{
node *new_node; int num;
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (node *)malloc(sizeof(node)); new_node -> data = num;
new_node -> next = start; start = new_node;
return start;
}
```

CASE 2: Inserting a Node at the End of a Linked List

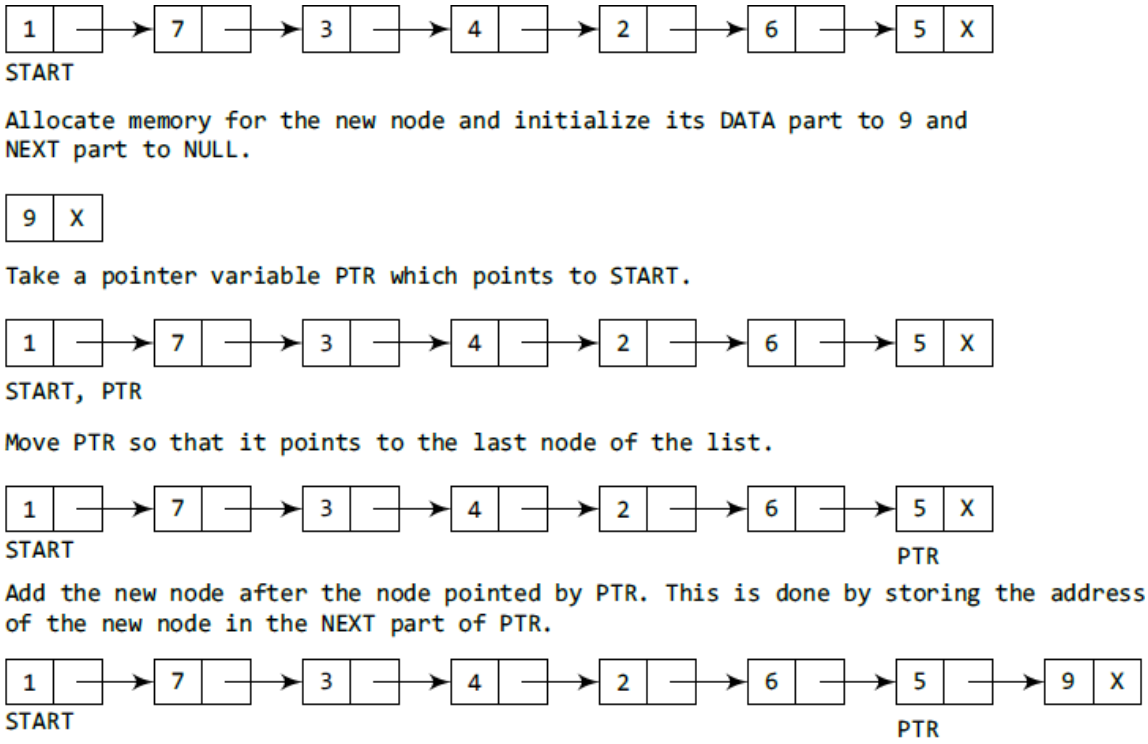


Figure 6.14 Inserting an element at the end of a linked list

- Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

```
Step 1:  IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
         [END OF IF]
Step 2:  SET NEW_NODE = AVAIL
Step 3:  SET AVAIL = AVAIL -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET NEW_NODE -> NEXT = NULL
Step 6:  SET PTR = START
Step 7:  Repeat Step 8 while PTR -> NEXT != NULL
Step 8:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 9:  SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.15 Algorithm to insert a new node at the end

- In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.

- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.

program

```
struct node *insert_end(struct node *start)
{
struct node *ptr, *new_node; int num;
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (node *)malloc(sizeof(node)); new_node -> data = num;
new_node -> next = NULL; ptr = start;
while(ptr -> next != NULL) ptr = ptr -> next;
ptr -> next = new_node; return start;
}
```

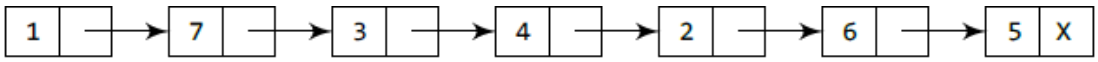
- ***CASE 3: Inserting a Node After a Given Node in a Linked List***
- Consider the linked list shown in Fig. 6.17. Suppose we want to add a new node with value 9 after the node containing data 3

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR->DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR->NEXT
          [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

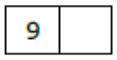
Figure 6.16 Algorithm to insert a new node after a node that has value NUM

- In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR.
- Initially, PREPTR is initialized to PTR.
- So now, PTR,PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.

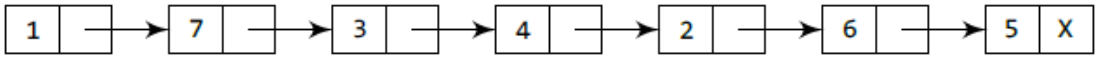
- Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.



START
Allocate memory for the new node and initialize its DATA part to 9.

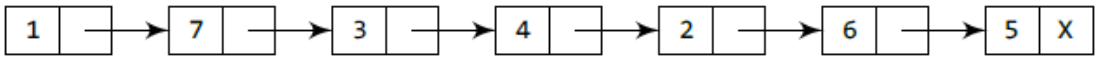


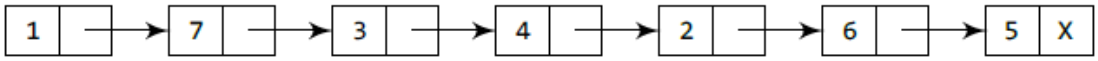
Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.



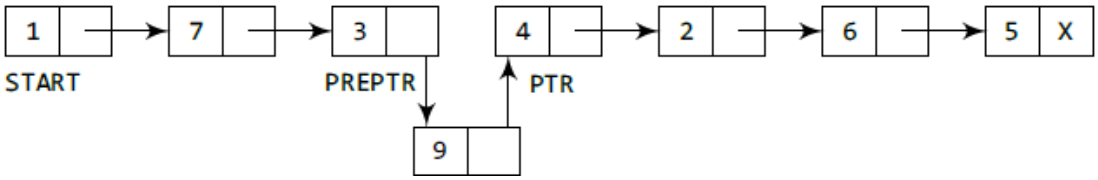
START
PTR
PREPTR

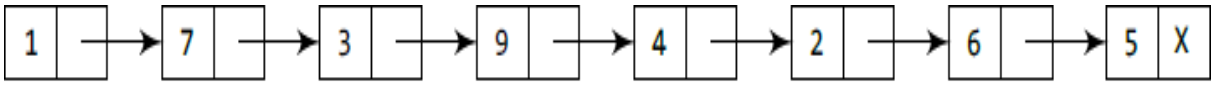
Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.





Add the new node in between the nodes pointed by PREPTR and PTR.





Program

```
node *insert_after(node *start)
{
```

```
node *new_node, *ptr, *preptr; int num, val;
printf("\n Enter the data : "); scanf("%d", &num);
printf("\n Enter the value after which the data has to be inserted : "); scanf("%d", &val);
new_node = (node *)malloc(sizeof(node)); new_node -> data = num;
ptr = start; preptr = ptr;
while(preptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next=new_node; new_node -> next = ptr; return start;
}
```

Deleting a Node from a Linked List

- We will consider three cases and then see how deletion is done in each case.
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.

CASE 1:Deleting the First Node from a Linked List

- Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW.
- Underflow is a condition that occurs when we try to delete a node from a linked list that is empty.
- This happens when START = NULL or when there are no more nodes to delete.
- Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node.
- The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

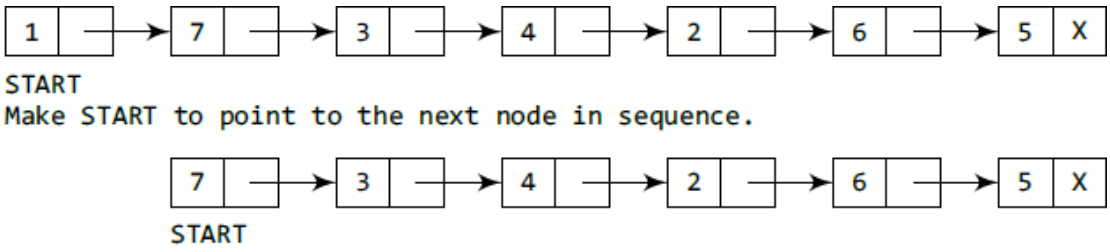


Figure 6.20 Deleting the first node of a linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Figure 6.21 Algorithm to delete the first node

- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Program

```
struct node *delete_beg(node *start)
{
struct node *ptr; ptr = start;
start = start -> next;
```

free(ptr); return start;

}

CASE2: Deleting the Last Node from a Linked List

- Consider the linked list shown in Fig. 6.22. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

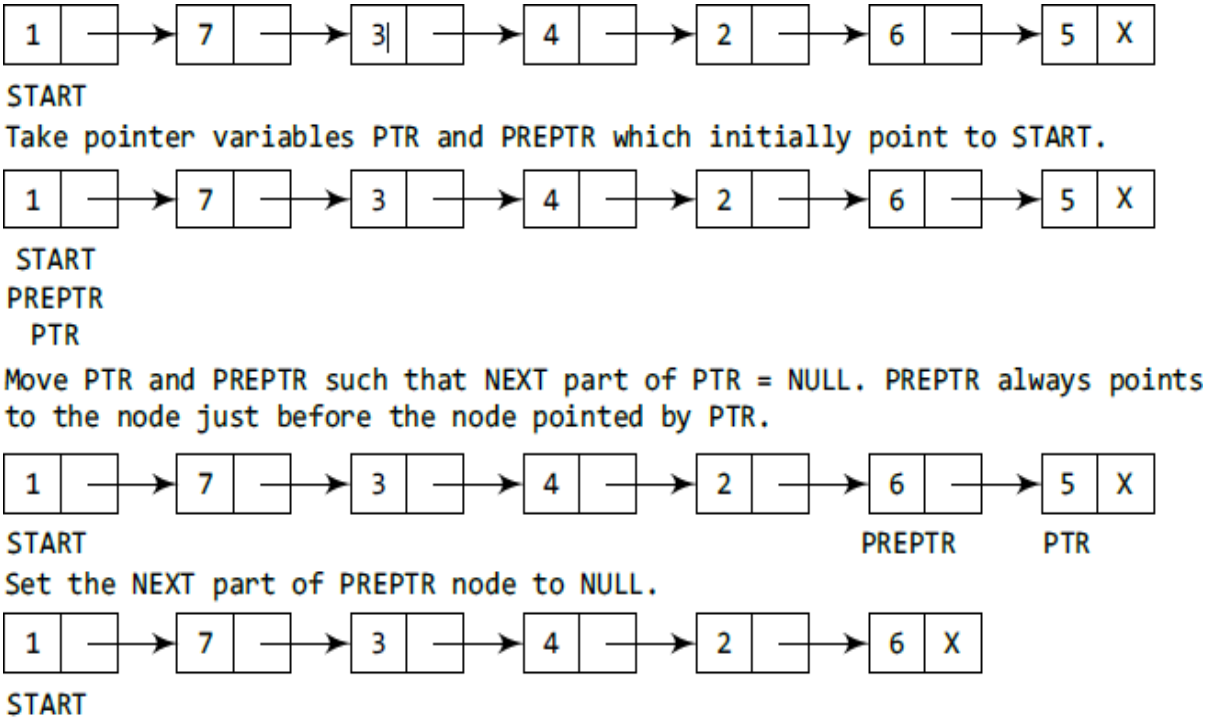


Figure 6.22 Deleting the last node of a linked list

- Figure 6.23 shows the algorithm to delete the last node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START.

- That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned back to the free pool

---

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.23 Algorithm to delete the last node

Program

```
node *delete_end(node *start)
{
node *ptr, *preptr; ptr = start;
while(ptr -> next != NULL)
```

{

preptr = ptr;

ptr = ptr -> next;

}preptr -> next = NULL; free(ptr);

return start;}

CASE 3:Deleting the Node After a Given Node in a Linked List

- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the node containing VAL and the node
- succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it.
- The memory of the node succeeding the given node is freed and returned back to the free pool.

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Figure 6.25 Algorithm to delete the node after a given node

- Consider the linked list shown in Fig. 6.24.
- Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list

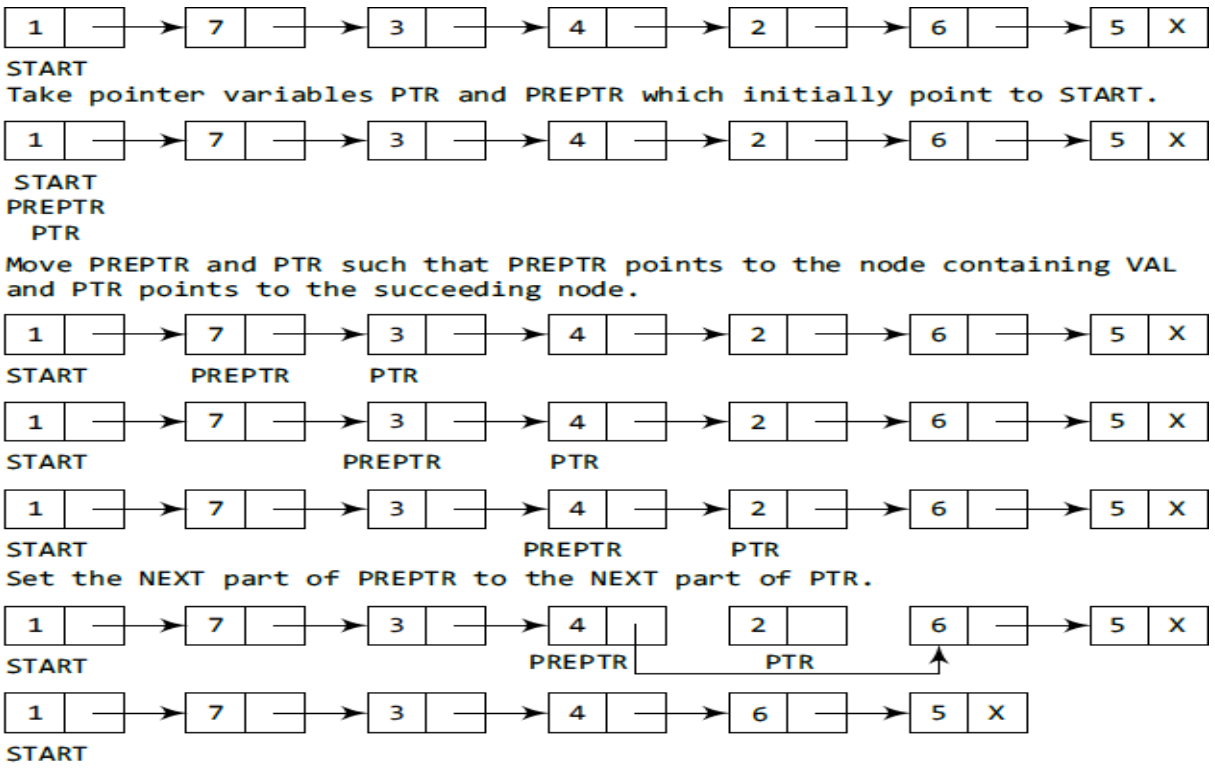


Figure 6.24 Deleting the node after a given node in a linked list

-

Program

```
node *delete_after(node *start)
{
node *ptr, *preptr; int val;
printf("\n Enter the value after which the node has to deleted : ");
```

```
scanf("%d", &val); ptr = start;
preptr = ptr;
while(preptr -> data != val)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next=ptr -> next; free(ptr);
return start;
}
```

CIRCULAR LINKED LISTs

- In a circular linked list, the last node contains a pointer to the first node of the list.
- We can have a **circular singly linked list** as well as a **circular doubly linked list**.
- While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.
- Thus, a circular linked list has no beginning and no ending. Figure

6.26 shows a circular linked list.

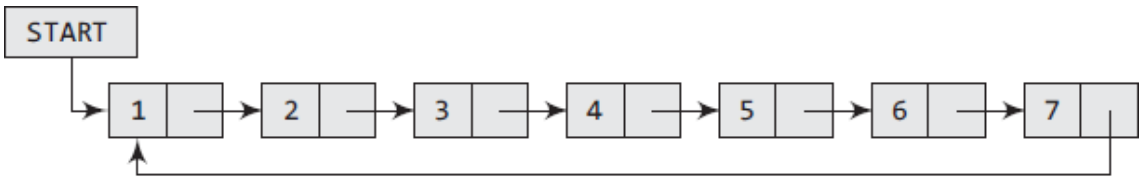


Figure 6.26 Circular linked list

Inserting a New Node in a Circular Linked List

- Case 1: The new node is inserted at the beginning of the circular linked list.
- Case 2: The new node is inserted at the end of the circular linked list.

CASE 1:Inserting a Node at the Beginning of a Circular Linked List

- Consider the linked list shown in Fig. 6.29. Suppose we want to add a new node with data 9 as the first node of the list.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Figure 6.30 Algorithm to insert a new node at the beginning

- Figure 6.30 shows the algorithm to insert a new node at the beginning of a linked list. In Step 1, we first check whether memory

is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed.

- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node,that is, the START pointer variable will now hold the address of the NEW_NODE.
- While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list.
- Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START

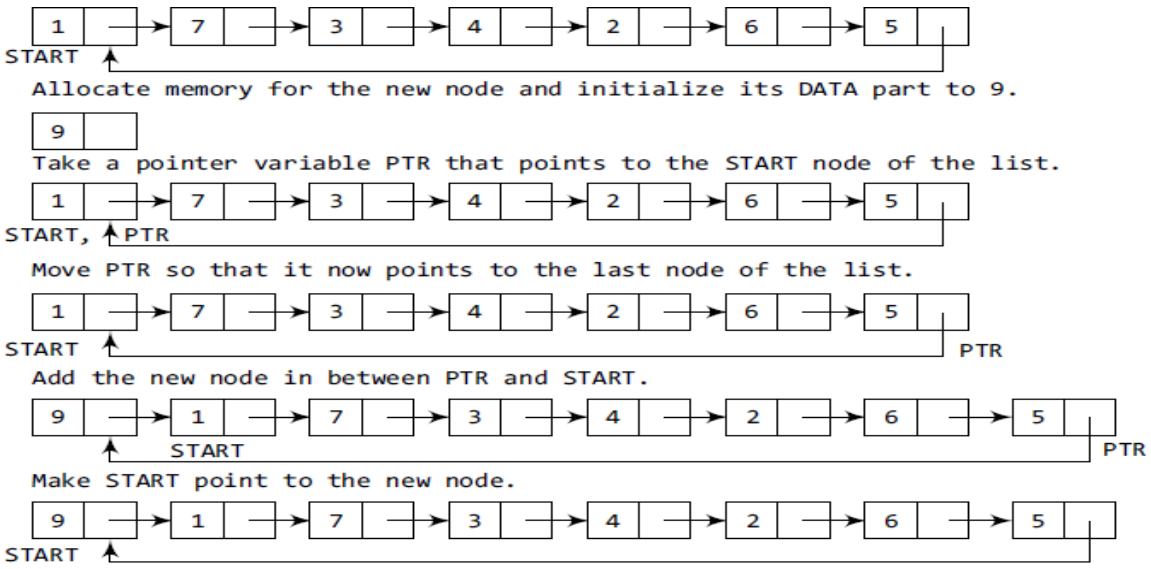


Figure 6.29 Inserting a new node at the beginning of a circular linked list

Program insert new node at beginning

```
struct node *insert_beg(struct node *start)
{
struct node *new_node, *ptr; int num;
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
ptr = start;
while(ptr -> next != start) ptr = ptr -> next;
ptr -> next = new_node; new_node -> next = start; start = new_node;
return start;
}
```

CASE 2: Inserting a Node at the End of a Circular Linked List

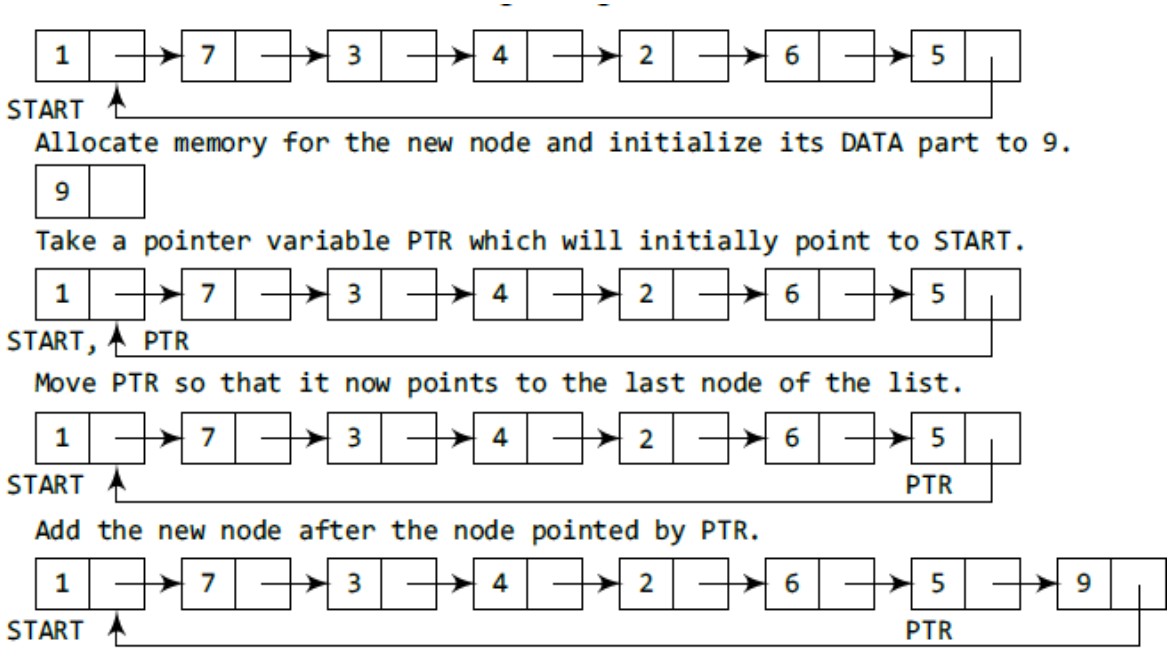


Figure 6.31 Inserting a new node at the end of a circular linked list

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure 6.32 Algorithm to insert a new node at the end

- Figure 6.32 shows the algorithm to insert a new node at the end of a circular linked list.
- In Step 6, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

Program

```
struct node *insert_end(struct node *start)
```

```
{
struct node *ptr, *new_node; int num;
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
ptr = start;
while(ptr -> next != start) ptr = ptr -> next;
ptr -> next = new_node; new_node -> next = start; return start;}
```

Deleting a Node from a Circular Linked List

- Case 1: The first node is deleted.
- Case 2: The last node is deleted**.**

CASE 1: Deleting the First Node from a Circular Linked List

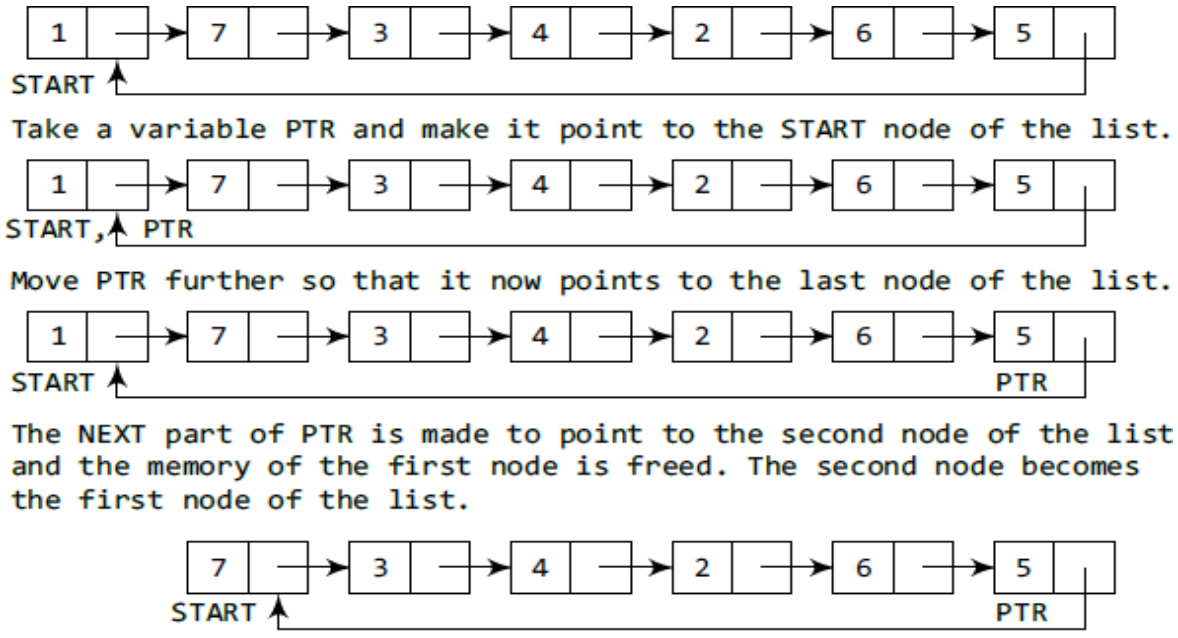


Figure 6.33 Deleting the first node from a circular linked list

- In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list.
- In Step 6, the memory occupied by the first node is freed.
- Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:         SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

Figure 6.34 Algorithm to delete the first node

Program

struct node *delete_beg(struct node *start)

{

struct node *ptr;

ptr = start;

while(ptr -> next != start) ptr = ptr -> next;

ptr -> next = start -> next; free(start);

start = ptr -> next; return start;

}

CASE 2: Deleting the Last Node from a Circular Linked List

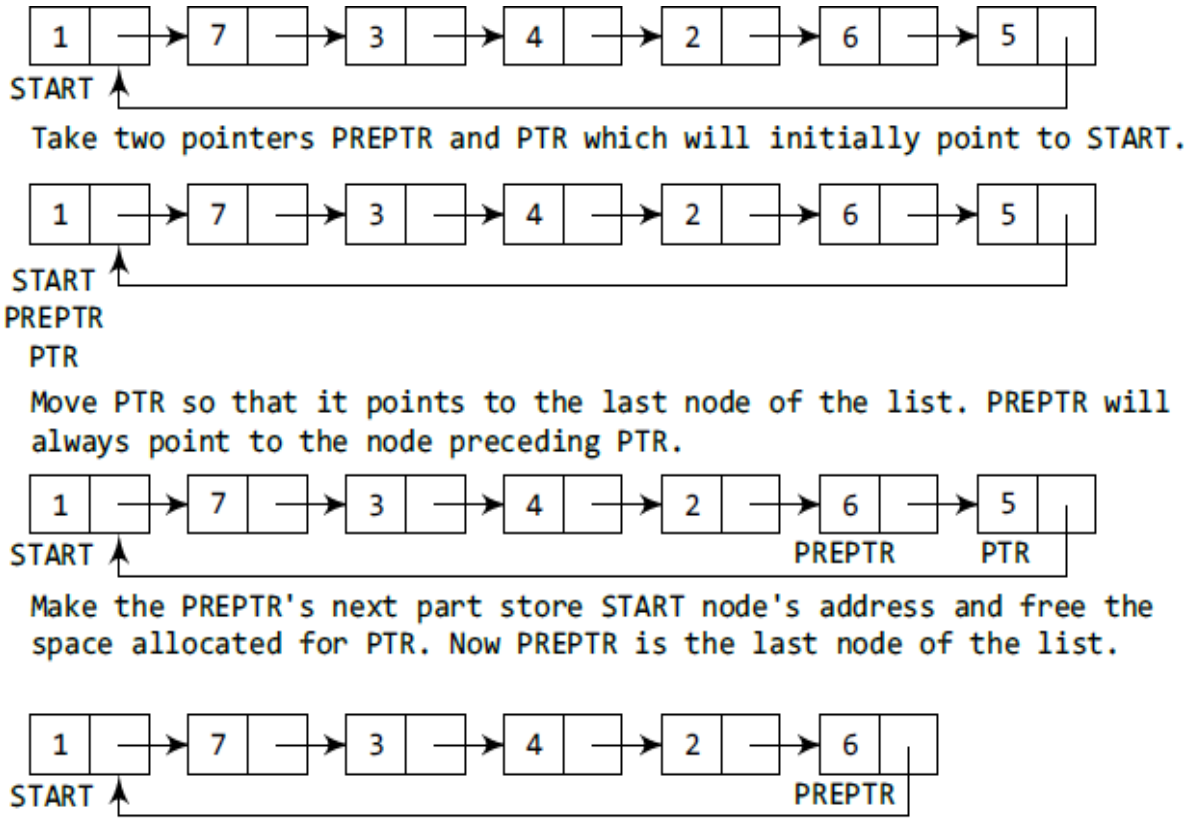


Figure 6.35 Deleting the last node from a circular linked list

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is,PTR now points to the first node of the linked list.

- In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.
- Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

program

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

Figure 6.36 Algorithm to delete the last node

program

```
struct node *delete_end(struct node *start)
{
```

```
struct node *ptr, *preptr; ptr = start;
while(ptr -> next != start)
{
preptr = ptr;
ptr = ptr -> next;
}
preptr -> next = ptr -> next; free(ptr);
return start;}
```

DOUBLY LINKED LISTS

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node .

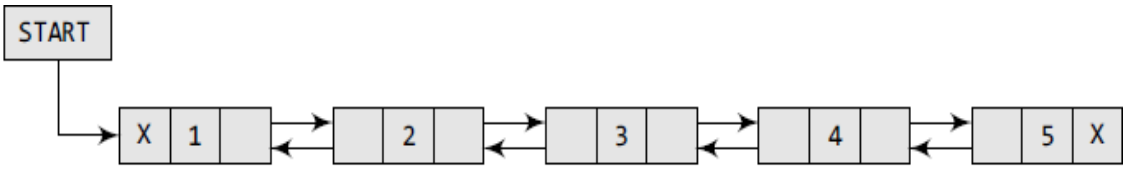


Figure 6.37 Doubly linked list

ı C, the structure of a doubly linked list can be given as,

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- Thus, we see that a doubly linked list calls for more space per node and more expensive basic operations.
- However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Inserting a New Node in a Doubly Linked List

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.

CASE 1: Inserting a Node at the Beginning of a Doubly Linked List

=

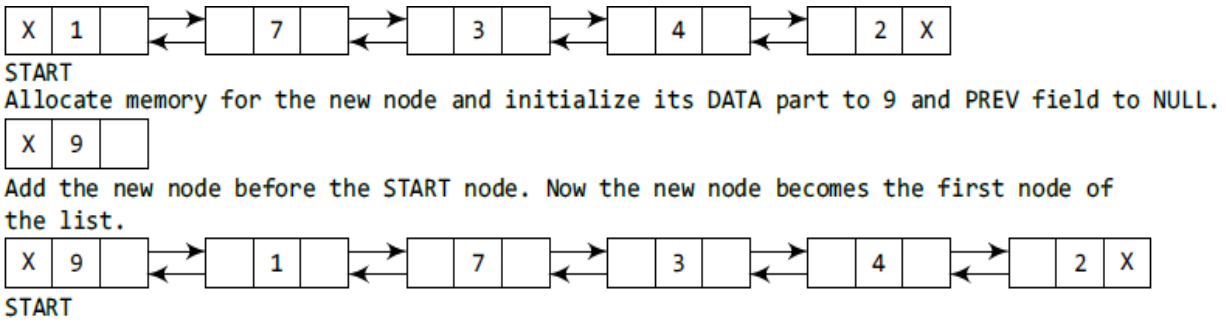


Figure 6.39 Inserting a new node at the beginning of a doubly linked list

- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Figure 6.40 Algorithm to insert a new node at the beginning

program

```
struct node *insert_beg(struct node *start)
{
struct node *new_node;
```

```
int num;
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
start -> prev = new_node;
new_node -> next = start;
new_node -> prev = NULL; start = new_node;
return start;
}
```

CASE 2: Inserting a Node at the End end of a Doubly Linked List

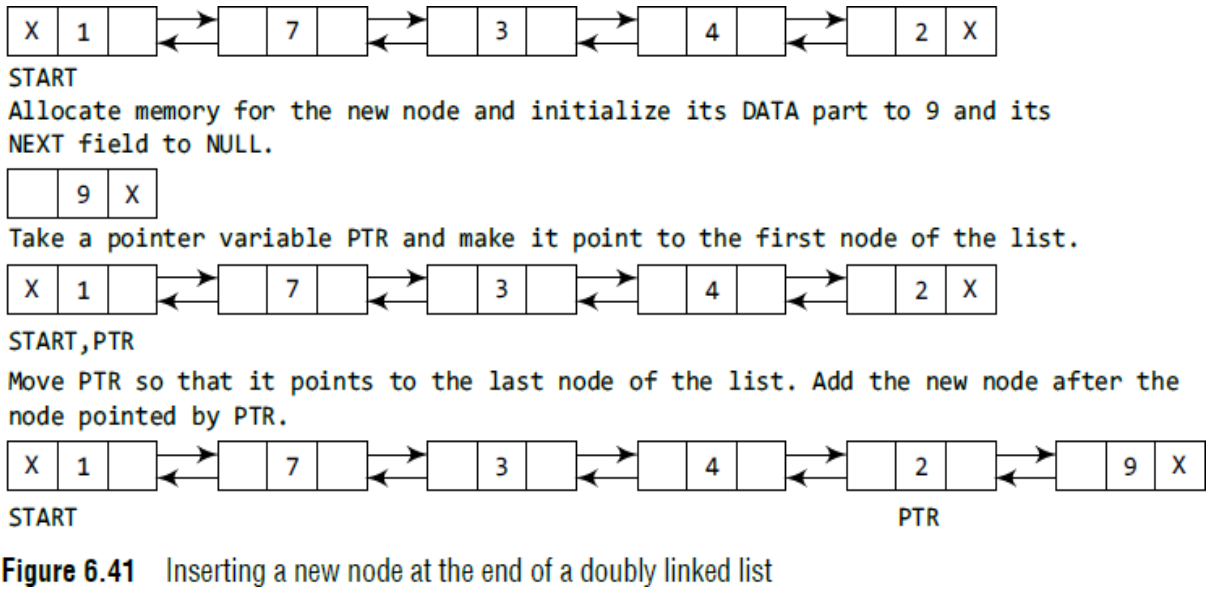


Figure 6.41 Inserting a new node at the end of a doubly linked list

- Figure 6.42 shows the algorithm to insert a new node at the end of a doubly linked list. In Step 6, we take a pointer variable PTR and initialize it with START.

- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the
- new node contains NULL which signifies the end of the linked list.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Figure 6.42 Algorithm to insert a new node at the end

Program

```
struct node *insert_end(struct node *start)
{
struct node *ptr, *new_node; int num;
```

```
printf("\n Enter the data : "); scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
ptr=start;
while(ptr -> next != NULL) ptr = ptr -> next;
ptr -> next = new_node; new_node -> prev = ptr; new_node -> next = NULL; return start;
}
```

CASE 3: Inserting a Node After a Given Node in a Doubly Linked List

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 6.43 Algorithm to insert a new node after a given node

Figure 6.43 shows the algorithm to insert a new node after a given node in a doubly linked list.

In Step 5, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.

We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

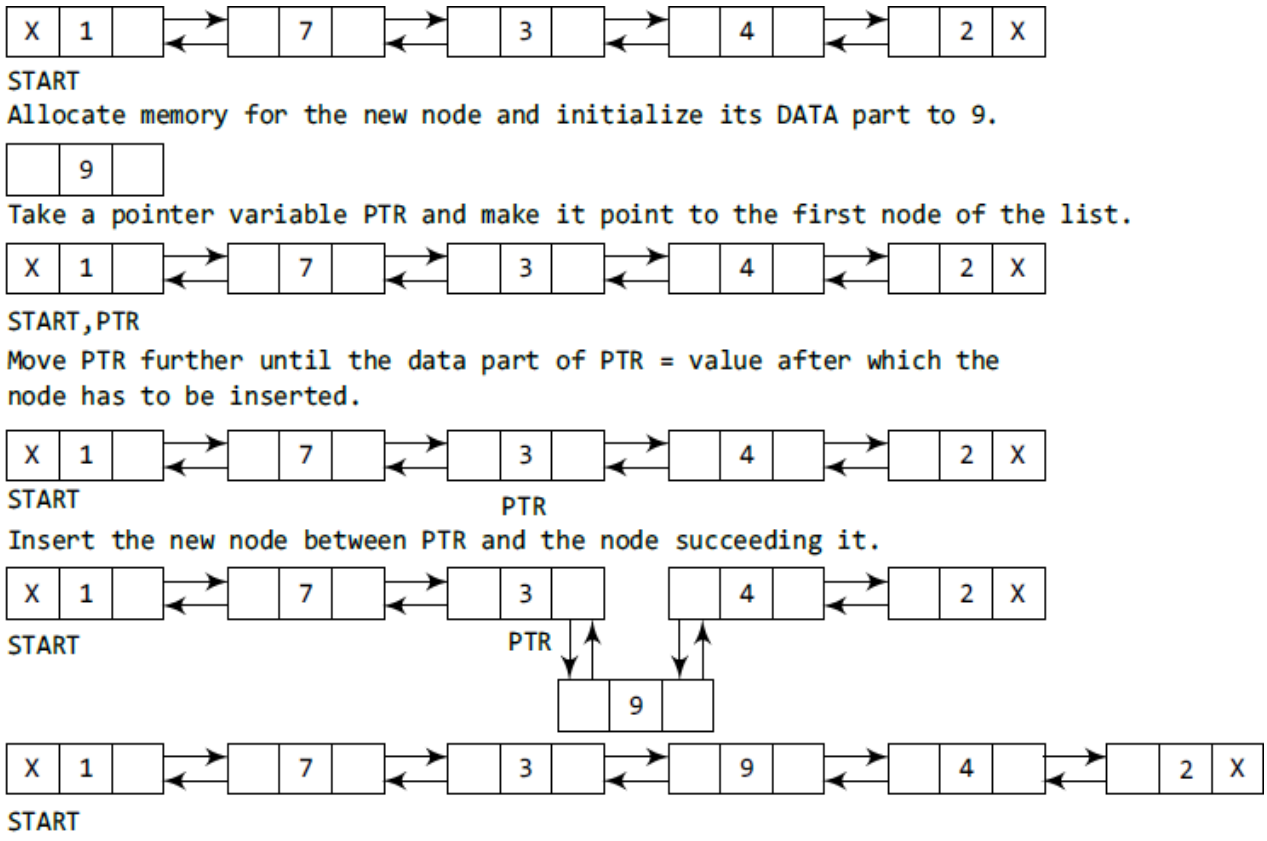


Figure 6.44 Inserting a new node after a given node in a doubly linked list

Program

```
struct node *insert_after(struct node *start)
{
```

```
struct node *new_node, *ptr; int num, val;
printf("\n Enter the data : "); scanf("%d", &num);
printf("\n Enter the value after which the data has to be inserted : "); scanf("%d", &val);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
ptr = start;
while(ptr -> data != val) ptr = ptr -> next; new_node -> prev = ptr;
new_node -> next = ptr -> next; ptr -> next -> prev = new_node; ptr -> next = new_node;
return start;
}
```

Deleting a Node from a Doubly Linked List

- **Case 1: The first node is deleted.**
- **Case 2: The last node is deleted.**
- **Case 3: The node after a given node is deleted.**

CASE 1: Deleting the First Node from a Doubly Linked List

- When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

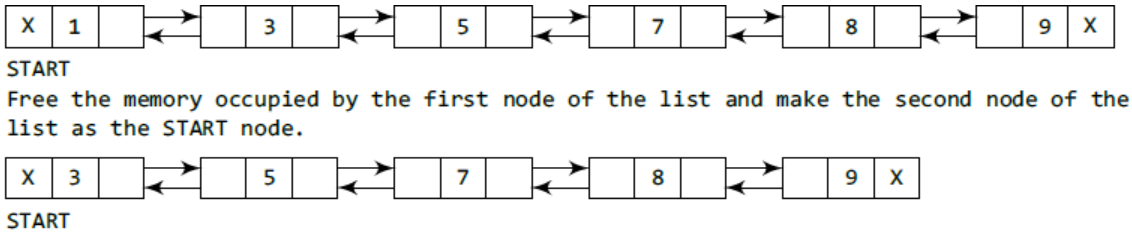


Figure 6.47 Deleting the first node from a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Figure 6.48 Algorithm to delete the first node

- Figure 6.48 shows the algorithm to delete the first node of a doubly linked list.
- In Step 1 of the algorithm, we check if the linked list exists or not. If START =NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list.
- For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

Program

```
struct node *delete_beg(struct node *start)
{
struct node *ptr; ptr = start;
start = start -> next;
```

start -> prev = NULL; free(ptr);

return start;

}

CASE 2: Deleting the Last Node from a Doubly Linked List

- Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

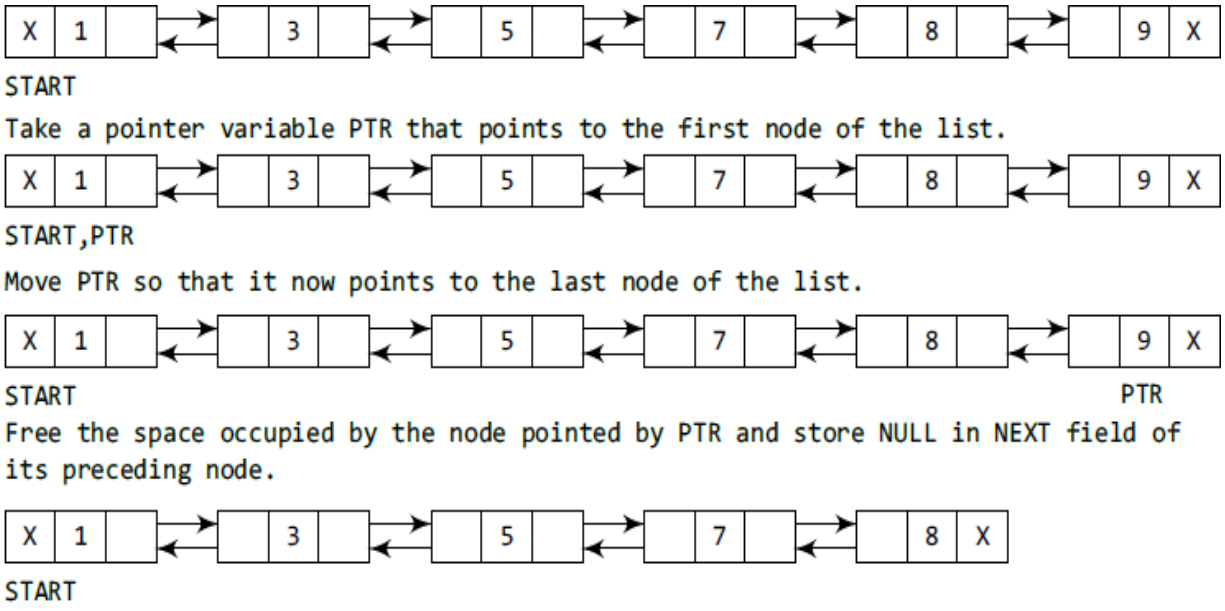


Figure 6.49 Deleting the last node from a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Figure 6.50 Algorithm to delete the last node

- Figure 6.50 shows the algorithm to delete the last node of a doubly linked list.

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list.
- The memory of the previous last node is freed and returned to the free pool.

Program

```
struct node *delete_end(struct node *start)
{
struct node *ptr; ptr = start;
while(ptr -> next != NULL) ptr = ptr -> next;
ptr -> prev -> next = NULL; free(ptr);
return start;
}
```

CASE 3 : Deleting the Node After a Given Node in a Doubly Linked List

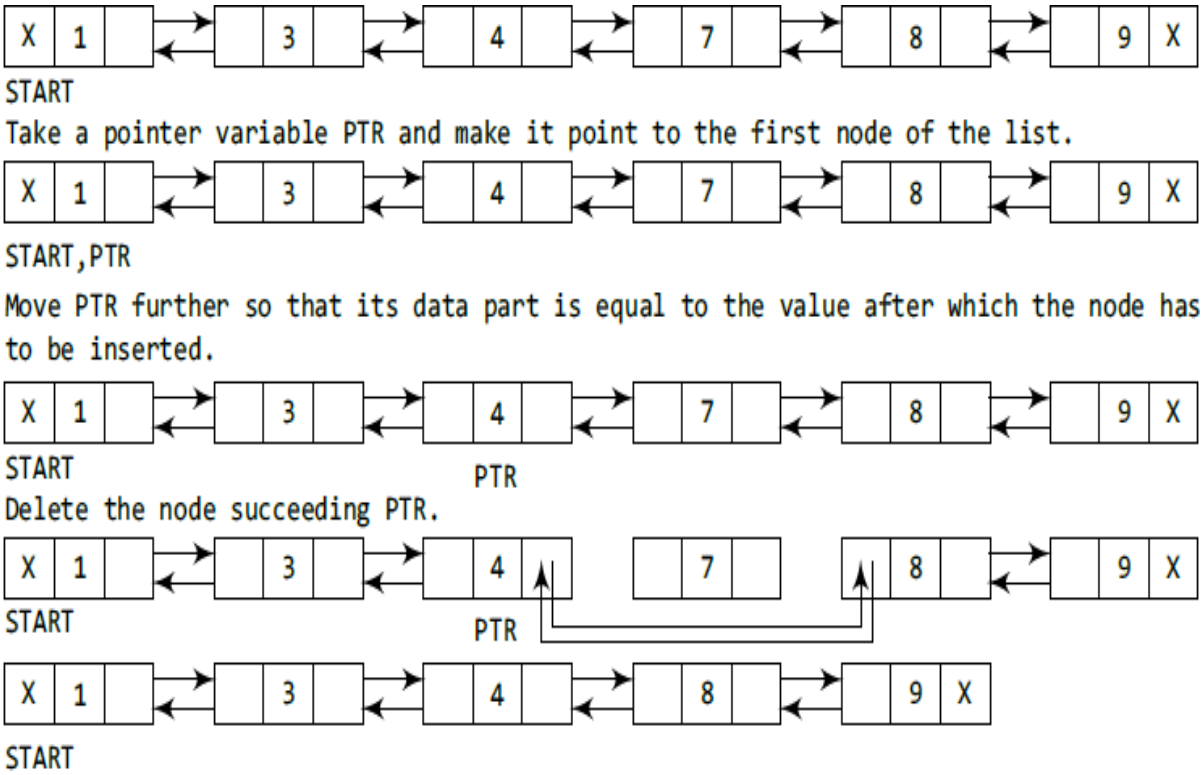


Figure 6.51 Deleting the node after a given node in a doubly linked list

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Figure 6.52 Algorithm to delete a node after a given node

In Step 2, we take a pointer variable PTR and initialize it with

START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node.

- Once we reach the node containing VAL, the node
- succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node.
- Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

Program

```
struct node *delete_after(struct node *start)
{
struct node *ptr, *temp; int val;
printf("\n Enter the value after which the node has to deleted : "); scanf("%d", &val);
ptr = start;
while(ptr -> data != val) ptr = ptr -> next;
temp = ptr -> next;
ptr -> next = temp -> next; temp -> next -> prev = ptr; free(temp);
return start;
}
```

Header Linked Lists

- A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, in a header linked list, START will not point to the first node of the list but START will contain the address of the header node.
- The following are the two variants of a header linked list:
- *Grounded header linked list* which stores NULL in the next field of the last node.
- *Circular header linked list* which stores the address of the header node in the next field of the last node. Here, the header node will denote the end of the list.

Look at Fig. 6.65 which shows both the types of header linked lists.

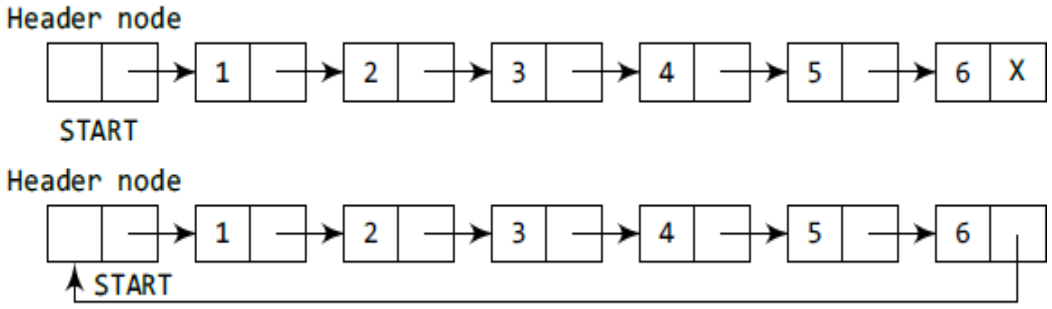


Figure 6.65 Header linked list

Application of linked list-Polynomial

Polynomial Addition

Linked list are widely used to represent and manipulate polynomials. Polynomials are the expressions containing number of terms with nonzero coefficient and exponents.In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields

- Coefficient field

. Exponent field

- Link field

The coefficient field holds the value of the coefficient of a term and the exponent field contains the exponent value of the term. And the link field contains the address of the next term in the polynomial. The polynomial node structure is

Coefficient(coeff)	**Exponent(expo)**	**Address of the next node(next)**

Algorithm

Two polynomials can be added. And the steps involved in adding two polynomials are given below

1. Read the number of terms in the first polynomial P
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial Q
4. Read the coefficient and exponent of the second polynomial
5. Set the temporary pointers p and q to travers the two polynomials respectively
6. Compare the exponents of two polynomials starting from the first nodes

a) If both exponents are equal then add the coefficient and store it in the resultant linked list

b) If the exponent of the current term in the first polynomial P is less than the exponent of the current term of the second polynomial then added the second term to the resultant linked list. And, move the pointer q to point to the next node in the second polynomial Q.

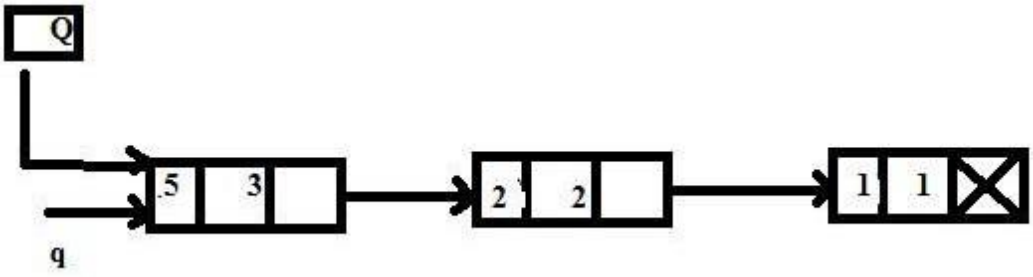
c) If the exponent of the current term in the first polynomial P is greater than the exponent of the current term in the second polynomial Q, then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.

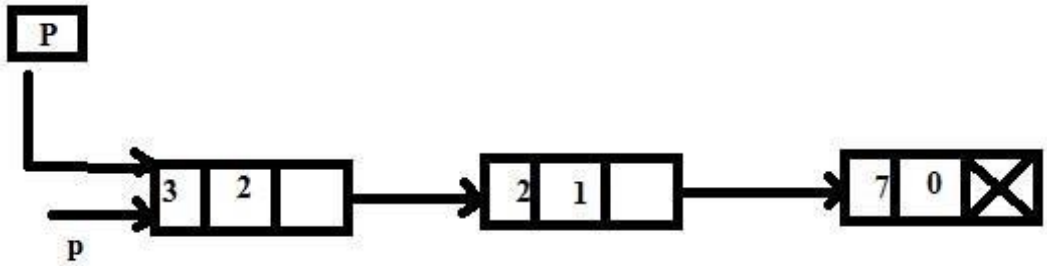
d) Append the remaining nodes of either of the polynomials to the resultant linked list.

Let us illustrate the way the two polynomials are added. Let p and q be two polynomials having three terms each.

$P=3x2+2x+7$ $Q=5x^3+2x^2+x$

These two polynomial can be represented as





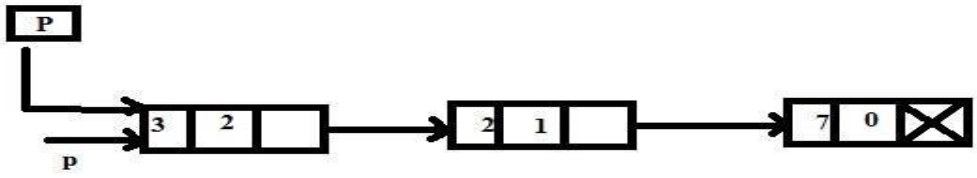
Step 1. Compare the exponent of p and the corresponding

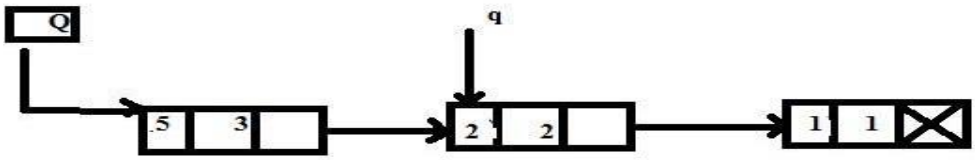
exponent of q. Here,

expo(p)<expo(q)

So, add the terms pointed to by q to the resultant list. And now advance the q pointer.

Step 2.

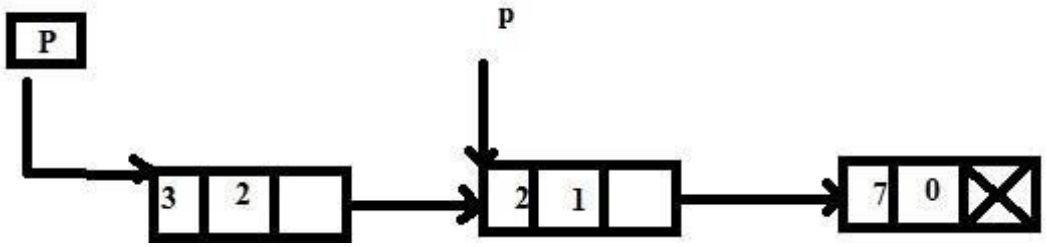


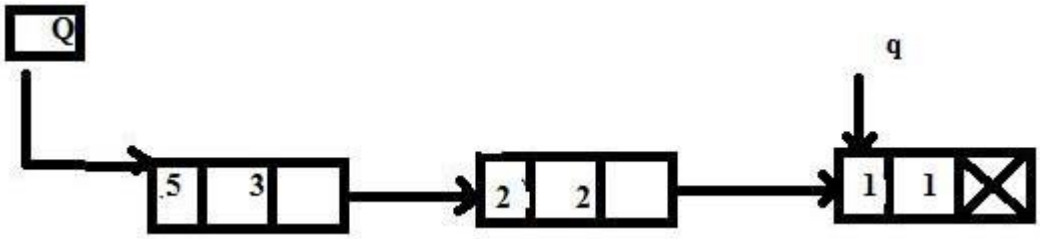

Q
q
5
3
2
2
1
1

Compare the exponent of the current terms. Here,

expo(p)=expo(q)

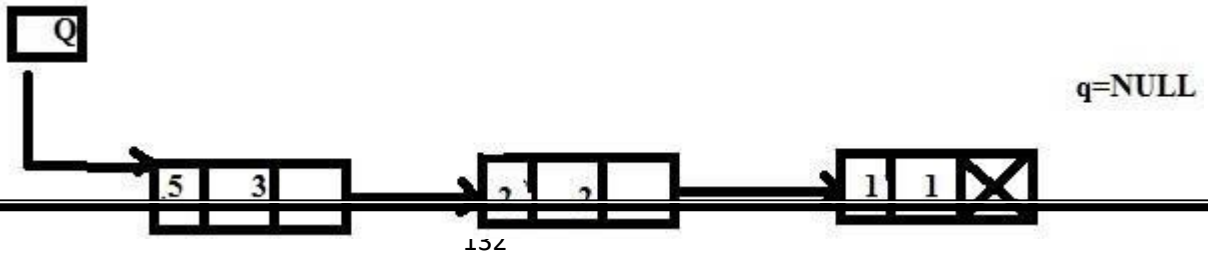
So, add the coefficients of these two terms and link this to the resultant list. And, advance the pointers p and q to their next nodes.



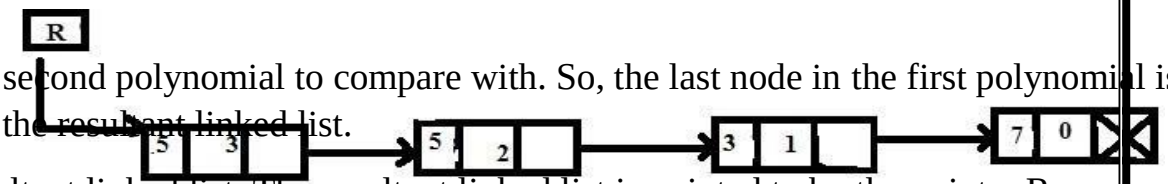


Compare the exponents of the current terms again expo(p)=expo(q)

So, add the coefficients of these two terms and link this to the resultant linked list. And, advance the pointers to their next nodes. Q reaches the NULL and p points the last node.



There is no node in the second polynomial to compare with. So, the last node in the first polynomial is added to the end of the resultant linked list.



Step 5. Display the resultant linked list. The resultant linked list is pointed to by the pointer R

Algorithm for Polynomial Multiplication

1. Read the number of terms in the first polynomial
2. Read the coefficient and exponent of the first polynomial
3. Read the number of terms in the second polynomial

4. Read the coefficient and exponent of the second polynomial
5. if one of the list is empty then the nonempty linked list is added to the resultant linked list Otherwise goto step 6.
6. for each term of the first list

a) multiply each term of the second linked list with a term of the first linked list

b) add the new term to the resultant polynomial

c) reposition the pointer to the starting of the second linked list

d) go to the next node

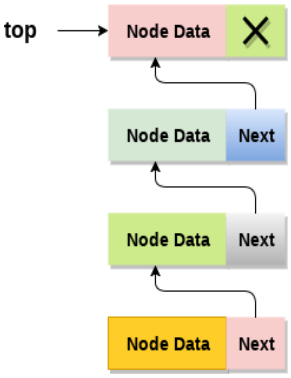
e) adds a term to the polynomial in the descending order of the exponent

7. Display the resultant linked list

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non- contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

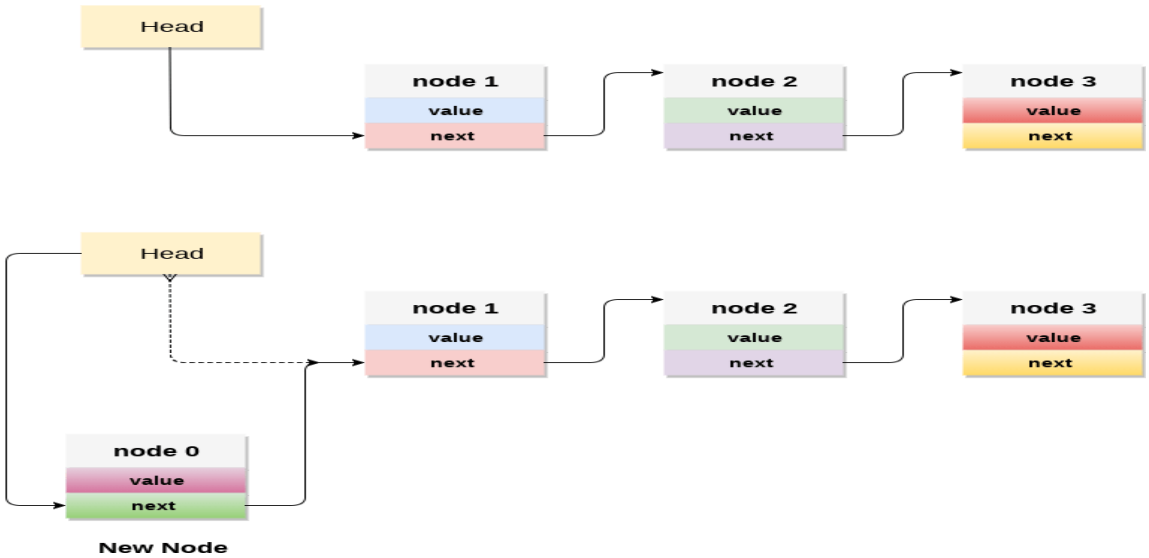
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)



C implementation :

```
void push ()
{
int val;
struct node *ptr =(struct node*)malloc(sizeof(struct node)); if(ptr == NULL)
{
printf("not able to push the element");
}
else
{
printf("Enter the value"); scanf("%d",&val); if(head==NULL)
{
ptr->val = val;
ptr -> next = NULL; head=ptr;
}
else
{
ptr->val = val; ptr->next = head; head=ptr;

}
printf("Item pushed");

}
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to

as **pop** operation. Deleting a node from the linked list

implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
void pop()
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val; ptr = head;
head = head->next; free(ptr);
printf("Item popped");

}
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

3. Copy the head pointer into a temporary pointer.

4. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
void display()
{
int i;
struct node *ptr; ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n"); while(ptr!=NULL)
{
printf("%d\n",ptr->val); ptr = ptr->next;
}
}
}
```

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation cannot be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

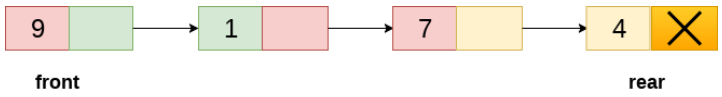
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

ptr -> data = item; if(front == NULL)

{

front = ptr; rear = ptr;

front -> next = NULL; rear -> next = NULL;

}

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

rear -> next = ptr; rear = ptr;

rear->next = NULL;

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL SET FRONT = REAR = PTR

SET FRONT -> NEXT = REAR -> NEXT = NULL ELSE

SET REAR -> NEXT = PTR SET REAR = PTR

SET REAR -> NEXT = NULL [END OF IF]

- **Step 4:** END

C Function

```
void insert(struct node *ptr, int item; )
{



ptr = (struct node *) malloc (sizeof(struct node)); if(ptr == NULL)
{
printf("\nOVERFLOW\n"); return;
}
else
{
ptr -> data = item; if(front == NULL)
{
front = ptr; rear = ptr;
front -> next = NULL;
```

```
rear -> next = NULL;
}
else
{
rear -> next = ptr; rear = ptr;
rear->next = NULL;
}
}
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements

```
ptr = front;
front = front -> next; free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF FRONT = NULL Write " Underflow "

Go to Step 5 [END OF IF]

- **Step 2:** SET PTR = FRONT

- **Step 3:** SET FRONT = FRONT -> NEXT
- **Step 4:** FREE PTR
- **Step 5:** END

C Function

```
void delete (struct node *ptr)
{
if(front == NULL)
{
printf("\nUNDERFLOW\n"); return;
}
else
{
ptr = front;
front = front -> next; free(ptr);
}
}
```

Memory management

Basic task of any program is to manipulate data.These data should be stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:

1. **Static storage management**
2. **Dynamic storage management**

2.1 **Static storage management** In case of static storage management scheme, the net amount of memory required for various data for a program is allocated before the starting of the execution of the program.Once memory is allocated, it neither can be extended nor can be returned to the memory bank for the use of other programs at the same time.

2.2 Dynamic storage management

- Dynamic storage management scheme allows the user to allocate and deallocate as per the necessity during the execution of programs.
- This dynamic memory management scheme is suitable in multiprogramming as well as single-user environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution.
- Operating systems (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is linked list.
- Various principles on which the dynamic memory management scheme is based on

3. Allocation schemes: here we discuss how a request for a memory block will be serviced.

(e) Fixed block allocation

(f) Variable block allocation.

(i) First fit and its variant, (ii) Next fit , (iii) Best fit, (iv) Worst fit

4. Deallocation schemes: here we discuss how to return a memory block to the memory bank whenever it is no more required.

(zzzzz) Random deallocation

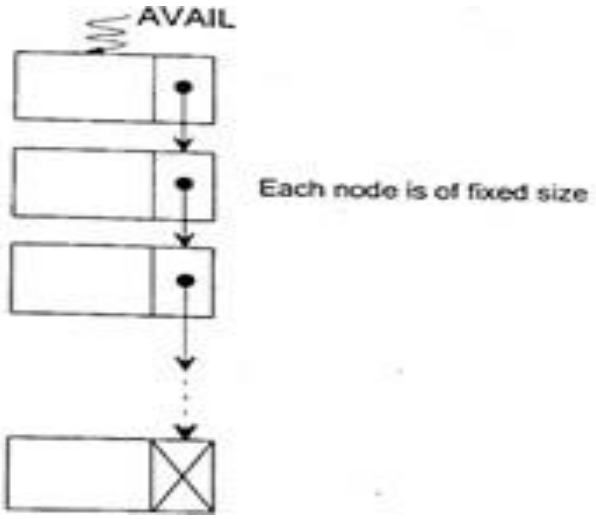
(aaaaaa) Ordered deallocation.

5. Garbage collection: to maintain a memory bank so that it can be utilized efficiently.

2. Allocation schemes

- Memory bank or pool of free storages is often a collection of non- contiguous blocks of memory.

- Their linearity can be maintained by means of pointers between one block to another or in other words, memory bank is a linked list where links are to maintain the adjacency of blocks.



- Regarding the size of the blocks there are two practices: fixed block storage and variable block storage. Let us discuss each of them individually.

- **2.1 Fixed Block Storage**:

Here each block is of the same size.

- The size is determined by the system manager (user).

- Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks.

- A user program communicates with the memory manager by means of two functions GETNODE(NODE) and RETURNNODE(ptr)

- **Procedure GETNODE(NODE)**

Steps

4. **Start**
5. **If (AVAIL = NULL) then**

Print "The memory is insufficient"

6. **Else**

ptr = AVAIL

AVAIL = AVAIL->LINK

Return(ptr)

7. **Endlf**
8. **Stop**

- Whenever a memory block is no more required, it can be returned to the memory bank through a procedure RETURNNODE( )

- **Procedure RETURNNODE(PTR)**
- Steps

3. **Start**
4. **ptrl = AVAIL**
5. **While (ptrl->LINK # NULL) do ptrl = ptr1->LINK**
6. **EndWhile**
7. **ptr1->LINK= PTR**
8. **ptr->LINK= NULL**
9. **Stop**

- Fixed block allocation is the simplest strategy. But the main drawback of this strategy is the wastage of space.
- For example, suppose each memory block is of size 1 K (1024 bytes); now for a request of memory block, say, of size 1.1 K we have to avail 2 blocks (that is 2 K memory space) thus wasting 0.9 K memory space.

3.2 Variable Block Storage

- Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks.
- To overcome the disadvantages of fixed block storage, we can maintain blocks of variable sizes, instead of fixed size blocks.
- **Procedure GETNODE(NODE)**

Steps

7. **Start**
8. **If (AVAIL = NULL) then**
1. **Print "Memory bank is insufficient"**
2. **Exit**
3. **Endlf**
4. **ptr = AVAIL**
5. **While (ptr->LINK ≠NULL) and (ptr->SIZE < SIZEOF(NODE)) do**
1. **ptrl = ptr**
2. **ptr = ptr->LINK**
6. **EndWhile**
7. **If (ptr->LINK = NULL) and (ptr->SIZE < SIZEOF(NODE)) then**
1. **Print "Memory request is too large: Unable to serve"**
8. **Else**
1. **ptrl->LINK = ptr->LINK**

o **2. Return(ptr)**

9. **Endlf 10.Stop**

- This procedure assumes that blocks of memory are stored in ascending order of their sizes.
- **Procedure RETURNNODE(PTR)**

Steps

8. **Start**
9. **ptrl = AVAIL**
10. **While (ptrl->SIZE < ptr->SIZE) and (ptrl->LINK # NULL)) do**
1. **ptr2 = ptrl**
2. **ptrl = ptrl->LINK**
11. **EndWhile**

12. **ptr2->LINK = PTR**
13. **PTR->LINK = ptrl**
14. **Stop**

- The dynamic memory management system should provide the following services:

(d) Searching the memory for a block of requested size and servicing the request (allocation)

(e) Handling a free block when it is returned to the memory manager.

(f) Coalescing the smaller free blocks into larger block(s) (garbage collection and compaction).

3.2.1 Storage allocation strategies

- (a) First-fit allocation, (b) Best-fit allocation, (c) Worst-fit allocation, (d) Next-fit allocation
- **First-fit storage allocation:**

This is the simplest storage allocation strategy.

- Here the list of available storages will be searched and as soon as a free storage block of size N will be found pointer of that block will be sent to the calling program after retaining the residue space.
- For example, for a block of size 2 K, if the first-fit strategy found a block of 7 K, then after retaining a storage block of size 5 K, 2 K memory will be sent to the caller.
- Leads to fast allocation of memory space.
- Leads to memory waste

- **Best-fit storage allocation**
- This strategy will not allocate a block of size > N, as it is found in first- fit method, instead will continue searching to find a suitable block so that the block size is closer to the block size of request.
- Goal: find the smallest memory block into which the job will fit
- Results in least wasted space.
- Slower in making allocation
- For example, for a request of 2 K, if the list contains the blocks of sizes, 1 K, 3 K, 7 K, 2.5 K, 5 K, then it will find the block of size 2.5 K as suitable block for allocation. From this block after retaining 0.5 K, pointer for 2 K block will be returned.
- **Worst-fit storage allocation**
- Slower in making allocation
- Allocates the largest free available block to the new job
- Opposite of best-fit
- Best-fit finds a block which is small and nearest to the block size as 'requested, whereas, worst-fit strategy is a reverse of it. It allocates the largest block available in the available storage list.
- The idea behind the worst-fit is to reduce the rate of production of small blocks which are quite common when best-fit strategy is used for memory allocation.
- **Next-fit storage allocation**
- The idea behind the worst-fit is to reduce the rate of production of small blocks which are quite common when best-fit strategy is used for memory allocation.
- Next-fit allocation strategy is a modification of first-fit strategy.

- Starts searching from last allocated block, for the next available block when a new job arrives.
- In case of first-fit strategy, searching will always occur from beginning of the free list whereas in next-fit strategy, search begins where the last allocation has been done; in this strategy, pointer to the free list is saved following an allocation and is used to begin for the subsequent request.

The idea of this strategy is to reduce the search by avoiding examination of smaller blocks that, in long run, tends to be created at the beginning of the free list as it happens in case of first-f

Module 4 Trees and Graphs

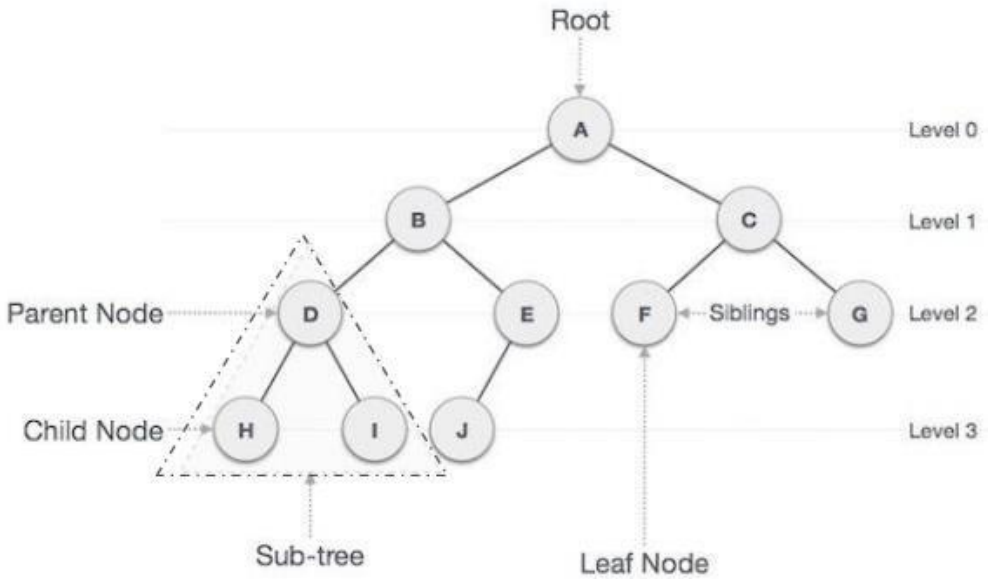
Trees, Binary Trees-Tree Operations, Binary Tree Representation, Tree Traversals, Binary Search Trees- Binary Search Tree Operations Graphs, Representation of Graphs, Depth First Search and Breadth First Search on Graphs, Applications of Graphs

10. The tree has one node called root. The tree originates from this, and hence it does not have any parent.

11. Each node has one parent only but can have multiple children.

12. Each node is connected to its children via edge.

Following diagram explains various terminologies used in a tree structure.



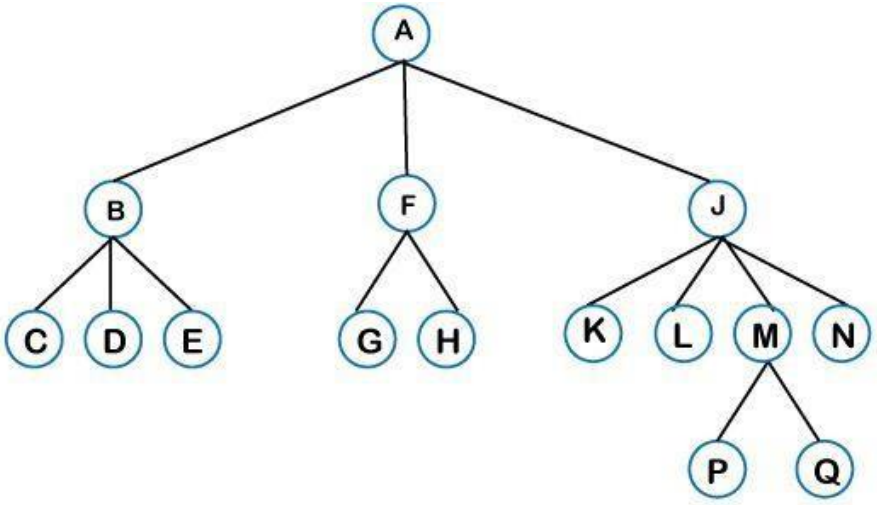
Terminology	Description	Example From Diagram
Root	Root is a special node in a tree. The entiretree originates from it. It does not have a parent.	Node A
Parent Node	Parent node is an immediate predecessor ofa node.	B is parent of D & E
Child Node	All immediate successors of a node are itschildren.	D & E are children of B
Leaf	Node which does not have any child iscalled as leaf	H, I, J, F and G are leaf nodes
Edge	Edge is a connection between one node to another. It is a line between two nodes or anode and a leaf.	Line between A & B is edge
Siblings	Nodes with the same parent are called Siblings.	D & E are siblings
Path / Traversing	Path is a number of successive edges fromsource node to destination node.	A – B – E – J is path from nod A toE
Height of Node	Height of a node represents the number of edges on the longest path between that nodeand a leaf.	A, B, C, D & E can have height. Height of A is no. of edges betweenA and H, as that is the longest path,which is 3. Height of C is 1
Levels of node	Level of a node represents the generation ofa node. If the root node is at level 0, then itsnext child node is at level 1, its grandchild is at level 2, and so on	Level of H, I & J is 3. Level of D, E,F & G is 2
Degree of Node	Degree of a node represents the number ofchildren of a node.	Degree of D is 2 and of E is 1

Sub tree	Descendants of a node represent subtree.	Nodes D, H, I represent one subtree.

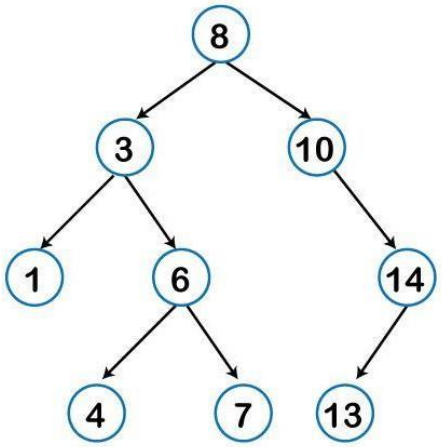
Types of Trees

Types of trees depend on the number of children a node has. There are two majortree types:

- General Tree: A tree in which there is no restriction on the number of children a node has, is called a General tree. Examples are Family tree, Folder Structure.

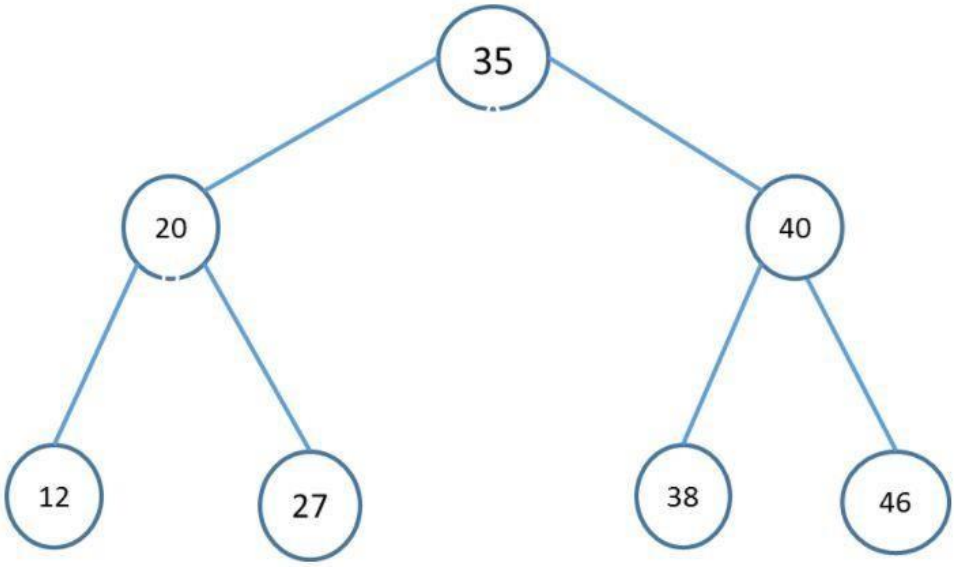


- Binary Tree: In a Binary tree, every node can have at most 2 children, left and right. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



Binary trees are further divided into many types based on its application.

- Full Binary Tree: If every node in a tree has either 0 or 2 children, thenthe tree is called a full tree.
- Perfect Binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- Binary Search Tree: It is a binary tree with binary search property. Binary search property states that the value or key of the left node is less than its parent and value or key of right node is greater than its parent. And this is true for all nodes.


35
20
40
12
27
38
46

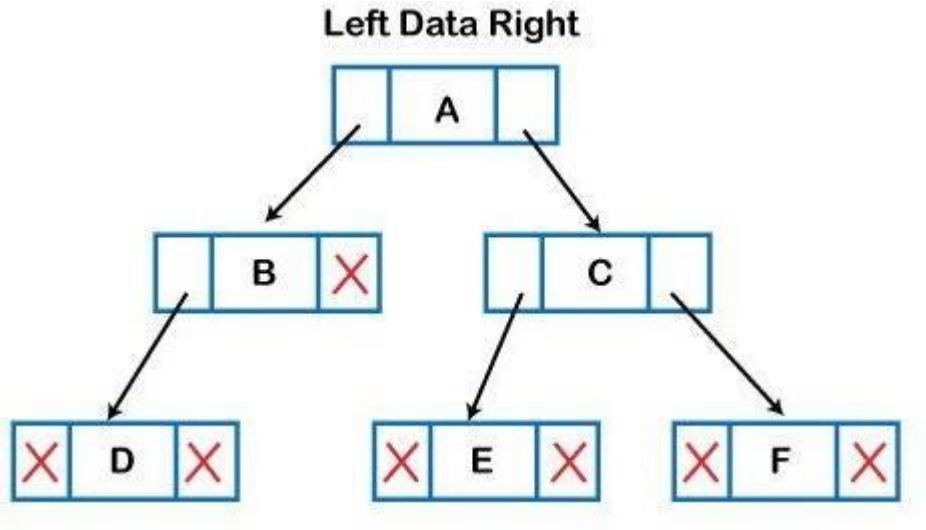
Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored onthe disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routingtables in the routers.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the addressof the right child.

In programming, the structure of a node can be defined as:

```
struct node
{
int data;
struct node *leftChild; struct node
    *rightChild;
};
```

Binary Search Tree Basic Operations

The basic operations that can be performed on a binary search tree data structure,are the following −

- Insert − Inserts an element in a tree/create a tree.
- Search − Searches an element in a tree.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree andinsert the data.

Algorithm

```
Step 1: Start
Step 2:If root is NULL
              then create root node
              return

Step 3:If root exists then
   compare the data with node.data

Step 4:while until insertion position is located
Step 4.1:If data is greater than node.data
          goto right subtree
Step 4.2:else
          goto left subtree

Step 5:endwhile
```

```
Step 7: Stop
```

Program

```
void insert(int data) {
struct node *tempNode = (struct node*) malloc(sizeof(structnode));
struct node *current;struct node
    *parent;

tempNode->data = data; tempNode-
    >leftChild = NULL; tempNode-
    >rightChild = NULL;

//if tree is empty, create root nodeif(root == NULL) {
root = tempNode;
} else {
current = root;parent   = NULL;

while(1) {
parent = current;

//go to left of the tree if(data < parent->data) {
current = current->leftChild;

//insert to the left if(current == NULL) {
parent->leftChild = tempNode;return;
}
}

//go to right of the treeelse {
```

```
current = current->rightChild;

//insert to the rightif(current == NULL) {
parent->rightChild = tempNode;return;
}
}
}
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
Step 1: Start

Step 2:If root.data is equal to search.data
   return root

Step 3:else

   while data not found

Step 3.1:If data is greater than node.data
         goto right subtree

Step 3.2:else

         goto left subtree

Step 4:If data found

         return node
   endwhile
```

Program

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
```

```
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
```

Tree Traversals

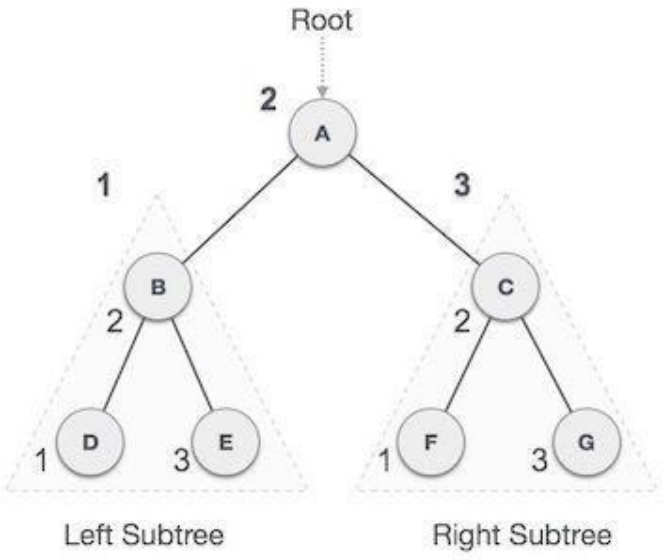
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtreeitself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm:

Step 1: Start

Step 2: Until all nodes are traversed Step 3:

Recursively traverse left subtree.

Step 4: Visit root node.

Step 5: Recursively traverse right subtree.Step 6:

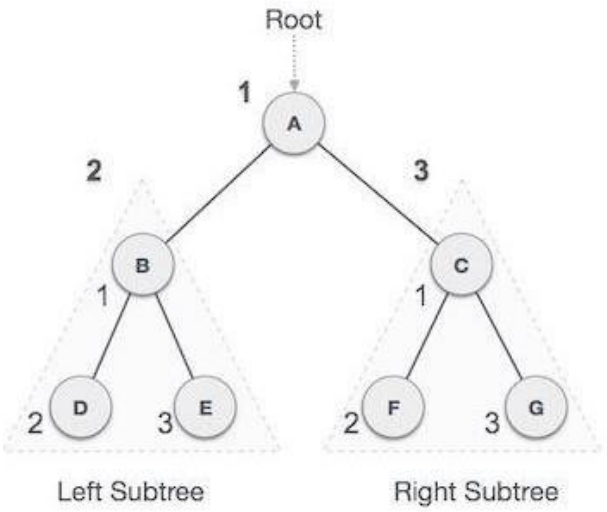
Stop

Program:

```
void inorder_traversal(struct node* root) {
   if(root != NULL) {

      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);

   }
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree andfinally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm:

Step 1: Start

Step 2: Until all nodes are traversedStep 3:

Visit root node.

Step 4: Recursively traverse left subtree. Step 5:

Recursively traverse right subtree.Step 6:

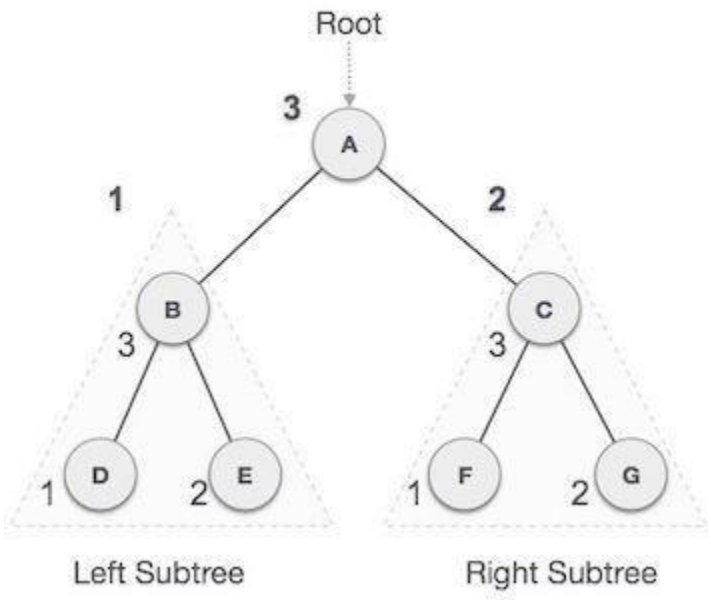
Stop

Program:

```
void pre_order_traversal(struct node* root) {
   if(root != NULL) {

      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);

   }
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Step 1: Start

Step 2: Until all nodes are traversed Step 3:

Recursively traverse left subtree.

Step 4: Recursively traverse right subtree.Step 5:

Visit root node.

Step 6: Stop

Program:

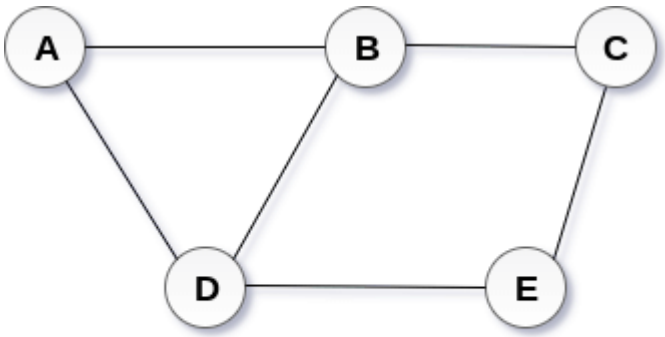
```
void post_order_traversal(struct node* root) {
   if(root != NULL) {

      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);

   }
```

GRAPH:

- A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.
- Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges.

Directed and Undirected Graph

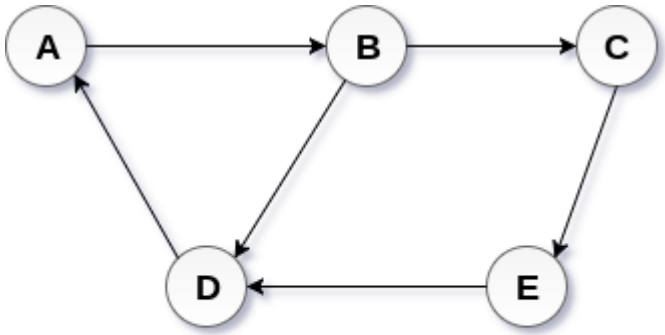
- A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the below figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.



Undirected Graph

- In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node

while node B is called terminal node. A directed graph is shown in thefollowing figure.



Directed Graph

- Before we proceed further, let's familiarize ourselves with some importantterms −

- **Vertex** − Each node of the graph is represented as a vertex.
- **Edge** − Edge represents a path between two vertices or a line betweentwo vertices.
- **Adjacency** − Two node or vertices are adjacent if they are connectedto each other through an edge.
- **Path** − Path represents a sequence of edges between the two vertices.
- **Closed Path** - A path will be called as closed path if the initial node issame as terminal node. A path will be closed path if $V_0=V_N$.
- **Simple Path** - If all the nodes of the graph are distinct with anexception $V_0=V_N$, then such path P is called as closed simple path.
- **Cycle -** A cycle can be defined as the path which has no repeated edgesor vertices except the first and last vertices.

- **Connected Graph** - A connected graph is the one in which some pathexists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Weighted Graph** - In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost oftraversing the edge.
- **Degree of the Node** - A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolatednode.

Application of Graph in Data Structure

Graphs data structure have a variety of applications. Some of the most popular applications are:

- Helps to define the flow of computation of software programs.
- Used in Google maps for building transportation systems. In google maps, the intersection of two or more roads represents the node while the road connecting two nodes represents an edge. Google maps algorithm uses graphsto calculate the shortest distance between two vertices.
- Used in social networks such as Facebook and Linkedin.
- Operating Systems use Resource Allocation Graph where every process and resource acts as a node while edges are drawn from resources to the allocatedprocess.
- Used in the world wide web where the web pages represent the nodes.

- Blockchains also use graphs. The nodes are blocks that store manytransactions while the edges connect subsequent blocks.
- Used in modeling data.

Breadth First Search in Data Structure

Traversal means to visit each node of a graph. For graphs, there are two types of traversals: Depth First traversal and Breadth-First traversal. In this article, we are going to study Breadth-first traversal or BFS in detail.

What is Breadth First Search?

Breadth First Search is a traversal technique in which we traverse all the nodes of the graph in a breadth-wise motion. In BFS, we traverse one level at a time and thenjump to the next level.

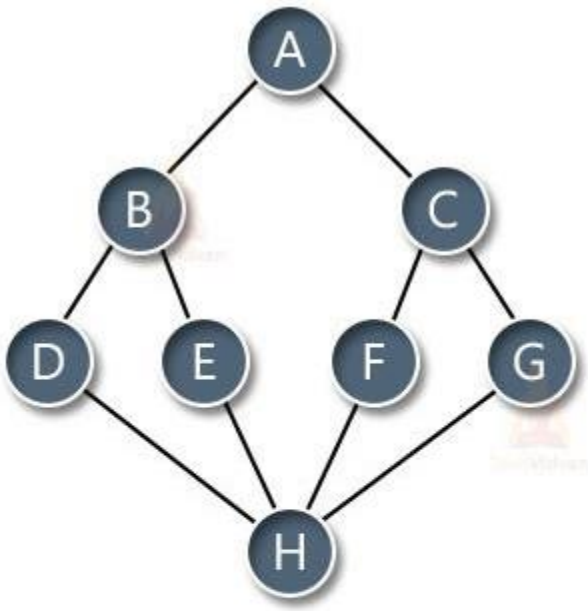
In a graph, the traversal can start from any node and cover all the nodes level-wise.The BFS algorithm makes use of the queue data structure for implementation. In BFS, we always reach the final goal state (which was not always the case for DFS).

Algorithm for BFS:

Step 1: Choose any one node randomly, to start traversing.Step 2:
Visit its adjacent unvisited node.
Step 3: Mark it as visited in the boolean array and display it.Step 4:
Insert the visited node into the queue.
Step 5: If there is no adjacent node, remove the first node from the queue.Step 6:
Repeat the above steps until the queue is empty.

Working of Breadth First Search:

Consider the following graph having 8 nodes named as A, B, C, D, E, F, G and H asshown:

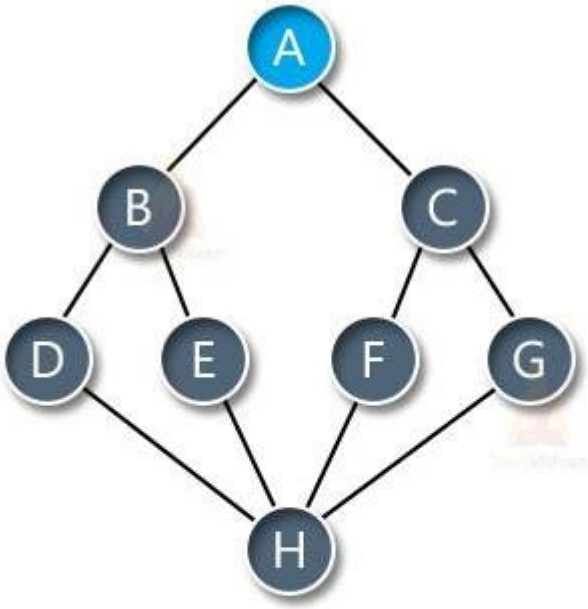


	A	B	C	D	E	F	G	H
Visited:	0	0	0	0	0	0	0	0

Queue: **Empty**

Initially, the queue will be empty.

Let us start traversing the graph from node A. Once we visit node A, we mark it asvisited and also place it inside the queue as shown:

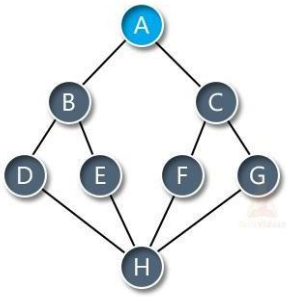


	A	B	C	D	E	F	G	H
Visited:	1	0	0	0	0	0	0	0

Queue: **A**

The next step is to traverse its adjacent nodes i.e. B and C and place them inside thequeue.

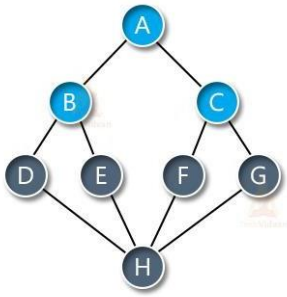
When we place the adjacent node of A in the queue, we will remove A fromthe queue and display it in the output array as shown:



	A	B	C	D	E	F	G	H
Visited:	1	0	0	0	0	0	0	0

Queue: **Empty**

Output: **A**

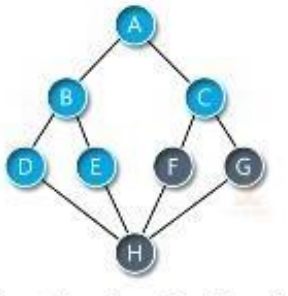


	A	B	C	D	E	F	G	H
Visited:	1	1	1	0	0	0	0	0

Queue: **B,C**

Output: **A**

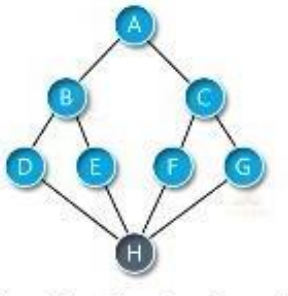
Next, we don't have any more adjacent nodes for A. therefore, we will remove node B from the queue and place its adjacent nodes into the queue. Similarly, we will traverse the nodes of C and put them into the queue



	A	B	C	D	E	F	G	H
Visited:	1	1	1	1	1	0	0	0

Queue: **C,D,E**

Output: **A,B**

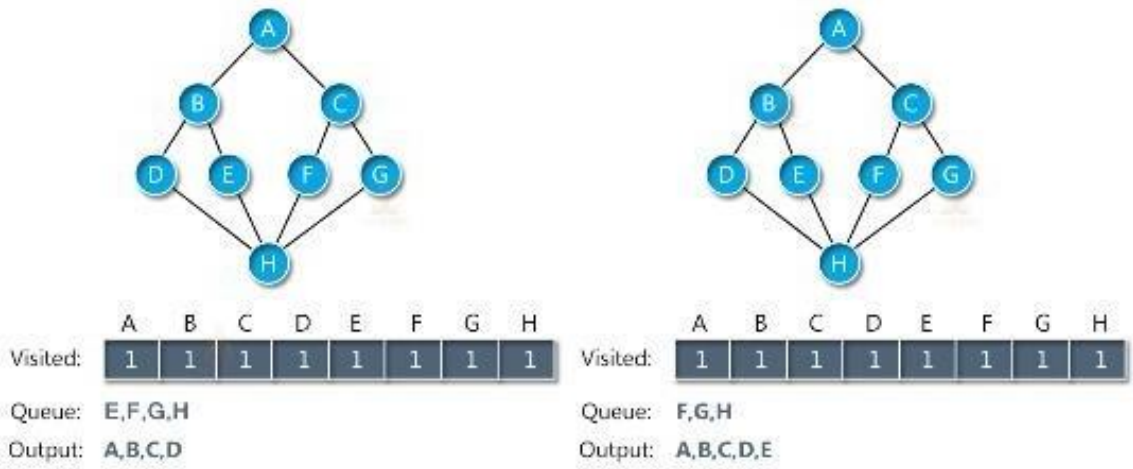


	A	B	C	D	E	F	G	H
Visited:	1	1	1	1	1	1	1	0

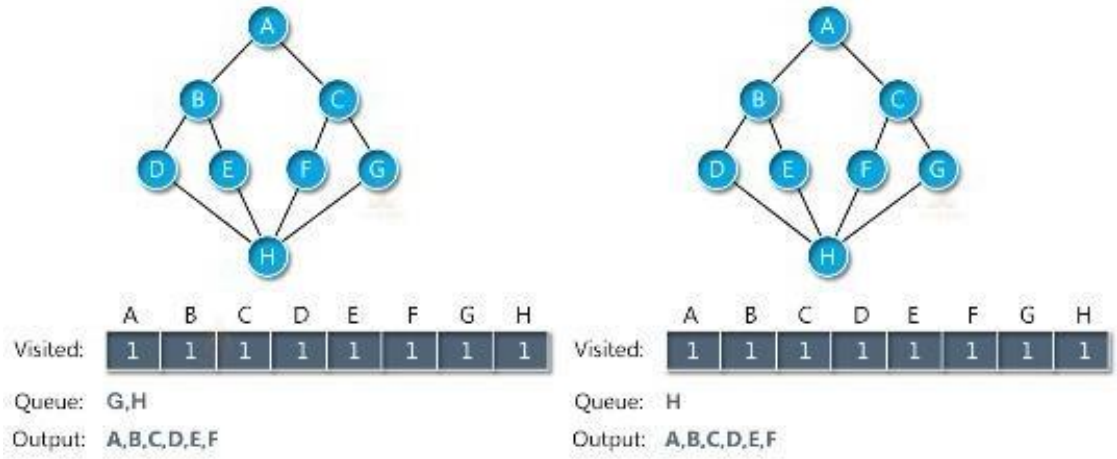
Queue: **D,E,F,G**

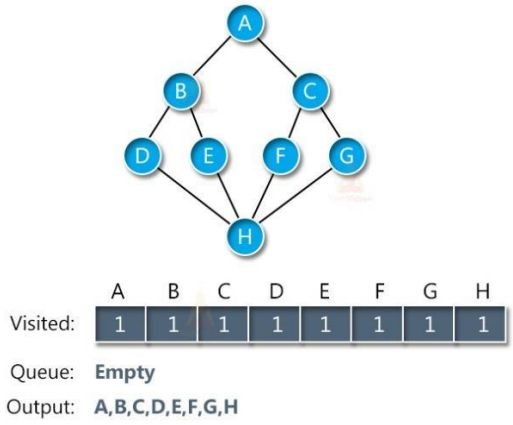
Output: **A,B,C**

The next adjacent node is node H. thus, we will traverse that as well and place it in the queue. Once all the nodes have entered the queue, we will start removing them from the queue and putting them into the output array.



Thus, slowly the queue starts decreasing in size and the output array will be fullas shown:





Complexity of BFS

Time complexity: Since we are visiting all the nodes exactly once, therefore, the time complexity is **O(V+E).** Here, O(E) may vary between O(1) and O(V2). Thus, in the worst case, the time complexity of BFS is **O(V2).**

Space complexity: The space complexity of the BFS algorithm is **O(V)** where V denotes vertices/nodes of the graph.

Applications of Breadth First Search

- To find the shortest path between two edges when the path length is equivalentto the number of edges.
- To check whether a graph is bipartite or not
- To copy garbage collection by Cheney's algorithm
- Used in unweighted graphs to find the minimum cost spanning tree
- To form peer-to-peer network connections
- To find neighboring locations in the GPS navigation system
- To detect cycle in an undirected graph
- To broadcast packets in a network
- To find all the nodes within one connected component in an otherwise disconnected graph

Depth First Search in Data Structure

Depth First Search is a traversal technique in which we follow a depth-wise motion to traverse all the nodes. This technique is based on backtracking.

What is DFS:

Depth first search is a recursive technique to traverse all the nodes of a graph. It makes use of the stack data structure for traversing and remembering the nodes.

DFS follows the backtracking approach i.e. whenever there are no more nodes in thecurrent path, it goes back to the initial node and starts traversing the next available path.

While using the stack, we first choose the initial node and push all its adjacent nodesinto the stack. To visit a node, we pop a node from the stack and push all its adjacentnodes to the stack.

This goes on until all the nodes are popped out i.e. the stack is empty. In the whole process, we need to make sure that we don't visit the same node more than once, especially if the cycle exists. The output of DFS is always in the form of a spanning tree.

Algorithm:

Step 1: Algorithm DFS(vertex v)Step 2:

visited[v] = 1

Step 3: for all nodes 'w' adjacent to v:

if(visited[w] == 0):

DFS(w)

Step 4: End for loopStep 5:

End DFS

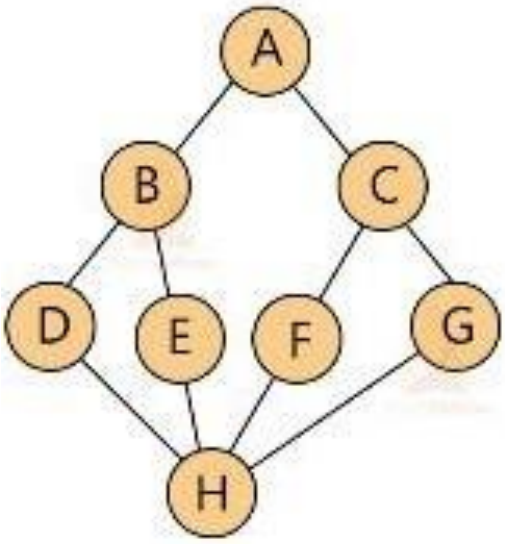
Working of DFS:

Before looking at the working of the algorithm, let us first understand the followingterms:

- **Visit:** It means we reached the node or we are reaching the node.
- **Explore:** It means processing every child of the node.
- **Discovery:** This term is used when we visit a node for the first time.

In DFS, we can take any node as the initial node and start traversing its adjacent nodes.

Let us see the working of DFS with the help of the following example:

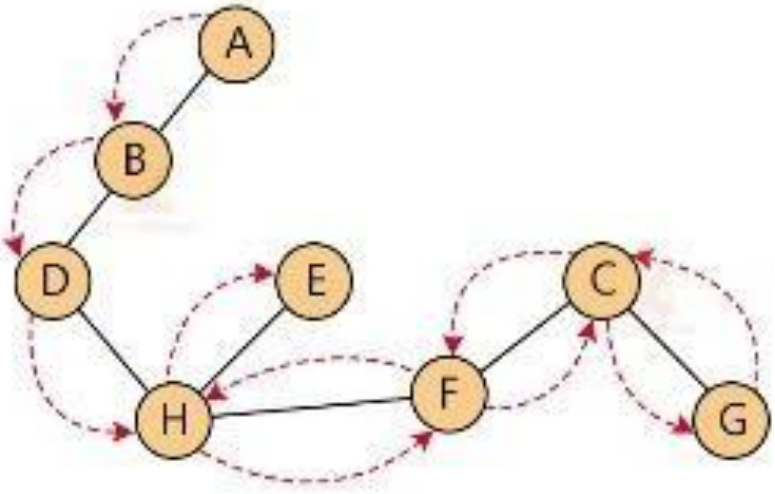


Let's start with node A. once we reach the child of A, we start exploring the grandchild of A i.e. the child of B and so on. In this manner, we will traverse the whole graph.

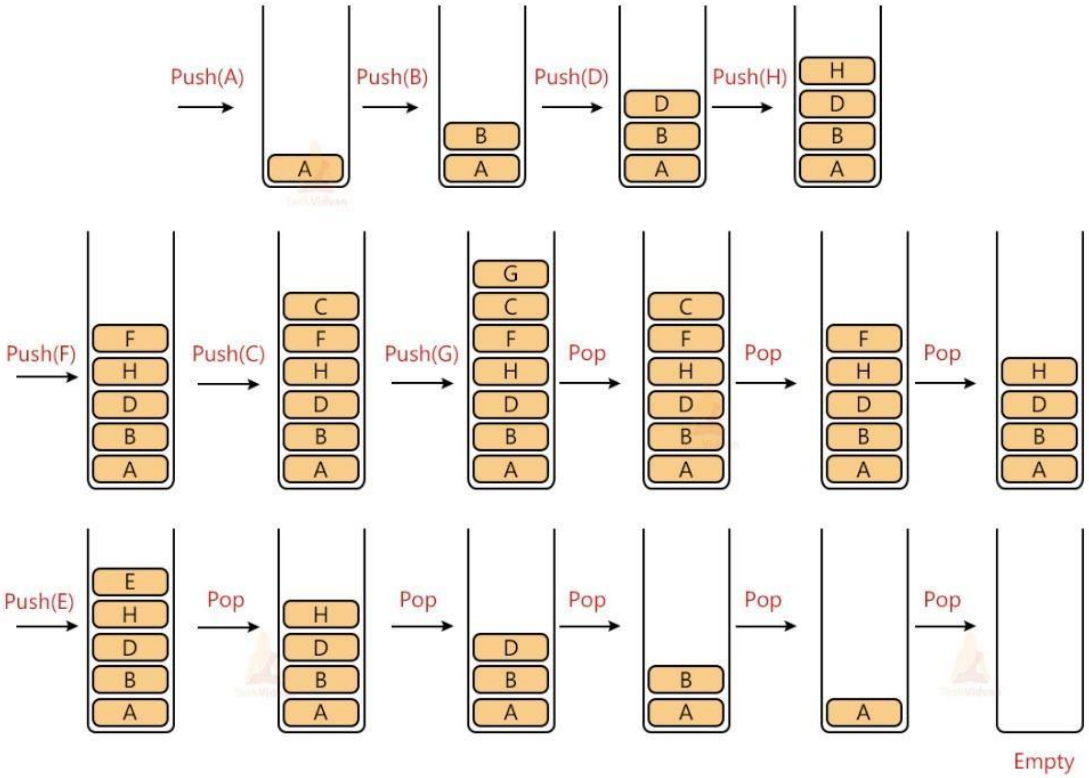
We go from A to B to D to H. when we reach H, we have 3 choices: E, F, G. We canchoose to go with any of them. Let us go with F. from F, we got to C as C is the onlyunvisited node of F.

Similarly, from C we will visit G. Now G does not have any unvisited node. Therefore, G is the dead node and we will backtrack from here.

Thus, the spanning tree formed by following this path will be:



The push and pop operations in the stack will be:



Thus, the path followed is A→B→D→H→F→C→G→E. We need to note that there could multiple paths for the same graph depending upon which node is traversed first.

Complexity Analysis for DFS:

Time complexity: We need to traverse the whole graph while implementing DFS. therefore, its time complexity is O(V + E).

Space complexity: The algorithm makes use of an extra array. Therefore, in the worst case, the space complexity is O(V).

Applications of DFS:

- Finding the number of connected components in a disconnected graph
- Detecting a cycle in a graph
- Finding all the articulation points in a graph
- Finding whether a graph is biconnected or not
- For finding strongly connected components in a graph
- To find bridges in a graph
- For topological sorting
- To solve puzzles with only one solution (Eg- mazes)

Module 5 Sorting and Hashing

Sorting Techniques – Selection Sort, Insertion Sort, Quick Sort, Merge Sort and Heap Sort Hashing- Hashing Techniques, Collision Resolution, Overflow handling, Hashing functions – Mid square, Division, Folding, Digit Analysis

SORTING TECHNIQUES:

- Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields.
- Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

Complexity of Sorting Algorithm

- The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:
- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program
- Various sorting techniques are analyzed in various cases and named these cases as

follows:

- Best case
- Worst case

- Average case

SELECTION SORT

- Selection sort is a simple sorting algorithm.
- This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped withthe leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to theright.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.
- Consider the following

- For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

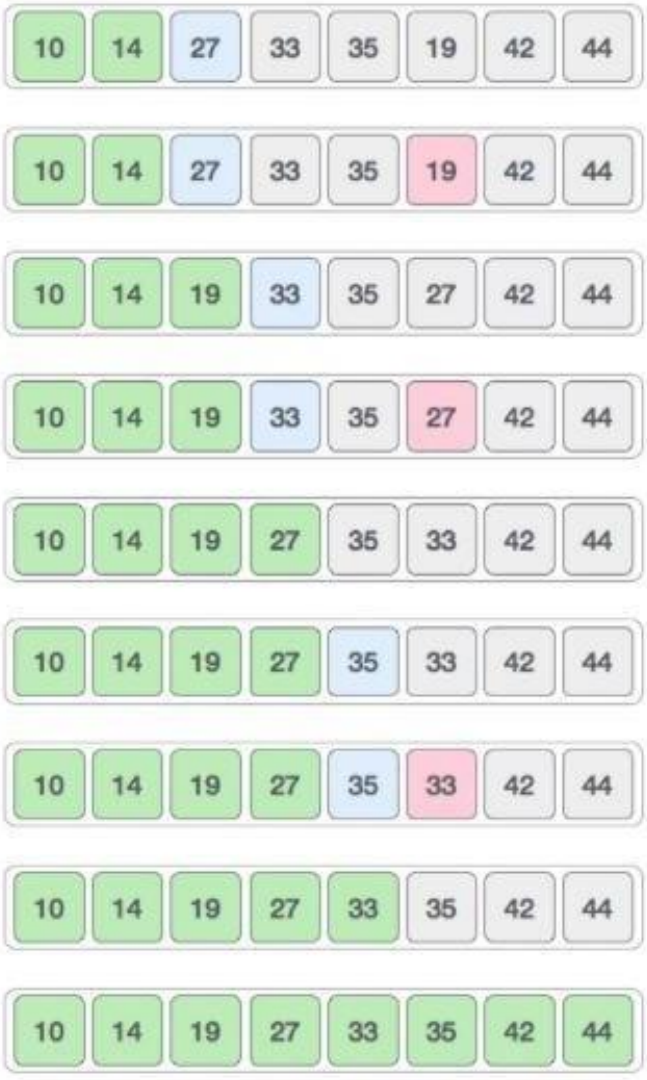
- So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

- For the second position, where 33 is residing, we start scanning the rest of thelist in a linear manner.

- We find that 14 is the second lowest value in the list and it should appear atthe second place. We swap these values.

- After two iterations, two least values are positioned at the beginning in asorted manner.

The scenario will perform as follows:



Program for selection sort

```
#include <stdio.h>int
    main()
{
int arr[10]={6,12,0,18,11,99,55,45,34,2};
int n=10;
int i, j, position, temp; for (i = 0; i < (n
    - 1); i++)
{
position = i;
for (j = i + 1; j < n; j++)
{
if (arr[position] > arr[j])position = j;
}
if (position != i)
{
temp = arr[i];
arr[i] = arr[position];arr[position] = temp;
}
}
for (i = 0; i < n; i++)
```

```
printf("%d\t", arr[i]);return 0;
}
```

Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

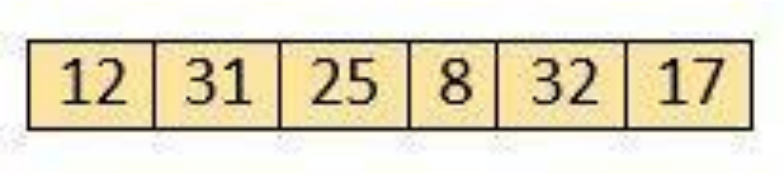
- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sortis **$O(n^2)$**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

INSERTION SORT

- Insertion sort works similar to the sorting of playing cards in hands.
- It is assumed that the first card is already sorted in the card game, and then weselect an unsorted card.
- If the selected unsorted card is greater than the first card, it will be placed atthe right side; otherwise, it will be placed at the left side.
- Similarly, all unsorted cards are taken and put in their exact place.
- The same approach is applied in insertion sort.
- The idea behind the insertion sort is that first take one element, iterate it through the sorted array.
- Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $\mathbf{O(n^2)}$, wheren is the number of items.
- Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.
- Insertion sort has various advantages such as –
- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantiallysorted.

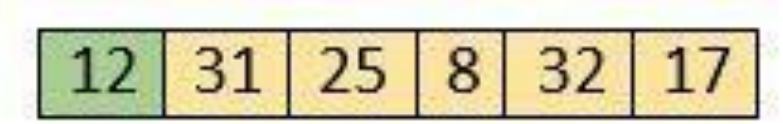
Working of Insertion sort Algorithm

Let the elements of array are –

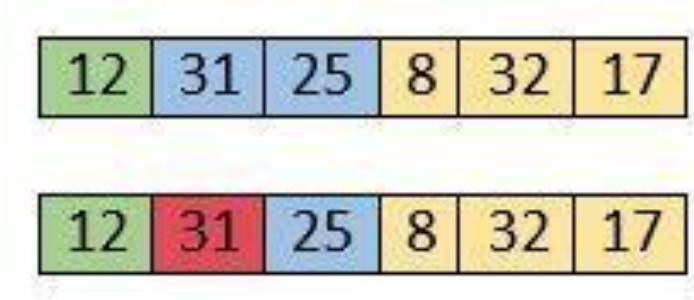


Initially, the first two elements are compared in insertion sort.

Here, 31 is greater than 12. That means both elements are already in ascendingorder. So, for now, 12 is stored in a sorted sub-array.



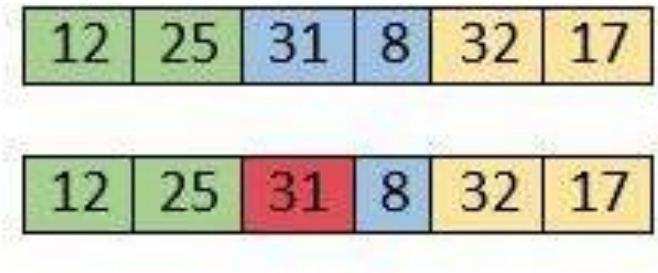
Now, move to the next two elements and compare them.



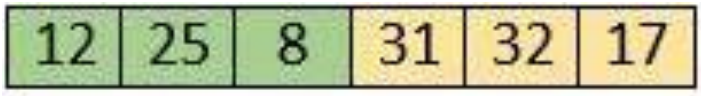
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in thesorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12.Hence, the sorted array remains sorted after swapping.

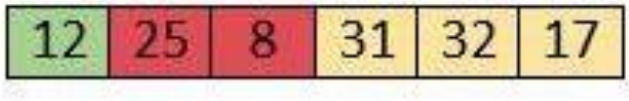
Now, two elements in the sorted array are 12 and 25. Move forward to the nextelements that are 31 and 8.



Both 31 and 8 are not sorted. So, swap them.

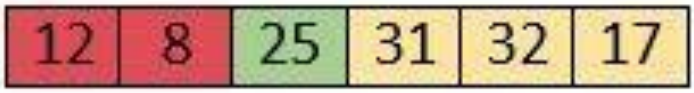


After swapping, elements 25 and 8 are unsorted.

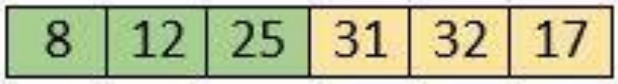


So, swap them.

Now, elements 12 and 8 are unsorted.



So, swap them too.

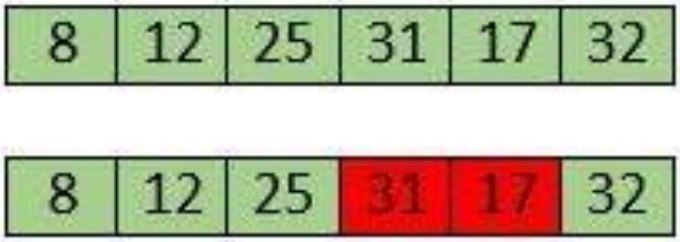


Now, the sorted array has three items that are 8, 12 and 25. Move to the next itemsthat are 31 and 32.

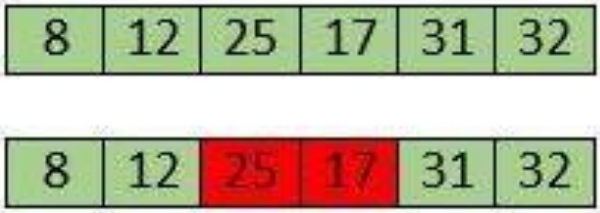
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Move to the next elements that are 32 and 17

17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.

Now, the array is completely sorted.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return1.

Step2 - Pick the next element, and store it separately in a **key. Step3 -** Now,

compare the **key** with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, thenmove to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Program #include<stdio.h>int

main()

{

/* Here i & j for loop counters, temp for swapping,count for

total number of elements, number[] to

* store the input numbers in array. You can increase

* or decrease the size of number array as per requirement

*/

int i, j, count, temp, number[25];

printf("How many numbers u are going to enter?: ");

```
scanf("%d",&count);
printf("Enter %d elements: ", count);
// This loop would store the input numbers in array
    for(i=0;i<count;i++) scanf("%d",&number[i]);
// Implementation of insertion sort algorithmfor(i=1;i<count;i++)
{
temp=number[i];j=i-1;
while((temp<number[j])&&(j>=0))
{
number[j+1]=number[j];j=j-1;
}
number[j+1]=temp;
}
printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
printf(" %d",number[i]);return 0;
}
```

Time Complexity

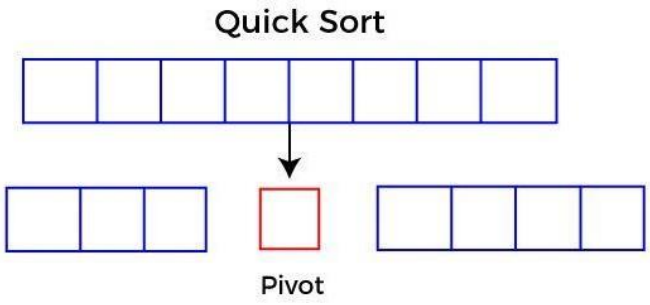
Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

QUICK SORT

- Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements.
- It is a faster and highly efficient sorting algorithm.
- This algorithm follows the divide and conquer approach.
- Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

- **Divide:** In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
- **Conquer:** Recursively, sort two subarrays with Quicksort
- Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element.
- In quick sort, a large array is divided into two arrays in which one holds valuesthat are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.
- After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

- Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -
- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of thegiven array.

- Select median as the pivot element.

Algorithm

Step 1: Start

Step 2: Consider first element as a pivot element Step 3:

Initialize ‘i’ to low index, ‘j’ to high indexStep 4: Repeat

the following steps until i<j

Step 4.1: Keep on incrementing ‘i‘while a[i] <=pivotStep 4.2: Keep on

decrementing ‘j’ while a[j]>pivot Step 4.3: if i<j then swap (a[i],

a[j])

Step 5: If i>j then swap (a[j], pivot), j is the position of pivotStep 6: Stop

Program

```
#include<stdio.h>
void quicksort(int number[25],int first,int last)
{
int i, j, pivot, temp;
    if(first<last)
{
pivot=first;i=first;
    j=last;
    while(i<j)
```

```
{
while(number[i]<=number[pivot]&&i<last)i++;
while(number[j]>number[pivot])j--;
if(i<j)
{
temp=number[i]; number[i]=number[j];
    number[j]=temp;
}
}
temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
}
}
int main()
{
int i, count, number[25];
printf("How many elements are u going to enter?: ");
```

```
scanf("%d",&count);
printf("Enter %d elements: ", count);
    for(i=0;i<count;i++)
    scanf("%d",&number[i]);
    quicksort(number,0,count-1);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
printf(" %d",number[i]);return
    0; }
```

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

MERGE SORT

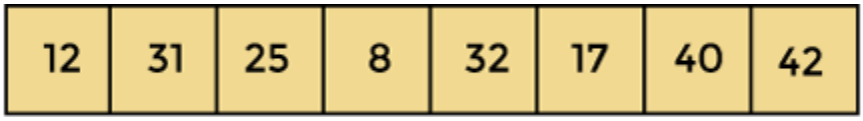
- Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements.
- It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves.
- We have to define the **merge()** function to perform the merging.
- The sub-lists are divided again and again into halves until the list cannot be divided further.
- Then we combine the pair of one element lists into two-element lists, sorting them in the process.

- The sorted two-element pairs is merged into the four-element lists, and so onuntil we get the sorted list.

Working of Merge Sort

To understand the working of the merge sort algorithm, let's take an unsorted array.It will be easier to understand the merge sort via an example.

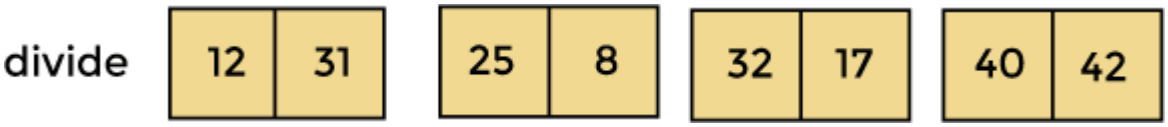
Let the elements of array are -



According to the merge sort, first divide the given array into two equal halves. Mergesort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size4.

Now, again divide these two arrays into halves. As they are of size 4, so divide theminto new arrays of size 2.

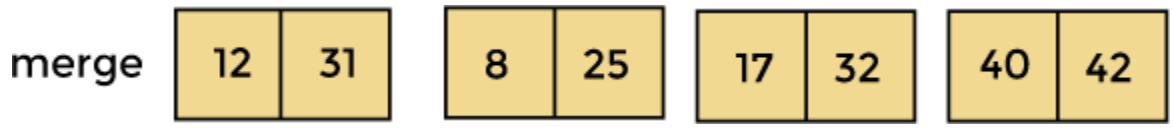


Now, again divide these arrays to get the atomic value that cannot be further divided.

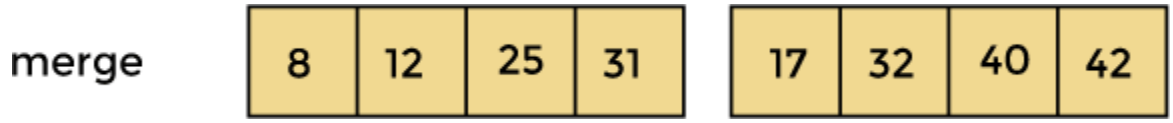
Now, combine them in the same manner they were broken

In combining, first compare the element of each array and then combine them into another array in sorted order.

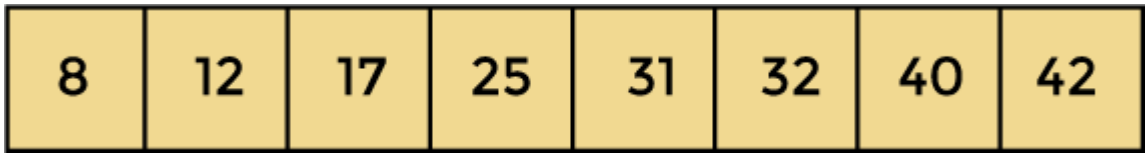
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values andmerge them into an array of found values in sorted order.


merge
8
12
25
31
17
32
40
42

Now, there is a final merging of the arrays. After the final merging of above arrays,the array will look like -



Now, the array is completely sorted.

Program

```
mergesort(arr[], l, r)
{
if(l<r)
{
                        m=(l+r)/2;
                            mergesort(arr,l,
                            m);
                            mergesort(arr,
}                           m+1,r);
        }                   merge(arr,l,m,r
                            );

merge(arr,l,m,r)
{
int i=l,j=m+1,k=1; int temp[];
    while(i<=m&&j<=r)
{
if(arr[i]<=arr[j])
{
```

```
                        temp[k]=arr[i]
                            ;i++;
                        k++;
}
else
{
                        temp[k]=arr[j]
                            ;j++;
                        k++;
}
            }
while(i<=m)
{
temp[k]=arr[i];i++;
k++;
}
while(j<=r)
{
temp[k]=arr[j];j++;
k++;
}
for(int p=l;p<=r;p++)
{
arr[p]=temp[p];
}
}

#include<stdio.h>int
    main()
{
int arr[],l,r,n,i;
printf("Enter the number of elements:");
    scanf("%d",&n);
```

```
printf("Enter the elements:")
    for(i=0;i<n;i++)
{
scanf("%d",&arr[i]);
} l=0;
r=n-1;
mergesort(arr[],l,r);
}
```

Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sortis **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array

elements in ascending order, but its elements are in descending order. Theworst-case time complexity of merge sort is **O(n*logn)**.

HEAP SORT

- Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.
- Heap sort basically recursively performs two main operations -

o Build a heap H, using the elements of array.

o Repeatedly delete the root element of the heap formed in 1st phase.

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.
- **There are two variants of a heap: max-heap and min-heap.** The heap properties change a bit with each variant.
- **According to the heap property, the key or value of each node in a heap is always greater than its children nodes, and the key or value of the root node is always the largest in the heap tree.**

- The heap property for min-heap states that **the value or key of each child node is always greater than its parent node, and the value of the root node is always the smallest in the heap.**

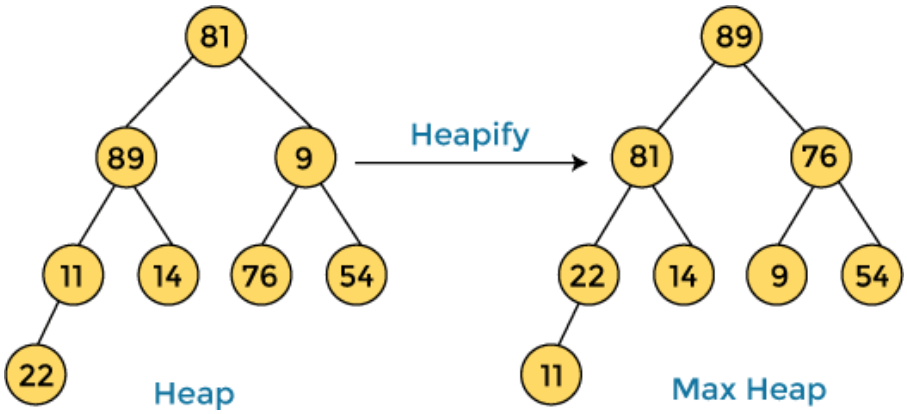
Working of Heap sort Algorithm

In heap sort, basically, there are two phases involved in the sorting of elements. Byusing the heap sort algorithm, they are as follows -

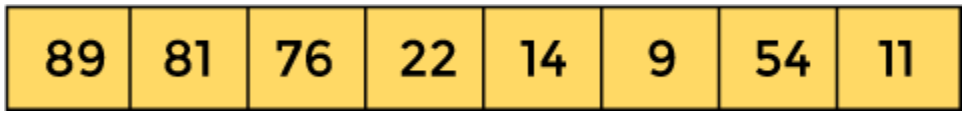
- The first step includes the creation of a heap by adjusting the elements of thearray.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with theremaining elements.

Let's take an example:

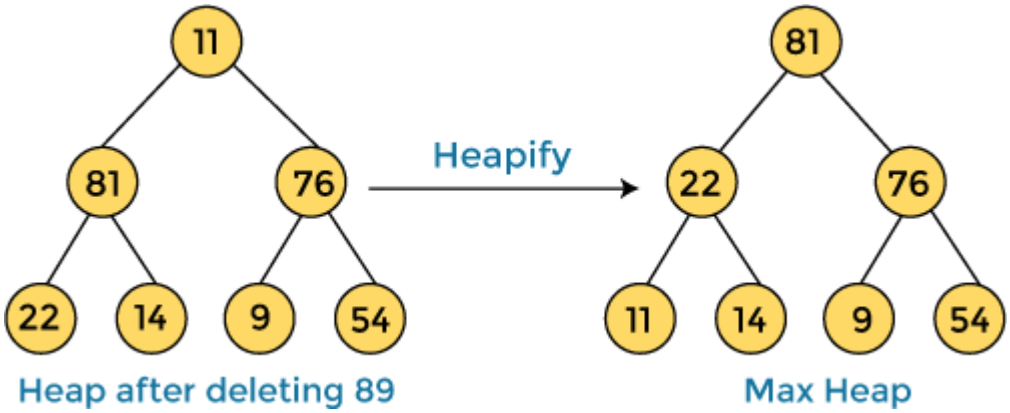
First, we have to construct a heap from the given array and convert it into max heap.



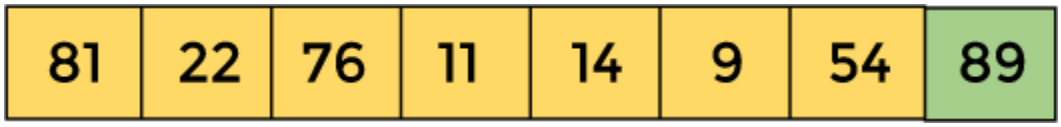
After converting the given heap into max heap, the array elements are -



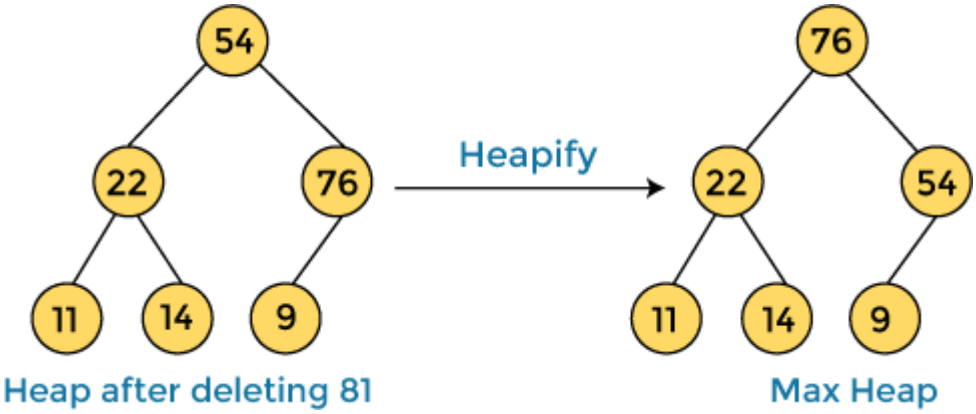
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



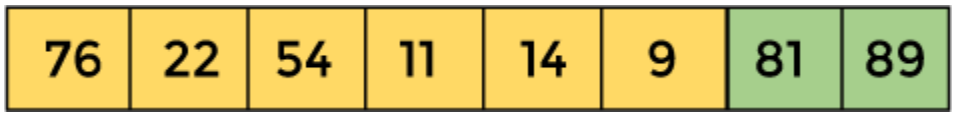
After swapping the array element **89** with **11,** and converting the heap into max- heap, the elements of array are -



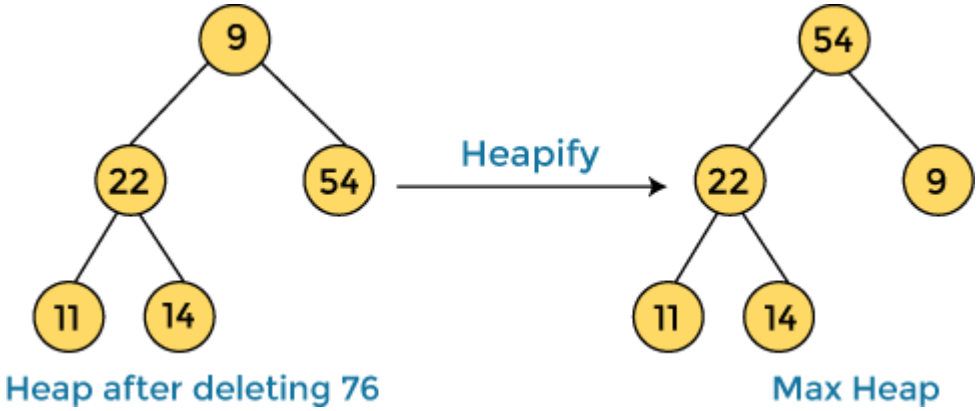
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are –

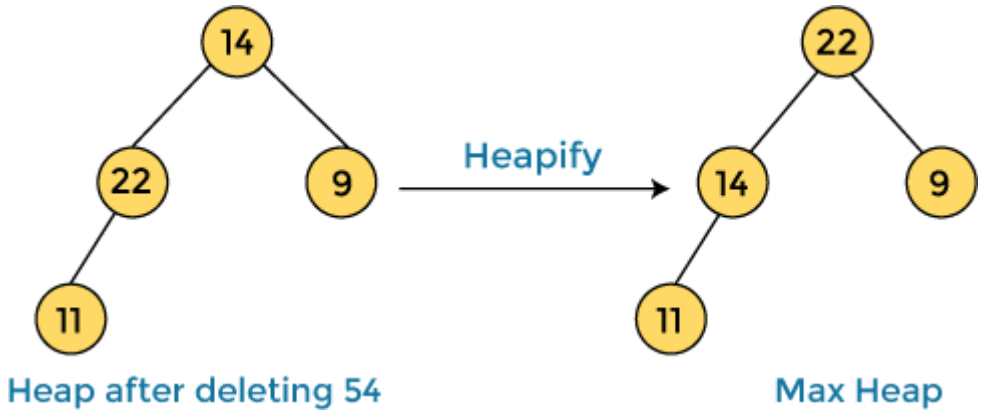


In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

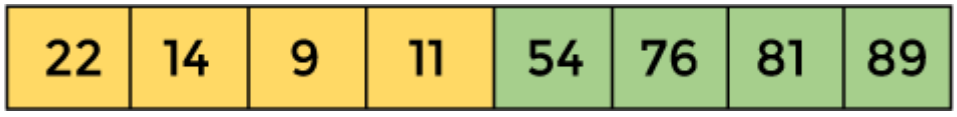


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are –

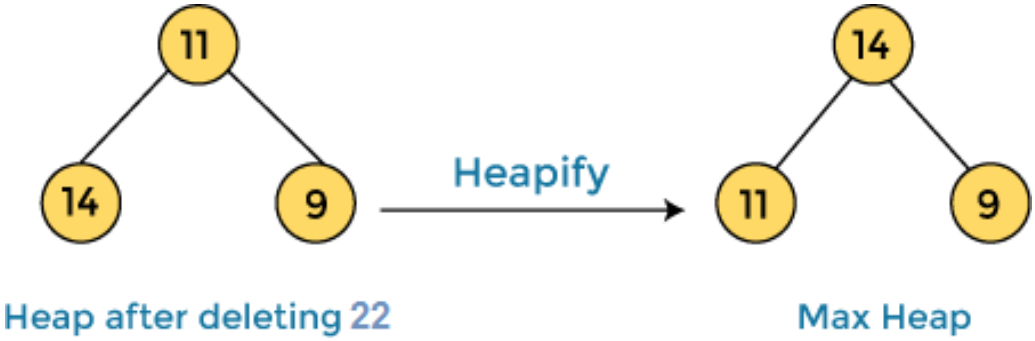
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



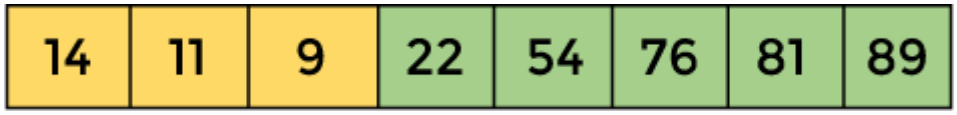
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are –



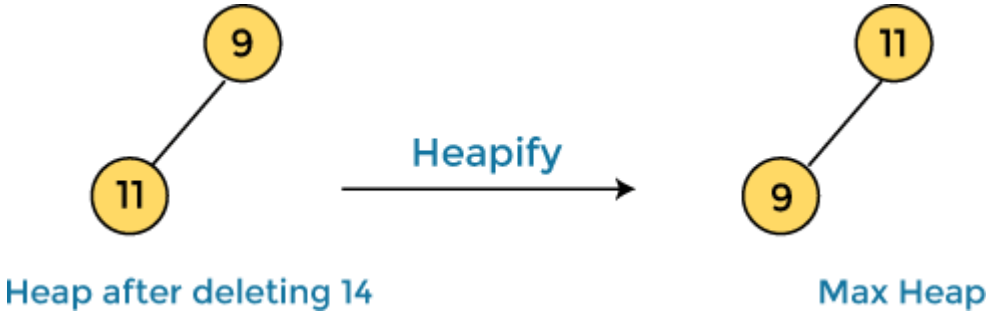
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



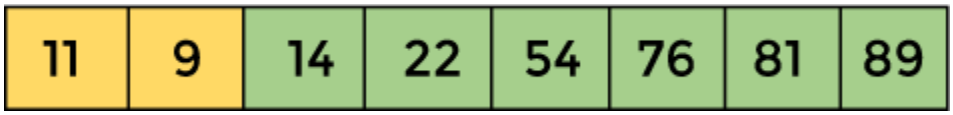
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting theroot element, we again have to heapify it to convert it into max heap.

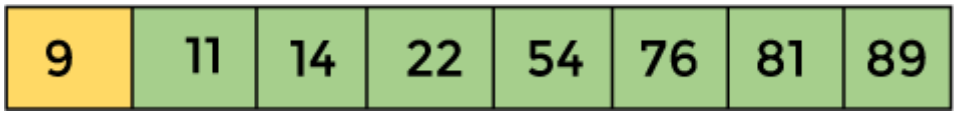


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are –



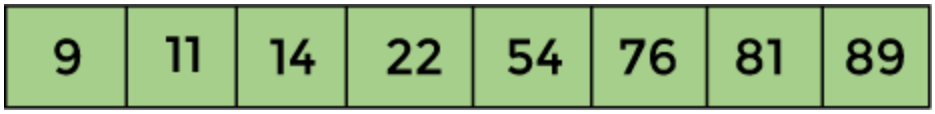
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting theroot element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are –



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Algorithm

Step 1 - Start

Step 2 - Construct a Binary Tree with given list of Elements.Step 3 -

Transform the Binary Tree into Max Heap.

Step 4 - Delete the root element from Max Heap using Heapify method.Step 5 -

Put the deleted element into the Sorted list.

Step 6 - Repeat the same until Max Heap becomes empty.Step 7 -

Display the sorted list

Step 8 – Stop

Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

Program

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
    void heapify(int a[], int n, int i)
{
int largest = i; // Initialize largest as rootint
    left = 2 * i + 1; // left child
int right = 2 * i + 2; // right child
// If left child is larger than root
if (left < n && a[left] > a[largest])
    largest = left;
// If right child is larger than root
if (right < n && a[right] > a[largest])
    largest = right;
// If root is not largestif
    (largest != i) {
// swap a[i] with a[largest]int
    temp = a[i];
```

```
a[i] = a[largest];
      a[largest] = temp;
```

```
heapify(a, n, largest);
}
}
/*Function to implement the heap sort*/
    void heapSort(int a[], int n)
{
for (int i = n / 2 - 1; i >= 0; i--)
    heapify(a, n, i);
// One by one extract an element from heapfor
    (int i = n - 1; i >= 0; i--) {
/* Move current root element to end*/
// swap a[0] with a[i]int
    temp = a[0];
a[0] = a[i]; a[i] =
    temp;

heapify(a, i, 0);
}
}
/* function to print the array elements */
    void printArr(int arr[], int n)
{
for (int i = 0; i < n; ++i)
{
printf("%d", arr[i]);
    printf(" ");
}
```

```
}
int main()
{
int a[] = {48, 10, 23, 43, 28, 26, 1};
int n = sizeof(a) / sizeof(a[0]);
printf("Before sorting array elements are - \n");
```

```
printArr(a, n);
    heapSort(a, n);
printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
return 0;
}
```

HASHING

- Hashing is an important data structure designed to solve the problem of efficiently finding and storing data in an array.
- For example, if you have a list of 20000 numbers, and you have given a number to search in that list- you will scan each number in the list until you find a match.
- It requires a significant amount of your time to search in the entire list and locate that specific number.
- This manual process of scanning is not only time-consuming but inefficient too.
- With hashing in the data structure, you can narrow down the search and find the number within seconds.

What is Hashing in Data Structure?

- Hashing in the data structure is a technique of mapping a large chunk of datainto small tables using a hashing function.
- It is also known as the message digest function. It is a technique that uniquely identifies a specific item from a collection of similar items.

- It uses hash tables to store the data in an array format. Each value in the array has assigned a unique index number.
- Hash tables use a technique to generate these unique index numbers for each value stored in an array format. This technique is called the hash technique.
- You only need to find the index of the desired item, rather than finding the data.
- With indexing, you can quickly scan the entire list and retrieve the item you wish.
- Indexing also helps in inserting operations when you need to insert data at a specific location. No matter how big or small the table is, you can update and retrieve data within seconds.
- Hashing in a data structure is a two-step process.
  - The hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.
  - It stores the data in a hash table. You can use a hash key to locate data quickly.

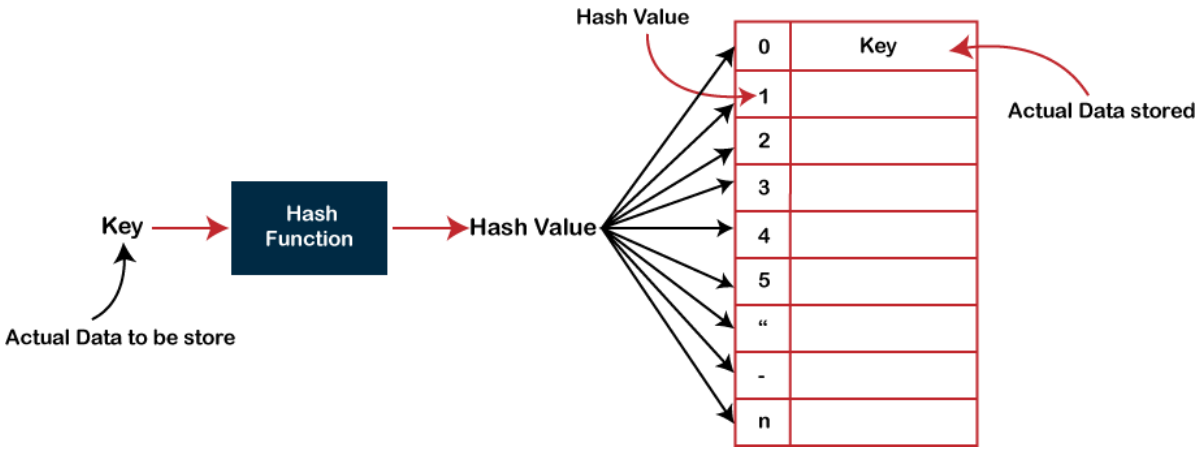
Hash Function

- The hash function in a data structure maps arbitrary size of data to fixed-sized data.
- It returns the following values: a small integer value (also known as hash value), hash codes, and hash sums.
- The has function must satisfy the following requirements:
  - A good hash function is easy to compute.
  - A good hash function never gets stuck in clustering and distributes keys evenly across the hash table.

o A good hash function avoids collision when two elements or itemsget assigned to the same hash value.

Hash Table

- Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.
- A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value.
- The hash table can be implemented with the help of an associative array.
- The efficiency of mapping depends upon the efficiency of the hash function used for mapping.
- **Drawback of Hash function** is that hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a **collision** because the hash function generates the same key of two differentvalues.



- There are three ways of calculating the hash function:
  - Division method
  - Folding method
  - Mid square method

Collision

- When the two different values have the same value, then the problem occurs between the two values, known as a collision.
- For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

h(6) = 6%10 = 6

The index is 6 at which the value is stored.

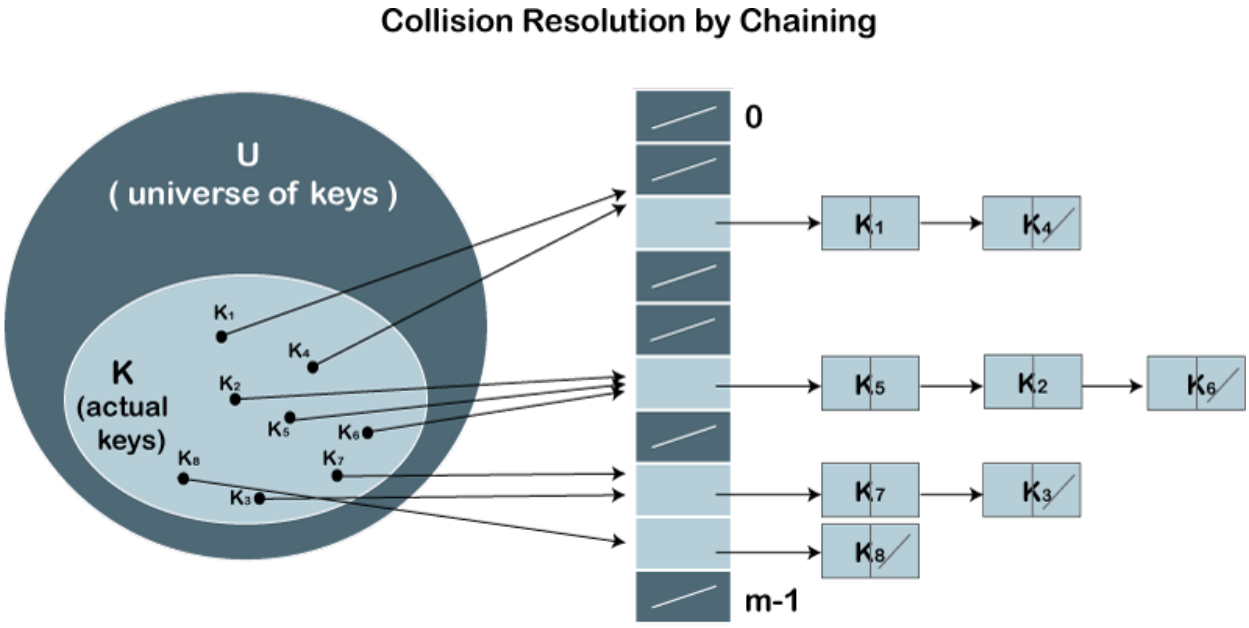
- In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

h(26) = 26%10 = 6

- Therefore, two values are stored at the same index, i.e., 6, and this leads to thecollision problem. To resolve these collisions, we have some techniques known as collision techniques.
- The following are the collision techniques:
  - Open Hashing: It is also known as closed addressing.
  - Closed Hashing: It is also known as open addressing.

Open Hashing

- In Open Hashing, one of the methods used to resolve the collision is knownas a chaining method.



- **Let's first understand the chaining to resolve the collision.**

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

In this case, we cannot directly use $h(k) = k_i/m$ as $h(k) = 2k+3$

- The index of key value 3 is: index =

  $h(3) = (2(3)+3)\%10 = 9$

The value 3 would be stored at the index 9.

- The index of key value 2 is:

index = h(2) = (2(2)+3)%10 = 7

The value 2 would be stored at the index 7.

- The index of key value 9 is: index =

  h(9) = (2(9)+3)%10 = 1

The value 9 would be stored at the index 1.

- The index of key value 6 is: index =

  h(6) = (2(6)+3)%10 = 5

The value 6 would be stored at the index 5.

- The index of key value 11 is: index =

  h(11) = (2(11)+3)%10 = 5

The value 11 would be stored at the index 5. Now, we have two values(6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

- The index of key value 13 is: index =

  h(13) = (2(13)+3)%10 = 9

The value 13 would be stored at index 9. Now, we have two values (3,13) stored at the same index, i.e., 9. This leads to the collision problem, so we

will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

- The index of key value 7 is: index =

  h(7) = (2(7)+3)%10 = 7

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

- The index of key value 12 is: index =

  h(12) = (2(12)+3)%10 = 7

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	((2*3)+3)%10 = 9
2	((2*2)+3)%10 = 7
9	((2*9)+3)%10 = 1
6	((2*6)+3)%10 = 5

11	((2*11)+3)%10 = 5
13	((2*13)+3)%10 = 9
7	((2*7)+3)%10 = 7
12	((2*12)+3)%10 = 7

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

Linear Probing

- Linear probing is one of the forms of open addressing.
- As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell.
- In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.
- **Let's understand the linear probing through an example.**

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

- The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by anotherkey value, i.e., 6. When linear probing is applied, the nearest empty cell tothe index 5 is 6; therefore, the value 11 will be added at the index 6.
- The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.
- The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.
- The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.
- The final hash table would be:

0	**13**
1	**9**
2	**12**
3	
4	
5	**6**

6	11
7	2
8	7
9	3

Quadratic Probing

- In case of linear probing, searching is performed linearly.
- In contrast, quadratic probing is an open addressing technique that usesquadratic polynomial for searching until a empty slot is found.
- It can also be defined as that it allows the insertion ki at first free locationfrom **$(u+i^2)\%m$ where i=0 to m-1.**

Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h(k) = 2k+3

- The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.
- The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

Index= $(5+0^2)\%10 = 5$

When i=1

Index = $(5+1^2)\%10 = 6$

Since location 6 is empty, so the value 11 will be added at the index 6.

- The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(9+0^2)\%10 = 9$

When i=1

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

- The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

Index = $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

- The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, weapply the Quadratic probing technique on 12 to determine the free location.

When i=0

Index= $(7+0^2)\%10 = 7$

When i=1

Index = $(7+1^2)\%10 = 8$

When i=2

Index = $(7+2^2)\%10 = 1$

When i=3

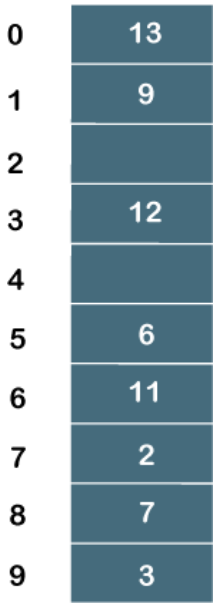
Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

- The final hash table would be:



Double Hashing

- Double hashing is an open addressing technique which is used to avoid the collisions.
- When the collision occurs then this technique uses the secondary hash of thekey.
- It uses one hash value as an index to move forward until the empty location isfound.
- In double hashing, two hash functions are used.
- Suppose $h_1(k)$ is one of the hash functions used to calculate the locationswhereas $h_2(k)$ is another hash function.
- It can be defined as "insert k_i at first free place from *(u+v*i)%m* where i=(0to m-1)".
- In this case, u is the location computed using the hash function and v is equalto ($h_2(k)$%m).
- Consider the same example that we use in quadratic probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and h1(k) =

2k+3

h2(k) = 3k+1

ke	Location (u)	v
3	((2*3)+3)%10 = 9	-
2	((2*2)+3)%10 = 7	-
9	((2*9)+3)%10 = 1	-
6	((2*6)+3)%10 = 5	-
11	((2*11)+3)%10 = 5	(3(11)+1)%10 =4
13	((2*13)+3)%10 = 9	(3(13)+1)%10 = 0
7	((2*7)+3)%10 = 7	(3(7)+1)%10 = 2
12	((2*12)+3)%10 = 7	(3(12)+1)%10 = 7

- As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.
- On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. Whenthe key value is 11, the value of v is 4.
- Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (5+4*0)%10 =5

When i=1

Index = (5+4*1)%10 = 9

When i=2

Index = (5+4*2)%10 = 3

- Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.
- The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.
- Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (9+0*0)%10 = 9

- We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.
- The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.
- Now, substituting the values of u and v in (u+v*i)%m

When i=0

Index = (7 + 2*0)%10 = 7

When i=1

Index = (7+2*1)%10 = 9

When i=2

Index = (7+2*2)%10 = 1

When i=3

Index = (7+2*3)%10 = 3

When i=4

Index = (7+2*4)%10 = 5

When i=5

Index = (7+2*5)%10 = 7

When i=6

Index = (7+2*6)%10 = 9

When i=7

Index = (7+2*7)%10 = 1

When i=8

Index = (7+2*8)%10 = 3

When i=9

Index = (7+2*9)%10 = 5

- Since we checked all the cases of i (from 0 to 9), but we do not find suitableplace to insert 7. Therefore, key 7 cannot be inserted in a hash table.

- The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.
- Now, substituting the values of u and v in (u+v*i)%m

When i=0

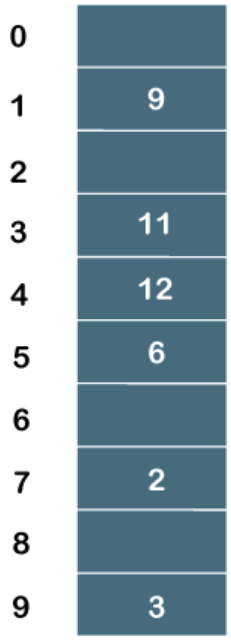
Index = (7+7*0)%10 = 7

When i=1

Index = (7+7*1)%10 = 4

- Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.

- The final hash table would be:



Types of Hash functions

There are many hash functions that use numeric or alphanumeric keys. This articlefocuses on discussing different hash functions:

1. **Division Method.**
2. **Mid Square Method.**
3. **Folding Method.**
4. **Digit Analysis.**

Let's begin discussing these methods in detail.

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hashfunction divides the value k by M and then uses the remainder obtained.

Formula:

h(K) = k mod M

Here,
k is the key value, and
M is the size of the hash table.

It is best suited that **M** is a prime number as that can make sure the keys are moreuniformly distributed. The hash function is dependent upon the remainder of a division.

Example:

k = 12345

M = 95

h(12345) = 12345 mod 95

= 90

k = 1276

M = 11

h(1276) = 1276 mod 11

=

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to chose value of M.

2. Mid Square Method:

The mid square method is a very good hashing method. It involves two steps tocompute the hash value-

1. Square the value of the key k i.e. k^2

2. Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,
k is the key value.

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits arerequired to map the key to the memory location.

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottomdigit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is ofbig size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,….,kn**, where each part has the same number of digits except for the last part that can havelesser digits than the other parts.
2. Add the individual parts. The hash value is obtained by ignoring the lastcarry if any.

Formula:

k = k1, k2, k3, k4, ….., kn

s = k1+ k2 + k3 + k4 +….+ knh(K)=

s

Here,

s is obtained by adding the parts of the key **k**

Example:

k = 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

= 12 + 34 + 5

= 51

h(K) = 51

Note:

The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have twodigits except for the last part that can have a lesser number of digits.

4. Digit Analysis:

- The last method we will examine, digit analysis, is used with static files. A static file is one in which all the identifiers are known in advance.
- Using this method, we first transform the identifiers into numbers using someradix, r.
- We then examine the digits of each identifier, deleting those digits that have the most skewed distributions. We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hashtable.
- The digits used to calculate the hash address must be the same for all identifiers and must not have abnormally high peaks or valleys (the standard deviation must be small).
- In general, a hash function is one that accepts an input of arbitrary length and distribution and transforms it (typically via a technique that non-recoverably discards information) into an output that is of fixed size and evenly distributed.

- So (for example), if our inputs were 1938m, 3391i, 3091b, 4903a, 4930a, 6573b, and 4891c, we analyze the digits: there are three 1s, six 9s, seven 3s, two 8s, one m, one i, three 0s, two bs, three 4s, two as, one 6, one 5, one 7, and one c.
- So we might decide to eliminate 1, 8, m, i, 0, b, 4, a, 6, 5, 7 and c from the inputs to produce 93, 339, 39, 93, 93, 3, 9.
- Let's further say we only want a one-character hash, so we take only the firstcharacter: 9, 3, 3, 9, 9, 3, 9. We now have a hash function that hashes these seven identifiers more-or-less evenly into one of two buckets.

OVERFLOW CONDITION IN HASHING

- An overflow occurs at the time of the home bucket for a new pair (key, element) is full. There are two methods for detecting collisions and overflows in a static hash table; each method using different data structure to represent the hash table.

Two Methods:

Linear Open Addressing (Linear probing) Chaining

- Open addressing is performed to ensure that all elements are stored directly into the hash table, thus it attempts to resolve collisions implementing various methods.
- Linear Probing is performed to resolve collisions by placing the data into the next open slot in the table.

CONTENT BEYOND SYLLABUS

AVL Trees

Tree is one of the most important data structure that is used for efficiently performing operations like insertion, deletion and searching of values. However, while working with a large volume of data, construction of a well-balanced tree for sorting all d

is stored as a tree, and the actual volume of data being used continually changes through the insertion of new data and deletion of existing data. You will find in some cases where the NULL link to a binary tree

links is called as threads and hence it is possible to perform traversals, insertions, deletions without using either stack or recursion. In this chapter, you will learn about the Height balance tree which is also known as the AVL tree.

What is AVL Tree:

AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by

and Landis and henc

Binary Tree.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this

II Semester :: Data Structures :: AVL Trees ::

AVL Trees

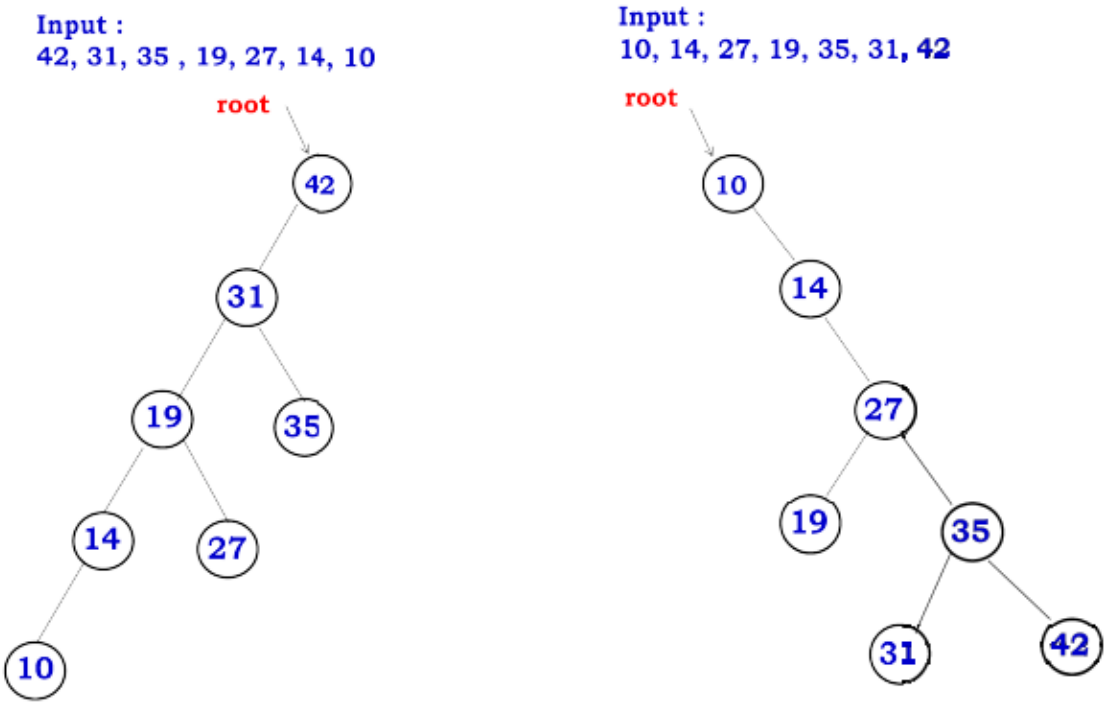
Height Balanced Trees

Tree is one of the most important data structure that is used for efficiently performing operations like insertion, deletion and searching of values. However, while working with a large volume of data, construction of a balanced tree for sorting all data is not feasible. Thus only useful data

is stored as a tree, and the actual volume of data being used continually

changes through the insertion of new data and deletion of existing data. You will find in some cases where the NULL link to a binary tree links is called as threads and hence it is possible to perform traversals, insertions, deletions without using either stack or recursion. In this chapter, you will learn about the Height balance tree which is also known AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velsk and hence given the short form as AVL tree or Balanced

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



t is observed that BST's worst

search algorithms, that is O(n). In real

pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor

height balancing binary searc

and the right sub-trees and assures that the difference is not more than 1.

This difference is called the

Consider the following trees.

and the next two trees are not balanced

In the second tree, the left subtree of

has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the diffe

again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left

II Semester :: Data Structures :: AVL Trees ::

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the

Named after their inventor Adelson, Velsky and Landis, AVL trees

height balancing binary search tree. AVL tree checks the height of the left trees and assures that the difference is not more than

This difference is called the Balance Factor.

Consider the following trees. Here we see that the first tree is balanced

next two trees are not balanced −

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree has height 2 and the left is missing, so it is 0, and the diffe

again. AVL tree permits difference (balance factor) to be only 1.

= height(left-subtree) − height(right-subtree)

case performance is closest to linear

time data, we cannot predict data

pattern and their frequencies. So, a need arises to balance out the AVL trees.

and the next two trees are not balanced –