Introduction to Data Structure

CONTENTS

Hours: 6

Marks: 8

1.1 Basic Terminology

1. Elementary data structure organization

i. Data can be organized in many ways and data structures is one of these ways.

ii. It is used to represent data in the memory of the computer so that the processing of data can be done in easier way.

iii. Data structures is the logical and mathematical model of a particular organization of data.

2. Classification of data structure

(Question: Explain the classification of Data Structure – 2 Marks)

The data structures can be of the following types:

i. Linear Data structures: In these data structures the elements form a sequence. Such as Arrays, Linked Lists, Stacks and Queues are linear data structures.

ii. Non-Linear Data Structures: In these data structures the elements do not form a sequence. Such as Trees and Graphs are non-linear data structures.

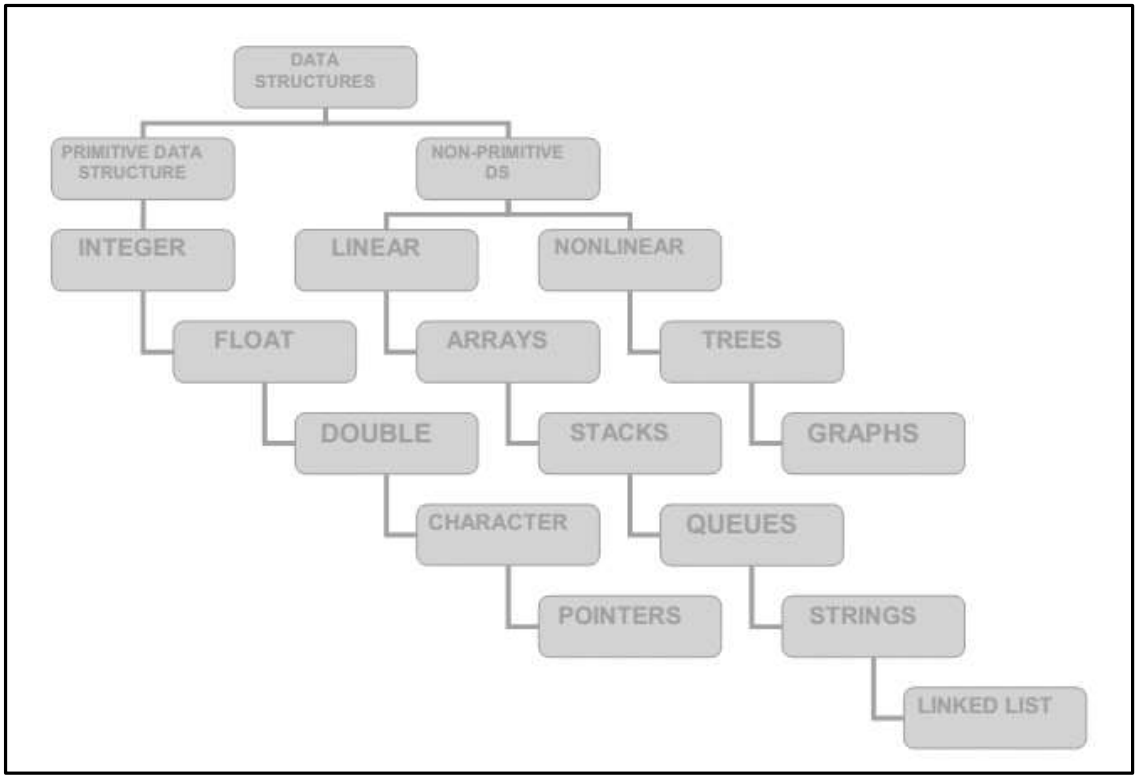


Figure 1.1

1.2 Operations on data structures

(Question: List and explain the various operations performed on data structures- 4 Marks)

1. **Traversing, Inserting, deleting**

   The following four operations play a major role in this text.

   i. **Traversing**: Accessing each record exactly once so that certain items in the record may be processed.

   ii. **Inserting**: adding a new record to the structure.

   iii. **Deleting**: Removing a record from the structure. Sometimes two or more of the operations may be used in a given situation; for example we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

2. **Searching, sorting, merging**

   i. **Searching**: Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

   ii. **Sorting:** Arranging the records in some logical order (for example alphabetically according to some Name key, or in numerical order according to some Number key. such as account number.

   iii. **Merging:** Combining the records in two different sorted files into a single sorted file

1.3 Different Approaches to designing an algorithm

(Question: List two points each to explain the concept of top down and bottom up approaches – 4 Marks)

1. **Top-Down approach**

   i. This approach divides the problem in to manageable segments.

   ii. This technique generates diagrams that are used to design the system.

   iii. Frequently several alternate solutions to the programming problem are developed and compared during this phase.

2. **Bottom-up approach**

   i. The Bottom-up approach is an older approach which gives early emphasis on coding.

   ii. Since the programmer does not have a master plan for the project, the resulting program may have many error ridden segments.

1.4 Complexity

1. Time complexity

(Question: Explain time complexity of an algorithm with operation count and step count- 4 Marks)

Time Complexity is divided in to THREE Types.

i. **Best Case Time Complexity**: Efficiency of an algorithm for an input of size N for which the algorithm takes the smallest amount of time.

ii. **Average Case Time Complexity**: It gives the expected value of T(n). Average case is used when worst case and best case doesn't gives any necessary information about algorithm's behavior, then the algorithm's efficiency is calculated on Random input.

iii. **Worst Case Time Complexity**: efficiency of an algorithm for an input of size N for which the algorithm takes the longest amount of time.

Time complexity is calculated on the basis of

i. **Operation Count**: Find the basic operation a= a * b; this code takes 1 unit of time

```
For(i=0;i<n;i++)
{
  for(j=0;j<n;j++)
       if(j==5)
         {
            printf("Internal Instruction");
         }
}
```

Figure 1.2

ii. **Step Count**

Example

Statement	Step Count
x = a + b;	1
for(i =1; i <= n; i++)	n

for(i=1; i <= n; i++)
for(j=1; j <= n; j++)
x = a + b;
	n2

Table 1.1

Example

```
int sum (int a[], int n)
{
     int i , sum=0;
     sum-count ++;
     for (i=0; i<n; i++)
     {
          for-count ++;
          sum = sum + a[i];
     }
     for-count++;
     return-count ++;
     return sum;
}
```

Figure 1.3

The Step count for "sum" = 1

The step count for, "for statement" = n + 1

The step count for, "assignment" = n

The step count for, "return" = 1

Total steps = 2n + 3

2. Space complexity

(Question: Explain the concept of space complexity – 4 Marks)

i. Specifies the amount of temporary storage required for running the algorithm.

ii. We do not count storage required for input and output when we are working with the same type of problem and with different algorithms.

iii. Space needed by algorithm consists of the following component.

   a. The fixed static part: it is independent of characteristics ex: number of input/output. This part includes the instruction space (i.e. space for code). Space for simple variables, space for constants, and fixed size component variables. Let Cp be the space required for the code segment (i.e. static part)

   b. The variable dynamic part: that consists of the space needed by component variables whose size is dependent on the particular problem instance at runtime. i.e. space needed by reference variables, and the recursion stack space. Let Sp be the space for the dynamic part.

   c. Overall space required by program: S(p)= Cp + Sp

Finding the sum of three numbers:

```
#include<stdio.h>
Void main()
{
 int x, y, z, sum;
Printf("Enter the three numbers");
Scanf("%d, %d, %d", &x, &y, &z);
Sum= x + y + z;
Printf("the sum = %d", sum);
}
```

Figure 1.4

iv. In the above program there are no instance characteristics and the space needed by x, y, z and sum is independent of instance characteristics. The space for each integer variable is 2. We have 4 integer variables and space needed by x, y, z and sum are 4 * 2 = 8 bytes.

S (p) = Cp + Sp

S (p) = 8 + 0

S (p) = 8

1.5 Big 'O' Notation

Asymptotic Notation

The asymptotic behavior of a function is the study of how the value of a function varies for larger values of "n" where "n" is the size of the input. Using asymptotic behavior we can easily find the time efficiency of an algorithm.

Different asymptotic notations are:

(Question: Explain the concept of Big 'O' asymptotic notation - 4 marks)

i. **Big O (big oh)**: it is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of longest amount of time it could possibly take for the algorithm to complete. Let f(n) is the time efficiency of an algorithm. The function f(n) is said to be Big-oh of g(n), denoted by

f(n) € O(g(n)) OR f(n) ≈ O(g(n)).

Such that there exist a +ve constant "c" and +ve integer n0 satisfying the constraint.

f(n) ≤ c * g(n) for all n≥ n0

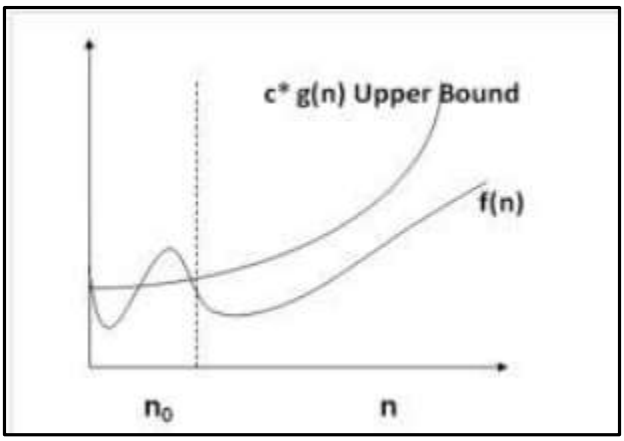


Figure 1.5

(Question: Explain the concept of Big Omega asymptotic notation - 4 marks)

ii. **Ω notation (Big-Omega)**: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be Big-Omega of g(n) which is denoted by

f(n) € Ω(g(n)) OR f(n) ≈ Ω(g(n))

such that there exist a +ve constant "c" and non-negative integer n0 satisfying the constraint

f(n) ≥ c * g(n) for all n ≥ n0

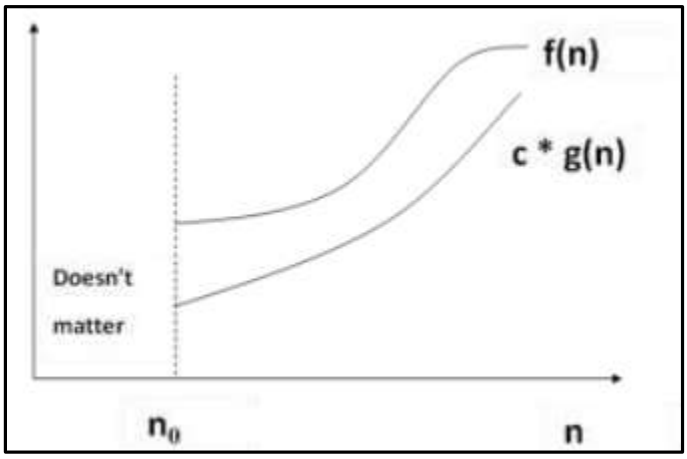


Figure 1.6

(Question: Explain the concept of Theta asymptotic notation - 4 marks)

iii. **Θ notation (Theta)**: Let f(n) be the time complexity of an algorithm. The function f(n) is said to be Big-Theta of g(n) which is denoted by

f(n) € θ(g(n)) OR f(n) ≈ θ (g(n))

such that there exist a +ve constant "c1, c2" and non-negative integer n0 satisfying the constraint c2g(n) ≤ f(n) ≥ c1g(n) for all n ≥ n0.

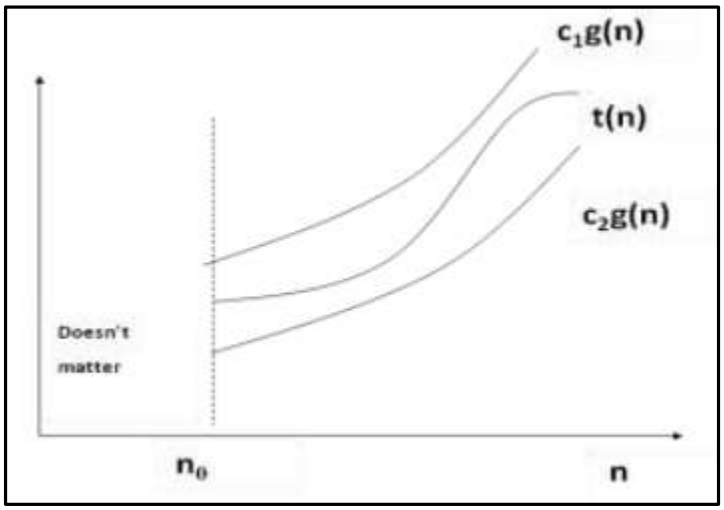


Figure 1.7

Sorting & Searching

CONTENTS

Hours: 10

Marks: 16

2.1 Sorting Techniques

1. Introduction

i. Sorting is a process that organizes a collection of data into either ascending or descending order.

ii. An internal sort requires that the collection of data fit entirely in the computer's main memory.

iii. We use an external sort when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

iv. We will analyze only internal sorting algorithms.

v. Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.

2. Selection Sort

(Question: Explain the concept of selection sort with program, advantages and its disadvantages – 8 Marks)

i. The list is divided into two sub lists, sorted and unsorted, which are divided by an imaginary wall.

ii. We find the smallest i.e the minimum element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.

iii. After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.

iv. Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.

v. A list of n elements requires n-1 passes to completely rearrange the data.

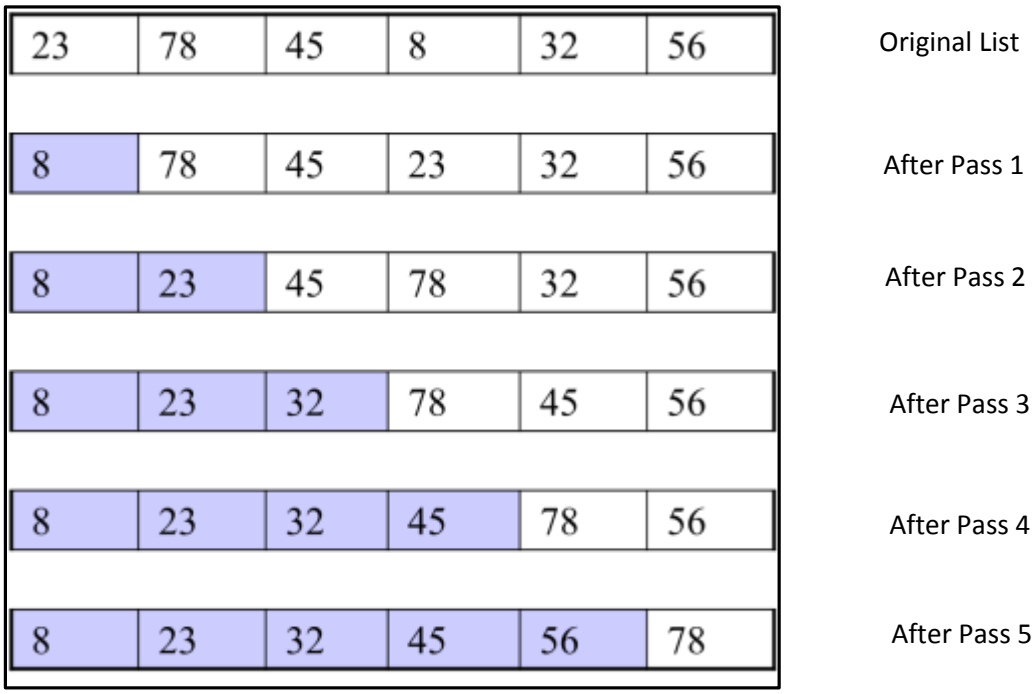


Figure 1: Selection Sort

Advantages

i. The main advantage of the selection sort is that it performs well on a small list.

ii. It is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list.

Disadvantages

i. The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.

ii. Its performance is easily influenced by the initial ordering of the items before the sorting process.

WAP to sort the elements using selection sort

```
#include <stdio.h>
void main()
{
   int array[100], n, i, j, min, swap;

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

   printf("Enter %d integers\n", n);

   for ( i = 0 ; i < n ; i++ )
      scanf("%d", &array[i]);
    i=o
while(i<n)
{
min = a[i];
for(j=i+1; j < 5; j++)
{
 if(min > a[j])
 {
  min = a[j];
        temp = a[i];
        a[i] = a[j];
        a[j]  temp;
 }
}
 i++;
}

   printf("Sorted list in ascending order:\n");
    for ( i = 0 ; i < n ; i++ )
      printf("%d\n", array[i]);
}
```

3. Insertion Sort

(Question: Explain the concept of insertion sort with program, advantages and its disadvantages – 8 Marks)

i. It always maintains two zones in the array to be sorted: sorted and unsorted.

ii. At the beginning the sorted zone consist of one element (the first element of array that we are sorting).

iii. On each step the algorithms expand the sorted zone by one element inserting the first element from the unsorted zone in the proper place in the sorted zone and shifting all larger elements one slot down.

Advantages

i. The insertion sorts repeatedly scans the list of items, each time inserting the item in the unordered sequence into its correct position.

ii. The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

Disadvantage

i. The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms.

ii. With n-squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.

iii. Therefore, the insertion sort is particularly useful only when sorting a list of few items.

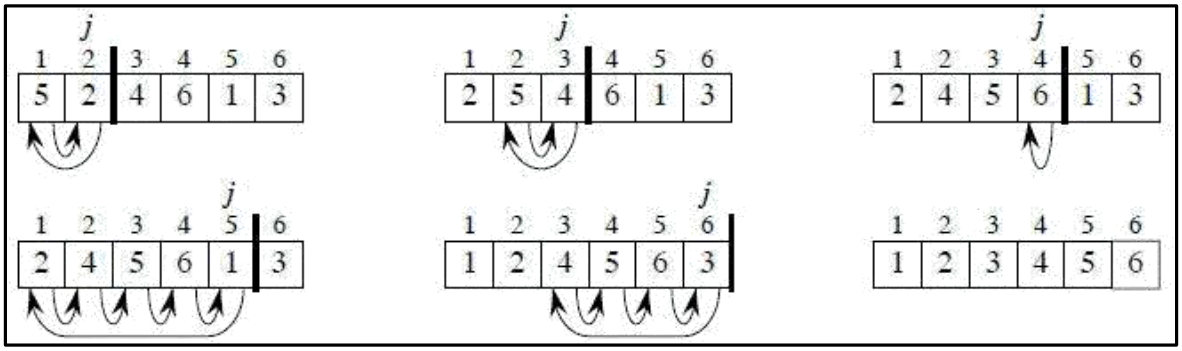


Figure 2: Insertion Sort

WAP to sort the elements using insertion sort.

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int i, j, temp,a[10];
  printf("enter the elements ➔");
  for(i = 0; i <= 10; i ++)
  {
    scanf("%d", &a[i]);
  }
  for (i = 0; i <= 10; i++)
  {
    for(j = i +1;j <= 10; j++)
    {
        if(a[i] > a[j])
        {
          temp = a[i];
          while( j != i)
          {
                a[j] = a[j-1];
                j --;
          }
          a[i] = temp;
        }
    }
  }
printf (" the sorted array is ➔");
for( i = 0; i <=10; i++)
        printf("%d \n", a[i]);
}
```

4. Bubble Sort

(Question: Explain the concept of bubble sort with program, advantages and its disadvantages – 8 Marks)

i. Bubble sort algorithms cycle through a list, analyzing pairs of elements from left to right, or beginning to end.
ii. If the leftmost element in the pair is less than the rightmost element, the pair will remain in that order. If the rightmost element is less than the leftmost element, then the two elements will be switched.
iii. This cycle repeats from beginning to end until a pass in which no switch occurs.

Advantages

i. The bubble sort requires very little memory other than that which the array or list itself occupies.
ii. The bubble sort is comprised of relatively few lines of code.
iii. With a best-case running time of O(n), the bubble sort is good for testing whether or not a list is sorted or not. Other sorting methods often cycle through their whole sorting sequence, which often have running times of O(n^2) or O(n log n) for this task.
iv. The same applies for data sets that have only a few items that need to be swapped a few times.

Disadvantages

i. The main disadvantage of the bubble sort method is the time it requires.
ii. With a running time of O (n^2), it is highly inefficient for large data sets.

WAP to sort the elements in the array using bubble sort.

```
#include <stdio.h>

int main()
{
  int array[100], n, i, j, temp;

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

  printf("Enter %d integers\n", n);

  for (i = 0; j < n; i++)
    scanf("%d", &array[i]);

  for (i = 0 ; i < ( n - 1 ); i++)
  {
    for (j = i+1 ; j < n; j++)
    {
      if (array[i] > array[j]) /* For decreasing order use < */
      {
        temp       = array[i];
        array[i]   = array[j];
        array[j] = temp;
      }
    }
  }

  printf("Sorted list in ascending order:\n");

  for ( c = 0 ; c < n ; c++ )
     printf("%d\n", array[c]);

  return 0;
}
```

5. Merge Sort

(Question: Explain the concept of merge sort with program, advantages and its disadvantages – 8 Marks)

i. It follows the fundamental of dividing the elements and sorting, as sorting is easier in smaller bits of elements.
ii. If the sub list is 1 in length, then the sub list is sorted.
iii. If the list is more than one in length, then divide the unsorted list into roughly two parts.
iv. Keep dividing the sub lists until each one is only 1 item in length.
v. Now merge the sub-lists back into a list twice their size, at the same time sorting each items into order.
vi. Keep merging the sub list until the list is complete once again.

Advantage

i. Good for sorting slow-access data.
ii. It is excellent for sorting data that are normally accessed sequentially.

Disadvantage

i. If the list is N long, then it takes 2*N memory space to sort the elements.
ii. For large data its time and space complexity is of the order of nlogn.

WAP to sort the elements in the array using merge sort.

```
#include<stdio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){

   int merge[MAX],i,n;

   printf("Enter the total number of elements: ");
   scanf("%d",&n);

   printf("Enter the elements which to be sort: ");
   for(i=0;i<n;i++){
      scanf("%d",&merge[i]);
   }
   partition(merge,0,n-1);
```

```
    printf("After merge sorting elements are: ");
    for(i=0;i<n;i++){
        printf("%d ",merge[i]);
    }

   return 0;
}

void partition(int arr[],int low,int high){

    int mid;

    if(low<high){
        mid=(low+high)/2;
        partition(arr,low,mid);
        partition(arr,mid+1,high);
        mergeSort(arr,low,mid,high);
    }
}
void mergeSort(int arr[],int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

        if(arr[l]<=arr[m]){
            temp[i]=arr[l];
            l++;
        }
        else{
            temp[i]=arr[m];
            m++;
        }
        i++;
    }
```

```
    if(l>mid){
        for(k=m;k<=high;k++){
            temp[i]=arr[k];
            i++;
        }
    }
    else{
        for(k=l;k<=mid;k++){
            temp[i]=arr[k];
            i++;
        }
    }

    for(k=low;k<=high;k++){
        arr[k]=temp[k];
    }
}
```

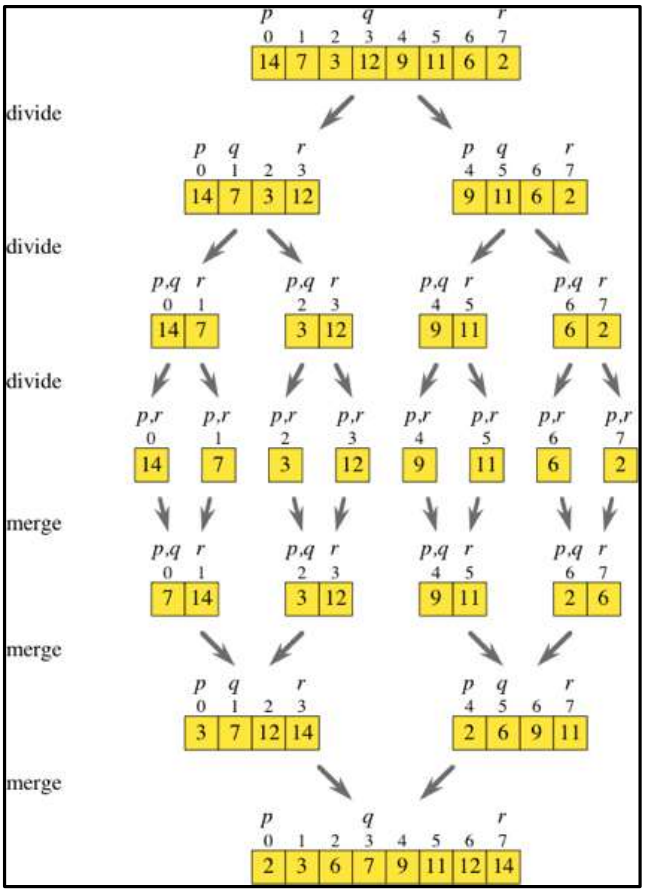


Figure 3: Merge Sort

6. Radix Sort

(Question: Explain the concept of radix sort the given series of numbers – 8 Marks)

i. The array for the bucket sort is divided in to 10 buckets from 0 to 9.
ii. This sort is commonly also known as bucket sort.
iii. The numbers are sorted on the LSB (Least Significant Bit) of any number for the first pass.
iv. The graphical representation of the same is as shown below

Bucket	0	1	2	3	4	5	6	7	8	9
Content		1 81	–	–	64 4	25	36 16	–	–	9 49

Figure 4: Radix Sort Pass 1

Pass 1: 1, 81, 64, 4, 25, 36, 16, 9, and 49.

v. This sort is implemented for the good functionality for a three bit number.
vi. After the Pass 1 print the list and continue with the same on the middle bit and put in the respective buckets.

Bucket	0	1	2	3	4	5	6	7	8	9
Content	01 04 09	16	25	36	49		64		81	

Figure 5: Radix Sort Pass 2

vii. Then print the pass 2.
Pass2: 01, 04, 09, 16, 25, 36, 49, 64, 81

7. Shell Sort

(Question: Explain the concept of shell sort the given series of numbers – 8 Marks)

i. The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort.
ii. The unique way that these sub lists are chosen is the key to the shell sort.
iii. Instead of breaking the list into subsists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sub list by choosing all items that are i items apart.
iv. This can be seen in Figure 6. This list has nine items.
v. If we use an increment of three, there are three sub lists, each of which can be sorted by an insertion sort.
vi. After completing these sorts, we get the list shown in Figure 7.
vii. Although this list is not completely sorted, something very interesting has happened. By sorting the sub lists, we have moved the items closer to where they actually belong.

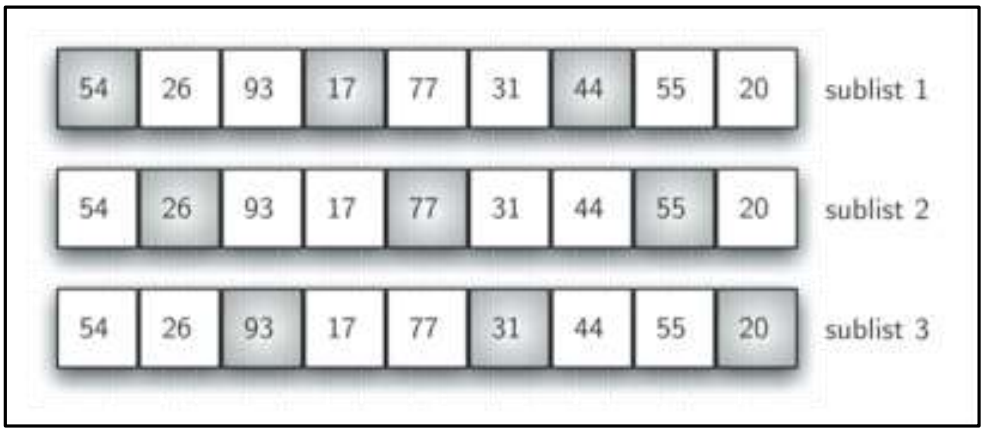


Figure 6: Shell sort with increment of 3

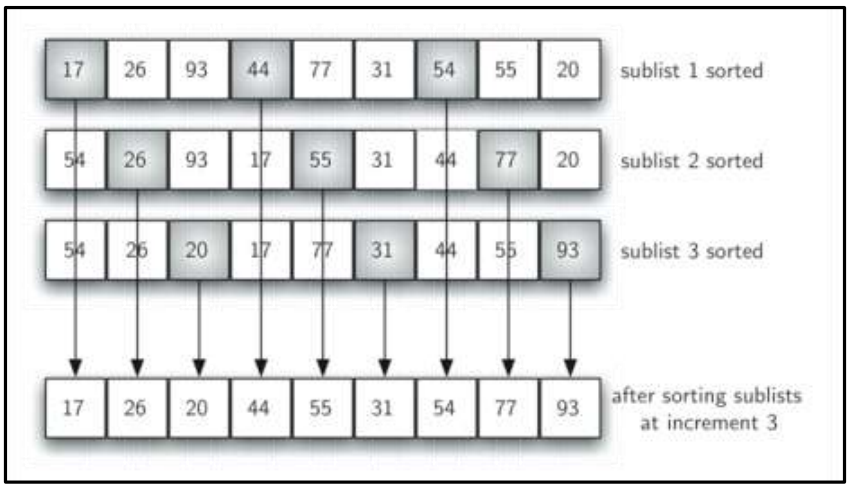


Figure 7: Shell Sort after sorting each sub list

viii. Figure 8 shows a final insertion sort using an increment of one; in other words, a standard insertion sort.

ix. Note that by performing the earlier sub list sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order.

x. For this case, we need only four more shifts to complete the process.

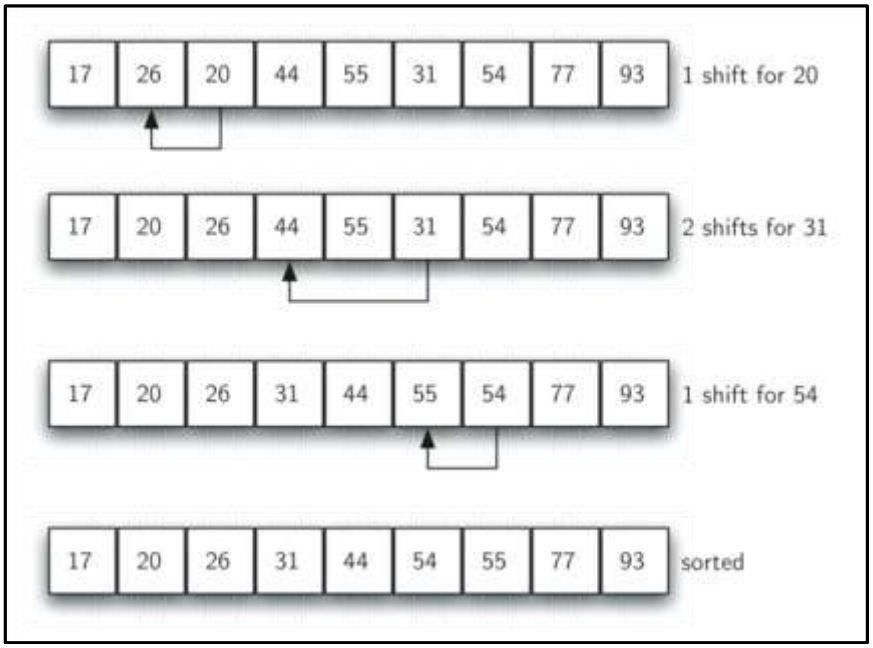


Figure 8: sorted list after increment of 1

8. Quick Sort

(Question: Explain the concept of quick sort the given series of numbers – 8 Marks)

i. The quicksort algorithm partitions an array of data into items that are less than the pivot and those that are greater than or equal to the pivot item.
ii. The first step is to create the partitions, in which a random pivot item is created, and the partition algorithm is applied to the array of items.
iii. This initial step is the most difficult part of the Quicksort algorithm.

Advantages

i. The efficient average case compared to any aforementioned sort algorithm, as well as the elegant recursive definition, and the popularity due to its high efficiency.
ii. The quick sort produces the most effective and widely used method of sorting a list of any item size.

Disadvantages

i. The difficulty of implementing the partitioning algorithm and the average efficiency for the worst case scenario, which is not offset by the difficult implementation.
ii. Quicksort works by "partitioning" the array to be sorted, then recursively sorting each partition. Here is the pseudocode.

```
procedure QuickSort (array A, int L, int N)
if L < N
M := Partition (A, L, N)
QuickSort (A, L, M - 1)
QuickSort (A, M + 1, N)
endif
endprocedure
```

```
Int function Partition (array A, int L, int N)
select M, where L <= M <= N
reorder A(L) ... A(N) so that I < M implies A(I) <= A(M), and I > M
implies A(I) >= A(M)
return M
endfunction
```

Name	Average	Worst	Space	Stable	Method
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection

Table 1: Time and Space complexity

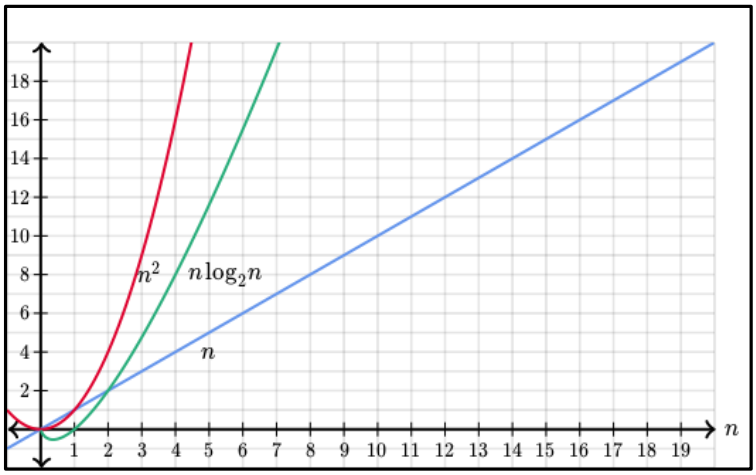


Figure 9: Graphical representation of complexity

2.2 Searching

1. Linear Search

(Question: Explain the concept of linear the given series of numbers – 8 Marks)

i. This is the simplest searching technique available.
ii. To search whether an element is present or absent in the array, we compare it with each and every other element in the array.
iii. If the element is found after comparison than it is printed as element found, else element not found or absent in the array.

Advantage

i. This is the simplest searching algorithm
ii. The space complexity is very less.
iii. Efficient for small data, if the dataset created is small.

Disadvantage

i. The time complexity increases.
ii. The number of comparisons become large as n, for the large set of array.

WAP to check whether an element is found in the array or not.

```
#include<stdio.h>
int main(){
   int a[10],i,n,m,c=0;
   printf("Enter the size of an array: ");
   scanf("%d",&n);
   printf("Enter the elements of the array: ");
   for(i=0;i<=n-1;i++){
      scanf("%d",&a[i]);
   }
   printf("Enter the number to be search: ");
   scanf("%d",&m);
   for(i=0;i<=n-1;i++){
      if(a[i]==m){
         c=1;
         break;
      }
   }
```

```
   if(c==0)
      printf("The number is not in the list");
   else
      printf("The number is found");

   return 0;
}
```

WAP to find the element present in the array and also print its location.

```
#include <stdio.h>
 int main()
{
  int array[100], search, c, n;
   printf("Enter the number of elements in array\n");
  scanf("%d",&n);
   printf("Enter %d integer(s)\n", n);
   for (c = 0; c < n; c++)
     scanf("%d", &array[c]);
   printf("Enter the number to search\n");
  scanf("%d", &search);
   for (c = 0; c < n; c++)
  {
     if (array[c] == search)     /* if required element found */
     {
        printf("%d is present at location %d.\n", search, c+1);
        break;
     }
  }
  if (c == n)
     printf("%d is not present in array.\n", search);

  return 0;
}
```

2. Binary Search

(Question: Explain the concept of binary search for the given series of numbers – 8 Marks)

i. A binary search divides a range of values into halves, and continues to narrow down the field of search until the unknown value is found.
ii. It is the classic example of a "divide and conquer" algorithm.

WAP to search the element in the array using binary search

```
#include <stdio.h>
void main ()
{
  int c, first, last, middle, n, search, array[100];
   printf("Enter number of elements\n");
  scanf("%d",&n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d",&array[c]);
   printf("Enter value to find\n");
  scanf("%d", &search);
   first = 0;
  last = n - 1;
  middle = (first+last)/2;
   while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d is not present in the list.\n", search);
}
```

Stacks

CONTENTS

Hours: 12

Marks: 18

3.1 Introduction to Stacks

1. Stack as Abstract Data Type

(Question: Explain stack as an abstract data type- 4 marks)

i. An abstract data type (ADT) consists of a data structure and a set of **primitive operations**. The main primitives of a stack are known as:
ii. The stack abstract data type is defined by the following structure and operations.
iii. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top."
iv. Stacks are ordered LIFO.
v. The stack operations are given below in the form of function are.
   a. Stack () creates a new stack that is empty. It needs no parameters and returns an empty stack.
   b. Push (item) adds a new item to the top of the stack. It needs the item and returns nothing.
   c. Pop () removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

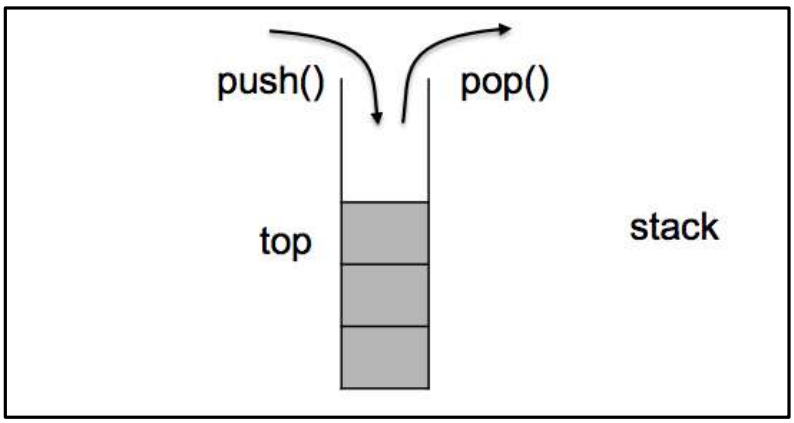


Figure 1: Stack Representation

2. Representation of Stack through arrays

(Question: Explain the concept of representing stack through arrays – 4 Marks)

i. Stack is represented using arrays as a simple array, with any variable name and the top of the stack is represented as top and initialized to -1 for the stack to be empty.
ii. Order produced by a stack:
iii. Stacks are linear data structures.

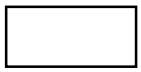
iv. This means that their contexts are stored in what looks like a line (although vertically) as shown in Figure 1.

v. This linear property, however, is not sufficient to discriminate a stack from other linear data structures.

vi. For example, an array is a sort of linear data structure. However, you can access any element in an array--not true for a stack, since you can only deal with the element at its top.

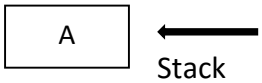
vii. One of the distinguishing characteristics of a stack, and the thing that makes it useful, is the order in which elements come out of a stack.

a) To start with stack empty:



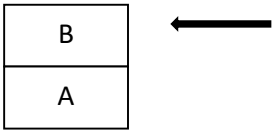
Stack empty

b) Now, let's perform Push(stack, A), giving:



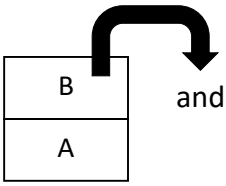
Stack

c) Again, another push operation, Push(stack, B), giving:



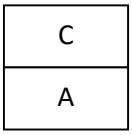
Stack

d) Now let's remove an item, letter = Pop(stack), giving:



Pop the element B from the Top of the stack and print it.

e) And finally, one more addition, Push(stack, C), giving:



Stack

f) In the last step pop out all the elements from the stack. The final printed stack will be B, C, A

3.2 Application of Stacks

1. Reversing the list

(Question: WAP to reverse the number using stack- 4 Marks)

i. The application of the stack is it prints any given series of numbers whether sorted or unsorted in reverse order.

ii. As the stack follows Last In First Order (LIFO), it gives the series of number in reverse order.

iii. Given below is the program to print the stack.

WAP to PUSH and POP the elements on to the stack, without using functions

```
#include<stdio.h>
void main()
{
        int stack[5], top = -1;
        while (top <5)
        {
                top++;
                printf ("\nenter the element for the stack➔");
                scanf ("%d", &stack[top]);
        }
        while (top!=0)
        {
                printf("\nthe popped element is➔ %d", st[top]);
                top = top -1;
        }
}
```

OUTPUT:
enter the element for the stack➔10
enter the element for the stack➔20
enter the element for the stack➔30
enter the element for the stack➔40
enter the element for the stack➔50
the popped element is➔50
the popped element is➔40
the popped element is➔30
the popped element is➔20
the popped element is➔10

2. Polish Notation

(Question: Explain the concept of polish notation- 4 Marks)

i. Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions known as Polish notation.

ii. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.
Infix notation: X + Y
Postfix notation (also known as "Reverse Polish notation"): X Y +
Prefix notation (also known as "Polish notation"): + X Y

iii. Operators are written in-between their operands.

iv. An expression such as A * (B + C) / D is usually taken to mean by multiplication first, then followed by division and then +.

3. Conversion of Infix to Postfix Expression

(Question: With example explain the steps to convert infix expression to postfix expression)

i. To convert an expression to its postfix notation signifies that operators have to be after the operands.

ii. We parenthesize the expression following the BODMAS rule, i.e., exponent (^) having the highest priority, followed by multiplication (*) and division (/), and the least priority to plus (+) and (-).

iii. If the two operators are with the same priority, i.e. (+) and (-), then we parenthesize them from left to right.

iv. Then we move the operators to the closest closing brace for post fix notation as shown below.

Examples: A * B + C / D ➔ ((A * B) + (C / D))

Postfix Expression is: AB*CD/+

WAP to convert infix expression to postfix notation

```
#define SIZE 50            /* Size of Stack */
#include <ctype.h>
char s[SIZE];
int top=-1;       /* Global declarations */

push(char elem)
{                          /* Function for PUSH operation */
   s[++top]=elem;
}
 char pop()
{                          /* Function for POP operation */
```

```
    return(s[top--]);
}
 int pr(char elem)
{               /* Function for precedence */
    switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
}
 main()
{                       /* Main Program */
    char infx[50],pofx[50],ch,elem;
    int i=0,k=0;
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
    while( (ch=infx[i++]) != '\0')
    {
        if( ch == '(') push(ch);
        else
            if(isalnum(ch)) pofx[k++]=ch;
            else
                if( ch == ')')
                {
                    while( s[top] != '(')
                        pofx[k++]=pop();
                    elem=pop(); /* Remove ( */
                }
                else
                {       /* Operator */
                    while( pr(s[top]) >= pr(ch) )
                        pofx[k++]=pop();
                    push(ch);
                }
    }
    while( s[top] != '#')     /* Pop from stack till empty */
        pofx[k++]=pop();
    pofx[k]='\0';          /* Make pofx as valid string */
    printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n",infx,pofx);
}
```

OUTPUT

Read the Infix Expression? A+B*C
Given Infix Expn: A+B*C Postfix Expn: ABC*+

4. Evaluating Postfix Expression

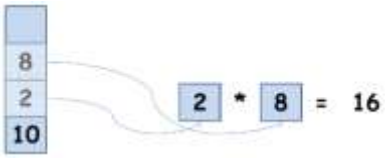
i. To evaluate the given postfix expression, let's start with the example
10 2 8 * + 3 -

ii. First, push (10) into the stack.

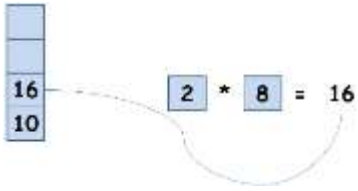
iii. As the next element is also an number we push (2) on to the stack.

iv. The next element is an number, so we again push (8) on to the stack.

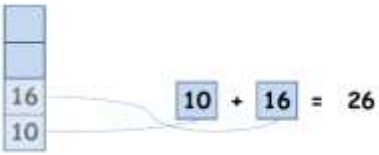
v. The next element is an operator (*), so we pop out the top two number from the stack and perform * on the two numbers.



vi. Then push the result on top of the stack.



vii. The next element is a + operator, so we perform the addition on the next two elements on the stack.



viii. Now we push (26) on to the stack.

ix. The next element is 3, so we push (3) on to the stack.

x. The last element is the operator –

WAP to convert the infix to postfix and evaluate the expression

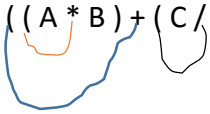
```
#include<stdio.h>
#include <ctype.h>
#define SIZE 50            /* Size of Stack */
char s[SIZE];
int top = -1; /* Global declarations */

push(char elem) { /* Function for PUSH operation */
 s[++top] = elem;
}

char pop() { /* Function for POP operation */
 return (s[top--]);
}

int pr(char elem) { /* Function for precedence */
 switch (elem) {
 case '#':
  return 0;
 case '(':
   return 1;
 case '+':
 case '-':
  return 2;
 case '*':
 case '/':
  return 3;
 }
}
pushit(int ele){                    /* Function for PUSH operation */
 s[++top]=ele;
}

int popit(){                    /* Function for POP operation */
 return(s[top--]);
}
 main()
{ /* Main Program */
  char infx[50], pofx[50], ch, elem;
 int i = 0, k = 0, op1, op2,ele;
 printf("\n\nRead the Infix Expression   ");
 scanf("%s", infx);
 push('#');
 while ((ch = infx[i++]) != '\0') {
  if (ch == '(')
```

```
   push(ch);
  else if (isalnum(ch))
   pofx[k++] = ch;
  else if (ch == ')') {
   while (s[top] != '(')
    pofx[k++] = pop();
   elem = pop(); /* Remove ( */
  } else { /* Operator */
   while (pr(s[top]) >= pr(ch))
    pofx[k++] = pop();
   push(ch);
  }
 }
  while (s[top] != '#') /* Pop from stack till empty */
  pofx[k++] = pop();
 pofx[k] = '\0'; /* Make pofx as valid string */
 printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n", infx, pofx);

 while( (ch=pofx[i++]) != '\0')
 {
  if(isdigit(ch)) pushit(ch-'0'); /* Push the operand */
  else
  {       /* Operator,pop two  operands */
   op2=popit();
   op1=popit();
   switch(ch)
   {
   case '+':pushit(op1+op2);break;
   case '-':pushit(op1-op2);break;
   case '*':pushit(op1*op2);break;
   case '/':pushit(op1/op2);break;
   }
  }
 }
 printf("\n Given Postfix Expn: %s\n",pofx);
 printf("\n Result after Evaluation: %d\n",s[top]);
}
```

5. Converting an Infix expression to Prefix Expression

i. To convert an infix expression to its prefix expression, the operators have to move to the beginning of the expression.

ii. Examples: A * B + C / D ➔ ((A * B) + (C / D))



iii. The prefix expression is ➔ +*AB/CD

WAP to convert the Infix expression to Prefix expression

```
#define SIZE 50          /* Size of Stack */
#include<string.h>
#include <ctype.h>
char s[SIZE];
int top=-1;       /* Global declarations */

push(char elem)
{                    /* Function for PUSH operation */
   s[++top]=elem;
}

char pop()
{                    /* Function for POP operation */
   return(s[top--]);
}

int pr(char elem)
{              /* Function for precedence */
   switch(elem)
   {
   case '#': return 0;
   case ')': return 1;
   case '+':
   case '-': return 2;
   case '*':
   case '/': return 3;
   }
}

main()
{                    /* Main Program */
   char infx[50],prfx[50],ch,elem;
   int i=0,k=0;
```

```
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
    strrev(infx);
    while( (ch=infx[i++]) != '\0')
    {
       if( ch == ')') push(ch);
       else
          if(isalnum(ch)) prfx[k++]=ch;
          else
             if( ch == '(')
             {
                while( s[top] != ')')
                   prfx[k++]=pop();
                elem=pop(); /* Remove ) */
             }
             else
             {      /* Operator */
                while( pr(s[top]) >= pr(ch) )
                   prfx[k++]=pop();
                push(ch);
             }
    }
    while( s[top] != '#')     /* Pop from stack till empty */
       prfx[k++]=pop();
    prfx[k]='\0';          /* Make prfx as valid string */
    strrev(prfx);
    strrev(infx);
    printf("\n\nGiven Infix Expn: %s  Prefix Expn: %s\n",infx,prfx);
}
```

Example of conversions

Infix	**Postfix**	**Prefix**	**Notes**
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

6. On Stack conversion of infix to postfix expression

i. The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.
ii. This is shown in a table with three columns.
iii. The first will show the symbol currently being read.
iv. The second will show what is on the stack and the third will show the current contents of the postfix string.
v. The stack will be written from left to right with the 'bottom' of the stack to the left.

Example 1: A * B + C becomes A B * C +

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

Example 2: A + B * C becomes A B C * +

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

Example 3: A * (B + C) becomes A B C + *

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(	* (	A B
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	)	*	A B C +
8			A B C + *

i. Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker.
ii. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

Example 4. A - B + C becomes A B - C +

i. When operators have the same precedence, we must consider association.
ii. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

iii. In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack.
iv. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. A * B ^ C + D becomes A B C ^ * D +

i. The exponentiation and the multiplication must be done before the addition.

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

ii. When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack.
iii. Since it has lower precedence, the '^' is popped and printed.
iv. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'.
v. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. A * (B + C * D) + E becomes A B C D * + * E +

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

A summary of the rules follows:

i. Print operands as they arrive.
ii. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
iii. If the incoming symbol is a left parenthesis, push it on the stack.
iv. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
v. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
vi. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
vii. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
viii. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

7. On stack conversion of Infix to Prefix

i. First step is to reverse the expression before converting into its infix form.
ii. The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.
iii. This is shown in a table with three columns.
iv. The first will show the symbol currently being read.
v. The second will show what is on the stack and the third will show the current contents of the postfix string.
vi. The stack will be written from left to right with the 'bottom' of the stack to the left.

Example: A+B*C

Step 1: Reverse the string as C*B+A

Step 2:

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	C		C
2	*	*	C
3	B	*	CB
4	+	+	C B *
5	A	+	C B * A
12	#		C B * A +

Step 3: now reverse the output string as + A * B C

WAP to convert infix to prefix expression

```
# include <stdio.h>
# include <string.h>
# define MAX 20
void infixtoprefix(char infix[20],char prefix[20]);
void reverse(char array[30]);
char pop();
void push(char symbol);
int isOperator(char symbol);
int prcd(symbol);
int top=-1;
char stack[MAX];
main() {
        char infix[20],prefix[20],temp;
        printf("Enter infix operation: ");
        gets(infix);
        infixtoprefix(infix,prefix);
        reverse(prefix);
        puts((prefix));
}
//-------------------------------------------------------
void infixtoprefix(char infix[20],char prefix[20]) {
        int i,j=0;
        char symbol;
        stack[++top]='#';
```

```
        reverse(infix);
        for (i=0;i<strlen(infix);i++) {
                symbol=infix[i];
                if (isOperator(symbol)==0) {
                        prefix[j]=symbol;
                        j++;
                } else {
                        if (symbol==')') {
                                push(symbol);
                        } else if(symbol == '(') {
                                while (stack[top]!=')') {
                                        prefix[j]=pop();
                                        j++;
                                }
                                pop();
                        } else {
                                if (prcd(stack[top])<=prcd(symbol)) {
                                        push(symbol);
                                } else {
                                        while(prcd(stack[top])>=prcd(symbol)) {
                                                prefix[j]=pop();
                                                j++;
                                        }
                                        push(symbol);
                                }
                                //end for else
                        }
                }
                //end for else
        }
        //end for for
        while (stack[top]!='#') {
                prefix[j]=pop();
                j++;
        }
        prefix[j]='\0';
}
////-------------------------------------------------------
void reverse(char array[30]) // for reverse of the given expression {
```

```
        int i,j;
        char temp[100];
        for (i=strlen(array)-1,j=0;i+1!=0;--i,++j) {
                temp[j]=array[i];
        }
        temp[j]='\0';
        strcpy(array,temp);
        return array;
}
//--------------------------------
char pop() {
        char a;
        a=stack[top];
        top--;
        return a;
}
//---------------------------------
void push(char symbol) {
        top++;
        stack[top]=symbol;
}
//-----------------------------------------
int prcd(symbol) // returns the value that helps in the precedence {
        switch(symbol) {
                case '+':
                        case '-':
                        return 2;
                break;
                case '*':
                        case '/':
                        return 4;
                break;
                case '$':
                        case '^':
                        return 6;
                break;
                case '#':
                        case '(':
                        case ')':
```

```
                return 1;
            break;
        }
}
//-----------------------------------------------
int isOperator(char symbol) {
        switch(symbol) {
            case '+':
                case '-':
                case '*':
                case '/':
                case '^':
                case '$':
                case '&':
                case '(':
                case ')':
                return 1;
            break;
            default:
                return 0;
            // returns 0 if the symbol is other than given above
        }
}
```

Queues

CONTENTS

Hours: 8

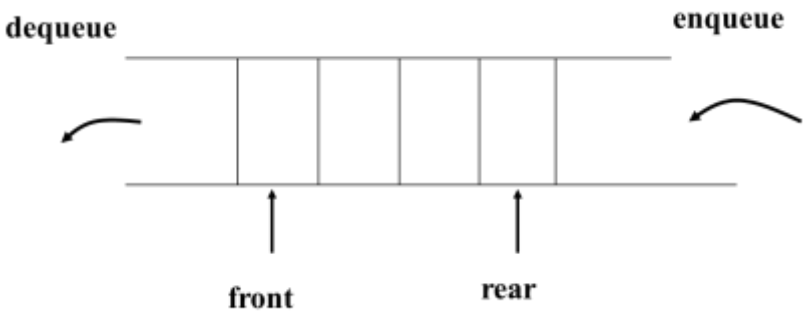
Marks: 12

4.1 Introduction to Queues

1. Queue as Abstract Data Type

(Question: Explain queue as an abstract data type- 4 marks)

i. A queue is a list from which items are deleted from one end (front) and into which items are inserted at the other end (rear, or back)
ii. It is like line of people waiting to purchase tickets:
iii. Queue is referred to as a first-in-first-out (FIFO) data structure.
iv. The first item inserted into a queue is the first item to leave



Operations on Queue

i. ***createQueue()*** :Create an empty queue
ii. ***destroyQueue()***: Destroy a queue
iii. ***isEmpty():Boolean****:* Determine whether a queue is empty
iv. ***enqueue(in newItem:QueueItemType) throw QueueException***: Inserts a new item at the end of the queue (at the **rear** of the queue)
v. ***dequeue() throw QueueException***
dequeue(out queueFront:QueueItemType) throw QueueException
Removes (and returns) the element at the **front** of the queue
Remove the item that was added earliest
vi. ***getFront(out queueFront:QueueItemType) throw QueueException***: Retrieve the item that was added earliest (without removing).

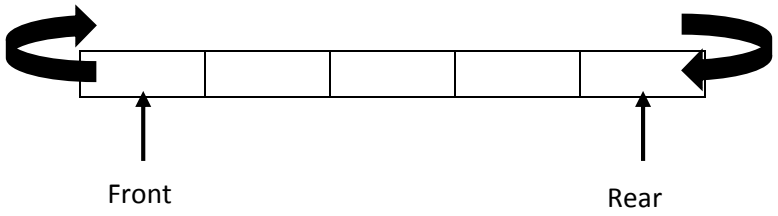
2. Representation of a Queue as an array

(Question: Explain the concept of queue using array- 4 marks)

i. With queues two integer indexes are required, one giving the position of the front element of the queue in the array, the other giving the position

of the back element, or rather in a similar way to stacks, the position after the back element.

ii. We add the elements from rear, and remove the elements from the front as shown in the Figure.



iii. In queue as we delete one data element, the front is incremented by one.

iv. We delete the elements from the end, known as the rear of the queue.

v. The **disadvantage** of using the queue is as we delete the elements, the front moves ahead, even though the queue is empty we still cannot add the elements in the queue.

(Question: WAP to perform add and delete elements from the queue - 6 marks)

WAP to perform add and delete elements from the queue.

```
#include<stdio.h>
#include<conio.h>
void addq(int);
void delq();
int qu[5],front = -1, rear = -1, item;
void addq(int item)
{
 if ((front == rear) && (front == -1))
 {
   printf("Queue is empty");
   front++;
   rear++;
   qu[rear]= item;
 }
 else
 {
   rear++;

qu[rear] = item;
```

```
 }
}
void delq()
{
 printf(" The removed item is ➔ %d", qu[front]);
 front= front + 1;
}
void main()
{
 int ch = 1,ans =0,ele,temp;
 clrscr();
 do
 {
  printf("\n 1: Add the element in the queue");
  printf("\n 2: Delete the element from the queue");
  printf("\n 3: Display the queue");
  printf("\n 4: Exit");
  printf("\n Enter your choice ➔");
  scanf("%d",&ch);
  switch(ch)
  {
    case 1: {   if (rear<=5)
                {
                    printf("Enter the element to be added ➔");
                    scanf("%d",&ele);
                    addq(ele);
                    break;
                }
                else
                  {
                    printf("Queue Full");
                    break;
                  }
            }



case 2: {
```

```
            delq();
            break;
          }
case 3: { temp = front;
            while (front <= rear)
            {
             printf("\n The element of the queue are ➔ %d", qu[front]);
             front++;
            }
            front = temp;
            break;
           }
case 4: {
                exit();
                break;
          }
}
printf(" \n Do you want to continue(1 to continue , 0 to exit) ➔");
scanf("%d", &ans);
}while(ans != 0);
}
```

OUTPUT

```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔1
Enter the element to be added ➔10
Queue is empty
Do you want to continue (1 to continue, 0 to exit) ➔1
```

```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔1
Enter the element to be added ➔20
Do you want to continue (1 to continue, 0 to exit) ➔1
```

```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔1
Enter the element to be added ➔30
Do you want to continue (1 to continue, 0 to exit) ➔1
```

```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔1
Enter the element to be added ➔40
Do you want to continue (1 to continue, 0 to exit) ➔1
```

```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔3
The element of the queue are ➔10
The element of the queue are ➔ 20
The element of the queue are ➔ 30
The element of the queue are ➔ 40
Do you want to continue (1 to continue, 0 to exit) ➔1
```

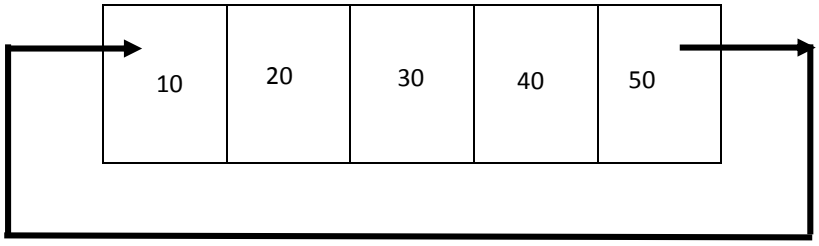
```
1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice ➔2
The removed item is ➔10
Do you want to continue (1 to continue, 0 to exit) ➔0
```

4.2 Types of Queue

1. Circular Queue

(Question: Explain the concept of circular queue with diagram using arrays - 6 marks)

i. Circular queue is a linear data structure. It follows FIFO principle.
ii. In circular queue the last node is connected back to the first node to make a circle.
iii. Circular linked list fallow the First In First Out principle
iv. Elements are added at the rear end and the elements are deleted at front end of the queue
v. Both the front and the rear pointers points to the beginning of the array.
vi. It is also called as "Ring buffer".
vii. Items can inserted and deleted from a queue in O(1) time.



viii. Circular Queue can be created in three ways they are

a. Using arrays

- In arrays the range of a subscript is 0 to n-1 where n is the maximum size.
- To make the array as a circular array by making the subscript 0 as the next address of the subscript n-1 by

using the formula subscript = (subscript +1) % maximum size.

- In circular queue the front and rear pointer are updated by using the above formula.
- The function below explains to add an element in the circular queue using arrays.

```
Step 1. Start
Step 2. if (front == (rear+1)%max)
        Print error “circular queue overflow “
Step 3. Else
        { rear = (rear+1)%max
        Q[rear] = element;
        If (front == -1 ) f = 0;
        }
Step 4. Stop
```

- The function below explains to delete an element from circular queue using arrays.

```
Step 1. Start
Step 2. if ((front == rear) && (rear == -1))
        Print error “circular queue underflow “
step 3. else
        { element = Q[front]
        If (front == rear) front=rear = -1
        Else
        Front = (front + 1) % max
 Step 4. Stop
```

WAP to insert, delete and display the elements of the circular queue.

```
#include<stdio.h>
#define SIZE 5
void insert();
void delet();
void display();
int queue[SIZE], rear=-1, front=-1, item;
main()
{
        int ch;
 do
 {
        printf("\n\n1. \tInsert\n2. \tDelete\n3. \tDisplay\n4. \tExit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
        switch(ch)
        {
        case 1:
                insert();
                break;
        case 2:
                delet();
                break;
        case 3:
                display();
                break;
        case 4:
                exit(0);
        default:
                printf("\n\nInvalid choice. Please try again... \n");
         }
        } while(1);
        getch();
}
void insert()
{
        if((front==0 && rear==SIZE-1) || (front==rear+1))
                printf("\n\nQueue is full.");
```

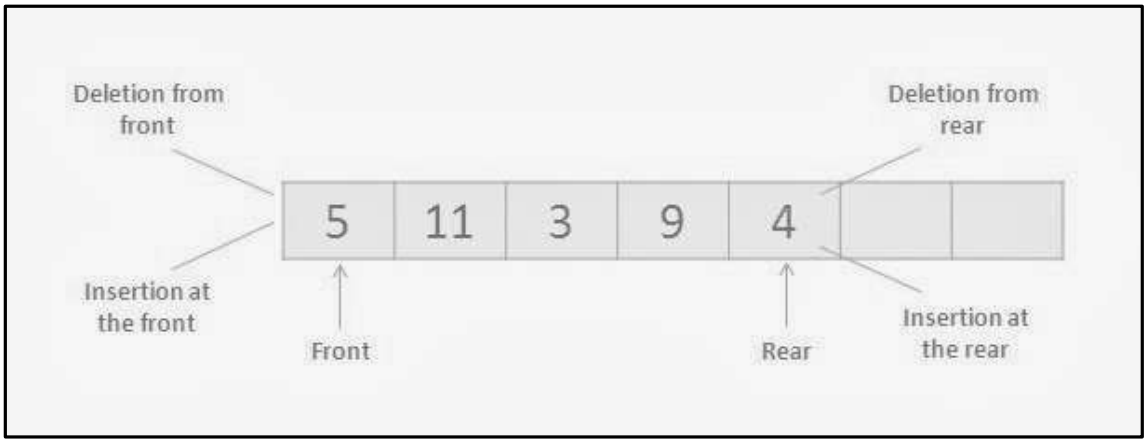
```
else
        {
                printf("\n\nEnter ITEM: ");
                scanf("%d", &item);
                if(rear == -1)
                {
                        rear = 0;
                        front = 0;
                }
                else if(rear == SIZE-1)
                        rear = 0;
                else
                        rear++;
                        queue[rear] = item;
                        printf("\n\nItem inserted: %d\n", item);
                }
        }
void delet()
{
        if(front == -1)
                printf("\n\nQueue is empty. \n");
        else
        {
item = queue[front];
                if(front == rear)
                {
                        front = -1;
                        rear = -1;
                }

                else if(front == SIZE-1)
                        front = 0;
                else
                        front++;
                printf("\n\nITEM deleted: %d", item);
                }
}
```

```
void display()
{
 int i;
        if((front == -1) || (front==rear+1))
                printf("\n\nQueue is empty. \n");
        else
        { printf("\n\n");
                for(i=front; i<=rear; i++)
                        printf("\t%d",queue[i]);
        }
}
```

2. Double Ended Queue

(Question: Explain the concept of double ended queue with diagram using arrays - 6 marks)

i. Dequeue is also called as double ended queue and it allows user to perform insertion and deletion from the front and rear position.
ii. And it can be easily implemented using doubly linked list.
iii. On creating dequeue, we need to add two special nodes at the ends of the doubly linked list (head and tail).
iv. And these two nodes needs to be linked between each other (initially).



Algorithm – Insertion at rear end

(Question: Write the algorithm to insert the element at the rear - 4 marks)

```
Step -1 : [Check for overflow]
            if(rear==MAX)
                  Output "Queue is Overflow"
                  return;
Step-2  : [Insert element]
            else
                   rear=rear+1;
                   q[rear]=no;
                  [Set rear and front pointer]
if rear=0
                        rear=1;
                  if front=0
                         front=1;
Step-3 : return
```

Algorithm: Insertion at front end

(Question: Write the algorithm to insert the element at the front - 4 marks)

```
Step-1 : [Check for the front position]
            if(front<=1)
                   Ourput “Cannot add value at front end”
                  return;
Step-2  : [Insert at front]
            else
                  front=front-1;
                  q[front]=no;
Step-3  : Return
```

Algorithm: Deletion from front end

(Question: Write the algorithm to delete the element at the front - 4 marks)

```
Step-1 [ Check for front pointer]
        if front=0
                Output " Queue is Underflow"
                return;
Step-2 [Perform deletion]
        else
                no=q[front];
                Output "Deleted element is",no
 [Set front and rear pointer]
if front=rear
                                front=0;
                                rear=0;
                        else
                                front=front+1;
Step-3 : Return
```

Algorithm: deletion from rear end

(Question: Write the algorithm to delete the element at the rear - 4 marks)

```
Step-1 : [Check for the rear pointer]
                if rear=0
                        Output "Cannot delete value at rear end"
                        return;
Step-2: [ perform deletion]
                else
                        no=q[rear];
                        [Check for the front and rear pointer]
                        if front= rear
                                front=0;
                                rear=0;
                        else
                                rear=rear-1;
                                Output "Deleted element is",no
Step-3 : Return
```

(Question: WAP to add, delete, and display the elements in a double ended queue. - 6 marks)

WAP to add, delete, and display the elements in a double ended queue.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 10

int q[MAX],front=0,rear=0;
void add_rear();
void add_front();
void delete_rear();
void delete_front();
void display();
void main()
{
        int ch;
        clrscr();
        do
        {
        clrscr();
        printf("\n DQueue Menu");
        printf("\n--------------");
        printf("\n 1. Add Rear");
        printf("\n 2. Add Front");
        printf("\n 3. Delete Rear");
        printf("\n 4. Delete Front");
        printf("\n 5. Display");
        printf("\n 6. Exit");
                printf("\n Enter your choice:-");
                scanf("%d",&ch);
```

```
switch(ch)
        {
            case 1:
            {
                add_rear();
                printf("\n Queue after insert at rear");
               display();
               break;
            }
            case 2:
            {
               add_front();
               printf("\n Queue after insert at front");
               display();
               break;
             }
            case 3:
            {
                delete_rear();
               printf("\n Queue after delete at rear");
               display();
               break;
            }
            case 4:
            {
            delete_front();
            printf("\n Queue after delete at front");
            display();

            break;
            }
            case 5:
                {
                    display();
                    break;
                }
```

```
case 6:
                        {
                                exit(0);
                                break;
                        }
                default:
                        {
                                printf("\n Wrong Choice\n");
                        }
                }
        } while(ch!=6);
}
void add_rear()
{
        int no;
        printf("\n Enter value to insert : ");
        scanf("%d",&no);
        if(rear==MAX)
        {
                printf("\n Queue is Overflow");
                return;
        }
        else
        {
                rear++;
                q[rear]=no;
                if(rear==0)
                        rear=1;
                 if(front==0)
                        front=1;
        }
}
```

```
void add_front()
{
        int no;
        printf("\n Enter value to insert:-");
        scanf("%d",&no);
        if(front<=1)
        {
                printf("\n Cannot add value at front end");
                return;
        }
        else
          {
             front--;
             q[front]=no;
          }
        }
void delete_front()
{
        int no;
        if(front==0)
        {
                printf("\n Queue is Underflow\n");
                return;
        }
        else
        {
                no=q[front];
                printf("\n Deleted element is %d\n",no);
                if(front==rear)
                {
                        front=0;
                        rear=0;
                }
                else
                {
                        front++;
                }
        }
}
```

```
void delete_rear()
{
        int no;
        if(rear==0)
        {
                printf("\n Cannot delete value at rear end\n");
                return;
        }
        else
        {
                no=q[rear];
                if(front==rear)
                {
                        front=0;
                        rear=0;
                }
                else
                {
                        rear--;
                        printf("\n Deleted element is %d\n",no);
                }
        }
}
void display()
{        int i;
        if(front==0)
        {
                printf("\n Queue is Underflow\n");
                return;
        }
        else
        {
                printf("\n Output");
                for(i=front;i<=rear;i++)
                {           printf("\n %d",q[i]);
                }
        }
}
```

3. Priority Queue

(Question: Explain the concept of priority queue - 4 marks)

i. A priority queue is a collection in which items can be added at any time, but the only item that can be removed is the one with the highest priority.

ii. Examples of the priority queue are operating system scheduler, patients waiting in the operation threater.

iii. Operations
- add(x): add item x.
- remove: remove the highest priority item.
- peek: return the highest priority (without removing it).
- size: return the number of items in the priority queue.
- isEmpty: return whether the priority queue has no items.

(Question: WAP to insert and delete elements from priority queue using arrays- 4 marks)

WAP to insert and delete elements from priority queue using arrays

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

#define MAX 5

void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();

int pri_que[MAX];
int front, rear;

void main()
{
   int n, ch;
```

```
    printf("\n1 - Insert an element into queue");
    printf("\n2 - Delete an element from queue");
    printf("\n3 - Display queue elements");
    printf("\n4 - Exit");

    create();
     while (1)
    {
        printf("\nEnter your choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            printf("\nEnter value to be inserted : ");
            scanf("%d",&n);
            insert_by_priority(n);
            break;
        case 2:
            printf("\nEnter value to delete : ");
            scanf("%d",&n);
            delete_by_priority(n);
            break;
        case 3:
            display_pqueue();
            break;
        case 4:
            exit(0);
        default:
            printf("\nChoice is incorrect, Enter a correct choice");
        }
    }
}
```

```
/* Function to create an empty priority queue */
void create()
{
   front = rear = -1;
}

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
   if (rear >= MAX - 1)
   {
      printf("\nQueue overflow no more elements can be inserted");
      return;
   }
 if ((front == -1) && (rear == -1))
   {
      front++;
      rear++;
      pri_que[rear] = data;
      return;
   }
   else
      check(data);
   rear++;
}

/* Function to check priority and place element */
void check(int data)
{
   int i,j;

   for (i = 0; i <= rear; i++)
   {
      if (data >= pri_que[i])
      {
         for (j = rear + 1; j > i; j--)
         {
            pri_que[j] = pri_que[j - 1];
         }
```

```
pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}

/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;

    if ((front==-1) && (rear==-1))
    {
        printf("\nQueue is empty no elements to delete");
        return;
    }

    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }

        pri_que[i] = -99;
        rear--;

        if (rear == -1)
            front = -1;
        return;
        }
    }
    printf("\n%d not found in queue to delete", data);
}
```

```
/* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }

    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }

    front = 0;
}
```

```
1 - Insert an element into queue
2 - Delete an element from queue
3 - Display queue elements
4 - Exit
Enter your choice : 1

Enter value to be inserted : 20
```

```
Enter your choice : 1

Enter value to be inserted : 45
```

```
Enter your choice : 1

Enter value to be inserted : 89
```

```
Enter your choice : 3
 89  45  20
```

```
Enter your choice : 1

Enter value to be inserted : 56
```

```
Enter your choice : 3
 89  56  45  20
```

```
Enter your choice : 2

Enter value to delete : 45
```

```
Enter your choice : 3
 89  56  20
Enter your choice : 4
```

4.3 Applications

i. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

ii. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Link List

CONTENTS

Hours: 8

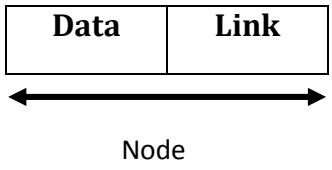
Marks: 12

5.1 Introduction to Link List

1. Terminologies

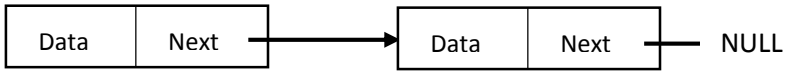
(Question: Define node, address, pointer, information, next and Null pointer – 6 marks)

a. **Node:** The structure combination of data and its link to the next structure is called as node.



Node

b. **Address:** Stores the address of the next node of the link list.
c. **Pointer:** A data type used to point to the address of the next node.
d. **Information:** The data stored in the data field of the link list structure.
e. **Next:** The link which points to the next node of the link list
f. **Null Pointer:** Signifies the end of link list, when the next or the link pointer points to NULL
g. **Empty list:** Signifies no data in the link list.



5.2 Type of lists

1. Linear list

(Question: Explain the concept of linear link list- 6 Marks)

i. Linked lists stores data with structures, create a new place to store data.
ii. It is written using a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type.
iii. Each of these individual struct in the list is commonly known as a node or element of the list.

No	Element	Explanation
1	Node	Linked list is collection of number of nodes
2	Address Field in Node	Address field in node is used to keep address of next node
3	Data Field in Node	Data field in node is used to hold data inside linked

Code to create a node in the beginning

(Question: Write the code to create node in the beginning- 4 Marks)

```
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};

int main()
{
    struct node *root;
    root = (struct node *) malloc( sizeof(struct node) );
    root->next = NULL;
    root->data = 18;
}
```

To create a node and add it at the end.

(Question: Write the code to create node and add it at the end- 4 Marks)

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
int main()
{
    struct node *root;
    struct node *temp;
    root = malloc( sizeof(struct node) );
    root->next = NULL;
    root->x = 18;
    temp = root;
    if ( temp != NULL )
    {
while ( temp->next != NULL)
        {
            temp = temp->next;
        }
    }
    temp->next = malloc( sizeof(struct node) );
    temp = temp->next;
    if ( temp == 0 )
    {
        printf( "Out of memory" );
        return 0;
    }
    /* initialize the new memory */
    temp->next = NULL;
    temp->data = 22;
    return 0;
}
```

To print the node

(Question: Write the code to print the node- 4 Marks)

```
temp = root;
if ( temp != 0 ) { /* Makes sure there is a place to start */
   while ( temp->next != NULL )
   {
      printf( "%d\n", temp->data);
      temp = temp->next;
   }
   printf( "%d\n", temp->x );
}
```

Advantages

1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment. Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
2. Insertion and deletion node operations are easily implemented in a linked list.
3. Linear data structures such as stacks and queues are easily executed with a linked list.
4. They can reduce access time and may expand in real time without memory overhead.

Disadvantages

1. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.
2. They have a tendency to use more memory due to pointers requiring extra storage space.
3. Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
4. Nodes are stored in contiguously, greatly increasing the time required to access individual elements within the list.
5. Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards and

while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

WAP to add, delete, display and search the element in the link list

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
	int data;
	struct Node *next;
}node;
void insert(node *pointer, int data)
{
	while(pointer->next!=NULL)
	{
		pointer = pointer -> next;
	}
	pointer->next = (node *)malloc(sizeof(node));
	pointer = pointer->next;
	pointer->data = data;
	pointer->next = NULL;
}
int search(node *pointer, int key)
{
	pointer =  pointer -> next; //First node is dummy node.
	while(pointer!=NULL)
	{
		if(pointer->data == key) //key is found.
		{
			return 1;
		}
		pointer = pointer -> next;//Search in the next node.
	}
	return 0;
}
```

```
void delete(node *pointer, int data)
{
	while(pointer->next!=NULL && (pointer->next)->data != data)
	{
		pointer = pointer -> next;
	}
	if(pointer->next==NULL)
	{
		printf("Element %d is not present in the list\n",data);
		return;
	}
	node *temp;
	temp = pointer -> next;
	pointer->next = temp->next;
	free(temp);
	return;
}
void print(node *pointer)
{
	if(pointer==NULL)
	{
		return;
	}
	printf("%d ",pointer->data);
	print(pointer->next);
}
int main()
{
	node *start,*temp;
	start = (node *)malloc(sizeof(node));
	temp = start;
	temp -> next = NULL;
	printf("1. Insert\n");
	printf("2. Delete\n");
	printf("3. Print\n");
	printf("4. Search\n");
```

```
while(1)
	{
		int query;
		scanf("%d",&query);
if(query==1)
		{
			int data;
			scanf("%d",&data);
			insert(start,data);
		}
		else if(query==2)
		{
			int data;
			scanf("%d",&data);
			delete(start,data);
		}
		else if(query==3)
		{
			printf("The list is ");
			print(start->next);
			printf("\n");
		}
		else if(query==4)
		{
			int data;
			scanf("%d",&data);
			int status = seach(start,data);
			if(status)
			{
				printf("Element Found\n");
			}
			else
			{
				printf("Element Not Found\n");
			}
		}
	}
}
```

2. Circular list

(Question: explain the concept of circular list- 4 Marks)

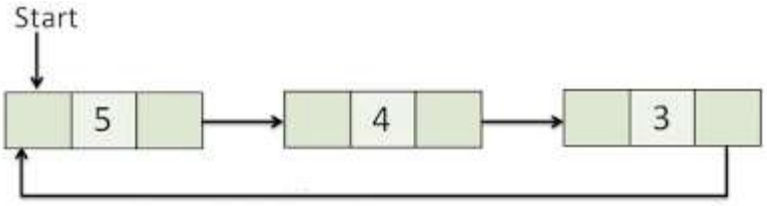
a. Circular Linked List is divided into 2 Categories.
   - Singly Circular Linked List
   - Doubly Circular Linked List

b. In Circular Linked List Address field of Last node contain address of "First Node".

c. In short First Node and Last Nodes are adjacent.

d. Linked List is made circular by linking first and last node, so it looks like circular chain [shown in following diagram].

e. Two way access is possible only if we are using "Doubly Circular Linked List" Sequential movement is possible, No direct access is allowed.



WAP to add, delete and display circular link list

(Question: Write the code to add, delete, and display circular link list - 6 Marks)

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
	int data;
	struct Node *next;
}node;
void insert(node *pointer, int data)
{
	node *start = pointer;
	while(pointer->next!=start)
	{
		pointer = pointer -> next;
	}
```

```
        pointer->next = (node *)malloc(sizeof(node));
        pointer = pointer->next;
        pointer->data = data;
        pointer->next = start;
}
int find(node *pointer, int key)
{
        node *start = pointer;
        pointer =  pointer -> next; //First node is dummy node.
        while(pointer!=start)
        {
                if(pointer->data == key) //key is found.
                {
                        return 1;
                }
                pointer = pointer -> next;//Search in the next node.
        }
        /*Key is not found */
        return 0;
}
void delete(node *pointer, int data)
{
        node *start = pointer;
        while(pointer->next!=start && (pointer->next)->data != data)
        {
                pointer = pointer -> next;
        }
        if(pointer->next==start)
        {
                printf("Element %d is not present in the list\n",data);
                return;
        }
        node *temp;
        temp = pointer -> next;
        pointer->next = temp->next;
        free(temp);
        return;
}
```

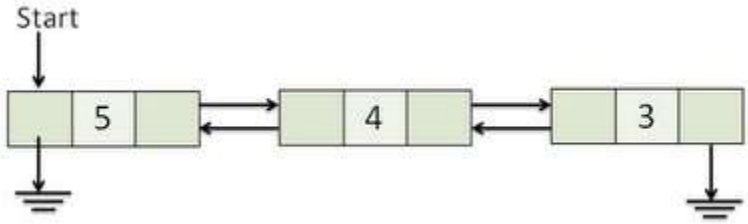
```
void print(node *start,node *pointer)
{
	if(pointer==start)
	{
		return;
	}
	printf("%d ",pointer->data);
	print(start,pointer->next);
}
int main()
{
	node *start,*temp;
	start = (node *)malloc(sizeof(node));
	temp = start;
	temp -> next = start;
	printf("1. Insert\n");
	printf("2. Delete\n");
	printf("3. Print\n");
	printf("4. Find\n");
	while(1)
	{
		int query;
		scanf("%d",&query);
		if(query==1)
		{
			int data;
			scanf("%d",&data);
			insert(start,data);
		}
		else if(query==2)
		{
			int data;
			scanf("%d",&data);
			delete(start,data);
		}
```

```
else if(query==3)
        {
            printf("The list is ");
            print(start,start->next);
            printf("\n");
        }
        else if(query==4)
        {
            int data;
            scanf("%d",&data);
            int status = find(start,data);
            if(status)
            {
                printf("Element Found\n");
            }
            else
            {
                printf("Element Not Found\n");
            }
        }
    }
}
```

3. Doubly list

(Question: Explain the concept of doubly link list- 4 Marks)

a. In Doubly Linked List, each node contain two address fields.
b. One address field for storing address of next node to be followed and second address field contain address of previous node linked to it.
c. So Two way access is possible i.e we can start accessing nodes from start as well as from last.
d. Like Singly Linked List also only Linear but Bidirectional Sequential movement is possible.
e. Elements are accessed sequentially, no direct access is allowed.



5.3 Operations on a singly linked list (only algorithm)

1. Create a node

(Question: Write the code to create a node- 4 Marks)

```
void creat()
{
char ch;
 do
 {
 struct node *new_node,*current;
 new_node=(struct node *)malloc(sizeof(struct node));

 printf("nEnter the data : ");
 scanf("%d",&new_node->data);
 new_node->next=NULL;

  if(start==NULL)
  {
  start=new_node;
  current=new_node;
  }
  else
  {
  current->next=new_node;
  current=new_node;
  }
 printf("nDo you want to creat another : ");
 ch=getche();
 }while(ch!='n');
}
```

Traversing a singly linked list

(Question: Write the code to traverse in singly link list- 4 marks)

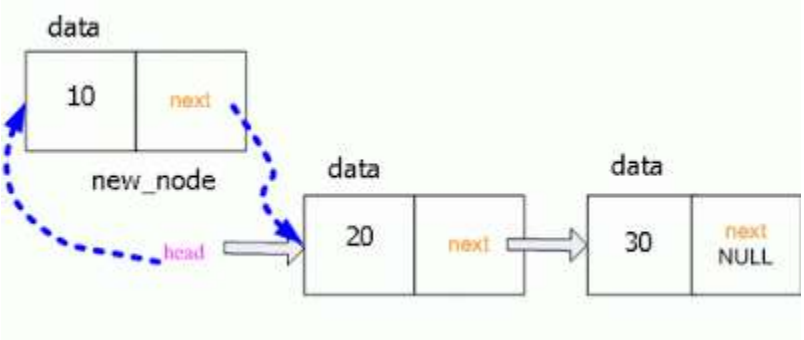
```
struct node *temp;  //Declare temp
temp = start;
while(temp!=NULL)
   {
   printf("%d",temp->data);
   temp=temp->next;
   }
```

Searching a linked list

```
int search(int num)
{
int flag = 0;
struct node *temp;

temp = start;
  while(temp!=NULL)
  {
    if(temp->data == num)
       return(1); //Found

    temp = temp->next;
  }

if(flag == 0)
   return(0); // Not found
}
```

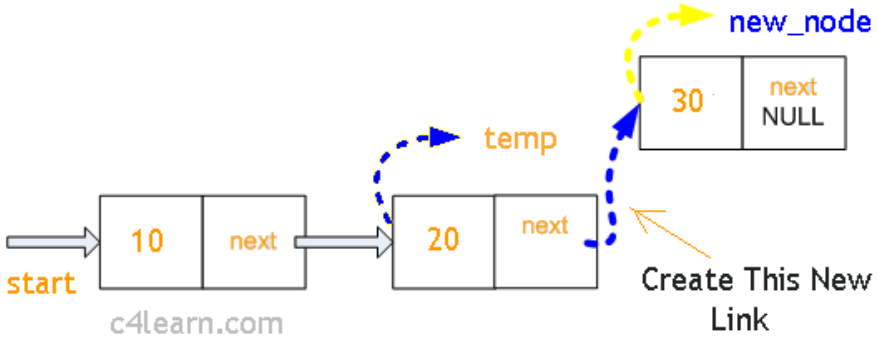
Inserting a new node in a linked list at the Beginning

```
void insert_at_beg()
{
struct node *new_node,*current;

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

 if(new_node == NULL)
   printf("nFailed to Allocate Memory");

 printf("nEnter the data : ");
 scanf("%d",&new_node->data);
 new_node->next=NULL;

  if(start==NULL)
  {
  start=new_node;
  current=new_node;
  }
  else
  {
  new_node->next=start;
  start=new_node;
  }
}
```



Inserting a new node in a linked list at the End

```
void insert_at_end()
{
struct node *new_node,*current;
new_node=(struct node *)malloc(sizeof(struct node));
if(new_node == NULL)
        printf("nFailed to Allocate Memory");
 printf("nEnter the data : ");
scanf("%d",&new_node->data);
new_node->next=NULL;
 if(start==NULL)
 {
   start=new_node;
   current=new_node;
 }
 else
 {
   temp = start;
     while(temp->next!=NULL)
     {
          temp = temp->next;
     }
   temp->next = new_node;
 }
}
```

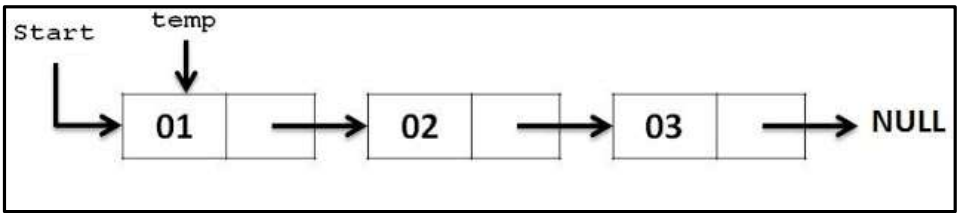


Inserting a new node in a linked list at the Middle

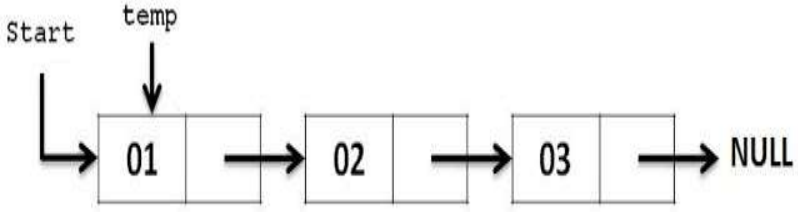
```
void insert_mid()
{
   int pos,i;
   struct node *new_node,*current,*temp,*temp1;
   new_node=(struct node *)malloc(sizeof(struct node));
   printf("nEnter the data : ");
   scanf("%d",&new_node->data);
   new_node->next=NULL;
   st :
   printf("nEnter the position : ");
   scanf("%d",&pos);
   if(pos>=(length()+1))
     {
     printf("nError : pos > length ");
     goto st;
     }

   if(start==NULL)
     {
     start=new_node;
     current=new_node;
     }
   else
     {
     temp = start;
          for(i=1;i< pos-1;i++)
          {
          temp = temp->next;
          }
     temp1=temp->next;
     temp->next = new_node;
     new_node->next=temp1;
     }
}
```

Deleting a node from Beginning of a linked list

```
void del_beg()
{
struct node *temp;

temp = start;
start = start->next;

free(temp);
printf("nThe Element deleted Successfully ");
}
```

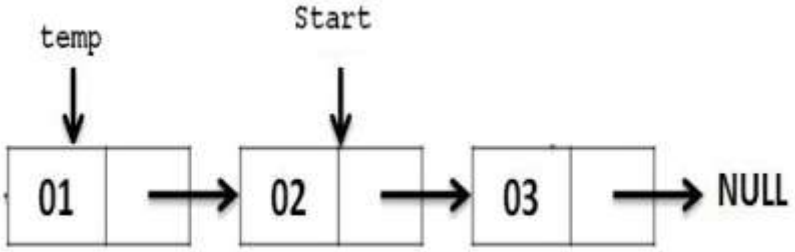


Step 1: Store Current Start in Another Temporary Pointer temp = start



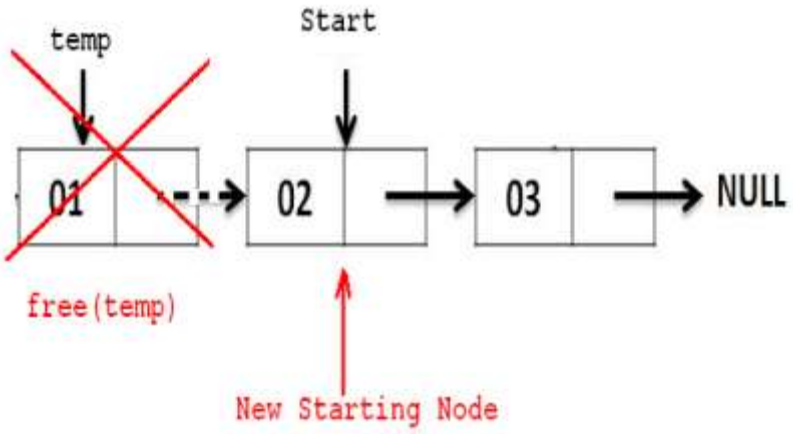
Step 2: Move start pointer one position ahead

start = start -> next



Step 3: Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer.

free(temp)



Trees

CONTENTS

6.1 Introduction

1. Terminologies: tree ,degree of a node, degree of a tree, level of a node, leaf node, Depth / Height of a tree, In-degree & out-Degree, Directed edge, Path, Ancestor & descendant nodes.

6.2 Type of Trees

1. General tree
2. Binary tree
3. Binary search tree (BST).

6.3 Binary tree traversal (only algorithm)

1. In order traversal
2. Preorder traversal
3. Post order traversal
4. Expression tree

Hours: 8

Marks: 12

6.1. Introduction to Trees

1. Terminologies

(Question: Define any six tree terminology – 6 marks)

i. **Parent:** The predecessor of a node**.**

ii. **Child:** Any successor of a node.

iii. **Siblings:** Any pair of nodes that have the same parent**.**

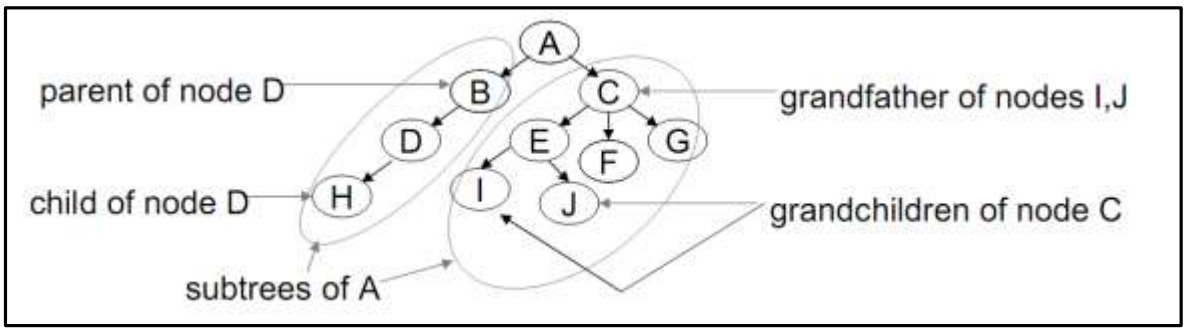
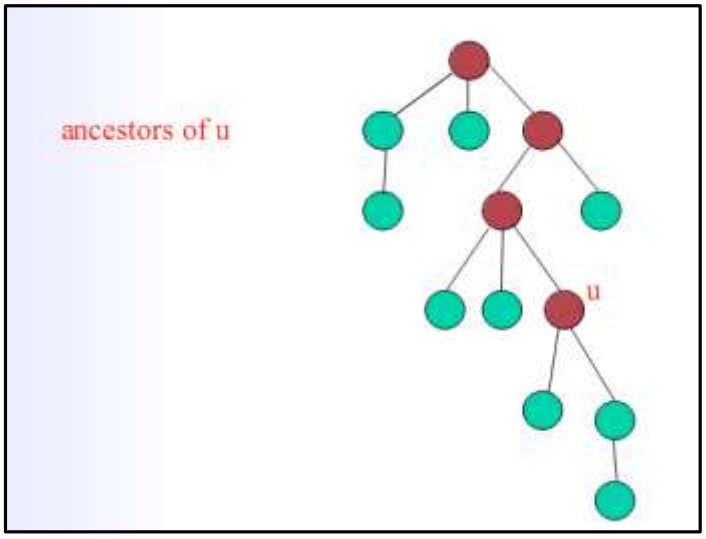
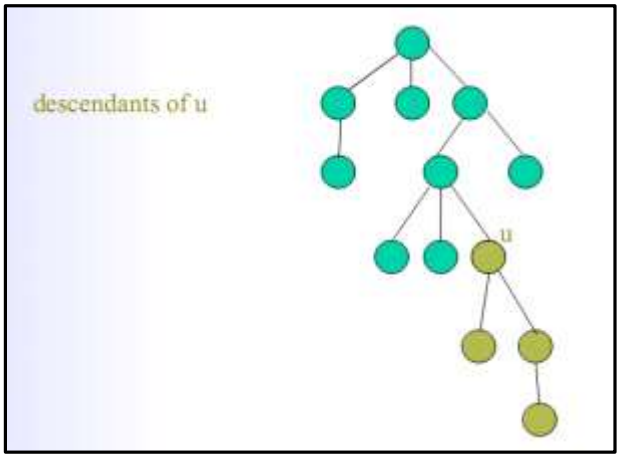


Figure 1: Parent, child, subtree

iv. **Ancestor:** The predecessor of a node together with all the ancestors of the predecessor of a node. The root node has no ancestors.

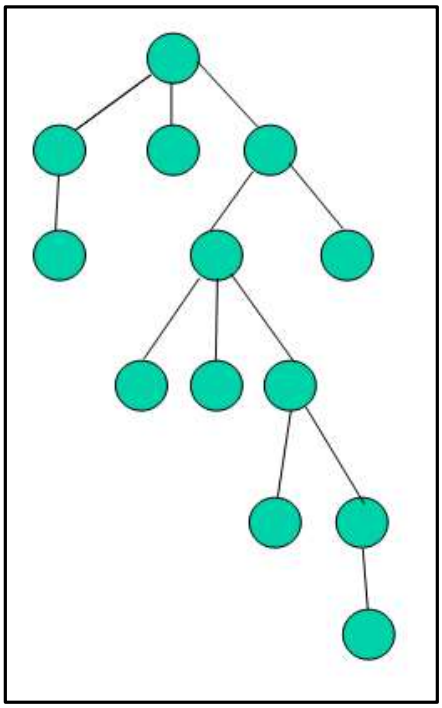


v. **Descendant:** The children of a node together with all the descendants of the children of a node. A leaf node has no descendants.



vi. **Subtree:** A node together with its descendants**.**

vii. **Root:** The top node in a tree.

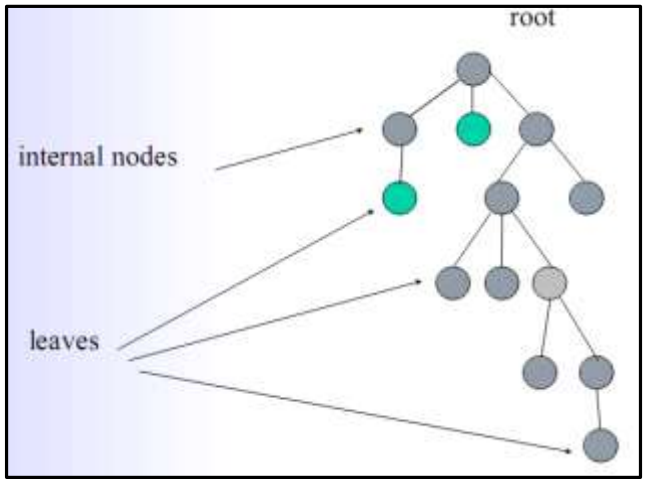


viii. **Parent:** The converse notion of child.

ix. **Leaf:** A node with no children.

x. **Internal node:** A node with at least one child.

xi. **External node:** A node with no children.



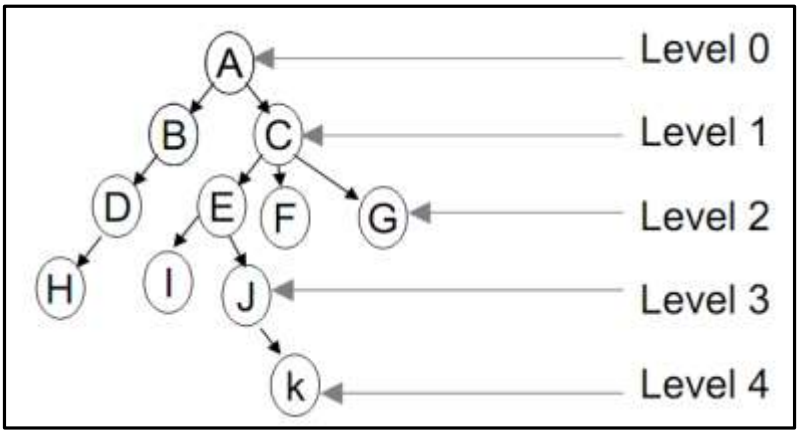
xii. **Degree:** Number of sub trees of a node.

xiii. **Edge:** Connection between one nodes to another.

xiv. **Path:** A sequence of nodes and edges connecting a node with a descendant.

xv. **Level:** The level of a node is defined by 1 + the number of connections between the node and the root.

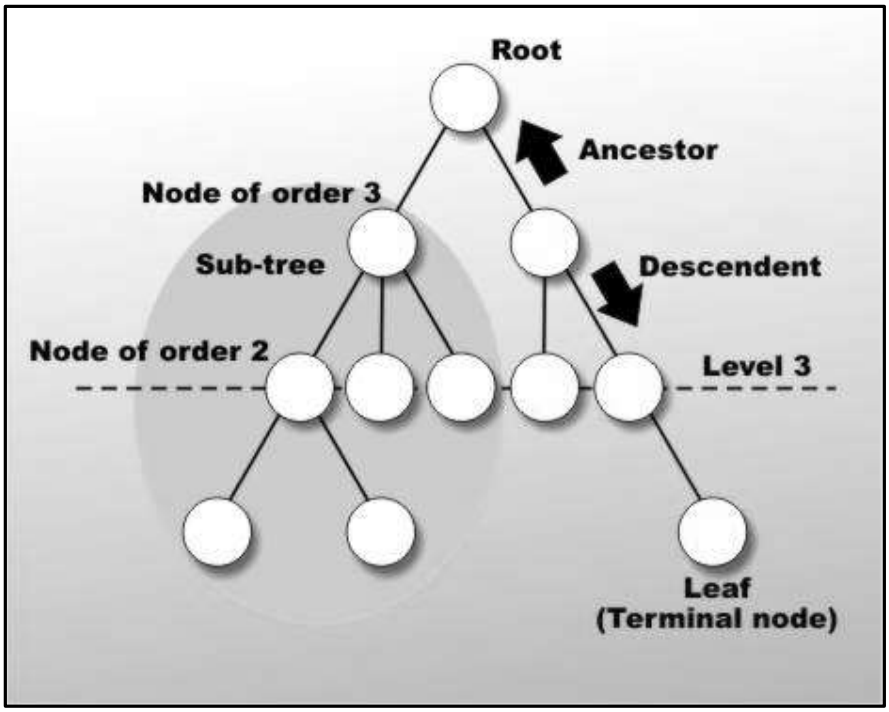
xvi. **Height of tree:** The height of a tree is the number of edges on the longest downward path between the root and a leaf.



xvii. **Height of node:** The height of a node is the number of edges on the longest downward path between that node and a leaf.

xviii. **Depth:** The depth of a node is the number of edges from the node to the tree's root node.

xix. **Forest:** A forest is a set of n ≥ 0 disjoint trees.



6.2 Type of Trees

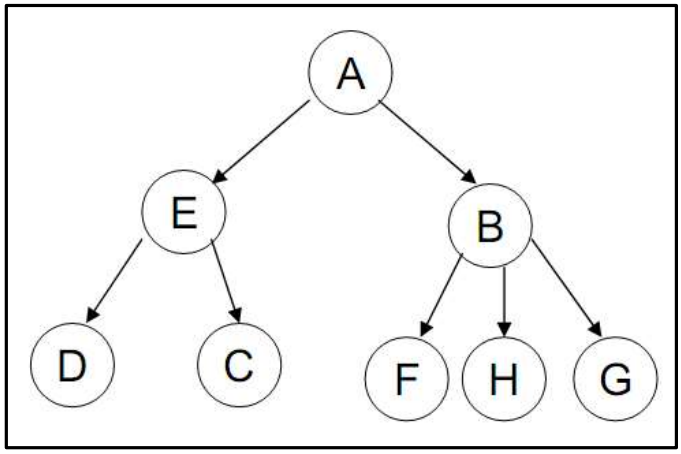
1. General tree

(Question: Explain the concept of general tree- 4 Marks)

i. A tree, is a finite set of nodes together with a finite set of directed edges that define parent-child relationships. Each directed edge connects a parent to its child. Example:

Nodes={A,B,C,D,E,f,G,H}

Edges={(A,B),(A,E),(B,F),(B,G),(B,H),(E,C),(E,D)}



A tree satisfies the following properties

(Question: Explain the properties of general tree- 4 Marks)

i. It has one designated node, called the root that has no parent.
ii. Every node, except the root, has exactly one parent.
iii. A node may have zero or more children.
iv. There is a unique directed path from the root to each node.
v. In a general tree, there is no limit to the number of children that a node can have.
vi. They are suitable for:
   - Hierarchical structure representation, e.g.,
     ⇒ File directory.

⇒ Organizational structure of an institution.

⇒ Class inheritance tree.

- – Problem representation, e.g.,

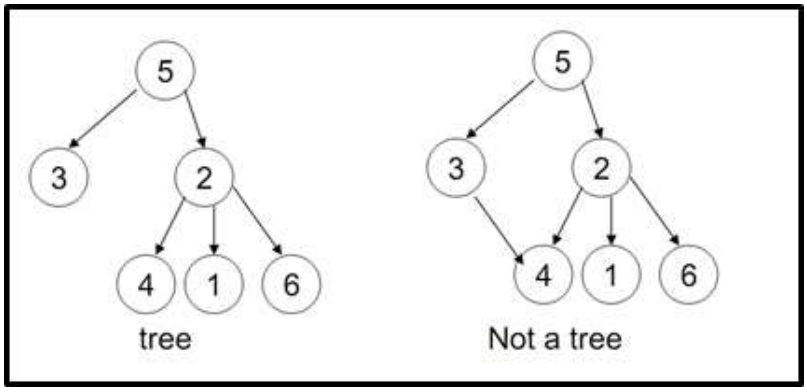
⇒ Expression tree.

⇒ Decision tree.

- Efficient algorithmic solutions, e.g.,

⇒ Search trees.

⇒ Efficient priority queues via heaps.



WAP to create tree

(Question: WAP to create tree- 8Marks)

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
 #define SIZE 9
struct treeNode
{
        int item;
        struct treeNode *left;
        struct treeNode *mid;
        struct treeNode *right;
 }treeNode;
```

```
int (*comparer)(int, int);
 int compare(int a,int b)
{
    if(a < b)
        return 1;
    if(b<a<=b+5)
        return 2;
    if (a>b+5)
        return 3;
    return 0;
}
treeNode* create_node(int value)
{
    treeNode *new_node = (treeNode*)malloc(sizeof(treeNode));
    if(new_node == NULL)
    {
        fprintf (stderr, "Out of memory!!! (create_node)\n");
        exit(1);
    }
    new_node->item = value;
     new_node->left = NULL;
    new_node->mid = NULL;
    new_node->right = NULL;
    return new_node;
}
int (*comparer)(int, int);
void insertNode(treeNode *root,comparer compare,int item)
```

```
{
if (root == NULL){
      root = create_node(item);
   }
   else{
      int is_left = 0;
      int is_mid = 0;
      int is_right = 0;
      int r;
      treeNode* cursor = root;
      treeNode* prev = NULL;
      while(cursor != NULL){
         r = compare(item,cursor->item);
         prev = cursor;
         if (r==1){
            is_left = 1;
            cursor = cursor->left;
         }else if(r==2){
            is_mid = 1;
            cursor = cursor->mid;
         }else if(r==3){
            is_right = 1;
            cursor = cursor->right;
         }
         }
      if(is_left)
        prev->left = create_node(item);
```

```
        else if(is_mid)
          prev->mid = create_node(item);
        else if(is_right)
          prev->right = create_node(item);
     else{
        printf("duplicate values are not allowed!");
    }
    return root;
}
treeNode * search(treeNode *root,const int item,comparer compare)
{
 if (root == NULL)
    return NULL;
  int r;
 treeNode* cursor = root;
 while (cursor!=NULL){
    r=compare(item,cursor->item);
    if(r==1)
        cursor = cursor->left;
    else if(r == 2)
        cursor = cursor->mid;
    else if(r == 3)
        cursor = cursor->right;
 }
 return cursor;
 }
            //delete a node
```

```
treeNode* delete_node(treeNode* root,int item,comparer compare)
{
   if(root==NULL)
      return NULL;
   treeNode *cursor;
   int r = compare(item,root->item);
   if(r==1)
      root->left = delete_node(root->left,item,compare);
   else if(r==2)
      root->mid = delete_node(root->mid,item,compare);
   else if(r==3)
      root->right = delete_node(root->right,item,compare);
   else{
      if(root->left == NULL && root->mid == NULL)
         {
         cursor = root->right;
         free(root);
         root = cursor;
         }
      else if(root->left == NULL && root->right==NULL){
         cursor = root->mid;
         free(root);
         root = cursor;
      }
      else if(root->right == NULL && root->mid == NULL){
         cursor = root->left;
         free(root);
```

```
            root = cursor;
        }
     }
   return root;
}
void dispose(treeNode* root)
{
    if(root != NULL)
    {
        dispose(root->left);
        dispose(root->mid);
        dispose(root->right);
        free(root);
    }
}

void display(treeNode* nd)
{
    if(nd != NULL)
        printf("%d ",nd->item);
}
void display_tree(treeNode* nd)
{
    if (nd == NULL)
        return;
    /* display node item */
    printf("%d",nd->item);
```

```
    if(nd->left != NULL)
        printf("(L:%d)",nd->left->item);
    if(nd->mid != NULL)
        printf("(R:%d)",nd->mid->item);
    if(nd->right != NULL)
        printf("(R:%d)",nd->right->item);
    printf("\n");
    display_tree(nd->left);
    display_tree(nd->mid);
    display_tree(nd->right);
}
int main(void)
{
    treeNode* root = NULL;
    comparer int_comp = compare;
    int a[SIZE] = {8,3,10,1,6,14,4,7,13,15,2,5,20};
    int i;
    printf("--- C Binary Search Tree ---- \n\n");
    printf("Insert: ");
    for(i = 0; i < SIZE; i++)
    {
        printf("%d ",a[i]);
        root = insert_node(root,int_comp,a[i]);
    }
    printf(" into the tree.\n\n");
    /* display the tree */
    display_tree(root);
```

```
   /* remove element */
int r;
   do
   {
      printf("Enter data to remove, (-1 to exit):");
      scanf("%d",&r);
      if(r == -1)
         break;
      root = delete_node(root,r,int_comp);
      /* display the tree */
      if(root != NULL)
         display_tree(root);
      else
         break;
   }
   while(root != NULL);
   /* search for a node */
   int key = 0;
   treeNode* s;
   while(key != -1)
   {
      printf("Enter item to search (-1 to exit):");
      scanf("%d",&key);
      s = search(root,key,int_comp);
      if(s != NULL)
      {
         printf("Found it %d",s->item);
```

```
            if(s->left != NULL)
                printf("(L: %d)",s->left->item);
            if(s->mid != NULL)
                printf("(R: %d)",s->mid->item);
            if(s->right != NULL)
                printf("(R: %d)",s->right->item);
            printf("\n");
        }
        else
        {
            printf("node %d not found\n",key);
        }
    }
    /* remove the whole tree */
    dispose(root);
    return 0;
}
}
```

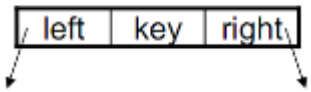
2. Binary tree

(Question: Explain the concept of binary tree- 4 Marks)

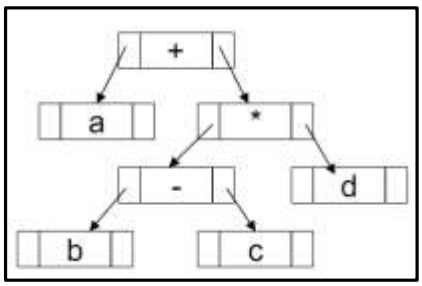
i. A complete binary tree is either an empty binary tree or a binary tree in which:
   - Each level k, k > 0, other than the last level contains the maximum number of nodes for that level that is 2k.
   - The last level may or may not contain the maximum number of nodes.
   - If a slot with a missing node is encountered when scanning the last level in a left to right direction, then all remaining slots in the level must be empty.

ii. Thus, every full binary tree is a complete binary tree, but the opposite is not true.

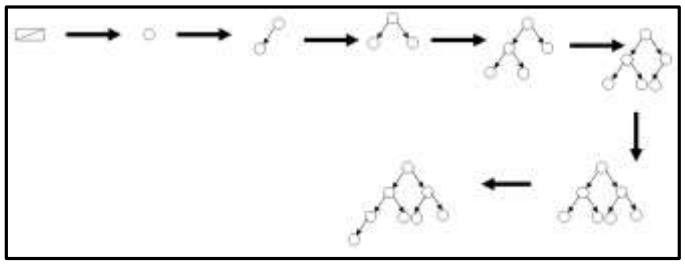
iii. Using double ended link list, binary tree is represented as



iv. Example: A binary tree representing a + (b - c) * d



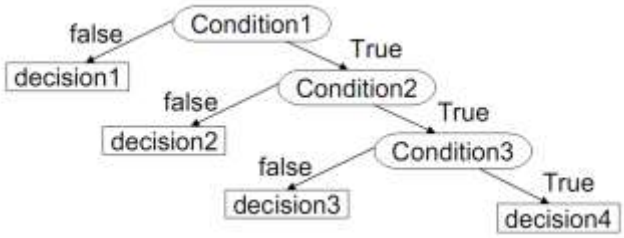
v. Example showing the growth of a complete binary tree:



vi. Application of Binary Trees

1. Binary decision trees.

- Internal nodes are conditions. Leaf nodes denote decisions.



- Expression Tree

3. Binary search tree (BST)

(Question: Explain the concept of binary search tree- 6 Marks)

i. Binary tree is the data structure to maintain data into memory of program.

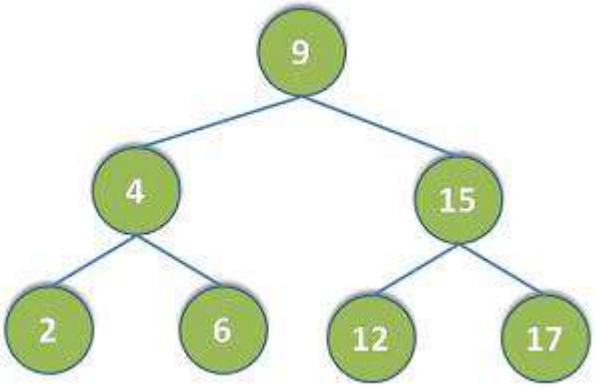
ii. There exists many data structures, but they are chosen for usage on the basis of time consumed in insert/search/delete operations performed on data structures.

iii. Binary tree is one of the data structures that are efficient in insertion and searching operations.

iv. Binary tree works on O (logN) for insert/search/delete operations.

v. Binary tree is basically tree in which each node can have two child nodes and each child node can itself be a small binary tree.

vi. To understand it, below is the example figure of binary tree.



vii. Binary tree works on the rule that child nodes which are lesser than root node keep on the left side and child nodes which are greater than root node keep on the right side.

viii. Same rule is followed in child nodes as well that are itself sub-trees. Like in above figure, nodes (2, 4, and 6) are on left side of root node (9) and nodes (12, 15, and 17) are on right side of root node (9).

Searching into binary tree

(Question: Explain the concept of searching in binary tree- 4 Marks)

i. Searching is done as per value of node to be searched whether it is root node or it lies in left or right sub-tree. It will search node into binary tree.

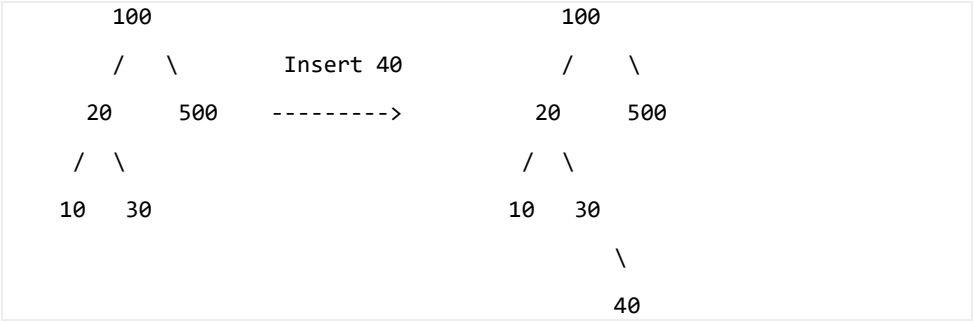
```
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
    if (root->key < key)
        return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```

Insertion of a key

(Question: Explain how to insert insertion of a key in binary tree- 4 Marks)

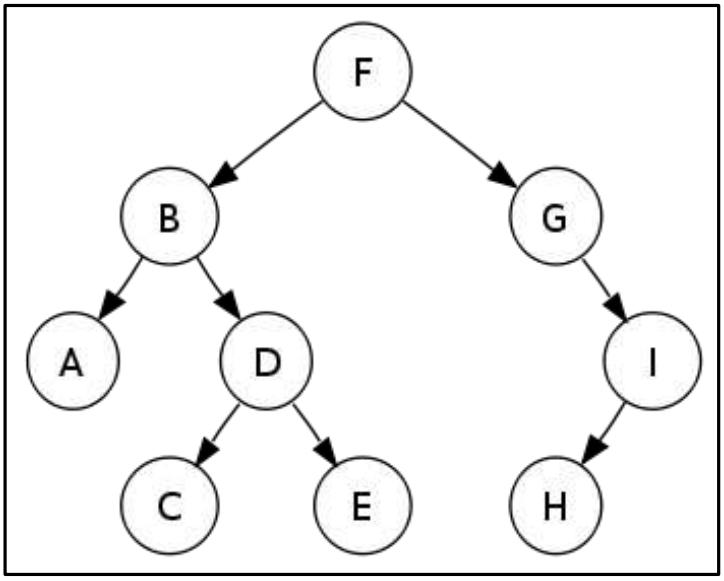
i. A new key is always inserted at leaf.

ii. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.




```
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

6.3 Binary tree traversal



Preorder traversal sequence: F, B, A, D, C, E, G, I, H

(Root, left, right)

(Question: Explain the preorder traversal sequence of a tree- 4 Marks)

i. Visit the root (we will print it when we visit to show the order of visiting)
ii. Traverse the left subtree in pre-order
iii. Traverse the right subtree in pre-order

```
/* Print Pre-order traversal of the tree */
void preOrder(node* r){
   if(r){
      printf("%d", r->data );
      preOrder(r->left);
      preOrder(r->right);
   }
}
```

In order traversal sequence: A, B, C, D, E, F, G, H, I

(Left, root, right)

(Question: Explain the Inorder traversal sequence of a tree- 4 Marks)

i. Visit the root node in between the left and right node (in).
ii. Traverse the left subtree in in-order
iii. Visit the root (we will print it when we visit to show the order of visiting)
iv. Traverse the right subtree in in-order

```
/* Print In-order traversal of the tree */
void inOrder(node* r){
   if(r){
      inOrder(r->left);
      printf("%d", r->data );
      inOrder(r->right);
   }
}
```

Post order traversal sequence: A, C, E, D, B, H, I, G, F

(Left, right, root)

(Question: Explain the post order traversal sequence of a tree- 4 Marks)

i. Traverse the left subtree in in-order
ii. Traverse the right subtree in in-order
iii. Visit the root (we will print it when we visit to show the order of visiting)

```
/* Print Post-order traversal of the tree */
void postOrder(node* r){
   if(r){
      postOrder(r->left);
      postOrder(r->right);
      printf("%d", r->data );
```

```
   }
}
```

Traversal Type	Inorder	Preorder	Postorder
Short Cut	L V R	V L R	L R V

WAP to convert the expression to in order pre order and post order

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
struct BST {
   int data;
   struct BST *lchild, *rchild;
} node;

void insert(node *, node *);
void inorder(node *);
void preorder(node *);
void postorder(node *);
node *search(node *, int, node **);

void main() {
   int choice;
   char ans = 'N';
   int key;
   node *new_node, *root, *tmp, *parent;
   node *get_node();
```

```
root = NULL;
clrscr();

printf("\nProgram For Binary Search Tree ");
do {
  printf("\n1.Create");
  printf("\n2.Search");
  printf("\n3.Recursive Traversals");
  printf("\n4.Exit");
  printf("\nEnter your choice :");
  scanf("%d", &choice);
   switch (choice) {
  case 1:
    do {
      new_node = get_node();
      printf("\nEnter The Element ");
      scanf("%d", &new_node->data);

      if (root == NULL) /* Tree is not Created */
        root = new_node;
      else
        insert(root, new_node);

      printf("\nWant To enter More Elements?(y/n)");
      ans = getch();
    } while (ans == 'y');
    break;
```

```
    case 2:
      printf("\nEnter Element to be searched :");
      scanf("%d", &key);

      tmp = search(root, key, &parent);
      printf("\nParent of node %d is %d", tmp->data, parent->data);
      break;

    case 3:
      if (root == NULL)
        printf("Tree Is Not Created");
      else {
        printf("\nThe Inorder display : ");
        inorder(root);
        printf("\nThe Preorder display : ");
        preorder(root);
        printf("\nThe Postorder display : ");
        postorder(root);
      }
      break;
    }
  } while (choice != 4);
}
/*
 Get new Node
 */
node *get_node() {
```

```
  node *temp;
  temp = (node *) malloc(sizeof(node));
  temp->lchild = NULL;
  temp->rchild = NULL;
  return temp;
}
/*
 This function is for creating a binary search tree
 */
void insert(node *root, node *new_node) {
  if (new_node->data < root->data) {
    if (root->lchild == NULL)
      root->lchild = new_node;
    else
      insert(root->lchild, new_node);
  }

  if (new_node->data > root->data) {
    if (root->rchild == NULL)
      root->rchild = new_node;
    else
      insert(root->rchild, new_node);
  }
}
/*
 This function is for searching the node from
 binary Search Tree
```

```
 */
node *search(node *root, int key, node **parent) {
   node *temp;
   temp = root;
   while (temp != NULL) {
      if (temp->data == key) {
         printf("\nThe %d Element is Present", temp->data);
         return temp;
      }
      *parent = temp;
      if (temp->data > key)
         temp = temp->lchild;
      else
         temp = temp->rchild;
   }
   return NULL;
}
/*
 This function displays the tree in inorder fashion
 */
void inorder(node *temp) {
   if (temp != NULL) {
      inorder(temp->lchild);
      printf("%d", temp->data);
      inorder(temp->rchild);
   }
}
```

```
/*
 This function displays the tree in preorder fashion
 */
void preorder(node *temp) {
   if (temp != NULL) {
      printf("%d", temp->data);
      preorder(temp->lchild);
      preorder(temp->rchild);
   }
}
 /*
 This function displays the tree in postorder
 */
void postorder(node *temp) {
   if (temp != NULL) {
      postorder(temp->lchild);
      postorder(temp->rchild);
      printf("%d", temp->data);
   }
}
```

OPERATIONS ---
1 - Insert an element into tree
2 - Delete an element from the tree
3 - In order Traversal
4 - Preorder Traversal
5 - Post order Traversal
6 - Exit

Enter your choice: 1
Enter data of node to be inserted: 40

Enter your choice: 1
Enter data of node to be inserted: 20

Enter your choice: 1
Enter data of node to be inserted: 10

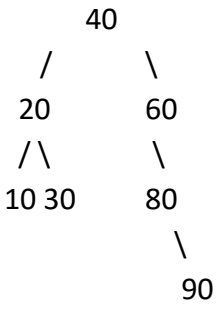
Enter your choice: 1
Enter data of node to be inserted: 30

Enter your choice: 1
Enter data of node to be inserted: 60

Enter your choice: 1
Enter data of node to be inserted: 80

Enter your choice: 1
Enter data of node to be inserted: 90

Enter your choice: 3
10 -> 20 -> 30 -> 40 -> 60 -> 80 -> 90 ->



Graph and Hashing

CONTENTS

Hours: 8

Marks: 16

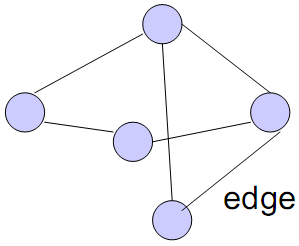
7.1 Introduction to Graphs

1. Terminologies

(Question: Define any graph terminology – 6 marks)

i. A graph consists of:

- A set, V, of vertices (nodes).
- A collection, E, of pairs of vertices from V called edges (arcs).
- Edges, also called arcs, are represented by (u, v) and are either:
  - ✓ Directed if the pairs are ordered (u, v),u the origin, v the destination.
  - ✓ Undirected if the pairs are unordered.



ii. Then a graph can be:

- **Directed graph** (di-graph): if all the edges are directed.
- **Undirected graph** (graph): if all the edges are undirected.
- **Mixed graph** if edges are both directed and undirected.
- **End-vertices** of an edge are the endpoints of the edge.
- **Two vertices** are **adjacent** if they are endpoints of the same edge.
- **An edge** is **incident** on a vertex if the vertex is an endpoint of the edge.
- **Outgoing edges** of a vertex are directed edges that the vertex is the origin.
- **Incoming edges** of a vertex are directed edges that the vertex is the destination.
- **Degree of a vertex**, v, denoted deg(v) is the number of incident edges.
- **Out-degree**, outdeg(v), is the number of outgoing edges.
- **In-degree**, indeg(v), is the number of incoming edges.

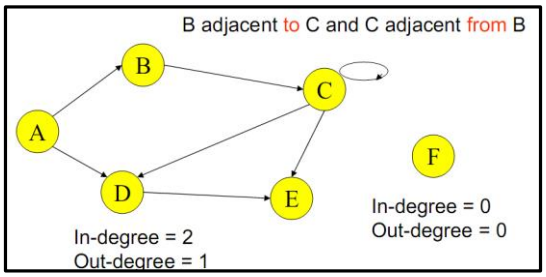


Figure 7.1

- **Parallel edges** or multiple edges are edges of the same type and end-vertices
- **Self-loop** is an edge with the end vertices the same vertex
- **Simple graphs** have no parallel edges or self-loops

7.2 Representations of a graph

1. Array Representation:

i. The graphs are represented using arrays in the form of two dimensional array as adjacency Matrix.

ii. Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

iii. The 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex I to vertex j.

iv. Adjacency matrix for undirected graph is always symmetric.

v. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex I to vertex j with weight w.

vi. For the given undirected graph

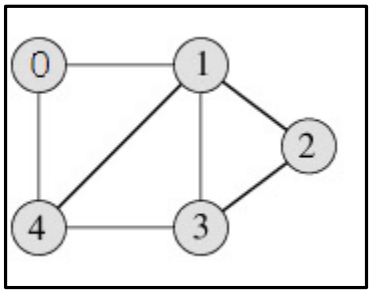


Figure 7.2: Graph

vii. The adjacency matrix for the above graph is as shown below

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure 7.3: Adjacency Matrix

viii. Representation is easier to implement and follow. Removing an edge takes O (1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

ix. Consumes more space O (V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O (V^2) time.

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v)
{
 int i;
 reach[v]=1;
 for(i=1;i<=n;i++)
  if(a[v][i] && !reach[i])
  {
   printf("\n %d->%d",v,i);
   dfs(i);
  }
}

void main()
{
 int i,j,count=0;
 clrscr();
 printf("\n Enter number of vertices:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
  reach[i]=0;
  for(j=1;j<=n;j++)
```

```
  a[i][j]=0;
 }
 printf("\n Enter the adjacency matrix:\n");
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   scanf("%d",&a[i][j]);
 dfs(1);
 printf("\n");
 for(i=1;i<=n;i++)
 {
  if(reach[i])
   count++;
 }
 if(count==n)
  printf("\n Graph is connected");
 else
  printf("\n Graph is not connected");
 getch();
}
```

2. Linked list Representation

(Question: Explain the link list representation of adjacency matrix – 4 Marks)

i. An array of linked lists is used.
ii. Size of the array is equal to number of vertices.
iii. An entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex.
iv. This representation can also be used to represent a weighted graph.
v. The weights of edges can be stored in nodes of linked lists.
vi. Following is adjacency list representation of the above graph.

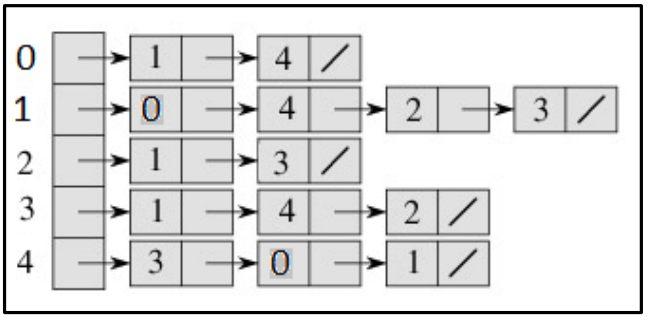


Figure 7.4: Adjacency List

7.3 Traversal of graphs

1. Depth-first search (DFS).

(Question: Explain DFS with diagram – 4 Marks)

i. The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node.

ii. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph.

iii. As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

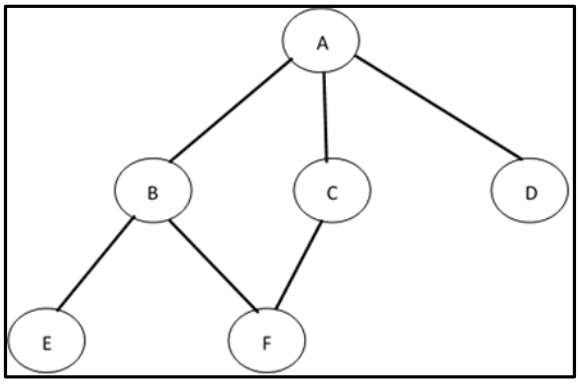


Figure 7.5: DFS

iv. Algorithmic Steps

Step 1: Push the root node in the Stack.

Step 2: Loop until stack is empty.

Step 3: Peek the node of the stack.

Step 4: If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.

Step 5: If the node does not have any unvisited child nodes, pop the node from the stack.

2. Breadth-first search (BFS).

(Question: Explain BFS with diagram – 4 Marks)

i. This is a very different approach for traversing the graph nodes.

ii. The aim of BFS algorithm is to traverse the graph as close as possible to the root node.

iii. Queue is used in the implementation of the breadth first search.

iv. Working of BFS traversal for graph

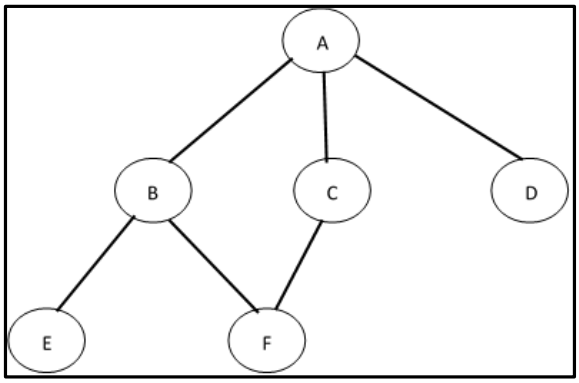


Figure 7.6: BFS

v. If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F".

vi. The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

vii. Algorithmic Steps

Step 1: Push the root node in the Queue.

Step 2: Loop until the queue is empty.

Step 3: Remove the node from the Queue.

Step 4: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

/*Program to implement DFS AND BFS*/

```
#include<stdio.h>
int q[ 20 ], top = -1, front = -1, rear = -1, a[ 20 ][ 20 ], vis[ 20 ], stack[
20 ];
int delete();
void add ( int item );
bfs( int s, int n );
void dfs( int s, int n );
void push( int item );
int pop();
main()
{
   int n, i, s, ch, j;
   char c, dummy;
   printf( "ENTER THE NUMBER VERTICES " );
   scanf( "%d", &n );
   for ( i = 1;i <= n;i++ )
      {
         for ( j = 1;j <= n;j++ )
            {
               printf( "ENTER 1 IF %d HAS A NODE WITH %d ELSE 0
", i, j );
               scanf( "%d", &a[ i ][ j ] );
            }
      }
   printf( "THE ADJACENCY MATRIX IS\n" );
   for ( i = 1;i <= n;i++ )
      {
         for ( j = 1;j <= n;j++ )
            {
               printf( " %d", a[ i ][ j ] );
            }
         printf( "\n" );
      }
   do
      {
         for ( i = 1;i <= n;i++ )
            vis[ i ] = 0;
         printf( "\nMENU" );
         printf( "\n1.B.F.S" );
         printf( "\n2.D.F.S" );
         printf( "\nENTER YOUR CHOICE" );
         scanf( "%d", &ch );
         printf( "ENTER THE SOURCE VERTEX :" );
```

```
            scanf( "%d", &s );
            switch ( ch )
               {
               case 1:
                  bfs( s, n );
                  break;
               case 2:

                  dfs( s, n );
                  break;
               }
            printf( "DO U WANT TO CONTINUE(Y/N) ? " );
            scanf( "%c", &dummy );
            scanf( "%c", &c );
         }
      while ( ( c == 'y' ) || ( c == 'Y' ) );
}
void bfs( int s, int n )
{
   int p, i;
   add
      ( s );
   vis[ s ] = 1;
   p = delete();
   if ( p != 0 )
      printf( " %d", p );
   while ( p != 0 )
      {
         for ( i = 1;i <= n;i++ )
            if ( ( a[ p ][ i ] != 0 ) && ( vis[ i ] == 0 ) )
               {
                  add
                     ( i );
                  vis[ i ] = 1;
               }
         p = delete();
         if ( p != 0 )
            printf( " %d ", p );
      }
   for ( i = 1;i <= n;i++ )
      if ( vis[ i ] == 0 )
         bfs( i, n );
}
void add( int item )
   {
```

```
        if ( rear == 19 )
            printf( "QUEUE FULL" );

        else
            {
                if ( rear == -1 )
                    {
                        q[ ++rear ] = item;
                        front++;
                    }
                else
                    q[ ++rear ] = item;
            }
    }
int delete()
{
    int k;
    if ( ( front > rear ) || ( front == -1 ) )
        return ( 0 );
    else
        {
            k = q[ front++ ];
            return ( k );
        }
}
void dfs( int s, int n )
{
    int i, k;
    push( s );
    vis[ s ] = 1;
    k = pop();
    if ( k != 0 )
        printf( " %d ", k );
    while ( k != 0 )
        {
            for ( i = 1;i <= n;i++ )
                if ( ( a[ k ][ i ] != 0 ) && ( vis[ i ] == 0 ) )
                    {
                        push( i );
                        vis[ i ] = 1;
                    }
            k = pop();
            if ( k != 0 )
                printf( " %d ", k );
        }
```

```
    for ( i = 1;i <= n;i++ )
        if ( vis[ i ] == 0 )
            dfs( i, n );
}
void push( int item )
{
    if ( top == 19 )
        printf( "Stack overflow " );
    else
        stack[ ++top ] = item;
}
int pop()
{
    int k;
    if ( top == -1 )
        return ( 0 );
    else
        {
            k = stack[ top– ];
            return ( k );
        }
}
```

7.4 Applications of Graph

i. Weighted Graphs(to Analyze network traffic)
ii. Digraphs.
iii. To find Shortest Path First using Prims, Krushkal spanning tree.

7.5 Hashing

1. Hash function

(Question: Explain hashing with diagram – 4 Marks)

i. The problem at hands is to speed up searching.
ii. Consider the problem of searching an array for a given value.
iii. If the array is not sorted, the search might require examining each and all elements of the array.
iv. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n).
v. We could search even faster if we know in advance the index at which that value is located in the array.

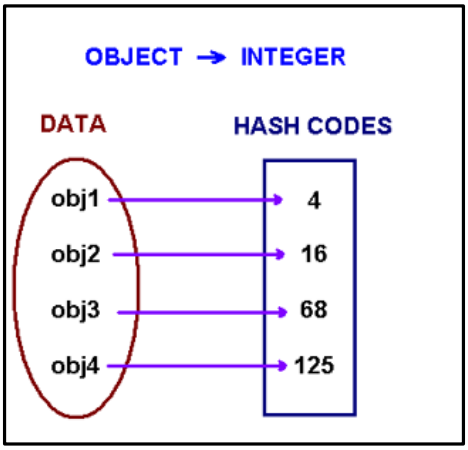


Figure 7.7: Hashing

vi. A hash function is a function which when given a key, generates an address in the table.

vii. A hash function that returns a unique hash number is called a universal hash function.

viii. In practice it is extremely hard to assign unique numbers to objects.

ix. The latter is always possible only if you know (or approximate) the number of objects to be processed.

x. Thus, we say that our hash function has the following properties

- It always returns a number for an object.
- Two equal objects will always have the same number
- Two unequal objects not always have different numbers

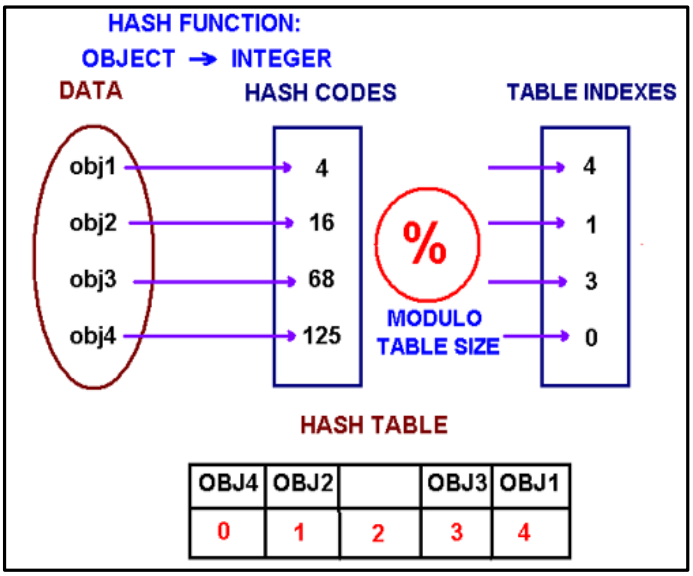


Figure7.9: Hash Table

2. Collision resolution techniques

i. When we put objects into a hash table, it is possible that different objects (by the equals () method) might have the same hash code.

ii. This is called a collision.

iii. Example of collision. Two different strings ""Aa" and "BB" have the same key:
"Aa" = 'A' * 31 + 'a' = 2112
"BB" = 'B' * 31 + 'B' = 2112

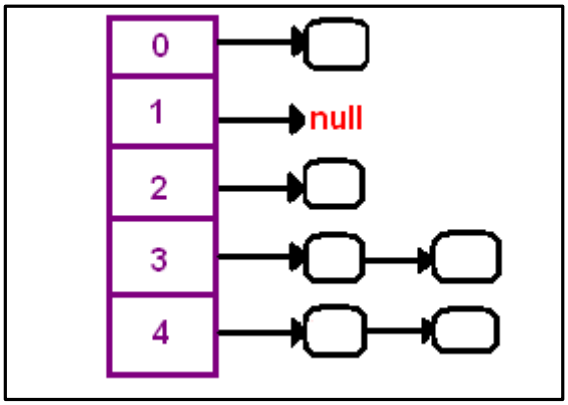


Figure 7.10: Collision

7.6 AVL Trees

i. An AVL tree (Georgy Adelson-Velsky and Evgenii Landis' tree, named after the inventors) is a self-balancing binary search tree.

ii. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

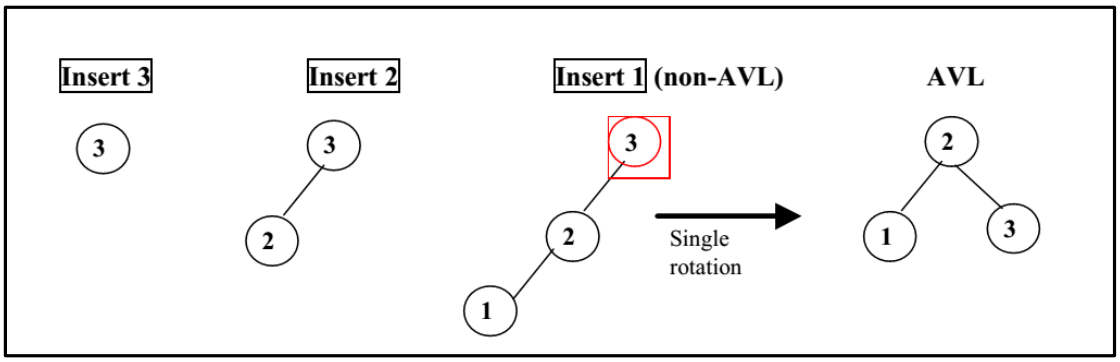


Figure 7.11: Step 1

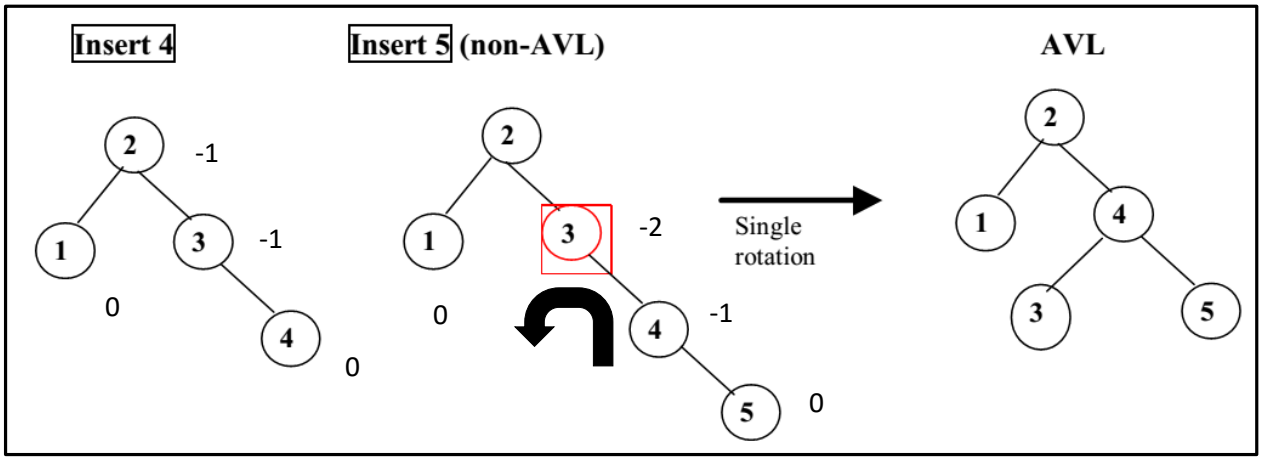


Figure 7.12: Step 2

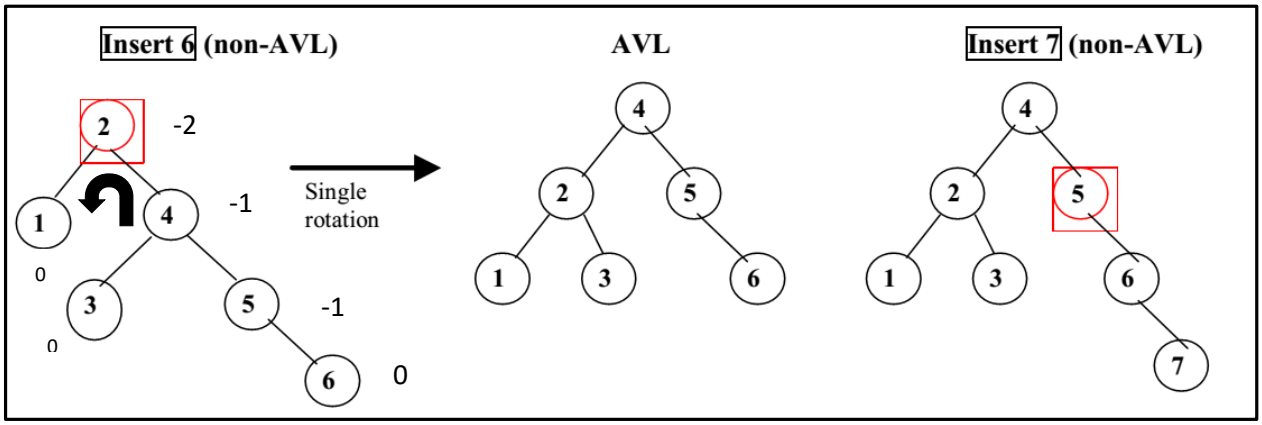


Figure 7.13: Step 3

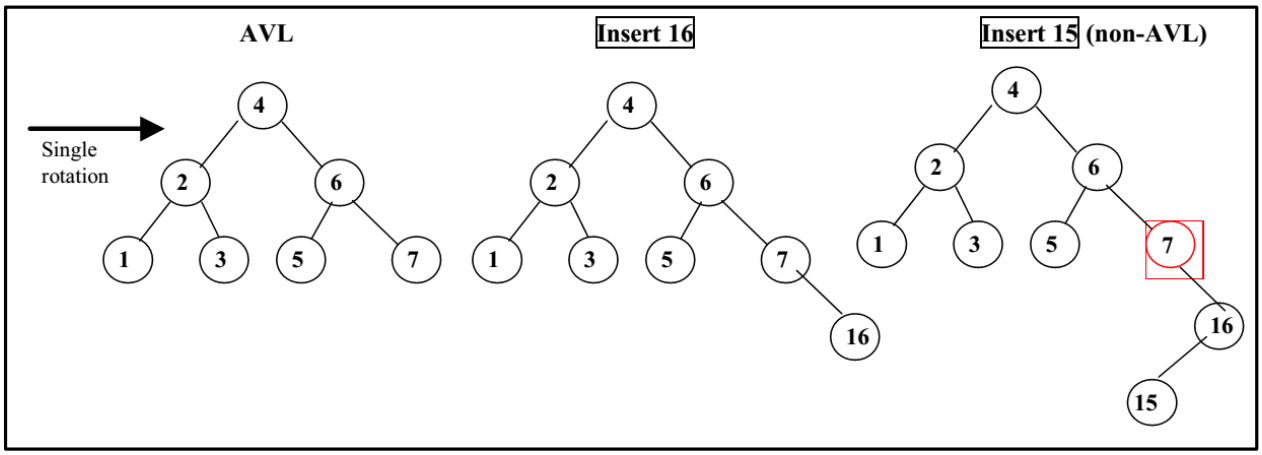


Figure 7.14: Step 4

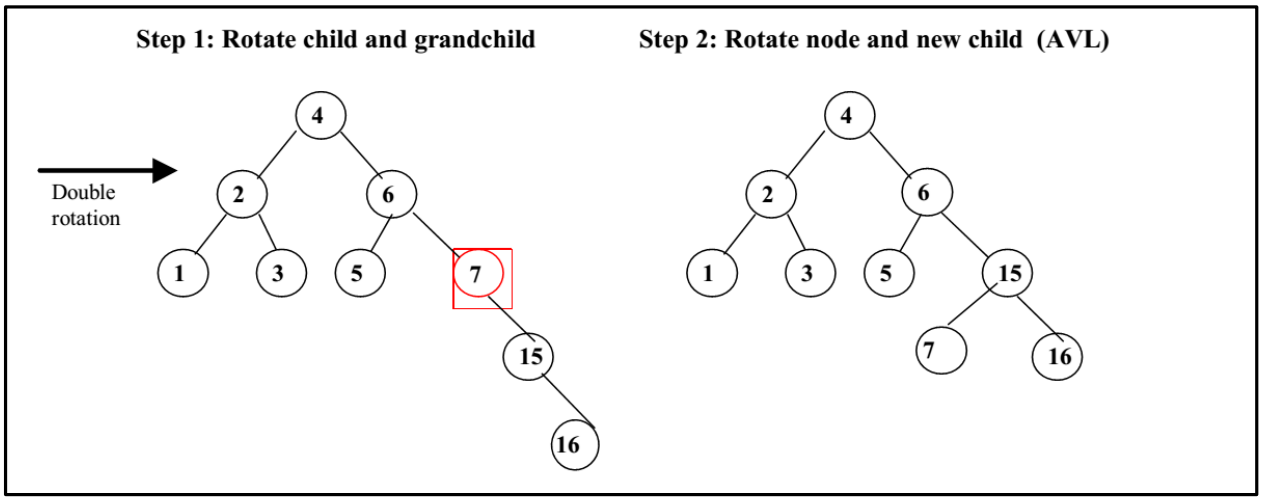


Figure 7.15: Step 5

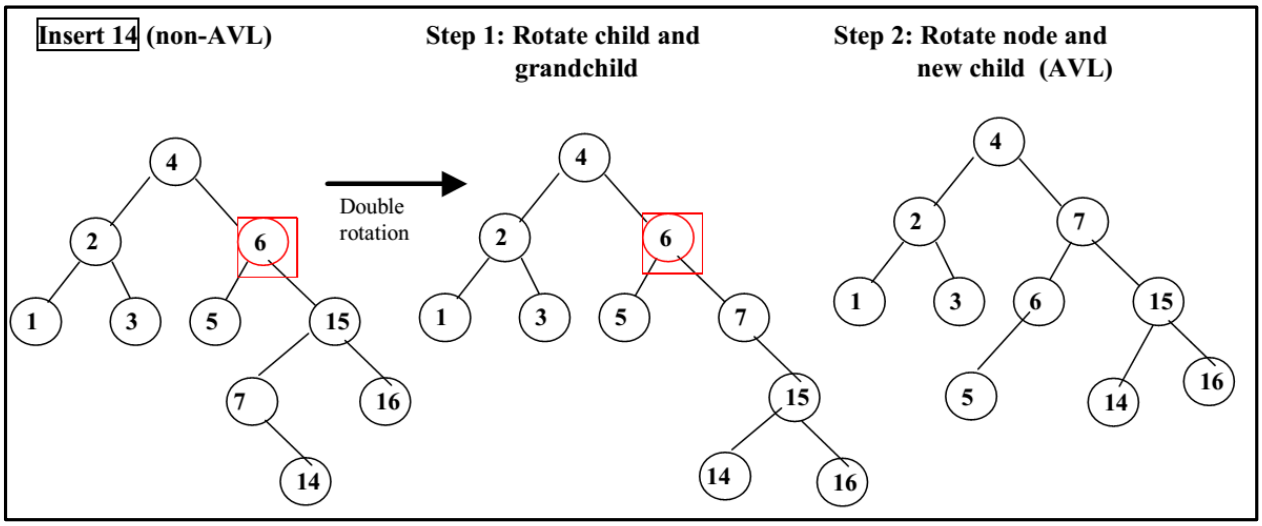


Figure 7.16: Step 6