TRIBHUVAN UNIVERSITY

Institution of Science and Technology

Bachelor Level/Second Year/Third Semester/Science
Computer Science and Information Technology [CSC 206]

Full Marks: 60
Pass Marks: 24
Time: 3 hrs.

TU QUESTIONS-ANSWERS 2077

Bachelor Level/Second Year/Third Semester/Science
Computer Science and Information Technology
Data Structure and Algorithms

Full Marks: 60
Pass Marks: 24
Time: 3 hrs.

Section A

Attempt any two questions. **(2 × 10 = 20)**

1. What is stack? What are the different applications of stack? Explain stack operations with example. **(1 + 3 + 7)**

Ans: A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack. Stack uses a variable called top which points topmost element in the stack. top is incremented while **pushing** (inserting) an element in to the stack and decremented while **popping** (deleting) an element from the stack.

A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

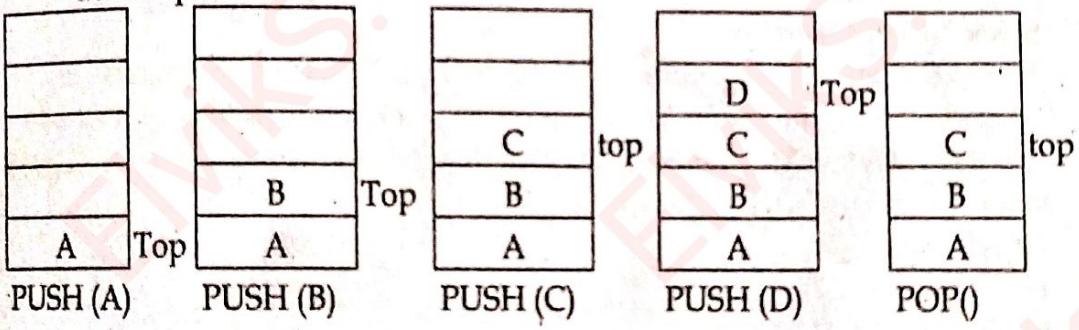


Fig: push, pop operations in Stack

Applications of Stack

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To check the correctness of parentheses sequence
- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures
- Tree Traversals

Stack operations

Structure for stack

```
#define MAXSIZE 10
struct stack
{
int items[MAXSIZE]; //Declaring an array to store items
int top;      //Top of a stack
};
typedef struct stack st;
```

Creating Empty stack

The value of top=-1 indicates the empty stack in C implementation.

```
void create_empty_stack(struct stack e) /*Function to create an empty stack*/
{
        e.top=-1;
}
```

Stack Empty or Underflow

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int IsEmpty()
{
if(top==-1)
return 1;
else
return 0;
}
```

Stack Full or Overflow

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAXSIZE- 1) of the stack. The following function returns true (1) if stack is full, false (0) otherwise.

```
int IsFull()
{
if(top==MAXSIZE-1)
return 1;
else
return 0;
  }
```

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

- Increment the variable Top so that it can now refer to the next memory location.
- Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is over flown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm for Push Operation of Stack

Let Stack [MAXSIZE] is an array to implement the stack. The variable top denotes the top of the stack. This algorithm adds or inserts an item at the top of the stack

1. Start
2. Check for stack overflow as
   if top==MAXSIZE-1 then
   print "Stack Overflow" and Exit the program
   else
   Increase top by 1 as
   Set, top=top+1
3. Read elements to be inserted say element
4. Set, Stack[top]= element //Inserts item in new top position
5. Stop

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be decremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in another variable and then the top is decremented by 1. The operation returns the deleted value that was stored in another variable as the result. The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm for POP operation

This algorithm deletes the top element of the stack and assigns it to a variable element

1. Start
2. Check for the stack Underflow as
   If top<0 then
   Print "Stack Underflow" and Exit the program
   else
   Remove the top element and set this element to the variable as
   Set, element=Stack [top]
   Decrement top by 1 as
   Set, top=top-1
3. Print 'element' as a deleted item from the stack
4. Stop

Visiting each element of the stack (Peek or Display operation)

Display operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Display algorithm

1. Start
2. Check for the stack Underflow as
   If top<0 then
   Print "Stack Underflow" and Exit the program
   Else

Set, i=top
While (i>=0)
Display stack[i]
Decrement i by 1 as
Set, i=i-1.
End while
3. Stop

2. **Differentiate between singly linked list and doubly linked list. How do you insert and delete a node from doubly linked list? Explain.** (3+7)

Ans: Both Singly linked list and Doubly linked list are the executions of a Linked list. The singly-linked list holds data and a link to the next component. While in a doubly-linked list, every node includes a link to the previous node.

Difference between Singly Linked List and Doubly Linked List

Singly linked list	Doubly linked list
In case of singly linked lists, the complexity of insertion and deletion is O(n)	In case of doubly linked lists, the complexity of insertion and deletion is O(1)
The Singly linked list has two segments: data and link.	The doubly linked list has three segments. First is data and second, third are the pointers.
It permits traversal components only in one way.	It permits two way traversal.
We mostly prefer a singly linked list for the execution of stacks.	We can use a doubly linked list to execute binary trees, heaps and stacks.
When we want to save memory and do not need to perform searching, we prefer a singly linked list.	In case of better implementation, while searching, we prefer a doubly linked list.
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.

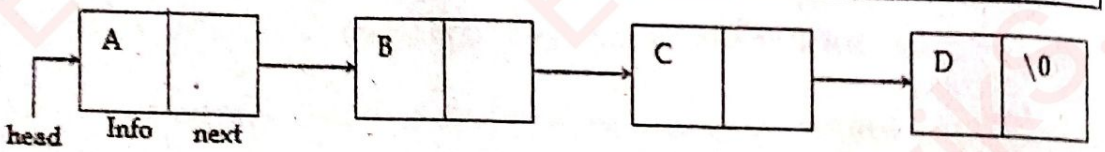


Fig: - Singly linked list with four nodes

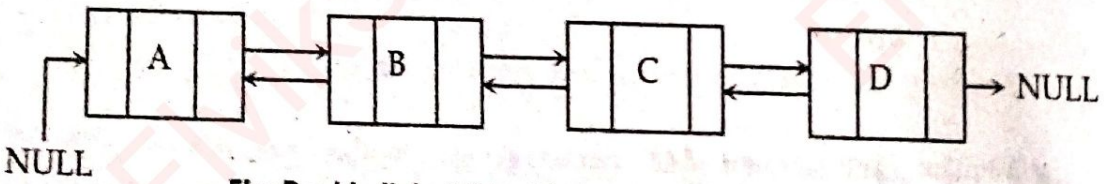


Fig: Doubly linked list with three nodes

Representation of doubly linked list

```
struct node
{
int info;
struct node *prev;
struct node *next;
};
typedef struct node NodeType;
```

```
NodeType *first=NULL;
NodeType *last=NULL;
```

New element is inserted at the specified position in the doubly linked list

To insert new node at specific position in the DLL, at first a new node with given element is created. If the insert position is 1, then the new node is made to head. Otherwise, traverse to the node that is previous to the insert position and check if it is null or not. In case of null, the specified position does not exist. In other case, update the links. The below figure describes the process, if the insert node is other than the head node.

The function push at is created for this purpose. It is a 6-step process.

```
void push_at(int newElement, int position)
{
 //1. allocate node to new element
 Node* newNode = new Node();
 newNode->data = newElement;
 newNode->next = NULL;
 newNode->prev = NULL;

 //2. check if the position is > 0
 if(position < 1)
 {
  cout<<"\nposition should be >= 1.";
 }
 else if (position == 1)
 {
  //3. if the position is 1, make new node as head
  newNode->next = head;
  head->prev = newNode;
  head = newNode;
 }
 else {
  //4. Else, make a temp node and traverse to the
  //   node previous to the position
  Node* temp = head;
  for(int i = 1; i < position-1; i++)
  {
      if(temp != NULL)
     {
            temp = temp->next;
      }
  }
//5. If the previous node is not null, adjust the links
 if(temp != NULL)
 {
   newNode->next = temp->next;
   newNode->prev = temp;
   temp->next = newNode;
   if(newNode->next != NULL)
```

```
            newNode->next->prev = newNode;
    }
    else
    {
        //6. When the previous node is null
        cout<<"\n The previous node is null.";
    }
  }
}
```

A node at the specified position in the doubly linked list is deleted

First, the specified position must be greater than equal to 1. If the specified position is 1 and head is not null, then make the head next as head and delete the previous head. Else, traverse to the node that is previous to the specified position. If the specified node and previous to the specified node are not null then adjust the link. In other case, the specified node will be already null. The below figure describes the process, if the deletion node is other than the head node.

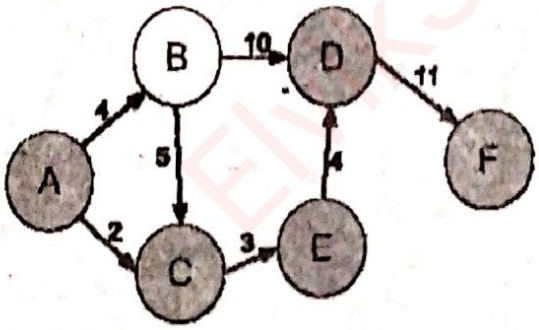
The function pop at is created for this purpose. It is a 5-step process.

```
void pop_at(int position)
{
 //1. check if the position is > 0
 if(position < 1)
 {
  cout<<"\nposition should be >= 1.";
 }
 else if (position == 1 && head != NULL)
 {
  //2. Make head next as head and     delete previous head
  Node* nodeToDelete = head;
  head = head->next;
  free(nodeToDelete);
  if(head != NULL)
        head->prev = NULL;
 }
 else
 {
  //3. Else, make a temp node and traverse to the
  //   node previous to the position
  Node* temp = head;
  for(int i = 1; i < position-1; i++)
  {
    if(temp != NULL)
    {
     temp = temp->next;
    }
  }
 //4. If the previous node and next of the previous is not null, adjust links
  if(temp != NULL && temp->next != NULL)
  {
```

```
        Node* nodeToDelete = temp->next;
        temp->next = temp->next->next;
        if(temp->next->next != NULL)
         temp->next->next->prev = temp->next;
        free(nodeToDelete);
      }
      else
      {
       //5. Else the given node will be empty.
       cout<<"\n The node is already null.";
      }
    }
}
```

3. What is shortest path? Explain Dijkstra algorithm for finding shortest path using suitable example. **(2+8)**

Ans: In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



Dijkstra's algorithm

This algorithm makes a tree of the shortest path from the starting node, the source, to all other nodes (points) in the graph.

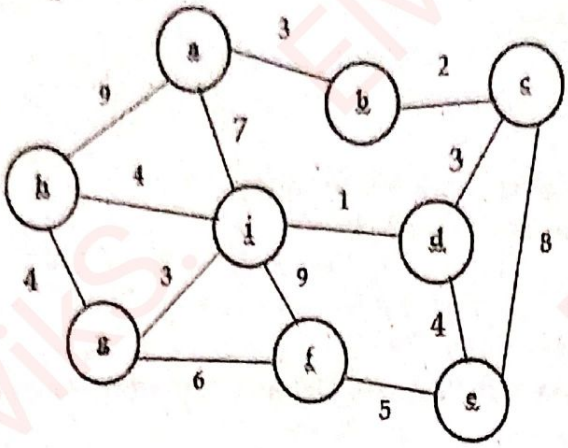
Dijkstra's algorithm makes use of weights of the edges for finding the path that minimizes the total distance (weight) among the source node and all other nodes. This algorithm is also known as the single-source shortest path algorithm.

step by step process of algorithm implementation

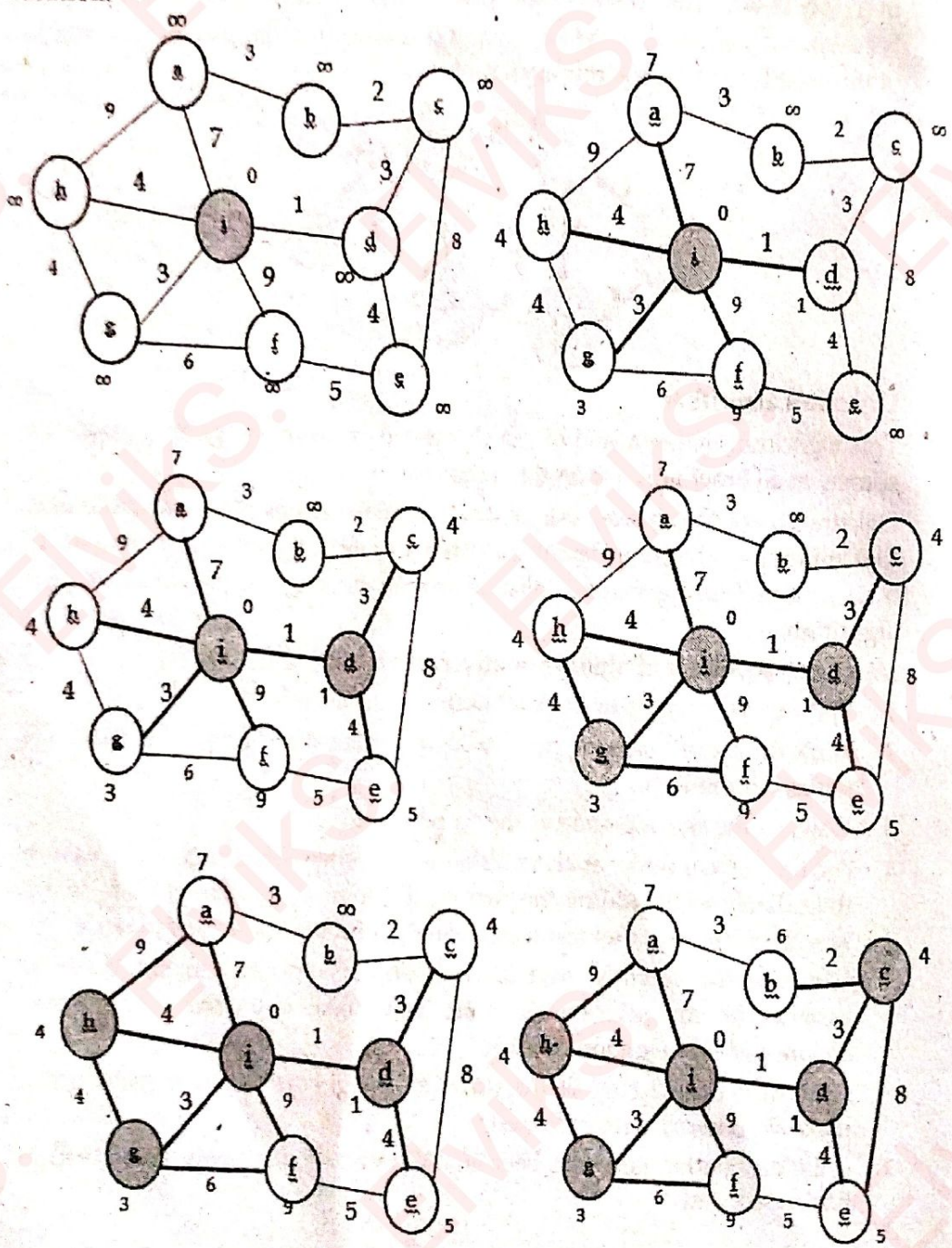
1. The very first step is to mark all nodes as unvisited,
2. Mark the picked starting node with a current distance of 0 and the rest nodes with infinity,
3. Now, fix the starting node as the current node,
4. For the current node, analyze all of its unvisited neighbors and measure their distances by adding the current distance of the current node to the weight of the edge that connects the neighbor node and current node,
5. Compare the recently measured distance with the current distance assigned to the neighboring node and make it as the new current distance of the neighboring node,
6. After that, consider all of the unvisited neighbors of the current node, mark the current node as visited,
7. If the destination node has been marked visited then stop, an algorithm has ended, and

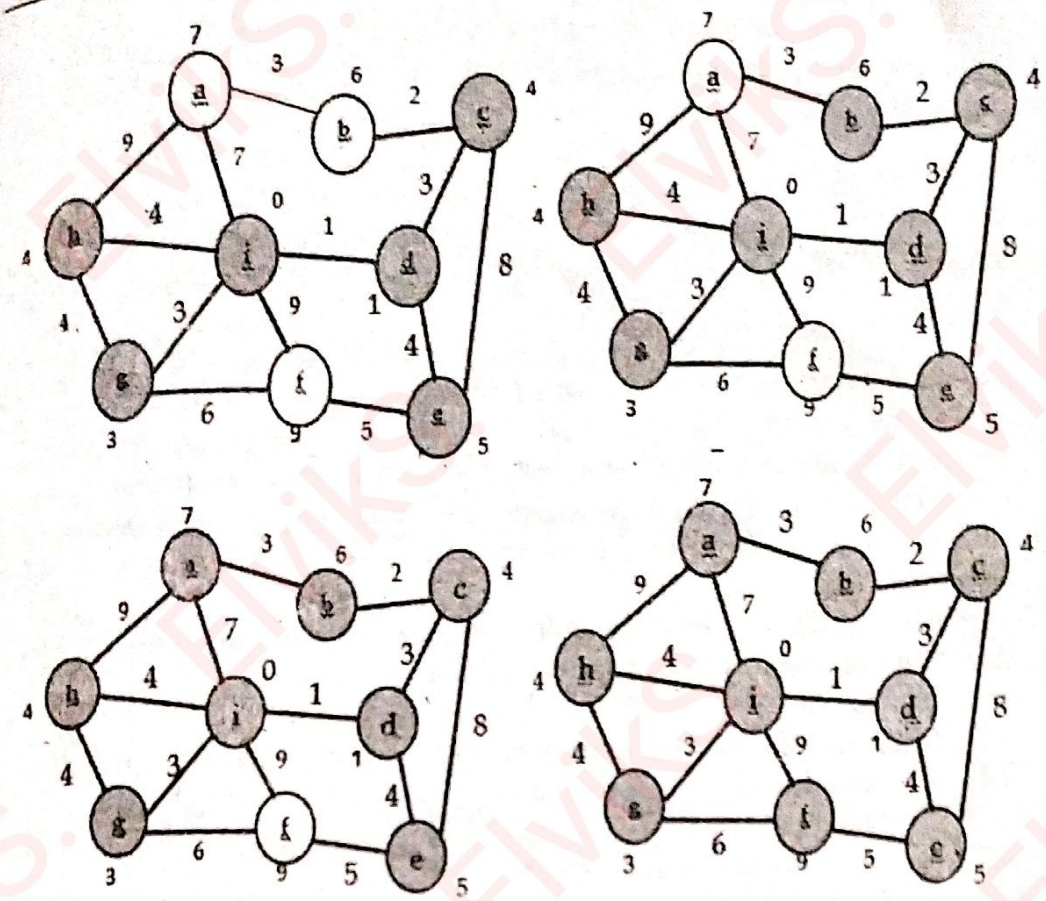
8. Else, choose the unvisited node that is marked with the least distance, fix it as the new current node, and repeat the process again from step 4.

Example: Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.



Solution:





Short questions:

Attempt any Eight questions: **(8 x 5 = 40)**

4. **What is dynamiç memory allocation? Compare data structure with abstract data type.** **(2+3)**

Ans: Allocation of memory at the time of execution (run time) is known as dynamic memory allocation. The functions calloc() and malloc() support allocating of dynamic memory. Dynamic allocation of memory space is done by using these functions when value is returned by functions and assigned to pointer variables. In this case, variables get allocated only if your program unit gets active. It uses the data structure called heap for implementing dynamic allocation. There is memory reusability and memory can be freed when not required.

Example: Program to find sum of n different numbers by using malloc function for DMA.

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
void main()
{
   int n, i, *a, s=0;
   printf("Enter number of elements");
   scanf("%d",&n);
   a=(int*)malloc(sizeof(int)*n);
   printf("Enter %d numbers", n);
   for(i=0; i<n; i++)
   {
```

```
        scanf("%d", (a+i));
    }
    printf("Sum of given numbers is:");
    for(i=0;i<n;i++)
    {
      s=s+*(a+i);
    }
    printf("%d", s);
    getch();
    }
```

Compare data structure with abstract data type

ADT is the logical picture of the data and the operations to manipulate the component elements of the data. Data structure is the actual representation of the data during the implementation and the algorithms to manipulate the data elements. ADT is in the logical level and data structure is in the implementation level.

5. Explain algorithm for evaluation of postfix expression using stack. (5)

Ans: Algorithm to evaluate the prefix expression

Here we use only one stack called vstack (value stack).

1. Scan one character at a time from right to left of given prefix expression
1.1 if scanned symbol is operand then
- read its corresponding value and push it into vstack
1.2 if scanned symbol is operator then
- pop and place into op1
- pop and place into op2
- Compute result according to given operator and push result into vstack
2. Pop and display which is required value of the given prefix expression
3. Stop

Trace of Evaluation of prefix expression

Consider an example to evaluate the prefix expression tracing the algorithm

+A-/B$CD*E+F-/GHI where A=1, B=2, C=3, D=2, E=1, F=5, G=9, H=3, I=2

Scanned character	Value	Op1	Op2	Result	vstack
I	2	.....	.....	.....	2
H	3	.....	.....	.....	23
G	9	.....	.....	.....	239
/	.....	9	3	3	23
-	.....	3	2	1	1
F	5	.....	.....	.....	15
+	.....	5	1	6	6
E	1	.....	.....	.....	61
*	.....	1	6	6	6
D	2	.....	.....	.....	62
C	3	.....	.....	.....	623
$	.....	3	2	9	69
B	2	.....	.....	.....	692
/	.....	2	9	0	60
-	.....	0	6	-6	-6
A	1	.....	.....	.....	-61
+	.....	1	-6	-5	-5

Its final value is -5.

6. **Explain queue as an ADT.** (5)

Ans: A queue q of type T is a finite sequence of elements with the operations

- **MakeEmpty(q):** To make q as an empty queue
- **IsEmpty(q):** To check whether the queue q is empty or not. Return true if q is empty, return false otherwise.
- **IsFull(q):** To check whether the queue q is full or not. Return true if q is full, return false otherwise.
- **Enqueue(q, x):** To insert an item x at the rear of the queue, if and only if q is not full.
- **Dequeue(q):** To delete an item from the front of the queue q. if and only if q is not empty.
- **Traverse (q):** To read entire queue that is display the content of the queue.

7. **Write a recursive program to find GCD of two numbers.** (5)

Ans:

```
#include <stdio.h>
int GCD(int, int );
int main( )
{
    int a, b, g;
    printf("Enter any two numbers");
    scanf("%d%d", &a, &b);
    g=GCD(a, b);
    printf("GCD of given two numbers=%d", g);
    return 0;
}
int GCD(int x, int y)
{
    if(y==0)
    return x;
    else
    return (GCD(y, x%y));
}
```

8. **What is linked list? How is it different from array?** (2+3)

Ans: Arrays work well for unordered sequences and even for ordered sequences if they don't change much. But if we want to maintain an ordered list that allows quick insertions and deletions, we should use a linked data structure. The basic linked list consists of a collection of connected, dynamically allocated nodes.

Linked List

To overcome the disadvantage of fixed size arrays linked list were introduced. A linked list consists of nodes of data which are connected with each other. Every node consists of two fields' data (info) and the link (next). The nodes are created dynamically.

- **Info:** the actual element to be stored in the list. It is also called data field.
- **Link:** one or two links that point to next and previous node in the list. It is also called next or pointer field.

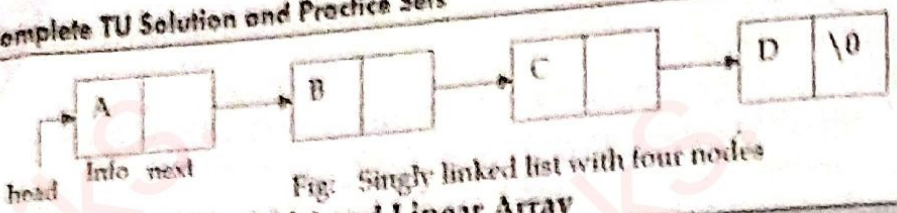


Fig: Singly linked list with four nodes

Difference between Linked List and Linear Array

Array	Linked List
Insertions and deletions are difficult.	Insertions and deletions can be done easily.
It needs movements of elements for insertion and deletion	It does not need movement of nodes for insertion and deletion.
In it space is wasted	In it space is not wasted.
It is more expensive	It is less expensive.
It requires less space as only information is stored	It requires more space as pointers are also stored along with information.
Its size is fixed	Its size is not fixed.

9. Hand test bubble sort with array of numbers 53, 42, 78, 3, 5, 2, 15 in ascending order. **(4+1)**

Ans: A[ ]={25, 57, 48, 37, 12, 92, 86, 33}

Array position	0	1	2	3	4	5	6
Initial state	53	42	78	3	5	2	15
Pass 1	42	53	78	3	5	2	15
	42	53	78	3	5	2	15
	42	53	3	78	5	2	15
	42	53	3	5	78	2	15
	42	53	3	5	2	78	15
	42	53	3	5	2	15	78
	42	53	3	5	2	15	78
Pass 2	42	53	3	5	2	15	78
	42	53	3	5	2	15	78
	42	3	53	5	2	15	78
	42	3	5	53	2	15	78
	42	3	5	2	53	15	78
	42	3	5	2	15	53	78
	42	3	5	2	15	53	78
Pass 3	3	5	2	15	42	53	78
Pass 4	3	2	5	15	42	53	78
Pass 5	2	3	5	15	42	53	78
Pass 6	2	3	5	15	42	53	78
Pass 7	2	3	5	15	42	53	78

10. What is hashing? Explain concept of hash table and hash function with example. **(1 + 4)**

Ans: Hashing

It is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

Hashing is a different approach to searching which calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from O(n), as in a sequential search, or from O(log n), as in a binary search, to at least O(1); regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated.

Hashing is a method and useful technique to implement dictionaries. This method is used to perform searching, insertion and deletion at a faster rate. A function called Hash Function is used to compute and return position of the record instead of searching with comparisons. The data is stored in array called as Hash table. The Hash Function maps keys into positions in a hash table. The mapping of keys to indices of a hash table is known as Hash Function. The major requirement of hash function is to map equal keys to equal indices.

Given a key, the algorithm computes an index that suggests where the entry can be found:

Index = f(key, array_size)

The value of index is determined by 2 steps

- hash = hash_func(key)
- index = hash % array_size

Hash Function

A function that transforms a key into a table index is called a hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value **h(item)=item%11**. Table below gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table: Simple Hash Function Using Remainders

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure below.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

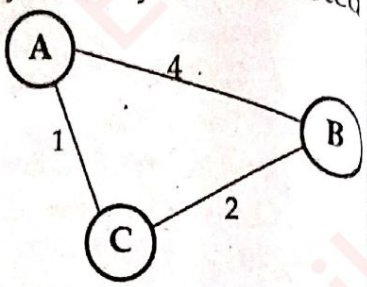
Figure: Hash Table with Six Items

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is O(1), since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

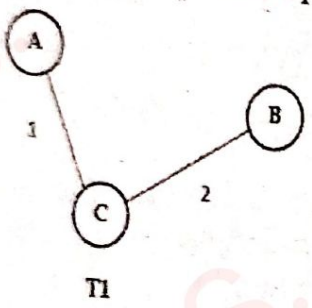
11. What is minimum spanning tree? Explain. (5)

Ans: A minimum spanning tree in a connected weighted graph is a spanning tree that has the smallest possible total weights of its edges out of all possible spanning trees. In other words in a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

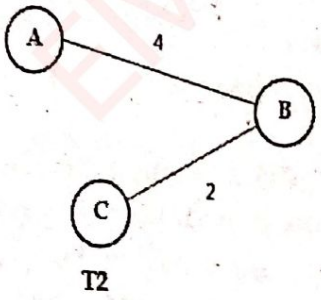
Example: Find MST of given graph



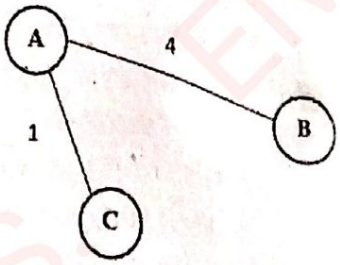
Solution: The possible spanning trees of given graph are:



T1



T2



T3

Total weight of T1 = 3
Total weight of T2 = 6
Total weight of T3=5

Since T1 has minimum weight out of all possible spanning trees of given graph hence, tree T1 acts as minimum spanning tree of given graph.

12. Write short notes on:

a. Tail recursion (2 x 2.5 = 5)

Ans: A function that returns the value of its recursive call is said to be tail **recursive.** Simply a function is said to be tail recursive if there are no any calculations occur in the recursive stage and it only returns the value.

Example: Complete program in C to showing the use of tail recursion

```
Fact (n, accumulator)
{
  if (n == 0)
return accumulator;
      else
    return Fact(n - 1, n * accumulator);
}
Factorial (n)
{
    return Fact(n, 1);
}
```

```
void main( )
{
    int num, f;
    printf("Enter any number");
    scanf("%d", &num);
    f=Factorial (num);
    printf ("Factorial of given number is %d", f);
    getch( );
}
```

b. Collision resolution techniques

Ans: When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing. Some popular methods for minimizing collision are:

- Open addressing
  - ✓ Linear probing
  - ✓ Quadratic probing
  - ✓ Double hashing
- Rehashing
- Chaining
- Hashing using buckets etc.

Linear probing

A hash table in which a collision is resolved by putting the item in the next empty place within the occupied array space is called linear probing. It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location. Hence this method searches in straight line, and it is therefore called linear probing. The main disadvantage of this process is clustering problem.

Example: Insert keys {89, 18, 49, 58, 69} with the hash function h(x) =x mod 10 using linear probing.

Solution: when x=89

h(89)=89%10=9

Insert key 89 in hash-table in location 9

When x=18

h(18)=18%10=8

Insert key 18 in hash-table in location 8

When x=49

h(49)=49%10=9 (Collision occur)

So insert key 49 in hash-table in next possible vacant location of 9 is 0

When x=58

h(58)=58%10=8 (Collision occur)

Insert key 58 in hash-table in next possible vacant location of 8 is 1 (since 9, 0 already contains values).

When x=69

h(89)=69%10=9 (Collision occur)

Insert key 69 in hash-table in next possible vacant location of 9 is 2 (since 0, 1 already contains values).

0	1	2	3	4	5	6	7	8	9
49	58	69	None	None	None	None	None	18	89

TU QUESTIONS-ANSWERS 2078

Bachelor Level/Second Year/Third Semester/Science
Computer Science and Information Technology
Data Structure and Algorithms

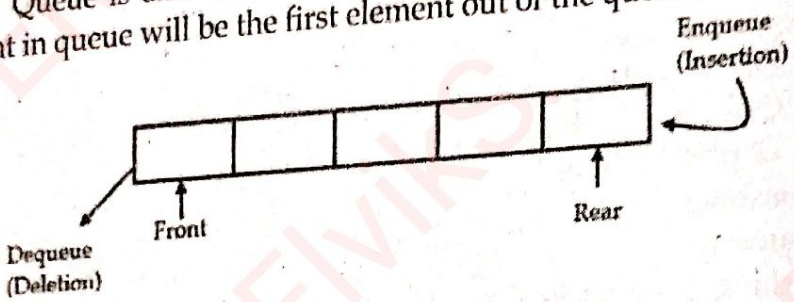
Full Marks: 6
Pass Marks: 2
Time: 3 hr

Section A $(2 \times 10 = 20$

Attempt any two questions.

1. **Define Queue. What are different applications of queue? Explain queue operations with example.**

Ans: Queue is a Linear Data Structure which follows First in First out mechanism. It means: the first element inserted is the first one to be removed. Queue uses two variables rear and front. Rear is incremented while inserting an element into the queue and front is incremented while deleting element from the queue. Queue is also called First in First out (FIFO) system since the first element in queue will be the first element out of the queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
2. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
3. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g. pipes, file IO, sockets.
4. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
5. Queue is used to maintain the play list in media players in order to add and remove the songs from the play-list.
6. Queues are used in operating systems for handling interrupts.
7. Task waiting for the printing

Structure of linear queue

```
#define SIZE 10
Struct LQueue
{
    int rear;
    int front;
    int item[SIZE];
}; typedef Structu LQueue qt;
qt rear, front;
rear=-1;
front=0;
```

Algorithm for insertion an item in queue (Enqueue)

1. Start
2. Initialize front=0 and rear=-1

If rear>=MAXSIZE-1
Print "queue overflow" and return
Else
Set, rear=rear+1
queue[rear]=item

3. Stop

Algorithm to delete an element from the queue (Dequeue)

1. Start
2. If rear<front

Print "queue is empty" and return
Else
Item=queue [front++]
Print "item" as a deleted element

3. Stop

Defining the operations of linear queue

The MakeEmpty function

```
void makeEmpty(qt *q)
{
q→rear=-1;
q→front=0;
}
```

The IsEmpty function

```
int IsEmpty(qt *q)
{
if(q→rear<q→front)
return 1;
else
return 0;
}
```

The Isfull function

```
int IsFull(qt *q)
{
if(q→rear==MAXSIZE-1)
return 1;
else
return 0;
}
```

2. Explain circular linked list with example. How do you implement linked list operations in singly linked list? Explain.

Ans: A circular linked list is a list where the link field of last node points to the first node of the list. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node. The main advantage of circular linked list is that it requires minimum time to traverse the nodes which are already traversed, without moving to starting node.

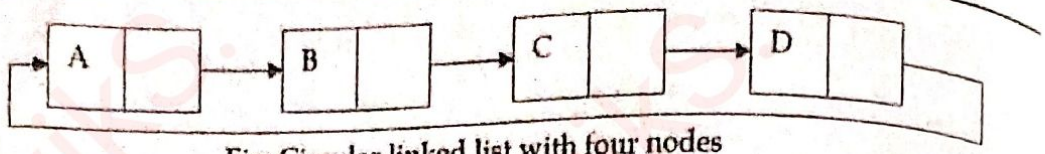


Fig: Circular linked list with four nodes

Singly linked list

A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer. The address of first node can be contained by an external pointer called 'first'.

Structure of a node of singly linked list

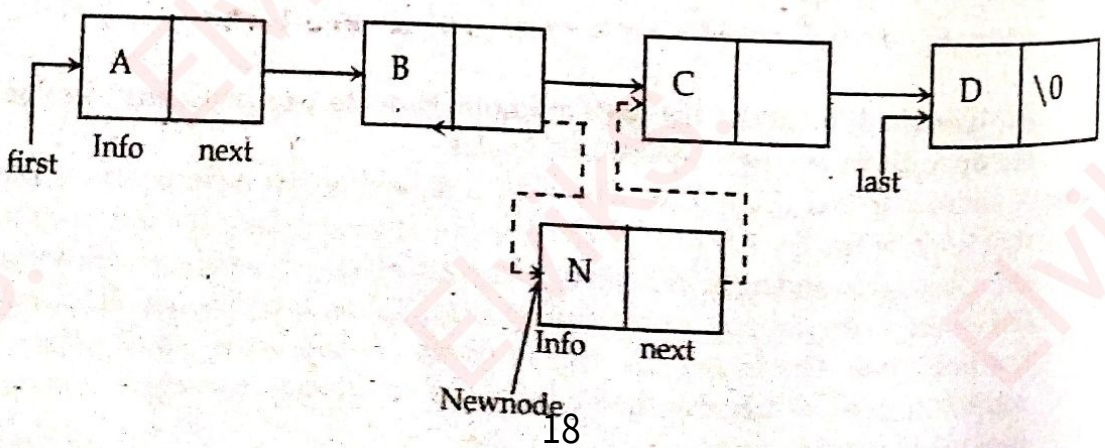
We can define a node as follows:

```
struct Node
{
int info;
struct Node *next;
};
typedef struct Node NodeType;
NodeType  *first;
NodeType  *last;
```

An algorithm to insert a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

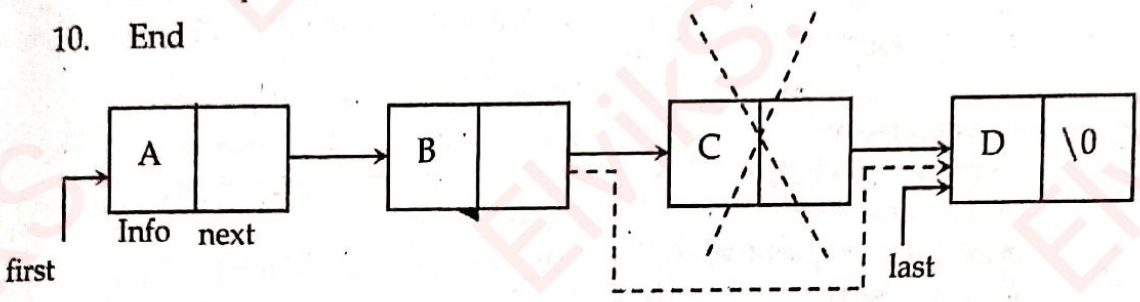
1. Start
2. Create a new node using malloc function as,
   Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Assign data to the info field of new node
   Newnode→info=el;
4. Enter position of a node at which you want to insert a new node. Let this position is 'pos'.
5. Set, temp=first;
6. if (first==null) then
   print "void insertion" and exit.
7. for(i=1; i<pos-1; i++)
   temp=temp→next;
8. Set, Newnode→next=temp→next;
9. Set, temp→next =Newnode.
10. End



An algorithm to delete a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Read position of a node which to be deleted let it be 'pos'.
3. if first==null then,
   Print "void deletion" and exit
   Otherwise,
4. Enter position of a node at which you want to delete a new node. Let this position is 'pos'.
5. Set, temp=first
6. for(i=1; i<pos-1; i++)
   Set, temp=temp→next;
7. Print deleted item is temp.next.info
8. Set, loc=temp→next;
9. Set, temp→next =loc→next;
10. End

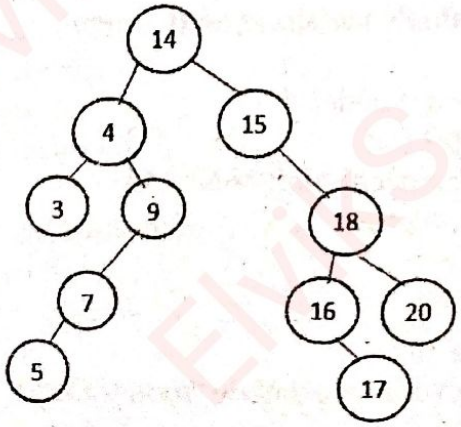


3. What is binary search tree? Write a program to implement insertion and deletion algorithms in binary tree.

Ans: A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:

- All keys in the left sub-tree of the root are smaller than the key in the root node
- All keys in the right sub-tree of the root are greater than the key in the root node
- The left and right sub-trees of the root are again binary search trees

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right sub-tree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.



```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int key;
 struct node *left, *right;
};
// Create a node
struct node *newNode(int item)
{
 struct node *temp = (struct node *)malloc(sizeof(struct node));
 temp->key = item;
 temp->left = temp->right = NULL;
 return temp;
}
// Inorder Traversal
void inorder(struct node *root)
{
 if (root != NULL)
 {
  // Traverse left
  inorder(root->left);
  // Traverse root
  printf("%d -> ", root->key);
  // Traverse right
  inorder(root->right);
 }
}
// Insert a node
struct node *insert(struct node *node, int key)
{
 // Return a new node if the tree is empty
 if (node == NULL) return newNode(key);
 // Traverse to the right place and insert the node
 if (key < node->key)
  node->left = insert(node->left, key);
 else
  node->right = insert(node->right, key);
 return node;
}
// Find the inorder successor
struct node *minValueNode(struct node *node)
{
 struct node *current = node;
 // Find the leftmost leaf
 while (current && current->left != NULL)
  current = current->left;
 return current;
}
// Deleting a node
struct node *deleteNode(struct node *root, int key)
```

```
{
 // Return if the tree is empty
 if (root == NULL) return root;
 // Find the node to be deleted
 if (key < root->key)
  root->left = deleteNode(root->left, key);
 else if (key > root->key)
  root->right = deleteNode(root->right, key);
 else
{
  // If the node is with only one child or no child
  if (root->left == NULL)
  {
   struct node *temp = root->right;
   free(root);
   return temp;
  }
  else if (root->right == NULL)
  {
   struct node *temp = root->left;
   free(root);
   return temp;
  }
  // If the node has two children
  struct node *temp = minValueNode(root->right);
  // Place the inorder successor in position of the node to be deleted
  root->key = temp->key;
  // Delete the inorder successor
  root->right = deleteNode(root->right, temp->key);
 }
 return root;
}
// Driver code
int main()
{
 struct node *root = NULL;
 root = insert(root, 8);
 root = insert(root, 3);
 root = insert(root, 1);
 root = insert(root, 6);
 root = insert(root, 7);
 root = insert(root, 10);
 root = insert(root, 14);
 root = insert(root, 4);
 printf("Inorder traversal: ");
 inorder(root);
 printf("\nAfter deleting 10\n");
 root = deleteNode(root, 10);
 printf("Inorder traversal: ");
 inorder(root);
}
```

Section B

Short questions

Attempt any Eight questions: (8 x 5 = 40)

4. How do you find complexity of algorithms? Explain.

Ans: Algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n. If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n. For this reason, complexity is calculated asymptotically as n approaches infinity. While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of space, which translates to the algorithm's memory requirements.

For Iterative Programs

For analyzing iterative algorithms we use RAM model for step counting. Consider the following programs that are written in simple English and does not correspond to any syntax.

```
A()
{
int i;
for (i=1 to n)
printf("Edward");
}
```

Since i equals 1 to n, so the above program will print Edward, n number of times. Thus, the complexity will be O(n).

For Recursive Program

For analyzing recursive algorithm we need to define recurrence relation and then solve such a recurrence relation by using any one of the solving method. Consider the following recursive programs.

Example:

```
A(n)
{
if (n>1)
   return (A(n-1))
}
```

Solution: Here we will see the simple Back Substitution method to solve the above problem.

$T(n) = 1 + T(n-1)$ ...Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

$T(n-1) = 1 + T(n-2)$ ...Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

$T(n-2) = 1 + T(n-3)$ ...Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

$T(n) = 1 + 1 + T(n-2) = 2 + T(n-2)$ ...Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

$T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = \ldots\ldots = k + T(n-k)$ ...Eqn. (5)

Now, according to Eqn. (1), i.e. $T(n) = 1 + T(n-1)$, the algorithm will run until $n>1$. Basically, n will start from a very large number, and it will decrease gradually. So, when $T(n) = 1$, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for $k = n-1$, the T(n) will become

Step5: Substitute k = n-1 in eqn. (5)

$T(n) = (n-1) + T(n-(n-1)) = (n-1) + T(1) = n-1+1$

Hence, $T(n) = n$ or $O(n)$.

5. **Evaluate the expression ABCD-*+ using stack where A=5, B=4, C=3 and D=7.**

Ans: ABCD-*+

Scanned character	Value	Op2	Op1	Result	vstack
A	5	.....	.....	.....	5
B	4	.....	.....	.....	5, 4
C	3	.....	.....	.....	5, 4, 3
D	7	.....	.....	.....	5, 4, 3, 7
-	.....	7	3	-4	5, 4, -4
*	.....	-4	4	-16	5, -16
+	.....	-16	5	-11	-11

Its final value is -11

6. **What is priority queue? Why do you need this type of queue?**

Ans: A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules. Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue

The best application of priority queue is observed in CPU scheduling. A short job is given higher priority over the longer one. Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

7. **Write a recursive program to find nth Fibonacci number.**

Ans:
```
#include<stdio.h>
#include<conio.h>
void main( )
{
int n,i;
int fibo(int);
printf("Enter value of n:");
scanf("%d", &n);
printf("nth Fibonacci term is:\n");
printf("%d", fibo(n));
getch( );
}
int fibo(int k)
{
if(k == 1 || k == 2)
return 1;
else
return fibo(k-1)+fibo(k-2);
}
```

8. Explain array implementation of list.

Ans: Array of linked list is an important data structure used in many applications. It is an interesting structure to form a useful data structure. It combines static and dynamic structure. Static means array and dynamic means linked list used to form a useful data structure. This array of linked list structure is appropriate for applications. Here, elements of list are stored in the (contiguous) cells of an array. The operations that are commonly performed in list are listed below:

- Creating of a list
- Inserting new element at required position of list
- Deletion of any element from list
- Modification of any element of list
- Traversing of a list
- Merging of two lists

Creating of a list

Creating of a list means at first declaring memory for a list and then inserts elements to given list.

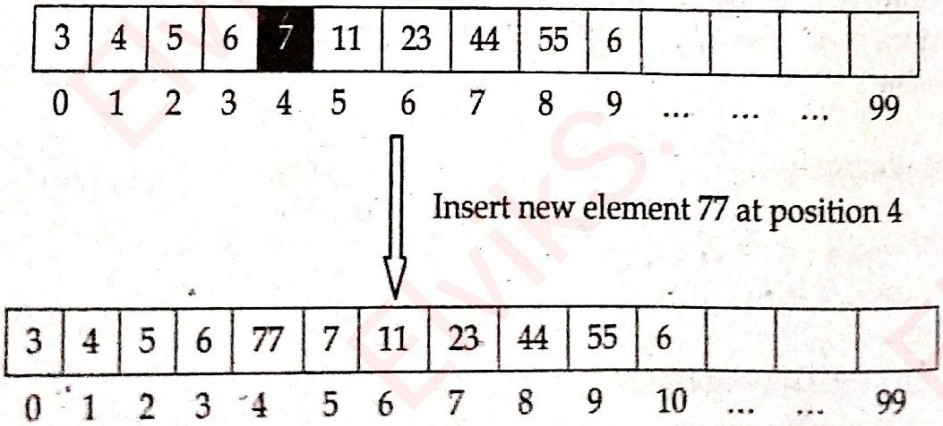
3	4	5	6	7	11	23	44	55	6
0	1	2	3	4	5	6	7	8	9

Example: Program for creation of a list and inserting data items to given list

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a[100], i, n;
printf("Enter no of elements to be inserted\n");
scanf("%d", &n);
printf("Enter any %d elements of given list", n);
for(i=0; i<n; i++)
{
scanf("%d", &a[i]);
}
getch( );
}
```

Inserting new element at required position

This operation is used to insert new element at required position from 0 to n^{th} index of given list. Where 0^{th} index is the first index of given list and n^{th} index is the $(last+1)^{th}$ index of given predefined list of size n. Here we need to shift every element one position right from last element to specified positions element.



Algorithm

1. Start
2. Read position of element to be inserted say it be 'pos'
3. Read element to be inserted say it be 'el'
4. Swap all elements one position right from last index to pos-1
5. Now location for new element is free so set new element at that location as,

a. list[pos]=el

6. Increment size of an array by one as,
   n=n+1;
7. Stop

9. Hand test selection sort with array of numbers 4, 71, 32, 19, 61, 2, -5 in descending order.

Ans:

Array position	0	1	2	3	4	5	6
Initial state	4	71	32	19	61	2	-5
Pass 1	-5	71	32	19	61	2	4
Pass 2	-5	2	32	19	61	71	4
Pass 3	-5	2	4	19	61	71	32
Pass 4	-5	2	4	19	61	71	32
Pass 5	-5	2	4	19	32	71	61
Pass 6	-5	2	4	19	32	61	71
Pass 7	-5	2	4	19	32	61	71

10. Write a program to implement sequential search algorithm.

Ans: Linear search algorithm finds given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Algorithm

1. Start
2. Read the search element from the user
3. Compare, the search element with the first element in the list.
4. If both are matching, then display "Given element found!" and terminate the function
5. If both are not matching, then compare search element with the next element in the list.
6. Repeat steps 4 and 5 until the search element is compared with the last element in the list.
7. If the last element in the list is also doesn't match, then display "Element not found!" and terminate the function
8. Stop

Complete C program for Sequential search

```
#include <stdio.h>
int main()
{
    int a[100], key, i, n;
    printf("Enter the number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d numbers \n", n);
    for (i = 0; i < n; i++)
    {
    scanf("%d", &a[i]);
    }
    printf("Enter the number to search\n");
    scanf("%d", &key);
    for (i = 0; i < n; i++)
    {
    if (a[i] == key)

        {
    printf("%d is present at location %d.\n", key, i+1);
    break;
    }
    }
    if (i == n)
    printf("%d is not present in array.\n", key);
    return 0;
}
```

11. What is graph traversal? Explain.

Ans: Traversing a graph means visiting all the vertices in a graph exactly one. In tree traversal, there is a special node root, from which the traversal starts. But in graph all nodes are treated equally. So the starting node in graph can be chosen arbitrarily. In general, for diagrammatic illustration, the left-up node is taken as starting vertex for traversal. Two common approaches for traversal are:

- Breadth First Search Traversal (BFS)
- Depth First Search Traversal (DFS)

Breadth First Search Traversal (BFS)

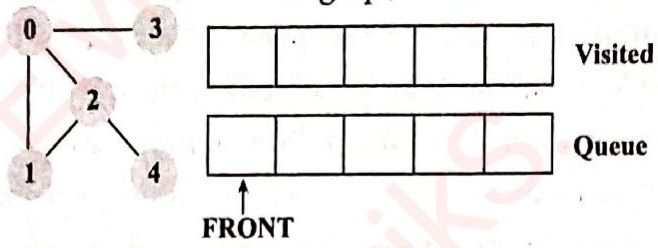
The traversal starts at a node v, after marking the node the traversal visits all incident edges to node v after marking the nodes and then moving to an adjacent node and repeating the process. The traversal continues until all unmarked nodes in the graph have been visited. A queue is maintained in the technique to maintain the list of incident edges and marked nodes.

Algorithm

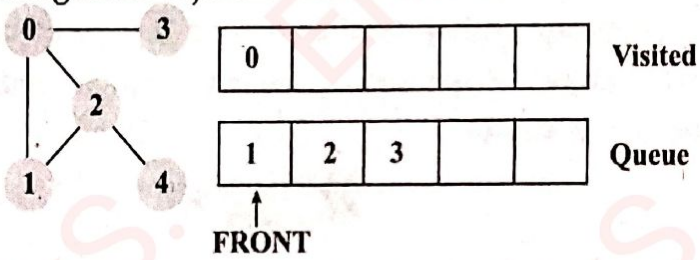
```
BFS (G, s)
//s is start vertex
{
T = {s};
L =Φ;     //an empty queue
Enqueue (L, s);
```

```
while (L != Φ )
{
v = Dequeue (L);
For each neighbor w to v
if ( w∉ L && w∉ T )
{
Enqueue (L, w);
T = T U {w};
//put edge {v, w} also
}
}
}
```

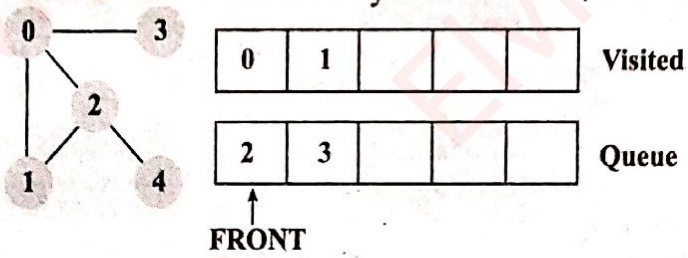
BFS example: Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



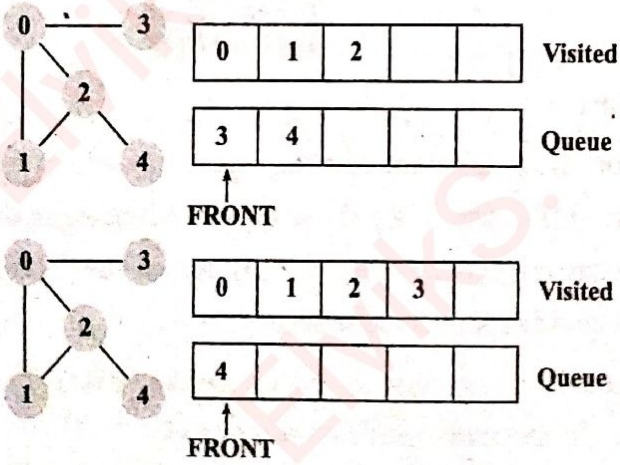
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

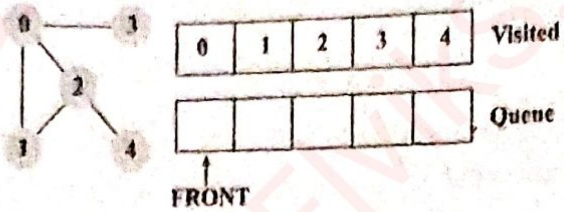


Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





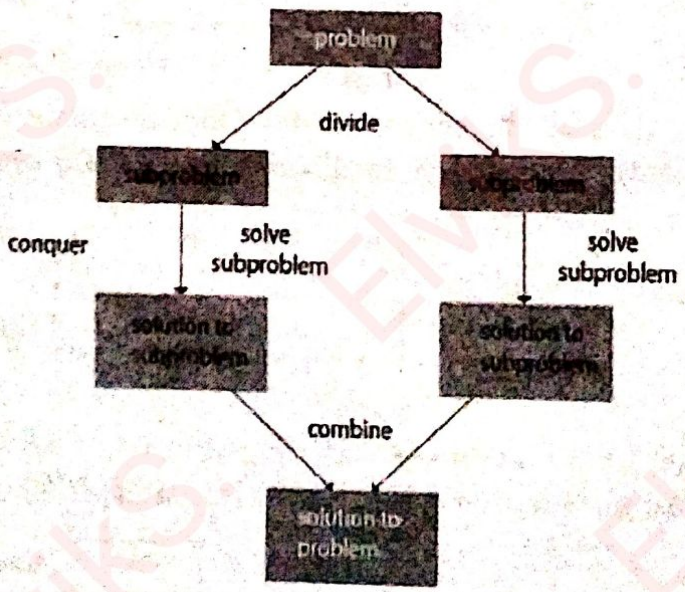
12. **Write short note on:**

a) **Divide and Conquer sorting**

Ans: An important problem-solving technique that makes use of recursion is divide and conquer. A divide-and-conquer algorithm is an efficient recursive algorithm that consists of two parts:

- **Divide,** in which smaller problems are solved recursively (except, of course, base cases)
- **Conquer,** in which the solution to the original problem is then formed from the solutions to the sub-problems

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call are not. Consequently, the recursive routines presented so far in this section are not divide-and-conquer algorithms. Also, the sub-problems usually must be disjoint (i.e., essentially no overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers.



Merge Sort

Merge sort is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

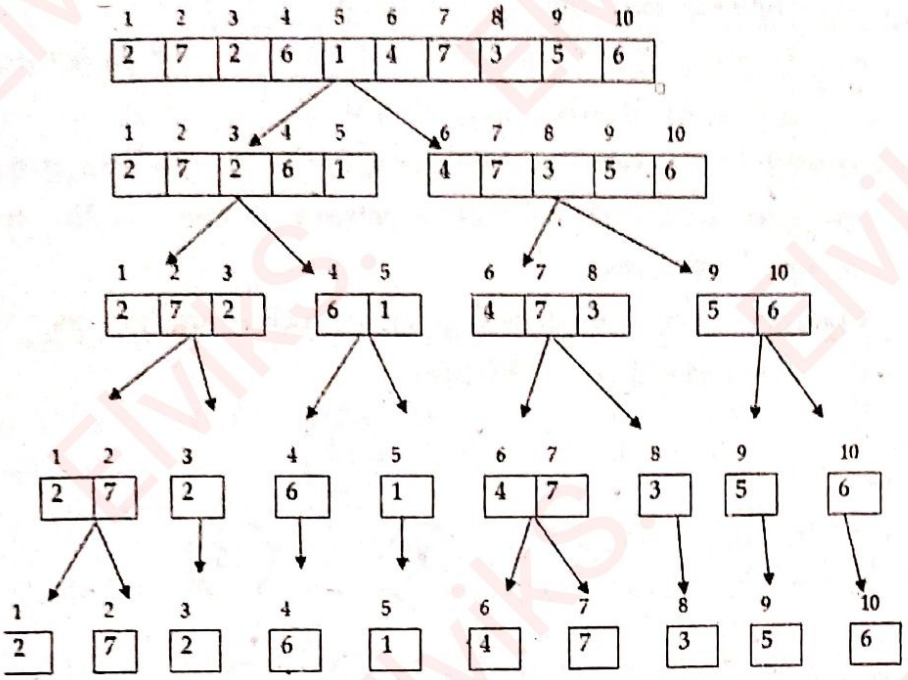
The process of merge sort can be formalized into three basic operations.

1. Divide the array into two sub arrays
2. Recursively sort the two sub arrays

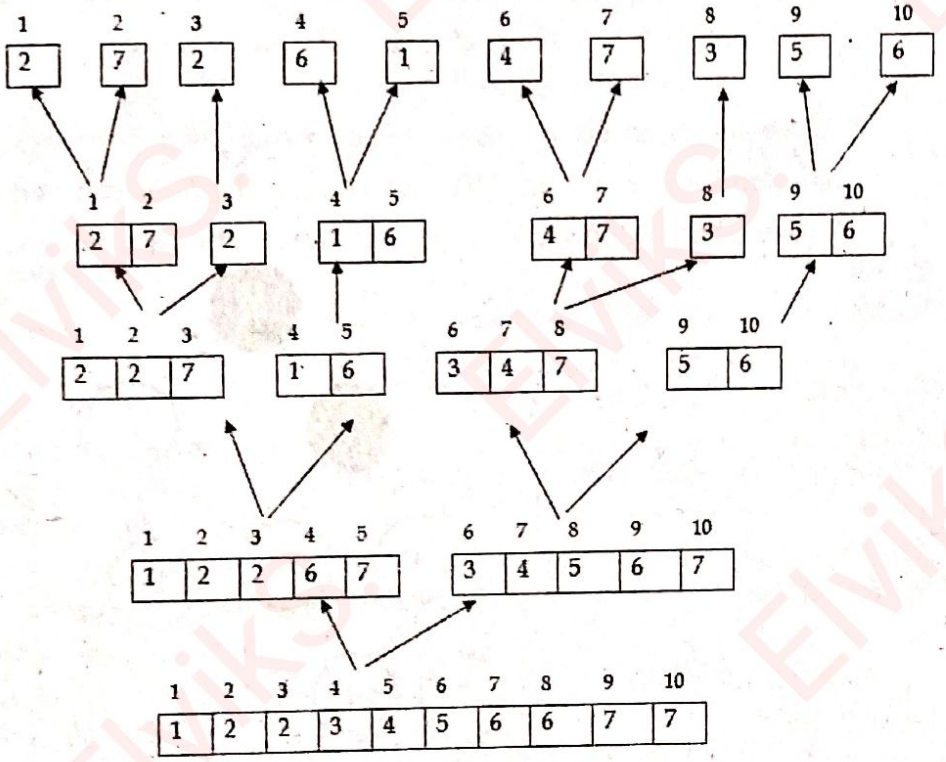
3. Merge the newly sorted sub arrays

Tracing: A[ ] = {4, 7, 2, 6, 1, 4, 7, 3, 5, 6}

Pass 1: Dividing



Pass 2: Merging



b) AVL tree

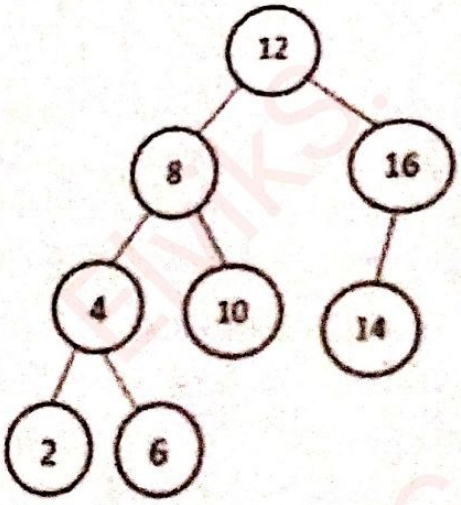
Ans: AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of AVL tree.

Example: The tree shown in Figure below satisfies the AVL balance condition and is thus an AVL tree.



The tree shown in Figure below, which results from inserting 1, using the usual algorithm, is not an AVL tree because the darkened nodes have left sub trees whose heights are 2 larger than their right sub trees.