SYLLABUS
DATA STRUCTURES USING C (BCA-108)

Internal	:	25 Marks
External (End Term)	:	75 Marks
Total	:	100 Marks
Duration of Exam.	:	3 Hrs.

UNIT-I

INTRODUCTION TO DATA STRUCTURES: Basic Terminology, Elementary Data Organizations, Classification of data structures and its operations. Arrays: Representation of single and multidimensional arrays (up to three dimensions) ; sparse arrays - lower and upper triangular matrices and Tri-diagonal matrices; addition and subtraction of two sparse arrays. (Multidimensional, and, sparse arrays, to be given elementary treatment.) Stacks and Queues: Introduction and primitive operations on stack; Stack application: Polish Notations; Evaluation of postfix expression; Conversion from infix to postfix; Introduction and primitive operations on queues; D-queues and priority queues.[T1,T2,T3]

[No. of Hrs: 11]

UNIT-II

LISTS: Introduction to linked lists; Sequential and linked lists, operations such as traversal, insertion, deletion, searching, Two way lists and Use of headers Trees: Introduction and terminology; Traversal of binary trees; Recursive algorithms for tree operations such as traversal, insertion and deletion; [T1, T2, T3]

[No. of Hrs: 11]

UNIT-III

Introduction to and creation of AVL trees and m-way search trees - (elementary treatment to be given); Multilevel indexing and B-Trees: Introduction; Indexing with binary search trees; Multilevel indexing, a better approach to tree indexes; Example for creating a B-tree. [T1, T2, T3]

[No. of Hrs: 11]

UNIT-IV

SORTING TECHNIQUES: Insertion sort, selection sort and merge sort. Searching Techniques: linear search, binary search and hashing. (Complexities NOT to be discussed for sorting and searching) [T1, T2, T3]

[No. of Hrs: 11]

END TERM EXAMINATION
SECOND SEMESTER (BCA)–MAY-JUNE 2014
DATA STRUCTURE USING 'C'–[BCA-108]

Time : 3 hrs. **M.M. : 75**

Note: *Q.No. 1 is compulsory and is of 25 marks. Attempt one question from each unit. Each question carries 12.5 marks.*

Q.1.(a) What do you mean by data structure? Explain the difference between linear and non-linear data structures. **(5×5=25)**

Sol. A data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

Difference between Linear and Nonlinear Data Structures

Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this nonlinear structure, they might be difficult to be implemented in computer's linear memory compared to implementing linear data structures. Selecting one data structure type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.

Q.1.(b) Explain the sparse matrix & its representation.

Sol. A matrix in which number of zero entries are much higher than the number of non zero entries is called sparse matrix. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices. One may save space by storing for only non-zero entries. Examples of sparse matrices includes triangular matrices, tri-diagonal matrices.

Representation of sparse Matrices

We can represent a sparse matrix by using three tuple method of storage.

(1) Row-major 3-tuple method.

(2) Column-major 3-tuple method.

e.g.:

$$M[4][5] = \begin{bmatrix} 0 & 8 & 6 & 0 & 2 \\ 9 & 0 & 0 & 3 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 7 \end{bmatrix}$$

Scanned by CamScanner

Row-major 3-tuple method

0	1	2
Row (4)	**Column (5)**	**No. of non zero elements (8)**
0	1	8
0	2	6
0	4	2
1	0	9
1	3	3
2	1	3
3	0	6
3	4	7

Note: Matrix is stored where no. of column are fixed 3.

Column-major 3-tuple methods

Column (5)	Row (4)	No. of non-zero elements (8)
0	1	9
0	3	6
1	0	8
1	2	3
2	0	6
3	1	3
4	0	2
4	3	7

Note: No. of column fixed which is 3 and the matrix is stored column wise.

Q.1.(*c*) Prove that in a complete binary tree of height h, total no. of modes are $2^{h+1} - 1$.

Sol. A Complete Binary Tree of height h has at least one and as many as 2^h nodes at level h (the final level is not empty, and may be full). In any event, level h - 1, (h >0), must be full by definition of a Complete Binary Tree. The number of nodes in a Full Binary Tree of height h - 1 is $2^{(h-1)+1} - 1$. A Complete Binary Tree of height h has at least one more node than a Full Binary Tree of height h -1.

Therefore, the number of nodes n in a Complete Binary Tree is bounded by

$$2^{(h-1)+1} - 1 + 1 <= n <= 2^{h+1} - 1$$

Therefore,

$$2^h <= n < 2^{h+1}$$

Equivalently, we can say $n = 2^{h+1} - 1$.

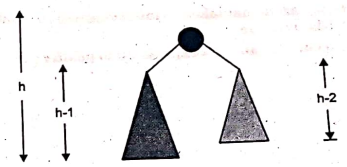
Q.1.(*d*) What is an AVL tree? Why is it useful is searching?

Sol. An **AVL tree** is another balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(logn)*** search time. Addition and deletion operations also take ***O(logn)*** time.

Definition of an AVL tree

An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height

One of the advantage of AVL tree is better search times for keys. The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Log n). So time complexity of AVL delete is O(Log n).

Q.1.(*e*) Explain binary search with suitable example.

Ans. Example of Binary Search: Let LIST be the following sorted 13-element array:

LIST: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to LIST for different values of ITEM(value to be searched).

Suppose ITEM = 40. The search for ITEM in the array LIST is pictured in where the values of LIST[LB] and LIST[UB] in each stage of the algorithm are indicated by **parenthesis** and the value of LIST[MID] by a **bold**. Specifically, LB(lower bound), UB(upper bound) and MID will have the following successive values:

Step 1: Initially, LB = 0 and UB = 12. Hence,
MID = INT[(0 + 12) / 2] = 6 and so LIST[MID] = LIST[6]=55
Since 40 < 55, UB = MID – 1 = 6 – 1 = 5.

Step 2: Now in this step, LB=0 and UB=5 (from previous step)
MID = INT[(0 + 5) / 2] = 2 and so LIST[MID] = LIST[2] = 30
Since 40 > 30, LB = MID + 1 = 2 + 1 = 3.

Step 3: Now for this step, LB=3(from previous step) and UB=5
MID = INT[(3 + 5) / 2] = 4 and so LIST[MID] = LIST[4] = 40
The search is successful and LOC = MID + 1 = 4 + 1 = 5.

Processing of each step:

(1) (11), 22, 30, 33, 40, 44, **55**, 60, 66, 77, 80, 88, (99)
(2) (11), 22, **30**, 33, 40, (44), 55, 60, 66, 77, 80, 88, 99

Scanned by CamScanner

(3) 11, 22. 30, (33), 40, (44), 55, 60, 66, 77, 80, 88, 99 [Successful]

Note: Parenthesis shows positions of LB and UB for each step and element in bold is the element at middle position.

UNIT-I

Q.2.(a) Wrie a c program that takes an input expression in infix notation and convert it into postfix notation. (8)

Sol. Program to convert an infix expression to postfix expression

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
struct infix
{
    char target[50] ;
    char stack[50] ;
    char *ss, *ts ;
    int top ;
};
void initinfix ( struct infix * ) ;
void setexpr ( struct infix *, char * ) ;
void push ( struct infix *, char ) ;
char pop ( struct infix * ) ;
void convert ( struct infix * ) ;
int priority ( char ) ;
void show ( struct infix ) ;
/* initializes structure members */
void initinfix ( struct infix *p )
{
    p -> top = -1 ;
    strcpy (  p -> target, "" ) ;
    strcpy ( p -> stack, "" ) ;
    p -> ts = p -> target ;
    p -> ss = "" ;
}
/* sets s to point to given expr. */
void setexpr ( struct infix *p, char *str )
{
    p -> ss = str ;
}
/* adds an operator to the stack */
```



```
void push ( struct infix *p, char c )
{
    if ( p -> top == MAX )
        printf ( "\nStack is full.\n" ) ;
    else
    {
        p -> top++ ;
        p -> stack[p -> top] = c ;
    }
}
/* pops an operator from the stack */
char pop ( struct infix *p )
{
if ( p -> top == -1 )
{
    printf ( "\nStack is empty.\n" ) ;
    return -1 ;
}
else
{
    char item = p -> stack[p -> top] ;
    p -> top— ;
    return item ;
}
}
/* converts the given expr. from infix to postfix form */
void convert ( struct infix *p )
{
char opr ;
while ( *( p -> ss ) )
{
if ( *( p -> ss ) == ' ' || *( p -> ss ) == '\t' )
{
    p -> ss++ ;
    continue ;
}
while ( isdigit ( *( p -> ss ) ) || isalpha ( *( p -> ss ) ) )
{
    *( p -> ts ) = *( p -> ss ) ;
    p -> ss++ ;
```

Scanned by CamScanner

```
        p -> ts++ ;
    }
    if ( *( p -> ss ) == '(' )
    {
        push ( p, *( p -> ss ) ) ;
        p -> ss++ ;
    }
    if ( *( p -> ss ) == '*' || *( p -> ss ) == '+' || *( p -> ss ) == '/' || *( p -> ss ) == '%' ||
p -> ss ) == '-' ||
        *( p -> ss ) == '$' )
    {
        if ( p -> top != -1 )
        {
            opr = pop ( p ) ;
            while ( priority ( opr ) >= priority ( *( p -> ss ) ) )
            {
                *( p -> ts ) = opr ;
                p -> ts++ ;
                opr = pop ( p ) ;
            }
        push ( p, opr ) ;
        push ( p, *( p -> ss ) ) ;
        }
        else
            push ( p, *( p -> ss ) ) ;
    p -> ss++ ;
    }
    if ( *( p -> ss ) == ')' )
    {
        opr = pop ( p ) ;
        while ( ( opr ) != '(' )
        {
            *( p -> ts ) = opr ;
            p -> ts++ ;
            opr = pop ( p ) ;
        }
    p -> ss++ ;
    }
}
while ( p -> top != -1 )
{
    opr = pop ( p ) ;
```



```
    *( p -> ts ) = opr ;
    p -> ts++ ;
}
*( p -> ts ) = '\0' ;
}
/* returns the priority of an operator */
int priority ( char c )
{
if ( c == '$' )
    return 3 ;
if ( c == '*' || c == '/' || c == '%' )
    return 2 ;
else if ( c == '+' || c == '-' )
        return 1 ;
else
        return 0 ;
}
/* displays the postfix form of given expr. */
void show ( struct infix p )
{
printf ( " %s", p.target ) ;
}
void main( )
{
struct infix p ;
char expression[50] ;
initinfix ( &p ) ;
clrscr( ) ;
printf ( "\nEnter an expression in infix form: " ) ;
gets (expression ) ;
setexpr ( &p, expression ) ;
convert ( &p ) ;
printf ( "\nThe postfix expression is: " ) ;
show ( p ) ;
getch( ) ;
}
```

Q.2.(*b*) Give an algorithm for insertion of an element in circular queue. **(4.5)**

Sol. Algorithm to insertion of an element in circular queue

Insert CircularQ ( ): Description: Here **QUEUE** is an array with **N** locations. **FRONT** and **REAR** points to the front and rear elements of the **QUEUE. ITEM** is the

Scanned by CamScanner

value to be inserted.

1. If (FRONT == 0 and REAR == N-1) or (FRONT == REAR + 1) Then
2. Print: Overflow
3. Else
4. If (REAR == – 1) Then
   [Check if QUEUE is empty]
   (*a*) Set FRONT = 0
   (*b*) Set REAR = 0
5. Else If (REAR == N – 1) Then
   [If REAR reaches end of QUEUE]
6. Set REAR = 0
7. Else
8. Set REAR = REAR + 1
   [Increment REAR by 1]
   [End of Step 4 If]
9. Set QUEUE[REAR] = ITEM
10. Print: ITEM inserted
    [End of Step 1 If]
11. Exit

Q.3.(*a*) Write a c function that reverse a string and finds its length. (6)

Sol.

```
void rev_len(char a[100])
 {
   char b[100];
   int i,j,count=0;
   for(i=0;i<=100;i++)
   {
   if(a[i]=='\0')
     break;
   count++;
 }
 printf("Length of String %d", count);
 for(i=count-1,j=0; i>=0; i—,j++)
 {
   b[j]=a[i];
 }
b[j]='\0';
printf("Reverse of a String is %s", b);
}
```


Aliter: (Using Pointers)

```
void reverse(char *string)
{
  int length, c;
  char *begin, *end, temp;
  length = string_length(string);
  begin = string;
  end = string;
  for (c = 0; c < length - 1; c++)
    end++;
  for (c = 0; c < length/2; c++)
  {
    temp = *end;
    *end = *begin;
    *begin = temp;
    begin++;
    end– –;
  }
}
int string_length(char *pointer)
{
  int c = 0;
  while( *(pointer + c) != '\0' )
    c++;
return c;
}
```

Q.3.(*b*) Implement an stack using linked list. (6.5)

Ans. Implementation of Stack using Linked List

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
struct node
  {
    int data;
    struct node *link;
 }*top=NULL;
 void push(struct node **q, int num)
 {
   struct node **temp;
```

Scanned by CamScanner

```
        if(q==NULL)                /* if list is empty */
        {
          top=(struct node *) malloc(sizeof(struct node));
          top->data=num;
          top->link=NULL;
          *q=top;
        }
        else
        {
          temp=*q;
          while(temp->link!=NULL)
            temp=temp->link;
          /* add node at the end */
          top=malloc(sizeof(struct node));
          top->data=num;
          top->link=NULL;
          temp->link=top;
        }
        }
        void pop(struct node **q)
        {
        struct node *temp=*q;
          while(temp->link!=top)
            temp=temp->link;
      printf("%d is deleted\n", top->data);
      temp->link=NULL;
     top=temp;
     }
    void traverse(struct node **q)
    {
      struct node *temp=*q;
      printf("Elements of list is\n");
      while(temp!=NULL)
      {
        printf("%d\t",temp->data);
        temp=temp->link;
  }
}
void peek(struct node **q)
{
```



```
  printf("%d is the top element of the list\n", top->data);
}
void main()
{
int ch;
struct node *q;
int i, n, element;
q=NULL;
do
{
clrscr();
printf("\n\n1.PUSH\n2.POP\n3.PEEP\n4.TRAVERSE\n5.EXIT");
printf("\nEnter your choice:");
scanf("%d", &ch);
switch(ch)
{
case 1:    printf("Enter the no. of Elements-:");
           scanf("%d",&n);
           printf("Enter the Element-:");
           for(i=0;i<n;i++)
           {
           scanf("%d",&element);
           push(&q,element);
           }
           getch();
           break;
case 2:    pop(&q);
           getch();
           break;
case 3:    peek(&q);
           getch();
           break;
case 4:    traverse(&q);
           getch();
           break;
case 5:    exit(0);
           break;
default:
           printf("\n\nInvalid choice. Please try again...\n")
}
```

Scanned by CamScanner

```
getch();
}
```

UNIT-II

Q.4.(a) Write a c function that takes a linear linked list as its input and display it in the reverse order. (6)

Ans. *Function to reverse a linear linked list */

```
void reverse(struct node **q)
  {
   struct node *s;
   struct node *p;
   struct node *t;
   t=*q;
   s=NULL;
   while(t!=NULL)
   {
    p=s;
    s=t;
    t=t->link;
    s-> nk=p;
   }
  *q=s;
  }
```

Q.4.(b) Write a c function that cancatenates two linear linked list. (6.5)

Sol. * Function to concatenate two linear linked list */

```
void concat ( struct node **s, struct node **t )
{
struct node *temp ;
if ( *s == NULL )
*s = *t ;
else
{
   if ( *t != NULL )
   {
      temp = *s ;
      while ( temp -> link != NULL )
          temp = temp -> link ;
   temp -> link = *t ;
   }
}
}
```


Q.5. Define binary search tree. Write algorithm for pre-order, post-order & in-order transversal of a binary tree. (12.5)

Sol. A Binary Search Tree is a binary tree that is either empty or a node containing a key value, left child and right child. By analysing this definition, we note that BST comes in two variants namely empty BST and non-empty BST.

The empty BST has no further structure, while the non-empty BST has three components.

The non-empty BST satisfies the following conditions:

(*a*) The key in the left child of a node (if exists) is less than the key in its parent node.

(*b*) The key in the right child of a node (if exists) is greater than the key in its parent node.

(*c*) The left and right subtrees of the root are again binary search trees.

Binary Tree allows three types of traversals through its nodes. They are as follows:

1. Pre Order Traversal
2. In Order Traversal
3. Post Order Traversal

All the above traversals can be performed in recursive or iterative manner.

1. Preorder traversal: Each node is visited before its children are visited; the root is visited first.

Recursive Algorithm for pre order traversal:

1. visit root node
2. traverse left sub-tree in preorder
3. traverse right sub-tree in preorder

Iterative PreOrder Traversal(Using stack):

1. Start with node equal to root node
2. Store the node in the stack, print it and visit it's left child.
3. Repeat step 2 while node is not NULL, if it's NULL then pop it's parent (i.e. the last node in stack).
4. Now move to node's right child and repeat step 2
5. Repeat whole procedure while node is not NULL and stack is not empty

2. Postorder traversal: The children of a node are visited before the node itself; the root is visited last. Every node is visited after its descendents are visited.

Recursive Algorithm for postorder traversal:

1. traverse left sub-tree in post order
2. traverse right sub-tree in post order
3. visit root node.

Iterative PostOrder Traversal(Using stack):

1.1 Create an empty stack

2.1 Do following while root is not NULL

(*a*) Push root's right child and then root to stack.

(*b*) Set root as root's left child.

Scanned by CamScanner

2.2 Pop an item from stack and set it as root.

(*a*) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.

(*b*) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

3. Inorder traversal: The left sub tree is visited, then the node and then right sub tree.

Recursive Algorithm for inorder traversal:

1. traverse left sub-tree
2. visit node
3. traverse right sub-tree

Iterative InOrder Traversal(Using stack):

(1) Create an empty stack S.

(2) Initialize current node as root

(3) Push the current node to S and set current = current->left until current is NULL

(4) If current is NULL and stack is not empty then

(*a*) Pop the top item from stack.

(*b*) Print the popped item, set current = popped_item->right

(*c*) Go to step 3.

(5) If current is NULL and stack is empty then we are done.

UNIT-III

Q.6.(*a*) Define the time complexity & search a mode in a binary search tree. Whether your time complexity is ratio for left or light skew & binary search tree? Justify. (6)

Sol. **Algorithm complexity** is a **measure** which evaluates the order of the **count of operations**, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough **approximation of the number of steps** necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.

Algorithm complexity is commonly represented with the **O(*f*) notation**, also known as **asymptotic notation** or **"Big O notation"**, where ***f*** is the function of the size of the input data. The asymptotic computational complexity **O(*f*)** measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.

Complexity can be **constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential**, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of **"algorithms complexity"** or just **"complexity"** we use the term **"running time"**.

Algorithm complexity is a rough approximation of the number of steps, which will be executed depending on the size of the input data. Complexity gives the order of steps count, not their exact count.

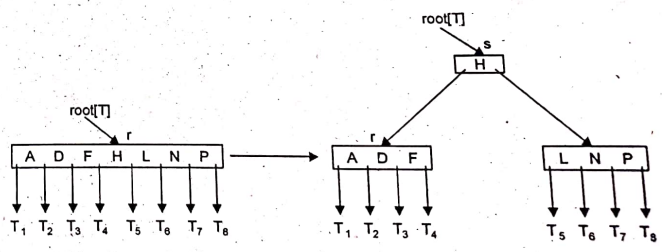
Searching a key: To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
/* C function to search a given key in a given BST */
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);
}
```

The worst case time complexity of search operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Q.6.(*b*) Explain the insertion of a mode in B-tree. **(6.5)**

Sol. Inserting a key k into a B-tree T of height h is done in a single pass down the tree, requiring $O(h)$ disk accesses. The CPU time required is $O(th) = O(t \log_t n)$. The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.



Splitting the root with $t = 4$. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

The number of keys accesses performed by B-TREE-INSERT is $O(h)$ for a B-tree of height h, since only $O(1)$ KEY-READ and KEY-WRITE operations are performed between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t \log_t n)$.

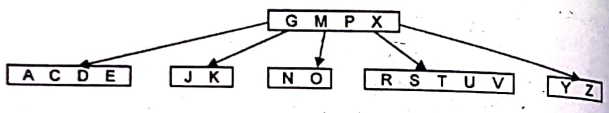
Scanned by CamScanner

Since B-TREE-INSERT-NONFULL is tail-recursive, it can be alternatively implemented as a **while** loop, demonstrating that the number of pages that need to be in main memory at any time is $O(1)$.

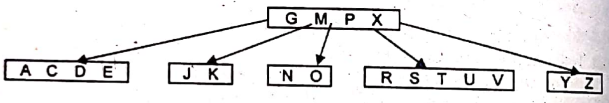
Explanation of the below figure:

Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. (a) The initial tree for this example. (b) The result of inserting B into the initial tree; this is a simple insertion into a leaf node. (c) The result of inserting Q into the previous tree. The node RSTUV is split into two nodes containing RS and UV, the key M is moved up to the root, and Q is inserted in the leftmost of the two halves (the RS node). (d) The result of inserting L into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. (e) The result of inserting F into the previous tree. The node ABCDE is split before F is inserted into the rightmost of the two halves (the DE node).

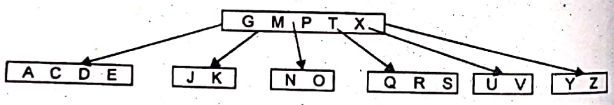
(*a*) Initial tree



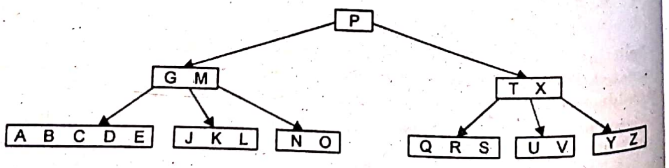
(*b*) B inserted



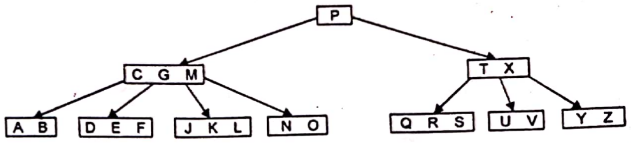
(*c*) Q inserted



(*d*) L inserted



(*e*) F inserted



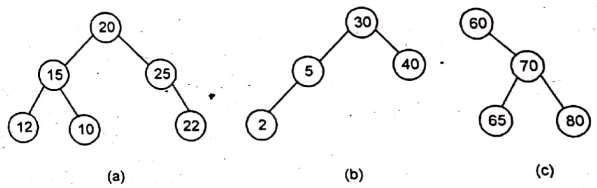
Q.7. What do you mean by binary search tree based indexing? How is it different from multilevel indexing? **(12.5)**

Sol. Binary Search Tree

Definition: A binary tree that may be empty. A nonempty binary search tree satisfies the following properties:

1. Each node has a key (or value), and no two nodes have the same key (*i.e.*, all keys are distinct).
2. For every node x, all keys in the left subtree of x are smaller than that in x.
3. For every node x, all keys in the right subtree of x are larger than that in x.
4. The left and right subtrees of the root are also binary search trees

Examples of Binary Search Trees:



Binary Search Trees

- Which of the above trees are binary search trees?

— (b) and (c)

- Why isn't (a) a binary search tree?

— It violates the property #3

Indexed Binary Search Trees

Definition

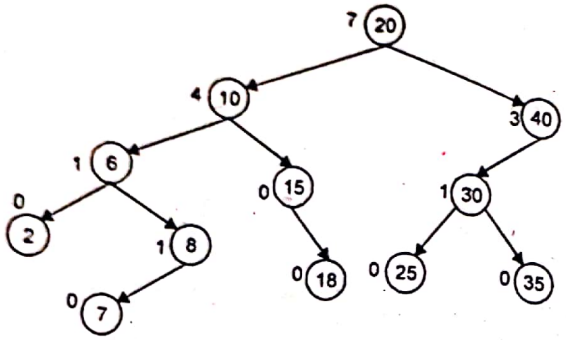
- Binary search tree.
- Each node has an additional field 'LeftSize'.
- Support search and delete operations by rank as well as all the binary search tree operations.
- Left Size

– the number of elements in its left subtree

– the rank of an element with respect to the elements in its subtree (e.g., the fourth element in sorted order)

Scanned by CamScanner

Indexed Binary Search Tree Example



- What is the Leftsize for each node?
- LeftSize values are in red.

LeftSize and Rank

- Rank of an element is its position in inorder (inorder = ascending key order) [2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2)=0
- rank(15)=5
- rank(20)=7
- LeftSize(x) = rank(x)

with respect to elements in the subtree rooted at x.

Multi-Level Indexes: Indexing schemes so far have looked at an ordered index file.

- Binary search performed on this index to locate pointer to disk block or record.
- Requires approximately $\log_2 n$ accesses for index with n blocks.
- Base 2 chosen because binary search reduces part of index to search by factor of 2
- The idea behind multi-level indexes is to reduce the part of the index to search by bfr , the blocking factor of the index (which is bigger than 2!
- The value of bfr for the index is called the fan-out. We will refer to it using the symbol of.
- Searching a multi-level index requires approximately $\log_{fo} n$ block accesses
- This is smaller than binary search for fo > 2
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk blocks.
- Such a multi-level index is a form of search tree; however, insertion and deletion of new index entries is a severe problem because every level of the index is an ordered file
- Hence most multi-level indexes use B-tree tree data structures, which leave space in each tree node disk block) to allow for new index entries.

UNIT-IV

Q.8. Explain selection sort and merge sort with suitable example. Show various stages. **(12.5)**

Sol. Performing selection sort on following list:

77, 33, 44, 11, 88, 22, 66, 55

Pass 0 : 77, 33, 44, 11, 88, 22, 66, 55

Step I:

0–1	77	33	44	11	88	22	66	55
0–2	33	77	44	11	88	22	66	55
0–3	33	77	44	11	88	22	66	55
0–4	11	77	44	33	88	22	66	55
0–5	11	77	44	33	88	22	66	55
0–6	11	77	44	33	88	22	66	55
0–7	11	77	44	33	88	22	66	55
	11	77	44	33	88	22	66	55

Pass 1:

Step II:

1–2	11	77	44	33	88	22	66	55
1–3	11	44	77	33	88	22	66	55
1–4	11	33	77	44	88	22	66	55
1–5	11	33	77	44	88	22	66	55
1–6	11	22	77	44	88	33	66	55
1–7	11	22	77	44	88	33	66	55
	11	22	77	44	88	33	66	55

Scanned by CamScanner

Pass 2:

Step III:

2–3	11	22	77	44	88	33	66	55
2–4	11	22	44	77	88	33	66	55
2–5	11	22	44	77	88	33	66	55
2–6	11	22	33	77	88	44	66	55
2–7	11	22	33	77	88	44	66	55
	11	22	33	77	88	44	66	55

Pass 3:

Step IV:

3–4	11	22	33	77	88	44	66	55
3–5	11	22	33	77	88	44	66	55
3–6	11	22	33	44	88	77	66	55
3–7	11	22	33	44	88	77	66	55
	11	22	33	44	88	77	66	55

Pass 4:

Step V:

4–5	11	22	33	44	88	77	66	55
4–6	11	22	33	44	77	88	66	55
4–7	11	22	33	44	66	88	77	55
	11	22	33	44	55	88	77	66

Pass 5:
Step VI:

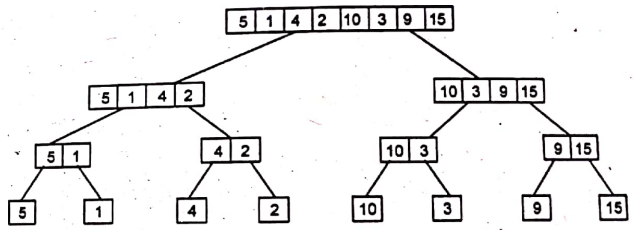
5–6	11	22	33	44	55	88	77	66
5–7	11	22	33	44	55	77	88	66
	11	22	33	44	55	66	88	77

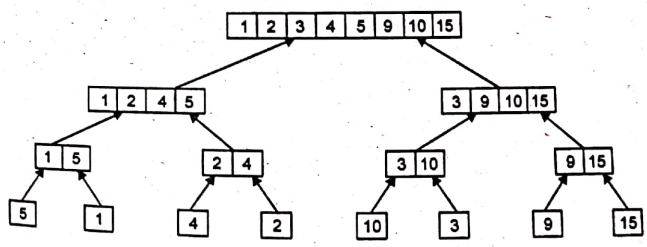
Pass 6:
Step VII:

6–7	11	22	33	44	55	66	88	77
	11	22	33	44	55	66	77	88

(Sorted List)

Example of Merge Sort:





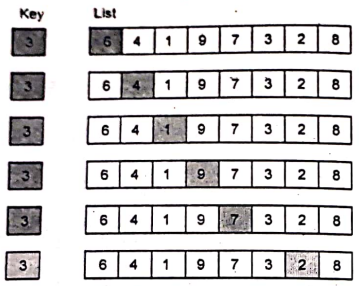
Q.9.(a) How linear search is ued to find an element? **(6)**

Sol. Suppose that **A** is an array having **N** elements. **ITEM** is the value to be searched then case 1:-unsorted list: The ITEM is compared with each and every element of the

Scanned by CamScanner

- if the element is found then no further checking/comparison is required
- if all the element are compared/checked then ITEM is not found

Example



Case 2:-sorted list

(a) the ITEM is less than the first element and greater than last element of the list then searching is performed by comparing each element in the list with item [provided list is in ascending order] otherwise

(b) ITEM is not present in the list.

Q.9.(b) Explain the concept of collision in hashing and its remedies. (6.5)

Ans. 9 (b) : Refer Q. No.8(a) of End Term Exam 2015 for hashing and hash function.

Collision resolution: If two keys hash to the same index, the corresponding record cannot be stored in the same location. So, if it's already occupied, we must find another location to store the new record, and do it so that we can find it when we look it up later on.

To give an idea of the importance of a good collision resolution strategy, consider the following result, derived using the birthday paradox. Even if we assume that our hash function outputs random indices uniformly distributed over the array, and even for an array with 1 million entries, there is a 95% chance of at least one collision occurring before it contains 2500 records.

There are a number of collision resolution techniques, but the most popular are chaining and open addressing.

Chaining Hash collision resolved by chaining.

In the simplest chained hash table technique, each slot in the array references a linked list of inserted records that collide to the same slot. Insertion requires finding the correct slot, and appending to either end of the list in that slot; deletion requires searching the list and removal. Chained hash tables have advantages over open addressed hash tables in that the removal operation is simple and resizing the table can be postponed for a much longer time because performance degrades more gracefully even when every slot is used. Indeed, many chaining hash tables may not require resizing at all since performance degradation is linear as the table fills. For example, a chaining

hash table containing twice its recommended capacity of data would only be about twice as slow on average as the same table at its recommended capacity.

Chained hash tables inherit the disadvantages of linked lists. When storing small records, the overhead of the linked list can be significant. An additional disadvantage is that traversing a linked list has poor cache performance.

Alternative data structures can be used for chains instead of linked lists. By using a self-balancing tree, for example, the theoretical worst-case time of a hash table can be brought down to O(log n) rather than O(n). However, since each list is intended to be short, this approach is usually inefficient unless the hash table is designed to run at full capacity or there are unusually high collision rates, as might occur in input designed to cause collisions. Dynamic arrays can also be used to decrease space overhead and improve cache performance when records are small.

Some chaining implementations use an optimization where the first record of each chain is stored in the table. Although this can increase performance, it is generally not recommended: chaining tables with reasonable load factors contain a large proportion of empty slots, and the larger slot size causes them to waste large amounts of space.

Open addressing Open addressing hash tables can store the records directly within the array. A hash collision is resolved by probing, or searching through alternate locations in the array (the probe sequence) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequences include:

linear probing

in which the interval between probes is fixed-often at 1,

quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function), and

double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.

The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic hashing falls in-between in both areas. Double hashing can also require more computation than other forms of probing. Some open addressing methods, such as last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

A critical influence on performance of an open addressing hash table is the *load factor*; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering.

Open addressing versus chaining

Chained hash tables have the following benefits over open addressing:

- They are simple to implement effectively and only require basic data structures.

Scanned by CamScanner

- From the point of view of writing suitable hash functions, chained hash tables are insensitive to clustering, only requiring minimization of collisions. Open addressing depends upon better hash functions to avoid clustering. This is particularly important if novice programmers can add their own hash functions, but even experienced programmers can be caught out by unexpected clustering effects.
- They degrade in performance more gracefully. Although chains grow longer as the table fills, a chained hash table cannot "fill up" and does not exhibit the sudden increases in lookup times that occur in a near-full table with open addressing.
- If the hash table stores large records, about 5 or more words per record, chaining uses less memory than open addressing.
- If the hash table is sparse (that is, it has a big array with many free array slots), chaining uses less memory than open addressing even for small records of 2 to 4 words per record due to its external storage.

END TERM EXAMINATION [JUNE-2015]
SECOND SEMESTER [BCA]
DATA STRUCTURE USING C [BCA-108]

Time: 3 Hours **MM : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory.*

Q.1. Answer the following questions: **(5 × 5 = 25)**

Q.1. (a) Explain the working of insertion sort with suitable example.

Ans. Description: Here **A** is an unsorted array having **N** elements from A[0] to A[N-1]. Looping variables are I, J and K. TEMP is swapping variable.

```
1. Repeat For I = 1 to N-1 by 1
2.   Repeat For J = 0 to I-1 by 1
3.       If (A[J] > A[I] Then
4.           TEMP = A[J]
5.           A[J] = A[I]
6.           Repeat For K = I to J + 1 by – 1
7.               A[K] = A[K-1]
             [End of Step 6 For Loop]
         A[K+1] = TEMP
         [End of If]
     [End of Step 2 For Loop]
   [End of Step 1 For Loop]
8. Exit
```

Giving List is: 25, 17, 31, 13, 2

Pass 1 : 25 17 31 13 2

1—0 [25] [17] 31 13 2

17 25 31 13 2

Pass 2:

2—0 [17] 25 [31] 13 2

2—1 17 [25] [31] 13 2

17 25 31 13 2

Scanned by CamScanner

Pass 3:

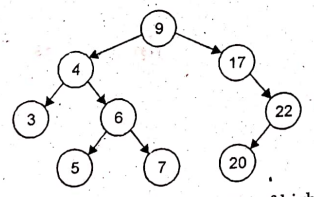
3—0	17	25	31	13	2
3—1	13	17	25	31	2
2—1	13	17	25	31	2
	13	17	25	31	2

Pass 4:

4—0	13	17	25	31	2
4—1	2	13	17	25	31
4—2	2	13	17	25	31
4—3	2	13	17	25	31

Q.1. (*b*) What do you mean by binary search tree? How is different from conventional binary tree?

Ans. A binary search tree (BST), also known as an ordered binary tree, is a node based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node.



The BST data structure is the basis for a number of highly efficient sorting and searching algorithms, and it can be used to construct more abstract data structures including sets, multisets, and associative arrays.

Binary tree is a generic version of binary search tree and it is not ordered. While constructing, binary tree we only follow a rule that every node should have at most two children.

Whereas in case of BST, along with at most two children rule, we follow below rules

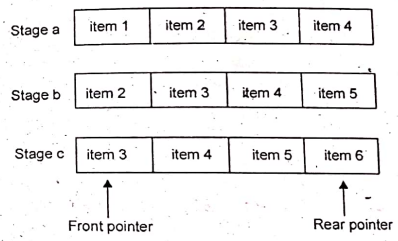
1. All left descendants should possess smaller values than root value.
2. All right descendants should possess larger values than root value.

BST are effective only when we are dealing with data which resides in main memory (RAM).

Q.1. (*c*) What is circular queue and how is it different from linear queue?

Ans. Circular queue is a particular kind of queue where new items are added to the rear of the queue as items are read off the front of the queue. So there is constant stream of data flowing into and out of the queue. Another name for it is 'circular buffer'.

Advantage of a circular queue: It is a structure that allows data to be passed from one process to another whilst making the most efficient use of memory. Circular Queue is a linear data structure and also similar to simple queue. The only difference between linear queue and circular queue is that, in linear queue when the rear points to the last array we cannot insert data even if we have deleted some array. But in circular queue we can insert the items as long as there is free space available. The main advantage of circular queue to simple queue is that it avoids the wastage of space.



A Circular queue (circular Buffer)

The diagram above shows items entering the queue at the rear whilst the item at the front of the queue is read (and removed). Normally, one process is writing items into the queue and a different process is reading items from the queue.

Q.1. (*d*) Explain the differences between linear and non linear data structures.

Ans. Linear data structures: Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a list where data elements can only be added or removed from the top of the list. A queue is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

Scanned by CamScanner

Nonlinear data structures

In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run. Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures. A multidimensional array is simply a collection of one-dimensional arrays. A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements. A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Difference between Linear and Nonlinear Data Structures

Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this nonlinear structure, they might be difficult to be implemented in computer's linear memory compared to implementing linear data structures. Selecting one data structure type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.

Q.1. *(e)* Write a C function for addition of an element in 1 D array.

Ans. 'C' function for addition of an element in 1-D array:

```
void add_in_oned_arr (int a[ ], int n, int pos, int num)
{
int i;
 if(pos>n)
 {
      printf("Value of position should be less than equal to n");
 }
 else
 {
   for(i=n-1;i>=pos-1;i--)
   {
        a[i+1]=a[i];
   }
   a[pos-1]=num;
   n++;
        printf("New array\n");
        for(i=0;i<n;i++)
 {
       printf("%d\n",a[i]);
 }
}
}
```


Q.2. *(a)* What do you mean by a sparse matrix? Write a procedure for addition of two sparse matrices. (6)

Ans. Sparse Matrices: A matrix is sparse if a large number of its elements are 0. Rather than store such a matrix as a two-dimensional array with lots of zeroes, a common strategy is to save space by explicitly storing only the non-zero elements. This topic is of interest to the scientific computing community because of the large sizes of some of the sparse matrices it has to deal with. The specific approach used to store matrices depends on the nature of sparsity of the matrix. Some matrices, such as the tridiagonal matrix have a well-defined sparsity pattern. The tridiagonal matrix is one where all of the nonzero elements lie on one of three diagonals: the main diagonal and the diagonals above and below it.

$$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 3 & 4 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 0 & 0 & 4 & 7 & 4 \\ 0 & 0 & 0 & 3 & 5 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 1 & 2 & 0 & 0 \\ 3 & 3 & 2 & 1 & 0 \\ 2 & 4 & 1 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 3 & 1 & 5 & 8 \\ 0 & 1 & 3 & 2 & 4 \\ 0 & 0 & 2 & 3 & 4 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 4 \end{bmatrix}$$

Tridiagonal Matrix (a) Lower Triangular Matrix (b) Upper Triangular Matrix (c)

Fig. Matrices with regular structures.

There are several ways to represent this matrix as a one-dimensional array. We could order elements by rows giving [2,1,1,3,4,1,1,2,4,7,4,3,5] or by diagonals giving [1,1,4,3,2,3,1,7,5,1,4,2,4]. Above Figure shows other special matrices: the upper and lower triangular matrices which can also be represented using a one-dimensional representation. Other sparse matrices may have an irregular or unstructured pattern.

The advantages of using the *sparse matrices* are:

- Diminishing the memory usage requests;
- Diminishing the processing time by eliminating the useless operations with null values.

The disadvantages of using the *sparse matrices* are:

- The memory space needed to store the *sparse matrix* is much larger than in the case the number of not null values is not much smaller than the number of null values;
- More difficult implementation of the operations at matrix level due to the indirect access way through the data structure defined for storing the *sparse matrix*.

```
/* 'C' function to add two Sparse Matrices */
void addsparse(int b1[MAX][3],int b2[MAX][3],int b3[MAX][3])
{
int t1,t2,i,j,k;
 if(b1[0][0]!=b2[0][0] || b1[0][1]!=b2[0][1])
 {
 printf("\n You have entered invalid matrix!!Size must be equal");
  exit(0);
 }
  t1=b1[0][2];
  t2=b2[0][2];
```

Scanned by CamScanner

```
i=j=k=0;
b3[0][0]=b1[0][0];
b3[0][1]=b1[0][1];
while(i<=t1&&j<=t2)
{
    //row numbers are not equal
    if(b1[i][0]<b2[ j ][0])****
    {
        b3[k][0]=b1[i][0];
        b3[k][1]=b1[i][1];
        b3[k][2]=b1[i][2];
        k++;
        i++;
}
else if(b2[ j ][0]<b1[i][0])          //row numbers are not equal
{
        b3[k][0]=b2[ j ][0];
        b3[k][1]=b2[ j ][1];
        b3[k][2]=b2[ j ][2];
        k++;
        j++;
}
else if(b1[i][1]<b2[ j ][1])          //row numbers are equal, compare column
{
        b3[k][0]=b1[i][0];
        b3[k][1]=b1[i][1];
        b3[k][2]=b1[i][2];
        k++;
        i++;
}
else if(b2[ j ][1]<b1[i][1])          //row numbers are equal, compare column
{
        b3[k][0]=b2[ j ][0];
        b3[k][1]=b2[ j ][1];
        b3[k][2]=b2[ j ][2];
        k++;
        j++;
}
else
{
    b3[k][0]=b1[i][0];                //row and column numbers are equal
    b3[k][1]=b1[i][1];
    b3[k][2]=b1[i][2]+b2[ j ][2];
```



```
    k++;
    i++;
    j++;
}
}
while(i<=t1)        //copy remaining terms from b1
{
    b3[k][0]=b1[i][0];
    b3[k][1]=b1[i][1];
    b3[k][2]=b1[i][2];
    i++;
    k++;
}
while(j<=t2)        //copy remaining terms from b2
{
    b3[k][0]=b2[ j ][0];
    b3[k][1]=b1[ j ][1];
    b3[k][2]=b1[ j ][2];
    j++;
    k++;
}
    b3[0][2]=k-1;   //set number of terms in b3
}
}
```

Q.2. (*b*) Explain the stack data structure. Also write a C function to evaluate a postfix expression. (6.5)

Ans. Stack: It is an ordered collection of data items in which insertions and deletions are made at one end called the top. OR we may define a stack as a **linear data structure** in which items may be inserted or removed only at one end called the top of the stack. A stack may be seen in our daily life, for example, following Figure depicts a stack of dishes. We can observe that any dish may be added or removed only from the top of the stack. It concludes that the item added last will be the item removed first. Therefore, stacks are also called LIFO (Last In First Out) or FILO (First In Last Out) lists. We also call these lists as "piles" or "push-down list".

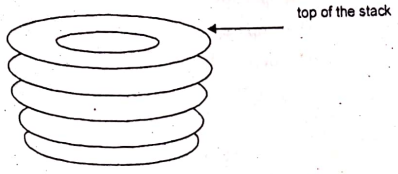


Fig. A Stack of dishes

Scanned by CamScanner

The size of the stack changes dynamically. As elements are entered into the stack its size grows and as elements are removed the size decreases.

An **abstract data type** in stack includes the following operations

1. **MAKENULL(S)** – Stack ***S*** is made ***empty***.
2. **TOP(S)** – The element at the ***top*** of the stack ***S*** is ***returned***.
3. **POP(S)** – The ***top*** element of the stack ***S*** is ***deleted***.
4. **PUSH(x, S)** – The element ***x*** is ***inserted*** into the stack ***S***.
5. **EMPTY(S)** – It returns ***true*** if stack ***S*** is ***empty*** otherwise it returns ***false***

```
/* evaluates the postfix expression */
void calculate( struct postfix *p )
{
    int n1, n2, n3 ;
    while ( *( p -> s ) )
    {
        /* skip whitespaces, if any */
        if ( *( p -> s ) == ' ' || *( p -> s ) == '\t' )
        {
            p -> s++ ;
            continue ;
        }
        /* if digit is encountered */
        if ( isdigit ( *( p -> s ) ) )
        {
            p -> nn = *( p -> s ) - '0' ;
            push ( p, p -> nn ) ;
        }
        else
        {
            /* if operator is encountered */
            n1 = pop ( p ) ;
            n2 = pop ( p ) ;
            switch ( *( p -> s ) )
            {
                case '+' :
                    n3 = n2 + n1 ;
```



```
                    break ;
                case '-' :
                    n3 = n2 - n1 ;
                    break ;
                case '/' :
                    n3 = n2 / n1 ;
                    break ;
                case '*' :
                    n3 = n2 * n1 ;
                    break ;
                case '%' :
                    n3 = n2 % n1 ;
                    break ;
                case '$' :
                    n3 = pow ( n2 , n1 ) ;
                    break ;
                default :
                    printf ( "Unknown operator" ) ;
                    exit ( 1 ) ;
            }
            push ( p, n3 ) ;
        }
        p -> s++ ;
    }
}
```

Q.3. (a) Write C function that finds the total number of nodes in a linked list. (6)

Ans. /* Counting total number of nodes in a linked list */

Method 1: Returning Value

```
int count(struct node *q)
{
int c=0;
    while(q != NULL)
    {
    c++;
    q=q->next;
```

Scanned by CamScanner

```
    }
    return c;
}
```

Method 2: Not Returning Value

```
void count()
{
    struct node *temp;
    int length = 0;
    temp = start;
    while(temp!=NULL)
    {
        length++;
        temp=temp->next;
    }
    printf("nLength of Linked List : %d",length);
}
```

Method 3: Recursive function to Count Number of Nodes in Linked List

```
int count(node *temp)
{
    if(temp == NULL)
        return(0);
    return(1 + count(temp->next));
}
```

Q.3. (b) Write a C function to concatenate two input strings. **(6.5)**

Ans. /* 'C' function to concatenate two strings */

Method 1: Using Pointers

```
void xstrcat ( char *t, char *s )
{
    while ( *t )
        t++ ;
    while ( *s )
        *t++ = *s++ ;
    *t = '\0' ;
}
```

Method 2: Without Using Pointers

```
void concat (char s1 [], char s2 [] {
    int i, j;
    i = strlen (s1);
    for (j = 0; s2 [j] = '\0'; i++, j++) {
        s1[i] = s2 [j];
    }
    s1[i] = '\0';
}
```


Q.4. (a) Define a binary tree. Compute total number of nodes in a binary tree of height *h*. **(6)**

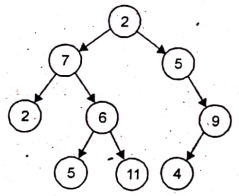
Ans. A binary tree T is defined as a finite set of elements, called nodes, such that:

(a) T is empty (called the null tree or empty tree), or

(b) T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.

If T does contain a root R, then the two trees T_1 and T_2 are called, respectively, the left and right subtrees of R. If T_1 is nonempty, then its root is called the left successor of R; similarly, if T_2 is nonempty, then its root is called the right successor of R.

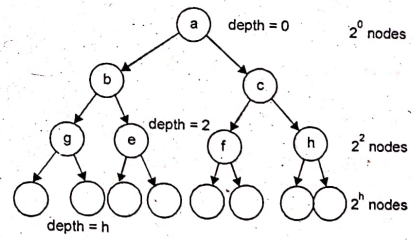
Example:



A labeled binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is unbalanced and not sorted.

A binary tree of height h has $2^{h+1}-1$ nodes. The total number of nodes in a perfect binary tree of height h:

Perfect binary tree of height = h



Total number of nodes = $2^0 + 2^1 + \ldots 2^h$ [It is a. G.P. Series with common ratio 2]

$= [1(2^{h+1} - 1)]/[2 - 1]$

$= 2^{h+1} - 1$

Q.4. (b) Explain with suitable example the insertion and deletion operations in a binary search tree. **(6.5)**

Ans. A binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:

Scanned by CamScanner

- *The left subtree of a node contains only nodes with keys less than the node's key.*
- *The right subtree of a node contains only nodes with keys greater than the node's key.*
- *Both the left and right subtrees must also be binary search trees.*
- *There must be no duplicate nodes.*

Generally, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

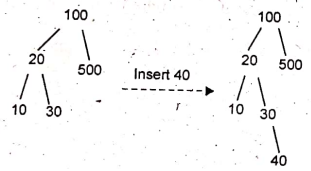
The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Insertion of an element in Binary Search Tree:

A new key is always inserted at leaf. 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.

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

Example:



Deletion of an element in Binary Search Tree:

Let x be a value to be deleted from the BST and let X denote the node containing the value x. Deletion of an element in a BST again uses the BST property in a critical way. When we delete the node X containing x, it would create a "void" that should be filled by a suitable existing node of the BST. There are two possible candidate nodes that can fill this void, in a way that the BST property is not violated:

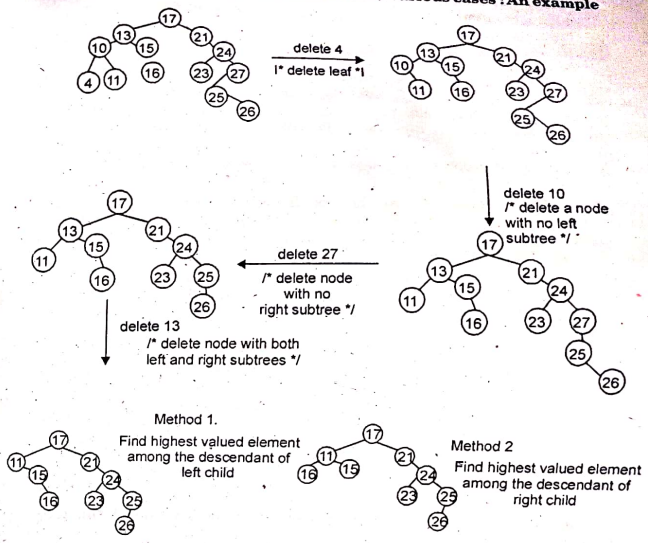
(1) Node containing highest valued element among all descendants of left child of X.

(2) Node containing the lowest valued element among all the descendants of the right child of X.

In case (1), the selected node will necessarily have a null right link which can be conveniently used in patching up the tree.

In case (2), the selected node will necessarily have a null left link which can be used in patching up the tree. Given below figure illustrates several cases for deletion in BSTs.

Figure: Deletion in binary search trees with various cases : An example



Q.5. Explain the preorder, inorder and post order traversals of a binary tree with suitable example. What are left and right skewed binary search trees? (12.5)

Ans. Binary Tree Traversal: Preorder, Inorder, and Postorder.

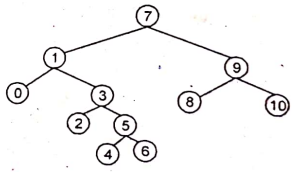
In order to illustrate the binary tree traversals, let us consider the below binary tree:

Preorder traversal: To traverse a binary tree in Preorder, following operations are carried-out

(i) Visit the root,

(ii) Traverse the left subtree, and

(iii) Traverse the right subtree.



Therefore, the Preorder traversal of the above tree will outputs:

7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

Inorder traversal: To traverse a binary tree in Inorder, following operations are carried-out

(*i*) Traverse the left most subtree starting at the left external node,

(*ii*) Visit the root, and

Scanned by CamScanner

(*iii*) Traverse the right subtree starting at the left external node.

Therefore, the Inorder traversal of the above tree will outputs:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Postorder traversal: To traverse a binary tree in Postorder, following operations are carried-out

(*i*) Traverse all the left external nodes starting with the left most subtree which is then followed by bubble-up all the internal nodes,

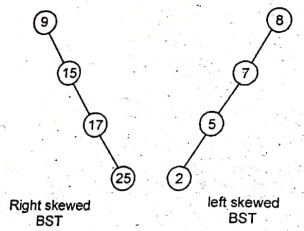
(*ii*) Traverse the right subtree starting at the left external node which is then followed by bubble-up all the internal nodes, and

(*iii*) Visit the root.

Therefore, the Postorder traversal of the above tree will outputs:

0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Left and Right skewed Binary Search Trees: The disadvantage of a BST is that if every item which is inserted to be next is greater than the previous item, then we will get a right skewed BST or if every item which is to be inserted is less than to the previous item, then we will get a left skewed BST.



Q.6. What is AVL tree? Explain various possible nodes types in this tree. Also explain various rotations to balance an AVL tree. (12.5)

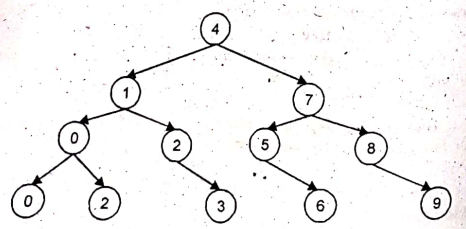
Ans. An AVL tree is a height-balanced binary search tree, where the balance factor is calculated as follows:

Balance Factor = height(left subtree) – height(right subtree)

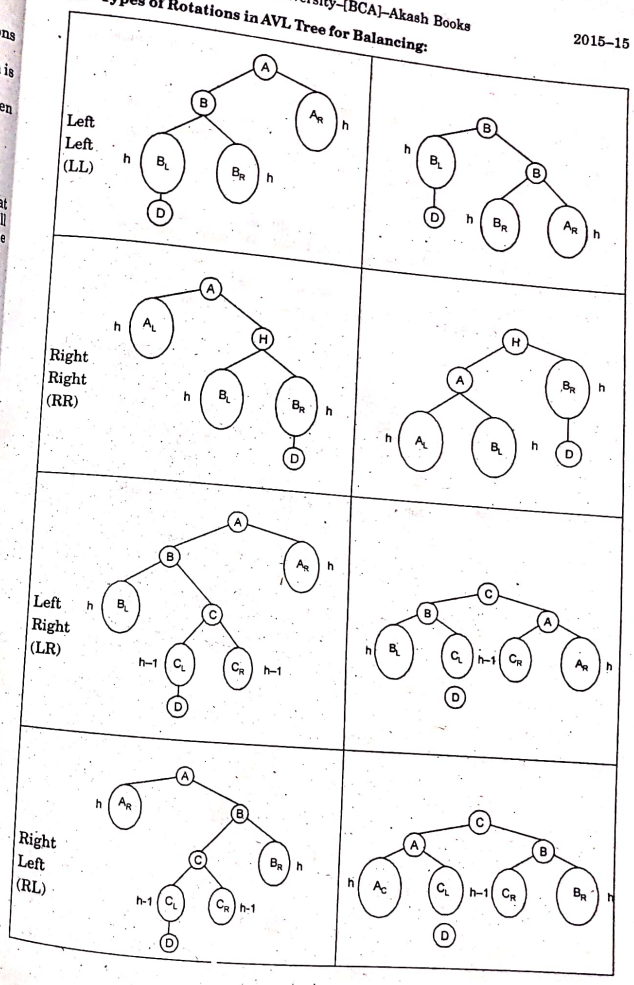
In an AVL tree, the balance factor of every node is no more than 1. In practice, the balance factor is often stored at each tree's node. However a node's balance factor can be also computed from the heights of its subtrees.

The AVL stands for Adelson-Velskii and Landis, who are the inventors of the AVL tree.

Example AVL Tree:



Types of Rotations in AVL Tree for Balancing:

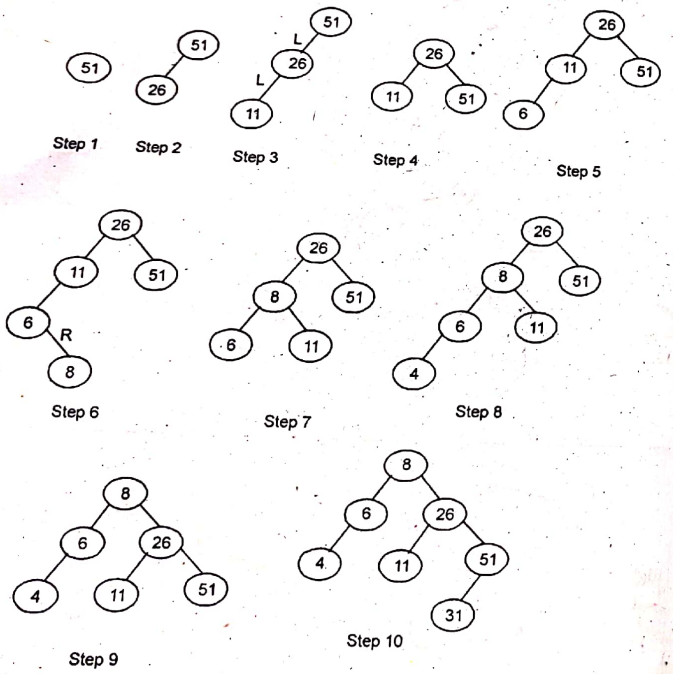


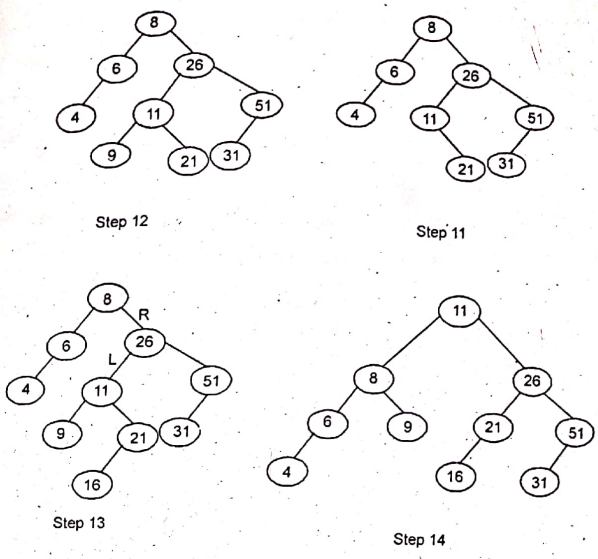
Scanned by CamScanner

Example:

Let us construct the AVL tree with numbers 51, 26, 11, 6, 8, 4, 31, 21, 9, 16. AVL tree is a balanced Binary search tree. So to construct AVL tree, we need to follow below steps

- Insert the new element in such a way that if the element is less than the current node go left and if the element is greater than go right.
- Check for the balance at each node after insertion (difference between left subtree and right subtree should be atmost one) we will see step by step after inserting each number into the tree from starting. The numbers are 51, 26, 11, 6, 8, 4, 31, 21, 9, 16.
- In step1 the tree is empty, so 51 is the first node, in step2, 26 is inserted as a left child to the root node 51 and the tree is balanced.
- In step3, 11 is added as a left child to the node 26, because of this root node 51 is un-balanced, as it is shown in the figure. We need to do the left left rotation which is a single rotation.
- After the rotation, tree is balanced as shown in step4.
- In step5 we are inserting 6 into the tree and which is as a left child to the node value 11.





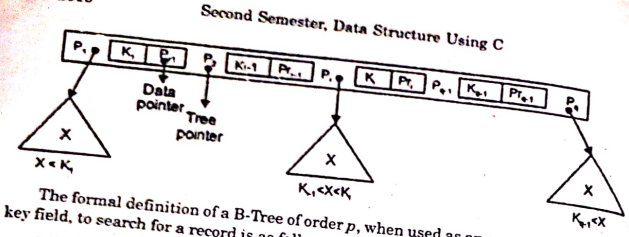
- In step6 we are inserting 8 into the tree and which is as a right child to the node value 6, because of this tree is un-balanced at node value 11 which is marked in the figure.
- For balancing the node we need to do right-left rotation which is a double rotation.
- After the rotation, the tree is balanced and it is like as shown in step7.
- In step8 we are inserting 4 into the tree and which is as a left child to node value 6. Because of this insertion, the tree is un-balanced at node value 26, and we need to do the left-left rotation as we did it earlier.
- After the rotation the tree is like shown in step9. After that we need to insert 31 into the tree and which is as a left child to the node value 51 as shown in step10. And the tree is balanced.
- In step11 we are inserted 21 into the tree which is as a right child to the node value 11. In step12 we inserted 9 as a left child to the node value 11. And the tree is balanced.
- In step13, because of inserting 16 as a left child to the node value 21, the tree is un-balanced at node value 8.
- We need to do the left right rotation which is a double rotation. And the resultant tree is as shown in step14.

Q.7. (a) Describe B-tree indexing with suitable example. **(6)**

Ans. The B-Tree addresses this problem by specifying additional constraints.

B Trees: The B-Tree has additional constraints to ensure the tree is always balanced, and the space wasted by deletion never becomes excessive.

Scanned by CamScanner



The formal definition of a B-Tree of order p, when used as an access structure on a key field, to search for a record is as follows:

1. Each internal node in the tree is of the form:

$(P_1, < K_1, P_{r1} >, P_2, < K_2, Pr_2 > \ldots < K_{q-1}, Pr_{q-1} >, P_q >$, where $q <= p$. Each P is a tree pointer, a pointer to a node in the tree, and each Pr is a data pointer, a pointer to the record whose search key field value is equal to K.

2. The key values, $K_i \ldots K_{q-1}$ are ordered within each node.

3. For all search key values X in the subtree pointed at by P_i, the ith subtree, we have:

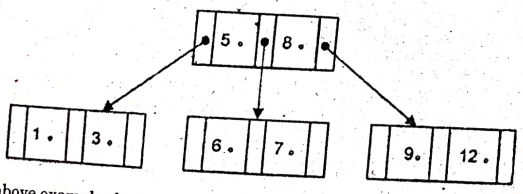
- For $1 < i < q$, $K_{i-1} < X < K_i$,
- For $i = 1$, $X < K_i$,
- For $i = q$, $K_{i-1} < X$

4. Each node has at most p tree pointers.

5. Each node except the root has at least $p/2$ tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

6. A node with q tree pointers, $q<=p$ had $q - 1$ search key field values and hence $q - 1$ data pointers.

7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers P_i are null.



The above example shows a B-Tree of order 3.

The example assumes the B-Tree access structure is on a key field, therefore the values are unique. If the B-Tree is used on a non-key field, the pointer would point to a cluster of blocks that contain blocks of file pointers, similar to option 3 for secondary indexes.

Q.7. (b) Explain different types of searching techniques Give a suitable example to illustrate binary search. **(6.5)**

Ans. Linear Search: In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item. All the elements preceding the search element are traversed before the search element is traversed. i.e. if the element to be searched is in position 10, all elements form 1-9 are checked before 10.

Example: Assume the element 45 is searched from a sequence of sorted elements 12, 18, 25, 36, 45, 48, 50. The Linear search starts from the first element 12, since the value to be searched is not 12 (value 45), the next element 18 is compared and is also not 45, by this way all the elements before 45 are compared and when the index is 5, the element 45 is compared with the search value and is equal, hence the element is found and the element position is 5.

List							i	Result of Comparison
12	18	25	36	45	48	50	1	12 <> 45 : false
12	18	25	36	45	48	50	2	18 <> 45 : false
12	18	25	36	45	48	50	3	25 <> 45 : false
12	18	25	36	45	48	50	4	36 <> 45 : false
12	18	25	36	45	48	50	5	45 = 45 : true

Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. if the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

Example of Binary Search: Let LIST be the following sorted 13-element array:

LIST: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

We apply the binary search to LIST for different values of ITEM(value to be searched).

Suppose ITEM = 40. The search for ITEM in the array LIST is pictured in where the values of LIST[LB] and LIST[UB] in each stage of the algorithm are indicated by **parenthesis** and the value of LIST[MID] by a **bold**. Specifically, LB(lower bound), UB(upper bound) and MID will have the following successive values:

Step 1: Initially, LB = 0 and UB = 12. Hence,

MID = INT[(0 + 12) / 2] = 6 and so LIST[MID] = LIST[6]=55

Since 40 < 55, UB = MID – 1 = 6 – 1 = 5.

Step 2:- Now in this step, LB=0 and UB=5 (from previous step)

MID = INT[(0 + 5) / 2] = 2 and so LIST[MID] = LIST[2] = 30

Scanned by CamScanner

Since 40 > 30, LB = MID + 1 = 2 + 1 = 3.

Step 3:- Now for this step, LB=3(from previous step) and UB=5

MID = INT[(3 + 5) / 2] = 4 and so LIST[MID] = LIST[4] = 40

The search is successful and LOC = MID + 1 = 4 + 1 = 5.

Processing of each step:

(1) (11), 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, (99)

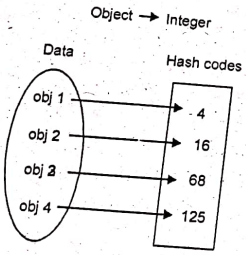
(2) (11), 22, 30, 33, 40, (44), 55, 60, 66, 77, 80, 88, 99

(3) 11, 22. 30, (33), 40, (44), 55, 60, 66, 77, 80, 88, 99 [Successful]

Note: Parenthesis shows positions of LB and UB for each step and element in bold is the element at middle position.

Q.8. (a) Discuss the role of 'Hashing in data searching. How is it different from other searching techniques? (6)

Ans. Role of Hashing in data searching and Hashing procedure: The problem at hands is to speed up data searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array(Linear Search). If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function** . A hash function is a function which when given a key, generates an address in the table



The example of a hash function is a *book call number*. Each book in the library has a *unique* call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, uses Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.

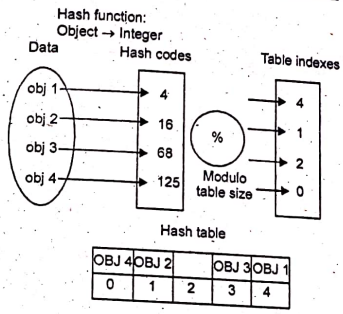
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be processed.

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*

The procedure of storing objects using a hash function is the following.

- Create an array of size *M*. Choose a hash function *h*, that is a mapping from objects into integers *0, 1, ..., M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



Hashing is the process of mapping large amount of data item to a smaller table with the help of a **hashing function.** The essence of hashing is to facilitate the text level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to hand vast amount of data items in a given collection (i.e. collection size). Due to this hashing process, the result is a Hash **data structure** that can store or retrieve data items in an average time disregard lii the collection size.

There are two types of hashing :

1. ***Static hashing:*** In static hashing, the hash function maps search-key values to a fixed set *of locations.*

2. ***Dynamic hashing:*** In dynamic hashing a hash table can grow to handle more items.

The associated hash function must change as the table grows.

The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table. Note: The higher the load factor, the slower the retrieval.

Hashing provides the direct access of record from the file no matter where the record is in the file. This is possible with the help of a hashing function H which map the key with the corresponding key address or location. It provides the key-to-address transformation.

Scanned by CamScanner

We may define *hashing* a process of deriving a storage address from a record key. A function that transforms a key into a table index is called *hash function*. Hashing avoids unnecessary comparisons or searches and each key can be retrieved in a single access.

Five popular hash functions are as follows

Division Method: An integer key *x* is divided by the table size *m* and the remainder is taken as the hash value. It can be defined as

$$H(x) = x\%m+1$$

For example, x = 42 and m = 13, H(42) = 45% 13 + 1 = 3 + 1 = 4

Remember, two records cannot occupy the same position. Such a situation is called a hash collision. To minimize the chances of hash collision in case of division method, always choose a divisor which is a prime number closer to the number of records. For example, in case of 20 records we should choose 19 as divisor.

Midsquare Method: A key is multiplied by itself and the hash value is obtained by selecting an appropriate number of digits from the middle of the square. The same positions in the square must be used for all keys. For example if the key is 12345, square of this key is value 152399025. If 2 digit addresses is required then position 4th and 5th can be chosen, giving address 39.

The mid square method has the advantage that it is a flexible method and it can be suitably modified if needed. The disadvantage of this method is that it does not yield uniform hash values.

Folding Method: A key is broken into several parts. Each part has the same length as that of the required address except the last part. The parts are added together, ignoring the last carry, we obtain the hash address for key K.

This method is flexible like midsquare method and can be modified as desired. In case the key is string of characters, this can be converted into a number consisting of strings of ASCII value of the characters.

Following are the two ways of carrying out the addition in folding method:

***(a)* Shift folding:** In this, all but the last partition are shifted so that the least significant bit of each lines up with the corresponding bit of last partition. For example, let x = 12320324111220. We will partition this digit into five parts, four are three, digits long and one is two digit long.

So, the hash value generated will be 699 (=123 + 203 + 241 + 112 + 20).

(b) Folding at boundaries: In this, the key is folded at the partition boundaries, and digits falling into the same position are added together. For example, taking the same key x, if we apply this method then first we reverse 203 and 112 into 302 and 211 respectively and then all the partitions will be added and a hash value will be generated i.e. 897 (=123+302+241+211+20).

One of the great advantages of folding method is its ability to transform non integer values that are suitable for further hashing action.

Multiplicative method: In this method a real number c such that 0<c<l is selected. For a nonnegative integral key x, the hash function is defined as

$$H(x) = [m(cx\%l)] + 1$$

Here, cx%l is the fractional part of e*x* and [] denotes the greatest integer leas than or equal to its contents.

Digit Analysis: This method forms addresses by selecting and shifting digits of the original key. For a given key set, the same positions in the key and same rearrangement pattern must be used. For example, a key 7654321 is transformed to the address 1247 by selecting digits in position 1, 2, 4 and 7 then by reversing order.

Q.8. (*b*) Write an algorithm for merge sort **(6.5)**

Ans. Merge Sort algorithm

Description: Here 'A' is Linear Array and N is the number of values stored in A. Here recursive function is going to use for MERGE SORT.

Algorithm: Merge Sort (A, BEG, END)

[Here A is an unsorted array. BEG is the lower bound and END is the upper bound.]

1. If (BEG I END) Then

(a) Set MID = (BEG + END)/2

(b) Call MergeSort (A, BEG, MID)

(c) Call MergeSort (A, MID + 1, END)

(d) Call MERGE (A, BEG. MID, END)

[End of If of step–1]

2. Return

Algorithm: MERGE (A, BEG, MID, END)

[Here A is an unsorted array. BEG is the lower bound, END is the upper bound and MID is the middle value array, B is an empty array.]

1. Repeat for I = BEG to END

Set B[I] = A [I [Assign array A to B]

[End of For Loop]

2. Set I = BEG. J = MID + 1, K = BEG

3. Repeat step-4 While (I <=MID) and (J <=END)

4. If (B[I] <=B[J]) Then [Assign smaller value to A]

(a) Set A[K] = B[I]

(b) Set I = 1 + 1 and K = K + 1

Else

(a) Set A [K] = B[J]

(b) Set J = J + 1 and K = K + 1

[End of If]

[End of While Loop of step-3]

Scanned by CamScanner

5. If (I <=MID) Then [Check whether first half has finished or not]

(a) Repeat While (I <=MID)

(i) Set A[K] = B[I]

(ii) Set I = I + 1 and K = K + 1

[End of While Loop step-5 (a)]

Else

(b) Repeat White (J < = END)

i. Set A[K] = B [J]

ii. Set J = J + 1 and K = K + 1

[End of While Loop step-5 (b)]

[End of If of step–5]

6. End

END TERM EXAMINATION [MAY-JUNE–2016]
SECOND SEMESTER [BCA]
DATA STRUCTURE USING C [BCA–108]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q no. 1 which is compulsory.*

Q.1. Define the following terms:

Q.1. (a) Priority Quene **(5×5=25)**

Ans. One way to represent priority queue is through arrays. If an array is used to store the elements of priority queue then insertion is easy but deletion of elements would be difficult. This is because while inserting elements in the priority queue they are not inserted in an order. As a result, deleting an element with the highest priority would require examining the entire array to search for such an element. Moreover, an element in a queue can be deleted from the front end only.

The array element of priority queue can have the following structure:

```
struct data
{
int item;
int priority;
int order;
};
```

The structure holds the type of data item, priority of the element and the order in which the element has been added.

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with same priority are processed according to the order in which they were added to the queue.

An example of a priority queue is time sharing system: programs of high priority are processed first, and programs with the same priority form a standard queue. An efficient implementation for the Priority Queue is to use heap, which in turn can be used for sorting purpose called heap sort.

Q.1.(b) Linked List.

Ans. A linked list, or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

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

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

Scanned by CamScanner

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.

2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

1. Singly/linear Linked List.
2. Doubly Linked List.
3. Circular Linked List.
4. Circular Doubly Linked List.

Q.1.(c) Binary tree

Ans. A Binary tree is a tree data structure in which each node has **at most** two children (referred to as the *left* child and the right child). In a binary tree, the *degree* of each node can be at most two. Binary trees are used to implement binary search trees and binary heaps, and are used for efficient searching and sorting. A binary tree is a special case of a K-ary tree, where k is 2.

Properties of binary trees:

- The number of nodes n in a **perfect binary tree** can be found using this formula: $n = 2^{h+1}-1$ where h is the depth of the tree.
- The number of nodes n in a binary tree of height h is at least $n = h + 1$ and at most $n = 2^{h+1}-1$ where h is the depth of the tree.
- The number of leaf nodes l in a **perfect binary tree** can be found using this formula: $l = 2^h$ where h is the depth of the tree.

The number of nodes n in a **perfect binary tree** can also be found using this formula: $n = 2l-1$ where l is the number of leaf nodes in the tree.

For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.

Q.1.(d) Selection sort

Ans. Algorithm for Selection Sort:

Description: Here **A** is an unsorted array having **N** elements from A[0] to A[N-1]. Looping variables are I and J. TEMP is swapping variable.

1. Repeat For I = 0 to N-2 by 1
2. Repeat For J = I+1 to N-1 by 1
3. If (A[I] > A[J]) Then
4. TEMP=A[I]
5. A[I]=A[J]
6. A[J]=TEMP

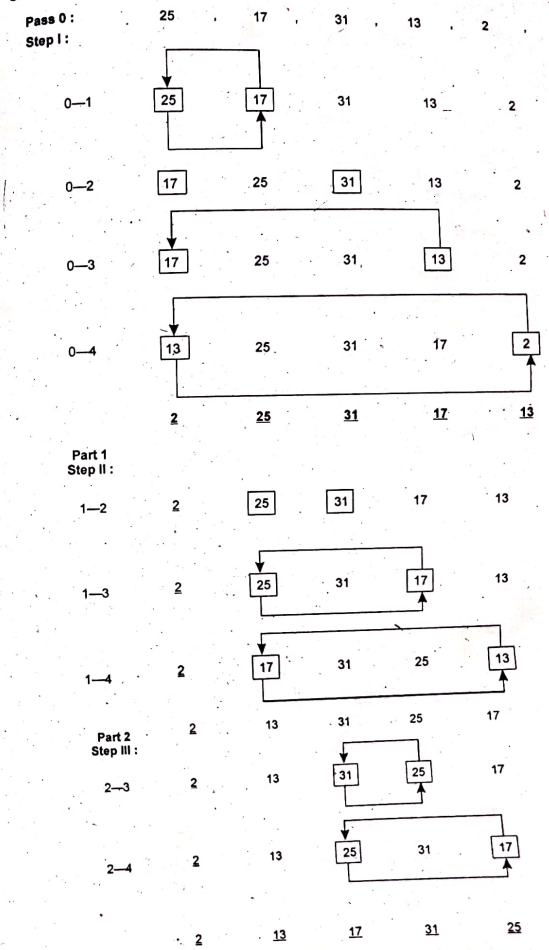
   [End of If]

   [End of Step 2 For Loop]

   [End of Step 1 For Loop]
7. Exit

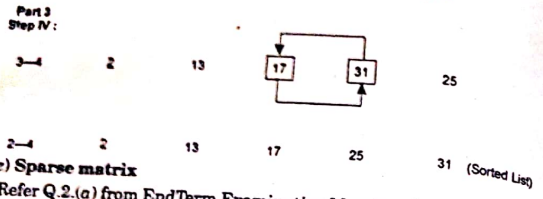
Various stages during Selection Sort procedure on the given list: There are 5 elements in the list *i.e.*, $n = 5$.

Given list is: 25, 17, 31, 13, 2



Scanned by CamScanner

Part 3
Step IV :



Q.1.(e) Sparse matrix

Ans. Refer Q.2.(a) from EndTerm Examination May-June 2015.

Q.2. Write C program to perform PUSH and POP operations on a stack. (12.5)

Ans. Stack can be implemented in two ways

(a) Using Arrays

(b) Using Linked list

Here we are going to discuss Array Implementation of Stack as below:

```
#include<stdio.h>
#include<conio.h>
#define MAXSIZE 10
void push();
int pop();
void traverse();
int stack[MAXSIZE];
int top=-1;
void main()
{
int choice;
char ch;
do
{
    clrscr();
    printf("\n1. PUSH");
    printf("\n2. POP" );
    printf("\n3. TRAVERSE");
    printf("\n Enter your choice");
    scanf("%d", &choice);
    switch(choice)
    {
    case 1: push();
        break;
    case 2: printf("\n The deleted element is %d", pop());
        break;
    case 3: traverse();
        break;
default: printf("\n You Entered Wrong Choice");
```



```
    }
    printf("\n Do You Wish To Continue (Y/N)");
    fflush(stdin);
    scanf("%c", &ch);
    } while(ch=='Y' || ch=='y');
}
    Void push()
{
    int item;
    if(top == MAXSIZE - 1)
 {
      printf("\nThe Stack Is Full");
      getch();
  exit(0);
  }
else
 {
  printf("Enter the element to be inserted");
  scanf("%d", &item);
  top= top+1;
  stack[top] = item;
  }
}
int pop()
{
int item;
if(top == -1)
 {
  printf("The stack is Empty");
  getch();
  exit(0);
  }
else
  {
  item = stack[top];
  top = top-1;
  }
return(item);
}
void traverse()
{
int i;
if(top == -1)
```

Scanned by CamScanner

```
    {
     printf("The Stack is Empty");
     getch();
     exit(0);
    }
  else
    {
     for(i = top; i>=0; i)
     {
      printf("Traverse the element");
      printf("\n %d", stack[i]);
     }
    }
}
```

Q.3. (a) Differentiate between linear search and Binary search. **(6.5)**

Ans. Linear Search: This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is $O(n)$.

Binary search: If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] $= x$ (successful search). If 'x' is not in the list then j is to set to zero (unsuccessful search).

In Binary search we jump into the middle of the list, where we find key a[mid], and compare 'x' with a[mid]. If $x = a$[mid] then the desired element has been found. If $x <$ a[mid] then 'x' must be in that portion of the list that precedes a[mid]. Similarly, if a[mid] $> x$, then further search is only necessary in that part of the list which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a list, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Linear Search v/s Binary Search

- Input data needs to be sorted in case of Binary Search but data can be sorted/ unsorted in Linear Search
- Linear search use the sequential access whereas Binary search access data randomly.
- Time complexity in case of linear search is O(n) whereas Binary search has time complexity of $O(\log_2 n)$.

- Linear search performs equality comparisons and Binary search performs ordering comparisons

Q. 3. (b) Write an algoritm to convert an infix expression to post fix expression. **(6)**

Ans. Algorithm to convert an infix expression to postfix expression:-

The algorithm transforms the infix expression Q into equivalent postfix P. The algorithm uses stack to temporarily hold operators and left parenthesis. The postfix expression P will be constructed left to right using operands from Q and operators which are removed from STACK. We begin by pushing left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is finished when STACK is empty.

Step 1: PUSH the left parenthesis "(" on to STACK and add right parenthesis ")" at the end of Q.

Step 2: Scan Q from left to right and repeat step 3 to 6 for each element of Q, until the attack is empty.

Step 3: If an operand is encountered, add it to P.

Step 4: If "(" is encountered, push it onto stack and this "(" can be removed or popped only when ")" is encountered .

Step 5: If an operator @ is encountered

a. Repeatedly pop from the STACK and add to P each operator (on the top of stack, which has the same precedence or higher precedence than operator @.

b. PUSH to stack.

Step 6: If ")" is encountered

a. Repeatedly pop from the STACK and add to P each operator (on the top of stack) until left parenthesis "(" is encountered .

b. Remove the left parenthesis "(".

[Do not add left parenthesis to P].

Step 7. Exit

Q.4. Write an algoritm to sort a number list using insertion sort and provide an example to verify the algorithm. **(12.5)**

Ans. Algorithm for Insertion Sort

Description: Here A is an unsorted array having N elements from A[0] to A[N-1]. Looping variables are I, J and K. TEMP is swapping variable.

1. Repeat For I = 1 to N-1 by 1
2. Repeat For J = 0 to I-1 by 1
3. If (A[J] > A[I]) Then
4. TEMP=A[J]
5. A[J]=A[I]
6. Repeat For K = I to J+1 by -1
7. A[K]=A[K-1]

[End of Step 6 For Loop]

A[K+1]=TEMP

[End of If]

[End of Step 2 For Loop]

[End of Step 1 For Loop]

8. Exit

Scanned by CamScanner

Process of Insertion Sort:

The following steps are involved in the insertion sort:

Let, if these are 'n' elements in an array, then,

(I) Compare 0th element with 1st element and if found greater, then space for insection is created by shifting all the elements one position to the right and then insert the element.

(II) Compare 2nd element with 0th and 1st element.

(III) Compare 3rd element with 0th, 1st and 2nd element and so on.

Last Step. Compare $(n-1)^{th}$ element with 0th to $(n-2)^{th}$ elements.

there are 5 elements in the list (i.e.) $n=5$

Assuming List is: 25, 17, 31, 13, 2

Pass 1: 25, 17, 31, 13, 2

Step 1:

1 – 0 [25] [17] 31 13 2

17 25 31 13 2

Part 2

Step II

2–0 [17] 25 [31] 13 2

2–1 17 [25] [31] 13 2

17 25 31 13 2

Part 3

Step III

3–0 [17] 25 31 [13] 2

3–1 13 [17] 25 [31] 2

3–2 13 17 [25] [31] 2

13 17 25 31 2

Part 4.

Step IV.

4–0 [13] 17 25 31 [2]

4–1 2 [13] 17 28 [31]

4–2 2 13 [17] 25 [31]

4–3 2 13 17 [25] [31]

2 13 17 23 31 (Sorted List)

Q.5. (a) What do you know about B tree? Write the steps to create a B Tree? **(7.5)**

Ans. A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

A B-tree of order m is a multiway search tree of order m such that:

- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least ceil(m / 2) (nonempty) children.
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least ceil(m / 2) - 1 keys.

A B-tree is a fairly well-balanced tree by virtue of the fact that all leaf nodes must be at the bottom. Condition (2) tries to keep the tree fairly bushy by insisting that each node have at least half the maximum number of children. This causes the tree to "fan out" so that the path from root to leaf is very short even in a tree that contains a lot of data.

In a B-Tree, the new element must be added only at leaf node. That means, always the new key Value is attached to leaf node only. The insertion operation is performed as follows:

Step 1: Check whether tree is Empty.

Step 2: If tree is Empty, then create a new node with new key value and insert into the tree as a root node.

Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.

Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

Step 5: If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.

Step 6: If the splitting occurs to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Q.5.(b) The following sequence given the pre order and inorder of the Binary Tree T.

Pre Order: A B D G C E H I F

Inorder D G B A H E I C F

Draw the diagram of the tree. **(5)**

Ans. Give sequence in

Pre-order A B D G C E H I F

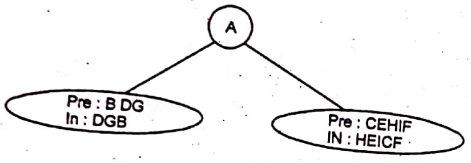
In-Order D G B A H E I C F

Remember, Pre order always gives us the root value. By looking at the Pre-order sequence we can predict and that value 'A' is the root of the tree and the value on the left

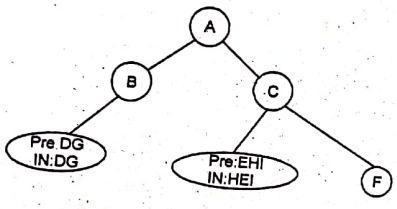
Scanned by CamScanner

side of the of 'A' in Inorder sequence makes its left subtree and values on the right side of 'A' in inorder sequence makes its right subtree. Following this, we can draw the tree as below:

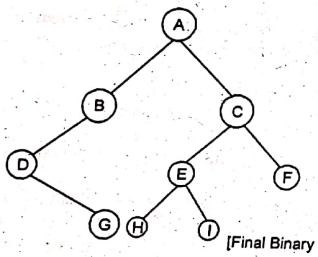
Step 1



Step 2



Step 3



[Final Binary tree]

Q.6. Write a program to search an element using Binary search technique. **(12.5)**

Ans. Method-I : Implementation of Binary search using Iteration

```
#include <stdio.h>
#include <conio.h>
void read(int a[100]);
void traverse(int a[100]);
int binarysearch(int a[100], int num);
int n;
void read(int a[100])
{
int i;
```



```
for(i = 0; i<= n-1; i++)
    scanf("%d", &a[i]);

}
void traverse(int a[100])
{
int i;
for(i=0; i<= n-1; i++)
    printf ("%d\t", a[i]);

}
int binarysearch(int a[100], int num)
{
int lower = 0, upper = n-1 , mid, flag=1;
mid = ( lower + upper ) / 2 ;
    while(lower <= upper)
    {
        if ( a[mid] == num )
        {
        printf ( "The number is at position %d in the array.", mid+1) ;
        flag = 0 ;
        break ;
        }
        if ( a[mid] > num )
        upper = mid - 1 ;
        else
        lower = mid + 1 ;
    mid = ( lower + upper ) / 2 ;
}
return flag;
}
void main()
{
        int a[100] ;
        int flag, num;
        clrscr( ) ;
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter elements in array");
        read(a);
        printf ( "Enter number to search: " ) ;
        scanf ( "%d", &num ) ;
        flag = binarysearch(a,num);
        if ( flag == 1)
        printf ( "Element is not present in the array." ) ;
        getch( ) ;
}
```

Scanned by CamScanner

Method 2: Recursive implementation of Binary Search

```
#include <stdio.h>
void read(int a[100], int n);
void traverse(int a[100], int n);
int binary_search(int a[100],int n,int num,int lower,int upper);
void read(int a[100], int n)
{
int i;
    for(i = 0; i<=n-1;i++)
        scanf("%d", &a[i]);
}
void traverse(int a[100], int n)
{
int i;
    for(i=0; i<=n-1;i++)
        printf("%d\t", a[i]);
}
int binary_search(int a[100],int n,int num,int lower,int upper)
{
  int mid,flag=1;
  if(lower<=upper)
  {
    mid=(lower+upper)/2;
        if(num==a[mid])
    {
      printf("%d is found at position %d", num, mid+1);
      flag=0;
    }
    else if(num<a[mid])
      {
        return binary_search(a,n,num,lower,mid-1);
      }
      else
        return binary_search(a,n,num,mid+1,upper);
  }
return flag;
}
void main()
{
 int a[100],i,n,num,flag,lower,upper;
 printf("Enter the size of an array:");
 scanf("%d", &n);
printf("Enter the elements of the array:");
```



```
read(a,n);
printf("Enter the number to be search:");
scanf("%d",&num);
lower=0,upper=n-1;
flag=binary_search(a,n,num,lower,upper);
if(flag==1)
    printf("Number is not found.");
getch();
}
```

Q.7. (a) Define Queue in what ways a queue can be implemented? (5)

Ans. A queue is a linear list in which items may be added only at one end and items may be removed-only at the other end. The name "queue" likely comes from the everyday use of the term. Consider: queue of people waiting at a bus stop, as pictured in fig. below. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first Clearly, the first person in the line is the first person to leave. Thus queues are also called first-in first-out (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others.

A queue is a linear list of elements in which deletions can take place only at one end, called the front, and insertions can take place only at the other end, called the rear. The terms "front" and "rear" are used in describing a linear list only when it is implemented as, a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on. An important example of a queue in computer science occurs in a timesharing system in which programs with the same priority form a queue while waiting to be executed.

Operations performed on Queues:-

The operations on a queue are

1. ***Insert(queue, element):*** The arguments of the function are the *queue* and *element* is the item to be inserted into the queue at the *rear* end.

2. ***Deli tent = remove(queue)***: This function deletes the front element from the queue *queue* and sets *delitem* to its contents.

3. ***Empty (queue):*** The function returns a false or a true depending on whether or not the queue contains any elements.

The insert function can always be performed since there is no limit on the number of elements in a queue in linked list represenation only.

The remove function can be performed only if the queue is non-empry. The result of an attemept to remove an element from an empty queue is called the underflow.

The empty operation can always be performed.

Scanned by CamScanner

Implementation of Queues

Each type of Queue can be implemented by using two ways:

- Array Representation [Static Representation]
- Linked List Representation [Dynamic Representation]

Q.7.(b) Write code for insertion and deletion in a queue. (7.5)

Ans. Function to insert into and delete the element from linear queue using Linked List implementation:

```
struct node
{
    int data ;
    struct node *next ;
};
struct queue
{
    struct node *front ;
    struct node *rear ;
};
/* adds an element to the queue */
void addq ( struct queue *q, int item )
{
    struct node *temp ;
    temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
if ( temp == NULL )
    printf ( "\nQueue is full." ) ;
temp -> data = item ;
temp -> next = NULL ;
if ( q -> front == NULL )
{
    q -> rear = q -> front = temp ;
    return ;
}
q -> rear -> next = temp ;
q -> rear = q -> rear -> next ;
}
/* delete an element from the queue */
int delq ( struct queue * q )
{
    struct node *temp ;
    int item ;
    if ( q -> front == NULL )
    {
        printf ( "\nQueue is empty." ) ;
```



```
        return NULL ;
    }
    item = q -> front -> data ;
    temp = q -> front ;
    q -> front = q -> front -> next ;
    free ( temp ) ;
    return item ;
}
```

Q.8. Write an algorithm for inserting a node in linked list. (12.5)

(a) At the beginning

Ans. **Algorithm for Insertion of a NODE at the beginning in Linear Linked List:-**

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode().
  newnode = getnode();
- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:
  newnode -> next = start;
  start = newnode;

(b) At the end

Ans. Algorithm for Insertion of a NODE at the end in Linear Linked List:-

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()
  newnode = getnode();
- If the list is empty then *start = newnode.*
- If the list is not empty follow the steps given below:
  temp = start;
  while(temp -> next != NULL)
  temp = temp -> next;
  temp -> next = newnode;

(c) At the specified location

Ans. Algorithm for Insertion of a NODE at specified location in Linear Linked List:-

Description: Here **START** is a pointer variable which contains the address of first node. **NEW** is a pointer variable which will contain address of new node. **N** is the value after which new node is to be inserted and **ITEM** is the value to be inserted.

1. If (START == NULL) Then
2. Print: Linked-List is empty. It must have at least one node
3. Else

Scanned by CamScanner

4. Set PTR = START, NEW = START
5. Repeat While (I != LOC)
6. I=I+1
7. PTR=PTR->LINK
   [End of While Loop]
8. NEW = New Node
9. NEW->INFO = ITEM
10. NEW->LINK = PTR->LINK
11. PTR->LINK = NEW
12. Print: ITEM inserted
    [End of Step 1 If]
13. Exit

END TERM EXAMINATION [MAY-JUNE 2017]
SECOND SEMESTER [BCA]
DATA STRUCTURE USING C [BCA–108]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q no. 1 which is compulsory. Select one question from each unit.*

Q.1. (a) Add and subtract the following two sparse matrices. **(5)**

00000	00000
00000	02000
20000	00000
00000	00000
00090	00000

Ans. Find of all, convert the given matrices in sparse representation as below:

Matrix 1: (5 × 5)

Row	Column	Value
2	0	2
4	3	9

Matrix 2: (5×5)

Row	Column	Value
1	1	2

Result of Addition: (5×5)

Matrix 1 + Matrix 2

Row	Column	Value
1	1	2
2	0	2
4	3	9

Result of subtraction: (5×5)

Matrix 1 – Matrix 2

Row	Column	Value
1	1	–2
2	0	2
4	3	9

Note Row-major order (3-tuple) representation is used for sparse representation.

Q.1. (b) Perform insertion sort on the following values. **(5)**

6, 55, 11, 10, 18

Ans. Insertion Sort

Given list is

6, 55, 11, 10, 18

Step 1 : $\boxed{6}$ $\boxed{55}$ 11 10 18

No swap

6 55 11 10 18

Scanned by CamScanner

Step 2 : [6] 55 [11] 10 18

No swap

6 [55] [11] 10 18

6 11 55 10 18

Step 3 : [6] 11 55 [10] 18

No swap

6 [11] 55 [10] 18

6 10 [11] [55] 18

No swap

6 10 11 55 18

Step 4 : [6] 10 11 55 [18]

No swap

6 [10] 11 55 [18]

No swap

6 10 [11] 55 [18]

No swap

6 10 11 [55] [18]

6 10 11 18 55

[Final sorted list]

Note: Please note that total no. of steps are 4 (if n = 5) and comparisons in each step are 1, 2, 3, 4 respectively.

Q.1. (c) Convert the following infix expression into postfix expression. (5)

(A + B – C * D) /H

Ans. Given inflix Expression is

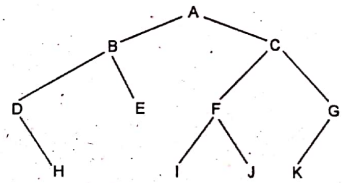
Q : (A + B – C * D)/H

Symbol Scanned (From Left to Right)	Operator Stack	Postfix Expression (P)
(	(	Empty
A	(	A
+	(+	A
B	(+	AB
–	(–	AB+
C	(–	AB+C
*	(–*	AB + C
D	(–*	AB + CD
)	Empty	AB + CD*–
/	/	AB + CD *–
H	/	AB + CD* – H
Empty	Empty	AB + CD* – H/

Final Postfix Expression is

$$\boxed{AB+CD*-H/}$$

Q.1: (d) Write the preorder traversal of the following tree. (5)



Ans. Pre-order Traversal of the given tree

Pre–Order Traversal (Root–Left–Right):

A B D H E C F I J G K

Q.1. (e) Write a Recursive function to count number of nodes in Tree. (5)

Ans. Function to count number of nodes in a tree

```
#include <stdio.h>
#include <stdlib.h>
/* Structure of node for a tree*/
struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
};
void createbinary();
void preorder(node *);
int count(node*);
node* add(int);
typedef struct btnode node;
node *ptr, *root = NULL;
int main()
```

Scanned by CamScanner

```
{
 int c;

    createbinary();
    preorder(root);
    c = count(root);
printf("\nNumber of nodes in binary tree are:%d\n", c);
}
/* constructing the binary tree */
void createbinary()
{
    root = add(50);
    root->l = add(20);
    root->r = add(30);
    root->l->l = add(70);
    root->l->r = add(80);
    root->l->l->l = add(10);
    root->l->l->r = add(40);
    root->l->r->r = add(60);
}
/* Add the node to binary tree */
node* add(int val)
{
    ptr = (node*)malloc(sizeof(node));
    if (ptr == NULL)
    {
        printf("Memory was not allocated");
        return;
    }
    ptr->value = val;
    ptr->l = NULL;
    ptr->r = NULL;
return ptr;
}
/* counting the number of nodes in a tree */
int count(node *n)
{
        int c = 1;
        if (n == NULL)
            return 0;
        else
        {
```



```
            c += count(n->l);
            c += count(n->r);
            return c;
        }
}
/* Displaying the nodes of tree in preorder */
void preorder(node *t)
{
        if (t != NULL)
        {
            printf("%d->", t->value);
            preorder(t->l);
            preorder(t->r);
        }
}
```

UNIT-I

Q.2. (a) Classify primitive and non-primitive data structures. Discuss the operations performed on data structures. (6)

Ans. A data structure in computer science is also a way of storing data in a computer so that it can be used efficiently. A well-designed data structure allows a variety of critical operations to be performed, using a few resources, both execution time and memory space, as possible.

Types of Data Structure:-

A data structure can be broadly classified into

(i) Primitive data structure

(ii) Non-primitive data structure

(i) Primitive data structure: Those data structures that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double in case of 'C' are known as primitive data structures.

(ii) Non-primitive data structure: The data structures, which are not primitive (fundamental) are called non-primitive data structures. There are further two types of non-primitive data structures.

(a) Linear Data Structures:- This is the data structure which shows the relationship of adjacency between elements, is said to be linear data structure means in linear data structures, values are arranged in linear fashion. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data. Arrays, Linked Lists, Stacks and Queues are the examples of linear data structures in which values are stored in a sequence.

(b)Non-linear data structure:- The data structure in which data values are not arranged in order/sequence. Tree, graph, table and sets are examples of non-linear data structures.

Common Operations performed on Data Structures:-

Data are processed by means of certain operations which appearing in the data structure. Data has situation on depends largely on the frequency with which specific operations are performed.

Scanned by CamScanner

(1) Traversing: Accessing each records exactly once so that certain items in the record may be processed.

(2) Searching: Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.

(3) Inserting: Adding a new record to the structure.

(4) Deleting: Removing the record from the structure.

(5) Sorting: Managing the data or record in some logical order(Ascending or descending order).

(6) Merging: Combining the record in two different sorted files into a single sorted file.

Q.2. (b) Evaluate the following postfix expression using stacks. **(6.5)**

320, 10, *, 10, 60, 100, +, *, /

Ans. Given Portfix Expresion:

320, 10, *, 10, 60, 100, +, *, /

Evaluation Using Stack

Symbol Scanned	Stack
320	320
10	320,10
*	3200
10	3200,10
60	3200,10,60
100	3200,10,60,100
+	3200,10,160
*	3200,1600
/	2

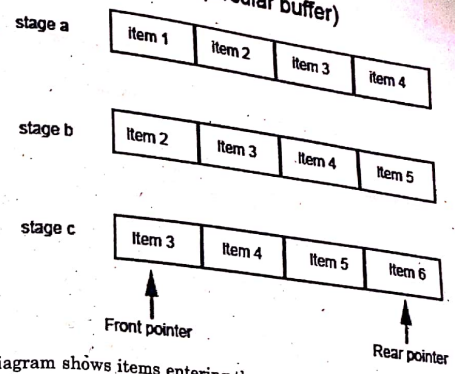
Final Evaluated Result of Given postfix Expresion is $\boxed{2}$.

Q.3. (a) Explain why circular queue is better than linear queue? **(6)**

Ans. This is a particular kind of queue where new items are added to the rear of the queue as items are read off the front of the queue. So there is constant stream of data flowing into and out of the queue. Another name for it is 'circular buffer.

Advantage of a circular queue:- It is a structure that allows data to be passed from one process to another whilst making the most efficient use of memory. Circular Queue is a linear data structure and also similar to linear queue. The only difference between linear queue and circular queue is that, in linear queue when the rear points to the last element of array we cannot insert data even if we have deleted some array elements. But in circular queue we can insert the items as long as there is free space available. The main advantage of circular queue to linear queue is that it avoids the wastage of space. The advantage of this queue is that the overflow occurrences will be very minimal. The actual queue content will move around in the ring in an anti-clockwise direction. This means that the location of the front end as well as the location of the rear end of the queue will move in an anti-clockwise direction.

A circular queue (circular buffer)



The diagram shows items entering the queue at the rear whilst the item at the front of the queue is read (and removed). Normally, one process is writing items into the queue and a different process is reading items from the queue.

1. In Circular Queue, we utilize memory efficiently because in linear queue when we delete any element only from the front using increment by 1, but that position is not used later so when we perform more add and delete operations, memory wastage increases. But in Circular Queue memory is utilized, if we delete any element that position will be used later, because it is circular.

2. Circular queue have less memory consumption as compared to linear queue because while doing insertion after deletion operation it allocate an extra space, the first one remains vacant but in circular queue the first vacant space is used as it comes immediate after the last.

3. In circular queue the memory of the deleted process can be used by some other new process..

4. A standard queue suffers from a re-buffering problem during dequeue operations. By making the queue circular and linking the head to the tail, this alleviates the problem and allows insertion and deletion in constant time.

Q.3. (*b*) Discuss D-queues and priority queues. What are the applications of stacks and queues? **(6.5)**

Ans. DEQUES [DOUBLY ENDED QUEUE]:-

A deque (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name double-ended queue.

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE [I] comes after DEQUE [N] in the

Scanned by CamScanner

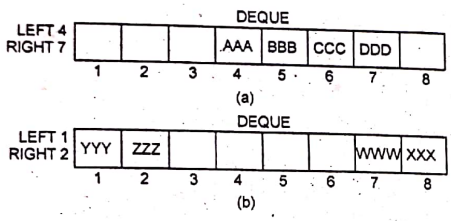
array. Figure below pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two kinds of a deque

Input-restricted deque- is a deque, which allows insertions at only one end of the list but allows deletions at both ends of the list;

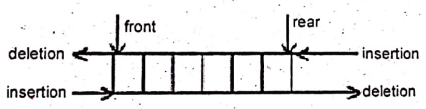
Output-restricted deque- is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

As with queues, a complication may arise (a) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (b) when there is underflow, that is, when an element is to be deleted from a deque, which is empty.



It supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear. .

The DeQueue is represented as follows.



PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.

(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one-way list, and the other uses multiple queues. The ease or difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List [Linked List] Representation of a Priority Queue:

One way to maintain a priority queue in memory is by means of a one-way list is as follows:

(a) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

(b)A node X precedes a node Y in the list

(1)when X has higher priority than Y or

(2)when both have the same priority but X was added to the list before Y

This means that the order in the on the way list corresponds to the order of the priority queue. Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

Array Representation of a Priority Queue:

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.

Stack and queue applications:

Stacks and queues have numerous useful applications.

- *Arithmetic expression evaluation.* An important application of stacks is in *parsing*. For example, a compiler must parse arithmetic expressions written using *infix notation*. For example the following infix expression evaluates to 212.

  (2 + ((3 + 4) * (5 * 6)))

- Function-call abstraction. Most programs use stacks implicitly because they support a natural way to implement function calls, as follows: at any point during the execution of a function, define its state to be the values of all of its variables and a pointer to the next instruction to be executed. The natural way to implement the function-call abstraction is to use a stack. To call a function, push the state on a stack. To return from a function call, pop the state from the stack to restore all variables to their values before the function call and resume execution at the next instruction to be executed.

- M/M/1 queue. One of the most important queueing models is known as an M/M/1 queue, which has been shown to accurately model many real-world situations. It is characterized by three properties:

  →There is one server—a FIFO queue.

  →Interarrival times to the queue obey an exponential distribution with rate λ per minute.

  →Service times from a nonempty queue obey an exponential distribution with rate μ per minute.

- *Load balancing* simulate the process of assigning *n* items to a set of *m* servers. For each item, it chooses a sample of *s* servers and assigns the item to the server that has the fewest current items.

UNIT-II

Q.4. (a) Write a function to insert a node at the end of single linked list. (6)

Ans. Function to insert a node at the end of single linked list:

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

Scanned by CamScanner

```
/* Addition of a node when the list is empty or at the end of the list */
void append(struct node **q,int num)
{
struct node *temp,*r;
if(*q==NULL)
{
    temp=(struct node *)malloc(sizeof(struct node));
    temp->data=num;
    temp->next=NULL;
    *q=temp;
}
else
{
    temp=*q;
    while(temp->next!=NULL)
    {
     temp=temp->next;
    }
r=(struct node *)malloc(sizeof(struct node));
r->data=num;
r->next=NULL;
temp->next=r;
}
}
```

Q.4. (b) Write a function to delete a node from beginning of double linked list. **(6.5)**

Ans.

```
/* Basic structure of a Node in doubly linked list */
struct node
{
  int data;
  struct node * prev;
  struct node * next;
}*head, *last;
/* Delete or remove the first node of the doubly linked list */
void deleteFromBeginning()
{
  struct node *toDelete;
  if(head == NULL)
  {
    printf("Unable to delete. List is empty.\n");
  }
  else
```



```
{
  toDelete = head;
  head = head->next;      // Move head pointer to next node
  head->prev = NULL;      // Remove the link to previous node
  free(toDelete); // Delete the first node from memory
   printf("SUCCESSFULLY DELETED NODE FROM BEGINNING OF THE LIST.\n");
  }
}
```

Q.5. (a) A binary tree T has 09 nodes. The inorder and preorder traversals of T yield the following sequences of nodes. **(6)**

Inorder: D G B A H E I C F

Preorder: A B D G C E H I F

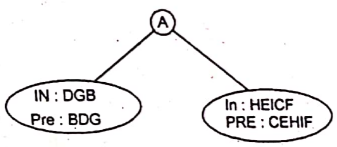
Draw the tree T

Ans. Given Traversals of T

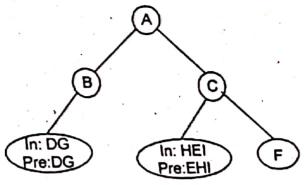
Inorder: D G B A H E I C F

Preorder: A B D G C E H I F

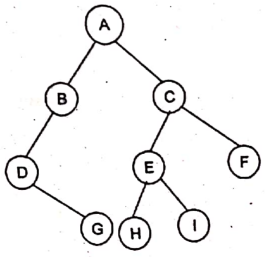
Step 1



Step 2



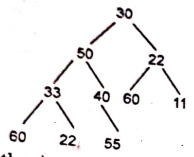
Step 3



Scanned by CamScanner

Finally constructed Binary Tree.

Q.5. (b) Consider the following binary tree T with N=10 nodes. What is the inorder traversal of the tree? **(6.5)**



Ans. In order Traversal of the given tree with 10 nodes:
Inorder Traversal (Left – Root–Right):
60, 33, 22, 50, 55, 40, 30, 60, 22, 11

UNIT–III

Q.6. (a) Construct B-tree of order 3 by inserting the following keys in the order shown.
18, 19, 6, 10, 40 **(6.5)**

Ans. We need to create a B-tree of order 3 with key value:
18, 19, 6, 10, 40

Minimum no. of children for a non-leaf node = $\frac{n}{2} = \frac{3}{2} = 2$

Maximum no. of children for a non-leaf node = n = 3

Minimum and Maximum no. of values in a non-leaf node = 1 (2 – 1) and 2 (3 – 1) respectively

Minimum and Maximum no. of children for a root node = 2 and 3 repectively

Minimum and Maximum no. of values in root node = 1 and 2 respectively.

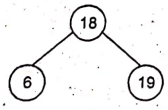
We need to split a node if it contains more than maximum no. of values allowed from $\frac{n}{2} = \frac{3}{2} = 2$ nd value.

Step 1: Insert 18

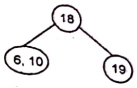
Step 2 : Insert 19

Step 3 : Insert 6

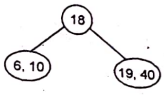
Now, the root node contains more values than maximum no. of values, so, split the root node from 2nd value as shown



Step 4 : Insert 10



Step 5 : Insert 40



final B–Tree

Q.6. (b) Construct Binary Search Tree of the following keys in the order shown.
1, 2, 3, 15, 8, 25, 7, 9, 10, 13 **(6)**

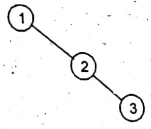
Ans. Given List is

1, 2, 3, 15, 8, 25, 7, 9, 10, 13

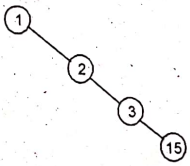
Step 1: Insert 1

Step 2: Insert 2

Step 3: Insert 3

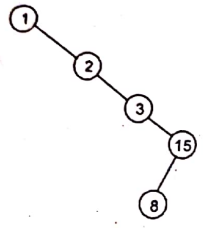


Step 4 : Insert 15

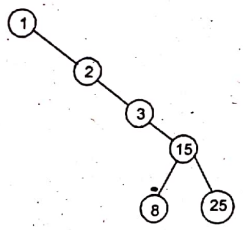


Scanned by CamScanner

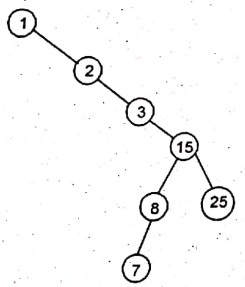
Step 5 : Insert 8



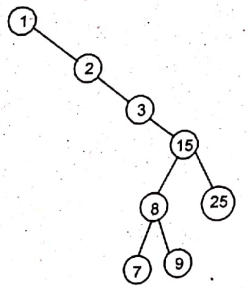
Step 6: Insert 25



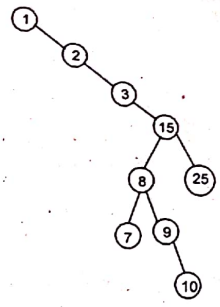
Step 7 : Insert 7



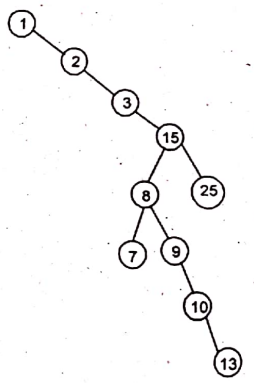
Step 8: Insert 9



Step 9: Insert 10



Step 10: Insert 13



Final Binary Search tree

Q.7. (a) Construct an AVL search tree of the following values **(6)**

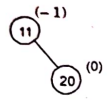
11, 20, 23, 5, 3

Ans. Given list of values for AVL Tree Creation: 11, 20, 23, 5, 3

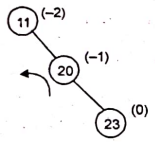
Step 1 : Insert 11

Step 2 : Insert 20

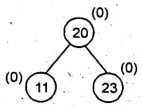
Scanned by CamScanner



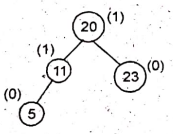
Step 3: Insert 23



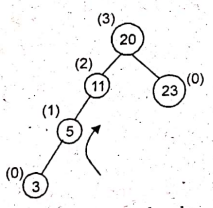
(Right unbalanced Tree)
So, we need to use left rotation to balance the above tree.



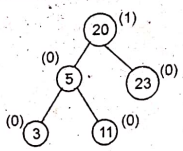
Step 4: Insert 5



Step 5: Insert 3



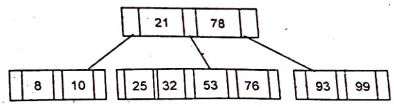
Now, we need to use right rotation to balance the above tree.



Final AVL Tree.

Q.7. (b) Insert the following values in the order of their occurrence 30, 31 in

the given B tree of order 5. **(6.5)**



Ans. For a B-Tree of order 5, we should know the following

Minimum no. of children for a non leaf node = $\frac{n}{2} = \frac{5}{2} = 3$

Maximum no. of children for a non-leaf node = n = 5

Minimum and maximum no. of values in a non-leaf node

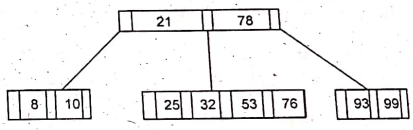
= 2 (3 – 1) and 4 (5 – 1) respectively.

Minimum and maximum no. of children for a root node = 2 and 5 respectivelly.

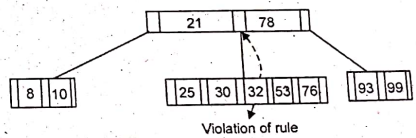
Minimum and maximum no. of values in a root node = 1 and 4 respectively.

We need to split a node if it contain more than maximum no. of value allowed, from the $\frac{n}{2} = \frac{5}{2}$ = 3rd value in the node by moving 3rd value to the parent node.

Now, Given B-Tree

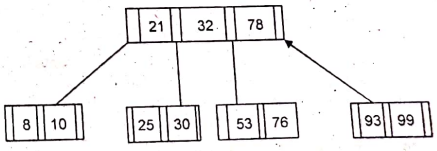


Insert 30



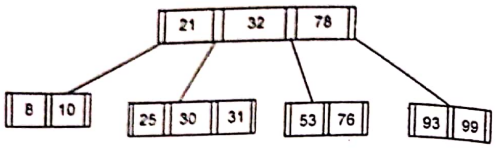
Violation of rule

Now, we need to break the node from 3rd value i.e. from 32 as this node is having more than the allowed no. of values.



Now, Insert 31

Scanned by CamScanner



Final B-Tree

UNIT–IV

Q.8. (a) Define hashing. Why do we use hashing? Discuss any two hashing methods with example. (6.5)

Ans. Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function. The essence of hashing is to facilitate the next level searching method when compared with the linear or binary search. The advantage of this searching method is its efficiency to handle vast amount of data items in a given collection (i.e. collection size). Due to this hashing process, the result is a Hash data structure that can store or retrieve data items in an average time disregard to the collection size.

There are two types of hashing :

1. Static hashing: In static hashing, the hash function maps search-key values to a fixed set of locations.

2. Dynamic hashing: In dynamic hashing a hash table can grow to handle more items.

The associated hash function must change as the table grows.

The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table. Note: The higher the load factor, the slower the retrieval.

Why do we use hashing?

Hashing provides the direct access of record from the file no matter where the record is in the file. This is possible with the help of a hashing function H which map the key with the corresponding key address or location. It provides the key-to-address transformation.

We may define *hashing* a process of deriving a storage address from a record key. A function that transforms a key into a table index is called *hash function*. Hashing avoids unnecessary comparisons or searches and each key can be retrieved in a single access.

Three popular hashing functions are as follows:-

Division Method: An integer key x is divided by the table size m and the remainder is taken as the hash value. It can be defined as

H(x)=x%m+1

For example, x=42 and m=13, H(42)=45%13+1=3+1=4

Remember, two records cannot occupy the same position. Such a situation is called a hash collision. To minimize the chances of hash collision in case of division method, always choose a divisor which is a prime number closer to the number of records. For example, in case of 20 records we should choose 19 as divisor.

Midsquare Method: A key is multiplied by itself and the hash value is obtained by selecting an appropriate number of digits from the middle of the square. The same positions in the square must be used for all keys. For example if the key is 12345, square

of this key is value 152399025. If 2 digit addresses is required then position 4th and 5th can be chosen, giving address 39.

The mid square method has the advantage that it is a flexible method and it can be suitably modified if needed. The disadvantage of this method is that it does not yield uniform hash values.

Folding Method: A key is broken into several parts. Each part has the same length as that of the required address except the last part. The parts are added together, ignoring the last carry, we obtain the hash address for key K.

This method is flexible like midsquare method and can be modified as desired. In case the key is string of characters, this can be converted into a number consisting of strings of ASCII value of the characters.

Following are the two ways of carrying out the addition in folding method:-

(a) Shift folding:- In this, all but the last partition are shifted so that the least significant bit of each lines up with the corresponding bit of last partition. For example, let x= 12320324111220. We will partition this digit into five parts, four are three digits long and one is two digit long.

So, the hash value generated will be 699 (=123+203+241+112+20).

(b) Folding at boundaries:- In this, the key is folded at the partition boundaries, and digits falling into the same position are added together. For example, taking the same key x, if we apply this method then first we reverse 203 and 112 into 302 and 211 respectively and then all the partitions will be added and a hash value will be generated i.e. 897 (=123+302+241+211+20).

One of the great advantages of folding method is its ability to transform non integer values that are suitable for further hashing action.

Q.8. (b) Which searching technique is best and under what conditions? Justify your answer with the help of an example. (6)

Ans. Linear search is used on a collections of items. It relies on the technique of traversing a list from start to end by exploring properties of all the elements that are found on the way.

For example, consider an array of integers of size N. You should find and print the position of all the elements with value x. Here, the linear search is based on the idea of matching each element from the beginning of the list to the end of the list with the integer x, and then printing the position of the element if the condition is 'True'.

For example, consider the following:

0 5 3 8 7 9 2 1 6 4

If you want to determine the position of the occurrence of the number 7 in this array. To determine the positions, every element in the array from start to end, i.e., from index 0 to index 9 will be compared with number 7, to check which element matches the number 7. Linear search works on unsorted as well as sorted list. For unsorted list linear search is better than Binary search but it becomes worst when the list is large in size, in that case it is better to sort the list and then use the Binary search to locate the element.

Time Complexity: The time complexity of the linear search is O(N) because each element in an array is compared only once.

Binary search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems. If all the names in the world are written down together in order and you want to search for the position of

Scanned by CamScanner

a specific name, binary search will accomplish this in a maximum of 35 iterations. Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be sorted. When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced on the basis of the value that is being searched.

Let us consider the following array:

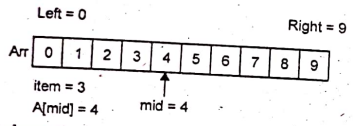
Arr 0 1 2 3 4 5 6 7 8 9

By using linear search, the position of element 8 will be determined in the 9th iteration. Let us see how the number of iterations can be reduced by using binary search. Before we start the search, we need to know the start and end of the range. Lets call them Low and High.

Low = 0

High = n-1

Now, compare the search value K with the element located at the median of the lower and upper bounds. If the value K is greater, increase the lower bound, else decrease the upper bound.



Referring to the above, the lower bound is 0 and the upper bound is 9. The median of the lower and upper bounds is (lower_bound + upper_bound) / 2 = 4. Here a[4] = 4. The value 4>2, which is the value that you are searching for. Therefore, we do not need to conduct a search on any element beyond 4 as the elements beyond it will obviously be greater than 2. Therefore, we can always drop the upper bound of the array to the position of element 4. Now, we follow the same procedure on the same array with the following values:

Low: 0

High: 3

Repeat this procedure recursively until Low > High. If at any iteration, we get a[mid]=key, we return value of mid. This is the position of key in the array. If key is not present in the array, we return –1.

Time complexity : $O(\log_2 N)$

Binary search is always better than the linear search when we need to search the element in sorted list, no matter how much the size of the list is.

Q.9. (a) Compare section sort and merge sort. **(6)**

Ans. Selection Sort: Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted. Selection sort is an unstable sort i.e it might change the occurrence of two similar elements in the list while sorting. But it can also be a stable sort when it is implemented using linked list data structures. Selection sort should not be used to sort large number of elements if the performance matters in that program.

If someone really want to use the selection sort for some reason then try to avoid sorting lists of more than 1000 elements with it or repetitively sorting lists of more than a couple hundred elements.

Time Complexity of Selection Sort is:

Average Case : $O(n^2)$

Worst Case : $O(n^2)$

Best Case: $O(n^2)$

Space Complexity of Selection Sort : O(1)

Advantages:

- Easy Implementation
- In Place Sorting Algorithms

Disadvantages:

- Unstable Sorting Algorithm
- Complexity of $O(n^2)$

Merge Sort: Merge Sort follows the rule of **Divide and Conquer**. In merge sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

The merge sort is slightly faster than the heap sort for the larger sets of elements, but requires twice the memory of the heap sort because of the second array. In merge sort, there are two phases : a partition phase and a merge phase. In the partition phase, if the list of items contains more than one item, the list is split in half and the merge sort algorithm is applied on each half of the list. To merge two lists, need to compare the lowest value in the one list with the lowest value in the other list.

Time Complexity of Merge Sort is:

Worst Case Time Complexity : O(n log n)

Best Case Time Complexity : O(n log n)

Average Time Complexity : O(n log n)

Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Space Complexity of Selection Sort is: O(n)

Advantages:

- Complexity of O(n log(n))
- Stable Sorting Algorithm.
- In-place sorting algorithm
- It is the best Sorting technique used for sorting **Linked Lists**.

Disadvantages:

- Relatively harder to implement.
- Quick sort usually outperforms merge sort
- Not in place sorting algorithm
- It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.

Q.9. (b) Which sorting technique is better and why? Explain with an example. **(6.5)**

Ans. Quick Sort is better among the sorting algorithms. One can say that quick sort is a faster in-place version of the merge sort. The efficiency of the algorithm in quick sort

Scanned by CamScanner

is mainly impacted by which element is chosen as the pivot element. The quick sort is by far the fastest of the common sorting algorithms. Quick sort is best because of its less time and space complexity.

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

This algorithm divides the list into three main parts :

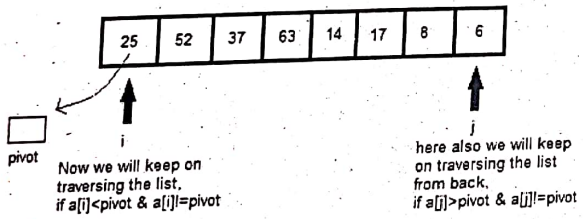
1. Elements less than the Pivot element
2. Pivot element(Central element)
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep on doing this until the complete list is sorted.

How Quick Sort Works



if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.

DIVIDE AND CONQUER - QUICK SORT

Time Complexity of Quick Sort:

- Worst Case Time Complexity : $O(n^2)$
- Best Case Time Complexity : O(n log n)
- Average Time Complexity : O(n log n)

Space Complexity of Quick Sort: O(log n)

Although the worst case time complexity of Quick Sort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, Quick Sort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. Quick Sort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data.

Scanned by CamScanner

END TERM EXAMINATION [MAY-JUNE 2018]
SECOND SEMESTER [BCA]
DATA STRUCTURE USING C [BCA–108]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q. no. 1 which is compulsory. Select atleast one question from each unit.*

Q. 1. (a) Define data structure. In how many ways can you categorized data structures? Explain each of them. **(5)**

Ans. A data structure is an arrangement of data in a computer's memory. In other words, a mathematical or logical model of a particular organization of data is called data structure. A data structure in computer science is also a way of storing data in a computer so that it can be used efficiently.

Thus, a data structure is the portion of memory allocated for a model, in which the required data can be arranged in a proper manner. A well-designed data structure allows a variety of critical operations to be performed, using a few resources, both execution time and memory space, as possible.

Types of Data Structure:-

A data structure can be broadly classified into

(i) Primitive data structure

(ii) Non-primitive data structure

(i) Primitive data structure: Those data structures that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double in case of 'C' are known as primitive data structures.

(ii) Non-primitive data structure: The data structures, which are not primitive (fundamental) are called non-primitive data structures. There are further two types of non-primitive data structures.

(a) Linear Data Structures:- This is the data structure which shows the relationship of adjacency between elements, is said to be linear data structure means in linear data structures, values are arranged in linear fashion. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data. Arrays, Linked Lists, Stacks and Queues are the examples of linear data structures in which values are stored in a sequence.

Types	Description
Arrays	Array is a collection of elements stored in continuous memory locations. It is used in mathematical problems like matrix, algebra etc. each element of an array is referenced by a subscripted variable or value, called subscript or index enclosed in parenthesis.
Linked list	Linked list is a collection of data elements. It consists of two parts: Info and Link. Info gives information and Link is an address of next node. Linked list can be implemented by using pointers.
Stack	Stack is a list of elements. In stack, an element may be inserted or deleted at one end which is known as top of the stack. It performs two operations: Push and Pop. Push means adding an element in stack and Pop means removing an element in stack. It is also called Last-in-First-out (LIFO).

Scanned by CamScanner

Queue	Queue is a linear list of element. In queue, elements are added at one end called rear and the existing elements are deleted from other end called front. It is also called as First-in-First-out (FIFO).

(b) Non-linear data structure:- The data structure in which data values are not arranged in order/sequence. Tree, graph, table and sets are examples of non-linear data structures.

Type	Description
Tree	Tree is a flexible, versatile and powerful non-linear data structure. It is used to represent data items processing hierarchical relationship between the grandfather and his children & grandchildren. It is an ideal data structure for representing hierarchical data.
Graph	Graph is a non-linear data structure which consists of a finite set of ordered pairs called edges. Graph is a set of elements connected by edges. Each elements are called a vertex and node.

Q. 1. (b) Perform selection sort on the following values. **(5)**

6, 55, 11, 10, 18

Ans. Selection Sort

Given list is

6, 55, 11, 10, 18

Step 1: [6] [55] 11 10 18

No Swap

[6] 55 [11] 10 18

No Swap

[6] 55 11 [10] 18

No Swap

[6] 55 11 10 [18]

No Swap

6 55 11 10 18

No Swap

In the step, we compared element at 1st position with each element of the list and because no element is smaller than the element at 1st position so we do not need to swap

Step 2: 6 [55] [11] 10 18

6 [11] 55 [10] 18

6 [10] 55 11 [18]

No Swap

6 10 55 11 18

Step 3: 6 10 [55] [11] 18

6 10 [11] 55 [18]

No swap

6 10 11 55 18

Step 4: 6 10 11 [55] [18]

6 10 11 18 55

[Final Sorted list]

Note: Total no. of steps are 4 (if n = 5) and comparisons in each step are 4, 3, 2, 1, repectively.

Q. 1. (c) Convert the following infix expression into postfix expression. **(5)**

(A*B-C/D)+H

Ans. Given Infix expression is

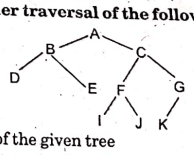
(A*B – C/D) + H

Symbole Scanned (from Left to Right)	Operator stack	Postfix Expression (P)
(	(	Empty
A	(	A
*	(*	A
B	(*	AB
–	(–	AB*
C	(–	AB*C
/	(–/	AB*C
D	(–/	AB*CD
)	Empty	AB*CD/–
+	+	AB*CD/–
H	+	AB*CD/–H
Empty	Empty	AB*CD/–H+

Final Postfix Expression is

AB * CD/ – H+

Q. 1. (d) Write the postorder traversal of the following tree.



Ans. Post-Order Traversal of the given tree

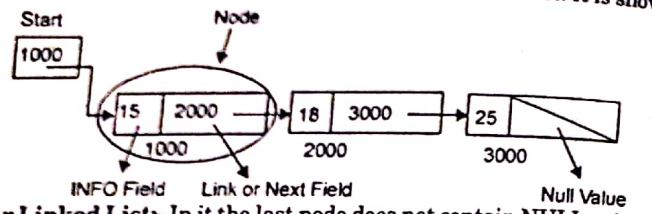
Post-order Traversal (left-Right-Post): D E B I J F K G C A

Q. 1. (e) Explain the difference between a circular linked list and a singly linked list. **(5)**

Ans. Linear Linked List or One Way List or Singly Linked List:- It is linear collection of data elements which are called 'Nodes'. The elements may or may not be stored in consecutive memory locations. So pointers are used maintain linear order. Each node is divided into two parts. The first part contains the information of the element and is called 'INFO Field'. The second part contains the address of the next node and is called 'LINK

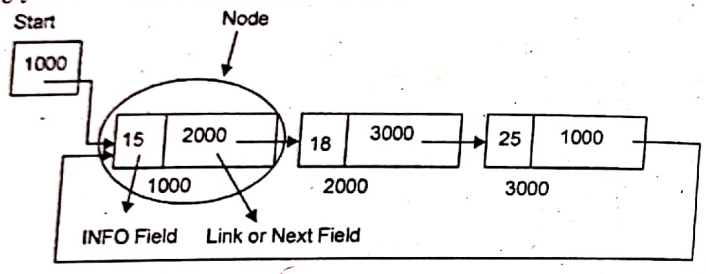
Scanned by CamScanner

Field' or 'NEXT Pointer Field'. The START contains the starting address of the linked list i.e. it contains the address of the first node of the linked list. The LINK Field of last node contains NULL Value which indicates that it is the end of linked list. It is shown below:



Circular Linked List:- In it the last node does not contain NULL pointer. Instead the last node contains a pointer that has the address of first node and thus points back to the first node. It can be Singly as well as Doubly circular linked list.

Singly Circular Linked List is shown below:



UNIT-I

Q. 2. (a) Write a short note on different operations that can be performed on data structures. **(4.5)**

Ans. Refer Q. 2. (a) End Term Examination May-June 2017.

Q. 2. (b) Evaluate the following postfix expression using stacks. **(8)**

320, 10, *, 10, 60, 100, +, *, /

Ans. Given Portfix Expresion:

320, 10, *, 10, 60, 100, +, *,/

Evaluation Using Stack

Symbol Scanned	Stack
320	320
10	320,10
*	3200
10	3200,10
60	3200,10,60
100	3200,10,60,100
+	3200,10,160
*	3200,1600
/	2

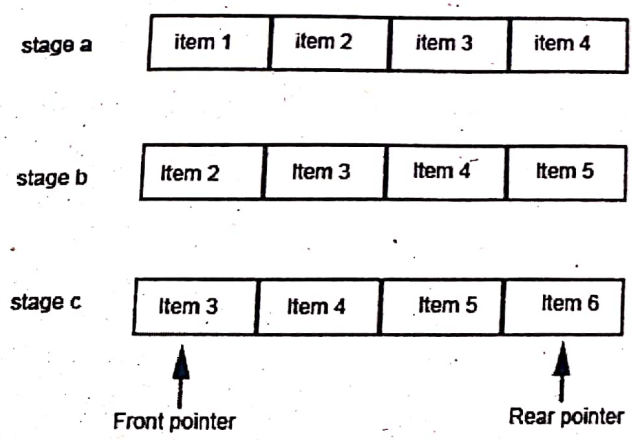
Final Evaluated Result of Given postfix Expresion is $\boxed{2}$.

Q. 3. (a) Explain why circular queue is better than linear queue. Give examples. **(4.5)**

Ans. This is a particular kind of queue where new items are added to the rear of the queue as items are read off the front of the queue. So there is constant stream of data flowing into and out of the queue. Another name for it is circular buffer.

Advantage of a circular queue:- It is a structure that allows data to be passed from one process to another whilst making the most efficient use of memory. Circular Queue is a linear data structure and also similar to linear queue. The only difference between linear queue and circular queue is that, in linear queue when the rear points to the last element of array we cannot insert data even if we have deleted some array elements. But in circular queue we can insert the items as long as there is free space available. The main advantage of circular queue to linear queue is that it avoids the wastage of space. The advantage of this queue is that the overflow occurrences will be very minimal. The actual queue content will move around in the ring in an anti-clockwise direction. This means that the location of the front end as well as the location of the rear end of the queue will move in an anti-clockwise direction.

A circular queue (circular buffer)



The diagram above shows items entering the queue at the rear whilst the item at the front of the queue is read (and removed). Normally, one process is writing items into the queue and a different process is reading items from the queue.

1. In Circular Queue, we utilize memory efficiently because in linear queue when we delete any element only from the front using increment by 1, but that position is not used later so when we perform more add and delete operations, memory wastage increases. But in Circular Queue memory is utilized, if we delete any element that position will be used later, because it is circular.

2. Circular queue have less memory consumption as compared to linear queue because while doing insertion after deletion operation it allocate an extra space, the first one remains vacant but in circular queue the first vacant space is used as it comes immediate after the last.

3. In circular queue the memory of the deleted process can be used by some other new process.

4. A standard queue suffers from a re-buffering problem during dequeue operations. By making the queue circular and linking the head to the tail, this alleviates the problem and allows insertion and deletion in constant time.

Scanned by CamScanner

Q. 3. (b) Discuss D-queues and priority queues. What are the applications of stacks, queues, D-queues and priority queues? (8)

Ans. DEQUES [DOUBLY ENDED QUEUE:- A deque (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name double-ended queue.

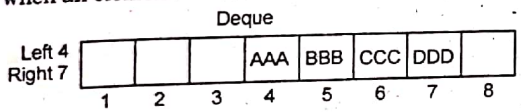
There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE [I] comes after DEQUE [N] in the array. Figure below pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two kinds of a deque

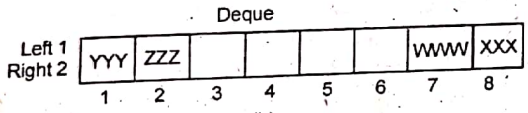
Input-restricted deque- is a deque, which allows insertions at only one end of the list but allows deletions at both ends of the list;

Output-restricted deque- is a deque, which allows deletions at only one end of the list but allows insertions at both ends of the list.

As with queues, a complication may arise (a) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (b) when there is underflow, that is, when an element is to be deleted from a deque, which is empty.



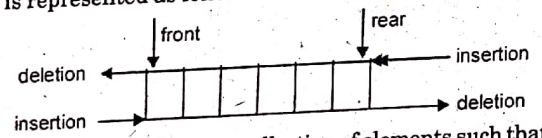
(a)



(b)

It supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear. .

The DeQueue is represented as follows.



Priority Queues: A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.

(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one-way list, and the other uses multiple queues. The ease or

difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List [Linked List] Representation of a Priority Queue:

One way to maintain a priority queue in memory is by means of a one-way list is as follows:

(a) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

(b) A node X precedes a node Y in the list

(1) when X has higher priority than Y or

(2) when both have the same priority but X was added to the list before Y

This means that the order in the on the way list corresponds to the order of the priority queue. Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

Array Representation of a Priority Queue: Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.

Applications of D-queues:-

1. A-Steal job scheduling algorithm

- The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
- The processor gets the first element from the deque.
- When one of the processor completes execution of its own threads it can steal a thread from another processor.
- It gets the last element from the deque of another processor and executes it.

2. Undo-Redo operations in Software applications.

Applications of Priority Queues:-

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation, [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm. Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]

UNIT-II

Q. 4. (a) Write a function to insert a node at the beginning of single linked list. (4.5)

Ans. /* Structure of a node */

```
struct node {
   int data;        // Data
   struct node *next; // Address
}*head;
```

/* Insertion of a new node at the beginning of Single linked list */

Scanned by CamScanner

```
void insertNodeAtBeginning(int data)
{
  struct node *newNode;
  newNode = (struct node*)malloc(sizeof(struct node));
  if(newNode == NULL)
  {
    printf("Unable to allocate memory.");
  }
  else
  {
    newNode->data = data;          // Link data part
    newNode->next = head;          // Link address part
    head = newNode;                // Make newNode as first node
    printf("New node added successfully.\n");
  }
}
```

Q. 4. (b) Write a function to delete a note from the end of double linked list. (8)

Ans. /* Basic structure of Node */

```
struct node {
  int data;
  struct node * prev;
  struct node * next;
}*head, *last;
/* Function to delete the last node of the doubly linked list */
void deleteFromEnd()
{
  struct node *toDelete;
  if(last == NULL)
  {
    printf("Unable to delete. List is empty.\n");
  }
  else
  {
    toDelete = last;
    last = last->prev; // Move last pointer to 2nd last node
    if (last != NULL)
      last->next = NULL; // Remove link to of 2nd last node with last node
    free(toDelete);     // Delete the last node
    printf("SUCCESSFULLY DELETED NODE FROM END OF THE LIST.\n");
  }
}
```

Q. 5. (a) A binary tree T has 09 nodes. The inorder and preorder traversals of T yield the following sequences of nodes. (4.5)

Inorder: D G B A H E I C F

Preorder: A B D G C E H I F.

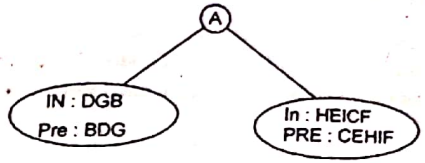
Draw the tree T

Ans. Given Traversals of T

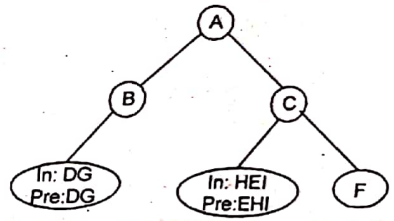
Inorder: DGBAHEICF

Preorder: ABDGCEHIF

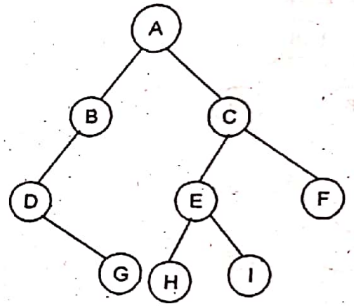
Step 1:



Step 2:

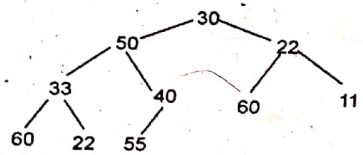


Step 3:



Finally constructed Binary Tree.

Q. 5. (b) Consider the following binary tree T with N = 10 nodes. What is the inorder traversal of the tree? (8)



Ans. In order Traversal of the given tree with 10 nodes:

Inorder Traversal (Left – Root–Right):

60, 33, 22, 50, 55, 40, 30, 60, 22, 11

UNIT-III

Q. 6. (a) Construct B-tree of order 3 by inserting the following keys in the order shown. 18, 19, 6, 10, 40, 45, 5, 8. (4.5)

Ans. We need to create a B-Tree of order 3 with key values.

18, 19, 6, 10, 40, 45, 5, 8

Scanned by CamScanner

Minimum no. of children for a non-leaf node = $\frac{n}{2} = \frac{3}{2} = 2$

Maximum no. of children for a non-leaf node = n = 3.

Minimum and maximum no. of values in a non-leaf node m= 1 (2 – 1) and 2 (3 – 1) respectively

Minimum and Maximum no. of children for a root node = 2 and 3 respectively

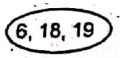
Minimum and Maximum no. of values in root node = 1 and 2 respectively.

We need to split a node if it contains more then maximum no of values allowed from $\frac{n}{2} = \frac{3}{2} = 2nd$ value.

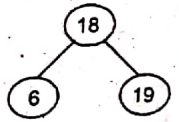
Step 1: Insert 18

Step 2: Insert 19

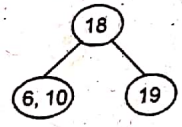
Step 3: Insert 6



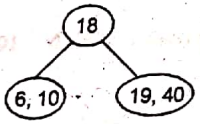
Now, the root node contains more values than maximum no. of values, So, split the root node from 2nd values as shown



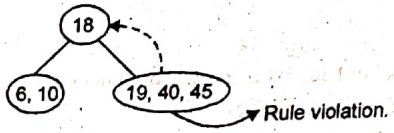
Step 4: Insert 10



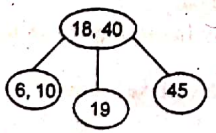
Step 5: Inset 40



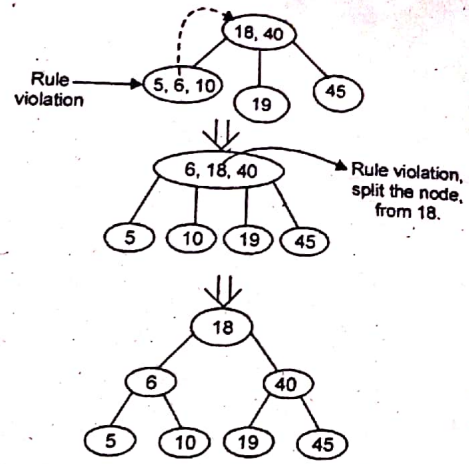
Step 6: Insert 45



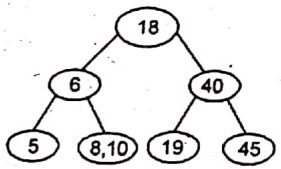
So, split the node from 2nd value



Step 7: Insert 5



Step 8: Insert 8



Final B-Tree.

Q. 6. (b) Construct Binary search Tree of the following keys in the order shown. **(8)**

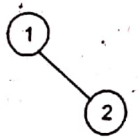
1, 2, 3, 15, 8, 25, 7, 9

Ans. Given list is

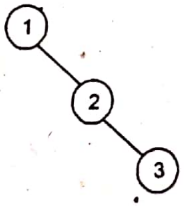
1, 2, 3, 15, 8, 25, 7, 9,

Step 1: Insert 1

Step 2: Insert 2

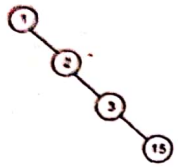


Step 3: Insert 3:

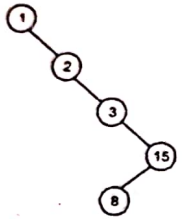


Scanned by CamScanner

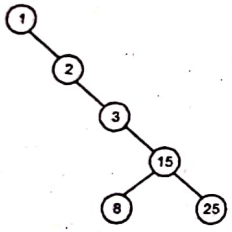
Step 4: Insert 15



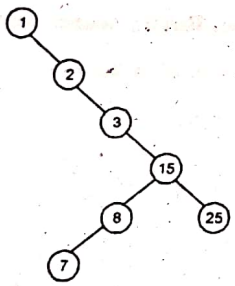
Step 5: Insert 8



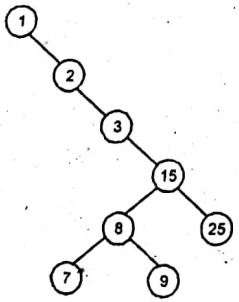
Step 6: Insert 25



Step 7: Insert 7



Step 8: Insert 9



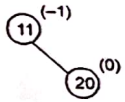
Q. 7. (a) Construct an AVL search tree of the following values: 11, 20, 23, 5, 3, 7, 9, 6, **(4.5)**

Ans. Given list of values to AVL Tree.

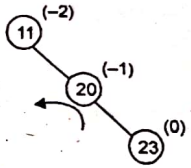
Creation: 11, 20, 23, 5, 3, 7, 9, 6

Step 1: Insert 11

Step 2: Insert 20

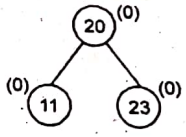


Step 3: Insert 23

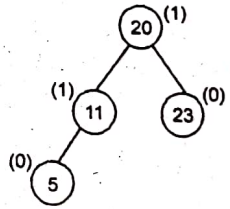


(Right unbalanced Tree)

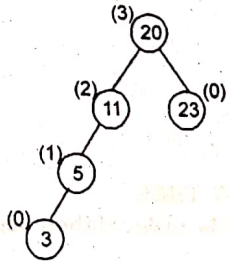
So, we need to use left rotation to balance the above tree.



Step 4: Insert 5

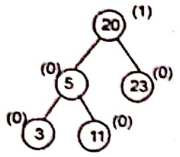


Step 5: Insert 3

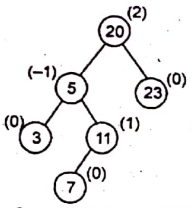


Scanned by CamScanner

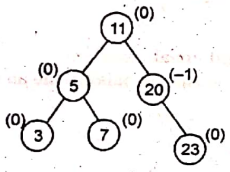
Now, we need to use right rotation to balance the above tree.



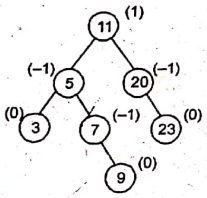
Step 6: Insert 7



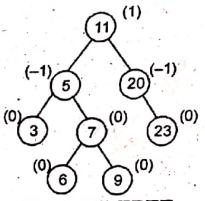
Now, double rotation required after which tree will be



Step 7: Insert 9

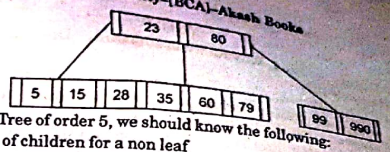


Step 8: Insert 6



Final AVL TREE

Q. 7. (b) Insert the following values in the order of their occurrence 32, 34 in the given B tree of order 5. **(8)**



Ans. For a B-Tree of order 5, we should know the following:

Minimum no. of children for a non leaf

$$\text{node} = \frac{n}{2} = \frac{5}{2} = 3$$

Maximum no. of children for a non-leaf node = n = 5

Minimum and maximum no. of values in a non-leaf node

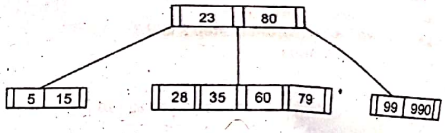
= 2(3 – 1) and 4 (5 – 1) respectively.

Minimum and maximum no. of children for a root node = 2 and 5 respectively.

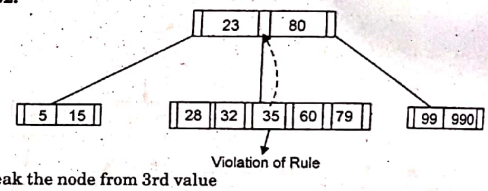
Minimum and maximum no. of values in a root node = 1 and 4 respectively.

We need to split a node if it contains more than maximum no. of value allowed, from the $\frac{n}{2} = \frac{5}{2}$ = 3rd value in the node by moving 3rd value to the parent node.

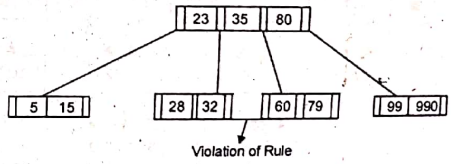
Given B-Tree



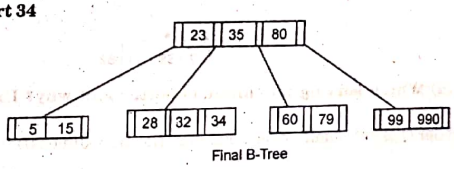
Insert 32:



Now, Break the node from 3rd value



Now, **Insert 34**



Final B-Tree

Scanned by CamScanner

UNIT-IV

Q. 8. (a) Define hashing. Why do we use hashing? Discuss any two hashing methods with example. **(4.5)**

Ans. Refer Q.8.(a) from EndTerm Examination May-June 2017.

Q. 8. (b) Which searching technique is best and under what conditions? Justify your answer with the help of an example. **(8)**

Ans. Refer Q. 8. (b) from EndTerm Examination May-June 2017.

Q. 9. (a) Explain merge sort with example. **(4.5)**

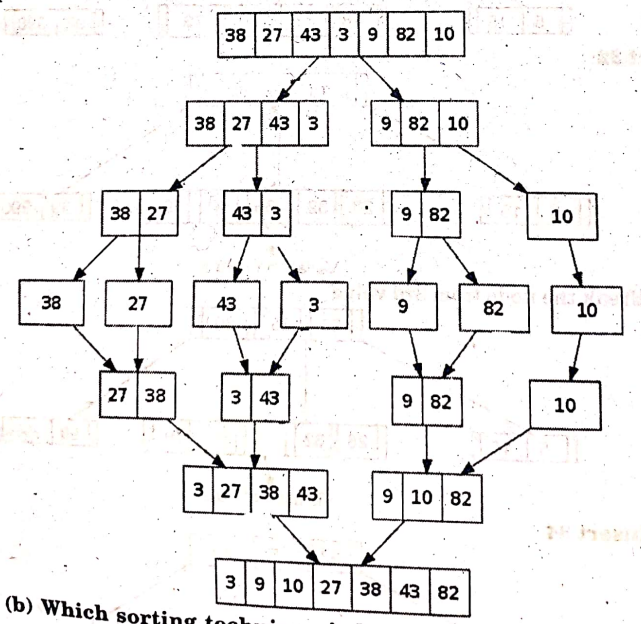
Ans. Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:
   Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
   Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
   Call merge(arr, l, m, r)

Example:



Q. 9. (b) Which sorting technique is better and why? Explain with an example. **(8)**

Ans. Refer Q. 9. (b) from EndTerm Examination May-June 2017.

Scanned by CamScanner