Solution for Internal Assessment Test III – Nov 2019

Sub:	Data Structures & Algorithms					Sub Code:	18CS32	Branch :	CSE
Date:	19/11/2019	Duration:	90 min's	Max Marks:	50	Sem / Sec:	3 'A','B' & 'C'		

Answer any FIVE FULL Questions

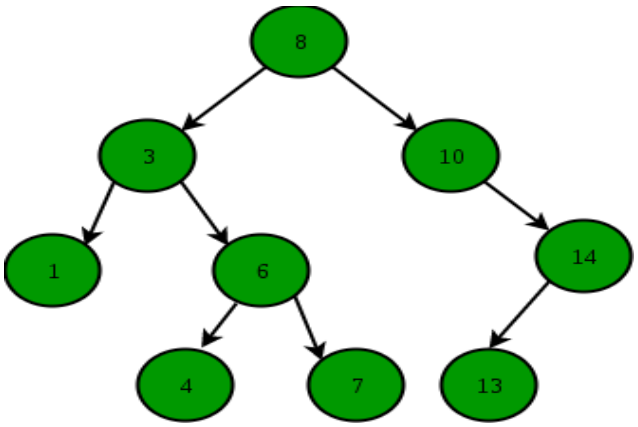
MARKS

1 Define Binary Search tree. Write the recursive search and iterative search algorithm for a binary search tree. [10]

Solution:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left sub tree of a node contains only nodes with keys lesser than the node's key.
- The right sub tree of a node contains only nodes with keys greater than the node's key.
- The left and right sub tree each must also be a binary search tree.



The recursive search and iterative search algorithm for a binary search tree is as follows:

Algorithm for recursive search in a Binary Search Tree:

Input : A pointer to BST & skey(Binary Search Tree with data and left and right pointer & search key value)

Output: found / not-found flag counting number of hits/comparisons/for locations

1. Hits=0
2. found = false
3. function searchbst(BST *p,)

```
if p=NULL then
return found // false
        endif
hits=hits+1
If skey = p->data then
found = true
return found // true
        else if skey < p->data then
searchbst(p->left) // recursion to search left sub-tree
                else
searchbst(p->right) // recursion to search right sub-tree
Endif
```

4. End function searchbst

Display the found status and count of hits

Iterative search for binary search tree:

```
element* iterSearch(treePointer tree, int k)
{
   /* return a pointer to the element whose key is k, if there is no such
   element, return NULL. */
```

```
    while (tree)
    {
        If(k==tree->datakey)
                return&(tree->data);
        If(k<tree->datakey)
              tree=tree->leftChild;
       else
            tree=tree->rightChild;
    }
    return NULL;
}
```

Note: If h is the height of the binary search tree, iterSearch: O(h)However, search has an additional stack space requirement which is O(h).

(OR)

Write Recursive functions/algorithm for In-order, Pre-order & Post-order traversals of a Binary tree.

Solution:

Inorder Traversal :

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Preorder Traversal :

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left sub tree, i.e., call Postorder(left-subtree)
2. Traverse the right sub tree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Post order

Post order traversal is used to delete the tree. Post order traversal is also useful to get the postfix expression of an expression tree.

Functions:

/* Given a binary tree, print its nodes according to the "bottom-up" **Postorder traversal.** */

```
void printPostorder(struct node* node)
{
    if (node == NULL)
       return;

     printPostorder(node->left);    // first recur on left subtree

    printPostorder(node->right);       // then recur on right subtree
```

```
    printf("%d ", node->data);  // now deal with the node
}

/* Given a binary tree, print its nodes in inorder */

void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    printInorder(node->left);       /* first recur on left child */

    printf("%d ", node->data);         /* then print the data of node */

    printInorder(node->right);         /* now recur on right child */

}

/* Given a binary tree, print its nodes in preorder*/

void printPreorder(struct node* node)
{
    if (node == NULL)
        return;

        printf("%d ", node->data);   /* first print data of node */

        printPreorder(node->left);        /* then recur on left sutree */

        printPreorder(node->right);    /* now recur on right subtree */
}
```

2 Demonstrate insertion sort and radix sort techniques for the following elements: 77, 33,44,11,88,22,66,55. [10]

Solution:

Insertion Sort:

Insertion sort always maintains two zones in the array to be sorted: sorted and unsorted. At the beginning the sorted zone consists of one element. On each step the algorithms expand it by one element inserting the first element from the unsorted zone in the proper place in the sorted zone and shifting all larger elements one slot down.

- **Step 1 -** Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Input: 77, 33,44,11,88,22,66,55.

First iteration:

Sorted	Unsorted
77	33,44,11,88,22,66,55.

Second iteration:

Sorted	Unsorted
33, 77	44, 11, 88,22,66,55.

Third iteration:

Sorted	Unsorted
33, 44, 77	11, 88,22,66,55.

Fourth iteration:

Sorted	Unsorted
11, 33, 44, 77	88,22,66,55.

Fifth iteration:

Sorted	Unsorted
11, 33, 44, 77, 88	22, 66, 55.

Sixth iteration:

Sorted	Unsorted
11, 22, 33, 44, 77, 88	66, 55.

Seventh iteration:

Sorted	Unsorted
11, 22, 33, 44, 66, 77, 88	55.

Eight iteration:

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

Radix Sort:

Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.

Original, unsorted list:

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

Sorting by least significant digit (1s place) gives:

7<u>7</u>, 3<u>3</u>, 4<u>4</u>, 1<u>1</u>, 8<u>8</u>, 2<u>2</u>, 6<u>6</u>, 5<u>5</u>

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

Sorting by most significant digit (10s place) gives:

<u>1</u>1, <u>2</u>2 , <u>3</u>3 , <u>4</u>4 , <u>5</u>5 , <u>6</u>6 , <u>7</u>7 , <u>8</u>8

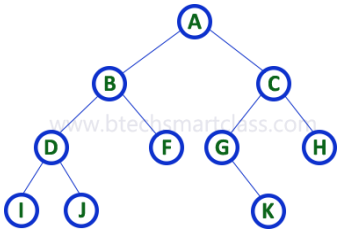
11, 22 , 33 , 44 , 55 , 66 , 77 , 88

3 Define binary tree. Construct an Expression Tree for the given expression ((6+ (3-2)*5)^2+3). [10]

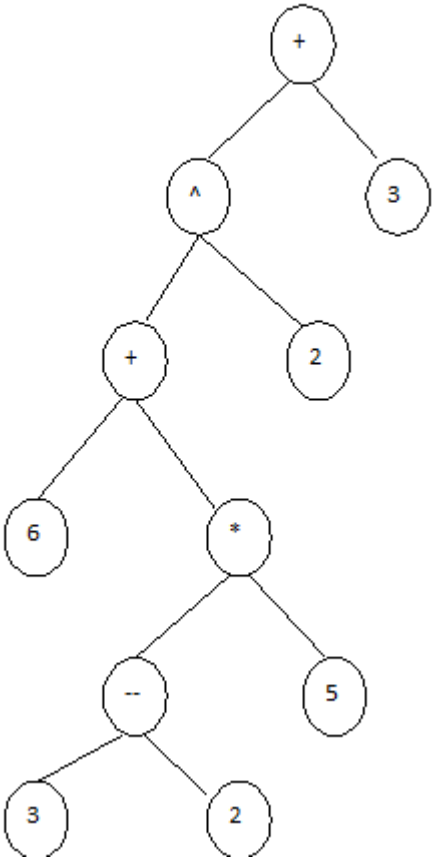
Solution:

A tree in which every node can have a maximum of two children is called as Binary Tree.In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



Expression Tree for the given expression ((6+ (3-2)*5)^2+3).



4 Construct a binary tree from the traversal order given below and also find the post order traversal. [10]

Pre order: A B D E F C G H L J K In order: D B F E A G C L J H K

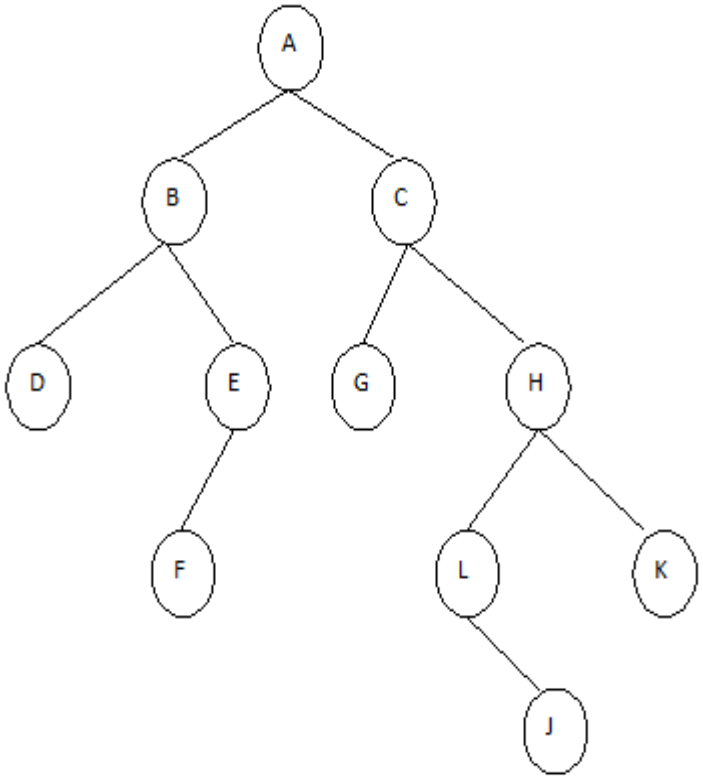
Solution:

Step 1:

Read the first element from Pre order traversal which is root node. Now, we come to know which is the root node.

Step 2:

Search the same element in In-order traversal, when the element is found in In-order traversal, then we can very well say that all the elements present before the node found are Left children/sub-tree and all the elements present after the node found are Right children/sub-tree of node found. (as in In-order traversal Left child's are read first and then root node and then right nodes.)



Post order :DFEBGJLKHCA

5 Write Program/functions to illustrate “copying of binary trees” and “testing equality of binary trees”. [10]

Solution:

First defining structure:

```
typedef struct TREE
{
        int data;
        struct TREE *left;
        struct TREE *right;
}TREE;
```

(i) Routine for Copying Binary Tree

```
void t_cpy(NODE *t1,NODE *t2)
{
  int
  val,op
  t=0;
  NOD
  E
  *temp
  ;
  if(t1==NULL || t2==NULL)
  {
    printf("Cannot copy !\n");
  }
  inorder(t1);

  printf("\nEnter the node value where tree 2 should be
  copied\n"); scanf("%d",&val);
  temp=t1;
  while(temp
  !=NULL)
  {
    if(val<temp-
      >data)
      temp=tem
      p->llink;
    else
      temp=temp->rlink;
  }
  if(temp->llink!=NULL || temp-
    >rlink!=NULL) printf("Not possible to
    copy tree to this node\n");
  else
  {
    printf("Copy tree to \n 1.Left Node \n 2.Right Node\n Enter
```

```
    your choice : "); scanf("%d",&opt);
    if(opt==1)
    {
      temp->llink=t2;
    }
    else if(opt==2)
    {
      temp->rlink=t2;
    }
    else
      printf("Invalid choice\n");
  }
  printf("Tree1 after
  copying is\n");
  inorder(temp);
}
```

(ii)Routine for testing equality of Binary Trees

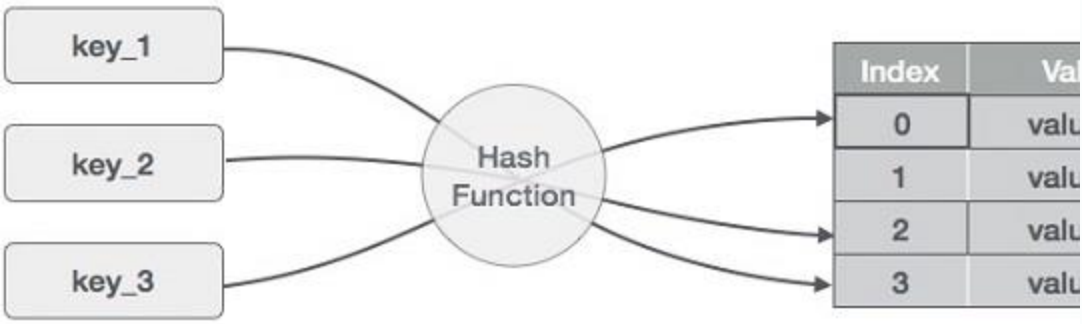
```
/* Two binary trees are equal if their topologies are same and
corresponding nodes contain identical values */

int compare( TREE *p1, TREE *p2)
 {
        While(p1 || p2)          /* while both are not NULL */
        {
                If((p1 && !p2) || (p2 && !p1) /* if one of the tree points to NULL
                & other doesn't */
                        return (0);       /* returns false */
                If( p1->data != p2->data)
                        return (0);       /* returns false */ p1=p1->next;
                p2=p2->next;
        }
        return (1);       /* returns true if not returned by while loop */
  }
```

6 Explain hashing and collision. What are the methods used to resolve collision. [10]

Solution:

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Problem with Hashing –collision

The method discussed above seems too good to be true as we begin to think more about the hash function. First of all, the hash function we used, that is the sum of the letters, is a bad one. In case we have permutations of the same letters, “abc”, “bac” etc in the set, we will end up with the same value for the sum and hence the key. In this case, the strings would hash into the same location, creating what we call a “collision”. This is obviously not a good thing. Secondly, we need to find a good table size, preferably a prime number so that even if the sums are different, then collisions can be avoided, when we take mod of the sum to find the location. So we ask two questions.

Question 1: How do we pick a good hash function?

Question 2: How do we deal with collisions?

The problem of storing and retrieving data in O(1) time comes down to answering the above questions. Picking a “good” hash function is key to successfully implementing a hash table. What we mean by “good” is that the function must be easy to compute and avoid collisions as much as possible. If

the function is hard to compute, then we lose the advantage gained for lookups in O(1). Even if we pick a very good hash function, we still will have to deal with “some” collisions.

The process where two records can hash into the same location is called collision. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location. This is called collision resolution.

Collision Resolution Strategies

1. Open Addressing/Closed Hashing
2. Chaining

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position

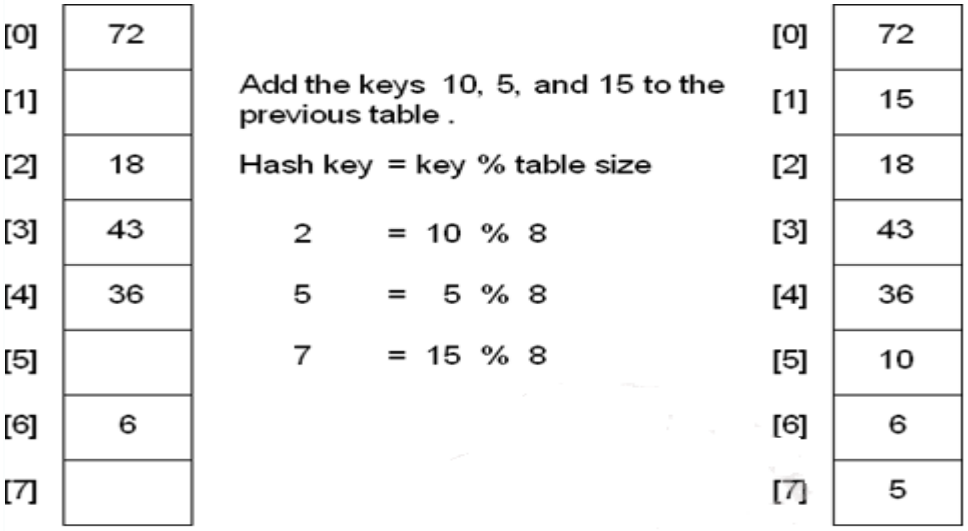
The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing.

Linear Probing

When using a linear probe to resolve collision, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.If an empty slot is not found before reaching the point of collision, the table is full.

If h is the point of collision, probe through h+1,h+2,h+3. ..... h+i. till an empty slot is
found



Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as O(1). If the key does not match, then the search function begins a sequential search of the array that continues until the value is found, or the search function encounters a vacant location in the array, indicating that the value is not present, or the search function terminates because it reaches the end of the table and the value is not present.

Probe Sequence ::(h+i)%Table size

Disadvantage:

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. It is possible for

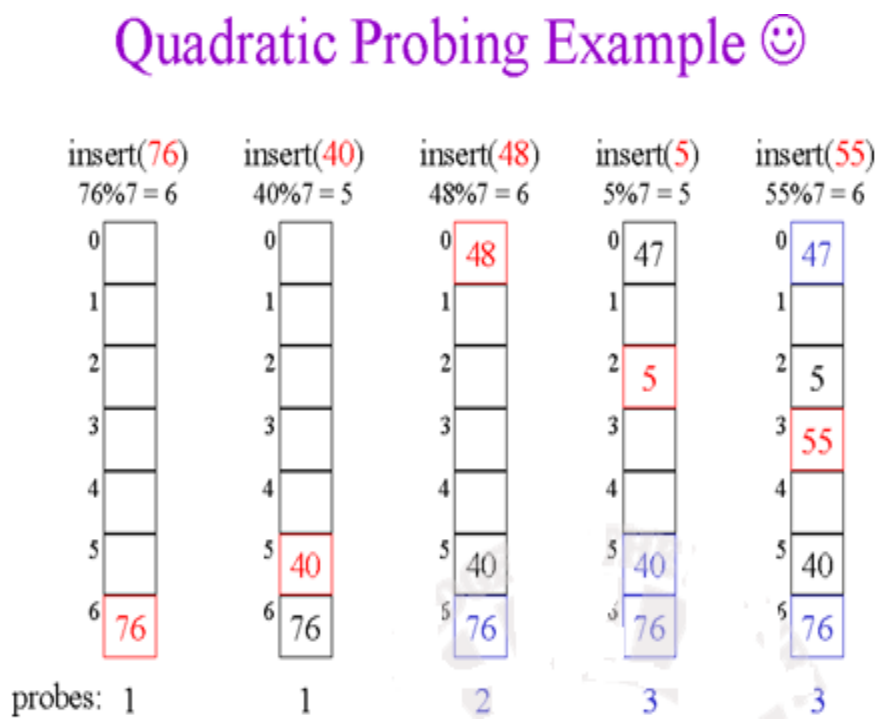
blocks of data to form when collisions are resolved. This means that any key that hashes into the cluster will require several attempts to resolve the collision. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called clustering. To avoid clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, if the first hash value that has resulted in collision is *h*, the successive values which are probed are *h*+1, *h*+4, *h*+9, *h*+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

Probe sequence :h,h+1^2,h=2^2,h=3^2 ····· h+i^2

H(k)=(h+$i^{2)}$%Tablesize

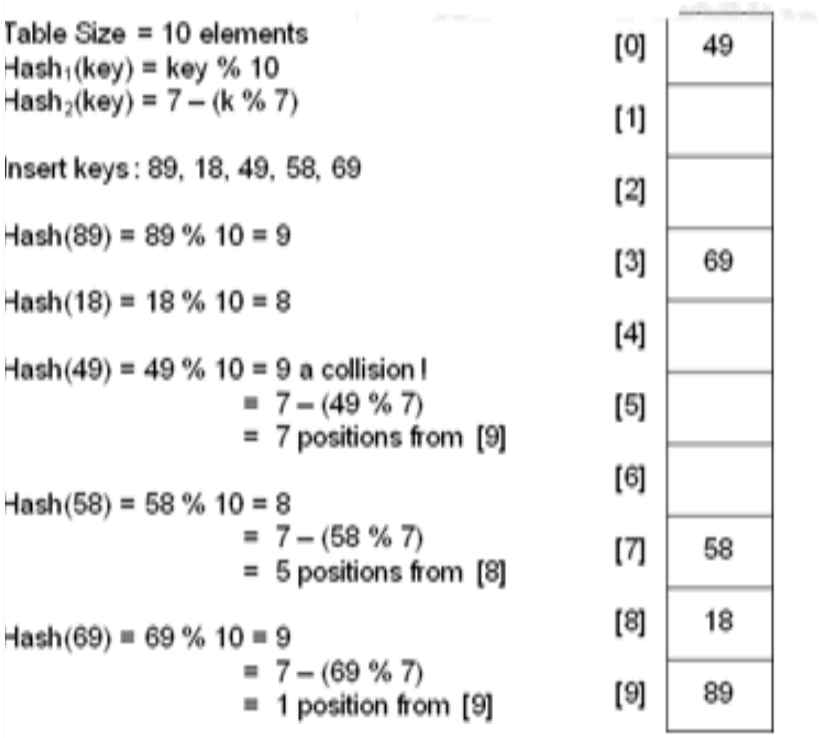


Double Hashing

In double hashing, we use two hash functions rather than a single function. Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.There are a couple of requirements for the second function:

- It must never evaluate to 0
- Must make sure that all cells can be probed

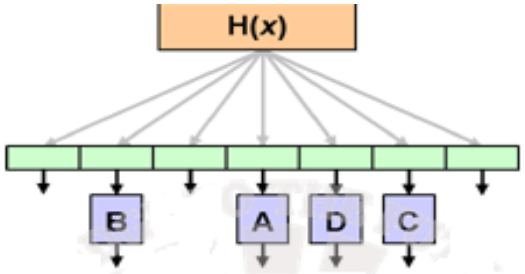
A popular second hash function is: $Hash_2(key) = R - (key \% R)$ where R is a prime number that is smaller than the size of the table.But any independent hash function may also be used.



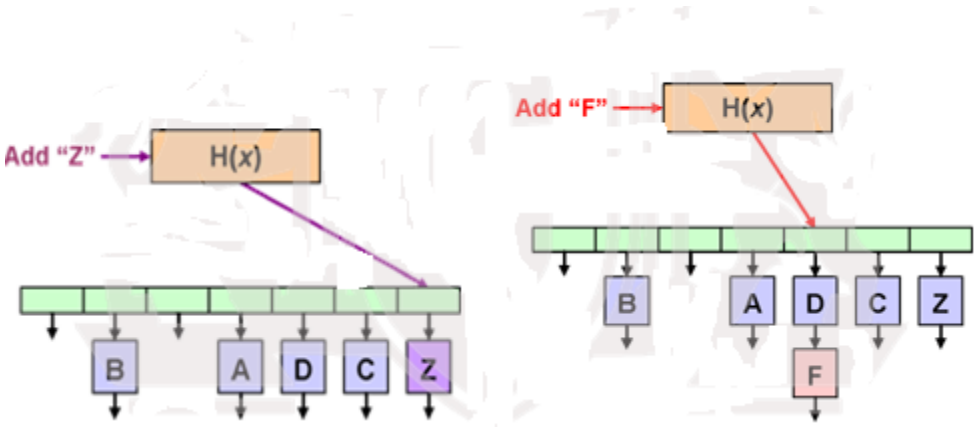
Double hashing minimizes repeated collisions and the effects of clustering.

Chaining

Chaining is another approach to implementing a hash table; instead of storing the data directly inside the structure, have a linked list structure at each hash element. That way, all the collision, retrieval and deletion functions can be handled by the list, and the hash function's role is limited mainly to that of a guide to the algorithms, as to which hash element's list to operate on.



The linked list at each hash element is often called a chain. A *chaining* hash table gets its name because of the linked list at each element -- each list looks like a 'chain' of data strung together. Operations on the data structure are made far simpler, as all of the data storage issues are handled by the list at the hash element, and not the hash table structure itself.



Chaining overcomes collision but increases search time when the length of the overflow chain increases

7 Write a C program to implement the DFS & BFS graph traversal methods. [10]

Solution:

```
/* C program to implement BFS(breadth-first search) and DFS(depth-first search)
algorithm */

#include<stdio.h>
int q[20],top=-1,front=-1,rear=-1,a[20][20],vis[20],stack[20];
int delete();
void add(int item);
void bfs(int s,int n);
void dfs(int s,int n);
void push(int item);
int pop();

void main()
{
      int n,i,s,ch,j;
      char c,dummy;
      printf("ENTER THE NUMBER VERTICES ");
      scanf("%d",&n);
    for(i=1;i<=n;i++)
   {
         for(j=1;j<=n;j++)
        {
           printf("ENTER 1 IF %d HAS A NODE WITH %d ELSE 0 ",i,j);
           scanf("%d",&a[i][j]);
      }
  }
   printf("THE ADJACENCY MATRIX IS\n");
```

```
    for(i=1;i<=n;i++)
   {
          for(j=1;j<=n;j++)
          {
              printf(" %d",a[i][j]);
          }
             printf("\n");
}

do
{
          for(i=1;i<=n;i++)
          vis[i]=0;
          printf("\nMENU");
          printf("\n1.B.F.S");
          printf("\n2.D.F.S");
          printf("\nENTER YOUR CHOICE");
          scanf("%d",&ch);
         printf("ENTER THE SOURCE VERTEX :");
          scanf("%d",&s);
        switch(ch)
       {
          case 1:bfs(s,n);
         break;
        case 2:
        dfs(s,n);
          break;
       }
          printf("DO U WANT TO CONTINUE(Y/N) ? ");
          scanf("%c",&dummy);
         scanf("%c",&c);
         }
        while((c=='y')||(c=='Y'));
```

```
}
```

```
//***************BFS(breadth-first search) code**************//
void bfs(int s,int n)
{
          int p,i;
          add(s);
          vis[s]=1;
         p=delete();
         if(p!=0)
         printf(" %d",p);
          while(p!=0)
        {
          for(i=1;i<=n;i++)
         if((a[p][i]!=0)&&(vis[i]==0))
        {
             add(i);
           vis[i]=1;
          }
         p=delete();
        if(p!=0)
       printf(" %d ",p);
 }
for(i=1;i<=n;i++)
  if(vis[i]==0)
   bfs(i,n);
}

void add(int item)
```

```
{
    if(rear==19)
    printf("QUEUE FULL");
     else
    {
        if(rear==-1)
      {
          q[++rear]=item;
          front++;
       }
       else
      q[++rear]=item;
   }
}

int delete()
{
       int k;
       if((front>rear)||(front==-1))
       return(0);
       else
      {
              k=q[front++];
              return(k);
       }
}

//***************DFS(depth-first search) code*****************//
void dfs(int s,int n)
{
      int i,k;
      push(s);
      vis[s]=1;
```

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

```
            k=stack[top--];
            return(k);
        }
}
```

/* Output of BFS(breadth-first search) and DFS(depth-first search) program */

```
ENTER THE NUMBER VERTICES 3
ENTER 1 IF 1 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 1 HAS A NODE WITH 3 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 1 ELSE 0 1
ENTER 1 IF 2 HAS A NODE WITH 2 ELSE 0 0
ENTER 1 IF 2 HAS A NODE WITH 3 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 1 ELSE 0 0
ENTER 1 IF 3 HAS A NODE WITH 2 ELSE 0 1
ENTER 1 IF 3 HAS A NODE WITH 3 ELSE 0 1
THE ADJACENCY MATRIX IS
 1 1 0
 1 0 1
 0 1 1
```

```
MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE 1
ENTER THE SOURCE VERTEX : 2
 2 1  3 DO U WANT TO CONTINUE(Y/N) ? y

MENU
1.B.F.S
2.D.F.S
ENTER YOUR CHOICE2
ENTER THE SOURCE VERTEX :2
 2  3  1 DO U WANT TO CONTINUE(Y/N) ?
```

8 Define a file & basic operations of a file. Briefly Explain Serial, Sequential, Relative and Indexed file organization with its advantages & disadvantages. [10]

Solution:

File: A file is a collection of related records.

Example: A file of all the employees working in an organization.

BASIC FILE OPERATIONS

The basic operations that can be performed on a file are given in below figure

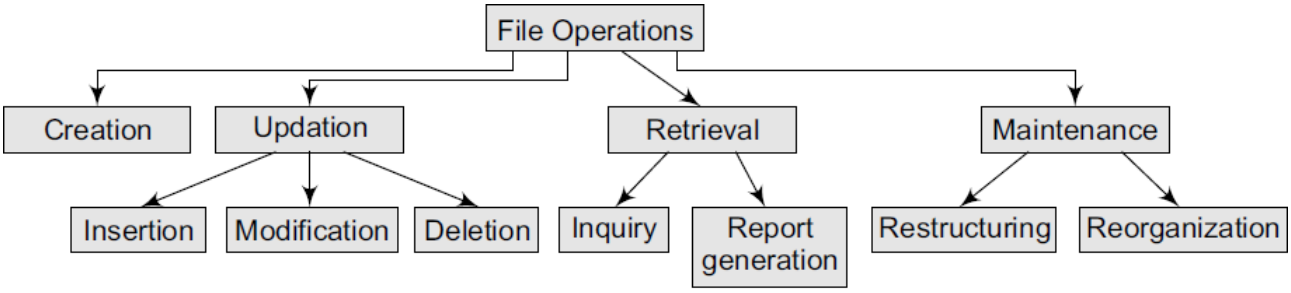


Figure File operations

Creating a File

A file is created by specifying its name and mode. Then the file is opened for writing records that are read from an input device. Once all the records have been written into the file, the file is closed. The file is now available for future read/write operations by any program that has been designed to use it in some way or the other.

Updating a File

Updating a file means changing the contents of the file to reflect a current picture of reality. A file can be updated in the following ways:

- Inserting a new record in the file. For example, if a new student joins the course, we need to add his record to the STUDENT file.
- Deleting an existing record. For example, if a student quits a course in the middle of the session, his record has to be deleted from the STUDENT file.
- Modifying an existing record. For example, if the name of a student was spelt incorrectly, then correcting the name will be a modification of the existing record.

Retrieving from a File

It means extracting useful data from a given file. Information can be retrieved from a file either for an inquiry or for report generation. An inquiry for some data retrieves low volume of data, while report generation may retrieve a large volume of data from the file.

Maintaining a File

It involves restructuring or re-organizing the file to improve the performance of the programs that access this file.

Restructuring a file keeps the file organization unchanged and changes only the structural aspects of the file.

Example: changing the field width or adding/deleting fields.

File reorganization may involve changing the entire organization of the file

FILE ORGANIZATION

Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media.

The following considerations should be kept in mind before selecting an appropriate file organization method:

- Rapid access to one or more records
- Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record
- Efficient storage of records
- Using redundancy to ensure data integrity

1. Sequential Organization

A sequentially organized file stores the records in the order in which they were entered.

Sequential files can be read only sequentially, starting with the first record in the file.

Sequential file organization is the most basic way to organize a large collection of records in a file

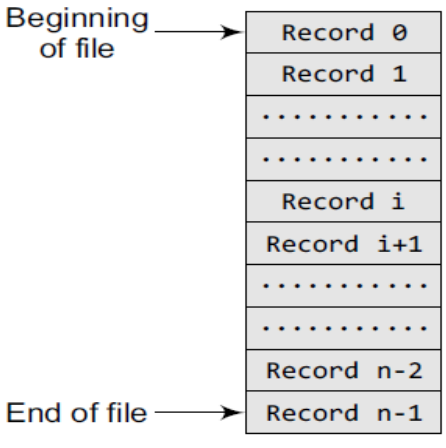


Figure Sequential file organization

Features

- Records are written in the order in which they are entered
- Records are read and written sequentially
- Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes
- Records have the same size and the same field format
- Records are sorted on a key value
- Generally used for report generation or sequential reading

Advantages

- Simple and easy to Handle
- No extra overheads involved
- Sequential files can be stored on magnetic disks as well as magnetic tapes
- Well suited for batch– oriented applications

Disadvantages

- Records can be read only sequentially. If i^{th} record has to be read, then all the i–1 records must be read
- Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes
- Cannot be used for interactive applications

2. Relative File Organization

Figure shows a schematic representation of a relative file which has been allocated space to store

Relative record number	Records stored in memory
0	Record 0
1	Record 1
2	FREE
3	FREE
4	Record 4
.....	.....
98	FREE
99	Record 99

Figure Relative file organization

100 records

If the records are of fixed length and we know the base address of the file and the length of the record, then any record i can be accessed using the following formula:

$$\text{Address of } i^{th} \text{ record} = \text{base\_address} + (i-1) * \text{record\_length}$$

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given as:

$$1000 + (5-1) * 20 \quad = 1000 + 80 \quad = 1080$$

Features

- Provides an effective way to access individual records
- The record number represents the location of the record relative to the beginning of the file
- Records in a relative file are of fixed length
- Relative files can be used for both random as well as sequential access
- Every location in the table either stores a record or is marked as FREE

Advantages

- Ease of processing
- If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously
- Random access of records makes access to relative files fast
- Allows deletions and updations in the same file
- Provides random as well as sequential access of records with low overhead
- New records can be easily added in the free locations based on the relative record number of the record to be inserted
- Well suited for interactive applications

Disadvantages

- Use of relative files is restricted to disk devices
- Records can be of fixed length only
- For random access of records, the relative record number must be known in advance

3. Indexed Sequential File Organization

The index sequential file organization can be visualized as shown in figure

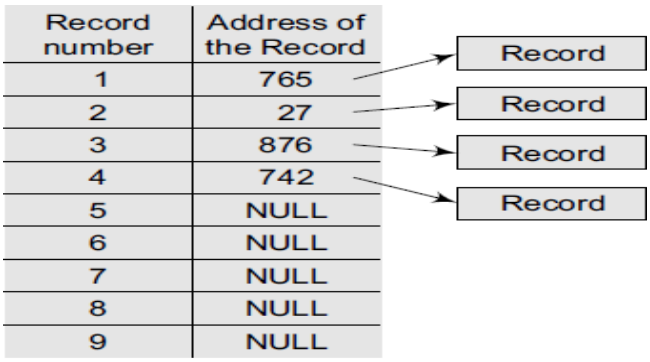


Figure Indexed sequential file organization

Features

- Provides fast data retrieval
- Records are of fixed length
- Index table stores the address of the records in the file
- The ith entry in the index table points to the ith record of the file
- While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly
- Indexed sequential files perform well in situations where sequential access as well as random access is made to the data

Advantages

- The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs
- Supports applications that require both batch and interactive processing
- Records can be accessed sequentially as well as randomly
- Updates the records in the same file

Disadvantages

- Indexed sequential files can be stored only on disks
- Needs extra space and overhead to store indices
- Handling these files is more complicated than handling sequential files
- Supports only fixed length records