Experiment No: 1

Aim: - To implement functions of Dictionary using Hashing (division method, Multiplication method, Universal hashing)

Description: -

A *dictionary* is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. *hash tables* provide an efficient implicit realization of a dictionary.

Division Method

- A key is mapped into one of h slots using the function

$$H(k) = k \bmod h$$

  - h should not be a power of 2, since if $h = 2^p$, then $h(k)$ is just the p lowest order bits of k
  - Good values for h primes not too close to exact powers of 2.

Multiplication Method

There are two steps:

1. Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA
2. Multiply this fractional part by h and take the floor.

$$H(k) = \lfloor h(kA \bmod 1) \rfloor \qquad A \approx (\sqrt{5} - 1)/2$$

Universal Hashing

This involves choosing a hash function randomly in a way that is independent of the keys that are actually going to be stored. We select the hash function at random from a carefully designed class of functions.

Algorithm:

h : the size of hash table

index : the index value of hash table returned by functions

A : is constant whose value is ≈ (√5 - 1)/2

H[]: the array which indicates the hash table

DIVISION_HASH_FUNCTION(K)

1. index = K mod h
2. Return index

MULTIPLICATION_HASH_FUNCTION (K)

1. j = K*A mod 1
2. index = floor(h * j)
3. Return (index)

INSERT_ HASH_TABLE(K)

1. flag = false
2. Index = HASH_FUNCTION(K)
3. If(K = H[index]) then

   3.1.Print " the element exist in the table"

   3.2.Exit
4. Else

4.1.i = index + 1 mod h

4.2.while (i ≠ index) and (not flag) do

4.2.1. if (H[i] = NULL) or (H[i] < 0) then

4.2.1.1.H[i] = K

4.2.1.2.flag = true

4.2.2. Else

4.2.2.1.if (H[i] = K) then

4.2.2.1.1. flag = true

4.2.2.1.2. Exit

4.2.2.2.Else

4.2.2.2.1. i = i mod h

4.2.2.3.End If

4.2.3. End if

4.3.End while

4.4.If (flag = false) and (i = index) then

4.4.1. Print " the table is overflow"

4.4.2. Exit

4.5.End if

5. End if
6. Stop

DELETE_ HASH_TABLE(K)

1. index = HASH_FUNCTION(K)
2. If(K = H[index]) then

   2.1.H[K] = NULL

   2.2.Exit
3. Else

   3.1.i = index + 1 mod h

   3.2.while (i ≠ index) and H[i] ≠ NULL do

3.2.1. if (H[i] = K) then

3.2.1.1.H[i] = NULL

3.2.1.2.exit

3.2.2. Else

3.2.2.1.1. i = i + 1

3.2.2.1.2. i = i mod h

3.2.3. End if

3.3.End while

4. End if
5. Stop

SEARCH_HASH(K)

1. hashval = HASH_FUNCTION(K)
2. while (H[hashval] ≠ null) do

2.1.if (H[hashval] = K) then

2.1.1. Return H[hashval]

2.2.End if

2.3.hashval = hashval + 1

2.4.hashval = hashval % h

3. End while
4. Return null
5. Stop

Sample INPUT:

Enter maximum size of hash table: 13

1. Insert
2. Delete
3. Search
4. Exit

Enter your choice : 1

The elements inserted in the order are

12 , 17, 21, 28, 35, 40, 43, 52, 66, 69

Enter your choice : 3

Enter element to be searched :

43

Enter your choice : 2

Enter element to be deleted :

35

Expected output:

<< Table >>

0 : 52

1 : 40

2 : 28

3 : 66

4 : 17

5 : 43

6 : 69

8 : 21

9 : 35

12 : 12

43 not found

35 deleted

Conclusion: Student get the knowledge on creating the Dictionary which incorporate to develop the Real World Applications like GPS and WPS .it attains the PO1,PO2, PO3, PSO1,PSO2 and CO1

Viva Questions:

1. What is hashing?
2. What is hash function?
3. Define Collision?

4. What is a good hash function?
5. What are the different collision resolution techniques?
6. What are the different closed hashing techniques?
7. Define linear probing?
8. Define quadratic probing?
9. Define separate chaining?
10. Define universal hashing?
11. What are the drawbacks in separate chaining?
12. What is dictionary?
13. What are the applications of dictionary?
14. What are the drawbacks in linear probing?
15. Define load factor?

Experiment No: 2

Aim: - To perform various operations i.e, insertions and deletions on AVL trees

Description: -

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. The worst case height of an AVL tree with *n* nodes is 1.44log *n*.

Definition of an AVL tree

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

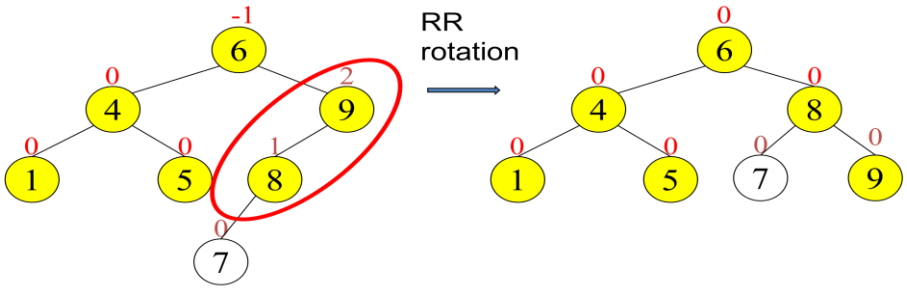
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Thus a balanced binary tree (AVL tree) is a Binary Tree in which heights of the two subtrees of every mode never differ by more than 1. The balance of a node in a Binary tree is defined as the height of its left subtree minus height of its right subtree. Each node in a balanced Binary Tree has a balance of 1, -1, or 0, depending on whether the height of its left subtree is greater than, less than, or equal to the height of its right subtree.

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

Insertion

Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the balance factor to each node. This factor indicates whether the tree is left heavy (the height of the left sub-tree is 1 greater than the right sub-tree), balanced (both sub-trees are the same height) or right heavy (the height of the right subtree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.AVL trees which remain balanced - and thus guarantee O(logn) search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in O(logn) time.



Algorithm:

COMPUTE_HEIGHT (ptr)

1. If ptr = NULL then // height of empty tree is zero
   1.1.height = 0
   1.2.Return (height)
2. Else
   2.1.Lptr = ptr.lchild
   2.2.Rptr = ptr.rchild
   2.3.H_L = COMPUTE_HEIGHT(Lptr)
   2.4.H_R = COMPUTE_HEIGHT(Rptr)
   2.5.If $H_L \leq H_R$ then // maximum of left sub tree and right sub tree
      2.5.1. height = 1 + H_R
   2.6.Else // $H_L > H_R$
      2.6.1. height = 1 + H_L
   2.7.End if
3. Return (height) // return height of the tree
4. End if
5. Stop

ROTATION_LEFT_TO_LEFT (Pptr)

1. Aptr = Pptr.lchild // initialize the pointer to left child of pivot node P
2. Pptr.lchild = Aptr.rchild // set left child of P to right child of left child(A) of P
3. Aptr.rchild = Pptr // set P as right child for left child(A) of pivot P

4. Pptr.height = COMPUTE_HEIGHT(Pptr) // Recompute the height of P and A
5. Aptr.height = COMPUTE_HEIGHT(Aptr)
6. Pptr = Aptr // modify the pointer in the parent of pivot
7. Stop

ROTATION_RIGHT_TO_RIGHT (Pptr)

1. Bptr = Pptr.rchild // initialize the pointer to right child of pivot node P
2. Pptr.rchild = Bptr.lchild // set right child of P to left child of right child(B) of P
3. Bptr.lchild = Pptr // set P as left child for right child (B) of pivot P
4. Pptr.height = COMPUTE_HEIGHT(Pptr) //Recompute the height of P and B
5. Bptr.height = COMPUTE_HEIGHT(Bptr)
6. Pptr = Bptr // modify the pointer field in the parent of pivot node
7. Stop

ROTATION_LEFT_TO_RIGHT (Pptr)

1. Aptr = Pptr.lchild //initialize pointer to left child of pivot node (P)
2. ROTATION_RIGHT_TO_RIGHT(Aptr) // single rotation taking A as pivot node
3. ROTATION_LEFT_TO_LEFT(Pptr) // another single rotation taking P as pivot
4. Stop

ROTATION_RIGHT_TO_LEFT (Pptr)

1. Aptr = Pptr.Rchild // initialize pointer to right child of pivot node P
2. ROTATION_LEFT_TO_LEFT(Aptr) //single rotation taking A as pivot node
3. ROTATION_RIGHT_TO_RGHT(Pptr) // another single rotation taking P as pivot node
4. Stop

INSERT_AVLTREE(ROOT,NEW)

1. ptr = ROOT
2. If(ptr = NULL) // insertion into a null tree
   2.1.ptr = NEW //create a new node
   2.2.ptr.height = 1 //height of one node tree

2.3.return //insertion is done

3. else

3.1.if (NEW.data < ptr.data) then

3.1.1. INSERT_AVLTREE(ptr.lchild, NEW)

3.1.2. Lptr = ptr.lchild

3.1.3. Rptr = ptr.rchild //right sub tree of ptr

3.1.4. If (Rptr = NULL) then // if right sub tree is empty

3.1.4.1.$H_R = 0$

3.1.5. Else

3.1.5.1. H_R = Rptr.height //height of the right sub tree

3.1.6. End if

3.1.7. H_L = Lptr.height

3.1.8. bf = $H_L - H_R$

3.1.9. if (bf = 2) then // if left high

3.1.9.1.if (NEW.data < Lptr.Data) then // if insertion take place at left
//of left child of pivot

3.1.9.1.1. ROTATION_LEFT_TO_LEFT(ptr)

3.1.9.2.Else // if insertion take place at right
//of left child of pivot

3.1.9.2.1. ROTATION_LEFT_TO_RIGHT(ptr)

3.1.9.3.End if

3.1.9.4.ptr.height = COMPUTE_HEIGHT(ptr)

3.1.10. End if

3.2.End if

3.3.Else

3.3.1. if (NEW.data > ptr.data) then

3.3.1.1.INSERT_AVLTREE(ptr.rchild, NEW)

3.3.1.2.Lptr = ptr.lchild

3.3.1.3.Rptr = ptr.rchild

3.3.1.4.If (Lptr = NULL) then

3.3.1.4.1. $H_L = 0$

3.3.1.5.Else

3.3.1.5.1. H_L = Rptr.height

3.3.1.6.End if

3.3.1.7.H_R = Lptr.heght

3.3.1.8.bf = H_L - H_R

3.3.1.9.if (bf = -2) then // if right high

3.3.1.9.1. if (NEW.data > Lptr.Data) then //if insertion take place at right //of right child of pivot

3.3.1.9.1.1.ROTATION_RIGHT_TO_RIGHT(ptr)

3.3.1.9.2. Else // if insertion take place at left //of right child of pivot

3.3.1.9.2.1.ROTATION_RIGHT_TO_LEFT(ptr)

3.3.1.9.3. End if

3.3.1.9.4. ptr.height = COMPUTE_HEIGHT(ptr)

3.3.1.10. End if

3.3.2. End if

3.3.3. Else

3.3.3.1.Print " new.data is already exist in the tree"

3.3.4. End if

3.4.End if

4. End if
5. Stop

DELETE_AVLTREE(ROOT,ITEM)

1. if(ROOT = null) then
   1.1. return null
2. else
   2.1. if(ITEM < ROOT.data) then
      2.1.1. root.left = DELETE_AVLTREE(ROOT.left, ITEM)
      2.1.2. Lptr = ROOT.left
      2.1.3. Rptr = ROOT.right
      2.1.4. H_L = Lptr.HEIGHT
      2.1.5. H_R = Rptr.HEIGHT
      2.1.6. bf = H_L – H_R

2.1.7. if(bf = -2) then

2.1.7.1. rrptr = Rptr.right

2.1.7.2. lrptr =Rptr.left

2.1.7.3. H_L = lrptr.HEIGHT

2.1.7.4. H_R = rrptr.HEIGHT

2.1.7.5. bf = H_L – H_R

2.1.7.6. if(bf = 1) then

2.1.7.6.1. ROOT = ROTATION_RIGHT_TO_LEFT(ROOT)

2.1.7.7. Else

2.1.7.7.1. ROOT =ROTATION_RIGHT_TO_RIGHT(ROOT)

2.1.7.8. End if

2.1.8. End if

2.2. End if

2.3. Else

2.3.1. if(ITEM > ROOT.data) then

2.3.1.1. ROOT.right = DELETE_AVLTREE(ROOT.right , ITEM)

2.3.1.2. Lptr = ROOT.left

2.3.1.3. Rptr = ROOT.right

2.3.1.4. H_L = Lptr.HEIGHT

2.3.1.5. H_R = Rptr.HEIGHT

2.3.1.6. bf = H_L – H_R

2.3.1.7. if(bf = 2) then // if left high

2.3.1.7.1. rlptr = Lptr.right

2.3.1.7.2. llptr =Lptr.left

2.3.1.7.3. H_L = llptr.HEIGHT

2.3.1.7.4. H_R = rlptr.HEIGHT

2.3.1.7.5. bf = H_L – H_R

2.3.1.7.6. if(bf = -1) then //left child is right high

2.3.1.7.6.1. ROOT = ROTATION_LEFT_TO_RIGHT(ROOT)

2.3.1.7.7. Else

2.3.1.7.7.1. ROOT =ROTATION_LEFT_TO_LEFT(ROOT)

2.3.1.7.8. End if

2.3.1.8. End if

2.3.2. End if

2.3.3. Else

2.3.3.1. Ptr = ROOT

2.3.3.2. Lptr = ROOT.left

2.3.3.3. Rptr = ROOT.right

2.3.3.4. if(Lptr =null) and (Rptr = null) then

2.3.3.4.1. ROOT = null

2.3.3.5. Else

2.3.3.6. if(Lptr = null) then

2.3.3.6.1. root = Ptr.right

2.3.3.6.2. Ptr = null

2.3.3.7. Else

2.3.3.8. if(Rptr = null) then

2.3.3.8.1. ROOT = Ptr.left

2.3.3.8.2. Ptr = null

2.3.3.9. Else

2.3.3.9.1. ROOT.right = DELETE_MIN(Ptr.right, Ptr)

2.3.3.9.2. Lptr = ROOT.left

2.3.3.9.3. Rptr = ROOT.right

2.3.3.9.4. H_L = Lptr.HEIGHT

2.3.3.9.5. H_R = Rptr.HEIGHT

2.3.3.9.6. bf = $H_L - H_R$

2.3.3.9.6.1. if(bf = 2) then

2.3.3.9.6.2. rlptr = Lptr.right

2.3.3.9.6.3. llptr =Lptr.left

2.3.3.9.6.4. H_L = llptr.HEIGHT

2.3.3.9.6.5. H_R = rlptr.HEIGHT

2.3.3.9.6.6. bf = $H_L - H_R$

2.3.3.9.6.7. if(bf = -1) then

2.3.3.9.6.7.1. ROOT = ROTATION_LEFT_TO_RIGHT(ROOT)

2.3.3.9.6.8. Else

2.3.3.9.6.8.1. ROOT =ROTATION_LEFT_TO_LEFT(ROOT)

2.3.3.9.6.9. End if

2.3.3.9.7. End if

2.3.3.10. End if

2.3.4. End if

2.4. End if

3. Return ROOT
4. Stop

DELETE_MIN (succ, ptr)

1. temp = succ
2. if (succ.left ≠ null) then

2.1. succ.left = DELETE_MIN (succ.left, ptr)

3. Else

3.1. temp = succ

3.2. ptr.data = succ.data

3.3. succ = succ.right

3.4. temp= null

4. end if
5. Return(succ)
6. Stop

Sample INPUT:

1. Insert
2. Delete
3. Search
4. Display
5. Exit

Enter your choice : 1

Enter element to be inserted in to AVL tree :

46 34 75 22 41 55 80 11 25 38 44 60 79 90 9 36 40 45 85 35

Enter your choice : 2

Enter element to be deleted from AVL tree : 75

Enter your choice : 3

Enter element to be searched: 44

Enter your choice : 3

Enter element to be searched: 87

Enter your choice : 5

Expected output:

AVL Tree nodes after insertion (in order traversal):

9 11 22 25 34 35 36 38 40 41 44 45 46 55 60 75 79 80 85 90

AVL Tree nodes after deletion (in order traversal) :

9 11 22 25 34 35 36 38 40 41 44 45 46 55 60 79 80 85 90

44 found

87 not found

Conclusion: Student can understand and analyze the AVL trees for balancing the trees so that increase the speed of insertion, deletion and searching of elements in the given list. It is attained to PO1, PO2, PO3, PO4, PSO1 and CO2

Viva Questons:

1. What is AVL tree?
2. The time complexity of searching in AVL tree is?
3. The time complexity of insertion n AVL tree is ?
4. When to perform RIGHT- to - RIGHT rotation?
5. When to perform LEFT- to - LEFT rotation?
6. When to perform double RIGHT Rotation?
7. When to perform double LEFT Rotation?
8. Give maximum height of AVL tree with n nodes.

Experiment No: 3

Aim: - To perform various operations i.e., insertions and deletions on 2-3 trees.

Description: -

In computer science, a 2–3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. According to Knuth, "a B-tree of order 3 is a 2-3 tree

Algorithm:

1. **The lookup operation**

Recall that the lookup operation needs to determine whether key value k is in a 2-3 tree T. The lookup operation for a 2-3 tree is very similar to the lookup operation for a binary-search tree. There are 2 base cases:

1. T is empty: return false
2. T is a leaf node: return true iff the key value in T is k

And there are 3 recursive cases:

1. k <= T.leftMax: look up k in T's left subtree
2. T.leftMax < k <= T.middleMax: look up k in T's middle subtree
3. T.middleMax < k: look up k in T's right subtree

It should be clear that the time for lookup is proportional to the height of the tree. The height of the tree is O(log N) for N = the number of nodes in the tree. You may think this is a problem, since the actual values are only at the leaves. However, the number of leaves is always greater than N/2 (i.e., more than half the nodes in the tree are leaves). So the time for lookup is also O(log M), where M is the number of key values stored in the tree.

2.The insert operation

The goal of the insert operation is to insert key k into tree T, maintaining T's 2-3 tree properties. Special cases are required for empty trees and for trees with just a single (leaf) node. So the form of insert will be:

if T is empty replace it with a single node containing k

else if T is just 1 node m:

(a) create a new leaf node n containing k

(b) create a new internal node with m and n as its children,

and with the appropriate values for leftMax and middleMax

else call auxiliary method insert(T, k)

The auxiliary insert method is the recursive method that handles all but the 2 special cases; as for binary-search trees, the first task of the auxiliary method is to find the (non-leaf) node that will be the parentof the newly inserted node.

The auxiliary insert method performs the following steps to find node n, the parent of the new node:

- base case: T's children are leaves - n is found! (T will be the parent of the new node)
- recursive cases:
  - k < T.leftMax: insert k into T's left subtree
  - T.leftMax < k < T.middleMax, or T only has 2 children: insert k into T's middle subtree
  - k > T.middleMax and T has 3 children: insert k into T's right subtree

Once n is found, there are two cases, depending on whether n has room for a new child:

Case 1: n has only 2 children

- Insert k as the appropriate child of n:
    1. if k < n.leftMax, then make k n's left child (move the others over), and fix the values of n.leftMax and n.middleMax. Note that all ancestors of n still have correct values for their leftMax and middleMax fields (because the new value is not the "max" child of n).
    2. if k is between n.leftMax and n.middleMax, then make k n's middle child and fix the value of n.middleMax. Again, no ancestors of n need to have their fields changed.
    3. if k > n.middleMax, then make k n's right child and fix the leftMax or middleMax fields of n's ancestors as needed.

Case 2: n already has 3 children

- Make k the appropriate new child of n, anyway (fixing the values of n.leftMax and/or n.middleMax as needed). Now n has 4 children.
- Create a new internal node m. Give m n's two rightmost children and set the values of m.leftMax and m.middleMax.
- Add m as the appropriate new child of n's parent (i.e., add m just to the right of n). If n's parent had only 2 children, then stop creating new nodes, just fix the values of the leftMax and middleMax fields of ancestors as needed. Otherwise, keep creating new nodes recursively up the tree. If the root is given 4 children, then create a new node m as above, and create a new root node with n and m as its children.

What is the time for insert? Finding node n (the parent of the new node) involves following a path from the root to a parent of leaves. That path is O(height of tree) = O(log N), where N is the number of nodes in the tree (recall that it is also log M, where M is the number of key values stored in the tree).

Once node n is found, finishing the insert, in the worst case, involves adding new nodes and/or fixing fields all the way back up from the leaf to the root, which is also O(log N).

So the total time is O(log N), which is also O(log M).

3.The delete operation

Deleting key k is similar to inserting: there is a special case when T is just a single (leaf) node containing k (T is made empty); otherwise, the parent of the node to be deleted is found, then the tree is fixed up if necessary so that it is still a 2-3 tree.

Once node n (the parent of the node to be deleted) is found, there are two cases, depending on how many children n has:

case 1: n has 3 children

- Remove the child with value k, then fix n.leftMax, n.middleMax, and n's ancestors' leftMax and middleMax fields if necessary.

case 2: n has only 2 children

- If n is the root of the tree, then remove the node containing k. Replace the root node with the other child (so the final tree is just a single leaf node).
- If n has a left or right sibling with 3 kids, then:
    - remove the node containing k
    - "steal" one of the sibling's children
    - fix n.leftMax, n.middleMax, and the leftMax and middleMax fields of n's sibling and ancestors as needed.
- If n's sibling(s) have only 2 children, then:
    - remove the node containing k
    - make n's remaining child a child of n's sibling
    - fix leftMax and middleMax fields of n's sibling as needed
    - remove n as a child of its parent, using essentially the same two cases (depending on how many children n's parent has) as those just discussed

The time for delete is similar to insert; the worst case involves one traversal down the tree to find n, and another "traversal" up the tree, fixing leftMax and middleMax fields along the way (the traversal up is really actions that happen after the recursive call to delete has finished).

So the total time is 2 * height-of-tree = O(log N).

Sample INPUT:

1. Insert
2. Delete
3. Search
4. Display
5. Exit

Enter your choice : 1

Enter element to be inserted in to AVL tree :

46 34 75 22 41 55 80 11 25 38 44 60 79 90 9 36 40 45 85 35

Enter your choice : 2

Enter element to be deleted from AVL tree : 75

Enter your choice : 3

Enter element to be searched: 44

Enter your choice : 3

Enter element to be searched: 87

Enter your choice : 5

Conclusion: Student can understand and analyze the 2-3 trees for balancing the trees so that increase the speed of insertion, deletion and searching of elements in the given list. it is attained to PO1, PO2,PO3,PO4, PSO1 and CO2

Viva Questions:

1. What are 2-3 trees?
2. The time complexity of searching in 2-3 trees is?
3. The time complexity of insertion n 2-3 trees is?

Experiment No: 4

Aim: - To implement operations on binary heap.

Description: -

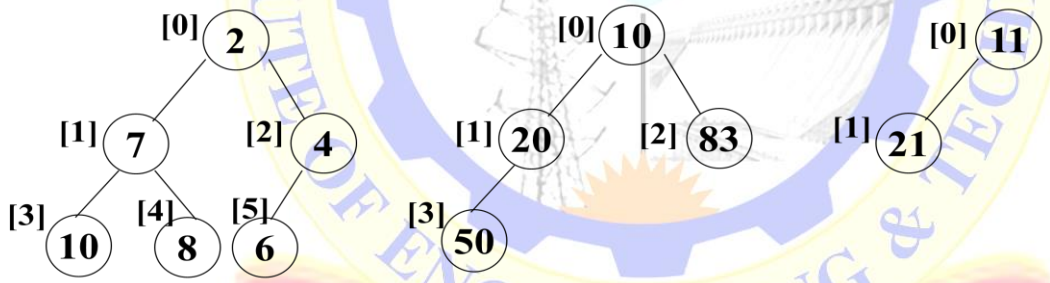
A binary heap is a binary tree which has the following two properties:

Structural Property: A heap is a complete binary tree as left as possible.

Heap or Order property : Any node key is less than or equal to its children- is called min heap. Any node key is greater than or equal to its children- is called max heap.

Insertion: A new node is inserted in the first available array cell , just to the right of the last node on the bottom row of the heap. From there it moves up to the appropriate position.

Deletion : A root node is deleted and the last element in the array cell , the last node on the bottom row of the heap is set as root node. From there it moves down to the appropriate position.



Algorithm:

BUILD_HEAP(A,SIZE)

1. i = 2
2. while (i < SIZE) do
   2.1.REHEAP_UP(A, i)
   2.2.i = i + 1
3. end while
4. stop

REHEAP_UP(A, i)

1. if(i ≠ 0) then
   1.1.p = i div 2 // parent of new node
   1.2.if (p > 0 and A[i] > A[p])then // continue till root is reached
      1.2.1. temp = A[i]
      1.2.2. A[i] = A[p]
      1.2.3. A[p] = temp
      1.2.4. REHEAP_UP(A,p) // parent becomes child & reorder
   1.3.End if
2. End if
3. Stop

MIN_HEAP_INSERT(ITEM)

1. If (N>Size) then
   1.1. Print " heap tree is saturated"
   1.2. Exit
2. Else
   2.1.N = N+1
   2.2. A[N] = ITEM // insert data in the first available cell
   2.3. i = N // last node is current node
   2.4.p = i div 2 // parent of current node
   2.5.While (p > 0) and (a[p] > a[i]) do //continue root node is reached or out of order
      2.5.1. Temp = A[i]
      2.5.2. A[i] = A[p]
      2.5.3. A[p] = Temp
      2.5.4. i = p // parent becomes child
      2.5.5. p = p div 2 // parent of parent becomes new parent
   2.6.end while
3. End if
4. Stop

MIN_HEAP_DELETE()

1. If (N = 0) then
   1.1.Print " heap tree is exhausted"
   1.2.Exit
2. End if
3. Item = A[1] // value at the root node under deletion
4. A[1] = A[N] // replace the value at the root by its counterpart at last node on last level
5. N = N-1 // size of heap tree is reduced by 1
6. Flag = false, i = 1
7. While(Flag = false) and (I <N) do // rebuild the heap
   7.1.Lchild = 2 * i, Rchild = 2 * i + 1 //address of the left and right child of the current node
   7.2.If(Lchild ≤ N) then
      7.2.1. X = A[Lchild]
   7.3.Else
      7.3.1. X = -∞
   7.4.End if
   7.5.If(Rchild ≤ N) then
      7.5.1. Y = A[Rchild]
   7.6.Else
      7.6.1. Y = -∞
   7.7.End if
   7.8. if (A[i] < X) and (A[i] < Y) then //if parent is smaller than its children
      7.8.1. Flag = true // reheap is over
   7.9. else //Any child may have small value
      7.9.1. if (X < Y) and (A[i] > X) // if left child is smaller than right child
         7.9.1.1. Swap (A[i] , A[Lchild]) // interchange data between parent & left child
         7.9.1.2. i = Lchild //left child becomes current node
      7.9.2. else
         7.9.2.1. if(Y < X) and (A[i] > Y) // if right child is smaller than left child

7.9.2.1.1. swap (A[i], A[Rchild]) //interchange data between parent & right child

7.9.2.1.2. i = Rchild //right child becomes current node

7.9.2.2. End If

7.9.3. End If

7.10. End if

8. End while

9. Stop

Sample INPUT:

Maximum size of heap :20

Enter no of elements to build the heap : 7

Enter elements :

17 10 21 28 45 32 43 29 39

1.Insert 2. Delete 3. Display 4.exit

Enter your choice : 1

Enter elements to be inserted : 34

Enter your choice : 2

Enter your choice : 4

Expected output:

Heap after build:

0	1	2	3	4	5	6	7	8
10	17	21	28	45	32	43	29	39

Heap after insertion:

0	1	2	3	4	5	6	7	8	9

10	17	21	28	34	32	43	29	39	45

Heap after deletion:

0	1	2	3	4	5	6	7	8
17	28	21	29	34	32	43	45	39

Conclusion: Student can understand and analyze the Binary Heap implementation so that he can implement the priority queues very effectively. So it is attained with PO1, PO2, PSO2 and CO3

Viva Questions:

1. How you can differentiate between a Max and Minimum Heap
2. How Maximum Heap Deals with the root node
3. How Minimum Heap Deals with the tail Nodes
4. Is there any Complexity maintaining Heap? If so explain.
5. What is the Order of Best Complexity Case
6. Give applications of binary heap.
7. What is the difference between heap and binary tree.
8. What are the similarities between heap and binary search tree.
9. Explain heapify procedure.

Experiment No: 5

Aim: - To implement operations on graphs

i) Vertex insertion ii) Vertex deletion

iii) Finding vertex iv) Edge addition and deletion

Description: -

A Graph G consists of two sets:

(i) A non empty set V called set of all vertices or nodes.

(ii) A finite set E , called set of edges or arcs or links. This set E is the set of pair of elements from V.

There are six primitive operations that provide the basic modules needed to maintain a graph:
Add vertex, delete vertex, add an edge, delete an edge, find a vertex, traverse a graph.

ADD VERTEX : Add vertex inserts a new vertex into a graph. When a vertex is added it is disjoint ; that is not connected to any other vertices in the list. Then update if any adjacent vertices to it.

DELETE VERTEX : Delete vertex removes a vertex from the graph. When a vertex is deleted, all connecting edges are also removed. To delete a vertex V_i from the graph, first look for the adjacency matrix of V_i . all the vertices which are present in the adjacency matrix of V_i , the node labeled V_i has to be deleted from the adjacency lists of those vertices.

ADD EDGE : Add edge connects a vertex to a destination vertex. If a vertex requires multiple edges, then add an edge must be called once for each adjacent vertex. To add an edge , two vertices must be specified. If the graph is a digraph then one of the vertices must be specified as the source and one as the destination.

DELETE EDGE: Delete edge removes one edge from a graph. To delete an edge between the two nodes V_i and V_j in an undirected – delete the node having label V_j in the adjacency matrix of V_i as well as the node having label V_i in the adjacency matrix of V_j.

FIND VERTEX : find vertex traverses a graph looking for a specified vertex. If the vertex is found , its data are returned. If it is not found , an error is indicated.

Algorithm:

INSERT_VERTEX(V_X, X)

1. N = N + 1,V_X = N // no of vertices is increased by 1 and label of new vertex is N
2. For i = 1 to V_X do // insert a row and column for V_x
   2.1.Gptr[V_X][i] = 0 // row vector is initialized
   2.2.Gptr[i][V_X] = 0 // column vector is initialized
3. End for

   // to add adjacency matrix of new vertex V_x in the graph
4. For I = 1 to N do
   4.1.j = X[i] // j is the label of i^{th} adjacent vertex
   4.2.if(j ≥ N)then
      4.2.1. print " no vertex labeled X[i] does not exist"
   4.3.else
      4.3.1. Gptr[V_x][j] = 1 // add entry from V_X to X[i]
      4.3.2. Gptr[j][V_x] = 1 // add entry from X[i] to V_X
   4.4.End if
5. End for
6. Stop

INSERT_EDGE(V_i , V_J)

1. If (V_i > N) or (V_J > N) then
   1.1.Print " edge is not possible between V_i and V_j "
2. Else
   2.1.Gptr[V_i][V_J] = 1 // Set (i,j) entry
   2.2.Gptr[V_j][V_i] = 1 // Set (j,i) entry
3. End if
4. Stop

DELETE_VERTEX(V_X)

1. If N = 0 then
   1.1.Print " graph is empty : deletion is not possible"
   1.2.Exit
2. End if
3. If (V_X > N) then
   3.1.Print "Vertex does not exist"
   3.2.Exit
4. End if

   // to remove the row and column of V_X vertex

5. j = V_X
6. for I = j to N-1 do
   6.1.for k = 1 to N do
      6.1.1. Gptr[k][i] = Gptr[k][i+1] //shift all the columns after V_X towards left once
      6.1.2. Gptr[i][k] = Gptr[i+1][k] //Move all the rows after V_X towards up once
   6.2.End for
7. End for
8. N = N-1
9. Stop

DELETE_EDGE(V_i , V_j)

1. If (V_i > N) or (V_J > N) then
   1.1. Print " vertex does not exist : error in edge removal "
2. Else
   2.1.Gptr[V_i][V_J] = 0 // Set (i,j) entry
   2.2.Gptr[V_j][V_i] = 0 // Set (j,i) entry
3. End if
4. Stop

FIND_VERTEX(V_X)

1. If (V_X > N) then
   1.1.Print " vertex does not exist "
   1.2.Exit
2. Else
   2.1.For i = 1 to N do
      2.1.1. If (Gptr[V_X][i] = 1) then
         2.1.1.1.Print " i "
      2.1.2. End if
   2.2.End for
3. End if
4. Stop

Sample INPUT:

Enter no of vertices in the graph : 7

Enter adjacency matrix for graph

0 1 1 0 0 0 0

1 0 0 1 1 0 0

1 0 0 1 0 1 0

0 1 1 0 0 0 1

0 1 0 0 0 0 1

0 0 1 0 0 0 1

0 0 0 1 1 1 0

1. insert vertex

2. delete vertex

3. insert edge

4. delete edge

5. find vertex

6. display

7. exit

Enter the graph operation: 1

Enter adjacent vertices for new vertex: 2 3 6

Enter the graph operation : 2

Enter vertex to be deleted: 3

Enter graph operation: 3

Enter two vertices where edge to be added: 4 7

Enter graph operation: 4

Enter two vertices where edge to be deleted: 2 3

Enter graph operation: 5

Enter vertex to be searched: 3

Enter graph operation : 7

Expected output:

After inserting a new node:

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	1
1	0	0	1	0	1	0	1
0	1	1	0	0	0	1	0
0	1	0	0	0	0	1	0
0	0	1	0	0	0	1	1
0	0	0	1	1	1	0	0
0	1	1	0	0	1	0	0

After deleting node 3 :

0 1 0 0 0 0 0

1 0 1 1 0 0 1

0 1 0 0 0 1 0

0 1 0 0 0 1 0

0 0 0 0 0 1 1

0 0 1 1 1 0 0

0 1 0 0 1 0 0

After adding edge between 4, 7 :

0 1 0 0 0 0 0

1 0 1 1 0 0 1

0 1 0 0 0 1 0

0 1 0 0 0 1 1

0 0 0 0 0 1 1

0 0 1 1 1 0 0

0 1 0 1 1 0 0

After deleting edge between 2, 3:

0 1 0 0 0 0 0

1	0	0	1	0	0	1
0	0	0	0	0	1	0
0	1	0	0	0	1	1
0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0

The vertex 3 has the following adjacent vertices: 6

Conclusion: Student can get knowledge and understand and analyze the Graph operations so that he can implement the computer networks related applications in real world very effectively. So it is attained with PO1, PO2, PO3, PO4, PSO1, and PSO3 and CO4

Viva Questons:

1. Define a graph?
2. What are the different graph storage representations?
3. Give applications of graph?
4. Give basic operations on graphs?
5. Give advantages and disadvantages of adjacency matrix representation?
6. State advantages and disadvantages of adjacency list representation?
7. Define null graph?

Experiment No: 6

Aim: - To implement Depth First Search for a graph non-recursively.

Description: -

Depth-first search is a generalization of the preorder traversal of a tree. DFS can serve as a structure around which many other efficient graph algorithms can be built.

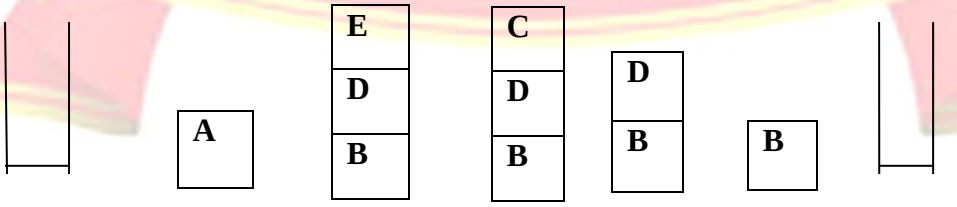
- ✓ DFS traversal begins at vertex V and visit the vertex V first.
- ✓ Select any vertex adjacent to V and visit.
- ✓ Visit each vertex, select adjacent vertex until a vertex with no adjacent entries is reached.
- ✓ Then backtrack along the path to V, if it has another vertex other than previous one or unvisited adjacent vertex then continue same procedure.

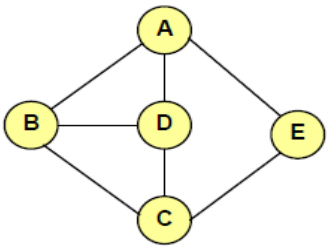
If the graph is not connected, DFS must be called on a node of each connected component. The VISIT array gives the order of visit of vertices during traversal. A stack is used to maintain the track of all paths from any vertex. As an example of DFS, suppose in the graph of the following figure, we start at node A.

VISIT :

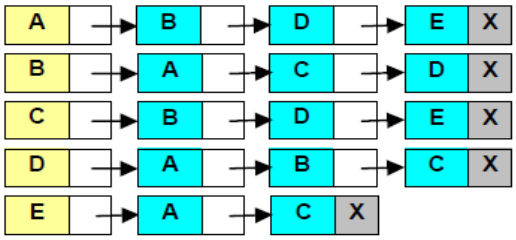
A	E	C	D	B

STACK :





Undirected Graph



Adjacency list

	A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	1	0	0
E	1	0	1	0	0

Adjacency matrix

Algorithm:

V, the starting vertex. Visit is an array . Initially for all vertices in the graph visit[i] =0 , where i is vertex in graph. adjmat[u] is adjacent matrix that is gives adjacent vertices for vertex u. N , the number of vertices in the graph.

DEPTH_FIRST_SEARCH()

1. If adjmat=NULL then
   1.1. Print "Graph is empty"
   1.2. exit
2. end If
3. u = v // start from v
4. stack.PUSH(u) // push the starting vertex into stack
5. while (stack.TOP ≠ NULL) do // till the stack is not empty
   5.1. u = stack.POP() // pop the top element from stack
   5.2. if (visit[u] = 0) then // if u is not in visit
      5.2.1. visit[u] = 1 // mark u as visited
      5.2.2. print " u"
      5.2.3. for i=1 to N // to push all the adjacent vertices of u into stack
         5.2.3.1. if (adjmat[u][i] = 1) then
            5.2.3.1.1. stack .PUSH(v)
         5.2.3.2. End if
      5.2.4. end for
   5.3. end if

6. end while
7. stop

Sample INPUT:

Enter number of vertices in graph : 5

Enter adjacency matrix for graph:

0	1	0	1	1
1	0	1	1	0
0	1	0	1	1
1	1	1	0	0
1	0	1	0	0

Expected output:

In DFS the vertices are traversed in the following order :

0 4 2 3 1

Conclusion: Student can get knowledge and understand and analyze the Graph traversing problem so that he can implement the computer networks related applications in real world very effectively. So it is attained with PO1, PO2, PO3, PO4, PSO1, and PSO3 and CO4

Viva Questions:

1. Explain DFS.
2. Why DFS is named to be a traversal technique?
3. How DFS is more convenient than BFS?
4. How internal Node and Visited Node are Different?
5. Can we use this techniques for disconnected graphs also. If So Explain ?
6. Does this supports recursion factors?
7. Explain about the adjacency matrix?
8. How Adjacency matrix useful in traversal techniques?
9. Which data structure supports for implementation of DFS?

Experiment No: 7

Aim: - To implement Breadth First Search for a graph non-recursively.

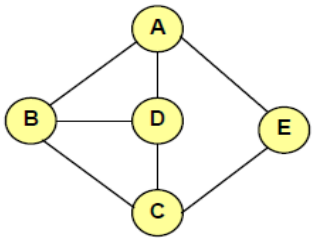
Description: -

Another systematic way of visiting the nodes is called *breadth-first search* (BFS). The approach is called "breadth-first" because from each node *v* that we visit we search as broadly as possible by next visiting all nodes adjacent to *v*.

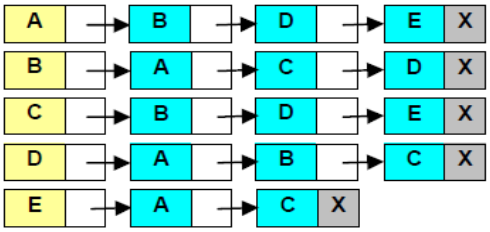
- ✓ The BFS begins by picking a starting vertex.
- ✓ After processing it, process all of its adjacent vertices.
- ✓ After processing all of the first adjacent vertices , pick the first adjacent vertex and process all of its adjacent vertices.
- ✓ Then second adjacent vertex , process all of its adjacent vertices.
- ✓ This process continues until finished.

The Breadth First Search algorithm inserts a node into a queue, which we assume is initially empty. Every entry in the array *mark* is assumed to be initialized to the value *unvisited*. If the graph is not connected, BFS must be called on a node of each connected component. Note that in BFS we must mark a node visited before inserting it into the queue, to avoid placing it on the queue more than once. The algorithm terminates when the queue becomes empty.

The VISIT array gives the order of visit of vertices during traversal. A stack is used to maintain the track of all paths from any vertex. As an example of DFS, suppose in the graph of the following figure, we start at node A.



Undirected Graph



Adjacency list

	A	B	C	D	E
A	0	1	0	1	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	1	0	0
E	1	0	1	0	0

Adjacency matrix

A	B	D	E	C

VISIT:

QUEUE:

	A	E D B	C E D	C E	C	

Algorithm:

BREADTHTH_FIRST_SEARCH(V)

1. If (adjmat = NULL) then
   1.1. Print " Graph is empty"
   1.2. exit
2. End if
3. u = V // start from V
4. queue.ENQUEUE(u) // push the starting vertex into queue
5. while (queue.STATUS() ≠ EMPTY) do // till the queue is not empty
   5.1. u = queue.DEQUEUE() // delete the first element from queue
   5.2. if (visit[u] = 0) then // if u is not in visit
      5.2.1. visit[u] = 1 // mark the vertex u as visited
      5.2.2. print "u "
      5.2.3. for I =1 to N do //to enter all the adjacent vertices of u
         5.2.3.1. if (adjmat[u][i] = 1) then // into queue
            5.2.3.1.1. queue.ENQUEUE(i)
         5.2.3.2. End If

5.2.4. End for

5.3. End If

6. end while
7. stop

Sample INPUT:

Enter number of vertices in graph : 5

Enter adjacency matrix for graph:

0 1 0 1 1

1 0 1 1 0

0 1 0 1 1

1 1 1 0 0

1 0 1 0 0

Expected output:

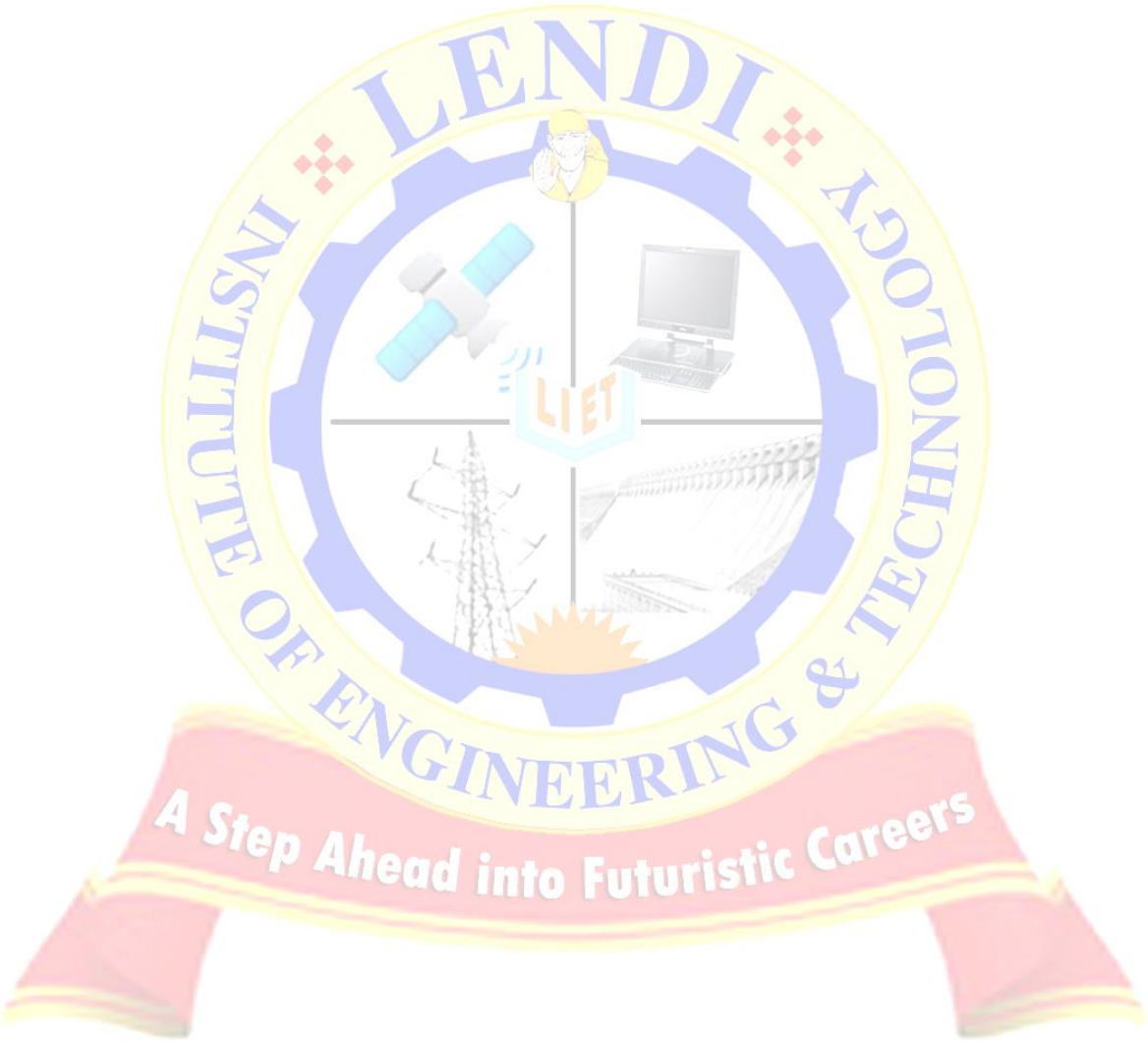
In BFS the vertices are visited in the following order :

0 1 3 4 2

Conclusion: Student can get knowledge and understand and analyze the Graph traversing problem so that he can implement the computer networks related applications in real world very effectively. So it is attained with PO1, PO2, PO3, PO4, PSO1, and PSO3 and CO4

Viva Questions:

1. Give any two differences between DFS and BFS?
2. Explain the Main logic in BFS?
3. Which data structure is used for implementation of BFS?
4. Give applications of BFS & DFS?
5. Is two vertices are connected or not ? this question cane answered with BFS & DFS? Explain.



Experiment No: 8

Aim: To implement Prim's algorithm to generate a min-cost spanning tree.

Description: -

Definition : Spanning trees –

Let G = (V , E) be an undirected connected graph. A subgraph t= (V , E') of G is a spanning tree if and only if t is a tree. (No cycle exists.)

In practical situations, the edges have weights assigned to them, weights are positive. These weights may represent the cost of construction, lengths of links etc.

Given such weighted graph one would like to select cities (vertices) to have minimum total cost / minimum total length. So one can find a spanning tree with minimum cost.

Since identification of minimum spanning tree involves selection of a subset of the edges, this problem fits into the Subset Paradigm.

Prim's algorithm works by attaching a new edge to a single growing tree at each step: Start with any vertex as a single-vertex tree; then add V-1 edges to it, always taking next (coloring black) the minimum-weight edge that connects a vertex on the tree to a vertex not yet on the tree (a crossing edge for the cut defined by tree vertices).

Algorithm:

PRIM():

1. for i = 0 to N-1 do
   1.1.selected[i] = false

2. End for
3. Selected[0]=1,ne=1,sum=0
4. While (ne < N) do

   4.1.min = ∞

   4.2.for i=0 to N-1 do

      4.2.1. If(slelcted[i]=true) then

         4.2.1.1.for j=1 to N-1 do

            4.2.1.1.1. if(selected[j]=false) then

               4.2.1.1.1.1.if(min > adjMat[i][j]) then

                  4.2.1.1.1.1.1. min = adjMat[i][j]

                  4.2.1.1.1.1.2. Row = i

                  4.2.1.1.1.1.3. Col = j

               4.2.1.1.1.2.End if

            4.2.1.1.2. End if

         4.2.1.2.End for

      4.2.2. End if

   4.3.End for

   4.4.selected[col]=true

   4.5.ne = ne + 1

   4.6.sum = sum + adjMat[row][col]

   4.7.Print " edge(row,col) cost : adjMat[i][j]"
5. End while
6. Print "total min cost – sum"
7. Stop

Sample INPUT:

Enter number of vertices in a graph: 7

Enter weight matrix for graph:

99999 9 99999 99999 99999 12 99999

9	99999	5	99999	99999	99999	4
99999	5	99999	4	99999	99999	99999
99999	99999	4	99999	7	99999	6
99999	99999	99999	7	99999	8	8
12	99999	99999	99999	8	99999	99999
99999	4	99999	6	8	99999	99999

Expected output:

Min-Cost Spanning tree:

Edge (0 , 1) cost : 9

Edge (1 , 6) cost : 4

Edge (1 , 2) cost : 5

Edge (2 , 3) cost : 4

Edge (3 , 4) cost : 7

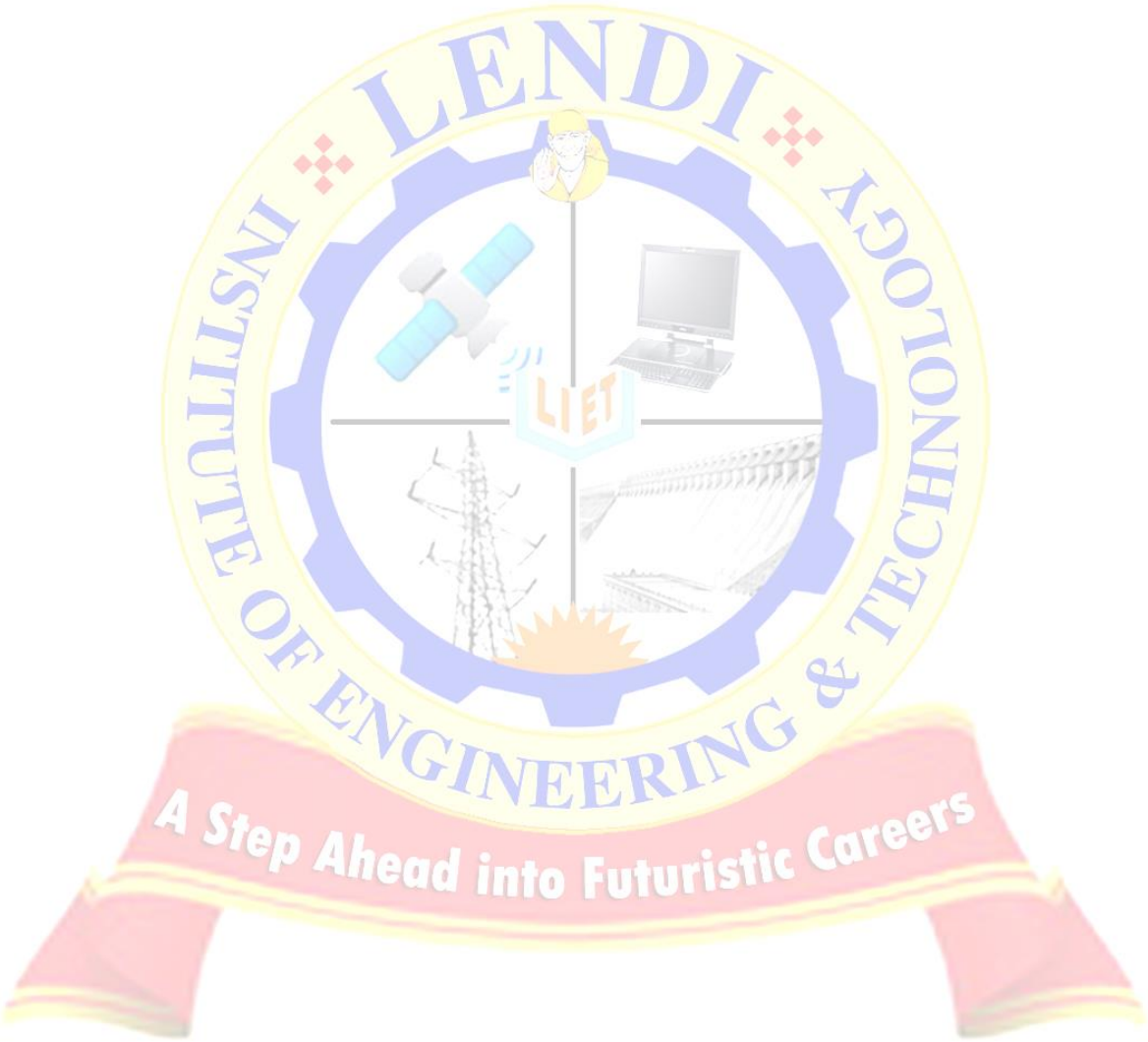
Edge (4 , 5) cost : 8

Total Min-cost = 37

Conclusion: Student can get knowledge and analyzing the Minimum spanning tree for the given graph so that he can implement the applications like GPS and traffic analysis. So it is attained with PO1, PO2, PO3 and CO5

Viva Questons:

1. What is minimum cost spanning tree?
2. What is the time complexity for prim's algorithm
3. What are the differences between prims and kruskal algorithm



Experiment No: 9

Aim: - To implement Krushkal's algorithm to generate a min-cost spanning tree.

Description: -

AN edge – weighted graph is a graph where we associate weights or costs with each edge. A minimum spanning tree (MST) of an edge – weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

- *The graph is connected . The spanning tree condition in our definition implies that* the graph must be connected for an MST to exist. If a graph is not connected, we can adapt our algorithms to compute the MSTs of each of its connected components, collectively known as a *minimum spanning forest*.
- *The edge weights are not necessarily distances.* Geometric intuition is sometimes beneficial, but the edge weights can be arbitrary.
- *The edge weights may be zero or negative.* If the edge weights are all positive, it suffices to define the MST as the sub graph with minimal total weight that connects all the vertices.
- *The edge weights are all different.* If edges can have equal weights, the minimum spanning tree may not be unique. Making this assumption simplifies some of our proofs, but all of our algorithms work properly even in the presence of equal weights.

Krushkal's algorithm processes the edges in order of their weight values (smallest to largest), taking for the MST (coloring black) each edge that does not form a cycle with edges previously added, stopping after adding V-1 edges. The black edges form a forest of trees that evolves gradually into a single tree, the MST.

Algorithm:

1. K = 1
2. For I =1 to n-1 do

   2.1.For j = i+1 to N do

2.1.1. If adjMat[i][j] > 0 then

2.1.1.1.X[k].V_i = i

2.1.1.2.X[k].V_j = j

2.1.1.3.X[k].weight = adjMat[i][j]

2.1.1.4.X[k].select = false

2.1.1.5.K = k + 1

2.1.2. End if

2.2.End for

3. End for
4. Ne = k
5. If ne < N-1 then

5.1.Print " no spanning tree possible in the graph"

5.2.Exit

6. End if
7. X.sort_edges()
8. For I =1 to n-1 do

8.1.For j = i+1 to N do

8.1.1. TREE[i][j]=0

8.2.End for

9. End for
10. K = 1 , l = 1
11. While k < N do

11.1. temp = TREE

11.2. i = X[l].V_i, j = X[l].V_j

11.3. temp[i][j] = l, temp[j][i] = l

11.4. WARSHALL(temp)

11.5. Flag = false

11.6. For p = 1 to N do

11.6.1. If temp[p][p] =1 then

11.6.1.1. Flag =true

11.6.1.2. Break

11.6.2. End if

11.7. End for

11.8. If not flag then

11.8.1. TREE[i][j] =1, TREE[j][i] = 1

11.8.2. K = k + 1

11.8.3. X[l].select = true

11.9. End if

11.10. L = l + 1

12. End while

13. Return TREE

14. Stop

Sample INPUT:

Enter number of vertices in a graph:

7

Enter weight matrix for graph:

99999	9	99999	99999	99999	12	99999
9	99999	5	99999	99999	99999	4
99999	5	99999	4	99999	99999	99999
99999	99999	4	99999	7	99999	6
99999	99999	99999	7	99999	8	8
12	99999	99999	99999	8	99999	99999

99999	4	99999	6	8	99999	99999

Expected output:

Min-Cost Spanning tree:

Edge (0 , 1) cost : 9

Edge (1 , 6) cost : 4

Edge (1 , 2) cost : 5

Edge (2 , 3) cost : 4

Edge (3 , 4) cost : 7

Edge (4 , 5) cost : 8

Total Min-cost = 37

Conclusion: Student can get knowledge and analyzing the Minimum spanning tree for the given graph so that he can implement the applications like GPS and traffic analysis. So it is attained with PO1,PO2, PO3, PSO2,PSO3 and CO5

Viva Questons:

1. What is minimum cost spanning tree?
2. What is the time complexity for kruskal 's algorithm
3. What are the differences between prim's and kruskal's algorithm

Experiment No:10

Aim: - To implement Dijkstra's algorithm to find shortest path in the graph.

Description: -

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959,[1][2] is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network routing protocols, most notably IS-IS and OSPF(Open Shortest Path First).

Dijkstra's original algorithm does not use a min-priority queue and runs in $O(|V|^2)$. The idea of this algorithm is also given in (Leyzorek et al. 1957). The implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V| \log |V|)$ is due to (Fredman & Tarjan 1984). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded nonnegative weights.

In the following algorithm, the code u := vertex in Q with smallest dist[], searches for the vertex u in the vertex set Q that has the leastdist[u] value. That vertex is removed from the set Q and returned to the user. dist_between(u, v) calculates the length between the two neighbor-nodes u and v. The variable *alt* on line 15 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this *alt*path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

Algorithm:

DIJKSTRA(SOURCE):

1. Found[0] = true, d[0] = 0
2. For I =1 to n-1 do
   - 2.1. found[i] = false
   - 2.2. d[i] = adjMat[0][i]
3. End For
4. For i =1 to n-1 do
   - 4.1. min = ∞
   - 4.2. for w = 1 to n-1 do
     - 4.2.1. if not found[w] && d[w] < min then
       - 4.2.1.1. x = w
       - 4.2.1.2. min = d[w]
     - 4.2.2. End if
   - 4.3. found[x] = true
   - 4.4. for w =1 to n-1 do
     - 4.4.1. if not found[w] then
       - 4.4.1.1. if min+adjMat[x][w] < d[w] then
         - 4.4.1.1.1. d[w] = min + adjMat[x][w]
       - 4.4.1.2. End if
     - 4.4.2. End if
   - 4.5. End for
5. End for
6. Stop

Sample INPUT:

Enter number of vertices in a graph : 5

Enter weight matrix for graph:

99999	2	99999	6	20
99999	99999	10	99999	99999
99999	99999	99999	99999	2
99999	99999	4	99999	12
99999	99999	99999	99999	99999

Expected output:

Shortest path from vertex 0 to other vertices:

0 -> 1 : 2

0 -> 2 : 10

0 -> 3 : 6

0 -> 4 : 12

Conclusion: Student can get knowledge and analyzing the Shortest Path between the vertices for the given graph so that he can implement the applications like network routing protocols. So it is attained with PO1, PO2, PO3, PSO2,PSO3 and CO5

Viva Questions:

1. Dijkistra algor ithm is used for?
   Ans:- Solve shortest path problem in weighted graph.
2. Does Single Dimensional Array usefull for finding out the Adjacency Matrix
3. What are the limitations of Dijkstras Algorithm
4. Explain the main logic of Dijkstras Algorithm
5. What is a petty graph?

6. Using dijkstra's algorithm how do I calculate a route from a start node to an end node via a specific node?
7. Will Dijkstra's algorithm work properly with a link of cost 0. Explain your answer.
8. Why it is called as Single Source Shortest Path Algorithm.
9. What is the order of Time Complexity
10. How it is related with Greedy problem

Experiment No: 11

Aim: - To implement pattern matching using Boyer-Moore algorithm.

Description: -

The Boyer-Moore algorithm searches for occurrences of ***P*** in ***T*** by performing explicit character comparisons at different alignments. Instead of a brute-force search of all alignments (of which there are***m - n***), Boyer-Moore uses information gained by preprocessing ***P*** to skip as many alignments as possible.

The algorithm begins at alignment ***k = n***, so the start of ***P*** is aligned with the start of ***T***. Characters in ***P*** and ***T*** are then compared starting at index ***n*** in ***P*** and ***k*** in ***T***, moving downward: the strings are matched from the end and toward the beginning of ***P***. The comparisons continue until either a mismatch occurs or the beginning of ***P*** is reached (which means there is a match), after which the alignment is shifted to the right according to the maximum value permitted by a number of rules. The comparisons are performed again at the new alignment, and the process repeats until the alignment is shifted past the end of ***T***.

The shift rules are implemented as constant-time table lookups, using tables generated during the preprocessing of ***P***.

- ***S[i]*** refers to the character at index ***i*** of string ***S***, counting from 1.
- ***S[i..j]*** refers to the substring of string ***S*** starting at index ***i*** and ending at ***j***, inclusive.
- A prefix of ***S*** is a substring ***S[1..i]*** for some ***i*** in range ***[1, n]***, where ***n*** is the length of ***S***.
- A suffix of ***S*** is a substring ***S[i..n]*** for some ***i*** in range ***[1, n]***, where ***n*** is the length of ***S***.
- The string to be searched for is called the **pattern**.
- The string being searched in is called the **text**.
- The pattern is referred to with symbol ***P***.
- The text is referred to with symbol ***T***.
- The length of ***P*** is ***n***.
- The length of ***T*** is ***m***.

- An **alignment** of ***P*** to ***T*** is an index ***k*** in ***T*** such that the last character of ***P*** is aligned with index ***k*** of ***T***.

A **match** or **occurrence** of ***P*** occurs at an alignment if ***P*** is equivalent to ***T[(k-n)..n]***.

Algorithm:

BoyerMoore(T,P)

1. n = T.length, m = P.length
2. i = m-1
3. for j=0 to N-1 do
   3.1. delta[j] = m
4. for j=0 to m-1
   4.1. delta[P_j] = m-j-1
5. while (i < n) do
   5.1. j = m-1
   5.2. while(j≥ 0 and $P_j = T_i$)
      5.2.1. i = i – 1
      5.2.2. j = j – 1
   5.3. if (j = -1) then
      5.3.1. return true
   5.4. i = i + max(delta[T_i],m-j)
6. End while
7. Return false
8. Stop

Sample INPUT:

Enter the text:

THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA

Enter the pattern:

SSIPP

Expected output:

Pattern SSIPP is present in text:

THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA

Conclusion: Student can get knowledge and analyzing the Pattern matching problems so that he can implement the different searching algorithms. it is attained with PO1,PO2,PO3,PSO1,PSO2,PSO3 and CO6

Viva Questions:

1. How Boyer – Moore algorithm compares the pattern to the text?
2. the worst – case complexity of BM algorithm?

Experiment No: 12

Aim: - To implement Knuth-Morris-Pratt algorithm for pattern matching.

Description: -

The KMP algorithm compares the pattern to the text in *left-to-right*, but shifts the pattern, *P* more intelligently than the brute-force algorithm. When a mismatch occurs, what is the *most* we can shift the pattern so as to avoid redundant comparisons. The answer is that the largest prefix of *P*[0..*j*] that is a suffix of *P*[1..*j*].

Algorithm:

KMPMatch(T,P)

1. n = T.length, m = P.length
2. Fail = failFunction(P)
3. i = 0, j = 0
4. while(i < n) do
   - 4.1. if (P[j] = T[i]) then
     - 4.1.1. if (j= m-1)then
       - 4.1.1.1. return true
     - 4.1.2. end if
     - 4.1.3. i=i+1,j=j+1
   - 4.2. End if
   - 4.3. Else
     - 4.3.1. If(j>0) then
       - 4.3.1.1. j = fail[j-1]
     - 4.3.2. End if
     - 4.3.3. Else
       - 4.3.3.1. i = i+1

5. End while
6. Return false
7. Stop

failFunction(P)

1. m = P.length
2. i=1,j=0,fail[0]=0
3. while(i< m) do
   3.1. if(P[i]=P[j]) then
      3.1.1. fail[i] = j+1
      3.1.2. i=i+1
      3.1.3. j=j+1
   3.2. End if
   3.3. Else
      3.3.1. If (j > 0) then
         3.3.1.1. j = fail[j-1]
      3.3.2. End if
      3.3.3. Else
         3.3.3.1. fail[i] = 0
         3.3.3.2. i=i+1
      3.3.4. End else
4. End while
5. return fail
6. Stop

Sample INPUT:

Enter the text:

THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA

Enter the pattern:

SSIPP

Expected output:

Pattern SSIPP is present in text:

THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA

Conclusion: Student can get knowledge and analyzing the Pattern matching problems so that he can implement the different searching algorithms. It is attained with PO1,PO2, PO3, PSO1, PSO2, PSO3 and CO6

Viva Questions:

1. How KMP algorithm compares the pattern to the text?
2. KMP algorithm is used for
3. the worst – case complexity of KMP algorithm?

Experiments beyond the Syllabus

Experiment No: 1

Aim: - To implement the Skip List.

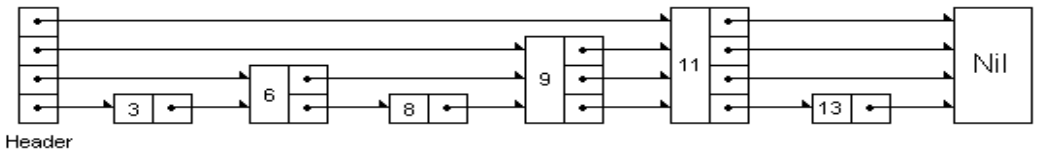
Description: -

Skip lists are made up of a series of nodes connected one after the other. Each node contains a key/value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

Each node will have at least one node reference, and the first reference will always point to the next node in the list. In this way, skip lists are very much like linked lists. However, any additional node references can skip one or more intermediate nodes and point to nodes later in the list. This is where skip lists get their name.

Two nodes that are always present in a skip list are the header node and the NIL node. The header node marks the beginning of the list and the NIL node marks the end of the list. The NIL node is unique in that when a skip list is created, it is given a key greater than any legal key that will be inserted into the skip list. This is important to how skip lists algorithms work.

Skip lists have three more important properties: maximum level, current overall level, and probability. Maximum level is the highest level a node in a skip list may have. In other words, maximum level is the maximum number of references a skip list node may have to other nodes. The current overall level is the value of the node level with the highest level in the skip list. Probability is a value used in the algorithm for randomly determining the level for each node.



Algorithm:

Determining Node Level

The level for each node is determined using the following algorithm:

Hide Copy Code

```
randomLevel()
    lvl := 1
    --random() that returns a random value in [0...1)
    while random() < p and lvl < MaxLevel do
        lvl := lvl + 1
    return lvl
```

Where p is the skip list's probability value and MaxLevel is the maximum level allowed for any node in the skip list.

The node level is initialized to a value of 1. Each time the while loop executes, the level value is incremented by 1. If p is set to a value of 0.5, then there is a 50% chance that the while loop will execute once, a 25% chance it will execute twice, and a 12.5% chance it will execute three times. This creates a structure in which there will be more nodes with a lower level than higher ones.

There is room for optimization here. Suppose the overall level of a skip list is 4 and a value of 7 is returned by the randomLevel algorithm for a new node. Since 7 is larger than 4, the new skip list level will be 7. However, 7 is 3 levels greater than 4. What this means is that when searching the skip list, there will be 2 additional levels that will have to be traversed unnecessarily (this will become more clear when we examine the search algorithm). What is needed is a way to limit the results of the randomLevel algorithm so that it never produces a level greater than one more than the present overall skip list level. Pugh makes a suggestion to "fix the dice." Here is the altered randomLevel algorithm:

Hide Copy Code

```
randomLevel(list)
    lvl := 1
```

```
--random() that returns a random value in [0...1)

while random() < p and lvl < MaxLevel and lvl <= list->level do

   lvl := lvl + 1

return lvl
```

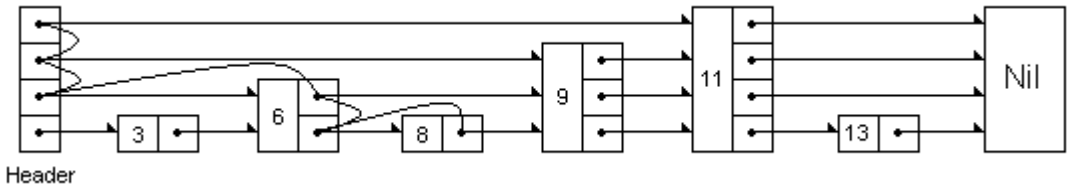
Searching

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the search key. If the node key is less than the search key, the search continues moving forward at the same level. If on the other hand, the node key is equal to or greater than the search key, the search drops down one level and continues forward. This process continues until the search key has been found if it is present in the skip list. If it is not, the search will either continue to the end of the list or until the first key with a value greater than the search key is found.

Hide Copy Code

```
Search(list, searchKey)

   x := list->header

   --loop invariant: x->key < searchKey

   for i := list->level downto 1 do

      while x->forward[i]->key < searchKey do

         x := x->forward[i]

   --x->key < searchKey <= x->forward[1]->key

   x := x->forward[1]

   if x->key = search then return x->value

      else return failure
```

Where forward is the array of node references each node has to nodes further in the list.



Searching for the key with a value of 8.

Inserting

Insertion begins with a search for the place in the skip list to insert the new key/value pair. The search algorithm is used with one change: an array of nodes is added to keep track of the places in the skip list where the search dropped down one level. This is done because the pointers in those nodes will need to be rearranged when the new node is inserted into the skip list.

Hide Copy Code

```
Insert(list, searchKey, newValue)
   local update[1..MaxLevel]
   x := list->header
   --loop invariant: x->key < searchKey
   for i := list->level downto 1 do
      while x->forward[i]->key < searchKey do
         x := x->forward[i]
      --x->key < searchKey <= x->forward[1]->key
      update[i] := x
   x := x->forward[1]
   if x->key = search then x->value := newValue
```

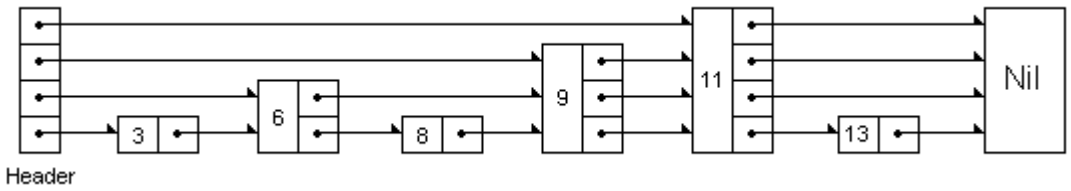
```
else
    lvl := randomLevel()
    if lvl > list->level then
        for i := list->level + 1 to lvl do
            update[i] := list->header
        list->level := lvl
    x := makeNode(lvl, searchKey, newValue)
    for i := 1 to lvl do
        x->forward[i] := update[i]->forward[i]
        update[i]->forward[i] := x
```

The first part of this algorithm should look familiar. It is the same as the search algorithm except that it uses the update array to hold references to the nodes where the search drops down one level. After the search has ended, a check is made to see if the key in the node where the search stopped matches that of the search key. If so, the value for that key is replaced with the new value. If on the other hand, the keys do not match, a new node is created and inserted into the skip list.

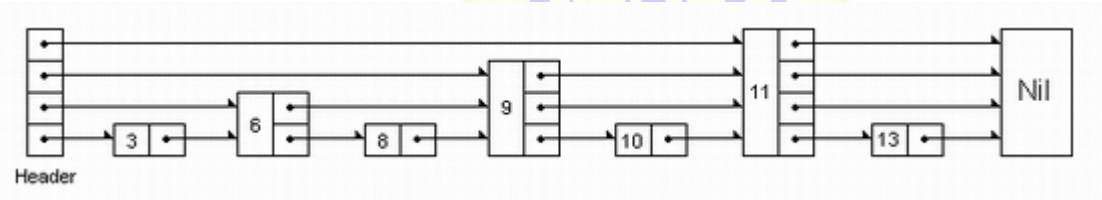
To insert a new node, a node level is retrieved from the randomLevel algorithm. If this value is greater than the current overall level of the skip list, the references in the update array from the overall skip list level up to the new level are assigned to point to the header. This is done because if the new node has a greater level than the current overall level of the skip list, the forward references in the header will need to point to this new node instead of the NIL node. This reassignment takes place during the next step of the algorithm.

Next, the new node is actually created and it is spliced into the skip list in the next for loop. What this loop does is work from the bottom of the skip list up to the new node's level reassigning the forward references along the way. It's much the same as rearranging the references in a linked list when a new

node is inserted except that with a skip list there are an array of references that have to be reassigned rather than just one or two.



The skip list before inserting key 10.



The skip list after inserting key 10.

Deletion

Deletion uses the same search algorithm as insertion; it keeps track of each place in the list in which the search dropped down one level. If the key to be deleted is found, the node containing the key is removed.

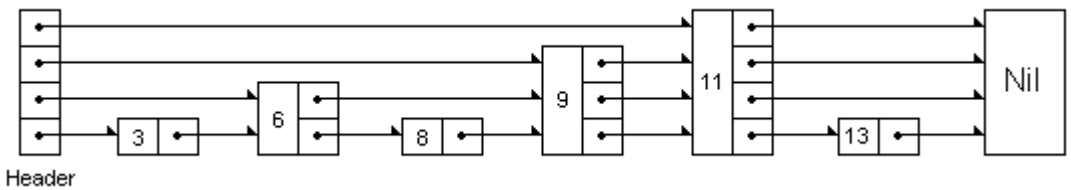
Hide Copy Code

```
Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list->header
    --loop invariant: x->key < searchKey
    for i := list->level downto 1 do
        while x->forward[i]->key < searchKey do
            x := x->forward[i]
```

```
    --x->key < searchKey <= x->forward[1]->key

    update[i] := x

  x := x->forward[1]

  if x->key = searchKey then

    for i := 1 to list->level do

      if update[i]->forward[i] != x then break

      update[i]->forward[i] := x->forward[i]

    free(x)

    while list->level > 1 and

        list->header->forward[list->level] = NIL do

      list->level := list->level - 1
```

After the key is found, the for loop begins from the bottom of the skip list to the top reassigning the nodes with references to the soon to be deleted node to the nodes that come after it. Again, very much like a linked list except that here there are an array of links to nodes further along in the list that must be managed.

Once this has been done, the node is deleted. The only thing left to do is to update the overall current list level if necessary. This is done in the final while loop.



Expected output:

Insert: ----------

1[1]->2[2]->3[3]->4[4]->6[6]->9[9]->11[11]->NIL

Search: ----------

Key = 3, value = 3

Key = 4, value = 4

Key = 7, not found

Key = 10, not found

Key = 111, not found

Search: ----------

1[1] ->2[2]->4[4]->6[6]->11[11]->NULL

Viva Questions:

1. What is the Skip List?
2. How we can perform operations on Skip List?

Experiment No: 2

Aim: - To implement the Floyd's Algorithm

Description:

Starting point: a graph of vertices and weighted edges

Each edge is of a direction and has a length if there's path from vertex i to j, there may not be path from vertex j to i path length from vertex i to j may be different than path length from vertex j to i

Objective: finding the shortest path between every pair of vertices (i → j)

Application: table of driving distances between city pairs

Algorithm:

Input: n — number of vertices

a — adjacency matrix

Output: Transformed a that contains the shortest path lengths

for k ← 0 to n – 1

for i ← 0 to n – 1

for j ← 0 to n – 1

a[i, j] ← min(a[i, j], a[i, k] + a[k, j])

end for

end for

end for

Sample Input:

```
graph [V][V] = { {1, 1, 0, 1},
                 {0, 1, 1, 0},
                 {0, 0, 1, 1},
                 {0, 0, 0, 1}
               };
```

Sample Output:

Following matrix is transitive closure of the given graph

1 1 1 1

0 1 1 1

0 0 1 1

0 0 0 1

Viva-Voce Questions:

1. What is the Spanning tree?
2. What are the techniques for all pair-shortest path algorithms?
3. What is the Floyd's Algorithm Time Complexity?
4. What is the Warshall's Algorithm Time Complexity?