Aravind · V · V
MCA S1 A
Roll no: 24

DS LAB

4 a. Aim: Merge two sorted arrays and store in a third array.

Algorithm

:) Let the two sorted arrays be

a[ ], and b[ ]

m and n are no. of elements of a and b respectively.

i and j are used as index elements for a and b and both are set to zero

i = 0, j = 0

Let res[ ], an array to store merged result.

k -> index of res[ ], k = 0.

:) while (i<m && j<n) { // loop to iterate through array elements.

if (a[i] < b[j]) { // comparing elements to find smaller value.

res[k] = a[i] // storing result in res[ ]

i++, k++; } // increment to move to next index.

```
if (a[i] > b[j]) {          //check if b is smaller
    res[k] = b[j]           // b is stored to res
    j++, k++, }
        }                   // while loop ends
```

) Atleast One element will be left in either of the arrays due to while loop termination.

```
while (i < m)               // see if a has any elements left
    {
    res[k] = a[i];          // those elements are added to res.
    i++; k++; }

while (j<n)                 // see if b has any elements left.
    {
    res[k] = b[j];          // those elements are added to res.
    j++; k++; }
```

) The final array res[] is a sorted and merged array of a and b.

) Display res[]

```
for (k=0; k<m+n; k++)
    { print res[k]
    }
```

4.b Aim:

Implement circular queue - Add, Delete, Display Search Elements

Algorithm

START

.) Initialize the array and the variables

Array, int A[MAX]; max is maximum array size

Beginning of queue, int front = -1;

End of the queue, int rear = -1;

.) Circular Queue - Enqueue (Insertion)

1) Check wheather queue is FULL

if (rear = MAX-1 && front == 0) || front == rear+1)

print " Queue is Full, Insertion not possible"

terminate

2) Else, if queue is NOT FULL,

check if (front == -1)

if true, then set front = rear = 0

A [rear] = a;

a is the value thats being inserted

else,
check if (rear == MAX-1 && front != 0){

if true, set rear = 0

A[rear] = a ; exit

else, set rear = rear + 1 ;

A[rear] = a ; exit

.) Circular Queue Dequeue Algorithm (Deletion)

1) if (front == -1),
display "queue is empty"

2) else, check if (front == rear)

set front == -1 and

rear == -1

//Queue had single element and its deleted

else,
if (front == MAX-1) // element at the end

set front == 0; exit

else,

set front = front + 1, exit.

·) Circular Queue Display Algorithm

1) if (front == -1)

// Queue is Empty

2) else,

set int i, used as an index variable.

for (i = front ; i <= rear, i++)

print A[i]

// Elements are displayed from front till rear.

·) Circular Queue Search an element.

1) if (front == -1)

// Queue is empty

2) set int i, index variable,

set int key, element to search, flag = 0;

3) If Queue is not empty

check

if (A [front] == key)

print "Element found, location = front ;

else , // traverse

for (i = front ; i <= rear, i++)

if (A [i] == key) // check if element = key for each i

4) if (A[i] == KEY)

print " Element found at location i

flog = = 1 ;

5) if (flog == 0)

print " Element not found in the queue"

STOP

4.C Aim: Stack Operation Singly Linked list
Pop, Push, Linear Search

Algorithm

.) Initialize the linked list using structure.

```
struct node {
        int data;      // holdes data value
        struct node *next; //address of next node
        }
```

Also a head value, set it to NULL

```
struct node * head = NULL;
```

Push Operation

1) int data is the value to be pushed.
newnode is the newly initialized node of linked list.

```
if (head == NULL) //stack empty
        newnode -> data = data;
        newnode -> next = NULL;
        head = newnode;
        exit // Item pushed
if head is not NULL
```

newnode -> data = data

newnode -> next = head ;

head = newnode ;

exit // Item pushed

Pop Operation

1) If head == NULL

print "Stack Empty"

.) else ,

Initialize a struct node variable

struct node * temp ;

2) Set head value to temp

temp = head

head = temp -> next

free (temp) ; // first element poped from beginning.

Display

```
if head != NULL

    set head to temp.

    while (temp != NULL)

        print temp->data //stock elements displayed.

        temp = temp->next;
```

Search Operation

1) Initialize key, i=and flag=1, *temp.

```
if head != NULL

    while (temp != NULL)

    { if (temp->data == item)

        print Element found at location
                                    i+1;

                flag = 0;

    i++
    temp = temp->next; }

    if(flag = 1)

        print Element not found.

STOP.
```

4.4 Doubly Linked List Implementation, Insertion, Deletion, Display and Search using C program.

ALGORITHM

START

·) Initialize the node, Let int data : be the data part of the node
struct node *next : as the next part
struct node * prev : as the previous part
both embold respective addresses.

·) Initialize head
struct node *head = NULL;
This always points to first node.

A) Insertion At the Beginning of Linked List:

·) Initialize newnode and allocate memory.

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

newnode -> data = entered value to be inserted

temp is a temporary variable pointer.

PageWork
D D M M Y Y Y Y

·) if (head == NULL){ // list is empty
set

```
newnode -> next = NULL;
newnode -> prev = NULL;
head = newnode;
}
```

else

```
temp = head
```

check if (temp -> prev == NULL){ //confirming temp is first node.

```
newnode -> next = temp;
temp -> prev = newnode;
newnode -> prev = NULL;
head = newnode;
}
```

END.

ie newnode is inserted to the beginning

B) Insertion At the End of Doubly Linked List:

·) if (head == NULL) {

```
newnode -> next = NULL;
newnode -> prev = NULL;
head = newnode;
}
```

```
else        temp = head.
    while (temp -> next != NULL) {
            temp =  temp -> next; }

    if ( temp -> next == NULL) {
            temp -> next = newnode
            newnode -> prev = temp;
            newnode -> next = NULL;
            END                 }
```

Element is inserted at the END.

c) Insertion After the Given Key Value of the Doubly Linked List:

:) Key = key value after which newnode should be inserted

```
if (head == NULL) {
        Key value not present. List Empty }.
```

:) else temp = head.

```
while (temp -> data != key && temp -> next != NULL){
        temp = temp -> next; }
        //Looping through temp to
          find key value or reach end
          of list.
```

PageWork
D D M M Y Y Y Y

:) if (temp -> data == key) {
check if (temp -> next == NULL) {
// if yes, ie end of list.

temp -> next = newnode
newnode -> prev = temp
newnode -> next = NULL
exit

Node Element is inserted after key at the end.

else

newnode -> prev = temp
newnode -> next = temp -> next
temp -> next -> prev = newnode
temp -> next = newnode
exit

Element is inserted after key between two nodes.

:) if (temp -> data != key && temp -> next == NULL) {

Key element not found in the list.
Insertion Failed. }
exit.

D) Deletion At the Beginning of Doubly Linked List.

:) if (head == NULL)

Linked List is Empty, Deletion not Possible.

exit.

else {

temp = head

:) if (temp -> next == NULL && temp -> prev == NULL) {

// only one node in the list.

head == NULL ;

free (temp);

Node Element deleted from beginning. }

exit.

else

temp -> next -> prev = NULL

free (temp)

Node deleted from the beginning

exit.

}

E) Deletion At the End of Doubly Linked List

.) if (head == NULL

List is Empty, Deletion Failed.
exit.

.) else

```
temp = head
while ( temp -> next ) = NULL ) {

        temp = temp->next }

if (temp -> next == NULL ) {

        temp -> prev -> next = NULL;
        free (temp );

        Node Deleted from the end
exit
```

F Delete the node with given key value in a Doubly Linked List.

key = value of the node to be deleted.

:) if (head == NULL)
    Empty List, Deletion Not possible.
    exit().

:) else

    temp = head;
:) if (temp -> next == NULL && temp -> prev == NULL) {
        // only one node
        if (temp -> data == key) {
            head = NULL;
            free (temp);
            Node deleted with element value key.
            exit
        }
    else
        Key value not found in any node. No deletion
        exit

:) if (temp -> prev == NULL && temp -> data == key) {
        // first node
        head = temp -> next;
        temp -> next -> prev = NULL;
        free (temp);
        exit. Node deleted with element value key

·) while (temp -> data != key && temp -> next != NULL) {

temp = temp -> next;

}

// looping until temp data is key or temp next is null.

·) if (temp -> data == key) {

if (temp -> next == NULL) { // last node

temp -> prev -> next = NULL;

free (temp);

exit.

Last node deleted with value key.

else // node is between two other nodes

temp -> prev -> next = temp -> next;

temp -> next -> prev = temp -> prev;

free (temp);

Node deleted with value key

·) if (temp -> next == NULL && temp -> data != key) {

Key value not found

no deletion

}

exit.

}

G) Display Elements in Nodes of Doubly Linked List

```
if (head == NULL)
        Linked list is empty.

else    temp = head
    while (temp != NULL) {

            printf ("%d", temp -> data).
            temp = temp -> next.
                    }
                Exit.
```

H) Search Node values in Doubly Linked List.

Let,
key = value of node to be searched
i = 0

```
if (head == NULL)
            List is empty)

else

    temp = head
    while (temp -> data != key && temp -> next != NULL)
            temp = temp -> next
            i ++
```

if (temp -> data == key)

Node with key value is found
i + 1 is the location.

else

Element is not found in this list.

exit

END /STOP

4.e Aim: Binary Search Tree. Insertion, Deletion Search.

Algorithm:

1) Initalize a node with a data part and two pointer parts to left and right childs of the bst

```
struct node {
        int data;
        struct node * left;
        struct node * right; }
```

~~Also~~

Insert Operation

1) Insertion should be at proper location. Insert data.

```
if (root is NULL
        newnode -> data = data // input data into set newnode data part

    set newnode -> left =
                   newnode -> right = NULL.
```

2) If root has a data value compare

```
if (root -> data > data)
        search for an empty location in
```

the left subtree

if (root -> data < data)

search for an empty location in right subtree. root will be changed to each parent addresses

3) Once an empty location is found.

newnode is set,

newnode -> data = data

newnode -> right = newnode -> left = NULL.

Also, address of newnode is set to parent nodes right or left respectively

ie

parentnode -> right = newnode

or

parentnode -> left = newnode respectively.

Insertion complete exit.

Search Operation

1) Let 'key' be the value to search and locate

2) If root == NULL

print "tree is empty"

else.

Search starts from Root node Search (node, data)

if (data > node->data)

Search right subtree

Search (node->right, data)

if (data < node->data)

Search left subtree

Search (node->left, data)

.) Once data matches node->data

Element is found in the bst.

.) If not and Search returns a NULL.

Element is not found in the bst.

Deletion Operation

Let `data` be the value that need to be deleted from the bst. *temp is a temporary variable.

.) Find node->data value matching data from the tree. Start at root node. Delete (node, data)

.) if (node = NULL)
print Tree Empty.

else,

if (data < node -> data)
Search the left subtree.
node->left = Delete (node->left, data)

if (data > node-> data)
Search the right subtree
node->right = Delete (node->right, data)

.) Once node->data is located, 3 cases exists.

.) if (node->right && node->left) {
// the node has right and left childs
Then find the highest value node of left subtree of current node.

Save that node to temp

```
temp = Left (node -> left)
        Left (node->left) {
        if (node->left right)
            return Left (node -> right)
        else
            return node; // highest node
        }                in left subtree.
```

.) once the highest node of left subtree is located and saved.

```
node -> data = temp -> data // set temp's data value
                               to node data
```

Do the delete operation again with temp->data and node->right

```
node -right = Delete ( node -> right, temp -> data);
        set the return value to node -> right.
}
```

Case 2

if only one child exists for the node.

```
temp = node
if (node -> left == NULL)
        node = node -> right;
else if (node -> right == NULL)
        node = node -> left;
free (temp)
```

case 3 :

Finally when node doesn't have childrens

just delete the node

free (temp);

Traversal Inorder

:) If node, ie rootnode is NULL

Tree empty.

:) else

Inorder (node -> left),

print node -> data

Inorder (node -> right),

STOP

5. Set data structure and set operations, union, intersection and difference using bit vector.

Algorithm

START

1) Initalize two arrays, A[] and B[],
Insert the bit elements inside both arrays.
Let i be index variable, innalize C[], D[] and E[] also.

2) Compare the length of both Array.
Length of first array = A[] length = l1
l2 = Length of second array B[].

if (l1 != l2) then

The sets are not compatible for operations.
Set should be equal in length

End if

3) else
operations can be performed

Union

for(i = 0; i < l1; i++)

C[i] = A[i] || B[i]

OR

End For

·) Print C[], union of sets

Intersection

1) for (i = 0 ; i < li ; i++)

A D[i] = A[i] && B[i] // && = AND

EndFor

·) Print D[], Intersection of sets

Difference

1) for (i=0; i < li, i++)

E[i] = A[i] && ! B[i] // ! = NOT

EndFor

·) Print E[], Difference of those two sets

6) Disjoint Sets and the associated operations (create, union, find)

START

.) Makeset (create)

Make-set (x): Create a new set with only x, assume x is not already in some other set.

```
Makeset (x)
    p[x] <- x        // array is used for implementation
    rank [x] <- 0    // rank is zero initially.
```

.) Find - to determine if two elements are in the same set, by finding which subset a particular element is in.

Find-set (x): return the representative of set containing x.

```
Find-set (x)
    if x ≠ p[x]
        then p[x] <- Find-set (p[x])
    return p[x]
```

.) Union

Join two subsets into a single subset.

.)Union (x, y): two sets containg x and y are combined into one new set. A new representative is selected.

.)Find-set (x), Find-set (y)

To confirm that x and y belong to different sets.

.) Link (x, y)

if rank [x] > rank [y]

then p[y] <- x

else p[x] <- y

if rank [x] = rank [y]

then rank [y]++

Union is done based on rank, higher rank becomes parents.

STOP

10 Graph traversal and topological sorting.

Depth First Search Algorithm:

:) Initialize vertex

```
Struct vertex {
        Char label,
        bool visited; };
Struct vertex *;
```

Each vertex will have a char, and a bool value which is either true or false.

•) vertex->label => A
Vertex -> visited => False.

Also have a stack to store visited vertices.

int stack [MAX], Also initialize adjacency matrix to check edges.
adj [V] [V]

•) Search starts from any node, let it be A.

Mark A as visited = true;

Display A

·) Visit the adjacent unvisited vertex of A. Mark it as visited. Display it and Push it into the stack.

·) If no adjacent unvisited vertex is found, we apply backtracking.

·) Pop up a vertex from the stack untill an unvisited vertex is found as an adjacent vertex.

·) Repeat above two,3 steps till stack becomes empty.
Ones stack becomes empty and no unvisited vertex exist, the program is over.

·) The displayed vertices are the spanning tree, with no loops.

Output

·) addvertex ('S')

addvertex ('A')

addvertex ('B')

addvertex ('C')

addedge (0, 1)

addedge (0, 2)

addedge (1, 2)

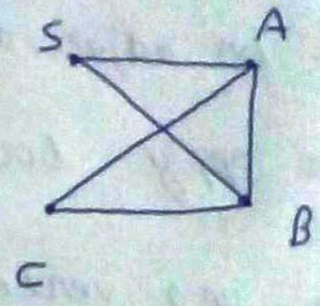
addedge (2, 2)

addedge (1, 3)

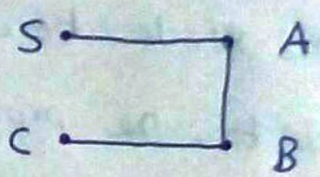
addedge (3, 2)

addedge (3, 3)

Graph



Result graph



Depth First Search : S A B C

Result graph

Breadth First Search

.) Let vertex have a character label and visited boolean part

vertex->label = A

vertex -> visited = false

Initialize a queue

int queue [10]; and adjacency matrix [V][V] to check edges connection of vertices

.) Here search start with any vertex. Let it be A. Display A, mark it as visited.

.) Adjacent unvisited vertex of A are visited. They are marked as vertex -> visited = true; and Displayed : printf("%d", vertex->label);

.) These vertices are then inserted into queue with an index.

.) If no adjacent vertex is found, remove the first vertex from the queue

.) Repeat the above 3 steps ('A' for any vertex) until queue is empty.

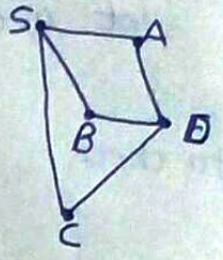
·) The displayed vertices are all the vertices without forming any cycles and in spanning tree format.

Output:

·) addvertex

S A B C D

Graph



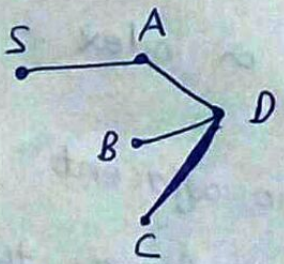
addedges:

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

Breadth first search:

S A B C D

Result graph



·) Topological Sorting:

In this sorting technique for every edge u - v of a directed graph, the vertex u will come before vertex v in ordering.

Graph should be acyclic.

Steps are:

Let G be the graph.

·) Identify a vertex v with no incoming edges.
ie indegree = 0.

·) Add this node/vertex to an ordering, first into a queue[MAX] to keep track of it.

·) Remove this vertex from the graph.

·) Repeat.

Looping goes on until no nodes are left with indegree zero
or
Graphs with incoming edge are still left, ie cyclic graph, print "topological sorting not possible".

·) Vertex in the queue are transfered to a stack topological sort[] along with setting/updating the degree of other vertices.

·) If graph is not cyclic, the topological sort[] stack is printed. This is the topological ordering of graph G.

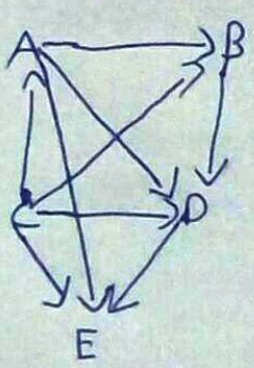
Output

Vertices of Graph:

A
B
C
D
E

Edges: (0, 1)
(0, 3)
(0, 4)
(1, 3)
(2, 0)
(2, 1)
(2, 3)
(2, 4)
(3, 4)

Graph picture:



Topological order of vertices:

C A B D E

12. Prim's Algorithm for finding the minimum cost spanning tree.

Algorithm

Minimum spanning tree - vertices must be connected without the minimum weight edge to make it a Minimum spanning tree.

.) Create a set mstSet that keeps track of vertices already in the MST.

.) Assign a key value to all vertices in the input graph. Initialize all key vertices as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

.) While mstSet doen't include all vertices

a) Pick a vertex u which is not there in mstSet and has minimum key value.

b) Include u to mstSet.

c) Update key value of all adjacent vertices of u.

To update key values iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update key value as weight of u-v.

•) Key values are only used for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weigh edges connecting them to the set of vertices included in MST.

Output

The adjacency matrix of a weighted graph :

G1[V][V] , V = 5

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

↑ vertices

The minimum cost spanning tree edges are :

0 - 1

1 - 2

1 - 3

3 - 4

Total cost = 5 + 1 + 6 + 2 = 14

13. Kruskals algorithm.

Used to find minimum cost spanning tree.

Minimum cost spanning tree has (v-1) edges where v is the number of vertices in the graph.

:) A minimum cost spanning Tree T is build from Graph G by adding edges to T one by one.

.) Select the edges for inclusion in T in decreasing/ increasing order of the cost.

.) An edge is added to T if it does not form a cycle.

-) Since G is connected and has $n>0$ vertices, exactly $n-1$ edges will be selected.

Steps:

1. Sort all edges in increasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat setep 2 unttll there are (n - 1) edges in the spanning toree.

·) It wes disjoint set data structure to maintain several disjoint sets of element.

·) Eoch set contain te vertices in one toree of te owrent forest.

·) Operation FIND-SET(U) returns a representative element from te set that contain U.

·) So wheather U ond V belong to some toree can be determined by FIND-SET(U) equals FIND-SET(V)

·) To combine toreeg, Kruskal's olgorithm colls the UNION Procedure.

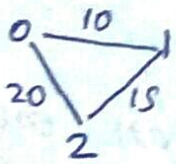
```
algorithm Kruskal (G) is
    F := ∅
    for each v ∈ G.V do
        Make-SET (v)
    for each (u, v) in G.E ordered by weight (u, v)
        increasing
    do  if FIND-SET (u) ≠ FIND-SET (v) then
            F := F ∪ {(u, v)}
                UNION (FIND-SET (u), FIND-SET (v))
    return F.
```

Output:

:) Adjacency matrix of a graph

	0	1	2
0	0	10	20
1	10	0	15
2	20	15	0

Graph



Minimum cost Spanning tree:

No of edges : 2

edges: (1, 2) = 10

edge (2, 3) = 15

Total cost = 25