Deleting ~~into~~ from a Heap

23/7/16

```
i = n-1;
temp = A[i]
A[i] = A[0];
k = 0;
if (i == 1)
    j = -1;
else
    j = 1;
if (i > 2, && A[2] > A[1])
    j = 2;
while (j > 0 && A[j] > temp)
{
    A[k] = A[j];
    k = j;
    j = (2 * k) + 1
    if (j+1 <= i-1 &&
            A[j+1] > A[j])
        j++;
    if (j > i-1)
        j = -1;
}
A[k] = temp;
n--;
```

In Deletion elements are arranged in sorted Ascending order for Max heap and in sorted Decending order for min heap

21/7/16

Heap UNIT – III

- Heap is a complete binary tree or an almost complete binary tree, having all the parent node values are either greater (or) ~~(lesser)~~ lesser than its children.

Almost Complete binary tree :-

- It is a binary tree which satisfies the following two properties.

1) A node must have a left child, in case if it has a right child.
2) If the height of the tree is h, then all the leaf nodes must be at the level h (or) h-1

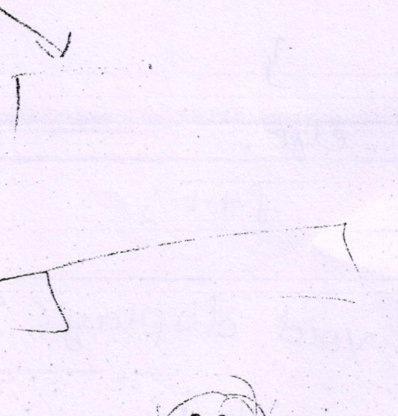
Types of Heaps.

1) Max Heap :- Parent node values are greater than its children
2) Min Heap :- Parent node values are lesser than its children

Inserting into a Heap (Max Heap)

```
insert (int n, int x, int A [ ])
{
  // n no. of existing elements in the Array.
  // x. elements to be added;
    int i = n;
    while (i>0 && A[(i-1)/2] < x)
      {
        A[i] = A[(i-1)/2];
        i = (i-1)/2;
      }
    A[i] = x; ✓
}
```

```
else {
  for (i=f ; i<=r ; i++){
      pf ("%d ", q[i]);
      }
   }
}

void main() {
 int a;
 pf ("Enter elements");
 sf ("%d", &a);
 enqueue (a);
 display();
 dequeue();
 display();
 }.
```



21/7/16

– Heap
binar
either
Almos
– It is
two
1) A
a
2) If
node
Types of
1) Max
2) Min
Inserti
insert (
{
// n
// x
ir
While

ich :

Ascending Order Priority Queue:

```
Void enqueue (int x)
{
  int j ;
  if (f == -1)
   { f++ ; }

  j = r ;
  while ( j >= 0 && Q[j] > x)
   {
    Q [j+1] = Q[j] ;
     j-- ;
   }
  Q[j+1] = x ;
  r++ ;
}.
```

Here:-

f = -1

~~r = 0~~

r = 0

les .

```
void dequeue () {
 if (f == -1) {
     pf ("can't delete);
   }
 else
    f++ ;

void display( ) {
  int i ;
 if (f == -1) {
     pf ("no elements to display") ;
   }
```

there as

ment .

all the

ocessor

on the

20/7/16

Priority Queue:

It is a set of ordered elements in which each element is associated with same priority.

we have two types of Priority Queues.

1. Ascending order Priority Queue.
2. Decending order Priority Queue

1. Ascending order Priority Queue.

In this insertions can happen in any order where as we can always remove only the smallest element in the Queue.

2. Decending Order Priority Queue:-

In this insertions can happen in any order where as we can always remove only the biggest element in the queue.

Applications:-

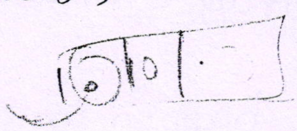
- It is used in the CPU scheduling in which all the Processes are assigned priorities and the processor will be allocated to the processes based on the priority order.

```
temp = temp -> right ;
if (temp == head)
return;
Pf("%d", temp -> data);
}
temp = temp -> right;
}
}

void main ()
{
create ();
Pf("\n inorder non recursive : ");
inordernonrec(root);
}
```

};

Temp
2000
3000
5000
3000
6000
2000
4000
7000
4000
8000
1000

10, 20, 30, 40, 50, 60, 70

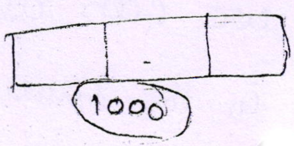


19/7/16

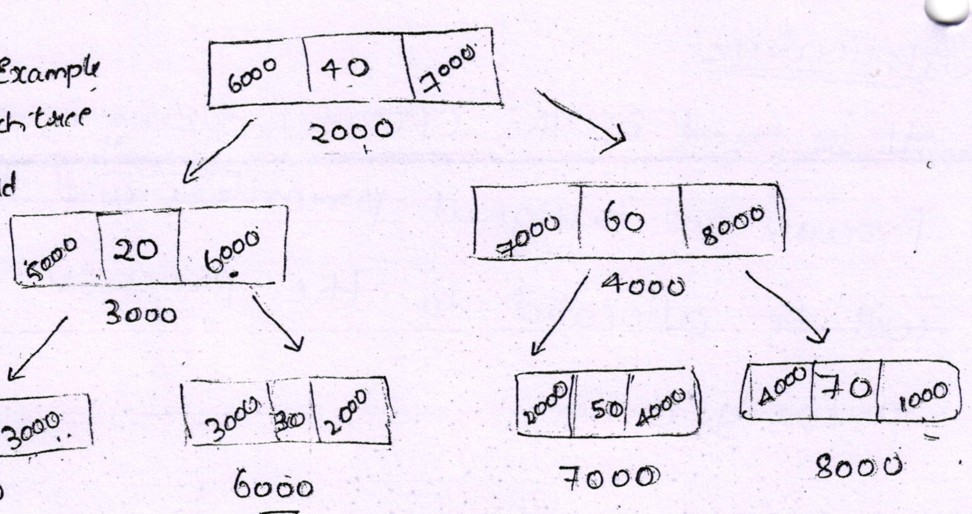
LNR

Inorder Threaded Binary Trees.

thead node



→ This is also Example for binary search tree i.e., left child should have less value than the parent node and right child should have ≥ value of Parent node.



10, 20, 30, 40, 50, 60, 70

```
if (root == NULL)
  {
  root = nn;
  head = (struct tree *)malloc ( sizeof (struct tree));
  head -> left = root;
  head -> data = -999;
  root -> left = head;
  root -> right = head;
  }
else
  insert (root, nn);
pf ("\n do you wish to continue (y/n)");
getchar();
ch = getchar();
}
while (ch == 'y');
}
void inordernonrec (struct tree *temp)
{
 while (temp != head)
 {
 while (temp -> haslchild == 1)
        temp = temp -> left;
   pf("%d", temp -> data);
 while (temp -> hasrchild == 0)
 {
```

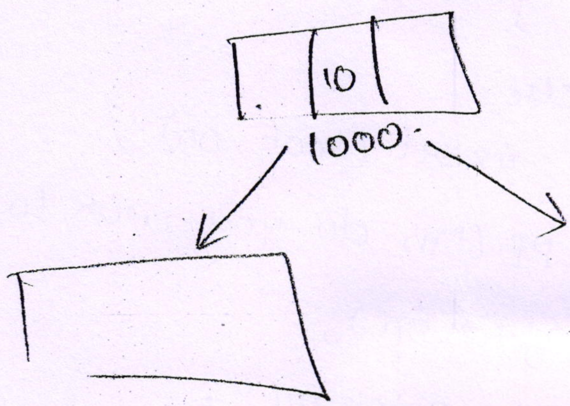
```
tc
if
r(
Pf
}
temp
}
}
void r
{
Creat
Pf("\
inord
}
```

19/7/16

Inor.

-> This is
for hi
i.e., left
have les
the parent
right chil
have >= va
Parent
node ..

```
else
  insert (temp->left, nn);
  }
else
 {
  if(temp-> hasrchild ==0)
   {
    nn->right = temp->right;
    nn-> left =temp;
   temp-> hasrchild =1;
   temp-> right=nn;
   }
else
  insert (temp->right,nn);
  }
}
```




```
void create()
 {
 struct tree *nn;
 char ch; int x;
 do
  {
  pf("enter data");
  sf ("%d", &x);
  nn = (struct tree*)malloc(sizeof(struct tree));
 nn -> left = NULL;
 nn-> haslchild=0;
 nn -> data = x;
 nn->hasrchild =0;
 nn -> right=NULL;
```

lata);

Inorder
Threaded Binary Trees

```
# include <stdio.h>
# include <stdlib.h>

struct tree
{
struct tree *left;
int haslChild;
int data;
int hasrChild;
struct tree * right;
}* root = NULL, *head = NULL;

void insert (struct tree *temp, struct tree *nn)
{
char ch;
pf("\n inser to the left or right of %d", temp->data);
getchar();
sf("%c", &ch);
if (ch == '1')
 {
  if (temp -> haslchild == 0)
   {
    nn -> left = temp -> left;
    nn -> right = temp;
    temp -> left = nn;
    temp -> haslchild = 1;
   }
```

```
// after visiting LR subtrees
while (st[top]. check ==0)
{
temp = st[top].addr;
pf("%d", temp->data);
top--;
if (top = -1)
  return;
}
// after visiting left subtree
temp= st[top]. addr;
temp =temp→ right;
st[top]. check=0;
} // closing of infinite while loop
}
```

18/7/16

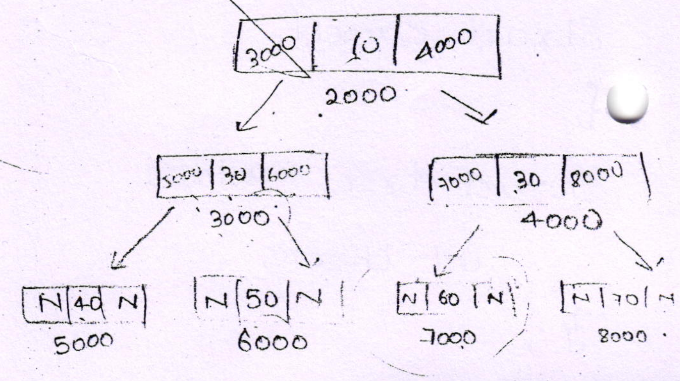
Post order Iterative Method

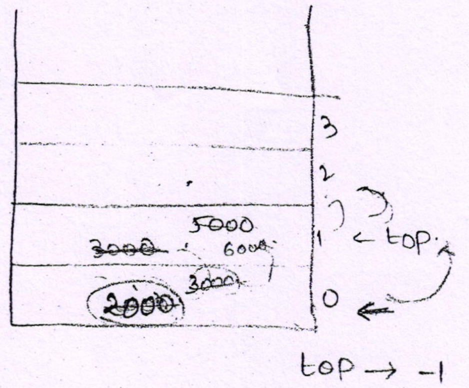
```
Struct tree
{
Struct tree *left;
int data;
Struct tree * right;
};
Struct element
{
Struct tree *addr;
    int check;
};
Struct element St[20];

Void Ipost order ( )
{
Struct tree *temp = root;
While(1)
{      // for each new node
While (temp != NULL)
 {
  top ++;
  St [top]. addr = temp;
  St [top]. Check = 1;
  temp = temp -> left;
 }
```

```
pf("%d", temp->data);
    top++; st[top]=temp;
    temp = temp->left;
  }
 if (top == -1)
    return;
 else{
   temp = st[top]; top--;
    temp = temp->right;
  }
 }
}.
```

Inorder : 40, 20, 50, 10, 60, 30, 7

~~Preorde~~ : 10, 30, 40, 50, 30, 60, 70.





Preorde : NLR

temp → ~~2000~~

16/7/16

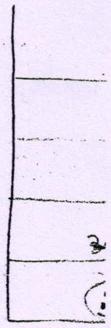
Without Recursion. (Iterative Travelsal Approach)

```
Void inorder( )
{
Struct tree *temp = root;
Struct tree *St[20]; int top=-1;
while (1)
{
while (temp != NULL)
{  top ++; St[top] = temp;
   temp = temp -> left;
}
If (top == -1)
     return;
else {
   temp = St[top]; top--;
   Pf("%d", temp->data);
   temp = temp -> right;
   }
 }
}
```

->

```
Void Preorder( )
{
Struct tree *temp = root;
Struct tree *St[20]; int top=-1;
while(1)
{
while (temp != NULL)
{
```

Pf(
}
If (
else
ter
t
}
}
}.



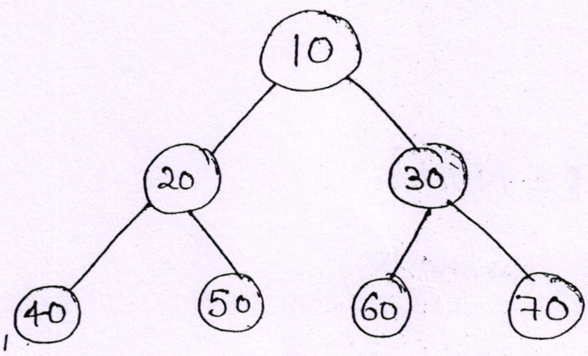
```
newnode = (Struct tree *) malloc (Sizeof (Struct tree));
newnode -> left = NULL;
newnode -> data = x;
newnode -> right = NULL;
 if (root == NULL)
      root = newnode;

 else
    insert (root, newnode);

 pf ("do you wish to contine (y/n)");
 getchar();
 ch = getchar();
 }
 while (ch == 'y');
 }
void Preorder (Struct tree * temp)
{
 if (temp != NULL)
  {
   pf ("%d", temp->data);
   Preorder (temp->left);
   preorder (temp->right);
  }
 }
                    void main() {
                    create ();
                    Preorder (root);
                    inorder (root);
                    Postorder (root);
                    }
```

```
Void insert (Struct tree *temp, Struct tree *newnode)
{
  Char ch;
  Pf (" insert to the left or right of %d", temp->data);
  getchar( );
  Ch = getchar( );
  if (Ch == 'l')
 {
   if (temp->left == NULL)
        temp->left = newnode;
    else
       insert (temp->left, newnode);
  }
else {
    if (temp->right == NULL) temp->right = newnode;
    else insert (temp->right, newnode);
  }
}

Void Create ( )
{
  Struct node * newnode;
  Char ch; int x;
  do
  {
  Pf (" Enter data");
  Sf ("%d", &x);
```

~~Tree~~ Traversing

(i) Tree Traversal Method / Technique.

1) Pre order : (Node), left Tree, Right tree.
2) In order : L (N) R.
3) Post order : L R (N)



Preorder : 10, 20, 40, 50, 30, 60, 70

Inorder : 40, 20, 50, 10, 60, 30, 70

Postorder : 40, 50, 20, 60, 70, 30, 10.

Binary Tree :- (Recursive Approach)

```
# include <stdio.h>
# include <stdlib.h>

struct tree   → Node
{
struct tree *left;
int data;
struct tree *right;
} * root = NULL;
```

Tre

(i)

Preor

Ino

Posto

Binc

inc
incl

Struct
{
Struc
int
Stru
} *

Height = (3) ⇒ h.

No. of nodes in a complete binary tree = $2^{h+1} - 1$

$= 2^4 - 1$

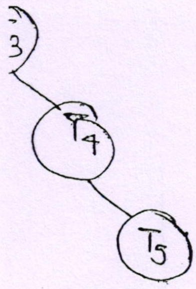
$= 16 - 1$

$= 15$

no. of leaf nodes = 2^h

$= 2^3$

$= 8$



14/7/16

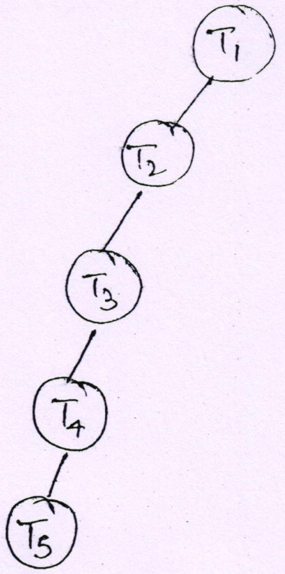
Binary Tree ADT.

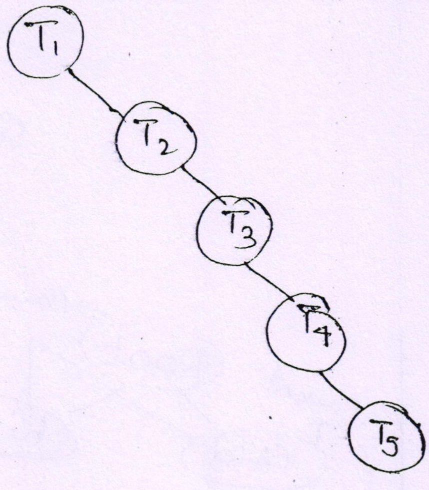
```
Struct
{
instances :
    Struct tree *root ;

operations :
    create();
    insert();
    delete();
    Search();
    display();
}
```

Applications of Binary Trees :

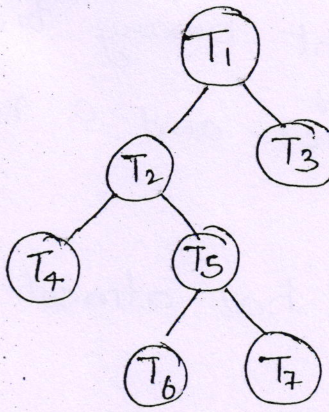
- Binary Trees are used to construct expression trees used in evaluation of expressions by compiler.
- Tree structure is used by operating systems to organise directories and files (application of Trees not only for binary trees).
- Binary Trees are used in developing binary search trees.

⇒ Left Skewed Binary Tree: ⇒ Right Skewed Binary Tree.



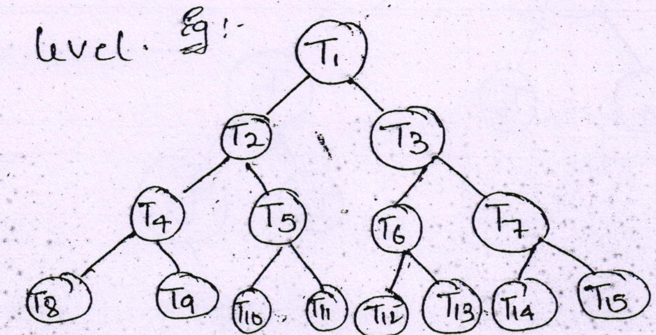


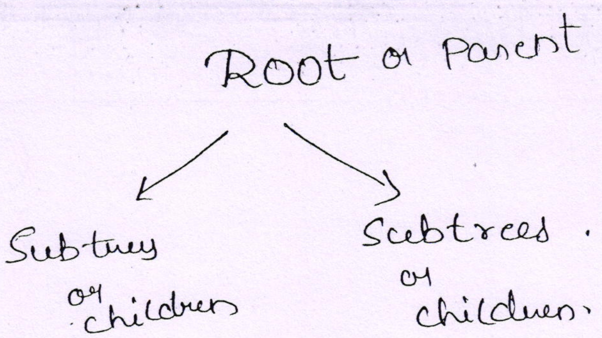
Full Binary Tree :- It is a binary tree and has the property of any node can have either 2 children or no children but not one children.

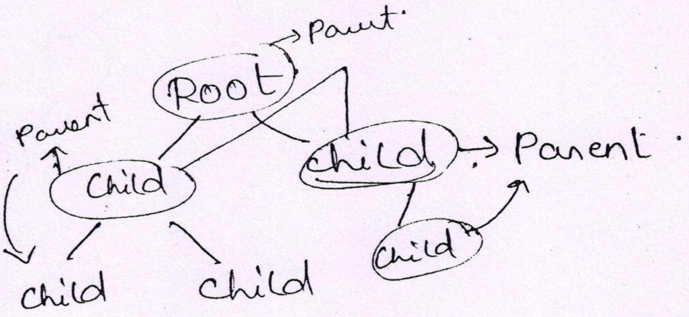


Complete Binary Tree :-

It is full binary tree in which all the leaf nodes are at same level. Eg:-



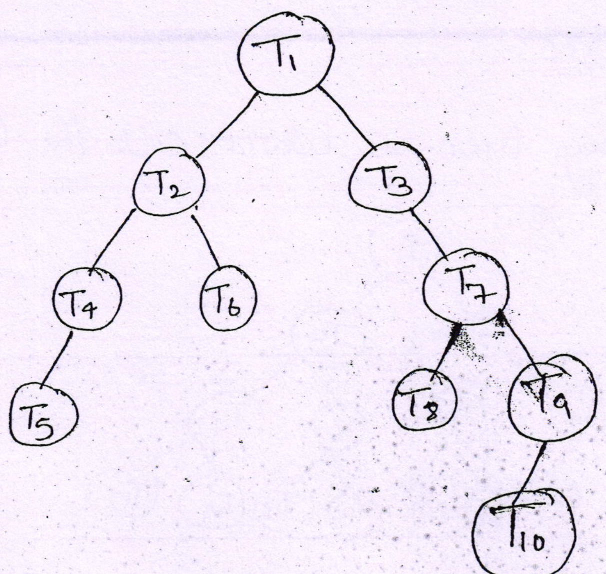




Binary Tree.

- Binary tree is a finite set of nodes which is either NULL or having one root node and a left binary tree, a right binary tree which are also called as left sub binary tree and a right sub binary tree. (Or)
- A tree in which everynode has atmost outdegree (children) '2'

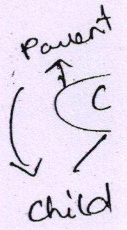
Ex:-



13/7/16

TREES

- Tree is a finite set of nodes having
  - (i) A specially designated node called root of the tree.
  - (ii) All other nodes can be Partitioned into disjoint sets $t_1, t_2, t_3 \ldots\ldots t_n$ (n disjoint sets).
- Each of this sets is a sub tree of the given tree

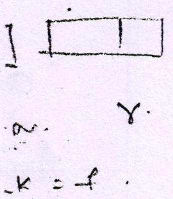


Tree Terminologies :-

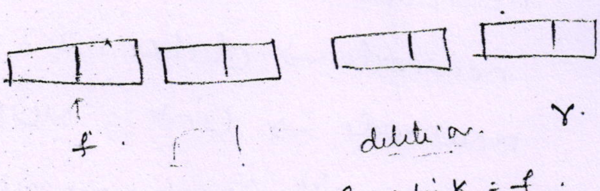
- root
- Parent node.
- child node.
- sibling
- Degree of node / Tree.
- level of the Tree.
- Height / Depth of the tree.
- Internal nodes
- external nodes / leaf nodes.
- Predessor & Successor:-

* A tree is a finite set of one or more nodes.

```
void delete_r( )
{
Struct node * temp
 if (front == NULL){
  pf ("queue Empty ");
 }
elseif (r == f) {
     temp = r;
     r = f = NULL;
   free(temp);
  }
else {
    temp = f;
    while (temp -> link != r) {
              temp = temp -> link;
         }
     temp -> link = NULL;
       r = temp;
   }
  }

void delete_f () {
     Struct node * temp;
       temp = f;
       f = f -> link;
       free (temp);
     }
```



```
void insert_r( )
{
Struct node* newnode;
int r ;
newnode = (Struct node*) malloc ((Size of (Struct node)));
newnode → data = x ;
newnode → link = NULL ;
if (f = = NULL)
  {
  f = newnode ;
  }
else
  {
  r → link = newnode ;
  }
  r = newnode ;
  } ;
```




```
void insert_f ( )
Struct node *newnode ;
newnode =
  ____
  ____
if (f = = NULL){
    f = r = newnode;
  }
else
  {
  newnode → link = front ;
  f = newnode ;
  }
}
```

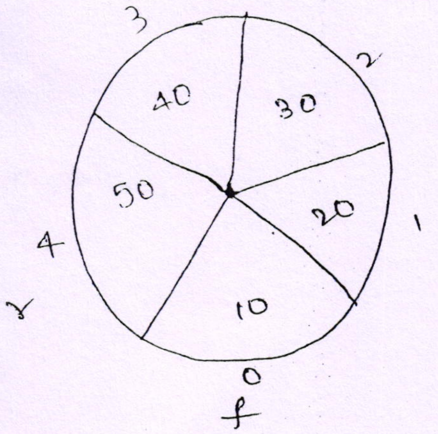
DeQueue (Linked List)

```
# include <Stdio.h>
# include <Stdlib.h>

Struct node{
int data;
Struct node * link;
}* rear = NULL ,  * front = NULL ;

void Create ()
{
Struct node *newnode,* Prevnode;
int x,  char choice;
do {.
  pf.("Enter the Numbers");
  sf(" %d", & x);
newnode = (struct node*) malloc (sizeof (struct node));
newnode -> data = x;
newnode -> link = NULL;
      if (root == NULL)
            root = new node;
      else
         Prevnode -> link = newnode;
   Prevnode = newnode;
  Pf(" do you wish to Continue (y/n)");
  getchar();
  choice = getchar();
  }
  While (Choice = = 'y' );
  }
```

```
void display( ) {
 Struct node *temp = front ;
 While ( temp! = NULL ) {
    pf(" %d ", temp ->data) ,
     temp = temp -> link ;,
   }
   }.
```

),

Circular Queue.



→ In circular queues to known whether the queue is full. rear should be always followed by front.

→ If rear is not followed by front then we can move rear by (R+1) % size

```
# define SIZE 5
void enqueue (int x)
{
 if ((r+1) % SIZE == f)
   pf ("CQ full");
 else {
   if (f == -1)
     { f = r = 0 ;
     }
   else
     {
     r = (r+1) % SIZE ;
     }
   cq [r] = x ;
 }.
```

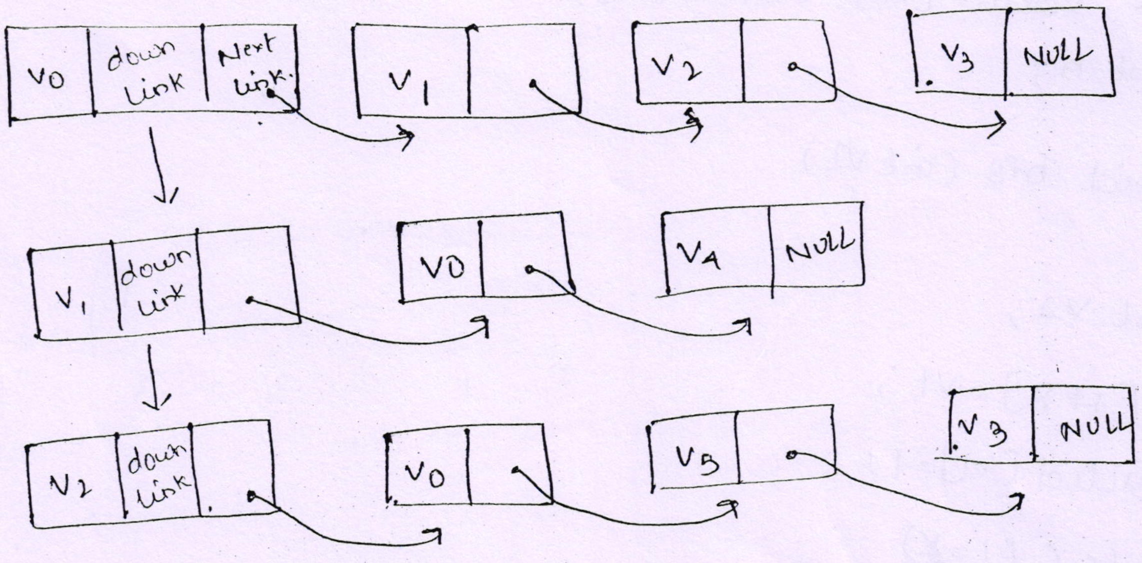
23/7/16

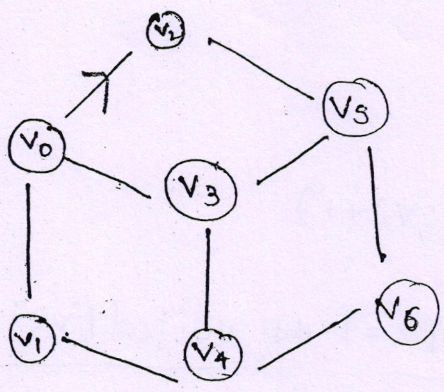
GRAPHS

Adjacency Matrix representation:

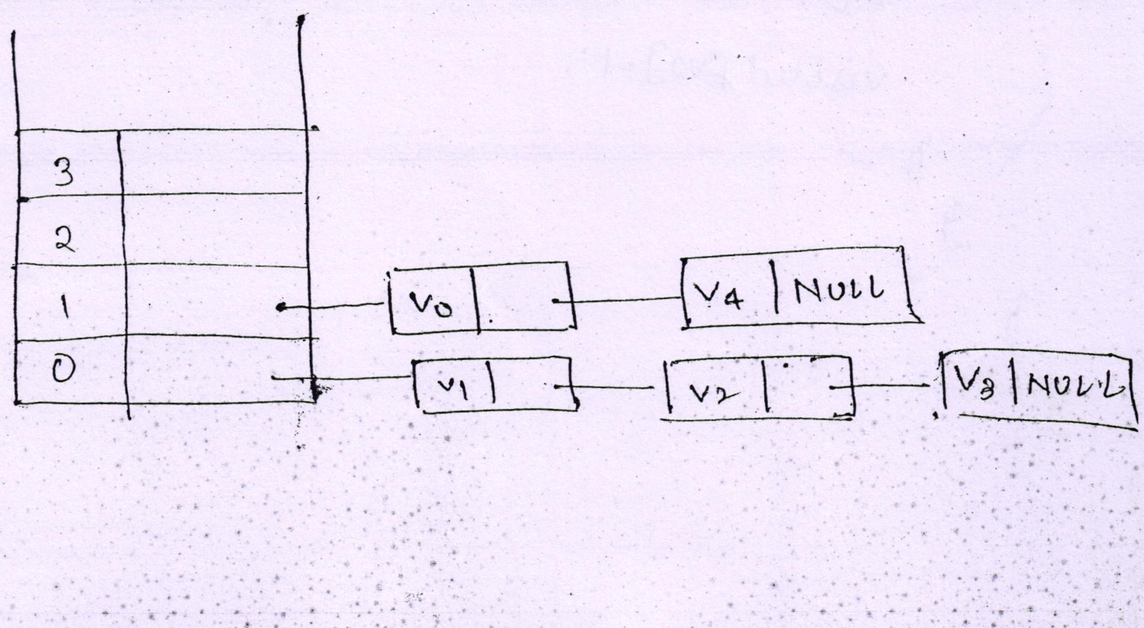
Adjacency link Representation (linked list):

1st approach:





2nd approach:



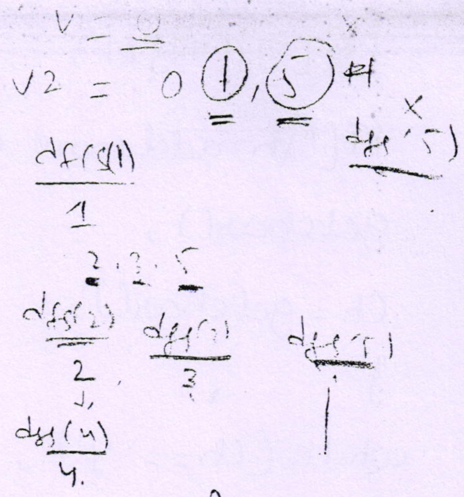
Traversing Graph Using BFS and DFS :

(breadth first Search) (Depth first Search)

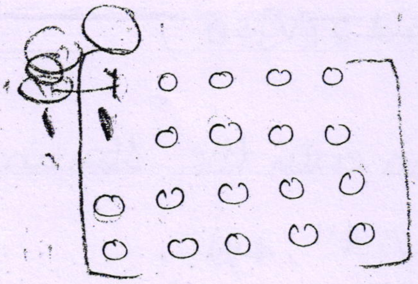
```
#include <stdio.h>
int Q[20], f=-1, r=-1;
int G[20][20];
int visited[20], visited2[20];
int n;

void bfs (int v1)
{
int v2;
Q[++r]=v1;
visited[v1]=1;
while (f!=r)
{
v1=Q[++f];
pf("%d", v1);
for (v2=0; v2<n; v2++)
{
if (G[v1][v2]==1 && visited[v2]==0)
{
Q[++r]=v2;
visited[v2]=1;
}
}
}
}
```

DFS

Depth first Search



```
void dfs(int V1)
{
 int V2;
 pf("%d", V1);
 visited2[V1]=1;
 for(V2=0; V2<n; V2++)
  {
   if(G[V1][V2]==1 && visited2[V2]==0)
    {
     dfs(V2);
    }
  }
}

void main()
{
int V1, V2, V;
char ch;
pf("enter the number of vertices :");
sf("%d", &n);
//initializing the adjacency matrix G of the graph to 0
 for(V1=0; V1<n; V1++)
 for(V2=0; V2<n; V2++)
  G[V1][V2]=0;
```



```
 pf("\n enter the edges details:");
 do
 {
 pf("\n enter source vertex and destion vertex");
 sf("%d %d", &V1, &V2);
```

```
G[v1][v2]=1;
pf("\n add more edges (y/n)");
getchar();
ch = getchar();
}
while ((ch=='y');

pf ("\n the adjacency matrix for the graph is :\n\n");
for (v1=0; v1<n; v1++).
{
for(v2=0; v2<n; v2++)
{
pf(" %d", G[v1][v2]);
}
pf("\n");
}
// intializing visited status to not visited for all the
   vertices'
for (v=0; v<n; v++)
{
visited [v] =0;
visited 2 [v]=0;
}
pf ("\n enter the starting vertex to transverse the graph:");
sf(" %d", &v);
pf ("\n traversing using bfs : \n");
bfs (v);
pf ("\n traversing using dfs : \n");
dfs (v);
pf("\n");
}
```

Edges
v1 v2
0 - 1
0 -
1 - 2
1 - 3
1 - 5
2 - 4
4 - 3
5 - 6
6 - 8
7 - 3
7 - 8
8 - 10
9 - 7
10 - 9

DFS
BFS