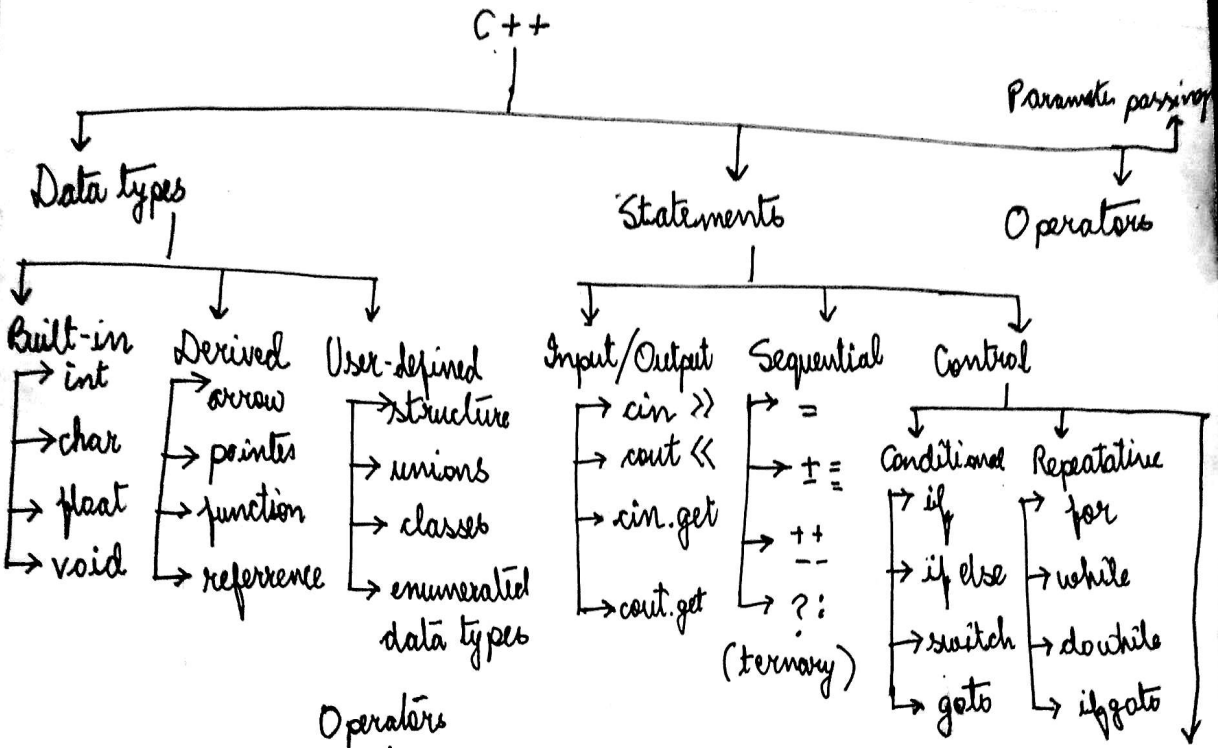
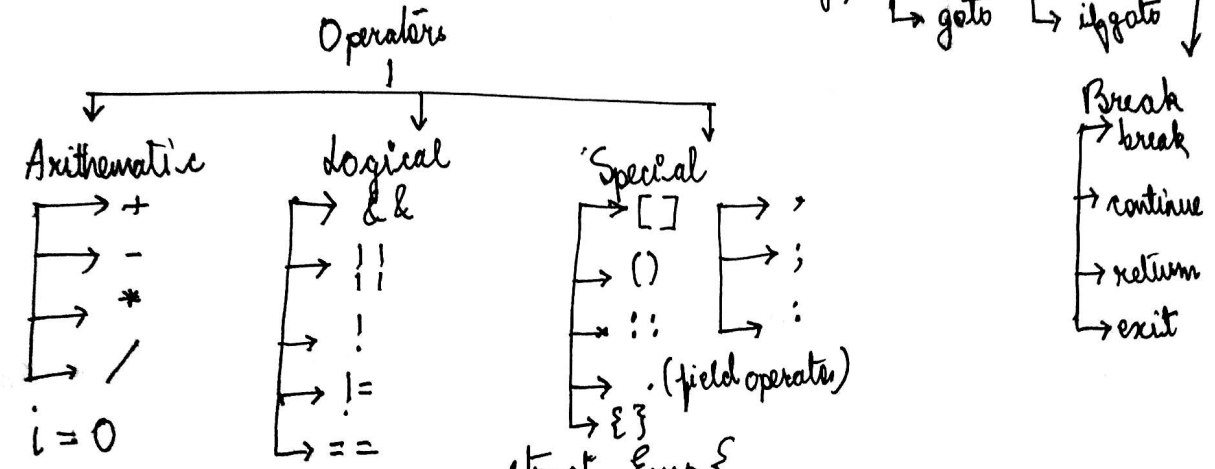
INDEX

NAME: Shibam Basu STD.: B.Tech SEC.: A ROLL NO.: 157153 SUB.: DSA

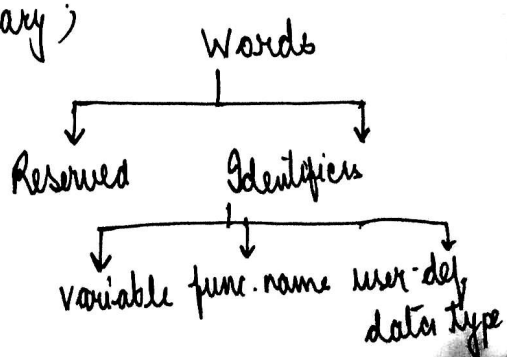
S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		6-2-1		

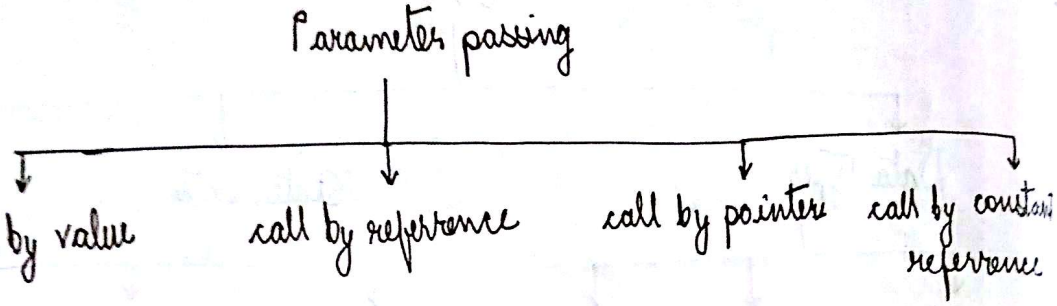





```
i = 0
while (1)
{
   i++;
  if(i==8) continue;
  cout << if(i>10) break;
  cout << i;
}
```

```
struct Emp {
  char emp_name[20];
  int emp_id;
  float salary;
  };
```






```
# main ()
{
  int a, b;
  a = 3; b = 5;
  cout << a << b;
  swap(&a, &b);
  cout << a << b;
}
```

```
swap (int **p, int **q)
{
  int temp;
  temp = *p;
  *p = *q;
  *q = temp;
}
```

```
void swap (int& p, int& q)
{
  int temp;
  temp = p;
  p = q;
  q = temp;
}
```

→ Calender program (use 10 statements in a function) Write for 2016.

→ Parameter passing (all 3)

Mon Tue Wed Thu

Mon
Tue
Wed
Thu

Horizontal Calender

```
#include<iostream>
using namespace std;
{
 int month[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
 int d,m,y,val;
 string s[]={" ","January","February","March"...};
 cout<<" Enter year "<<endl;
 cin>>y;
 if(((y%4==0)&&(y%100!=0))||(y%400==0))month[2]=29;
 y--;
 val=(y+y/4-y/100+y/400+1)%7;
 for(int i=1;i<=12;i++)
 {
  cout<<s[i]<<endl;
  cout<<" Sun Mon Tue Wed Thu Fri Sat "<<endl;
  for(int space=0;space<val;space++) cout<<"  ";
  for(int days=1;days<=month[i];days++,val++)
   {
     if(val==7) { val=0;
                  cout<<endl;
                }
     if(days<10) cout<<days<<"   ";
     else cout<<days<<"  ";
   }
  cout<<endl;
 }
return 0;
}
```

Vertical Calender

```
string month[] = {" ","JAN","FEB","MAR"... };
string days[] = {" ","MON","TUE","WED"... };
int daysinmonth[13] = {0,31,29,31,30,31,30... };
int i, j, temp, val, firstday = 5;
for(i=1; i<=12; i++)
{
  val = 9 - firstday;
  cout << month[i] << endl;
  for(j=1; j<=7; j++, val++)
  {
    cout << days[j] << "    ";
    temp = val - 7;
    if (temp > 0)
    {
      while (temp <= 31)
      {
        cout << temp << "    ";
        if (temp == daysinmonth[i]) firstday = j+1;
        temp += 7;
      }
    }
    else {
      temp += 7;
      cout << "    ";
      while (temp <= 31)
      {
        cout << temp << "    ";
        if (temp == daysinmonth[i]) firstday = j+1;
        temp += 7;
      }
    }
  }
  cout << endl;
}
```

Demonstration of Call by Reference -

```
#include <iostream>
using namespace std;
int main()
{
    int a = 3, b = 5;
    cout << a << b << endl;
    swap(a,b);
    cout << a << b << endl;
    return 0;
}
void swap (int& x, int& y)
{
    int t;
    t = x; x = y; y = t;
}
```

Demonstration of Call by Pointer

```
#include <iostream>
using namespace std;
void swap(int *, int *);
int main()
{
    int a = 3, b = 5;
    cout << a << b << endl;
    swap(&a, &b);
    cout << a << b;
    return 0;
}
void swap(int *x, int *y)
{
    int t;
    t = *x; *x = *y; *y = t;
}
```

Call By Value :-

```
#include<iostream>
using namespace std;
void swap(int, int);
int main()
{
    int a = 3, b = 5;
    cout << a << b << endl;
    swap(a, b);
    cout << a << b << endl;
}
void swap(int x, int y)
{
    int t;
    t = x; x = y; y = t;
}
```

Call by reference and Pointer

```
#include<iostream>
using namespace std;
void swap(int &, int *)
int main()
{
    int a = 3, b = 5;
    cout << a << b << endl;
    swap(a&b);
    cout << a << b << endl;
}
void swap(int &x, int *y)
{
    int t;
    t = x; x = *y; *y = t;
}
```

Recursion

Any recursive program should have 3 statements

1. Termination statement
2. Self-call
3. Change

```
for(i=1; i<=10; i++)
  s = s+i;
cout << s;
```

#
```
int sum(int i)
{
  if (i >=10) return (i%10 + sum(i/10));
  else return i;
}
```

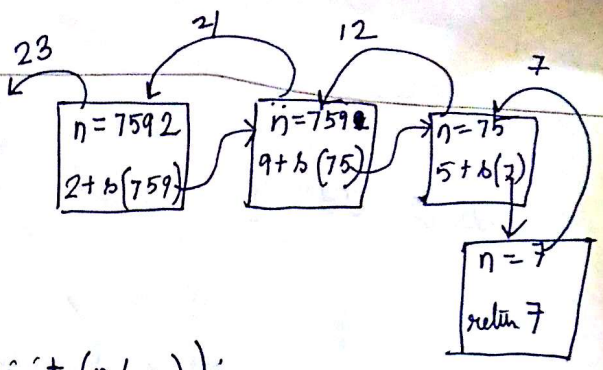
#
```
void sum (int i)
{
  static int s = 0;
  if (i>10) cout << s;
  else { s = s + i;
         sum(i+1);
       }
}
```

```
int sum(int i)
{
  if (i == 10) return i;
  else
  {
    return i + sum(i+1);
  }
}
```

#
```
void sum(int i, int &s)
{
  if (i>10) return;
  else
  {
    s = s+i;
    sum(i+1, s);
  }
}
```

Printing sum of digits

```
int sumofdigit (int n)
{
  if (n < 10) return n;
  else return (n%10 + sumofdigit (n/10));
}
```

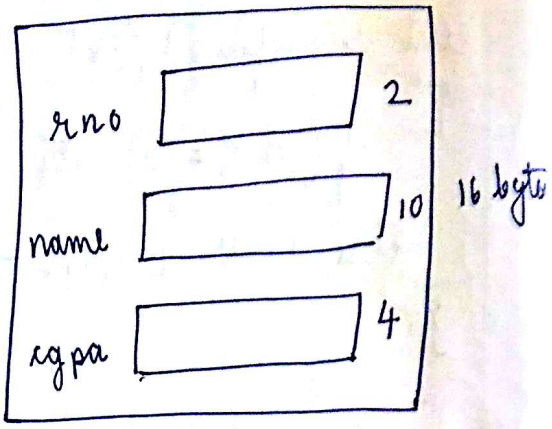



```
void print (int n)
{
  if (n < 10) cout << n;
else {
    print (n/10);
    cout << n%10;
  }
}
```

Recursive function to find min, max, sum of element of array

```
single (int a[],
```

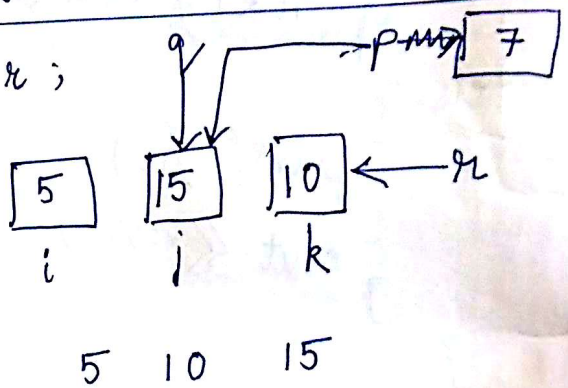
```
struct std {
        int rno;
        char name[20];
        float cgpa;
};
struct std s;
```



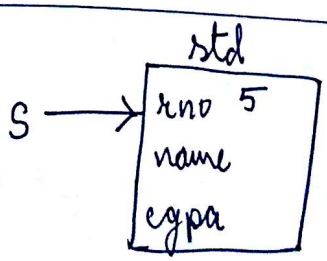
No bytes are allocated while defining structure.

`int i=5, j, k, *p, *q = &j, *r;`

```
*p = 7;
 p = &j;
*p = i;
 p = q;
 r = &k;
*r = *p + *q;
*q = k + i;
cout << i << *r << *q;
```



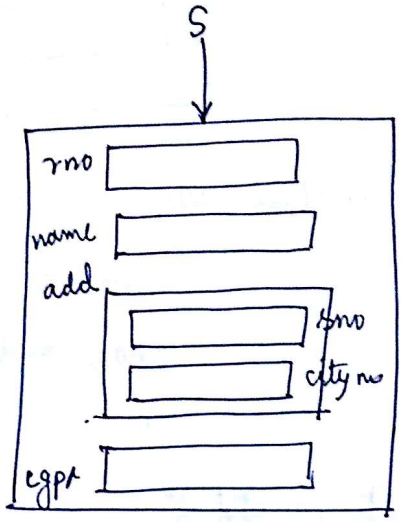
```
struct std
{
   int rno;
   char name[10];
   float cgpa;
};
struct std *s;
 s = new std;
```

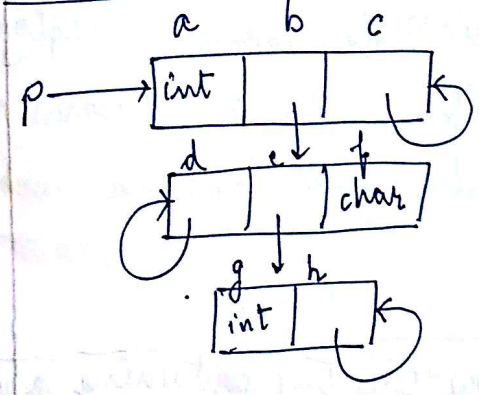


`s → rno = 5`

Three semi-colons three statements.

```
struct add{
        int street ;
        int cityno ;
      };
struct std {
        int rno;
        char name [10] ;
        struct add address;
        float cgpa ;
      };
struct std *s = new std;
s->address.street = 7;
```

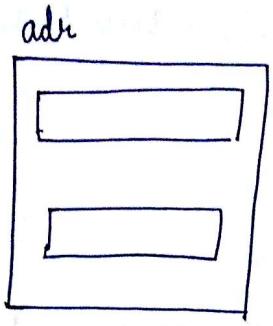


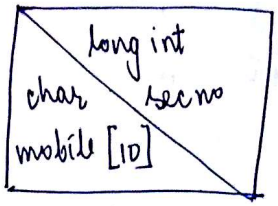



```
p = new(struct s3);
p->b = new(struct s2);
p->b->e = new(struct s1);
p->b->e->g = 5;
```

4 declarations;

```
struct s1
    {
      int g;
    struct s1 *h;
    };
struct s2
    {
      char f;
     struct s2 *d;
      struct s1 *e;
    };
struct s3
    { int a;
     struct s2 *b;
     struct s3 *c;
    }; struct s3 *p;
```





} either one (we use union)

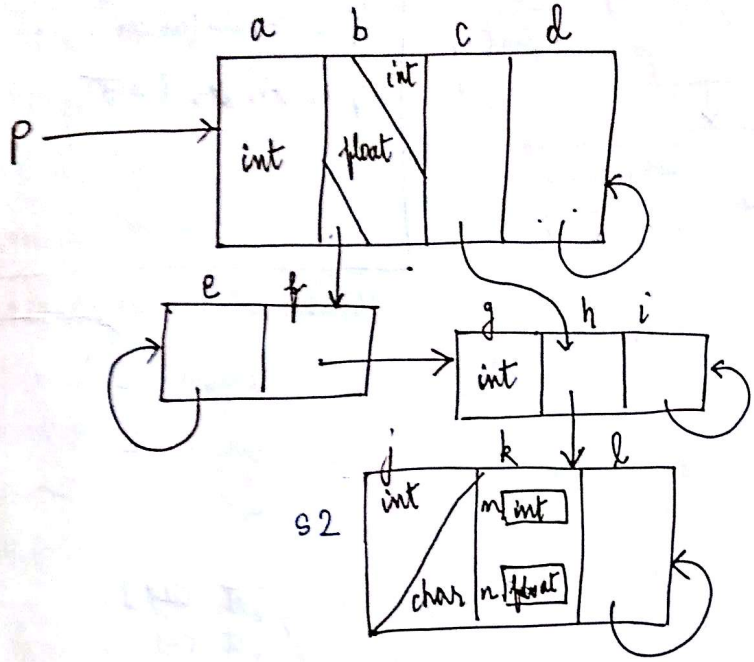
```
union adr {
        long int secno;
        char mobile [10];
        };

struct std {
    2.  int rno;
    15. char name [15];
    1   int flag;
    10  union adr address;
    4   float cgpa;
        };

struct std s[10];   320 bytes
```

```
int n;
for (i = 0; i < 10; i++)
{
  cout << "Enter roll no" << endl;
  cin >> s[i].rno;
  cout << "Enter name" << endl;
  cin >> s[i].name;
  cout << "Security or mob no?" << endl;
  cin >> n; s[i].flag = n;
  if (n == 1) cin >> s[i].address.secno;
  else cin >> s[i].address.mobile;
}
```

Every structure containing a union must contain a flag field for choosing union field

```
for (i = 0; i < 10; i++)
{
  cout << s[i].rno;
  cout << s[i].name;
  if (s[i].flag == 1) cout << s[i].address.secno;
      else cout << s[i].address.mobile;
```

In union take larger field for calculating memory field.



s1

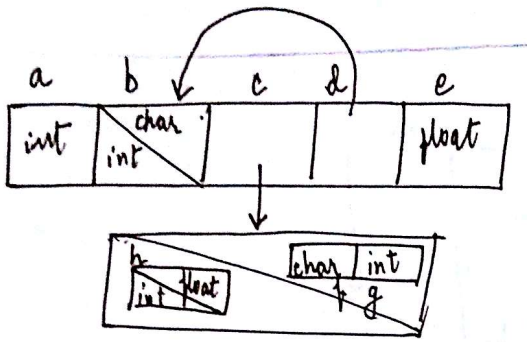
```
struct s1 {
        int m;
        float n;
        };
struct s2 {
        struct s1 k;
        union u1 j;
        struct s2 *l;
        int tag;
        };
struct s5
        {
        int a;
        union u2 b;
        struct s5 *d;
        struct s3 * c;
}; int tag;
```

```
union u1 {
        int o;
        char p;
        };
struct s3{
        int g;
        struct s3 * i;
        struct s2 *h;
        };
```

```
union u2 {
        int q;
        float r;
        struct s4 *s;
        };
struct s4 {
        struct s4 * e;
        struct s3 *f;
        };
```

```
struct s5 * p = new (struct s5);
p->c = new (struct s3);
p->c->h = new (struct s2);
p->c->h->k.m = 5;
```

③




```
struct s2 *obj = new(struct s2);
obj->c = new(union u2);
obj->c->h. i = 5
```

→

```
union u1{
    int i;
    float j;
};

struct s1{
    char f;
    int g;
};

union u2{
    union u1 h;
    struct s1 k;
};

union u3{
    char l;
    int m;
};

struct s2{
    int a;
    int tag1;
    int tag3;
    int tag;
    union u3 b;
    union u2 *c;
    union u3 *d;
    float e;
};
```

② Printing digits using recursion –

```
#include<iostream>
using namespace std;
void printdigit(int);
int main()
{
    printdigit(7942);
}
```

```
void printdigit(int n)
{
    if(n<10) cout<<n<<endl;
    else
    {
        printdigit(n/10);
        cout<<n%10;
    }
}
```

Assignment :-

1. Array of student structure using union → program.
2. Printing of 9742 using recursion
3. given diagram
4. Finding max, min, sum, avg using single recursion.

1.

```
#include<iostream>
using namespace std;
union contact {
        long mobile;
        int tele;
        };
struct std {
        char name[20];
        int roll;
        int t;
        contact c;
        float cgpa;
        };
int main()
{
  struct std s[10]; int ch;
  for(int i=0; i<10; i++)
{
  cout<<" Details of student "<<i+1;
  cout<<" Enter name "<<endl;
  cin>> s[i].name;
  cout<<" Enter roll number "<<endl;
  cin>> s[i].roll;
  cout<<"1. Mobile 2. Telephone ?"<<endl;
  cin>> s[i].t; cout<<" Enter "<<endl;
  if (s[i].t == 1)
  {
    cin>> s[i].c.mobile;
  }
  else cin>> s[i].c.tele;
  cout<<" Enter CGPA"<<endl;
  cin>> s[i].cgpa;
  }
    for(i=0; i<10; i++)
  {
  cout<< s[i].name<<endl;
  cout<< s[i].roll;
  if(s[i].t==1) cout<< s[i].c.mobile;
   else cout<< s[i].c.tele;
   cout<< s[i].cgpa;
  }
   return 0;
  }
```

④

```
#include<iostream>
using namespace std;
void recursion(int [], int, int, int, int);
int main()
{
  int a[10], n;
  cout << "Enter the number of integers";
  cin >> n;
  for(int i=0; i<n ; i++) cin >> a[i];
  recursion(a, a[0], a[0], a[0], n);
}
void recursion(int a[], int min, int max, int sum, int n)
{
  static int i = 1;
  if(i == n)
  {
    cout << "Minimum = " << min << " Max = " << max << " Sum = " << sum
         << "Average = " << (sum/n); (float)
    return ;
  }
  else
  {
    if (a[i] < min) min = a[i];
    if(a[i] > max) max = a[i];
    sum+ = a[i];
    i++;
    recursion(a, min, max, sum, n);
  }
}
```

Data Structures

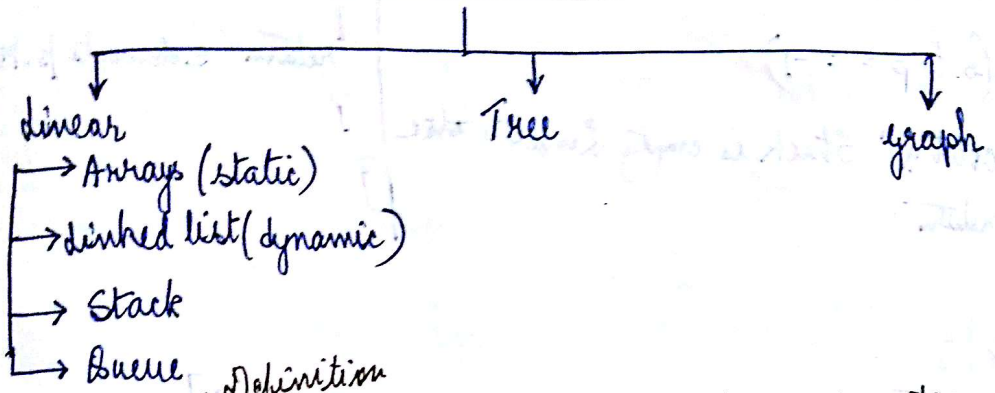
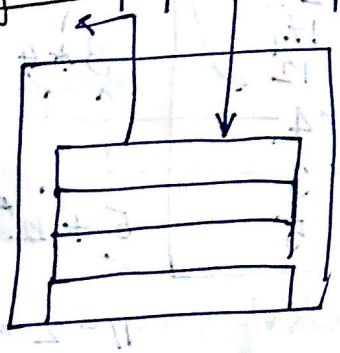


Diagram { Definition / Declaration / Operation / Application / Variation

Abstract Data Type - type which has data structure as well as operations associated with it.

Stacks (LIFO)

Diagram -



Definition

```
struct stack {
        int size;
        int elements[50];
        int top;
    };
```

Declaration

```
struct stack stk;
stk.top = -1;
stk.size = 50;
```

Operation -

```
void push(struct stack& s, int x)
{ if(s.size-1==s.top)
  { cout<<" stack is full"<<endl;
  }
else
{
  s.top++;
  s.elements[s.top]=x;
}
```

```
int pop(struct stack s)
{
  if(s.top==-1)
  {
    cout<<" Stack is empty "<<endl; else
    return
  }
  else
  {
  return s.elements[s.top--];
  }
}
```

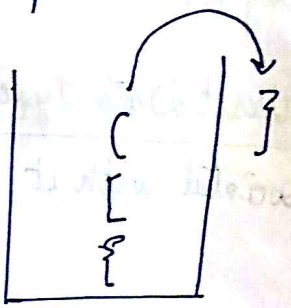
Application -

(1) Balancing symbols eg - {a+[b*c-(d*e+f)}]

Take an array,

opensym[] = {'{', '[', '('}

closesym[] = {'}', ']', ')'}



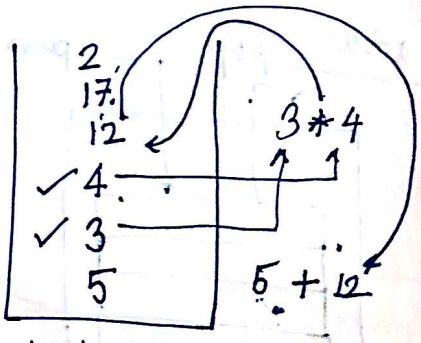
(2) Evaluating post-fix expression -

Infix → 5+3*4-2

Postfix → 534*+2-

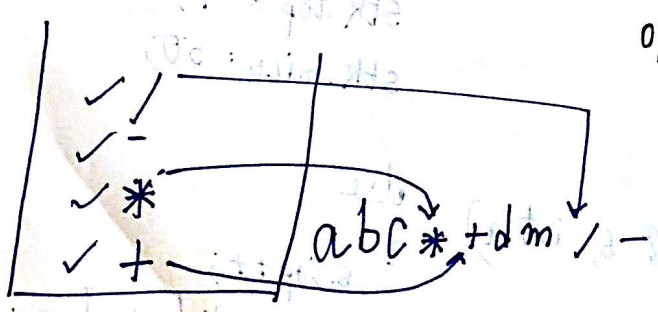
Infix → a/(b*c)+d/m

Postfix → abc*+dm/-



stack of operands

17 - 2 = 15



stack of operator

Infix → postfix (stack of stronger operator sits on weaker one)

An operator cannot sit on operator of same rank.

Every operation should enter stack ~~at least~~ once atleast.

Queues (FIFO) → First In First Out.

Definition

```
struct queue{
        int size;
        int elements[50];
        int front;
        int rear;
};
```

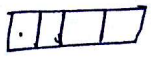
Queue is full is $(r+1)\%.size = f$

↑ since queue is round

Declaration: - struct queue Q;

→ Initialisation - Q.size = 50;
Q.front = -1;
Q.rear = -1;

```
void enqueue(queue& Q, int x);
{
    if((Q.rear+1)%.Q.size == Q.front) cout << "Queue is full" << endl;
    else
    {   if(Q.front == -1) Q.front = 0;
        Q.rear = (Q.rear+1)%.Q.size;
        Q.elements[Q.rear] = x;
    }
}
int dequeue(queue& Q)
{ if(Q.front == -1){cout << "Queue is empty" << endl;
  else
```



```
{
    if (Q.rear == Q.size-1)
    {
        Q.rear = 0;
        return Q.elements[Q.rear++];
    }
    else
        temp = Q.front;
        if (Q.front == Q.size-1) Q.front = 0;
        else Q.front++;
        return temp;
}
```

```
int dequeue(queue &Q)
{   int t;
    if(Q.front == -1) cout << "Q is empty" << endl;
    else
    {
        t = Q.elements[Q.front];
        if (Q.front == Q.rear)
        {   Q.front = -1;
            Q.rear = -1;
        }
        else
            Q.front = (Q.front + 1) % size;
    }
    return t;
}
```

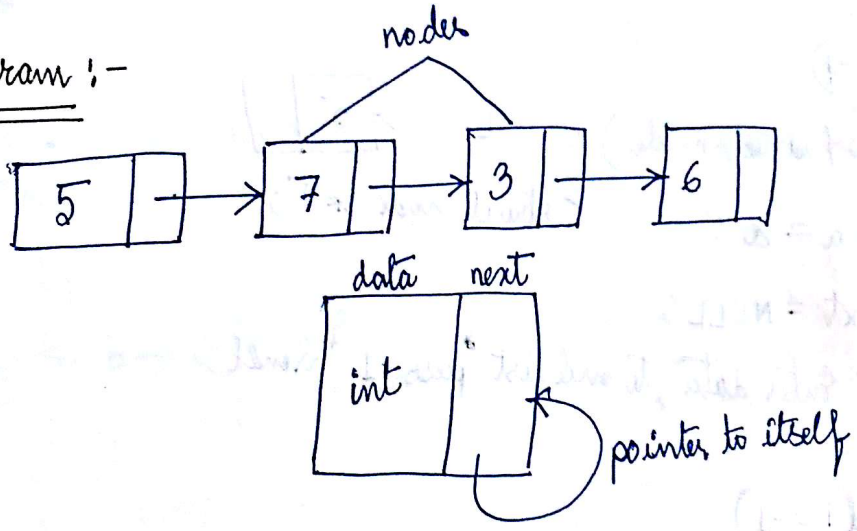
Note:

Assignments :-

1. Reverse a queue using stack
2. Implementation of queue using two stacks
3. Implementation of stack using queue.
4. enqueue, dequeue, eval.

Linked List

Diagram :-



Definition -

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

Declaration

```
struct node *L;
```

(Declaration is initialisation itself)

Note :- Use complex user-defined data types through pointers so that you can deallocate the memory later.

Operations -

1. addbegin
2. addend
3. addafter
3. addbefore
4. ~~remove~~ deletebegin
5. deleteend
6. createlist
7. printlist

Creating linked list -

```
cout<<" Enter data "<<endl;
cin>>d;
if(d!=-1)
L=new(struct node);
```


```
struct node *T;
L->data=d;
L->next=NULL;
cout<<" Enter data, to end list press -1 "<<endl;
cin>>d;
while(d!=-1)
{
    T=L;
    while(T->next!=NULL) T=T->next;
    T->next
    struct node *p=new(struct node);
    T->next->data=d;
    T->next->next=NULL;
}
```

☆ typedef int length ; → renaming int as length.

```
struct node          → typedef
{
    int data ;
    struct node * next ;
} *Lptr ;
```

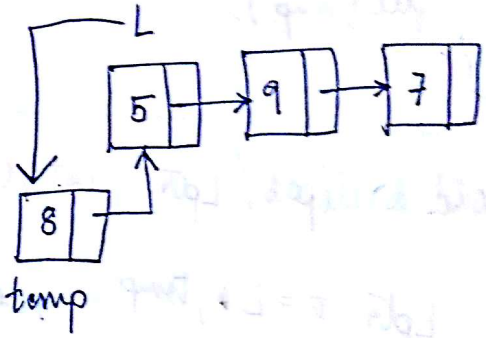
} 0 bytes allocation.

Lptr is a pointer to node structure type.

Lptr L ; (2 bytes)

void addbegin(Lptr L, int d) (Call by value)

```
{
    Lptr temp = new(struct node);
    temp -> data = d;
    temp -> next = L;
    L = temp;
}
```



void addend (Lptr L, int d)

```
{
    Lptr temp = new(struct node), v = L;
    temp -> data = d; temp -> next = NULL;
    while (v -> next != NULL) v = v -> next ;
    v -> next = temp ;
}
```

void deletend (Lptr L)

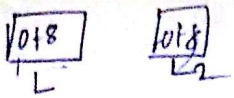
```
{
  Lptr temp = L, v = L;
  while(temp -> next != NULL) temp = temp -> next ;
  while (v -> next != temp) v = v -> next ;
  v -> next = NULL;
  free (temp);
}
```

void deletbegin (Lptr L)

```
{
  Lptr temp = L ~~->next~~ ;
  ~~free(temp)~~ L = L -> next ;
  free (temp);
}
```

void deletepos (Lptr L, int d)

```
{
  Lptr t = L, temp;
  while (t -> next -> next (data) != d ) t = t -> next ;
  temp = t -> next ;
  t -> next = t -> next -> next ;
  ~~free (temp);~~ delete temp ;
}
```




```
# void rprint(Lptr t)
{
  if (t != NULL)
  {
    cout << t->data << "  ";
    rprint(t->next);
  }
}
```

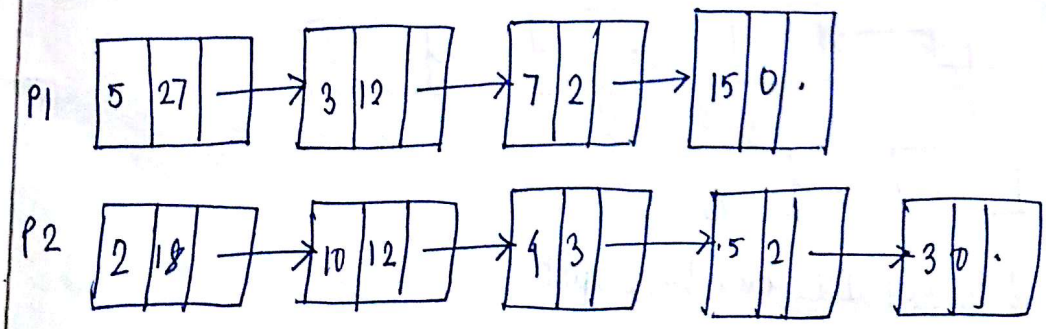
```
# void reverse print (Lptr t)
{
  if (t == NULL) return;
  else
  {
    reverseprint(t->next);
    cout << t->data << "  ";
  }
}
```

Polynomial addition -

$P1 : 5x^{27} + 3x^{12} + 7x^{2} + 15$

$P2 : 2x^{18} + 10x^{12} + 4x^{3} + 5x^{2} + 3$

A [i] → 3 information
- exponent
- value of array element = co-efficient.




```
void MergeL (Lptr l1, Lptr l2, Lptr l)
{
  Lptr t1, t2, l;
  while (t1 != NULL & t2 != NULL)
  { if (t->data > t2->data) { addend (t, t2->data); t2=t2->next;}
  } else {addend(t, t1->da); t1= t1->next;}
```

```
while (t1 != NULL)
{
    addend(t, t1->data);
    t1 = t1->next;
}
while (t2 != NULL)
{
    addend(t, t2->data);
    t2 = t2->next;
}
```

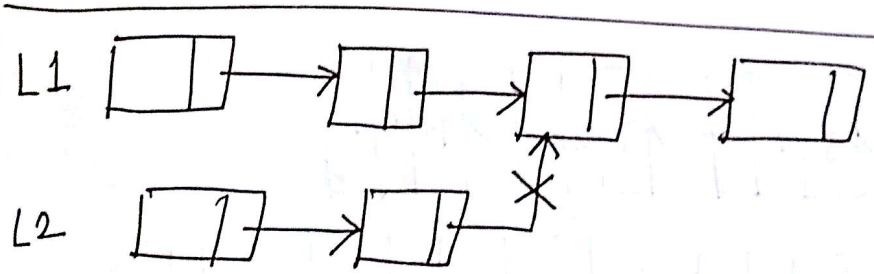
① Polynomial Addition
② Union Intersection of 2 lists
③ Insertion Sort using Lists
④ Insertion Sort Using Lists Recursively
⑤ Remove duplicates

(5) Stack
(9) Queue
(10) Josep

S E M I D (sorting methods).

- S → Selection
- E → Exchange sort or Bubble sort
- M → Merge sort
- I → Insertion Sort
- D → Distribution

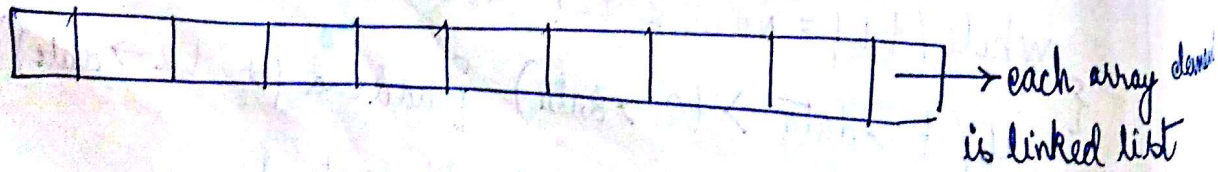
(6)



Create 2 linked lists and then split.

(7) Radix Sort :- (Bucket Sort)

98 517 33 26 801 745 4 19



→ each array element is linked list

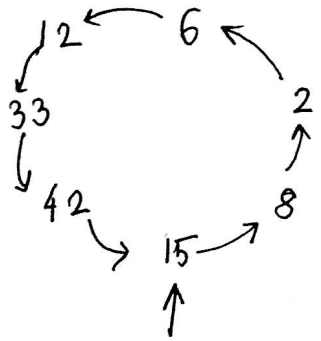
#

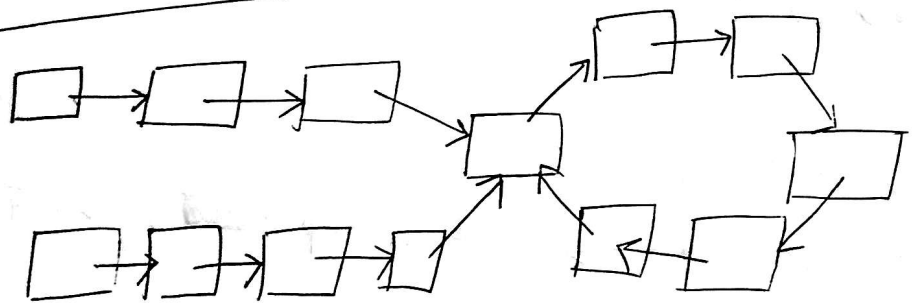
int x = (rand()%100);

(8) Stack as linked list

(9) Queue as linked list

(10) Joseph's Problem





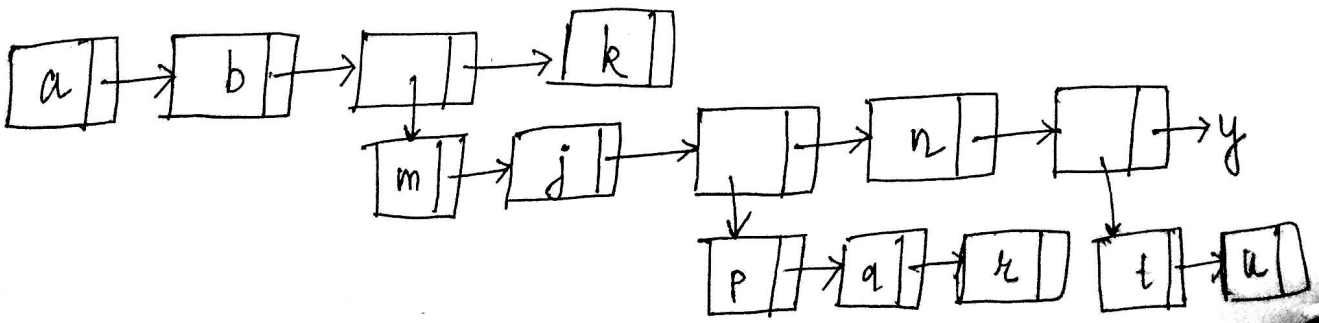
$S = (a, b, (m, j, (p, q, r), n, (t, u), y), k)$ 4 elements 4 sets

Let $A = (p, q, r)$

$B = (t, u)$

$C = (m, j, A, n, B, y)$

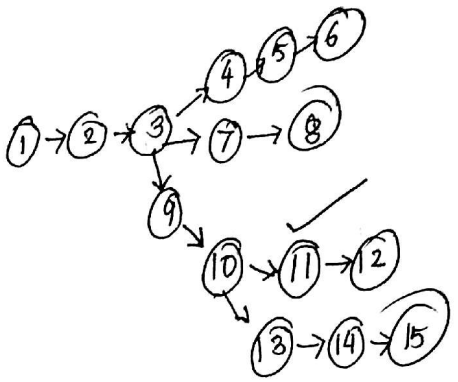
$\therefore S = (a, b, C, k)$



```
struct list {
        int tag ;
        union u {
                char data ;
                struct list * head ;
              } obj ;
        struct list * next ;
      } L ;
```

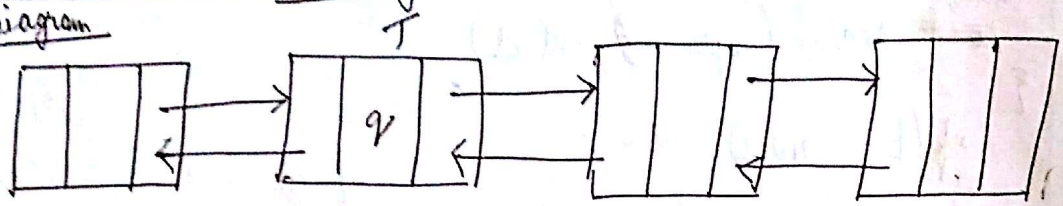
```
void print(struct list * L)
{
   if (L == NULL) return ;
   else
   {
      if (L->tag == 1) cout << L->obj.data ;
      else
      {
         cout << " ( " ;
         print(L->obj.head) ;
         cout << " ) " ;
      }
      print (L->next) ;
   }
}
```

```
bool
~~void~~ search( Lptr L, int d)
{
  if (L == NULL) return ;
  else if ( L → data == d)
  {
    cout << " Item found " ;
    return true ;
  }
  search (L → next 1) ;
  search (L → next 2) ;
  search (L → next 3) ;
  cout << " Item not found " << endl ;
  return false ;
}
```



Doubly linked list

Diagram



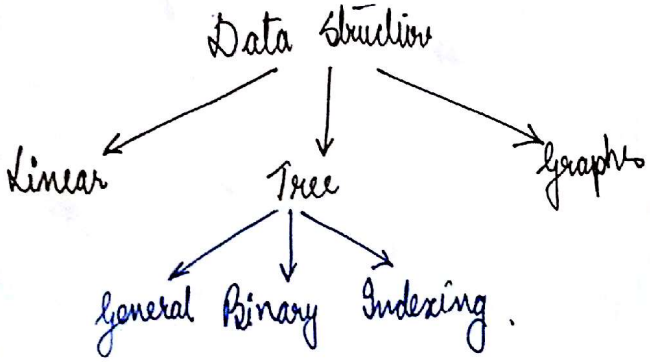
Definition-

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

Declaration:- `struct dNode* DPTR;`

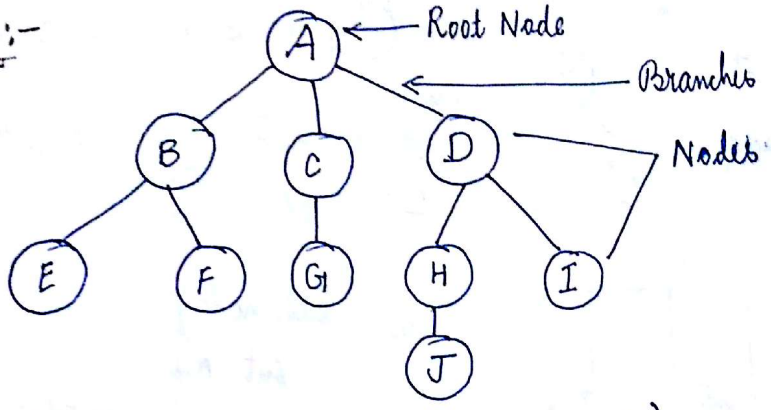
Deleting node T:- $T \rightarrow left \rightarrow right = T \rightarrow right$

$T \rightarrow right \rightarrow left = T \rightarrow left$.



Binary Search Tree

Tree:-



B C D → siblings (child of same parent node).

The number of branches from root to a node is its pathlength.

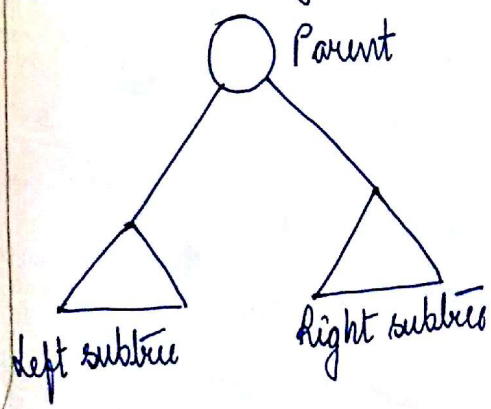
A node without children are called external nodes / terminal nodes / leaf nodes.

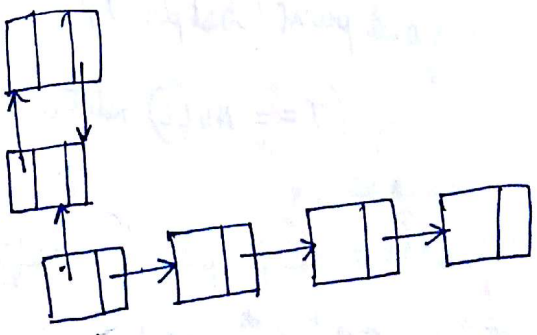
A node having atleast one child is internal nodes / non-terminal / non-leaf nodes.

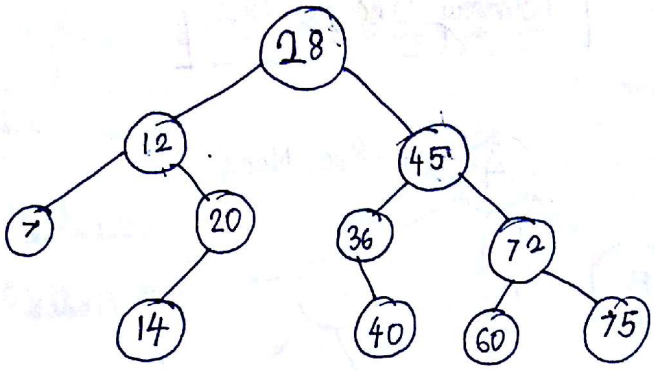
A binary tree is a tree so that each node has atmost 2 children.

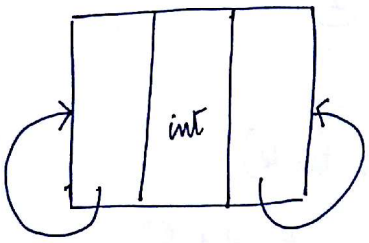
A binary search tree is a tree with such property that at every node the key value must be greater than all the values in its left subtree and smaller than right subtree.

#










```
struct bst_node {
    int key;
    bst_node *leftchild;
    bst_node *rightchild;
};
```

```
void insert(bstptr T, int k)
{ if (T == NULL) { T = create node(k); return; }
else { if (k > T→key) insert(T→rightchild, k);
    else if (k < T→key) insert(T→leftchild, k);
  }
}
```

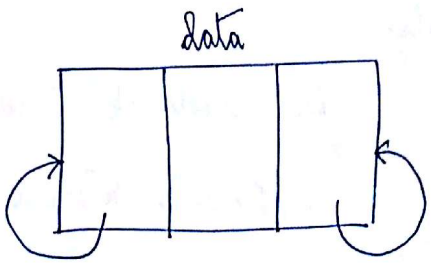
```
void print(bstptr T)
{ if (T == NULL) return;
  else
  { print(T→left);
    cout << T→key;
    print(T→rightchild);
  }
}
```

Height of tree :- height of a tree is pathlength from deepest node to root.

Depth of a node - branches from root to that node

Binary tree - Tree in which a node has atmost 2 children.

```
struct bst_node {
    int data ;
    bstnode *lchild ;
    bstnode *rchild ;
};
typedef bstnode* BSTPTR
```



Binary search tree is a BT such that at every node the data value is greater than data values of all nodes in left subtree and ~~gre~~ smaller than those of all nodes in right subtree.

Q- Find node at which search property is not satisfied.

```
boot
void find (BSTPTR root, int k)
{
    if (root == NULL) return false ;
    else
    {
        if (root -> data == k) return true ;
        else if (root -> data > k)
            return find (root -> lchild, k);
        else if (root -> data < k)
            return find (root -> rchild, k);
        else return false ;
    }
}
```

~~else if (find (root -> l~~
~~{ ssss find (root -> lchild, k);~~

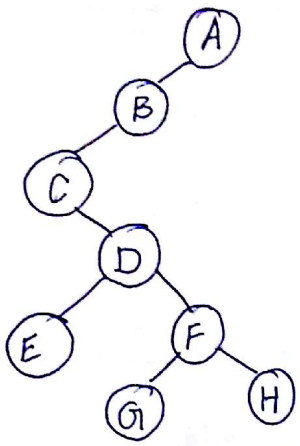
```
void insert (BSTPTR T, int k)
{ if (T==NULL)
  {
    T = new (bst_node);
    T→data = k; T→leftchild = null; T→rightchild = null;
  }
else
  {
     if (T→data > k) insert (T→leftchild, k);
     if (T→data < k) insert (T→rightchild, k);
  }
```

Inorder Printing	Preorder printing	Post order printing	Level Order
LDR	DLR	LRD	ABCDEFG

DBEAFGC (inorder)
ABDECFG (preord)
DEBGFCA (postorder)

```
void preorder(BSTPTR t)
{ if (t==NULL) return;
else
{ cout << t→data;
  preorder (t→leftchild);
  postorder (t→rightchild);
}
}
```

```
void postorder (BSTPTR t)
{
   cout << postorder (t→rightchild);
   postorder (t→leftchild);
   cout << t→data;
}
```



LRD: EGHFDCBA

LDR: CEDGFHBA

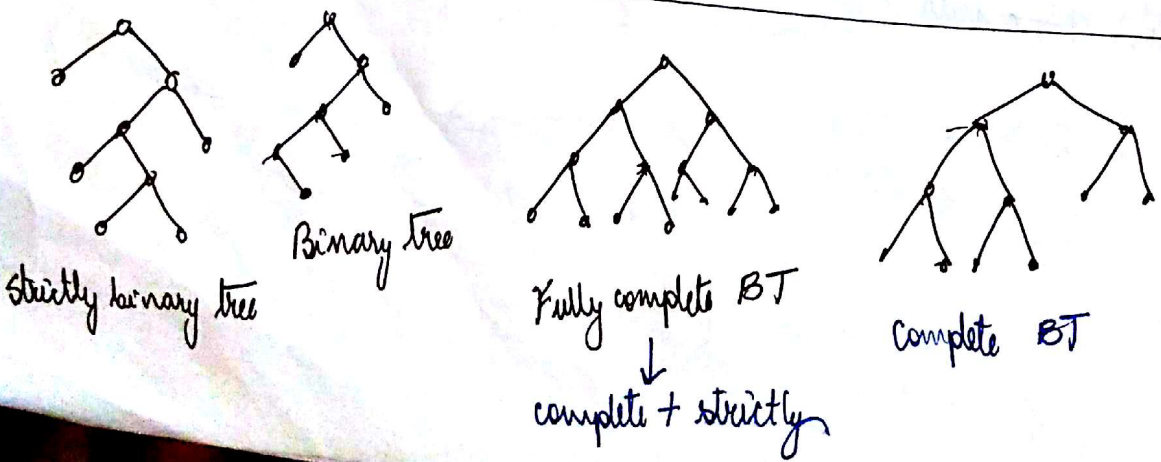
```
int findmin(BSTPTR, T      )
{
   if (T == NULL) return ;
else
 {  ~~if (rechild~~ while (T->lchild) T=T->lchild
    return T->data ;

 }
int findmax (BSTPTR T)
{
    if (T == NULL) returns ;
    while (T->rightchild) T=T->rightchild;
    return T-> data ;
}
```

Insertion (iterative)-

```
void Insert (BSTPTR t, int key)
{
  if (t == NULL) return ;
  else
  {
    while (key < t -> data) t = t -> lchild ;
    while (k > t -> data) t = t -> rchild ;
```

```
void Insert (BSTPTR T, int val
{
  while (T)
  {
    if (T -> data > val) T = T -> lchild ;
    else if (T -> data < val) T = T -> rchild ;
  }
  T = new bstnode ;
  T -> data = val ;
  T -> lchild = T -> rchild = NULL ;
}
```

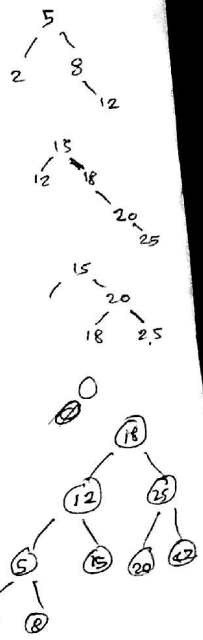
Tree Traversal (LDR, DLR or LRD or level order (breadth first))



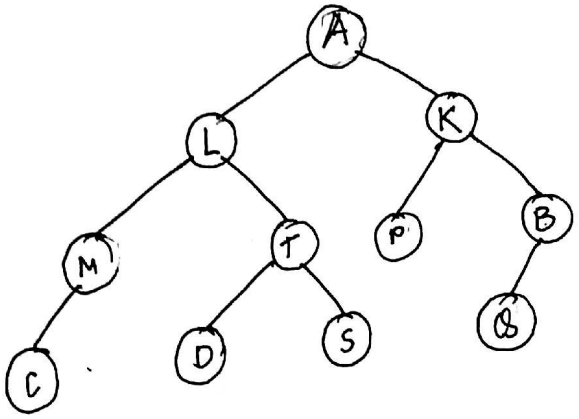
Q- Construct BST if i/p is such that binary search tree is complete.

2	5	8	12	15	18	20	25	42
0	1	2	3	4	5	6	7	8

Q - Construct BST in iterative manner.



Binary Tree



A
L K
M T P B

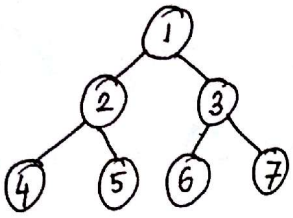
```
while (Q is not empty)
{
    T = dequeue(Q)
    cout << T->data;
    if (T->lchild != NULL) enqueue (Q, T->lchild)
    if (T->rchild != NULL) enqueue (Q, T->rchild)
}
```

Q- Print a binary tree line by line (level order printing)

Q- Construct binary tree (user input → A L M C ...)
If a node does not have a left or right child he will put '.' when asked.

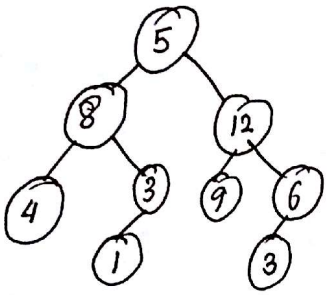
Q- Find maximum element in binary tree

Q-



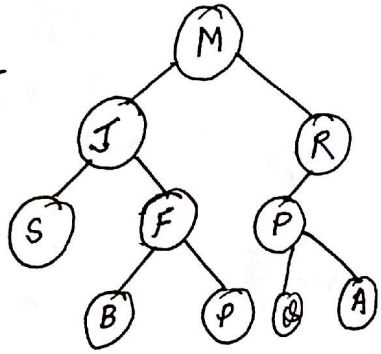
Print 4 5 6 7 2 3 1.

Q-



Print all possible paths.

Q-



Print a binary tree as it is.

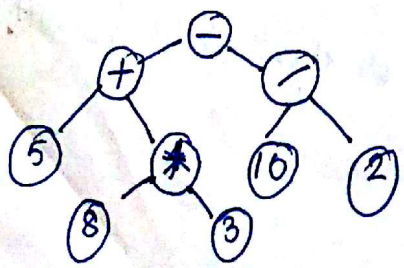
Y co-ordinate ⟶ level order

X co-ordinate ⟶ inorder

Expression Trees

⟶ it is a binary tree where internal or non-leaf nodes are operators and external/leaf nodes are operands.

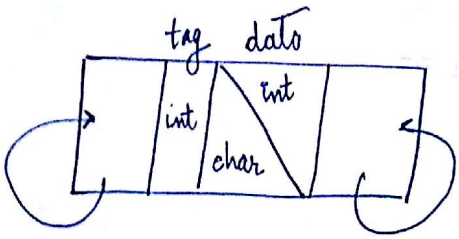
Infix :- 5 + 8 * 3 - 10/2 .



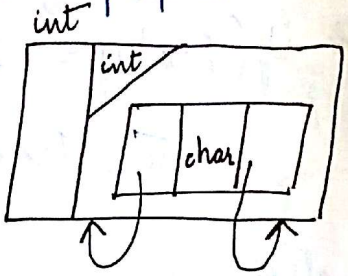
```
int eval(BTPTR T)
{
    if(T->data == '+' || T->data ==
    char x,y;
    if(T->tag == 1)
    {
        return T->leftchild, T->data;
    }
    else
    {
        eval(T->leftchild); int x =
        eval(T->rightchild);
        return function(T);
        eval(T->leftchild)
        int y = eval(T->rightchild);
                                  right
        switch(T->data):
            case '+' : return x+y;
            :
    }
}
```

```
int eval(BTPTR T)
{
    if(T->leftchild == 1) return T->data;
                tag
    else
    {
        int a = eval(T->leftchild);
        int b = eval(T->rightchild);
        switch(T->data):
            case '+': return a+b;
```

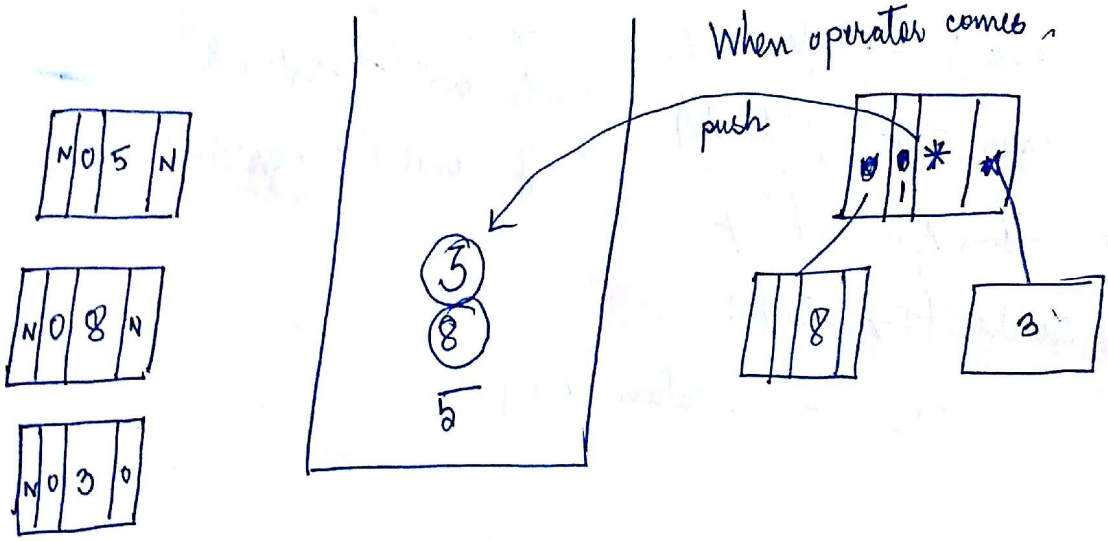
Postfix :- 5 8 3 * ^ 1 0 2 / +



Node structure of expression tree.



Postfix evaluation of expression tree :-



---

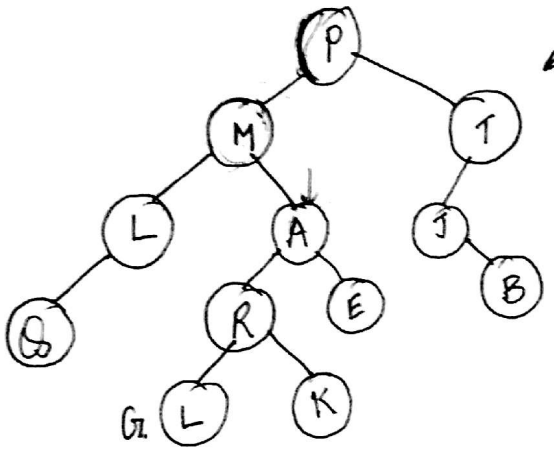
1. Multidigit evaluation
2. ET construction and evaluation
3. ET construction using 2nd structure
4. BST deletion
5. LDR, DLR
6. LDR, LRD
7. Top View of Binary Tree

in BST

Remember:- To delete a node, substitute leftmost child node of right subtree or rightmost child of left subtree.

LDR: Ø L M L R K A E P J B T

DLR: P M L Ø A R L K E T J B




```
root->left = dNode (R->left);
temp = Node * root;
root = root->right;
delete temp;
return root;
```

LRD: Ø L G K R E A M B J T P ← root of binary tree

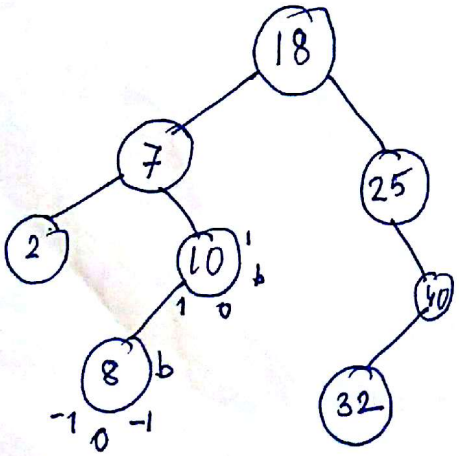
LDR: Ø L M L R K A E P J B T

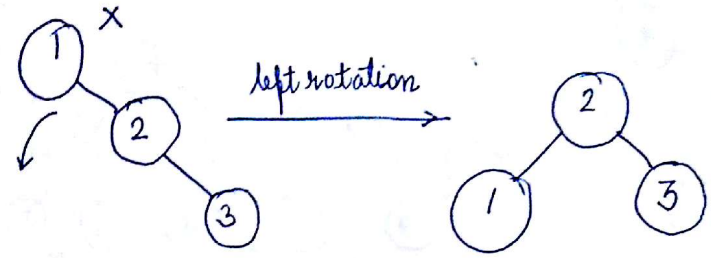
Height of a Binary Tree :-

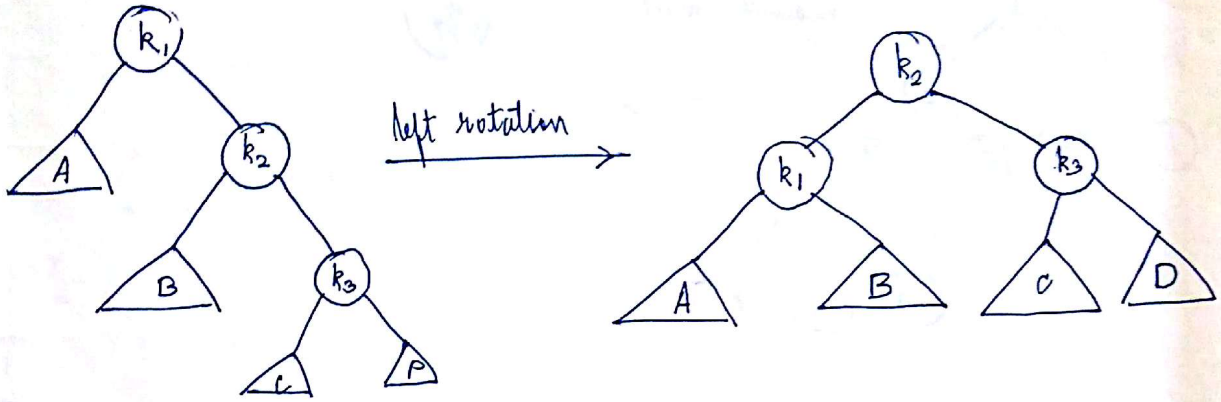
```
int calcheight (Bptr &R)
{
  if (R == NULL) return -1;
  else
  {
    if (calcheight (R->left) > calcheight (R->right)) return
                                    1 + calcheight (R->left
    else return 1 + calcheight (R->right);
  }
}
```

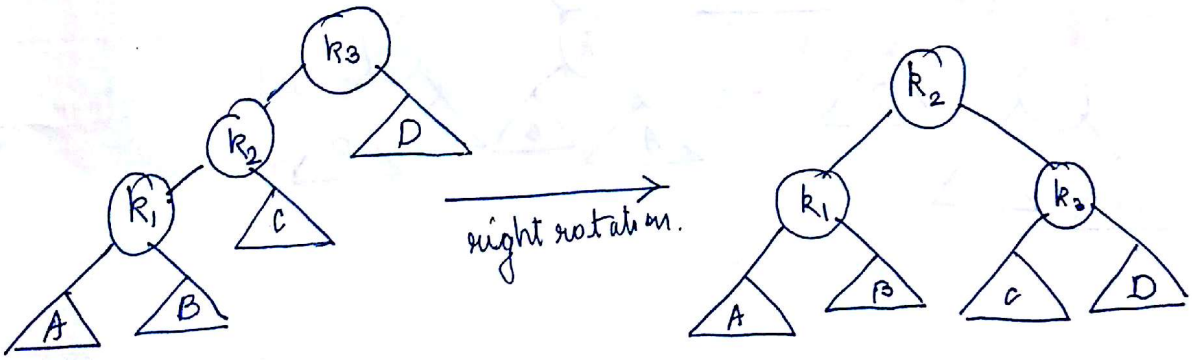
AVL Tree

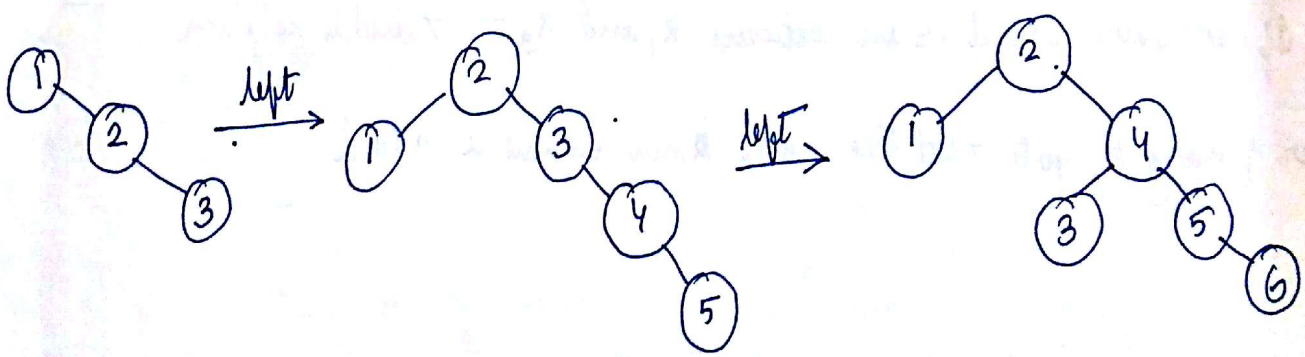
AVL Tree is a Binary Search Tree with height balanced condition. At every node height of left subtree and height of right subtree must differ at most by 1.

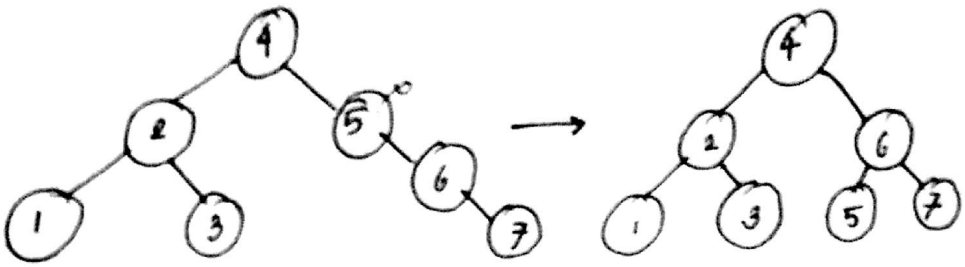


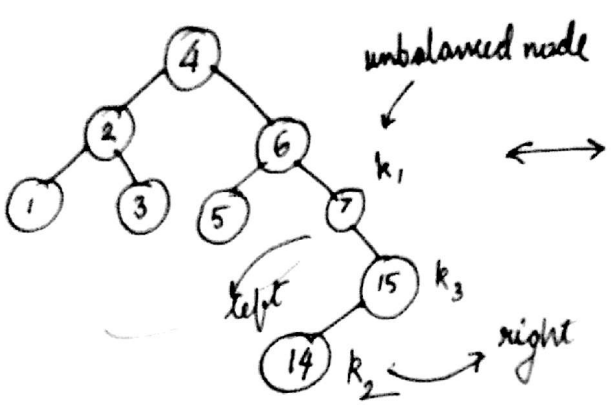


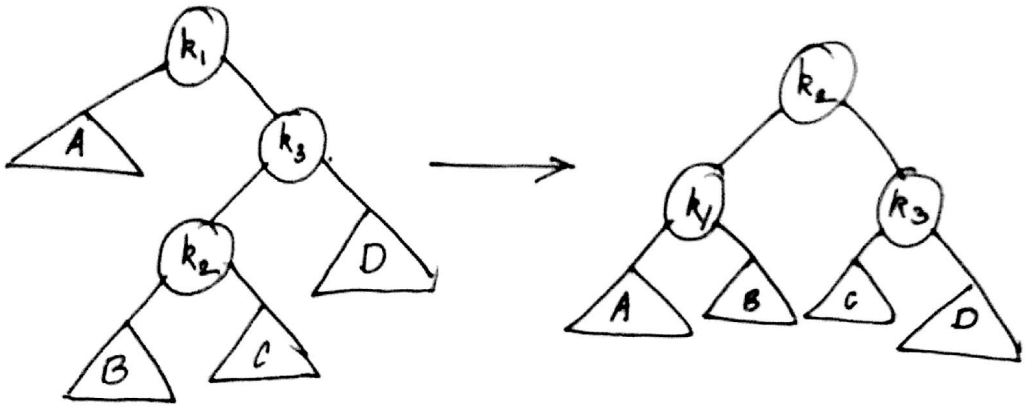






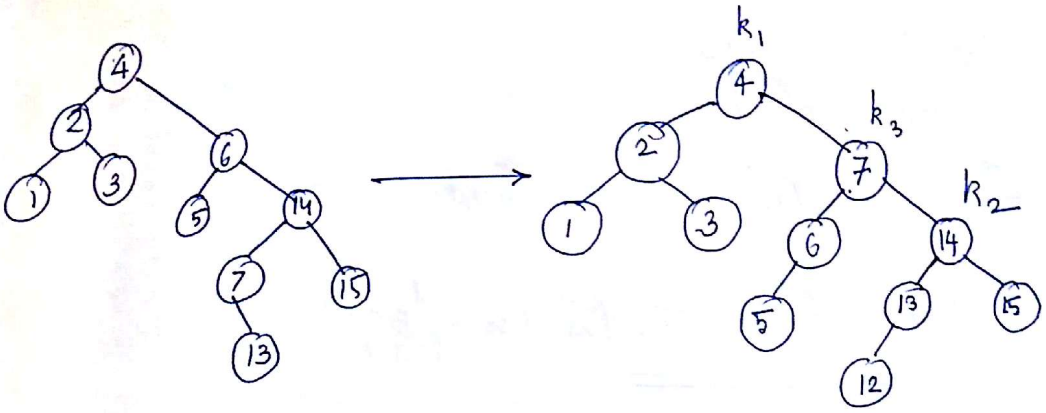


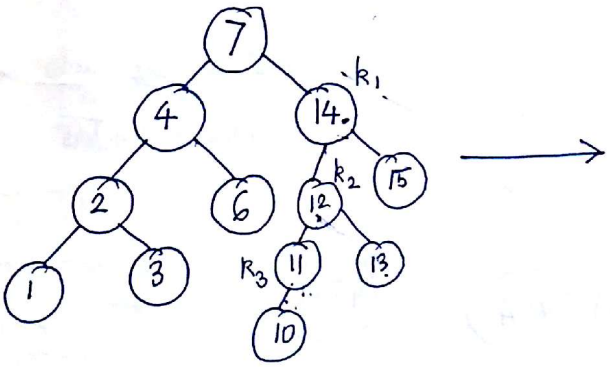




If new value entered is in between k_1 and k_3 ⟶ double rotation

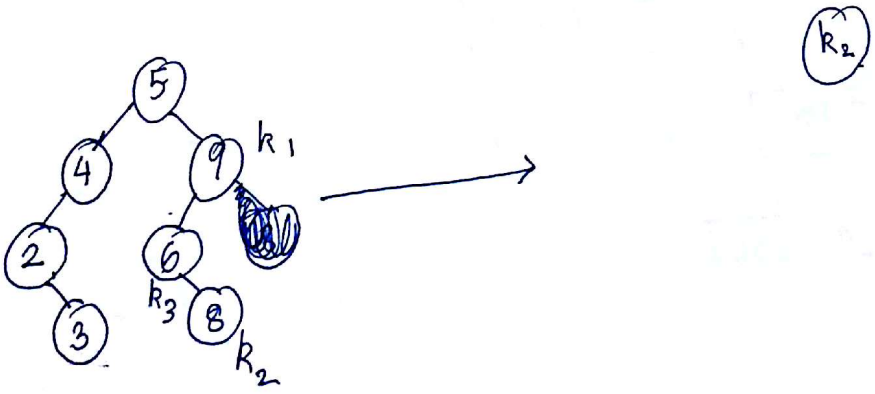
For finding k_2 go to that tree where the new element is added.

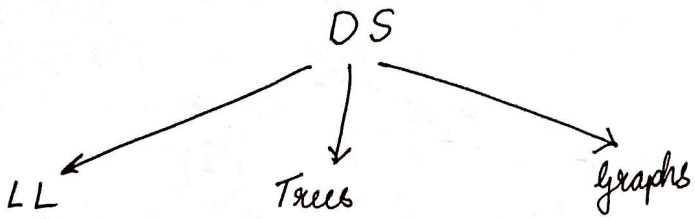




Q Create AVL Tree and tell the no. of single and double rotation -

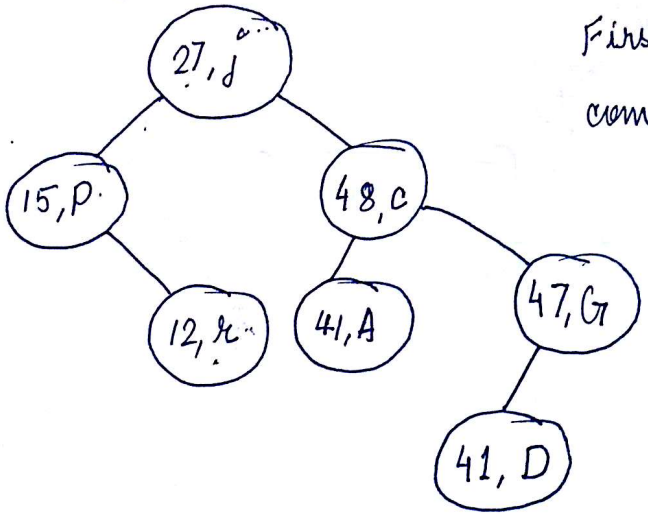
5 9 4 2 6 3 8



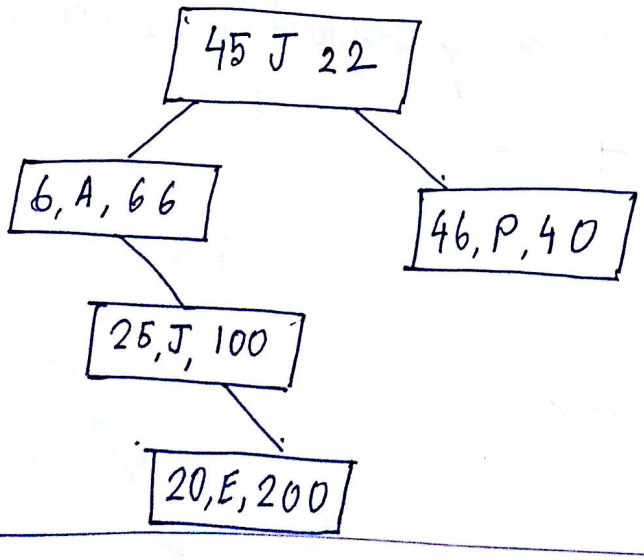


K-d tree (variation of BST)

Each node has two keys - a number and a character



First compare number then compare characters



```
struct kdNode {
    kdNode * leftchild;
    kdNode * rightchild;
    int data[d];
};   → keys.
```

Q. int Kdsearch (KdPTR, T, int key[], int d)

```
{   static int i = 0;
    if (T == NULL) return -1;
    else
    {
        if (key[i] > T->data[i]) T = T->left;
        else T = T->right; i++;
        else for (i = 0; i < d; i++)
        {
            if (key[i] > T->data[i]) kdsearch (T->right, key, d);
            else if (key[i] < T->data[i]) kdsearch (T->left, key, d);
            else
            {
```

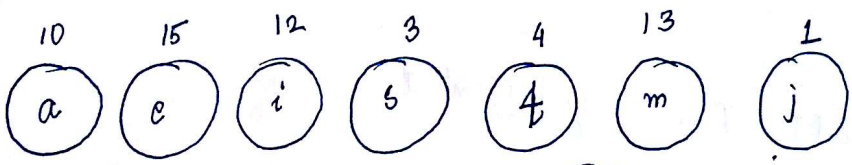
int kdsearch (kdPTR T, int key[], int d)

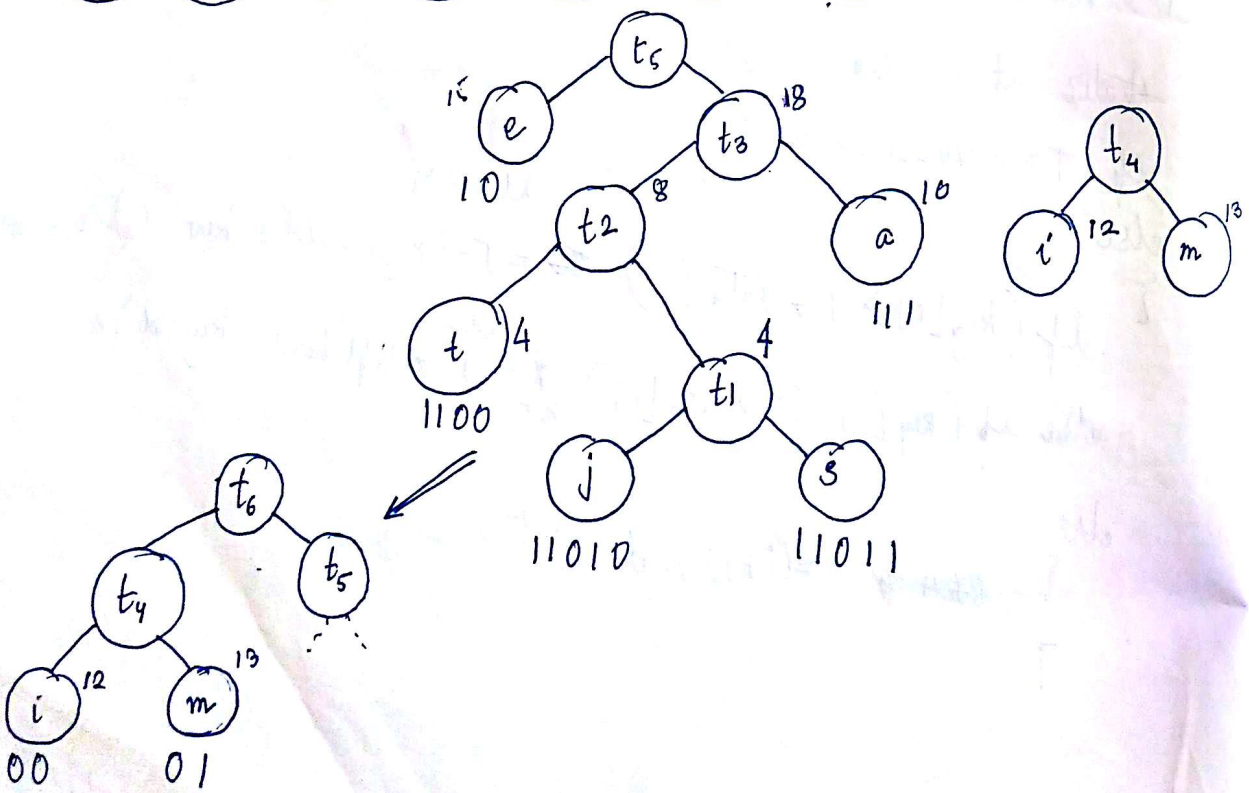
```
{   static int i = 0, x = 0;
    if (T == NULL) return -1;
    else
    {
        if (key[i] > T->data[i]) x = kdsearch (T->rightchild, key, d); i = (i+1)%d;
        else if (key[i] < T->data[i]) x = kdsearch (T->leftchild, key, d);
        else
        {   i = (i+1) % d; return 1;
        }
```

Huffman Code Tree

	frequency		Codes got from HCT	
a	10	000	111	30
e	15	001	10	30
i	12	010	00	24
s	3	011	1011	15
t	4	100	1100	16
m	13	101	01	26
j	1	110	11010	5
	58 bytes → 464 bits		174 bits	146 ← bits

(each character uses 3 bits)





Now we

Suppose

Sende

Ass

1
2
3
4
5
6
7

To d

Pre-order

Now we follow, for going left → 0 , for going right → 1.

Suppose data is sent → 1000011101110 10 (eims)

Sender and receiver share same HCT.

Assignment :-

1. AVL Tree
2. K_d Tree
3. Huffman Code Tree → do everything taught
4. Digital Search Tree
5. DLR
6. LDR
7. LRD

} Iteratively.

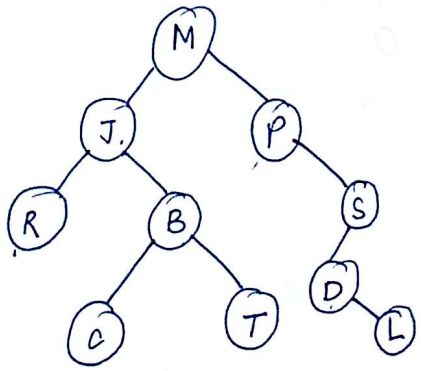
To eliminate recursion use stack compulsorily.

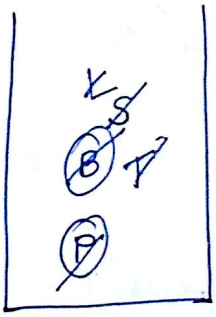
Print jump push

D L R

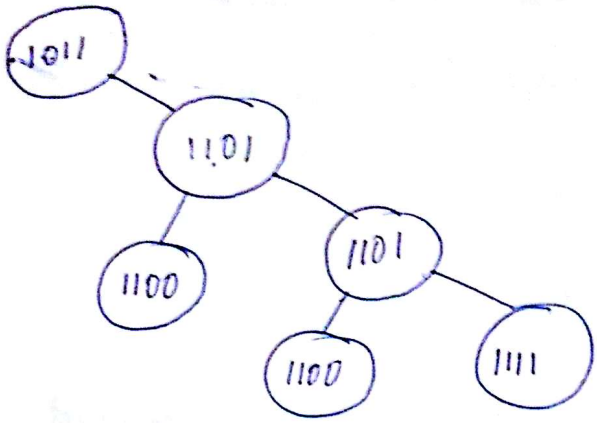
MJRBCTPSDL

Pre-order M J R B C T P S D L

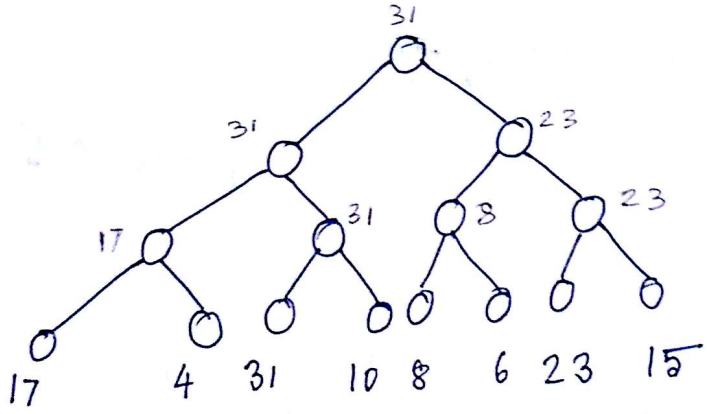




Digital Search Tree



Tournament Trees

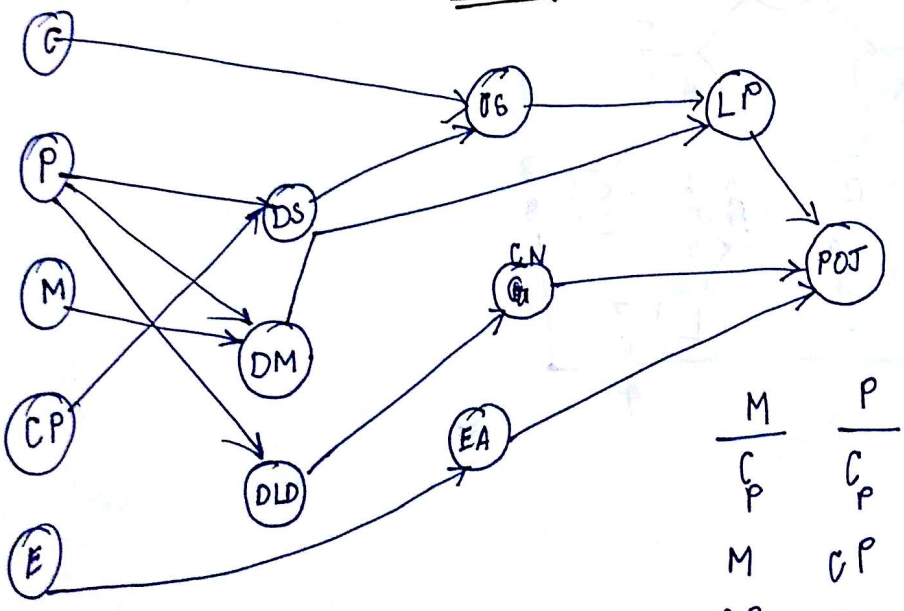


31 23 17 15 10

```
void visitleafnodes(T)
{ if(T== NULL) return;
  if(T->left == NULL && T->right == NULL)
  { insertatend(S,T); }  else { visitleafnodes(T->left);
                               visitleafnodes(T->right); }
```

```
void insertatend (stackds, Bptr&T)
{
  if (s.top==-1) push(s,T);
  else
  {
    temp=po if (T->data<s.peek()->data)
    {
      temp=pop(s);
      insertatend(s,T);
      push(s,T);
    }
  }
}
void printstack(stack&s)
{
  while(s.top>-1) cout<<pop(s)<<endl;
}
```

Topological Sort

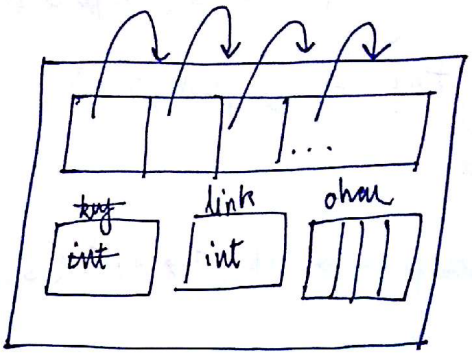


DLD	DS	DM	OS
DS	DS	OS	
DM	DM	DM	
DLD			

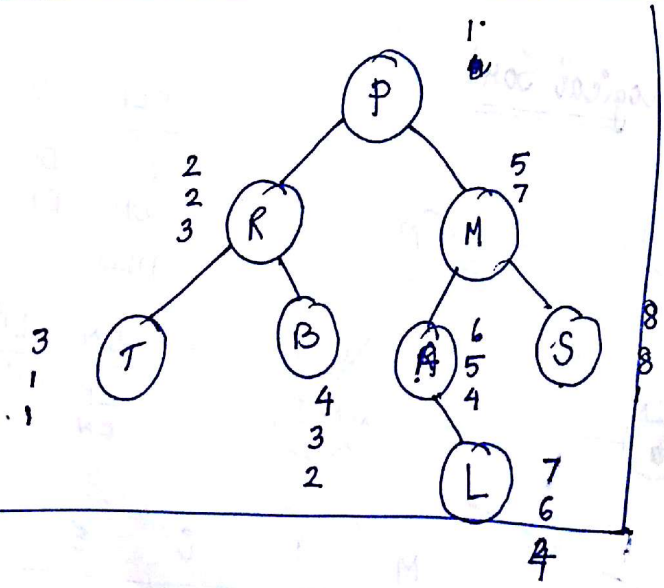
CN	LP
LP	
CN	

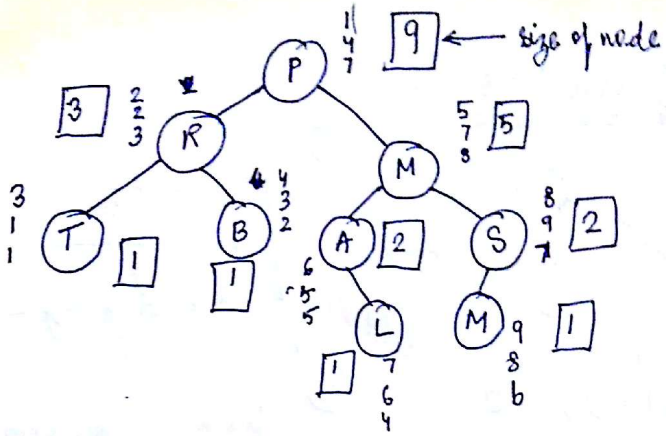
M	P	C	E	EA	CP
C	C	C	CP	DM	DM
P	C	CP	E	DLD	DL
M	CP	E	DM	CP	CP
CP	E	DM		EA	
E		DLD			

```
struct courses{
    char subject[10];
    ~~int key~~;  int links;
    courses** b;
};
typedef courses * Cptr;
Cptr T = NULL;
T = new courses;
cout<<"Enter name of course";
cin>>name;
strcpy(T->subject, name);
cout<<"Enter no. of links"; cin>>T->links;
T->b = new Cptr[links];
```



2/09/16



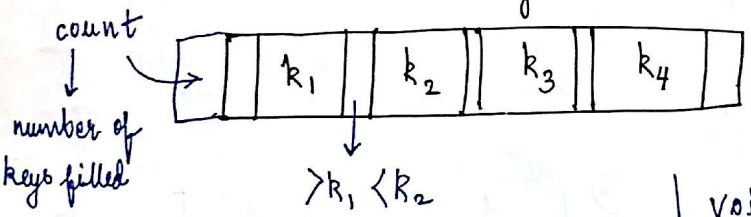


DLR
LDR
LRD

2/09/16

M-Way Search Tree (multi-way)

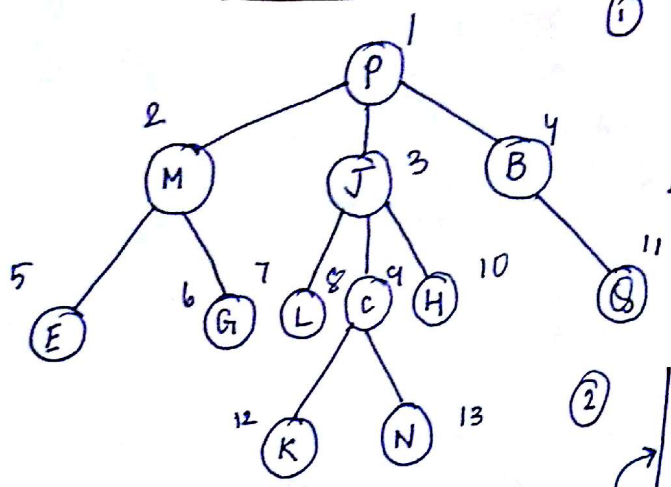
4 way search tree

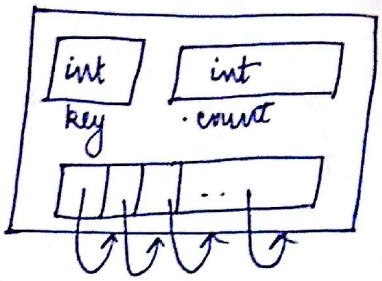


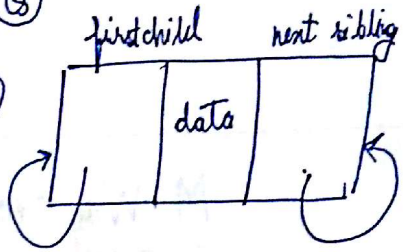

```
struct mstNode {
    int count;
    int key[m];
    mstNode * ptr[m+1];
};
```

```
void printtree (mstptr T)
{ int i;
  if (T == NULL) return ;
  else
  {
    for (i = 0; i < T->count; i++)
    {
      printtree (T->ptr [i]);
      cout << T-> key [i] ;
    }
    printtree (T->ptr [T->count]);
  }
}
```

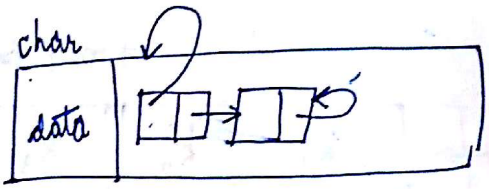
General Trees

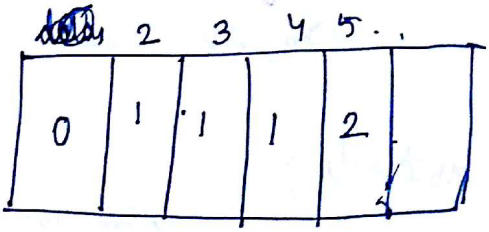


① 

② 


```
struct gtNode {
        int count ;
        char data ;
        gtNode** ptr ;
};
```

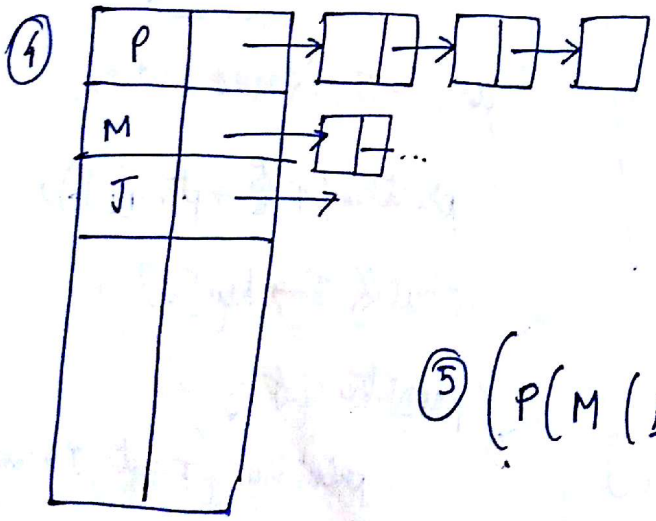
③ 

④ 

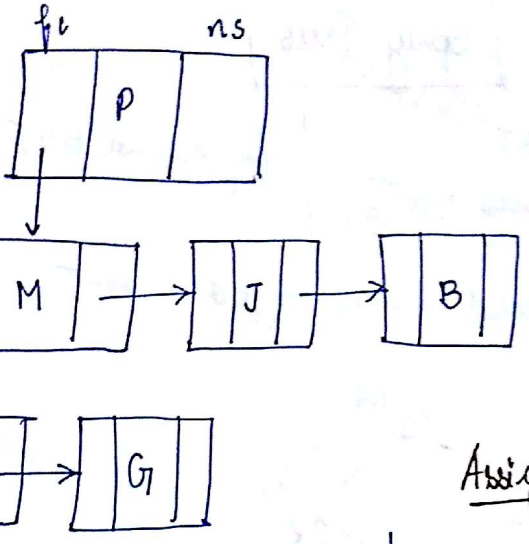
information of parent

↓

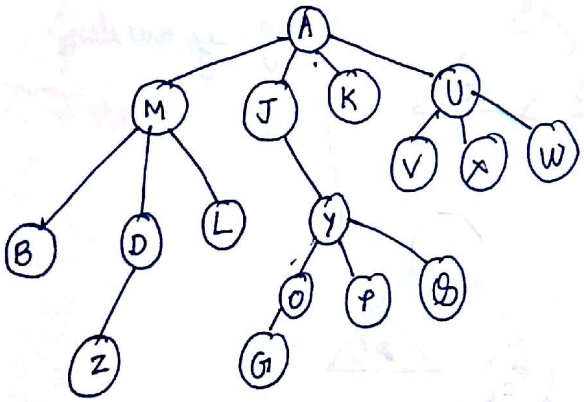
From child go to parent then find sibling.

④ 

⑤ (P(M(E,G)J(L C(K N)H) B)Q)




```
void create (struct gtNode *T, char c[])
{
    if (T == NULL)
    {
        T->data = c;
        T->fc = NULL; T->ns = NULL;
    }
}
```



Assignments

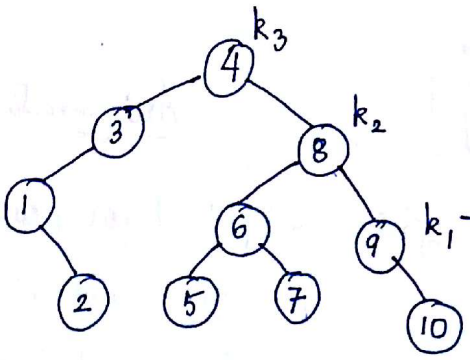
1. Topological Sort
2. Tournament Sort
3. General Tree – Recursive & Iterative
   - Print BFT, DFT
4. General Tree as it is
5. m-way search tree, creation, insertion, finding printing.
6. m-way search delete
7. Split a tree into binary trees

Splay Trees

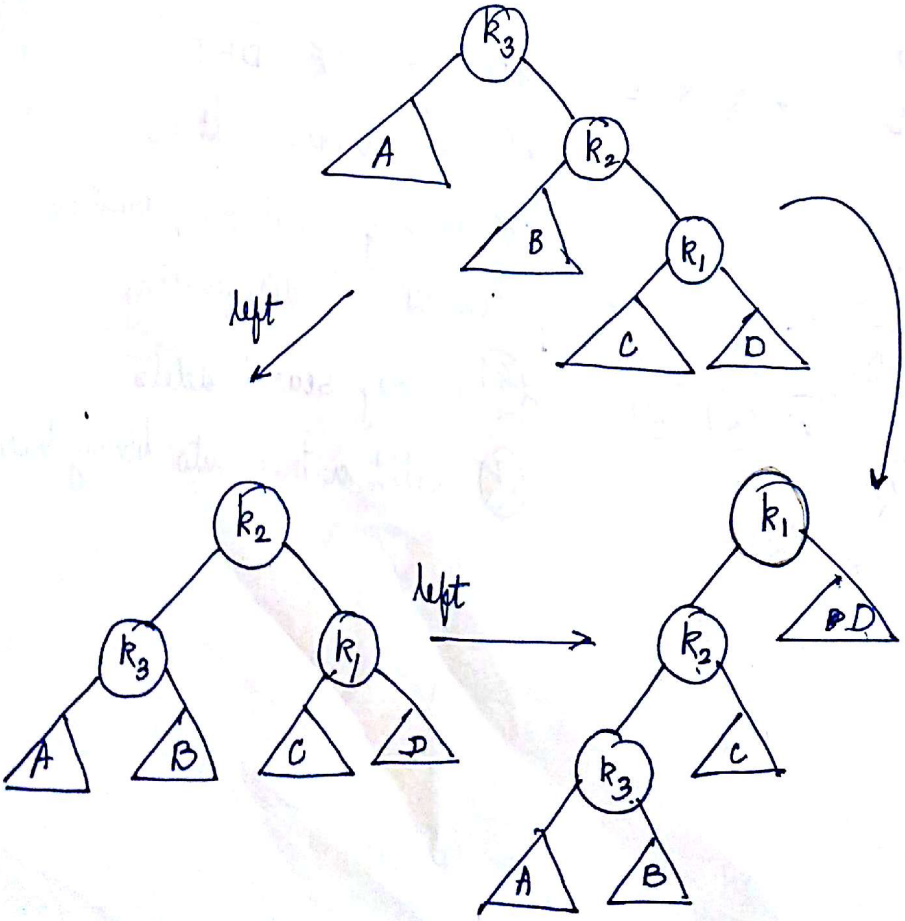
→ it is a BST
→ self-adjusting property

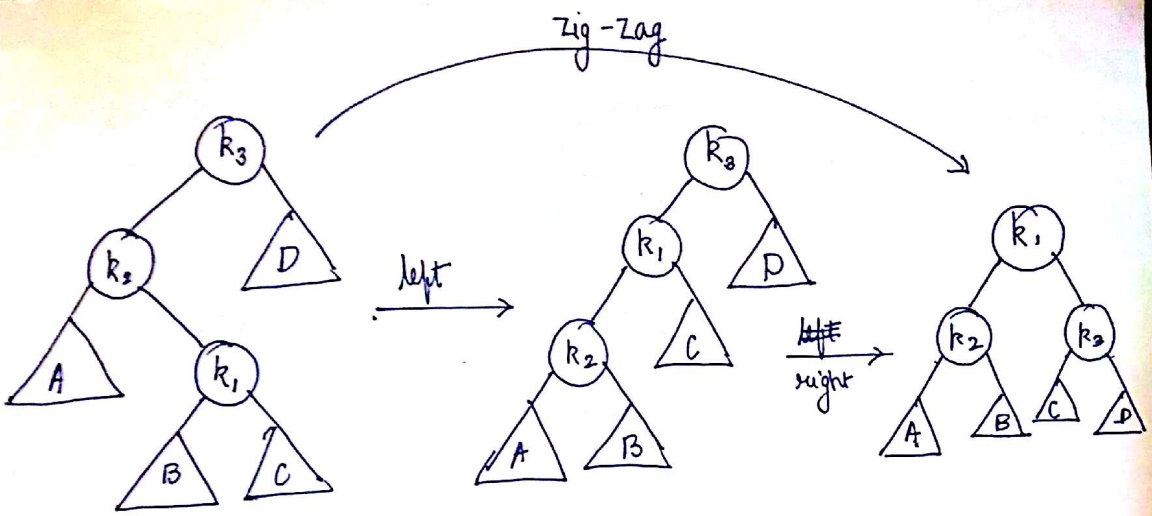
Splay trees are BST with splaying property

Most frequently searched node is shifted to root.

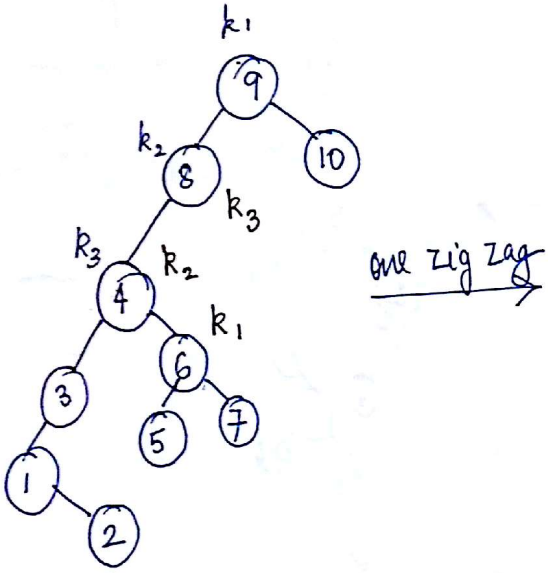


→ most frequently searched.

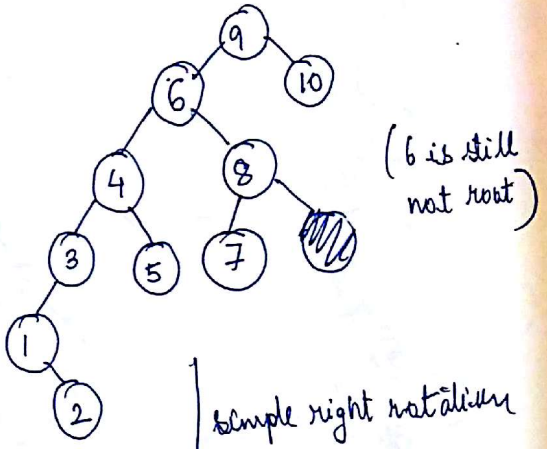




Don't bother height balancing in zig-zig and zig-zag rotations.

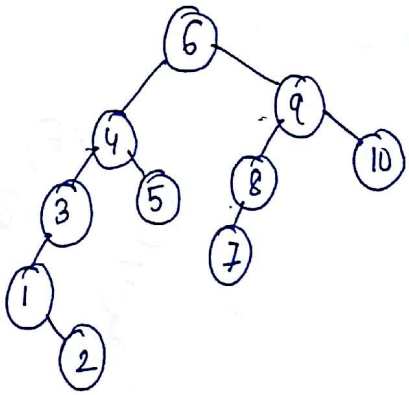


one zig zag

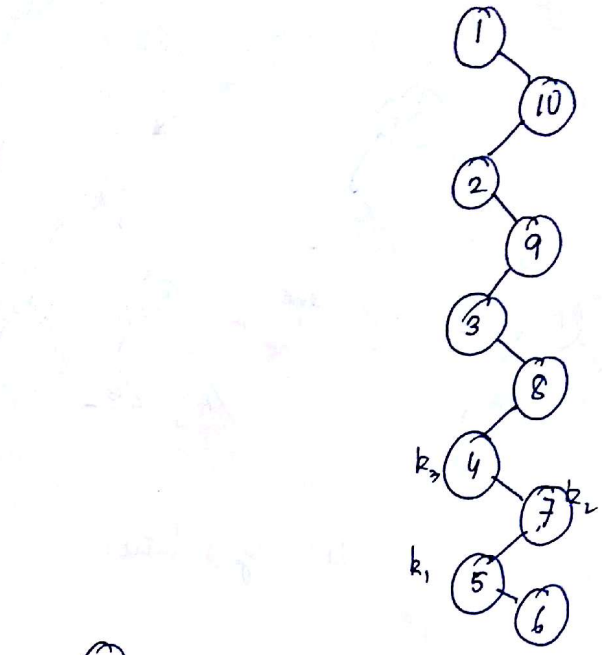


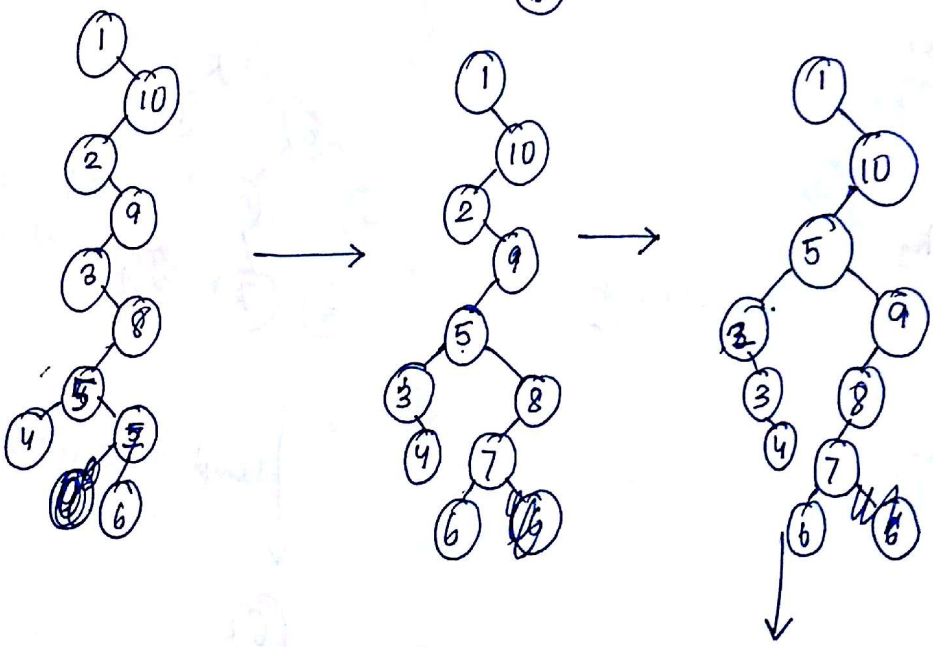
(6 is still not root)

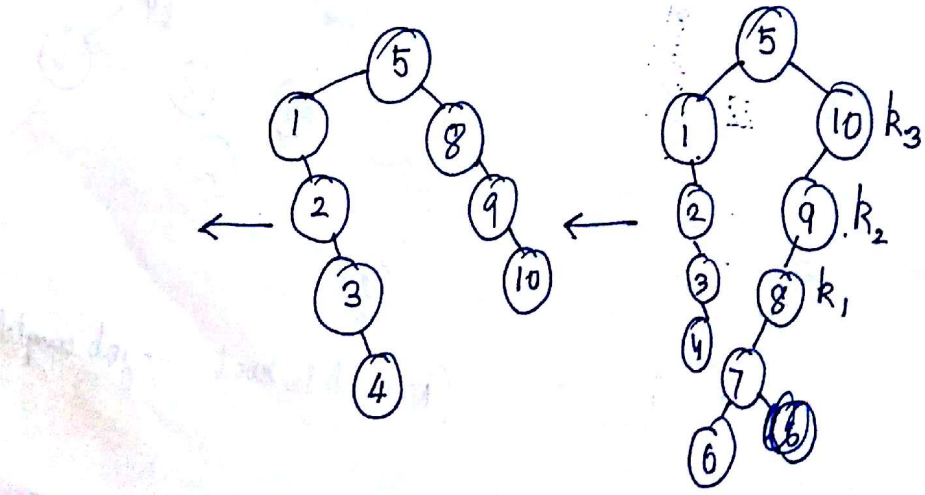
simple right rotation



(Now, 6 is root → job complete)



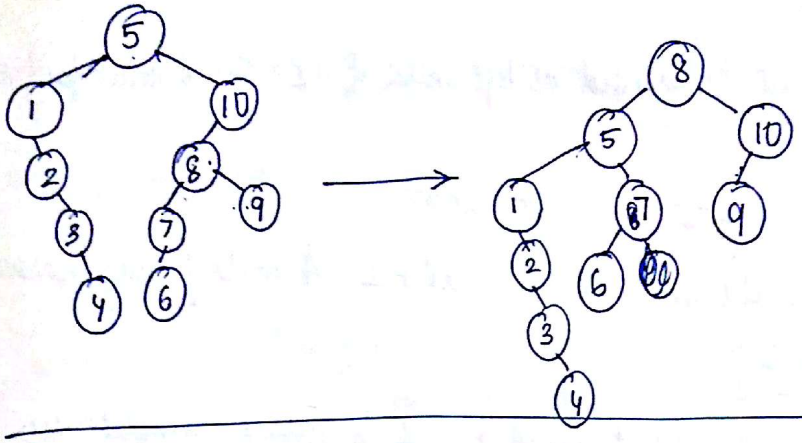




→ Heap
→ Heap
→ Heap

value

→ Hea



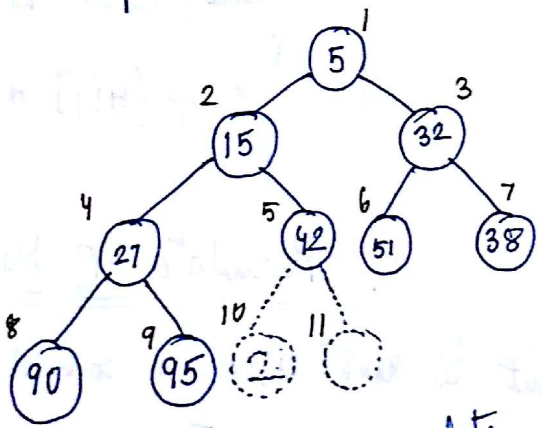
Heaps (implement heap using tree in Mids)

→ Heap is a complete binary tree with heap property.

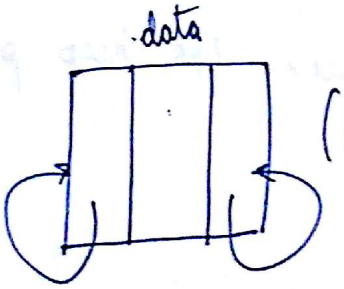
→ Heap generally means min heap.

→ Heap property - At every node the node's value is smaller than left child's value and right child value.

→ Heap is one of the priority queue (It is a special form of priority queue)



To identify heaps
→ First check complete binary tree.
→ Then check heap property.



(For BST)

The next child must be inserted as left child of 42. So no need for BST data structure.

We have to use array to implement heaps.

A node i has children in $2i$ and $2i+1$. A node j has parent $\lfloor j/2 \rfloor$.

Heap array $H[n]$.

Suppose ② is added as left child of 42 to ensure completeness.

Now, ensure heap property.

```
arrayp(H, n, s, k)
{
  j = s+1; s++; H[j] = k;
  while (j != 1)
  {
    if (H[j] < H[j/2]) swap(H[j], H[j/2]);
    j = j/2;
  }
}
```

② becomes root node.

Better code

```
while (H[j] > H[j/2] && j != 1)
{
  swap(H[j], H[j/2])
}
```

Percolate Up Method.

Deletion –

→ substitute bottommost element to root to ensure completeness

↔ Then check coming down for heap property.

er BST

arent $\lfloor j/2 \rfloor$

o.

& j != 1)

```
void PercolateDown(int H, int n, int s, int k)
{
    int j = 1;
    while (j <= s/2)
    {   cout << H[j]; s--;
        if (H[j] > H[2j] || H[j] > H[2j+1])
        {
            if (H[2j] > H[2j+1])
            {   swap(H[j], H[2j+1]);  j = 2j+1;
            }
            else {swap(H[j], H[2j]);
                  j = 2j
                 }
        }
    }
```

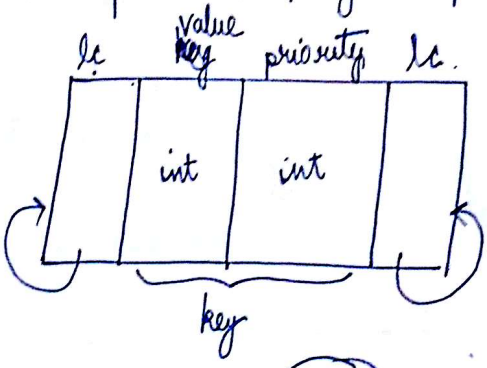
```
void func(Hptr T)
{   if (T->lc == T->rc == NULL) return;
    else
    {   if (T->data < T->lc->data && T->data < T->rc->data)
        {   func(T->lc); func(T->rc);
        }
        else
        {   cout << " Not heap "; return 1
        }
    }
}
```

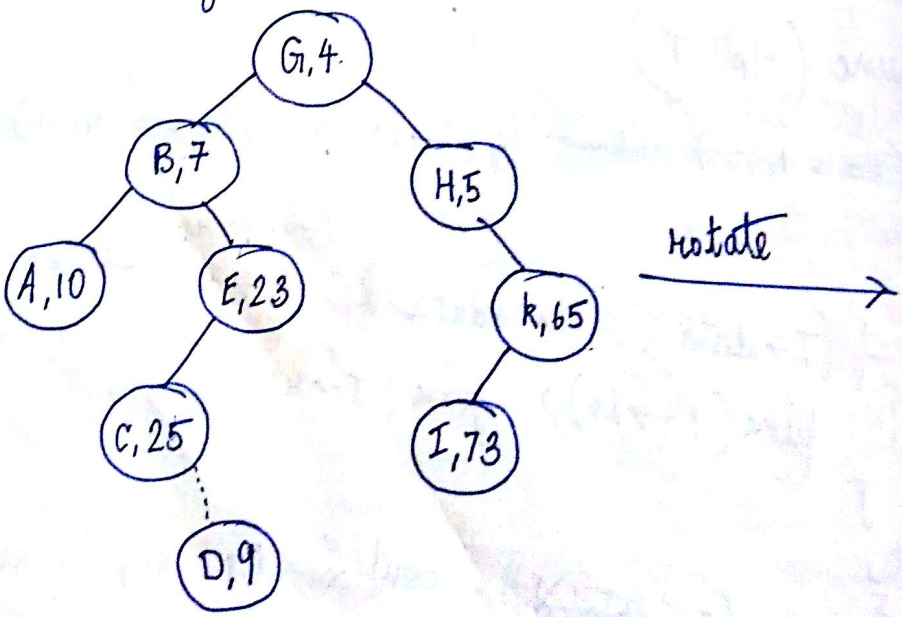
```
bool
void check (Bptr T1, Bptr T2)
{
    if (T1->data == T2 == NULL) return true;
    else → if (T1→ data == T2→ data)
    {
        return check (T1→lc, T2→lc) && check (T1→rc, T2→rc);
    }
    else return false;
}
```

Treap [in Mids]

Treap = BST + heap.

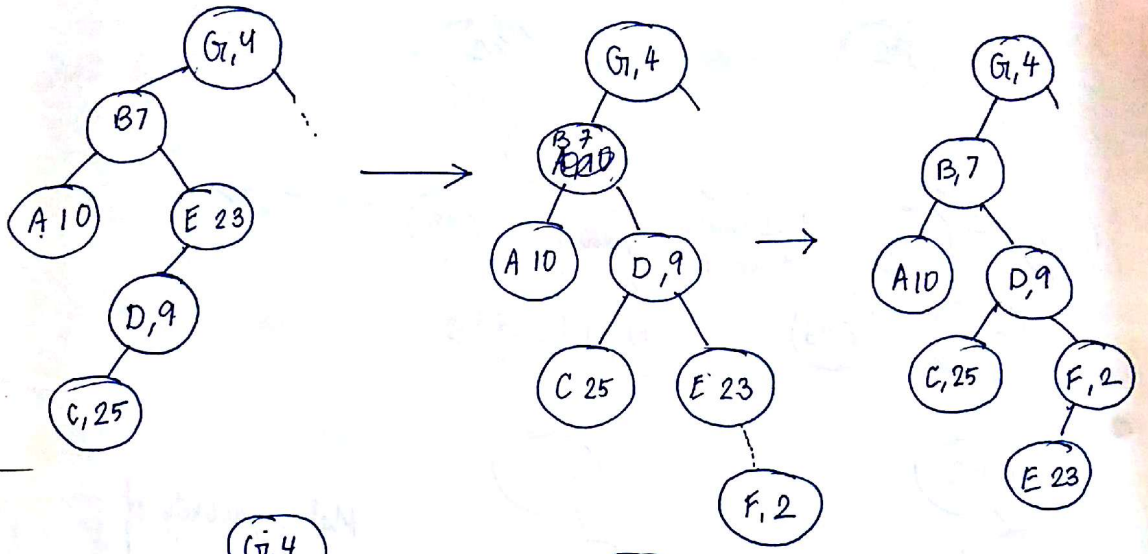
A node in treap has lc, rc, key and priority value.

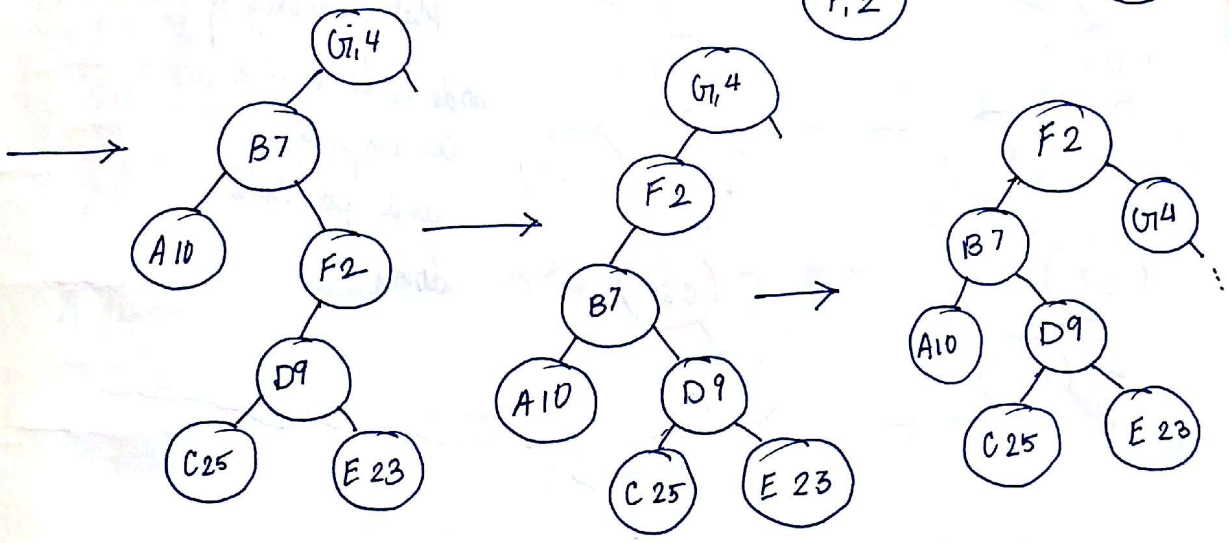




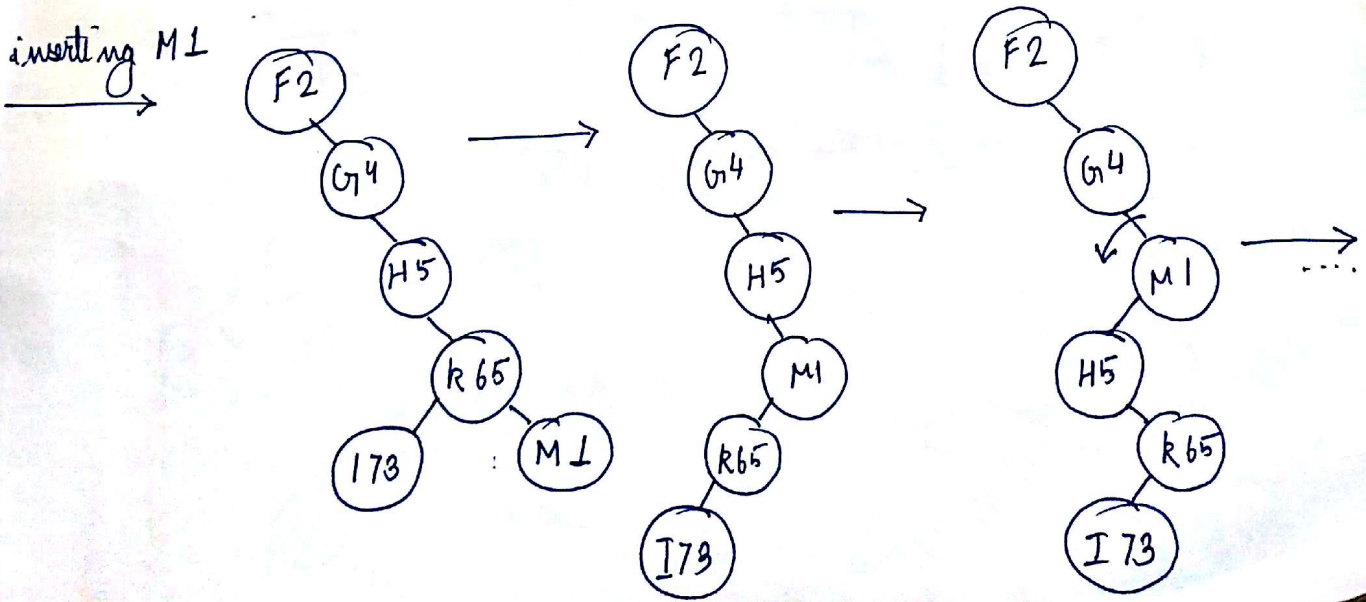
BST

While inserting in treap, insert as in ~~treap~~. Next you maintain heap property.

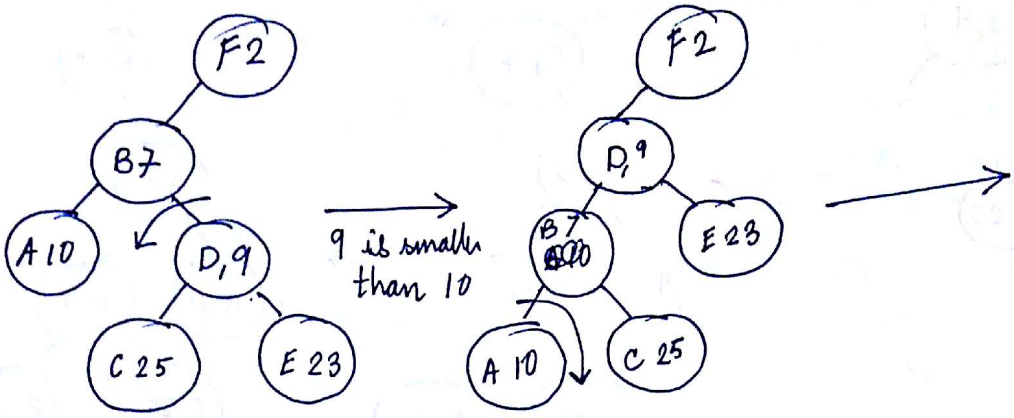


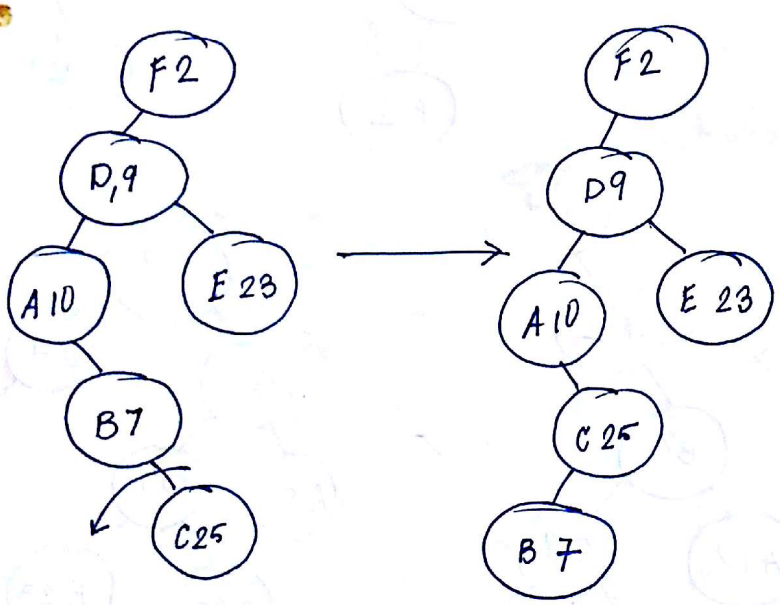


inserting M1



Deletion - (B, 7)





Make priority of node to be deleted as infinity and percolate down.

d-heap

[any node will contain at max d children].

If parent is at index i of array, then children are at di and di+d-1.

Disjoint Sets (used in graph algorithms)

1 2 5 7 8 4 6 9 3

(2,9) (3,6) (3,9) (7,5) (1,5)

Set-1 = {2,3,6,9} → 9

Set-2 = {1,5,7} → 1

Set-3 = {4} Set-4 = {8}

A disjoint set can be implemented using array S.

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

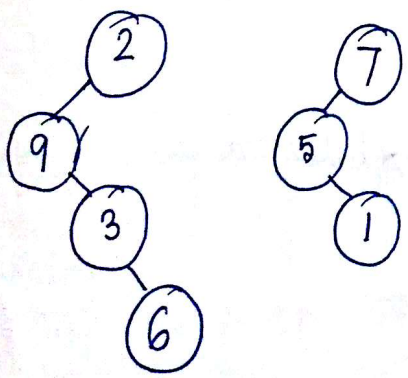
→ initially all 0.

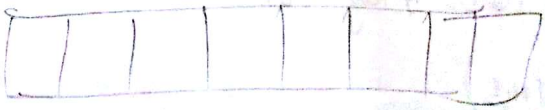
For a pair (i,j) S[j] = i.

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

(2,9), (3,9)

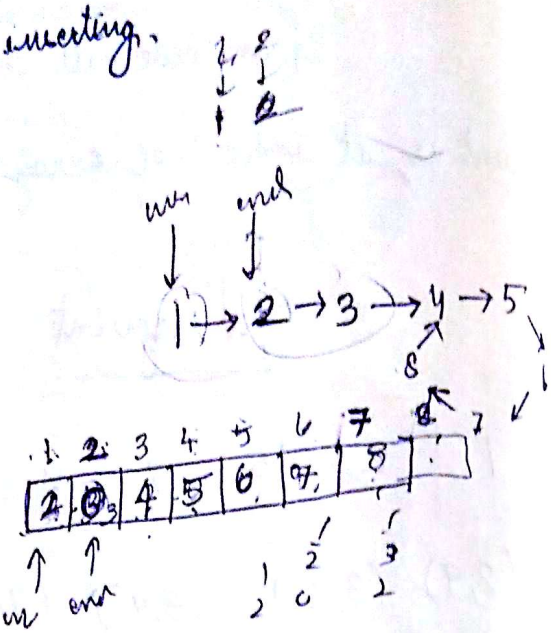
→ ∵ S[9] is filled we put S[3] = 9.





```
int find(int s[], i)
{
  if (s[i]==0) return i;
  else return find(s, s[i]);
}

void union(int s[], int i, int j)
{ if (s[j]==0) s[j]=i;
  else if (s[i]==0) s[i]=j;
  else s[find(s,i)] = find(s,j);
}
```

For inserting.



(Bayer's Tree) **B - Trees** (Bushy Tree) or (Balanced Tree) or (Boeing Tree)

B-Tree is a multiway search tree

Just like in m way search tree we mention count of a node, in B-Tree we use notation B_d-tree for telling count.

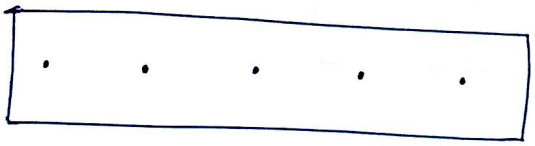
[d is minimum number of keys in a node, maximum is 2d].

Definition - ① No node except root can have less than d keys. ② All other nodes must have d keys (d+1 pointers) and can have maximum 2d keys (2d+1 pointers).

③ All leaf nodes must be at same level → balanced.

```
struct btNode {
    int *key;              int key[2*d];
    btNode* pointers;      btNode *ptr[2*d+1];
    int count;             int count;
};
typedef btNode * Bdptr;
```

4d+1 fields in B_d tree.
count field is just for our use.

 B_2 tree node [has 5 pointer]

B_2 tree construction grows from bottom to top.

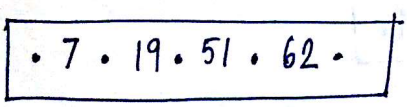
Construction - (62, 51, 7, 19, 49, 24, 57, 11, 1, 21, 39) Insert & Sort

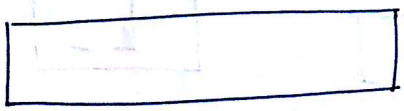
Overflow function -

While adding 49 overflow occurs,

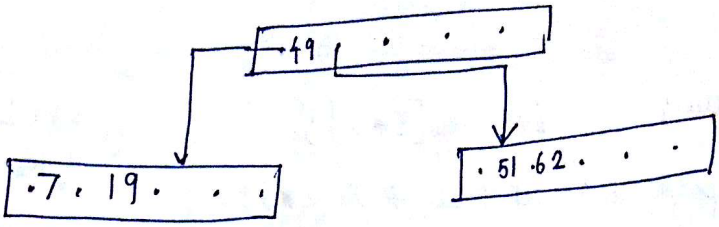
1. hard
2. Node splitting()
   1. create a new node
   2. find middle element of overflow node
   3. add left elements to original node and right elements to new node. Add mid element to parent node. If parent is not there create parent and adjust pointer. (Recursion)

Parents new elements left pointer → old node, right pointer → new node

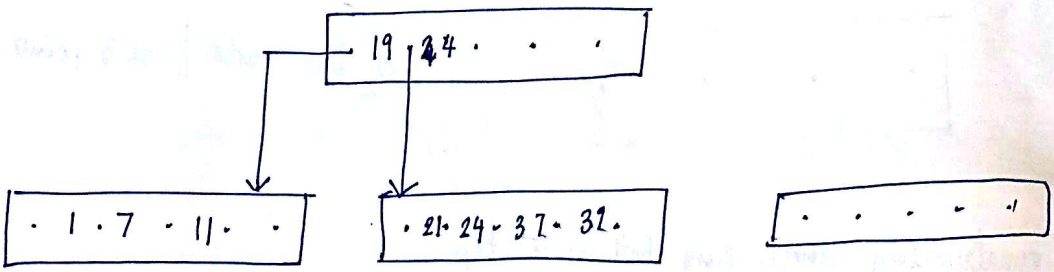




7 19 (49) 52 62



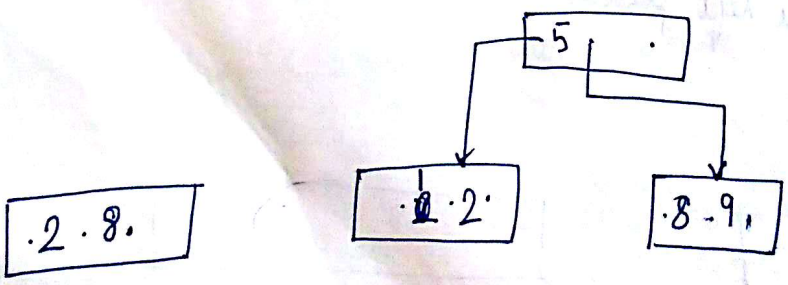
To add a node find leaf node where it should be added. To check if a node is leaf check its leftmost pointer.



Whenever overflow occurs do node splitting and insert-sort mid element to parent. While inserting shift keys and pointers accordingly.

B-tree → (3, 9, 1, 8, 2, 6, 4, 5, 7)

(2 8 5 9 1 6 4 7 3)

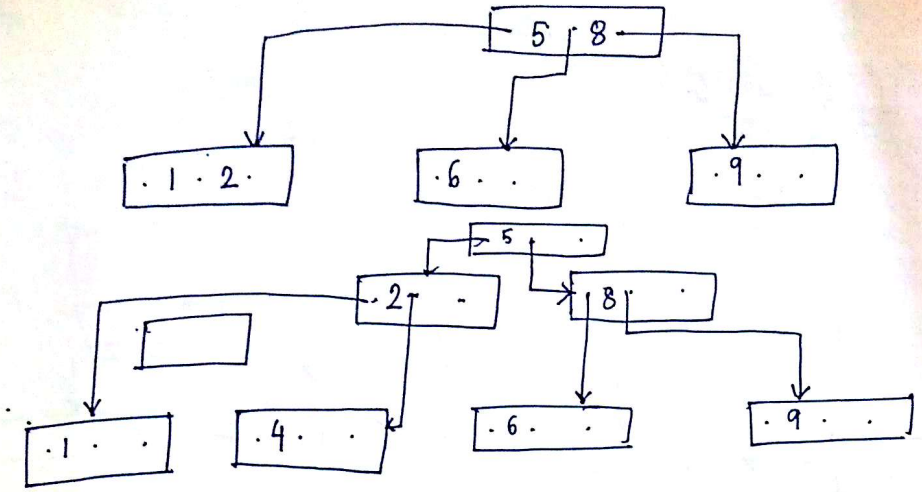


2 5 8

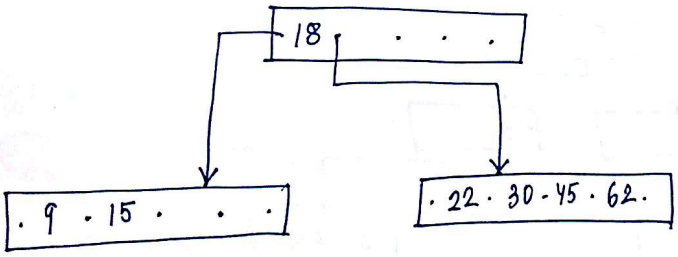
6 8 9

1, 2, 4

Scanned by CamScanner

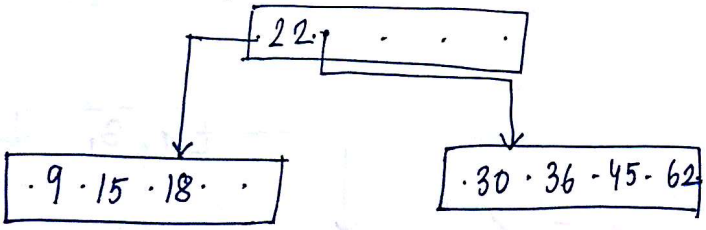


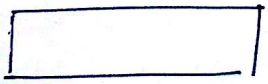
B_2 - tree Creation And Node splitting



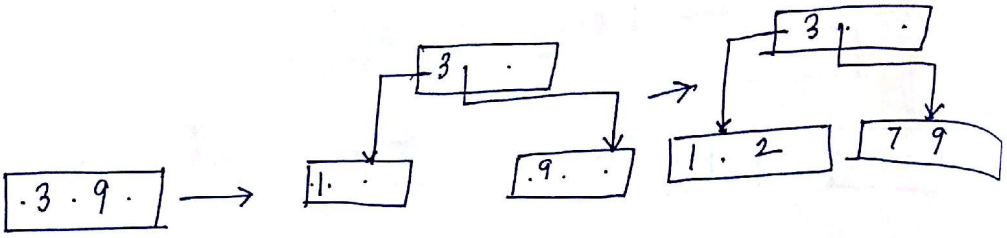
Overflow :-

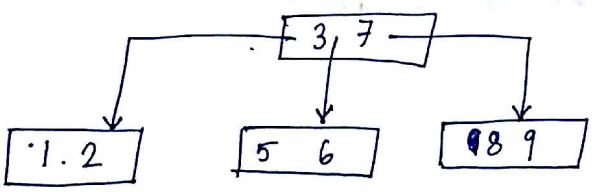
① Key redistribution → check if immediate left or right sibling has empty space. If yes then Key redi, else node splitting.

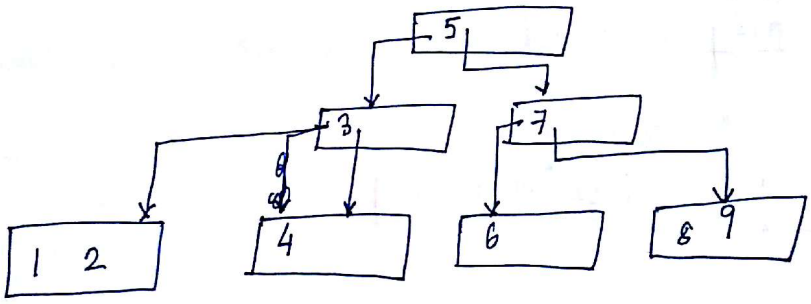




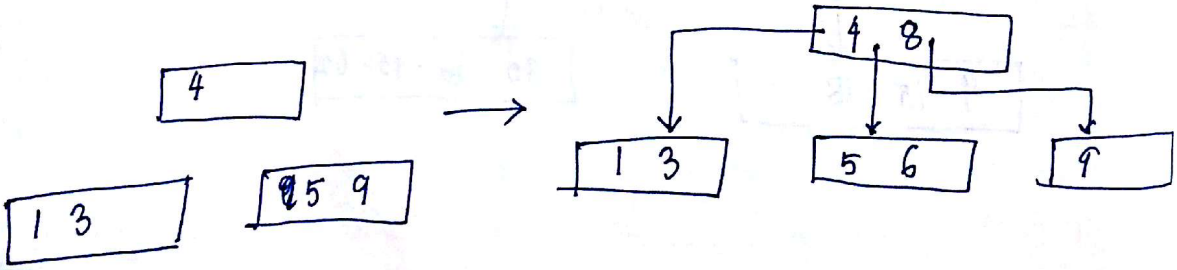
3 9 1 7 2 5 8 6 4

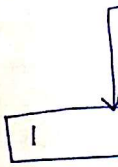


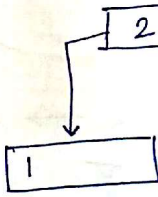




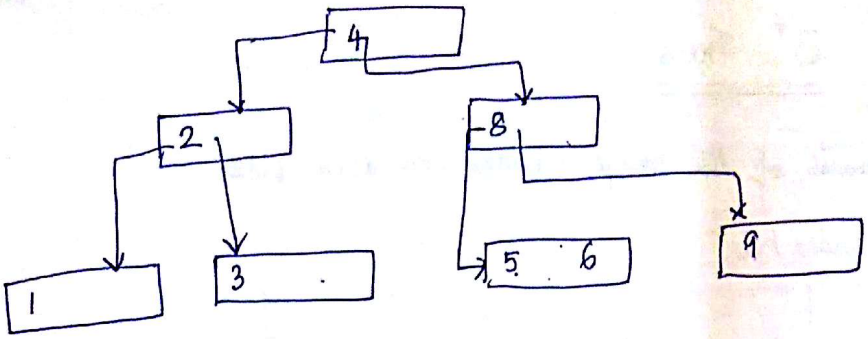
4 9 1 3 5 8 6 2 7

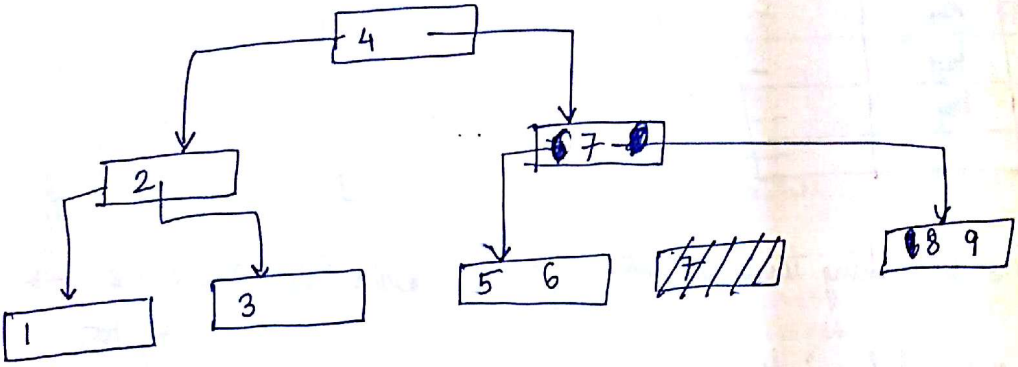






insert (T)
{
i = T→
while (d
{
T→k
T→p
i--
}
T→key [
}

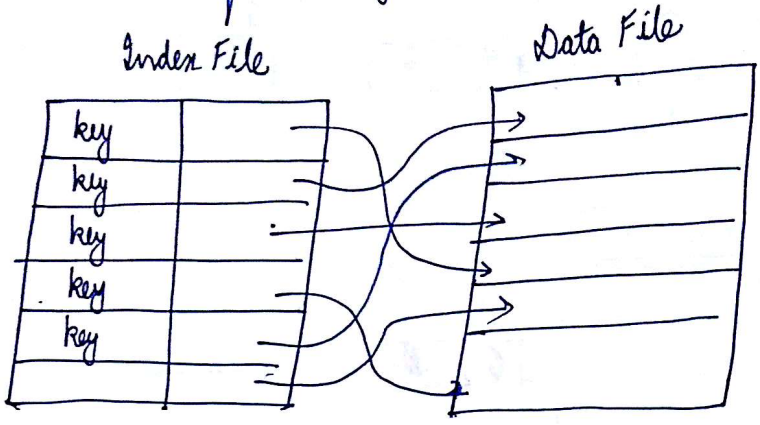





```
insert (T → Bptr &T, int d
{
    i = T → count;
    while (d < T → key[i-1] && i > 0)
    {
        T → key [i] = T → key [i-1];
        T → ptr [i] = T → ptr [i-1];
        i --;
    }
    T → key [i] = d;
}
```

B^+ - Trees

Index File consists of two things - index and data file.



In B_d trees searching time is $\log_{2d+1} n$.

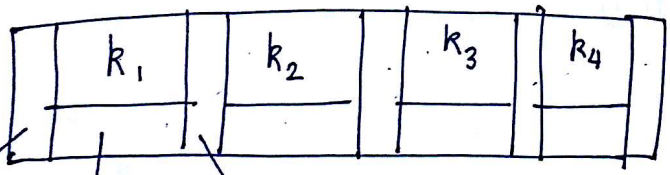
B^+ - Trees are used for indexing purpose.

Suppose we want to create B^+ - indexed tree.

Note: B_d tree is not used for searching.

Data File + Index File → B_d trees

B_2 tree indexed node structure -

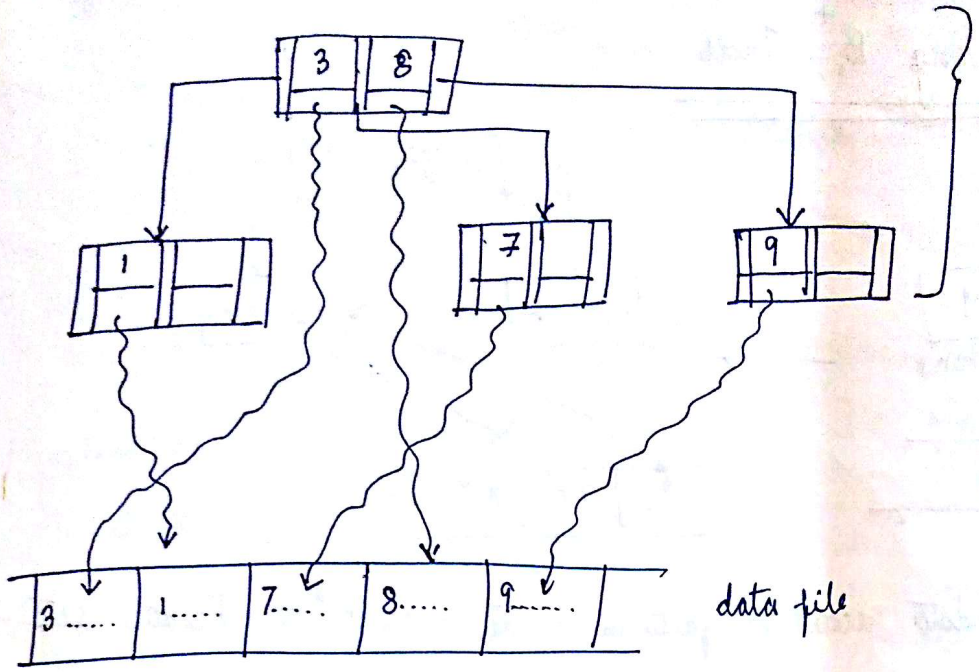


All records $< k_1$

record with key k_1

keys $> k_1$ and $< k_2$.

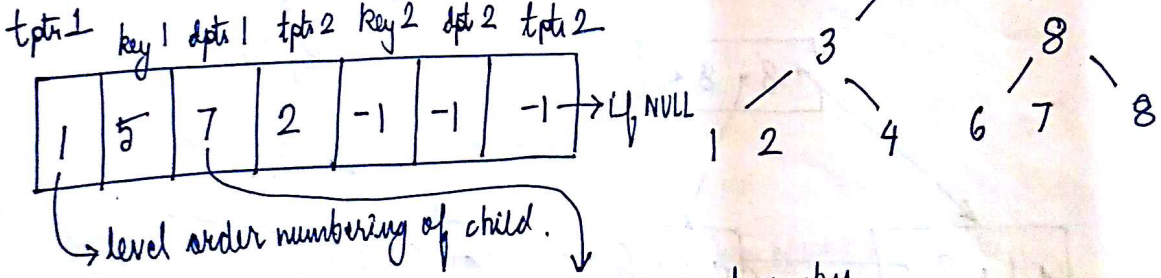
This pointer points to a record in data file.



B_1 - Tree index.

The moment we write B_1 tree index in harddisk it becomes B_1 tree index file.

Each record in B_1 tree index file has 7 fields (6.1+1).



Index Node

↓

Non leaf nodes → has only index

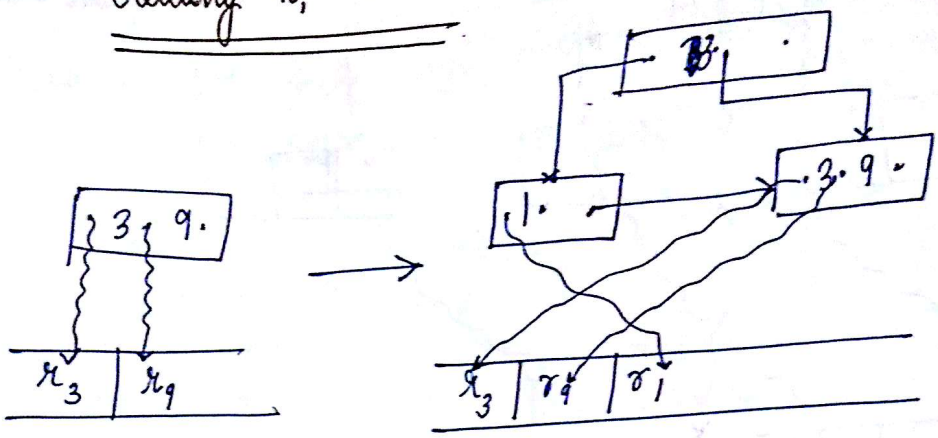
leaf nodes → has only data pointe

} B_+ tree has heterogeneous structure

(Data Node)

Total number of fields in B_+ tree = $4d + 1$.

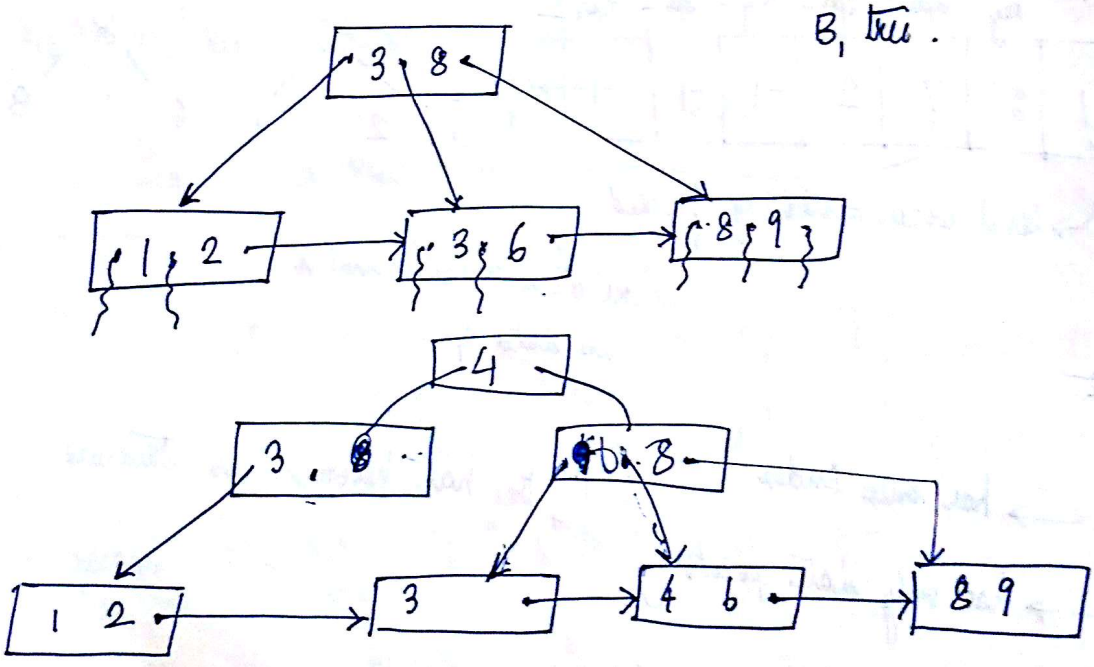
Creating B_1^+ - Trees

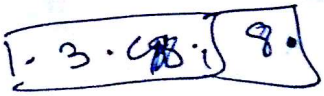


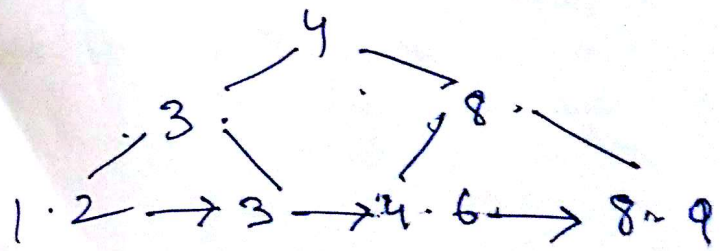
V. Imp- Retrieving data records is faste in B_1-tree than B_1^+ tree because data pointer is there itself.

But B_1 tree is used for static data structure

Non-leaf node splitting is same as B_1 tree.







struct

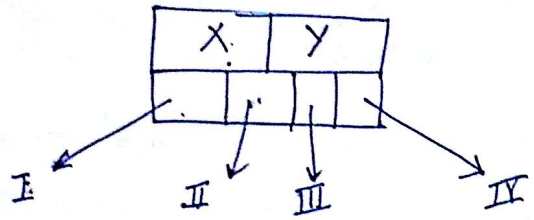
typedef

void

{

}

Quad Trees



Always insert at leaf.

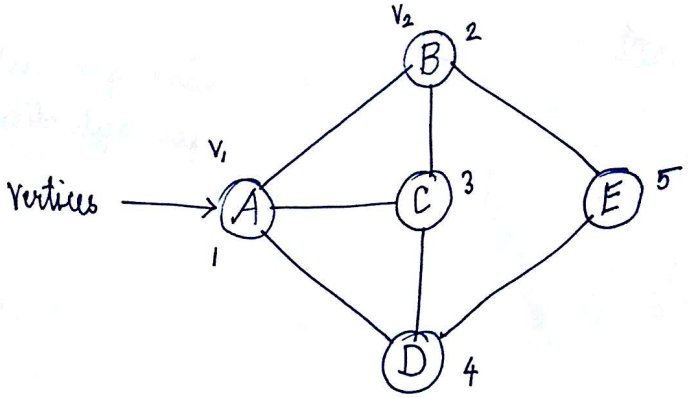
Look upon internet for applications.

```
struct qNode{
        int x, y;
        qNode * pointer[4];
          };
typedef qNode* Qptr;
void insert (Qptr &T, int x, int y)
{
   if (T==NULL)
   {
      T= new qNode;
      T→x = x; T→y = y;
      for (int i=0; i<4; i++) T→point[i] = NULL;
      return;
   }
   if (x<T→x && y<T→y) insert (T→point[0], x, y);
  else if (x<T→x && y>T→y) insert (T→point[1], x, y);
  else if (x>T→x && y>T→y) insert (T→point[2], x, y);
  else insert (T→point[3], x, y);
}
```

Graphs

- Undirected
- Directed

A, B, C.. → value of vertex

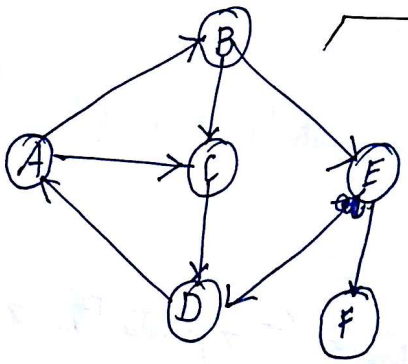


Vertices $V = \{V_1, V_2 \ldots V_5\}$

Edges $E = \{(u,v) / u, v \in V\}$

adjacent

If edges are labelled then graph is weighted.



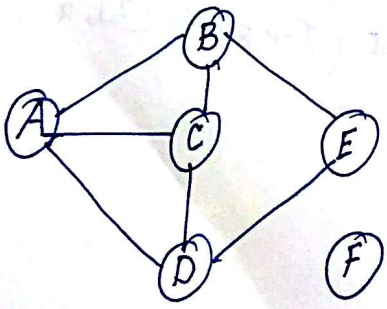
→ unweighted weekly connected directed unweighted graph.

B is adjacent to A, not vice-versa.

C adj to A.

$i \rightarrow j$ means j is adjacent to i.

Connected graph is used for undirected.



excluding F → unweighted connected

including F → unweighted unconnected

In a directed graph if we can reach from any vertex to other then → strongly connected graph. otherwise weekly connected graph.

vertex

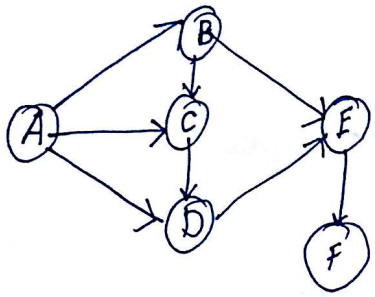
Path :- In undirected graph, each edge contributes 1 to path length.

In directed graph, weight of each edge contributes to path length.

If a directed graph has cycles it is cyclic. Mostly strongly connected graph is cyclic.

A cycle is a path from i to i whose pathlength is more than 1.

eighted.



excluding F → directed acyclic graph (DAG)
↓
neither weakly connected
neither strongly "

including F → weekly connected DAG.

The number of edges incident on a vertex is its degree.

In directed graph- Indegree (Fan-in) Outdegree (Fan-out).

Topological sort is only for DAG.

Representation of matrix :- size = $|V|^2$

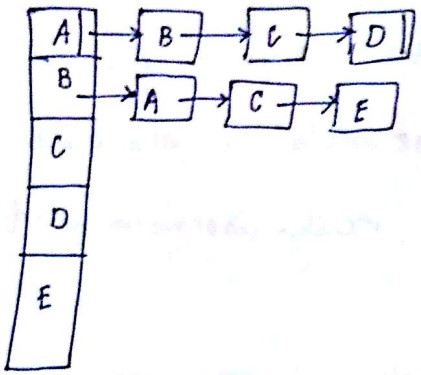
	1	2	3	4	5
1	1	1	1	1	0
2	1	1	1	0	1
3	1	1	1	1	0
4	1	0	1	1	1
5	0	1	0	1	1

adjacency matrix of undirected graph is symmetric.

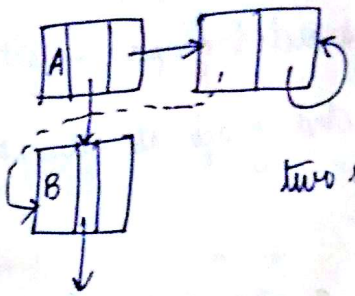
represent using A in program

$|V| \times |V|$

Adjacency list :- (requires only 1 structure) represented by G1.

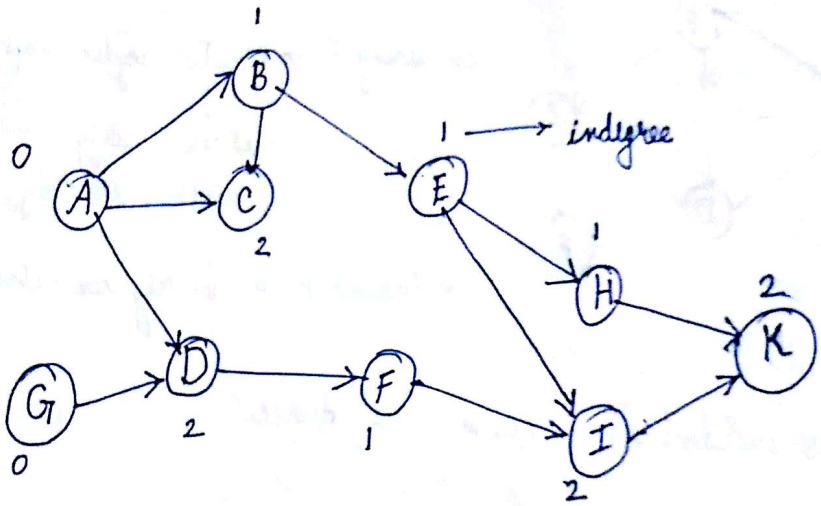


For saving memory we can use two structures.



two structures

size = |V| + 2|E|.



Topological Sort -

Calculate indegree of each vertex and store in indegree array .[].

	A	B	C	D	E	F	G	H	I	J
a[] =	0	1	2	2	1	1	0	1	2	2

Check for 0. → print "A".

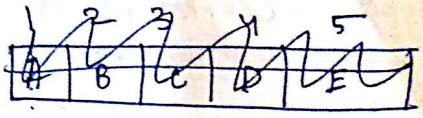
Delete vertex A by decreasing degrees of every vertex adjacent to A by.

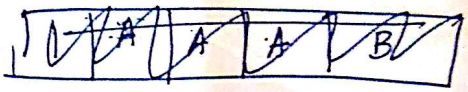
① a []
② Queue
③ Print
④ Find

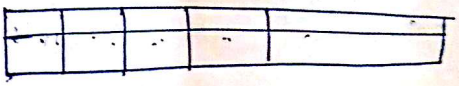
This

P[] B

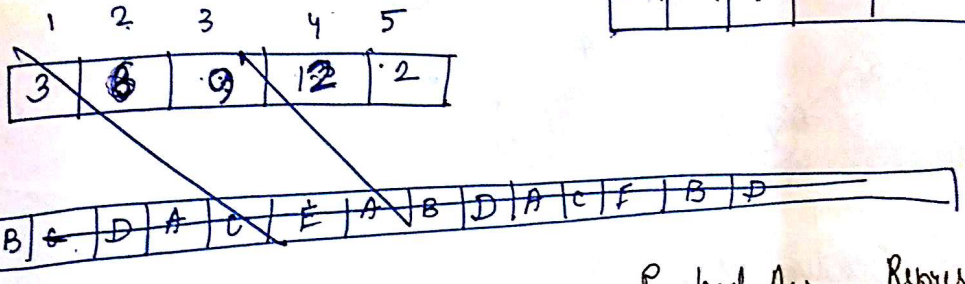
① a [ ]
② Queues.
③ Print all paths.
④ Find cyclic or acyclic.



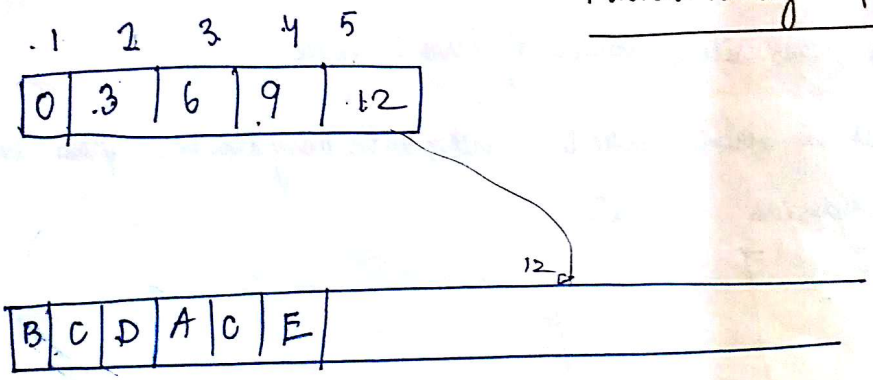




Third representation

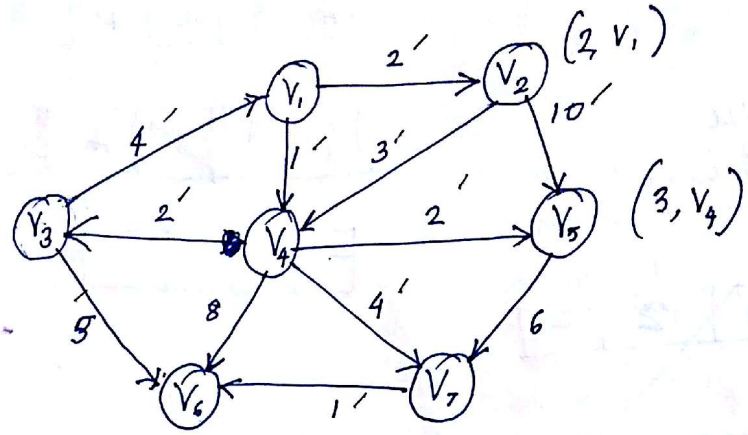


Packed Array Representation



A by 1.

Dijkstra's Algorithm for Shortest Path



source vertex, destination vertex, current.

Update adjacent, unvisited vertices of current node.

Among all unvisited vertices, take the vertex with many distance from sour.

distance	previous	visited
[ ]	[ ]	[ ]

General Tree -

For level order → D Ns Fc

D Fs Ns for A

B

D

J

C

Traversal in graphs :-

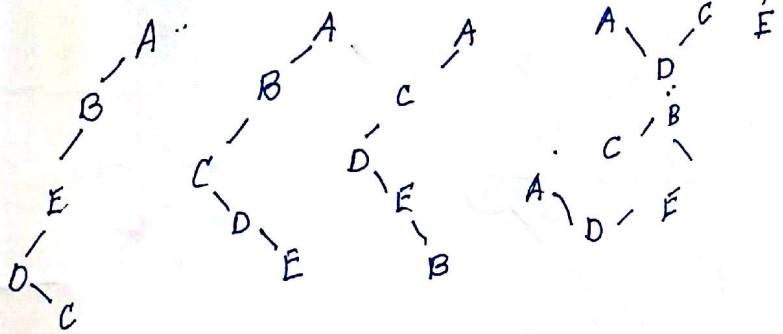
Level order → First print node and then visit all of its adjacent vertices.

① BFT (Breadth First Search) → Traversal

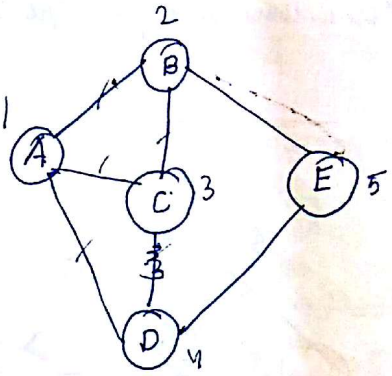
② DFT (Depth First Traversal) → easier in recursion

For graph, compulsorily give starting vertex.

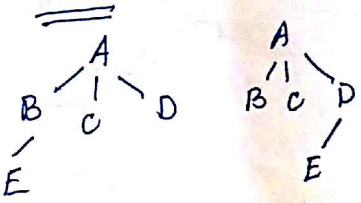
DFT :- (V.Imp) → no. of DFT possible



6 possibilities



BFT (2 possibilities)



Starting from E

E → B → A → C → D

E → D → C → B → A

E → B → C → D → A

E → B → A → D → C

E → D → A → B → C

E → D → A → C → B

E → B → C → A → D

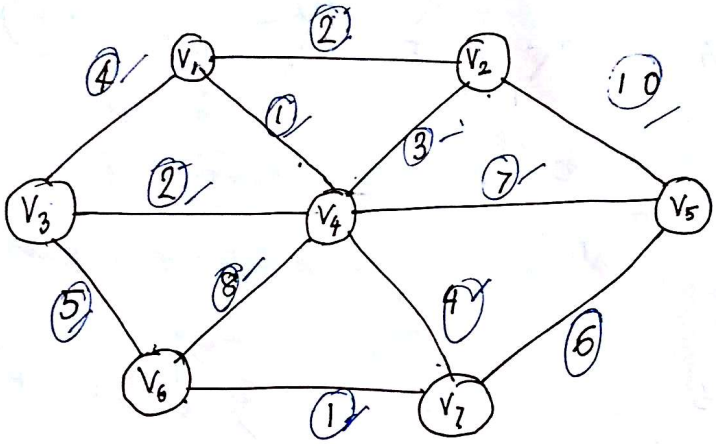
```
void dfs(v)
{
    visit(v) = 1;
    Print(v);
    for each adjacent vertices w of v,
        if (!visited(w))
        {
            dfs(w);
        }
}
```

BFT :-

A B C D

As soon it enters queue, mark it visited.

For unconnected graph enque separate initial vertex



Applications of graphs-

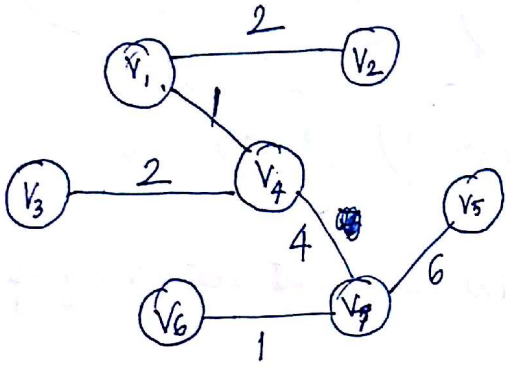
Traversal / Graph	BFT	DFT
Undirected	Prim	Euler Path
Directed	Dijkstra's Alg Topological Sort.	Strongly Connected Component

Minimum Spanning Trees - (tree with minimum sum of edges and no loop).

1. Prim's Algorithm
2. Kruskal's Algorithm.

(1) Prim's Algorithm -

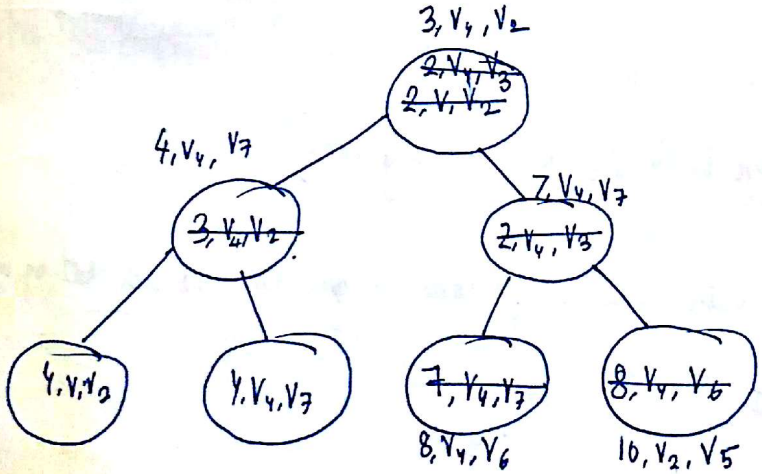
Input as starting (vertex).



sequence of edges selected in finding minimum spanning tree.

V_1 V_4
V_1 V_2
V_4 V_3
V_4 V_7
V_7 V_6
V_7 V_5

Using heap:- (always keep track of last visited vertex).



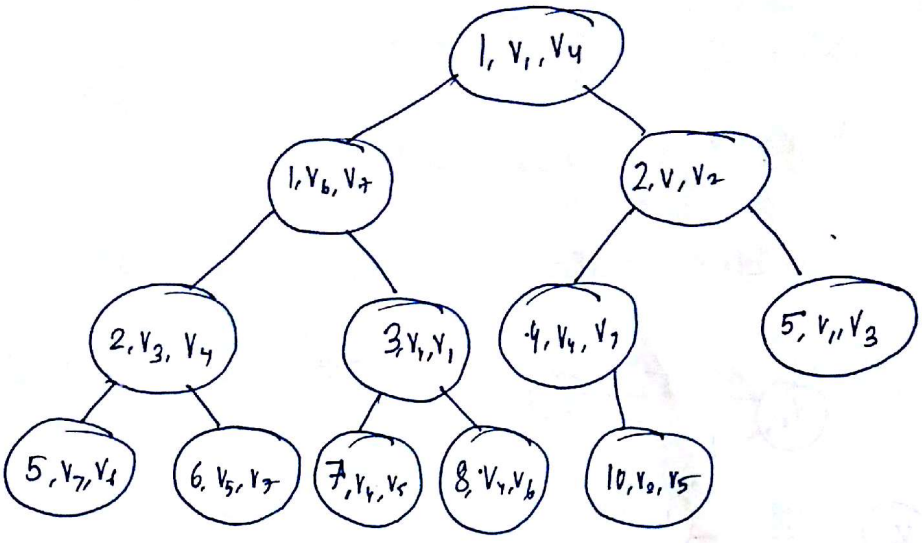
1, V1, V4

2, V1, V2

2, V4, V6

Add all edge weights and adjacent unvisited vertices of present visited vertex (V_4). Delete the minimum.

① Insert all edges in heap while creating graph.



Delete from heap and if the edge does not contain any of visited vertices store it somewhere.

If the deleted ~~eg~~ edge is adjacent to only one of the visited vertices, add that edge to MST.

If deleted edge contains ~~only~~ two vertices as visited then discard as cycle is formed.

② Kruskal — (unlike starting vertex in Prim's, use starting edge).

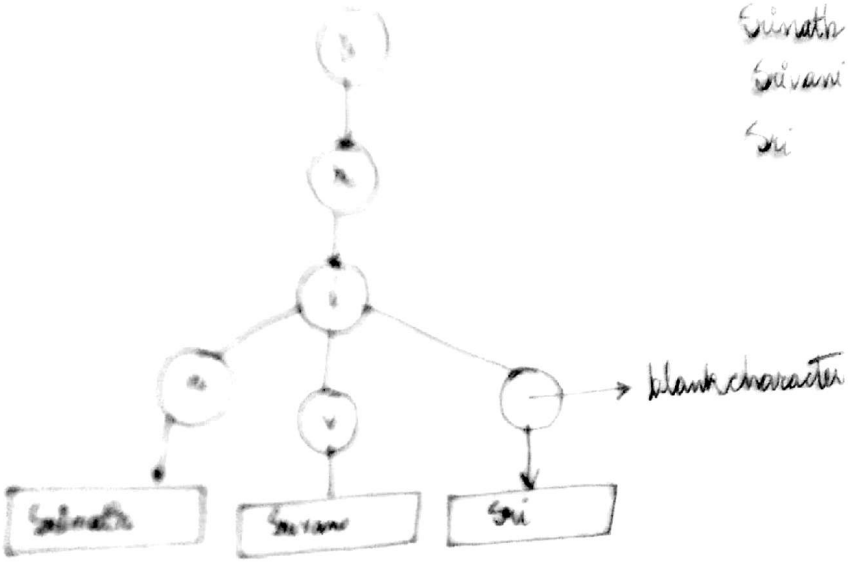
Use disjoint sets to know that both vertices of edge ~~is~~ are in same set or not

If same set dont add that set.

Assignment

1. Trie with DJ change
2. Trie using heap
3. Radix

Trie / Digital tree



The letters present in key are called Alphabet (denoted by A[])

→ blank character

A = { A, B, C, }

If the key have a letter of different type A[] will have n+1 entries.

AAB

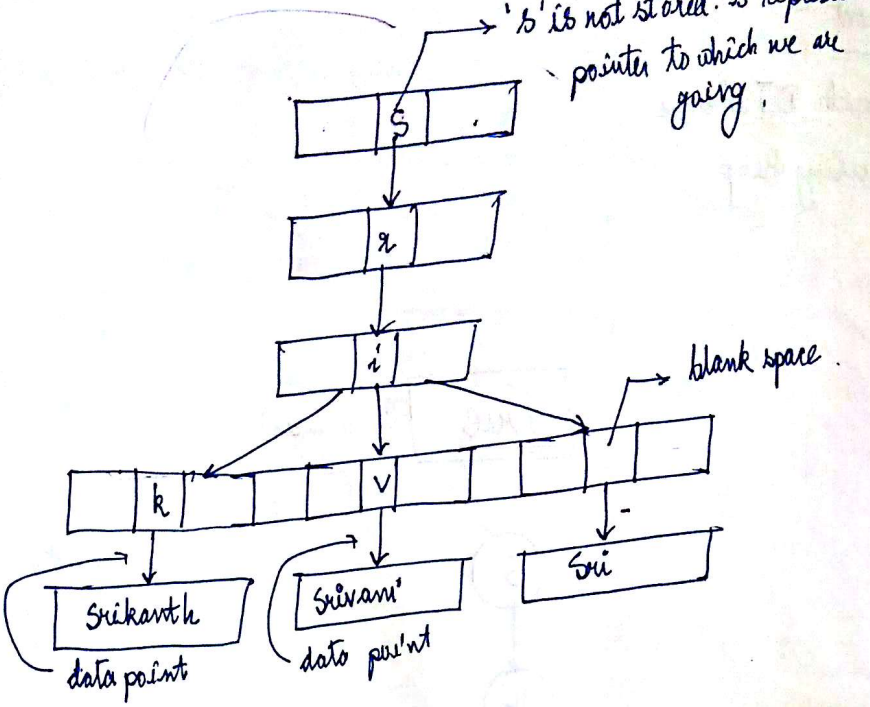
ABBB C

BAA

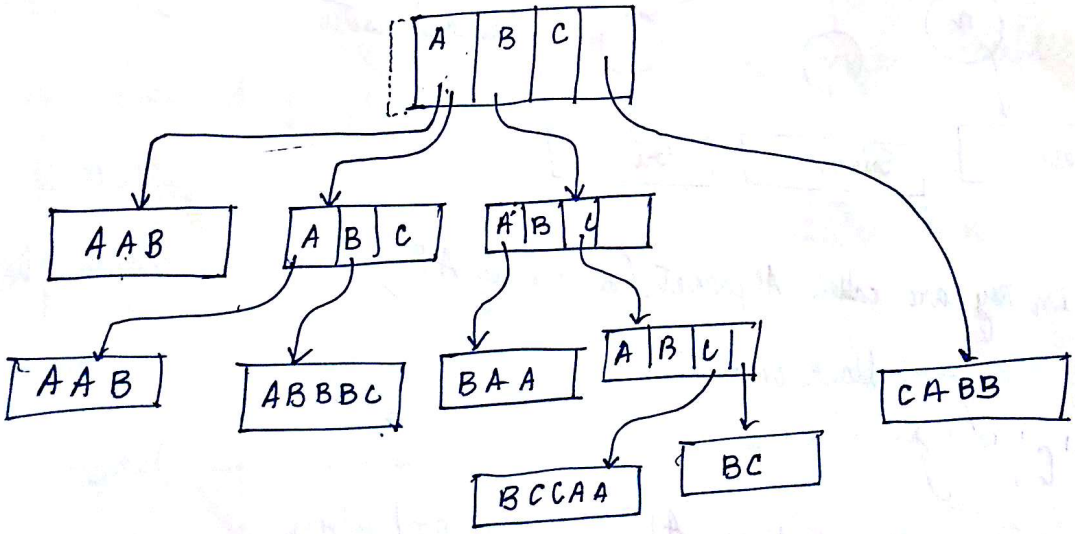
BCCAA

BC

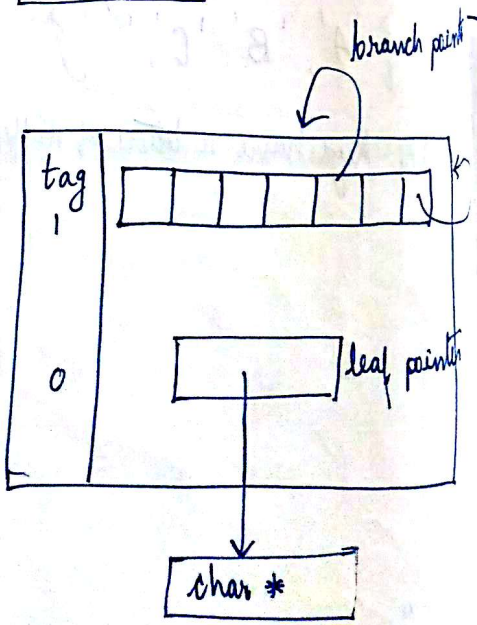
CABB

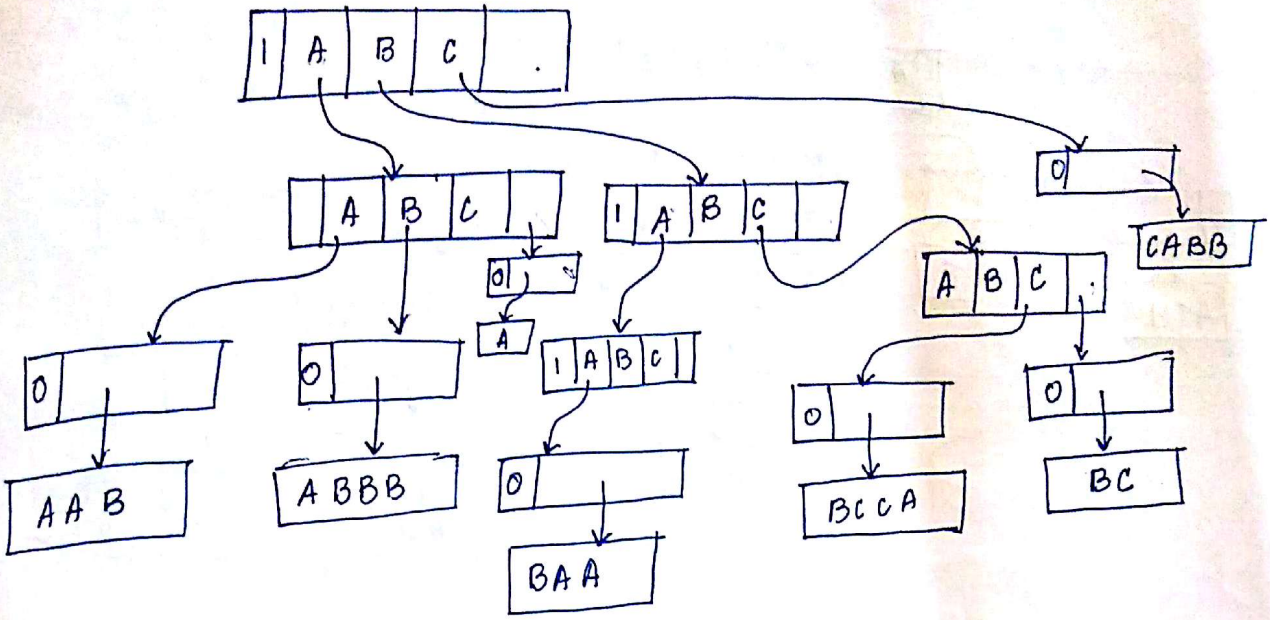


Branch Node Structure -

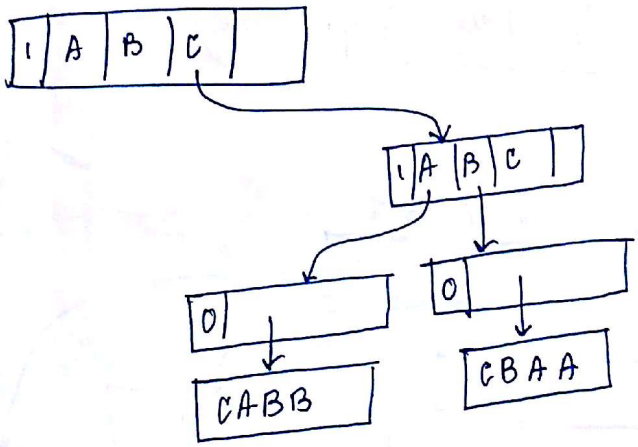



```
struct trienode {
    int tag;
.
```

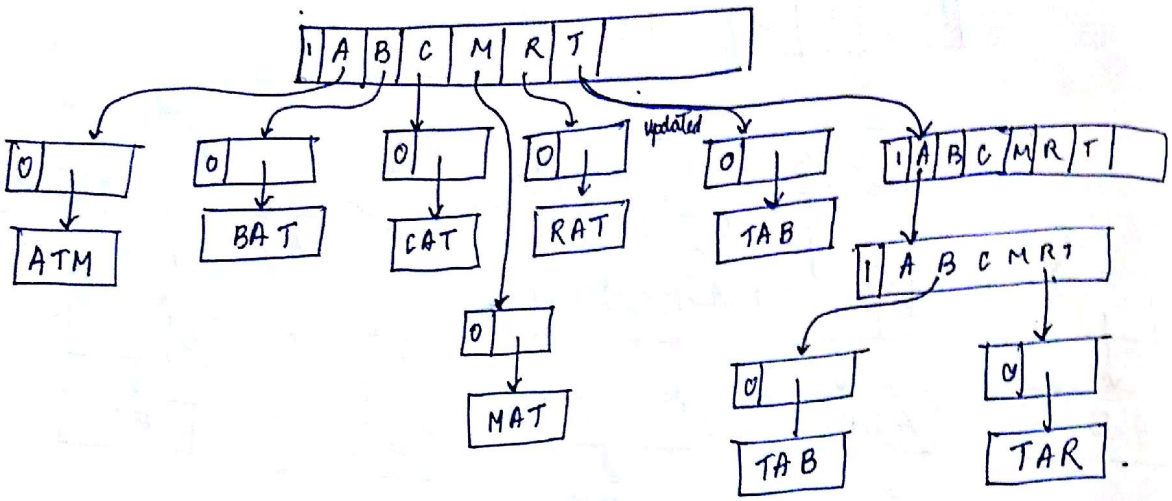




Inserting CBAA,



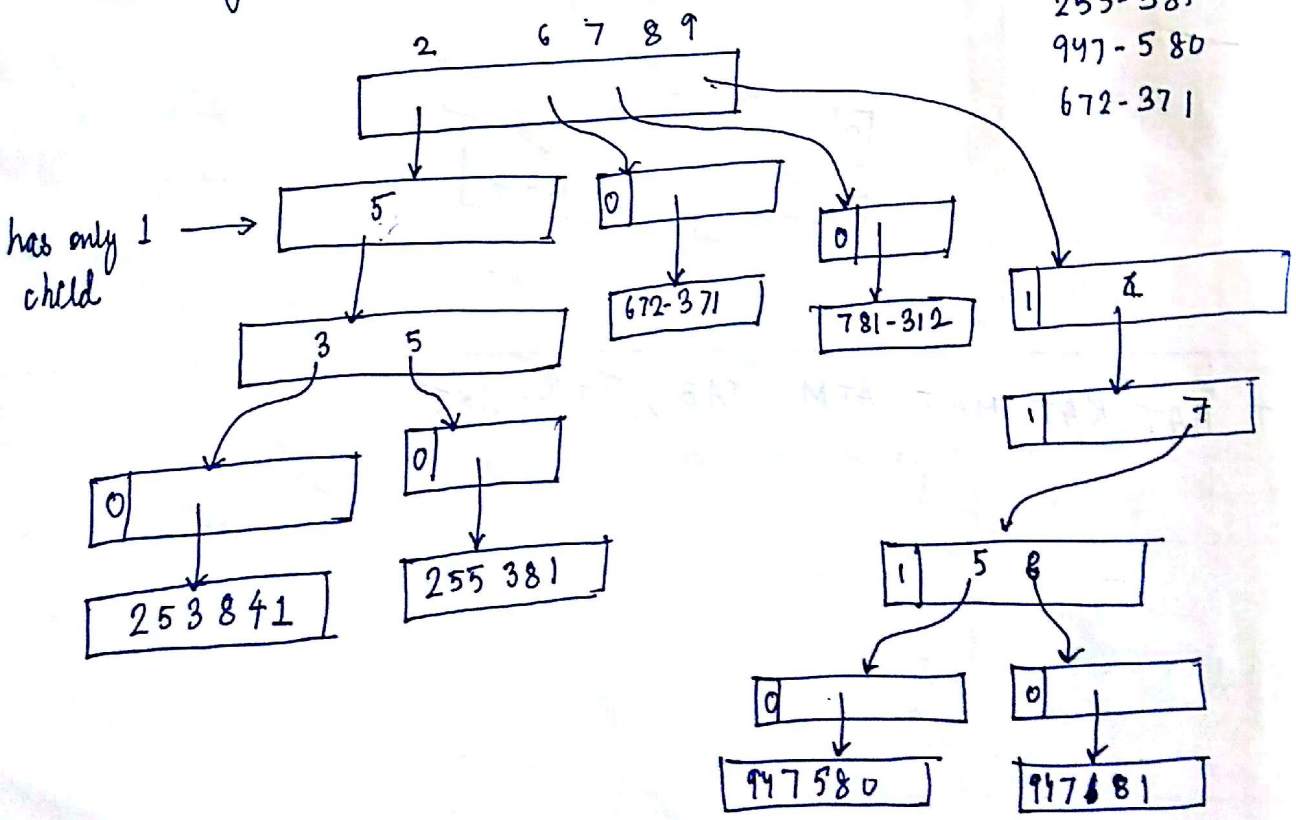
CAT BAT RAT MAT ATM TAB, TAR, ART,



C-Trie (Compact Trie)

253-841
781-312
947-81
255-381
947-580
672-371

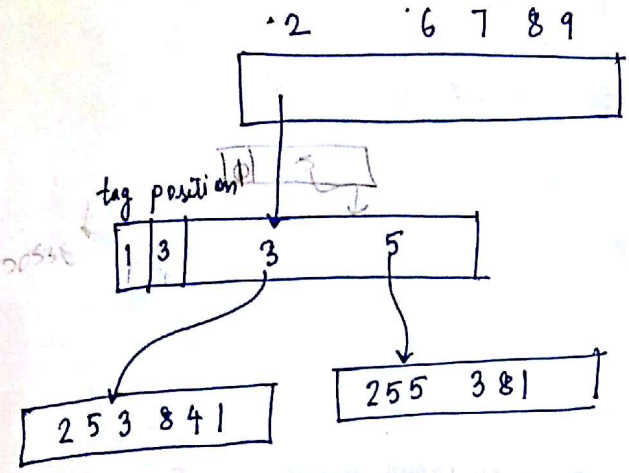
Constructing Trie:



In a C-trie every branch node should have atleast 2 children.

This is done by 3 methods :—

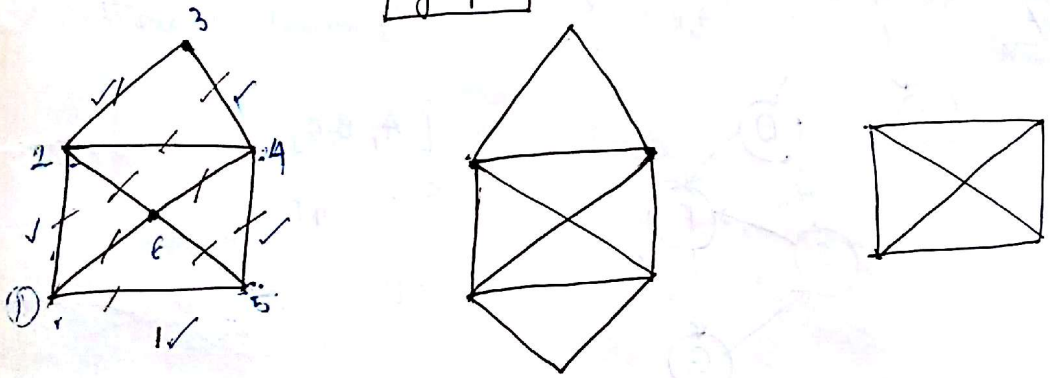
① Character position method - store position or index of string in branch node.



2 5 3 8

1-2-3-4-5 → 1-4-2-

Graphs



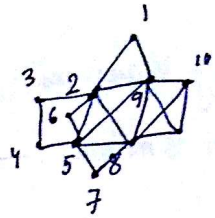
unweighted undirected connected graphs

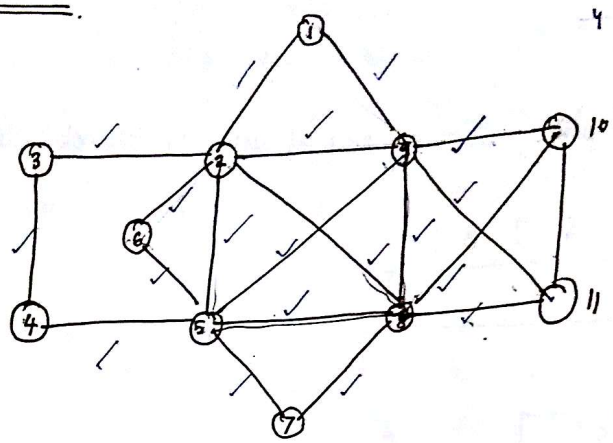
→ All vertices must have even degree → start and end at ~~to~~ same vertex

OR

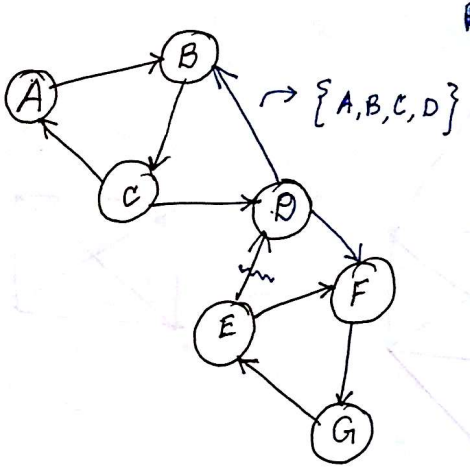
→ Exactly two vertices must have odd degree → start and end at two different odd degree vertices.

Euler's Path :-





10 – 9 – 1 – 2 – 9 – 11 – 8 – 10 – 11 .



Unweighted, weekly connected

Strongly connected components,

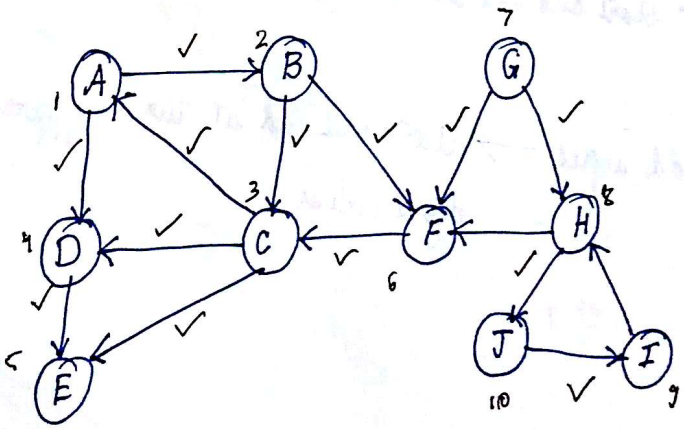
{A, B, C, D}

{A, B, C}

{E, F, G}

{D}.

Strongly Connected Components –



Directed, weekly connected

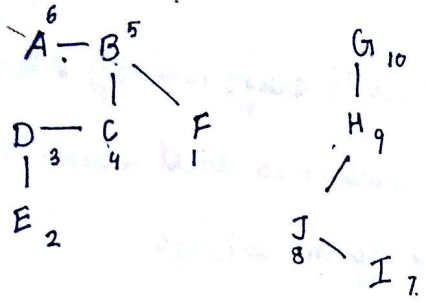
{A, B, C, F}

{H, I J}

{G} {D} {E}

Algorithm - (3 steps)

① do DFT and number vertices in post-order.



② reverse the graph

③ do DFT on graph. (start on higher number vertex).

{ G }

{ H, I, J }.

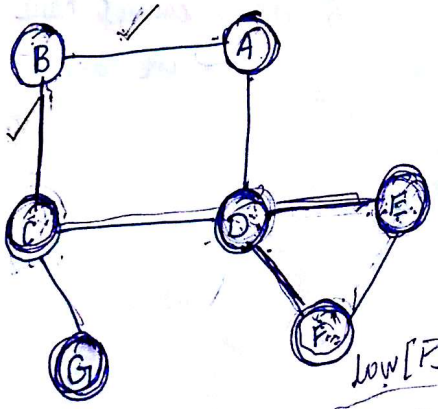
{ A, C, B, F }.

{ D }

{ E }.

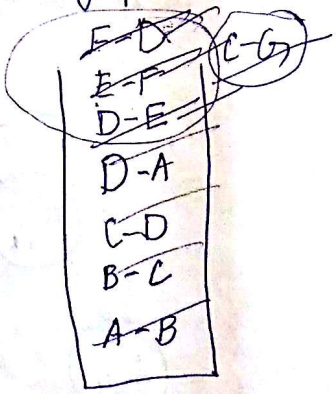
Articulation Point - Cut-vertices.

Biconnected component - Removal of any vertex still keeps the ~~vertex~~ graph connected.



low[F] ≥ dfn[E]

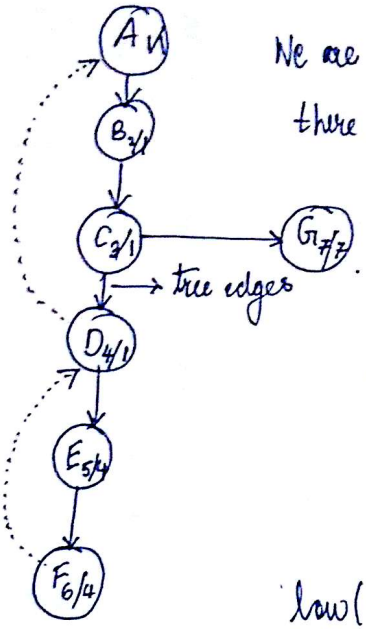
low[E] ≥ dfn[D]



C-G

Algorithm -

① Ignore preorder numbering after dfs.



We are creating a dft tree. From D we can go again to A so there is a backedge

Write a function to assign numbering to vertex.

Calculate low(v) which is lowest numbered vertex you can reach using only one back edge.

low(E) = 4.

low(G1) = 7.

low(v) :- minimum of

(1) num(v)

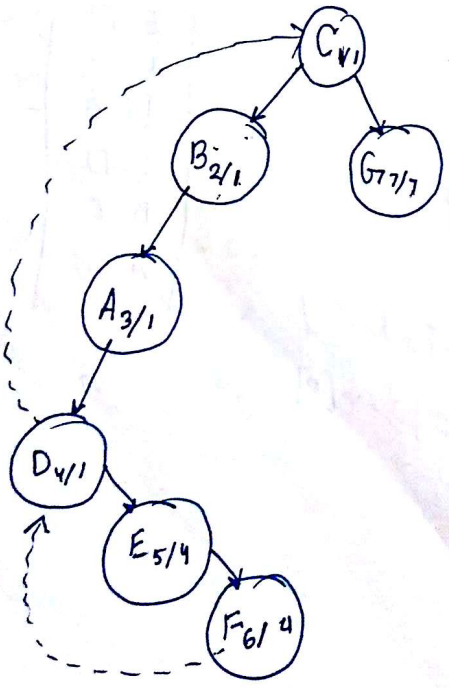
(2) min low(w) if v → w is an edge.

(3) w is backedge of v take num(w)

v is an articulation point if low(w) ≥ num(v) in the tree.

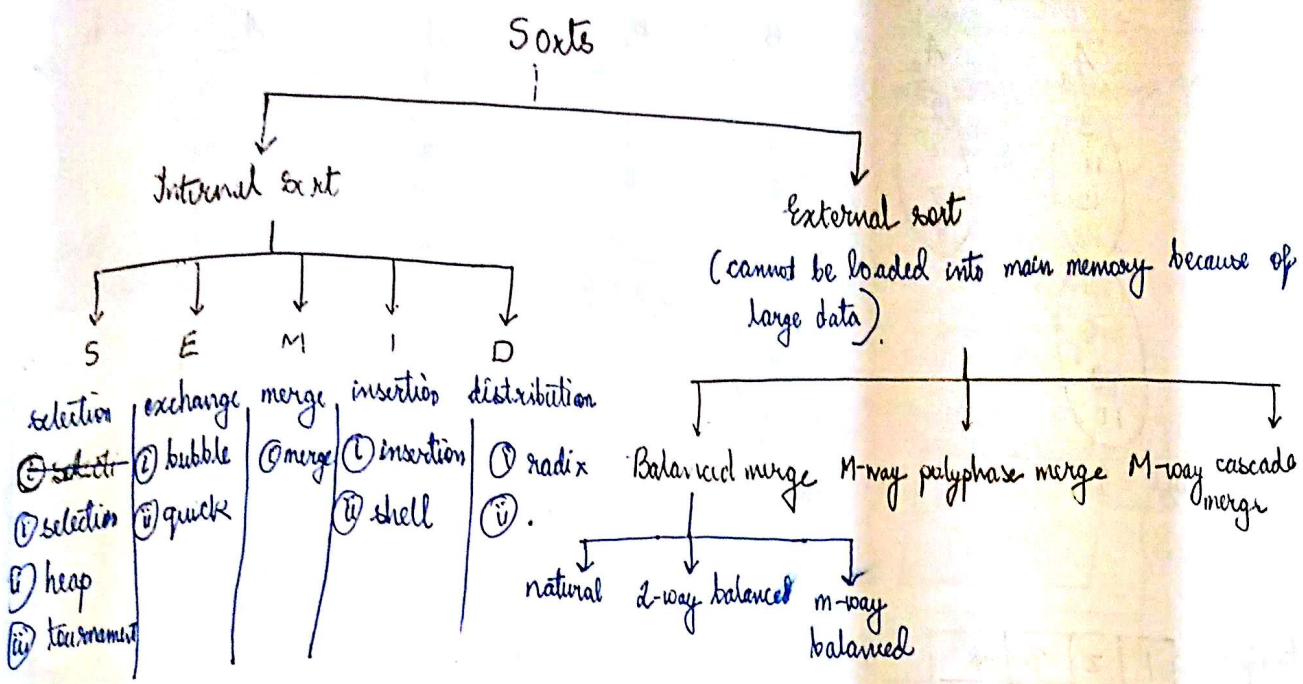
v and w are adjacent vertices.

If v is a root and has one child then the rule is relaxed.



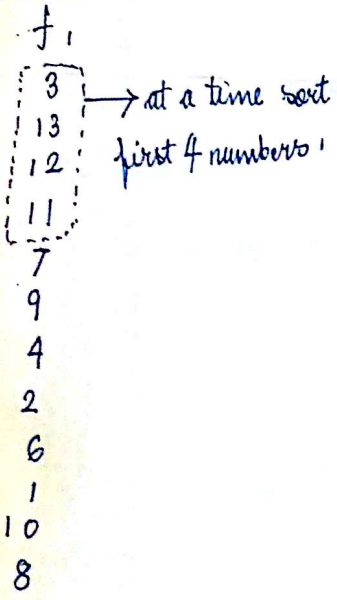
A vertex cannot have more than one back edge

Sorting (questions on either FS or OS)

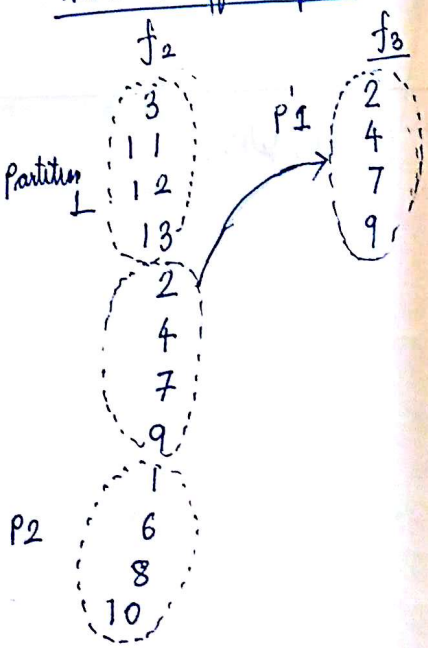


External Sort

File contains.

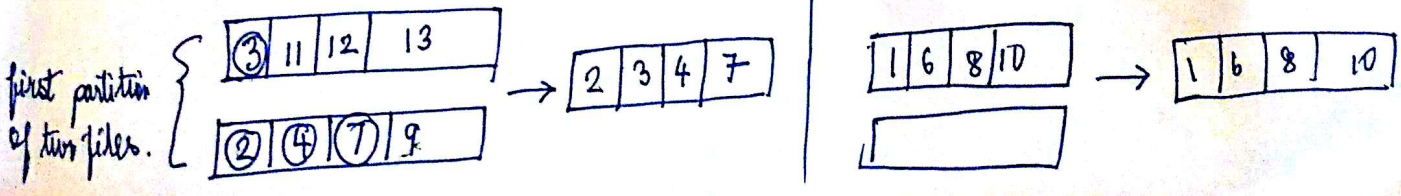


Write in different file

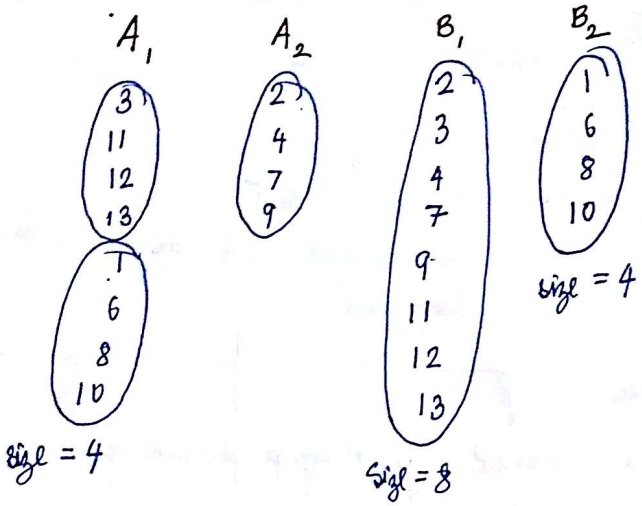


f1

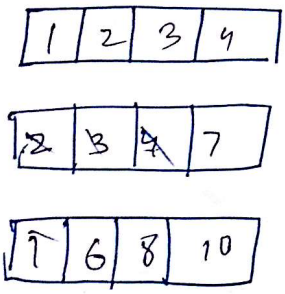
2
3
4
7
9
11
12
13
1
6
8
10



Second Logic. (2-way balanced merge sort) (2 files used)



buffer



---

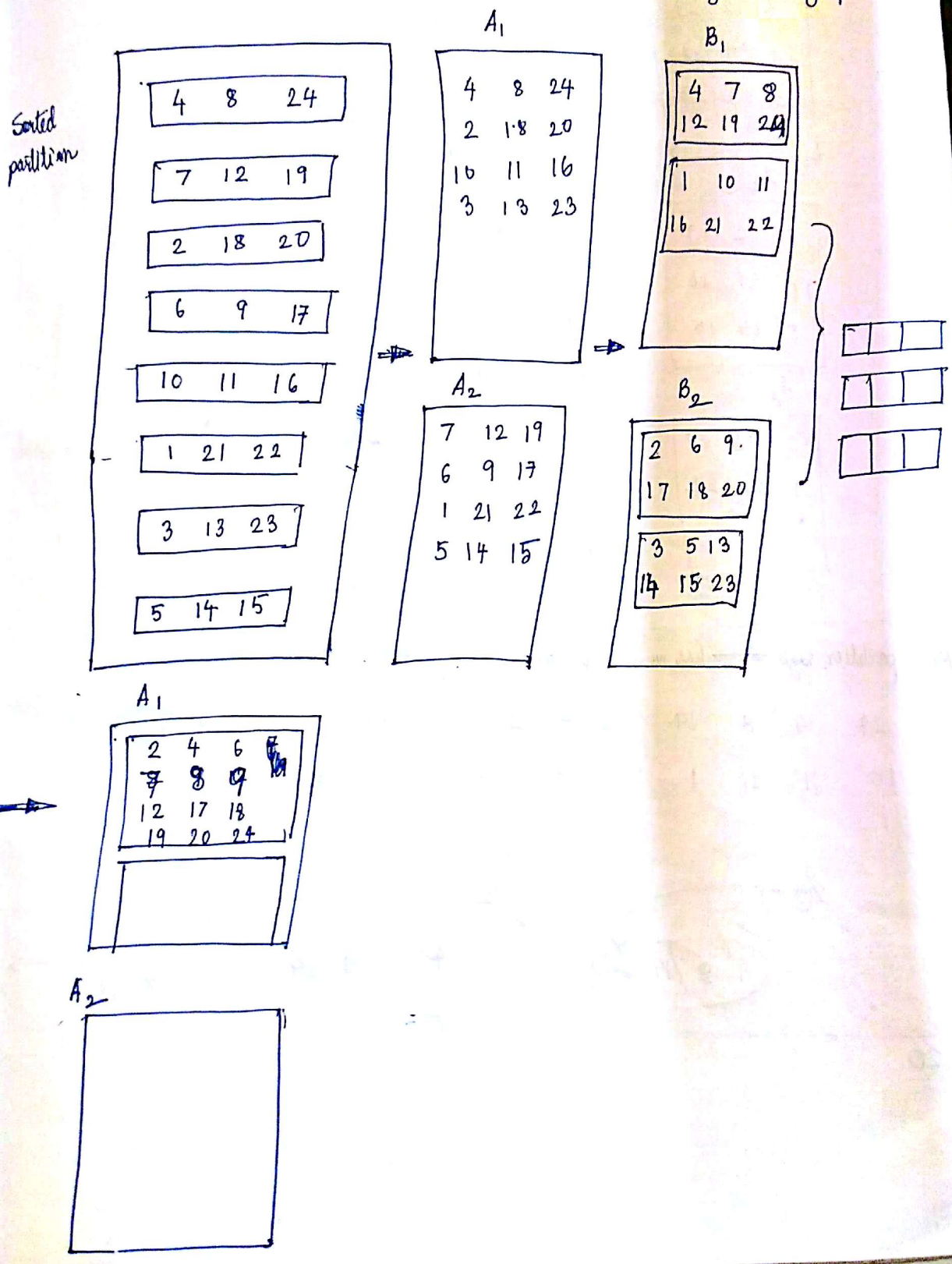
m-way balanced merge sort - (2m+1) files.

24
16

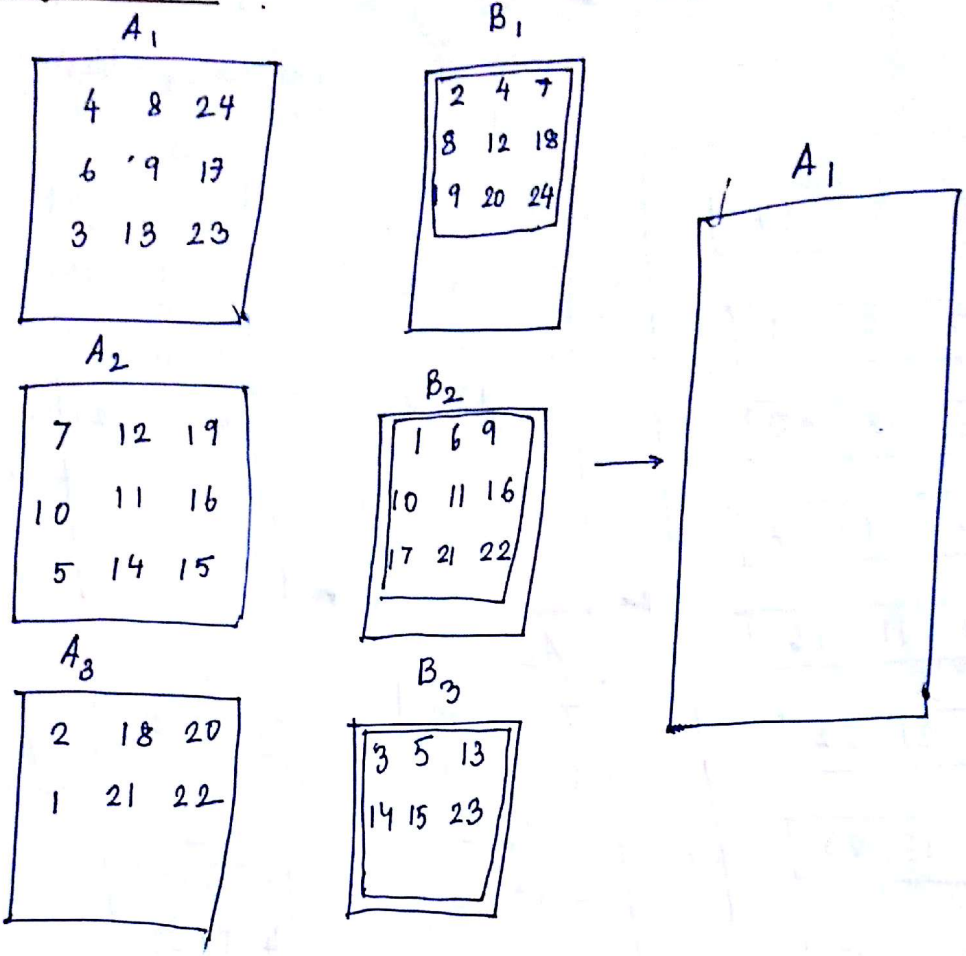
Sorted partition

24 4 8 19 12 7 2 20 18 6 17 9
16 10 11 21 22 1 23 3 13 15 14 5

Partition size = blocking factor



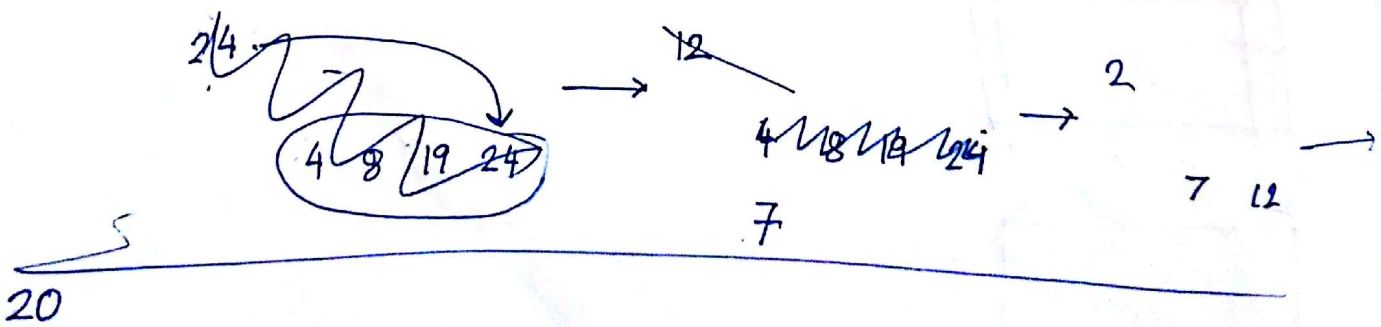
3-way balanced



More partition size ⟶ less number of merges.

24 4 8 19 12 7 2 20 18 6 17 9 16

10 11 21 1 23 3 13 15 14 5



Three variables use us.

I
J
K

Sorted partition generate :

I 24
J 4 8 19
K 8 4 8 12

4 8 12 19 24

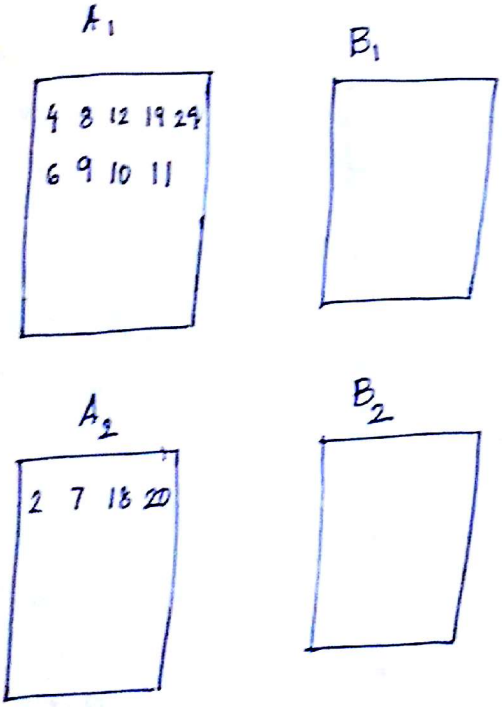
I 26
J 7
K 2

2 7 18 20

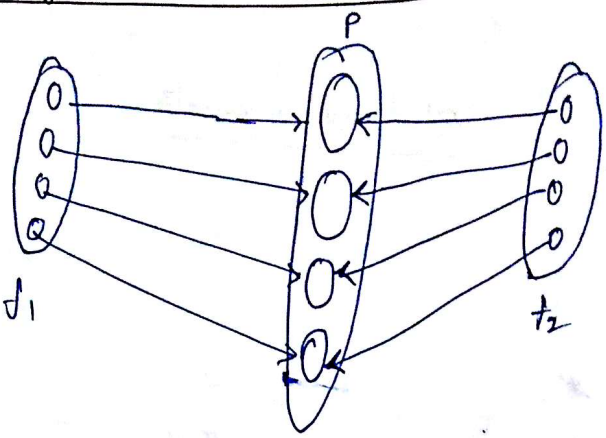
I 7
J 9 16
K 8 9 10 11

6 9 10 11

Now merge.



m-way natural balanced merge sort – (compulsorily 3 files).



Again distribute p in f1 and f2 and repeat process until p contains one partiti

---

m-way polyphase merge – (m+1) files required compulsorily.

3-way (17 partitions initially)

	f1	f2	f3	f4
1	7	6	4	0
2	3	2	0	4
3	1	0	2	2
4	0	1	1	1
5	1	0	0	0

Since this case is not for balanced so distribute the partitions such that partition in f1 f2 and f3 are not equal in number.

6-way polyphase merge

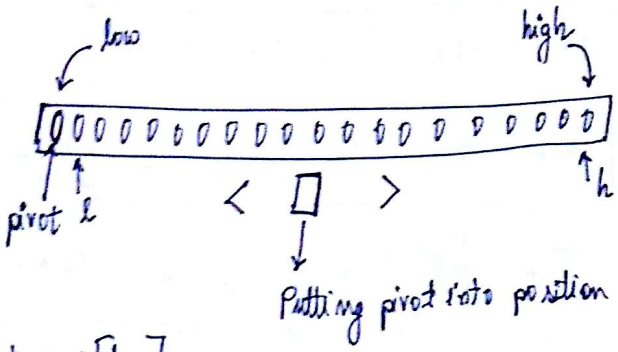
125 partitions of size 1

636

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	π (partitions reqd)	∇ (no. of merges)
	125_1	123_1	119_1	111_1	95_1	63_1	0	636	0
1	62_1	60_1	56_1	48_1	32_1	0_1	63_6	321	$63 \times 6 = 378$
2	30_1	28_1	24_1	16_{16}	0_1	32_{11}	31_6	161	32×11
3	14_1	12_1	8_1	0	16_{21}	16_{11}	15_6	81	336
4	6_1	4_1	0	8_{41}	8_{21}	8_{11}	7_6	1_{41}	328
5	2_1	0	4_{81}	4_{41}	4_{21}	4_{11}	3_6	21	324
6.	0	2_{161}	2_{81}	2_{41}	2_{21}	2_{11}	1_6	11	322
7.	1_{321}	1_{161}	1_{81}	1_{41}	1_{21}	1_{11}	0	6	321
8.	0	0	0	0	0	0	1_{636}	1	636

Quick Sort

Time Complexity = $O(n \log n)$.



pivot = a[low]

if a[i] < pivot → i = i+1;

if a[h] > pivot → h = h-1;

```
void quicksort (int A[], int low, int high)
{
    if (low < high)
    {
        int j = partition(A, low, high);
        quicksort (A, low, j-1);
        quicksort (A, j+1, high);
    }
}

int partition (int A[], int low, int high)
{
    int j, i, h, p;
    i = low + 1; h = high;
    p = A[low];
    while (i < h)
    {
        while(A[i] < p) i++;
        while (A[h] > p) h--;
        if (h > i) swap (A[i], A[h]);
    }
```

if (low < h) swap(A[low], A[h]);

return h;

}

62 21 14 97 87 22 74 85 76 45 84 78

P l l' h h

45 22 87 97

Shell Sort

k=5

	1	2	3	4	5	6	7	8	9	10
	77	62	14	9	30	21	80	25	70	55
	21	62	14	9	30	77	80	25	70	55
k=2	14	9	21	62	30	25	70	77 55	80	77
k=1	9	14	21	30	62 25	62	80 55	70	77	80

Shell sort is called diminishing increament sort.

Take initial (i) $k = n/2$.

(ii) or take prime numbers

(iii) or take decreasing Fibonacci number

```
void createtree(Tptr &T, int d)
{
    cout << "Enter value" << endl; cin >> d
    T = createnode(d);
    cout << "Lc ?" << endl; cin >> ch;
    if (ch == 1) createtree(T->lc);
    cout << "Rc ?" << endl; cin >> ch;
    if (ch == 1) createtree(T->rc);
}
```