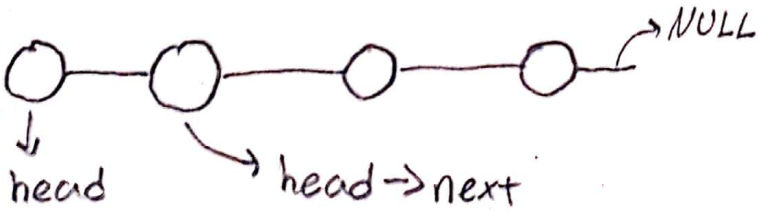
Algorithm in $O(n^2)$ finds primes less than N

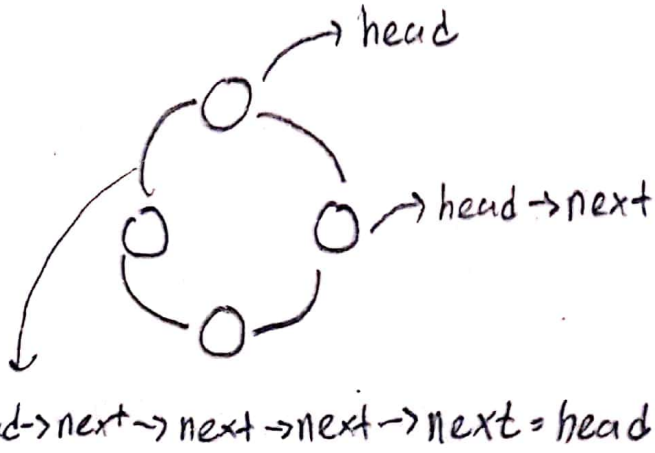
```
for (a[1]=0, i=2; i<=N; i++){
        a[i] = 1;
}
for(i=2; i<=N/2; i++){
   for(j=2; j<=N/i; j++){
      a[i*j] = 0;
   }
}
for(i=1; i<=N; i++){
   if(a[i]) printf("%d", i);
}
```

The sieve of Eratosthenes

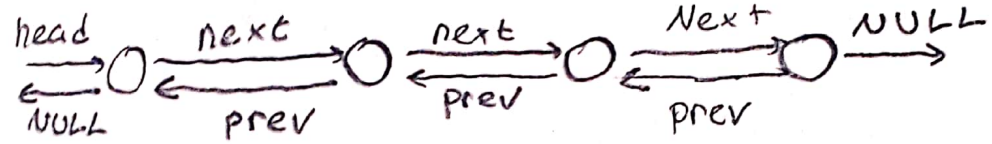
Simple Linked List



Circular List



Doubly Linked List



-Josephus Problem

N people have decided to kill mth person around circle. Closing ranks as each peson drops out of the circle. The problem is to find out the person which is the last to die.

Example N=9, m=5 → 5 1 7 4 3 6 9 2 8.

Single Circular Linked List Code:

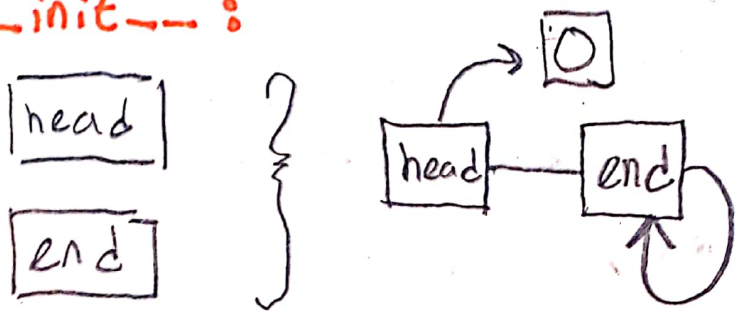
```
void allocate(NODE* head, int n){
    NODE* current = head;
    current->val=1;
    current->next= NULL;
    for(int i=2; i<n; i++){
        while(current->next != NULL){
            current = current->next;
        }
        current->next=(NODE*)malloc(sizeof(NODE));
        current->next->val = i;
        current->next->next = NULL;
    }
    current->next->next=(NODE*)malloc(sizeof(NODE));
    current->next->next=head;
}
```

Stack

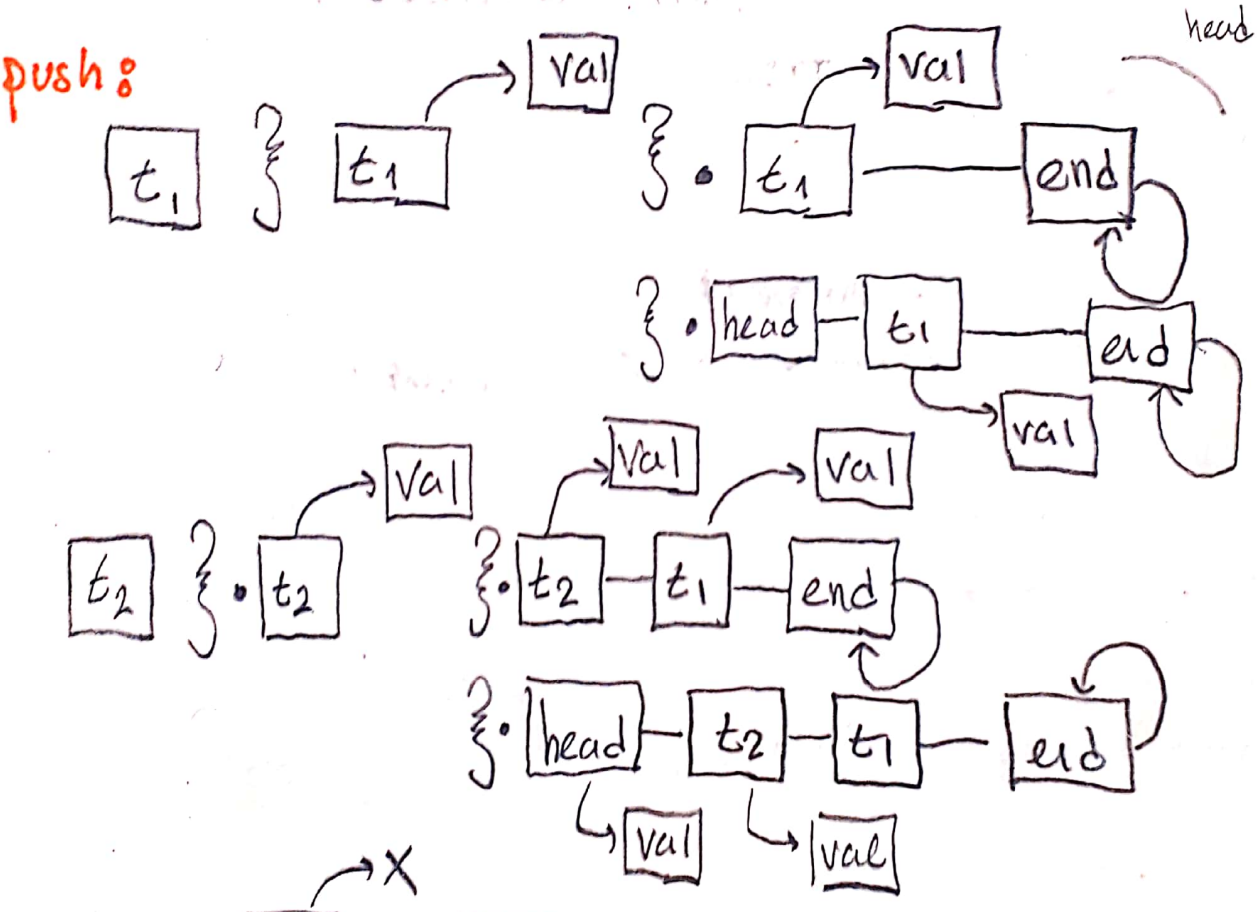
```
Stackinit (NODE* head, NODE* end) {
    head->next = end;
    head->val = 0;
    end->next=end;
}

Void push (NODE* head, int n){
    NODE*currut=(NODE*)malloc(sizeof(NODE));
    Currut->val=num;
    Currut->next=head->next;
    head->next = currut

Int pop(NODE* head) {
    int pop;
    NODE* currut = (NODE*) malloc (sizeof(NODE));
    Currut= head->next;
    head->next= Currut->next;
    pop= Currut->val;
    free (currut);
    return pop;
}
```

Visualization of Stack operations

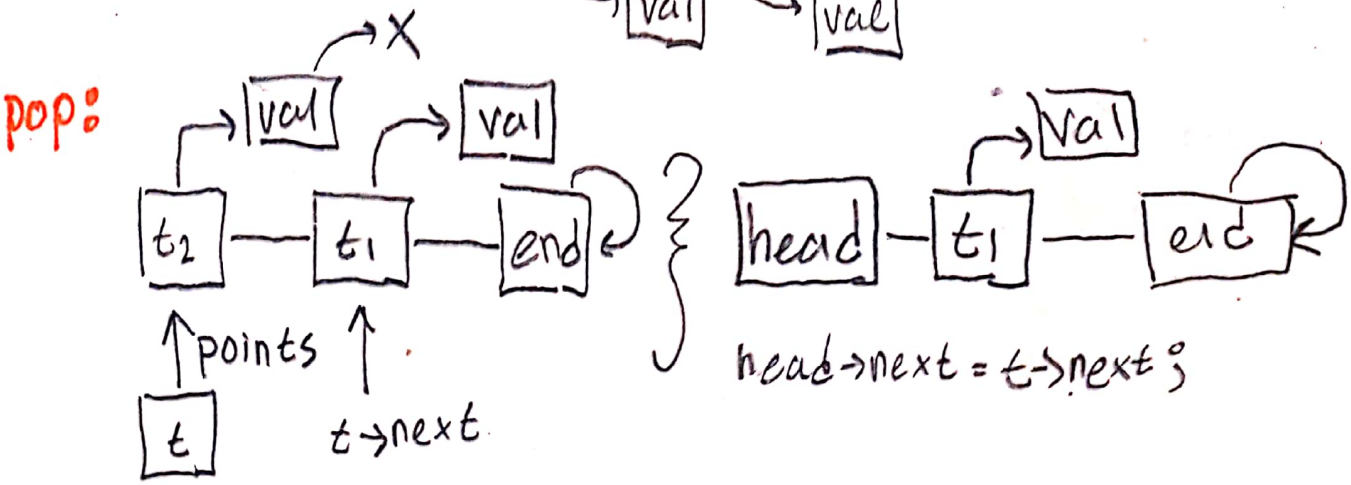
__init__ :



push:



pop:



Another Approach For Stack

```
#define MAX 100

typedef struct{
    int item[MAX];
    int top;
}STACK;

void init(STACK* s){
    s->top=0;
}

int isEmpty(STACK* s){
    if(s->top==0){
        return 1;
    }
    else{
        return 0;
    }
}

int isFull(STACK* s){
    if(s->top==MAX)
        return 1;
    else
        return 0;
}
```

LIFO

```
int pop(STACK* s, int* x){
    if(isEmpty(s))
        return -1;
    else{
        --s->top;
        *x=s->item[s->top]
        return 1;
    }
}

void push(STACK* s, int x){
    if(isFull(s))
        printf("Stack is full");
    else{
        s->item[s->top]=x;
        s->top++;
        printf("done");
    }
}

int peak(STACK* s, int x){
    if(isEmpty(s))
        return 0;
    else{
        adr=s->top-1;
        *x=s->item[adr];
        return 1;
    }
}
```

Doubly Linked List Functions

```
Void push (NODE** head, int val){
    NODE* newnode= (NODE*)malloc(sizeof(NODE));
    newnode->val= val;
    if(*head==NULL){
      newnode->next=NULL;
      newnode->prev=NULL;
      (*head)=newnode;
    }
    else{
      newnode->next = (*head)
      newnode->prev = NULL;
      (*head)->prev = newnode;
      (*head)=head;
    }
}
Void insert(NODE** head, int val){
    NODE* current= *head;
    NODE* newnode= (NODE*)malloc(sizeof(NODE));
    newnode->val= val;
    while (current->next != NULL){
        current=current->next;
    }
    newnode->next=NULL;
    newnode->prev= current;
    current->next= newnode;
}
```

Doubly Linked List Functions 2

```
void del(NODE** head, int position){
    NODE* current = *head;
    if(*head==NULL){
        exit(0);
    }
    else{
        while(current->next !=NULL && position !=0){
            current=current->next;
            position--;
        }
        if(current->next==NULL){
            current->prev->next=NULL;
            free(current);
        else{
            current->prev->next = current->next;
            current->next->prev = current->prev;
        }
    }
}
```

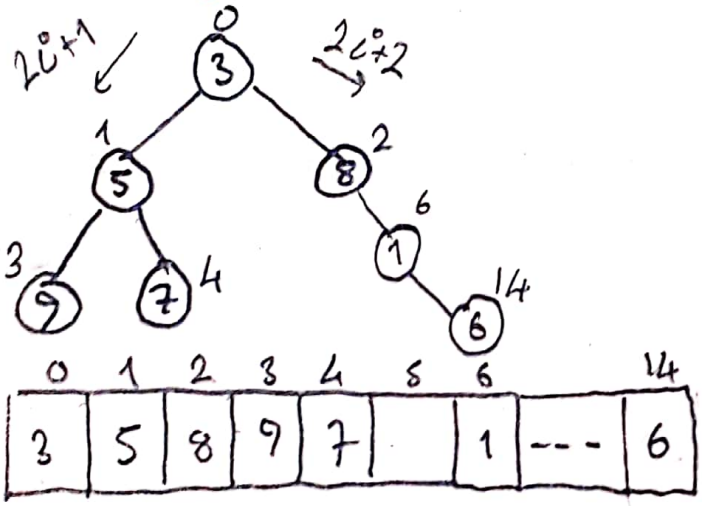
FIFO

Queue

```
void enqueue(Q**q, int val){
    Q* new= (Q*)malloc(sizeof(Q));
    new->val=val;
    if((*q)==NULL){
       new->next=NULL;
       *q=new;
    }
    else{
       new->next = *q;
       *q=newnode;
    }
}
void dequeue(Q**q, int* x){
     if((*q)==NULL){
       printf("Empty");
     }
     else{
        Q* currut= *q;
        Q* before= currut;
        while(currut->next != NULL){
            before=currut;
            currut=currut->next;
        }
        *x=currut->val;
        before->next=NULL;
        free(currut);
    }
}
```

Binary Trees

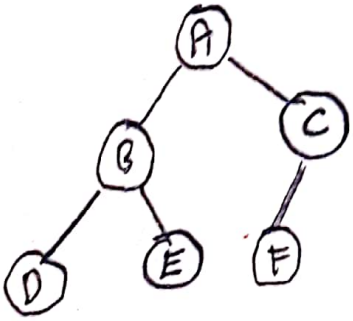
Array



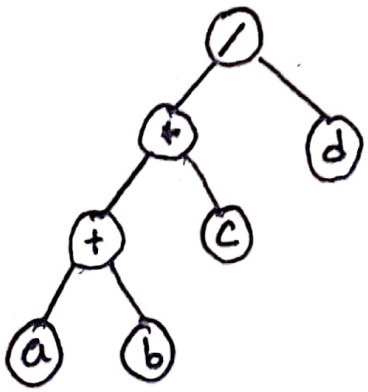
LinKed List

```
typedef struct NODE{
    int val;
    struct NODE* left, *right;
}TREE;
```

Tree Traversal



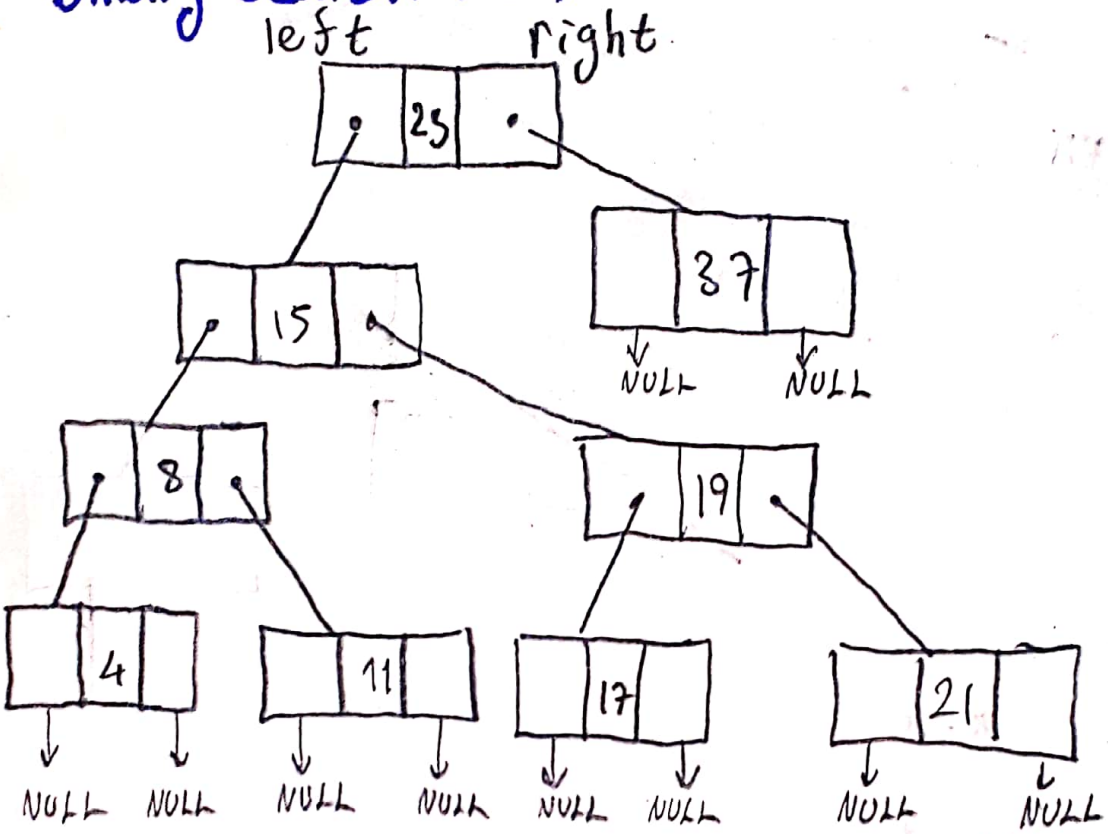
Preorder ⟶ ABDECF
Inorder ⟶ DBEAFC
Postorder ⟶ DEBFCA

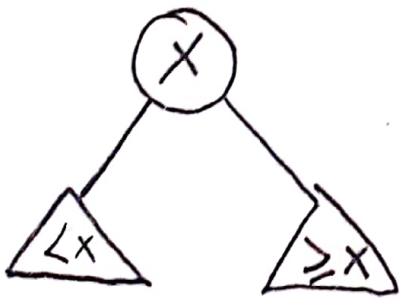


Preorder ⟶ /*+abcd
Inorder ⟶ a+b*c/d
Postorder ⟶ ab+c*d/

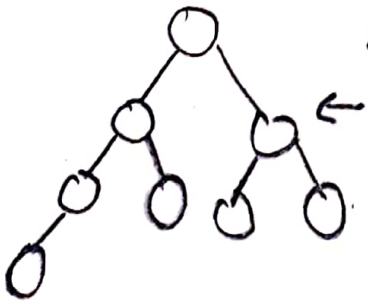
```
void printPreorder(TREE* root){
        if(root==NULL){
            return;
        }
        else{
            printf("%d\n", root->val);
            printPreorder(root->left);
            printPreorder(root->right);
        }
}
void printInorder(TREE* root){
    if(root==NULL){
        return;
    }
    else{
        printInorder(root->left);
        printf("%d\n", root->val);
        printInorder(root->right);
    }
}
void printPostorder(TREE* root){
    if(root==NULL)
        return;
    else{
        printPostorder(root->left);
        printPostorder(root->right);
        printf("%d\n", root->val);
    }
}
```

Binary Search Trees



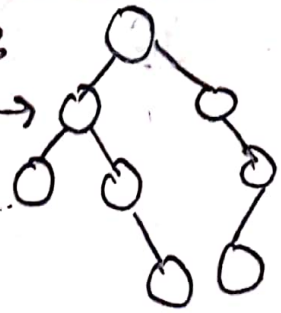


Best Case



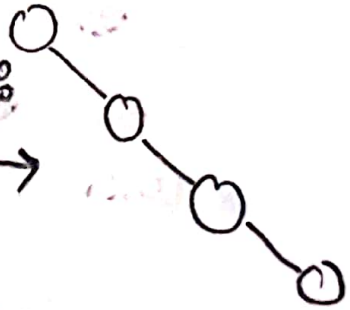
Av. Case

Search: $\leftarrow \log_2 N \rightarrow$

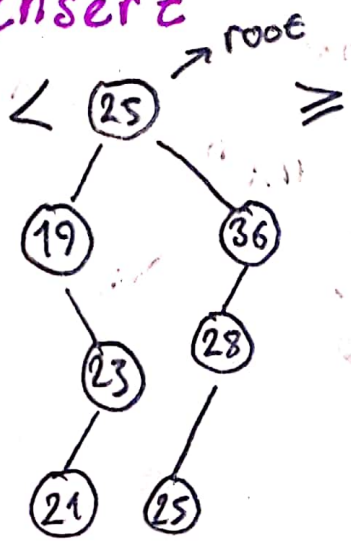


Worst Case

Search: $N \rightarrow$

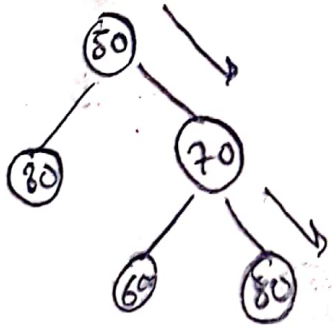


① insert



average : $\log_2 N$
worst : n

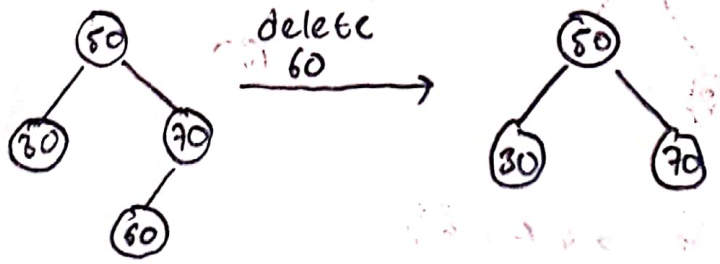
② Search



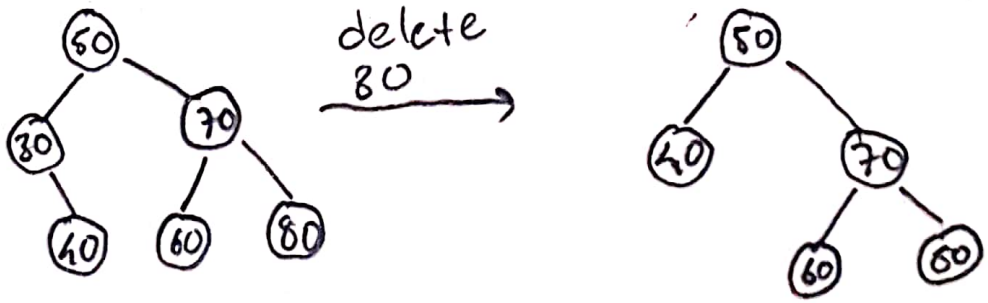
X=80

③ Delete

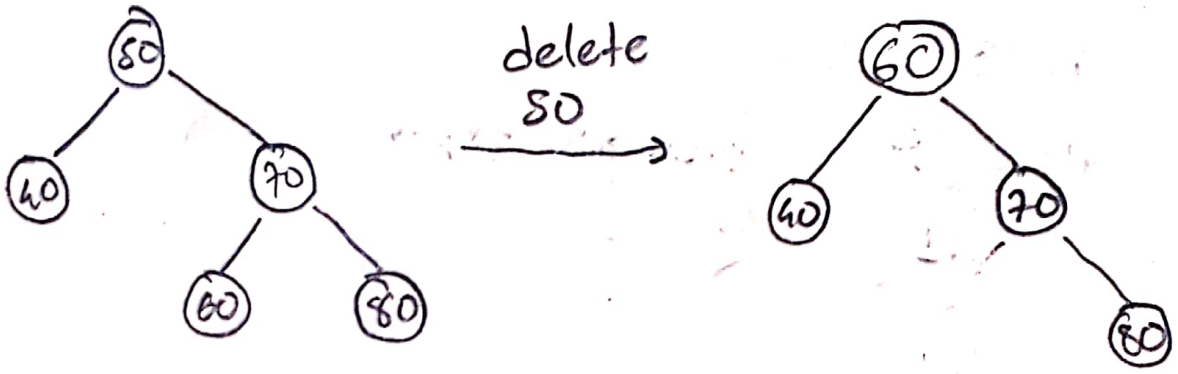
- Node to be deleted is Leaf



- Node to be deleted has one child



- Node to be deleted has two children



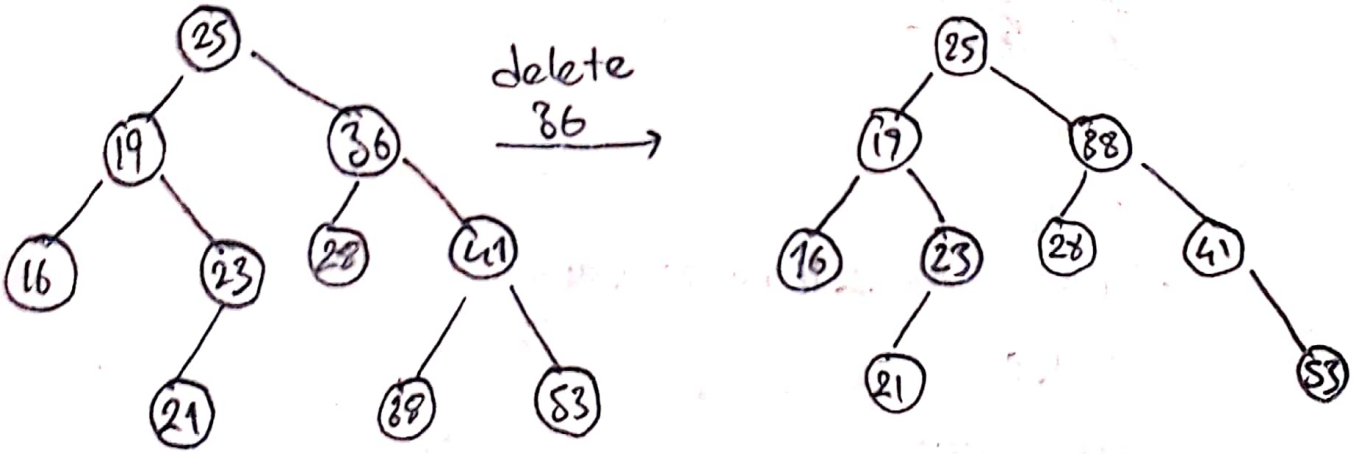
inorder:

40 50 60 70 80

a. find min element at the right subtree (inorder successor)

b. copy it to that node

c. delete inorder successor.



16 19 21 23 25 28 ~~36~~ 38 41 83

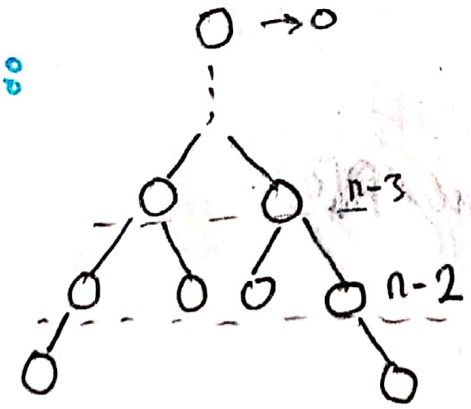
```
NODE* search (NODE* root, int x){
        NODE* tmp = root;
        while (tmp != NULL && tmp->val != x){
            if (x < tmp->val){
                tmp = tmp->left;
            }
            else{
                tmp = tmp->right;
            }
        }
        return tmp;
}

NODE* search(NODE* root, int val){
        if (root == NULL || root->val)
            return root;
        if(val < root->val)
            return search(root->left, val);
        else
            return search(root->right, val);
}
```

```
NODE* insert(NODE* root, int value){
    if (root==NULL){
        NODE* new=(NODE*)malloc(sizeof(NODE));
        new->val=value;
        new->left=NULL;
        new->right=NULL;
        return new;
    }
    if(value<root->val){
        root->left=insert(root->left, value);
    }
    else{
        if(value>root->val){
            root->right=insert(root->right, value);
        }
    }
    return root;
}
```

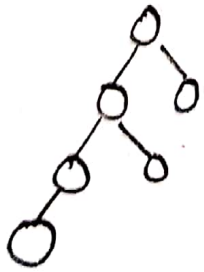
Heap Tree

- Heap tree is a complete binary tree: balanced, left justified.
  - A binary tree of depth n is balanced if: all the nodes at depths 0 through n-2 have two children
  - A balanced binary tree of depth n is left-justified if: all the leaves are the same depth, or all the leaves at depth n are to the left of all the nodes at depth n-1.

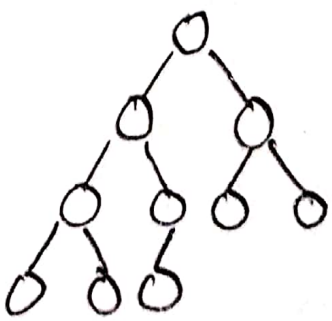
Balanced:



Left Justified:

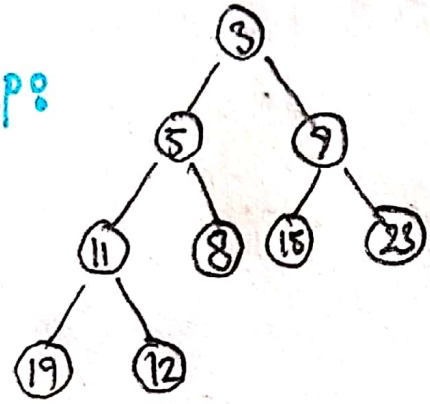


Balanced + Left Justified:



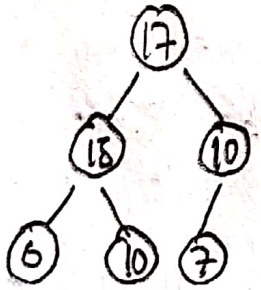
Max-Heap and Min-Heap

Min-Heap:



tree[parent] <= tree[child]

Max-Heap:



tree[parent] >= tree[child]

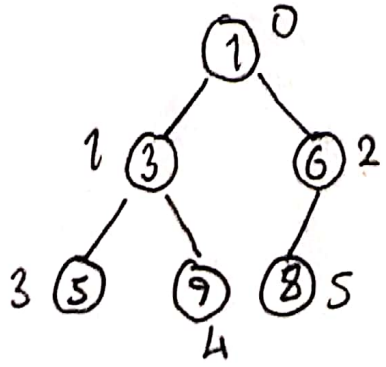
Array Representation of Heap Tree

- The root element will be at A[0].
- For the ith node A[i]:

  A[$(i-1)/2$] ⟶ parent node

  A[$2*i+1$] ⟶ the left child node

  A[$2*i+2$] ⟶ the right child node



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

Applications

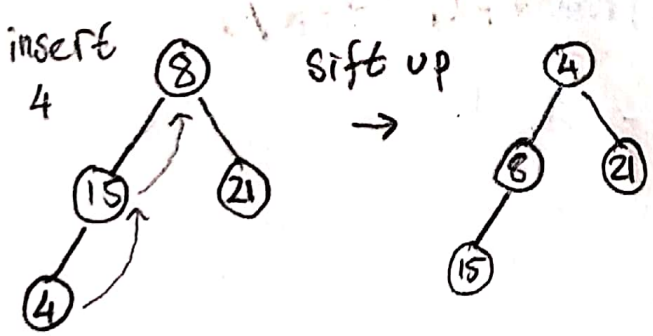
- Sorting an array with $O(n \log n)$
- Implementing priority queue.
- Finding Kth largest in array.
- Meging K sorted arrays.

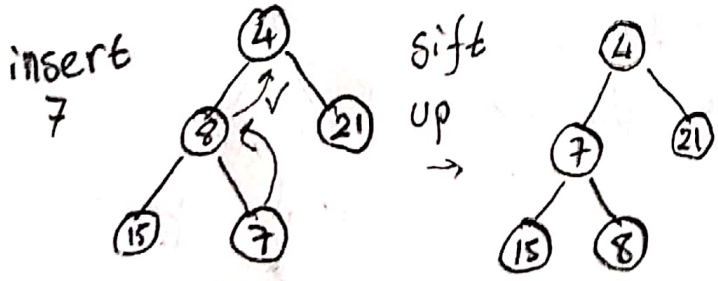
Building a Heap Tree

Insert

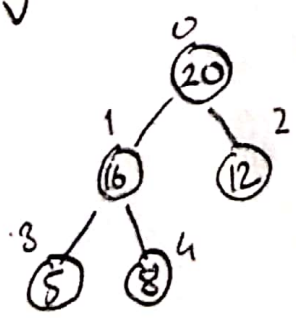
- put the item in the next available node
- sift up the new item until:

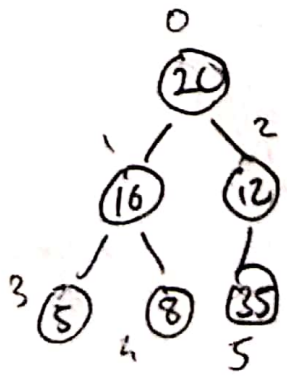
  New item >= its parent, if min-heap

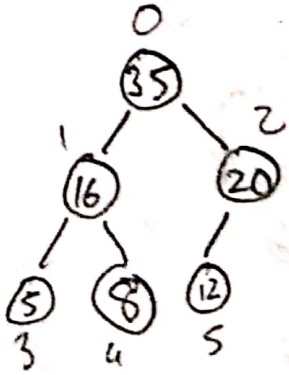


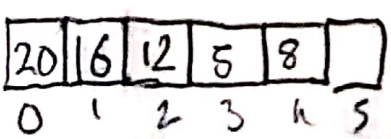


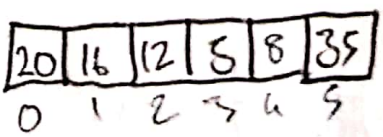
```
Void insert(int item, int heap[], int *n){
    heap[*n] = item;
    siftUp(item, heap, *n);
    (*n)++;
}

Void siftUp (int item, int heap[], int adr){
    int stop=0;
    while(adr>0 && !stop){
        parent=(adr-1)/2
        if(heap[parent] < item){
            swap(&heap[adr], &heap[parent]);
            adr=parent;
        }
        else{
            stop=1;
        }
    }
}
```

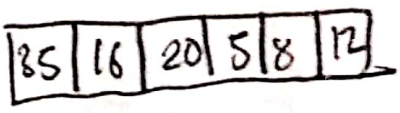








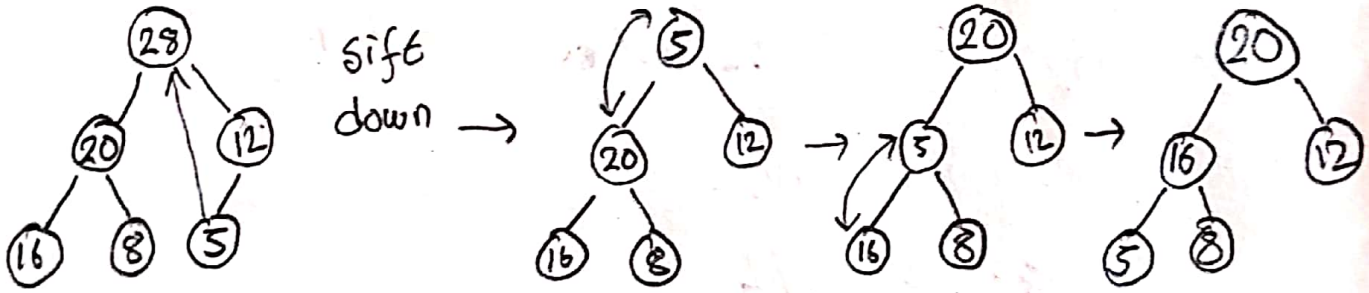




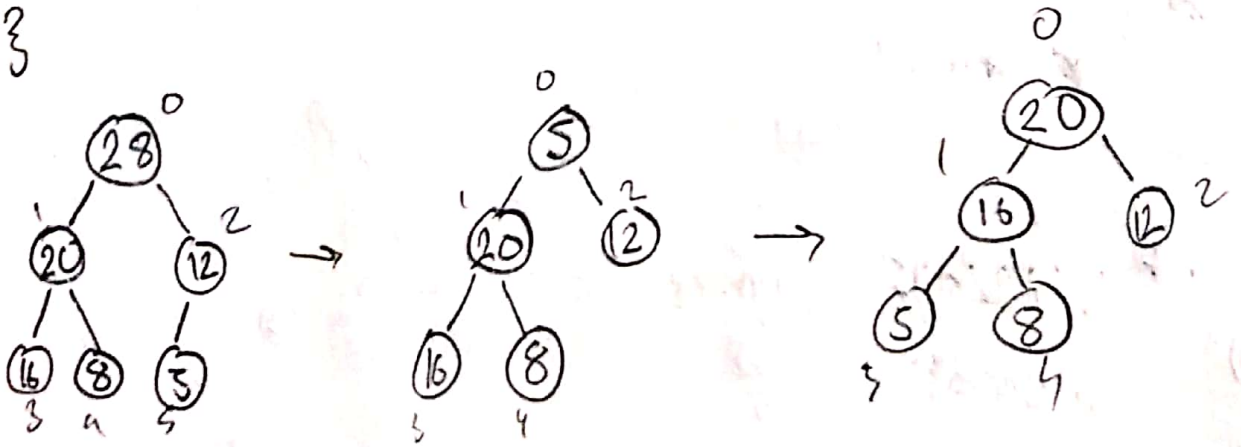
Remove the Root

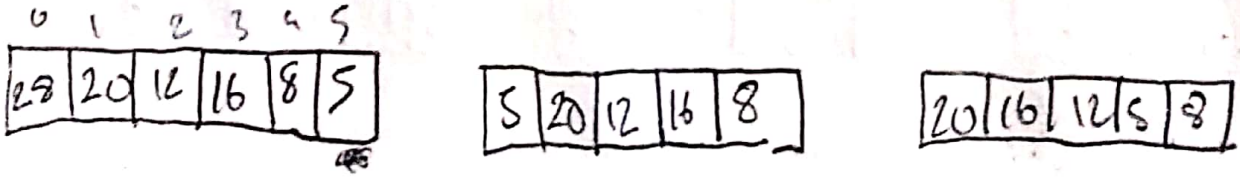
- make a copy of root node
- move the last item in the heap to the root.
- sift down the new root item until

  it is >= its children
- return the copy of old root node.




```
int remove (int heap[], int *n){
    int retVal= heap[0]
    heap[0]= heap[*(n-1)];
    (*n)--;
    SiftDown(heap, *n);
    return retval;
}
```





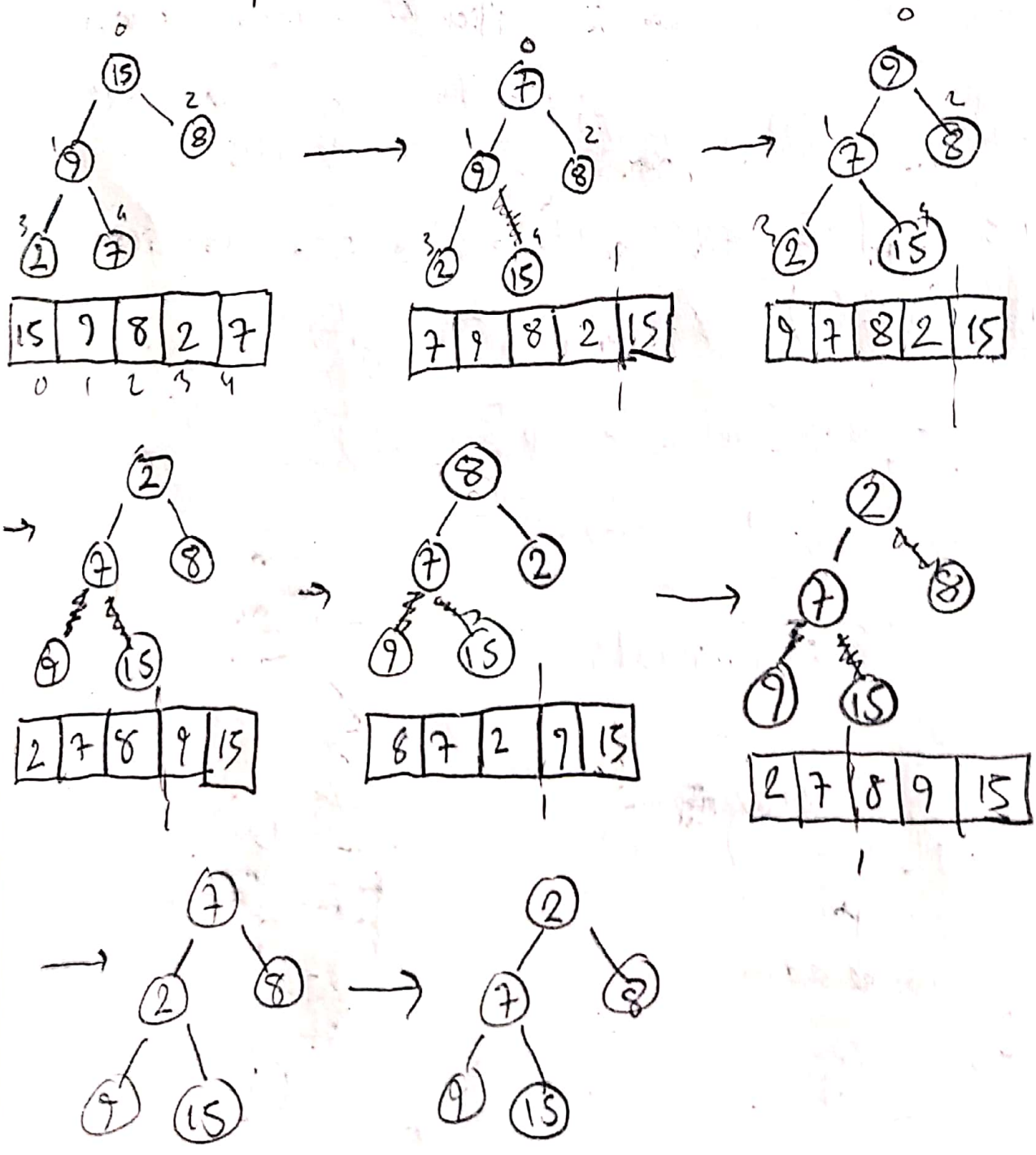
```
void siftDown(int heap[], int n){
    int parent=0, child;
    child = findSmallestChild(heap, parent, n);
    while(child!=-1 && heap[child]>heap[parent]){
        swap(&heap[child], &heap[parent]);
        parent= child;
        child = findSmallestChild(heap, parent, n);
    }
}
int findSmallestChild(int heap[], int parent, int n){
    int child1 =2*parent+1, child2 = 2*parent+2;
    if(child2<n){ // parent has two children
        if(heap[child2]>heap[child1]){
            return child2;
        }
        else{
            return child1;
        }
    }
    else if(child1<n){ //parent has one child.
        return child1;
    }
    else{
        return -1;
    }
}
```

Complexity of Heap

- A heap containing n items has a height $\leq \log_2 n$
- Thus, removal and insertion are both $O(\log n)$

Heap Sort

- Heap sort repeatedly finds the largest remaining element and puts it in place.
- Heapsort is $O(n \log n)$

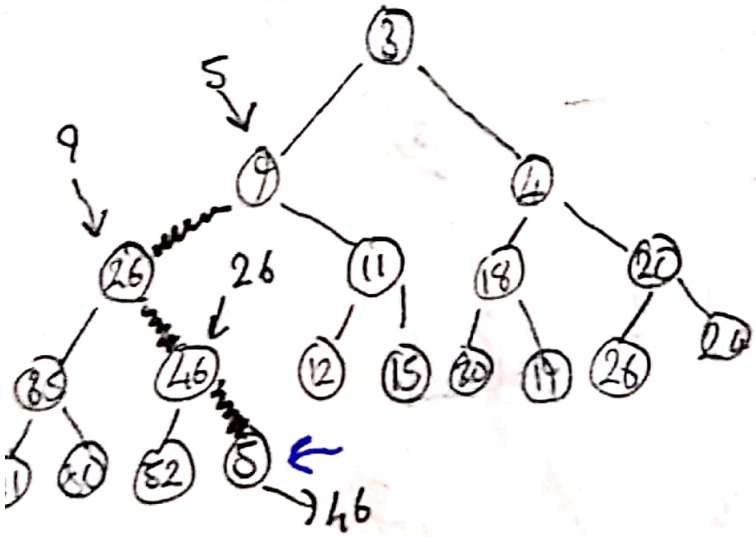


```
Void heapSort(int heap[], int n){
    for(i=n-1; i>0; i--){
        swap(&heap[0], &heap[i]);
        siftdown(heap, i);
    }
}
```

Priority Queue

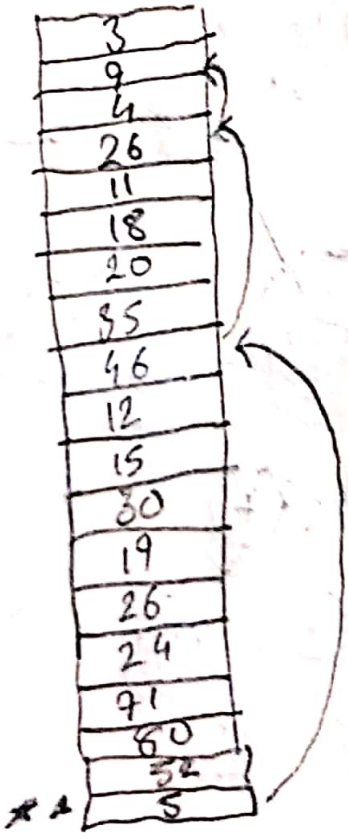
- A priority queue is similiar to a regular queue, but each item in queue has an additional piece of information: Priority.
- A priority queue supports two operations: insert, and remove the item with the highest priority.

Heap Based Priority Queue- Insert



Insert 5 at the end of tree

heapify the tree



Huffman Tree

Example Text: aba ab cabbb

- Count the frequeaes of the characters

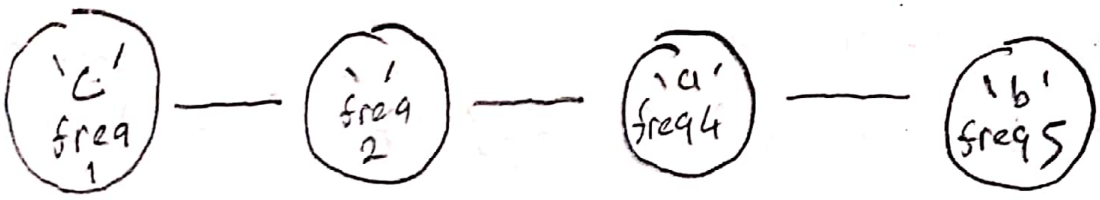
  ' ' → 2

  'a' → 4

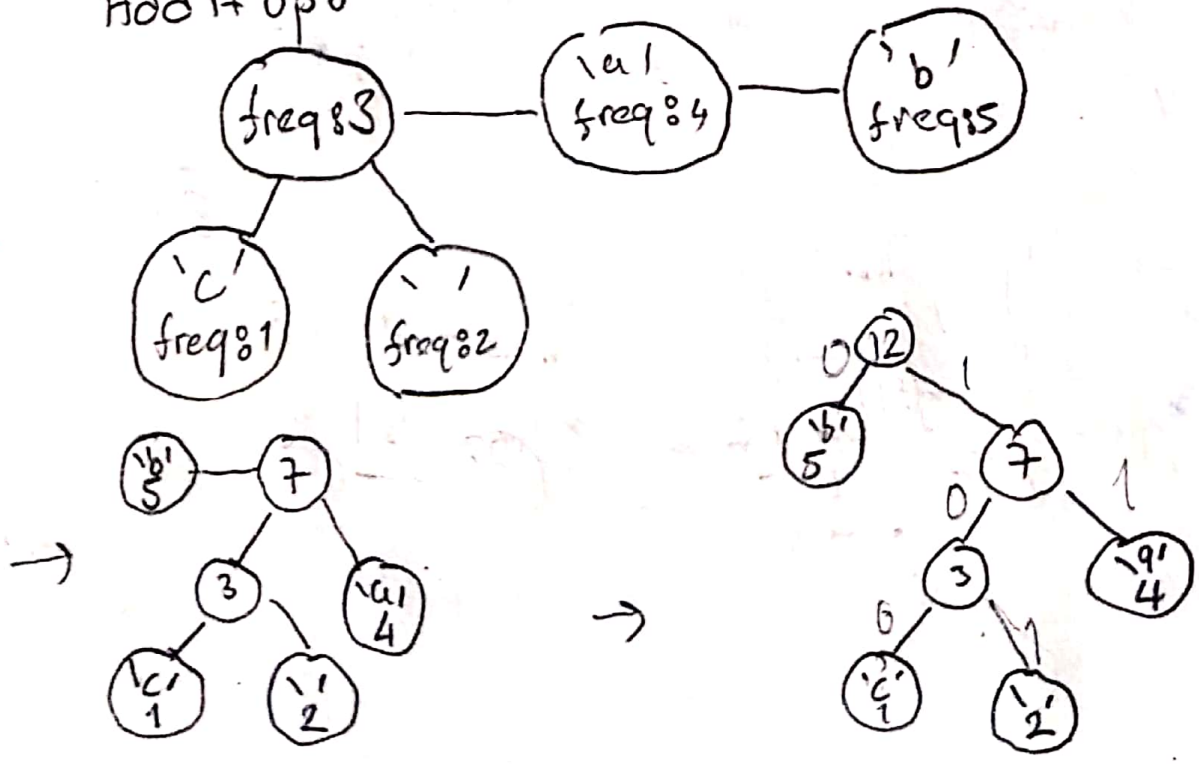
  'b' → 5

  'c' → 1

- Create a leaf node for each symbol and add it to th priority queue.



- Remove two nodes of highest priority
  Create new interal node with these as children and freq is addition of two.
  Add it up.



Divide and Conquer

Merging Two Sorted array

```
void merge(int A[], int B[], int C[], int nA, int nB){
    int iA=0, iB=0, iC=0;
    while((iA<nA)&&(iB<nB)){
        if(A[iA]<B[iB]){
            C[iC++]=A[iA++];
        }else{
            C[iC++]=B[iB++];
        }
    }
    while(iA<nA) C[iC++]=A[iA++];
    while(iB<nB) C[iC++]=B[iB++];
}
```

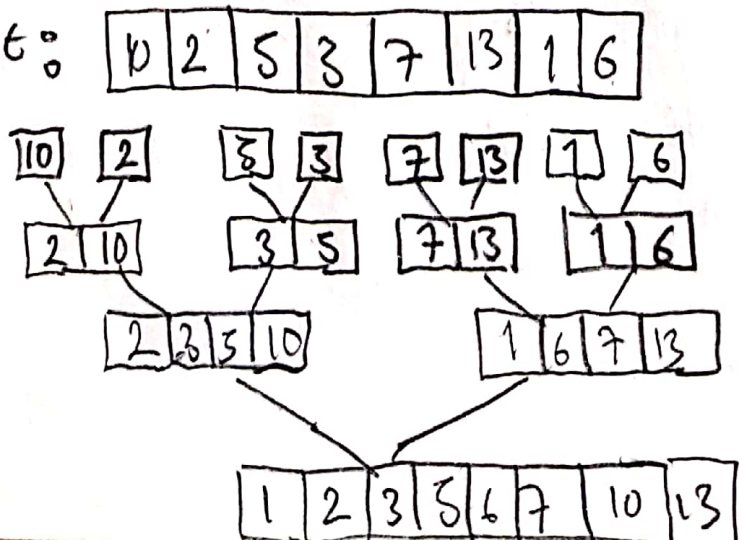
O(n)

Merge Sort

```
void mergeSort(int A[], int p, int r){
                    ↳ Sort subarray A[p...r]
    int q;
    if(p<r){
        q=(p+r)/2
        mergeSort(A,p,q); // sort A[p...q]
        mergeSort(A,q+1,r); //sort A[q+1...r]
        merge(A,p,q,r);
    }
}
```

```
Void merge (int A[], int p, int q, int r){
    int i, j, k, nB, *B;
    nB = r-p+1
    B = (int*)malloc(nB*sizeof(int));
    i = p;
    j = q+1;
    k = 0;
    while((i<=q) && (j<=r)){
        if(A[i]<A[j]){
            B[k++] = A[i++];
        }
        else{
            B[k++] = A[j++];
        }
    }
    while(i<=q) B[k++] = A[i++];
    while(j<=q) B[k++] = A[j++];
    copy(A, B);
    free(B);
}
```

input:



A[0--7]

mergeSort(A,0,7)

mergeSort(A,0,3)

mergeSort(A,0,1)

mergeSort(A,0,0)

mergeSort(A,1,1)

merge(A,0,0,1)

mergeSort(A,2,3)

mergeSort(A,2,2)

mergeSort(A,3,3)

merge(A,2,3)

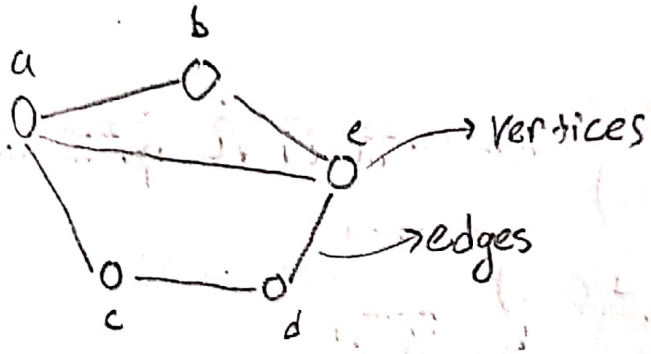
merge(A,0,1,3)

mergesort(A,4,7)

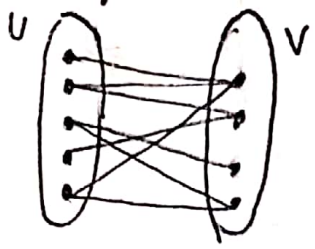
merge(A,0,3,7)

Graphs

$G(V, E)$

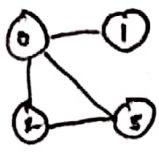


- When there is an edge connecting two vertices, we say that the vertices are adjacent to one another. The edge is incident to both vertices.
- Degree of vertex a is 3.
- Undirected graph is a tree, does not contain cycle.
- Bipartite graph contains two set of vertices U, V. The vertices of U join only with the vertices of V. The vertices of same set do not join.



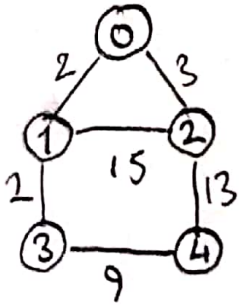
Undirected Graph - Adjacy Matrix

For each edge v-w in graph $adj[w][v] = adj[v][w]$



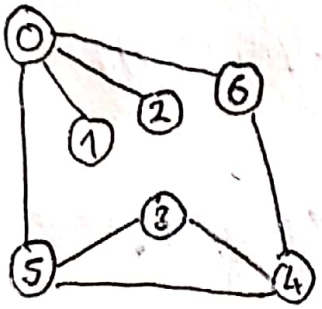
	0	1	2	3
0	0	1	1	1
1	1	0	0	0
2	1	0	0	1
3	1	0	1	0

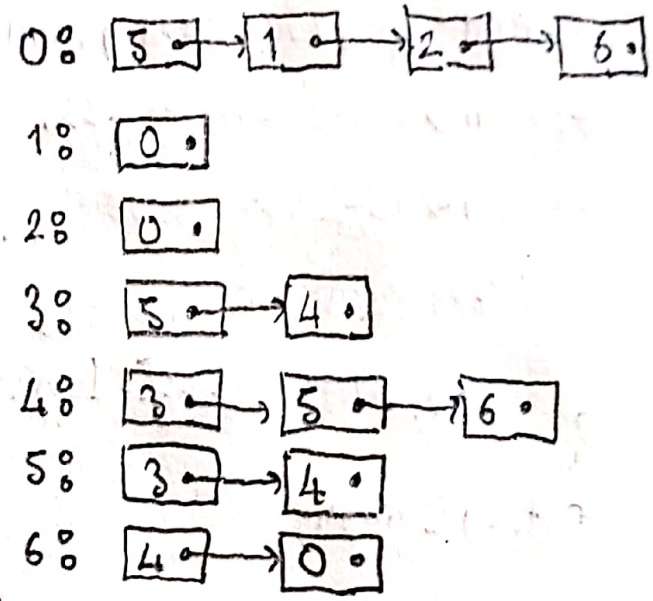
Undirected Weighted Graph - Adjacy Matrix



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Undirected Graph - Adjacy List






```
struct node {
    int vertex;
    struct node* next;
};
struct graph {
    int nVertex;
    struct node** adjList;
};
typedef struct node NODE;
typedef struct graph GRAPH;
NODE* createNode(int vertex){
    NODE* newNode = (NODE*)malloc(sizeof(NODE));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}
```

```
GRAPH* createGraph(int nvertices){
    int i;
    GRAPH* _graph = (GRAPH*) malloc(sizeof(GRAPH));
    _graph->nvertex = nvertices;
    _graph->adjList = (NODE**)(nvertices * sizeof(NODE*));
    for(i=0; i<nvertices; i++){
        _graph->adjList[i] = NULL;
    }
    return _graph;
}

Void addEdge(GRAPH* _graph, int src, int dest){
    NODE* newNode = createNode(dest);
    newNode->next = _graph->adjList[src];
    _graph->adjList[src] = newNode;
    newNode = createNode(src);
    newNode->next = _graph->adjList[dest];
    _graph->adjList[dest] = newNode;
}
```

Minimum Spanning Tree

- Spanning tree is a tree which contains all the vertices of the grapho
- Minimum spanning tree is a spanning tree with the minimum sum of weights.

Prim's Algorithm

Start with vertex 0 and greedily grow tree T. At each step, add cheapest edge that has exactly endpoint in T.

Kruskal Algorithm

Consider edges in ascending order of cost.
Add the next edge to T unless doing so would create a cycle.

```
typedef struct EDGE{
    int u,v;
    int weight;
}EDGE;
```

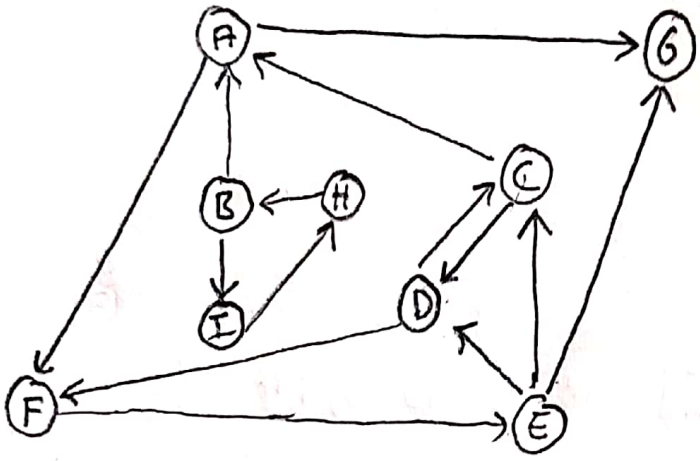
```
void Kruskal(EDGE gr[], EDGE mst[], int nV){
    int labelNo=0, i, j;
    for(i=0; i<nV; i--){
        label[i]=0;
    }
    i=j=0;
    while(i< nV-1; i++){
        uu = gr[j].u;
        vv = gr[j].v;
        if(label[uu]+label[vv] ==0){
            mst[i].u =uu;
            mst[i].v=vv;
            mst[i].weight = gr[j].weight;
            label[uu]=label[vv]=labelNo++;
            i++;
        }
        if(label[uu] != label[vv]){
            mst[i].u=uu;
            mst[i].v=vv;
            mst[i].weight = gr[j].weight;
            i++;
            if( !label[uu]){
                label[uu]= label[vv];
            }
            else if(!label[vv]){
                label[vv]=label[uu];
            }
```

```
    else{
      union(label, nr, uu, vv);
    }
  }
  d++;
}
```

```
Void union(int label[], int nr, int uu, int vv){
    int c;
    for(c=0; c<nr; c++){
        if(label[c] ==uu){
            label[c]=vr;
        }
    }
}
```

Graph Traversal

Depth First Search - DFS



Starting point is A

Visited:

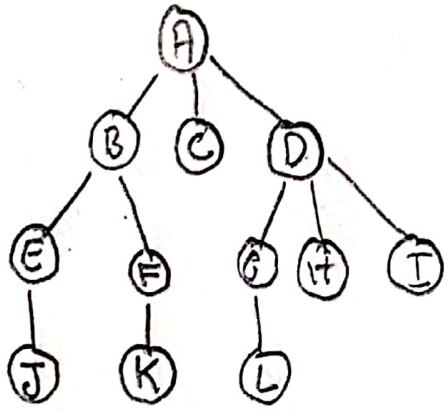
A	B	C	D	E	F	G	H	I
1		1	1	1	1			

AF
FE
EC
CD
D---
.....
EG
G---
. . --
. .
-- .

→ AFECDG

```
DFS(G, s):
    mark s as visited
    for all neighbors w of s in graph G
        if w is not visited:
            DFS(G,w)
```

Breadth First Search - BFS



A
B C D
C D E F
D E F
E F G H I
F G H I J
G H I J K
H I J K L

```
BFS(G,s):
    Q.enqueue(s)
    mark s as visited
    While(Q is not empty):
        V=Q.dequeue(),
        for all neighbors w of v in G:
            if w is not visited:
                Q.enqueue(w)
                mark w as visited
```