Binary trees(bt)

- terms :
  - `level of node` : Root has level 0; level of any other node is one more than its parent
  - `depth of tree` : Maximum level of any leaf in the tree (path length from the deepest leaf to the root)
  - `depth of node` : path length from root to node
  - `Height of a tree` : Path length from the root node to the deepest leaf
  - `Height of a node` : Path length from the node to the deepest leaf
- types :
  - `strictly bt` : every node has either 0/2 children
  - `fully bt` : all leaves at same level.no of nodes in level i = 2^i (this is actually perfect bt)
  - `complete bt` : all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible

BST - Binary search tree

- elements in left subtree of node n are less than contents of node n, elements in right subtree of node n are greater than/equal to contents of node n
- bst using linked list

```
typedef struct node
{
    int data;
    struct node *left;
    struct node *right;
}NODE;
NODE* createnode(int data)
{
    NODE *nn=(NODE*)malloc(sizeof(NODE));
    nn->data=data;
    nn->left=NULL;
    nn->right=NULL;
    return nn;
}
```

- insert operation for bst in ll

```
struct node* insert(struct node* node,int key)
{
    if (node == NULL)
        return createnode(key);
    if (key < node->key)//left child so insert at left subtree
        node->left = insert(node->left, key);
    else if (key > node->key // right child so insert ar right subtree
            node->right = insert(node->right, key);
    return node;
}
```

- delete node operation for bst in ll :
  - assume node to be deleted is t, parent is p
  - case 1 : node t to be deleted has no child(leaf node)
  - if t is right child of p, then do p->right=null, free (t)
  - if t is left child of p, then do p->left=null, free(t)
  - case 2 : node t to be deleted has 1 child
  - if t is right child of p,
    - if t's child is rightchild then do p->right = t->right, free (t)
    - if t's child is leftchild then do p->right = t->left, free (t)
  - if t is left child of p,
    - if t's child is rightchild then do p->left = t->right, free (t)
    - if t's child is leftchild then do p->left = t->left, free (t)
  - case 3 : node t to be deleted has 1 child
    - replace t with inoder successor , then delete inorder successor(using recursion of same function)
- implementation :

```
void searchElement(NODE **t,int data,NODE **parent){
    while(!isempty(*t)&& data!=(*t)->data){
            *parent=*t;
            if(data <= (*t)->data)
                *t=(*t)->left;
            else{
                *t=(*t)->right;
            }
        }
}
NODE* inordersuccessor(NODE *t)
{
    NODE *s;
```

```
    if(t->right!=NULL)
        s=t->right;
    while(s->left!=NULL)
        s=s->left;
    return s;
}
NODE* deletenode(NODE *root,int data){
    NODE *parent=NULL,*t=root;
    if(t->data==data)//Root element
        parent=NULL;
    else
        searchElement(&t,data,&parent);
    if(t==NULL){
       printf("Element not found");
       return root;
    }
    //case 1: Leaf Node
    if(t->left==NULL && t->right==NULL){
          if(parent==NULL)
            root=NULL;
          else{
            if(parent->left==t){//left child
                parent->left=NULL;}
            else if(parent->right==t){//right child
                parent->right=NULL;}
          }
          free(t);
        }
    //Case 2: One child
    else if(t->left==NULL || t->right==NULL){
        if(parent==NULL){//root node with one child
            if(t->left==NULL){//t is right child
                root=root->right;
                t->right=NULL;
                free(t);}
            else{           //t is left child
                   root=root->left;
                   t->left=NULL;
                   free(t);
                }
        }
        else{
            if(parent->left==t){//left child
                if(t->left==NULL){
                    parent->left=t->right;}
                else{
```

```
                    parent->left=t->left;}
                }
            else if(parent->right==t){//right child
                if(t->left==NULL){
                    parent->right=t->right;}
                else{
                    parent->right=t->left;}
                }
            free(t);
            }

        }
        //case 3: two children
        else
            {
            int val;
            NODE *s;
            s=inordersuccessor(t);
            val=s->data;
            root=deletenode(root,s->data);
            t->data=val;
            }
    return root;
}
```

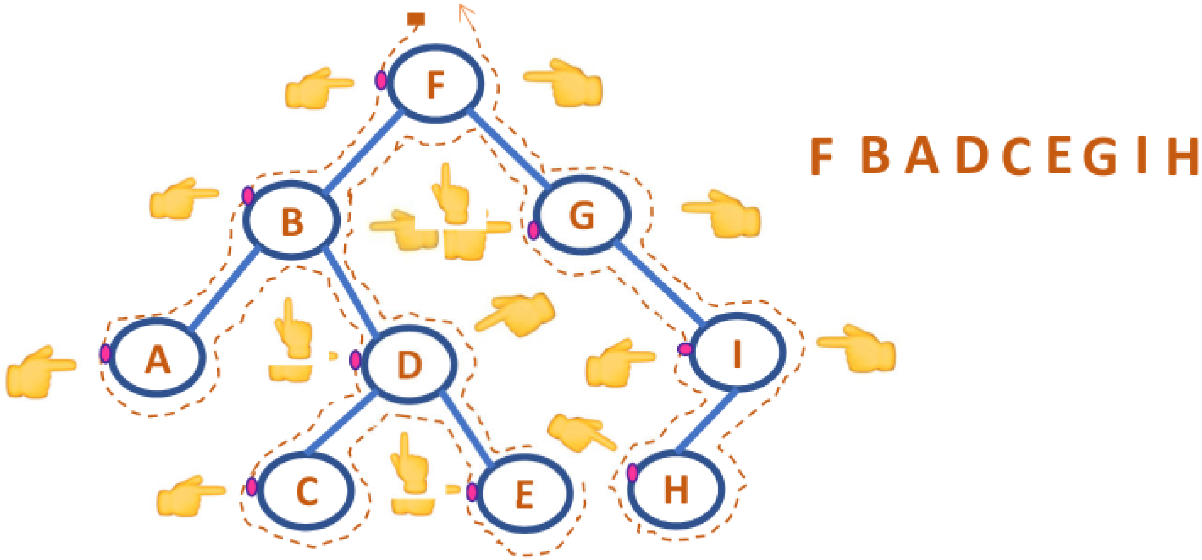
- bst using array :
  - root position i = 0,
  - if i indicates current node then
    - `left child` : 2i+1
    - `right child` : 2i+2
  - each node has its data and another field by name used to contain whether it is a valid node or not
- code :

```
typedef struct tree_array
{
        int info;
        int used;
}NODE;
NODE bst[MAX];
```

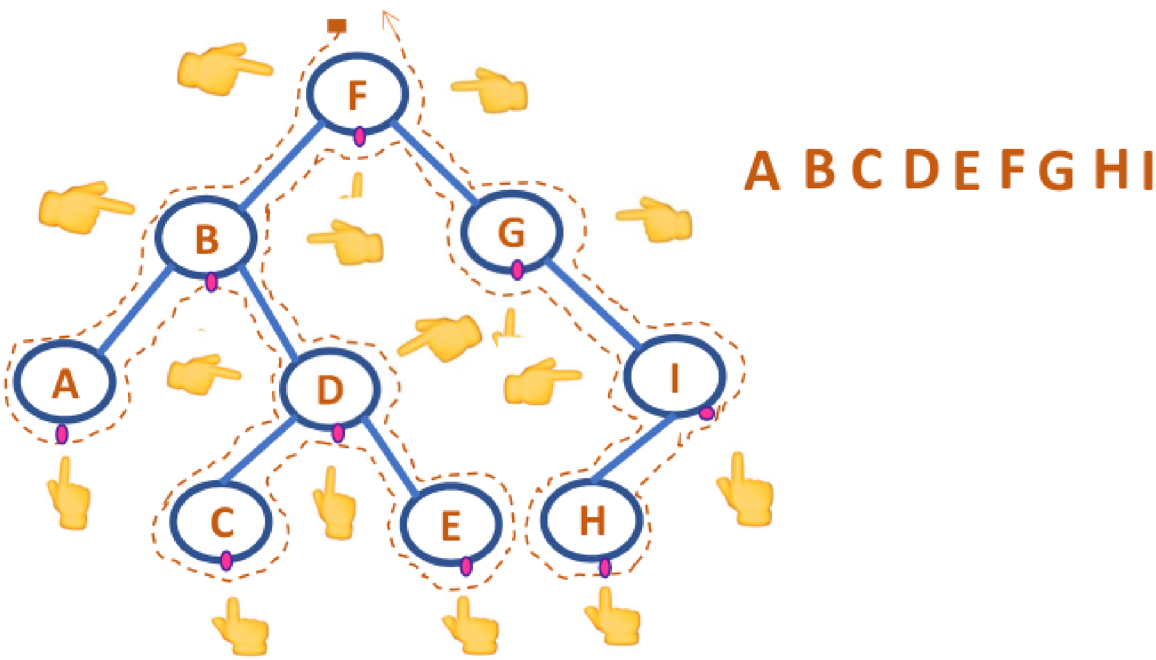
- bst traversal based on order of visiting nodes :
  - `preorder` : first ROOT NODE visited , then LEFT SUBTREE , then RIGHT SUBTREE

- `inorder` : first LEFT SUBTREE visited , then ROOT NODE , then RIGHT SUBTREE
- `postorder` : first LEFT SUBTREE visited , then RIGHT SUBTREE , then ROOT NODE

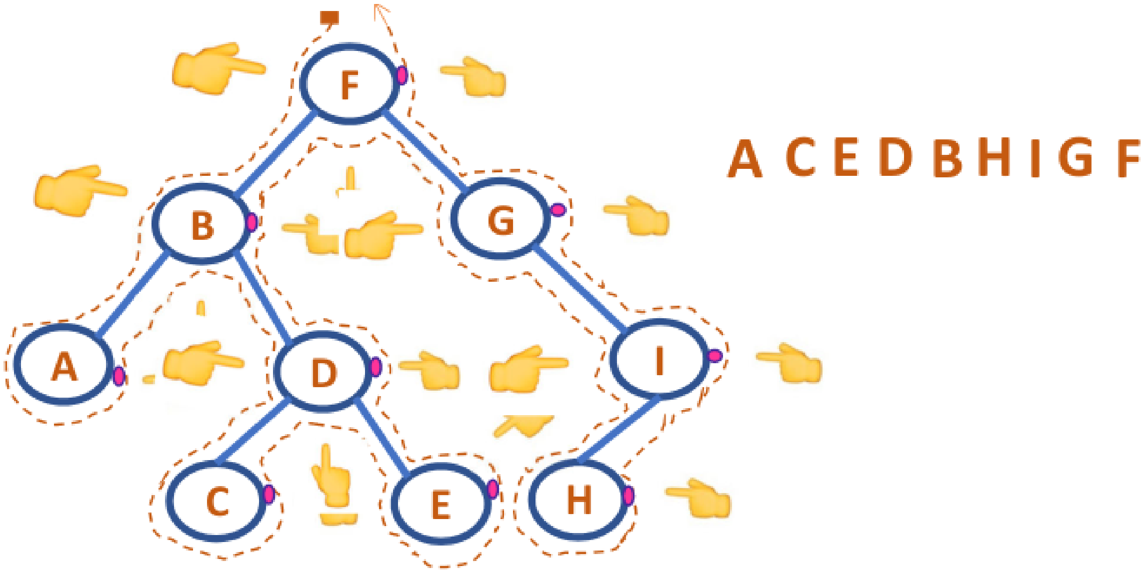
preorder traversal : ROOT,LEFT,RIGHT



inorder traversal : LEFT,ROOT,RIGHT



postorder traversal : LEFT,RIGHT,ROOT



- implementation of traversal in ll

```
void preorder(NODE *t1){
    if(isempty(t1))
        printf("\nEmpty tree");
    else
    {
        printf("%d ",t1->data);
        if(t1->left != NULL)
            preorder(t1->left);
        if(t1->right != NULL)
            preorder(t1->right);
    }
}
void inorder(NODE *t1){
    if(isempty(t1))
        printf("\nEmpty tree");
    else
    {
        if(t1->left != NULL)
            inorder(t1->left);
        printf("%d ",t1->data);
        if(t1->right != NULL)
            inorder(t1->right);
    }
}
void postorder(NODE *t1){
    if(isempty(t1))
```

```
        printf("\nEmpty tree");
    else
    {
        if(t1->left != NULL)
            postorder(t1->left);
        if(t1->right != NULL)
            postorder(t1->right);
        printf("%d ",t1->data);
    }
}
```

- implementation of traversal in array

```
void preorder(int *t,int i){
    if(t[i]!=-1)
    {
        printf("%d ",t[i]);
        preorder(t,2*i+1);
        preorder(t,2*i+2);
    }
}
void inorder(int *t,int i){
    if(t[i]!=-1)
    {
        inorder(t,2*i+1);
        printf("%d ",t[i]);
        inorder(t,2*i+2);
    }
}
void postorder(int *t,int i){
    if(t[i]!=-1)
    {
        postorder(t,2*i+1);
        postorder(t,2*i+2);
        printf("%d ",t[i]);
    }
}
void insert(int *t,int key){
        int i=0;
        while (t[i]!=-1){
                if key>t[i]
                i=2*i+2;
                else
                i=2*i+1;
        }
```

```
        t[i]=key;
}
```

- iterative traversal algorithm:

```
iterativeInorder(root)
{
        s = emptyStack
        current = root
        do {
                while(current != null)
                {
                        /* Travel down left branches as far as possible
                        saving pointers to nodes passed in the stack*/
                        push(s, current)
                        current = current->left
                } //At this point, the left subtree is empty
                poppedNode = pop(s)
                print poppedNode ->info
                //visit the node
                current = poppedNode ->right //traverse right subtree
        } while(!isEmpty(s) or current != null)
}
iterativePreorder(root)
{
        current=root
        if (current == null)
                return
        s = emptyStack
        push(s, current)
        while(!isEmpty(s)) {
                current = pop(s)
                print current->info
                //right child is pushed first so that left is processed
first
                if(current->right !=NULL)
                push(s, current->right)
                if(current->left !=NULL)
                push(s, current->left)
        }
}
iterativePostorder(root)
{
        s1 = emptyStack ; s2 = emptyStack ; push(s1, root)
        while(!isEmpty(s1)) {
```

```
                current = pop(s1)
                push(s2,current)
                if(current->left !=NULL)
                push(s1, current->left)
                if(current->right !=NULL)
                push(s1, current->right)
        }
        while(!isEmpty(s2)) { //Print all the elements of stack2
                current = pop(s2)
                print current->info
        }
}
```

Queue

Simple queue

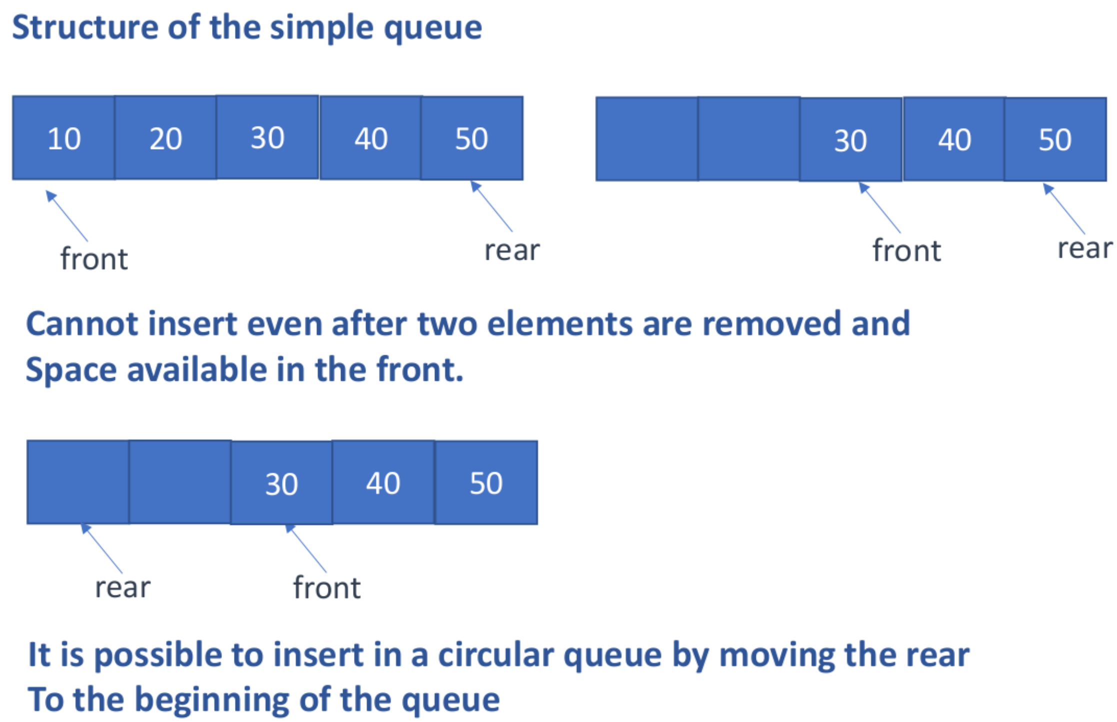
- items are deleted at front,inserted at rear
- FIFO structure
- definition :

```
struct queue
{
int items [MAXQUEUE];
int front, rear;
} ;
```

- insert : do q->items[q->rear++]=new item
- remove : popditem = q->items[q->front++]
- display : for (i=q->front;i<=q->rear;i++) print(q->items[i])
- handle edge cases accordinly depending on queue size,overflow,underflow,single element

Circular queue

- instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.



- FIFO structure
- insert : rear = (rear + 1) % size ; q->item[rear]=x (if rear=front then cannot insert)
- delete : popditem=q[front] ; front = (front + 1) % size ;
- for delete check for underflow(front==-1),single element(front==rear)
- for insert check overflow ((rear+1)%size==front) , empty queue (front==-1)
- display : while (front!=rear) { print q[front] ; front=(front+1)%size : }

Priority Queue

- every item has a priority associated with it.
- In Ascending Priority queue ,the item with lowest priority is removed , but items can be inserted arbitrarily/or on priority
- In descending priority queue, the item with the highest priority is removed , but items can be inserted arbitrarily/or on priority
- definition :

```
struct pqueue
{
        int data;
        int pty;
}
struct pqueue pq[10];
```

- pqinsert : items inserted according to priority nd other items shifted

```
void pqinsert(int x,int pty,struct pqueue *pq,int *count)
{
        // x is item to be inserted
        // pty is the priority of the item
        // pq is the pointer to the priority queue
        // count is the number of items in the queue
        int j;
        struct pqueue key;
        key. data=x;
        key.pty=pty;
        j=*count-1; // index of the initial position of the element
        //compare the priority of the item being inserted with the
        //priority of the items in the queue
        // shift the items down while the priority of the item being
        //inserted is greater than priority of the item in the queue
        while((j>=0)&&(pq[j].pty>key.pty))
        {
                pq[j+1]=pq[j];
                j--;
        }
        pq[j+1]=key; // insert the element at its correct location
        (*count)++;
}
```

- pqdelete : first element popped,other items shifted

```
struct pqueue pqdelete(struct pqueue *pq,int *count)
{
        // pq is a pointer to the priority queue
        // count is the number of elements in the queue
        int i;
        struct pqueue key;
        // if queue is empty, return a structure with priority -1
        if(*count==0)
        {
                key.data=0;
                key.pty=-1;
        }
        //delete the first item
        //shift the other items to the left
        else
        {
                key=pq[0];
                for(i=1;i<=*count-1;i++)
                pq[i-1]=pq[i];
```

```
                (*count)--;
        }
        return key; //return the key with the lowest priority
}
```

Double ended queue : Dequeue

- allows insertion and deletion at both ends
- dequeue - array algorithm

```
Insert Elements at Rear end :
Check whether the queue is full
If rear = size-1
initialise rear to 0.
else
increment rear by 1
insert element at location rear

Insert element front end
Check if the queue is full
If Front =0
move front to last location (size -1)
else
decrement front by 1

Delete element at Rear end
check if the queue is empty
delete the element pointed by rear
If dequeue has one element
front=-1 rear=-1;
If rear is at first index
make rear = size-1
else
decrease rear by 1

Delete element at front end
check if the queue is empty
delete the element pointed by front
If dequeue has one element
front=-1 rear=-1;
If front is at last index
make front = 0
else
increase front by 1
```

- dequeue dll (forget abbout above algo,just insert nd delete like we do dll)
    - structure :

```
struct dequeue
{
        struct node * front;
        struct node * rear;
};
struct node
{
        int data;
        struct node * prev, *next;
        };
struct dequeue dq;
dq.front=dq.rear = NULL
```

- inserthead :

```
temp->next=dq->front; // insert in front
dq->front->prev=temp;
temp->prev=NULL;
dq->front=temp;
```

- inserttail :

```
dq->rear->next=temp;
temp->prev=dq->rear;
dq->rear=temp;
```

- deletehead:

```
dq->front=dq->front->next;
dq->front->prev=NULL;
```

- deletetail:

```
dq->rear=dq->rear->prev;
dq->rear->next=NULL;
```

- check for usual boundary conditions in above code like empty quueue,queue with one element

Simple queue - using LL

- structure of simple queue using ll

```
struct node
{
        int data;
        struct node *next;
};
struct queue
{
        struct node * front;
        struct node *rear;
};
Struct queue q;
q.front=q.rear = NULL;
```

- insert : q.rear->next = newelement , q.rear=newelement
- delete : popditem = q.front , q.front=q.front->next
- display : f=q.front;r=q.rear; while (f!=r) {f=f->next}

CPU scheduler using queue

- First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU.
- new process go to the rear
- when CPU is free , it is allocated to process at the front
- running process is then removed
- `shortest job first preemptive` : when process with short burst time appears, existing process removed and shortest job is executed first
- `shortest job first non preemptive` : process with shortest burst time scheduled first . no interrupts
- `longest job first preemptive` : priority given to long burst time , current process interrupted during execution if process with longer burst time appears
- `longest job first non preemptive` : priority given to long burst time but process cant be interrupted before complete execution
- `round robin scheduling` :
    - process are kept at queue
    - CPU scheduler picks first process, sets timer to interrupt after 1 time quantum
    - if burst time of process < 1 time quantum , process itself releases CPU

- if burst time > 1 time quantum , process interrupted and put at rear of queue
- CPU scheduler selects next process

- `preemptive priority based scheduling` : priority of new proces compared with process in ready queue and one being executed , nd given priority accordingly.highest priority process is given the CPU next.process may get interupted
- `non preemptive priority based scheduling` : process scheduled acc to priority assigned.once process scheduled it runs to completion i.e no interrupts

Josephus problem using queue

- soldiers form a circle and number n is picked from a hat . name is also picked
- starting from name picked , they begin to count clockwise
- when count reaches n soldier is removed from circle and count begins with next soldier
- input : n , list of names (cw)
- output : print in order : names eliminated , soldier who escapes
- implementation using cll

```
int survivor(struct node **head, int n)
{
// head is pointer to first node
        struct node *p, *q;
        int i;
        q = p = *head;
        while (p->next != p)
        {
                for (i = 0; i < n - 1; i++)
                {
                        q = p;
                        p = p->next;
                }
                q->next = p->next;
                printf("%d has been killed.\n", p->num);
                free(p);
                p = q->next;
        }
        *head = p;
        return (p->num);
}
```

- pseudocode of josephus problem using circular queue

```
Pseudo code of implementation using circular queue
Enter n
while(all the names are read)
{
insert name into the queue
read(name)
}
while( q has one element)
{
dequeue n-1 names from the queue and enqueue it.
dequeue the n th name
print the n th name
}
dequeue the only name of the queue
print the name
```