UNIT 1 : Linked Lists and Stacks

» recap of sem-2 C

Functions:

1. Call by value
   - values of actual parameters are copied to formal parameters.
   - any change in formal parameters is not reflected in actual parameters.

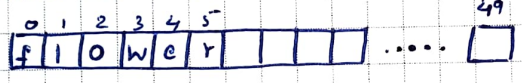
2. Call by reference
   - values of actual parameter address gets stored to formal parameters
   - ∴ any change in formal is reflected in actual

Memory Allocation

Static Memory Allocation

- no. of bytes for a variable is assigned as fixed
- memory allocation takes place at run time
- eg: int a;
  'a' gets fixed 4 bytes during compile time and allocated during run time
- memory allocation happens in stack.

say: char arr[50];



44 bytes are empty :
↳ i.e. underutilization of memory
→ aka internal fragmentation

- size & type of memory must be known
  ↳ allocated by compiler
- memory cannot be deleted explicitly → only overwritten
- there can be underutilization or overutilization of memory.

Dynamic memory Allocation

- used to obtain & release memory during program execution

1. to allocate memory use: `void* malloc (size_t size)`
   - takes no of bytes to allocate
   - returns pointer of type void*
   - if memory unavailable → returns NULL

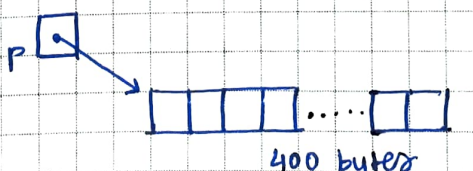
* memory assigned in heap

size_t → means unsigned → size must be +ve

```
int *p;
if ((p = (int*)malloc (100*sizeof(int)) == NULL) {
        printf ("not allocated")
}
free(p); p = NULL;
```

deallocates memory

p → stores address of allocated memory



- malloc allocates a block of contiguous bytes.
  ↳ if not enough contiguous memory it returns NULL.

2. ~~calloc~~
2. void * calloc (size_t nitem, size_t size);

→ all values initialized to default value (0)

3 ~~realloc~~ void* realloc (ptr, size_t size);

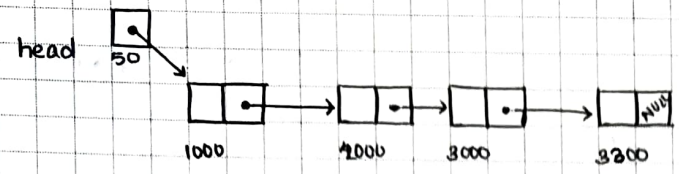
eg: p = (int *) realloc (p, 200 * sizeof(int))

Lists:

Lists in C can be created in two ways
1. contiguous memory - Array list
2. non contiguous memory - Linked list.

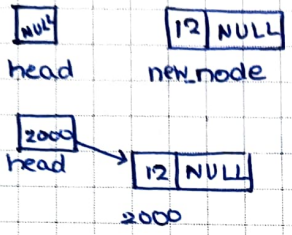
Array List	Linked List
• fixed size : resizing is expensive	• Dynamic sizing
• Insertion/Deletion are inefficient	• Insertion/Deletion are efficient
• Elements are in contiguous memory	• Elements are in non contiguous memory
• sequential and random access is faster	• Sequential and random access is slower.
• may result in memory wastage if allocated space is not used.	• since memory is allocated dynamically (as per requirement) there's no waste

SINGLY LINKED LIST

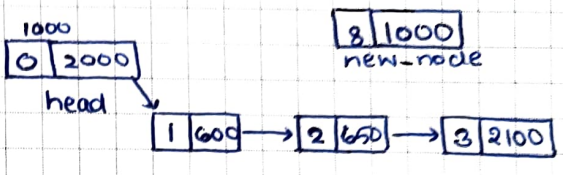


Insertion at beginning

Case 1: list is empty (head is NULL)

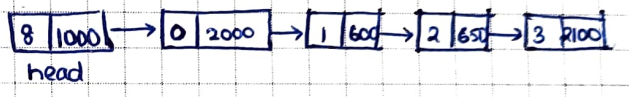


Case 2: list is not empty (head is not NULL)



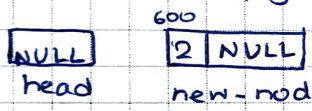
new_node → link = head

head = new_node



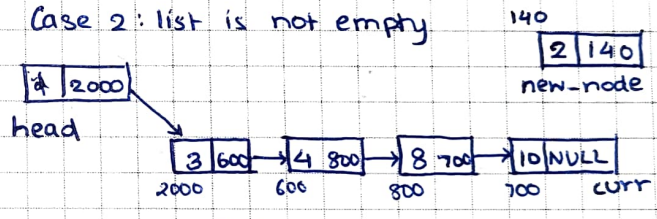
Insertion at end

Case 1: list is empty (head is NULL)



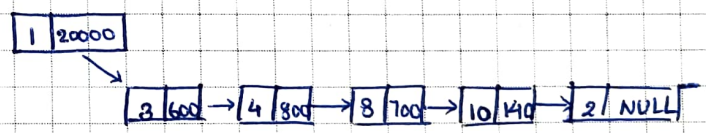
head will now be the new_node

Case 2: list is not empty

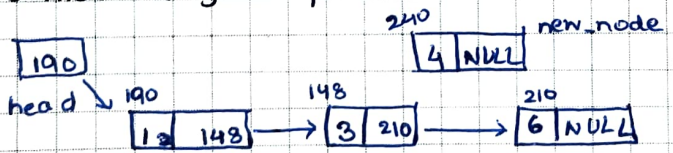


curr → link = new_node

new_node → link = NULL

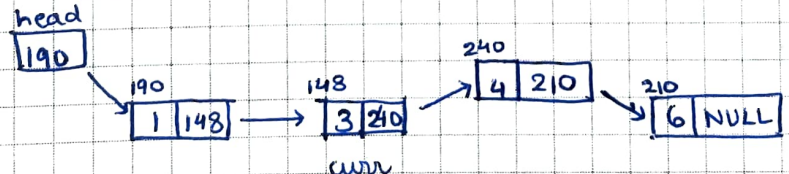


Insertion at given position



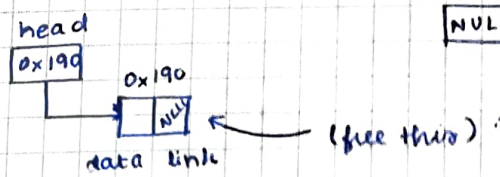
new_node → link = curr → link;

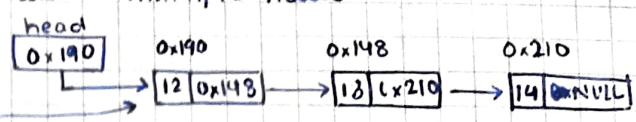
curr → link = new_ele



Deletion At Beginning

Case 1: only one node





* head = NULL

Case 2: multiple nodes

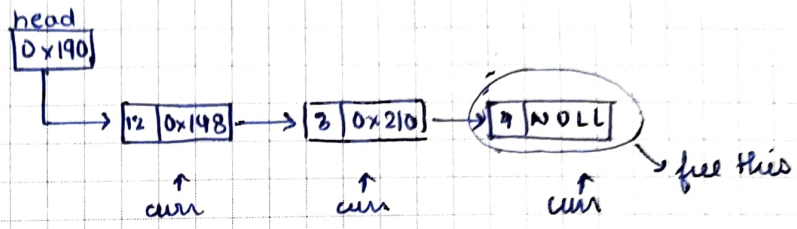
* head = (*head) → link

Deletion At the End

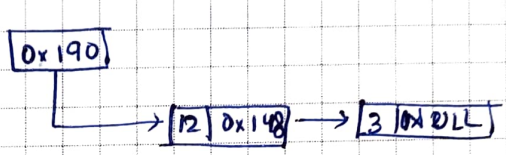
Case 1: only one node

.same as above

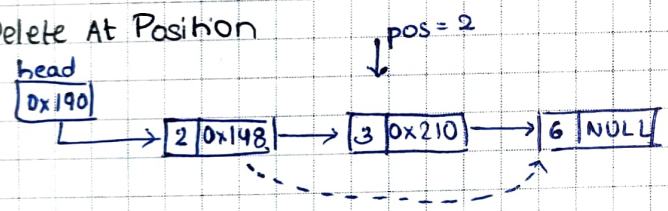
Case 2: multiple nodes.

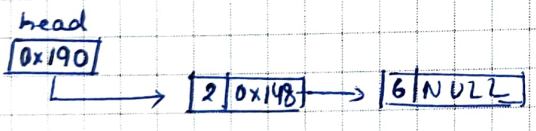



```
while (curr -> link != NULL){
        prev = curr;
        curr = curr -> link; }
if (prev != NULL){
      prev -> link = NULL; }
else{
      *head = NULL; }
```



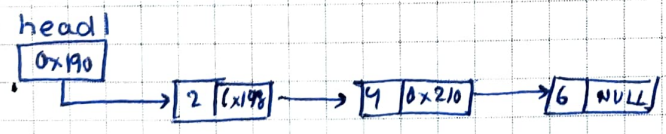
Delete At Position

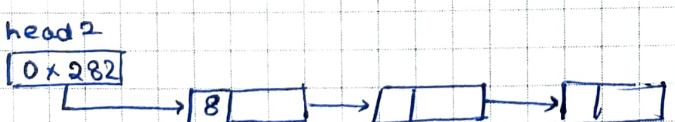


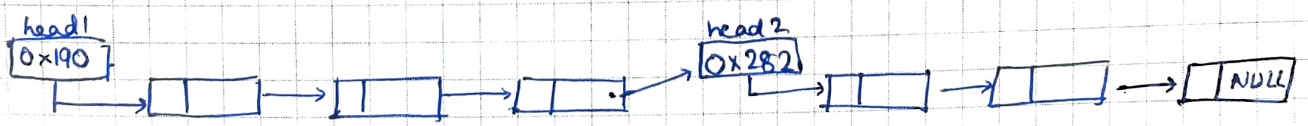



```
while (curr -> link != NULL && i != (pos - 1)){
      prev = curr;
      curr = curr -> link;
        i++;
}
prev -> link = curr -> link;
curr -> link = NULL;
free(curr);
```

Concatinate



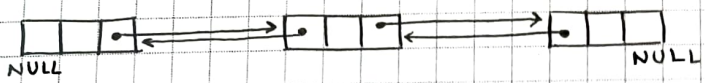




To Solve : (common to SLL, DLL)

1. Insert at alternate positions
2. Delete at alternate positions
3. Polynomials (addition and subtraction)
4. Remove 'nth' node from end of list
5. swap nodes in pairs
6. reverse a linked list
7. perform a sorting algorithm (bubble, insertion, quick ...)
8. middle of a linked list.
9. find intersection of two linked lists.
10. rotate list by "k" places
11. Remove duplicates

DOUBLY LINKED LIST

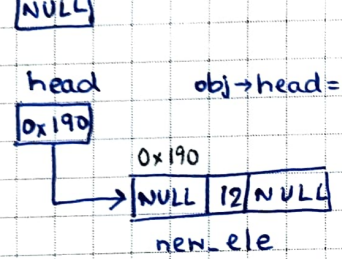


- can be traversed in either direction
- some operations, such as deletion & inserting before a node becomes easier.

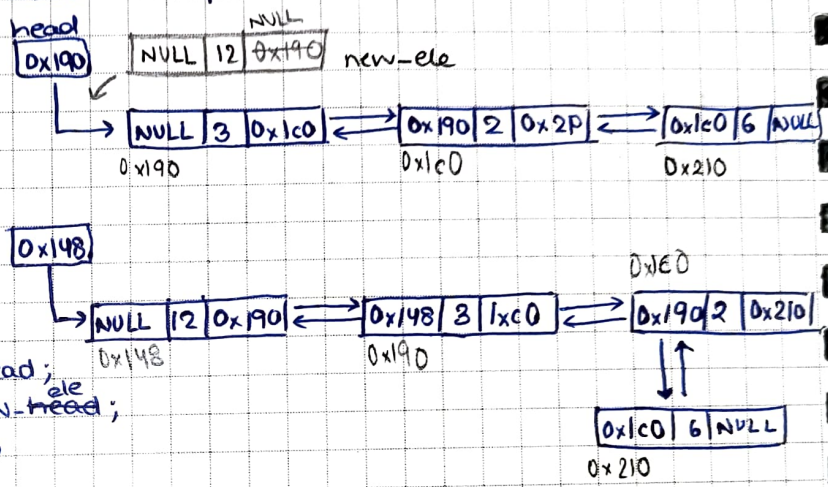
- requires more space
- list manipulations are slower
- greater chance of having bugs (bc more links)

Insertion at Beginning

Case 1: empty list



Case 2: multiple nodes

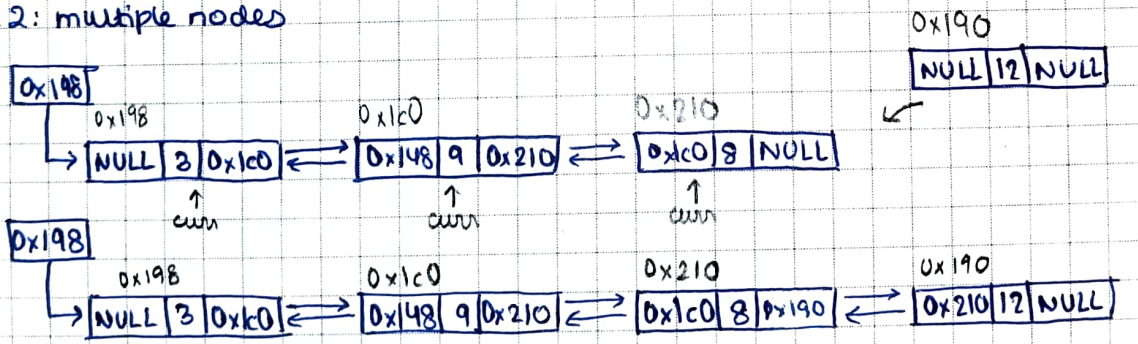



```
new_ele→rlink = obj→head;
obj→head→llink = new_ele;
obj→head = new_ele;
```

Insertion at End

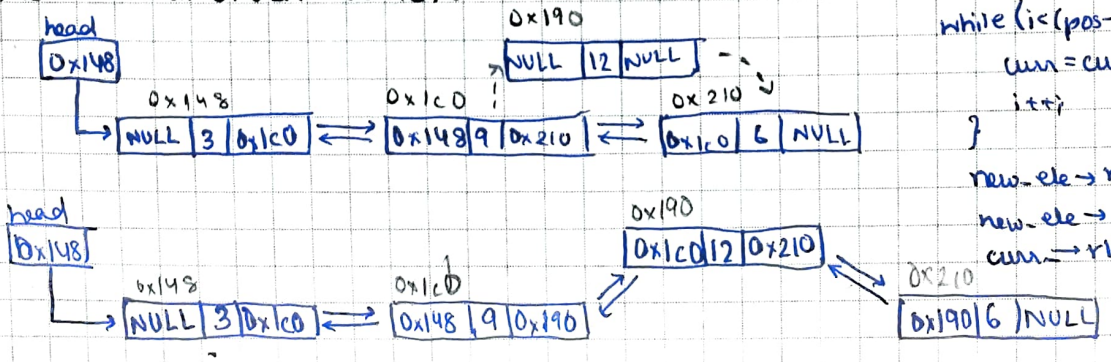
Case 1: empty list : same as case 1 above

Case 2: multiple nodes




```
curr→rlink = new_ele;
new_ele→llink = curr;
```

Insertion at Given Position

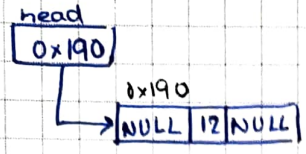



```
while (i<(pos-1) && curr→rlink!=NULL){
    curr = curr→link;
    i++;
}
new_ele→rlink = curr→rlink;
new_ele→llink = curr;
curr→rlink = new_ele;
```

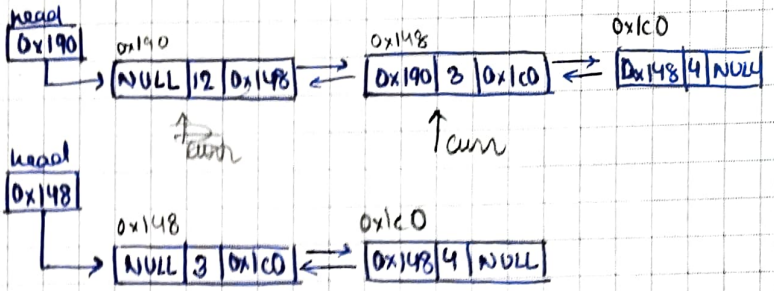
Delete First Node

Case 1: empty linked list - no deletion takes place

Case 2: linked list with single node



Case 3: multiple nodes




```
if(obj->head->rlink==NULL){
    obj->head=NULL;
    return;
}
```

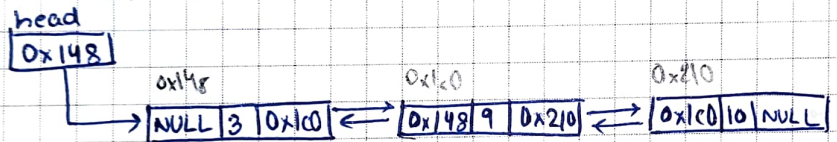
```
node* curr = obj->head->rlink;
curr->llink = NULL;
obj->head = curr;
```

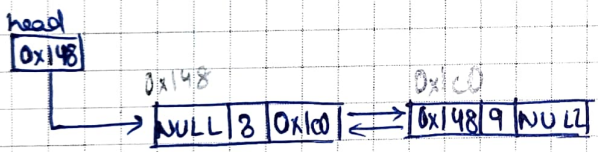
Deletion At End

Case 1: empty - no deletion takes place.

Case 2: single node - same as case 2 in previous

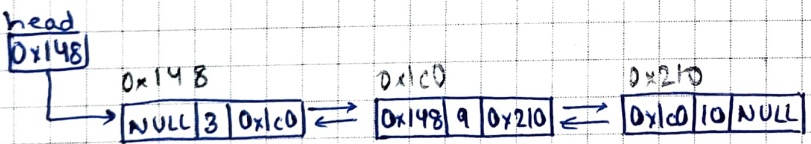
Case 3: multiple nodes

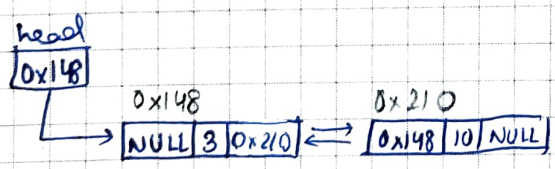





```
while(curr->rlink!=NULL){
    prev=curr;
    curr=curr->rlink;
}
prev->rlink=NULL;
```

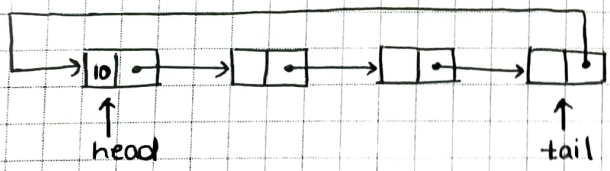
Deletion At Position





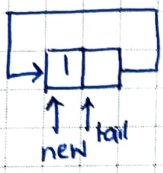

```
while(i<(pos) && curr->rlink!=NULL){
    prev=curr;
    curr=curr->rlink;
    i++;
}
prev->rlink=curr->rlink;
free(curr);
```

CIRCULAR SINGLY LINKED LIST

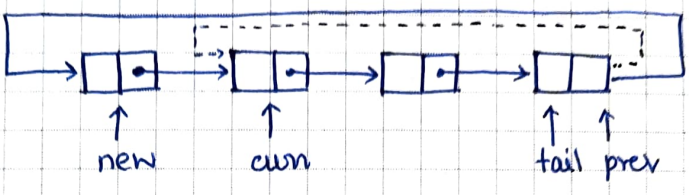


Insertion at beginning

Case 1: empty list

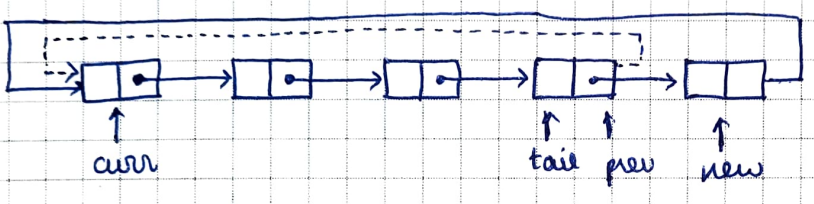


Case 2: non-empty list



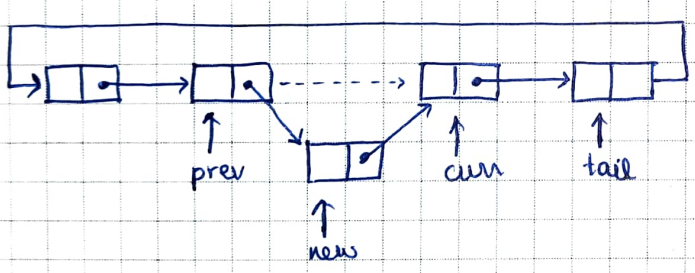

```
new->link = obj->tail->link;   (curr)
obj->tail->link = new;
(prev->link)
```

Insertion at End

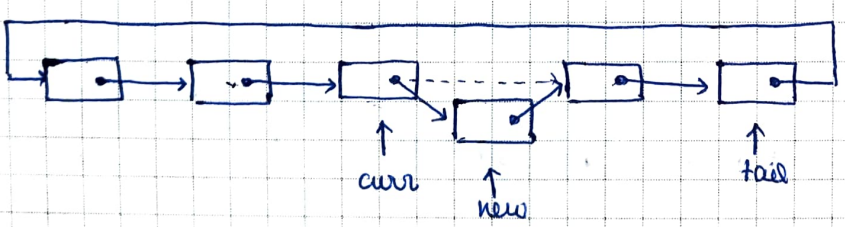



```
new->link = obj->tail->link;
obj->tail->link = new;
obj->tail = new;
```

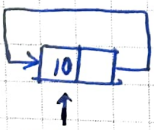
Insert at Middle



Insert at Position

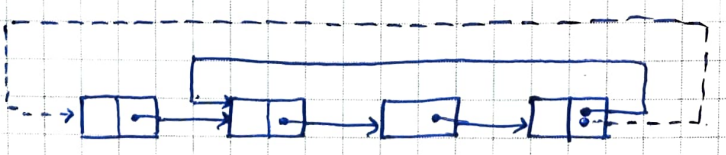


Delete at Beginning



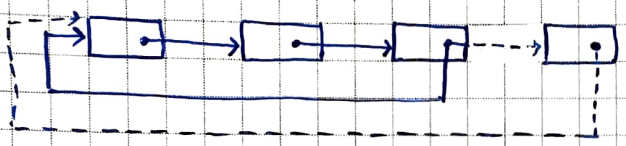
tail = curr = prev

tail = NULL;



```
node *curr = obj->tail->link;
obj->tail->link = curr->link;
free(curr);
```

Delete at End



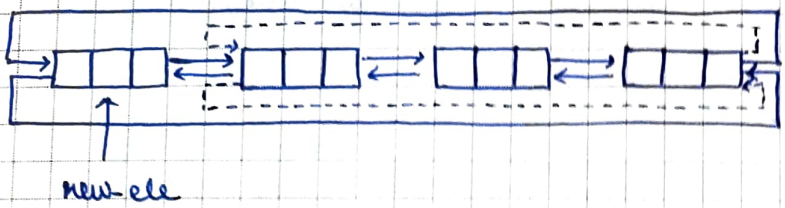
```
node *curr = obj → tail → link;
while (curr → link != obj → tail){
        curr = curr → link;
}
curr → link = obj → tail → link;
obj → tail = curr;
```

Delete at Position

```
node *curr = obj → tail → link;
node *prev = NULL;
int j = count(obj);

if (pos < 1 || pos > j){ printf("out of bounds"); }
else
int i=1;
while (i<pos && curr → link != obj → tail) {
        prev = curr;
        curr = curr → link;
        i++;
}
prev → link = curr → link;
```

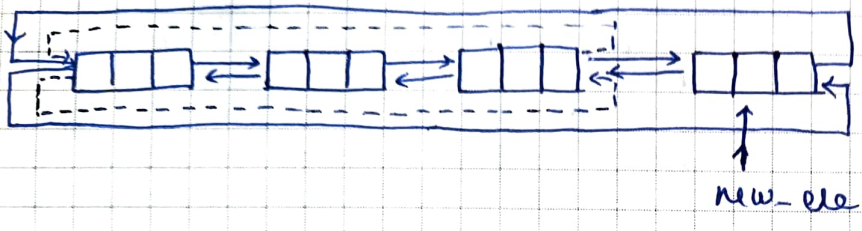
CIRCULAR DOUBLY LINKED LIST

Insert at Beginning



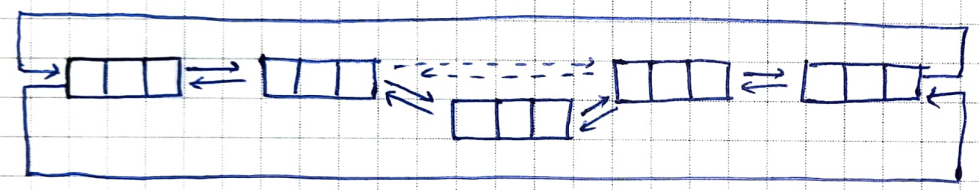

```
node *last = obj->head->llink;
new_ele->rlink = obj->head;
new_ele->llink = last;
last->rlink = new_ele;
obj->head->llink = new_ele;
obj->head = new_ele;
```

Insert at End



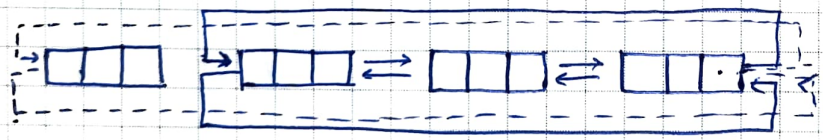

```
node *last = obj->head->llink;
last->rlink = new_ele;
new_ele->llink = last;
new_ele->rlink = obj->head;
obj->head->llink = new_ele;
```

Insert at Position



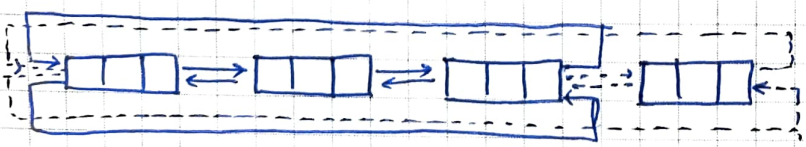
```
while (i < (pos-1) && curr->rlink != obj->head)
{
    curr = curr->rlink;
    i++;
}
new_ele->rlink = curr->rlink;
new_ele->llink = curr;
curr->rlink->llink = new_ele;
curr->rlink = new_ele;
```

Delete at Beginning



```
node *curr = obj->head;
node *last = obj->head->llink;
last->rlink = curr->rlink;
curr->rlink->llink = last;
obj->head = curr->rlink;
```

Delete at End



```
node *second_last = last->llink;
second_last->rlink = obj->head;
obj->head->llink = second_last;
```

Delete at Position

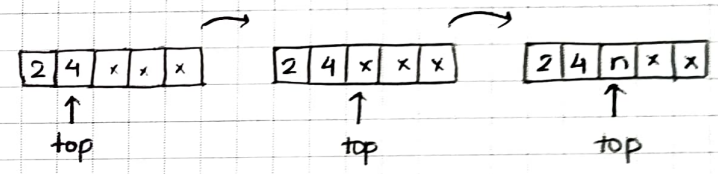
```
while(i < pos && curr->rlink != obj->head) {
    curr = curr->rlink;
    i++;
}
curr->llink->rlink = curr->rlink;
curr->rlink->llink = curr->llink;
```

STACK - USING ARRAY

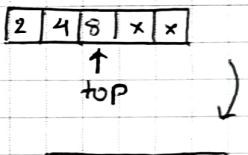
(LIFO)

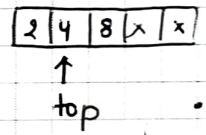
```
#define MAX 5
typedef struct Stack{
        int top;
        int s[MAX];
}Stack;

void init(Stack *obj){
     obj->top = -1;
}

void push(int n, Stack *obj);
void pop(Stack *obj);
void display(Stack *obj);

int main(){
        .....
      }
```

```
void push(int n, Stack *obj){
        if(obj->top == (MAX-1)){
                printf("overflow");
        }
        else{
                obj->top++;
                obj->s[obj->top] = n;
        }
}
```




```
void pop(Stack *obj){
        if(obj->top == -1){
                printf("underflow");
        }
        else{
                printf("deleted element : %d", obj->s[obj->top]);
                obj->top--;
        }
}
```





- no actual memory deletion, i.e. physical deletion. It is a virtual deletion.

```
void display(Stack *obj){
        if(obj->top == -1){
                printf("underflow");
        }
        else{
                for(int i=obj->top; i>0; i--){
                        printf("%d", obj->s[i]);
                }
                printf("\n\n");
        }
}
```

STACK - USING SINGLY LINKED LIST

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

typedef struct stack{
        node *top;
}stack;

void init(stack *obj){
        obj->top = NULL;
}

int main(){
        ...
}

void push(int n, stack *obj){
        node *new_ele = (node*)malloc(sizeof(node));
        new_ele->data = n;
        new_ele->next = obj->top;
        obj->top = new_ele;
}

void pop(stack *obj){
        if(obj->top == NULL){
                printf("underflow");
        }
        else{
                node *p = obj->top;
                printf("deleted: %d", obj->top->data);
                obj->top = obj->top->next;
                free(p);
        }
}

void display(stack *obj){
        node *p = obj->top;
        if(obj->top == NULL){
                printf("overflow");
        }
        else{
                while(p!=NULL){
                        printf("%d\n", p->data);
                        p = p->next;
                }
                printf("\n\n");
        }
}
```

SIMPLE QUEUE - USING ARRAY

(FIFO)

```
#define MAX 30

typedef struct Queue {
    int front, rear;
    int q[MAX];
} Queue;

void init (Queue *obj) {
    obj -> front = 0;
    obj -> rear = -1;
}

void enqueue (int n, Queue *obj) {
    if((obj->rear) == (MAX-1)) {
        printf("overflow");
    }
    else {
        (obj -> rear)++;
        obj -> q[obj->rear] = n;
    }
}

void dequeue(Queue *obj){
    if(obj -> front > obj -> rear){
        printf("empty ");
    }
    else {
        printf("deleted element : %d\n", obj -> q[obj -> front]);
        obj -> front++;
    }
}

void display (Queue *obj){
    if(obj -> rear == -1) {
        printf ("underflow ");
    }
    else{
        for (int i = obj -> front ; i <= obj -> rear; i++) {
            printf ("%d\n", obj -> q[i]);
        }
    }
}
```

SIMPLE QUEUE - USING SINGLY LINKED LIST

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

typedef struct queue{
        node *front;
        node * rear;
}queue;

void init(queue *obj){
        obj->front = NULL;
        obj->rear = NULL;
}

void enqueue (int n, queue *obj){
        node *new_ele = (node*)malloc(sizeof(node));
        new_ele->data = n;
        new_ele->next = NULL;

        if(obj->rear == NULL){
                obj->front = new_ele; obj->rear = new_ele;
        }
        else{
                obj->rear->next = new_ele;
                obj->rear = new_ele;
        }
}

void dequeue(queue *obj){
        if(obj->rear == NULL){
                printf("underflow");
        }
        else{
                printf("deleted: %d\n", obj->front->data);
                obj->front = obj->front->next;
        }
}

void display(queue *obj){
        node *p = obj->front;
        if(obj->rear == NULL){
                printf("underflow");
        }
        else{
                while(p!=NULL){
                        printf("%d\n", p->data);
                        p = p->next;
                }
                printf("\n\n");
        }
}
```

CIRCULAR QUEUE - USING CIRCULAR ARRAY

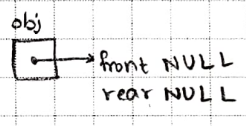
(FIFO)

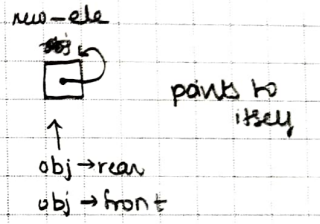
```
#define size 5

typedef struct Queue {
        int front;
        int rear;
        int q[size];
}Queue;

void init (Queue *obj){
    obj -> rear = -1;
    obj -> front = -1;
}

void enqueue(int n, Queue *obj) {
    if((obj->rear+1)% size == obj -> front){   → checks that the next position is
                printf ("queyflow \n");          not same as front
    }
    else {
        obj -> rear = (obj -> rear + 1) % size;
        obj -> q[obj -> rear] = n;
        if (obj -> front == -1) {                → if its the first element
                obj -> front = 0;
        }
    }
}

void dequeue (Queue *obj) {
        if (obj -> front == -1) {
            printf (" underflow \n");
        }
        else if (obj -> front == obj -> rear){
            printf ("%d \n", obj -> q[obj -> front]);  obj -> front = obj -> rear = -1;
        }
        else{
            printf ("%d\n", obj -> q[obj -> front]);
            obj -> front = (obj -> front + 1) % size);
        }
}

void display (Queue *obj){
            if (obj -> front == -1){ printf ("underflow \n"); }
            else{
                int curr = obj -> front;
                while (curr != obj -> rear){
                        printf ("%d ", obj -> q[curr]);
                        curr = (curr + 1) % size;
                }
            printf ("%d ", obj -> q[curr]);
}
```

CIRCULAR QUEUE - USING SINGLY LINKED LIST

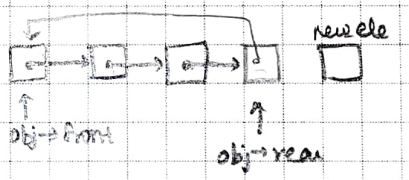
(doubly linked list may also be asked)

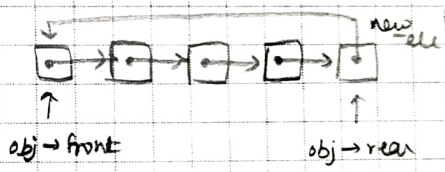
```
typedef struct Node{
        int data;
        struct Node *next;
}node;

typedef struct Queue{
        node *front;
        node *rear;
}queue;

void init(queue *obj){
        obj->front = NULL;
        obj->rear = NULL;
}

void enqueue(int d, queue *obj){
        node* new_ele = (node*) malloc(sizeof(node));
        new_ele->data = d;
        new_ele->next = NULL;

        if(obj->rear == NULL){
                obj->front = obj->rear = new_ele;
                obj->rear->next = obj->front;
        }
        else{
                obj->rear->next = new_ele;
                new_ele->next = obj->front;
                obj->rear = new_ele;
        }
}

void dequeue(queue*obj){
        if(obj->front == NULL){ printf("underflow\n"); return; }
        node *temp = obj->front;
        if(obj->front == obj->rear){
                obj->front = obj->rear = NULL;
        }
        else{
                obj->front = obj->front->next;
                obj->rear->next = obj->front;
        }
}

void display(queue *obj){..
```





↳ to make it point to itself

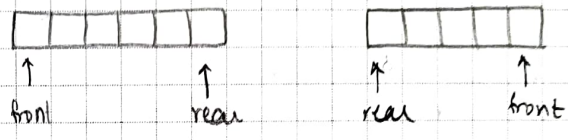




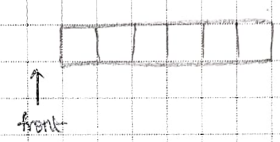
DOUBLE ENDED QUEUE - DEQUE - USING ARRAYS (CIRCULAR?)

(not FIFO)

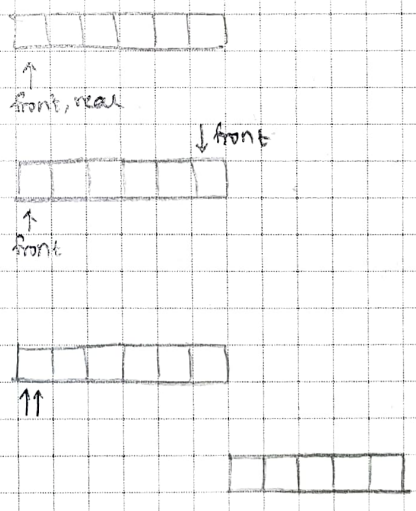
```
#define sz 100

typedef struct deque{
        int arr[sz];
        int front;
        int rear;
}deque;

void init(deque *dq){
        dq->front = -1;
        dq->rear = -1;
}

int isFull(deque *dq){
        if(dq->front==0 && dq->rear == sz-1) || (dq->front == (dq->rear+1)%sz)){
                return 1;
        }
        else{ return 0; }
}
```




```
int isEmpty(deque *dq){
        if(dq->front==-1){ return 1; }
        else{ return 0; }
}
```




```
void insert_front(int n deque *dq){
    if(isFull(dq))
            return;
    if(isEmpty(dq)){
                    dq->front = dq->rear = 0;
    }
    else if(dq->front == 0){
                dq->front = sz-1;
    }
    else if(dq->front == -1 && dq->rear == -1){
            dq->front = 0;
            dq->rear = 0;
    }
    else{
        dq->front--;
    }
    dq->arr[dq->front] = n;

}
```



```
void insert_rear (int n, deque *dq) {
        if isFull (dq){
                return;
        }
        else if ( dq -> front == -1 && dq -> rear == -1) {
                dq -> front = dq -> rear = 0;
        }
        else if ( dq -> rear == sz-1){
                dq -> rear = 0;
        }
        else {
            dq -> rear++;
        }
        dq -> arr [dq -> rear] = n;
}

void delete_front (deque *dq){
        if(isEmpty (dq)) {
                return;
        }
        if ( dq -> front == dq -> rear) {
                dq -> front = dq -> rear = -1;
        }
        else if (dq -> front == sz-1){
                dq -> front = 0;
        }
        else {
            dq -> front++;
        }
}

void delete_rear(deque *dq) {
        int ele = dq -> arr [dq -> rear];
        if (isEmpty (dq)) {
                return;
            }
        else if (dq -> front == dq -> rear) {
                dq -> front = dq -> rear = -1;
        }
        else if (dq -> rear == 0) {
                dq -> rear = sz-1;
        }
        else {
            dq -> rear --;
        }
}

void display (deque *dq){
        if (isEmpty (dq)) {return;}
        int i = dq -> front;

        while (1) {
                printf ("%d ", dq -> arr [i]);
                if (i == dq -> rear); break;
                i = (i+1) % sz;
        }
}
```

DEQUE - USING DOUBLY LINKED LIST

```
typedef struct Node{
        int data;
        struct Node* prev;
        struct Node* next;
} Node;

typedef struct Deque {
        Node *front;
        Node *rear;
} Deque;

void init(Deque *dq) {
        dq -> front = NULL;
        dq -> rear = NULL;
}

void insert_front( Deque *dq, int data){
        Node *new_node = (Node*) malloc (sizeof (Node));
        new_node -> data = data;
        new_node -> prev = NULL;
        new_node -> next = dq -> front;

        if (dq -> front == NULL) {
                dq -> rear = new_node;
        }
        else {
                dq -> front -> prev = new_node;
        }
        dq -> front = new_ele;
}

void insert_rear (Deque *dq, int data) {
        Node *new_node = (Node*) malloc (sizeof (Node));
        new_node -> data = data;
        new_node -> next = NULL;
        new_node -> prev = q -> rear;

        if (dq -> front == NULL) {
                dq -> front = new_node;
        }
        else {
                dq -> rear -> next = new_node;
        }
        dq -> rear = new_node;
}
```

```
void delete_front(Deque *dq){
        if (dq->front == NULL) {
                return;
        }
        Node *temp = dq->front;
        dq->front = dq->front->next;
        if (dq->front==NULL){
                dq->rear == NULL;
        }
        else{
                dq->front->prev = NULL;
        }
        free(temp);
}
```


```
void delete_rear(Deque *dq){
        if(dq->front == NULL) { return; }
        Node *temp = dq->rear;
        dq->rear = dq->rear->prev;

        if (dq->rear == NULL){
                dq->front = NULL;
        }
        else{
                dq->rear->next = NULL;
        }
        free(temp);
}

void display(Deque *dq){

        if (dq->front == NULL){
                        return;
        }
        Node *curr = dq->front;
        while (curr != NULL){
                printf("%d ", curr->data);
                curr = curr->next;
        }
}
```

DEQUE - USING ARRAYS

```
#define MAX 6

typedef struct queue{
        int q[MAX];
        int front;
        int rear;
} Queue;

void init (Queue *obj){
        obj -> rear = MAX;
        obj -> front = -1;
}

void insert_rear (Queue *obj, int n){
    if (obj -> front == obj -> rear - 1){
                        printf("overflow");
                        return;
    }
    obj -> rear--;
    obj -> q[obj -> rear] = n;
}

void insert_front (Queue *obj, int n){
        if(obj -> front == obj -> rear - 1) {
                        printf("overflow");
                        return;
        }
        obj -> front++;
        obj -> q[obj -> front] = n;
}
```

PRIORITY QUEUE - ARRAY

```
#define MAX 6

typedef struct{
        int data;
        int priority;
}Element;

typedef struct{
        Element elements[MAX];
        int size;
}queue;

void init(queue* pq){
        pq->size = 0;
}

void insert(queue*pq, int d, int p){
        pq->elements[pq->size].data = d;
        pq->elements[pq->size].priority = p;
        pq->size++;
}

void display(queue*pq){
    if(pq->size == 0){
        printf("empty!\n");
        return;
    }
    printf("Data : Priority\n");
    for(int i=0; i<pq->size; i++){
        printf("%d: %d", pq->elements[i].data, pq->elements[i].priority);
    }
}

void deque_min(queue *pq){
    if(pq->size==0){ printf("emphy"); return;}

    int min_index = 0;
    for(int i=1; i< pq->size; i++){
        if(pq.elements[i].priority < pq.elements[min_index].priority){
            min_index = i;
        }
    }

    for(int i= min_index; i< pq->size-1; i++){
        pq->elements[i] = pq->elements[i+1];
    }
    pq->size--;
}
```

```
void deque_max( queue *pq){
    if (pq->size == 0){
            printf("empty ! \n"); return;
    }

    int max_index = 0;

    for (int i=1; i<pq->size; i++){
        if (pq->elements[i].priority > pq->elements[max_index].priority){
            max_index = i;
        }
    }

    printf (" dequeued : %d , priority: %d \n", pq->elements[max_index].data,
                                                pq->elements[max_index].priority);


    for (int i= max_index; i<pq->size-1; i++){
        pq->elements[i] = pq->elements[i+1];
    }
    pq->size--;
}
```

PRIORITY QUEUE - LINKED LIST

```
typedef struct node{
        int d;
        int priority;
        struct node* link;
}node;

typedef struct list{
        node *head;
}list;

void init(list *obj){
        obj->head = NULL;
}

node * create_node(int d, int p){
        node* new_node = (node*)malloc(sizeof(node));
        new_node->d = d;
        new_node->priority = p;
        new_node->link = NULL;
        return new_node;
}

void display(list *obj){
        node* curr = obj->head;
        while(curr != NULL){
                printf("%d ", curr->d);
                curr = curr->link;
        }
}

void insert(list *obj, int d, int p){
        node * new_node = create_node(d,p);

        if(obj->head == NULL || obj->head->priority >= p){
                new_node->link = obj->head;
                obj->head = new_node;
        }

        else{
                node* curr = obj->head;
                while(curr->link != NULL && curr->link->priority < p){
                        curr = curr->link;
                }
                new_node->link = curr->link;
                curr->link = new_node;
        }
}
```

```
void deque_min (list *obj){
        node *curr = obj->head;

        if (obj->head == NULL){
                printf("empty! \n");
                return;
        }

        node *prev = NULL;
        node *prevmin = NULL;
        node* min = curr;

        while (curr != NULL){
                if (curr->priority < min->priority){
                        min=curr;
                        prevmin = prev;
                }
                prev = curr;
                curr = curr->link;
        }
        if (prevmin == NULL){
                obj->head = min->link;
        }
        else {
                prevmin->link = min->link;
        }
}


void deque_max(list *obj){
        node* curr = obj->head;

        if (obj->head == NULL){
                printf("empty! \n");
                return;
        }

        node *prev = NULL;
        node* prevmax = NULL;
        node *max = curr;

        while (curr != NULL){
                if (curr->priority > max->priority){
                        max = curr;
                        prevmax = prev;
                }
                prev = curr;
                curr = curr->link;
        }
        if (prevmax == NULL){
                obj->head= max->link;
        }
        else {
                prevmax->link = max->link;
        }
}
```

INFIX TO POSTFIX

1. scan from left to right
2. if scanned character is an operand, put it in postfix expression
3. Otherwise:
   - if precedence & associativity of scanned operator is greater than operator on top of stack
     - when right associativity: when operator at top of the stack and scanned are '^' → pushed into stack
     - when left associativity: pop operators from the stack if their precedence is greater than or equal to the scanned operator, then push the scanned operator.
4. If you encounter parantheses:
   - push '(' to the stack
   - for ')', pop the stack to the postfix expression till a '(' is encoontered, then discard both paranthesis
5. Repeat till entire infix expression is scanned
6. After scanning, pop all remaining operators from the stack to the postfix expression.
7. Output the final postfix expression.

INFIX TO PREFIX

1. Reverse the infix expression, swap '(' with ')' and vice versa
2. if the character is an operand add it directly
   ~~scan from left to right~~
   If the character is an operator
   - pop operators & add to expression if their precedence is greater than the current operator.
   - don't pop operators with equal precedence; just push the current operator onto the stack
   - push current operator onto stack

   If character is paranthesis → '(' → push onto stack
   ')' → pop till '(' is encountered } discard the paranthesis
3. Reverse the postfix expression

1. A+B*C

infix	stack	postfix
A		A
+	+	A
B	+	AB
*	* / +	AB
C	* / +	ABC
		ABC*+

2. (A+B)*(C+D)

infix	stack	postfix
(	(	
A	(	A
+	+ / (	A
B	+ / (	AB
)		AB+
*	*	AB+
(	(/ *	AB+
C	(/ *	AB+C
+	+ / (/ *	AB+C
D	+ / (/ *	AB+CD
)		AB+CD+*

3. A+B*C/D-F+A^E

infix	stack	postfix
A		A
+	+	A
B	+	AB
*	* / +	AB
C	* / +	ABC
/	/ / +	ABC*
D	/ / +	ABC*D
-	-	ABC*D/+
F	-	ABC*D/+F
+	+	ABC*D/+F-
A	+	ABC*D/+F-A
^	^ / +	ABC*D/+F-A
E	^ / +	ABC*D/+F-AE
		ABC*D/+F-AE^+

4. A^B*C-D+E/F/(G+H)

infix	stack	postfix
A	⌊⌋	A
^	⌊^⌋	A
B	⌊^⌋	AB
*	⌊*⌋	AB^
C	⌊*⌋	AB^C
-	⌊-⌋	AB^C*
D	⌊-⌋	AB^C*D
+	⌊+⌋	AB^C*D-
E	⌊+⌋	AB^C*D-E
/	⌊/ / +⌋	AB^C*D-E
F	⌊/ / +⌋	AB^C*D-EF
/	⌊/ / +⌋	AB^C*D-EF/
(	⌊(/ +⌋	AB^C*D-EF/
G	⌊(/ +⌋	AB^C*D-EF/G
+	⌊+ (/ +⌋	AB^C*D-EF/G
H	⌊+ (/ +⌋	AB^C*D-EF/GH
)	⌊/ +⌋	AB^C*D-EF/GH+
		AB^C*D-EF/GH+/+

5. K+L-M*N+(O^P)*W/U/V*T+Q

infix	stack	postfix
K		K
+	⌊+⌋	K
L	⌊+⌋	KL
-	⌊-⌋	KL+
M		KL+M
*	⌊* / -⌋	KL+M
N		KL+MN
+	⌊+⌋	KL+MN*-
(	⌊(/ +⌋	KL+MN*-
O		KL+MN*-O
^	⌊^ (/ +⌋	KL+MN*-O
P		KL+MN*-OP
)	⌊+⌋	KL+MN*-OP^
*	⌊* / +⌋	KL+MN*-OP^
W		KL+MN*-OP^W
/	⌊/ / +⌋	KL+MN*-OP^W*
U		KL+MN*-OP^W*U
/	⌊/ / +⌋	KL+MN*-OP^W*U/
V		KL+MN*-OP^W*U/V
*	⌊* / +⌋	KL+MN*-OP^W*U/V/
T		KL+MN*-OP^W*U/V/T
+	⌊+⌋	KL+MN*-OP^W*U/V/T*+
Q		KL+MN*-OP^W*U/V/T*+Q
		KL+MN*-OP^W*U/V/T*+Q+

6. (A+B)*C

6. C*(B+A)

infix	stack	prefix
C		C
*	*	C
(	(*	C
B		CB
+	+ (*	CB
A		CBA
)	*	CBA+

CBA+*

*+ABC

7. (A+B)*(C-D)^E*F

F*E^(D-C)*(B+A)

infix	stack	prefix
F		F
*	*	F
E		FE
^	^ *	FE
(	(^ *	FE
D		FED
-	- (^ *	FED
C		FEDC
)	^ *	FEDC-
*	*	FEDC-^*
(	(*	FEDC-^*
B		FEDC-^*B
+	+ (*	FEDC-^*B
A		FEDC-^*BA
)	*	FEDC-^*BA+
		FEDC-^*BA+**

**+AB^-CDEF

MULTILIST – SPARSE MATRIX

TRIPLE NOTATION

<row no column no value>

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 \end{bmatrix}$$

row no	column no	value
0	2	3
1	4	5
3	1	8

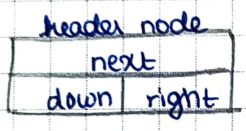
so the triple notation would be,

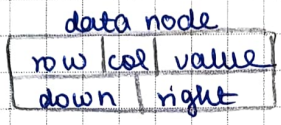
row: [0, 1, 3]
column: [2, 4, 1]
values: [3, 5, 8]

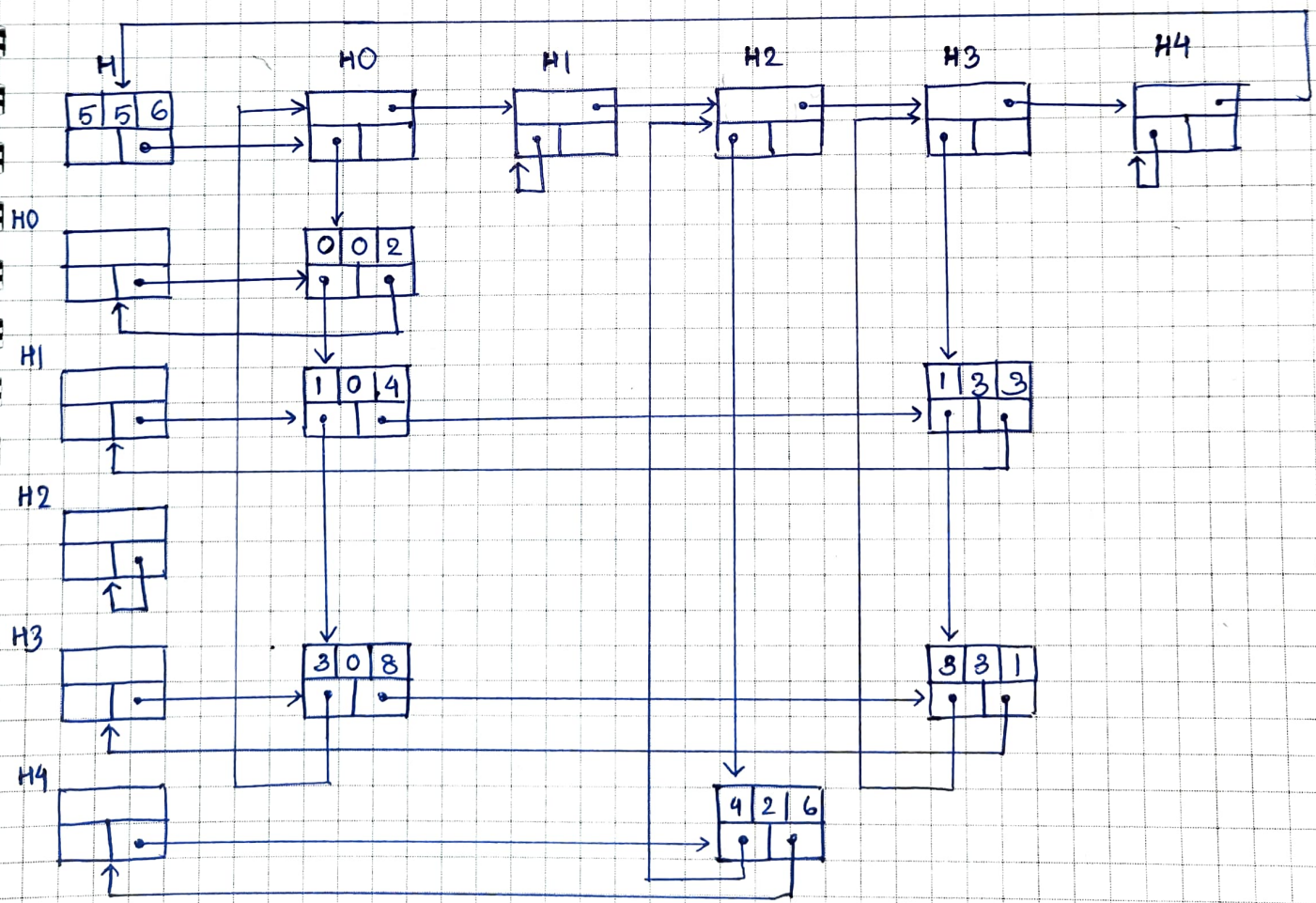
- storage efficiency: more space efficient for storing sparse matrices, especially when the matrix is very large but only a few elements are non-zero.

→ ideal for algorithms that need to efficiently handle sparse matrices without storing large amounts of zero data.

MULTILIST (CIRCULAR LINKED LIST)







SKIP LIST

Key Features:

1. size(): O(1) if size is explicitly stored; otherwise, O(n)
2. isEmpty(): O(1)
3. FindElement(k): O(n) is the worst case
   O(log n) on average
4. InsertItem(k, e): O(1) if the item is already found, otherwise O(log n) on average
5. Remove(k): O(1) if the item is already found, otherwise O(log n) on average

Structure:

Level 0 : the base level, which is a sorted, singly linked list of all nodes
1 : links alternate nodes from the level below
2 : links every 4th node from level below
i : links every 2th node from the level below

no. of levels: approx $[\log_2 n]$ where n is no. of elements

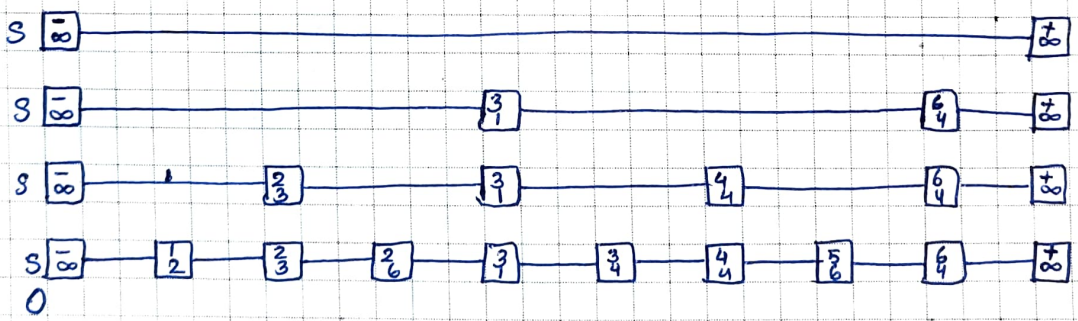
Operations:

Search – starts in sparsest subsequence & navigates down thru successive levels until reaching the full sequence.
The search complexity is O(log n) on average.

Insertion – Involves finding the correct position and updating multiple levels, complexity is O(log n) on average.

Deletion – Involves removing nodes from multiple level and is O(log n) on average.

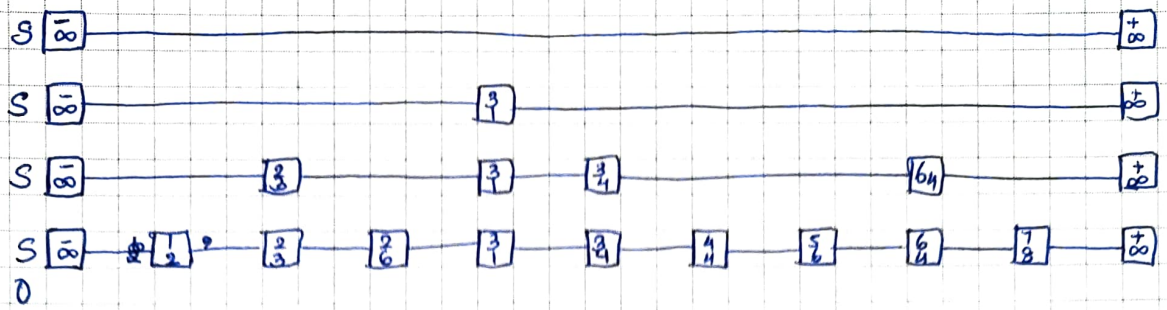
Example of Perfect Skip List



- when we add a new node, our beautifully precise structure might become invalid
  - → We may have to change the level of every node
  - → One option is to move all the elements around
  - → But it takes O(n) time, which is back to where we began.

Is it possible to achieve a net gain?

Example of a Randomized Skip List



Skip list Definition

A skip list for a set S of distinct (key, element) items is a series of lists, $S_0, S_1, \cdots, S_h$ such that:

→ Each list S_i contains special keys $+\infty$ and $-\infty$
→ List S_0 contains the keys of S in non-decreasing order

→ Each list is a subsequence of the previous one, i.e.,
$S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$

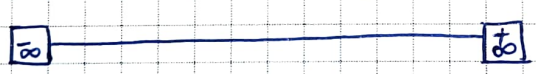
→ list S_h contains only two special keys

Initialisation

A new list is initialized as follows:

→ A node NIL ($+\infty$) is created and its key is set to a value greater than the greatest key that could possibly used in the list.

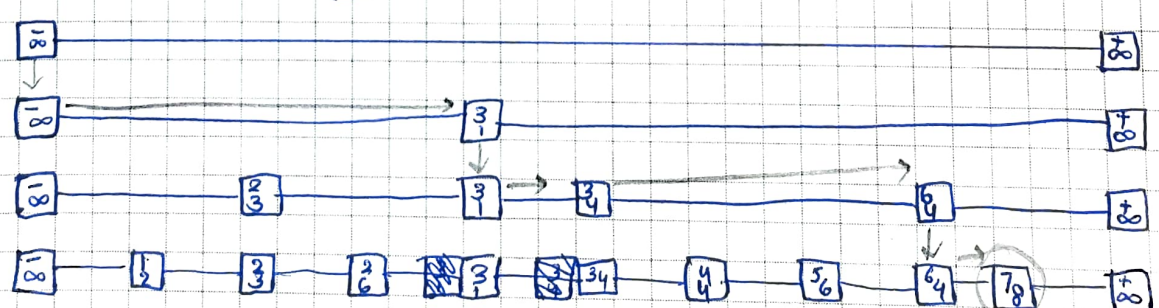
→ Another node NIL ($-\infty$) is created, value set to lowest key that could be used



Search

We search for a key x in a skip list as follows:
- start at the 1st position of the top list
- at the current position p, we compare x with y i.e., key (after (p))
  - $x = y$: return element (after (p))
  - $x > y$: we "scan forward"
  - $x < y$: we "drop down"
- If we try to drop down past the bottom list, we return NO_SUCH_KEY

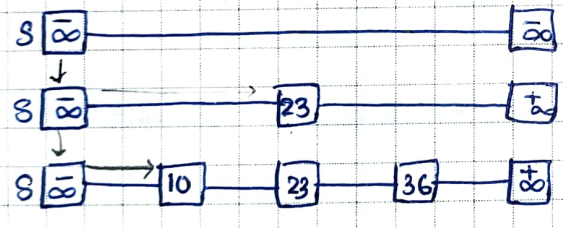
search 78



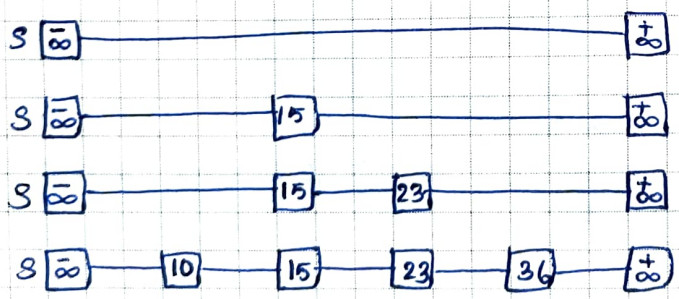
Insertion

- algorithm uses randomization to decide how many references to the new item should be added to the skip list.
- we then insert the new item in this bottom level list at its appropriate position. After inserting the new item at this level we quite honestly "flip a coin"
  - heads, move to higher level, insert at appropriate position & repeat the step
  - tails, we stop right there
- A randomized algorithm performs coin tosses (i.e. uses random bits) to control its execution.
- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions:
  - coins are unbiased
  - tosses are independent
- Worst case running time of a randomized algorithm is LARGE but has low probability
  - basically if you get heads every time.

Say I add 15 and coin comes up "head" 2 times followed by a "tail"



Do a search, & find a spot b/w 10 & 23



"tail"

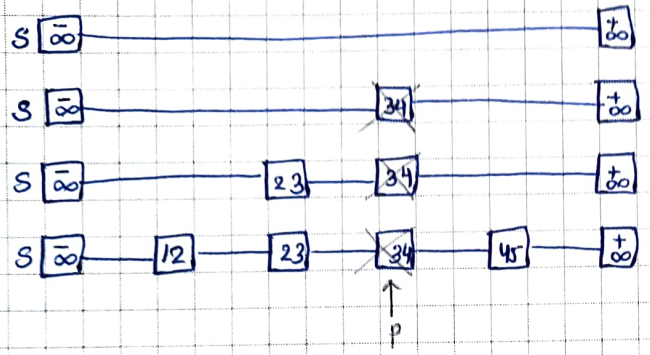
one "head"

one "head"

insert into position

Deletion

- Begin by performing a search for the given key k
- If a position p with key k is not found, then we return NO_SUCH_KEY element
- If a position p with key k is found
  - ↳ on the bottom level
    - ↳ then remove all position above p
- If more than one upper level is empty, remove it



delete 34

① do a search, find the spot b/w 23, 45

② remove all levels of that element, i.e., all positions above p

Q. Starting with an empty skip list, insert these keys, with these "randomly generated" levels.

5 with level 1	6 with level 3	22 with level 2
26 with level 1	21 with level 0	
25 with level 4	3 with level 2	

Note: Drawing of keys in a skip list is determined by the value in the keys only; the levels of the nodes are determined by the random number generator only

CPU scheduling → using queue

① First Come First Serve

- Process that requests the CPU first is allocated the CPU
- Implemented by a simple Queue.
  - ↳ When process enters the ready queue its PCB is linked onto the rear of the queue

PCB
Process Control Block
- data structure used by operating system to store info about a process

- When CPU is free, it is allocated to the process at front of queue
- The running process is then removed from the queue.

② Shortest Job First (Pre-emptive)

- jobs are put into the ready queue as they arrive.
- as a process with short burst time arrives, the existing process is preempted or removed from execution & the shorter job is executed first.

(priority queue is one method)

burst time
↳ total amount of time a process requires on the CPU for execution.
↳ time needed to complete task, excluding any waiting or I/O time.

preemptive
↳ scheduling strategy where a currently running process can be interrupted and moved to a waiting state to allow another process to execute.

③ Shortest Job First (Non-Pre-emptive)

- ~~S~~ process which has shortest burst time is scheduled first
- If 2 processes have the same burst time then "first come first serve" is used to break the tie.
- consider the processes
  A (6s)
  B (20s)
  C (6s)
  D (3s)

① sort: D (3s)
A (6s)
C (6s)
B (20s)
} first come first serve

② time:

D completes at	3s
A	9s (3+6)
C	15s (9+6)
B	35s (15+20)

non-preemptive
↳ processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion

④ Longest Job First (Preemptive)

- similar to shortest job first

- priority is given to process having the largest burst time remaining

⑤ Longest Job First (Non-Preemptive)

- same as 'sjf' but priority is for larger burst time remaining

⑥ Round Robin Scheduling

- The processes are kept in the queue of processes
- New processes are added to rear of simple queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1 time quantum. In this case, the process will release the CPU voluntarily. The scheduler will proceed to next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process is put at the rear of the ready queue. The CPU scheduler will then select the next process in the ready queue.

BINARY TREES

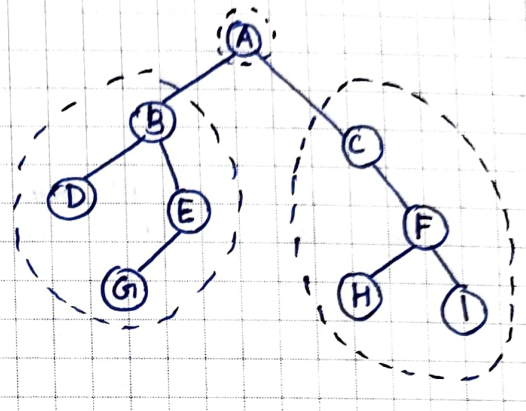
Linear Data Structures

Array
- fixed size :(
  - × expansion
  - × shrinking
- random insert/delete is time consuming

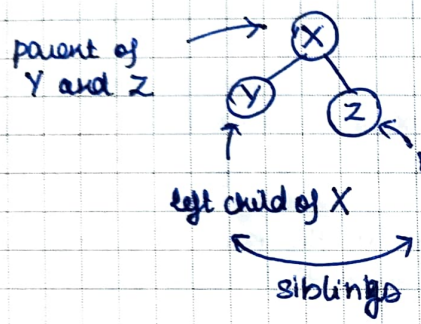
Linked lists
- random access is time consuming

Non linear Data Structure

- finite set of elements that is either empty or is partitioned into 3 subsets
  1. first subset - root - single element
  2. second - left binary tree
  3. third - right binary tree



- Each element of a binary tree is called a node of the tree

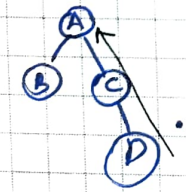


- a node which has no children is called leaf node/ external node.
  - ↳ otherwise non leaf node/ internal node

- a node N_1 is called ancestor of N_2 if
  - → N_1 is parent of N_2 → C is parent of F
  - → N_1 is parent of some other ancestor of N_2 → A is ancestor of F

  and N_2 becomes descendant of N_1
  - ↳ can be left descendent or right

- Level of a node
  - root has level 0; level of any node is +1 than parent.



level of A = 0
B = 1 C = 1
D = 2

- Depth of a tree
  - max. level of any leaf in the tree (path length from deepest leaf to root)

depth = 2

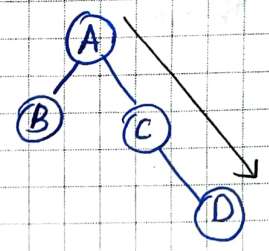
- Depth of node
  - path length from node to the root

depth of A : 0
B : 1
C : 1
D : 2

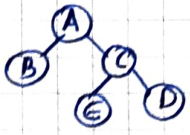
- **Height of tree**: path length from root node to deepest leaf
  height = 2

- **Height of a node**: path length from node to deepest leaf

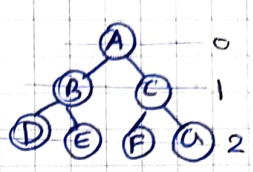
  height of A = 2 B = 0 C = 1 D = 0



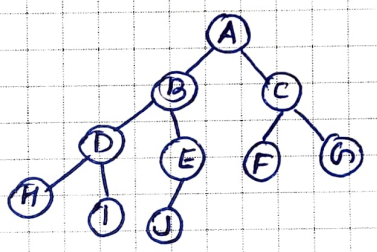
Strictly Binary Tree
→ every node has either 2 or 0 children



Fully Binary Tree
→ all leaves at the same level
→ if depth is d then there are 0 to d levels
→ total no. of nodes
$= 2^0 + 2^1 + \ldots + 2^d$
$= 2^{d+1} - 1$



Complete Binary Tree
→ all levels are completely filled except possibly the lowest one, which is filled from the left.



Properties of Binary Trees

1. every node except root has exactly 1 parent
2. A tree with n nodes has n-1 edges
3. tree consisting of only root node has height of zero
4. total no. of nodes in a full binary tree of depth d $= 2^{(d+1)} - 1$ $\boxed{d \geq 0}$
5. For any non-empty binary tree, if n_0 is the number of leaf nodes and n_2 the nodes of degree 2, then $n_0 = n_2 + 1$

Traversal Terminology:

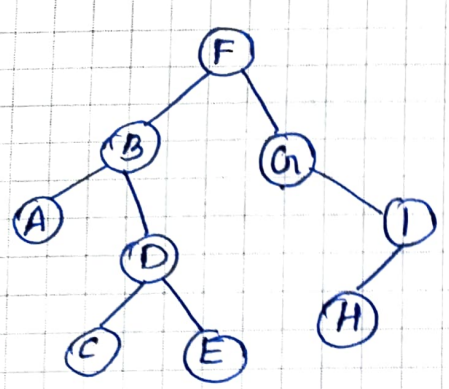
Tasks: V - visiting a node denoted by V
traversing left subtree denoted by L
traversing right subtree denoted by R

VLR, LVR, LRV, VRL, RVL, RLV

Standard Traversals Include: VLR - Preorder
LVR - Inorder
LRV - Postorder

Preorder

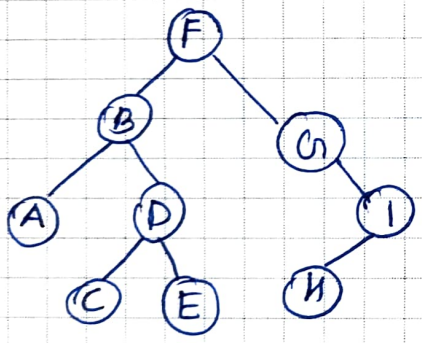
1. Root Node is visited
2. left tree is traversed in pre order
3. Right tree is traversed in pre order



F B A D C E G I H

Inorder

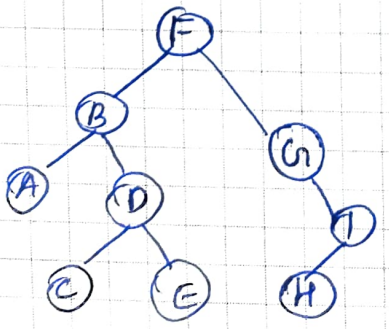
1. left subtree is traversed in inorder
2. root node is visited
3. right subtree is traversed in inorder



A B C D E F G H I

Post order

1. left subtree is traversed in postorder
2. right subtree is traversed in postorder
3. root node is visited.

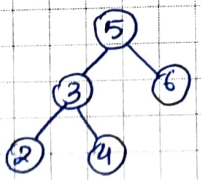


A C E D B H I G F

Iterative Inorder Traversal

atp → at this point

```
iterativeInorder(root)
    s = emptyStack
    current = root
    do{
        while (current != null)
        { travel down left branches as far as possible saving pointer to nodes
          passed in the stack
            push(s, current)
            current = current → left
        } atp left subtree is empty
        poppedNode = pop(s)
        print poppedNode → info        visit node
        current = poppedNode → right   traverse right subtree
    }
    while (!isEmpty(s) or current != null)
```



① current = 5 (push 5)
current = 3 (push 3)
current = 2 (push 2)
current = NULL (push 2)

② poppedNode = 2 (print 2)

③ poppedNode = 3 (print 3)
current = 4

④ 4 in stack
current = NULL
poppedNode = 4 (print 4)
poppedNode = 5 (print 5)

current = 6
current = NULL
poppedNode = 6 (print 6)

2 3 4 5 6

Iterative Preorder Traversal

```
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)
    }
```

Iterative Post Order Traversal

```
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)){
        current = pop(s2)
        print current → info
    }
```

N-ary Tree

- rooted tree in which each node has no more than n children
- a binary tree is n-ary tree with n=2

Representation of trees

1. tree node options

```
struct treenode{
        int info;
        struct treenode *child[MAX];
};
```

where MAX is a constant

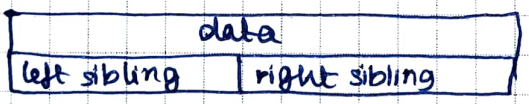
Restrictions with this implementation → a node cannot have more than MAX children, so cannot expand the tree

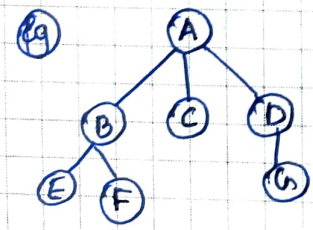
2. "all" the children of a given node are linked and only the oldest child is linked to the parent.
   - a node has link to first child and a link to immediate sibling

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

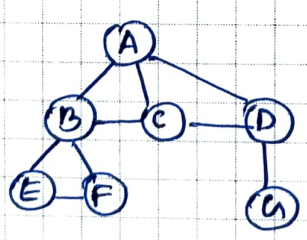
3. Left Child - Right Sibling Representation

- Link all siblings of a node
- delete all links from a node to its children except for the link to its leftmost child
- the left child in binary tree is the node which is the oldest child of the given node in an n-ary tree, and the right ~~tree~~ child is the node to the immediate right to given node on the same level.
  - will NOT have a right subtree.

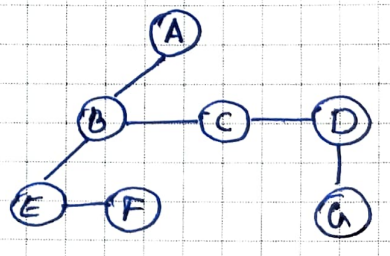




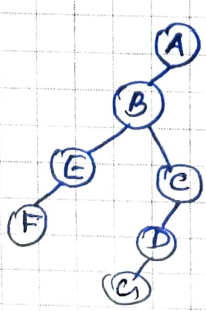
3ary tree



link all siblings of a node



delete all links from a node to its children except for link to leftmost child

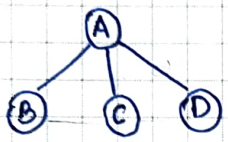


← binary tree

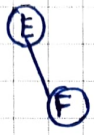
Forest To Binary Tree

- Right child of the root node of every resulting binary tree will be empty. This is bcs the root of the tree we are transforming has no siblings.
- BUT if we have a forest then these can all be transformed into a single binary tree as follows:
  1. first obtain binary tree representation of each of the trees in the forest.
  2. Link all the binary trees together through the right sibling field of the root nodes

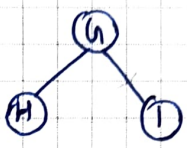
Consider the following forest with 3 trees



T1

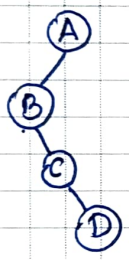


T2

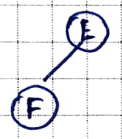


T3

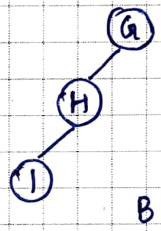
Corresponding Binary Trees



B(T1)

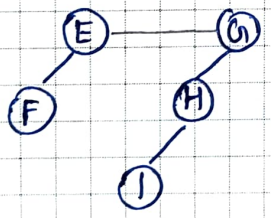


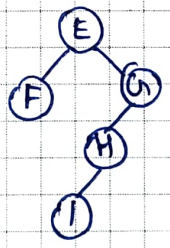
B(T2)



B(T3)

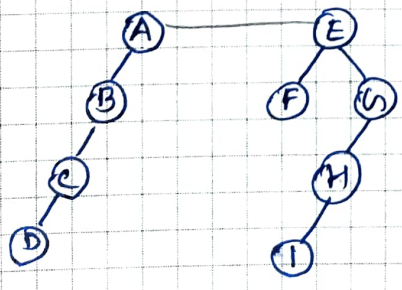
take B(T2), B(T3)

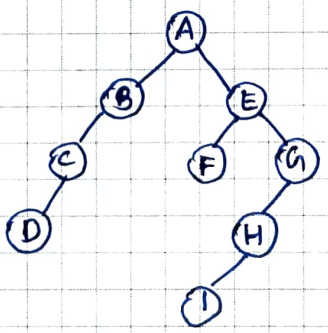




B(T2, T3)

now, B(T1), B(T2, T3)



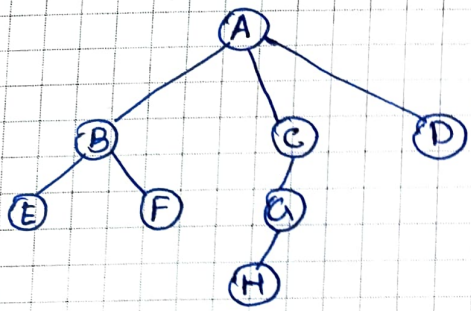


B(T1, T2, T3)

N-ary Tree Traversal

Preorder

1. visit the root of the first tree in the forest
2. traverse in preorder the forest formed by the subtrees of the first tree, if any
3. traverse in preorder the forest formed by the remaining trees in the forest if any

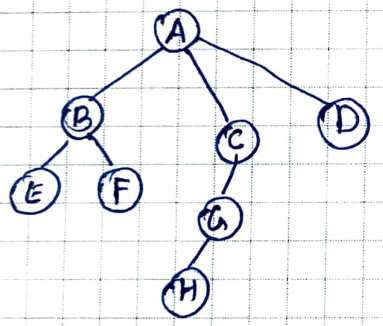


ABEFCGHD

```
void preorder (TREE *root){
    if (root != NULL){
        printf("%d", root→info);
        preorder (root→child);
        preorder (root→sibling);
    }
}
```

Inorder

1. Traverse in inorder the forest formed by the subtrees of the first tree, if any
2. Visit the root of the first tree in the forest
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any

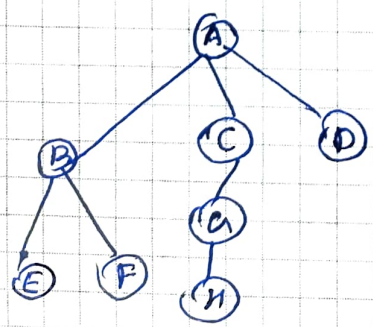


EFBHGCDA

```
void inorder (TREE* root){
    if (root != null){
        inorder (root→child);
        printf ("%d ", root→info);
        inorder (root→sibling);
    }
}
```

Postorder

1. Traverse in postorder the forest formed by the subtrees of the first tree, if any
2. Traverse in postorder the forest formed by the remaining trees in the forest, if any
3. Visit the root of the first tree in the forest



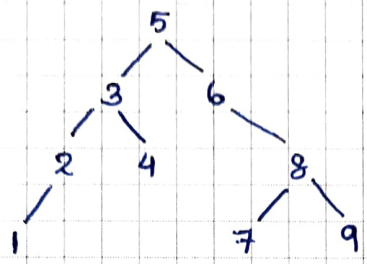
FEHGDCBA

```
void postorder (TREE *root){
    if (root != null){
        postorder (root→child);
        postorder (root→sibling);
        printf ("%d ", root→info);
    }
}
```

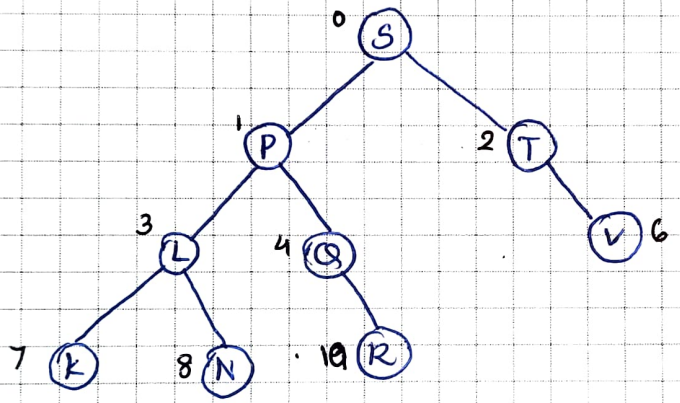
BINARY SEARCH TREE (BST)

1. all the elements in the left subtree of a node n are less than the contents of node n
2. all the elements in the right subtree of a node n are greater than or equal to the contents of node n

eg: BST of nodes entered in order 5, 3, 6, 4, 2, 8, 1, 7, 9



ARRAY IMPLEMENTATION



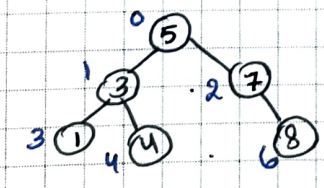
array:

S	P	T	L	Q	...	V	K	N	...	R
0	1	2	3	4	5	6	7	8	9	10

```
typedef struct tree_array {
        int info;
        int used;
} NODE;

NODE bst[100];
```

→ information I am trying to store
→ if it is used, like above 5, 9 are unused

If I start indexing from 0



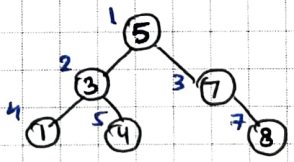
root position: $i = 0$
left child position: $2i+1$
right child position: $2i+2$

info	5	3	7	1	4		8
used	1	1	1	1	1	0	1
position	0	1	2	3	4	5	6

5 3 7 8 1 4

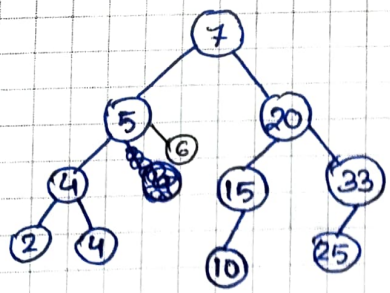
5 → 0
3 → 2(0)+1 = 1
7 → 2(0)+2 = 2
8 → 2(0)+2 = 2 → occupied
↳ 2(2)+2 = 6
1 → 2(0)+1 = 1 → occupied
↳ 2(1)+1 = 3
4 → 2(0)+1 = 1 → occupied
↳ 2(1)+2 = 4

If I start from 1



root position: $i = 1$
left child position: $2i$
right child position: $2i+1$

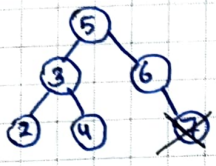
q. input: 7, 20, 5, 15, 10, 4, 4, 33, 2, 25, 6 . Make a BST.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	7	5	20	4	6	15	33	2	4			10		25
used	1	1	1	1	01	1	0	1	1	0	0	1	0	1

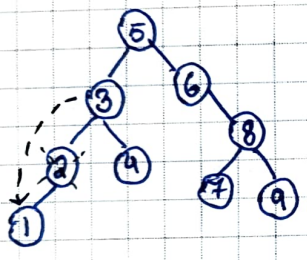
DELETION OF A NODE

Case 1: Node with no child



if I want to delete 7,
1. set parents left child field to point to NULL
2. free memory allocated to 7

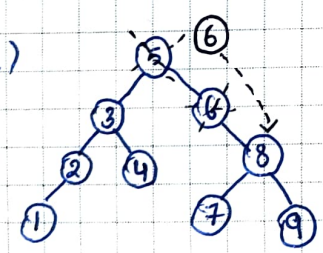
Case 2: Node with one child



if I want to delete 2
1. set parents left child field to point to its only child
2. free memory allocated to 2

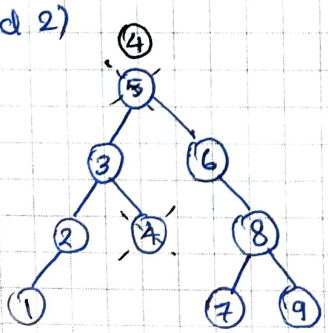
Case 3: Node with 2 children

(method 1)



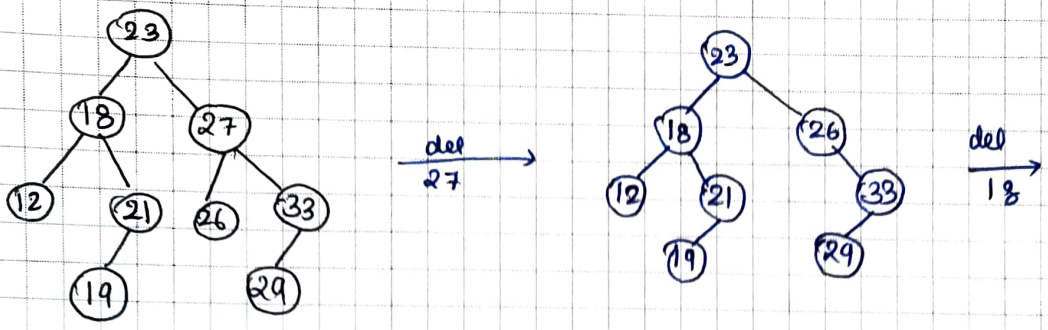
If I want to delete 5
1. replace 5 with inorder successor & delete that successor
2. It usually changes to a different case after replacing

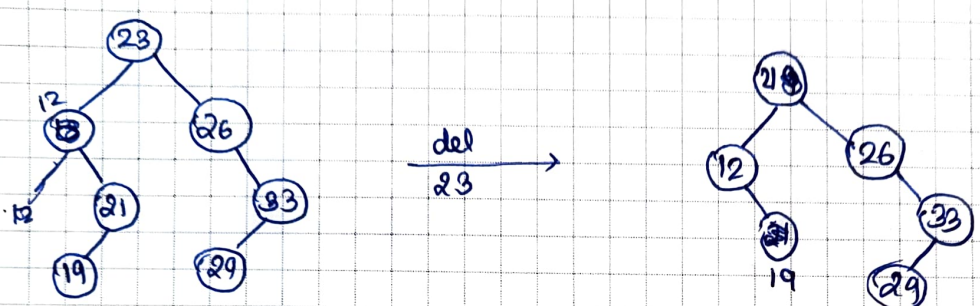
(method 2)



If I want to delete 5
1. replace 5 with inorder predecessor and delete that inorder predecessor

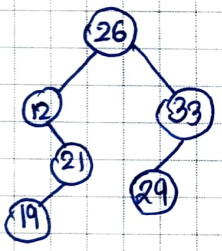
9. delete 27, 18, 23





(method - 2)

del 23



(method - 1)

Insertion:

```
typedef
    void insert(tree *obj, int ele){
        node *new_node = (node*)malloc(sizeof(node));
        new_node->data = ele;
        new_node->left = NULL;
        new_node->right = NULL;

        if(obj->root == NULL){
            obj->root = new_node;
        }
        else{
            node *p = obj->root;
            node *q = NULL;

            while(p!=NULL){
                q=p;
                if(new_node->data >= p->data){
                    p=p->right;
                }
                else{
                    p=p->left;
                }
            }

            if(new_node->data >= q->data){
                q->right = new_node;
            }
            else{
                q->left = new_node;
            }
        }
    }
```

Traversals with Recursion

```
void Inorder(node *p){
    if(p!=NULL){
        Inorder(p->left);
        printf("%d", p->data);
        Inorder(p->right);
    }
}

void InorderTrav(tree *obj){
    if(obj->root==NULL){
        printf("empty");
    }
    else{
        Inorder(obj->root);
    }
}
```

```
void Postorder(node*p){
    if(p!=NULL){
        Postorder(p->left);
        Postorder(p->right);
        printf("%d", p->data);
    }
}

void PostorderTrav(tree*obj){
    if(obj->root==NULL){
        printf("empty");
    }
    else{
        Postorder(obj->root);
    }
}
```

```
void Preorder(node *p){
        if(p!=NULL){
                printf("%d", p->data);
                Preorder(p->left);
                Preorder(p->right);
        }
}
```

```
void PreorderTrav(tree *obj){
        if(obj->root==NULL){
                printf("empty");
        }
        else{
                Preorder(obj->root);
        }
}
```

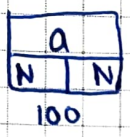
```
void levelOrder(tree *obj){
        if(obj->root==NULL){
                printf("empty");
        }
        NODE* queue[1000];
        int front = rear = -1;

        queue[++rear] = pt->root;

        while(front<rear){
                NODE* curr = queue[++front];
                printf("%d", curr->info);

                if(curr->left!=NULL){
                        queue[++rear] = curr->left;
                }
                if(curr->right!=NULL){
                        queue[++rear] = curr->right;
                }
        }
}
```

* Also do the traversals iteratively

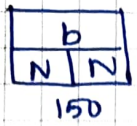
Expression Tree

Postfix Expression : abc*+

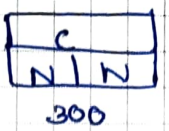
Symbol = a

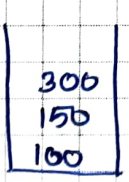


Symbol = b

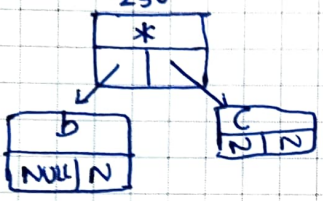


Symbol = c

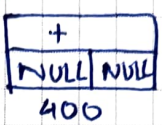


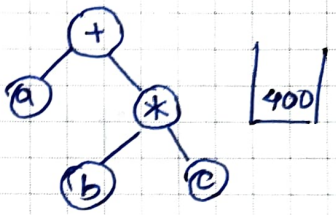


Symol = *



Symbol = +





Algorithm:

1. Scan the postfix expression till the end, one symbol at a time
   - Create a new node, with symbol as info and left and right link as NULL
   - If symbol is an operand, push address of node to stack
   - If symbol is an operator
     - pop address from stack and make it right child of new node
     - pop address from stack and make it left child of new node
     - now push address of new node to stack
2. Finally, stack has only element which is the address of the root of the expression tree.

Code:

```
typedef struct node{
        char data;
        struct node *left;
        struct node *right;
}node;

int isOperator(char ch){
        switch(ch){
                case '+';
                case '-';
                case '*';
                case '/'; return 1;
                default: return 0;
        }
}
```

```
void push (node **stk, int *top, node *temp){
        ++(*top);
        stk[*top] = temp;
}

node *pop(node **stk, int *top) {
        node *x;
        x = stk[*top];
        --(*top);
        return x;
}

node* create(char* exp){
        int i=0;
        char ch;
        node *temp;
        node *stk[100];
        int top=-1;

        while(exp[i] != '\0') {
                ch = exp[i];
                temp = (node*) malloc (sizeof(node));
                temp->data = ch;
                temp->left = temp->right = NULL;

                if (isOperator(ch)) {
                        temp->right = pop(stk, &top);
                        temp->left = pop(stk, &top);
                        push(stk, &top, temp);
                }
                else {
                        push(stk, &top, temp);
                }
                i++;
        }
        pop(stk, &top);
}

void inorder(node *t){
        if (t != NULL){
                inorder(t->left);
                printf("%c", t->data);
                inorder(t->right);
        }
}

int eval(node *t){
        int x;
        switch(t->data){
                case '+': return (eval(t->left) + eval(t->right));
                case '-':                          -
                case '*':                          *
                case '/':                          /
                default: printf("%c= ", t->data); scanf("%d", &x); return x;
        }
}
```

THREADED BINARY TREE

HEAP

A heap can be defined as a binary tree with keys assigned to its nodes, provided the following 2 conditions are met:

1. The tree's shape requirement
   → binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost nodes may be missing.

2. The parental dominance requirement
   → the key at each node is greater than or equal to the keys at its children.

Properties:

1. height = $\log_2 n$ where n = no. of nodes

A heap can be implemented as an array by recording its elements in the top-down, left to right fashion. It is convenient to store the heaps elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap.

In such a representation:

a) The parental node keys will be in the first $(n/2)$ positions of the array, while the leaf keys will occupy the last $(n/2)$ positions.

b) The children of a key in the array's parental position i $(1 <= i <= n/2)$ will be in positions 2i and 2i+1, and correspondingly, the parent of a key in position i $(2 <= i <= n)$ will be in position $(i/2)$.

Bottom Up Heap construction

1. first value becomes root
2. second value becomes left child, third value becomes right child
3. the heap rules are applied to the tree (max or min)

Top Down Heap construction

- Incrementally inserts each element

The main difference lies in array management.

Top Down Heap Construction

1. Insertion-based approach: elements are inserted one by one into the heap & heap property is restored each time using heapify-up

Steps: 1. Start with an empty heap
2. Insert each element of the array sequentially
3. Apply heapify up (bubble up) from the newly added element until the heap property is restored.

eg: [4, 10, 3, 5, 1]

1. [ ]
2. [4]
3. Insert 10 → [4, 10] ⟶ heapify → [10, 4]
4. Insert 3 ⟶ [10, 4, 3] → no change
5. Insert 5 ⟶ [10, 4, 3, 5] ⟶ heapify ⟶ [10, 5, 3, 4]
6. Insert 1 ⟶ [10, 5, 3, 4, 1] ⟶ no change

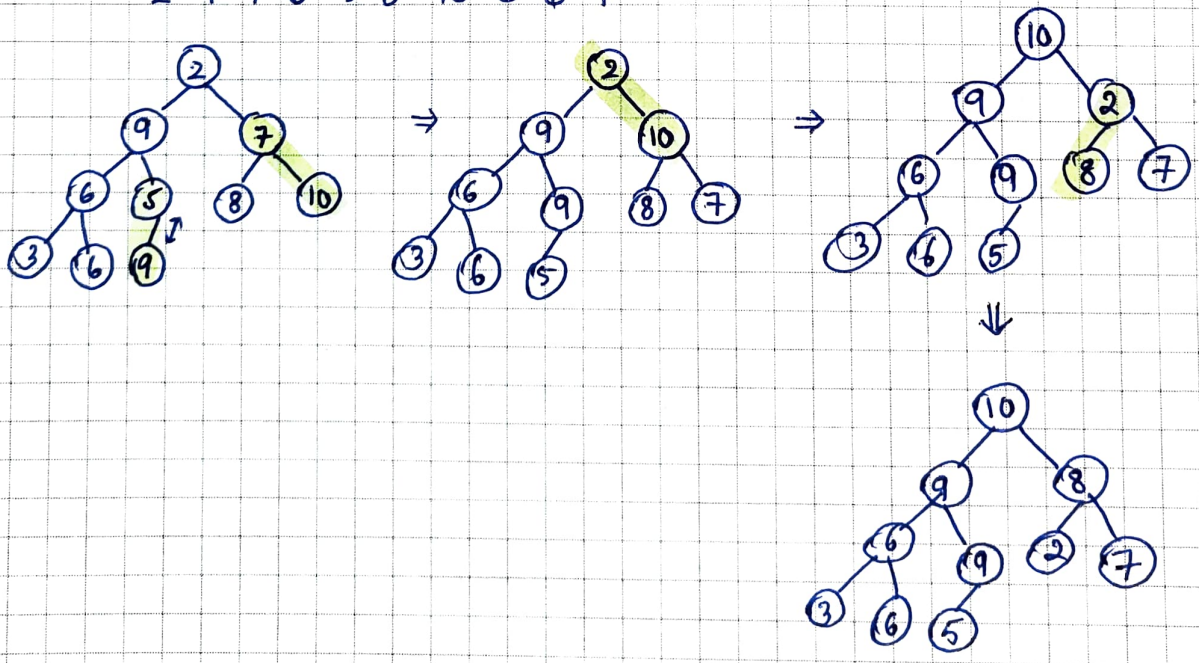
Bottom Up Heap Construction

1. Heapify-based approach: A complete binary tree is first formed from the array, then adjusted to satisfy the heap property

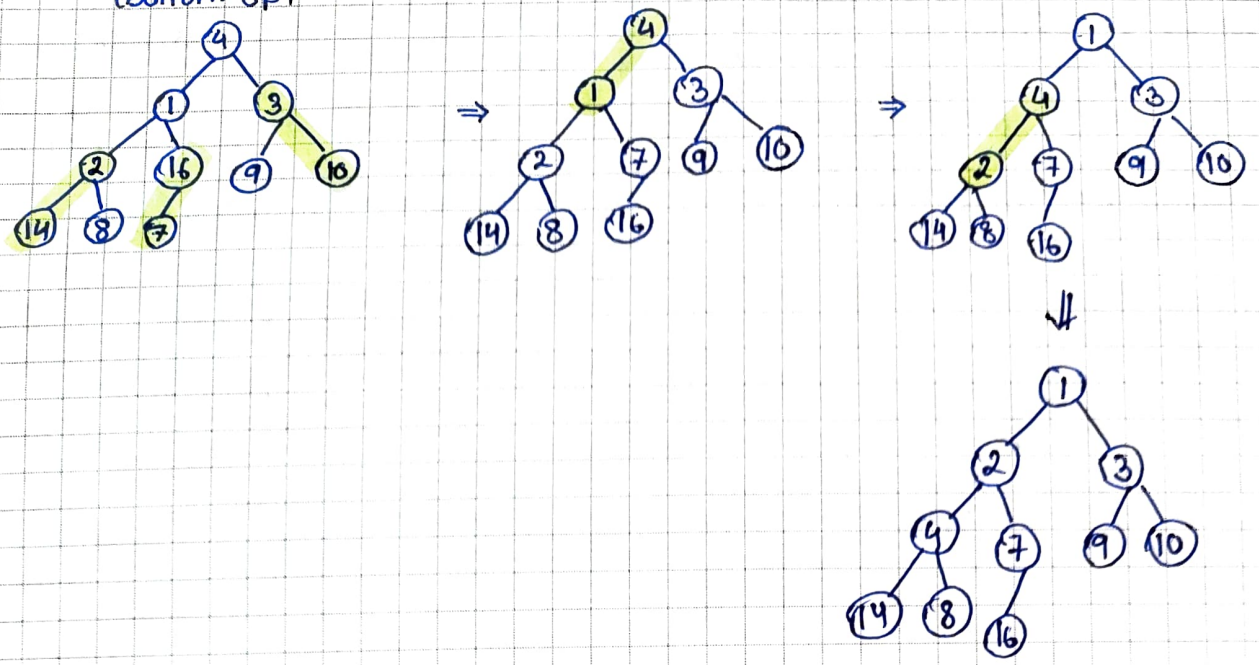
Steps: 1. start with the entire array as a complete binary tree
2. begin heapifying from the last non-leaf node ($n/2 - 1$)
3. apply heapify-down from this node up to the root

eg 1) construct max heap for the given array of n elements: (bottom-up)

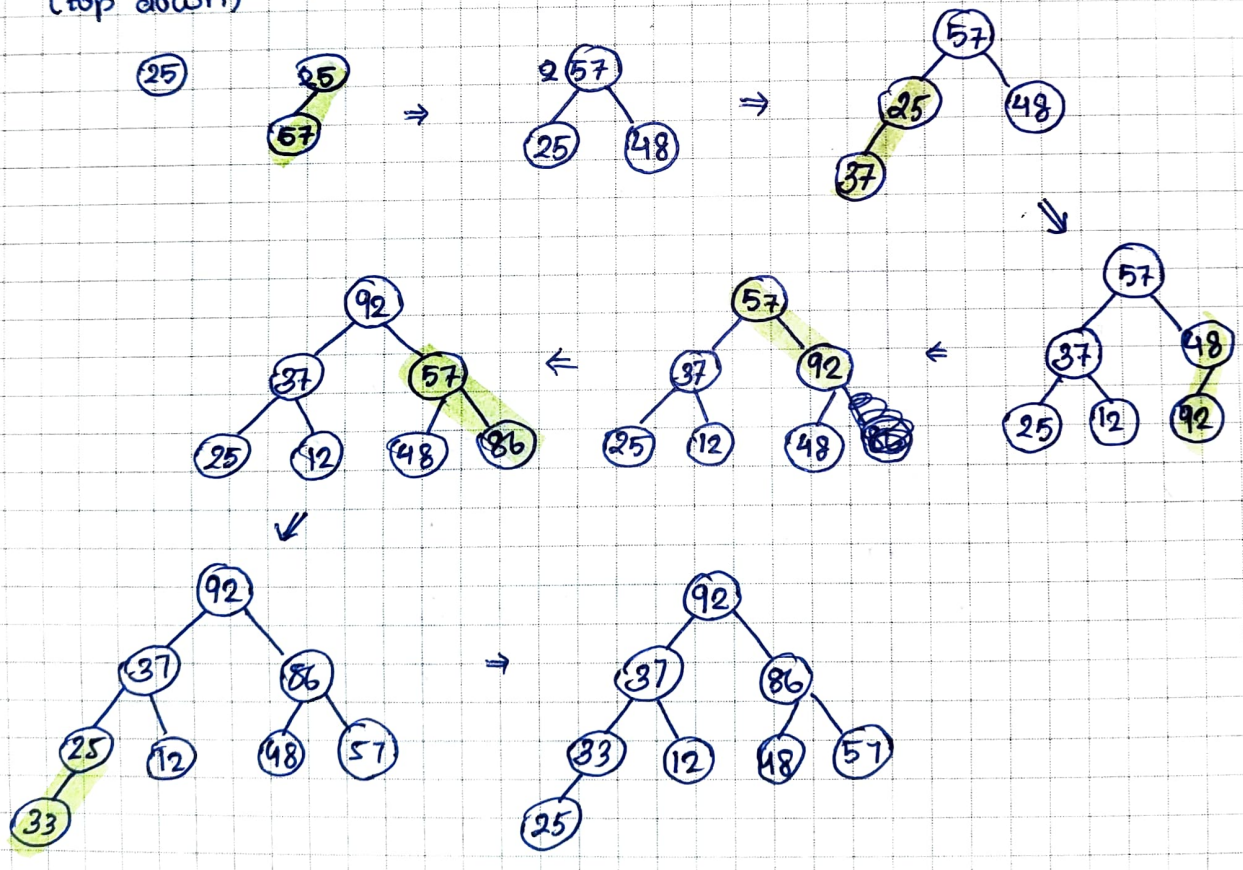
2 9 7 6 5 8 10 3 6 9



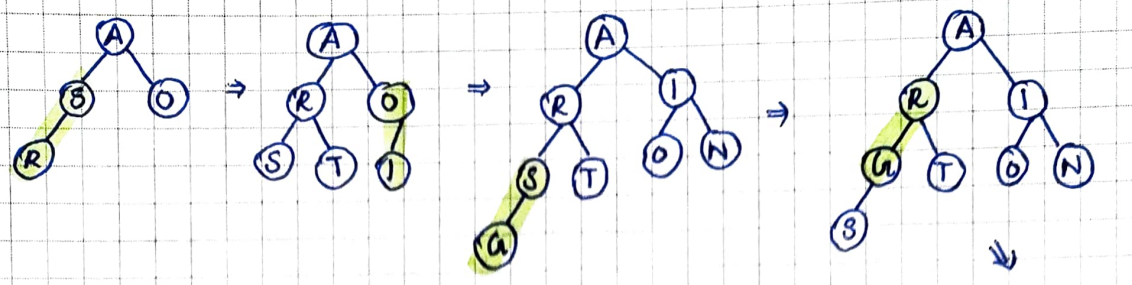
eg 2) Construct min heap: 4 1 3 2 16 9 10 14 8 7
(bottom up)



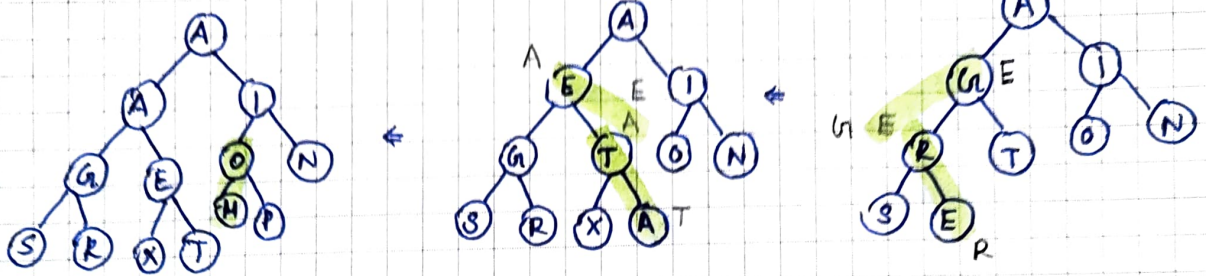
eg 3) Construct max heap: 25 57 48 37 12 92 86 33
(top down)

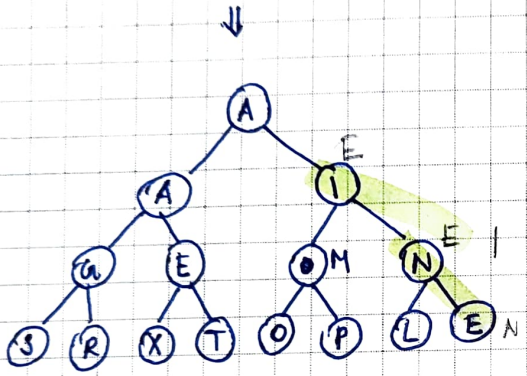


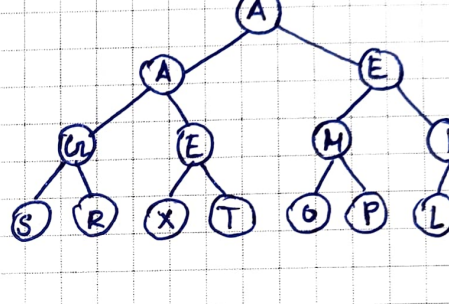
eg 4) construct a min heap: A, S, O, R, T, I, N, G, E, X, A, M, P, L, E (top down)



A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Y Z

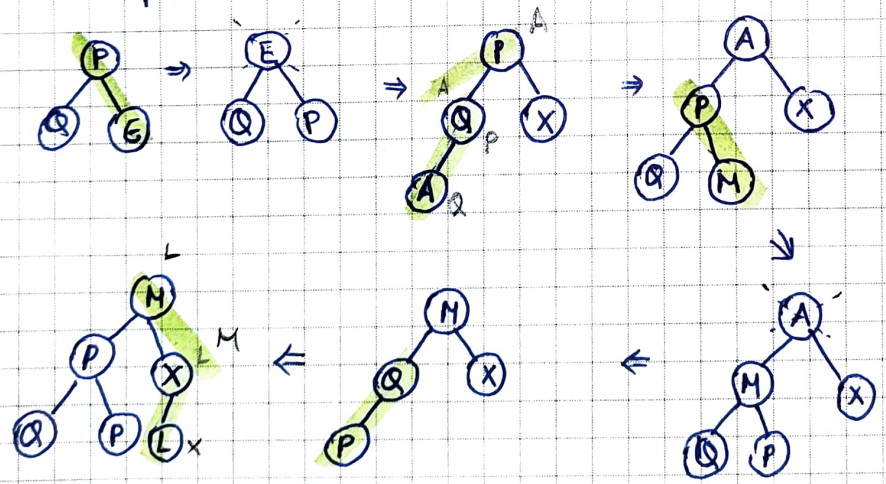


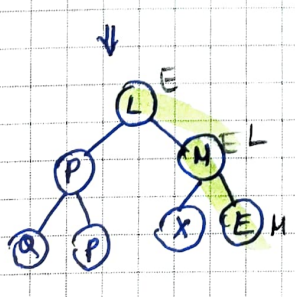




eg 5) min heap

insert P
Q
E
remove min
insert X
A
M
remove min
insert P
L
E
remove min





BALANCED TREES

- height of tree is $\log(n)$

AVL TREES

balance factor = height of left subtree − height of right subtree

in an AVL tree every node's balance factor should be 0, −1, +1

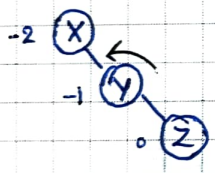
If not balanced AVL,
Rotation is performed on the nodes whose balance factor is +2, −2 after an insertion/deletion operation

The rotation is performed on the closest newly inserted leaf

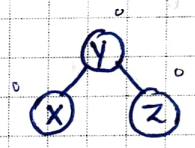
- Single
  - Left
  - Right
- Double
  - Left-Right
  - Right-Left

If tree becomes unbalanced, we use rotations to rebalance. If node X is inserted in balanced BST, we need to find the youngest ancestor which becomes unbalanced

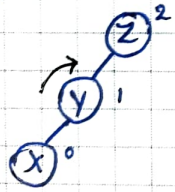
① XYZ



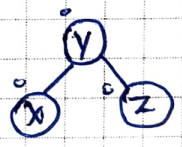
if tree is unbalanced towards the right, we do left rotation
↓
anticlockwise



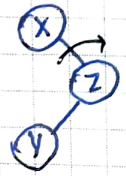
② ZYX



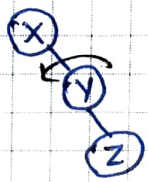
if tree is unbalanced towards the left, we do right rotation
↓
clockwise

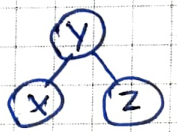


③ XZY

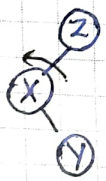


it was balanced till Y was inserted
Y is is inserted right of X
and left of Z
↳ so right left

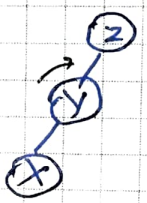


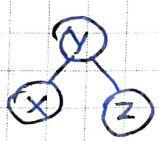


④ ZXY

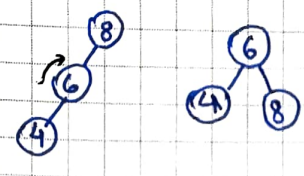


it was balanced till Y was inserted
Y is inserted to left of Z
and right of X
↳ so left right

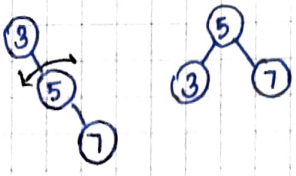




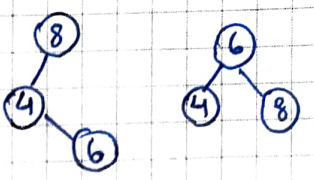
① 8, 6, 4 (R)



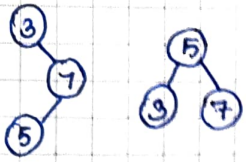
② 3, 5, 7 (L)



③ 8, 4, 6 (LR)



④ 3, 7, 5 (RL)

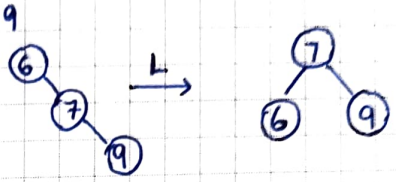


⑤ Insert elements 6, 7, 9, 3, 2, 5, 8 into an AVL

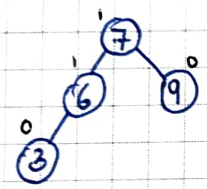
Insert 6

Insert 7

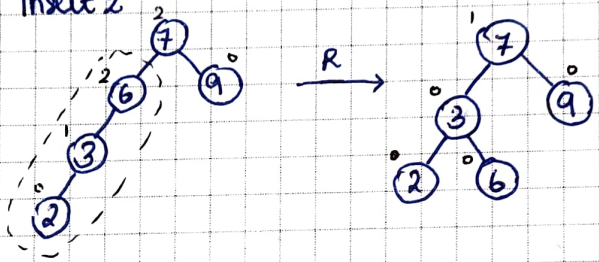
Insert 9



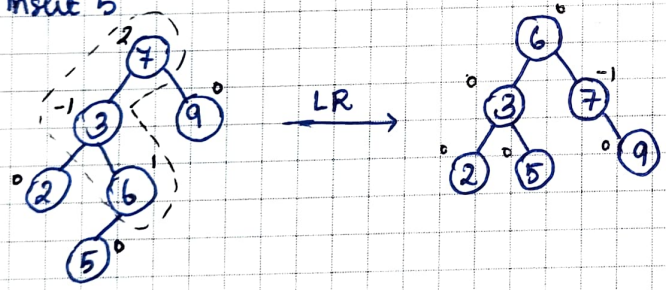
Insert 3



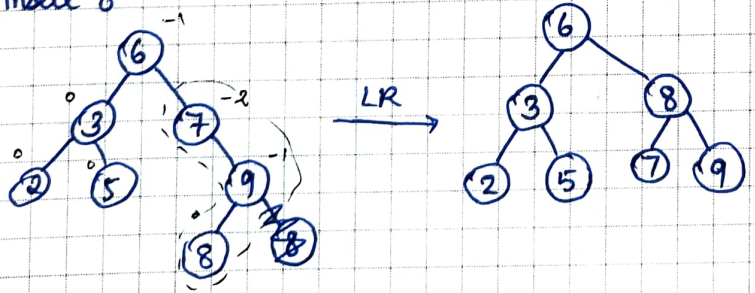
Insert 2



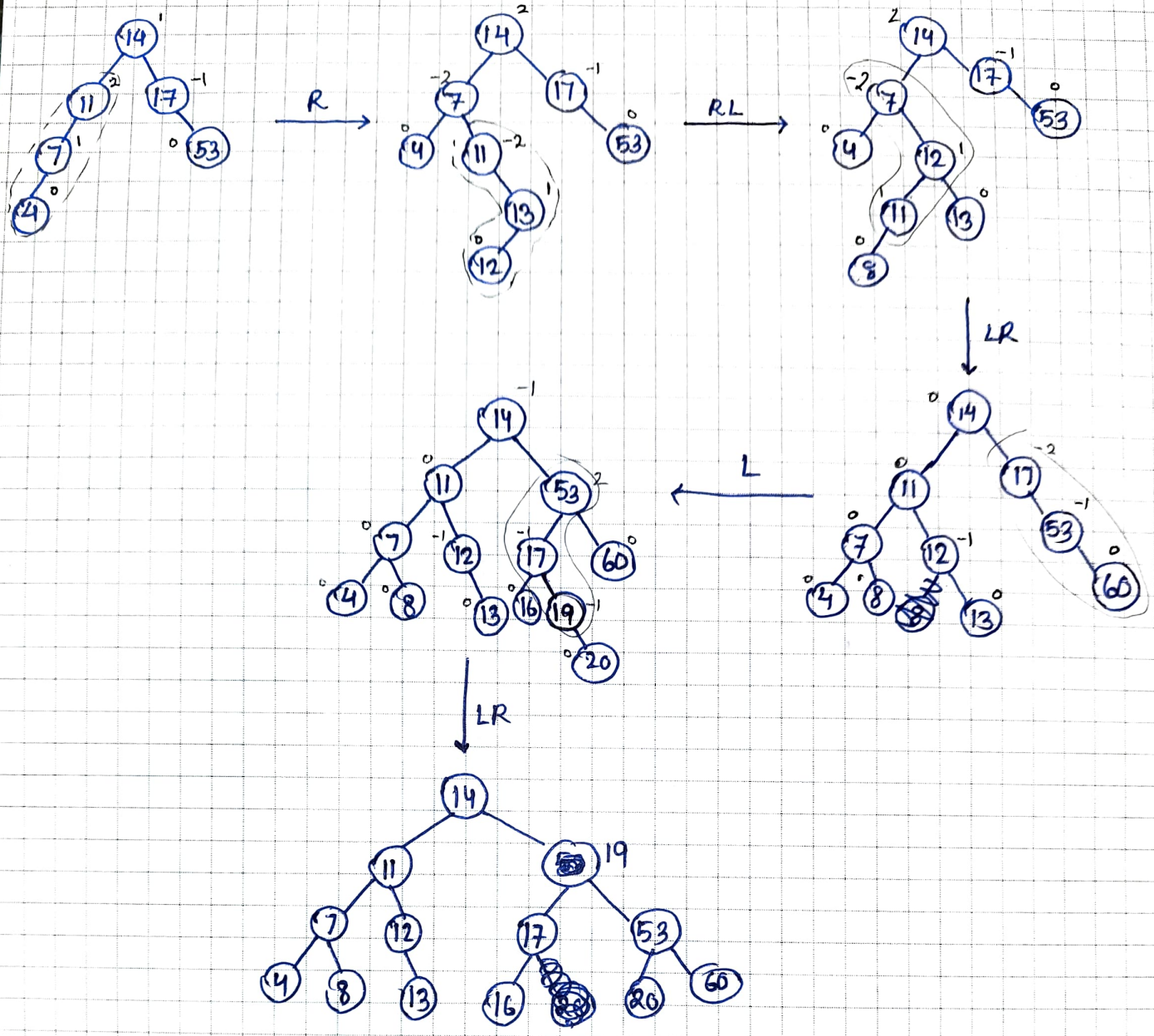
Insert 5



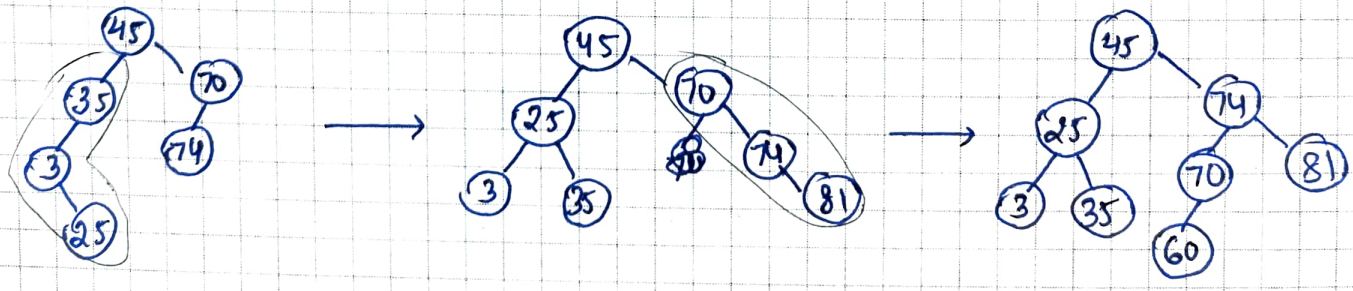
Insert 8



⑥ Insert 14, 17, 11, 7, 53, 4, 18, 12, 8, 60, 19, 16, 20

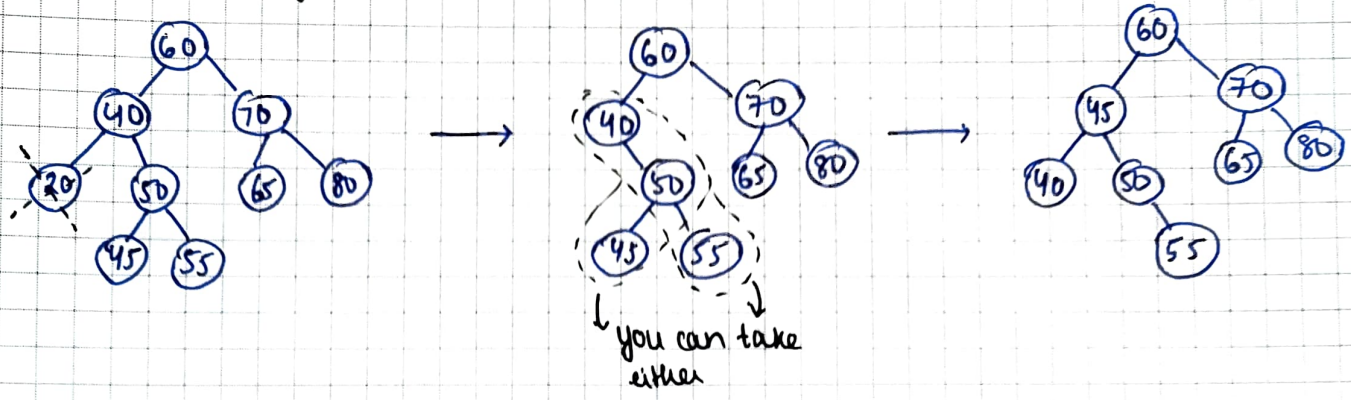


⑦ 45, 70, 35, 3, 74, 25, 81, 60

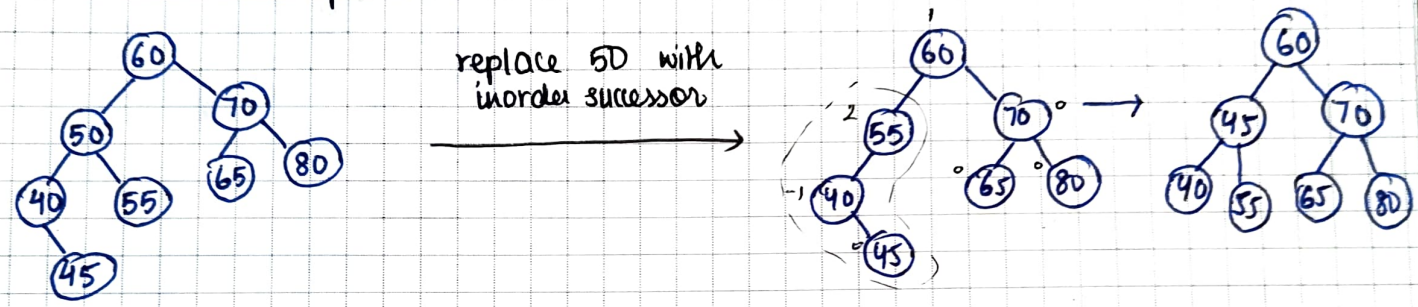


Deletion in AVL Trees

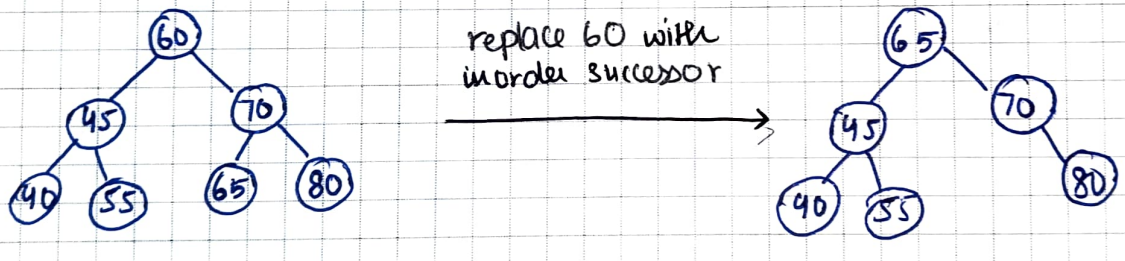
Delete 20 (leaf node - no children)



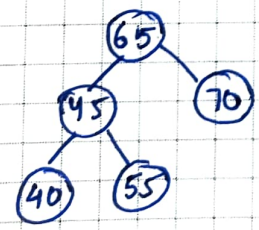
Delete 50 (parent node)



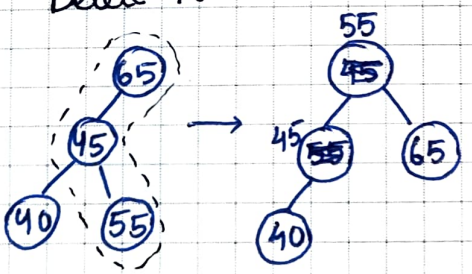
Delete 60 (root node)



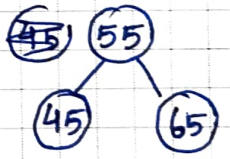
Delete 80



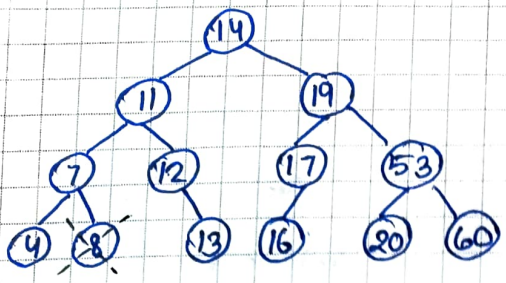
Delete 70



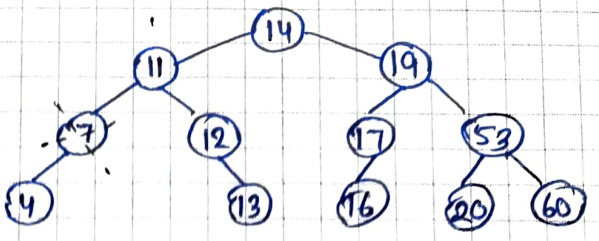
Delete 40



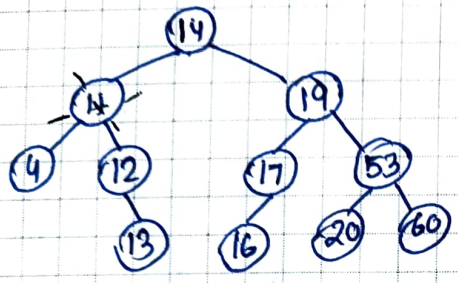
Delete 8, 7, 11, 14, 17



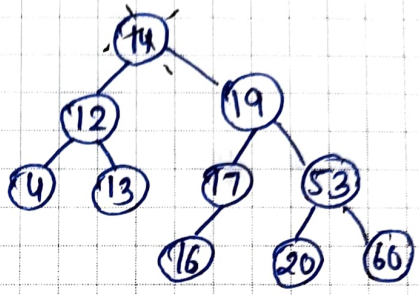
del 8



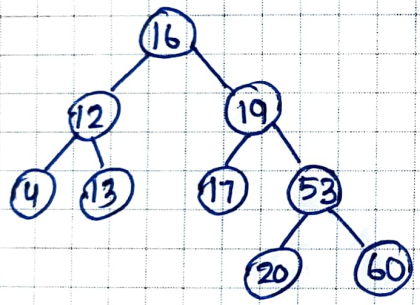
del 7



del 11



del 14



del 17

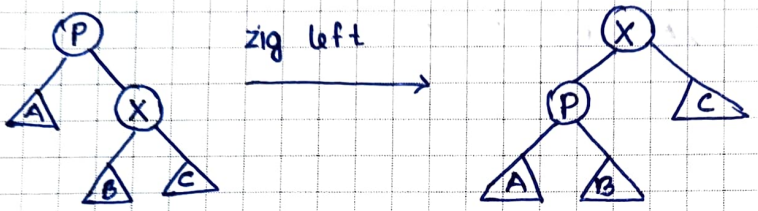
SPLAY TREE

- roughly balanced BST
- additional property: recently accessed elements are quick to access again
- main application: hospital records.

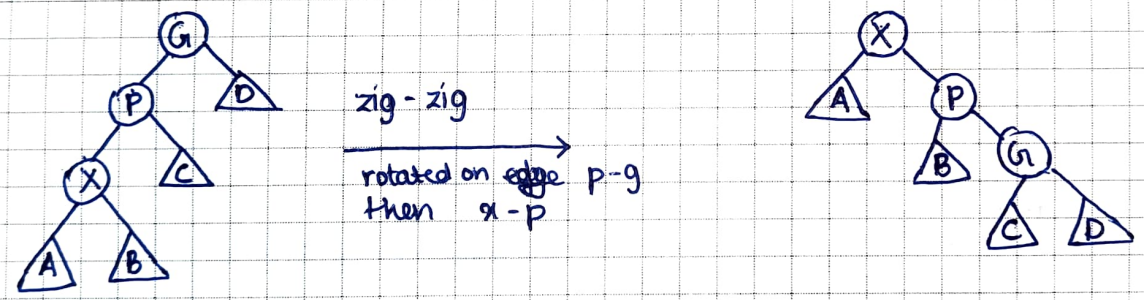
Splaying

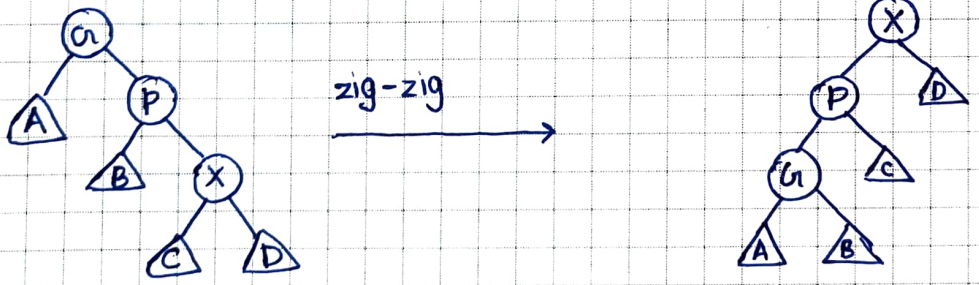
X - node being searched
P - parent of X
G - grandparent of X

Zig (P is root)

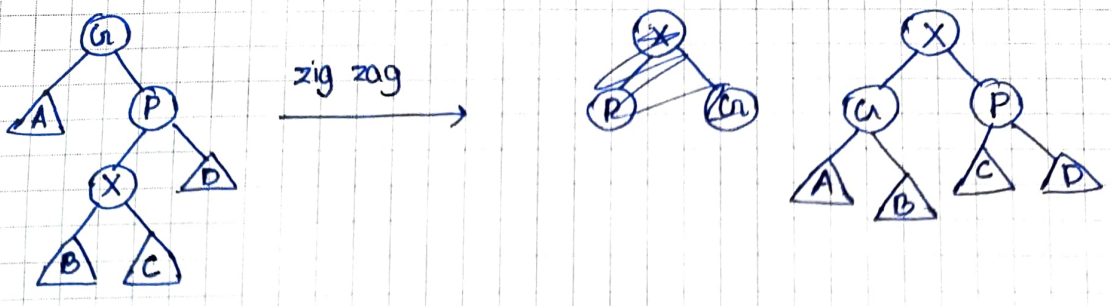


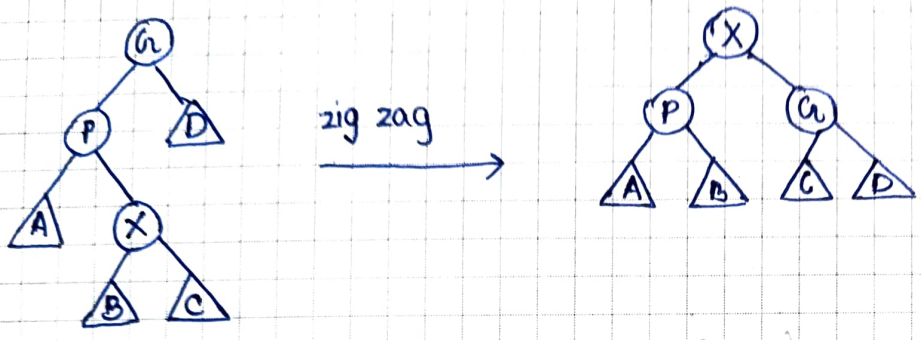
Zig-Zig (P is not root; X, P are both left or both right children)





Zig Zag (p is not root: x is left, p is right or vice versa)





Insertion

- to insert a value X into a splay tree
- Insert X as with normal bst
- when an item is inserted, a splay is performed
- as a result, the newly inserted node X becomes the root of the tree

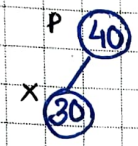
Deletion

To delete a node X, use the same method as bst

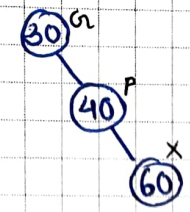
- If X has 2 children
  1. Swap its value with inorder predecessor or successor
  2. Remove that node instead

After deletion we splay parent of the removed node to the top of the tree

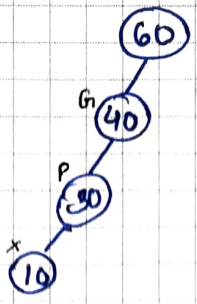
① Insert 40, 30, 60, 10, 35, 50 into a splay tree



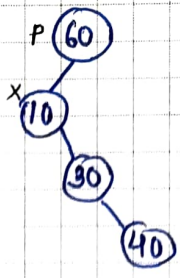
Insert 40, 30

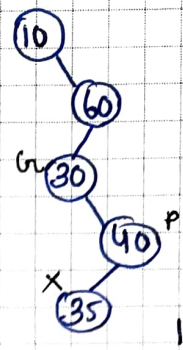


Insert 60

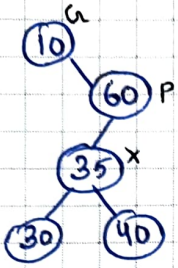


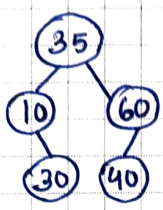
Insert 10

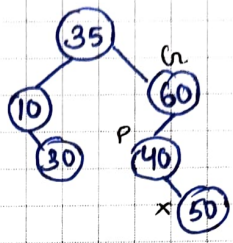




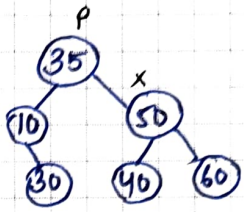
Insert 35

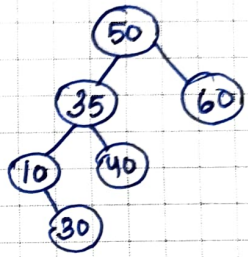




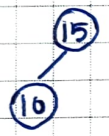


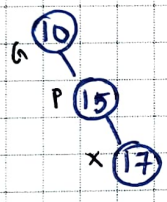
Insert 50

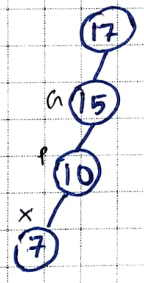


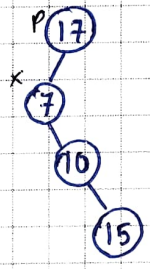


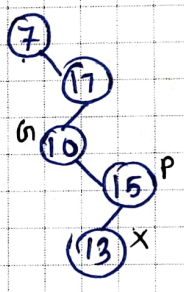
② Insert 15, 10, 17, 7, 13, 16

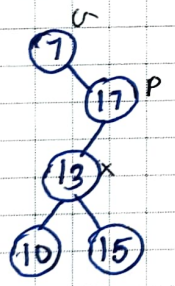


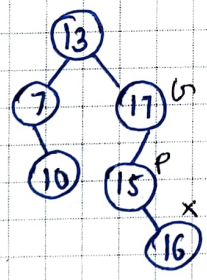




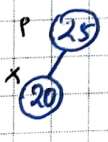


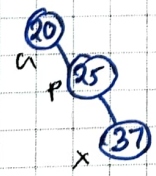


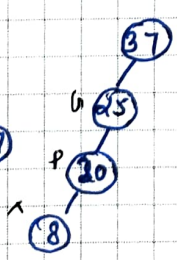


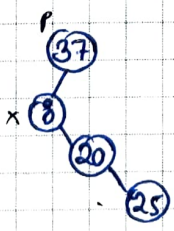


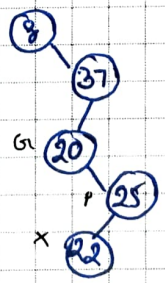
③ Insert 25, 20, 37, 8, 22, 16

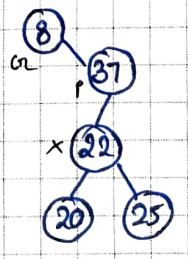


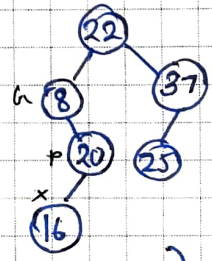


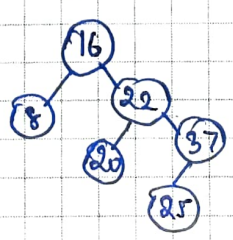


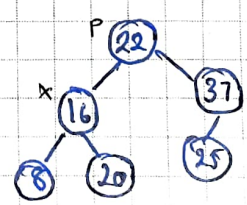






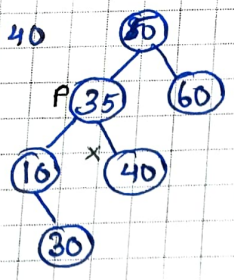


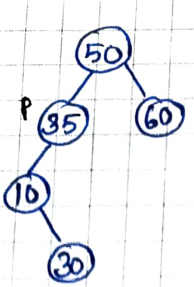


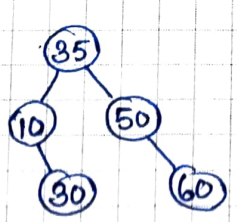


Delete 40, 10, 35, 70

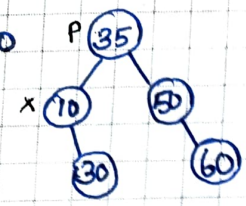
del 40

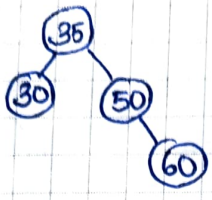






del 10

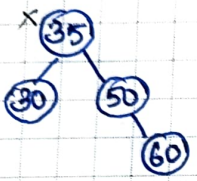


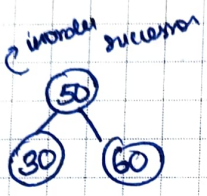


del 70

if node doesn't exist, add where it would be & splay its parent

del 35

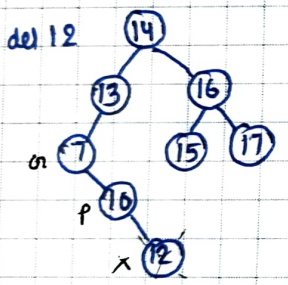


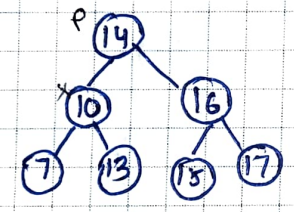


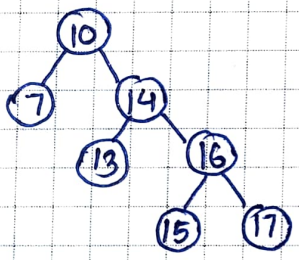
no parent node, so no need to splay

Delete 12, 14, 16, 20, 17

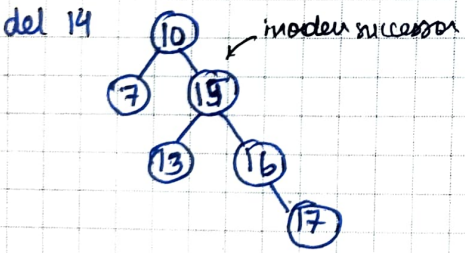
del 12



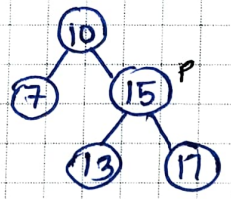


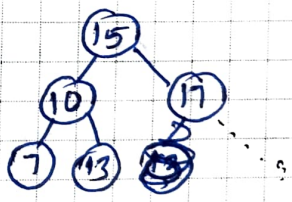


del 14

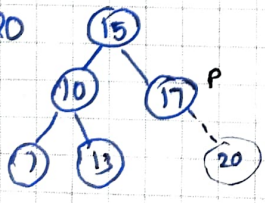


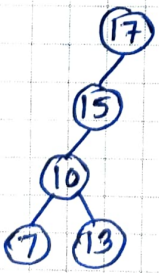
del 16



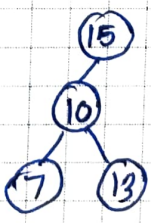


del 20

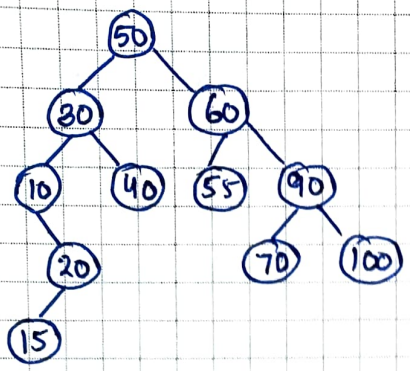


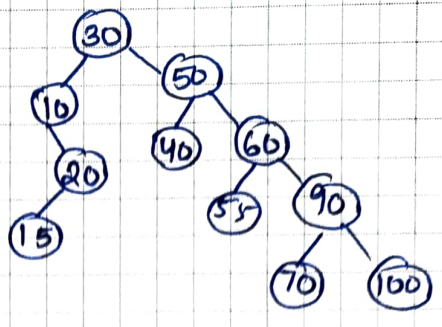


del 17



Search 30 → splay 30





HEAPS (heapify function)

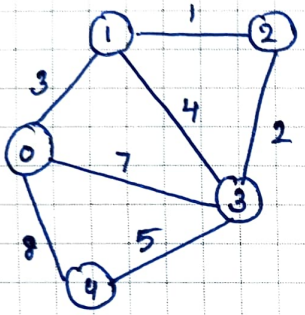
```
void heapify (int arr[], int sz, int index) {
        int largest = index;
        int left = (2*index) +1;
        int right = (2*index)+2;

        if ( left < sz && arr[right] > arr[largest]) {
                largest = right;
        }

        if (right < sz && arr[right] > arr [largest]) {
                largest = right;
        }

        if (largest != index) {
                int temp = arr[index];
                arr[index] = arr [largest];
                arr[ largest) = temp;

                heapify (arr, sz, largest);
        }
}
```

GRAPHS

- data structure that consists of set of vertices & a set of edges that relate the node to each others.

$G = (V, E)$ V: finite non-empty set of vertices
E: set of edges

- weighted graph : graph where each edge has a numerical value called weight



adjacent node: n is adjacent to m if there is an edge from m to n

m is adjacent to n

path: sequence of vertices that connect two nodes in a graph.

cycle: a path from a node to itself.

acyclic: graph with no cycles ; a directed acyclic graph is called dag

incident: a node n is incident to an edge x, if node is one of the two nodes the edge connects.

degree: the degree of vertex i is the no. of edges incident on vertex i

degree (A) = 2 degree (D) = 1

Indegree - no. of edges incident to i
Out-degree - no. of edges incident from i

Directed Graph

no. of possible pairs in m vertex graph: $m(m-1)$
no. of edges = $m(m-1)$

no. of edges $\leq m(m-1)$

Undirected graph

no. of pairs $m(m-1)$
no. of edges $\leq \frac{m(m-1)}{2}$

Graph Representation.

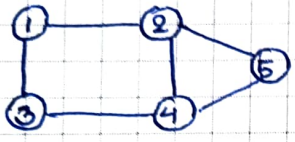
Depending on the density of edges, ease of use and types of operation performed, graphs can be represented by:

1. Adjacency Matrix - 2D Array
2. Adjacency List - Linked List

Adjacency Matrix

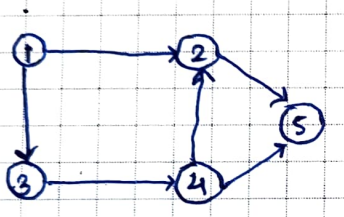
adjacency matrix = $n \times n$ matrix M, graph of n vertices/nodes

M[i][j] = 1 if (i,j) is an edge = 0, no edge b/w the vertices (i,j)

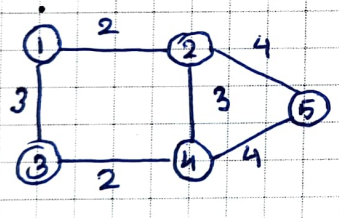


	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	1
3	1	0	0	1	0
4	0	1	1	0	1
5	0	1	0	1	0

- for undirected graph, M is symmetric
- assum no edge from node to itself, so diagonal elements has value 0.

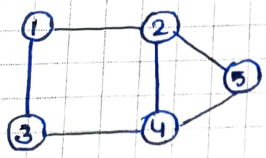


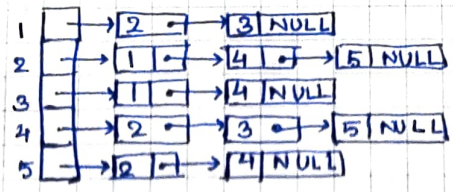
	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	0	1	0	0	1
5	0	0	0	0	0

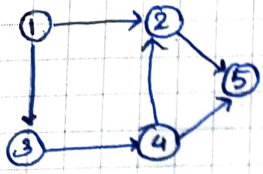


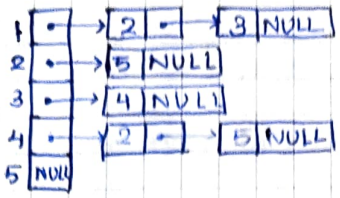
	1	2	3	4	5
1	0	2	3	0	0
2	2	0	0	3	4
3	3	0	0	2	0
4	0	3	2	0	4
5	0	4	0	4	0

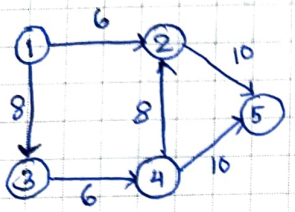
Adjacency List

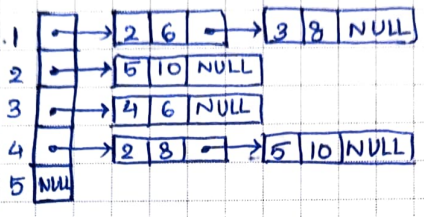






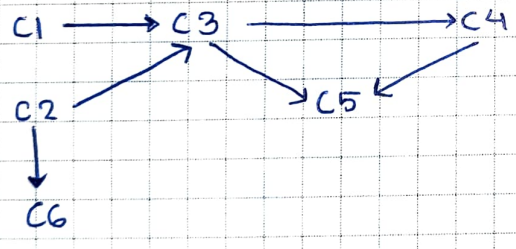




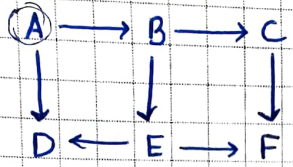


DEPTH FIRST SEARCH (DFS)

- visits all the nodes related to one neighbour before visiting the other neighbour & visiting the other neighbours & its related nodes.
- uses stack behaviour, hence implemented using recursive algorithm.



push(s)	v = adj(s[top])	visited nodes	pop()
C1	C3	C1	–
C3	C4 C5	C1 C3	–
C4	C5	C1 C3 C4	–
C5	–	C1 C3 C4 C5	C5
C4	–	C1 C3 C4 C5	C4
C3	–	C1 C3 C4 C5	C3
C1	–	C1 C3 C4 C5	C1
C2	C3 C6	C1 C3 C4 C5 C2	–
C6	–	C1 C3 C4 C5 C2 C6	C6
C2	–	C1 C3 C4 C5 C2 C6	C2



push(s)	v=adj[s[top]]	visited nodes	pop()
A	B D	A	-
B	C E	A B	-
C	F	A B C	-
F	-	A B C F	F
C	E	A B C F	C
E	D F	A B C F E	-
D	-	A B C F E D	D
E	-	A B C F E D	E
B	-	A B C F E D	B
A	-	A B C F E D	A

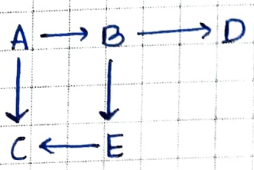
```
#include <stdio.h>   #include<stdlib.h>

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

node *a[100]
int visited[100];

void insert (int i, int j) {
        node *temp = (node*)malloc(sizeof(node));
        temp->data = j;
        temp-> next = a[i];
        a[i] = temp;
}

void CreateGraph (int n) {
        for(int i=1; i<=n; i++){
                a[i] = NULL;
                visited[i] = 0;
        }
        int i,j;
        while (1) {
                printf ("enter src & dest: ");
                scanf ("%d %d", &i, &j);
                if (i==0 && j==0){
                        break;
                }
                else {
                   insert (i,j);
                }
}
```

```
void DFS (int v){
        visited[v] = 1;
        printf ("%d ", v);
        node *p;
        int w;

        for ( p= a[v]; p!= NULL; p=p->next)
        {
            w = p->data;
            if (visited[w]==0){
                    DFS(w);
            }
        }
}

int main(){
        int n;
        scanf("%d",&n);
        CreateGraph(n);

        int v;
        scanf("%d", &v);
        DFS(v);
}
```

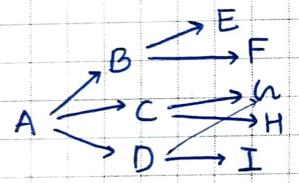
BREADTH FIRST SEARCH (BFS)

- bfs(v)

```
//visits recursively all the unvisited vertices connected to vertex v by a path
while the queue is not empty
    for each vertex w in v adjacent to front vertex do
        if w is marked with 0
            mark w as visited
            add w to queue
            remove the front vertex from the queue.
```



Enqueue(s)	v = adj(s.front())	visited nodes	Dequeue()
A	B C	A	-
B	E D	A B	A
C	-	A B C	B
D	-	A B C D	C
E	-	A B C D E	D
-	-	A B C D E	E



Enqueue (s)	v=adj(s.front())	visited nodes	Dequeue()
A	B C D	A	-
B	E F	A B	A
C	G H	A B C	B
D	G I	A B C D	C
E	-	A B C DE	D
F	-	A B C D E F	E
G	-	A B C D E F G	F
H	-	A B C D E F G H	G
I	-	A B C D E F G H I	H
-	-	A B C D E F G H I	I

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

#define M 100

typedef struct queue{
        int data[M];
        int f;
        int r;
}queue;

void init (queue *q) {
        q->f = -1;
        q->r = -1;
}

int empty (queue *q){
        return q->f == -1;
}

void enqueue (queue *q, int val){
        if(q->r == M-1){
                return;
        }
        if (q->f == -1){
                q->f = 0;
        }
        q->data[++(q->r)] = val;
}

int dequeue (queue *q ){
        if (empty(q)){
                return -1;
        }
        int d = q->data[q->f];
        if (q->f == q->r){
                q->f = q->r = -1;
        }
        else{
                q->f++;
        }
        return d;
}
```

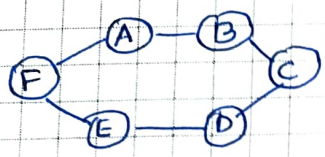
```c
int BFS (int arr[M][M], int src, int dest, int n){
        int visited[M] = {0};
        int path[M];

        for(int i=0; i<n; i++){
                path[i] = 10000;
        }
        path[src] = 0;

        if(src == dest){
                return 0;
        }

        queue q;
        init(&q);
        enqueue(&q, src);
        visited[src]=1;

        while(!empty(&q)){
                int node = dequeue(&q);

                for(int i=0; i<n; i++){
                        if(arr[node][i] == 1 && !visited[i]){
                                visited[i]=1;
                                path[i] = path[node]+1;
                                enqueue(&q, i);

                                if(i == dest){
                                        return path[i];
                                }
                        }
                }
        }
        return -1;
}

int main(){
        int n,m,src,dest;
        scanf("%d %d %d %d", &n,&m,&src,&dest);
        int graph[M][M] = {0};

        for(int=0; i<m; i++){
                int u,v;
                scanf("%d %d", &u,&v);
                graph[u][v] = 1;
                graph[v][u] = 1;
        }
        int res = BFS(graph, src, dest, n);

}
```

NETWORK TOPOLOGY

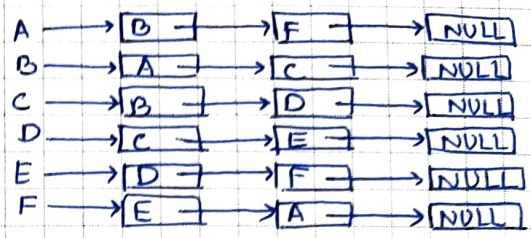
Topology - order in which nodes and edges are arranged in the network

2 types of topology
1. Physical
2. Logical

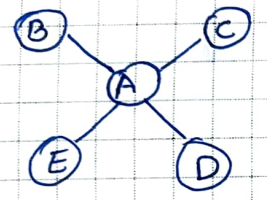
1. Ring Topology



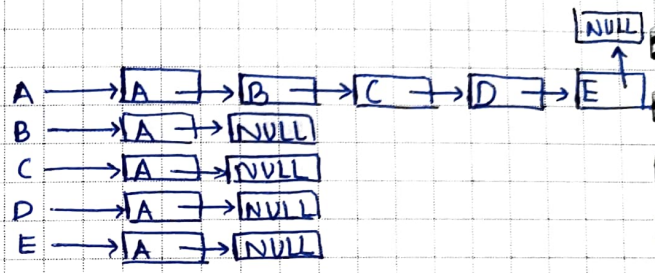
	A	B	C	D	E	F
A	0	1	0	0	0	1
B	1	0	1	0	0	0
C	0	1	0	1	0	0
D	0	0	1	0	1	0
E	0	0	0	1	0	1
F	1	0	0	0	1	0



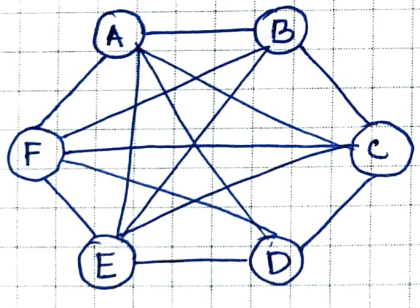
2. Star Topology



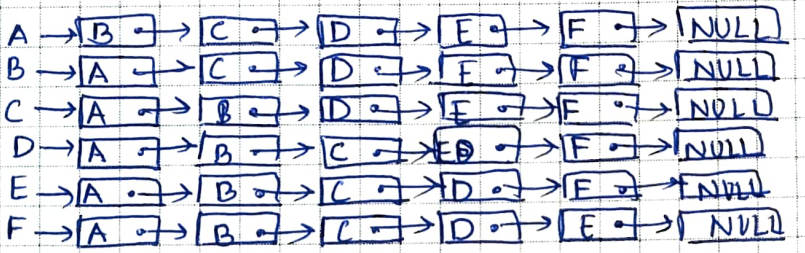
	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	0	0
D	1	0	0	0	0
E	1	0	0	0	0



3. Mesh Topology



	A	B	C	D	E	F
A	0	1	1	1	1	1
B	1	0	1	1	1	1
C	1	1	0	1	1	1
D	1	1	1	0	1	1
E	1	1	1	1	0	1
F	1	1	1	1	1	0



4. Bus Topology

	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	0	1	0	1
D	0	0	1	0

CONNECTIVITY

- connectivity - ability to reach any node in the graph from any other node -

Undirected Graph

Connected

- for any pairs vertices, there should be a sequence of edges connecting them.

1. pick any node
2. perform DFS/BFS from that node
3. if all nodes are visited by the end of the traversal, the graph is connected.
4. If some nodes are not visited, the graph is disconnected.

Disconnected

- if there exists at least one pair of nodes such that no path exists between them.

Directed Graph

Strongly Connected

- for 2 nodes u & v, there exists a path from u-v and v-u

1. perform DFS/BFS starting from an arbitrary node to see if all nodes are reachable.
2. Then, reverse the graph & perform DFS/BFS from the same node.
3. if all nodes are reachable in both the original and the reversed graph, the graph is strongly connected.

Weakly Connected:

Ignoring directions, if there's atleast one edge.

HASHING

- hash table stores key and value pairs in a list that is accessible through its index.
- Collision: The phenomenon when 2 or more keys generate the same hash
- Collision Resolution: Open addressing (Closed Hashing)
  Seperate Chaining (Open Hashing)
- Open Addressing
  - handles collisions by storing all data in the hash table itself and then seeking out availability in the next spot created by the Algorithm
  1. Linear Probing 2. Quadratic Probing 3. Double Hashing
- Seperate Chaining
  - handles collision by making every hash table cell point to linked lists of records with identical hash function values
- Rehashing
  - increase the table size when the load factor becomes more than a predefined value (say 0.75) and hash all the keys present in the table again & store in the new table of increased size.

Load factor = no. of filled records / total capacity

Linear Probing

$h(key) = (h(key) + i) \% tableSize$

1. Insert 7, 20, 41, 31, 18, 8, 9 into a table size of 7. Use LP to resolve collision. Use key % size as hash function.

$h(7) = 7 \% 7 = 0$

$h(20) = 20 \% 7 = 6$

$h(41) = 41 \% 7 = 6$
$(6+1) \% 7 = 0$
$(6+2) \% 7 = 1$

$h(31) = 31 \% 7 = 3$

$h(18) = 18 \% 7 = 4$

$h(8) = 8 \% 7 = 1$
$(1+1) \% 7 = 2$

$h(9) = 9 \% 7 = 2$
$(2+1) \% 7 = 3$
$(2+2) \% 7 = 4$
$(2+3) \% 7 = 5$

0	7
1	41
2	8
3	31
4	18
5	9
6	20

2. 22, 9, 5, 18, 14, 28, 30, 19 into a hash table of size 10 with hash function (1+key) %10 & LP to resolve collision.

$h(22) = (23) \% 10 = 3$
$h(9) = 10 \% 10 = 0$

$h(5) = 6 \% 10 = 6$

$h(18) = 19 \% 10 = 9$

$h(14) = 15 \% 10 = 5$

$h(28) = 29 \% 10 = 9$
$(9+1) \% 10 = 0$
$(9+2) \% 10 = 1$

$h(30) = 31 \% 10 = 1$
$(1+1) \% 10 = 2$

$h(19) = 20 \% 10 = 0$
$(0+1) \% 10 = 1$
$(0+2) \% 10 = 2$
$(0+3) \% 10 = 3$
$(0+4) \% 10 = 4$

0	9
1	28
2	30
3	22
4	19
5	14
6	10
7	
8	
9	18

3. Assume the following hash table was created using LP as a collision resolution technique.

0	1	2	3	4	5	6	7	8
	46	10	3		68	23	14	15

if $h(k) = k \%$ size in which order the elements have been added to the hash table.

(a) 23, 68, 3, 15, 14, 46, 10
(b) 48, 68, 10, 14, 23, 3, 15
(c) 46, 68, 23, 14, 15, 3, 10
(d) 15, 68, 23, 3, 14, 46, 10

4. Same question as (3)

0	1	2	3	4	5	6	7	8	9
		62	13	44	72	56	23		

(a) 56, 44, 62, 13, 72, 23
(b) 44, 62, 72, 23, 56, 13
(c) 44, 62, 23, 72, 56, 13

5. 1, 3, 8, 10 into a hash table of size 7 with hash function (3k+4) %7 & use LP to resolve collision.

$h(1) = (3(1)+4) \% 7 = 0$

$h(3) = (3(3)+4) \% 7 = 13 \% 7 = 6$

$h(8) = (3(8)+4) \% 7 = 0$
$(0+1) \% 7$

$h(10) = (3(10)+4) \% 7 = 34 \% 7 = 6$
$(6+1) \% 7 = 0$
$(6+2) \% 7 = 1$
$(6+3) \% 7 = 2$

0	1
1	8
2	10
3	
4	
5	
6	3

```
typedef struct ele{
        int key;
        char name[50];
        int marked;
}ele;

void insert(ele *hashTab, int tabSz, int key, char name[], int *nor){
        int index;
        if(nor==tabSz){
                printf("table is full\n");
        }
        else{
                index = key % tabSz;
                while(hTab[index].marked != 0){
                        index = (index+1) % tabSz;
                }
                hTab[index].key = key;
                hTab[index].marked = 1;
                strcpy(hTab[index].name, name);
        }
}

void delete(ele *hashTab, int tabSz, int key, char name[], int *nor){
        int index;
        if(nor==tabSz){
                printf("no records");
        }
        else{
                index = index % tabSz;
                ~~while(hTab[index]~~
                int i=0;
                while(i<*nor){
                        if(hashTab[index].marked==1){
                                if(hashTab[index].key==key){
                                        hashTab[index].marked=0;
                                        (*nor)--;
                                }
                                i++;
                        }
                        index = (index+1)% tabSz;
                }
        }
        printf("Deletion complete\n");
}
```

Quadratic Probing

1. 7, 20, 41, 31, 18, 8, 9 into a hash table of size 7 used k%.tabSz as hash function, use QP

h(7) = 7%.7 = 0
h(20) = 20%.7 = 6
h(41) = 41%.7 = 6
(6+1²)%.7 = 0
(6+2²)%.7 = 3

h(31) = 31%.7 = 3
(3+1²)%.7 = 4

h(18) = 18%.7 = 4
(4+1²)%.7 = 5

h(8) = 8%.7 = 1

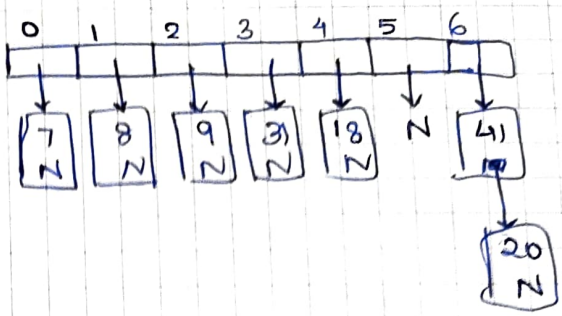
h(9) = 9%.7 = 2

0	1	2	3	4	5	6
7	8	9	41	31	18	20

```
type struct ele{
        int key;
        char name[50];
        int marked;
}ele;

void insert (ele *hashTab, int tabSz, int key, char name[], int *nor){

    if (*nor == tabSz){ printf ("full\n"); }
    else{
        int index = key % tabSz;
        int j = 0;

        while ( hashTab[ (index + j*j) % tabSz ].marked == 1 ){
            j++;
            if (j == tabSz){
                printf ("full\n");
            }
        }

        int new_index = (index + j*j) % tabSz
        hashTab[new_index].key = key;
        hashTab[new_index].marked = 1;
        strcpy ( hashTab[new_index].name, name);
        (*nor)++;
}

void delete (ele *hashTab, int tabSz, int key, char name[], int *nor){
        int index = key % tabSz;
        int j = 0;

        while (hashTab[ (index + j*j) % tabSz ].marked != 0){
            int ni = (index + j*j) % tabSz;
            if (hashTab[ni].marked == 1 && hashTab[ni].key == key){
                hashTab[ni].marked = 0;
                (*nor)--;
                return;
            }j++;
            if(j == tabSz) { printf("full\n"); return; }
}     }
```

Seperate Chaining

1. Insert 7, 20, 41, 31, 18, 8, 9 into a hash table of size 7. Use key % tableSz as hash function & seperate chaining to resolve collision

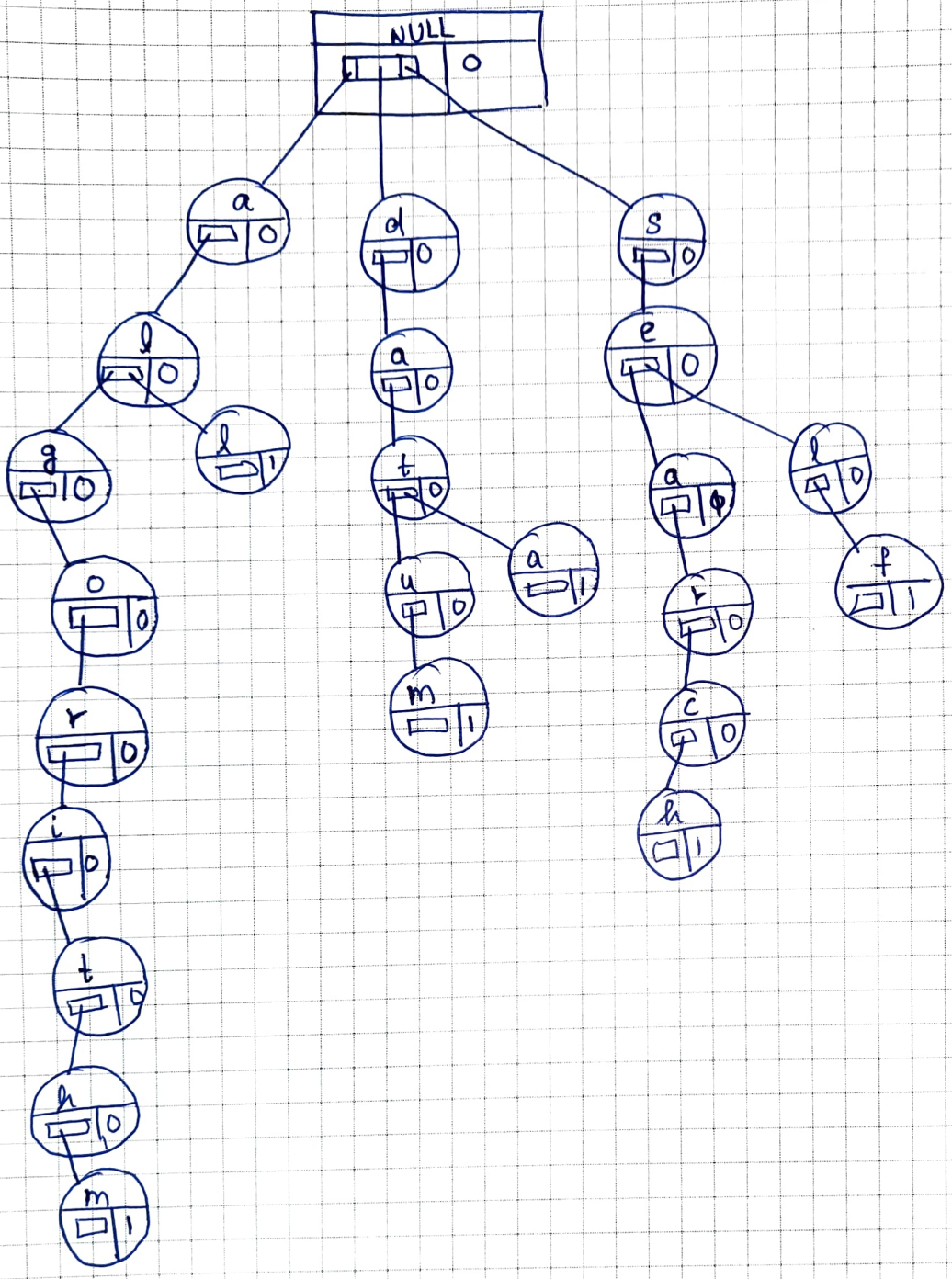


$h(7) = 7 \% 7 = 0$
$h(20) = 20 \% 7 = 6$
$h(41) = 41 \% 7 = 6$
$h(31) = 31 \% 7 = 3$
$h(18) = 18 \% 7 = 4$
$h(8) = 8 \% 7 = 1$
$h(9) = 9 \% 7 = 2$

```
typedef
struct node{
        int key;
        char name[100];
        struct node *next;
}node;

typedef struct hash{
        node *head;
        int count;
}hash;

void insert(hash *ht, int sz, int key, char *name){
        node *temp;
        int index;
        temp = (node*)malloc(sizeof(struct node));
        temp->key = key;
        strcpy(temp->name, name);
        temp->next = NULL;
        index = key % sz;
        temp->next = ht[index].head
        ht[index].head = temp;
        ht[index].count++;
}

void delete(hash *ht, int sz, int key){
        node *prev, *temp;
        int index;
        index = key % sz;

        prev = NULL; temp = ht[index].head;

        while(temp!=NULL && temp->key!=key){
                prev = temp; temp = temp->next;
        }
        if(temp==NULL){
                printf("not found\n");
        }
        else{
                if(prev==NULL){ ht[index].head = temp->next; }
                else{ prev->next = temp->next; }
                free(temp);
                ht[index].count--;
        }
}

void display(hash *ht, int sz){
        int i;
        node *temp;
        printf("\n");

        for(i=0; i<sz; i++){
                printf("%d: ", i);
                if(ht[i].head!=NULL){
                        temp = ht[i].head;
                        while(temp!=NULL){
                                printf(temp->key);
                                printf(temp->name);
                                temp = temp->next;
                        }
                }
        }
        printf("\n");
}
```

TRIE

1. Insert sea, self, search, data, datum, algorithm, all



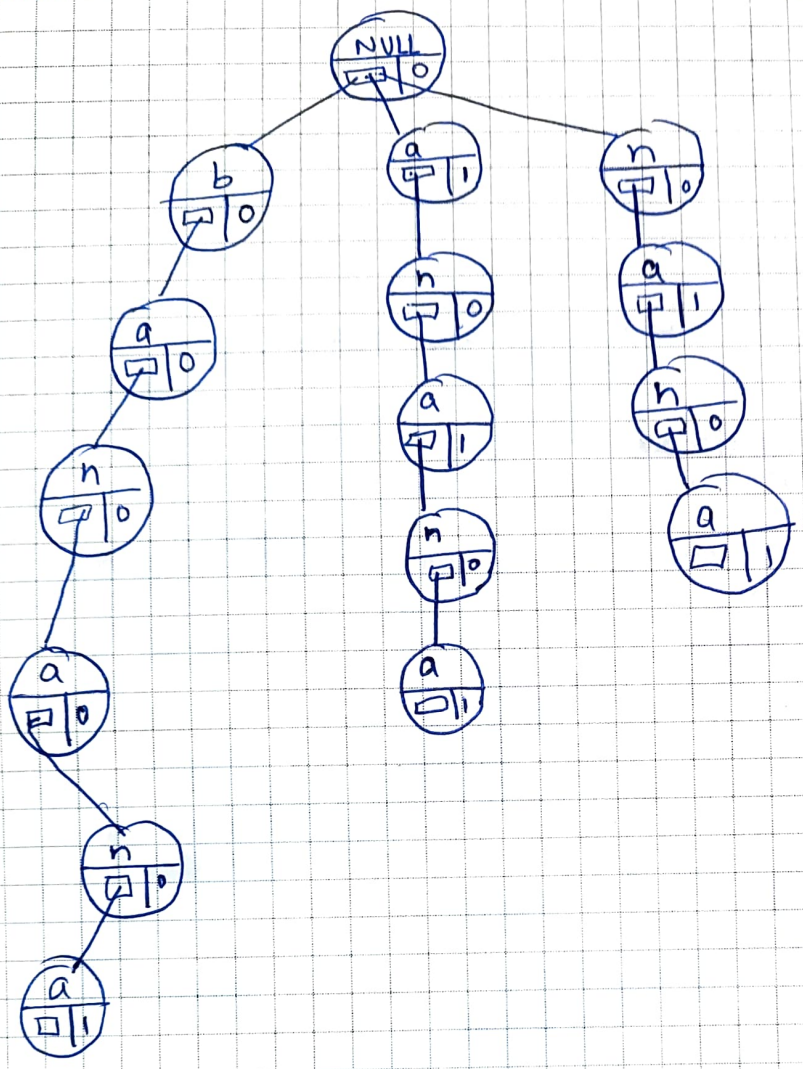
```
typedef trienodedef
        struct trienode* child[26];
        int endofword;
} trienode;

trienode * getnode(){
    int i;
    struct trienode* temp = (trienode*)malloc(sizeof(trienode));

    for(int i=0; i<26; i++){
            temp->child[i] = NULL;
    }
    temp-> endofword = 0;
    return temp;
}


void insert(struct trienode*root, char*key){
        trienode* curr;
        int index;

        curr = root;

        for(int i=0; key[i] != '\0'; i++){
                index = key[i] - 'a';
                if(curr->child[index] == NULL){
                        curr->child[index] = getnode();
                }
                curr = curr->child[index];
        }
        curr->endofword=1;
}

int search(trienode* root, char* key){
        trienode* curr;
        int index;
        curr = root;

        for(int i=0; key[i] != '\0'; i++){
            index = key[i] - 'a';

                if(curr->child[index] == NULL){
                        return 0;
                }
                curr = curr->child[index];
        }
        return (curr != NULL && curr->endofword);
}
```

```
int delete(trienode* root, char *key, int depth) {
    if (root == NULL) {
        return 0;
    }
    if (key[depth] == '\0') {
        if (root->endofword) {
            root->endofword = 0;
            return 1;
        }
        return 0;
    }
    int index = key[depth] - 'a';
    int deleted = delete(root->child[index], key, depth+1);

    if (deleted) {
        free(root->child[index]);
        root->child[index] = NULL;

        if (root->endofword == 0 && isEmpty(root)) {
            return 1;
        }
    }
    return 0;
}

int isEmpty(trienode *root) {
    for (int i=0; i<26; i++) {
        if (root->child[i] != NULL) {
            return 0;
        }
    }
    return 1;
}

void display(trienode *root, char *str, int level) {
    if (root->endofword) {
        str[level] = '\0';
        printf("%s\n", str);
    }
    for (int i=0; i<26; i++) {
        if (root->child[i] != NULL) {
            str[level] = i + 'a';
            display(root->child[i], str, level+1);
        }
    }
}
```

SUFFIX TREE

1. Insert banana

banana anana nana ana na a



Top Down Heap Construction

```
void heapifyTopDown(int heap[], int index){
        int parent = (index-1)/2;
        while (index>0 && heap[index]>heap[parent]){
                int temp = heap[index];
                heap[index] = heap[parent];
                heap[parent] = temp;
                index = parent;
                parent = (index-1)/2;
        }
}

void top_down(int heap[], int n){
        for(int i=1; i<n; i++){
                heapifyUp(heap, i); heapifyTopDown(heap, i);
        }
}

int main(){
        int heap[] = {4,10,3,5,1};
        int n = sizeof(heap)/sizeof(heap[0]);
        top_down(heap, n);
}
```

Bottom Up Heap Construction

```
void heapifyBottomUp(int heap[], int n, int index){
        int largest = index;
        int left = 2*index+1;
        int right = 2*index+2;
        if(left<n && heap[left]>heap[index]){ largest = left; }
        if(right<n && heap[right]>heap[index]){ largest = right; }
        if(largest != index){
                int temp = heap[index];
                heap[index] = heap[largest];
                heap[largest] = temp;
                heapifyBottomUp(heap, n, largest);
        }
}

void bottom_up(int heap[], int n){
        for(int i=(n/2)-1; i>=0; i--){
                heapifyBottomUp(heap, n, i);
        }
}
```