```
Struct Node * Delete( struct Node *p , int key)
{
        Struct Node *q;

        if (p == NULL)
                return NULL;

        if (p -> lchild = NULL && p -> rchild = NULL)   // Leaf Node
        {
                if (p == root)
                        root = NULL;
                free(p);
                return  NULL;
        }

        if ( key < p -> data)
                p -> lchild = Delete (p -> lchild , key);
        else if ( key > p -> data)
                p -> rchild = Delete ( p -> rchild , key);
        else
        {
                if ( Height ( p -> lchild) > Height (p -> rchild))
                {
                        q = InPre ( p -> lchild);
                        p -> data = q -> data;
                        p -> lchild = Delete (p -> lchild , q -> data);
                }

                else
                {
                        q = InSucc ( p -> rchild);
                        p -> data = q -> data;
                        p -> rchild = Delete (p -> rchild , q -> data);
                }
        }
        return  p;
}
```

GENERATING BINARY SEARCH TREE

```
void Createpre( int pre[], int n)      O(n)
{
        Stack st;
        Node *t;
        int i=0;

        root = new Node;
        root -> data = pre[i++];
        root -> lchild = root -> rchild = NULL;
        p = root;

        while (i<n)
        {
                if (pre[i] < p -> data)
                {
                        t = new Node;
                        t -> data = pre[i++];
                        t -> lchild = t -> rchild = NULL;
                        p -> lchild = t;
                        push(&stk, p);
                        p = t;
                }
                else
                {
                        if( pre[i] > p -> data && pre[i] < stacktop(Stk) -> data)
                        {
                                t = new Node;
                                t -> data = pre[i++];
                                t -> lchild = t -> rchild = NULL;
                                p -> rchild = t;
                                p = t;
                        }
                        else
                        {
                                p = pop(&stk);
                        }
                }
        }
}
```

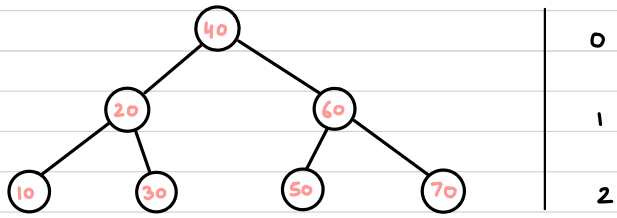
BST can be generated using

- Preorder + Inorder
- Postorder + Inorder

Sorted preorder of BST gives its Inorder /postorder

DRAWBACK OF BINARY SEARCH TREE

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

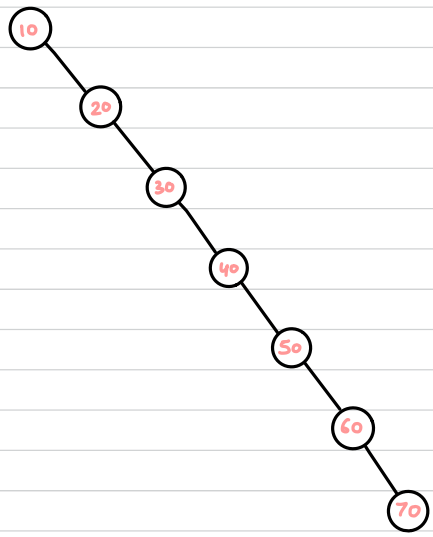


0
1
2

$h = \log_2(n+1) - 1$

$O(\log n)$

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



0
1
2
3
4
5
6

$h = n-1$

$O(n)$

There is no control over height of binary search tree.

It only depends on user's input or sequence of insertion

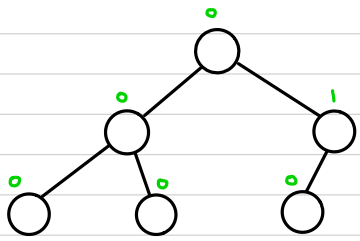
AVL TREES

↳ Height Balanced Binary Search Trees

No of edges

Balance factor = Height of left subtree − Height of right subtree

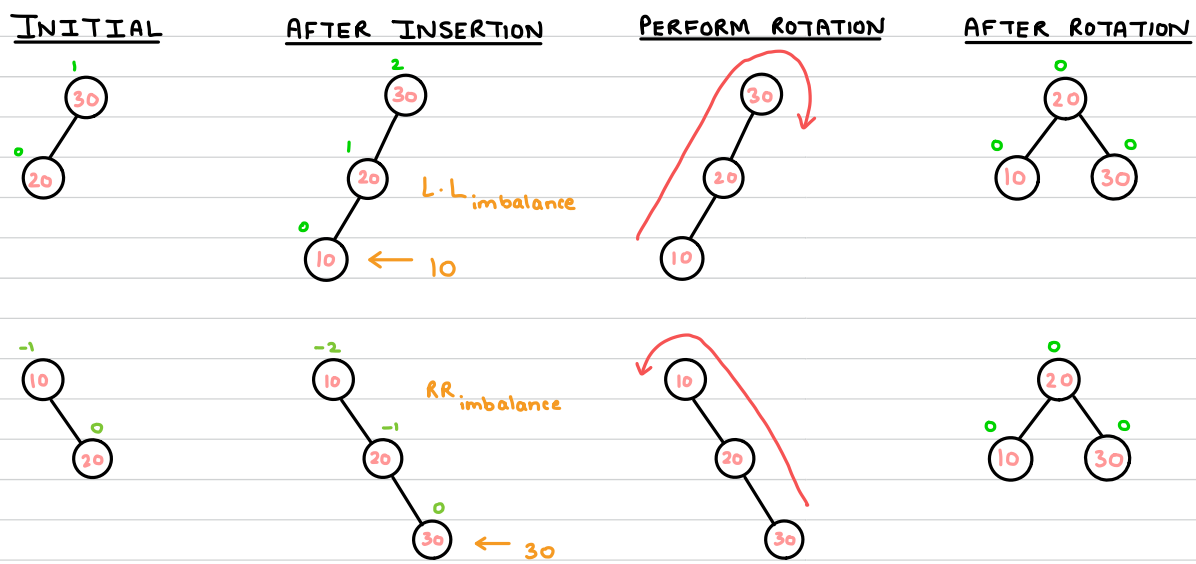
{−1, 0, 1} ⟶ Balanced Tree

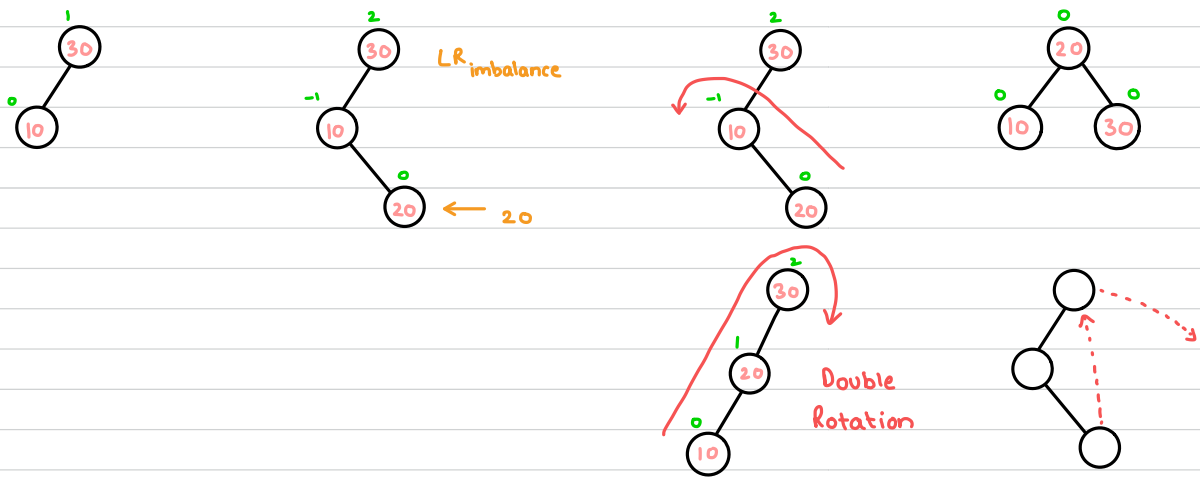


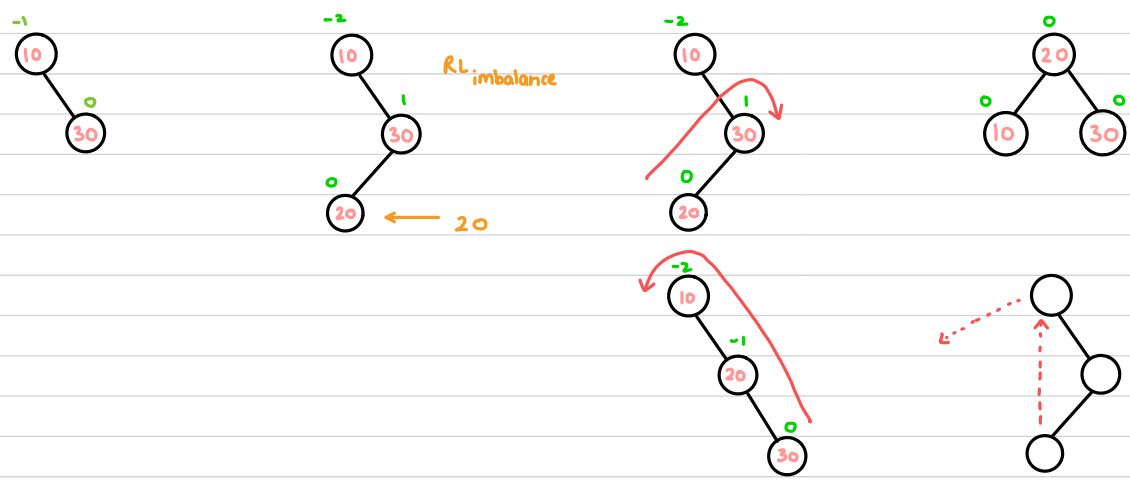
Balanced Tree

Rotations for insertion in AVL trees

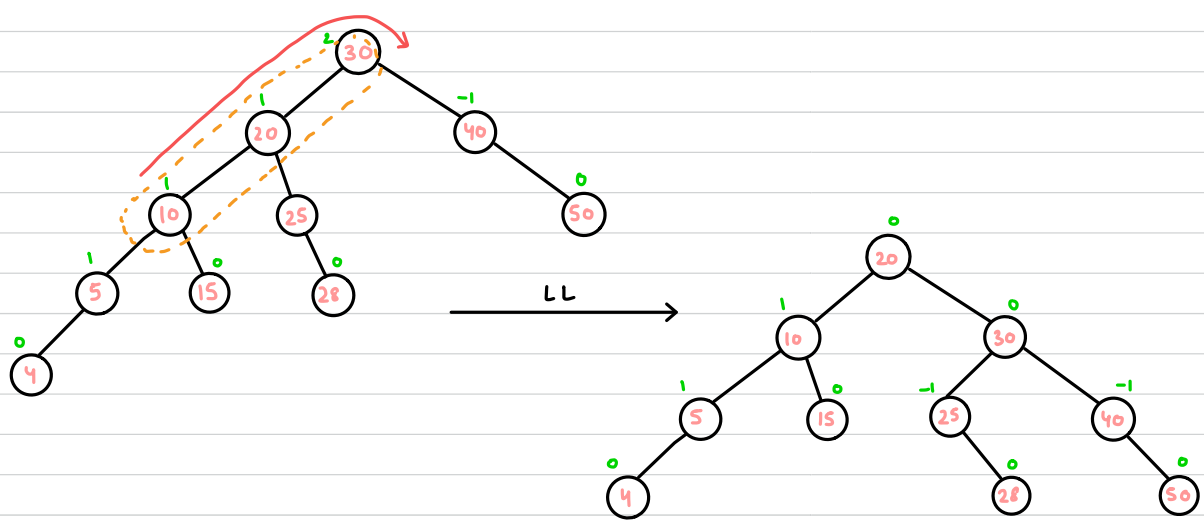
(1) LL Rotation] Single Rotation
(2) RR Rotation
(3) LR Rotation] Double Rotation
(4) RL Rotation

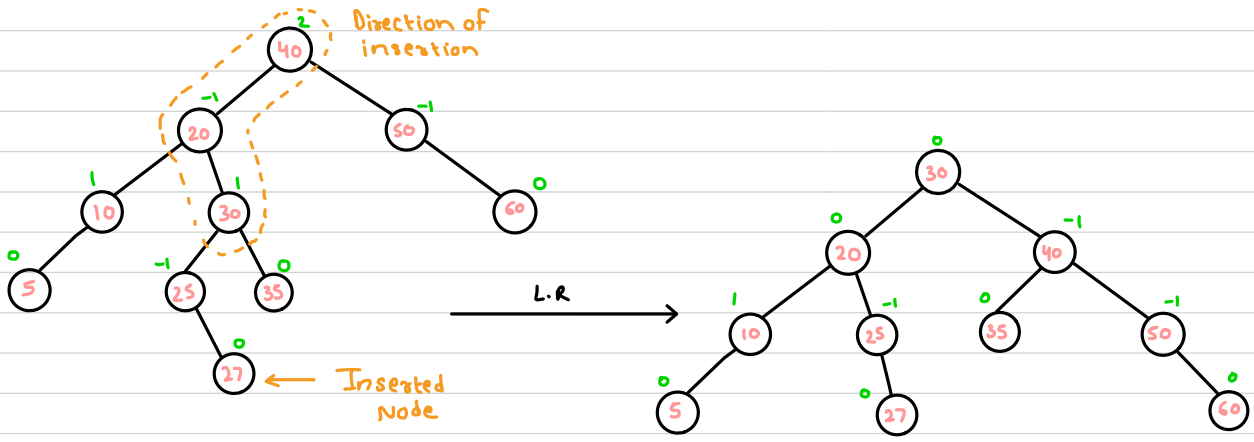






FORMULA OF ROTATION FOR INSERTION

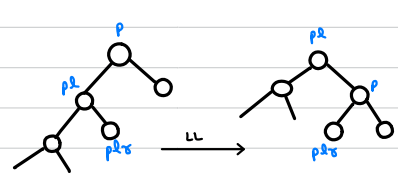




PROGRAM FOR LL ROTATION

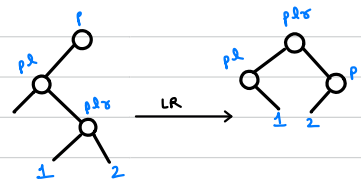
```
Struct Node
{
        Struct Node *lchild;
        int data;
        int height;      // we will set height for
        Struct Node *rchild; each and every Node
} *root = NULL;

Struct Node* Insert(Node *p, int key)
{
        if (p == NULL)
        {
                t = new Node;
                t -> data = key;
                t -> height = 1;
                t -> lchild = t -> rchild = NULL;
                return t;
        }
        if ( key < p -> data)
                p -> lchild = insert (p -> lchild, key);
        else if ( key > p -> data)
                p -> rchild = insert (p -> rchild, key);
        p -> height = NodeHeight(p);   // update height of each
                                          node at returning time
```

```
    if ( BalanceFactor (p) == 2 && BalanceFactor (p -> lchild) == 1)
            return LLRotation (p);
    else if ( BalanceFactor (p) == 2 && BalanceFactor ( p -> rchild ) == -1)
            return LRRotation (p);
    else if ( BalanceFactor (p) == -2 && BalanceFactor (r -> child ) == -1)
            return RRRotation (p);
    else if ( BalanceFactor (p) == -2 && BalanceFactor (r -> child ) == 1)
            return RLRotation ( p);

    return p;
}

int NodeHeight ( struct Node *p)
{
    int hl , hr ;   // Height of left subtree (HL) , height of right subtree (HR)
    hl = p && p -> lchild ? p -> lchild -> height : 0;
    hr = p && p -> rchild ? p -> rchild -> height : 0;
    -> NOT NULL
    return hl + hr ? hl+1 : hr+1;
}

int BalanceFactor ( struct Node *p)
{
    int hl , hr ;
    hl = p && p -> lchild ? p -> lchild -> height : 0;
    hr = p && p -> rchild ? p -> rchild -> height : 0;

    return hl - hr ;
}

Struct Node * LLRotation ( struct Node * p)
{
    Struct Node *pl = p -> lchild ;
    Struct Node *plr = pl -> lchild;

    pl -> rchild = p;
    p -> lchild = plr;
    p -> height = NodeHeight (p);
    pl -> height = NodeHeight (pl);
```



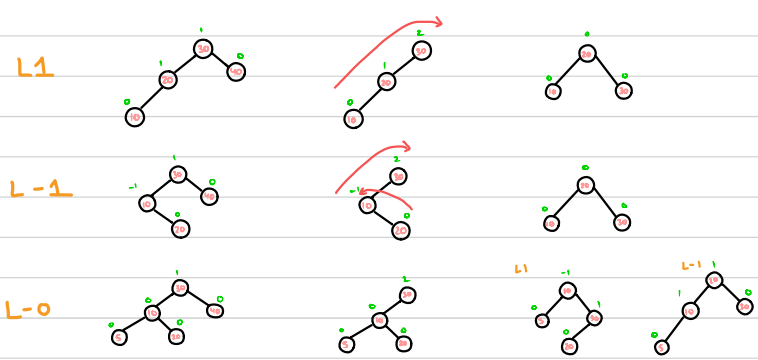
```
    if (root == p)        // If rotation was performed on
        root = pl;        root node, root needs to be
                          updated.
    return pl;
}
```

```
Struct Node * LRRotation(struct Node *p)
{
    Struct Node *pl = p -> lchild;
    Struct Node *plr = pl -> rchild;

    pl -> rchild = plr -> lchild;
    p -> lchild = plr -> rchild;

    plr -> lchild = pl;
    plr -> rchild = p;

    pl -> height = NodeHeight(pl);
    p -> height = NodeHeight(p);
    plr -> height = NodeHeight(plr);

    if (root == p)
        root = plr;
    return plr;  // New root;
}
```



DELETION FROM AVL TREES WITH ROTATION

1. L 1 Rotation
2. L -1 Rotation
3. L 0 Rotation
4. R 1 Rotation
5. R -1 Rotation
6. R - 0 Rotation

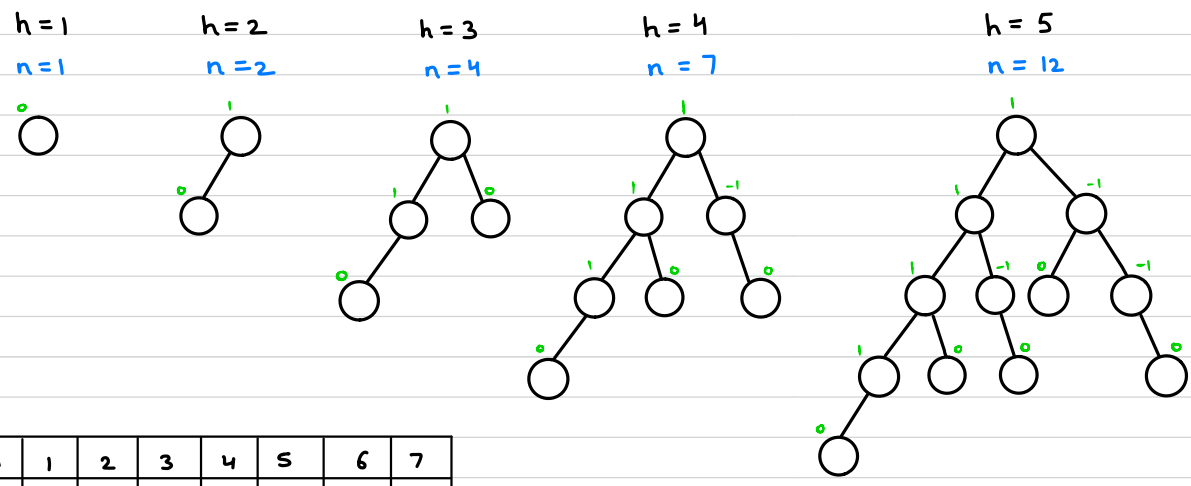


// Other three will be mirror images

HEIGHT VS NODES OF AVL TREES

If height is given :

- Max nodes $n = 2^h - 1$ // Not $2^{h+1} - 1$ because height is starting from 1.
- Min nodes n = Look in table



h	1	2	3	4	5	6	7
n	1	2	4	7	12	20	33

+ +1

$$N(h) = \begin{cases} 0 & 0 \\ 1 & 1 \\ N(h-2) + N(h-1) + 1 & \text{otherwise} \end{cases}$$

// formula same as Fibonacci series

↳ balanced series

If 'N' Nodes are given find:

- Min height = $\log_2 (n+1)$
- Max height = Look in table

Foreg for 13 nodes h = 5 // Look from node towards height

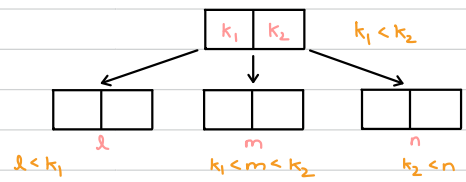
19 nodes h = 5

SEARCH TREES

2 - 3 TREES

- Multiway Search Tree (M Way Search Tree) → degree
- Degree 3 (2-3 trees are Multiway Search Trees with degree 3)
- B Trees (These are height Balanced Search Trees)
- Rules
  - All leaf nodes at same level
  - Every node must have $\left\lceil \frac{n}{2} \right\rceil$ children

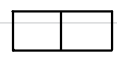
$$\left\lceil \frac{3}{2} \right\rceil = 2$$



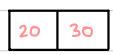
  - Cannot have duplicates

CREATION OF 2-3 TREE

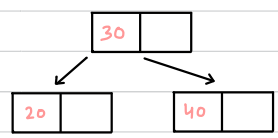
KEYS : 20 , 30 , 40 , 50 , 60 , 10 , 15 , 70 , 80 , 90



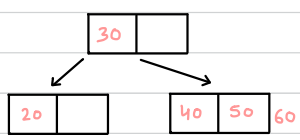
20 , 30

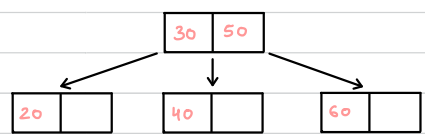


40

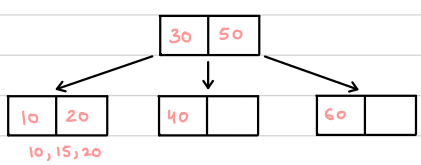


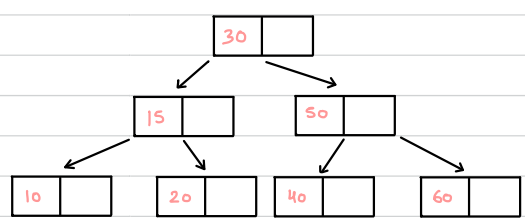
50, 60



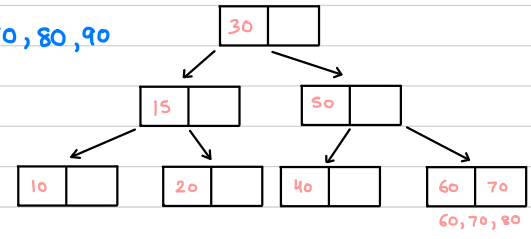


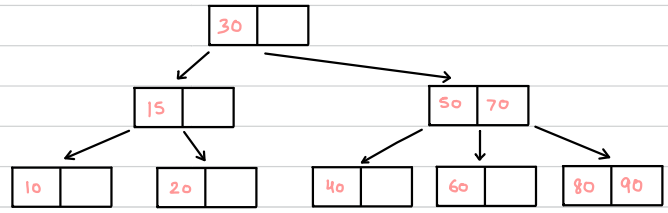
10 , 15





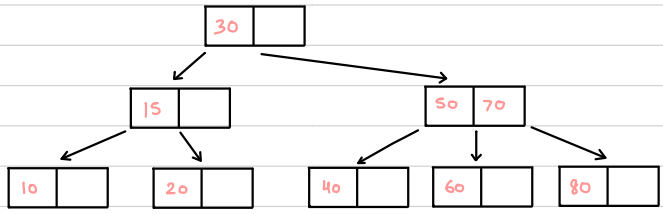
70, 80, 90



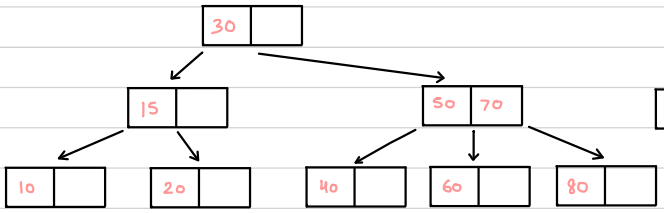


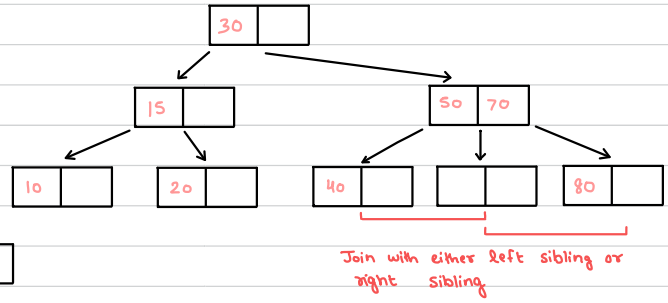
DELETING FROM 2-3 TREE

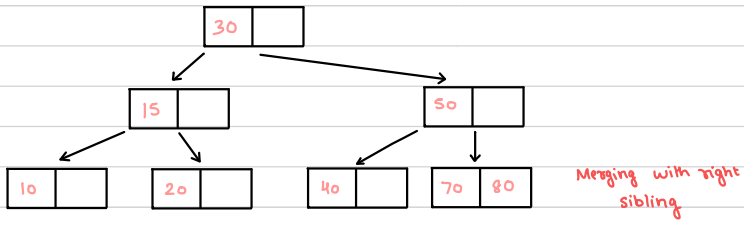
CASE 1 : Simply Delete 90



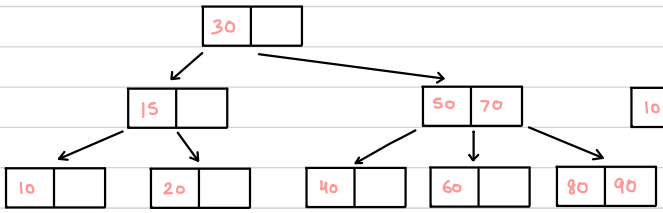
CASE 2 : Delete and merge 60

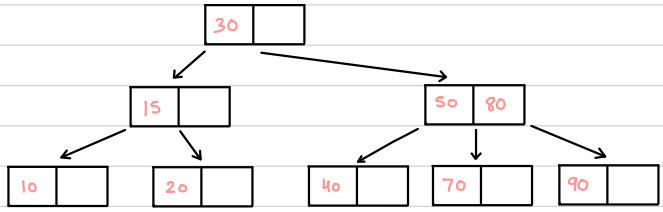




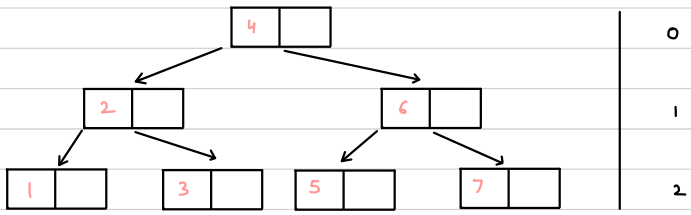


CASE 3 : Borrow 60





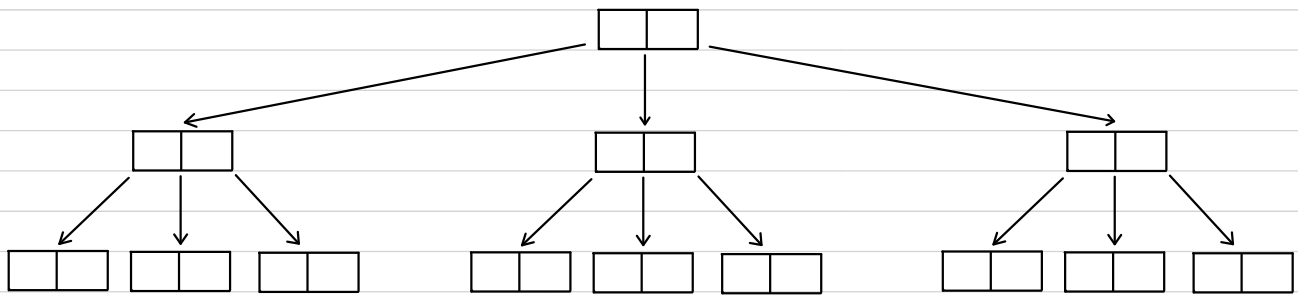
ANALYSIS



Tree with min nodes for given height

Minimum $n = 1 + 2 + 2^2 \ldots$
$= 2^{h+1} - 1$

Max $h = \log_2 (n-1) - 1$
$= O(\log_2 n)$



Max $n = 1 + 3 + 3^2 \ldots\ldots$

$= \dfrac{3^{h+1} - 1}{3-1}$

Min $h = \log_3 [n(3-1) + 1] - 1$

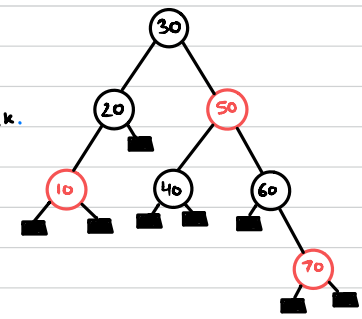
$O(\log_3 n)$

Minimum as well as maximum height is $\log n$

These trees are used for DBMS softwares because a node can have more than one value

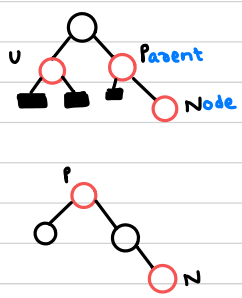
RED BLACK TREE

- It is a height balanced Binary Search Tree, similar to 2-3-4 tree.
- Every node is either Red or Black.
- Root of a Tree is Black
- NULL is also Black.
- Number of Blacks on paths from Root to leaf are same.
- No 2 consecutive Red, Parent and Children of red are Black.
- New inserted Node is Red.
- Height in $\log n \le h \le 2\log n$



CREATION OF RED BLACK TREE

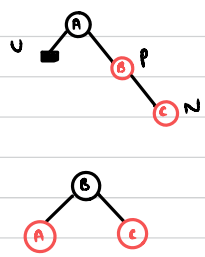
Uncle is Red (for Node)



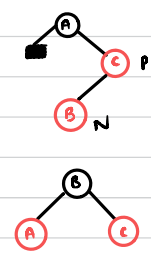
RECOLOURING

Uncle is Black

Zig-Zig (LL/RR)



Zig-Zag (RL/LR)



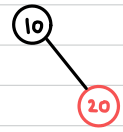
ROTATION

KEYS : 10, 20, 30, 50, 40, 60, 70, 80, 4, 8

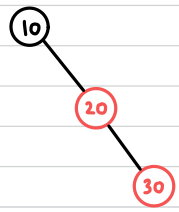
INSERT

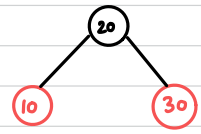
10

20

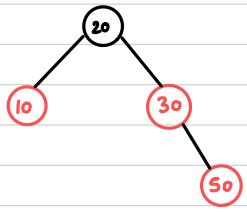


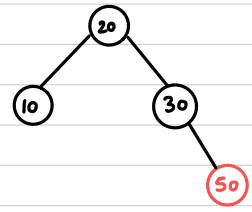
30





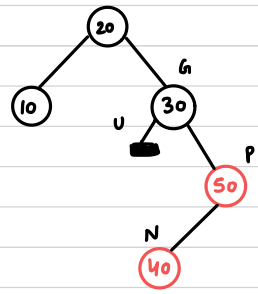
50

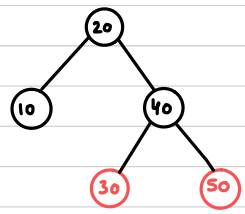




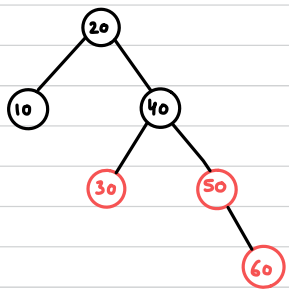
Root must be black

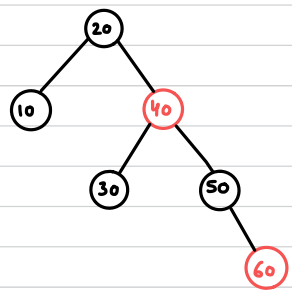
40



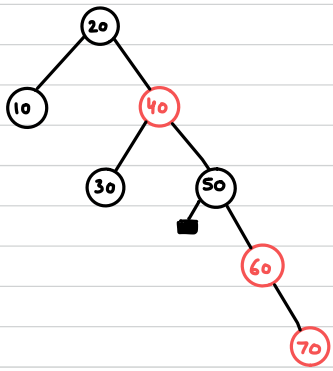


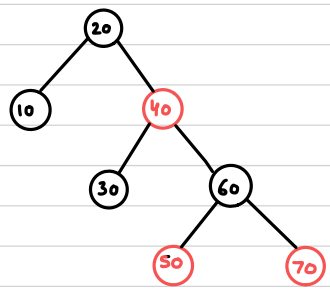
60





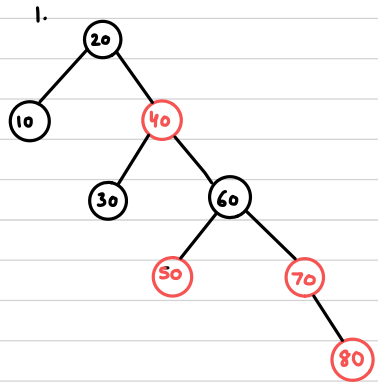
70



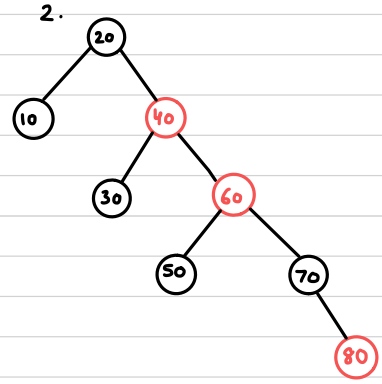


80

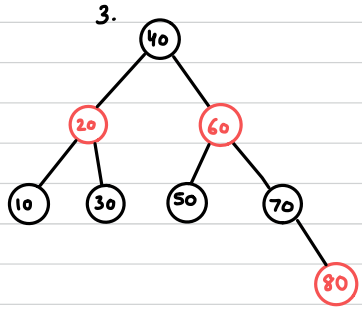
1.



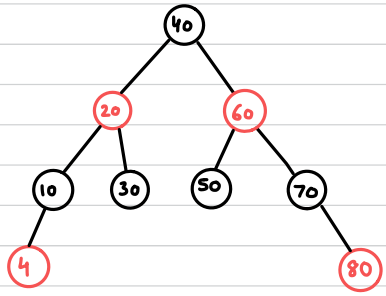
2.



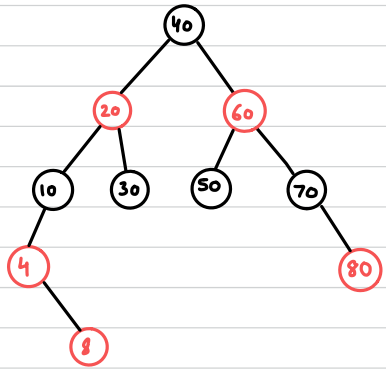
3.

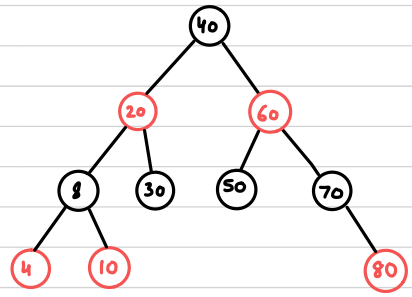


4



8

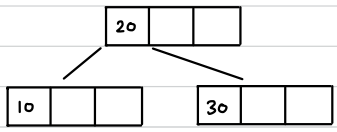


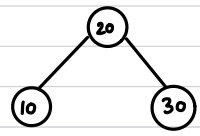


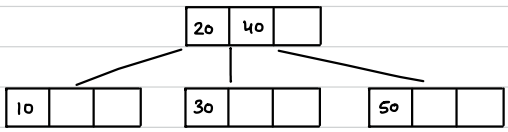
height only red / black = $\log n$

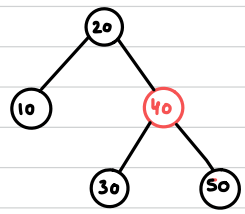
height red and black = $2\log n$

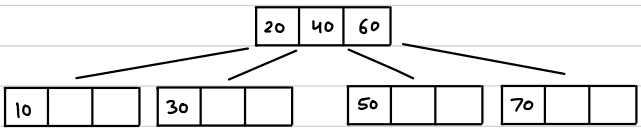
2-3-4 TREES VS RED BLACK TREES

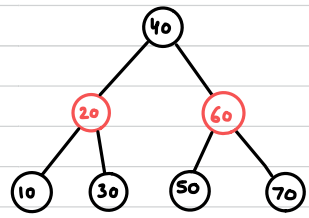








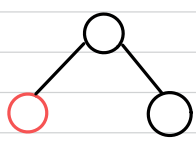


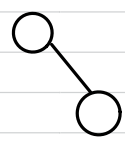


RED BLACK TREE DELETION CASES

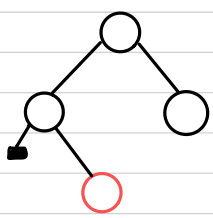
CASE 1 : Deleted Node is Red Node

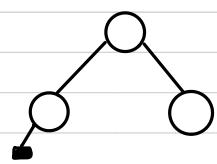
Before Deletion | **After Deletion**

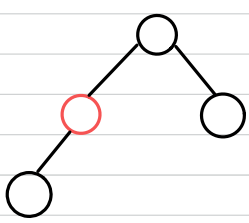


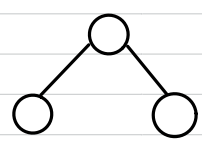


Simply delete as it is a leaf node and red





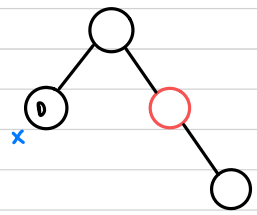




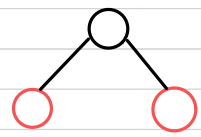
Simply delete because if red node is deleted, the path of black nodes remains unchanged

CASE 2 : Node is black and sibling is red

Before Deletion



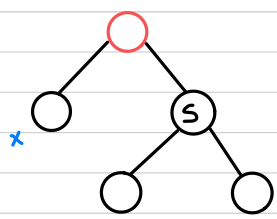
After Deletion



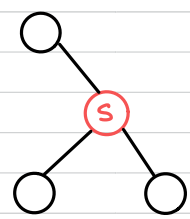
Perform rotation

CASE 3 : Node is black and sibling is also black

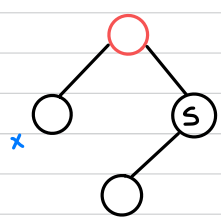
Before Deletion

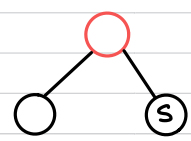


After Deletion



Change sibling to red and parent to black
Recolour





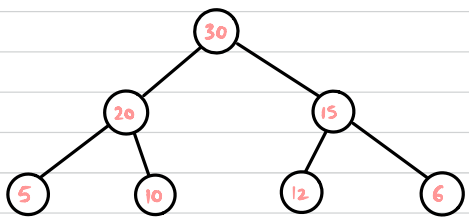
Perform rotation

HEAP

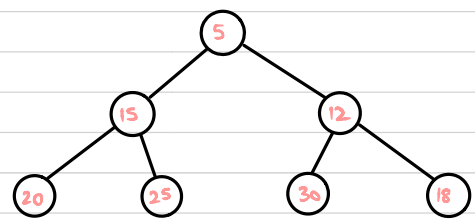
↳ It is a complete binary tree

- what is a heap ?
- Insert in a heap
- Deleting from heap
- Heap Sort
- Heapify
- Priority Queues

Max Heap



Min Heap



1	2	3	4	5	6	7	8	9	10
30	20	15	5	10	12	6			

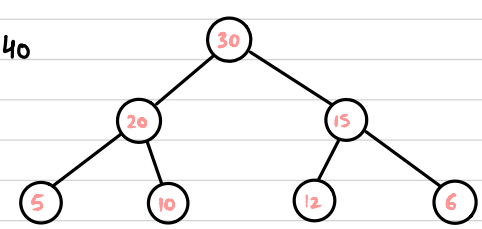
- For complete binary tree, there should not be any free space between the elements of array

Node at index i
Left child at index 2*i
Right child at index 2*i + 1

- Every node should have an element greater than or equal to all its decendants (duplicates can be there)

INSERTING IN HEAP

Element to insert = 40



1	2	3	4	5	6	7	8	9	10
30	20	15	5	10	12	6			

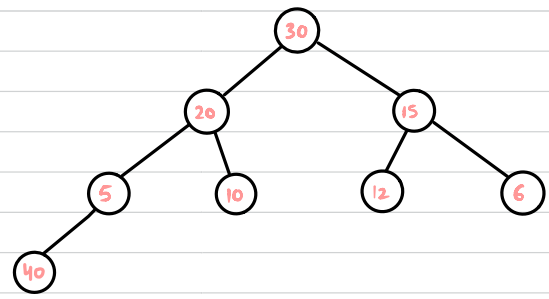
Step 1 : Insert element at next free index in array

1	2	3	4	5	6	7	8	9	10
30	20	15	5	10	12	6	40		

Step 2 : Insert the element in the tree

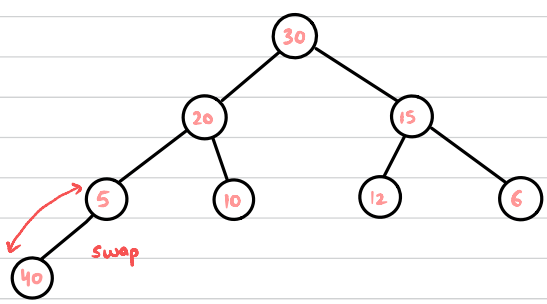
Parent of 40 $= \frac{8}{2} = 4$

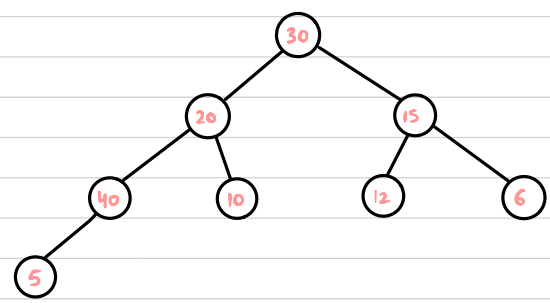
1	2	3	4	5	6	7	8	9	10
30	20	15	5	10	12	6	40		



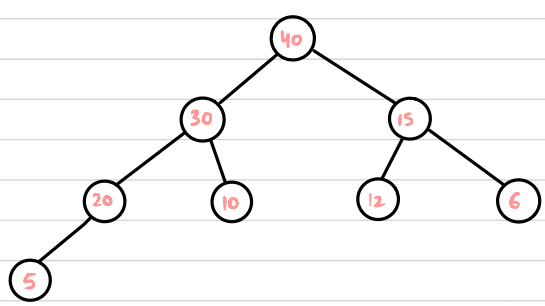
Step 3 : Now tree does not satisfy max heap condition

Compare 40 with its parent / ancestor, if node is found greater than ancestor, swap.





Repeat until condition is satisfied



Check this condition in array also. Compare node with parent $\left(\frac{i}{2}\right)$, if found greater than parent, replace

1	2	3	4	5	6	7	8	9	10
40	30	15	20	10	12	6	5		

PROGRAM

```
void Insert (int A[], int n)        O(logn)
{
        int temp, i=n;
        temp = A[n];

        while( i>1 && temp > A[i/2] )
        {
                A[i] = A[i/2]);
                i= i/2;
        }
        A[i] = temp;
}
```

```
void createheap()   O(nlogn)        1 element - logn
{                                   n elements - nlogn
        int A[] = {0,10,20,30,25,5,40,35};
        int i;
                 because first element is the heap and next element onwards,
        for (i=2; i<=7; i++)        insertion in heap is done.
                Insert (A,i);
}
```

Heapify : Faster method to implement creation of binary heap ← Separate Topics

Element Priority

Elements → 6, 8, 3, 10, 15, 2, 9, 17, 5, 8

- Element is itself a priority
- Smaller number higher priority

1. Insert in same order O(1) O(logn)
   Delete max priority by searching it O(n) O(logn)

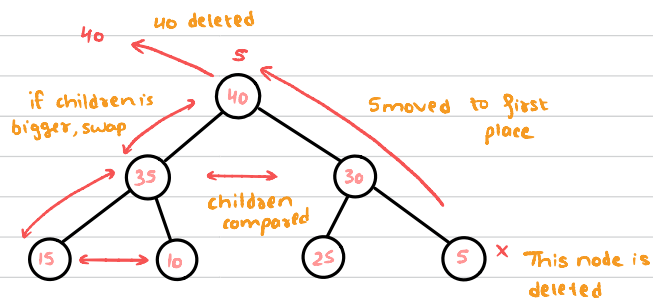
] After implementing using heap

Heap is used to implement priority queues

DELETION FROM HEAP

Only the root node can be deleted from heap i.e the first node

```
void Delete (int A[], int n)
{
        int x, i, j;
        x = A[n];
        A[1] = A[n]      // Smallest or last node is copied at first place
        i=1; j= 2*i;     // i is at index 1 and j is at its left child

        while (j<n-1)
        {
            if (A[j+1] > A[j])   // Right child is greater than left child
                j=j+1;

            if (A[i] < A[j])     // Parent is smaller than child
            {
                swap(A[i], A[j]);
                i=j;
                j = 2*j;
            }
            else
                break;

        A[n] = x;
}
```



HEAP SORT

While deleting, store the deleted element at vacant position

Through this, you will get sorted array in ascending order after deleting all the elements

1	2	3	4	5	6	7	8	9	10
40	35	30	15	10	25	5			

1	2	3	4	5	6	7	8	9	10
5	35	30	15	10	25				

$x = 40$

1	2	3	4	5	6	7	8	9	10
35	15	30	5	10	25	40			

STEPS

- nlogn • Create heap of 'n' elements
- nlogn • Delete 'n' elements 1 by 1

O(nlogn)

SORTING TECHNIQUES

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

$O(n^2)$

4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Tree Sort

$O(n \log n)$

8. Shell Sort — $O(n^{3/2})$

Comparison based Sorts

There is no best sorting technique.

You have to chose the technique which matches your requirements

9. Count Sort
10. Bucket / Bin Sort
11. Radix Sort

$O(n)$

Index Based Sort

// Faster but memory consumption is high

CRITERIA FOR ANALYSIS

1. Number of comparisons
2. Number of swaps
3. Adaptive // If already the array is sorted, it should make minimum comparisons
4. Stable
5. Extra Memory required

1. List sorted on basis of name

Name :	A	B	C	D	E	F	G
Marks :	5	8	6	4	6	7	10

Duplicate elements

2. List sorted on basis of marks

Name :	D	A	C	E	F	B	G
Marks :	4	5	6	6	7	8	10

Here also 'C' should come before 'E' as order must be preserved

This type of algorithms are useful in databases

If the sorting algorithm is preserving the order of duplicate elements in the sorted list then that algorithm is called 'Stable'.

1. BUBBLE SORT

A	8	5	7	3	2
	0	1	2	3	4

$n = 5$

1st Pass	2nd Pass	3rd Pass	4th Pass
8 5 7 3 2	5 7 3 2 8	5 3 2 7 8	3 2 5 7 8
5 8 7 3 2	5 7 3 2 8	3 5 2 7 8	2 3 5 7 8
5 7 8 3 2	5 7 3 2 8	3 2 5 7 8	2 3 5 7 8
5 7 3 8 2	5 3 7 2 8	3 2 5 7 8	2 3 5 7 8
5 7 3 2 8	3 3 2 7 8	5 2 7 8	2 3 5 7 8
Largest element is sorted in first pass			
4 comp	3 comp	2 comp	1 comp
4 swap	3 swap	2 swap	1 swap

→ Not actual number of swaps but maximum number of swaps

No of passes : 4
: $(n-1)$

No of comparison : $1+2+3+4$
: $1+2+3+4 \ldots\ldots (n-1)$
: $\frac{n(n-1)}{2}$ $O(n^2)$

No of swaps : $1+2+3+4$
$1+2+3+4 \ldots\ldots (n-1)$
$\frac{n(n-1)}{2}$ $O(n^2)$

```
void BubbleSort(int A[],int n)
{
    int flag;
    for(i=0;i<n-1;i++)
    {
        flag=0;
        for(j=0;j<n-1-i;j++)
        {
            if(A[j]>A[j+1])
            {
                swap(A[j],A[j+1]);
                flag=1;
            }
        }

        if(flag==0)
            break;
    }
}
```

- Called as bubble sort as lighter or smaller elements come up same as bubbles
- Performing selected number of passes can give same number of largest elements as number of passes performed
- Bubble Sort is adaptive
- It is stable
- 'k' passes give 'k' number of largest element

2. INSERTION SORT

Inserting element in a sorted array at a sorted position

2	6	10	15	20	

ele = 12

// Start comparison from last element

EXAMPLE : A

0	1	2	3	4
8	5	7	3	2

// Assume only first element to be sorted

n = 5

1st Pass

0	1	2	3	4
8		7	3	2

5

1 comp
1 swap

Take out one element and compare and insert

0	1	2	3	4
5	8	7	3	2

2nd Pass

0	1	2	3	4
5	8		3	2

7

2 comp
2 swap

0	1	2	3	4
5	7	8	3	2

3rd Pass

0	1	2	3	4
5	7	8		2

3

3 comp
3 swap

0	1	2	3	4
3	5	7	8	2

4th Pass

0	1	2	3	4
3	5	7	8	

2

4 comp
4 swap

0	1	2	3	4
2	3	5	7	8

No of passes : $(n-1)$

No of comparison : $\frac{n(n-1)}{2}$ $O(n^2)$

No of swaps : $\frac{n(n-1)}{2}$ $O(n^2)$

- Insertion sort is adaptive as no swapping is done if array is sorted. Only 'n' comparisons are done which take minimum time
  It is adaptive by nature (no flag used)
- Insertion sort is stable
- Used in linked list.

```
void InsertionSort (int A[], int n)
{
    for(i=1 ; i<n ; i++)
    {
        j = i-1;
        x = A[i];
        while ( j > -1 && A[j] > x )
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = x;
    }
}
```

3. SELECTION SORT

→ Selecting a position in the array and finding the smallest element suitable for that.

A

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

$n = 6$

1st Pass	2nd Pass	3rd Pass	4th Pass	5th Pass	
0 8 ← i	0 2	0 2	0 2	0 2	0 2
1 6	1 6 ← i	1 3	1 3	1 3	1 3
2 3	2 3 ← k	2 6 ← i	2 4	2 4	2 4
3 2 ← k	3 8	3 8	3 8 ← i	3 5	3 5
4 5	4 5	4 5	4 5 ← k	4 8 ← i	4 6
5 4	5 4	5 4 ← k	5 6	5 6 ← k	5 8

No of comparison : $\frac{n(n-1)}{2}$ $O(n^2)$

No of swaps : $n-1$ $O(n)$

- k passes give k number of smallest elements
- Selection Sort is not adaptive
- Selection Sort is not stable
- Selection Sort performs minimum number of swaps

```
void SelectionSort ( int A[] , int n)
{
    int i, j, k;

    for (i=0 ; i < n-1 ; i++)
    {
        for ( j=k=i ; j<n ; j++)
        {
            if ( A[j] < A[k])
                k = j ;
        }
        swap ( A[i] , A[k]);
    }
}
```

5. MERGING

- Merging two lists in third array
- Merging two lists in single array
- Merging multiple list

• MERGING TWO LISTS IN THIRD ARRAY

$O(m+n)$

	A(m)			B(n)			C	
0	2	i	0	4	j	0	2	k
1	10		1	9		1	4	
2	18		2	19		2	9	
3	20		3	25		3	10	
4	23					4	18	
						5	19	
						6	20	
						7	23	
						8	25	

```
void Merge (int A[], int B[], int m, int n)
{
    int i, j, k;
    i = j = k = 0;

    while(i<m && j<n)
    {
        if (A[i] < B[j])
            c[k++] = A[i++];
        else
            c[k++] = B[j++];
    }

    for (; i<m; i++)          // for remaining elements
        c[k++] = A[i];        in either of list
                              // Only one of these loops
    for (; j<n ; j++)         will work
        c[k++] = B[j];
}
```

• MERGING TWO LISTS IN SINGLE ARRAY

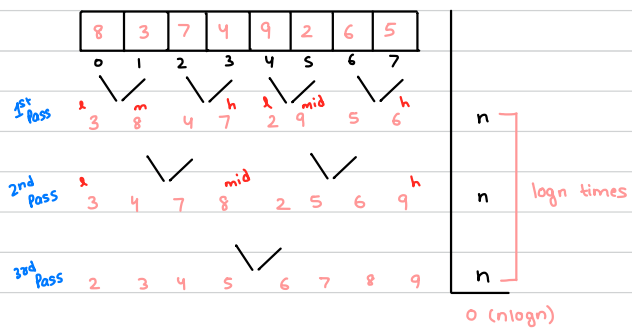
	l			mid				h
A	2	5	8	12	3	6	7	10
	0	1	2	3	4	5	6	7
	i				j			

B	2	3	5	6	7	8	10	12
	0	1	2	3	4	5	6	7
	k							

```
void Merge (int A[], int l, int mid, int h)
{
    int i, j, k;
    int B[h+1];
    i = l ; j = mid+1 ; k = l;

    while( i<=mid && j<=h)
    {
        if (A[i] < A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++];
    }

    for (; i<=mid; i++)
        B[k++] = A[i];

    for (; j<=h; j++)
        B[k++] = A[j];
}
```

ITERATIVE MERGESORT

We consider each of these elements to be a sorted list



RECURSIVE MERGE SORT O(nlogn)

```
void RMergeSort (int A[], int l, int h)
{
    if (l < h)
    {
        mid = ⌊(l+h) /2⌋ ;
        RMergeSort (A, l, mid);
        RMergeSort (A, mid+1, h);
        Merge (A, l, mid, h);
    }
}
```

```
void IMergeSort (int A[], int h)
{
    int p, i, l, mid, h;

    for ( p = 2 ; p <= n ; p = p*2 )
    {
        for (i = 0 ; i + p - 1 < n ; i = i + p)
        {
            l = i ;
            h = i + p - 1;
            mid = ⌊(l+h) /2⌋ ;

            Merge (A, l, mid, h);
        }
    }

    if ( p/2 < n)
        Merge (A, 0, p/2, n-1);
}
        // If number of elements are odd
```

6. QUICK SORT → Selecting a element and then finding its position

Pivot

50	70	60	90	40	80	10	20	30
i								j

↳ Starts searching for an element greater than pivot (STEP 1)

↳ Starts searching for element smaller or equal than pivot (STEP 2)

When larger and smaller elements found, exchange them ← STEP 3

PARTITIONING PROCEDURE

50	30	60	90	40	80	10	20	70
	i							j
50	30	20	90	40	80	10	60	70
		i					j	
50	30	20	10	40	80	90	60	70
			i			j		
50	30	20	10	40	80	90	60	70
				j	i			

If i becomes greater than j, we have checked entire list Now j^{th} index element and pivot should be swapped (STEP 4)

(40	30	20	10)	50	(80	90	60	70)
				j				

perform quick sort left and right side

partitioning position

```
int Partition(int A[], int l, int n)
{
    int Pivot = A[l];
    int i=l, j=h;

    do
    {
        do {i++} while (A[i] ≤ pivot);
        do {j--} while (A[i] > pivot);

        if (i<j)
            swap(A[i], A[j]);
    }while (i<j);
    swap(A[l], A[j]);
    return j;
}
```

lowest index (0) — highest index (8) in this example

```
void QuickSort(int A[], int l, int h)
{   int j;
    if (l<h)
    {       j= Partition(A, l, h);
            QuickSort(A, l, j);
            QuickSort(A, l+1, h);
    }
}
```

Best Case: If partitioning is in middle
$O(n\log n)$

Worst Case: If partioning is on any end
$O(n^2)$

Avg case: $O(n\log n)$

- best case: sorted list
  (first make middle element as pivot)
- worst case: partitioning on any end
  $O(n^2)$

8. SHELL SORT

Used for large arrays

```
void ShellSort( int A[], int n)
{
        int gap, i, j, temp;

        for( gap = n/2 ; gap >= 1 ; gap /= 2)
        {
                for ( i = gap ; i < n ; i++)
                {
                        temp = A[i];
                        j = i - gap;

                        while( j >= 0 && A[j] > temp)
                        {
                                A[j + gap] = A[j];
                                j = j - gap;
                        }

                        A[j + gap] = temp;
                }
        }
}

void main()
{
        int A[] = { 11, 13, 7, 12, 16, 9, 24, 5, 10, 3 }, n = 10;

        ShellSort (A, n);
}
```

9. COUNT SORT

	0	1	2	3	4	5	6	7	8	9
A	3	6	8	8	10	12	15	15	15	20

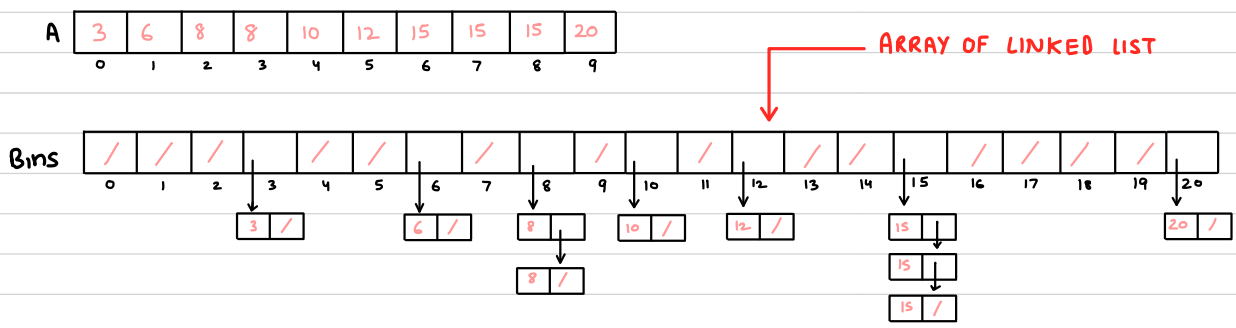
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C	0	0	0	~~0~~ 1	0	0	~~0~~ 1	0	~~0~~ 2	0	~~0~~ 1	0	~~0~~ 1	0	0	~~0~~ 3	0	0	0	0	~~0~~ 1

Fastest sorting method but high memory consumption

```
void CountSort(int A[], int n)
{
    int max, i, j;
    int *c;

    max = findMax(A,n);
    C = new int[max+1];          C = (int *) malloc (sizeof (int) * (max+1));

    for(i=0; i<max+1; i++)       n
        c[i] = 0;

    for (i=0; i<n; i++)          n
        C[A[i]]++;

    i=0, j=0;

    while(i<max+1)               n
    {
        if (c[i] > 0)
        {
            A[j++] = i;
            c[i] --;
        }

        else
            i++;
    }
}
```

O(n)

10. BUCKET / BIN SORT




```
void BinSort(int A[], int n)
{
        int max, i, j;
        max = findMax(A,n);
        Bins = new Node*[(max+1)];

        for (i=0; i<max+1; i++)
                Bins[i] = NULL;

        for (i=0; i<n; i++)
                Insert(Bins[A[i]], A[i]);        // Insert at end of linked list

        i=0, j=0;

        while(i < max+1)
        {
                while(Bins[i] != NULL)
                {
                        A[j++] = Delete(Bin[i]);        O(n)
                }
                i++;
        }
}
```

11. RADIX SORT

1st Pass

Elements arranged on basis of one's digit

	0	1	2	3	4	5	6	7	8	9	
A	237	146	259	348	152	163	235	48	36	62	

	0	1	2	3	4	5	6	7	8	9	
Bin	/	/	152 62	163	/	235	146 36	237	348 48	259	← ARRAY OF LINKED LIST

	0	1	2	3	4	5	6	7	8	9	
A	152	62	163	235	146	36	237	348	48	259	

// Take out the elements in FIFO way and store them

2nd Pass

two's digit

	0	1	2	3	4	5	6	7	8	9	
Bin	/	/	/	235 36 237	146 348 48	152 259	62 163	/	/	/	

	0	1	2	3	4	5	6	7	8	9	
A	235	36	237	146	348	48	152	259	62	163	

3rd Pass

three's digit

	0	1	2	3	4	5	6	7	8	9	
Bin					/	/	/	/	/	/	
	36 48 62	146 152 163	235 237 259	348							

	0	1	2	3	4	5	6	7	8	9	
A	36	48	62	146	152	163	235	237	259	348	

two digit number's third digit considered as 0

1. $[A[i] / 1] \% 10$
2. $[A[i] / 10] \% 10$
3. $[A[i] / 100] \% 10$

Decimal numbers require 10 sized bin (0 - 9)
Binary numbers require 2 sized bin (0, 1)
Octal numbers require 8 sized bin (0-7)

O(n) and minimum storage required
Storage depends on maximum number of digits of largest element

HASHING TECHNIQUE

- **Why hashing?**

There are three methods of searching
1. Linear Search – $O(n)$
2. Binary Search – $O(\log n)$ // But in binary search, elements must be arranged in sorted order, so we have to do some extra work
3. Hashing – $O(1)$

→ It is the fastest method, but requires very large space as it depends on maximum element

DRAWBACK

	0	1	2	3	4	5	6	7	8	9
A	3	6	8	8	10	12	15	15	15	20

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
C	0	0	0	3	0	0	6	0	8	0	10	0	12	0	0	15	0	0	0	0	20

- **Ideal Hashing**

$h(x) = x$ hash function

Mapping

✓ One to one func

one to many

✓ many to one func

many to many

$h(x) = x \% 10$ gives remainder

many-one

COLLISION

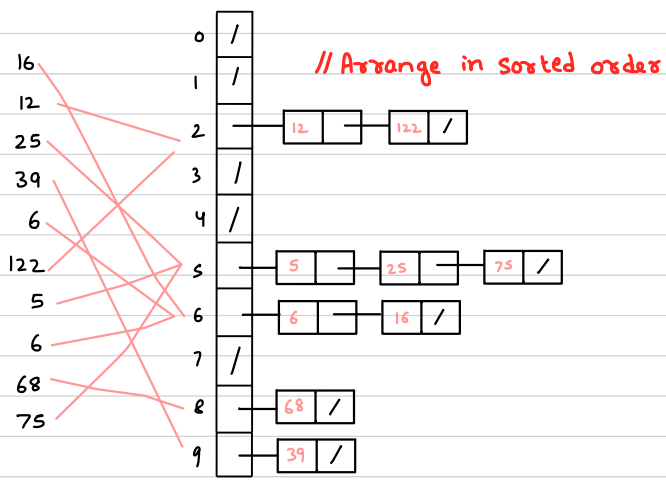
- **How to avoid COLLISION?**

Open Hashing // No space restriction
- Chaining

Closed Hashing // Limited Space
- Open Addressing // If collision, store element elsewhere in free space
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

  Methods for storing in free space

- **CHAINING**



// Arrange in sorted order

Arrange on basis of first digit

$h(x) = x \% 10$ ← Hash Function

$n = 100$	No of keys
size = 10	No of index
$\lambda = \frac{n}{size}$	LOADING FACTOR

Average Successful Search

$t = 1 + \frac{\lambda}{2}$

Average Unsuccessful Search

$t = 1 + \lambda$

If keys are: 5, 35, 95, 145, 175, 265, 845

Then modify your hash function

PROGRAM

```
void Insert( struct Node *H[], int key)
{
        int index = hash(key);
        SortedInsert( &H[index], key);
}

int hash( int key)
{
        return key % 10;
}

void SortedInsert( struct Node **H, int x)
{
        Struct Node *t, *q = NULL, *p = *H;
        t = (struct Node *) malloc( sizeof (struct Node));
        t -> data = x;
        t -> next = NULL;

        if ( *H = NULL)
                *H = t;
        else
        {
                while ( p && p -> data < x)
                {
                        q = p;
                        p = p -> next;
                }

                if ( p == *H)
                {
                        t -> next = *H;
                        *H = t;
                }
                else
                {
                        t -> next = q -> next;
                        q -> next = t;
                }
        }
}
```

```
Struct Node
{
        int data;
        Struct Node *next;
};
```

```
Void main()
{
        Struct Node * HashTable[10], *temp;
        int i;

        for (i=0; i<10; i++)
                HashTable[i]= NULL;

        Insert (HashTable, 12);
        Insert (HashTable, 22);
        Insert (HashTable, 42);

        temp = Search (HashTable[hash(21)], 21);
}


Struct Node *Search (Struct Node *p, int key)
{
        while (p != NULL)
        {
                if (key == p->data)
                        return p;
                else
                        p = p->next;
        }
        return NULL;
}
```

LINEAR PROBING

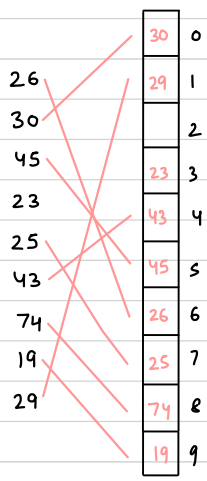
$h(x) = x \% 10$

$h'(x) = (h(x) + f(i)) \% 10$ // For probing

$f(i) = i$

where $i = 0, 1, 2 \ldots$

In case of collision, insert at next free space



$h'(25) = (h(25) + f(0)) \% 10$
$= (5 + 0) \% 10 = 5$

$h'(25) = (h(25) + f(1)) \% 10$
$= (5 + 1) \% 10 = 6$

$h'(25) = (h(25) + f(2)) \% 10$
$= (5 + 2) \% 10 = 7$

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

Cyclic behaviour

For searching:

Key : 45 ✓ $45 \% 10 = 5$ found at index 5

Key : 74 ✓ $74 \% 10 = 4$
- Not found at index 4
- Search Next index until 74 is found or blank space is found
- Found at index 8

// Search takes more than constant time

Key : 40 ✗ $40 \% 10 = 0$
- Not found at index 0
- At index 2, blank space found
- Element does not exist

ANALYSIS

$\lambda = \frac{n}{\text{size}} = \frac{9}{10} = 0.9$

DRAWBACK

** $\lambda \leq 0.5$ // Table should be atmost half filled so, there are blank spaces and searching can be made faster

Average Successful Search

$t = \frac{1}{\lambda} \ln\left(\frac{1}{1-\lambda}\right)$

Average Unsuccessful Search

$t = \frac{1}{1-\lambda}$

DELETION IS NOT EASY IN LINEAR PROBING

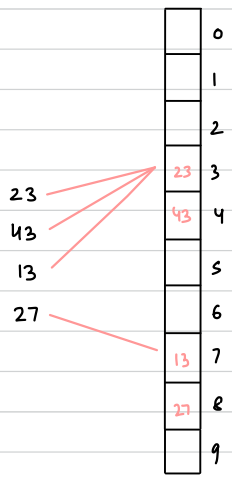
PROGRAM

```
int hash( int key)
{
        return key % 10;
}

int probe ( int H[], int key)
{
        int index = hash (key);
        int i = 0;

        while ( H[ (index + i) % 10] ! = 0)
                i++;
        return (index + i) % 10;
}

void Insert ( int H[] , int key)
{
        int index = hash (key);
        if ( H[ index] ! = 0)
                index = probe (H , key);
        H[ index] = key;
}

int Search (int H[], int key)
{
        int index = hash (key);
        int i = 0;

        while ( H[ (index + i) % 10] ! = key)
                i++;
        return (index + i) % 10;
}
```

```
Void main()
{
        struct Node * HashTable [10], * temp;
        int i;

        for (i = 0 ; i < 10 ; i++)
                HashTable [i] = NULL;

        Insert (HashTable , 12);
        Insert (HashTable , 22);
        Insert (HashTable , 42);

        temp = Search ( HashTable , 35);
}
```

QUADRATIC PROBING

The drawback of linear probing is that elements cluster together and form a group. To resolve this issue, quadratic probing is introduced.



$h'(x) = (h(x) + f(i)) \% 10$ where $f(i) = i^2$

$i = 0, 1, 2 \ldots$

$h'(23) = (h(13) + f(0)) \% 10$

$= (3 + 0) \% 10 = 3$

$h'(43) = (h(43) + f(0)) \% 10$

$= (3 + 0) \% 10 = 3$

$= (3 + 1) \% 10 = 4$

$h'(13) = (h(13) + f(2)) \% 10$

$= (3 + 4) \% 10 = 7$

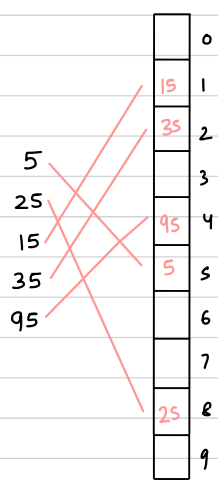
Average Successful Search

$$\frac{-\log_e(1-\lambda)}{\lambda}$$

Average Unsuccessful Search

$$\frac{1}{1-\lambda}$$

DOUBLE HASHING



$h1(x) = x \% 10$

$h2(x) = R - (x \% R)$

$h'(x) = (h1(x) + i * h2(x)) \% 10$

where R is prime num smaller than size of Hash Table

where $i = 0, 1, 2 \ldots$

$h'(25) = (5 + 1 * 3) \% 10 = 8$

$7 - (25 \% 7)$

$7 - 4 = 3$

$h'(15) = (5 + 1 * 6) \% 10 = 1$

$7 - (15 \% 7)$

$7 - 1 = 6$

$h'(35) = (5 + 1 * 7) \% 10 = 2$

$7 - (35 \% 7)$

$7 - 0 = 7$

$h'(95) = (5 + 3 * 3) \% 10 = 4$

$7 - (95 \% 7)$

$7 - 4 = 3$

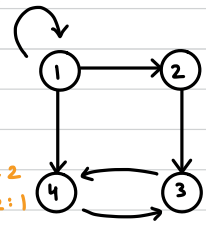
GRAPHS

$G = (V, E)$

V = Set of vertices

E = Set of edges

Self loop

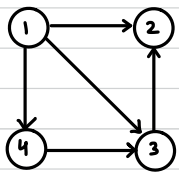


Indegree: 2
Outdegree: 1

Parallel Edges

Directed Graph

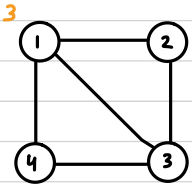
- Edges are having directions



Simple Diagraph

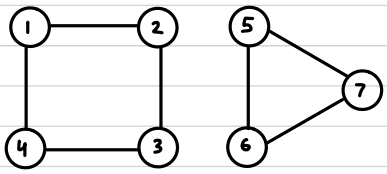
- Without self loop
- Without parallel edges

degree 3



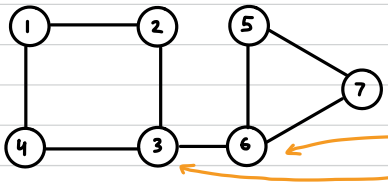
Graph / Non-directed Graph

- Undirected Edges



Non-connected Graph

2 components

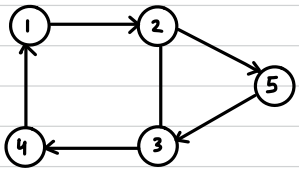


Connected Graph

Articulation Points

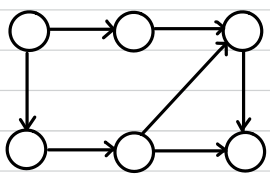
- Those vertices whose removal will divide the graph into multiple components

Connected components



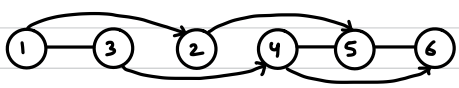
Strongly Connected Graph (Directed Graph)

- All vertices can be reached from any vertex
- There is a path between every pair of vertices
- **Path** is set of vertices which are connecting pair of vertices
- **Cycle** is a circular path that is starting from same vertex and ending at same vertex



Directed Acyclic Graph (DAG)

- Directed Graph
- With no cycles
- These can be arranged linearly such that edges are going in only forward direction

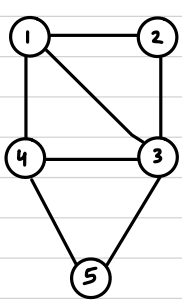


Topological Ordering

REPRESENTATION OF UNDIRECTED GRAPH

(1) Adjacency Matrix
(2) Adjacency List
(3) Compact List

ADJACENCY MATRIX



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

5 x 5

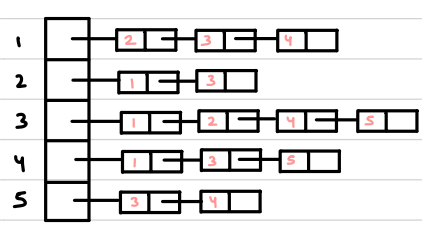
$|V| = n = 5$

$|E| = e = 7$

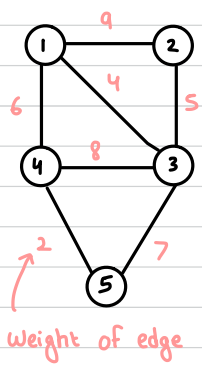
(i, j)

$A[i][j] = 1$

ADJACENCY LIST



COST ADJACENCY MATRIX (Representation of weight)



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

5 x 5

COST ADJACENCY LIST

Add another area to node for weight of edge

COMPACT LIST

Ending point of last vertex

/	7	10	12	16	19	21	2	3	4	1	3	1	2	3	4	1	3	5	3	4
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Vertices: 1 (1–5), 2 (7–9), 3 (10–11), 3 (12–15), 4 (16–18), 5 (19–20)

Starting indexes

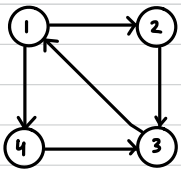
$|V| + 2|E| + 1$

$5 + 2 \times 7 + 1 = 20$

$20 + 1 = 21$

REPRESENTATION OF DIRECTED GRAPH

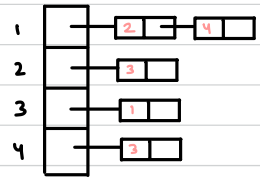
ADJACENCY MATRIX



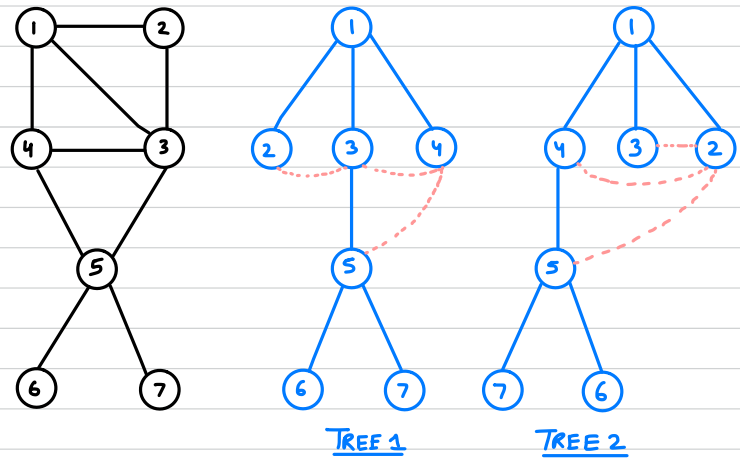
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	0
4	0	0	1	0

4 x 4

ADJACENCY LIST



BREADTH FIRST SEARCH (BFS)



Multiple trees can be generated from the same graph

- We can start from any vertex
- Each vertex should be explored fully
- Dotted pink lines represent edges that make a complete cycle called cross edges
- These trees are called BFS Spanning Trees

BFS TREE 1 : 1, 2, 3, 4, 5, 6, 7 // Same as Level Order Traversal in trees

BFS TREE 2 : 1, 4, 3, 2, 5, 7, 6

```
                    ← Starting vertex
void BFS (int i)                    O(n²)
{
    int u, v;
    printf("%d", i);
    visited[i] = 1;
    enqueue (q, i);

    while ( ! isEmpty (q))
    {
        u = dequeue(q);             ← Take out vertex from queue

For scanning row of matrix →  for( v = 1; v <= n; v++)
                              {
                                  if ( A[u][v] == 1 && visited[v] == 0)
                                  {
                                      printf("%d", v);
                                      visited[v] = 1;
                                      enqueue (q, v);
                                  }
                              }
    }
}
```

Should not be visited

A	0	1	2	3	4	5	6	7
0								
1		0	1	1	1	0	0	0
2		1	0	1	0	0	0	0
3		1	1	0	1	1	0	0
4		1	0	1	0	1	0	0
5		0	0	1	1	0	1	1
6		0	0	0	0	1	0	0
7		0	0	0	0	1	0	0

QUEUE q

visited

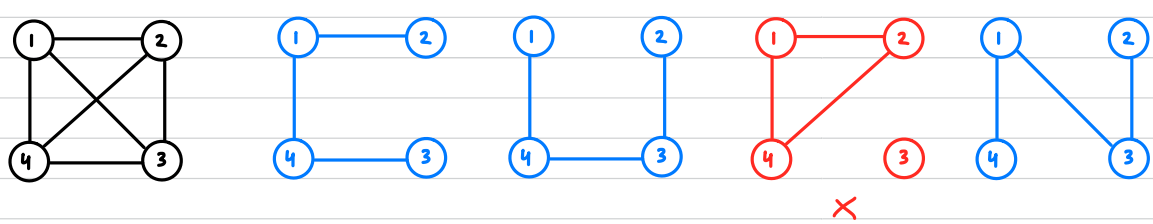
0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0

DEPTH FIRST SEARCH (DFS)

```
void DFS (int u)
{
    int v;

    if ( visited [u] == 0)
    {
        printf (" %d ", u);
        visited [u] = 1;

        for ( v = 1 ; v <= n ; v++)
        {
            if ( A [u] [v] == 1 && visited [v] ! = 0)
                DFS (v);
        }
    }
}
```

SPANNING TREES

Spanning tree is a sub graph of a graph having all vertices of a graph and |V|-1 edges and there should not be any cycle.



$|V| = 4$

$|E| = 6$

$^{|E|}C_{|V|-1} = {}^{6}C_{3}$ ways or number of spanning trees

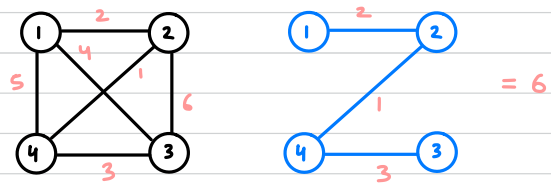
but not include trees forming cycles

$^{|E|}C_{|V|-1} - \text{cycles} = {}^{6}C_{3} - 4$ // In this example

$= 16$

MINIMUM COST SPANNING TREE

If weights are added to edges of graph, then the spanning tree with minimum cost of weights is called minimum cost spanning tree.

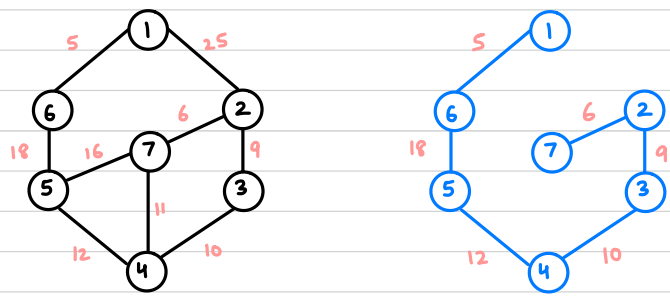


PRIM'S MINIMUM COST SPANNING TREE

$(|V| - 1)\ |E|$

$ne = n \times n$

$O(n^2)$

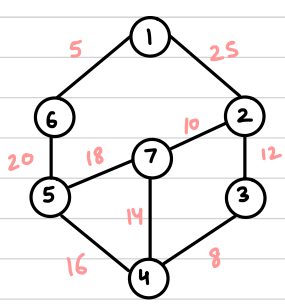


STEP 1 : Select the edge with least weight along with its vertices

STEP 2 : Then compare the edges joined with the vertices and select the one with least weight along with its vertex

STEP 3 : Repeat step 2 until spanning tree is obtained

PROGRAM



COST	0	1	2	3	4	5	6	7
0	-	-	-	-	-	-	-	-
1	-	-	25	-	-	-	5	-
2	-	25	-	12	-	-	-	10
3	-	-	12	-	8	-	-	-
4	-	-	-	8	-	16	-	14
5	-	-	-	-	16	-	20	18
6	-	5	-	-	-	20	-	-
7	-	-	10	-	14	18	-	-

0	1	2	3	4	5	6	7
-	-	-	-	-	-	-	-

near

t

	0	1	2	3	4	5
0						
1						

// Only either of lower or upper triangular matrix will be sufficient

```
#define  I  32767
```

Largest possible integer value

```
int cost[8][8], near[8], t[2][6];
```

Initialize as {I}

Initialize as the given matrix (Replace '-' with 'I')

INITIAL PROCEDURE

```
void main()
{
    int i, j, k, u, v, n=7, min = I;

    for (i=1; i<=n; i++)
    {
        for(j=i; j<=n; j++)
        {
            if(cost[i][j]<min)
            {
                min = cost[i][j];
                u=i, v=j;
            }
        }
    }

    t[0][0] = u;
    t[1][0] = v;

    near[u] = near[v] = 0;

    for(i=1; i<=n; i++)
    {
        if(near[i]!=0 && cost[i][u] < cost[i][v])
                near[i]=u;
        else
                near[i]=v;
    }
```

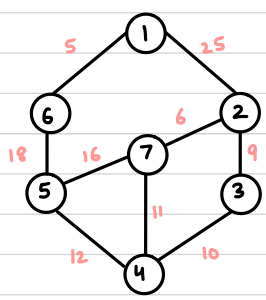
For accessing upper triangular matrix only

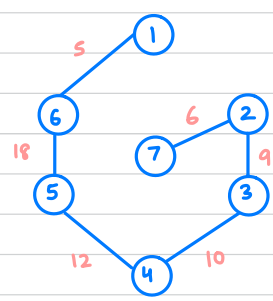
RECURSIVE PROCEDURE

```
    for(i=1; i<n-1; i++)
    {
        min = I;

        for(j=1; j<=n; j++)
        {
            if (near[j]!=0 && cost[j][near[j]]<min)
            {
```

```
                    min = cost[j][near[j]],
                    k = j;
                }
            }

            t[0][i] = k;
            t[1][i] = near[k];
            near[k] = 0;

            for (j=1; j<=n; j++)
            {
                if ( near[j]!=0 && cost[j][k] < cost[j][ new[j]])
                    near[j] = k;
            }
        }

        Print t;
    }
```

KRUSKAL'S METHOD





STEP 1 : Select the edge with least weight along with its vertices

STEP 2 : Select the next minimum edge if it is not forming a cycle

DISJOINT SUBSET

$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Consider its two subsets

$A = \{3, 5, 9\}$

$B = \{4, 7, 8, 10\}$

$A \cap B = \phi$ // Disjoint subset

A = { 3, 5, 9 } B = { 4, 7, 8, 10 }

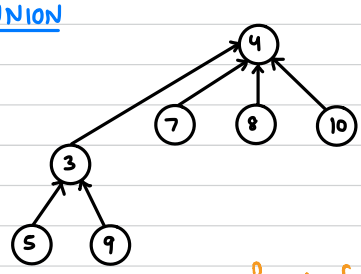
0	1	2	3	4	5	6	7	8	9	10
			-3	-4	3		4	4	3	4

No of nodes in negative
(4 is having 4 nodes)

Rest are representing parent

Representation of set

(1) UNION



The parent with more number of nodes

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

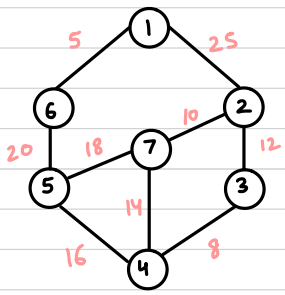
Parent of nodes

```
void Union (int u, int v)
{
    if ( s[u] < s[v])
    {
        s[u] = s[u] + s[v];
        s[v] = u;
    }
    else
    {
        s[v] = s[u] + s[v];
        s[u] = v;
    }
}
```

(2) FIND

```
int Find (int u)
{
    int x = u;

    while ( s[x] > 0)
        x = s[x];

    while ( u != x)
    {
        v = s[u];
        s[u] = x;
        u = v;
    }
}
```

connecting vertices directly to parent



	0	1	2	3	4	5	6	7	8
0	1	1	2	2	3	4	4	5	5
1	2	6	3	7	4	5	7	6	7
2	25	5	12	10	8	16	14	20	18

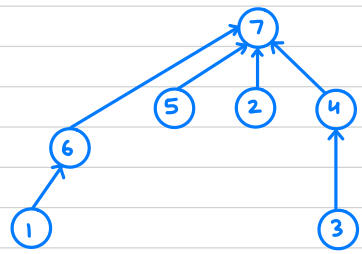
edges

x	6	7	4	7	7	-2	-5
0	1	2	3	4	5	6	7

set

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

included



t

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

```
#define  I  32767

int  edges[3][9] = { { 1,1, 2,2, 3,4,4, 5,5},
                     { 2,6, 3,7, 4,5,7, 6,7},
                     { 25,5, 12,10, 18,16,14,20,18}};

int  set[8] = {-1};
int  included[9] = {0} , t[2][6];


void main()
{
    int i=0, j, k, n=7, e=9, min, u, v;

                    ← No of vertices -1
    while (i < n-1)
    {
        min = I;
                        For finding minimum cost edge
        for (j=0; j<e; j++)
        {
            if ( included[j] == 0  && edges[2][j] < min)
            {
                min = edges[2][j];
                k = j;
                u = edges[0][j];
                v = edges[1][j];
            }
        }
```

```
        if ( find (u) ! = find (v))
        {
            t[0][i] = u ;
            t[1][i] = v ;
            union ( find (u) , find (v) ) ;
            i++ ;
        }

        included[k] = 1;
    }
}
```

To check if cycle is forming or not

ASYMPTOTIC NOTATION

$$f(n) = \sum_{i=1}^{n} i \quad - \quad 1+2+3 \ldots\ldots +n = \frac{n(n+1)}{2} = O(n^2)$$

We can get exact value

$$f(n) = \sum_{i=1}^{n} i \times 2^i$$

Exact value is not possible that is simplified

So, we use asymptotic notations

Lower	Bound	Ω	Omega	less than or equal to
Upper	Bound	O	Big O	Greater than or equal to
Tight	Bound	θ	Theta	equal