DATA STRUCTURE USING C

- Way of organising data
- Data should be efficiently used by the program

VARIOUS DATA STRUCTURES

- ARRAY
- MATRIX
- LINKED LIST

→ Physical Data Structure (How data is arranged)

- STACK
- QUEUE
- TREE
- GRAPH
- HASHING

STACK, QUEUE → Linear
TREE, GRAPH → Non Linear
HASHING → Tabular

→ Logical Data Structure (How data is utilized)

- RECURSION
- SORTING TECHNIQUE

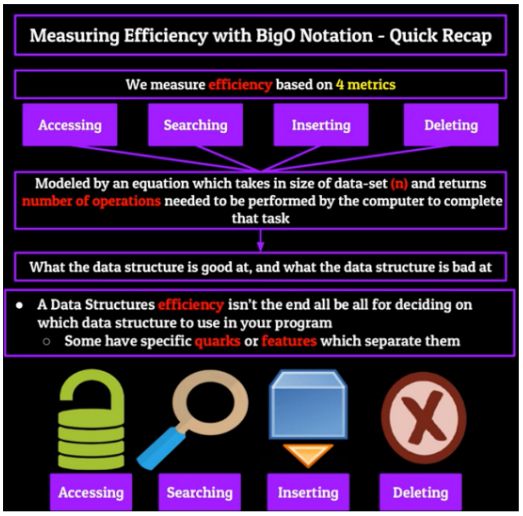
Logical Data structures are implemented using physical Data Structures

Data Structure

- Accessing
- Searching
- Inserting
- Deleting

} Report card of data-structure. [used in judging data structure]

↳ BigO notation ⇒ no of operations done to complete a function.



best → good → good → bad → worst

good: $O(k)$, $O(\log n)$, $O(n)$

bad: $O(n \log n)$, $O(n^2)$, $O(2^n)$

↑ better time complexities.

Arrays.

- An array is fundamentally a list of similar values
- Can be used to store anything
  - Usernames
  - High Scores
  - Prices
- Store values of the same type
  - Integer
  - String
  - Float

} homogeneous.

array → name, type → type of elements, size

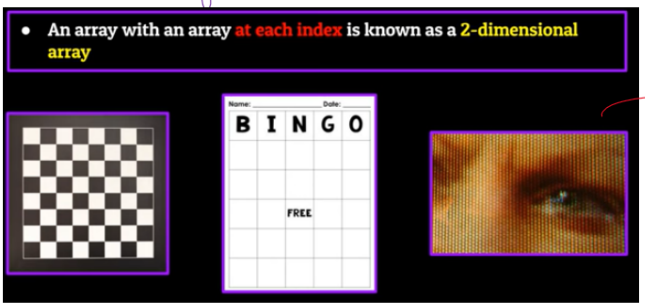
parallel array.

↳ 2+ array w/t same size & have corresponding values in same posit^n.

↳ size of array should be constant / fixed & can't be changed (variable).

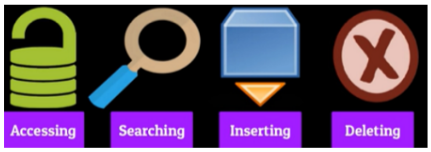
↳ array is zero indexed.

- An array with an array at each index is known as a 2-dimensional array



→ we can even create images.

↳ Array as a data structure.



O(1) O(n) O(n) O(n)

given we have extra space / size.

↑ linear search, because most of the time we will get unsorted array.

Pros	Cons
Good for storing similar contiguous data	Size of the array cannot be changed once initialized
O(1) Accessing Power	Inserting and Deleting are not efficient
Very basic. Easy to learn and master	Can be wasting storage space

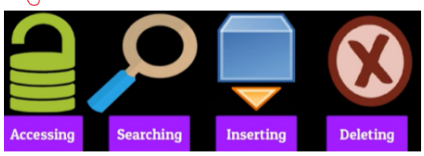
↳ binary search : $O(\log n)$

ArrayList (Vector) (Dynamic Array)

↳ backed by an array (implemented by array behind the scenes)

↳ inbuilt functions:
- add → push_back( ), insert( )
- remove → pop_back( ), erase( )
- get, set, clear, toArray

↳ Arraylist as a data structure.



O(1) O(n) O(n) O(n)

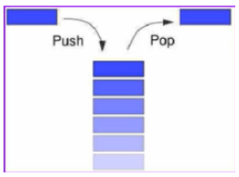
Arrays	ArrayLists
Fixed Size	Dynamic Size
Can store all data types	Can only store Objects
Methods need to be created	Methods are created for you
Doesn't require much memory use or upkeep	Requires more memory use and upkeep

Stack

↳ each element is dependent on the others

Random Access Data Structures	Sequential Access Data Structures
Provides O(1) Accessing	Do not provide O(1) Accessing
Independent Elements	Dependent Elements
Arrays and ArrayLists	Stacks, Queues, Linked Lists

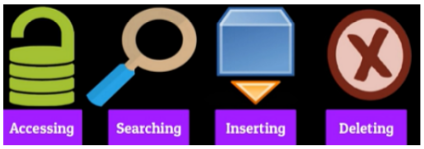
- Stack is LIFO. → last in first out.



- push → at top
- pop → from top
- peek → to see top element
- contains → search

↳ everything in stack can only be done from top.

- Stack as a datastructure.



O(n) O(n) O(1) O(1)

can only be done from top. } only one place

note: stacks are used in recursion, undo, redo, back paging.

Queue

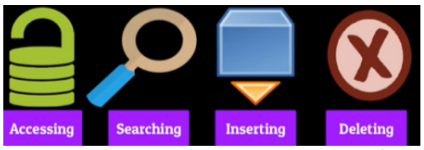
↳ sequentional access DS that follows FIFO.
↳ first in first out.

↳ add element from back (tail) & remove from front (head)

↳
- Enqueue → add from tail
- Dequeue → remove from head
- peek → element at head
- contains → search

note: queue w/t 1 element has head & tail at same position.

- Queue as a data structure



O(n) O(n) O(1) O(1)

↓ from tail ↓ from head } just one place

note: queue is used for (CPU) job scheduling, (multiple pages) printers, zero shutter lag in google pixels.

(A) Linked List

↳ sequential access linear DS.

↳ every element is seperate obj called (node) → it has 2 parts.

↳ last node / tail node always point to null value.

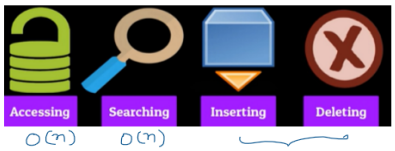
(a) adding and removing nodes

↳ from head node : (a) set new node to point to current head.
(b) set current head to point to null.

↳ at middle node : (a) make pointer of the new node point to node after the locatⁿ we want to insert at, set the node before the locatⁿ we want to insert at to point toward new node.
(b) make pointer of the prev. node we are removing to point to the node after one we are removing & make node to be removed to point to null.

↳ from tail node : (a) make current tail point to new node we want to add & make new node point to null.
(b) make node before tail to point to null.

(b) Linked list as a DS.



O(n) O(n)

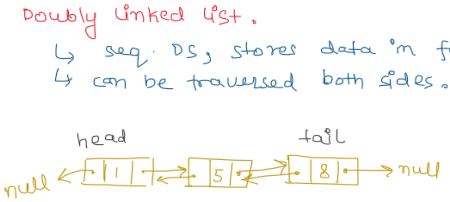
O(1) → if from head / tail.
O(n) → if in b/w

note: linked list is used in backing of DS.
↳ eg. stack, queues etc.
↳ just like array used for arraylist.
(a) songs in music player.
(b) movie player.

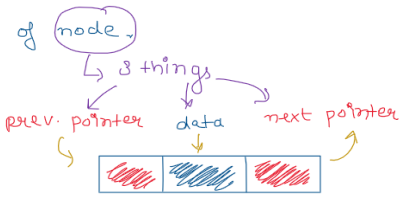
note: one drawback is we can only go forward not backward.

(B) Doubly linked list.

↳ seq. DS, stores data in form of (node).
↳ can be traversed both sides.



↳ 3 things



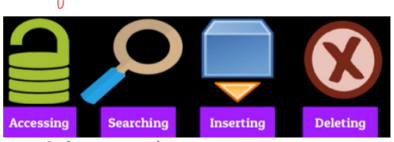
(a) adding & removing from doubly linked list.

↳ from head node : (a) set the new node's next to current head of list & prev. to null, set the current head's prev. to point to this new node.
(b) set the head node's next to point towards a null value & the set second node's prev. to also point toward null.

↳ from middle node : (a) set the new node's previous to point towards the node previous to the positⁿ you wanna insert at, set the new node's next to point towards the node after the position you wanna insert at, set the prev. node's next to point to new node & the next node's prev to point to new node.
(b) set the node before the node to be rem.'s next to point to the node after the one to be rem, set the node after the node to be rem's prev. to point to the node before one to be rem. then set the prev. & next of node to be rem. to point to null.

↳ from tail node : (a) set current tail's next to point to new node, set new node's prev. to current tail, set new node's next to null.
(b) set node before current tail 's next to point to null, & current tail's prev. to null as well.

(b) doubly linked list as a DS.



O(n) O(n)

O(1) → if from head / tail.
O(n) → if in b/w

note: doubly linked list can be used for caching in web page, undo, redo etc.

Dictionaries.

↳ also known as maps & associative arrays.
↳ Store data as (key-value) pair.
↳ key must be unique. ↳ can be anything.
↳ each key can have only one value.
↳ values can be duplicates.

Imp ⓐ dictionaries are implemented using hash tables

↳ Hash table

- A **Hash function** will take all the keys for a given dictionary and **strategically map** them to certain index locations in an array so that they can eventually be **retrieved easily**

- The goal of a **good hashing function** is to take in a key and **reliably** place it somewhere in the table so that it can be **accessed later** by the computer

- **Dictionaries** are built upon these **Hash Tables**, and the key's in our key/value pairs are stored in memory **IN** these hash tables at indexes which are **determined** by a **hash function**

Hash collision.

↳ when we try to input already existing keys.

↳ open addressing → **Put the key in some other index location separate from the one returned to us by the hash function** → mostly the next nil index. (X)

↳ close addressing

↳ **Uses Linked Lists to chain together keys which result in the same has value** → a linked list of values in that index

dictionary as a D.S.

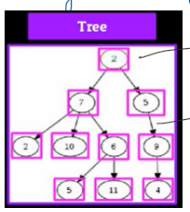
O(n) O(n) O(n) O(n)
O(1) O(1) O(1) O(1) → Avg.

- To access, search for, insert, or delete a **key/value pair** from our dictionary, all we need to do is **run that key** throughout hash function and it'll tell is what index in the hash table to go to in order to perform that operation

Trees (non-linear DS)

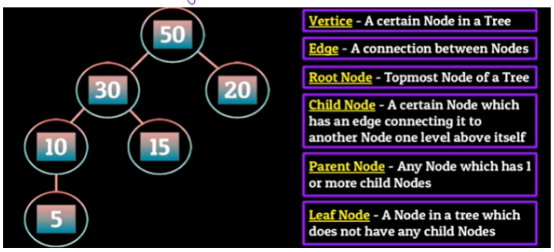
↳ Store data hierarchically. eg: file structure, family tree etc.

- **A Tree...**
  - An **abstract** Data Structure which contains a series of linked nodes connected together to form a **hierarchical representation** of information
- Like a **LinkedList** where each Node has the option of pointing towards **multiple Nodes**

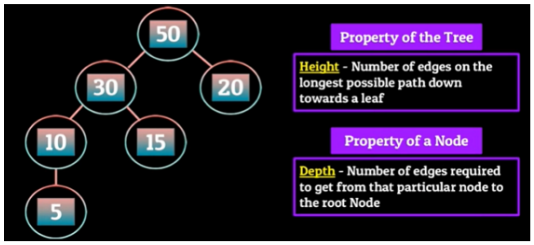


also called Vertices
also called edge

Terminology



note:



BST.

- A **Binary Search Tree** is a simple variation on the standard tree which has **three restrictions** on it to help organize the data
  - A Node can have at most 2 children
  - For any given **parent Node**, the child to the **left** has a value **less** than or equal to itself, and the child to the **right** has a value **greater** than or equal to itself
  - No 2 Nodes can contain the **same value**

↳ Searching in O(logn) →

- Makes them extremely popular for storing **large quantities** of data that need to be easily searchable
  - Also translates to accessing, inserting, and deleting Nodes

Trie. → (only used in specific situation)

Imp

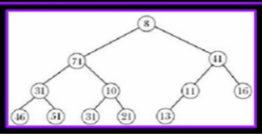
Trees can become even more useful once you start **setting restrictions** on **how** and **where** data can be stored within them

- A **Trie** is a **tree-like** data structure whose nodes store **letters** of an alphabet in the form of **characters**
- We can **carefully construct** this tree of characters in such a way which allows us to quickly **retrieve words** in the form of Strings by traversing down a path of the trie in a certain way

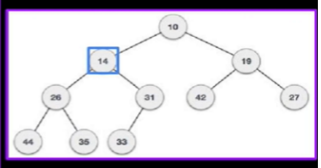
eg: used in autocorrect, spellcheck.

⑦ Heap. (kind of like BST).

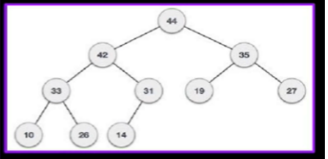
- **A Heap**
  - A special tree where all parent Nodes compare to their children Node's in some specific way by being more or less extreme
    - Either greater than or less than
    - Determines where the data is stored
    - Usually dependent on the parent Node's value



- **Min-Heap**
  - The value at the Root Node of the tree must be the minimum amongst all of its children
  - This fact must be the same recursively for all parent Node's contained within the Heap



- **Max-Heaps**
  - The value at the Root Node of the tree must be the maximum amongst all of its children
  - This fact must be the same recursively for all parent Node's contained within the Heap



Deleting From the Root Node

1. Remove the root node from our Heap
2. Replace it with the Node furthest to the right
3. "Heapify" the Heap by comparing parent Node's to their children and swapping if necessary

⑤ uses.

- Heaps are most commonly used in the implementation of **HeapSort**
  - A sorting algorithm which takes in a list of elements, builds them into a min or max heap, and the removes the root Node continuously to make a sorted list

↳ coz we start getting min/max elements.

- **Priority Queues**
  - An advanced data structure which your computer uses to designate tasks and assign computer power based on how urgent the matter is

⑥ graphs.

- **A Graph**
  - A nonlinear Data Structure consisting of Nodes and Edges
    - Finite set of Nodes (Vertices)
    - Nodes are connected by the edges

→ no specified starting point unlike tree.

↳
- **An Undirected Graph**
  - A Graph in which the direction you traverse the Nodes ISN'T important
    - Usually indicated by a lack of arrows

eg: friends in facebook.

↳
- **An Directed Graph**
  - A Graph in which the direction you traverse the Nodes IS important
    - Usually indicated by a arrows representing direction

eg: followers in insta

↳
- **A Cyclic Graph**
  - One which contains a path from at least one Node back to itself

→ all undirected graph.

↳
- **An Acyclic Graph**
  - One which contains no path from any one Node which leads back in on itself

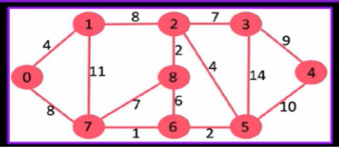
→ only directed graph, but not all.

- **Weighted Graphs**
  - Associating a numerical value with each edge (Cost)
  - Each weight represents some property of the information you're trying to convey

also unweighted.

⑧ w/t their combo different graphs are formed.

eg:
- **Undirected Cyclical Heaps with weighted Edges** can be used through **Dijkstra's shortest path algorithm**
  - Compiles a list of the shortest possible paths from that source vertex to all other Nodes within the Graph



- **Unweighted Cyclical Graphs (Undirected and Directed)** are used in the follower system of a majority of social media websites
  - Facebook, Snapchat, Instagram, Twitter, etc.

eg:

eg: g-maps, ip-routing, telephone routing.

3) Containers.

Array.

type, size
↳ array<int, 3> arr;
name

Note:
↳ if we define anything globally it will be initialized w/t '0'.
↳ if we define inside int main it will be initialized w/t garbage value

Initializing array w/t zero (1D, 2D etc)

int a[100] = {};
array<int, 100> a = {};
specify size then use {}.

↳ fill() method to fill entire array with 1 same value.
eg. arr.fill(1); (fill entire array w/t 1)

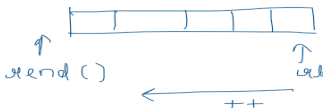
↳ accessing members of array container.
using at() method.
eg. arr.at(1); // similar to arr[1]
index.

Note: Iterators. (kind of pointer that give address).
a) begin()
b) end()
c) rbegin()
d) rend()
reverse begin.
reverse end.

↳ used to iterate through container.
↳ let the iterator be i so to get the value from i use *i (dereferencing)

eg: array<int, 5> a = {1, 2, 3, 4, 5};
for (auto i = a.begin(); i != a.end(); i++) {
cout << *i << " ";
}
1 2 3 4 5
auto: take type of a.begin() (iterator)
i++: just like pointer arithmetics

Note: If we are working w/t reverse iterator i++ means going backward.
eg. for (auto i = a.rbegin(); i > a.rend(); i++) {
cout << *i << " "; // prints array.
}



[∵ they are reverse iterator]

Note: auto i = □
↳ take the type of & assign it to i & create it.

Note: other imp. fn. arr.size();
arr.front(); // arr.at(0);
arr.back(); // arr.at(arr.size()-1);

↳ max size of int, double, char array → globally → 10^7.
int main() → 10^6.
↳ bool → globally → 10^8.
int main() → 10^7.

vector (dynamic array)

vector<int> v;
name, type
dynamic array
can dyamically increase till

↳ Accessing elements. v[0] (or) v.at(3)
index

↳ size. v.size();

↳ adding & removing elements. (NOT WITH ARRAYS)
↳ v.push_back(3); ↳ add 3 at the end
↳ v.pop_back(); ↳ remove last element.
↳ v.clear(); ↳ erase the entire vector.

↳ interesting declarat^n.
what to initialize them w/t.
vector<int> v(5, 3);
type, size, initilize entire vector w/t this initially.

vector<int> v1(v.begin(), v.end());
vector<int> v2(v);
like slicing, don't include v.end() value.
copy entire v in v1 & v2.

vector<int> v = {1, 2, 3, 4};
vector<int> v3(v.begin(), v.begin()+2);
↳ {1, 2}
not include this.

Note: emplace_back() is slightly faster than push_back().

↳ lower bound, upper bound, swap(v1, v2), clear(), begin(), end(), rbegin(), rend().

2-D vector.

```
vector<vector<int>> v;
```
↳ a vector v that will contain vector of integers.

↳ accessing. `v[m][n];` // w/o index.

→ individual vectors inside can be of different size unlike 2D array.
↳ for every row we can have different columns.

↳ printing. (m-1)
```
for (auto i: v) {
    for (auto j: i) {
        cout << j << " ";
    }
    cout << endl;
}
```
i is of type vector<int>.
j is of type int.

(m-2)
```
for (int i=0; i<v.size(); i++) {
    for (int j=0; j<v[i].size(); j++) {
        cout << v[i][j] << " ";
    }
    cout << endl;
}
```
v.size() → rows of vector
v[i].size() → column of ith row of vector

↳ interesting declaration.

```
vector<vector<int>> v(10, vector<int>(5,3));
```
↳ Create a vector v that will contain vector of integers.
↳ in v form 10 index & in each index add a vector of int with 5 index each initialized w/o 3.
(10 → rows, 5 → columns)

Note. Container that only contain unique elements is Set, unordered_set.

(*) Set. (set contain unique elements & is unindexed)
↳ all operation is log n.
↳ `set<int> s;`
↳ `s.insert(5);`) if we insert multiple elements it will take O(log n)

(#) store unique elements in linear ascending way.
↳ `s.erase(s.begin());` → delete element at this iterator.
↳ `s.erase(s.begin(), s.begin()+2);`
↳ to erase a range of elements from s.begin() till s.begin()+2. excluding
↳ `s.erase(5);` → element of set.
↳ `s.find(5);` // O(log n)
↳ return an iterator.

(O(log n))

↳ s.size(), s.emplace() is faster than s.insert(), s.clear()

(*) unordered_set.
↳ order is not remembered here. (no fashion of storing).
↳ Average time of operation is O(1)
↳ Operation same as set
↳ O(n) (worst one)
↳ if ascending order isn't req. use unordered_set.

(*) multiset.
↳ help to store elements in sorted order.
↳ methods same as set.
↳ doesn't store unique elements.
} O(log n)

map
- stores key - value pair.
  - key → unique.

```
map<string, int> m;
m["Dheeraj"] = 100;
```

- map stores everything in sorted order of "key"

```
m.emplace("susu", 10);
```

- .erase( ) ← can take iterator or key.
- .clear( )
- .find( ) ← key.
- .empty( ) ← return boolean.
- printing:

```
for(auto i: map)
cout << i.first << " " << i.second << endl;
```

i is a pair type here.

unordered_map (no order)
- O(1) in all cases
- O(n) in worst.

pair
- can only have 2 things.

```
pair<int, int> p;
p.first = 10;
p.second = 20;
```

} this is kind of defined type

- nested pair.

```
pair<pair<int, int>, int> p = {{1, 2}, 3};
cout << p.first.second; // 2.
```

multimap → sorted & multiple keys. ✓

stack and queue (no iterators)
- first in - last out.
  - `stack<int> s;`
  - push / emplace, pop, top, size, empty (→ bool)
    - top → last entered element.
    - don't use in empty stack.
- first in - first out.
  - `queue<int> q;`
  - push, pop, front, size, empty. } O(1).

priority_queue
- stores everything in sorted way
  - all fn have O(1).
  - `priority_queue<int> pq;`
  - push, pop, top, size, empty.
  - by default stores in descending order (heighest at top).
- minimum priority_queue (only declaration syntax is different).

```
priority_queue<int, vector<int>, greater<int>> pq;
```

- this time min at top (stores in ascending orders).
- should be same.

dequeue
`dequeue<int> d;`
- can work on both direction.
- push_front( ), push_back( ), pop_front( ), pop_back( ), begin( ), end( ), rbegin( ), rend( ), size, clear, empty, et.

list
`list<int> l;`
- push_front( ), push_back( ), pop_front( ), pop_back( ), begin( ), end( ), rbegin( ), rend( ), size, clear, empty, et, remove( ).
  - remove( ): O(1) → remove any element.

bitset
- takes only 1 bit (yes 1 bit not 1 byte)
- very low space.
- `bitset<4> b;` (take a 4 bit thing) (only stores 0 or 1) (set bit / unset bit)
  - eg. 0110

```
b.all();   // if all set bit then true else false
b.any();   // if any bit is set bit then true else false.
b.count(); // count no. of set bit.
b.flip();  // flip all bits (1's complement).
b.flip(2); // flip the bit at index 2.
b.none();  // if no set bit then true else false
b.set();   // all bit gets set bit
b.set(,);  // what to set the bit at that index.
           // index
b.reset(); // all bits become 0.
b.reset(2); // bit at index 2 is 0.
b.test(1); // check if bit is set or not at index 1.
```

✪ Sorting. O(n log n)

↳ sort (arr, arr+n)
↳ sort (v.begin(), v.end())

Imp
sort (~~, ~~, ~~)
↑ start ↑ end (not inc) ↳ iterator.
(comparator how to sort ↓)

✪ Reverse.

↳ reverse (~~, ~~)

✪ max of container / min

↳ max_element (~~, ~~)
min_element (~~, ~~)
} give iterator of max & min so to get the element use *.

✪ Sum of all elements.

↳ accumulate (~~, ~~, ~~);
↑ start iterator ↑ end iterator ↑ initial sum.

✪ count a particular element.

↳ count (~~, ~~, ~~)
↑ element to count

✪ First occurance of an element.

↳ find (~~, ~~, ~~)
↑ returns iterator. ↑ element to find.

✪ Binary search. (only works on sorted array) (O(log N))

↳ binary_search (~~, ~~, ~~)
↳ returns bool (Yes/No)

✪ lower_bound (only work in sorted array)

↳ return an iterator that points to the 1st element that is not less than x.

✪ upper_bound (only work in sorted array)

↳ return an iterator that points to the 1st element that is just more than x.

✪ Permutation.

↳ next_permutation (~~, ~~)
↳ prev_permutation (~~, ~~)
} Change the main container to next & prev permutatn & if done returns true else false

v.imp permutatn are done in sorted order.

BASICS

1. Arrays
2. Structure
3. Pointer
4. Reference
5. Parameter Passing
6. Classes
7. Constructor
8. Templates

ARRAYS

→ Collection of similar datatypes

STRUCTURES

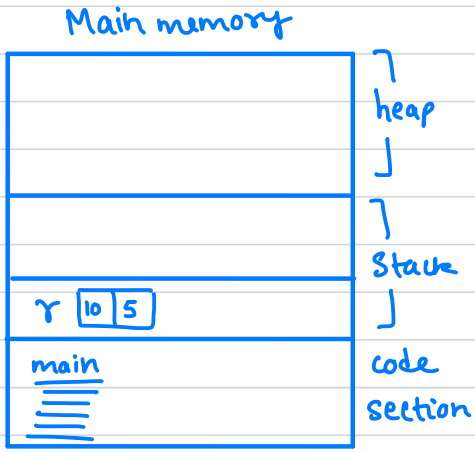
→ Collection of different data types

```
Struct rectangle
{
    int length;     — 2
    int bredth;     — 2
};                    4 bytes
```

but till here, it does not occupy any space in memory

```
int main()
{
    Struct rectangle r = {10,5}
    r.length = 10;
    r.bredth = 5;
}
```

→ here it occupies memory



```
Struct card
{
    int face;
    int shape;
    int colour;
};

int main()
{
    Struct card deck[52] = {....};
    printf("%d", deck[0].face);
    printf("%d", deck[0].shape);
}
```

Pattern Problems

13 November 2020
13:38

1) Break problem into smaller problem
2) Trust in abstraction.

Tips:

a) Know number of rows & how to print it
b) Know number of columns in a particular column & how to print it.
   b-1) sometime we have space & data in column
   ↳ use different loop for different thing seperate for spaces & seperate for data.

Note:

[Remember only 3 Questns needed.
↳ how many rows
↳ how many columns in each row
↳ what to print in each column]

Imp

If in a particular row data - space - data become a lot to handle eg: 3 data, 3 spaces (6 loops total)

↳ In these situations try to find a pattern & use if-else.

eg:

n = 5.

pattern. for $i < (5/2+1)$ → if $j = 1$ (or) n then print.

for $i >= (5/2+1)$ → if $j = 1$ (or) n (or) $j = n/2 - x$ (or) $j = n/2 + x$

then print

↳ here $x = 0$ the 1 the 2 & so on.

Code:

```
int x=0;
for(int i=1;i<=n;i++){
    if(i<=n/2){
        for(int j=1;j<=n;j++){
            if(j==1||j==n){
                cout<<"*\t";
            }
            else{
                cout<<"\t";
            }
        }
    }else{
        for(int j=1;j<=n;j++){
            if(j==1||j==n||j==n/2+1-x||j==n/2+1+x){
                cout<<"*\t";
            }
            else{
                cout<<"\t";
            }
        }
        x++;
    }
    cout<<endl;
}
```

Cases for columns

⭐ Computer only understand binary, $(\ )_2$.

⭐ Bit operator.

↳ &, |, ~, ^, >>, <<

- & → and → all 1 then 1 else 0
- | → or → any 1 then 1
- ~ → not → flip bit
- ^ → xor → even 1 = 0; odd 1 = 0
- >> → right shift → division of 2^a → a right most bit removed
- << → left shift → multiplication of 2^a → a left most bit removed. → add a zero to right most side.

⭐ Properties.

↳ $a \wedge 0 = a$

↳ $a \wedge a = 0$

↳ swaping numbers using ^

↳
```
a = a^b
b = a^b
a = a^b
```
} swapping without a 3rd variable.

↳ If $f(n) = 1 \wedge 2 \wedge 3 \wedge 4 \dots \wedge n$.

↳
- $f(n\%4 == 0) = n$
- $f(n\%4 == 1) = 1$
- $f(n\%4 == 2) = n+1$
- $f(n\%4 == 3) = 0$

↳ if (N & 1 == 0) then even (fastest).

[odd will always have 2^0 → only odd number]

↳ N & 1 gives us the last bit.

Note: 1 → set bit 0 → unset bit

↳ if we wanna check i^{th} bit of N.

↳ (N & (1 << i)) → 0 → i^{th} bit is unset.
→ 1 → i^{th} bit is set.
(1 << i) → mask.

Imp Note: never alter the data given.

↳ Set the i^{th} bit of N.

↳ N = (N | (1 << i)) (1 << i) → mask

↳ Clear the i^{th} bit of N.

↳ N = (N & (~(1 << i))) (~(1 << i)) → mask

↳ Clear last set bit.

↳ N = (N & N-1)

Note: ⭐ if N is a power of 2, then (N & N-1) = 0.

↳ to check if a power of 2

↳ (N & N-1) = 0 → true
↳ else false.

↳ count number of set bits.

POINTERS

↳ address variables

1. Why pointers
2. Decleration
3. Initialization
4. Dereferencing
5. Dynamic allocation

```
int a=10;
int *p; ⟶ decleration
p=&a; ⟶ initialization
printf("%d", a);
printf("%d", *p); ⟶ dereferencing
```

USES OF POINTER

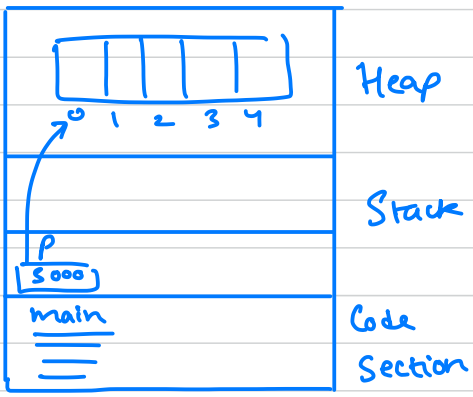
1. Accessing heap
2. Accessing resources like keyboard or mouse.
3. Parameter passing.

HOW TO USE POINTER FOR ALLOCATING HEAP

```
#include<stdlib.h> ⟶ header file for malloc()

int main()
{
    int *p;
    p=(int*)malloc(5*sizeof(int));
}
```

↳ for returning as malloc function returns void pointer (type casting)

↓ Creating space for 5 integer values in heap.



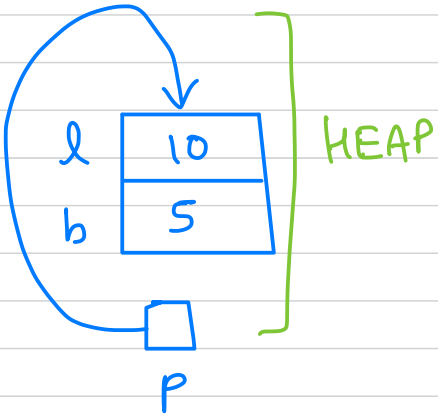
REFERENCE IN C++

```
int main()
{
    int a=10;
    int &r=a;
    cout<<a;
    cout<<r;
}
```

POINTER TO STRUCTURE

```
Struct rectangle
{
    int length;
    int bredth;
};

int main()
{
    struct rectangle r = {10, 5};
    struct rectangle *p = &r;

    r.length = 15;
    (*p).length = 20;
    p -> length = 20;
}
```

doesn't occupy 4 bytes of memory (2 bytes)




```
int main()
{
    struct rectangle *p;

    p = (struct rectangle *)malloc(sizeof(struct rectangle));

    p -> length = 10;
    p -> bredth = 5;
}
```

FUNCTIONS

What are functions ⟶ Performs a specific task

Parameters of passing functions
- → Pass by value
- → Pass by address
- → Pass by reference

Pass by value, Pass by address → Only in C

Pass by reference → In C++

- Pointers are variables that store the address of other variables.

```
//Syntax
//data_type* name_of_ptr
int *myIntPointer;
char *myCharPointer;
```

so as I know how many bytes to read & how to interpret.

Note: adding * in front of any data type make a pointer that stores value of that Data type.
eg: int*, char*, int**
int* → integer pointer; char* → char. pointer; int** → pointer that store address of a integer pointer.

- Every variable is stored in the memory and each memory location has its own memory address. It enables us to pass variables by reference.
  - '&' Operator : It gives the address of the variable.
  - '*' Operator : It gives the value stored at the address.
    i.e dereferences the value stored at the address.

❖ Array Pointer

- In C++, The name of the array is a pointer that points to the first element of the array.

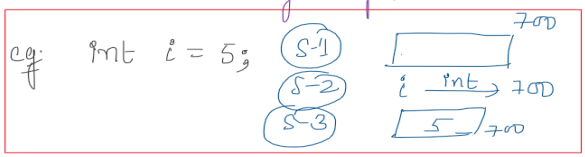
```
int arr[] = {10, 20, 30};
cout << *arr << endl;

int *ptr = arr;
for (int i = 0; i < 3; i++) {
    cout << *(arr + i) << endl;
}
```

- *(arr + i) is equivalent to arr[i].
- (arr + i) is the address of the ith element of the array.

✱ Pointers Introduction

↳ 3 step process → memory is created
- in symbol table during compile time ← entry of memory is done
- filling the memory

eg: int i = 5;



↳ To find the address of the variable
↳ use address of operator i.e. & operator.
eg: &i
↳ address is in hexadecimal form (size of pointer = 8 bytes)

↳ variables that can store address of other variables
↳ these are pointers
↳ To create this special variable we use *.
eg: int *p = &i (or) int* p = &i
int → type of pointer; int* → int type pointer

✱ Accessing Variables

by pointers: p →
by symbol table: i → 700

↳ dereferencing the value stored at the address stored in the pointer can be done using * operator.

```
cout << *p;   // value of i
cout << p;    // address of i
cout << &p;   // address of p
int *q = p;   // address of i (value of p)
```

Note: whenever we declare a pointer we must initialize it.
(or) if we can't initialize it for a moment then initialize it with 0.
i.e. int *p = 0;
↳ null pointer (no value here).

Imp: we should not let a pointer take garbage address.

⊛ Pointer arithemetics

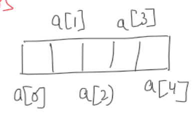
⊛ Size of all the pointers are mostly same, (depend on compiler).

↳
```
int i = 1;
int* p = &i;
p += 1;
```
← here 1 memory size of the pointer type is added. (i.e. 4 bytes in case of int & 1 byte in case of char).

⊛ This help us to move in array (continious memory blocks).
⊛ we can even compare pointers, according to which one is ahead.

⊛ Arrays and Pointers

↳ `int a[5];`



here a is like the pointer that stores the address of a[0].
we can treat a as a pointer.

∴ a has address of a[0]
a+1 has address of a[1]
*a has value of a[0]
*(a+1) has value of a[1]

Note: a[i] ≡ *(a+i) ≡ *(i+a) ≡ i[a]] this is to get the i^{th} element
&a[i] ≡ a+i ≡ i+a ≡ &i[a]] this is to get the address of i^{th} element

Note: Difference b/w pointer & a
↳ `int *p;` — There a new memory is allocated for pointer where it store the address.
`int a[5];` — There the memory is allocated for a[5] & no new memory for pointer a.

↳ sizeof(a) ⟶ gives the whole size of array. (20 bytes here)
↳ a and &a will give the same result i.e. address of a[0].
because a new memory isn't allocated for a so no entry in symbol table.
↳ '&' checks the symbol table for address.
↳ symbol table is read only (एक बार जो entry ho gya so हो गया)
↳ we can't do a = a+3; as a is like a pointer not a pointer.
we can't do pointer arithemetics w/t 'a'. ← we can move like a+i, but can't reassign a like pointers.

⊛ character pointer

```
char a[] = "abc";
cout << a;
```
↳ unlike array it won't give address of a[0] instead it will show the value of array.
this is due to implementation of "cout".
This implementation is on all character pointer, it will keep printing till it finds '\0' (null character).

Note: | `char* p = &a;` ⊗ `char* p = a;` { pointer Jha hai wha se \0 tak sab print hoga }
`cout << p;` ⟶ (abc)

Note: `char s[] = "abc";`]⟶ first we get temperory memory then a permanent memory for array.
`char* ps = "abc";`]⟶ first we get temperory memory then the pointer point to it. (Bad habit)

⊛ functions & Pointers

↳ when we pass array in function actually they are passed as pointers.
↳ we can pass part of array in function, afterall arrays are passed as pointers.
↳ we can also pass pointer in function.

⊛ Double pointer

↳ we specifies the pointer type as it is helpful in dereferencing & pointer arithemetics.
↳ To store address of pointer in a pointer we use double pointer.

eg:
```
int i = 5;
int* p = &i;
int** q = &p;
```

5 ← (i), (p), (*p), (q), (*q), (**q) ⟶ value of address stored in p (value of i) (*p, i)

- (p): address of i (&i)
- (*p): value of i (5) (i)
- (q): address of p (&p)
- (*q): ↳ value of p (address of i) (p, &i)

(6) inline keyword.

↳ this keyword before a fn just copy body & place it everywhere the fn is called.

↳ note: it won't pause main() & put fn in stack it will copy body & paste just like we wrote it everywhere.

eg:
```
inline int min(int a, int b){
    return a>b ? b : a;
}
```

Note: weather to copy body (or) not depend on compiler.
- ↳ 1 liner fn are done
- ↳ 2-3 liner depend on compiler
- ↳ 4+ liner not.

↳ If we make every fn inline our output file gets bulky.

(7) const keyword.

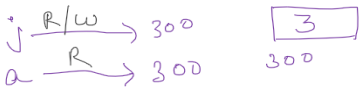
↳ constant variable (ERROR)

↳ `const int i = 5;` // can't change the variable now

Imp ↳ always initilize the const variable at time of declaration

note: a) storage isn't constant path is constant.

eg:
```
int j = 3;
const int& a = j;
```



j path has both read & write access but path a only has read access.

b) A path with read access can't grant write access to a path derived from it.

eg:
```
const int i = 4;
int& a = i;
```

not possible.

↳ i has only Read access so a made from i can't have write access.

↳ A parent can only give what they have.

↳ reference var. from const var should be const.

Note:
```
const int a = 10;   } both are same.
int const a = 10;
const int& b = a;   } both are same
int const &b = a;
const int* c = &a;  } both are same
int const *c = &a;
```

↳ Similarly w/t pointers, a normal pointer can't store address of const. variable.

(1) eg:
```
const int a = 10;
int* p = &a;          (X)   { This say p can have R/W but a only had R access }
const int* p = &a;    (✓)
```

(2) eg:
```
int a = 10;
int* p = &a;          (✓)
const int* p = &a;    (✓)
```

Keep only rem 2 rules.
↳ path is const, not storage.
↳ we can at max provide those access which we have.

Tip: how to read a variable.
↳ read a variable backward.

eg: int a; a is an integer
int const a; a is a constant integer.
int const &a; a is a reference constant integer.
int const *p; p is a pointer pointing to a const integer.
int * const p; p is a constant pointer pointing to an int.
int const * const p; p is a const pointer pointing to a const int.

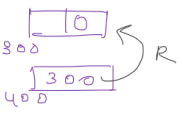
Note: int const *p
↳ p is a pointer pointing to constant integer.
↳ (*p)++ is invalid.
↳ here the thing where p is ponting to is const

vs

int *const p.
↳ p is a constant pointer pointing to an integer
↳ p++ is invalid.
↳ here p is const

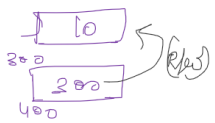
eg: int i = 10;
int const *p = &i;

(R|W) i → 300
(R|W) p → 400



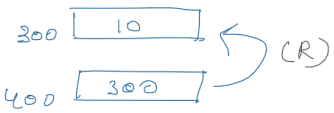
eg: int i = 10;
int *const p = &i;

(R|W) i → 300
(R) p → 400



eg: int i = 10;
int const *const p = &i;

i → 300 (R|W)
p → 400 (R)



* #include <iostream> // preprocessor directives.
↳ happen even before compiling

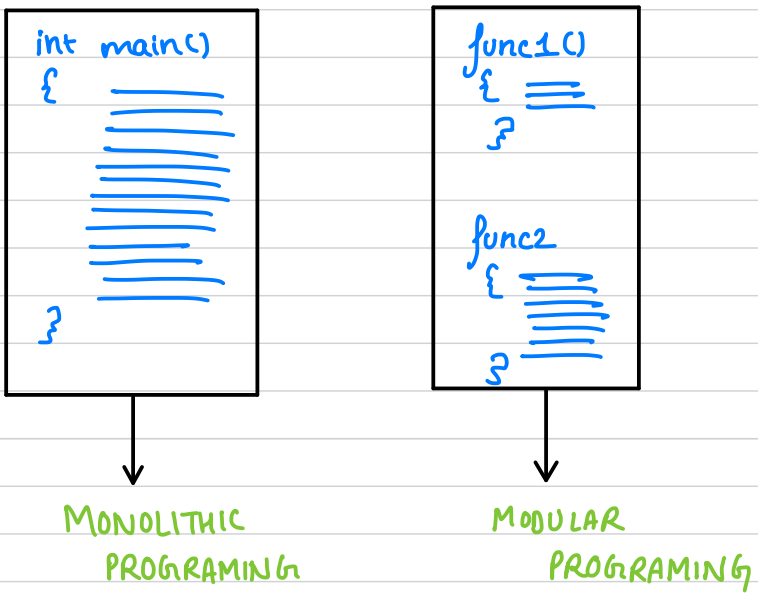
Note: # is macro, (preprocessor of C++).
↳ anything w/t # eg: #include, #define.
↳ preprocessor work before compiling.
eg: #include <iostream> → before compiling this is gonna write the whole iostream code on top.

↳ similarly, #define A = 10
int main() {
cout << A * 5;
}

advantage
↳ value A is at one place so if we wanna change then we have to do it at one place only.
↳ no extra mem is created unlike var.

How it work.
↳ before compiling it's gonna write 10 wherever A is written, just like any other preprocessor.
↳ eg: (scss, styled comp)

Note: remember macro don't end w/ ;.



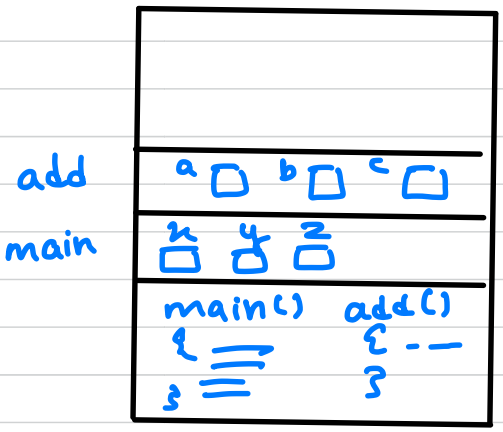
FUNCTION EXAMPLE

```
prototype
int add (int a , int b)
{
        int c;
        c = a+b;
        return c;
 }

int main()
{
        int x, y, z ;
        x = 10;
        y = 5;
        z = add (x, y);
        printf(" %d", z );
}
```

Formal Parameter

Actual Parameter

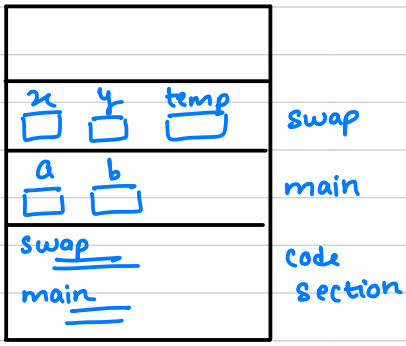
15



CALL BY VALUE

```
Void swap( int x , int y)
{
    int temp;
    x = y;
    y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (a,b);
    printf (" %d %d", a,b);
}
```

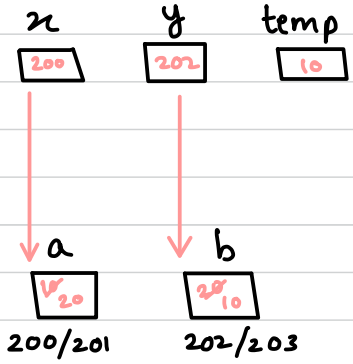
10 20

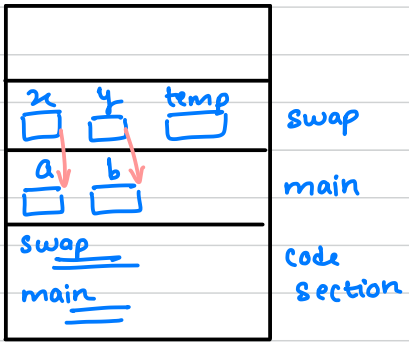


CALL BY ADDRESS

```
Void swap( int *x , int *y)
{
    int temp;
    *x = *y;
    *y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (&a,&b);
    printf (" %d %d", a,b);
}
```

20 10

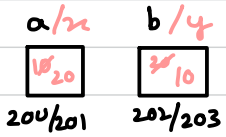


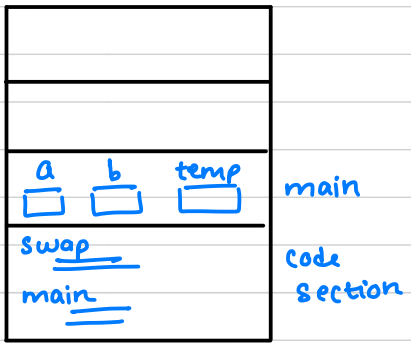


CALL BY REFERENCE (Only in C++)

```
Void swap( int &x , int &y)
{
    int temp;
    x = y;
    y = temp;
}

int main()
{
    int a, b;
    a = 10;
    b = 20;
    swap (a,b);
    printf (" %d %d", a,b);
}
```

20 10





Only 2 bytes of memory is used as in reference, only another name is given to the pre-existing variable

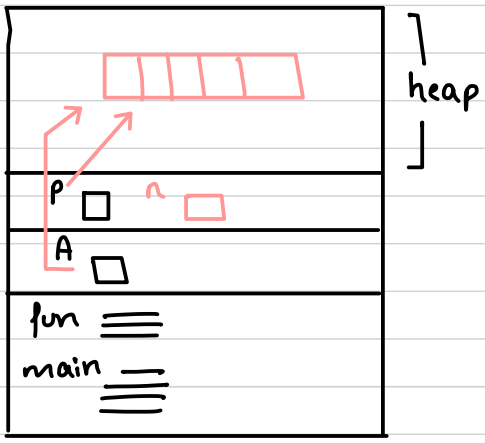
---

ARRAY AS PARAMETER

```
void fun (int A[], int n)
{                  ↳ or int *A
    int i;
    for (i=0 ; i<n; i++)
        printf (" %d", A[i]);
}

int main()
{
    int a[5] = {2,4,6,8,10};
    fun ( A, 5);
}
```

as ARRAYS can only be passed as call by address.

```
        ↱ or  int * fun(int n)
int [] fun(int n)
{       ↳ it returns an array
    int *p;
    p = (int*)malloc(n * sizeof(int));
    return p;
}

int main()
{
    int *A;
    A = fun(5);
}
```



STRUCTURE AS PARAMETER

CALL BY VALUE

```
int area(struct rectangle r1)
{
    r1.length++;
    return r1.length * r1.breadth;
}

struct rectangle
{
    int length;
    int breadth;
};

int main()
{
    struct rectangle r = {10,5};
    printf("%d", area(r));
}
```

CALL BY REFERENCE

```
int area(struct rectangle &r1)
```

CALL BY ADDRESS

```
void changel(struct rectangle *p, int l)
{
    p->length = l;
}

int main()
{
    struct rectangle r = {10,5};
    changel(&r, 20);
}
```

PASSING ARRAYS AS CALL BY VALUE USING STRUCTURES

```
Void fun(struct test t1)
{
    t1.A[0] = 10;
}

int main()
{
    struct test t={{2,4,6,8,10}, 5};
    fun(t);
}
```

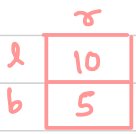
```
struct test
{
    int A[5];
    int n;
}
```

Changes made in the array in the function will not be reflected as it is call by value

---

STRUCTURES AND FUNCTIONS

```
struct rectangle
{
    int length;
    int bredth;
}

void initialize (struct rectangle *r, int l, int b)
{
    r -> length = l ;
    r -> bredth = b;
}

int area ( struct rectangle r )
{
    return r.length * r.bredth;
}

void change1 ( struct rectangle *r, int l)
{
    r -> length = l ;
}
```



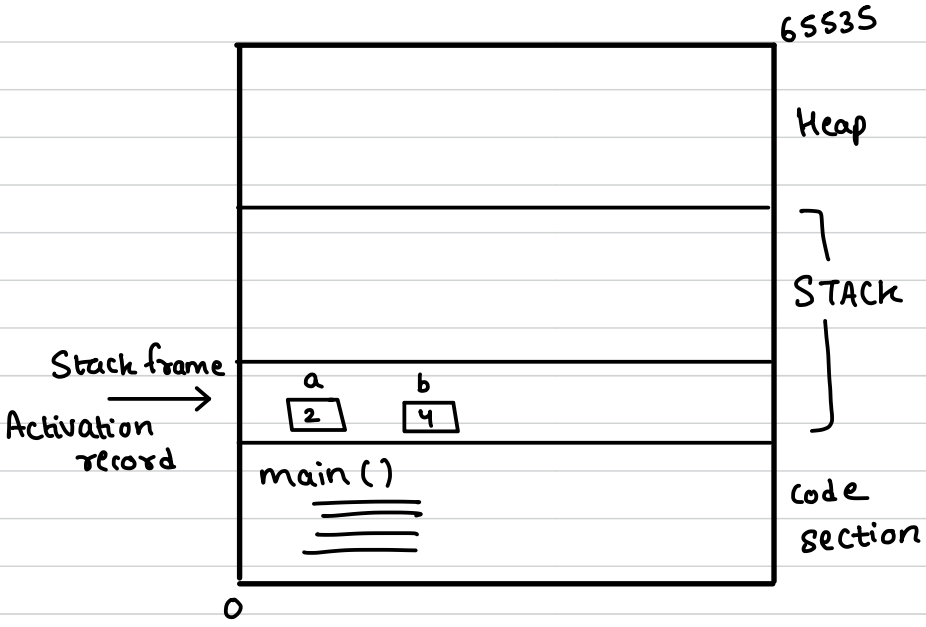

```
int main()
{
    struct rectangle r ;

    initialize (&r, 10, 5);
    area (r);
    change1 (&r, 20);
}
```

because values need to be changed, so call by address.

```
rectangle :: rectangle (int l, int b)
{
        length = l;
        bredth = b;
}

int rectangle :: area()
{
        return length * bredth;
}

int rectangle :: perimeter()
{
        return 2* (length + bredth);
}

rectangle :: ~ rectangle ()
{
}



int main()
{
        rectangle r (10, 5);
        cout << r. area;
        cout << r. perimeter;
        r. setlength (20);
        cout << r. getlength();
}
```

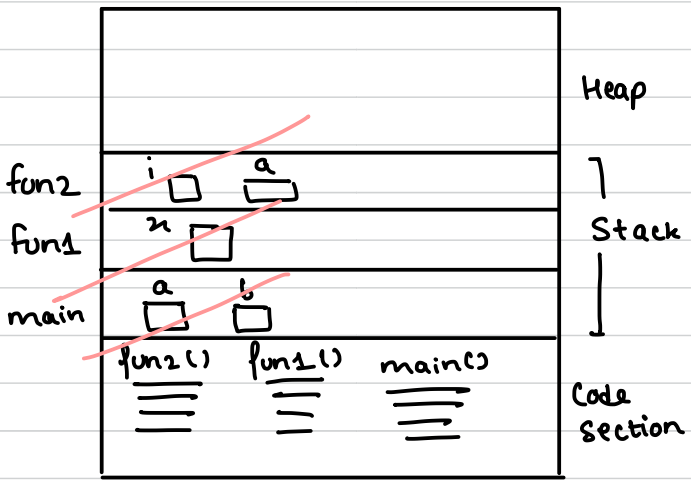
STATIC VS DYNAMIC MEMORY ALLOCATION

```
void main()
{
        int a;
        float b;
}
```



HOW DOES STACK WORK?

```
void fun2(int i)
{
        int a;
        ========
        ========
}

void fun1()
{
        int x;
        fun2(x);
}

void main()
{
        int a;
        float b;
        fun1();
}
```



1. Stack is always organised memory.

❖ Stack Memory Allocation

- The memory is allocated on the function call stack. The memory gets deallocated as soon as the function call gets over. Deallocation is handled by the compiler.

❖ Heap Memory Allocation

- Allocation takes place on the pile of memory space available to programmers to allocated and de-allocate. The programmer has to handle the deallocation.

NOTE: It is different from the heap data structure.

➢ Delete Operator

To de-allocate a memory p, we pass its address to the delete() function.

```
//to de-allocate a memory,
//pointed by pointer 'p'
delete(p)
```

➢ New Operator

New operator is used to allocate a block of memory of the given data type.

```
//Syntax
//myPointer = new <data_type>[size];
int *p = new int[10];
```

❖ Dangling Pointer

- If the memory location pointed by the pointer gets freed/ deallocated, then the pointer is known as the Dangling Pointer.

Dynamic memory allocation.

↳ There are 2 kind of memory → Stack (Static) (Compile time) limited
→ Heap (Dynamic) (Run time) unlimited.

↳ All primitive data types are stored in stack.

eg. `int a[50];` ✓ good — alloting mem statically

`int n; cin>>n; int a[n];` ✗ bad — alloting mem dynamically

Both gonna stored in stack

Note: ↳ we should know the size of array prior to creating, if not create it in heap, eg: vector.

↳ Creating memory in heap.

↳ we use `new` keyword for that

↳ So new keyword will create a memory in heap of specific size and returns the address.

Remember it create mem. & return address, it don't link it to var. for later use, we can store it in pointer for later use.

eg: `int * i = new int;` — pointer i store the address for later use. — create mem. of int in heap

static	dynamic
↳ value in mem. can be accessed directly.	↳ value in mem. can be accessed indirectly.
eg: cout << i;	eg: cout << *i;
↳ mem. deallocated automatically	↳ need to manually deallocate memory.

↳ dynamic memory and array.

allocate addr to pointer: `int * a = new int[10];` → create an int array of size 10 in heap

rest is same as array.

eg:
```
int n;
cin >> n;
int * a = new int[n];
```
→ size of 4×n bytes in heap.
→ size of 8 bytes in stack.

Note:
heap mem. don't have any scope after which it is cleared.
↳ to clear a memory use delete keyword

eg: `delete [] a;`
↳ this delete the address a is pointing to. for array.

Note:
To delete single memory. delete i;
To delete array memory. delete []a;

Remember:
delete don't delete pointer rather it delete the mem. that pointer is pointing to in heap.
↳ pointer is gonna delete automatically as it is stored in stack.

⑧ 2D array dynamically → n=rows, m=columns

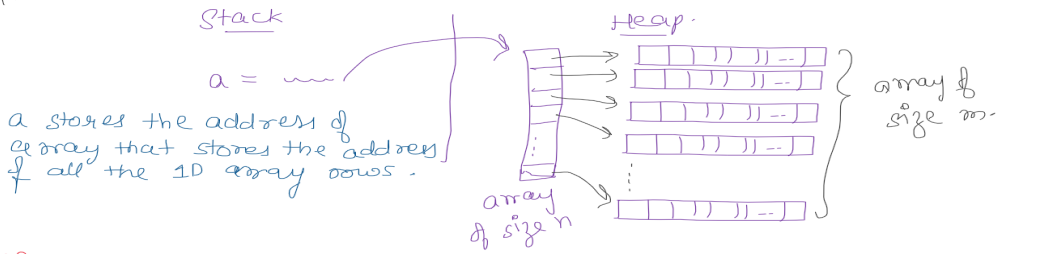
```
cin>>n>>m
int ** a = new int * [n];
for (int i=0; i<n; i++){
    a[i] = new int [m];
}
```

`new int*[n]` → pointer array of size n in heap

a pointer a that will store the single array pointers

loop to iterate thro. each row of a & add a pointer of 2D array.

so overall.



note: as mem. is allocated dynamically so we can create columns of different size

```
int r;
cin >> r;
int ** a = new int* [r];
for (int i=0; i<r; i++){
    int c;
    cin >> c;
    a[i] = new int [c];
}
```

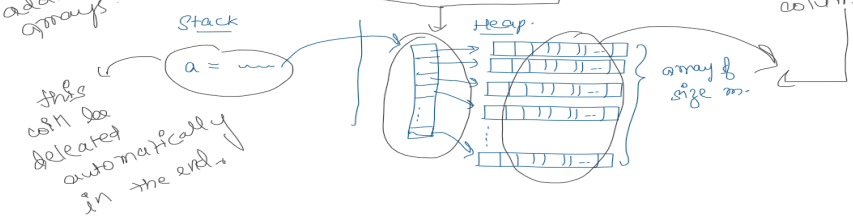
Let say r = 4, c = 1,2,3,4

↳ deallocating mem. in 2D array.

```
for (int i=0; i<r; i++){
    delete [] a[i];
}
delete [] a;
```

this delete the array holding the address of columns arrays

this loop will delete the column mem. one by one



Address typecasting

↳ we can only do explicit type casting here.

eg:
```
int i = 65;
int *p = &i;
char *c = p;        // error (implicit not allowed)
char *c = (char*) p;
```
explicit type casting.

How int is stored.

{little ending, big ending}

int i = 65;

↳ number is stored in mem in reverse order

Reference variable & pass by reference.

↳ (&)

(*) benefits of reference variables
- ↳ no new memory is created for them
- ↳ they take an existing memory & modifing data at that address make changes for original variable.

↳
```
int i = 10;
int& j = i;   // this means in symbol table j will have address of i.
```
Remember no new memory for j, it points to an existing address.

e.g. `int &j; j = i;` ⊗ doesn't make sense as int &j; won't create memory for j so we can later fill it.

NOTE: if creating reference var. always initialize it.

Functions.

↳ functions get soft copy of variable passed (pass by value).

↳ to do pass by reference use ref var.

eg:
```
int f(int& i){
    i++;
}
```
this will not take value but point to address

this will alter value of original variable passed.

NOTE: never make a function return a pointer or an address. Coz when a fn is returned it clear the entire memory of the scope.

↳ returning * (@) & means we will get mem. of something from scope of fn, () which is cleared. (Dangerous)

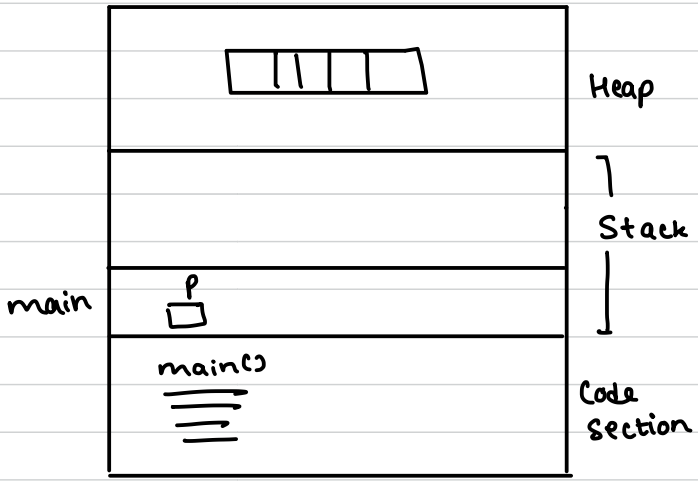
eg:
```
int* f(int a){      ⊗
}

int& f(int a){      ⊗
}
```

(#) be cautious while doing this.

HOW DOES HEAP MEMORY WORK?

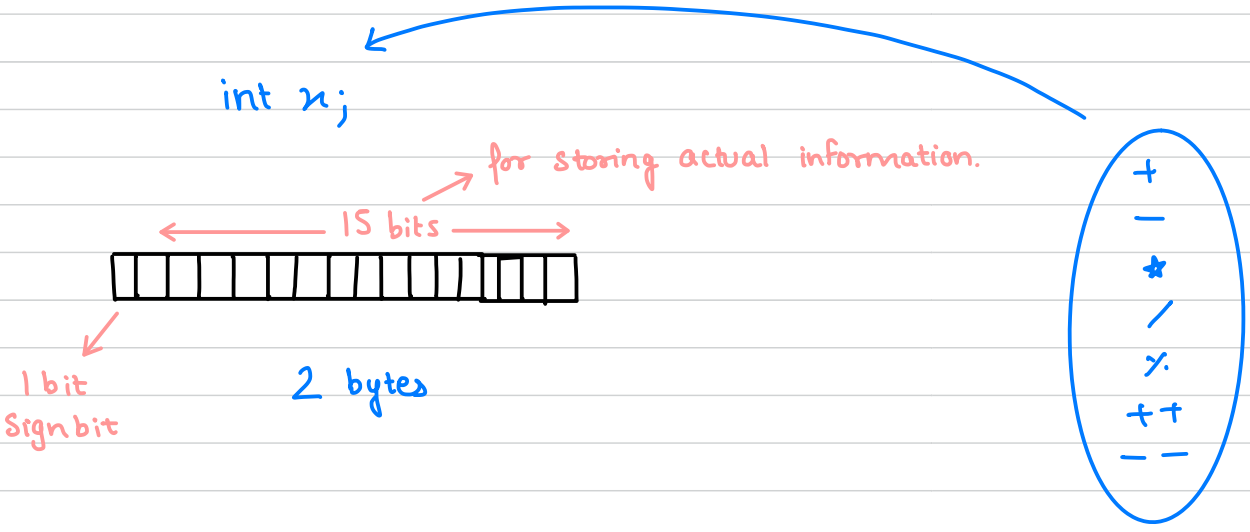
1. Heap may be organised memory or unorganised memory.
2. Heap must be treated as a resource i.e when it is needed, we must use it and when not needed, free it so that it can be used by other applications.
   For eg- printer is a resource.

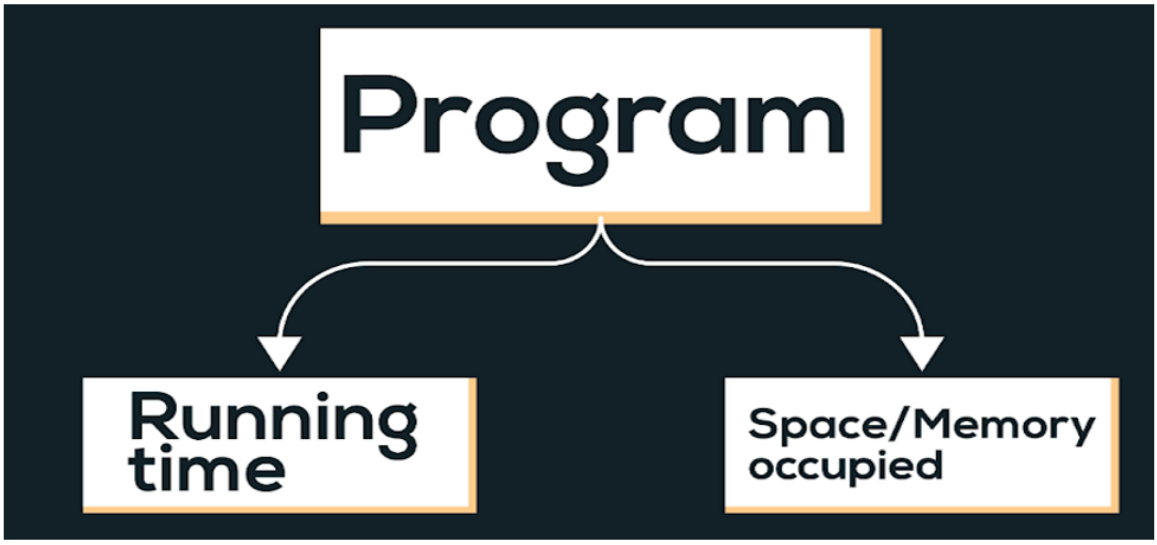
```
void main()
{
        int *p;

for C++ p= new int[5];
 for C  p= (int*) malloc(2*5);
                  ⟶ for array

DEALLOCATION delete []p;
        p= NULL;
}
```



DATATYPE?

↳ 1. Representation of data.
   2. Operation on data.





❖ Time Complexity

- ✓ Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- ✓ Time Complexity of algorithm/code is not equal to the actual time required to execute a particular code but the number of times a statement executes.

Types of notations

- ➢ **O-notation**: It is used to denote asymptotic upper bound.
  For a given function g(n), we denote it by O(g(n)).
  Pronounced as "big-oh of g of n".
  It also known as worst case time complexity as it denotes the upper bound in which algorithm terminates.
- ➢ **Ω-notation**: It is used to denote asymptotic lower bound.
  For a given function g(n), we denote it by Ω(g(n)).
  Pronounced as "big-omega of g of n".
  It also known as best case time complexity as it denotes the lower bound in which algorithm terminates.
- ➢ **Θ-notation**: It is used to denote the average time of a program.

- ✓ Order in case of time complexity
  $O(n^n) > O(n!) > O(n^3) > O(n^2) > O(n.log(n)) > O(n.log(log(n))) > O(n) > O(sqrt(n)) > O(log(n)) > O(1)$

  NOTE : Reverse is the order for better performance of a code with corresponding time complexity, i.e. a program with less time complexity is more efficient.

❖ Space Complexity

- ✓ Space complexity of an algorithm quantifies the amount of time taken by a program to run as a function of length of the input.
- ✓ It is directly proportional to the largest memory your program acquires at any instance during run time.
- ✓ For example: int consumes 4 bytes of memory.

ABSTRACT DATATYPE

↳ hiding internal details

(part of object oriented programming)

What does log mean?

$\log_2 n$ → This gets divided by this until it reaches 1.

Time and Space complexity

```
void swap(x,y)
{
    int t;
    t=x;      ——— 1
    x=y;      ——— 1
    y=t;      ——— 1
}                 3
```

O(1)

```
int sum(int A[], int n)
{
    int s,i;
    s=0;                    ——— 1
    for(i=0; i<n; i++)      ——— n+1
    {        1      n
        s = s + A[i];       ——— n
    }
    return s;               ——— 1
}                           2n+3
```

O(n)

because i will be initialized (+1) and i will be incremented for n no of times and 1 time it will fail too (+n)

// because one time it will fail too.

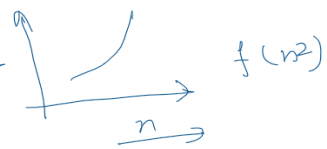
```
Void Add (int n)
{
    int i,j;
    for(i=0;i<n;i++)          ——— n+1
    {
        for(j=0;j<n;j++)      ——— n*(n+1)
        {
            c[i][j] = A[i][j] + B[i][j]; ——— n*n
        }
    }
}
```

because the nested loop will run again and again for (n+1) no of times.

$f(n) = 2n^2+2n+1$

$O(n^2)$

- Time complexity analysis.
  - experimental analysis (using graph)
    - we check it for different data



note: merge sort is a lot faster than selection sort.

note: analysis of time complexity is important when n is very large.
- ∴ we rounds off.

eg: $O(3n) = O(n) = O(5n+6)$.

eg: $O(n^2) = O(3n^2) = O(5n^2+6n+80)$.

- we focus on only highest powers.
- we don't care about coefficients.

- we do theoritical analysis mostly.

- For recursive algorithms.
  - we take into account the time taken for only n inputs & assume we are given for n-1 i.e. T(n-1).
    - then we form a telescopic series to get the answer.

find T(n) for 1 call & assume smaller ones already given.

- fibonacci by recursion is $O(2^n)$.
- Space complexity.
  - we only take auxiliary space (not input space).
  - we take the max. space at any point of time in program.
  - in case of reccursn we don't take max, we take sum.
    - [∵ in reccursn fn is paused so the local scopes aren't cleared].

↳ master theorem for divide & conquer Recurrance

CLASS AND CONSTRUCTOR

```
class rectangle
{
    private:
        int length;
        int bredth;
    public:
        rectangle (int l, int b)
        {
            length = l;
            bredth = b;
        }

        int area()
        {
            return length * bredth;
        }

        void changelength (int l)
        {
            length = l;
        }
};
```

```
int main()
{
    rectangle r (10, 5);
    r. area()
    r. changelength (20);
}
```

```
Class rectangle
{
    private:
        int length;
        int bredth;
    public:
        rectangle()
        {
            length = bredth = 1;
        }
        rectangle (int l, int b);
        int area();
        int perimeter();
```

constructors (overloaded)

facilitators (They do operations on data members)

Accessor

```
int getlength
{
    return length;
}
```

Mutator

```
void setlength (int l)
{
    length = l;
}
```

Destructor →

```
~ Rectangle();
};
```

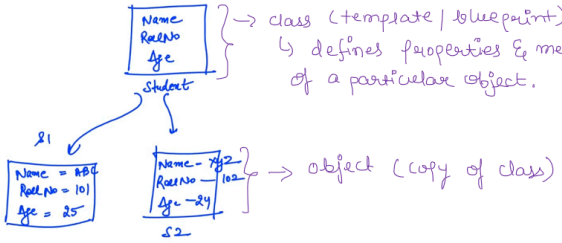
TEMPLATE CLASS

→ for using various datatypes. Using the same class for different datatypes.

```
template < class T >
class arithmetic
{
    private :
        T int a;
        T int b;

    public :
                        T     T
            arithmetic (int a, int b)
        T int  add ();
        T int  sub ();
};

template < class T >  ⟶ because effect of previous template has ended
arithmetic <T> :: arithmetic (int a, int b)
                              T     T
{
    this . a = a;
    this . b = b;
}
template <class T>
T int  arithmetic <T> :: add()
{
    T int c;
      c= a+b;
      return c;
}

template < class T >
int  arithmetic <T> :: sub()
{
    T int c;
      c= a-b;
      return c;
}
```

```
int main()
{
    arithmetic < int > ar(10, 5);
    cout << ar. add();
    arithmetic < float > ar1 (1.5,1.2);
    cout << ar1 . add();
}
```

① class is blueprint & object as real thing.
↳ every object will have some
- property → (class / obj var).
- method → (class / obj fn).



code
↳ keyword - class.
↳ end w/t ;

② creating class.

```
class Student {
  int roll;          → property
  int getroll(){     } method.
    return roll;
  }
};
```

② creating objects.

```
#include <iostream>
using namespace std;
#include "Student.cpp"        → adding our custom file

int main() {
  // Create objects statically
  Student s1;                 } object in stack
  Student s2;

  Student s3, s4, s5;

  // Create objects dynamicaaly
  Student *s6 = new Student;  → object in heap

}
```

↳ objects in memory.

student s1; creates mem. that of size of student class in stack, do the entry in symbol table and the assign garbage values to all prop. of s1.

student *s1 = new student; create mem. that of size of student in heap & 8 bytes in stack does symbol table entry of s1 & assign it the mem. address created in heap. In heap it assign garbage to all prop. of s1.

③ accessing members of object.

```
cout << s1.age << endl;
cout << s1.rollNumber << endl;
```
} from static object

```
(*s6).age = 23;
(*s6).rollNumber = 104;

s6 -> age = 23;
s6 -> rollNumber = 104;
```
} → (m-1) → from dynamic obj
} → (m-2)

④ access modifiers.
↳ define scope & members / prop. of class.
- private (default in cpp).
- public
- protected

using other access modifiers

```
class Student {

  public :

  int rollNumber;   → public

  private :
  int age;          → private
};
```

note:
public : } all these are public
private : } all these are private

⑤ getter and setter.
↳ public functions / methods in class to get & set private properties.
↳ setters are generally void, as they only set, edit, update values.

```
int getAge() {
  return age;
}                   } getter

void setAge(int a) {
  age = a;
}                   } setter
```

⑧ constructors

↳ Student S1; (S-1) call the constructor → function w/t same name as class & that run automatically when an obj is constructed.

↳ initializes all the prop. of obj. w/t either values by user, default values (or) garbage values.
↳ no return type
↳ no input argument (default values) } default constructor.
↳ only called once in a lifetime, & only one type is called.

note: if we don't create a constructor class will use inbuilt constructor.

⑧ default constructor.

```
Student ( ) {

}
```

↳ we can have multiple constructor in a class.

note: the moment we create a constructor the inbuilt constructor is overwritten.

note: constructor → default → allots garbage value.
constructor → parameterized → allots default / user values.

```
Student S1;
Student *S1 = new student;
```

```
Student S1(3);
Student * S1 = new Student(3);
```

↳ w/t out constructor obj can't be constructor.

⑧ this keyword.

↳ holds the address of current object.
↳ in function.

use: address of current object

```
Student(int a, int r) {
  cout << "this : " << this << endl;
  cout << "Constructor 3 called ! " << endl;
  this -> age = a;
  this -> rollNumber = r;
}
```

⑧ Inbuilt constructor & destructor.

↳ default constructor
↳ copy constructor (also provided by default).
↳ eg. Student S1(3); // call param. const.
Student S2(S1); // call copy const,
↑ object
(or)
Student * S2 = new Student(S1);
Student * S3 = new Student(*S2);
↑ can't use S2 as S2 = address, *S2 = value.

⑧ copy assignment operator.

↳ if in case we didn't used copy constructor & then we wanna copy entire object.

eg. S2 = S1;
(or)
(a) *S2 = S1;
(b) *S3 = *S2;

⑧ destructor

↳ also same name as class, but begin w/t ~ ← to differentiate from default constructor.
↳ no return type
↳ no input argument
(a) deallocates mem. of the object.
(b) called w/t object is destroyed.
↳ only called once in a lifetime.

Destructor: ~Student() { }

```
public :

~Student() {
  cout << "Destructor called ! " << endl;
}
```

note: if we create custom destructor then inbuild destructor is not available

V. imp. destructor only deallocates memory of objects in stack.
↳ if obj. is created in heap, deallocate memory manually w/t delete. → destructor not called for obj created in heap.
↳ destructor is called just before main end / return.

note: student S1 = S2;
↑ it is copy assignment operator, but because at this time S1 isn't in memory so.

student S1;
S1 = S2
copy assignment operator (X)

(or)

student S1(S2);
copy constructor (✓)

V. imp. behind the scenes student S1 = S2; will work as copy constructor.

⑦ shallow v/s deep copying.

```
class Student {
  int age;
  char *name;

  public :

  Student(int age, char *name) {
    this -> age = age;
    // Shallow copy
    // this -> name = name;

    // Deep copy
    this -> name = new char[strlen(name) + 1];
    strcpy(this -> name, name);

  }

  void display() {
    cout << name << " " << age << endl;
  }
};
```

shallow copy : instead of copying the entire array we copy the address of 1st element of array.

deep copy : here we create a new array & copy the entire array in this array.

note: copy constructor & copy assignment operator creates shallow copy.

⑧ custom copy constructor.

↳ whenever we work w/t array remember that = (or) copy constructor will do shallow copy.
↳ very bad

```
// Copy constructor
Student(Student s) {
  this -> age = s.age;
  // this -> name = s.name;   // Shallo Copy

  // Deep copy
  this -> name = new char[strlen(s.name) + 1];
  strcpy(this -> name, s.name);
}
```

custom copy constructor

here just like any fn we do pass by value ⇒ student S = S1;
passed arg.
act as copy constructor

↳ now when we created custom copy constructor inbuilt copy constructor gets unavailable now in the argument of custom copy constructor we need a copy copy constructor that is unavailable so it run our custom one when again ask for another copy → infinite loop.
↳ the solution is in the parameter of our custom copy constructor we shouldn't make a copy
  ↳ we should use reference var → no new memory is allocated for them.
  ↳ but w/t ref. var we may accidently change original var. passed.
  ↳ so we use const. reference var.

Syntax

```
// Copy constructor
Student(Student const &s) {
  this -> age = s.age;
  // this -> name = s.name;   // Shallo Copy

  // Deep copy
  this -> name = new char[strlen(s.name) + 1];
  strcpy(this -> name, s.name);
}
```

this is how a custom copy constructor is properly created.

⑨ Initialization list.

↳ used to initialise const and reference var. in class w/t constructor.

```
class Student {
  public :

    int age;
    const int rollNumber;
    int &x;   // age reference variable

    Student(int r, int age) : rollNumber(r), age(age), x(this -> age) {
      //rollNumber = r;
    }

};
```

can't be initialised later

late initialisation not allowed

initialisatn list: note: we can initialise any kind of variable w/t this but mainly const & reference var. are used as we have no alternative for em.

⑩ Constants

↳ constant object. eg: const Student S1; (or) Student const S1;

note: ① w/t constant object we can't change value of any property of that object.
② w/t constant objects we can only call constant functions.
③ Constant fns are those fns which don't change any property value of current object.

```
int getNumerator() const {
  return numerator;
}
```

this is how we create constant fn.

↳ if in a const fn we try to change the property of current obj we will get a big FAT error.

(*) Static members.

↳ any static member belongs to a class only; it is not given to the object.

```
class Student {

  public :

  int rollNumber;
  int age;

  static int totalStudents;   // total number of students present

};
```

→ static variable → only belongs to Student.

note: static members are initialised outside of class.

```
int Student :: totalStudents = 0; // initialze static data members
```

↳ accessing static members of class. ⇒ classname :: var_name → name of variable.

(as it only belong to class; :: scope resolution operator)

eg: Student::totalStudents;

note:
```
Student S1;
cout << S1.totalStudents;   → this will not show error but it's logically incorrect.
S1.totalStudents = 13;      → this will again not show error but it's again logically incorrect.
```

↳ as static vars belongs to the class it's kinda like global vars for objects.

↳ we also have static functions.

```
static int getTotalStudent() {   // this don't need an obj. to be called.
  return totalStudents;
}
```

(#) If we feel like any property/method of class don't need to create an object to access them make/ers static

note:
ⓐ static fns can only access static properties.
ⓑ static fns don't have this pointer.

(*) Operator overloading

↳ extending functionality of pre existing operator so they work for user defined data types as well.

```
Fraction operator+(Fraction const &f2) {
    int lcm = denominator * f2.denominator;
    int x = lcm / denominator;
    int y = lcm / f2.denominator;

    int num = x * numerator + (y * f2.numerator);

    Fraction fNew(num, lcm);
    fNew.simplify();
    return fNew;
  }
```

```
Fraction operator*(Fraction f2) {
  int n = numerator * f2.numerator;
  int d = denominator * f2.denominator;
  Fraction fNew(n, d);
  fNew.simplify();
  return fNew;
}
```

binary operators

other can be operator==( )

↳ pre-increment & post-increment (unary operators)

↳ how variables are assigned values.

eg: int i = 10; → memory is created acc. to data type.
→ 10 is stored in buffer (temp. memory).
→ entry of i is done on symbol table.
→ in the memory of i the value of buffer is stored.

eg: ++(++i); → ++i happen to i & the returned value is stored in buffer.
→ ++(buffer_value) happen & returned value is stored in another variable & the assigned to i.

eg: cout << ++i; → ++i happen to i & returned value is stored in buffer. (act as temp. variable.)
→ cout prints the value of buffer.

```
Fraction operator++() {
  numerator = numerator + denominator;
  simplify();

  return *this;
}
```

note:
```
Fraction f1(5/1);
++(++f1);   → nesting of pre-increment.
```
this → address of f1
↳ num = 5+1 = 6
↳ f1 = 6/1
↳ value of f1 = 6/1 is returned.
stored in temp. buffer

address of temp buffer ← (this
↳ num = 6+1
↳ buffer = 7/1
↳ value of 7/1 is returned.

⇒ f1 = 6/1
↳ but we wanted 7/1

↳ the problem arises because we are creating copy of the returned value.
↳ Soln is make the return type Fraction reference so no copy is created.

```
Fraction& operator++() {
  numerator = numerator + denominator;
  simplify();

  return *this;
}
```

note:
```
Fraction f1(5/1);
++(++f1);   → nesting of pre-increment.
```
↳ this → address of f1
↳ num = 5+1 = 6
↳ return address of f1

because return type is by reference, no copy is created.

this → address of f1
↳ num = 6+1 = 7
↳ return address of f1.

Post-increment

```
// Post-increment
Fraction operator++(int) {
  Fraction fNew(numerator, denominator);
  numerator = numerator + denominator;
  simplify();
  fNew.simplify();
  return fNew;
}
```

note: nesting isn't allowed w/t post-increment.

to distinguish it from pre-increment.

more operator overloading -

```
Fraction& operator+=(Fraction const &f2) {
  int lcm = denominator * f2.denominator;
  int x = lcm / denominator;
  int y = lcm / f2.denominator;

  int num = x * numerator + (y * f2.numerator);

  numerator = num;
  denominator = lcm;
  simplify();

  return *this;
}
```

by reference so copy won't be created & changes happen to same address as nesting allowed w/t this

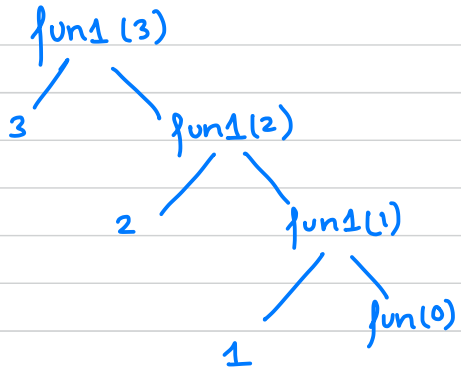
RECURSION

↳ When a function calls itself

EXAMPLE #1

```
void fun1(int n)
{
    if(n>0)
    {
        printf("%d", n);
        fun1(n-1);
    }
}

void main()
{
    int x = 3;
    fun1(x);
}
```

TRACING TREE



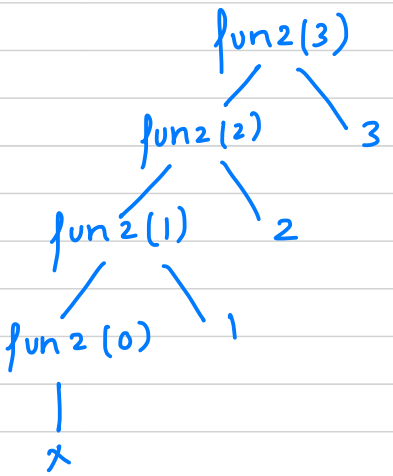
OUTPUT

3 2 1

EXAMPLE #2

```
void fun2(int n)
{
    if(n>0)
    {
        fun2(n-1);
        printf("%d", n);
    }
}

void main()
{
    int x = 3;
    fun2(x);
}
```

TRACING TREE



OUTPUT

1 2 3

```
void fun(int n)
{
    if (n>0)
    {
ASCENDING 1.  calling
          2.  fun(n-1)
DESCENDING 3.  returning
    }
}
```

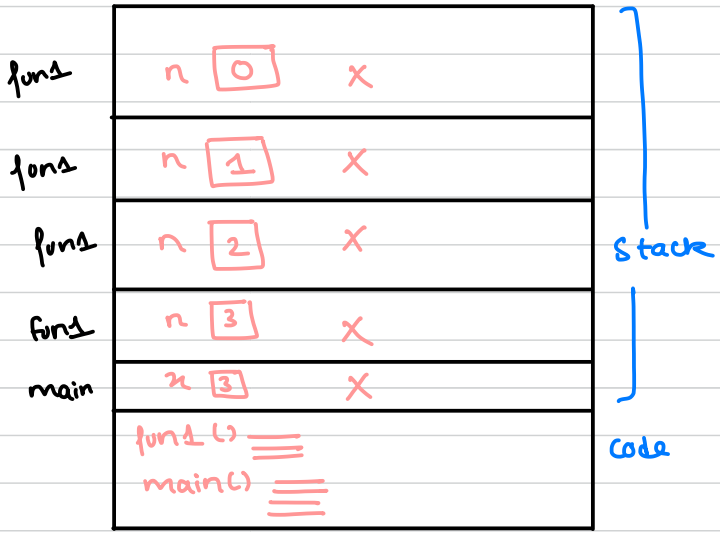
DIFFERENCE BETWEEN LOOP AND RECURSION

The main difference between loop and recursion is that recursion allows two phases i.e ascending and descending while loop allows only ascending

HOW STACK IS UTILISED IN RECURSIVE FUNCTIONS?

EXAMPLE #1

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d", n);
        fun1(n-1);
    }
}

void main()
{
    int x = 3;
    fun1(x);
}
```



Here, value of n was 3, so there were 4 calls (Tracing tree).

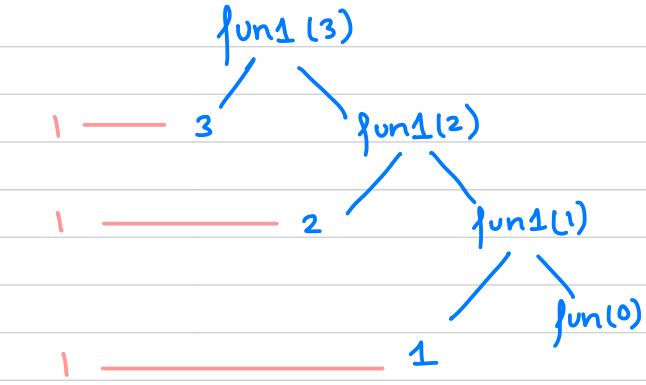
So for value of n, there will be n+1 calls

TIME COMPLEXITY (using Tree)

EXAMPLE #1

```
void fun1(int n)
{
    if (n>0)
    {
        printf("%d", n);
        fun1(n-1);
    }
}

void main()
{
    int x = 3;
    fun1(x);
}
```

TRACING TREE



Thus, for n calls → n units of time.

$O(n)$

TIME COMPLEXITY (using recurrence relation)

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+2 \text{ (1)} & n>0 \end{cases}$$

Assume it as 1 for constant

$T(n)$ — This function takes n time.

```
void fun1(int n)
{
    if (n>0)          // 1
    {
        printf("%d", n);   // 1
        fun1(n-1);         // T(n-1)
    }
}
```

As it is similar to $T(n)$

$T(n) = T(n-1) + 1$

$T(n) = T(n-1) + 1$

$T(n-1) = T(n-2) + 1$

$T(n) = T(n-2) + 1 + 1$

$T(n-3) + 1$

$T(n) = T(n-1) + 2$

Assume $n-k=0 \therefore n=k$

$T(n) = T(n-3) + 1 + 2$

$\vdots$

$T(n) = T(n-k) + k$

$T(n) = T(n-n) + n$

$T(n) = T(0) + n$

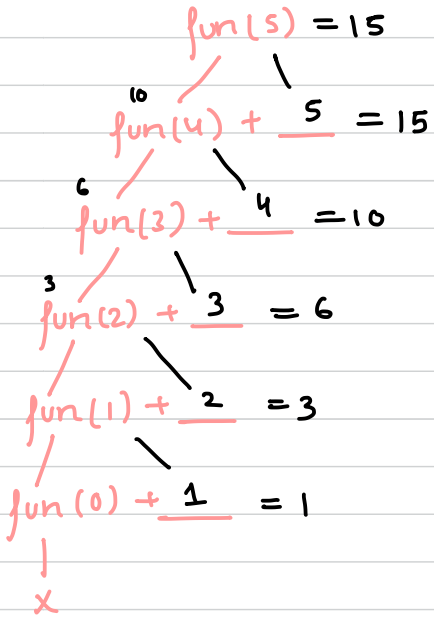
$T(n) = 1 + n$

$O(n)$

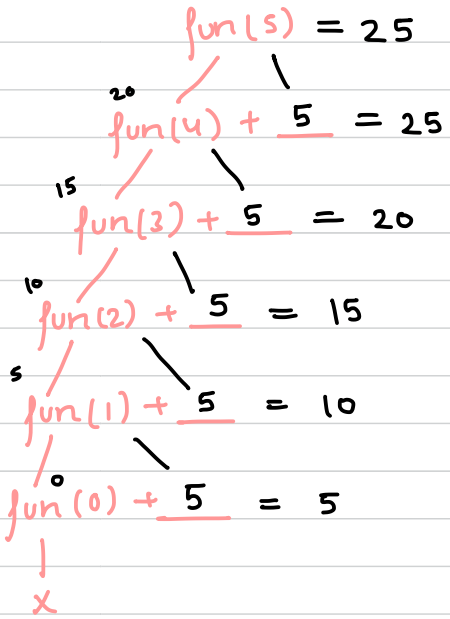
STATIC VARIABLES IN RECURSION

```
int fun (int n)
{
        if (n>0)
        {
            return fun (n-1) +n;
        }
        return 0;
}

main()
{
        int a= 5;
        printf (" %d", fun (a));
}
```

TRACING TREE




```
int fun (int n)
{       Static int x=0;
        if (n>0)
        {   x++;
            return fun (n-1) + x;
        }
        return 0;
}

main()
{
        int a= 5;
        printf (" %d", fun (a));
}
```



(*) Recursion can be done in 3 steps.
- Believe smaller problem will be solved automatically (Abstraction)
- find relation b/w f(n) and f(n-1).
- find the base case.
  - to break recursion.
  - relation for return statement
  - will work for n-1.

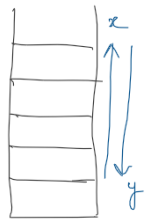
(M-2)
- High level thinking.
  - establish a relation.
  - eg. f(n) give n!
    - have faith.
    - eg. f(n-1) give (n-1)! surely
  - create relation between relation & faith.
    - eg. f(n) = n * f(n-1).

Low level thinking.
- find base case.
  - by thinking
  - from call stack or dry run
  - eg.
```
if (n == 1) {
    return 1
}
```

eg. factorial of n. (n) = f(n)
- (S-I) Believe we can get (n-1).
- (S-II) n = n x (n-1)
  f(n) = n x f(n-1)
- (S-III) If n = 1 return 1.

⇒ code.
```
int fac (int n) {
    if (n == 1)
        return 1;
    return n * fac(n-1);
}
```
make habit of storing this in a var.

(*) Type of recursion
- head recc.
- tail recc.
- tree recc. → recc. w/t multiple recc. calls in it. create a tree structure. eg. fibonacci.
- indirect recc. → recc. which call other fn & form cycle.
  - eg. A → B → C
    - A call B, B call C, C call A.
    - w/t some base case this loop will end.
- nested recc.
  - recc. fn w/t more recc.

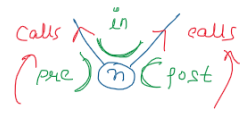
(*) note.
```
A (n) {
    base case, this call create the flip.        (resumes fn)
x   these lines will run going up in stack. (pushing fn in stack)
    A(n-1)   (1) this pauses the fn.
y   these lines will run going down in stack. (poping fn from stack)
}
```


note.
- if y code isn't present then it's tail recc.
- if x code isn't present then it's head recc.

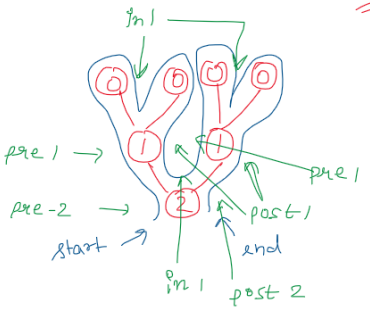
note. In tree recc. there are multiple calls → mostly 2.
- divide the code in 3 parts pre-in-post. → we do their analysis by drawing tree structure.

```
public static void pzz(int n){
    if(n == 0){
        return;
    }
    System.out.println("Pre " + n);   → pre
    pzz(n - 1);                       → left call
    System.out.println("In " + n);    → in
    pzz(n - 1);                       → right call
    System.out.println("Post " + n);  → post
}
```

note. drawing call stack for tree recc is time consuming so always draw tree for this.
- node → calls
- edge → pre-in-post region.

note.
```
A {
    → base case
    → pre reqn
left call ← A
    → in reqn
right call ← A
    → post reqn
}
```


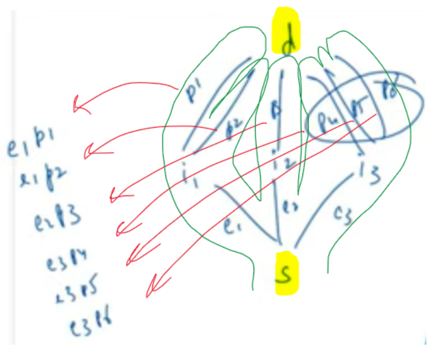
eg. n = 2



Imp.
- (S-1) draw tree by ignoring code & following calls only.
- (S-2) Analyse.

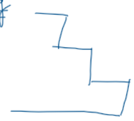
code.
- pre 2
- pre 1
- in 1
- post 1
- in 2
- pre 1
- in 1
- post 1
- post 2

(5) whenever we get problem of source & destination :



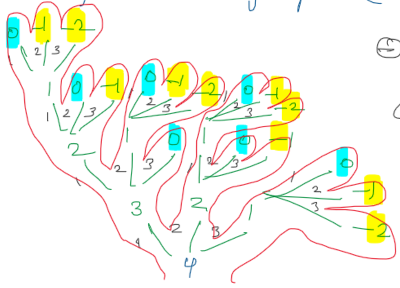
↳ check initial ways, ask for others & add that move on the head.

eg:



in stair if 1, 2 and 3 jumps are allowed then :-

eg:



(-) ve answers
↳ return empty vec.
0 → return v w/t

Soln.
↳ 1111, 112, 121, 13, 211, 22, 31 .

note: reccursion is paused & stay at stack until returnes so whenever we create variables in recc. f^n it might happen that for a descent recc. call (mem) might end up filling completly.
↳ RAM.

eg: for getSubSeq( ) if string is of size 31 char.
↳ it will take 32 gb of ram to store all its subseq.
so instead of storing, printing can sometimes be a good option, so no mem. is used / less mem is used.

↳ so in printing ones of tree recc. we take 2 things in f^n ques, ans & at base case print the ans.

```
string code[] = {".;", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tu", "vwx", "yz"};

void printKPC(string ques, string asf){
    // write your code here
    if(ques.length()==0){
        cout<<asf<<endl;
        return;
    }

    int s = ques[0] - '0';
    for(int i=0;i<code[s].length();i++){
        printKPC(ques.substr(1),asf+code[s].substr(i,1));
    }
}
```

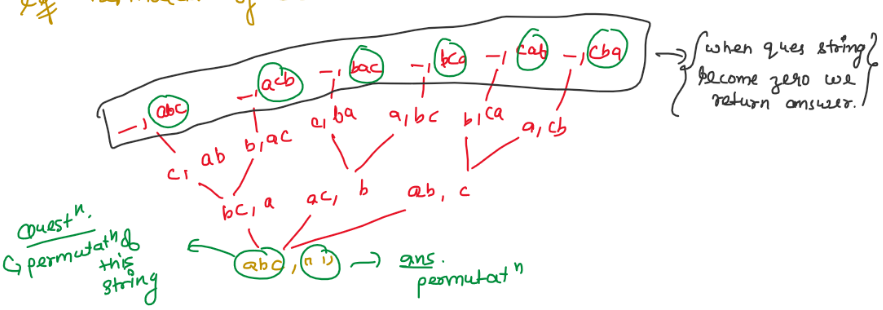
for these questions :

HL- thought
(1) asf will contain my the code of ques.
(2) asf should have the code of ques.substr(1)
relatn ; add the initial letter at the end.

LL- thought
(1) At base case just print & return.

note: (1) here printing is at the top so we can take it as recc. on way up, so put ans. in seq. order.
(2) always in this we have a question & an answer.

eg: Permutatn of abc.



1. A recursive approach mirrors the problem that we are trying to solve.
2. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers. But, recursion adds overhead for each recursive call (needs space on the stack frame).
3. Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
4. Any problem that can be solved recursively can also be solved iteratively.

TYPES OF RECURSION

1. Tail Recursion
2. Head Recursion
3. Tree Recursion
4. Indirect Recursion
5. Nested Recursion

1. TAIL RECURSION

When the function is calling itself and that call is the last call in the function.

```
fun (n)
{
    if (n > 0)
    {
        ========
        fun (n-1);
    }
}
```

1. Every operation is performed at calling time.
2. Tail recursions can easily converted into loops.

TAIL RECURSION AND LOOPS

Some compilers convert your program to loop if you have used tail recursion as they are more effecient

```
void fun (int n)
{
    while (n>0)
    {
        printf (" %d", n);
        n--;
    }
}

fun(3);
```

space O(1)

```
void fun ( int n)
{
    if (n>0)
    {
        printf (" %d", n);
        fun (n-1);
    }
}

fun (3);
```

O(n)

2. HEAD RECURSION

It means that the the function does not need to perform any operation at the time of calling. It has to do all operations at returning time.

```
void fun (int n)
{
    int i= 1;
    while (i<= n)
    {
        printf (" %d ", i);
        i++;
    }
}
    fun(3);
```

```
void fun(int n)
{
    if (n > 0)
    {
        fun (n-1);
        printf (" %d ", n);
    }
}
    fun(3);
```

Head recursions cannot be so easily converted into loops.

3. TREE RECURSION

LINEAR RECURSION

```
fun(n)
{
    if (n>0)
    {
        ==
        fun(n-1);
        ==
    }
}
```

When the function calls itself only once.

TREE RECURSION

```
fun(n)
{
    if(n>0)
    {
        ==
        fun(n-1);
        ==
        fun(n-1);
        ==
    }
}
```

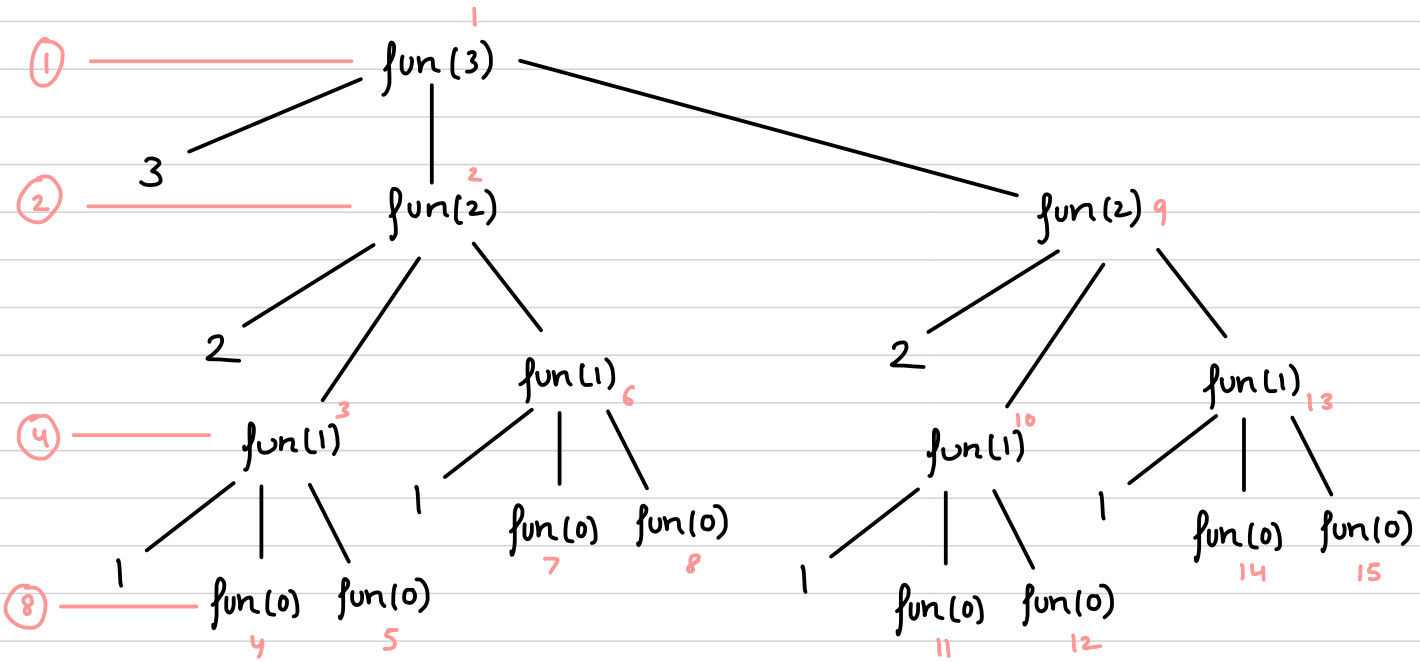
When the function calls itself more than one time.

EXAMPLE OF TREE RECURSION

```
void fun(int n)
{
    if(n>0)
    {
        printf("%d", n);
        fun(n-1);
        fun(n-1);
    }
}
```

```
fun(3);
```

- ACTIVATION RECORDS



$1 + 2 + 4 + 8 = 15$

$2^0 + 2^1 + 2^2 + 2^3 = 2^{3+1} - 1$ $\quad 2^{n+1} - 1$

GP Series

Time = $O(2^n)$

Space = $O(n)$

[No of levels of activation records $n+1$

3 – parameter

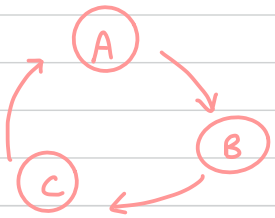
4 – levels]

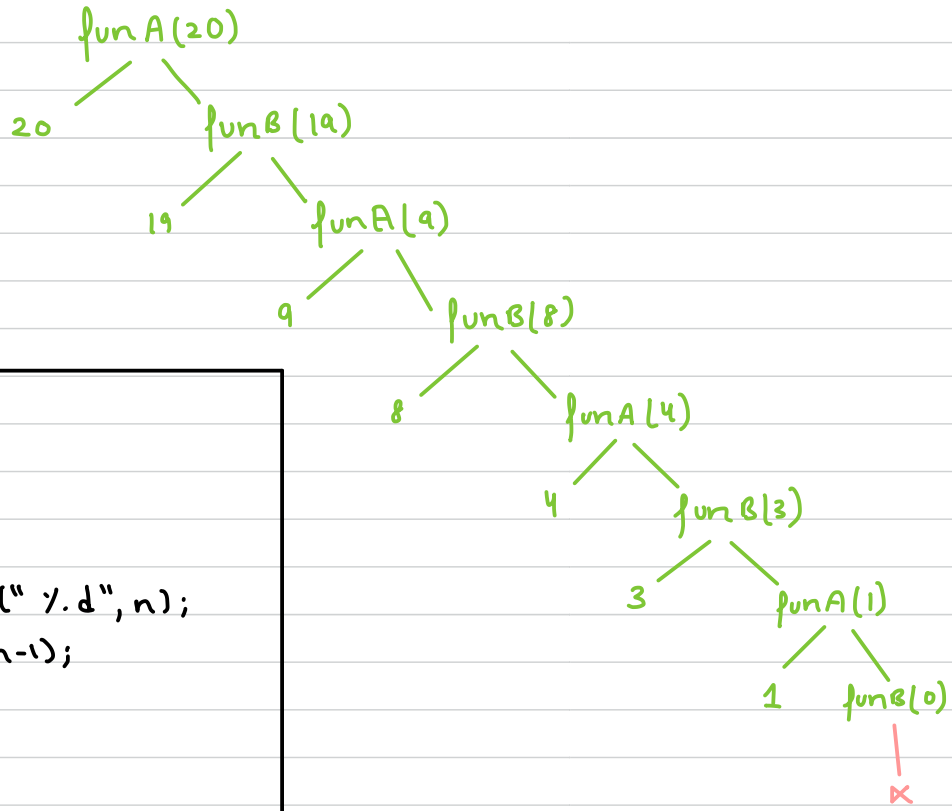
OUTPUT

3 2 1 1 2 1 1

4. INDIRECT RECURSION

In indirect recursion, there are more than one function and they are calling one another in a circular manner.



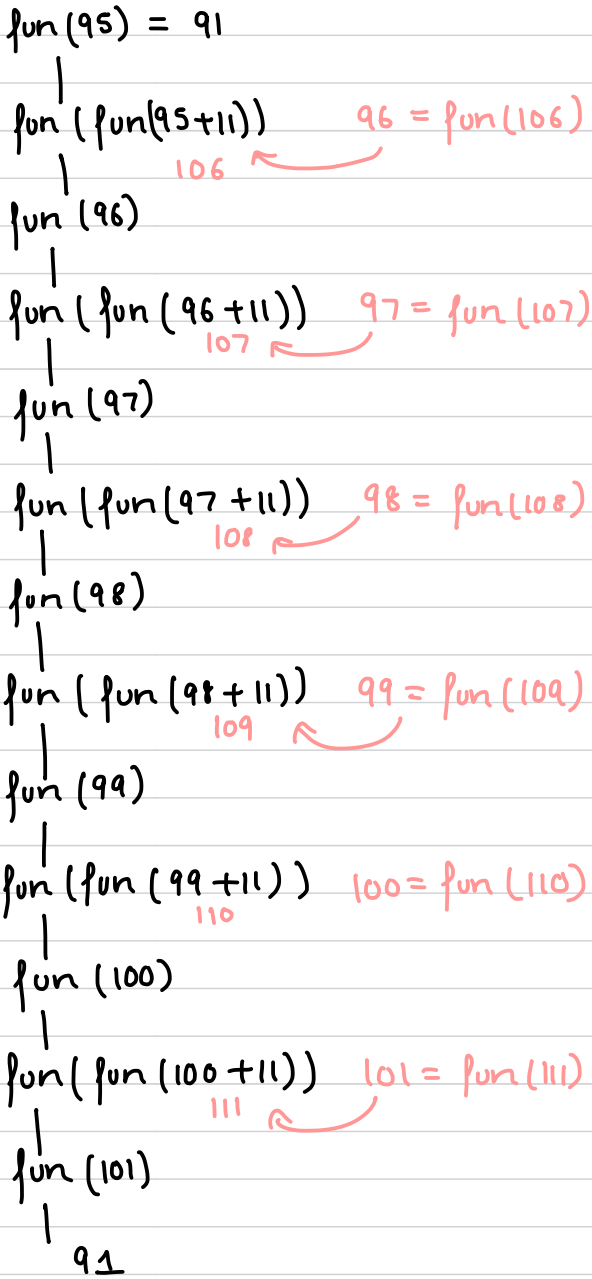



```
void funA(int n)
{
    if (n>0)
    {
        printf(" %d", n);
        funB(n-1);
    }
}

void funB(int n)
{
    if (n>1)
    {
        printf(" %d", n);
        funA(n/2);
    }
}
```

5. NESTED RECURSION

In a nested recursion, the recursive function will pass parameter as a recursive call.

```
int fun(int n)
{
    if(n>100)
        return n-10;
    else
        return fun(fun(n+11));
}

fun(95);
```



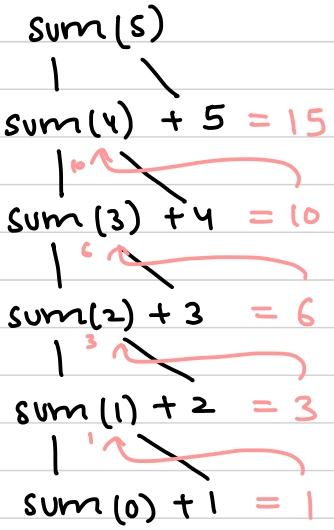
SUM OF FIRST N NATURAL NUMBERS

$1+2+3+4 \ldots\ldots + n$

$Sum(n) = 1+2+3+4+\ldots\ldots+(n-1)+n$

$sum(n) = sum(n-1)+n$

$$sum(n) = \begin{cases} 0 & n=0 \\ sum(n-1)+n & n>0 \end{cases}$$




```
int sum (int n)
{
    if (n==0)
        return 0;
    else
        return sum(n-1)+n;
}
```

Time = O(n)
Space = O(n)

```
int sum (int n)
{
    return n*(n+1)/2;
}
```

O(n)

```
int sum (int n)
{
    int i,s=0;          ——— 1
    for (i=1;i<=n;i++)  ——— n+1
        s=s+i;          ——— n
    return s;           ——— 1
}                           2n+3
```

O(n)

FACTORIAL USING RECURSION

$fact(n) = 1 * 2 * 3 * \ldots\ldots (n-1) * (n)$

$fact(n) = fact(n-1) * n$

$$fact(n) = \begin{cases} 1 & n=0 \\ fact(n-1) * n & n>0 \end{cases}$$

```
int fact (int n)
{
    if (n==0)
        return 1;
    else
        return fact(n-1) * n;
}
```

POWER USING RECURSION

$m^n = m * m * m * m \ldots\ldots$ for n times

$pow(m,n) = m * m * m \ldots\ldots * (n-1)$ times $* m$

$pow(m,n) = pow(m,n-1) * m$

$$pow(m,n) = \begin{cases} 1 & m=0 \\ pow(m,n-1) * m & m>0 \end{cases}$$

```
int pow(int m, int n)
{
    if (n==0)
        return 1;
    return pow(m,n-1);
}
```

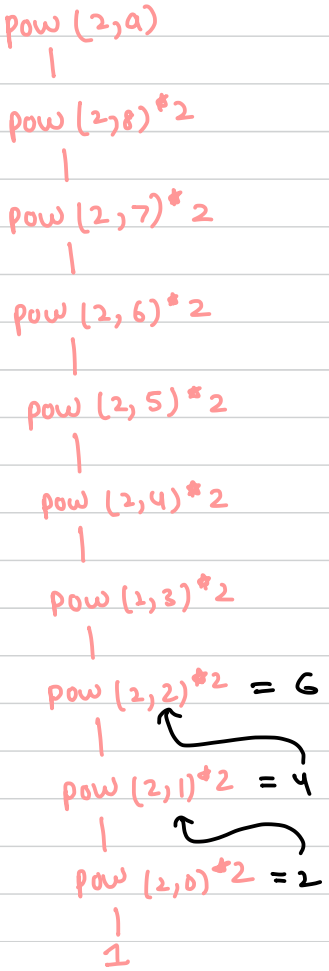
Here 10 calls are being performed

But this way is taking longer

$2^8 = (2^2)^4$

$= (2*2)^4$

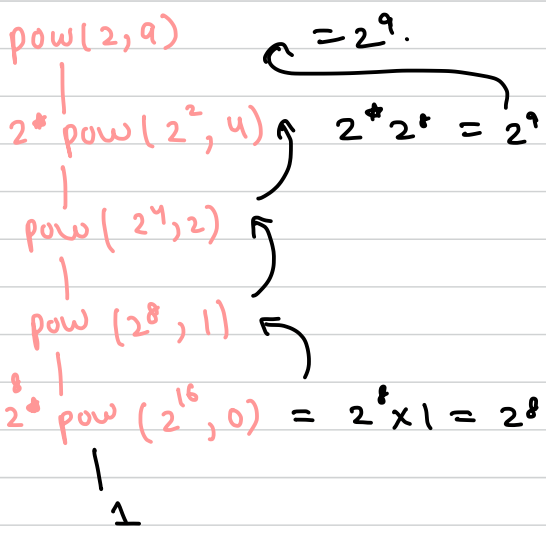
$2^9 = 2 * (2^2)^4$



REWRITING POWER FUNCTION

FASTER METHOD

```
int pow(m,n)
{
    if (n==0)
        return 1;

    else
        if (n%2==0)
            return pow(m*m, n/2);
        else
            return m*pow(m*m, (n-1)/2);
}
```



TAYLOR'S SERIES

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots\ldots \frac{x^n}{n!}$$

$e(x,4) = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!}$

$e(x,3) = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3!}$

$e(x,2) = 1 + \frac{x}{1} + \frac{x^2}{2}$

$e(x,1) = 1 + \frac{x}{1}$

$e(x,0)$ = 1

$p = p * x$ $\quad f = f * 1$ $\quad 1 + p/f$

```
int  e(int x, int n)
{
      static  int p=1, f=1;
      int r;

      if(n==0)
            return 1;

      else
      {
            r = e(x,n-1);
            p = p*x;
            f = f*n;
            return r + p/f;
      }
}
```

p	f
1	1
x*x*x*x	1X2X3X4

TAYLOR'S SERIES USING HORNER'S RULE

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \cdots \frac{x^n}{n!}$$

$\frac{x \times x}{2 \times 1}$ $\frac{x \times x \times x}{3 \times 2 \times 1}$

No of multiplications ↓

0 0 1+1 2+2 4 6 8 10

2

$2(1+2+3+\ldots n)$

$2\left(\frac{n)(n+1)}{2}\right.$

$O(n^2)$ Quadratic

$$1 + \frac{x}{1} + \frac{x^2}{1 \times 2} + \frac{x^3}{1 \times 2 \times 3} + \frac{x^4}{1 \times 2 \times 3 \times 4}$$

$$1 + \frac{x}{1}\left[\frac{x}{2} + \frac{x^2}{2 \times 3} + \frac{x^3}{2 \times 3 \times 4}\right]$$

$$1 + \frac{x}{1}\left[\frac{x}{2}\left[\frac{x}{3} + \frac{x^2}{3 \times 4}\right]\right]$$

$$1 + \frac{x}{1}\left[\frac{x}{2}\left[\frac{x}{3}\left[1 + \frac{x}{4}\right]\right]\right]$$

↑ ↑ ↑ ↑

$O(n)$ Linear

```
int e(int x, int n)
{
    int s=1;

    for(;n>0;n--)
        s= 1+ x/n*s;

    return s;
}
```

USING FOR LOOP

```
int e(int x, int n)
{
    Static int s=1;

    if(n==0)
        return s;
    else
        s= 1+ x/n*s;

    return e(x,n-1);
}
```

USING RECURSION

FIBONACCI SERIES

fib(n)	0	1	1	2	3	5	8	13
n	0	1	2	3	4	5	6	7

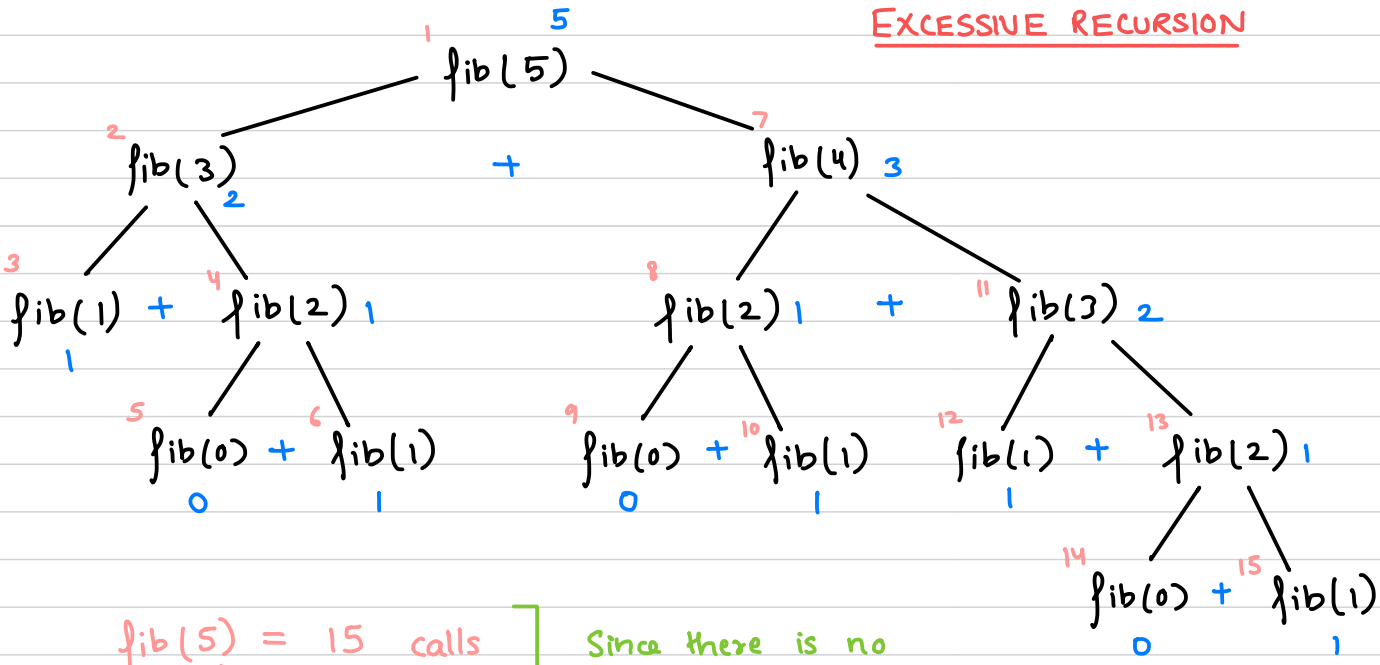
$$fib(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2) + fib(n-1) & n>1 \end{cases}$$

PROGRAM USING ITERATION

```
int fib (int n)
{
        int t0 = 0, t1 = 1, s, i;        — 1
        if (n<=1)
                return n;                — 1
        for (i= 2; i<=n; i++)            — n
        {
                s = t0 + t1;             — n-1
                t0 = t1;                 — n-1
                t1 = s;                  — n-1
        }

        return s;                        — 1
}                                          4n

                                   O(n)
```

PROGRAM USING RECURSION

```
int fib (int n)
{
        if (n<=1)
            return n;
        else
            return fib(n-2) + fib(n-1);
}
```



fib(5) = 15 calls
fib(4) = 9 calls
fib(3) = 5 calls

Since there is no definite pattern, than we have to assume that the function is calling itself

fib(n-2) + fib(n-1);
2 fib(n-1)

↓

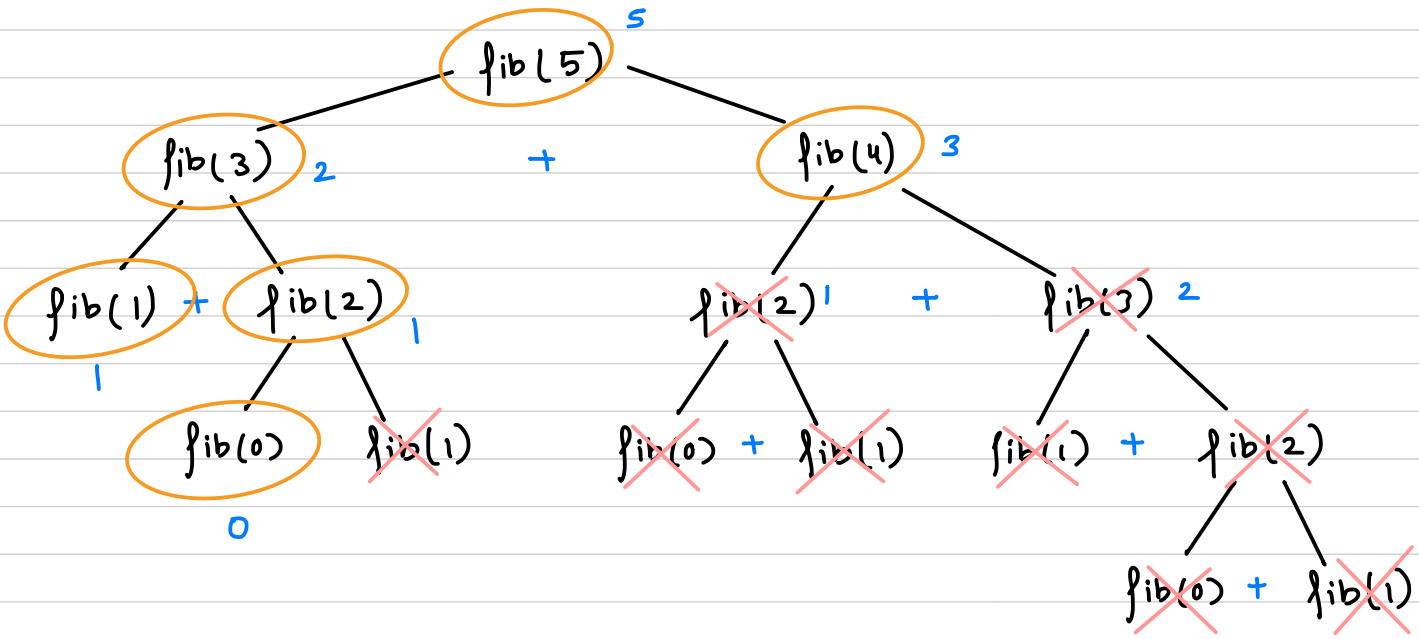
order of (2^n)

This tree lies in the category of excessive recursion because it calls itself multiple times for the same parameters.

To reduce the order of this function, we will write another program using static array and initialize all its values with -1.

F

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



So, for 5 as n, 6 calls are made.
∴ for n, n+1 calls are made

$O(n)$

This approach of storing result in an array is called MEMOIZATION.

↳ Storing the result of function calls, so they can be utilized again for avoiding excessive calls

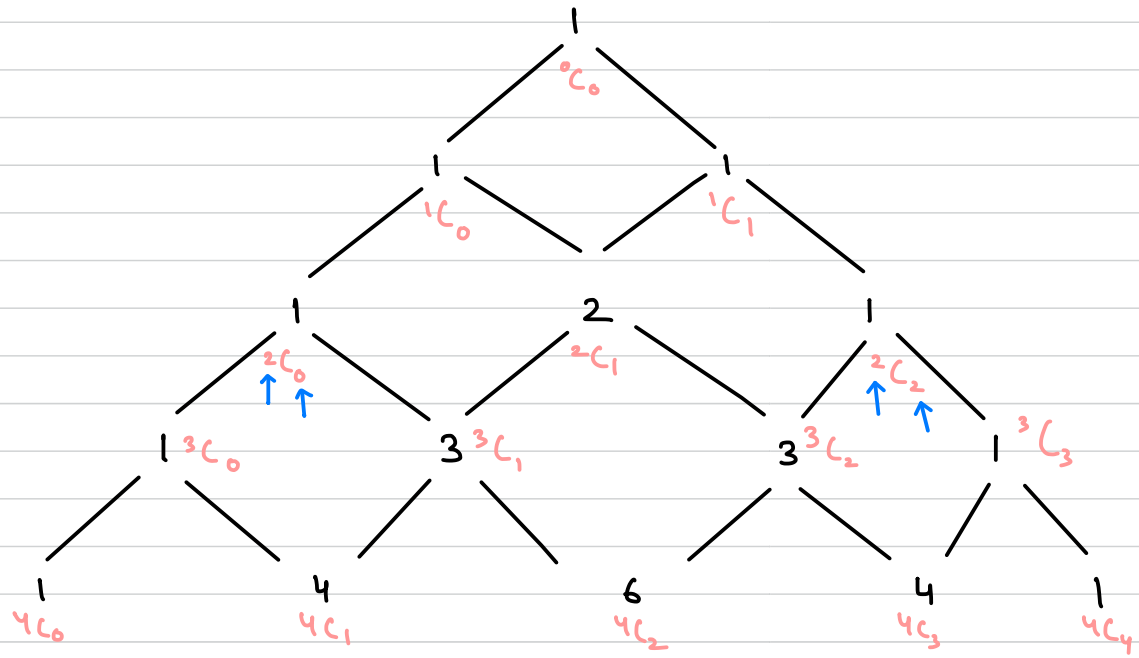
```
int F[10];

int fib(int n)
{
    if (n<=1)
    {
        F[n]=n;
        return n;
    }
    else
    {
        if(F[n-2]==-1)
            F[n-2]=fib(n-2);

        if(F[n-1]==-1)
            F[n-1]=fib(n-1);

        return F[n-2]+F[n-1];
    }
}
```

COMBINATION FORMULA

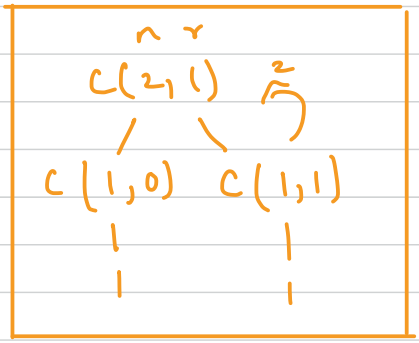
$${}^{n}C_{r} = \frac{n!}{(n-r)!\,r!}$$

```
int C (int n, int r)
{
        int t1, t2, t3;
        t1 = fact (n);  ---------- n
        t2 = fact (r);  ---------- n
        t3 = fact (n-r); --------- n

        return  t1 /(t2 * t3); ---- 1
}                                  ---
                                   3n
                                   O(n)
```

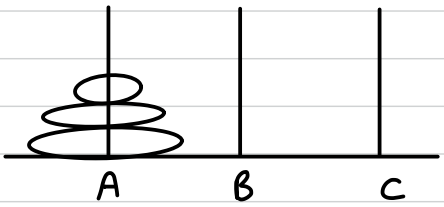
PASCAL'S TRIANGLE




```
int C (int n, int r)
{
        if (r == 0 || n == r)
                return 1;
        else
                return C (n-1, r-1) + C (n-1, r);
}
```

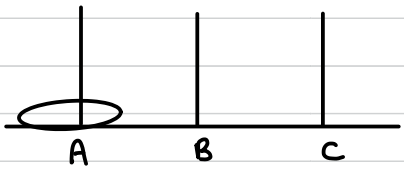


TOWER OF HANOI

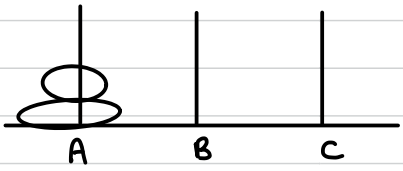


1. Move one disk at a time.
2. No bigger disk can be there above a smaller one.

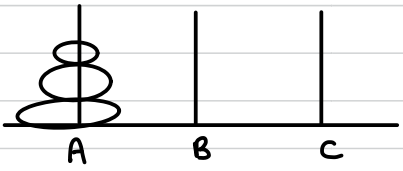
TOH (1, A, B, C)
Move disk A to C using B.



TOH (2, A, B, C)
1. TOH (1, A, C, B)
2. Move disk A to C using B.
3. TOH (1, B, A, C)



TOH (3, A, B, C)
1. TOH (2, A, C, B)
2. Move disk from A to C using B
3. TOH (2, B, A, C).



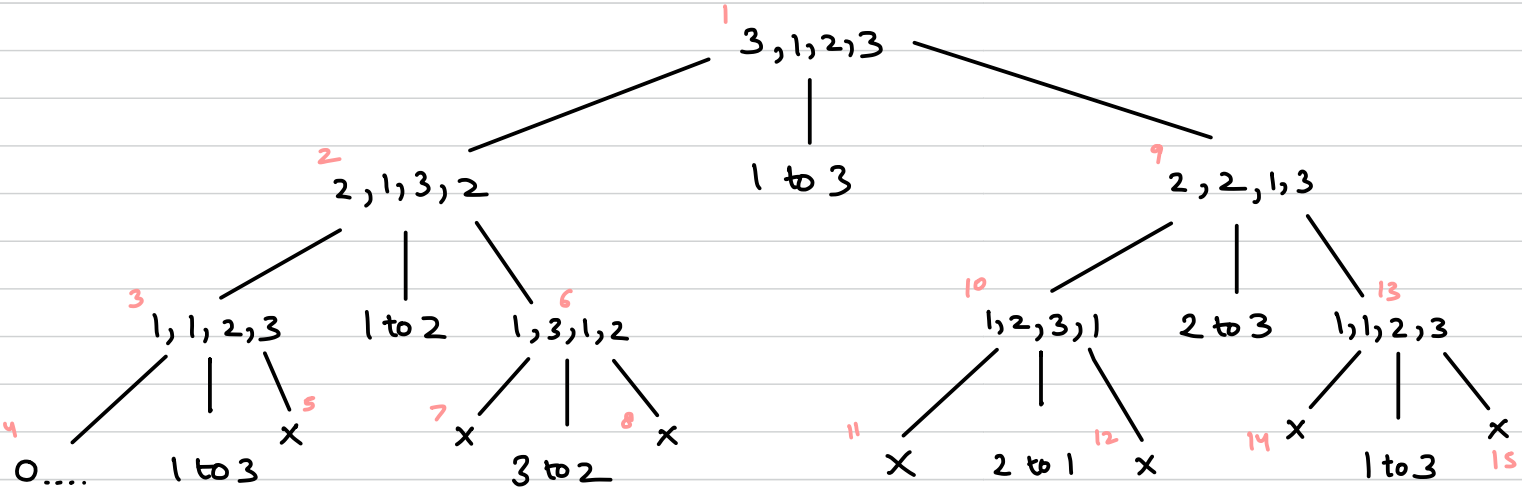
FOR n number of disk.

TOH ($\not{3}^{n}$, A, B, C)
1. TOH ($\not{2}^{n-1}$, A, C, B)
2. Move disk from A to C using B
3. TOH ($\not{2}^{n-1}$, B, A, C).

```
void TOH(int n, int A, int B, int C)
{
    if (n>0)
    {
        TOH(n-1, A, C, B);
        printf("from %d to %d", A, C);
        TOH(n-1), B, A, C);
    }
}
```

TOH (3,1,2,3)

A → 1
B → 2
C → 3



OUTPUT

(1 to 3), (1 to 2), (3,2), (1,3), (2,1), (2,3), (1,3)

	Calls	
n = 3	15	$1+2+2^2+2^3 = 2^4-1$
n = 2	7	$1+2+2^2 = 2^3-1$

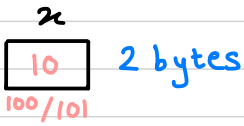
$2^{n+1}-1$

$O(2^n)$

ARRAYS

Scalar

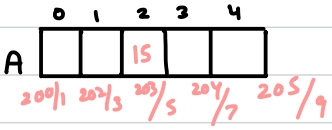
```
int x = 10;
```



2 bytes

vector

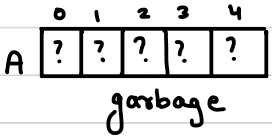
```
int A[5];
A[2] = 15;
```



10 bytes

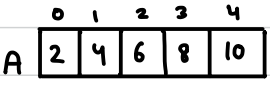
DECLARATION OF ARRAYS

① `int A[5];`

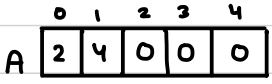


garbage

② `int A[5] = {2,4,6,8,10};`

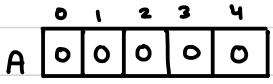


③ `int A[5] = {2,4};`

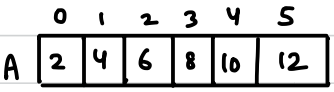


→ Rest of the elements get automatically initialized by 0.

④ `int A[5] = {0};`

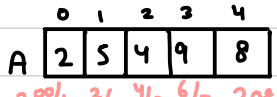


⑤ `int A[] = {2,4,6,8,10,12}`



→ Depending upon the number of elements, size of the array is automatically allocated.

`int A[5] = {2,5,4,9,8};`



→ Array addresses are contagious.

```
printf("%d", A[2]);
printf("%d", 2[A]);
printf("%d", *[A+2]);
```

→ OUTPUT
4

```
for(i = 0; i<5; i++)
    printf("%u", &A[i]);
```

%u for printing address.

- ✓ An Array is a data structure used to store blocks of information in contiguous memory allocation The data can be integer strings characters class objects etc .
- ✓ The type of data stored in array should of same data type . (Homogeneous storage)
- ✓ Indexing of array starts from zero (0) .

```
//Syntax
//<data_type> Name[size]
int intArray[100];
char characterArray[100];
```

(Array is immutable)

❖ DATA ENTRY IN ARRAY

✓ DATA ENTRY BY CODER

- int a[3] = {3,4,12} ;
- int a[3] ;
  a[0] = 3 ;

int a[4]; {all garbage values}
int a[5] = {1} {a[0]=1 remaining all are zero}
int a[] = {3,4,5}
↳ array of size 3×4 byte.
↳ Note: zero not garbage value

✓ DATA ENTRY BY USER

- cin>>a[0] ;
- for (int i=0; i<n; i++)
  {
  cin>>a[i] ;
  }

for 1-D array-
↳ a.length()
for 2-D array-
↳ a.length()
↳ a[i].length()

✪ Storing array.

↳ arrays are stored in contigious memory block.

eg: int a[10] → in memory 4×10 = 40 bytes of memory is allocated.
but a only stores address of a[0].

↳ to go to other address we go w/t help of array name which act as a pointer.

⇒ a[3] = a + 3×4 =
address of a

✪ Array and Function

writing function.

type name (int a[]) { }
this acts as pointer
↳ asking for array or address of array

∴ size of a = size of pointer = 8 bytes
↳ address stored in Hex.

Calling function-
all we have to do is pass name of array = address of a[0].

int b[10];
name(b);

Note: &a = &a[0] = a

④ Strings are 1D character array & it acts quite different from other array.

eg: char c[] = "hello";

In any other type of array printing c will give & c[0], but w/t char. array printing c will start printing characters till \0.

↳ functioning of cout.

Similarly w/t cin ⇒ char c[10];
cin >> c; (no need of for loop)

So if we give "dheeraj"
so storge will be

d	h	e	e	r	a	j	\0	

⇒ \0 will be appended automatically at the end

Note: So in character array if we need a string of length 'n' we should make the array of length (n+1), one extra space for '\0'.

↳ we can also acess characters from index. eg. c[3], c[0] etc.

Note: cout<< c; (this will keep printing characters from & c[0] till it encounter c\0'.

⑤ Some inbuilt functions of char. array from <cstring>.

1. Strlen (a)
2. strcmp(a, b) # 0 = same, any other int = not same
3. strcpy (destinat^n_string, source_string).
4. strncpy (destinat^n_string, source_string, n).

⑥ 2D - Array.

eg: int a[2][3];

rows - columns.

a[0][1] = 4;

↳ Storing 2-D Arrays.

int a[r][c];

in memory a r×c size of 1-D array is created.

int a[i][j] = 5;

in this, system calculate (r*i+j) & insert the value 5 in the 1-D array of memory.

↳ W/t functions.

void abc(int b[][7])

asking for 2-D array.

in 2D array we must specify columns (must)

↳ Just like 1D array if we don't initialize 2D array they will take garbage value, if we initialize a few of them the rest will take zero (0).

STATIC VS DYNAMIC ARRAY

Size of the array is static

Size of the array is dynamic

→ Once an array is created, it's size cannot be modified.

In C

→ Size of the array is decided at compile time

In C++

→ The size of the array can be decided at run time also.

```
int n;
cin>>n;
int A[n];
```

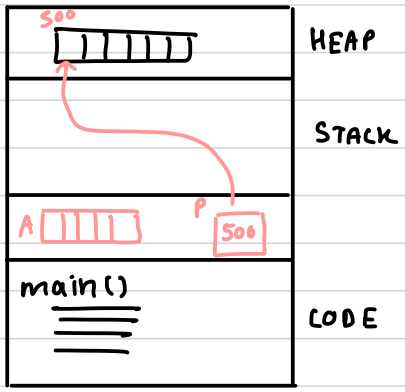
ACCESSING HEAP

```
void main()
{
    int A[5];
    int *p;

C++ p = new int[5];

C   p = (int *)malloc(5 * sizeof(int));
                <stdlib.h>
    :
    :

C++ delete []p;
  C free(p);
}
```

p[0] = 5

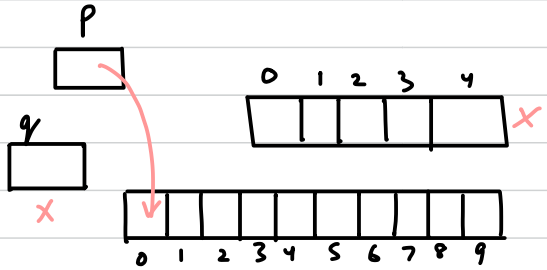
Otherwise MEMORY LEAK

Shortage of memory.



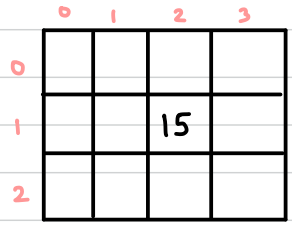
ONE WAY OF INCREASING SIZE OF ARRAY

```
int *p = new int[5];
int *q = new int[10];

for(i=0; i<5; i++)
    q[i] = p[i]

delete []p;
p = q;
q = NULL;
```



2D - ARRAY

① `int A[3][4] = { {1,2,3,4}, {2,4,6,8}, {3,5,7,9} };`



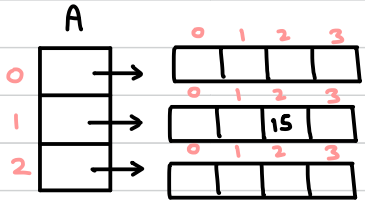
↓ Array will be stored inside stack.

`A[1][2] = 15`

② `int *A[3];` → Array of pointers

```
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

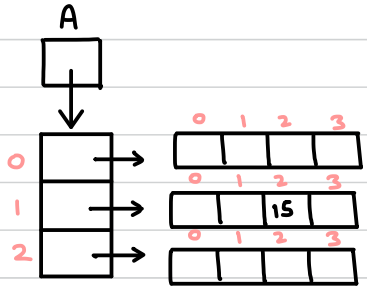
Memory is created inside heap.



`A[1][2] = 15;`

We can access array of pointers in the same way.

③



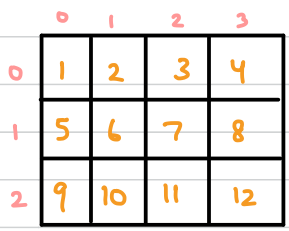

```
int **A;
A = new int*[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

All the memory is allocated in heap.

for column

```
for(i=0; i<3; i++)
{
    for(j=0; j<4; j++)
    {
        A[i][j] = __;
    }
}
```

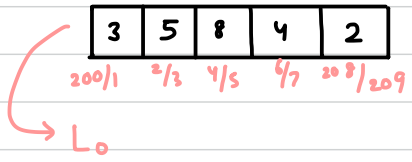
for row



- Elements accessed in order

HOW COMPILER GENERATES FORMULA FOR ADDRESS OF AN ARRAY

int A[5] = {3, 5, 8, 4, 2};



ADD (A[3]) = 200 + 3 * 2 = 206

ADD (A[3]) = L_0 + 3 * 2

ADD (A[i]) = L_0 + i * w → FORMULA

No of operations = 2

L_0 : Base Address

i : Index

w : Size of datatype.

ADD (A[i]) = L_0 + (i-1) * w → Formula, when indices are starting from one onwards

No of operations = 3

More time consuming.

That's why C and C++ do not start array index with 1. Only for one extra operation, time taken by the program will increase and this will make the program slower.

ROW MAJOR

int A[3][4];

m n

Add (A[1][2]) = 200 + [4 + 2] * 2 = 212

Add (A[2][3]) = 200 + [2 * 4 + 3] * 2 = 222

Add (A[i][j]) = L_0 + [i * n + j] * w

4 operations

Add([i][j]) = L_0 + [(i-1) * n + (j-1)] * w

6 operations

COLUMN MAJOR

$$ADD(A[1][2]) = 200 + [2*3+1]*2 = 214$$

$$ADD(A[1][3]) = 200 + [3*3+1]*2 = 220$$

$$ADD(A[i][j]) = L_0 + (j*m+i)*\omega$$

FORMULAS FOR nD ARRAYS

$A[d1][d2][d3][d4];$ 4D ARRAY

Row Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1*d_2*d_3*d_4 + i_2*d_3*d_4 + i_3*d_4 + i_4]*\omega$$

Column Major

$$ADD(A[i_1][i_2][i_3][i_4]) = L_0 + [i_4*d_3*d_2*d_1 + i_3*d_2*d_1 + i_2*d_1 + i_1]*\omega$$

ROW MAJOR FOR nD

$$L_0 + \sum_{p=1}^{n}\left[i_p * \prod_{q=p+1}^{n} d_q\right]*\omega$$

COLUMN MAJOR for nD

$$L_0 + \sum_{p=n}^{1}\left[i_p * \prod_{q=p-1}^{1} d_q\right]*\omega$$

(self tried)
(please check)

A[d1][d2][d3][d4]; 4D ARRAY HORNER'S RULE

Row Major

$$Add(A[i_1][i_2][i_3][i_4]) = L_0 + [i_1 * d_2 * d_3 * d_4 + i_2 * d_3 * d_4 + i_3 * d_4 + i_4] * \omega$$

3 2 1

4D → 3 + 2 + 1
5D → 4 + 3 + 2 + 1
nD → n-1 + n-2 + n-3 .... + 1 = $\frac{n(n-1)}{2}$

$O(n^2)$

$$i_4 + i_3 * d_4 + i_2 * d_3 * d_4 + i_1 * d_2 * d_3 * d_4$$

$$i_4 + d_4[i_3 + i_2 * d_3 + i_1 * d_2 * d_3]$$

$$i_4 + d_4[i_3 + d_3[i_2 + i_1 * d_2]]$$

$O(n)$

FORMULA FOR 3D ARRAYS

int A[l][m][n];

Row Major

$$Add(A[i][j][k]) = L_0 + [i * m * n + j * n + k] * \omega$$

Column Major

$$Add(A[i][j][k]) = L_0 + [k * m * l + j * l + i] * \omega$$

ARRAY ADT

↳ Abstract Datatype

↓

1. Representation of data
2. Operations on data

DATA

1. Array Space
2. Size
3. Length (No of elements)

① `int A[10];`

② `int *A;`
`A = new int[size];`

OPERATIONS

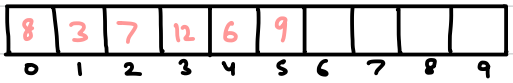
1. Display()
2. Add(x) / Append(x)
3. Insert(index, x)
4. Delete(index)
5. Search(x)
6. Get(index)
7. Set(index, x)
8. Max(), Min()
9. Reverse()
10. Shift() / Rotate()

1. DISPLAY

pseudo code

```
for(i=0; i<length; i++)
{
        print(A[i])
}
```

Array Size = 10
Length = 6



2. Add(x) / Append(x)

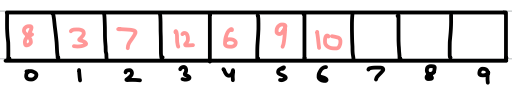
```
A[Length] = x;      ——— 1
Length++;           ——— 1
```

$f(x) = 2$

$f(n) = 2n^0$

$O(n^0) = O(1)$

Array Size = 10
Length = 7

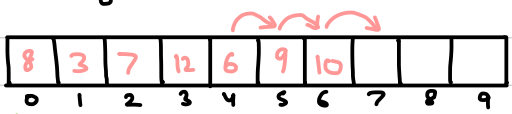


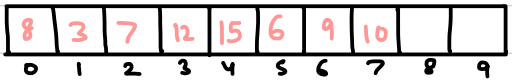
3. Insert (index 4, x 15)

```
for (i = length ; i > index ; i--)
{
    A[i] = A[i-1] ;        — 0-n
}
A[index] = x ;             — 1
Length++ ;
```

Inserting at index 8 ↵ ↓ Inserting at 0

Array Size = 10
Length = 8





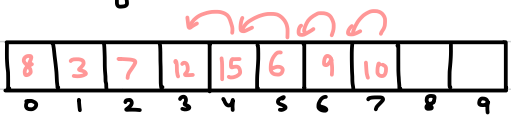
O(1) min O(n) max

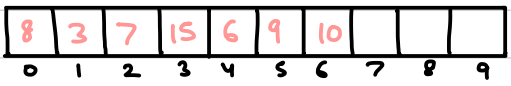
4. Delete (index 3)

```
x = A[index] ;             — 1
for ( i = index ; i <= Length-1 ; i++)
    A[i] = A[i+1] ;        — 0-n
Length-- ;                 — 1
```

min = 2
max = n+2

Array Size = 10
Length = 7



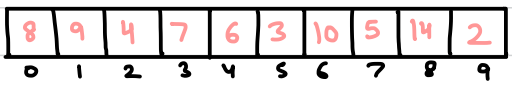


Best O(1) Worst O(n)

Index should be in range of length
we cannot leave empty space between two elements in an array.

LINEAR SEARCH → Searching each element and incrementing

Array Size = 10
Length = 10



Key = 5 ← Succesful

Key = 12 ← Unsuccesful

```
for (i=0 ; i < length ; i++)
    if (key == A[i])
        return i ;
return -1 ;
```

Best — O(1)
Worst — O(n)
Average — O(n)

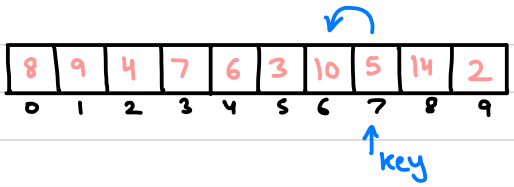
Average

found at location 1
found at 2

$$\frac{1+2+3+\dots n}{n} = \frac{(n+1)n}{2} \cdot \frac{1}{n} = \frac{n+1}{2}$$

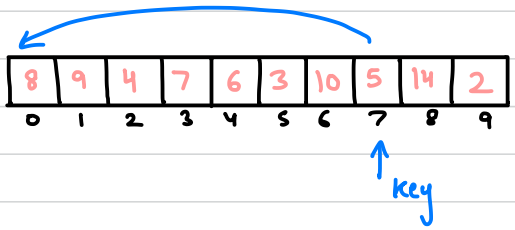
FASTER WAY OF LINEAR SEARCH

1. Transposition : Moving the searched element one step back




```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[i-1]);
            return i-1;
        }
}
```

2. Move to front / head : The searched element is brought to first position in array.

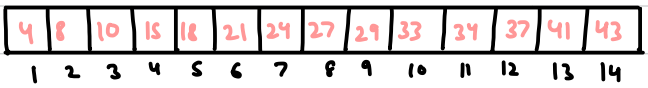



```
for (i=0;i<length; i++)
 {
        if ( key == A[i])
        {
            swap( A[i], A[0 ]);
            return 0;
        }
}
```

BINARY SEARCH

Size = 15
Length = 15

A



Sorted Array

ITERATIVE VERSION

```
Algorithm BinSearch (l, h, key)
{
    while (l <= h)
    {
        mid = [(l+h)/2];

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            h = mid - 1;
        else
            l = mid + 1;
    }
    return -1;      # key not found
}
```

Binary search is faster than linear search

RECURSIVE VERSION

Tail Recursion

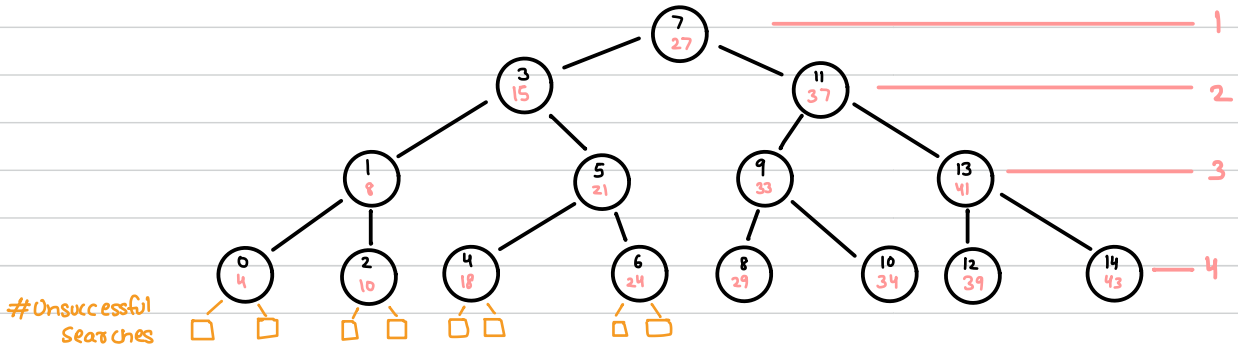
```
Algorithm RBinSearch (l, h, key)
{
    if (l <= h)
    {
        mid = [(l+h)/2];

        if (key == A[mid])
            return mid;

        else if (key < A[mid])
            return RBinsearch (l, mid - 1, key);
        else
            return RBinsearch (mid + 1, h, key);
    }
    return -1;      # key not found
}
```

ANALYSIS OF BINARY SEARCH

Size = 15
Length = 15

A

4	8	10	15	18	21	24	27	29	33	34	37	41	43
1	2	3	4	5	6	7	8	9	10	11	12	13	14



Best — min — O(1)
Worst — max — O(logn)
Unsuccesful — max — O(logn)

Why logn ?

logn

Let size of array be 16

16 / 2 / 2 / 2 / 2

$2^4 = 16$

$4 = \log_2 16$

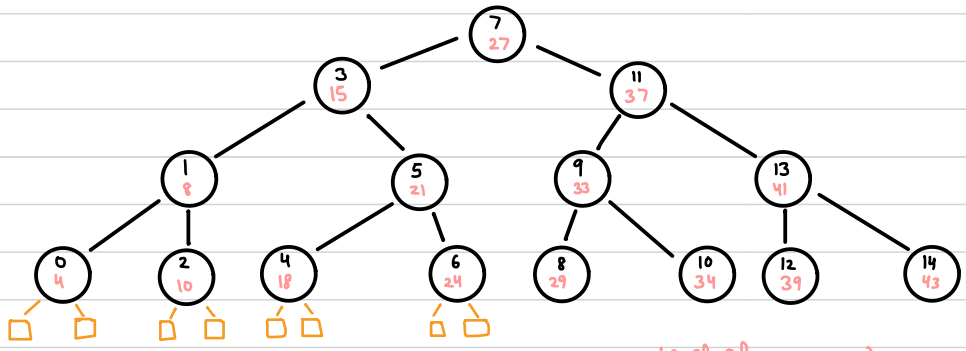
$\log_2 n$

Inverse of power is log.

16 is divided by 2 multiple times

AVERAGE CASE ANALYSIS OF BINARY SEARCH *

$$= \frac{\text{Total time taken in all possible cases}}{\text{Number of cases}}$$



$1 + 1 \times 2 + 2 \times 4 + 3 \times 8$

No of comparisons ↵ ↳ No of elements

$1 + 1 \times 2^1 + 2 \times 2^2 + 3 \times 2^3$

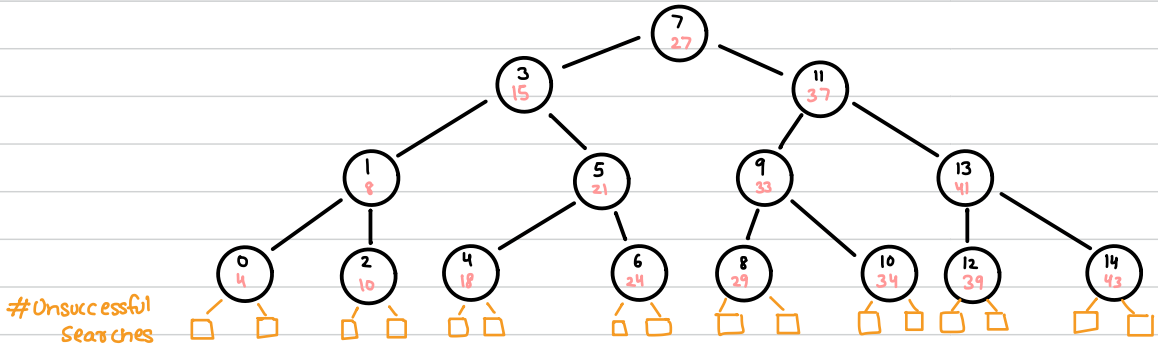
$\sum_{i=1}^{\log n} \frac{i \times 2^i}{n}$ — $\log n$ → Level of tree (Here 4)

$= \frac{\log n \times 2^{\log n}}{n}$ — i is replaced with logn

Same as formula *

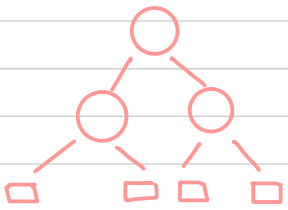
$= \frac{\log n \times n^{\log 2}}{n}$

$= \log n$



I = sum of all internal nodes
E = sum of all external nodes

$E = I + 2n$



n = 3
I = 2
E = 2x4 = 8

$E = I + 2n$

i = num of internal nodes
e = num of external nodes

n = num of nodes

$e = i + 1$

Average successful time for n elements

$$A_S(n) = 1 + \frac{I}{n} \Rightarrow 1 + \frac{E-2n}{n} \Rightarrow 1 + \frac{n\log n}{n} - 2$$

$$\Rightarrow 1 + \frac{E}{n} - 2 \Rightarrow \log n$$

Average unsuccessful time

$$A_U(n) = \frac{E}{n+1} = \frac{n\log n}{n+1} = \log n$$

$E = n\log n$
$E = I + 2n$
$I = E - 2n$

6. **Get (index) :**

```
if (index >= 0 && index < length)        O(1)
    return A[index];
```

7. **Set (index, x)**

```
if (index >= 0 && index < length)        O(1)
    A[index] = x;
```

8. Max()

```
max = A[0];                          ——— 1
for (i = 1; i < length; i++)         ——— n
{
        if (A[i] > max)              ——— n-1
            max = A[i];
}
return max;                          ——— 1
                                     2n+1   O(n)
```

9. Min()

```
min = A[0];
for (i = 1; i < length; i++)
{
        if (A[i] < min )
            min = A[i];
}
return min;
```

10. Sum()

```
Total = 0;                           ——— 1
for (i = 0; i < length; i++)         ——— n+1
        Total += A[i]                ——— n
return total                         ——— 1
                                     2n+3
                                     O(n)
```

$$Sum(A,n) = \begin{cases} 0 & n < 0 \\ Sum(A,(n-1)) + A[n] & n >= 0 \end{cases}$$

11. Avg()

```
Total = 0;
for (i = 0; i < length; i++)
        Total += A[i]
return total/n;
```

RECURSIVE CALL

```
int Sum (A, n)
{
    if (n < 0)
        return 0;
    else
        return sum (A, n-1) + A[n];
}
```

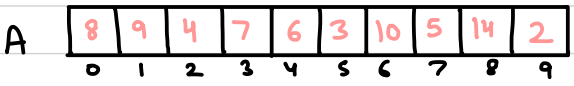
Sum (A, length-1) — call

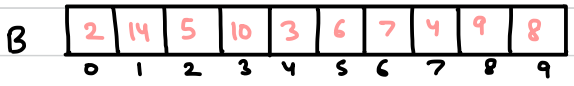
REVERSE AND SHIFT AN ARRAY

1. Reverse
2. Left Shift
3. Left Rotate
4. Right Shift
5. Right Rotate

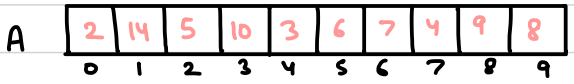
1. Reversing an array

Reversing using auxillary array B.





A → B n



B → A n

2n

O(n)

```
for (i = length-1; j=0; i>=0 ; i-- , j++)
        B[j] = A[i];
```

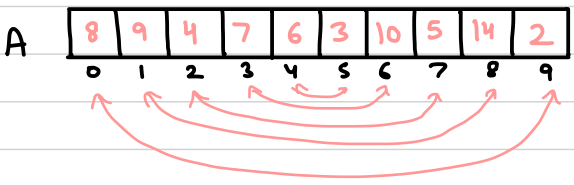
] Reverse copying array

```
for (i=0 ; i < length ; i++)
        A[i] = B[i];
```

ANOTHER METHOD OF REVERSING ARRAY

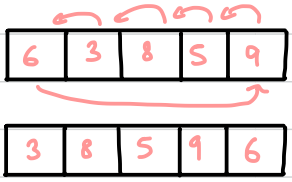
```
for ( i=0 ;  j = length-1 ; i<j ; i++ ; j--)
{
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
}
```

O(n)



2. Left Shift

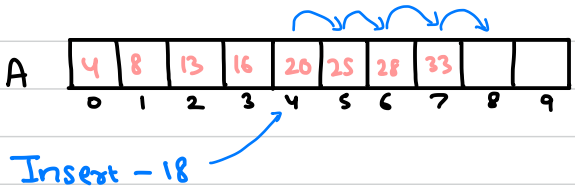
3. Left Rotate



CHECK IF ARRAY IS SORTED

1. Inserting in an sorted array
2. Checking if array is sorted
3. Arranging -ve elements on left side and +ve on right side.

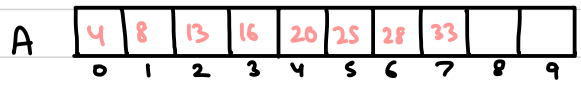
1. Inserting in an sorted array



Starting from last element, loop will run until the element is greater than the element to be inserted

```
x = 18 ;
i = length - 1 ;
while ( A[i] > x )
{
      A[i+] = A[i];
      i-- ;
}
A[i+1] = x ;
```

2. Checking if array is sorted




```
Algorithm isSorted ( A, n)
{
      for (i=0 ; i<n-1 ;i++)
      {
            if ( A[i] > A[i+1])
                  return false;
      }
      return true;
}
```

We will check false condition first by checking if any element is greater than its next element.

O(n) - Max Worst
O (1) - Min Best

3. Arranging -ve elements on left side and +ve on right side.

```
i=0;
j= length-1;

while (i<j)
{
        while (A[i]<0)
            i++;
        while (A[j]≥0)
            j--

        if (i<j)
            swap(A[i], A[j])
}
```

A

-6	3	-8	10	5	-7	-9	12	-4	2
0	1	2	3	4	5	6	7	8	9

i, j

A

-6	-4	-8	-9	-7	5	10	12	3	2
0	1	2	3	4	5	6	7	8	9

j, i

$O(n)$ $n+2$ comparisons are made

MERGING ARRAYS

1. Append / Concat
2. Compare
3. Copy

```
i=0;
j=0;
k=0;
while (i<m && j<n) ⟶ # When either of i or j have reached end of array.
{
    if (A[i]<B[j])
    {
        C[k]=A[i];
        k++;
        i++;
    }
```

Can also be written as `C[k++]=A[i++];`

m

A

3	8	16	20	25
0	1	2	3	4

i

n

B

4	10	12	22	23
0	1	2	3	4

j

C

3	4	8	10	12	16	20	22	23	25
0	1	2	3	4	5	6	7	8	9

k

```
    else
    {
        c[k]=B[j];
        k++;
        j++;
    }
```

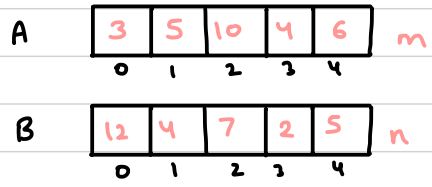
`c[k++]=B[j++]`

$\Theta = (m+n)$ ↳ Time is known

```
    for(; i<m; i++) ⟶ # When some elements in either of array are remaining to be copied.
        c[k++]=A[i];
    for(; j<n; j++)
        c[k++]=B[j];
}
```

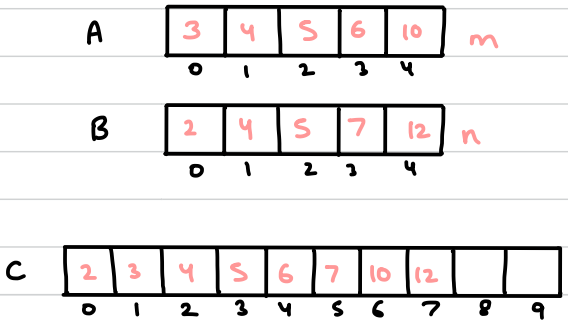
SET OPERATIONS

1. UNION



- Copy all elements of A in C
- Then copy elements of B which are not there in C.

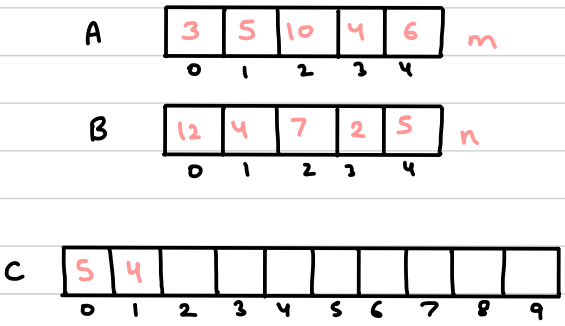
$m + m*n$
$n + n*n$
$n + n^2$
$O(n^2)$



- Use merge procedure
- Copy the smaller element to C
- If same element, copy once
- Use i, j, k method

$\Theta(m+n)$
$\Theta(n+n)$
$\Theta(n)$

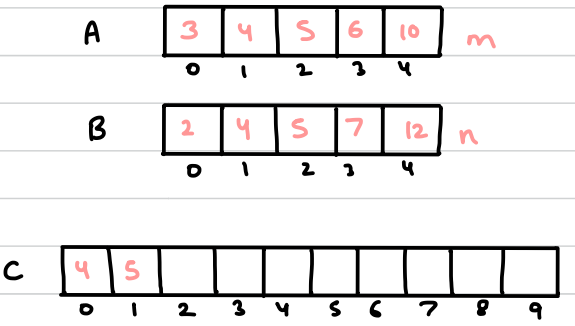
2. INTERSECTION



Copy the common elements of A and B to C

- One by one, take each element of A If it is present in B, copy it to C otherwise move on to next element

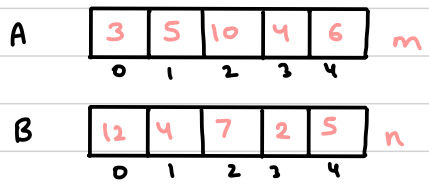
$n*m$
$n*n$
$O(n^2)$



- Use merge method
- If element is smaller, don't copy.
- If element is same, copy
- Not merging but similar to it.

$\Theta(m+n)$
$\Theta(n)$

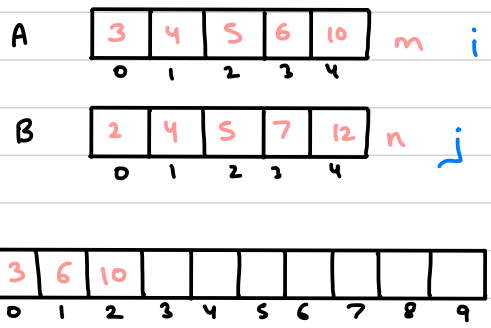
3. DIFFERENCE



A-B

- We want elements of A which are not there in B.
- One by one, take each element of A If it is not present in B, copy it to C, otherwise move to next element

$m * n$
m^2
$O(n^2)$



- Use merge procedure.
- Compare A[i] and B[j], if A[i] is small, copy it to C otherwise increment j.
- If same, don't copy.

$\theta(m+n)$
$\theta(n)$

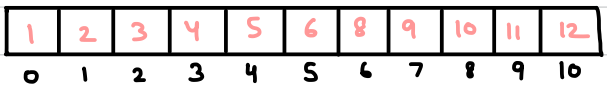
FIND MISSING ELEMENT

1. Single missing element in an sorted array.
2. Multiple missing element in an sorted array.
3. Missing element in unsorted array.

1. Single missing element in an sorted array.

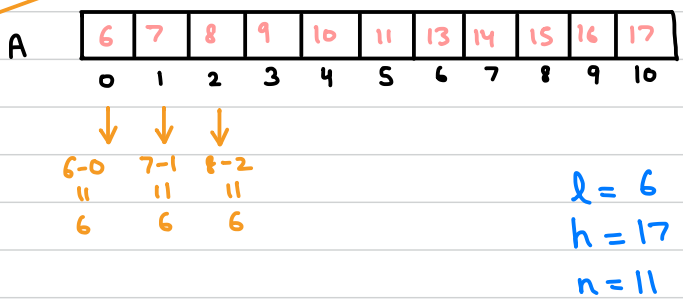
ITERATIVE METHOD

METHOD 1




```
for (i = 0 ; i < 11 ; i++)
        sum += A[i];
S =  n*(n+1)/2 ;      # Formula for sum of
                        n natural numbers
S - sum
78 - 71 = 7
```
↳ Missing num

METHOD 2



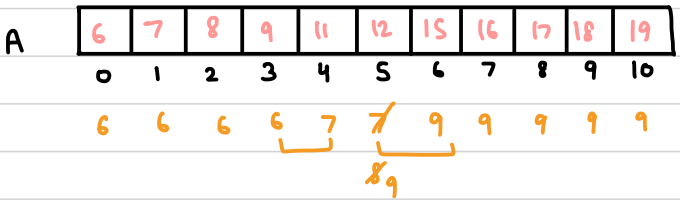
l = 6
h = 17
n = 11
To be known

```
diff = l-0;
for (i=0; i<n; i++)
{
    if (A[i]-i != diff)
    {
        printf(" missing element : %d", i+diff);
        break;
    }
}
```

O(n)

2. Multiple missing element in an sorted array.

METHOD 1




```
diff = 6-0;
for (i=0; i<n; i++)
{
    if (A[i]-i != diff)
    {
        while (diff < A[i]-i)
        {
            printf("%d", i+diff);
            diff++;
        }
    }
}
```

negligible (few elements)

O(n)

METHOD 2

A

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

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12

Hash Table / Bit Set

A table is created of size equal to the largest element of array A. The array is initialized with 0 One by one, each element present in array A, the index of hash array is initialized by 1 corresponding to it.

```
for (i=0; i<n; i++)                    ---------- n
    H[A[i]]++;

for (i=1; i<=h; i++)                   ---------- n
    if (H[i]==0)                       ----------
        printf("%d", i);               2n
                                       O(n)
```

FINDING DUPLICATE IN A SORTED ARRAY

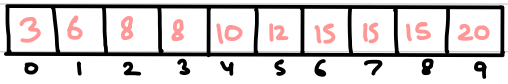
```
lastDuplicate = 0;
```

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

```
for (i=0; i<n; i++)
{
    if (A[i]==A[i+1] && A[i]!=lastDuplicate)
    {
        printf("%d\n", A[i]);
        lastDuplicate = A[i];
    }
}
```

So that if there are more than 1 duplicate, it will print only one time.

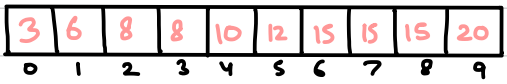
COUNTING NO OF TIMES OF DUPLICATE ELEMENT

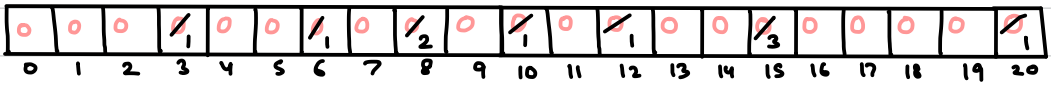



```
for (i=0; i<n-1; i++)
{
    if (A[i] == A[i+1])
    {
        j = i+1;

        while (A[j] == A[i])
            j++;

        printf("%d is appearing %d times", A[i], j-1);
        i = j-1;
    }
}
```

O(n)

FINDING DUPLICATES IN SORTED ARRAY USING HASHING





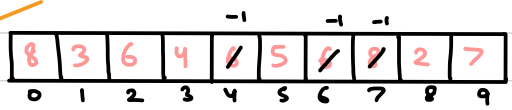

```
for (i=0; i<n; i++)
    H[A[i]]++;

for (i=0; i<=max; i++)
    if (H[i]>1)
        printf("%d %d", i, H[i]);
```

n+n = 2n

O(n)

FINDING DUPLICATES IN AN UNSORTED ARRAY

METHOD 1

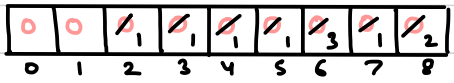



```
for (i=0; i<n-1; i++)
{
        count = 1;

        if( A[i] != -1)
        {
                for (j = i+1; i<n; j++)
                {
                        if (A[i] == A[j])
                        {
                                count ++;
                                A[j] = -1;
                        }
                }

                if (count > 1)
                        printf("%d %d ", A[i], count);
        }
}
```

$O(n^2)$

METHOD 2

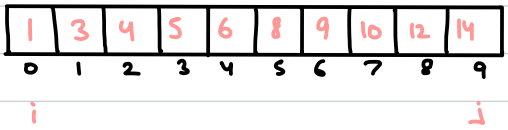
USING HASH TABLE



$$\frac{n + n}{O(n)}$$

FIND PAIR WITH SUM K (a+b=k)

SORTED



Let a+b = 10 (to be searched)

i+j

If i+j > 10
decrement j

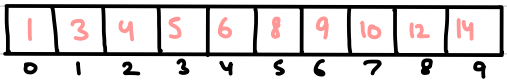
i+j < 10
increment i

i+j = 10
increment i
decrement j

```
i = 0, j = n-1
while (i<j)
{
    if (A[i] + A[i] == k)
    {
        printf("%d+ %d = %d", A[i], A[j], k);
        i++;
        j--;
    }
    else if (A[i] + A[i] < k)
        i++;
    else                                  O(n)
        j--;
}
```

FIND PAIR WITH SUM K (a+b=k)

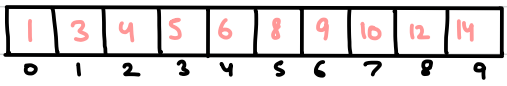
UNSORTED




```
for (i=0; i<n-1; i++)
{                                         O(n²)
    for (j= i+1; j<n; j++)
    {
        if (A[i] + A[j] == k)
            printf("%d+ %d = %d", A[i], A[j], k);
    }
}
```

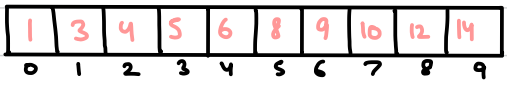
FIND PAIR WITH SUM K (a+b=k)

USING HASHING




```
for (i=0; i<n; i++)
{
        if (H[k-A[i]] != 0)
                printf ("%d + %d = %d", A[i], k-A[i], k);
        H[A[i]]++;
}
```

FIND MAX AND MIN IN SINGLE SCAN




```
min = A[0];
max = A[0];

for (i=1; i<n; i++)
{
        if (A[i]< min)
                min = A[i];
        else if (A[i] > max)
                max = A[i];
}
```

n=10

O(n)

Best = 10, 9, 8, 7, 2, 1 ←

comp = n-1

Worst = 1, 2, 3, 5, 8, 9 →

comp = 2(n-1)

STRINGS

1. Character Sets / ASCII codes
2. Character Array
3. String
4. Creating a string

1. Character Sets / ASCII codes

→ For English Language

American Standard Code for Information Interchange

A - 65
B - 66
⋮
Z - 90

a - 97
b - 98
⋮
z - 122

0 - 48
1 - 49
⋮
9 - 57

UNICODES
2 byte
16 bits
4x4 bits
Represented in form of hexadecimal

↵ enter - 10
Space - 13
esc - 27

0 - 127 Total 128
$2^7 = 128$
7 bits
1 byte

2. Character Array

65 (Stored as 65)

```
Char temp;
temp = 'A';
```

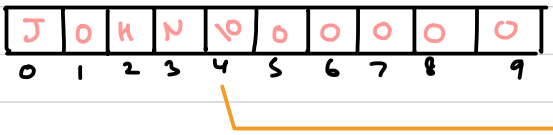
A

```
printf("%c", temp);
```

Displays 'A' and not 65

```
Char X[5];
Char X[5] = {'A','B','C','D','E'};

Char X[5] = {65,66,67,68,69};
```

```
char name[10] = { 'J', 'O', 'H', 'N', '\0' };
```



String delimeter
End of string char
NULL char

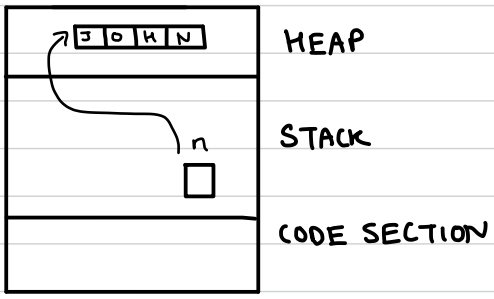
CREATING A STRING

(1) `char name[10] = { 'J', 'O', 'H', 'N', '\0' };`

(2) `char name[] = { 'J', 'O', 'H', 'N', '\0' };` Size of array = 5

(3) `char name[] = "John";` compiler takes null character on its own

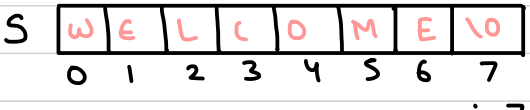
(4) `char *n = "John";`



DISPLAYING A STRING

```
char name[10] = "David";
printf("%s", name);
scanf("%s", name);   → cannot read after space
gets(name)           → can read until you hit enter
```

FINDING LENGTH OF STRING

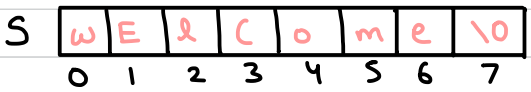



```
int main()
{
        char *s = "welcome";   // No size is required
        int i;
        for (i=0; s[i]!='\0'; i++)
        {
                // This will remain empty
        }
        printf("Length is %d", i);
}
```

CHANGING CASE OF A STRING

```
int main()
{
    char A[] = "WELCOME";
    int i;
    for(i=0; A[i]!='\0'; i++)
        A[i] = A[i] + 32;
    printf("%s", A);
}
```

TOGGLE CASES




```
int main()
{
    char A[] = "WElCome";
    int i;

    for(i=0; A[i]!='\0'; i++)
    {
        if( A[i]>=65 && A[i]<=90)
            A[i] += 32;
        else if( A[i]>='a' && A[i]<=122)
            A[i] -= 32;
    }
}
```

COUNTING NO OF VOWELS AND CONSONANTS

```
for(i=0; A[i]!='\0'; i++)
{
    if( A[i]=='a' || A[i]=='e'.... )
        vcount ++;

    else if ((A[i]>=65 && A[i]<=90) || (A[i]>=97 && A[i]<=122))
        ccount ++;
}
```

COUNTING NO OF WORDS

```
int main()
{
    char A[] = " How are    you ";
    int i, word = 1;

    for (i=0; A[i]!='\0'; i++)
        if(A[i]==' ' && A[i-1]!=' ')
            word ++;
}
```

white space

↳ when there are more than one space

VALIDATING A STRING

```
int valid ( char * name)
{
    int i;
    for (i=0; name[i]!='\0'; i++)
    {
        if ( ! ( name[i]>=65 && name[i]<=90 ) &&
             ! ( name[i]>=97 && name[i]<=122) &&
             ! ( name[i]>=48 && name[i]<=57))
                return 0;
        else
                return 1;
    }
}

int main()
{
    char * name = " Anil321";
    if ( validate(name))
        printf (" Valid String");
    else
        printf (" Invalid String");
}
```

This type of string is not modifiable

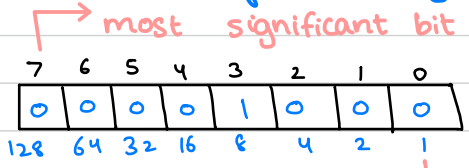
FINDING DUPLICATES IN A STRING

```
int main()
{
        char A[] = "finding";
        int H[26], i;

        for(i=0; A[i]!='\0'; i++)
                H[A[i]-97]+=1;

        for(i=0; i<26; i++)
                if(H[i]>1)
                        printf("%c", i+97);
}
```

FINDING DUPLICATES IN A STRING USING BITWISE OPERATIONS

1. Left Shift <<
2. Bits ORing (Merging)
3. Bits ANDing (Masking)

Manipulating bits of a byte



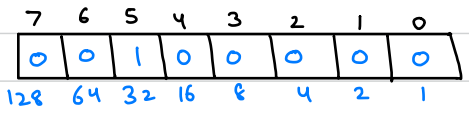
(Represents how 8 is stored)

1 BYTE = 8 BITS

```
char H=8;
```

(1) LEFT SHIFT

H = H << 2



(2) BITS ORing

a = 10 ⟶ 1010
b = 6 ⟶ 0110
1110 ⟶ 14

(3) BITS ANDing

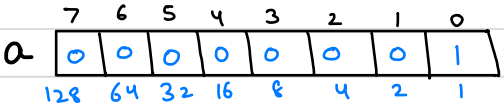
a = 10 ⟶ 1010
b = 6 ⟶ 0110
0010 ⟶ 2

MASKING

Q We want to know whether 4th bit in H is on or off.

Sol

a = 1
a = a<<4
a & H



(perform ANDing)

MERGING

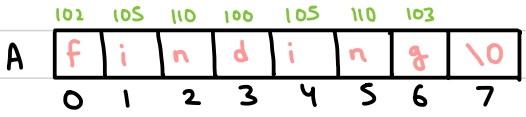
Q we want to set 3rd bit in H as on.

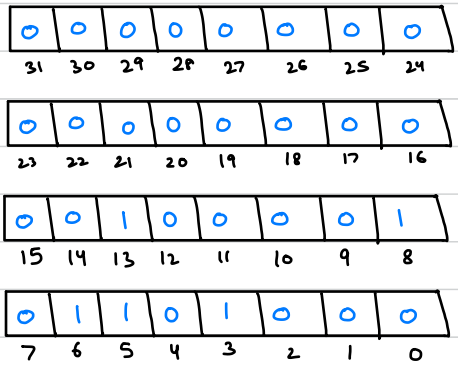
Sol

a = 1
a = a<<2
H = a | H

FINDING DUPLICATES

ASCII Codes





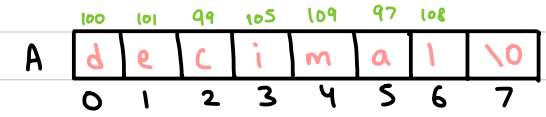
32 bits created for 26 alphabets

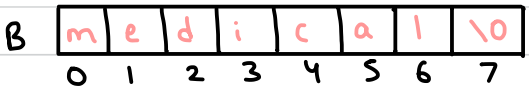
X also consists of 32 bits

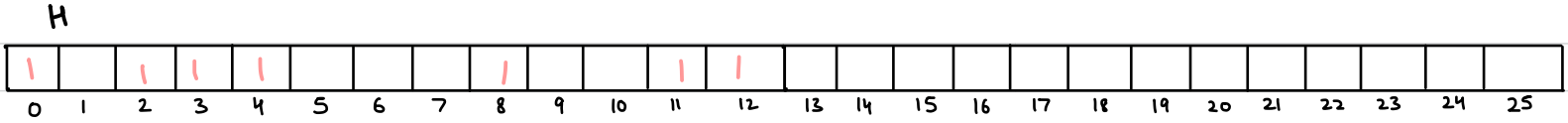
```
int main()
{
    char A[] = "finding";
    long int H=0, x=0;
    for(i=0; A[i]!='\0'; i++)
    {
        x=1;
        x = x<< A[i]-97;
        if(x & H >0)
            printf("%c is duplicate", A[i]);
        else
            H = x | H;
    }
}
```

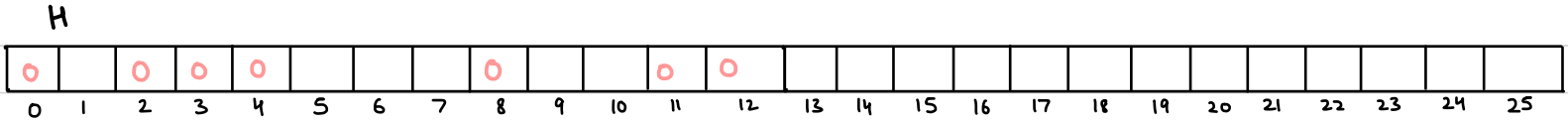
CHECK FOR ANAGRAM

When two strings are made up of same alphabets









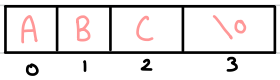

```
int main
{
    char A[] = "decimal";
    char B[] = "medical";
    int i, H[26] = {0};

    for (i=0; A[i]!='\0'; i++)          // Making it 1
        H[A[i]-97] += 1;

    for (i=0; B[i]!='\0'; i++)          // Making it 0
        if (H[A[i]-97] < 0)
        {
            printf("Not Anagram");
            break;
        }

    if (B[i]=='\0')
        printf("Anagram");
```

First check if the strings are of same length.

**

PERMUTATION OF A STRING

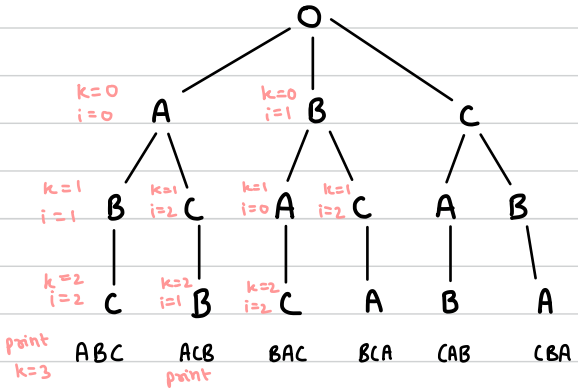
1st Method



3!

n!

State Space Tree



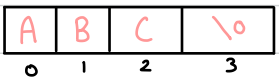
Brute Force: Finding all possible permutations

Back Tracking

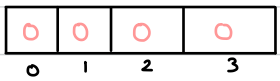
- Implemented using recursion
- To achieve brute force.

```
void perm(char s[], int k)
{
    static int A[10] = {0};
    static char Res[10];
    int i;

    if (s[k] == '\0')
    {
        Res[k] = '\0'
        printf("%s", Res);
    }
    else
    {
        for (i=0; s[i] != '\0'; i++)
        {
            if (A[i] == 0)
            {
                Res[k] = S[i];
                A[i] = 1;
                perm(s, k+1);
                A[i] = 0;
            }
        }
    }
}
```

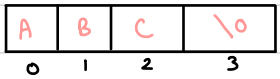
S



A



Res




```
main()
{
    char s[] = "ABC";
    perm(S, 0);
}
```

2nd Method

```
void perm(char s[], int l, int h)
{
    int i;
    if (l == h)
        printf("%s", s);
    else
    {
        for (i = l; i <= h; i++)
        {
            swap(s[l], s[i]);
            perm(s, l+1, h);
            swap(s[l], s[i]);
        }
    }
}
```

MATRICES

SPECIAL MATRICES

↳ Only square matrix
nxn

1. Diagonal Matrix
2. Lower Triangular
3. Upper Triangular
4. Symmetric Matrix
5. Tridiagonal Matrix
6. Band Matrix
7. Toeplitz Matrix
8. Sparse Matrix

1. DIAGONAL MATRIX

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

A

3	7	4	9	6
0	1	2	3	4

ROWS COLUMNS ELEMENT TO BE INSERTED

```
int A[5];
void set ( int A[], int i, int j, int x)
{
        if (i==j)
                A[i-1] = x;
}
void get ( int A[], int i, int j)
{
        if (i==j)
                return A[i-1];
        else
                return 0;
}
```

C++ Class For Diagonal Matrix

```
Class Diagonal()
{
    private:
        int n;
        int *A;

    public:
        Diagonal(int n)
        {
            This->n = n;
            A = new int(n);
        }

        void set(int i, int j, int k);
        void get(int i, int j);
        void Display();
        ~Diagonal()
        {
            delete []A;
        }
};

void Diagonal :: set(int i, int j, int x);
{
    if (i == j)
        A[i-1] = x;
}

int Diagonal :: get(int i, int j)
{
    if (i==j)
        return A[i-1];
    else
        return 0;
}

void Diagonal :: Display()
{
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            if(i==j)
                cout<< A[i-1];
            else
                cout<<" 0 ";
        }
        Cout << endl;
    }
}
```

LOWER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

$M[i,j] = 0$ if $i < j$

$M[i,j]$ = Non Zero if $i >= j$

$$\text{Non Zero} = 1+2+3+4+5$$
$$= 1+2+3+4 \ldots\ldots n$$
$$= \frac{n(n+1)}{2}$$

$$\text{Zero} = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

ROW MAJOR

a_{11}	a_{21}	a_{22}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}	a_{44}	a_{51}	a_{52}	a_{53}	a_{54}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
row1	row2		row3			row 4				row 5				

Index (A[4][3]) = [1+2+3] +2 = 8
Index (A[5][4]) = [1+2+3+4] +3 = 13

$$\text{Index}(A[i][j]) = \left[\frac{i(i-1)}{2}\right] + j - 1$$

COLUMN MAJOR

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	a_{22}	a_{32}	a_{42}	a_{52}	a_{33}	a_{43}	a_{53}	a_{44}	a_{54}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
col 1					col2				col3			col 4		col5

Index (A[4][4]) = [5 + 4 + 3] + 0 = 12
Index (A[5][4]) = [5 + 4 + 3] + 1 = 13
Index (A[5][3]) = [5 + 4] + 2 = 11

$$\text{Index}(A[i][j]) = [n + n-1 + n-2 + \ldots. n - (j-2)] + (i-j)$$

$$= [n(j-1) - [1+2+3+\ldots. j-2]] + (i-j)$$
$$= \left[n(j-1) - \frac{(j-2)(j-1)}{2}\right] + (i-j)$$

UPPER TRIANGULAR MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} \\ 0 & 0 & a_{33} & a_{34} & a_{35} \\ 0 & 0 & 0 & a_{44} & a_{45} \\ 0 & 0 & 0 & 0 & a_{55} \end{bmatrix}$$

$M[i,j] = 0 \quad \text{if} \quad i > j$

$M[i,j] = \text{Non Zero} \quad \text{if} \quad i <= j$

$$\text{Non Zero} = 5+4+3+2+1 = \frac{n(n+1)}{2}$$

$$\text{Zero} = n^2 - \frac{n(n+1)}{2} = \frac{n(n-1)}{2}$$

ROW MAJOR

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{22}	a_{23}	a_{24}	a_{25}	a_{33}	a_{34}	a_{35}	a_{44}	a_{45}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		row1					row2				row3		row4	row5

INDEX (A[4][5]) = [5+4+3]+1 = 13

INDEX (A[i][j]) = [n+n-1+n-2+.... + n-(i-2)] + (j-i)

$$= \left[(i-1)n - \frac{(i-2)(i-1)}{2}\right] + (j-i)$$

COLUMN MAJOR

a_{11}	a_{12}	a_{22}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}	a_{44}	a_{15}	a_{25}	a_{35}	a_{45}	a_{55}
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

INDEX (A[4][5]) = [1+2+3+4]+3 = 13

INDEX (A[i][j]) = [1+2+3+.... j-1] + i-1 = $\left[\frac{j(j-1)}{2}\right] + i - 1$

SYMMETRIC MATRIX

$$M = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 3 & 3 & 3 & 3 \\ 2 & 3 & 4 & 4 & 4 \\ 2 & 3 & 4 & 5 & 5 \\ 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

if $M[i,j] = M[j,i]$

Either we can store lower triangular matrix, or we can store upper triangular matrix

TRI DIAGONAL MATRIX

$$M = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

a_{21}	a_{32}	a_{43}	a_{54}	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{12}	a_{23}	a_{34}	a_{45}
0	1	2	3	4	5	6	7	8	9	10	11	12

lower diagonal | main diagonal | upper diagonal

Main diagonal $i-j=0$

Lower diagonal $i-j=1$

Upper diagonal $i-j=-1$

$|i-j| \leq 1$

$M[i,j]$ = Non Zero if $|i-j| \leq 1$

$M[i,j]$ = Zero if $|i-j| > 1$

$5+4+4$

$n+n-1+n-1$

$3n-2$

Index (A[i][j])

case 1 : if $i-j=1$ index = i-1

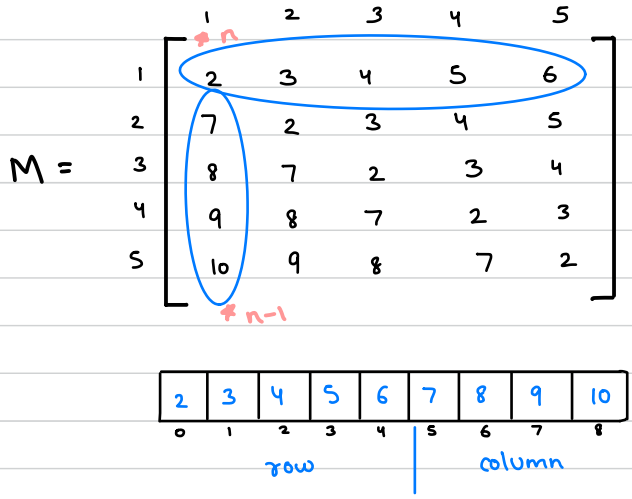
case 2 : if $i-j=0$ index = n-1 + i-1

case 3 : if $i-j=-1$ index = 2n-1 + i-1

SQUARE BAND MATRIX (Same as TRI DIAGONAL matrix)

When there are more than one diagonals below the main diagonal and the number of lower and upper diagonal is equal.

TOEPLITZ MATRIX



$M[i,j] = M[i-1, j-1]$

No of elements we want to store : $n + n-1$*

Index A[i][j]

case 1 : if $i <= j$ Index = $j-1$

case 2 : if $i > j$ Index = $n + i - j - 1$

CREATING A DYNAMICALLY ALLOCATED ARRAY

```
int *A, n;
printf("Enter dimension");
scanf("%d", &n);
A = (int*)malloc(n*sizeof(int));
A = new int[n]; C++
```

SPARSE MATRIX

↳ Having many number of non - zero elements

Methods for storing sparse matrix

1. Coordinate list / Three column representation
2. Compressed sparse row

1. Coordinate list / Three column representation

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

8 x 9 → 72 elements

72 x 2 = 144 bytes

row	column	element
8	9	8
1	8	3
2	3	8
2	6	10
4	1	4
6	3	2
7	4	6
8	2	9
8	5	5

2. Compressed Sparse row

A[3, 8, 10, 4, 2, 6, 9, 5] → Non Zero elements

IA[0, 1, 3, 3, 4, 4, 5, 6, 8] (0 1 2 3 4 5 6 7 8 → Row Number) → Cummulative sum of number of non zero elements in each row

JA[8, 3, 6, 1, 3, 4, 2, 5] → No of column in which each non zero element is located

8 + 9 + 8 = 25 x 2 = 50 bytes

1. Coordinate List / Three column representation (ADDITION)

$A =$

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

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

$B =$

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

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

$C =$

	1	2	3	4	5	6
1	0	0	0	6	0	0
2	0	10	0	0	5	0
3	0	2	2	5	0	7
4	0	0	0	9	0	0
5	12	0	0	0	0	0

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

Q

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

4 x 5 mxn

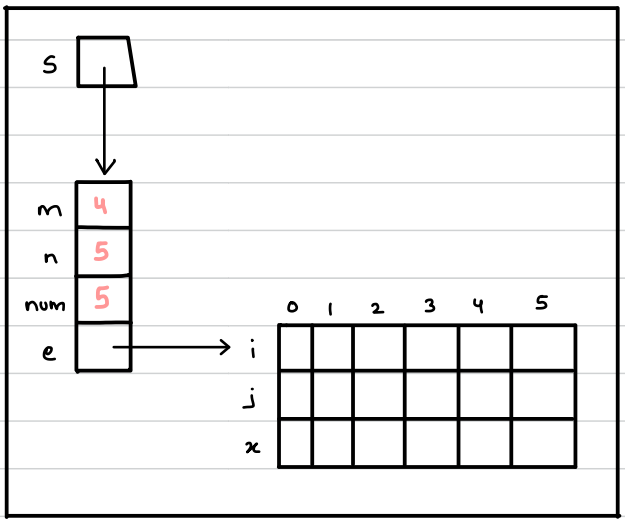
	0	1	2	3	4	5
i	4	1	2	2	3	4
j	5	3	1	4	1	5
k	5	7	2	5	9	4

CREATING SPARSE MATRIX PROGRAM

```
Struct Element
{
        int i;
        int j;
        int x;
};
Struct  sparse
{
        int m;
        int n;
        int num;
        Struct  Element  *e;
};
void main()
{
        Struct sparse  s;
        create (&s);
}
```

```
void create ( struct sparse *s)
{
    int i;
    printf(" Enter dimensions");
    scanf (" %d %d", &s->m, &s->n);
    printf(" Enter number of non-zero elements");
    scanf (" %d", &s->num);
    S->e = new Elements [s->num];
    printf(" Enter all elements ");

    for (i=0; i< s->num; i++)
            scanf(" %d %d %d", &s->e[i].i,
                               &s->e[i].j,
                               &s->e[i].x);
}
```

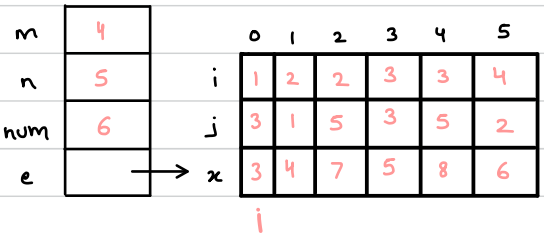
S->e = (Struct Element *) malloc
(s->num* sizeof(Struct Element))



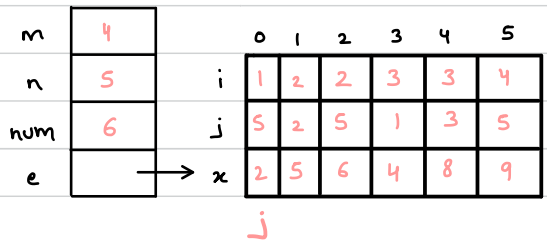
ADDING SPARSE MATRIX PROGRAM

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

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

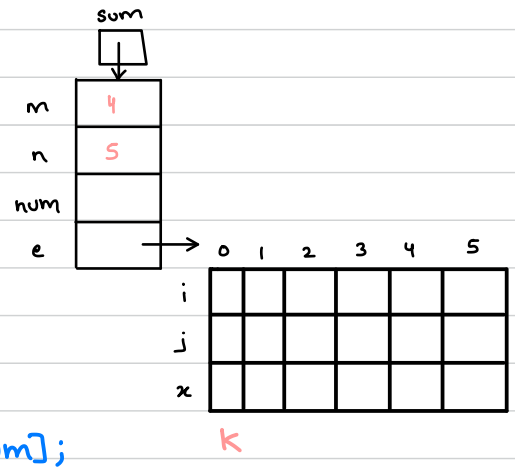


i



j

```
add ( struct sparse *s1 , struct sparse s2)
{
        struct Sparse *sum;

        if ( s1 -> m != s2 -> m || s1->n != s2->n)
                return 0 ;
        sum = new sparse;
        sum -> m = s1 -> m;
        sum -> n = s1 -> n;
        sum -> e = new Element [s1 -> num + s2 -> num];

        while (i < s1 -> num && j < s2 -> num)
        {
                if ( s1. e[i].i < s2. e [j].i )
                        sum -> e[k++] = s1 -> e [i++];

                else if ( s1 -> e[i] .i > s2 -> e[j] .i)
                        sum -> e[k++] = s2 -> e [j++] ;
                else
                {
                        if ( s1 -> e[i] .j < s2 -> e [j] .j)
                                sum -> e [k++] = s1 -> e[i++] ;
                        else if ( s1 -> e[i].j > s2 -> e [j] .j )
                                sum -> e[k++] = s2 -> e [j++];
                        else
                        {
                                sum -> e[k] = s1 -> e [i++] ;
                                sum -> e [k++] .x += s2 -> e [j++] .x ;
                        }
                }
        }
```



k

Rows are Compared

Column compared

POLYNOMIAL REPRESENTATION

1. Polynomial Representation
2. Evaluation of Polynomial
3. Addition of two Polynomials

$p(x) = 3x^5 + 2x^4 + 5x^2 + 2x + 7$

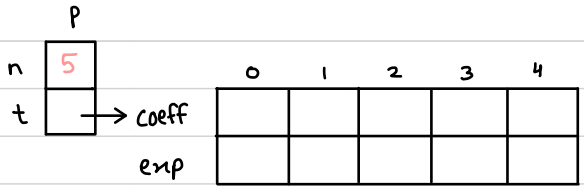
coeff	3	2	5	2	7
exp	5	4	2	1	0

$n = 5$

```
Struct Term
{
    int coeff;
    int Exp;
}
```

```
Struct Poly
{
    int n;
    Struct Term *t;
}
```

```
Struct Poly P;
printf(" No of non-zero terms");
scanf (" %d", &p.n);
p.t = new Term[p.n];
printf(" Enter polynomial terms ");
for(i=0; i<p.n; i++)
{
    printf(" Term No : %d ", i+1);
    scanf("%d %d", &p.t[i].coeff, &p.t[i].Exp);
}
```



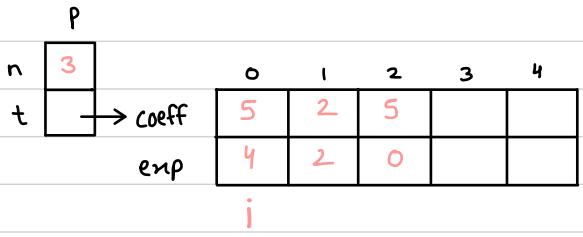
EVALUATION

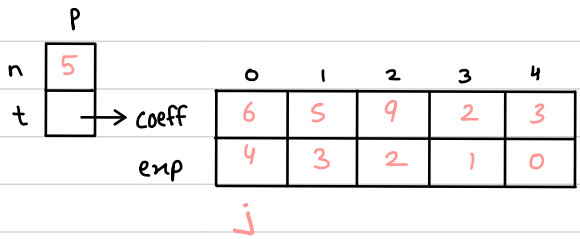
```
Struct Poly P;
x = 5; sum = 0;

for(i=0; i<p.n; i++)
    sum += p.t[i].coeff * pow(x, p.t[i].Exp);
```

POLYNOMIAL ADDITION

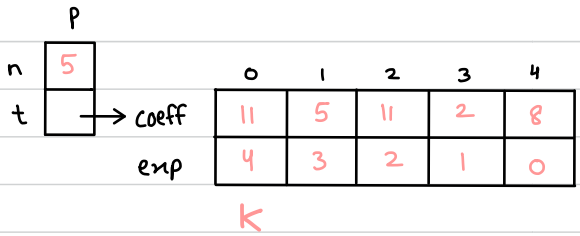
$p1(x) = 5x^4 + 2x^2 + 5$

$p2(x) = 6x^4 + 5x^3 + 9x^2 + 2x + 3$






```
while ( i < p1.n && j < p2.n)
{
        if ( p1.t[i].Exp > p2.t[j].Exp)
                p3.t[k++] = p1.t[i++];

        else if ( p2.t[j].Exp > p1.t[i].Exp)
                p3.t[k++] = p2.t[j++];

        else
        {
                p3.t[k].Exp = p1.t[i].Exp;
                p3.t[k++].coeff = p1.t[i++].coeff + p2.t[j++].coeff;
        }
}
```

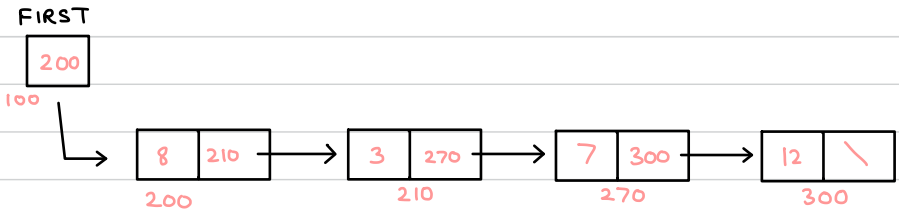


LINKED LIST

1. **Problem with arrays** : Fixed Size

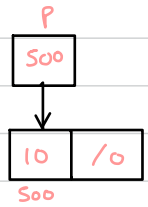
Q What is a linked list?

Ans Linked list is a collection of nodes where each node contain data and pointer to next node.




```
Struct Node                    SELF REFERENTIAL STRUCTURE
{
      int data;          ———— 2
      Struct Node * next; ———— 2      (same as datatype)
};                                4 bytes
```

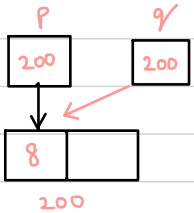
```
Struct Node *p;
p = (struct Node *)malloc(sizeof(Struct Node));
p = new Node;        C++

p → data = 10;
p → next = 0;
```



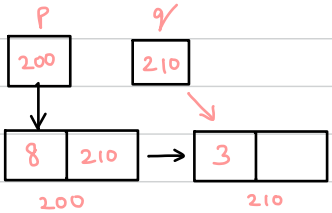
X X

```
Struct Node *p, *q;
```

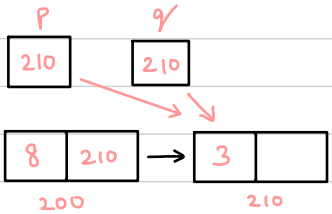
1. q = p



2. q = p → next



3. p = p → next



```
Struct Node *p = NULL;
```

```
if (p == NULL)
if (p == 0)
if (!p)
if (p->next == NULL)
```

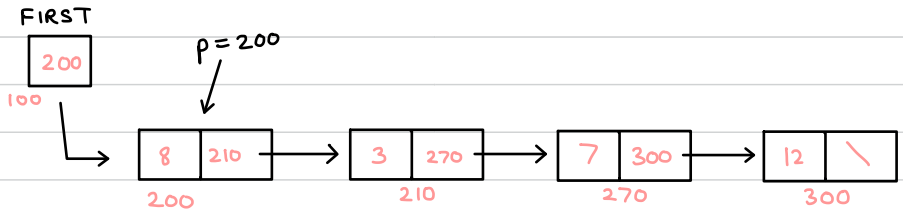
To check if pointer is not pointing anywhere

```
if (p != NULL)
if (p != 0)
if (p)
if (p -> next != NULL)
```

To check if pointer is not NULL.

TRAVERSING THROUGH LINKED LIST

```
Struct Node * p = first;
while (p != 0)
{
        p = p->next;
}
```



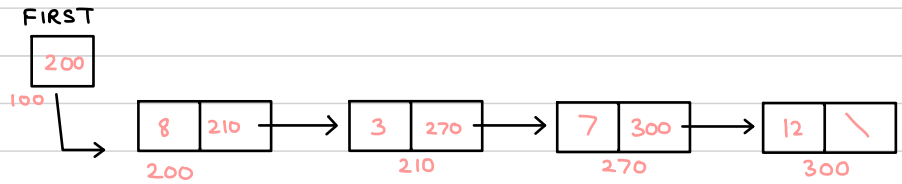

```
#include<stdio.h>
#include<stlib.h>

Struct Node
{
      int data;
      Struct Node *next;
} *first = NULL;
```

Check Struct Node * first = NULL in main

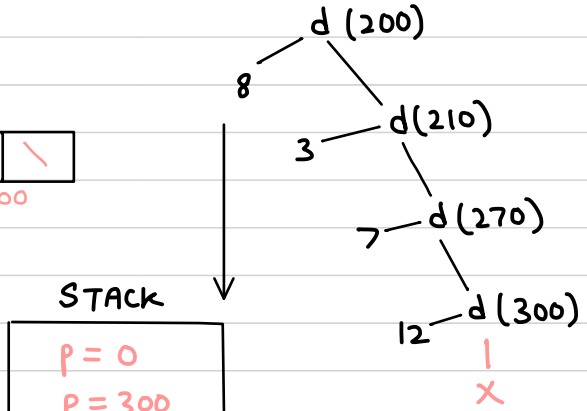
```
void create (int A[], int n)
{
     int i;
     Struct Node *t, *last;
     first = (Struct Node *)malloc (sizeof (Struct Node));
     first->data = A[0]
     first -> next = NULL;
     last = first;
```

```
    for( i=1; i<n; i++)
    {
        t = (struct node *)malloc( sizeof ( struct Node));
        t -> data = A[i];
        t -> next = NULL;
        last -> next = t;
        last = t;
    }
}

void Display ( struct Node *p)
{
    while (p != NULL)
    {
        printf ("%d ", p -> data);
        p = p -> next;
    }
}

void main()
{
    struct Node * temp;
    int A[] = { 3, 5, 7, 10, 25, 8, 32, 2};
    Create (A, 8);
    Display (first);
}
```

RECURSIVE DISPLAY OF LINKED LIST




```
void Display ( struct Node *p)
{
    if (p != NULL)
    {
        printf (" %d", p -> data);
        Display ( p -> next);
    }
}
    Display ( first);
```

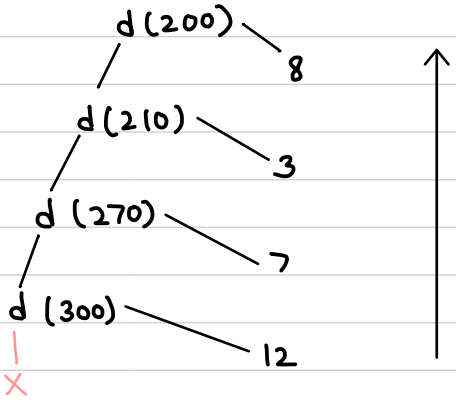


STACK

p = 0
p = 300
p = 270
p = 210
p = 200

```
void Display (struct Node *p)
{
    if (p != NULL)
    {
        Display (p -> next);
        printf(" %d", p -> data);
    }
}
```

O(n)



COUNTING NODES IN LINKED LIST

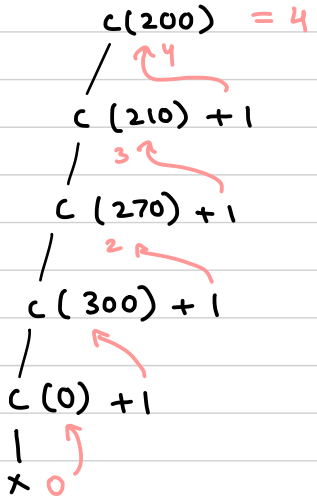
```
int count (struct Node *p)
{
    int c=0;
    while (p!=0)
    {
        c++;
        p = p -> next;
    }
    return (c);
}
```

O(n)

RECURSIVE FUNCTION FOR COUNTING NUMBER OF NODES

```
int count(struct Node *p)
{
    if (p==0)
        return 0;
    else
        return count(p->next)+1;
}
```

O(n)



SUM OF ALL ELEMENTS IN A LINKED LIST

```
int add (Struct Node *)
{
    int sum = 0;
    while (p)
    {
        sum = sum + p -> data;
        p = p -> next;
    }
    return sum;
}
```

USING RECURSION

```
int Add (struct Node *p)
{
    if (p==0)
        return 0;
    else
        return Add (p->next) + p -> data;
}
```

O(n)

MAXIMUM ELEMENT IN A LINKED LIST

```
int max ( struct Node *p)
{
    int m = -32768;
    while (p)
    {
        if (p->data > m)
            m = p->data;
        p = p->next;
    }
    return (m);
}
```

RECURSION

```
int max ( Node *p)
{
    int x = 0;
    if ( p == 0)
        return MIN_INT;
    else
    {
        x = max (p->next);
        if ( x > p->data)
            return x;
        else
            return p->data;
    }
}
```

SEARCHING IN A LINKED LIST

Binary search is not suitable for linked list as we cannot go directly in the middle of the list

LINEAR SEARCH

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

RECURSIVE

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

IMPROVING SEARCHING

```
Node * search ( Node *p, int key)
{
        Node *q = NULL;

        while ( p != NULL)
        {
                if ( key == p->data)
                {
                        q->next = p->next;
                        p->next = first;
                        first = p;
                }
                q = p;
                p = p->next;
        }
}
```

INSERTING IN A LINKED LIST

Before first Node

```
Node *t = new Node;
t->data = x;
t->next = first;
first = t
```

Pos == 4

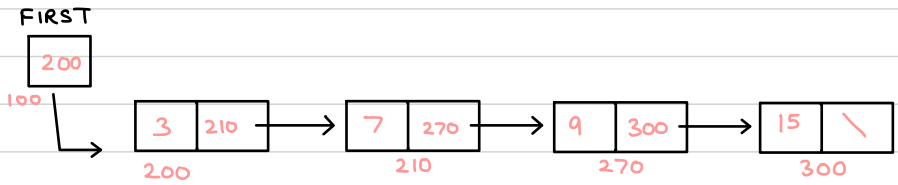
```
Node *t = new Node;
t->data = x;
p = first;
for(i=0; i < pos-1; i++)
        p = p->next;
t->next = p->next;
p->next = t;
```

```
void Insert ( int pos, int x)
{
    Node *t, *p;
    if (pos == 0)
    {
        t = new Node;
        t->data = x;
        t->next = first;
        first = t;
    }

    else if (pos > 0)
    {
        p = first;
        for (i=0; i< pos-1 && p ; i++)
            p = p->next;
        if (p)
        {
            t = new Node;
            t->data = x;
            t->next = p->next;
            p->next = t;
        }
    }
}
```

INSERTING AT LAST

```
void Insertlast (int x)
{
        Node *t = new Node;
        t -> data = x;
        t -> next = NULL;

        if ( first == NULL)
        {
                first = last = t;
        }
        else
        {
                last -> next = t;
                last = t;
        }
}
```

INSERTING IN A SORTED LIST




```
p = first;
q = NULL        (Tailing Pointer)

while ( p && p -> data < x)
{
        q = p;
        p = p -> next;
}
t = new Node;
t -> data = x;
t -> next = q -> next;
q -> next = t;
```

DELETION FROM LINKED LIST

(1) Deletion of first Node

```
Node *p = first;          O(1)
first = first -> next;
x = p -> data;
free (p);
```

(2) Deletion from a given position

```
Node *p = first;          min O(1)
Node *q = NULL;           max O(n)

for (i = 0 ; i < pos-1; i++)
{
    q = p;
    p = p -> next;
}

q -> next = p -> next;
x = p -> data;
free (p);
```

CHECK IF LIST IS SORTED

```
int x = -32768;           O(n)
Node *p = first;
while (p != NULL)
{
    if (p -> data < x)
        return -1;
    x = p -> data;
    p = p -> next;
}
return 0;
```

REMOVE DUPLICATES FROM LIST

```
Node *p = first;                    O(n)
Node *q = first->next;

while (q != NULL)
{
    if (p->data != q->data)
    {
        p = q;
        q = q->next;
    }
    else
    {
        p->next = q->next;
        free(q);
        q = p->next;
    }
}
```

REVERSING A LINKED LIST

(1) Reversing Elements
(2) Reversing Links

Reversing links is preferred over reversing elements because maybe there are lots of values in a single node.

(1) Reversing Elements

First create an array A equal to size of length of linked list.

```
p = first;                  O(n)
i = 0;

while (p != NULL)
{
    A[i] = p->data;
    p = p->next;
    i++;
}
```

```
p = first ; i--;

while (p != NULL)
{
    p->data = A[i--];
    p = p->next;
}
```

(2) Reversing Links

TRACE IT!

```
p = first;
q = NULL;
r = NULL;
while (p != NULL)
{
    r = q;
    q = p;
    p = p->next;
    q->next = r;
}
first = q;
```

Sliding pointers

REVERSING LINKED LIST USING RECURSION

```
void Reverse (Node *q, Node *p)
{
    if (p != NULL)
    {
        Reverse (p, p->next)
        p->next = q;
    }
    else
        first = q;
}
```

CONCATENATING TWO LINKED LIST

```
p = first;                    O(n)
while (p->next != NULL)
        p = p->next
p->next = second;
second = NULL;
```

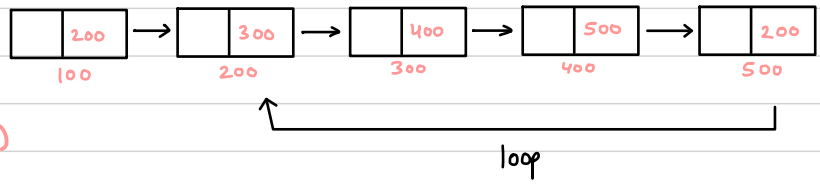
MERGING TWO LINKED LIST

We have two sorted linked list and we want to combine it into a single list.

$\Theta(m+n)$

```
if ( first -> data < second data)
{
        third = last = first;
        first = first -> next;
        last -> next = NULL;
}
else
{
        third = last = second;
        second = second -> next;
        last -> next = NULL;
}

while ( first != NULL && second != NULL)
{
        if ( first -> data < second -> data)
        {
                last -> next = first;
                last = first;
                first = first -> next;
                last -> next = NULL;
        }
        else
        {
                last -> next = second;
                last = second;
                second = second -> next;
                last -> next = NULL;
        }
}
if ( first != NULL )
        last -> next = first;
else
        last -> next = second;
```

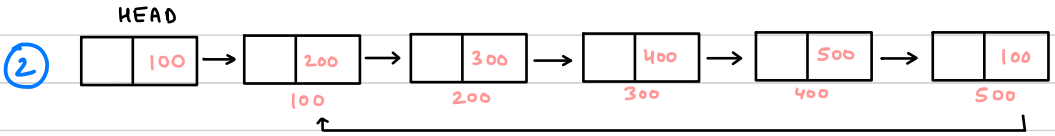
CHECK FOR LOOP IN LINKED LIST



O(n)

```
int isloop ( Struct Node *f)
{
    Struct Node *p, *q;
    p=q= f;

    do
    {
        p = p → next;
        q = q → next;
        q = q ? q → next : q;
    } while ( p && q && p != q);

    if (p == q)
        return 1;
    else
        return 0;
}
```

CIRCULAR LINKED LIST



DISPLAY CIRCULAR LINKED LIST

```
void Display ( Node* p)
{
    do
    {
        printf (" %d", p → data);
        p = p → next ;
    } while ( p != Head);
}
```

Display (Head);

DISPLAY CIRCULAR LINKED LIST USING RECURSION

```
void Display (Node *p)
{
        static int flag = 0 ;
        if ( p != Head || flag = 0)
        {
                flag = 1;
                printf("%d", p -> data);
                Display ( p -> next);
        }
}
```

CREATION OF CIRCULAR LINKED LIST

```
Void create (int A[], int n)
{
        int i;
        Struct Node *t, *last ;
        Head = (Struct Node *) malloc (sizeof (Struct Node));
        Head -> data = A[0];
        Head -> next = Head;
        last = Head;

        for (i= 1; i <n ; i++)
        {
                t = (struct Node *)malloc (sizeof (Struct Node));
                t -> data = A[i];
                t -> next = last -> next;
                last -> next =t;
                last =t;
        }
}

int main()
{
        int A[] = { 2,3,4,5,6};
        create (A , 5);
}
```

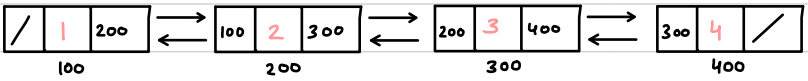
INSERTING IN A CIRCULAR LINKED LIST

```
void Insert ( struct Node *p, int index , int x)
{
        struct Node *t;
        int i;

        if (index < 0 || index > length())
                return;

        if (index == 0)
        {
                t = (struct Node *) malloc ( sizeof (struct Node));
                t -> data = x;

                if ( Head == NULL)
                {
                        Head = t;
                        Head -> next = Head;
                }
                else
                {
                        while ( p -> next != Head )
                                p = p -> next;
                        p -> next = t;
                        Head = t;
                        t -> next = Head;
                }
        }

        else
        {
                for (i = 0; i < index-1 ; i++)
                        p = p -> next;
                t = (struct Node *) malloc ( sizeof (struct Node));
                t -> data = x;
                t -> next = p -> next;
                p -> next = t;
        }
}
```

DELETION IN CIRCULAR LINKED LIST

```
int Delete (struct Node *p, int index)
{
        Struct Node *q;
        int i, x;

        if (index<0 || index > length (Head))
                return -1;

        if (index == 1)
        {
                while (p->next != Head)
                        p=p->next;
                x = Head->data;

                if (Head == p)          // If there is only one node
                {
                        free (Head);
                        Head = NULL;
                }

                else
                {
                        p->next = Head->next;
                        free (Head);
                        Head = p->next;
                }
        }

        else
        {
                for (i=0 ; i < index-2; i++)
                        p = p->next;
                q = p->next;
                p->next = q->next;
                x = q->data;
                free (q);
        }
}
```

INSERT IN A DOUBLY LINKED LIST DOUBLY LINKED LIST




```
Struct Node
{
        Struct Node * prev;
        int data;
        Struct Node * next;
} * first = NULL;


void  Create (int A[], int n)
{
        Struct Node *t, *last;
        int i;

        first = (Struct Node *) malloc ( sizeof ( Struct Node));
        first → data = A[0];
        first → prev = first → next = NULL;
        last = first;

        for (i=1 ; i<n ; i++)
        {
              t = (Struct Node *) malloc ( sizeof ( Struct Node));
              t → data = A[i];
              t → next = last → next
              t → prev= last;
              last → next = t;
              last = t
        }
}
```

INSERT IN A DOUBLY LINKED LIST

(1) Insert at first node

```
Node *t = new Node;
t -> data = x;
t -> prev = NULL;
t -> next = first;
first -> prev = t;
first = t;
```

(2) Insert at any given position

```
Node *t = new Node
t -> data = x;

for (i = 0; i < pos-1; i++)
        p = p -> next;
t -> next = p -> next;
t -> prev = p;
if (p -> next)
        p -> next -> prev = t
p -> next = t;
```

// To check if a node is available after p or not

DELETE FROM A DOUBLY LINKED LIST

(1) Deleting first node

```
p = first;
first = first -> next;
x = p -> data;
delete p;
if (first)    // If first is not NULL
    first -> prev = NULL;
```

(2) Deleting Node from given index

```
p = first;
for (i=0; i<pos-1; i++)
        p = p -> next;
p -> prev -> next = p -> next;

if (p -> next)
        p -> next -> prev = p -> prev;

x = p -> data;
delete p;
```

if p -> next is not NULL

REVERSING A DOUBLY LINKED LIST

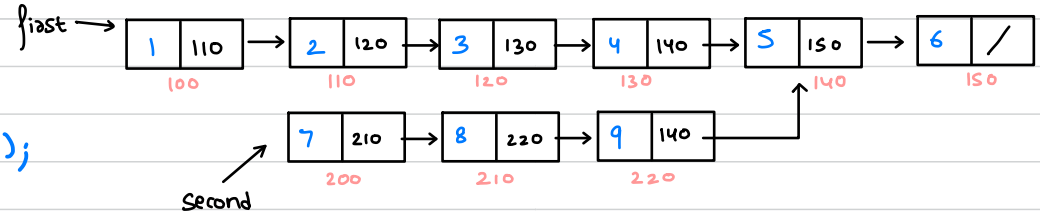
```
p = first;

while (p)
{
        temp = p → next;
        p → next = p → prev;
        p → prev = temp;
        p = p → prev

        if ( p != NULL && p → next = NULL)
              first = p;
 }
```

FINDING MIDDLE NODE IN A LINKED LIST

```
p = q = first;

while (q)
{
       q = q → next;
       if (q)
           q = q → next;
       if (p)
           p = p → next;
}
```

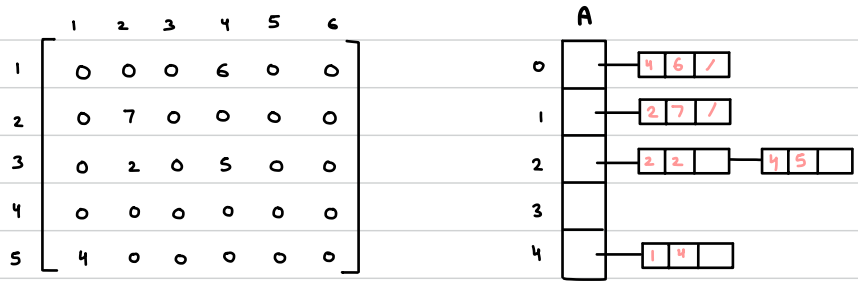
Two pointers p and q will move at the same time.
q will move 2 nodes and p will move 1 node.

FINDING INTERSECTION POINT OF TWO LINKED LIST




```
p = first
while ( p != NULL)
        push (&Stk1, p);
p = second;
while ( p ! NULL)
        push (&Stk2, p);
while ( Stacktop (Stk1) == Stacktop (Stk2))
{
        p = pop(&Stk1);
        pop (&Stk2);
}
```

SPARSE MATRIX USING LINKED LIST



POLYNOMIAL REPRESENTATION USING LINKED LIST

$p(x) = 4x^3 + 9x^2 + 6x + 7$

Recursion uses stack **STACK**

↳ LIFO : Last In First Out

It is a collection of elements that follow LIFO for insertion and deletion.

ADT Stack (Abstract data type)

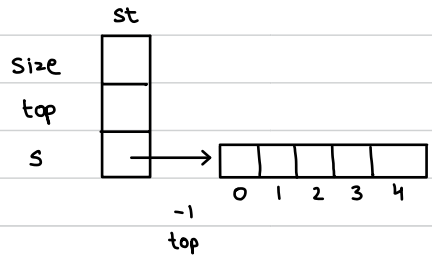
Data : 1. Space for storing elements
2. Top pointer

Implementation of stack using
1. Array
2. Linked List

Operations: 1 push(x)
2. pop()
3. peek(index)
4. StackTop()
5. isEmpty()
6. isFull()

IMPLEMENTATION OF STACK USING ARRAY

```
Struct stack
{
    int size;
    int top;
    int *s;
};

int main()
{
    Struct stack st;
    printf("Enter size of stack");
    scanf("%d", &st.size);
    st.s = new int[st.size];   for heap memory
    st.top = -1;
}
```



Stack empty – if (top == -1)
Stack full – if (top == size-1)

push() O(1)

```
void push (Stack *st, int x)
{
        if ( st -> top == st -> size -1)
                printf (" Stack Overflow ");
        else
        {
                st -> top ++;
                st -> s [st -> top] = x;
        }
}
```

pop() O(1)

```
int pop (Stack *st)
{
        int x = -1;

        if ( st -> top == -1)
                printf (" Stack Underflow");
        else
        {
                x = st -> s [st -> top];
                st -> top --;
        }
        return x;
}
```

peek()

```
int peek ( Stack st, int pos)
{
        int x = -1;

        if ( st top - pos +1 < 0 )
                printf (" Invalid Position");
        else
                x = st . s [st . top - pos+1];

        return x;
}
```

pos	Index = Top - pos + 1
1	3
2	2
3	1
4	0

isEmpty()

```
int isEmpty (Stack st)
{
        if (st . top == -1)
                return 1;
        else
                return 0;
}
```

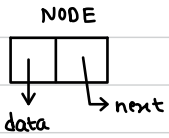
Stacktop()

```
int Stacktop (Stack st)
{
        if (st . top == -1)
                return -1;
        else
                return st . s [st . top];
}
```

isFull()

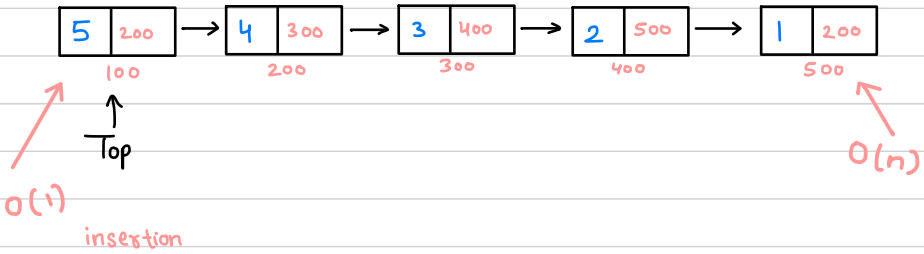
```
int isFull ( Stack st)
{
        if (st . top == -1)
                return 1;
        else
                return 0;
}
```

STACK USING LINKED LIST



Empty : if (top == NULL)
Full : Node *t = new Node;
if (t==NULL)

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



push()

```
void push( int x)
{
    Node * t = new Node;

    if ( t==- NULL)
        printf(" Stack Overflow");
    else
    {
        t -> data = x;
        t -> next = top;
        top = t;
    }
}
```

pop()

```
int pop()
{
    Node *p;
    int x=-1;

    if ( top == NULL)
        printf (" Stack is empty ");
    else
    {
        p = top;
        top = top -> next;
        x = p -> data;
        free(p);
    }
    return x;
}
```

peek()

```
int Peek ( int pos)
{
    int i;
    Node *p = top;

    for (i= 0 ; p != NULL && i <pos-1 ; i++)
        p = p -> next;

    if ( p != NULL)
        return p -> data;
    else
        return -1;
}
```

Stacktop()

```
int  Stacktop()
{
      if(top)
          return  top->data;
      return  -1;
}
```

isFull

```
int  isFull()
{
      Node *t = new  Node;
      int  r = t ? 1 : 0;
      free(t);
      return r;
}
```

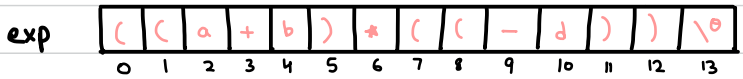
isEmpty

```
int  isEmpty()
{
      return  Top ? 0 : 1;
}
```

False ↙ (0)

↳ True (1)

Paranthesis Matching




```
int  isBalance(char *exp)
{
      Struct  Stack  st;
      st.size = Strlen(exp);
      st.top = -1;
      st.s = new  char[st.size];

      for(i=0; exp[i] != '\0'; i++)
      {
          if(exp[i] == '(')
              push(&st, exp[i]);
          else  if(exp[i] == ')')
              if(isEmpty(st))
                  return false;
              else
                  pop(&st);
      }
      return isEmpty(st)? true : false;
}
```

initializing of stack

ASCII

(	40
)	41
[	91
]	93
{	123
}	125

INFIX TO POSTFIX CONVERSION

1. What is postfix
2. Why postfix
3. Precedence
4. Manual Conversion

1. Infix: Operand Operator Operand
   a + b
2. Prefix: Operator Operand Operand
   +ab
3. Postfix: Operand Operand Operator
   ab +

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, -	1	L-R
*, /	2	L-R
^	3	R-L
— (Unary minus)	4	R-L
()	5	L-R

Eg a + b * c

(a + (b * c)) First fully paranthecise it.

prefix: (a + [*bc]) → [+a*bc]

postfix: (a + [bc*]) → [abc*+]

ASSOCIATIVITY

Left to Right: a + b + c - d → (((a+b)+c)-d)

Right to left: a = b = c = 5 → (a = (b = (c = 5)))

Power operator Example

a ^ b ^ c

(a ^ (b ^ c))

Postfix: (a ^ [bc^])

abc^^

Unary Operators Example

(1) −a negation of a

pre : −a

post : a−

(−(− a))

(2) *p

pre : *p

post : p*

(*(* p))

(3) n!

pre : !n

post : n!

(4) logx

pre : logx

post : xlog

Example : − a + b * logn !

− a + b * log [n!]

− a + b * [n! log]

[a −] + b * [n! log]

[a −] + [bn! log*]

a − bn! log* +

INFIX TO POSTFIX CONVERSION

a + b * c − d / e

SYMBOL	PRECEDENCE	ASSOCIATIVITY
+, −	1	L-R
*, /	2	L-R

Symbol	Stack	Postfix
a		a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
−	−	abc *+
d	−	abc *+d
/	/, −	abc *+d
e	/, −	abc *+ de

abc * +de /− Ans

PROGRAM

infix

a	+	b	*	c	–	d	/	e	\0
0	1	2	3	4	5	6	7	8	9

```
char *convert ( char *infix)
{
        Struct stack st ;     // Initialized
        char *postfix = new char [strlen (infix +1)] ;
        int i=0 ; j=0 ;                          ↳ for null string

        while ( infix [i] ! = '\0')
        {
              if ( isOperand (infix [i]))
                    postfix [j++] = infix [i++];

              else
              {
                    if ( pre (infix [i]) > pre (stacktop (st))
                          push ( &st , infix [i++]);
                    else
                          postfix [j++] = pop ( &t );
              }
        }

        while ( ! isEmpty (st))
                postfix [j++] = pop ( &st);

        postfix [j] = '\0';
        return postfix ;
}
```

```
int pre ( char x)
{
    if ( x == '+' || x == '-')
          return 1;
    else if ( x == '*' || x == '/')
          return 2 ;
    return 0;
}
```

```
int isOperand ( char x)
{
        if (x == '+' || x == '-' || x == '*' || x == '/')
              return 0;
        else
              return 1;
}
```

Q ((a+b)*c) - d^e^f

([ab+]*c) - d^e^f

[ab+c*] - d^e^f

[ab+c*] - d^[ef^]

[ab+c*] - [def^^]

ab+c*def^^-

SYMBOL	OUT STACK PRE	IN STACK PRE
+, -	1	2
*, /	3	4
^	6	5
(	7	0
)	0	?

because of R-L associativity

closing bracket cannot be pushed into Stack

EVALUATION OF POSTFIX

35*62/+4-

SYMBOL	STACK	OPERATION
3	3	
5	5, 3	
*		5*3
	15	
6	6, 15	
2	2, 6, 15	
/		6/2
	3, 15	
+		15+3
	18	
4	4, 18	
-		18-4
	14	

---

x = 6+5+3*4

x = 65+34*+

Here + gets executed first instead of * because presedence and associativity are meant for parenthesisation, they don't decide which operator gets executed first.

PROGRAM FOR EVALUATION OF POSTFIX

postfix	3	5	*	6	2	/	+	4	−	\0
	0	1	2	3	4	5	6	7	8	9

```
int Eval( char *postfix)
{
        Struct stack st;
        int i, x1, x2, r;

        for ( i=0; postfix[i] != '\0'; i++)
        {
            if ( isOperand ( postfix[i]))
                    push (&st, postfix[i]-'0');
            else
            {
                    x2 = pop(&st);
                    x1 = pop (&st);

                    switch ( postfix[i])
                    {
                        case '+' : r = x1+x2 ;  push (&st,r ; break;
                        case '-' : r = x1 - x2 ; push (&st,r ; break;
                        case '*' : r = x1 * x2 ; push (&st,r ; break;
                        case '/' : r = x1 / x2; push (&st,r ; break;
                    }
            }

            return pop(&st);
}
```

because operand will be pushed into the stack in its ASCII value because postfix expression is in char.

Foreg : 3

'51' − '48' = 3

QUEUE

↳ FIFO

(First In First Out)

Queue ADT

Data : (1) Space for storing elements
(2) Front : for deletion
(3) Rear : for insertion

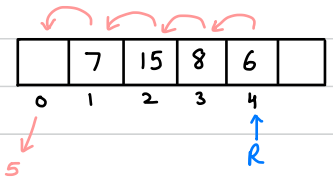
Operations : (1) enqueue (x)
(2) dequeue ()
(3) isEmpty()
(4) isfull()
(5) first()
(6) last()

(1) Array
(2) Linked List

QUEUE USING ARRAY

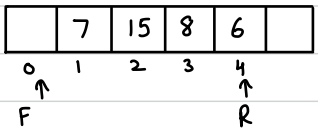
1. Queue using single pointer
2. Queue using front and rear.
3. Drawbacks of queue using array.

1. Queue using single pointer



Insert – O(1)
Delete – O(n)

2. Queue using front and rear



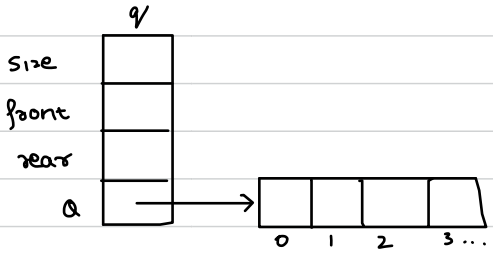
initially front = rear = -1

Insertion : increment rear and insert O(1)
Deletion : increment front and delete O(1)

Empty : if (front == rear)
Full : if (rear = Size-1)

PROGRAM FOR QUEUE USING ARRAY

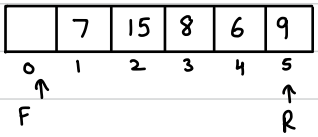
```
Struct Queue
{
    int size;
    int front;
    int rear;
    int * Q;
}

int main ()
{
    Struct Queue q;
    printf(" Enter Size : ");
    scanf (" %d ", &q.size);
    q.Q = new int[q size];
    q.front = q.rear = -1;
}

void enqueue (Queue *q , int x)
{
    if ( q->rear == q->size-1)
        printf(" Queue is full ");
    else
    {
        q->rear++;
        q->Q[q->rear] = x;
    }
}
```




```
Void dequeue ( Queue *q)
{
    int x = -1;

    if (q->front == q->rear)
        printf(" Queue is empty");
    else
    {
        q->front++;
        x = q->Q[q->front];
    }
    return x;
}
```

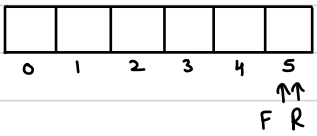
DRAWBACK OF QUEUE USING ARRAY

1. We cannot utilize space of deleted element.



2. Every location can be used only once

3. A situation where queue is empty and full



USING SPACE AGAIN SOLUTION

(1) Resetting Pointers

Whenever Front and Rear are pointing at same place, initialize them as -1

(2) Circular Queue

In circular queue, front and rear initialize with array's first position.

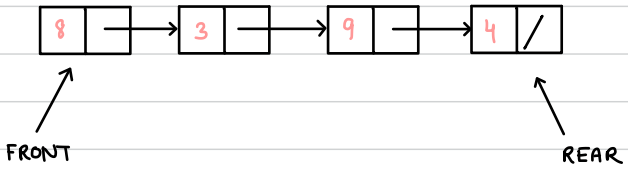
```
void enqueue( struct Queue *q, int x)
{
    if((q->Rear+1)%q->size == q->front)
            printf("Queue is Full");
     else
     {
            q->Rear = (q->rear+1)%q->size;
            q->Q[q->rear] = x;
     }
}
```

To check if rear's next is front

rear	rear = (rear +1) % size
0	(0+1) % 7 = 1
1	(1+1) % 7 = 2
2	(2+1) % 7 = 3
3	(3+1) % 7 = 4
4	(4+1) % 7 = 5
5	(5+1) % 7 = 6
6	(6+1) % 7 = 0

```
void dequeue ( struct Queue *q)
{
    int x=-1
    if( q->front == q->rear)
            printf("Queue is Empty");
    else
    {
            q->front = (q->front+1)%q->size;
            x = q->Q[q->front];
    }
    return x;
}
```

QUEUE USING LINKED LIST



Empty : if (front == NULL)

Full : Node *t = new Node;
if (t == NULL)
// No more nodes can be created
i.e heap is full

```
void enqueue (int x)    O(1)
{
      Node *t = new Node;
      if (t == NULL)
            printf(" Queue is Full");
      else
      {
            t -> data = x;
            t -> next = NULL;
            if (front == NULL)
                  front = rear = t;
            else
            {
                  rear -> next = t
                  rear = t;
            }
      }
}

int   dequeue()    O(1)
{
      int x = -1;
      Node *p;
      if (front == NULL)
            printf(" Queue is Empty");
      else
      {
            p = front;
            front = front -> next;
            x = p -> data;
            free (p);
      }
      return x;
}
```

DEQueue

↳ Double Ended Queue

QUEUE

	Insert	Delete
front	×	✓
rear	✓	×

DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	✓

INPUT RESTRICTED DEQUEUE

	Insert	Delete
front	×	✓
rear	✓	✓

OUTPUT RESTRICTED DEQUEUE

	Insert	Delete
front	✓	✓
rear	✓	×

PRIORITY QUEUES

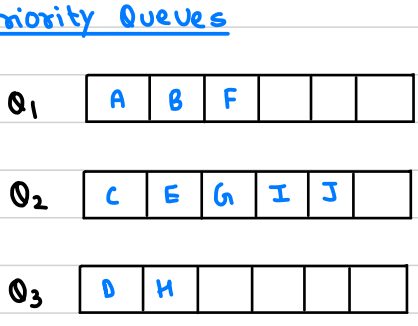
1. Limited Set of priorities
2. Element Priority

1. Limited Set of priorities

Priorities = 3

Element ⟶	A	B	C	D	E	F	G	H	I	J
Priority ⟶	1	1	2	3	2	1	2	3	2	2

Priority Queues



INSERTION

DELETION

For deletion, elements of Q_1 will be deleted first, then Q_2 and then Q_3

Elements will be deleted in FIFO

2. Element Priority

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

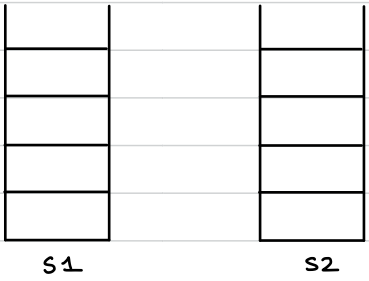
- Element is itself a priority
- Smaller number higher priority

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

2. Insert in increasing order of priority O(n)
   Delete the last element of array O(1)

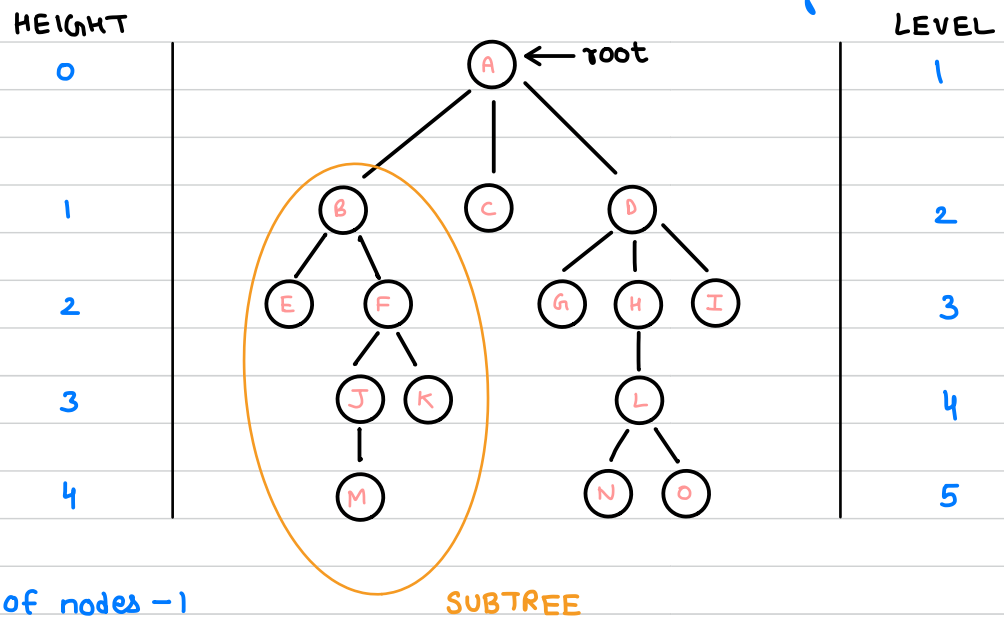
QUEUE using 2 STACKS

assume stacks to be implemented using linked list




```
enqueue ( int x)
{
      push ( &S1, x);
}

int  dequeue()
{
      int  x = -1;
      if ( isEmpty (S2))
      {
            if ( isEmpty S1)
            {
                  printf (" Queue Empty ");
                  return  x;
            }
            else
            {
                  while ( ! isEmpty (S1))
                        push (&S2, pop (&S1));
            }
      }
     return  pop ( &S2);
}
```

TREES → Tree is a collection of nodes and edges.



No of edges = No of nodes − 1

TERMINOLOGY

(1) **Root** : The first node on the top

(2) **Parent** : A node is a parent to its very next descendents.

(3) **Child**

(4) **Sibling** : Children of same parent G, H, I

(5) **Descendents** : Set of all those nodes which can be reached from a particular node.
E, F, J, K, M are descendents of B.

(6) **Ancestors** : For any node, all the nodes along the path from that node to root node
For M, J, F, B, A are ancestors.

(7) **Degree of a node** : No of children it is having L − 2

(8) **Internal Nodes / Non-Leaf Nodes / Non-Terminal Nodes** : Nodes with degree > 0

External Nodes / Leaf-Nodes / Terminal Nodes : Nodes with degree 0

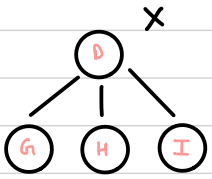
(9) **Levels** (10) **Height**

(10) **Forest** : Collection of trees.

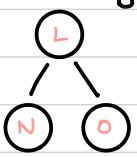
BINARY TREE

degree {2}

children – 0, 1, 2



X

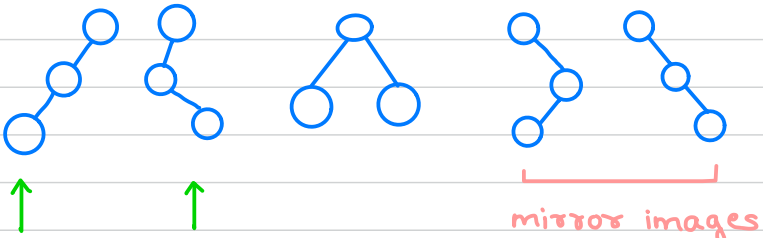


BINARY TREE ✓

NUMBER OF BINARY TREES USING N NODES

(1) Unlabelled Nodes
(2) Labelled Nodes

n = 3



height = 2

mirror images

T(3) = 5

No of trees with maximum height = $4 = 2^2$ (2)

n = 4

T(4) = 14

+ other 7 mirror images

No of trees with maximum height = $8 = 2^3$

$$T(n) = \frac{^{2n}C_n}{n+1}$$

Catalan Number

No of trees with maximum height = 2^{n-1}

2nd Method for finding Catalan Number

n	0	1	2	3	4	5	6
$T(n) = \frac{^{2n}C_n}{n+1}$	1	1	2	5	14	42	

$$T(6) = 1 \times 42 + 1 \times 14 + 2 \times 5 + 5 \times 2 + 14 \times 1 + 42 \times 1$$

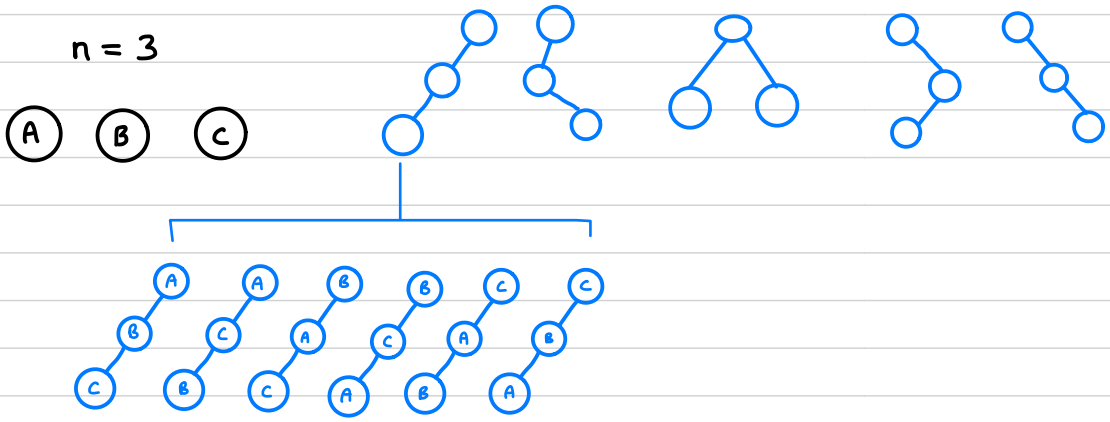
$$T(6) = T(0) \times T(5) + T(1) \times T(4) + T(2) \times T(3) + T(3) \times T(2) + T(4) \times T(1) + T(5) \times T(0)$$

$$= 132$$

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

(2) Labelled Nodes

n = 3



6 = 3!

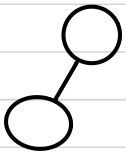
$$T(n) = \frac{^{2n}C_n}{n+1} * n!$$

Shapes — Filling

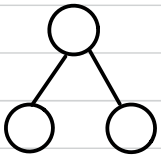
Height vs Node

Height
h = 1

min

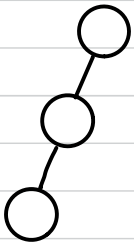


max

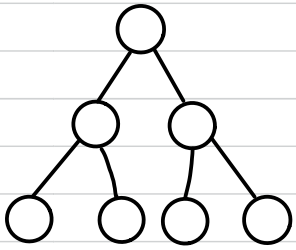


Height
h = 2

min

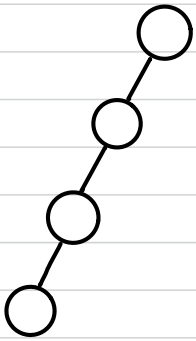


max

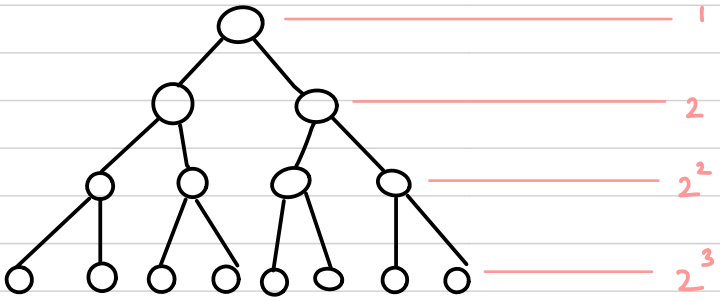


Height h = 3

min



max



$1 + 2 + 2^2 + 2^3 = 15$

Minimum Number of Nodes

min nodes = h+1

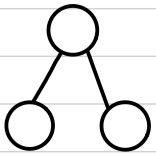
Maximum Number of Nodes

$$a + ar + ar^2 + ar^3 + ar^k = \frac{a(r^{k+1}-1)}{r-1}$$

$$1 + 2 + 2^2 + 2^3 + \ldots\ldots 2^h = \frac{1 \cdot (2^{h+1}-1)}{2-1} = 2^{h+1}-1$$

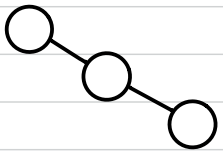
Nodes $n = 3$

min



$h = 1$

max



$h = 2$

If nodes are given

(1) Max height $h = n-1$

↳ Can be obtained by previous formula

(2) For minimum height

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

$$\Rightarrow 2^{h+1} = n+1$$

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

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

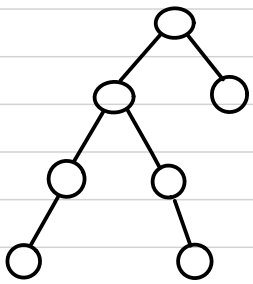
Height of a binary tree

$$\underset{O(\log n)}{\log_2(n+1) - 1} \leq h \leq \underset{O(n)}{n-1}$$

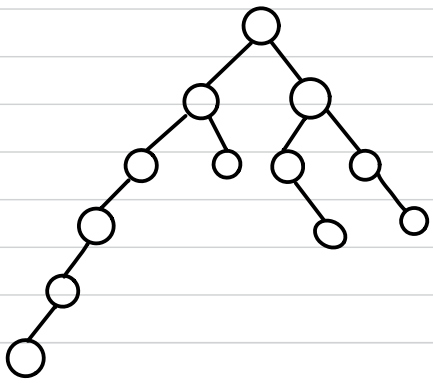
Number of Nodes in a binary tree

$$h+1 \leq n \leq 2^{h+1} - 1$$

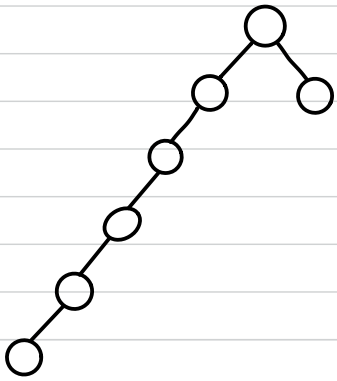
INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE



deg(2) = 2
deg(1) = 2
deg(0) = 3



deg(2) = 3
deg(1) = 5
deg(0) = 4

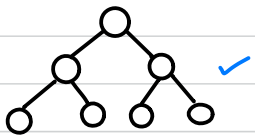


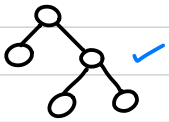
deg(2) = 1
deg(1) = 4
deg(0) = 2

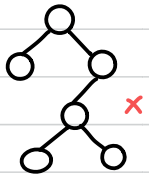
$deg(0) = deg(2) + 1$ True in binary tree

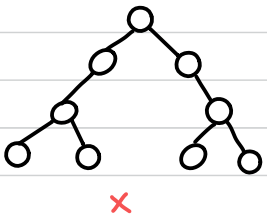
STRICT / PROPER / COMPLETE BINARY TREES

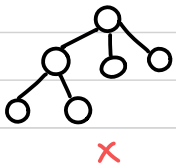
deg = { ✓ 0, ✗ 1, ✓ 2 }

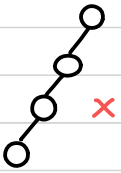








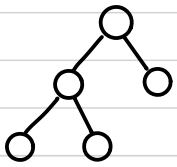




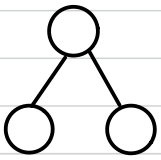
Height vs Node (Strict Binary Tree)

Height
h = 1

min

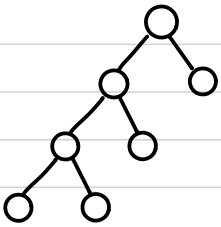


max

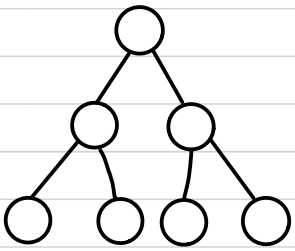


Height
h = 2

min

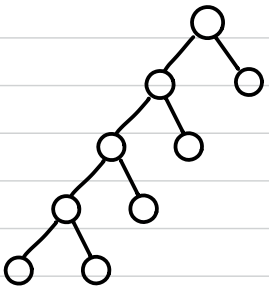


max

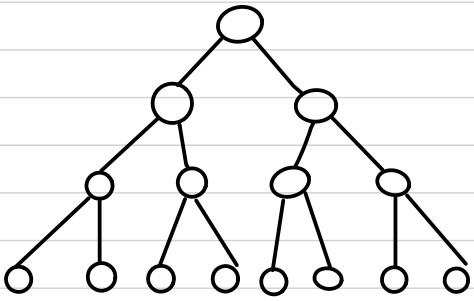


Height h = 3

min



max



Only minimum number changed

If height 'h' is given

Min Nodes $n = 2h+1$

Max Nodes $n = 2^{h+1}-1$

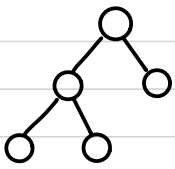
If 'n' nodes are given

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

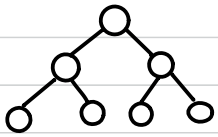
Max height $h = \frac{n-1}{2}$

$$\log_2(n+1)-1 \leq h \leq \frac{n-1}{2}$$

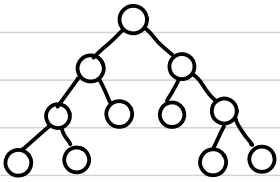
INTERNAL NODES VS EXTERNAL NODES IN A BINARY TREE (STRICT)



i = 2
e = 3



i = 3
e = 4

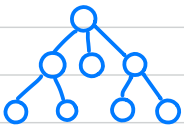


i = 5
e = 6

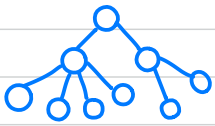
$e = i+1$

n-ary Trees

3-ary Tree {0,1,2,3}



4-ary Tree {0,1,2,3,4}

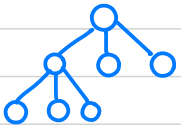


Strict 3-ary Tree {0,3}

Strict n-ary trees

h = 2

min



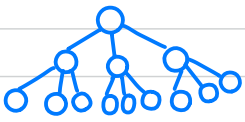
$n = 3+3+1$

$n = 2 \times 3 + 1$

$i = 2$

$e = 5$

max



$i = 5$

$e = 9$

If height 'h' is given

Min Nodes $n = nh+1$

Max Nodes $n = \frac{m^{h+1}+1}{m-1}$

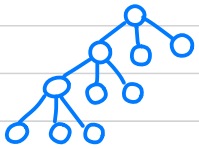
If 'n' nodes are given

Minimum height $h = \log_m[n(m-1)+1]-1$

Max height $h = \frac{n-1}{m}$

h = 3

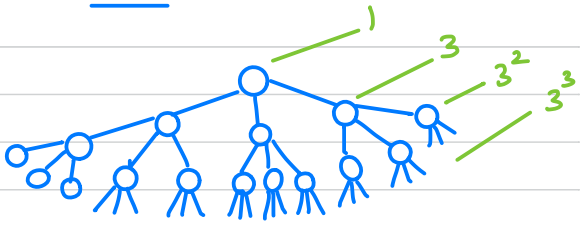
min



$i = 3$

$e = 7$

max



$\Rightarrow 1 + 3 + 3^2 + 3^3$

$\Rightarrow 1 + n + n^2 + n^3$

$\Rightarrow \frac{(n^{h+1}-1)}{n-1}$

$i = 13$

$e = 27$

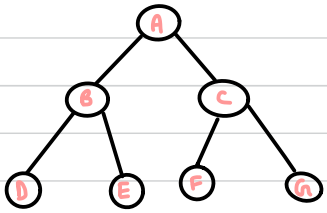
$e = 2i+1$

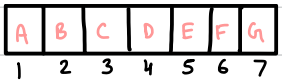
$e = (n-1)i+1$

REPRESENTATION OF BINARY TREE

1. Array Representation
2. Linked Representation

1. Array Representation





Element	Index	L. Child	R. Child
A	1	2	3
B	2	4	5
C	3	6	7
	i	2 * i	2 * i + 1

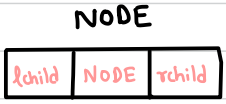
Element ⟶ i

L Child ⟶ 2 * i

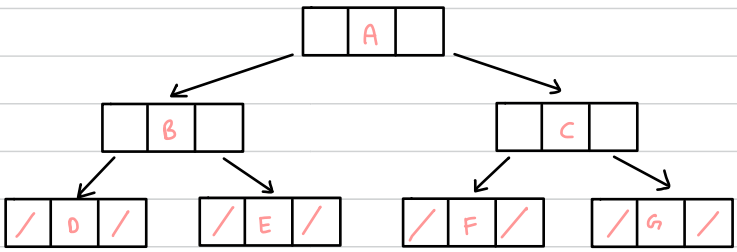
R Child ⟶ 2 * i + 1

Parent ⟶ $\left\lfloor \frac{i}{2} \right\rfloor$

2. Linked Representation




```
Struct Node
{
    Struct Node *lchild;
    int data;
    Struct Node *rchild;
};
```

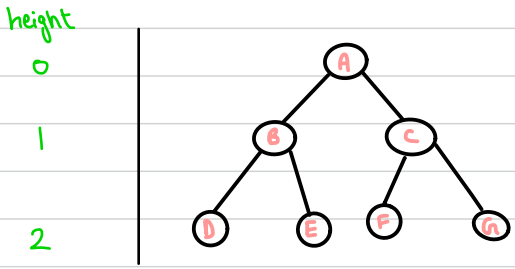


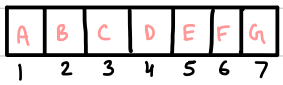
n = 7

nulls = n + 1

Full Binary Tree: A binary tree of height h having maximum number of nodes.

Eg

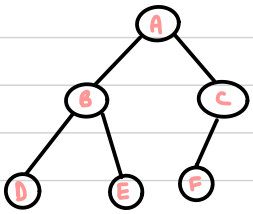


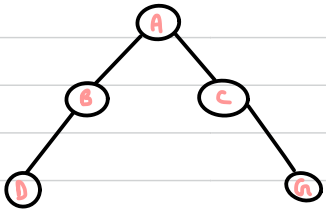


Complete Binary Tree: A complete binary tree of height h will be a full binary tree upto height h-1

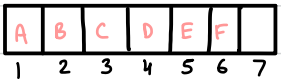
A full binary tree is always a complete binary tree.

Eg

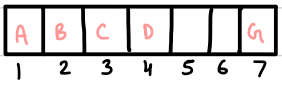




NOT A COMPLETE BINARY TREE



NO BLANK SPACES BETWEEN ELEMENTS

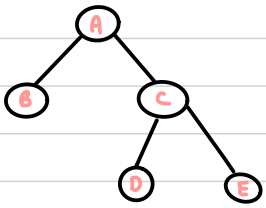


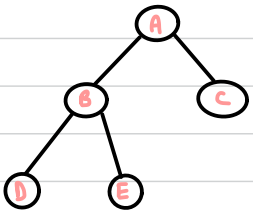
BLANK SPACES

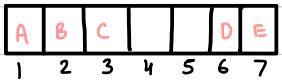
degree {0, 2}

STRICT VS COMPLETE

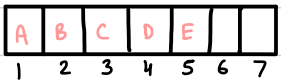
- Complete
- almost complete







Strict ✓
complete ✗



Strict ✓
complete ✓

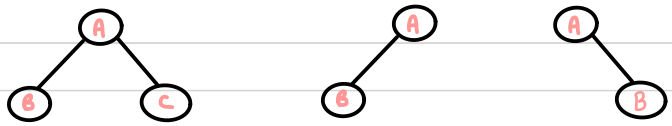
TREE TRAVERSALS

Preorder : NLR
Node , Left , Right

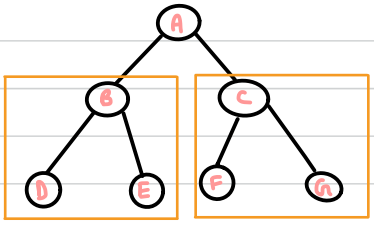
Inorder : LNR
Left , Node , Right

Postorder : LRN
Left , Right , Node

Level Order : Level by Level



Preorder	ABC	AB	AB
Inorder	BAC	BA	AB
Postorder	BCA	BA	BA
Level	ABC	AB	AB



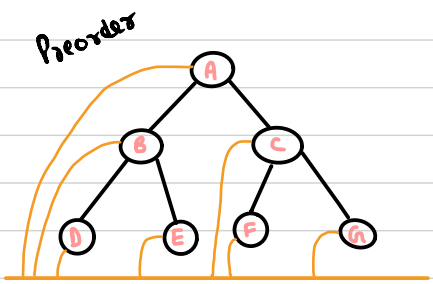
Preorder A , (B, D, E) , (C, F, G)
A, B, D, E , C , F, G

Inorder (D, B, E), A, (F , C , G)
D, B, E , A , F , C, G

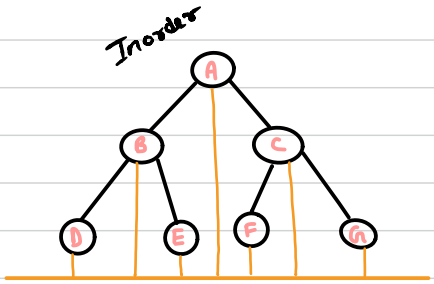
Postorder (D, E , B) , (F, G , C) , A
D , E , B, F , G , C, A

Level A , B, C , D, E, F, G

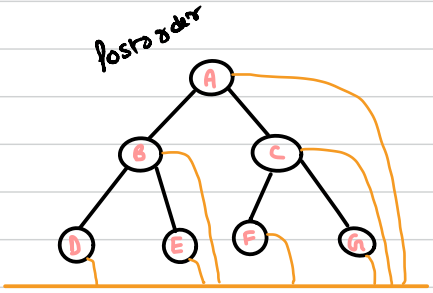
BINARY TREE TRAVERSAL EASY METHOD 1



A, B, D, E, C, F, G

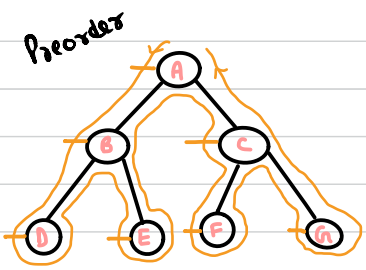


D, B, E, A, F, C, G

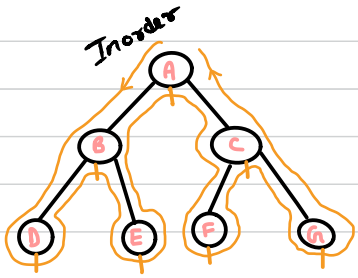


D, E, B, F, G, C, A

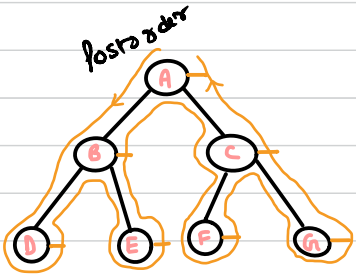
BINARY TREE TRAVERSAL EASY METHOD 2



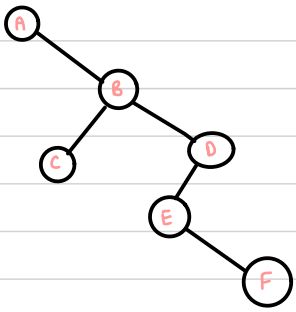
A, B, D, E, C, F, G



D, B, E, A, F, C, G



D, E, B, F, G, C, A



Preorder : A, B, C, D, E, F

Inorder : A, C, B, E, F, D

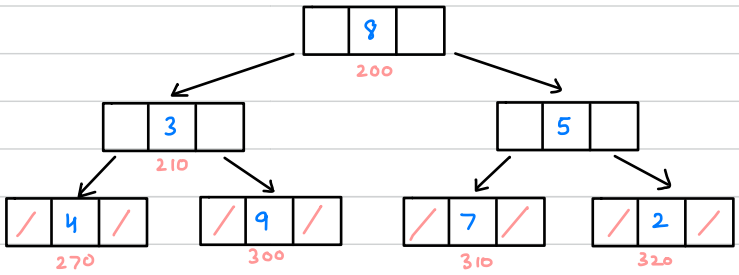
Postorder : C, F, E, D, B, A

PROGRAM TO CREATE BINARY TREE

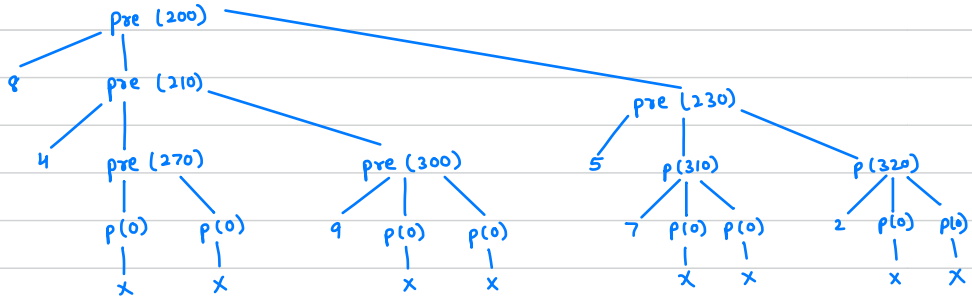
```
void create()
{
    Node *p, *t;
    int x;
    Queue q;        initializing a queue
    printf("Enter root value");
    scanf("%d", &x);
    root = malloc(.....);        initializing root
    root -> data = x;
    root -> lchild = root -> rchild = 0;
    enqueue(root);    Storing address of root in queue

    while(!isEmpty(q))
    {
        p = dequeue(&q);    take out a value from queue into p
        printf("Enter left child");
        scanf("%d", &x);

        if(x != -1)
        {
            t = malloc(....);
            t -> data = x; t -> lchild = t -> rchild = 0;
            p -> lchild = t;      p -> rchild = t      for right child everything
            enqueue(t);                                else will be same
        }
    }
}
```

RECURSIVE PREORDER TREE TRAVERSAL

```
void Preorder (Node *t)
{
    if (t != NULL)
    {
        printf ("%d", t -> data);
        Preorder (t -> lchild);
        Preorder (t -> rchild);
    }
}
```



TRACING TREE



ACTIVATION RECORDS ARE CREATED USING STACK

RECURSIVE INORDER TREE TRAVERSAL

```
void Inorder (Node *t)
{
    if (t != NULL)
    {
        Inorder (t -> lchild);
        printf ("%d", t -> data);
        Inorder (t -> rchild);
    }
}
```

RECURSIVE POSTORDER TREE TRAVERSAL

```
void Postorder (Node *t)
{
    if (t != NULL)
    {
        Postorder (t -> lchild);
        Postorder (t -> rchild);
        printf ("%d", t -> data);
    }
}
```

ITERATIVE PREORDER TRAVERSAL

```
Void Preorder ( Node *t)
{
        Struct stack st;
                                  to check for empty stack
        while ( t != NULL || isEmpty (st) )
        {
                if ( t != NULL)
                {
                        printf(" %d ", t -> data);
                        push ( &st , t );   push address of t in stack
                        t = t -> lchild;
                }

                else
                {
                        t = pop ( &st);   pop out address
                        t = t -> rchild;  from stack
                }
        }
}
```

ITERATIVE INORDER TRAVERSAL

```
Inorder ( Node *t)
        push ( &st , t );
        t = t -> lchild;

        t = pop ( &st);
        printf(" %d ", t -> data);
        t = t -> rchild;
```

ITERATIVE POSTORDER TRAVERSAL*

```
void   Inorder ( Node *t)
{
        Struct stack st;
        long int temp;
        while ( t != NULL || isEmpty (st))
        {
                if ( t != NULL)
                {
                        push ( &st, t);
                        t = t -> lchild;
                }
                else
                {
                        temp = pop (&st);

                if( temp > 0)
                {
                        push (&st, -temp);
                        t=((Node *) temp) -> rchild;
                }
                else
                {
                        printf("%d",((Node *) temp)
                                        ->data)
                        t= NULL;
                }
        }
    }
}
```

LEVEL ORDER TRAVERSAL

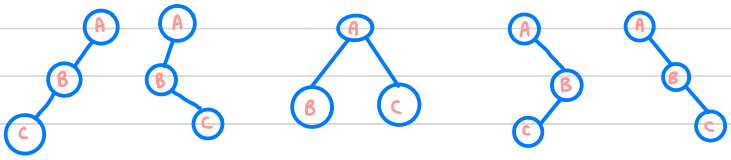
```
void Levelorder (Node *p)
{
        Queue q;
        printf("%d", p->data);
        enqueue (&q, p);

        while( ! isEmpty (q))
        {
                p = dequeue (&q);
                if (p->lchild)
                {
                        printf("%d", p->lchild->data);
                        enqueue (&q, p->lchild);
                }

                if (p->rchild)
                {
                        printf("%d", p->rchild->data);
                        enqueue (&q, p->rchild);
                }
        }
}
```

CAN WE GENERATE TREE FROM TRAVERSALS ?

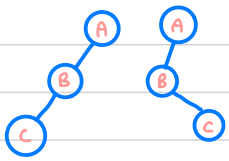
n = 3

① Given preorder - A, B, C

$\frac{2nCn}{n+1}$



② Given preorder A, B, C — more than one
posorder C, B, A



CONCLUSION

If we are given either preorder, inorder, postorder we cannot generate unique tree.

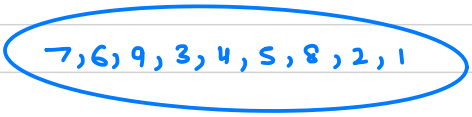
If we are given both preorder and postorder, then also unique tree cannot be generated

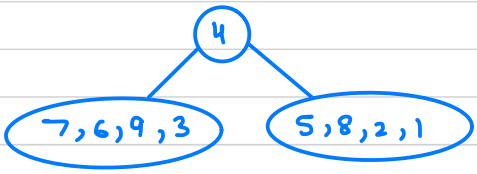
① preorder + inorder
② postorder + inorder

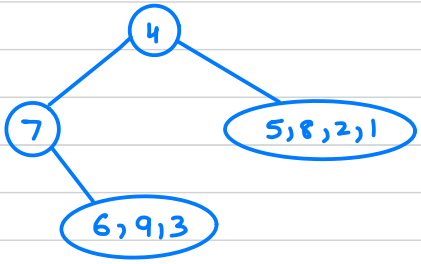
for generating unique tree

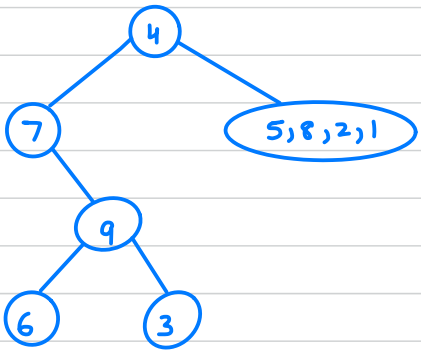
Q Preorder – 4,7,9,6,3,2,5,8,1
Inorder – 7,6,9,3,4,5,8,2,1

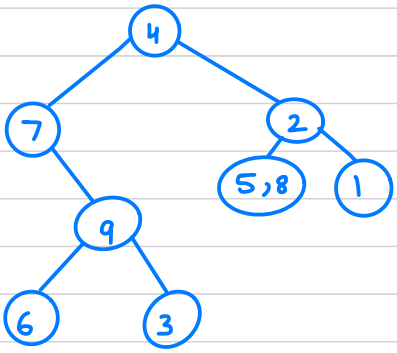
Sol



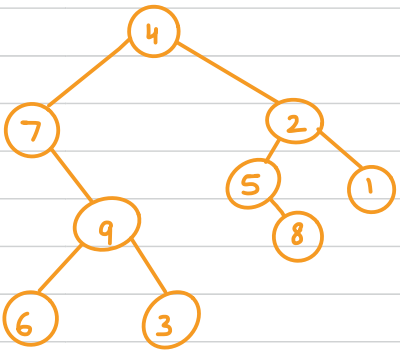






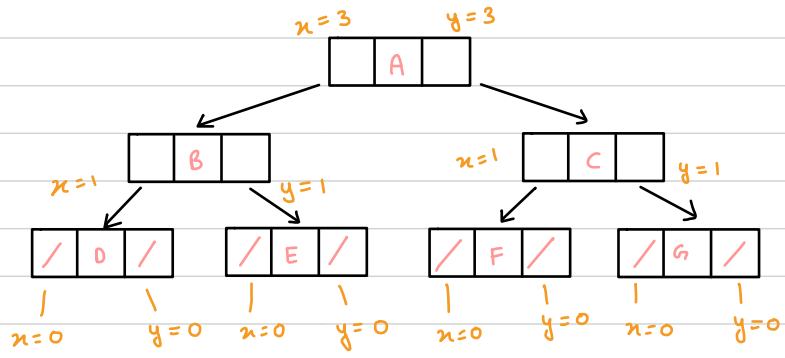


Unique Tree



COUNTING NO OF NODES IN A BINARY TREE

```
int count (Node *p)
{
    int x, y;

    if (p != NULL)
    {
        x = count (p → lchild)
        y = count (p → rchild)
        return x + y + 1;
    }
    return 0;
}
```



COUNTING LEAF NODES

```
int count (Node *p)
{
    int x, y;

    if (p != NULL)
    {
        x = count (p → lchild);
        y = count (p → rchild);

        if(p → lchild == NULL && p → rchild == NULL)
            return x + y + 1;
        else
            return x + y;
    }
    else
        return 0;
}
```

COUNTING NODES WITH DEGREE 2

COUNTING NODES WITH DEGREE 2 OR 1

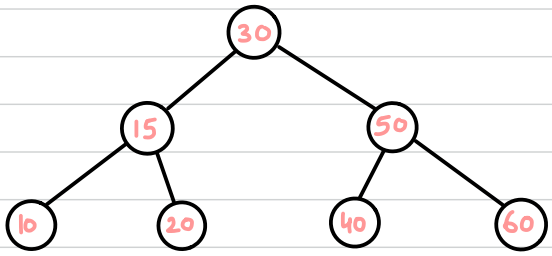
COUNTING NODES WITH DEGREE 1

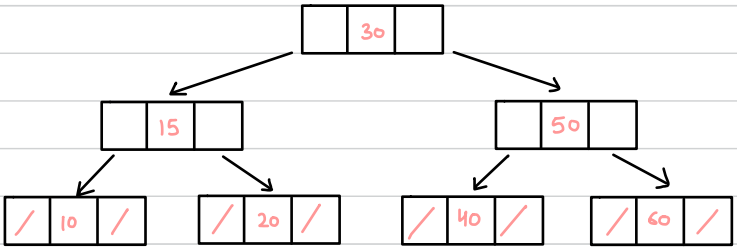
```
if(p → lchild != NULL && p → rchild != NULL)

if(p → lchild != NULL || p → rchild != NULL)

if ((p→lchild != NULL && p→rchild == NULL) ||
    (p→lchild == NULL && p→rchild != NULL))
```

XOR

```
if (p → lchild != NULL ^ p→rchild != NULL)
```

BINARY SEARCH TREE





- Left subtree is smaller
- Right subtree is greater
- Useful for searching (in less number of comparisons)
- No duplicates
- Inorder gives sorted order
- For n number of nodes, catalan number of trees can be generated

$$T(n) = \frac{2n C_n}{n+1}$$

- BST is represented using linked representation.
- Can also be represented using arrays.

SEARCHING IN A BINARY SEARCH TREE (searching takes maximum time same as height of tree) O(logn)

(1) RECURSIVE SEARCH (tail recursion)

O(h)

$\log n \le h \le n$

```
Node * Rsearch(Node *t, int key)
{
    if (t == NULL)              // If element is not found
        return NULL;
    if (key == t->data)
        return t;
    else if (key < t->data)
        return Rsearch(t->lchild, key);
    else
        return Rsearch(t->rchild, key);
}
```

(2) ITERATIVE VERSION O(log n)

```
Node * Search(Node *t, int key)
{
    while (t != NULL)
    {
        if (key == t->data)
            return t;
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    return NULL;
}
```

INSERTING IN BINARY SEARCH TREE

```
void Insert(Node *t, int key)
{
    Node *r = NULL, *p;

    while (t != NULL)
    {
        if (key == t->data)
            return ;          // To avoid duplication
        else if (key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    p = new Node;
    p->data = key;
    p->lchild = p->rchild = NULL;

    if (p->data < r->data)
        r->lchild = p;
    else
        r->rchild = p;
}
```

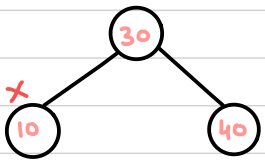
RECURSIVE INSERT IN BST

```
Node* Insert(Node *p, int key)
{
    if (p == NULL)
    {
        t = new Node;
        t->data = key;
        t->lchild = t->rchild = NULL;
    }
    if (key < p->data)
        p->lchild = insert(p->lchild, key);
    else if (key > p->data)
        p->rchild = insert(p->rchild, key);
    return p;
}
```

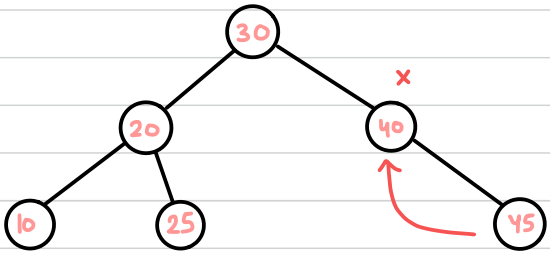
```
void main()
{
    Node *root = NULL;
    root = insert(root, 30);
    insert(root, 20);
    insert(root, 25);
}
```

DELETING FROM BST

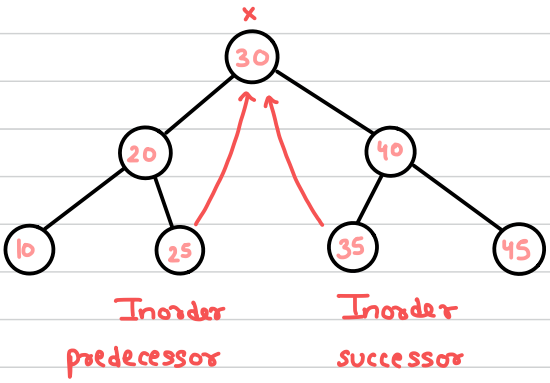
CASE 1 :



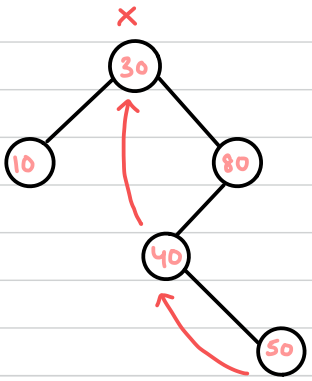
CASE 2 :



CASE 3 :



CASE 4 :



// Multiple changes to be made

```
int Height (struct Node *p)
{
        int x, y;
        if (p == NULL)
                return 0;
        x = Height (p->lchild);
        y = Height (p->rchild);
        return x>y? x+1 : y+1;
}
```

```
Struct Node * InPre (struct Node *p)
{
        while (p && p->rchild != NULL)
                p = p->rchild;
        return p;
}
```

```
Struct Node * InSucc (Struct Node *p)
{
        while (p && p->lchild != NULL)
                p = p->lchild;
        return p;
}
```

```
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

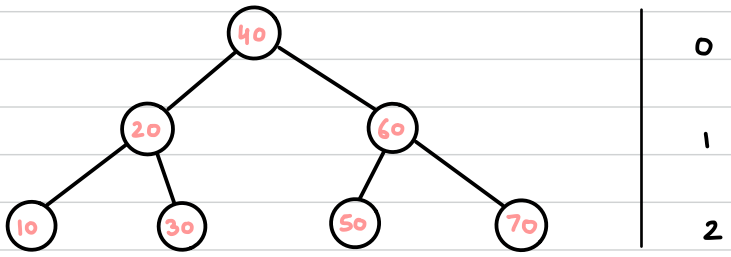
```
Void Createpre(int pre[], int n)        O(n)
{
        Stack st;
        Node *t;
        int i=0;

        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

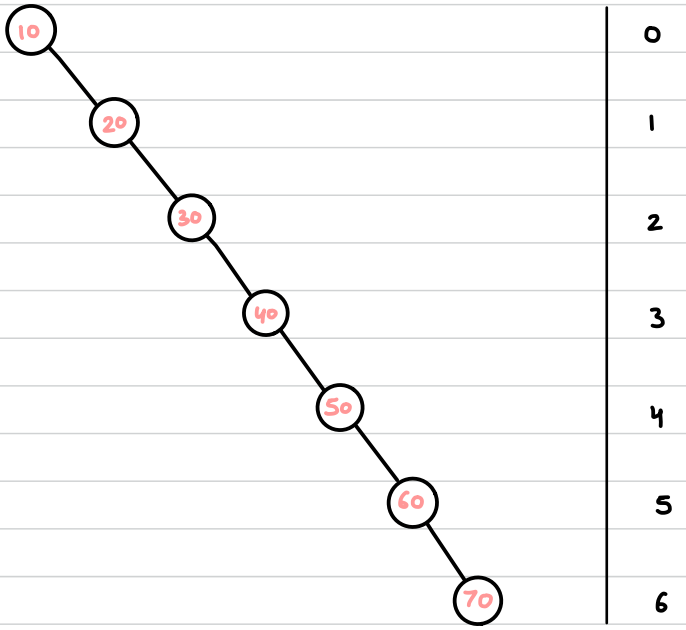
KEYS: 40, 20, 30, 60, 50, 10, 70



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

$O(\log n)$

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



$h = n - 1$

$O(n)$

There is no control over height of binary search tree.

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

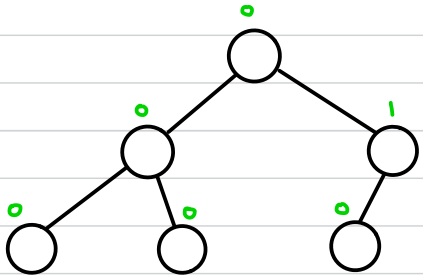
AVL TREES

↳ Height Balanced Binary Search Trees

No of edges →

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

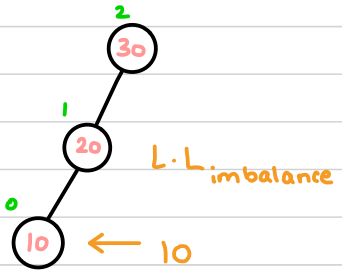
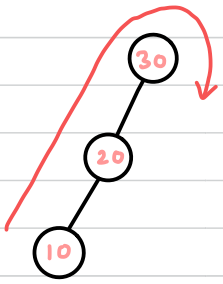
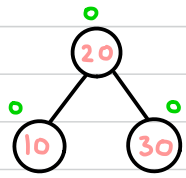
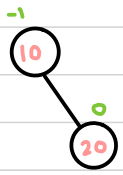
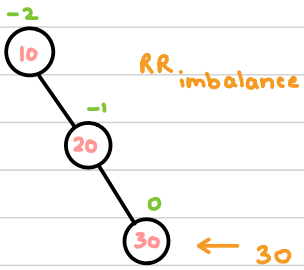
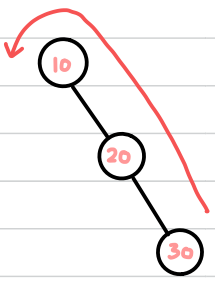
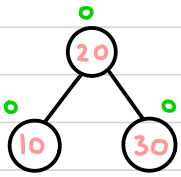
{-1, 0, 1} ⟶ Balanced Tree



Balanced Tree

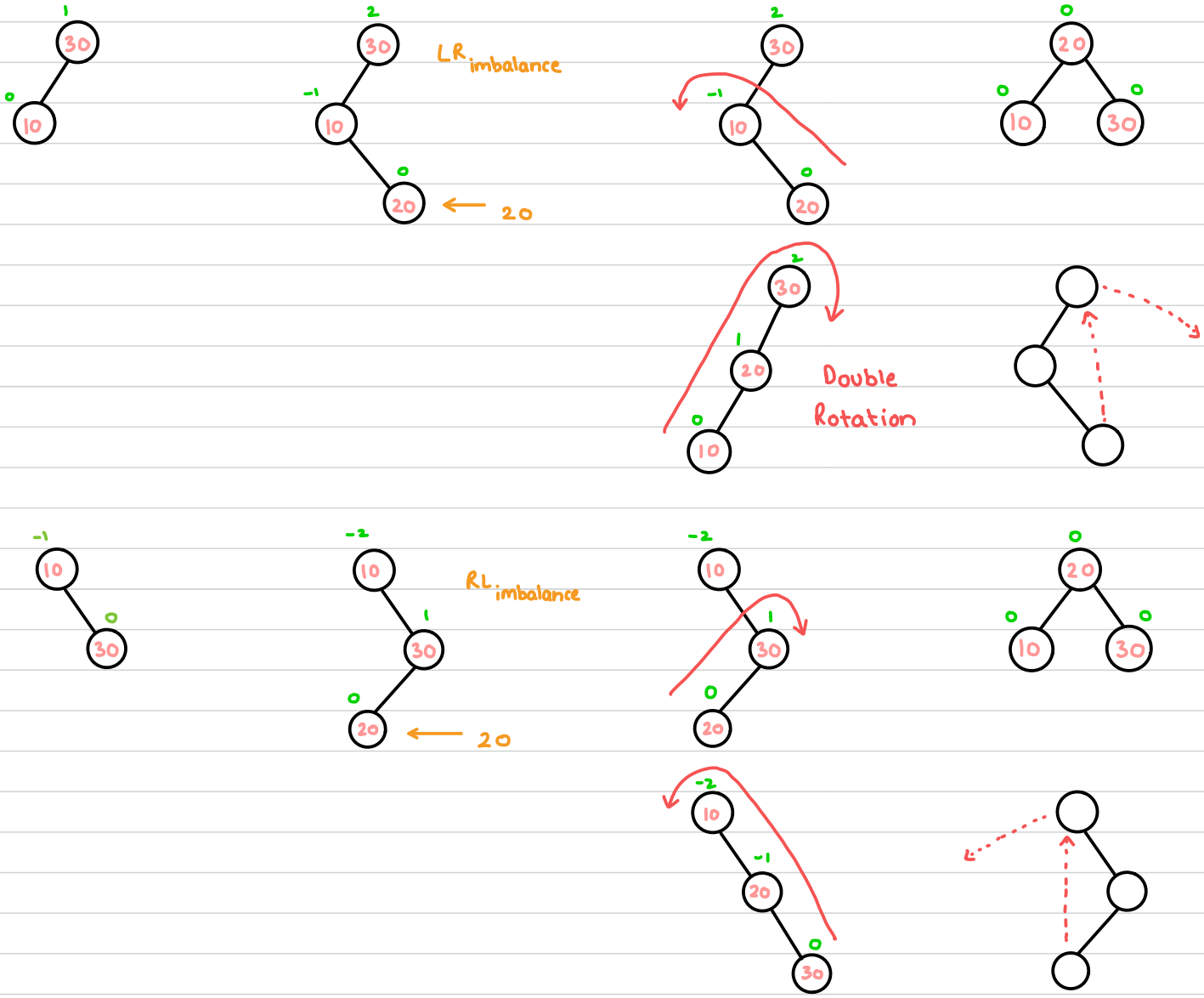
Rotations for insertion in AVL trees

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

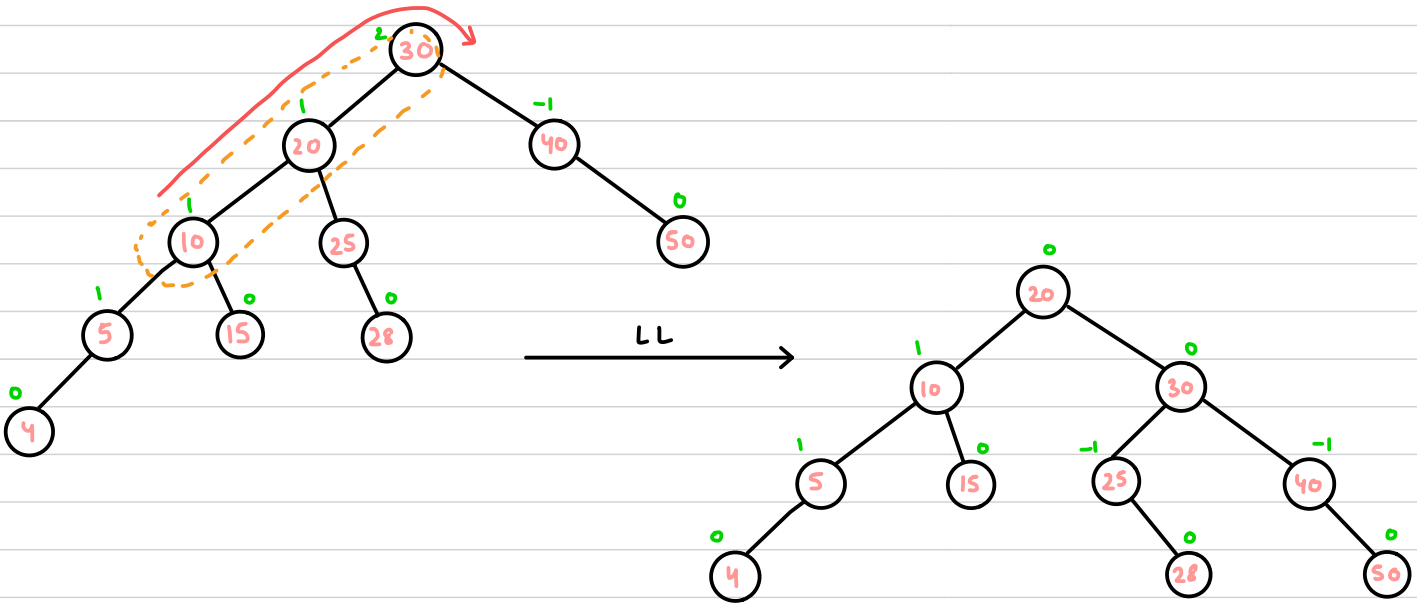
INITIAL	AFTER INSERTION	PERFORM ROTATION	AFTER ROTATION
			
			

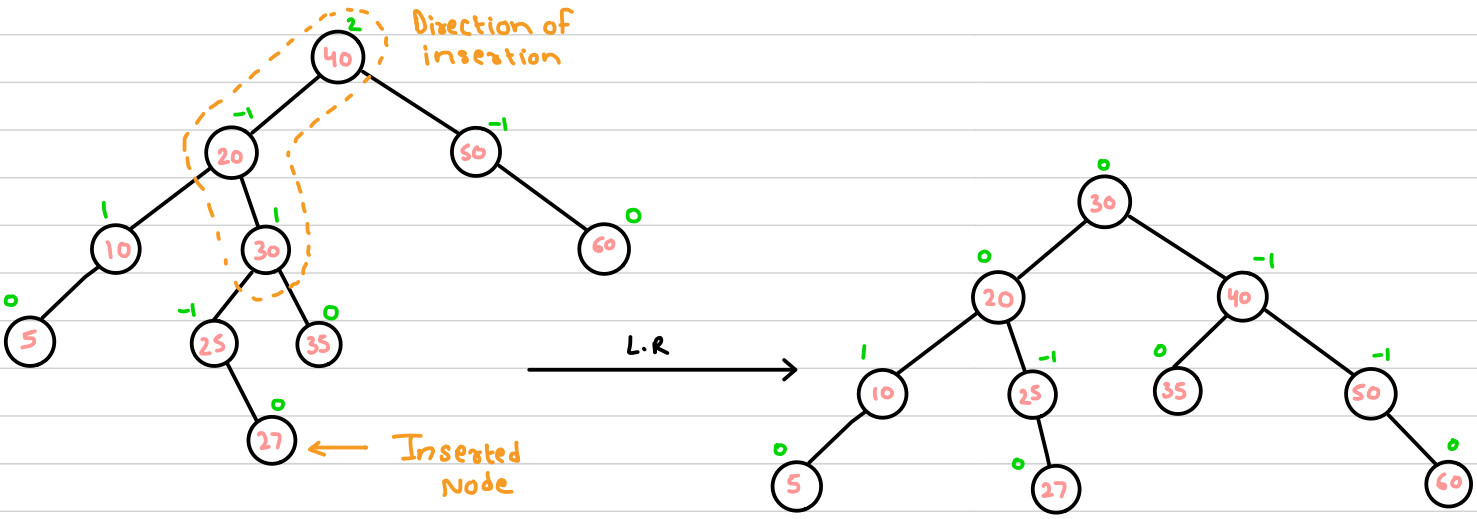
L.L imbalance

RR imbalance



FORMULA OF ROTATION FOR INSERTION

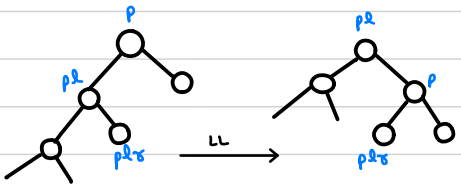




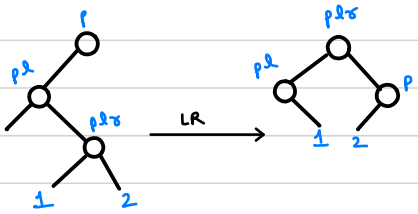
PROGRAM FOR LL ROTATION

```
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);
```



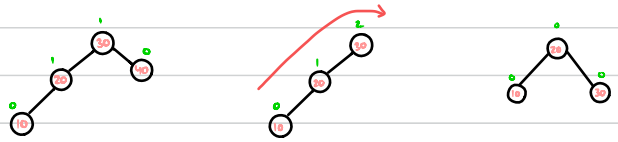
```
    if (root == p)      // If rotation was performed on
        root = pl;      root node, root needs to be
                        updated.
    return pl;
}

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;
}
```



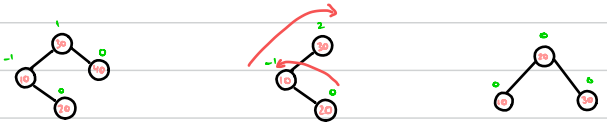
DELETION FROM AVL TREES WITH ROTATION

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

L1

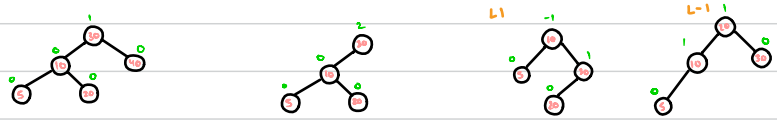


L -1



// Other three will be mirror images

L-0



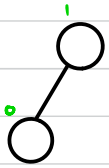
HEIGHT VS NODES OF AVL TREES

If height is given :

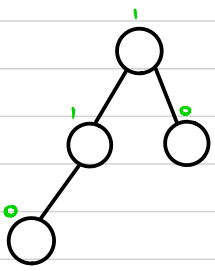
- Max nodes $n = 2^h - 1$ // Not $2^{h+1} - 1$ because height is starting from 1.
- Min nodes n = Look in table

h = 1, n = 1

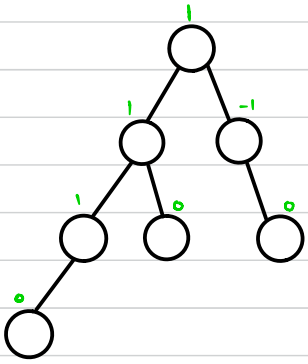
h = 2, n = 2



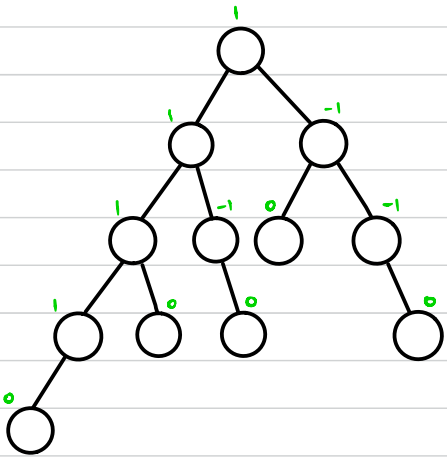
h = 3, n = 4



h = 4, n = 7



h = 5, n = 12



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

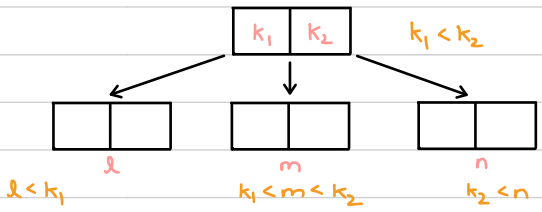
Ex: for 13 nodes h = 5 // Look from node towards height
19 nodes h = 5

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 $\lceil \frac{n}{2} \rceil$ children

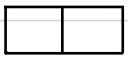
$$\left\lceil \frac{3}{2} \right\rceil = 2$$



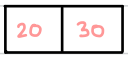
- Cannot have duplicates

CREATION OF 2-3 TREE

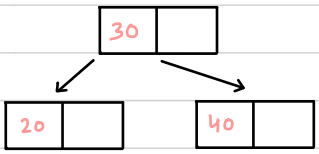
KEYS : 20 , 30 , 40 , 50 , 60 , 10 , 15 , 70 , 80 , 90



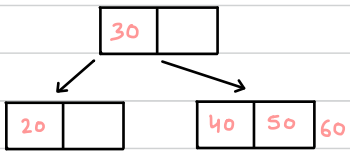
20 , 30

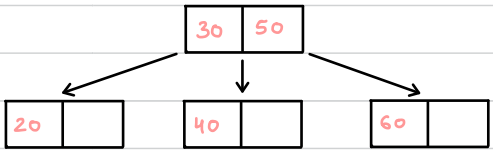


40

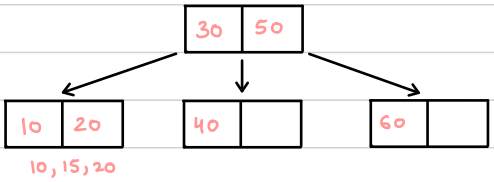


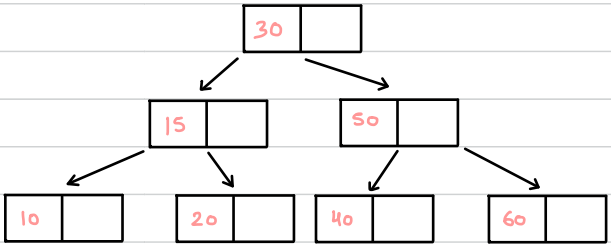
50, 60



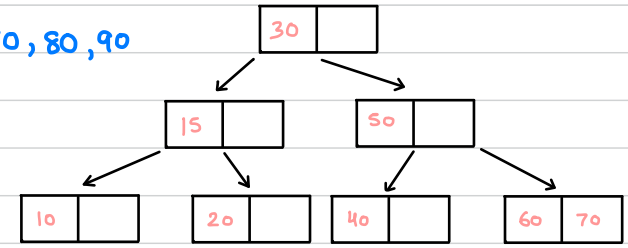


10 , 15

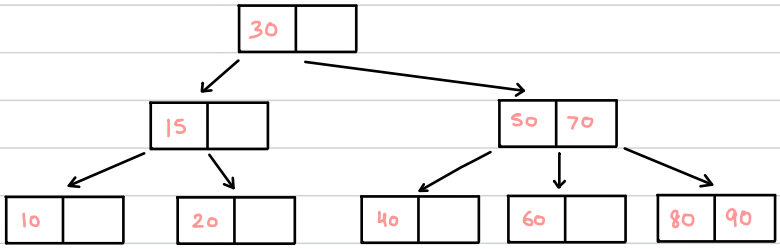




70, 80, 90

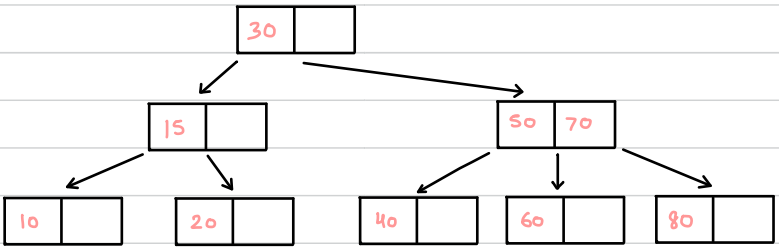


60, 70, 80

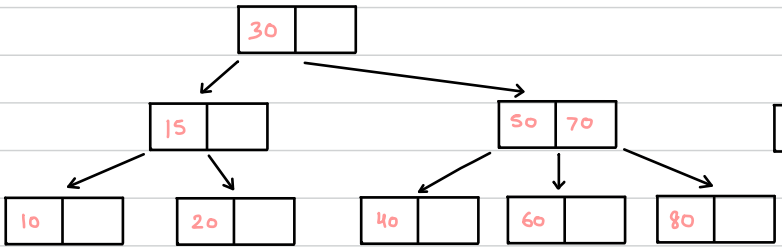


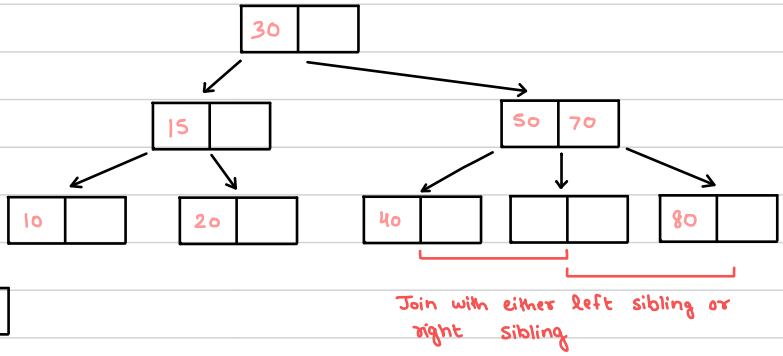
DELETING FROM 2-3 TREE

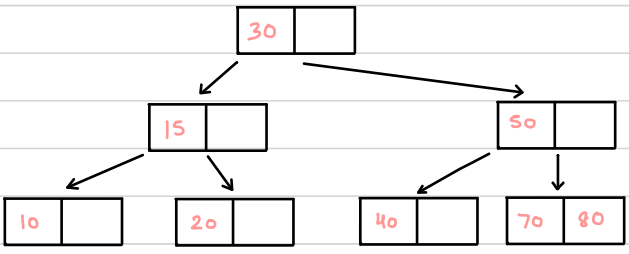
CASE 1 : Simply Delete 90



CASE 2 : Delete and merge 60

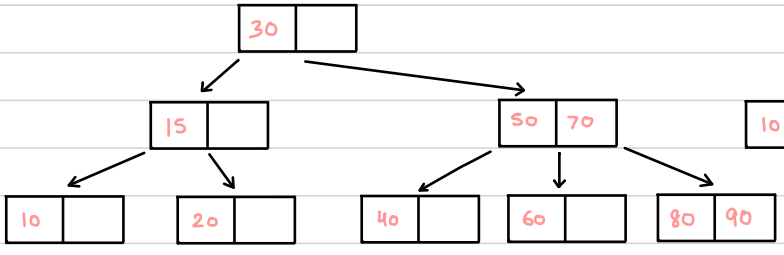


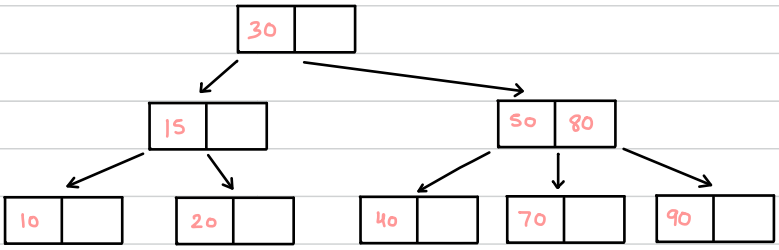




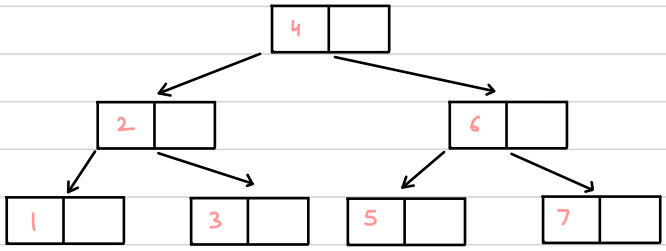
Merging with right sibling

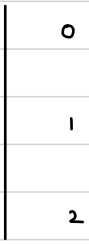
CASE 3 : Borrow 60





ANALYSIS





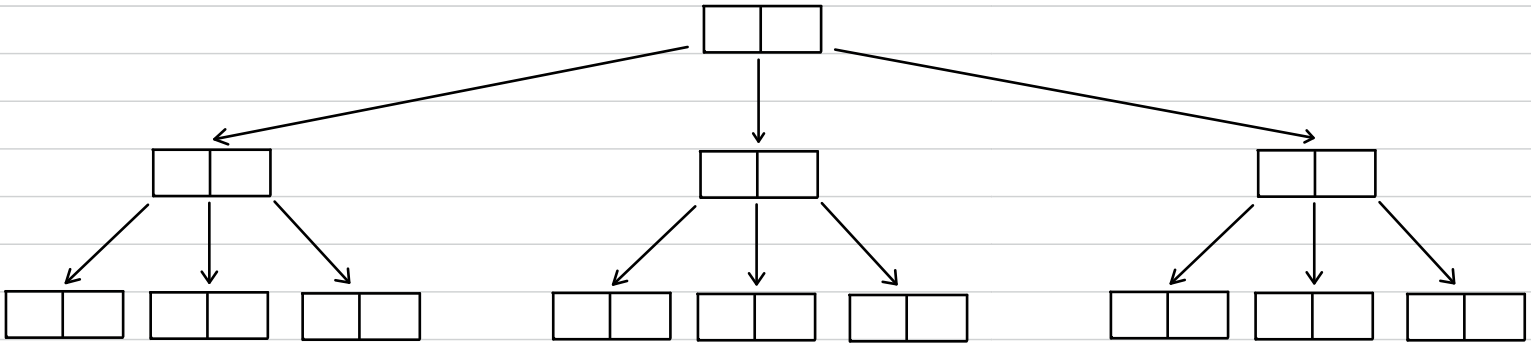
Tree with min nodes for given height

Minimum $n = 1+2+2^2 \ldots$

$= 2^{h+1} - 1$

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

$= O(\log_2 n)$



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

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

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

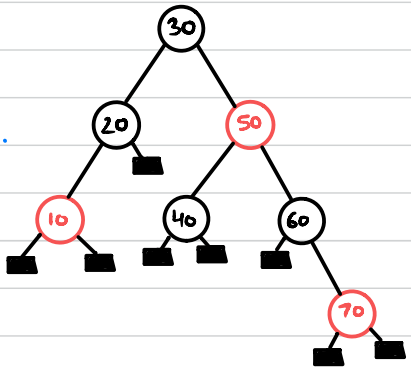
$O(\log_3 n)$

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

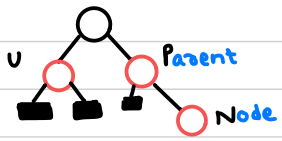
RED BLACK TREE

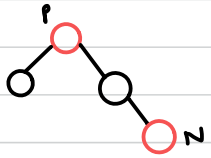
- It is a height balanced Binary Search Tree, similar to 2-3-4 tree.
- Every node is either Red or Black.
- Root of a Tree is Black
- NULL is also Black.
- Number of Blacks on paths from Root to leaf are same.
- No 2 consecutive Red, Parent and Children of red are Black.
- New inserted Node is Red.
-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



CREATION OF RED BLACK TREE

Uncle is Red (for Node)

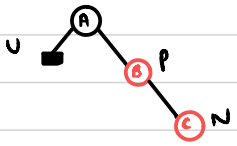


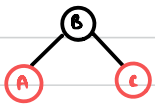


RECOLOURING

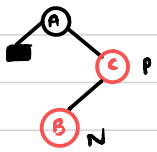
Uncle is Black

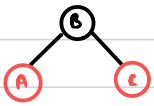
Zig - Zig (LL/RR)





Zig - Zag (RL/LR)





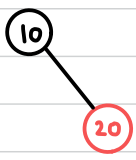
ROTATION

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

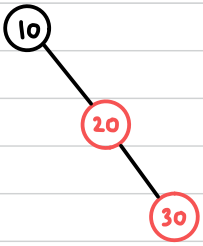
INSERT

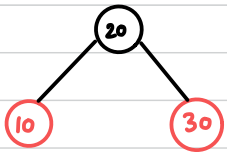
10

20

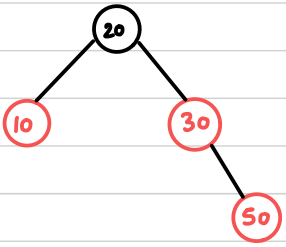


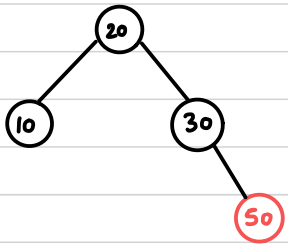
30





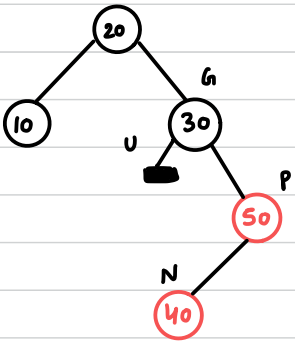
50

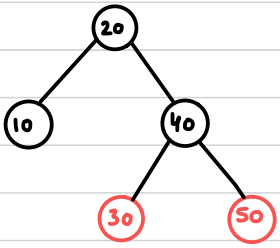




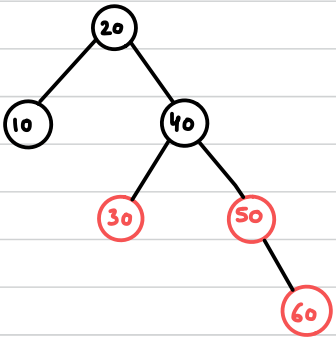
Root must be black

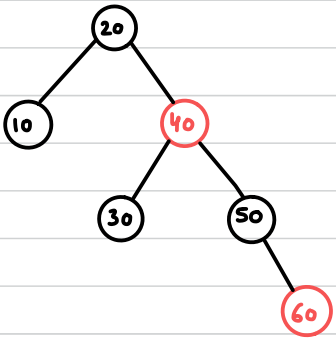
40



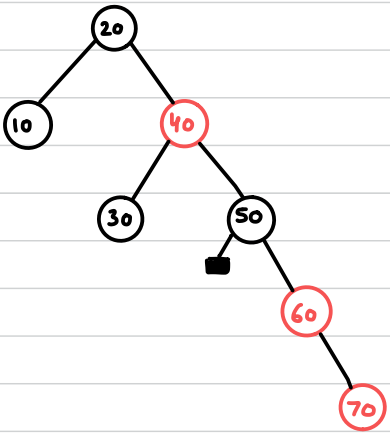


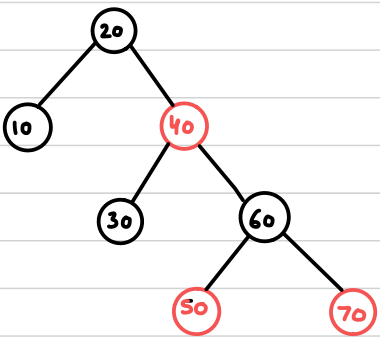
60





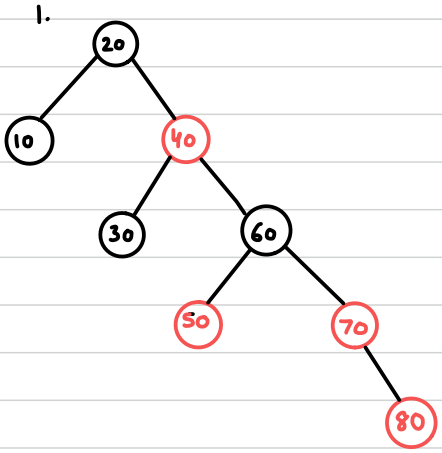
70



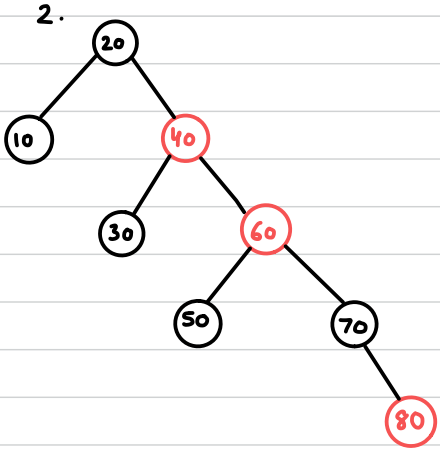


80

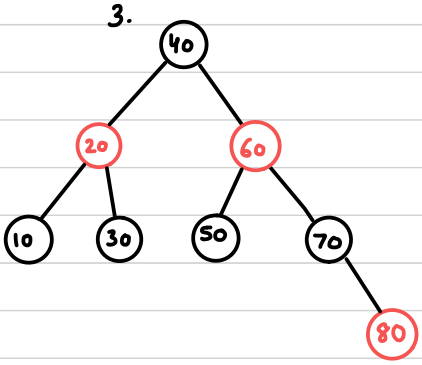
1.



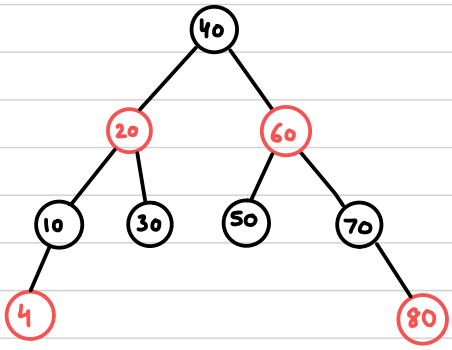
2.



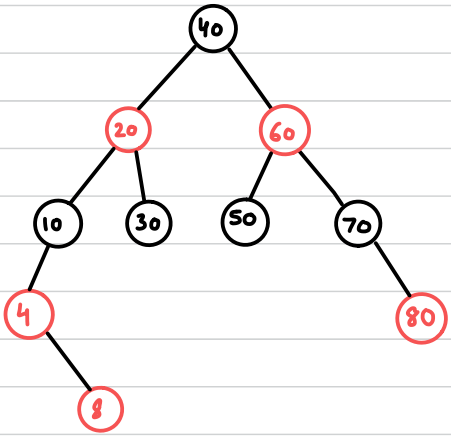
3.

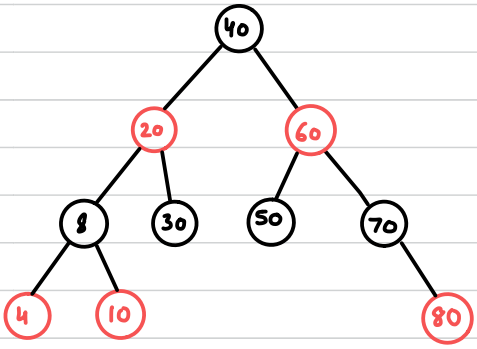


4



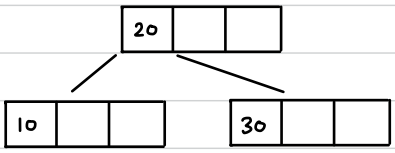
8

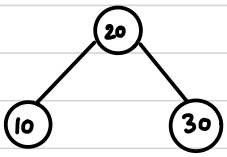


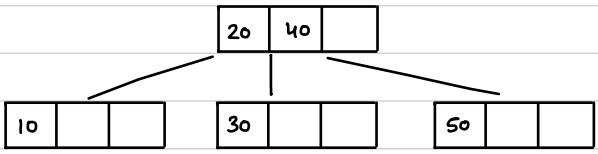


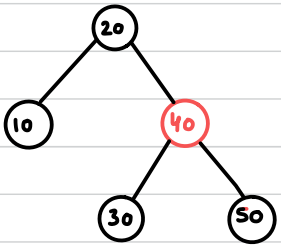
height only red / black = log n
height red and black = 2 log n

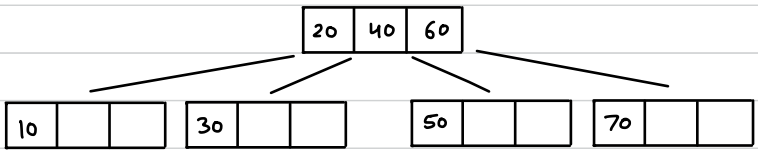
2-3-4 TREES VS RED BLACK TREES

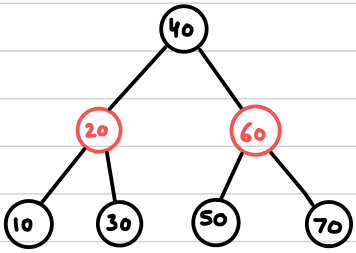










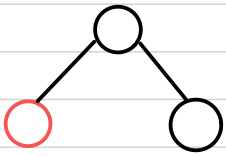


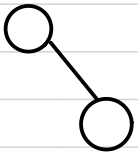
RED BLACK TREE DELETION CASES

CASE 1 : Deleted Node is Red Node

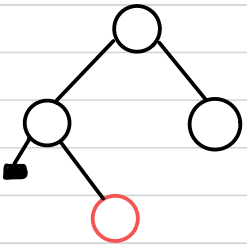
Before Deletion

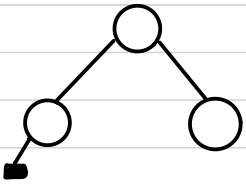
After Deletion

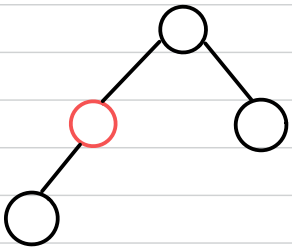


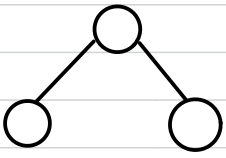


Simply delete as it is a leaf node and red





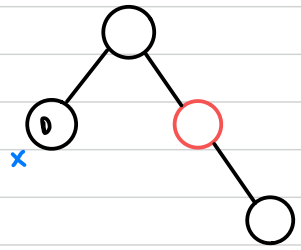




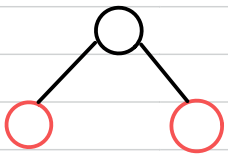
Simply delete because if red node is deleted, the path of black nodes remains unchanged

CASE 2 : Node is black and sibling is red

Before Deletion



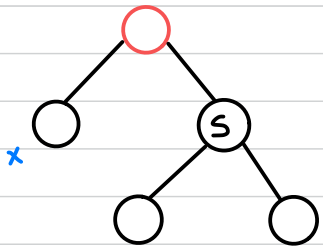
After Deletion



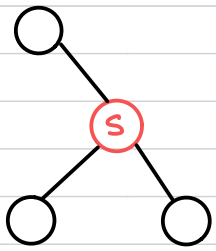
Perform rotation

CASE 3 : Node is black and sibling is also black

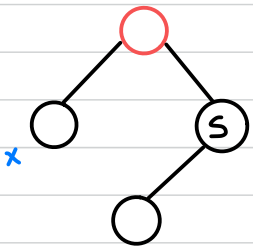
Before Deletion

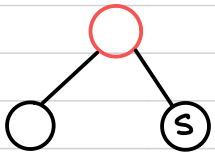


After Deletion



Change sibling to red and parent to black
Recolour





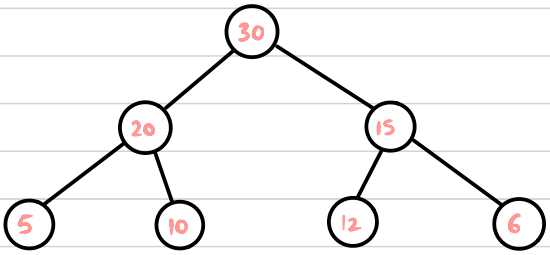
Perform rotation

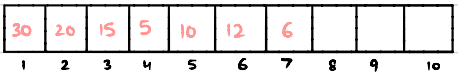
HEAP

↳ It is a complete binary tree

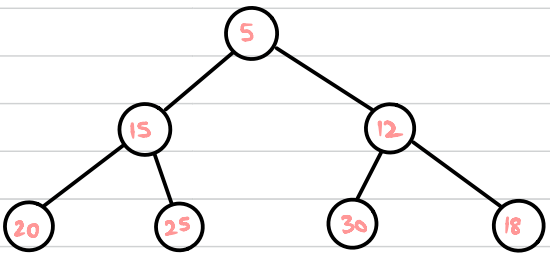
- what is a heap ?
- Insert in a heap
- Deleting from heap
- Heap Sort
- Heapify
- Priority Queues

Max Heap





Min Heap



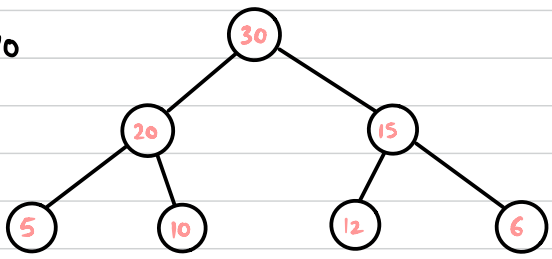
- For complete binary tree, there should not be any free space between the elements of array

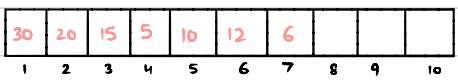
Node at index i
Left child at index 2*i
Right child at index 2*i + 1

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

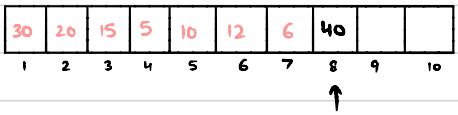
INSERTING IN HEAP

Element to insert = 40



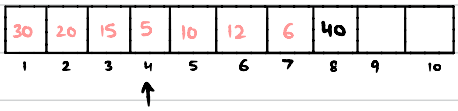


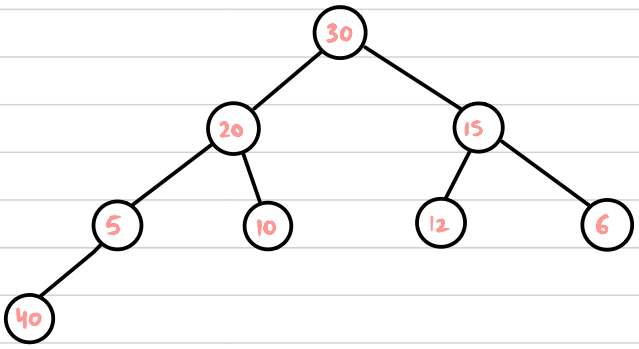
Step 1 : Insert element at next free index in array



Step 2 : Insert the element in the tree

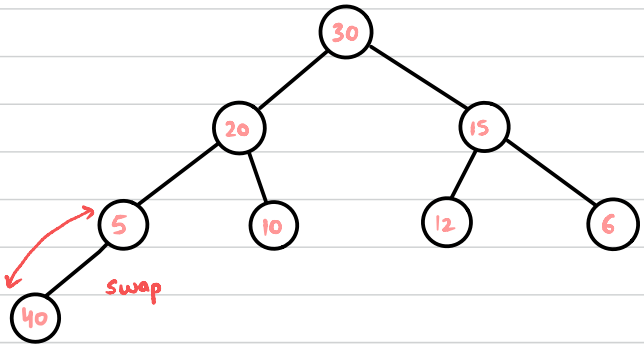
Parent of 40 = $\frac{8}{2}$ = 4

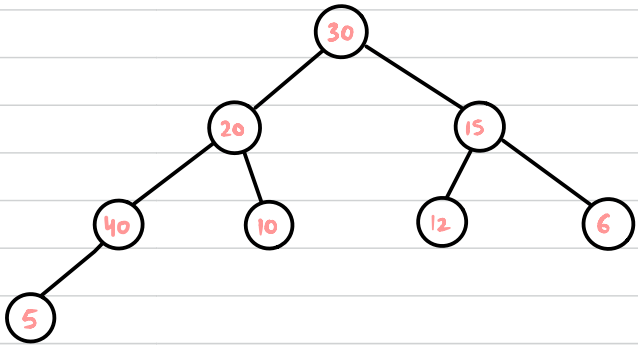




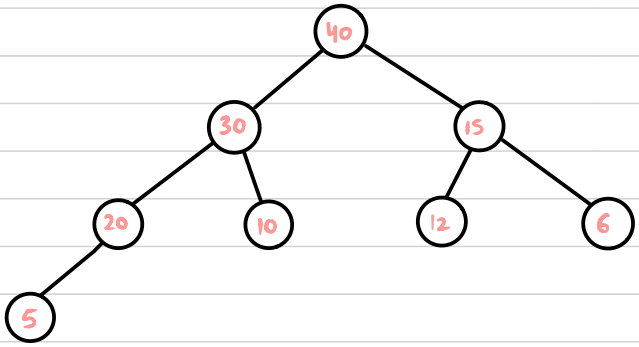
Step 3 : Now tree does not satisfy max heap condition

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

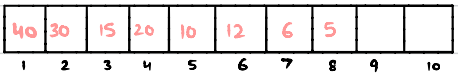




Repeat until condition is satisfied



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



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

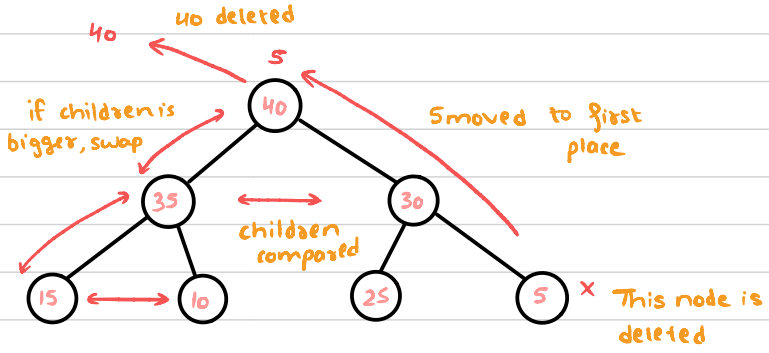
1. Insert in same order O(1) O(logn)
   Delete max priority by searching it O(n) O(logn)] After implementing using heap

Heap is used to implement priority queues

DELETION FROM HEAP

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

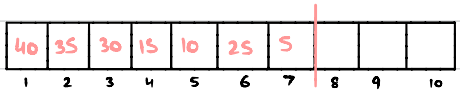
```
void Delete(int A[], int n)
{
    int x, i, j;
    x = A[n];
    A[1] = A[n]      // Smallest or last node is copied at first place
    i = 1; j = 2*i;  // i is at index 1 and j is at its left child

    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;
}
```

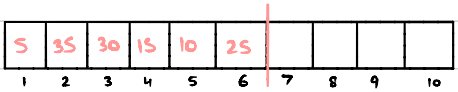


HEAP SORT

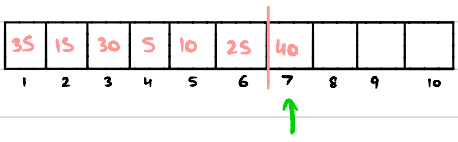
While deleting, Store the deleted element at vacant position

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





x = 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 — $O(n^2)$
3. Selection Sort
4. Heap Sort
5. Merge Sort — $O(n \log n)$
6. Quick Sort
7. Tree Sort

Comparison based sorts

8. Shell Sort — $O(n^{3/2})$

9. Count Sort
10. Bucket / Bin Sort — $O(n)$
11. Radix Sort

Index Based Sort
// Faster but memory consumption is high

There is no best sorting technique.

You have to chose the technique which matches your requirements

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

8	5	5	5	5
5	8	7	7	7
7	7	8	3	3
3	3	3	8	2
2	2	2	2	8

Largest element is sorted in first pass

4 comp
4 swap

2nd Pass

5	5	5	5
7	7	3	3
3	3	7	2
2	2	2	7
8	8	8	8

3 comp
3 swap

→ Not actual numbers of swaps but maximum number of swaps

3rd Pass

5	3	3
3	5	2
2	2	5
7	7	7
8	8	8

2 comp
2 swap

4th Pass

3	2
2	3
5	5
7	7
8	8

1 comp
1 swap

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

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

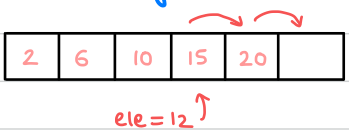
No of swaps : $1+2+3+4$
$1+2+3+4 \ldots\ldots (n-1)$
$\frac{n(n-1)}{2}$ $O(n^2)$

- 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

```
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;
    }
}
```

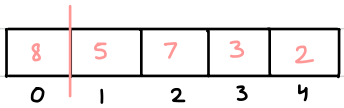
2. INSERTION SORT

Inserting element in a sorted array at a sorted position



// Start comparison from last element

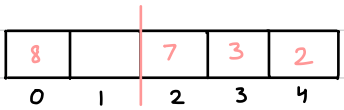
EXAMPLE : A



// Assume only first element to be sorted

n = 5

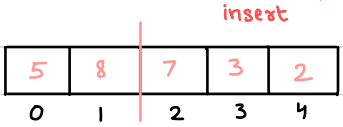
1st Pass



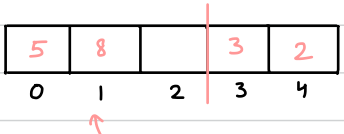
1 comp
1 swap

5

Take out one element and compare and insert

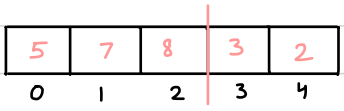


2nd Pass

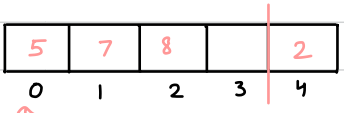


2 comp
2 swap

7

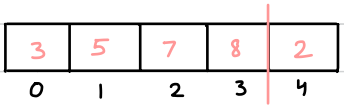


3rd Pass

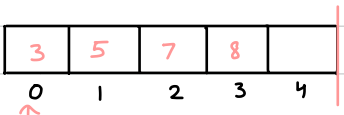


3 comp
3 swap

3

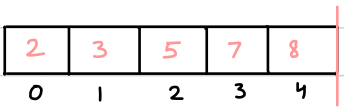


4th Pass



4 comp
4 swap

2



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

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

n = 6

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

0	2
1	3
2	4
3	5
4	6
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

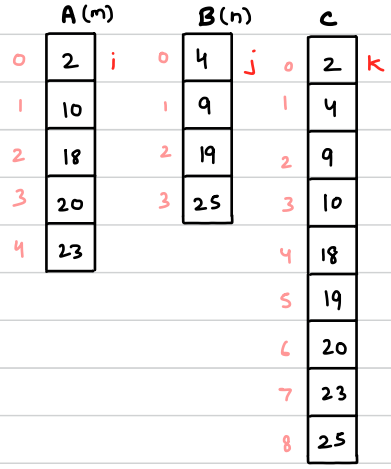
```
void SelectionSort(int A[], int n)
{
    int i, j, k;

    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

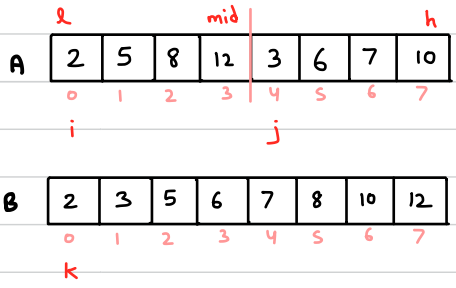
• MERGING TWO LISTS IN THIRD ARRAY

O(m+n)



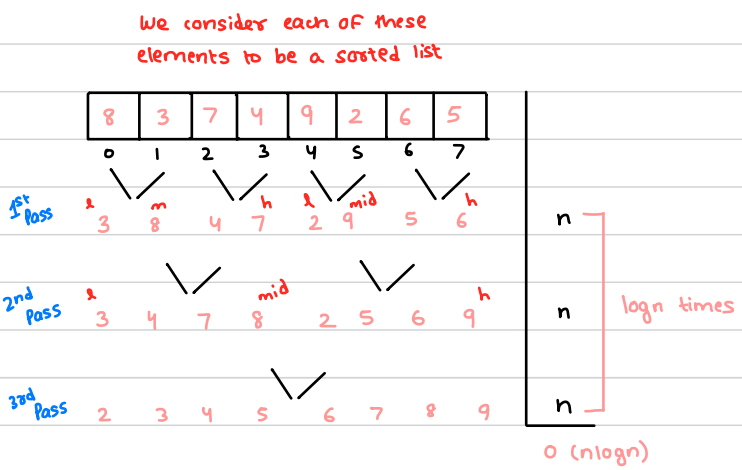

```
Void Merge (int A[], int B[], int m, int n)
{
    int i, j, k;
    i = j = k = 0;

    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




```
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

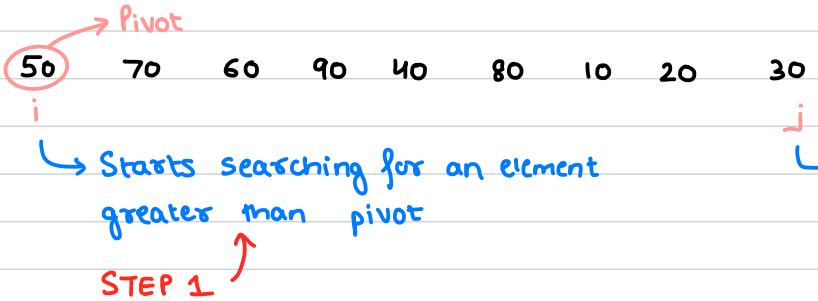


RECURSIVE MERGE SORT O(nlogn)

```
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 numbers of elements are odd
```

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

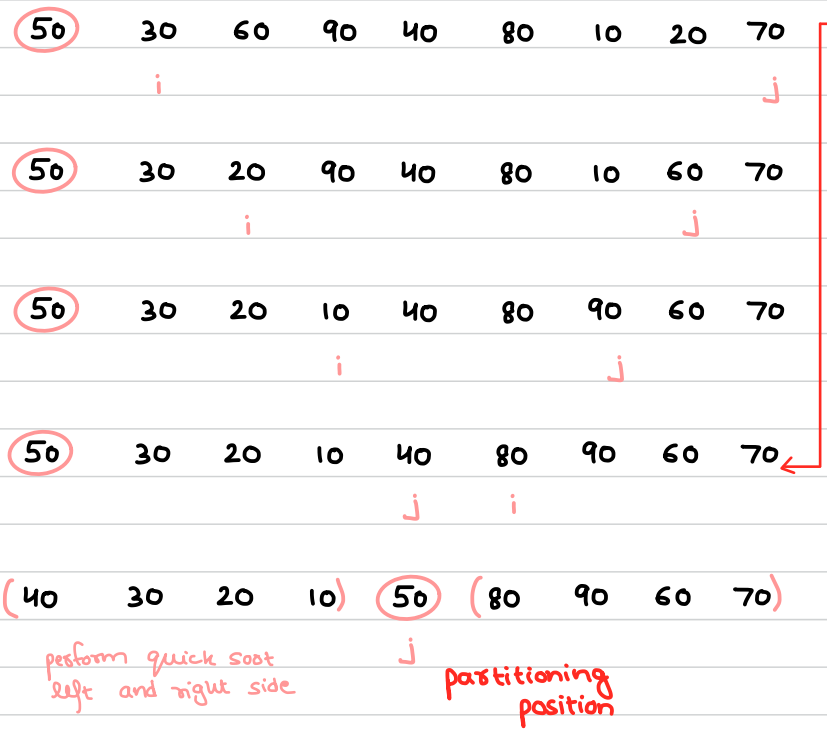


↳ Starts searching for an element greater than pivot — STEP 1

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

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

PARTITIONING PROCEDURE



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

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;
}
```

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
  $O(n^2)$

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

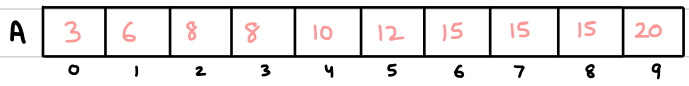
```
void QuickSort(int A[], int l, int h)
{
    int j;
    if (l<h)
    {
        j = Partition(A, l, h);
        QuickSort(A, l, j);
        QuickSort(A, l+1, h);
    }
}
```

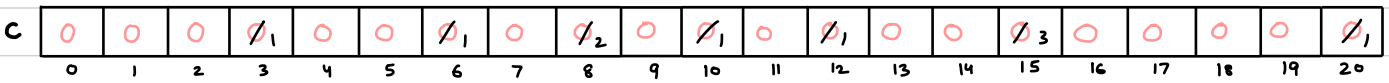
8. SHELL SORT

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

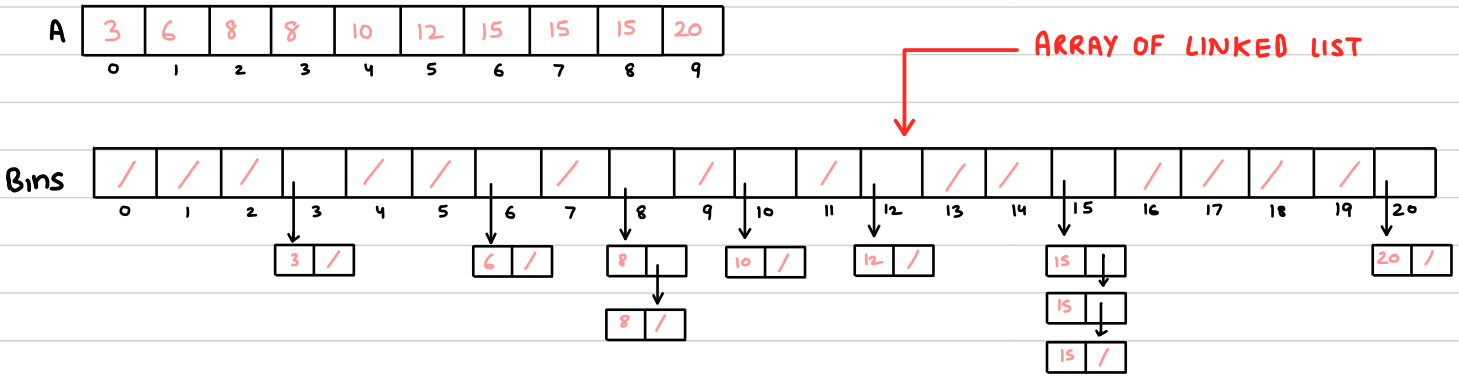





```
Void CountSort(int A[],int n)
{
    int max,i,j;
    int *c;

    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]--j;
        }

        else
            i++;
    }
}                                O(n)
```

Fastest sorting method but high memory consumption

10. BUCKET / BIN SORT

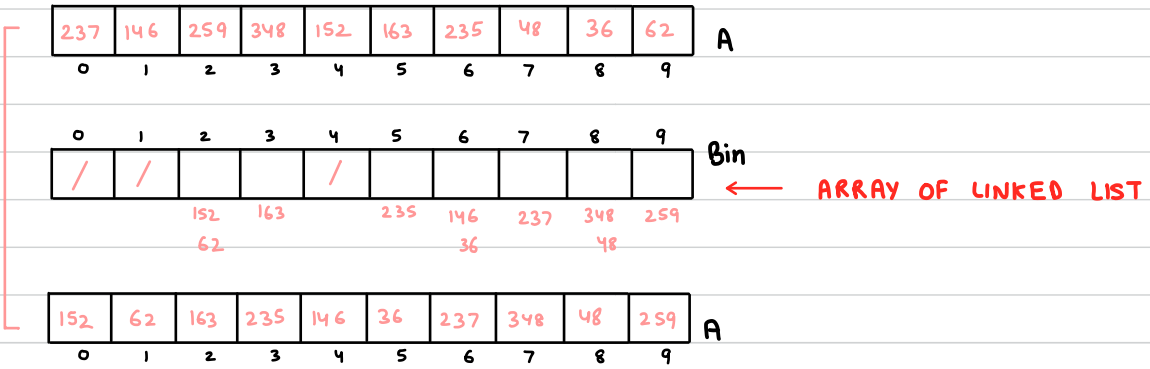



```
void BinSort(int A[], int n)
{
    int max, i, j;
    max = findMax(A,n);
    Bins = new Node*[(max+1)];

    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

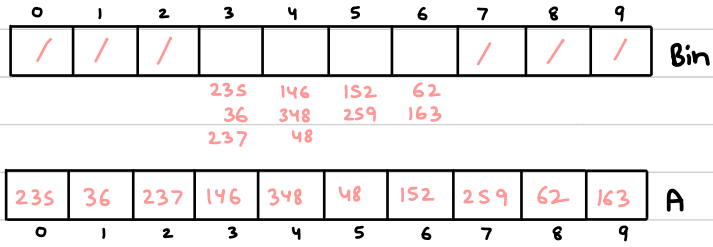
Elements arranged on basis of one's digit



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

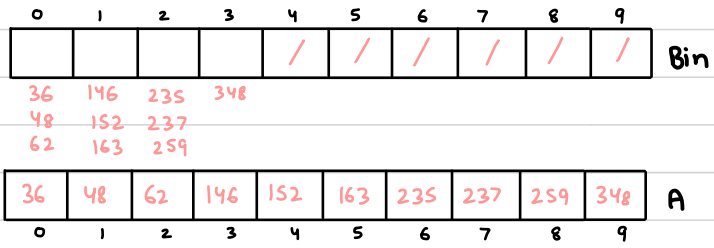
2nd Pass

two's digit



3rd Pass

three's digit



two digit numbers's third digit considered as 0

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

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

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(logn) // 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

A

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

C

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

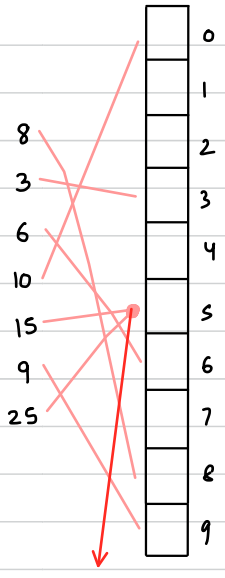
- **Ideal Hashing**

$h(x) = x$ — 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

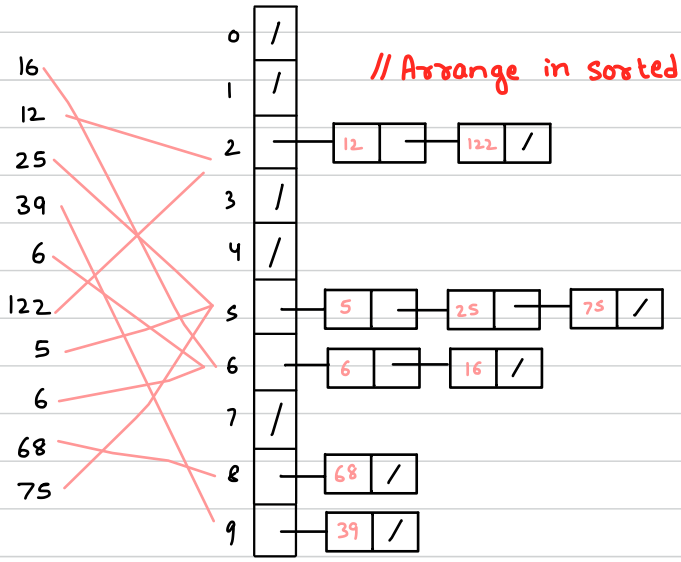
- How to avoid COLLISION ?

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;
}
```

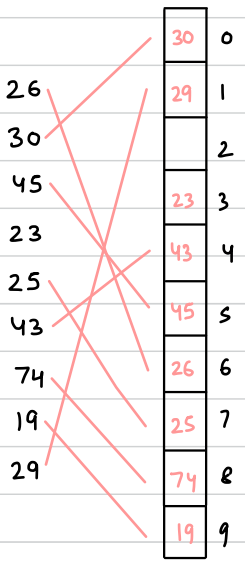
LINEAR PROBING

$h(x) = x \% 10$

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

$i = 0, 1, 2 \ldots$

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

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

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

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

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

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

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

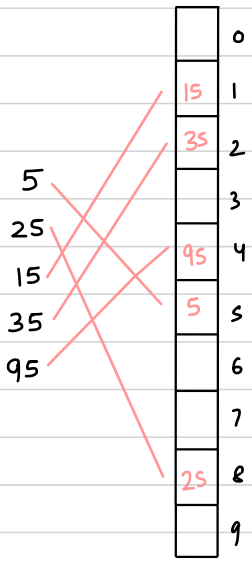
Average Successful Search

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

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 $i = 0, 1, 2 \ldots$

where R is prime num smaller than size of Hash Table

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

$7 - (25 \% 7)$

$7 - 4 = 3$

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

$7 - (15 \% 7)$

$7 - 1 = 6$

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

$7 - (35 \% 7)$

$7 - 0 = 7$

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

$7 - (95 \% 7)$

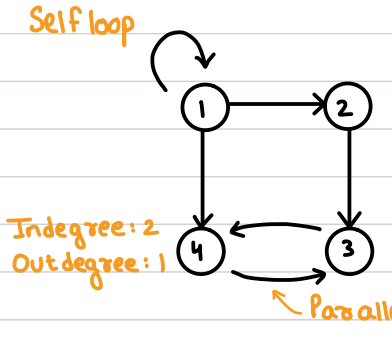
$7 - 4 = 3$

GRAPHS

$G = (V, E)$

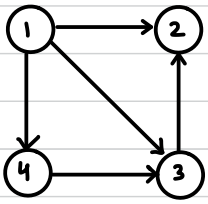
V = Set of vertices

E = Set of edges



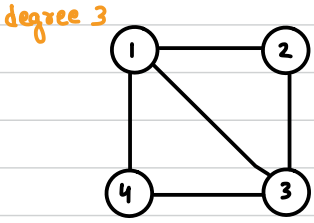
Directed Graph

- Edges are having directions



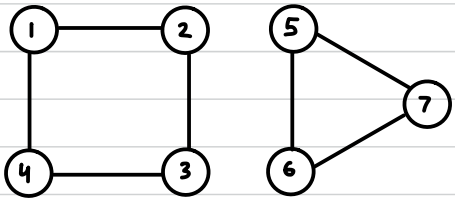
Simple Diagraph

- Without self loop
- Without parallel edges



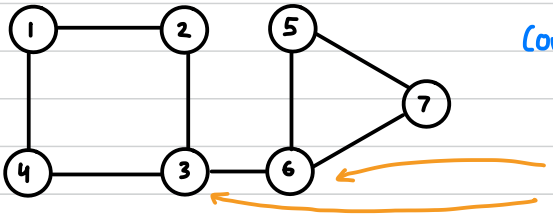
Graph / Non-directed Graph

- Undirected Edges



2 components

Non-connected Graph

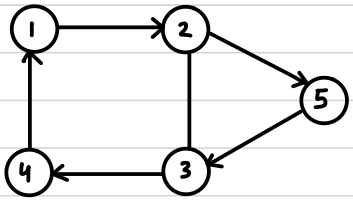


Connected Graph

Connected components

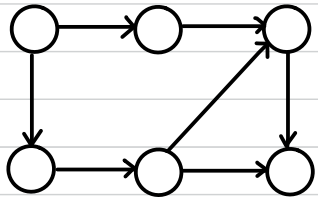
Articulation Points

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



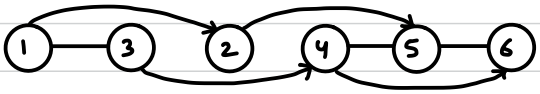
Strongly Connected Graph (Directed Graph)

- 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

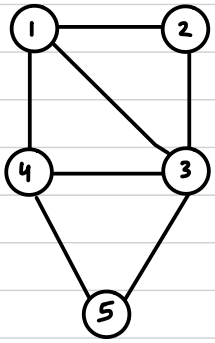


Topological Ordering

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

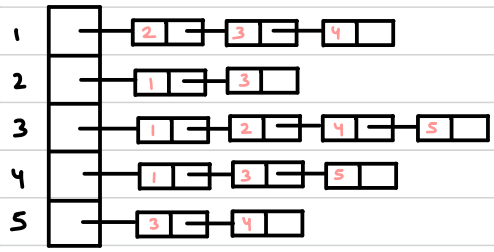
$|V| = n = 5$

$|E| = e = 7$

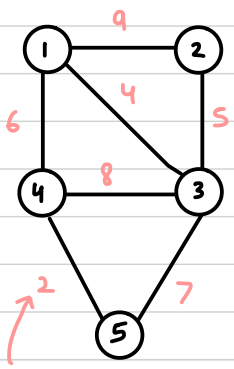
(i, j)

$A[i][j] = 1$

ADJACENCY LIST



COST ADJACENCY MATRIX (Representation of weight)



Weight of edge

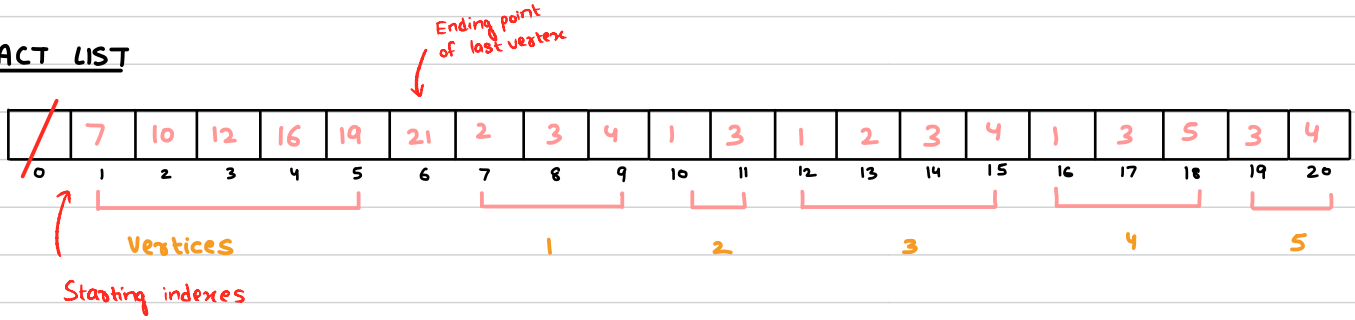
	1	2	3	4	5
1	0	9	4	6	0
2	9	0	5	0	0
3	4	5	0	8	7
4	6	0	8	0	2
5	0	0	7	2	0

5 x 5

COST ADJACENCY LIST

Add another area to node for weight of edge

COMPACT LIST



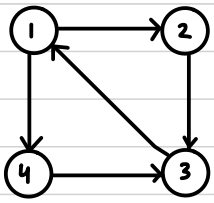
$|V| + 2|E| + 1$

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

$20 + 1 = 21$

REPRESENTATION OF DIRECTED GRAPH

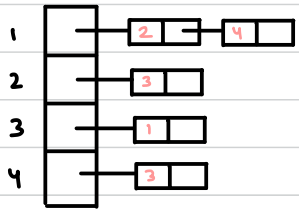
ADJACENCY MATRIX



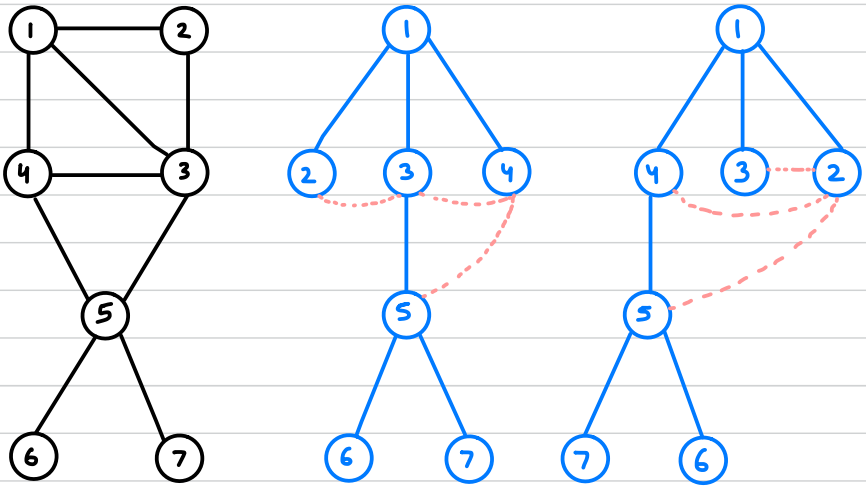
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	0
4	0	0	1	0

4 x 4

ADJACENCY LIST



BREADTH FIRST SEARCH (BFS)



TREE 1

TREE 2

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

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

    while(!isEmpty(q))
    {
        u = dequeue(q);
        for(v = 1; v <= n; v++)
        {
            if(A[u][v] == 1 && visited[v] == 0)
            {
                printf("%d", v);
                visited[v] = 1;
                enqueue(q, v);
            }
        }
    }
}
```

← Starting vertex

← Take out vertex from queue

For scanning row of matrix →

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

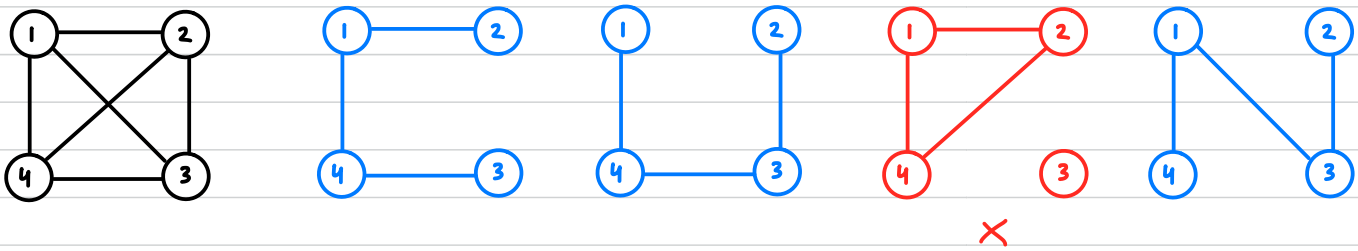
0	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7

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.



×

$|V| = 4$

$|E| = 6$

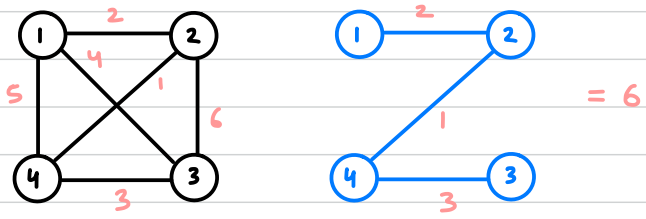
$^{|E|}C_{|V|-1} = {}^{6}C_{3}$ ways or number of spanning trees but not include trees forming cycles

$^{|E|}C_{|V|-1}$ - 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.

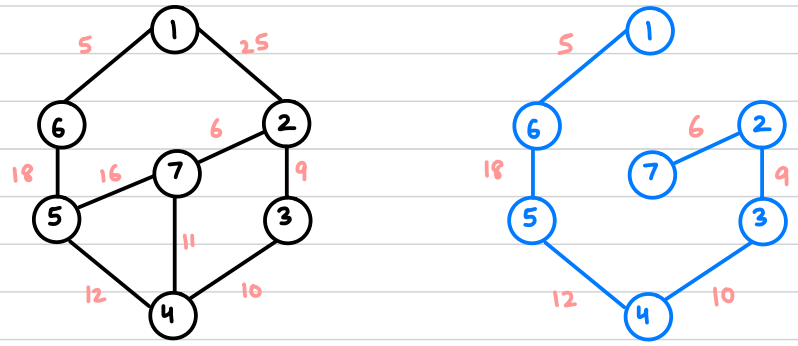


PRIM'S MINIMUM COST SPANNING TREE

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

$ne = n \times n$

$O(n^2)$

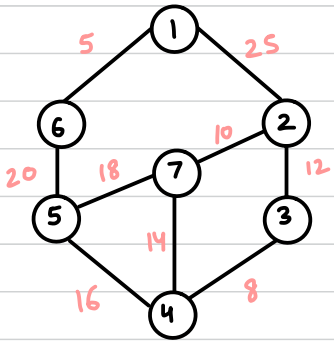


STEP 1 : Select the edge with least weight along with its vertices

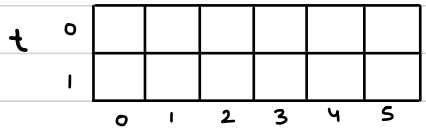
STEP 2 : Then compare the edges joined with the vertices and select the one with least weight along with its vertex

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	-	-

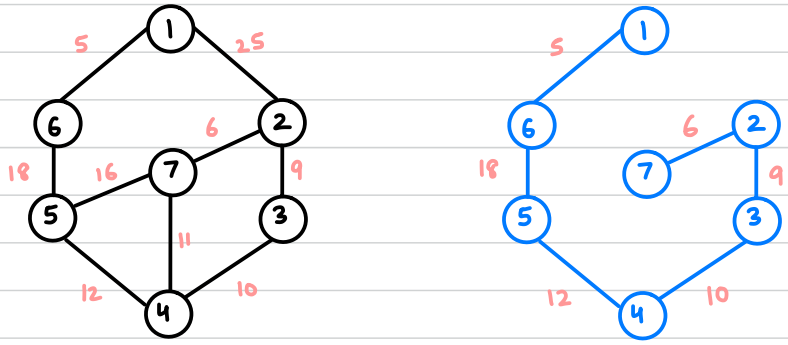


// Only either of lower or upper triangular matrix will be sufficient

```
#define  I  32767                    // Largest possible integer value

int cost[8][8], near[8], t[2][6];    // Initialize as {I}

void main()                          // Initialize as the given matrix (Replace '-' with 'I')
{                                                                          INITIAL PROCEDURE
    int i,j,k,u,v,n=7,min=I;

    for(i=1;i<=n;i++)
    {
        for(j=i;j<=n;j++)            // For accessing upper triangular matrix only
        {
            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(i=1;i<n-1;i++)                                                     RECURSIVE PROCEDURE
    {
        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

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

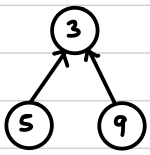
Consider its two subsets

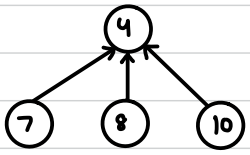
$A = \{3, 5, 9\}$

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

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

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



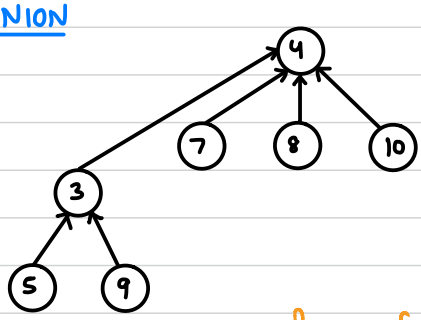


No of nodes in negative (4 is having 4 nodes)

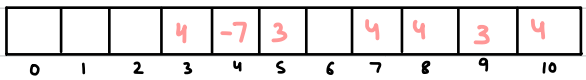
Rest are representing parent

Representation of set

(1) UNION



The parent with more number of nodes



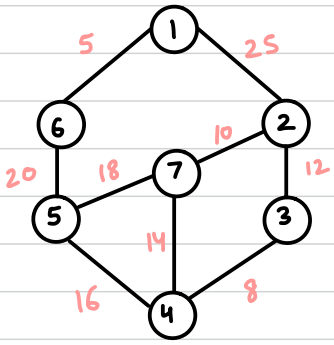
Parent of nodes

```
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

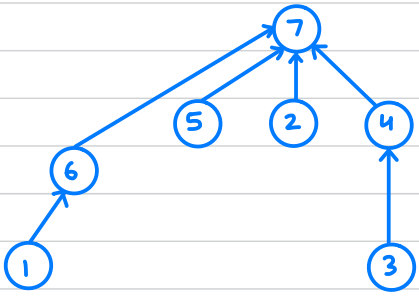
```
int Find (int u)
{
    int x = u;

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

```
#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) )        To check if cycle is
        {                                forming or not
                t[0][i] = u;
                t[1][i] = v;
                union( find(u), find(v) );
                i++;
        }

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

ASYMPTOTIC NOTATION

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

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