Programs Book

1 # Basic Helloworld

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  cout << "Hello world";

  return 0;
}
```

2 # Taking input / variable / output

- Sum of two numbers

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int a, b, c;

  cout << "enter two numbers to
          add";

  cin >> a >> b;

  c = a+b;

  cout << c;

  return 0;
}
```

3#

Using If else write CPP school grading system

below 25 - F
25 - 44 - E
45 - 49 - D
50 - 59 - C
60 - 79 - B
80 - 100 - A

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int x;
    cout << "enter your marks!";
    cin >> x;
    if (x<25)
    {
        cout << "F";
    }
    if (x>=25 && x<=44)
    {
        cout << "E";
    }
    if (x>=45 && x<=49)
    {
        cout << "D";
    }
```

here the if statements are executed one by one all ifs are checked for the ranges which consumes more time

like ex: in 1st if x is checked if its less than 25 but in 2nd if x is checked again saying it should be >=25 so if condition takes more time

all blocks will be executed even after printing the 1st	all ifs treated as seperate new block

```
int main ()
{
  int x;
  cout << "enter marks";
  cin >> x;

if (x<25)
 {
   cout << "F";
 }
else if (x<=44)
 {
   cout << "E";
 }
```

So if we use else if ladder it consumes less time

ex: 1st if is executed if its true then it stops execution doesn't go to else if here unlike seperate if's all are connected

so after 1st if is executed checks if its true it stop execution or else it goes to else if having first checked condition in memory

4# Switch statement program
take the day no and print the day

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int x;
  cout << "enter a number from 1-7:";
  cin >> x;
```

```
Switch (x)
{
  case 1: cout << "monday";
          break;
  case 2: cout << "Tuesday";
          break;
          .
          .
  case 7: cout << "Sunday";
          break;

  default: cout << "invalid no";
           break;

return 0;
}
```

#5 For loops

5.1> Print 1-10 numbers

```
#include <bits/stdc++.h>
using namespace std;

int main();
{
  for (int i = 1; i <= 10; i++)
  {
    cout << i << endl;
  }
  return 0;
}
```

O/P: 1
2
3
4
5
6
7
8
9
10

5.2> Print tables of any number from 1-10

```
int main();
{
  for (int i = 0; i <= 70; i = i+7)
  {
    cout << i << endl;
  }
  return 0;
}
```

O/P: 0
7
14
21
28
35
42
49
56
63
70

5.3) Print a name with the value of i

```
int main();
{
  for (int i=1 ; i<=5 ; i=i+1)
    {
      cout << "Aditya" << i;
    }
  return 0;
}
```

#6 Functions

6.1) write a function to print name

```
#include<bits/stdc++.h>
using namespace std;

void printname();
{
  cout << "Hi aditya";
}
int main()
{
  printname();
  return 0;
}
```

6.2> take input pass it to function parameter and print name

```
void printname (string name)
{
  cout << " Hey " << name;
}
int main ()
{
  string name1;
  cin >> name1;
  print name (name1);

  string name2;
  cin >> name2;
  printname (name2);

  return 0;
}
```

o/p:

Hey name1

Hey name2

6.3> Sum of two numbers (return function)

```
int sum (int num1, int num2)
{
  int num3 = num1 + num2
  return num3;
}
int main ()
{
  int a, b, c;
  cin >> a >> b;
  c = sum(a, b)
```

```
    cout << c;
}
    return 0;
}
```

or using void function.

```
void sum(int num1, int num2)
{
    cout << num1 + num2;
}
int main
{
  int a, b;
  sum(a, b)
}
```

C++ Basics

```
#include <bits/stdc++.h>   → Includes libraries
using namespace std;       → includes header files
int main()                 → Main function
{
    cout << "Hello world";  → Print line
    return 0;
}
```

Variables

- Declared in main function
- an empty space is created for the variable which can be input by the user or can be changed in the code ahead.

ex.
```
int main()
{
    int x, y;          → variables

    cin >> x >> y;
    cout << x+y;

    return 0;
}
```

o/p: 7 7
= 14

Datatypes

* int (integer) - 4 bytes ⇒ range $(-10^9 - 10^9)$ → -2,147,483,648 to 2,147,483,647.
- long or long long (integer) $(-10^{18} - 10^{18})$ } Usually used datatype to store integer type

* Float - 4 byte
- Double - 8 byte
- long Double - 10 byte } used to store decimal variables.

* String

```
ex: int main()
    {
    string str;
    cin >> str;
    cout << str;
    }
```

I/P: Hi I am adity
O/P: Hi

String reads until it finds a space you'll have to define multiple variables for each word.

declaring string variable value should be in double quotes

* so to pick up the whole line we use
string str;
getline(cin, str);
cout << str;
getline(cin, str) → Picks the whole line

* char (character type)

- usually used to store single character
- declaration, enclosed in single quotes

```
Ex: int main()
    {
      char c = 'A';
      cout << c;
    }
```

int, long long, float, Double string, getline char	are almost the only needed datatypes for DSA

* Relational Operators
or Conditional statements

- == - checks if 'A' value = B value
- != - not equal to
- <= - less than or equal to
- >= - greater than or equal to
- && - all statements should be true
- || - any one should be true

to check multiple statements

* char (Character type)

- usually used to store single character
- declaration enclosed in single quotes

```
int main()
{
    char c = 'A';
    cout << c;
}
```

int, long long, float, Double string, getline char	almost the only needed datatypes for DSA

* Relational Operators
or Conditional statements

- == - checks if 'A' value = B value
- != - not equal to
- <= - less than or equal to
- >= - greater than or equal to
- && - all statements should be true
- || - any one should be true

to check multiple statements

classmate
Date
Page

Ex sample program
take age input & print adult or not

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
  int a;
  cout << "enter age:";
  cin >> a;
  if (a >= 18)
  {
    cout << "adult";
  }
  else
  {
    cout << "not an adult";
  }

return 0;

}
```

* Decision Making Statements

- **If statement**

Syntax:
```
if (condition)
{
  // statements
}
```

if condition is true then excutes the next line

- **If else**

Syntax:
```
if (condition)
{
  //
  //
}
else
{
  //
}
```

- **Nested if**

Syntax:
```
if (condition)
{
  //
  if (condition)
  {
    //
  }
}
```

If else if ladder

Syntax:
```
if (condition)
{
  //
}else
{

else if (condition)
{
  //
}
else
{
  //
}
```

Switch statements

Syntax:
```
switch (expression)
{
case 'const-exp':
    statements
    break;
  //
default:
    statements;
}
```

single quotes for character

none for integer

* Break

- Used to terminate a loop when loop comes across a break the command or flow breaks out & moves out of the loop or to the next command.

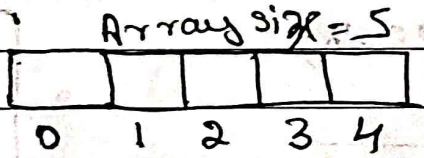
Syntax: Break;

* Arrays Basics

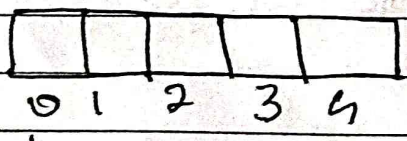
- Arrays mainly used to stored data multiple items of same data type.

Syntax: datatype arrayname [arrsize];

Ex: int arr[5]



- The entire array is stored in a memory location

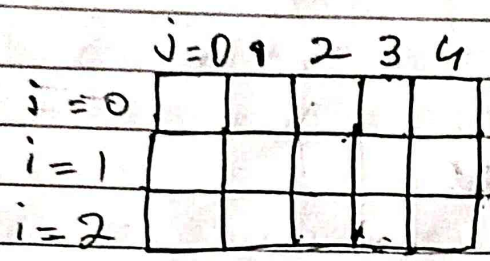


↓
randomized address location (unknown)

but first index address will be known
eg. it is right after 0th index address.

* 2-D Array

- int arr[3][5] 3 x 5
  - i → rows
  - j → columns.



* Strings

Syntax: data type name = " ";
String

ex. String S = "Aditya";
0 1 2 3 4 5

↓

Stored like in array
each character is stored
seperately.

cout << S[2];

O/P

ex: string s = "Adityaz";
int leng = s.size();
s[leng-1] = 'z'; ↳ Count starts from 1 to 6

cout << s[leng-1]; string indexing starts from 0 so 6-0 = 5 replace 5th letter
cout << s;

o/p: z
Adityaz

★ For Loops

Syntax:

for(initialize exp; check condition; update)
{
Statements
}

ex: for(int i=0; i<=10; i++)
{
cout << i;
}

- Conditions are checked first if true then the statements are executed

* **While loops** • same method as for loop

Syntax:
```
initialize expression
while (cond)
{
  // statements
  update_expression.
}
```

Ex:
```
int i=0
while (i<=10)
{
  cout<<i<<endl;
  i=i+1
}
```

* **do while loop**

! body of the loop or the statements are executed atleast once before the condition is checked

Syntax:
```
initialize expression
do{
  Statement
  increment/updak
}
while (check cond)
```

ex: int i=2;

```
do{
    cout << i;
    i= i+1;
}
while (i<=1)
```

* Functions

- Functions are a set of code that performs a specific operation in its own block
- increases readability
- reuse the same code multiple times.

~~casually used function~~

Types of functions

- Void function

- a void function doesnt return anything

~~Syntax:~~

ex: to print name using a void function

```
void Printname()
{
    cout<<" Hi Aditya";
}
```

```
int main()
{
    printname();
}
```

o/p: Hi Aditya

- **Parameterised function**

ex:

```
void printname(string name)
{
    cout << "Hey" << name;
}
int main()
{
    string name1;
    cin >> name1;
    printname(name1);
    string name2;
    cin >> name2;
    printname(name2);
```

- Difference between return function and void function is that the

void function does the operation in its block and outputs there itself returns nothing to main function

return function does the operation in its own block and returns the value to the main function

Pass by value

```
void inc(int n)
{
  n = n+5;
  cout << n;
}

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

o/P:

15 ← inc function value

10 → main function value

- The value printed in the inc function is 15 cuz it takes a copy of 10 from main function and does changes to the copy but not the original value in the main function.

* **Pass by Refrence**

```
Or: void inc (int &n)
    {
      n = n + 5;
      cout << n;
    }

int main()
{
    int a = 10;
    inc(a);
    cout << a;

}
```

o/p: 15 → inc function value
15 → changed main function value.

- The value printed in inc function is 15 and the value printed in main function is also 15 because here in Pass by reference the original value is passed to the inc function by using (&) in the function parameters which gets the address of the original value in main function so the inc function has control over the original variable in main function.

write about array functions

- Whenever you're passing arrays to function its always Pass by refrence.

* Things to remember about arrays

- While taking input for an array few variables are required.
like
  - int n;
  - int a[n] → here to allow the user to specify the size of the array

- Then a for loop is used to take in input sequentially
  for(int i=0; i<n; i++)
  - here i is used to represent index position number of the array starting from 0.
  so index [i=0] i is initialized to 0
  - [i<n] - here i<n says how much input the for loop should take for each index position in the array
  and it should be i<n or i<=n-1

ex: n = 5 which is 0 1 2 3 4 size = 5

- if i is given as $i < n$
  $i < 5$
  the loop takes from 0 to 4 - 5 digits.

- if it's given as $i <= n-1$ then it would be since n = 5 it would be 6 digits - 0, 1, 2, 3, 4, 5
  So n-1 would be 5 digits 0, 1, 2, 3, 4, 5.

- [cin >> a[i]] - represents taking input taking input
  - a[i] ⇒ a[0] - represents the arrays index position where the input has to be stored

ex: User inputs firs number as 7
7 is stored in a[0] - the first element of the array.

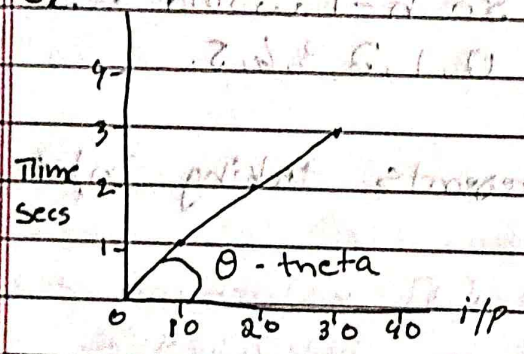
- a[i/j] ⇒ represents that positions value where as i or j seperately represents index only.

Time & space complexity

- Time complexity -
it cant be called as time taken by the code to run because it is dependent on the machines hardware.
TC != time taken

- rate at which time taken increases depending on input size.

Ex:



: how much time increases wrt input size.

and since time complexity is not represented in seconds it is represented with Big-oh-notation: O( )

Time taken

Ex:
```
for(i=1; i<=N; i++)
{
    cout << "Hey";
}
```

This piece of code would be like O(N x 3)
N x 3 because
- it runs upto N iterations
- and in each iteration theres 3 steps happening that is ① condition checking
② priniting
③ incrementing.

* Three rules to follow in computing TC
1) always take the worst case scenario.
2) avoid constants
3) avoid lower values

① we have to always compute TC in worst case scenario because

ex: if (marks<25) cout << "D"
else if (marks<45) cout << "C";
else if (marks<65) cout << "B";
else cout << "A";

here if entered theres 3 cases
Best, average, worst
- if user inputs <25 it just executes 2 steps which takes less time so that is the Best case

- if i/p is above 65 its gonna take the most time so that is the worst case
and we always take the worst case since i/p can be anything not always for the best case.

(2) Avoid constants

when inputs sizes are large the constants are very less significant

ex: $O(2 \times 10^{15} + 3 \times 10^{10} + 8)$

↓

this will have very less significance

(3) Avoid lower values

- other notations than Big-oh (O) theres theta θ and omega (Ω)

Big oh (O)	omega (Ω)	theta θ
↓	↓	↓
worst case	Best case	average case
↓	↓	↓
upper bound complexity	lower bound	

& Time complexity problems examples

Q1)

```
for(int i=0; i<N; i++)
{
    for(int i=0; i<N; i++)
    {
        |__|
    }
}
```

classmate
Date
Page

$i=0$ {$j=0, 1, 2, 3 \ldots N$}

$i=1$ { $j=0, 1, 2, 3 \ldots N$}

$\vdots$

$i=n-1$ { $j=0 \ldots N$}

basically N iterations twice
i= N iterations
j= N iterations.
so $\boxed{O(N^2)}$

Q2)
```
for(i=0; i<N; i++)
{
    for(j=0; j<i; j++)
    {
        ..... -
    }
}
```

$i=0$ {$j=0$}
$i=1$ {$i=0, 1$}
$\vdots$
$i=n-1$ {$j=0, 1, 2 \ldots n-1$}

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

$$\frac{N \times (N+1)}{2} = \boxed{\frac{N^2}{2} + \frac{N}{2}}$$

$$= O\left(\frac{N^2}{2}\right) \approx O(N^2)$$

* Space complexities

- It is the memory space that a program takes again represented Big-oh

Auxillory space + input space

↓ space taken to solve the problem

↓ space taken to store the input.

```
Ex: Sum
int a, b;
int c;
```

a, b takes input which is input space
c ~~used~~ memory used for solving - auxillory space

So O(3)

- Always use an extra variable and do not tamper with input.

* Pattern Problems

things to remember

- always usually all pattern problems have 2 loops ~~for one~~
  the outer loop - for rows
  the inner loop - for columns

- for the outer loop - count the number of ~~rows~~ rows
- for the inner loop - count the number of columns and connect it with the rows
- always print inside the inner loop.
- observe symmetry

* C++ STL - Standard library

ex: or: include<math.h> math library
to include
math functions

include<string.h> - String library

- instead of including all libraries individually we use include <bits/stdc++.h>

- using namespace std. if we dont use this we'll have to use std::cin or cout everytime.

- STL is mainly divided into 4 parts
- Algorithms
- containers
- functions
- iterators

Pairs

- It is a part of utility library

- used to store any type of data in a pair

ex:
```
void pair1()
{
    pair<int, int> P = {1, 3};
    cout << P.first << " " << p.second;
}
```
(can be any data type)

o/p: 1 3

- Nested Pair - used to store multiple varible nested can be any number of time

ex:
```
Pair<int Pair<int,int>> P = {1, {2, 3}};
cout << P.first << "P.second.first<<"<< p.second.second
```

o/p 1 2 3

- Pairs can also be used as a datatype in arrays

like int arr[] = { }

or char arr[] = { }

we can also use

Pair<int, int> arr[] = { }

index 0 1 2

ex: Pair<int, int> arr[] = {{1,2}, {3,6}, {7,8}}

cout << arr[0].first;

o/p: 1

Containers

1) Vectors

- Vector is a container which is dynamic in nature

ex: when you declare an array size it cant be altered again further but in vectors you can alter the size as per your need.

- A container vector is a container that stores elements similar to array

can be any datatype

* declaration: vector <int> V;
  - {} → creates an empty container
  - V → name

used to insert elements into vector

- V.push_back(1);
  - ↳ push back pushes an element into the empty vector container
  - ↳ {1}
- V.emplace_back(2);
  - ↳ pushes element 2 into container
  - ↳ {1, 2}

- Pairs can be used as datatype as well in vectors

ex: vector <pair <int> <int>> vec;

~~vector<pair~~ V.push_back({1, 2}); or V.emplace_back(1, 2); → not this

- to declare a ~~variable~~ container vector of size 5 and those 5 instances are 100 it'll be written as

vector <int> V(5, 100);

↓

{100, 100, 100, 100, 100}

- you can also just mention the size

  vector<int> v(5);

  (dependson compiler)

  size 5 but assings random 5 instances which can be changed later

- you can also declare one vector and use that in another vector by using the original vectors copy.

  ex: vector<int> v1(5, 20);

  vector<int> v2(v1);

- To increase the size of the vector you can use push_back emplace_back at any point in the code (dynamic nature)

  ex: vector<int> v(5, 20);

  v.push_back(1);

  this increases the size of vector v by 1 5 to 6

- accessing elements in a vector. is similar to array

  ex: v - { 1, 8, 9, 11, 14 }
  0 1 2 3 4

  v[3] = 11 or v.at(2) = 9

* Iterating in a vector

- another way of accessing the vector is iterators

here it is declared as

iterator name

```
vector<int>:: iterator it = v.begin();
cout << *(it)
it++;
```

ex: {10, 20, 30, 40}

here the iterator 'it' is set to begin so what it does is it points to the memory location or address of the first element

and in cout a star is used to get the element of that address.

here begin so iterator pointing at 10s location
address *(it) → prints 10
it ++ → iterator moves by one

- vector<int> :: iterator it = v.end();

this points to location of the space after the last element

{1, 2, 3}
↑
v.end

So it -- from v.end would point at 3

- V.rend()
  points to the opposite end

  ↑ {1, 2, 3}

- V.rbegin()
  points to the opposite begin

  {1, 2, 3}
  ↑

- cout << V.back() << " ";

  points to the last element
  or gets the last element

  {1, 2, 3}
  ↑

  not vector type
  doesn't point
  it's an int type
  directly gets the last
  element.

* Ways to print a vector

1) cpx: {10, 20, 30} = V

loop from beginning to end

```
for(vector<int>::iterator it = V.begin(); it != V.end(); it++)
{
    cout << *(it);
```

2)
```
for (auto it = v.begin(); it != v.end(); it++)
{
cout<<*(it)<<" ";
```

* here when a data type is defined already then the key word auto defines the datatype automatically according to the previously assigned datatype for the data.

↓

like int a = 5;

infor loop for auto a = 5;

3)
```
for (auto it:v)
{
cout<< it <<" ";
```

→ Known as for each loop

* Deletion in Vector

- An Erase function is used to delete something in a vector

eg: v.erase(v.begin() + 1);

this deletes 20 so

{10, 30, 40}

classmate
Date
Page

- and if you wanna delete multiple consecutive elements from a vector the

ex: V. erase (v.begin()+1, v.begin()+3)

here for the vector {10, 20, 30, 40}

it erases 20 & 30

- but the end point should be pointing after the element we wanna delete.

begin	+1	+2	+3
10	(20	30)	40

begin+1 to +3
removes 20 & 30.

* **Insert function**

- used to insert element at a desired position

ex: in a vector vector<int> v(2,100) // {100,100}

```
v.insert(v.begin(),300);
```

inserts single element at begin position
{300, 100, 100}

to insert in another location just

```
V.insert(v.begin()+1,300);
```

- To insert multiple elements

Position to insert the element

ex: `V.insert(V.begin()+1, 2, 10);`

ex: vector<int> V{2, 10};

V.insert(V.begin()+2, 3, 40);

{10, 10, 40, 40, 40}

* Erase and Insert is important in vectors just know about copy.

* Copy function

ex: vector<int> V = {10, 20, 30, 40};
vector<int> copy(2, 50);

vector

`V.insert(V.begin(), copy.begin(), copy.end());`

- V.insert - to insert elements into Vector 'V'
- V.begin - The position where the elements are to be inserted.
- copy.begin - elements to add starting elements from the copy vector to be copied to V

- copy_end - end position of the elements from copy vector.

So from vector copy your copy begin to end elements into vector V.

* V.size() - Gives the size of a vector

ex: v = {10,20}
v.size() = 2

* V.pop_back - removes the last element

ex: v = {10,20}
v.pop_back();
{10}.

* V1.swap(V2); - swaps two vectors

ex: V1 = {10,20}
V2 = {30,40}

V1.swap(V2);
V2 = {10,20}
V1 = {30,40}

* V.clear(); - clears the whole vector

* V.empty(); - Tells if a vector is empty or not

Lists

2) Lists

- ◊ Declaration

list<int> ls;

- ls.pushback(2);

pushes back element 2 at the last position
↳ {2}

- ls.emplaceback(4);

{2, 4}

- ls.push_front(5);

{5, 2, 4}

- ls.Emplace_front();

note: all other functions such as begin, rbegin, rend, clear, insert, swap end & end are the same as vectors.

3) Deque

- all functions same way

deque <int> dq;

- dq.push_back(1);

{1}

- dq.emplace_back(2);

{1,2};

- push_front(), emplace_front, pop_back(), pop_front(), back(), front(),

- begin, end, rbegin, rend, clear, insert, size, swap.

4) Stack

all operations happens in @ O(1)

- Declaration

stack <int> st;

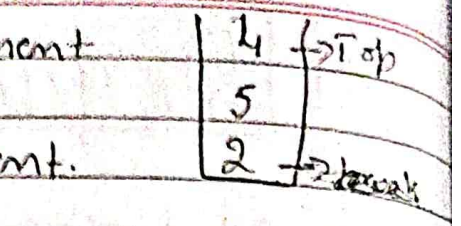
- st.push(1); // {1}
- st.push(2); // {2,1}
- st.push(3); // {3,2,1}

- st.emplace same as push operation

note: stack always follows LIFO Last in first out.

basically just based on a stack ex stack of books LIFO

- st.top() - gives the last element
- st.pop() - pops the last element.
- st.size() - gives the stack size
- st.empty() - checks if the stack is empty
- st1.swap(st2); - Swaps two stacks



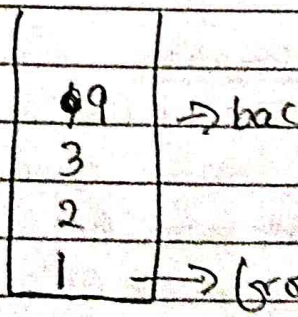
5) Queue

- all operations happen in big O(1)
- basically based on a general queue like first come first serve so (FIFO)
  First in first out

- queue <int> q;

```
q.push(1);
q.push(2);
q.push(3);
q.emplace(4);

q.back+=5;
```



cout<<q.back; o/p 9

- q.front() prints 1
- q.pop() → pops 1

front back

{1, 2, 3, 9}

- size, swap and empty same as stack.

6) Priority Queue

- works same as queues but prioritizes the queues as we want & declare it as to prioritize min value or max value
- when the queue prioritizes max value then it is know as [max heap]
- when the queue prioritizes min value then it is known as [min heap]
- The values are not stored as in a linear form but stored in a tree form

- q.front() prints 1
- q.pop() → pops 1

front back

{1, 2, 3, 9}

- size, swap and empty same as stack

6) Priority Queue

- works same as queues but prioritizes the queues as we want & declare it as to prioritize min value or max value
- when the queue prioritizes max value then it is known as [max heap]
- when the queue prioritizes min value then it is known as [min heap]
- The values are not stored in a linear form but stored in a tree form

- q.front() = prints 1
- q.pop() → pops 1

front back

{1, 2, 3, 9}

- size, swap and empty same as stack.

6) Priority Queue

- works same as queues but prioritizes the queues as we want & declare it as to prioritize min value or max value
- when the queue prioritizes max value then it is know as [max heap]
- when the queue prioritizes min value then it is known as [min heap]
- The values are not stored as in a linear form but stored in a tree form

Max heap

- The largest value element is always at top

declaration: priority_queue <int> pq;

```
pq.push(5);   // {5}
pq.push(2);   // {5, 2}
pq.push(8);   // {8, 5, 2}
pq.emplace(10); // {10, 8, 5, 2}
```

pq.top(); //prints 10;

pq.pop(); pops 10

- Size, swap and empty functions are the same as others

Min heap

- The smallest value element is always prioritized and is at top.

declaration: priority_queue <int, vector<int>, greater<int>> pq;

```
pq.push(10);
pq.push(5);
pq.push(8);
pq.push(2);
```

{2, 5, 8, 10}

pq.top() // prints 2

Time complexity

note: push - logn
top - O(1)
pop - logn

7) [Set] all functions is in (log N) runtime complexity

- Stores everything in sorted order and stores uniqe

decleration: set<int> st;

- st.insert(1);
  st.insert(2);
  st.emplace(2);
  st.emplace(4);
  st.emplace(3);

  // set will be in order
  {1, 2, 3, 4} -> sorted in order & no repetitions so uniqe.

- begin, end, rbegin, rend, size, empty and swap are the same functions.

- auto it = st.find(3);
  points to the element 3 in the set.

- auto it = st.find(6)
  if there use give an element which is not found on the set it shows the last element of the set.

- st.erase(5) - simply removes the element mentioned

7) Set — all functions is in (log N) runtime complexity

- Stores everything in sorted order and stores unique

declaration: set <int> st;

- st.insert(1);
st.insert(2);
st.emplace(2);
st.emplace(4);
st.emplace(3);

// set will be in order
{1, 2, 3, 4} → sorted in order
& no repetitions so unique.

- begin, end, rbegin, rend, size, empty and swap are the same functions.

- auto it = st.find(3);
points to the element 3 in the set.

- auto it = st.find(6)
if there we give an element which is not found on the set it shows the last element of the set.

- st.erase(5) - simply removes the element mentioned.

- `int count = St.count(i)`

  shows the number of occurrences of the entered element

- `auto it = St.find(3)`
  `St.erase(it);`

  iterates till 3 when the element 3 is found erase 3.

- `auto it1 = St.find(2);` {1, 2, 3, 4, 5}
  `auto it2 = St.find(4);`
  `St.erase(it1, it2);`

  erases from [2 -> 4)

  {1, 4, 5} not including 4.

find(); erase()
count();
insert();
important functions in sets

- `auto it = St.lower_bound`

lower bound

it returns the first occurence of the element if it exists in the set or else it returns the iterator pointing to the element which comes right after the element we want

Ex: {1, 3, 7, 9, 10} if 11 it points outside the set

auto it = st.lower_bound(7);

here the iterator points 7s position i.e 2nd index.

auto it = st.lower_bound(5);

here the iterator points 7s position because 7 is the immediate greater element present in the set

- auto it = st.upper_bound()

Ex: here it always returns the the next greater elements index of the element we want

Ex: {1, 4, 5, 6, 9, 9}

auto it = upper_bound(4)

points to 5

auto it = upper_bound(7)

points to first 9

auto it = upper_bound(10);

points after second 9

8) Multiset

- works in the same way as normal set but it is not unique
sorted but not unique

: declaration

```
Multiset<int> ms;
ms.insert(1);
ms.insert(1);   {1,1}
```

can have any number of repeated occurrences.

Erase

- when you try to erase 1 all 1s are erased

```
ms.erase(1);
```

- `int cnt = ms.count(1)` (count: keyword)
counts the occurences of 1 in the set

- `ms.erase(ms.find(1));`
erases just one occurence of 1
since find() points to the iterator and we're erasing that iterator the first element will be erased since the iterator starts at the beginning and moves further

[erase address]

ex: {1, 1, 1}
    ↑ it

first occurence of 1 gets erased.

to erase multiple elements

- ms.erase(ms.find(1), ms.find(1)+2);

erase(start address, end address)

- rest all functions same as sets.

9) unordered set

unordered set

TC = O(1) in worst case O(N)

- all the operations are the same as set but doesn't store it in sorted order & is unique and upper & lower ~~function~~ bound function doesn't work
- has better complexity except when there's a name collission.

10) Map TC = log n

- Map is a Data structure or a container which stores everything in terms of keys & value

Keys can be of any data type

Decleration:

- map<int, int> mp;
  (key, value)

- mpp[1] = 2; // {1,2}
  mpp.emplace({3,1});
  mpp.insert({2,4});

{1,2}
{2,4}
{3,1}

map

note: Map stores unique keys in sorted order and the keys are always unique even for repeated values.

- To print the above map we use a for each loop with an iterator

```
- for (auto it : mp)
  {
    cout << it.first << " " << it.second << endl;
  }
```

it.first → to print the key

it.second → to print the value of key

Since it is stored in pairs.

- cout<<mp[i];

this is to access Map of of 1
that is value of key 1 which is 2

- cout<<mp[5] - // gives 0 since there is no key 5.

- auto it = mp.find(3);
cout<<*(it).second; // finds the address of element ~~there~~ 3 using iterator and prints the value of key 3 i.e 1.

- auto it = mp.find(4) // this wont be found on the map since it doesnt exist so the find() points to mp.end() which is after the last element.

- auto it = mpp.lower_bound(2);
auto it = mp.upper_bound(3);

- erase, swap, size, empty, are all the same way.

- cout<<mp[1];

this is to access Map of of 1
that is value of key 1 which is 2

- cout<<mp[5]-// gives 0 since there is no key 5.

- auto it = mp.find(3);
cout<<*(it).second; // finds the address of element ~~there~~ 3 using iterator and prints the value of key 3 i.e 1.

- auto it = mp.find(4) // this wont be found on the map since it doesnt exist so the find() points to mp.end() which is after the last element.

- auto it = mpp.lower bound(2);
auto it = mp.upper bound(3);

- erase, swap, size, empty, are all the same way.

11) Multimap

- everything is same as a map but multiple keys can be stored and in sorted order

  Ex: {1, 2}
  {1, 3}

12) Unorderd Map

- Same as Maps but not sorted
  unique keys but not sorted
- O(1)
  worst case O(N)

✳ Algorithms

most important algos

Sorting

- assuming an array ~~or vector~~ a[n]
  to sort the array we can use

  `Sort(a, a+n)`

  Ex: {1, 3, 5, 2}
  a a+n
  {1, 2, 3, 5}

  Sorts the array in ascending order

- for vector

```
sort(v.begin(), v.end());
```

- starting address of element to the location after the last element.

- if you wanna just sort a ~~get~~ subset then

```
sort(a+2, a+4);
```

value included ↑ , not included →

```
   0  1  2  3  4
ex: {1, 3, 5, 2, 6}
    a+0 +1 +2 +3 +4
```

```
    ↓  ↓
{1, 3, 2, 5, 6}
```

- to sort in descending order

```
sort(a, a+n, greater<int>());
```

→ comperator

```
ex: {1, 3, 5, 2, 6}
          ↓
    {6, 5, 3, 2, 1}
```

* [Comparators]

just a function used to sort anything in our preffered way.

ex: bool comp (pair<int, int> p1, pair<int, int> p2)
{
if (p1.second < p2.second) return true;
if (p1.second > p2.second) return false;

if (p1.first > p2.first) return true;
return false
}

pair<int, int> a[] = {{1, 2}, {2, 1}, {4, 1}}

sort (a, a+n, comp);

// sorts in wrt to 2nd element in the array
ie by considering only 2nd element

// if second element is same then consider first element

* [Built in popcount()]

ex: int num = 7;
int cnt = __builtin_popcount();

returns the number of set bits or the number of 1s here 7 is binary is 111 so cnt = 3

- if the number is long long then

Ex: long long num = 165687982431
=
long long int cnt = __builtin_popcountll();

Permutation

- next_permutation(s.begin(), s.end());

this command gives all permutations of a string ex:

Ex: string s = "123"
=
sort(s.begin(), s.end());
do
{
cout << s << endl;
}
while(next_permutation(s.begin(), s.end()));

- here sort is used because if the assigned string is 231 then the permutation will start from 231 i.e just 231
312
321
but sorted makesit 123
132
213
231
312
321

* Max & Min T/C

max

- int max = *max_element(a, a+n)

min

- int min = *min_element(a, a+n)

* Basic Math Concepts

1) Digits

- basically when a number N is given to extract all the digits individually one by one you just have to get the remainder of the number which is by doing N % 10 (N modulus of 10)

- and then divide N by 10 to eliminate the last number.

ex: 7789 % 10 = 9
÷10 → 778 % 10 = 8
÷10 → 77 % 10 = 7
÷10 → 7 % 10 = 7
÷10 → 0

Pseudo code

```
int n;

while (N>0)
{
  int last digit = N %. 10;
    print (last digit)

  N = N/10;
}
```

Time complexity

$O(\log_{10}(N))$

whenever there is division happening in a problem here N is being divided by 10 so here $(\log_{10}(N))$

if it was being divided by the $(\log_5(N))$

- when the number of iterations depends on division at that time the TC will be logarithmic

$O(\log(N))$

123/10 ⇒ removes the last digit
here ans = 12
12.3 but stores only 12 [quotient]

123%10 ⇒ fetched the last digit so here ans is 3
12.3 stores 3 [Remainder]

Recursion

- When a function calls itself until a specific condition is met
  Ex: to print 1

```
#  void f()
   {
     print(1);

     f(); — // keeps calling the f() function
   }          intionelly until a condition is
int main()    given
{
    f();
}
```

- this above ex creates a **stack overflow** which means it exceeds a memory limit or also known as **segmentation error**

this happens when there are multiple function calls waiting in the stack to be called (from main function)
ifinite recursion

- **Stak space** - Stores the functions that are yet to be completed — it won't be completed & the flow won't be returned to main

Base condition

- a base condition has to be specified to stop the recursion.

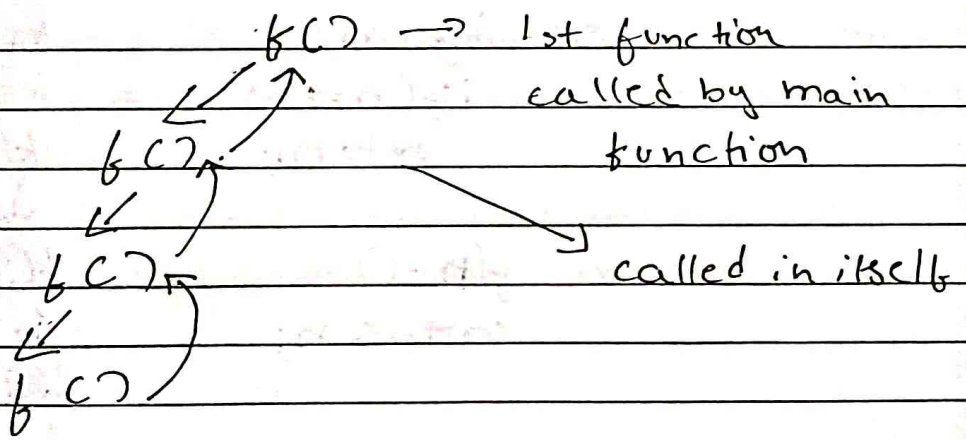
ex:
```
void f(int
{
count=0; count=0;

if (count==4);
return;

print(count)
count++;
f();
}
```

Recursion tree

for the above program
recursion tree can be represented as



Basic Steps to solve a recursion Problem

1. first try & get a base case by doing the problem with the least value so that you'll know where to stop the recursion

2. any problem you're given breaks it down into smaller problem its like building up to the big solution

ex: to get n! n=4

4! = 4 x 3!

3! = 3 x 2!

ex: to get nth fibonacci

get fibonacci of (n-1) & (n-2)

f(n) = f(n-1) + f(n-2)

f(7) = f(6) + f(5)

Note: any statements after recursive function call will be waiting until the process returns.

ex: with f(n)

```
if(n==0)
    return;

f(n-1)
cout<< n;
```

n = 4

f(4) → cout waiting to print 4

f(3)

f(2)

f(1) & here cout n value is print 1.

f(0) returns to last execution point

n=0

Hashing

- basically what hashing is to find out the number of times an element can be found in an array. [vvv Basic thing of function]
- Data retrival [main function]
- efficiently.

- whenever youre declaring an array of int type the max you can declare inside the main function is $[10^6]$ larger cant be allocated in memory
  - but if declared globally it can be $[10^7]$
  - and bool array can be declared with $[10^7]$

but in main if int array is inside main exceeds 10^7 then it gives a segmentation fault

So in main max is - int arr $[10^6]$
globally max is - int arr $[10^7]$

Character hashing

Pseudocode - (Brute force)

```
f (char c, s)
{
    count=0;

    for(i=0; i<n; i++)
    {
      if(s[i]==c)
        count++;
    }
    return count;
}
```

T.C= for every query Q you take N time

$O(Q \times N)$

Time complexity of Hashing using Maps

two main operations storing & fetching (arr or char)

takes $(\log N)$ time

↳ No of elements

↓

in all cases

best, avg & worst

- $O(\log N)$ - using hashing maps — best avg worst
- $O(1)$ - Hasing using unorder map — best, avg

- ~~in worst~~ O(N) / O(N²) - Hashing in unordered map worst case (rarely)

* always use unordered map to hash.

hashing methods or working

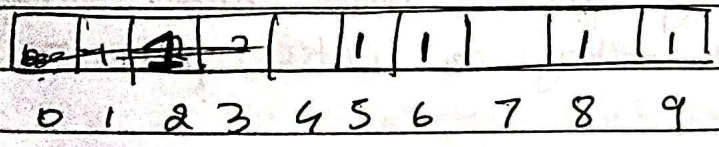
- Division method -> ✳
- Folding method
- mid square method } not so important.

Division method

Ex: [2, 5, 16, 28, 139] and array size shouldnt be greater than 10 n=10

at that time the elements in the array are reduced to single digit by doing [element %. 10] and these modulated single digits are mapped to hash arrays index.

2%.10=2 , 5%.10=5 , 16%.10=6 , 28%.10=8 , 139%.10=9



just stores the numbers based on last digit

- But if multiple numbers have the same last digit it gets chained in a sorted order using linked list

Ex: {2, 5, 6, 18, 28, 8, 38}

0
1
2 → 1
3
4
5 → 1
6 → 1
7
8 → 8 → 18 → 28 → 38 (chaining)
9

collision

- for ex in the above example if there were a lot more elements chained to 8 it would take a lotta time to find the element and this is known as collission

- Map - any type of data structure can be used as a key ex vector, pair or anything

- Unmap - Only single type like chars, int, Double etc can be keys

Sorting

1. Selection Sort - Basically the array into smaller like you're finding the array and swap element

ex: n = 5

step1:

step2: since 0th whole arr and find repeat

step3:

logic: move th from t 0th ele compare to fi and s place secon

- but if multiple numbers have the same last digit it gets chained in a sorted order using linked list.

Ex: {2, 5, 6, 18, 28, 8, 38}

0
1
2 → 1
3
4
5 → 1
6 → 1
7
8 → 8 → 18 → 28 → 38 (chaining)
9

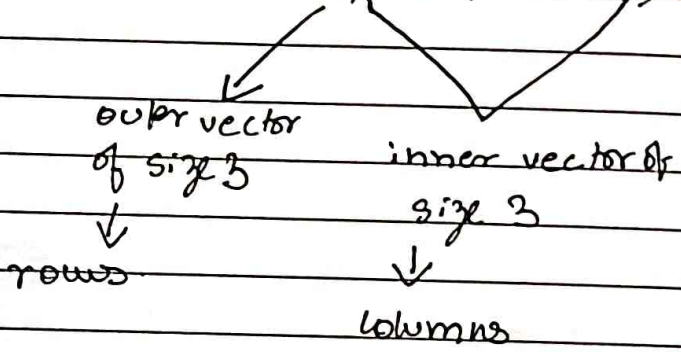
collision

- for ex in the above example if there were a lot more elements chained to 8 it would take a lotta time to find the element and this is known as collision

- Map - any type of data structure can be used as a key ex vector, pair or anything
- Unordered map - only single type like char, int, Double etc can be keys.

Vector initialization

- Vector<int> - Declares a vector of 1D array
- Vector<int> V(3) - Declares a vector of size 3
- Vector<int> V(3,5) - Declares a vector of size 3 all 3 as 5
  V = {5,5,5}
- ~~Vector<int>~~
- Vector<vector<int>> V; - Declares a vector of 2D array
- vector<vector<int>> V(3, vector<int>(3));



- vector<int> v / vector<vector<int>> V;
  n = V.size(); → gets size of the vector.

gets the whole size.

1 2 3

n = 3 ex: {(1,0) (1,1) (0,1)}

returns 3 size.

if given like v[0].size();

then it would return the size of first subarray i.e 2 (1,0)

1D - vector<int>V;
- V.front() - returns first element of array
- V.back() - returns last element of the array
- V[1] - returns 1st element [0, 1 . . . .] index from any specific index

2D - vector<vector<int>>V; ex: {{1,0,1},{1,1,1},{2,0,1}}
- V.front() - returns the first subarray i.e {1,0,1}
- V.front()[2] - returns first subarrays 2nd index ele i.e 1
- V.back() - returns the last subarray i.e {2,0,1}
- V.back()[1] - returns 0

- reverse(a.begin(), a.begin()+k); k=2, a[1,2,3,4,5]
this would reverse from 0→2 i.e 0, 1 index 0 to k
- reverse(a.begin()+k, a.end());
would reverse from index 2 to end

31/10/24

Binary Search

- Binary search is a search algorithm that is used to search an element in an array or any other search space where the search space is sorted.

working

- So basically in a given search space to find the target element
- we divide the search space into two halves
- & compare the mid to target if mid = target then we just return that since we found the element
- if target is greater than mid that means target is in the right half of mid or lies after mid to end so we move start point to mid+1 (decrease search space to righ half of search space mid+1 to end)
- but if target is lesser then we decrease the search space to left half i.e start to mid-1 (so high becomes mid-1)
- keep on repeating this until you find the element i.e when mid = target or until low > high (i.e when theres no target ele present)

Time complexity $O(\log n)$

- we're dividing the search space into 2 equal halves each time
- Basically a number n is divided by 2 x times until we find our target or until it's no longer dividable

So n being divided by 2 x times

$$2^x = N$$

Ex: N = 32

$2^5 = 32$

$x = 5$

1 - 32/2 = 16
2 - 16/2 = 8
3 - 8/2 = 4
4 - 4/2 = 2
5 - 2/2 = 1

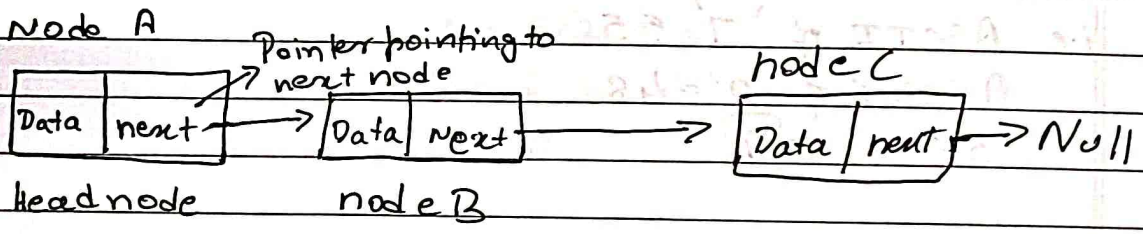
* base is usually 2 which we divide the given thing by 2 for x times to get N

* logn means we keek dividing N by 2 x times until no longer dividable

Linked Lists

- not stored in contigious memory locations in Heap memory its random
- size can be increased anytime or decrease
- Starting point of linked list is called head
- last point is ~~Node~~ tail.

ex: used in browser forward & backward page nav



main operations of linked list

1. Traversing a linked list
start from first node & start moving through each node till the last node.

2. Append a new node
adding a new node to the end of the list

3. prepend a new node
create and add a new node to start of the list

4. Insert in specific location

5. Deleting a node

6. Updating a node

Types of linked list

1. Singly linked list
- only one way traversal only from start to end no backword traversal since there no backword link [D|N] → [D|N]

2. Doubly linked list
- two way traversal possible

link for previous node

3. Circular linked list
a singly linked list but the last node link points back to first node instead of null

4. Pointer - basically a variable used to store the memory & location of a variable

eg: int var = 5;
int *ip;
ip = &var;

~~pointer~~ ip = 0x6ffe34
*ip = 5 (pointing to memory location value)

Structure or Struct

- a structure is basically a user defined data type

```
ex: Struct person
    {
      string name;
      int age;
      int salary;
    }
```

this would take a lot of lines taking input for multiple people so instead use array

```
int main()
{
  person P1;

  cin >> P1.name;
  cin >> P1.age;
  cin >> P1.salary;

  cout << the same of all P1.
```

#

```
int main()
{
  Person p[2];

  for(int i=0 ; i<2 ; i++)
    {
      cin >> p[i].name;
             p[i].age;
             p[i].salary;
    }
```

this takes input for 2 people

each index of array will have all 3 details of 1 person

using pointers

```
Person *ptr;
person p1;

    cin >> ptr -> name;
    cin >> ptr -> age;
    cin >> ptr -> salary;
```

Linked list continue

```
class Node
{
  Public:
  int data;
  Node *next;

  Public:
    Node (int d, Node *ne)
    {
    data = d;
    next = ne;
    }

  Public:
    Node(int d)
    {
    data = d;
    next = NULL;
    }
};
```

```
int main()
{
vector<int> V = {2,5,8,9}
Node *obj = new Node (V[1])
cout << obj; // prints memory
                address

cout << obj -> data; // prints
                value of that
                address.
```

3

Stacks & Queues

* Stack
- A stack is a DS that holds any kind of data
- follows LIFO - Last in First Out

functions
- Push()
- Pop()
- top()
- size()

* Queues

- A stack is a DS that holds any kind of data
- uses FIFO - First in First out

functions.
- Push()
- Pop()
- top()
- size()

Inbuilt library
- stack <int> s;
- queue <int> q;

* Stack using arrays (somewhat we'll have extra space left if not filled)

- size should be defined

```
siz = [10];

class stkimp
{
  int top = -1;
  int st = [10];

Push (x)
{
   if (top >= 10) exceeds size
   top = top+1;
   st [top] = x;
  }

top ()
{
  if (top == -1) return empty

  return st [top];
}

pop ()
{
   if( top == -1)
      top = top-1;
}

siz ()
{
  return top+1;
}
}
```

* Queue using Arrays

* Prefix, Infix, Postfix conversions

* Operator & Operand
  - Operators - ^, *, /, +, -
  - Operand - A-Z, a-z, 0-9

* Priority order
  ^ - 3 highest
  *, / - 2
  +, - 1 lowest
  anything else - 1

* Post, pre & Infix
  - (P+Q) * (M-N) - Infix → mostly used in all langs
  - * + QP - MN - Prefix → used in LISP & DCCS
  - PQ + MN - * Postfix → stack based calculators

Trees

- Trees is a data structure used to store and organize data in a hirearchial way or in multilevel scavences.

- consists of nodes where each nodes can have at most 2 chid nodes left & right child

- Nodes - contains a piece of data
  - also has refrence pointer to its childs.

- Internal node - non leaf node
- Root node - top most node

- Child nodes.

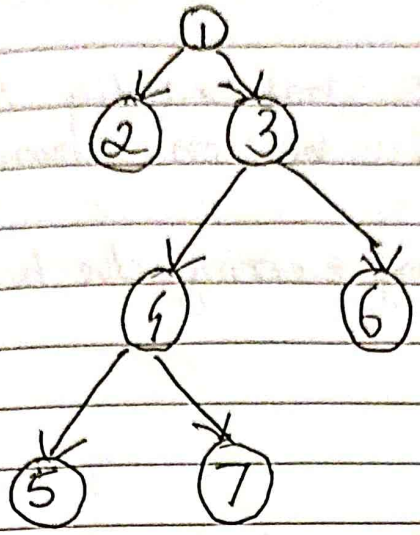
- Leaf nodes - doesnt have any chids

- ancestors - the path that leads to the root node from a certain node is all ancestors.

* Full binary tree (strict binary tree)

- all internal nodes should have only 0 or 2 children (no single child allowed)

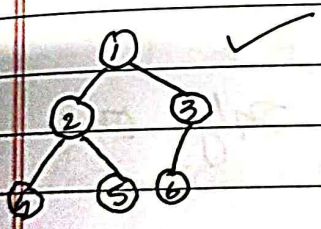
- this balances the tree
- makes traversal, searching & insertion effecient



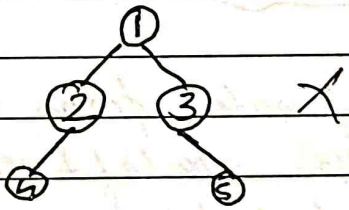
* complete binary tree

- here all levels are supposed to be filled except last level or 2nd last level

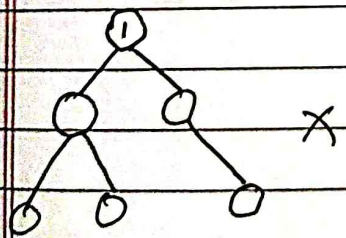
- if last level is not filled then it should be filled from left



✓



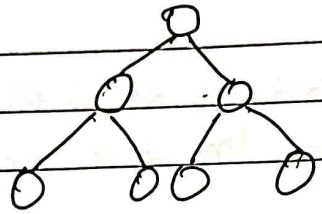
X


X

- used in heap based sort

* Perfect binary tree

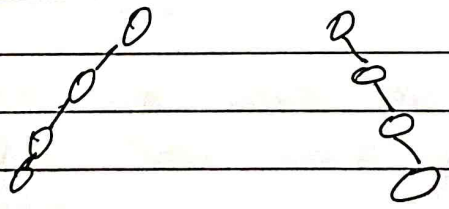
- every internal node has exactly two children & all leaf nodes will be on same level
- effecient for searching & sorting due to balanced nature



* Balanced Binary Tree

* Degenerate trees

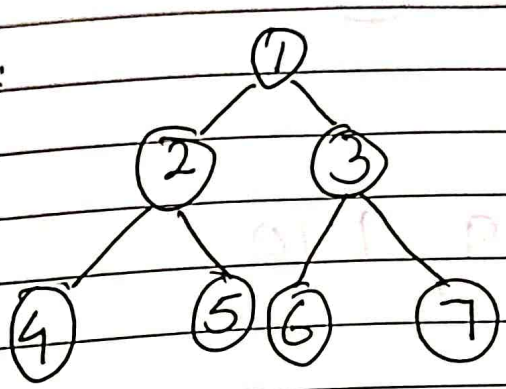
- Every node from the root has only one child to left or right like linked list



Traversal techniques

* DFS (Depth first search)
  - a. Inorder (Left Root Right)
  - b. Preorder (Root left Right)
  - c. Post order traversal (Left Right root)

ex1:



(a) Inorder (left root right) root in between

4 2 5 1 6 3 7

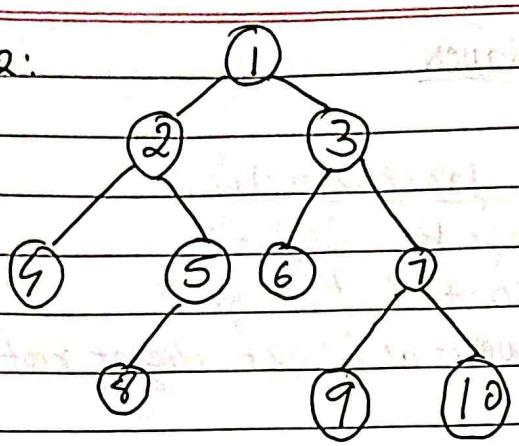
(b) Preorder (Root left Right) root at pre

1 2 4 5 3 6 7

(c) Post order (Left right root) root at post

4 5 2 6 7 3 1

Ex2:



- Inorder

4 2 8 5 1 6 3 9 7 10

- Preorder

1 2 4 5 8 3 6 7 9 10

- Postorder

4 8 5 2 6 9 10 7 3 1

* Breadth first search

- level wise traversal

for above tree

1 2 3 4 5 6 7 8 9 10