* Linked List :

One disadvantage of using array to store data is that ; arrays are static structure's & therefore cannot be easily extended or reduced to fit the dataset.

Arrays are also expensive to maintain new insertions & deletions.

A Linked list is a linear data structure where each elt is a separate object. Each elt of the list contains 2 items

node ← i) data ii) Reference / Pointer to next node.

The last node has a ptr to "NULL".

A LL is a dynamic DS, the no. of nodes in a list is not fixed ; can grow & shrinked on demand.

Any appl^n which has to deal with an unknown no of objects will need to use a LL.

One disadvantage of LL against array is that it does not allow direct access to individual elts.

If you want to access particular item than you have to start at the head & follow the references until you get to that item.

→ When to use Linked List ?

i) You need constant time, insertion, deletion from the list.

ii) you don't know how many items user will i/p.
iii) you do not need random access to any elts.
iv) If you want to insert an item in the middle.

→ When to Use Array?

i) you need random access to elts.
ii) you know the no of elts in the array, ahead of time so that you can allocate the correct amt of memory for the array.
iii) you need speed when iterating through all the elts in sequence.
iv) Memory is a concern for you.

- Types of Linked List:

1] Singly Linked List -

2] Doubly 3] Circular

*Note Abstract Data Types (ADT)

The ADT is an abstraction of a DS. An ADT specifies following:
i) Data store
ii) Operations on data (functions/methods)
iii) Error conditions associated with operations.

i) Data -
This part describe the structure (int, float, class, structure) of the data used in ADT.

ii) Operations -

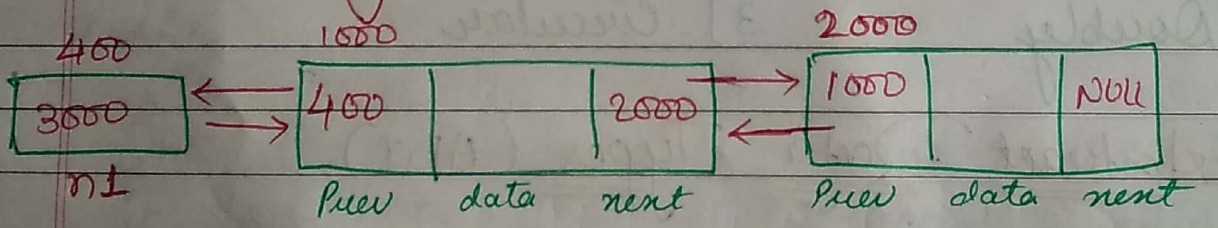
This part describes valid operations for this ADT. we use the special operations constructor to describe the action which are to be performed once an entity of this ADT is created.
object

and destructor to describe the actions which are to be performed once an entity is about to destroy.

iii) Error -

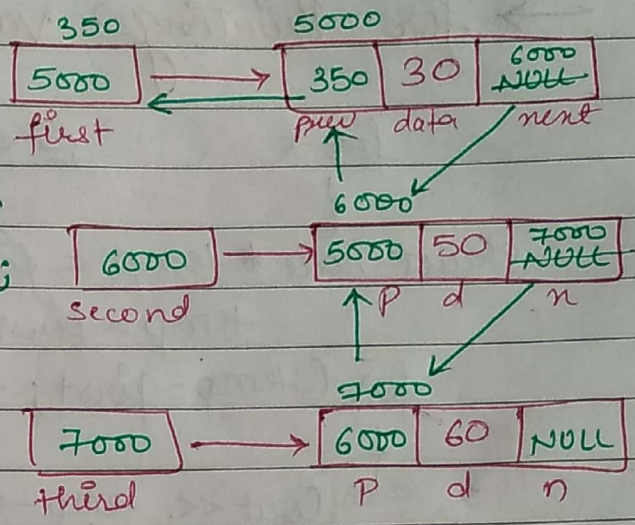
This part describes the error that can occur & how we are going to deal with this errors.

* Doubly Linked List :-



Steps :- i) create a node.
ii) initialize datamember of a node.
iii) Connecting current → next & then prev → current.

```
Struct node
{
    int data;
    Struct node * next;
    Struct node * prev;
};

int main()
{
```




```
Struct node * first = (Struct node *) malloc (- " - );
        first -> data = 30;
        first -> next = NULL;
**      first -> prev = & first;

Struct node * second = ( - " - );
        second -> data = 50;
        second -> next = NULL;
        second -> prev = NULL;

**  [ first -> next = second;
      second -> prev = first; ]

Struct node * third = ( - " - );
        third -> data = 60;
        third -> next = NULL;
        third -> prev = NULL;

**  [ second -> next = third;
      third -> prev = second; ]
    return 0; }
```

→ for Printing values :

~~for (~~ ~~i < sizeof(node) ; i++)~~

~~{~~

struct node * temp = ~~first~~ NULL;

temp = first;

for (temp = first; temp != NULL ; temp = temp → next)

{

cout << temp → data ;

}

→ using while

struct node * temp = first;

while (temp != NULL)

{

cout << temp → data ;

temp = temp → next ;

}

→ for printing data in reverse order.

for (temp = first; temp != NULL ; temp = temp → next);

{

for (temp = last ; temp != first ; temp = temp → prev)

{

cout << temp → data ;

~~temp = temp~~ }

PAGE No.
DATE / /

→ using while Reverse logic

```
while( temp != first)
{
    cout << temp -> data ;
    temp = temp -> prev ;
}
```

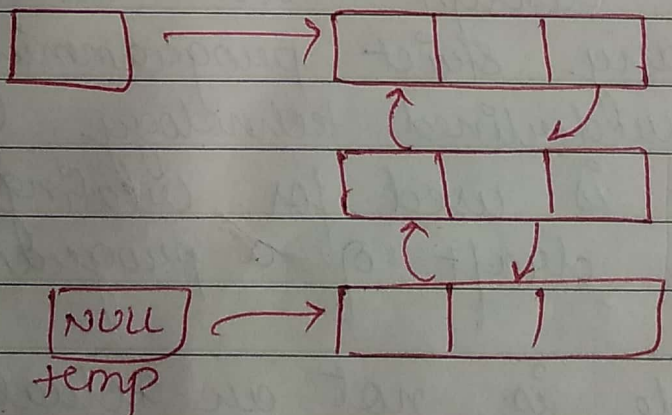
5th March 2020

153

#

temp->next != NULL

```
for (temp= first ; temp != NULL ; temp= temp -> next) ;
{
    while (temp != first) // (temp != NULL)
    {
    cout << temp -> data ;
    temp = temp -> prev;
```



* Algorithm :

An algorithm is sequence of unambiguous instruct^{n} for solving a problem. An algo is a series of content steps which you follow in order to achieve some goal or to produce some o/p.

eg

Receipe for making a cake.

One problem can be solved in 100 different ways. some solutions are just more efficient taking less time & require less space than others

* Pseudo Code :

Pseudo code is an informal way of programming descript^{n} that doesn't require any strict programming language syntax or underlined technology consider^{t}.

It is used for creating an outline or a rough draft of a progedam.

Pseudo Code is not an actual Progeamming language so it cannot be compiled into an executable progeam. It uses short terms or simple english language syntax's to write Pseudo Code.

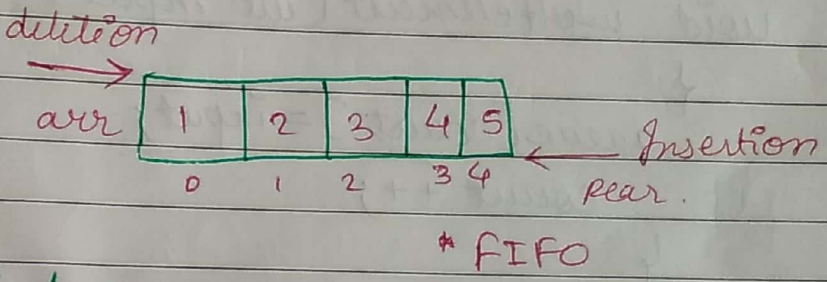
PAGE No.
DATE / /

→ Advantages of Pseudo Code

i) Pseudo code is understood by the programmers of all types.

ii) It enables the programmer to concentrate only on the algorithm part of the code development.

* QUEUE



* FIFO

Basic Code :-

(1)
```
int queue[3];
q[0] = 10;
q[1] = 20;
q[2] = 30;

cout << q[0];
cout << q[1];
cout << q[2];
```

(2)
```
int queue[3];
int count=0;
or insertelement(10);

q[count]=10;
    count++;
or insertelement(20);
```

```
q[count]=20;
  count++;
or
  insertelement(20);

  q[count]=30;
  count++;
  or
    insertelement(30);
}
void insertelement(int input)
  {
    queue[count]=input;
      count++;
    }

void
```

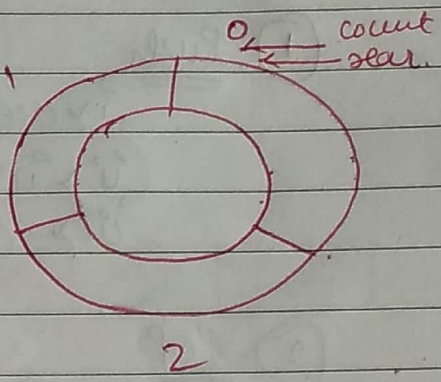
Simple Code

```
int queue[3];
int main()
{
    int read=0;        // → front
    int count=0;       // → rear
    insertelement(10);
    insertelement(20);
    insertelement(30);

    show();


    return 0;
}

void insertelement(int input)
{
    int res = isempty();   // full
    if(res==1)
    {
        queue[count]=input;
        count++;
    }
    else
    {
        cout<<"Queue is full";
    }
}
```

```
int
void isfull()
{
    if (count == 3)
    {
        cout << "Queue is full";
        return 0;
    }
  else
    {
      return 1;
    }}

void
void show()
{
  int res = isempty();
          0
      if (res == 1)
      {
      cout << "Queue is empty"; //return 0;
      }
      else
      {
  cout << queue[read] << endl;
      read--;
    }}

void isempty()
{
    if (count == read)
    {
        return 0;
    }
```

else
{ return 1; }

* Circular Queue :



Insert
{
rear++;
queue[rear] = 10;
}

rear = -1
front = -1

Deletion
{
(rear == size) // Queue is full.
{
rear+1/3;
front+1/3;
// rear+1 % 3;
front+1 % 3;

* Stack Using Linked List :-

(1) Push
   i) create node using rear elt.
   ii) insert given data/input.
   iii) connect node

(2) Pop
   print poped elt
   disconnect node with previous one.

6th March 2020

Program to demonstrate Linked List Using Class.

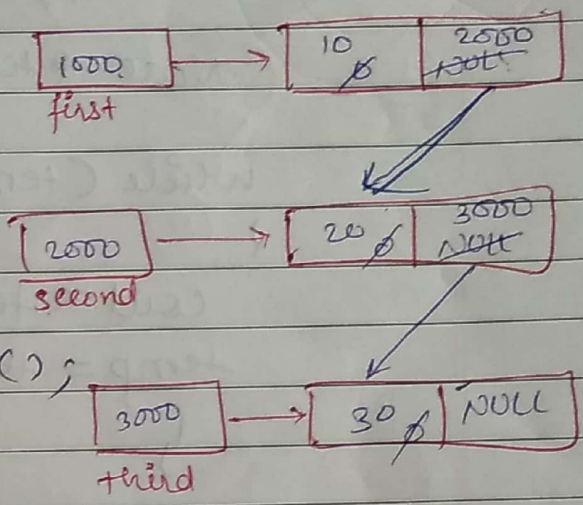
```
Class Node
{
  int data;
  Node *next;
Public:
void setData(int input)
{
    data = input;
}

void setNext(Node *temp)
{
    next = temp
}
```

```
int getData()
{
return data;
}
```

so that code will become more meaningful

```
Node()
{ data=0;
next=NULL;
}
```

```
Node *getNext()
{
return next;
}};
```




```
int main()
{
Node *first = new node();
first → setData(10);
first → setNext(NULL);

Node *second = new node();
second → setData(20);
second → setNext(NULL);
```

n1 → next = n2

```
✗ first → getNext() = second;
```

first → getNext(second)

```
Node *third = new node();
third → setData(30);
third → getNext(NULL);
```

second → getNext(third).

```
cout << first → getData(); 10
cout << first → getNext() → getData(); 20
cout << first → getNext() → getNext(); NULL
cout << first → getNext() → getData(); 30
return 0;
}
```

Using loop for printing Purpose.

```
/* while (first->getNext() != NULL)
    {
      cout << first->getData();
    } */
Node *temp = first;

while (temp != NULL)
{
  cout << temp->getData();
  temp = temp->getNext();
}
```

* Circular Queue:

A circular Queue is a linear DS in which operations are performed based on FIFO principle. The last position is connected back to the first position to make a circle.

In a queue we can insert elts until the queue becomes full. But once it is full we cannot insert an elt even if there is a space infront of it.

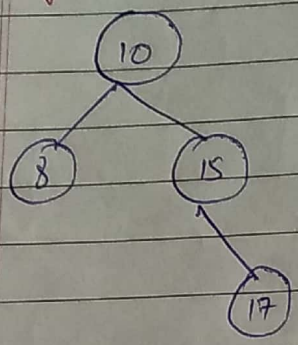
Once your entire CQ is filled, & if you delete some data you will be able to put in more data as the rear variable will start repeating itself.

* Recursion :

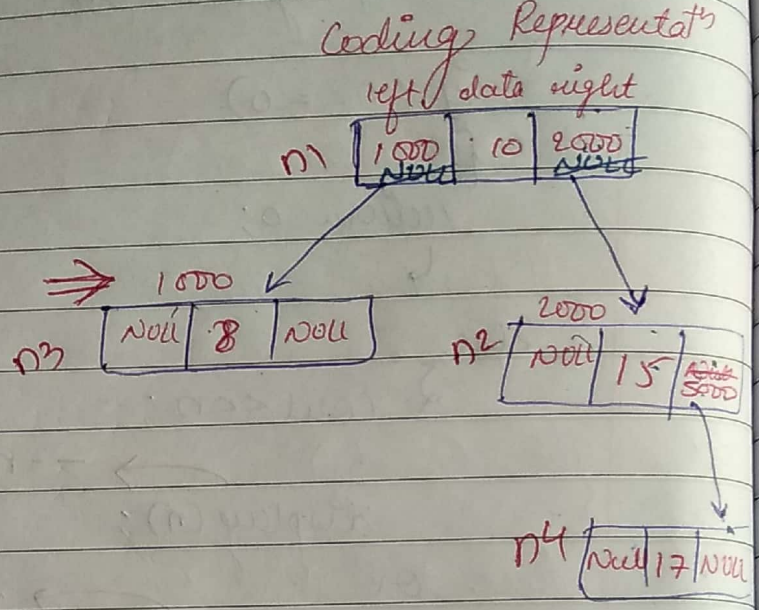
```
int display (int n)
{
  if (n==0)
  {
    return 0;
  }
  else
  { cout<<n;
      --n;
    display(n);
  or
    n--;
    display(n);
  }
}

void display ()
{
  for(int i=0;i<=5;i++)
  {
    cout<<i<<endl;
  }}
```

* TREE *

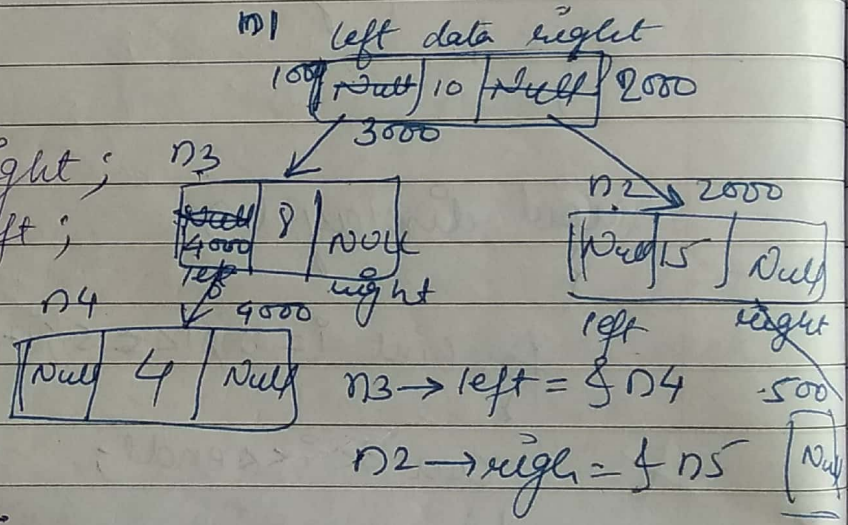
Logical Representation



Coding Representation




```
Struct node
{
 int data;
Struct node * right;
Struct node * left;
};
int main()
{
Struct node n1;
 n1.data = 10;
 n1.right = NULL;
 n1.left = NULL;

Struct node n2;
 n2.data = 15;
 n2.right = NULL;
 n2.left = NULL;
 n1.right = &n2;
```



```
struct node n3;
n3.data = 8;
n3.right = NULL;
n3.left = NULL;
n1.left = &n3
```

```
struct node n4;
n4.data = 17;
n4.right = NULL;
n4.left = NULL;
```

```
n2.right = &n4;   OR   n1.right->right = n4
```

```
return 0;
}
```

→ Program to traverse the list without using n2, n3 & n4.

```
{ struct  cout << n1.data;
          cout << n1.right;
          cout << n1.left;
```

n1 : 10, 2000, 1000

n5 : 17 / NULL

n2 :
```
cout << n1.right->data;
cout << n1.right->right;
cout << n1.right->left;
```
15. 5000. NULL

n3 :
```
cout << n1.left->right;
cout << n1.left->left;
cout << n1.left->data;
```
NULL, NULL, 8.

```
cout << n1.right->right->right;
cout << n1.right->right->left;
cout << n1.right->right->data;
```
NULL, NULL, 17

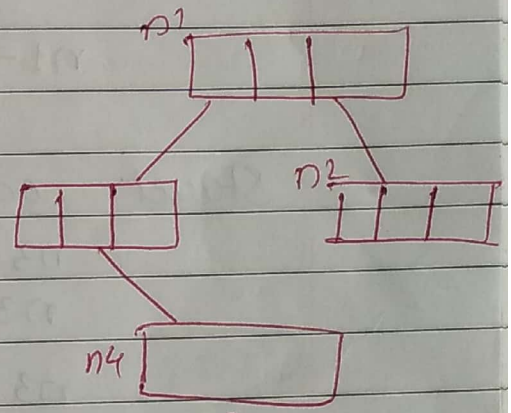
→ Using malloc() funct^n (DMA)

```
#include<iostream>

using namespace std;

struct node;
{
  int data;
  struct node * right;
  struct node * left;
};

int main()
{

struct node *n1 = ((struct node *) malloc
                    (sizeof(struct node));
        n1 → data = 10;
        n1 → right = NULL;
        n1 → left = NULL;

struct node *n2 = ((struct node *) malloc
                    (sizeof(struct node));
        n2 → data = 20;
        n2 → right = NULL;
        n2 → left = NULL;

        n1 → right = n2;
        n1 → left = n3;
```

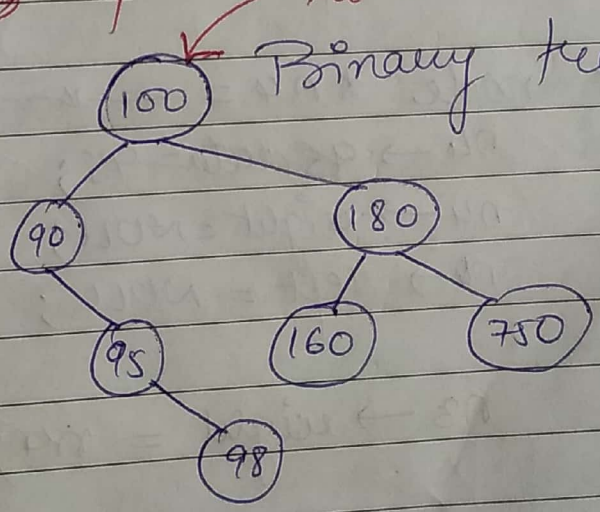
DATE / /

```
Struct node *n3 =((struct node *) malloc
                  (struct node ));
n3 -> data = 30;
n3 -> right = NULL;
n3 -> left = NULL;

Struct node *n4 =((struct node *) malloc
                  (struct node));
n4 -> data = 40;
n4 -> right = NULL;
n4 -> left = NULL;
```

```
n3->right=n4;
```



→ Program to demonstrate tree with funct & loops.




```
Struct node
{
  int data;
  Struct node * right;
  Struct node * left;
};
```

int main()
{
struct node *n1 = ((struct node *) malloc(--));
n1 → data = 100;
n1 → right = NULL;
n1 → left = NULL;

struct node *n2 = ——— ;
n2 → data = 180;
n2 → right = NULL;
n2 → left = NULL;

n1 → right = n2;

struct node *n3 = ——— ;
n3 → data = 90;
n3 → right = NULL;
n3 → left = NULL;

n1 → left = n3;

struct node *n4 = ——— ;
n4 → data = 95;
n4 → right = NULL;
n4 → left = NULL;

n3 → right = n4;

struct node *n5 = ——— ;
n5 → data = 98;
n5 → right = NULL;
n5 → left = NULL;

```
n4 -> right = n5;

Struct node * n6 = ---11---;
    n6 -> data = 160;
    n6 -> right = NULL;
    n6 -> left = NULL;

    n2 -> left = n6;

Struct node * n7 = ---11---;
    n7 -> data = 750;
    n7 -> right = NULL;
    n7 -> left = NULL;

    n6 -> right = n7;
```


7th March 2020

* TREE

Tree is a non-linear DS. It doesn't store data in a linear way. It organize data in a Hierarchical way.

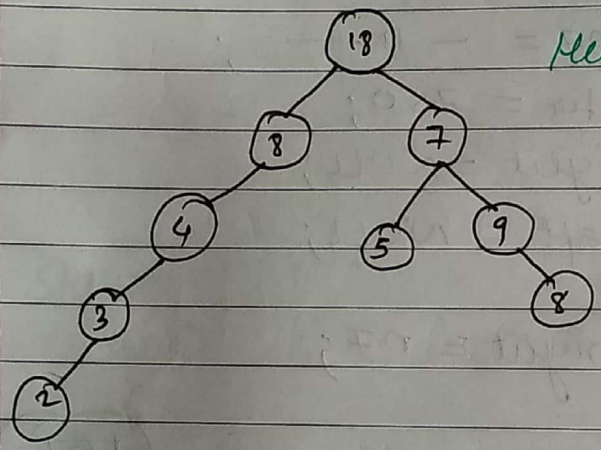
A tree is a collection of entities called node connected by edges. Each node contains a value or data & it can also have child node or not. The first node of a tree is called root.

If the root node is connected by another node, the root is a parent node & the connected node is child.

Leaves are the last node from the tree.
(nodes without children).

→ Height of tree :-

The height of a tree is the length of the longest path of a leaf.

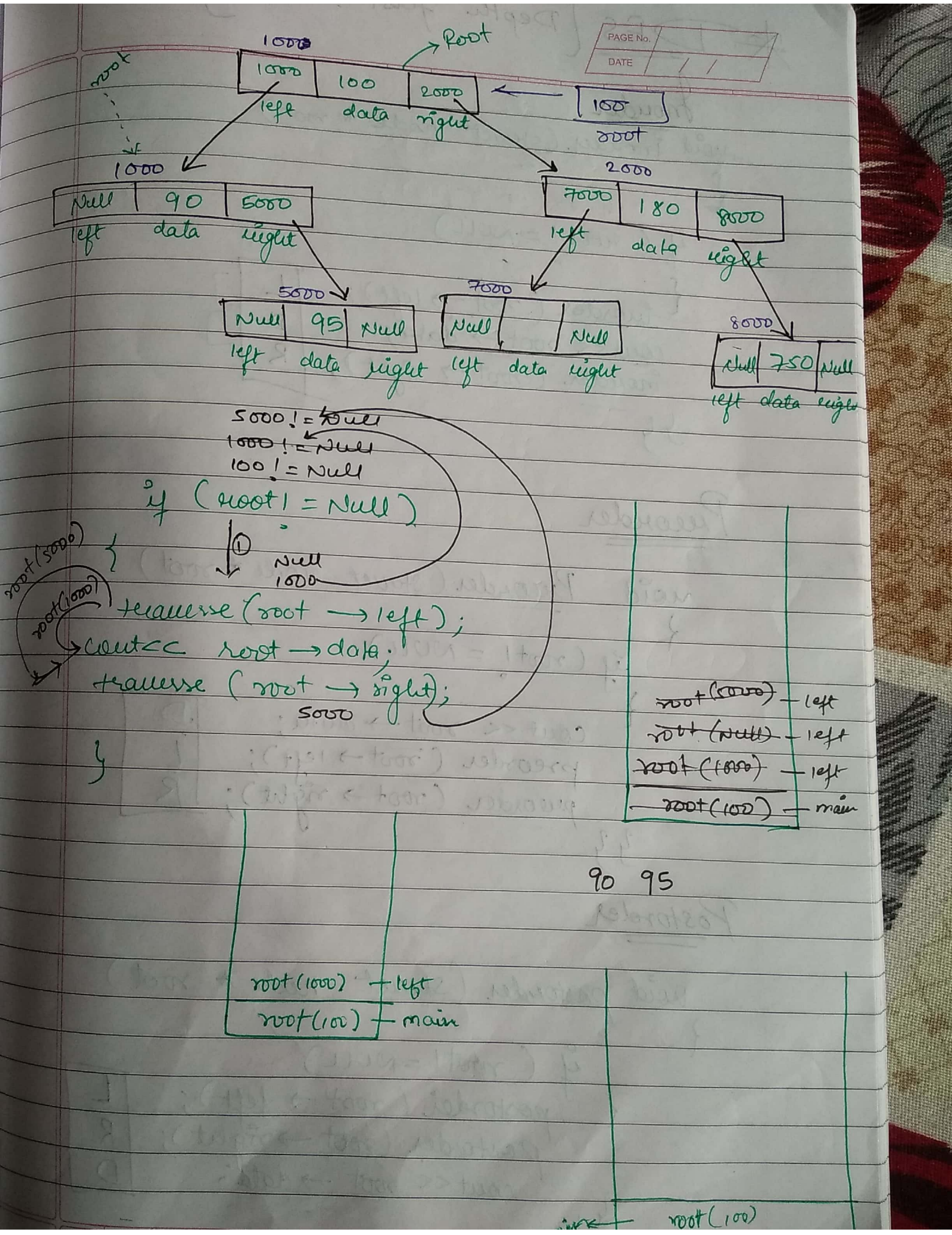


Height of tree = 4

→ To traverse tree :

```
int traverse (node * root)
{
cout << " calling traverse function";
    if (root != NULL)
    {
        return; // Syntax error
    }x
traverse (root → left);
cout << root → data;
traverse (root → right);
}✓
```

cout << root → data;
traverse (root → left);
traverse (root → right);



* DFS [Depth first Search]

PAGE No.
DATE / /

Inorder

```
void inorder (struct node * root)
{
    if ( root != NULL)
    {
        inorder (root -> left);
        cout << root -> data;
        inorder (root -> right);
    }
}
```

L
D
R

Preorder

```
void Preorder (struct node * root)
{
    if (root != NULL)
    {
        cout << root -> data;
        preorder (root -> left);
        preorder (root -> right);
    }
}
```

D
L
R

Postorder

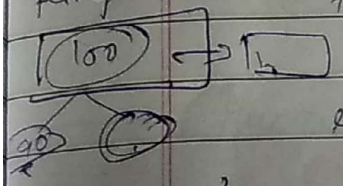
```
void postorder (struct node * root)
{
    if (root != NULL)
    {
        postorder (root -> left);
        postorder (root -> right);
        cout << root -> data;
    }
}
```

L
R
D

Program to demonstrate a Basic Tree functionality

PAGE No. / DATE / /

```
Struct * newNode (int input)
{

Struct node * temp = malloc ( -- // -- );
    temp -> data = input;
    temp -> right = NULL;
    temp -> left  = NULL;
```




```
    return temp;
}

Struct * insertNode ( Struct node * root, int x)
{
    if ( root == = NULL)
    {
Struct node * temp1 = newNode (x);
    return temp1;
```

OR

```
    {return newNode (x);}
}
else if ( x > root -> data )
{
// root -> right = insertNode ( root -> right, x);
}
else
{
root -> left = insertNode ( root -> left, x);
}
    return root;
```

LDR

PAGE No.
DATE

```
int main()
{
    newNode(100);
insertNode(root, 10);
  preorder(root);  //postorder(root);
}
```

DLR LRD

12th March 2020

PAGE No.
DATE / /

* Binary Tree:

In CS Binary tree is a tree DS {non-linear DS} in which each node has atmost 2 children which are referred to as the "left" & "right" child

→ Program to search element in an array:
OR Linear Search

arr

50	10	5	20	15
0	1	2	3	4

```
int main()
{ int arr[ ] = {50, 10, 5, 20, 15};
  int searchelement;
    int i;
cout << "Enter elt to be search :";
cin >> searchelement;
for (i=0; i< 5; i++
  while (arr[i]!= searchelement)
  {
    cout << "Element found" << i;
for (i=0; i<5; i++)
  {
    if (arr[i] == searchelement)
    {
    cout << "Element found at:" << i;
    }
  else
  { cout << "Element not found";
  } return 0; }
```

* Binary Search : [Binary Search Tree]

↓ when BS is for tree

Try :- BST ("insert") [recursion]

```
BST (struct node * root, int searchelement)
{
// to do
}
```

* Note :- It must be sorted.

arr | 50 | 10 | 5 | 20 | 60 |

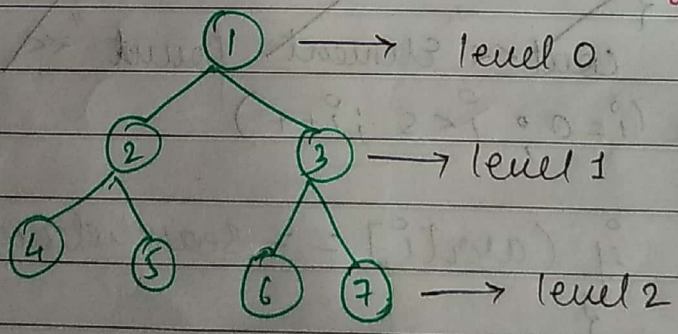
sort → sorting(algo)

arr | 5 | 10 | 20 | 50 | 60 |

* Breath First Search (BFS)

BFS is a tree traversing algorithm - It traverse the tree level by level (Depth by Depth).

eg



BFS → 1 2 3 4 5 6 7

PAGE No.
DATE / /

* 1. DFS

DFS explores a path all the way to a leaf before backtracking & exploring another part.

We can achieve this in 3 ways:

1] Preorder :-

D → Print the current nodes value.
L → Go to the left child & print it. Backtrack
R → Go to the right child & print it.

2] Inorder :

L → we go way down to the left child & print it first. Backtrack
D → and print it [left child root node]
R → and go way down to right child & print it.

3] Postorder :-

L → we go way down to the left child & print it first. Backtrack.
R → then we go way down to right child & print it. Backtrack.
D → and print it. [from right child to root & print root].

* Asymptotic Notation :

Complexity
- time (CPU)
- space (Ram + HD)

PAGE No.
DATE / /

→ Time Complexity

How efficient is an algo or piece of code.

Efficiency covers lots of resources including CPU uses, memory uses, disk uses, n/w uses all are the important but we will mostly talk about talk complexity (CPU uses).

→ Performance

How much time, memory, disk etc is actually used when a program is run. This depends on the m/c, compiler etc as well as the code.

→ Complexity

How do the resource requirement of a program or algorithm is scale (up & down)

For eg- what happens as the size of the problem being solved gets longer.?

Complexity affects performance but not the other way around.

$O(1)$
$O(N)$
$O(N^2)$

Ω (Best)

PAGE No.
DATE

Big - O (worst)

```
int main() → 1                    n
{                    1            n
cout<<"1";           1            n
cout<<"2";           1            n
cout<<"3";           1            n
cout<<"4";           1            n
Cout<<"5";           1            n
cout<<"6";           1            n
return 0;            1            n
}                    1            n
                    [7]  [10]     7n
```

1 operation = 1 unit

$7n = O(N)$

Constant; No need to write this

*N is constant

$O(1)$ → complexity

→ How to Calculate Complexity?

In general, how can you determine the running time of a code?

→ It depends on what kinds of statements are used

- Sequence of statements

```
Stmnt 1
stmnt 2
:
stmnt K
```

Total time = time (stmnt 1) + time (stmnt 2) + ... time (stmnt K)

The total time is found by adding the times for all stmnts.

If each stmnt is simple (only involves basic operations) then time for

each stmnt is constant & the total time is also constant.

Complexity $O(1) \rightarrow$ constant

- iii) if - else statements -

```
if (cond^n)
{
  Stmnt 1
  :
  Stmnt K
}
else
{
  Stmnt 1
  :
  Stmnt K
}}
```

O(1) [assume]

Total time = max(if Blk, else Blk)

O(N) [assume]

$= \max(O(1), O(N))$

$= O(N)$

- iii) Loops -

```
for (;N;)
{
  Stmnt 1;
  Stmnt 2;
  :
  Stmnt K;
}
```

→ 10 times, 100 times OR N times

limit the loop will continue.

Block 1 → 10 units

5 * 10 = 50 units

O(N)

size/limit of the loop

until the condition is satisfied the stmnts/Block will execute.

- iv) Nested for loops:

```
for (i=0; i<5; i++) → N
{
    stmnt          1
for (j=0; j<10; j++) → M
    {
        stmnt      1
    }
}
```

Complexity
It is also correct. → $= O((1*N)*(M*1))$
$= O(N*M)$
$= O(N*N)$
$= O(N^2)$

The outer loop execute 'N' times, every time the outer loop execute; the inner loop executes 'M' times. As a result the statements in the inner loop execute a total of (M*N) times.
Thus the complexity is order of
(M*N) (OR) $O(N^2)$

- v) if & loop:

```
if( )
{
cout<<"3";
cout<<"4";
}else
{
for(  )
{
cout<<" ";
}}
```

O(1)
O(N)
O(N)

```
if( )
{
cout<<" ";
}else
{
cout<<" ";
}
```

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

Recursive functions → $O(N^2)$

PAGE No.
DATE

ii) Loops & Recursive functions/methods :

```
void demo(int a, int b)
{
for( )
{                    → O(N)
cout<<"i";
}
}
```

```
int main()
{
demo(10,20); → O(N)  +
demo(30,40); → O(N)
O(2N)
O(N)
}
```

constant

Example

```
int main()
{

cout<<" ";                    1
for(i=0; i<5; i++)            5N
{for(int j=0; j<5; j++)       5N
  {
   cout<<"5";                 1
  }
}

for(int K=0; K<2; K++)        2N
{
  cout<<"K";
}
cout<<"All loops ended";

return 0;
}
```

$F(N) = 5n^2 + 2N + 2$

$G(N) = c * n^2$

$c * n^2$

$n = 4$

$= 36$

$n = 4$
$f(N) = 90$

$f(N) <= c * g(N)$

$F(N) = 5n^2 + 2n + n$

This will not affect the highest Power of it. Hence we can discard it.

→ $O(N^2)$ $\{ c > n \}$

Asymptotic Notation is the notation of the performance of an algorithm (program) in terms of just the i/p size n, where n is very large.

→ O Notation [Big-Oh]

It is the upper bound on the complexity of an algorithm. Hence it is used to denote the worst Behavior of an algorithm.

essentially this denotes the maximum-time for an algorithm no matter the i/p size.

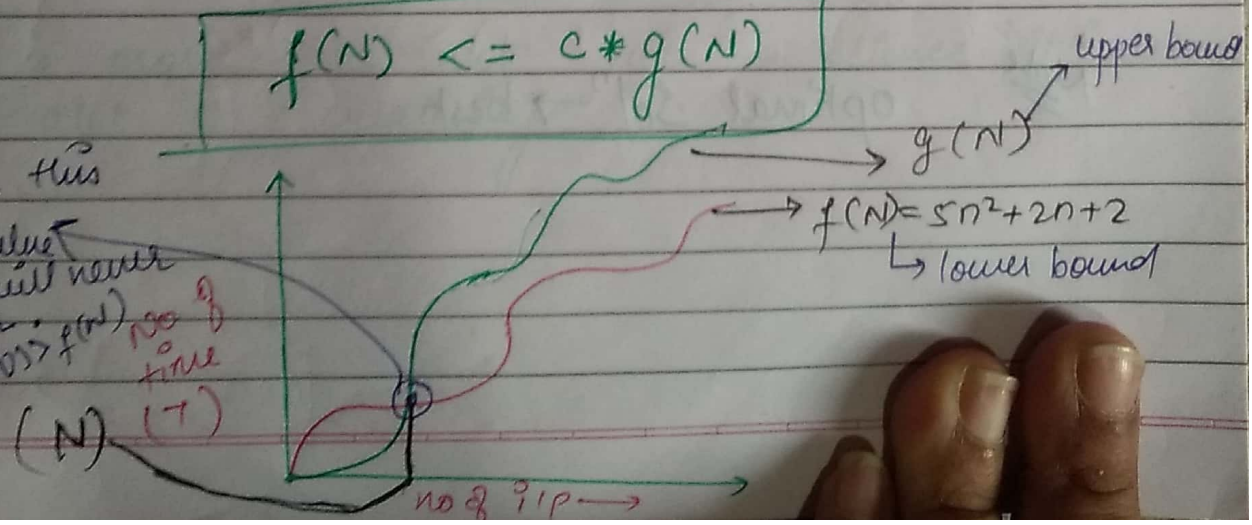
It is most widely used notation bcoz of its ease of analyzing an algorithm by learning about its worst behavior.

A funct^n $f(N)$ is order of $g(N)$ if for some constant 'C' & for values of "n"

$$f(N) = O(g(N))$$

greater than some value.

$$f(N) <= C * g(N)$$



13th March 2020

PAGE No.
DATE / /

* Greedy Algorithm :

→ Concept wise program

```
int i;
int t=6;
int sum=0;
int activity=0;
```

arr | 5 | 3 | 4 | 1 | 2 |
0 1 2 3 4

sort (Bubble)

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

```
for(i=0; i<=4; i++)
{
    activity++;
    sum=sum+arr[i];
    if(sum==t)
    {
        break;
    }
}
```

→ // cout << arr[i]; we just have to check where to place this statement.

```
cout << "Total count for activity performed:" <<
    activity;
```

* Note optimal Sol^n → best

A greedy algo. always makes the choice that seems to be the best at that moment.

→ How to decide which solution/choice is optimal?

Assume that you have an objective function that needs to be optimized (max, min) at a given point.

A greedy algo. makes greedy choice at each step to ensure that the objective function is optimized.

The Greedy algo has only one shot to compute the optimal soln so that it never goes back & reverse the decision.

→ Advantages

- It is very easy to come up with a greedy algo for a problem.
- The difficult part is that for greedy algo you have to work much harder to understand correctness issues. Even with the correct algorithm it is hard to prove why it is correct.

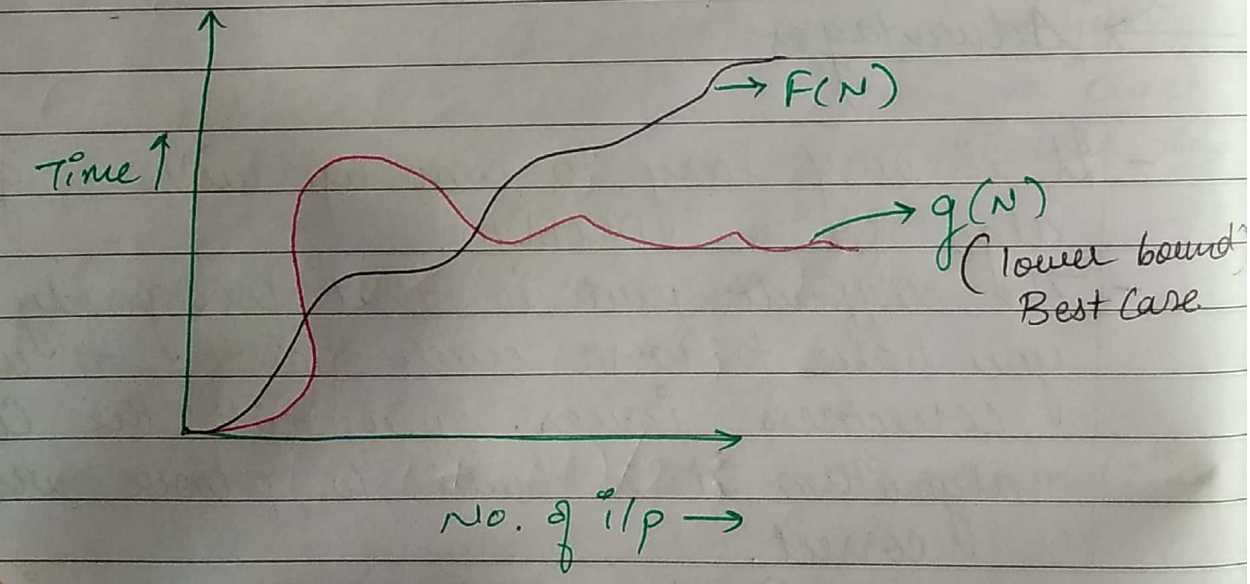
"Proving that greedy algo is correct is more of an art than a science. It involves a lot of creativity."

* Big Omega (Ω):

Ω is similar to Big-oh notation. The Ω notation is used to defined an asymptotic lower bound on the performance of an algorithm.

Hence this is used for representing the Best case Scenario.

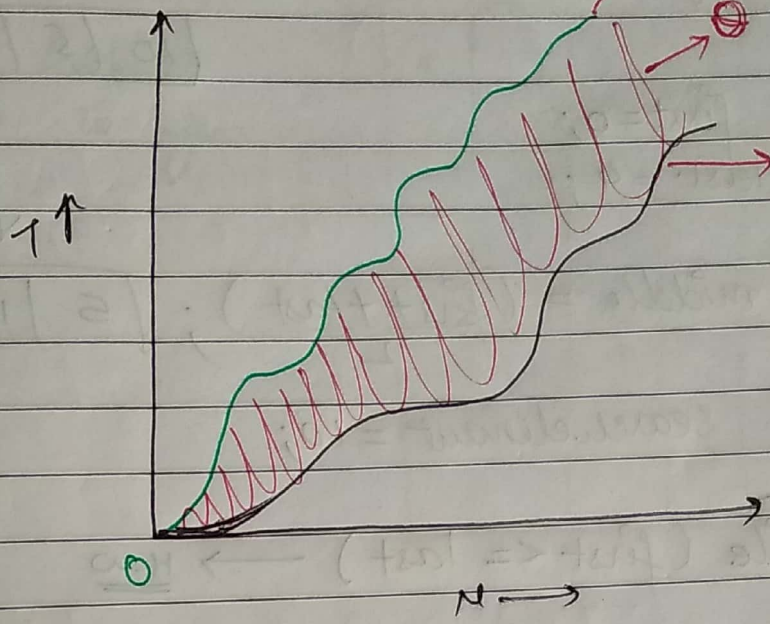
The Ω bound essentially means the minimum amount of time that our algorithm will take to execute. This notation is not often use ~~obtain~~ in practical scenario's since studying the best behavior cannot be a correct measure for comparison.



$\Omega \Rightarrow F(N) \geq g(N) * C$

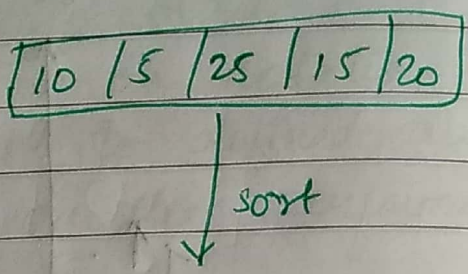
PAGE No.
DATE / /

* Θ (Average case)



PAGE No.
DATE / /

→ Binary Search without Recursion:

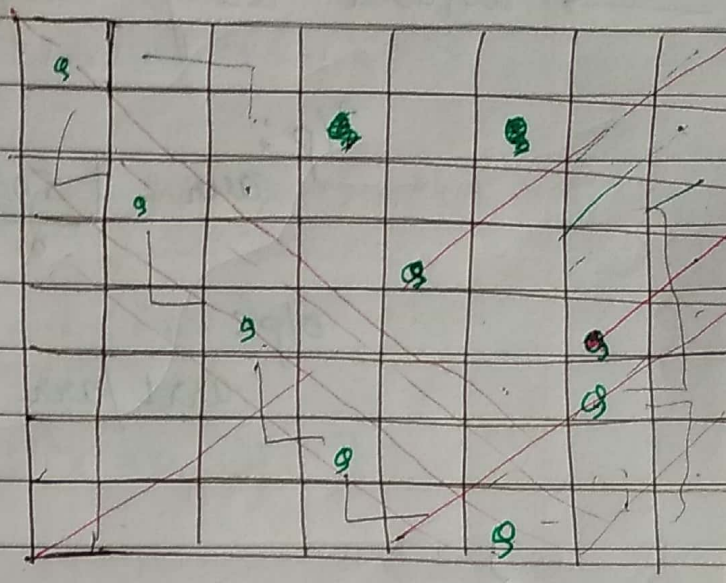



```
int first=0;
int last=4;
int middle = (first+last)/2;
int searchelement=10;
while (first<=last) ⟶ H.W
{
if (arr[middle]==searchelement)
    break;
if (arr[middle]>searchelement)
    last = middle-1;
if (arr[middle]<searchelement)
    first = middle+1;
  middle = (first+last)/2;
}
```

* Dynamic Programming

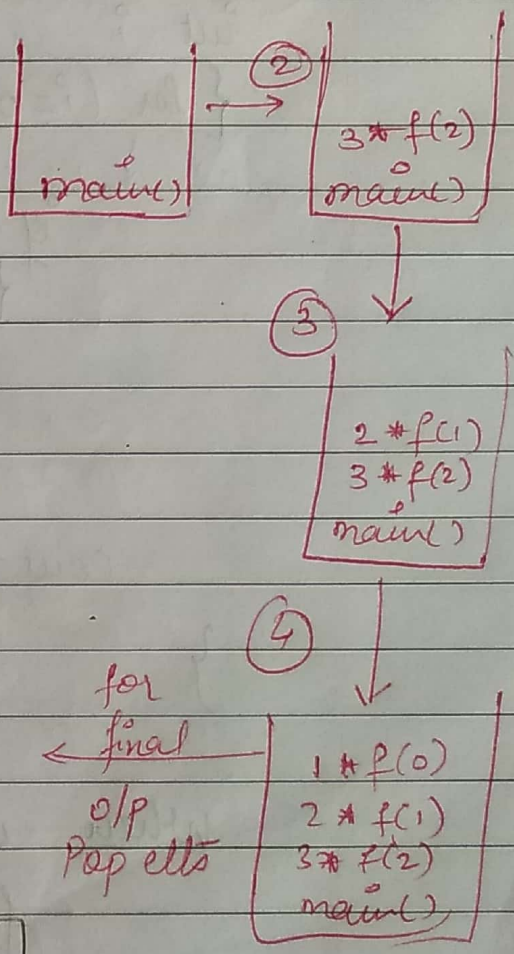
PAGE No.
DATE / /

→ 8-Queens Problem
we have to try



→ Recursion

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



f(1) → 1 * f(0) = 1
f(2) → 2 * 1 = 2
f(3) → 3 * 2 = 6

f(3) = 3*2*1 = 6

* Note
try fibonacci series using recursion.

SORTING

→ Without using any Sorting Algo.

i/p:

arr

50	10	30	20	40
0 i=	1 i+1	2	3	4

o/p:

arr1 / arr

10	20	30	40	50
0	1	2	3	4

```
{
int arr[] = {50, 10, 30, 20, 40};
int i
{ for (i=0; i<=4; i++)
    }
      if (arr[i] > arr[i+1])
      { int temp
        arr[i+1] = arr[i];
      }}}
cout << Sorted List << a[i];
}
```

→ `for(j=0; j<=4; j++)`

`temp = arr[i];`
`arr[i] = arr[i+1];`
`arr[i+1] = temp;`

When we will write this

```
for (i=0 ; i<=4; i++)
  { for (j=0; j<=4; j++)
    { if (      )
        ——— ;
        ——— ;
    }
```

It will only compare 2 elts inside 'j' loop & only j++.

PAGE No.
DATE / /

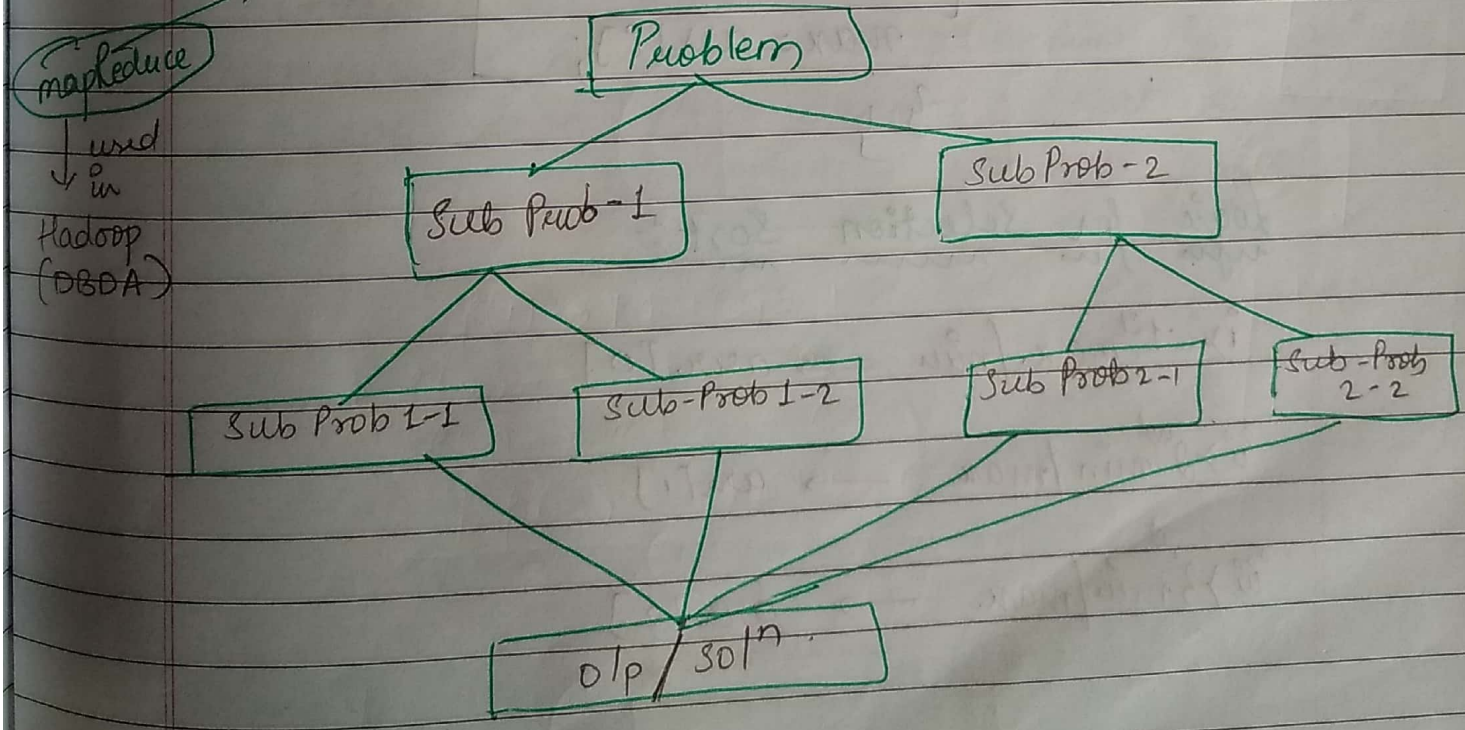
(1) Bubble Sort

Using recursion

$j <= i$

```
for (int i=0; i<=4; i++)
  {
    for (j=0; j<4; j++)        <-- j <= i
    {
      if (arr[j] > arr[j+1])
      {
        int temp;
        temp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = temp;
      }}}
```

→ Divide & Conquer [merge sort



→ Brute Force

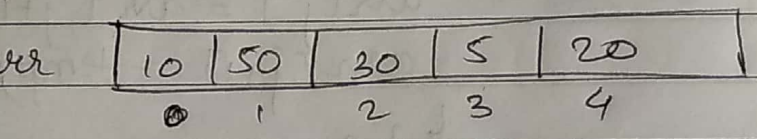
(trial & error)

② Insertion Sort

14th March 2020

③ Selection Sort

144




```
max=arr[0]int i;
    for (i=0; i<5; i++)
    {
        if (arr[i] > max)
        {
            max = arr[i];
        }}
```

max i

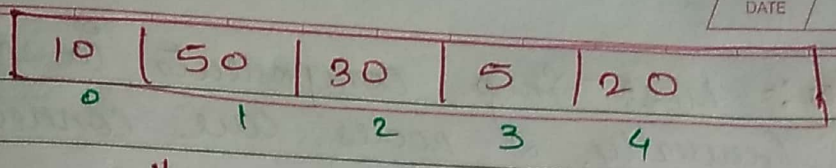
Logic for Selection Sort :-

i) 1st max/min → arr[0]

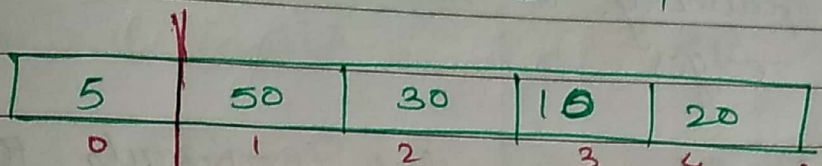
ii) 2nd min/max → arr[1]

iii) 3rd min/max → arr[2]

:



i)



new array. (now find min num. of this array)

ii) 5 10 30 50 20

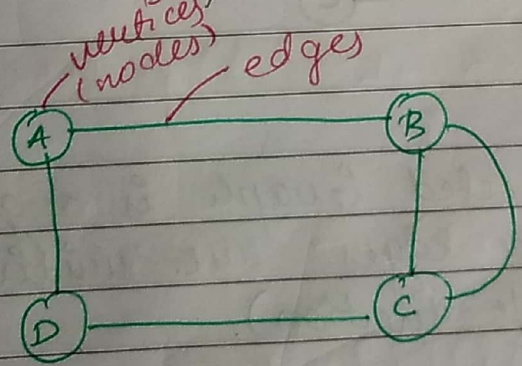
iii) 5 10 20 50 30

iv) 5 10 20 30 50

sorted array.

*GRAPH:

Graphs are mathematical structures that represent pair wise relationship betn objects. A graph is a flow structure that represents the relationship between various objects.



Graph has 2 basic components:

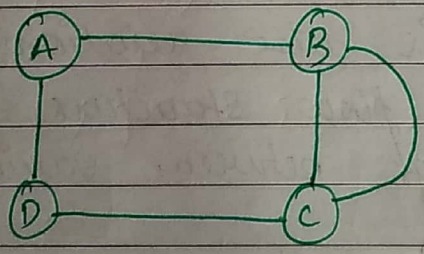
- **Nodes:-** Most imp. components in any graph. Generally 2 nodes are connected via edges.

- **Edges:-** Edges are the components that are used to represent the relationship bet' various nodes in a graph.
  - An edge bet' 2 nodes expresses a one way or 2 way relationship bet' the nodes.

→ Types of Graphs:

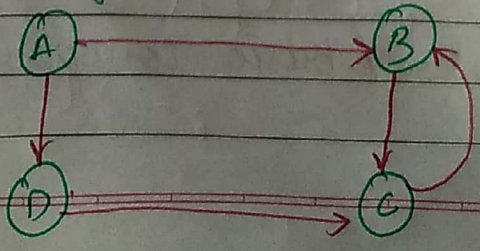
1. Undirected

An undirected graph is a graph. Graph in which all the edges are bidirectional (forward & Backward).



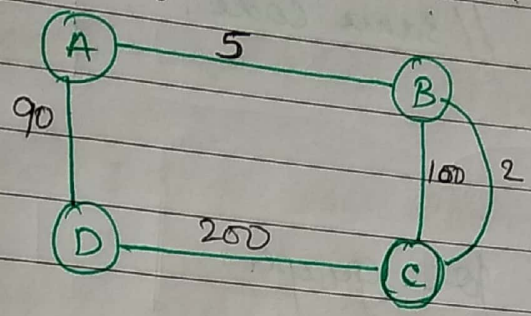
2. Directed

A directed Graph is a graph in which all the edges are unidirectional. (Single direction).



3. Weighted

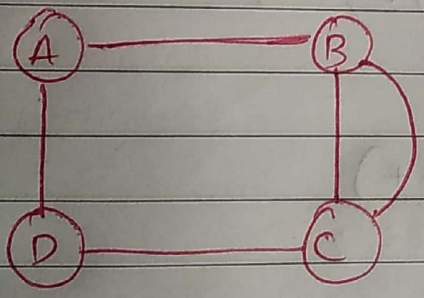
In a weighted graph each edge is assign a weight or cost.



→ Graph Representation:

You can represent a graph in many ways. The 2 most common ways of representing graph is as follows:

1. Adjacency Matrix



	0 A	1 B	2 C	3 D
0 A	0	1	0	1
1 B	1	0	1	0
2 C	0	1	0	1
3 D	1	0	1	0

```
int main()
{
  int input;
int arr[4][4] = {{0,0,0,0},{0,0,0,0},{0,0,0,0},
                 {0,0,0,0}};

cout << "Nodes are 1,2,3,4";
cout << " are 1 & 2 Connected :";
cin >> input;
```

```
if (input == 1)
{ arr[0][1] = 1;
  }
  :
  a[3][3]; // same code.
}
```

→ Basic code for graph.

```
int main()
{
  int input;
  int i, j;
int arr[][] = {{0,0,0,0}, {0,0,0,0}, {0,0,0,0},
               {0,0,0,0}};
cout << "Nodes are 1,2,3,4";
for (i=0; i<=3; i++)
 {
  for (j=0; j<=3; j++)
  }
cout << "Are" << i+1 << "&" << J+1 << "Connected";
cin >> input;
  if (input == 1)
    {
      arr[i][j] = 1;
    }}}
```

→ for o/p:

```
if (arr[i][j] == 1)
cout << i+1 << " & " << j+1 << " are connected";
```

cout << " 1 & 2 are Connected";

or

cout << "1 → 2 are connected";

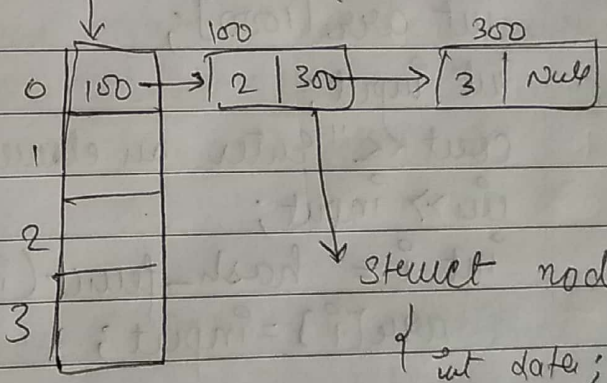
(2.) Adjacency List

1 & 2 connected
3 & 4 connected

steps to be followed:

i) array create
ii) array = NULL
iii) n1 = newnode()
n1 → data = 2
n1 → next = NULL;
arr[0] = n1;

struct *arr[4] = {NULL, Null, ...}



* HASHING

↳ O(1) [w.c]

```
int hash_funct^n (int search_element)
{
    index_no = search_element % prime no;
    eg
    =>  = 60 % 7 => [4]
                    ↳ at index 4 ; 60 is
                      Placed.
    return index_no;
}
```

→ Basic Program to demonstrate Hash function

```
int main()
{
int arr[1000];
int input;
cout << "Enter an element:";
cin >> input;
int i = hash_funct(input);
  arr[i] = input;

cout << "Enter an element:";
cin >> input;

i = hash_funct(input);
arr[i] = input;
```

```
int searchele ;
cout << " Enter a no, you want search :" ;
search_index = hash_funct ( search_ele ) ;
Cout << " Element found at " << search_index ;
return 0 ;
}
int hash_funct (int ele)
{
index_no = ele % 7 ;
return index_no ;
}
```

cin >> search_ele;

Hash funct^n Backbone stmnt. we can modify it with operators.

Hashing is a technique that is used to uniquely identify a specific object. (no, string, char etc) from a group of similar objects.

eg-

i) In universities each student is assigned a unique roll no. that can be used to retrieve info. about them.

ii) In libraries each book is assigned a unique no. that can be used to determine info about book. Such as its exact position in the library or the user it has been issued to.

Hashing can be implemented in 2 ways:

1) An element is converted into an integer by using a hash function. This element can be used as an index to store the original element.

2) The element is stored in the hash table where it can be quickly retrieved using hashed key.

→ **Hash Function**

A hash funct^n is any mathematical funct^n that can be used to map a dataset of an arbitrary size to a dataset of a fixed sized which falls into the hash table.

The value return by a hash funct^n are called hash values or hash codes / hash sums / hases.

→ **How to Create a good Hash function :-**

- Easy to compute, it should be easy to compute & must not become an algorithm in itself.
- Uniform distribution.
- Less Collison.

Collision occur when pair of elements are mapped to the same

hash value.

Irrespective of how good a hash funct^n is collision are bound to occur.

Therefore to maintain the performance of a hash-table it is important to manage collision through various collision resolution technique.

15th March 2020

* Solutions to Avoid Collision:

1. Linear Probing [open addressing or closed Hashing]

In open addressing all entries. are stored in the array when a new entry has to be inserted the hash index of the hashed value is computed & then the array is examined. If the slot at the ~~highest~~ hashed index is unoccupied that this entry record is inserted; else it proceeds in ~~some~~ some probe sequence until it find an unoccupied slot.

The probe ~~is~~ sequence is the sequence that is followed while traversing through entries.

eg

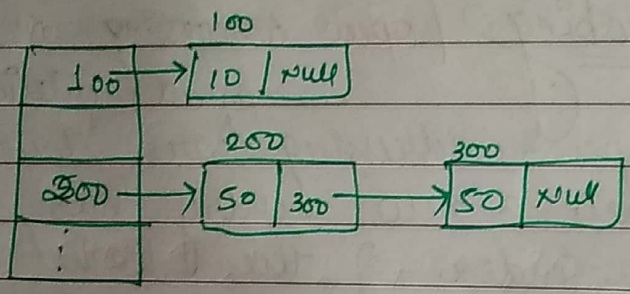
input_value % 7
(input_value +1) % 7
(input_value +2) % 7.

2. Separate Chaining: [open Hashing]

It is the one of the most commonly used as collision resolution technique.
It is usually implemented using linked list. To store an element in the hash table you must insert it into a specific linked list.

In the case of 2 different element have same hash value then store both the elements in the same linked list.

eg



3. Quadratic Probing:

It is similar to linear probing & only difference is the ~~trouble~~ interval bet successive probes or entry slots.

Here when the slot at a hashed index for an entry record is already occupied you must start traversing until you find an unoccupied slot.

The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original has indexed.

eg

(input_index + 1²) % 7
(—"— + 1³) % 7
(—"— + 1⁴) % 7
:

* Double Hashing

DH is similar to Linear probing & the only diff. is the interval betn successive probes. Here the interval betn probes is computed using 2 hash funct°.

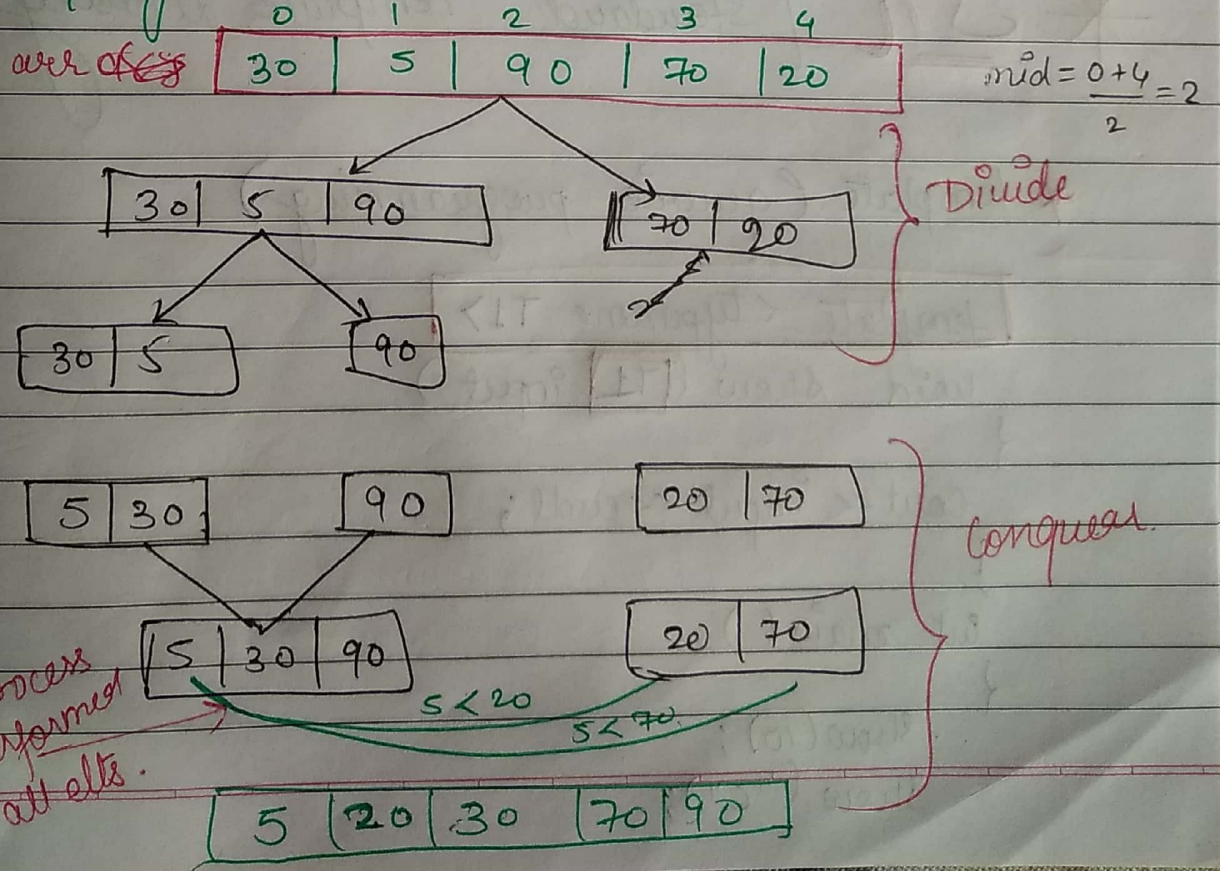
eg

(input_value) % 7 ⟶ hash_funct°-1
hash (input_value + 1) % 7) ⟶ hash_funct°-2

Sorting

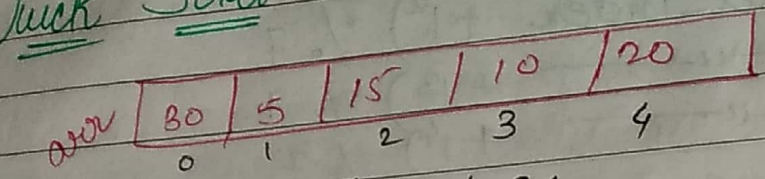
—xox—

(4) Merge Sort :



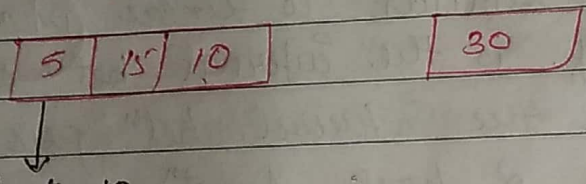
DATE

(5) Quick Sort :



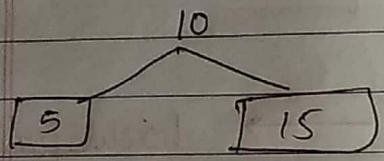
pivot = first / last / mid
pivot = last = 20

left < (20) < right



pivot = 10

30.



→ ~~5 10~~ 5, 10, 15, 20, 30

* STL [Standard Template Library] (Collection) Java

* template (generic programming)

```
template <typename T1>
void show (T1 input)
{
  cout << input << endl;
}
int main()
{
  show(10);
  show("C");
```

```
show ("CDAC");
show (10.5);

return 0;
}
```

```
void show (int input)
{
 cout << input;
}

void show (char input)
{
cout << input;
}

void show (string input)
{
cout << string;
}
```

instead of writing this many function we can write a generic program using template keyword.
i.e to overcome this problem we use template (generic Programming).

DATE / /

* Sorting Algo. — When to use?

1. Bubble → i) The complexity of the code doesn't matter.
   ii) The short code is preferred.

2. Selection → i) when small list is to be sorted. time
   ii) Cost of swapping doesn't matter iii) when checking of all elts is compulsory.

3. Insertion → i) The array has a small no. of elements.
   ii) There are only a few elements left to be sorted.

4. Quick → i) when the programming language is good for recursion.
   ii) Time & space complexity matters.

* STL

→ **Templates :-**

A template is a C++ entity that defines one of the following :-

i) A family of classes → Class template
ii) A family of functions → Func^{tn} templates
iii) A family of variables → Variable template

Containers :-

The containers library is a generic collection of class templates & algorithms that allow programmers to easily implement common DS like stack, list & queue, tree etc.

There are 3 class of containers :-

(1) Sequence (2) Associative (3) Unorder Associative

Each of is designed to support a different set of operations. The container manages the storage space that is allocated for its element & provide member function to access them.

(1) Sequence Containers

SC implement DS which can be accessed sequentially.

i) array - static contiguous array. (C++ 11)

ii) vector - dynamic array. (legacy)

iii) deque - double ended queue.

iv) list - DLL (legacy)

v) forward_list - singly linked list. (C++ 11)

(2) Associative Containers

AC implemented sorted DS that can be quickly searched. (order of logN complexity)

Sorted

i) set - Collection of unique keys, sorted by keys.
ii) map - Collection of key-value pairs sorted by keys & keys are unique.

→ Container Adaptors

eg -
Container adaptors provide a different interface for sequential containers.

i) Stack ii) Priority Queue iii) Queue

→

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
  deque <int> d1;

  d1.push_front(10);
  d1.push_back(20);

  for (int n:d1)  } c++ 11
  {
    cout << n; }

  return 0;
    }
```

PAGE No.
DATE / /

```
int arr[] = {10, 20, 30, 40};
for (int mama : arr)
{
cout << mama << endl;
}
```

O/p: 10
20
30
40