Day-1 16/6(Introduction)

- It is used in designing

- It is abstract data type

- It has some set of rules that we cannot change

- We uses DS for abstract data type

- What is data structure?

Defination -> working -> real life example

- In computer terms a ds is a specific way to store and organize data in computers memory so that these data can be used efficiently later

- It belong to structuring the data

- Need of Data Structure

    - Effiency

    - Reusability

    - Abstraction

- There is two types of DS

    - Linear

    - Non-Linearas

- Linear

    - Array

    - Queue

    - Stack

    - Linked list

- Non Linear

    - Graph

    - Tree

***

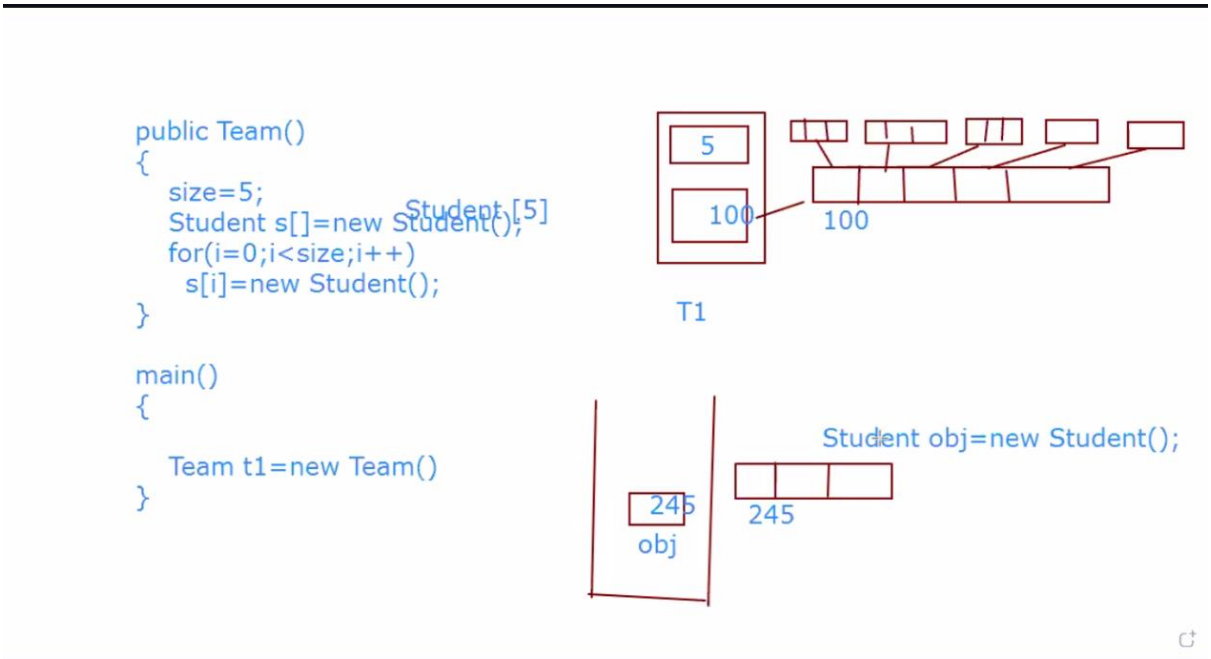
***

Day-2 17/6(Array,Linkedlist)

Array

```java
class Student
{
	private int rollno;
	private string name;
	private int marks;

	public Student()
	{
	}

	public Student(int rollno, String name, int marks)
	{
		this.name=name;
		this.rollno=rollno;
		this.name=name;
	}

	public void accept()
	{
```

```
        Scanner sc =new Scanner(System.in);
        Syso(name,rollno,marks);
        sc=name,rollno,marks;
    }

    public void display()
    {
        System.out.println(name,rollno,name);
    }
}
```



![array](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/1_array.png)

```java
class Team
{
```

```
	private int size;
	Student s[];

	public Team(int size)
	{
		this.size=size;
		s=new Student[size];
		for(int i=0;i<size;i++)
			s[i]=new Student();
	}

	public void accept()
	{
		for(int i=0;i<size;i++)
			s[i].accept();
	}


	public void display()
	{
		for(int i=0;i<size;i++)
			s[i].display();
	}

}

```

```java

class Program
{
	public ststic void main(String [] arr)
	{
		Team cri = new Team(11);
		cri.accept();
		cri.display();



		Team kabadi = new Team(7);
		kabadi.accept();
		kabadi.display();
	}

}

```

Linked List

- It is Linear Data Structure
- It connected via links

- Disadventage of Array
	- Slow searching in array so linked list come
	- Inseration and deletion is slow in array

- fixed size in array

- memory weastage in array

- Because of this disadvantages of array linked list is come in picture

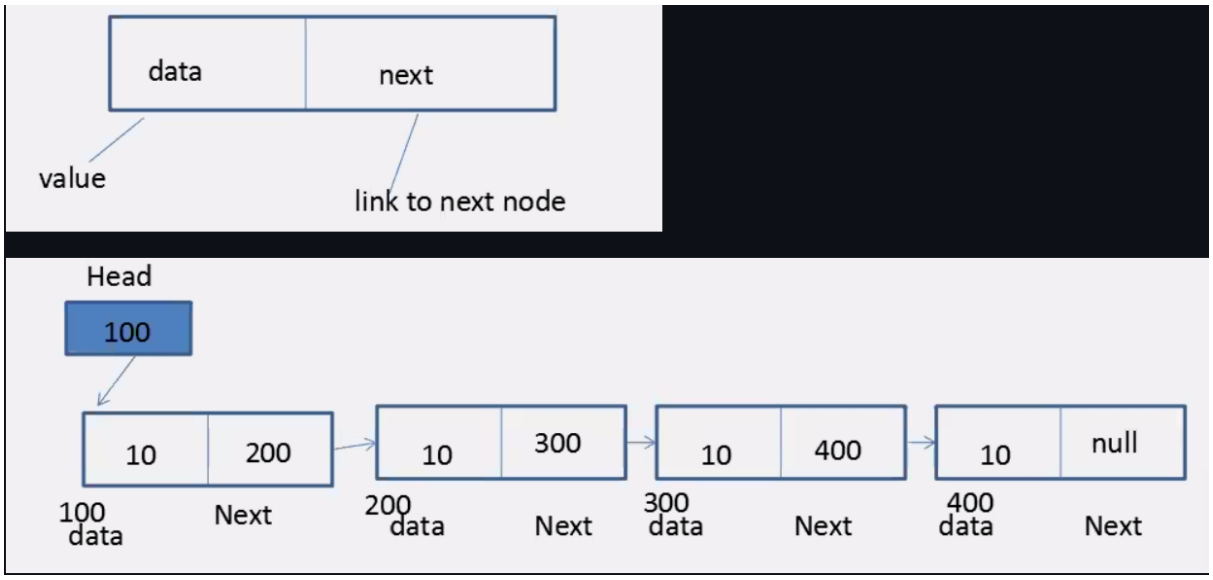
- In other hand linked list able to grow in size as needed

- Does not required the shifting of items during inseration or deletion.

- Each data value is linked with address of next value

- Each element is called node

- Data values need not be stored in adjacent memory cell



![linkedList](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/2_linkedList.png)

![linkedList](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/3_linkedList.png)

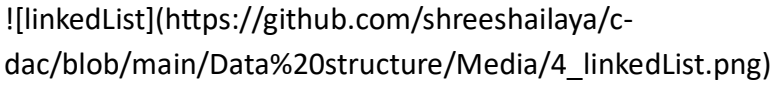
- Structure of Linked List

```java

class Node
```

```
{
        private int data;
        private Node next;      //Next  is data member which will hold the referance of next
variable

        // One linked list may have multiple node
        public Node()
        {
                data=0;
                next=null;
        }

        public void display()
        {
                syso(data);
        }
}
class Linkedlist
{
        Node head;
        createNode(int data);
        createLinkedList(int numberOfNodes);
}

class Program()
{
        PSVM( )
        {
```

```
		Linkedlist l1= new Linkedlist();
		l1.createLinklist(5);
		l1.display();
	}
}
```

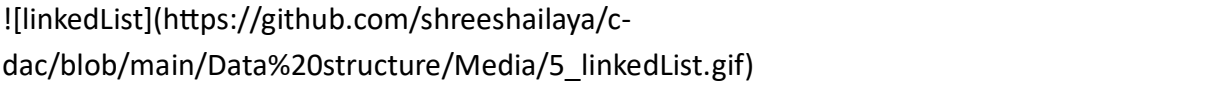
- Linked list real program
- link = next;




```java
public class Linkedlist
{
	private Node head;
	public Linkledlist()
	{
		head=null;

	}

	public void createLinkedlist(int terms)
	{
		int data =5;
```

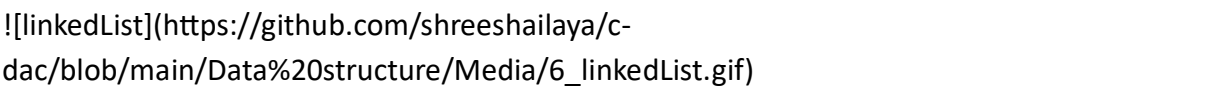
```
        int i;
        node newnode=null,move=null;
        for(int i=1;i<=term;i++)
        {
                newnode=new Node(data); //create new node
                if(head==null)
                        head=newnode;
                else
                {
                        move=head;
                        while(move.getNext())!=null)
                        {
                                move=move.getNext();
                        }
                        move.setLink(newnode);
                }
                data=data+5;
        }

}

public void display()
{
        Node move=head;
        while(move!=null)
        {
                System.out.println("  "+move.getData());
                move=move.getLink();
```

```
                }
        }
}
```



- Inseration in linked list

```
move.setLink(newnode);
```



- Steps for creating Linked List
    - Ask user number of nodes to insert
    - Create node first
    - First check whether the linklist is empty or not, if linklist is empty this node will be first node so assign this node as head
    - Otherwise travel the linkedlist till the end and attach node at end.
    - Repeat these steps till number of nodes user entered

Assignments and code for [Array](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/17juneDay2/assignment1.java) and [LinkedList](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/17juneDay2/assignment2.java)

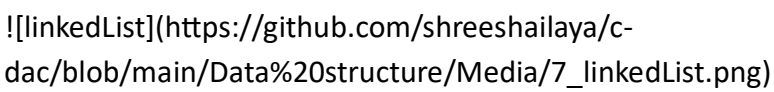
***

***

Day3 18/6(Operations on Linkedlist)

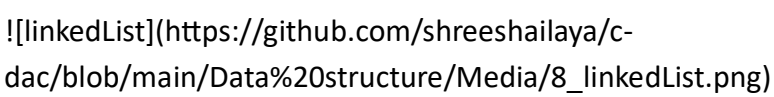
Operations on Linkedlist

- Add begining

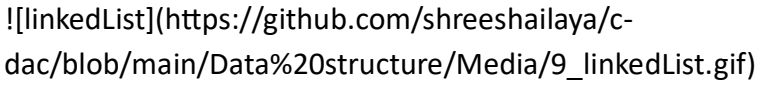


(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/7_linkedList.png)

- Add mid



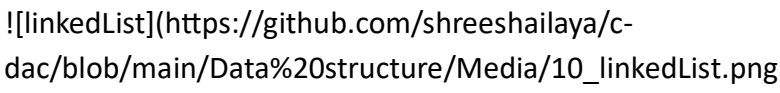
(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/8_linkedList.png)



(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/9_linkedList.gif)

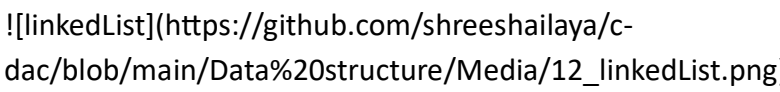
- Delete first node

- Note Head is 200 in below img



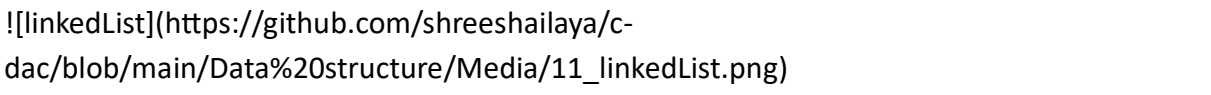
(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/10_linkedList.png)

- Delete Mid Node



(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/12_linkedList.png)

- Delete last node



***

***

Day 4 19/6(Searching algorithms)

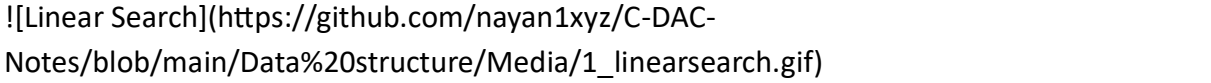
- Searching algorithms

Linear Search

- In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

- It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.

- Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.



Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudo code for linear search

```
LinearSearch(list, target_element):
{
    INITIALIZE index = 0
    WHILE (index < number of items in the list)
    {
        IF (list[index] == target element)
        {
            RETURN index
        }
        INCREMENT index by 1
    }
    RETURN -1
}
```

Code for Linear Search

```cpp
#include < iostream >
 using namespace std;

void linearSearch(int a[], int n) {
 int temp = -1;

 for (int i = 0; i < 5; i++) {
  if (a[i] == n) {
```

```
      cout << "Element found at position: " << i + 1 << endl;
      temp = 0;
      break;
    }
  }

  if (temp == -1) {
    cout << "No Element Found" << endl;
  }

}

int main() {
  int arr[5];
  cout << "Please enter 5 elements of the Array" << endl;
  for (int i = 0; i < 5; i++) {
    cin >> arr[i];
  }
  cout << "Please enter an element to search" << endl;
  int num;
  cin >> num;

  linearSearch(arr, num);

  return 0;
}
```

Time and Space Complexity

- Best Time Complexity: O(1)

- Average Time Complexity: O(n)

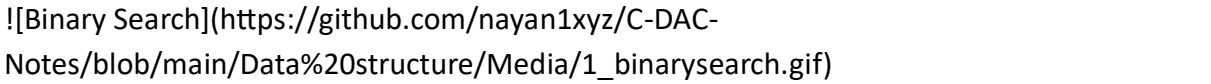
- Worst Time Complexity: O(n)

- Best Space Complexity: O(1)

Binary Search

- Binary search begins by comparing the middle element of the list with the target element. If the target value matches the middle element, its position in the list is returned. If it does not match, the list is divided into two halves.

- The first half consists of the first element to middle element whereas the second half consists of the element next to the middle element to the last element. If target value is greater than middle element, first half is discarded. Then same steps follow on until we find the target value's position.



Algorithm

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```

Pseudo code for binary search

```cpp
BinarySearch(array, target):
{
   left = lowestBound(array)
   right = highestBound(array)
   WHILE (left <= right)
   {
      middle = (left + right) / 2
      if(target = array[middle])
         RETURN middle
      else if(target < array[middle])
        right = middle - 1
      else
        left = middle + 1
   }
   RETURN -1
}
```

C++ Code for Binary Search

```cpp
#include <iostream>
using namespace std;

int binarySearch(int[], int, int, int);

int main()
```

```
{
  int num[10] = {10, 22, 37, 55, 92, 118};
  int search_num, loc=-1;

  cout<<"Enter the number that you want to search: ";
  cin>>search_num;

  loc = binarySearch(num, 0, 6, search_num);

  if(loc != -1)
  {
    cout<<search_num<<" found in the array at the location: "<<loc;
  }
  else
  {
    cout<<"Element not found";
  }
  return 0;
}

int binarySearch(int a[], int first, int last, int search_num)
{
  int middle;
  if(last >= first)
  {
    middle = (first + last)/2;
    //Checking if the element is present at middle loc
    if(a[middle] == search_num)
```

```
    {
      return middle+1;
    }

    //Checking if the search element is present in greater half
    else if(a[middle] < search_num)
    {
      return binarySearch(a,middle+1,last,search_num);
    }

    //Checking if the search element is present in lower half
    else
    {
      return binarySearch(a,first,middle-1,search_num);
    }

  }
  return -1;
}
```

Time and Space Complexities

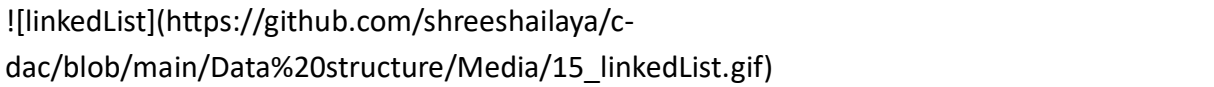
- Best Time Complexity: O(1)
- Average Time Complexity: O(log n)
- Worst Time Complexity: O(log n)
- Best Space Complexity: O(1)

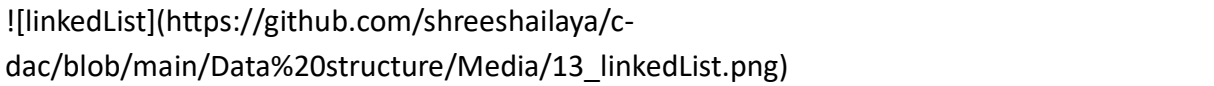
***

***

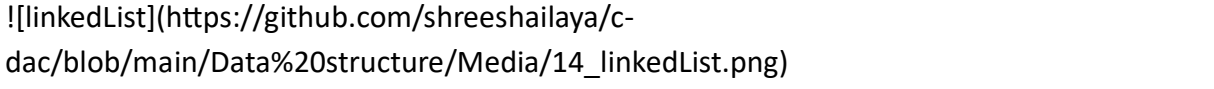
Day 5 21/6(Sorting LinkedList,Doubly LinkedList)

Sorting LinkedList





Doubly LinkedList



***

***

Day 6 22/6(Stack,Dynamaic Stack,Reverse String using Stack)

Stack

- Some functions in stact

- peak()

- isFull()

- push

- pop

- push()

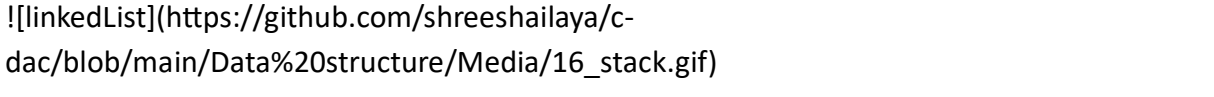
1. Check stack is full.

2. if the stack is full produce an error and exit

3. is the stack is not full increment top to point next empty space

4. adds data element to the stack locationwhere top is pointing

5. return success



- pop()

1. Checks if the stack is empty
2. if the stack is empty produce error and exit
3. if the stack is not empty access the data element at which top is pointing
4. decrease the value of top by 1
5. return success

```java
class MyStack
{
private int top,size;
private int arr[];

	public MyStack()
	{
		size=3;
		top=-1;
		arr=new int[size];
	}

	public MyStack(int size)
	{
		this.size=size;
		top=-1;
		arr=new int[size];
```

```
}

public boolean isEmpty()
{
	if(top==-1)
	{
		return true;
	}
	else
	{
		return false;
	}
}

public boolean isFull()
{
	if(top==size-1)
		return true;
	else
		return false;
}


public void push(int element)
{
	if(isFull==true)
	{
		syso("Stack is full");
```

```
                return;
        }
        else
        {
                top++;  //increment top
                arr[top]=element;
        }
}

// it is mandatery that we have to pass paramert to push

public int pop()
{
        int element=-9999;
        if(isEmpty()==false)
        {
                element= arr[top];
                top--;
        }else
        {
                syso("Stack is empty")
        }
        return element;
}

public void display()
{
        syso("*** Stack is ***");
```

```
                for(int i=top;i>=0;i--)
                {
                        syso("   "arr[i]);
                }
        }
}

```

- Push take parameter
- pop return parameter
- so pop always return something

Dynamaic Stack (LikeList stack)

- Create Node class
    - data and next
    - default constructor
    - para constructor
    - getter setter
    - toString/display()
- Create MyClass
    - having top as Node data type
    - default Constructor-assign null to top
    - implement is empty method which will return status whether stack is empty or not

    if top is null then stack is empty
    - push method will take data as parameter

    ```
    ```

create newnode

if top isnull

newnode become top

else

add newnode at beg

assign top as new node

```

- pop()
- check is stack is not empty
- mark first node as del
- move top to next node
- get del.data into data
- assign del is null

if empty stack return 99999

- display()
- assign move=top
- displaymove.data
- increment or shift move to next node.
- repeat till move !=null;

```java

class Node

{

private char data;

private Node next;

```
	publuc Node()
	{
		data="0";
		next=null;
	}
	public Node(char ch)
	{
		this.data=ch;
		next=null;
	}

	public String toString()
	{
		return " "+data;
	}

	//gettersetter
}

class MyStack
{
	Node top;
	public MyStack()
	{
		top=null;
	}
	public boolean isEmpty()
	{
```

```
        if(top==null)
        {
                return true;
        }else{
                return false;
        }
}

public void push(char ch)
{
        Node newnode=new Node(ch);
        if(top==null)
        {
                top=newnode;
        }
        else
        {
                newnode.setNext(top);
                top=newnode;
        }
}

public char pop()
{
        char data='0';
        Node del=null;
        if(isEmpty()==false)
        {
```

```
                del=top;
                data=del.getData();
                top=top.getNext();
                del=null;
            }

            return data;
        }

        public String toString()
        {
            String str="";
            Node move=top;
            while(move!=null)
            {
                str=str+"\n\t"+move.getData();
                move=move.getNext();
                return str;
            }


        }
}
```

Reverse String using Stack

***

***

Day 7 23/6(C2 Stack,Queue)

C2 Stack

```java

class MyStack
{
	private int top1,top2,size;
	private int [] arr;

	public MyStack()
	{

	}

	public boolean isFull()
	{
		if(top1==top2-1)
			return true
	}

	public boolean isEmpty()
	{
		if(top1==-1 && top2==size)
			return true;
```

```
}

public void pushFront(int data)
{
	if(isFull())
	{
		syso("Stac is full")
	}else
	{
		top1++;
		arr[top]=data;
	}
}

public void pushBack(int data)
{
	if(isFull)
	{
		syso("Stack is full")
	}else
	{
		top2--;
		arr[top2]=data;
	}
}

public void display()
{
```

```
		/*
		int i;
		for(i=top1;top1!=top2;i=(i-1)%size)
		{
			syso("  "+arr[i]);
		}

		double i =0,size =5;
i = (int) ((i-1)-(size*Math.floor((i-1)/size)));
		*/


		for(i=top1;i>=0;i--)
			syso("  "+arr[i]);
		for(i=size-1;i>=top2;i--)
			syso("  "+arr[i]);
	}
}
```

Queue

- there is two element i.e front and rear
- there is enqueue=insert and dequeue=retrive
- Insert element
	- check whether queue is full or not
	- increment rear and insert data rear position

- delete element
    - check whether queue is empty or not
    - increment front and return data avalible at front location

- isEMPTY()

```
public boolean is Empty()
{
    if(rear==front)
        if(rear==front)
        return true;
}
```

- isFull()

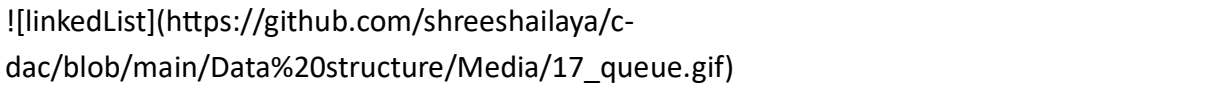
```
public boolean isFull()
{
    if(rear=size-1)
        return true;
}
```

***

***

Day 8 24/6(Queue,Dynamic Queue,Circular Queue)

Queue



class MyClassMyQueue

{

- rear(to insert data)

- front (to print exit)

- size(size of queue())

- array

}

public void enQueue(int data)

{

- check queue is not full

    - increment rear and add data at rear position

- print queue is full

}

public int deQueue()

{

- check queue is not empty

    - increment front and return data avalible at front position

-return -9999

}

public int isFull()

{

- ifrear reach to size-1

}

public int isEmpty()

{

- when front and rear pointing to same index

-

}

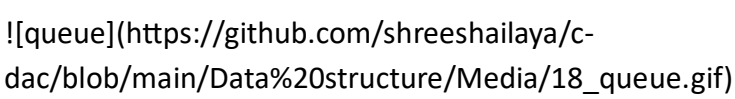
public void display()

{

- from front+1 to rear display all array elements

}

Dynamic Queue




```java

class Node

{

	Book data;
```

```
	Node next;

	public Node()
	{
		data=null;
		next=null;
	}
	public Node(Book data)
	{
		this.data=data;
		this.next=null;
	}

	public String toString()
	{
		return data.toString();
	}

	//getter and setter
}

class QueueLinkedList
{
	Node rear,front;
	public QueueLinkedList()
	{
		rear=null;
		front=null;
```

```
}

public boolean isEmpty()
{
	if(front==null)
	{
		rear=null;
		return true;
	}else
	{
		retur false;
	}
}

public void enQueue( Book data)
{
	Node newnode = new Node(data);
	if(rear == null)
	{
		rear=front=newnode;
	}else
	{
		rear.setNext(newnode);
		rear=newnode;
	}
}

public Book deQueue()
```

```
{
        Book data=null;
        if(isEmpty())
        {
                return data;
        }
        else
        {
                Node del;
                del=front;
                front=front.getNext();
                data=del.getData();
                del=null;
        }
        return data;
}


public String toString()
{
        String str="";
        Node move;
        for(move=front;move!=null;move=move.getNext())
        {
                str=str+" "move.getData().toString();
        }
        return str;
}
```

```
}

class Book
{
	int bookid;
	String bookname;

	public Book()
	{
		bookid=0;
		bookname=null;
	}
	public Book(int bookid, String bookname)
	{
		this.bookid=bookid;
		this.bookname=bookname;
	}

	// getter setters

	public String toString()
	{
		return "  "+ bookid "and "+ bookname;
	}
}
```

```

### Circular Queue

Enqueue in circular Queue

- check queue is not full

  -- if it is first element in queue

     increment rear and front by one

     add data ar rear position

  --- change rear and add element in array at rear position

     rear=(rear+1)%size;

-- Print Queue is full

DeQueue element from queue

--- check queue is not empty

  --return value at front position

  -- change front to next index.

Display circular Queue

  -- for(int i=front; i!=rear ; i= (i+1)%size)

       print arr[i]

  print last element

front =2
```

rear=1

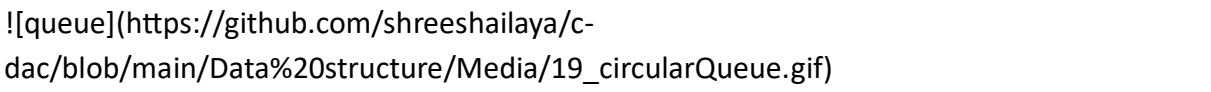
i=2 30

i=3 40

i=4 50

i=0 60

i=1 X



***

***

Day 9 25/6(Queue Types)

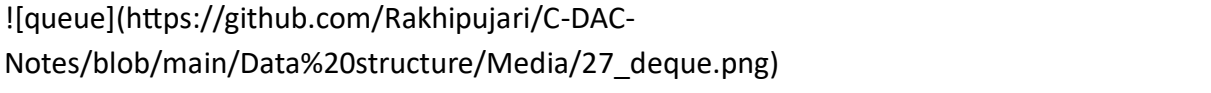
Deque

- Stands for Double Ended Queue.

- Insertion and Deletion operations can be performed from both ends.

- can be used as both stack and queue.

Representation of Deque



- Operations on Deque:

1. push_front()
2. push_back()
3. pop_front()
4. pop_back()
5. front()
6. back()

- Implementation of Deque with the above operations using doubly linked list:

```java
class Node {
    int data;
    Node prev, next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}

class Deque {
    private Node front, rear;

    public Deque() {
        this.front = null;
        this.rear = null;
```

```
}

public void push_front(int data) {
    Node newNode = new Node(data);
    if (front == null) {
        front = rear = newNode;
    } else {
        newNode.next = front;
        front.prev = newNode;
        front = newNode;
    }
}

public void push_back(int data) {
    Node newNode = new Node(data);
    if (back == null) {
        front = rear = newNode;
    } else {
        newNode.prev = rear;
        back.next = newNode;
        rear = newNode;
    }
}

public int pop_front() {
    if (front == null) {
        System.out.println("Deque is Empty!");
    }
```

```
    int data = front.data;
    front = front.next;
    if (front != null) {
        front.prev = null;
    } else {
        rear = null;
    }
    return data;
}

public int pop_back() {
    if (rear == null) {
        System.out.println("Deque is Empty!");
    }
    int data = rear.data;
    rear = rear.prev;
    if (rear != null) {
        rear.next = null;
    } else {
        front = null;
    }
    return data;
}

public int front() {
    if (front == null) {
      System.out.println("Deque is Empty!");
    }
```

```
        return front.data;
    }

    public int back() {
        if (rear == null) {
            System.out.println("Deque is Empty!");
        }
        return rear.data;
    }
}
```

- Applications of Deque:

1. Job scheduling algorithm
2. Undo Redo Functionality

***

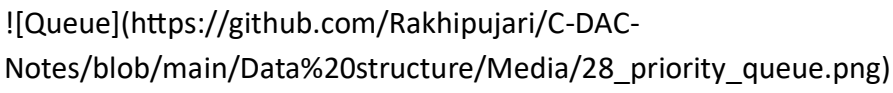
***

Day 10 26/6(Queue Types)

Priority Queue

- Every element in the queue has some priority associated with it.
- the element with highest priority will be the first element.
- it supports only comparable elements and are arranged in some order.

- For example consider the below Ascending order priority queue



- Operations of Priority queue

1. Insertion

2. Deletion

3. Peek

- Implementation of Priority queue using linkedList

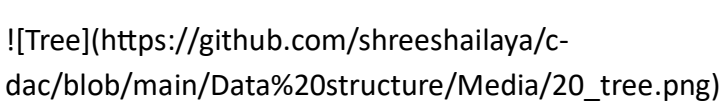
```java

class priorQueue

{

static class Node {

   int data;

   int priority;

   Node next;

}

static Node node = new Node();

static Node newNode(int d, int p)

{

   Node temp = new Node();

   temp.data = d;

   temp.priority = p;
```

```
    temp.next = null;

    return temp;
}

static int peek(Node head)
{
    return (head).data;
}

static Node pop(Node head)
{
    Node temp = head;
    (head) = (head).next;
    return head;
}

static Node push(Node head, int d, int p)
{
    Node start = (head);

    Node temp = newNode(d, p);

    if ((head).priority < p) {

        temp.next = head;
        (head) = temp;
    }
```

```
    else {
        while (start.next != null &&
            start.next.priority > p) {
            start = start.next;
        }
        temp.next = start.next;
        start.next = temp;
    }
    return head;
}

static int isEmpty(Node head)
{
    return ((head) == null)?1:0;
}

public static void main(String args[])
{

    Node pq = newNode(10, 0);
    pq =push(pq, 2, 2);
    pq =push(pq, 5, 3);
    pq =push(pq, 4, 1);

    while (isEmpty(pq)==0) {
        System.out.printf("%d ", peek(pq));
        pq=pop(pq);
    }
```

}

}

```

- Applications of Priority queue

1. Used in Dijkstra's shortest path and prim's algorithm.

2. for priority based scheduling in operating system.

***

***

# Day 11 28/6(tree,AVL Tree,Binary tree types)

### tree




- Binary tree has max 2 node

- General tree has as many node as you want

- In binary search tree left must be less than root and right is greater than root

### AVL Tree
```

- Balanced Tree

Binary tree types

Full Binary tree

- All Internal node have must 0 or 2 nodes

Complete binary tree

- Nodes are left to right only
- Complesary need left node

Perfect binary

- All the nodes at the same level
- Every perfect tree is complete

- Implement binary tree

- Create Node Class
    - data
    - left
    - right
- class BinaryTree
{
    root
    void addNode(int data)
    {
        - create node with data
        - check if root is null
            - root is newnode
        - assign move as root
        - ask where you want to attach node

-left

= Check if left is avalible(null)

attach at left of move

break;

- if left is not empty

- check if right pointer is null

- attach node at right of move

- break;

}

}

```java
class Node
{
private int data;
private Node left,right;

public Npde ()
{
data=0;
left=rigit=nuu;
}

public Node(int data)
{
data=data
```

```
        }
}


class BinaryTree
{
private Node root;
        public Tree()
        {

        root=null;

        }
        get set root

        public void addNode()
        {
                Node newnode=new Node(data);
                Node move;
                char ans;
                Scanner sc = new Scanner();
                if(root==null)
                {
                  root=newnode;
                  syso("ROOT IS CREATED");
                }else
                {
                  move = root;
```

```
while(true)
{
        syso("LEFT or RIGHT of"+move.getData());
        ans=sc.next().charAt(0);
        if(ans=='l' || ans=='L')
        {
         if(move.getLeft()==null)
         {
                move.setLeft((nenode)
                break;
         }else
         {
                move=move.getLeft();
         }
        }else if(ans==r)
        {
         if (move.getRight()==null)
         {
                move.serRight(newnode);
                break
         }else
         {
                move=move.getRight();
         }
        }else
        {
                syso("Wrong option");
                break;
```

```
                }
            }
        }

        void inOrder(Node root)
        {
                Node move=root;

        }
    }


}
```

Day 12 29/6(Inorder,Preorder,Postorder,Search Element from tree,Graph)

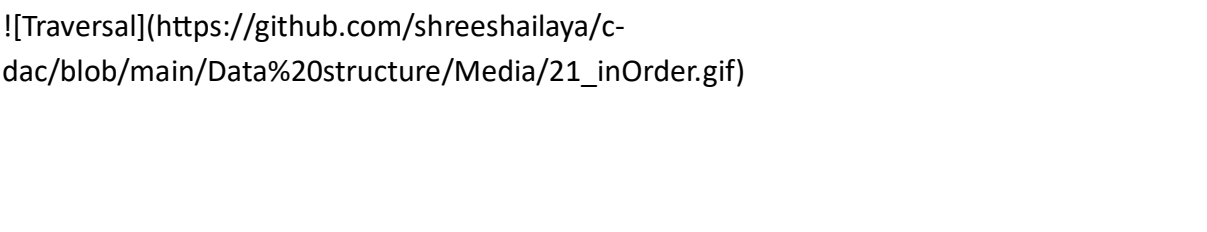
- Print Tree Recrusion function

Inorder (LEFT,ROOT,RIGHT)

```java
    public void inOrder(Node root)
      {
        Node move=root;
        if(move!=null)
        {
```

```
                inOrder(move.getLeft());
                System.out.println("  "+move.getData());
                inOrder(move.getRight());
            }
        }
```

```
```



Preorder(ROOT,LEFT,RIGHT)

```java
```

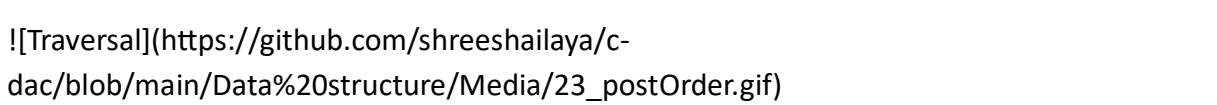
```
    public void preOrder(Node root)
      {
            Node move=root;
            if(move!=null)
            {
                System.out.println("  "+move.getData());
                preOrder(move.getLeft());
                preOrder(move.getRight());
            }
        }
```

```
```

![Traversal](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/22_preOrded.gif)

Postorder(LEFT,RIGHT,ROOT)

```java

	public void postOrder(Node root)
	  {
		Node move=root;
		if(move!=null)
		{
			postOrder(move.getLeft());
			postOrder(move.getRight());
			System.out.println("  "+move.getData());
		}
	  }

```

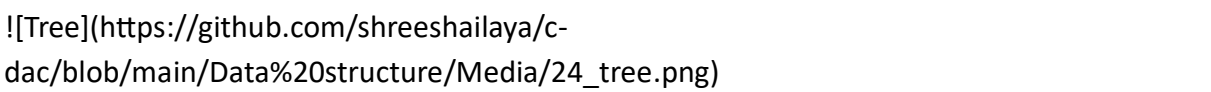


- Recursion function saves values in stack so it print stack wise.

- **In order gives data in sorted manner**

Search Element from tree

Delete Node from tree



(https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/Media/24_tree.png)

Graph

- Vertex
- Edge
- Path
- Adjacent Node

- in degree o0ut degree
- sink node/Source node

Day 13 30/6(Graph,unDirectedGraph,Graph Traversa,Adjency List)

Graph

```java

class Graph
{
	int vertex;
	int graph[][];
```

```
public Graph()
{
        vertex =5;
        graph=new int[vertix][vertex]
}

public Graph(int size)
{

}

public void acceptGraph()
{
        Scanner sc = new Scanner(SI)
        syso("Enter adjency of ")
        fot(int i=0;i<vertex;i++)
        {
        for(int j=0j<vertex;j++)
        {
                syso("[i][j]");
                graph[i][j]=sc.nextInt();
        }

        }
}
```

```
	public void displayGraph()
	{
		Scanner sc = new Scanner(SI)
		syso("Enter adjency of ")
		fot(int i=0;i<vertex;i++)
		{
			syso();
		for(int j=0j<vertex;j++)
		{
			syso(+graph[i][j]);
		}

		}
	}
}
```

```
### unDirectedGraph
```java
public void addEdge(int i,int j)
{
	graph[i][j]=1;
	graph[j][i]=1;
}
```
```

```

### Graph Traversa

- DFS-depth

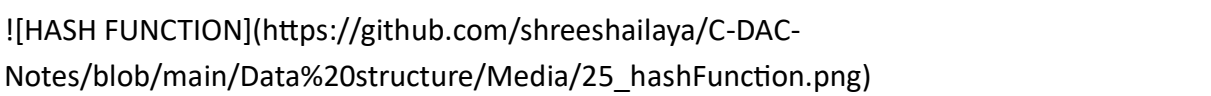
- BFS-level

### Adjency List

# Day 14 1/7(Hash,time complixity,kruskal Algorithm,Prime's Algo,AVL Tree Balanced Factor)

### Hash

**Definition:** It is a process of converting an input of any length into a fixed size string of text using a mathematical function.




There are many formulas to Hash such as SHA.

**Why do we need Hashing?**

In a large array of values, it is difficult to search for a particular item in an array.
```

We would need to do a binary search. However, if we are aware of the index of the element, it is easier to retrieve the element itself.

For example:

We need to find “Thiru” in the following array:

Sven, Anna, Mia, Olaf, Thiru, Jaya, Amit, Harry, Rupert, Justin

We would need to iterate through each element to check if the element matches “Thiru”.

Alternatively, had we known the index of Thiru, It would have been easier to access.

In the following array:

Sven,

Anna,

Mia,

Olaf,

Thir,

Jaya,

Amit,

Harr,

Rupe,

Just,

Array(4) would have fetched us “Thiru”.

We could further devise a function which could create a relationship between the value and the index we store values in.

Let the function we used to find the index to store the value be = Sum of the ASCII of the word mod the size of the array

For Sven: ASCII of S + ASCII of v + ASCII of e + ASCII of n

= 115 + 118 + 101 + 110

=444

The size of the array is 10. Thus 444 mod 10 is 4.

Index	Element
0	-
1	-
2	-
3	-
4	Sven
5	-
6	-
7	-
8	-
9	-

Similarly for Mia, Sum of ASCII = 77 + 105 + 97

= 279 % 10 (size of the array) = 9

The element “Mia” would go in the 9th index of the array

| Index | Element |

| ---------- |:----------:|

| 0 | - |

| 1 | - |

| 2 | - |

| 3 | - |

| 4 | Sven |

| 5 | - |

| 6 | - |

| 7 | - |

| 8 | - |

| 9 | Mia |

Similarly,

The following are the ASCII sum

| String | ASCII sum |

| ---------- |:----------:|

| Mia | 279|

| Anna | 382 |

| Olaf | 386 |

| Thiru | 524 |

| Jaya | 389 |

| Amit | 395 |

| Harry | 518 |

| Rupert | 642 |

| Justin | 637 |

| String | ASCII sum % 10 |

| ---------- |:----------:|

| Mia | 9 |

| Anna | 2 |

| Olaf | 6 |

| Thiru | 4 |

| Jaya | 9 |

| Amit | 5 |

| Harry | 8 |

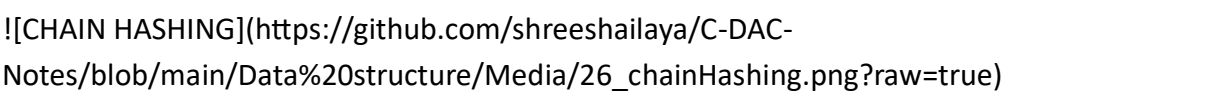
| Rupert | 2 |

| Justin | 7 |

Anna goes to the 2nd row, Olaf to the 6th. But Thiru and Sven have the same index. This is known as collision i.e two or more keys are mapped to same value.

One solution to **collision** is to insert Thiru in the next index i.e is the 5th index.

Another solution to the problem is to use **Chain Hashing**. The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value.



(https://github.com/shreeshailaya/C-DAC-Notes/blob/main/Data%20structure/Media/26_chainHashing.png?raw=true)

We have different ways to achieve Hashing. Few of them are described below

| Data Structure | Description |

| ---------- |:----------:|

| HashTable | Synchronised |

| HashMap | - Non-Synchronised, Faster|

| LinkedHashMap |Keeps the order of inserts |

| ConcurrentHashMap |Synchronised, Faster as locks are used |

| HashSet |Maintains only keys |

| LinkedHashSet |Keeps the order of inserts, Maintains only Key |

Universal Hashing

Let H be a finite collection of hash functions that map a given universe U of keys into the range {0, 1..... m-1}. Such a collection is said to be universal if for each pair of distinct keys k,l∈U, the number of hash functions h∈ H for which h(k)= h(l) is at most |H|/m. In other words, with a hash function randomly chosen from H, the chance of a collision between distinct keys k and l is no more than the chance 1/m of a collision if h(k) and h(l)were randomly and independently chosen from the set {0,1,...m-1}.

Rehashing

In case the hash table becomes nearly full, the running time for the operations of will start taking too much time, insert operation may fail in such situation, the best possible solution is as follows:

Create a new hash table double in size.

Scan the original hash table, compute new hash value and insert into the new hash table.

Free the memory occupied by the original hash table.

Example in Java for an HashMap:

```java
import java.util.HashMap;
import java.util.Map;

//Given a string, count the frequency/occurence of each Character
public class FindTheOccurenceOfCharacter {
   static HashMap<Character, Integer> getOccurence(String inputString) {
      HashMap<Character, Integer> characterFrequencyMap
```

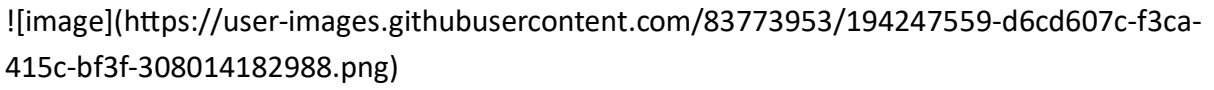
```
    = new HashMap<Character, Integer>();



char[] inputArray = inputString.toLowerCase().toCharArray();

for (char character : inputArray) {
    if (characterFrequencyMap.containsKey(character)) {
        characterFrequencyMap.put(character, characterFrequencyMap.get(character) + 1);
    }
    else {
        characterFrequencyMap.put(character, 1);
    }
}

return characterFrequencyMap;
}

static void  printOccurence(HashMap<Character, Integer> map){
    for (Map.Entry<Character, Integer> value: map.entrySet()) {
        System.out.println(value.getKey() + "->" + value.getValue());
    }
}

public static void main(String[] args) {
    String inputString = "Test";

    HashMap<Character, Integer> map = getOccurence(inputString);
```

```
        printOccurence(map);


    }
}
```

Time Complixity

Time complexity is defined as the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm. It is not going to examine the total execution time of an algorithm.

So, if computing 10 elements take 1 second, computing 100 elements takes 2 seconds, 1000 elements take 3 seconds, and so on.



What are the Different Types of Time complexity Notation Used?

As we have seen, Time complexity is given by time as a function of the length of the input. And, there exists a relation between the input data size (n) and the number of operations performed (N) with respect to time. This relation is denoted as Order of growth in Time complexity and given notation O[n] where O is the order of growth and n is the length of the input. It is also called as 'Big O Notation'

Big O Notation expresses the run time of an algorithm in terms of how quickly it grows relative to the input 'n' by defining the N number of operations that are done on it. Thus, the time complexity of an algorithm is denoted by the combination of all O[n] assigned for each line of function.

There are different types of time complexities used, let’s see one by one:

1. Constant time – O (1)

2. Linear time – O (n)

3. Logarithmic time – O (log n)

4. Quadratic time – O (n^2)

5. Cubic time – O (n^3)

and many more complex notations like Exponential time, Quasilinear time, factorial time, etc. are used based on the type of functions defined.

Lets Understand With Some Examples:

```` java
import java.io.*;

class TC {
	public static void main(String[] args)
	{
		System.out.print("Hello World");
	}
}

output : Hello World
````

Time Complexity: In the above code “Hello World” is printed only once on the screen.

So, the time complexity is constant: O(1) i.e. every time a constant amount of time is required to execute code, no matter which operating system or which machine configurations you are using.

```` java
class TC {

   public static void main(String[] args)
   {
      int i, n = 8;
      for (i = 1; i <= n; i++) {
         System.out.printf("Hello World !!!\n");
      }
   }
}
````

Output :

```
Hello World !!!
Hello World !!!
Hello World !!!
Hello World !!!
Hello World !!!
Hello World !!!
Hello World !!!
Hello World !!!
```

Time Complexity: In the above code “Hello World !!!” is printed only n times on the screen, as the value of n can change.

So, the time complexity is linear: O(n) i.e. every time, a linear amount of time is required to execute code.

Comparison of functions on the basis of time complexity

It follows the following order in case of time complexity:

O(nn) > O(n!) > O(n3) > O(n2) > O(n.log(n)) > O(n.log(log(n))) > O(n) > O(sqrt(n)) > O(log(n)) > O(1)

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

Space Complexity

Space complexity of an algorithm quantifies the amount of time taken by a program to run as a function of length of the input. It is directly proportional to the largest memory your program acquires at any instance during run time.

For example: int consumes 4 bytes of memory.

How Significant Are Space and Time Complexity?

Significant in Terms of Time Complexity

The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

Here is an example.

Assume you have a set of numbers S= (10, 50, 20, 15, 30)

There are numerous algorithms for sorting the given numbers. However, not all of them are effective. To determine which is the most effective, you must perform computational analysis on each algorithm.

Here are some of the most critical findings from the graph:

* This test revealed the following sorting algorithms: Quicksort, Insertion sort, Bubble sort, and Heapsort.

* Python is the programming language used to complete the task, and the input size ranges from 50 to 500 characters.

* The results were as follows: "Heap Sort algorithms performed well despite the length of the lists; on the other hand, you discovered that Insertion sort and Bubble sort algorithms performed far worse, significantly increasing computing time." See the graph above for the results.

* Before you can run an analysis on any algorithm, you must first determine its stability. Understanding your data is the most important aspect of conducting a successful analysis.

Time Complexity vs. Space Complexity

You now understand space and time complexity fundamentals and how to calculate it for an algorithm or program. In this section, you will summarise all previous discussions and list the key differences in a table.

| Time Complexity | Space Complexity |

| ----------| ----------|

|Calculates the time required | Estimates the space memory required |

|Time is counted for all statements | Memory space is counted for all variables, inputs, and outputs.|

|The size of the input data is the primary determinant. | Primarily determined by the auxiliary variable size|

| More crucial in terms of solution optimization | More essential in terms of solution optimization|

kruskal Algorithm

Defenition: Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph.

Minimum Spanning Tree Contains all the vertices in the graph but has the minimum sum of weights among all the trees that can be formed from the graph.

How does it work

Kruskal's algorithm is a greedy algorithm that finds a local optimum as an initiation step to find the global optimum.

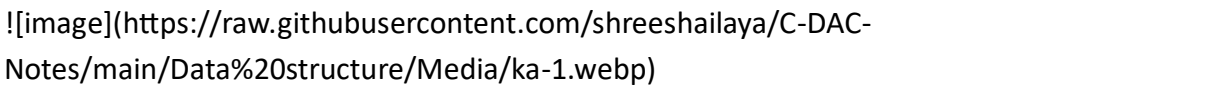
The steps for implementing Kruskal's algorithm are as follows:

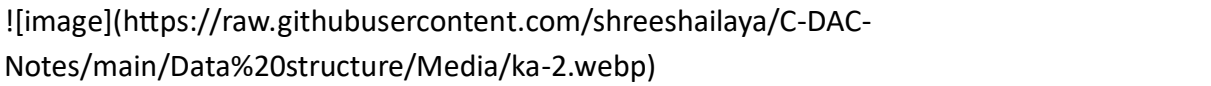
1. Sort all the edges from low weight to high.

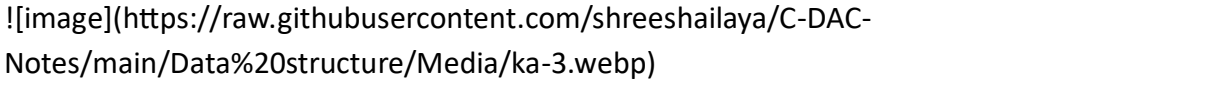
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

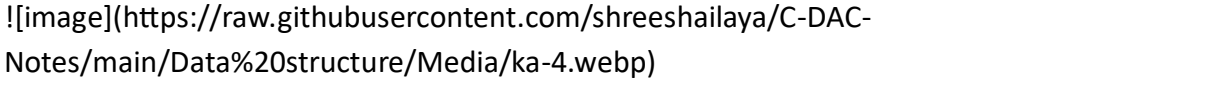
3. Keep adding edges until all vertices are reached.

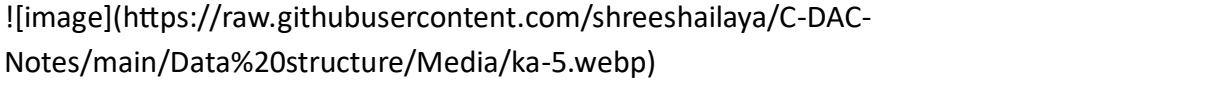
Let us consider a graph. Now the minimum spanning tree using Kruskal's algorithm can be created as follows

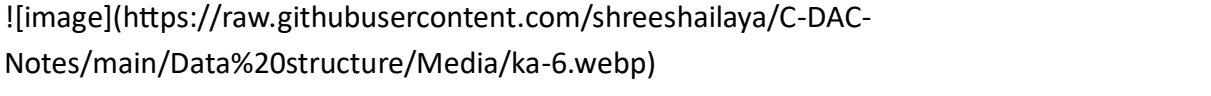












Time Complexity: This algorithm has a time complexity of O(e*log(e)) where e is the number of edges in the graph

For the code snipet the weight of the edges are as given below.

|Edge |0-1|0-2|1-2|2-3|2-5|2-4|3-4|5-4|

|Weight| 4 | 4 | 2 | 5 | 2 | 4 | 3 | 3 |

Next step is to sort the table according to the ascending order of the weights

Which will given the following result.

|Edge |1-2|2-5|3-4|5-4|0-1|0-2|2-4|2-3|

|Weight| 2 | 2 | 3 | 3 | 4 | 4 | 4 | 5 |

```java
// Kruskal's algorithm in Java

import java.util.*;

class kruskal {
  class Edge implements Comparable<Edge> {
    int src, dest, weight;

    public int compareTo(Edge compareEdge) {
      return this.weight - compareEdge.weight;
    }
  };

  // Union
  class subset {
    int parent, rank;
  };

  int vertices, edges;
  Edge edge[];

  // Graph creation
  kruskal(int v, int e) {
```

```
  vertices = v;
  edges = e;
  edge = new Edge[edges];
  for (int i = 0; i < e; ++i)
    edge[i] = new Edge();
}

int find(subset subsets[], int i) {
  if (subsets[i].parent != i)
    subsets[i].parent = find(subsets, subsets[i].parent);
  return subsets[i].parent;
}

void Union(subset subsets[], int x, int y) {
  int xroot = find(subsets, x);
  int yroot = find(subsets, y);

  if (subsets[xroot].rank < subsets[yroot].rank)
    subsets[xroot].parent = yroot;
  else if (subsets[xroot].rank > subsets[yroot].rank)
    subsets[yroot].parent = xroot;
  else {
    subsets[yroot].parent = xroot;
    subsets[xroot].rank++;
  }
}

// Applying Krushkal Algorithm
```

```
void KruskalAlgo() {
  Edge result[] = new Edge[vertices];
  int e = 0;
  int i = 0;
  for (i = 0; i < vertices; ++i)
    result[i] = new Edge();

  // Sorting the edges
  Arrays.sort(edge);
  subset subsets[] = new subset[vertices];
  for (i = 0; i < vertices; ++i)
    subsets[i] = new subset();

  for (int v = 0; v < vertices; ++v) {
    subsets[v].parent = v;
    subsets[v].rank = 0;
  }
  i = 0;
  while (e < vertices - 1) {
    Edge next_edge = new Edge();
    next_edge = edge[i++];
    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);
    if (x != y) {
      result[e++] = next_edge;
      Union(subsets, x, y);
    }
  }
```

```
  for (i = 0; i < e; ++i)
    System.out.println(result[i].src + " - " + result[i].dest + ": " + result[i].weight);
}

public static void main(String[] args) {
  int vertices = 6; // Number of vertices
  int edges = 8; // Number of edges
  kruskal G = new kruskal(vertices, edges);

  G.edge[0].src = 0;
  G.edge[0].dest = 1;
  G.edge[0].weight = 4;

  G.edge[1].src = 0;
  G.edge[1].dest = 2;
  G.edge[1].weight = 4;

  G.edge[2].src = 1;
  G.edge[2].dest = 2;
  G.edge[2].weight = 2;

  G.edge[3].src = 2;
  G.edge[3].dest = 3;
  G.edge[3].weight = 3;

  G.edge[4].src = 2;
  G.edge[4].dest = 5;
  G.edge[4].weight = 2;
```

```
    G.edge[5].src = 2;
    G.edge[5].dest = 4;
    G.edge[5].weight = 4;

    G.edge[6].src = 3;
    G.edge[6].dest = 4;
    G.edge[6].weight = 3;

    G.edge[7].src = 5;
    G.edge[7].dest = 4;
    G.edge[7].weight = 3;
    G.KruskalAlgo();
  }
}
```

Prime's Algo

Defenition: Prim's algorithm is used to find the minimum spanning tree from a graph. This algorithm gives the same output as that of the Kruskal Algorithm.

How does it work

Like Kruskal's algorithm Prim's algoritm is also a greedy algorithm.

The steps for implementing Prim's algorithm are as follows:

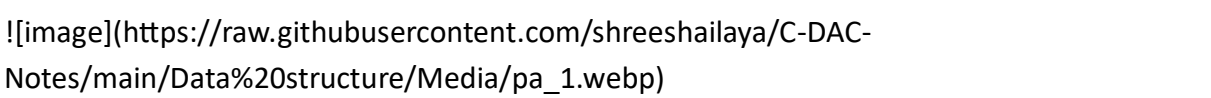
1. Initialize the minimun spanning tree by choosing a random vertex from the graph.

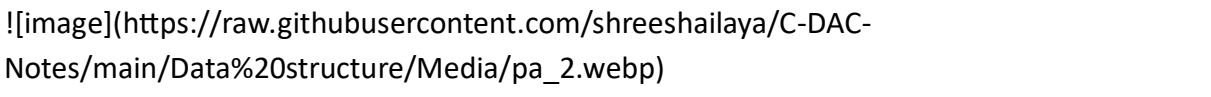
2. Add to the spanning tree the vertex which is connectd to the first vertex with least weight.

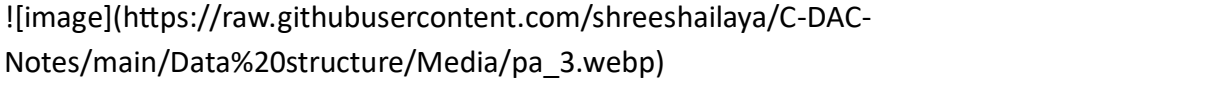
3. Keep choosing the next nearest (lowest weight) vertex not in the spanning tree.

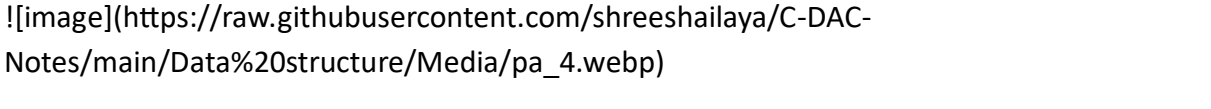
4. Repeat step 3 util the spanning tree has all the vertices as in the original graph.

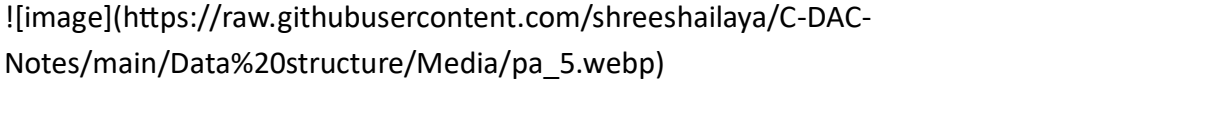
Let us consider a graph. Prim's algorithm finds the minimum spanning tree as follows

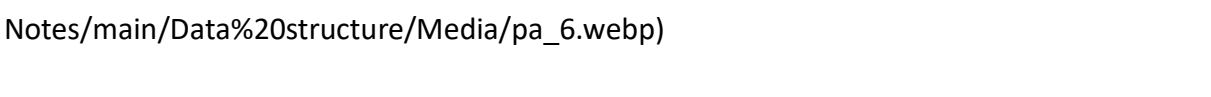












Time complexity The time complexity of Prim's algorithm is O(e*log(v)) where e is the number of edges in the graph and v the number of vertices in the graph.

For the code snippet weight of each edges of the graph are as given below

|Edge |0-1|0-2|1-2|1-3|1-4|2-3|2-4|3-4|
|Weight| 9 | 75| 95| 19| 42| 51| 66| 31|

```java
// Prim's Algorithm
import java.util.Arrays;

class PGraph {

  public void Prim(int G[][], int V) {

    int INF = 9999999;

    int no_edge; // number of edge

    // create a array to track selected vertex
    // selected will become true otherwise false
    boolean[] selected = new boolean[V];

    // set selected false initially
    Arrays.fill(selected, false);

    // set number of edge to 0
    no_edge = 0;

    // the number of egde in minimum spanning tree will be
```

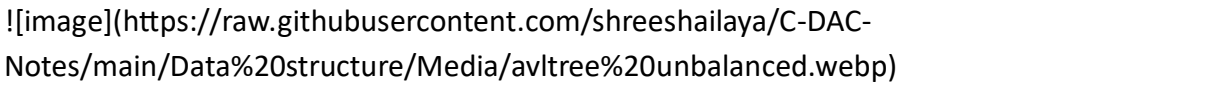
```
// always less than (V -1), where V is number of vertices in
// graph

// choose 0th vertex and make it true
selected[0] = true;

// print for edge and weight
System.out.println("Edge : Weight");

while (no_edge < V - 1) {
 // For every vertex in the set S, find the all adjacent vertices
 // , calculate the distance from the vertex selected at step 1.
 // if the vertex is already in the set S, discard it otherwise
 // choose another vertex nearest to selected vertex at step 1.

 int min = INF;
 int x = 0; // row number
 int y = 0; // col number

 for (int i = 0; i < V; i++) {
  if (selected[i] == true) {
   for (int j = 0; j < V; j++) {
    // not in selected and there is an edge
    if (!selected[j] && G[i][j] != 0) {
     if (min > G[i][j]) {
      min = G[i][j];
      x = i;
      y = j;
```

```
            }
          }
        }
      }
    }
    System.out.println(x + " - " + y + " :  " + G[x][y]);
    selected[y] = true;
    no_edge++;
  }
 }

 public static void main(String[] args) {
  PGraph g = new PGraph();
  // number of vertices in grapj
  int V = 5;
  // create a 2d array of size 5x5
  // the adjacency matrix to represent graph
  int[][] G = { { 0, 9, 75, 0, 0 }, { 9, 0, 95, 19, 42 }, { 75, 95, 0, 51, 66 }, { 0, 19, 51, 0, 31 },
     { 0, 42, 66, 31, 0 } };

  g.Prim(G, V);
 }
}
```

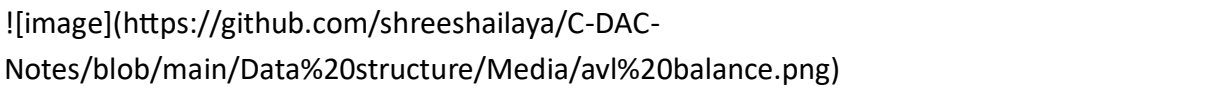
AVL Tree Balanced Factor

Defenition: Balance factor of an AVL Tree is the height difference between the left and rigth subtree of a vertex in the AVL Tree.Each vertex in an AVL Tree sholud have a balance factor 1,0 or -1.

AVL Tree: A Binary tree with each vertex having a balance factor of 1,0 or -1 is an AVL Tree.



In the above diagram since three of the vertices are having balance factor as 2 it is not an AVL Tree.Such Binary trees can be converted to AVL Tree by any of the four rotation methods.



In the above diagram the numbers noted above each vertex is the balance factor of that vertex.Since each vertex is having either balance factor 1,0 or -1 it is an AVL Tree.

Sort

***

***

Sorting

Bubble Sort

Defination: It is a comparison based sorting algorithm wherein comparing adjacent elements is a primitive operation. In each pass, it compares the adjacent elements in the array and exchanges those that are not in order.As camparision goes from bottom to top that's the reason this sorting being called BUBBLE SORT.

Working Steps:

1.) Start with the first element of the array.

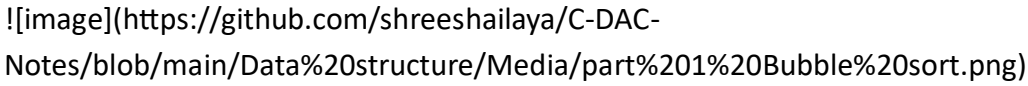
2.) Compare it with the next element of array. If the first one is bigger then swap the positions , if not then don't change anything.

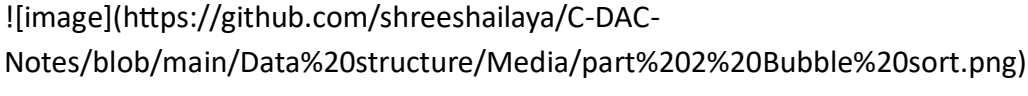
3.) Again do the same thing with second and third element if we find bigger then swap them. Once you cover all element , you will find that the biggest element is on the end of array.

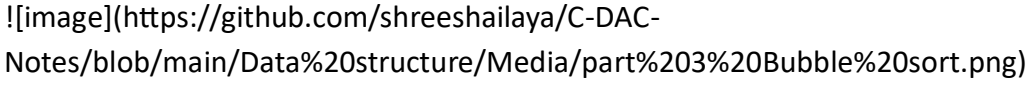
4.) Repeat steps again and then you will find the 2nd biggest element is on then 2nd last position on array.

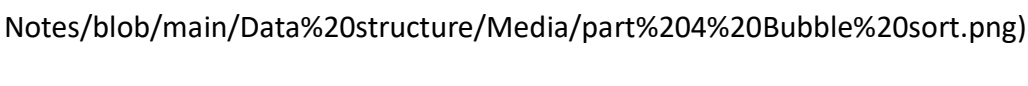
5.) Repeat all these steps until you find the sorted array.

For better understanding lets grasp this concept by a pictorial view.









On third Iteration we got our sorted array but our compiler will run this code till N Iterations where N is the number of elements in an array.

Code In Java

```java

class Main
{
   static void bubbleSort(int a[])
{
int len = a.length; // calculating the length of array
for (int i = 0; i < len-1; i++)
for (int j = 0; j < len-i-1; j++)
if (a[j] > a[j+1]) //comparing the pair of elements
{
// swapping a[j+1] and a[i]
 int temp = a[j];
 a[j] = a[j+1];
 a[j+1] = temp;
}
}

  /* Prints the array */
  static void printArray(int a[])
  {
     int len = a.length;
     for (int i = 0; i < len; i++)
        System.out.print(a[i] + " ");
           //printing the sorted array

     System.out.println();
```

```
    }

    // Main method to test above
    public static void main(String args[])
    {
        int arr[] = {5 , 3 , 9 , 1 , 7};

        bubbleSort(arr);//calling the bubbleSort function

        System.out.println("Sorted array");

        printArray(arr); //calling the printArray function
    }
}
```

Complexity

This algorithm has a worst-case time complexity of O(n^2). And has a space complexity of O(1).

Cyclic Sort

Insertion Sort

Selection Sort

Merge Sort

Shell sort

***

***

| Concept | What | Why | Where | Comment | Referance |

|---|---|---|---|---|---|

| Data Structure | A data structure is a particular way of organizing data in a computer so that it can be used effectively. | Data structures are used as a framework for organizing and storing information in virtual memory forms. | In evey application | - | - |

| Array | An array is a collection of items stored at contiguous memory locations. | For Storing data | in heap | - | [Array](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/17juneDay2/assignment1.java) |

| Linked List | Linked list is linear data structure where each element is dynamically allocated. Each node contain two parts, namely the data and the reference | to overcome disadvantage of array | Data Structure(Ralilway)| - | [LinkedList](https://github.com/shreeshailaya/c-dac/blob/main/Data%20structure/17juneDay2/assignment2.java) |

***

Compiled by [Shreeshail Vitkar](https://github.com/shreeshailaya)

Feel free to fork @ [C-dac Notes](https://github.com/shreeshailaya/c-dac)

***