**Data structure and Algorithm:**

-arrangementof data in computer memory usage/storage

-implementation of abstract datatypes ADT

-insert, search, delete data

-efficient way, without data structure also we can do code

1.Primitive datatype

2.Abstract datatype

1. Linear data structure- Array, linked list, stack, queue
2. Non linear structure- Tree, Graph

Big ‘O’ notation: function is of order of n, written as o(n) like that.

1.O(1) constant 2.O(log n)logarithmic 3.O(n log n) 4.O(quadratic) 5.O(cubic) 6.O(exponential)

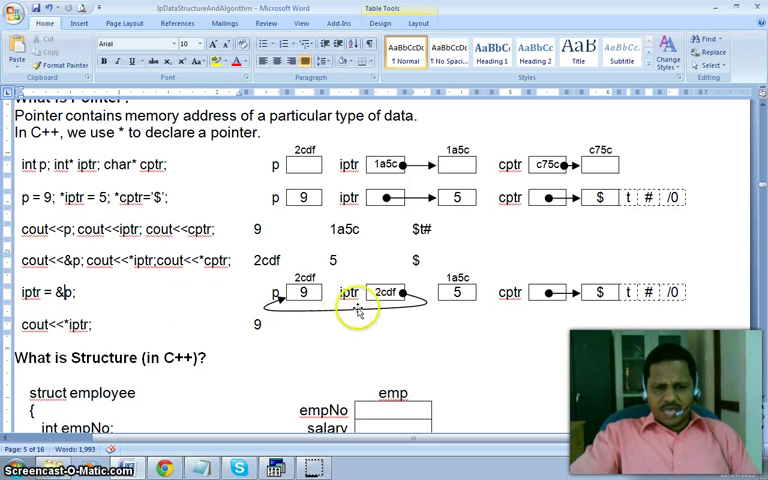
**Array data structure:**  group of continuous similar type of variables

We cant delete operation in array, just to put null or 0 value

**Pointers-** it is a interger which contain address

if we print interger pointer like cout<<iptr; it will print address

If we print character pointer like cout<<cptr; it will print linear character in that address



**Structure:** struct employee{int a; int b;} emp;

emp.a=4; emp.b=5; //normal method

struct employee {int a; int b;}

employee\* ptr;

ptr= new employee;

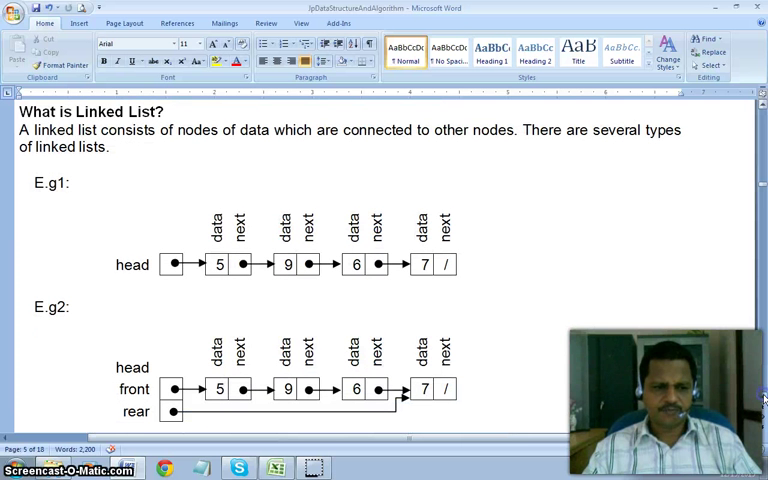
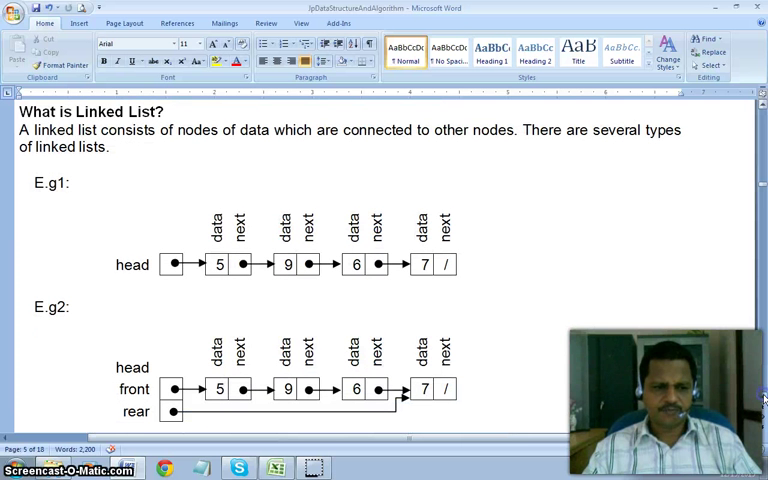
ptr->a=4;

ptr->b=5; //using pointer instead of object

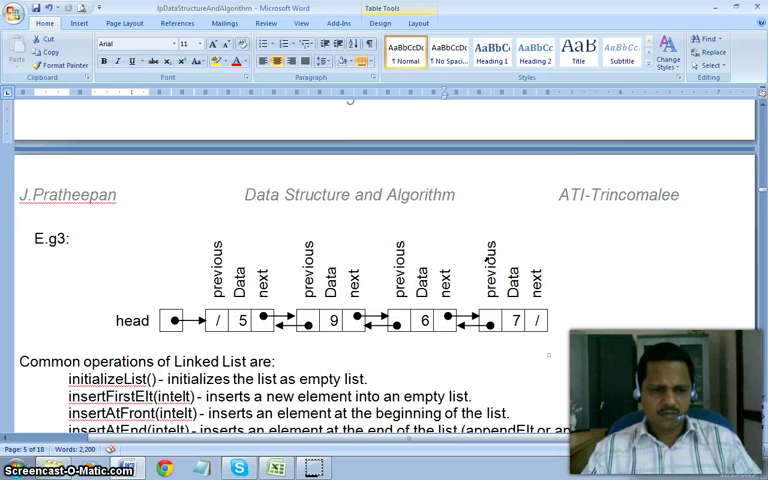
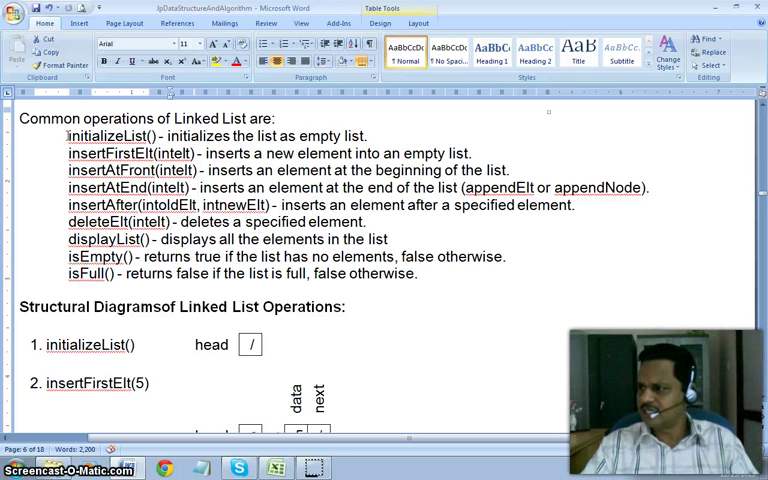
# **Linkedlist:**

consistes of nodes of data

## 1.Singly linked list

## 2.Doubly linked list

|  |  |  |
| --- | --- | --- |
| intializeList(); | Initialize list as empty list |  |
| insertFirstElt(5); | Insert element in empty list |  |
| insertAtFront(3); | Insert beginning of list |  |
| insertAtEnd(8); | Insert at end of list |  |
| insertAtAfter(5,7); | Insert after specified elemt |  |
| deleteList(); | Delete specified element |  |

Program:

#include <conio.h>

class LinkedList

{

private:

struct listNode

{ int data;

listNode \*next; //pointer has another list node

};

listNode \*head;

public: LinkedList();

void intializeList();

void insertFirstElt(int elt);

void insertAtFront(int elt);

void insertAtEnd(int elt);

void insertAfter(int oldElt, int newElt);

void deleteElt(int elt);

void displayList();

int isEmpty();

int isFull(); }

LinkedList::LinkedList() //constructor

{ head=NULL; }

void LinkedList::intializeList()

{ head=NULL; }

void LinkedList::insertFirstElt(int elt)

{ head= new listNode;

head->data=elt;

head->next=NULL; }

void LinkedList::insertAtFront(int elt)

{ listNode \*newNode;

newNode= new listNode;

newnode->data=elt;

newnode->next=head;

head=newNode; }

void LinkedList::insertAtEnd(int elt)

{ listNode \*newNode, \*curNode;

newNode= newlistNode;

newNode->data=elt;

newNode->next=NULL;

if(!head)

head=newNode;

else

{ while(curNode->next!=NULL)

curNode=curNode->next;

curNode->next=newNode; } }

void LinkedList::displayList()

{ listNode\* curNode;

curNode=head;

while(curNode)

{cout<<curNode->data<<” “;

curNode=curNode->next; }}

int LinkedList::isEmpty()

{ if(head==NULL )

{ return 1;}

else{return 0;}}

int LinkedList:: isFull() // it always return false only

{return 0;}

void main(){

LinkedList lst;

Ist.insertAtEnd(4);

Ist.insertAtEnd(6);

Ist.insertAtEnd(5);

Ist.displayList(); }

**Adv:** easy to add delete element, no wasted memory

**DisAdv:** slow in process

# **Stack-**

handles LIFO, we can remove most recently element

Common opertions of stack are:

intializeStack()-intializes the stack as empty stack

push()-adds an element on top of the stack

pop()-removes and returns top most element from the stack

topElt()-returns top element without removing it

isEmpty()-reuturns true if stack has no elements or else false

isFull()-returns true is stack full or else false

displayStack()-displays all elements from top to bottom

Two types of stack implementation:

Static implementation(array based)

Dynamic implementation

Graphical representation of stack operation:

|  |
| --- |
|  |

1.intializeStack()

|  |
| --- |
|  |

top=-1  
2.p=isEmpty()

top=-1

p=true

|  |
| --- |
| 5 |

3.push(5)

top=0

|  |
| --- |
| 7  6  5 |

4.push(7)

top=2

|  |
| --- |
| 7  6  5 |

5.q=isEmpty(); r=isFull();

top=2

q=false; r=false;

|  |
| --- |
| 6  5 |

6.x=pop()

x=7

top=1

|  |
| --- |
| 6  5 |

7.y=topElt()

y=6

top=1

**Array implemetation c++ program:**

#include <iostream.h>

#include <conio.h>

const STK\_SIZE=5;

class Stack

{ private: int top; int stk[STK\_SIZE];

public: Stack();

void push(int);

int pop();

int topElt();

int isEmpty();

int isFull();

void displayStack(); }

Stack::Stack()

{ top=(-1);}

void Stack::push(int elt){

if(top<STK\_SIZE)stk[++top]=elt; }

int Stack::pop(){

if(top>-1) return stk[top--];

else return 999; //some invalid integer should be returned

}

int Stack::topElt(){

if(top>-1) return stk[top];

else return 999; // some invalid integer should be returned }

int Stack::isEmpty(){

return(top==-1) ; }

int Stack::isFull(){

return(top==(STK\_SIZE-1)); }

void Stack::displayStack(){

int i=top;

while(i>-1){ cout<<stk[i]<<endl; i--; }}

void main(){

clrscr();

Stack s;

s.push(5);

s.push(6);

s.push(7);

int x=s.pop();

s.push(9);

s.displayStack(); }

output: 9 6 5

**Dynamic (Linked list based) implementation of stack:** no limit memory

intializeStack()-top=NULL ; //similar to intializeList() and use top instead of head

push()-newNode->next=top;top=newNode; ///similar to insertAtFront()

pop()-x=top->data; top=top->next; return x;

topElt()-returntop->data

isEmpty()-if(top==NULL) return 1 else return 0

is Full()- return 0l //always return false

displayStack()-similar to displayList()

Adavantage: LIFO Disadvantage:difficult to access other items

# Queue data structure:

FIFO, we can remove element which added first earlier

Common operations of queue are:

intializeQueue()-intialize the queue as empty queue

enQueue()-adds an element at the rear of the queue

deQueue()-removes and return the front element from the queue

frontElt()-returns the front element without removing it

isEmpty()-returns true if the queue has no elements or else return false

isFull()-returns true if the queue is full or else false

displayQueue()-display all elements from front to rear

Graphical representation of queue operation:

1.intializeQueue()

0 1 2 3 4

|  |
| --- |
|  |

front=-1

rear=-1

size=0

2.p=isEmpty()

p=true0

0 1 2 3 4

|  |
| --- |
|  |

front=-1

rear=-1

size=0

3.enQueue(5)

0 1 2 3 4

|  |
| --- |
| 5 |

front=-1

rear=0

size=1

4.enQueue(9)

enQueue(7)

0 1 2 3 4

|  |
| --- |
| 5 9 7 |

front=-1

rear=2

size=3

5.x=deQueue()

x=5

0 1 2 3 4

|  |
| --- |
| 9 7 |

front=0

rear=2

size=2

6.enQueue(2)

enQueue(6)

0 1 2 3 4

|  |
| --- |
| 9 7 2 6 |

front=0

rear=4

size=4

7.q=isFull()

q=false

0 1 2 3 4

|  |
| --- |
| 9 7 2 6 |

front=0

rear=4

size=4

8.enQueue(3)

0 1 2 3 4

|  |
| --- |
| 3 9 7 2 6 |

front=0

rear=0

size=5

9.r=isFull()

y=deQueue()

r=true

y=9

0 1 2 3 4

|  |
| --- |
| 3 7 2 6 |

front=1

rear=0

size=4

Static (array based) implementation of queue operation

#include<iostrean.h>

#include<conio.h>

const Q\_SIZE=5;

class Queue{

private:

int front,rear,soze;

int que[Q\_SIZE];

public:

Queue();

void intializeQueue();

void enQueue();

int deQueue();

int frontElt();

int isEmpty();

int isFull();

void dispalyQueue(); }

Queue::Queue() {

front=(-1);

rear=(-1);

size(0); }

void Queue::enQueue(int elt)[

if(size<Q\_SIZE){

rear=(rear+1)%Q\_SIZE;

que[rear]=elt;

size++; } //else cout<<”queue is full”

}

int Queue::deQueen(){

if(size>0){

front=(front+1)%Q\_SIZE;

size--;

return que[front]; }

else return 999; //some invalid should be returned or cout<<”queue is empty”

}

int Queue::frontElt(){

if(Size>0)

return que[front];

else

return 999; //some invalide integer should be returned or cout<<”queue is empty”

}

int Queue::isEmpty(){

return (size==0); }

int Queue::isFull(){

return(size==Q\_SIZE); }

void Queue::displayQueue()

{ int i=front;

for(int j=1;j<=size;j++)

{ i=(i+1)%Q\_SIZE;

cout<<que[i]<<endl;

}

}

void main(){

clrscr();

Queue q;

q.enQueue(5);

q.enQueue(9);

q.enQueue(7);

int x=q.deQueue();

q.enQueue(2);

q.enQueue(6);

q.enQueue(3);

int y=q.deQueue();

q.displayQueue();

}

output:7 2 6 3

**Dynamic (Linked list based) implementation of Queue:** no limit memory

intializeQueue()-front=NULL ;//similar to intializeList() and use top instead of head

enQueue()-newNode->next=top;top=newNode; ///similar to insertAtFront()

deQueue()-move to thte last node, get the data, remove the last node, return data

topElt()-returntop->data

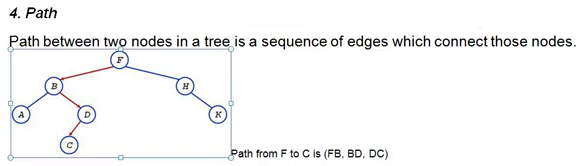
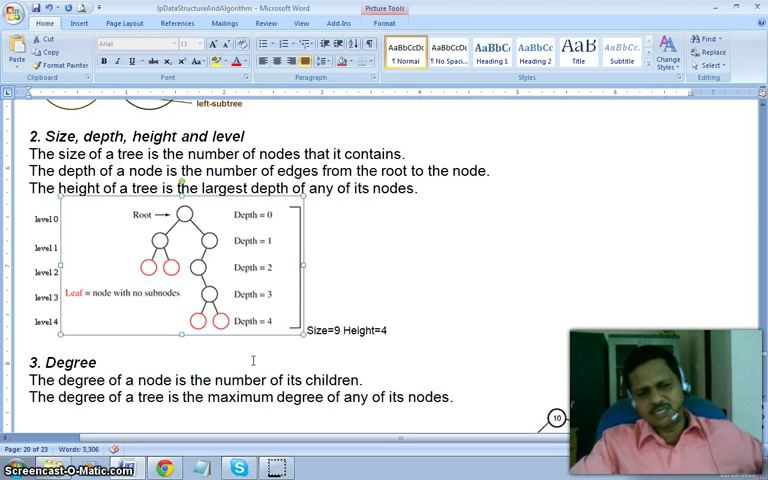
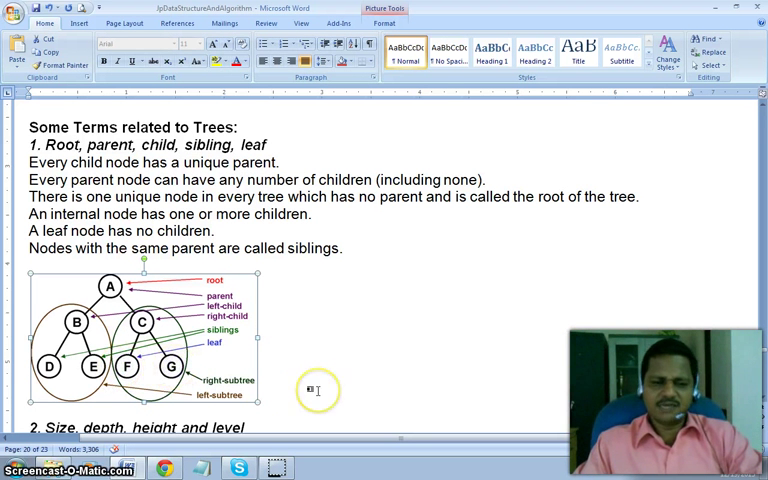
isEmpty()-if(top==NULL) return 1 else return 0

is Full()- return 0l //always return false

displayQueue()-similar to displayList()

Advantages:FIFO Disadvantages: difficult to access other items

Tree

widely used datastructure that emulates hierrarical tree structure with set of linked nodes

Special types of tress:

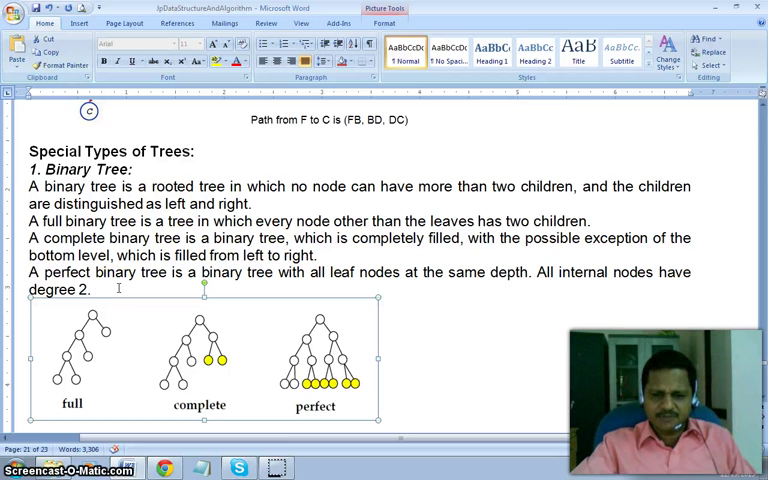
1.Binary tree:

It is a rooted tree in which no node can have more than two children, and children are distinguished as left and right

A full binary tree is a tree in which every node other than the leaves has two children

Acomplete binary tree is a binary tree which is completely filled with the possible exception of the bottom leve, which is filled from left to right

A perfect binary tree is a binary tree all leaf nodes at the same depth, all internal nodes have degree



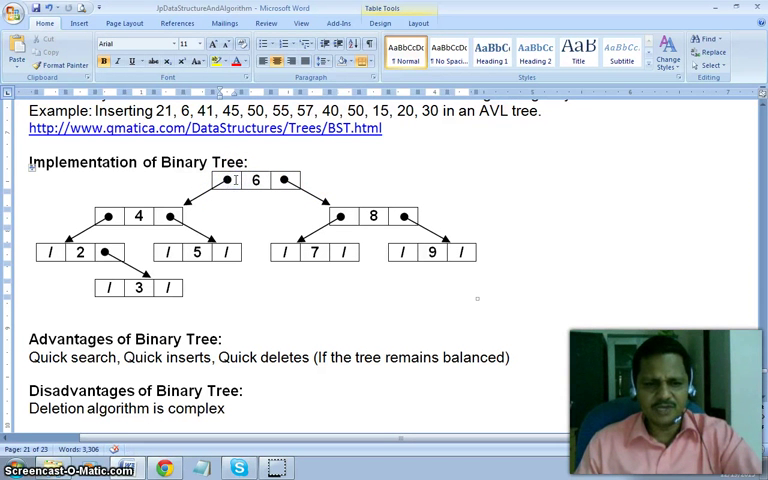
2.Binary search tree

Small element should goto left side and large or equal element foto right side of tree

 Example: binary search tree for 8 10 3 14 13 6 1 7 4

3.AVL (adelson velsky landis) tree:

no different height between left right by more than 1 . it wil automaticaly arrange to equal heights of both sides



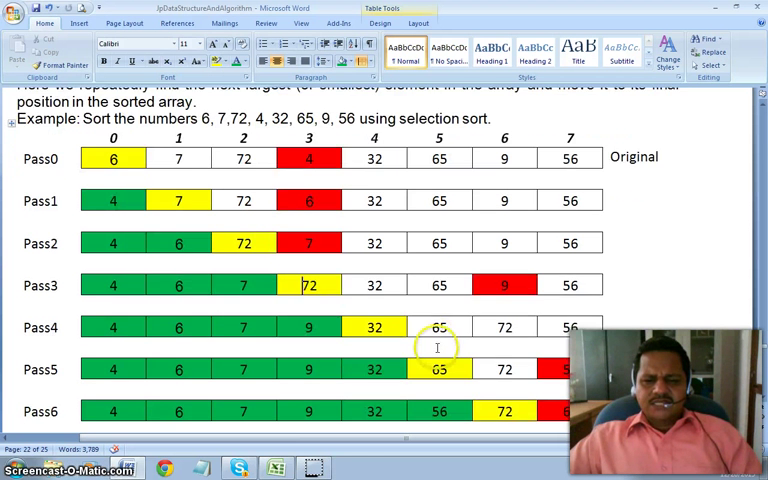
Advantages: quick search, quick insert,quick delete(if the tree remains balanced)

Disadvntage: deleting algorithm complex

**Sorting**

Arranging items in ascending or decending order. the cases occurs are nearly sorted, reversed, random etc.

Selection sort:

we repeatedly fine the next large(smallest) element in array and move it to final position 

Algorithm/pseudocode:

swap(x,y)

t=x

x=y

y=t

selectionSort(a[],n) //let ‘a’ be array containing ‘n’ items

for i=0 to n-1

m=i

for j=I to n-1

next j;

swap(a[i],a[m])

next i

C++ code:

#include<iostream.h>

#include<conio.h>

void displayArray(int\*a,int n)

{ int i;

for(i=0;i<n;i++)

cout<<a[i]<<” “; }

void swap(int \*x. int\*y){

int t;

t=(\*x);

\*x=(\*y);

\*y=t; }

void selectionSort(int \*a, int n){

int I,j,m,t;

for(i=0;j<n;j++){

if(a[j]<a[m] m=j;

swap(&a[i],&a[m];) }}

void main(){

clrscr();

int a[]={4,65,2,-31,0,99,2,83,782,1};

int n=10;

displayArray(a,n);

cout<<endl;

selectionSort(a,n);

displayArray(a,n); }

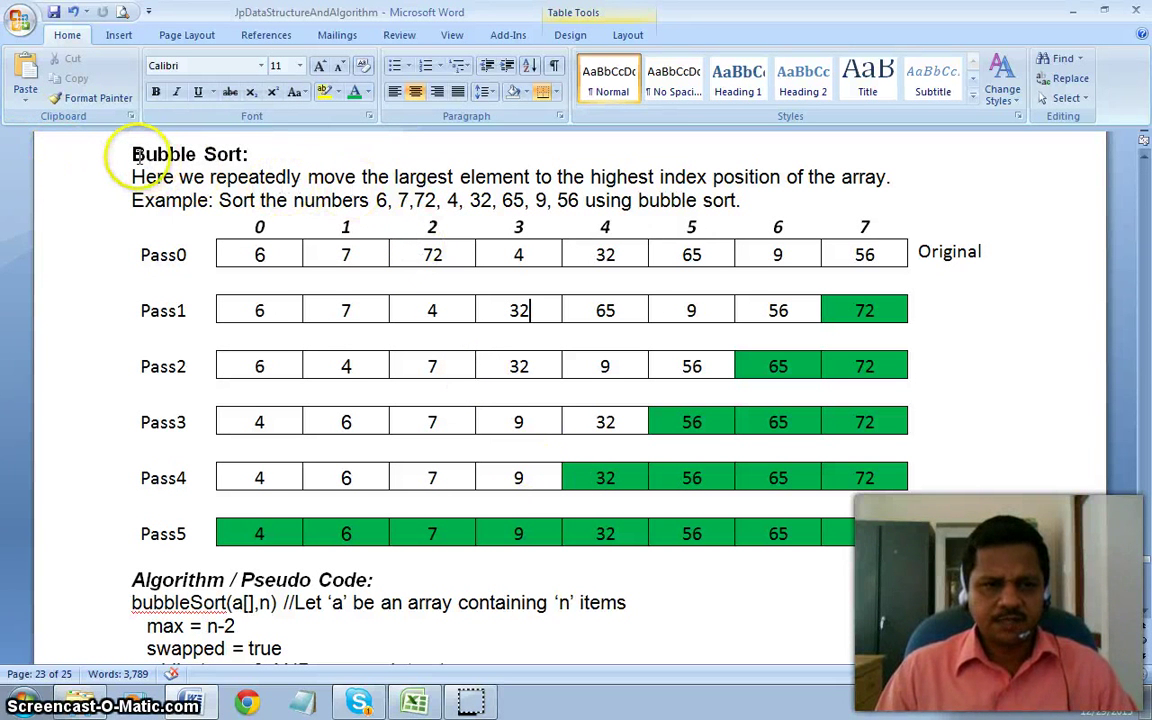
output:

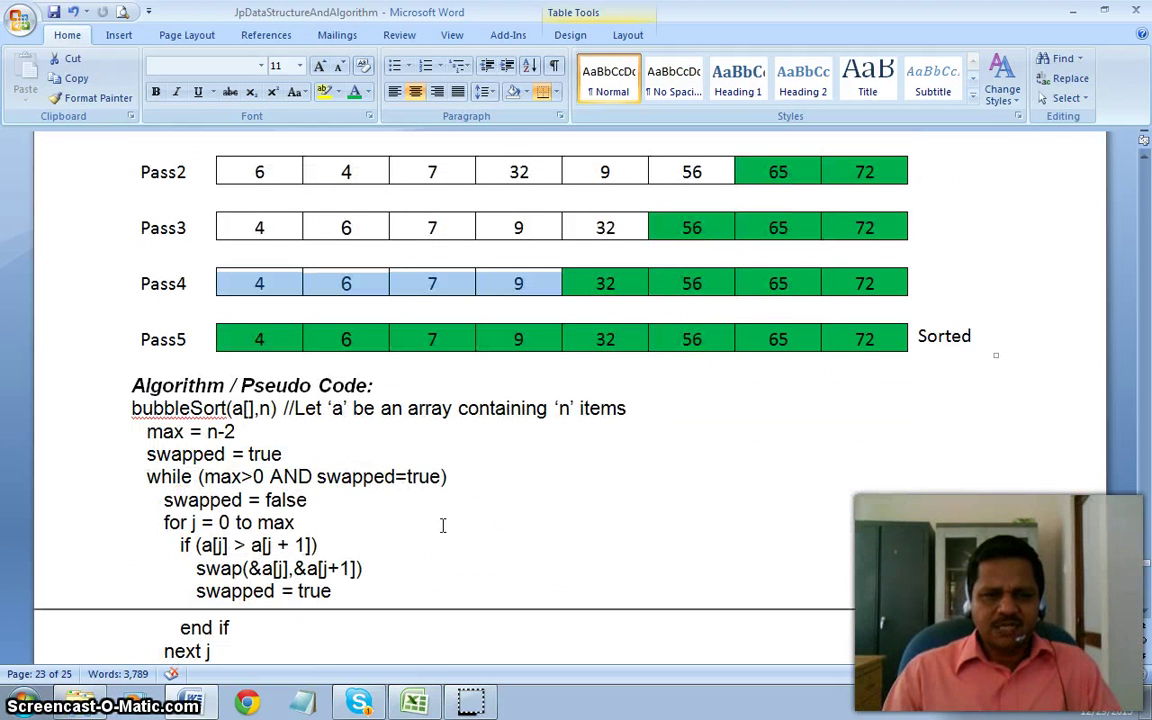
4 65 2 -31 0 99 2 83 782 1

-31 0 1 2 2 4 65 83 99 782

Bubble sort:

repeatedly move the largest element to the highest index position





Algorithm/pseudocode:

bubbleSort(a[],n) //let ‘a’ be an array containing ‘n’ terms

mx=n-2

swapped=true

while(max>0 AND swapped=true)

swapped=false

for j=0 to max

if(a[j]>a[j+1])

swap(&a[j].&a[j+1])

swapped=true

end if

next j

max=max-1

end while

C++ code:

void bubbleSort(int \*a, int n)

{

int j;

int max=n-2;

int swapped=1;}}

max--; }}

**Searching algorithm:**

Find a item among collection of items

**Linear Search algortihm:** examine first, second and so on

pseudo code:

int sequentialSearch(a[],n,t) //will return location of target in array a[] with n elements

for i=0 to n-1

if (a[i]=t) return I;

next i

return -1;

C++ implementation:

#include <iostream>

#include<conio.h>

int sequentialSearch(int \*a, int n, int t){

int i;

for(i=0;i<n;i++)

if (a[i]==t) return i;

return (-1); }

void main(){

int num[]={4,64,2,-31,0,99,2,83,782,1};

int n=10;

int t=99;

cout<<t<<”is found at”<<sequentialSearch(num,n,t)<<” “; }

output: 99 is found at 5

Binary search algorithm:

here the elements arranged in order(ascending) and saved in array, the basic algorithm is to find middle element of list and compare it against the key/target, decide which half of list contain key and repeat that half

pseudocode:

int binarySearch(a[],I,u,t) // returns location of t in array a[] fron index I to u

p=(i+u)/2;

while(a[p]!=t AND i<u)

if(a[p]>t)

u=p-1

else\i=p+1

p=(i+u)/2

end while

if(i<=u) return p

else return -1

C++ implementation

int binarySearch(int\*a, int i, int u, int t){

int p;

p=(i+u)/2;

while((a[p]!=t)&&(l<=u))

{ if (a[p]>t)

u=p-1;

else

i=p+1;

p=(l+u)/2; }

if(l<=u) return p;

else return -1; }

void main(){

int nun[]={1,2,7,9,50,99,100,150,190,200};

int n=10;

int t=99;

cout<<t<<”is found at”<<binarySearch(num,0,n-1,t)<<” “; }

output: 99 is found at 5

Recursive pseudo code:

int recBinarySearch(a[],I,u,t) //returb location of t in array a[] from index I to u

if i>u then

return -1

else

mid=(l+u)/2

if t=a[mid] then

return mid

else if t<a[mid] then

return recBinarySearch(a[],I,mid-1,t)

else return recBinarySearch(a[],mid+1,u,t)

end if

end if

C++ code:

int recBinarySearch(int \*a, int I, int u , int t)

{

int mid,

if(i>u) return(-1);

else {

mid=(l+u)/2;

if(t==a[mid])

return mid;

else if (t<a[mid])

return recBinarySearch(a,I,mid-1,t);

else

return recBinarySerch(a,mid+1,u,t); }}

void main()

{clrscr();

int a[]={1,2,7,9,50,99,100,150,190,200};

int n=10;

int t=99;

cout<<t<<””is found at<<recBinarySearch(a,0,n-1,t)<<” “;

}