Data - Structures

Analysis of algorithm - Time complexity

→ Types of Analysis

i) Worst case analysis (O)
   ↳ Also called upper bound.
ii) Best case analysis Ω
   ↳ Also called Lower bound.
iii) Average case Analysis Θ
   ↳ prediction of running time of an Algorithm

Lower bound ≤ Running time ≤ upper bound.

→ How to compare two or more than two Algorithms.

i) Compare on the basis of execution time (time taken by machine to execute a piece of code) (Bad approach)

ii) Count number of statements execute at any input. // Bad approach

iii) Count number of instructions writen by a programer. Bad approach.

→ Good Approach:
Compare the algorithms in terms of time at any input (n)

e.g $T(n)$ - Time in terms of n i.e input size.

e.g $f(n) = 2n+100$ if ($n = 0, 1, 2, 3 \ldots$) what will be values of $f(n)$ at each n.

Scanned with CamScanner

The measure of efficiency or the approximation of $T(n)$ is called Asymptotic Analysis ~~of~~ or Asymptotic complexity.

Lec 3

Asymptotic Analysis.

Worst Case Analysis / Upper Bound

Notation: O (Big Oh)
Equation:

$$f(n) = O(g(n))$$

$f(n)$ is big oh of $g(n)$

$$f(n) \leq c\, g(n) \qquad n \geq n_0$$

- cost
- positive (Can be in fraction)

~~Bad~~ Best Case Analysis / Lower Bound.

Notation: Ω (Big Omega)

Equation: $f(n) = \Omega\, g(n)$

f_n is big omega of $g(n)$

$$f_n \geq c\, g(n) \qquad n \geq n_0$$

Average case analysis / Tight Bound.

Notation: Θ (Big theta)
Equation: $f_n = \Theta(g(n))$ $f(n)$ is the big theta of $g(n)$

$$c_1 g(n) \leq f_n \leq c_2\, g(n)$$

$$n \geq n_0$$

Scanned with CamScanner

→ Step count or finding Asymtotic time complexity of an algorithm

e.g

```
for(int i=0, i<n, i++) n+1
{
    cout<<"inside loop\n";n
    cout<<"Last line\n";n
}
```

```
for
for(int i=0, i<n, i++)
{
  for(int j=0; j<m; j++)
  {
    cout<<"okay\n"
  }
}
```

$T(n) / O(n) = (n+1)+n+n$

$= n+1+n+n$

$= 3n+1$

$O(n) = 3n+1$

$\lg n$

n

$n \lg n$

n^2

n^3

2^n

$f(n) = 3n+4$

$g(n) = n$

$3n+4 \leq c(n)$

→ Stop Count or frequency count

```
{for (int i=0, i<n; i++)        [n+1]
{
    for (int j=0, j<m; j++)     [n(m+1)]
{
    Cout << "Testing\n";        [nm]
}
}
```

$T_{(n)} = n+1+nm+n+nm = 2nm+2n+1$
$O(nm)$

Drabacks of Array Data structures

1) Fix Size

2) Can not add single block in existing array

3) Cannot remove single blocking from existing array.

4) Wastage of memory

5) Contigous allocation.

Linked List.

Scanned with CamScanner

Linked List (Singly Linked List)

Implementation in C++

```
-> Class Linkedlist;

Class Node
{
    friend class Linkedlist
  int data;
  Node * next;
};// end of class

Class Linkedlist
{
    Node * head
public:
  Linked-list ()
{
    head = nullptr;
  // head = null
  // head = NULL
}
  bool insertAt Start (int val)
{
    Node* temp = new Node
    temp -> data = val;
    temp -> next = head;
    head = temp;
    return 1;
} // end of insert Astart.

bool delete from Start ()

{ ~~return~~ 0;   if (head == null)
                  { return 0;
                  }
```

```
else
{
    Node * curr=head.
    head = curr->next;
    // head = head->next;
    delete curr;
    curr=0
    return 1;
}
} // end of delete from start.

void display()
{   Node * curr = head;

{   while(curr != null)
    cout<<curr->data<<" ";
        curr = curr->next;
}
  } // end of display

  bool insertAtEnd(int val)
{
    Node* temp = new Node;
    temp->data = val;
    temp->next = null;
    Node *prev=0, curr = head
{   while(curr != null)
    {
        prev = curr;
        curr = curr->next
    }
    // if element is first in list.
```

Scanned with CamScanner

```
if(prev=0)
{ head = temp;
    return 1
}
else
{     prev -> next = temp;
      return 1;
}
}end of insert atfirst.
```

Continuation of Lot # 5

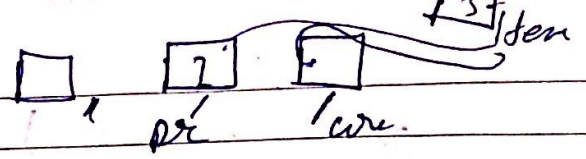
```
bool deleteFromEnd()
{
    if(head == 0)
    {
        cout << "List is Empty";
        return 0;
    }
    else
    {
        Node *curr = head, *last = 0,
        *secondLast = 0;
        while (curr != NULL)
        {
            secondLast = last;
            last = curr;
            curr = curr -> next;
        } // end of while

        if(SecondLast == 0)
        {
```

```
{
    head = NULL;
    delete last;
    last = 0;
    cout << "Last Block deleted";
    return 1;
}

else
{
    SecondLast->next = NULL;
    delete Last;
    Last = 0;
    cout << "Last block deleted";
    return 1
}
  }// end of external else
} // end of deleteFromend.

// Destructor

~LinkeList()
{
  while( head != NULL)
  {
    Node *curr = head;
    head = curr->next;
    delete curr;
    curr = head;
  }
  } // end of destructor.
} ; // end of Linked List class.
```

Scanned with CamScanner

```
void main( )
{
    LinkedList l1;

l1.insertAtStart (6);
l1.insertAtEnd(7);
l1.deleteFromEndStart();
l1.deleteFromEnd();

} // end of main( )
```

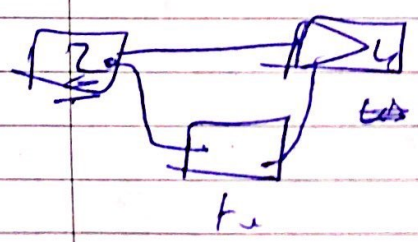
Scanned with CamScanner

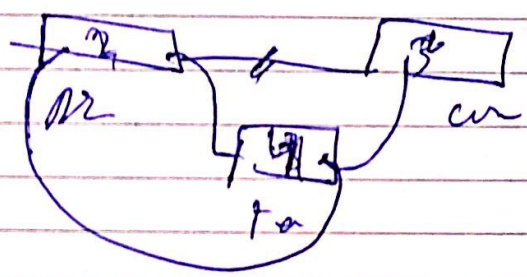


Lec #7

→ Write a function for LinkedList that inserts in a sorted way.

```
void SatedArr (int val)

    Node *tem = new Node
        tem->data = val
    if(head == 0)
{
        hed = temp;

}
    Node *pre = 0, ~~*last = 0,~~ *curr = head

    while (curr != 0)
{
        pre = curr
        curr = curr->next
    if (pre-> data < tem->data)

    {       ~~tem->next = pre->next~~ pre->next = temp;
            tem = ~~pre~~ curr;
    }
    else {
            tem->next = pre;
            tem = hed;
    }
}
```



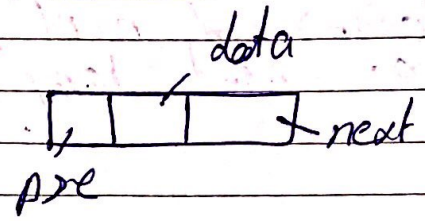


```
if (temp->next > pre->next)
{
    temp->next = null;
    pre->next = temp;

}

}
```

COPY

Scanned with CamScanner

Doubly Linked list.




```
Class Doublylinkedlist;
Class Node
{
    friend Doublylinklist;
    int data;
    Node *next, *pre
};
Class DoublyLinkedlist.
{
    Node * head;
publink.

DoublyLinkedList( );
DoublyLinkedlist( );
bool insertAtStart(int val);
bool insertAtEnd(int val);
bool Sorted insert (int val);
bool deletefromStart( );
bool dele from End( )
Node* Search(int val);
bool delespecific value (int val);
void display( );
```

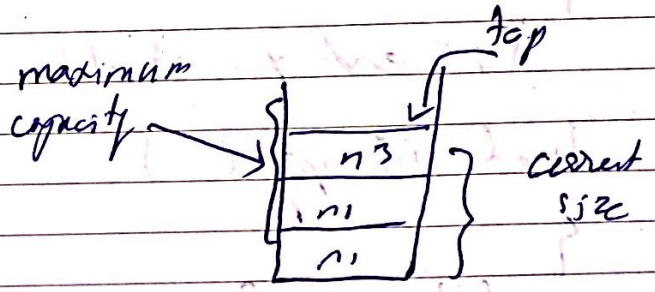
Scanned with CamScanner

What if we want to make a circular singly or doubly linked list ??

Ans:-

STACK

Last-In First-out




```
class Stack
{
    int maxSize;
    int curSize;
    int top;
    int * arr;

public:
    Stack (int n=10)
{
    maxSize = n;
```

```
    curSize = 0;
    top = -1;

    arr = new int [maxSize];
  }

~Stack ( )
{
    if (arr != 0)
    {
      delete [ ] arr;
      arr = 0;
    }
  } // end of ~stack

bool isEmpty ( )
{
  if ( curr Size == 0)
  {
      return 1;
  }
      return 0;

bool isFull ( )
{
  if ( cursize == maxSize)
  {
      return 1
  }   return 0;
```

Scanned with CamScanner

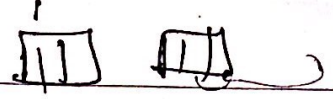
```
bool pop();
bool push(int n);
};// end of class.
```

~~class pre -> pre~~

Continuation of stack.

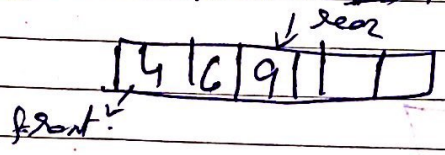
```
bool stack::push(int val)
{
    if (isFull())
        return false;
    else
    {
        top++;
        arr[top] = val
        curSize++;
        return true;
    }
}
bool stack::pop()
{   if (isEmpty())
        return false;
    else
    {
        top--;
        curSize--;
        return true; }
}
```

Scanned with CamScanner



Queue

Datastructures that stores data on first come first-serve or we can say Queue is first-In First-Out (FIFO)




```
Class Queue
{
    int front;
    int rear;
    int maxsize;
    int cursize;
    int ~~rear~~;
public:
    Queue (int n=10)
    {
        front = -1
        ~~rear~~
        rear = -1
        maxsize = n
        cursize = 0
        arr = new int [maxsize]
    }
    ~ Queue ( )
    {   if( arr! = 0)
        {   delete [ ] arr;
            arr = 0;
        }
```

Scanned with CamScanner

```
bool isFull( )
{
    if(curSize == maxSize)
        return true;
    return false;
}
bool ~~int~~ isEmpty( )
{
    if(curSize==0)
        return true;
    return false;
}

bool enqueue(int val)
{
    if (isfull())
        return false;
    else
    {
        rear=(rear+1) % maxSize;
        arr[rear] = val;
        curSize++;
        return true;
    }
}
```

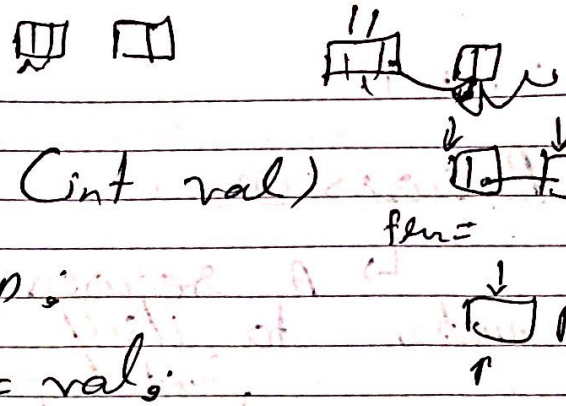
Scanned with CamScanner

212-5617 M.Ahmad

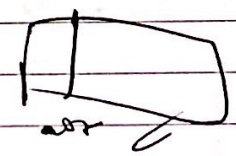
```
Class Queue;

Class Node
{
    friend class Queue;
    int data;
    int next;
};

Class qQueue
{
    Node *front, *rear;
public:
    Queue( )
{
    front = 0;
    rear = 0;
}
~Queue( )
  is empty( );
~~bool~~

bool instAtEnd(int val);
bool enqueue(in val);
```

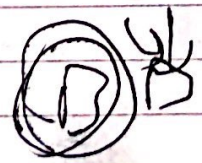
Scanned with CamScanner

fr = fr->nt




```
bool enqueue (int val)
{
    Node *temp;
    temp->data = val;
    temp->next = Null;
    if (a Empty() ) (rear == NULL)
    {
        front = temp;
        rear = temp;
    }
    else
    {
        rear->next = temp;
        rear = rear->next;
    }
}

bool dequeue()
{
    if (Front == Null)
        return false;
    else
    {
        Node *curr = front;
        front = front->next;
        delete curr;
        curr = 0;
    }
}
```

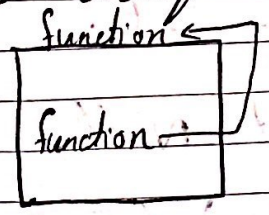




Scanned with CamScanner

Recurssion

↳ A recurssive call of a function to itself.



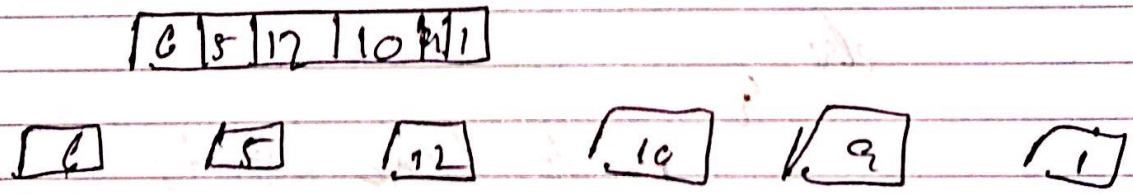
When to stop recurssion:

There should be a condition, called base condition, on which termination of recurssive call depends.

→ **Merge Sort:** Its a devid and conquer approch

↓

We divide ~~by~~ big problems and solved them separately, then merge them to get the output of given ~~prob.~~ problem.



Scanned with CamScanner

Merge Sort

Implementation

```
Void merge(int *arr, int l1, int l2, int r1
                                  , int r2) // header
                                  of helper funtion.

Void mergeSort( int * arr, int start, int end)
{
    if (start < end)
    {
        int mid = (start + end) / 2;
        mergeSort (arr, start, mid);
        mergeSort (arr, mid+1, end);
        merge (arr, start, mid, mid+1, end)
    }
}
// Definition of merge
Void merge(int * arr, int l1, int l2, int r1, int r2)

int *temp = new int [r2 - l1 + 1];
int i = l1, j = r1, k = 0;
while (i <= l2 && j <= r2)
{
```

```
if (arr[i] < arr[j])
{
    temp[k] = arr[i];
    i++;
} // end of if
else ~~if (arr[i] > arr[j])~~
{
    temp[k] = arr[j];
    j++;
}
k++;
} // end of while
// copying left side remaining element.
while(i<=l2)
{
    temp[k] = arr[i];
    i++;
    k++;
}
Copying right side remaining elements.
while(j<=r2)
{
```

```
        temp [k] = arr[j]
            j++;
            k++;
    }
// Copying element back to original array.
int start;
for (start = l1, k=0; start <= r2, start++, k++)
// element
{
    arr [start] = temp[k];
}
    delete [] temp;
    temp = 0;

} //end of merge
int main ()
{
    // Take size from user.
    // for arr, create and
    // intialize array. Display array.
    mergeSort( arr 0, n-1);
```

Scanned with CamScanner

Analysis of Merge Sort:

complexity of merge Sort complexity of merge.

$T(n) = 2T(\frac{n}{2}) + On \rightarrow k$

$T(1) = O(1)$ + Base case equation.

To find $T(\frac{n}{2})$ put $n = \frac{n}{2}$ in eq k we get

$T(\frac{n}{2}) = 2T(\frac{n/2}{2}) + O(\frac{n}{2})$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + O(\frac{n}{2})$

put $T(\frac{n}{2})$ in eq k we get

$T(n) = 2[2T(\frac{n}{4}) + O(\frac{n}{2})] + O(n)$

$= 4T(\frac{n}{4}) + \cancel{2}O(\frac{n}{\cancel{2}}) + O(n)$

$T(n) = 2^2 T(\frac{n}{4}) + O(n) + O(n)$

Let's find $T(\frac{n}{4})$, put $n = \frac{n}{4}$ in eq k we get

$T(\frac{n}{4}) = 2T(\frac{n/4}{2}) + O(\frac{n}{4})$

$T(\frac{n}{4}) = 2T\frac{n}{8} + O(\frac{n}{4})$

~~put T~~ $\cancel{\frac{n}{8}}$

put $T(\frac{n}{4})$ in eq we get.

Scanned with CamScanner

$$T(n) = 2^2\left[2T\left(\frac{n}{8}\right) + O\left(\frac{n}{4}\right)\right] + 2\,O(n)$$

$$= 2^2\,2T\left(\frac{n}{8}\right) + 2^2\,O\left(\frac{n}{4}\right) + 2\,O(n)$$

$$= 2^3\,T\left(\frac{n}{8}\right) + O(n) + 2\,O(n)$$

$$= 2^3\,T\left(\frac{n}{2^3}\right) + 3\,O(n)$$

$$= 2^k\,T\left(\frac{n}{2^k}\right) + k\,O(n) \rightarrow k_2$$

Eventually $\frac{n}{2^k}$ array size reaches to 1

So $\frac{n}{2^k} = 1$ $\quad n = 2^k$

$\log n = \log 2^k$

$\log n = k \log 2$ $\quad = \log 2 = 1$

$\log n = k$

lets rewrite eq (k2)

$n\,T(1) + \log O(n)$ $\quad \because T(1) = O(1)$

$n\,O(1) + \log n\,O(n)$

$O(n) + O(\log n \cdot n)$

OR

$\cancel{\Theta n} = \Theta(n) + \Theta(n \lg n)$

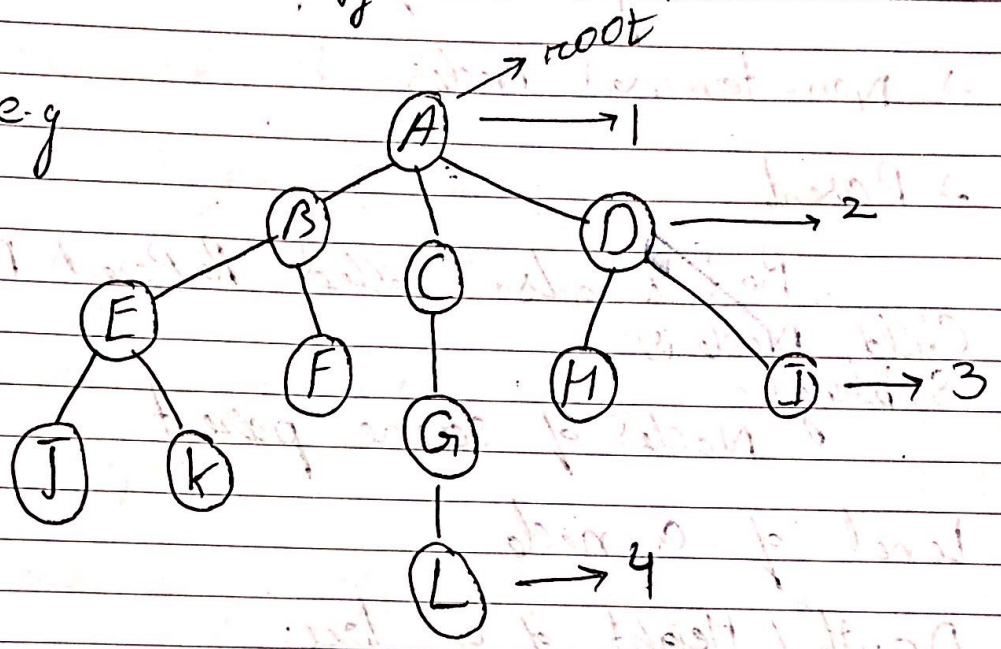
$$\underbrace{T(n) = \Theta(n \lg n)}_{\text{Best case / worst case}}$$

Scanned with CamScanner

TREE

Some Termenology related to Tree

e.g



→ Root of Tree

→ Degree of a Node

no. of subtrees of node

e.g. Degree of A is 3

→ Degree of tree

maximum of all the degree of nodes

e.g Degree of Tree in Example is ?

Scanned with CamScanner

Leaf Node(s) / Terminal Node(s)

Node(s) having 0 degree

-> Non-terminal Nodes

-> Parent

Root is also called Parent Node.

-> Child Node(s)

-> Siblings:

Nodes of same parent.

-> Level of a node

-> Depth / Height of a tree:

maximum level at which node exist.

-> Binary Tree(s)

Tree in which a node can have at most 2 child nodes

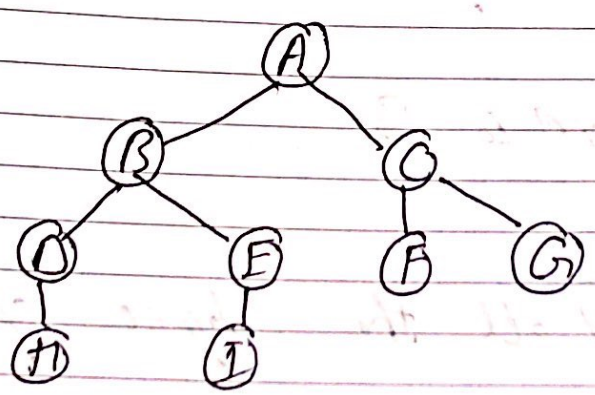


Diagram 1

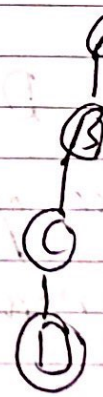


Diagram 2

Scanned with CamScanner

e.g Diagram 1 is Complete Binary Tree

→ Skewed Binary Tree.

Diagram 2 is left Skewed Binary Tree.

→ max no. of nodes at a specific level (k)

2^{k-1}

→ max no. of nodes in a binary Tree.

$2^{k}-1$

→ Full Binary Tree:

Binary Tre with max no. of nodes.

→ Ancestor

Removal of first node transforms the other sub-trees in forest.

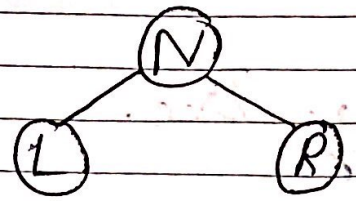
COPY

Scanned with CamScanner

Lec 15

Tree Traversal

e.g



→ In Order Traversal

LNR

→ Pre Order

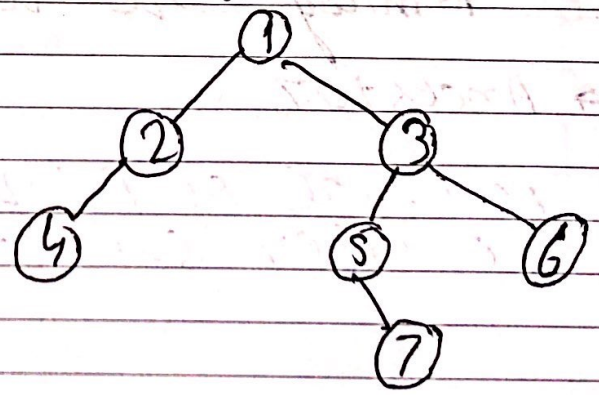
NLR

→ Post Order

LRN

→ Implementation of Binary Tree.

Using array

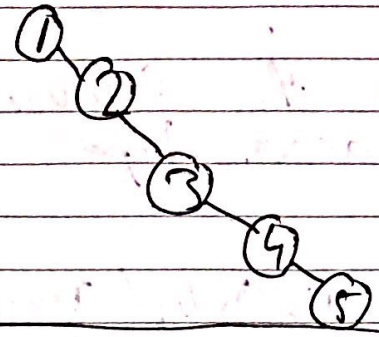


Root: will be at index 1

Left Child $2i$;
Right child $2i+1$;
Parent : ~~$\lfloor i/2 \rfloor$~~ $\lfloor i/2 \rfloor$

Scanned with CamScanner

Drawback?



Solution.

Implementation using Linked List.

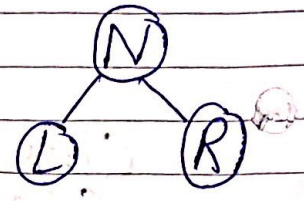
```
Class Biniary Tree;
{Class TreeNode
{
    friend class Biniary Tree;
    int data;
    TreeNode *left; *right;

};

Class Biniary Tree
{
    TreeNode * root;

Public:
BinaryTree ( )

{
    root = 0

}
Void inOrder( )

{
```

Lec 16

Continuation of Lec 15

Define the following function with private access, before inorder( )

```
Void inOrder( TreeNode *T)
{
    if (t != 0)
    {
        inOrder (t->left);
        cout << t->data << endl;
        inOrder (t->right);
    }
}
```



Binary Search Tree

→ A binary tree is BST if it has following:

1) Every node has unique value/key.

2) There should be smaller value than root value on left side and right side of tree should contain higher value then root value / key

3) Left and Right sub tree(s) should also be BST(s)

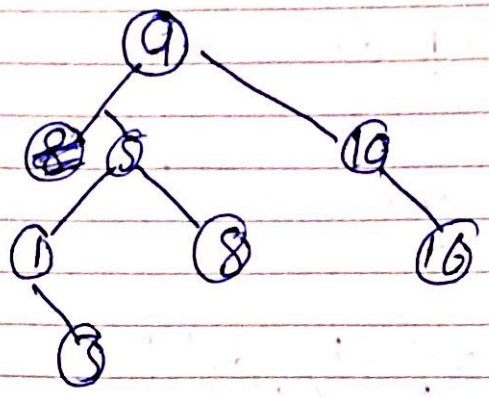
e.g:

1 5 9 8 3 10 16

9 → root.

Scanned with CamScanner

Draw BST



Implementation of BST:

```
Class BST;
Class TreeNode
{
    // Same as in Binary Tree
};
Class BST
{
    TreeNode *root;
public:
    BST();
    ~BST();
Void Search(int val)
```

Scanned with CamScanner

```
{
    TreeNode *t = root
    while (t != 0)
    {
        if ( t->data == val )
            return true
        else
            if (val < t->data)
                t = t->left.
            else if (val > t->data)
                t = t->right.
    } //end of wile leep.
    return false;
} // end of function.
```

Time complexity
$O(h)$ h is the height of tree

<u>Lec 17</u>

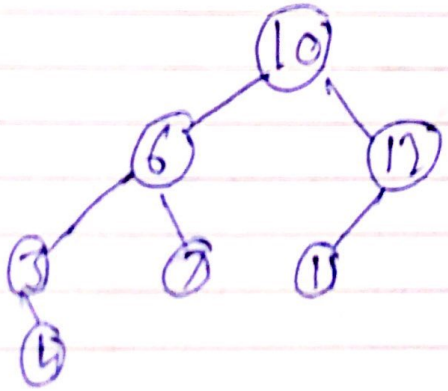
<u>Binary Search Tree</u> (BST)

→ Insert a node in BST

```
Void isert (TreeNode *t, int val);
Void insert (int val)
{
    insert (root, val);
} // Defination of insert (TreeNode *t, int val)
```

```
TreeNode
Void insert (TreeNode * t, int val)
{
    if (t==0)
    {
        t = new TreeNode;
        t->data = val;
        t->left = 0;
        t->right = 0;
        return t
} //end of if
else if (val < t->data)
{
    insert (t->left, val)
}
else if (val > t->data)
{
    insert(t->right, val);
}
} // end of insert.
```

→ Find minimum and maximum values of BST.



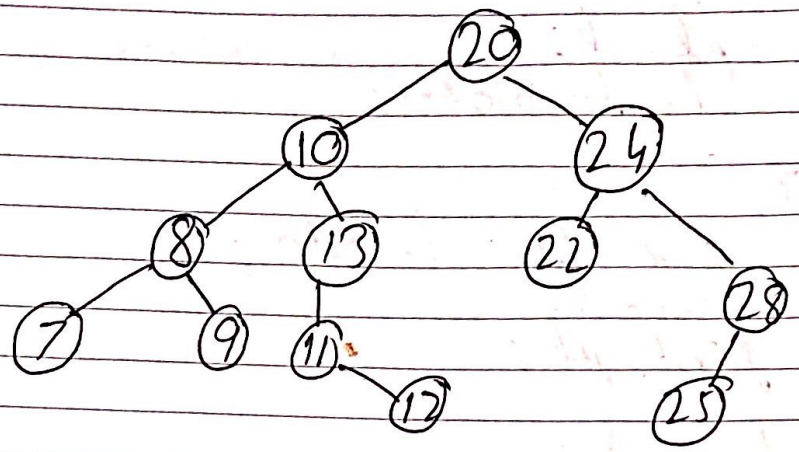
Scanned with CamScanner

```
TreeNode * minimum (TreeNode *t)
{
    while(t && t->left != Null
    {
        t = t->left
    }
    return t
}
```

Scanned with CamScanner

Lec 18

Binary Search Tree




```
-> bool remove (int val)
{
    TreeNode * curr = root, * parent = 0;
    while(curr!=0 && curr->data!=val)
    {
        Parent = curr;
        if (val < curr->data)
            curr = curr->left.
        else
            curr = curr->right
    }
    if( curr==0) // tree is empty -
                 // or value not found.
        return false;
    else
{       if(curr->left!=0 && curr->right!=0)
        {
```

```
TreeNode * l=curr->left, *pl=curr;
while ( l->right != 0)
{
    pl=l;
    l=l->right;
}
    curr->data=l->data;
    curr=l;
    parent=pl;
}// end of if
TreeNode * child;
if ( curr->left != 0)
    child=curr->left;
else
    child=curr->right;
if (curr==root)
{
  root = child;
}
else
{
  if(curr=parent->right)
    parent->right=child
  else
    parent->left=child;
}
```

Scanned with CamScanner

```
delete curr
curr = 0
return true;
} //end of external else
} //end of remove
```

Scanned with CamScanner

Data Structures

Height - Balanced Tree(s)

(AVL)

Problem with BST? $n = h$ (left or right skewed in BST)

height

no. of nodes

Solution:

Height-Balanced Tree(s) - AVL

→ Balancing Factor (Bf) of a node:

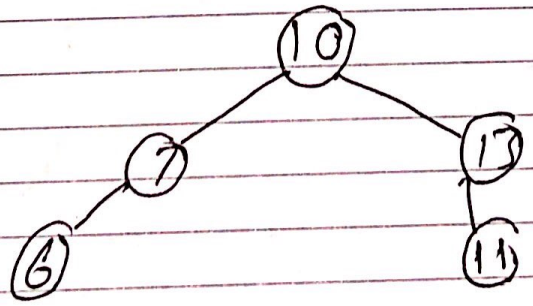
* $Bf = H_L - H_R$

Height of right subtree

Height of left sub-tree

values of Bf should be:
$-1, 0, +1$

AVL

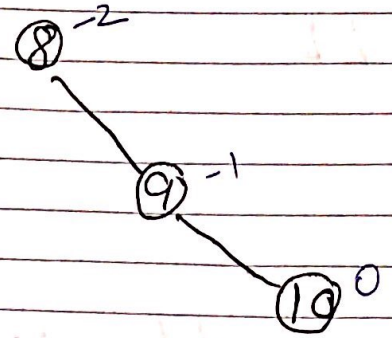


BST.
It is ballanced?

Ans. YES

Scanned with CamScanner

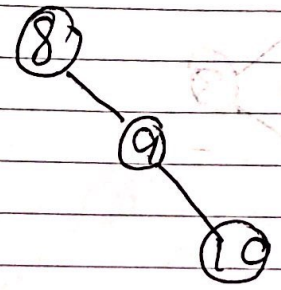
e.g



Is it ballenced?? Ans NO

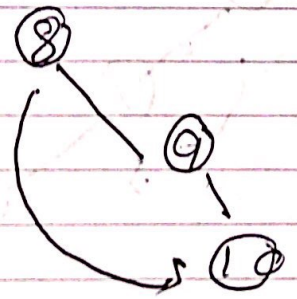
→ What steps are required to make a BST a ballenced tree by following rules of AVL.

Case 1 RR

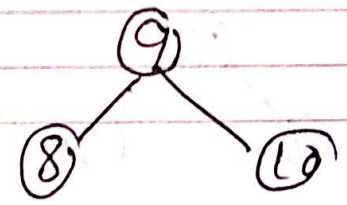


Solution.

Make an anti-clockwise rotation.

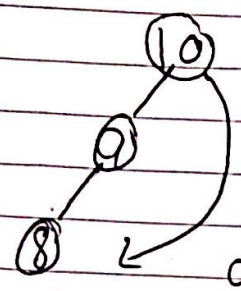


Result.



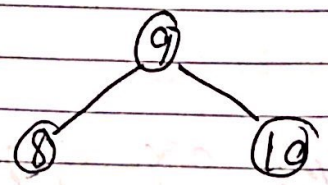
Scanned with CamScanner

Case 2 LL

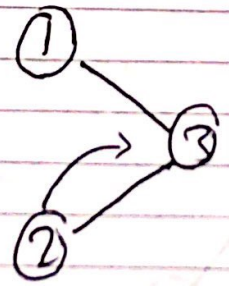


clockwise rotation required.

Result

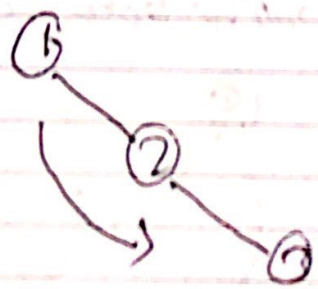


Case 3. RL

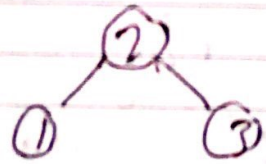


Step 1

Convert the tree in RR case.

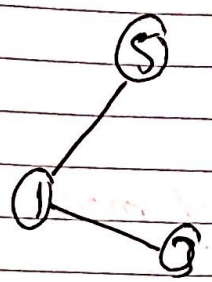


Result

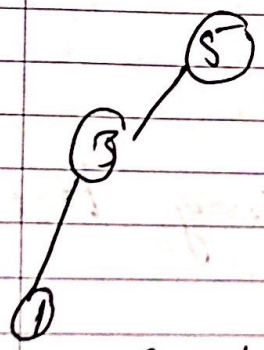


Scanned with CamScanner

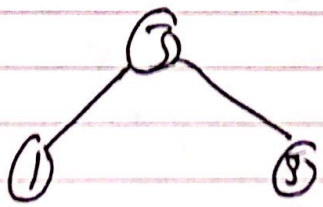
Case 4 LR



Step 1 Make it LL case.



Result



Scanned with CamScanner

Lec 20

Heap

Binary Tree in which parent is greater (in case of max Heap) than children

Properties of Heap:

1. structural property:

ACBT / CBT

Almost Complete Tree

Complete Binary tree.

2. Ordering property:

i) Max_Heap

parent > children

Insertion in Heap:

i) Adding values elements_by_Element.

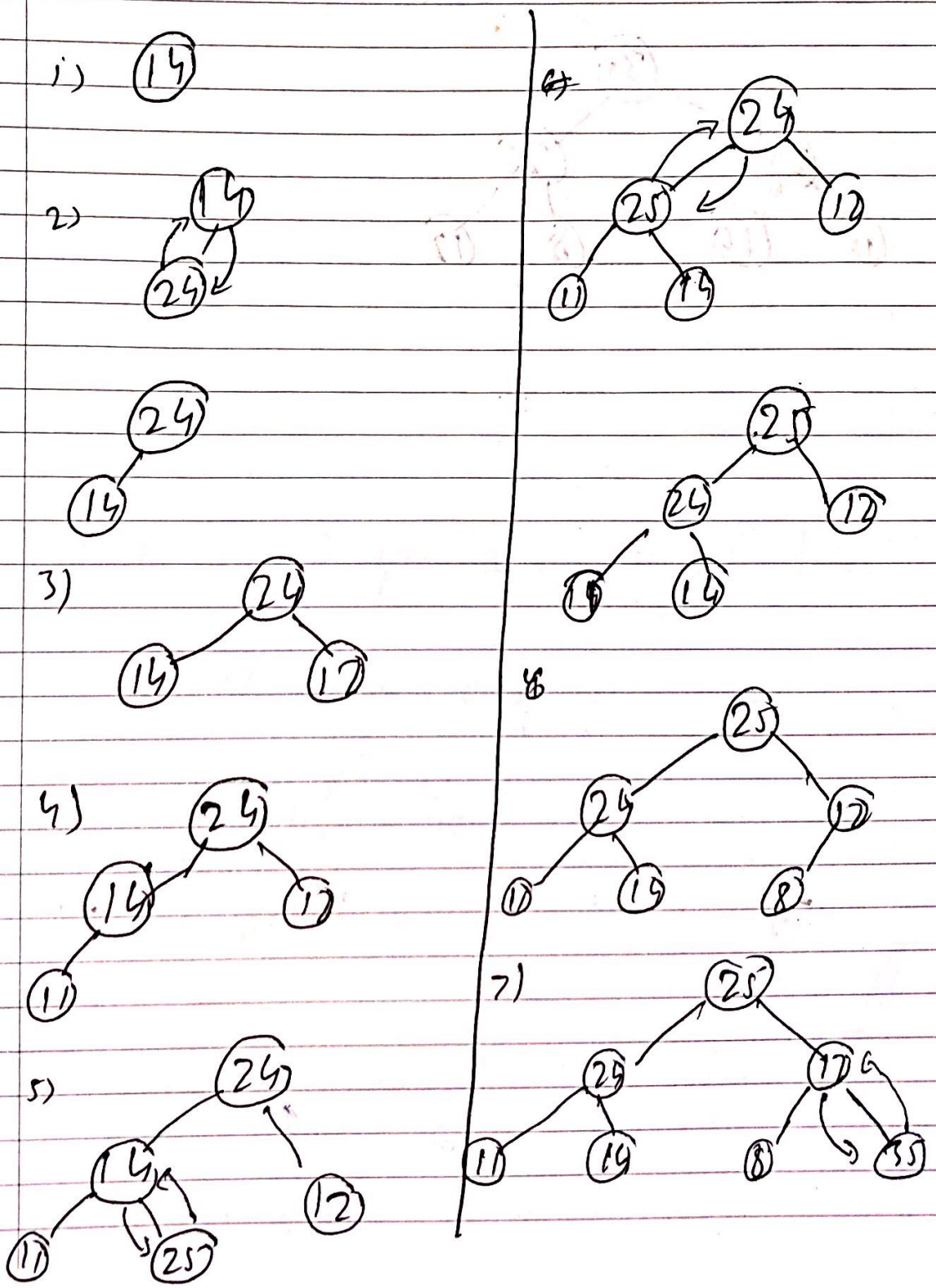
ii) Add all values without any specific order and then convert that into Heap (max_heap or min_Heap)

Scanned with CamScanner

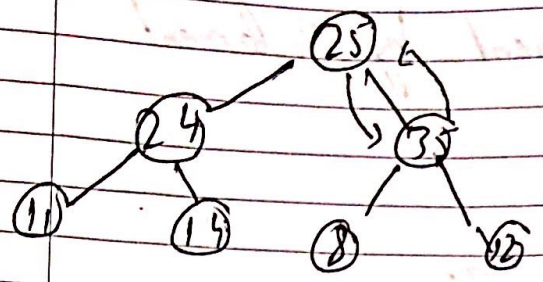
i) Adding values Element by Element

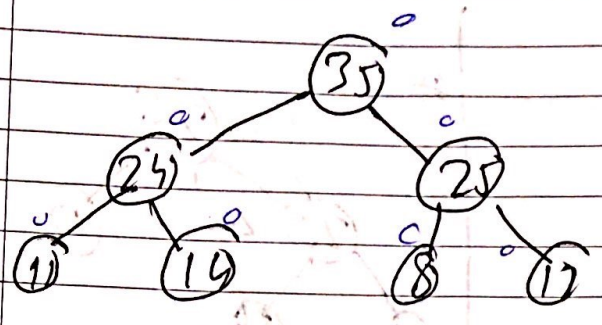
eg: (Max Heap)

14, 24, 12, 11, 25, 8, 35



Scanned with CamScanner





Scanned with CamScanner

Heap Sort

Working of Heap Sort

```
Void MaxHeap::removeMax(int &a)
{
    // Check of isEmpty
    a = h[1];
    h[1] = h[curSize]
    curSize--;
    heapify(1);
}

Void heapSort(int *arr, int n)
{
    MaxHeap mh;
```

COPY

Scanned with CamScanner

```
mh.buildHeap(arr,n);
for(int i=n-1, i>=0, i--)
{
    mh.removeMax(arr[i])
}
}
Void main()
{
 // Input array.
 // display array.
 heapSort(arr,n)
          |  \size
          name.
 // display array.
} endy min()
```

Lec #24

File/Code Compression Through Hoffman Coding Scheme

Generic Code Compression

file

a, b, c, d, e

Lets say:

Above file consists of different words. All the data of file has following letters/characters.

a, b, c, d, e

Letters	frequency/P (probability)	Code (Generic Method) (no. of bits Required)
a	0.1	
b	0.4	
c	0.2	
d	0.2	
e	0.1	
	1.0	

= (bits for each letter)(frequency) + ...

= 3(0.1 + 0.4 + 0.2 + 0.2 + 0.1)

= 3(1) = 3

Scanned with CamScanner

Now Lets find out the codes using Huffmen Method

Method 1:

a (0.1)	Sort in decendiny order →	b (0.4)	b (0.4)
b (0.4)		c (0.2)	c (0.2)
c (0.2)		d (0.2)	d (0.2)
d (0.2)		a (0.1) 0	a ae (0.2) 0
e (0.1)		e (0.1) 1	

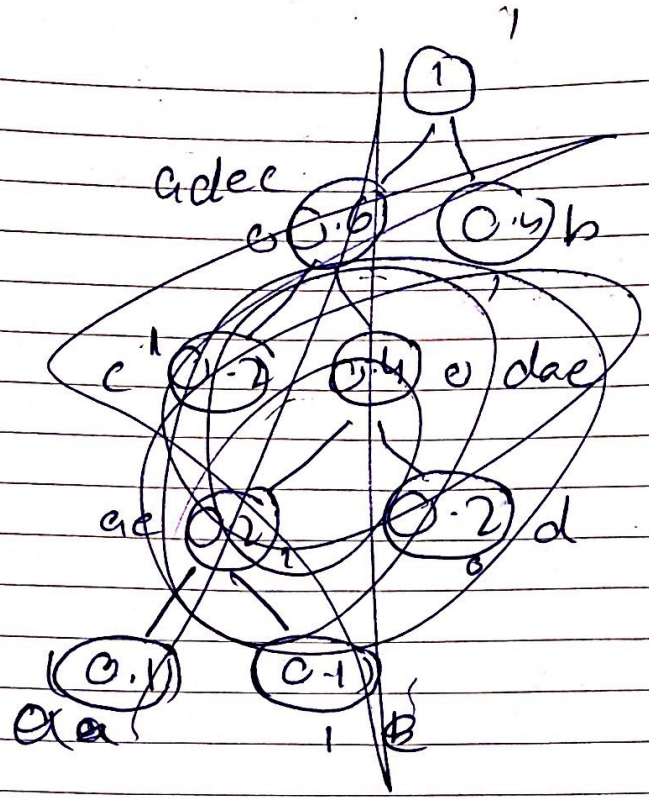
b (0.4)	b (0.4)		dace (0.6) 0
dae (0.4)	dae (0.4) 0		b (0.4) 1
c (0.2)	c (0.2) 1		

(dacb) 1

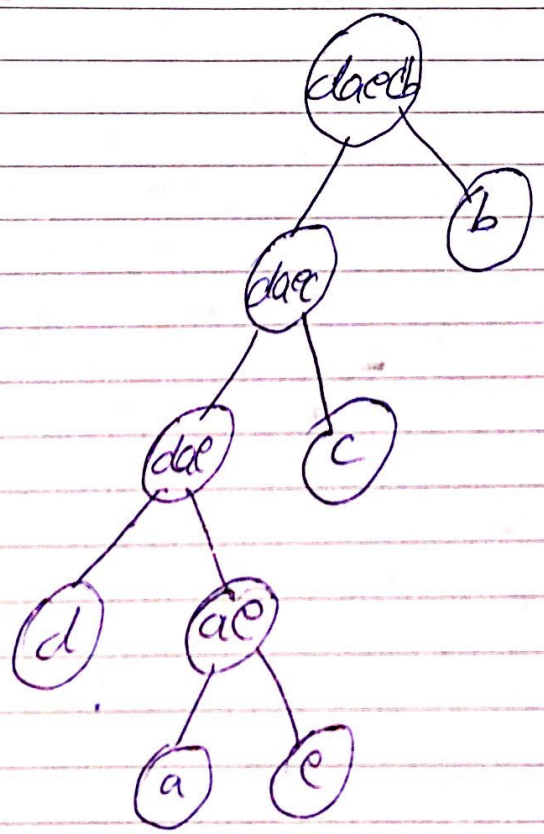
		Codes (using Huffmen Coding Scheme)
a		0010
b		01
c		000
d		0011
e		

$= 4(0.1) + 1(0.4) + 2(0.2) + 3(0.2) + 4(0.1)$
$= 2.2$

Scanned with CamScanner



a : 0 0 1 0
b : 1
c : 0 1
d : 0 0 0
e : 0 0 1 1



Scanned with CamScanner

Lec 23

Hashing and hash table

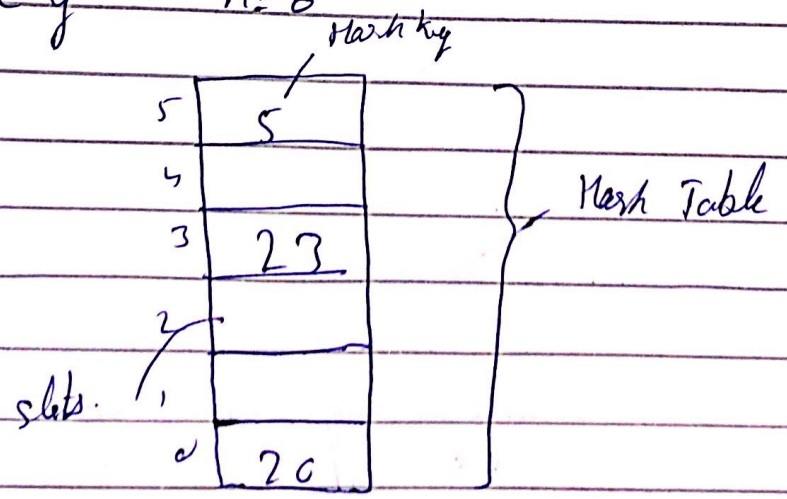
→ Purpose of Hashing:

To do insertion and searching or retrieval in constant time - O(1)

→ Some terminologies related to Hashing

i) key(s) / Search key(s)
ii) Hash function
iii) Hash Table

e.g $n = 6$



Mostly we use following Hash functions

$H(k) = (k \bmod 10, k \bmod n, etc)$

↑ key

Lets take:

$H(k) = (k \bmod 10)$

keys / Data = 23, 20, 5

$H(23) = 23 \bmod 10$

$H(23) = 3$

Similarly:

$H(20) = 0$, $H(5) = 5$

Issue:

Key = 3

$H(3) = 3$

We have already value at index 3. This is called collission.

Solution to collission

- chaining Methods (open Hashing)
  - AVL
  - Linked List
- openaddressing (Closed Hashing)
  - Linear probing
  - Quadratic probing
  - Double Hashing

Scanned with CamScanner

Lets correct the previous Collission using "Chaining Method"

n = 5

4	
3	
2	
1	
0	

Hash function - H(k) = k mode n
Keys / Data = 23, 20, 5, 3, 13
H(23) = 3, H(20) = 0, H(5) = 0,
H(3) = 3, H(13) = 13

4		
3	•	23 → 3 → 13
2		
1		
0	•	20 → 5

Scanned with CamScanner

Data Structures

Lecture - 24:

Open Addressing

OR

Closed Hashing

i) Linear Probing → no. of attempts.
ii) Quadratic Probing
iii) Double Hashing

i) Linear Probing:

Given Data:

$n = 10$

$H(k) = k \bmod n$

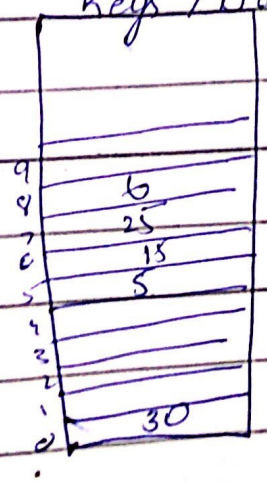
If collision:

$H'(k) = (k(k) + i) \bmod n$

i → no. of attempts.

$i = 1$

Keys / Data: 5, 10, 30, 25, 6



$H(5) = 5$ place

$H(15) = 5$ $H'(k) = (5+1) \bmod 10 = 6$

$H(30) = 0 \rightarrow$

$H(25) = 5 \rightarrow (5+1) \bmod 10 = 6$, $6 \rightarrow (5+2) = 7$

$H(6) = 6 \rightarrow (6+1) \bmod 10 \rightarrow 7 \rightarrow (6+2) \bmod 10 = 8$

Data	Probing Factor
5	1
15	2
30	1
25	3
6	3

Terms,

→ Primary Clustring
→ Secondary Clustring

ii) Quadratic Probing
Given data
$n = 10$
~~K K~~ $H(k) = k \bmod n$
If collision:
$H'(k) = (H(k) + i^2) \bmod n$
$i = 1$
Data: _____

iii) Double Hashing:

Given Data:

$n = 10$

$H(k) = (k) \bmod n$

If collision:

$H_1(k) = 10 - (k \bmod n)$

$H'(k) = (H(k) + j h_1(k)) \bmod n$

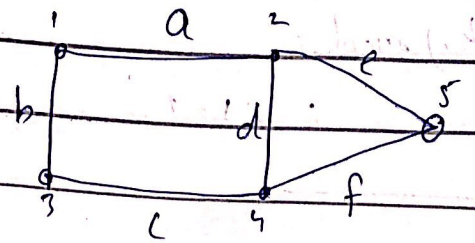
$= 1 \cdot 1 = 1$

Data: _____

Scanned with CamScanner

Data Structures:-

Graph / Graph Theory



Graph (Vertex, Edge) or G (V, E)

∴ Vertex or node:

$V = \{1, 2, 3, 4, 5\}$

⇒ order of graph = $|V|$

∴ Edge

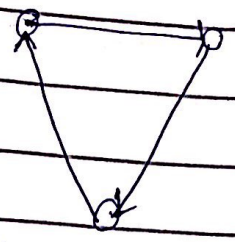
$E = \{a, b, c, d, e, f\}$

⇒ Size of graph = $|E|$

Types:

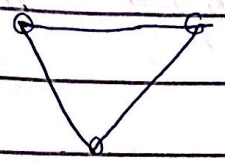
⇒ Directed Graph (has specific direction(s))

e.g:



Un-directed Graph:

e.g.



⇒ Adjacent Vertex: (Vertex with common edge)

e.g: $\{(1,2), (1,3), \ldots\}$ edge

$(1,4)^{\times}$

⇒ Adjacent Edge (Edge with common vertex)

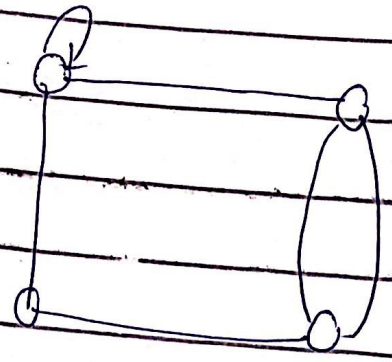
e.g $\{(a,b)\}$

→ Graph with self loop:

Pseudo Graph:

Having Self loop and Multipath.

Scanned with CamScanner



Multigraph

Having multipath but no self loop

→ Simple Graph:

No self-loop,

No, multipath.

Traversal of Graph.

Visit and Explore

i) BFS Scheme

ii) DFS

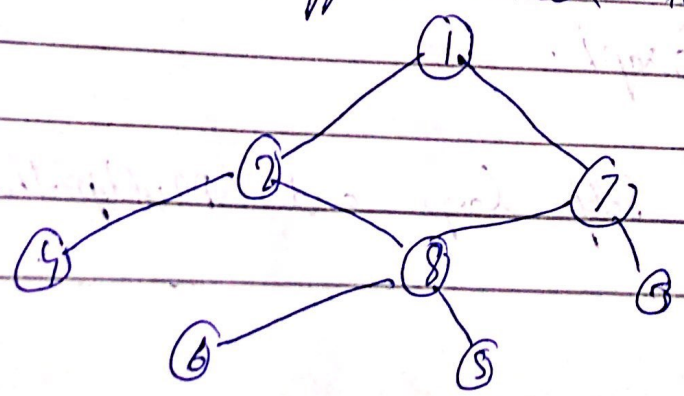
- BFS (Breadth - First - Search)

(No back tracking) or Level-wise traversal

→ DFS (Depth - First - Search)

(Supports back tracking)

e.g



BFS

Use Queue

1	2	7	4	8	3	6	5

DFS

Stack

~~1~~
~~8~~
~~2~~
~~1~~

1, 2, 4, 8, 6, 5, 7, 3

Scanned with CamScanner