Lecture - 2

Analysis of Algorithm - Time Complexity

→ Types of Analysis

- Worst case analysis O
  - ↳ also called upper bound
- Best case analysis Ω
  - ↳ also called lower bound
- 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:

- compare on the basis of execution time // bad approach
  - ↓ time taken by machine to execute a piece of code.
- count number of statements executed at any input // bad approach
- count number of instructions written by a programmer // bad approach.

→ Good Approach:

compare the algorithms in term of time at any input n

e.g $T(n)$

↓ time in term of n → input size

Scanned with CamScanner

e.g $f(n) = 7n + 100$
if $(n = 0, 1, 2, 3, \ldots)$ what will be the values of $f(n)$ at each n.

→ The measure of efficiency or the approximation of $f(n)$ is called Assymptotic Analysis / Assymptotic complexity.

Lecture-3:
Asymptotic Analysis

→ Worst case Analysis:
Notation: O (Big Oh)
Equation: $f(n) = O(g(n))$
$f(n)$ is big Oh of $g(n)$
$f(n) \leq c\,g(n)$
(+ive + floating point value)
•) cost.
•) positive value
(can be in fraction)
$n \geq n_0$
↓
base value
(+ive + integer)

→ Best case Analysis / Lower Bound:
Notation: Ω (Big Omega)
Equation: $f(n) = \Omega\, g(n)$
$f(n)$ is big omega of $g(n)$
$f(n) \geq c\,g(n)$
$n \geq n_0$

Scanned with CamScanner

Class code:
kxpsppc

→ Average case Analysis or Tight Bound:

Notation = θ (Big Theta)

Equation = $f(n) = \theta(g(n))$

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

$c_2 g(n) \leq f(n) \leq c_1 g(n)$

$n > n_0$

→ Step count or finding asymptotic time complexity of an algorithm

e.g

```
1- for(int i=0; i<n; i++)   n+1
2- {
3- cout<<"inside loop\n";   n
4- cout<<"last line\n";     n
5- }
```

$T(n) = (n+1) + n + n = 3(n)+1 = 3n+1$

$O(n)$

e.g

```
1- for(int i=0; i<n; i++)
2- {
3- for(int j=0; j<n; j++)
4- {
5- cout<<"Okay\n";
6- }
7- }
```

Scanned with CamScanner

Lecture - 4:

→ Step 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
  }
}
```

$T(n) = n+1 + \cancel{nm}\ nm + n + nm = 2nm+2n+1$

$O(nm)$

→ Drawbacks of Array Data Structure:

1) Fixed size
2) cannot add single block in existing array
3) cannot remove single block from existing array
4) Wastage of memory
5) Contiguous allocation

→ ~~Linked list~~

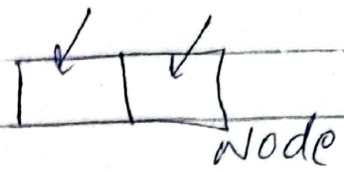
Scanned with CamScanner

→ Lecture 5:

Linked List

Implementation in C++

→ class linkedlist:




```
class Node
{
friend class LinkedList;
int data;
Node *next;
}; //end of class
class LinkedList
{
  Node *head;
public:
  LinkedList()
  {
    head = nullptr;
    // head = null;
    // head = NULL;
  }
bool insertAtStart(int value)
{
    Node *temp = new Node;
    temp->data = val;
    temp->next = head;
    head = temp;
    return 1;
} //end of class insertAtStart
bool deletefromStart
{
    if(head == null)
}
```

(Singly Linked List)

Scanned with CamScanner

```
else
{ Node * curr = head;
  head = curr -> next;
 // head = head -> next;
  delete curr;
   curr = 0;
   return1;
  }
 } //end of deletefromStart
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 ~~find~~ first in list
```

Scanned with CamScanner

```
if (prev == 0)
{
  head = temp;
  return1;
}
else
{
  prev -> next = temp;
  return1;
}
} //end of insertAtEnd
```

1bua76

→ Lecture 6:
Linked List/Singly Linked List
→ Continuation of Lec #5

```
bool deleteFromEnd()
{
  if (head == 0)
  {
    cout << "List is Empty\n";
    return 0;
  }
  else
  {
    Node *curr = head, *last = 0, *secondlast
while (curr != NULL)
{
  secondlast = last;
```

Scanned with CamScanner

```
last = curr;
curr = curr->next;
} //end of while
if (secondlast == 0)
{
head = NULL;
delete last;
Last = 0;
cout << "Last Block deleted \n";
return 1;
}
else
{
secondlast -> next = NULL;
delete last;
last = 0;
cout << "Last Block deleted\n";
return 1;
}
} //end of external else
} //end of deleteFromEnd.

// Destructor
~LinkedList()
{ while
(head != NULL)
{
Node* curr = head;
head = curr -> next;
delete curr;
```

Scanned with CamScanner

```
curr = head;
}  // end of destructor
};  // end of LinkedList class

void main()
{
LinkedList l1;
l1.insertAtStart(6);
l1.insertAtEnd(7);
l1.deleteFromStart();
l1.deleteFromEnd();
}  // end of main()
```

(v)

```
int i=0

t cur=hed

while (cur = 0)
if (cur->data == v)
    i+=1
cur= cur->next
}
if( i !=0)
```

Scanned with CamScanner

→ Lecture 7:

•) Write a function for Linked List that inserts in a sorted way

```
bool {insertElement (int value)
{
    Node *temp = new Node;
    Node *previous = ~~temp~~ ~~new Node~~;
    ~~pre~~ Node *curr = ~~new Node head~~
    if(head == 0)
      temp -> data = value;
      ~~temp -> next = head~~;
      head = temp;
      temp -> next = nullptr;
    else
      while(head != 0 && temp -> data < value
      {
        ~~temp -> next = previous~~.
        previous = curr;
        curr = curr -> next;
        ~~temp -> data = val~~
      }
      if(temp -> data > value)
      {
        ~~previous = temp -> data~~;
        ~~temp -> next = curr -> next~~;
        ~~if(temp -> data~~
        ~~curr = value~~;
        ~~if(curr~~
        temp -> next = previous;
      }     temp = head;
```

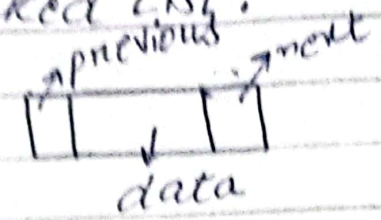
[2] [4]

2 4

→ Lecture 8:

→ Doubly Linked List:

Node:



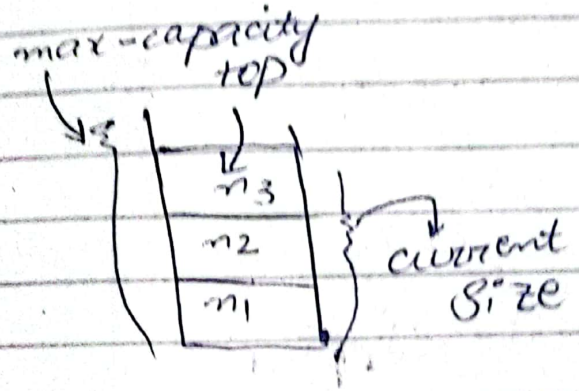

```
class DoublyLinkedList;
class Node
{
 friend DoublyLinkedList;
 int data;
 Node *next, *prev;
};
class DoublyLinkedList
{
 Node *head;
public:
 DoublyLinkedList();
~DoublyLinkedList();
 bool insertAtBeginStart(int val);
 bool insertAtEnd(int val);
 bool sortedInsert(int val);
 bool deleteFromStart();
 bool deleteFromEnd();
Node* search(int value);
 bool deleteSpecificValue(int value);
 void display();
};
```

curr → prev → prev

Scanned with CamScanner

→ What if we want to make a ~~calculator~~ circular singly or doubly linkedList ???

→ Stack
↓
Last-In First-Out




```
class Stack
{
  int maxSize;
  int currSize;
  int top;
  int *arr;
public:
  Stack (int n = 0)
  {
    maxSize = n;
    currSize = 0;
    top = -1;
    arr = new int [maxSize];
  }
~ Stack
```

Scanned with CamScanner

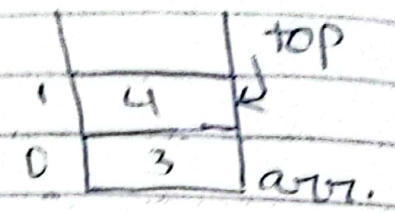
```
}
  if (arr != 0)
  {
  delete [ ]arr;
  arr = 0;
  }
}
bool isEmpty()
{
  if (currSize == 0)
  {
  return 1;
  }
  return 0;
}
bool isFull()
{
  if (currSize == maxSize)
  {
   return 1;
  }
  return 0;
  }
}
bool pop();
bool push(int m);
};
```

Scanned with CamScanner

→ Lecture - 9:

Continuation of Stack:

```
bool Stack :: push (int val)
{
  if (isFull())
  {
    return false;
  }
  else
  {
    top++;
    arr [top] = val;
    currSize++;
    return true;
  }
}
```

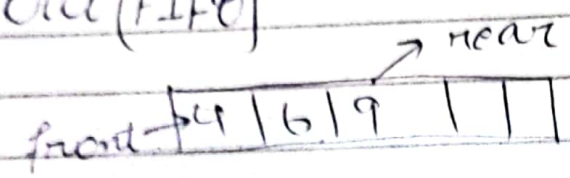



```
bool Stack :: pop()
{
  if (isEmpty())
  {
    return false;
  }
  else
  {
    top--;
    curr size--;
    return true;
  }
}
```

Scanned with CamScanner

→ Queue :

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



•) Implementation :

```
class Queue
{
    int front;
    int rear;
    int maxSize;
    int currSize;
    int *arr;
public:
    Queue (int n=10)
    {
        front = -1;
        rear = -1;
        maxSize = n;
        currSize = 0;
        arr = new int [maxSize];
    }

~Queue()
{
    if(arr != 0)
```

```
{
  delete [] arr;
  arr = 0;
}

bool isFull()
{
  if(currSize == maxSize)
    return true;
  return false;
}

bool isEmpty()
{
  if(currSize == 0)
    return true;
  return false;
}
// insert at end
bool enqueue(int val)
{
  if (isFull())
    return false;
```

Scanned with CamScanner

```
else
{
    rear = (rear + 1) % maxSize;
    arr[rear] = val;
    currSize ++;
    return true;
}
}
```

Scanned with CamScanner

Date: 22-9-22

→ Lecture 10:
Queue with Linked List

```
class Queue;
class Node
{
  friend class Queue;
  int data;
  Node * next;
};

class Queue
{
  Node * front, * rear;
public:
Queue()
{
  front = 0;
  rear = 0;
}
~Queue();
isEmpty();

// bool insertAtEnd(int val);

bool enqueue(int val)
{   Node *temp ; temp->data = val;
    if (rear == nullptr)
    {
      rear->data = temp->data;
      front->data = temp->data;
```

Scanned with CamScanner

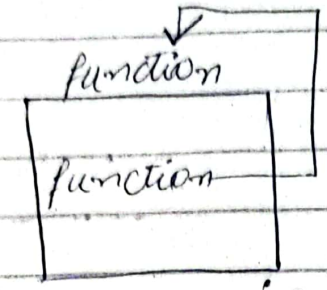
```
        front = temp;
        rear = temp;
        temp -> next = nullptr;
    }
else
    {
        while (rear != nullptr)
        {
            rear = rear -> next;
        }
            rear -> next -> data = temp ->
                                   data;
            rear -> next = nullptr;
        }

        rear -> next = temp;
        rear = rear -> next;
    }

// bool deleteFromStart( )
    bool dequeue()
    {
        Node *temp;
        if (front == nullptr)
        {
            cout << "List is Empty" << endl;
        }
else
        {
            temp ~~->next~~ = front ~~->next~~;
            front = front -> next;
            delete temp;
            temp = 0;
        }
```

Scanned with CamScanner

→ Lecture 11:

→ Recursion:

A recursive call of a function to itself.



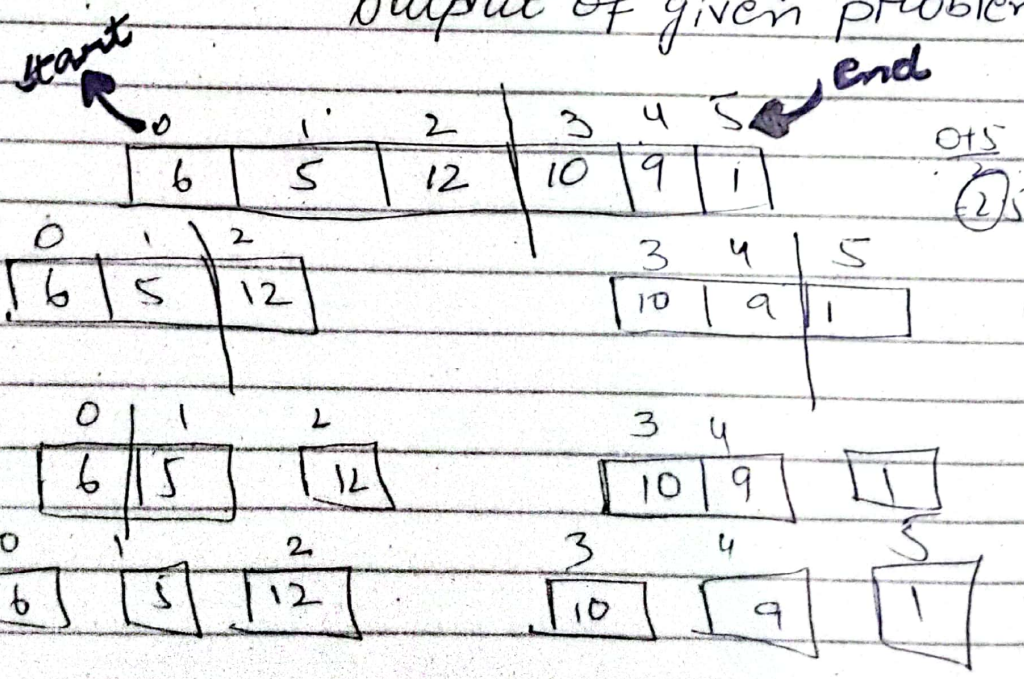
→ When to stop recursion:

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

→ Merge Sort:

It's a divide and conquer approach

We divide big problem into sub-program and solve them seperately, then merge them to get the output of given problem.



→ Lecture 12:

Merge Sort

Implementation:

```
//header of helper function
void merge (int *arr, int l1, int l2,
            int r1, int r2);

void mergeSort (int *arr, int start,
                int end)
{
  if (~~data~~ 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++;
    }
```

Scanned with CamScanner

```
else ~~if (arr[i] > arr[j])~~
{
  temp[k] = arr[j];
  j++;
}
k++;
} //end of while

// copying left side remaining elements
while(i<=l2)
{
  temp[k] = arr[i];
  i++;
  k++;
}

// copying right side remaining
elements

while(j<=r2)
{
  temp[k] = arr[j];
  j++;
  k++;
}

// copying elements back to
                    // orginal array
int start;
```

Scanned with CamScanner

```
for(start = l1, k=0; start<=r2;
                start++, k++)
{
 arr[start] = temp[k];
 }
delete []temp;
temp = 0;
} // end of merge

void main()
{
 // Take size from user
 // for array, create and initialiaze
 // array. Display array
 mergeSort(arr, 0, n-1);
 // display array again
 }
```

Scanned with CamScanner

→ Lecture-13:

← Analysis of Merge Sort →

complexity of merge sort

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad \text{— (K)}$$

complexity of merge

$T(1) = O(1)$ // base case equation

to find $T\left(\frac{n}{2}\right)$ put $n = \frac{n}{2}$ in eq (K),

we get

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

put $T\left(\frac{n}{2}\right)$ in eq (K), we get

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

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

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

$$T(n) = 2^2 T\left(\frac{n}{4}\right) + 2O(n) \quad \text{— (K1)}$$

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

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

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

Scanned with CamScanner

Put $T\left(\frac{n}{4}\right)$ in eq (11), we get

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

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

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

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

$$= 2^k T\left(\frac{n}{2^k}\right) + kO(n) \quad \text{— (12)}$$

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

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

$$n = 2^k$$

$$\log n = \log 2^k$$

$$\log n = k \log 2$$

$$\log n = k \qquad \because \log 2 = 1$$

Let's rewrite eq (12)

$$= nT(1) + \log O(n) \qquad \because T(1) = O(1)$$

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

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

OR

$$= O(n) \cancel{= \Theta} + O(n \log n)$$

Best case + Worst case

$$T(n) = O(n \log n)$$

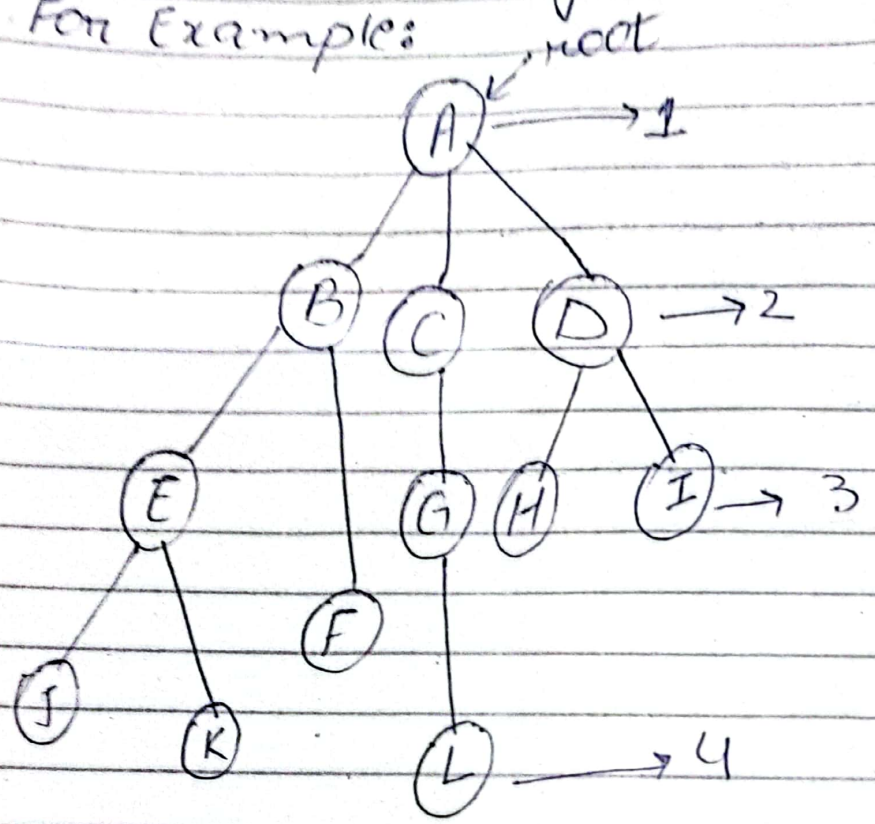
Scanned with CamScanner

→ Lecture 14:

Tree

→ Some Terminologies related to "Tree":

For Example:



→ Root of Tree:

→ Degree of a Node:

no. of sub-trees of a node.

For Example:

Degree of A is 3

→ Degree of Tree:

maximum of all the degree of nodes.

For Example:

Degree of Tree in Example is 3

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 exists.

→ Binary Tree(s):
Tree in which a node can have at most 2 child nodes.

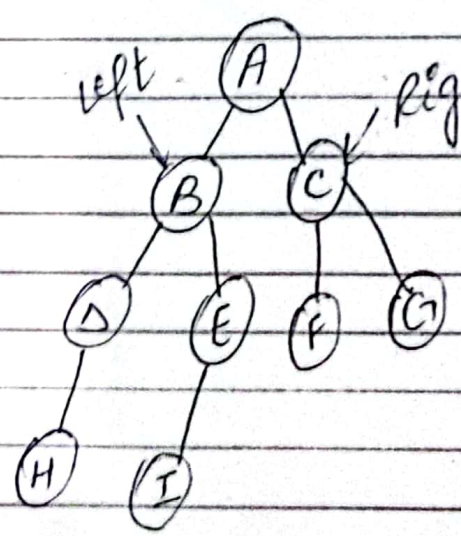


Diagram 1

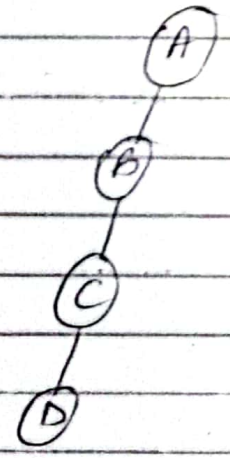


Diagram 2

Scanned with CamScanner

•) Removal of first node creates the transform sub-trees in forest.

→ Complete Binary Tree:
Binary tree having leaf nodes at adjacent level.
For Example:
Diagram 1 is complete binary tree.
~~Binary~~

→ Skewed Binary Tree:
Diagram 2 is left skewed Binary Tree.

→ Max no. of nodes at a specific level (K) $2^{K-1} = 2^{4-1} = 2^3 = 8$

→ Max no. of nodes in a Binary Tree $2^K - 1 = 2^4 - 1$
$= 16 - 1 = 15$

→ Full Binary Tree:
Binary tree with max no. of nodes.

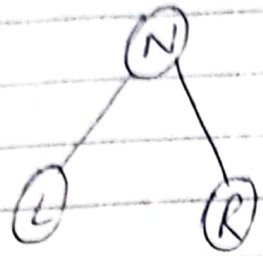
→ Ancestor

Scanned with CamScanner

→ Lecture - 15:

→ Tree Traversal:

e.g



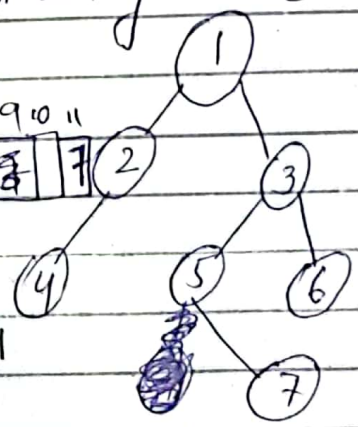
→ In-Order Traversal:
LNR / RNL

→ Pre-Order: NLR / NRL

→ Post-Order: LRN / RLN

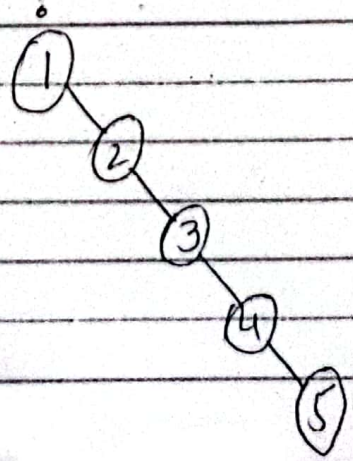
→ Implementation of Binary Tree using Array:

1	2	3	4	5	6	7	8	9	10	11
	1	2	3	4		5	6		~~7~~	7



→ Root: will be at index 1
Left Child: $2i$
Right child: $2i+1$
Parent: $\lfloor i/2 \rfloor$

Drawback?



Scanned with CamScanner

Solution:

Implementation using Linked List

```
class BinaryTree;
class TreeNode
{
 friend class BinaryTree;
 int data;
 TreeNode *left, *right;
 {

class BinaryTree
{
 TreeNode *root;
public:
  BinaryTree()
  {
    root = 0;
  }
  void inOrder()
  {
    inOrder(root);
  }
};
```

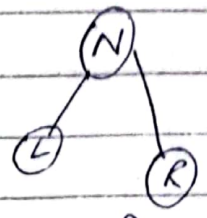
Scanned with CamScanner

→ Lecture-16:

Continuation of lecture-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);
```



LNR.

Binary Search Tree

→ A binary tree is BST if it has following

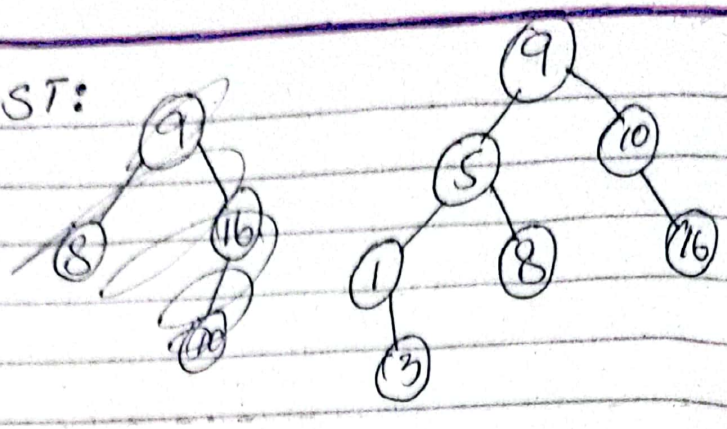
1) Every node has unique value/key.
2) There should be a smaller value than root value on left side and right side of tree should contain higher value than root value/key.
3) Left and right sub-tree(s) should also be BST(s).

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~~
  bool search(int val)
  {
      TreeNode *t = root
      while (t!=0)
      {
          if (t->data == val)
          {
```

Scanned with CamScanner

```
    return true;
}
else if (val < t -> ~~right~~ data)
    t = t -> left
else if (val > t -> data)
    t = t -> right;
} // end of while
return false;
} // end of function
```

$O(h)$ h is the height of tree.

Scanned with CamScanner

→ Lecture - 17:

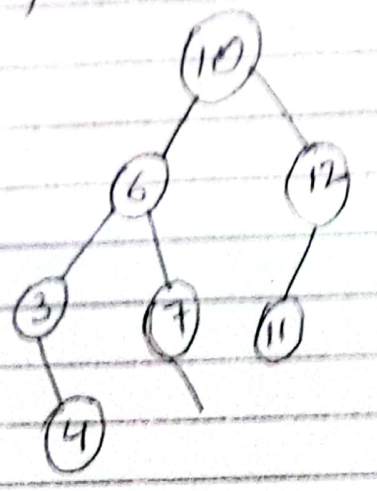
Binary Search Tree (BST)

→ Insert a Node in BST:

```
Private:
void insert (TreeNode *t, int val);
Public:
void insert (int val) {

    insert (root, val);
}

// Definition of insert (Node*t, int val)

void insert (TreeNode *t, int val)
{
    if (t == 0)
    {
        t = new TreeNode;
        t -> data = val;
        t -> left = 0;
        t -> right = 0;

    } // end of if
else if (val < t -> data)
{
    insert (t -> left, val);
}
else if (~~val~~ val > t -> data)
{
    insert (t -> right, val);
}
} // end of insert
```

Scanned with CamScanner

→ Find minimum and maximum value of BST :




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

Scanned with CamScanner

Lecture -19:-

n → no. of nodes
h → height

Date: Data Structures

Height-Balanced Tree(s)
(AVL)

→ Problem with BST?

→ Solution:

Height-Balanced Tree(s) - AVL

→ Balancing Factor (Bf) of a node:

$*Bf = H_L - H_R$

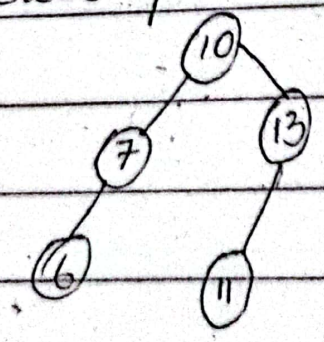
H_R ← Height of right sub-tree

H_L ↑ Height of left subtree

values of Bf should be:

-1, 0, +1

For Example:

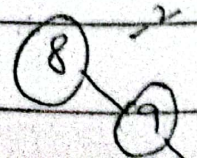


BST

Is it Balanced?

Answer: Yes

For Example:



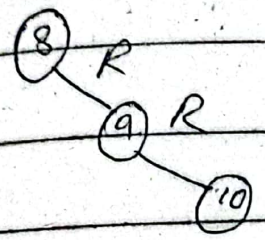
Scanned with CamScanner

Is it Balanced?

Answer: NO

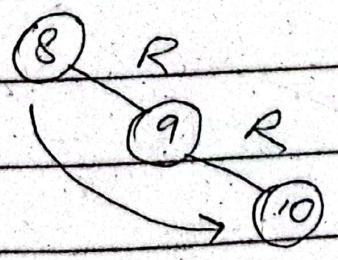
What steps are required to make as a BST a balanced tree by following rules of AVL:

Case-1: RR

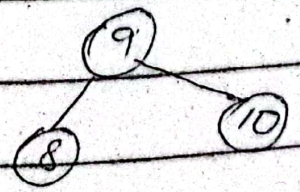


Solution:

Make an anti-clockwise rotation



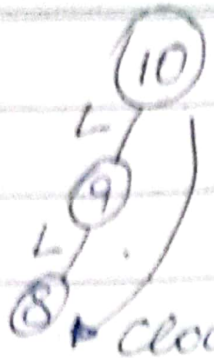
Result:



Case-2: LL

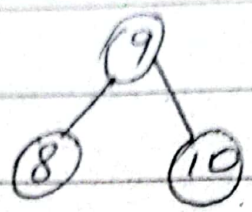
Scanned with CamScanner

Date:

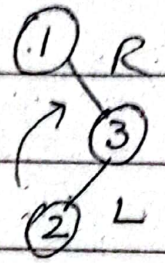


clockwise Rotation Required

Result:

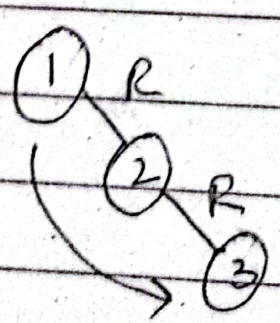


→ Case-3: RL



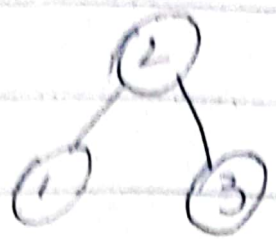
Step 1:

Convert the tree in RR case:

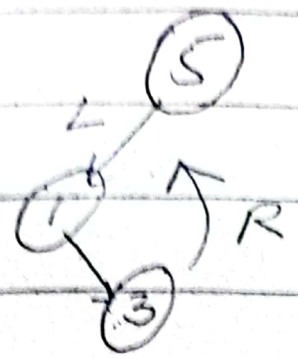


Scanned with CamScanner

Result:

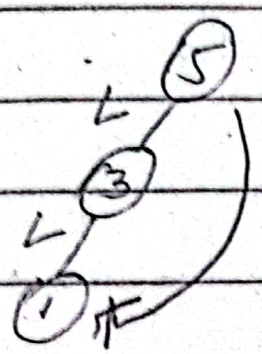


Case-4: LR

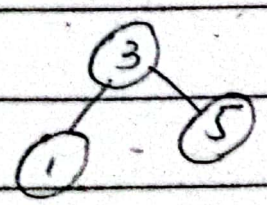


Step1:

convert the tree in LL case:



Result:



Scanned with CamScanner

Date: Data Structures

→ Lecture-20:

Heap

→ Heap:
Binary Tree
↑ BST in which parent is greater (in case of max Heap) than children or less (in case of min Heap) than children.

→ Properties of Heap:

•) Structural Property:

ACBT / CBT

Almost complete Binary Tree

complete Binary Tree

•) Ordering Property:

i) Max_Heap

parent > children

ii) Min-Heap

parent < children

→ Insertion in Heap:

Scanned with CamScanner

i) Adding values element-by-element

ii) Add all values without any specific order and then convert that into Heap (max-heap or min-heap)

Adding values element by element-by-element

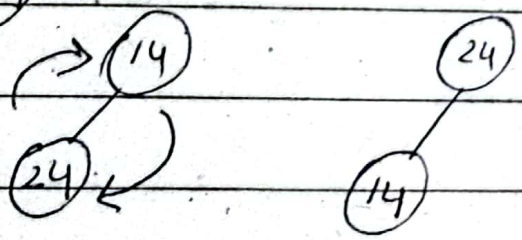
For Example:

(Max-Heap)

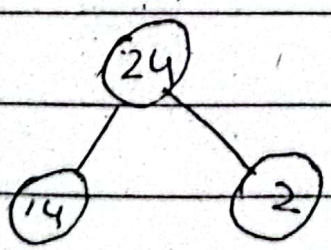
14, 24, 12, 11, 25, 8, 35.

1)

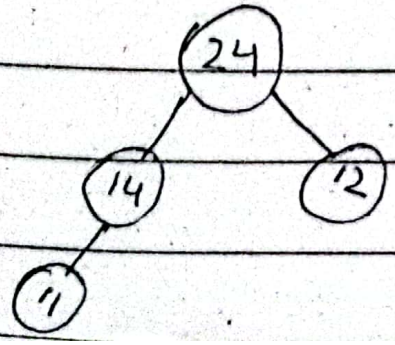
2)



3)



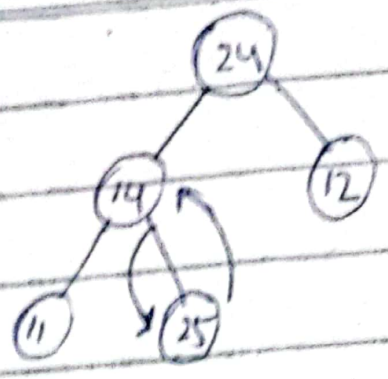
4)

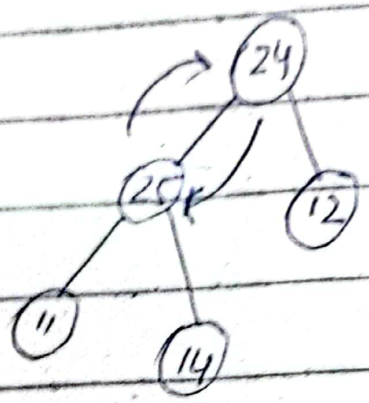


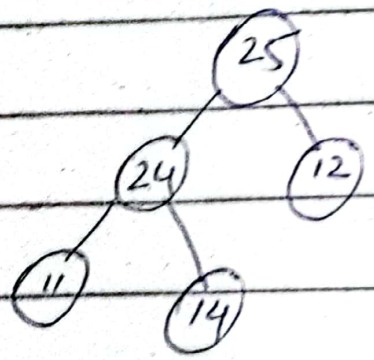
Scanned with CamScanner

Date:

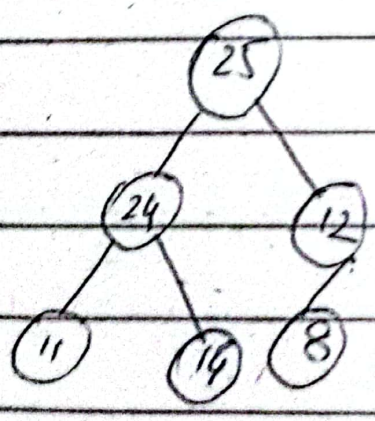
5)





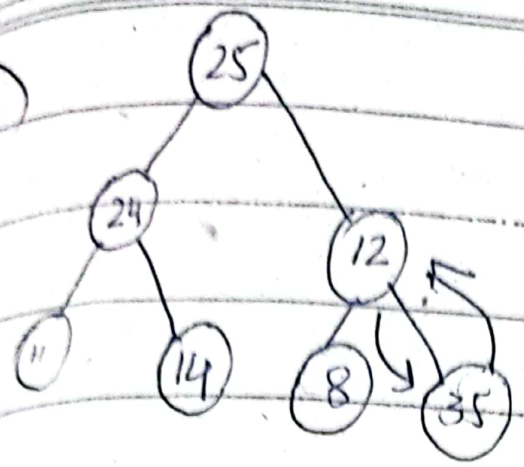


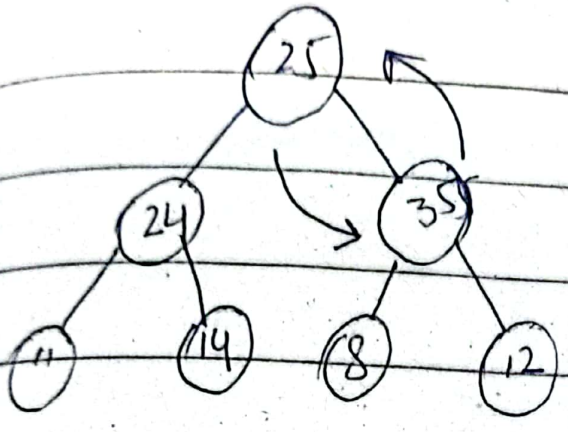
6)

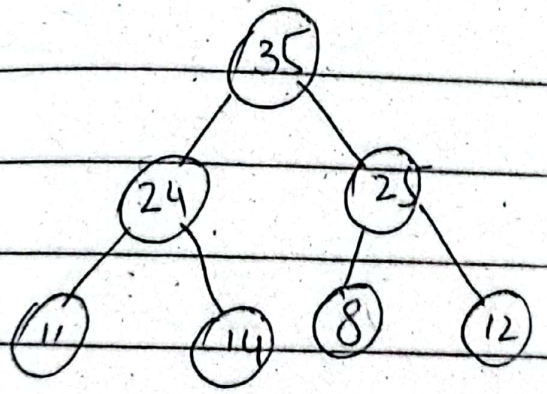


Scanned with CamScanner

5)







Scanned with CamScanner

Date: 5/1/2022 Data Structures

→ Lecture 21:

Heap Sort

Working of Heap Sort:

1	4	7	0

```
void MaxHeap::removeMax(int&a)
{
  // Check of isEmpty
    a = h[1];
    h[1] = h[currSize];
    currSize--;
    heapify(1);
  }
void heapSort (int *arr, int n)
{
  MaxHeap mh;
```

Scanned with CamScanner

15/11/2022

```
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 of array

name of array

```
// display array
} // end of main.
```

Scanned with CamScanner

Lecture 22

Date: 17/11/2022 Data Structures

File / Code compression Through Huffman Coding Scheme

→ Generic Code Compression:

file

a,b,c,d,e

Let's say:

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

a, b, c, d, e

letters	frequency/ probability
a	0.1
b	0.4
c	0.2
d	0.2
e	0.1
	1.0

no. of bits required

= (bits for each letter)(frequency)

= 3(1)

10/2022 - 3.0

(0.2 0.2 0.1)

...ode (Generic Method)

000

001

010

011

100

Now lets find out the codes using Huffman Method

Method 1:

a(0.1)
b(0.4)
c(0.2)
d(0.2)
e(0.1)

Sort in descending order → ~~New~~ Next Key

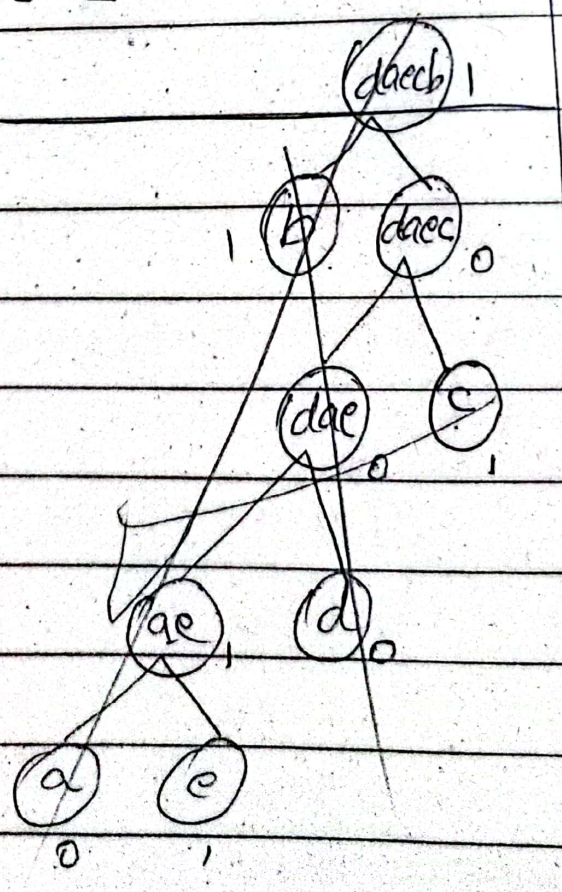
Date: 17/11/2022

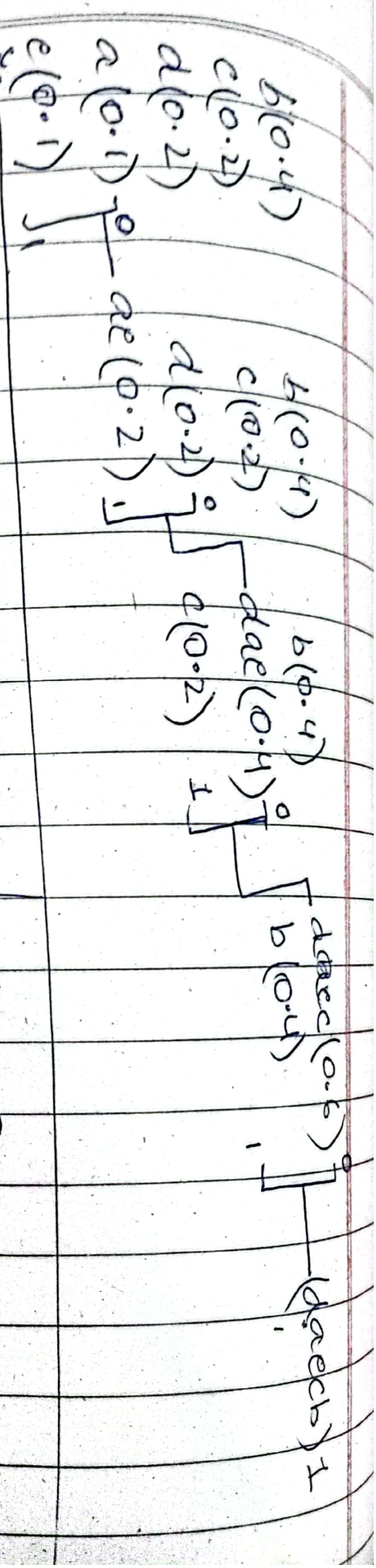
Codes (using) scheme Huffman coding scheme

a	~~0000~~ 0010
b	1
c	01
d	000
e	0011

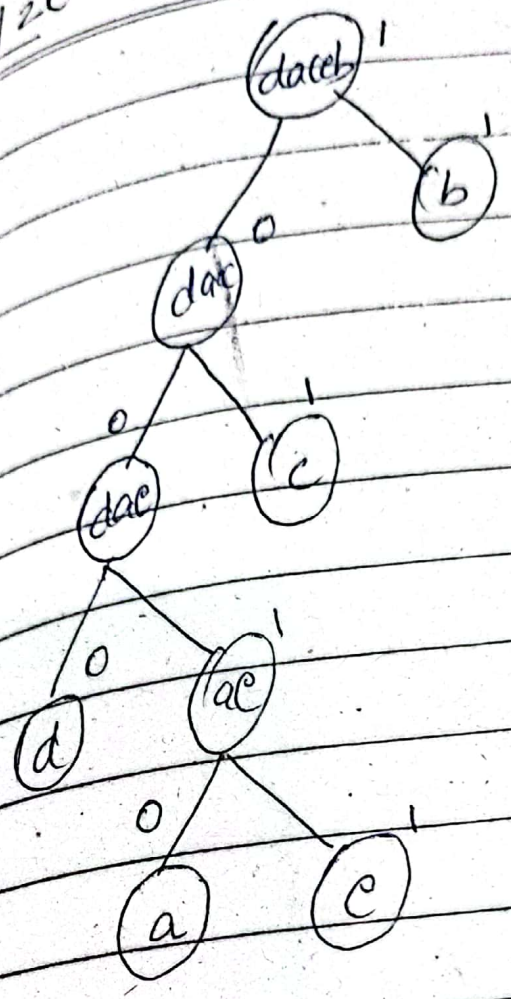
$= 4(0.1) + 1(0.4) + 2(0.2) + 3(0.2) + 4(0.1)$

$= 2.2$





12/11/2022



Task 1A / Task 2B / login

Date: 22/11/2022

Data Structures

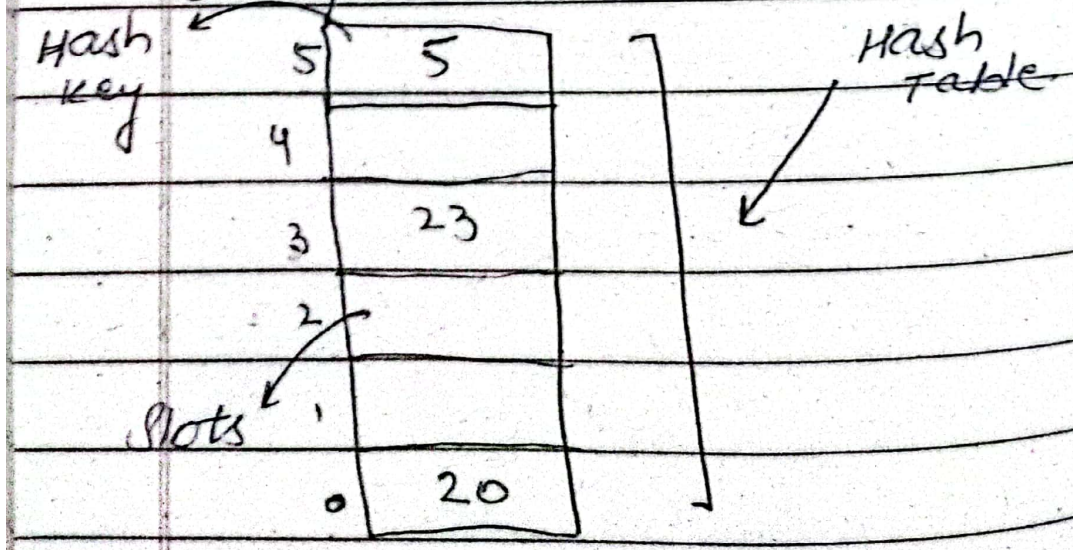
→ Lecture-23:
Hashing and Hash Table

→ 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

Example: n = 6



Scanned with CamScanner

22/11/2022

Mostly we use following hash function

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

key

Let's 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.

Scanned with CamScanner

Date: 22/11/2022

Solution to collission

- Chaining Method (Open Hashing)
  - AVL
  - Linked List
- Open Addressing (closed Hashing)
  - Linear Probing
  - Quadratic ~~Hashi~~ Probing
  - Double Hashing

Let's correct the previous collission using "Chaining Method"

4	
3	
2	
1	
0	

$m = 5$

Hash function - $H(K) = K \bmod n$

Keys/Data = 23, 20, 5, 13, 3

$H(23) = 3$, $H(20) = 0$,

$H(5) = 0$, $H(3) = 3$, $H(13) = 3$

Scanned with CamScanner

2/11/2022

4	
3	23 → 3 → 13
2	
1	
0	20 → 5

Scanned with CamScanner

24/11/2022

Data Structures

Lecture - 24:

Open Addressing / Closed Hashing:
Linear Probing → no. of attempts
Quadratic Probing
Double Hashing

Linear Probing:

Given Data: $n = 10$

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

If Collision:

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

$\therefore$ i is no. of attempts

$$i >= 1$$

Keys/Data: 5, 15, 30, 25, 6

Scanned with CamScanner

Date: 24/11/2022

9	
8	
7	6
6	25
5	15
4	5
3	
2	
1	
0	30

$H(5) = 5$ probe = 1

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

$H(30) = 0$

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

$(5+2) \bmod 10 = 7$

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

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

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

Terms:

→ Primary clustering

→ Secondary clustering

ii) Quadratic Probing:

Given Data:

$n = 10$

Scanned with CamScanner

24/11/2022

$H(K) = K \bmod n$

If collission:

$H'(K) = (H(K) + i^2) \bmod n$

$\because i >= 1$

Data:

Double Hashing:

Given data:

$n = 10$

$H(K) = K \bmod n$

If collission:

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

$H'(K) = (H(K) + iH_1(K)) \bmod n$

$\because i >= 1$

Data:

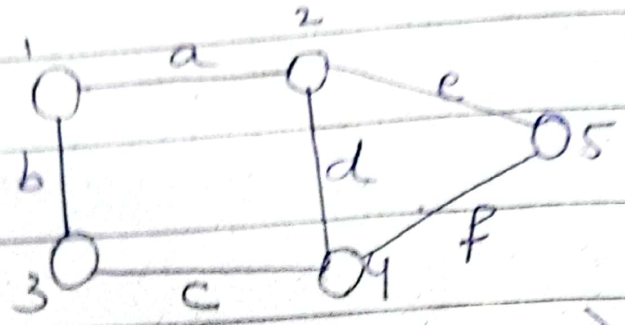
Scanned with CamScanner

Date: 29/11/2022

Data Structures

→ Lecture-25:

Graph/Graph Theory

→ 

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

∴ Vertex or node:

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

→ Order of graph = $|V|$

∴ Edge:

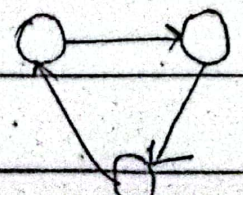
$E = \{a, b, c, d, e, f\}$

⇒ Size of Graph = $|E|$

→ Types:

⇒ Directed Graph (has specific direction(s))

e.g

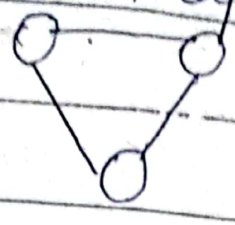


Scanned with CamScanner

29/11/2022

Un-directed Graph:

e.g



Adjacent Vertex:

(Vertex with common edge)

e.g {(1,2), (1,3), ...}

(1,4) X

Adjacent Edge:

(Edges with the common vertex)

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

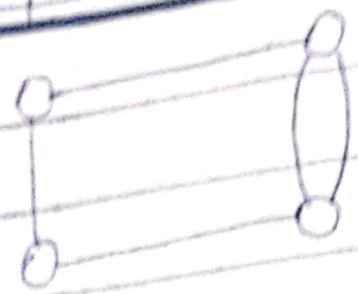
Graph with self loop:

Pseudo Graph:

Having self loop and multi-path.

Scanned with CamScanner

Date: 29/11/2022



→ Multi-Graph:
Having multi-path but no self loop

→ Simple Graph:
No self loop, No multi-path

→ Traversal of Graph:
(visit and Explore Scheme)

i) BFS

ii) DFS

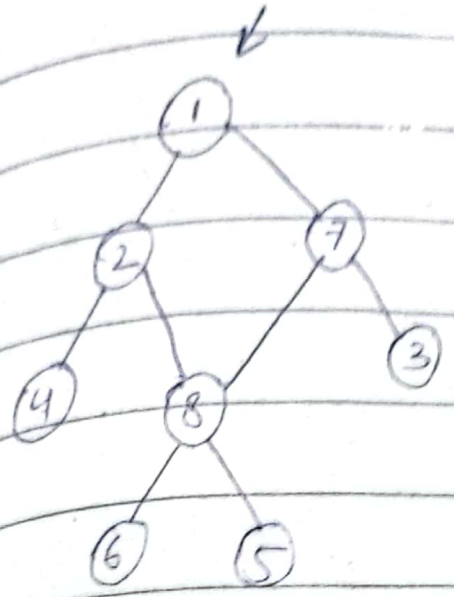
→ BFS (Breadth-First-Search)
(No back tracking) on Level wise traversal

→ DFS (Depth-First-Search)
(supports back tracking)

Scanned with CamScanner

29/11/2022

e.g



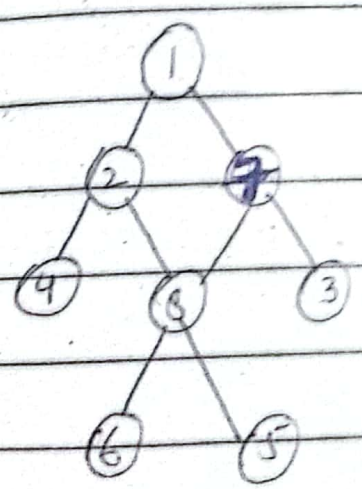
BFS:

Data Structure:
"Queue"

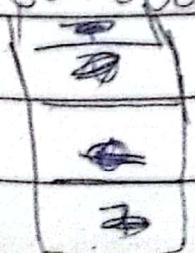
~~1~~	~~2~~	~~7~~	~~4~~	~~8~~	~~3~~	~~6~~	~~5~~

12748365

DFS:



Data structure : "Stack"



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

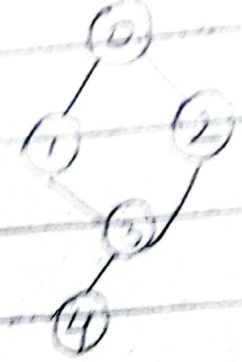
Scanned with CamScanner

Date: 1/12/2022

Data Structures

→ Lecture - 26:

Representation of Graph using Adjacency List:



→ Adjacency List:

We maintain an array of vertices. Every vertex will have a linked list of adjacent edges.

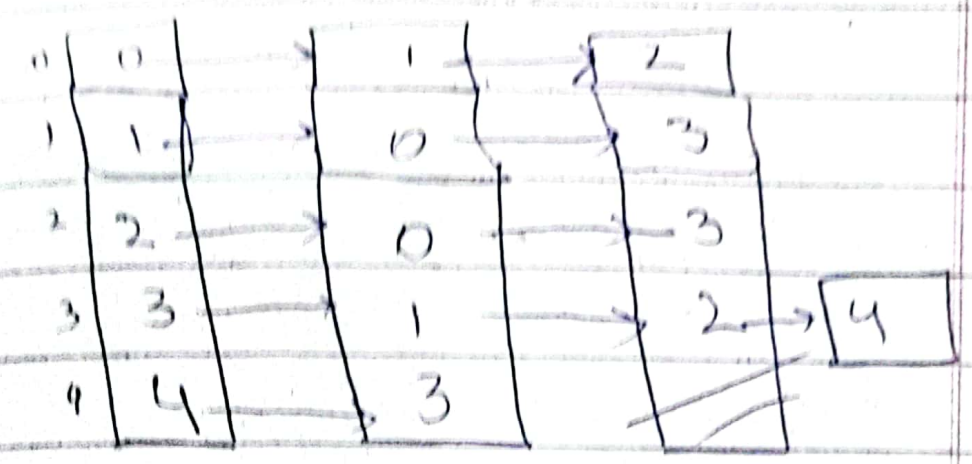
→ Size of Array in Adjacency list ???

Ans.

Scanned with CamScanner

Date:

Representation:



Undirected : $O(V+2E)$

Directed : $O(V+E)$

Scanned with CamScanner