CSA03 – DATA STRUTURES

Concept Mapping

Topic 1: Introduction to Data Structures 1
- Data
- Data Object
- Data Structures
- Abstract Data Types (ADT)
- Realization of ADT in C
- Primitive and Non-Primitive DS
- Linear and Non-Linear DS
- Static and Dynamic DS
- Persistent Data Structures
- Linked Data Structures
- Succinct Data Structures
- Implicit Data Structures
- Compressed Data Structures
- Search Data Structures
- Recursion

Topic 2: Array 2
- ADT Array
- Operations on Arrays
- Searching Algorithms
  - Linear Search
  - Binary Search

Topic 3: Linked List 3
- Singly Linked List
- Doubly Linked List
- Circularly Linked List

Topic 4: Stack 4
- Array Implementation
- Linked List Implementation

Topic 5: Stack Application 5
- Arithmetic Expression – Notations
  - Infix Notation
  - Prefix Notation
  - Postfix Notation
- Expression Conversion
- Expression Evaluation
- Balancing Symbols

Topic 6: Queue 6
- Array Implementation
- Linked List Implementation
- Circular Queue
- Applications

Topic 7: Trees 7
- Preliminaries
- Binary Tree
- Binary Search Tree
- Tree Traversal

Topic 8: AVL Trees 8
- Single Rotations
  - Left-Left (LL)
  - Right-Right (RR)
- Double Rotations
  - Left-Right (LR)
  - Right-Left (RL)

Topic 9: Applications of Search Trees 9
- B-Tree
- TRIE
- 2-3 Tree
- 2-3-4 Tree

Topic 10: Red-Black Tree 10

Topic 11: Splay Trees 11
- Zig Rotations
- Zag Rotations
- Zig-Zig Rotations
- Zag-Zag Rotations
- Zig-Zag Rotations
- Zag-Zig Rotations

Topic 12: Hashing 12
- Hash Function
- Separate Chaining
- Open Addressing
- Linear Probing

Topic 13: Priority Queue 13
- Heap
- Heap Sort

Topic 14: Sorting 14
- Insertion Sort
- Merge Sort
- Radix Sort

Topic 15: Quick Sort 15
- Divide & Conquer
- Time Complexity

Topic 16: Sorting 16
- Bubble Sort
- Selection Sort
- Time Complexity

Topic 17: Graph 17
- Terminologies
- Types
- Topological Sort

Topic 18: Shortest Path Algorithms 18
- Unweighted Shortest Path
- Dijkstra's Algorithm

Topic 19: Minimum Spanning Tree (MST) 19
- Prim's Algorithm
- Kruskal's Algorithm

Topic 20: Graph Traversal 20
- Breadth First Search
- Depth First Search

Topic 21: Advanced Data Structures 21
- Future Scope
- Real Time Applications
- Research Area

CSA03 – DATA STRUTURES

Concept Mapping

Topic 1: Introduction to Data Structures 1
- Data
- Data Object
- Data Structures
- Abstract Data Types (ADT)
- Realization of ADT in C
- Primitive and Non-Primitive DS
- Linear and Non-Linear DS
- Static and Dynamic DS
- Persistent Data Structures
- Linked Data Structures
- Succinct Data Structures
- Implicit Data Structures
- Compressed Data Structures
- Search Data Structures
- Recursion

Topic 2: Array 2
- ADT Array
- Operations on Arrays
- Searching Algorithms
  - Linear Search
  - Binary Search

Topic 3: Linked List 3
- Singly Linked List
- Doubly Linked List
- Circularly Linked List

Topic 4: Stack 4
- Array Implementation
- Linked List Implementation

Topic 5: Stack Application 5
- Arithmetic Expression – Notations
  - Infix Notation
  - Prefix Notation
  - Postfix Notation
- Expression Conversion
- Expression Evaluation
- Balancing Symbols

Topic 6: Queue 6
- Array Implementation
- Linked List Implementation
- Circular Queue
- Applications

Topic 7: Trees 7
- Preliminaries
- Binary Tree
- Binary Search Tree
- Tree Traversal

Topic 8: AVL Trees 8
- Single Rotations
  - Left-Left (LL)
  - Right-Right (RR)
- Double Rotations
  - Left-Right (LR)
  - Right-Left (RL)

Topic 9: Applications of Search Trees 9
- B-Tree
- TRIE
- 2-3 Tree
- 2-3-4 Tree

Topic 10: Red- Black Tree 10

Topic 11: Splay Trees 11
- Zig Rotations
- Zag Rotations
- Zig-Zig Rotations
- Zag-Zag Rotations
- Zig-Zag Rotations
- Zag-Zig Rotations

Topic 12: Hashing 12
- Hash Function
- Separate Chaining
- Open Addressing
- Linear Probing

Topic 13: Priority Queue 13
- Heap
- Heap Sort

Topic 14: Sorting 14
- Insertion Sort
- Merge Sort
- Radix Sort

Topic 15: Quick Sort 15
- Divide & Conquer
- Time Complexity

Topic 16: Sorting 16
- Bubble Sort
- Selection Sort
- Time Complexity

Topic 17: Graph 17
- Terminologies
- Types
- Topological Sort

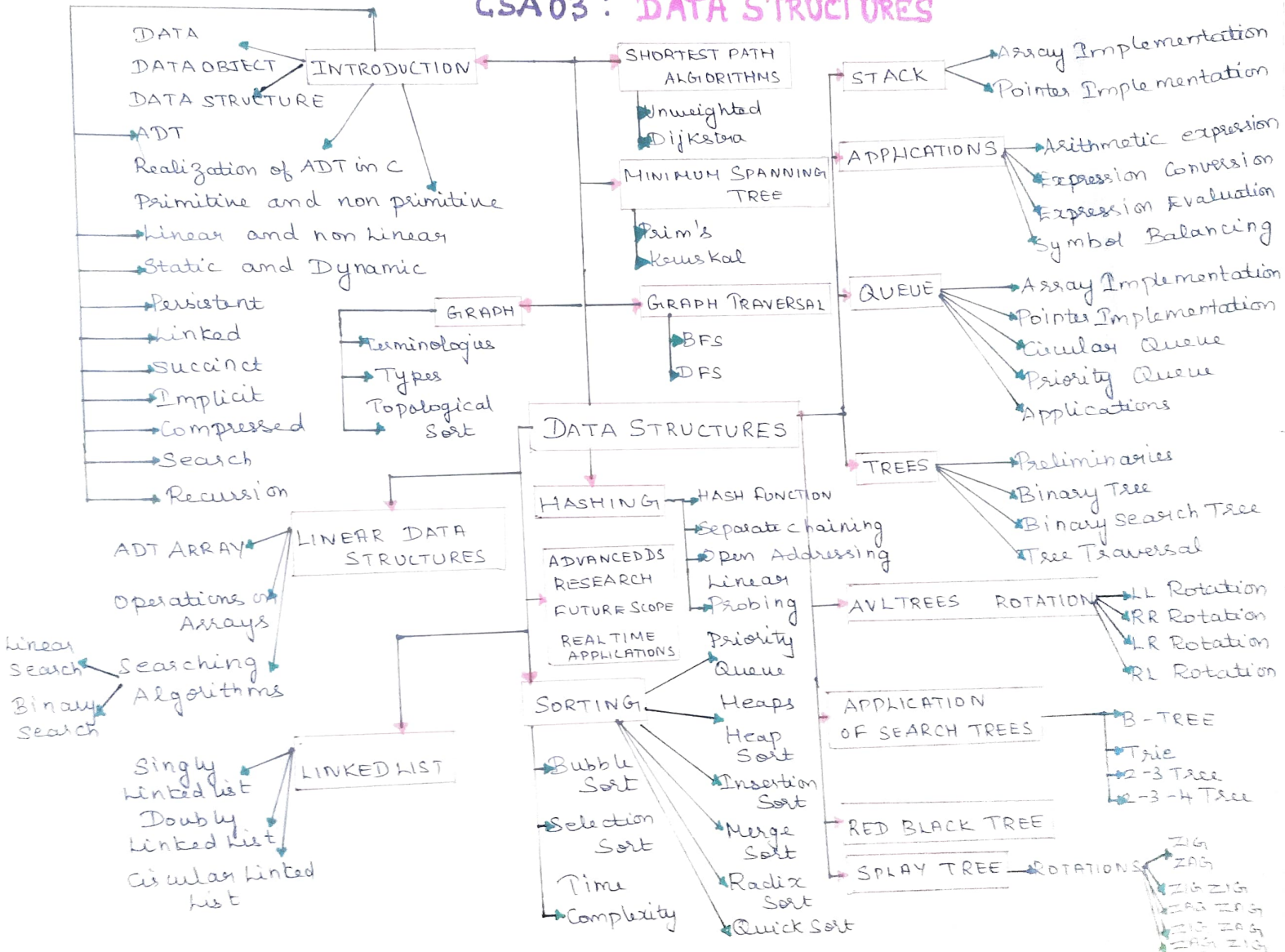
Topic 18: Shortest Path Algorithms 18
- Unweighted Shortest Path
- Dijkstra's Algorithm

Topic 19: Minimum Spanning Tree (MST) 19
- Prim's Algorithm
- Kruskal's Algorithm

Topic 20: Graph Traversal 20
- Breadth First Search
- Depth First Search

Topic 21: Advanced Data Structures 21
- Future Scope
- Real Time Applications
- Research Area

CSA03 : DATA STRUCTURES



Topic 1: Introduction to Data Structures

- Data
- Data Object
- Data Structures
- Abstract Data Types (ADT)
- Realization of ADT in C
- Primitive and Non-Primitive DS
- Linear and Non-Linear DS
- Static and Dynamic DS
- Persistent Data Structures
- Linked Data Structures
- Succinct Data Structures
- Implicit Data Structures
- Compressed Data Structures
- Search Data Structures
- Recursion

TOPIC 1 - INTRODUCTION - DATA STRUCTURES

DATA - A Piece of Information

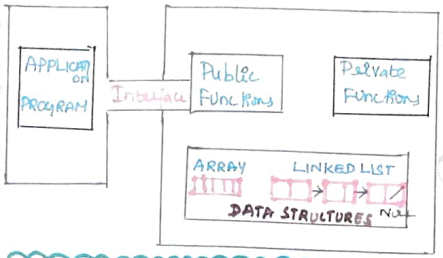
DATA OBJECT - Region of storage
- Values / group of values

DATA STRUCTURES - Storage used to store and organize data.
⇒ Way of organising data in to Memory.
⇒ Can be accessed & Updated efficiently.

ADT - ABSTRACT DATA TYPE
- type for objects with well-defined interface
- Set of Values & operations
- to hide how the Operation is performed on the data.

- homogeneous - Contains elements of same type eg: Array
- heterogeneous - Contains elements of different types eg: Java Stack

REALIZATION OF ADT IN C
- provides only the Interface to which Data Structure must adhere.
- Usually defined as a Pointer to a Structure.
- A header file contains the ADT declaration



PRINITIVE AND NON-PRINITIVE
- Primitive: ⇒ Stores the data of only one type. eg: Integer, Character, float
  ⇒ Cannot be NULL
- Non-Primitive: ⇒ User-defined that stores the data of different types.
  ⇒ Contain NULL Values.

LINEAR AND NON-LINEAR
- Linear - Data elements are arranged sequentially or linearly eg: Stack, Array, Queue, Linked List
  - easy to implement
- Non-Linear - Data elements are not arranged sequentially or linearly. eg: Trees, Graphs
  - Utilizes Computer memory efficiently

Applications of Linear ⇒ Application Software Development

Applications of Non-Linear ⇒ Artificial Intelligence & Image Processing

STATIC AND DYNAMIC
- Static: ⇒ Size of the structure is fixed. ⇒ Content cannot be modified without changing memory space.
- Dynamic: ⇒ Size of the structure is not fixed. ⇒ Content can be modified during the operations performed on it.

PERSISTENT DATA STRUCTURE
- Partially Persistent: all versions can be accessed but only newest version be modified.
- Fully Persistent: every version can be accessed or modified.
- Confluently Persistent: allow merge operations to combine two versions
- Ephemeral: neither persistent nor partially persistent.

LINKED DATA STRUCTURE
⇒ Data structure consists of a set of data records (nodes) linked together
⇒ organized by sequences (links or pointers)
- Using Dynamic Allocation
- Using Array Index Linking

SUCCINCT DATA STRUCTURE
⇒ Succinct representation of data + succinct index

IMPLICIT DATA STRUCTURE
⇒ Stores little information
⇒ requires low overhead

COMPRESSED DATA STRUCTURE
⇒ Intersection of Data Structure and Data Compression
⇒ Used for Suffix Array

SEARCH DATA STRUCTURE
⇒ efficient retrieval of specific items from a set of items.
⇒ Specific record from a database

RECURSION - Technique of making a function call itself.
- used with functions
- Small Code Size
- terminates when the base becomes true
- every recursive call needs extra space

Example: Factorial

$$Fact(n) = \begin{cases} 1, & \text{when } n=0 \\ n*fact(n-1), & \text{when } n>0 \end{cases}$$

$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
$4! = 4(3!) \cdots 4(6) = 24$
$3! = 3(2!) \cdots 3(2) = 6$
$2! = 2(1!) \cdots 2(1) = 2$
$1! = 1(0!) = 1(1) = 1$

Topic 2: Array

- ADT Array
- Operations on Arrays
- Searching Algorithms
  - ✓ Linear Search
  - ✓ Binary Search

TOPIC 2 : LINEAR DATA STRUCTURES [ARRAY]

ADT Array

- Abstract Data type (ADT)
- Way of classifying data structures

Array

- Array is a collection of elements accessible by an index

Array Representation

- Index starts with 0
- Array length is 7. it can store 7 elements
- Each elements can be accessed its index.

Array Operations

- Traverse
- Insertion
- Deletion
- Search
- Update

Traverse

→ Visiting every element of an array

Insertion

→ Insert element at a specific position.

→ For this operation to succeed, the array must have enough capacity.

→ Insert an element to at index, then elements after index must get shifted to their adjacent right to make way for a new element.

Elements need to be shifted to maintain relative order.

→ No position specified then insert element at end and avoid shifting.

Deletion

→ An element at a specified position can be deleted

→ creating void that needs to be fix-ed by shifting all elements to their adjacent left.

Delete ele at index 2

Shift the ele after index 2 to their left.

Deletion Done

Search

→ Traversing array until the element found.

A[2] = 5

Update

→ updating an existing element at given index

Searching Algorithms

Searching

→ To check whether a particular elements is present in the list

Types

Linear Search

→ Sequential search and can be applied on unsorted list

Algorithm Steps

Example

To search an element '6' in the array using LS.

i=0, check 8 == 6 NO

i=1, check 2 == 6 NO

i=2, check 6 == 6 YES

Code

```
                    //Not found
```

Advantage

→ Simple, easy to understand and implement

→ Not required any order to store the data.

Disadvantage:

→ Slow & Poor efficiency.

Binary Search

→ Searching is done for a sorted Array

→ Uses Divide & conquer method

Algorithm Steps

low = 0, high = n-1, mid = (low+high)/2

a) If search element is greater than the mid value, find a new middle element from the right side of mid value.

b) If search element is less than the mid value, find a new middle element from the left

Example

mid = (low+high)/2

mid = (0+4)/2 = 2 : A[2]

check 7 == 5 (A[2])

↳ Not same (7 > 5)

mid = (low+high)/2 = (3+4)/2 = 3

check 7 == A[3]

return A[3] = 7

Advantage

→ Faster compared to LS

→ No of comparison less

→ More efficient.

Disadvantage

→ Works only on sorted list and complex.

Time Complexity

	Best case	Average	Worst
Linear			
Binary			

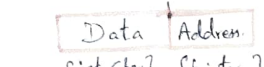
Topic 3: Linked List

- Singly Linked List
- Doubly Linked List
- Circularly Linked List

Topic: 3. Linked List – Single, Doubly & Circular

Single Linked List (SLL)

- Collection of nodes connected from head node to tail.
- Each node contains two fields.
  1) Data 2) Address.
- Connection in one direction.



- Linear data structure, in which nodes are created dynamically in memory & linked to the list.

Operations in SLL

- Insertion, -> Deletion, -> Searching
- Sorting -> Updation & Modification.

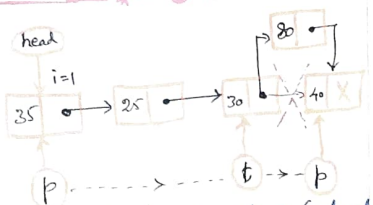
Insert at Begin:

- Assign p = head
- newnode -> is equal to 'p' pointer.
- head is equal to newnode.
- newnode -> next = p.

Insert at End:

- Move the pointer from head to tail.
- p -> next = newnode.

Insert at any position:

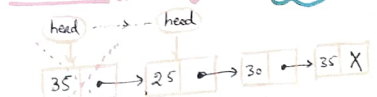


Assume: position = 4 ; ele = 80 (node value)

- Move the pointer 'p' from head to position node.
- Also move 't' from head to its previous position.
- Now t -> next = newnode.
- newnode -> next = p.

APPLICATION: FACEBOOK

Delete at Begin:



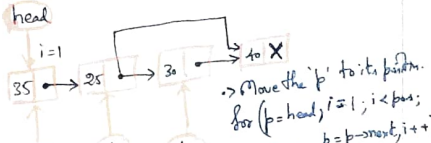
- Assign p = head
- head = p -> next
- Free (p) : Delete 'p' node.

Delete at End:

- Move the pointer 'p' from head to last node.
- Similarly move 't' to its previous position.
- Now t -> next = NULL.
- Free (p) : Delete 'p' node from the list.

Delete at any position:

Assume: position = 3

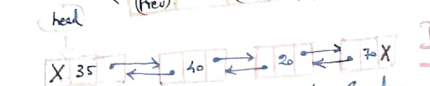


- Move the 'p' to its position. for (p = head; i = 1; i < pos; t = p; p = p -> next; i++)

Now t -> next = p -> next.

Doubly Linked List: (DLL)

- Collection of nodes connected in two directions in which node contains two addresses.



- Operations -> Insertion, Deletion, Searching Sorting, Modifications.

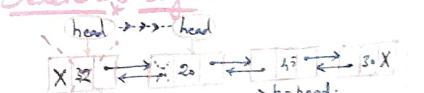
Insert at Begin:

- p = head
- newnode -> next = p
- head = newnode.
- p -> prev = newnode.

Insert at End:

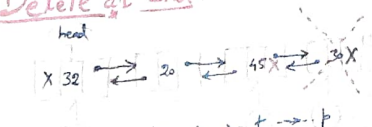
- Move 'p' to last node.
- p -> next = newnode
- newnode -> prev = p.

Delete at Begin:



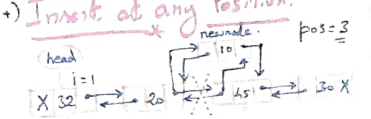
- p = head;
- head = p -> next;
- Free (p)
- head -> prev = NULL;

Delete at End:



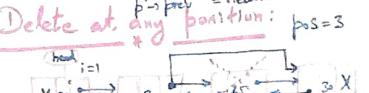
- Move 'p' pointer from head to end node.
- With respect to make 't' to its previous position.
- t -> next = NULL.
- Free (p) ; Delete the 'p' node.

Insert at any Position:



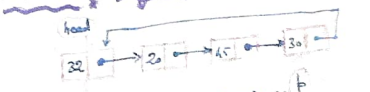
- Move 'p' pointer from head to position & 't' to its previous position.
- Now t -> next = newnode;
  newnode -> next = p;
  newnode -> prev = t;
  p -> prev = newnode.

Delete at any position:



- Move the pointer from head to position 3 and 't' to its previous.
- Now t -> next = p -> next;
- p -> next -> prev = t;

Circular Single Linked List:



- Move the 'p' from first to last
- p -> next = head.

Circular Double Linked List

- Move 'p' to last node.
- p -> next = head
- head -> prev = p.

Topic 4: Stack

- Array Implementation
- Linked List Implementation

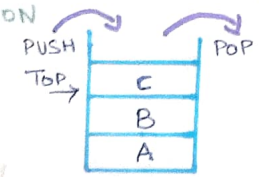
TOPIC -4 STACK IMPLEMENTATION

Stack: An Abstract Data Type (ADT)

* A linear DS in which the operations are performed at only one end - Top
* Eg: Pile of coins, stack of Trays
* **PRINCIPLE**: Last in First out LIFO
  Lastly Inserted element will be removed first

IMPLEMENTATION: 2 Types
- ARRAYS (Fixed size DS)
- LINKED LIST - (Variable size DS)

BASIC OPERATION
* PUSH
* POP
* PEEK
* IS FULL
* IS EMPTY



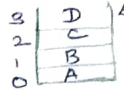
ARRAY IMPLEMENTATION

PUSH Process of inserting a new data element when stack is empty **Top = -1**

* check if the stack is Full (overflow)
* If the stack is Full, display stack Full and exit
  [Top = Maxsize - 1]
* If stack is not Full increment Top Top++
* Add data element to the stack stack[Top] = Value

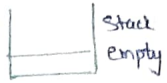
ISFULL - check if stack is Full

If (top == Maxsize - 1)
return true;



IS Empty - check if stack is empty

If (top == -1)
return true;



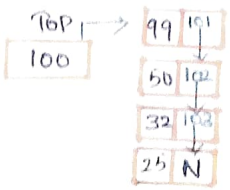
POP: Process of removing top element from the stack

Steps
* check if stack is empty (underflow)
* If empty display stack empty (Top = -1)
* If not empty decrements
* Top value by 1 (Top = Top - 1)
  Return

PEEK: Returns the value of top most element

Steps
Checks if the stack is empty (underflow)
If empty displays "stack empty" (Top = -1)
If not empty, returns the element which Top is pointing (Stack[Top])

LINKED LIST IMPLEMENTATION



PUSH: Process of inserting new element into the stack
* Create a newNode with given value
* check whether stack is Empty (Top == NULL)
* If it is Empty then set newNode -> next = NULL
* If it is Not Empty then set newNode -> next = Top
* Finally, set Top = newNode

POP: Process of Deleting an Element from a Stack
* Check whether stack is empty (Top == NULL)
* If it is Empty then "stack is empty" Deletion is not possible
* If it is Not Empty then define a Node pointer 'temp' and set it to Top
* Then set Top = Top -> next
  Finally delete 'temp' (Free(temp))

PEEK: Returns the value of Top most element
* check whether stack is empty (Top == NULL)
* If it is empty then "Stack is empty cannot Display"
* If it is Not Empty then check (temp -> next != NULL) and display the remaining

Topic 5: Stack Application

- Arithmetic Expression – Notations
  - ✓ Infix Notation
  - ✓ Prefix Notation
  - ✓ Postfix Notation
- Expression Conversion
- Expression Evaluation
- Balancing Symbols

TOPIC-5 (STACK APPLICATIONS)

Arithmetic Expression

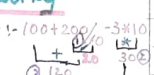
Types of Notations:

* **Infix Notation** - arithmetic operator appears between operands (op1 operator op2) (eg:- a+b)
* **Prefix Notation** - arithmetic operator appears before operands (Polish Notation) (eg:- +ab)
* **Postfix Notation** - arithmetic operator appears after operands (Reverse Polish Notation) (eg:- ab+)

Precedence of Operators & Associativity

Operator	Precedence	Value
Exp(↑,^)	Highest	3
*, /	Next high	2
+, -	lowest	1

Eg:- 100+200/10-3*10



Algorithms for Infix-Postfix Conversion

1) Scan infix expression from left to right, if it is operand, output it into postfix expression.
2) If the input is an operator and stack is empty, push operator into operators stack.
3) If the operators stack is not empty
   * if the precedence of operator is greater than the top most op of stack, push it into stack.
   * if the precedence of operator is less than the top most operator of stack, pop it from the stack until we find a low precedence op.
   * if the precedence of operator is equal, then check the associativity of the operator. If LTR (Left to Right) associativity, then pop the operator from stack. If RTL (Right to Left) associativity, then push into the stack.
   * if the input is '(', push it into stack.
   * if the input is ')', pop out all operators until we find '('
4) Repeat step 1,2 & 3 till expression has reach '#' [# - Delimiter].
4) Now pop all the remaining operators from the stack and push into the output postfix expression.
5) Exit.

Example: A+(B*(+D)/E#

i/p	stack	output Postfix Exp	Action
A		A	Add A into o/p exp
+	+	A	Push '+' into stack.
(	+(	A	Push '(' into stack
B	+(	AB	Add B into o/p exp.
*	+(*	AB	Push '*' into stack.
(	+(*	ABC	Add C into o/p exp
+	+(* / +(+	AB*C	'+' operator has less Precedence than '*' & add it to o/p exp. Push '+'
D	+(+	ABC*D	Add D into o/p exp.
)	+(+ / +	ABC*D+	')' has come, So pop + & add it to o/p exp
/	+/	ABC*D+	'/' has higher Precedence than '+', so push '/' to stack
E	+/	ABC*D+E	Add E into o/p exp.
#	+/ / +	ABC*D+E/ / ABC*D+E/+	'#' - delimiter, So pop all operators from the stack one by one and add it to o/p expression

Output: ABC*D+E/+

Evaluating Arithmetic Expression

Algorithm Steps:

1) Scan the input string from left-right
2) for each input symbol
   a) if it is a digital number, then Push it onto the stack.
   b) if it is an operator, then pop out the top most 2 contents from the stack & apply the operator on them. Later, push the result to stack.

Example: (5+3)*(8-2) ⇒ Infix

1) Convert into postfix expression
   53+82-*
2) Evaluate the postfix expression.

READ Symbol	STACK OPERATIONS	EVALUATED Part of Expression
Initial	Stack-Empty	Nothing
5	Push(5)	Nothing
3	Push(3)	Nothing
+	Value1=PoP()=3 Value2=PoP()=5 result=value2+value1 Push(result)	Value1=3 Value2=5 result=5+3=8 Push(8)
8	Push(8)	(5+3)
2	Push(2)	(5+3)
-	Value1=PoP()=2 Value2=PoP()=8 result=value2-value1	Value1=2 Value2=8 result=8-2 Push(6)
*	Value1=PoP()=6 Value2=PoP()=48 result=value2*value1 Push(result)	(5+3),(8-2) Value1=6 Value2=8 Result=8*6 Push(48)
$ end	result=PoP()	(5+3)*(8-2) Display(result) Result=48 [5+3)*(8-2)]

Note:- if input is '0' (end), empty the stack.

Balancing Symbols

Algorithm Steps:

Read 1 character at a time until it encounters the delimiter '#'
1) If the character is an opening symbol, push it onto the stack.
2) If the character is a closing symbol, and if the stack is empty report an error as missing symbol.
3) If it is a closing symbol & if it has a corresponding opening symbol in the stack, Pop it from the stack. Otherwise, 'error - Mismatched Symbol'.
4) At the end of file, if the stack is not empty, report as "Unbalanced Symbol" Otherwise, report as "Balanced Symbol"

Example: (a+b)#

1 2 3 4 5 6

Step	Read character	STACK	Step	Read character	STACK
1	(	(	2	a	(
3	+	(	4	b	(
5	)		6	#	empty

(a+b)# ➡ Balanced Symbol

Example: ((a+b)#

1 2 3 4 5 6 7

Step	Read character	STACK	Step	Read character	STACK
1	(	(	2	(	(
3	a	(	4	+	(
5	b	(	6	)	(
7	#	(← Stack Not empty			

((a+b)# ➡ Unbalanced Symbol

Applications

* Frequently accessed Websites - Cookies
* Visited Machines - JVM
* Redo/Undo Operations - Google Docs
* Forward-Backward Surfing - Google
* Scratch Card - Google Pay

Topic 6: Queue

- Array Implementation
- Linked List Implementation
- Circular Queue
- Applications

TOPIC - 6 (QUEUE)

→ Queue ADT: Linear Data Structure
* Insertion : Rear End - ENQUEUE
* Deletion : Front - DEQUEUE
* First In First Out (FIFO)

→ Real Time Example:
Bill Counter, Bank Counter, etc.

→ Applications:
Printer, CPU task scheduling, etc.

→ Implementations:
* Array * Linked List.

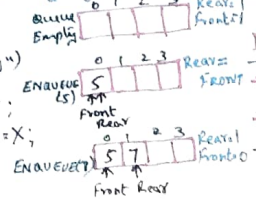
→ Basic Operations:
* Enqueue
* Dequeue
* Display.

→ Implementation - Using Array:
Fixed Size DS.

Enqueue:
* Before inserting, check whether Queue is full (Overflow Condition)
* If full, display Queue Overflow. Rear=N
* If not full, element is added to the end of the Array at rear side.

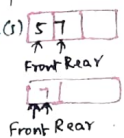
Note:- N - maximum size of Queue.

```
if (rear ≥ maxSize)
    Print ("Overflow")
{   Rear = Rear + 1;
    Queue[Rear] = X;
}
```



Dequeue:
Note:- Before deleting, check whether Queue is empty (Underflow Condition)
* If empty, display "Queue Underflow" Front=-1 (empty)
* If not empty, element in the front is deleted from the Queue and front is pointed to the next element in the Queue.

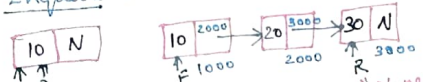
```
if (Front == -1)
    Print ("Underflow")
  else { Front = Front + 1;
}
```



→ Implementation - Using Linked List:
Each node contains 2 fields - Data field (Data) & Pointer field (Address of next node)

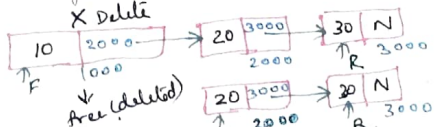
Node: [Data | Address of Next]

Enqueue: F → Front ; R → Rear



Note:- Before inserting [Enqueue] check whether Queue is full (Since dynamic DS). "No need to"

Dequeue:
Before deleting, check whether Queue is empty (Underflow Condition)

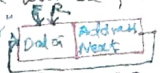


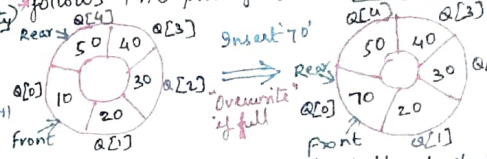
Advantages over Arrays:
Dynamic Size, Easy to insert and delete.

Drawbacks
* Random access not allowed.
* Utilization of memory space is more.

→ Types:
* Circular Queue * Priority Queue * DEQUE

Circular Queue: Last node will be pointing to the first node. "Ring Buffer"
* follows FIFO principle





Disadvantage:
* When the Circular Queue is full, it starts overwriting the existing values when insert.

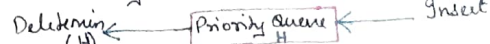
→ Basic Operations:- Enqueue & Dequeue.

Priority Queue: Elements will be assigned by a 'priority' & will be processed based on order.
* higher order priority element is processed before lower priority element.

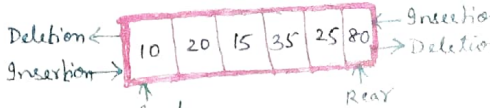
→ Types:
* Ascending Priority Queue
* Descending Priority Queue.

→ Basic Operations: Insert (Enqueue) & Delete (Deletemin)

→ Implementations:-
* Linked List
* Binary Search Tree (BST)
* Binary Heap.



Double-Ended Queue (DEQUE)
Both insertion and deletion operations are performed at both the ends.



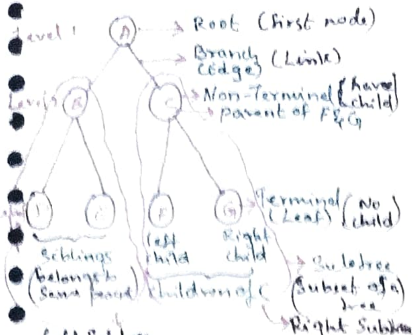
Queue Applications:
* Operating Systems - Job Scheduling
* CPU Scheduling
* Printer Spooler

Topic 7: Trees

- Preliminaries
- Binary Tree
- Binary Search Tree
  - ✓ Tree Traversal

TOPIC - I (PRELIMINARIES - BINARY TREE - TREE TRAVERSAL - BINARY SEARCH TREE)

TREES → Non Linear DS
* represents hierarchical relation



Note: N nodes → N-1 Edges

Height - No. of edges from leaf ... node

Depth - No. of edges from root ... node

Length - No. of nodes in that path

Degree - No. of children of a node

Climbing - Traversing from leaf to root

Descending - Traversing from root to leaf

Simple Tree

Binary Tree

Null Tree - tree with only root node

Binary Tree Application

Binary Tree types

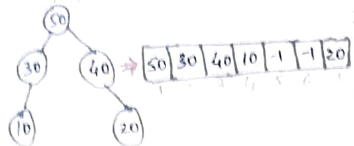
* Skewed Binary Tree

* Strictly Binary Tree

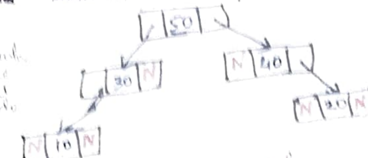
* Complete Binary Tree

Binary Tree - Representation

* Array Representation
  - Root → position 1
  - Parent (i) → i/2 (i → Node Position)
  - Left child → 2i
  - Right child → 2i+1
  - Empty Position → -1 (No Node)



* Linked List Representation



Binary Tree Traversal

- Inorder
- Preorder
- Postorder

Inorder

Left → Root → Right

DGBAHEICF

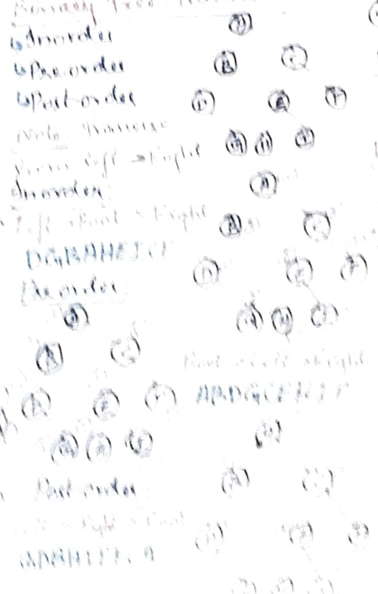
Preorder

Root → Left → Right

ABDGCEHIF

Postorder

Left → Right → Root



Binary Tree - Expression Tree

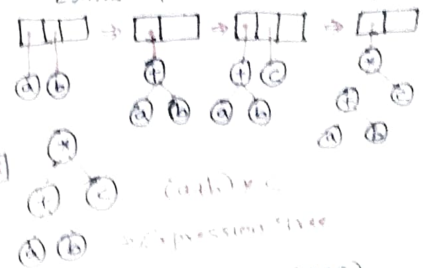
→ Leaf Node Operands
→ Non Leaf Nodes Operators
→ Traversal by inorder, preorder & postorder
→ Find correct infix, prefix, postfix expression

Steps:

* Input symbol operand → create a new node
* push a pointer onto the stack
* input symbol operator → ...

Eg: (a+b)*c → ab+c*

[Infix expression → Postfix expression]



Binary Search Tree (BST)

Left Subtree < Root < Right Subtree (elements)

BST Operations

Insert, Delete, Search

Insert - comparing with root / Parent

45, 15, 79, 90, 10, 55, 20

45 first element → root

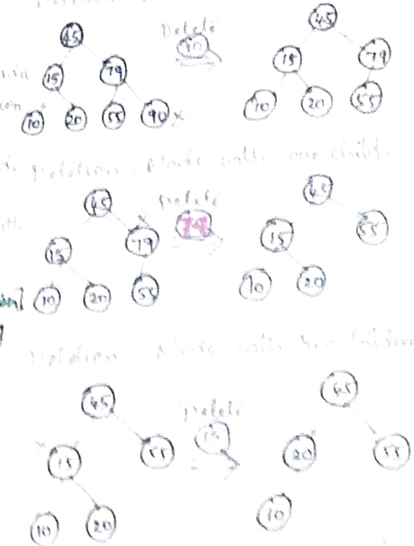
Delete - ... node deletion

Node ... children

... one child

... two children

Deletion - Node with No children



Deletion - Node with one child

Deletion - Node with two children

Search - comparing with root / Parent

Search (20)

Note

* Minimum element ... left most node

* Maximum element ... right most node

* Game Engine
* Compiler ...

Topic 8: AVL Trees

- Single Rotations
  - ✓ Left-Left (LL)
  - ✓ Right-Right (RR)
- Double Rotations
  - ✓ Left-Right (LR)
  - ✓ Right-Left (RL)

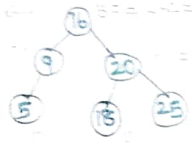
TOPIC 8 - AVL TREE

Balanced Search Tree

Balance Factor (BF) = Height of the Left Sub Tree − Height of the Right

BF = -1, 0, 1 => Balanced

Eg:



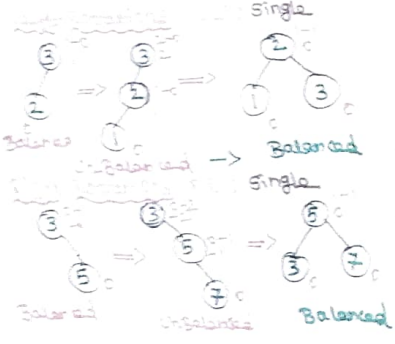
Tree is Balanced

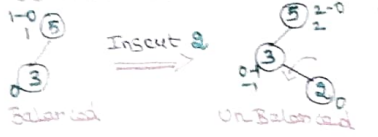
Height (Left Subtree) = 2

Height (Right Subtree) = 2

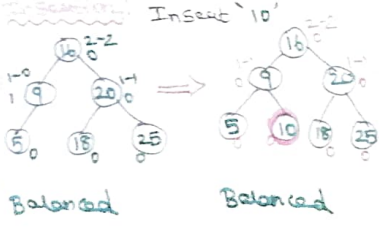
Balance Factor = 2 - 2 = 0

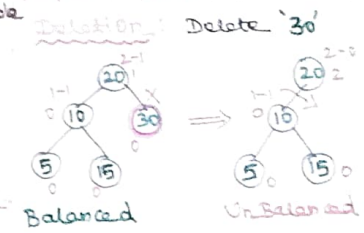
* If tree not balanced (BF = 1, -1, 0) then perform rotations to balance
* AVL causes unbalanced when insert

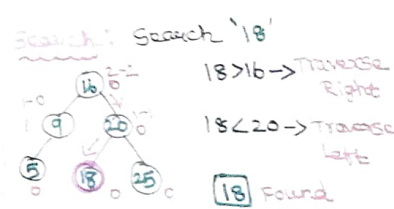




* Insertion
* Deletion
* Search

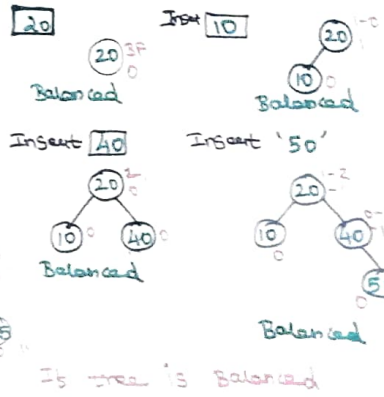






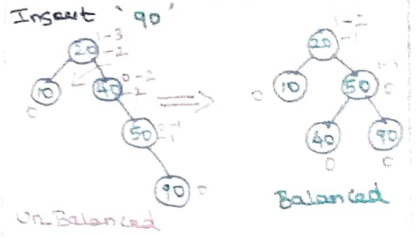
Example: Balance a tree using AVL

Insert 20, 10, 40, 50, 90, 30



Is tree is Balanced

No rotations required



APPLICATIONS: ONLINE DICTIONARY

* In memory sorts of Dictionaries
* Search Engine Optimization
* Equal Distance balancing from Server to client node in networks.

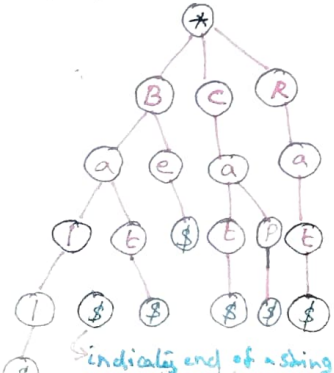
Topic 9: Applications of Search Trees

- B-Tree
- TRIE
- 2-3 Tree
- 2-3-4 Tree

TOPIC-9 (APPLICATIONS OF SEARCH TREES - B TREE - TRIE - 2-3 TREE - 2-3-4 TREE)

TRIE

→ All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings.
→ TRIE data structure makes retrieval of a string from the collection of strings more easily.
→ Term 'TRIE' comes from the word retrieval
→ It is a special kind of tree that is used to store the dictionary.
→ It is a fast & efficient way for dynamic spell checking.
→ Every node in Trie can have one or a number of children.
→ All the children of a node are alphabetically ordered. If any two strings have a common prefix then they will have the same ancestors.
Eg: Consider the following list of strings to construct Trie Cat, Bat, Ball, Rat, Cap & Be



B-TREE (2-3 Tree, 2-3-4 Tree)

→ Self Balancing Search Tree
→ Disk access time is very high compared to main memory access time.
→ The main idea of using B-Tree is to reduce the number of disk accesses.
→ Time complexity for Insert, Delete - O(log n)
→ All leaves are at the same level.
→ 'm' order can have m children & m-1 key values

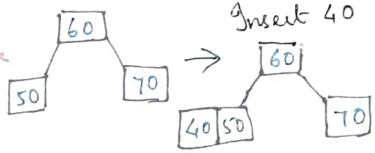
Types
2-3, 2-3-4, 2-3-4-5, ... and so on.

2-3 Tree.

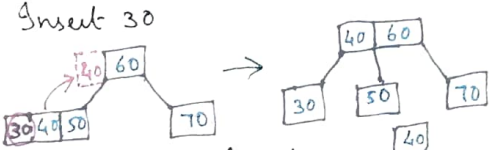
→ B tree of order 3
→ Each node has either 2 or 3 children & atmost 2 key values.
Ex: Insert 50, 60, 70, 40, 30, 20

Insert 50 → Insert 60 → Insert 70

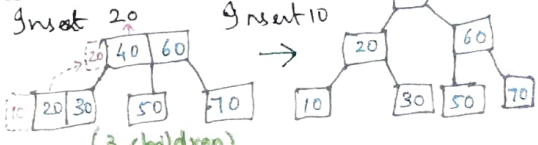
Insert 40



Insert 30



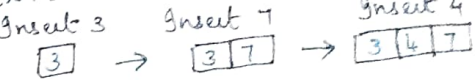
Insert 20 → Insert 10



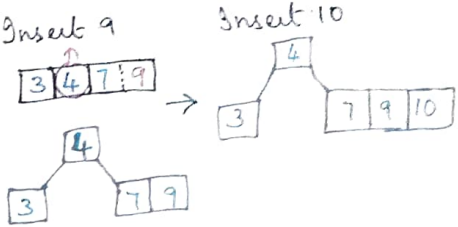
2-3-4 Tree

→ B tree of order 4
→ Each node has either 2/3/4 children & atmost 3 key values.
Ex: Insert 3, 7, 4, 9, 10, 0, 5, 6, 8, 2

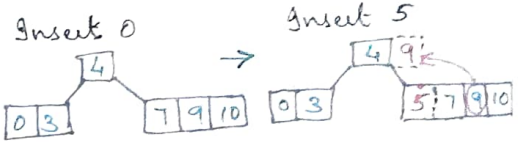
Insert 3 → Insert 7 → Insert 4



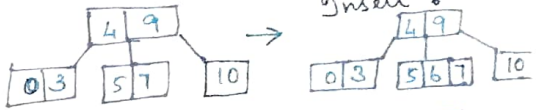
Insert 9 → Insert 10



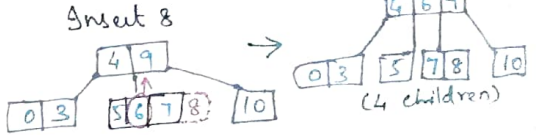
Insert 0 → Insert 5



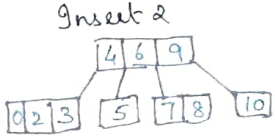
Insert 6



Insert 8



Insert 2



Topic 10: Red- Black Tree

- Properties
- Operations

TOPIC:10 RED BLACK TREE

RED-BLACK TREE:

BST in which every node is colored either **RED** or **BLACK**

Properties

→ BST → Newnode - **RED**
→ Root Node - **BLACK**
→ No 2 consecutive **RED** nodes
→ All paths same no of BLACK nodes
→ Leaf node - **BLACK**

Insertion Need to satisfy properties if not, perform operations to make it **RED-BLACK** Tree

1. Recolor
2. Rotations
3. Rotations Followed by Recolor

STEPS

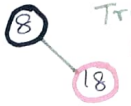
1. Check Tree is empty
2. Empty insert Newnode as ROOT (BLACK)
3. Not empty insert New node as a Leaf node (RED).
4. Parent (New Node) BLACK exit
5. Parent (NewNode) RED check Color of Parent node's sibling of New Node
   - Sibling **BLACK** or NULL Rotate and Recolor it
   - Sibling **RED** Recolor repeat the steps until tree becomes **RED-BLACK** Tree

Example: Create **RED-BLACK** Tree by inserting 8, 18, 5, 15, 17, 25, 40, 80

Insert 8

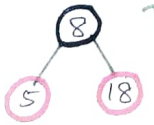
NewNode **BLACK**

Insert 18



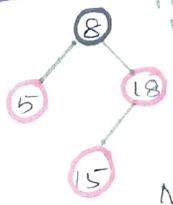
Tree Not empty
New Node **RED**

Insert 5



Tree Not empty
New Node **RED**

Insert 15

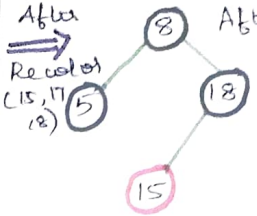


Tree not empty
New Node **RED**
18, 15 (**RED**) (Consecutive)
New Node Parent
Sibling **RED** (5)
Recolor

Note: Dont Recolor if Parent - root (**BLACK**)

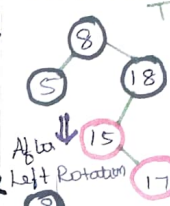
BLACK TREE

After ⇒ Recolor (15, 17, 18)



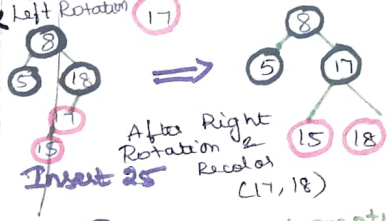
After Recolor (5, 18)

Insert 17



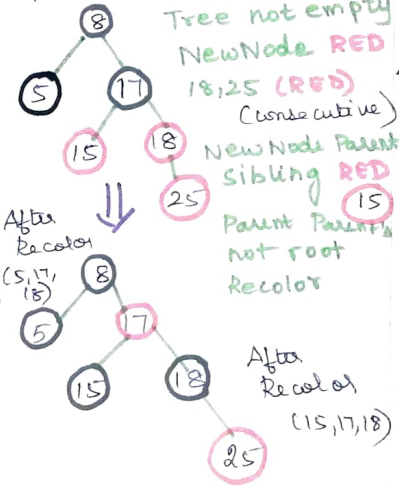
Tree not empty
New Node **Red**
15, 17 **RED** (Consecutive)
New Node Parent
Sibling NULL
Rotate & Recolor

After Left Rotation ⇒



After Right Rotation & Recolor (17, 18)

Insert 25

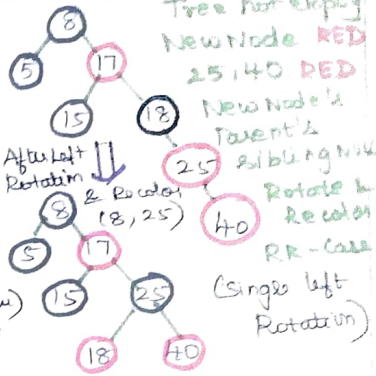


Tree not empty
New Node **RED**
18, 25 (**RED**) (Consecutive)
New Node Parent
Sibling **RED** (15)
Parent Parent's not root
Recolor

After Recolor (5, 17, 18)

After Recolor (15, 17, 18)

Insert 40



Tree not empty
New Node **RED**
25, 40 **RED**
New Node's Parent's sibling NULL
Rotate & Recolor
RR - Case (Single left Rotation)

After Left Rotation ⇒ & Recolor (18, 25)

Insert 80

Tree not empty
New Node **RED**
40, 80 **RED**
New Node Parent sibling **RED** (18)
Parent's Parent not root
Recolor (17, 25)

(Single left Rotation)
After Recolor ⇒

After left Rotation (8, 17)

Searching words in Dictionaries
Add or Delete items in cart - Online Shoppi

Topic 11: Splay Trees

- Zig Rotations
- Zag Rotations
- Zig-Zig Rotations
- Zag-Zag Rotations
- Zig-Zag Rotations
- Zag-Zig Rotations

TOPIC: 11

Splay Tree

Splay tree is a Self adjusted Binary Search tree in which every operation on an element rearrange the tree so that element is placed at the root position of the tree.

Splaying an element in the process of bringing it to the root position by performing suitable rotation operation.

Rotation in Splay Tree.

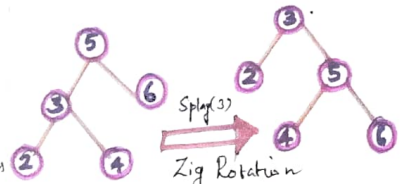
* Zig Rotation
* Zag Rotation
* Zig-Zig Rotation
* Zag-Zag Rotation
* Zig-Zag Rotation
* Zag-Zig Rotation

Did you know?

SplayNet [Facebook]

Zig Rotation:

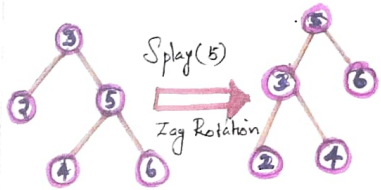
The Zig Rotation in a Splay tree is Single right rotation in AVL Tree rotation. In Zig rotation every node moves one position to the right from its Current position.



Single right Rotation

Zag Rotation:

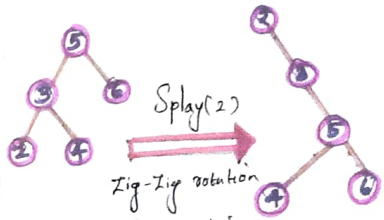
The Zag Rotation in a Splay tree is Similar to the Single left Rotation in AVL tree rotations. In Zag rotation every one moves one position to the left from its Current position.



Single left Rotation

Zig-Zig Rotation:

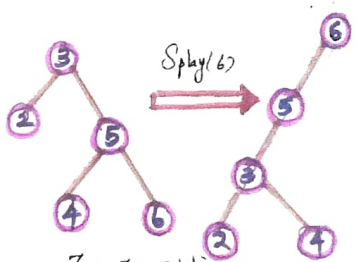
The Zig-Zig rotation in a Splay tree is a double Zig rotation. In Zig-Zig rotation every nodes Moves two position to the right from its Current position.



Double right rotation.

Zag-Zag Rotation:

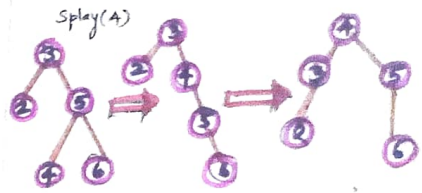
The Zag-Zag rotation in a Splay tree is a double Zag rotation. In Zag-Zag rotation every node moves two position to the left from its current position.



Zag-Zag rotation
Double left rotation

Zig-Zag Rotation:

The Zig-Zag rotation in a Splay tree is a sequence of Zig-Zag rotation every node moves one position to the right followed by one position to the left from its Current position.

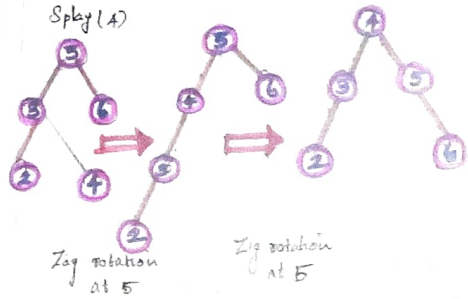


Zig Rotation at 5

Zag rotation at 3

Zag-Zig Rotation:

The Zag-Zig Rotation in a Splay tree is a Sequence of Zag rotation followed by Zig rotation. In Zag-Zig rotation every one moves one position to the left followed by one position to the right from its Current.



Zag rotation at 5

Zig rotation at 5

TOPIC: 11

Splay tree

Splay tree is a self adjusted Binary Search tree in which every operation on an element rearranges the tree so that element is placed at the root position of the tree.

Splaying an element is the process of bringing it to the root position by performing suitable rotation operation.

Rotation in Splay Tree.

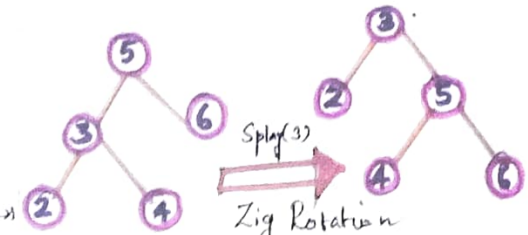
* Zig Rotation
* Zag Rotation
* Zig-Zig Rotation
* Zag-Zag Rotation
* Zig-Zag Rotation
* Zag-Zig Rotation

Did you know?

SplayNet [Facebook]

Zig Rotation:

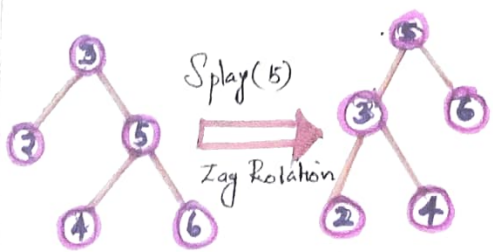
The Zig Rotation in a Splay tree is Single right rotation in AVL Tree rotation. In Zig rotation every node moves one position to the right from its Current position.



Zig Rotation
Single right Rotation

Zag Rotation:

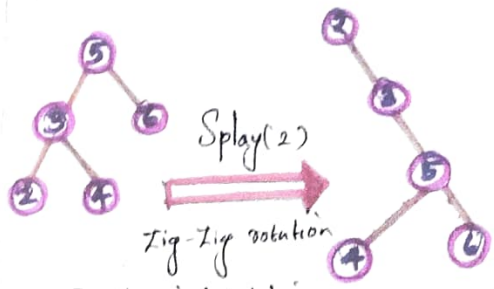
The Zag Rotation in a Splay tree is Similar to the Single Left Rotation in AVL tree rotations. In Zag rotation every one moves one position to the left from its Current position.



Zag Rotation
Single Left Rotation

Zig-Zig Rotation

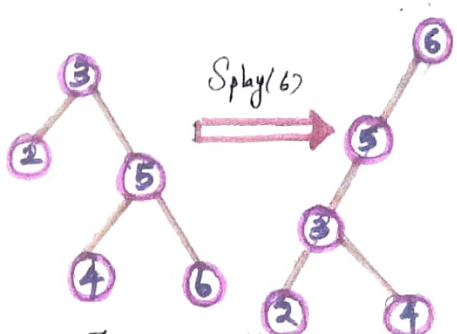
The Zig-Zig rotation in a Splay tree is a double Zig rotation. In Zig-Zig rotation every nodes Moves two position to the right from its Current position.



Zig-Zig rotation
Double right rotation.

Zag-Zag Rotation:

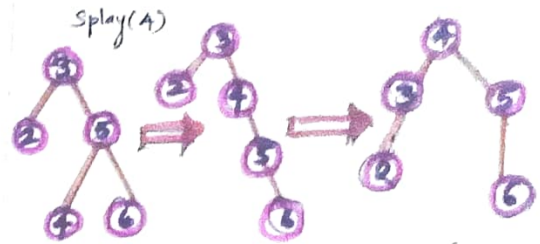
The Zag-Zag rotation in a Splay tree is a double Zag rotation. In Zag-Zag rotation every node moves two position to the left from its Current position.



Zag-Zag rotation
Double Left rotation

Zig-Zag Rotation:

The Zig-Zag Rotation in a Splay tree is a Sequence of Zig-Zag rotation every node moves one position to the right followed by one position to the left from its Current position.

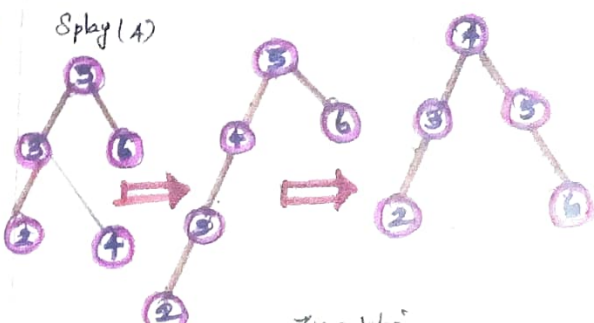


Zig Rotation at 5

Zag rotation at 5

Zag-Zig Rotation:

The Zag-Zig Rotation in a Splay tree is a Sequence of Zag rotation followed by Zig rotation In Zag-Zig rotation every one moves one position to the left - followed by one position to the right from its Current.



Zig rotation at 5

Zig rotation at 5

Topic 12: Hashing

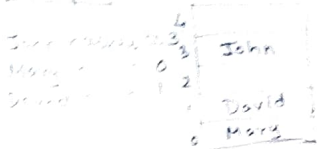
- Hash Function
- Separate Chaining
- Open Addressing
- Linear Probing

TOPIC - 12 (HASHING - HASH FUNCTION - SEPARATE CHAINING - OPEN ADDRESSING)

Hashing

Hash Table Data Structure - Arrays of Keys
Keys - Strings with associated value.
Hashing - Implementation of Hash Table.
Hash Function - Mapping each Key into Hash Table index with size
Hsize - Hash Table Size

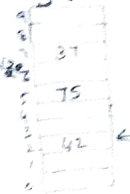
Example



Hash Functions - Types:

* Division Method:
  Index = Key Mod Hash Table Size
  Eg. - hash(92) = 92 Mod 10 (TableSize=10)
* Mid-Square Method: Key is squared and mid part of result forms index.
  Eg. - hash(3101) = 3101 * 3101 = 9616201 ⇒ 162
* Digit-Folding Method: Key is divided into separate parts & combine all by using some operations.
  Eg. - hash(124655|12) = 124+655+12 = 791

Collision: Hash function returns same hash key for more than one record.



Eg. -
hash(42) = 42 % 10 = 2
hash(37) = 37 % 10 = 7
hash(75) = 75 % 10 = 5
hash(12) = 12 % 10 = 2

Collision Resolution Techniques is used to resolve collisions

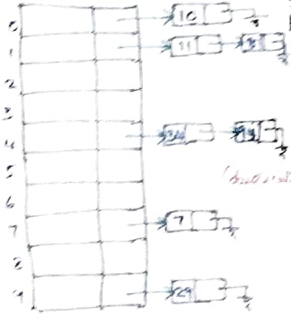
Collision Resolution Techniques

* Separate Chaining
* Open Addressing
  * Linear Probing
  * Quadratic
  * Double Hashing

Separate Chaining:

* Open Hashing
* No collision occurs
* Use linked list concept
* Utilize more memory space.
* Less Cache Performance

Example: Insert: 10, 11, 21, 10, 7, 34, 94, 29
(Table Size = 10)



Insert 10,
Hash(10) = 10 Mod 10 = 0
Insert 11,
Hash(11) = 11 % 10 = 1
Insert 21,
Hash(21) = 21 % 10 = 1
Insert 10,
Hash(10) = 10 % 10 = 1
Insert 7,
Hash(7) = 7 % 10 = 7
Insert 34,
Hash(34) = 34 % 10 = 4
Insert 94,
Hash(94) = 94 % 10 = 4
Insert 29,
Hash(29) = 29 % 10 = 9

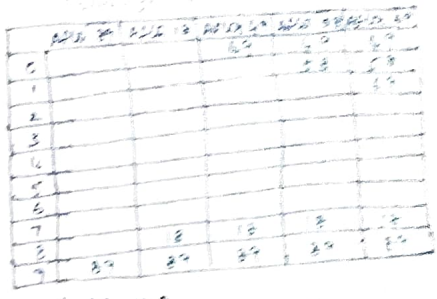
Separate Chaining has disadvantage of Separate Chaining we go for Open Addressing.

Open Addressing:

* Closed Hashing
* Use Array Concepts
* Alternate cells are tried until empty cell is found.
* 3 Strategies:
  * Linear Probing
  * Quadratic Probing
  * Double Hashing

Linear Probing:

Index = (Hash(Key) + i) % Hsize
H → Key; H → TableSize; i → Iteration
Eg - Insert Keys = {89, 18, 49, 58, 69}



Insert: 89, U=9
index = (9+0) % 10 = 9 Empty
Insert: 18, index = 18 % 10 = 8 ⇐ U
i=0, index = (8+0) % 10 = 8 Empty
Insert 49, index = 49 % 10 = 9 ⇐ U
i=0, index = (9+0) % 10 = 9 Occupied
i=1, index = (9+1) % 10 = 0 Empty
Insert 58, index = 58 % 10 = 8 ⇐ U
i=0, index = (8+0) % 10 = 8 Occupied
i=1, index = (8+1) % 10 = 9 Occupied
i=2, index = (8+2) % 10 = 0 Occupied
i=3, index = (8+3) % 10 = 1 Empty
Insert 69, index = 69 % 10 = 9 ⇐ U
i=0, index = (9+0) % 10 = 9 Occupied
i=1, index = (9+1) % 10 = 0 Occupied
i=2, index = (9+2) % 10 = 2 Empty

Quadratic Probing:
Index = (Hash(Key) + i²) % Hsize
U → Key; H → TableSize; i → Iteration
* Similar to Linear Probing

Double Hashing:
Index = (Hash1(Key) + i * Hash2(Key)) % Hsize
Hash1 → Hash value computed by another Hash function

Topic 13: Priority Queue

- Heap
- Heap Sort

TOPIC-13 (PRIORITY QUEUE, HEAP & HEAP SORT)

Priority Queue:

* Processing of an object based on priority.
* Priority Heap

Implementation:

→ Linked List
→ Binary Search Tree
→ Binary Heap

Binary Heap: Heap DS created using binary tree.

Heapify: process of creating binary heap DS either in Min_Heap or Max_Heap.

Types:

→ Max_Heap (parent > children)
→ Min_Heap (parent < children)

3 properties:

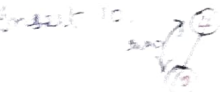
→ Shape property
  → Complete Binary Tree (BT)
→ Heap property
  → Max_Heap or Min_Heap

Operations:

Build, Insert, Delete.

Example: Build binary heap by inserting {4, 10, 3, 5, 1}

Insert 4:



Insert 10:

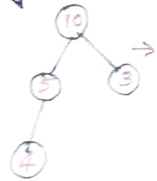
check properties:
→ Shape property - complete BT
→ Heap property - Not satisfied (Max_Heap). So swap (10, 4)

Insert 3,

check properties:
→ Shape property - satisfied
→ Heap property - satisfied (Max_Heap)

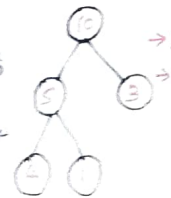
Insert 5,

check properties:
→ Shape property - satisfied
→ Heap property - Not " (Max_Heap). So swap (4, 5)



→ Heap property satisfied.

Insert 1,



Check Properties:
→ Shape property - Satisfied
→ Heap property - "

Binary Heap (Max_Heap) is created successfully.

Note:

In Max_Heap, maximum element will be at root.

In Min_Heap, minimum element will be at root.

Heap Sort:

Sorting technique based on Binary Heap DS

* 2 ways:
  → Min_Heap Sort - ascending order
  → Max_Heap Sort - descending order

Max_Heap Sort:

uses Delete_Max operation.

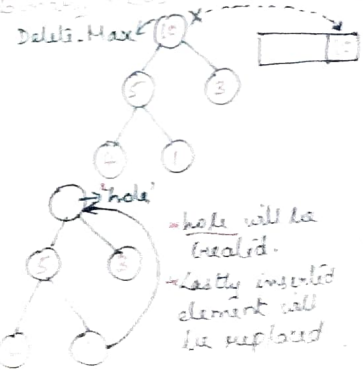
Example: Heap Sort - {4, 10, 3, 5, 1}

Steps:

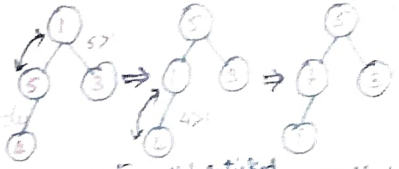
1) Construct Binary Heap (Max/Min)
2) Call Delete_Max / Delete_Min operation (root element - deleted)
3) Place the deleted element in the sorting list.
4) 'Hole' will be created.
5) Replace lastly inserted element at the root.
6) Check binary heap properties and satisfy it.
7) Repeat steps till get sorted list.

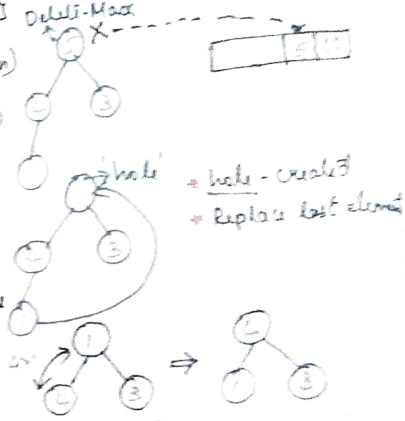
Now,

Solve {4, 10, 3, 5, 1} using Max Heap Sort (Ascending order)

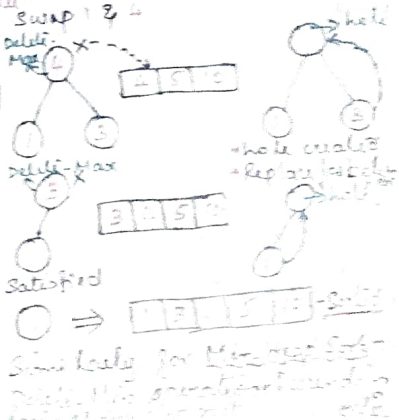


* hole will be created.
* Lastly inserted element will be replaced





* hole - created
* Replace last element



* hole created
* Rep are last element

Similarly for Min_Heap Sort -

Topic 14: Sorting

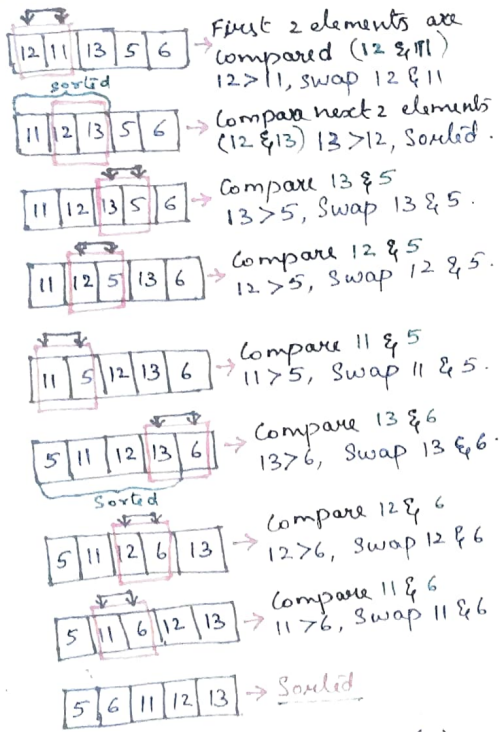
- Insertion Sort
- Merge Sort
- Radix Sort

TOPIC-14 (INSERTION SORT, MERGE SORT, RADIX SORT)

Insertion Sorting:

*Simple
*Efficient for small data values.

Example: Sort 12,11,13,5,6 Using Insertion sort.

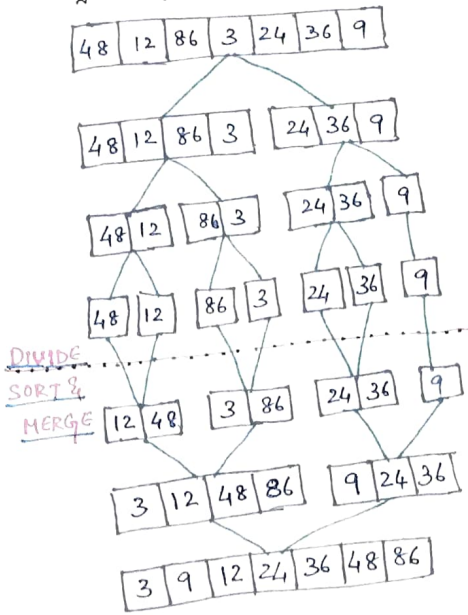


Best Case Time Complexity - $O(n)$
Worst Case Time Complexity - $O(n^2)$
Average Case Time Complexity - $O(n^2)$

Merge Sort:

* Divide & Conquer Method
* Recursively dividing the array into 2 halves, sort & then merge

Example:- Sort 48,12,86,3,24,36,9 Using merge sort.



Sorted

Time Complexity - $O(n \log n)$

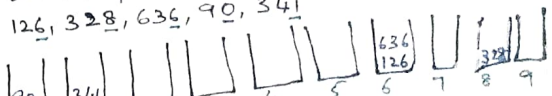
Radix Sort:

*Non Comparative integer Sorting algorithm
* Digit-by-digit Sorting (Sorting done from least significant digit)

Example:- Sort 126, 328, 636, 90, 341 Using Radix Sort.

Pass I: Consider the 1's place and keep it in respective buckets (0-9)

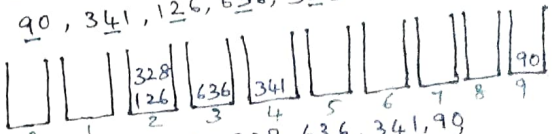
126, 328, 636, 90, 341



→Arrange:- 90, 341, 126, 636, 328
input for Pass II

Pass II: Consider the 10's place and keep it in respective buckets (0-9)

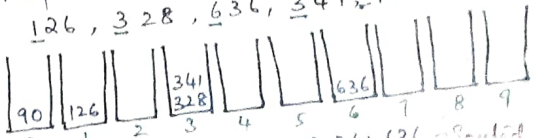
90, 341, 126, 636, 328



→Arrange:- 126, 328, 636, 341, 90
input for Pass III

Pass III: Consider the 100's place and keep it in respective buckets (0-9)

126, 328, 636, 341, 90



→Arrange:- 90, 126, 328, 341, 636 - Sorted

Note:- No. of digits = No. of Passes

Time Complexity - $O(nd)$
(n → array size, d = no of digits)

Topic 15: Quick Sort

- Divide & Conquer
- Time Complexity

TOPIC - 15 (QUICK SORT)

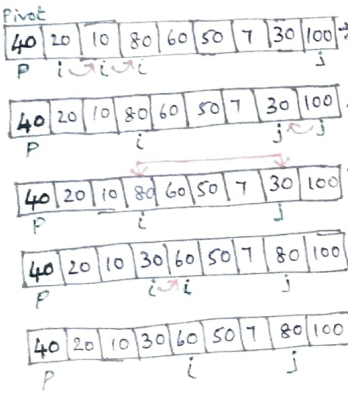
Quick Sort:

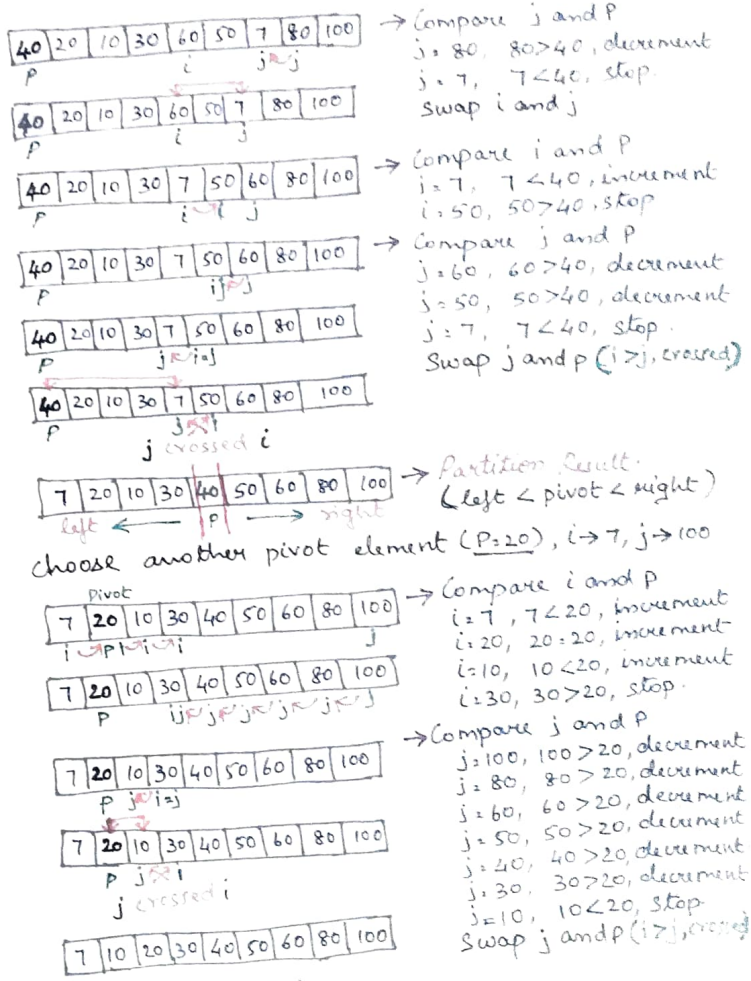
* Divide & Conquer Method
  - Divide - partition the array into 2 sub-arrays
  - Conquer - recursively, sort 2 subarrays
  - Combine - combine the already sorted array
* choosing a pivot element - [Mean, Median, First, Last]

Steps:-

Step 1: Choose a pivot element from the list.
Step 2: Define 2 variables i & j. i → 1st & j → last value
Step 3: Increment i until list[i] > pivot, then stop
Step 4: Decrement j until list[j] < pivot, then stop
Step 5: If i < j, then swap list[i] and list[j]
Step 6: Repeat steps 3, 4, 5 until i > j.
Step 7: If i > j (crossed), then swap crossed index with pivot element.
* Partition Result → left element < pivot < right element
Step 8: Choose another pivot element & repeat the steps till gets sorted.

Example: Sort 40, 20, 10, 80, 60, 50, 7, 30, 100 Using Quick sort. pivot (P) → 40, i → 20, j → 100 ⇐ Consider.





Best Case Time Complexity - $O(n \log n)$
Worst Case Time Complexity - $O(n^2)$
Average Case Time Complexity - $O(n \log n)$

Topic 16: Sorting

- Bubble Sort
- Selection Sort
- Time Complexity

TOPIC 16: BUBBLE SORT, SELECTION SORT, TIME COMPLEXITY

BUBBLE SORT:

* Internal Sorting Algorithm
* Comparing two consecutive elements (a[i] with a[i+1])
* Largest element in the array bubbles towards the right end of first pass
* Repeated for all pairs till Sorted
* (n-1) passes where 'n' is the no. of elements to be sorted

Algorithm:

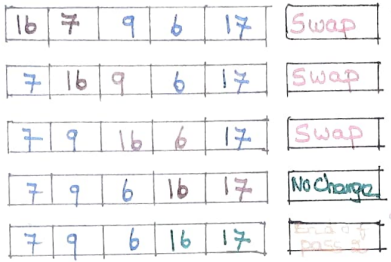
```
for (i=0; i<n-1; i++)
{  for(j=0; j<n-1; j++)
   {  if(a[j] > a[j+1])
      {  temp = a[j];
         a[j] = a[j+1];
         a[j+1] = temp;
      }
   }
}
```

Example:

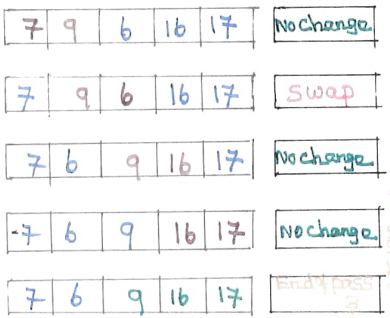
Pass 1:

Largest element is placed at last index

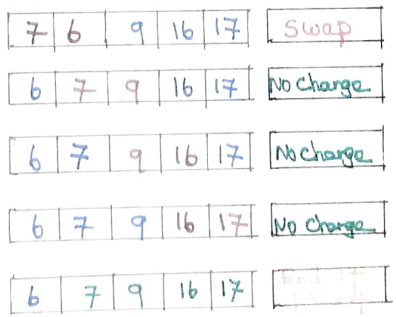
Pass 2:



Pass 3:



Pass 4:



O/P: 6 7 9 16 17

Sorted order

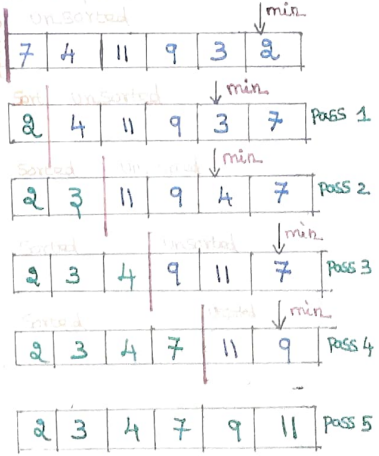
SELECTION SORT:

* Repeat until no unsorted element remain
  -> Search the unsorted part of the data to find smallest value
  -> Swap smallest found value with the first element of the unsorted part
* It contains N-1 passes

Algorithm:

```
for(i=0; i<n-1; i++)
{  int min = i;
   for(j=i+1; j<n; j++)
   {  if (a[j] < a[min])
      {  min = j;
      }
   }
   temp = a[i];
   a[i] = a[min];
   a[min] = temp;
}
```

Example:



Sorted order

2 3 4 7 9 11

Sorting	Advantage	Disadvantage
Bubble Sort	* Simple & easy for implement * Efficient when input data is sorted	Inefficient for large volume of input data
Selection Sort	* Efficient for small amount of I/p data * 60% more efficient than bubble sort	* perform all n comparisions for sorted I/p data * Inefficient for large volume of input data * Take more time

Time Complexity:

	Time Complexity			Space
	Best	Average	Worst	Worst
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

Topic 17: Graph

- Terminologies
- Types
- Topological Sort

TOPIC - 17 (GRAPH - TERMINOLOGIES - TYPES - TOPOLOGICAL SORT)

Graphs: $G = (V, E)$

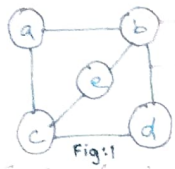


Fig:1

→ Vertices, $V = \{a, b, c, d, e\}$

→ edges, $E = \{(a,b), (a,c), (b,d), (d,c), (c,e), (b,e)\}$

Terminologies: (Refer Fig:1)

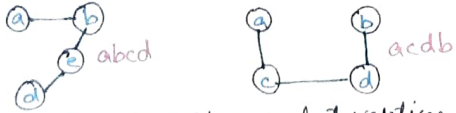
Adjacent vertices → Connected by an edge a → b, c ; c → a, e, d

Length → No. of edges.

Degree → No. of adjacent vertices

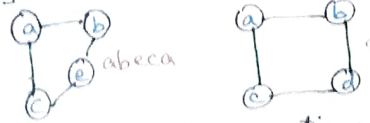
Degree |a| = 2, Degree |b| = 3

Path → Sequence of vertices.

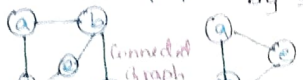


Simple path → No repeated vertices.

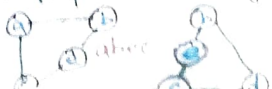
Cycle → Last & first vertex - Same.



Connected Graph → 2 vertices connected by some path.



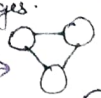
Sub Graph → Subset of vertices & edges



Connected Components - max. connected Subgraph

Directed Graph (Digraph): specify directions in edges. →

Undirected Graph: No directed edges. →



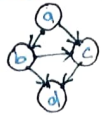
Weighted Graph: have weights in edges →

Complete Graph: all vertices are connected to each other (Undirected Graph) →

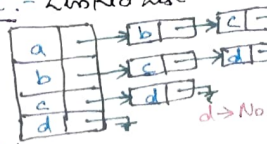
Strongly Connected Graph: all vertices are connected in both directions (Directed Graph) →

Acyclic Graph: No cycles. →

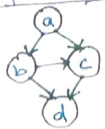
Adjacency List :- Linked list.



→



Adjacency Matrix :-



→

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

* 4 Vertices
* 4 x 4 Matrices

Topological Sort : Used in Directed Acyclic Graph (DAG) - directed & No cycles.

* Sorting - DAGs.
* if u → v, then v appears after u

Algorithm steps:

→ Compute indegrees of all vertices

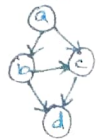
→ Find vertex 'v' with indegree 0, place in ordered list

→ Remove 'v' and its edges (v, x)

→ Update indegree of remaining vertices

→ Repeat till all vertices are in ordered list

Example:



⟹

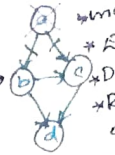
	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

indegree[a] = 0, indegree[b] = 1
indegree[c] = 2, indegree[d] = 2

Vertex	1	2	3	4
a	0			
b	1			
c	2			
d	2			

Enqueue : a
Dequeue : a

→



* indegree[a] = 0.
* Enqueue a
* Dequeue a → list.
* Remove adj. edges of a → b, c

indegree[b] = 0, indegree[c] = 1

V	1	2	3	4
a	0	0	0	0
b	1	0		
c	2	1		
d	2	2		

Enqueue : ab
Dequeue : ab

→

* indegree[b] = 0
* Enqueue : b
* Dequeue b → list
* Remove adj. edges of b → c, d

indegree[c] = 0, indegree[d] = 1

V	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	
d	2	2	1	

Enqueue : abc
Dequeue : abc

→

* indegree[c] = 0
* Enqueue : c
* Dequeue : c → list
* Remove adj. edges of c → d.

indegree[d] = 0

V	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	0
d	2	2	1	0

Enqueue : d
Dequeue : abcd

→

* Enqueue : d
* Dequeue : d → list
* No adj. edges for d.

Topic 18: Shortest Path Algorithms

- Unweighted Shortest Path
- Dijkstra's Algorithm

TOPIC-18 SHORTEST PATH ALGORITHMS - UNWEIGHTED SHORTEST PATH - DIJKSTRA'S ALGORITHM

Shortest Path Algorithms

→ Finds minimum cost from source to other vertex.

Types:

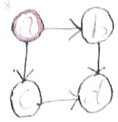
→ Single Source Shortest path (SSSP)
→ All Pair Shortest paths (APSP)

Single Source Shortest Path

Steps:

1. Source Node - 'S', Enqueue
2. Dequeue 'S' → known as 1 and find it adjacency vertex.
3. Distance, dv → Source vertex distance increment by 1 & enqueue the vertex.
4. Repeat from step (2) until queue becomes empty.

Example:



a → Source Node

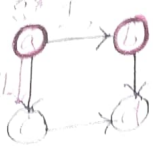
V	known	dv	pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue :
Dequeue :

V	known	dv	pv
a	1	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

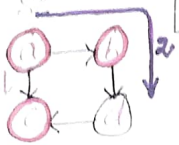
a → known
Enqueue : a
Dequeue : a



V	known	dv	pv
a	1	0	0
b	1	1	a
c	0	1	a
d	0	0	0

Enqueue : b c
Dequeue : a b

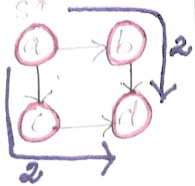
dv(b) = 1
dv(c) = 1
Enqueue : b, c
Dequeue : b



V	known	dv	pv
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b

Enqueue : c, d
Dequeue : a, b, c

dv(a)
a → b → d (2)
a → c → d (2)
Same distance



V	known	dv	pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

Queue : Empty

Initial configuration

V	known	dv	pv
a	0	∞	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue :
Dequeue :

V	known	dv	pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

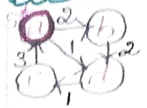
Queue : Empty

Shortest path from source vertex 'a'
a → b = 1
a → c = 1
a → d = 2

Dijkstra's Algorithm (Weighted)

→ Single source shortest path algorithm
→ Similar to unweighted SSSPA, Needs to calculate the distance using weights.

Example:

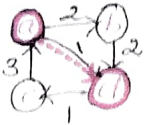


Initial configuration

V	known	dv	pv
a	0 →	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

a → known (1)

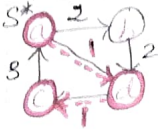


a → b = 2
a → d = 1
Shortest path

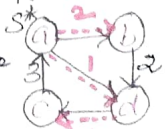
V	known	dv	pv
a	1	0	0
b	0	2	a
c	0	∞	0
d	1	1	a

Enqueue : b d
Dequeue : a d

Enqueue : d
Dequeue : d
(Path from a)



a → d → c = 2
Enqueue : c
Dequeue : c
Path from d



a → b = 2
Enqueue : b
Dequeue : b

V	known	dv	pv
a	1	0	0
b	1	2	a
c	1	∞	0
d	1	1	a

Enqueue : c
Dequeue : a d c

V	known	dv	pv
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

Enqueue : b
Dequeue : a b c b.

Shortest Path from Source Vertex 'a'
a → b = 2, a → c = 2
a → d = 1

Topic 19: Minimum Spanning Tree (MST)

- Prim's Algorithm
- Kruskal's Algorithm

TOPIC-19 MINIMUM SPANNING TREE - PRIM'S & KRUSKAL'S ALGORITHM

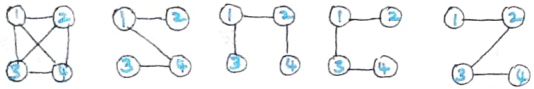
Minimum Spanning Tree (MST)

Spanning Tree - Connected Graph, & No cycles
Subgraph with all Vertices

Minimum Spanning Tree - Spanning tree with minimum cost.

n^{n-2} Spanning trees will obtain for n nodes/ Vertex

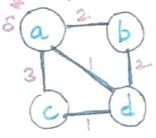
Example: 4 Nodes $[n^{n-2} = 4^2 = 16$ Spanning trees$]$



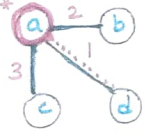
Algorithms:
* Prim's Algorithm
* Kruskal's Algorithm

Prim's Algorithm:
↳ Uses Greedy Techniques
↳ Based on Vertices.

Example:

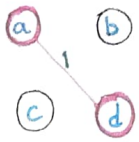


V	KNOWN	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

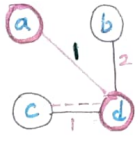


V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	3	a
d	0	1	a

a → d (1) minimum distance
d → KNOWN (1), path from 'a'

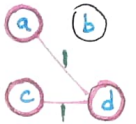


V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	3	a
d	1	1	a



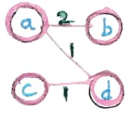
V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	1	d
d	1	1	a

adjacent Vertices
d → b, c
d → c = 1
d → b = 2
(Min. cost)



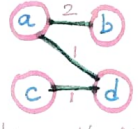
V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	1	1	d
d	1	1	a

C → KNOWN
Path from 'd'



V	KNOWN	dv	Pv
a	1	0	0
b	1	2	a
c	1	1	d
d	1	1	a

Remaining Vertex 'b',
a → b = 2,
d → b = 2
(Same)



Minimum Spanning Tree

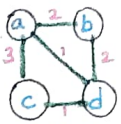
Minimum Cost:

$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

Kruskal's Algorithm:
↳ Uses Greedy Techniques.
↳ Based on Edges.

Steps:
1) Remove all loops & parallel edges (High cost)
2) Arrange all edges in Ascending Order.
3) Add the edges which has the least cost.

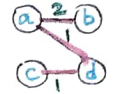
Example:



Edge	dv
(a,b)	2
(a,c)	3
(a,d)	1
(b,d)	2
(c,d)	1

⇒ Sort edges in Ascending order

Edge	dv
(a,d)	1
(c,d)	1
(a,b)	2
(b,d)	2
(a,c)	3



⇐

Edge	dv	Visit
(a,d)	1	✓
(c,d)	1	✓
(a,d)	1	✓
(b,d)	2	X
(a,c)	3	X

cycle (bdab)
cycle (adca)

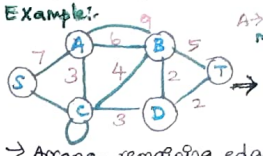
Minimum Spanning Tree

Minimum Cost:

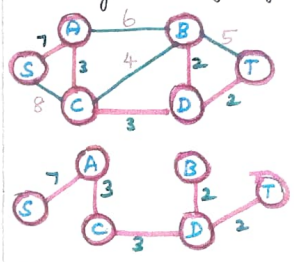
$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

→ Remove loops & Parallel edges

Example:



→ Arrange remaining edges in Ascending order.



Edge	dv	Visit
(B,D)	2	✓
(D,T)	2	✓
(A,C)	3	✓
(C,D)	3	✓
(B,A)	4	✓
(B,T)	5	X
(A,B)	6	X
(S,A)	7	✓
(S,C)	8	X

cycle (BDTB)
cycle (ACDBA)
cycle (SACS)

Minimum Spanning Tree

Minimum Cost: $C_{S,A} + C_{A,C} + C_{C,D} + C_{D,B} + C_{D,T}$
$= 7+3+3+2+2 = 17$

Applications

Google Maps

Topic 20: Graph Traversal

- Breadth First Search
- Depth First Search

TOPIC-20 (BREADTH FIRST SEARCH, DEPTH FIRST SEARCH)

Breadth First Search (BFS)

* Traverses Breadth-wise
* Uses Queue concept (FIFO)

Applications:

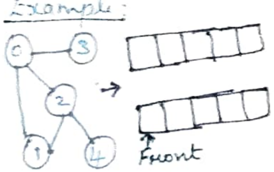
* GPS Navigation
* Path Finding Algorithms
* Minimum Spanning Tree

↳ visited & not visited

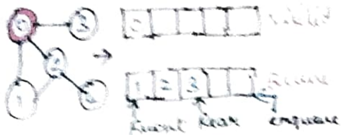
Algorithm steps

1) Create Queue - size (No of vertices)
2) Enqueue any one (source) vertex - visited list
3) Enqueue front item from Queue to visited list. Also enqueue its adjacent vertices to Queue.
4) Repeat the steps till Queue is empty.

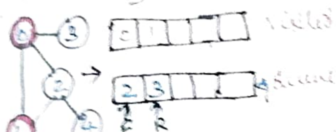
Example:



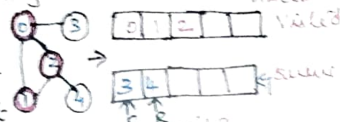
Vertex, 0 → Source.
Enqueue 0 → visited
Adjacent vertices (0) → 1, 2, 3 Queue



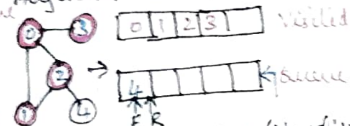
Dequeue front item '1' & move to visited list
Enqueue 1 → visited
Adj. vertices (1) → Queue (0, 2) → already visited



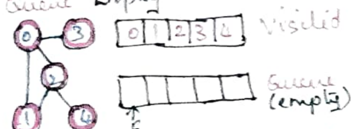
Dequeue '2' (Queue) & Enqueue → visited
Adjacent vertices (2) → 0, 1, 4 → Queue visited



Enqueue 3 → visited
Adjacent vertices (3) → 0 → visited



Enqueue 4 → visited (No adj. vertex)
Queue Empty.



Note: Similarly, for directed graphs, follow adjacent nodes (outdegree)

Advantage:

* less memory space & time period

Depth First Search (DFS)

* Traverse depthwise (top-bottom)
* Uses Stack concept (LIFO)

Applications:

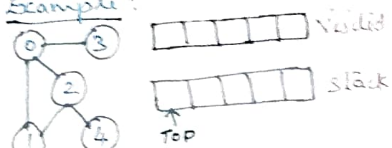
* Topological sorting
* Detect cycles in the graph

↳ visited & not visited

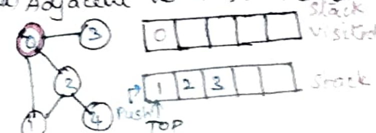
Algorithm steps

1) Create stack - size (No of vertices)
2) Insert any one (source) vertex - visited list.
3) Push top item from stack to visited list. Also, push its adjacent vertices to stack.
4) Repeat the steps till stack is empty.

Example:



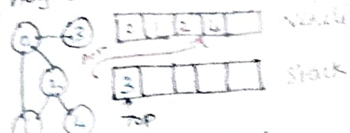
Vertex, 0 → Source
Insert 0 → visited
Adjacent vertices (0) → 1, 2, 3 stack visited



Pop Top item '1' & move to visited list
Adj. vertices (1) → stack (0, 2) visited

Pop '2' (stack), Push → visited
Adjacent vertices (2) → 0, 1, 4 → stack visited

Pop '4' (stack), Push → visited
Adjacent vertices (4) → 2 (visited)

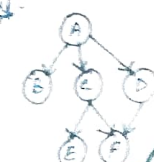


Pop '3' (stack), Push → 3 visited
(No adjacent vertices) stack empty

Advantage:

* Solutions will definitely found

Activity:



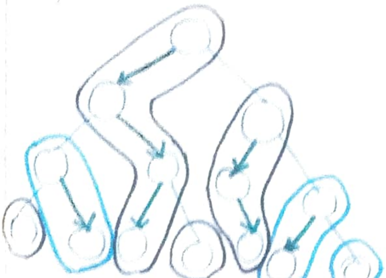
BFS : ABCDEF
DFS : ADFCEB

* GPS Navigation System
* Google Map

Topic 21: Advanced Data Structures

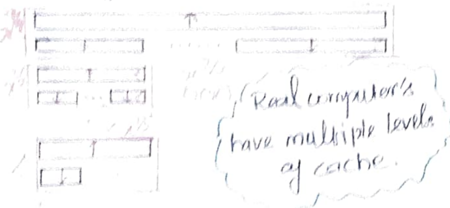
- Advanced Data Structures
- Future Scope
- Real Time Applications
- Research Area

TOPIC - 2.1



* Binary Search Tree (BST) that's as good as all others we still don't know, but we're close.

Memory Hierarchy



* We can optimize the number of cache misses, often without even knowing the size of the cache.

Future of Data Structure

* Problem's in handling data
* Never ending growth of data
* Increased demand for Data Structure.

* Emerging Sources of data

* MATLAB experience in accelerating Direct-Gree Algorithm for constrained global optimization through dynamic data structure and parallelization.

* A data driven analysis on bridging techniques for heterogeneous materials and structures.

* data structure in proline - RNA interaction structure

* Parallel Algorithms in cached shared memory.

* Sensor data acquisition and asset maintenance using ...

* "..." processing algorithms for organizing and searching and deducing connection's in vast data collection.

* "Numerical" Algorithm for accurate and stable solution of differential equations, especially in parallel.

* image viewer software
* Train coaches
* Escalator's

Stack

* Converting infix - postfix expression
* History of visited website.
* call logs
* Recursions
* Media play lists
* Java virtual machine
* Pile of Dinner plates
* Stacked chairs

Queue

* Operating systems - Job scheduling.
* CPU scheduling
* Escalator's
* Printer Spooler
* Handle website traffic
* Sending email
* Ticket window
* Vehicles - Toll - tax badge
* Networking - Router's and Switches

* Data Sizes
* Computer Graphics
* Queue

* Shortest path
* D-Game Engine
* Data Analysis
* Data Mining
* Dictionary App
* Airline Scheduling
* Sudoku's puzzles
* Search Engine