Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Continuous Assessment for Laboratory / Assignment sessions

Academic Year 2022-23

Name: Ayush Jain SAP ID: 60004200132

Course: Advance Algorithm Course Code: DJ19CEC602

Year: **T.Y. B.Tech.** Sem: VI Batch: B3

Department: Computer Engineering

Performance Indicators (Any no. of Indicators) (Maximum 5 marks per indicator)	1	2	3	4	5	6	7	8	9	10	11		Avg	A1	A2		Avg
Course Outcome	1	1	2	2	~~3~~	~~3~~	~~3~~	~~4~~ 3	5	4	6						
1. Knowledge (Factual/Conceptual/Procedural/ Metacognitive)	5	5	5	5	4	4	5	5						5	5		
2. Describe (Factual/Conceptual/Procedural/ Metacognitive)	4	5	5	5	4	4	5	4						4	4		
3. Demonstration (Factual/Conceptual/Procedural/ Metacognitive)	5	4	5	5	5	4	4	4						5	4		
4. Strategy (Analyse & / or Evaluate) (Factual/Conceptual/ Procedural/Metacognitive)	5	5	4	5	4	5	4	5						4	5		
5. Interpret/ Develop (Factual/Conceptual/ Procedural/Metacognitive)	–	–	–	–	–	–	–	–	–					–	–		
6. Attitude towards learning (receiving, attending, responding, valuing, organizing, characterization by value)	5	5	5	4	5	5	4	5						5	5		
7. Non-verbal communication skills/ Behaviour or Behavioural skills (motor skills, hand-eye coordination, gross body movements, finely coordinated body movements speech behaviours)	–	–	–	–	–	–	–	–	–					–	–		
Total	24	24	24	24	22	22	22	22						23	23		
Signature of the faculty member																	

Outstanding (5), Excellent (4), Good (3), Fair (2), Needs Improvement (1)

Laboratory marks Avg. = 23	Assignment marks Avg. = 23	Total Term-work (25) = 23
Laboratory Scaled to (15) = 14	Assignment Scaled to (10) = 09	Sign of the Student:

Signature of the Faculty member:
Name of the Faculty member: Ms. Chetashri B

Signature of Head of the Department
Date: 8/5/23

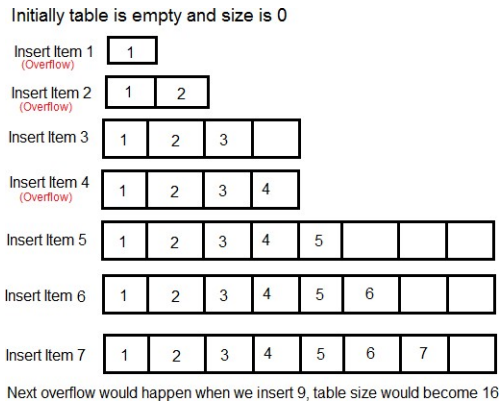
ADVANCE ALGORITHM
Experiment 1

Ayush Jain 60004200132 B3

Aim : To implement Amortized Analysis.

Theory:
Amortized Analysis.

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time which is lower than the worst-case time of a particular expensive operation. The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.
Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.

1. Allocate memory for a larger table of size, typically twice the old table.
2. Copy the contents of old table to new table.
3. Free the old table.

Aggregate Method

The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is −

$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

The Accounting Method

The accounting method is aptly named because it borrows ideas and terms from accounting. Here, each operation is assigned a charge, called the amortized cost. Some operations can be charged more or less than they actually cost. If an operation's amortized cost exceeds its actual cost, we assign the difference, called a credit, to specific objects in the data structure. Credit can be used later to help pay for other operations whose amortized cost is less than their actual cost. Credit can never be negative in any sequence of operations.

The Potential Method

The potential method is similar to the accounting method. However, instead of thinking about the analysis in terms of cost and credit, the potential method thinks of work already done as potential energy that can pay for later operations. This is similar to how rolling a rock up a hill creates potential energy that then can bring it back down the hill with no effort. Unlike the accounting method, however, potential energy is associated with the data structure as a whole, not with individual operations.

Code:

Aggregate Table:

```
class DynamicTable:
    def __init__(self, capacity=1):
        self.table = [0] * capacity
        self.size = 0
        self.capacity = capacity

    def add(self, element):
        if self.size == self.capacity:
            # Double the capacity of the table if it is full
            new_table = [0] * (self.capacity * 2)
            for i in range(self.size):
                new_table[i] = self.table[i]
            self.table = new_table
            self.capacity *= 2
            # print(f"Table doubled, new size: {self.capacity}")
        self.table.append(element)
        self.size += 1

    def size(self):
        return self.size

    def capacity(self):
        return self.capacity

    def numDoublings(self):
        # Calculate the number of times the table was doubled
        return int(math.log2(self.capacity))

    def numCopyings(self):
        # Calculate the number of times elements were copied during resizing
        num_copyings = 0
        for i in range(1, self.numDoublings() + 1):
            num_copyings += 2**(i-1)
        return num_copyings

table = DynamicTable()
cost = 0
f = 0
operation_cost = 1

print("Item No\tTable Size\tTable Cost\tCost of Operation")
print("====================================================")

for i in range(1, 18):
    table.add(i)
    if f == 1:
```

```
        cost = table.size
        f = 0
    else:
        cost = 1
    if table.size == table.capacity:
        f = 1
    print(f"{i}\t{table.capacity}\t\t{cost}\t\t\t{operation_cost}")
```

Output:

```
Item No Table Size      Table Cost      Cost of Operation
=====================================================
1       1               1                       1
2       2               2                       1
3       4               3                       1
4       4               1                       1
5       8               5                       1
6       8               1                       1
7       8               1                       1
8       8               1                       1
9       16              9                       1
10      16              1                       1
11      16              1                       1
12      16              1                       1
13      16              1                       1
14      16              1                       1
15      16              1                       1
16      16              1                       1
17      32              17                      1


...Program finished with exit code 0
Press ENTER to exit console.
```

Accounting using Multipop
Code:

```
class Stack:
    def __init__(self):
        self.items = []
        self.cost = 0
        self.balance = 0

    def push(self, item):
        self.items.append(item)
        self.cost += 1
        self.balance += 1

    def pop(self):
        if not self.is_empty():
            self.cost += 1
            self.balance -= 1
            return self.items.pop()

    def multi_pop(self, k):
        if k > len(self.items):
            k = len(self.items)
        for i in range(k):
            self.pop()

        self.cost += k
        self.balance -= k

    def is_empty(self):
        return len(self.items) == 0

    def get_cost(self):
        return self.cost

    def get_balance(self):
        return self.balance

    def display_table(self, n):
        print("{:<15}{:<15}{:<15}{:<15}".format("Operation", "Total Cost",
"Amortized Cost", "Balance"))
        print("-"*60)
        for i in range(1, n+1):
            if i % 3 == 1:
                self.push(i)
                print("{:<15}{:<15}{:<15.2f}{:<15}".format(f"push({i})",
self.get_cost(), self.get_cost()/i, self.get_balance()))
            elif i % 3 == 2:
                self.pop()
```

```
                print("{:<15}{:<15}{:<15.2f}{:<15}".format("pop()",
self.get_cost(), self.get_cost()/i, self.get_balance()))
            else:
                self.multi_pop(i//3)
                print("{:<15}{:<15}{:<15.2f}{:<15}".format(f"multi_pop({i//3})
", self.get_cost(), self.get_cost()/i, self.get_balance()))
        print()


n = int(input("Enter the number of operations to perform on the stack: "))


stack = Stack()
stack.display_table(n)
```

Output:

```
Enter the number of operations to perform on the stack: 5
Operation      Total Cost     Amortized Cost Balance
-------------------------------------------------------------
push(1)        1              1.00           1
pop()          2              1.00           0
multi_pop(1)   2              0.67           0
push(4)        3              0.75           1
pop()          4              0.80           0



...Program finished with exit code 0
Press ENTER to exit console.
```

Potential Method:

```
class DynamicTable:
    def __init__(self, capacity=1):
        self.table = [0] * capacity
        self.size = 0
        self.capacity = capacity

    def add(self, element):
        if self.size == self.capacity:
            # Double the capacity of the table if it is full
            new_table = [0] * (self.capacity * 2)
            for i in range(self.size):
                new_table[i] = self.table[i]
            self.table = new_table
            self.capacity *= 2
        self.table[self.size] = element
        self.size += 1

    def size(self):
        return self.size

    def capacity(self):
        return self.capacity

def potential(table):
    # Potential function, defined as the double of the number of elements
minus the size of the table
    return 2*table.size - table.capacity

def cost(table, operation):
    # Calculate the amortized cost of the operation
    if operation == "add":
        if table.size == table.capacity:
            # Double the table
            return potential(table) + 1
        else:
            return 1
    else:
        raise Exception("Invalid operation")

table = DynamicTable()
total_cost = 0
operation_cost = 1
operation_cos = 1
prev = 0
print("Item No.\tTable Size\tPotential\tOperation Cost\tTotal Cost\tAmortized
Cost")
for i in range(1, 18):
    operation_cost = cost(table, "add")
```

```
    total_cost += operation_cost
    table.add(i)
    # print("="*8)
    pot = potential(table)
    amortized_cost = operation_cost + (pot - prev)
    prev = pot
    # print(amortized_cost)
    print(f"{i}\t\t  {table.capacity}\t\t  {potential(table)}\t\t  {operation_
cos}\t\t  {operation_cost}\t\t  {amortized_cost}")
```

Output:

```
Item No.        Table Size      Potential       Operation Cost  Total Cost      Amortized Cost
1                 1               1               1               1               2
2                 2               2               1               2               3
3                 4               2               1               3               3
4                 4               4               1               1               3
5                 8               2               1               5               3
6                 8               4               1               1               3
7                 8               6               1               1               3
8                 8               8               1               1               3
9                 16              2               1               9               3
10                16              4               1               1               3
11                16              6               1               1               3
12                16              8               1               1               3
13                16              10              1               1               3
14                16              12              1               1               3
15                16              14              1               1               3
16                16              16              1               1               3
17                32              2               1               17              3


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: In conclusion, we learned the Amortized Analysis of the Algorithm where an occasional operation is very slow, but most of the other operations are faster.

ADVANCE ALGORITHM
Experiment 2

Ayush Jain 60004200132 B3

Aim : To implement Hiring Problem.

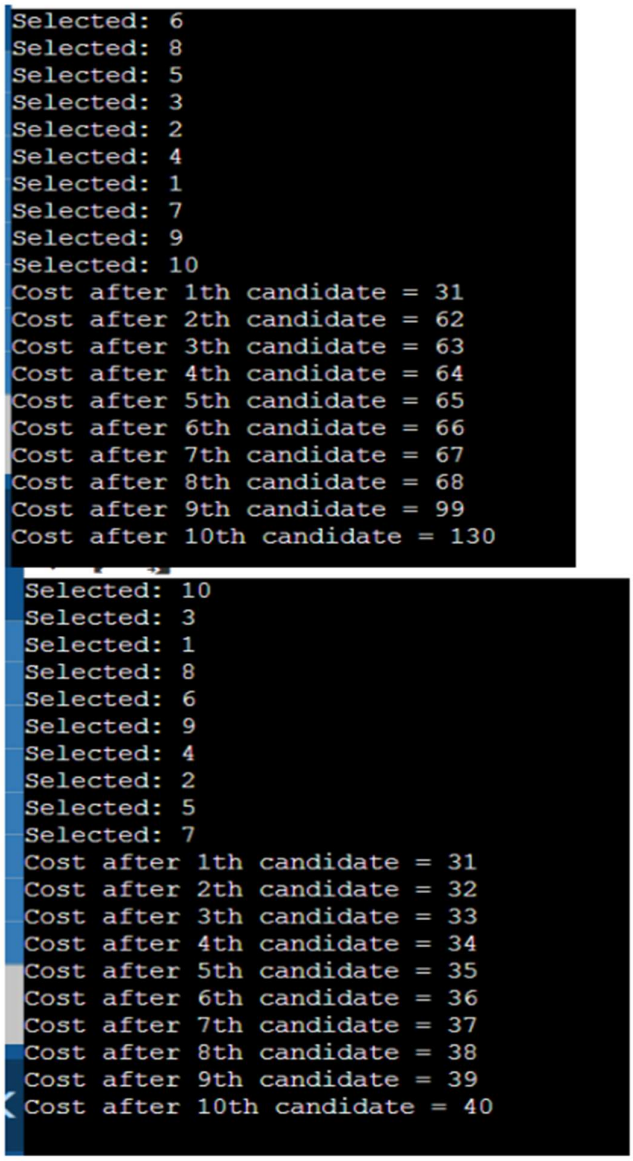
Theory:
Amortized Analysis.

Hiring Problem. The hiring problem is a classic problem in probability theory and decision theory that involves selecting the best candidate from a pool of applicants who arrive sequentially. The goal is to maximize the probability of selecting the best candidate while minimizing the expected number of applicants that are interviewed. One common approach to the hiring problem is to use the "optimal stopping" or "secretary problem" strategy. This strategy involves interviewing a fixed number of applicants (n) and then selecting the best candidate out of the n. The value of n is determined by a formula that depends on the total number of applicants and the probability distribution of their quality. Another approach to the hiring problem is to use a Bayesian decision-theoretic framework. This approach involves updating a prior probability distribution on the quality of the applicants as each new applicant is interviewed. The decision to hire a candidate is then based on the updated probability distribution and a decision threshold that balances the cost of making a mistake (hiring a suboptimal candidate) with the cost of delaying the decision. Other decision-making strategies have also been proposed for the hiring problem, including the "rank and select" strategy, which involves ranking the candidates as they arrive and then selecting the best candidate from a fixed number of the top-ranked candidates, and the "threshold acceptance" strategy, which involves setting a threshold quality level for the applicants and accepting the first applicant who meets or exceeds that threshold. In practice, the optimal strategy for the hiring problem may depend on the specific details of the hiring situation, such as the size of the applicant pool, the distribution of applicant qualities, and the cost of interviewing candidates.

Code:

```
import java.util.*;

public class Main
{
public static void main(String[] args) {
int order[]=new int[10];
int k=0,hc=20,fc=10,threshold=3,cost=0;
ArrayList<Integer> list = new ArrayList<Integer>(10);
 for(int i = 1; i <= 10; i++) {
 list.add(i);
 }
 Random rand = new Random();
 while(list.size() > 0) {
 int index = rand.nextInt(list.size());
 System.out.println("Selected: "+list.get(index));
 order[k++]=list.remove(index);
 }
 for(int j=0;j<10;j++){
 cost++;
 if(order[j]>threshold){
 cost+=hc+fc;
 threshold=order[j];
 }
 System.out.println("Cost after "+(j+1)+"th candidate = "+cost);
 }
}
}
```

Output:



Conclusion: In conclusion, we learned the Hiring Problem where by randomization we can decrease the cost for hiring a best candidate

ADVANCE ALGORITHM
Experiment 3

Ayush Jain 60004200132 B3

Aim : To implement Randomized Quick Sort.

Theory:

Randomized QuickSort is a sorting algorithm that uses the divide-and-conquer approach to sort an array of elements. The basic idea of QuickSort is to partition the array into two sub-arrays, where all elements in the left sub-array are less than or equal to a pivot element, and all elements in the right sub-array are greater than the pivot.

The algorithm then recursively sorts the left and right sub-arrays. The pivot element is chosen randomly in the array to ensure that the algorithm's worst-case time complexity is avoided.

Here's the theory for the randomized QuickSort algorithm:

1. Partitioning:
    - Choose a random pivot element from the array.
    - Rearrange the elements in the array such that all elements less than or equal to the pivot element are on the left, and all elements greater than the pivot element are on the right.
    - Return the index of the pivot element.
2. Recursive Sorting:
    - Recursively sort the left sub-array using the partition function.
    - Recursively sort the right sub-array using the partition function.
3. Termination:
    - If the sub-array to be sorted has length 1 or 0, then it is already sorted, and no further action is needed.

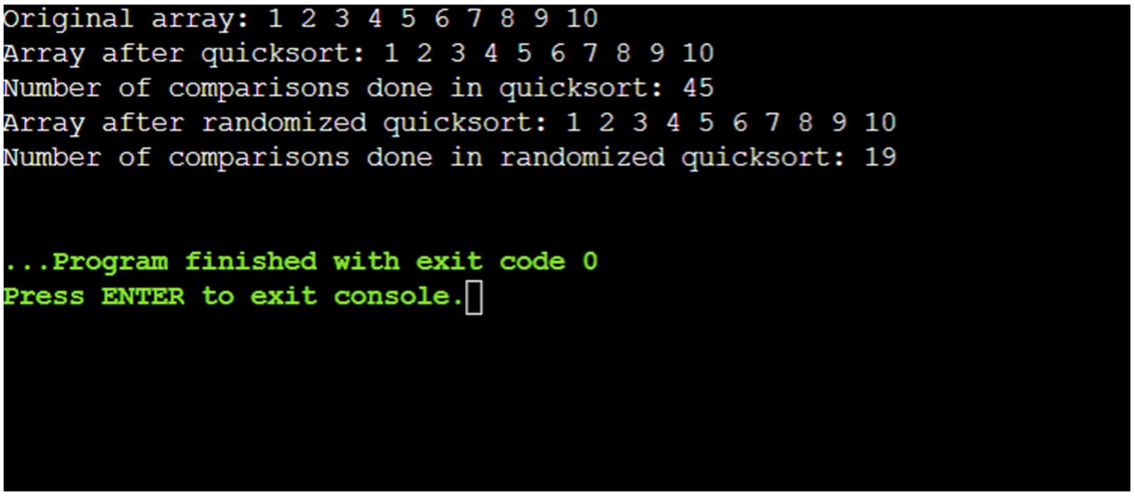
The worst-case time complexity of QuickSort is O(n^2), but the randomized version of QuickSort has an average-case time complexity of O(nlogn). This is because the randomized pivot selection ensures that the probability of choosing a bad pivot is low, leading to a more balanced partition of the array.

Code:

```
#include <iostream>
#include <algorithm>
#include <random>

using namespace std;

int partition(int arr[], int low, int high, int& comparisons) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j <= high - 1; j++) {
        comparisons++;
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quicksort(int arr[], int low, int high, int& comparisons) {
    if (low < high) {
        int pi = partition(arr, low, high, comparisons);
        quicksort(arr, low, pi - 1, comparisons);
        quicksort(arr, pi + 1, high, comparisons);
    }
}

int randomized_partition(int arr[], int low, int high, int& comparisons) {
    int i = rand() % (high - low + 1) + low;
    swap(arr[i], arr[high]);
    return partition(arr, low, high, comparisons);
}


void randomized_quicksort(int arr[], int low, int high, int& comparisons) {
    if (low < high) {
        int pi = randomized_partition(arr, low, high, comparisons);
        randomized_quicksort(arr, low, pi - 1, comparisons);
        randomized_quicksort(arr, pi + 1, high, comparisons);
    }
}
int main()
{
     int arr[] = {1,2,3,4,5,6,7,8,9,10};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    int comparisons = 0;
    quicksort(arr, 0, n - 1, comparisons);

    cout << "Array after quicksort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Number of comparisons done in quicksort: " << comparisons <<
endl;

    comparisons = 0;
    randomized_quicksort(arr, 0, n - 1, comparisons);

    cout << "Array after randomized quicksort: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Number of comparisons done in randomized quicksort: " <<
comparisons << endl;

    return 0;
}
```

Output:



Conclusion: In conclusion, we learned that the number of swaps can be decreased by randomization of elements in array while quicksort.

ADVANCE ALGORITHM
Experiment 4

Ayush Jain 60004200132 B3

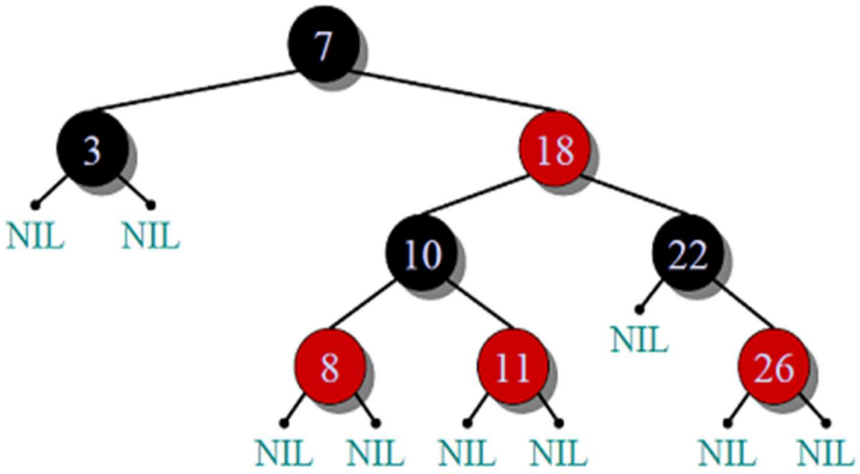
Aim : To implement Red Black Tree Insertion.

Theory:

Red–black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the colour (red or black) of the node. These colour bits are used to ensure the tree remains approximately balanced during insertions and deletions.
We will explore the insertion operation on a Red Black tree in the session. Inserting a value in Red Black tree takes O (log N) time complexity and O(N) space complexity.

In addition to the requirements imposed on a binary search tree the following must be satisfied by a red–black tree:

- Each node is either red or black.
- The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
- All leaves (NIL) are black.
- If a node is red, then both its children are black.
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.



The figure depicts the basic structure of Red Black Tree.

Algorithm

Basic operations associated with Red Black Tree:

To add an element to a Red Black Tree, we must follow this algorithm:

1) Check whether tree is Empty.

2) If tree is Empty then insert the newNode as Root node with color Black and exit from the operation.

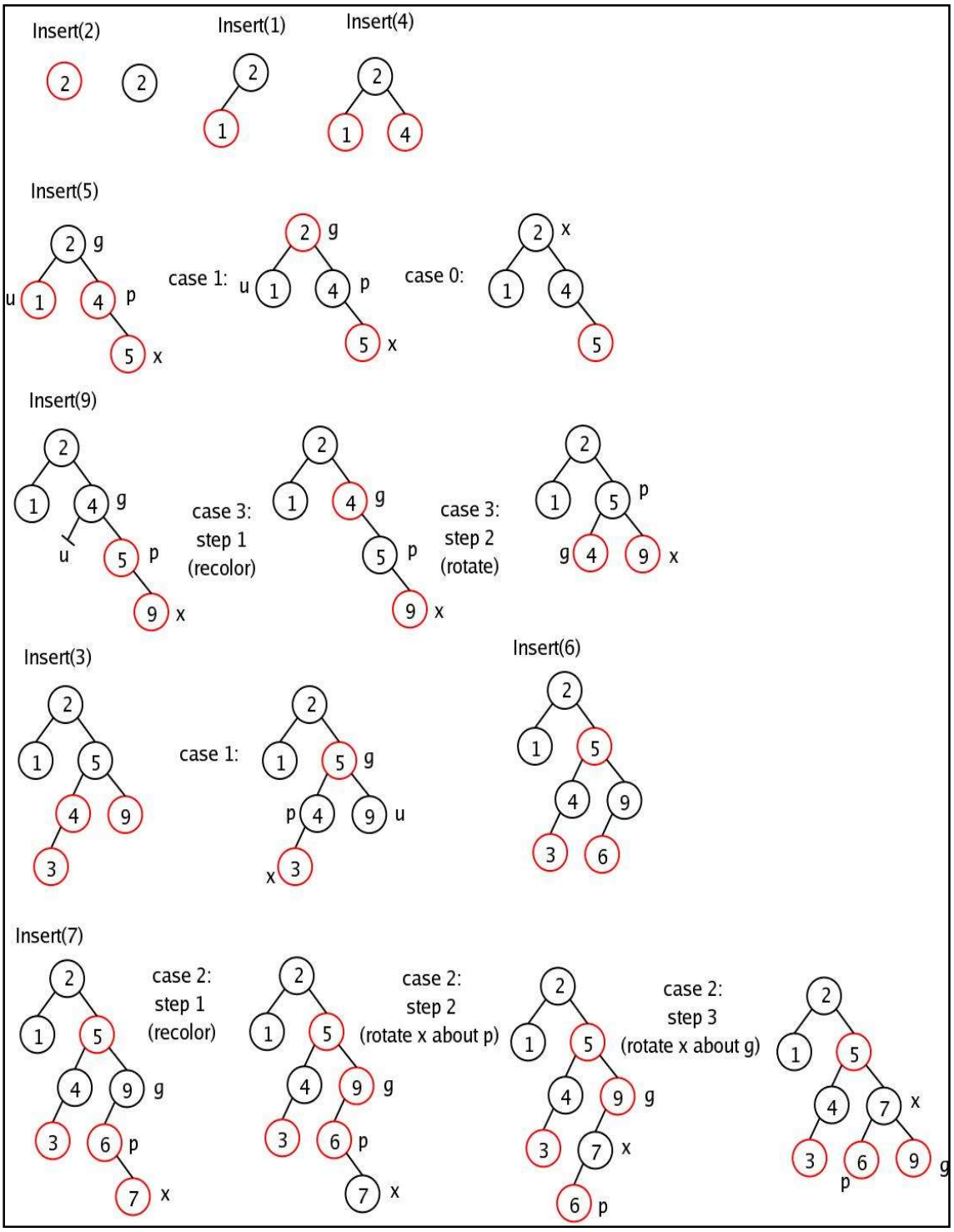
3) If tree is not Empty then insert the newNode as a leaf node with Red color.

4) If the parent of newNode is Black then exit from the operation.

5) If the parent of newNode is Red then check the color of parent node's sibling of newNode.

6) If it is Black or NULL node then make a suitable Rotation and Recolor it.

7) If it is Red colored node then perform Recolor and Recheck it. Repeat the same until tree becomes Red Black Tree.


Insert(2)
Insert(1)
Insert(4)
Insert(5)
case 1:
case 0:
Insert(9)
case 3:
step 1
(recolor)
case 3:
step 2
(rotate)
Insert(3)
case 1:
Insert(6)
Insert(7)
case 2:
step 1
(recolor)
case 2:
step 2
(rotate x about p)
case 2:
step 3
(rotate x about g)

Code:

```
import sys

# Node creation
class Node():
    def __init__(self, item):
        self.item = item
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1


class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL




    # Balance the tree after insertion
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
                if u.color == 1:
                    print("Case 1: Uncle is Red => performing only
Recoloring:")
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    print("Case 3: Uncle is black => perform rotation followed
by recoloring")
                    if k == k.parent.left:
                        print("Rotating Right")
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    print("Rotating left")
```

```
                    self.left_rotate(k.parent.parent)

            else:
                u = k.parent.parent.right

                if u.color == 1:
                    print("Case 1: Uncle is Red => perform only Recoloring")
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent

                else:
                    print("Case 3: Uncle is Black => peroform rotation
followed by recoloring:")
                    if k == k.parent.right:
                        k = k.parent
                        print("Rotating Left")
                        self.left_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    print("Rotating Right")
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0

    # Printing the tree
    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
            if last:
                sys.stdout.write("R---> ")
                indent += "     "
            else:
                sys.stdout.write("L---> ")
                indent += "|    "

            s_color = "RED" if node.color == 1 else "BLACK"
            print(str(node.item) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)


    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
```

```
            y.left.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
            y.right.parent = x

        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1

        y = None
        x = self.root

        while x != self.TNULL:
            y = x
            if node.item < x.item:
                x = x.left
            else:
                x = x.right

        node.parent = y
        if y == None:
```

```
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node

        if node.parent == None:
            node.color = 0
            return

        if node.parent.parent == None:
            return

        self.fix_insert(node)

    def get_root(self):
        return self.root

    def print_tree(self):
        self.__print_helper(self.root, "", True)


if __name__ == "__main__":
    bst = RedBlackTree()
    n = int(input("Enter Number of nodes: "))
    for i in range(n):
        value  =  int(input("\nEnter value of Node {} : ".format(i+1)))
        bst.insert(value)

    bst.print_tree()

    while(True):
        print("Menu:\n1.Add new Node\n2.Exit:")
        option = int(input("Enter Option: "))
        if option == 1:
            add_new = int(input("Enter new Node: "))
            bst.insert(add_new)
            bst.print_tree()
        if option == 2:
            print("Exit")
            break
```

Output:

```
Enter Number of nodes: 5

Enter value of Node 1 : 1

Enter value of Node 2 : 2

Enter value of Node 3 : 3
Case 3: Uncle is black => perform rotation followed by recoloring
Rotating left

Enter value of Node 4 : 4
Case 1: Uncle is Red => performing only Recoloring:

Enter value of Node 5 : 5
Case 3: Uncle is black => perform rotation followed by recoloring
Rotating left
R---> 2(BLACK)
     L---> 1(BLACK)
     R---> 4(BLACK)
          L---> 3(RED)
          R---> 5(RED)
Menu:
1.Add new Node
2.Exit:
Enter Option: 2
Exit
```

Conclusion: In conclusion, we have studied how to perform insertion in Red Black Tree.

ADVANCE ALGORITHM
Experiment 5

Ayush Jain 60004200132 B3

Aim : To implement Red Black Tree Deletion.

Theory:

Red-Black Trees are a type of self-balancing binary search tree, where each node in the tree is either red or black, and it satisfies certain properties to ensure that the tree remains balanced. When a node is deleted from a Red-Black Tree, the tree may become unbalanced, and it needs to be rebalanced to maintain the properties of a Red-Black Tree. Here are the different cases and how to resolve them in Red-Black Tree deletion:

Case 1: The node to be deleted has no children (it is a leaf node).

Solution: Simply delete the node and update its parent's child pointer to NULL.

Case 2: The node to be deleted has one child.

Solution: Replace the node with its child, and color the child black.

Case 3: The node to be deleted has two children.

Solution: Find the node's successor (the node with the smallest key in its right subtree), and replace the node to be deleted with its successor. Then delete the successor node using Case 1 or Case 2. Note that the successor node cannot have a left child, since it is the smallest key in the right subtree. Case 4: The node to be deleted is red, and has two black children.

Solution: Simply delete the node and adjust the colors of its parent and sibling as follows:

If the node's sibling is black, and has at least one red child, rotate the tree to make the sibling the new parent of its red child, and recolor the nodes as necessary.

If the node's sibling is black, and has two black children, color the sibling red, and repeat the same steps on the parent node.

If the node's sibling is red, rotate the tree to make the sibling the new parent of its red child, and recolor the nodes as necessary. Then repeat the steps for Case 4.

Case 5: The node to be deleted is black, and has one red child and one black child.

Solution: Replace the node with its red child, and color the child black.

Case 6: The node to be deleted is black, and has two black children.

Solution: Rebalance the tree using a series of rotations and recolorings as follows:

If the node is the root of the tree, simply delete it.

Otherwise, let sibling be the node's sibling (the other child of its parent), and let parent be the node's parent. Then:

If sibling is red, rotate the tree to make the parent the new parent of sibling, recolor sibling black, and repeat the steps for Case 4 on the original node.

If sibling is black, and has at least one red child, rotate the tree to make sibling the new parent of its red child, recolor the nodes as necessary, and repeat the steps for Case 4 on the original node.

If sibling is black, and has two black children, color sibling red, set node to its parent, and repeat the steps from the beginning.

Code:

```
import java.util.*; class Node {   int data;   Node parent;
  Node left;
Node right;   int
color;
}

public class Main{   private
Node root;   private Node
TNULL;

  // Preorder   private void preOrderHelper(Node
node) {     if (node != TNULL) {
System.out.print(node.data + " ");
preOrderHelper(node.left);
preOrderHelper(node.right);
    }
  }

  // Inorder   private void inOrderHelper(Node
node) {     if (node != TNULL) {
inOrderHelper(node.left);
System.out.print(node.data + " ");
inOrderHelper(node.right);
```

```
  }
 }

 // Post order   private void
postOrderHelper(Node node) {     if (node !=
TNULL) {       postOrderHelper(node.left);
postOrderHelper(node.right);
System.out.print(node.data + " ");
  }
 }

 // Search the tree   private Node searchTreeHelper(Node
node, int key) {     if (node == TNULL || key == node.data)
{       return node;
  }

  if (key < node.data) {       return
searchTreeHelper(node.left, key);
  }
  return searchTreeHelper(node.right, key);
 }

 // Balance the tree after deletion of a node
private void fixDelete(Node x) {     Node s;
while (x != root && x.color == 0) {       if (x
== x.parent.left) {         s = x.parent.right;
if (s.color == 1) {           s.color = 0;
        x.parent.color = 1;
leftRotate(x.parent);           s =
x.parent.right;
      }

      if (s.left.color == 0 && s.right.color == 0) {
        s.color = 1;           x =
x.parent;         } else {           if
```

```
(s.right.color == 0) {
s.left.color = 0;
          s.color = 1;
rightRotate(s);            s =
x.parent.right;
        }

        s.color = x.parent.color;
        x.parent.color = 0;
        s.right.color = 0;
leftRotate(x.parent);          x
= root;        }
    } else {        s =
x.parent.left;        if
(s.color == 1) {
s.color = 0;
        x.parent.color = 1;
rightRotate(x.parent);          s =
x.parent.left;
       }

      if (s.right.color == 0 && s.right.color == 0) {
s.color = 1;          x = x.parent;        } else {
        if (s.left.color == 0) {
s.right.color = 0;
          s.color = 1;
leftRotate(s);            s =
x.parent.left;
        }

        s.color = x.parent.color;
        x.parent.color = 0;
        s.left.color = 0;
rightRotate(x.parent);          x =
root;
```

```
      }
     }
    }
    x.color = 0;
  }

  private void rbTransplant(Node u, Node v) {
if (u.parent == null) {       root = v;
    } else if (u == u.parent.left) {
u.parent.left = v;
    } else {
      u.parent.right = v;
    }
    v.parent = u.parent;
  }

  private void deleteNodeHelper(Node node, int key) {
    Node z = TNULL;
    Node x, y;     while (node !=
TNULL) {       if (node.data ==
key) {         z = node;
      }

      if (node.data <= key) {
node = node.right;       } else
{         node = node.left;
      }
    }

    if (z == TNULL) {
      System.out.println("Couldn't find key in the tree");       return;
    }

    y = z;     int yOriginalColor =
y.color;     if (z.left == TNULL) {
x = z.right;       rbTransplant(z,
z.right);     } else if (z.right ==
```

```
TNULL) {       x = z.left;
rbTransplant(z, z.left);
    } else {
      y = minimum(z.right);
yOriginalColor = y.color;       x
= y.right;       if (y.parent == z)
{         x.parent = y;       } else {
rbTransplant(y, y.right);
y.right = z.right;
        y.right.parent = y;
      }

      rbTransplant(z, y);
y.left = z.left;
      y.left.parent = y;
      y.color = z.color;     }
    if (yOriginalColor == 0) {
fixDelete(x);
    }
  }

  // Balance the node after insertion   private
void fixInsert(Node k) {     Node u;     while
(k.parent.color == 1) {       if (k.parent ==
k.parent.parent.right) {         u =
k.parent.parent.left;         if (u.color == 1) {
u.color = 0;
          k.parent.color = 0;
          k.parent.parent.color = 1;
k = k.parent.parent;         } else {
if (k == k.parent.left) {             k =
k.parent;             rightRotate(k);
          }
          k.parent.color = 0;
```

```
        k.parent.parent.color = 1;
leftRotate(k.parent.parent);
      }      } else {        u =
k.parent.parent.right;

      if (u.color == 1) {
u.color = 0;
        k.parent.color = 0;
        k.parent.parent.color = 1;
k = k.parent.parent;        } else {
if (k == k.parent.right) {            k
= k.parent;            leftRotate(k);
        }
        k.parent.color = 0;
        k.parent.parent.color = 1;
rightRotate(k.parent.parent);
      }
    }      if (k ==
root) {        break;
    }    }
root.color = 0;
  }

  private void printHelper(Node root, String indent, boolean last) {    if
(root != TNULL) {      System.out.print(indent);      if (last) {
      System.out.print("R----");
indent += "   ";
    } else {
      System.out.print("L----");
indent += "|  ";
    }
    String sColor = root.color == 1 ? "RED" : "BLACK";
System.out.println(root.data + "(" + sColor + ")");
```

```
printHelper(root.left, indent, false);       printHelper(root.right,
indent, true);
    }
  }

  public Main() {     TNULL
= new Node();
    TNULL.color = 0;
    TNULL.left = null;
TNULL.right = null;     root =
TNULL;
  }

  public void preorder() {
preOrderHelper(this.root);
  }

  public void inorder() {
inOrderHelper(this.root);
  }

  public void postorder() {
postOrderHelper(this.root);
  }

  public Node searchTree(int k) {     return
searchTreeHelper(this.root, k);
  }

  public Node minimum(Node node) {
while (node.left != TNULL) {       node =
node.left;
    }     return
node;
  }
```

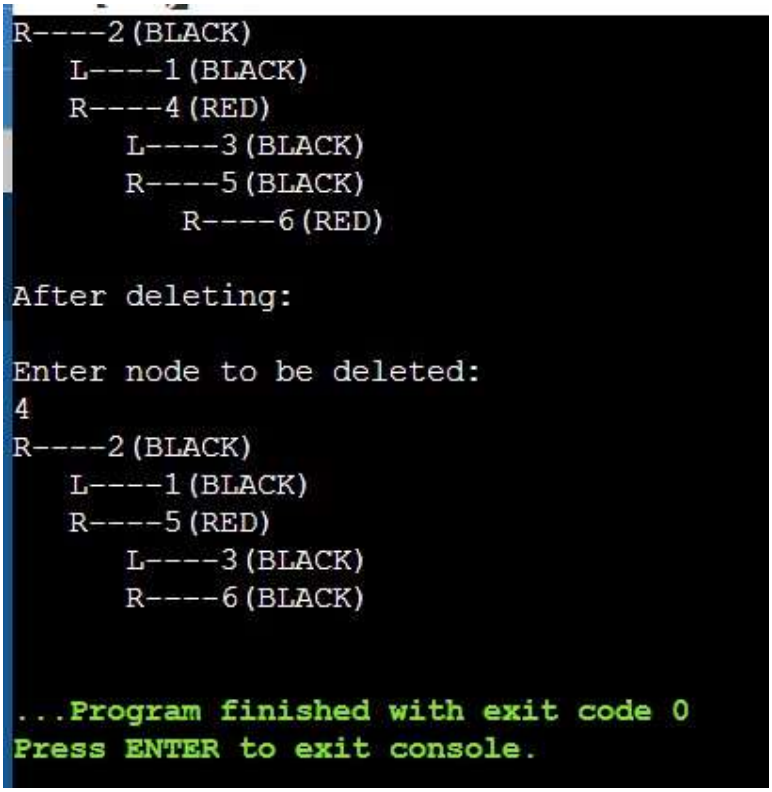
```
  public Node maximum(Node node) {
while (node.right != TNULL) {       node =
node.right;
   }     return
node;
  }
  public Node successor(Node x) {
if (x.right != TNULL) {       return
minimum(x.right);
   }

   Node y = x.parent;     while (y != TNULL
&& x == y.right) {
     x = y;       y =
y.parent;     }
return y;
  }
  public Node predecessor(Node x) {
if (x.left != TNULL) {       return
maximum(x.left);
   }
   Node y = x.parent;     while (y != TNULL
&& x == y.left) {       x = y;       y =
y.parent;
   }

   return y;
  }
  public void leftRotate(Node x) {
Node y = x.right;     x.right = y.left;
if (y.left != TNULL) {
y.left.parent = x;
```

```
    }
    y.parent = x.parent;     if
(x.parent == null) {       this.root =
y;     } else if (x == x.parent.left) {
x.parent.left = y;
    } else {
      x.parent.right = y;
    }
    y.left = x;
    x.parent = y;
  }

  public void rightRotate(Node x) {
Node y = x.left;     x.left = y.right;     if
(y.right != TNULL) {
y.right.parent = x;
    }
    y.parent = x.parent;
    if (x.parent == null) {
this.root = y;     } else if (x ==
x.parent.right) {       x.parent.right =
y;
    } else {
      x.parent.left = y;
    }
    y.right = x;
    x.parent = y;
  }

  public void insert(int key) {
Node node = new Node();
node.parent = null;     node.data
= key;     node.left = TNULL;
node.right = TNULL;
node.color = 1;

    Node y = null;
    Node x = this.root;
```

```
    while (x != TNULL) {
      y = x;       if (node.data <
x.data) {         x = x.left;       }
else {         x = x.right;
      }
    }

    node.parent = y;     if (y == null)
{       root = node;     } else if
(node.data < y.data) {       y.left =
node;
    } else {
      y.right = node;
    }

    if (node.parent == null) {
node.color = 0;       return;
    }

    if (node.parent.parent == null) {
return;
    }

    fixInsert(node);
  }

  public Node getRoot() {     return
this.root;
  }

  public void deleteNode(int data) {
deleteNodeHelper(this.root, data);   }

  public void printTree() {
printHelper(this.root, "", true);
  }
```

```
  public static void main(String[] args) {
Main bst = new Main();     bst.insert(1);
bst.insert(2);     bst.insert(3);
bst.insert(4);     bst.insert(5);
bst.insert(6);     bst.printTree();
    System.out.println("\nAfter deleting:");
    System.out.println("\nEnter node to be deleted:");
Scanner sc=new Scanner(System.in);     int
toBeDeleted=sc.nextInt();
bst.deleteNode(toBeDeleted);     bst.printTree();
  }
}
```

Output:



Conclusion: In conclusion, we have studied how to perform deletion in Red Black Tree.

ADVANCE ALGORITHM
Experiment 6

Ayush Jain 60004200132 B3

Aim : To implement KD Tree.

Theory:

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space. A non-leaf node in K-D tree divides the space into two parts, called as half-spaces. Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed. For the sake of simplicity, let us understand a 2-D Tree with an example. The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Code:

```
importjava.util.Arrays;


public class KDTree {  private static final int K = 2; // Number
   of dimensions  private Node root;


    private static class Node {
        private double[] point; // Data point
       private Node left; // Left child  private
       Node right; // Right child


        public Node(double[] point) {
```

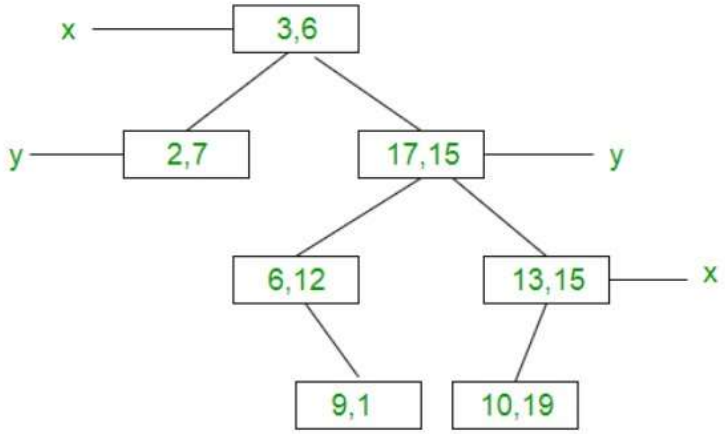
```
            this.point = point;
            this.left = null;
            this.right = null;
        }
    }

    public void insert(double[] point) {
        root = insert(root, point, 0);
    }
    private Node insert(Node node, double[] point, int depth) {
        if (node == null) {
            return new Node(point);
        }

        int axis = depth % K;  if (point[axis]
       < node.point[axis]) {
            node.left = insert(node.left, point, depth + 1);
        } else {  node.right = insert(node.right, point, depth +
           1);
        }
        return node;
    }

    public void printTree() {
    printTree(root, 0, "R"); // Start with root node labeled as "R"
}
```

```
private void printTree(Node node, int depth, String label) {
    if (node == null) {
        return;
    } for (int i = 0; i < depth; i++) {

         System.out.print("       "); // Indentation for tree-like format
    }
              System.out.println(label + "--->" + Arrays.toString(node.point));

    printTree(node.left, depth + 1, "L");  printTree(node.right,
   depth + 1, "R");
}

    public static void main(String[] args) {
        KDTree kdTree = new KDTree();
        double[] point1 = {3.0, 6.0};  double[]
        point2 = {17.0, 15.0};  double[] point3
        = {13.0, 15.0};  double[] point4 = {6.0,
        12.0};  double[] point5 = {9.0, 1.0};
        double[] point6 = {2.0, 7.0};  double[]
        point7 = {10.0, 19.0};
        kdTree.insert(point1);
        kdTree.insert(point2);
        kdTree.insert(point3);
        kdTree.insert(point4);
        kdTree.insert(point5);
```

```
        kdTree.insert(point6);
        kdTree.insert(point7);


         System.out.println("\nKD Tree :\n");
         // Print the k-d tree in a tree-like format  kdTree.printTree();
    }
}
```

Output:

```
KD Tree :

R--->[3.0, 6.0]
      L--->[2.0, 7.0]
      R--->[17.0, 15.0]
            L--->[6.0, 12.0]
                  R--->[9.0, 1.0]
            R--->[13.0, 15.0]
                  L--->[10.0, 19.0]
```



Conclusion: We have successfully implemented KD Tree

Advance Algorithm

Experiment 7

Ayush Jain 60004200132 B3

Aim : To implement Ford Fulkerson

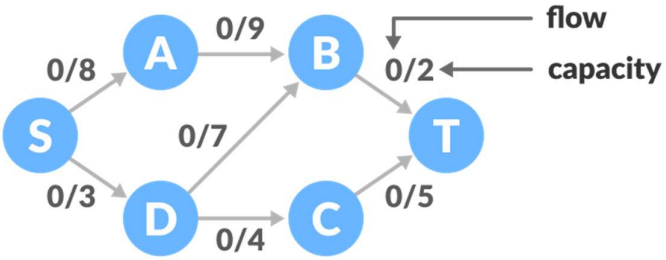
Theory:

Ford-Fulkerson Algorithm

Ford-Fulkerson algorithm is a greedy approach for calculating the maximum possible flow in a network or a graph.

A term, flow network, is used to describe a network of vertices and edges with a source (S) and a sink (T). Each vertex, except S and T, can receive and send an equal amount of stuff through it. S can only send and T can only receive stuff.

We can visualize the understanding of the algorithm using a flow of liquid inside a network of pipes of different capacities. Each pipe has a certain capacity of liquid it can transfer at an instance. For this algorithm, we are going to find how much liquid can be flowed from the source to the sink at an instance using the network.



Flow network graph

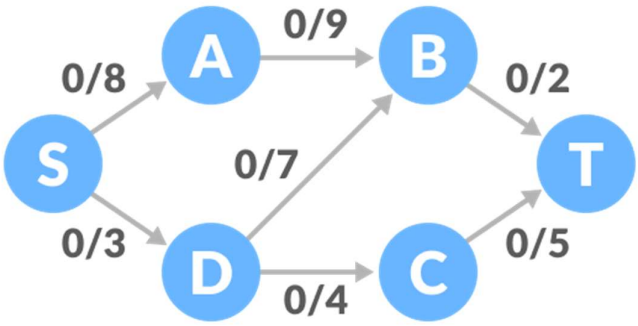
How Ford-Fulkerson Algorithm works?

The algorithm follows:

1. Initialize the flow in all the edges to 0.
2. While there is an augmenting path between the source and the sink, add this path to the flow.
3. Update the residual graph.
4. We can also consider reverse-path if required because if we do not consider them, we may never find a maximum flow.

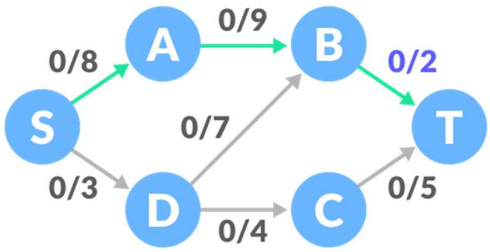
The above concepts can be understood with the example below.

Ford-Fulkerson Example
The flow of all the edges is 0 at the beginning.



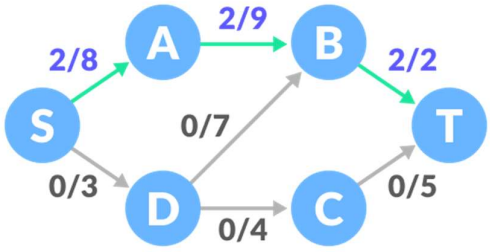
Flow network graph example

1. Select any arbitrary path from S to T. In this step, we have selected path S-A-B-T.



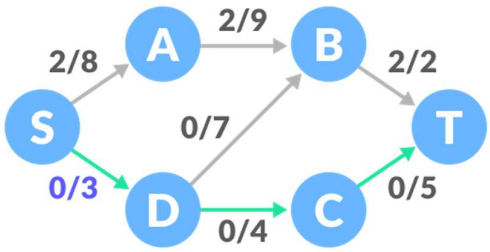
Find a path

The minimum capacity among the three edges is 2 (B-T). Based on this, update the flow/capacity for each path.



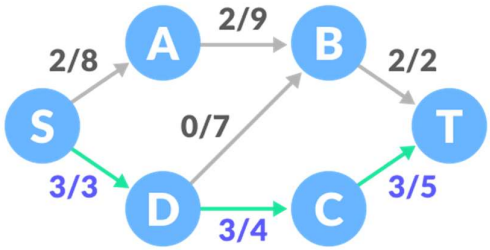
Update the capacities

2. Select another path S-D-C-T. The minimum capacity among these edges is 3 (S-D).



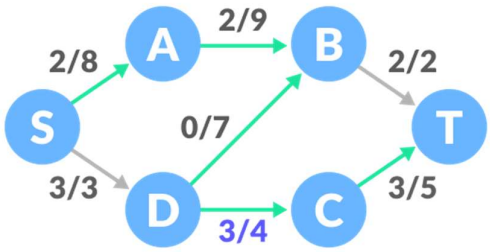
Find next path

Update the capacities according to this



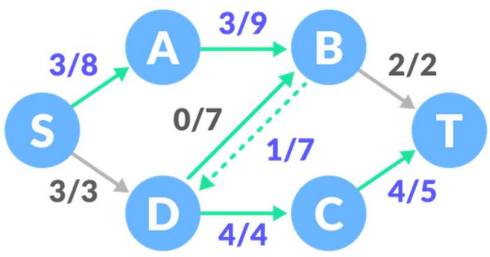
Update the capacities

3. Now, let us consider the reverse-path B-D as well. Selecting path S-A-B-D-C-T. The minimum residual capacity among the edges is 1 (D-C).



Find next path

Updating the capacities.



Update the capacities

The capacity for forward and reverse paths are considered separately.
4. Adding all the flows = 2 + 3 + 1 = 6, which is the maximum possible flow on the flow network.

Code:

```
from collections import defaultdict

class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.num_vertices = len(graph)

    def bfs(self, source, sink, parent):
        visited = [False] * self.num_vertices
        queue = [source]
        visited[source] = True

        while queue:
            u = queue.pop(0)
            for v, capacity in enumerate(self.graph[u]):
                if not visited[v] and capacity > 0:
                    queue.append(v)
                    visited[v] = True
                    parent[v] = u
                    if v == sink:
                        return True

        return False

    def find_max_flow(self, source, sink):
        parent = [-1] * self.num_vertices
        max_flow = 0

        while self.bfs(source, sink, parent):
            path_flow = float('inf')
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            max_flow += path_flow

            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
```

```
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow

graph = [[0, 8, 0, 0, 3, 0],
         [0, 0, 9, 0, 0, 0],
         [0, 0, 0, 0, 7, 2],
         [0, 0, 0, 0, 0, 5],
         [0, 0, 7, 4, 0, 0],
         [0, 0, 0, 0, 0, 0]]

g = Graph(graph)

source = 0
sink = 5

print("The maximum possible flow is %d " % g.find_max_flow(source, sink))
```

Output:

```
The maximum possible flow is 6
```

Conclusion: We successfully implemented Ford Fulkerson.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Engineering
Academic Year 2022-2023

Efficient Phone Book Management System Using Randomized BST

Advance Algorithm Laboratory

By

Jiya Patel **60004200105**
Ayush Jain **60004200132**
Dhruv Gandhi **60004200133**

Guide:

Prof. Chetashri Bhadane

Assistant Professor

Department of Computer Engineering
Academic Year 2022-2023

1. PROBLEM STATEMENT

The problem statement is to maintain the Phone Book Dictionary of a particular user. Phone Book dictionary is the one, which contain details of an individual along with their contact numbers. In Phone Book Dictionary names should be present in alphabetical order so that one can easily find the required person along with their telephone numbers. The phonebook application works specifically for tracking people. The Phonebook application contains a set of basic functions for adding, searching, updating, and deleting new contacts. This mini-C phonebook design allows you to perform simple tasks in your phonebook, such as mobile phones. You can add text to the phonebook, find, edit, search, and delete.

2. INTRODUCTION

a. Need

Phonebook project is a very simple tool that helps you understand the basic concepts of creation, file extensions and data structure. This software teaches you how to add, view, edit or modify, receive and delete data from files.

Adding new items, viewing them by logging in, editing and updating, searching for saved contacts and deleting data in the phonebook is one of the main features of the main phonebook application.

The phonebook application works specifically for tracking people. The Phonebook application contains a set of basic functions for adding, searching, updating, and deleting new contacts.

This application provides information on adding, viewing, modifying, receiving, and deleting data from/to files. Adding new entries, browsing them, editing and updating, searching for saved contacts, and deleting contacts in the phonebook is one of the most important services that become the main menu in the phonebook application.

b. Working

The following are the basic functionalities of the Phonebook Dictionary:

i. Main Menu:

When you start the project from any compiler or by double-clicking the executable.exe file, you'll see the Main Menu screen on window.

ii. Add Contact:

When you Choose to add contact, it opens a form which will be accepting the input from user in order to add new phonebook in the database.

iii. Delete Contact:

When you choose to delete contact its open a form which will be accepting the name of contact which you want to delete and the it will be deleting the phonebook information in the database.

iv. Search by Number:

When you choose to search a contact by number, system will be accepting the number you entered and searching throughout the database and will return the retrieved information and if not found it will be returning a message of Not Found.

v. Display Contacts in Ascending Order:

Its will be displaying the all the contacts in Phonebook Dictionary in Ascending Order.

vi. Display Contacts in Descending Order:

Its will be displaying the all the contacts in Phonebook Dictionary in Descending Order.

c. Applications

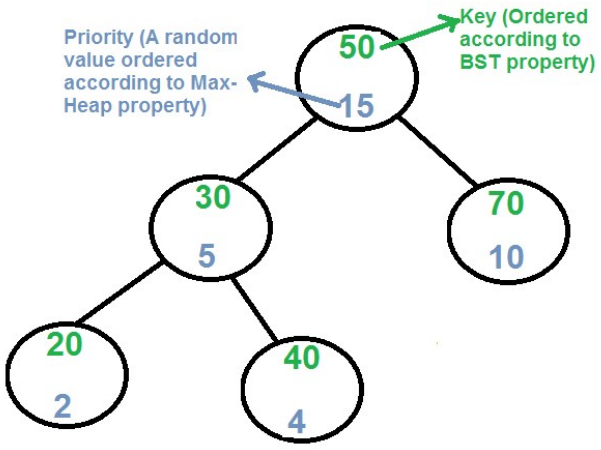
- The phonebook application works specifically for tracking people.
- Adding new records, listing them, modifying them and updating, search for contacts saved, and deleting the phonebook records are the basic functions which make up the main menu of this Phonebook application (as shown in the main menu screenshot below).
- Phonebook application can be used to maintain the users contacts.
- Phonebook records can be modified, listed, searched for and removed.
- And in searching it can be searched in both Ascending and Descending Order.

3. ADVANCED DATA STRUCTURE

a. Theory/Working

Treap (Randomized BST).:

Like Red-Black and AVL Trees, Treap is a Balanced Binary Search Tree, but not guaranteed to have height as O(Log n). The idea is to use Randomization and Binary Heap property to maintain balance with high probability. The expected time complexity of search, insert and delete is O(Log n).



- Every node of Treap maintains two values.
- Key Follows standard BST ordering (left is smaller and right is greater)
- Priority Randomly assigned value that follows Max-Heap property.

Basic Operation on Treap: Like other self-balancing Binary Search Trees, Treap uses rotations to maintain Max-Heap property during insertion and deletion.

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
     y                               x
    / \     Right Rotation          /  \
   x   T3   – – – – – – – >       T1   y
  / \       < - - - - - - -            / \
 T1  T2     Left Rotation            T2  T3
```

Keys in both of the above trees follow the following order:

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)

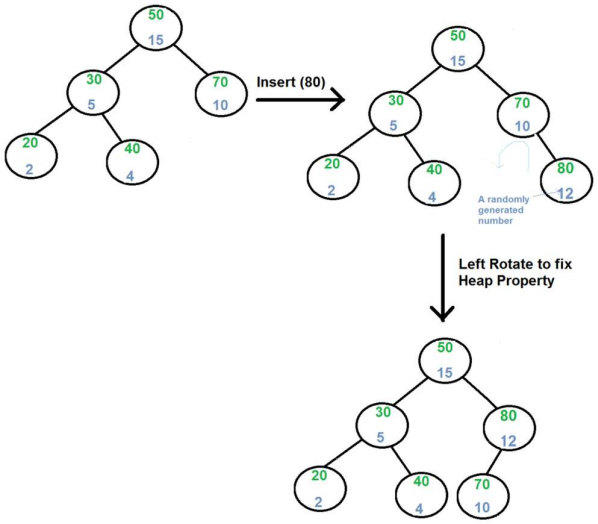
So BST property is not violated anywhere.

search(x):

Perform standard BST Search to find x.

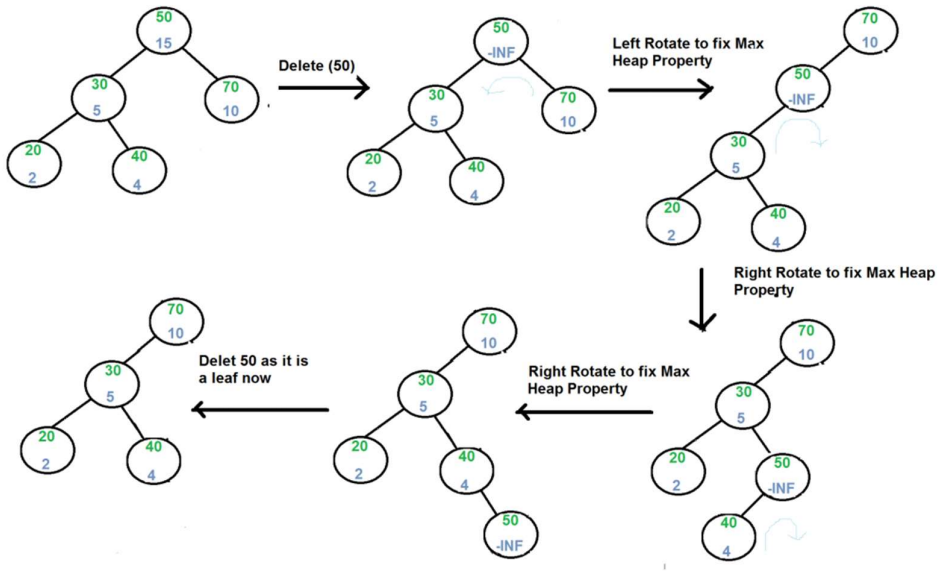
Insert(x):

- Create new node with key equals to x and value equals to a random value.
- Perform standard BST insert.
- Use rotations to make sure that inserted node's priority follows max heap property.



Delete(x):

1. If node to be deleted is a leaf, delete it.
2. Else replace node's priority with minus infinite (-INF), and do appropriate rotations to bring the node down to a leaf.



b. Applications

Randomized BSTs are used for a lot of applications due to its ordered structure.

- Randomized BSTs are used for indexing and multi-level indexing.
- They are also helpful to implement various searching algorithms.
- It is helpful in maintaining a sorted stream of data.
- TreeMap and TreeSet data structures are internally implemented using self-balancing BSTs.

c. Complexity Analysis

A treap provides the following operations:

i. **Insert (X, Y)** in O (log N).

Adds a new node to the tree. One possible variant is to pass only X and generate Y randomly inside the operation.

ii. **Search (X)** in O (log N).

Looks for a node with the specified key value X. The implementation is the same as for an ordinary binary search tree.

iii. **Erase (X)** in O (log N).

Looks for a node with the specified key value X and removes it from the tree.

iv. **Build (**X1**, ...,** XN**)** in O(N).

Builds a tree from a list of values. This can be done in linear time (assuming that X1..., XN are sorted).

v. **Union (**T1**,** T2**)** in O (M log(N/M)).

Merges two trees, assuming that all the elements are different. It is possible to achieve the same complexity if duplicate elements should be removed during merge.

vi. **Intersect (**T1**,** T2**)** in O (M log(N/M)).

Finds the intersection of two trees (i.e., their common elements). We will not consider the implementation of this operation here.

4. IMPLEMENTATION

a. Important screen shots

```
==================Phone Book Management System Using Treaps====================
--------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

--------------MENU--------------

--------------------------------
Enter the choice: 1

--------------------------------


--------------------------------
Enter Name:
Ayush
Enter Phone No.:
9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 3

---------------------------------


---------------------------------
Enter the Phone No.:
9137716225
1 Name :            Ayush, Phone No. 9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 4

---------------------------------


---------------------------------
Printing Contact in Ascending Order:
1 Name :            Ayush, Phone No. 9137716225
2 Name :            Dhruv, Phone No. 9867517870
3 Name :             Jiya, Phone No. 9137716224
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 5

---------------------------------


---------------------------------
Printing Contact in Descending Order:
1 Name :             Jiya, Phone No. 9137716224
2 Name :            Dhruv, Phone No. 9867517870
3 Name :            Ayush, Phone No. 9137716225
```

```
---------------MENU--------------
1. Add Contacts
2. Remove Contacts
3. Search by Phone No.
4. Shown in Ascending Order
5. Shown in Descending Order
6. Exit Program

---------------MENU--------------

---------------------------------
Enter the choice: 2

---------------------------------


---------------------------------
Enter Name to Delete:
Jiya
```

b. Code of important functions

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct node{
   char name[40];
   char phone_no[12];
   struct node*left;
   struct node*right;
};
struct node*head = NULL;
char temp_n[40];
char temp_p[12];

struct node* createNode(char*name, char*phone_no){
   struct node*new_node = (struct node*)malloc(sizeof(struct node));
   strcpy(new_node->name, name);
   strcpy(new_node->phone_no, phone_no);
   new_node->left = new_node->right = NULL;
   return new_node;
}

void insert(char*name, char*phone_no)
{
   struct node*root = head;
   struct node*prev = NULL;
   if(root == NULL)
   {
      struct node*new_node = createNode(name, phone_no);
      head = new_node;
      return;
```

```
    }

    while(root!=NULL)
    {
       prev = root;
       if(strcmp(root->name,name)==0)
       {
          printf("Name is already there!\n");
          return;
       }
       else if(strcmp(root->name, name)>0)
       {
          root = root->left;
       }
       else
       {
          root = root->right;
       }
    }
    struct node* new_node = createNode(name, phone_no);
    if(strcmp(prev->name, name)>0)
    {
       prev->left = new_node;

    }
    else
    {
       prev->right = new_node;
    }
}

int count_asc = 1;
void show_ascending(struct node*root)
```

```
{
   if(root!=NULL)
   {
      show_ascending(root->left);
      printf("%d Name : %15s, Phone No. %-10s \n",count_asc, root->name,
root->phone_no);
      count_asc++;
      show_ascending(root->right);
   }
}

int count_desc = 1;
void show_descending(struct node*root)
{
   if(root!=NULL)
   {
      show_descending(root->right);
      printf("%d Name : %15s, Phone No. %-10s \n",count_desc, root->name,
root->phone_no);
      count_desc++;
      show_descending(root->left);
   }
}

int found = 0;
int count_search = 1;
void search(struct node*root, char*phone_no)
{
   if(root!=NULL)
   {
      search(root->left, phone_no);
      if(strcmp(root->phone_no, phone_no) == 0)
      {
```

```
            printf("%d Name : %15s, Phone No. %-10s \n",count_search, root->name, root->phone_no);
            found = 1;
            count_search++;
        }
        search(root->right, phone_no);
    }
}

struct node*predecessor(struct node*root)
{
    while(root->right != NULL)
    {
        root = root->right;
    }
    return root;
}

struct node*delete(struct node*root, char*name)
{
    if(root == NULL)
    {
        return root;
    }

    if(strcmp(root->name, name)>0)
    {
        root->left = delete(root->left, name);
    }
    else if(strcmp(root->name, name)<0)
    {
        root->right = delete(root->right, name);
    }
```

```
    else
    {
       if(root->left == NULL)
       {
          struct node*temp = root->right;
          free(root);
          return temp;
       }
       else if(root->right == NULL)
       {
          struct node*temp = root->left;
          free(root);
          return temp;
       }

       struct node*pre = predecessor(root->left);
       strcpy(root->name, pre->name);
       root->left = delete(root->left, pre->name);

    }
    return root;
}

int main()
{
    printf("\n==================Phone Book Management System Using
Treaps==================");
    //
printf("================================================================
====================\n");
    menu:
    printf("\n---------------MENU--------------\n");
    printf("1. Add Contacts\n");
```

```
printf("2. Remove Contacts\n");
printf("3. Search by Phone No.\n");
printf("4. Shown in Ascending Order\n");
printf("5. Shown in Descending Order\n");
printf("6. Exit Program\n");
printf("\n---------------MENU--------------\n");
int choice;
printf("\n---------------------------------\n");
printf("Enter the choice: ");
scanf("%d", &choice);
printf("\n---------------------------------\n");
printf("\n");
if(choice == 1)
{
   printf("\n---------------------------------\n");
   printf("Enter Name: \n");
   while((getchar())!= '\n');
   scanf("%[^\n]%*c", temp_n);
   printf("Enter Phone No.: \n");
   scanf("%[^\n]%*c", temp_p);
   insert(temp_n,temp_p);
}
else if(choice == 2)
{
   printf("\n---------------------------------\n");
   printf("Enter Name to Delete: \n");
   while((getchar())!= '\n');
   scanf("%[^\n]%*c", temp_n);
   delete(head, temp_n);
}
else if(choice == 3)
{
   printf("\n---------------------------------\n");
```

```
    struct node*root = head;
    if(root == NULL)
    {
       printf("\n No Contacts are there to Search!");
       goto menu;
    }

    printf("Enter the Phone No.: \n");
    while((getchar())!= '\n');
    scanf("%[^\n]",temp_p);
    search(root, temp_p);
    if(found == 0)
    {
       printf("No Contacts Found!\n");
    }
    found = 0;
    count_search = 1;
    printf("\n");
}
else if(choice == 4)
{
    printf("\n---------------------------------\n");

    struct node*root = head;
    if(root == NULL)
    {
       printf("\n No Contacts are there to show!");
       goto menu;
    }
    printf("Printing Contact in Ascending Order: \n");
    show_ascending(root);
    printf("\n");
```

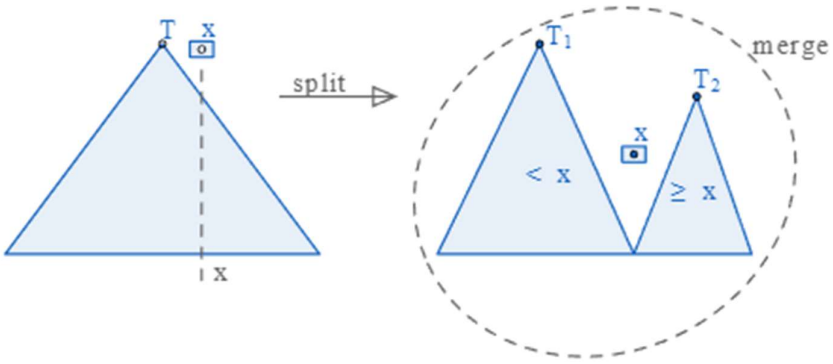
```
            count_asc = 1;
        }
        else if(choice == 5)
        {
            printf("\n---------------------------------\n");
            struct node*root = head;
            if(root == NULL)
            {
                printf("\n No Contacts are there to show!");
                goto menu;
            }
            printf("Printing Contact in Descending Order: \n");
            show_descending(root);
            printf("\n");
            count_desc = 1;
        }
        else if(choice == 6)
        {
            printf("\n-----------------------------------------------------------------\n");
            printf("\nContact Dictinary Signing off! Don't Forget to save
    Contacts!\n");
            printf("\n-----------------------------------------------------------------\n");
            exit(0);
        }
        else
        {
            printf("\n---------------------------------\n");
            printf("Enter the valid option! \n");
        }
        goto menu;

        return 0;
    }
```

5. COMPLEXITY ANALYSIS

Operations	Array and List	Binary Search Tree	Treap(Randomized BST).
Insertion	O(N)	O(N) or O(LOG N)	O(LOG N)
Deletion	O(N)	O(N) or O(LOG N)	O(LOG N)
Sorting	$O(N^2)$ or O(N LOG N)	O(LOG N)	O(LOG N)
Searching	O(N) or O(LOG N)	O(LOG N)	O(LOG N)

In terms of implementation, each node contains X, Y and pointers to the left (L) and right (R) children.

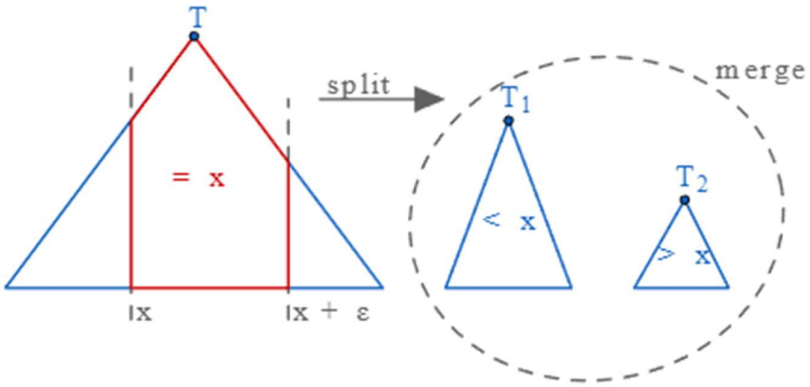
1. Insert



Implementation of **Insert (X, Y)** becomes obvious. First we descend in the tree (as in a regular binary search tree by X), and stop at the first node in which the priority value is less than Y. We have found the place where we will insert the new element. Next, we call **Split (T, X)** on the subtree starting at the found node, and use returned subtrees L and R as left and right children of the new node.

Alternatively, insert can be done by splitting the initial treap on X and doing 2 merges with the new node (see the picture).

2. Delete.



Implementation of **Erase (X)** is also clear. First we descend in the tree (as in a regular binary search tree by X), looking for the element we want to delete. Once the node is found, we call **Merge** on it children and put the return value of the operation in the place of the element we're deleting.

Alternatively, we can factor out the subtree holding X with 2 split operations and merge the remaining treaps (see the picture).

3. Searching:

For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of O(n). In general, time complexity is O(h) where h is height of Randomized BST.

In which h can be evaluated as the f (log n) in which n is the number of elements present in tree, It can be n in some cases if you get random priority in increasing order. But it's not every time possible in computer environment so Time complexity if Treap Tree which is Randomized Binary Search Tree.

6. CONLCUSION

The application software has been implemented successfully by using test cases. And the language used is C language. This Phonebook application is used to add, search, delete and some functions which is used to remember our contact details more easily. Application is implemented using Treap a Randomized Binary Search Tree. Compared to balanced BSTs that are not probabilistic (AVL trees), a treap is more simplistic and thus, quicker to code, since its structural property relies solely on the priority keys. A Treap can be used to quickly join and split sets of data. Given two sets of data, you can perform a split by merely inserting a dummy node with an infinite priority value. By doing this, the dummy ends up as the root of the tree, with its left and right children being the two sets of data. Treap Data Structure is a self-organizing data structure. They take after themselves and do not require supervision. Unlike other self-balancing trees, they do not require sophisticated algorithms (simple tree rotations will suffice, although simpler algorithms involving arrays can do the job too).

Name: Ayush Jain
SAP ID: 60004200132
Div: B
Computer Engineering

AA - Assignment 1

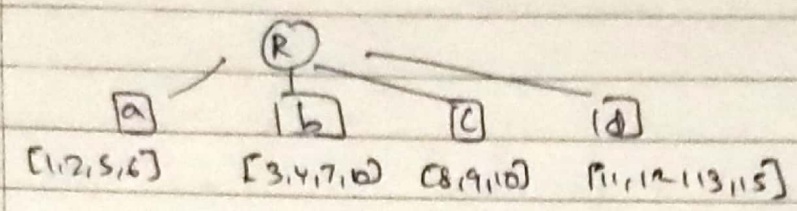
Q 1) Discuss need of R Tree and demonstrate its working?

→ 1) R trees is an advanced height balanced Tree Data Structure that is widely used in production for spatial problems.
2) Spatial data can be defined as any data that points to specific location on earth.
3) This data is stored in database and common operations for any database are storing data and performing all sort of queries among them.
4) Although there are many indexing based mechanism such as B-trees and ISAM index, they won't work well with multi-dimensional data.

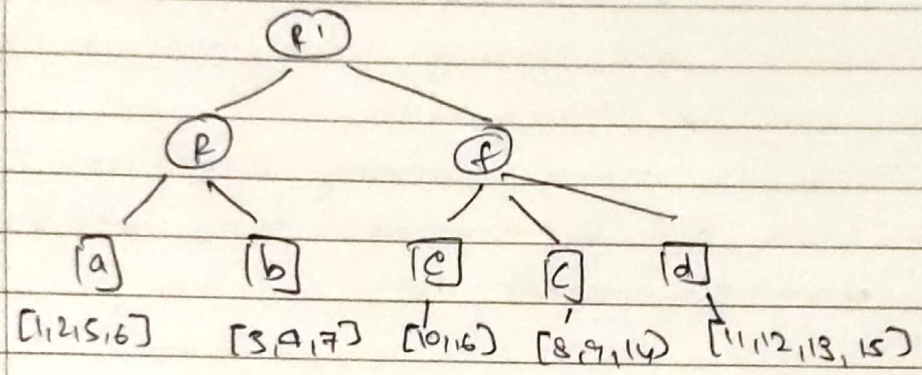
Working of R-Tree:

1) Insertion:
- Traverse the R-tree top-down, starting from root to each node.
- If there is a node where directory rectangle contains the D be inserted there search subtree.
- If more than one node satisfy this, choose the one with smallest area.
- Repeat with a leaf node is reached.
- If the leaf node is not full, an empty is inserted else
split the leaf node.
- Update the directory rectangles of ancestor nodes if necessary.

Insert object is , M=2 , m=1



Insert 16, M=2 , m=1



Q. 2) Explain Edmonds-Karp and max flow min cut Algorithm with suitable example?

→ 1) It is used to find max flow in a flow n/w and is an implementation of ford-fulkerson method.

2) Difference between Edmond-Karp and Ford-Fulkerson algo is that the selection of augmenting path is defined in Edmond Karp.

3) Edmond Karp algorithm:

Begin

Initialize flow s to 0

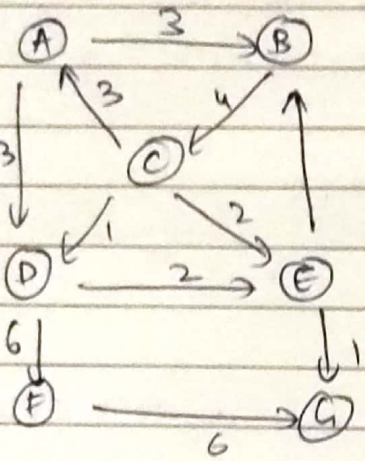
While there exists an augmenting path P0 the residual network as du

Choose the shortest augmenting path
Augment flow f along P
Return flow.

End.

9) Time complexity of Algorithm is $O(V(E^2))$

5) Example:



Step 1.

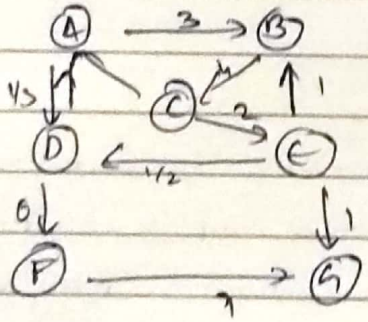
A B C D E F G

Parent map
B→A D→A
C→B E→D
F→D G→E

Visited Set
A B C G
E F C

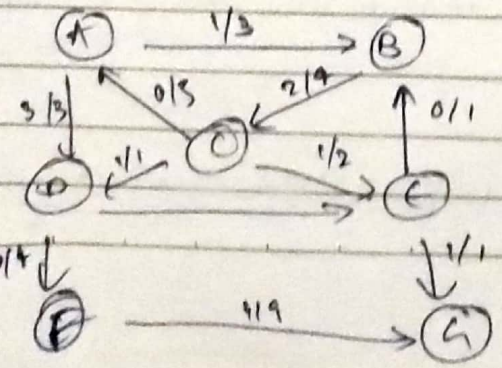
By BFS, Augmenting Path
$A \xrightarrow{3} D \xrightarrow{2} E \xrightarrow{1} G \Rightarrow 1$

Step 2:



Next BFS path would be,
$A \xrightarrow{2} D \xrightarrow{6} F \xrightarrow{1} G \Rightarrow 2$

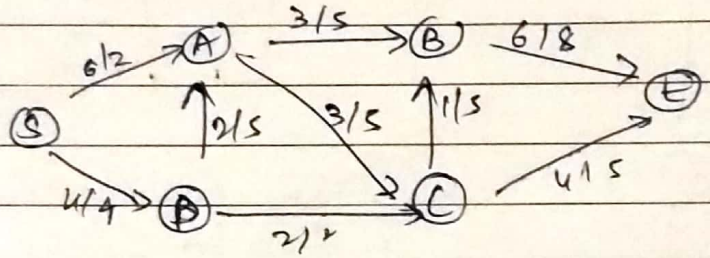
∴ Final Graph:



∴ max. flow = 1+2+1+1 = 5

Max flow Min Cut theorem

- A s-t cut is a portion of the vertices of flow network into 2-sets, such one set including source s and other one contains t. The capacity of s-t cut is defined as sum of capacities of from source to sink.
- The max flow is bounded by maximum cut capacity. Max flow min cut states that capacity of maximum flow has to be equal to the capacity of minimum cut.



In the above diagram, capacity of cut is $5+3+2=10$.

Q. 3) Explain Weighted Non-Bipartite matching with suitable examples.

→ 1) Weighted non-bipartite matching is a mathematical problem that includes fixed the optimal matching b/w two set of nodes (where the nodes may be non-bipartite i.e. there may be edges b/w nodes with the same set.

2) The goal of this problem is to assign weights to each edge and find the maximum weight matching b/w ~~two~~ set of node.

3) For example:

Consider a group of n people who need to be paired

up for a competition. Each person has a skill level, and the goal is to pair up people with complement skills to maximize the total skill level of the pairs. Let the set of people be denoted by A and their response skills levels by $w_1, w_2, w_3 \ldots w_n$. The weights of the edges between any two people can be calculated as the absolute difference in their skill levels of person i and person j are w_i and w_j. Now suppose that the people are divided into 2-groups - male and Female.

- This is the bi-partite matching problem. However if we want to allow pairing blw people of some gender, we need to solve a non-bipartite matching problem.

Name: Ayush Jain
SAP ID: 60009200132
Div: B
Computer Engineering

AA - Assignment 2

Q.1) Discuss the technique to find the closest pair of points?

→ • The Euclidian b/w two points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$ is $d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

• A brute force approach is to compute the distance b/w two random points and take minimum. However, running time is $\Theta(n^2) = O(n^2)$

• A high level description of a much better algorithm is given below:

• let Q be a set of planar points. $|Q| \leq 3$, then the distances b/w all pairs of points are computed and the closest pair is reported. If $|Q| > 3$, we use divide and conquer and continue the procedure.

• Each recursive call receives as i/p.

• Set $P \subseteq Q$

• Array X and Y containing points P sorted by x and y co-ordinates respectively.

Q.2) Explain Travelling Salesperson problem as an approximation problem.

→ i) Travelling Salesperson problem is a graph computational problem where the salesman needs to visit all cities in a list exactly once and the distances between all the cities are known. The goal is to find the shortest possible route in which the salesman visits all the

cities and return to the initial city.

2) The approximate sol. to the problem works only if the problem instances satisfy triangle inequality.

3) dist(i,j) ≤ dist(i,k) + dist(k1,k2) + ... dist(k,j)

When the cost function satisfies the triangle inequality, we can design an approx algo that returns a path whose cost is never more than twice the cost of an optimal flow.

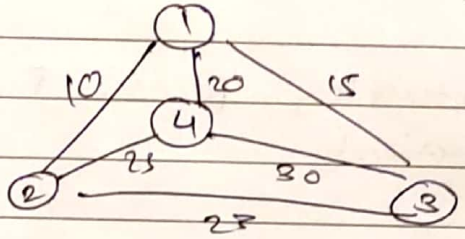
4) The algorithm is:

(1) Let i be the starting and ending point for salesman.

(2) construct MST from graph with 1 as root using prim's algo.

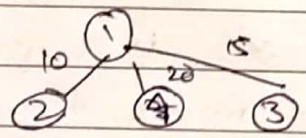
(3) list vertices visited in pre-order walk of the constructed MST and add 1 at the end.

Let us consider the foll. example:



pre-order traversal is
1 → 2 → 4 → 3 → 1

Minimum spanning tree:



The approximation algo may not always provide the optimal tour.

Q. 3) Write a short note on:

(1) Competitive Ratio:

→ • In the context of algorithm, the competitive ratio is a measure of the performance of an online algorithm compared to an optimal algorithm. Online algorithm are designed to work with incomplete information making decisions on the fly as new data becomes available.
• The ratio is designed as the worst case ratio of the cost of online algorithm (A) to the cost of the optimal offline algorithm.

Competitive ratio = max(A / OPT)

• The ratio is always greater than or equal to 1, as the running cost of A is greater or equal to the optimal cost.
• The ratio is a useful tool for comparing the performance of different online algorithms to the same problem. It can also provide insight into the difficulty of a problem and inherent limitations of online algorithms.

(2) K - Server:

→ • The K-Server problem is a classical problem in theoretical computer science that deals with the online management of K-server or a mutable space. The problem can be stated as:

• Given a set of n points in a metric space and k servers, the goal is to service a sequence of requests for the points, such that each request is received by one of the k servers, and the cost of servicing a request is proportional to the distance between the requested point and the server that is used to service it.

• The problem is an np-hard problem which means that there is no known algo that can solve the problem in polynomial time, unless P = NP.

• One of the widely used algo is the greedy problem which works as follows:

→ Given a request for a point P, the algo choises the server that is closest to P, and mooves the server to P.

→ If there are ties, the algo arbitiary choose the closest servers.

→ The greedy approach has a competitive ratio of k, which means that produced by the algo is at most k times a optimal cost.

→ Not the best but a good sol. considering the problem has. The k-server has important applications in computer network, distributed computing and logistics.