School of Engineering and Applied Science (SEAS), Ahmedabad University

Probability and Stochastic Processes (MAT 277)

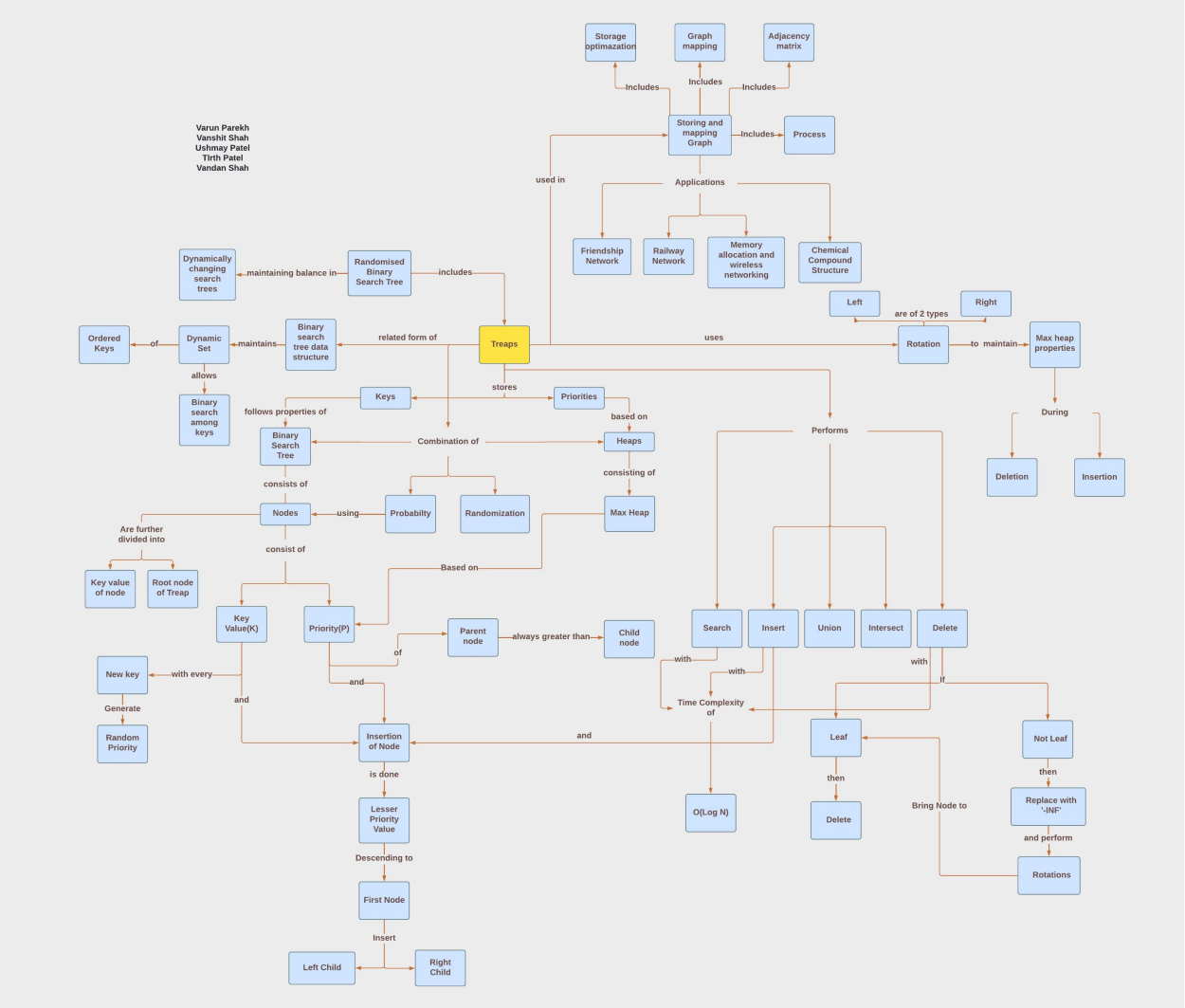
Special Assignment Report

Group Name: hn_a2_07_s1

Team Members:

- Varun Parekh (AU2040011)
- Vanshit Shah (AU2040098)
- Tirth Patel (AU2040020)
- Vandan Shah (AU2040196)
- Ushmay Patel(AU2040253)

I. Team Activity Learning and Concept Map



Activity Learning

As a team, we all worked together. We had consistency and teamwork amongst us which helped in our overall development. Initially the group members were more like speakers only that is they would try to keep their point, but now we all became good listeners as well. Everyone helped one another and we also learned many new latex instructions while encountering troubles. Overall, it was a pleasure to work with each one of them.

II. Background

Basically, treap is a combination of binary search tree and heap. It is a collection of nodes where each node contains a key and priority. Treaps satisfy two properties: binary search tree property and heap property. Binary search tree property states that the key of each parent node is higher than all the keys of the right subtree and is less than all the keys of the left subtree. Heap property states that the priority of the parent node is always maximal of its children. The priority of the tree is used for maintaining the height balance of the tree. It is used for storing data in sequence in the tree, which makes it useful for things like searching. It is also used as a self balancing binary tree and to solve connectivity problems. It is a self organizing algorithm. If we insert or delete an element it will do self adjusting by performing rotation without disturbing BST and heap property, there is a high probability that the tree will be balanced. They look after themselves, without any need to be managed. A treap will perform three major operations: insert, search and delete. Basically , a treap is a combination of two popular CS data structures: probability and randomization.

III. Motivation

WHY TREAPS?

Treap is a very useful data structure in computer science, as it is used to solve problems like connectivity problems, range query problems and can be used as an alternative to segment trees/ spare tables and splay trees in some cases as they are relatively easy to code

They easily allow implementations of split and join operations as well as some basic implementations like insert, delete and find.

They are the basis of randomized search trees, which have performance comparable to balanced search trees but are easy to implement.

IV. Algorithm Description

Treap Algorithm

A treap is a binary tree in which each node has a search key and a priority, with the inorder sequence of search keys sorted and each node's priority smaller than its children's priorities. 1 In other terms, a treap is a (max)heap for the priorities and a binary search tree for the search keys.

Treap Operations

Search: It takes O(log(v)) time to find key k, where v is the node with key(v) = k. Let v- be the inorder predecessor of k (the node whose key is just barely smaller than k) and v+ be the inorder successor of k for an unsuccessful search (the node whose key is just barely larger than k). Because the binary search's last node is either v- or v+, an unsuccessful search takes either O(log(v)).

```
1: function SEARCH(k, S)          ▷ Returns the node with the largest key smaller or equal to k
2:     v ← S.start
3:     while (v.next ≠ nil and v.next.val ≤ k) or (v.down ≠ nil) do
4:         if v.next ≠ nil and v.next.val ≤ k then
5:             v ← v.next
6:         else
7:             v ← v.down
8:     return v
```

Insert: Create a random priority y for x to insert a new key x into the tree. Create a new node at the leaf point where the binary search determines that a node for x should exist and perform a binary search for x in the tree. Perform a tree rotation that reverses the parent-child relationship between x and z, as long as x is not the root of the tree and has a higher priority number than its parent z.

```
1: procedure TREAP-INSERT(T, v)
2:     BST-INSERT(T, v)
3:     while v.priority < v.parent.priority do
4:         ROTATE(v)
```

Delete: To delete a node x from the treap, remove it if it is a leaf of the tree. Remove x from the tree and make z the child of x's parent if x has only one child, z. (or make z the root of the tree if x had no parent). Finally, if x has two children, in the sorted order, swap its place in the tree with that of its immediate successor z, resulting in one of the previous cases. In this situation, the swap may cause the heap-ordering property for z to be violated, requiring further rotations to restore it.

```
1: function TREAP-DELETE(v)
2:     TREAP-INCREASE-PRIORITY(v, ∞)
3:     DELETE(v)
```

Split: Once the insert function is implemented(insert(k,p)), The higher priority will be the root

of treap after insertion. Applying BST ordering property, the left subtree of the root will be treap with keys less than k, and the right subtree will be keys greater than k. This subtrees are the result of split.

```
1: function TREAP-SPLIT(T, key)
2:     v ← new node(key, −∞)
3:     TREAP-INSERT(T, v)
4:     T_< ← v.left
5:     T_> ← v.right
6:     DELETE(v)
7:     return (T_<, T_>)
```

Merge: Joining 2 subtrees X and Y where X is less than all keys in Y. Create a dummy node and make X as it's left child and Y as it's right child. Perform delete operation on dummy node.

```
1: function TREAP-MERGE(T_<, T_>)
2:     v ← new node(nil, nil)
3:     v.left ← T_<
4:     v.right ← T_>
5:     T ← v
6:     TREAP-DELETE(v)
7:     return T
```

V. Application

Application in CRL-EXPIRATION DATES

When data is entered into a database, it can be tagged with time values that indicate when it will expire, i.e., when it will be considered stale or invalid and hence no longer be considered part of the database. Expiration times are known in a variety of applications and can be assigned at the time of insertion.The techniques are based on fully functioning, persistent treaps, which are a hybrid of binary search trees and heaps with regard to a primary and secondary property are used. The primary attribute implements primary keys, whereas the secondary attribute keeps a priority queue of tuples to expire by storing expiry times in a minimal heap.

TO PREDICT EFFECTIVENESS OF DRUGS

In stage 3 clinical trials, more than half of medications fail, many due to a lack of understanding of the drug's mechanisms of action (MoA). Research and development will be greatly aided by a better understanding of medication MOA (RD). ProTINA and DeMAND, for example, are currently proposed algorithms that are perhaps overcomplicated. They also can't tell whether drug-induced gene expression or the structure of the networks employed to describe gene regulation has the most impact on accurate drug target inference. Betweenness, a network topology metric, can be utilized to improve drug target prediction. For target inference, TREAP combines betweenness values and modified p-values. TREAP is a modern technique to drug target prediction that is beneficial since it is computationally light, produces simple conclusions, and is often more accurate than current state-of-the-art approaches at predicting drug targets.

VI. Mathematical Analysis

1. Treap Join Analysis

Firstly check the priorities of the two roots and use whichever is greater as the new root.We assume that you are only given two trees and no middle element. If you do have a middle element, you can call this function twice, once with the left (or right) tree and the middle element and then with the result and the remaining tree.

Cost of Split $\propto$ Depth of the node.

Expected depth of node = $O(\log n)$

Expected Cost of Split = $O(\log n)$

For Join (T_1, m_1, T_2)

For T1:
It follows down the right Child and terminates, where right child is empty.

For T2:
It follows down the left Child and terminates, where left child is empty.

Hence:

The work is proportional to the sum of depth of the right-most and left-most keys.

The work of Join = Sum of Expected depth of nodes

$$= 0(\log |T|).$$

2. Treap Split Analysis

For the Split code for treaps, it does not look anything special. It can be Split code for any regular BST, because it doesn't do anything for re-balancing or checking priority. Again, we will check if it indeed implements the desired functionality and if the return values satisfy both the BST and Heap properties.

Therefore:

Time for splitting a tree into sizes of m and n is O(log Max $\{m, n\}$)

3. **More analysis of randomized search trees**.

   (a) Expected value of the node depth in RST is O(log N), and thus average time cost for successful search/find is O(log N).

   (b) Similar considerations show that unsuccessful delete, split find,insert and join in a RST all of them have an average time cost of O(log N)

   (c) It is also possible for a randomized search tree to be badly unbalanced, with height significantly worse than $log_2 N$, where N is the number of nodes in the treap; however, this is unlikely to happen.To actually be badly unbalanced, the random priorities have to be correlated in a certain way with key values, and with a good random number generator this will be unlikely to occur.

   (d) The expected time costs are like average time costs, averaged over many constructions of a treap with the same N keys, but different random priorities.

   (e) In practice, randomised search trees will very definitely deliver good performance, despite the fact that they cannot guarantee it.

4. **Delete in Treaps**

   As we know that while deleting we search for the node X containing K using the usual BST algorithm method. As the worst case includes rotations to delete, the deletion in treaps can be performed in time O(H), where H is the height of the treap.

   Or simply → O(Log N)

5. **Insert in Treaps**

   Since Insertion is just deletion in reverse that is the worst case would require AVL rotations that would insert the element in its proper position. We would again perform it in O(H) time, where H is the height of the treap.

   That is : O(log N)

   Since the depth of a node in a treap is $\theta(n)$ in the worst case, each of these operations has a worst-case running time of $\theta(n)$.

Now we will do the analysis for finding the **expected depth of the node.**

Suppose there are:

Nodes $\rightarrow \quad x_1, x_2, \ldots, x_N$

Keys $\rightarrow \quad k_1, k_2, \ldots, k_N$

Priorities $\rightarrow \quad p_1, p_2, \ldots, p_N$

x_i node has key k_i and priority p_i

Making 2 probabilistic assumptions.

(a) All the keys $k_1, \ldots, k_N$ are equally likely to be searched for.

(b) All the priorities are $p_1, \ldots, p_N$ are randomly uniformly generated independent of each other.

Also assuming, the keys are in sorted order ($k_i < k_{i+1}$)

(a) But the keys can be inserted in any order.

(b) All the priorities are distinct.

Expected node depth

Depth of node $x_i \rightarrow d(x_i)$

This means that $d(x_i)$ comparisions are required to find the key k_i

$P_r(p_1, \cdots, p_N) \rightarrow$ Probability of generating N priority values.

This will determine shape of the treap and also the location of every key in it. This helps us to determine depth of node x_i

$$E[d(x_i)] = \sum_{p_1, \cdots, p_n} \Pr(p_1, \ldots, p_N)\, d(x_i)$$

Let A_{ji} be the indicator function.

$A_{ji} = 1$, if x_j is the ancestor of x_i

$\therefore d(x_i) = \sum_{j=1}^{N} A_{ji}$ (As depth of node is equal to the number of ancestors).

$E[d(x_i)] = E[\sum_{j=1}^{N} A_{ji}]$

$= \sum_{j=1}^{N} E\left[A_{ji}\right]$

$\left[E\left[A_{ji}\right] = \Pr\left(A_{ji} = 1\right)\right] -$¿ Probability that node x_j is ancestor of x_i

($E[X] = Pr[X = 1]$ for any indicator random variable X)

$\therefore$ The probability that x_j is ancestor of x_i is just the probability that the random priority generated for x_m is higher then the other $|j - i|$ priorities generated for nodes with indexes between j and i inclusive.

But priorities are generated randomly and independently.
Remaining $|j - i| + 1$ nodes have equal probability of having the highest priority.

$$\therefore E\left[A_{ji}\right] = \frac{1}{|j - i| + 1}$$

$$\begin{aligned} E[d(x_i)] &= \sum_{j=1}^{N} \frac{1}{|j - i| + 1} \\ &= \sum_{j=1}^{i-1} \frac{1}{i - j + 1} + \sum_{j=i+1}^{n} \frac{1}{j - i + 1} \\ &= H_i - 1 + H_{n-i+1} - 1 \quad \left(H_n = \sum_{i=1}^{n} \frac{1}{n}\right) \\ &= \quad < 2 \ln n \\ &= \quad O(\log n) \end{aligned}$$

The expected time for search is proportional to the expected number of ancestors.

Therefore the search time complexity will be $O(logn)$.

Average number of comparisions needed to find the key.

This will be equal to expected node depth averaged over all nodes.

$$D_{\text{avg}}(N) = \sum_{i=1}^{N} \frac{1}{N} (E[d(x_i)]) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{1}{|j - i| + 1}$$

$$\begin{aligned} &= \frac{2}{N} \sum_{i=1}^{N} \frac{N - i + 1}{i} - N \\ &= \frac{2(N + 1)}{N} \sum_{i=1}^{N} \frac{1}{i} - 3N \\ &= \frac{2(N + 1)}{N} \sum_{i=1}^{N} \frac{1}{i} - 3 \end{aligned}$$

VII. Code(with description of each line)

Code 1: Treap code explaination

```
#import the random lib to generate random numbers
from random import randrange


# A Treap Node
class TreapNode:

    def __init__(self, data, priority=100, left=None, right=None):
        #Initializing data
        self.data = data
        #Initializing priority
        self.priority = randrange(priority)
        #Initializing left
        self.left = left
        #Initailizing right
        self.right = right

#recursive function to left rotate a given key.
def rotateLeft(root):
    #stores the right node of root in R
    R = root.right
    X = root.right.left
#stores the value of left node of R in X.

    # rotates structure and the initial root becomes left node of the R and X becomes
        right node of the initial root
    R.left = root
    root.right = X

    # here, R is set as a new root.
    return R

# recursive function to right rotate a given treap.
def rotateRight(root):
    # initialises the left node of root as L and Y as right node of Y.
    L = root.left
    Y = root.left.right

    # rotate the given treap and the initial root becomes right node of L and Y
        becomes left node of the initial root.
    L.right = root
    root.left = Y

    # L becomes   the new root of the treap
    return L


# Defining a recursive function to insert a given key with a priority into treap
def insertNode(root, data):

    #base case: if root is none
    if root is None:
  #return the data by calling function TreapNode
        return TreapNode(data)

    # if the given data is less than the root node
```

```
    if data < root.data:
  #insert in the left subtree
        root.left = insertNode(root.left, data)

        #if heap property is violated
        if root.left and root.left.priority > root.priority:
            #rotate right
            root = rotateRight(root)
    else:
        # If not left, insert in the right subtree
        root.right = insertNode(root.right, data)

        # rotate left if the heap property is violated
        if root.right and root.right.priority > root.priority:
            root = rotateLeft(root)
    #return the root
    return root


# Defining a recursive function to search for a key in a given treap
def searchNode(root, key):

    # if the key is not present in the tree
    if root is None:
        #return false if it is not there
        return False

    # if the key is found
    if root.data == key:
        #return true if it is there
        return True

    # if the key is less than the root node
    if key < root.data:
        #calling the searchNode function again and searching in left subtree
        return searchNode(root.left, key)

    # if the key is greater than the root node
    #calling the searchNode function again and searching in right subtree
    return searchNode(root.right, key)


#Defining a recursive function to delete a key from a given treap
def deleteNode(root, key):

    # base case: the key is not found in the tree
    if root is None:
  #return none
        return None

    # if the key is less than the root node
    if key < root.data:
      #recur the function again in left subtree
        root.left = deleteNode(root.left, key)

    # if the key is more than the root node
    elif key > root.data:
        #recur the function in right subtree
        root.right = deleteNode(root.right, key)

```

```
    # if the key is found
    else:

        # Case 1: node to be deleted has no children (it is a leaf node)
        if root.left is None and root.right is None:
            # deallocate the memory and update root to None
            root = None

        # Case 2: node to be deleted has two children
        elif root.left and root.right:
            # if the left child has less priority than the right child
            if root.left.priority < root.right.priority:
                # call ‘rotateLeft()‘ on the root
                root = rotateLeft(root)

                # recursively delete the left child
                root.left = deleteNode(root.left, key)
            else:
                # call ‘rotateRight()‘ on the root
                root = rotateRight(root)

                # recursively delete the right child
                root.right = deleteNode(root.right, key)

        # Case 3: node to be deleted has only one child
        else:
            # choose a child node
            child = root.left if (root.left) else root.right
            root = child
    #returning the root value
    return root


# Utility function to print two-dimensional view of a treap using
# reverse inorder traversal
def printTreap(root, space):
    #defining the height of the treap
    height = 10

    # Base case
    if root is None:
        #if root is none then return
        return

    # increase the distance between levels
    space += height

    # print the right child first
    printTreap(root.right, space)

    # print the current node after padding with spaces
    for i in range(height, space):
        print(' ', end='')
    #printing the data and priority of root
    print((root.data, root.priority))

    # print the left child
    printTreap(root.left, space)

def s_op():
```

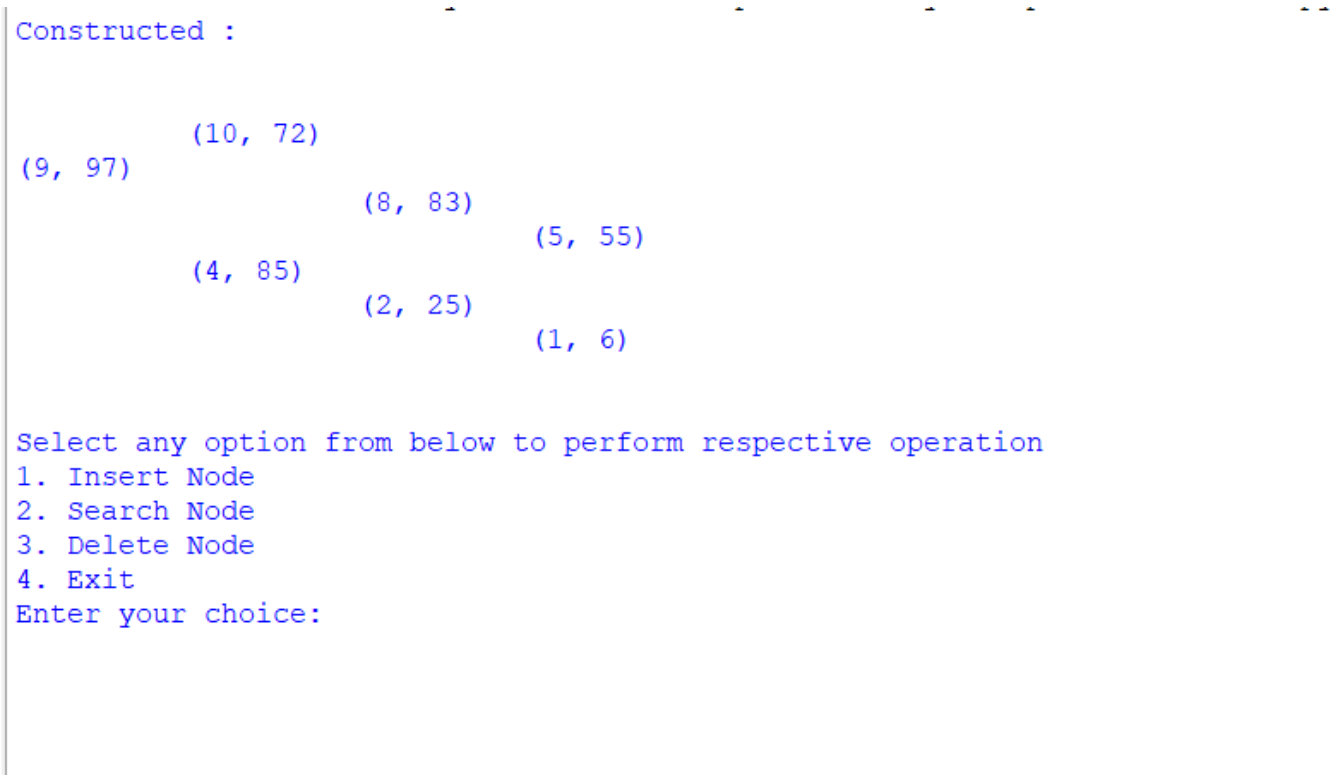
```
    # taking some random treap keys
    keys = [5, 2, 1, 4, 9, 8, 10]

    # constructing a treap, initial root is none
    root = None
    for key in keys:
        #inserting each keys in treap
        root = insertNode(root, key)
    #printing the constructed treap
    print("Constructed :\n\n")
    printTreap(root, 0)
    while(1):
        #giving option to user to select
        print("\n\nSelect any option from below to perform respective operation")
        print("1. Insert Node")
        print("2. Search Node")
        print("3. Delete Node")
        print("4. Exit")
        #taking input from user to perform operation
        select=int(input("Enter your choice: "))
        #if selected option is 1 then perform insert
        if select==1:
            #taking node value from user to insert into treap
            i1=int(input("\nInsert node: "))
            #inserting the node value given by user by calling insertNode function
            root = insertNode(root, i1)
            #printing the treap
            printTreap(root, 0)
        #if selected option is 2 then perform search
        elif select==2:
            #taking node value from user to search into treap
            s1=int(input("\nSearch node: "))
            #if the node is present then print true else false by calling searchNode
                function
            print(searchNode(root, s1))
        #if selected option is 3 then perform delete
        elif select==3:
            #taking node value from user to delete from treap
            d1=int(input("\nDelete node: "))
            #deleting the node value taken from user by calling deleteNode function
            root = deleteNode(root, d1)
            #printing the treap after deleting the node
            printTreap(root, 0)
        #if selected option is 4 then exit
        elif select==4:
            exit
            break
        #else not a valid choice go back again
        else:
            print("\nPlease enter valid choice\n")
#to check whether the current script is running own or being imported somewhere
if __name__ == '__main__':
    #calling the s_op function to construct and to perform operations
    s_op()
```

```
def split(root, data):
    if root is None:
        return None
    elif root.data is None:
        return None
    else:
        if data<root.data:
            left, root.left=split(root.left, data)
            return left, root
        else:
            root.right, right=split(root.right, data)
            return root, right
```

We couldn't find the code for splitting the treap but we tried it ourselves.We weren't able to execute it successfully. There might be some syntax problem which we will continue to study and try to get it executed properly.

VIII. Results and Inferences

Here, we have added the functionalities of insert, update and delete in a treap and we also tried to add the functionalities of splitting and merging treaps and worked on the code but couldn't achieve major success but we have explained that concept in the mathematical analysis part.



(a) Constructed treap

This is the constructed treap which follows BST property as well as the heap property. NOTE that "9" is the root with 97 as the random priority. 97 is the highest priority among all, which shows that priorities follow max heap property.

```
Select any option from below to perform respective operation
1. Insert Node
2. Search Node
3. Delete Node
4. Exit
Enter your choice: 1

Insert node: 3
          (10, 72)
(9, 97)
                    (8, 83)
                              (5, 55)
          (4, 85)
                    (3, 73)
                              (2, 25)
                                        (1, 6)
```

(b) insert operation

Here we are inserting node 3. Priority is random. The inserted node takes it place by performing AVL rotations. Treap is formed by following its properties.

```
Select any option from below to perform respective operation
1. Insert Node
2. Search Node
3. Delete Node
4. Exit
Enter your choice: 3

Delete node: 8
          (10, 72)
(9, 97)
                    (5, 55)
          (4, 85)
                    (3, 73)
                              (2, 25)
                                        (1, 6)
```

(c) delete operation

Delete operation removes the key from the treap. After deletion, AVL rotations are performed so that treap property does not get violated. If it is the leaf, it is directly deleted. If priority of right child is greater, perform left rotation at node. if priority of left child is greater, perform right rotation at node.

```
          (10, 72)
(9, 97)
                    (8, 83)
                              (5, 55)
          (4, 85)
                    (3, 73)
                              (2, 25)
                                        (1, 6)


Select any option from below to perform respective operation
1. Insert Node
2. Search Node
3. Delete Node
4. Exit
Enter your choice: 2

Search node: 8
True
```

(d) search operation

```
          (10, 72)
(9, 97)
                    (5, 55)
          (4, 85)
                    (3, 73)
                              (2, 25)
                                        (1, 6)


Select any option from below to perform respective operation
1. Insert Node
2. Search Node
3. Delete Node
4. Exit
Enter your choice: 2

Search node: 7
False
```

(e) search operation

Search operation returns true if the user input key is found in the treap, else returns false.

IX. References

[1] G. E. Blelloch and M. Reid-Miller, "Fast set operations using treaps," in *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pp. 16–26, 1998.

[2] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4, pp. 464–497, 1996.

[3] I. Treaps, "High probability bounds for treaps (and quicksort),"

[4] D. Golovin, "B-treaps: A uniquely represented alternative to b-trees," in *International Colloquium on Automata, Languages, and Programming*, pp. 487–499, Springer, 2009.