Advanced Data Structures and Algorithms (CS6013)

K.Naveen Kumar
CS19MTECH11009
Assignment-II
IIT Hyderabad

October 12, 2019

1 Treap Data Structure

Treap is a data structure which omits the property of Binary Search Tree and Heap data structures (Min Heap). In this assignment we are allowed to implement the following functions.

1.1 Insertion of a key into Treap

1.1.1 Explanation

We use same concept of inserting the key into binary search tree. We check our key with the root node, if the our key is less-than the root's key then we go through left sub-treap and if our key is greater than or equals to root's key then we move towards right sub-treap of the root. We do this process until we get correct location to insert the given key. After insertion, we check for min heap property based on priority of the node. We do rotations (left rotation or right rotation) if any node violates the min heap property.

Below mentioned algorithm is implemented to insert a key into Treap. Our INSERT_KEY() function takes tnode pointer and key value as arguements. This function also takes help of INSERT() function to insert new key into treap.This INSERT() function takes two tnode pointers as arguments.One pointer is pointing to the root node of the given treap and other pointer is pointing to the node to be inserted into the treap. We recursively inserted the key into treap based on BST insertion and we recursively check for min-heap property as well in INSERT() function. If any node is making the violation , we do rotation to overcome that violation. We recursively check for violations of the node and if any we do overcome using rotations recursively. At the end of algorithm we are returning the node pointer that points to newly inserted node.

This INSERT() algorithm uses rotations to overcome treap property violations. We have implemented LEFTROTATION and RIGHTROTATION functions to do this task. This two functions do rotations as for given root pointer and returns node pointer that points to new root node in the treap(After rotations root node may be changed in treap).

The LEFTROTATION() function takes one argument over which we perform left rotation. This function appropriately rotate the treap and takes care of sizes of nodes that goes under rotation. This function will return the pointer to the new root node after left rotations are done.

The RIGHTROTATION() function takes one argument over which we perform right rotation. This function appropriately rotate the treap and takes care of sizes of nodes that goes under rotation. This function will return the pointer to the new root node after right rotations are done.

1.1.2 pseudo-code of insert_key() Algorithm

Algorithm 1 Insertion of a key

```
1:  procedure INSERT_KEY(*&root,key)
2:      temp←Create New tnode()
3:      temp.key←key
4:      temp.priority←random()//Random priority
5:      temp.size←1
6:      Make temp node's right, left and parent link as NULL
7:      if root==NULL then root←temp
8:      else then
9:                  tnode * dup←root
10:                 root←INSERT(dup,temp)
11:     return temp
12: end procedure=0
```

Algorithm 2 Utility function INSERT()

```
1:  procedure INSERT(*root,*temp)
2:      ifroot==NULL then return temp
3:      if temp.key < root.key then
4:              root.left←INSERT(root.left,temp)
5:              root.left.parent←root
6:              root.size←root.size+1
7:              if root.left.priority<root.priority
8:              then root←RIGHTROTATION(root)
9:      else  then
10:             root.right←INSERT(root.right,temp)
11:             root.right.parent←root
12:             root.size←root.size+1
13:             ifroot.right.priority<root.priority
14:             then root←LEFTROTATION(root)
15:     return root
16: end procedure
```

Algorithm 3 utility function LEFT ROTATION

```
1:  procedure leftRotation(*root)
2:      if root==NULL return root
3:      tnode * y←root.right , *x←y.left
4:     tnode * p←root.parent
5:      y.left←root
6:      root.right←x
7:     y.parent←p
8:     root.parent←y
9:      if root.right! =NULL then root.right.parent←root
10:     xOwn←root.size−y.size
11:     yLeft←0
12:     if x! =NULL then yLeft←x.size
13:     root.size←xOwn+yLeft
14:     yRight←0
15:     if y.right! =NULL then yRight←y.right.size
16:     y.size←root.size+yRight+1
17:     return y
18: end procedure
```

Algorithm 4 utility function RIGHT ROTATION

```
1:  procedure rightRotation(*root)
2:      if root==NULL return root
3:      tnode * x←root.left , *y←x.right
4:     tnode * p←root.parent
5:      x.right←root
6:      root.left←y
7:     x.parent←p
8:     root.parent←x
9:      if root.left! =NULL then root.left.parent←root
10:     nRoot←root.size−x.size
11:     xRight←0
12:     if y! =NULL then xRight←y.size
13:     root.size←nRoot+xRight
14:     xLeft←0
15:     if x.left! =NULL then xLeft←x.left.size
16:     x.size←root.size+xLeft+1
17:     return x
18: end procedure
```

1.2 Deleting a node from treap

1.2.1 Explanation

The deletion function deletes the node that is pointed by the given pointer. The DELETE_NODE() function takes two arguments , one pointer is pointing to root node of the treap and other pointer points to the node to be deleted. We will check all possible cases while deleting the given node. Because , by deleting the given node the treap property should not be violated. We firstly search for the node to be deleted. We perform binary search to find the node that has to be deleted. If we find the node to be deleted, we will check for several cases to delete it. The possible cases when we find the node to be deleted in the treap.

Case 1: If the node to be deleted has no children. Then we directly delete it and decrease size of it's parent by one.
Case 2: If the node to be deleted has only right child but no left child. We append it's right child to it's parent node and we decrease parent's size attribute by one.
Case 3: If the node to be deleted has only left child but no right child , then we append it's left child to it's parent node and we decrease parent's size attribute by one.
Case 4: If the node to be deleted has both right and left children. Then we check it's priority with both children. If the right child has less priority then we do left rotation and do deletion recursively on it's left child after rotation. But, if we have left child has less priority then we do right rotation on the node and do deletion recursively on right child after rotation.

This deletion function takes help from LEFTROTATION() and RIGHTROTATION() functions (Mentioned in Algorithm 3 and Algorithm 4). This function will return the pointer to the new root node after deleting the given node.This deletion function also uses one more utility function called COUNTNODES() to assign exact value of each size attribute value.(Mentioned in Algorithm 9).

1.2.2 pseudo-code of DELETE_NODE() Algorithm

Algorithm 5 Deleting a node from treap

```
1: procedure DELETE_NODE(*root,*x)
2:     if root==NULL or x==NULL then return root
3:     if x.key<root.key then root.left←DELETE_NODE(root.left,x)
4:     else if x.key>root.key then root.right←DELETE_NODE(root.right,x)
5:     else then
6:         if root.left==NULL and root.right==NULL then
7:             if root.parent! =NULL
8:              then root.parent.size←root.parent.size−1
9:             delete(root)
10:            root←NULL
11:        else if root.left==NULL and root.right! =NULL
12:            tnode * parent←root.parent
13:            root.right.parent←parent
14:            tnode * temp←root.right
15:            if parent! =NULL then parent.size←parent.size−1
16:            delete(root)
17:            root←temp
18:        else if root.left! =NULL and root.right==NULL
19:            tnode * parent←root.parent
20:            root.left.parent←parent
21:            tnode * temp←root.left
22:            if parent! =NULL then parent.size←parent.size−1
23:            delete(root)
24:            root←temp
25:        else
26:            if root.left.priority>root.right.priority
27:                if root.parent! =NULL
28:                 then root.parent.size←root.parent.size−1
29:                 root←LEFTROTATION(root)
30:                 root.left←DELETE_NODE(root.left,x)
31:            else then
32:                ifroot.parent! =NULL
33:                 then root.parent.size←root.parent.size−1
34:                root←RIGHTROTATION(root)
35:                root.right←DELETE_NODE(root.right,x)
36:    COUNTNODES(root)   return root
37: end procedure=0
```

1.3 Searching a key in treap

1.3.1 Explanation

The SEARCH_KEY() function takes two arguments. One of them is pointer to the root node of the treap on which we do searching for the given key. Second argument is a key. We search for the given key , if we find key then we return pointer to that node . If we are unable to find the key in the treap then we will return NULL. In this SEARCH_KEY() function we use binary search technique to search for the given key. If we able to get node which contains the given key then we return that node. If the given key is less than the node key then we traverse to the left child of the node. And, else we move towards the right child of that node. We perform this binary search recursively. Finally, this function will return pointer to the node if the key exist, else it returns NULL.

1.3.2 pseudo-code of SEARCH_KEY() Algorithm

Algorithm 6 Searching a key in treap

```
1: procedure SEARCH_KEY(*root,k)
2:     if root==NULL or root.key==k  then return root
3:     if k<root.key  then return SEARCH_KEY(root.left,k)
4:     else  then return SEARCH_KEY(root.right,k)
5: end procedure=0
```

1.4 Splitting the treap

1.4.1 Explanation

The SPLIT_TREAP() function takes reference to root node of the treap and a key as arguments. This function takes help from another utility function SPLIT() to split the treap. One subtreap will have keys less than given key and other subtreap will have keys greater than the given key. But our SPLIT() function split the given treap as follows , 1. One treap contains keys less-than the given key . 2. Second treap will contain keys that are greater than or equals to the given key. So , if given key exist in the treap then we will delete that node from treap if exist, by using DELETE_NODE() function mentioned previously. So that we will

reach our requirement of getting subtreaps with keys less-than and greater than the given key.

After getting the splitted treaps, we are storing one treaps address in root node and other treap address will be returned to main. Before this address assignment, we have to calculate the size attribute of the node in both treaps. So we use COUNTNODES() function to calculate no.of nodes and assign everynode's value in size attribute.

The COUNTNODES() function takes one pointer that points to the root node of the treap. This function recursively calculates no.of nodes in the subtreap , root pointer as subtreap root and stores in the nodes of the treap. Our SPLIT_TREAP() function also uses one more utility function called MAKEPARENT(). This MAKEPARENT() function recursively assigns each node's parent correctly.

1.4.2 pseudo-code for SPLIT_TREAP() algorithm

Algorithm 7 Splitting the treap

```
1:  procedure SPLIT_TREAP(*&root,k)
2:      tnode *x←root
3:      tnode *left←NULL   right←NULL
4:      SPLIT(x,k,left,right) //Calling utility function SPLIT()
5:      if right! =NULL
6:                  tnode *t←new tnode()
7:                  t.key←k
8:                  right.parent←NULL
9:                  right←DELETE_NODE(right,t)
10:                 MAKEPARENT(right)
11:     if left! =NULL
12:                 leftt.parent←NULL
13:                 tnode *t←new tnode()
14:                 t.key←k
15:                 left←DELETE_NODE(left,t)
16:                 MAKEPARENT(left)
17:     if right! =NULL then COUNTNODES(right)
18:     if left! =NULL then COUNTNODES(left)
19:     root←right
20:     return left
21: end procedure=0
```

Algorithm 8 utility function SPLIT()

```
1: procedure SPLIT(*root,k,*&l,*&r)
2:     if root==NULL then l←NULL and r←NULL
3:     else if k<root.key then
4:             r←root;
5:             SPLIT(root.left,k,l,root.left)
6:     else then
7:             l←root
8:             SPLIT(root.right,k,root.right,r)
9: end procedure
```

Algorithm 9 Utility function COUNTNODES()

```
1: procedure COUNTNODES(*root)
2:     if root==NULL then return 0
3:     else then
4:         root.size←1+COUNTNODES(root.left)+COUNTNODES(root.right)
5:         return root.size
6: end procedure
```

Algorithm 10 Utility function MAKEPARENT()

```
1: procedure MAKEPARENT(*root)
2:     if root! =NULL then
3:         if root.left! =NULL then root.left.parent←root
4:         if root.right! =NULL then root.right.parent←root
5:         MAKEPARENT(root.left);
6:         MAKEPARENT(root.right);
7: end procedure
```

1.5 Joining the treaps

1.5.1 Explanation

Our JOIN_TREAPS() function takes two pointers as arguments, in which one pointer will point to one treap and another pointer will point to another treap. This function will join these two treaps and will return the pointer to the root node of the treap after joining two treaps. In this function we use a

temparory node to join these two treaps. We assign some random values to the temparory node . We assign one treap (T1) to the left child of the temparory node and another treap (T2) to the right child of the temparory node. After assigning given two treaps as children of the temporary node , we calculate size attribute's value of the temporary node. Now, we delete the temporary node created in this function. This function uses DELETE_NODE() to delete the temporary node. After deleting this temporary node, the resulting treap is joined treap. This function will return pointer to the root node of the new treap.

1.5.2 pseudo-code for JOIN_TREAPS() algorithm

Algorithm 11 Joining two treaps

```
1:  procedure JOIN_TREAPS(*root1,*root2)
2:      if root1==NULL then return root2
3:      if root2==NULL then return root1
4:      tnode *temp ← create new tnode()
5:      temp.left←root1
6:      temp.right←root2
7:      temp.parent←NULL
8:      temp.key← −100
9:      temp.priority← −1
10:     root1.parent←temp
11:     root2.parent←temp
12:     tCount←0
13:     tCount2←0
14:     if root1! =NULL then tCount←root1.size
15:     if root2! =NULL then tCount2←root1.size
16:     temp.size←1+tCount+tCount2
17:     temp←DELETE_NODE(temp,temp)
18:     return temp
19: end procedure=0
```

1.6 Inorder traversal of the treap

1.6.1 Explanation

In our INORDER_PRINT() function we print inorder traversal of a treap. This function takes pointer to the root node of the treap as argument and prints inorder traversal of the nodes in the treap. This function will traverse

towards left side of the treap and then prints the value of the node after that will traverse towards the right side of the treap. The output of this function will give us a ascending order of the keys in the treap.

1.6.2 pseudo-code for INORDER_PRINT() algorithm

Algorithm 12 Printing inorder traversal of treap

```
1: procedure INORDER_PRINT(*root)
2:     if root! =NULL then
3:             INORDER_PRINT(root.left)
4:             print root.key
5:             INORDER_PRINT(root.right)
6: end procedure=0
```

1.7 Successor of a node in the treap

1.7.1 Explanation

The function SUCCESSOR() will take one pointer as argument. This function will return the pointer to the node that is successor of the node given as argument (if exist). This function will return NULL if the node doesn't have successor. In this function, if given pointer's right subtreap is not NULL then we find the minimum element in the right subtreap of the the pointer given. If the right subtreap of the given pointer is NULL then we approach other way to find the successor. In this case, we find the root node pointer of the treap. After finding it, we move downwards based on binary search and find the successor of the node. We will return the pointer to the node if successor is found, else we will return NULL.

The SUCCESSOR() function takes help form FINDMINRIGHTSUBTREAP() function. The FINDMINRIGHTSUBTREAP() function takes one pointer as argument , which points to the root of the treap. This function uses iterative method to find smallest element in the right subtreap of the given root node.

1.7.2 pseudo-code for SUCCESSOR() algorithm

Algorithm 13 Finding successor of a node in the treap

```
1:  procedure SUCCESSOR(*root)
2:      if root==NULL then return NULL
3:      if root.right! =NULL
4:       then return FINDMINRIGHTSUBTREAP(root.right)
5:      key←root.key
6:      while root.parent! =NULL  root←root.parent
7:      temp←root
8:      tnode * successor←NULL
9:      while TRUE : then
10:             if key<temp.key then
11:                 successor←temp
12:                 temp←temp.left
13:             else if key>temp.key then
14:                 temp←temp.right
15:             else then
16:                 if temp.right
17:                     successor←FINDMINRIGHTSUBTREAP(temp.right)
18:                 break
19:             if temp==NULL  then  return NULL
20:     return successor
21: end procedure=0
```

Algorithm 14 Utility function FINDMINRIGHTSUBTREAP()

```
1:  procedure FINMINRIGHTSUBTREAP(*root)
2:      if root==NULL then  return root
3:      while (root.left! =NULL):  root←root.left
4:      return root
5:  end procedure
```

1.8 Finding predecessor of a node in the treap

1.8.1 Explanation

The function PREDECESSOR() will take one pointer as argument. This function will return the pointer to the node that is predecessor of the node given as argument (if exist). This function will return NULL if the node does not have predecessor. In this function, if given pointer's left subtreap is not NULL then we find the maximum element in the left subtreap of the the pointer given. If the left subtreap of the given pointer is NULL then we approach other way to find the predecessor. In this case, we find the root node pointer of the treap. After finding it, we move downwards based on binary search and find the predecessor of the node. We will return the pointer to the node if predecessor is found, else we will return NULL.

1.8.2 pseudo-code for PREDECESSOR() algorithm

Algorithm 15 Finding predecessor of a node in the treap

```
1:  procedure PREDECESSOR(*root)
2:      if root==NULL then return NULL
3:      if root.left! =NULL
4:       then return FINDMAXLEFTSUBTREAP(root.left)
5:      key←root.key
6:      while root.parent! =NULL  root←root.parent
7:      temp←root
8:      tnode * predecessor←NULL
9:      while TRUE : then
10:             if key>temp.key then
11:                 predecessor←temp
12:                 temp←temp.right
13:             else if key>temp.key then
14:                 temp←temp.left
15:             else then
16:                 if temp.left
17:                     successor←FINDMAXLEFTSUBTREAP(temp.left)
18:                 break
19:             if temp==NULL then return NULL
20:     return predecessor
21: end procedure=0
```

Algorithm 16 Utility function FINDMAXLEFTSUBTREAP()

```
1: procedure FINMAXLEFTSUBTREAP(*root)
2:     if root==NULL then return root
3:     while (root.right! =NULL): root←root.right
4:     return root
5: end procedure
```

1.9 Finding next greater element than the given key

1.9.1 Explanation

This function takes two parameters as arguments. One of them is point to the root node of the treap and second argument key (k). Our task is to find one element such that the element is smallest element which is greater than the given key (k). This function will return pointer to the node if such type of key exist or NULL if no key is greater than the given key (k). In this function we are trying to find the node which is greater than or equals to the given key. If such node exist we take that node and check that node's key with the given key. If the node's key is equals to the given key then we try to find successor that node and will return the successor's pointer. If the node's key is greater than the given key then we just return the pointer to that node. At-last, if both above cases don't work then we will return NULL from the function

1.9.2 pseudo-code for FIND_NEXT() algorithm

Algorithm 17 Finding next element that is smallest greater than the given key

```
1:  procedure FIND_NEXT(*root,k)
2:      if root==NULL then return root
3:      tnode * dup←root
4:      tnode * parent←NULL
5:      while dup! =NULL
6:              parent←dup
7:              if k<dup.key then dup←dup.left
8:              else if k>dup.key then dup←dup.right
9:              else break
10:     if parent.key==k then return SUCCESSOR(parent)
11:     else if parent.key<k then
12:             while parent! =NULL then
13:                 if parent.key<k then parent←parent.parent
14:                 else break
15:             if parent==NULL then return NULL
16:             else if parent.key>k then return parent
17:             else return NULL
18:     else if parent.key>k then return parent
19:     else return NULL
20: end procedure=0
```

1.10 The number of nodes in the treap that are less than given key

1.10.1 Explanation

This function also takes two arguments. One is pointer to the root node of the treap and other one is key. This NUM_LESS_THAN() function will return an integer value that shows the no.of nodes that are less-than the given key. We are using recursive method to calculate the no.of nodes that are lesser than the given key. This function will check for basic condition in which we check for root node. If root node is NULL we will return 0. In next case, we check given key with the root node's key. If the root node's key is greater than or equals to the given key then we will move to the left subtreap of the root. Because we don't need such nodes whose keys are greater or equals to the given key. Now, the case in which root node's key is less-than the given key then we take count of

the root node and add it's left subtreap size attribute value. Because if root node is less than the given key then automatically it's left subtreap's elements will be less-than the given key. Now we will check for root's right subtreap recursively. We add all nodes in each case and return sum value to the main function.

1.10.2 pseudo-code for NUM_LESS_THAN() algorithm

Algorithm 18 Finding the no.of nodes that are less than given key

```
1:  procedure NUM_LESS_THAN(*root,k)
2:      n←0
3:      if root==NULL then return 0
4:      else if root.key>=k  then  return NUM_LESS_THAN(root.left,k)
5:      else  then
6:                n←1
7:                 if root.left! =NULL then n←n+root.left.size
8:                if root.right! =NULL then n←n+NUM_LESS_THAN(root.right,k)
9:                 return n
10: end procedure=0
```