Analysis and implementation of optimised treaps.

November 7, 2022

Anshul Mittal(2021csb1070 ,
Arpit Gautam (2021csb1073) ,
Aryaman Gupta (2021csb1074)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha Singh

Summary: Treap data structure is a hybrid of a binary search tree and a heap. The structure of the tree is a random variable with the probability distribution after any sequence of key insertions and deletions; in particular, with high probability, its height is proportional to the logarithm of the number of key, so that each search, insertion, or deletion operation takes logarithmic time to perform. It is a self-organizing data structure. They take after themselves and do not require supervision. To put it simply, treaps are a data structure that stores two values keys and priorities, the keys are arranged according binary search trees and priorities are arranged using heap property. Treaps can be used to implement various optimised data structures like Ropes and Fast set in O(log n) time.

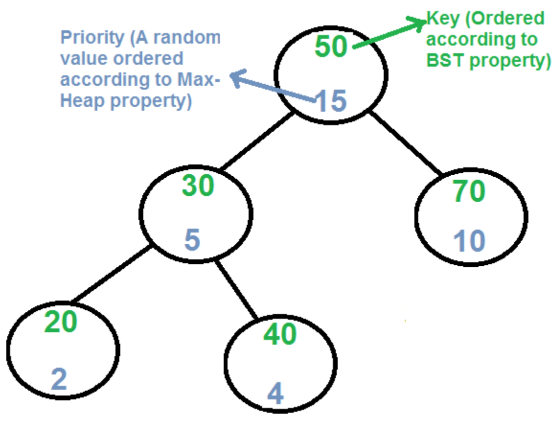
Prerequisites

1. BST or binary search tree is tree in which the each node has maximum of 2 children(left node and right node) and each node satisfies the property - value(left node) $<$ value(parent node) $<$ value (right node).

2. Heap is a complete binary tree in which the value(parent node) $>$ max(value(left node), (right node)).

1. Introduction

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. 1) Key- Follows standard Binary-Search-Tree ordering (left is smaller and right is greater) 2) Priority- Randomly assigned value that follows Max-Heap property.
It's easy to visualise that such a tree exists. Now, since the priority are random, there is high probability of getting a balanced tree.



Priority (A random value ordered according to Max-Heap property)
Key (Ordered according to BST property)
50
15
30
5
70
10
20
2
40
4

Figure 1: A basic treap.

1.1. Insert

1) Create new node with key equals to x and priority equals to a random value. 2) Perform standard Binary search insertion. 3) Use rotations to make sure that inserted node's priority follows max heap property. A, B and C are sub-trees of the tree rooted with P (on left side) or Q (on right side)

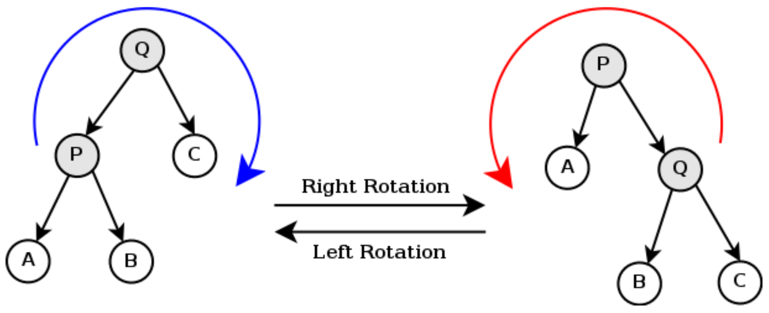


Figure 2: Treap rotation

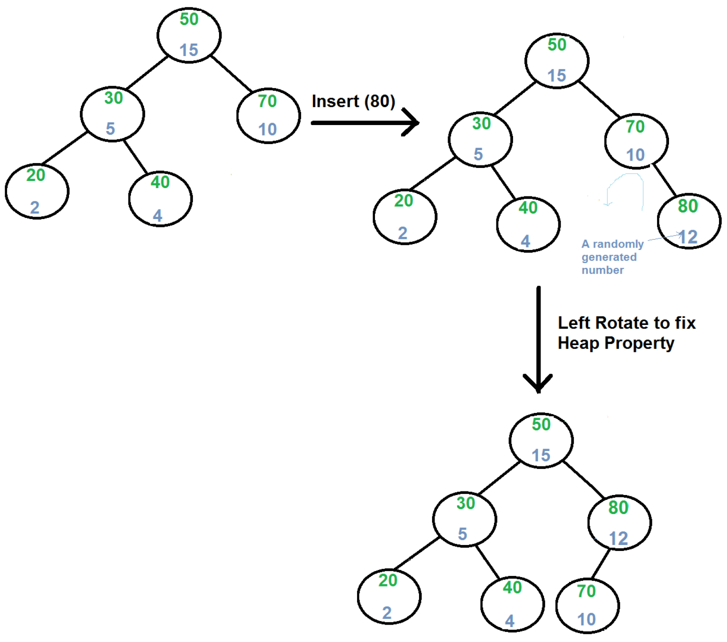


Figure 3: Treap insertion

Keys in both of the above trees follow the following order A < P < B < Q < C.
So Binary search property is not violated anywhere.
Average Time complexity - O(h) where h is the height of the treap. which is generally log n. The worst time -complexity is O(n) in the case of skewed tree.

1.2. Search

It is the standard binary search operation with respect to the key in the treap. If the key is found it will return the node, else will search in the left subtree till it's leaf node then search right subtree. If the element is not found then it will return NULL.
The average time complexity is O(h) i.e. O(log n). The worst time complexity is O(n).

1.3. Delete

It deletes the node given by the user. The key value is searched, if the node leaf, it is directly deleted. Else if the key has only right or left child it is deleted its corresponding child is replaced with node. Else if the key has both left and right child then find the maximum of the two children and delete the node and replace it with the max child.

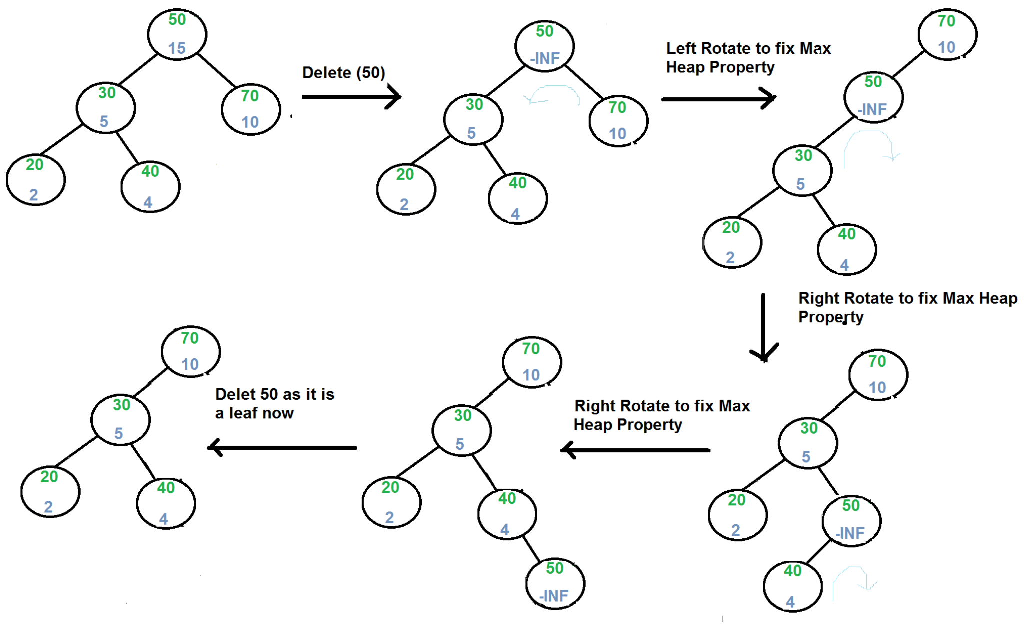

Delete (50)
Left Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Right Rotate to fix Max Heap Property
Delet 50 as it is a leaf now
50
15
30
5
70
10
20
2
40
4
-INF

Figure 4: Treap deletion

The average time complexity is O(h) i.e. O(log n). The worst time complexity is O(n).

1.4. In-order traversal

It's the standard in-order traversal of Binary search tree. The first the left child is printed and then the parent node and then the right child.

1.5. Split

To split a treap into two smaller treaps, those smaller than key x, and those larger than key x, insert x into the treap with maximum priority—larger than the priority of any node in the treap. After this insertion, x will be the root node of the treap, all values less than x will be found in the left subtreap, and all values greater than x will be found in the right subtreap. This costs as much as a single insertion into the treap.
Note that x is added twice so one of the splitted treap will still have x.
Expected time complexity is O(log n).

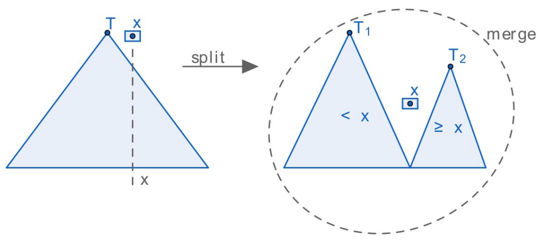


Figure 5: Treap splitting

1.6. Merge

Joining two treaps that are the product of a former split, one can safely assume that the greatest value in the first treap is less than the smallest value in the second treap. Create a new node with value x, such that x is larger than this max-value in the first treap and smaller than the min-value in the second treap, assign it the minimum priority, then set its left child to the first heap and its right child to the second heap. Rotate as necessary to fix the heap order. After that, it will be a leaf node, and can easily be deleted. The result is one treap merged from the two original treaps. This is effectively "undoing" a split, and costs the same. More generally, the join operation can work on two treaps and a key with arbitrary priority (i.e., not necessary to be the highest).
Expected time complexity for the joining two treaps of height n and m is O(log n + log m).

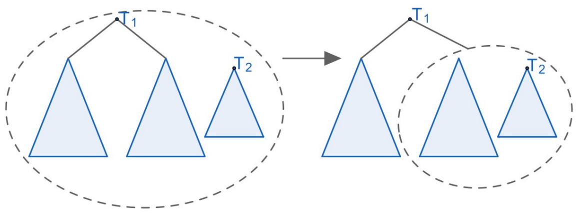


Figure 6: Treap merging

2. Equations

Expected Node Depth for Treap (Randomised Search Tree)
Suppose you have a Treap with N nodes x1, ..., xN , holding keys k1, ..., kN and priorities p1, ..., pN , such that xi is the node holding key ki and priority pi
Let Pr(p1, ..., pN) be the probability of generating the N priority values p1, ..., pN . Note that under the assumption that keys k1, ..., kN are listed in sorted order, the priority values p1, ..., pN determine the shape of the treap and the location of every key in it, so in particular they determine the depth of node xi.
Let the depth of node xi be d(xi), so the number of comparisons required to find key ki in this tree is d(xi)

$$\sum_{p1,....pN} Pr(p1,.....pN)d(xi) = E[d(xi)]$$

3. Tables and Algorithms

3.1. Tables

Given below is the comparison of the time complexities(time taken by each operation) of treaps, heaps and binary trees of various functions.

	Insertion	Search	Deletion	Splitting	Merging
BST	O(log n)	O(log n)	O(log n)	O(n)	O(m+n)
Heaps	O(log n)	O(log n)	O(log n)	O(n*log n)	O(m+n)
Treaps	O(log n)	O(log n)	O(log n)	O(log n)	O(log n+log m)

Table 1: Comparison of time complexity of different data structures.

As you can clearly see treap optimise every operation of both its parts.

3.2. Algorithms

Algorithm 1 Create

```
1: It creates a node with its pointer being NULL and values of key and priority being 0.
2: New.key = 0
3: New.priority = 0
4: New.right = NULL
5: New.left = NULL
6: return New
```

Algorithm 2 Insert(root, k, num)

```
1: It inserts the element at it correct position using rotations.
2: if Root=NULL then
3:     allocating memory to a new Node N
4:     N.key = k
5:     N.left = NULL
6:     N.right = NULL
7:     if num=0 then
8:         N.priority = maximum random number.
9:     else
10:        N.priority = random num
11:    end if
12:    return N
13: else if Root.key<k then
14:    Root.right=INSERT(Root.right,k,num)
15:    if Root.right.priority>Root.priority then
16:        Root=LEFT ROTATE(Root)
17:    end if
18: else
19:    Root.left=INSERT(Root.left,k,num)
20:    if Root.left.priority>Root.priority then
21:        Root=RIGHT ROTATE(Root)
22:    end if
23: end if
24: return Root
```

Algorithm 3 Delete(root, val)

```
if Root=NULL then
    return Root
end if
if val<Root.key then
    Root.left = DELETE(Root.left,val)
else if val>Root.key then
    Root.right=DELETE(Root.right,val)
else if Root.left=NULL then
    Root = Root.right
else if Root.right=NULL then
    Root = Root.left
else if Root.left.priority < Root.right.priority then
    Root = LEFT ROTATE(Root)
    Root.left = DELETE(Root.left,val)
else
    Root = RIGHT ROTATE(Root)
    Root.right = DELETE(Root.right,val)
end if
return Root
```

Algorithm 4 Search(root, val)

```
if Root=NULL or Root.key=val then
    return Root
else if Root.key>val then
    return SEARCH(Root.left,val)
else
    return SEARCH(Root.right,val)
end if
```

Algorithm 5 Merge(root1, root2)

```
Node Root
if Root1=NULL then
    return Root2
end if
if Root2=NULL then
    return Root1
end if
if Root1.pr > Root2.pr then
    Root = EMPTY TREAP(Root2.key,Root2.priority)
    Root.left = MERGE(Root1,Root2.left)
    Root.right = Root1.right
else
    Root = EMPTY TREAP(Root1.key,Root1.priority)
    Root.left = Root1.left
    Root.right = MERGE(Root1.right,Root2)
end if
return Root
```

Algorithm 6 Right Rotate Treap(root)

```
Declaring a new Node 'x'
x = T.left
Declaring a new Node 'z'
z = x.right
x.right = T
T.left = z
return x
```

Algorithm 7 Left Rotate Treap(root)

```
Declaring a new Node 'y'
y = T.right
Declaring a new Node 'z'
z = y.left
y.left = T
T.right = z
return y
```

Algorithm 8 Split(root, val)

```
if SEARCH(Root,val)!=NULL then
    DELETE(Root,val)
    Root = INSERT(Root,val,0)
    Declaring a new Node 'l root'
    Declaring a new Node 'r root'
    l root = Root
    r root = Root.right
    Root.right = NULL
    Root = NULL
    if l root.left!=NULL and l root.right!=NULL then
        if l root.left.priority > l root.right.priority then
            l root.priority = l root.left.priority + 1
        else
            l root.priority = l root.right.priority + 1
        end if
    else if l root.left == NULL and l root.right!=NULL then
        l root.priority = l root.right.priority + 1
    else if l root.left!=NULL and l root.right==NULL then
        l root.priority = l root.left.priority + 1
    else
        l root.priority = 1
    end if
else
    Root = INSERT(Root,val,0)
    l root = Root.left
    r root = Root.right
    Root.left = NULL
    Root.right = NULL
end if
```

Algorithm 9 Inorder(root)

```
1: if root == NULL then
2:     return;
3: end if
4: inorder(root->left)
5: print root->value , print root->priority
6: inorder(root->right)
```

4. Applications

Some applications of treaps are as follows.

Proposition 4.1. ***Implicit Treaps*** *: array indices of elements as keys , instead of the values - to simplify all the operations supported by a segment tree along with the power to split an array into two parts and merge two different arrays into a single one, both of them in O(logN) time.*

Implementation of Implicit Treap: Since we are using the array index as the key of the BST this time, with each update (insertion / deletion), we will have to change O(n) values (the index of O(n) nodes would change upon an insertion/deletion of an element in array). This would be very slow.; To avoid this, we will not explicitly store the index i (i.e. the "key" or Bk value) at each node in the implicit treap and calculate this value on the fly. Hence the name Implicit Treap because the key values are not stored explicitly and are implicit. The key value for any node x would be 1 + no of nodes in the BST that have key values less than x . (where node x means node representing A[x]). **Note** Nodes having key less than x would occur not only in the left subtree of x , but also in the left subtree of all the parents p of x such that x occurs in the right subtree of p.
• Hence the key for a node t = sz(t->l) + sz(p->l) for all parents of t such that t occurs in the right subtree of p.

Applications of implicit treaps.

1. Inserting an element in the array in any location.;
2. Delete an element at any position.
3. Finding sum, minimum / maximum element etc. on an arbitrary interval.
4. Addition, painting on an arbitrary interval
5. Reversing elements on an arbitrary interval.
6. and many more uses like implementation of Ropes!

Proposition 4.2. ***Ropes :*** *A rope is a binary tree where each leaf (end node) holds a string and a length (also known as a "weight"), and each node further up the tree holds the sum of the lengths of all the leaves in its left subtree. A node with two children thus divides the whole string into two parts: the left subtree stores the first part of the string, the right subtree stores the second part of the string, and a node's weight is the length of the first part.*

Implementation of Ropes:

1. Create a new root node (that stores the root of the new concatenated string)
2. Mark the left child of this node, the root of the string that appears first.
3. Mark the right child of this node, the root of the string that appears second.

Since this method only requires to make a new node, it's time complexity is O(1).

Advantages of ropes:

1. Ropes drastically cut down the cost of appending two strings.
2. Unlike arrays, ropes do not require large contiguous memory allocations.
3. Ropes do not require O(n) additional memory to perform operations like insertion/deletion/searching.
4. In case a user wants to undo the last concatenation made, he can do so in O(1) time by just removing the root node of the tree.

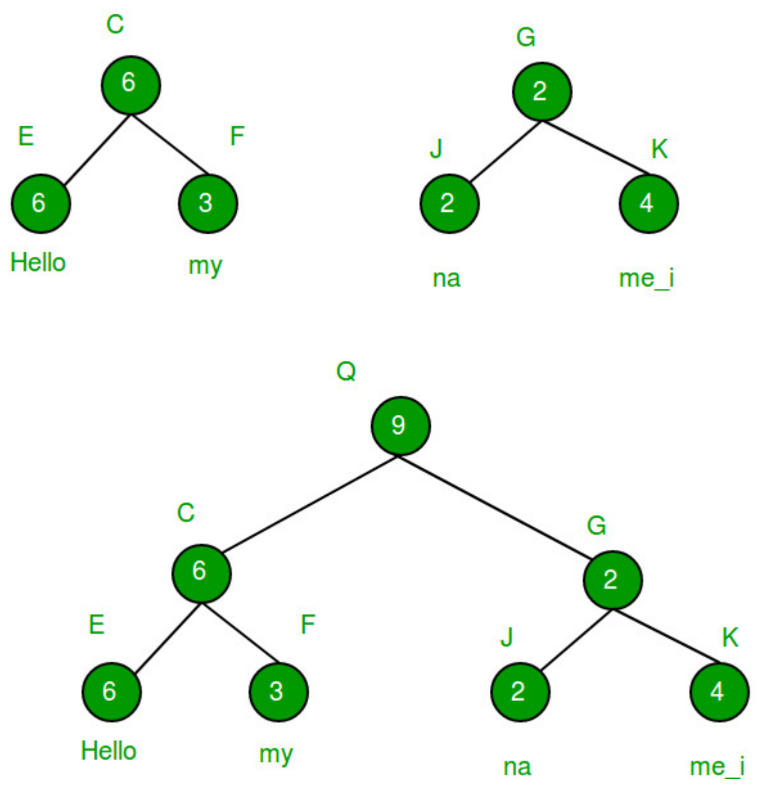


Figure 7: String concatenation

Algorithm 10 $Createrope(ropes **boot1, char * arr, intLeft, intRight)$

```
1: Rope structure is a similar to a treap structure in which priorities are replaced with string and key contains the size of the strings.
2: arr contains the string of which a rope has to be formed and Left and Right are index of first character and last character respectively.
3: maxleaf stores the maxsize of the each node of the rope
4: Declare a new node rope.
5: **if** Right - Left > maxleaf **then**
6:     temp->data = (Right - Left) / 2;
7:     *boot1 = temp;
8:     createrope(&$(*boot1)->left, arr, left, (left + right)/2)$
9:     createrope(&$(*boot1)->right, arr, (left + right)/2 + 1, right)$
10: **else**
11:     temp->data = Right - Left
12:     int j = 0
13:     temp->str = (char *)malloc((maxleaf)*sizeof(char))
14:     **for** i= Left and i<Right **do**
15:         temp->str[j++] = arr[i]
16:     **end for**
17:     *boot1 = temp;
18: **end if**
```

Algorithm 11 concatenate(root, root1, int size)

```
1: root and root1 contains the ropes of the string to be concantenated.
2: root2 =NULL;
3: declare a new rope temp
4: temp->left =root
5: temp->right = root1
6: temp->data = size
7: root2 = temp
8: return root2
```

Proposition 4.3. ***Implementation of Fast set:***

1. Treaps use randomization to maintain balance in dynami- cally changing search trees. Each node in the tree has an associated key and a random priority. The data are stored in the internal nodes of the tree so that the tree is in in-order with respect to the keys and in heap-order with respect to the priorities.
2. Set operations are used extensively for in-dex searching—each term (word) can be represented as a set of the "documents" it appears in and searches on log- ical conjunctions of terms are implemented as set opera- tions (intersection for and, union for or, and difference for and-not)
3. For two sets of size n and m with algorithms for union, intersection, and difference run in expected O(m lg(n/m)) serial time or parallel work. This is optimal due to implemetation of fast split function in Treaps(used as a set here).

5. Conclusions

In this report, we discussed the implementation of treaps to make operations like split,merge quite faster as compared to other data structures.Moreover, we learned how insertion,deletion,search of data entries present at random/arbitrary index in the structure are fast in the case of balanced tree i.e. mathematically we proved that treaps are highly probable to be always balanced thus making these operations quick.This arises the conclusion that Treaps are essentially Randomised Search Trees with priority settings hence, they form an excellent data structure

6. Bibliography and citations

- Introduction to algorithms by Cormen, Leiserson, Rivest, Stein(shortly CLRS).
- Fast Set Operations Using Treaps by Guy E. Blelloch Margaret Reid-Miller from Carnegie Mellon University Pittsburgh.
- Wikipedia - treaps.
- Geekforgeeks - treaps
- usaco - treaps.
- Lecture 8 by Dave mount from CMSC.

Acknowledgements

I would like to express my gratitude and appreciation for our teacher Dr. Anil Shukla whose teaching layed the foundation for our project. I would also like thank our Teaching Assistant Monisha Mam whose invaluable insights helped us reach our destination