Analysis and Implementation of Optimized Treaps

November 21, 2021

Janmeet Singh Makkar (2020CSB1175) ,
Tarushi (2020CSB1135) ,
Princy Malhotra (2020MCB1193)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Napendra Solanki

Summary: Treaps are data structures which are a combination of Trees and Heaps.They are used for reducing time complexity while simultaneously setting a priority for each value a user enters. This helps in an optimised ordering and arrangement of Tree instead of a heap where order depends upon either maximum or minimum of a value.To simplify, Treaps execute the priority parameter using property of Heaps and the keys take place using the property of trees.This genius structural combination gives us the power to perform operations like insert,search,delete,intersection,union along with special applications like optimised implementation of Ropes and Sets using implicity of keys in O(log n) time.These data structures although may sound simple but tend to make a lot of complex practical tasks easy, taking for instance a normal activity monitor(A treap) in your laptop which arranges applications you are currently using(Keys) according to order of battery consumption(Priority).

---

1. Introduction

Storing sets of items so as to allow for fast access to an item given its key is a ubiquitous problem in computer science. Let X be a set of n items each of which has associated with it a *key* and a *priority*. The keys are drawn from some totally ordered universe ,and so are the priorities The two ordered universes need not be the same. A *treap* for X is a rooted binary tree with node set X that is arranged in in-order with respect to the keys and in heap-order with respect to the priorities."In-order for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and x.key for y in the right subtree of x."Heap order" means that for any node x with parent z the relation $x.priority \leq z.priority$ holds. It is easy to see that for any set X such a treap exists. With the assumption that all the priorities and all the keys of the items in X are distinct – a reasonable assumption for the purposes of this paper – the treap for X is unique: the item with largest priority becomes the root and the allotment of the remaining items to the left and right subtree is then determined by their keys. Put differently the treap for an item set X is exactly the binary search tree that results from successively inserting the items of X in order of decreasing priority into an initially empty tree using the usual leaf insertion algorithm for binary search trees.
The various operations that can be performed on Treaps are given below :

1.1. INSERT

The Insert function is used to insert a new node into the treap. The key value of the node is given as input by the user and priority is alloted randomly. There is a high probability that giving random priority will violate the Max-Heap Property of the Treap. So, to avoid this, we perform left and/or right rotations on the treap.
The average case time complexity for inserting a node is O(h) (h = height of treap, which is log n) i.e. O(log n) (where n is the number of nodes) and the worst case time complexity is O(n) (in case of a skewed treap)

1.2. DELETE

The delete function is used to delete a node from the treap. The key value of the node to be deleted is given by the user. If the node to be deleted is a leaf node, it is directly deleted. In case it has exactly one child (either left or right), the node is replaced with that child and then deleted. Else if it has both children (left and right), then find the maximum out of the children and perform rotations on the node accordingly.
The average case time complexity is, again, O(h) i.e. O(log n) and the worst case time complexity is O(n) (in case of skewed treap).

1.3. INORDER TRAVERSAL

The inorder traversal function is used to traverse the treap in non-descending (ascending) order. The inorder traversal of a treap is performed in exactly similar manner as in Binary Search Trees, depending on the key value of the node.
Since all the nodes of the treap are visited exactly once in inorder traversal, the expected time complexity is O(n).

1.4. SEARCH

The search function is used to search for a node in the treap. The key value of the node to be searched is given as input by the user. The search performed on a treap is exactly similar to that performed in a BST (Binary Search Tree).
We will start at the root and compare the value to be searched with the root node. If the value to be searched, say, val, is smaller than the root node value, perform same operation on the left sub-tree of root else perform the same operation on right sub-tree of root.
The average case time complexity is, again, O(h) i.e. O(log n) and the worst case time complexity is O(n) (in case of skewed treap).

1.5. SPLIT

Recursively, split function splits the right child of the root by the 'pivot' value, and then makes the resulting tree with keys less than pivot the new right child of the root and makes the resulting tree with keys greater then pivot the "greater-than" tree. Similarly, if the root key is greater than pivot split recursively splits the left child of the root. If the root key is equal to pivot, split returns the root and the left and right children as the "less-than" and "greater-than" trees, respectively.
The expected time to split a treap is O(log n)

1.6. MERGE

To merge two treaps T1 with keys less than a and T2 with keys greater than a, join traverses the right spine of T1 and the left spine of T2. A left (right) spine is defined recursively as the root plus the left (right) spine of the left (right) sub-tree. To maintain the heap order, join interleaves pieces of the spines so that the priorities descend all the way to a leaf.
The expected time to join two treaps of size n and m is O(log n + log m)

2. Equations

Expected Node Depth for Treap (Randomised Search Tree)

Suppose you have a Treap with N nodes $x_1, ..., x_N$, holding keys $k_1, ..., k_N$ and priorities $p_1, ..., p_N$, such that x_i is the node holding key k_i and priority p_i

Let Pr($p_1, ..., p_N$) be the probability of generating the N priority values $p_1, ..., p_N$.
Note that under the assumption that keys $k_1, ..., k_N$ are listed in sorted order, the priority values $p_1, ..., p_N$ determine the shape of the treap and the location of every key in it, so in particular they determine the depth of node x_i

Let the depth of node x_i be d(x_i), so the number of comparisons required to find key k_i in this tree is d(x_i)

$$\boldsymbol{E[d(x_i)]} = \sum_{p_1,\ldots,p_N} Pr(p_1, ..., p_N)d(x_i), \tag{1a}$$

Expected Depth and ancestors

The depth of a node is just the number of ancestors it has ; so

$$\boldsymbol{E[d(x_i)]} = \sum_{p_1,\ldots,p_N} Pr(p_1, ..p_N) \sum_{m=1}^{N} A_{mi} = \sum_{m=1}^{N} E[A_{mi}] \tag{2a}$$

$$\text{where indicator function } A_{ij} = \begin{cases} 1 & \text{if } x_i \text{ is an ancestor of } x_j \\ 0 & \text{,otherwise} \end{cases} \tag{2b}$$

The probability that node x_m is an ancestor of x_i is just the probability that the random priority generated for x_m is higher than the other $|m-i|$ priorities generated for nodes with indexes between m and i inclusive. But since the priorities are generated randomly and independently, each of those $|m-i|$ +1 nodes have equal probability of having the highest priority. So,

$$\boldsymbol{E[A_{mi}]} = \frac{1}{|m-i|+1} \tag{2c}$$

$$\boldsymbol{E[d(x_i)]} = \sum_{m=1}^{N} E[A_{mi}] = \sum_{m=1}^{N} \frac{1}{|m-i|+1} \tag{2d}$$

Assuming that all the keys are equally likely to be searched. So, the expected node depth, averaged over all nodes:

$$D_{avg}(N) = \sum_{i=1}^{N} \frac{1}{N} E[d(x_i)] = \frac{1}{N} \sum_{i=1}^{N} \sum_{m=1}^{N} \frac{1}{|m-i|+1} \tag{2e}$$

Upon simplifying the double summation we get

$$\sum_{i=1}^{N} \sum_{m=1}^{N} \frac{1}{|m-i|+1} = 2 \sum_{i=1}^{N} \frac{N-i+1}{i} - N = 2(N+1) \sum_{i=1}^{N} \frac{1}{i} - 3 \tag{2f}$$

3. Figures, Tables and Algorithms

3.1. Figures

Below are the graphs comparing Time complexities of various functions for various Data Structures.In each Graph X axis represents the Time taken for a particular function to run and Y axis represents 'n' the total elements in the data structure.

Figure 1: Comparison of Time complexity Of Insert Function for various Data Structures

Figure 2: Comparison of Time complexity Of Delete Function for various Data Structures

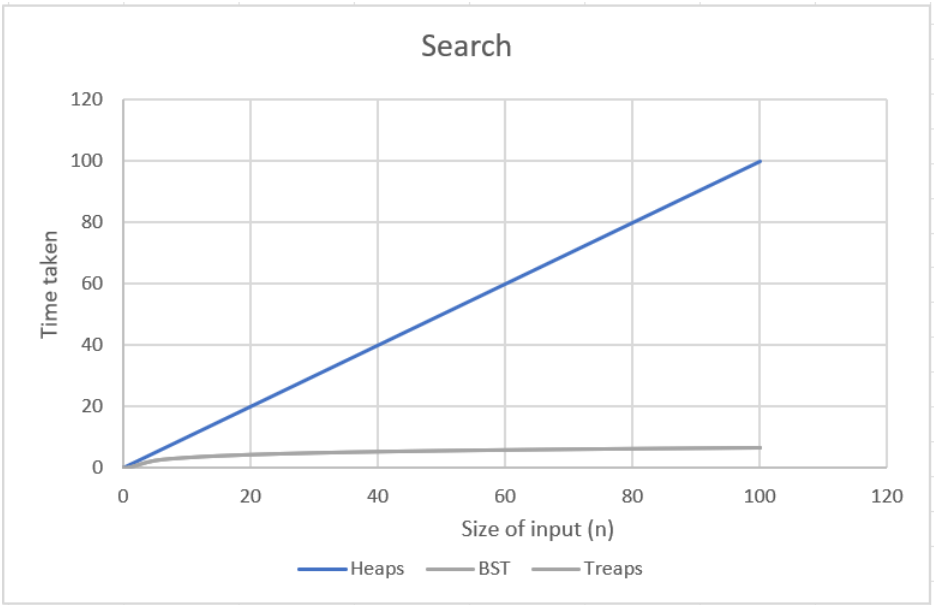

Figure 3: Comparison of Time complexity Of search Function for various Data Structures

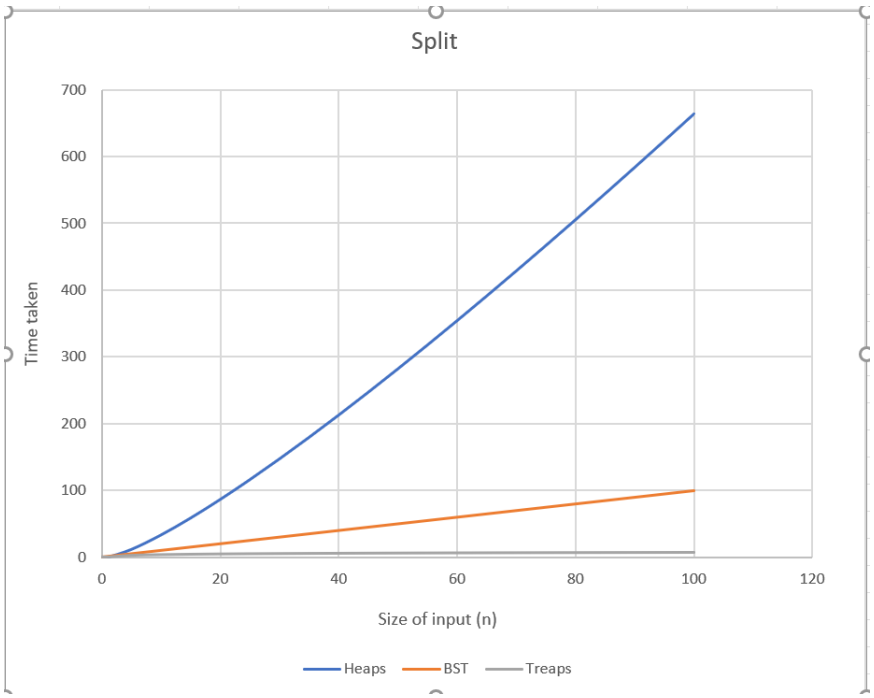

Figure 4: Comparison of Time complexity Of split Function for various Data Structures

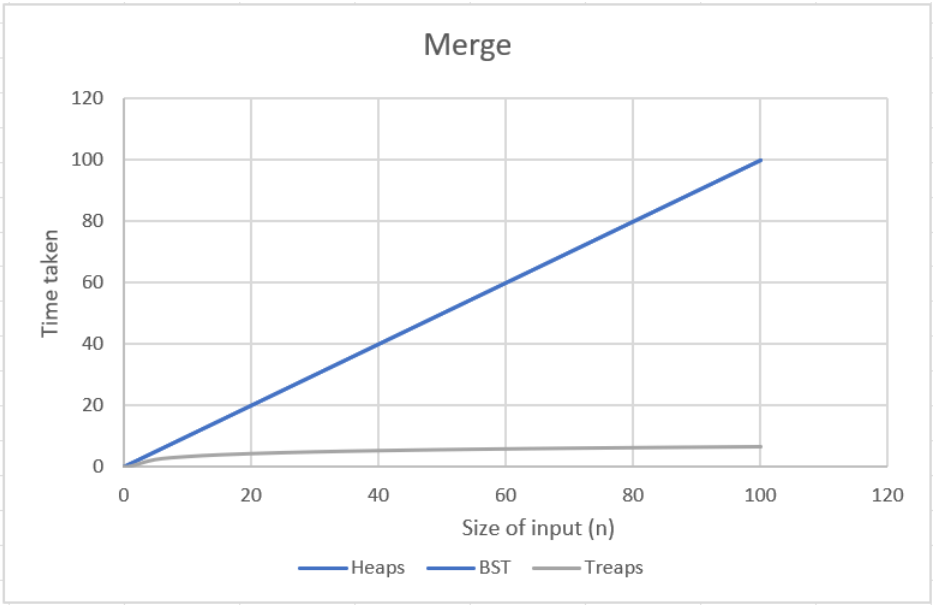

Figure 5: Comparison of Time complexity Of merge Function for various Data Structures

3.2. Tables

Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. Here, the length of input indicates the number of operations to be performed by the algorithm. Below Table compares the time complexities of various operations like Insert,Delete,Search,Join and Split for some Data Structures.

Table showing the Time Complexities of various operations

	Insert	Delete	Search	Join	Split
Heaps	O(log n)	O(log n) (avg. case)	O(n)	O(m+n)	O(n log n)
BST	O(log n) (avg. case)	O(log n) (avg. case)	O(log n) (avg. case)	O(m+n)	O(n)
Treaps	O(log n) (avg. case)	O(log n) (avg. case)	O(log n) (avg. case)	O(log m + log n)	O(log n)

Table 1: **Time Complexities**

3.3. Algorithms

Algorithm 1 Create an Empty Treap

```
EMPTY TREAP(key,num)
Assigning memory to a new Node
New.key = key
New.priority = num
New.right = NULL
New.left = NULL
return New
```

Algorithm 2 Insert an element(nodes) in a Treap

```
INSERT(Root,k,num)
if Root=NULL then
    allocating memory to a new Node N
    N.key = k
    N.left = NULL
    N.right = NULL
    if num=0 then
        N.priority = 1e9
    else
        N.priority = random num
    end if
    return N
else if Root.key<k then
    Root.right=INSERT(Root.right,k,num)
    if Root.right.priority>Root.priority then
        Root=LEFT ROTATE(Root)
    end if
else
    Root.left=INSERT(Root.left,k,num)
    if Root.left.priority>Root.priority then
        Root=RIGHT ROTATE(Root)
    end if
end if
return Root
```

Algorithm 3 Delete an element in a Treap

```
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

```
SEARCH(Root,val)
1: if Root=NULL or Root.key=val then
2:    return Root
3: else if Root.key>val then
4:    return SEARCH(Root.left,val)
5: else
6:    return SEARCH(Root.right,val)
7: end if
```

Algorithm 5 Merge two Treaps

```
MERGE(Root1,Root2)
1: Node Root
2: if Root1=NULL then
3:    return Root2
4: end if
5: if Root2=NULL then
6:    return Root1
7: end if
8: if Root1.pr > Root2.pr then
9:    Root = EMPTY TREAP(Root2.key,Root2.priority)
10:   Root.left = MERGE(Root1,Root2.left)
11:   Root.right = Root1.right
12: else
13:   Root = EMPTY TREAP(Root1.key,Root1.priority)
14:   Root.left = Root1.left
15:   Root.right = MERGE(Root1.right,Root2)
16: end if
17: return Root
```

Algorithm 6 Right Rotate Treap (when an element is deleted or treap is split)

RIGHT ROTATE(T)

```
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 (when an element is deleted or treap is split)

LEFT ROTATE(T)

```
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 a Treap

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 Traversal

```
INORDER(Root)
1: if Root!=NULL then
2:    inorder(Root.left)
3:    printf("Key : %d, Priority : %d",Root.key,Root.priority)
4:    if Root.left then
5:       printf(" , Left Child : %d ",Root.left.key)
6:    else
7:       printf("Left child = NULL ")
8:    end if
9:    if Root.right then
10:      printf(" , Right Child : %d ",Root.right.key)
11:   else
12:      printf("Right child = NULL")
13:   end if
14:   Print a new line character
15:   inorder(Root.right)
16: end if
```

4. Some further useful suggestions and Applications

Applications of Treaps can be proposed 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 that 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, or cord, is a data structure composed of smaller strings that is used to efficiently store and manipulate a very long string. For example, a text editing program may use a rope to represent the text being edited, so that operations such as insertion, deletion, and random access can be done efficiently.*

A Rope is a binary tree structure where each node except the leaf nodes, contains the number of characters present to the left of that node. Leaf nodes contain the actual string broken into substrings (size of these substrings can be decided by the user). Like dictionaries, ropes fall under the category of abstract data types,

but that's where the similarities end. They extend arrays with the two new abilities beyond indexing:

1. Join (i.e. concatenate) two ropes into a single one
2. Split a rope at a point into two halves

By using **random** numbers for the Cartesian tree, we ensure our operations are almost always O(logn) i.e. fast. However, we've lost the ability to store anything useful. To remedy this, we attach an additional value to each node (and call the random numbers the nodes' **priority**, to avoid confusion). Since the value can change independently to the priority, we can allow updating it in-place thus making it an **application of treap.** Or, we can forbid it. Since each node only ever knows about its subtree, we can update by instead by splitting out the element, cloning it, then joining the pieces back together. And with that, we have a fully functional rope. You can store sequences of any type, join them, split them, index them, even summarise them.

Proposition 4.3. ***Fast Set Operations Using Treaps*** *:*

- 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.

- 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)

- 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

Following are the sources that were used to make this report and program credible and refined -

- Fast Set Operations Using Treaps by Guy E. Blelloch and Confluently persistent sets and maps by Olle Liljenzin [6, 8] made us understand about the practicality and math behind treaps.
- Randomized Search Trees by Seidal and Aragon [1] helped in laying the foundation of treaps using random priorities.
- Furthermore, we were introduced with the concept of Ropes data structure by Hans-j. Boehm ,Russ Atkinson and, Michael Plass [3] which ultimately emerged as an excelled application of treaps.
- Visual Introduction to Treaps was provided to this report with the help of Igor Carpanese's aricle on Medium[2].
- Moreover , lecture notes by Professor Dave Mount [7], Kube[5], and Jeff Erickson[4] were helpful to clear basic concepts involved in this report.

Acknowledgements

We would like to thank the person responsible for laying our foundation in Data Structures, **Dr.Anil Shukhla** and acknowledge our teaching assistant Sir Napendra Solanki for helping us out through this project.Moreover, throught this report, we were inspired by Professor Guy E. Blelloch and Dr.Margaret Reid-Miller for their extensive research and understanding of Treaps.

References

[1] Raimund Seidel ; Cecilia R. Aragon. *Randomized Search Trees(Algorithmica)*. 1996.

[2] Igor Carpanese. A visual introduction to treap data structure.

[3] Hans j. Boehm ;Russ Atkinson and; Michael Plass. *Ropes: an Alternative to Strings*, volume 25 of *12*. 3333 Coyote Hill Rd., Palo Alto, CA 94304, U.S.A, 1993.

[4] Jeffe. Lecture 3: Treaps and skip lists [sp'17].

[5] Kube. Cls 100: Lecture 8 trees, heaps, and treaps.

[6] Olle Liljenzin. Confluently persistent sets and maps. 2013.

[7] Dave Mount. Cmsc 420: Lecture 8 treaps.

[8] Guy E. Blelloch;Margaret Reid-Miller. *Fast Set Operations Using Treaps*. ACM, New York, USA, 1998.