Report on Assignment -2 (Treaps)

Submitted by

Suresh Kumar Amalapuram

Roll No. CS19RESCH11001

AdvancedDataStructures & Algorithms

Correctness of the Algorithm

1. Operation like insertion,deletion,split etc are Implemented per the description given in the **Introduction to Algorithms, 3e CLRS**
2. Correctness is guaranteed by maintaining
   - Binary Search Tree structure
   - Min-Heap Property
3. We maintain above properties by using Rotations (Left,Right), whose correctness was discussed in the class

Efficiency of the Algorithm

0.1 Insertion

0.1.1 Time Complexity

1. Insertion always occurs at the leaf Node, we may need multiple rotations to maintain treap structure
2. Worst case time for Multiple rotations is proportion to **height of the treap(h)**
3. Total time Complexity $= O(h * c) = O(h) = O(\log N)$
   - c= constant time for rotation
   - h=height of the treap
   - N= Number of keys in Treap

0.1.2 Space Complexity

1. Size for treap = $O(n)$

- n = number of keys in treap

2. Depth of the stack space = $O(h) = O(\log N)$

   - **Stack Space will be characterized by number of activation records** it has.
   - For **Each Recursive call there will be an one activation record** maintained in the stack space
   - Stack space will contains **h activation records**, since worst case rotations will progress up to height of the treap

3. Total Space Complexity $= O(n+h) = O(n)$

0.2 Deletion

0.2.1 Time Complexity

1. Deletion at the given Node, involves multiple rotations to push the node to the leaf level
2. Worst case time for Multiple rotations is proportion to **height of the treap(h)**
3. Total time Complexity $= O(h*c) = O(h) = O(\log N)$

   - h= height of the treap
   - c= constant time for rotation
   - N= Number of keys in Treap

0.2.2 Space Complexity

1. Size for treap = $O(n)$

   - n = number of keys in treap

2. Depth of the stack space = $O(h) = O(\log N)$

   - **Stack Space will be characterized by number of activation records** it has.
   - For **Each Recursive call there will be an one activation record** maintained in the stack space
   - Stack space will contains **h activation records**, since worst case rotations will progress up to height of the treap

3. Total Space Complexity $= O(n+h) = O(n)$

0.3 Merge

0.3.1 Time Complexity

1. creates a new temporary node, attach both the trees as left/right child to temporary node and removes the temporary node
2. Worst case time for Merge is the time for deletion of temporary node which proportion to **height of the treap(h)**
3. Total time Complexity $= Oh*c) = O(h) = O(\log N)$

   - c= constant time for rotation
   - h=Height of the Treap
   - N=Number of keys in Treap

0.3.2 Space Complexity

1. Size for treap1 = $O(n_1)$

   - n1 = number of keys in treap1

2. Size for treap2 = $O(n_2)$
    - n2 = number of keys in treap2
3. Depth of the stack space for deletion = $O(h) = O(\log N)$
4. h= height of the treap
5. N=Number of elements in Treap
    - **Stack Space will be characterized by number of activation records** it has.
    - For **Each Recursive call there will be an one activation record** maintained in the stack space
    - Stack space will contains **h activation records**, since worst case rotations will progress up to height of the treap
6. Total Space Complexity $= O(n_1 + n_2 + h) = O(n_1 + n_2)$

0.4 Split

0.4.1 Time Complexity

1. Searches for the given key k.if key k not found, then insert k into treap
2. Splits the Treap into two treapsT_1 and T_2 based on Key K
3. Based on the position of Splitter node, Splitting is an walk from root to the splitter node and manipulating the pointers, which is proportion to the height of the treap
4. Total time Complexity $= O(h * c) = O(h) = O(\log N)$
    - h= height of the treap
    - c= constant time for pointer manipulations
    - N= Number of keys in Treap

0.4.2 Space Complexity

1. Size for treap1 = $O(n_1)$
    - n1 = number of keys in treap1
2. Size for treap2 = $O(n_2)$
    - n2 = number of keys in treap2
3. Number of elements in the Original Treap=$n_1 + n_2$

Total Space Complexity $= O(n_1 + n_2 + h) = O(n_1 + n_2)$

0.5 Search,Predecessor,Successor,Find_next

0.5.1 Time Complexity

1. All these operations are proportion to the Height of an Treap
2. Total time Complexity $= O(h) = O(\log N)$
    - h= height of the treap
    - N=Number of keys in Treap

_

Description of each Function in Code

0.5.2 int _getnodePriority()

- Randomly assigns priorities to the Keys

0.5.3 tnode* _findRootNode(tnode* p_root)

- Returns Root of the Treap with p_root is one of its element

0.5.4 void _calculateNodeSize

- calculates current node size based on its left and right child

0.5.5 void _adjustAncestorNodeSize(tnode* p_node)

- Adjusts Node sizes of all nodes in the path from from current node to root of the treap

0.5.6 tnode* _getTreapMaxValue(tnode* p_node)

- Returns a pointer to the node with maximum value in treap rooted with p_node

0.5.7 tnode* _getTreapMinValue(tnode* p_node)

- Returns a pointer to the node with minimum value in treap rooted with p_node

0.5.8 tnode* _leftRotateforInsert(tnode* p_root)

- This function is invoked after inserting a element in the treap, to maintain heap property

0.5.9 tnode* _RightRotateforInsert(tnode* p_root)

- This function is invoked after inserting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.5.10 tnode* _leftRotateforDelete(tnode* p_root)

- This function is invoked after deleting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.5.11 tnode* _RightRotateforDelete(tnode* p_root)

- This function is invoked after deleting a element in the treap, to maintain Treap structure property (Binary search Tree + Heap property)

0.5.12 tnode* _validateHeapProperty(tnode* p_root)

- This function will ensure maintaining Treap structure property (Binary search Tree + Heap property) at p_root node

0.5.13 tnode* search_key(tnode *root,float key)

- This function will searches for the node whose value is key, returns NULL if not found

0.5.14 void inorder_printProxy(tnode *root)

- This is proxy method for inorder_print method

0.5.15 void inorder_print(tnode* root)

- This method will print in order sequence of the treap

0.5.16 tnode* _createNode(float p_key)

- creates new node with key value

0.5.17 tnode* insert_keyProxy(tnode *root, float k)

- This is an proxy method inserting an element into the treap, similar to binary search tree insertion

0.5.18 tnode* insert_key(tnode * & root, float k)

- This method will insert an element into treap and may apply multiple rotations (from bottom to top) to maintain Treap structure property (Binary search Tree + Heap property)
- This method will adjust all the node sizes of the ancestors nodes from newly inserted node to the root of the treap
- Validates the Heap property

0.5.19 tnode* _delete_nodeproxy(tnode *x)

- This is an proxy method for deleting an element from treap, similar to binary search tree deletion. may apply multiple rotations (from top to bottom) to maintain Treap structure property (Binary search Tree + Heap property)
- This method will adjust all the node sizes of the ancestors nodes from deleted node to the root of the treap
- Validates the Heap property

0.5.20 tnode* delete_node(tnode *root, tnode *x)

- This method invokes proxy delete method to perform deletion on treap.

0.5.21 tnode* join_treaps(tnode *root1, tnode *root2)

- k ⟵ (maxkey(root1)+minkey(root2))/2
- create a new root with the key k
- Root1 will be the left sub-tree of k and Root2 will be the right sub-tree
- Delete the node k

0.5.22 tnode* successor(tnode *x))

- Similar to binary search tree successor finding

0.5.23 tnode* predecessor(tnode *x)

- Similar to binary search tree Predecessor finding

0.5.24 tnode* find_next(tnode* root, float k)

- Returns the smallest key larger than the key k, uses recursive procedure

0.5.25 tnode* find_previous(tnode *root, float key)

- Returns the largest key smaller than the key k, uses recursive procedure

0.5.26 int num_less_than(tnode *root,float k)

- Returns Number of nodes in the treap rooted at root, with keys less than k

0.5.27 tnode* _split_treapProxy(tnode *root,tnode *node,bool keynot-found)

- This is an proxy method for treap split, it is similar to typical binary search tree split

0.5.28 tnode* split_treap(tnode *root, float k))

- Searches for the key K in the treap with rooted root, if k found then invokes for proxy treap split
- if k not found in the treap, we insert key k with high priority, whihc will becomes the new root node and invoke the proxy split method.