Assignment 5 - Treap Report

Matthew Prost

CS 314H

11/13/2015

Table of Contents

Introduction/Goals

This project involved creating an `iterable` binary search tree that maintained the heap property, using randomly generated priority values. This is the first non-partner project in a while, so it will require the use of skills and reflections acquired during the pair programming assignments. Peer reviews will be an integral part of this assignment, as this is the only way to use others for feedback. In addition, testing others' code will be useful because it will help to come up with a very thorough test harness. There is a lot of freedom in this particular project, with only a few functions outlined in the interface. This will require thorough planning to achieve an optimal design, given all of this creative freedom. The goals during this project were to learn about and implement recursive algorithms, binary search trees, and heaps.

Solution Design

The `Treap` interface was provided, outlining the `lookup, insert, remove, split, join, toString,` and `iterator` methods. Three classes were created to help with the functionality of the Treap data structure: `TreapMap, TreapNode,` and `TreapIterator`. The `TreapMap` implemented the functionality of the `Treap`, using `TreapNode` and `TreapIterator` as helper classes. The `Treap` stores each of the values and is able to access, modify, and rearrange them. It stores them by using `TreapNodes`, which contain all the valuable information for each node in the `Treap` as well as its connections. With access to the `TreapNodes`, the `TreapMap` is able to traverse through the `Treap` and modify the nodes to change the `Treap's` structure. The `TreapIterator` uses the information stored in the `TreapMap` to present all data in order to the user. This is good because it helps the user to access the values of the `TreapMap`. While designing this, special consideration was given to modularity and flow.

`TreapMap` class

- `setRoot(TreapNode<K, V> r)`
- `getRoot()`
- `lookup(K key)`
- `insert(K key, V value)`
- `add(TreapNode<K, V> newNode)`
- `remove(K key)`
- `split(K key)`
- `join(Treap<K, V> t)`
- `toString()`
- `iterator()`

setRoot(TreapNode<K, V> r)

Helper method that changes the root value.

getRoot()

Returns the root value.

lookup(K key)

Retrieves the value associated with a key in the Treap. Sets pointer to the root. Searches through the Treap until the key is found. If the pointer finds a null node, then the key has no associated value in the Treap. If the key is larger than the present node, then it must be in the right subtree. If the key is smaller than the present node, then it must be in the left subtree.

insert(K key, V value)

Inserts a new node with key K and value V. Removes node with same key value if applicable. Creates a new node with the proper values. Finds the proper position for the `newNode` to be inserted. Incorporates the new node into the tree. Adds the children from the pointer to the new node and sets the `newNode's` parent. Finally, the node is rotated upwards until it fulfill the heap property.

add(TreapNode<K, V> newNode)

Adds a node into an existing tree without deleting any values. Unlike the insert function, this is used to add entire trees rather than just single nodes, it also allows duplicate values for implementation in the `split()` and `join()` methods. Finds the proper position for the `newNode` to be inserted. Incorporates the new node into the tree, storing its original children. It rotates the `newNode` upward to restore the heap property, then it recursively adds its children until they are `nullNodes`.

remove(K key)

Removes a key from the `TreapMap`. It sets a pointer to the root and utilizes the `lookup` function to find the correct spot for the expected key value. If the pointer finds a `nullNode`, then the key value is not present in the `TreapMap`. If it is present, then it gets rotates through the `TreapMap` until the node becomes a leaf. If the right child is null or the left child is greater than the right, it rotates the node to the right. If the left child is null or the right child is greater than or equal to the left, it rotates the node to the left. Finally, it removes the node from the graph, by having the node's parent point at a `nullNode`.

split(K key)

Splits a Treap into two Treaps where all nodes with a key < a certain key value are in the first Treap, and all nodes with a key >= a certain key value are in the second Treap. This creates a large node with maximum priority and the given key value. It adds it to the Treap, sending it to the top. It creates new TreapMaps for the values in the left and right subtrees, by setting the large node's children to roots. Finally it stores these two values in an array and returns them.

join(Treap<K, V> t)

This joins two `Treaps` together, with all keys in one `Treap` being smaller than all keys in the other `Treap`. It creates a node with maximum priority and key equal to the root's key. It adds this node, to the current `TreapMap`, moving all of the values to one side of the node. It then

adds the root of the other `TreapMap`, moving all values to the other side of the node. Finally, the node is removed, leaving a joined version of the two `Treaps`.

toString()

Represents the nodes as a `String` with pre-order traversal. Calls the `toString` of the root which recursively adds the subsequent nodes.

iterator()

Returns a `TreapIterator` to traverse the `TreapNodes` in order.

`TreapNode` class

- `setParent(TreapNode<K, V> p)`
- `getParent()`
- `setLeft(TreapNode<K, V> l)`
- `getLeft()`
- `setRight(TreapNode<K, V> r)`
- `getRight()`
- `rotateRight()`
- `rotateLeft()`
- `getKey()`
- `getValue()`
- `setPriority(int p)`
- `getPriority()`
- `toString()`
- `toString(int level, boolean l, boolean r)`

setParent(TreapNode<K, V> p)

Helper method that changes a `TreapNode's` parent.

getParent()

Returns the parent.

setLeft(TreapNode<K, V>)

Helper method that changes a `TreapNode's` left child.

getLeft()

Returns the left child.

setRight(TreapNode<K, V>)

Helper method that changes a `TreapNode's` right child.

getRight()

Returns the right child.

rotateRight()

This method rotates a node to the right within a tree. Stores grandchild of left child.
Moves left child to current node's position. Gives the parent the node's left child. Checks if the current node is a left child or right child. Moves `currentNode` into new position. Adds the stored grandchild as new left child.

rotateLeft()

This method rotates a node to the left within a tree. Stores grandchild of right child. Moves right child to current node's position. Gives the parent the node's right child. Checks if the current node is a left child or right child. Moves `currentNode` into new position. Adds the stored grandchild as new right child.

getKey()

Returns the node's key.

getValue()

Returns the value stored in the node.

setPriority(int p)

Helper method that modifies a node's priority.

getPriority()

Returns the node's priority.

toString()

Represents each node as a `String`.

toString(boolean l, boolean r, int level)

Recursively adds to the `String` representation of the `TreapMap`. It adds the appropriate number of tabs depending on the level of the tree that the node is on. Adds an L if it is a left child. Adds an R if it is a right child. Adds the priority, key, and value of the node. Recursively adds the left child, then recursively adds the right child.

`TreapIterator` class

- `hasNext()`
- `next()`

hasNext()

This method checks if there are more iterations in the `TreapMap`. It checks the `currentNode` pointer and the path `Stack`. There are still more iterations if the `currentNode` pointer is not directed at a `nullNode` or if there are some traced, but not returned, nodes in the `path`.

next()

This method returns the key value of the next `TreapNode`. In order to traverse the nodes in order, it must store the parents as it traverses to the left because there are other nodes that come before. Traverses to the left all the way, and adds traversed nodes to the path `Stack` until it encounters a `nullNode`. Sets pointer to the last traversed node. Returns the key of that node. Sets pointer to the current node's right child.

Scope and Quality of Solution

Assumptions/Scope

The solution provided works for large amounts data, as long as the inserted items contain unique comparable key values. For the `join()` operation, the program assumes that all of the values in one tree are less than all of the values in the other one. The `join()` and `split()` operations both assume that both `Treaps` are the same type.

Quality

This structure used a node system to store the values. This is good because rearranging the pointers is a very cheap operation, which is important especially because of how many rotations go on to maintain the heap property. Operations functioned in O(log n) time because the height of a large number n was O(log n) over a large set of data because the Treap rebalances itself based off of random priorities.

Software Test Methodology

Peer Review Process

The peer review process was a very helpful because of the feedback received from other classmates. The people that tested this code, responded with very specific and clear feedback for some very thorough tests. They also gave some of the insight they found in those results and

sometimes had suggestions for fixes. In addition, testing others' code will be useful because it helped to develop a very thorough test harness that considered many edge cases.

Black Box Testing

For this program, black box testing involved adding expected values to the Treaps and manually checking if the values resulted in valid Treaps, as represented by the toString() method. Because of information hiding, it was difficult to test this automatically, so it was done manually with small amounts of predictable values in order to see if the functions were operating as expected.

White Box Testing

White Box Testing was used to test the outcomes of different functions. A test harness was created using JUnit tests. These functions were called in individual tests, over a large number of tests to account for inherent randomness. By testing small segments of the project at a time, it was apparent to see where implementation errors were.

White Box testing was also used to test the error handling and edge cases in Treaps. Many of these involved operations on/ with empty, single-node, or null Treaps and values.

Conclusion

With the large freedom to implement this interface, it was very important to do thorough planning.nOne thing that I discovered how useful it is to record your thoughts when programming alone. This helps you analyze your own ideas before implementing them without having to talk to a partner. By being very methodical, the flow of the code was developed to be very cohesive with a very complimentary structure. Testing was also a very large part of the success of this project. Testing during development was very helpful in fixing bugs early on and not compounding their results over time.