Nicolas Mavromatis

CSPB 2270 Summer 2020 Final Project Proposal

**Set Operations Using a Treap**

**Overview**

This project implements a treap in order to perform set operations quickly, which may be particularly useful for sets of large numbers, and in any real world scenario in which sets are utilized. The treap data structure was chosen because it was previously unused, and is an efficient data structure for storing and processing set data in order to calculate unions and intersections. This data structures was chosen rather than other self-balancing trees like red black trees because treaps may utilize less memory and are more straightforward to implement. Crucially, other data structures can have drastically different shapes for different insertion orders, but this is not the case for treaps. In addition to being used for set operations, they may be used for security such that a user cannot gather information through insertion history and order.

A treap is a combination of a max heap and binary search tree, where each node has a main key that is greater than the left child’s key, but less than the right child’s key (a BST property). It also assigns each node a random priority, organizing the structure so that each node has a greater priority than its children (a heap property). Because the priority of each node is randomly selected, the tree has a high probability of being balanced, regardless of the order of operations performed on the tree. The height is usually log2N and the thus the average time complexity of search, insert, and delete are O(LogN). The worst case for these operations are O(N) but the random selection of each node’s priority means this is very unlikely to occur.

**Treap Class: Data Struct**

* Node: Contains- Char C, int Priority, shared\_ptr<Node> left, shared\_ptr<node> right. C is the main key.

**Treap Class: Private Class Functions**

* int randPriority()

This function returns a random priority, likely by using rand(), which is between 0-100.

* shared\_ptr<Node> initNode(char c)

This creates and returns a shared pointer to a node, initializes it with char c, and a random priority.

* void insert(shared\_ptr<Node>& ptr,  char c)

This will be a recursive function to insert a new node in a treap. It will call the above functions, then performs rotations. A reference to a shared pointer is passed.

* void remove(shared\_ptr<Node>& ptr,  char c)

This will be a recursive function to remove a node containing char c. It then rotates nodes to maintain treap properties. It will probably use search() as a helper function.

* shared\_ptr<Node> search(shared\_ptr<Node> root, char c)

This returns the node containing char c, or NULL if not found.

* shared\_ptr<Node> rotateLeft(shared\_ptr<node> root)

This function rotates the subtree rooted at root to the left, and returns the new root.

* shared\_ptr<Node> rotateRight(shared\_ptr<node> root)

This function rotates the subtree rooted at root to the right, and returns the new root.

* vector<char> findUnion(shared\_ptr<Node> A, shared\_ptr<Node> B)

This returns the union of sets A and B as a vector.

* vector<char> findIntersecion(shared\_ptr<Node> A, shared\_ptr<Node> B)

This returns the intersection of sets A and B as a vector.

* vector<char> findDifference(shared\_ptr<Node> A, shared\_ptr<Node> B);

This returns the difference of sets A and B as a vector.

* void print(shared\_ptr<Node> root)

This prints the tree node’s chars inorder.

Possibly other helper functions will be necessary. I did not list public functions here to save space, because most functions above are called by an overloaded public version on the private shared\_ptr<Node>Root variable, so that user does not manage shared\_ptrs. Here is an example:

* void insert(char c) //public version

   { insert(Root, c)}; //calls private version

**Tests:** I will create a variety of different trees with different values and insertion orders, and thoroughly test all functions on each tree. The union, intersection, and set functions are the main points of the project, but in order to function, all other functions must also work properly. A proper end state is where all functions and helper functions have been thoroughly tested, including for uncommon or edge cases like empty treaps.

**Sources:** <https://stackoverflow.com/questions/16009361/when-to-use-a-treap>

<https://medium.com/carpanese/a-visual-introduction-to-treap-data-structure-part-1-6196d6cc12ee>