Generic Programming Project Report

Project Title: Implementation of Treaps
Project ID: 5

Team Details:

Darshan D - PES1201801456
Karan Kumar G - PES1201801883
Mayur P L - PES1201801439

Abstract:

Treap is a data structure that stores pairs (X, Y) in a binary tree in such a way that it is a binary search tree by X and a binary heap by Y. Assuming that all X and all Y are different, we can see that if some node of the tree contains values (X_0, Y_0), all nodes in the left subtree have $X<X_0$, all nodes in the right subtree have $X>X_0$, and all nodes in both left and right subtrees have $Y<Y_0$.

In such an implementation X values are the keys (and at same time the values stored in the treap), and Y values are called priorities. Priorities allow us to uniquely specify the tree that will be constructed. The priorities are random. Hence this is also called a randomized binary search tree. The main idea is to use Randomization and Binary Heap property to maintain balance with high probability.

Some operations supported by a treap:

- Build(X_1, X_2, …, X_n) in O(N)
- Insert(X, Y) in O(log N)
- Search(X), Delete(X) in O(log N)
- Set operations like Union(T_1, T_2), Intersect(T_1, T_2) in O(M log(N/M))

Implementation Details:

We have implemented treaps in C++ and this file will be included as a header file that the clients can use in their programs.

Each node of the treap has been represented as a generic class including the following attributes:

- Pair of key and priority - the key is of generic type and priority is an integer that is assigned randomly

- Pointer to left child of the node
- Pointer to right child of the node
- An implementation field to mark a node as a duplicate

The Treap_node class has been made canonical by providing a constructor, a copy constructor, copy assignment operator and a destructor.

Treap has been represented as a generic class including the following attributes:

- Pointer to the root of the treap

- An implementation field to decide the way priorities are defined. By default, the
  priorities are assigned randomly

- An implementation field to store all the priorities defined already. This is used to ensure that priorities are not repeated for different nodes

The Treap class has been made canonical by providing the following:

- **Constructor:** This initializes the data members when the object is created

- **Copy Constructor:** This calls copy_treap function of the treap node class where copying of the nodes is done following the preorder traversal of the treap

- **Copy Assignment Operator:** First the memory assigned to the treap on the left hand side of the assignment operator is deallocated by calling the destructor, if it is not self assignment. Then the copy_treap function of the treap node class is called where copying of the nodes is done following the preorder traversal of the treap

- **Destructor:** This calls the delete_treap function which releases the allocated resources following the postorder traversal of the treap.

A special constructor called the build constructor has been provided that takes in a pair of iterators pointing to a collection of data items and creates a Treap consisting of these data items in linear time.

Operations that can be performed on treaps:

- **Insert Node:**

  We can insert a new node into the treap with a key value.

The insert function of the treap class is a wrapper function that calls the insert_node function of the treap node class.
Here, insertion of a new node is done as it is done in a normal binary search tree in a recursive manner. But to ensure that the max heap property with respect to the priorities is maintained, rotations need to be performed. Thus left and rotations are performed appropriately.

The insert node operation has a time complexity of O(logN).

- **Delete Node:**

  A delete function was implemented to delete a node from the treap given the key value.

  The delete function of the treap class is again a wrapper function that calls the delete_node function of the treap node class.

  In the delete_node function, we first search for the key following the Binary Search Tree property. If we find the key in the treap, we then check the number of children of the node.

  - If the current node is a leaf we can free it directly.

  - If the current node is a node with one child, we replace the current node with the child and delete the child node from the treap.

  - If the current node is a node with both children, we rotate the nodes appropriately while maintaining the max heap property such that the current node becomes one of the children and recursively call the delete_node function on the current node once it has been moved down the tree.

  The delete node operation has a time complexity of O(logN).

- **Split:**

  The split function separates the given tree into two sub trees based on the given key. The given key need not be present in the tree.

  The split subtrees are such that one of them will have all nodes lesser than the given key value and the other subtree will have all nodes greater than the given key value.

This operation has O(logN) complexity and is implemented by making use of the previously defined insert function. We insert the given key as a node in the tree such that the key will have the highest priority and hence will undergo rotations to become the root.

Now that the given key is the root, if the given key was not already a part of the original tree, we split the tree by assigning the two subtrees as the new split trees. If the given key was present in the tree, we split the tree in the same aforementioned way but we ensure that the root is not deleted and is assigned to the right subtree.

- **Merge:**

  Merge combines two given trees while maintaining BST and heap property and returns the new tree. This operation has O(logN) complexity.

  It is implemented under the assumption that elements of the left subtree are all lesser than the elements of the right subtree. Thus, since the BST property is already satisfied by the pre-condition, we only need to ensure that the priorities of the merged tree follow the max-heap property. We use recursion to merge two trees.

  If both the trees are empty, an empty tree is returned as the merged tree. If either of the trees is empty the other tree is returned as the merged tree. These form the base condition of the recursion.

  During recursion, If the root of the left tree has a higher priority than that of the right subtree, we make a recursive merge call on the right node of the left subtree and the root of the right subtree and the result would be assigned to the right node of the left subtree which would then be assigned to merged tree pointer.

  Similarly, If the root of the right tree has a higher priority than that of the left subtree, we make a recursive merge call on the left node of the right subtree and the root of the left subtree and the result would be assigned to the left node of the right subtree which would then be assigned to the merged tree pointer.

  Merging would result in the deletion of the sub trees sent as parameters to the function and would return as result, only the merged tree.

- **Union:**

Union functionality is equivalent to set union operation and has a complexity of O(Mlog(N/M)). Union does not change the trees that were sent as parameters and inturn returns a new tree containing the union of the the two given trees,

The function follows a divide and conquer approach. To maintain heap order, we ensure that the root of the resultant tree, r, is the root of the subtrees with the largest priority. Now if the key of this root is k, to maintain the BST order, we split the other sub tree whose root had a lesser priority into two subtrees which are ordered, that is, the left split will have all values < k and the right split will have all values >= k.

We then recursively find the union of the left child of r and the left split of the other tree, and then the union of the right child of r and the right split of the other tree. The resultant of the two calls become the left and right subtrees of r, respectively.

Union would not change the trees sent as parameters and would return as result, the union of the two parameters whose nodes are all copies of the nodes in the two subtrees.

- **Intersection:**

  Intersection functionality is equivalent to set intersection operation and has a complexity of O(Mlog(N/M)). Intersection does not change the trees that were sent as parameters and inturn returns a new tree containing the intersection of the given two trees.

  As with union, intersection starts by splitting the treap with the smaller priority root by k, the key of the root with the greater priority. It then finds the intersection of the two left subtrees, which have keys less than k, and the intersection of the two right subtrees, which have keys greater than k. If k appeared in both trees then these results become the left and right children of the root used to split. Otherwise it returns the join of the two recursive call results.

- **Difference:**

  Difference functionality is equivalent to set difference operation and has a complexity of O(Mlog(N/M)). Difference does not change the trees that were sent as parameters and inturn returns a new tree containing the difference of the given two trees. The new tree contains the elements of the first tree that are not in the second tree.

To find the difference of two treaps T1 and T2, difference splits the treap with the smaller priority root by k, the key of the root of the other treap. Then it finds the difference of the two left subtrees, which have keys less than k, and the difference of the two right subtrees, which have keys greater than k. Because difference is not symmetric, difference considers two cases: when T2 is the subtrahend (the set specifying what should be removed) and when T2 is not, as specified by the boolean rt_subt. If T2 is the subtrahend and it does not contain k, then it sets the left and right children of the root of T1 to the results of the recursive calls and returns this root. Otherwise it returns the join of the results of the recursive calls.

- **Traversals:**

  We have provided functionality to allow for the preorder, postorder and inorder traversal of the treaps.

  In Inorder traversal, we traverse the left subtree, then visit the root and then finally traverse the right subtree.

  In Preorder traversal, we visit the root, then traverse the left subtree and then finally traverse the right subtree.

  In Postorder traversal, we traverse the left subtree, then traverse the right subtree and then finally we visit the root.

- **Iterators:**

  Iterator has been implemented as a nested class within the treap class. It has the following attributes:
  - Pointer to a Treap_node
  - An implementation field which is a pointer pointing to the root of the treap

  A **bidirectional iterator** has been implemented by supporting the following:

  - **Pre increment operator:** The pre increment operator has been overloaded to work for the treap. It updates the iterator and points it to the inorder successor of the current node.

  - **Post increment operator:** A temporary iterator is created and initialized with *this. The pre increment operator is called on *this and the temporary iterator is returned.

- **Pre decrement operator:** The pre decrement operator has been overloaded to work for the treap. It updates the iterator and points it to the inorder predecessor of the current node.

- **Post decrement operator:** A temporary iterator is created and initialized with *this. The pre decrement operator is called on *this and the temporary iterator is returned.

- **Dereferencing (operator *):** The operator * has been overloaded to support dereferencing the iterator for rvalue usage. It calls the overloaded operator * of the treap_node class which returns the key value of the node as an rvalue.

- **Equality operator:** The equality operator has been overloaded to support comparison for equality between two iterators. This calls the overloaded equality operator of the treap_node class which compares if the key values of the two nodes are the same and returns the appropriate result.

- **Inequality operator:** The inequality operator has been overloaded to support comparison for inequality between two iterators. This calls the equality operator of the iterator class and inverts the result.

- It has been made sure that the iterator holds on to the location on dereferencing it.

Iterator begin: Function begin has been provided in the treap class encapsulating the iterator class. This function returns an iterator pointing to the first node of the treap based on the inorder traversal. Thus it will point to the node with the smallest key value.

Iterator end: Function end has been provided in the treap class encapsulating the iterator class. This returns an Iterator initialized with nullptr as the end iterator of a container is supposed to point to an invalid location.

- **Member Algorithms:**

  A few member algorithms have been provided as follows:

  - **Member find:**

    If the generic find is called on treap, the complexity of the find operation would be linear [O(n)] as it would use the increment operator which

would make use of the inorder successor at each point.

Member find has been implemented such that the complexity of the find operation becomes log(n).

The find operation uses the binary search tree property to search for a node in log(n) time.

- **Member replace:**

  If the generic replace is called on treap, once the key of a node is replaced with a new key, the binary search tree property and the max heap property of the treap may be broken.

  The member replace replaces the key with the new key value mentioned by doing the following.

  It first deletes the node containing the old key value. It then inserts a new node with the new key value. The insert function takes care of all rotations to ensure that the binary search tree property and the max heap property of the treap are maintained.

Details about running the software:

- The implementation file consisting of all the implementation code and functions with respect to treaps has been provided as a header file that the client can include in their programs. The way to include it is as follows:
  #include “treap.h”

- Once this is included, clients can create treaps and call functions to perform different operations as explained above or in the **readme** file associated with this document using the provided interface.

- The client file is then compiled using the g++ utility.
  g++ client.cpp -o exec

- The generated executable is then loaded and executed.
  ./exec