ECE250: Lab Project 3

Due Date: Monday, October 29, 2018 – 11:00PM

1. Project Description

A *mergeable heap* is an ADT that stores an element from an ordered set, and supports the operations Insert, Min, and Extract-Min (just like Min Priority Queues). Mergeable heaps support an additional operation named "Merging Two Heaps". Two implementations of the mergeable heap are the *leftist tree* and the *binomial heap*. We implement leftist heap in this project. Essentially, a leftist heap is very similar to an AVL tree but rather than height it uses the *null-path length* of a node; that is, the minimum path length from the node to a descendent that is not a full node. The name leftist comes from the fact that, in leftist heap, the left subtree of each node is usually taller than the right subtree. You can visually see the steps involved in each operation of leftist heap on the course website.

2. How to Test Your Program

We use drivers and tester classes for automated marking, and provide them for you to use while you build your solution. We also provide you with basic test cases, which can serve as a sample for you to create more comprehensive test cases. You can find the testing files on the course website.

3. How to Submit Your Program

Once you have completed your solution, and tested it comprehensively on ECELinux server, you need to build a compressed file, in tar.gz format, which should contain the files:

- Leftist_node.h
- Leftist_heap.h

Build your tar file using the UNIX tar command as given below:

- *tar –cvzf xxxxxxxx_pn.tar.gz Leftist_node.h Leftist_heap.h*

where *xxxxxxxx* is your **UW user id (ie. jsmith)**, and *n* is the project number which is 3 for this project. All characters in the file name must be lowercase. Submit your tar.gz file using LEARN, in the drop box corresponding to this project.

4. Class Specifications

In this project you will implement two classes, *Leftist_node* and *Leftist_heap*. The *Leftist_heap* class contains the root node from where every other node in the tree can be reached. The *Leftist_node* class represents each node in the tree.

4.1 Leftist_heap.h

You are expected to implement or modify the functions that are listed. The run time of each member function is specified in parentheses at the end of the function description. For the most part, the function simply calls the corresponding function on the root. The exceptions are size and empty (there is no requirement for each node in the heap to know the size of its descendant sub-tree), and pop is converted into a push of one sub-tree into the other. The clear function also requires a few assignments.

Member Variables
The class has two member variables:

- *Leftist_node<Type> *root_node* - A pointer to the root node.
- *int heap_size* - Number of elements in the heap.

Accessors
This class has five accessors:

- *bool empty() const* - Returns true if the heap is empty, false otherwise. (**O**(1))
- *int size() const* - Returns the number of nodes in the heap. (**O**(1))
- *int null_path_length() const* - Returns the null-path length of the root node. (**O**(1))
- *int count(const Type &obj) const* - Return the number of instances of the argument in the heap. (**O**(n))
- *Type top() const* - Returns the element at the top of the heap. If the tree is empty, this function throws an underflow exception. (**O**(1))

Mutators
This class has three mutators that may need to be modified:

- *void push(const Type &):* (known as *insert* in ADT definition) Insert the new element into the heap by creating a new leftist node and calling push on the root node using root_node as a second argument. Increment the heap size. (**O**(ln(n))).
- *Type pop():* (known as *extract-min* in ADT definition) Pop the least element in the heap and delete its node (extracts min from the heap). If the tree is empty, this function throws an underflow exception. Otherwise, the left sub-tree of the root node is made the root node and the right-sub tree of the original root node is pushed into the new root node. Return the element in the popped node and decrement the heap size. (**O**(ln(n)))
- *void clear():* Call clear on the root node and reset the root node and heap size. (**O**(n)

4.2 Leftist_node.h
A leftist node is a node within a leftist heap. You are expected to implement or modify the member functions that are listed.

Member Variables
This class has three member variables:

- *Leftist_node* left_tree* – A pointer to the left subtree
- *Leftist_node* right_tree* - A pointer to the right subtree
- *int heap_null_path_length* - The null path length of this node. The null-path length of a tree is defined as the shortest path to a node that has an empty sub-tree. This can be calculated as follows: i) an empty node has a null-path length of -1, otherwise, ii) the null-path length of a node is one plus the minimum of the null-path lengths of the two children. Note that a consequence of the second point is that a node with no children (a leaf node) or a node with exactly one child has a null-path length of 0.

Accessors

This class has five accessors that may need to be modified where *n* is the number of nodes in this sub-tree.

- *Type retrieve() const* - Returns the element stored in this node. (**O**(1))
- *Leftist_node *left() const* - Returns the address of the left sub-tree. (**O**(1))
- *Leftist_node *right() const* - Returns the address of the right sub-tree. (**O**(1))
- *int null_path_length() const* - Returns the member variable null-path length unless this is the null pointer, in which case, return -1. (**O**(1))
- *int count(const Type &obj) const* - Returns the number of instances of the argument in this sub-tree. (**O**(n))

Mutators

This class has two mutators that may need to be modified:

- *void push(Leftist_node *new_heap, Leftist_node *&ptr_to_this):* If the new heap is null, return. Otherwise, insert the new_heap into this heap: i) if this is null, set the pointer to this to be the new heap and return, ii) if the current node is storing a value less-than-or-equal-to the value stored in the root of the new heap, push the new node onto the right sub-tree with right_tree. Now, update the null-path length and if the left-sub-tree has a smaller null-path length than the right sub-tree, swap the two sub-trees, iii) otherwise, set the pointer to this to be the new heap and push this node into the new heap (with an appropriate second argument). (**O**(ln(n)))
- *void clear():* If this is nullptr, return; otherwise, call clear on the left sub-tree, then on the right, and finally delete this. (**O**(n))