University of Michigan – Dearborn
Department of Computer and Information Science
CIS 350: Data Structures and Algorithm Analysis
(Winter 2024)
Instructor: Luis E. Ortiz

Program 2: Implementing Basic ADT Functions and Operations for Linked-Structure Heap-based Priority Queues within the Context of a Simple Stock Market Simulator

Overview

- In the *source code* file provided: implement the missing subroutines for basic ADT and operations for a heap-based priority queue used for a simple stock market trading simulator
- In the *memo* file provided: provide a worst-case running time and space, using big-Oh notation as function of the number of nodes n in the heap, which roughly characterizes the input size, of some of the methods used in the implementation

Stock-Market Simulator Implementation: Using Priority Queues

The *stock market* simulator consists of two *limit-order books*, one for *buy orders* and another for *sell orders*. Each *order book* is implemented using a *priority queue*; the priority queue ADT is implemented using a heap (see `Heap` class).

Let $\texttt{PQ}_{\text{buy}}$ and $\texttt{PQ}_{\text{sell}}$ denote the priority queues corresponding to the *buy* and *sell* limit-order books, respectively. Each *element* $e = (k, v)$ of a priority queue corresponds to an (stock transaction) *order* and is composed of a key k and a value v (see `Elem` class/struct definition). Each *key* $k = (p, t)$ is composed of a *price* p and a *time stamp* t (see `Key` class/struct definition). Each value $v = (s, i)$ is composed of a number of shares s and a trader's id i (see `Value` class/struct definition).

The priority of elements in a priority queue is defined as follows. Given any pair of elements $e_1 = (k_1, v_1)$ and $e_2 = (k_2, v_2)$, where $k_1 = (p_1, t_1)$ and $k_2 = (p_2, t_2)$, we have $e_1 < e_2$, if and only if $(p_1 < p_2)$, or $((p_1 = p_2)$ and $t_1 < t_2)$ (for C++, see overloaded operators $<$ and $>$ for `Elem` struct objects, and overloaded operator $<$ for `Key` struct objects; for Java, see functions `isLessThan` and `isGreaterThan` for `Elem` class objects, and function `isLessThan` for `Key` class objects). The prices for the keys of the elements in $\texttt{PQ}_{\text{buy}}$ are negative, while those in $\texttt{PQ}_{\text{sell}}$ are positive; that is, if $e = (k, v)$ is an element with key $k = (-p, i)$ in $\texttt{PQ}_{\text{buy}}$, for some price $p > 0$, while if e is in $\texttt{PQ}_{\text{sell}}$ instead, then $k = (p', i')$ for some price $p' > 0$. Prices p are represented using doubles, while the number of shares s and the trader's id i are represented using integers. Time stamps t are represented using integers and assigned using a simple integer (global) counter.

For your implementation, the only thing you need to know about the element comparison operator is how to invoke it.

- For C++: Let `w` and `z` be variables declared as `Node*`. If `w` has been assigned a pointer to a non-`NULL` node in the heap/tree that is not the root, so that if we make the assignment

  ```
  z = w->parent;
  ```

  we know that `z` will also be non-`NULL` pointer, then the following conditional expression

  ```
  *(z->elem) < *(w->elem)
  ```

  will evaluate to `true` if the element of `z` has higher priority (i.e., lower key) than that of `w`; and to `false` otherwise. This is similar/analogous for the $>$ operator.

- For Java: Let `w` and `z` be variables declared as `Node`. If `w` has been set to a non-`null` node in the heap/tree that is not the root, so that if we make the assignment

  ```
  z = w.parent;
  ```

  we know that `z` will also be non-`null`, then the following conditional

  ```
  isLessThan(z.elem, w.elem)
  ```

  will evaluate to `true` if the element of `z` has lower priority than that of `w`; and to `false` otherwise. This is similar for the `isGreaterThan` operator.

- For Python: Let `w` and `z` be variables declared as `Node`. If `w` has been assigned to a non-`NULL` node in the heap/tree that is not the root, so that if we make the assignment

  ```
  z = w.parent
  ```

  we know that `z` will also be non-`NULL`, then the following conditional expression

  ```
  z.elem < w.elem
  ```

  will evaluate to `True` if the element of `z` has higher priority (i.e., lower key) than that of `w`; and to `False` otherwise. This is similar/analogous for the $>$ operator.

As for how the simulator works, this is all you need to know. Every time a buy or sell limit order is submitted, it is inserted in the corresponding priority queue. Trading occurs after an order is inserted if there is a matching order on the opposite priority queue. So after every insertion, the simulator checks the `min` elements (i.e., the elements with highest priority) in each queue, $\texttt{PQ}_{\text{buy}}$ and $\texttt{PQ}_{\text{sell}}$. Let e_{buy} and e_{sell} be the `min` elements in the buy and sell limit-order books $\texttt{PQ}_{\text{buy}}$ and $\texttt{PQ}_{\text{sell}}$, respectively. Suppose that e_{buy} has key $(-p_{\text{buy}}, t_{\text{buy}})$, for some price $p_{\text{buy}} > 0$, and value $(s_{\text{buy}}, i_{\text{buy}})$. Similarly, suppose that e_{sell} has key $(p_{\text{sell}}, t_{\text{sell}})$, for some price $p_{\text{sell}} > 0$, and value $(s_{\text{sell}}, i_{\text{sell}})$. A trade is executed if $p_{\text{buy}} \geq p_{\text{sell}}$. The simulator performs the following steps to execute a trade. First, each of the `min` elements of $\texttt{PQ}_{\text{buy}}$ and $\texttt{PQ}_{\text{sell}}$ are removed. If $s_{\text{buy}} < s_{\text{sell}}$, then a new element with key $(p_{\text{sell}}, t_{\text{sell}})$ and value $(s_{\text{sell}} - s_{\text{buy}}, i_{\text{sell}})$ is inserted into $\texttt{PQ}_{\text{sell}}$; while if $s_{\text{buy}} > s_{\text{sell}}$, then a new element with key $(-p_{\text{buy}}, t_{\text{buy}})$ and value $(s_{\text{buy}} - s_{\text{sell}}, i_{\text{buy}})$ is inserted into $\texttt{PQ}_{\text{buy}}$. These steps are repeated until no more trades could be executed, and before any new additional order is processed.

Source Code Details

We will provide the source code for the priority queue implementation in three languages: C++, Java, and Python. This program constructs the relevant data structures for your algorithm implementations and handles input/output. However, the source will be missing the implementation for some important functions.

We expect you to complete the following functions in the `BT (Binary Tree)` class: The functions take a Node `w` in the BT as input.

- `lastLeftDescendant`: Return the leftmost Node of the BT rooted at `w`.

- `firstRightAncestor`: Return the first ancestor `x` of `w` (inclusive) found in the path to the root such that `w` is in the right subtree of `x`. Return NULL if no such ancestor exists.

Also complete the following functions in the `CompleteBT` class:

- `getParentOfNewLastNode`: Return the node in the complete BT where any new node inserted would be placed.
- `getNewLastNode`: Return the node in the BT that would become the last node of the complete BT should the current last node be removed
- `add`: Insert and return a new Node `x` containing the input element `e` into the complete BT in the correct location. Also, `x` becomes the new last node, and if `x` is the only Node, `x` becomes the root. The size of the tree `n` increases by 1.
- `remove`: Removes the last node from the complete BT and returns it. Update the last node to the appropriate location (or null if the tree becomes empty). The size of the tree `n` decreases by 1.

Finally, complete the following functions in the `Heap` class:

- `insert`: Add the input element e to the Heap. The ”heap-order property” is maintained.
- `min`: Returns the minimum element of the heap.
- `removeMin`: The minimum element of the heap is removed by swapping the last node with the root node and then deleting the last node. The new last node is set, and the ”heap-order property” is maintained.
- `upHeapBubbling`: Perform a series of up-heap bubbling operations by starting at the last node and then swapping elements as needed. The heap-order property should be maintained after all bubbling operations are completed.
- `downHeapBubbling`: Perform a series of down-heap bubbling operations by starting at the root and then swapping elements as needed. The heap-order property should be maintained after all bubbling operations are completed.

You may find the existing `swapElem`, `makeChild`, `removeNode`, `firstLeftAncestor`, `lastRightDescendant`, and `minChild` functions useful for your implementation.

NOTE: Do not forget to write your name in the source code file.

Input File Format and Sample File

Use a file named `input.txt`, which is assumed to be located in the same subdirectory where the program is executed, to debug, test, and evaluate your implementation.

Input File Format

One line for every basic stock-market trading operation (buy or sell), or to print out the state of some of the components of the simulator (print):

- `<trans> <num_shares> <price> <trader_id>`

  Insert a new `<trans>` order, where `<trans> = buy | sell`, for `<num_shares>` (as an integer) at `<price>` (as a double) from `<trader_id>` (as an integer) into the into the respective (buy or sell) limit-order book of the stock-market simulator.

- `print <type>`

  Display the current state of a component of the simulator, depending on the value of `<type>`. For the purpose of testing, debugging, and evaluating the task you are asked to performed (i.e., implementing the basic ADT functions and operations for the heap-based priority queue needed for the instantiation of the simple stock-market simulator), the only relevant values for `<type>` are `buy` or `sell` (see input file and output example). Each heap is displayed using a simplified string representation of the underlying tree:

  ```
            right_subtree [<key>,<value>]
  root [<key>,<value>]
            left_subtree [<key>,<value>]
  ```

  Here, each key is displayed as a pair `(<price>,<time_stamp>)`, where `<price>` is displayed as a positive or negative number (with 2 decimal digits), depending on the limit-order book, sell or buy, respectively, and `<time_stamp>` is displayed as a positive integer. Each value is displayed as a pair `(<num_shares>,<trader_id>)`, where both `<num_shares>` is displayed as a positive integer and `<trader_id>` is displayed as an integer.

Sample Input File `input.txt` and Sample Output

Refer to the `input.txt` and `sample_out.txt` files which contains an example of a series of transactions in the simulated stock market as well as the expected result of those transactions.

Submission Details

Code submissions will be handled using autograder.io. To submit your code, follow these steps:

1. Go to https://autograder.io and log in using your uniqname/password.
2. Choose the "CIS 350" course and the appropriate project.
3. Drag and drop or browse to your source code file.
    - Only upload **one** of Main.cpp, Main.py, or Main.java
4. Wait for the feedback results. Typically this takes about 30 seconds or less.

There are a couple important things to note about the autograder service.

- You are restricted to 3 submissions with feedback per day (resets at midnight). You may submit more times in one day, but you will not receive feedback about test cases for submissions that day after the third.
- You will always receive feedback on whether your submission compiles or not.
- The "final graded submission" will be your most recent one.
- You are allowed to submit code late, but you may lose points.
- There may be **hidden** test cases that you will be graded on.
- *C++ only*: C++ is not a cross platform language, so it is possible that code which compiles in one machine does not compile on another. The autograder uses g++ to compile C++. It is ultimately **your** responsibility to ensure that the autograder can compile your code.

How the program will be graded

Memo

What	**pts**
Name	1
Time/Space analysis	40

Source Code

What	**pts**
Name	1
Style	8
Functionality	70