NIU CSCI 340 Grade-o-Matic Assignment

Binary Trees - Heaps/Priority Queue

Introduction

The purpose of this assignment is to have you implement the heap data structure. You will also implement a priority queue class using that heap implementation, which will work like the STL priority queue

Your Task

You are responsible for implementing all of the following functions:

- For heap functionality (in `heap.h`):
    - `heap_left(node)`
    - `heap_right(node)`
    - `heap_parent(node)`
    - `heap_preorder`
    - `heap_inorder`
    - `heap_postorder`
    - `heap_levelorder`
    - `is_heap`
    - `heap_bubble_up`
    - `heap_bubble_down`
    - `heap_insert`
    - `heap_extract`
    - `heapify_in_place_up`
    - `heapify_in_place_down`
    - `heap_sort`
- For the `heap_priority_queue` class (in `heap_priority_queue.h`):
    - Constructor `heap_priority_queue(ITERATOR, ITERATOR)`
    - `top()`
    - `empty()`
    - `size()`
    - `push()`
    - `pop()`

Heaps

As a quick review, a heap is a type of tree that is useful for quickly finding the "best" thing according to some criterion, called the heap criterion.

The heap we are using is a complete binary tree, filled from the left. Instead of using the pointer-based nodes that we had used for other trees, we will be storing the heap into an array by mapping the nodes in the tree to an element in the array based on level order.

Placing items in the heap involves the "King of the Hill" method discussed in class. For minheaps, smaller values must be above lower values. For maxheaps, the greater values must be on top.

When comparison functions are used in the functions below, they are used to compare two values (parent value is the left hand side, child value is the right hand side) and return `true` if the parent value and the child value conform to the heap criterion for the heap we are using.

Thus, less than, `parent < child` would be used for minheaps, and greater than, `parent > child` would be used for maxheaps.

The Heap Functionality (in `heap.h`):

`heap_left(node)`

- `node` - index of the node whose left child's index we'd like

This function returns the index in the heap that would contain the left child of node `node`.

heap_right(node)

- node - index of the node whose right child's index we'd like

This function returns the index in the heap that would contain the right child of node node.

heap_parent(node)

- node - index of the node whose right child's index we'd like

This function returns the index in the heap that would contain the parent of node node, unless node is already the root, in which case it returns 0.

heap_preorder(heapdata, heapnodes, nodes, fn)

- heapdata - The array-like object containing the heap data
- heapnodes - The number of nodes in the heap
- node - Index of the node in the heap that is the root of the subtree to traverse.
- fn - The function to call when visiting.

Traverses, preorder, the subtree of the binary tree represented by the heap array that has node as its root, calling fn on each element. Returns nothing.

heap_inorder(heapdata, heapnodes, nodes, fn)

- heapdata - The array-like object containing the heap data
- heapnodes - The number of nodes in the heap
- node - Index of the node in the heap that is the root of the subtree to traverse.
- fn - The function to call when visiting.

Traverses, inorder, the subtree of the binary tree represented by the heap array that has node as its root, calling fn on each element. Returns nothing.

heap_postorder(heapdata, heapnodes, nodes, fn)

- heapdata - The array-like object containing the heap data
- heapnodes - The number of nodes in the heap
- node - Index of the node in the heap that is the root of the subtree to traverse.
- fn - The function to call when visiting.

Traverses, postorder, the subtree of the binary tree represented by the heap array that has node as its root, calling fn on each element. Returns nothing.

heap_levelorder(heapdata, heapnodes, fn)

- heapdata - The array-like object containing the heap data
- heapnodes - The number of nodes in the heap
- fn - The function to call when visiting.

Visits each of the nodes in the heap in level order, calling fn to visit. Returns nothing.

is_heap(heapdata, nodes, compare)

- heapdata - The array-like object containing the heap data
- nodes - The number of nodes in the heap

Returns true if, and only if, the heap condition remains true for every node in the given heap. Depending on your implementation, you might get a different answer when there are nodes in a heap with the same key, but duplicates will not be tested for grading, so you don't need to worry about that.

heap_bubble_up(begin, nodes, start, compare)

- begin - Beginning of random access iterator range containing the tree data.
- end - Random access iterator one past the last element in the iterator range.
- start - Index of the node to start bubbling up from.

- `compare` - The comparison function to be used when deciding whether to swap.

This performs the heap bubble up procedure on the given heap starting from the element. `start`, as would be needed, for example, when inserting a new node into the heap. Returns the number of swaps that had to be performed.

`heap_bubble_down(begin, nodes, start, compare)`

- `begin` - Beginning of random access iterator range containing the tree data.
- `end` - Random access iterator one past the last element in the iterator range.
- `start` - Index of the node to start bubbling down from.
- `compare` - The comparison function to be used when deciding whether to swap.

This performs the heap bubble down procedure on the given heap starting from the element. `start`, as would be needed, for example, when moving the number that was swapped with the root during the root extraction procedure.

`heap_insert(heapdata, nodes, key, compare)`

- `heapdata` - The STL container, supporting random access, that contains the data for the heap.
- `nodes` - A **reference** to the `size_t` passed in containing the number of nodes in the heap. You will need to report the changes back to the caller by changing the value stored in this.
- `key` - The key to insert into the heap.
- `compare` - The comparison function to use.

Performs the heap insertion procedure, inserting `key` into the heap of `nodes` nodes stored in `heapdata`. You can assume that the data is a heap before you start, and must ensure that the data is a heap again by the time you finish. Returns nothing. Make sure to change the `size_t` value referenced by nodes in order to report that the additional node was added.

If there is not enough room in the container to add another key, you are responsible for resizing the container to make that room.

`heap_extract(heapdata, nodes, compare)`

- `heapdata` - The STL container, supporting random access, that contains the data for the heap.
- `nodes` - A **reference** to the `size_t` passed in containing the number of nodes in the heap. You will need to report the changes back to the caller by changing the value stored in this.
- `compare` - The comparison function to use.

Removes the key stored at the root of the heap and returns its value. You can assume that the data contains a heap when you start, and you are responsible for ensuring that the data is still a heap once you've removed the node.

`heapify_in_place_up(begin, end, compare)`

- `begin` - Random access iterator representing the beginning of the data to heapify.
- `end` - Random access iterator one past the end of the data that will be heapified.
- `compare` - The comparison function to be used when deciding whether swaps must occur.

This function takes the data in the given, random access iterator range, which may not yet be a heap, and heapifies it *in place*, by bubbling up nodes as new values are inserted. You are not allowed to use another array/`vector`/etc. to store things as you do this. Returns the number of swaps that had to be performed.

`heapify_in_place_down(begin, end, compare)`

- `begin` - Random access iterator representing the beginning of the data to heapify.
- `end` - Random access iterator one past the end of the data that will be heapified.
- `compare` - The comparison function to be used when deciding whether swaps must occur.

This function takes the data in the given, random access iterator range, which may not yet be a heap, and heapifies it *in place*, by bubbling *down* for all of the nodes that have children. You are not allowed to use another array/`vector`/etc. to store things as you do this. Returns the number of swaps that had to be performed.

`heap_sort(begin, end, compare)`

- `begin` - Random access iterator representing the beginning of the data to sort.
- `end` - Random access iterator one past the end of the data that will be sorted.
- `compare` - The comparison function to be used when deciding whether swaps must occur.

This function sorts the data in the given, random access iterator range using the heapsort algorithm. You should heapify in place using the `compare` function as your heap condition, and then extract the root. What remains in the array after all the nodes have been extracted will be sorted.

You should choose the heapification algorithm that tends to require fewer swaps when you implement this.

Returns nothing.

NOTE Your tree is built using `compare` as the heap condition, and when it is unrolled, it will be in the opposite order of the order you would expect using the `compare` function directly in the `std::sort` algorithm. This means that heapsorting with a minheap gives you the items in descending order, and using a maxheap would give you the items in ascending order in the end.

THE `heap_priority_queue` CLASS (IN `heap_priority_queue.h`:

Constructor `heap_priority_queue(ITERATOR, ITERATOR)` This constucts your new `heap_priority_queue` with the the elements contained in the range specified with the two iterators already present, in a proper heap.

You should choose the heapification algorithm that tends to require fewer swaps when you implement this.

`top()` This returns a reference to the element at the top of the heap. As was also the case in `std::priority_queue`, this should never be called on an empty heap. Thus, you can assume that it never will be, and don't need to worry about what would happen if it were, as that would be *undefined behavior*.

`empty()` Returns `true` if, and only if, the heap is currently empty.

`size()` Returns the number of nodes currently in the heap.

`push(x)` Inserts the given key, `x`, into the heap.

`pop()` Removes the value currently at the top of the heap from the heap. Returns nothing. This should never be called on an empty heap, so you don't need to plan around that happening. Just like `std::priority_queue`, popping when empty is *undefined behavior*.

NOTES

Your work should be done in `heap.h` and `heap_priority_queue.h`. Do not alter the other files that were provided with the assignment.

You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission.

TESTING

There are a number of testing programs included. These will be used to evaluate the functionality of your implementations of the required functions. Typing `make` will attempt to compile them all, and will succeed to the degree it can with whatever you have implemented at that point.

The table below has a list of the tests available, and they are shown in order from least complex to most complex.

Order	Test	Purpose
1	`1-basic`	Tests the most basic heap functions with known data.
2	`2-bubble`	Allows you to test your `heap_bubble_up` and `heap_bubble_down` implementations.
3	`3-insdel`	Tests `heap_insert` and `heap_exract`
4	`4-heapify`	Tests `heapify_in_place_up` and `heapify_in_place` down.
5	`5-heapsort`	Tests your `heap_sort` implementation.
6	`6-pq`	Tests your `heap_priority_queue` implementation.

The expected output is contained in the `*.refout` files in the `output/` directory.

How To Submit

Submit, through the autograder, the following files:

- `heap.h` containing implementations of the heap functions described above.
- `heap_priority_queue.h` containing implementation of the `heap_priority_queue` class.

Grading Considerations

- Does it compile? Does it run? All of the tests should compile and run on turing/hopper with the `Makefile` provided, and points will be deducted for each test that will not compile.
- Does the output match for all of the tests? I have provided reference output for each of the test programs, so you can compare your program's output to what is expected. This output can be found in the files ending in `.refout`.
- Did you submit all of the source code needed to compile your program?
- Did you indent your code?
  - Indentation aids in the readability of source code, and if you're not indenting your code blocks, the grader will legitimately dislike you for it. I'm authorizing them to mark you off if you subject them to reading that.
- Did you document your code?
  - You need a docbox at the top of every one of the files you're required to change including:
    - Your name
    - Your zid
    - Your course section
    - A description of what the program does
  - You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
  - Add other comments inside your code blocks describing what you're doing and why.
  - The use of `doxygen` style comments is encouraged, but not required.