Programming Assignment 7

Priority Queue (Heap)

Approved Includes

```
<functional>
<deque>
<initializer_list>
<iostream>
<sstream>
<stdexcept>
<vector>
"heap.h"
"priority_queue.h"
```

Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

Starter Code

```
for X in {heap, priority_queue}
      X.h
      X_tests.cpp
      X_compile_test.cpp
Makefile
```

Files to Submit

```
heap.h
heap_tests.cpp
priority_queue.h
priority_queue_tests.cpp
```

Task 1: Heap

Implement stand-alone heap methods: `heapify`, `insert`, `get_min`, and `delete_min`.

Requirements

Files

`heap.h` - contains the template definitions
`heap_tests.cpp` - contains the test cases and test driver (main)

Functions

The `Container` type must satisfy the requirements of *SequenceContainer*, and its iterators must satisfy the requirements of *LegacyRandomAccessIterator*. Additionally, it must provide the following functions with the usual semantics:

- `front()`
- `push_back()`
- `pop_back()`

The standard containers std::vector and std::deque satisfy these requirements.

`template <class Container, class Compare=std::less<typename Container::value_type>>`
`void heapify(Container*, Compare <var name> =std::less<typename Container::value_type>{})` - build a heap with the data in the given container (passed by pointer value to indicate that this function modifies the container) using the specified comparator (comparison functor, default is std::less to make a min heap), the first element of the heap should be in index 1.

`template <class Container, class Compare=std::less<typename Container::value_type>>`
`void heap_insert(Container*, const typename Container::value_type&, Compare=std::less<typename Container::value_type>{})` - insert the specified value into the specified heap (passed by pointer value to indicate that this function modifies the container, which is assumed to already be in heap order with first element in index 1) using the specified comparator (default `std::less` for a min heap). If the container is empty, do the user a solid and make it a heap before inserting.

`template <class Container>`
`const typename Container::value_type& heap_get_min(const Container&)` - return the "minimum" value (whichever value is at the root of the heap) in the specified heap (which is passed by constant reference to indicate that this function does not modify the container). Throw `std::invalid_argument` if the heap is empty.

`template <class Container, class Compare=std::less<typename Container::value_type>>`

`void heap_delete_min(Container*, Compare=std::less<typename Container::value_type>{})` - remove the “minimum” value (whichever value is at the root of the heap) in the specified heap (passed by pointer value to indicate that this function modifies the container, which is assumed to already be in heap order with first element in index 1) using the specified comparator (default `std::less` for a min heap). Throw `std::invalid_argument` if the heap is empty.

Optional

`template <class Container, class Compare=std::less<typename Container::value_type>>`
`void heapify(Container&, Compare=std::less<typename Container::value_type>{})` - wrapper function for `heapify` if you have difficulty remembering to pass by pointer value, sends the address of the container on to the actual function.

`template <class Container, class Compare=std::less<typename Container::value_type>>`
`void heap_insert(Container&, const typename Container::value_type&, Compare=std::less<typename Container::value_type>{})` - wrapper function for `insert` if you have difficulty remembering to pass by pointer value, sends the address of the container on to the actual function.

`template <class Container, class Compare=std::less<typename Container::value_type>>`
`void heap_delete_min(Container&, Compare=std::less<typename Container::value_type>{})` - wrapper function for `delete_min` if you have difficulty remembering to pass by pointer value, sends the address of the container on to the actual function. Throw `std::invalid_argument` if the heap is empty.

Example

```
std::vector<int> heap{150,80,40,30,10,70,110,100,20,90,60,50,120,140,130};

std::cout << "before heapify: ";
for (int i : heap) { std::cout << i << " "; }
std::cout << std::endl;

heapify(&heap);

std::cout << "after heapify: ";
for (int i : heap) { std::cout << i << " "; }
std::cout << std::endl;

for (unsigned j = 0; j < 4; j++) {
    std::cout << "minimum is " << heap_get_min(heap) << std::endl;

    std::cout << "delete min" << std::endl;
    heap_delete_min(&heap);

    std::cout << "heap: ";
    for (int i : heap) { std::cout << i << " "; }
    std::cout << std::endl;
}

int values[] = {47,54,57,43,120,3};
for (unsigned j = 0; j < 6; j++) {
    std::cout << "insert " << values[j] << std::endl;
    heap_insert(&heap, values[j]);

    std::cout << "heap: ";
    for (int i : heap) { std::cout << i << " "; }
    std::cout << std::endl;
}
```

Example Output

Note about the "after heapify" lines: value in index 0 is unimportant (index 0 is not used, shown here only for completeness and example).

```
before heapify: 150 80 40 30 10 70 110 100 20 90 60 50 120 140 130
after heapify: 0 10 20 40 30 60 50 110 100 150 90 80 70 120 140 130
minimum is 10
delete min
heap: 0 20 30 40 100 60 50 110 130 150 90 80 70 120 140
minimum is 20
delete min
heap: 0 30 60 40 100 80 50 110 130 150 90 140 70 120
minimum is 30
delete min
heap: 0 40 60 50 100 80 70 110 130 150 90 140 120
minimum is 40
delete min
heap: 0 50 60 70 100 80 120 110 130 150 90 140
insert 47
heap: 0 47 60 50 100 80 70 110 130 150 90 140 120
insert 54
heap: 0 47 60 50 100 80 54 110 130 150 90 140 120 70
insert 57
heap: 0 47 60 50 100 80 54 57 130 150 90 140 120 70 110
insert 43
heap: 0 43 60 47 100 80 54 50 130 150 90 140 120 70 110 57
insert 120
heap: 0 43 60 47 100 80 54 50 120 150 90 140 120 70 110 57 130
insert 3
heap: 0 3 43 47 60 80 54 50 100 150 90 140 120 70 110 57 130 120
```

Task 2: Priority Queue

Implement a priority queue using the heap functions from Task 1.

Requirements

Files

`priority_queue.h` - contains the template definitions
`priority_queue_tests.cpp` - contains the test cases and test driver (main)

Class

```
template <class Comparable, class Container=std::vector<Comparable>, class
Compare=std::less<typename Container::value_type>>
class PriorityQueue;
```

Note: Read the compile test to see how to properly instantiate the class. Basically, the declaration looks like

```
PriorityQueue<T, std::vector<T>, std::less<T>> priority_queue;
```

which default constructs a priority queue of `T` values using a `std::vector` that holds `T` values and the `std::less` comparator which compares `T` values using "strictly less" (<). If you want to use a different value type, or a different container type, or a different comparator type, you change those in the template arguments (between the angle brackets: < … >). You put the values of the container and comparator you want to use into the arguments to the constructor:

```
PriorityQueue<T, std::vector<T>, std::less<T>> priority_queue(compare,
container);
```

where compare is a `std::less<T>` object and container is a `std::vector<T>` object.
The types of the constructor arguments and the types in the template arguments must match.

Functions (public)

Constructors

`PriorityQueue()` - makes an empty priority queue using the default container (std::vector) and default comparator (std::less).
`explicit PriorityQueue(const Compare&)` - makes an empty priority queue using the default container (std::vector) and specified comparator.
`explicit PriorityQueue(const Container&)` - makes a priority queue using the specified container (which may not be empty and is not a heap) and default comparator (std::less).
`PriorityQueue(const Compare&, const Container&)` - makes a priority queue using the specified container (which may not be empty and is not a heap) and the specified comparator.

Iterators

Optional

Element Access

`typename Container::const_reference top() const` - return the top of this priority queue. Throw std::invalid argument if the queue is empty.

Capacity

`bool empty() const` - returns true if this priority queue is empty
`size_t size() const` - returns the number of values in this priority queue

Modifiers

`void make_empty()` - remove all values from this priority queue
`void push(const typename Container::value_type&)` - insert the given lvalue reference into this priority queue
`void pop()` - remove the top of this priority queue. Does not throw any exceptions.

Visualization

`void print_queue(std::ostream&=std::cout) const` - pretty print the queue as a comma-and-space separated list of values to the specified output stream (default std::cout), prints "<empty>\n" if this priority queue is empty. E.g. `1, 2, 4, 3, 6, 5, 11`

"Optional" - They must work correctly but that doesn't mean *you* have to implement them

`PriorityQueue(const PriorityQueue&)` - constructs a copy of the specified priority queue (using same container type and comparator)
`~PriorityQueue()` - destroys this priority queue.
`PriorityQueue& operator=(const PriorityQueue&)` - assigns this priority queue to be a copy of the specified priority queue (with the same container type and comparator)

Optional for Real - not tested

`PriorityQueue(const Compare& compare, Container&& container)` - move constructs a priority queue using the specified comparator and container.
`PriorityQueue(PriorityQueue&&)` - move constructs a copy of the specified priority queue (using same container type and comparator)
`PriorityQueue& operator=(PriorityQueue&&)` - move assigns this priority queue to be a copy of the specified priority queue (with the same container type and comparator)
`void push(typename Container::value_type&&)` - insert the given value into this priority queue using move semantics

Example

```
std::cout << "SELECTION PROBLEM" << std::endl;
std::cout << "make a priority queue from N = 168 elements in O(N) time" <<
std::endl;
std::vector<int> values{509, 887, 53, 739, 491, 307, 727, 223, 919, 263, 983,
7, 809, 353, 659, 769, 173, 431, 619, 139, 2, 3, 181, 23, 283, 617, 463, 757,
89, 541, 997, 743, 907, 13, 337, 349, 523, 857, 97, 827, 661, 67, 373, 59, 11,
277, 379, 19, 941, 607, 367, 101, 457, 929, 599, 971, 967, 647, 71, 991, 211,
467, 881, 137, 311, 673, 197, 179, 859, 239, 233, 631, 449, 281, 499, 269,
877, 421, 419, 613, 593, 383, 937, 569, 487, 839, 479, 461, 683, 653, 227, 61,
107, 113, 947, 191, 103, 313, 733, 151, 257, 73, 821, 547, 521, 691, 83, 823,
443, 31, 5, 643, 131, 389, 571, 163, 271, 601, 359, 199, 853, 29, 167, 557,
157, 193, 977, 37, 41, 773, 347, 709, 251, 331, 829, 503, 409, 719, 397, 241,
47, 641, 787, 863, 109, 587, 17, 751, 229, 911, 811, 317, 563, 701, 797, 953,
293, 149, 439, 127, 883, 577, 79, 433, 43, 761, 401, 677};
PriorityQueue<int> pq(values);

std::cout << "pop k = 42 = O(N / log N) elements in O(k log N) = O(N) time" <<
std::endl;
for (int i = 0; i < 41; i++) { pq.pop(); }
// ^^^ pops 41, top is would-be 42nd pop vvvvv
std::cout << "found k-th smallest element = " << pq.top() << " in O(N + k log
N)= O(N) time" << std::endl;

std::cout << std::endl << "INTERMISSION" << std::endl;

std::cout << "empty the queue in O(1) time" << std::endl;
pq.make_empty();

std::cout << std::endl << "DISCRETE EVENT SIMULATION" << std::endl;
int t = 0;
size_t busy_start = 0, busy_stop = 0, busy_time = 0, wait_time = 0, cnt = 0,
cust_id = 0, next = 7;
bool busy = false;
std::list<size_t> line;
pq.push(t);
while (!pq.empty()) {
    t = pq.top();
    pq.pop();
    if (t%2) {
        // departure
        std::cout << "customer departed at time " << t << std::endl;
        cnt++;
        if (line.empty()) {
```

```
            busy = false;
            busy_stop = t;
            busy_time += busy_stop - busy_start;
        } else {
            // next in line
            int arrival_time = line.front();
            line.pop_front();
            wait_time += (t - arrival_time);
            std::cout << "next customer in line has been waiting " << (t -
arrival_time) << " time units" << std::endl;
            // schedule departure at odd time (t is odd right now), >0 from
now
            int service_time = 2*((next+3)%5)+2;
            pq.push(t + service_time);
            next = (next+5)%11;
        }
    } else {
        // arrival
        std::cout << "customer " << ++cust_id << " arrived at time " << t <<
std::endl;
        if (cust_id < 10) {
            // schedule next arrival at even time (t is even right now)
            int interarrival_time = 2*((next+3)%5)+2;
            pq.push(t + interarrival_time);
            next = (next+5)%11;
        }
        if (busy) {
            // wait in line
            line.push_back(t);
            std::cout << "  server is busy, customer must wait in line, there
are " << line.size() << " in line" << std::endl;
        } else {
            // serve
            busy = true;
            busy_start = t;
            std::cout << "  service begins immediately" << std::endl;
            // schedule departure at odd time (t is even right now), >0 from
now
            int service_time = 2*((next+3)%5) + 1;
            pq.push(t + service_time);
            next = (next+5)%11;
        }
    }
}
```

```
std::cout << "end of simulation\n-----------------" << std::endl;
std::cout << "served 10 customers in " << t << " time units" << std::endl;
std::cout << "server was busy for " << busy_time << " total time units" <<
std::endl;
std::cout << "customers waited " << wait_time << " total time units" <<
std::endl;
```

Example Output

```
SELECTION PROBLEM
make a priority queue from N = 168 elements in O(N) time
pop k = 42 = O(N / log N) elements in O(k log N) = O(N) time
found k-th smallest element = 181 in O(N + k log N)= O(N) time

INTERMISSION
empty the queue in O(1) time

DISCRETE EVENT SIMULATION
customer 1 arrived at time 0
  service begins immediately
customer 2 arrived at time 2
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 9
next customer in line has been waiting 7 time units
customer 3 arrived at time 12
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 17
next customer in line has been waiting 5 time units
customer 4 arrived at time 20
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 25
next customer in line has been waiting 5 time units
customer 5 arrived at time 26
  server is busy, customer must wait in line, there are 1 in line
customer 6 arrived at time 30
  server is busy, customer must wait in line, there are 2 in line
customer departed at time 31
next customer in line has been waiting 5 time units
customer departed at time 33
next customer in line has been waiting 3 time units
customer 7 arrived at time 34
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 35
next customer in line has been waiting 1 time units
customer 8 arrived at time 44
```

```
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 45
next customer in line has been waiting 1 time units
customer 9 arrived at time 52
  server is busy, customer must wait in line, there are 1 in line
customer departed at time 53
next customer in line has been waiting 1 time units
customer departed at time 59
customer 10 arrived at time 60
  service begins immediately
customer departed at time 65
end of simulation
-----------------
served 10 customers in 65 time units
server was busy for 64 total time units
customers waited 28 total time units
```

Notes

Generic Programming

The beauty of generic programming (templates) is that you don't have to know the specific types of parameters (or that you actually do know, but it's a generic type). Look:

```
template <class Container,
          class Compare=std::less<typename Container::value_type>>
                                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

You see that part (**bold**, underlined by **^**s)? That's the type of object that is in the container, e.g. `int` or `std::string` or whatever.

We can make one those: `typename Container::value_type value;`

So, if the container is holding `int`s, this is just `int value;`
If the container is holding `std::string`s, this is just `std::string value;`

If you want an anonymous value (i.e. rvalue reference): `typename Container::value_type{}`