```
#ifndef priority_queue_h_
#define priority_queue_h_

#include <cassert>
#include <iostream>
#include <vector>
#include <map>

// The DistancePixel_PriorityQueue is a customized, non-templated
// priority queue that stores DistancePixel pointers in a heap.  The
// elements in the heap can be looked up in a map, to quickly find out
// the current index of the element within the heap.

// ASSIGNMENT: The class implementation is incomplete.  Finish the
// implementation of this class, and add any functions you need.

// ========================================================================

class DistancePixel_PriorityQueue {

public:

    // --------------------------
    // CONSTRUCTORS
    // default constructor
    DistancePixel_PriorityQueue() {}
    // construct a heap from a vector of data
    DistancePixel_PriorityQueue(const std::vector<DistancePixel*> &values) {
        // ASSIGNMENT: Implement this function
        for (int i = 0; i < values.size(); ++i) {
            push(values[i]);
        }
    }

    // -------------------------
    // ACCESSORS
    int size() { return m_heap.size(); }
    bool empty() { return m_heap.empty(); }
    int last_non_leaf() { return (size()-1) / 2; }
    int get_parent(int i) { assert (i > 0 && i < size()); return (i-1) / 2; }
    int has_left_child(int i) { return (2*i)+1 < size(); }
    int has_right_child(int i) { return (2*i)+2 < size(); }
    int get_left_child(int i) { assert (i >= 0 && has_left_child(i)); return 2*i
        + 1; }
    int get_right_child(int i) { assert (i >= 0 && has_right_child(i)); return 2*
        i + 2; }

    // read the top element
    const DistancePixel* top() const  {
        assert(!m_heap.empty());
        return m_heap[0];
    }

    // is this element in the heap?
    bool in_heap(DistancePixel* element) const {
        std::map<DistancePixel*,int>::const_iterator itr = backpointers.find
```

```
            (element);
        return (itr != backpointers.end());
    }

    // add an element to the heap
    void push(DistancePixel* element) {
        std::map<DistancePixel*,int>::iterator itr = backpointers.find(element);
        assert (itr == backpointers.end());
        //-----------------------------------------------------------------
        m_heap.push_back(element);
        backpointers[element] = m_heap.size()-1;
        this->percolate_up(int(m_heap.size()-1));
        //-----------------------------------------------------------------
    }

    // the value of this element has been edited, move the element up or down
    void update_position(DistancePixel* element) {
        std::map<DistancePixel*,int>::iterator itr = backpointers.find(element);
        assert (itr != backpointers.end());
        this->percolate_up(itr->second);
        this->percolate_down(itr->second);
    }

    // remove the top (minimum) element
    void pop() {
        assert(!m_heap.empty());
        int success = backpointers.erase(m_heap[0]);
        assert (success == 1);
        m_heap[0] = m_heap.back();
        m_heap.pop_back();
        this->percolate_down(0);
    }

private:

    // REPRESENTATION
    // the heap is stored in a vector representation (the binary tree
    // structure "unrolled" one row at a time)
    //-------------------------------------------------------------------------
    std::vector<DistancePixel*> m_heap;
    // the map stores a correpondence between elements & indices in the heap
    std::map<DistancePixel*,int> backpointers;
    //-------------------------------------------------------------------------

    // private helper functions
    void percolate_up(int i) {
        // ASSIGNMENT: Implement this function
        while (i > 0) {
            if (*m_heap[i] < *m_heap[get_parent(i)]) {
                // Adjust the heap vector:
                //-----------------------------------
                DistancePixel* temp = m_heap[i];
                m_heap[i] = m_heap[get_parent(i)];
                m_heap[get_parent(i)] = temp;
                // Adjust the map:
                //-----------------------------------
```

```
                backpointers[temp] = get_parent(i);
                backpointers[m_heap[get_parent(i)]] = i;

                i = get_parent(i);
            } else {
                break;
            }
        }
    }

    void percolate_down(int i) {
        // ASSIGNMENT: Implement this function
        int temp = 0;
        while (get_right_child(i) < m_heap.size()) {
            if (*m_heap[get_left_child(i)] < *m_heap[get_right_child(i)]) {
                temp = get_left_child(i);
            } else {
                temp = get_right_child(i);
            }
            if (m_heap[temp] < m_heap[i]) {
                // Adjust the heap vector:
                //-----------------------------------------
                DistancePixel* temp_pointer = m_heap[temp];
                m_heap[temp] = m_heap[i];
                m_heap[i] = temp_pointer;
                // Adjust the map:
                //-----------------------------------------
                backpointers[temp_pointer] = i;
                backpointers[m_heap[i]] = temp;

                i = temp;
            } else {
                break;
            }
        }
    }

};

#endif
```