Optional Project Part 01: Priority Queue via min-heap

Complete By: Saturday, May 6th @ 11:59pm
Assignment: Complete PQueue class
Policy: Individual work only, late work *not* accepted
Submission: online via zyLabs section 5.16

PQueue with a min-heap

The assignment is to complete the implementation of a priority queue specifically designed for Dijkstra's shortest weighted path algorithm. This implies the class must support O(logN) push and pop operations. This requires a min-heap implementation, along with support for fast lookup to enable deletion of values already in the queue.

If you are completely new to the min-heap data structure, see section 4.27 in the online Zybook. Then review the lecture notes / recording from the last day of class (PDF). Another good resoruce is the following online summaries of the PopMin (deletion) and Push (insertion) algorithms:

PopMin: http://www.algolist.net/Data_structures/Binary_heap/Remove_minimum
Push: http://www.algolist.net/Data_structures/Binary_heap/Insertion

Note that these are the base algorithms, and will need to be modified to suit our goals.

To implement the priority queue you are required to use min-heap with a C-style array implementation. Here's part of the class definition from the provided "pqueue.h" header file in Zyante section 5.16:

```
/*pqueue.h*/

class PQueue
{
private:
  class Elem
  {
  public:
    int     V;
    double  D;

    Elem()  // default constructor:
    {
      V = -1;
```

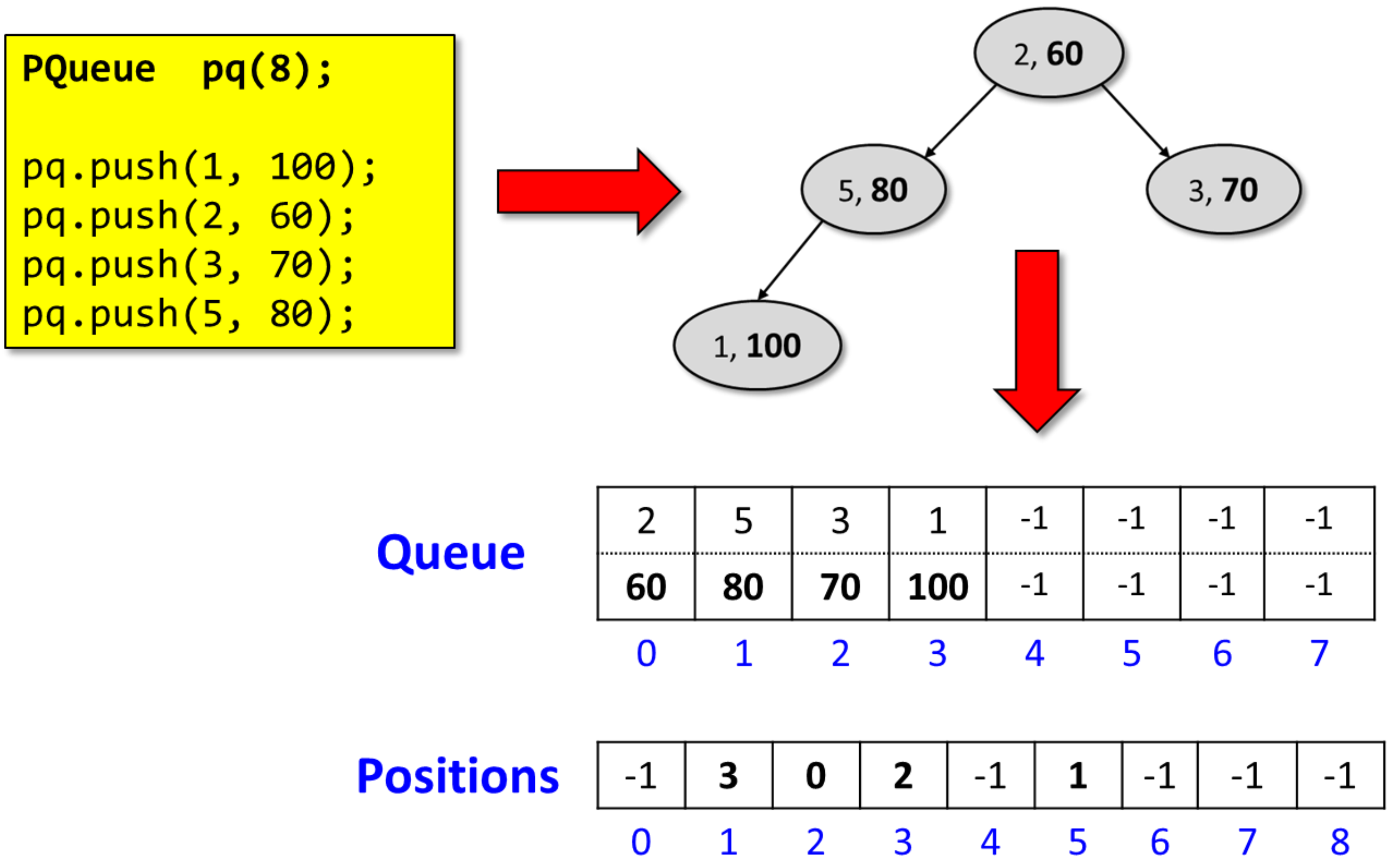
```
            D = -1.0;
          }

          Elem(int v, double d)
          {
            V = v;
            D = d;
          }
        };

        int   *Positions;     // position of every vertex in queue (-1 if not present)
        Elem *Queue;          // actual priority queue

        int   NumElements;    // # of elements currently in queue
        int   Capacity;       // max # of vertices we can support
```

The **Queue** denotes the min-heap: a C-style array of (vertex, distance) pairs, in ascending order by distance. The **Positions** is used for direct hashing by vertex, to enable fast lookup of whether a vertex is in the queue, and if so, what position it is in. This is needed when Dijkstra's algorithm finds a better path and needs to update the vertex's distance in the priority queue. Here's a diagram to help you understand how Positions and Queue relate to one another:



```
PQueue  pq(8);

pq.push(1, 100);
pq.push(2, 60);
pq.push(3, 70);
pq.push(5, 80);
```

Queue	2	5	3	1	-1	-1	-1	-1
	60	**80**	**70**	**100**	-1	-1	-1	-1
	0	1	2	3	4	5	6	7

Positions	-1	**3**	**0**	**2**	-1	**1**	-1	-1	-1
	0	1	2	3	4	5	6	7	8

For example, notice that vertex 0 has a position of -1, which means it is not in the queue. Vertex 1 has a position of 3 — notice (1, 100) is stored at index 3 in the Queue.

Insert and Delete

Suppose (v, d) is the first element in the queue. **PopMin()** deletes (v, d) from the queue, changes v's position to -1, and returns v. To delete the first element from a min-heap, the algorithm is well-known:

PopMin: http://www.algolist.net/Data_structures/Binary_heap/Remove_minimum

Keep in mind that whenever 2 elements in the queue are swapped, you must also update the corresponding positions in the **Positions** array.

The **Push(v, d)** function is the more interesting one. It first checks to see vertex v already exists in the queue, and if so, deletes v and its associated distance. This is more challenging than PopMin, because you could be deleting from anywhere in the queue — first element, last element, or somewhere in-between. Finding v's position is easy — that's the purpose of the **Positions** array. The harder part is generalizing the delete algorithm. Here's the pseudo-code:

Deleting element @ position p:

1. Let (v, d) = the element at position p
2. Mark v as deleted in Positions array
3. If p denotes the last element in the queue, reduce # of elements by 1 and stop;
4. Move last element into position p
5. Reduce # of elements by 1
6. Let e be the element at position p
7. Does e have a parent? If so, compare and see if e should swap upward… If swap occurs, update p and e and repeat...
8. Does e have children? One child or two? If just one child, compare e with child and swap if necessary. If two children, compare e with smaller of children, and swap if necessary. If swap occurs, update p and e and repeat…

Keep in mind that whenever an element in the queue moves / swaps, you must update the Positions array appropriately. Once the vertex v has been deleted form the queue, then simply follow the standard insert algorithm for a min-heap:

Push: http://www.algolist.net/Data_structures/Binary_heap/Insertion

As (v, d) swaps into position, remember to update the Positions array.

NOTE: it turns out that the test cases do in fact depend on the order in which you swap things, because the fill() function stores duplicated in the queue. To pass the test cases, here are the rules to follow:

1. *when swapping up the tree, only swap if distance is > than parent (if <= then stop)*
2. *when swapping down the tree, only swap if distance is > than smaller child (if <= then stop)*
3. *when swapping down the tree, *if* there are two children and those children have the same distance, swap with the *right* child.*

In hindsight, when distances are equal we probably should have put the elements in vertex order.

Requirements

Most of the PQueue class has been implemented for you. Your job is focused on implementing the private helper functions **Insert(v, d)** and **Delete(position)**. See the header comments in “pqueue.cpp” for more details. You are required to follow the implementation as provided, in particular using an array-based min-heap for the **Queue** and direct hashing via the **Positions** array. Feel free to add additional helper functions to the PQueue class, but do not change the existing function declarations, and do not change the testing code in “main.cpp”. Your changes should be limited to (1) the private section of “pqueue.h”, and (2) “pqueue.cpp”.

Submission

The program is to be submitted via zyLabs: see “**Optional_Project_01 PQueue**”, section 5.16. Submit your three files in the tabs provided: main.cpp, pqueue.h, and pqueue.cpp. When you’re ready to submit, copy-paste your program into the zyLabs editor pane, switch to “Submit” mode, and click “SUBMIT FOR GRADING”. You are limited to at most **20** submissions.

Policy

Late work is *not* accepted. All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .