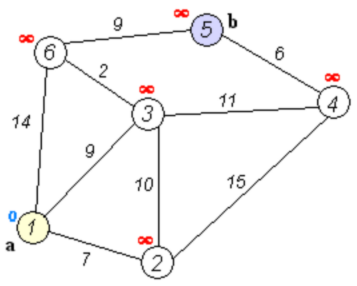
Assignment 5

For this assignment you'll implement Dijkstra's algorithm for finding the shortest path between nodes in a graph.



To achieve this goal you'll need two important data structures (1) a `Graph` and (2) a `Priority Queue`, after those two are complete, you can begin implementing (3) the actual Dijkstra algorithm.

(1) Graph Object
For the `Graph` object, you will implement your own. I have provided you with an incomplete `Graph.h` which includes only the public facing aspects of the `Graph` class. You should modify it (adding to the private section mostly).

Sometimes when working on a software project you'll have some code to work with that was left over by someone working on the project in the past. In this assignment we have `GraphAttempt.cpp` which we can pretend was left over in this way. It is partially complete, and has several `TODO` comments to indicate places where code is missing. You should rename this file to `Graph.cpp` and complete the implementation.

I have also provided to you a lightly redacted `GraphTests.cpp` program, which includes some of the tests I will run on your `Graph`. You should fill in more tests and add test functions as you see necessary.

Please include a `Makefile` to compile this program (yours may differ slightly).

```
all:
     g++ -Wall -g Graph.cpp GraphTests.cpp -o graph-tests
```

(2) Priority Queue Object
For the `Priority Queue` object, it would be nice to be able to use the standard library `std::priority_queue` directly to implement Dijkstra's algorithm. Unfortunately, the interface is somewhat limited (http://www.cplusplus.com/reference/queue/priority_queue/, https://en.cppreference.com/w/cpp/container/priority_queue). To implement Dijkstra's algorithm we need an `Update()` method to change the priority of items *in the queue*. Additionally, it may not be necessary, but it would might be nice to have a `Contains()` method. **Both** of these are missing. I guess we can write a letter to Bjarne Stroustrup and ask him what the hell his problem is. For now

we're just going to have to fix the problem for ourselves by extending (writing a child-class of) `std::priority_queue.`

Create a `BetterPriorityQueue` class that extends `std::priority_queue` thusly:

```
class BetterPriorityQueue: public priority_queue<BPQNode, vector<BPQNode>,
greater<BPQNode>>::priority_queue {

// ... your code goes here ...

};
```

Oh yeah, that's a big one! Specifying the `priority_queue` this way makes three distinctions. First, the items inside this queue will be `BPQNode` which is a struct. The code for for `BPQNode` is provided to you in `BPQNode_struct.txt`, but you will have to decide where to incorporate it into your project. The second parameter specifies that a `vector` of `BPQNode` will be used as the underlying mechanism to implement the priority queue (fine, what would we want besides a vector anyway?). The third parameter specifies that this queue will place the minimum item first instead of the maximum item first (also fine).

Your `BetterPriorityQueue` should add **four** new public methods.

- `Contains()` takes a `BPQNode` and returns `true` if that `BPQNode` is in the queue.

- `Update()` takes a `BPQNode` and uses the values in that `BPQNode` to change the priority of a matching `BPQNode` in the queue. It returns `true` if any change is actually made. For example,

  suppose `BetterPriorityQueue` is
  ```
  [(1, pri: 0), (2, pri: 1), (3, pri: 2), (4, pri: 3), (5, pri: 4), (6, pri: 5)]
  ```

  and the inputted `BPQNode n` is `(6, pri: 0)`
  calling `bpq.Update(n)` should result in
  ```
  [(1, pri: 0), (2, pri: 1), (6, pri: 0), (4, pri: 3), (5, pri: 4), (3, pri: 2)]
  ```

  and the `Update()` method should return `true`.

  If the `BPQNode` doesn't match any existing in the queue, `Update()` should have no effect on the queue and return `false`. If the `BPQNode` doesn't have a lower priority than the matching `BPQNode` in the queue, `Update()` should have no effect on the queue and return `false`.

  Note: The order of the nodes is determined by an underlying binary heap, which is already implemented inside of `priority_queue`. You should be careful not to disrupt the state of that binary heap. Think back to CS2 and be thankful that you don't have to implement a binary heap again! Appreciate the author(s) of the C++ binary heap by not messing up the order of the items!

- `ToString()` returns a string representation of the queue like those shown above including `[]`s and commas etc. as necessary.

- `BPQNodeToString()` (static), takes a `BPQNode struct` and returns a string representation like those shown above (each item in the queue).

Extending `std::priority_queue` provides you access to the protected member `c`, which is the `vector<BPQNode>` containing the items of the queue. You can use `this->c` to implement the above methods. Take care not to mess with the items of the vector in a way that would make it an invalid binary heap.

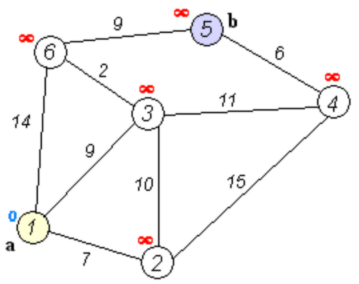
An incomplete test file `BPQTests.cpp` has been provided to you. You should fill in more tests and add test functions as you see necessary.

At this point you should ensure you have a well tested `Graph` and `BetterPriorityQueue`. Or not, I suppose. I can't really stop you from simply giving up. I'll leave it up to you to decide how much testing is enough testing for your own piece-of-mind. Just be sure to remember that your success and happiness for the rest of your professional life depends on your exhaustive completion of this specific homework assignment.

Add this program to your `Makefile` under the all directive (yours may differ slightly)

```
all:
	g++ -Wall -g Graph.cpp GraphTests.cpp -o graph-tests
	g++ -Wall -g Graph.cpp BetterPriorityQueue.cpp BPQTests.cpp -o bpq-tests
```

(3) The Actual Dijkstra Algorithm



In the above graph the shortest path from node a (1) to node b (5) costs 20. It is achieved by traveling from 1 → 3 (costing 9), then from 3 → 6 (costing an additional 2), and finally from 6 → 5 (costing an additional 9). Your program will output the final answer, 20.

Using your `Graph` and your `BetterPriorityQueue`, implement Dijkstra's algorithm. The header file `Dijkstra.h` is provided to ensure your Dijkstra function has the correct signature. Write the `dijkstra()` function in `Dijkstra.cpp` as specified by `Dijkstra.h`

`DijkstraTests.cpp` can be used to test your solution. Consider adding more tests! Seriously, it's a good idea.

Add a comment at the top of the `dijkstra()` method that explains the Big-O time-complexity of the algorithm. Be careful about your answer. It isn't straightforward as it depends on how you implemented the Dijkstra algorithm, but also it depends on the `Graph` and `BetterPriorityQueue` implementations!

In the end your `Makefile` should compile these three separate programs under the all directive.

```
Graph.cpp GraphTests.cpp → graph-tests
Graph.cpp BetterPriorityQueue.cpp BPQTests.cpp → bpq-tests
Graph.cpp BetterPriorityQueue.cpp Dijkstra.cpp DijkstraTests.cpp → dijkstra
```

Any other directives in your `Makefile` are up to you.

Sample Output:

```
user@machine$ ./dijkstra
Testing Dijkstra Algorithm...
---Graph---
        nodes: [(1), (2), (3), (4), (5), (6)]
        edges: [((1)->(2) w:7), ((1)->(3) w:9), ((1)->(6) w:14), ((2)->(1) w:7),
((2)->(3) w:10), ((2)->(4) w:15), ((3)->(1) w:9), ((3)->(2) w:10), ((3)->(6) w:2),
((3)->(4) w:11), ((4)->(2) w:15), ((4)->(3) w:11), ((4)->(5) w:6), ((5)->(6) w:9),
((5)->(4) w:6), ((6)->(1) w:14), ((6)->(3) w:2), ((6)->(5) w:9)]
---Graph---

start: 1  goal: 5
Output from Dijkstra(1, 5) => 20
DONE Testing Dijkstra Algorithm

Deep Testing Dijkstra Algorithm...
DONE Deep Testing Dijkstra Algorithm

ALL TESTS PASSED!
```

When you're done with this assignment you should consider the place you've arrived at. Could someone in CS1 do this? NOT A CHANCE! They don't have access to this PDF.

Proper Comments and Citations

Your code should have comments that explain "why" the code is written the way it is. You should also use comments to cite sources you used such as websites, tutors, and classmates.

Submission: Your program will be submitted using GitHub classroom.

Following the instructions on canvas you should have a repository cloned to your computer which contains these assignment instructions. That same repository will be used for you to submit your solutions.

1) Edit and write code in the repo on your computer. Use `git add`, `git commit`, and `git push` to put that code onto github. I have access to the repo. You can make as many commits as you like for this assignment with no penalty.

2) I will grade the final commit made before the due-date.

3) You don't have to make any submission on canvas.