Assignment #4

Due dates:

Tests (for Parts 1, 2, and 3): Monday, 4 March 2024, 8:00 pm
Code (for Parts 1, 2, and 3): Friday, 8 March 2024, 8:00 pm

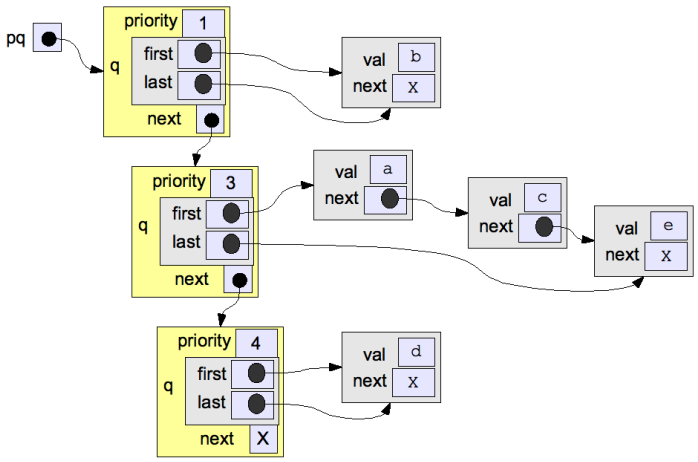
- *For all programming questions below, write your solutions in the dialect of C++ used in class. You may use the following libraries, and no others:* `iostream`, `string`, `cassert`, `vector`.
- *Store your set of functions in files named* `a4p1.cc`, `a4p2.cc`, *and* `a4p3.cc`*; you have been provided with stubs for these files to get you going. Your tests should be put in files named* `a4p1Test.cc`, `a4p2Test.cc` *and* `a4p3Test.cc`*. There are stubs for these as well, currently they contain copies of the Marmoset public tests.*
- *As usual, test coverage will be assessed by running your tests on the staff code. More info on the 't' Marmoset projects can be found in your* `a4pXTest.cc` *files.*
- *We will be checking for memory leaks; if you remove an element or nuke a list, or anything like that, don't forget to clean up after yourself.*
- *You need to write your unit tests using GTest framework. Give your tests some meaningful names to understand where you should look at in the case of test failures. You could use the Given-When-Then approach as a guide for unit test naming.*

Key ideas in this assignment

- Queue, Priority Queue, and Binary Search Tree data structures

Part I: Priority Queue - 40%

We will implement a priority queue using the LoL (List-of-Lists) approach we discussed in class. Have a look at the diagram below that combines several ideas shown in class. You might note that a priority queue is a list of nodes sorted by priority. In addition to a priority, each node also has a queue associated with it. The insertion order for this example was `(a 3)`, `(b 1)`, `(c 3)`, `(d 4)`, `(e 3)`. Thus, the removal order (assuming no more additions) would be `b a c e d`.



The definitions for `struct` types `Queue` and `Qnode` (the `Node` struct type renamed so as not to be confused with the `PQnode` struct type that we'll need to define later on; these are the grey-coloured nodes in the diagram), plus the implementations for the procedures `Q_init`, `Q_isEmpty`, `Q_enter`, `Q_first`, and `Q_leave` are provided for you in `a4p1.cc`. The implementations are almost identical to those discussed in class.

The `PQnode` struct type, which you can see pictorially in the diagram as the yellow-coloured nodes, are used to implement the list for priority queue. Your job is to implement the priority queue operations listed below. It is recommended to use a combination of direct use of the queue operations (with no tweaking at all) and a certain amount of tweaking of the sorted list routines discussed in class.

Reusing the `Queue` and `Qnode` types leads to the below definition of `PQNode`. We will also create a `typedef` called `PQ` which is just a pointer to `PQnode`; this is important as most of the priority queue routines will use this type.

```
struct Qnode {
    std::string val;
    Qnode* next;
};
struct Queue {
    Qnode* first;
    Qnode* last;
};
```

```
struct PQnode {
    int priority;
    Queue q;
    PQnode* next;
};
typedef PQnode* PQ;
```

We want you to define the following routines; the meaning of most are obvious, but a couple require some explanation.

```
void PQ_init (PQ& pq) { pq = nullptr;} // Free sample :-)
bool PQ_isEmpty (const PQ& pq) {...}
void PQ_enter (PQ& pq, std::string val, int priority) {...}
std::string PQ_first (const PQ& pq) {...}
void PQ_leave (PQ& pq) {...}
int PQ_size (const PQ& pq) {...}
int PQ_sizeByPriority (const PQ& pq, int priority) {...}
int PQ_numPriorities (const PQ& pq) {...}
void PQ_nuke (PQ & pq) {...}
```

- When you add a new element (*i.e.*call `PQ_enter`), you should first determine if there is already an existing queue for that priority. If there is, then just add it to that queue. If not, you'll have to create a new `PQnode` and insert it into the appropriate place in the list, as well as create a new queue and insert the new element into that queue. Note that `PQ_enter` will likely be the hardest function to write.
- When you remove the "top" element (*i.e.*call `PQ_leave`), you should check if that element is the last one of that priority. If it is, then you should delete the `PQnode` for that priority.
- The procedure `PQ_size` returns an integer that is the number of elements of all priorities in the queue. For example, the priority queue in the diagram has five elements.
- The procedure `PQ_sizeByPriority` takes an integer and returns the number of elements of that priority currently in the priority queue. For the example shown in the diagram, `PQ_sizeByPriority` of 4 would return 1, of 3 would return 3, of -17 would return 0, and of 5 would return 0. (Note that we don't treat a negative priority as an error here; we just find zero elements of that priority when we look).
- The procedure `PQ_numPriorities` should return the number of distinct priorities for which there is at least one active element. For example, it should return 3 for the example shown in the diagram.
- If `PQ_first` and `PQ_leave` are called on an priority queue, then abort via `assert`. We've provided an appropriate error message in the code skeleton. Finally, `PQ_nuke` should delete all heap-based storage associated with the provided priority queue instance.

We strongly advise you to simply use the queue operations pretty much as-is as servants to your greater purpose. It will make things much easier for you to treat the individual queues like a "black box" abstraction. If you want to augment the queue a bit, that's fine, but try to maintain a strict demarcation between the two ideas (queue versus priority queue).

Part II: Binary Search Tree - 20%

Consider the code for binary search trees that we presented in class, including the types `BST_Node` and `BST` as well as the procedures `BST_init`, `BST_isEmpty`, `BST_has`, `BST_print`, and `BST_insert` (the implementations of these procedures are given in `a4p2.cc`). Complete the package by implementing `BST_remove` using the algorithm described in class. Also, provide a definition of `BST_nuke`, which deletes all of the (heap-based) `BST_Node`s of the indicated BST. Give it a good workout with your testing, as you'll probably be re-using this code in the next part also. Here are the interfaces for the functions:

```
void BST_remove (BST& root, std::string key) { ... }
void BST_nuke (BST& root) { ... }
```

Within `BST_remove`, if the `key` cannot be found in the BST, then abort via `assert`; we've provided an appropriate error message in the code skeleton. Hint: Recursive solutions are probably the best bet here.

Part III: Stand-By List - 40%

Consider a hybrid data structure for storing information about passengers waiting for a stand-by seat on a flight; we're going to call this data structure a *stand-by list*, or SBL. The only information about the passenger we need to store explicitly is their name, but we'd like to store those names sorted both lexicographically (so we can quickly check who's in the list) and in arrival order (so the people who have been waiting longest can board first).

To accomplish this we are going to add a `next` pointer to our `BST_Node` struct to create what we call an `SBLnode`, which we will use as a part of two data structures simultaneously. Given an `SBL sbl`, following the `left` and `right` pointers of it's `SBLnode`s (starting at `sbl.root`) should lead to a BST containing all the passenger names sorted lexicographically, while following the `next` pointers of the `SBLnode`s (starting at `sbl.q.first` and ending at `sbl.q.last`) should lead to a Queue containging all the passengers names sorted by arrival order. Note that the `BST` and `Queue` of a `SBL` should contain the same nodes (*i.e.*one `SBLnode` per passenger).

It is strongly recommended that you copy over and reuse the queue and BST operations from parts 1 & 2 (with some changes as needed) to build your SBL implementation. The `typedef` statements defining `Q_Node`s and `BST_Node`s to be `SBLnode`s are provided to make this process easier.

Here are the `struct` definitions:

```
struct SBLnode {
    std::string name;
    SBLnode *next;
    SBLnode *left, *right;
};
struct Queue {
    SBLnode *first, *last;
};
typedef SBLnode* BST;
```

```
struct SBL {
    Queue q;
    BST root;
    int numElts;
};
typedef SBLnode Q_Node;
typedef SBLnode BST_Node;
```

Now onto the specific requirements. Your job here is to define the following operations:

- `void SBL_init (SBL& sbl)` — initialize the SBL data structure in the "obvious" way (see `a4p3Test.cc`).
- `int SBL_size (const SBL& sbl)` — returns the number of people currently in the stand-by list; the `SBL` type contains a counter to make this efficient, but you'll have to remember to update it in your other functions.
- `void SBL_arrive (SBL& sbl, std::string name)` — adds a new person to the SBL. Hint: Create only one node, but adjust the pointers in separate procedures.
- `void SBL_leave (SBL& sbl)` — removes the person at the front of the SBL (the person who has been waiting longest in the queue); it also removes them from the BST.

- `std::string SBL_first (const SBL& sbl)` — return the name of the person at the front of the SBL.
- `bool SBL_has (const SBL& sbl, std::string name)` — returns `true` iff the name corresponds to someone waiting in the SBL. Hint: Think about the quickest way to answer this question.
- `void SBL_toArrivalOrderString (const SBL& sbl)` — Puts all the currently stored names together in a string, sorted by arrival order. See `a4p3Test.cc` for formatting examples.
- `void SBL_toLexicographicalOrderString (const SBL& sbl)` — Puts all the currently stored names together in a string, sorted lexicographically. See `a4p3Test.cc` for formatting examples.
- `void SBL_nuke (SBL & sbl)` — delete all of the heap-based storage associated with the provided SBL instance.

If `SBL_leave` or `SBL_first` are called on an empty list, then abort via `assert`; we've provided an appropriate error message in the code skeleton.