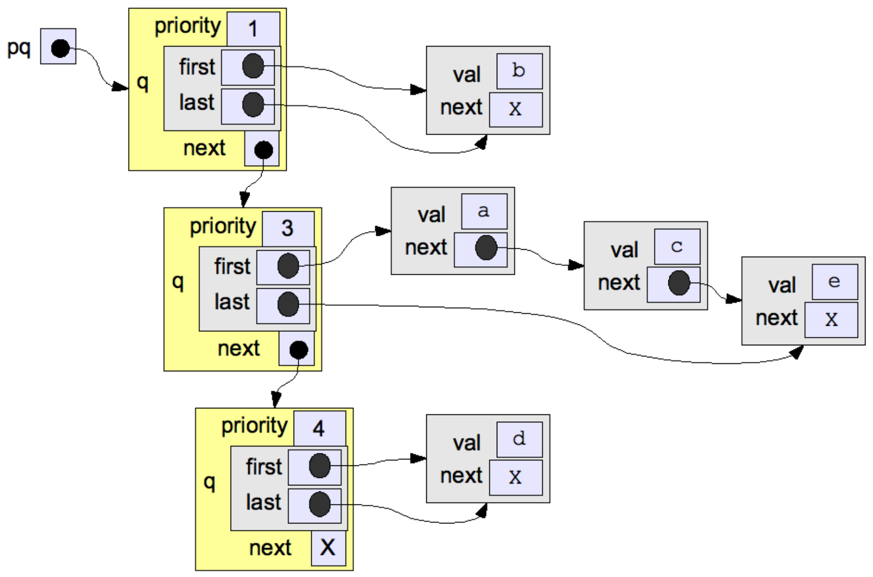
Assignment #4

Due date: Friday, 4 March 2016, 5:59 pm

- *Write your solutions in the dialect of C++ used in class.*
- *Store your set of functions in files named* `a4q1.cc`, `a4q2.cc`, etc.
- *You may define your own* `main` *function, but we will use our own* `main` *to test your code.*

Part I

Your job here is to 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`.



I implemented this by stealing the code from my queue implementation discussed in class: the `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 procedures `initQ`, `isEmptyQ`, `enterQ`, `firstQ`, and `leaveQ` (I also defined a `printQ` procedure for making debugging easier).

Then, I defined a `PQnode` struct type, which you can see pictorially in the diagram as the yellow-coloured nodes. The priority queue operations (see below) were implemented by a combination of direct use of the queue operations (with no tweaking at all), plus a certain amount of cannibalization and tweaking of the sorted list routines discussed in class. Feel free to do the same.

We created 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:

```
typedef PQnode* PQ;
```

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

```
void initPQ (PQ& pq) { pq = NULL;} // Free sample :-)
bool isEmptyPQ (const PQ& pq) {...}
void enterPQ (PQ& pq, string val, int priority) {...}
string firstPQ (const PQ& pq) {...}
void leavePQ (PQ& pq) {...}
int sizePQ (const PQ& pq) {...}
int sizeByPriority (const PQ& pq, int priority) {...}
int numPriorities (const PQ& pq) {...}
```

When you add a new element (*i.e.,* call `enterPQ`), 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 `enterPQ` will likely be the hardest function to write.

When you remove the "top" element (*i.e.,* call `leavePQ`), 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 `sizePQ` 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 `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, `sizeByPriority` of 4 would return 1, of 3 would return 3, and of 5 would return 0.

The procedure `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.

For the counting operations, you can choose to either keep track as you add / remove elements from the priority queue, or you can pause and count them on-demand. Either approach is perfectly acceptable for full marks.

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

Consider the code for the binary search tree 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`. Create copies of this code in a file called `a4p2.cc` and then complete the package by implementing `BST_delete` as outlined in class. Give it a good workout with your testing, as you'll be re-using this code in the next part also.

Part III

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 that you need to store explicitly is their name. You should use a BST to sort the names alphabetically, and use a queue to store their arrival order. I strongly suggest that you copy over and make use of the queue and BST operations almost verbatim, and build implementations of the desired operations below in terms of these operations.

Your job here is to define the following operations:

- `void SBL_init (SBL& sbl)` — initialize the SBL data structure in the "obvious" way.
- `int SBL_size (const SBL& sbl)` — returns the number of people currently in the stand-by list; you can keep a counter to the `SBL` type if you wish.
- `void SBL_arrive (SBL& sbl, string name)` — adds a new person to the end of the SBL (as well as to the BST that is used in printing).
- `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.
- `string SBL_first (const SBL& sbl)` — return the name of the person at the front of the SBL.
- `bool SBL_lookup (const SBL& sbl, 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_printInArrivalOrder (const SBL& sbl)` — print the names of the people in the SBL in the order of their arrival, with the "oldest" printed first.
- `void SBL_printInAlphabeticalOrder (const SBL& sbl)` — print the names of the people currently in the SBL in alphabetical order.