COMP 321: Design of programming languages
Homework 1: Structures, signatures, and functors

---

1. Written problems

There are no written problems for this assignment.

2. Coding problems (10 points)

Implement a functor `ListPQFn` that constructs a list-backed priority queue implementation that is parameterized by a `PRI_KEY` structure. Recall that a *priority queue*, viewed abstractly, is a collection of keys, each of which has a numeric priority. The priority of a key is an integer, with lower numbers indicating higher priority. The same key may appear more than once in a given queue, and keys with the same priority may also appear in a single queue.

The signature describing keys for priority queues is `PRI_KEY`, which specifies a type `key` for the types of keys and a function `pri : key -> int` for computing the priority of any given key.

`ListPQFn(K)` will return a structure that implements the following exceptions, types, values and functions:

- `exception Empty`: the exception that is raised when either `delMin` or `peek` is called on an empty queue.
- `type pq`: the type of a priority queue with keys of type `K.key`. This type must be `K.key list`.
- `val empty : pq`. `empty` is an empty priority queue.
- `val isEmpty : pq -> bool`. `isEmpty`(q) = `true` if q is the empty queue, `false` otherwise.
- `val insert : pq * K.key -> pq`. `insert`(q, x) is the queue that arises from inserting x into q.
- `val delMin : pq -> K.key*pq`. `delMin`$(q) = (x, q')$, where x is a key in q of highest priority (i.e., `K.pri`$(x) \leq$ `K.pri`(y) for all y in q) and q' is the queue that results from removing x from q. Raises `Empty` if q is empty.
- `val peek : pq -> K.key`. `peek`(q) is the value x such that `delMin`(q) would return (x, q') for some q'. Raises `Empty` if q is empty.

The `ListPQFn` functor will return a structure that uses a list-backed implementation, and in particular `delMin` and `peek` must run in constant time. In all likelihood, `insert(q, x)` will run in time linear in the size of `q`. This is not intended to be an efficient implementation; the point of the assignment is to get practice writing functors. If you want more of a challenge, write a heap-backed implementation; however, you must submit your list-backed implementation (the tests assume a list-backed implementation and will fail with anything else).

You must also add tests to the testing code that I provide in the code distribution by modifying the file `tests.sml`. Do so by adding to the value `tests` (and maybe adding other definitions). Remember that `tests : string*(UnitTest.test list) list` is a list of *test suites*. A test suite consists of a name and a list of `UnitTest.test` values, each of which is returned by a call to `UnitTest.assertXXX` for some `XXX`. For full credit, you must have some tests that verify your implementation is correct when keys with the same priority are added to a queue. Since keys with the same priority could appear in any order, writing such a test that allows for any of the possible correct results is not completely trivial. *Hint: create a list of all the possible results, and check that the actual result is in that list using* `UnitTest.assertPred`.

Your code must compile and execute without errors with the shell command

```
$ make tests && ./tests
```

If not, you will receive no credit for this assignment. "Execute without errors" means that there are no run-time (programming) errors like uncaught exceptions; failed tests are not errors (though they probably indicate logic errors).

3. More on signatures and specifications

Your `ListPQFn` returns a structure that is a bit suspect, because any client who uses it will know that you have implemented your priority queues using lists. In particular, when you look at the testing code, it uses this information by, for example, applying list operations to the results of your functions. As you may know, the right thing to do is hide this information, and this can be done by ascribing a signature to the structure that `ListPQFn` returns. That signature would just specify that the structure defines some type `pq`, but the client does not know what it is. However, it turns out that there are some non-trivial complications in doing this right in SML. Done naively, the type of the keys also becomes hidden from the client, with the peculiar result that while the client can create an empty queue, she cannot add any keys to it (because the type of keys is not known to the client, calls to `insert` will never type-check). SML has appropriate syntax for dealing with this issue, but it is more detail than I wish to get into in this assignment.

4. Code distribution and submission

Beyond the usual build and testing files, the code distribution consists of:

- `pri_key.sig`: signature defintiion for `PRI_KEY`.

Your submission will consist of:

- `listpq.sml`: your `ListPQFn` functor implementation.
- `tests.sml`: your modification of the `tests.sml` file provided in the code distribution.

Any other files you submit will be ignored.