COMP26120
Academic Session: 2019-20

Lab Exercise 6: Priority Queues

The deadline for this lab is at the start of Semester 2 (end of week 1)

For this lab exercise you should do all your work in your *ex6* branch.

Learning Objectives

By the end of this lab you will be able to:

- Describe the *Priority Queue* Abstract Datatype
- Describe how rotations can be used to balance an AVL tree
- Describe how the *heap-order property* can be preserved in a Binary Heap
- Describe how elements are inserted into, and searched for, in a Skip List
- Write C code to implement the above concepts

Introduction

This lab exercise is all about the *priority queue* abstract data type. As a reminder, this ADT normally supports the following fundamental operations:

`insert(k,e):` Insert an element e with key k into P

`pop_min():` Remove and return from P the element with the *smallest* key (may not be unique). Returns a special value or causes an error if P is empty.

`is_empty():` Returns true if P is empty and false otherwise

But in this lab we also consider the following operations that is necessary for one of the applications of priority queue we consider, and is typical of any *container*-type ADT.

`contains(k,e):` Return true if P contains element e with key k and false otherwise

This lab considers different data structures that can be used for implementing this ADT.

The lab consists of two parts. The first part gets you thinking about what a priority queue is and how it should be used. The second part asks you to complete the code for a few data structures for implementing priority queues.

Part 1: Before Implementing a Priority Queue

Part 1a: Testing

We can think of an ADT as presenting a *contract* i.e. a set of behaviours that an application wants and a data structure agrees to provide. Your first task is to extend the *tests.c* file to test whether a given priority queue implementation fulfils its (implicit) contract. We have given you one test to start with and you should provide at least 5 tests and COMPjudge will try running 20.

COMPjudge will run these tests against correct and incorrect implementations (but I won't have captured all the possible incorrect behaviours). You have been given two correct implementations using linked lists already. You should exit with -1 status on a test failure and otherwise return with 0. You should keep the functionality that an unrecognised test terminates successfully. Remember to use these tests in Part 2 to make sure you've done things properly!

Part 1b: Predicting Success

In the next part you will be asked to complete the implementation of three data structures that can be used to implement a priority queue. You have already been given two data structures in the previous section. Therefore, in this lab you will be considering the following five alternative data structures:

1. Linked List
2. Sorted Linked List
3. AVL Tree
4. Binary Heap
5. Skip List

In your branch you have been given two simple applications that make use of one or more priority queue.

- **Sorting**. This application sorts a set of strings by adding a code indicating their place in the final result as the priority and then popping them in this order. This is a bit similar to how *heap sort* works, which you should look up.
- **String Search**. This application takes a set of strings and searches to see if a given string can be constructed by concatenating those smaller strings. It does this by *depth-levelled saturation* e.g. it generates all possible concatenations from smallest to largest (this is what the priority queue is needed for). It should be noted that this is unlikely to be the best way to solve this problem!

In this part your job is, in a file called *discussion.txt*, answer three questions:

1. What priority queue operations does each application require and with what relatively frequency (roughly)? e.g. *few insertions, lots of checking for contains, as many pop min operations as insertions.*
2. With what complexity does each of the five considered alternative data structures implement those operations? Your answer should be in Big-O. You don't need to justify this here but the TA may ask why - you're expected to look these up and understand them, not necessarily compute them.
3. For each application, which data structure do you expect to be best?

You should not need to write more than a few sentences for each of the bullet points.

In addition, for extra marks, you should comment on whether sorting strings by their length in the first application would impact the practical complexity (the asymptotic complexity should be unaffected). Consider what impact sorting by length (where many inputs would have the same priority) might have on the different data structures.

Part 2: Three Implementations

In this second part you need to complete three implementations of data structures that can be used to implement priority queues. Note that these data structures can also be used to implement other ADTS.

The provided implementations include TODO remarks with some hints on what needs to be done in each place. In summary, what needs to be done is as follows:

- AVL Tree
  - Implement left rotation
  - Implement the bit of insertion that uses left rotation
- Binary Heap
  - Implement the sift-up operation
  - Fix the code getting the last index
  - Fix the insertion code
- Skip List
  - Complete the search function (moving forwards in a level)
  - Complete insertion (creating a new node)
  - Complete pop_min (repairing the list)

There is not a lot of code to write, but you need to understand how the data structures (and the given implementation) work to complete these parts. You may make any changes you want to the provided data structures (you can alter parts of the code that already exist). In fact, you can start from scratch if you prefer.

You should test your code to make sure it works as expected. You can use a combination of your tests and the provided applications to do this. COMPjudge uses your submitted tests to test the code, so if you got these wrong it may give the wrong result.

Finally, add a conclusion to *discussion.txt* where you reflect on whether you were right. You should use the results of running `make run_sorting_all` and `make run_concat_all` as the basis of this conclusion but you can run further tests if you want. In addition, for extra marks, extend your discussion to comment on the course where we sort by string length alone.

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Part 1a: Are there sensible tests and do they cover the major behaviours of a priority queue?	
There are at least 5 tests, they cover the main cases of a priority queue and are well-documented. The student can explain what the tests are checking and what kind of incorrect implementation would fail the tests.	(2)
There are at least 5 tests. They might not cover all behaviours but cover most of them. The student can explain them.	(1.5)
There are at least 5 tests. Some might be flawed or there may be major behaviours not covered (e.g. the role of priorities). The students explanations may simply be at the level of what happens rather than how this relates to priority queues.	(1)
No attempt has been made or there are fewer than 5 tests	(0)

Part 1b. Have the three questions been answered for the two applications and five data structures?	
All parts are answered well. The answer is concise and clear. The student can justify the Big-O complexities when asked. There is a reasonable reason for the predictions (they do not need to be correct)	(3)
The questions are mostly answered well. There may be some parts that are unclear but overall the student can explain the general idea.	(2.5)
At least two of the points are covered well but there is a significant problem with one of the parts e.g. it is missing or incomplete. The student can explain what is present.	(2)
An attempt has been made but it is unsatisfactory in more than one place.	(1)
No attempt has been made	(0)

Part 2. AVL Tree.	
The AVL Tree has been completed well. The code is neat. The student can explain clearly how rotations work to preserve the height-balance property (including what this is).	(4)
The code has been completed correctly but the explanation of how it works is unsatisfactory	(3)
The implementation is almost correct but there is a mistake that means it does not work in all cases	(2)
An attempt has been made	(1)
No attempt has been made	(0)

Part 2. Binary Heap.	
The Binary Heap has been completed well. The code is neat. The student can explain clearly how the sifting operations work to preserve the heap-order property (including what this is) and how the array-based implementation of trees works.	(4)
The code has been completed correctly but the explanation of how it works is unsatisfactory	(3)
The implementation is almost correct but there is a mistake that means it does not work in all cases	(2)
An attempt has been made	(1)
No attempt has been made	(0)

Part 2. Skip List.	
The Skip List has been completed well. The code is neat. The student can explain clearly how the search operator finds a node in the skip list and how this is used in both the insertion and popmin operations. The student can explain what would happen if the probability of creating additional levels was much higher (e.g. 0.9) and the impact this would have on the expected average case.	(4)
As above apart from the last sentence	(3.5)
The code has been completed correctly but the explanation of how it works is unsatisfactory	(3)
The implementation is almost correct but there is a mistake that means it does not work in all cases	(2)
An attempt has been made	(1)
No attempt has been made	(0)

Part 2. Are there any memory issues?	
No (or they don't come from the students code, unlikely but possible)	(1)
Yes but they are small	(0.5)
Yes, lots	(0)

Part 2. Are the concluding remarks in the discussion reasonable	
Yes, the student has run the code and identified the best data structure for each application and then reflected on why this might be the case. The reflection does not need to be perfect but should be sensible.	(1)
The student has just stated which data structure was best witout reflection	(0.5)
No attempt has been made	(0)

Has the student considered the case where getcode in apps/sorting just returns strlen(str) in the discussion?	
There is a good discussion that identifies what will happen in this case. They will probably include some running times for this case in their conclusion. The student can explain what they wrote.	(1)
There is some non-trivial discussion but it doesn't properly explain what is going on	(0.5)
No attempt has been made	(0)