1A Lent Algorithms

Supervision 3: Data structures (queues, trees)

Hayk Saribekyan

February 7, 2019

When you are asked to describe an algorithm, also analyse its time and space complexity. Mention how you would implement the solution.

In this supervision we will concentrate on data structures. We will cover queues, stacks and search trees. There are also problems from the previous chapter on algorithm design.

This example sheet is longer than I expected but all the problems are quite important to understand. Also many are just bookwork (e.g. warm up). I do not expect you to solve and write down precise solutions to everything, so think about the things you struggle with. You can go back to them during revision or when you have time later.

I would rather see a failed attempt or ideas handed in, than a beautifully written correct solution of a problem that you found easy. I do not mind if you skip a problem and write "I know how to do it." So focus on the concepts that you have most trouble with rather than writing down solutions to problems that are easy for you.

1 Refresher (Algorithm Design)

This is material from the previous supervision.

1. You are given positions of n mice and n holes on the x axis. A mouse moves from position x to $x \pm 1$ in 1 minute. Assign each hole to exactly one mouse in a way that minimises the time it takes the mice to reach their hole. Prove the correctness of your algorithm.

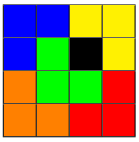
2. You are given a $2 \times n$ board. In how many ways is it possible to cover the board with n dominoes, if each domino should cover exactly two adjacent cells of the board?

   Explain your solution and give an implementation in any language.

3. Given a matrix of integers of size $n \times m$. Each row and column is sorted in ascending order, from left to right and from top to bottom, correspondingly. Find the location of a given number x.

   Bonus: Prove that no algorithm can solve it faster than in $\Omega(n)$ time if $n = m$.

4. Given an $n \times n$ board, where $n = 2^k$. The cell at (i, j) is removed. Describe an algorithm that will cover the remaining cells of the board with L-shaped tetris pieces. Below is a tiling for $n = 4$. The black cell is removed.



Hint: $n = 2^k$ *divides by two very nicely many times!* correct

2 Warm-up (Data Structures)

1. (Robert Harle) Discuss the (dis)advantages of linked-list and array based implementations of queues.

2. (Robert Harle) Consider implementing a stack using an array. When the array is full and a push() is requested, there are two common strategies to growing the array: increase the array by a constant number of elements or double the size of the array. Analyse both strategies to find the amortized costs associated with a push() operation.

3. Using the sequence $\{10, 85, 15, 70, 20, 60, 30, 50, 65, 80\}$ draw:

   (i) the BST tree

   (ii) the 2-3-4 tree

   (iii) the red-black tree

4. (Robert Harle) The main issue with BSTs is their tendency to become unbalanced. This can be a particular issue if the input keys have a lot of duplicates (e.g. insert $\{1, 1, 1, 1, 1, 1, 1\}$ gives a very unbalanced tree. Suggest a way in which duplicate entries in a BST could be addressed such as to produce a more balanced result.

3 Examples

1. (Robert Harle) Write a function (in Java/Python/Pseudocode/etc) that tests whether a string is a palindrome (the same backwards as forwards) using only the standard operations of a stack.

2. (Robert Harle) A *table* is a set of key-value pairs (e.g. dictionary in Python, map in Java), where $set(k, v)$ means to map the key k to value v. Usually there is also a $get(k)$ function, which returns the corresponding value v.

   Write a pseudocode of the $set(k, v)$ function for the linked-list based table. Duplicate elements are not permitted i.e. $set(k, v_1)$ and $set(k, v_2)$ operations would result only in one element in the table with key k with value v_2.

3. Given a string of parentheses, brackets and braces e.g. `(]{{]])]`. Describe an algorithm that decides whether the given string is a valid. The string above is not whereas `[(){[]}]` is.

4. Explain how to implement a queue using only stacks - no arrays, linked lists or any other data structures are allowed. Of course, you can use individual variables (e.g. to store sizes and indices).

   We want our queue to be efficient i.e. on average each operation that a queue supports should take $O(1)$ time. Notice that the naive implementation described below is not efficient.

   Naive implementation: To implement a queue `q`, keep a stack `s`. When `q.push(x)` is called, put the element on top of the stack `s` i.e. `s.push(x)`. On `q.front` operations, reverse `s` into another stack `s'` and return the top element of `s'`. Then, reverse the stack back into `s`.

5. Implement a function `Node predecessor(Node node)` or its equivalent in Python that returns the predecessor of `node` in BST. You can assume that the class `Node` has whatever standard fields a BST would have.

   What would you change in the code to fund the successor?

6. Given a BST, write code that prints the elements in the tree in sorted order. Explain how it is different from heap-sort.

7. Initially you have an empty set S. You receive a stream of queries of two types:

   - `Insert(x)`: you should add x to S.
   - `Range(a, b)`: return the number of elements in S in the that are in the range $[a, b]$.

   You should process all the queries in $O(\log |S|)$ time.

4 Implementation (Bonus)

Implementations are critical to sharpen your understanding of data structures.I would suggest that you work on these during the break if you do not have time now. Once you have implemented a RB-tree and have fallen in all the traps yourself, you will never make those mistakes again.

In online judges, do not use standard packages provided by your language. Instead implement the data structures yourself.

1. Many of the covered data structures use a *tree-rotation* operation to keep the trees balanced. The rotations are easy to understand but surprisingly hard to implement. Splay trees[1] are possibly the simplest of such trees because they do not keep any other information per node (e.g. color). All you need to do is few rotations.

   Try implementing Splay trees (based on the description from Wikipedia) and make sure they work. Solve problems from online judges to make sure your implementations are correct.

   Petar Velickovic goes into some details of how to do this[2].

2. `http://www.spoj.com/problems/RMQSQ/`

[1] `https://en.wikipedia.org/wiki/Splay_tree`
[2] `https://www.cl.cam.ac.uk/~pv273/supervisions/Algorithms/Algs-3.pdf`