CSE250, Spring 2022 Assignment 5 Due Wed. Apr. 20, 11:59pm

Lectures and Reading:

The **Second Prelim Exam** will be on **Friday, April 29**. It will cover Chapters 12, 13, 15, 16, and 18, and focus on assignments through that date.

Reading: For next week, finish Chapter 16. Contrary to how I have skimmed over long code examples in the text before, study the code in section 16.3 (pages 506–510) closely. Similar code in the ISR framework would avoid such a hard separation of the notions of "key" and "value" and thus might be simpler in parts. Then skip Chapter 17 (which might support a compilers course), and go into Chapter 18. While reading 18, hark back to chapter 13, while noting that 18 gives the signature implementation of a priority queue. The topics we will emphasize regarding Chapter 18 are (i) how a heap enables compiling "top-ten (or so)" lists in less time than the order-$n \log n$ needed for sorting, and (ii) how to quickly fix up a heap if the priority value of one item suddenly changes.

——— *Assignment 5, due Wed. Apr. 20 "midnight stretchy" as PDF on CSE Autograder* ———

(1) Let A be a data structure composed of a size-m array `arr` of linked lists, each of size r. Let B be a data structure composed of a linked-list `lis` of r nodes, such that each node `u` holds an array `u.arr` of size m. Both data structures have $n = mr$ data items that are kept globally sorted by key.[1] The three main "ISR" operations have the following algorithms and asymptotic running times:

- `A.find(item)`: Do binary search on `arr` to find i such that `arr(i).head.key <= item.key < arr(i+1).head.key`. It is vital to observe that the implementation of binary search never evaluates the variable `right`, so it does not bomb if `i = m-1` so that `i+1` is out of range of the array—we can treat the algorithm as though that value were $+\infty$. The implementation does not return `i = m` out of range, so the final `i` is always a valid list. Then do sequential search of that list, giving time $\Theta(\log(m) + r)$.

- `B.find(item)`: Do linear search to find the last node `u` such that `u.arr[0].key <= item.key`. Then do binary search of that array. This gives time $\Theta(r + \log(m))$, which in this notation is the same time as for data structure `A`.

- `A.insert(item)`: Call `findPlace(item)` to find where the item should go—the only difference from `find` being that it always returns an iterator to the location to insert before, rather than `end` when no item matching `item.key` is found. Inserting into the linked list at that point is just $O(1)$ extra time, so the overall time stays $\Theta(\log(m) + r)$. Same for `A.remove(item)`.

[1] In A this means: each linked list is sorted in nondescending order, and if $0 \le i < j <$ `arr.size` then every item in the list `arr(`i`)` has key $\le$ the key of every item in the list `arr(`j`)`. In B this means each array is sorted in nondescending order, and if node u comes before node v in the linked list, then every item in the array for node u has key $\le$ the key of every item in the array for node v. Pictures of these two data structures are at the end of the lecture notes https://cse.buffalo.edu/ regan/cse250/CSE250Week8.pdf]

- `B.insert(item)` and `B.remove(item)`: Same $\Theta(r + \log(m))$ time to find where the item is or should go, but inserting or removing in the middle of a size-m array takes time $\Theta(m)$. Since $\Theta(m)$ has higher asymptotic order than $\Theta(\log m)$, the time is now $\Theta(r + m)$.

The questions are all about: when you have n data items, how do you want to choose r and m subject to $rm = n$ in order to minimize the time needed in the following situations. You may suppose $rm = n$ exactly; in practice it does not matter if the lengths of the individual linked lists in A, or the individual arrays in B, differ by a few items. The choice of how r relates to m, keeping their product the same, is called a **tradeoff**. In all cases, you are defining $r = r(n)$ and $m = m(n)$ as functions of n, subject to their product being n, to minimize times like $\Theta(r + \log(m))$ and $\Theta(r + m)$. A fact you may find helpful is that $\log(\frac{n}{\log n}) = (\log n - \log\log n)$ has the same asymptotic order as $\Theta(\log n)$.

(a) Your application mainly uses just `find`, like the task of Assignment 6. (Note: regarding B, the `BALBOA` and `BALBOADLL` implementations are currently configured optimally this way, but regarding A, `AIOLI` and `AIOLISLL` are quite the opposite.)

(b) Your application does a lot of insertions and removals, and you are using data structure A. What is the optimal tradeoff?

(c) Your application does a lot of insertions and removals, and you are using data structure B. What is the optimal tradeoff now?

Also answer the following: In terms of asymptotic order, a data structure that gives time $\Theta(\sqrt{n})$ for an operation is slower than one giving time $\Theta(\log n)$. What if, however, the concrete times—putting principal constants in the same units of time—are $2\sqrt{n}$ in the former case and $8\log_2(n)$ in the latter. Find the maximum value of n below which the former data structure is concretely faster. (Here $\log_2$ means log to base 2. Points are $6 + 6 + 6 + 9 = 27$ pts.)

(2) Modify the nonrecursive form of breadth-first search, given as `classicBFS` in my notes https://cse.buffalo.edu/~regan/cse250/CSE250Week9MWF.pdf, so that when the `goal` node is found, the output also gives you a shortest path from `start` to `goal`. Use tuples so that whenever a node `v` is visited, the first node `u` that `v` is visited *from* is preserved in the tuple. Then say why the set of tuples you get efficiently gives you a path from `goal` back to `start`, which you can then reverse to get your answer. (When `goal` is not found, of course, you get no such path.)

[For some important chitchat, the text's algorithm on page 481 accomplishes this task—even when the edges have arbitrary weights—but is not efficient, as the text admits below it. Put more bluntly, the algorithm is (IMPHO) *bad*, and if you find it hard to read, well so do I. The same remarks extend to the maze-specific code on page 478. Your algorithm will, however, work correctly only because all the edges have the same unit weight. The footnote on page 481 takes the higher road, but *Dijkstra's algorithm* is properly a subject for CSE331, and in applications where you don't have different weights, it's overkill.]

Your answer need not be executable Scala code, but it should be "Scala-like" including the use of sets of tuples (or alternatively, a `Map`) and the `Queue[Node]` data structure. (27 pts., broken as 18 for code and 9 for explanatory comments, giving 54 on the set.)