CSE250 Assignment 5 Answer Key Spring 2022

(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)`.
- `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.)

[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) 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.)

Answer: For (a), choose $r = \log(n)$ and $m = n/\log(n)$. (It doesn't matter to the asymptotic complexity whether the base of the logarithm is 2 or e or whatever; moreover, the choices $r = 1$ and $m = n$, and any choice of r and $m = n/r$ in-between, give the same asymptotic answers. Then:

- $\log(m) = \log(n/\log n) = \log(n) - \log\log(n) = \Theta(\log n)$.
- $r = \log(n) = \Theta(\log n)$.

So the time order-of $\log(m) + r$ becomes order-of $(\log n + \log n) = \Theta(\log n)$. This is the best possible, since it comes when the contributions from $\log(m)$ and r are asynptotically equal and so balance each other. The choices $r = 1$ and $m = n$ (or etc. as above) don't give such a pure $\Theta(\log n)$ balance but they likewise give time $\Theta(\log n)$ overall. The "BALBOA" data structure as-configured gives $r = 1$ and $m = n$, since it builds just one array of size n, but that's optimal for a task like Assignment 6 which involves mostly just `find`. The "AIOLI" data structure currently has just one linked list with no middle access points, so it effectively gives $r = n$ and $m = 1$, which is the opposite of what we want here.

(b) When you build A with $r = O(\log n)$ and $m \approx n/\log(n)$, i.e., a biggish array of short linked lists, you not only get the same time for `find` in (a), but it also costs you only $O(1)$ time to add or remove from one of the linked list (or one of the marked-off segments of the one linked list, if the alternate implementation shown in the notes is used). So the same choices as in (a) give you $O(\log n)$ overall time for insertion or removal.

(c) With B, however, insertion has to be into one of the size-m arrays. An insertion near the middle always forces $\Theta(m)$ amount of work moving the succeeding elements up—or forcing a resizing and recopying of the entire array. Likewise a removal from the middle—or anywhere but the end, for that matter. So the time becomes order-of $r + \log(m) + m$, which just becomes order-of $r + m$. Subject to the product being n, the best balance is given by choosing $r = m = \sqrt{n}$. This gives $r + m = \Theta(\sqrt{n})$ as the optimal time. [This is worse than (b) but still works pretty well in practice.]

(d) We want to equate $2\sqrt{n}$ with $8\log_2(n)$, which simplifies to equating $\sqrt{n}$ with $4\log_2(n)$. We can compare values on some even powers of 2:

n	$\sqrt{n}$	$4\log_2(n)$
64	8	24
256	16	32
1024	32	40
4096	64	48

.

The values in the columns have "crossed over" so the optimum n is somewhere between 1024 and 4096. Binary search—done manually with a calculator—first tries $n = (1024 + 4096)/2 = 2560$. That gives $4\log_2(n) = 45.2877\ldots$ and $\sqrt{n} = 50.59644\ldots$. The first column is bigger, so we try going down: $n' = (1024 + 2560)/2 = 1792$. Now $4\log_2(n') = 43.2294\ldots$ and $\sqrt{n'} = 42.332\ldots$. Now the second column is bigger, so we go up: $n'' = (1792+2560)/2 = 2176$. Continuing in this vein soon finds $n = 1897$ as the greatest n for which the first column is less. For $n = 1898$ and above, the $8\log_2(n)$ time becomes concretely faster than the $2\sqrt{n}$ time.

Footnotes: The similarity with the 1898 year of the Samuel Fallows dictionary is pure accident—I intended $n = 256$ to be the answer, as you would get if it were $2\sqrt{n}$ versus $4\log_2(n)$. The fact that doubling the 4 to 8 moved the "crossover point" to be much more than double 256 is worthy of special note. Just a little worsening of the *principal constant* of the theoretically optimal implementation can move its tradeoff point versus "quick-and-dirty" code much higher. With many published algorithms, the crossover is well above billions, a fact which Richard Lipton and I called attention to under the heading "Galactic Algorithms."

(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.)

Answer: The basic idea is that whenever we find a new neighbor v of a node u, we store a tuple (v, u) or a Map element $(v \to u)$. The algorithmic invariant is that the node u is always closer to `start` than v is—else we would have already visited v. The neighbor v has to be freshly new—so this causes us to move up the test for whether v has been already expanded or visited. Here is the code, with some redundancy showing how you could use either a set of tuples or a `Map` (note, this is not from actual executable code but intended as pseudocode):

```
def classicBFS(start: Node, goal: Node): List[Node] = { //empty list == not found
   if (start == goal) { return List(start) }
   //else
   var visited = new Queue[Node]() //Scala Queue is mutable by default
```

```
   var backSteps = new Set[(Node,Node)]   //(to, from)
   var backMap = new Map[Node,Node]()    //(to -> from)
   var expanded = new scala.collection.mutable.Set[Node]()
   visited.enqueue(start) //so queue is nonempty
   while(!visited.isEmpty) {
      val u = visited.pop()
      expanded += u
      for (v <- u.nbhrs) {
         if (!(expanded.contains(v) || visited.contains(v))) { //v is freshly new
            backSteps += (v,u)
            backMap += (v -> u)    //or: backMap(v) = u
            if (v == goal) {
               { return the traced out path }
            } else {
               visited.enqueue(v)
            }
         } else {
            //do nothing
         }
      } //end of loop over neighbors of u
   } //exit of while loop means no more fresh nodes to visit/expand
   return List[Node]()
}
```

Returning the traced-out path is easier if you've defined a Map, or if your organization of set-of-tuples is more flexible than than Scala `Set` per-se. With the `Map`, the trace-out code is:

```
   var ret = List(goal)
   var y = goal
   do {
      y = backMap(y)
      ret ::= y
   } while (y != start)
   return ret
```

Because we prepend, the List actually comes out in the desired start-to-goal order. The number of iterations of the do-while loop is at most the length of the shortest path from `start` to `goal`, rather than be upwards-of the total number of nodes like in the original BFS. The loop body just has one prepend—which is stipulated to take $O(1)$ time for a `List`—and one evaluation of the `Map`. The time for one evaluation is theoretically no worse than $O(\log n)$ time with a well-sorted `Map`, and with the hash-table implementation it counts as $O(1)$ time but with a little fine-print. So this is efficient.

The set of tuples (v, u) has the property that every value v appears at most once. So if we can efficiently find it, then we can get the corresponding u and repeat just like above with the Map. The issue is that `Set[(Node,Node)]` forces us to know the whole tuple in advance in order to find it. We really want to do something like finding `(v,_)` but reading rather than ignoring the second component. Well, if your tuple is a simple class like on Assignment 4, and you define `keyComp` to use the first part, then "ISR" will find u for you just like in a Map.