Imperative Programming (Parts 1&2): Sheet 6

Joe Pitt-Francis

Most questions by Mike Spivey, adapted by Gavin Lowe

Final sheet, suitable for around Week 7, Hilary term, 2019

- Any question marked with † is a self-study question and an answer is provided at the end. Unless your tutor says otherwise, you should mark your attempts at these questions, and hand them in with the rest of your answers.
- Questions marked [**Programming**] are specifically designed to be implemented. Unless your tutor says otherwise, provide your tutor with evidence that you have a working implementation.
- You should give an invariant for each non-trivial loop you write, together with appropriate justification.

Question 1

Each bucket in a hash table represents a mapping from certain keys (all those that have a certain hash code) to values. In our implementation of hash tables, each bucket is represented as an unordered linked list, with a time for lookup of $O(N)$, where N is the length of the list. Wouldn't it be a good idea to replace these linked lists with some other data structure that represents a mapping with better asymptotic complexity?

Question 2 †

Suppose a hash table with N buckets is filled with n items, and assume that the values of the hash function for the n items are independent random variables uniformly distributed in $[0..N)$. What is the probability distribution for the number of items that fall in a particular bucket? What is the expected number of comparisons of keys that are performed when `get` is called and the search is (a) successful and (b) unsuccessful?

Question 3

The hash tables we have been studying use chains of dynamically allocated records to deal with the collisions that occur when two keys have the same hash code. This scheme has the advantages that it is easy to implement, and has a performance that degrades only gradually when the table contains many records.

An alternative that avoids dynamic allocation is to use a fixed array of records `table[0..MAX)`. If a key `k` hashes to `h`, and slot `table[h]` in the array is already occupied by another key, then slots `table[h+1]`, `table[h+2]`, ... are tried until we find either the key `k`, or an empty slot (meaning that `k` is not present); the search wraps round at the end of the array. This idea is called "closed hashing" (or "probing" in the lectures).

Write an implementation of the 'bag-of-words' module based on this idea, with operations to add a word, find the count of a word, and to delete a word. Think carefully about how to implement deletion. Give a suitable datatype invariant. Your implementation does not need to be able to store more than `MAX` entries (so you can ignore the issue of re-sizing).

Question 4

[**Programming**] It would help with visualizing the structure of a binary tree to print it out in prefix order (where the root of a tree precedes its left and right subtrees), using indentation to show the structure. For example given type

```
// Class defined as "case" to save you writing "new Tree(...)"
case class Tree(var word: String, var left: Tree, var right: Tree)
```

the tree

```
var tr = Tree("three", Tree("four", Tree("five",null,null), Tree("six",
  Tree("seven", Tree("one",null,null), null), null)), Tree("two",null,null))
```

would be printed as follows

```
three
 . four
 .  . five
 .  .  . null
 .  .  . null
 .  . six
 .  .  . seven
 .  .  .  . one
 .  .  .  .  . null
 .  .  .  .  . null
 .  .  .  . null
 .  .  . null
 . two
 .  . null
 .  . null
```

(a) Write a recursive procedure that prints a tree in this way.

(b) Now re-write the procedure to use a stack instead of recursion.

Use code from the binary tree example in the lectures as a model if you wish. You might find it useful to know that if `st` is a string, the expression `st * n` will produce a string containing `n` copies of `st` concatenated together.

Question 5

[Programming] Given the class of tree nodes:

```
case class Tree(var word: String, var left: Tree, var right: Tree)
```

Write a procedure with heading,

```
  def flip(t: Tree) : Unit
```

that (destructively) flips the tree `t`, exchanging left and right throughout.

Question 6

Suppose we want to know the total number of words in our binary tree representation of a bag of words, i.e. the sum of all the `count` fields on all the nodes of the tree. Of course, we could do this by adding a variable that keeps track of this number (satisfying a suitable datatype invariant); but let's ignore this possibility for the sake of this question.

(a) Write a recursive function to calculate this quantity.

(b) Now write an iterative function to do the same thing, using a stack to keep track of the parts of the tree still to be considered; give a loop invariant and a variant for your code.

Question 7

In the `BinaryTreeBag` example we have not considered what we might do to prevent the tree from getting too unbalanced. Suppose that we wished to, at the very least, *monitor the extent* to which a binary search tree has become unbalanced. One way to do this would be to report on the minimum depth of any leaf node from the `root`, and the maximum of any leaf node from the `root`.

(a) Write a single recursive function to calculate the minimum and maximum depths of the leaves in a subtree. Use it to report on the minimum and maximum depths in the entire search tree.

(b) Now write an iterative function to do the same thing, using a stack to keep track of the parts of the tree still to be considered; give a loop invariant for your code. Hint: your stack(s) can store both a subtree and its depth from the `root` in the original tree.

Question 8

[**Programming**] Suppose we want to find all the anagrams of a given word. This question investigates two ways of doing this.

(a) The first approach generates all distinct permutations of the given word, then selects those that are in a suitable dictionary. Implement this approach.

You might want to look at the API documentation for the class `scala.collection.immutable.StringOps`; all the operations in this class can also be applied to a `String` (via a piece of Scala magic called implicit conversion; see Section 6.12 of *Programming in Scala*). `StringOps` contains an operation `permutations` that produces all permutations of the `String`; however, that makes the question too easy, so try writing your own version.

You'll probably find that your program is rather slow on long strings. Explain why.

(b) If we want to find anagrams of longer words, or if we want to find lots of anagrams, it is worth creating a suitable data structure, which we'll call an *anagrammatical dictionary*[1]. This is basically an array containing all the words, but ordered such that anagrams are consecutive. We can create this anagrammatical dictionary by pairing each word with its sorted permutation, and then sorting the array according to those sorted permutations. The first few entries (using dictionary file `knuth_words`) will be

```
(a,a)
(aaaaabbcdrr,abracadabra)
(aaaaacgillmmnrty,anagrammatically)
(aaaaacgilmmnrt,anagrammatical)
(aaaaagiimmnnorttz,anagrammatization)
(aaaabcceelrtu,baccalaureate)
(aaaabcchiln,bacchanalia)
(aaaabcchilnn,bacchanalian)
(aaaabcdiillty,adiabatically)
(aaaabdeehiiiiilmmnnnrsssttt,antidisestablishmentarianism)
```

All of those have distinct sorted permutations, so none are anagrams of others. However, further down the dictionary are the entries

```
(aabceilrt,bacterial)
(aabceilrt,calibrate)
```

showing that "bacterial" and "calibrate" are anagrams of one another. To find all anagrams of a given word `w`, we just search the anagrammatical dictionary for those entries paired with the sorted version of `w`.

Create an anagrammatical dictionary in this way (hint: `s.sorted` will return a sorted version of `String s`, and `a.sorted` will return a sorted version of array `a`.) Use it to find anagrams.

[1] This idea comes from *Programming Pearls* by Jon Bentley, Chapter 2.

Optional: find the longest words that are anagrams of one another; find the largest set of words that are anagrams of one another.

Answer to question 2 †

The events of the k'th item falling in the first bucket for each k are independent Bernoulli trials with probability $p = 1/N$, so the number of items falling in the first bucket (or in any bucket) has the binomial distribution $B(n, p)$ with mean n/N. For an unsuccessful search, we will examine every item in the bucket, so the number of comparisons will have just this distribution.

To find the expected number of comparisons for a successful search, it is possible to use a number of methods, some more rigorous than others. One reasonably rigorous method is to calculate the expected total cost of retrieving all keys, then divide this by the number of keys to get the cost of retrieving a random key, all keys being equally likely. If a bucket contains k keys, then the cost of retrieving all of them one at a time is $1 + 2 + \ldots + k = k(k + 1)/2$. So the expectation for the cost C of retrieving all of a bucket containing X items is

$$E(C) = \sum_k \tfrac{1}{2}k(k+1)P(X = k) = \tfrac{1}{2}(E(X^2) + E(X)).$$

Since X has the binomial distribution $B(n, p)$ we have $E(X) = np$ and $E(X^2) = V(X) + E(X)^2 = npq + n^2p^2 = np(np + q)$. So

$$E(C) = \tfrac{1}{2}np(np + q + 1) = np(\tfrac{1}{2}(n - 1)p + 1).$$

To obtain the total expected cost for all buckets, we multiply by $N = 1/p$, and to get the mean cost per key retrieved, we divide by n, to obtain the answer

$$\frac{NE(C)}{n} = \frac{n-1}{2N} + 1.$$

Another plausible argument is that, given that the desired key hashes into a certain bucket, the number of other keys in the same bucket has distribution $B(n - 1, p)$ with mean $(n - 1)p = (n - 1)/N$. We expect on average to examine half of these keys before finding the one we want, so the expected number of comparisons is $(n - 1)/2N + 1$.