CPSC 221: Algorithms and Data Structures
Assignment #3, due Wednesday, 2014 April 2 at 17:00 (5pm) PDT

Submission Instructions

Type or write your assignment on clean sheets of paper with question numbers prominently labeled. Answers that are difficult to read or locate may lose marks. We recommend working problems on a draft copy then writing a separate final copy to submit. *Be sure to use the provided cover page!* Finally, **staple** your submission's pages together! We are not responsible for lost pages from unstapled submissions.

You may work in pairs, but not groups of three or more.

Submit your assignment to Box 35, in room ICCS X235. The deadline is 17:00 (5pm) on Wednesday, 2014-April-2. We will attempt to mark your assignment before the final exam, but in any case, we will release a sample solution shortly after the deadline. **Late submissions are not accepted.**

Note: the number of marks allocated to a question appears in square brackets before the question number.

Questions

[20] 1. The Best and Worst of Floyd or Not

Download the files `binary_heap.cpp`, `main.cpp`, and `Makefile` from the assignment page. These are very closely based on the files you used in Lab 5 on heaps. The changes are that we have provided an implementation of `swapUp` for you, we modified the implementation of Floyd's `heapify` function to be slightly slower (but asymptotically the same), and we have given you a new function that resembles Floyd's `heapify`, called `heapify2`. We have also changed the main testing program to fit the needs of the later parts of this problem.

(a) We've seen in lecture that Floyd's heapify algorithm runs in worst-case $\Theta(n)$. Now, study the code for `heapify2`, which looks a lot like the code for `heapify` (which implements Floyd's algorithm). What is the asymptotic big-Θ worst-case running time of `heapify2`? Briefly explain your answer, but you don't have to provide a formal derivation.

(b) What is the asymptotic big-Θ **best-case** running time of Floyd's `heapify`? Briefly explain your answer, but you don't have to provide a formal derivation.

(c) What is the asymptotic big-Θ **best-case** running time of `heapify2`? Briefly explain your answer, but you don't have to provide a formal derivation.

(d) Edit the `main.cpp` file so that `REPS` is set to be 1,000,000. You may need to adjust this number a bit. Comment out the calls to `cout` and `printList` inside the loop. Make sure the for-loop labeled "Worst Case" is uncommented, and the next line, labeled "Best Case" is commented out. Similarly, uncomment the call to `heapify` and comment out the call to `heapify2`. Adjust the value of `REPS` if necessary so that the runtime when $n = 100$ is around 1 to 5 seconds. What machine are you running on, and what value of `REPS` did you end up with?

(e) Now, without changing the code, time it with n equal to 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. What times did you get?

(f) Now, edit `main.cpp` so that you comment out the for-loop labeled "Worst Case" and uncomment the for-loop for "Best Case". Do not make any other changes, and time the code for the same set of values for n (namely 100, 200, ..., 1000). What times did you get?

(g) Now, edit `main.cpp` so that you are using `heapify2` instead of `heapify` (comment out the call to `heapify` and uncomment the call to `heapify2`). Repeat the timings. What times did you get?

(h) Now, edit `main.cpp` so that you are using the "Worst Case" for-loop instead of the "Best Case" for-loop, but do not make any other changes (so you are still using `heapify2`). Repeat the timings. (You may stop at $n = 500$ if you'd like.) What times did you get?

(i) Plot the timing results. Do these match your expectations and the theoretical runtimes?

[20] 2. An Iterative QuickSort

Download the file `qsortCount.cpp` from the *assignment* page. This is based on the file of the same name from Lab 7, but it's modified to add a new function `iter_quicksort` (and some associated declarations), which is supposed to be an iterative version of Quicksort. Your job is to prove that it's correct. (By the way, you should probably try it out before trying to prove that it works!)

Your proof must be a loop invariant proof. To make things easier, we will give you a loop invariant that will work:

> The invariant is that if we ignore the ordering within intervals on the stack, the list is sorted. Formally, for any array indices i and j between 0 and `NN-1` inclusive, if they are not both contained in the same interval on the stack, then if $i < j$, that implies that $x[i] \leq x[j]$.

However, you need to prove that this really is an invariant of the outer loop, and then you'll need to do a bit more reasoning to prove that the code really does sort correctly.

For your proofs, you may assume that the partitioning code (which is identical to what you had in your recursive `quicksort` function) works correctly. Formally, after that code runs, all the elements in $x[a], \ldots, x[m-1]$ are less than or equal to $x[m]$, and all the elements in $x[m+1], \ldots, x[b]$ are greater or equal to $x[m]$.

(Hint: The proof is actually easy! The hard part is understanding all of the formal definitions above. Take the time to make sure you understand what they all mean before trying to do the proof.)

(a) First, prove the base case. Note that **this must be a loop invariant proof**, so your base case should **not** be about when the array is empty, or $n = 0$, or a tree is NULL, or some other kind of induction. (Hint: This will be very short and simple.)

(b) Now, prove the inductive case. (Hint: Again, this is a loop invariant proof, so the induction is based on going through the loop body one more time. As noted already, you may assume the partitioning code does the right thing. There are two paths through the loop body, so this step will likely have two cases.)

(c) Prove that the loop will terminate. (Hint: Think about what portion of the array is covered by intervals on the stack.)

(d) Use the fact that the loop will terminate combined with the loop invariant to prove that when the function terminates, the entire array is sorted.

[6] 3. Basic BSTs

The next several problems are about various data structures for the Dictionary ADT. In each of these, we will give you a series of inserts and deletes, and ask you to draw the data structure at specific points in time.

For this problem, we will consider ordinary binary search trees (without any rebalancing operations). When a two-child node is deleted, replace it with its predecessor.

The sequence of operations is: insert(10), insert(20), insert(30), insert(25), insert(15), insert(17), delete(30), delete(20).

Draw the tree after the insert(17), and the two deletes.

[7] 4. Advanced AVL Trees

Now, we will consider AVL trees. When a two-child node is deleted, replace it with its predecessor (rather than its successor, which is the other choice).

The sequence of operations is: insert(10), insert(20), insert(30), insert(40), insert(25), insert(23), insert(5), delete(25).

Draw the tree before and after each time a rotation is needed (and indicate what operation caused the need for the rotation(s)). If a double rotation is needed, show the tree before the rotations, after the first rotation, and then after the second rotation.

[10] 5. B+ B+Trees

Let $M = 3$ and $L = 3$. (How many keys are in the internal nodes? 2!) When a node underflows, it will first try to borrow from its left sibling (if it exists), and then from its right sibling. If a node must merge, it will merge with its left sibling (if it exists), otherwise it will merge with its right sibling. When a node splits, divide keys evenly between the left and right; if there is an uneven number of keys to be divided, give the extra key to the left.

The sequence of operations is: insert(10), insert(20), insert(30), insert(40), insert(25), insert(23), insert(50), insert(60), insert(70), delete(30), delete(40).

Draw the tree after any operation that causes splitting, borrowing, or merging.

[14] 6. Hash Tables with Chaining

Consider a hash table of size 7 that uses chaining (with unsorted linked-list chains, recently inserted items inserted at the front of the chain). Hash values by modding by the table size.

The sequence of operations is: Insert(10), Insert(20), Insert(30), Insert(40), Insert(54), Insert(64), Insert(76), Insert(80), Insert(90), Delete(64), Delete(54), Insert(108).

Draw the hash table after every insertion that collides, every deletion, and the last insertion (whether it collides or not).

[16] 7. Hash Tables with Double Hashing

Consider a hash table of size 11 that uses open addressing with double hashing. The first hash is modding by the table size. The second hash is $h_2(n) = 5 - (n \mod 5)$.

The sequence of operations is the same as in the preceding problem: Insert(10), Insert(20), Insert(30), Insert(40), Insert(54), Insert(64), Insert(76), Insert(80), Insert(90), Delete(64), Delete(54), Insert(108).

Draw the hash table after every insertion that collides, every deletion, and the last insertion (whether it collides or not).

[7] 8. Making a Hash of Hash Tables

Show what happens if we try the preceding problem (open addressing, double hashing), but with a hash table of size 20. The first hash is modding by the table size. The second hash is $h_2(n) = 5 - (n \mod 5)$.

Draw the hash table after each insertion that collides. Go as far as you can until an inseration fails, and describe the failure.