CPSC 221: Algorithms and Data Structures
Assignment #2, due Friday, 2014 June 20 at 17:00 (5pm) PST

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.

Each submission should include the names and student IDs of the authors at the top of each page. (You are strongly encouraged to work in pairs, but may not work in groups of three or more. Each pair submits a single copy of the assignment.) On your first page, also sign the statement "I have read and complied with the CPSC 221 2014S1 academic conduct policy as posted on the CPSC 221 course website." (See: `http://www.ugrad.cs.ubc.ca/~cs221/2014S1/syllabus.shtml#conduct`.) In keeping with the policy, you should also acknowledge on your first page any collaborators or resources that helped you with the assignment. Finally, staple your submission's pages together! We are not responsible for lost pages from unstapled submissions.

Submit your assignment to Box 31, in room ICCS X235. Late submissions are not accepted.

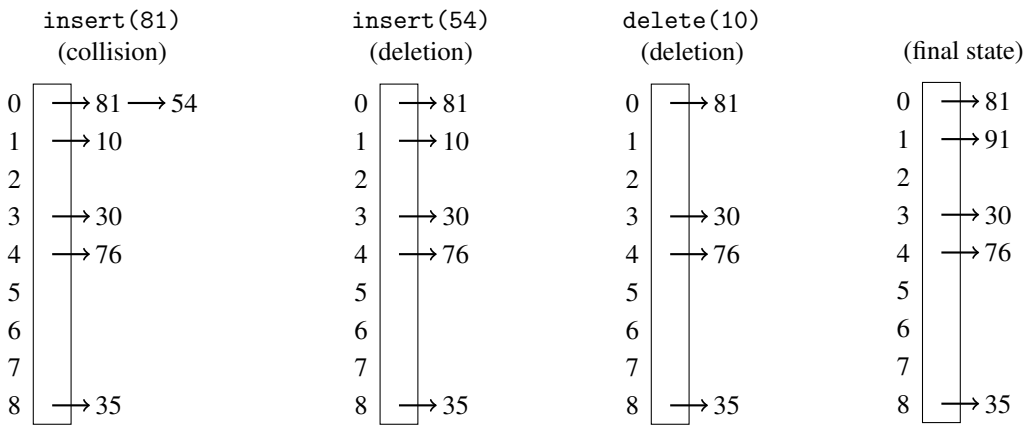
Questions

[8] 1. Hash Tables.

(a) Consider a hash table of size 9 that uses chaining. Its hash values are modded by the table size ($h(n) = n \mod 9$), the linked-list chains are unsorted, and items are inserted to the front of the chain. Observe the following operations: `insert(10)`, `insert(30)`, `insert(35)`, `insert(54)`, `insert(76)`, `insert(81)`, `delete(54)`, `delete(10)`, `insert(91)`.

Draw the hash table after every insertion that collides, every deletion, and the final state of the table. Do NOT simply draw the table after every operation.

Solution:



(b) Consider a hash table of size 11 that uses open addressing with double hashing. The first and second hash functions are

$$h_1(n) = (n+3) \mod 11$$
$$h_2(n) = 5 - (n \mod 5)$$

Observe the following operations: `insert(10)`, `insert(25)`, `insert(40)`, `insert(41)`, `insert(50)`, `delete(25)`, `delete(10)`, `insert(73)`, `insert(82)`.

Draw the hash table after every insertion that collides, every deletion, and the final state of the table. Do NOT simply draw the table after every operation.

Solution: Assuming we use -1 to symbolize a tombstone entry:

	`delete(25)` (deletion)	`delete(10)` (deletion)	`insert(73)` (collision)	(final state)
0	41	41	41	41
1			73	73
2	10	-1	-1	-1
3				
4				
5				
6	-1	-1	-1	-1
7				
8				82
9	50	50	50	50
10	40	40	40	40

(c) Consider the final state of the hash table in (b). What would happen if we ran the operation `insert(43)`?

Solution: If we ran `insert(43)`, the first hash function would return 2. Looking at bucket 2, we find a tombstone. Since it's okay to replace a tombstone with a value, 43 is placed into bucket 2 of the hash table.

[13] 2. AVL Trees.

Let $N(h)$ be the smallest number of nodes in an AVL tree of height h. For example, $N(0) = 1$ and $N(1) = 2$. Prove that $N(h) = F(h+3) - 1$ where $F(i)$ is the ith Fibonnacci number ($F(0) = 0$, $F(1) = 1$, and $F(i) = F(i-1) + F(i-2)$). To do this, you need to come up with a recurrence relation that defines $N(h)$ and argue why it is correct. Then you need to prove, by induction, that $N(h) = F(h+3) - 1$.

Solution:
If the tree has height h then, by the definition of height, at least one of the two subtrees has height $h-1$. Furthermore, since all AVL trees are balanced, the other tree has height $h-2$ or $h-1$. Since we want the *smallest* possible number of nodes in the tree and since trees of greater height have more nodes, the second branch must have height $h-2$. The following recurrence relation results from summing the size of the two subtrees and adding one for the root node.

$$N(h) = N(h-1) + N(h-2) + 1$$

Next, we show that $N(h) = F(h+3) - 1$ by induction.

Proof. We verify two bases cases: those corresponding to $h = 0$ and $h = 1$:

- $N(0) = F(0+3) - 1 = F(3) - 1 = 2 - 1 = 1$
- $N(1) = F(1+3) - 1 = F(4) - 1 = 3 - 1 = 2$

Now, we show that if the statement is true for heights $h-2$ and $h-1$, it must also be true for height h. Using our recurrence relation, the inductive hypotheses, and the reccurence relation definiting the fibonnacci sequence, it follows that

$$N(h) = N(h-1) + N(h-2) + 1 = (F(h+2) - 1) + (F(h+1) - 1) + 1 = F(h+2) + F(h+1) - 1 = F(h+3) - 1$$

which is the desired formula. By the principle of induction, $N(h) = F(h+3) - 1$. □

[15] 3. Mysterious Dictionaries.

Dictionaries can be implemented in many different ways. Imagine you are given three dictionaries, each implemented with a "mystery" data structure. You know that the three possible underlying data structures are the following:

(a) Resizing hash table with linear probing

(b) Resizing hash table with quadratic probing

(c) Unsorted linked list

For each of the three dictionaries, explain what strategy you would use to determine the corresponding data structure. You may assume the following:

- `find(key)`, `insert(key)`, and `delete(key)` operations may all be used.
- You have a tool that tells you how long each operation takes to complete
- All the keys are unique; inserting/removing a key already present in the dictionary will have no effect on its contents.
- The unsorted linked list adds to the end of the list, and moves most recently "found" nodes to the front of the list.

Solution:
To distinguish the unstored linked list from the hash tables, you could begin by inserting all the positive integers, in order, up to some large number, and graphing the time taken as a function of the operation executed. Both the hash tables and the unsorted linked list should have constant time insertions, so you should have graphs that look like horizontal lines, although the graphs of the hash tables should have occaisonal jumps (corresponding to resizing operations).

Afterwards, you could search for all the positive integers you inserted, in the same order you inserted them in. For the linked list, everytime we search for a key, we must move it to the fornt, so the further the key from the front of the list, the longer it takes to find it. It can verify that searching for the integers in the order you inserted them will take an amount of time proportional to the integer, so the graph should look like some scaling of $f(x) = x$. On the other hand, the retrieval operations for the hash table should be approximately constant (assuming that our hash function is good), so that graph should be a horizontal line.

Now, to seperate the two hash table dictionaries, we could check the performance of the insertion as a function of how full the table is, since linear probing performs better for small load factors while quadratic is better for larger load factors. This means we have to graph the insertion time in the hash tables as a function of the number of insertions. Also, we would have to do this between resizing operations, which, as mentioned before, are identifiable as high peaks in the time vs. operation graph.

[6] 4. Priority Queues.

The queue ADT is often described as FIFO (first in, first out). The stack ADT is often described as LIFO (last in, first out). One of the following is a *good* description of the priority queue ADT. Indicate which one it is and why. One of the others almost always describes priority queues. Identify it and give two reasons why it's not the best description.

- Last In Last Out
- Only Best Out
- Never Worst Out

Solution: The good description is *Only Best Out*. A priority queue will always return its best element next. The nearly good description is *Never Worst Out*. For any priority queue with at least two elements, this is true, but

(a) it doesn't explain precisely which element the priority queue will return (it just indicates which element the priority queue will not return) and

(b) it's actually false for a priority queue with just one element, which will return its "worst" element, which also happens to be its best element.

[15] 5. Sorting.

(a) *Only Sort of Helpful?*

A common mistake made by those new to sorting is to unnecessarily sort all data. Although many operations can be performed faster on sorted data than unsorted data, this is not always the case. State the

- time complexity on unsorted data,
- time complexity on sorted data, and

- if the time complexities differ, provide a brief (1-2 sentences) explanation why

for the following scenarios:

i. Determining if a given value is larger than the largest value in a singly linked list of integers.

Solution: The complexity for unsorted data is $O(n)$. If we sort the list in descending order, we only need to compare the given value with the first value in the list, giving us a time complexity of $O(1)$.

ii. Determining if the current index contains the smallest value in an array of integers.

Solution: The complexity for unsorted data is $O(n)$. If we sort the array the time complexity is $O(1)$; whether we sort the data in ascending or descending order is not relevant since arrays allow constant access to the first or last index.

iii. Determining whether or not a value already exists within a given array of integers.

Solution: The complexity for unsorted data is $O(n)$. If we sort the array the time complexity is $O(\log n)$ since we can use binary search.

iv. Determining the depth of a binary tree.

Solution: The complexity for sorted and unsorted data is $O(\log n)$ if the tree is balanced and $O(n)$ otherwise.

v. Computing the average of a set of integers.

Solution: The complexity for sorted and unsorted data is $O(n)$.

vi. Finding the median of a set of integers.

Solution: The complexity for unsorted data is $O(n)$ (using an algorithm known as `quickselect`). If we sort the set into an array, then we have constant time access to the middle element (whose index it takes one operation to calculate) so the time complexity is $O(1)$.

(b) *Context.*

For each of the following scenarios, state which sorting algorithm you would use and why. You may choose from one of: selection sort, insertion sort, merge sort, or quick sort. You may only use an algorithm once. Answers without justification will receive no marks.

i. You're working for a telemarketing company that keeps hundreds of thousands of phone numbers. You need to write a function that can take a list of phone numbers and sort them by area code. You have no idea how large this list will be, but you do know that the salespeople who put them together don't do it in any particular order.

Solution: *Quick sort* or *Merge sort* - These two sorts are good choices because on average they run at $O(n \log n)$ (although quick sort runs in $O(n^2)$ time when the data is already sorted, you know that the salespeople do not put the data together in any particular order). Since getting a sorted list is unlikely, using quick sort is fine. The average runtime complexity of $O(n \log n)$ is crucial since the company keeps such a large number of phone numbers.

ii. For the final question at your technical interview at Google, you're asked to write a sorting algorithm for an array of integers. You're told that the size of the array will be no more than 30, and you have 5 minutes until your interview time is up.

Solution: *Selection sort* or *Insertion sort* - Although selection and insertion sort are both slow ($O(n^2)$ with an average case that is also quadratic), it doesn't really matter since the array will be no more than 30 items large. As well, these two are the simplest to write, so you might actually be able to finish within 5 minutes!

iii. NASA needs you to write a very quick sorting algorithm that will have to deal with a large amount of information, but run on a microchip with a very small amount of main memory.

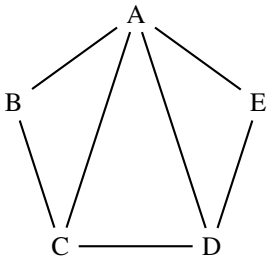
Solution: *Quick sort* - This is the best choice, as it generally runs quickly at $O(n \log n)$ but can take up less space than merge sort (merge sort takes $O(n)$ auxilary space but it's possible for quick sort to take up $O(\log n)$ auxilary space). We have to hope that the input will not already be sorted.

[15] 6. Graphs.

(a) Draw the undirected graph whose adjacency matrix is:

	A	B	C	D	E
A	0	1	1	1	1
B		0	1	0	0
C			0	1	0
D				0	1
E					0

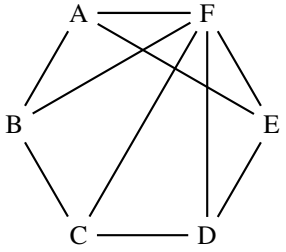
Solution:



(b) Draw the undirected graph whose adjacency list is:

A	B, E, F
B	A, C, F
C	B, D, F
D	C, E, F
E	A, D, F
F	A, B, C, D, E

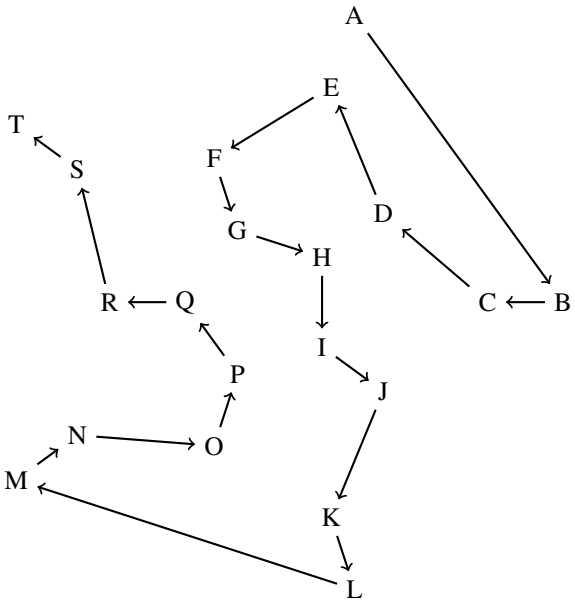
Solution:



(c) Perform Depth-first search, Breadth-first search, Dijkstra's algorithm, and Kruskal's algorithm on the following undirected, weighted graph. Use *A* as the source vertex for the first tree and, when there is a choice for which node to pick next, always chose the one that comes first in the alphabet.

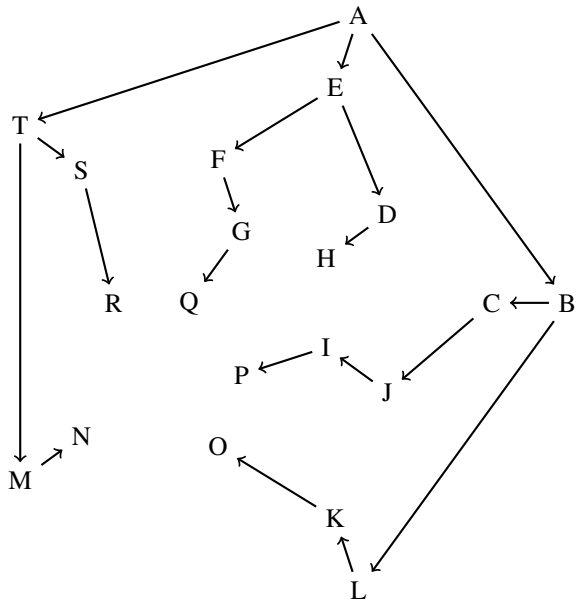
i. DFS-tree

Solution:



ii. BFS-tree

Solution:

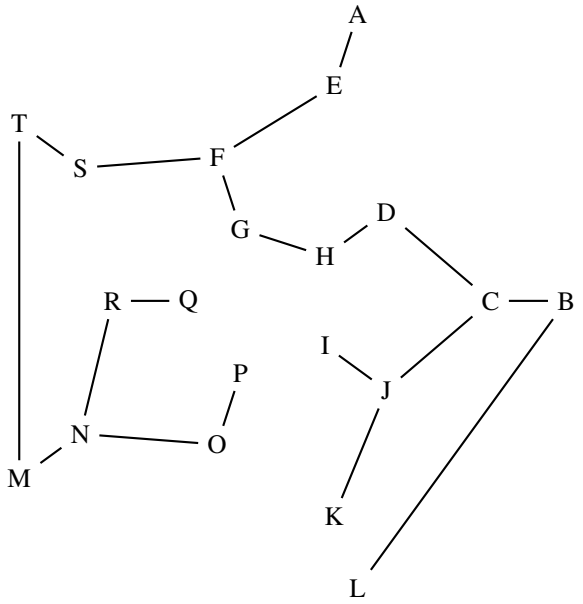


iii. shortest-path table

Solution:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
0	10	15	7	1	3	6	8	17	16	19	14	12	16	19	20	12	11	4	5

iv. minimum spanning tree.
Solution:



[10] 7. Combinatorics and Logic.

(a) A coin is tossed ten times. Each toss results in heads or tails. Calculate the following.

i. What is the total number of outcomes? **Solution:** $2^{10} = 1024$

ii. How many have exactly five heads? **Solution:** $C(10,5) = 252$

iii. How many have at least eight heads? **Solution:** $C(10,8)+C(10,9)+C(10,10) = 45+10+1 = 56$

iv. How many have at most one head? **Solution:** $C(10,1)+C(10,0) = 10+1 = 11$

(b) How many different strings can you form from the letters: "ALFALFA"? Explain.

Solution: Using the multinomial theorem, we permute the 7 letters in "ALFALFA", keeping in mind that we have 3 indistinguishable "A" characters, 2 indistinguishable "L" characters, and two indistinguishable "F" characters. Using the formula, this gives us

$$\frac{7!}{3!2!2!} = 210.$$

(c) For how many numbers in 1 to 999 is the sum of their digits 7? Explain.

Solution: First, we generate all sets of three digits that sum to 7. One helpful trick to enumerate these, while avoiding duplicates, is to list the elements of each set in ascending order. The sets you get are

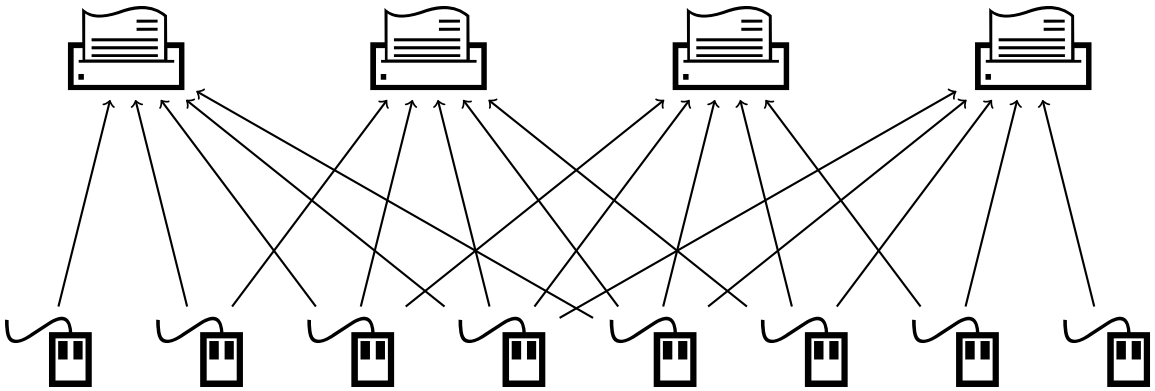
$$\{0,0,7\},\ \{0,1,6\},\ \{0,2,5\},\ \{0,3,4\},\ \{1,1,5\},\ \{1,2,4\},\ \{1,3,3\},\ \{2,2,3\}$$

Of these 8 sets 4 have a repeat digit and 4 have no repeat digits. Each of the latter can be arranged in $3! = 6$ ways while the former can only be arranged in $C(3,2) = 3$ ways. Then, the number of numbers whose digits sum to 7 is

$$4 \cdot 6 + 4 \cdot 3 = 36.$$

(d) Find the least number of cables to directly connect 8 computers to 4 printers so that any 4 computers can directly access 4 different printers. Prove that your answer is correct (i.e., show how to do it with that number and why it can't be done with fewer).

Solution: The "least number of cables" is 20, or some smaller number, since below we have an example that consumes 20 cables.



Now, we show that the "least number of cables" cannot be smaller than 20. Suppose, by contradiction, that it were 19 or less cables. Then, it would not be possible for all 4 printers to be connected to 5 computers; at least one printer would be connected to 4 or less computers. However, this would mean that there were 4 or more computers to which that printer is NOT connected. Then, if we picked those 4 computers, none would be able to access that printer. By contradiction, the "least number of cables" cannot be smaller than 20.

Since it is neither smaller nor larger than 20, the "least number of cables" is 20.

[18] 8. Proof.

Suppose that we have a *binary search tree* defined with a `Node` structure identical to the one from Lab 5 (still with integer keys – here we will use `int` instead of `KType`, however note that Lab 5 uses a `typedef` to make these equivalent). Furthermore, suppose we are given a `size(Node *root)` function that calculates the number of nodes in the tree rooted at the node `root`.

Then, consider the following function that uses the order property of binary search trees to extract the k-th smallest value stored in the tree rooted at the node `root`. We include the precondition that the tree at `root` has size greater than or equal to `k`, ensuring that the problem is well defined and there always is a k-th smallest value.

```
int find_kth_smallest_value(Node *root, int k) {

   Node *target = root;
   int p = k;
   int size_left_branch;

   while(true)
   {
      size_left_branch = size(target->left);

      if (size_left_branch == p-1)
         return target->value;
      else if (size_left_branch > p-1)
         target = target->left;
      else {
         target = target->right;
         p -= size_left_branch+1;
      }
   }
}
```

Take some time to make sure you understand what the function is doing before answering the following questions, in which you prove that the function works as you expect it to.

(a) Come up with and prove a loop invariant involving `k`, `p`, `root`, and `target`. (Hint: you may want to draw out a large BST and step through the function. As you do so, keep track of `p` and `target`).

i. State the loop invariant;

ii. Verify that it is true when the loop begins to execute; and

iii. Show that if it holds at the beginning of the loop, it must also hold at the end.

Solution:
Loop invariant: the k-th smallest value in the tree at `root` is the p-th smallest value in the tree at `target`.

Proof. When the loop begins to execute, p and `target` have just been assigned the values of k and `root`, so $k = p$ and the trees are the same. Trivially, the loop invariant must hold. Now assume the invariant holds at the beginning of the loop, we show that the invariant also holds at the end of the loop.

Since we are only interested in cases that reach the end of the loop, assume that `size_left_branch` and `p-1` are not equal. There are two possible remaining cases:

- If `size_left_branch > p-1` then the left branch contains the p-th smallest element (since there are more than $p-1$ elements elements in it and these are the smallest elements of the tree), and in particular, this element is also the p-th smallest element in the left branch, so when we assign `target` to be the left branch, the invariant still holds.
- If `size_left_branch < p-1` then the right branch contains the p-th smallest element (since there are less than $p-1$ elements elements in the left branch, there are not enough elements for the p-th smallest element to be there). Since we are taking the right branch to be the new `target`, the previous p-th smallest element is now the $(p-1-\text{size-of-left-branch})$-th smallest element in the right branch. This is because all of the nodes in the left subtree, along with the old `target` node, have smaller values.

At the beginning of the loop, the p-th smallest element in the tree `target` is the same as the k-th smallest element in the tree `root` so, at the end of the loop the new p-th smallest element in the new tree `target` is the same as the initial p-th smallest element in the initial tree `target` is the same as the k-th smallest element in the tree `root`. □

(b) Explain why the loop must terminate.

Solution: With every iteration of the loop, either `size_left_branch` and `p-1` are equal (in which the loop terminates), or they are not equal, in which case `target` is replaced by one of its branches. It follows that with every iteration of the loop, the size of `target` must decrease by at least one, so it is expected that `target` will reach size one in a finite number of steps or the function will return a solution before then.

(c) Use parts (a) and (b) to construct your argument as to why the function `find_kth_smallest_value(Node *root, int k)` returns the k-th smallest value in the tree at `root`.

Solution: From part (b), we know that eventually the loop will return. When this happens, the invariant (at the beginning of the last loop) will still hold, so (from (a)) the k-th smallest value in the tree at `root` is the p-th smallest value in the tree at `target`. However, since the size of the left branch is equal to $p-1$, there are $p-1$ elements smaller than the value at `target`, so the value of target, which must be the p-th smallest in the tree at `target`, is the k-th smallest value in the tree at `root`.