CSCE 221-200, Data Structures and Algorithms, Honors
Spring 2021
Homework 1 Solutions

(A) *(From Chapter 1) Design a recursive algorithm that takes as input a nonnegative integer n and outputs the number of 1's in the binary representation of n. Your algorithm can either be actual C++ code or pseudocode; it MUST be recursive. Hint: Use the fact that this quantity is equal to the number of 1's in the binary representation of $n/2$, plus 1 if n is odd.*

```
int num_ones(int n) {  // assume n >= 0
  if (n == 0) return 0;
  if ((n%2) == 0) // n is even
    return num_ones(n/2);
  else // n is odd
    return 1 + num_ones(n/2);
}
```

(B) *Exercise 2.7 (a): Give a big-Oh analysis of the running time of the six program fragments. Try to get your big-Oh value to be as small as possible while still being correct.*

(1) Since there is just one loop, which is done n times and whose body takes constant time (per iteration), the answer is $\Theta(n)$.

(2) Since there two nested loops, each of which is done n times, and the body takes constant time (per iteration), the answer is $\Theta(n^2)$.

(3) The outer loop, indexed by i, is done n times, the inner loop, indexed by j is done n^2 times, and the body takes constant time (per iteration). So the answer is $\Theta(n^3)$.

(4) The outer loop, indexed by i, is done n times. The inner loop, indexed by j, is done a different number of times, depending on which iteration of the outer loop we are considering. Specifically, for iteration i of the outer loop, the inner loop is done i times. The body takes constant time per iteration. The number of iterations of the body is

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = n(n+1)/2 = \Theta(n^2)$$

Thus the final answer is $\Theta(n^2)$.

(5) The outer loop, indexed by i, is done n times. The middle loop, indexed by j, is done i^2 times (so it depends on which iteration of the outer loop is being executed). The inner loop, indexed by k is done j times (so it depends on which iteration of the middle loop is being executed). The body takes constant time per iteration. The number of iterations of the body is

$$\sum_{i=1}^{n}\sum_{j=1}^{i^2}\sum_{k=1}^{j} 1 = \sum_{i=1}^{n}\sum_{j=1}^{i^2} j = \sum_{i=1}^{n} \frac{i^2(i^2+1)}{2} = \sum_{i=1}^{n} \Theta(i^4) = \Theta(n^5)$$

Refer to formulas on pages 4–5 of the textbook for the justification of these steps. Thus the final answer is $\Theta(n^5)$.

(6) The outer loop, indexed by i, is done n times. The middle loop, indexed by j, is done $i^2 - 1$ times. The "if" statement is done

$$\sum_{i=1}^{n}\sum_{j=1}^{i^2-1} 1 = \Theta(n^3)$$

times. The inner loop, indexed by k, is done j times but only if j is evenly divisible by i. The body takes constant time per iteration. We can get a rough upper bound on the number of iterations of the body by assuming the inner for loop is always executed:

$$\sum_{i=1}^{n}\sum_{j=1}^{i^2-1}\sum_{k=1}^{j}1=\sum_{i=1}^{n}\sum_{j=1}^{i^2-1}j=\sum_{i=1}^{n}\frac{(i^2-1)i^2}{2}=\sum_{i=1}^{n}\Theta(i^4)=\Theta(n^5).$$

Thus the running time is $O(n^5)$.

We can get a rough lower bound on the number of iterations of the body by assuming the inner for loop is only executed once, when $j=i^2-i$:

$$\sum_{i=1}^{n}(i^2-i)=\sum_{i=1}^{n-1}\Theta(i^2)=\Theta(n^3).$$

Thus the running time is $\Omega(n^3)$.

There is quite a gap here. By working harder at understanding exactly how many times the inner for loop is executed, it's possible to get a tight bound of $\Theta(n^4)$. Note that for a fixed value of i, the k-loop is executed when $j=1\cdot i, 2\cdot i, 3\cdot i, \ldots, (i-1)\cdot i$ and the number of iterations of the k-loop is j. We can write this number of iterations like this, letting $\ell=1,2,3,\ldots,i-1$:

$$\sum_{i=1}^{n-1}\sum_{\ell=1}^{i-1}\ell\cdot i=\sum_{i=1}^{n-1}i\sum_{\ell=1}^{i-1}\ell=\sum_{i=1}^{n-1}i\cdot\frac{(i-1)i}{2}=\sum_{i=1}^{n-1}\Theta(i^3)=\Theta(n^4).$$

(C) *Exercise 2.12: An algorithm takes 0.5 ms for input size 100. How large a problem can be solved in 1 min if the running time is the following?*

(a) linear: Assume $T(n)=c\cdot n$. We're given that $T(100)=c\cdot 100=.5$, so $c=.005$. Let x be the size of the problem we are looking for. Since 1 min is 60000 ms, we have $T(x)=.005\cdot x=60000$, so $x=12\times 10^6$.

(b) $O(n\log n)$: Assume $T(n)=c\cdot n\cdot\log_2 n$. We're given that $T(100)=c\cdot 100\cdot\log_2 100=.5$, so $c=.00075$ (approximately). Let x be the size of the problem we are looking for. Since 1 min is 60000 ms, we have $T(x)=.00075x\log_2 x=60000$, so x is about 3.66×10^6 (per Wolfram Alpha).

(c) quadratic: Assume $T(n)=c\cdot n^2$. We're given that $T(100)=c\cdot 100^2=.5$, so $c=.00005$. Let x be the size of the problem we are looking for. Since 1 min is 60000 ms, we have $T(x)=.00005x^2=60000$, so x is about 34,641 (per Wolfram Alpha).

(d) cubic: Assume $T(n)=c\cdot n^3$. We're given that $T(100)=c\cdot 100^3=.5$, so $c=5\times 10^{-7}$. Let x be the size of the problem we are looking for. Since 1 min is 60000 ms, we have $T(x)=(5\times 10^{-7})x^3=60000$, so x is about 4932 (per Wolfram Alpha).

(D) *Exercise 2.20:*

(a) Write a program to determine of a positive integer, N, is prime. Your "program" can either be in C++ or it can be a pseudocode description of an algorithm.

```
input:  N
output:  whether or not N is prime
for i = 2 to floor(sqrt(N)) do
  if N is divisible by i then return false
endfor
return true
```

(b) The worst-case running time occurs if the algorithm runs through the entire for loop before terminating, i.e., the input is prime. There are at most $\sqrt{N}$ iterations of the for loop and each one takes constant time. So the worst-case running time is $\Theta(\sqrt{N})$.

The reason this is correct is that N is composite (not prime) if and only if there are at least two (not necessarily distinct) integers p and q such that $1 < p, q < N$ and $p \cdot q = N$. In this case, at least one of p and q must be at most $\sqrt{N}$, otherwise their product would exceed N. So we only need to check divisibility up to $\sqrt{N}$.

(c) B, the number of bits in the binary representation of N, is approximately $\log_2 N$. More precisely, it is $\lceil \log_2(N+1) \rceil$.

(d) The worst-case running time of my solution, in terms of B: Since B is approximately $\log_2 N$, it follows that N is approximately 2^B. Plugging into the running time formula, we get $\Theta(\sqrt{2^B}) = \Theta(2^{B/2})$.

(e) Running time to determine if a 20-bit number is prime is about $2^{20/2}$, which is 1024, while running time to determine if a 40-bit number is prime is about $2^{40/2}$, which is 1,048,576. So the size of the input doubles (increases by a factor of 2) but the running time increases by a factor of over a thousand (1024).

(f) It is more reasonable to give the running time in terms of B, because of part (e). Also, no reasonable computing system accepts inputs in unary.

(E) *Exercise 2.23: Write the fast exponentiation routine without recursion. You can either use C++ or pseudocode.*

```
int power(int x, int n):   // assume x > 0 and n >= 0
  if (n = 0) then return 1 endif
  if (n = 1) then return x endif
  i = 0  // index into array A
  // fill in array A with decreasing exponents, inspired by recursive version
  while (n >= 1)
    A[i] := n
    if n is even then n := n/2 else n := n-1 endif
    i++
  endwhile
  // calculate answer using A, either doubling the accumulated answer if
  // exponent is even or multiplying it by one x if exponent is odd
  ans := 1
  for j := i-1 down to 0 do
    if A[j] is odd then ans := ans*x else ans := ans*ans endif
  endfor
  return ans
```

CSCE 221-200, Data Structures and Algorithms, Honors
Spring 2021
Homework 2 Solutions

A. *Exercise 3.20 (a): Lazy deletion for a linked list means just mark an element as deleted using an extra bit field and keep the number of deleted and nondeleted elements as part of the data structure. Once the number of deleted elements equals the number of nondeleted, then do the standard deletion on the marked nodes. What are the advantages and disadvantages of this approach?*

Advantages: Most of the time, deletion is faster, since we just have to change one bit instead of changing several pointers. This is not a huge speedup though.

Disadvantages: More space is needed (one bit per element, which might end up being one byte or even one word, and two more integers for the counters). However, if the "real" data per element is fairly large, this is negligible. Another disadvantage is that running time for deletion becomes highly variable, as once a while, the entire linked list needs to be traversed and the marked elements deleted, which will take time proportional to the length of the list instead of constant time.

B. *Exercise 3.24: Write routines to implement two stacks using one array. Your stack routines should not declare an overflow unless every slot in the array is used.*

The idea is to have one stack S_L grow "to the right" starting at the left end of the array and the other stack S_R grow "to the left" starting at the right end of the array. Keep two "pointers" (indices into the array) indicating the top of the two stacks. To push (resp. pop) S_L, increment (resp. decrement) its pointer. To push(resp. pop) S_R, decrement (resp. increment) its pointer.

C. *Exercise 3.25 (a): Propose a data structure that supports the stack push and pop operations and third operation findMin, which returns the smallest element in the data structure, all in $O(1)$ worst-case time.*

Store the data in an array `A`, indexed starting at 0. Variable `top` keeps track of the number of elements in the stack, initially 0. Each array element has two components, the value (`val`) and an array index (`pred`). If the value was, at any time, the smallest element in the stack, then `pred` holds the index containing the value that was the minimum until it was superseded by this value. Also keep track of the value of the minimum value (`minVal`) and its index (`minIndex`) in the array.

```
push(x):
  A[top] := (x,-1)
  if top = 0 then   // first element in stack
    minVal := x
    minIndex := top
  else
    if x < minVal then   // new minimum
      A[top].pred := minIndex   // keep track of former minimum
      minVal := x
      minIndex := top
    endif
  endif
  top++
--------------------
pop():
  if top > 0   // not empty
    (x,p) := A[top-1]   // latest element in the stack
    if top-1 = minIndex then  // this was the minimum element
      minVal := A[p].val      // get previous minimum
```

```
      minIndex := p           //   and its location
    endif
    top--
    return x
  else
    return "empty"
  endif
-------------------
findMin:
  if top > 0 then return minVal
  else return "empty"
```

D. *Present a (non-recursive) algorithm to determine whether a string is a palindrome. The algorithm must read in the string one character at a time and does not know in advance the length of the string. The algorithm is only allowed to use one or more stacks and a constant amount of extra space. (A palindrome is a string that is the same as its reverse; examples include "abba" and "abcba".)*

Idea: Each time a character is read, push it onto a stack and keep track of the number of characters. After reading all the characters, pop off half of them and put them in a second stack. Then pop characters off both stacks, comparing for equality. (If string is of odd length, then discard the middle character after transferring the first "half" to the second stack.)

```
int main(){
  stack<char> S1;
  stack<char> S2;
  // get the string and put the characters in stack S1
  cout << "Enter string to be checked for palindrome-ness, "
       << "one character at a time followed by return; ctrl-D to finish: ";
  char c;
  int sz = 0;
  while (cin >> c) {
    sz++;
    S1.push(c);
  }
  // transfer top half of the characters from S1 to S2
  for (int i = 0; i < sz/2; ++i) {
    c = S1.top(); S1.pop();
    S2.push(c);
  }
  // discard middle character if input string is odd
  if (sz % 2 == 1) S1.pop();
  // compare entries in S1 and S2 for equality
  while (!S1.empty()) {
    char c1; char c2;
    c1 = S1.top(); S1.pop();
    c2 = S2.top(); S2.pop();
    if (c1 != c2) {
      cout << "Not a palindrome.\n";
      return 0;
    }
  }
  cout << "Palindrome!\n";
  return 0;
}
```

E. *Let T be a tree with more than one node. Is is possible for the preorder traversal of T to give the same result as the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur. Is is possible for the preorder traversal of T to give the reverse of the postorder traversal of T? If so, give an example; otherwise, argue why this cannot occur.*

For the same order, no: preorder must visit the root first and postorder must visit the root last; since T has more than one node, these orders cannot be the same.

For the reverse order, yes: Consider the tree containing a root r and one child c (either left or right). The preorder traversal is r, c, while the postorder traversal is c, r.

F. *Exercise 4.5: Show that the maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$.*

Use induction on h.

Basis: $h = 0$. The tree consists of a single node and $1 = 2^{0+1} - 1$.

Induction. Suppose the maximum number of nodes in a binary tree of height h is $2^{h+1}-1$. Show that the maximum number of nodes in a binary tree of height $h+1$ is $2^{h+2} - 1$. To get the maximum number of nodes in a binary tree of height $h + 1$, have a root, both of whose children have height h and are roots of subtrees with the maximum number of nodes. In total, we have $1 + 2 \cdot (2^{h+1} - 1) = 1 + 2^{h+2} - 2 = 2^{h+1} - 1$.

G. Exercise 4.8: *Give the prefix, infix, and postfix expressions corresponding to the tree in Figure 4.75.*

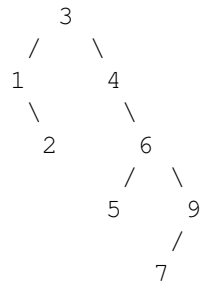
prefix: - * * a b + c d e

infix: a * b * c + d - e

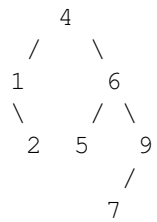
infix with parentheses: (((a*b)*(c+d)-e)

postfix: a b * c d + * e -

H. *Exercise 4.9 (a) and (b): Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empy binary search tree.*



Show the result of deleting the root.

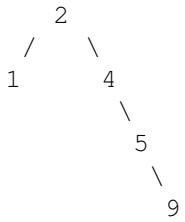


I. *Exercise 4.18 (b): What is the minimum number of nodes in an AVL tree of height 15? Justify your answer.*

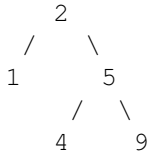
The answer is 2583. I determined this by writing a program that iteratively calculates $n(0), n(1), n(2), \ldots, n(15)$ using the recurrence given in class: $n(0) = 1$, $n(1) = 2$, $n(h) = 1 + n(h-1) + n(h-2)$ for $h > 1$.

J. *Exercise 4.19: Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.*

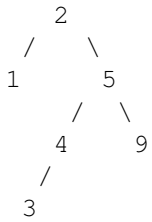
The basic BST insert algorithm works fine until inserting 9, which causes 4 to be unbalanced:



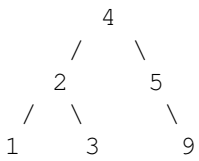
We can fix this with a single rotation:



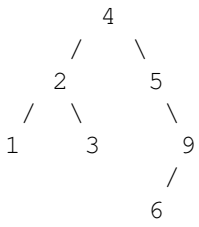
Then insert 3 which causes 2 to be unbalanced:



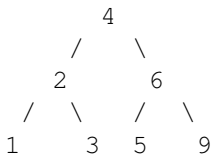
We can fix this with a double rotation:



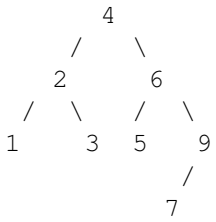
Then insert 6 which causes 5 to be unbalanced:



Fix with a double rotation:



Then insert 7 which doesn't cause any imbalance:



K. *Exercise 4.25: (a) How many bits are required per node to store the height of a node in an N-node AVL tree?*

The largest possible height of a node in an N-node AVL tree is less than $2 \cdot \log_2 N$ (using the better bound of $1.44 \cdot \log_2 N$ doesn't really save anything).

To represent the integer values from 0 to $2 \cdot \log_2 N$ in base 2 requires $\lceil \log_2(2 \cdot \log_2 N) \rceil$ bits, which is $1 + \lceil \log_2 \log_2 N \rceil$, a VERY slow-growing function.

(b) What is the smallest AVL tree that overflows an 8-bit height counter?

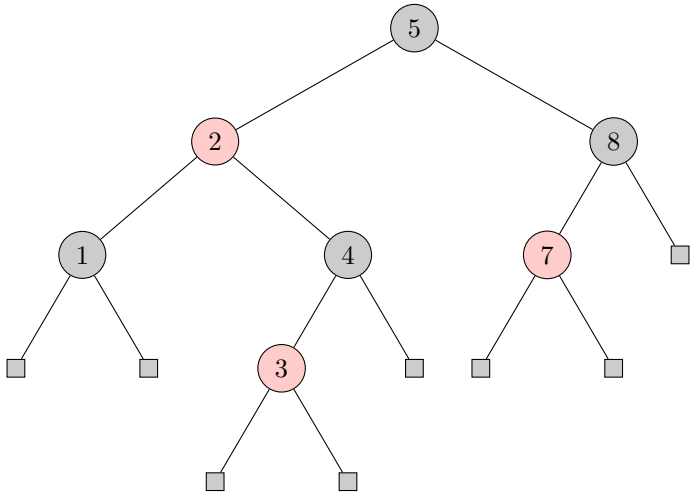
The largest value that can be represented with 8 bits is 256. So consider an AVL tree with height 256. The minimum number of nodes in an AVL tree of height 256 is HUGE. I wrote a little C++ program (using doubles) to calculate the recurrence from above, and discovered that the minimum number of nodes is about 6×10^{53}.

CSCE 221-200, Data Structures and Algorithms, Honors
Fall 2021
Homework 3 Solutions

(A) Consider the following recursive algorithm to color the nodes of an AVL tree.

1. color(t): // t is a tree node; in the top level call, t is the root
2. if t is empty then return // base case, nothing to do
3. color t black
4. color(left(t)) // recursively color the left subtree of t
5. color(right(t)) // recursively color the right subtree of t
6. if the height of t is even then
7. change the color of t's children with odd height to red

(i) Draw the result of applying the algorithm to the AVL tree on the left side of Figure 4.32 (p. 145) augmented with sentinel nodes. Explain why the result is a red-black tree.



The small squares indicate dummy sentinel leaves. Nodes 2, 3, and 7 are red while the rest are black. The root and the leaves are black; no red node has a red child; every leaf has the same black depth (3).

(ii) Prove that the output of the algorithm (when the input is an AVL tree) is always a red-black tree, using the following outline.
a) Check that the root is black.
b) Check that the leaves are black.
c) Check that no red node has a red child.
d) Check that the black depth of every leaf is the same. Use strong induction on h, the height of the tree, to show that the black depth of every leaf in the coloring produced by the algorithm is $\lceil h/2 \rceil + 1$.

(a) Line 3 ensures the root is black.

(b) The only place where a node's color is changed from black to red is in line 7; this only happens to nodes with odd height, but leaves have height 0 which is even.

(c) Show no red node has a red child. Suppose node x, a child of t, is colored red in line 7 because x's height is odd. But in the recursive call for x, x's children are never colored red because of the check in line 6 for even height.

(d) Show that the black depth of every leaf in the coloring produced by the algorithm is $\lceil h/2 \rceil + 1$, where h is the height of the tree.

Proof: By (strong) induction on h. For the basis, assume $h = 1$. This corresponds to an AVL tree with one data node and two sentinel leaves. The algorithm colors every node black in this case so the black depth is 2. Check that $\lceil 1/2 \rceil + 1 = 2$.

For the inductive case, assume the claim is true for every AVL tree with height less than h, where $h \geq 2$. Show it is true for every AVL tree T with height h.

By definition of AVL tree, at least one child of the root of T has height $h - 1$ and the other one has height $h-1$ or $h-2$. By the inductive hypothesis, after lines 4 and 5 (but before any possible recoloring in line 7), the black depths of the two subtrees are either both $\lceil (h-1)/2 \rceil + 1$ or one is $\lceil (h-1)/2 \rceil + 1$ and the other is $\lceil (h-2)/2 \rceil + 1$.

Case 1: h is even. Suppose both children of the root have height $h - 1$, which is odd. Then both of them are recolored red, but this decrease in the (common) black depth of all the leaves is compensated for by the addition of the black root.

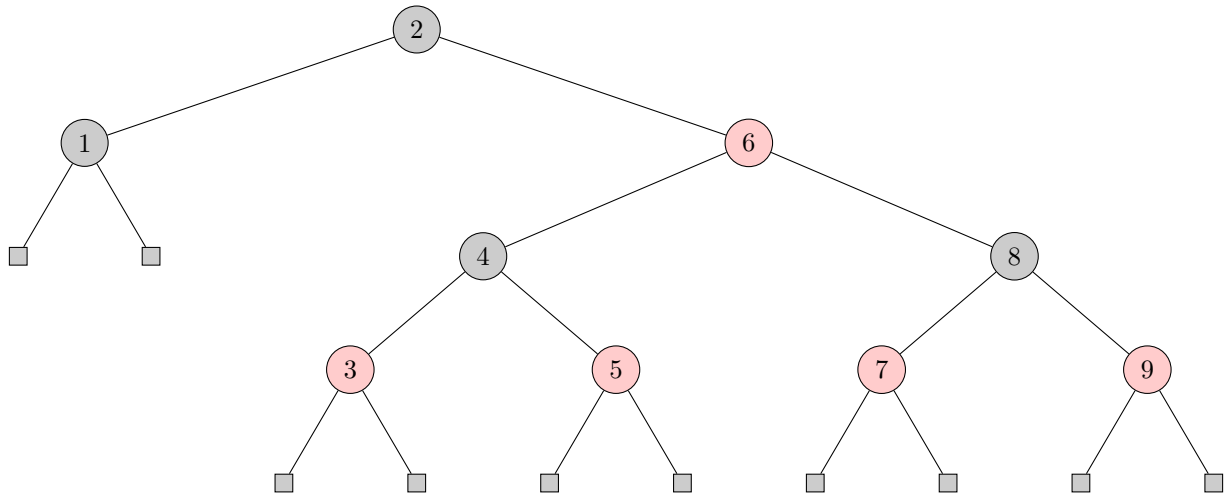
Suppose one child has height $h - 1$, which is odd, and the other child has height $h - 2$, which is even. Verify that in this case $\lceil (h-1)/2 \rceil + 1$ is 1 larger than $\lceil (h-2)/2 \rceil + 1$. The code ensures that the root of the subtree with height $h - 1$ is recolored red while the root of the other subtree stays black. Taking into account the addition of the black root, the black depth of the leaves in the subtree with larger height stays the same at $\lceil (h-1)/2 \rceil + 1$ (the original root becoming red is compensated for by the new root which is black), while the black depth of the leaves in the subtree with smaller height increases by one to become $\lceil (h-2)/2 \rceil + 2$, which equals $\lceil (h-1)/2 \rceil + 1$. Thus all the leaves have the same black depth. Note that since h is even, $\lceil (h-1)/2 \rceil + 1 = \lceil h/2 \rceil + 1$.

Case 2: h is odd. Then no nodes are recolored. Then $h - 1$ is even and $h - 2$ is odd. Verify that in this case $\lceil (h-1)/2 \rceil + 1 = \lceil (h-2)/2 \rceil + 1$. So for both possibilities for the heights of the subtrees of the roots, the black depths of the leaves are the same. The black root adds one to the black depth of all the leaves, causing them to continue to have the same black depth, which is $\lceil (h-1)/2 \rceil + 2$, which is equal to $\lceil h/2 \rceil + 1$ since h is odd.

Credit for the idea: https://cseweb.ucsd.edu/classes/su05/cse100/cse100wa2.txt

---

(B) Draw an example of a red-black tree that is not an AVL tree when the node colors are ignored.



The small squares indicate dummy sentinel leaves. Nodes 6, 3, 5, 7, and 9 are red while the rest are black. The root is unbalanced, as its left child has height 1 but its right child has height 3, and $|3 - 1| > 1$.

(C) Draw the 11-entry hash table that results from using the hash function $h(x) = (3x + 5) \bmod 11$ when you are given the following keys, in the given order: $12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5$. Assume collisions are handled by separate chaining.

The solution for Exercise C relies on the data in this table:

x	$h(x) = (3x + 5) \bmod 11$
12	8
44	5
13	0
88	5
23	8
94	1
11	5
39	1
20	10
16	9
5	9

Here is the answer for Exercise C:

0: -> 13
1: -> 39 -> 94
2: (empty)
3: (empty)
4: (empty)
5: -> 11 -> 88 -> 44
6: (empty)
7: (empty)
8: -> 23 -> 12
9: -> 5 -> 16
10: -> 10 -> 20

*** (D), (E) and (F) are postponed to HW 4 ***