CPSC 221: Algorithms and Data Structures
Assignment #2, due Monday, 2012 March 26 at 17:00

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 may work in pairs, but not in groups of three or more.) On your first page, also sign the statement "I have read and complied with the CPSC 221 2011W2 academic conduct policy as posted on the CPSC 221 course website." (See: http://www.ugrad.cs.ubc.ca/~cs221/current/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. Also, clearly mark "Wolfman" or "Hu" on the front page, depending which lecture section you attend, so we can return your assignment to you more easily. Finally, **staple** your submission's pages together! We are not responsible for lost pages from unstapled submissions.

Submit your assignment to Box 12, outside of ICCS 005. The deadline is 17:00 (5pm) on Monday, 2012-March-26. **Late submissions are not accepted.**

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

Questions

[30] 1. Going Loopy Over Induction

Consider the following code, which is supposed to check whether the string "INDUCTION" appears anywhere in the given string (idea from http://en.wikipedia.org/wiki/BoyerMoore_string_search_algorithm). **Correction: The code below doesn't actually work!**

```
CONTAINS_INDUCTION(string s):

  const string INDUCTION = "INDUCTION";
  int i = 0;

  while (i < s.length() - 8) {
    int j;
    for (j = 1; j <= 9; j++) {
      if (s[i+9-j] != INDUCTION[9-j]) break;
    }
    // NOTE: the value of j at this point is important!
    // The break ends the loop right away, leaving j
    // unchanged.  So, what does your invariant tell you
    // that j means?

    if (j == 10) return true;
    else if (j == 2 && s[i+7] == 'I') i += 7;
    else if (j == 1 && s[i+8] == 'I') i += 8;
    else i += 9;
```

```
}
return false;
```

Your task was to prove this code correct. Fortunately, there's a bug in the code, so this problem is now a bit easier! You should proceed in these steps:

(a) A good loop invariant for the inner (`for`) loop is: *The string "INDUCTION" and the string s[i..i+8] (the substring of s running from index i to index i+8, inclusive) match exactly in their final j-1 characters.* Prove using induction that the loop invariant holds.

Solution : For the base case, when the program first reaches the loop, `j` will be 1, so `j-1` is 0. So, the invariant holds vacuously. (BTW, for loop invariant proofs, the base case will **always** be about whether the loop invariant holds when the code first reaches the loop.)

For the inductive case, assume that the invariant holds at the top of the loop body (in this case, the `if` statement). If the `if`-statement test is true, the loop exits without changing j, so the loop invariant still holds. If the test is false, it means one more character matches between the two strings, so even after `j` gets incremented, the loop invariant still holds. QED

(b) Prove that the `for`-loop terminates. (Hint: This is really easy — like a single sentence.)

Solution : On each iteration, either the `break` happens, which terminates the loop, or else `j` increases by 1, which means the loop must terminate when `j` exceeds 9.

(c) The update code (the `if/else if/else` code) is where the bug is. Explain what the first two cases (the case where `(j == 10)` and the case where `(j == 2 && s[i+7] == 'I')`) are doing, and why they are correct. This explanation should give the intuition behind the cases. Do **not** just say things like "Add 7 to i"!! It will help greatly to describe what it means for the loop to terminate with a particular value of j.

Solution : In the `j==10` case, by our preceding proof about the `for`-loop, we know that the two strings match for $10 - 1 = 9$ characters. The string "INDUCTION" is 9 characters long, so we have a complete match and should return true. In the case (`j==2` etc.), because "INDUCTION" contains two Ns, it's possible for the `for`-loop to break after matching just the last N of INDUCTION with an N at `s[i+8]`. If the preceding letter is I, then this might be the start of the string "INDUCTION", so we continue searching the string `s` at `i+7`.

(d) Prove that the `while` loop terminates.

Solution : On each iteration, we either `return` from the function, which terminates the loop, or we increase `i`, and `i` can't exceed `s.length()-8`.

(e) Give an example of a string `s` that contains "INDUCTION", but for which the function will return false. (In other words, give an example showing the code is buggy.) Explain what happens.

Solution : A simple example is "xINDUCTION". The for-loop starts comparing backwards from `s[8]`, where the O is not equal to N, so it breaks immediately with `j==1`. This will cause `i` to be incremented by 9, the while loop terminates, and the function returns false.

[35] 2. AVL Heaps

In this problem, we will design a new data structure to implement the priority queue ADT. The new data structure is called an AVL Heap. It is a balanced binary tree, like an AVL tree (where the maximum imbalance at every node is ± 1), but instead of having the search tree property, all nodes obey the heap order property instead.

(a) Let $m(h)$ denote the smallest number of nodes that can be in an AVL Heap with height h, with $h \geq 0$ (so ignore the empty tree). The function $m(h)$ obeys the recurrence relation $m(h) = m(h-1) + m(h-2) + 1$. Briefly explain why. Then, prove that $m(h) \geq 1.6^h$. Don't forget your base cases! (Aside: This result applies to AVL trees, too.)

Solution : The recurrence relation holds because in order for the tree to have height h, one of its children must have height $h-1$. By the balance property, the other child must have height at least $h-2$. So, the smallest tree with height h will have two children, which are the smallest trees with height $h-1$ and $h-2$. The $+1$ is for the root.

For the base cases, the smallest tree with height 0 is a single node, so $m(0) = 1 \geq 1.6^0$. The smallest tree with height 1 has 2 nodes, so $m(1) = 2 \geq 1.6^1$.

For the inductive case, assume for all k with $h > k \geq 0$, that $m(k) \geq 1.6^k$.

$$\begin{aligned} m(h) &= m(h-1) + m(h-2) + 1 && \text{by definition} \\ &\geq 1.6^{h-1} + 1.6^{h-2} + 1 && \text{by inductive hypothesis} \\ &= 1.6^{h-2}(1.6+1) + 1 \\ &> (1.6^{h-2})(1.6)^2 + 1 && \text{because } 1.6+1 = 2.6 > 1.6^2 \\ &> 1.6^h \end{aligned}$$

QED

(b) In an AVL Heap, the functions `percolate_down` and `percolate_up` work just like in a normal heap, except that you follow pointers from node to node, instead of computing array indices. For example, suppose the nodes are implemented as:

```
struct node {
    int key;
    int height;
    node *parent;
    node *left;
    node *right;
}
```

Implement the function `void percolate_down(node *x)`. You should assume a min-heap. (It's easiest to do this recursively, but iteration is fine, too. It will also be easier to write your code to swap the values of the keys in nodes, instead of trying to adjust all the pointers.) Your result in part (a) means that the height of an AVL Heap containing n nodes is $\Theta(\log n)$. So, the runtime of `percolate_down` should be $O(\log n)$.

Solution : In reviewing the lecture notes we provided, I realized we gave a different API for `percolate_down` on heaps: we passed in the value of the key, and the function returned the **location** to put it. However, here, we've made the function return `void`, so it has to actually put the key in the right place.

Here's a recursive solution:

```
void percolate_down(node *x) {
  // x points to a node in the AVL Heap.
  // Performs a percolate_down starting at x.

  // If we're at a leaf, nothing more to do.
  if ((x->left==NULL) && (x->right==NULL) return;
```

```
  if (x->right==NULL) {
    // Only has left child.
    if (x->key > x->left->key) {
      swap(x->key, x->left->key);
      percolate_down(x->left);
    }
    return;
  }

  if (x->left==NULL) {
    // Only has right child.
    if (x->key > x->right->key) {
      swap(x->key, x->right->key);
      percolate_down(x->right);
    }
    return;
  }

  // Has two children, so we must pick smaller one.
  if (x->left->key < x->right->key) {
    // left side is smaller
    if (x->key > x->left->key) {
      swap(x->key, x->left->key);
      percolate_down(x->left);
    }
  } else {
    // right side is smaller
    if (x->key > x->right->key) {
      swap(x->key, x->right->key);
      percolate_down(x->right);
    }
  }
}
```

with helper function `swap` that swaps two integers:

```
void swap(int &x, int &y) {
  int temp = x;
  x = y;
  y = temp;
}
```

Notice that the recurisve solution is tail recursive, so it's very easy to make an iterative version.

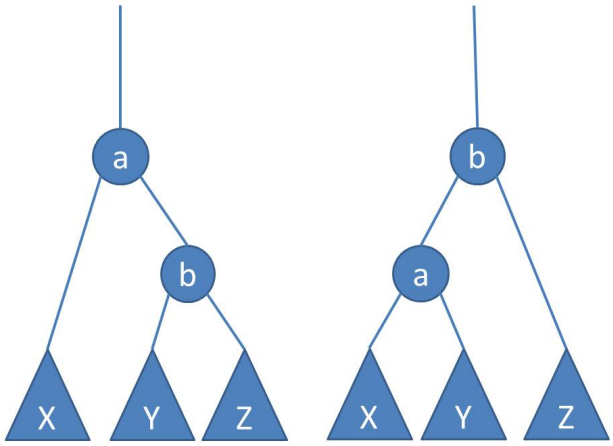
(c) In order to keep the AVL Heap balanced, we will use the same rotation operators as we do for AVL Trees. Give an example of a small AVL Heap where doing an AVL Tree `rotate_left` at the root results in a tree that is still balanced, but that violates the heap order property. (Hint: Your example can have as few as 2 nodes.)

Solution : Consider a two-node tree with a root and only a right child. The root has key 1; the child has key 2. This is a balanced AVL Heap. If you do a `rotate_left` on it, you get a tree with a root and only a left child. The 2 is at the root; the 1 is at the left child. This no longer has the heap order property. (Note that Steve's lecture notes sometimes have left and right flipped, so we'll give full credit if you think of `rotate_left` as rotating the other direction.)

(d) Prove that if we start with a (possibly unbalanced) AVL Heap, and then do a single `percolate_down` after calling `rotate_left`, we restore the heap order property. (The `percolate_down` is called on the new root of the subtree that was rotated.)

Solution : A complete solution should first note that if the `rotate_left` is called on a subtree, everything in that subtree is greater than the parent of the subtree (by the heap order property), so nothing outside of the subtree is affected by the rotation.

Now, consider the subtree that is rotated. Here, we see the subtree before (on the left) and after (on the right) a call to `rotate_left`. The triangles represent entire subtrees.



(Aside: If this had been an AVL Tree, we'd have the search tree property: $X < a < Y < b < Z$, where X represents all values in that subtree, etc. As you can see, the rotation preserves the search tree property, which is why AVL trees work so well.)

Since the tree on the left is an AVL Heap, we have the heap order property: $a < X$, $a < b$, $b < Y$, and $b < Z$. By transitivity, we also have $a < Y$. Now, look at the tree on the right. The subtree rooted at a already obeys the heap order property, since $a < X$ and $a < Y$. The other subtree Z is unchanged. So, b is the only item that violates the heap order property. Therefore, a single call to `percolate_down` will restore the heap order property to the entire subtree. (Aside: $b > a$, so they definitely will be swapped. After that, we know $b < Y$, so the `percolate_down` will only go into X.)

(e) In lecture, we've learned that an insertion into an AVL Tree never requires more than two rotations to restore balance. Consider the following algorithm for Insert in an AVL Heap:

i. From the root, follow child pointers arbitrarily until finding a node with a NULL child pointer (i.e., a leaf or a node with only one child). Attach a new node at the pointer and store the new key there. (Hint: "arbitrarily" means you can pick anywhere to insert the node to make your example easier.)

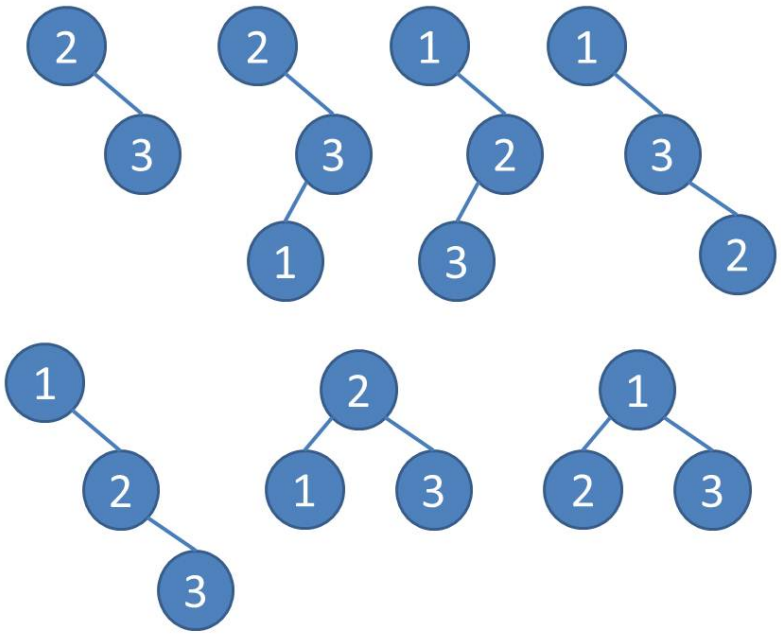
ii. Do a `percolate_up` from that new node.

iii. Do the rotations necessary to restore balance. After each rotation, do the `percolate_down` in order to restore the heap order property.

Give an example of an AVL Heap insertion that forces double rotation, **and** requires the new key to percolate all the way to the root. You should show:

i. your original (legal) AVL Heap before the insertion,
ii. the value you are inserting,
iii. the tree after you've added the new node but before you've done `percolate_up`,
iv. the tree after the first rotation but before the `percolate_down`,
v. the tree after that `percolate_down`,
vi. the tree after the second rotation but before the `percolate_down`, and
vii. the final tree with all AVL Heap properties restored.

Solution : It's actually possible to do this problem starting with a 2-node tree!



The top row, leftmost, is the original tree. Next is the tree after I've chosen to add a 1 (so it's smallest and will percolate to the top) in a "zig-zag" position to make the tree unbalanced and to force double rotation. Then I show the result of percolating the 1 up (which wasn't asked for in the question), and last on the top row is the first rotation (rotate right from the node labeled 2).

In the second row, you then see the propagate down of the 3. Next is the second rotation. And then the propagate down of the 2. This would be a full credit solution.

If you used a bigger tree, you'd see the rotations doing basically the same things, but the propagates go farther. In general, to get the double rotation, you must insert the node somewhere to create the "zig-zag" imbalance, and to make it percolate to the top, your newly inserted key must be smaller than anything already in your AVL Heap.

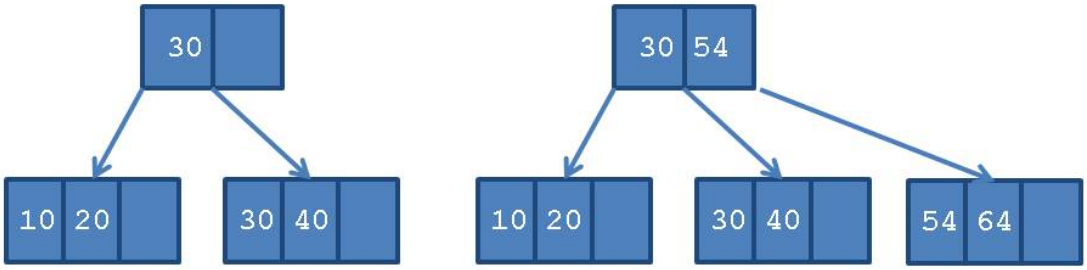
[35] 3. In this problem, you are given the following sequence of dictionary operations:

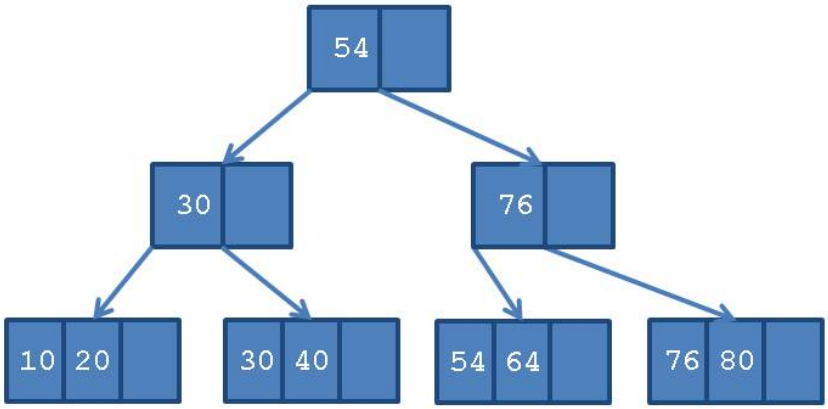
Insert(10), Insert(20), Insert(30), Insert(40), Insert(54), Insert(64), Insert(76), Insert(80), Insert(90), Delete(64), Delete(54), Insert(108).

Draw the dictionary data structures *after* each "interesting" operation is complete. An "interesting" operation is defined for each part.

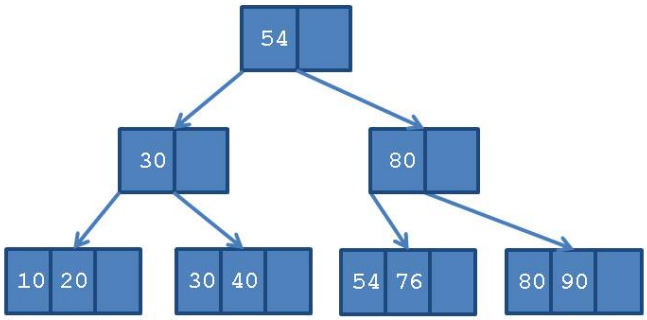
(a) A B+-Tree with $M = 3$ and $L = 3$. When nodes are split, put half of the children/items on the left and half on the right, putting the extra child/item on the left if there is one. Every insertion or deletion that causes a split, borrow from a neighbor, or merge is interesting, as is the last operation.

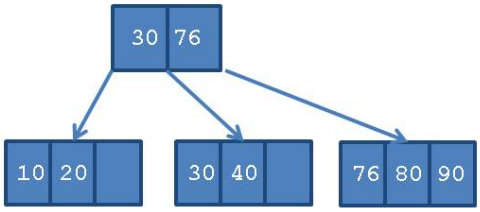
Solution : Lots of operations are interesting (unfortunately!). These graphs show the first few iteresting B+trees:



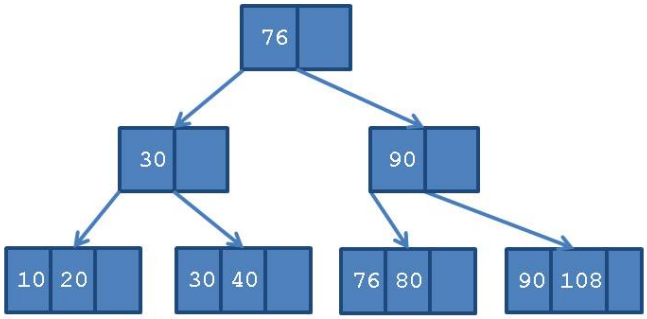


The first 3 inserts go into a single leaf node. With the 4th insert, the leaf splits, with two keys into each leaf, and growing a new root. That's the first graph above on the left. The 54 would be inserted into the same leaf as the 30 and 40. Then, the insert(64) forces the leaf to split again, yielding the graph above right. The 76 goes into the last leaf, and then the 80 forces a split. This split causes overflow in the root, forcing a split there, too, which generates a new root. The insert(90) puts the 90 in the last leaf, and then the delete(64) happens. This causes that leaf to underflow, so it borrows 76 from its neighbor. (Note that at this point, it could NOT merge.)





The graph above shows what happens after deleting 54. This causes underflow, but there's nothing to borrow from a neighbor, so this forces the leaves to merge. That causes underflow in the node with key 80 (which isn't needed anymore), and this causes underflow in the root, so the tree loses a level. After that, we insert 108, and we get splits propagating to the root again:

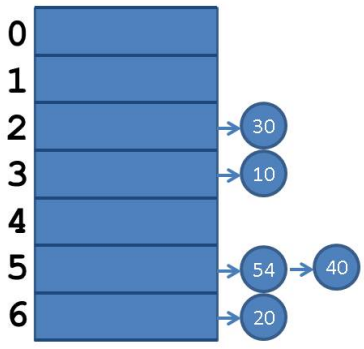


(b) A B+-Tree with $M = 20$ and $L = 20$. Every insertion or deletion that causes a split, borrow from a neighbor, or merge is interesting, as is the last operation.

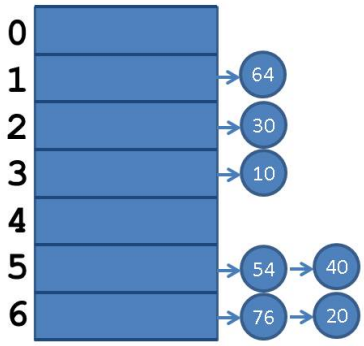
Solution : This is the way B+-trees are supposed to work! With *tons* of room for nodes inside a leaf, there are actually no splits at all after all the operations. So, only the last insertion is interesting:

(c) 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. Every insertion that collides is interesting, as is the last insertion. All deletions are interesting.

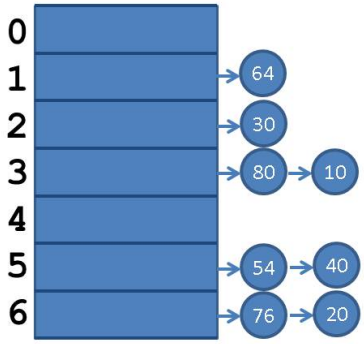
Solution : The 10, 20, 30, and 40 all hash with no collisions. The 54, though, collides with 40 in bucket 5.

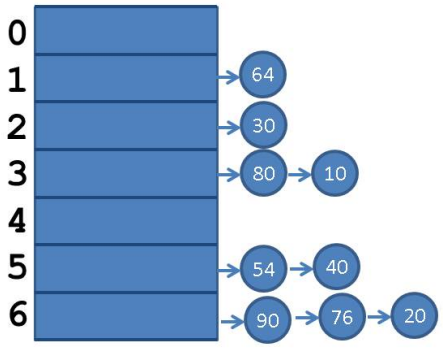


The 64 gets added with no collisions, but the 76 collides with the 20 in bucket 6.

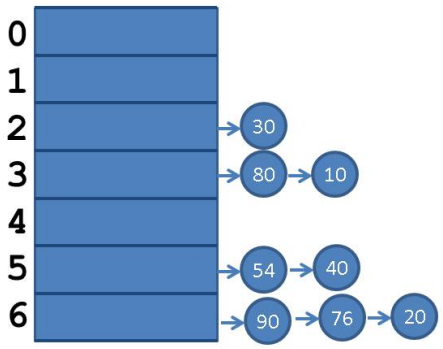


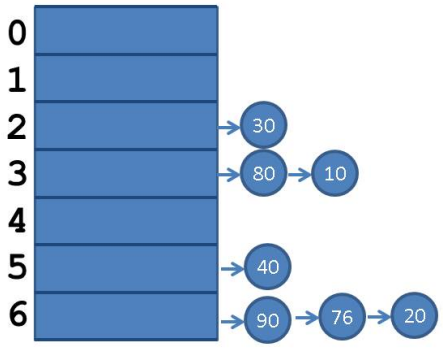
The 80 collides on bucket 3, and then the 90 collides on bucket 6.



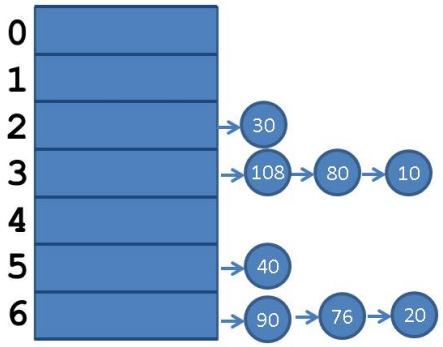


The deletions are straightforward, too. Deletion is easy with chaining:





And then the final insert of 108:



(d) 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 $h_2(n) = 5 - (n \mod 5)$. Every insertion that collides is interesting, as is the last insertion. All deletions are interesting.

Solution : The first few insertions (10, 20, 30, 40) go by with no collisions. The 54, unfortunately, collides with with 10. Computing $h_2(54) = 5 - (54 \mod 5) = 1$, so we probe at 0, which is free:

0	54
1	
2	
3	
4	
5	
6	
7	40
8	30
9	20
10	10

Next, we insert 64, which unfortunately also collides, this time at bucket 9. Computing $h_2(64) = 5 - (64 \mod 5) = 1$, so we probe at 10, then 0, then 1, which is free. Double hashing isn't working very well for this example so far!

0	54
1	64
2	
3	
4	
5	
6	
7	40
8	30
9	20
10	10

Next, we insert 76, which collides on bucket 10. Computing $h_2(76) = 5 - (76 \mod 5) = 4$, so this time, we get a different probe sequence. We probe bucket $(10 + 4) \mod 11 = 3$, which is free. Hooray for double hashing!

0	54
1	64
2	
3	76
4	
5	
6	
7	40
8	30
9	20
10	10

Unfortunately, the next insert(80) also hashes to bucket 3, so we collide again. Computing $h_2(80) = 5 - (80 \mod 5) = 5$, so we probe bucket 8, which is full, and then bucket 2, which is free:

0	54
1	64
2	80
3	76
4	
5	
6	
7	40
8	30
9	20
10	10

Our next insert(90) collides on bucket 2! What bad luck (or evil professors) we have! Computing $h_2(90) = 5 - (90 \mod 5) = 5$, so we probe bucket 7, which is full, and then bucket 1, which is full, and then bucket 6, which is free. Notice that the load factor is getting high, but because the hash table size is prime, double hashing is guaranteed to find a space if there is any left.

0	54
1	64
2	80
3	76
4	
5	
6	90
7	40
8	30
9	20
10	10

For the delete of 64, **you** could just look at the table and put a tombstone on 64, but the computer can't. It has to find 64 first. 64 hashes to bucket 9, which doesn't have 64. So, we compute $h_2(64) = 5 - (64 \mod 5) = 1$. Probing bucket $(9 + 1) \mod 11 = 10$, we don't find the 64. Checking bucket 0 doesn't find 64, but checking bucket 1 finally does. We mark it as deleted with a tombstone. (I will draw the tombstone as XX.)

0	54
1	XX
2	80
3	76
4	
5	
6	90
7	40
8	30
9	20
10	10

The delete 54 proceeds similarly:

0	XX
1	XX
2	80
3	76
4	
5	
6	90
7	40
8	30
9	20
10	10

BTW, you can see the importance of tombstones if we had instead deleted 54 first, and then 64. If we had just erased the 54, then when we tried to find 64, we'd give up when we hit the empty bucket 0. Instead, by putting a tombstone there, the find of 64 knows to keep probing.

Finally, we insert 108, which hashes to bucket 9. So, we compute $h_2(108) = 5 - (108 \mod 5) = 2$. Probing bucket $(9 + 2) \mod 11 = 0$, which contains a tombstone. When inserting into a hash table with open addressing, you can treat a tombstone as an empty slot, so we put the 108 there.

0	108
1	XX
2	80
3	76
4	
5	
6	90
7	40
8	30
9	20
10	10

(e) Show what happens if we try the above operations on a Hash Table of size 20. that uses open addressing with double hashing. The first hash is modding by the table size. The second hash is $h_2(n) = 5 - (n \mod 5)$. Every insertion that collides is interesting. Go as far as you can until an inseration fails, and explain the failure.

Solution : The first two insertions (10 and 20) work with no collisions. The 30, unfortunately, collides with with 10. Computing $h_2(30) = 5 - (30 \mod 5) = 5$, so we probe at 15, which is free:

0	20
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	10
10	
11	
12	
13	
14	
15	30
16	
17	
18	
19	

Similarly, the 40 collides at bucket 0, and then probes at 5.

0	20
1	
2	
3	
4	
5	40
6	
7	
8	
9	
10	10
10	
11	
12	
13	
14	
15	30
16	
17	
18	
19	

The next three inserts (54, 64, 76) go in no problem:

0	20
1	
2	
3	
4	64
5	40
6	
7	
8	
9	
10	10
10	
11	
12	
13	
14	54
15	30
16	76
17	
18	
19	

But things get ugly when we insert the 80: 80 hashes to bucket 0, which is full. Computing $h_2(80) = 5 - (80 \mod 5) = 5$, so we probe at 5, which is full, then 10, which is full, then 15, which is full, and then back to 0 again (which is still full), ad infinitem! Even though there is plenty of space in the table, we never find it. This is because the table size and the step size for probing (5) are not relatively prime. Note that we could get the same problem with a step size of 2 (and a few more values in the table). By having a prime hash table size, we guarantee this can never happen.