University of Illinois at Urbana-Champaign
Department of Computer Science

Second Examination *Solution*

CS 225 Data Structures and Software Principles
Fall 2007
7p-9p, Thursday, November 8

Name:
NetID:
Lab Section (Day/Time):

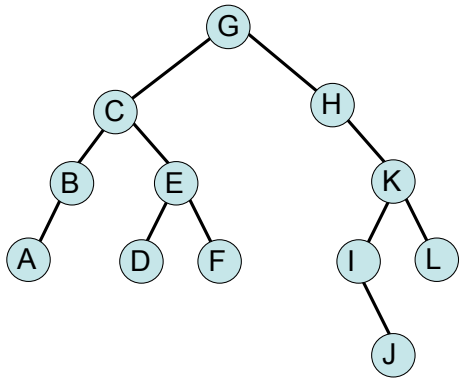
- This is a **closed book** and **closed notes** exam. No electronic aids are allowed, either.
- You should have 5 problems total on 20 pages. The last two sheets are scratch paper; you may detach them while taking the exam, but must turn them in with the exam when you leave.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.
- Please put your name at the top of each page.

Problem	Points	Score	Grader
1	20		
2	20		
3	20		
4	20		
5	20		
Total	100		

1. **[Choices, Choices! – 20 points].**

MC1 (2.5pts)

A post-order traversal of the following tree visits the nodes in which order?



(a) G C B A E D F H K I J L

(b) A B C D E F G H I J K L

(c) A B D F E C J I L K H G

(d) All of these are valid post-order traversals.

(e) None of these is a valid post-order traversal.

MC2 (2.5pts)

The `buildTree()` function in MP5 took two arguments, a BMP object `source` and an integer `resolution`, and built a quadtree to represent the image. Which of the following function declarations uses `const` appropriately in this context?

(a) `void Quadtree::buildTree(BMP const & source, int resolution);`

(b) `void Quadtree::buildTree(BMP & source, int resolution) const;`

(c) `void Quadtree::buildTree(BMP & const source, int resolution);`

(d) none of the above

MC3 (2.5pts)

Suppose that we have numbers between 1 and 1000 in a binary search tree and we want to search for the number 363. Which of the following sequences can not be the sequence of nodes visited in the search?

(a) 2, 252, 401, 398, 330, 344, 397, 363

(b) 924, 220, 911, 244, 898, 258, 362, 363

(c) 2, 399, 387, 219, 266, 382, 381, 278, 363

(d) 925, 202, 911, 240, 912, 245, 363

(e) 935, 278, 347, 621, 399, 392, 358, 363

MC4 (2.5pts)

Which of the following is the strongest valid statement made about AVL trees?

(a) They are minimal height trees

(b) They are height-balanced trees

(c) They are minimal height and height-balanced trees

(d) They are full, minimal height, and height-balanced trees

(e) None of the above are valid statements

MC5 (2.5pts)

Suppose an order m B-tree contains n items. In the worst-case, how many CPU operations would be required to search the tree for a specific key?

(a) $O(\log_2 n)$

(b) $O(\log_m n)$

(c) $O(m \log_2 n)$

(d) $O(m \log_2 m)$

(e) $O(m \log_m n)$

MC6 (2.5pts)

In an array-based implementation of a Heap, the right-child of the right-child of the node at index i, if it exists, can be found at what array location?

(a) $\boxed{4i + 3}$
(b) $2i + 1$
(c) $4i + 1$
(d) $2i + 2$
(e) $4i + 2$

MC7 (2.5pts)

Which of these is a Huffman Code for the character frequencies: 'a' = 5, 'b' = 3, 'c' = 4, 'd' = 10?

(a) $\boxed{\text{'a' is 10, 'b' is 110, 'c' is 111, and 'd' is 0.}}$
(b) 'a' is 10, 'b' is 01, 'c' is 00, and 'd' is 11.
(c) 'a' is 1, 'b' is 01, 'c' is 00, and 'd' is 0.
(d) 'a' is 111, 'b' is 0, 'c' is 10, and 'd' is 110.
(e) None of these.

MC8 (2.5pts)

For which of the following data structures does the `Find` function require no worse than $O(\log n)$ running time?

(a) Binary Search Tree (worst case analysis)
(b) Heap (worst case analysis)
(c) $\boxed{\text{Hashing (under simple uniform hashing assumption, and with an ideal hash function)}}$
(d) Two or more of the structures in (a) through (d) require no worse than $O(\log n)$ running time.
(e) $\boxed{\texttt{Find} \text{ does not run in time } O(\log n) \text{ for any of these structures.}}$

The correct answer for this problem is (c), since, under the described conditions, hashing provides $O(1)$-time find. BUT, we did not emphasize the fact that if a function $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$, so we also counted (e) as a correct response.

2. **[QuadTrees – 20 points].**

For this question, you will be writing a private member function called `blockyBlur` whose purpose is to add very basic "blur" functionality to the MP5 `Quadtree` class. It should be implemented using the following rules:

(a) Block

i. If the quadtree is null or just a single node, don't do anything.

ii. If a node is a parent of 4 leaves in the original tree, remove the leaves from the tree (do not do this recursively). This creates a blocky effect, where every group of four pixels is now represented by one pixel.

(b) Blur

i. If a node has no parent or if it has had no children removed, don't change the value of its `element` (Hint: your code will be simplified if you observe that the average of a number and itself is just that number).

ii. If a node has had its children removed and if it has a parent, make its `element` the average of its own `element` and its parent's. Recall that in our quadtrees, the `element` is an RGBApixel.

You may assume that the quadtree is complete and that it has been built from an image that has size $2^k \times 2^k$. You need not assume anything in particular about the value of the `element` in any node, except that it contains some valid pixel (presumably assigned when the tree was built). Finally, you MAY write private `Quadtree` helper functions to solve this problem, but you are not required to do so.

We have provided the public member function `blurTree` which calls `blockyBlur` on the `root` node of the quadtree, and another function `avg` which computes the average of two RGBApixel objects (you are welcome to *use* this function).

```
// public interface for blurring a Quadtree
void Quadtree::blurTree(void)
{
   // note that in this function call, we're using root->element as the value
   // of the parentPixel of the root because we want the element of a single
   // node quadtree to remain unchanged.  That is, if the root HAD a parent,
   // it's element would have the same value as root->element.

   blockyBlur(root, root->element);
}

// returns an RGBApixel representing the average of two RGBApixels
RGBApixel avg(RGBApixel p1, RGBApixel p2);
```

Write your code for `blockyBlur` and any helper functions on the next page(s).

```
// performs a basic blur on the tree, starting at the parameter node
void Quadtree::blockyBlur(QuadtreeNode* current, RGBApixel parentPixel)
{
```

Ideal Solution:

```
// performs a basic blur on the tree, starting at the parameter node

void Quadtree::blockyBlur(QuadtreeNode* current, RGBApixel parentPixel) {
  setAvg(current);
  combined(current, parentPixel);
}

void Quadtree::setAvg(QuadtreeNode* current) {
   if(current->neChild == NULL) {
      return;
   }
   setAvg(current->neChild);    // recurse on all four children
   setAvg(current->nwChild);
   setAvg(current->swChild);
   setAvg(current->seChild);
   // set the current nodes element to avg of its children
   current->element.Red =   (current->nwChild->element.Red
     + current->neChild->element.Red + current->seChild->element.Red
     + current->swChild->element.Red) / 4;
   current->element.Green = (current->nwChild->element.Green
     + current->neChild->element.Green + current->seChild->element.Green
     + current->swChild->element.Green) / 4;
   current->element.Blue =  (current->nwChild->element.Blue
     + current->neChild->element.Blue + current->seChild->element.Blue
     + current->swChild->element.Blue) / 4;
}

void Quadtree::combined(QuadtreeNode* current, RGBApixel parentPixel) {
  if(current->neChild == NULL) {   // handle single node tree
     return;
  }
  // you must have four children at this point
  if(current->neChild->neChild == NULL) {   // current is parent of four leaves
    removeLeaf(current->neChild);
    removeLeaf(current->nwChild);
    removeLeaf(current->seChild);
    removeLeaf(current->swChild);
    // since children have been removed at this point and if current
    // doesnt have a parent, its pixel wont be changed
```

```
    current->element = avg(current->element, parentPixel);
  }
  else {
    combined(current->neChild, current->element);   // recurse on all four children
    combined(current->nwChild, current->element);
    combined(current->swChild, current->element);
    combined(current->seChild, current->element);
  }
}

void Quadtree::removeLeaf(QuadtreeNode*& current) {
  delete current;
  current = NULL;
}
```

Grading Rubric:

Setting element in non-leaf nodes to average of children (1 point)
Handling tree with single node (1 point)
Recursion (4 points)

- Correct base case (2 points)
- Correct recursive calls (2 points)

Block Functionality (4 points)

- Correctly checking if node is a parent of four leaves (2 points)
- Proper memory management (2 points)
  - Correctly deallocating dynamic nodes (1 point)
  - Setting deallocated pointers to NULL (1 point)

Blur Functionality (4 points)

- Correctly checking if node had its children removed and if it has a parent or using the hint to simplify this check(2 point)
- Correctly setting element to the correctly computed average (2 point)

Comments (6points)

- 1 point for commenting about when to recurse
- 1 point for commenting about the base cases
- 2 points for commenting about how and when to block
- 2 points for commenting about how and when to blur

3. **[Analysis – 20 points].**

(a) Suppose we have a min-Heap A and a max-Heap Z, each of which store integers. Assume also that all the keys in A are *less than* all the keys in Z, and A and Z have the same number of keys. (Note: a max-Heap is just a Heap where all paths from root to leaf are *decreasing*.) Finally, assume that we have a Binary Search Tree class called `BST` with all the usual member functions.

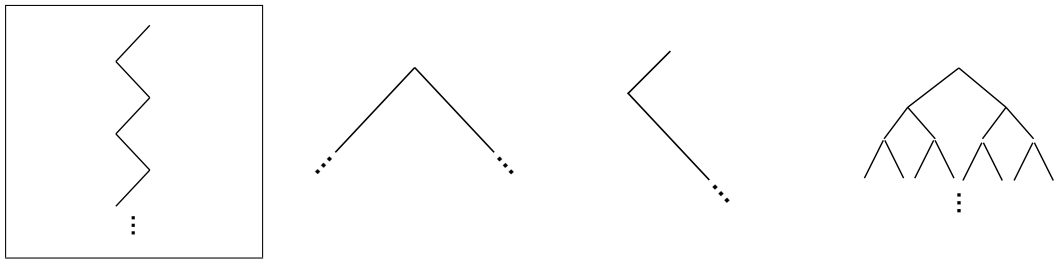
Consider this function `play`:

```
BST  play(minHeap A, maxHeap Z){

   BST B;
   while (!A.isEmpty && !Z.isEmpty) {
      B.insert(Z.removeMax());
      B.insert(A.removeMin());
   }
   return B;

}
```

i. Circle the illustration that *best* describes the BST returned by `play`.



we have:
A: min-Heap
Z: max-Heap
B: BST
two rules of the play:
1) all the keys in A are less than all the keys in Z
2) A and Z have the same number of keys (let it be n)

We start building BST B by removing the max (let call b1) from Z, and adding it to B as it's root. (note that each removing from maxHeap or minHeap needs heapifying to preserve the max or min heap structure). The next node that we will insert into the BST is the minumum of the minHeap, let call it b2. As it mentioned in the rules all the keys of minHeap are less than all the keys in maxHeap, so b2 is less than b1, and it will be the left child b1. Suppose that there are still some nodes remain in min and maxHeap. In the next iteration of while again we remove the

max from the maxHeap, b3. This would be less than b1, because it is second max of maxHeap, (remember when we removeMax nodes from maxHeap one by one, they are sorted in decreseaing order). so we must add it somewhere in left child of b1. But regarding the rule of play b3¿b2, so we must add it as the right child of b2.

ii. Analyze the total running time of `play` if each Heap contains n data items.

Solution:
For simplifying the runtime analysis we can write the code as following:

```
BST play(minHeap A, maxHeap Z)  {
   BST B;
   while(!A.isEmpty() && !Z.isEmpty()) {
      N=Z.removeMax();
      B.insert(N);
      N=Z.removeMin();
      B.insert(N);
   }
   return B;
}
```

It is obvious that the while loop goes through all n nodes of minHeap and maxHeap. As it is shown all the operations inside the loop are done sequentially, so we need to find the maximum order of the operations inside the loop and just multiply the maximum order by n for finding the total running time. removeMax and removeMin from heap each takes $O(\log(n))$. Insert into BST takes $O(n)$ in the worst case. Therefore the maximum order of operations inside the loop is $O(n)$. By multiplying $n * O(n)$ the order is $O(n^2)$.

(b) Suppose we have a min-Heap A and a max-Heap Z, each of which store integers. Assume also that all the keys in A are *greater than* all the keys in Z, and A and Z have the same number of keys. (Note: a max-Heap is just a Heap where all paths from root to leaf are *decreasing*.) Finally, assume that we have an AVL Tree class called `AVLTree` with all the usual member functions.

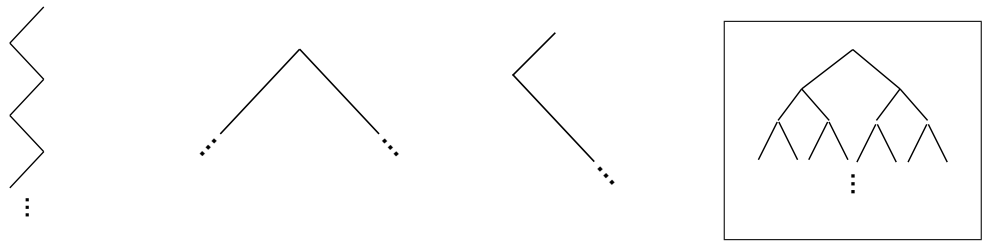
Consider this function `work`:

```
AVLTree  work(minHeap A, maxHeap Z){

   AVLTree B;
   while (!A.isEmpty && !Z.isEmpty) {
      B.insert(Z.removeMax());
      B.insert(A.removeMin());
   }
   return B;

}
```

i. Circle the illustration that *best* describes the AVL Tree returned by `work`.



we have:
A: min-Heap
Z: max-Heap
B: AVL tree
two rules of the play:
1) all the keys in A are greater than all the keys in Z
2) A and Z have the same number of keys (let it be n)

We start building AVLTree B by removing the max (let call b1) from Z, and adding it to B as it's root. (note that each removing from maxHeap or minHeap needs heapifying to preserve the max or min heap structure). The next node that we will insert into the BST is the minumum of the minHeap, let call it b2. As it mentioned in the rules all the keys of minHeap are greater than all the keys in maxHeap. We just need to remember that after each insertion into AVLTree we need to re-balance it, so the final tree is balanced.

ii. Analyze the total running time of `work` if each Heap contains n data items.

Solution:
For simplifying the runtime analysis we can write the code as following:

```
AVLTree work(minHeap A, maxHeap Z)  {
   AVLTree B;
   while(!A.isEmpty() && !Z.isEmpty()) {
      N=Z.removeMax();
      B.insert(N);
      N=Z.removeMin();
      B.insert(N);
   }
   return B;
}
```

like previous section, It is obvious that the while loop goes through all n nodes of minHeap and maxHeap. As it is shown all the operations inside the loop are done sequentially, so we need to find the maximum order of the operations inside the loop and just multiply the maximum order by n for finding the total running time. removeMax and removeMin from heap each takes $O(\log(n))$. Insert into AVLTree takes $O(\log(n))$. Therefore the maximum order of operations inside the loop is $O(\log(n))$. By multiplying $n * O(\log(n))$ the order is $O(n \log(n))$.

My Rubric for grading this question:
There are four parts that each has 5 points. So highlighting the correct structure of the tree in the first parts of a and b gets 5 points for each, all or nothing.
In the analysis part:

- 1.5 points if you detect the loop.
- 1.5 points if you detect the correct order of operations inside the loop ($O(n)$ for part a-ii & $O(\log(n))$ for part b-ii).
- 1 point if you multiply n by whatever you got from the previous step
- 1 point if the final answer is correct

Also, minus 1 point for wrong statements.

4. **[All about AVL trees – 20 points].**

(a) Recall that AVL Trees perform rotations to maintain height balance. Draw a picture of a 7 node binary search tree that would be an AVL tree after a right-left double rotate, and show that it becomes an AVL tree after those rotations. In your illustrations, be sure to include a valid key in each node, and an indication of which node was originally out of balance.

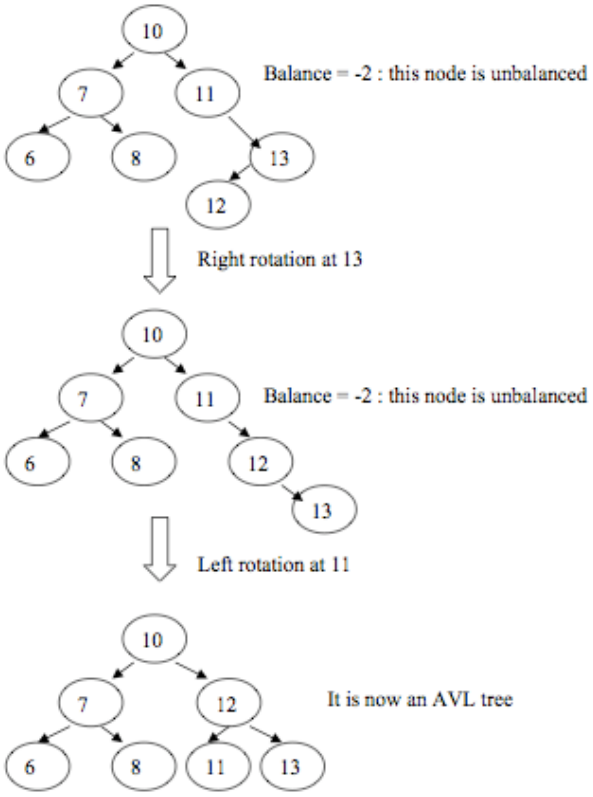


Figure 1: right-left rotation

Sol. 4(a). Figure 1 gives one possible solution. Many other solutions are also possible.
Grading scheme: total 5 points
Correct example of tree = 2 points
Valid keys in tree = .5 point
Writing balance to indicate which node was out of balance = .5 point
First correct rotation = 1 point
Second correct rotation = 1 point

(b) In this part of the problem you will implement the `rotateLeft` member function for the `AVLTree` class. Here is the partial `AVLTree` class definition:

```
class AVLTree {
```

```
public:
   ...
private:
   class AVLTreeNode {
   public:
      ...
      int element;
      int height;
      AVLTreeNode* left;
      AVLTreeNode* right;
   };
   int max(int a, int b) const;
   int height(AVLTreeNode const * x) const;
   ...
   // function declarations for rotations go here.
   // We're not telling you what they look like.
};
```

The function `height` takes an `AVLTreeNode` pointer argument and returns the integer height of the subtree rooted at the parameter node. The function `max` takes two integer arguments and returns an integer that is the larger of the two parameters. Each of the rotation functions takes a reference to an `AVLTreeNode` pointer, performs its rotation, and returns nothing.

Using this information, implement the `rotateLeft` member function here:

Sol. 4(b). Figure 2 gives the diagram for left rotation. One solution is given below (other solutions are also possible):

```
void AVLTree::rotateLeft(AVLTreeNode * & node) {

// no need to check for conditions, as they are supposed
// to be checked before calling this function
AVLTreeNode* y = node;
node = node->right;
y->right = node->left;
node->left = y;
y->height = height(y);

}
```

Grading scheme: total 5 points
Considering possible presence of T2 (see the figure) = 1 point
Correct rotation process = 3 points
Updating height = 1 point

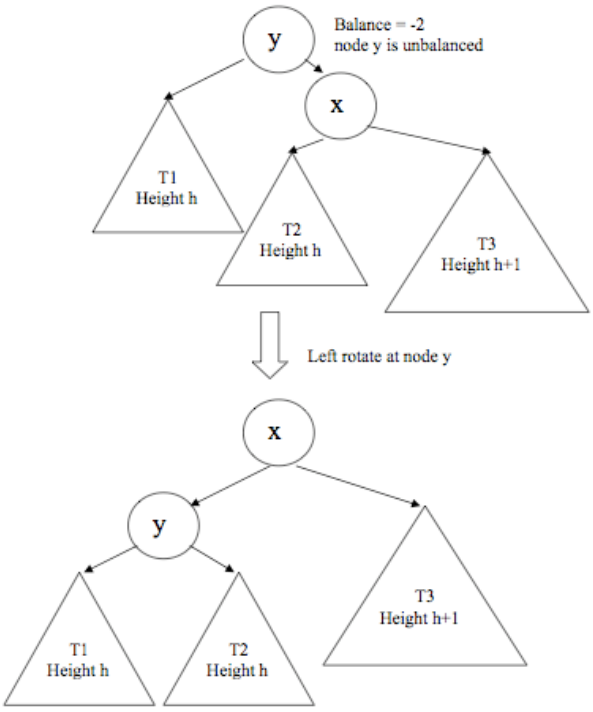


Figure 2: left rotation

(c) Prove by induction that the number of null pointers in an n node AVL tree is $n + 1$.

Let T be an arbitrary AVL tree with n nodes. Assume (Inductive Hypothesis) that any AVL tree with $j < n$ nodes has $j + 1$ null pointers.

Base Case ($n = 0$): Suppose T is a tree with zero nodes (an empty tree). Then, since we represent T using one null pointer, the number of null pointers is one more than the number of nodes.

Inductive Case ($n > 0$): By definition of AVL trees, T's left and right children, T_L and T_R, are AVL trees. Denote by a and b the number of nodes in T_L and T_R respectively. Since $a < n$ and $b < n$ we can apply the inductive hypothesis to T_L and T_R, and thus T_L has $a+1$ null pointers and T_R has $b+1$ null pointers. The total number of null pointers in T is $(a+1)+(b+1)$, and this is $n+1$ since the total number of nodes is $n = a+b+1$.

Grading Rubric:

Almost no one approached this the correct way, so I gave full credit for some incorrect

responses (what was I thinking!!??). The common error was to remove a single node from a tree of size $n+1$, apply the inductive hypothesis to the resulting tree, and adjust the pointer count by adding back in the removed node. The problem is that you cannot apply the inductive hypothesis to the tree in this case, because you can't be sure that it's an AVL tree.

- 1 point for base case
- 2 points for stating inductive hypothesis
- 2 points for the inductive argument

(d) Suppose your study partner suggests that the two of you should implement an AVL tree using an array, much like we use an array to implement a Heap. First, explain why your partner thinks this might be a good idea. Second, give your argument that it really isn't such a good idea after all. In your dialogue, you should address basic functionality (find, insert, remove) and rotations. You may also wish to discuss space usage and running times. The graders will be delighted if you illustrate your arguments using small example trees.

Partner: Though an n node AVL tree isn't complete, it has height which is $O(\log n)$, which isn't so bad. If we store the elements in an array like we do for heaps (or any other complete tree) we probably won't be wasting too much space. (An analysis that I didn't expect shows that the space required is no more than n^2 for an n node AVL tree.) The simplicity of the parent and child relationships makes this really attractive!

You: Nah. While find would work in $O(\log n)$ time, insert and remove would be disastrous because of the rearrangement that occurs with rotations. It's possible that every single element of the array would need to be moved to a different spot upon a rotation, even if the relative positions of nodes in subtrees didn't change. This is not a problem in a pointer based AVL tree.

Grading rubric:

Maximum of 2 points if there was no explanation of how an AVL tree would be stored in a manner similar to a Heap. To detect what you were imagining, I looked for pictures, or for mention of the indices of children (2i and 2i +1), or for mention that there may be gaps in the array if the tree is not complete.

An argument similar to the correct one but with no mention of rotations received 3 points.

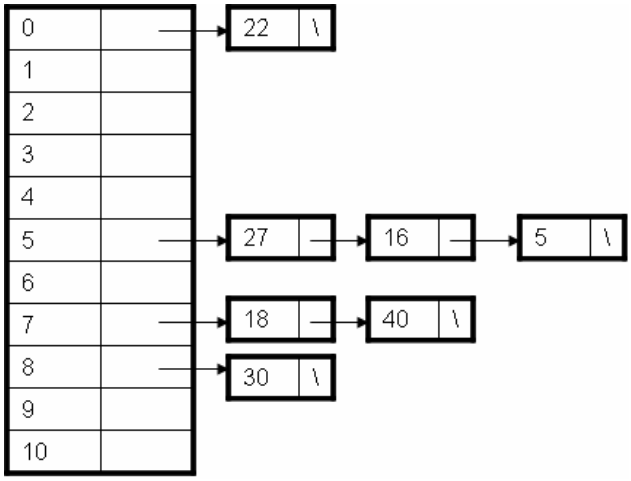
Beyond that, i basically just gave a point for each *correct* observation, and subtracted a point for ea ch *incorrect* observation.

5. **[Make Hash of It – 20 points].**

Suppose you are given a table T of size 11 and a set S = {5, 40, 18, 22, 16, 30, 27} to hash into the table, using the hash function *h(k) = k%11*.

(a) Show T after the values from S are entered into it, using separate chaining.

Solution:



(b) Show T after the values from S are entered into it, using linear probing with the function *h(k,i) = (h(k)+2i)%11*.

Solution:

0	22
1	
2	16
3	
4	27
5	5
6	
7	40
8	30
9	18
10	

(c) For this particular T and S, which is the better choice, and why? Your answer should discuss the number of probes necessary to `Find` a particular key.

Solution:

Separate chaining is the better choice for this set S. The longest list in the separate chaining case contains 3 elements to search through, while linear probing requires as many as 6 probes to insert or find elements.

(d) Give an example of a set S with 7 elements for which the other method is the better choice.

Solution:

A good set for linear probing is one that has no collisions, e.g., {1, 2, 3, 4, 5, 6, 7}. In this case, the extra space required for the lists of separate chaining is a waste of memory, as a `find` is equally fast for either method.

Rubric:

(a) 7 pts
-1 pt if array is not used to hold header nodes
-1 pt if elements are not inserted at the head of the list
-1 pt each for arithmetic errors in placing elements
-3 pts if no lists were used
-2 pts if collisions were resolved using $h(k, i) = (h(k) + i)\%11$

(b) 7 pts
-1 pt each for arithmetic errors in placing elements
-4 pts if consistently misusing $h(k, i) = (h(k) + 2i)\%11$ with respect to i
-3 pts if $h(k, i) = (h(k) + i)\%11$ was used correctly

(c) 4 pts
2 pts for a correct answer
2 pts for a correct reason
-3 pts if answer was wrong because (a) or (b) was done incorrectly

(d) 2 pts
-1 pt for right set, wrong reason
-1 pt if correct relative to (c) (and (c) was wrong) and a reason was given