**Chapter 10 – 5 Points**

**4) 0.5 Points**

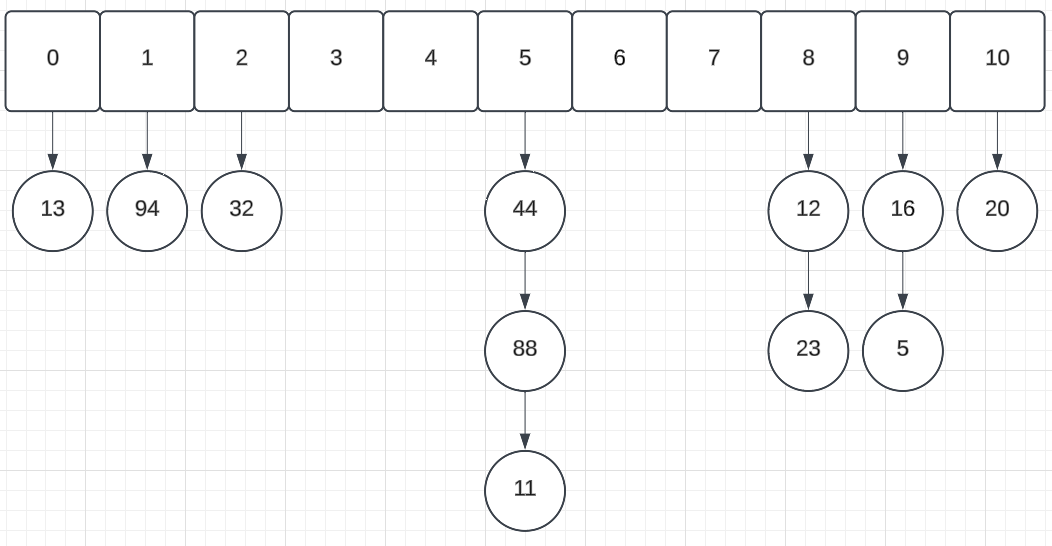
**Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?**

Open addressing can **NOT** tolerate a load factor above 1.

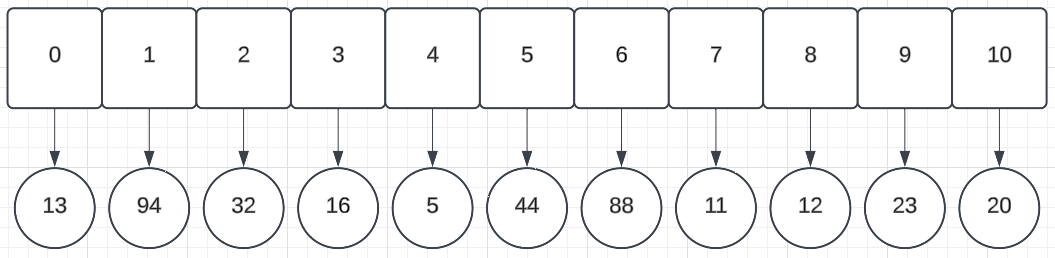
Separate chaining **CAN** handle a load factor above 1. It’s not ideal though

**6-9, 11) 1.5 Points**

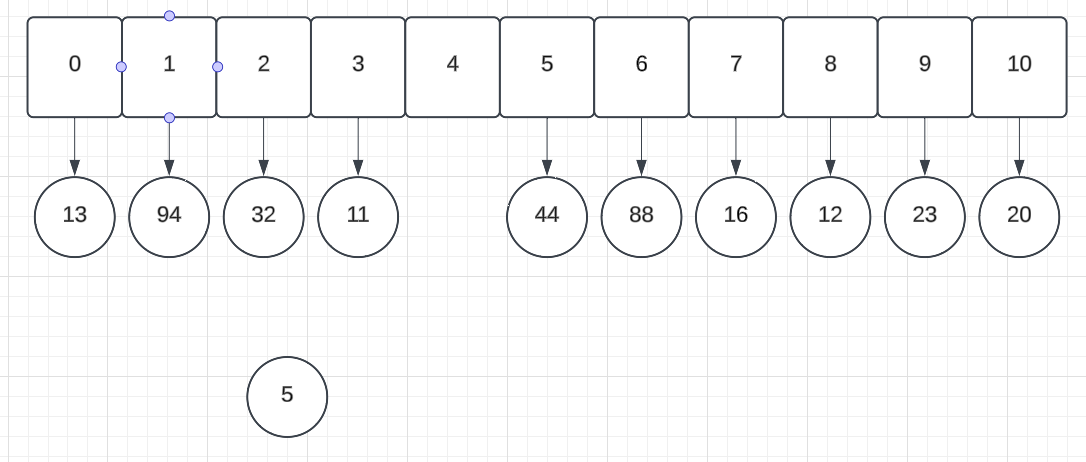
**6) Draw the 11-entry hash table that results from using the hash function h(k) = (3k+5)mod11 when inserting the sequence of keys 12, 44, 13, 88, 23, 94, 11, 32, 20, 16, 5, in that order, assuming collisions are handled by chaining.**



**7) What is the result of Exercise 10.6.6, assuming collisions are handled by linear probing?**



**8) Show the result of Exercise 10.6.6, assuming collisions are handled by quadratic probing, up to the point where the method fails.**



Fails at 5

H(k) = (5 \* 3 + 5) % 11 = (15 + 5) % 11 = 20 % 11 = 9 Collision

Attempt 1 = (9 + 1^2) % 11 = 10 % 11 = 10 **Occupied**

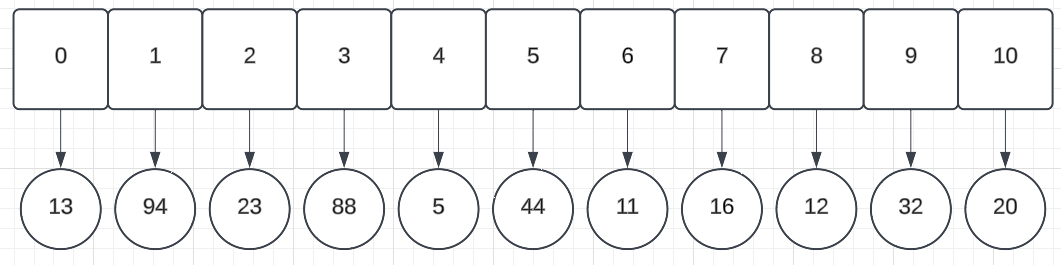
Attempt 2 = (9 + 2^2) % 11 = 13 % 11 = 2 **Occupied**

Attempt 3 = (9 + 3^2) % 11 = 18 % 11 = 7 **Occupied**

Attempt 4 = (9 + 4^2) % 11 = 25 % 11 = 3 **Occupied**

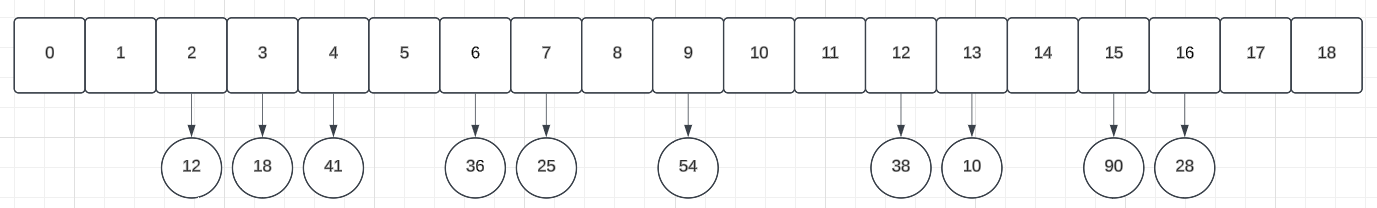
Attempt 5 = (9 + 5^2) % 11 = 34 % 11 = 1 **Occupied – FAILED**

**9) What is the result of Exercise 10.6.6 when collisions are handled by double hashing using the secondary hash function h’(k) = 7 – (kmod7)?**



**11) Show the result of rehashing the hash table shown in Figure 10.2.4 into a table of size 19 using the new hash function h(k) = 3kmod17.**

**Insertion order assumption – 54, 18, 10, 25, 28, 36, 38, 41, 12, 90**



**20) 0.5 Points**

**Describe how a sorted list implemented as a doubly linked list could be used to implement the sorted map ADT.**

Functions:

firsEntry()

We look for the node after the sentinel head node assuming we have nodes within the list

lastEntry()

We look for the node before the sentinel tail node assuming we have nodes within the list

ceilingEntry()

We traverse the linked list and return the first node we find that is equal to or greater than our current key value.

floorEntry()

We traverse the linked list and return current node once node.next is greater than current node

lowerEntry()

We traverse using current node. Once current.next is greater than or equal to our key value, return the current node.

higherEntry()

We traverse using current node. Once current node is greater than our key value, return the current node.

subMap()

We traverse using current node. Once current is greater than or equal to k1, we begin to insert into an ArrayList. Once current node.next is greater than or equal to k2, we stop inserting into the ArrayList and return it.

**22) 1 Point**

**What is the expected running time of the methods for maintaining a maxima set if we insert pairs such that each pair has lower cost and performance than one before it? What is contained in the sorted map at the end of this series of operations? What if each pair had a lower cost and higher performance than the one before it?**

In situation 1 with decreasing cost and decreasing performance:

The maxima set would contain all pairs that are being compared. It is unknown from the question what the cost to performance is. It is just stated that they are both lowering. Thus, it could be that certain people may want to purchase some of those lower performance items because they are lowest cost or offer better cost to performance. Thus, we would retain all of them.

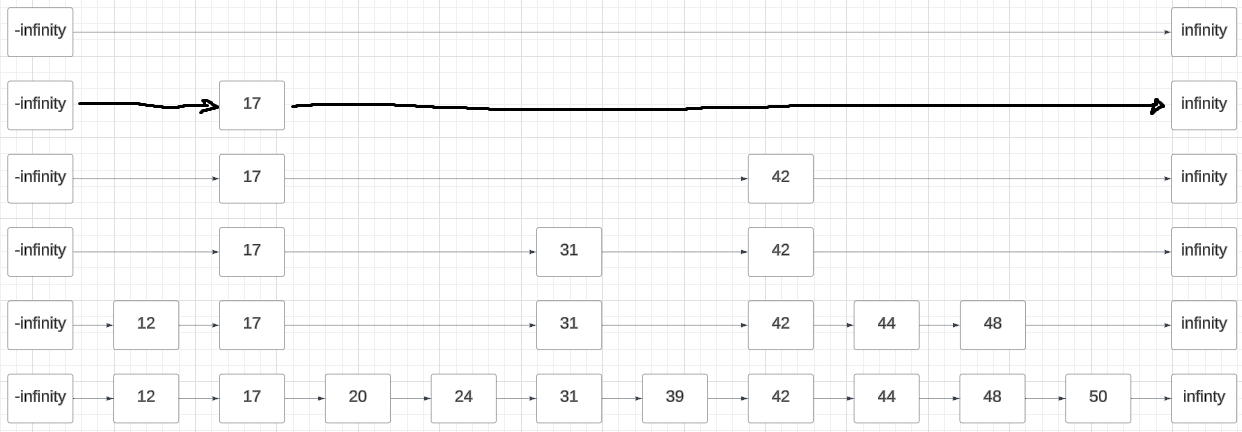
This would O(n^2). O(n) once to check if anything is dominated and then O(n) for insertion leading to O(n^2)

In situation 2 where we are reducing cost and increasing performance, only the last item would be contained in the list. Because the item is both cheaper AND more performant than the previous entry, it strictly dominates all previous entries. All previous entries can thus be removed and leave only the last entry.

This would be O(n) time because we need only compare against the first element before we remove and insert the new strictly better element.

**23) 0.5 Points**

**Draw the result after performing the following series of operations on the skip list shown in Figure 10.4.2: remove(38), put(48,x), put(24,y), remove(55). Use an actual coin flip to generate random bits as needed (and report your sequence of flips).**



Put 48: Heads then tails

Put 24: Tails

Apologies on the one thick line, I ran out of free shapes on lucid.app

**41) 1 Point**

**Describe how to perform a removal from a hash table that uses linear probing to resolve collisions, yet without using a special marker to represent deleted elements. Instead, consider rearranging the contents to replace the deleted element in a way that does not disrupt future operations.**

* Removal is called
  + Save intial hashposition in variable initHashPosition
  + If we don’t find anything in the hash position // Assuming that hash should not be empty because of how we’re going to be removing stuff
    - Return and say that no such element was found
  + If found in position
    - Remove it
  + While current hash position isn’t empty
    - If hashPosition is greater than or equal to size of hash
      * hashPosition = 0
    - Check hash position
    - If element is what we were looking for
      * Remove it
      * Set lastRemoved position equal to this position
      * Break;
    - hashPosition = hashPosition + 1
    - if hashPosition = initHashPosition
      * Return saying that no such element was found
  + While currentPosition is NOT empty AND currentPosition +1 is NOT empty
    - If currentPosition + 1 hash value is greater than our lastRemoved position
      * currentPosition moves one slot forward
      * Continue to next iteration (Skip rest of logic)
    - lastRemoved key = currentPosition + 1 key
    - lastRemoved value = currentPoisition + 1 value
    - lastRemoved moves forward to the currentPosition
    - currentPosition moves one slot forward
  + Delete value and key in lastRemoved
  + return removed item

Essentially, we are going to delete the current slot and save the current position. The current position will be saved in last removed. We will traverse the hash We will check the next slots to see if any of the slots we run into can be moved back to where we last removed an element. We repeat this process until we run into our first blank spot because a blank spot means no linear probe has taken that slot yet. We delete the key and value in the last removed position because no data will be replacing it and then return the initial element we deleted.