# CptS 223 Homework #3 - Heaps, Hashing, Sorting

Due Date: Nov 20th 2020

Please complete the homework problems and upload a pdf of the solutions to blackboard assignment and upload the PDF to Git.

By: Hunter McClure

1. [6] Starting with an empty hash table with a fixed size of 11, insert the following keys in order into three distinct hash tables (one for each collision mechanism): {12, 9, 1, 0, 42, 98, 70, 3}. You are only required to show the final result of each hash table. In the **very likely** event that a collision resolution mechanism is unable to successfully resolve, simply record the state of the last successful insert and note that collision resolution failed. For each hashtable type, compute the hash as follows:

hashkey(key) = (key * key + 3) % 11

Separate Chaining (buckets)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | | 0 | 12<br>1<br>98 | | | 9<br>42 | 70 | | |

To probe on a collision, start at hashkey(key) and add the current probe(i') offset. If that bucket is full, increment i until you find an empty bucket.

Linear Probing: probe(i') = (i + 1) % TableSize

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | | 0 | 12 | 1 | 98 | 9 | 42 | 70 | |

Quadratic Probing: probe(i') = (i * i + 5) % TableSize

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 12 | | 42 | 9 | 98 | | 1 |

***Collision resolution failure: last successful node was 98.***

2. [3] For implementing a hash table. Which of these would probably be the best initial table size to pick?

Table Sizes:

1 100 101 15 500

Why did you choose that one?

Because it isn't too big incase we need to rehash and it also is a prime number.

3. [4] For our running hash table, you'll need to decide if you need to rehash.
You just inserted a new item into the table, bringing your data count up to 53491 entries. The table's vector is currently sized at 106963 buckets.

- Calculate the load factor (λ):

  53491/106963 = 0.50

- Given a linear probing collision function should we rehash? Why?

We should rehash because since the load factor is half of the total size, the collisions will start to occur more often and the probing will start to take a longer amount of time. (Be aware this also depends on your hash function).

- Given a separate chaining collision function should we rehash? Why?

We do not need to rehash right now. You would usually rehash when the load factor is 0.75 and this is because chaining is meant to hold multiple keys in one bucket instead of having to go to the next available node.

4. [4] What is the Big-O of these actions for a well designed and properly loaded hash table with N elements?

| Function | Big-O complexity |
|---|---|
| Insert(x) | O(1) |
| Rehash() | O(n) |
| Remove(x) | O(1) |
| Contains(x) | O(1) |

7. [3] I grabbed some code from the Internet for my linear probing based hash table at work because the Internet's always right (totally!). The hash table works, but once I put more than a few thousand entries, the whole thing starts to slow down. Searches, inserts, and contains calls start taking *much* longer than O(1) time and my boss is pissed because it's slowing down the whole application services backend I'm in charge of. I think the bug is in my rehash code, but I'm not sure where. Any ideas why my hash table starts to suck as it grows bigger?

```
/**
* Rehashing for linear probing hash table.
*/
void rehash( )
{
    ArrayList<HashItem<T>> oldArray = array;

    array = new ArrayList<HashItem<T>>( 2 * oldArray.size() );

    for( int i = 0; i < array.size(); i++ )
        array.get(i).info = EMPTY;
// Copy old table over to new larger array
    for( int i = 0; i < oldArray.size(); i++ ) {
        if( oldArray.get(i).info == FULL )
        {
            addElement(oldArray.get(i).getKey(),
                    oldArray.get(i).getValue());
        }
    }
}
```

The reason it starts to suck as the table grows bigger is because the code makes a copy of the array and so when you have a huge array and you make a copy of that array and then double the size of one of the copies, the program will start to hit a wall and slow down due to memory. Once the hash table goes over the 0.5 load factor, depending on when you rehash will also depend on how much slower the program will run.

8. [4] Time for some heaping fun! What's the time complexity for these functions in a Java Library priority queue (binary heap) of size N?

| Function | Big-O complexity |
|---|---|
| push(x) | O(1) |
| top() | O(1) |
| pop() | O(log n) |
| PriorityQueue(Collection<? extends E> c) // BuildHeap | O(n) |

9. [4] What would a good application be for a priority queue (a binary heap)? Describe it in at least a paragraph of why it's a good choice for your example situation.

A good application would be customer service based on how bad the situation is. In a situation like this, we could put the people on hold who have less of an emergency for a longer amount of time (or lower in the queue) while the people who need service because of a higher emergency would be higher in the queue. We could also do it based on time. If service is needed for someone who only needs 2 minutes of the service's time compared to someone who needs 20 minutes of the services time, then we could put the shorter amount of people first so they can get as many people in as they can before the longer service.

10. [4] For an entry in our heap (root @ index 1) located at position i, where are it's parent and children?

Parent:
The parent can be found at i/2.

Children:
However, for the children, there are two different scenarios. If you want to find the node's left child then it will be 2*i, but if you want to find the node's right child then it will be at 2*i + 1

11. [6] Show the result of inserting 10, 12, 1, 14, 6, 5, 15, 3, and 11, one at a time, into an initially empty binary heap. Use a 1-based array like the book does. After insert(10): min heap

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | | | | | | | | | | |

After insert (12):

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 12 | | | | | | | | | |

etc:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 12 | 10 | | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 12 | 10 | 14 | | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 10 | 14 | 12 | | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 5 | 14 | 12 | 10 | | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 5 | 14 | 12 | 10 | 15 | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 | | |

12. [4] Show the same result (only the final result) of calling buildHeap() on the same vector of values: {10, 12, 1, 14, 6, 5, 15, 3, 11}

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 | | |

13. [4] Now show the result of three successive deleteMin / pop operations from the prior heap:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 11 | 12 | 10 | 15 | 14 | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 10 | 11 | 12 | 14 | 15 | | | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 6 | 11 | 10 | 15 | 12 | 14 | | | | | |

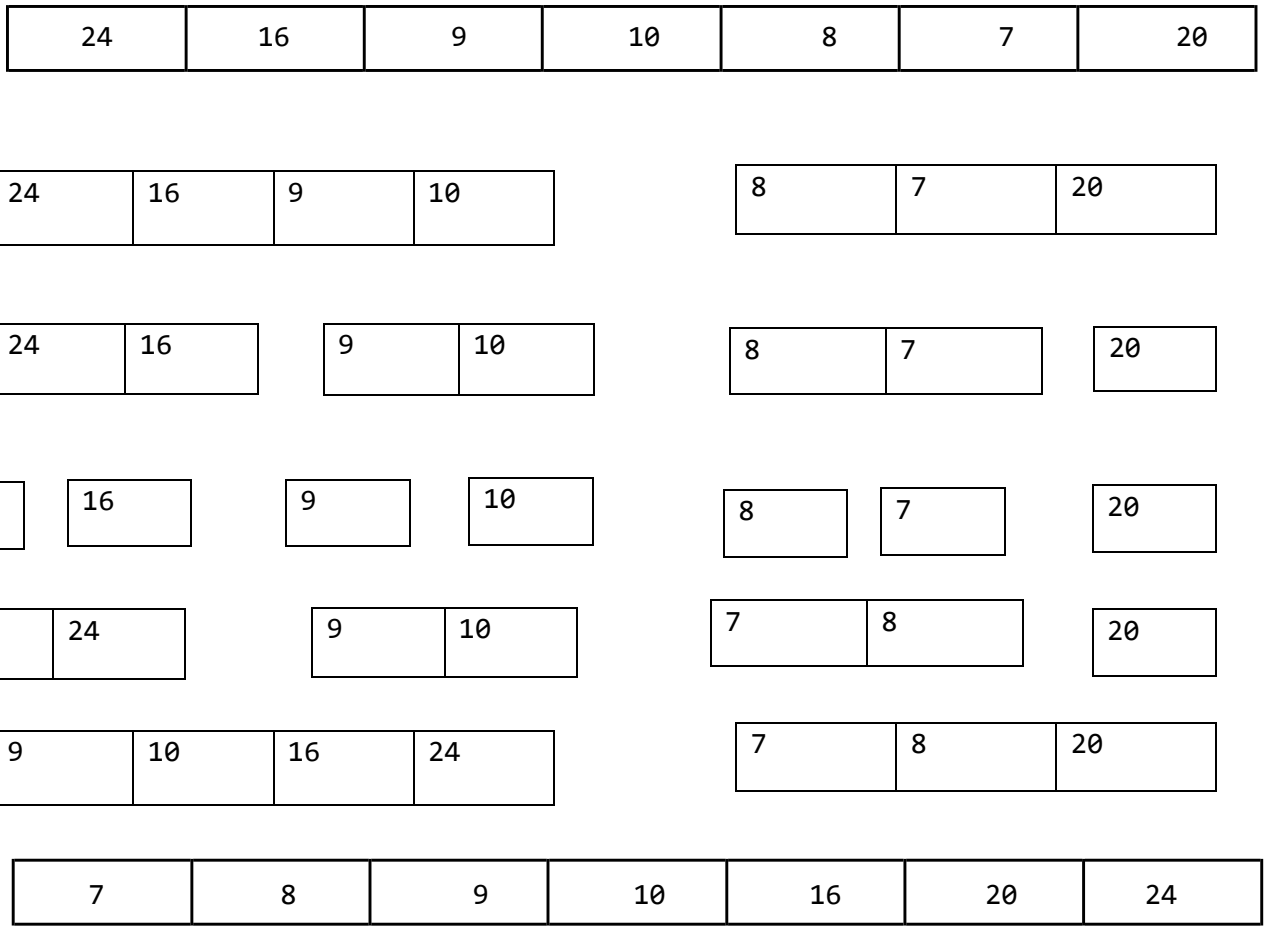
14. [4] What are the average complexities and the stability of these sorting algorithms:

| Algorithm | Average complexity | Stable (yes/no)? |
|---|---|---|
| Bubble Sort | $\Theta(n^2)$ | Yes |
| Insertion Sort | $\Theta(n^2)$ | Yes |
| Heap sort | $\Theta(n\log n)$ | No |
| Merge Sort | $\Theta(n\log n)$ | Yes |
| Radix sort | $\Theta(m * n)$ | Yes |
| Quicksort | $\Theta(n\log n)$ | No |

15. [3] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

The major differences between Merge and Quick is that merge is stable and quick is not. However, Quick is a bit faster in sorting compared to Merge Sort. Also Merge Sort uses a bit more extra memory since it is not an in-place sort. Also Merge sorts preserve the relative order of the elements. Thus, since quick sort is faster and uses less memory, even though it is not stable, many languages use quick over merge.

16. [4] Draw out how Mergesort would sort this list:



17. [4] Draw how Quicksort would sort this list:

| 24 | 16 | 9 | 10 | 8 | 7 | 20 |
|---|---|---|---|---|---|---|

| 24 | 16 | 9 | ***10 (pivot)*** | 8 | 7 | 20 |
|---|---|---|---|---|---|---|

| 24 | 16 | 9 | 20 | 8 | 7 | ***10 (pivot)*** |
|---|---|---|---|---|---|---|

| 7 | 16 | 9 | 20 | 8 | 24 | ***10 (pivot)*** |
|---|---|---|---|---|---|---|

| 7 | 8 | 9 | 20 | 16 | 24 | ***10 (pivot)*** |
|---|---|---|---|---|---|---|

| 7 | 8 | 9 | ***10 (pivot)*** | 16 | 24 | 20 |
|---|---|---|---|---|---|---|

Let me know what your pivot picking algorithm is (if it's not obvious): The Pivot picking algorithm is based around 10 since it is the middle value.