Bogdan Shmat
https://github.com/boshma/Cpts223Homework3.git

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.

[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 | 1 | 98			9 | 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 Assuming I starts at 0

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 Assuming I starts at 0

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

1. [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?

101 (its prime)

2. [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 = ~.5001

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

  Yes, the load factor is .5 so the hash functions will still start getting slower as it will run into more collisions.

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

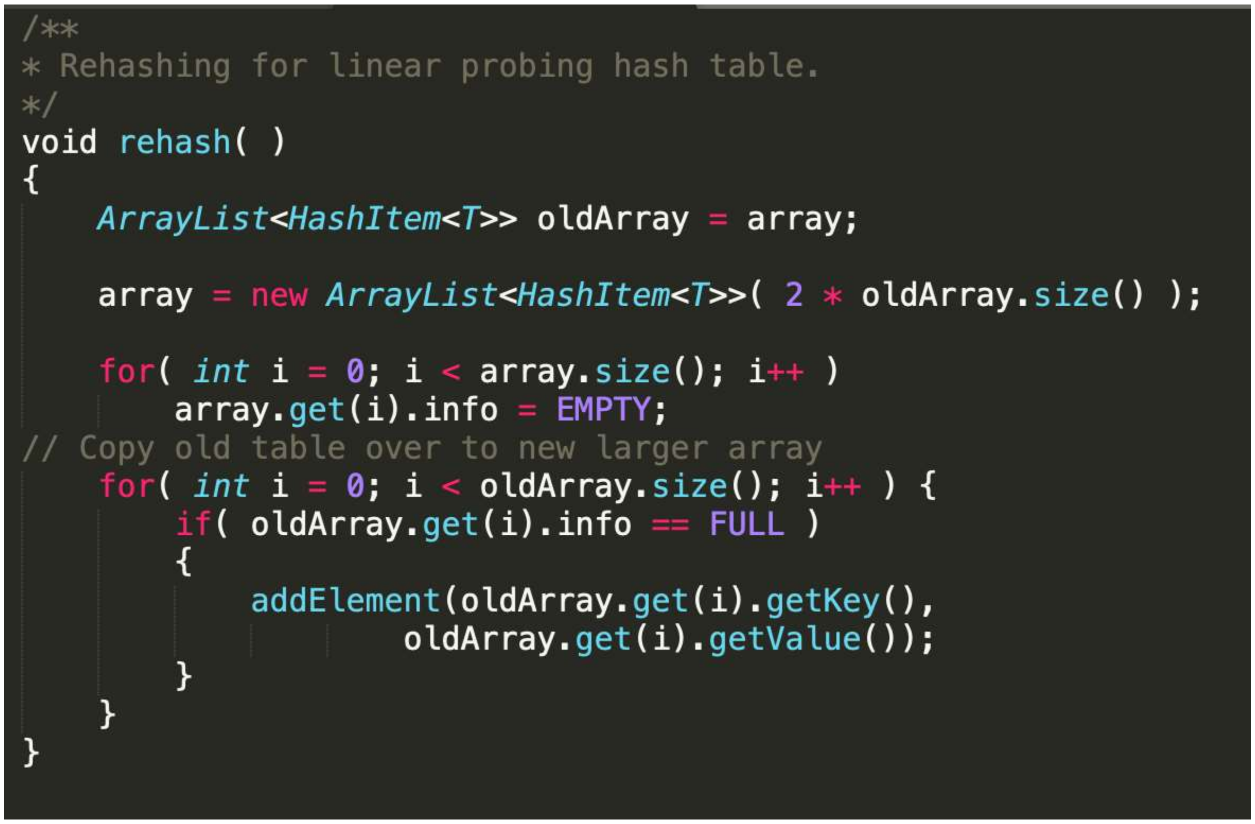
  No, the load factor is .5 which is lower then .75. Chaining wont become O(N) until the load factor gets up to .75.

3. [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?

Since the best table size for collisions is prime, the oldArray.size() may grow and become a number with 1 non-zero leading digit (for example, 1000,4000,50000) followed by multiple zeros and it would get stuck in that loop creating a huge amount of collisions. Making sure that the table size is always a prime number will make sure collisions are kept to a minimum.



```
/**
* 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());
        }
    }
}
```

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(log(n))
top()	O(1) for max heap, o(nlog(n)) for min heap
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 for a priority queue is prioritizing patients in a hospital waiting lobby. As patients enter the waiting lobby, if their condition is severe enough, they are given more priority. This prevents someone with a severe injury being forced to wait very long for people with very minor conditions. A priority queue would prevent people with serious conditions from perhaps dying in the hospital lobby by pushing people with minor conditions back in the queue because they have no risk from delayed treatment.

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

Parent: Floor value of i/2

Children:

Left child: 2i

Right child : 2i+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):

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:

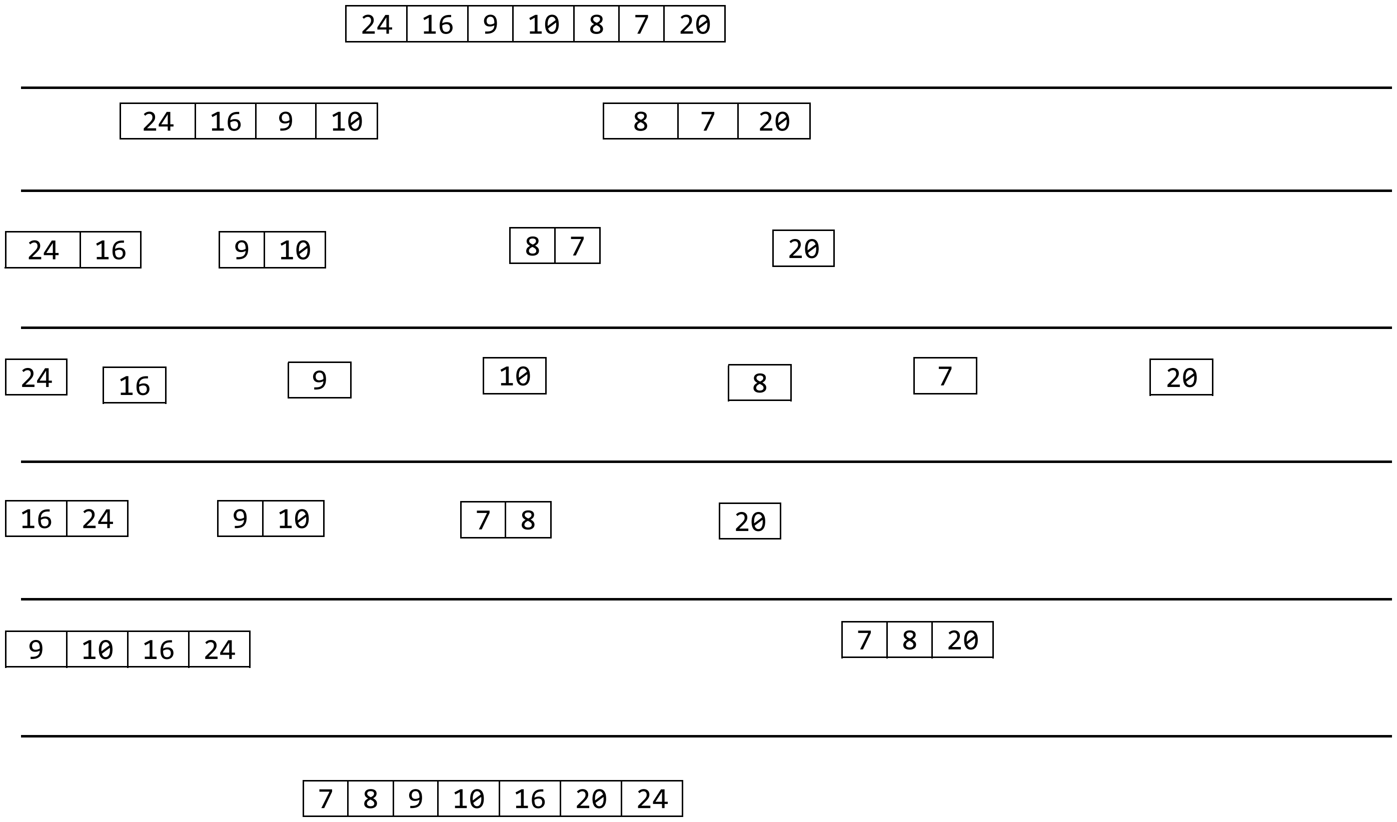
Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	O(N^2)	Yes
Insertion Sort	O(N^2)	Yes
Heap sort	O(Nlog(N))	No
Merge Sort	O(Nlog(N))	Yes
Radix sort	O(N)	Yes
Quicksort	O(Nlog(N))	No

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

Quicksort uses a partitioning swapping algorithm that pivots around a number, recursively swapping and picking a new pivot using divide and conquer until the array is sorted. Mergesort (also divide and conquer) recursively divides the array in half until all elements are partitioned, and then reattaches them in sorted order which leaves a sorted array. Some factors that influence why languages choose one over another is the worst time complexity of each. Mergesort is O(nlog(n)) while Quicksort is O(n^2). Quicksort is also more efficient with smaller data sets and vice versa. Mergesort also requires additional memory space to store arrays while quicksort does not. These factors will certainly play a role for when a programmer decides which sort is best depending on what language is being used.

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

24	16	9	10	8	7	20



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

24	16	9	10	8	7	20

Pivot = Size of array/2 (floor value). 7/2 = 3.5 == index 3. Pivot = 10

I = 24, J = 7, Swap

7	16	9	10	8	24	20

Now I = 16, J = 8, Swap

7	8	9	10	16	24	20

Now I = 16, J = 9, We can partition at 10 as 7,8,9 < 10 and 16,24,20 > 10

The left hand side of 10 is already sorted so there is no further work to be done.
For right hand side we will have 24 as the pivot of the partitioned size 3 array.

16	24	20

I = 24, J = 20 Swap

16	20	24

Both sides are sorted and the final array is sorted.

7	8	9	10	16	20	24

Let me know what your pivot picking algorithm is (if it's not obvious):

Let the pivot equal the floor value of the size of the array divided by 2.
(For array based at 0).
I = value from left greater then pivot
J = value from right smaller then pivot