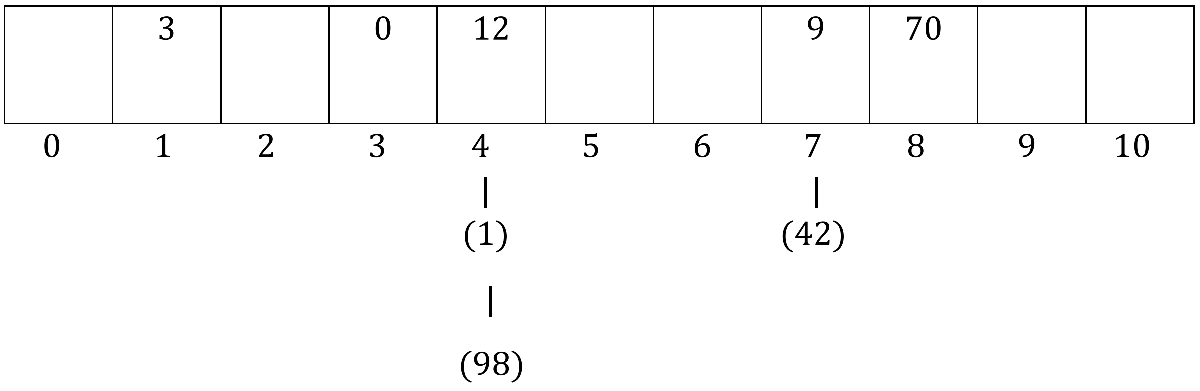
Github URL: https://github.com/krmesfin42/HW3.git

HW3CptS233

1.

{12, 9, 1, 0, 42, 98, 70, 3}
Hash key(key) = (key * key + 3) % 11
Hash key(12) = (12 * 12 + 3) % 11
Hash key(12) = 4
Hash key(9) = 7
Hash key(1) = 4 (collision)
Hash key(0) = 3
Hash key(42) = 7 (collision)
Hash key(98) = 4 (collision)
Hash key(70) = 8
Hash key(3) = 1



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

Linear Probing: probe(i') = (i + 1) % 11

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

Quadratic Probing: probe(i') = (i * i + 5) % 11

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

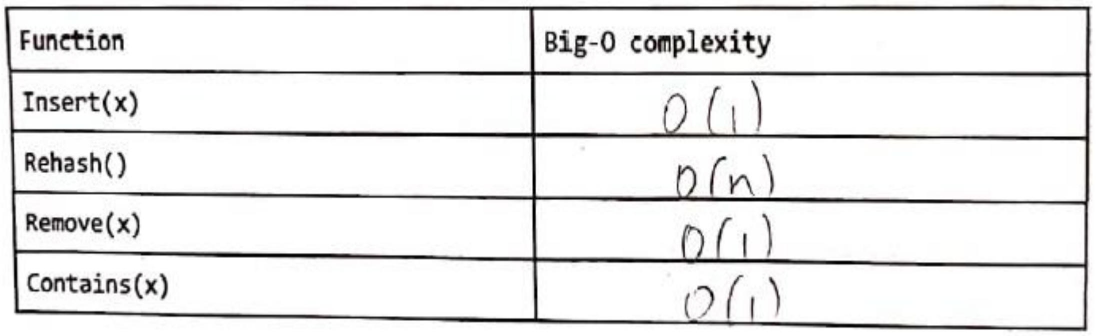
2.

I’d choose 500. To get a value below the default 0.75, we need to pick a larger size table since we don’t know the number of entries.

3.

- Load factor = 53491/106963 = 0.5
- The collision makes the load factor to be greater than 0.5 so we need to increase the size of the table to minimize the time to 0.5.
- We don’t need to rehash because each bucket are independent and different values can be placed at the same index. In this case, the load factor is 0.5 so $0.5 < 0.75$ which we don’t need to rehash.

4.

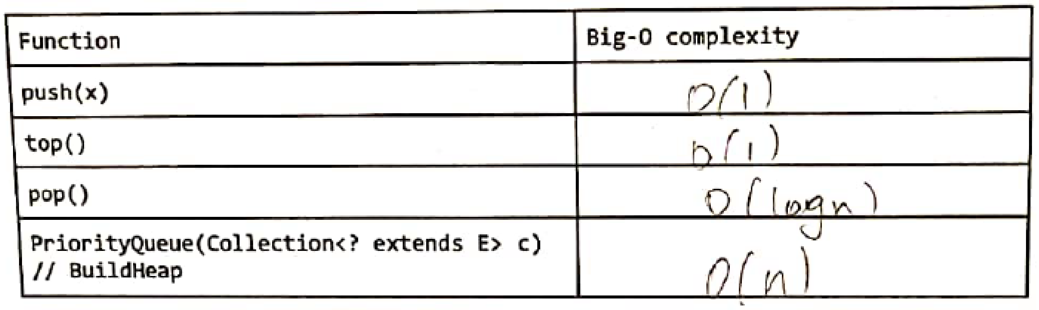


Function	Big-O complexity
Insert(x)	O(1)
Rehash()	O(n)
Remove(x)	O(1)
Contains(x)	O(1)

7.

- The one main reason that led the program to get slower is as the entries gets larger, all the elements need to shift so that the new elements inserted in the new array. And that increase the time for search, inserts and contains calls. The 2nd line of the code which copies the old array to new one needs to be removed.

8.



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

9.

- One good example I think is online shopping store. When customers purchase and checkout, priority queue come to place to give the first customer finish shopping since its first in first out and move to the next one. Also, the most watched/checked item in a website will be shown on our homepage or when we are using other websites.

10.

- Parent – i/2
- Children: left child: 2*i
  Right child: 2*i + 1

11.

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		

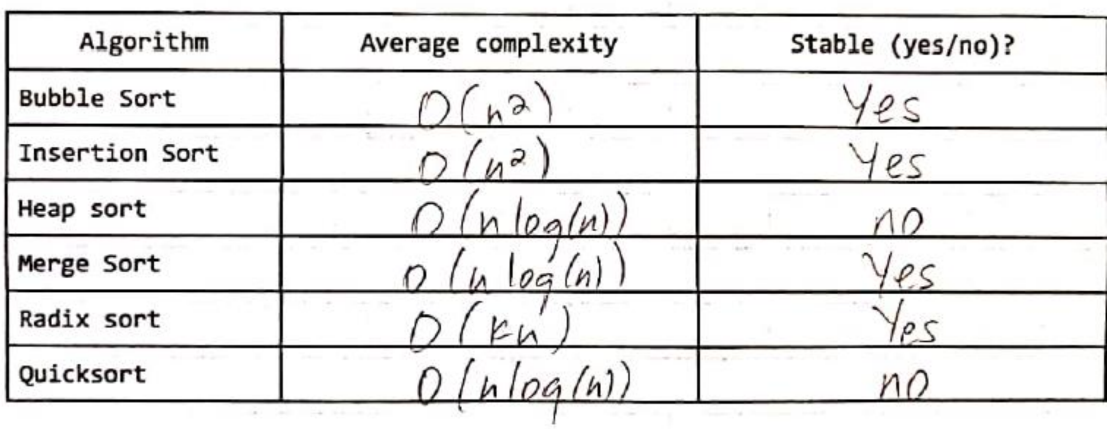
12.

1	3	5	6	12	10	15	14	11		

13.

delete(1)	3	6	5	11	12	10	15	14			
delete(3)	5	6	10	11	12	14	15				
delete(5)	6	11	10	15	12	14					

14.



Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	$O(n^2)$	Yes
Insertion Sort	$O(n^2)$	Yes
Heap sort	$O(n\log(n))$	no
Merge Sort	$O(n\log(n))$	Yes
Radix sort	$O(kn)$	Yes
Quicksort	$O(n\log(n))$	no

15.

- Quicksort uses a pivot for partioning the elements whereas merge sort uses additional storage for sorting the auxiliary array.
- Languages pick merge sort for larger array size over quicksort and quicksort for smaller array size since the data is sorted in main memory over merge sort.

16.

24	16	9	10	8	7	20

```
24  16  9  10        8   7   20

24  16     9  10     8   7       20
16  24     9  10     7   8       20

9  10  16  24         7  8   20
9  10  16  24            8   20
9  10  16  24                20
  10   16  24               20
   16  24                   20
     24                     20
     24
```

7	8	9	10	16	20	24

17.

24	16	9	10	8	7	20	
			pivot				
24	16	9	20	8	7	10	switch 20 & 10
7	16	9	20	8	24	10	
7	16	9	20	8	24	10	
7	8	9	20	16	24	10	
7	8	9	20	16	24	10	
7	8	9	20	16	24	10	
7	8	9	10	16	24	20	switch 10&20
7	9	8	10	16	24	20	
7	9	8	10	16	24	20	
7	8	9	10	16	24	20	
7	8	9	10	16	20	24	
7	8	9	10	16	20	24	
7	8	9	10	16	20	24	
7	8	9	10	20	16	24	
7	8	9	10	16	20	24	

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

7	8	9	10	16	20	24