

Group Write up

Name: Jeremy Liem
Qinghao Meng

1. $Parent(i) = (i - 1) / 4$
 $Child(i, j) = 4i + j$ where $1 \leq j < 5$

This formula uses an array with index starting at 0.

2.

Instead of compare the parent node to every of their children, we can create a helper method to compare and get the smallest children. Then, compare if the parent node is smaller than its smallest children. If so, we are done with percolate down method. Otherwise, we will swap the parent node and its smallest children.

Design: First, check if the node has children. If so, we will loop through every of its children and get the smallest children. Then, we compare it with the node(so we only need to compare one time between parent and the children). If the parent is bigger than its smallest children, we swap them. Otherwise, the loop will end.

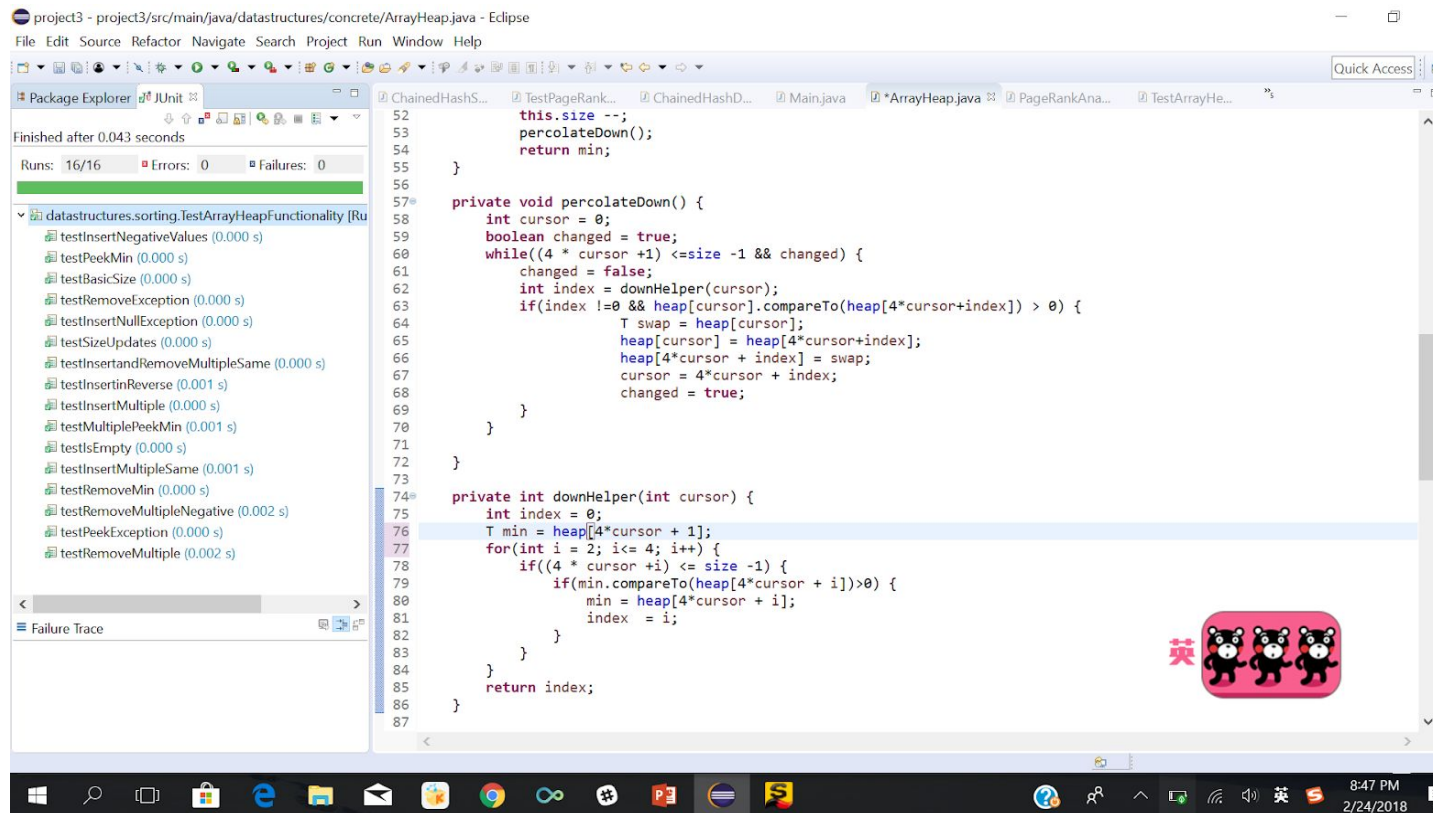
Challenge: how to determine when we should stop the percolate down method and exit the while loop.

My solution: Initial a boolean variable as a condition of while loop(another one is the node has to have a child). We initialize the boolean to true in order to go into the loop at the first time., and set to false when the while loop start.

In the loop, If we swapped the parent and children, we changed the boolean to true, and the while loop will keep going.

The run time before and after the change:

Before:



project3 - project3/src/main/java/datastructures/concrete/ArrayHeap.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit


Finished after 0.043 seconds

Runs: 16/16 Errors: 0 Failures: 0

datastructures.sorting.TestArrayHeapFunctionality [Run]

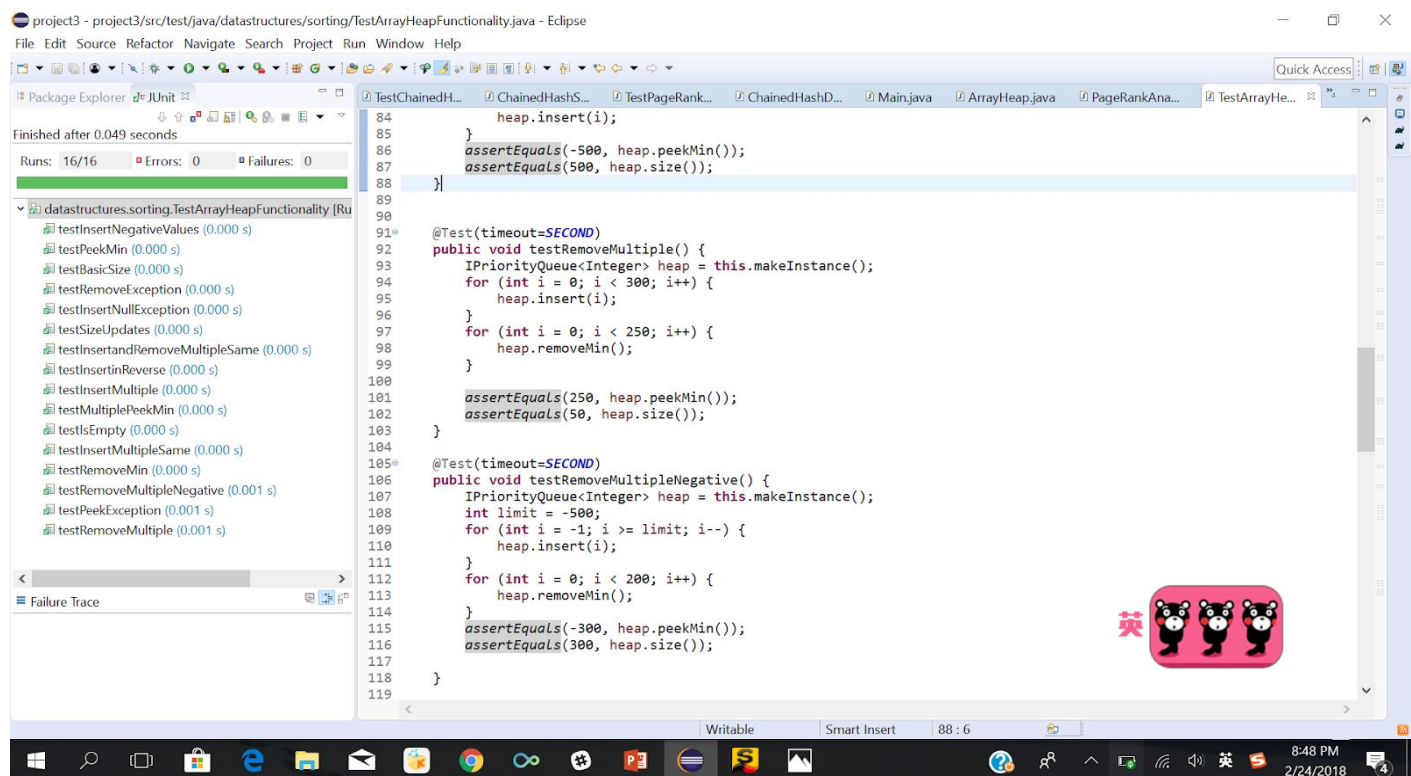
- testInsertNegativeValues (0.000 s)
- testPeekMin (0.000 s)
- testBasicSize (0.000 s)
- testRemoveException (0.000 s)
- testInsertNullException (0.000 s)
- testSizeUpdates (0.000 s)
- testInsertandRemoveMultipleSame (0.000 s)
- testInsertinReverse (0.001 s)
- testInsertMultiple (0.000 s)
- testMultiplePeekMin (0.001 s)
- testIsEmpty (0.000 s)
- testInsertMultipleSame (0.001 s)
- testRemoveMin (0.000 s)
- testRemoveMultipleNegative (0.002 s)
- testPeekException (0.000 s)
- testRemoveMultiple (0.002 s)

```
52     this.size--;
53     percolateDown();
54     return min;
55 }
56
57 private void percolateDown() {
58     int cursor = 0;
59     boolean changed = true;
60     while((4 * cursor + 1) <= size - 1 && changed) {
61         changed = false;
62         int index = downHelper(cursor);
63         if(index != 0 && heap[cursor].compareTo(heap[4*cursor+index]) > 0) {
64             T swap = heap[cursor];
65             heap[cursor] = heap[4*cursor+index];
66             heap[4*cursor + index] = swap;
67             cursor = 4*cursor + index;
68             changed = true;
69         }
70     }
71 }
72
73 private int downHelper(int cursor) {
74     int index = 0;
75     T min = heap[4*cursor + 1];
76     for(int i = 2; i <= 4; i++) {
77         if((4 * cursor + i) <= size - 1 {
78             if(min.compareTo(heap[4*cursor + i]) > 0) {
79                 min = heap[4*cursor + i];
80                 index = i;
81             }
82         }
83     }
84     return index;
85 }
86 }
87
```

英 

8:47 PM 2/24/2018

After:



project3 - project3/src/test/java/datastructures/sorting/TestArrayHeapFunctionality.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer JUnit


Finished after 0.049 seconds

Runs: 16/16 Errors: 0 Failures: 0

datastructures.sorting.TestArrayHeapFunctionality [Run]

- testInsertNegativeValues (0.000 s)
- testPeekMin (0.000 s)
- testBasicSize (0.000 s)
- testRemoveException (0.000 s)
- testInsertNullException (0.000 s)
- testSizeUpdates (0.000 s)
- testInsertandRemoveMultipleSame (0.000 s)
- testInsertinReverse (0.000 s)
- testInsertMultiple (0.000 s)
- testMultiplePeekMin (0.000 s)
- testIsEmpty (0.000 s)
- testInsertMultipleSame (0.000 s)
- testRemoveMin (0.000 s)
- testRemoveMultipleNegative (0.001 s)
- testPeekException (0.001 s)
- testRemoveMultiple (0.001 s)

```
84     heap.insert(i);
85 }
86 assertEquals(-500, heap.peekMin());
87 assertEquals(500, heap.size());
88 }
89
90
91 @Test(timeout=SECOND)
92 public void testRemoveMultiple() {
93     IPriorityQueue<Integer> heap = this.makeInstance();
94     for (int i = 0; i < 300; i++) {
95         heap.insert(i);
96     }
97     for (int i = 0; i < 250; i++) {
98         heap.removeMin();
99     }
100     assertEquals(250, heap.peekMin());
101     assertEquals(50, heap.size());
102 }
103
104 @Test(timeout=SECOND)
105 public void testRemoveMultipleNegative() {
106     IPriorityQueue<Integer> heap = this.makeInstance();
107     int limit = -500;
108     for (int i = -1; i >= limit; i--) {
109         heap.insert(i);
110     }
111     for (int i = 0; i < 200; i++) {
112         heap.removeMin();
113     }
114     assertEquals(-300, heap.peekMin());
115     assertEquals(300, heap.size());
116 }
117 }
118
119
```

英 

Writable Smart Insert 88 : 6

8:48 PM 2/24/2018

From the images, we can see many for the removeMin test has its run-time reduced from 0.002s - 0.001s.

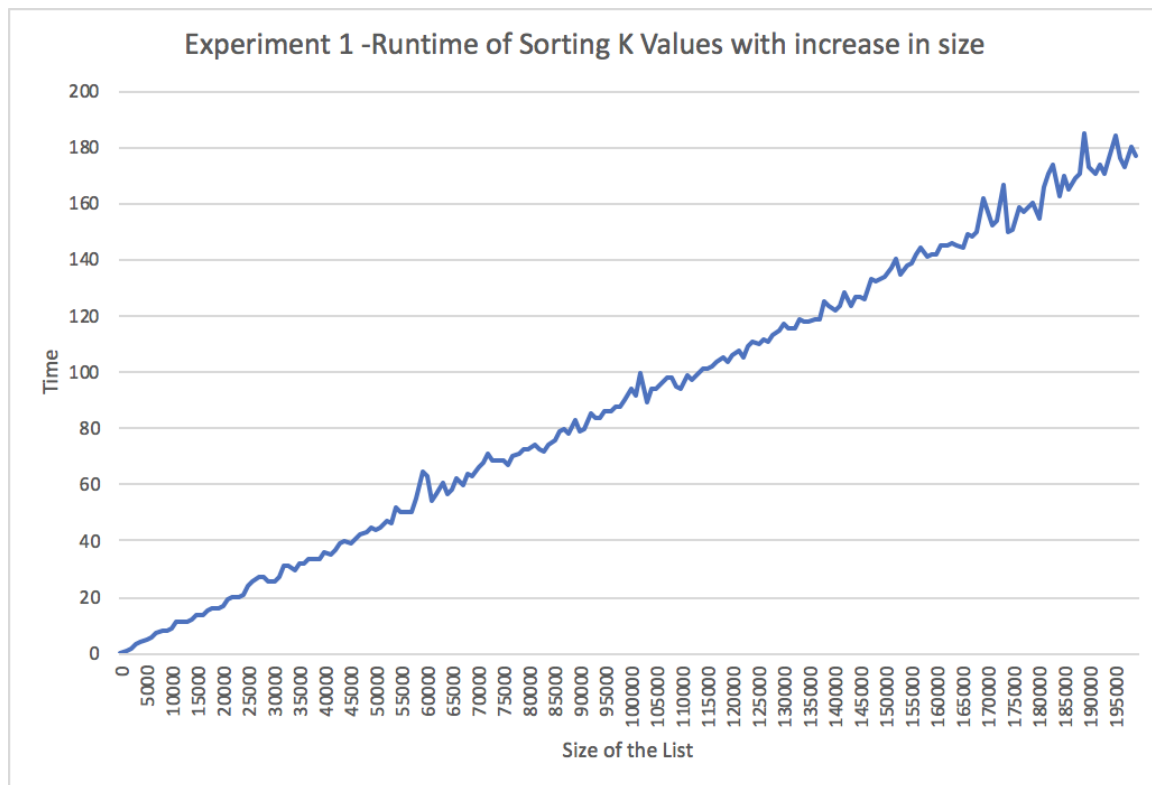
3.

Experiment 1

a) This test calculates the time to sort top 500 values from a list that increases its size every single iteration.

b) Our Hypothesis is that the time would increase linearly. This is because the runtime for topKSort is $O(n \log(k))$. n is the size of the list while K is the number of output expected. Since n is the constant that is changing, as n increases we should expect it to have a larger runtime. Therefore, it is why I believe that the graph would increase linearly.

c)

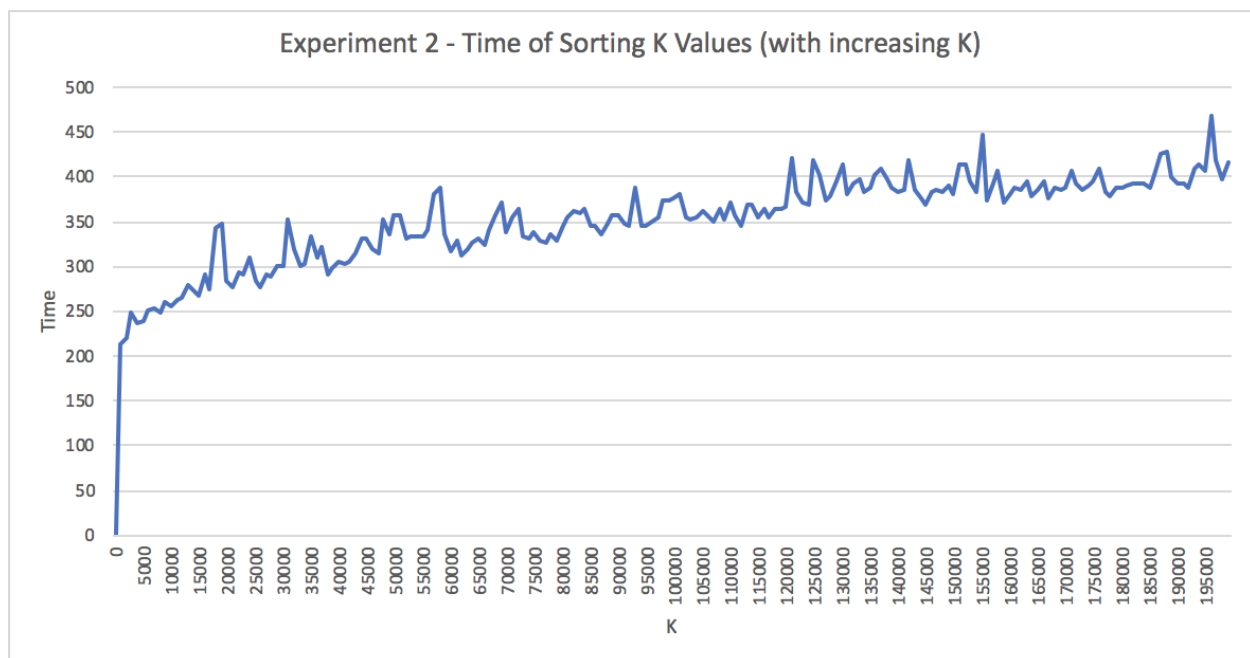


d) Our Hypothesis is correct as the graph shows the runtime increasing linearly as the size of the list increases. As stated in the hypothesis, this is because we are changing the list size and as runtime is $O(n \log(k))$, we are only changing n . As n increases, the runtime increases linearly. However, there is a bit of noise when the size of the list is going to about 200,000. This may be because since the size of the list is bigger, the number needed to sort may be higher (worst case).

or lower(best case). This is most likely what would cause the difference in runtimes. Overall, we should expect to see the graph grow linearly with a few noise.

Experiment 2

- a) This test calculates the time to sort top K values from a constant list size at every single iteration. In this test, instead of changing the list size, we increase K.
- b) Our hypothesis would be that the time should grow logarithmic. This is because we are changing K and since the runtime should be $O(n \log(k))$. We can expect that since K is the one increasing, we expect that that the graph to increase logarithmically.
- c)



d) Our hypothesis is partially correct as it shows that increases logarithmically. We did not expect that the runtime would be 0 when we have to sort 0 values, which would be true but we failed to predict that. However the rest of graphs grow as predicted since it increases logarithmically. There is also noise in this case and this happen because as K increases, we need to sort more and at the worst case, we should sort more and there would be jumps in runtime. Overall, the graph's shape is logarithmic.

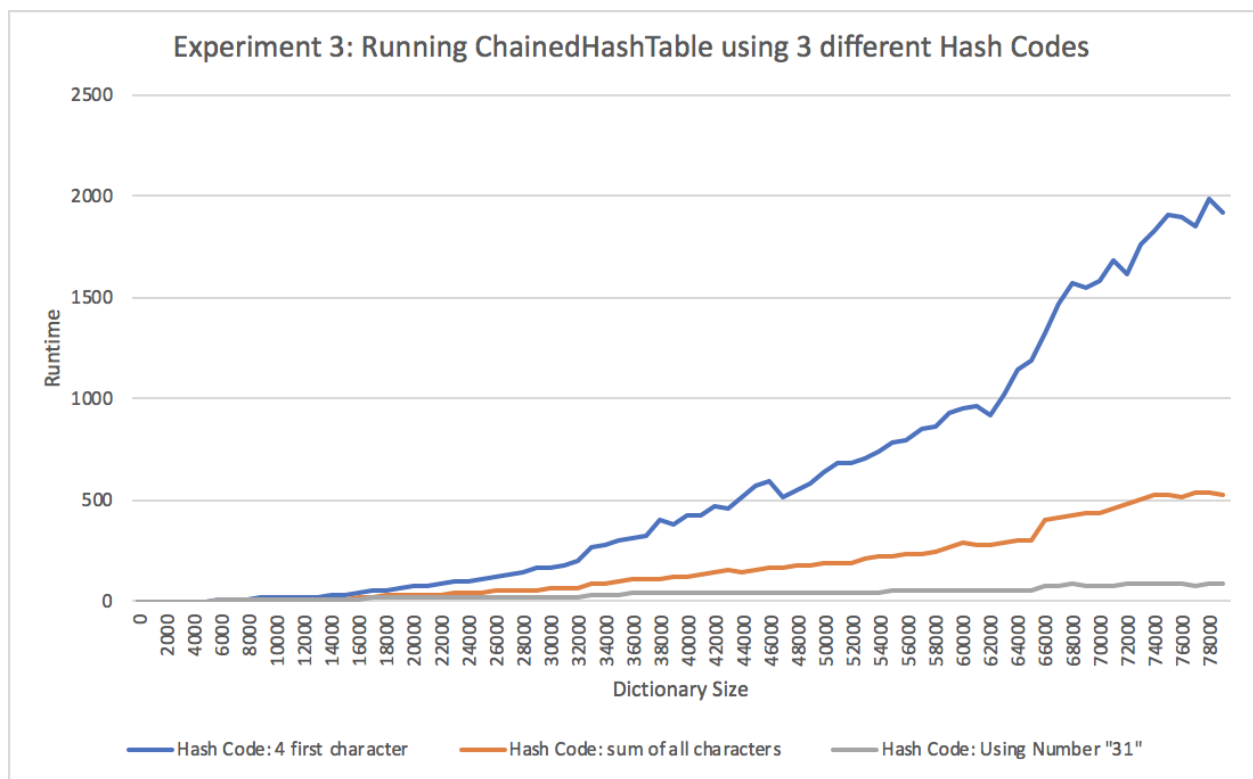
Experiment 3

- a) Experiment 3 is testing the time of adding KVpair to the Chained Hash Dictionary. Test one sets the hashcode of each key by add the value of the first four chars. test2 calculates the hashcode of the key by summing up the value of every characters. Test3 calculates

the hash code in a similar way except it times the old hashcode value by 31 before adding the value of the new char.

b) hypothesis: Test 1 should take the most time. It randomly generate a Char Array and add the chars into Chained Hash Dictionary. In test1, the test set the hashcode of each key by add the value of the first four chars. It will take a long time since it's very easy for many keys to has a same hashcode. The test2 will be faster than test1, since it calculate the hashcode of the key by summing up the value of every characters. Test3 will be the fastest, since it reduces the change the collision of the hashcode progressively.

c)



d) Our Hypothesis looks to be correct according to the graph. It shows test1 is the slowest, test2 is the second slowest and test3 is the fastest. As stated above, this is probably because test1 uses the value of the first 4 chars and because of this, it causes a lot of collisions when there are many similar inputs. Test2 is neither the slowest or the fastest because that it sums up the value of the characters, which I think has a smaller chance of collision compared to test1. Test3 will most likely have the lowest number of collisions due to the calculation of hash code using the magic number "31".