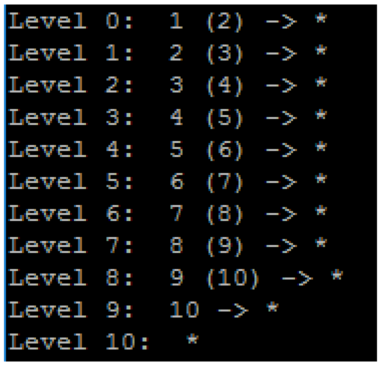
Jay Offerdahl
EECS 560
Lab 13-14 Report Part 2

Verifying the Construction of a Pairing Heap

After writing the code to insert into a pairing heap, I am confident that I have properly constructed the data structure. I have setup a basic GUI in my main.cpp class to handle user interaction, and through this, I was able to test several different cases of input while building a pairing heap. For example, when inserting values in decreasing order, the output will simply be a heap with nodes having left children down to the first value inserted. Please note, my output below shows numbers in parentheses which denotes the value of the left child carried by that node.

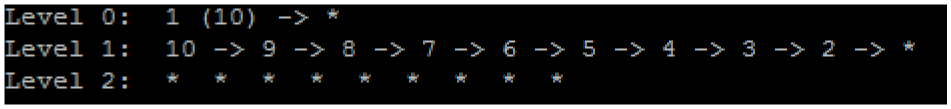
Inserting 10, 9, 8, 7, 6, 5, 4, 3, 2, 1:



```
Level 0:  1 (2) -> *
Level 1:  2 (3) -> *
Level 2:  3 (4) -> *
Level 3:  4 (5) -> *
Level 4:  5 (6) -> *
Level 5:  6 (7) -> *
Level 6:  7 (8) -> *
Level 7:  8 (9) -> *
Level 8:  9 (10) -> *
Level 9:  10 -> *
Level 10:  *
```

When inserting in increasing order, the smallest value will always stay at the root, and all of the following values will be sent to the second level, as shown below:

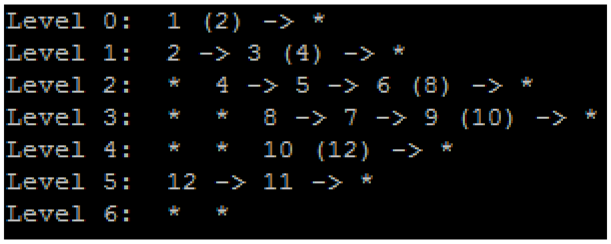
Inserting 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



```
Level 0:  1 (10) -> *
Level 1:  10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> *
Level 2:  *  *  *  *  *  *  *  *  *
```

Next, to generate a more complex heap, I tried inserting values which would cause more ‘zig-zags’.

Inserting 10, 11, 12, 9, 6, 7, 8, 3, 5, 4, 1, 2



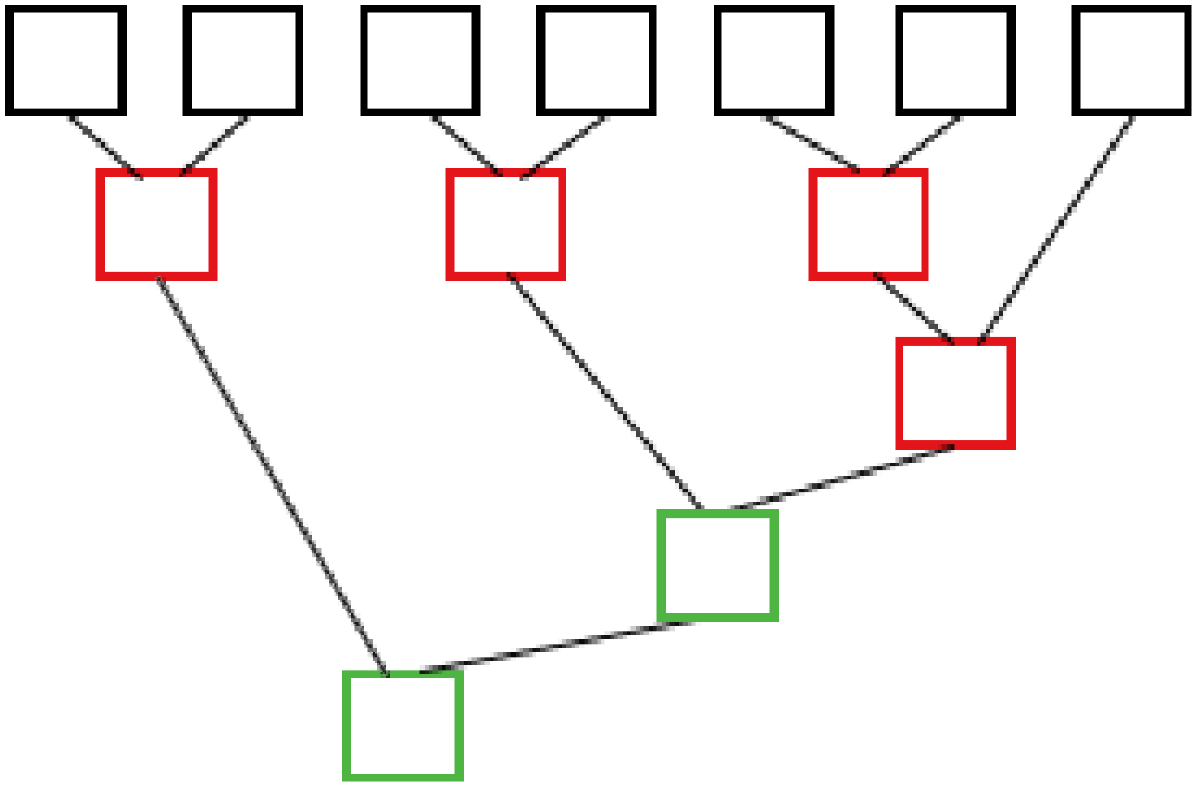
```
Level 0:  1 (2) -> *
Level 1:  2 -> 3 (4) -> *
Level 2:  *  4 -> 5 -> 6 (8) -> *
Level 3:  *  *  8 -> 7 -> 9 (10) -> *
Level 4:  *  *  10 (12) -> *
Level 5:  12 -> 11 -> *
Level 6:  *  *
```

The pairing heap generated here is correct, and we can tell because of the helpful (num) statements. 1 is the root with a left child 2, 2 has right sibling 3 which has left child 4, and so on. At this stage it’s not possible to make more complex trees because inserting does not use the combine siblings method. With that said, zig-zag trees are the best form of testing at this point.

Jay Offerdahl
EECS 560
Lab 13-14 Report Part 2

Verifying the Correctness of a Pairing Heap

To verify my deleteMin operation, I implemented several different test cases on several different data sets. For a general discussion, I'm confident in my solution because of the simplicity of the implementation. To begin, deleteMin simply checks if the root has a left child, if it doesn't, the tree becomes empty. Next, if the root does have a left child, the program sets root to the result of the combineSiblings method, which returns a pointer to the PairingNode that is the new tree. Inside this method, I used the two-pass merging scheme described in lab. This implementation combines children of the root in sets of two first, from right to left, then moves back left to right combining two of the resulting trees from the first merge. Please see below:

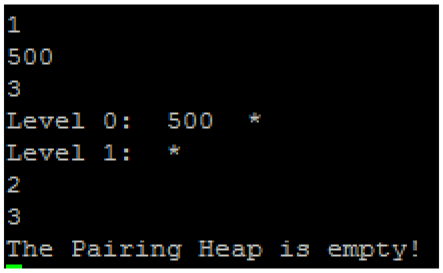


This shows that when I delete the minimum value from a pairing heap with a level two with 7 children, the 7 children will first be converted down to three subtrees. The first two are combined, the third and fourth are combined, and the fifth and sixth are combined. Since there are an odd number of trees, the program then combines the last combined subtrees with the odd sibling. After this step is complete, the program then begins the second pass, which moves right to left. This step combines the last two combined pairs, and then finally combines the result of that operation and the first combined pair. After this step is complete, we have successfully combined all of the siblings.

Jay Offerdahl
EECS 560
Lab 13-14 Report Part 2

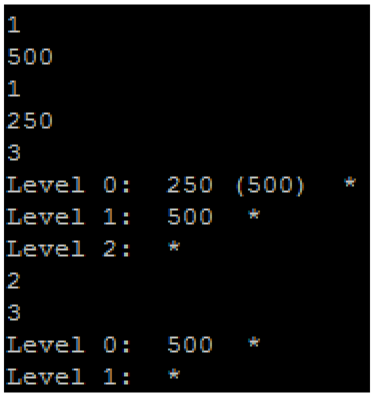
To show a few examples, I've laid out the following test cases:

1. Root doesn't have and children (1 node heap): This would simply yield an empty tree.



```
1
500
3
Level 0:  500  *
Level 1:  *
2
3
The Pairing Heap is empty!
```

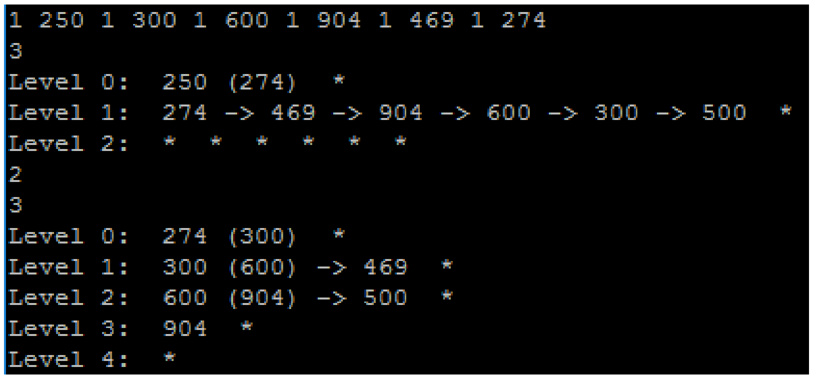
2. Root has only left child: This simply sets the left child as the new root.



```
1
500
1
250
3
Level 0:  250 (500)  *
Level 1:  500  *
Level 2:  *
2
3
Level 0:  500  *
Level 1:  *
```

3. Left child has right-sibling (even number of siblings total), and siblings don't have children.

In this example, we have subtrees 274 with left child 469, 600 with left child 904, and 300 with left child 500 after the first pass. Then, after the second pass, 300 merges with 600, so the intermediate tree is 300 with left child 600, 600 has right sibling 500 and left child 904. Next, the 300 tree merges with the 274 tree, since 274 is smaller, 274 becomes the root, 300 becomes the left child, and 469 becomes the right sibling of 300.

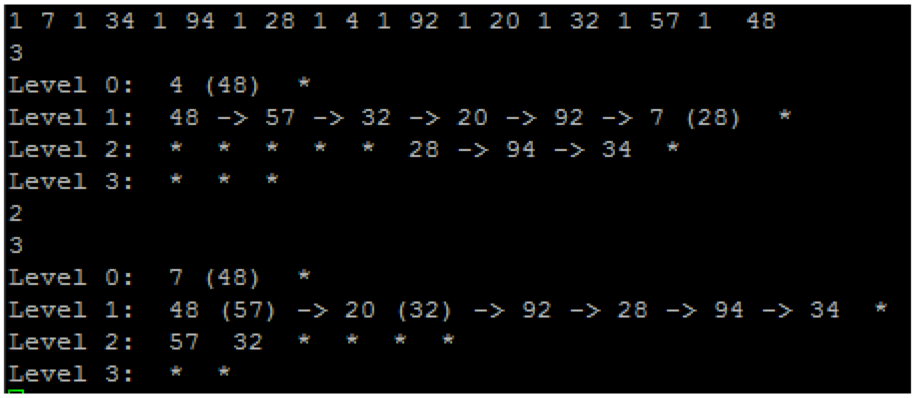


```
1 250 1 300 1 600 1 904 1 469 1 274
3
Level 0:  250 (274)  *
Level 1:  274 -> 469 -> 904 -> 600 -> 300 -> 500  *
Level 2:  *  *  *  *  *  *
2
3
Level 0:  274 (300)  *
Level 1:  300 (600) -> 469  *
Level 2:  600 (904) -> 500  *
Level 3:  904  *
Level 4:  *
```

Jay Offerdahl
EECS 560
Lab 13-14 Report Part 2

4. Left child has right-sibling (even number of siblings total), and siblings have children

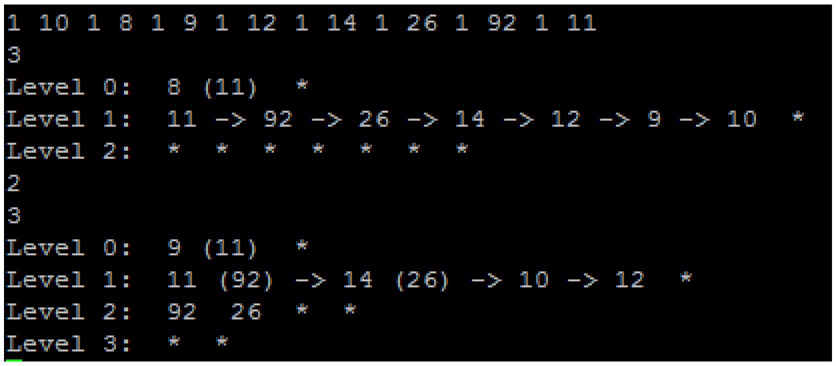
In this example, the first pass results in the subtrees 48 with left child 57, 20 with left child 32, and 7 with left child 92 whose right sibling starts with 28. Next, the right to left pass combines the 7 subtree with the 20 subtree, so 7 becomes the root, 20 becomes the left child, and so on. After this, the 7 subtree is merged with the 48 subtree, making 7 the root, 48 the left subtree, and moving 20 (the previous left child of 7) to the right sibling of 48, which preserves heap structure.



```
1 7 1 34 1 94 1 28 1 4 1 92 1 20 1 32 1 57 1  48
3
Level 0:  4 (48)   *
Level 1:  48 -> 57 -> 32 -> 20 -> 92 -> 7 (28)   *
Level 2:  *   *   *   *   *  28 -> 94 -> 34   *
Level 3:  *   *   *
2
3
Level 0:  7 (48)   *
Level 1:  48 (57) -> 20 (32) -> 92 -> 28 -> 94 -> 34   *
Level 2:  57  32   *   *   *   *
Level 3:  *   *
```

5. Left child has left-sibling (odd number of siblings total), and siblings don't have children

In this example, we have an odd number of siblings as children of the root. After the first pass, subtrees of 11 with child 92, 14 with child 26, and 9 with child 12 are created. Afterwards, 10 is merged with 9 to make a subtree of 9 with left child 10 whose right sibling is 12. Next, the right to left pass occurs, which merges the subtrees with roots 9 and 14, making a tree of 9 with left child 14, right child 10, and so on. Next, this result is merged with the 11 subtree, making the final tree with root of 9, left child of 11, right sibling of 14, and so on. It's interesting to notice that we have two siblings with left children, so there's no right sibling connection present on level 2.

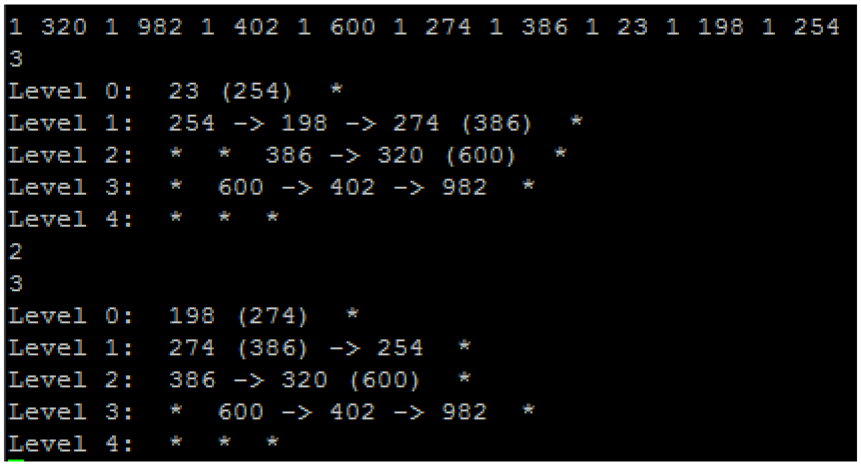


```
1 10 1 8 1 9 1 12 1 14 1 26 1 92 1 11
3
Level 0:  8 (11)   *
Level 1:  11 -> 92 -> 26 -> 14 -> 12 -> 9 -> 10   *
Level 2:  *   *   *   *   *   *   *
2
3
Level 0:  9 (11)   *
Level 1:  11 (92) -> 14 (26) -> 10 -> 12   *
Level 2:  92  26   *   *
Level 3:  *   *
```

Jay Offerdahl
EECS 560
Lab 13-14 Report Part 2

6. Left child has left-sibling (odd number of siblings total), and siblings have children

In this example, after the first pass, 198 will be a subtree with left child 254, and then this will be merged with the 274 subtree because there's an odd number of siblings. This results in a tree that has 198 at the root, 274 as the left child, and 254 as the right sibling of the left child. As you can see, this is the output.



```
1 320 1 982 1 402 1 600 1 274 1 386 1 23 1 198 1 254
3
Level 0:  23 (254)  *
Level 1:  254 -> 198 -> 274 (386)  *
Level 2:  *  *  386 -> 320 (600)  *
Level 3:  *  600 -> 402 -> 982  *
Level 4:  *  *  *
2
3
Level 0:  198 (274)  *
Level 1:  274 (386) -> 254  *
Level 2:  386 -> 320 (600)  *
Level 3:  *  600 -> 402 -> 982  *
Level 4:  *  *  *
```