CSC 505 Homework 2 program analysis

Yiwei Li
Zirun Han

First, we need to clarify the meanings of our indicators : n is the maximum size of the heap, b is the user-specified branching factor. And the bitShifting is the $\log_2(branchingfactor)$, which means when we set branching factor is 4, the bitShifting is 2; when we set branching factor is 64, the bitShifting is 6.

The running time of insertValue( ):

when we insert a new pair to our heap, we need to percolate-up it, compare it with its parent. If its key is smaller than its parent's key, we need to swap it with its parent. Because what we built is a minHeap.

In my function, I need to get the index of the inserted pair which locates at the last element of the ArrayList. The index equals to the current heap size minus one (a.k.a (n - 1)).

And then we need to find its parent, which takes constant time cause we just need to do the bitShifting operation in the expression of " parent =(insertValueIndex + branchingFactor - 1) >> bitShifting - 1".The consuming time for this part is $O(1)$.

Then we need to compare the keys of the inserted element and its parent. If the inserted key is smaller than its parent's key, we'll swap them and assign its parent's index to it, iterate swapping until its key no longer smaller than its parent's key. In the worst case, we will traverse the tree's height, then the time complexity is $O(\log_b n)$.

All in all, the running time of insertValue( ) is $O(\log_b n)$.

The running time of removeValue( )

when we do the remove operation, first record the root element, and then substitute the root element with the last element of the

heap, and minus the heap size by one. The then do the Min-Heapify( ) operation to fix the heap's structure.

The swap and minus operate is $O(1)$.

When we do the heapify, first we should traverse the child element of the current element to find the minimum child of it. Hence one node in the tree can have no larger than b : branching factor children, so we need to compare b times each level, and there are $\log_b n$ levels, so the cost of this procedure is $O(b \log_b n)$. The difference time cost between the two function is the removeValue( ) function needs to traverse all children of the node in one branch of the tree.

To compare the time with the common, regular binary tree.

As for the Insert operation, the cost of regular binary tree is $O(\log_2 n)$, which will always greater than the $O(\log_b n)$, since b is exponential times of 2.

As for the Remove operation, we need to traverse all children nodes of one ones to find the minimum one, but in regular binary tree, we only need to traverse two nodes. Meanwhile, the regular binary tree has higher height of the tree, so we need to compare $O(b \log_b n)$ with $O(2 \log_2 n)$, if b will be some value that very large, like 1024, and let assume n is 1024^2, the time that regular binary tree cost, is 20, but the cost of 1024 branching factor tree is 2048. So we can't say that the larger factor means the better performance.

The function $O(b \log_b n)$ will first decrease and the then increase. And when b = e, we can calculate the lowest bound. So as b grows the remove function cost will grow, the best b for our are 2 and 4. So , there is a trade off. As b grows, it means a worse performance of removing.

But in our code's performance, as for the input-4, it seems like 64 factor performs better than the binary heap. Cause the insert cost of

```
hanzirundeMacBook-Pro:src hanzirun$ java heap 2 < input-4.txt
branching factor is 2; bitShifting is 1
key comparisons: 17744135
hanzirundeMacBook-Pro:src hanzirun$ java heap 64 < input-4.txt
branching factor is 64; bitShifting is 6
key comparisons: 3697227
```

64 heap is smaller than the binary heap, and I think the reason is that the number of inserting far exceeds the number of removing in the input file.

```
hanzirundeMacBook-Pro:src hanzirun$ time java heap 2 < input-4.txt > /dev/null

real    0m2.246s
user    0m3.606s
sys     0m0.378s
hanzirundeMacBook-Pro:src hanzirun$ time java heap 64 < input-4.txt > /dev/null

real    0m2.217s
user    0m3.602s
sys     0m0.401s
hanzirundeMacBook-Pro:src hanzirun$
```