Stephen Chambers
CS858
February 17, 2016

Assignment 3 Writeup

1.
The program is working correctly. Plots are attached.

2.
The bad hash and add3 hash predicatively performed poorly. This is because they only use a small part of the key when computing the hash function, and thus have a greater chance of overlapping.

11.1-4:

Discussion:
We have a huge array that we don't want initialized. We will therefore take have a stack holding the actual values that are inserted into the array. The array will hold the index into the stack for the value of that element. Note: I am assuming that each key will only map to one value, and that the result of the hash function will not exceed the bounds of the array.

The following is the code for O(1) insertion, search, and delete. 'x' is the result of hashing an object X.

```
ary = (huge unitialized array)
stack = []
head = -1
```

INSERT:
```
head++
ary[x] = head
stack[head] = x
```

SEARCH:
```
return stack[ary[x]] == x
```

DELETE:
```
stack[x] = stack[head]
ary[stack[head]] = ary[x]
ary[x] = NULL
stack[head] = NULL
head--
```

11.3-1:
Before searching the list for a string x, compute the hash value of x. When iterating through the list, only perform the expensive string comparison if the hash values are the same.

11.3-1:

The procedure is as follows:

1. Generate a random number from {0..m}. Let that number be x. Let y be the number of elements in the bucket table[x].
2. Generate a random number from {1..L}. Let that number be z.
3. If z <= y, return the z^{th} element in that bucket.

The procedure is uniformly random because we are taking a random number from {1..L}.

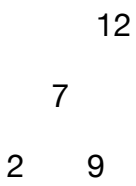
Proof of O(L* (1 + m/n))

The expected number of elements in a bucket is alpha, which equals n/m. Therefore, the probability of selecting a bucket that we can traverse is alpha/L. From the assignment description, if the chance of an event occurring at each trial is a/b, then the expected number of trials needed until the event occurs is b/a. Therefore, the expected number of trials needed until we can traverse a bucket is L/alpha. There is also a cost O(L), in the worst case, to actually retrieve the element. Therefore, the worst case time complexity for this procedure is O(L+L/alpha), which simplifies to O(L*(1+1/alpha)).

12.1-5:

Building a binary tree requires a comparison based sorting of the input. More formally, a binary tree can print out its elements in O(n) time. Therefore, any algorithm that create a binary tree from a list of n elements can be used to solve a sorting problem. If you could construct a binary tree in better than $\Omega(nlog(n))$ time, you could then solve a sorting problem in better than $\Omega(nlog(n))$ time, which is not possible.

12.2-4:



Set A is empty, Set B contains {12, 7, 2} and Set C contains {9}. The root, which is in set B, is larger than the only element in set C.

12.2-5

If a node has a right child, its successor is the *leftmost* child in its right subtree. Therefore, that node cannot have a left child.

Additionally, if a node has a left child, its predecessor is the *rightmost* child in its left

subtree. Therefore, that node cannot have a right child.

12.3-3

Worst case is $O(n^2)$, because you could have a long chain of insertions, as the tree is not balanced.

Best case is $\Omega(nlog(n))$, as discussed in question 12.1-5, when discussing the best case for a comparison based sorting algorithm.