CSCE 221 Cover Page

Programming Assignment#5

Due by April 14st midnight to eCampus

First name: Aldo Last Name: Leon Marquez UIN: 326004699

User Name: allema98 E-mail address: allema98@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more: Aggie Honor System Office

Types of Sources			
People	TA sec 501		
Web Pages	https://piazza.com	Stack Overflow	Geeks for Geeks
Printed material	-		
Other Sources	Lecture Slides		

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.
"On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work."

Name: Aldo Leon Marquez Date: 04/15/2019

Program Description:

The Objective of his assignment is to implement and understand the concept of a skip List. To do so will also use the "iterator" feature of C++. The Skip list was developed with a base class with a helper struct. The class support the following functions:

- Constructor
- Number of Coin Flips generator
- Insert
- Search
- Delete
- Clear
- Overloaded "in" operator
- Overloaded "out" operator

To compile the code is only necessary to call" g++ *.cpp" and to run it "./a.exe" from the "Code" folder, in the same folder the input file and driver used are provided.

Data Structure Description:

A skip List is a sorted list built with a rather unique feature, a coin flip dependent insertion. We will first generate a Random number of "tails" before a "head", after doing so we insert in ascending order in as many levels the coin flips determine. The result will be a list that looks like a city with different sized skyscrapers. The code was done with a base class for the List itself and a struct for a node that will the value and an iterator to the value below it (to connect the levels). I used a vector of list to implement the "levels" and the "values" of the skip list.

Implementation of the following functions:

a) Insert Cost: For the insert cost, I declared the type of the function to return an int and return a local sum for the number of comparisons for each individual insert. And by calling the "in" operator I sum all of he individual costs and then print divided by the number of times done.
b) Search Cost: Similar to the insert function I got individual values for each cost and then just divide by the number of values, but this one is calculated in the main.cpp file.
c) Delete Cost: An almost identical process to the Search was done for this value, expect of course the values were deleted this time.

Best case, worst case, and average case theoretical runtimes (Big-O) for the insert, search, and delete functions.

Runtime	Insert	Search	Delete
Worst Case	O(n)	O(n)	O(n)
Best Case	O(logn)	O(logn)	O(logn)
Average Case	O(logn)	O(logn)	O(logn)

The best case assuming the list does have values inside, i.e a bounded height of O(logn).

Answer the following questions:

(a) How likely is it that an item will be inserted into the nth level of the skip list?

R: It's the probability that for n tries we don't get tails, so $(1-p)^{n-1} * (p)$with a probability of 50/50 we have the resulting probability of $\left(\frac{1}{2}\right)^n$

(b) If you were to increase the probability of getting a "head" (positive result, keep flipping the "coin"), what would this do to the average runtime of insert, search, and delete?

R: The expected Value for the height would change accordingly to the new increased probability of getting "heads", The new geometric random variable will have a different expected value.

(c) How does the order of the data (sorted, reverse sorted, random) affect the number of comparisons?

R: It would make the number of comparisons to be either O(log(n)) for a sorted input or O(n) for an unsorted O(n), and for the random it would be bounded by O(log(n) as well.

(d) How does the runtime compare to a Binary Search Tree for the insert, search, and delete operations?

R:For a regular Binary tree (Not an auto balancing tree) it's the same runtime for all of the functions and for Both Best and Worst Cases.

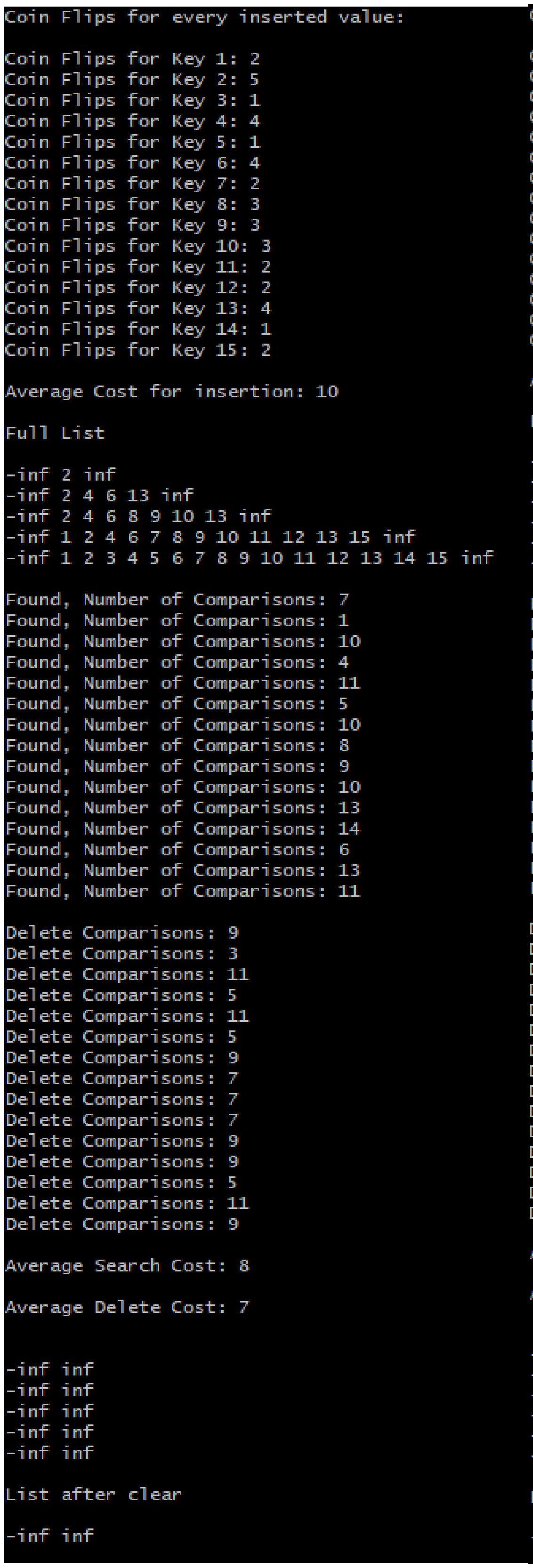
Comparing it to an auto balancing tree, a Skip List Falls short against the runtime for the worst cases (O(n) compared to O(log(n))).

(e) In what cases might a Binary Search Tree be more efficient than a skip list? In what cases might it be less efficient?

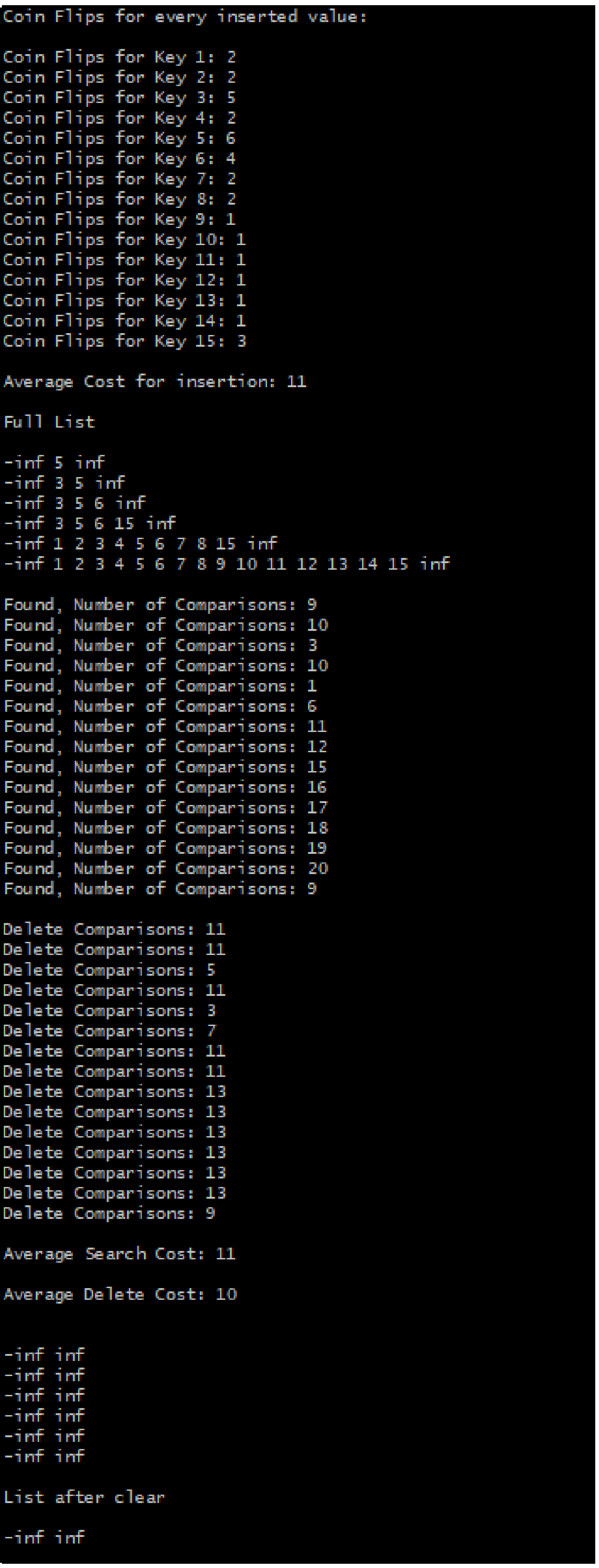
R: The Only case would depend on the Worst probable space complexity of the list O(nlog(n)), However the probability of this happening is low so the skip list almost equality efficient.

Since the Skip List depends on the number of "flips" and the number of flips has a low probability for a high number both approaches are "expectedly'" as efficient

Code Output Proof (same list with two different coin flips):



```
Coin Flips for every inserted value:

Coin Flips for Key 1: 2
Coin Flips for Key 2: 5
Coin Flips for Key 3: 1
Coin Flips for Key 4: 4
Coin Flips for Key 5: 1
Coin Flips for Key 6: 4
Coin Flips for Key 7: 2
Coin Flips for Key 8: 3
Coin Flips for Key 9: 3
Coin Flips for Key 10: 3
Coin Flips for Key 11: 2
Coin Flips for Key 12: 2
Coin Flips for Key 13: 4
Coin Flips for Key 14: 1
Coin Flips for Key 15: 2

Average Cost for insertion: 10

Full List

-inf 2 inf
-inf 2 4 6 13 inf
-inf 2 4 6 8 9 10 13 inf
-inf 1 2 4 6 7 8 9 10 11 12 13 15 inf
-inf 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 inf

Found, Number of Comparisons: 7
Found, Number of Comparisons: 1
Found, Number of Comparisons: 10
Found, Number of Comparisons: 4
Found, Number of Comparisons: 11
Found, Number of Comparisons: 5
Found, Number of Comparisons: 10
Found, Number of Comparisons: 8
Found, Number of Comparisons: 9
Found, Number of Comparisons: 10
Found, Number of Comparisons: 13
Found, Number of Comparisons: 14
Found, Number of Comparisons: 6
Found, Number of Comparisons: 13
Found, Number of Comparisons: 11

Delete Comparisons: 9
Delete Comparisons: 3
Delete Comparisons: 11
Delete Comparisons: 5
Delete Comparisons: 11
Delete Comparisons: 5
Delete Comparisons: 9
Delete Comparisons: 7
Delete Comparisons: 7
Delete Comparisons: 7
Delete Comparisons: 9
Delete Comparisons: 9
Delete Comparisons: 5
Delete Comparisons: 11
Delete Comparisons: 9

Average Search Cost: 8

Average Delete Cost: 7


-inf inf
-inf inf
-inf inf
-inf inf
-inf inf

List after clear

-inf inf
```



```
Coin Flips for every inserted value:

Coin Flips for Key 1: 2
Coin Flips for Key 2: 2
Coin Flips for Key 3: 5
Coin Flips for Key 4: 2
Coin Flips for Key 5: 6
Coin Flips for Key 6: 4
Coin Flips for Key 7: 2
Coin Flips for Key 8: 2
Coin Flips for Key 9: 1
Coin Flips for Key 10: 1
Coin Flips for Key 11: 1
Coin Flips for Key 12: 1
Coin Flips for Key 13: 1
Coin Flips for Key 14: 1
Coin Flips for Key 15: 3

Average Cost for insertion: 11

Full List

-inf 5 inf
-inf 3 5 inf
-inf 3 5 6 inf
-inf 3 5 6 15 inf
-inf 1 2 3 4 5 6 7 8 15 inf
-inf 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 inf

Found, Number of Comparisons: 9
Found, Number of Comparisons: 10
Found, Number of Comparisons: 3
Found, Number of Comparisons: 10
Found, Number of Comparisons: 1
Found, Number of Comparisons: 6
Found, Number of Comparisons: 11
Found, Number of Comparisons: 12
Found, Number of Comparisons: 15
Found, Number of Comparisons: 16
Found, Number of Comparisons: 17
Found, Number of Comparisons: 18
Found, Number of Comparisons: 19
Found, Number of Comparisons: 20
Found, Number of Comparisons: 9

Delete Comparisons: 11
Delete Comparisons: 11
Delete Comparisons: 5
Delete Comparisons: 11
Delete Comparisons: 3
Delete Comparisons: 7
Delete Comparisons: 11
Delete Comparisons: 11
Delete Comparisons: 13
Delete Comparisons: 13
Delete Comparisons: 13
Delete Comparisons: 13
Delete Comparisons: 13
Delete Comparisons: 13
Delete Comparisons: 9

Average Search Cost: 11

Average Delete Cost: 10


-inf inf
-inf inf
-inf inf
-inf inf
-inf inf
-inf inf

List after clear

-inf inf
```

Both Found and Delete are set up to go from 1 to 15 (the values inserted to the lost) in order.