(CSCI 251) Activity Two and Exam Two Review

Student Name **Matthew Boyea**

***Instruction:*** *The students are encouraged to type the answer use WORD and submit the word file through blackboard. To learn how to type math notation in word, please watch video* [*https://www.youtube.com/watch?v=SRGaW3maK38*](https://www.youtube.com/watch?v=SRGaW3maK38)*. You may search other videos to learn how to do this faster.*

*However, if the student feels it takes too much time to type the answer, then the student can use handwriting to write down the answers on paper. The students then scan the paper into pdf or image file then upload the file to blackboard. In this case, any unclear handwriting may result 0 points to the problem.*

*Exam Two consists of two parts. Part One is online multiple choices/fill in blanks. The students should review the embedded homework problems to prepare Part One. This activity is Practice Problem Set for Exam Two Part Two review. The questions in real Exam Two may not be the same questions listed here. However, the testing concepts will be the same or similar.*

Problem One: (25 points. 4 points each for first five questions. 5 points for question 6) You textbook listed common functionality that a list should have (refer to Table 2.1.1). However, your book only implemented algorithm for insertAfter and remove. In this problem, you are asking to implement the rest common functionality in table 2.1.2.

1. append: which append the given node after the tail of the list



1. prepend: which insert the given node to the front of the head of the list



1. printList: which print all node data in order as their node in the list



1. sort: Sort the list items in ascend order



1. isEmpty: Return true if the list is empty; false otherwise



1. For printListReversely functionality, if the list is a single linked list, we cannot step through the list reversely. How would you implement this printListReversely algorithm?



Problem Two: (15 points, 5 points each) In problem one, we didn’t implement getLength function. This function has some challenge.

1. For getLength function, one implement is that we can step through the list, from head to tail, and count. Write algorithm for this implementation



1. In previous implementation, what is the run time?

The run time would be O(n) where n is the number of nodes.

1. If I want constant run time for getLength function, what change needs to be made for List data structure? How this change will affect the implementation of other functions such as insertAfter and remove?

To achieve a constant run time for the GetLength() function, we could keep a variable that is initialized to 0 when the data structure is initialized. Then we would increment and decrement this variable any time a node is added or removed respectively. ie. in InsertAfter we would increment the variable and in Remove we would decrement the variable.

Problem Three: (10 points. 5 points each)

1. Write a insertAfter algorithm for double linked list



1. Write a remove algorithm for double linked list



Problem Four: (10 points. 5 points each) In your textbook, the search algorithm returns the node reference that refers to the first node whose data match the search target. Write search algorithm to return the node reference that refers to the last node whose data match the search target. Assume your algorithm input is

1. A single linked list and target data



1. A double linked list and target data



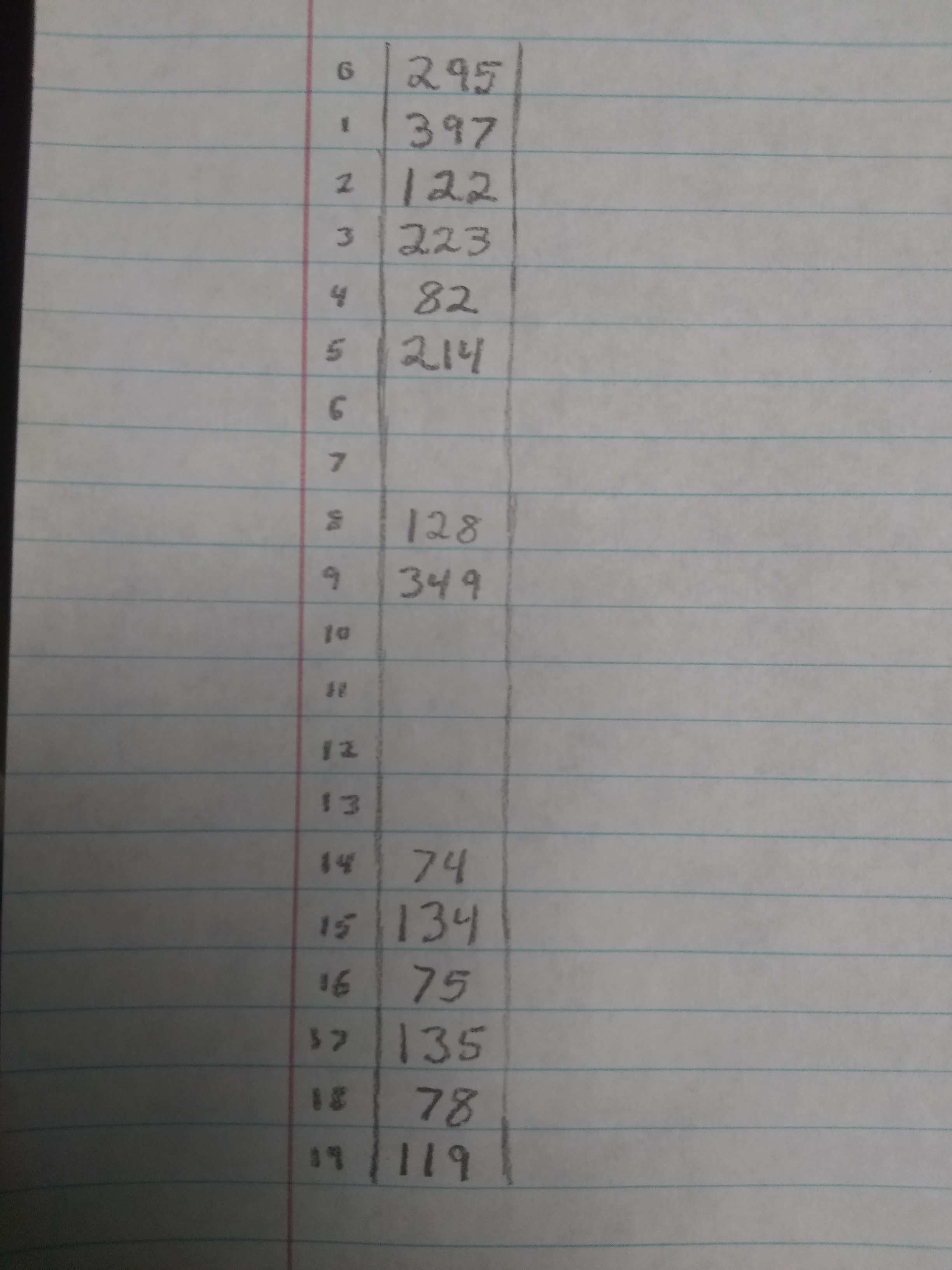
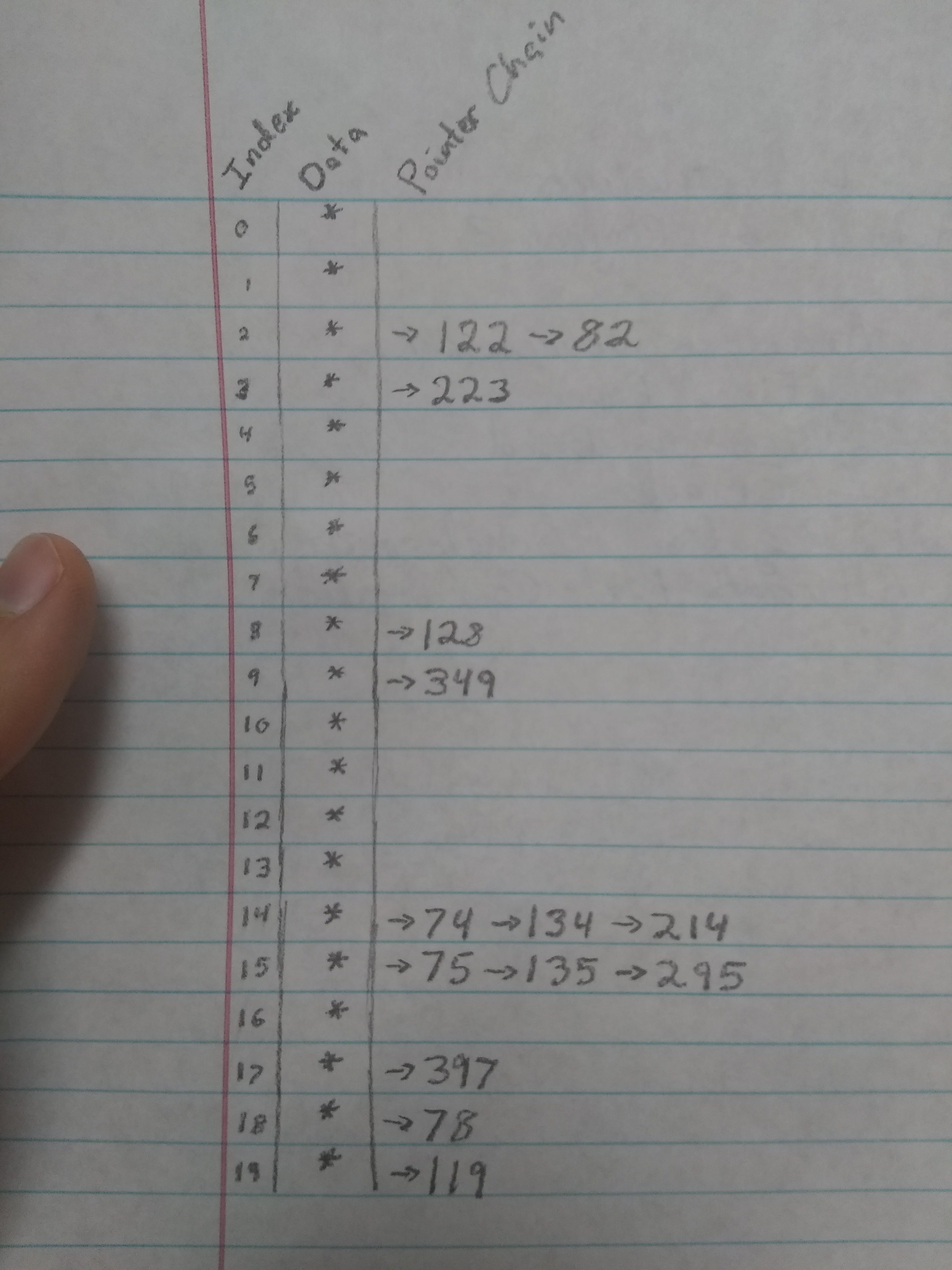
Problem Four: (20 points. 10 points each)

1. Implement Queue ADT using two stacks

.

1. Implement Stack ADT using two queues



Problem Five: (10 points, 5 points each) Assume that we use %20 to hold the following data set (by the order that is inserted to the table): 122, 74, 134, 78, 75, 128, 349, 119, 223, 135, 295, 397, 82, 214.

1. Draw the hash table if we use chain to handle collision

*Depicted in left figure:*

1. Draw the hash table if we use linear probing to handle collision

*Depicted in right figure:*

Problem Six: (10 points, 5 points each) Using same data set of Problem Five. Using mid-square hash, number of bucket is 100, key is 243, and R = 2

*The index of key 243 would equal 90. The following tables depict 122, 74, 134, 78, 75, 128, 349, 119, 223, 135, 295, 397, 82, 214 under this hash function.*

*Note that there was no instance of collision, so the two arrays’ data are functionally identical, aside from the extra pointers.*

1. Draw the hash table if we use chain to handle collision

|  |  |
| --- | --- |
| Index | Data |
| 0 | \* -> nullptr |
| 1 | \* -> nullptr |
| 2 | \* -> nullptr |
| 3 | \* -> nullptr |
| 4 | \* -> nullptr |
| 5 | \* -> nullptr |
| 6 | \* -> nullptr |
| 7 | \* -> nullptr |
| 8 | \* -> 78 -> nullptr |
| 9 | \* -> nullptr |
| 10 | \* -> nullptr |
| 11 | \* -> nullptr |
| 12 | \* -> nullptr |
| 13 | \* -> nullptr |
| 14 | \* -> nullptr |
| 15 | \* -> nullptr |
| 16 | \* -> nullptr |
| 17 | \* -> nullptr |
| 18 | \* -> 349 -> nullptr |
| 19 | \* -> nullptr |
| 20 | \* -> nullptr |
| 21 | \* -> nullptr |
| 22 | \* -> nullptr |
| 23 | \* -> nullptr |
| 24 | \* -> nullptr |
| 25 | \* -> nullptr |
| 26 | \* -> nullptr |
| 27 | \* -> nullptr |
| 28 | \* -> nullptr |
| 29 | \* -> nullptr |
| 30 | \* -> nullptr |
| 31 | \* -> nullptr |
| 32 | \* -> nullptr |
| 33 | \* -> nullptr |
| 34 | \* -> nullptr |
| 35 | \* -> nullptr |
| 36 | \* -> nullptr |
| 37 | \* -> nullptr |
| 38 | \* -> nullptr |
| 39 | \* -> nullptr |
| 40 | \* -> nullptr |
| 41 | \* -> 119-> nullptr |
| 42 | \* -> nullptr |
| 43 | \* -> nullptr |
| 44 | \* -> nullptr |
| 45 | \* -> nullptr |
| 46 | \* -> nullptr |
| 47 | \* -> 74 -> nullptr |
| 48 | \* -> 122 -> nullptr |
| 49 | \* -> nullptr |
| 50 | \* -> nullptr |
| 51 | \* -> nullptr |
| 52 | \* -> nullptr |
| 53 | \* -> nullptr |
| 54 | \* -> nullptr |
| 55 | \* -> nullptr |
| 56 | \* -> nullptr |
| 57 | \* -> 214 -> nullptr |
| 58 | \* -> nullptr |
| 59 | \* -> nullptr |
| 60 | \* -> nullptr |
| 61 | \* -> nullptr |
| 62 | \* -> 75 -> nullptr |
| 63 | \* -> 128 -> nullptr |
| 64 | \* -> nullptr |
| 65 | \* -> nullptr |
| 66 | \* -> nullptr |
| 67 | \* -> nullptr |
| 68 | \* -> nullptr |
| 69 | \* -> nullptr |
| 70 | \* -> 295 -> nullptr |
| 71 | \* -> nullptr |
| 72 | \* -> 82 -> nullptr |
| 73 | \* -> nullptr |
| 74 | \* -> nullptr |
| 75 | \* -> nullptr |
| 76 | \* -> 397 -> nullptr |
| 77 | \* -> nullptr |
| 78 | \* -> nullptr |
| 79 | \* -> 134 -> nullptr |
| 80 | \* -> nullptr |
| 81 | \* -> nullptr |
| 82 | \* -> 135 -> nullptr |
| 83 | \* -> nullptr |
| 84 | \* -> nullptr |
| 85 | \* -> nullptr |
| 86 | \* -> nullptr |
| 87 | \* -> nullptr |
| 88 | \* -> nullptr |
| 89 | \* -> nullptr |
| 90 | \* -> nullptr |
| 91 | \* -> nullptr |
| 92 | \* -> nullptr |
| 93 | \* -> nullptr |
| 94 | \* -> nullptr |
| 95 | \* -> nullptr |
| 96 | \* -> nullptr |
| 97 | \* -> 223 -> nullptr |
| 98 | \* -> nullptr |
| 99 | \* -> nullptr |

1. Draw the hash table if we use linear probing to handle collision

|  |  |
| --- | --- |
| Index | Data |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 | 78 |
| 9 |  |
| 10 |  |
| 11 |  |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |
| 16 |  |
| 17 |  |
| 18 | 349 |
| 19 |  |
| 20 |  |
| 21 |  |
| 22 |  |
| 23 |  |
| 24 |  |
| 25 |  |
| 26 |  |
| 27 |  |
| 28 |  |
| 29 |  |
| 30 |  |
| 31 |  |
| 32 |  |
| 33 |  |
| 34 |  |
| 35 |  |
| 36 |  |
| 37 |  |
| 38 |  |
| 39 |  |
| 40 |  |
| 41 | 119 |
| 42 |  |
| 43 |  |
| 44 |  |
| 45 |  |
| 46 |  |
| 47 | 74 |
| 48 | 122 |
| 49 |  |
| 50 |  |
| 51 |  |
| 52 |  |
| 53 |  |
| 54 |  |
| 55 |  |
| 56 |  |
| 57 | 214 |
| 58 |  |
| 59 |  |
| 60 |  |
| 61 |  |
| 62 | 75 |
| 63 | 128 |
| 64 |  |
| 65 |  |
| 66 |  |
| 67 |  |
| 68 |  |
| 69 |  |
| 70 | 295 |
| 71 |  |
| 72 | 82 |
| 73 |  |
| 74 |  |
| 75 |  |
| 76 | 397 |
| 77 |  |
| 78 |  |
| 79 | 134 |
| 80 |  |
| 81 |  |
| 82 | 135 |
| 83 |  |
| 84 |  |
| 85 |  |
| 86 |  |
| 87 |  |
| 88 |  |
| 89 |  |
| 90 |  |
| 91 |  |
| 92 |  |
| 93 |  |
| 94 |  |
| 95 |  |
| 96 |  |
| 97 | 223 |
| 98 |  |
| 99 |  |