'l Name: Alexander Miller CNetID: 12177451

CS162: Honors Introduction to Computer Science II (Winter '19)

Exam I

February 18, 2019

Instructions

- This exam is closed-book, with one double-sided handwritten sheet of notes permitted.
- You have 50 minutes to complete the exam.
- There are 4 problems, plus 1 extra credit problem, on 13 pages.
- Please read each problem carefully and write down your answer in the space provided under the question.
- Good luck!

Full Name: _____ **CNetID:** _____

	Question	Points	
Prob 1	Data Structures	5	3.5
Prob 2	Arrays in C	10	9
Prob 3	AVL Trees	20	15
Prob 4	Hashing	10	10
Extra Credit	More on Linked List	6	3
	Total:	45 + 6	

40.5

Full Name: Alex Miller CNetID: 12177451

1. **Data Structures**

3.5 (a) (5 points) For each of the following data structures each of which has N unique integers, give the *worst case* ***exact*** count number of comparisons required to find a given integer (assuming it exists in the data structure).

- (1pt) Unordered Array

-0.5 N comparisons

- (1pt) Ordered Array

Log(N) comparisons

- (1pt) Chained Hash Table of capacity $N/4$ (assume good, evenly distributed hash function, and N is divisible by 4)

-0.5 4 comparisons

- (1pt) Linked List

-0.5 N comparisons

- (1pt) Binary Search Tree (assume complete tree)

Log N comparisons

0, 1, 2, 3 $\xrightarrow{2}$ 23, 0, 1

5, 6, 7, 8, 9 $\xrightarrow{3}$ 7, 8, 9, 56

9 8 7 6 5

7 8 9 6 5

7 8 9 5 6

Full Name: Alex Miller CNetID: 12177451

2. **Arrays in C**

Your task in this question is to implement the shift operation on array of integers, using only reverse operations. In particular, assume you have access to a function called:

```
void reverse(int* source, int *target, int n);
```

which copies the first n integers in `source` array into `target` array in reverse order. For example, if you have `A = {0,1,2,3,4}` and $n = 3$, then:

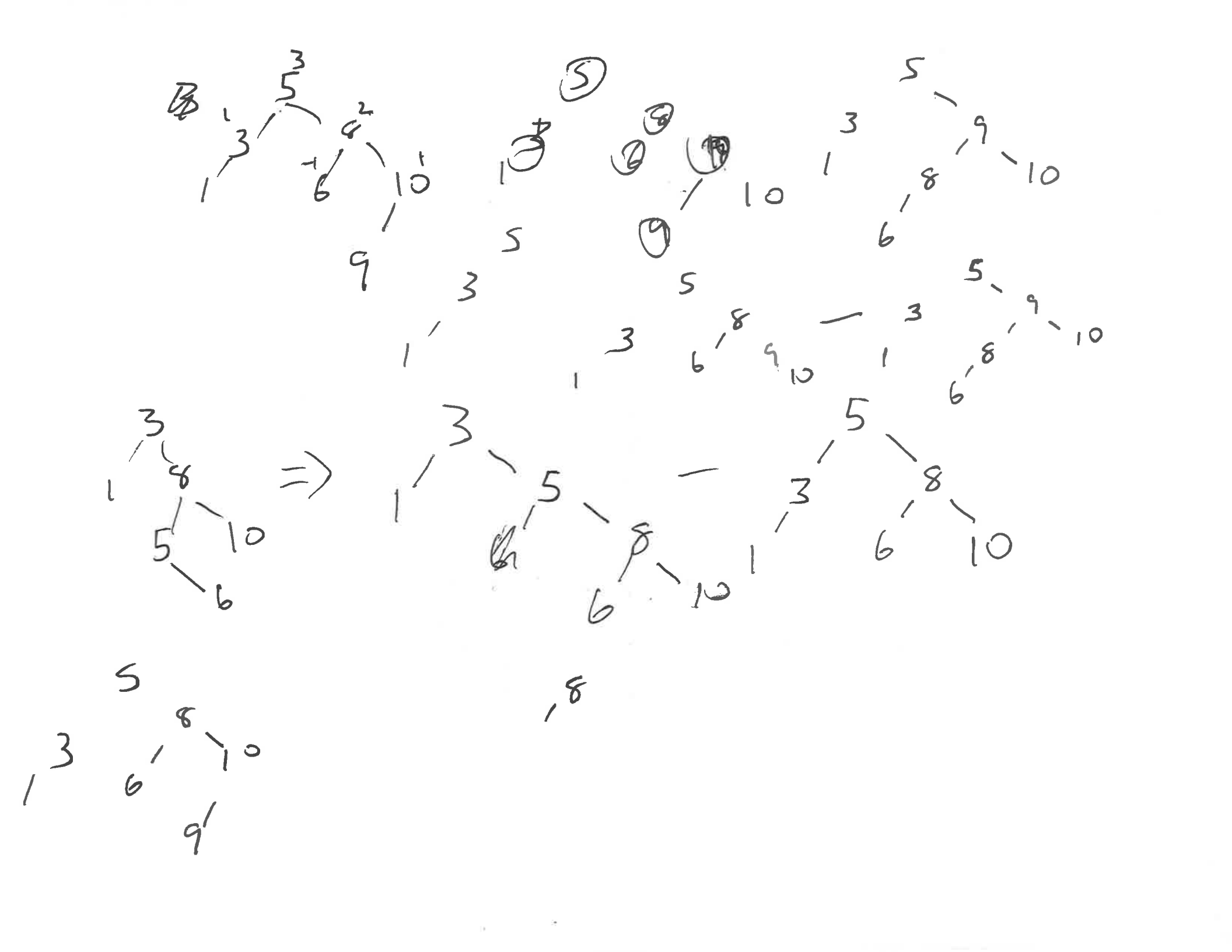
- `reverse(A, target, n)` should result in `target = {2,1,0}`.
- `reverse(&A[1], target, n)` should result in `target = {3,2,1}`.

(a) (10 points) Give an algorithm that outputs a new array of n integers. The contents of the new array should be the contents of input array A (also, of n integers) by d places to the right (with wrap-around). For example, if `A = {0,1,2,3,4,5,6,7}` and `d = 3`, then `shift_right(A, d, 8)` should allocate and return a new array of integers `B = {5,6,7,0,1,2,3,4}`. Note that in order to use `reverse` function you will need to allocate the target array properly. Solutions that contain any loops or recursions will not receive any points. Hint: you can accomplish this by three calls to `reverse`.

9

```
int *shift_right(int* A, int d, int n) {
    if (A == NULL) return NULL;
    int *target = malloc(n* sizeof(int));
    reverse(A, target, n);
    reverse(target, target, d);
    reverse(&target[d], &target[d], n-d);
    return target;
}
```

check if NULL
-1



Full Name: Alex Miller **CNetID:** 12177451

3. **AVL Trees**

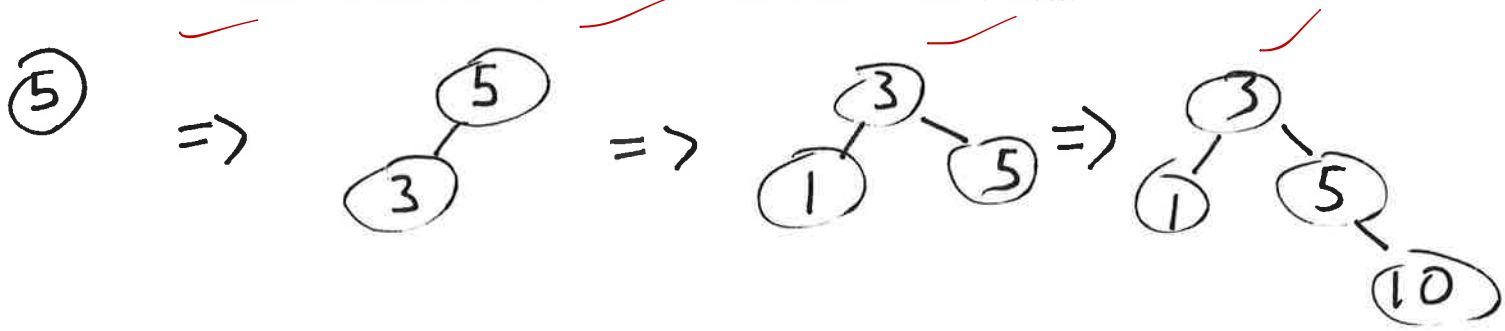
AVL trees are binary search trees that are balanced, so that the height of the tree is guaranteed to be $O(\log n)$. The *balance invariant* will be maintained, such that for every node in the tree, the height of the left and right subtrees can differ by at most 1, where the height of a tree is defined as the maximum number of nodes from the root to a leaf.

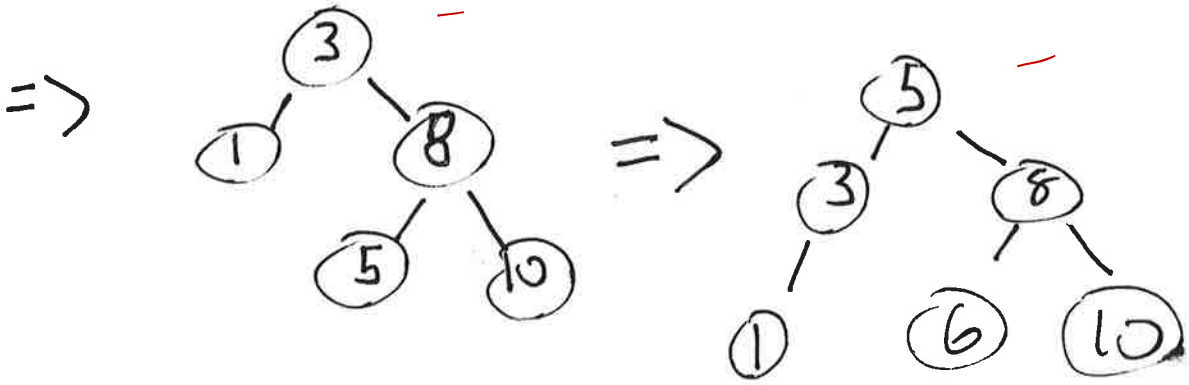
5

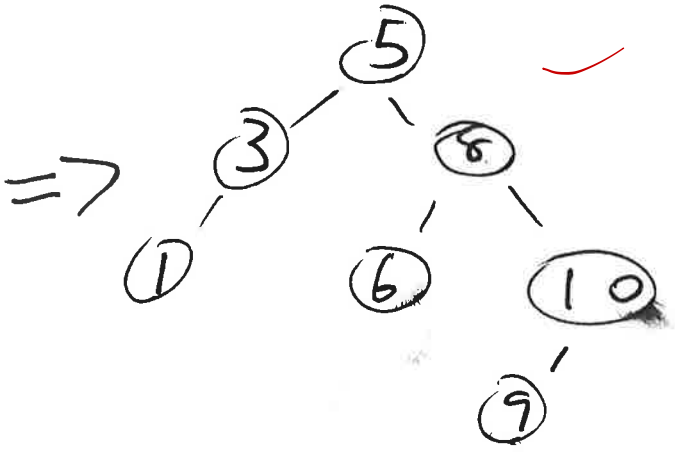
(a) (5 points) Draw the AVL tree that results from inserting the following keys in the order given:

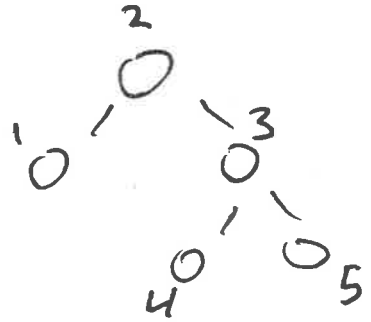
5 3 1 10 8 6 9

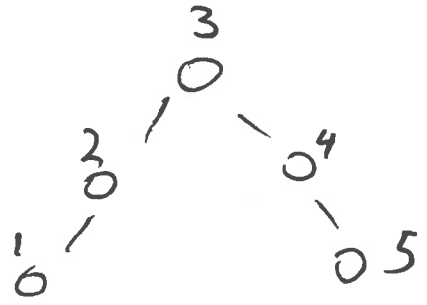
Show the tree after each insertion. You need to draw 7 trees.

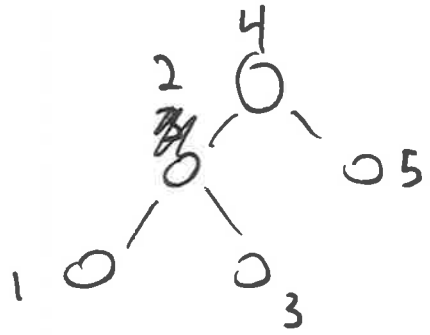


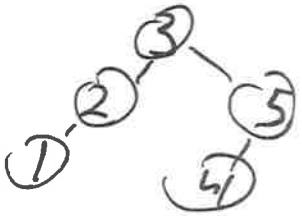





2
1
3
4
5


3
2
4
1
5


4
2
5
1
3


3
2
5
1
4

Full Name: Alex Miller **CNetID:** 12177451

(b) (5 points) Which nodes could be roots in an AVL tree if constructed using the following five keys:

1 2 3 4 5

5

inserted in any arbitrary order? Write down all the possible roots.

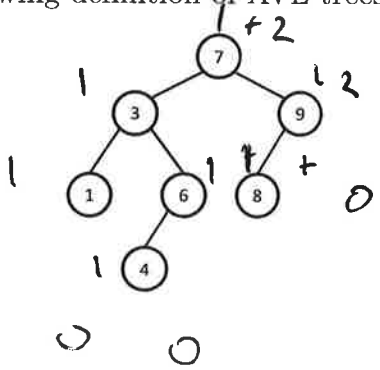
1 Only 2, 3, & 4 can act as THE Root noodes in an AVL tree containing this data

Full Name: Alex Miller CNetID: 1217451

(c) (5 points) The size of an AVL tree is the number of nodes in the tree, including the root. For example, the AVL tree below, rooted at the node 7, has size 7. Notice the size of a tree can be defined recursively if the size of the left and right subtrees are known.

Suppose we are working with the following definition of AVL trees.

```
1 typedef struct node {
2     int data;
3     struct node *parent;
4     struct node *left;
5     struct node *right;
6     int height;
7 } tree;
```

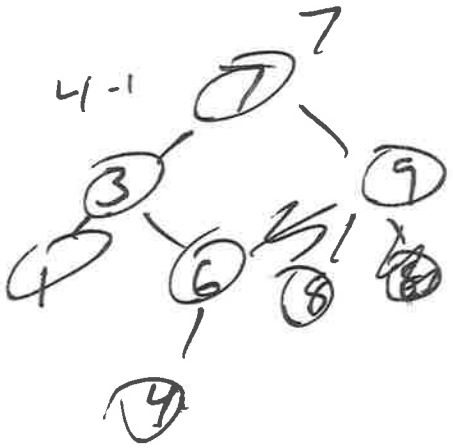


5

Assume empty subtrees are represented with NULL pointers. Complete the following program which finds the size of the tree given. Your solution must be recursive.

```
int size_of_tree(tree *T) {
  if (T == NULL) return 0;
  return 1 + size_of_tree(T->left) + size_of_tree(T->right);
}
```

}



size of Left children

Size of parent - 1 - size right child

Size of Parent->Left

```
if (T->data == ((T->Parent)->right)->data)
    Size of parent -1 - size Right
if (T= left child
```

Full Name: Alex Miller CNetID: 12177451

(d) (5 points) One way to find the median in an AVL tree is to find a node which has the same number of elements smaller than it as larger than it (verify in the example above 6 is the median; three elements are smaller and three are larger). Here, you will implement one piece of this – finding the number of elements in the tree *smaller* than a given node. Suppose the definition of the AVL tree above is modified to include the size of the tree and the parent node, i.e. we now have an additional field `int size;`. Assume every element in the tree is unique.

0

Complete the following function. Hint: No comparison of elements is needed.

```
int num_smaller(tree *T) {
    return (T->parent)->size) - 1 - ((T->right)->size);
    int comp = T->data;
    tree *ptr = T
    int count = 0
    while (ptr != NULL){
        ptr = ptr->parent;
    }
    int Left =
}
```

if you return here nothing else will happen

Full Name: Alex Miller CNetID: 12177151

4. **Hashing**

Recall from lecture and lab, we implement hash table using an array of size m. And we use hash function, $hash(x)$, to index into the hash table. We define two elements to have *collision* if their keys are different ($x_1 \neq x_2$), yet hashed to the same cell in the hash table, i.e. $hash(x_1) = hash(x_2)$.

(a) (3 points) For a hash table of capacity m with n keys to be inserted, if $n = 4m^2$ and the keys are *not* evenly distributed and *chaining* is used to resolve collisions. What is the number of comparisons required in searching for a specific key in the *worst case*? Please use the *exact count* rather than an asymptotic bound. Explain why.

3

$4m^2$ comparisons. In the worst case, all of the keys collide at one hash table index, & the desired key is at the end of the linked list at that index. Therefore we need to check every single key in the hash table.

(b) (3 points) In linear probing, we attempt to insert a key x into successive cells with index: $h_0(x), h_1(x), h_2(x), \ldots$ until an empty cell is found, according to the function

$$h_i(x) = hash(x) + i \mod m$$

Using $hash(x) = x$ as your hash function and $m = 11$ as your table capacity, draw the resulting hash table after inserting the keys: 15, 23, 49, 26, 66, 60, in order.

4 1 5 4 4 5

3

0	1	2	3	4	5	6	7	8	9	10
66	23			15	49	26	60			

1 4 3 4 1 0

1 4 3 4 1

— 1 2 3 4 5

7 7 7 7

7 7 7

Full Name: Alex Miller **CNetID:** 12177459

(c) (4 points) In quadratic probing, we attempt to insert a key x into the cells with index: $h_0(x), h_1(x), h_2(x), \ldots$ until an empty cell is found, according to the function

$$h_i(x) = hash(x) + i^2 \mod m$$

Quadratic probing suffers from one problem that linear probing does not – given a non-full hashtable, insertions with linear probing will always succeed, while insertions with quadratic probing may or may not succeed (i.e. they may never find an open spot to insert).

Using $hash(x) = x$ as your hash function and $m = 6$ as your table capacity, give an example of a non-full hashtable and a key that cannot be successfully inserted using quadratic probing.

4

0	1	2	3	4	5
	1	2	3	4	5

Key to insert: 7

1 2 3 4 5 6

1
4
9
16
25
36
49
64
81
100
121
142

Full Name: Alex Miller CNetID: 12172151

5. **More on Linked List** (Extra Credit)

In this extra credit problem, we will revisit the linked list data structure, and discover how we can make it more robust for computation.

Assume we have the integer linked list data structure defined as follows:

```
1  struct node {
2      int data;
3      struct node *next;
4  };
5  typedef struct node list;
6  struct header{
7      list *head; // start of linked list
8      list *tail; // end of linked list
9  };
10 typedef struct header linked_list;
```

(a) (2 points) Given the definition of linked list, we say it is cyclic if following the next pointer in the list could lead to nodes that you have already visited. Given a linked list with N nodes, how many cycles can it have? Explain why.

2

~~It can have infinite cycles. That is because you can keep going around the~~

It can only have one cyclic path. At some point (the end) the last node loops back to only one previous node. To have more than one cycle would require the last node to link to multiple nodes, which the struct does not allow for.

Full Name: Alex miller CNetID: 12177451

(b) (4 points) One inefficient way to detect cycles is to step through the linked list and keep track of all the nodes you've visited. If you visit a node you've already visited you know you must have encountered a cycle.

Complete the following function which *efficiently* determines if there is a cycle in the given linked list. Your solution should not use any sort of external data structure (e.g. you can't alloc an array), and has $O(n)$ running time. Hint: use two pointers.

```
bool contains_cycle(linked_list *L) {
  node * head = L->head
  node *tail = L->tail
  while(head != tail) {
    if (head tail->next==head) return true;
    head = head -> next;
  }
  return false (tail->next == tail);
}
```

Assumes valid linked list

Full Name: _____ CNetID: _____

Scrap Paper