5.1 Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \bmod 10$, show the resulting:

a. Separate chaining hash table.
b. Hash table using linear probing.
c. Hash table using quadratic probing.

A.

1: 4371
3: 1323 → 6173
4: 4344
9: 4199 → 9679 → 1989

B.

0: 9679
1: 4371
2: 1989
3: 1323
4: 6173
5: 4344
⋮
9: 4199

C:

0: 9679
1: 4371
. . . .
3: 1323
4: 6173
5: 4344
. . .
8: 1989
9: 4199

5.1 Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function $h(x) = x \bmod 10$, show the resulting:

a. Separate chaining hash table.
b. Hash table using linear probing.
c. Hash table using quadratic probing.
d. Hash table with second hash function $h_2(x) = 7 - (x \bmod 7)$.

5.2 Show the result of rehashing the hash tables in Exercise 5.1. table size = 23.

$h(x) = x \bmod 23$

A.

1: 4371
- - -
9: 6173
- - -
11: 1989
- - -
12: 1323
13: 4199
- - -
19: 9679
20: 4344

B.

1: 4371
- - -
9: 6173
- - -
11: 1989
- - -
12: 1323
13: 4199
- - -
19: 9679
20: 4344

C.

1: 4371
- - -
9: 6173
- - -
11: 1989
- - -
12: 1323
13: 4199
- - -
19: 9679
20: 4344

the results of A. B. C are the same. because there is no collision.

6.2 a. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

b. Show the result of using the linear-time algorithm to build a binary heap using the same input.

A.

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

B. Initial build: 10, 12, 1, 14, 6, 5. 8, 15, 3, 9. 7, 4. 11. 13, 2

Heap Bottom level: 10, 12, 1, 3, 6, 4, 2, 15. 14. 9. 7. 5, 11. 13. 8

Heap next level up: 10, 3, 1, 12, 6. 4. 2, 15, 14. 9. 7. 5. 11. 13. 8

Final heap: 1, 3, 2, 12, 6, 4. 8, 15. 14. 9. 7. 5. 11. 13. 10

Data Structure - Problem Set #4

Xiaoli Sun (xs2338) - xs2338@columbia.edu

April 26, 2020

Problem 6.16

```
/**
     * find tree node that is at implicit position i
     */
    Node<AnyType> TreeNodeAtI(Node<AnyType> root, int i, int index) {
        /**
         * if i equals to the index of the current node in the array representation,
         * the current node is the required node and thus return the current node.
         * Also, if the current node is empty, return null.
         */
        if(i==index || root==null) {
            if(root!=null) {
                return root;
            }
        }
        /**
         * if current node has right child.
         */
        if(root.right != null) {
            //call recursively TreeNodeAtI
            //where 2*index+2 is the index of the right child of current node
            Node<AnyType> right = TreeNodeAtI(root.right, i, 2*index+2);
            if(right != null) {
                return right;
            }
        }
        /**
         * if current node has left child.
         */
        if(root.left != null) {
            Node<AnyType> left = TreeNodeAtI(root.left, i, 2*index+1);
            if(left!=null) {
                return left;
            }
        }
        return null;
    }
```

The above algorithm takes the root node of Binary Heap tree, implicit position i of tree node to be found and the index of current node as input. It first call to TreeNodeAtI, index=1, since the root node will always be at index 1.
The algorithm compares whether the implicit position i and index of current node are equal or not.

Problem 6.18

a

The root is the minimum element.
The maximum element can be find out by comparing the two child of the two root.
The bigger element between the two child nodes of the root is the max element.

b

Insertion in minmax heap
To add an element to a Min-Max Heap perform following operations:
Insert the required key into given Min-Max Heap.
Compare this key with its parent. If it is found to be smaller (greater) compared to its parent, then it is surely smaller (greater) than all other keys present at nodes at max(min) level that are on the path from the present position of key to the root of heap. Now, just check for nodes on Min(Max) levels.
If the key added is in correct order then stop otherwise swap that key with its parent.
The following algorithm is used to insert a new node in to the min-max heap:

```
Step 1:
Create a function to insert the new node in the min-max heap.

Function min_max_insert(element heap[], int n, element item)

   /* insert item into the min-max heap */
   int parent;
   n++;

Step 2:
Create an if statement to check the max size of the node

   if(n == MAX_SIZE)
   {
      print("The heap is full");
      return;
   }

   parent = n / 2;
   if(!parent)
   /* heap is empty, insert item into first position */
      heap[1] = item;
   Else switch(level(parent))

Step 3:
Create an if statement to find the min element in the node.
      case FALSE:
      /* min level */
      If item.key < heap[parent].key
            heap[n] = heap[parent]
            verify_min(heap, parent, item)

         Else
            verify_max(heap, n, item)
         break
      case TRUE:
         /* max level */
         If item.key > heap[parent].key
            heap[n] = heap[parent]
            verify_max(heap, parent, item)

         Else
            verify_min(heap, n, item)
```