PP02: Classical Data Structures

CST501: Advanced Algorithms Programming Project Write-Up

Posting ID: 5582-527 .

Pledge

I Manohara Rao Penumala pledge, on my honor, that the work submit is my own and that I have neither sought nor provided inappropriate help to any other. I understand that I may not share any part of my solution or design (even if I have a bug and want help!) with any person other than the grader, or instructor.

I Manohara Rao Penumala further understand that I must submit this completed write-up with my name inserted into the pledge, and also a .zip file containing my source-code as presented here as two separate attachments to the course website.

Highlighted Source Code:

```
1.  """
2.  This file provides implementation of Singly Linked List, Chained Hash Dictionary,
3.  Open Addressed Hash Dictionary and Binary Search Tree.
4.  All these functions allow only unique inputs as the nodes or keys.
5.  Non unique nodes or keys are not added to the information sets represented by these
    classes.
6.  """
7.  #from decimal import Decimal
8.  from decimal import *
9.  import random
10.
11.
12. class SinglyLinkedNode(object):
13.     def __init__(self, item=None, next_link=None):
14.         super(SinglyLinkedNode, self).__init__()
15.         self._item = item
16.         self._next = next_link
17.
18.     @property
19.     def item(self):
20.         return self._item
21.
```

```
22.     @item.setter
23.     def item(self, item):
24.         self._item = item
25.
26.     @property
27.     def next(self):
28.         return self._next
29.
30.     @next.setter
31.     def next(self, next):
32.         self._next = next
33.
34.     def __repr__(self):
35.         return repr(self.item)
36.
37.
38. class SinglyLinkedList(object):
39.     def __init__(self):
40.         super(SinglyLinkedList, self).__init__()
41.         # Setting head to none to indicate the list is empty.
42.         self._head = None
43.         self._length = 0
44.
45.     def __len__(self):
46.         """
47.        Gives the length of the Linked List
48.        :return: Length
49.        >>> len(slist)
50.        1
51.        """
52.         return self._length
53.
54.     def __iter__(self):
55.         """
56.        Loops through the content of the Linked list and yields the item values.
57.        :return: Yields item values of each node
58.        """
59.         temp_node = self._head
60.         while temp_node:
61.             yield str(temp_node.item)
62.             temp_node = temp_node.next
```

```
63.
64.     def contains_element(self, item):
65.         """
66.        Checks if the given item is present in the linked list
67.        :param item: Value to be searched in the list
68.        :return: Boolean value based on the search result
69.        """
70.         for temp_node_item in self:
71.             if temp_node_item == item:
72.                 return True
73.         return False
74.
75.     def __contains__(self, item):
76.         """
77.        Converts the given element to string and checks if it is present
78.        in the linked list using contains_element method
79.        :param item: Value to be searched in the list
80.        :return: Boolean value from the contains_element method
81.        >>> slist.prepend(10)
82.        >>> slist.__contains__(10)
83.        True
84.        """
85.         return self.contains_element(str(item))
86.
87.     def getnode(self):
88.         """
89.        Loops through the list to yields the nodes.
90.        :return: Node
91.        """
92.         temp_node = self._head
93.         while temp_node:
94.             yield temp_node
95.             temp_node = temp_node.next
96.
97.     def remove(self, item):
98.         """
99.        Deletes an item from the linked List if present.
100.               :param item: Value to be deleted
101.               :return: None
102.               >>> slist.remove(10)
103.               >>> len(slist)
```

```
104.                1
105.                """
106.                previous = None
107.                found = False
108.                for temp_node in self.getnode():  # searches to see if item is present
109.                    if temp_node.item == item:
110.                        found = True
111.                        break
112.                    else:
113.                        previous = temp_node  # holds previous node in case item not
    found
114.                if found:
115.                    if previous is None:
116.                        self._head = temp_node.next
117.                    else:
118.                        previous.next = temp_node.next  # removing the item
119.                    self._length -= 1
120.
121.            def prepend(self, item):
122.                """
123.                Adds new item to the Linked List at the beginning if not already present.
124.                :param item: Value to be added
125.                :return: None
126.                >>> slist.prepend(20)
127.                >>> len(slist)
128.                2
129.                """
130.                new_node = SinglyLinkedNode(item)
131.                if not self.__contains__(item):  # check if item already present
132.                    if self._head is not None:
133.                        new_node.next = self._head
134.                    self._head = new_node
135.                    self._length += 1
136.
137.            def __repr__(self):
138.                s = "List:" + "->".join([item for item in self])
139.                return s
140.
141.
142.        class ChainedHashDict(object):
143.            def __init__(self, bin_count=10, max_load=0.7, hashfunc=hash):
```

```
144.                 super(ChainedHashDict, self).__init__()
145.                 # Construct a new table
146.                 self.max_bin_capacity = bin_count
147.                 self.max_load_factor = max_load
148.                 self.hashing_func = hashfunc
149.                 self.length = 0
150.                 self.hash_array = [None]*bin_count
151.
152.             @property
153.             def load_factor(self):
154.                 """
155.                Gives the load factor of the hash table.
156.                :return: load factor
157.                >>> chained.load_factor
158.                Decimal('0.3')
159.                """
160.                 return Decimal(self.length)/Decimal(self.bin_count)
161.
162.             @property
163.             def bin_count(self):
164.                 """
165.                Gives the capacity of the hash table.
166.                :return: Maximum bin capacity
167.                >>> chained.bin_count
168.                10
169.                """
170.                 return self.max_bin_capacity
171.
172.             def rebuild(self, bincount):
173.                 """
174.                Doubles the size of the hash table and rearranges the contents based on
    new bin size.
175.                :param bincount: New Bin size
176.                :return: None
177.                >>> chained[0] = 10
178.                >>> chained.rebuild(20)
179.                >>> print chained.display()
180.                0:(0, 10)
181.                1:None
182.                2:(2, 10)
183.                3:None
```

```
                4:None
                5:None
                6:None
                7:None
                8:None
                9:None
                10:(10, 20)
                11:None
                12:None
                13:None
                14:None
                15:None
                16:None
                17:None
                18:None
                19:None
                """
                self.length = 0
                temp_hash_array = [None]*bincount  # create a temporary array that will be of new bin size.
                for i in range(self.bin_count):
                    sll1 = self.hash_array[i]
                    if sll1 is not None:
                        for list_node in sll1.getnode():
                            new_hash_index = self.hashing_func(list_node.item[0], bincount)
                            sll2 = SinglyLinkedList()
                            if temp_hash_array[new_hash_index] is not None:
                                sll2 = temp_hash_array[new_hash_index]
                            sll2.prepend(list_node.item)
                            temp_hash_array[new_hash_index] = sll2  # moving nodes to the temporary hash array
                            self.length += 1
                self.hash_array = []
                self.hash_array = list(temp_hash_array)  # populating the hash array with new arrangement.
                self.max_bin_capacity = bincount

            def __getitem__(self, key):
                """
                Gives the value associated with a key
```

```
221.                :param key: key to a particular value in the hash array
222.                :return: Value corresponding to key or message indicating key not present
223.                >>> chained[0] = 10
224.                >>> chained[0]
225.                10
226.                """
227.                 hash_index = self.hashing_func(key, self.bin_count)
228.                 sll = self.hash_array[hash_index]
229.                 if sll is not None:
230.                     for node in sll.getnode():
231.                         return node.item[1]
232.                 else:
233.                     return "Value not present."
234.
235.            def __setitem__(self, key, value):
236.                 """
237.                Inserting the given key, value pair into the hash array
238.                :param key: key to a particular value
239.                :param value: the actual value
240.                :return: None
241.                >>> chained[10] = 20
242.                >>> print chained[10]
243.                20
244.                """
245.                 if not self.__contains__(key):
246.                     if self.load_factor >= self.max_load_factor:  # if load factor
    greater than maximum load factor
247.                         self.rebuild(self.bin_count*2)  # Rebuilding the hash array
248.                     hash_index = self.hashing_func(key, self.bin_count)  # evaluating
    hash index
249.                     sll = self.hash_array[hash_index]
250.                     if sll is None:
251.                         sll = SinglyLinkedList()
252.                     tup = (key, value)
253.                     sll.prepend(tup)
254.                     self.hash_array[hash_index] = sll
255.                     self.length += 1
256.
257.            def __delitem__(self, key):
258.                 """
259.                Deletes key,value pair from the hash array
```

```
            :param key: key to identify key,value pair to be deleted
            :return: None
            >>> del chained[10]
            >>> print chained[10]
            Value not present.
            """
             hash_index = self.hashing_func(key, self.bin_count)
             sll = self.hash_array[hash_index]
             if sll is not None:
                 found = False
                 previous = None
                 for temp_node in sll.getnode():
                     if temp_node.item[0] == key:
                         found = True
                         break
                     else:
                         previous = temp_node
                 if found:
                     if previous is None:
                         sll._head = temp_node.next
                     else:
                         previous.next = temp_node.next
                 self.length -= 1

        def __contains__(self, key):
             """
            Checks if there is any key, value pair present in the hash arrray
            :param key: key to check
            :return: Boolean value based on search result
            >>> chained[2] = 10
            >>> chained[2]
            10
            """
             hash_index = self.hashing_func(key, self.bin_count)
             sll = self.hash_array[hash_index]
             if sll is None:
                 return False
             else:
                 for node in sll.getnode():
                     if node.item[0] == key:
                         return True
```

```
                    return False

        def __len__(self):
            """
           Gives the number of key, value pairs present in the hash array
           :return: length of hash array
           >>> len(chained)
           2
           """
            return self.length

        def display(self):
            """
           Returns a string showing the table with multiple lines
           and also items in each bin
           :return: Complete Hash table
           """
            string_list = None
            for i in range(self.bin_count):
                sll = self.hash_array[i]
                if sll is None:
                    s = str(i) + ':' + str('None')
                else:
                    s = str(i) + ':' + "->".join([item for item in sll])
                if string_list is None:
                    string_list = s
                else:
                    string_list += "\n"
                    string_list += s
            return string_list


    class OpenAddressHashDict(object):
        def __init__(self, bin_count=10, max_load=0.7, hashfunc=hash):
            super(OpenAddressHashDict, self).__init__()
            # initialize
            self.max_bin_capacity = bin_count
            self.max_load_factor = max_load
            self.hashing_func = hashfunc
            self.length = 0
            self.hash_array = [None]*bin_count
```

```
342.
343.             @property
344.             def load_factor(self):
345.                 """
346.                Gives the load factor of the hash table.
347.                :return: load factor
348.                >>> openAdd.load_factor
349.                Decimal('0.3')
350.                """
351.                 return Decimal(self.length)/Decimal(self.bin_count)
352.
353.             @property
354.             def bin_count(self):
355.                 """
356.                Gives the maximum capacity of the hash table.
357.                :return: maximum bin capacity
358.                >>> openAdd.bin_count
359.                10
360.                """
361.                 return self.max_bin_capacity
362.
363.             def get_hash_key(self, hash_array, key, probe_count, bincount):
364.                 """
365.                Evaluates the appropriate hash index of the given key
366.                :param hash_array: hash table which consists of all the key, value pairs
367.                :param key: key to identify the key, value pair
368.                :param probe_count: to increment probe count when searching for
    appropriate hash index
369.                :param bincount: capacity of the hash table
370.                :return: Hash index
371.                """
372.                 hash_index = self.hashing_func(key, probe_count, bincount)
373.                 if hash_array[hash_index] is not None:
374.                     probe_count += 1
375.                     hash_index = self.get_hash_key(hash_array, key, probe_count, bincoun
    t)
376.                 return hash_index
377.
378.             def rebuild(self, bincount):
379.                 """
```

```
380.                Rebuilds the hash table and Rearranges the existing data with new bin
    size
381.                :param bincount: new bin size
382.                :return: None
383.                >>> openAdd[0] = 10
384.                >>> openAdd.rebuild(20)
385.                >>> print openAdd.display()
386.                0:(0, 10)
387.                1:None
388.                2:(2, 10)
389.                3:None
390.                4:None
391.                5:None
392.                6:None
393.                7:None
394.                8:None
395.                9:None
396.                10:None
397.                11:(11, 20)
398.                12:None
399.                13:None
400.                14:None
401.                15:None
402.                16:None
403.                17:None
404.                18:None
405.                19:None
406.                """
407.                 # Rebuild this hash table with a new bin count
408.                 self.length = 0
409.                 temp_hash_array = [None]*bincount
410.                 for i in range(self.bin_count):
411.                     sll1 = self.hash_array[i]
412.                     if sll1 is not None:
413.                         for list_node in sll1.getnode():
414.                             new_hash_index = self.get_hash_key(temp_hash_array, list_nod
    e.item[0], 0, bincount)
415.                             sll2 = SinglyLinkedList()
416.                             sll2.prepend(list_node.item)
417.                             temp_hash_array[new_hash_index] = sll2
418.                             self.length += 1
```

```
419.                 self.hash_array = []
420.                 self.hash_array = list(temp_hash_array)
421.                 self.max_bin_capacity = bincount
422.
423.             def search(self, key, probe_count):
424.                 """
425.                Search to see if given key in present in the hash table.
426.                :param key: key to be searches
427.                :param probe_count: to keep track of no. of probes when searching for
    appropriate hash index
428.                :return: Key,Value node
429.                """
430.                 hash_index = self.hashing_func(key, probe_count, self.bin_count)
431.                 sll = self.hash_array[hash_index]
432.                 if sll is None:
433.                     return None
434.                 else:
435.                     for node in sll.getnode():
436.                         if node.item[0] == key:
437.                             return node
438.                     probe_count += 1
439.                     self.search(key, probe_count)
440.
441.             def __getitem__(self, key):
442.                 """
443.                Gives the value associated with the given key.
444.                :param key: key to get the associated value
445.                :return: Value
446.                >>> openAdd[0] = 10
447.                >>> openAdd[0]
448.                10
449.                """
450.                 node = self.search(key, 0)
451.                 if node is not None:
452.                     return node.item[1]
453.                 else:
454.                     return "Value not present."
455.
456.             def __setitem__(self, key, value):
457.                 """
458.                Insert the given key,value pair into the hash table.
```

```
459.                :param key: key to identify a value
460.                :param value: the actual value
461.                :return: None
462.                >>> openAdd[11] = 20
463.                >>> print openAdd[11]
464.                20
465.                """
466.                if not self.__contains__(key):
467.                    if self.load_factor >= self.max_load_factor:
468.                        self.rebuild(self.bin_count*2)
469.                    hash_index = self.get_hash_key(self.hash_array, key, 0, self.bin_cou
    nt)
470.                    sll = SinglyLinkedList()
471.                    tup = (key, value)
472.                    sll.prepend(tup)
473.                    self.hash_array[hash_index] = sll
474.                    self.length += 1
475.
476.            def __delitem__(self, key):
477.                """
478.                Remove the key, value pair represented by the key from the hash table
479.                :param key: key to identify the pair
480.                :return: None
481.                >>> del openAdd[10]
482.                >>> print openAdd[10]
483.                Value not present.
484.                """
485.                probe_count = 0
486.                found = False
487.                hash_index = self.hashing_func(key, probe_count, self.bin_count)
488.                sll = self.hash_array[hash_index]
489.                if sll is not None:
490.                    for node in sll.getnode():
491.                        if node.item[0] == key:
492.                            found = True
493.                            break
494.                        else:
495.                            probe_count += 1
496.                            hash_index = self.hashing_func(key, probe_count, self.bin_co
    unt)
497.                if found:
```

```
498.                     if self.hash_array[hash_index] is not None:
499.                         self.hash_array[hash_index] = None
500.                         self.length -= 1
501.
502.             def __contains__(self, key):
503.                 """
504.                Checks if a key, value pair is present in the hash table
505.                :param key: key to identify the key,value pair
506.                :return: Boolean value based on the search result
507.                >>> openAdd[2] = 10
508.                >>> openAdd[2]
509.                10
510.                """
511.                 node = self.search(key, 0)
512.                 if node is not None:
513.                     return True
514.                 return False
515.
516.             def __len__(self):
517.                 """
518.                Gives the number of items present in the hash table.
519.                :return: Length
520.                >>> len(openAdd)
521.                2
522.                """
523.                 return self.length
524.
525.             def display(self):
526.                 """
527.                Gives a string showing the table with multiple lines and items in each
    bin
528.                :return: String representing complete hash table.
529.                """
530.                 string_list = None
531.                 for i in range(self.bin_count):
532.                     sll = self.hash_array[i]
533.                     if sll is None:
534.                         s = str(i) + ':' + str('None')
535.                     else:
536.                         s = str(i) + ':' + "->".join([item for item in sll])
537.                     if string_list is None:
```

```
                        string_list = s
                    else:
                        string_list += "\n"
                        string_list += s
                return string_list


        class BinaryTreeNode(object):
            def __init__(self, data=None, left=None, right=None, parent=None):
                super(BinaryTreeNode, self).__init__()
                self.data = data
                self.left = left
                self.right = right
                self.parent = parent


        class BinarySearchTreeDict(object):
            def __init__(self):
                super(BinarySearchTreeDict, self).__init__()
                # initialize
                self.root = None
                self.length = 0

            def height_of_tree(self, root):
                """
               Evaluates the height of the BST recursively
               :param root: root node in the BST
               :return: height
               """
                if root is None:
                    return 0
                else:
                    left = self.height_of_tree(root.left)
                    right = self.height_of_tree(root.right)
                    return max(left, right) + 1

            @property
            def height(self):
                """
               Gives the height of the BST
               :return: height
```

```
579.             >>> print tree.height
580.             3
581.             """
582.              temp = self.root
583.              return self.height_of_tree(temp)
584.
585.         def inorder_traversal(self, node):
586.              """
587.             Recursively evaluates the the BST to yield nodes in "in-order" fashion
588.             :param node: a node in the BST
589.             :return: Yields the consequent node
590.             """
591.              if node is None:
592.                  raise StopIteration
593.              for node1 in self.inorder_traversal(node.left):
594.                  yield node1
595.              yield node
596.              for node2 in self.inorder_traversal(node.right):
597.                  yield node2
598.
599.         def preorder_traversal(self, node):
600.              """
601.             Recursively evaluates the the BST to yield nodes in "pre-order" fashion
602.             :param node: a node in BST
603.             :return: Yields the consequent node
604.             """
605.              if node is None:
606.                  raise StopIteration
607.              yield node
608.              for node1 in self.preorder_traversal(node.left):
609.                  yield node1
610.              for node2 in self.preorder_traversal(node.right):
611.                  yield node2
612.
613.         def postorder_traversal(self, node):
614.              """
615.             Recursively evaluates the the BST to yield nodes in "post-order" fashion
616.             :param node: a node in BST
617.             :return: Yields the consequent node
618.             """
619.              if node is None:
```

```
                raise StopIteration
            for node1 in self.postorder_traversal(node.left):
                yield node1
            for node2 in self.postorder_traversal(node.right):
                yield node2
            yield node

        def inorder_keys(self):
            """
           Prints the keys present in the BST using an INORDER traversal
           :return: Keys in INORDER traversal
           >>> tree[1] = 4
           >>> tree[6] = 12
           >>> tree[3] = 6
           >>> tree.inorder_keys()
           [1, 3, 6]
           """
            temp = self.root
            print ([node.data[0] for node in self.inorder_traversal(temp)])

        def postorder_keys(self):
            """
           Prints the keys present in the BST using an POSTORDER traversal
           :return: Keys in POSTORDER traversal
           >>> tree.postorder_keys()
           [3, 6, 1]
           """
            temp = self.root
            print ([node.data[0] for node in self.postorder_traversal(temp)])

        def preorder_keys(self):
            """
           Prints the keys present in the BST using an PREORDER traversal
           :return: Keys in PREORDER traversal
           >>> tree.preorder_keys()
           [1, 6, 3]
           """
            temp = self.root
            print ([node.data[0] for node in self.preorder_traversal(temp)])

        def items(self):
```

```
661.                """
662.               Gives the items (key and value) using an INORDER Traversal
663.               :return: Yields the node (key and value)
664.               """
665.                temp = self.root
666.                for node in self.inorder_traversal(temp):
667.                    yield node.data
668.
669.           def search_node(self, node, key):
670.                """
671.               Searches for key in the node and its children
672.               :param node: where the key is searched
673.               :param key: key being searched
674.               :return: node that contains the key
675.               """
676.                if node is None:
677.                    return None
678.                elif key == node.data[0]:
679.                    return node
680.                elif key < node.data[0]:
681.                    return self.search_node(node.left, key)
682.                else:
683.                    return self.search_node(node.right, key)
684.
685.           def __getitem__(self, key):
686.                """
687.               Gives the value associated with a key.
688.               :param key: key to get the associated value
689.               :return: Value
690.               >>> tree[1] = 4
691.               >>> tree[6] = 12
692.               >>> tree[3] = 6
693.               >>> tree[3]
694.               6
695.               """
696.                temp = self.root
697.                if temp is None:
698.                    return "Empty Tree."
699.                temp = self.search_node(temp, key)
700.                if temp is not None:
701.                    return temp.data[1]
```

```
            else:
                return "Item not available."

        def __setitem__(self, key, value):
            """
           Insert the key,value pair in the BST at its appropriate position
           :param key: identifier of position in BST
           :param value: actual value
           :return: None
           >>> tree[6] = 12
           >>> print tree[6]
           12
           """
            if not self.__contains__(key):
                new_item = BinaryTreeNode((key, value))
                temp1 = self.root
                temp2 = None
                while temp1 is not None:
                    temp2 = temp1
                    if key < temp1.data[0]:
                        temp1 = temp1.left
                    else:
                        temp1 = temp1.right
                new_item.parent = temp2
                if temp2 is None:
                    self.root = new_item
                elif new_item.data[0] < temp2.data[0]:
                    temp2.left = new_item
                else:
                    temp2.right = new_item
                self.length += 1

        @staticmethod
        def tree_minimum(node):
            """
           Gives the minimum key present in the node and its children
           :param node: where the key is searched
           :return: minimum key node
           >>> tree[1]
           4
           """
```

```
743.                 while node.left is not None:
744.                     node = node.left
745.                 return node
746.
747.             def transplant(self, node1, node2):
748.                 """
749.                Identify the successor of the node being deleted
750.                :param node1: left sub tree
751.                :param node2: right sub tree
752.                :return: None
753.                """
754.                 temp1 = node1.parent
755.                 if node1.parent is None:
756.                     self.root = node2
757.                 elif node1 == temp1.left:
758.                     temp1.left = node2
759.                 else:
760.                     temp1.right = node2
761.                 if node2 is not None:
762.                     node2.parent = node1.parent
763.
764.             def __delitem__(self, key):
765.                 """
766.                Remove the node identified by the given key
767.                :param key: key to identify node in the BST
768.                :return: None
769.                >>> tree[6] = 12
770.                >>> del tree[6]
771.                >>> tree[6]
772.                'Empty Tree.'
773.                """
774.                 temp1 = self.root
775.                 if temp1 is None:
776.                     return False
777.                 temp1 = self.search_node(temp1, key)
778.                 if temp1 is not None:
779.                     if temp1.left is None:
780.                         self.transplant(temp1, temp1.right)
781.                     elif temp1.right is None:
782.                         self.transplant(temp1, temp1.left)
783.                     else:
```

```
                    temp2 = self.tree_minimum(temp1.right)
                    if temp2.parent != temp1:
                        self.transplant(temp2, temp2.right)
                        temp2.right = temp1.right
                        temp3 = temp2.right
                        temp3.parent = temp2
                    self.transplant(temp1, temp2)
                    temp2.left = temp1.left
                    temp3 = temp2.left
                    temp3.parent = temp2
                self.length -= 1

        def __contains__(self, key):
            """
           Searches for the key in the BST
           :param key: node key being searched
           :return: Boolean value based on the search result
           >>> tree[6] = 12
           >>> tree[6]
           12
           """
            temp = self.root
            if temp is None:
                return False
            temp = self.search_node(temp, key)
            if temp is not None:
                return True
            else:
                return False

        def __len__(self):
            """
           Gives the number of nodes in the BST
           :return: length
           >>> len(tree)
           3
           """
            return self.length

        def display(self):
            """
```

```
            Print the keys using INORDER on one line and PREORDER on the next
            :return: None
            >>> tree.display()
            [1, 3, 6]
            [1, 6, 3]
            """
             self.inorder_keys()
             self.preorder_keys()


    def chaining_hash_func(key, bin_count):
        """
       Hash function that generates hash index for chained hash tables.
       :param key: the value for which hash index should be calculated
       :param bin_count: Hash Table capacity
       :return: Hash Index
       """
        if isinstance(key, int):
            hash_value = key % bin_count
        else:
            hash_value = ord(key) % bin_count
        return hash_value


    def open_addressing_hash_func(key, probe, bin_count):
        """
       Hash function that generates hash index for open addressing hash tables
       using Linear probing.
       :param key: the value for which hash index should be calculated
       :param probe: probe to find next available bin.
       :param bin_count: Hash Table capacity
       :return: Hash Index
       """
        auxiliary_hash_value = chaining_hash_func(key, bin_count)
        return (auxiliary_hash_value + probe) % bin_count


    def terrible_hash(bin):
        """A terrible hash function that can be used for testing.

       A hash function should produce unpredictable results,
```

```
        but it is useful to see what happens to a hash table when
        you use the worst-possible hash function.  The function
        returned from this factory function will always return
        the same number, regardless of the key.

        :param bin:
            The result of the hash function, regardless of which
            item is used.

        :return:
            A python function that can be passes into the constructor
            of a hash table to use for hashing objects.
        """
         def hashfunc(x, item):
             return bin
         return hashfunc


    def testing_singly_linked_list():
        # Singly Linked List testing
        singlyLL = SinglyLinkedList()
        max_list_length = random.randrange(1, 20)
        for i in range(1, max_list_length):
            listitem = random.randrange(1, 50)
            singlyLL.prepend(listitem)
        print (singlyLL)
        # length
        l = 'length = ' + str(len(singlyLL))
        print l
        # contains
        check_item1 = random.randrange(1, 50)
        if check_item1 in singlyLL:
            print 'Is ' + str(check_item1) + ' present: True'
        else:
            print 'Is ' + str(check_item1) + ' present: False'
        check_item2 = 20
        singlyLL.prepend(check_item2)
        print (singlyLL)
        if check_item2 in singlyLL:
            print 'Is ' + str(check_item2) + ' present: True'
        else:
```

```
907.                 print 'Is ' + str(check_item2) + ' present: False'
908.             # delete
909.             singlyLL.remove(check_item2)
910.             print 'After Delete: '
911.             print (singlyLL)
912.             print ' '
913.
914.
915.         def testing_chained_hash_dictionary():
916.             chd = ChainedHashDict(10, 0.8, chaining_hash_func)
917.             for i in range(0, 9):
918.                 chd[i] = random.randrange(1, 500)
919.             print chd.display()
920.             print chd.length
921.             print chd.bin_count
922.             print chd.load_factor
923.             check_chd1 = random.randrange(1, 500)
924.             if check_chd1 in chd:
925.                 print 'Is ' + str(check_chd1) + ' present: True'
926.             else:
927.                 print 'Is ' + str(check_chd1) + ' present: False'
928.             check_chd2 = 2
929.             chd[check_chd2] = 200
930.             if check_chd2 in chd:
931.                 print 'Is ' + str(check_chd2) + ' present: True'
932.             else:
933.                 print 'Is ' + str(check_chd2) + ' present: False'
934.             del chd[2]
935.             print chd.display()
936.
937.
938.         def testing_open_addressing_hash_dictionary():
939.             oahd = OpenAddressHashDict(10, 0.9, open_addressing_hash_func)
940.             for i in range(0, 9):
941.                 oahd[i] = random.randrange(1, 500)
942.             print oahd.display()
943.             print oahd.length
944.             print oahd.bin_count
945.             print oahd.load_factor
946.             check_oahd1 = random.randrange(1, 500)
947.             if check_oahd1 in oahd:
```

```
948.                print 'Is ' + str(check_oahd1) + ' present: True'
949.            else:
950.                print 'Is ' + str(check_oahd1) + ' present: False'
951.            oahd[2] = 200
952.            if check_oahd1 in oahd:
953.                print 'Is ' + str(check_oahd1) + ' present: True'
954.            else:
955.                print 'Is ' + str(check_oahd1) + ' present: False'
956.            del oahd[2]
957.            print oahd.display()
958.
959.
960.        def testing_binary_search_tree():
961.            # Binary Search Tree testing
962.            bst = BinarySearchTreeDict()
963.            for i in range(1, 20):
964.                bst[i] = random.randrange(1, 500)
965.            bst[1] = 45
966.            print (bst[1])
967.            print (bst.height)
968.            print (len(bst))
969.            print " "
970.            bst.inorder_keys()
971.            bst.preorder_keys()
972.            bst.postorder_keys()
973.            print " "
974.            bst.display()
975.            print " "
976.            for node in bst.items():
977.                print node
978.            print " "
979.            del bst[1]
980.            print (bst[1])
981.
982.
983.        def main():
984.            # Thoroughly test your program and produce useful out.
985.            #
986.            # Do at least these kinds of tests:
987.            #  (1)  Check the boundary conditions (empty containers,
988.            #       full containers, etc)
```

```
989.            #  (2)  Test your hash tables for terrible hash functions
990.            #       that map to keys in the middle or ends of your
991.            #       table
992.            #  (3)  Check your table on 100s or randomly generated
993.            #       sets of keys to make sure they function
994.            #  (4)  Make sure that no keys / items are lost, especially
995.            #       as a result of deleting another key
996.
997.            testing_singly_linked_list()
998.            testing_chained_hash_dictionary()
999.            testing_open_addressing_hash_dictionary()
1000.           testing_binary_search_tree()
1001.           import doctest
1002.           doctest.testmod(verbose=1, extraglobs={'tree': BinarySearchTreeDict(),
1003.                                                  'chained':
    ChainedHashDict(10, 0.8, chaining_hash_func),
1004.                                                  'openAdd':
    OpenAddressHashDict(10, 0.9, open_addressing_hash_func),
1005.                                                  'slist': SinglyLinkedList()})
1006.
1007.       if __name__ == '__main__':
1008.           main()
```

Example Output:

Output from flake8:

```
C:\Users\MadhulikaBushi\Desktop\501\Coding Assignments\Coding Assignment 2>flake
8 --max-complexity 8 ds_assignment(1).py

C:\Users\MadhulikaBushi\Desktop\501\Coding Assignments\Coding Assignment 2>
```

Output from Chained Hash Dictionary class:

```
0:(0, 430)
1:(1, 300)
2:(2, 367)
3:(3, 244)
4:(4, 9)
5:(5, 47)
6:(6, 289)
7:(7, 468)
8:(8, 279)
9:None
9
10
0.9
Is 94 present: False
Is 2 present: True
0:(0, 430)
1:(1, 300)
2:
3:(3, 244)
4:(4, 9)
5:(5, 47)
6:(6, 289)
7:(7, 468)
8:(8, 279)
9:None

Process finished with exit code 0
```

Output from Singly Linked List class:

```
List:23->27->49->22->15->39->30->42
length = 8
Is 2 present: False
List:20->23->27->49->22->15->39->30->42
Is 20 present: True
After Delete:
List:23->27->49->22->15->39->30->42
```

Output from Open Addressed Hash Dictionary class:

```
0:(0, 248)
1:(1, 376)
2:(2, 74)
3:(3, 151)
4:(4, 35)
5:(5, 398)
6:(6, 176)
7:(7, 188)
8:(8, 69)
9:None
9
10
0.9
Is 138 present: False
Is 138 present: False
0:(0, 248)
1:(1, 376)
2:None
3:(3, 151)
4:(4, 35)
5:(5, 398)
6:(6, 176)
7:(7, 188)
8:(8, 69)
9:None

Process finished with exit code 0
```

Output from Binary Search Tree Dictionary class:

```
47
19
19

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

(1, 47)
(2, 15)
(3, 37)
(4, 184)
(5, 430)
(6, 416)
(7, 374)
(8, 160)
(9, 291)
(10, 270)
(11, 229)
(12, 305)
(13, 121)
(14, 295)
(15, 238)
(16, 194)
(17, 75)
(18, 399)
(19, 111)

Item not available.

Process finished with exit code 0
```

Output from DocTests:

```
   1 tests in __main__.SinglyLinkedList.__len__
   2 tests in __main__.SinglyLinkedList.prepend
   2 tests in __main__.SinglyLinkedList.remove
56 tests in 70 items.
56 passed and 0 failed.
Test passed.
```

Reflection

1. Understood the implementation of classes in python.
2. Understood the standards of writing docstrings in python.
3. Understood the usage of yield and StopIteration in handling list of data.
4. Traversing through the lists in all the classes was tedious and involved lot of book keeping.
5. Open addressing hash index generation was complicated even though the probing chosen was linear probing.