UNIVERSITY OF TORONTO
Faculty of Arts and Science

August 2016 Examinations

CSC148H1, Section L5101
Duration – 3 hours

No Aids Allowed

Student Number: ⎵⎵⎵⎵⎵⎵⎵⎵⎵⎵

Last Name: ______________________________

First Name: ______________________________

Do **not** turn this page until you have received the signal to start.
In the meantime, please fill out the identification section above,
and read the instructions below carefully.

This exam consists of 7 questions on 18 pages (including this one). Pages 19 to 29 are Python reference sheets including classes that we developed in lectures/labs. When you receive the signal to start, please make sure that your copy of the test is complete, and feel free to tear off the reference sheets, from which you can use any of the classes in your answers.

Please answer questions in the space provided. You will earn 20% for any question you leave blank or write "I cannot answer this question," on. We think we have provided a lot of space for your work, but please do not feel you need to fill all available space.

You must achieve 40% of *Max(ThisExam, WeightedAverage(Test1, Test2, ThisExam))*, to pass this course.

Write neatly and concisely. If we cannot read it, we cannot grade it.

GOOD LUCK!

Question	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**Total**
Initial								
Mark	/10	/8	/6	/8	/8	/10	/8	/58

Question 1. Time Complexity. [10 Marks]

For each of the following parts, circle the big-oh expression—from the list—that gives the best upper bound for each code fragment, and briefly explain your choice.

Part (a) [2 Marks]

```
 i, j, sum = 0, 0, 0
while i**2 < n:
    while j**2 < n:
        sum += i * j
        j += 2
    i += 5
```

$O(1)$ $O(log\ n)$ $O(\sqrt{n})$ $O(n)$ $O(n\ log\ n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n!)$

Part (b) [2 Marks]

```
sum = 0
for i in range(n):
    j = n
    while j>i:
        sum += j - i
        j -= 1
```

$O(1)$ $O(log\ n)$ $O(\sqrt{n})$ $O(n)$ $O(n\ log\ n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n!)$

Part (c) [2 Marks]

```
for i in rang(n):
    for j in range(n):
        k = 0
        while k<n:
            c += 1
            k +=100
```

$O(1)$ $O(log\,n)$ $O(\sqrt{n})$ $O(n)$ $O(n\,log\,n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n!)$

Part (d) [2 Marks]

```
i, j, c = 0, 1, 1
while c < n:
    s, i, j = i+j, j, s
    c +=1
```

$O(1)$ $O(log\,n)$ $O(\sqrt{n})$ $O(n)$ $O(n\,log\,n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n!)$

Part (e) [2 Marks]

```
def f(n):
    if n==0 or n == 1: return n
    else: return f(n-2)+f(n-1)
```

$O(1)$ $O(log\,n)$ $O(\sqrt{n})$ $O(n)$ $O(n\,log\,n)$ $O(n^2)$ $O(n^3)$ $O(2^n)$ $O(n!)$

Question 2. BST Insertion/Deletion. **[8 Marks]**

Read the `bst_insert` in the reference sheets.

Part (a) [3 Marks]

Assume the following data—from left to right—are inserted in a new BST:

28 12 18 40 48 14 12 35 33 31

Draw the final BST.

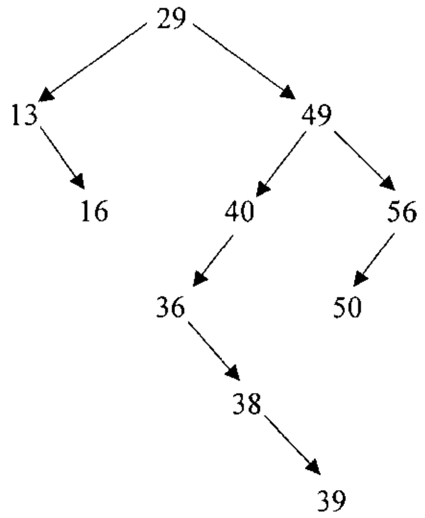
Part (b) [2 Marks]

Draw a BST that is as *balanced* as possible and represents the following data:

28 18 40 48 14 35 33 31

Part (c) [3 Marks]

Read the `bst_delete` method in the reference sheets, and consider the following BST:



Draw one final BST, after deletion of 29 and 49.

Question 3. Binary (Search) Trees. **[6 Marks]**

Read the declaration of class `BinaryTree` in the reference sheets, as well as the `docstring` and the example for `concatenate` below. Then, implement `concatenate(t1,t2)`.

Note: You may mutate `t2`; but, you should not mutate `t1`.

```
def concatenate(t1, t2):
    """
    Concatenate binary search trees t1 and t2

    Precondition: the biggest value in t1 is smaller than
                  the smallest value in t2

    :param t1: a binary search tree
    :type t1: BinaryTree|None
    :param t2: a binary search tree
    :type t2: BinaryTree|None
    :return a binary search tree produced by concatenation of t1 and t2
    :rtype: BinaryTree|None

    >>> t1 = BinaryTree(4,BinaryTree(3,BinaryTree(1)), BinaryTree(5))
    >>> t2 =BinaryTree(10,BinaryTree(8,BinaryTree(6),BinaryTree(9)),
                                                            BinaryTree(14))
    >>> t3 = concatenate(t1, t2)
    >>> print(t3)
        14
    10
            9
        8
            6
                    5
                4
                    3
                        1
    <BLANKLINE>
    """
```

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.
Clearly label each such solution with the appropriate question and part number.

Question 4. Binary Trees. [8 Marks]

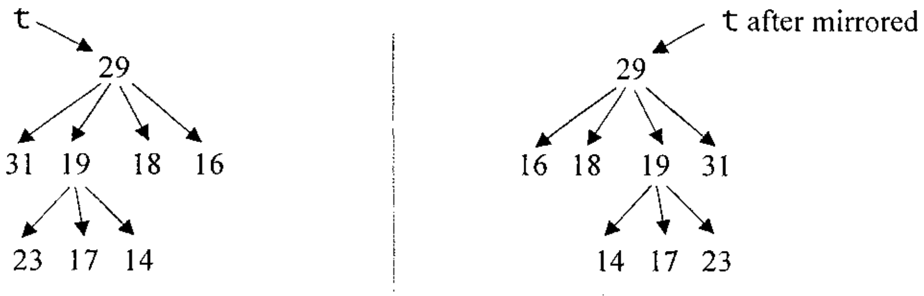
Read the declaration of class `BinaryTree` in the reference sheets. Also, read the `docstring` and examples below. Then, implement `is_subtree(t1, t2).`

```
def is_subtree(t1, t2):
    """
    Return whether t1 is a subtree of t2

    :param t1: a binary tree
    :type t1: BinaryTree|None
    :param t2: a binary tree
    :type t2: BinaryTree|None
    :rtype: bool

    >>> t1 = BinaryTree(4,BinaryTree(3,BinaryTree(1)), BinaryTree(5))
    >>> t2 = BinaryTree(10,\
             BinaryTree(4,BinaryTree(3,BinaryTree(1)),BinaryTree(5)))
    >>> is_subtree(t1, t2)
    True
    >>> is_subtree(t2, t1)
    False
    >>> t3 = BinaryTree(10,\
             BinaryTree(4,BinaryTree(3),BinaryTree(1)),BinaryTree(5))
    >>> is_subtree(t1, t3)
    False
    >>> is_subtree(BinaryTree(5), t1)
    True
    >>> is_subtree(t2, t2)
    True
    >>> is_subtree(None, None)
    True

    """
    # to be implemented by you
```

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.
Clearly label each such solution with the appropriate question and part number.

Question 5. General Trees. [8 Marks]

When a general tree is *mirrored*, the children's order of each subtree is reversed.



Read the declaration of class `Tree` in the reference sheets, as well as the `docstring` and examples for `mirror(t)` below. Then, implement `mirror(t)`.

```
def mirror(t):
    """
    Mutate tree t recursively such that the order of children of each subtree
    is reversed

    :param t: a general tree
    :type t: Tree|None
    :rtype: None

     >>> t = Tree(17)
     >>> print(t)
     17
     >>> mirror(t)
     >>> print(t)
     17
     >>> t1 = Tree(19, [Tree(14), t, Tree(23)])
     >>> print(t1)
     19
        14
        17
        23
     >>> mirror(t1)
     >>> print(t1)
     19
        23
        17
        14
     >>> t3 = Tree(29, [Tree(31), t1, Tree(18), Tree(16)])
     >>> print(t3)
     29
        31
        19
           23
           17
           14
```

```
       18
       16
    >>> mirror(t3)
    >>> print(t3)
    29
       16
       18
       19
          14
          17
          23
       31

    """
```

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.
Clearly label each such solution with the appropriate question and part number.

Question 6. Linked Lists. **[10 Marks]**

Read the following API as well as the docstring and examples of insert_sorted below. Then, implement insert_sorted(self, item).

```
class DoublyLinkedListNode:
    """
    Node to be used in linked list

    === Public Attributes ===
    :param object value: data this DoublyLinkedListNode represents
    :param DoublyLinkedListNode next_: successor to this DoublyLinkedListNode
    :param DoublyLinkedListNode last: predecessor to this DoublyLinkedListNode
    """

    def __init__(self, value, next_=None, last=None):
        """
        Create DoublyLinkedListNode self with data value, successor next_,
        and predecessor last.

        :param value: data of this linked list node
        :type value: object
        :param next_: successor to this DoublyLinkedListNode.
        :type next_: DoublyLinkedListNode|None
        :param last: predecessor to this DoublyLinkedListNode.
        :type last: DoublyLinkedListNode|None

        """
        self.value, self.next_, self.last = value, next_, last


class DoublyLinkedList:
    """
    Bidirectional linked list

    === Attributes ==
    :param: front: front node of this DoublyLinkedList
    :type front: DoublyLinkedList|None
    :param: back: back node of this CircularLinkedList
    :type back: DoublyLinkedListNode|None
    """

    def __init__(self, value):
        """
        Create DoublyLinkedList self with data value.

        :param value: data of this circular linked list
        :type value: object
        """
        self.front = DoublyLinkedListNode(value)
        self.back = self.front
    def __str__(self):
        """
        Return a user-friendly representation of this DoublyLinkedList.

        :rtype: str

        >>> my_list = DoublyLinkedList(5)
        >>> print(my_list)
        <- 5 ->
        >>> my_list.insert_sorted(16)
        >>> print(my_list)
        <- 5 <=> 16 ->
        """
        if self.front is None:
            return "<-->"
        else:
            s = "<- {}".format(self.front.value)
            curr = self.front
            while curr.next_:
                s += " <=> {}".format(curr.next_.value)
                curr = curr.next_
            s += " ->"

            return s

    def insert_sorted(self, item):
        """
        Insert value in a correct spot in the sorted DoublyLinkedList self

        :param item: the data being inserted
        :type item: object
        :rtype: None

        >>> lst = DoublyLinkedList(19)
        >>> print(lst)
        <- 19 ->
        >>> lst.insert_sorted(11)
        >>> print(lst)
        <- 11 <=> 19 ->
        >>> lst.insert_sorted(28)
        >>> print(lst)
        <- 11 <=> 19 <=> 28 ->
        >>> lst.insert_sorted(14)
        >>> print(lst)
        <- 11 <=> 14 <=> 19 <=> 28 ->
        """
        # to be implemented by you
```

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.

Clearly label each such solution with the appropriate question and part number.

Question 7. Stacks. [8 Marks]

Read the declaration of class Stack in the reference sheets, and develop a method to copy a stack.

The copy method should be *a non-mutating one*: when the method ends it should return a new stack, but the original stack should be unchanged.

Note: You should only use the Stack methods in this question; no other data structure (such as Python lists) are allowed.

```
def copy(self):
```

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.
Clearly label each such solution with the appropriate question and part number.

Use the space on this "blank" page for scratch work, or for any solution that did not fit elsewhere.
Clearly label each such solution with the appropriate question and part number.

Last Name.................................. First Name................................ Student# ....

Short Python function/method descriptions:

```
__builtins__:
 len(x) -> integer
   Return the length of the list, tuple, dict, or string x.
 max(L) -> value
   Return the largest value in L.
 min(L) -> value
   Return the smallest value in L.
 range([start], stop, [step]) -> list of integers
   Return a list containing the integers starting with start and ending with stop - 1 with step specifying
   the amount to increment (or decrement). If start is not specified, the list starts at 0.
   If step is not specified, the values are incremented by 1.
  sum(L) -> number
   Returns the sum of the numbers in L.

dict:
 D[k] -> value
   Return the value associated with the key k in D.
 k in d -> boolean
   Return True if k is a key in D and False otherwise.
 D.get(k) -> value
   Return D[k] if k in D, otherwise return None.
 D.keys() -> list of keys
   Return the keys of D.
 D.values() -> list of values
   Return the values associated with the keys of D.
 D.items() -> list of (key, value) pairs
   Return the (key, value) pairs of D, as 2-tuples.

float:
 float(x) -> floating point number
   Convert a string or number to a floating point number, if possible.

int:
 int(x) -> integer
   Convert a string or number to an integer, if possible. A floating point argument will be truncated
   towards zero.

list:
 x in L -> boolean
   Return True if x is in L and False otherwise.
 L.append(x)
   Append x to the end of list L.
 L1.extend(L2)
   Append the items in list L2 to the end of list L1.
 L.index(value) -> integer
   Return the lowest index of value in L.
 L.insert(index, x)
   Insert x at position index.
```

L.pop()
Remove and return the last item from L.
L.remove(value)
Remove the first occurrence of value from L.
L.sort()
Sort the list in ascending order.

Module random: randint(a, b)
Return random integer in range [a, b], including both end points.

str:
x in s -> boolean
Return True if x is in s and False otherwise.
str(x) -> string
Convert an object into its string representation, if possible.
S.count(sub[, start[, end]]) -> int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.
S.find(sub[,i]) -> integer
Return the lowest index in S (starting at S[i], if i is given) where the string sub is found or -1 if sub does not occur in S.
S.split([sep]) -> list of strings
Return a list of the words in S, using string sep as the separator and any whitespace string if sep is not specified.

set:
{1, 2, 3, 1, 3} -> {1, 2, 3}
s.add(...)
Add an element to a set
set()
Create a new empty set object
x in s
True iff x is an element of s

list comprehension:
[<expression with x> for x in <list or other iterable>]

functional if:
<expression 1> if <boolean condition> else <expression 2>
-> <expression 1> if the boolean condition is True, otherwise <expression 2>

======Class Container ====================

```
class Container:
    """
    A data structure to store and retrieve objects.
    This is an abstract class that is not meant to be instantiated itself,
    but rather subclasses are to be instantiated.
    """
    def __init__(self):
        """
        Create a new and empty Container self.
        """
        self._content = None
        raise NotImplemented ("This is an abstract class, define or use its subclass")

    def add(self, obj):
        """
        Add object obj to Container self.
        :param obj: object to place onto Container self
        :type obj: Any
        :rtype: None
        """
        raise NotImplemented ("This is an abstract class, define or use its subclass")

    def remove(self):
        """
        Remove and return an element from Container self.
        Assume that Container self is not empty.
        :return an object from Container slef
        :rtype: object
        """
        raise NotImplemented ("This is an abstract class, define or use its subclass")

    def is_empty(self):
        """
        Return whether Container self is empty.
        :rtype: bool
        """
        return len(self._content) == 0

    def __eq__(self, other):
        """
        Return whether Container self is equivalent to the other.

        :param other: a Container
        :type other: Container
        :rtype: bool
        """
        return type(self)== type(other) and self._content == other._content

    def __str__(self):
        """
        Return a human-friendly string representation of Container.
        :rtype: str
        """
        return str(self._content)
```

```
======Class Stack ====================
from container import Container

class Stack(Container):
 """Last-in, first-out (LIFO) stack.
 """
  def __init__(self):
    """Create a new, empty Stack self.

    Overrides Container.__init__

    """
    self._content = []

  def add(self, obj):
    """ Add object obj to top of Stack self.

    Overrides Container.add

    :param obj: object to place on Stack
    :type obj: Any
    :rtype: None
    >>> s = Stack()
    >>> s.add(1)
    >>> s.add(2)
    >>> print(s)
    [1, 2]
    """
    self._content.append(obj)

  def remove(self):
    """
    Remove and return top element of Stack self.

    Assume Stack self is not empty.

    Overrides Container.remove

    :rtype: object
    >>> s = Stack()
    >>> s.add(5)
    >>> s.add(7)
    >>> s.remove()
    7
    """
    return self._content.pop()
```

```
======Class Queue =====================
from container import Container

class Queue (Container):
  """A first-in, first-out (FIFO) queue.
  """

  def __init__(self):
    """
    Create and initialize new Queue self.

    Overrides Container.__init__

    """
    self._content = []

  def add(self, obj):
    """
    Add object at the back of Queue self.

    Overrides Container.add

    :param obj: object to add
    :type obj: object
    :rtype: None
    >>> q = Queue()
    >>> q.add(1)
    >>> q.add(2)
    >>> print(q)
    [1, 2]
    """
    self._content.append(obj)

  def remove(self):
    """
    Remove and return front object from Queue self.

    Queue self must not be empty.

    Overrides Container.remove

    :rtype: object

    >>> q = Queue()
    >>> q.add(3)
    >>> q.add(5)
    >>> q.remove()
    3
    """
    return self._content.pop(0)
```

```
======Class LinkedListNode====================
class LinkedListNode:
    """
    Node to be used in linked lists

    === Public Attributes ===
    :param LinkedListNode next_: successor to this LinkedListNode
    :param object value: data this LinkedListNode represents
    """

    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.

        :param value: data of this linked list node
        :type value: object
        :param next_: successor to this LinkedListNode.
        :type next_: LinkedListNode|None
        """
        self.value, self.next_ = value, next_

    def __str__(self):
        """
        Return a user-friendly representation of this LinkedListNode.

        :rtype: str

        >>> n = LinkedListNode(5, LinkedListNode(7))
        >>> print(n)
        5 -> 7 ->|
        """
        s = "{} ->".format(self.value)
        cur_node = self
        while cur_node is not None:
            if cur_node.next_ is None:
                s += "|"
            else:
                s += " {} ->".format(cur_node.next_.value)
            cur_node = cur_node.next_
        return s

    def __eq__(self, other):
        """
        Return whether LinkedListNode self is equivalent to other.

        :param LinkedListNode self: this LinkedListNode
        :param LinkedListNode|object other: object to compare to self.
        :rtype: bool

        >>> LinkedListNode(5).__eq__(5)
        False
        >>> n1 = LinkedListNode(5, LinkedListNode(7))
        >>> n2 = LinkedListNode(5, LinkedListNode(7, None))
        >>> n1.__eq__(n2)
        True
        """
        self_node, other_node = self, other
        while (self_node is not None and type(self_node) is type(other_node) and
                self_node.value == other_node.value):
            self_node, other_node = self_node.next_, other_node.next_
        return self_node is None and other_node is None

======Class (general) Tree=====================
class Tree:
    """A bare-bones Tree ADT that identifies the root with the entire tree.

    === Public Attributes ===
    :param object value: data for this binary tree node
    :param list[Tree] children: children of this binary tree node
    """

    def __init__(self, value=None, children=None):
        """Create Tree self with content value and 0 or more children

        :param value: value contained in this tree
        :type value: object
        :param children: possibly-empty list of children
        :type children: list[Tree]
        """
        self.value = value
        # copy children if not None
        self.children = children.copy() if children else []

    def __eq__(self, other):
        """Return whether this Tree is equivalent to other.

        :param other: object to compare to self
        :type other: object}Tree
        :rtype: bool

        >>> t1 = Tree(5)
        >>> t2 = Tree(5, [])
        >>> t1 == t2
        True
        >>> t3 = Tree(5, [t1])
        >>> t2 == t3
        False
        """
        return (type(self) is type(other) and self.value == other.value and self.children == other.children)

def descendants_from_list(t, list_, arity):
    """Populate Tree t's descendants from list_, filling them in level order, with up to arity children per
    node. Then, return t.

    :param t: tree to populate from list_
    :type t: Tree
    :param list_: list of values to populate from
    :type list_: list
    :param arity: maximum branching factor
    :type arity: int
    :rtype: Tree

    >>> descendants_from_list(Tree(0), [1, 2, 3, 4], 2)
    Tree(0, [Tree(1, [Tree(3), Tree(4)]), Tree(2)])
    """
    q = Queue()
    q.add(t)
    list_ = list_.copy()
    while not q.is_empty():  # unlikely to happen
        new_t = q.remove()
        for i in range(0, arity):
            if len(list_) == 0:
                return t  # our work here is done
            else:
                new_t_child = Tree(list_.pop(0))
                new_t.children.append(new_t_child)
                q.add(new_t_child)
    return t
```

```
======Class BinaryTree=====================
class BinaryTree:
  """ A Binary Tree, i.e. arity 2.

  === Public Attributes ===
  :param object data: data for this binary tree node
  :param BinaryTree|None left: left child of this binary tree node
  :param BinaryTree|None right: right child of this binary tree node
  """

  def __init__(self, data, left=None, right=None):
    """
    Create BinaryTree self with data and children left and right.

    :param data: data of this node
    :type data: object
    :param left: left child
    :type left: BinaryTree|None
    :param right: right child
    :type right: BinaryTree|None
    """
    self.data, self.left, self.right = data, left, right

  def __eq__(self, other):
    """
    Return whether BinaryTree self is equivalent to other.

    :param other: object to check equivalence to self
    :type other: Any
    :rtype: bool

    >>> BinaryTree(7).__eq__("seven")
    False
    >>> b1 = BinaryTree(7, BinaryTree(5))
    >>> b1.__eq__(BinaryTree(7, BinaryTree(5), None))
    True
    """
    return (type(self) == type(other) and
            self.data == other.data and
            (self.left, self.right) == (other.left, other.right))


  def __str__(self, indent=""):
    """
    Return a user-friendly string representing BinaryTree (self)
    inorder.  Indent by indent.
```

```
        >>> b = BinaryTree(1, BinaryTree(2, BinaryTree(3)), BinaryTree(4))
        >>> print(b)
            4
        1
            2
                3
        <BLANKLINE>
        """
        right_tree = (self.right.__str__(indent + "    ") if self.right else "")
        left_tree = self.left.__str__(indent + "    ") if self.left else ""
        return (right_tree + "{}{}\n".format(indent, str(self.data)) + left_tree)

    def __contains__(self, value):
        """
        Return whether tree rooted at node contains value.

        :param value: value to search for
        :type value: object
        :rtype: bool

        >>> BinaryTree(5, BinaryTree(7), BinaryTree(9)).__contains__(7)
        True
        """
        return (self.data == value or
            (self.left and value in self.left) or
            (self.right and value in self.right))

def bst_insert(node, data):
    ''' (BTNode, object) -> BTNode

    Insert data in BST rooted at node if necessary, and return new root.

    >>> b = BTNode(5)
    >>> b1 = bst_insert(b, 3)
    >>> print(b1)
    5
        3
    <BLANKLINE>
    '''
    return_node = node
    if not node:
        return_node = BTNode(data)
    elif data < node.data:
        node.left = bst_insert(node.left, data)
    elif data > node.data:
        node.right = bst_insert(node.right, data)
    else:  # nothing to do
        pass
    return return_node
```

```
def bst_delete(root, data):
    parent = None
    current = root
    while current is not None and current.data != data:
        if data < current.data:
            parent = current
            current = current.left
        elif data > current.data:
            parent = current
            current = current.right
        else: pass # Element is in the tree pointed at by current

    if current is None: return False # Element is not in the tree

    # Case 1: current has no left child
    if current.left is None:
        # Connect the parent with the right child of the current_node
        # Special case, assume the node being deleted is at root
        if parent is None:
            current = current.right
        else:
            # Identify if parent left or parent right should be connected
            if data < parent.data:
                parent.left = current.right
            else:
                parent.right = current.right
    else:
        # Case 2: The current node has a left child
        # Locate the rightmost node in the left subtree of
        #  the current node and also its parent
        parent_of_right_most = current
        right_most = current.left

    while right_most.right is not None:
        parent_of_right_most = right_most
        right_most = right_most.right # Keep going to the right
        #  Replace the element in current by the element in rightMost
    current.element = right_most.element
    # Eliminate rightmost node
    if parent_of_right_most.right == right_most:
        parent_of_right_most.right = right_most.left
    else:        # Special case: parent_of_right_most == current
        parent_of_right_most.left = right_most.left
    return True # Element deleted successfully
```