PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

April 2017 Examinations

CSC148H1S
Duration — 3 hours
No aids allowed.

PLEASE HAND IN

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.)

This exam consists of 7 questions on 22 pages (including this one). *When you receive the signal to start, please make sure that your copy of the exam is complete.*

Please answer questions in the space provided. If you need additional space, clearly indicate on the question page where to find your answer.

You will earn 20% for any question you leave blank or write "I cannot answer this question" on. You may earn substantial part marks for writing down the outline of a solution and indicating which steps are missing.

You must achieve 40% of the marks on this final exam to pass this course.

There is a Python API at the end of this exam.

1: _____/ 8

2: _____/ 8

3: _____/10

4: _____/ 6

5: _____/ 6

6: _____/10

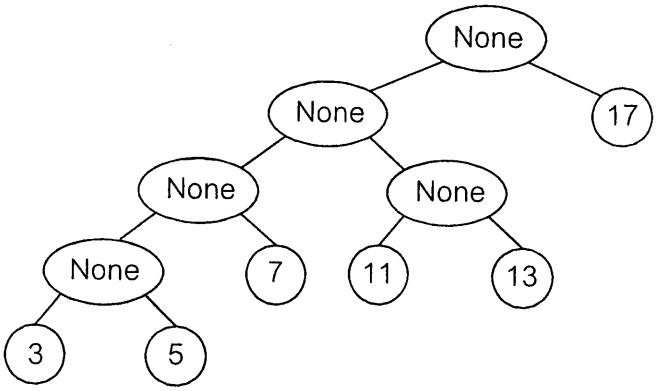
7: _____/ 8

TOTAL: _____/56

Good Luck!

Question 1. [8 MARKS]

Answer the questions below using this Huffman tree diagram:



Part (a) [3 MARKS]

Write down the dict that maps bytes (AKA symbols) to bits (AKA codes) corresponding to the given Huffman tree.

Part (b) [3 MARKS]

Write down the string of bits that encode these bytes:

`[3, 17, 5, 17, 11, 7, 13]`

Part (c) [2 MARKS]

Is it possible for an encoding based on a Huffman tree built from a frequency dictionary to have an average of more than 8.0 bits/symbol? Justify your answer. **Warning**: This question is more challenging than the 2 marks allocated suggests, and you should skip it unless you have a completely convincing argument.

Question 2. [8 MARKS]

In each subquestion below circle the Θ expression that best expresses the run-time complexity of function f. Briefly justify your choice.

Part (a) [2 MARKS]

```
def f(n):
    sum = 0
    i = 0
    while i**2 < n**2:
        sum = sum + i
        i = i + 1
    return sum
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n\lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

Part (b) [2 MARKS]

We measure the following running times for **f(n)**:

```
f(1):       1 second
f(10):      100 seconds
f(100):     10000 seconds
f(1000):    1000000 seconds
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n\lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

Part (c) [2 MARKS]

Assume that **d** is a dictionary with **n** items where no collisions have occurred, that **list_** is an unsorted list with n^2 elements, and **key** is an immutable Python object.

```
def f(key, list_):
    return d[key] in list_
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n\lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

Part (d) [2 MARKS]

We measure the following running times for **f(n)**:

```
f(2):     2 seconds
f(4):     4 seconds
f(8):     6 seconds
f(16):    8 seconds
f(32):    10 seconds
```

$\Theta(1)$ $\Theta(\lg(n))$ $\Theta(n)$ $\Theta(n\lg(n))$ $\Theta(n^2)$ $\Theta(n^3)$ $\Theta(2^n)$

Question 3. [10 MARKS]

Consider a Help Queue system, like the one in the CSC Help Centre. Students can optionally choose a course they want help with (or not), then swipe their TCard to be added to the help queue. Then, instructors and TAs can choose the next student from the help queue to help. Assume that there are no duplicate student names.

Students can swipe their card again if they're already in the help queue to be reminded of their number. Instructors and TAs can either choose the next student from the whole help queue, or the next student that matches a particular course.

On the following page, write the **HelpQueue** class so that it implements the Help Queue system described above, and interacts properly with the client code in the **if __name__ == '__main__'** block below. (That is, that the client code runs without any AssertionErrors.)

```
class HelpQueueEntry:
    """ A simple class to represent one entry in the help queue """

     def __init__(self, name, number, course=''):
        """ Create this new HelpQueueEntry representing a student name,
        the queue position number, and an optional course.

        @param self HelpQueueEntry: this help queue entry
        @param name str: the name of the student
        @param number int: the position in the queue
        @param course str: the course to get help with
        """
        self._name, self._number, self._course = name, number, course

    def _get_name(self):
        return self._name

    def _get_number(self):
        return self._number

    def _get_course(self):
        return self._course

    name = property(_get_name)
    number = property(_get_number)
    course = property(_get_course)


if __name__ == '__main__':
    hq = HelpQueue()
    assert hq.process_swipe('Amy', 'CSC108') == 0
    assert hq.process_swipe('Bo') == 1
    assert hq.process_swipe('Chen', 'CSC148') == 2
    assert hq.process_swipe('Amy') == 0
    assert hq.get_next_student('CSC148').number == 2
    assert hq.get_next_student('CSC148') is None
    assert hq.get_next_student('CSC165') is None
    assert hq.get_next_student().number == 0
    assert hq.get_next_student().number == 1
    assert hq.get_next_student() is None
```

```
class HelpQueue:

    """ A class to represent a help queue.

    === Attributes ===
    @param list[HelpQueueEntry] helpqueue: student entries waiting for help
    @param int next_number: number to assign to next entry to join helpqueue
    """
```

This page has been left intentionally (mostly) blank, in case you need space.

Question 4. [6 MARKS]

Read the header of **width** and then implement its body. **Hint**: a helper function to find the number of items at a given depth in all sub-lists combined may be a good idea here. You are given **max_depth** in case you need it to know when to stop.

```
def width(list_, max_depth):
    """ Return the maximum number of items that occur at the same depth
    in list_ or its sub-lists combined.  These could be list or non-list
    items. Elements may be lists or non-lists.

    @param list[list|object] list_: a possibly nested list
    @param max_depth int: maximum depth of list_
    @rtype: int

    >>> list_ = [0, 1]
    >>> width(list_, 1)
    2
    >>> list_ = [[0, 1], 2, [3, [[], 4]]]
    >>> width(list_, 4)
    4
    >>> # 4 elements: 0, 1, 3, [[], 4]
    """
```

Question 5. [6 MARKS]

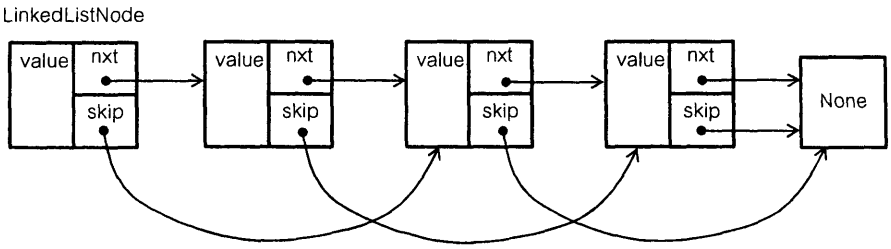
Read the declaration of class Tree below, and then implement function prune on the next page.

```
class Tree:
    """
    A bare-bones Tree ADT that identifies the root with the entire tree.
    """

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

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

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

        @param Tree self: this tree
        @param object|Tree other: object to compare to self
        @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 prune(t, predicate):
    """ Return a new tree with the same values as t, except it prunes (omits) all paths of t that start
    with nodes where predicate(node.value) == False. If predicate(t.value) == False, then prune
    returns None.

    Assume that all values in t are ints, and that predicate always takes an int and returns a bool.

    @param Tree t: tree to prune
    @param function[int, bool] predicate: predicate on t's values.
    @rtype: Tree|None

    >>> t1 = Tree(6, [Tree(8), Tree(9)])
    >>> t2 = Tree(4, [Tree(11), Tree(10)])
    >>> t3 = Tree(7, [Tree(3), Tree(12)])
    >>> t = Tree(5, [t1, t2, t3])
    >>> t3_pruned = Tree(7, [Tree(12)])
    >>> def predicate(v): return v > 4
    >>> prune(t, predicate) == Tree(5, [t1, t3_pruned])
    True
    """
```

Question 6. [10 MARKS]

A skiplist is defined as a sorted linked list which in addition to the **nxt** reference, stores an additional "skip" reference. A skip reference refers to the node located immediately after nxt, if such a node exists, or None otherwise, as shown in the diagram below.



Your task is to implement two methods: **precursors**, and **insert**. These two methods are necessary to insert a new element with a given value in the list, in a way that the list remains sorted in ascending order of the node values, and the skip references are correctly maintained for every list node.

1. The **insert** method takes as a parameter a value (the value of the new node to be created and inserted in the list by this method), and the two nodes that would precede the new node once inserted (the two nodes with the highest two values less than the value passed as a parameter). This method will insert the new node into the list, and adjust all links necessary. Please be careful to correctly adjust the front, back, and size as well, when necessary.

2. The **precursors** method takes a value as a parameter and returns a tuple containing two nodes. The second node in the tuple is the list node with the largest value less than the value passed as a parameter, if such a node exists, or None otherwise. The first node is the predecessor of the second node from the tuple, if such a node exists, or None otherwise. If the list is empty, or the insertion must be done before the front node, then this method will return a tuple containing two None values.

Read the declaration of classes **LinkedListNode** and **LinkedList** below, and then implement the two methods of class LinkedList, on Pages 14 and 15. **Hint**: draw diagrams first!

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

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

        @param LinkedListNode self: this LinkedListNode
        @param int value: data of this linked list node
        @param LinkedListNode|None nxt: successor to this LinkedListNode.
        @rtype: None
        """
        self.value, self.nxt = value, nxt
        if nxt is not None:
            self.skip = nxt.nxt
        else:
            self.skip = None

        # continued on following page
```

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

        @param LinkedListNode self: 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.nxt is None:
                s += "|"
            else:
                s += " {} ->".format(cur_node.nxt.value)
            cur_node = cur_node.nxt
        return s

    # end of LinkedListNode class


class LinkedList:
    """ Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList a non-negative integer
    """

    def __init__(self):
        """ Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        """
        self.front, self.back = None, None
        self.size = 0

    def __str__(self):
        """ Return a human-friendly string representation of LinkedList self.

        @param LinkedList self: this LinkedList
        @rtype: str

        >>> lnk = LinkedList()
        >>> lnk.insert(5, lnk.precursors(3)[0], lnk.precursors(3)[1])
        >>> print(lnk)
        5 ->|
        """
        return str(self.front)
```

```
def precursors(self, value):
    """
    Returns a tuple containing the two list nodes with the two highest
    values which are less than the method argument 'value'.

    @param LinkedList self: this LinkedList
    @param int value: value to insert
    @rtype: (LinkedListNode|None, LinkedListNode|None)

    >>> lnk = LinkedList()
    >>> lnk.precursors(3)
    (None, None)
    >>> a = LinkedListNode(3)
    >>> lnk.front, lnk.back, lnk.size = a, a, 1
    >>> lnk.precursors(1)
    (None, None)
    >>> b = LinkedListNode(1, a)
    >>> lnk.front, lnk.size = b, 2
    >>> pre1 = lnk.precursors(5)[0]
    >>> pre2 = lnk.precursors(5)[1]
    >>> pre1.value, pre2.value
    (1, 3)
    """
```

```
def insert(self, value, prev, cur):
    """
    Inserts a new node with value after node cur. Updates all links correctly.
    This is a method of class LinkedList.

    @param LinkedList self: this LinkedList
    @param int value: value to insert
    @param LinkedListNode|None cur: node before the one we are inserting
    @param LinkedListNode|None prev: node before cur
    @rtype: None

    >>> lnk = LinkedList()
    >>> lnk.insert(3, lnk.precursors(3)[0], lnk.precursors(3)[1])
    >>> lnk.insert(0, lnk.precursors(0)[0], lnk.precursors(0)[1])
    >>> lnk.insert(2, lnk.precursors(2)[0], lnk.precursors(2)[1])
    >>> lnk.insert(1, lnk.precursors(1)[0], lnk.precursors(1)[1])
    >>> print(lnk.front)
    0 -> 1 -> 2 -> 3 ->|
    >>> print(lnk.back)
    3 ->|
    >>> lnk.size
    4
    >>> print(lnk.front.skip)
    2 -> 3 ->|
    >>> print(lnk.front.nxt.skip)
    3 ->|
    """
```

This page has been left intentionally (mostly) blank, in case you need space.

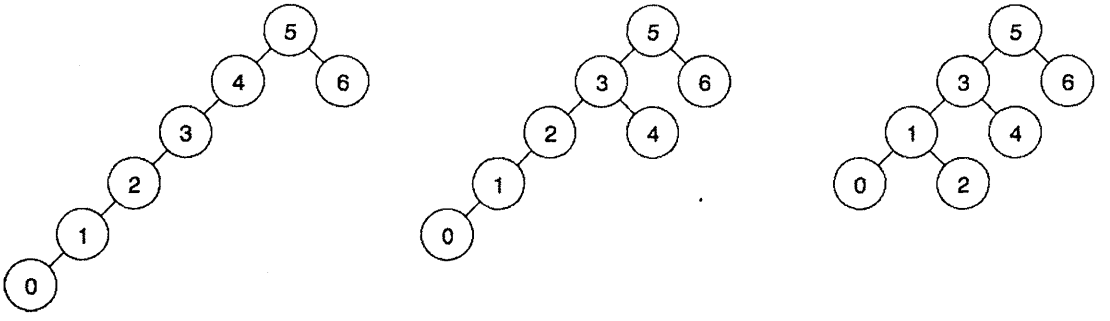
Question 7. [8 MARKS]

A *left streak* is defined as a sequence of three or more nodes on the leftmost side of the tree where none of the nodes in the sequence have a right child. A property of a left streak is that it can be broken to reduce the height of the tree by 1, while maintaining the binary search tree property. The *streak parent* is the node whose left child is the shallowest (closest to the root) node in the streak.

In the example below, the tree on the left contains a left streak whose parent is the node with data 5.

The tree in the middle has that streak fixed, but still contains a left streak whose parent is the node with data 3.

The tree on the right shows the same tree with all streaks fixed.



Your task is to implement the function that fixes all left streaks in a BST. You should use the provided helper function that finds a streak in your solution, and you do not need to handle any streaks that this helper function would not find.

Read the declaration of class BSTree and the find_left_streak helper function below and on the next page, and then implement function fix_left_streaks on Page 19. (Note: the height method has been included for use in a docstring example to help you understand the problem. We do not expect it to be useful to you in your solution.) Hint: draw diagrams!

```
class BSTree:

    """ A simple binary search tree ADT """

    def __init__(self, data, left=None, right=None):
        """ Create BSTree self with content data, left child left,
        and right child right.

        @param BSTree self: this binary search tree
        @param object data: data contained in this tree
        @param BSTree left: left child of this tree
        @param BSTree right: right child of this tree
        @rtype: None
        """
        self.data, self.left, self.right = data, left, right
```

```
    # BSTree class continued

    def height(self):
        """ Return the number of nodes on the longest path from the tree
        rooted at this binary search tree to a leaf.

        @param BSTree self: this binary search tree
        @rtype: int
        """
        if self.left is None and self.right is None:
            return 1
        if self.left is None:
            return 1 + self.right.height()
        if self.right is None:
            return 1 + self.left.height()
        return 1 + max(self.left.height(), self.right.height())

    # end of the BSTree class



def find_left_streak(t):
    """ Return the parent node of the shallowest left streak in t,
    or None if there is no left streak.

    @param BSTree t: the root of the whole tree
    @rtype: BSTree|None

    >>> left = BSTree(4, (BSTree(3, BSTree(2, BSTree(1, BSTree(0))))))
    >>> t = BSTree(5, left, BSTree(6))
    >>> find_left_streak(t).data
    5
    >>> t.left.right = BSTree(4.5)
    >>> find_left_streak(t).data
    4
    """
    streak_parent = t
    while streak_parent.left is not None:
        node = streak_parent.left
        if (node.left is not None and node.right is None) and \
           (node.left.left is not None and node.left.right is None):
            return streak_parent
        streak_parent = node
    # if the end of the loop is reached, there is no left streak - return None
```

```
def fix_left_streaks(t):
    """ Modify t by fixing all left streaks.

    @param BSTree t: the tree to fix
    @rtype: None

    >>> left = BSTree(4, (BSTree(3, BSTree(2, BSTree(1, BSTree(0))))))
    >>> t = BSTree(5, left, BSTree(6))
    >>> t.height()
    6
    >>> t.left.right is None
    True
    >>> t.left.left.right is None
    True
    >>> fix_left_streaks(t)
    >>> t.height()
    4
    >>> t.left.right.data == 4
    True
    >>> t.left.left.right.data == 2
    True
    """
```

This page has been left intentionally (mostly) blank, in case you need space.

Total Marks = 56

Short Python function/method descriptions, and classes

```
__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.pop(i)
    Remove and return L[i]
  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
  {1, 2, 3}.union({2, 4}) -> {1, 2, 3, 4}
  {1, 2, 3}.intersection({2, 4}) -> {2}
  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>
```