Name: _____ Net ID: _____

NYU, Tandon School of Engineering
CS-1134: Data Structures and Algorithms — Fall 2017

CS-1134 – Final Exam

Monday, December 18, 2017

- You have two hours.
- There are 6 questions all together, with 100 points total.
- The exam has **TWO Parts**:
  - The first part of the exam contains:
    - This cover page.
    - Documentation of the interface of some of the classes we implemented in the lectures. **You may use these classes and methods** without implementing them, unless explicitly stated otherwise**.**
    - A couple of pages for scratch work. **What you write in those pages will not be graded**, but you must hand it in with your exam.
  - The second part of the exam contains the questions you need to answer. Write your answers clearly and concisely, in the spaces on the exam.
    Try to avoid writing near the edge of the page.
    YOU MAY NOT USE THE BACKSIDE OF THE EXAM PAPERS, as they will not be looked at.
    If you need extra space for an answer, use the **extra page at the end of the exam** and **mark it clearly**, so we can find it when we're grading.
- **Don't use pencils**, as they don't show up well when scanned.
- Write your Name and NetID at the head of each page.
- Calculators are not allowed.
- Read every question completely before answering it.
- For any questions about runtime, give an asymptotic analysis.
- You do not have to do error checking. Assume all inputs to your functions are as described
- Cell phones, and any other electronic gadgets must be turned off.
- Do not talk to any students during the exam. If you truly do not understand what a question is asking, you may raise your hand when one of the CS1134 instructors is in the room.

```
class DoublyLinkedList:
    class Node:
        def __init__(self, data=None, prev=None, next=None):
            """initializes a new Node object containing the
            following attributes:
            1. data - to store the current element
            2. next - a reference to the next node in the list
            3. prev - a reference to the previous node in the list """

        def disconnect(self):
            """deprecates the node by setting all its attributes to None"""


    def __init__(self):
        """initializes an empty DoublyLinkedList object.
        A list object holds references to two "dummy" nodes:
        1. header - a node before the primary sequence
        2. trailer - a node after the primary sequence
        also a size count attribute is maintained"""

    def __len__(self):
        """returns the number of elements stored in the list"""

    def is_empty(self):
        """returns True if"f the list is empty"""

    def first_node(self):
        """returns a reference to the node storing the
        first element in the list"""

    def last_node(self):
        """returns a reference to the node storing the
        last element in the list"""

    def add_after(self, node, data):
        """adds data to the list, after the element stored in node.
        returns a reference to the new node (containing data)"""

    def add_first(self, data):
        """adds data as the first element of the list
        returns a reference to the new node (containing data)"""

    def add_last(self, data):
        """adds data as the last element of the list
        returns a reference to the new node (containing data)"""

    def add_before(self, node, data):
        """adds data to the list, before the element stored in node.
        returns a reference to the new node (containing data)"""

    def delete(self, node):
        """removes node from the list, and returns the data stored in it"""

    def __iter__(self):
        """an iterator that allows to iterate over the
        elements of the list from start to end"""

    def __str__(self):
        """returns a string representation of the list"""
```

```
class LinkedBinaryTree:
    class Node:
        def __init__(self, data, left=None, right=None, parent=None):
            """initializes a new Node object with the following attributes:
            1. data - to store the current element
            2. left - a reference to the left child of the node
            3. right - a reference to the right child of the node
            4. parent - a reference to the parent of the node"""


    def __init__(self, root=None):
        """initializes a LinkedBinaryTree object with the structure
        given in root (or empty if root is None). A tree object holds:
        1. root - a reference to the root node or None if tree is empty
        2. size - a node count"""

    def __len__(self):
        """returns the number of nodes in the tree"""

    def is_empty(self):
        """returns True if"f the tree is empty"""

    def subtree_count(self, curr_root):
        """returns the number of nodes in the subtree rooted by curr_root"""

    def preorder(self):
        """generator allowing to iterate over the nodes of
        the (entire) tree in a preorder order"""

    def subtree_preorder(self, curr_root):
        """generator allowing to iterate  in a preorder order
        over the nodes of the subtree rooted with curr_root"""

    def postorder(self):
        """generator allowing to iterate over the nodes of
        the (entire) tree in a postorder order"""

    def subtree_postorder(self, curr_root):
        """generator allowing to iterate  in a postorder order
        over the nodes of the subtree rooted with curr_root"""

    def inorder(self):
        """generator allowing to iterate over the nodes of
        the (entire) tree in an inorder order"""

    def subtree_inorder(self, curr_root):
        """generator allowing to iterate  in an inorder order
        over the nodes of the subtree rooted with curr_root"""

    def breadth_first(self):
        """generator allowing to iterate over the nodes of the
        (entire) tree level by level, each level from left to right"""

    def __iter__(self):
        """generator allowing to iterate over the data stored in the
        tree level by level, each level from left to right"""
```

```
class HashTableMap:
    class Item:
        def __init__(self, key, value):
            """initializes a new Item object with the following attributes:
            1. key - to store a key
            2. value - to store the value (associated to the key) """


    def __init__(self):
        """initializes an empty HashTableMap (hash-table) object"""

    def __len__(self):
        """returns the number of entries in the table"""

    def is_empty(self):
        """returns True if"f the table is empty"""

    def __getitem__(self, key):
        """returns the value associated to key, or raises a KeyError
        exception if key is not in the table.
        runs in O(1) average time"""

    def __setitem__(self, key, value):
        """adds the value associated by key, to the table. If key is
        already associated to an old value, it replaces it with value.
        runs in O(1) average time"""

    def __delitem__(self, key):
        """removes the value associated to key from the table, or raises
         a KeyError exception if key is not in the table.
         runs in O(1) average time"""

    def __iter__(self):
        """generator allowing to iterate over the keys in the table"""
```

Name: _____ Net ID: _____

Scratch
(This paper will not be graded)

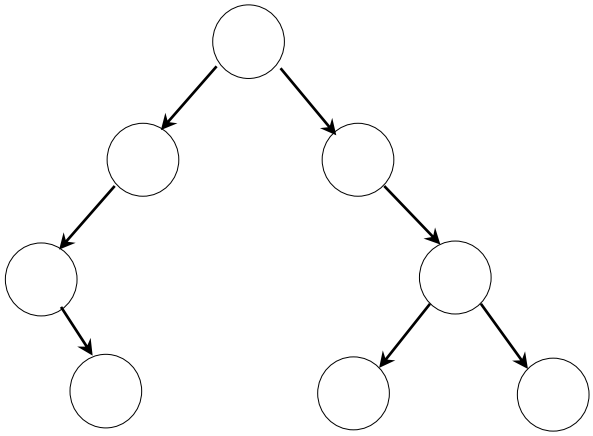
Name: _____ Net ID: _____

Scratch
(This paper will not be graded)

Name: _____ Net ID: _____

Question 1 (16 points)

a. Fill the nodes of the following tree with the numbers 1, 2, 3, 4, 5, 6, 7, 8, so that the resulting tree would be a **binary-search tree**.



b. Let *T* be a binary-search tree. If we traverse *T* in preorder we get the following sequence:

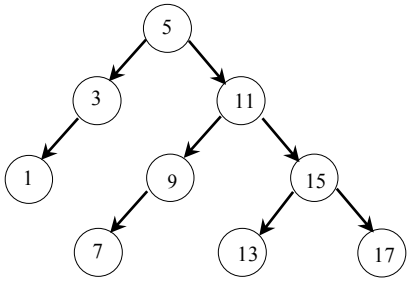
Preorder(T): 4, 2, 1, 3, 10, 7, 5, 6, 9, 8

Draw *T*.

Name: _____ Net ID: _____

Question 2 (9 points)

Given the following AVL tree:



Draw the AVL tree you get after inserting 14.

Name: _____ Net ID: _____

Question 3 (11 points)

In this question, you will insert items to a $N=11$ length **open-addressing** hash table, where we use the **division method** for hashing (the hash function is: $h(k) = k \bmod 11$), and **quadratic probing** for resolving collisions.

Draw the table resulted after inserting the following keys: 36, 71, 58, 90, 25, 17, 80, 60

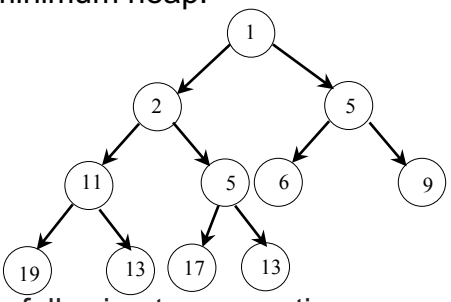
Note:

1. Do not rehash/resize the table
2. When using quadratic probing, you try to place k in the i^{th} try ($i = 0, 1, 2, \ldots$), at index: $(h(k) + i^2) \bmod N$.

Name: _____ Net ID: _____

Question 4 (16 points)

Given the following minimum heap:



We are executing the following two operations:

- We start with inserting 3
- We then call delete_min

For each one of the operations above, draw the resulting heap **both in its tree representation and in its array representation**.

After inserting 3:

After calling delete_min:

Name: _____ Net ID: _____

Question 5 (20 points)

Give a **recursive implementation** for the following function:

```
def level_list(root, level)
```

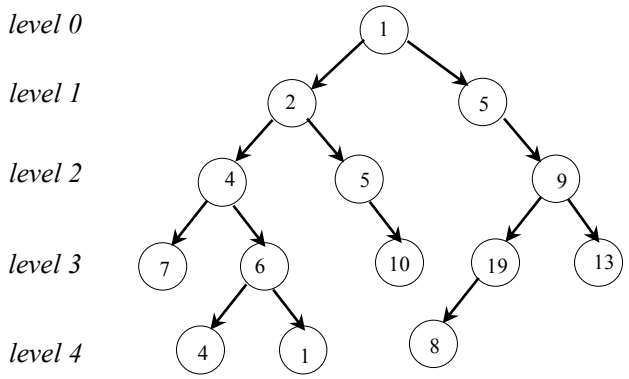
The function is given:

1. `root` - a reference to a node, that indicates the root of a subtree in a `LinkedBinaryTree` object.
2. An integer `level`

When called, the function will **return a list**, containing the data stored in all the nodes of level `level` in the subtree rooted by `root`.

The order of the data in the returned list, should be as if you traverse the level's nodes from left to right.

For example, if `bin_tree`, is a `LinkedBinaryTree` object, representing the following tree:



The call `level_list(bin_tree.root, 3)`, should return `[7, 6, 10, 19, 13]`.

The call `level_list(bin_tree.root, 6)`, should return `[]`.

Name: _____ Net ID: _____

```
def level_list(root, level):
```

Name: _____ Net ID: _____

Question 6 (30 points)

In this question, we will look at the *Play List ADT*. This ADT is used to maintain a sequential collection of songs. Each song could be played individually, or all songs could be played sequentially.

This ADT supports:

- *pl = PlayList()* – creates an empty *PlayList* object.
- *pl.add_song(new_song_name)* – adds the song *new_song_name* to the end of the songs sequence
- *pl.add_song_after(song_name, new_song_name)* – adds the song *new_song_name* to the songs sequence, right after the song *song_name*, or raises a *KeyError* exception if *song_name* is not in the play list.
- *pl.play_song(song_name)* – plays the song *song_name*, **or raises a *KeyError* exception if *song_name* is not in the play list**.
- *pl.play_list()* – plays all the songs in the *PlayList* by their sequential order.

Define the `PlayList` class that implements the *Play List ADT*.

Note:

1. Assume there is a function `play(song_name)`, that plays (on the computer's speakers) the song with the name `song_name`.
2. Assume the user will not add the same song more than once.

.

For example, after the following interaction:

```
>>> pl = PlayList()
>>> pl.add_song("Feel It Still")
>>> pl.add_song("Perfect")
>>> pl.add_song("Havana")
>>> pl.add_song_after("Perfect", "Thunder")
>>> pl.add_song("Feel It Still", "Something Just Like This")
```

The call `pl.play_song("Perfect")` will play the song "Perfect".
The call `pl.play_list()` will play the songs (in this order): "Feel It Still", "Something Just Like This", "Perfect", "Thunder" and "Havana".

Implementation requirements:

1. In your implementation, the runtime of the methods should be as follows:
   - `add_song(…)` should run in $\theta(1)$ **average time**.
   - `add_song_after(…)` should run in $\theta(1)$ **average time**.
   - `play_song(…)` should run in $\theta(1)$ **average time**.
   - `play_list(…)` should run in $\theta(n)$ **average time**, where n is the number of songs in the play list.
2. In this question, you are not allowed to use Python's build-in dictionary. However, you may use the `HashTableMap` for the same behavior (see documentation in page 4 of the exam).
3. If your solution needs more than 1-3 lines for each method, it is probably over complicated.
4. **Make sure `add_song_after` runs in $\theta(1)$ average time.**

Name: _____ Net ID: _____

```
class PlayList:

    def __init__(self):
```

_____

_____

_____

_____

_____

_____

```
    def add_song(self, new_song_name):
```

_____

_____

_____

_____

_____

_____

```
    def add_song_after(self, song_name, new_song_name):
```

_____

_____

_____

_____

_____

_____

Name: _____ Net ID: _____

```
def play_song(self, song_name):
```

_____

_____

_____

_____

_____

_____

```
def play_list(self):
```

_____

_____

_____

_____

_____

_____

Name: _____ Net ID: _____

EXTRA PAGE IF NEEDED

Note question numbers of any questions or part of questions that you are answering here.
Also, write “ANSWER IS ON LAST PAGE” near the space provided for the answer.