CS1134 Exam 2

Lee;Cindy

TOTAL POINTS

100 / 100

QUESTION 1

1 15 pts

1.1 1.1 **5 / 5**

✓ **- 0 Correct**

- 5 Incorrect Binary Search Tree

1.2 1.2 **5 / 5**

✓ **- 0 Correct**

- 5 Incorrect insertion order

- 5 Didn't give numbers to insert

- 1 Missing number

1.3 1.3 **5 / 5**

✓ **- 0 Correct**

- 5 Part (i) was wrong

- 5 Incorrect Deletion

- 2 Only one of the two steps

- 5 Blank

QUESTION 2

2 2 **5 / 5**

✓ **- 0 Correct**

- 5 Incorrect tree

QUESTION 3

3 3 **20 / 20**

✓ **- 0 Correct**

- 6 The data is added as the first item in the list, not as the second.

- 4 Failure to set the new node's prev field correctly

- 4 Failure to set the new node's next field correctly

- 4 Failure to set the first node's next field to the new node

- 4 Failure to set the second node's prev field to the new node

- 4 Failure to make a new node

- 20 Use of PList's methods

- 20 Wrong

QUESTION 4

4 4 **20 / 20**

✓ **- 0 Correct**

- 15 You do not use/modify the result of the recursive calls

- 10 You need two recursive calls if you have two children

- 5 Bad use of default parameter

- 7 You should return a list of size three

- 20 Wrong

- 10 Infinite recusion

- 2 + and += concatenates lists

- 5 You do not use the results of both recursive calls

- 5 Slow. Should run in time linear in the size of the tree.

- 0 Click here to replace this description.

QUESTION 5

5 5 **20 / 20**

✓ **- 0 Correct**

- 8 Failure to move variable containing node to the right.

- 1 Should be yourvar._right, not self.right(yourvar)

- 20 Blank

- 5 Invalid use of a default value

- 20 Wrong

- 2 Function has too many parameters, need default values for extra one(s).

- 10 Code doen't work

- 5 Slow. Should run in O(h) for a tree of height h.

- 5 Need to do something with the return value of the recursive call

- 4 Minor error

QUESTION 6

6 6 20 / 20

✓ - 0 **Correct**

- 20 You have an instance variable other than a PList
- 5 Boost does not work for k smaller than the number of items in the structure
- 5 Boost does not work for k larger than the number of items in the list
- 10 Basic functionality (other than boost) is incorrect
- 20 Wrong
- 20 Blank
- 5 Basic functionality (other than boost) has some errors

Exam 2

John Iacono's 1134 Sections

Instructions

- **This exam will be scanned! Do not fold or un-staple the pages. If you need more space go to the last page.**
- The exam is closed book. You may have two sheets of paper. No calculators, electronic devices, or magnifying device of any kind.
- Slow algorithms that are correct are with more than fast ones which are incorrect.
- You do not have do do error checking unless otherwise indicated. Assume all inputs to your functions are as described.
- Code is provided for the positional list, linked binary tree, and binary search tree classes. Nothing written on these sheets will be graded.
- **No questions during the exam.** If you think something is unclear, say so, and make (and write) a reasonable assumption to answer the question.

What is your name?

Cindy Lee

Positional list code:

```
class PList:
    class _Node:
        __slots__='_data','_prev','_next'
        def __init__(self,data,prev,next):
            self._data=data
            self._prev=prev
            self._next=next
    class Position:
        def __init__(self,plist,node):
            self._plist=plist
            self._node=node
        def data(self):
            return self._node._data
        def __eq__(self,other):
            return type(other) is type(self) and other._node is self._node
        def __ne__(self,other):
            return not (self == other)
    def _validate(self,p):
        if not isinstance(p,self.Position):
            raise TypeError("p must be proper Position type")
        if p._plist is not self:
            raise ValueError('p does not belong to this PList')
        if p._node._next is None:
            raise ValueError('p is no longer valid')
        return p._node
    def _make_position(self,node):
        if node is self._head or node is self._tail:
            return None
        else:
            return self.Position(self,node)
    def __init__(self):
        self._head=self._Node(None,None,None)
        self._head._next=self._tail=self._Node(None,self._head,None)
        self._size=0
    def __len__(self):
        return self._size
    def is_empty(self):
        return self._size==0
    def first(self):
        return self._make_position(self._head._next)
    def last(self):
        return self._make_position(self._tail._prev)
    def before(self,p):
        node=self._validate(p)
        return self._make_position(node._prev)
    def after(self,p):
        node=self._validate(p)
        return self._make_position(node._next)
    def __iter__(self):
        pos = self.first()
        while pos:
            yield pos.data()
            pos=self.after(pos)
    def _insert_after(self,data,node):
        newNode=self._Node(data,node,node._next)
        node._next._prev=newNode
        node._next=newNode
        self._size+=1
        return self._make_position(newNode)
    def add_first(self,data):
        return self._insert_after(data,self._head)
    def add_last(self,data):
        return self._insert_after(data,self._tail._prev)
    def add_before(self,p,data):
        node=self._validate(p)
        return self._insert_after(data,node._prev)
    def add_after(self,p,data):
        node=self._validate(p)
        return self._insert_after(data,node)
    def delete(self,p):
        node=self._validate(p)
        data=node._data
        node._prev._next=node._next
        node._next._prev=node._prev
        node._prev=node._next=node._data=None
        self._size-=1
        return data
    def replace(self,p,data):
        node=self._valdiate(p)
        olddata=node._data
        node._data=data
        return olddata
    def rev_itr(self):
        #Code omitted
```

Code for BST's from class:

```
class BST:
    class _Node:
        def __init__(self,parent,left,right,data):
            self._left=left
            self._right=right
            self._parent=parent
            self._data=data
    def __init__(self):
        self._root=None
    def insert(self,x):
        if self._root == None:
            self._root=BST._Node(None,None,None,x)
        else:
            self._rec_insert(self._root,x)
    def _rec_insert(self,n,x):
```

```
    if x<n._data:
        if n._left == None:
            n._left=BST._Node(n,None,None,x)
        else:
            self._rec_insert(n._left,x)
    else:
        if n._right == None:
            n._right=BST._Node(n,None,None,x)
        else:
            self._rec_insert(n._right,x)
def search_le(self,x):
    if self._root==None:
        return None
    else:
        return self._rec_search_le(self._root,x)
def _rec_search_le(self,n,x):
    if x<n._data:
        if n._left:
            return self._rec_search_le(n._left,x)
        else:
            return None
    else:
        if n._right:
            rv=self._rec_search_le(n._right,x)
            if rv:
                return rv
            else:
                return n._data
        else:
            return n._data
def delete(self,x):
    n=self._root
    while n and n._data != x:
        if x < n._data:
            n=n._left
        else:
            n=n._right
    if not n:
        raise KeyError("Attempt to delete item not in BST: "+str(x))
    if n._left and n._right:
        replace=n._right
        while replace._left:
            replace=replace._left
        n._data=replace._data
        n=replace
    child = n._left if n._left else n._right
    if child:
        child._parent=n._parent
    if n is self._root:
        self._root=child
    elif n==n._parent._left:
        n._parent._left=child
    else:
        n._parent._right=child
```

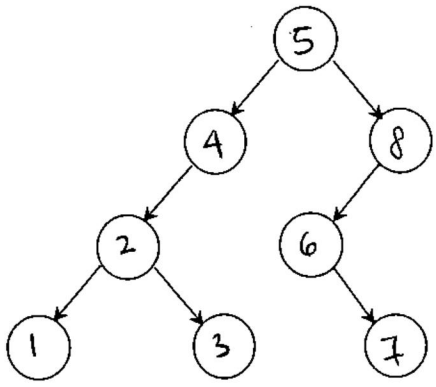
Linked binary tree code:

```
class Tree:
    class _Node:
        __slots__ = '_element', '_parent', '_left', '_right'
        def __init__(self, element, parent=None, left=None, right=None):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right
            self._textBox = TextBox(str(element))
    class Position:
        def __ne__(self, other):
            return not (self == other)
        def __init__(self, container, node):
            self._container = container
            self._node = node
        def element(self):
            return self._node._element
        def __eq__(self, other):
            return type(other) is type(self) and other._node is self._node
    def is_root(self, p):
        return self.root() == p
    def is_leaf(self, p):
        return self.num_children(p) == 0
    def is_empty(self):
        return self.root()== None
    def __iter__(self):
        for p in self.positions():
            yield p.element()
    def positions(self):
        return self.preorder()
    def preorder(self):
        #Code omitted
    def postorder(self):
        if not self.is_empty():
            for p in self._subtree_postorder(self.root()):
                yield p
    def _subtree_postorder(self, p):
        for c in self.children(p):
            for other in self._subtree_postorder(c):
                yield other
        yield p
    def breadthfirst(self):
        #Code omitted
    def _validate(self, p):
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
```

```
            raise ValueError('p does not belong to this container')
        if p._node._parent is p._node:
            raise ValueError('p is no longer valid')
        return p._node
    def _make_position(self, node):
        return self.Position(self, node) if node is not None else None
    def __init__(self):
        self._root = None
    def root(self):
        return self._make_position(self._root)
    def parent(self, p):
        node = self._validate(p)
        return self._make_position(node._parent)
    def left(self, p):
        node = self._validate(p)
        return self._make_position(node._left)
    def right(self, p):
        node = self._validate(p)
        return self._make_position(node._right)
    def num_children(self, p):
        node = self._validate(p)
        count = 0
        if node._left is not None:
            count += 1
        if node._right is not None:
            count += 1
        return count
    def sibling(self, p):
        parent = self.parent(p)
        if parent is None:
            return None
        else:
            if p == self.left(parent):
                return self.right(parent)
            else:
                return self.left(parent)
    def children(self, p):
        if self.left(p) is not None:
            yield self.left(p)
        if self.right(p) is not None:
            yield self.right(p)
    def add_root(self, e):
        if self._root is not None:
            raise ValueError('Root exists')
        self._root = self._Node(e)
        return self._make_position(self._root)
    def add_left(self, p, e):
        node = self._validate(p)
        if node._left is not None:
            raise ValueError('Left child exists')
        node._left = self._Node(e, node)
        return self._make_position(node._left)
    def add_right(self, p, e):
        node = self._validate(p)
        if node._right is not None:
            raise ValueError('Right child exists')
        node._right = self._Node(e, node)
        return self._make_position(node._right)
    def replace(self, p, e):
        node = self._validate(p)
        old = node._element
        node._element = e
        return old
    def delete(self, p):
        node = self._validate(p)
        if self.num_children(p) == 2:
            raise ValueError('Position has two children')
        child = node._left if node._left else node._right
        if child is not None:
            child._parent = node._parent
        if node is self._root:
            self._root = child
        else:
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        node._parent = node
        return node._element
    def attach(self, p, t1, t2):
        node = self._validate(p)
        if not self.is_leaf(p):
            raise ValueError('position must be leaf')
        if not type(self) is type(t1) is type(t2):
            raise TypeError('Tree types must match')
        if not t1.is_empty():
            t1._root._parent = node
            node._left = t1._root
            t1._root = None
        if not t2.is_empty():
            t2._root._parent = node
            node._right = t2._root
            t2._root = None
```

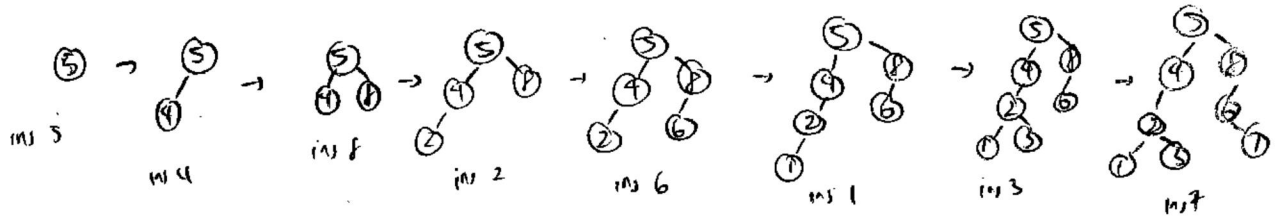
1. BST

(i) [5 Points] 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

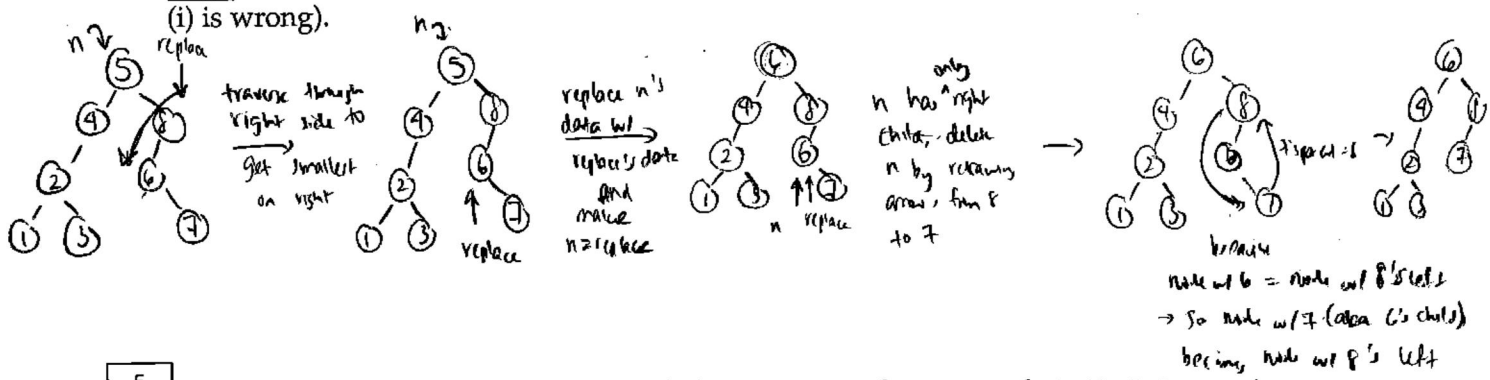


(ii) [5 Points] Suppose you wanted to create the BST you just drew by inserting the numbers 1-8. What order would you insert them?

insert in 5, 4, 8, 2, 6, 1, 3, 7

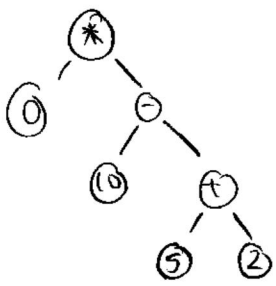


(iii) [5 Points] Draw what would happen to the BST in (i) if you deleted 5. (No credit if your answer to (i) is wrong).



2. [5 Points] Draw the arithmetic tree that has the following postorder traversal: 0 10 5 2 + - *

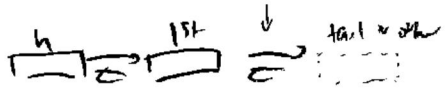
postorder = left right, not



0 * (10 - (5+2))

Check traversal:

0 10 5 2 + - *



3. [20 Points] Implement `add_second(self, data)`, a new method in the class `PList`. When called, it adds `data` to the list in the second position, meaning it would be the second item produced by iterating over the `PList`.

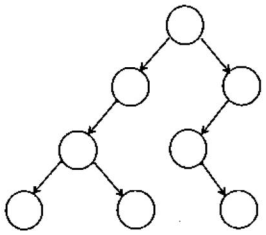
You are not allowed to use any of the other methods of the class `PList`. You can and should use the instance variables such as `_head _tail _next _prev` and `_data`. You can and should use `_Node`'s constructor. You can assume the list has a `len` of at least 1.

```
def add_second(self, data):
    first = self._head._next
    secondnext = first._next
    newnode = self._Node(data, first, secondnext)
    first._next = newnode
    secondnext._prev = newnode
    self._size += 1
```

4. [20 Points] Give a recursive implementation of the following method to be added to the linked binary tree class:

`def subtree_children_dist(self, p)}:`

When given p, a position, it will return a list of length 3, which will contain [number of leaves, number of nodes with one child, number of nodes with two children] in p's subtree.

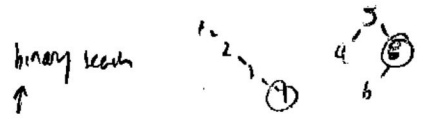


For example `print(T.subtree_children_dist(T.root())` on the above tree will print [3,3,2] as there are 3 leaves, 3 nodes with a single child, and 2 nodes with 2 children).

Note: No recursion, no credit.

leaf = no child

```
def subtree_children_dist(self, p, lst=None):
    if lst == None:
        lst = [0, 0, 0]
    if self.is_leaf(p):
        lst[0] += 1
    elif self.left(p) and self.right(p):
        lst[2] += 1
        self.subtree_children_dist(self.left(p), lst)
        self.subtree_children_dist(self.right(p), lst)
    elif self.left(p):
        lst[1] += 1
        self.subtree_children_dist(self.left(p), lst)
    else:
        lst[1] += 1
        self.subtree_children_dist(self.right(p), lst)
    return lst
```



5. [20 Points] Write a method `largest` to be added to the BST class that returns the largest item stored in the BST.

```
def largest (self):
    if self._root == None:
        return None
    node = self._root
    while node._right:
        node = node._right
    return node._data
```

6. [20 Points] You are to code a class called `boostQueue`. A `boostQueue` is like a regular queue, but it also supports a special boost operation, that moves the element currently at the back of the queue a specified number of steps forward.

The methods of the `boostQueue` that you are to code are:

- `__init__(self)`: creates an empty `BoostQueue` object.
- `__len__(self)`: returns the number of elements in the queue
- `is_empty(self)`: returns True if and only if the queue is empty
- `enqueue(self,elem)`: adds elem to the back of the queue. No return value.
- `dequeue(self)`: removes and returns the element at the front of the queue. If the queue is empty an exception is raised.
- `first(self)`: returns the element in the front of the queue without removing it from the queue. If the queue is empty an exception is raised.
- `boost(self,k)`: moves the element from the back of the queue k steps forward. If the queue is empty an exception is raised. If k is too big (greater or equal to the number of elements in the queue) the last element will become the first. No return value.

For example, your implementation should provide the following behavior:

```
>>> boost_q = BoostQueue()
>>> boost_q.enqueue(1)
>>> boost_q.enqueue(2)
>>> boost_q.enqueue(3)
>>> boost_q.enqueue(4)
>>> boost_q.boost(2)
>>> boost_q.dequeue()
1
>>> boost_q.dequeue()
4
>>> boost_q.dequeue()
2
>>> boost_q.dequeue()
3
```

Implementation requirements:

(i) BoostQueue objects may only use single instance of a `PList` as a instance variable. If you do anything else you will get no credit.

(ii) No inheritance!

(iii) All queue operations should run in worst case $O(1)$ time, except for the `boost(k)` operation, that should run in worst case $O(k)$ time.

Answer question 6 here. Make sure to include code for all the methods requested.

```
class BoostQueue():

    def __init__(self):
        self._plist = PList()

    def __len__(self):
        return len(self._plist)

    def is_empty():
        return len(self) == 0

    def first(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self._plist.first().data()

    def enqueue(self, elem):
        self._plist.add_last(elem)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self._plist.delete(self.first())

def boost(self, k):
        if self.is_empty():
            raise IndexError("Queue is empty")
        length = len(self)
        x = self._plist.delete(self.last())
        if k >= length:
            self._plist.add_first(x)
        else:
            p = self._plist.first()
            count = 1
            while count < k:
                p = self._plist.after(p)
                count += 1
            self.add_before(p, x)
```

Extra space. If you use it, write "answer on page 11" in the relevant question.

Extra space. If you use it, write "answer on page 12" in the relevant question.