1. Consider the parse tree for \((9 + (5 \times 3)) / (8 - 4)\):

```
/  
+   -
  
9   8
*   4
  
*   5
  
3
```

a) Identify the following items in the above tree:
- node containing "*"
- edge from node containing "-" to node containing "8"
- root node
- children of the node containing "+
- parent of the node containing "3"
- siblings of the node containing "*"
- leaf nodes of the tree
- subtree who's root is node contains "+
- path from node containing "+" to node containing "5"
- branch from root node to "3"

b) Mark the levels of the tree (level is the number of edges on the path from the root)

c) What is the height (max. level) of the tree?

2. In Python an easy way to implement a tree is as a list of lists where a tree look like:

```
[ "node value", remaining items are subtrees for the node each implemented as a list of lists]
```

Complete the list-of-lists representation look like for the above parse tree:

```
["/", "+", 
  
, 
  [, 
  [, ]], 
  [, ]], 
  
, [, ]]
```

3. Consider a "linked" representations of a BinaryTree. For the expression \(((4 + 5) \times 7)\), the binary tree would be:

```python
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

```
key  
leftChild rightChild
  
key  
leftChild rightChild
  
key  
leftChild rightChild
  
key  
leftChild rightChild
```

```
key  
leftChild rightChild
  
key  
leftChild rightChild
  
key  
leftChild rightChild
  
key  
leftChild rightChild
```
a) Fix the insertLeft and insertRight code:
(Listing 6.6 and 6.7 are wrong in the text on pp. 242-3)

def inorder(tree):
    if tree == None:
        return
    inorder(tree.getLeftChild())
    print(tree.getRootVal())
    inorder(tree.getRightChild())

def printexp(tree):
    if tree.leftChild:
        print('(', end=' ')
    printexp(tree.getLeftChild())
    print(tree.getRootVal(), end=' ')
    if tree.rightChild:
        printexp(tree.getRightChild())
    print(')', end=' ')

def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()

def printexp(self):
    if self.leftChild:
        print('(', end=' ')
    self.leftChild.printexp()
    print(self.key, end=' ')
    if self.rightChild:
        self.rightChild.printexp()
    print(')', end=' ')

def postordereval(self):
    operas = {'+':operator.add, '-':operator.sub,
              '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if self.leftChild:
        res1 = self.leftChild.postordereval()
    if self.rightChild:
        res2 = self.rightChild.postordereval()
    if res1 and res2:
        return operas[self.key](res1, res2)
    else:
        return self.key

Some corresponding external (non-class) functions:
b) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets printed by a call to:

<table>
<thead>
<tr>
<th><code>myTree.inorder()</code></th>
<th><code>myTree.preorder()</code></th>
<th><code>myTree.postorder()</code></th>
<th><code>inorder(myTree)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

c) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets printed by a call to `myTree.printexp()`?

d) If `myTree` is the `BinaryTree` object for the expression: `(4 + 5) * 7`, what gets returned by a call to `myTree.postordereval()`?

e) Write the `height` method for the `BinaryTree` class.

4. Consider the Binary Search Tree (BST). For each node, all values in the left-subtree are < the node and all values in the right-subtree are > the node.

```
      50
     /   \
   30    70
  /   \
 9     58
   / \
18   32
  /   |
34   47
   |
 80
```

a. Starting at the root, how would you find the node containing “32”?

b. Starting at the root, when would you discover that “65” is not in the BST?

c. Starting at the root, where would be the “easiest” place to add “65”?

d. Where would we add “5” and “33”?

e. What is the order of node processing in a preorder traversal of the above BST?

f. What is the order of node processing in a postorder traversal of the above BST?

g. What is the order of node processing in an inorder traversal of the above BST?
1. Consider the partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def __iter__(self):
        if self:
            if self.hasLeftChild():
                for elem in self.leftChild:
                    yield elem
            yield self.key
            if self.hasRightChild():
                for elem in self.rightChild:
                    yield elem
```

A BinarySearchTree object

- **size**: Represents the number of nodes in the tree.
- **root**: The root node of the tree.
- **TreeNode objects**: Represents the nodes in the tree.

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root, __iter__()

    def __str__(self):
        """Returns a string representation of the tree rotated 90 degrees counter-clockwise""

        def strHelper(root, level):
            resultStr = ""
            if root:
                resultStr += strHelper(root.rightChild, level+1)
                resultStr += "| " * level
                resultStr += str(root.key) + "\n"
                resultStr += strHelper(root.leftChild, level+1)
            return resultStr

        return strHelper(self.root, 0)
```

a) How do the BinarySearchTree __iter__ and __str__ methods work?
More partial TreeNode class and partial BinarySearchTree class.

class BinarySearchTree:
    ...
    def _contains_(self, key):
        if self._get(key, self.root):
            return True
        else:
            return False

def get(self, key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self, key, currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key, currentNode.leftChild)
    else:
        return self._get(key, currentNode.rightChild)

def __getitem__(self, key):
    return self.get(key)

def __setitem__(self, k, v):
    self.put(k, v)

def put(self, key, val):
    if self.root:
        self._put(key, val, self.root)
    else:
        self.root = TreeNode(key, val)
        self.size = self.size + 1

def _put(self, key, val, currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            ...
    else:
        ...

def A BinarySearchTree object:
    size
    root
    TreeNode objects

b) The _get method is the "work horse" of BST search. It recursively walks currentNode down the tree until it finds key or becomes None. In English, what are the base and recursive cases?

c) What is the put method doing?

d) Complete the recursive _put method.

e) Draw the "shape" of the BST after put s of: 50, 60, 30, 70, 90, 40, 65

f) If n items are in the BST, what is put's: Best-case O( )? Worst-case O( )? Average-case O( )?
2. More partial TreeNode class and partial BinarySearchTree class.

```python
class BinarySearchTree:
    ... 
    def delete(self, key):
        if self.size > 0:
            nodeToRemove = self._get(key, self.root)
            if nodeToRemove:
                self.remove(nodeToRemove)
                self.size = self.size - 1
            else:
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')
    
    def _delitem_(self, key):
        self.delete(key)
    
    def remove(self, current_node):
        if current_node.isLeaf():  # leaf
            if current_node == current_node.parent.leftChild:
                current_node.parent.leftChild = None
            else:
                current_node.parent.rightChild = None
            if current_node.hasBothChildren():  # interior
                succ = current_node.findSuccessor()
                succ.spliceOut()
                current_node.key = succ.key
                current_node.payload = succ.payload
        else:  # this node has one child
            if current_node.isLeftChild():
                current_node.leftChild.parent = current_node.parent
                current_node.parent.leftChild = current_node.leftChild
            elif current_node.isRightChild():
                current_node.leftChild.parent = current_node.parent
                current_node.parent.rightChild = current_node.leftChild
            else:
                current_node.replaceNodeData(current_node.leftChild.key,
                                              current_node.leftChild.payload,
                                              current_node.leftChild.leftChild,
                                              current_node.leftChild.rightChild)
            if current_node.isLeftChild():
                current_node.rightChild.parent = current_node.parent
                current_node.parent.leftChild = current_node.rightChild
            elif current_node.isRightChild():
                current_node.rightChild.parent = current_node.parent
                current_node.parent.rightChild = current_node.rightChild
            else:
                current_node.replaceNodeData(current_node.rightChild.key,
                                              currentNode.rightChild.payload,
                                              currentNode.rightChild.leftChild,
                                              currentNode.rightChild.rightChild)
```

a) Update picture where we delete a leaf.

![Tree Structure](image1)

b) Where in the code is each handled?

c) Draw all pictures deleting all nodes with one child.
3. Yet even more partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    ...
    def findSuccessor(self):
        succ = None
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ
    
def findMin(self):
        current = self
        while current.hasLeftChild():
            current = current.leftChild
        return current
    
def spliceOut(self):
        if self.isLeaf():
            if self.isLeftChild():
                self.parent.leftChild = None
            else:
                self.parent.rightChild = None
        elif self.hasAnyChildren():
            if self.hasLeftChild():
                if self.isLeftChild():
                    self.parent.leftChild = self.leftChild
                else:
                    self.parent.rightChild = self.leftChild
                    self.leftChild.parent = self.parent
            else:
                if self.isLeftChild():
                    self.parent.leftChild = self.rightChild
                else:
                    self.parent.rightChild = self.rightChild
                    self.rightChild.parent = self.parent
```
Objectives: You will gain experience BST performance and implementation

To start the lab: Download and unzip the lab9.zip file from eLearning.

Part A: Consider the Binary Search Tree (BST) below. For each node in a BST, all values in the left-subtree are < the node and all values in the right-subtree are > the node.

![](image)

(a) Review section 6.5.2 on Tree Traversals to determine the order nodes are processed in each tree traversal.

- What is the order of node processing in a preorder traversal of the above BST?
- What is the order of node processing in an inorder traversal of the above BST?
- What is the order of node processing in a postorder traversal of the above BST?

(b) Starting with an empty BST, what would be the shape of the BST after put's for keys: 50, 60, 30, 70, 90, 40, and 65?
Part B: Run the `timeBinarySearchTree.py` program that:
- creates a list, `evenList`, that holds 3,000 sorted, even values (e.g., `evenList = [0, 2, 4, 6, 8, ..., 5996, 5998]`)
- puts (adds) all the `evenList` items into an initially empty `BinarySearchTree` object, `bst`
- times the searches (in) `bst` for target values 0, 1, 2, 3, 4, ..., 5998, 5999 so half of the searches are successful and half are unsuccessful
  a) How long does it take to search for target values of 0, 1, 2, 3, 4, ..., 5998, 5999?
  b) Explain why these searches take so long. (Hint: consider the shape of the `BinarySearchTree bst`)
  
  c) Uncomment the "`shuffle(evenList)`" which randomizes the items in `evenList` before adding them to the `BinarySearchTree bst`. Now how long does it take to search for target values from 0, 1, 2, 3, 4, ..., 5998, 5999?
  d) Explain why these searches take so little time.

  e) What is the search time with the `timeOpenAddrHashDictSearch.py` program? Why is it faster?

Part C: a) Complete the recursive height method in the `BinarySearchTree` class. Model it after the postorder traversal, since the height of the whole BST can be determined after you know the height of the left-subtree and height of the right-subtree. For example if the left-subtree has a height of say 8 and the right-subtree has a height of 5, then the overall height including the root is 9 (i.e., one more than the tallest subtree's height). For the base case of the recursion, if we define the empty subtree's height to be -1 (i.e., `subtreeRoot` points to `None` since it has no `TreeNode` to point at), then the recursive definition still works for a leaf node which should have a height of 0.
  
  b) Uncomment the call to the height method at the end of the `timeBinarySearchTree.py` program. What is the height of `bst` if we are shuffling the `evenList`?

  c) What would be the shortest possible height for a binary tree with 3,000 items?

After you have answers and correct code for all parts of the lab, submit a `lab9.zip` containing your code on eLearning. If you do not get done today, then submit it by next week's lab period.

Remember to save your lab9 files for later usage on homework assignments!
1. Consider the Binary Search Tree (BST):

   ![BST Diagram]

   a. What would need to be done to delete 32 from the BST?

   b. What would need to be done to delete 9 from the BST?

   c. What would be the result of deleting 50 from the BST?  Hint: One technique when programming is to convert a hard problem into a simpler problem. Deleting a BST node that contains two children is a hard problem. Since we know how to delete a BST node with none or one child, we can convert “deleting a node with two children” problem into a simpler problem by overwriting 50 with another node’s value. Which nodes can be used to overwrite 50 and still maintain the BST ordering?

   d. Which node would the TreeNode's findSuccessor method return for succ if we are deleting 50 from the BST?

2. When the findSuccessor method is called how many children does the self node have?

3. How could we improve the findSuccessor method?

4. When the spliceOut method is called from remove how many children could the self node have?

5. How could we improve the spliceOut method?
6. The shape of a BST depends on the order in which values are added (and deleted).
a) What would be the shape of a BST if we start with an empty BST and insert the sequence of values:
   70, 90, 80, 5, 30, 110, 95, 40, 100

b) If a BST contains \( n \) nodes and we start searching at the root, what would be the worst-case big-oh \( O() \) notation for a successful search? (Draw the shape of the BST leading to the worst-case search)

7. We could store a BST in an array like we did for a binary heap, e.g. root at index 1, node at index \( i \) having left child at index \( 2 \times i \), and right child at index \( 2 \times i + 1 \).
a) Draw the above BST (after inserting: 70, 90, 80, 5, 30, 110, 95, 40, 100) stored in an array (leave blank unused slots)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
</table>

b) What would be the worst-case storage needed for a BST with \( n \) nodes?

8. a) If a BST contains \( n \) nodes, draw the shape of the BST leading to best, successful search in the worst case.

b) What is the worst-case big-oh \( O() \) notation for a successful search in this “best” shape BST?
This assignment has several parts -- a comparison of dictionary implementations (from labs 7, 9, 10) and a concordance-production application using the dictionary ADT/interface. A Webster's dictionary definition of concordance is: "an alphabetical list of the main words in a work." In addition to the main words, I want you to keep track of all the line numbers where these main words occur. (e.g. like an index at the back of a textbook, but with line #s instead of page #s)

**WORD & LINE CONCORDANCE APPLICATION**

The goal of this assignment is to process a textual, data file (WarAndPeace.txt) to generate a word concordance with line numbers for each main word. A dictionary ADT is perfect to store the word concordance with the word being the dictionary key and a Python list of the word's line numbers being the associated value with the key. Since the concordance should only keep track of the "main" words, there will actually be a second stop-words file (stop_words.txt). The stop-words file will contain a list of stop words (e.g., "a", "the", etc.) -- these words will not be included in the concordance even if they do appear in the data file. Sample input/output files might be:

<table>
<thead>
<tr>
<th>Sample stop words small.txt file</th>
<th>Sample hw6small.txt file</th>
<th>Sample output file</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bigger: 4</td>
<td>bigger: 4</td>
</tr>
<tr>
<td>about</td>
<td>data: 14</td>
<td>data: 14</td>
</tr>
<tr>
<td>be</td>
<td>file: 14</td>
<td>file: 14</td>
</tr>
<tr>
<td>be</td>
<td>much: 4</td>
<td>much: 4</td>
</tr>
<tr>
<td>can</td>
<td>processed: 2</td>
<td>processed: 2</td>
</tr>
<tr>
<td>do</td>
<td>program: 2</td>
<td>program: 2</td>
</tr>
<tr>
<td>do</td>
<td>real: 4</td>
<td>real: 4</td>
</tr>
<tr>
<td>i</td>
<td>sample: 1</td>
<td>sample: 1</td>
</tr>
<tr>
<td>in</td>
<td>text: 1</td>
<td>text: 1</td>
</tr>
<tr>
<td>is</td>
<td>word-concordance: 2</td>
<td>word-concordance: 2</td>
</tr>
<tr>
<td>is</td>
<td>your: 2</td>
<td>your: 2</td>
</tr>
<tr>
<td>of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>this</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>was</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1) Words are defined to be sequences of letters separated by any white space, and stripped of non-letters (e.g., punctuation, parentheses, dashes, double quotes, digits, etc.)

2) There is to be no distinction made between upper and lower case letters. (e.g., "CAT" is the same word as "cat")

3) Blank lines are to be counted in the line numbering. (e.g., line 3 above is blank)

The general algorithm for the word-concordance program is:

1) Read the stop_words_small.txt (or stop_words.txt) file into a dictionary (use the same type of dictionary that you're timing) containing only stop words, called stopWordDict. (WARNING: Strip the newline ("\n") character from the end of the stop word before adding it to stopWordDict, e.g., word = word.strip()). The stopword is the key in this dictionary and you can just use None as its value.

2) Process the hw6small.txt (or WarAndPeace.txt) file one line at a time to build the word-concordance dictionary (called wordConcordanceDict) containing "main" words for the keys with a Python list of their associated line numbers as their values. The main loop is something like:

   ```
   lineCounter = 1
   for each line in the data file do
       processLine( lineCounter, line, wordConcordanceDict, ...)  
       lineCounter += 1
   ```

3) Traverse the wordConcordanceDict alphabetically by key (by "for word in sorted(wordConcordanceDict):") to generate a text file containing the concordance words printed out in alphabetical order along with their corresponding line numbers.

The general algorithm for the processLine (lineCounter, line, wordConcordanceDict,...) function is:

   ```
   wordList = createWordList(line)  #<<< creates a Python list of words on a line 
   for each word in the wordList do
       if the word is not in the stopWordDict then
           if the word is in the wordConcordanceDict then
               look up the line-#-list value associated with the word in the wordConcordanceDict
               append the lineCounter to the end of the line-#-list
           else
               add the word with an associated [lineCounter] list value to the wordConcordanceDict
   ```
Note: I strongly suggested that the logic for reading words and assigning line numbers to them be developed and tested separately from other aspects of the program. This could be accomplished by reading a sample file and printing out the words recognized with their corresponding line numbers without any other word processing.

**DICTIONARY ADT COMPARISON**

We have 6 dictionary ADT implementations from labs 7, 9, and 10: Python dictionary, ChainingDict, OpenAddrHashDict with linear probing, OpenAddrHashDict with quadratic probing, single BST-based dictionary, and single AVL-tree-based dictionary. None of these should need to be modified since you just use their dictionary operations.

Time your word-concordance application using all six dictionary ADT implementations to complete the following table: (FYI, for WarAndPeace.txt there are about 2,700 stop words and less than 20,000 non-stop words). Both the stop words and non-stop (i.e., word-concordance) words should use the same type of dictionary (e.g., both are different ChainingDict’s). Have just one word-concordance program with 6 pairs of dictionaries with all but one pair commented out, i.e., the dictionary pair you are timing is the only pair uncommented. For example:

```python
stopWordDict = {}               # Pair for Python dictionary
wordConcordanceDict = {}        
## Other pairs commented out while timing Python dictionary version
## stopWordDict = ChainingDict(2**15)  # Pair for ChainingDict
## wordConcordanceDict = ChainingDict(2**15)
## stopWordDict = OpenAddrHashDict(2**15; linear = True) # Pair for linear probing
## wordConcordanceDict = OpenAddrHashDict(2**15, linear = True)
## stopWordDict = OpenAddrHashDict(2**15, linear = False) # Pair for quadratic probing
## wordConcordanceDict = OpenAddrHashDict(2**15, linear = False)
## stopWordDict = BinarySearchTree() # Pair for BST dictionary
## wordConcordanceDict = BinarySearchTree()
## stopWordDict = AVLTree()   # Pair for AVL-tree dictionary
## wordConcordanceDict = AVLTree()
```

Complete the following table in the hw6 Timing.docx (or .pdf) file:

<table>
<thead>
<tr>
<th>Dictionary ADT Implementation Used</th>
<th>Word-concordance Program Execution Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python dictionary</td>
<td></td>
</tr>
<tr>
<td>ChainingDict (hash table sizes 2**15 = 32768)</td>
<td></td>
</tr>
<tr>
<td>OpenAddrHashDict with linear probing (hash table sizes 2**15 = 32768)</td>
<td></td>
</tr>
<tr>
<td>OpenAddrHashDict with quadratic probing (hash table sizes 2**15 = 32768)</td>
<td></td>
</tr>
<tr>
<td>Dictionaries implemented using BSTs</td>
<td>(Warning: might take 5 - 10 minutes to finish!)</td>
</tr>
<tr>
<td>Dictionaries implemented using AVL trees</td>
<td></td>
</tr>
</tbody>
</table>

**DATA FILES** - Download and extract the hw6.zip file from eLearning at Course Content | Unit 3 | Homework #6 it contains:
- ChainingDict in the file chaining_dictionary.py and OpenAddrHashDict in the file open_addr_hash_dictionary.py
- BinarySearchTree in the file binary_search_tree.py and AVLTree in the file avl_tree.py. **NOTE:** For the AVL tree, you will need to complete/copy rotateRight code from lab 10
- The sample data files for testing your word-concordance program: hw6small.txt and stop_words_small.txt
- The "real" stop words are in the file stop_words.txt
- The "real" data file to be processed by your word-concordance program is in the file WarAndPeace.txt
On eLearning (Course Content | Unit #3 | Homework #6 subfolder), submit a single .zip file, hw6.zip, containing:

- all the files for all 6 dictionaries
- your word-concordance program in the file hw6.py
- the completed timing table in the hw6_Timing.docx (or .pdf) file
- the "real" stop words file stop_words.txt and "real" data file WarAndPeace.txt

(If you miss the deadline, you can still submit it without a late penalty up to noon on Wednesday November 25)

**EXTRA CREDIT POSSIBILITY:**

Implement a new `ClosedAddrUsingBSTDict` dictionary ADT that combines binary search trees (BSTs) and closed-address hashing. Recall that the `ChainingDict` class uses a hash table with UnorderedLists at each slot as shown on the left below. You are to implement a `ClosedAddrUsingBSTDict` class using a hash table with BSTs at each slot as shown on the right below:

![ChainingDict Object](image1.png)

![ClosedAddrUsingBSTDict Object](image2.png)

The hw6/closed_addr_using_bst_dictionary.py file contains a template for the `ClosedAddrUsingBSTDict` class. After you implement the `ClosedAddrUsingBSTDict` class, incorporate it into your word-concordance program and supply a timing for processing WarAndPeace.txt.
1. An AVL Tree is a special type of Binary Search Tree (BST) that it is height balanced. By height balanced I mean that the height of every node’s left and right subtrees differ by at most one. This is enough to guarantee that a AVL tree with $n$ nodes has a height no worst than $O(1.44 \log_2 n)$. Therefore, insertions, deletions, and search are worst case $O( \log_2 n )$. An example of an AVL tree with integer keys is shown below. The height of each node is shown.

Each AVL-tree node usually stores a balance factor in addition to its key and payload. The balance factor keeps track of the relative height difference between its left and right subtrees, i.e., height(left subtree) - height(right subtree).

a) Label each node in the above AVL tree with one of the following balance factors:
   - 0 if its left and right subtrees are the same height
   - 1 if its left subtree is one taller than its right subtree
   - -1 if its right subtree is one taller than its left subtree

b) We start a put operation by adding the new item into the AVL as a leaf just like we did for Binary Search Trees (BSTs). Add the key 90 to the above tree.

c) Identify the node “closest up the tree” from the inserted node (90) that no longer satisfies the height-balanced property of an AVL tree. This node is called the pivot node. Label the pivot node above.

d) Consider the subtree whose root is the pivot node. How could we rearrange this subtree to restore the AVL height balanced property? (Draw the rearranged tree below)
2. Typically, the addition of a new key into an AVL requires the following steps:
   - compare the new key with the current tree node's key (as we did in the _put function called by the put method in the BST) to determine whether to recursively add the new key into the left or right subtree
   - add the new key as a leaf as the base case(s) to the recursion
   - recursively (updateBalance method) adjust the balance factors of the nodes on the search path from the new node back up toward the root of the tree. If we encounter a pivot node (as in question (c) above) we perform one or two "rotations" to restore the AVL tree's height-balanced property.

For example, consider the previous example of adding 90 to the AVL tree. Before the addition, the pivot node (60) was already -1 ("tall right" - right subtree had a height one greater than its left subtree). After inserting 90, the pivot's right subtree had a height 2 more than its left subtree (balance factor -2) which violates the AVL tree's height-balanced property. This problem is handled with a left rotation about the pivot as shown in the following generalized diagram:

Before the addition:

After the addition, but before rotation:

Recursive updateBalance method finds the pivot and calls the rebalance method to perform proper rotation(s)

(D’s balance factor was already adjusted before the pivot is found by the recursive updateBalance method which moves toward the root)

After left rotation at pivot:

a) Assuming the same initial AVL tree (upper, left-hand of above diagram) if the new node would have increased the height of T_C (instead of T_E), would a left rotation about the node B have rebalanced the AVL tree?
Before the addition, if the pivot node was already -1 (tall right) and if the new node is inserted into the left subtree of the pivot node's right child, then we must do two rotations to restore the AVL-tree's height-balance property.

Before the addition:                           After the addition, but before first rotation:

```
        from parent
          
          
        B        B
          -1       -2
            /
          /  
          F  D      F
            /
          /  
          T  T
          A  B
              /
             /
             T  T
             C  E
                 /
                 /
                 T  T
                 G  F
```

Recursive updateBalance finds the pivot and calls rebalance method to perform rotation(s).

D's & F's balance factors have already been adjusted before the pivot was found.

```
        from parent
          
          
        B        F
          -1       1
            /
          /  
          F  D
            /
          /  
          T  T
          A  E
              /
              /
              T  T
              B  G
```

After the left rotation at pivot and balance factors adjusted correctly:

```
        from parent
          
          
        D        B
          0       -1
            /
          /  
          D  F
            /
          /  
          T  T
          B  A
              /
              /
              T  T
              C  E
```

After right rotation at F, but before left rotation at pivot:

```
        from parent
          
          
        B        D
          -2       -2
            /
          /  
          F  D
            /
          /  
          T  T
          F  T
              /
              /
              T  T
              G  E
```

b) Suppose that the new node was added in T_C instead of T_E, then the same two rotations would restore the AVL-tree's height-balance property. However, what should the balance factors of nodes B, D, and F be after the rotations?
Consider the AVLTreeNode class that inherits and extends the TreeNode class to include balance factors.

```python
from tree_node import TreeNode

class AVLTreeNode(TreeNode):
    def __init__(self, key, val, left=None, right=None, parent=None, balanceFactor=0):
        TreeNode.__init__(self, key, val, left, right, parent)
        self.balanceFactor = balanceFactor
```

Now let's consider the partial AVLTree class code that inherits from the BinarySearchTree class:

```python
from avl_tree_node import AVLTreeNode
from binary_search_tree import BinarySearchTree

class AVLTree(BinarySearchTree):
    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = AVLTreeNode(key, val)
            self.size = self.size + 1

    def _put(self, key, val, currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key, val, currentNode.leftChild)
            else:
                currentNode.leftChild = AVLTreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.leftChild)
        elif key > currentNode.key:
            if currentNode.hasRightChild():
                self._put(key, val, currentNode.rightChild)
            else:
                currentNode.rightChild = AVLTreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.rightChild)
        else:
            currentNode.payload = val
            self.size += 1

    def updateBalance(self, node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            return node.rebalance(node)
        if node.parent != None:
            if node.isLeftChild():
                node.parent.balanceFactor += 1
            elif node.isRightChild():
                node.parent.balanceFactor -= 1
            if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)

    def rotateLeft(self, self, rotRoot):
        newRoot = rotRoot.rightChild
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.leftChild != None:
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        rotRoot.balanceFactor = rotRoot.balanceFactor + 1 - min(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor + 1 + max(rotRoot.balanceFactor, 0)

    def rebalance(self, node):
        if node.balanceFactor > 0:
            if node.rightChild.balanceFactor > 0:
                self.rotateRight(node.rightChild)
                self.rotateLeft(node)
            else:
                self.rotateLeft(node)
        elif node.balanceFactor < 0:
            if node.leftChild.balanceFactor < 0:
                self.rotateLeft(node.leftChild)
                self.rotateRight(node)
            else:
                self.rotateRight(node)
```

**NOTE:** You will complete rotateRight in Lab.
c) Trace the code for myAVL.put(90, None) by updating the below diagram:

Consider balance factor formulas for rotateLeft. We know:
newBal(B) = h_A - h_C and oldBal(B) = h_A - (1 + max(h_C, h_E))
newBal(D) = 1 + max(h_A, h_C) - h_E and oldBal(D) = h_C - h_E

Before left rotation:

After left rotation at pivot:

rotRoot

D

B

rotRoot

D

B

Consider: newBal(B) - oldBal(B)
newBal(B) - oldBal(B) = (h_A - h_C) - [h_A - (1 + max(h_C, h_E))]
newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))
newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C
newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C
newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))
newBal(B) = oldBal(B) + 1 - min(0, oldBal(D)), so
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
min(newRoot.balanceFactor, 0)

min(-x, -y) = -y
-max(x, y) = -y
max(-x, -y) = -x

min(-x, -y) = -y
-max(x, y) = -y
max(-x, -y) = -x

Consider: newBal(B) - oldBal(B)
newBal(B) - oldBal(B) = (h_A - h_C) - [h_A - (1 + max(h_C, h_E))]
newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))
newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C
newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C
newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))
newBal(B) = oldBal(B) + 1 - min(0, oldBal(D)), so
rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
min(newRoot.balanceFactor, 0)
3. Complete the below figure which is a “mirror image” to the figure on page 2, i.e., inserting into the pivot’s left child’s left subtree. Include correct balance factors after the rotation.

Before the insertion:

After the insertion, but before rotation:

After right rotation at pivot:

b) Complete the below figure which is a “mirror image” to the figure on page 3, i.e., inserting into the pivot’s left child’s right subtree. Include correct balance factors after the rotation.

Before the insertion:

After the insertion, but before first rotation:

After the right rotation at pivot and balance factors adjusted correctly:

After left rotation at B, but before right rotation at pivot:
Objectives: You will gain experience with AVL put implementation

To start the lab: Download and unzip the lab10.zip file eLearning.

Part A: Starting with an empty AVL tree, what would be the shape of the AVL tree be after put's for keys: 90, 60, 50, 55, 40, and 53? (Show all necessary rotation(s) needed for each put.)

Part B: In lecture 23 we discussed the AVL tree rotateLeft method. For Part B, you need to implement the rotateRight method. Start by copying the rotateLeft method code, and paste it as the starting point for rotateRight. Now, modify the pasted rotateRight code is two steps:
1) updating the "pointers" to the nodes to do the right-rotation
   • HINT: Since rotateRight is a mirror image of rotateLeft, change all the left's to right's, and all the right's to left's
2) updating the balanceFactors for the rotRoot and newRoot nodes. You will need to use math similar to lecture 23
   where were calculated values for the rotateLeft method. Use the next two pages to calculate needed
   balanceFactors for the rotateRight method. Remember the follow rules of algebra:

Algebra Review:
• a - (b - c) when removing the paretheses you get: a - (b - c) = a - b + c
• max(x, y) + c = max(x + c, y + c) should be clear from the following diagram:

Consider max(x, y) + c:

\[
\begin{align*}
\text{max}(x, y) &= y \\
0 &= x \\
x &= y \\
\text{max}(x, y) + c &= y + c
\end{align*}
\]

Consider max(x+c, y+c):

\[
\begin{align*}
\text{max}(x+c, y+c) &= y + c \\
0 &= x \\
x &= y \\
\text{max}(x+c, y+c) &= y + c
\end{align*}
\]

• \( \text{min}(x, y) + c = \text{min}(x + c, y + c) \) similarly

• \(-\text{max}(x, y) = +\text{min}(-x, -y) \) should be clear from the following diagram:

\[
\begin{align*}
\text{Clearly, min}(x, y) &= x \\
y &= -x \\
x &= -y \\
0 &= x \\
x &= y \\
\text{max}(-x, -y) &= -x \text{ and negating both sides gives:} \\
-\text{max}(-x, -y) &= -(-x) = x, \text{ so } -\text{max}(-x, -y) = \text{min}(x, y)
\end{align*}
\]

• \(-\text{min}(x, y) = +\text{max}(-x, -y) \) similarly
Calculate the needed balance factors for the rotateRight method below:

Before right rotation:

\[ \text{rotRoot} \]
\[ \text{newRoot} \]

\[ \text{D} \]
\[ \text{B} \]

\[ T_A \]
\[ \text{height} \]
\[ h_A \]
\[ T_C \]
\[ \text{height} \]
\[ h_C \]
\[ T_E \]
\[ \text{height} \]
\[ h_E \]

Rotate Right at Pivot

After right rotation at pivot:

\[ \text{newRoot} \]

\[ \text{B} \]
\[ \text{D} \]

\[ T_A \]
\[ \text{height} \]
\[ h_A \]
\[ T_C \]
\[ \text{height} \]
\[ h_C \]
\[ T_E \]
\[ \text{height} \]
\[ h_E \]

Consider the balance factor formulas for rotateRight. We know from the above diagram:

\[ \text{oldBal}(B) = h_A - h_C \text{ and} \]
\[ \text{oldBal}(D) = (1 + \text{max}(h_A, h_C)) - h_E \text{ and} \]

\[ \text{newBal}(B) = h_A - (1 + \text{max}(h_C, h_E)) \text{ and} \]
\[ \text{newBal}(D) = h_C - h_E \text{ and} \]

To determine \( \text{newBal}(D) \), consider:

\[ \text{newBal}(D) - \text{oldBal}(D) = \]

\[ \text{newBal}(D) = \]
Consider the balance factor formulas for `rotateRight`. We know from the above diagram:

\[
\begin{align*}
\text{oldBal}(B) &= h_A - h_C \\
\text{oldBal}(D) &= (1 + \max(h_A, h_C)) - h_E
\end{align*}
\]

To determine `newBal(B)`, consider:

\[
\text{newBal}(B) - \text{oldBal}(B) =
\]

After completing your implementation of `rotateRight`, test your code by running the `avl_tree.py` program. Once you think it is working, run the `timeAVLTree.py` program. The height of AVL tree after adding in sorted order should be 13, and the height of AVL tree after adding in shuffled order should be about 15.

After you have answers and correct code for all parts of the lab, submit a `lab10.zip` containing your code on eLearning. If you do not get done today, then submit it by next week’s lab period.

Remember to save your `lab10` files for later usage on homework assignments!
1. BST, AVL trees, and hash tables can all be used to implement a dictionary ADT.

<table>
<thead>
<tr>
<th>Dictionary Successful Search Comparisons with 10,000 integer items (Time in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items added in sorted order</td>
</tr>
<tr>
<td>BST</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Total add/put time</td>
</tr>
<tr>
<td>Total search time</td>
</tr>
<tr>
<td>Height of resulting tree</td>
</tr>
</tbody>
</table>

a) The puts of these 10,000 randomly ordered items into the BST took 0.119 seconds and 0.195 seconds into the AVL tree. Why did the BST puts take less time even though the final height was 30 vs. a final AVL tree height of 15?

b) With a very, very poor hash function or very, very bad choice of keys, then all keys could hash to the same home address.

- What would be the worst-case big-oh of open-address hashing with quadratic probing?

- What would be the worst-case big-oh of chaining using a linked list at each home address (i.e., ChainingDict)?

- What would be the worst-case big-oh of chaining using an AVL tree at each home address?

2. The data structures we have discussed so far are all in-memory, i.e., data is stored in main/RAM memory. Data can also be stored on secondary storage in a file (e.g., movieData.txt file). Currently, most secondary storage consists of hard-disks.

a) Complete the following table comparing main/RAM memory vs. hard-disk:

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Main/RAM memory</th>
<th>Hard-disk Drive</th>
<th>Solid-State Drive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size on a typical desktop computer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average access time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b) Which criterion seems to be the most important difference between the main and secondary memories?
Logical View of Disk as Linear Collection of Blocks

8-15 are on surface 1 (on the bottom of the disk)

(track #, surface #, sector #) to Linear block # mapping

<table>
<thead>
<tr>
<th>(0,0,0)</th>
<th>(0,0,1)</th>
<th>(0,0,2)</th>
</tr>
</thead>
</table>

Bits of linear block #: track # | surface # | sector #

3. Disk-access time = (seek time) + (rotational delay) + (data transfer time). How is each component of the disk-access time effected by increasing the disk's RPMs (revolutions per minute)?

b) If we want fast access to a collection of sectors, where can we place them to minimize seek time and rotational delay?
User Program - HLL programming language makes system calls to OS to:
1. open file - establish a link between file variable and file for either reading, writing, or both
2. access file - read or write one piece of data at a time (e.g., char, record, etc.)
3. close file - flush changes to disk

Operating System - manages and controls access to secondary storage through its file system which contains information about every file: location on disk, ownership and security/protection
- OS views disk as a linear sequence of blocks (block 0, block 1, etc.), but assumes closeness in block # means close with respect to access time.
- OS buffers some blocks in memory to improve efficiency
- OS maintains free disk space "list"

Secondary Storage - accepts R/W requests from OS for block# and maps block# to internals physical address Device (e.g., (track #, surface #, sector #) - more complex than above picture!

Kinds of File Access:
- serial/sequential files - open at the beginning and read sequentially from beginning to end linearly
- random-access files - "seek" to any position by specifying a byte-offset from the beginning of the file
- random-access of a record by key

Implementation of Files on Disk - how blocks are allocated
4. non-contiguous - scattered across linear address space of OS and disk

File system meta-data for file

linked-list of blocks on disk

a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

5. contiguous - sequential collection of blocks from OS linear view of disk

File system meta-data for file

10 11 12 13 14 15 16 17 18

a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?
6. file descriptor blocks - list of blocks hold the address of the physical location of data blocks

- File system meta-data for file
- file descriptor block(s)
- 2nd data block in file
- 1st data block in file
- 3rd data block in file
- 0th data block in file
- pointer to next file descriptor block

a) What types of file access are supported efficiently?

b) How easy is it for the file to grow in size?

7. To implement "random-access of a record by key" in a file how might we use hashing?

8. To implement "random-access of a record by key" in a file why would an AVL tree not work well?
9. A B+ Tree is a multi-way tree (typically in the order of 100s children per node) used primarily as a file-index structure to allow fast search (as well as insertions and deletions) for a target key on disk. Two types of pages (B+ tree "nodes") exist:
- Data pages - which always appear as leaves on the same level of a B+ tree (usually a doubly-linked list too)
- Index pages - the root and other interior nodes above the data page leaves. Index nodes contain some minimum and maximum number of keys and pointers bases on the B+ tree's branching factor \( b \) and fill factor. A 50% fill factor would be the minimum for any B+ tree. All index pages must have \( \lceil b/2 \rceil \leq \# \text{ child} \leq b \), except the root which must have at least two children.

Consider an B+ tree example with \( b = 5 \).

\[
\text{\begin{center}
\begin{tikzpicture}
\node (root) at (0,0) {80};
\node (left) at (-2,-1) {40 65};
\node (left_left) at (-4,-2) {8 25 40 60};
\node (left_right) at (-1,-2) {65 70 72 80 88};
\node (right) at (2,-1) {90 120 130};
\node (right_left) at (4,-2) {90 95 120 125};
\node (right_right) at (1,-2) {130 171};
\draw (root) -- (left);
\draw (left) -- (left_left) -- (left_right);
\draw (root) -- (right);
\draw (right) -- (right_left) -- (right_right);
\end{tikzpicture}
\end{center}
}\]

\( a \) How would you find 88?

\( b \) The insert algorithm for a B+ tree is summarized by the below table. Where would you insert 50, 100, 105, 110, 180, 200, 210?

<table>
<thead>
<tr>
<th>Situation</th>
<th>Insertion Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Page Full?</td>
<td>Parent Index Page Full?</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
| Yes | No | 1. Split data page with records < middle key going in left data page and records \( \geq \) middle key going in right data page.  
2. Place middle key in index page in sorted order with the pointer immediately to its left pointing to the left data page and the pointer immediately to its right pointing to the right data page. |
| Yes | Yes | 1. Split data page with records < middle key going in left data page and records \( \geq \) middle key going in right data page.  
2. Adding middle key to parent index page causes it to split with keys < middle key going into the left index page, keys \( > \) middle key going in right index page, and the middle key inserted into the next higher level index page. If the next higher index page is full continue to splitting index pages up the B+ tree as necessary. |
c). For a B+ tree with a branch factor 201, what would be the worst case height of the tree if the number of keys was 1,000,000,000,000?

10. The deletion algorithm for a B+ tree is summarized by the below table.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Deletion Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Delete record from the data page. Shifting records with larger keys to left to fill in the hole. If the deleted key appears in the index page, use the next key to replace it.</td>
</tr>
<tr>
<td>Yes</td>
<td>1. Combine data page and its sibling. Change the index page to reflect the change.</td>
</tr>
</tbody>
</table>
| Yes       | 1. Combine data page and its sibling.  
2. Adjusting the index page to reflect the change causes it to drop below the fill factor, so combine the index page with its sibling.  
3. Continue combining the next higher level index pages until you reach an index page with the correct fill factor or you reach the root index page. |

Consider an B+ tree example with $b = 5$ and 50% fill factor. Delete 89, 65, and 88. What is the resulting B+ tree?

![Diagram of B+ tree](image-url)
1. Consider the following directed graph (digraph) $G = (V, E)$:

![Graph Diagram]

a) What is the set of vertices? $V =$

b) An edge can be represented by a tuple (from vertex, to vertex [, weight]). What is the set of edges $E =$

c) A path is a sequence of vertices that are connected by edges. In the graph $G$ above, list two different paths from $v_0$ to $v_3$.

d) A cycle in a directed graph is a path that starts and ends at the same vertex. Find a cycle in the above graph.

2. Like most data structures, a graph can be represented using an array, or as a linked list of nodes. The array representation is a two-dimensional array (called an adjacency matrix) whose elements contain information about the edges and the vertices corresponding to the indices. (Python could use a list-of-lists)

   a) Complete the following adjacency matrix for the above graph. (Here a missing edge is represented by $\infty$.)

   |   | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
---|-----|-----|-----|-----|-----|
|$v_0$| 0   | 1   | \infty| 1    | 5    |
|$v_1$| 9   | 0   | 3    | 3    | \infty|
|$v_2$|     |     |      |      |      |
|$v_3$|     |     |      |      |      |
|$v_4$|     |     |      |      |      |

b) What is the big-oh to determine the edge-weight between any two vertices?

c) What is the big-oh amount of storage used to store the adjacency matrix?

d) The linked representation maintains a linked-list (or Python dictionary) of vertices with each vertex maintaining a linked list of other vertices that it connects to. Complete the adjacency list representation below:

   ![Adjacency List Diagram]

e) What is the big-oh to determine the edge-weight between any two vertices in an adjacency list?

f) What is the big-oh amount of storage used to store the adjacency list?
3. Below is the textbook’s edge, vertex, and graph implementations.
   a) How does this graph implementation maintain its set of vertices?

   b) How does this graph implementation maintain its set of edges?

   c) What is the big-oh to determine the edge-weight between any two vertices

```python
class Graph:
    def __init__(self):
        self.vertList = {}  
        self.numVertices = 0

    def addVertex(self, key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self, n):
        return n in self.vertList

    def addEdge(self, f, t, cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())

```
4. Graphs can be used to solve many problems by modeling the problem as a graph and using "known" graph algorithm(s). For example, consider the word-ladder puzzle where you transform one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL \rightarrow FOIL \rightarrow FAIL \rightarrow FALL \rightarrow PALL \rightarrow PALE \rightarrow SALE \rightarrow SAGE.

We can use a graph algorithm to solve this problem by constructing a graph such that
- a word represents a vertex
- an edge represents?
- a word ladder transformation from one word to another represents?

4. For the words listed below, draw the graph of question 3

```
fool  fail  fall
  /\    /\    /\  
foil foul pole
      /\  /\  /\  
poll poll sale
  /\  /\  /\  /\  
pale pale page

a) List a different transformation from FOOL to SAGE

b) If we wanted to find the shortest transformation from FOOL to SAGE, what does that represent in the graph?
```

There are two general approaches for traversing a graph from some starting vertex \( s \):
- Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from \( s \), before finding all vertices a distance 2 from \( s \), etc.
- Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a "dead end," we backtrack to the deepest vertex that allows us to try a different path.

c) Which of these traversals would be helpful for finding the shortest solution to the word-ladder puzzle?
Objectives: To understand how a graph can be represented and traversed.

To start the lab: Download and unzip the lab11.zip file from eLearning.

Part A: In a word-ladder puzzle (discussed in class) transforms one word into another by changing one letter at a time, e.g., transform FOOL into SAGE by FOOL → FOIL → FAIL → FALL → PALL → PALE → SALE → SAGE.

We used a graph algorithm to solve this problem by constructing a graph such that
- words are represented by the vertices, and
- edges connect vertices containing words that differ by only one letter

a) For the words listed below, complete the graph by adding edges as defined above.

```
fool  cool  pool
  |     |
  v     v
foul   pale
  |     |
  v     v
pope
```

b) To find the shortest transformation from FOOL to SAGE, why did we decide on using a Breadth First Search (BFS) traversal (i.e., where you find all vertices a distance 1 (directly connected) from FOOL, before finding all vertices a distance 2 from FOOL, etc) instead of a Depth-First Search (DFS) traversal?

c) Run the lab11/word_ladder_BFS.py program. Examine the “enqueue” and “dequeue” lines of output produced by the bfs(g, g.getVertex("fool")) call. Does this output match the expected “enqueues” and “dequeues” performed during a bfs of the above graph starting at “fool”?

d) The bfs algorithm sets the value of each vertex’s predecessor to point to the vertex object that enqueued it. Add code to the end of the word_ladder_BFS.py program that traverses the “linked list” of predecessor references from “sage” to “fool.” and prints the corresponding word ladder from “fool” to “sage.”

Hint: The code you need to write is similar to the __str__ code for traversing a singly-linked list:

```
UnorderedList Object

_head [ ]
```

```python
'data' 'next'
data next data next data next
'w' 'a' 'y' 'm' 'c'
```
def __str__(self):
    resultStr = ""
    current = self.head
    while current != None:
        resultStr += " " + str(current.getData())
        current = current.getNext()
    return resultStr

Your code (see partial code at the end of the word_ladder_BFS.py program) can:
- walk currentVert down the linked list of Vertex objects using getPred instead of getNext
- append to wordLadderList the currentVert's word gotten using getId instead of string concatenating using getData

After your while-loop executes, you can reverse the wordLadderList and print the transformation from "fool" to "sage."

Part B: Topological sort
Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The DFSGraph uses a time attribute to note when a vertex if first encountered (discovery attribute) in the depth-first search and when a vertex in backtracked through (finish attribute). Consider the graph for making pancakes where vertices are steps and edges represents the partial order among the steps.

![Graph Diagram]

a) Run the lab11/make_pancake_DFS.py program. Write on the above graph the discovery and finish attributes (e.g., 1/12 of "milk") assigned to each vertex by executing the dfs method.

b) A topological sort algorithm can use the dfs finish attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour 1/2 cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort from the finish attributes.

After you have answers and correct code for all parts of the lab, submit a lab11.zip containing your code on cLearning. If you do not get done today, then submit it by next week's lab period.

Remember to save your lab11!

EXTRA CREDIT:
Add code to the end of the make_pancake_DFS.py program to print the topological sort for making pancakes.
1. There are two general approaches for traversing a graph from some starting vertex s:
   - Depth First Search (DFS) where you explore as deeply into the graph as possible. If you reach a “dead end,” we backtrack to the deepest vertex that allows us to try a different path.
   - Breadth First Search (BFS) where you find all vertices a distance 1 (directly connected) from s, before finding all vertices a distance 2 from s, etc.

   What data structure would be helpful in each type of search? Why?
   a) Breadth First Search (BFS):

   b) Depth First Search (DFS):

2. Assuming a graph \( g \) containing the word-ladder graph from lecture 25, on the diagram trace the \( \text{bfs} \) algorithm by showing the value of each vertex’s color, predecessor, and distance attributes?
Data Structures (CS 1520)

Lecture 26

Name:

""" File: vertex.py """

class Vertex:
    def __init__(self, key, color = 'white',
                 dist = 0, pred = None):
        self.id = key
        self.connectedTo = {}
        self.color = color
        self.predecessor = pred
        self.distance = dist
        self.discovery = 0
        self.finish = 0

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo:
        + str([x.id for x in self.connectedTo])

def getConnections(self):
    return self.connectedTo.keys()

def getId(self):
    return self.id

def getWeight(self,nbr):
    return self.connectedTo[nbr]

def getColor(self):
    return self.color

def setColor(self, newColor):
    self.color = newColor

def getNext(self):
    return self.predecessor

def setPred(self, newPred):
    self.predecessor = newPred

def getDiscovery(self):
    return self.discovery

def setDiscovery(self, newDiscovery):
    self.discovery = newDiscovery

def getFinish(self):
    return self.finish

def setFinish(self, newFinish):
    self.finish = newFinish

def getDistance(self):
    return self.distance

def setDistance(self, newDistance):
    self.distance = newDistance

""" File: graph.py """

from vertex import Vertex

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,key):
        if key in self.vertList:
            return self.vertList[key]
        else:
            return None

    def __contains__(self,key):
        return key in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(t)
        self.vertList[t].addNeighbor(f)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())

""" File: graph_algorithms.py """

from graph import Graph
from vertex import Vertex
from linked_queue import LinkedQueue

def bfs(g,start):
    start.setDistance(0)
    start.setPred(None)
    vertQueue = LinkedQueue()
    vertQueue.enqueue(start)
    while (vertQueue.size() > 0):
        currentVert = vertQueue.dequeue()
        for nbr in currentVert.getConnections():
            if (nbr.getColor() == 'white'):
                nbr.setColor('gray')
                currentVert.setColor('gray')
                nbr.setDistance(currentVert.getDistance()+1)
                nbr.setPred(currentVert)
                vertQueue.enqueue(nbr)
                currentVert.setColor('black')
3. Section 7.5 uses recursion and the run-time stack to implement a DFS traversal. The `DPSGraph` uses a `time` attribute to note when a vertex is first encountered (discovery attribute) in the depth-first search and when a vertex in backtracked through (finish attribute). Consider the graph for making pancakes where vertices are steps and edges represent the partial order among the steps.

```python
from graph import Graph
class DPSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self, startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

a) Assume (why is this a bad assumption???) that the for-loops always iterate through the vertexes alphabetically (e.g., "eat", "egg", "flour", ...) by their id. Write on the above graph the discovery and finish attributes assigned to each vertex by executing the dfs method.

b) A topological sort algorithm can use the dfs discovery and finish attributes to determine a proper order to avoid putting the "cart before the horse." For example, we don't want to "pour ½ cup of batter" before we "mix the batter", and we don't want to "mix the batter" until all the ingredients have been added. Outline the steps to perform a topological sort.
4. Consider the following directed graph (diagraph).

Dijkstra's Algorithm is a greedy algorithm that finds the shortest path from some vertex, say \( v_0 \), to all other vertices. A greedy algorithm, unlike divide-and-conquer and dynamic programming algorithms, DOES NOT divide a problem into smaller subproblems. Instead, a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices). Dijkstra's algorithm builds a subgraph by repeatedly selecting the next closest vertex to \( v_0 \) that is not already in the subgraph. Initially, only vertex \( v_0 \) is in the subgraph with a distance of 0 from itself.

a) What would be the order of vertices added to the subgraph during Dijkstra's algorithm?

\( v_0, \)

b) What greedy criteria did you use to select the next vertex to add to the subgraph?

c) What data structure could be used to efficiently determine that selection?

d) How might this data structure need to be modified?
1. Suppose you had a map of settlements on the planet X
   (Assume edges connecting all vertices with their Euclidean distances as their costs)

We want to build roads that allow us to travel between any pair of cities. Because resources are scarce, we want the total length of all roads built to be minimal. Since all cities will be connected anyway, it does not matter where we start, but assume we start at “a”.

a) Assuming we start at city “a” which city would you connect first? Why this city?

b) What city would you connect next to expand your partial road network?

c) What would be some characteristics of the resulting "graph" after all the cities are connected?

d) Does your algorithm come up with the overall best (globally optimal) result?
2. Prim’s algorithm for determining the minimum-spanning tree (MST) of a graph is another example of a greedy algorithm. Unlike divide-and-conquer and dynamic programming algorithms, greedy algorithms DO NOT divide a problem into smaller subproblems. Instead a greedy algorithm builds a solution by making a sequence of choices that look best ("locally" optimal) at the moment without regard for past or future choices (no backtracking to fix bad choices).

a) What greedy criteria does Prim’s algorithm use to select the next vertex and edge to the partial minimum spanning tree?

b) What data structure would be useful it finding the next vertex to add to the partial MST?

The textbook’s Prim’s Algorithm code (Listing 7.12 p. 346) was incorrect, but fixed on-line to:

```python
def prim(G, start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(), v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newCost = currentVert.getWeight(nextVert) + currentVert.getDistance()
            if nextVert in pq and newCost < nextVert.getDistance():
                nextVert.setPred(currentVert)
                nextVert.setDistance(newCost)
                pq.decreaseKey(nextVert, newCost)
```

c) The `PriorityQueue` class used is shown on the next page and is similar to the `BinHeap` class we have been using, except:

| self.heapArray is a list of tuples with the first tuple value (tuple index 0) being the “priority” and second tuple value (tuple index 1) being its associated value. |
| In Prim’s algorithm what is the priority value? |

In Prim’s algorithm what is the associated value?

- a `__contains__` method is added to check if a value is in the priority queue.
  In Prim’s algorithm where is the `__contains__` method invoked?

- a `decreaseKey` method is added to allow a priority value to be reduced (i.e., increasing its priority).
  In Prim’s algorithm when is the `decreaseKey` method used?

d) As written what is the big-oh of each of the methods?

| `__contains__` method? |
| `decreaseKey` method? |
The PriorityQueue class including the 
Contains and decreaseKey methods:

3. If we want to speed-up the Contains and 
decreaseKey methods, then what type of data 
structure could we add to aid in:
- checking for the existence of a key value, and 
- if a key value exists in the heap, then at what 
index does it reside?

4. What modifications would need be needed to 
other methods to keep the data structure in 
question 3 up-to-date?
Objectives: To understand how a graph can be used to solve graph algorithms.

To start the lab: Download and unzip the lab12.zip file from eLearning.

Part A: In IDLE open the PriorityQueue class file: lab12/priorityQueue.py. The changes to our familiar BinHeap class that we discussed in lecture are included:

- `self.heapArray` is a list of tuples with the first tuple value (tuple index 0) being the “priority” and second tuple value (tuple index 1) being its associated “key” value.
- A `__contains__` method is added to check if a value is in the priority queue.
- A `decreaseKey` method is added to allow a priority value to be reduced (i.e., increasing its priority).

a) Why do the methods `percDown`, `minChild`, and `percUp` used the tuple index 0 when comparing tuples in the `self.heapArray`?

b) Why do the methods `decreaseKey` and `__contains__` used the tuple index 1 when comparing tuples in the `self.heapArray`?

c) Run the `lab12/make_min_spanning_tree.py` program which uses Prim’s algorithm on the graph from lecture. Does it give the expected output?

d) Predict the order of edges added by Prim’s algorithm if we start at vertex “e”:

![Graph Diagram]

d) Modify the `lab12/make_min_spanning_tree.py` program to verify your prediction. NOTE: This is a very easy modification. You just need to start Prim’s algorithm starting at the vertex labeled “e”.

Name:____________
Data Structures  

Lab 12 Graphs 2

Name: ______________________

**Part B:** The textbook’s Dijkstra’s Algorithm code (Listing 7.11 p. 341 and same on-line) is in the lab12/graph_algorithms.py file.

a) Run the lab12/test_dijkstra.py program which uses Dijkstra’s algorithm on the graph from lecture. Does it give the expected output?

![Graph Image]

b) Modify the dijkstra function in the lab12/graph_algorithms.py file by comparing it to the similar prim function. (HINT: Vertex objects are created with a default distance attribute value of 0. Dijkstra’s algorithm is missing initialization code similar to Prim’s algorithm is lab12/graph_algorithms.py file)

After you have fixed the dijkstra function in lab12/graph_algorithms.py, test your code by running the lab12/test_dijkstra.py program.

**Part C:** In IDLE open the PriorityQueue class file: lab12/Part_C/priorityQueue.py. This version adds a data attribute self.keyToIndexDict dictionary where the keys are all the second tuple values in the self.heapArray and their values are their corresponding index locations in the self.heapArray.

The methods buildHeap, percDown, minChild, and percUp have all been modified to correctly update the self.keyToIndexDict dictionary.

Your task for Part C is to complete the methods decreaseKey and __contains__ so that they use the self.keyToIndexDict dictionary. Thus, greatly improving the efficiency of decreaseKey and __contains__ methods. Test your methods by running the lab12/Part_C/make_min_spanning_tree.py program.

After you have answers and correct code for all parts of the lab, submit a lab12.zip containing your code on cLearning. If you do not get done today, then submit it by next Wednesday (11/25) at noon.

The EXTRA CREDIT Opportunities related to Part B:
Add code to the end of the test_dijkstra.py program to print the shortest paths from v0 to each of the other vertices. One line of output might look something like:
“Shortest path from v0 to v4 is v0 > v3 > v4 with a total distance of 4”
Final Review Topics

The Final exam is Tuesday November 24th from 8:00 - 9:50 AM in ITT 328. It will be closed-book and notes, except for three 8” x 11” sheets of paper containing any notes that you want. (Plus, the Python Summary Handout) About 75% of the test will cover the following topics (and maybe more) since the second mid-term test, and the remaining 25% will be comprehensive (mostly big-oh analysis and general questions about stacks, queues, priority queues/heaps, lists, and recursion).

Chapter 6: Trees
Terminology: node, edge, root, child, parent, siblings, leaf, interior node, branch, descendant, ancestor, path, path length, depth/level, height, subtree
General and binary tree recursive definitions
Tree shapes and their heights: full binary tree, balanced binary tree, complete binary tree
Applications: parse tree, heaps, binary search trees, expression trees
Traversals: inorder, preorder, postorder
Binary search tree ADT: interface, implementation, big-oh of operations
Balanced binary search trees: AVL tree ADT: interface, implementation, big-oh of operations

File Structures - Lecture 24 handout:
We talked about how the in memory data structures need to be adapted for slow disks. From this discussion you should understand the general concepts of Magnetic disks:
• layout (surfaces, tracks/cylinders, sectors, R/W heads)
• access time components (seek time - moving the R/W heads over the correct track, rotational delay - disk spins to R/W head, data transfer time - reading/writing of sector as it spins under the R/W head)
Hash Table as a useful file structure
B+ trees as a useful file structure - see web resources:
  http://www.sci.umich.it/~acciaro/bpiutrees.pdf
  http://en.wikipedia.org/wiki/B+_tree

Chapter 7: Graphs
Terminology: vertex/vertices, edge, path, cycle, directed graph, undirected graph
Graph implementations: adjacency matrix and adjacency list
Graph traversals/searches: Depth-First Search (DFS) and Breadth-First Search (BFS)
General Idea of the following algorithms: topological sort, Dijkstra’s algorithm (single-source, shortest path), Prim’s algorithm (determines the minimum-spanning tree

You should understand the graph implementations and algorithms listed above. You should be able to trace the algorithms on a given graph.