CS 302 – Assignment #5

Purpose: Learn concepts regarding balanced binary trees.
Due: Monday (10/08) → Must be submitted on-line before class.
Points: Part A → 150 pts, Part B → 50 pts

Assignment:

Part A:

Design and implement a C++ template class, ***avlTree.h***, to implement an AVL Tree[1] data structure. A main will be provided that performs a series of tests using AVL tree with different data types.

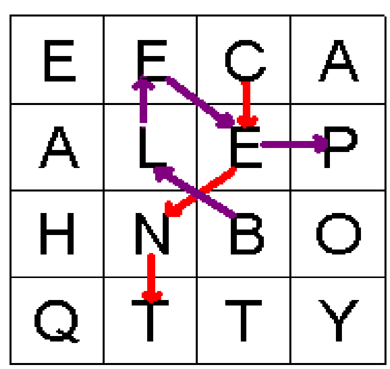
Part B:

When the ***avlTree*** data structure is implemented and tested, create a C++ ***wordPuzzle*** class for a word search (which inherits from the ***avlTree*** class). This class will include reading a dictionary file (storing the words in a ***avlTree***), reading a letter grid (as shown on right), and searching for words in the letter grid.

From any starting position, a word can be formed by a sequence of letters where the following letter must be adjacent to the previous letter in any direction. The goal of the word puzzle class is to find all legal dictionary words in the provided letter grid. There are 130 words in the provided example (using *smallDictionary.txt*). *Note*, a word is considered different if it has a different path. For example, PACE can be spelled two ways starting from (1,3) and thus can be counted as two words.

E	E	C	A
A	L	E	P
H	N	B	O
Q	T	T	Y

Example word search game board



Word search show words CENT and BLEEP

Part C:

When completed, create and submit a write-up (open document, word or PDF format) not too exceed ~500 words including the following:

- Name, Assignment, Section
- Summary of the AVL tree data structure.
- Compare using an AVL tree to a binary search tree.
    - Include advantages and disadvantages of each implementation approach.
- Big-O for the various AVL tree operations.

It should be noted that there are many implementation variations on these data structures and algorithms. These are the algorithms that must be implemented. Copying code from the net will result in a zero for the assignment and referral to the Office of Student Conduct.

1 For more information, refer to: http://en.wikipedia.org/wiki/AVL_tree

Visualization

The following web site provides a visualization for AVL Trees, including the rotate operations.

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Make File:

You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executables.

Submission:

- Part A/B → Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 4:00 PM.
- Part B → A copy of the write-up including the chart (see example). Must use PDF format.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

Class Descriptions

- AVL Tree Class
  The AVL tree template stack class will implement functions specified below.
  We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType        keyValue;
      int           nodeHeight;
      nodeType<myType>   *left;
      nodeType<myType>   *right;
};
```

Additionally, include the following enumeration definition:

```
enum  treeTraversalOptions { INORDER, PREORDER,
            POSTORDER, LEVELORDER, NONE };
```

avlTree<myType>
-nodeType<myType> *root
+avlTree()
+~avlTree()
+destroyTree(): void
+countNodes() const: int
+height() const: int
+search(myType) const: bool
+printTree(treeTraversalOptions) const: void
+insert(myType): void
+deleteNode(myType): void

+isPrefix(string) const: bool
-destroyTree(nodeType<myType> *): void
-countNodes(nodeType<myType> *) const: int
-height(nodeType<myType> *) const: int
-search(myType, nodeType<myType> *) const: nodeType<myType> *
-printTree(nodeType<myType> *, treeTraversalOptions) const: void
-printLevelOrder() const: void
-insert(myType, nodeType<myType> *): nodeType<myType> *
-rightRotate(nodeType<myType> *): nodeType<myType> *
-leftRotate(nodeType<myType> *): nodeType<myType> *
-getBalance(nodeType<myType> *) const: int
-deleteNode(myType, nodeType<myType> *): nodeType<myType> *
-minValueNode(nodeType<myType> *) const: nodeType<myType> *

Function Descriptions

- The *avlTree()* constructor should initialize the tree to an empty state.
- The *~avlTree()* destructor should delete the tree by calling the private *destroyTree()* function.
- The public *destroyTree()* function should delete the tree by calling the private *destroyTree()* function.
- The private *destroyTree()* function should delete the tree (including releasing all the allocated memory).
- The public *countNodes()* function should return the total count of nodes in the tree by calling the private *countNodes()* function.
- The private *countNodes()* function should recursively return the total count of nodes in the tree. Must be recursive.
- The public *height()* function should return the maximum height of the tree by calling the private *height()* function.
- The private *height()* function should recursively return maximum height of the tree. Must be recursive.
- The public *search()* function should call the private *search()* function to determine if the passed node key is in the tree. If the node if found, the function should return true and return false otherwise.
- The private *search()* function should recursively search the tree for the passed node key. Must be recursive.
- The public *printTree()* function should call the private *printTree()* function to print the tree in the order passed.
- The private *printTree()* function should recursively print the tree in the specified order. Must be recursive for post-order, pre-order, and in-order. *Note*, the LEVELORDER option calls the *printLevelOrder()* function which performs the printing that specific print option.
- The private *printLevelOrder()* function should print the tree in level order by performing a breadth first traversal (BFS). Use the provided algorithm (which uses the linked queue object from the previous assignment).
- The public *insert()* function should call the private *insert()* function to insert the passed key value into the tree. If the node is already in the tree, it should not be inserted again and no error message is required.
- The private *insert()* function should recursively insert the passed key value into the tree. The function will use the private *leftRotate()*, *rightRotate()*, and *getBalance()* functions.

- The public *deleteNode()* function should call the private *deleteNode()* function to delete the passed key value from the tree (if it exists). If the key does not exist, no error message is required.
- The *isPrefix()* function should determine if the passed prefix is in the tree. The prefix does not need to be a word. If the prefix is found, the function should return true and return false otherwise.
- The private *deleteNode()* function should recursively delete the passed key value from the tree (if it exists). The function will use the private *leftRotate()*, *rightRotate()*, *getBalance()* functions, and *minValueNode()* functions.
- The *minValuenode()* function should search the tree starting from the passed node and return the node with the minimum key value. Does not need to be recursive. *Hint*, need only follow the left tree brnach.
- The private *getBalance()* function should return the balance factor (left subtree height – right subtree height) of the passed node.
- The private *rightRotate()* function should perform a right tree rotate operation (as described in class, in the lecture notes, and in the text).
- The public *leftRotate()* function should perform a left tree rotate operation (as described in class, in the lecture notes, and in the text).

- Word Puzzle Class
  The word puzzle class should inherit from the ***avlTree*** class and implement functions specified below.

WordPuzzle: public avlTree<string>
-title: string
-order: int
-**letters: string
-wordsFound: avlTree<string>
+wordPuzzle()
+~wordPuzzle()
+readLetters(const string): bool
+readDictionary(const string): bool
+getArguments(int, char *[], string &, string &): bool
+findWords(): void
+showTitle() const: void
+showWordCount() const: void
+showWords() const: void
+printLetters() const: void
-findWords(int, int, string): void

Function Descriptions

- The *wordPuzzle()* constructor should initialize the class variables to an empty state.
- The *~wordPuzzle()* destructor should delete the letters array.
- The *showTitle()* function should display the puzzle title set by the *readLetters()* function.
- The *showWordCount()* function should use the appropriate ***avlTree*** function and display the number of words found.

- The *getArguments()* function read the passed command line information. If no arguments are entered, it should display a usage message ("**Usage: ./findWords -d <dictionaryFile> -w <wordsFile>**"). If the arguments are invalid, it should display an error message ("**Error, command line arguments invalid.**"). If the arguments are valid, it should return the dictionary file string and the words file string.
- The public *findWords()* should use the private *findWords()* function.
- The private *findWords()* function should find all words in the letter grid. The word and the location should be stored in the AVL tree (as a string) with the ending location in the format shown in the provided example. As noted, a word is considered different if the path is different. As such, the same word may be found multiple times with different paths.
- The *readLetters()* function should read the formatted letters grid from the passed file name and store the letters as strings in a dynamically allocated two-dimensional array, *letters*. The format includes a title line (1st line), the order (2nd line) and the letters (3rd line on) with *order* number rows and *order* number of letters per row. For example:

```
Simple Puzzle, Words #1
4
e e c a
a l e p
h n b o
q t t y
```

  The class variables for order and title should be set appropriately. If the file read is successful, the function should return true and false otherwise.
- The *readDictionary()* function should read the passed dictionary file name and store the words in the ***avlTree***. Some dictionary files are provided. If the file read is successful, the function should return true and false otherwise.
- The *printLetters()* function should display the title and the formatted letters grid. For example:

```
Letter Set Title: Simple Puzzle, Words #1

 ___ ___ ___ ___
|   |   |   |   |
| e | e | c | a |
|___|___|___|___|
|   |   |   |   |
| a | l | e | p |
|___|___|___|___|
|   |   |   |   |
| h | n | b | o |
|___|___|___|___|
|   |   |   |   |
| q | t | t | y |
|___|___|___|___|
```

- The *showWords()* function should display all the words found in alphabetical. The appropriate ***avlTree*** print function should be used. Refer to the example execution for the output formatting.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

AVL Tree Algorithms

The following is a summary of a some of the AVL tree algorithms.

AVL Tree Balance Function

- AVL Tree Get Balance Function
    - return height(left) - height(right)

AVL Tree Height Function

- AVL Tree Height → Private Function
    - if node is NULL
        - return 0
    - else
        - recursively get left height
        - recursively get right height
        - return max left height or right height + one

AVL Tree Insertion Function

- AVL Tree Insertion → Private Function
    - recursively perform normal BST insertion
        - if NULL
          insert new node
          return node
        - else
          based on key, go left or right
    - get balance factor
    - check for possible cases for unbalanced
        - if (balance factor > 1 AND key < left node value) // left left case
          return right rotate
        - if (balance factor < -1 AND key > right node value) // right right case
          return left rotate
        - if (balance factor > 1 AND key > left node value) // left right case
          left node = left rotate
          return right rotate
        - if (balance factor < -1 AND key < right node value) // right left case
          right node = right rotate
            - return left rotate
    - return node (possibly unchanged)

Level Order → Print Algorithm

A level order print of a binary tree can be performed multiple ways. The following approach uses the linked queue object from the previous assignment.

```
// -------------------------------
//  Print tree in level order.

//     1) create an empty queue Q
//     2) curr = root      /*start from root*/
//     3) loop forever
//            a) print curr->data.
//            b) enqueue curr's children (first left then right children) to Q
//            c) if Q is empty, exit loop
//            d) dequeue a node from Q and assign to curr
```

Note, you must the use algorithm.

Example Execution:

Below is an example program execution for the main.

```
ed-vm% ./mainAVLtest
-----------------------------------------------------------------
CS 302 - Assignment #5

-----------------------------------------------------------------
Test Set #0
      Nodes:  7
      Height: 3

In-order traversal:
  5   6   8   10   11   14   18

Pre-order traversal:
  10   6   5   8   14   11   18

Post-order traversal:
  5   8   6   11   18   14   10

BFS traversal:
  10   6   14   5   8   11   18


-----------------------------------------------------------------
Test Set #1
      Nodes:  24
      Height: 5

In-order traversal:
  1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18   19   20   21   22
23   24

Pre-order traversal:
  16   8   4   2   1   3   6   5   7   12   10   9   11   14   13   15   20   18   17   19   22   21
23   24

Post-order traversal:
  1   3   2   5   7   6   4   9   11   10   13   15   14   12   8   17   19   18   21   24   23   22
20   16

BFS traversal:
  16   8   20   4   12   18   22   2   6   10   14   17   19   21   23   1   3   5   7   9   11   13
15   24

BFS traversal:
  16   8   20   4   12   18   22   2   6   10   14   17   19   21   23   1   3   7   9   11   13
15   24
```

```
-----------------------------------------------------------------
Test Set #2
       Nodes:  15
       Height: 5

In-order traversal:
  3  11  17  21  28  29  32  44  54  65  76  80  82  88  97

Pre-order traversal:
  44  28  11  3  17  21  32  29  82  65  54  76  80  88  97

Post-order traversal:
  3  21  17  11  29  32  28  54  80  76  65  97  88  82  44

BFS traversal:
  44  28  82  11  32  65  88  3  17  29  54  76  97  21  80


Modified Tree:
       Nodes:  11
       Height: 4

BFS traversal:
  44  21  88  11  29  76  97  3  32  54  80

-----------------------------------------------------------------
Test Set #3
       Nodes:  5000
       Height: 13


Modified Tree:
       Nodes:  37
       Height: 6

BFS traversal:
  9955  15  9983  7  23  9967  9991  3  11  19  27  9959  9975  9987  9995  1
5  9  13  17  21  25  9957  9963  9971  9979  9985  9989  9993  9997  9961
9965  9969  9973  9977  9981  9999


-----------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
```

Below is an example program execution for the main.

```
ed-vm% ./findWords -d smallDictionary.txt -w words1.txt
-----------------------------------------------------------------
CS 302 - Assignment #6
Word Search Puzzle Solver

-----------------------------------------------------------------
Letter Set Title: Simple Puzzle, Words #1

    ___ ___ ___ ___
   |   |   |   |   |
   | e | e | c | a |
   |___|___|___|___|
   |   |   |   |   |
   | a | l | e | p |
   |___|___|___|___|
   |   |   |   |   |
   | h | n | b | o |
   |___|___|___|___|
   |   |   |   |   |
   | q | t | t | y |
   |___|___|___|___|

  ace     from: (0,1)
  ace     from: (1,2)
  ae     from: (0,0)
  ae     from: (0,1)
  ae     from: (1,2)
  ah     from: (2,0)
  aha     from: (1,0)
  al     from: (1,1)
  ala     from: (1,0)
  alae     from: (0,0)
  alae     from: (0,1)
  alan     from: (2,1)
  alane     from: (1,2)
  alant     from: (3,1)
  alant     from: (3,2)
  alb     from: (2,2)
  ale     from: (0,0)
  ale     from: (0,1)
  ale     from: (1,2)
  alec     from: (0,2)
  alee     from: (0,0)
  alee     from: (0,1)
  alee     from: (1,2)
  an     from: (2,1)
  ana     from: (1,0)
  anal     from: (1,1)
  ane     from: (1,2)
  anele     from: (0,0)
  anele     from: (0,1)
  anele     from: (1,2)
  anent     from: (3,1)
  anent     from: (3,2)
  ant     from: (3,1)
  ant     from: (3,2)
  apace     from: (0,1)
  apace     from: (1,2)
  ape     from: (1,2)
  be     from: (1,2)
  bebop     from: (1,3)
  becap     from: (1,3)
  bee     from: (0,1)
  bel     from: (1,1)
  beleap     from: (1,3)
  ben     from: (2,1)
```

```
bene     from: (1,2)
bent     from: (3,1)
bent     from: (3,2)
benthal     from: (1,1)
blae     from: (0,0)
blae     from: (0,1)
blah     from: (2,0)
bleb     from: (2,2)
bleep     from: (1,3)
blent     from: (3,1)
blent     from: (3,2)
bo     from: (2,3)
bob     from: (2,2)
bop     from: (1,3)
bot     from: (3,2)
bott     from: (3,1)
boy     from: (3,3)
boyo     from: (2,3)
by     from: (3,3)
caca     from: (0,3)
caeca     from: (0,3)
cap     from: (1,3)
cape     from: (1,2)
capelan     from: (2,1)
capo     from: (2,3)
ceca     from: (0,3)
cee     from: (0,0)
cee     from: (0,1)
cee     from: (1,2)
cel     from: (1,1)
celeb     from: (2,2)
cent     from: (3,1)
cent     from: (3,2)
cento     from: (2,3)
cep     from: (1,3)
cepe     from: (1,2)
clan     from: (2,1)
clean     from: (2,1)
clepe     from: (1,2)
eel     from: (1,1)
el     from: (1,1)
elan     from: (2,1)
en     from: (2,1)
epee     from: (0,1)
epopee     from: (0,1)
ha     from: (1,0)
hae     from: (0,0)
hae     from: (0,1)
hah     from: (2,0)
haha     from: (1,0)
halala     from: (1,0)
halalah     from: (2,0)
hale     from: (0,0)
hale     from: (0,1)
hale     from: (1,2)
hant     from: (3,1)
hant     from: (3,2)
la     from: (1,0)
lane     from: (1,2)
lea     from: (0,3)
lea     from: (1,0)
leal     from: (1,1)
lean     from: (2,1)
leant     from: (3,1)
leant     from: (3,2)
leap     from: (1,3)
leben     from: (2,1)
lee     from: (0,0)
lee     from: (0,1)
```

```
lee     from: (1,2)
lent     from: (3,1)
lent     from: (3,2)
lento     from: (2,3)
na     from: (1,0)
nae     from: (0,0)
nae     from: (0,1)
nah     from: (2,0)
nan     from: (2,1)
nana     from: (1,0)
ne     from: (1,2)
neap     from: (1,3)
neb     from: (2,2)
nee     from: (0,1)
nene     from: (1,2)
nth     from: (2,0)
obe     from: (1,2)
oboe     from: (1,2)
oe     from: (1,2)
op     from: (1,3)
ope     from: (1,2)
open     from: (2,1)
oy     from: (3,3)
pa     from: (0,3)
pac     from: (0,2)
paca     from: (0,3)
pace     from: (0,1)
pace     from: (1,2)
pap     from: (1,3)
papa     from: (0,3)
pe     from: (1,2)
pea     from: (0,3)
peace     from: (0,1)
peace     from: (1,2)
pec     from: (0,2)
pee     from: (0,1)
peel     from: (1,1)
pele     from: (0,0)
pele     from: (0,1)
pele     from: (1,2)
pen     from: (2,1)
penal     from: (1,1)
pent     from: (3,1)
pent     from: (3,2)
pep     from: (1,3)
pepo     from: (2,3)
pop     from: (1,3)
pope     from: (1,2)
pot     from: (3,2)
potboy     from: (3,3)
thae     from: (0,0)
thae     from: (0,1)
than     from: (2,1)
thane     from: (1,2)
to     from: (2,3)
toby     from: (3,3)
toe     from: (1,2)
toea     from: (0,3)
toecap     from: (1,3)
top     from: (1,3)
tope     from: (1,2)
topee     from: (0,1)
tot     from: (3,2)
toy     from: (3,3)
toyo     from: (2,3)
yo     from: (2,3)
yob     from: (2,2)
```

```
Stats:
Word Count: 180
Tree Max Height: 17
Tree Node Count: 80368

----------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
```