CS 302
Homework, Asst. #06

Purpose: Learn concepts regarding ***trie*** data structure.
Due: Tuesday (10/07) → Must be submitted on-line before class.
Points: Part A → 125 pts, Part B → 50 pts

Assignment:

Part A:

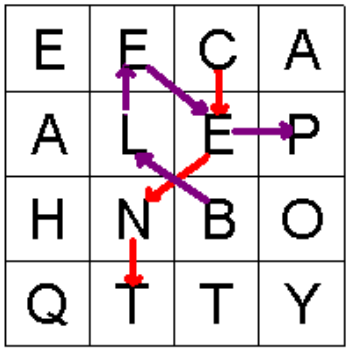
Design and implement a C++ class, ***trieTree***, to implement a ***trie***[1] data structure. A main will be provided that performs a series of tests. Refer to the UML descriptions for implementation details.

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

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

Word Search Game Board

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.



Word Search showing words CENT and BLEEP

Part B:

Create and submit a brief write-up including the following:

- Name, Assignment, Section.
- Summary of the ***trie*** data structure.
- Compare using an ***trie*** to other data structure (i.e., sorted array, binary search tree, and AVL tree). Include advantages and disadvantages of each implementation approach.
- Big-O for the various ***trie*** operations (insert, search, isPrefix).

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.
- Submit a copy of the write-up (open document, word, or 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*).

---

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

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

```
make
```

Which should create the executables.

Class Descriptions

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

```
struct trieNodeType {
      char  keyValue;
      bool  endWordMark;
      trieNodeType  *children[26];
};
```

trieTree
-trieNodeType *root
+trieTree()
+~trieTree()
+countNodes() const: int
+height() const: int
+insert(string): void
+search(string) const: bool
+isPrefix(string) const: bool
+printTree() const: void
+destroyTree(): void
-countNodes(trieNodeType *) const: int
-height(trieNodeType *) const: int
-destroyTree(trieNodeType *): void
-printTree(trieNodeType *) const: void

Function Descriptions

- The *trieTree()* constructor should initialize the tree to an empty state.
- The *~trieTree()* 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 *insert()* function should insert the passed word into the trie tree, including marking the end of word as appropriate.
- The *search()* function should determine if the passed string is a word in the tree. If the word is found, the function should return true and return false otherwise.
- 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 public *printTree()* function should call the private *printTree()* function. *Note*, this function display does not need to show the words in a meaningful manner and is used only for debugging.
- The private *printTree()* function should recursively print the nodes in the tree to print the tree in the order passed.

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

wordPuzzle
-title: string
-order: int
-**letters: string
-wordsFound: avlTree<string>
+wordPuzzle()
+~wordPuzzle()
+readLetters(const string): bool
+readDictionary(const 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 local letters array. *Note*, the base class destructor will automatically be called.
- 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 starting 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 *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 *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 ***trie***. 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. The appropriate ***avlTree*** print function should be used. Refer to the example execution for the output formatting.

- <u>AVL Tree Class</u>
  Note, this class declaration is unchanged from the previous assignment.
  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
-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
-printGivenLevel(nodeType<myType> *, int) 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. *Note*, the LEVELORDER option calls the *printGivenLevel()* function which performs recursion for that specific print option.
- The private *printGivenLevel()* function should print all nodes the passed level. Finding the nodes at the passed level should be performed recursively.
- 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 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).

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.***

Example Execution:

Below is an example program execution for the main.

```
ed-vm% ./main
------------------------------------------------------------------
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)
al     from: (1,1)
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)
ane     from: (1,2)
anele     from: (0,0)
anele     from: (0,1)
ant     from: (3,1)
ant     from: (3,2)
ape     from: (1,2)
be     from: (1,2)
becap     from: (1,3)
bee     from: (0,1)
bel     from: (1,1)
ben     from: (2,1)
bent     from: (3,1)
bent     from: (3,2)
benthal     from: (1,1)
blae     from: (0,0)
blae     from: (0,1)
blah     from: (2,0)
bleep     from: (1,3)
blent     from: (3,1)
blent     from: (3,2)
bo     from: (2,3)
bop     from: (1,3)
bot     from: (3,2)
bott     from: (3,1)
boy     from: (3,3)
by     from: (3,3)
cap     from: (1,3)
cape     from: (1,2)
capelan     from: (2,1)
capo     from: (2,3)
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)
clan     from: (2,1)
clean     from: (2,1)
eel     from: (1,1)
el     from: (1,1)
elan     from: (2,1)
en     from: (2,1)
ha     from: (1,0)
hae     from: (0,0)
hae     from: (0,1)
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)
  lean     from: (2,1)
  leant     from: (3,1)
  leant     from: (3,2)
  leap     from: (1,3)
  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)
  ne     from: (1,2)
  neap     from: (1,3)
  neb     from: (2,2)
  nee     from: (0,1)
  nth     from: (2,0)
  obe     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)
  pace     from: (0,1)
  pace     from: (1,2)
  pe     from: (1,2)
  pea     from: (0,3)
  peace     from: (0,1)
  pec     from: (0,2)
  pee     from: (0,1)
  peel     from: (1,1)
  pele     from: (0,0)
  pele     from: (0,1)
  pen     from: (2,1)
  penal     from: (1,1)
  pent     from: (3,1)
  pent     from: (3,2)
  pot     from: (3,2)
  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)
  toy     from: (3,3)
  yo     from: (2,3)
  yob     from: (2,2)


Stats:
```

```
Word Count: 130
Trie Max Height: 10
Trie Node Count: 153325
```

<output truncated>

```
ed-vm%
```