UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering
Computer & Information Science & Engineering

CAP4053

Manual: Fundamentals of AI

Overview

This exercise serves as a review of basic data structures and an introduction to behavioral modeling using the behavior tree concept. Students will examine and implement a list-based queue as well as a tree functionality for a behavior tree. To ensure adequate preparation for later projects, students are advised not to examine of use code they have written before (and not to copy past from an older project). Students should write the code from scratch.

This activity should take approximately 240m to complete. It should require about 45 minutes of research, 180 minutes of development, and 15 minutes for submission. If students find that this activity takes significantly more time than this estimate, they should contact the instructor.

Structure

The exercise is broken into two main parts:

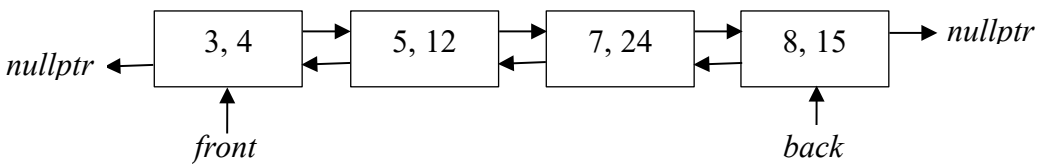
1) Implementation of a linked list (**`LinkedList`**) with standards methods and iterator support
2) Implementation of a tree supporting varying numbers of children (***n-ary***) with standard traversal methods

LinkedList

The LinkedList project is loaded by default as part of the solution. Students should be able to complete this part of the lab within a short period of time. **`LinkedList`** should be implemented as a doubly linked list which terminates in a null pointer (**`nullptr`**). Depending on how one adds to or remove items from either end, the linked list can behave either as a stack or as a queue.

A linked list is made up of ***nodes***. Each node in the list contains some data (in this case, a location represented by a pair of coordinates) and a pointer to the next and previous nodes in the list. The first node in the list is called the *front*, and the last node is called the *back*.

A linked list with location coordinates as data



Implementation

The **`LinkedList`** and **`LinkedList::Iterator`** classes in the ***LinkedList.h*** file should have these methods:

LinkedList<T>::Iterator

`public T operator*() const`
Return the element at the iterator's current position in the queue.

`public Iterator& operator++()`
Pre-increment overload; advance the operator one position in the list. Return this iterator. ***NOTE***: if the iterator has reached the end of the list (past the last element), ita should be equal to **`LinkedList<T>::end()`**.

`public bool operator==(Iterator const& rhs)`
Returns `true` it both iterators point to the same node in the list, and `false` otherwise.

`public bool operator!=(Iterator const& rhs)`
Returns `false` it both iterators point to the same node in the list, and `true` otherwise.

LinkedList<T>

`public LinkedList<T>()`
This is the constructor for **`LinkedList`**.

`public Iterator begin() const`
Returns an **`Iterator`** pointing to the beginning of the list.

`public Iterator end() const`
Returns an **`Iterator`** pointing past the end of the list (an invalid, unique state).

`public bool isEmpty() const`
Returns **`true`** if there are no elements, **`false`** otherwise.

`public T getFront() const`
Returns the first element in the list.

`public T getBack() const`
Returns the last element in the list.

`public void enqueue(T element)`
Inserts the specified element at the end of the list.

`public void dequeue()`
Removes the first element from the list.

`public void pop()`
Removes the last element from the list.

`public void clear()`
Removes all elements from the list.

`public bool contains(T element) const`
Returns **`true`** if you find a node whose data equals the specified element, **`false`** otherwise.

`public void remove(T element)`
Removes the first node you find whose data equals the specified element.

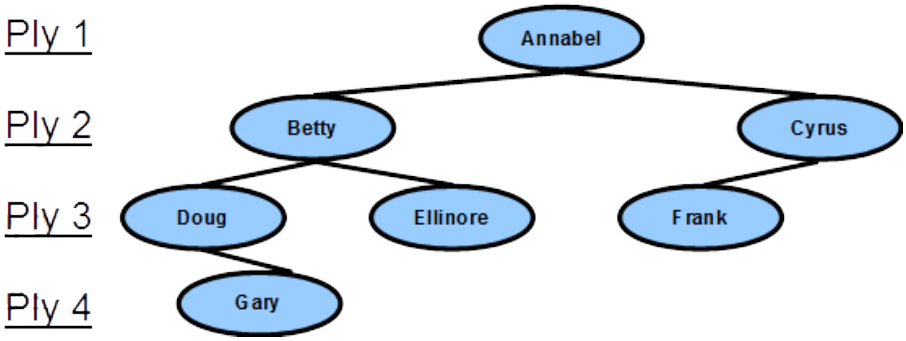
TreeNode

Trees are very common in Artificial Intelligence for both problem solving and decision making. One use is the building of **Behavior Trees**, which will be used as an example in this exercise and covered in more detail later. In a behavior tree, the basic node element is known as a behavior - a way of thinking or acting. All behaviors are said to either succeed (complete successfully) or fail (fail to complete.)

Once you have completed **`LinkedList`**, you can set **`WumpusWorld`** as the start-up project. To construct its Behavior Trees, **`WumpusWorld`** uses the **`TreeNode`** data structure which students will implement.

Structure

Information is often non-linear; it cannot be easily placed "in line" because of more complex relationships that exist between pieces of data. This kind of data is usually difficult to store in arrays, lists, or tables. For example, a descendant family tree would be difficult to implement in a list fashion, because each parent may have several children. Likewise, behaviors in AI are often composed of sub-behaviors in a branching fashion. A tree is made up of a series of tree nodes branches from the **root node**. Each tree node can have zero or more **children**; a node with no children is called a **leaf node**. The **ply level** of a node in a tree is its depth.



A tree holding a descendants. Betty is a child of Annabel, and Gary is a leaf node.

Traversal

As the data stored in trees are non-linear, there are several ways to visit the nodes of a tree. In this project, students will implement three traversals: **depth-first pre-order**, **depth-first post-order**, and **breadth-first**.

Pre-Order

In a (depth-first) pre-order traversal a node's data is processed ***before*** its children's data:

```
def preOrderTraversal(node, function):
    function(node)

    for child in node.children:
        preOrderTraversal(child, function)
```

A pre-order traversal of the example tree would yield:
Annabel, Betty, Doug, Gary, Ellinore, Cyrus, Frank

Post-Order

In a (depth-first) post-order traversal a node's data is processed ***after*** its children's data:

```
def postOrderTraversal(node, function):
    for child in node.children:
        preOrderTraversal(child, function)

    function(node)
```

A post-order traversal of the example tree would yield:
Gary, Doug, Ellinore, Betty, Frank, Cyrus, Annabel

Breadth-First

In a breadth-first traversal a node's data is processed ***before nodes in lower levels***. The techniques involved in a breadth first-traversal are very different from those of depth-first traversals. A queue (or list) and a loop are used to store nodes rather than relying on recursive calls:

```
def breadthFirstTraversal(root, function):
    queue = [ root ]

    while not queue.empty():
        node = queue.popfront()
        function(node)

        for child in node.children:
            queue.append(child)
```

The output of a breadth-first search would look like this:
Annabel, Betty, Cyrus, Doug, Ellinore, Frank, Gary

Implementation

You will implement a generic tree node with the following methods and the following traversal functions:

TreeNode<T>

public **TreeNode<T>()**
Constructor for **TreeNode**. Should store a default-constructed data value and start with no children.

public **TreeNode<T>(T element)**
Constructor for **TreeNode**. Should store **element** as its data value and start with no children.

public const T& **getData()** *const*
Returns a reference to the stored data.

public size_t **getChildCount()** *const*
Returns the number of children of this node.

*public TreeNode<T>** **getChild(size_t index)**
Returns the child node at specified by **index**.

*public TreeNode<T>** **getChild(size_t index)** *const*
Returns the child node at specified by **index**. *Note: this is the const version of this method.*

public void **addChild(TreeNode<T>* child)**
Add **child** to the children of this node.

public void **removeChild(size_t index)**
Remove the child node at specified by **index**.

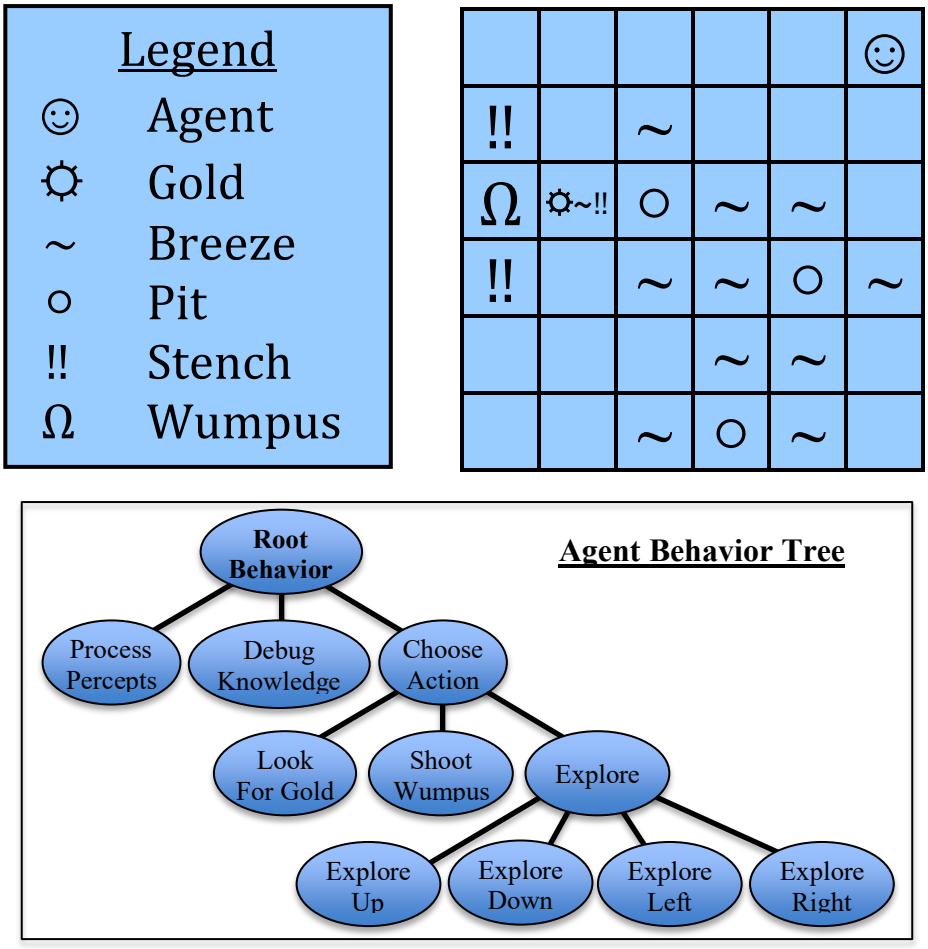
public void **breadthFirstTraverse(void (*dataFunction)(const T))** *const*
Breadth-first traversal starting at this node. Calls **dataFunction** on the element to process it.

public void **preOrderTraverse(void (*dataFunction)(const T))** *const*
Pre-order traversal starting at this node. Calls **dataFunction** on the element to process it.

public void **postOrderTraverse(void (*dataFunction)(const T))** *const*
Post-order traversal starting at this node. Calls **dataFunction** on the element to process it.

Testing

The **`TreeNode`** structure will be tested by using it to build the behavior tree of an AI player for a variant of the "Wumpus World" game. The Wumpus World is an old text-adventure where the player must navigate a dungeon. Within there is a single stash of gold, several bottomless pits which will kill the player, and a mean, stinky monster called the "Wumpus." The objective is to kill the Wumpus and retrieve the gold. The world we'll be working with is outlined below:



Surrounding each pit is a breeze and surrounding the Wumpus is a horrible stench. The player can attack the Wumpus from any adjacent square in order to try and kill it, but the player has only a single arrow to fire. If the player steps on a pit square or the Wumpus square, the player dies.

Using the **`TreeNode`** class, a behavior tree will be built for an agent that will play as the Wumpus World hero. If the class is correctly completed, the project's output should be identical to that of the example.

Submissions

Students will submit a **zip file** named **Ex0.zip** containing the following files at the end of this exercise on Canvas:

- LinkedList.h
- TreeNode.h

Place them in the **root directory** of your zip file, not in a subdirectory. Do not submit any other files.

NOTE: Each test case will be checked for memory leaks! If a test case leaks memory, it will be penalized by up to 30%, so if you leak a lot of memory on every test case, your maximum grade penalty will be 30% overall.

Check out the Visual Studio documentation and the **`_DEBUG`** symbol to track down memory leaks.