Fundamentals of AI - Lab

The Fundamentals of AI Lab Project as a review of basic data structures and an introduction to behavioral modelling using the Behavior Tree concept. In this lab we will examine and implement Queue-Lists as well as Behavior Trees. When implementing this lab **do not** look at **any** code you have written before. ***Do not** copy and paste from another project.* Write the code from scratch and avoid the temptation to seek help from others.

Objectives & Outcomes

Upon completion of this activity, students should be able to...

- Identify the ordering of basic tree traversal algorithms, including depth-first pre-order, depth-first post-order, and breath-first traversals.
- Implement a linked list data structure for the purpose of queueing elements in an AI system.
- Implement sequence and selector behaviors as part of a tree model for agent behavior in an AI system.
- Implement general tree functionality in specialized, custom built tree data structures.

Level of Effort

This activity should take approximately 420m to complete. It will require:

- 45m Research
- 15m Prep & Delivery
- 180m Work

If you find that this activity takes you significantly less or more time than this estimate, please contact me for guidance.

Reading & Resources

Artificial Intelligence for Games *(helpful)*
Chapter(s): 5.4
Page(s): 334-371
Course textbook

Instructions

Code Conventions

The grader of the lab will enforce some or all of these conventions by penalizing any violations.

- Do not change the public, protected, or friend interfaces of an existing class.
- If you absolutely have to add helper methods or variables, you must place them in private scope (not protected scope), and you must preserve const correctness.
- If you plan to declare helper function prototypes outside a class body, choose one of these options instead:
- You may define such functions in the source (.cpp) file where they are needed.
- You may declare them as private static methods.
- Do not const_cast. This implies a deficiency in the project API. File a bug report instead.
- Do not C-style cast in place of a const_cast.

Queue-List

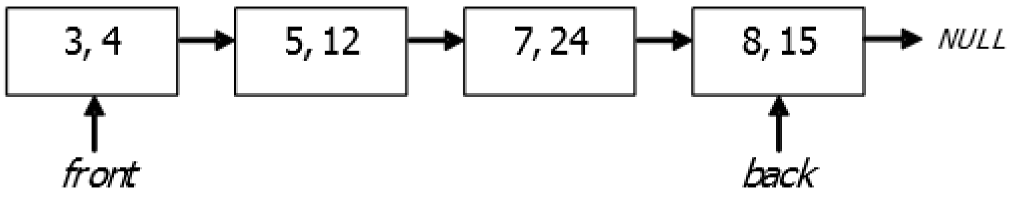
You should be able to complete this part of the lab within a short period of time (30 minutes or less). If you need more time, make sure you review the data structures concepts to prepare for future lab sessions.

Data Structure

A singly linked list is a type of linked list which is linked in only one direction: from front to back. It usually terminates in a pointer to `NULL` (0). 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. In this lab, you are to implement the add/remove behavior as if the list were a queue.

A queue-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 node in the list. The first node in the list is called the front, and the last node is called the back.

We should always keep separate pointers to the front and back of the list.



A queue list with coordinates as data

Implementation

Your `QueueList` class will span two header files. One holds the class definition, including the nested `Node` and `Iterator` classes, the front and back member variables, and the member function declarations. The other header file will hold the member function definitions, which you are to implement.

Todo List

Write the `QueueList::Iterator` operators and `QueueList` methods stated below. You must provide them from scratch. Look inside the primary header file for how the other methods are implemented in case you've forgotten.

QueueList<T>::Iterator

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

`Iterator& operator++()`
Advance the operator one position in the queue. Return this iterator. ***NOTE***: if the iterator has reached the end of the queue (past the last element), point the node member to **NULL**.

QueueList<T>

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

`void enqueue(T element)`
Insert the specified element to the list. Handle the special case when the list is empty.

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

`void dequeue()`
Remove the first element from the list. Handle the special case when it is also the last element.

`void removeAll()`
Remove all elements from the list.

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

`void remove(T element)`
Remove the first node you find whose data equals the specified element. Be sure to update the pointers appropriately. Test your code for the following scenarios:

- If you remove the first node from the list
- If you remove a node from the middle of the list
- If you remove the last node from the list
- If you remove the only node from the list

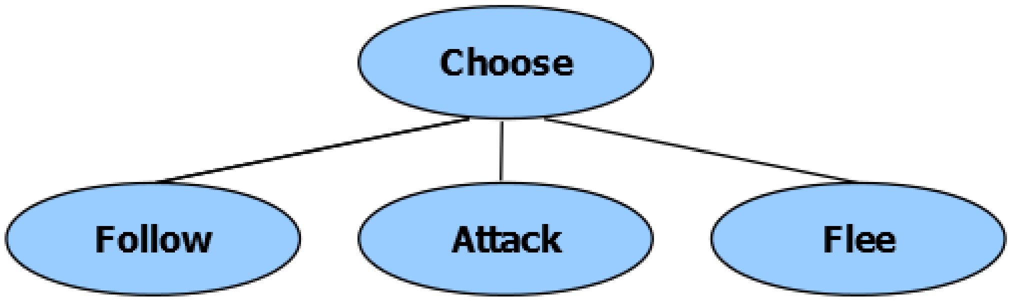
A remove method may look something like this:

```
void remove(data to remove)
{
  Traverse the list, starting at the beginning;

  while (there are nodes to traverse)
  {
    if (the node matches the data)
    {
      remove the node, adjust the list pointers, and exit;
    }
  }
}
```

Behavior Trees

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 lab and covered in more detail later. In a bahavior 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 (fails to complete.)



In this behavior tree, “Choose” is the parent of “Follow”, “Attack”, and “Flee”.

Tree Structure

As a behavior tree is a type of tree, standard tree functions are required for them to function correctly. A ***tree*** is made up of a ***root*** node and its descendants. Each node in the tree may have zero or more ***child*** nodes. A node with no children is called a ***leaf*** node. The ply level of node is its depth.

Tree Traversal

There are several ways to traverse a tree, each with benefits and drawbacks. Today we will examine three: ***depth-first preorder***, ***depth-first postorder***, and ***breadth-first***.

Preorder Traversal

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

```
void traverse(Node node, F function)
{
  function(node);
  traverse(each child of node);
}
```

A preorder traversal of the example tree would yield:
Choose, Follow, Attack, Flee

Postorder Traversal

In a (depth-first) postorder traversal a node's children's data is processed before its own:

```
void traverse(Node node, F function)
{
  traverse(each child of node);
  function(node);
}
```

A postorder traversal of the example tree would yield:
Follow, Attack, Flee, Choose

Breadth-First Traversal

A breadth-first traversal is one that starts from the root and visits all the nodes of each ply before visiting the nodes in the next ply. 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:

```
traverse(Node rootNode, F function)
{
  queue.enqueue(rootNode);
  while (queue.notEmpty())
  {
    Node node = queue.popfront();
    function(node);

    queue.enqueue(each child of node);
  }
}
```

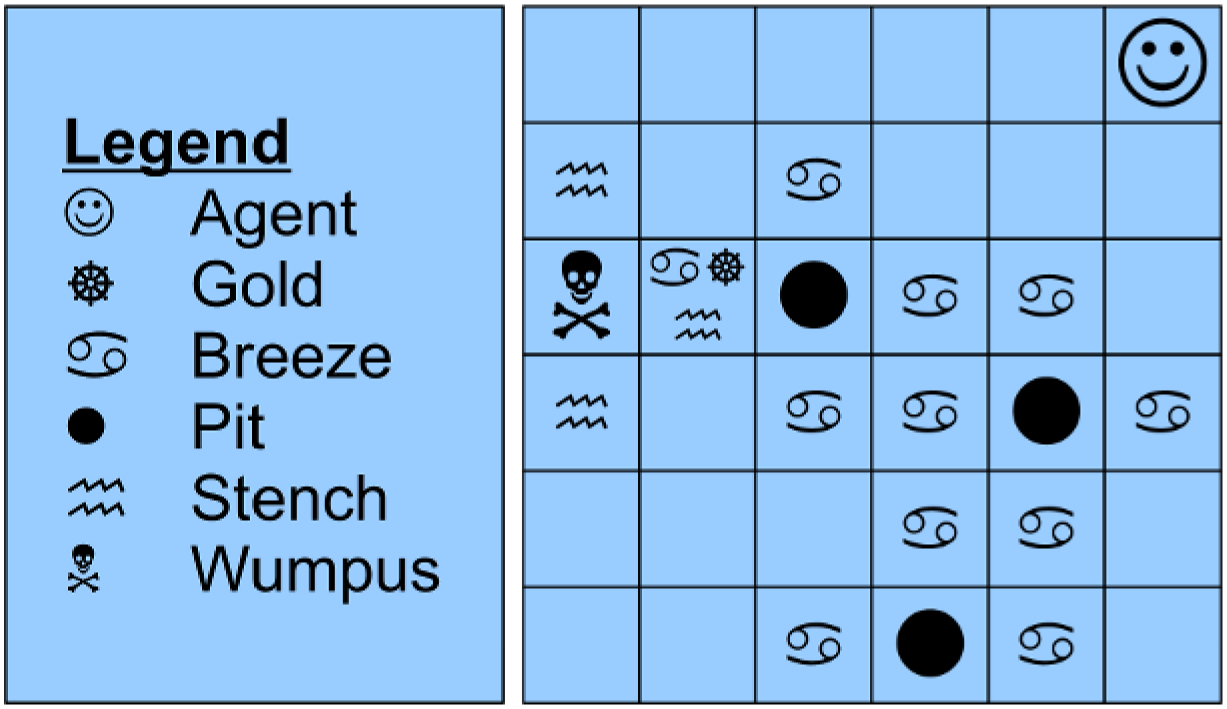
The output of this traversal would look something like this:
Choose, Follow, Attack, Flee

Implementation

In this part of the lab you will be responsible for implementing many functions of behavior trees. To test your implementation, two test trees will be constructed by the project from your classes. This will include standard tree functions & three different traversals on that tree. Your classes will then be tested in a real (though small and simple) game simulation known as the “Wumpus World.”

The Wumpus World

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 classes you complete, a behavior tree will be built for an agent that will play as the Wumpus World hero. If your behavior tree classes are correctly completed, your project's output should be identical to that of the example.

Todo List

Write the following Behavior methods:

```
size_t getChildCount() const
```

Access the number of children this node has.

```
Behavior* getChild(size_t i)
Behavior const* getChild(size_t i) const
```

Return the child at index i in the children container.

```
void addChild(Behavior* child)
```

Add child to the children container and set child's parent to this.

```
void breadthFirstTraverse(void (*dataFunction)(Behavior const*))
const
```

Traverse the node and all its sub-nodes in breadth-first fashion, passing each node to the specified function.

```
void preOrderTraverse(void (*dataFunction)(Behavior const*))
const
```

Traverse the node and all its sub-nodes in pre-order fashion, passing each node to the specified function.

```
void postOrderTraverse(void (*dataFunction)(Behavior const*))
const
```

Traverse the node and all its sub-nodes in post-order fashion, passing each node to the specified function.

Grading

The total grade for this assignment is 50 points. There are 12 test cases total: 4 for QueueList, worth 5 points each; and 8 for basic Tree functionality, worth 3.75 points each.

NOTE: Each test case will be checked for memory leaks! If a test case leaks memory, it will 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. If you need help checking for memory leaks, try using Visual Leak Detector (search for it online).

Deliverables

Students will submit the completed QueueList and BehaviorTree projects.

Deliverable File(s) and Contents

You will upload a compressed (zipped) file named `<lastName>.<firstName>.FundamentalsLab.zip` which should include:

```
QueueList (folder)
        \
        QueueList.h
        QueueList_TODO.h

BehaviorTree (folder)
        \
        Behavior.h
        Behavior_TODO.cpp
        BehaviorTree.vcxproj
        BehaviorTree.vcxproj.filters
```

Turn in Check List:

- Make sure your project compiles and runs in Release mode.

- Check your output against our example, and make sure it matches 100% for full grade.

- Zip up the above two folders with their corresponding files in it accordingly. When we unzip your lab zip file, it should have the above structure exactly, and not with another layer of folder in between. If you have created extra Cpp or Header files(which are generally not necessary), they need to be alongside the existing files within one of those two folders.

- If you have included "vld.h" for memory leak check purposes, please remove or comment it out before turning it in. Otherwise, it will be counted as a late turn-in/resubmission, because it will not compile when we attempt to grade it. We will have to manually edit your files and regrade, and give late a penalty charge.