Stacks and Queues

Introduction

Stacks and queues are actually data structures created from linked lists. If you're not familiar with linked lists, see my Linked Lists Tutorial. This tutorial will actually be rather short because all we need to do is modify the linked list class that we developed in the linked list tutorial. Stacks and queues both just limit the way data is accessed in a linked list.

Stacks

Before we look at the implementation of a stack, let's talk about what a stack is. The best way to see how the stack data structure works is to imagine a stack of plates. If you want to add a plate to the stack, you have to add it to the top. If you want to take a plate from the stack, you have to take if from the top.

This behaviour is know as "Last In, First Out" because the last item you add to a stack is always going to be the first item you get from it. Adding an element to a stack is called "pushing" and removing an element is called "popping". Accessing the top element is called "peeking".

An example of when to use a stack would be for a simple AI character. Let's say the the AI character starts of patrolling. The stack of instructions for the character would look like this:

[Patrol]

Now let's say the AI spots some ammo. We'd like the character to get the ammo, so we push the instruction to get the ammo onto the stack.

[Patrol]->[Get Ammo]

Note that because we're using a stack, the AI will only execute the top instruction. In this case, the AI will go get the ammo. After the AI has the ammo, we pop that instruction from the stack. The stack looks like this again:

[Patrol]

The AI will resume patrolling because that is the instruction at the top of the stack. But what if, while the AI is going to get the ammo, the player wanders by? We obviously want to push an instruction to attack the player onto the stack. This is what the stack would look like:

[Patrol]->[Get Ammo]->[Beat Up the Player]

After the player has been beaten down, **[Beat Up the Player]** gets popped off the stack. The stack always remembers the previous state the AI was in, so it's a good data structure for representing this kind of behaviour. Note that in each frame of our game, all the AI has to do is 'peek' the instruction at the top of the stack to know what to do.

Something to note is that if we were just using a linked list for this, we would use AddTail() when we wanted to push an instruction and we'd use RemoveTail() when we wanted to remove one.

Internally, a stack really is just a linked list. All we have to do is get rid of the functions that access data at the beginning and middle of the list, rename the functions that access the tail, and we have ourselves a stack! No actual coding is really needed, so just quickly read through the changes (I commented them!) in the downloadable source code and check out the test harness.

Click here to download the source code for this part of the tutorial.

Queues

The queue data structure acts just like a line up. Think of a line at a store. The first person to get to the line will be the first person out. If someone else enters the line they enter at the back and get out last. This behaviour is called "First In, First Out" because the first element added to the queue is the first one processed.

If you've ever used some sort of download manager, you've used a queue. As you select the files to download, they get added to the back of the queue. The first item you select will be the first item downloaded. If you're downloading one file at a time, your download manager will only have access to the front of the queue.

Adding an element to a queue is called "enqueue-ing", removing an element from a queue is called "dequeue-ing", and accessing the element at the front is called "peeking".

For a game example of using queues, think of a factory in a real-time strategy game. When you want to build units, the factory uses a queue to keep track of what to build first. Let's say we tell a factory to build a tank. Our queue would look like this:

[Tank]

Now we tell the factory to build an airplane. Instead of overriding the tank, the airplane gets placed in line (remember that the front of the list gets processed first).

[Tank]->[Airplane]

If we tell the factory to build a robot now, we get this:

[Tank]->[Airplane]->[Robot]

When the tank is built, it is dequeued and we get this:

[Airplane]->[Robot]

Once the airplane is built, it is dequeued and it will be the robot's turn:

[Robot]

As with the stack, we just need to modify our linked list class to create a queue. We change the name of `AddTail()` to `Enqueue()`, `RemoveHead()` to `Dequeue()`, and `GetFrontData()` to `Peek()`. Any functions that don't serve a purpose can be removed. Check out the downloadable source code for the full imlementation of a queue.

Click here to download the source code for this part of the tutorial.