Code Design and Data Structures Project Design Document

By Leonard Andrew Spencer, 2017

Contents

[Introduction 3](#_Toc486163368)

[Design Overview 4](#_Toc486163369)

[Game Design 5](#_Toc486163370)

[Introduction 5](#_Toc486163371)

[Controls and Gameplay 5](#_Toc486163372)

[Level Design 5](#_Toc486163373)

[Container Classes 6](#_Toc486163374)

[Introduction 6](#_Toc486163375)

[Array 6](#_Toc486163376)

[Stack 6](#_Toc486163377)

[List 7](#_Toc486163378)

[Queue 7](#_Toc486163379)

[Deque 7](#_Toc486163380)

[Map 7](#_Toc486163381)

[State Machine 9](#_Toc486163382)

[Entities 10](#_Toc486163383)

[Observer System 11](#_Toc486163384)

[Resource Management 12](#_Toc486163385)

[User Interface 13](#_Toc486163386)

# Introduction

# Design Overview

# Game Design

## Introduction

This project runs a short 2D platformer game. The player controls a character who can walk on other objects, jump over or onto obstacles, and will fall when walking off an edge. The player wins by reaching a door at the end of the level.

## Controls and Gameplay

The player uses the WASD keys to control the character

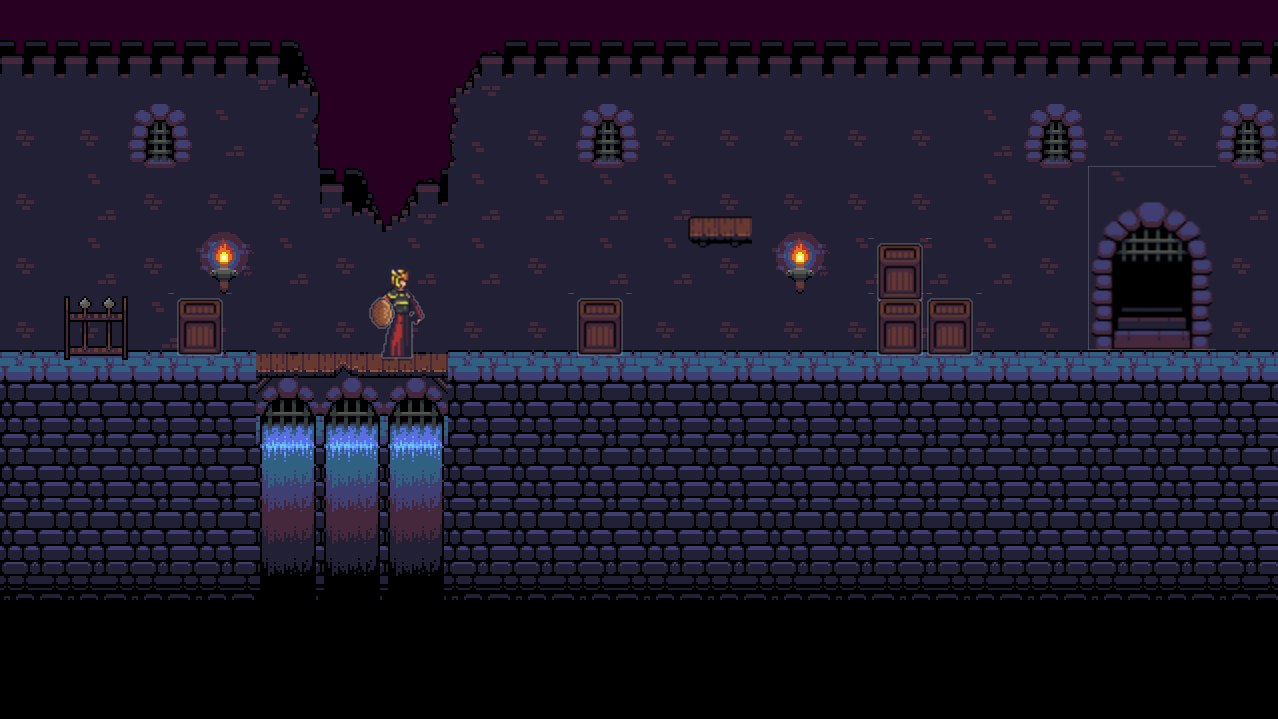
|  |  |
| --- | --- |
| **Key** | **Action** |
| A/D | Move left/right |
| W | Jump |
| S | Crouch |
| ESCAPE | Pause/Unpause |

The character can walk left and right, and move left and right while in the air, but cannot move while crouching. They are able to jump from a standing, crouching, or walking position, but cannot “double jump” while in the air.   
If the character walks off an edge, or is otherwise not supported by some object, they will fall until they land on an object which supports them.

Walking into the door at the end of the level triggers the win screen.

## Level Design

The level was designed to demonstrate basic platforming features. The player needs to reach the door (4) at the right, but the pile of blocks (3) is too high to jump over. So the player has to jump onto the block ahead (1), jump from it onto the platform(2) and then jump onto the pile of blocks(3) to get past them and reach the door(4).



4

3

2

1

Player

Figure Level design

# Container Classes

## Introduction

One of this project’s requirements was that custom container classes, rather than those from the Standard Template Library, were used. Dynamic array (Array), double linked list (List) and search tree map (Map) containers were created to meet this requirement. Stack, Queue, and Deque containers were also created, and implemented in terms of these structures.

Further information on the container classes can be found at <https://github.com/LASpencer/aie-DataStructuresProject/wiki/Containers>

## Array

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Array>

Array is a templated dynamic array, similar to the STL Vector container. Elements can be inserted at or erased from any position, and additional elements can be added to the Array. However, erasing or inserting elements becomes more expensive the more elements there are following this position, as they all have to be moved.  
The Array allocates memory for storing its elements as a contiguous block. By default, it allocates memory for 8 elements. If more elements are inserted than the array has capacity to store, it allocates twice as much memory as it previously had, and copies its elements to the new block. The new block is twice the size of the old one so that the number of allocations required for an array of size n is log n, and the number of copies that need to be made is n log n. If the number of elements required can be estimated in advance, the reserve method can be called to allocate enough memory for those elements beforehand. Also, the shrink\_to\_fit method allows the Array to free unused memory.

The Array class is able to use iterators for some of its functions. As the Array stores its elements as a contiguous block, the pointers to those elements can be used as iterators for the array. This is simple to implement, but does mean that resizing the Array invalidates all existing iterators. More importantly, the Array has templated functions which can take any iterator dereferencing to the type of element contained. These use SFINAE to check the traits of the iterator used. Also, when inserting from a range, the iterator traits are checked to determine if the multipass guarantee holds (that is, if incrementing the iterator does not invalidate its copies). If so, the distance between the first and last iterators is checked, so only one set of copies and resizes needs to take place.

### Stack

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Stack>

The Stack class is a container implemented in terms of an Array. A Stack contains an Array, and each of its methods calls the appropriate methods of the Array. These methods only allow the top element to be added, removed, or accessed.   
There is also a peek method, allowing lower elements in the stack to be accessed as constant references. This allows elements within the stack to be checked easily, while still ensuring it acts as a stack in only allowing changes to the top element.

## List

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/List>

List is a templated double linked list, similar to the STL List container. As a double linked list, elements can be inserted to and erased from any position in constant time. However, accessing an element requires traversing the list from one of the ends to reach it.

The List is implemented using ListNode objects, which contain a value and pointers to the previous and next ListNodes. The List class holds pointers to the first and last ListNodes in the list. These ListNodes cannot be accessed directly, but only through the ListIter class used as an iterator for the List.   
ListIter has references to the List it iterates over, and a ListNode in that list (or nullptr if it’s the end iterator). It dereferences to the value contained in its ListNode. Incrementing and decrementing changes the ListNode to the one pointed to by its current ListNode. Since it contains a reference to the List, the end iterator can be decremented by getting the last ListNode in that list. This could have been done using empty nodes, however several List methods take a ListIter as an argument, and referencing the List in the ListIter allows these methods to check they are only passed ListIters iterating over that List.

### Queue

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Queue>

Queue is a container implemented using a List. It contains a List, and rather than providing that List’s iterators to allow traversal through the list, it only provides access to the value in the front element. Elements can be pushed to the back of the Queue, and popped from the front.

### Deque

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Deque>

Deque is a double ended queue. It is a subclass of Queue, adding methods to enqueue elements at the front (push\_front) and dequeue them from the back (pop\_back).

## Map

<https://github.com/LASpencer/aie-DataStructuresProject/wiki/Map>

Map is an associative container, similar to the STL Map container. It is implemented with a Red-Black self balancing tree. As a binary search tree, order by the element’s key, key-value pairs can be found in logarithmic time.

The search tree is made of TreeNode objects. These contain a key and value, as well as references to their parent and child nodes. A TreeNode has two children: left, which has a lower key, and right, which has a greater key. It also has a flag to determine if it should be considered red or black, for balancing the tree.   
The TreeNode class has several methods for traversing the tree. As well as getting its parent and children, it also get its sibling (returning the child of its parent that isn’t itself), traverse up the tree to the root, and find the node with the previous or next key.

TreeNodes have a recursive copySubtree method, for deep copying a TreeNode. This returns a pointer to a copy of the TreeNode it was called on, with that copy having copies of the original’s descendants.

They also have recursive methods for testing whether their tree is a valid red-black tree.

A TreeNode is responsible for is descendants. When destroyed, it will destroy all its descendants and remove its parent’s reference to itself.

The Map contains a reference to the root node of its search tree, and is responsible for it. By destroying this root node, the rest of the tree will be recursively destroyed.

The Map is also responsible for ensuring the tree remains balanced. On inserting a new key-value pair, the new TreeNode containing it is marked as red. If its parent is also red, this indicates that the tree is unbalanced, and recolourings and rotations are performed to resolve this imbalance. The algorithm for doing so was based on the example given here: <http://www.geeksforgeeks.org/red-black-tree-set-2-insert/>

On erasing a node, if it has more than one child, the node’s successor changes places with it. Importantly, these nodes and those referencing them have their references changed, but the nodes keep the same memory location. This ensures that any iterator referencing the successor remains valid. Then, if the node to erase is black and has a black child (or no children, as nullptrs are considered black), a “double-black” occurs, unbalancing the tree. This is resolved by rotating and recolouring until this extra blackness can be moved to a sibling or falls off the top of the tree. The algorithm for resolving the imbalance was based on the example given here: <http://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/>

Maps have an iterator, MapIter. This is incremented and decremented using the referenced node’s getSuccessor and getPredecessor methods, which have logarithmic time. However, this does mean that MapIter is only invalidated by erasing the node they reference. An iterator using depth-first or breadth-first traversal, rather than in-order, could be incremented in constant time. However, these would be invalidated more easily. Some implementations could even be invalidated by incrementing a copy of the iterator (that is, they would be mutable iterators, rather than forward iterators).

# State Machine

## Introduction

One requirement of this project was that game state would be managed using a state machine.

# Entities

# Observer System

# Resource Management

# User Interface

# Testing