# Data Structures

## Doubly Linked List



A doubly linked list stores data in the form of Nodes. A node contains its data value, plus a pointer to the node in front of it and the node behind it in the list. The list itself contains a pointer to both the head node and the tail node.

Methods:

DoublyLinkedList<T>() – Basic constructor. “T” represents a generic data type, and must be defined on initialization. All instances of “T” in the parameters of the following methods must be the same as the one passed into the constructor.

Bool IsEmpty() – returns true if the doubly linked list is empty (both the head and tail pointers are set to null, and the size value for the list is zero).

Void AddHead(T value) – Adds a new node with value “value” to the front of the list. The head pointer will point to this node, and the size value for the list will increment.

Void AddTail(T value) – Adds a new node with value “value” to the back of the list. The tail pointer will point to this node, and the size value for the list will increment.

T RemoveHead() – Removes the head node from the list and returns its value. The head pointer will point to the next node in the list, and the size value for the list will decrement.

T RemoveTail() – Removes the tail node from the list and returns its value. The tail pointer will point to the previous node in the list, and the size value for the list will decrement.

Node GetHead() – Returns the head node of the list (not its value, but the Node object itself).

Node GetTail() – Returns the tail node of the list (not its value, but the Node object itself).

Int GetSize() – Returns the number of nodes currently in the list.

Void Clear() – Clears the list. Set the head and tail pointers to null and the size value to zero.

### Node

The Node, separately from the Doubly Linked List, has its own methods for operating on it specifically.

Methods:

Node(T value) – Constructor. Creates a new Node, which stores data of type “T” and value “value”.

Node GetNext() – Returns the next Node in the list.

Node GetPrev() – Returns the previous Node in the list.

Void SetNext(Node node) – Sets the current Node’s “next” pointer to the Node passed in as a parameter.

Void SetPrev(Node node) – Sets the current Node’s “previous” ponter to the Node pass in as a parameter.

T GetValue() – Returns the node’s value.

## Stack



A Stack is a Last-In-First-Out data structure. New values are Pushed onto the Stack, and then Popped off the Stack in the reverse order.

Methods:

Stack<T>(int maxCapacity) – Constructor. Takes the maximum number of elements it can contain as an argument. “T” represents a generic data type, and must be explicitly defined upon initialization.

Bool IsEmpty() – Returns true if the Stack contains no values. Otherwise returns false.

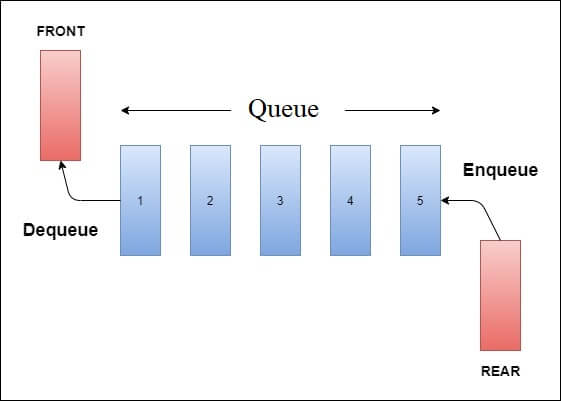
Void Push(T value) – Adds a new element to the top of the Stack with data type “T” and value “value”.

T Pop() – Removes the topmost element from the Stack and returns its value.

Int GetSize() – Returns the number of elements currently in the Stack.

Void Clear() – Clears the Stack.

## Queue



A Queue is a First-In-First-Out data structure. Elements are added to the back of the queue and removed from the front. This Queue is built on top of the Doubly Linked List.

Methods:

Queue<T>() – Constructor. “T” represents a generic data type and must be defined explicitly upon initialization.

Void Enqueue(T value) – Adds a new element with type “T” and value “value” to the rear of the Queue.

T Dequeue() – Removes the frontmost element from the Queue and returns its value.

T Peek() – Returns the value of the frontmost element of the Queue without removing it from the Queue.

Int GetSize() – Returns the number of elements currently in the Queue.

Bool IsEmpty() – Returns true if the Queue contains zero elements. Otherwise returns false.

Void Clear() – Clears the Queue.

# Tiles

## TileEnums

TileEnums defines enumerated lists to be used by other Tile-related classes.

Enums:

Suits { Man, Pin, Sou, Wind, Dragon }

## TileObject

TileObject is an abstract superclass for multiple different subtypes of tiles. It defines the following public values and methods.

Values:

Const int EAST = 1

Const int SOUTH = 2

Const int WEST = 3

Const int NORTH = 4

Const int GREEN = 1

Const int RED = 2

Const int WHITE = 3

Methods:

Int GetValue() – Returns the tile’s numerical value. For number tiles, this will be the face value of the tile (6 Man will return 6, for example). For winds and dragons, this will return the value corresponding to the values of the constants defined above.

TileEnums.Suit GetSuit() – Returns the tile’s suit in the form of an Enum value (defined in the TileEnums class).

## TileFacade

Tile creation is handled through a series of different methods belonging to different classes. For convenience, these methods have been compiled as simply as possible into a single class called TileFacade.

Values:

Const int EAST = 1

Const int SOUTH = 2

Const int WEST = 3

Const int NORTH = 4

Const int GREEN = 1

Const int RED = 2

Const int WHITE = 3

Note: These values are equivalent with those in the TileObject class and can be referenced without issue from either class.

Methods:

TileFacade() – Constructor. Allows for the creation of new tiles.

TileObject CreateTile(int value, TileEnums.Suit suit) – Creates a new TileObject with value “value” and suit “suit”. Returns that TileObject after creating it.

# Wall

## IWall

IWall defines an interface that can be implemented by different Wall objects such that they can be interchangeable with one another. IWall requires the following methods but does not define them.

Methods:

TileObject DrawTileFromEndOfWall()

TileObject DrawTileFromWall()

Void PopulateWall()

Int GetSize()

Void Clear()

### StandardWall

StandardWall implements the IWall interface. It acts as the typical four-player wall, using the typical 138 tiles used in a four-player game.

Methods:

TileObject DrawTileFromEndOfWall() – Removes a TileObject from the back end of the wall and returns the TileObject. Intended to be used for adding new tiles to the dead wall.

TileObject DrawTileFromWall() – Removes a TileObject from the front end of the wall and returns the TileObject. Intended to be used for adding new tiles to the players’ hands.

Void PopulateWall() – Generates all the tiles the wall requires, shuffles them, and puts the into the wall object.

Int GetSize() – Returns the number of TileObjects remaining in the wall.

Void Clear() – Clears the wall.

## IDeadWall

IDeadWall defines an interface that can be implemented by different DeadWall objects such that they can be interchangeable with one another. IDeadWall requires the following methods but does not define them.

Methods:

TileObject DrawTile()

Void PopulateDeadWall()

Void RevealDoraTile()

Void Clear()

List<TileObject> GetDoraIndicators()

List<TileObject> GetRevealedDoraIndicators()

List<TileObject> GetUraDoraIndicators()

List<TileObject> GetDrawableTiles()

### StandardDeadWall

StandardWall implements the IWall interface. It acts as the typical four-player dead wall.

Methods:

StandardDeadWall(IWall wall) – Constructor. Takes the Wall from which it is formed as an argument.

Void PopulateDeadWall() – Builds the dead wall with the fourteen backmost TileObjects in the Wall.

Void RevealDoraTile() – Adds a new TileObject to the private RevealedDoraTiles list.

TileObject DrawTile() – Removes a TileObject from the private DrawableTiles list and returns its value. Adds a TileObject from the end of the Wall to the private ExtraTiles list.

Void Clear() – Clears the dead wall.

List<TileObject> GetDoraIndicators() – Returns the List of potential Dora indicators.

List<TileObject> GetRevealedDoraIndicators() – Returns the List of revealed Dora indicators.

List<TileObject> GetUraDoraIndicators() – Returns the List of Ura Dora indicators.

List<TileObject> GetDrawableTiles() – Returns the List of Drawable tiles.