In this PA (Programming Assignment) you will:

- learn about Stacks, Queues, and Deques
- learn about an application of Queues called breadth-first-search (BFS)
- explore an encoding strategy for images called steganography
- become more skilled at C++ constructs, including the use of basic templates

Part 1: The `Stack`, `Queue`, and `Deque` Classes

The `Deque` Class

You will write a class named `Deque` which is a modification of a doubly ended queue structure. While the standard deque allows for insertion and removal at both ends of a contiguous arrangement of data, our deque will allow insertion and removal at one end, but only removal at the other. For convenience, we will refer to the removal-only end as the left end, and the other as the right.

Your deque should be implemented as follows: The underlying data structure will be a C++ standard vector. Following the convention of the queue we discussed in class, you will allow the data to "float" in the vector, with the following difference: If, upon a removal, you discover that the contiguous block of data (whose size is, say, k) will "fit" in the first k empty positions of the vector, then you should resize down by making a new vector and copying the k pieces of data into that new vector using the push_back function. Additions to the structure can use the standard vector functions, and can only occur at the "right" side of the contiguous data (i.e. the position of largest index). (Note that in this implementation the structure is not "circular"– we do not wrap the data using the modulo of the array size.)

The `Stack` Class

You will write a class named `Stack` that works just like the stack you heard about in lecture, with the addition of the `peek()` function. It is declared in the given file `stack.h`. You will implement it in `stack.cpp`.

Your Stack class must implement all of the methods mentioned in the given code. Please read the documentation in the header file to see what limitations we have placed on your Stack class and what the running times of each function should be.

The `Queue` Class

You will write a class named `Queue` that works just like the queue you heard about in lecture, with the addition of the `peek()` function. It is declared in the given file `queue.h`. You will implement it in `queue.cpp`.

Your `Queue` class must implement all of the methods mentioned in the given code. Please read the documentation in the header file to see what limitations we have placed on your `Queue` class and what the running times of each function should be.

Testing

We have provided a file `testStackQueue.cpp` which includes a few (albeit trivial) test cases that your code should pass if it is correct. To compile this executable, type:

```
make testStackQueue
```

These test cases are deliberately insufficient. We encourage you to augment this file with additional test cases, using the provided ones as examples. In particular, we have not provided test cases for your `Deque` class. You will want to add those yourself!

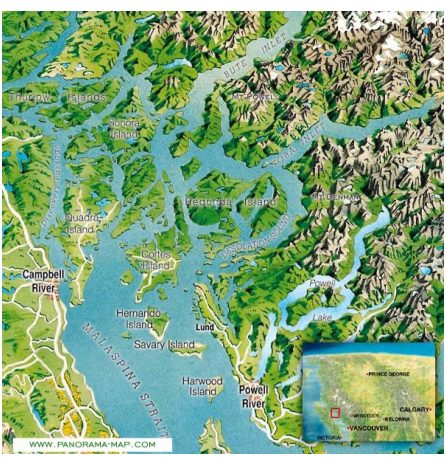
Part 2: Treasure Maps

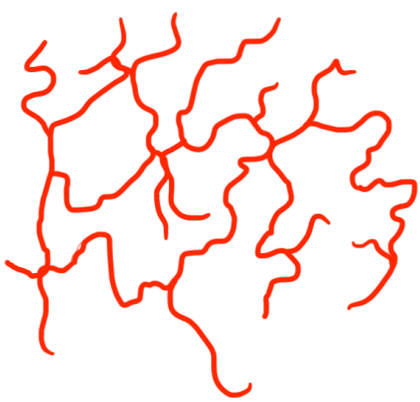
Two summers ago, a company called GoldHunt ran a treasure hunt adventure game, where for an entry fee of $75, Vancouverites could join in the hunt for a $100,000 treasure chest. Participants were given a map and collections of poetic clues that helped the hunters home in on the location of the treasure. It was a scavenger hunt, with a real prize!

The GoldHunt idea was good, but low-tech, and it didn't scale. In this programming assignment you are going to develop an algorithmic tool for creating (and solving) treasure maps by hiding them within images.

Part A -- Creating a Map

In this part of the problem we implement an algorithm for creating a treasure map. The image on the left is a base image onto which we will encode the maze on the right, resulting in an image which is essentially indistinguishable from the first.





A treasure map consists of a base image, a maze image, and a starting position. For this assignment, the location of the starting position will always be given, and the treasure is found at the point (pixel location) whose shortest distance to the start in the maze is longest.

A maze is simply an image containing a collection of points whose pixel colours are equivalent to the pixel colour of the start location, and reachable from the start location via a path of same-coloured pixels. To judge colour equivalence, we use the `==` operator within the `RGBAPixel` class.

The algorithm for embedding the maze into a treasure map is as follows:

1. The treasure map begins as a copy of the base map. The pixels in the treasure map corresponding to the maze image will be adjusted in a way that is not obvious to the naked eye.
2. The start position should be embedded with maze-value 0. (We'll describe embedding

maze-values in the next paragraph.)

3. If a point p has been embedded with maze-value v, then p's compass neighbors who are also in the maze, and whose maze-values are not yet set, should be embedded with maze-value $(v + 1)$.
4. Repeat 3 until all points in the maze have been embedded in the treasure map.

A point p is embedded with maze-value v as follows: Consider the 6 bit binary representation of maze-value $v = b_1b_2b_3b_4b_5b_6$, and the (r, g, b) representation of point p in the base image. In the treasure map, the two lower order bits of the binary representation of r are replaced by b_1b_2. The two lower order bits of the binary representation of g are replaced by b_3b_4. The two lower order bits of the binary representation of b are replaced by b_5b_6.

We have included more detailed instructions for orchestrating the embedding in the treasure map's header file. In particular, we have prescribed the auxiliary structures you will use to support the traversal of the maze.

The image below shows the maze and the start location superimposed on the base map.

Part B -- Finding the Treasure

In this part of the problem you will create a decoder for treasure maps created using the embedding algorithm from part A, and then solve the maze to find the treasure!

The algorithm for finding the maze embedded in the treasure map, given a starting point:

1. The given start point in the treasure map is in the maze.
2. If point p is in the maze, then any compass neighbor of p which is in the treasure map

and whose maze-distance is 1 from p (we'll describe this maze-distance in a moment) is also in the maze.

The maze-distance between points in the treasure map is computed as follows: Suppose a point p has colour (r, g, b) in the treasure map. The maze-value of p is $(r \mod 4) * 16 + (g \mod 4) * 4 + b \mod 4$. That is, we take the 2 lowest order bits from each of the colour channels, and we consider them together to be a number between 0 and 63, inclusive. The maze-distance from p to any of the 4 surrounding pixels is the absolute difference between their maze-values. If the maze-distance from p to a surrounding pixel q is 1, then q is also in the maze. Note that the maze-distances are computed "$\mod 64$", so that $63 + 1 == 0$.

Once you have found the maze, you have to solve it to find the treasure! The treasure is located at the point on the map corresponding to the pixel whose shortest distance to the start, within the maze, is longest. In this context, distance is just the number of pixels on the path in the maze between any pair of pixels. You will search the maze for the shortest distance from the start to each maze point, and then the solution will be the longest of those. The images below show the maze we discovered while decoding, and also the solution path to the treasure.

Finding and solving the maze will use an algorithm similar to the embedding algorithm that we described carefully in part A. We leave it to you to adapt that algorithm for decoding!

The TODO List

Specifications for each function you will write are contained in the given code. The list of functions here should serve as a checklist for completing the assignment.

In `deque.cpp`

- `Deque()`: constructor initializes member variables.
- `void pushR(T newItem)` : Add data to the "right" end of the structure.
- `T popL()` : Remove and return data from the "left" end of the structure.
- `T popR()` : Remove and return data from the "right" end of the structure.
- `T peekL()` : Return value of data from the "left" end of the structure.
- `T peekR()` : Return value of data from the "right" end of the structure.
- `bool isEmpty()` : Return true if there are no data elements in the structure.

In `stack.cpp`

- `void push(T newItem)`: Add data to the "top" of the structure.
- `T pop()`: Remove and return data from the "top" of the structure.

- `T peek()`: Return value of data from the "top" of the structure.
- `bool isEmpty()` : Return true if there are no data elements in the structure.

In `queue.cpp`

- `void enqueue(T newItem)`: Add data to the "rear" of the structure.
- `T dequeue()`: Remove and return data from the "front" of the structure.
- `T peek()`: Return value of data from the "front" of the structure.
- `bool isEmpty()` : Return true if there are no data elements in the structure.

In `treasureMap.cpp`

- `treasureMap(const PNG & baseim, const PNG & mazeim, pair<int,int> s)`: Constructor.
- `PNG renderMap()`: Returns an image with an embedded treasure map.
- `PNG renderMaze()`: Returns an image with a visible representation of the maze.
- `bool good(vector <vector <bool>> & v, pair <int,int> curr, pair<int,int> next)`: Returns true if the next point should be explored.
- `vector<pair<int,int>> neighbors(pair<int,int> curr)`: Returns the neighbors of a given location
- `void setLOB(PNG & im, pair<int,int> loc, int d)`: Sets the lower order bits in the given location.
- `void setGrey(PNG & im, pair<int,int> loc)`: Darkens the colour of the pixel in the given location.

In `decoder.cpp`

- `decoder(const PNG & tm, pair<int,int> s)`: Constructor explores and discovers the solution to the embedded maze.
- `PNG renderSolution()`: Returns an image with a path to the treasure.
- `PNG renderMaze()`: Returns an image with a visible representation of the maze.
- `pair<int,int> findSpot()`: Returns the location of the treasure.
- `int pathLength()`: Returns the length of the path to the treasure.
- `bool good(vector<vector<bool>> & v, pair<int,int> curr, pair<int,int> next)`: Returns true if the next point should be explored.
- `vector<pair<int,int>> neighbors(pair<int,int> curr)`: Returns the neighbors of a given location
- `bool compare(RGBAPixel p, int d)`: Returns true if `p`'s maze value differs from the given value by 1.
- `void setGrey(PNG & im, pair<int,int> loc)`: Darkens the colour of the pixel in the given location.

Implementation Constraints and Advice

For Part 1, we *strongly* encourage you to sketch out small examples to help you understand how the `Deque` class should work.

For Part 2, Treasure Maps, we have specified some functions we think you'll find useful, but you are also welcome to add your own. You'll be submitting both the `.h` and `.cpp` files for that part of the assignment. The same is *not* true for Part 1.

Note that we are not asking you to write any memory management functions for this assignment, because all of our structures are `vector`-based.