```
"How to shuffle a std::vector?"
https://stackoverflow.com/questions/6926433/how-to-shuffle-a-stdvector
Learned how to make a seed using std::default_random_engine with the chrono
library to shuffle vectors randomly.
```



```
   return m_sets;
}
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file disjoint_set.cpp
* @assignment 6: Maze Generation and Disjoint Sets
*/
#include "disjoint_set.h"

DisjointSet::DisjointSet(int sets)
   : m_elems(sets), m_sets(sets) {
   m_rank = new int[sets];
   m_parent = new int[sets];

   // Set default ranks and parents
   for (unsigned int i = 0; i < sets; ++i) {
      m_rank[i] = 0;
      m_parent[i] = i; // Default parent of an element is itself
   }
}

DisjointSet::~DisjointSet() {
   delete[] m_parent;
   delete[] m_rank;
}

void DisjointSet::merge(int elemA, int elemB) {
   // Bounds check on element indices
   if (elemA < 0 || elemB < 0 || elemA >= m_elems || elemB >= m_elems ) {
      throw std::out_of_range("Cannot merge sets");
   }
   // Link the two sets
   link(find(elemA), find(elemB));
}

int DisjointSet::find(int elem) {
   if (elem < 0 || elem >= m_elems) {
      throw std::out_of_range("Cannot find element");
   }
   if (elem != m_parent[elem]) {
      m_parent[elem] = find(m_parent[elem]);
   }
   return m_parent[elem];
}

void DisjointSet::link(int repElemA, int repElemB) {
   if (repElemA == repElemB) {
      return;
   }
   m_sets--;
   if (m_rank[repElemA] > m_rank[repElemB]) {
      m_parent[repElemB] = repElemA;
   }
   else {
      m_parent[repElemA] = repElemB;
      if (m_rank[repElemA] == m_rank[repElemB]) {
         m_rank[repElemB]++;
      }
   }
}

bool DisjointSet::isSameSet(int elemA, int elemB) {
   return find(elemA) == find(elemB);
}

int DisjointSet::getSize() {
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file disjoint_set.h
* @assignment 6: Maze Generation and Disjoint Sets
*/

#pragma once

#include <stdexcept>

/**
* ‘DisjointSet‘ implements a disjoint set with heuristics (path compression and
* union with a ranking system) to improve efficiency. Used in ‘Maze‘ in the maze
* generation algorithim to check if two squares have a path between them.
*/
class DisjointSet {
public:
   /**
   * ‘DisjointSet‘ constructor.
   * @param ‘sets‘ number of sets to instantiate
   */
   DisjointSet(int sets);

   /**
   * ‘DisjointSet‘ deconstructor.
   */
   ~DisjointSet();

   /**
   * Merges two sets given elements in these sets by first searching for the
   * representative elements.
   * @param ‘elemA‘ indice of element in first set
   * @param ‘elemB‘ indice of element in second set
   */
   void merge(int elemA, int elemB);

   /**
   * Checks if two elements are of the same set.
   * @param ‘elemA‘ indice of first element
   * @param ‘elemB‘ indice of second element
   * @return true if ‘elemA‘ and ‘elemB‘ have the same represenative element
   * (are of the same set), else false
   */
   bool isSameSet(int elemA, int elemB);

   /**
   * Finds the remaining number of sets.
   * @return the number of sets
   */
   int getSize();

   /**
   * Finds the representative element of the set containing an element. As it
   * searches for the representative element it self heals the data structure,
   * connecting elements directly to the representative element.
   * @param ‘elem‘ element in set to find its representative element
   */
   int find(int elem);

private:
   /**
   * Unions two sets from two given representative elements and updates their
   * ranks accordingly.
   * @param ‘repElemA‘ first representative element of set to link
```

```
    * @param ‘repElemB‘ second representative element of set to link
    */
    void link(int repElemA, int repElemB);

    int* m_rank; // Array of ranks (upper bounded height of a set)
    int* m_parent; // Array of parent element for each element
    int m_elems; // Number of elements (initially the number of sets)
    int m_sets; // The number of sets; decremented during a merge
};
```



```
p6: main.o disjoint_set.o maze.o
    g++ -o p6 -std=c++20 main.o disjoint_set.o maze.o

main.o: main.cpp
    g++ -c -std=c++20 main.cpp

disjoint_set.o: disjoint_set.h disjoint_set.cpp
    g++ -c -std=c++20 disjoint_set.cpp

maze.o: maze.h maze.cpp
    g++ -c -std=c++20 maze.cpp

clean:
    rm -f p6 *.o *~
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file main.cpp
* @assignment 6: Maze Generation with Disjoint Sets
*/

/*
* This program creates n*n sized mazes where n is given as a command line
* parameters and outputs the maze to STDOUT. The maze generation algorthim
* begins with all walls "filled" in the maze matrice then breaks them down. A
* disjoint set is used to stop the removal of a wall if there is already a path
* from that tile to the deletion candidates. I tested this program by creating
* mazes and inspecting them visually with Jeff Bergamini’s maze viewer
* and by checking for memory leaks. The program appears to be working to all
* specs.
*/

#include <iostream>

#include "maze.h"

int main(int argc, char** argv) {
    if (argc < 2) {
        throw std::invalid_argument("Missing command line param for maze size");
    }

    Maze maze(std::stoi(argv[1]));
    std::cout << maze;
}
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file maze.cpp
* @assignment 6: Maze Generation and Disjoint Sets
*/

#include "maze.h"

Maze::Maze(const int edge) {
    if (edge < 3) {
        throw std::invalid_argument("Maze cannot have side length less than 3");
    }

    m_area = edge * edge;
    m_sideLength = edge;
    m_set = new DisjointSet(m_area);

    generateSeed();
    scramble();
    initalizeWalls();
    removeWalls();
}

Maze::~Maze() {
    delete m_set;
}

void Maze::removeWalls() {
    while (m_set->getSize() > 1) {
        for (size_t i = 0; i < m_area; i++) {
            processCandidates(m_scrambles[i]);
        }
    }
}

void Maze::scramble() {
    // Fills vector with indices
    for (size_t i = 0; i < m_area; i++) {
        m_scrambles.push_back(i);
    }

    // Shuffles ‘m_scrambles‘
    std::shuffle(m_scrambles.begin(), m_scrambles.end(), m_seed);
}

std::ostream& operator<<(std::ostream& os, Maze& maze) {
    int totalIndex = 0;
    for (size_t i = 0; i < maze.m_sideLength; i++) {
        for (size_t j = 0; j < maze.m_sideLength; j++) {
            os << std::hex << (int)maze.m_squares[totalIndex];
            totalIndex++;
        }
        os << std::endl;
    }
    return os;

}

void Maze::processCandidates(uint index) {
    std::shuffle(m_candidates.begin(), m_candidates.end(), m_seed);
    for (size_t i = 0; i < m_candidates.size(); i++) {
        switch(m_candidates[i]) {
            case m_left: {
                int indexOfLeft = toLeft(index);
```

```
            if (existsLeft(index) && !m_set->isSameSet(index, indexOfLeft)) {
               breakLeft(index);
               breakRight(indexOfLeft);
               m_set->merge(index, indexOfLeft);
            }
            break;
         }
         case m_right: {
            int indexOfRight = toRight(index);
            if (existsRight(index) && !m_set->isSameSet(index, indexOfRight)) {
               breakRight(index);
               breakLeft(indexOfRight);
               m_set->merge(index, indexOfRight);
            }
            break;
         }
         case m_down: {
            int indexOfDown = toDown(index);
            if (existsDown(index) && !m_set->isSameSet(index, indexOfDown)) {
               breakDown(index);
               breakUp(indexOfDown);
               m_set->merge(index, indexOfDown);
            }
            break;
         }
         case m_up: {
            int indexOfUp = toUp(index);
            if (existsUp(index) && !m_set->isSameSet(index, indexOfUp)) {
               breakUp(index);
               breakDown(indexOfUp);
               m_set->merge(index, indexOfUp);
            }
            break;
         }
      }
   }
}

void Maze::generateSeed() {
   // std::duration::chrono object from current sys time to sys epoch
   auto epochDuration = std::chrono::system_clock::now().time_since_epoch();

   // Number of ticks in this duration
   auto seed = epochDuration.count();

   // Meets "UniformRandomBitGenerator" requirements for std::shuffle
   m_seed.seed(seed);
}

void Maze::initalizeWalls() {
   m_squares.push_back(m_entrance);
   for (size_t i = 0; i < m_area - 2; i++) {
      m_squares.push_back(m_block);
   }
   m_squares.push_back(m_exit);
}
```



```
   */
   void processCandidates(uint index);

   /**
   * Propogates ‘m_squares‘ vector with surrounding walls and makes the
   * entrance/exit.
   */
   void initalizeWalls();

   /**
   * Creates a random seed using system time for shuffling vectors and stores it
   * in ‘m_seed‘.
   */
   void generateSeed();

   /**
   * Finds the index of the square to the right of a square.
   * @param ‘index‘ index of square
   * @return index of square to the right of ‘index‘
   */
   int toRight(int index) { return index + 1; }

   /**
   * Finds the index of the square to the left of a square.
   * @param ‘index‘ index of square
   * @return index of square to the left of ‘index‘
   */
   int toLeft(int index) { return index - 1; }

   /**
   * Finds the index of the square above a square.
   * @param ‘index‘ index of square
   * @return index of square above ‘index‘
   */
   int toUp(int index) { return index - m_sideLength; }

   /**
   * Finds the index of the square below a square.
   * @param ‘index‘ index of square
   * @return index of square below ‘index‘
   */
   int toDown(int index) { return index + m_sideLength; }

   /**
   * Removes the left wall of a square.
   * @param ‘index‘ index of square
   */
   void breakLeft(uint index) { m_squares[index] &= 0xb; }

   /**
   * Removes the right wall of a square.
   * @param ‘index‘ index of square
   */
   void breakRight(uint index) { m_squares[index] &= 0xe; }

   /**
   * Removes the wall above a square.
   * @param ‘index‘ index of square
   */
   void breakUp(uint index) { m_squares[index] &= 0x7; }

   /**
   * Removes the wall below a square.
   * @param ‘index‘ index of square
   */
   void breakDown(uint index) { m_squares[index] &= 0xd; }
```



```
/*
* @author Cole Van Verth
* @pengo cverth
* @email colevanverth@gmail.com
* @file maze.h
* @assignment 6: Maze Generation and Disjoint Sets
*/

#pragma once

#include <vector>
#include <algorithm>
#include <chrono>
#include <string>
#include <random>
#include <ostream>

#include "disjoint_set.h"

/**
* ‘Maze‘ generates an n*n maze on instantiation by randomly selecting squares in
* the Maze and knocking down walls with adjacent squares when there is not a
* path between them already.
*/
class Maze {
public:
   /**
   * ‘Maze‘ constructor. Generates the maze.
   * @param ‘edge‘ height and width of the maze
   */
   Maze(const int edge);

   /**
   * ‘Maze‘ destructor.
   */
   ~Maze();

   /**
   * Outputs the contents of the ‘Maze‘.
   * @return ostream reference containing maze entries; rows are seperated by
   * whitespace
   */
   friend std::ostream& operator<<(std::ostream& os, Maze& maze);

private:

   /**
   * Iterates through ‘m_squares‘ using indices from ‘m_scrambles‘ to find a
   * random square, then calls ‘processCandidates‘ to attempt a wall removal if
   * applicable. Repeats this iteration until there is only one set remaining in
   * ‘m_set‘ (trivial with this particular maze generation algorithim but a
   * safeguard nonetheless).
   *
   */
   void removeWalls();

   /**
   * Scrambles ‘m_scrambles‘ vector containing indices of all squares,
   * allowing random connection of squares to be efficient.
   */
   void scramble();

   /**
   * Removes a wall in between a square and a randomly picked adjacent square if
   * it is in bounds and both squares are not in the same set.
   * @param ‘index‘ index of ‘square‘ in ‘m_squares‘
```



```
   /**
   * Checks if a square exists to the right of a square.
   * @param ‘index‘ index of square
   * @return true if square exists to right of ‘index‘, else false
   */
   bool existsRight(int index) { return (index + 1) % m_sideLength; }

   /**
   * Checks if a square exists to the left of a square.
   * @param ‘index‘ index of square
   * @return true if square exists to the left of ‘index‘, else false
   */
   bool existsLeft(int index) { return index % m_sideLength; }

   /**
   * Checks if a square exists above a square.
   * @param ‘index‘ index of square
   * @return true if square exists above ‘index‘, else false
   */
   bool existsUp(int index) { return !(index - m_sideLength < 0); }

   /**
   * Checks if a square exists below a square.
   * @param ‘index‘ index of square
   * @return true if squares exists below ‘index‘, else false
   */
   bool existsDown(int index) { return !(index + m_sideLength >= m_area); }

   int m_area; // Area of the maze

   int m_sideLength; // Side length of the maze

   DisjointSet* m_set; // Disjoint set containing the indices of squares

   std::vector<uint8_t> m_squares; // Array of wall info for the squares

   std::vector<uint> m_scrambles; // Scrambled indices of squares

   // Seed for shuffling vectors randomly
   std::default_random_engine m_seed = std::default_random_engine {};

   // Adjacent deletion candidates
   enum m_candidate : uint8_t {m_up = 0, m_left, m_right, m_down};

   // Vector of adjacent deletion candidates
   std::vector<m_candidate> m_candidates {m_up, m_left, m_right, m_down};

   const uint8_t m_block = 0xf; // Walls on every edge

   const uint8_t m_entrance = 0xb; // No wall on left

   const uint8_t m_exit = 0xe; // No wall on right
};
```