EECS 281 – Fall 2020
Programming Project 2
Mine All Mine
(Priority Queues)

Due Tuesday, October 13, 11:59pm

Table of Contents

Version 09-29-20

Credits: David Paoletti, Marcus Darden, Ian Lai, Spencer Kim, Brian Wang, Nathan Moos, Katie Matton

Project Identifier

You MUST include the project identifier at the top of every file you submit to the autograder as a comment. This includes all source files, header files, and your Makefile (in the first TODO block):

```
// Project Identifier: 19034C8F3B1196BF8E0C6E1C0F973D2FD550B88F
```

Overview

This project is broken into two parts (A and B). Part A is your `main()`, using the STL `std::priority_queue<>`; part B is you implementing several different versions of a priority queue.

Project Goals

- Understand and implement several kinds of priority queues.
- Be able to independently read, learn about, and implement an unfamiliar data structure.
- Be able to develop efficient data structures and algorithms.
- Implement an interface that uses templated “generic” code.
- Implement an interface that uses inheritance and basic dynamic polymorphism.
- Become more proficient at testing and debugging your code.

Part A: Gold Mining

For Part A, you should always use `std::priority_queue<>`, not your templates from Part B. In Part A you probably do not need or want to use pointers; in Part B you will have to (for the Pairing Heap).

Overview

You are an adventurous gold miner. However, in your avarice you've ignored several safe-tunneling practices, and a recent earthquake has left you trapped! Luckily, out of paranoia, you always carry ridiculous quantities of dynamite sticks with you. You need to blast your way out and make the most of each dynamite stick by blasting the piles of rubble which take the fewest sticks to destroy.

Breaking Out of the Mine

The mine you are trapped in can be represented with a 2-dimensional grid. There are 2 types of tiles:

- Tiles containing rubble.
    - Think of cleared tiles as containing 0 rubble. A tile in the mine could contain 0 before you clear it, that means that it never contained rubble.
- Tiles containing TNT.

You (the miner) start on a specified tile. At every iteration, you will attempt to blast away the "easiest" tile you can "access", until you escape!

Definition of Discovered

The mine is very dark and you cannot see very far. When standing on a clear tile there are only four tiles that you can discover from your current tile: up, down, left and right (except for edge-of-map conditions). This is true in every case except for the start of the simulation, when you can only discover the starting tile. Adding tiles to the primary priority queue counts as "discovering" them. They can never be discovered (added to the primary PQ) twice.

Definition of Investigating

The priority queue will always tell you what your "next" tile will be. Investigating is taking the "next" tile from the priority queue and making it your "current" location. When you investigate a tile, you must clear it if it contains rubble or TNT. After clearing the tile, you can then discover any of the four tiles visible from your current location, add them to the priority queue, etc. You use the priority queue to remember tiles that you have discovered, but have not yet investigated.

Definition of Easiest Tile

The easiest tile you can reach is defined as follows, in the stated order:

1. Any TNT tile you can reach.
2. The lowest rubble-value tile you can reach.

Tie-breaking

In the event of ties (two TNT tiles or two rubble tiles with the same rubble-value):

1. Investigate the tile with the lower column coordinate first.
2. If the tiles have the same column coordinate, investigate the tile with the lower row coordinate.

Clearing Tiles

When clearing away rubble tiles, the tile turns into a cleared tile.
When clearing away TNT tiles, the following happens:

- The TNT tile becomes a cleared tile.
- All tiles touching the TNT tile are also “cleared”
    - If a TNT tile is touching another TNT tile, this will cause a chain explosion.
    - Diagonals are not considered for TNT explosions.

Definition of Escape
The miner escapes when their current tile has been cleared and is at the edge of the grid.

Example

In the following example, you start at position `[1,2]` (row 1, column 2). The tiles that the miner has investigated are **red** and the tiles that the miner has discovered are **blue**. The tile that the miner just cleared is **red and underlined**. Note: all cleared tiles will be 0, because you must clear a tile as part of investigating it. Positive integers signify rubble tiles (0 is a clear tile) and the value -1 signifies a TNT tile.

Please note: This example mine is for illustrative purposes and is not the same as the input file described in the **Full I/O Example** section**.** To make things clearer, there are **bold** indices on the edges that refer to row and column number - they are not a part of the actual input file.

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100  10  20  15
2  20  15   5   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

At the first iteration, the only tile that can be discovered is the starting location, `[1,2]`. The miner clears this tile and then can discover other tiles.

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100   0  20  15
2  20  15   5   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

Next, the miner will investigate and clear tile `[2,2]` because there are no TNT tiles in view and it has the lowest rubble-value. Clearing that tile allows us to discover more tiles!

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100   0  20  15
2  20  15   0   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

The tiles [2,1], [2, 3], and [3,2] are now discoverable (but still uninvestigated). The miner then investigates tile `[3,2]`. Both this tile and tile `[2,3]` have an equally low level of difficulty. The tile `[3,2]` is chosen because its lower column value of `2` breaks the tie.

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100   0  20  15
2  20  15   0   0  20
3  20  20   0  -1 100
4 100  -1  -1  20  20
```

At this point, there are two TNT tiles which could be investigated next! However, due to the tie-breaking rules, the miner will choose to blow up the TNT at `[4,2]` instead of `[3,3]` (this choice is made because it is at the top of the priority queue). Blowing up the TNT tile at `[4,2]` clears all the tiles touching it, creating a chain reaction with the TNT tile at `[4,1]`. After all the TNT explosions have been resolved, the grid looks like the following:

```
    0   1   2   3   4
0  20  20  20  20  20
1  20 100   0  20  15
2  20  15   0   0  20
3  20   0   0  -1 100
4   0   0   0   0  20
```

An explosion makes tiles *discovered* but does not *investigate* the tile (or discover around it). So, for example, [3, 1] was blown up by TNT and thus discovered. Therefore it is a candidate to be investigated next, but [3, 0] (adjacent to it) is not. Since the current location is `[4,2]`, and this tile is now cleared and on the edge of the map, we have escaped!

TNT Explosions

As you will see in the **Output** section, you need to keep track of the order in which tiles are cleared.

When a TNT tile detonates, **all tiles** that are cleared as a result of the TNT detonation (including chain reactions) are cleared in order from "easiest" to "hardest" (as defined in **Breaking Out of the Mine**, including tie-breaker rules). These tiles should also be discovered. As stated in **Breaking Out of the Mine**, do **NOT** consider diagonals; even TNT only destroys rubble piles up, down, left and right of it.

When a TNT detonation occurs, you should use a priority queue to determine detonation order. Push all the detonated tiles into a separate TNT priority queue, and then pop them out in priority order. You need to use some type of priority queue, because TNT blows up other TNT first, followed by smaller piles of rubble, etc.

Notice that, as you progress through your TNT priority queue, you may blow up a tile that is already waiting in your primary priority queue. If this happens, you have to make sure that the new rubble value of 0 comes out of the primary PQ sooner than the old, non-0 value. Think about how it will be possible for the primary PQ to actually know that a tile has changed! What if there were two entries for the same tile, and you ignore the second one? See the **Full I/O Example** for more details.

Command Line

Your program **`MineEscape`** should take the following case-sensitive command-line options:

- `-h, --help`
  - Print a description of your executable and `exit(0)`.
- `-s, --stats N`
  - An optional flag that tells the program it should print extra summarization statistics upon termination. This optional flag has a required argument N. Details are covered in the **Output** section.
- `-m, --median`
  - An optional flag that tells the program it should print the median difficulty of clearing a rubble tile that the miner has seen. Details are covered in the **Output** section.
- `-v, --verbose`
  - An optional flag that tells the program it should print out every rubble value (or TNT) as a tile is being cleared. Details are covered in the **Output** section.

Examples of legal command lines:

- `./MineEscape -v < infile.txt`
- `./MineEscape --stats 15 < infile.txt > outfile.txt`

We will not be error checking your command-line handling, but we expect your program to accept any valid combination of input parameters.

Input and Output Redirection

In order to read an input file, you will use input redirection, just like you did in project 1. If you want to send your output to a file, you can also use output redirection, as seen in one of the command line

examples above. The < redirects the file specified by the next command line argument to be the standard input (`stdin/cin`) for the program. The > redirects the standard output (`stdout/cout`) of the program to be printed to the file specified by the next command line argument.

Input

Settings will be given from an input file, **'MINEFILE'** (this input file will not necessarily be named **'MINEFILE'**). There will be two different types of input: map (M) and pseudo-random (R). Map input is for your personal debugging, but pseudorandom allows easier testing of large grids.

Map input mode (M)

Input will be formatted as follows:

- '`M`' - A single character indicating that this file is in map format.
- '`Size`' - Positive integer number that specifies the size of the square grid (20 means a grid with 20 rows and 20 columns).
- '`Start`' - Coordinate indicating the starting location, row followed by column (two integers).

Map input consists of a map of all the tiles in the mine. Each tile will be separated from other tiles on the same line with whitespace (any number of spaces or tabs). There are 2 types of tiles:

- Rubble tiles which are signified by an integer between 0 and 999 (0 means the tile is already clear of rubble).
- TNT tiles, which are represented by the integer -1.

Tiles are indexed as follows:

- The tile in the top left corner is at location [0,0].
- The tile in the bottom right corner is at location [Size-1,Size-1].

Example of M input (starting location is at [1,2], or row 1 column 2, underlined):

```
M
Size: 5
Start: 1 2
    9    0    9    3    3
    6    9    6    8    3
    9    4    1    9    0
    2    0   -1   -1    9
    8    3    9    7    5
```

Pseudorandom Mode (R)

Input will be formatted as follows:

- `'R'` - character indicating that this file is in pseudorandom format.
- `'Size'` - Number that specifies the size of the square grid (unsigned integer)
- `'Start'` - Coordinate indicating the starting location (two unsigned integers, row first).
- `'Seed'` - Number used to seed the random number generator (unsigned integer)
- `'Max_Rubble'` - The max rubble value a tile can have (unsigned integer)
- `'TNT'` - Chance that a generated tile will be a TNT tile (20 = 1 in 20 chance of a given tile being a TNT tile, 0 = no chance of TNT; also an unsigned integer)

Example of R input:

```
R
Size: 5
Start: 1 2
Seed: 0
Max_Rubble: 10
TNT: 5
```

Generating your grid in R mode:
Included in Canvas with the project spec are the files **P2random.h** and **P2random.cpp** that contain definitions for the following function:

```
void P2random::PR_init(std::stringstream& ss, unsigned int floor_size,
                       unsigned int seed, unsigned int max_rubble,
                       unsigned int tnt_chance);
```

The function `P2random::PR_init(...)` will set the contents of the stringstream argument (`ss`) so that you can use it just like you would `cin` for M input mode. You may find the following (incomplete) C++ code helpful in reducing code duplication. Remember, **DO NOT** copy/paste from a PDF file, retype the code by hand!

```
stringstream ss;
if (mode == "R") {
    // Read some variables from cin
    P2random::PR_init(ss, floor_size, seed, max_rubble, tnt_chance);
} // if

// If map mode is on, read from cin;
// otherwise read from the stringstream
istream &inputStream = (mode == "M") ? cin : ss;
```

From this point on, read from `inputStream`; it doesn't matter whether the mode is M or R!

The example R and M input files given above are equivalent. That is, *both* should generate the exact same map!

Errors you must check for
You must make sure that the input file describes a valid mine by checking for each of the following:

- The character on the first line of the file is either a '`M`' or an '`R`'
- The coordinate specifying the '`Start`' is a valid location given the '`Size`' of the grid

If you detect invalid input at any time during the program, print a helpful message to `cerr` and `exit(1)`. **You do not need to check for input errors not explicitly mentioned here.**

Look in the Part A samples file, `Error_messages.txt`: if your program performs an `exit(1)` and produces one of those error messages for a valid test case, we'll show you your error output (to help you debug).

Output

Default Output

After completing the escape, your program should **always** print a summary line:

`Cleared` **`<NUM>`** `tiles containing` **`<AMOUNT>`** `rubble and escaped.`

`<NUM>`	the number of tiles cleared. This number **does** include tiles cleared by TNT, but does not include the TNT tile itself.
`<AMOUNT>`	the total rubble cleared. This number **does** include rubble cleared by TNT, but clearing (detonating) the TNT tile itself counts as 0 rubble cleared.

Verbose Output

During the program execution, if the `-v` or `--verbose` switch is present, each time you clear a tile whose rubble amount is greater than 0, you should print:

`Cleared:` **`<RUBBLE>`** `at [`**`<ROW>,<COL>`**`]`

`<RUBBLE>`	the amount of rubble being cleared
`<ROW> <COL>`	the coordinates of the tile being cleared

Any TNT tiles blown up should display:

`TNT explosion at` **`[<ROW>,<COL>]`**`!`

Any rubble tiles cleared by TNT should add the words "by TNT" to the cleared line; the general format is:

`Cleared by TNT: <RUBBLE> at [<ROW>,<COL>]`

The verbose mode output is produced as tiles are cleared, before the summary line. Consider getting verbose mode working early: it involves very little code, and will help you debug if you have any incorrect output.

Median Output

During the program execution, if the `-m` or `--median` switch is present, each time you clear a tile, you should print:

`Median difficulty of clearing rubble is: <MEDIAN>`

<MEDIAN> The median value of rubble cleared so far. This includes rubble tiles cleared by TNT, but does not include TNT tiles (-1 values should not be included in this calculation). Tiles that start out clear (rubble value 0) are not included here.

Median output must display 2 decimal digits. You can use:

```
cout << std::fixed << std::setprecision(2);
```

at the beginning of your program to guarantee this.

Stats Output

After printing the summary line, if the `-s` or `--stats` option is specified print the following output (where **N** is the argument given to the -s flag on the command line):

A. The line, **"**`First tiles cleared:`**"** (without quotes) followed by the first **N** tiles cleared in the following format:

`<TILE_TYPE> at [<ROW>,<COL>]`

<TILE_TYPE> the type of the tile being cleared. If it is a rubble tile, this is the rubble amount. If this is a TNT tile, print **"**`TNT`**"** without the quotes

<ROW>,<COL> the coordinates of the tile being cleared.

Remember: when a TNT tile detonates or when there is a chain reaction, all the tiles are cleared from easiest to hardest. Refer back to **TNT Explosions** for more details.

B. The line, **"**`Last tiles cleared:`**"** (without quotes) followed by the last **N** tiles cleared in the same format as part **A**. The **last** tile cleared should be printed first, followed by the second last, etc.

C. The line, **"**`Easiest tiles cleared:`**"** (without quotes) followed by the **N** easiest tiles you blew up in the same format as part **A** in *descending order* (easiest tile followed by next easiest tile)

D. The line, **"**`Hardest tiles cleared:`**"** (without quotes)' followed by the **N** hardest tiles you blew up in the same format as part **A**. in *ascending order* (hardest tile followed by next hardest tile)

If you have cleared less than N tiles, then simply print as many as you can. Tiles that start out clear (rubble value 0) are not included here.

Full I/O Example

Input (`spec-M.txt` and `spec-R.txt`):

Equivalent input files (the R input file will generate a grid that looks just like the M input file):

```
M
Size: 5
Start: 1 2
    9    0    9    3    3
    6    9    6    8    3
    9    4    1    9    0
    2    0   -1   -1    9
    8    3    9    7    5
```

```
R
Size: 5
Start: 1 2
Seed: 0
Max_Rubble: 10
TNT: 5
```

Output:

We ran the program with this command line: **`./MineEscape -v -m -s 10 < spec-M.txt`**
The output is shown below, with the verbose mode highlighted in blue. The median mode output is easy to identify, and statistics come after the "Cleared 6 tiles" summary line.

```
Cleared: 6 at [1,2]
Median difficulty of clearing rubble is: 6.00
Cleared: 1 at [2,2]
Median difficulty of clearing rubble is: 3.50
TNT explosion at [3,2]!
TNT explosion at [3,3]!
Cleared by TNT: 7 at [4,3]
Median difficulty of clearing rubble is: 6.00
Cleared by TNT: 9 at [4,2]
Median difficulty of clearing rubble is: 6.50
```

```
Cleared by TNT: 9 at [2,3]
Median difficulty of clearing rubble is: 7.00
Cleared by TNT: 9 at [3,4]
Median difficulty of clearing rubble is: 8.00
Cleared 6 tiles containing 41 rubble and escaped.
First tiles cleared:
6 at [1,2]
1 at [2,2]
TNT at [3,2]
TNT at [3,3]
7 at [4,3]
9 at [4,2]
9 at [2,3]
9 at [3,4]
Last tiles cleared:
9 at [3,4]
9 at [2,3]
9 at [4,2]
7 at [4,3]
TNT at [3,3]
TNT at [3,2]
1 at [2,2]
6 at [1,2]
Easiest tiles cleared:
TNT at [3,2]
TNT at [3,3]
1 at [2,2]
6 at [1,2]
7 at [4,3]
9 at [4,2]
9 at [2,3]
9 at [3,4]
Hardest tiles cleared:
9 at [3,4]
9 at [2,3]
9 at [4,2]
7 at [4,3]
6 at [1,2]
1 at [2,2]
TNT at [3,3]
TNT at [3,2]
```

PART B: Priority Queue Implementation

For this project, you are required to implement and use your own priority queue containers. You will implement a **"sorted array priority queue"**, a **"binary heap priority queue"**, and a **"pairing heap priority queue"** which implement the interface defined in **Eecs281PQ.h**.

To implement these priority queues, you will need to fill in separate header files, **SortedPQ.h**, **BinaryPQ.h**, and **PairingPQ.h**, containing all the definitions for the functions declared in **Eecs281PQ.h**. We have provided these files with empty function definitions for you to fill in. You may also add any additional functions needed to maintain the priority queue.

We provide a very bad priority queue implementation called the "**Unordered priority queue**" in **UnorderedPQ.h**, which does a linear search for the most extreme element each time it is requested. You can look at the code in this priority queue to see the use of `this->compare()` (more on that below), how to write your constructors, etc. There's also an **UnorderedFastPQ.h** that is faster; look at how it uses `mutable` to accomplish that.. You can also use this priority queue to ensure that your other priority queues are returning elements in the correct order. All priority queues should return elements in the same (priority) order no matter which implementation is being used.

These files specify more information about each priority queue type, including runtime requirements for each member function and a general description of the container.

You are not allowed to modify **Eecs281PQ.h** in any way. Nor are you allowed to change the interface (names, parameters, return types) that we give you in any of the provided headers. You are allowed to add your own private helper functions and variables to the other header files as needed, so long as you still follow the requirements outlined in both the spec and the comments in the provided files.

These priority queues can take in an optional comparison functor type, `COMP_FUNCTOR`. Inside the classes, you can access an instance of `COMP_FUNCTOR` with `this->compare()`. All of your priority queues must default to be MAX priority queues. This means that if you use the default comparison functor with an integer PQ, `std::less<int>`, the PQ will return the *largest* integer when you call `top()`. Here, the definition of max (aka most extreme) is entirely dependent on the comparison functor. For example, if you use `std::greater<int>`, it will become a min-PQ. The definition is as follows:

If A is an arbitrary element in the priority queue, and `top()` returns the "most extreme" element. `this->compare(top(), A)` should always return false (A is "less extreme" than `top()`).

It might seem counterintuitive that `std::less<>` yields a max-PQ, but this is consistent with the way that the STL `std::priority_queue<>` works (and other STL functions that take custom comparators, like `sort()`).

We will compile your priority queue implementations with our own code to ensure that you have correctly and fully implemented them. To ensure that this is possible (and that you do not lose credit for these tests), do not define a main function in one of the PQ headers, or any header file for that matter.

`Eecs281PQ<>` interface

Functions:

```
push(const TYPE& val)  //inserts a new element into the priority
                       //queue

top()                  //returns the highest priority element in the
                       //priority_queue

pop()                  //removes the highest priority element from
                       //the priority queue

size()                 //returns the size of the priority queue

empty()                //returns true if the priority queue is
                       //empty, false otherwise
```

Unordered Priority Queue

The *unordered priority queue* implements the priority queue interface by maintaining a vector. This has already been implemented for you, and you can use the code to help you understand how to use the comparison functor, etc. Complexities and details are in **UnorderedPQ.h** and **UnorderedFastPQ.h**.

Sorted Priority Queue

The *sorted priority queue* implements the priority queue interface by maintaining a **sorted** vector. Complexities and details are in **SortedPQ.h**. This should be written almost entirely using the STL. The number of lines of code needed to get this working is fairly low, generally <= 10.

Binary Heap Priority Queue

Binary heaps will be covered in lecture. We also highly recommend reviewing Chapter 6 of the CLRS book. Complexities and details are in **BinaryPQ.h**. One issue that you may encounter is that the examples and code in the slides use 1-based indexing, but your code must store the values in a vector (where indexing starts at 0). There are three possible solutions to this problem:

1) Add a "dummy" element at index 0, make sure you never let them access it, and make sure that `size()` and `empty()` work properly.
2) Translate the code from 1-based to 0-based. This is the best solution but the hardest.
3) Use a function to translate indices for you. Instead of accessing `data[i]`, use `getElement(i)`. The code for `getElement()` is given below (both versions are needed).

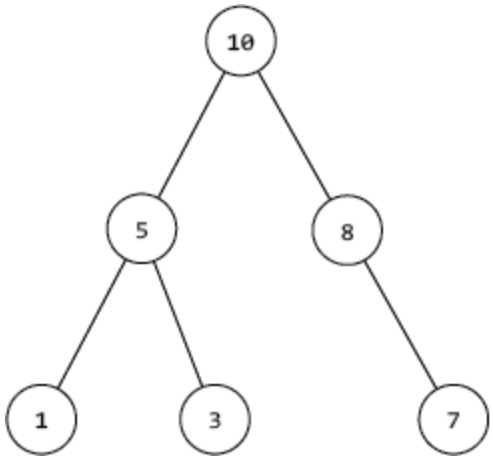
```
// Translate 1-based indexing into a 0-based vector
TYPE &getElement(std::size_t i) {
    return data[i - 1];
} // getElement()

const TYPE &getElement(std::size_t i) const {
    return data[i - 1];
} // getElement()
```

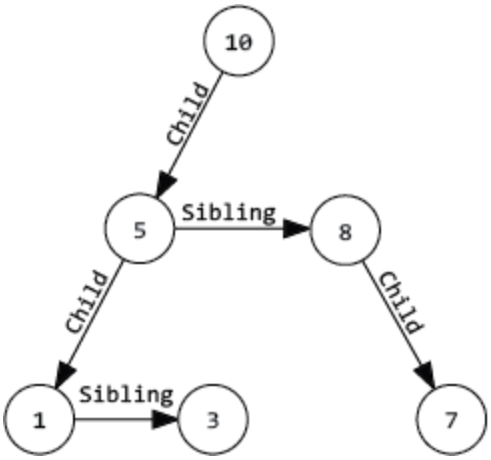
Pairing Priority Queue

Pairing heaps are an advanced heap data structure that can be quite fast. In order to implement the pairing priority queue, read the two papers we provide you describing the data structure. Complexity details can be found in **PairingPQ.h**. We have also included a couple of diagrams that may help you understand the tree structure of the pairing heap.

Below is the pairing heap modeled as a tree, in which each node is greater than each of its children:



To implement this structure, the pairing heap will use child and sibling pointers to have a structure like this:



Implementing the Priority Queues

Look through the included header files: you need to add code in **SortedPQ.h**, **BinaryPQ.h**, and **PairingPQ.h**, and this is the order that we would suggest implementing the different priority queues. Each of these files has TODO comments where you need to make changes. We wanted to provide you with files that would compile when you receive them, so some of the changes involve deleting and/or changing lines that were only placed there to make sure that they compile. For example, if a function was supposed to return an integer, NOT having a return statement that returns an integer would produce a compiler error. Also, functions which accept parameters have had the name of the parameter commented out (otherwise you would get an unused parameter error). Look at **UnorderedPQ.h** as an example, it's already done.

When you implement each priority queue, you cannot compare *data* yourself using the < operator. You can still use < for comparisons such as a vector index to the size of the vector, but you must use the provided comparator for comparing the data stored inside your priority queue. Notice that **Eecs281PQ.h** contains a member variable named `compare` of type `COMP` (one of the templated class types). Although the other classes inherit from **Eecs281PQ.h**, you cannot access the `compare` member directly, you must always say `this->compare()` (this is due to a template inheriting from a template). Notice that in `UnorderedPQ<>` it uses `this->compare()` by passing it to the `max_element()` algorithm to use for comparisons.

When you write the `SortedPQ<>` you cannot use `binary_search()` from the STL, but you wouldn't want to: it only returns a `bool` to tell you if something is already in the container or not! Instead use the `lower_bound()` algorithm (which returns an iterator), and you can also use the `sort()` algorithm -- you don't have to write your own sorting function. You do however have to pass the `this->compare` functor to both `lower_bound()` and `sort()`.

The `BinaryPQ<>` is harder to write, and requires a more detailed and careful use of the comparison functor, and you have to know how one works to write one in the first place, even for `UnorderedPQ<>` to use. See the **About Comparators** section below.

Compiling and Testing Priority Queues

You are provided with a testing file, **testPQ.cpp**. **testPQ.cpp** contains examples of unit tests you can run on your priority queues to ensure that they are correct; however, it is **not** a complete test of your priority queues; for example it does not test `updatePriorities()`. It is especially lacking in testing the `PairingPQ<>` class, since it does not have any calls to `addNode()` or `updateElt()`. You should add more tests to this source code file.

Using the 281 **Makefile**, you can compile **testPQ.cpp** by typing in the terminal: `make testPQ`. You may use your own **Makefile**, but you will have to make sure it does not try to compile your driver program as well as the test program (i.e., use at your own risk).

Logistics

The `std::priority_queue<>`

The STL `std::priority_queue<>` data structure is basically an efficient implementation of the binary heap which you are also coding in **BinaryPQ.h**. To declare a `std::priority_queue<>` you need to state either one or three types:

1) The data type to be stored in the container. If this type has a natural sort order that meets your needs, this is the only type required.
2) The underlying container to use, usually just a `std::vector<>` of the first type.
3) The comparator to use to define what is considered the highest priority element.

If the type that you store in the container has a natural sort order (i.e. it supports `operator<()`), the `std::priority_queue<>` will be a max-heap of the declared type. For example, if you just want to store integers, and have the largest integer be the highest priority:

```
std::priority_queue<int> pqMax;
```

When you declare this, by default the underlying storage type is `vector<int>` and the default comparator is `less<int>`. If you want the smallest integer to be the highest priority:

```
std::priority_queue<int, vector<int>, greater<int>> pqMin;
```

If you want to store something other than integers, define a custom comparator as described below.

About Comparators

The functor must accept two of whatever is stored in your priority queue: if your PQ stores integers, the functor would accept two integers. If your PQ stores pointers to units, your functor would accept two pointers to orders (actually two `const` pointers, since you don't have to modify units to compare them).

Your functor receives two parameters, let's call them `a` and `b`. It must always answer the following question: **is the priority of `a` less than the priority of `b`**? What does lower priority mean? It depends on your application. For example, refer back to the **Breaking Out of the Mine** section: if you have multiple tiles in your priority queue, you determine priority based on smallest rubble value. If rubble value is the same, break ties based on column or row number.

When you would have wanted to write a comparison, such as:

```
if (data[i] < data[j])
```

You would instead write:

```
if (this->compare(data[i], data[j])
```

Your priority queues must work **in general**. In general, a priority queue has no idea what kind of data is inside of it. That's why it uses `this->compare()` instead of `<`. What if you wanted to perform the comparison `if (data[i] > data[j])`? Use the following:

```
if (this->compare(data[j], data[i])
```

Libraries and Restrictions

For part A, we encourage the use of the STL, with the exception of these prohibited features:

- The thread/atomics libraries (e.g., boost, pthreads, etc) which spoil runtime measurements.
- Smart pointers (both unique and shared).

You **are** allowed to use `std::vector<>`, `std::priority_queue<>` and `std::deque<>`.

You are **not** allowed to use other STL containers. Specifically, this means that use of `std::stack<>`, `std::queue<>`, `std::list<>`, `std::set<>`, `std::map<>`, `std::unordered_set<>`, `std::unordered_map<>`, and the `'multi'` variants of the aforementioned containers are forbidden.

For part B,

You **are** allowed to use `std::sort()`.
You **are** allowed to use `std::lower_bound()`.

You are **not** allowed to use `std::partition()`, `std::partition_copy()`, `std::partial_sort()`, `std::stable_partition()`, `std::make_heap()`, `std::push_heap()`, `std::pop_heap()`, `std::sort_heap()`, `std::qsort()`, or anything that trivializes the implementation of the binary heap.

You are **not** allowed to use the C++14 regular expressions library (it is not fully implemented on gcc 6.2) or the thread/atomics libraries (it spoils runtime measurements).

You are **not** allowed to use other libraries (eg: boost, pthread, etc).

Furthermore, you may **not** use any STL component that trivializes the implementation of your priority queues (if you are not sure about a specific function, ask us).

Your main program (part A) **must** use `std::priority_queue<>`, but your PQ implementations (part B) **must not**.

Testing and Debugging

Part of this project is to prepare several test files that will expose defects in the program. **We strongly recommend** that you **first** try to catch a few of our buggy solutions with your own test files, before beginning your solutions. This will be extremely helpful for debugging. The autograder will also tell you if one of your own test files exposes bugs in your solution.

Test File Details

Your test files must be valid Map mode input files. We will run your test files on several buggy project solutions. If your test file causes a correct program and the incorrect program to produce different output, the test file is said to expose that bug.

Test files should be named **test-n-<FLAGS>.txt** where 0 < *n* <= 10. The <FLAGS> portion of the name should contain at least one of `'m'` (median), `'s'` (statistics), and/or `'v'` (verbose). You must specify at least one flag; you can specify two or three if you want. If the 's' flag is specified, the autograder will pick the number of statistics based on the test number (a larger value of *n* will generate more statistics). For example, **test-1-vs.txt** is a valid file name.

Your test files must be in map input mode and cannot have a size larger than 15. You may submit up to 10 test files (though it is possible to get full credit with fewer test files). The tests the autograder runs on your solution are NOT limited to having a `Size` of 15; your solution should not impose any size limits (as long as sufficient system memory is available). However you can assume that an `int` (signed or unsigned) can hold the size, row or column number, amount of rubble, etc.

Submitting to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your "submit directory". You can make two separate projects inside of your IDE: one for part A, another for part B. Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file: `// Project Identifier: 19034C8F3B1196BF8E0C6E1C0F973D2FD550B88F`
- The Makefile must also have this identifier (in the first TODO block).
- DO NOT copy the above identifier from the PDF! It might contain hidden characters. Copy it from the README file instead (this file is included on Canvas).
    - You have deleted all .o files and any executables. Typing '`make clean`' should accomplish this.
    - Your makefile is called Makefile. Typing '`make -R -r`' builds your code without errors and generates an executable file called `MineEscape`. The command line options `-R` and `-r` disable automatic build rules, which will not work on the autograder.
    - Your Makefile specifies that you are compiling with the gcc optimization option `-O3`. This is extremely important for getting all of the performance points, as `-O3` can often speed up execution by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with `-g`, as this will slow your code down considerably. If your code "works" when you don't compile with `-O3` and breaks when you do, it means you have a bug in your code!
    - Your test files are named `test-n-MODE.txt` and no other project file names begin with test. Up to 10 test files may be submitted.
    - The total size of your solution and test files does not exceed 2MB.
    - You don't have any unnecessary files (including temporary files created by your text editor and compiler, etc) or subdirectories in your submit directory (i.e. the .git folder used by git source code management).
    - Your code compiles and runs correctly using version 6.2.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 6.2.0 running on Linux (students using other compilers and OS did observe incompatibilities). **In order to compile with g++ version 6.2.0 on CAEN you must put the following at the top of your Makefile (or use the one that we've provided):**

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Turn in all of the following files:

- All your .h and .cpp files for the project (solution and priority queues)
- Your Makefile
- Your test files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run one of these two commands (assuming you are using our Makefile):

> `make fullsubmit` (builds a "tarball" named `fullsubmit.tar.gz` that contains all source and header files, test files, and the `Makefile`; this file is to be submitted to the autograder for any completely graded submission)
>
> `make partialsubmit` (builds a "tarball" named `partialsubmit.tar.gz` that contains only source and header files, and the `Makefile`; test files are not included, which will speed up the autograder by not checking for bugs; this should be used when testing the simulation only)

For Part A, you can submit test files (map files), for Part B you only submit code.

These commands will prepare a suitable file in your working directory. Submit your project files directly to either of the two autograders at:

https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu/.

You can safely ignore and override any warnings about an invalid security certificate. **When the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to 2 times per calendar day, per part, with autograder feedback (more in Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). We will use your best submission when running final grading. Part of programming is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you 'commit' your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and Canvas regarding the use of version control.

If you use late days on this project, it is applied to both parts. So if you use one late day for Part A, you automatically get a 1-day extension for Part B, but are only charged for the use of one late day.

Please make sure that you read all messages shown at the top section of your autograder results! These messages will help explain some of the issues you are having (such as losing points for having a bad Makefile). Also be sure to check whether the autograder shows that one of your own test files exposes a bug in your solution (at the bottom of the student test files section). Watch for the word "Hint" in the autograder feedback about specific test cases.

Grading

- 60 pts total for part A
    - 45 pts — correctness & performance
    - 5 pts — memory leaks
    - 10 pts — student-provided test files
- 40 pts total for part B
    - 20 pts — pairing heap correctness & performance
    - 5 pts — pairing heap memory leaks
    - 10 pts — binary heap correctness & performance
    - 5 pts — sorted heap correctness & performance

We also reserve the right to deduct up to 5 points for bad programming style, code that is unnecessarily duplicated, etc..

Refer to the Project 1 spec for details about what constitutes good/bad style, and remember:

It is **extremely helpful** to compile your code with the gcc options (default in our **Makefile**): `-Wall -Werror -Wextra -Wvla -pedantic`. This will help you catch bugs in your code early by having the compiler point out when you write code that is either of poor style or might result in unintended behavior.

Appendix A: Printing out the Grid

While this is never required for the project assignment, you may find it helpful to be able to print out the state of the grid in a human readable format. Unfortunately, the values of rubble may have a different number of digits, which can make it hard to read. For example, consider the following grid:

```
10 5 100
200 400 200
1 1 -1
```

It is fairly hard to see which tiles touch which other tiles because they do not line up well. Luckily, the header `<iomanip>` contains the definition for the `std::setw()` stream manipulator.

The following code snippet gives a quick example of how to use it:

```
#include <iomanip>
#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
  vector<int> ints1 = {0, 24, 100, 5};
  vector<int> ints2 = {100, 2, 40, 2};
  for(auto i : ints1)
    cout << setw(4) << i;
  cout << endl;
  for(auto i : ints2)
    cout << setw(4) << i;
  cout << endl;
  return 0;
} // main()
```

Output:

```
   0  24 100   5
 100   2  40   2
```

In the previous project, using `iomanip` caused a loss of points; in this project it is allowed by the autograder (you will need `iomanip` to set the digits of precision for the median output).

Appendix B: Tips (mostly PairingPQ, but others also)

There are helpful videos on the EECS 281 YouTube channel, specifically the running median: https://youtu.be/_T63Cajeyhw (which is used for many different versions of Project 2). Here is a video about the Mine Escape: https://youtu.be/ICgHr6ZbUDk. Here is a video all about Part B: https://youtu.be/kOqzBMvxfuw. There's also a video recorded a few years ago by a student (during office hours) about the Pairing PQ: https://youtu.be/irsHBpw2fhE.

Start coding the mine escape first, but keep thinking about the priority queue implementations. When you code them, the suggested order is SortedPQ, BinaryPQ, PairingPQ.

When you're working on statistics output, be efficient. You can use something from this project to make the easiest/hardest tiles cleared more efficient.

You will need TWO priority queues in Part A: one for things discovered, and another to handle the TNT chain explosion. Remember, once a TNT explosion starts, it must finish!

For Part A, resist the urge to use pointers! If you point to somewhere within a 2D vector, how do you know it's row and column number for printing? Trust us when we say that you'll get enough pointers in the PairingPQ.

You can always count on the `UnorderedPQ.h` being correct. If your `testPQ.cpp` works with this

and doesn't work with one of your other PQ implementations, it is most likely a bug in that PQ implementation.

Make sure ALL of your constructors initialize ALL of your member variables (but objects like a `vector` might be fine with just being default-constructed). This is most likely to be a problem with Pairing, but check the others also.

Even if you think it's working, run your Pairing Heap with valgrind to check for memory errors and/or leaks. This is good advice for the entire project.

Verbose mode doesn't require a lot of extra work and can make it easier to debug your program. For example, if a tile gets blown up in the wrong order, it would be very useful to know that right away, rather than wondering why the summary line has the wrong number of tiles or rubble cleared.

When you are writing a derived class and want to use the `compare` member variable (that is already declared in the base class), you must use `this->compare`. This is because you are working on a templated class that inherits from a templated class (referred to in C++ terms as a dependent context).

Your comparator should always answer the same question! If called with two parameters (let's say a, b), it should return `true` if: priority(a) < priority(b).

Make sure that each of your PQ implementation files contains every `#include` needed by that file! You might be including `deque` from your `main.cpp`, and counting on that file existing for `PairingPQ.h`. You would then be unable to compile when we test your PQ implementations with our `.cpp` file.

The rest of these tips are specific to the Pairing Heap.

After you're done reading about the Pairing Heap data structure, plan on several days of coding, testing, and optimizing JUST for `PairingPQ.h`.

DON'T USE RECURSION, it's very easy to have a large pairing heap that causes a stack overflow.

You can add other private member variables and functions, such as the current size and a `meld()` function.

You are not allowed to change from sibling and child pointers to a deque or vector of children pointers. You don't want to: this is slower than using sibling/child., and the autograder timings are based on the sibling/child approach.

You are allowed to add one more pointer to the `Node` structure. You can use either a parent (pointing up) or a previous (points left except leftmost points to parent). You can also add a public function to return it if you want to. We didn't put this variable or function in the starter file because some students want parent, some want previous. Neither is strictly a better choice than the other; each makes some part of Pairing harder, some part easier; both can pass the timings.

Pointers passed to `meld()` must each point to the "root" node of a Pairing Heap. Root nodes never

have siblings! This is important to making `meld()` as simple and as fast as it should be.

Don't write a copy constructor and copy the code for `operator=()`! Use the copy-swap method outlined in the “Arrays and Containers” lecture.

You are allowed to write an extra function, such as `print()` to display the entire pairing heap, and only use it for testing.

When you need to traverse an entire pairing heap (print, copy constructor, destructor, etc.), think about the Project 1 approach! Use a deque, add the “starting location”, while it’s not empty take the “next” one out, add things nearby, do something with the “next” one that you took out, etc.

Appendix C: Autograder Information

The test cases for **Part A** on the autograder (that show in the table) are for part A, and have the following naming convention:

- “INV”:
  - The test case is an invalid input file. Your solution should `exit(1)` because of an error.
- First letter (for anything not invalid):
  - ‘M’ (medium), ‘L’ (large), and ‘XL’ (extra-large) denote the size of the test case.
  - ‘S’ indicates that the test is the one given in this spec (see Full I/O Example).
  - ‘E’ indicates that the test case is some sort of hand-written edge case.
  - ‘R’ indicates that the grid contains only rubble (possibly 0s, but never any TNT)..
  - ‘N’ indicates that the grid contains no rubble or TNT (i.e. it’s all 0s).
  - ‘T’ indicates that the grid is made entirely of TNT squares.
- Second letter:
  - ‘M’ denotes map input format and ‘R’ denotes pseudo-random input format.
- Lower case letters after a hyphen:
  - ‘m’: The test is run with the median flag on.
  - ‘v’: The test is run with the verbose flag on.
  - ‘s’: The test is run with the statistics flag on.

When you start submitting test files to the autograder, it will tell you (in the section called "Scoring student test files") how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!

For **Part B**, when your priority queues (`SortedPQ`, `BinaryPQ`, and `PairingPQ`) are compiled with our `main()`, we will perform unit testing on your data structures. These test cases all have three-letter names:

1) First letter: **B**inary PQ, **S**orted PQ, **P**airing PQ
2) Second letter: **P**ush, **R**ange, **U**pdate priorities, **A**ddnode, update**E**lt, **C**opy constructor, **O**perator **=**

3) Third letter: **S**mall, **M**edium, **L**arge, e**X**tra-large

A “push” test uses the `.push()` member function to insert numerous values into your priority queue. After that, the values will be checked via `.top()` and `.pop()` until the container is empty, to make sure that every value came out in the correct order. If the “push” test goes over on time, it *might* be the fault of your `.pop()`, not your `.push()`, because both must be called to verify that your container works properly.

A “range” test uses the range-based constructor, from `[start, end)` to insert values into your container, then the test proceeds as described above for the “push” test. The start iterator is inclusive while the end is exclusive, as is normal for the STL.

The “update priorities” tests use `.push()` to fill your container, then half of the values that were given to you are modified (hint, this is accomplished with pointers). After `.updatePriorities()` is called, all of the values are popped out and tested as above.

The first three tests above are run for every priority queue type. The ones below are run only on the `PairingPQ`.

The “addNode” tests use `.addNode()` to fill the container instead of `.push()`, and every value is checked through the returned pointer to make sure that it matches.

The “updateElt” tests use `.addNode()` to fill the container, then half of the values have their priority increased by a random amount using `.updateElt()`. After that, values are popped off one at a time, checking to make sure that each value is correct.

The “copy constructor” and “operator=” tests first fill one `PairingPQ` using `.push()`. Then they use the stated method to create a second `PairingPQ` from the first. Lastly every value is popped from **both** priority queues, making sure that every value is correct (and thus ensuring that a deep copy was performed, not a shallow copy).

For the Pairing Heap, the `.addNode()` member function makes a promise: the item that was added will ALWAYS live at the pointer returned, until the user removes it via `.pop()` (or the destructor). When implementing `.updateElt()` and `.updatePriorities()`, you cannot delete existing nodes and create new ones, as this would break the promise made by `.addNode()`! You must move nodes around.