EECS 281 – Spring 2018
Programming Project 2
Mine All Mine
(Priority Queues)

Due Monday, May 21, 11:59pm

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

Part A: Gold Mining

For Part A, you should always use `std::priority_queue`, not your templates from Part B.

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.

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.

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.

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

- 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 Visibility
The mine is very dark and you cannot see very far. When standing on a clear tile there are only four tiles that are visible 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 see your current tile. This tile must be cleared before you can see the tiles around it.

Definition of Movement
You must always add any visible tiles to a priority queue (more on this below). Highest priority tiles are those that are the easiest to clear (this is also explained below). As in Project 1, make sure you don’t add tiles twice; you cannot move to a tile that you have previously visited. The priority queue will always tell you what your “next” tile will be. Movement is taking the “next” tile from the priority queue and making it your “current” location. When you move to a tile, you must clear it if it contains rubble or TNT. After moving, you can then see any of the four tiles visible from your new location, add them to the priority queue, etc. You use the priority queue to remember tiles that you have seen, but have not yet moved to.

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. Clear the tile with the lower column coordinate first.
2. If the tiles have the same column coordinate, clear 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 his or her 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 visited and the tiles that the miner can visit are **blue**. The tile that the miner is currently located at is **blue and underlined**. 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 visible to the miner is the starting location, `[1,2]`. The miner clears this tile and then can see 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 move to and clear tile `[2,2]` because there are no TNT tiles in view and it is has the lowest rubble-value. Clearing that tile allows us to reach 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 miner then moves to 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
```

Here, our miner sees two TNT tiles! However, due to the tie-breaking rules, the miner will choose to blow up the TNT at `[4,2]` instead of `[3,3]`. 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
```

Since the current location is `[4,2]`, and this tile is cleared, 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). 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.

<u>Map input mode (M)</u>

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:

```
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.**

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

Additionally, when a chain of TNT explosions starts, it should display

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

Once the chain of TNT explosions starts, any 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).

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.

Full I/O Example

Input (mine.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:

After running this on the 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 started at [3,2]!
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"** that implements the interface defined in **Eecs281PQ.h**, which we provide**.**

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.

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 use this priority queue for testing until you implement a more effective priority queue. You can also use this priority queue to ensure that your other priority queues are returning elements in the correct order. Any implementation of a priority queue should give the priority order no matter which implementation you're using.

These files specify more information about each priority queue type, including runtime requirements for each method 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 `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).

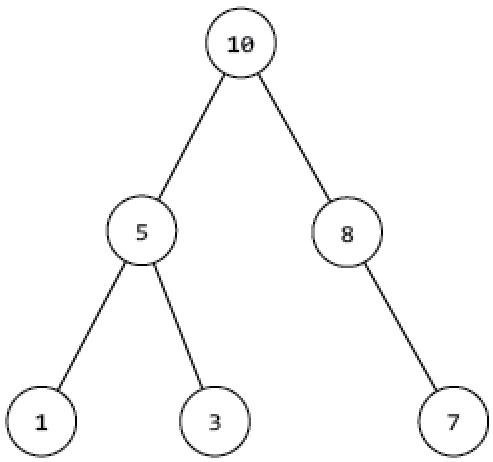
```
// Translate 1-based indexing into a 0-based vector
TYPE &getElement(std::size_t i) {
    return data[i - 1];
} // getElement()

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

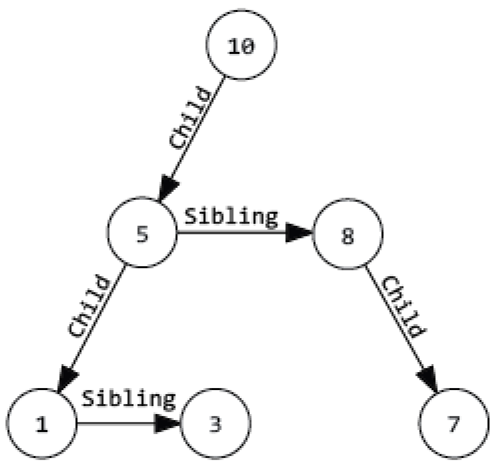
Pairing Priority Queue

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` contains a member variable named `compare` of type `COMP` (one of the templated class types). Although the other classes inherit from `Eecs281PQ`, 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 `priority_queue<>` data structure is basically an efficient implementation of the binary heap which you are also coding in `BinaryPQ.h`. To declare a `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 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 `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:

```
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:

```
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).

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". Before you turn in your code, be sure that:

1. You have deleted all .o files and your executable(s). Typing '`make clean`' should accomplish this.
2. 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).
3. 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 code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with the debug flag, -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!
4. Your test files are named `test-n-<FLAGS>.txt`. Up to 10 test files may be submitted.
5. The total size of your program and test files does not exceed 2MB.
6. 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).
7. 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 may observe incompatibilities).
8. In order to compile with g++ version 6.2.0 on CAEN and the autograder you must put the following at the top of your Makefile (already part of project 0 makefile):
   ```
   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
   ```
9. Turn in all of the following files:
   a. All your *.h and *.cpp files for the project.
   b. Your Makefile.
   c. 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 this command:

```
make fullsubmit
```

This will prepare a suitable file in your working directory.

Alternatively, you can run `make partialsubmit`, which will prepare a submission without test files and prevent you from losing an autograder submission if your project does not build.

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/. **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 three times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade**. If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to check if the autograder shows that one of your own test files exposes a bug in your solution; for the first such test file encountered the autograder will provide the correct output and your output.

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

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: Polymorphism

All of the priority queues (unordered, sorted, heap, and pairing) all inherit from the abstract class Eecs281PQ. This means that an Eecs281PQ pointer variable can correctly refer to any of its derived types, and can call the correct underlying member functions. Example:

```
Eecs281PQ<int>   *pq;
UnorderedPQ<int> unordered;
SortedPQ<int>    sorted;
BinaryPQ<int>    heap;
PairingPQ<int>   pairing;

// if using unordered
pq = &unordered;
// if using sorted
pq = &sorted;
// if using heap
pq = &heap;
// if using pairing
pq = &pairing;

// do stuff with the priority queue
pq->push(5);
pq->push(10);
cout << pq->top() << endl; // prints out 10
```

Appendix C: Using a priority queue of pointers

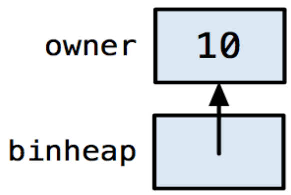
It is highly recommended that you use STL's vector<> for dynamic memory management in this project. In order to avoid duplicating information, other containers that want to access the data can hold pointers to members inside the "owner" container. For example:

```
vector<int> owner;           // the objects are ints
BinaryPQ<int*> binheap;      // holds pointers to ints

int x = 10;
owner.push_back(x);

binheap.push(&owner.back());
// *NOT* binheap.push(&x);
```

This creates the following relationship:



The above example is slightly incomplete in that what will be prioritized is the pointers themselves. To output the values in numerical order, a functor is needed (see next example below).

Warning about using vector<> as an owner container
Recall from lecture that vector runs out of space, it “moves” all of its data members into a new, larger space. This will invalidate any existing pointers at the time! If you do decide to use a vector, make sure that the vector never resizes while pointers to its members are in use. You can do so by using `vector::reserve()` or `vector::resize()`

Because STL containers automatically deallocate the memory they use, you should **never** call **delete** on a pointer to an object that exists in a vector<>. If you do, you will get a “double free” runtime error. This also means that referencing pointers to elements that have been removed from an owner container will cause a segmentation fault.

Example:

```
#include <iostream>
#include <vector>
#include "UnorderedManPQ.h"

using namespace std;

// Comparison functor for integer pointers
struct IntPtrComp {
  bool operator() (const int *a, const int  *b) const {
        return *a < *b;
  }
};

int main()  {
  UnorderedManPQ<int *, intptr_comp> pmheap;
  vector<int> owner = {10, 5, 20, 7};

  for(auto &i : owner) // reference needed for next line
    pmheap.push(&i);   // so that this is the address of the value in owner

  // Process each number in priority order
```

```
  while(!pmheap.empty()){

    // Pop one int pointer off the Unordered PQ;
    // the integer itself is still “alive” in the vector owner

    cout << *pmheap.top() << ' ';
    pmheap.pop();
  } // while

  cout << endl;

  // pmheap is empty, but owner still has all 4 integers.
  // program should print 20 10 7 5

  return 0;
} // main()
```

Appendix D: Autograder test case information

The test cases on the autograder have the following naming pattern:

- “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.
    - ‘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.

For your own test files, there are 12 bugs to find. You will start earning points with the 6th bug found, and will earn full points for finding 10.