Project 2a - Mine Escape

Due Friday, Feburary 23, 2024 at 11:59 PM

Overview

This project is broken into two parts, one is a pathfinding simulation using priority-based search, the other is multiple implementations of the priority queue ADT.

Priority-based Search

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.

Implementing Priority Queues

The Priority Queue ADT is a container which can store items that arrive at any time in any order, and retrieve the item that has the highest priority. Different implementations have different time complexities for these two vital operations (eg. constant time insert, linear time retrieval, logarithmic time insertion/retrieval). Additional functionalities such as number of items stored, insertion at initialization (range-based constructor), and copying, are helpful and will also be implemented as part of the assignment.

Project Goals

- Be able to develop efficient data structures and algorithms
- Use the `std::priority_queue<>` container in multiple contexts
- Understand and implement several kinds of priority queues
- Be able to independently read, learn about, and implement an unfamiliar data structure
- 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

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. 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.

This example mine is for illustrative purposes and is not the same as the input file described in the Full 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 the discovery of 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 grid, the miner has 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, TNT tiles detonating other TNT tiles) are cleared in order from “easiest” to “hardest” (using the Definition of Easiest Tile) and Tie-breaking). These tiles should also be discovered. As stated in Definition of Discovered, 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 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.

Legal Command Line Examples

```
$ ./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

Input files will have front matter that specifies settings not given at the command line. There will be two different types of input, selected by the first character of a valid input file: map mode (`'M'`) and pseudorandom mode (`'R'`). Map input mode is human readable and helpful while debugging, while pseudorandom input mode allows easier testing of large grids.

Map Input Mode

Input will be formatted as follows:

- `'M'` : A single character indicating that this file is in map format
- `Size: <N>` : Positive integer that specifies the size of the square grid (20 means a grid with 20 rows and 20 columns)
- `Start: <StartRow> <StartCol>`/`Start: <StartY> <StartX>` : Two non-negative integers separated by whitespace (any number of spaces or tabs) that specify the coordinates of the starting location, where location [0, 0] is the top-left corner, and location [`<N> - 1` , `<N> - 1`] is the bottom-right corner

After the front matter, map input mode files describe the two-dimensional layout of all the tiles in the mine. Each tile will be separated from other tiles on the same line with whitespace. Rubble tiles are signified by an integer between 0 and 999 (0 means the tile is already clear of rubble), and TNT tiles are represented by the integer -1.

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

p2-mine-escape/spec-M-in.txt

```
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 Input Mode

Input will be formatted as follows:

- `'R'` : A single character indicating that this file is in pseudorandom format
- `Size: <N>` : Positive integer that specifies the size of the square grid (20 means a grid with 20 rows and 20 columns)
- `Start: <StartRow> <StartCol>` / `Start: <StartY> <StartX>` : Two non-negative integers separated by whitespace (any number of spaces or tabs) that specify the coordinates of the starting location, where location [0, 0] is the top-left corner, and location [`<N> - 1` , `<N> - 1`] is the bottom-right corner

- `Seed: <S>` : Non-negative integer used to seed the random number generator
- `Max_Rubble: <R>` : Non-negative integer that specifies the maximum rubble value that can be randomly generated
- `TNT: <T>` : Non-negative integer that specifies the chance that any individually generated tile will be TNT (percentage is 0% when `<T>` is 0, or 1/ `<T>` when `<T>` is non-zero)

Example of R input:

p2-mine-escape/spec-M-in.txt

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

Generating the Grid in 'R' Mode

Included with the project spec are the files P2random.h and P2random.cpp that contain the declaration and definition of the following function:

```
void P2random::PR_init(std::stringstream& ss, uint32_t size,
                       uint32_t seed, uint32_t max_rubble,
                       uint32_t tnt);
```

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 map input mode. You may find the following (incomplete) C++ code helpful in reducing code duplication.

```
    stringstream ss;
    if (mode == "R") {
        // TODO: Read some variables from cin
        P2random::PR_init(ss, size, seed, max_rubble, tnt);
    }  // if ..'R'

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

    // Afterwards, instead of reading from cin or ss, the program reads from
    // inputStream and is unaware of the actual source of the input map
    // Examples:
```

```
    //    inputStream >> <variable>;
    //    getline(inputStream, <variable>);
    // TODO: Add looping here to read the map itself, reading from inputStream
```

The example R and M input files given above are equivalent, they both 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 coordinates specified describe a valid location given the size `[0, N)` of the map

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, the AG will show your error output to help with debugging.

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 includes tiles cleared by TNT, but does not include the TNT tile itself.
- `<AMOUNT>` : the total rubble cleared. This number includes rubble cleared by TNT, but detonating the TNT tile itself counts as 0 rubble cleared.

Verbose Output

During the program execution, if the `-v/--verbose` option is given at the command line, each time a tile whose rubble amount is greater than 0 is cleared, print the following:

```
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 print the following:

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

Any rubble tiles cleared by TNT should add `by TNT` to the cleared line:

```
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/--median` option is given at the command line, 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. Use the following at the beginning of your program to guarantee proper median formatting:

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

Stats Output

After printing the summary line, if the `-s <N>/--stats <N>` option is given at the command line, print the following output:

```
First tiles cleared:
<TILE_TYPE> at [<ROW>,<COL>]
... (lists up to <N> tiles, starting with the first, second, third, etc.)
Last tiles cleared:
<TILE_TYPE> at [<ROW>,<COL>]
... (lists up to <N> tiles, starting with the last, second to last, etc.)
Easiest tiles cleared:
<TILE_TYPE> at [<ROW>,<COL>]
... (lists up to <N> tiles, ascending order, starting with least rubble and
increasing)
```

```
Hardest tiles cleared:
<TILE_TYPE> at [<ROW>,<COL>]
... (lists up to <N> tiles, descending order, starting with most rubble and
decreasing)
```

- `<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` .
- `<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.

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

Full Example

Full Example Command Line

```
$ ./mineEscape -v -m -s 10 < spec-M-in.txt
```

Full Example Input (spec-M-in.txt or spec-R-in.txt)

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

p2-mine-escape/spec-M-in.txt

```
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
```

p2-mine-escape/spec-R-in.txt

```
R
Size: 5
Start: 1 2
Seed: 0
```

```
Max_Rubble: 10
TNT: 5
```

Full Example Output

The output is shown below, with the verbose mode output in **bold**. 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]
```

Logistics

The `std::priority_queue<>`

The STL `std::priority_queue<>` data structure is an efficient implementation of the binary heap which you are 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 type from #1.

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 `std::vector<int>` and the default comparator is `std::less<int>` . If you want the smallest integer to be the highest priority, use three types:

```
    std::priority_queue<int, std::vector<int>, std::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 a custom object, your functor would accept two pointers, etc. Both arguments should be declared as `const`, because nothing should be modified during comparison. Also, if object types are used, reference variables will prevent unnecessary copies from being made.

Your functor receives two parameters (eg. `a` and `b`), and 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 definitions in the Part A: Gold Mining section: if you have multiple tiles in your priority queue, you determine priority based on smallest rubble value. If rubble values are the same, break ties based on column number. If rubble values and column numbers are the same, break ties based on row number.

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.

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

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 input mode 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).
  `# Project Identifier: 19034C8F3B1196BF8E0C6E1C0F973D2FD550B88F`
- 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 11.3.0 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 11.3.0 running on Linux (students using other compilers and OS did observe incompatibilities). In order to compile with g++ version 11.3.0 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-11.3.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-11.3.0/lib64
LD_RUN_PATH := /usr/um/gcc-11.3.0/lib64
```

Turn in all of the following files:

- All your .h/.hpp 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 an autograder at:
https://eecs281staff.github.io/ag-status/.
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

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 -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.

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).

Part B: Priority Queues

The Part B Spec is in a separate document. The solution to Part B will be submitted separately from Part A, and should be developed separately in your development environment with the files from both solutions stored in separate directories.