EECS 281 – Fall 2019
Programming Project 2
The Walking Deadline

Due Thursday, October 17, 2019 at 11:59 PM

Table of Contents

Project Identifier

EECS 281 Project 2, Theme D: *The Walking Deadline*
Version 09-29-19
Credits: David Paoletti, Hector Garcia, Marcus Darden, James Juett, Andrew DeOrio, Waleed Khan, Anna Hua, Luum Habtemariam, and many others

You <u>MUST</u> 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: 9504853406CBAC39EE89AA3AD238AA12CA198043
```

Overview

Zombies are invading the Bob & Betty Beyster Building!

As the Hero of EECS 281, you have been entrusted with protecting the building, its contents, its occupants, and everything that CS stands for. You have your trusty bow, a seemingly infinite supply of arrows, and a collection of priority queues limited only by main memory itself. You need to hold off the zombie horde - and hope that they don't kill you while you're refilling your quiver.

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: Gameplay

For Part A, you should always use `std::priority_queue<>`, not your templates from Part B. In Part A you need to use pointers for keeping track of zombies (both "living" and dead); in Part B you will have to use pointers for the Pairing Heap.
The player operates in 'rounds', beginning with round 1. Each round:

- The player refills their quiver so that it contains `quiver_capacity` arrows.
- Existing zombies move toward the player and attack if they have reached the player.
- New zombies appear.
- The player shoots zombies with arrows.

The Zombies

A zombie is defined by five attributes:

- A **name**. Names will always be unique in our test cases (and you do not need to check for this).
- An unsigned integer **distance** between the zombie and the player
- An unsigned integer **speed** (>0) at which the zombie moves toward you
- An unsigned integer **health** (>0), the amount of damage the zombie can take before being destroyed
- The number of **rounds** the zombie has been active. This is measured as number of rounds, from and including the round it was created, to and including the round it was shot or when the game ends. If the zombie was created in round 2 and was shot or the player was eaten in round 5, it was active for 4 rounds (rounds 2, 3, 4, and 5).

Input

Command line Flags

Your executable must be named `zombbb`. You may assume the command line is well-formed: you do not need to error-check it. It takes the following command line flags:

- `-v`, `--verbose`: The `-v` flag is optional. If provided, indicates that you should print out extra messages during the operation of the program about zombies being created, moved, or destroyed.
- `-s <num>`, `--statistics <num>`: The `-s` flag is optional. If it is provided, then `<num>` is a required unsigned integer. If provided, you should print out statistics at the end of the program. You'll print out `<num>` entries for each type of statistic.
- `-m`, `--median`: The `-m` flag is optional. If provided, indicates that you should print out extra messages during the program indicating the median time that zombies have been active before being destroyed.
- `-h`, `--help`: When passed `-h`, print a useful help message explaining how to use the program to `cout`. Ignore all other input (including other command line flags and `cin`) and exit with status `0`.

Examples of legal invocations:

- `./zombbb -mv < infile.txt`
- `./zombbb --statistics 10 -v`

Examples of illegal invocations:

- `./zombbb -M`: The `-m` option must be lower case.
- `./zombbb -s`: When provided, the `-s` option has a required argument <num>.

Zombies are not particularly attentive to detail, so we will not be checking your command line handling. But it is to your benefit to add a reasonable amount of error-checking, just in case you mistype it.

Input Format

Information about the zombies will be given on standard input (`cin`). The input consists of a *header*, followed by any number of *rounds*. You may assume the input is well-formed: you do not need to error-check it. A full input/output example is provided in the appendices.

Header Format

Each header starts with exactly one comment line, beginning with a `#`. You should ignore this line. The file then contains the following parameters, all of which are followed by unsigned integer values. The `quiver-capacity` and `max-rand` values provided will not be zero.

- `quiver-capacity`: The number of arrows you can fit into your quiver. You refill your quiver to this number at the beginning of every round.
- `random-seed`: A "seed", which initializes our patent-pending Random Zombie Generator™ (see random zombie generation). The Random Zombie Generator™ always produces the same zombies when provided with the same seed.
- `max-rand-distance`: The maximum starting distance of a randomly-generated zombie.
- `max-rand-speed`: The maximum speed of a randomly-generated zombie.
- `max-rand-health`: The maximum health of a randomly-generated zombie.

Named zombies are not constrained to the `max-rand` parameters: they may be farther, faster, or healthier.

Example header:

```
# my test case
quiver-capacity: 10
random-seed: 123456
max-rand-distance: 50
max-rand-speed: 10
max-rand-health: 5
```

Round Format

After the header is a list of rounds. There is always at least one round. Each round starts with a delimiting triple hyphen ("`---`"). The round information contains a **round number**, information about **random zombies**, and possibly a list of **named zombies**.

The round numbers in the round list strictly increase. However some round numbers may be skipped! You should simulate all of the currently active zombies as normal during unlisted rounds, but you should not create any new ones. The only information provided about the random zombies is the number of them; they will be generated by the Random Zombie Generator™ (see the Random Zombie Generation section). Information about the named zombies is provided on the following lines. If the number of named zombies is given as 0, there will be no lines describing named zombies.

Example round:

```
---
round: 3
random-zombies: 10
named-zombies: 3
darden    distance: 20  speed: 10  health: 3
paoletti  distance: 30  speed: 10  health: 5
garcia    distance: 40  speed: 15  health: 4
```

Note on Parameter Names

For your convenience in writing test files, the name of any parameter may be abbreviated: it may be replaced with any non-empty string without whitespace. Additionally, whitespace is to be ignored, except that there is at least one whitespace character between a parameter name and value, and the comment line always ends with a newline character. For example, the input below is also valid:

```
# my test case
quiver    10  seed   123456
rand-dist 50  rand-speed 10  rand-health 5
---
r 3
rndzmbs 10
named 3

darden    distance: 20  speed: 10  health: 3
paoletti  distance 30 speed 10 health 5
garcia    d 40 s 15 h 4
```

Hint: read with >>, not getline! The text box below contains a complete and valid sample input file.

Example Input File

```
# Project 2 specification example
quiver-capacity: 10
random-seed: 2049231
max-rand-distance: 50
max-rand-speed: 60
max-rand-health: 1
---
round: 1
random-zombies: 15
named-zombies: 3
MarkSchlissel distance: 150 speed: 300 health: 15
MarySueColeman distance: 2 speed: 3 health: 6
LeeBollinger distance: 100 speed: 1 health: 100
---
round: 3
random-zombies: 10
named-zombies: 1
JamesDuderstadt distance: 20 speed: 10 health: 20
```

Random Zombie Generation

To standardize random number generation across all development platforms in the course we have created the `P2random` files (`P2random.h` and `P2random.cpp`). They generate random numbers using the Mersenne Twister (http://en.wikipedia.org/wiki/Mersenne_Twister). Before generating any random numbers, call `P2random::initialize()` passing it the the `random-seed`, `max-rand-distance`, `max-rand-speed`, and `max-rand-health`, values (read from the start of the input file), **in that order**. Call

this function **only once**. Include the header file in any module where random numbers are needed. There is no need to modify either of the `P2random` files.

Each round, you should generate random zombies before the player starts shooting zombies but after existing zombies have been updated. When randomly generating zombies, you should generate them as follows:

```
std::string  name     = P2random::getNextZombieName();
unsigned int distance = P2random::getNextZombieDistance();
unsigned int speed    = P2random::getNextZombieSpeed();
unsigned int health   = P2random::getNextZombieHealth();
```

You should always generate the zombie's statistics in the order given above, because the ordering will affect the result of the pseudo-random number generation. **DO NOT** put these 4 function calls into another function call or constructor call without the intermediate variables! Some compilers evaluate function parameters right-to-left, which would result in incorrect values.

Also: Explicitly specified zombies are always created **after** the random ones.

The Battle

Zombie Offense

At the beginning of each round (just after you reload your quiver), the zombies move in closer. Every zombie has a base speed, and they move a distance equal to their base speed. Each zombie's new distance is calculated as follows:

```
new_distance = max(0, distance - speed);
```

Be careful: if you use an unsigned integer to represent the distance/speed/movement, because the subtraction `distance - speed` could wrap around if the result would have been negative. You **must** move every zombie that has not been destroyed yet, even if the player is dead, because verbose mode prints a move message.

You should perform this update on the zombies in the order in which they were initially created. You should not update the number of rounds active for zombies that are inactive (dead). **This will require that you keep a data structure in addition to your priority queue** to have all of the zombies available in the order of their creation. Your priority queue should then refer to elements inside this data structure (think about how).

As each zombie moves, if it has reached the player (distance is 0) it attacks, eating the player's brain. If the player has their brain eaten, they die and the game is lost. After the existing zombies have moved and attacked, if the player is still alive new zombies may appear, but they do not move or attack until the next round.

Your Defense

You've strategically taken position behind a chokepoint in order to maximize your chances of survival. However, this limits your movement options: you may not move during the course of the game.

Each round, you fill your quiver of arrows to its maximum capacity (`quiver_capacity`) and shoot as many times as you have arrows in your quiver. You have an infinite supply of arrows overall, but only get the opportunity to refill your quiver in between rounds of zombies. Each arrow does one point of damage; a zombie is destroyed if its health reaches 0.

You must prioritize how to shoot the zombies in order to survive for as long as possible using a priority queue. In particular, you should approximate each zombie's **unsigned integer** estimated time of arrival (ETA) using the following formula:

```
ETA = distance / speed;
```

Prioritize shooting the zombie with the lowest ETA. You may shoot the same zombie with several arrows during a round, but do not continue to shoot a zombie that has been destroyed (i.e. after its health has reached zero).

In the event of ties in ETA, you should shoot the zombie with the lower health. If zombies are also tied in health, you should shoot the zombie with the lexicographically smaller name. For example, if zombies `paoletti` and `darden` are equally close to you with equal health, you should shoot `darden` first since he has the lexicographically smaller name. You should **not** ignore case for this comparison: given zombies `PAOLETTI` and `darden`, you should shoot `PAOLETTI` first, as he has the lexicographically smaller name (by ASCII value). **Use `std::string::operator<()` for this**: it handles the comparison correctly.

When you shoot a zombie, its health is decreased by one. When it reaches zero, it has been destroyed and is no longer active.

If you manage to destroy all of the zombies, then you have won the battle.

Round Breakdown

The following summarizes the actions that need to occur in a round and the order in which they should happen:

1. If the `--verbose` flag is enabled, print `Round:` followed by the round number.
2. You refill your quiver. Set the number of arrows you have equal to your quiver capacity.
3. All active zombies advance toward you, updated in the order that they were created.
    a. Update the zombie and move it closer to you.
    b. If the `--verbose` flag is enabled, print the zombie name, speed, distance, and health, along with “Moved:” For example:
    `Moved: paoletti0 (distance: 0, speed: 20, health: 1)`
    c. If at this point the zombie has reached you (has `distance == 0`), then it attacks you and you die.
    d. If you die, the first zombie that reached you is the one that has “eaten” you (see end-of-battle output). You still need to update the other zombies, so don’t exit the loop yet!
4. At this point, if the Player was killed in Step 3, the battle ends.
    a. Print any required messages and statistics.
5. New zombies appear:
    a. Random zombies are created.

b. Named zombies are created.
c. If the `--verbose` flag is enabled, print new active zombie name, speed, distance, and health information in the order they were created, along with "Created." This is true for both random and named zombies. For example:
`Created: paoletti0 (distance: 25, speed: 20, health: 1)`

6. Player shoots zombies.
   a. Shoot the zombie with the lowest ETA (tie breaking described above) until your quiver is empty.
   b. If you destroy a zombie AND the `--verbose` flag is enabled, display a message; for example:
   `Destroyed: paoletti0 (distance: 4, speed: 1, health: 0)`

7. If the `--median` flag is enabled, AND any zombies have been destroyed so far, display a message about the median time that zombies have been active; for example:
`At the end of round 1, the median zombie lifetime is 1`

8. If there are no more zombies and none will be generated in a future round, then you have won the battle.
   a. Print any required messages and statistics. The battle ends.

Output

If the player lives after all zombies have been destroyed and no new zombies will be generated in a later round, the output (sent to `cout`) should be:

```
VICTORY IN ROUND <ROUND_NUMBER>! <NAME_OF_LAST_ZOMBIE_KILLED> was the last zombie.
```

If the player is killed, the output should be:

```
DEFEAT IN ROUND <ROUND_NUMBER>! <NAME_OF_KILLER_ZOMBIE> ate your brains!
```

Notes:

- If multiple zombies would be able to eat your brains in the same round, you should print the name of the one who was updated first.
- This output should be printed on one line (i.e. no newline characters in the middle).

Example:

```
DEFEAT IN ROUND 2! FoxMcCloud ate your brains!
```

If and only if the '`--statistics N`' option is specified on the command line, the following additional output should be printed after the 'DEFEAT' or 'VICTORY' line in the following order without any blank lines separating them:

- The number of zombies still active at the end, in the format:
  `Zombies still active: <NUMBER_OF_ZOMBIES>`
  Where `NUMBER_OF_ZOMBIES` is the number of zombies still active (created but not destroyed.
- The names of the first `N` zombies that were killed, each followed by a space and the number (`1,2,3,...,N`) corresponding to the relative order they were killed in. These zombies should be printed in order, such that the very first one killed is printed first, and the `N`th first one is printed `N`th.

- The names of the last `N` zombies that were killed, followed by a space, and then the number (`N,N-1,...,1`) corresponding to the relative order they were killed in. These zombies should be printed in order, such that the very last one killed is printed first, and the `N`th-to-last zombie is printed `N`th.
- The names of the `N` zombies who were active for the most number of rounds followed by a space and then the number of rounds. These zombies should be printed in order such that the zombie who was in the most rounds appears first. In the event that there is more than one zombie who survived for the same amount of time, you should print the one who has a lexicographically smaller name first (use the < operator of `std::string` for this).
- The names of the `N` zombies who were active for the least number of rounds, but at least one round, followed by a space and then the number of rounds. These zombies should be printed in order such that the zombie who was in the least rounds appears first. Break ties using the same lexicographic rule as before.

Hint: Be efficient about how the last two statistics (that deal with how many rounds a zombie has been active) are calculated.

If for any of the above statistics you do not have at least `N` zombies to print, you should print the data for as many as you can.

Example: If the program is run with '`--statistics 10`', there were only three zombies in the game, and you killed FoxMcCloud before FalcoLombardi and they were both active for 3 rounds and then you killed DarkLink in round 5, its fourth round of being active, you should print the following statistics output (after the victory/defeat line):

```
Zombies still active: 0
First zombies killed:
FoxMcCloud 1
FalcoLombardi 2
DarkLink 3
Last zombies killed:
DarkLink 3
FalcoLombardi 2
FoxMcCloud 1
Most active zombies:
DarkLink 4
FalcoLombardi 3
FoxMcCloud 3
Least active zombies:
FalcoLombardi 3
FoxMcCloud 3
DarkLink 4
```

Note that '`FalcoLombardi`' < '`FoxMcCloud`' lexicographically and that some zombies may be included in several lists. This is not the output of the sample input given above; for that output, see the appendices.

If `--verbose` is specified, print the following at every round:

- "`Round: <round#>`".
- For each zombie updated, print "`Moved:` " and its name distance, speed and health.

- After a zombie is created, print "`Created: `" and its name, distance, speed and health.
- Whenever a zombie is killed, print "`Destroyed: `" and its name, distance, speed and health.

For example, your output generated via the `--verbose` flag may look like:

```
Round: 1
Created: paoletti0 (distance: 25, speed: 20, health: 1)
Created: juett1 (distance: 17, speed: 48, health: 1)
Created: FoxMcCloud (distance: 10, speed: 10, health: 100)
Created: FalcoLombardi (distance: 12, speed: 12, health: 100)
Round: 2
Moved: FoxMcCloud (distance: 0, speed: 10, health: 100)
Moved: FalcoLombardi (distance: 0, speed: 12, health: 92)
DEFEAT IN ROUND 2! FoxMcCloud ate your brains!
```

Again, this is not the correct output for the example given above; see the appendices for that.

Part B: Implementation of Priority Queues

For this part of the 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 good implementation of a very bad priority queue approach called the "**Unordered priority queue**" in **`UnorderedPQ.h`**, which does a linear search for the most extreme element each time it is needed. You can test your other priority queue implementations against `UnorderedPQ`. You can also use this priority queue to ensure that your other priority queues are returning elements in the correct order.

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`. Inside the classes, you can access an instance of `COMP` 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. `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

Member variables:

```
compare                 // More on this below;
                        // see "Implementing the Priority Queues"
```

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).

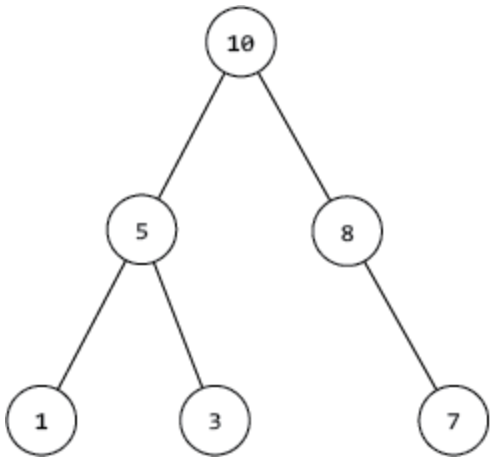
```
// Translate 1-based indexing into a 0-based vector
TYPE &getElement(std::size_t i) {
    return data[i - 1];
} // getElement()

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

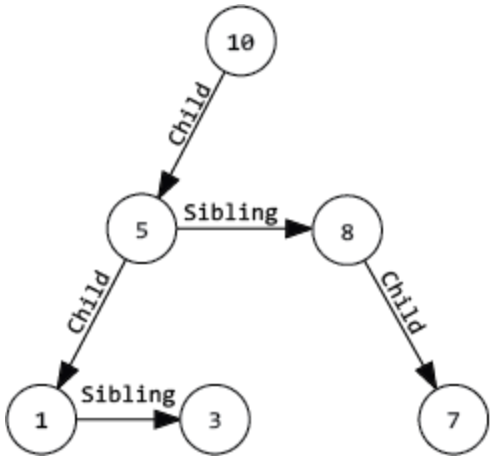
Pairing Priority Queue

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, as it's already completed for you.

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 test 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 code to this file for additional testing.

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 an efficient implementation of the binary heap which you are also coding in `BinaryPQ.h` (Part B). 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 objects, your functor would accept two pointers to objects (actually two `const` pointers, since you don't have to modify objects 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.

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**. It's important to realize that 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

We highly encourage the use of the STL for part A, 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::swap()`.
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 test files exposes bugs in your own solution, and for the first such file encountered we will provide you with the correct output and your program's output.

Each test file should consist of a valid `GAMEFILE`. 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-OPTION.txt` where 0 <= n <= 9. The autograder's buggy solutions will run your test files in the specified `OPTION`. Valid options are `v`, `m`, or `s`, only one can be used per input file (ie: a file named `test-1-v.txt` will run your test on a solution using the verbose flag). If you specify the `s` option, we will automatically pick a number of statistics for you (greater than zero).

Your test files may not generate more than 50 zombies (counting both random and explicitly specified zombies). Further, no test file can run more than 1000 rounds (to prevent running too long). 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 50 zombies or 1000 rounds; your solution should not impose any size limits (as long as sufficient system memory is available).

Testing the Priority Queues

We will be testing your priority queue implementations in isolation from the rest of your code (in addition to in the context of the zombie game). Specifically, we will be testing the construction (both empty and range) and destruction, push, and heap invariant (the highest priority element is always returned by the `top()` member function), and `updatePriorities()`. Additionally, your pairing heap will have the copy constructor, `operator=()`, `addNode()` and `updateElt()` functions tested. We strongly recommend that you create test files for your local use to evaluate whether or not your code is correct and performs well. For these tests, we recommend doing tests where update operations are much more frequent than removal operations. The Unordered PQ is already implemented properly.

When debugging, we highly recommend setting up your own system for quick, automated, regression testing. In other words, check your solution against the old output from your solution to see if it has changed. This will save you from wasting submits. You may find the Linux utility 'diff' useful as part of this.

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: 9504853406CBAC39EE89AA3AD238AA12CA198043`
- 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 with the starter files).
    - 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 `zombbb`. 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 (zombie simulation gamefiles), 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).

Grading

- 60 pts total for part A (all your code, using the STL, but not using your priority queues)
  - 45 pts — correctness & performance
  - 5 pts — no memory leaks
  - 10 pts — student-provided test files
- 40 pts total for part B (our main, your priority queues)
  - 20 pts — pairing heap correctness & performance
  - 5 pts — pairing heap has no memory leaks
  - 10 pts — binary heap correctness & performance
  - 5 pts — sorted heap correctness & performance

Although we will not be grading your code for style, we reserve the right to not help you in office hours if your code is unreadable.

It is **extremely helpful** to compile your code with the gcc options: -Wall -Wextra -pedantic -Werror -Wconversion. 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. This can make it a little more difficult to get your code compiling, but makes it less likely to have runtime errors.

Appendix A: Autograder Information

The test cases on the autograder follow a naming convention. For Part A the first letter describes the input file:

- `S`: The sample from the project specification, Appendix D.
- `M`: A medium sized input file
- `L`: A large input file
- `X`: An extra-large input file.

The last letter(s) tells you which options (if any) were used to run the test:

- `(nothing)`: No options, statistics and verbose are both off
- `m`: Median mode on
- `s`: Statistics are on
- `v`: Verbose mode is on

There are 14 bugs. You start earning points at 7 bugs, and earn full points for finding 12.

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).

Appendix B: Full Output with all modes

Here is a sample input and output of `zombbb` with the `--statistics`, `--mode` and `--verbose` flags. These are also included in the P2 sample files online.

spec.txt:

```
# Project 2 specification example
quiver-capacity: 10
random-seed: 2049231
max-rand-distance: 50
max-rand-speed: 60
max-rand-health: 1
---
round: 1
random-zombies: 15
named-zombies: 3
MarkSchlissel distance: 150 speed: 300 health: 15
MarySueColeman distance: 2 speed: 3 health: 6
LeeBollinger distance: 100 speed: 1 health: 100
---
round: 3
random-zombies: 10
named-zombies: 1
JamesDuderstadt distance: 20 speed: 10 health: 20
```

Full Output, when run with `./zombbb -s 10 -v -m < spec.txt`:

```
Round: 1
Created: paoletti0 (distance: 25, speed: 20, health: 1)
Created: darden1 (distance: 17, speed: 48, health: 1)
Created: garcia2 (distance: 15, speed: 57, health: 1)
Created: cris3 (distance: 8, speed: 9, health: 1)
Created: bing4 (distance: 11, speed: 45, health: 1)
Created: will5 (distance: 17, speed: 6, health: 1)
Created: fee6 (distance: 23, speed: 10, health: 1)
Created: noah7 (distance: 17, speed: 9, health: 1)
Created: potatobot8 (distance: 22, speed: 32, health: 1)
Created: paoletti9 (distance: 17, speed: 60, health: 1)
Created: darden10 (distance: 37, speed: 29, health: 1)
Created: garcia11 (distance: 21, speed: 19, health: 1)
Created: cris12 (distance: 35, speed: 7, health: 1)
Created: bing13 (distance: 5, speed: 22, health: 1)
Created: will14 (distance: 3, speed: 12, health: 1)
```

```
Created: fee15 (distance: 33, speed: 21, health: 1)
Created: noah16 (distance: 39, speed: 9, health: 1)
Created: potatobot17 (distance: 26, speed: 42, health: 1)
Created: paoletti18 (distance: 3, speed: 53, health: 1)
Created: darden19 (distance: 27, speed: 49, health: 1)
Created: garcia20 (distance: 26, speed: 52, health: 1)
Created: cris21 (distance: 8, speed: 26, health: 1)
Created: bing22 (distance: 10, speed: 16, health: 1)
Created: will23 (distance: 9, speed: 36, health: 1)
Created: fee24 (distance: 49, speed: 14, health: 1)
Created: MarkSchlissel (distance: 150, speed: 300, health: 15)
Created: MarySueColeman (distance: 2, speed: 3, health: 6)
Created: LeeBollinger (distance: 100, speed: 1, health: 100)
Destroyed: bing13 (distance: 5, speed: 22, health: 0)
Destroyed: bing22 (distance: 10, speed: 16, health: 0)
Destroyed: bing4 (distance: 11, speed: 45, health: 0)
Destroyed: cris21 (distance: 8, speed: 26, health: 0)
Destroyed: cris3 (distance: 8, speed: 9, health: 0)
Destroyed: darden1 (distance: 17, speed: 48, health: 0)
Destroyed: darden19 (distance: 27, speed: 49, health: 0)
Destroyed: garcia2 (distance: 15, speed: 57, health: 0)
Destroyed: garcia20 (distance: 26, speed: 52, health: 0)
Destroyed: paoletti18 (distance: 3, speed: 53, health: 0)
At the end of round 1, the median zombie lifetime is 1
Round: 2
Moved: paoletti0 (distance: 5, speed: 20, health: 1)
Moved: will5 (distance: 11, speed: 6, health: 1)
Moved: fee6 (distance: 13, speed: 10, health: 1)
Moved: noah7 (distance: 8, speed: 9, health: 1)
Moved: potatobot8 (distance: 0, speed: 32, health: 1)
Moved: paoletti9 (distance: 0, speed: 60, health: 1)
Moved: darden10 (distance: 8, speed: 29, health: 1)
Moved: garcia11 (distance: 2, speed: 19, health: 1)
Moved: cris12 (distance: 28, speed: 7, health: 1)
Moved: will14 (distance: 0, speed: 12, health: 1)
Moved: fee15 (distance: 12, speed: 21, health: 1)
Moved: noah16 (distance: 30, speed: 9, health: 1)
Moved: potatobot17 (distance: 0, speed: 42, health: 1)
Moved: will23 (distance: 0, speed: 36, health: 1)
Moved: fee24 (distance: 35, speed: 14, health: 1)
Moved: MarkSchlissel (distance: 0, speed: 300, health: 15)
Moved: MarySueColeman (distance: 0, speed: 3, health: 6)
Moved: LeeBollinger (distance: 99, speed: 1, health: 100)
DEFEAT IN ROUND 2! potatobot8 ate your brains!
Zombies still active: 18
First zombies killed:
bing13 1
bing22 2
bing4 3
cris21 4
cris3 5
darden1 6
darden19 7
garcia2 8
garcia20 9
paoletti18 10
Last zombies killed:
paoletti18 10
```

```
garcia20 9
garcia2 8
darden19 7
darden1 6
cris3 5
cris21 4
bing4 3
bing22 2
bing13 1
Most active zombies:
LeeBollinger 2
MarkSchlissel 2
MarySueColeman 2
cris12 2
darden10 2
fee15 2
fee24 2
fee6 2
garcia11 2
noah16 2
Least active zombies:
bing13 1
bing22 1
bing4 1
cris21 1
cris3 1
darden1 1
darden19 1
garcia2 1
garcia20 1
paoletti18 1
```

Appendix C: Project Tips (mostly PairingPQ, but others also)

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.

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

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 zombies are being randomly generated with the wrong name, distance, speed or health, it would be very useful to know that right away, rather than wondering why the wrong zombie killed you.

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).

When you're working on statistics output, be efficient. You can use something from this project to make the most/least active zombies more efficient.

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 do unit testing.

There are some helpful videos on the EECS 281 YouTube channel, specifically the running median: https://www.youtube.com/channel/UC9NgBiCdCl7SNEN-5Jq7wyA. There's also a video recorded by a student during office hours about the Pairing PQ: https://www.youtube.com/watch?v=irsHBpw2fhE

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.