EECS 281 Fall 2016

Project 2:
Star Wars Episode VIII: A New Heap

Due Friday, October 21 at 11:55pm

Table of Contents

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Part A: Galactic Warfare Simulation

Galaxy Logic Overview

Your program, named `galaxy`, will receive as input a series of "**battalion deployments**," or placements of troops on a certain planet. A **battalion deployment** consists of the following information:

- Timestamp — the time that this deployment order is issued
- General ID — the general who is issuing the deployment order
- Planet ID — the planet which the troops are being deployed to
- Jedi or Sith — Whether the general issuing the deployment is a Jedi or Sith (*Flavor note: in Star Wars, the Jedi are the good guys on the Light side of the Force, and the Sith are the bad guys on the Dark side of the Force. Click the link below for more info on the Force*.)
- Force Sensitivity — the average Force Sensitivity of the troops being deployed.
- Quantity — the number of troops being deployed.

As you read each battalion deployment from input, your program should see if the new battalion can be **matched** with a battalion previously deployed on the planet. If a match occurs, then the two battalions engage in warfare. A new battalion can be matched with a previously deployed battalion if:

- Both deployments are for the same planet.
    - General ID does not matter; generals are allowed to switch sides of the Force: i.e. order conflicting battalion deployments.
- The deployments were issued on different sides of the Force. That is, one deployment was a Jedi deployment and the other was a Sith deployment.
- The Jedi Force Sensitivity is **less than or equal** to the Sith Force Sensitivity.
    - The Sith always instigate fights, and they only fight battalions with Force Sensitivity which they are confident that they can overcome.
    - This does not mean that a battle will not occur if a Jedi battalion is being deployed. When a Jedi battalion is deployed, they may be ambushed by a previously deployed Sith battalion.

If the new battalion is a Sith battalion, it will **always** choose to attack the least Force-sensitive Jedi battalion on the planet, given that there is a Jedi battalion with lesser Force Sensitivity. If the new battalion is a Jedi battalion, it will **always** be ambushed by the most Force-sensitive Sith battalion on the planet, given that there is a Sith battalion with greater Force Sensitivity. In the event of a tie in Force Sensitivity, choose the battalion that was deployed first (came first in the input file).

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

When a battle occurs, the battalions trade troops one-for-one, regardless of their Force Sensitivity. That is to say that an equal number of troops from both battalions are eliminated, equal to the number of troops in the smaller battalion. If one of the battalions survives, it remains on the planet for future possible fights. For example, if a Sith battalion with 20 troops fights a Jedi battalion of 30 troops, the Sith battalion is eradicated and the Jedi battalion remains with 10 troops.

In the event that the newly deployed battalion survives, it is possible that a new fight will break out. If the new battalion is Sith, they will then look to attack another Jedi battalion. If the new battalion is Jedi, they may be attacked again after defeating the first Sith battalion. This happens until no more fights break out: that is, there are no pairs of Jedi and Sith battalions remaining on the planet such that the Sith Force Sensitivity is greater than or equal to the Jedi Force Sensitivity.

Input

Input will arrive from standard input (`cin`). There are two input formats, *deployment list* (`DL`) and *pseudorandom* (`PR`). The first four lines of input will always be in the following format, regardless of input format, which you may assume are correctly formatted:

`COMMENT: <COMMENT>`
`MODE: <INPUT_MODE>`
`NUM_GENERALS: <NUM_GENERALS>`
`NUM_PLANETS: <NUM_PLANETS>`

`<COMMENT>` is a string terminated by a newline, which should be ignored. (You should comment your test files to explain their purpose.)

`<INPUT_MODE>` will either be the string ”`DL`” or “`PR`”. DL indicates that the rest of input will be in the deployment list format, and PR indicates that the rest of input will be in pseudo-random format. Details for these input formats will be explained shortly.

`<NUM_GENERALS>` and **`<NUM_PLANETS>`**, respectively, will tell you how many generals and planets will exist.

Deployment List (DL) Input:

In Deployment List mode, the rest of the input will be a series of lines in the following format:

`<TIMESTAMP>` **`<SIT/JEDI>`** `G`**`<GENERAL_ID>`** `P`**`<PLANET_NUM>`** `F`**`<FORCE_SENSITIVITY>`** `#`**`<NUM_TROOPS>`**

Each line represents a unique deployment. For example, the line:

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

```
0 SITH G1 P2 F100 #50
```

Can be translated as:

“At timestamp 0, Sith general 1 deploys 50 troops with Force Sensitivity 100 to planet 2.”

Errors you must check for in DL Input mode

To detect corrupt deployment orders, you must check for each of the following:

- **<TIMESTAMP>** is a non-negative integer (you may assume it will be an integer)
- **<GENERAL_ID>** and **<PLANET_ID>** are integers in ranges **[0, <NUM_GENERALS>)** and **[0, <NUM_PLANETS>)**, respectively.
  - e.g. if **<NUM_GENERALS>** is 5, then valid general IDs are 0, 1, 2, 3, 4.
- **<FORCE_SENSITIVITY>** and **<NUM_TROOPS>** are positive (non-zero) integers.
- Timestamps are non-decreasing.
  - e.g. 0 cannot come after 1, but there can be multiple deployments with the same timestamp.

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.**

Pseudorandom (PR) Input:

If **<INPUT_MODE>** is PR, the rest of the input file will consist of these three lines in this format:

```
RANDOM_SEED: <SEED>
NUM_DEPLOYMENTS: <NUM_DEPLOYMENTS>
ARRIVAL_RATE: <ARRIVAL_RATE>
```

- **RANDOM_SEED** — An integer used to initialize the random seed.
- **NUM_DEPLOYMENTS** — The number of deployment orders to generate. You may assume that this value will fit in an `int`.
- **ARRIVAL_RATE** — An integer corresponding to the average number of deployments per timestamp.

You may assume PR input will always be correctly formatted!

Generating deployments with P2.h

In the project folder, we provide a pair of files to generate the deployments in PR mode. This is to make pseudo-random generation uniform across platforms. The class P2 contains the following function:

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

```
void P2::PR_init(std::stringstream& ss, int seed, int num_generals,
                 int num_planets, int num_deployments, int arrival_rate);
```

`P2::PR_init(...)` will set the contents of the stringstream argument (`ss`) so that you can use it just like you would `cin` for `DL` mode:

You may find the following C++ code helpful in reducing code duplication:

```
stringstream ss;
if (prMode)
    P2::PR_init(ss, seed, num_generals, num_planets, num_deployments,
rate);

// If prMode is on, read from the stringstream.  Otherwise, use cin.
istream &inputStream = prMode ? ss : cin;

while (inputStream >> <variables>) {
    // read and process deployments
}
```

DL & PR Comparison

The following two input files are in different modes, but should generate **the same deployments**.

```
COMMENT: DL mode generating some deployments.
MODE: DL
NUM_GENERALS: 3
NUM_PLANETS: 2
0 SITH G1 P0 F81 #11
1 SITH G1 P1 F92 #15
1 SITH G0 P1 F15 #26
2 SITH G0 P0 F7 #2
2 JEDI G2 P0 F71 #45
3 SITH G1 P1 F83 #20
3 SITH G2 P0 F75 #24
4 SITH G1 P1 F84 #17
4 JEDI G2 P1 F5 #50
4 SITH G0 P1 F13 #17
4 JEDI G0 P1 F58 #45
4 SITH G1 P0 F17 #49
```


Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

```
COMMENT: PR mode generating the same sequence of deployments.
MODE: PR
NUM_GENERALS: 3
NUM_PLANETS: 2
RANDOM_SEED: 3
NUM_DEPLOYMENTS: 12
ARRIVAL_RATE: 10
```

Example Scenario

Consider the following series of deployments**.** The first example shows the format in which your program will take input in DL mode. The second example explains what each line of input means.

```
0 SITH G1 P2 F100 #10
0 JEDI G2 P2 F10 #20
0 SITH G3 P2 F1 #10
```

Sith general 1 deploys battalion 1 with Force Sensitivity 100 on Planet 2 with 10 troops.
Jedi general 2 deploys battalion 2 with Force Sensitivity 10 on Planet 2 with 20 troops.
Sith general 3 deploys battalion 3 with Force Sensitivity 1 on Planet 2 with 10 troops.

Here is a detailed explanation of what battles would occur as a result of the above input:

1. Sith battalion #1 lands on Planet 2 with 10 troops with Force Sensitivity 100.
    - There are no other battalions on the planet yet, so they remain on standby.
2. Jedi battalion #2 lands on Planet 2 with 20 troops with Force Sensitivity 10.
    - Sith battalion #1 encounters Jedi battalion #2. The Sith battalion #1 has a higher Force Sensitivity, so they instigate a fight with Jedi battalion #2.
    - They do battle and both lose 10 troops, since the troops are exchanged one-for-one.
    - Jedi battalion #2 remains on the planet, while Sith battalion #1 has been wiped out. Battalion #2 can battle at a later time with their remaining 10 troops.
3. Sith battalion #3 lands on Planet 2 with 10 troops with Force Sensitivity 1.
    - They do not engage Jedi battalion #2 because their Force Sensitivity is lower.
    - They remain on the planet and wait to attack a Jedi battalion with lower Force Sensitivity

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Command Line Interface

Your program should take the following case-sensitive command-line options that will determine which types of output to generate. Details about each output mode are under the **Output Details** section.

- `-v, --verbose`
  An optional flag that indicates verbose output should be generated.
- `-m, --median`
  An optional flag that indicates median output should be generated.
- `-g, --general-eval`
  An optional flag that indicates that the general evaluation output should be generated.
- `-w, --watcher`
  An optional flag that indicates that movie watcher output should be generated.
- `-q, --queue`
  The type of priority queue to use in your driver program. Must be one of `JARJAR`, `SORTED`, `BINARY`, or `PAIRING`. If this flag is not set, the queue type defaults to `JARJAR`. This will become more clear after reading Part B of the spec.

Examples of legal command lines (the last 4 would require you to type or paste the input):

- `./galaxy < infile.txt > outfile.txt`
- `./galaxy --verbose --general-eval -q SORTED < infile.txt`
- `./galaxy --verbose --median > outfile.txt`
- `./galaxy --watcher`
- `./galaxy --general-eval --verbose`
- `./galaxy -vmgw -q BINARY`

Examples of illegal command lines:

- `./galaxy -q FIBONACCI`

We will not be specifically error-checking your command-line handling; however we expect that your program conforms with the default behavior of getopt_long. Incorrect command-line handling may lead to a variety of difficult-to-diagnose problems.

Any priority queues used in the simulation must be of the type specified on the command line. **It is an honor code violation to use a priority queue of a different type**; if you have not implemented that priority queue yet, you must immediately `exit(1)` rather than use a different priority queue.

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Output

The output generated by your program will depend on the command line options specified at runtime. With the exception of program startup and the end of program summary, all output is optional and should not be generated unless the corresponding command line flag is set.

Program startup

Your program should always print the following line **before** reading any deployments:

```
Deploying troops...
```

Verbose option

If and only if the **`--verbose/-v`** option is specified on the command line (see above), whenever a battle is completed you should print on a single line:

`General` **`<SITH_GENERAL_NUM>`**`'s battalion attacked General` **`<JEDI_GENERAL_NUM>`**`'s battalion on planet` **`<PLANET_NUM>. <NUM_TROOPS_LOST>`** `troops were lost.`

`<NUM_TROOPS_LOST>` is defined to be the total number of troops lost from both sides, or the number of Jedi troops lost + the number of Sith troops lost.

Example

Given the following list of deployments:

```
0 JEDI G1 P0 F125 #10
0 SITH G2 P0 F1 #100
0 JEDI G3 P0 F100 #10
0 JEDI G4 P0 F80 #10
0 SITH G5 P0 F200 #4
```

Note that **no battles** are possible until the **5th** battalion is deployed. When the 5th battalion is deployed, you should print:

```
General 5's battalion attacked General 4's battalion on planet 0. 8 troops were lost.
```

Median option

If and only if the **`--median/-m`** option is specified on the command line, at the times detailed in

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

the Galaxy Logic section, your program should print the current median troops lost in a battle for all planets in ascending order by planet ID. **If no battles have occurred on a specific planet, do not print the median message for that planet**. In the case that battles have occurred, you should print:

`Median troops lost on planet <PLANET_ID> at time <TIMESTAMP> is <MED_TROOPS_LOST>.`

If an even number of battles occur on a planet, take the average of the middlemost two to compute the median. **Note:** There will always be an even number of troops lost in a battle (the same number of troops from both sides), so the result will always be an integer, even when divided by two.

Example

Given the following battles:

```
General 5's battalion attacked General 4's battalion on planet 9. 8 troops were lost.
General 2's battalion attacked General 7's battalion on planet 9. 2 troops were lost.
```

The median lost troops for planet 9 after these two battles is ((2 + 8) / 2) = 5. If the timestamp changed, and the `--median` option was specified, your program should print:

```
Median troops lost on planet 9 at time 0 is 5.
```

Summary option

After all input has been read and all possible battles have been fought, the following output should **always** be printed without any preceding newlines before any optional end of day output:

```
---End of Day---
Battles: <NUM_BATTLES><NEWLINE>
```

<NUM_BATTLES> is the total number of battles that have happened over the course of the program.

General evaluation option

If and only if the **`--general-eval/-g`** option is specified on the command line, following the summary output, you should print the following line without any preceding newlines.

```
---General Evaluation---
```

Followed by lines in the following format for ***every*** general in ascending order (0, 1, 2, etc.), even if they did not engage in any battles:

`General <GENERAL_ID> deployed <NUM_JEDI> Jedi troops and <NUM_SITH> Sith`

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

`troops, and <NUM_SURVIVORS>/<NUM_DEPLOYED> troops survived.`

These numbers are troops across all planets. Example:

```
---General Evaluation---
General 0 deployed 40 Jedi troops and 0 Sith troops, and 20/40 troops survived.
General 1 deployed 0 Jedi troops and 0 Sith troops, and 0/0 troops survived.
General 2 deployed 60 Jedi troops and 30 Sith troops, and 44/90 troops survived.
```

Movie watcher option

As a fervent Star Wars fan, you want to find which pairs of battling Jedi and Sith deployments would have made for a maximally-exciting movie. For each planet, your job is to find the pairs of Jedi and Sith deployments that would have had the most “exciting” battles. The most “exciting” battle is one that has the greatest difference in Force Sensitivity between the battling parties. Note that a battle can only happen if the Jedi Force Sensitivity is less than or equal to the Sith Force Sensitivity.

You want to find the most exciting possible Sith **attack** and Sith **ambush** for each planet. A Sith **attack** occurs when the Jedi are deployed to the planet first, and the Sith are deployed to that planet afterward to attack the Jedi. A Sith **ambush** occurs when the Sith are deployed to the planet first, and they surprise Jedi that land on the planet afterward. One deployment comes “after” another one if it appears later in the file, even if the timestamps for the two deployments are the same.

For example, suppose you had these deployments:

```
0 JEDI G1 P0 F10 #10
0 SITH G2 P0 F20 #10
0 JEDI G1 P0 F30 #10
0 SITH G1 P0 F40 #10
```

Then the most exciting Sith attack on planet 0 would be the Sith with Force Sensitivity 40 attacking the Jedi with Force Sensitivity 10, but there would be no most-exciting Sith ambush because it’s not possible for a the Sith deployment to be paired against a Jedi deployment that came later.

Notice that an actual battle need not happen. In this output, the first and second deployments would be paired in the regular output, not the first and the fourth. You should report only the most exciting hypothetical battle, regardless of which battles actually occurred.

If and only if the **`--watcher/-w`** option is specified on the command line, you should print the following line without any preceding newlines once at the very end of the program:

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

`---Movie Watcher---`

Followed by a pair of movie watcher's output lines for every planet in ascending order in the following format:

`A movie watcher would enjoy an ambush on planet` **`<PLANET_ID>`** `with Sith at time` **`<TIMESTAMP1>`** `and Jedi at time` **`<TIMESTAMP2>.`**
`A movie watcher would enjoy an attack on planet` **`<PLANET_ID>`** `with Jedi at time` **`<TIMESTAMP1>`** `and Sith at time` **`<TIMESTAMP2>.`**

When finding exciting battles, the number of troops is not taken into consideration. If there would be more than one battle with the maximal level of excitement (i.e., difference in Force Sensitivity), you should prefer the battle with the lower **<TIMESTAMP2>**. If the second timestamps are equal, then use the lower **<TIMESTAMP1>**.

If there are no exciting battles (there are no pairs of Jedi/Sith deployments on a given planet, or none result in the Jedi Force Sensitivity being less than or equal to the Sith Force Sensitivity) you should print -1 for both the first and second timestamp.

Detailed Algorithm

Following these steps in order will help guarantee that your program prints the correct output at the proper times.

Note: `CURRENT_TIMESTAMP` starts at 0, and is maintained throughout the run of the program.

1. Print program startup output
2. Read the next deployment from input.
3. If the new deployment's `TIMESTAMP` is not the `CURRENT_TIMESTAMP`
    a. If the `--median` option is specified, print the median information
    b. Set `CURRENT_TIMESTAMP` to be the new deployment's `TIMESTAMP.`
4. Instigate all possible fights between the Jedi and Sith battalions.
    a. If the `--verbose` option is specified, you should print the details of each completed battle to `stdout/cout.`
5. Repeat steps 2-4 until there are no more deployments to be made.
6. Output median information **again** if the `--median` flag is set.
7. Print end-of-day summary output.
8. Output the general evaluation if the `--general-eval` flag is set.
9. Output the movie watcher's output if the `--watcher` flag is set.

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Full Example

Input File Contents:

```
COMMENT: Example full output.
MODE: DL
NUM_GENERALS: 3
NUM_PLANETS: 2
0 SITH G1 P0 F81 #11
1 SITH G1 P1 F92 #15
1 SITH G0 P1 F15 #26
2 SITH G0 P0 F7 #2
2 JEDI G2 P0 F71 #45
3 SITH G1 P1 F83 #20
3 SITH G2 P0 F75 #24
4 SITH G1 P1 F84 #17
4 JEDI G2 P1 F5 #50
4 SITH G0 P1 F13 #17
4 JEDI G0 P1 F58 #45
4 SITH G1 P0 F17 #49
```

Output when run with -v, -m, -g, and -w (line wrapping occurs in the document):

```
Deploying troops...
General 1’s battalion attacked General 2’s battalion on planet 0. 22 troops
were lost.
Median troops lost on planet 0 at time 2 is 22.
General 2’s battalion attacked General 2’s battalion on planet 0. 48 troops
were lost.
Median troops lost on planet 0 at time 3 is 35.
General 1’s battalion attacked General 2’s battalion on planet 1. 30 troops
were lost.
General 1’s battalion attacked General 2’s battalion on planet 1. 34 troops
were lost.
General 1’s battalion attacked General 2’s battalion on planet 1. 36 troops
were lost.
General 1’s battalion attacked General 0’s battalion on planet 1. 4 troops
were lost.
Median troops lost on planet 0 at time 4 is 35.
Median troops lost on planet 1 at time 4 is 32.
---End of Day---
Battles: 6
```


Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

```
---General Evaluation---
General 0 deployed 45 Jedi troops and 45 Sith troops, and 88/90 troops
survived.
General 1 deployed 0 Jedi troops and 112 Sith troops, and 49/112 troops
survived.
General 2 deployed 95 Jedi troops and 24 Sith troops, and 10/119 troops
survived.
---Movie Watcher---
A movie watcher would enjoy an ambush on planet 0 with Sith at time 0 and
Jedi at time 2.
A movie watcher would enjoy an attack on planet 0 with Jedi at time 2 and
Sith at time 3.
A movie watcher would enjoy an ambush on planet 1 with Sith at time 1 and
Jedi at time 4.
A movie watcher would enjoy an attack on planet 1 with Jedi at time 4 and
Sith at time 4.
```

Hints and Advice

The project is specified so that the various pieces of output are all independent. We strongly recommend working on them separately, implementing one command-line option at a time. The autograder has test cases are named so that you can get a sense of where your bugs might be.

We place a strong emphasis on time budgets in this project. This means that you may find that you need to rewrite sections of your code that are performing too slowly or consider using different data structures. Pay attention to the Big-O complexities of your implementation and examine the tradeoffs of using different possible solutions. Using `perf` on this project will be incredibly helpful in finding which parts of your code are taking up the most amount of time, and remember that `perf` is most effective when your code is well modularized (i.e. broken up into functions).

Running your code locally in `valgrind` can help you find and remove undefined (buggy) behavior and memory leaks from your code. This can save you from losing points in the final run when you mistakenly believe your code to be correct.

It is extremely helpful to compile your code with the following gcc options: `-Wall -Wextra -Wconversion -pedantic`. This way the compiler can warn you about parts of your code that may result in unintended/undefined behavior. Compiling with the provided Makefile does this for you.

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Part B: Priority Queues

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 "**Jar-Jar priority queue**" in **JarJarPQ.h**, which does a linear search for the most extreme element each time it is needed. 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.

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

Note: We will compile your priority queue implementations with our own code to ensure that you

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

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
```

Sorted Priority Queue

The *sorted priority queue* implements the priority queue interface by maintaining a **sorted** vector. Complexities and details are in SortedPQ.h.

Binary Heap Priority Queue

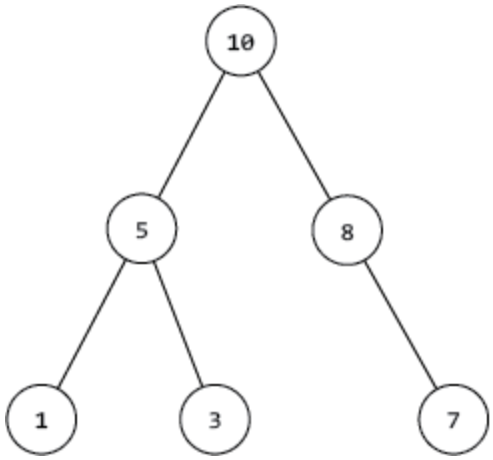
Binary heaps will be covered in lecture. We also highly recommend reviewing Chapter 6 of the CLRS book. Complexities and details are in BinaryPQ.h.

Pairing Priority Queue

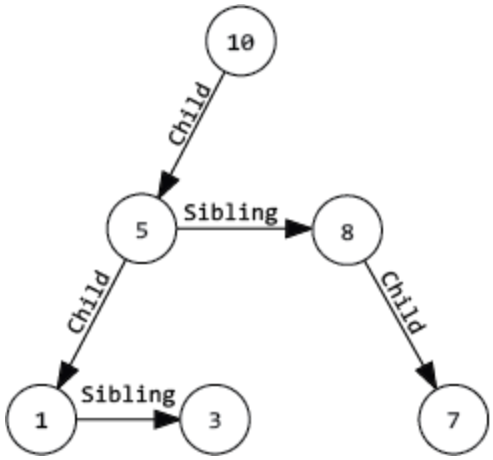
Pairing heaps are an advanced heap data structure that can be quite fast. In order to implement the pairing priority queue, read the two papers we provide you describing the data structure. Complexity details can be found in PairingPQ.h. We have also included a couple of diagrams that may help you understand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

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

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

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

accept parameters have had the name of the parameter commented out (otherwise you would get an unused parameter error). Look at `JarJarPQ.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 `JarJarPQ` 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()`, similar to the way that `JarJarPQ` passes it to `max_element()`.

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 `JarJarPQ` to use. 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 troop deployments, your functor would accept two pointers to troop deployments (actually two `const` pointers, since you don't have to modify deployments 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 section on "Galaxy Logic Overview": if you have multiple Sith on the same planet, which one has the highest priority if a fight occurs? Jedi will have a different way of determining who has the highest priority. This means you need at least two different functors: one for a priority queue full of Sith, and a different functor used by a PQ full of Jedi.

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**, not just for your galactic simulation, thus, 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 check `if (data[i] > data[j])`? Use the following:

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

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

Compiling and Testing Priority Queues

You are provided with a Makefile similar to the one provided in project0 and 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. You should add tests to this file or make your own test cases in this form.

Using the 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).

Part C: Logistics

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).

In addition to the above requirements, you may not use any STL facilities which trivialize your implementation of your priority queues, including but not limited to `priority_queue`, `make_heap`, `push_heap`, `pop_heap`, `sort_heap`, `partition`, `partition_copy`, `stable_partition`, `partial_sort`, or `qsort`. However, you may (and probably should) use `sort`. If you are unsure about whether a given function may be used, ask on Piazza.

Testing and Debugging

Part of this project is to prepare several test files that expose defects in buggy solutions.

Each test file is an input file. We will give your test files as input to intentionally buggy solutions and compare the output to that of a correct project solution. You will receive points depending on how many buggy implementations your test files expose.

The autograder will also tell you if one of your test files exposes bugs in your solution.

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

Test File Details

Your test files must be Deployment List input files, **and may have no more than 30 lines** in any one file. You may submit up to 10 test files (though it is possible to expose all buggy solutions with fewer test files).

Test files should be named in the following format:

`test-<N>-<FLAG>.txt`

- **<N>** is an integer in the range [0, 9]
- **<FLAG>** is one (and only one) of the following characters `v`, `m`, `g`, or `w`.
    - This tells the autograder which command-line option to test with.
    - Note that you cannot define a priority queue to use. These tests do not include solutions with buggy priority queues

`test-1-w.txt` and `test-5-v.txt` are both valid test file names.

Note: the tests on which the autograder runs your solution are NOT limited to 30 lines in a file; your solution should not impose any size limits (as long as 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:

- You have deleted all .o files and your executable(s). Typing ‘`make clean`’ shall accomplish this.
- Your makefile is called Makefile. Typing ‘`make -R -r`’ builds your code without errors and generates an executable file called "`galaxy`". (Note that 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 speed up code by an order of magnitude.
- Your test files are named as described and no other project file names begin with test. Up to 10 test files may be submitted.
- The total size of your program 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 (e.g., the .git folder used by git source code management).

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

- Your code compiles and runs correctly using version 5.1.0 of the g++ compiler on the CAEN servers. To compile with g++ version 5.1.0 on CAEN you **must** put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-5.1.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-5.1.0/lib64
LD_RUN_PATH := /usr/um/gcc-5.1.0/lib64
```

Turn in all of the following files:

- All your .h and or .cpp files for the project
- 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. In this directory, run

```
dos2unix -U *; tar -czf ./submit.tar.gz *.cpp *.h Makefile test-*.txt
```

This will prepare a suitable file in your working directory. Alternatively, the sample makefile has a useful `submit` that will do this for you.

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/. **Note that 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 last submission for your grade.

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 note if the autograder shows that one of your own test files exposes a bug in your solution.

Grading

- 60 pts total for part A (all your code)
  - 50 pts — correctness & performance
  - -5 pts — memory leaks
  - 10 pts — student-provided test files

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson

- 40 pts total for part B (our main, your priority queues)
    - 25 pts — pairing heap correctness & performance
    - -5 pts — pairing heap 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. Readability is generally defined as follows:

- Clean organization and consistency throughout your overall program
- Proper partitioning of code into header and cpp files
- Descriptive variable names and proper use of C++ idioms
- Omitting globals, `goto`s, unnecessary literals, or unused libraries
- Effective use of comments
- Reasonable formatting - e.g an 80 column display
- Code reuse/no excessive copy-pasted code blocks

Current version by: Luum Habtemariam, Rishin Doshi, and Waleed Khan
Originally composed by: Mark Isaacson