EECS 281 Winter 2022

Project 2:
Star Wars Episode X: A New Heap

Due Tuesday, February 22 at 11:59pm

Current version by: David Paoletti, Luum Habtemariam, Rishin Doshi, and Waleed Khan; Originally by: Mark Isaacson

Important Note

There are two parts to this project. In your IDE, make a separate "IDE Project" for each part. If you try to create a single project, you run into a problem of having two copies of `main()`, which won't compile, etc.

Part A: Galactic Warfare Simulation

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

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).

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> <SITH/JEDI> G<GENERAL_ID> P<PLANET_NUM> F<FORCE_SENSITIVITY> #<NUM_TROOPS>
```

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

```
0 JEDI G0 P1 F100 #44
```

Can be translated as:

“At timestamp 0, Jedi general 0 deploys 44 Jedi troops with Force-sensitivity 100 to planet 1.”

DL Input Error Checking
To detect corrupt deployment orders, you must check for each of the following:

- **`<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 greater than 0.
- 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 input 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 P2random.h

We provide a file to generate the deployments in PR mode. This is to make pseudo-random generation uniform across platforms. The `P2random` class contains the following function:

```
void P2random::PR_init(std::stringstream &ss,        unsigned int seed,
                       unsigned int num_generals,    unsigned int num_planets,
                       unsigned int num_deployments, unsigned int arrival_rate);
```

`P2random::PR_init(...)` will fill the stringstream argument (`ss`) with deployments, 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;

// inputMode is the "PR" or "DL" from line 2
if (inputMode == "PR") {
    // TODO: add code to read random seed, number of deployments, and arrival rate
    P2random::PR_init(ss, seed, num_gen, num_planets, num_deploys, rate);
} // if

// Create a reference variable that is ALWAYS used for reading input.
// If PR mode is on, refer to the stringstream.  Otherwise, refer to cin.
// This is a place where the ternary operator must be used: an equivalent
// if/else is impossible because reference variables must be initialized
// when they are created.
istream &inputStream = inputMode == "PR" ? ss : cin;

// Make sure to read an entire deployment in the while statement
while (inputStream >> var1 >> var2 ...) {
    // Process this deployment, use PQs, make fights happen (if possible), etc.
} // while
```

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 JEDI G0 P1 F100 #44
1 JEDI G0 P1 F56 #42
2 SITH G0 P1 F73 #19
2 SITH G0 P0 F34 #50
2 JEDI G0 P0 F86 #23
2 JEDI G0 P0 F20 #39
2 SITH G2 P0 F49 #24
2 JEDI G2 P0 F83 #45
3 JEDI G2 P1 F64 #22
3 JEDI G1 P0 F6 #19
3 JEDI G0 P1 F42 #37
4 JEDI G2 P0 F10 #44
```

```
COMMENT: PR mode generating the same sequence of deployments.
MODE: PR
NUM_GENERALS: 3
```

```
NUM_PLANETS: 2
RANDOM_SEED: 104
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

Command Line Flags

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.

Examples of legal command lines:

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

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.

Output Details

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

```
Deploying troops...
```

Verbose Output
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
```

No battles are possible until the **5th** battalion is deployed. When the 5th battalion is deployed and a battle occurs, you should print:

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

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

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 Output
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 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 Output
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. 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 :

```
---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> with a force difference of <NUM>.
A movie watcher would enjoy an attack on planet <PLANET_ID> with Jedi at time
<TIMESTAMP1> and Sith at time <TIMESTAMP2> with a force difference of <NUM>.
```

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.

The `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.

Full Example

Input File Contents:

```
COMMENT: DL mode generating some deployments.
MODE: DL
NUM_GENERALS: 3
NUM_PLANETS: 2
0 JEDI G0 P1 F100 #44
1 JEDI G0 P1 F56 #42
```

```
2 SITH G0 P1 F73 #19
2 SITH G0 P0 F34 #50
2 JEDI G0 P0 F86 #23
2 JEDI G0 P0 F20 #39
2 SITH G2 P0 F49 #24
2 JEDI G2 P0 F83 #45
3 JEDI G2 P1 F64 #22
3 JEDI G1 P0 F6 #19
3 JEDI G0 P1 F42 #37
4 JEDI G2 P0 F10 #44
```

Output when run with -v, -m, -g, and -w:

```
Deploying troops...
General 0's battalion attacked General 0's battalion on planet 1. 38 troops were lost.
General 0's battalion attacked General 0's battalion on planet 0. 78 troops were lost.
Median troops lost on planet 0 at time 2 is 78.
Median troops lost on planet 1 at time 2 is 38.
General 2's battalion attacked General 1's battalion on planet 0. 38 troops were lost.
Median troops lost on planet 0 at time 3 is 58.
Median troops lost on planet 1 at time 3 is 38.
General 2's battalion attacked General 2's battalion on planet 0. 10 troops were lost.
General 0's battalion attacked General 2's battalion on planet 0. 22 troops were lost.
Median troops lost on planet 0 at time 4 is 30.
Median troops lost on planet 1 at time 4 is 38.
---End of Day---
Battles: 5
---General Evaluation---
General 0 deployed 185 Jedi troops and 69 Sith troops, and 127/254 troops survived.
General 1 deployed 19 Jedi troops and 0 Sith troops, and 0/19 troops survived.
General 2 deployed 111 Jedi troops and 24 Sith troops, and 95/135 troops survived.
---Movie Watcher---
A movie watcher would enjoy an ambush on planet 0 with Sith at time 2 and Jedi at time 3
with a force difference of 43.
A movie watcher would enjoy an attack on planet 0 with Jedi at time 2 and Sith at time 2
with a force difference of 29.
A movie watcher would enjoy an ambush on planet 1 with Sith at time 2 and Jedi at time 3
with a force difference of 31.
A movie watcher would enjoy an attack on planet 1 with Jedi at time 1 and Sith at time 2
with a force difference of 17.
```

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 some test cases which 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 -Werror -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.

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 “**Unordered priority queue**” in **UnorderedPQ.h**, which does a linear search for the most extreme element each time it is requested. You can use this priority queue for testing until you implement a more effective priority queue. You can also use this priority queue to ensure that your other priority queues are returning elements in the correct order. Any implementation of a priority queue should give the priority order no matter which implementation you’re using.

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

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

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

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

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

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

Eecs281PQ interface

Functions:

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

const TYPE &top()           //returns the highest priority element
                            //in the priority_queue

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

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

bool 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.

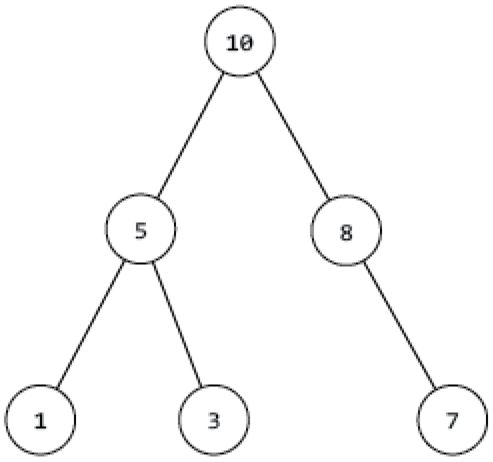
Binary Heap Priority Queue

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.

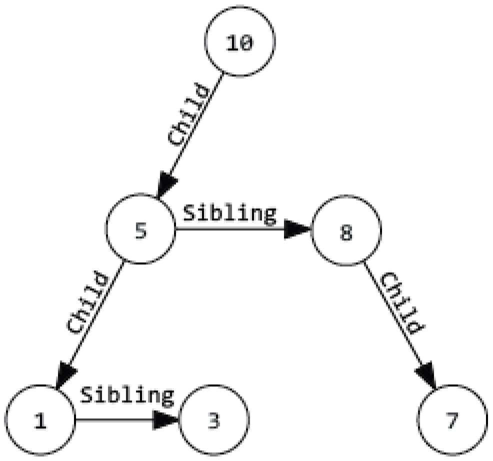
Pairing Priority Queue

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.

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



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



Implementing the Priority Queues

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

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

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

Compiling and Testing Priority Queues

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

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

Logistics

The `std::priority_queue<>`

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

1) The data type to be stored in the container. If this type has a natural sort order that meets your needs, this is the only type required.

2) The underlying container to use, usually just a vector<> of the first type.
3) The comparator to use to define what is considered the highest priority element.

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

```
priority_queue<int> pqMax;
```

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

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

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

About Comparators

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

Your functor receives two parameters, let's call them `a` and `b`. It must always answer the following question: **is the priority of `a` less than the priority of `b`**? What does lower priority mean? It depends on your application. For example, refer back to the section on "Galaxy Logic Overview": if you have multiple Jedi deployed on the same planet, which Jedi has the highest priority if a Sith arrives? In the same way, Sith have a different way of determining the highest priority. This means you will need at least two different functors: one for a priority queue containing Jedi and a different functor for a priority queue containing Sith.

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

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

You would instead write:

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

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

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

Libraries and Restrictions

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()`. Your main program (Part A) **must** use `priority_queue<>`, but your PQ implementations (Part B) **must not**. If you are unsure about whether a given function or container may be used, ask on Piazza.

Testing and Debugging

Part of this project is to prepare several test files that expose defects in a solution. 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. For the first such file that the autograder finds, it will give you the correct output and the output of your program.

Test File Details

Your test files must be Deployment List input files, **and may have no more than 30 deployments** 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.

For example, `test-0-w.txt` and `test-5-v.txt` are both valid test file names.

The tests on which the autograder runs your solution are NOT limited to 30 deployments 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:

- Every source code and header file contains the following project identifier in a comment at the top of the file:
- `// Project Identifier: AD48FB4835AF347EB0CA8009E24C3B13F8519882`
- 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 from Canvas instead.
- Your makefile is called `Makefile`. Typing ‘`make -R -r`’ builds your code without errors and generates an executable file called "`galaxy`". (The command-line options -R and -r disable automatic build rules; these automatic rules do not exist 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).
- Your code compiles and runs correctly using 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 running on CAEN Linux. At the moment, the default version installed on CAEN is 4.8.5, however we want you to use version 6.2.0 (available on CAEN with a command and/or Makefile); this version is also installed on the autograder machines.

For Part A (galactic combat simulation), turn in all of the following files:

- All your .h and or .cpp files for the project (NOT including your priority queue implementations)
- Your Makefile
- Your test files

For Part B (priority queues), turn in all of the following files:

- Your priority queue implementations: `SortedPQ.h`, `BinaryPQ.h`, `PairingPQ.h`
- Your `Makefile` (it includes itself by default, but we’ll replace this with our `Makefile`)

If **any** of your submitted priority queue files do not compile, **no** unit testing (Part B) can be performed. There are no student test files for Part B, only for Part A.

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. Our Makefile provides the command `make fullsubmit`. Alternately you can go into this directory and run this command:

```
tar czf ./submit.tar.gz *.cpp *.h *.hpp Makefile test*.txt
```

This 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/. **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 two times per calendar day, per part (A or B; more per day in Spring). If you use a late day to extend one part, the other part is automatically extended also. For this purpose, days begin and end at midnight (Ann Arbor local time). **We will use your best submission for final grading**. If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form.
Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to check whether 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, using the STL, but not using your priority queues)
  - 45 pts — correctness & performance
  - 5 pts — no memory leaks
  - 10 pts — student-provided test files to find bugs in our code (and yours!)
- 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

In your autograder output for Part A, the section named "Scoring student test files" will tell you how many bugs exist, how many are needed to start earning points, earn full points, and earn an extra submit per day (for Part A).

Part A Test Case Legend

Some of the test cases on the autograder follow a pattern and others do not. The test files that do not fit this pattern you can think of as handwritten cases which test a particular aspect of your program, though some can be very large.

INV*: Invalid test cases, derived from the “Errors you must check for” section.
DlSamp*: The example from the project specification, in DL mode.
PrSamp*: The example from the project specification, in PR mode.
Pr*: PR mode files, of a varying number of deployments.
K*: DL mode files, which contain a large number of deployments.
M*: DL mode files, which tend to be small-medium number of deployments.
F*: DL or PR mode files, which tend to have a small-medium number of trades.

A: Runs with all flags turned on: `-i -m -t -v`.
i, *m*, *t*, *v*: Run with just one of the flags (`-i` *or* `-m` *or* `-t` *or* `-v`).

Part B Test Case Legend

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 (P) uses the `.push()` member function to insert numerous values into your priority queue. After each push, the top will be tested for correctness. After all values have been pushed correctly, 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 (R) 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 (U) 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 (A) 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 (E) 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 (C) and operator= (O) 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 also ensuring that a deep copy was performed, not a shallow copy).

Hints for the PQ Implementations

When you implement the priority queues, read **UnorderedPQ.h** (and **UnorderedFastPQ.h**) first, understand them. Write `SortedPQ<>` first, then `BinaryPQ<>`, then `PairingPQ<>`. Remember to modify the testPQ.cpp file, add more tests to it, specifically for the `PairingPQ<>`. When writing the

PQ implementations, remember that each header file must #include whatever it needs, don't count on `main()` doing it for you, since we will be testing your PQ implementations with our `main()`. For example, if you need `swap()` for a particular PQ be sure to `#include <utility>` inside that PQ.

When compiling your unit testing (`make testPQ` for `testPQ.cpp`), it's important to realize how templates work: if the code in your cpp file doesn't call a particular member function, such as `updatePriorities()`, the compiler doesn't even compile it!

When you're implementing your PQs, remember that the `vector` class has a `.back()` member function! Don't write `data[data.size() - 1]`, it's much harder to read.

For the `BinaryPQ<>`, when you want to translate from 1-based to 0-based indexing, there's 3 options that you can consider:

1. Change all the formulas from the slides to be 0-based. Hardest but best option.
2. Push a "dummy" element at the start of the vector, and then lie about the size, empty, etc. Easy but poor option (it wastes space, is sometimes hard to get the syntax correct, and adds some confusion because the size of the data vector is not the same as the size of the PQ).
3. Use the functions provided below, put them inside the private section of the class declaration. When the slides say to access `heap[k]`, use `getElement(k)` instead. Easy and 2nd-best solution.

```
// 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()
```

Remember, don't leave the Pairing Heap until the last minute! Be reading and thinking about it before you start coding, and draw out examples on paper to help while you're coding. Here's some more tips:

- You can add other private member variables and functions, such as the current size and a `meld()` function.
- Make sure ALL of your constructors initialize ALL of your member variables.
- 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.
- You are not allowed to change from sibling and child pointers to a `std::deque<>` or `std::vector<>` of children pointers: this is slower than using sibling/child., and the autograder timings are based on the sibling/child approach.
- You are allowed (and need) 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.

- 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 lecture 5.
- You are allowed to write a public 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 (see below)!
- When you copy a pairing heap, it must match in values and behavior, but not necessarily structure.
- Don't use recursion, it's very easy to have a tree structure that causes a stack overflow.

What was the Project 1 approach?

- Add the “start” location to a deque
- While the deque is not empty:
  - Set the current location to the “next” in the deque
  - Pop the “next” location from the deque
  - Add current's “nearby” locations to the deque
  - Do something with the current location (print it, copy it, destroy it, etc.)

What is the “start” of a Pairing Heap? The root. What is “nearby” the current node? Its child and sibling, but NOT the parent/previous: this would violate the Project 1 approach of never adding anything to the deque twice. In P1 you skipped adding invalid choices, in P2 skip adding null pointers.

Final Notes, Including Video Links

If you’re asked about “cool projects” in an interview, note that this is simply a different wrapper on a more “marketable” project: stock market simulation. Sith and Jedi are like buyers and sellers, planets are commodities, and generals are stock market traders!

The “movie watcher mode” is a particular type of algorithm called a “streaming algorithm”, where we can only look at each input once, and cannot make a copy of all the input. This is similar to a stock market simulation also, where if we could go back in time, we could determine the optimal point at which to buy and then later sell the same commodity (planet) to achieve the highest profit margin.

Our EECS 281 YouTube channel has help for Part A in general, including most modes, one that discusses how to keep a faster running median, and another that explains movie watcher mode. There's also a video about Part B, especially Pairing Heap.