Assignment 6. Data Sagas

Due Friday, February 18 at 10:30 am

- Submissions received by the due date receive a small **on-time bonus**.
- All students are granted a pre-approved extension or "grace period" of 24 hours after the due date. Late submissions are accepted during the grace period with no penalty.
- The **grace period expires Sat, Feb 19 at 10:30 am**, after which we cannot accept further late submissions.
- All due dates and submission times are expressed in Pacific time.

You are permitted to work on this assignment in pairs.

Parts of this assignment adapted from one by Julie Zelenski and Jerry Cain.

We've spent a lot of time talking about searching, sorting, and runtime complexity. This assignment is designed to give you a sense of how to combine those ideas together in the service of something larger: diving deep into data sets. Over the course of this assignment, you'll build out a set of algorithms and data structures for processing large data sets. Once you've gotten them working, you'll get to see them in action as they power four data analyses.

This assignment has three parts:

- ***Array Exploration***: This debugging exercise is designed to get you comfortable exploring arrays in memory and seeing what lies beyond them.
- *HeapPQueue*: This workhorse of a data structure is useful for finding the best objects of various types. It's a powerful tool in its own right and can be used as a building block in other algorithms.
- ***Apportionment***: An algorithm for assigning seats in the US House of Representatives based on census results.

As usual, we suggest making slow and steady progress. Here's our recommended timetable:

- Aim to complete the debugging exercise within one day of this assignment going out.
- Aim to complete HeapPQueue within five days of this assignment going out.
- Aim to start Streaming Top-k within six days of this assignment going out.

Assignment Logistics

Starter Files

We provide a ZIP of the starter project. Download the zip, extract the files, and double-click the `.pro` file to open the project in Qt Creator.

Starter code

Getting Help

Keep an eye on the Ed forum for an announcement of the Assignment 6 **YEAH** (YEAH = Your Early Assignment Help) group session where our veteran section leaders will answer your questions and share pro tips. We know it can be daunting to sit down and break the barrier of starting on a substantial programming assignment – come to YEAH for advice and confidence to get you on your way!

We also here to help if you get run into issues along the way! The Ed forum is open 24/7 for general discussion about the assignment, lecture topics, the C++ language, using Qt, and more. Always start by searching first to see if your question has already been asked and answered before making a new post.

To troubleshoot a problem with your specific code, your best bet is to bring it to the LaIR helper hours or office hours.

Part One: Beyond the Bounds of Arrays

The first part of this assignment deals with pointers and dynamic allocation, and that introduces some new types of errors you'll need to learn to smoke out using the debugger. To get more familiar with what's going on, we're going to ask you to work the debugger to explore arrays and what you can find when you walk off the end of them.

Open the source file `ExploreArrays.cpp`. That file contains a function named `exploreArrays`. You'll trace this function in the debugger. The reason we'd like you to explore this function is that we'd like you to see what arrays look like in memory in a controlled environment before you encounter memory issues "in the wild" (that is, when writing up your own code). It's similar to how we wanted you to explore stack overflows in a simpler setting before turning you loose on writing your own recursive code.

Deliverables

Here’s what you need to do.

1. Set a breakpoint at the top of `exploreArrays` in `ExploreArrays.cpp`.
2. Run the program with the debugger turned on, choose the “Explore Arrays” option from the top menu, then click “Go!” to trigger the breakpoint. Follow the instructions in `ExploreArrays.cpp` and write your answers in `ShortAnswers.txt`.

Part Two: Priority Queues and Binary Heaps

Refresher: Priority Queues

As a refresher from Assignment 2, a ***priority queue*** is a modified queue in which elements are not dequeued in the order in which they were inserted. Instead, elements are removed from the queue in order of *priority*. For example, you could use a priority queue to model a hospital emergency room: patients enter in any order, but more critical patients are seen before less critical patients, regardless of how long the less-critical patients have been waiting. Similarly, if you were building a self-driving car that needed to process events, you might use a priority queue to respond to important messages (say, a pedestrian just walked in front of the car) before less important messages (say, a car switched on its turn signal).

For starters, here’s a refresher on the `DataPoint` type, which you first saw in Assignment 5:

```
struct DataPoint {
    string name;   // Name of this data point; varies by application
    double weight; // "Weight" of this data point. Points are sorted
by weight.
};
```

In this assignment, you'll implement your own priority queue type, **HeapPQueue**. Here’s the interface for the **HeapPQueue** type (we’ll explain the name in a minute):

```
class HeapPQueue {
public:
    HeapPQueue();
    ~HeapPQueue();
    void enqueue(const DataPoint& data);
    DataPoint dequeue();
    DataPoint peek() const;

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();

private:
    /* Details coming soon! */
};
```

Looking purely at the interface of this type, it sure looks like you're dealing with a queue. You can enqueue elements, dequeue them, and peek at them. The difference between a priority queue and a regular queue is the order in which the elements are dequeued. In a regular **`Queue`**, elements are lined up in sequential order, and calling **`dequeue()`** or **`peek()`** gives back the element added the longest time ago. In the **`HeapPQueue`**, the element that's returned by **`dequeue()`** or **`peek()`** is the element that has the ***lowest weight***. For example, let's imagine we set up a **`HeapPQueue`** like this:

```
HeapPQueue hpq;
hpq.enqueue({ "Amy",     103 });
hpq.enqueue({ "Ashley", 101 });
hpq.enqueue({ "Anna",    110 });
hpq.enqueue({ "Luna",    161 });
```

If we write

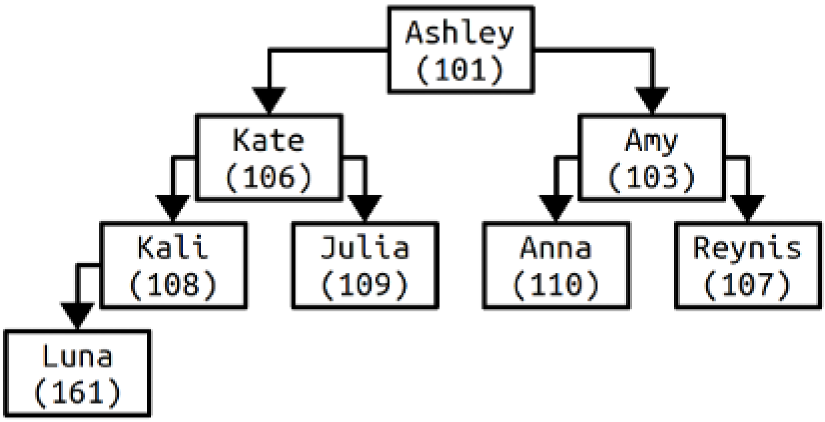
```
DataPoint data = hpq.dequeue();
cout << data.name << endl;
```

then the string printed out will be **`Ashley`**, since of the four elements enqueued, the weight associated with Ashley (101) was the lowest. Calling **`hpq.dequeue()`** again will return **`{"Amy", 103}`**, since her associated weight (103) was the lowest of what remains. Calling **`hpq.dequeue()`** a third time would return **`{"Anna", 110}`**.

Let's insert more values. If we call **`hpq.enqueue({"Chioma", 103})`** and then **`hpq.dequeue()`**, the return value would be the newly-added **`{"Chioma", 103}`** because her associated weight is lower than all others. And note that Chioma was the most-recently-added TA here; just goes to show you that this is quite different from a regular **`Queue`**!

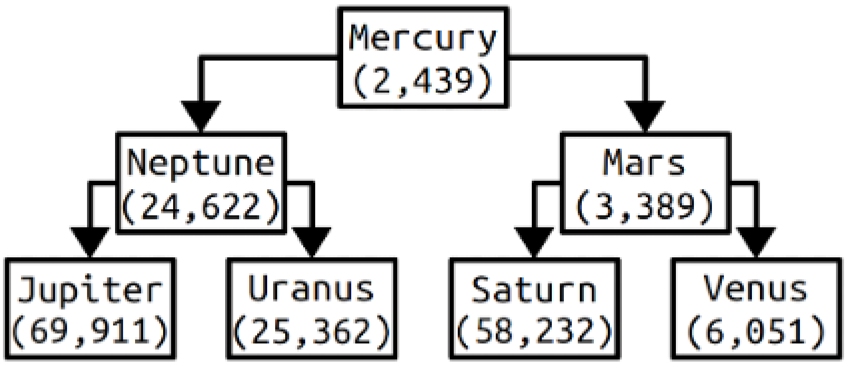
Background Binary Heaps

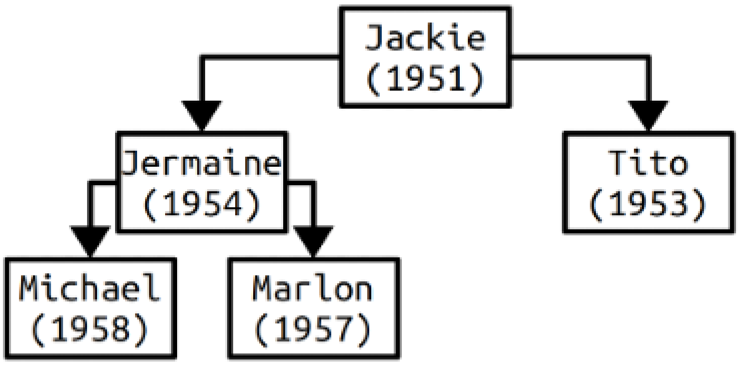
You'll implement HeapPQueue using a data structure called a ***binary heap***, hence the name HeapPQueue. Binary heaps are best explained by example. Below is a binary heap containing a collection of current and former TAs, each of whom is associated with a number corresponding to the class that they TAed for.



Let's look at the structure of this heap. Each value in the heap is stored in a ***node***, and each node has zero, one, or two ***child nodes*** descending from it. For example, Ashley has two children (Kate and Amy) while Kali has just one child (Luna) and Julia has no children at all.

In a binary heap, we enforce the rule that ***every row of the heap, except for the last, must be full***. That is, the first row should have one node, the second row two nodes, the third row four nodes, the fourth row eight nodes, etc., up until the last row. Additionally, that last row must be filled from the left to the right. You can see this in the above example – the first three rows are all filled in, and only the last row is partially filled. Here are two other examples of binary heaps, each of which obey this rule:





We also enforce one more property – ***no child node's weight is less than its parent's weight***. All three of the heaps you've seen so far obey this rule. However, there are no guarantees about how nodes can be ordered within a row; as you can see from the examples, within a row the ordering is pretty much arbitrary.

To recap, here are the three rules for binary heaps:

- Every node in a binary tree has either zero, one, or two children.
- No child's weight is less than the weight of its parent.
- Every row of the heap, except the last, is completely full, and the last row's elements are as far to the left as possible.

Before moving on, edit the file `ShortAnswers.txt` with answers to the following question.

Concept Check: Binary Heaps

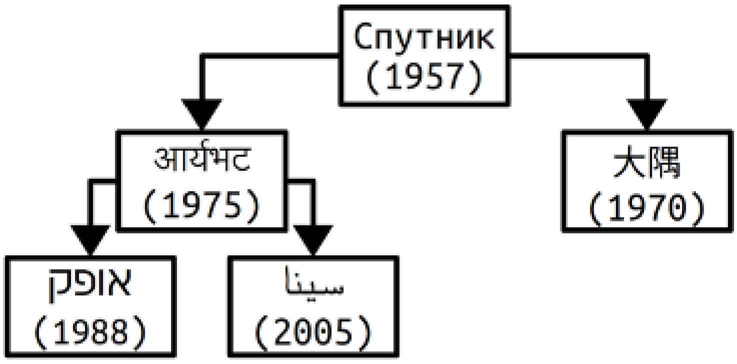
Q3: Draw three different binary heaps containing the DataPoints given in the example on the right-hand side above (the one containing Jackie, Jermaine, etc.) Yes, we know that we're asking you to draw pictures in a text file. Here's an example of what that could look like:

```
          Person A
           Weight
        /        \
    Person B     Person C
     Weight       Weight
    /        \
Person D     Person E
 Weight       Weight
```

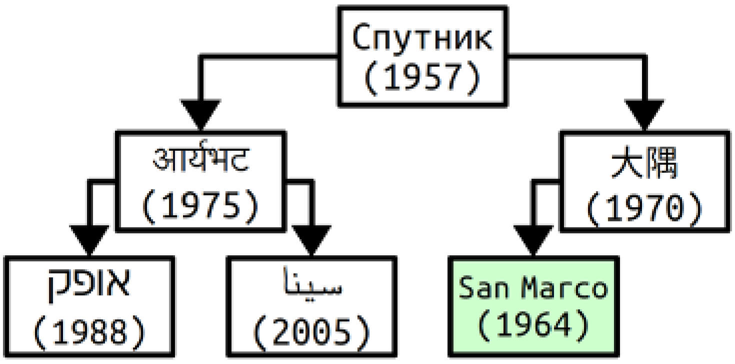
Maya Ziv, section leader extraordinaire, suggests using this website to draw the binary heaps.

Enqueuing Into a Binary Heap

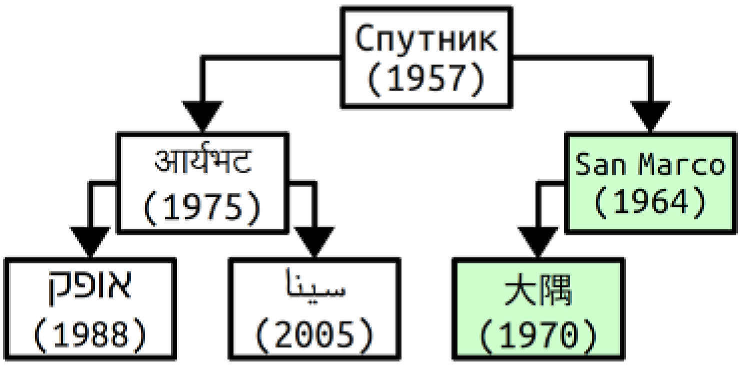
It is easy to read off which element has the smallest weight in a binary heap – it's the one at the top. It is also efficient to insert an element into a binary heap. Suppose, for example, that we have this binary heap containing some famous satellites:



Let's add the San Marco, the first Italian satellite, to this heap with weight 1964. Since a binary heap has all rows except the last filled, the only place we can initially place San Marco is in the first available spot in the last row. This is as the left child of Japan’s first space probe 大隅, so we place the new node for San Marco there:

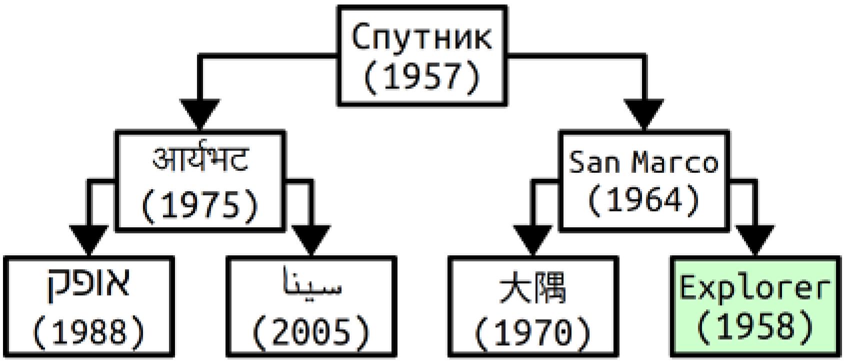


At this point, the binary heap is invalid because San Marco’s weight (1964) is less than that of 大隅 (1970). To fix this, we run a ***bubble-up*** step and continuously swap San Marco with its parent node until its weight is at least that of its parent’s. This means that we exchange San Marco and 大隅, shown here:

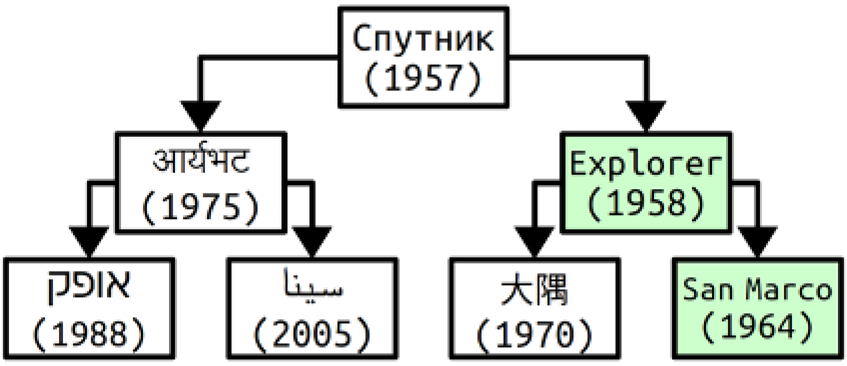


Since San Marco's weight (1964) is greater than its parent's weight (1957), it's now in the right place, and we're done. We now have a binary heap containing all of the original values, plus San Marco.

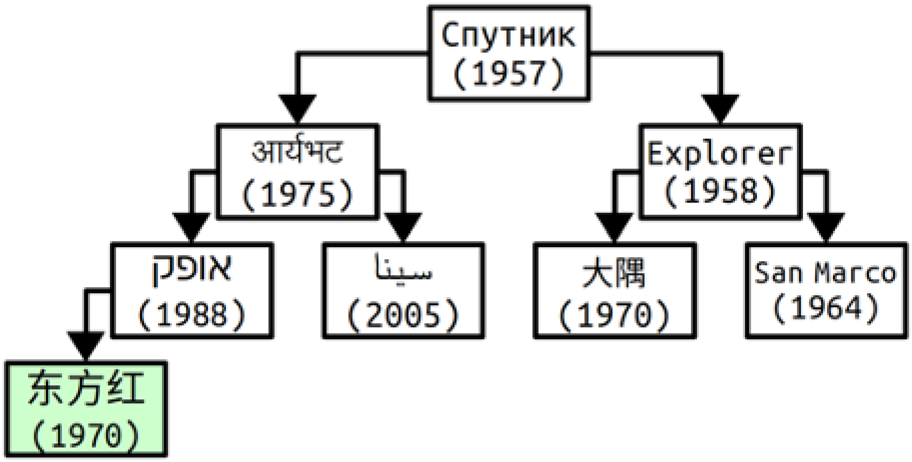
Let's suppose that we now want to insert Explorer, the first US space probe, into the heap with weight 1958. We begin by placing it at the next free location in the last row, as the right child of San Marco:



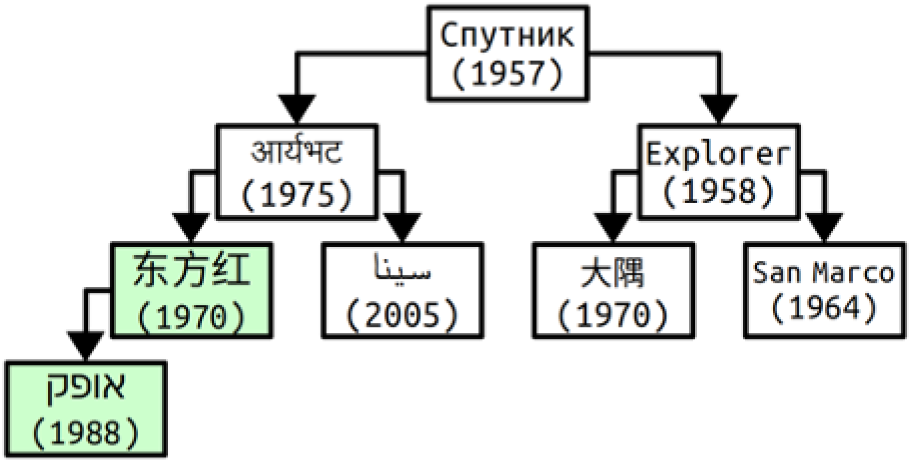
We then bubble Explorer up one level to fix the heap:



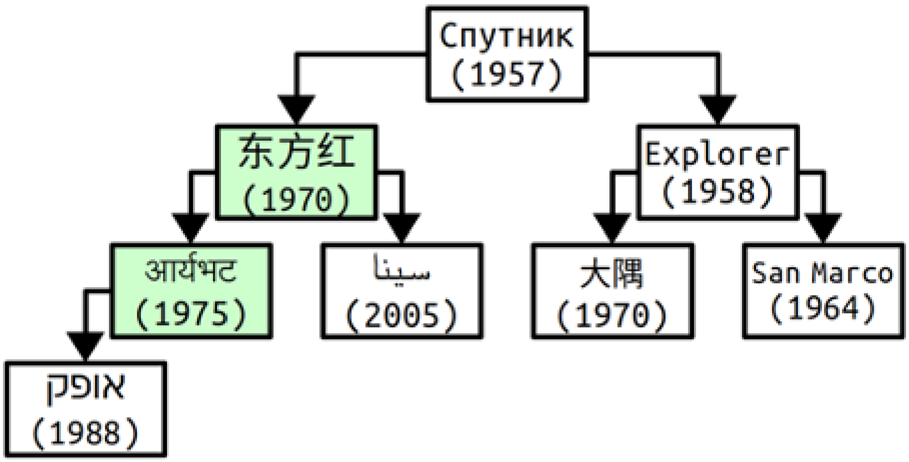
And, again, have a new heap containing these elements. As a final example, suppose that we want to insert 东方红, the first Chinese space probe, into this heap with weight 1970. We begin by putting it into the first free spot in the last row, which in this case is as the left child of the Israeli satellite אופק:



We now do a bubble-up step. We first swap 东方红 and אופק to get:



Notice that 东方红's weight is still less than its new parent's weight, so it's not yet in the right place. We therefore do another swap, this time with the Indian satellite आर्यभट, to get



This step runs very quickly. With a bit of math we can show that if there are n nodes in a binary heap, then the height of the heap is at most O(log n), and so we need at most O(log n) swaps to put the new element into its proper place. Thus the enqueue step runs in time O(log n). That's pretty fast! Remember that the base-2 logarithm of a million is about twenty, so even with a million elements we'd only need about twenty swaps to place things!

Before moving on, answer the following question in `ShortAnswers.txt`:

Concept Check: Enqueue

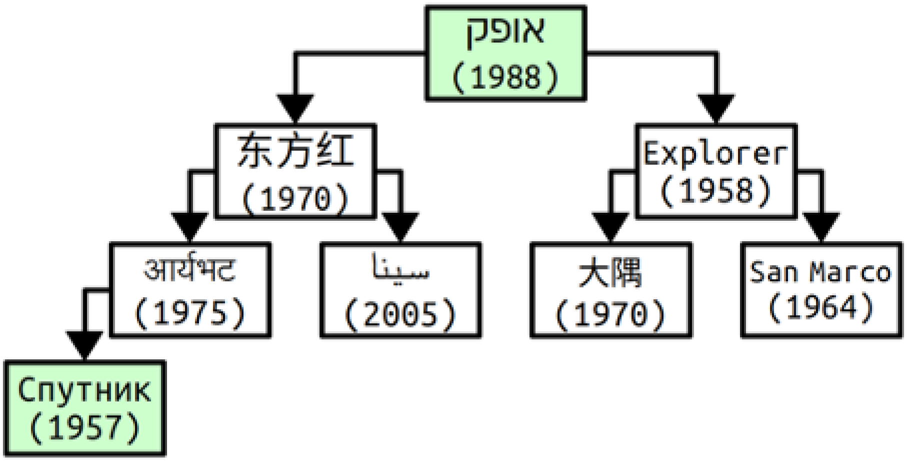
Q4: Draw the binary heap formed by inserting these nine DataPoints into an empty binary heap using the algorithm described above. Specifically, insert those data points in the order that's shown below. You only need to show your final answer, not your intermediate steps.

```
{ "R", 4 }
{ "A", 5 }
{ "B", 3 }
{ "K", 7 }
{ "G", 2 }
{ "V", 9 }
{ "T", 1 }
{ "O", 8 }
{ "S", 6 }
```

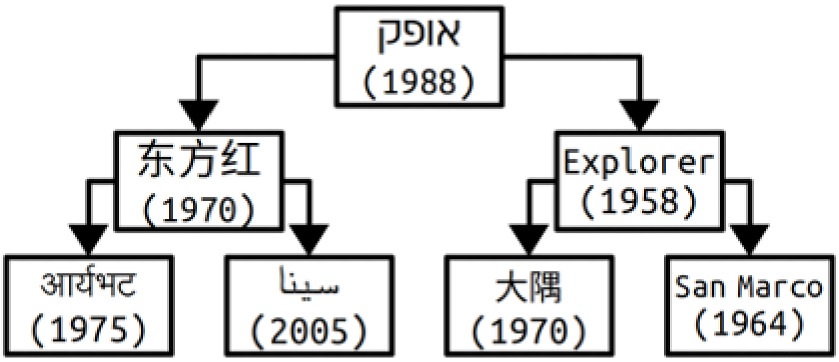
There are many binary heaps you can form that contain these elements, but only one of them will be what you get by tracing the algorithm.

Dequeuing from a Binary Heap

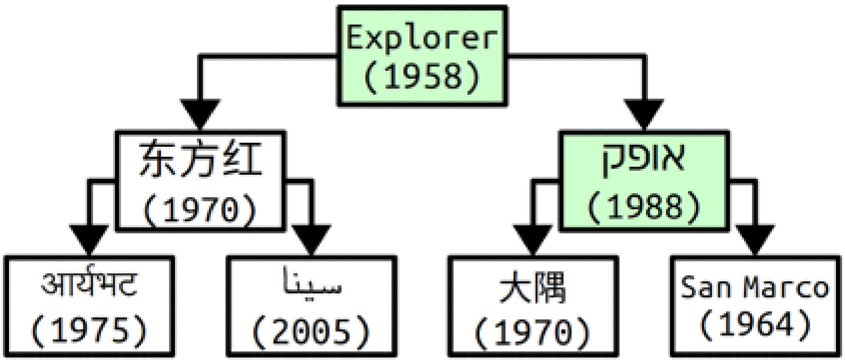
We now know how to insert an element into a binary heap. How do we implement dequeue? We know that the minimum-weight element of the binary heap is atop the heap, but we can't just remove it – that would break the heap into two smaller heaps. Instead, we use a more clever algorithm. First, we swap the top of the heap, the original Soviet satellite Спутник, with the rightmost node in the bottom row of the heap (אופק) as shown here:



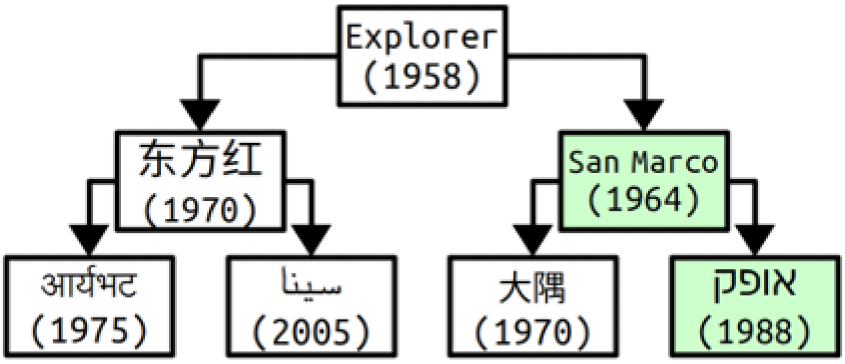
Now, we can remove Спутник from the heap, which is the element we'll return. We now have this:



Unfortunately, what we are left with isn't a binary heap because the top element (אופק) is one of the highest-weight values in the heap. To fix this, we will use a ***bubble-down*** step and repeatedly swap אופק with its ***lower-weight*** child until it either has no children or has lower weight than all of its children. First, we swap אופק with Explorer to get this heap:



Since אופק is not at rest yet, we swap it with the smaller of its two children (San Marco) to get this:



And we're done. That was fast! As with enqueue, this step runs in time O(log n), because we make at most O(log n) swaps. This means that enqueuing n elements into a binary heap and then dequeuing them takes time at most O(n log n). This method of sorting values is called *heapsort*.

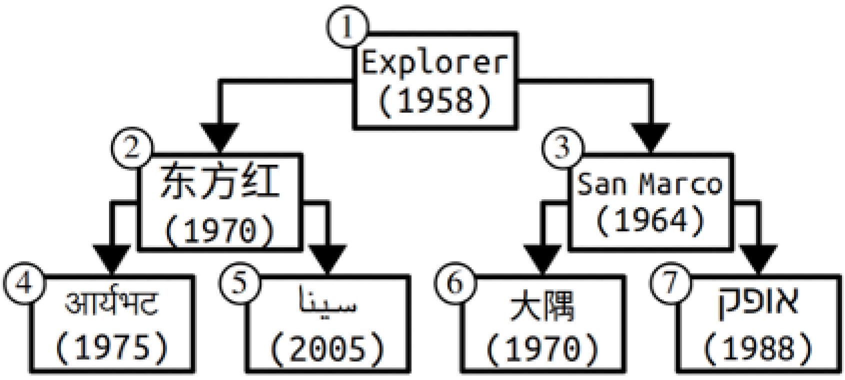
To confirm that this makes sense, answer this question in `ShortAnswers.txt`:

Concept Check: Dequeue

Q5: Draw the binary heap that results from following this **dequeue** procedure on the heap you drew in Q4 of this problem.

Binary Heaps As Arrays

How do we represent a binary heap in code? Amazingly, we can implement the binary heap using nothing more than a dynamic array. “An array?,” you might exclaim. “How is it possible to store that complicated heap structure inside an array?” The key idea is to number the nodes in the heap from top-to-bottom, left-to-right, starting at 1. For example, we might number the nodes of the previous heap like this:

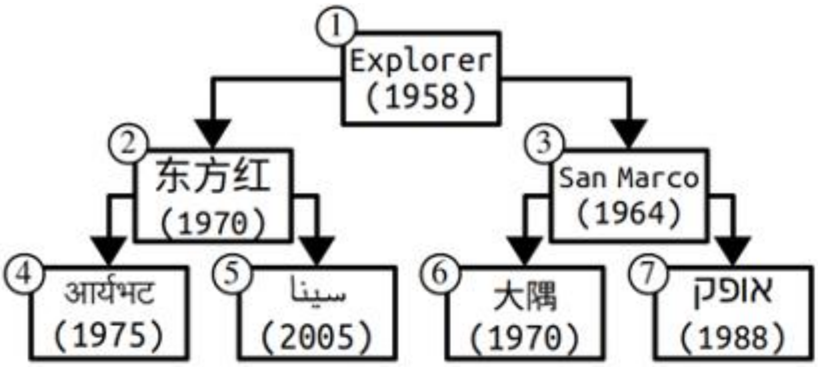


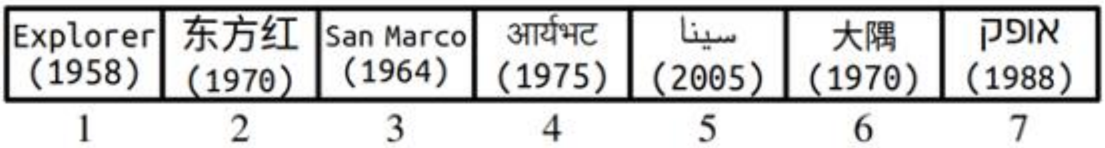
This numbering system has some amazing properties:

- Given a node numbered n, its children (if any) are numbered 2n and 2n + 1.
- Given a node numbered n, its parent is numbered n / 2, rounded down.

You can check this yourself in the above tree. That's pretty cool, isn't it? The reason that this works is that the heap has a rigid shape – every row must be filled in completely before we start adding any new rows. Without this restriction, our numbering system wouldn't work.

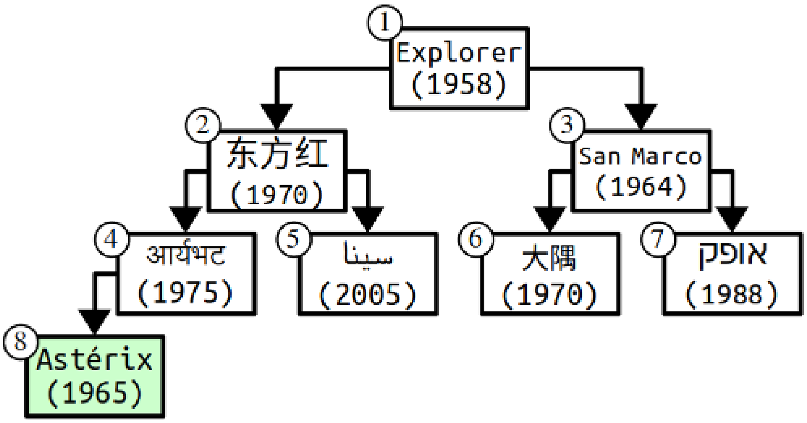
Because our algorithms on binary heaps only require us to navigate from parent to child or child to parent, it's possible to represent binary heap using just an array. Each element will be stored at the index given by the above numbering system. Given an element, we can then do simple arithmetic to determine the indices of its parent or its children. For example, we'd encode the above heap as the following array:





Explorer (1958)	东方红 (1970)	San Marco (1964)	आर्यभट (1975)	سينا (2005)	大隅 (1970)	אופק (1988)
1	2	3	4	5	6	7

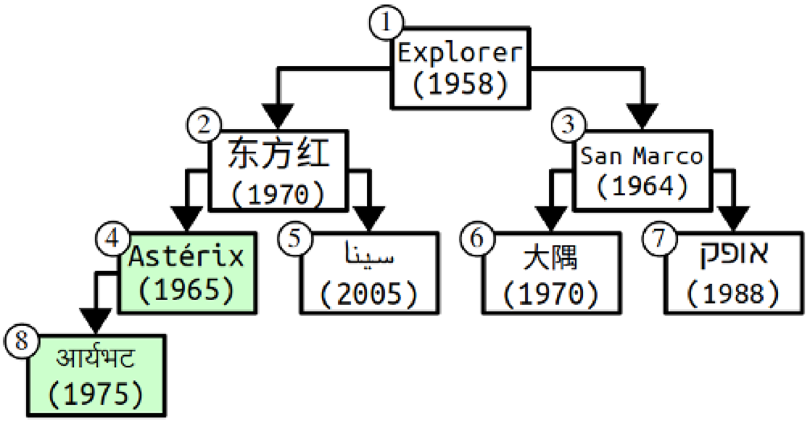
The enqueue and dequeue algorithms we have developed for binary heaps translate beautifully into algorithms on the array representation. For example, suppose we want to insert Astérix, the first French satellite, into this binary heap with weight 1965. We begin by adding it into the heap, as shown here:



Explorer (1958)	东方红 (1970)	San Marco (1964)	आर्यभट (1975)	سينا (2005)	大隅 (1970)	אופק (1988)	Astérix (1965)
1	2	3	4	5	6	7	8

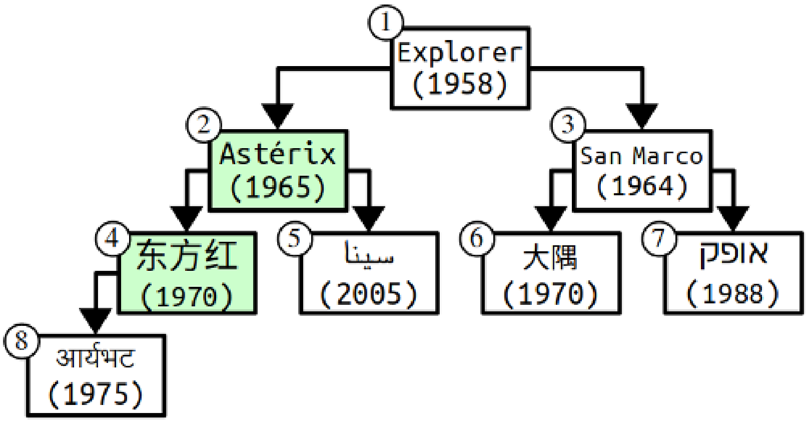
Notice that Astérix is at index 8, which is the last position in the array. This is not a coincidence; whenever you add a node to a binary heap, it always goes at the end of the array. (Do you see why?)

We then bubble Astérix up into its final position by repeatedly comparing it to its parent. Since Astérix is at position 8, its parent (आर्यभट) is at position 4. Since Astérix precedes आर्यभट, we swap them:



Explorer (1958)	东方红 (1970)	San Marco (1964)	Astérix (1965)	سينا (2005)	大隅 (1970)	אופק (1988)	आर्यभट (1975)
1	2	3	4	5	6	7	8

Astérix's parent is now at position 2 (东方红), so we swap Astérix and 东方红 to get the final heap:



Explorer (1958)	Astérix (1965)	San Marco (1964)	东方红 (1970)	سينا (2005)	大隅 (1970)	אופק (1988)	आर्यभट (1975)
1	2	3	4	5	6	7	8

An important detail to keep in mind when using this representation is the following. In these examples, ***the arrays are 1-indexed***, meaning that the first slot in the array is in position 1. However, in C++, ***arrays are 0-indexed***, which means that the first slot in the array is at position 0. There are many ways we could address this, but the option we're going to use in this assignment is the following. We'll use C++ arrays as usual, but simply ignore slot 0 in the array. So the array shown above, as encoded as a C++ array, would look like this:

(unused)	Explorer (1958)	Astérix (1965)	San Marco (1960)	东方红 (1970)	سينا (2005)	大隅 (1970)	אופק (1988)	आर्यभट्ट (1975)
0	1	2	3	4	5	6	7	8

And, more generally, each of the arrays shown above would have an unused slot zero before them.

To confirm that this all makes sense, answer the following question in ShortAnswers.txt:

Concept Check: Heaps as Arrays

Q6: Draw the array representation of the binary heap that you drew in Q5. Include an unused slot 0 at the beginning of your array.

Implementing a Binary Heap

Now, let's talk about the coding assignment. Open the HeapPQueue.h header file. You'll see (more or less) the following code:

```
class HeapPQueue {
```

```
public:
    HeapPQueue();
    ~HeapPQueue();

    void enqueue(const DataPoint& data);
    DataPoint dequeue();
    DataPoint peek() const;

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();

private:
    DataPoint* elems = nullptr;
    int logicalSize = 0;
    int allocatedSize = 0;

    static const int INITIAL_SIZE = 6;
};
```

Let's explore what's in the private section here. You will be doing your own memory management in this assignment. To help with this, we've provided you with three data members for the class:

- **`elems`**: A pointer to a dynamically-allocated array of elements. It defaults to **`nullptr`**, a pointer value that means "I'm not pointing at anything."
- **`logicalSize`**: The number of elements currently stored in the priority queue.
- **`allocatedSize`**: The number of slots in the dynamically-allocated array.

Your task is to implement the seven key functions in the **HeapPQueue** interface: the constructor, destructor, **enqueue**, **dequeue**, **peek**, **`isEmpty`**, and **size**. (You can optionally implement **`printDebugInfo`** if you'd like to print out information helpful for debugging.) In doing so, you'll do your own memory management to make sure there's sufficient space for elements in the array pointed at by **`elems`**, and you'll implement the enqueue ("bubble up") and dequeue ("bubble down") operations as described above.

While you can in principle implement these functions in whatever order you'd like, we recommend that you code things up in the following order.

Milestone One: Implement the Essentials

If you run the starter code without making changes to the **`HeapPQueue.h`** or **`HeapPQueue.cpp`** files and select "Run Tests," you'll find that the program almost immediately crashes. That's because the tests try to read from the priority queue's `elems` array, and that array is never set up to point at anything. Memory errors like these will often immediately crash the program, even in test mode. In fact, try doing this to see what it looks like! Run the program once without the debugger turned on to see what it looks like when your program crashes with a memory error. Then, do the following: without setting any breakpoints anywhere, run the program with the debugger turned on and click "Run Tests." Notice that as soon as the code crashes, the debugger kicks in and tells you the exact line where the error occured. Nifty! This is a useful tip going forward: ***if your code crashes, run it with the debugger turned on until the crash happens.*** No breakpoints needed - the debugger will engage and show you where things went wrong.

But of course, we'd like to make it so that things don't crash at all. 🙂 So your first milestone is to implementing the constructor to set up an initial array to work with. Your constructor should create an array whose size is given by the constant `INITIAL_SIZE`. Please use this exact size - we've chosen it so that our test cases will work correctly.

Once you've done this, you'll see that the tests no longer crash. Most of them fail because they check the behavior of functions like `enqueue` or `dequeue` that you haven't implemented yet. However, one of the early tests will fail and warn about a memory leak. hat's because your constructor is allocating memory, but that memory is never freed anywhere. To address this, implement the destructor for the `HeapPQueue` type.

Having done this, you should see the first few tests pass and everything else fail. And that's fine - you haven't actually implemented the basic `HeapPQueue` operations yet, and having the test fail is a marked step up from the test cases crashing!

And finally, once you've done that, implement the `size` and `isEmpty` functions.

To summarize, here's what you need to do.

1. *(Optional)* Run the starter files using the "Run" button without adding or changing anything. Click "Run Tests" to see what a crash looks like. Then, run the starter files again with the debugger engaged - without adding any breakpoints - and click "Run Tests" a second time. This will trigger the debugger at the crash site, and you can then backtrace to see where the crash was.
2. Implement the constructor for `HeapPQueue`.
3. Implement the destructor for `HeapPQueue`.
4. Implement the `size` and `isEmpty` functions on `HeapPQueue`.

5. Confirm that the first few tests, which just focus on these functions, are passing.

Milestone Two: Implement enqueue

It's now time to implement the enqueue function. There are two separate components to this function. First, you'll need to faithfully implement the "bubble down" logic as described above in the section above. Second, you'll need to manage the memory in your underlying array so that if you run out of space, you grow the underlying array.

Because these are two separate steps, we recommend that you do this in two separate steps. First, get the basic enqueue logic working, without making any effort to resize the array if you need more space. To prevent your code from crashing, we recommend that, at the top of your function, you add a check that detects whether you've run out of space for more elements and reports an `error` if so, like what we did with our initial bounded stack implementation. This will let you run the tests that purely look for the core logic in enqueue.

Once those tests are passing and you're confident that your basic enqueue logic works, add in the necessary logic to resize and grow the array. As in lecture, whenever you run out of space, double the size of the underlying array and copy elements from the old array to the new one.

To summarize:

1. Implement the enqueue function on HeapPQueue. Our recommendation is to first have this function report an `error` if there's no space left in the array so you can focus on getting the bubble-up logic right. Then remove the call to `error` and instead grow the array if you're out of space.
2. Confirm that tests for the enqueue function work correctly.

Some notes on this part:

- ***If your program mysteriously stops running without warning, it probably crashed.*** To see where it crashes, ***run the program in debug mode***. You don't need to set any breakpoints – simply running in debug mode will let the debugger catch where your program crashed. It'll stop the program at that point as if you'd set a breakpoint there, and you can then walk up and down the call stack, look at local variables, etc. to see what's going on. Common causes of program crashes this way include the following:
    - Writing to an array variable through an uninitialized pointer. If you don't initialize a pointer, it's not given a sensible default value. It might be a null pointer, and writing to a null pointer will cause a crash. It

  - might be a pointer to memory you don't own, which will crash your program. Or it might point to memory you do own, in which case you might see a crash or might not, depending on whether you were (un)lucky.
  - Writing to an array variable pointing at deallocated memory. Once you've used delete[] to deallocate an array, you should not try reading or writing that memory. (That would be like trying to let yourself into a building you just sold to someone else.)
  - Writing past the end of an array. You saw this in the warmup assignment.
  - Writing before the start of an array. This can happen if your indices are wrong.
- ***The indices in our array-based heap start at one.*** C++ arrays are zero-indexed but binary heaps are 1-indexed, so slot 0 in the `elems` array should never be read from or written to. Stated differently, your code should never touch the value in `elems[0]`. Many bugs in this part of the assignment arise from accidentally putting an element there or comparing the weight of one `DataPoint` against the weight of the (uninitialized) `DataPoint` in slot 0.
- ***Draw lots of pictures.*** Debugging the `HeapPQueue` is largely an exercise in determining where your code does something that diverges from what you expect to happen. If your code isn't working correctly, step through it using the debugger, drawing pictures of what you see and comparing them against pictures of what you think should happen.
- You saw how to use the debugger to view an array of `DataPoint`s in Part One of this assignment. That skill might come in handy here.
- You are welcome to define private helper functions in `HeapPQueue.h`. However, please do not change the signatures of any of the functions that we have provided to you, and do not change the names or types of any of the private data members we gave to you in the starter file.
- Bubbling up requires swapping elements in the array. We recommend using the `swap` function from the `<algorithm>` header for this. It takes in two arguments and swaps their values.
- If multiple data points are tied for the same weight, you can break those ties however you'd like.
- Our testing harness automatically checks for memory leaks and other types of memory errors. If you allocate memory without later deallocating it, the test driver will tell you how many objects you leaked. You will then need to explore your code to find the source of the memory error.
- The C++ standard libraries contain a function `std::push_heap` that implements the bubble-up algorithm. For the purposes of this assignment, please refrain from using those functions.

Milestone Three: Implement peek and dequeue

Implement the **peek** and **dequeue** functions. The **peek** function should return the element of the `HeapPQueue` with the lowest weight, and should call the `error` function to report an error if the priority queue is empty.

Your **dequeue** function should remove and return the `DataPoint` in the priority queue with the lowest weight, calling `error` if the priority queue is empty. You should use the "bubble-down" strategy outlined above. Because your binary heap elements are stored in an array, you'll need to do some arithmancy to figure out where an element's children are. This is a great spot to draw pictures of different binary heaps represented both as a tree and as an array. Once you've gotten this working, all the provided tests should pass!

To recap:

1. Implement the **peek** function on `HeapPQueue`.
2. Implement the **dequeue** function on `HeapPQueue`.
3. Confirm that all the provided tests pass.

Some notes on this problem:

- Much of the same advice from above applies here - draw lots of pictures, run the program in debug mode if it crashes or if a test fails, etc.
- As you're coding this one up, make sure not to read off past the end of the array. It's very easy to do this accidentally if you have a math error in your bounds-checking code or logic for determining where an element's children are. This can manifest in many different ways. You might see the code immediately crash, or you might just get back garbage values that don't correspond to array elements. Regardless of the source, if this happens, run the code with the debugger engaged to see if you can spot where the error occurs.
- Because C++ arrays have a fixed size that's set when the array is created, your array of elements will likely have more slots in it than there are elements in the priority queue. As a result, it's important to not try reading from array indices that don't correspond to elements in the binary heap. If you read array slots that don't have elements in them, it won't crash your program, and instead you'll see uninitialized `DataPoint` values. A common symptom that you've read these elements is that the tests will fail and report strange `DataPoint`s whose associated `weights` are bizarre values.
- Remember that, during the bubble-down step, an element may have 0 children, 1 child, or 2 children. You'll need to ensure your code works in each of those cases.

Milestone Four: Final Tests

Finally, validate that your implemention runs efficiently. If you insert *n* elements into an empty binary heap and then remove all *n* of them, the runtime should be O(*n* log *n*). Our provided "Time Test" menu option will you run this exact workflow on your `HeapPQueue` type and give you a runtime plot you can use to size up whether this workflow indeed runs as quickly as intended.

You'll also need to write at least one custom test for your `HeapPQueue`. There are lot of cases not covered by the existing tests. Adding in additional test support will help increase your confidence that your code works correctly.

To summarize:

1. Select the "Time Tests" option from the main menu and click the "`HeapPQueue`" button to get performance data on enqueuing *n* items into your `HeapPQueue`, then dequeuing them all.
2. Confirm the data support a runtime of O(*n* log *n*). If not, investigate your code and see if you can locate the source of the inefficiency.
3. Write at least one `STUDENT_TEST` in `HeapPQueue.cpp`. Before doing so, read over the existing tests to see what they do and don't cover, and aim to patch a hole in the test coverage.

Part Three: Apportionment

Background: The Huntington-Hill Method

(This part of the assignment requires you to have *HeapPQueue* ***working, so we recommend doing it last.)***

Every decade, the United States conducts a census to “count every person living in the United States and the five U.S. territories.” These numbers are then used to determine how many seats in the House of Representatives will be allocated to each state. That process – going from state populations to numbers of seats – is called ***apportionment***.

Apportioning seats fairly is trickier than it looks, and in fact the exact mechanism for doing this was hotly debated for much of US history. Eventually, in 1941, Congress settled on an algorithm called the ***Huntington-Hill method***, and that algorithm has been used ever since.

The Huntington-Hill algorithm works as follows. Initially, give each state one house seat. Then, drop all of the states into a `HeapPQueue`. The weight associated with each state is given by its population, divided by 2 (more on that later). Finally, repeat the following process to assign a single seat until all seats are allocated to the states.

- Remove the highest-weight state from the priority queue.
- Give that state another House seat.
- Reinsert that state into the priority queue with weight equal to PS(S+1)), where P is the state's population and S is the number of seats currently assigned to the state, including the one just awarded.

For example, let's suppose that we have five states: the State of Mind, the State of Denial, the State of Affairs, the State of Being, and the State of Matter. Their populations are as shown here. For consistency, I've listed these states in alphabetical order:

- Affairs: 7,984
- Being: 14,938
- Denial: 13,509
- Matter: 14,617
- Mind: 25,419

Further, let's suppose that there are 14 house seats to allocate. We begin by giving each state one seat, then adding each state into a HeapPQueue with priority equal to their population divided 2. Here are the values now stored in the priority queue. For simplicity, I've shown them in the same order they're listed above rather than in the order they'd be stored in in the priority queue. (These values are rounded to two decimal places.)

- Affairs: 5,645.54
- Being: 10,562.76
- Denial: 9,552.31
- Matter: 10,335.95
- Mind: 17,973.95

We now pick the state with the ***highest*** weight and assign it another seat. Here, that means that the State of Mind gets another seat. We compute its new weight using the formula from above. The State of Mind has a population of 25,419 and it currently has two seats assigned (the initial one from above, plus the one we just gave it), so we compute its new weight as 25,4192 · (2+1)≈10,377.26 and add that back to the priority queue. Here's the priorities now associated with the five states (again, in their original order, not sorted by weight):

- Affairs: 5,645.54

- Being: 10,562.76
- Denial: 9,552.31
- Matter: 10,335.95
- Mind: 10,377.26

The highest-weight state (State of Being) gets the next seat. We add it back to the priority queue, using the formula from above to compute its priority as 14,9382 · (2+1)≈6,098.41. Our ordering now looks like this:

- Affairs: 5,645.54
- Being: 6,098.41
- Denial: 9,552.31
- Matter: 10,335.95
- Mind: 10,377.26

Next up is the State of Mind, which now gets a third seat. We compute its new weight using the formula from above, giving it a new weight of 25,4193 · (3+1)≈7,337.83. That’s shown here:

- Affairs: 5,645.54
- Being: 6,098.41
- Denial: 9,552.31
- Matter: 10,335.95
- Mind: 7,337.83

If you carry on this process, the next seats will go to Matter, then Denial, then Mind, then Being, then Matter again, then Mind again. The final apportionment is then that

- Affairs gets 1 seat,
- Being gets 3 seats,
- Denial gets 2 seats,
- Matter gets 3 seats, and
- Mind gets 5 seats.

Milestone One: Implement Apportionment

Your task is to implement a function

```
Map<string, int> apportion(const Map<string, int>& populations, int
numSeats);
```

that takes as input a map from states to populations, along with a number of seats, then returns the apportionment given by the Huntington-Hill algorithm.

One of the key steps you'll need to figure out to solve this problem is the following. The `HeapPQueue` type is excellent at finding the entry with the *lowest* weight. However, for the Huntington-Hill algorithm, you need to find the entry with the *highest* weight. For the purposes of this problem, ***you must not make any changes to your HeapPQueue type***, and instead should just use the `HeapPQueue` as-is. See if you can find a clever way to "trick" the `HeapPQueue` into handing back the highest-weight item at each step rather than the lowest-weight item.

Here's what you need to do for this part of the assignment:

Coding Requirements

1. Implement the the `apportion` function in `Apportionment.cpp`. You will need to use the `HeapPQueue` type that you built in the previous part of this assignment in the course of doing this. However, you should ***not*** modify the files `HeapPQueue.h` or `HeapPQueue.cpp` to solve this problem (unless, of course, you're fixing bugs in the `HeapPQueue` type).
2. Add at least one custom test case – and, ideally, many more – and test your code thoroughly using the test suite.

Some notes on this problem:

- The Huntington-Hill method assumes that there are at least as many seats as there are states. If there are more states than seats, you should call `error()` to report an error.
- Remember that the `HeapPQueue`'s dequeue function removes and returns the *lowest-weight* item. However, the Huntington-Hill algorithm works by taking out the *highest-weight* item at each step. We'll leave it to you to determine how to handle this.
- Use the `sqrt` function from the header `<cmath>` to compute square roots. Remember that C++ doesn't have a `**` operator to compute powers, and that the `^` operator means something totally different than exponentiation.
- Don't worry about what happens if at some point in the process two states are tied with the same weight. If that happens, break ties arbitrarily. (In practice, you wouldn't actually do this, but it's a fair simplification for our model.)
- The Huntington-Hill algorithm, as we've outlined it, takes time O(*n* log *k*), where *n* is the number of seats and *k* is the number of states. (Great question to ponder: why?) Your implementation should meet this time bound. Assuming you more or less follow what we've outlined, your approach should take this much time.

Milestone Two: Explore and Evaluate

After the 2020 US Census, several news outlets reported the following claim: if 89 more people were living in New York, or 26 fewer people were living in Minnesota, then New York would have ended up with one more house seat and Minnesonta would end up with one fewer. We've provided you the 2020 US Census population totals that were used in the 2020 apportionment. You now have a tool you can use to explore how seats in the House of Representatives are allocated. So – are those claims true?

Use the provided demo to load the 2020 census numbers used in to determine apportionment (`2020.csv`), and see how many seats New York and Minnesota each received. Then compare that against the files `2020-NYUp89.csv` and `2020-MNDown26.csv`, which represent adding 89 people to New York or removing 26 from Minnesota, respectively, and see what happens. Then, answer the following questions in the file `ShortAnswers.txt`:

Q7: Is it true that 89 more people in New York would give New York another house seat?

Q8: Is it true that 26 fewer people in Minnesota would give New York another house seat?

Censuses, like elections, are a big deal. Please keep this in mind if you're living in the US in 2030!

(Optional) Part Four: Extensions

There are many, many ways you could do extensions on this assignment. Here are a few:

- ***Priority Queue***: There are so many different ways to implement priority queues, of which the binary heap is only one. Other heaps like the *Leonardo heap* and *poplar heap* store elements in a collection of small trees that in turn are encoded as a single array. Research one of these data structures and code up your own implementation.

  Using some clever algorithms, you can construct a binary heap from n items in time O(n) using an algorithm called ***heapify***. Research the heapify algorithm, then code it up.

- ***Apportionment***: You now have a tool you can use to explore the representational ramifications for various proposals that have been made over the years. Pick some proposal (for example, changing the number of seats in the House of Representatives, admitting Puerto Rico as a state, etc.) and evaluate what effect it would have on representation.

The Huntington-Hill method is one of many methods for apportionment. Historically, the US used several other methods, and in countries with party-list proportional representation a variety of similar methods are used to allocate seats. Research these methods, the tradeoffs involved, and the significance of the Balinski-Young Theorem, and report what you find. We recommend running some experiments of your own to quantitatively discuss their differences and tradeoffs.

We've looked at the Huntington-Hill method here because it's the one used in the US and we recently had a census. However, representative governments around the world face similar issues of determining how many seats to give to different representatives. Apply your algorithm to interesting data sets from aronud the world and let us know what you find!

Submission Instructions

Before you call it done, run through our submit checklist to be sure all your `t`s are crossed and `i`s are dotted. Make sure your code follows our style guide. Then upload your completed files to Paperless for grading.

Partner Submissions:

- If you forget to list your partner you can resubmit to add one
- Either person can list the other, and the submissions (both past and future) will be combined
- Partners are listed per-assignment
- You can't change/remove a partner on an individual submission

Please submit only the files you edited; for this assignment, these files will be:

- `ShortAnswers.txt`
- `HeapPQueue.cpp` and `HeapPQueue.h`. *(Don’t forget the header file!)*
- `Apportionment.cpp`.

You don't need to submit any of the other files in the project folder.

🏁 Submit to Paperless

If you modified any other files that you modified in the course of coding up your solutions, submit those as well. And that's it! You're done!

Good luck, and have fun!