

## Assignment 5: Data Sagas

---

*Parts of this assignment adapted from one by Julie Zelenski and Jerry Cain.  
A huge thanks to Michael Chang for his input on streaming top-k!*

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:

- **Child Mortality:** The United Nations Millennium Development Goals were a set of ambitious targets for improving health and welfare across the globe. Over twenty five years, the UN kept records of child mortality data worldwide. How did those numbers change since when they started keeping track in 1990 to when the most recent public numbers were released in 2013?
- **Earthquakes:** The US Geological Survey operates a global network of seismometers and publishes lists of earthquakes updated every hour. Where are these earthquakes? How big are they?
- **Women's 800m Freestyle:** The women's 800m freestyle swim race was introduced as a competitive event in the 1960s. How have the fastest times in that event improved since then? A certain Stanford-affiliated athlete might make an appearance here.
- **National Parks:** The US National Parks Service runs America's national parks, national monuments, national recreation areas, national seashores, etc. How many people visit those parks? Which ones are most popular? What trends can you detect by looking at those numbers?

These data explorations are layered on top of four algorithmic primitives:

- **Multiway merge.** We have sorted race results from several different 800m freestyle races, each of which is already in sorted order. We're interested in combining them all together into one giant sorted list of all times turned in over the decades. We could just combine everything together and sort it all from scratch, but there's a faster way to combining multiple sorted sequences.
- **Lower bound search.** If you want to see what national parks data looks like as a function of time, or to see how swim times have improved since the mid-1960s, you'll need some way of quickly finding the data points in a larger data set that fall within a certain range. We could do this with a giant linear scan, but there's a much faster way to do this.
- **Priority queue.** 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 as well.
- **Streaming top-k.** Let's say you want to figure out which seven countries have had the largest drop in their child mortality rates over time. You could do this by sorting all the countries by their drop in child mortality rates and taking the seven best, but that's overkill; you don't need *everything* in sorted order, just the top seven. There's a faster way to do this that, conveniently, layers on top of the priority queue.

This assignment is accordingly broken down into four smaller pieces. You have plenty of time to complete this assignment so that you can partition your time between studying for the exam and working to get everything finished. As usual, we recommend making slow and steady progress. It's much easier than trying to do everything in one sitting. The priority queue section of this assignment is probably the biggest, so you might want to budget a bit more time for that one.

***Due Wednesday, February 27<sup>th</sup> at the start of class.  
You are welcome to work on this assignment in pairs.***

## The Starter Files

If you run the starter files for this assignment without writing any code, you'll be presented with a screen with a bunch of options. Here's a quick rundown of what they do:

- **Run Tests:** Ah, that most familiar of buttons. This runs all the tests defined by the `ADD_TEST` macros across your files. This should be the first place you look when you're trying to see whether your code works. Initially, most of those tests will fail because you haven't implemented anything yet, but that will improve as you make progress on the assignment.
- **Time Tests:** This is the first assignment we've released where you'll be expected to implement algorithms and data structures meeting certain big-O runtime bounds. This tool will let you plot the efficiency of your solutions. Initially, the plots you get back won't mean much, since everything is unimplemented, but as you begin coding things up the numbers plotted will help you assess whether you've met the proper time bounds.
- **Interactive PQueue:** The third part of this assignment will ask you to implement a data type called a priority queue. Choosing this option will let you issue individual commands to your priority queue type, making it a bit easier to see what each operation is doing. Right now, all the member functions are stubbed out, so the values you see won't be correct.
- **Child Mortality:** Runs the child mortality demo. Without having written any code, you'll see an empty line graph, and none of the buttons will do anything. Once everything is working, you should be able to see trendlines showing how child mortality levels have changed over time.
- **Earthquakes:** Runs the earthquakes demo. Initially, you'll see a map of the world and some descriptive tests. Clicking the buttons will download live data sets from the US Geological Survey, but since you haven't implemented anything you won't see any data points plotted. Once you get everything put together, you'll be able to see where the five biggest earthquakes from each time period are.
- **Women's 800m Freestyle:** Runs a demo to explore women's race times. This demo will report an error when you choose it until you've implemented more of the parts of this assignment. Once everything is working, you'll see a slider bar you can sweep across years and watch the race times improve.
- **National Parks:** Runs a demo to explore national park and national recreation area attendance. If you haven't yet written any code, this demo will pop up a map of the United States, then report an error. As you start adding in more functionality, you'll eventually see a slider bar that lets you choose a year and see parks data from that year, along with the most popular parks from each year.

There are five files we expect you to modify in the course of this assignment:

- `MultiwayMerge.cpp`: You'll put your implementation of the `mergeSequences` function here, along with custom tests. You should not need to edit the header file.
- `LowerBound.cpp`: This is where your code for `lowerBoundOf` will go, plus your custom tests. Again, you should not edit the header file.
- `HeapPQueue.h`: This header file defines the `HeapPQueue` type, which you'll need to edit by adding in private data members and helper functions. Normally we ask that you not modify header files, but in this case it's necessary to do that to complete the assignment.
- `HeapPQueue.cpp`: You'll implement all the priority queue member functions in this `.cpp` file.
- `TopK.cpp`: You'll implement the `topK` function in this file. Please don't edit the header file.

You are welcome to peek around at the demo files if you'd like, though you're not expected to do so.

## Problem One: Multiway Merge

Consider the following scenarios:

- We have data from several swim meets, where for each meet we have the times sorted from fastest to slowest. We want to build a sorted list of all the finishing times across all those meets so that we can see the fastest times recorded.
- For each national park, we have records of the attendance at that park sorted by year. We want to aggregate this information into one giant timeline consisting of the yearly attendance of each park, in sorted order.

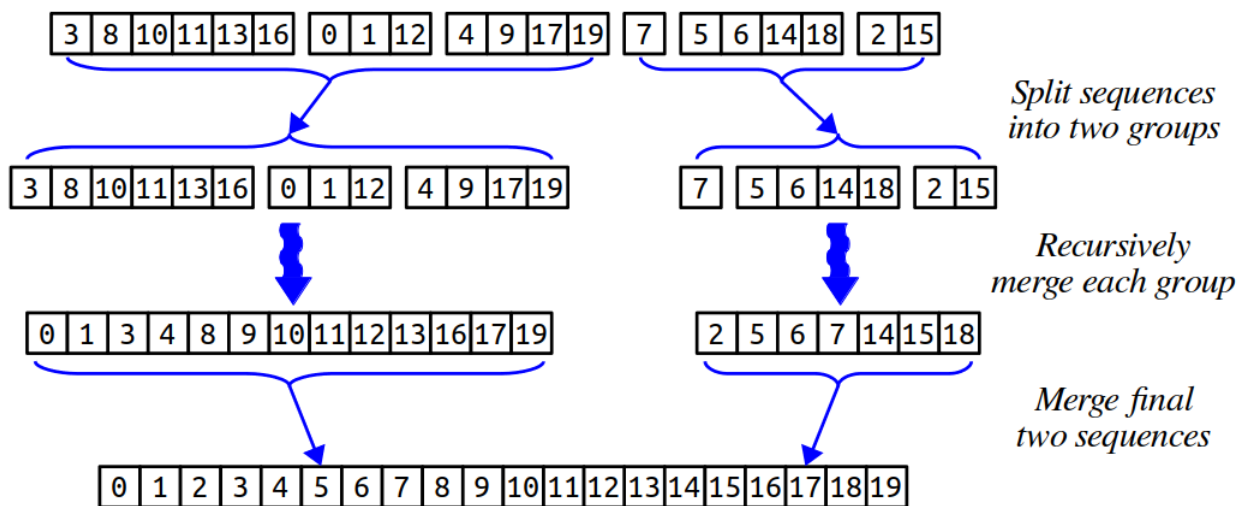
In some ways, these problems are similar to what we can solve with sorting algorithms. We're given a bunch of values, and we want to put them in order. What differentiates these two problems from regular sorting is that we already know that the elements are in roughly sorted order. We could ignore that extra information and just sort everything in time  $O(n \log n)$  using an existing sorting algorithm, but by taking this knowledge into account we can squeeze out some improvements to our runtime.

If you think back to the sorting algorithms we covered this quarter, you might remember mergesort. Mergesort works by recursively breaking the input array down into a bunch of tiny sequences, then using the merge algorithm to combine all those sequences back together into one giant, sorted sequence. In our particular setup, we already have the input broken down into smaller sorted sequences, and so we just need to do that second step of mergesort, merging things back together, to finish things off.

Let's imagine that we have  $k$  sequences that collectively have  $n$  total elements in them. We can follow the lead of mergesort to sort those sequences together:

- Split those  $k$  sequences apart into two groups of roughly  $k / 2$  sequences each.
- Recursively merge each of those groups of sequences, forming two large sorted sequences.
- Using the merge algorithm from class, merge those two sequences together.

Here's an example illustrating how to merge six sorted sequences ( $k = 6$ ) with twenty total elements across them ( $n = 20$ ) into one giant sorted sequence. For convenience, these are sequences of integers.



With a little bit of creativity you can prove that the runtime for this approach is  $O(n \log k)$ . In the case where you have a small number of sequences (low  $k$ ) with a large total number of elements (large  $n$ ), this can be dramatically faster than resorting things from scratch! For example, if  $n$  is roughly one million and  $k$  is, say, ten, then a multiway merge will be roughly ten times faster than a regular mergesort.

Your task is to implement a function

```
Vector<DataPoint> mergeSequences(const Vector<Vector<DataPoint>>& dataPoints)
```

that takes as input a list containing zero or more lists of data points, then uses the above algorithm to merge them into one giant sorted sequence.

The `DataPoint` type is a structure defined as follows, which you'll use throughout this assignment:

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

You can assume that the sequence of data points provided to you are sorted into nondecreasing order by their `weight` fields (that is, each data point's weight is at least as large as the preceding point's weight), and your resulting sequence should also be sorted by weight. Some notes on this problem:

- There may be multiple `DataPoints` that have the same weight. If that's the case, you should keep each of them in the resulting sequence, and you can break ties in weights arbitrarily.
- The sequences to merge aren't required to have the same size. Some of them may be gigantic. Some of them might be empty.
- It's perfectly legal to merge together a list of zero sequences. What do you think you should return in this case?
- The runtime cost of using the `Vector::subList` function is  $O(L)$ , where  $L$  is the length of the overall sublist formed. You are welcome to use this function if you'd like, but be careful not to use it in a way that degrades your overall runtime performance.
- The C++ standard libraries contain a function `std::merge` that merges two sequences. For the purposes of this assignment, please refrain from using that function. We're specifically interested in seeing you code this one up yourself.

**You must write at least one custom test case** for this part of the assignment, and we highly recommend including even more. The demos we've bundled with this starter files harness multiple parts of the assignment and are not designed to facilitate testing. The `ADD_TEST` macro is the best way to confirm that your code works as intended.

The testing framework we've provided you this quarter is great for checking whether the code you've written works correctly. Now that we've started talking about efficiency, you'll also need to make sure that your code has the proper big-O runtime. As a refresher, the code you write here should run in time  $O(n \log k)$ , where  $n$  is the total number of elements across all the lists and  $k$  is the number of lists.

To help you confirm that you have indeed met this runtime bound, we've bundled a runtime plotter along with the starter files. You can select it using the "Time Tests" button at the top of the demo app and then choosing the "Multiway Merge" button. You'll then see a graph of the runtime of your `mergeSequences` over a range of different values of  $n$  and  $k$ . The coordinate axes are on a standard linear scale, and the values of  $k$  that are shown go up by a factor of four on each run.

Take a look at the runtime plots you're getting back. Are they consistent with your function running in time  $O(n \log k)$ ? To answer that question, think through the following:

- What type of curve should you expect to see for a fixed value of  $k$ ?
- The values of  $k$  we've provided go up exponentially. Based on how  $\log k$  grows, how should the curves for each run relate to one another?

If you have questions about this, you're welcome to stop by the CLaIR (the Conceptual LaIR, which runs in parallel with the regular LaIR queue) to talk through these questions with one of the section leaders. Once you have a sense of what you think you should see, confirm that your runtime plots match what's expected. If so, great! If not, take a look back at your code. Think about where the inefficiencies might be coming from.

A note on the runtime plots – we've deliberately left off the labels on the y-axis because the *absolute* runtimes of your code (that is, what you'd see if you ran it with a stopwatch) is less important from a big-O perspective than the *relative* runtimes of your code (that is, how those runtimes scale). We don't have any target wall-clock runtimes in mind for this assignment, since that would depend on a bunch of factors like what computer you're using and whether it's plugged in.

## Problem Two: Lower Bound Searches

Consider the following problems:

- You have a list of all the race times recorded for a bunch of international swim meets, sorted by year. You want to find all of the swim times that occurred before the year 1980 so that you can examine the trends you see. How do you find just the data points in that range?
- You have a list of all the attendances of US National Parks sorted by year. That is, all the data points from 1904 come first, then the entries for 1905, then 1906, etc. You want to find all the data points that occur purely within some year. How do you isolate just those elements?

We could solve both of these problems by doing a giant linear scan across all of our data points, copying the elements we find that match our criteria. That would take time  $O(n)$ , which is fine for small  $n$  but is going to take a while if  $n$  is in the millions, especially if we want to make lots of queries of this sort.

There's a certain problem that, fortunately, can be solved in time  $O(\log n)$ , and that's the following:

Given a list of data points sorted by year, along with a specific year to search for, what is the index of the first data point that occurs at or after that year?

This is called a *lower bound search*. For example, in the sequence of data points given below, a lower bound search for 1915 would return index 3, a lower bound search for 1900 would return index 0, and a lower bound search for 1969 would return index 7.

Becquerel	M. Curie	P. Curie	W. Bragg	L. Bragg	Einstein	Fermi	Strickland
1903	1903	1903	1915	1915	1921	1938	2018
0	1	2	3	4	5	6	7

As an edge case, doing a lower bound search for 2137 would return index 8, the length of the list of data points, to indicate “this element isn't here.”

Once we can do fast lower bound searches, we can solve both of the problems at the top of this section. In the swimming case, we can do a lower bound search for the year 1980, giving us the index of the first data point at or after 1980. We can then iterate over the data points prior to this, which come from before 1980. For example:

```
Vector<DataPoint> raceTimes = /* ... */
int stopIndex = lowerBoundOf(raceTimes, 1980);
for (int i = 0; i < stopIndex; i++) {
    /* ... do something with raceTimes[i] ... */
}
```

In the second case, if we, say, want to look up all the data for the year 1956, we could do two lower bound searches: one for 1956, giving us the index of the first data point to look at, and one for 1957, giving us the index of the first data point that occurs after 1956. That might look like this:

```
Vector<DataPoint> parkAttendance = /* ... */
int stopIndex = lowerBoundOf(parkAttendance, 1957);
for (int i = lowerBoundOf(parkAttendance, 1956); i < stopIndex; i++) {
    /* ... do something with parkAttendance[i] ... */
}
```

Assuming we can quickly compute lower bound searches, the above code will spend almost all of its time processing data points that we actually care about.

Your task is to write a function

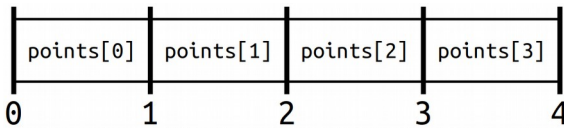
```
int lowerBoundOf(const Vector<DataPoint>& points, int key);
```

that takes as input a list of data points (sorted in increasing order by weight) and an integer key, then returns the index of the first data point within that list whose weight is greater than or equal to key. As an edge case, if key is bigger than the weights of all the elements in the Vector, this function should return as its “index” the length of the Vector, which represents “it's past the end.”

This function needs to run efficiently – specifically, in time  $O(\log n)$ . To meet that time bound, you’ll need to get creative. Start off by looking at binary search. Make sure you understand how binary search works and what the intuition is behind the algorithm. Also, make sure you can explain why it runs in time  $O(\log n)$ .

The binary search algorithm we presented in class simply returns whether the a specified key exists in an array. That’s not quite what you need here; do you see why?

Some notes on this problem:

- As a hint, instead of thinking of indices as counting *elements* in the array, think of indices as counting *separations between elements* in the array, as shown to the left. This perspective is extremely useful!
- 
- **Draw lots and lots of pictures.** You don’t need to write much code here, but that code will have to be fairly precise. It’s easy to make off-by-one errors here, so if you see your implementation giving back the wrong answer, step through your code in the debugger, drawing pictures to see what it’s doing and seeing if you can zero in on the wrong answer.
  - You are welcome to implement this function either iteratively or recursively, depending on whatever you think is going to be easier.
  - Remember that the runtime of  $O(\log n)$  means that you cannot make linear scans over the array. That is, if at any point in your solution you find yourself writing a for loop that iterates over parts of the input list, there is a very good chance that your algorithm runs in time  $O(n)$  rather than  $O(\log n)$ . Some tricky cases to consider: what happens if all the elements in the array are the same? If they’re all different? Will your code run in time  $O(\log n)$  in each case?
  - The C++ standard libraries contain a function `std::lower_bound` that performs a lower bound search. For the purposes of this assignment, please do refrain from using that function. We’re specifically interested in seeing you code this one up yourself.

**You must implement at least one test case** for this part of the assignment, and ideally more than this. Remember, the demos use all of the parts of this assignment in concert with one another and are not designed for testing. Go to town with `ADD_TEST`. Think about the sorts of edge cases that might occur.

As before, use our runtime plotter to see how long your code takes to run on different inputs. What should you expect to see when timing your function? Does that match what you see?

Once you’ve finished the first two parts of this assignment, much of the National Parks demo will be operational. You’ll be able to see the attendance at the parks rise and fall as a function of time. Pretty cool!

## Problem Three: Priority Queues and Binary Heaps

Now that we've started discussing class implementation techniques, it's time for you to implement your own collection class: the *priority queue*. A priority queue is a modified version of a 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. Similarly, if you were building a self-driving car that needed to process messages from multiple sensors, you might use a priority queue to respond to extremely important messages (say, that a pedestrian has just walked in front of the car) before less important messages (say, that a car two lanes over has just switched on its turn signal).

In this section of this assignment, you'll implement a priority queue. This will serve as a building block for the next part, where it will power some of the shiniest parts of the demo assignments.

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:
    /* Up to you! */
};
```

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 this priority queue type and a regular Queue is the order in which the elements that are added in get dequeued. In a regular Queue, the elements are lined up in sequential order, and calling dequeue() or peek() will give back the element that was added the longest time ago. In this HeapPQueue, the element that's returned by dequeue() or peek() is the element that has the *lowest weight* of all the remaining elements. 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 });
```

The elements were enqueued in this order. If we write

```
auto dataPoint = hpq.dequeue();
cout << dataPoint.name << endl;
```

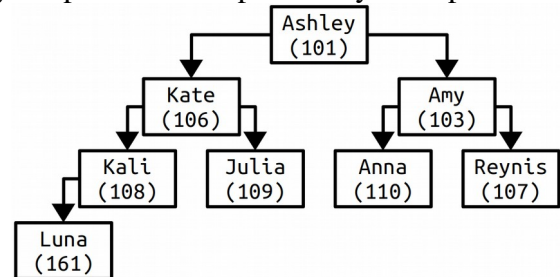
then the string printed out will be Ashley, since of the four elements enqueued (all former TAs for our CS courses, by the way!) the weight associated with her (101) was the lowest. Calling hpq.dequeue() again will return { "Amy", 103 }, since of the remaining TAs she has the lowest remaining priority. Calling hpq.dequeue() a third time would return { "Anna", 110 }.

We can then insert some more values. If we now call hpq.enqueue({ "Chioma", 103 }) and then hpq.dequeue(), the return value would be the newly-added { "Chioma", 103 } because her associated number is lower than all the remaining ones. 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!

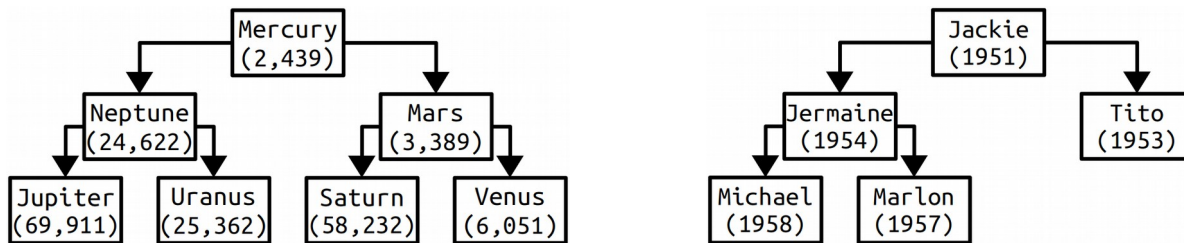
How should we implement this type? There are many approaches we could take that aren't very efficient. For example, we could store all the `DataPoints` in a `Vector` in the order in which they were enqueued, and then each time we dequeue an element we would scan all the entries in the `Vector` to find the lowest-weight one. That would make inserts fast ( $O(1)$ , the cost of appending to a `Vector`), but would make dequeues slow ( $O(n)$ , the cost of scanning all the elements in the `Vector`). With a small value of  $n$  this is fine, but for large values of  $n$  (say,  $n = 1,000,000$ ) this is going to be too slow to be practical.

Instead of implementing things this way, we'd like you to implement this type using a data structure called a **binary heap**, hence the name `HeapQueue`. Binary heaps are best explained by example. To the right 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:

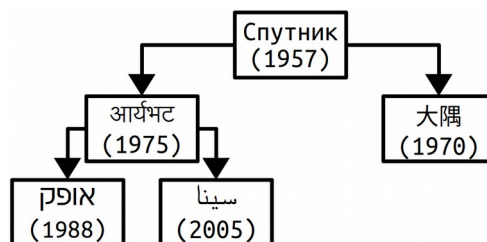


Inside a binary heap, we also enforce one more property – **no child node's weight is less than its parent's weight**. All three of the heaps you've seen so far obey this rule. However, there are no guarantees about how nodes can be ordered within a row; as you can see from the examples, within a row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

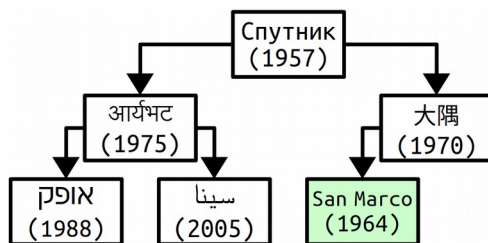
If the elements of a priority queue are stored in a binary heap, it is easy to read off which element is the smallest – it's the one at the top of the heap.

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:

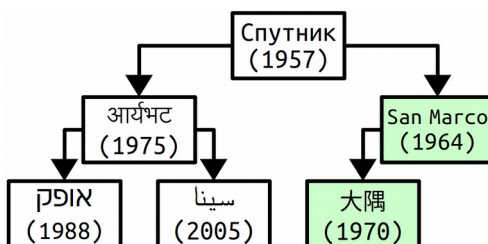




Let's add the San Marco, the first Italian satellite, to this heap with weight 1964. Since a binary heap has all rows except the last filled, the only place we can initially place San Marco is in the first available spot in the last row. This is as the left child of Japan's first space probe 大隅, so we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

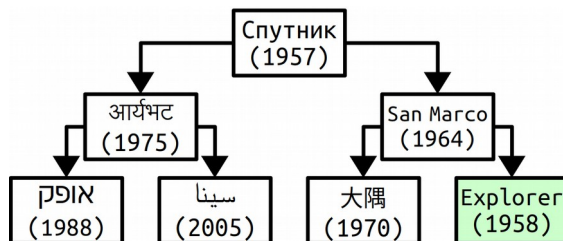


At this point, the binary heap is invalid because San Marco's weight (1964) is less than that of 大隅 (1970). To fix this, we run a *bubble-up* step and continuously swap San Marco with its parent node until its weight is at least that of its parent's. This means that we exchange San Marco and 大隅, shown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

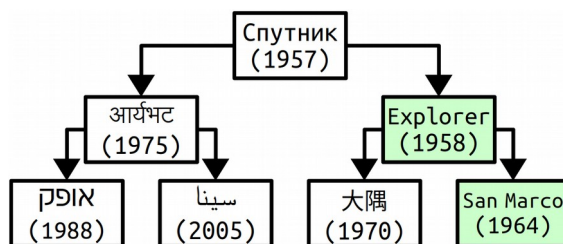


Since San Marco's weight (1964) is greater than its parent's weight (1957), it's now in the right place, and we're done. We now have a binary heap containing all of the original values, plus San Marco.

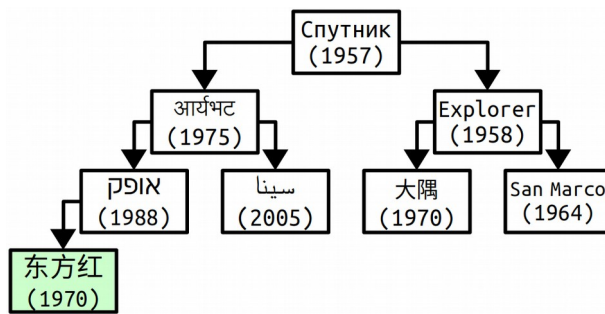
Let's suppose that we now want to insert Explorer, the first US space probe, into the heap with weight 1958. We begin by placing it at the next free location in the last row, as the right child of San Marco:



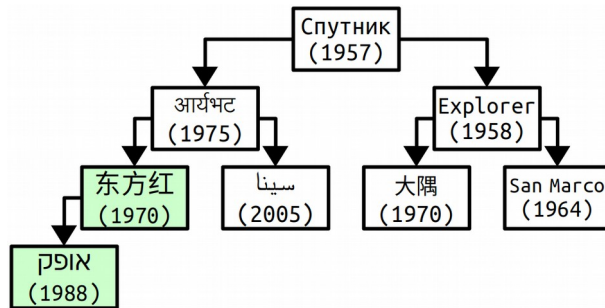
We then bubble Explorer up one level to fix the heap:



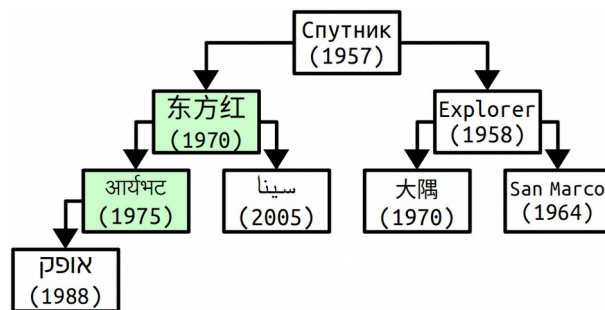
And, again, have a new heap containing these elements. As a final example, suppose that we want to insert 东方红, the first Chinese space probe, into this heap with weight 1970. We begin by putting it into the first free spot in the last row, which in this case is as the left child of the Israeli satellite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



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

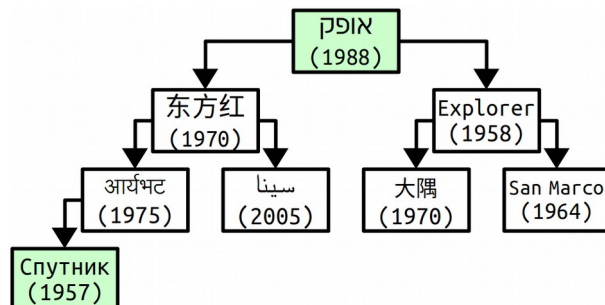


Notice that 东方红's weight is still less than its new parent's weight, so it's not yet in the right place. We therefore do another swap, this time with the Indian satellite आर्यभट, to get

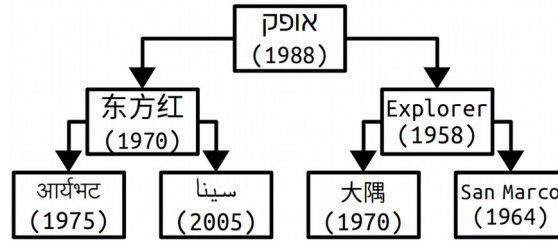


This step runs very quickly. With a bit of math we can show that if there are  $n$  nodes in a binary heap, then the height of the heap is at most  $O(\log n)$ , and so we need at most  $O(\log n)$  swaps to put the new element into its proper place. Thus the enqueue step runs in time  $O(\log n)$ . 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!

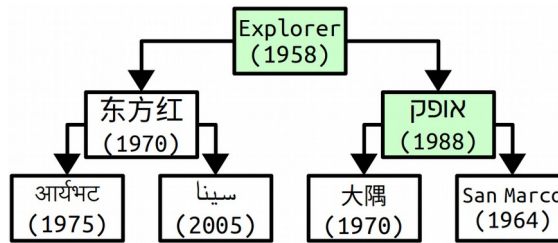
We now know how to insert an element into a binary heap. How do we implement dequeue? We know that the minimum-weight element of the binary heap is atop the heap, but we can't just remove it – that would break the heap into two smaller heaps. Instead, we use a more clever algorithm. First, we swap the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



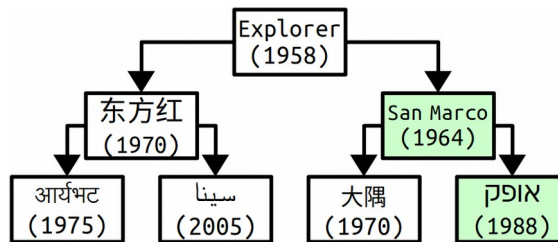
Now, we can remove *Спутник* from the heap, which is the element we'll return. We now have this:



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 comes to rest. First, we swap *אופק* with Explorer to get this heap:



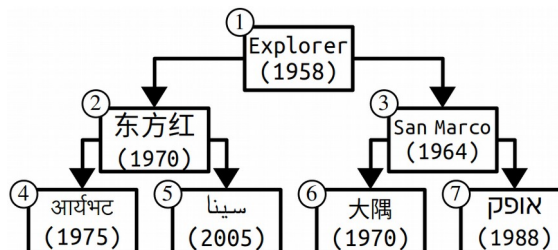
Since *אופק* is not at rest yet, we swap it with the smaller of its two children (San Marco) to get this:



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 enqueueing  $n$  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*.

How do we represent a binary heap in code? You might think that, like a linked list, we would implement the heap as cells linked together with pointers. This implementation, while possible, is difficult. Instead, we will 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. For example, we might number the nodes of the previous heap like this:

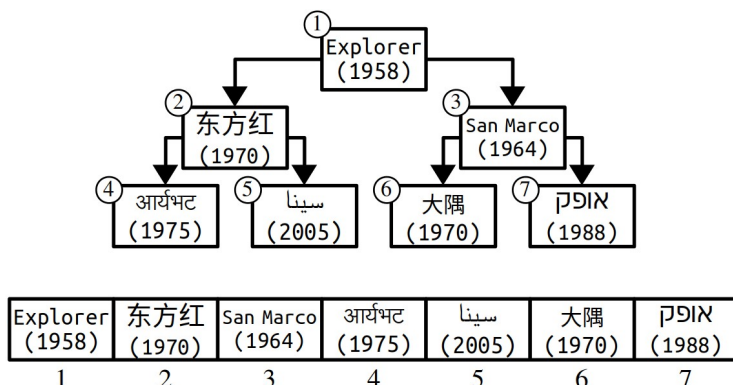


This numbering system has some amazing properties:

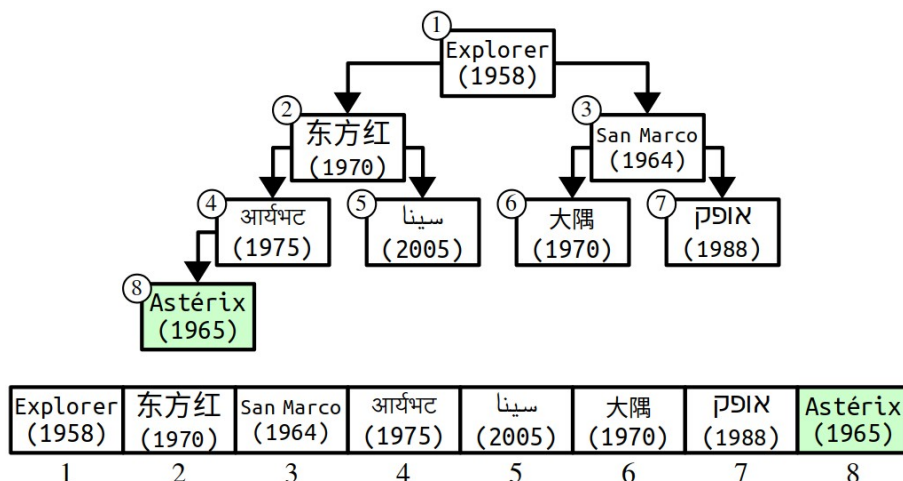
- Given a node numbered  $n$ , its children (if any) are numbered  $2n$  and  $2n + 1$ .
- Given a node numbered  $n$ , its parent is numbered  $n / 2$ , rounded down.

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:

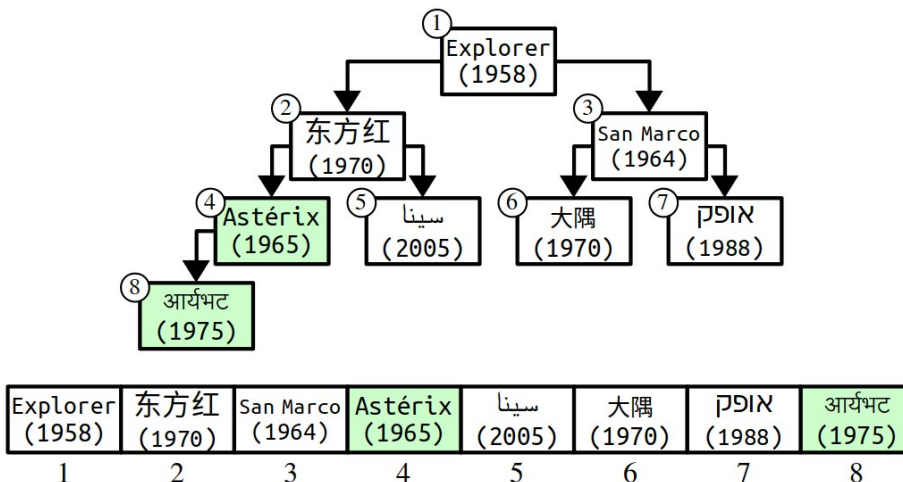


The enqueue and dequeue algorithms we have developed for binary heaps translate beautifully into algorithms on the array representation. For example, suppose we want to insert Astérix, the first French satellite, into this binary heap with weight 1965. We begin by adding it into the heap, as shown here:

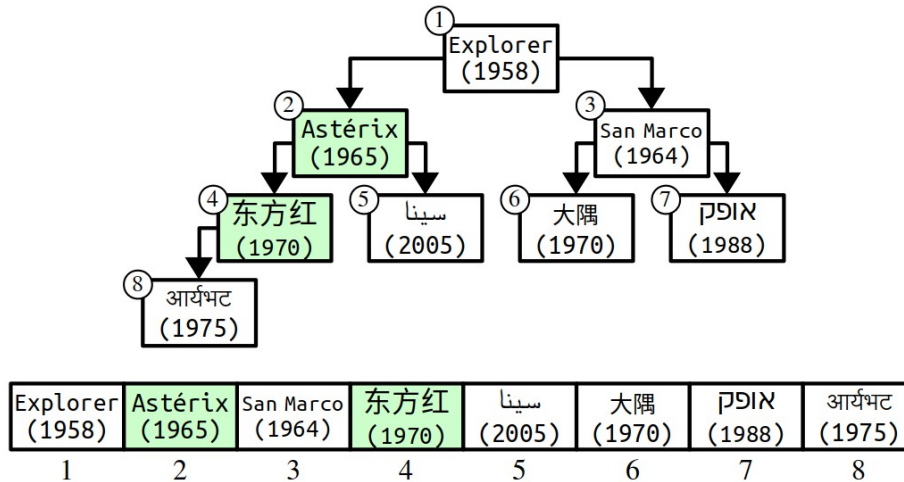


Notice that Astérix is at index 8, which is the last position in the array. This is not a coincidence; whenever you add a node to a binary heap, it always goes at the end of the array. (Do you see why?)

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:



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



Your task is to implement this data structure to power the `HeapPQueue` type. Although in practice you would layer this class on top of the `Vector`, for the purposes of this assignment **you must do all of your own memory management**. This means that you must dynamically allocate and deallocate the underlying array in which your heap is represented.

Some notes on this part of the assignment:

- Make sure you understand all the operations on binary heaps before you try coding them up. In particular, try working through the following exercise: what heap do you get if you insert the following data points, in this order, in a binary heap?

```
{ "Roberts", 4 }
{ "Alito", 5 }
{ "Breyer", 3 }
{ "Kagan", 7 }
{ "Ginsburg", 2 }
{ "Kavanaugh", 9 }
{ "Thomas", 1 }
{ "Gorsuch", 8 }
{ "Sotomayor", 6 }
```

Once you've done that, dequeue two elements out of the heap. What do you get back? What does the heap look like when you're done? If you were comfortable working through this exercise, great! You should be in good shape to move on. If not, stop by the CLaIR or ask your section leader for some input.

- Read over the comments in `HeapPQueue.h`, which detail what all of the member functions need to do. They're there for a reason. 😊
- You will need to do all your own memory management for this assignment. You've seen this before in lecture when we implemented the `Stack` type. How did we do that most efficiently? You might want to use a similar approach here.
- The indices in our array-based heap start at one. Remember that C++ arrays are zero-indexed, so you'll need to be clever about how you store things. There are many ways to do this – perhaps you will have a dummy element at the start of your array, or perhaps you'll adjust the math to use zero-indexing – but be sure that you keep this in mind when designing your implementation.
- If multiple data points are tied for the same weight, you can break those ties however you'd like.
- You are welcome to implement these functions either iteratively or recursively.
- The C++ standard libraries contain functions `std::push_heap` and `std::pop_heap` to enqueue or dequeue from a binary heap. For the purposes of this assignment, please refrain from using those functions.

To help you check whether your code is working correctly, we've provided an interactive environment where you can create and destroy priority queues and call different operations on them. To use it, choose the "Interactive PQueue" option from the top of the menu bar, then use the controls on the sides.

Testing is key in this assignment. This will be your first time testing a custom data structure, and in many ways the way to do so is similar to what you've done before. Write tests using the `ADD_TEST` macro, choosing workflows and edge cases strategically.

What's different about this particular assignment is that you have to implement an entire type rather than just a single function. You should be sure to write tests that look at different combinations of operations on the priority queue. For example, you might want to test that the dequeue operation works correctly in the cases where elements were enqueued in sorted order, reverse sorted order, or other types of orders that are crafted to make your binary heap look weird. You might also want to check that the size of your priority queue syncs up with what you expect it to be at different points in time.

As with the other parts of this assignment, ***you must write at least one custom test case*** for this part of the assignment, and ideally should put together many more than that. Remember – it is hard to verify your heap works purely by running the demo apps, since they tie together so many other components.

You also should make sure that the runtimes of the different operations match the time bounds specified in this handout and in the `HeapPQueue.h` header file. Our time tests will time what happens if we insert  $n$  elements into your priority queue and then dequeue them; this ends up sorting the inputs and is a slight modification of an algorithm called *heapsort*. Some things to think about:

- In terms of big-O notation, how much time does it take to insert  $n$  elements into an empty binary heap?
- In terms of big-O notation, how much time does it take to dequeue  $n$  elements from a binary heap that initially contains  $n$  elements?
- Based on that, how much time should it take to enqueue  $n$  elements and then dequeue them all?
- Based on that, what shape of curve should you expect to see when you run the time tests?

## Problem Four: Streaming Top- $k$

Consider the following problems:

- The US Geological Survey distributes data sets containing tens of thousands of recent earthquakes. You want to select just the five biggest earthquakes in that time range.
- The UN provides data about how child mortality rates have changed in the past twenty-five years. You want to find the seven countries with the lowest child mortality rates as of 2013.
- You've used your multiway merge algorithm to combine together all the different swim meets since the mid 1960s, and you've followed that up by using a lower-bound search to extract out all the race times up to 1990. You then want to find the ten fastest race times in that time range.
- After merging attendance data from national parks into one array and using lower bound searches to isolate attendance from 1956, you want to find the five most popular parks that year.

In each case, you have a collection of data points, and you want to find the  $k$  biggest or smallest data points in that year. This problem is called the *streaming top- $k$  problem*, and it's defined like this:

Given a stream of data points, find the  $k$  elements in that data stream with the highest weight.

Your task, specifically, is to implement a function

```
Vector<DataPoint> topK(istream& stream, int k);
```

that takes as input a data stream and a number  $k$ , then returns the  $k$  data points from that stream with the highest weight. If there are fewer than  $k$  elements in the data stream, you should return all the elements in that stream. The items should be returned in descending order of weight, so the zeroth element of the returned `Vector` should be the highest-weight data point, the first should be the data point with the highest weight less than that, etc.

You might have noticed that the input to this function is not provided by a `Vector<DataPoint>`, but rather by an `istream`, which is usually something you'd hook up to a file or to the console. Why is this?

Imagine, for example, that you're working at Google and have a file containing how many times each search query was made in a given day. You want to find the 1,000 most popular searches made that day. Google gets *billions* of search queries in a day, and most computers simply don't have the RAM to keep all those queries in memory at once. That would mean that you couldn't create a `Vector<DataPoint>` to hold all the data points from the file; it would take up more memory than your computer has available!<sup>1</sup>

When you read data from a file stream in C++, on the other hand, the full contents of that stream don't all have to be held in memory at the same time. Instead, whenever you try reading data from the stream, the stream calls out to your computer's hard drive to get a bit more information.<sup>2</sup> This means that you can process the elements from a data file one at a time without filling up your computer's memory; the space to hold each element from the file gets recycled each time you pull in a new element.

In the case of this problem, your goal is to implement your function such that you can find the top  $k$  elements using only  $O(k)$  space; that is, space proportional to the number of elements you'll need to return. In other words, if you wanted the top 1,000 search queries from Google, it wouldn't matter that there are billions or tens of billions of search queries per day. You'd only need to use enough space in RAM to hold about a thousand or so of them.

As a reminder, you can read one element at a time from an `istream` using this general pattern:

```
for (DataPoint pt; in >> pt; ) {  
    /* ... do something with pt ... */  
}
```

1 Technically, due to the wonder that is *virtual memory*, you actually could create this `Vector`. It would just degrade the performance on your machine so much that everything would grind to a seeming halt.

2 Technically, the stream usually uses a technique called *buffering* where, whenever you ask for data, it reads more data than you need from disk. Reading from disk is much, much slower than reading from memory, and this design helps reduce the amount of time spent asking the hard drive for data.

Your solution should not only use little space; it should also run quickly. Specifically, the runtime of your algorithm should be  $O(n \log k)$ , where  $n$  is the number of elements in the data stream.

Here's some hints to help you get started:

- Just to make sure you didn't miss it, we want you to return the *highest-weight* elements from the data stream sorted in *descending order*, which is the reverse of what we've asked you to do elsewhere in this assignment.
- Think about how you might solve this problem if you just wanted to find the highest-weight data point in the stream. How would you go about solving that problem? Do you see how you could do this without using much memory?
- There's a reason we asked you to implement `HeapPQueue` before tackling this problem.
- The runtime bound of  $O(n \log k)$  might actually give you a hint about how to solve this problem. There are  $n$  total elements to process, which means that you can only do  $O(\log k)$  work per element from the data stream.
- You can assume the data stream only has `DataPoints` in it and don't need to handle the case where the stream contains malformed data.

As with the other parts of this assignment, *you must write at least one custom test case* for this part of the assignment, and, ideally, should include more than just one. Repeating ourselves by repeating ourselves, trying to check whether your code works by running the demo apps is not going to be an effective use of your time. There's too many moving parts there and the data sets are too large to trace through in any principled way.

Also, be sure to run the time tests. You should have a sense of the general shapes of the curves you should expect to see back, since you already saw some curves from an  $O(n \log k)$ -time algorithm.

Once you've finished this section, all four of the bundled demos should be operational. Play around with them and see what you find! Did you learn anything? And remember – all the data processing that's being done there is using your code! We're just doing the graphics and UI.



## (Optional) Problem Five: Extensions

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

- **Multiway Merge:** If you gather numerical data from the real world, it typically has lots of increasing and decreasing runs within it. For example, if you measure peak temperatures each day of the year, you'll find that there's a lot of increasing runs as we move from spring to autumn, and there's a lot of decreasing runs as we move from autumn to spring. Many sorting algorithms are designed to work well in these sorts of cases. One, *natural mergesort*, works by splitting the input array into a sequence of already-increasing or already-decreasing runs, then repeatedly merging them together. Another, *Timsort*, is a more advanced algorithm that combines these sorts of techniques with a few others. Try implementing one of those algorithms and compare its performance against quicksort or mergesort on some real data sets. What do you find?
- **Lower Bound Searching:** Lower bound searches run in worst-case time  $O(\log n)$ , which is really fast. But in some cases, you can make it even faster! There's a search algorithm called *interpolation search* that works as follows. Rather than jumping right into the middle of the array, look at the two endpoints of the array. Then make a guess about where between those two endpoints your key is likely to be (this is the "interpolation" bit). For example, if the array endpoints are 0 and 100 and you're looking for the key 25, you'd guess that it's probably one-quarter of the way down the array. You can prove that this algorithm has an expected runtime of  $O(\log \log n)$  if the data are random values, which is exponentially faster than a regular binary search! Try implementing this approach and see how fast it is.
- **Priority Queue:** There are so many different ways to implement priority queues, of which one is the binary heap. Other approaches include binomial heaps, pairing heaps, leftist heaps, Fibonacci heaps, hollow heaps, thick heaps, and randomized meldable heaps. These other styles of heaps are designed to efficiently support additional operations on heaps, such as *meld*, which efficiently merges together two heaps, or *decrease-key*, which lowers the priority of an existing element in the heap, essentially bumping it up in line. Pick one of these other approaches, research it, and put together your own implementation.
- **Streaming Top-k:** The streaming top- $k$  algorithm is an example of a *streaming algorithm*, an algorithm that works on a data stream and tries to keep its memory footprint low. Streaming algorithms are an active area of research in algorithms and data structures, and there are some really beautiful ones out there. The *count-min sketch*, for example, lets you approximate how many times you've seen various elements in the data stream without actually storing everything you've encountered. Research a streaming algorithm and code up your own implementation.
- **Data Sagas:** This whole assignment is about exploring data sets with these sorts of algorithms to see what you find. So go out there and find a fun data set to explore! Get some data and tell us a good story about it. Which data set did you grab? Where did you find it? What did you find when you looked in that data set?

## Submission Instructions

Once you've finished writing up your solution, submit these files:

- MultiwayMerge.cpp
- LowerBound.cpp
- HeapPQueue.h and HeapPQueue.cpp. (*Don't forget the header file!*)
- TopK.cpp.

If you edited any of the other files, please be sure to include them as well. And remember – you need to write at least one custom test case for each of the four parts of the assignment.

*Good luck, and have fun!*