Assignment 5: Data Sagas

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

Due Friday, February 21st at 11:30AM.
You are welcome to work on this assignment in pairs.

This assignment has four parts:

- ***Combine***: You have a large collection of sorted sequences you want to assemble into a single sorted sequence. We could just toss everything together into a list and sort it from scratch, but there's a faster way to combine multiple sorted sequences.
- ***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.
- ***Streaming Top-k***: A clever algorithm that finds the *k* best objects of a given type, and does so remarkably quickly.

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

- Aim to complete the Combine algorithm by Thursday, February 13th. We'll have covered everything you need to know to solve this problem when this assignment goes out, but since we imagine you'll be studying for and then decompressing from the midterm, we figured there wasn't much rush in getting it started. 😀
- Aim to start the debugging exercise on Thursday, February 13th and complete it by Friday, February 14th. This debugging exercise references topics discussed in the lecture on Wednesday, February 12th, so starting it earlier than that might not be a good idea.
- Aim to start `HeapPQueue` on Friday, February 14th, with the goal of completing it on Wednesday, February 19th. This part of the assignment requires knowledge of topics covered on Friday, February 14th, so we don't recommend starting it before then.
- Aim to start Streaming Top-*k* on Wednesday, February 19th with the goal of complete Streaming Top-*k* by Thursday, February 20th. This part of the assignment builds on the `HeapPQueue`, so you probably shouldn't attempt it before then.

Problem One: Combine

Suppose you have several lists of numbers, each of which is already in sorted order. You want to combine those lists together into one giant list, also in sorted order. How should you do this?

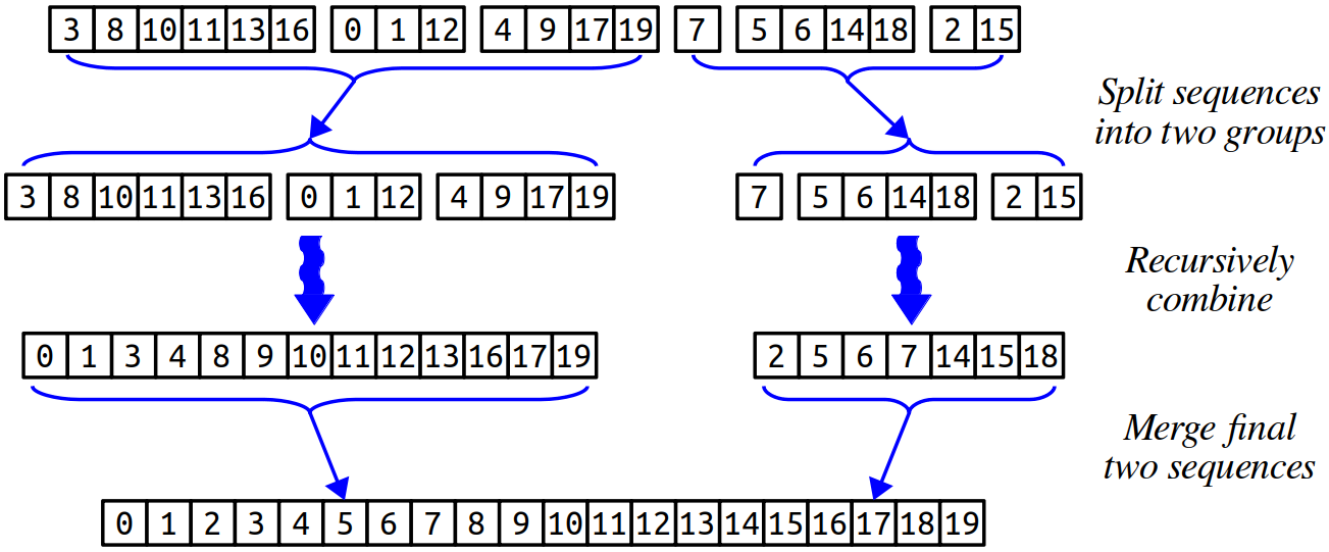
One approach would be to ignore the fact that we know these lists are already sorted and to do the following: create a giant `Vector` holding all the numbers, then sort it with mergesort in time O(n log n). This works, but by harnessing the fact that the input sequences are already sorted we can improve on this.

If you'll recall, 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. Here, 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 combine each of those groups of sequences, forming two large sorted sequences.
- Using the merge algorithm from class, merge those two large sequences together.

Here's an example illustrating how to combine 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 this combine algorithm will be roughly ten times faster than a regular mergesort.

Milestone 1: Implement Combine

Your task is to implement a function

```
Vector<DataPoint> combine(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 combine them into one giant sorted sequence. The `DataPoint` type is a defined in `Demos/DataPoint.h` 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 sequences of data points provided to you are sorted in 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.

Here's the first milestone you need to reach:

> ***Milestone 1:*** Get a working implementation of the `combine` function.
>
> 1. Add at least one custom test to `Combine.cpp` using `ADD_TEST`. We recommend doing this first, since it's a great way to confirm you understand what's being asked of you.
> 2. Implement `combine` from `Combine.cpp` using the following recursive strategy:
>   - Split the list of sequences into two groups with roughly the same number of sequences.
>   - Recursively `combine` each of those groups together, forming two sorted sequences.
>   - Use the merge algorithm to merge those resulting sequences into one overall sequence.
> 3. Test your code thoroughly to make sure that it works correctly.

Some notes on this problem:

- A key step in solving this problem will be implementing the merge algorithm. The version of merge that we outlined in class worked by repeatedly removing the first elements of the sequences to merge. ***Be careful*** – the `Vector::remove` function takes time O(n) if you remove the first element of a `Vector`, which is too slow for what we need. See if you can find a way to solve this problem without using `Vector::remove`.
- There may be multiple `DataPoint`s 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 combine aren't required to have the same size. Some of them may be gigantic. Some of them might be empty.
- It's legal to combine a list of zero sequences. What do you think you should return in this case?
- The C++ standard libraries contain a function `std::merge` that implements the merge algorithm from class. For the purposes of this assignment, *please refrain from using that function*. We're specifically interested in seeing you code this one up yourself.

Milestone 2: Ensure Combine Runs Quickly

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.

What, exactly, does O(n log k) mean? It might help to imagine that n is allowed to vary, while k stays constant. If you have a small value of k, then log k is also small, and as n varies you'll get a plot of a straight line with a small slope. As k increases, you'll see the slope of that line starting to increase, but not by much because log k grows *extremely* slowly. In particular, if you look at values of k that grow exponentially quickly (say, k = 1, 4, 16, 64, 256, 1024), the slope of the lines you see should appear to increase by some fixed rate.

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 "Combine" button. You'll then see a graph of the runtime of your `combine` function 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. So take a look a the runtime plots you're getting back. Are they consistent with your function running in time O(n log k)?

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.

Milestone 2: Ensure your code runs in time O(n log k), where n is the total number of elements and k is the number of different sequences.

1. Choose the "Time Tests" option from the top menu, then choose "Combine."
2. Run the time tests and look at the plots you get back. Is what you're seeing consistent with a runtime of O(n log k)? If so, great! If not, use those plots to form a hypothesis of what the runtime is, then go back to your code and see if you can spot the source of the inefficiency. Don't forget to run the regular tests whenever you make a change to the code; you need to both have efficient code and correct code.

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) are 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: Beyond the Bounds of Arrays

(This section assumes you're familiar with the topics from lecture on Wednesday, February 12th.)

The next 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.

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 `res/ShortAnswers.txt`.

Problem Three: Priority Queues and Binary Heaps

(This section assumes you're familiar with the topics from lecture on Friday, February 14th.)

Now that we're discussing class implementation techniques, it's time for you to implement your own collection class: the *priority queue*. 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).

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 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 });
```

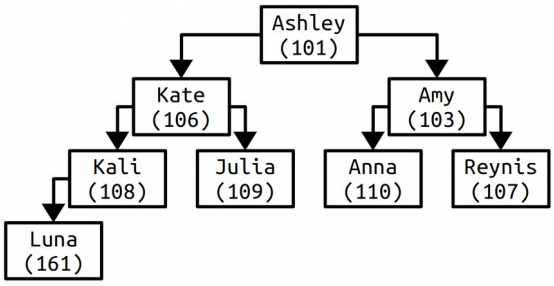
If we write

```
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}`.

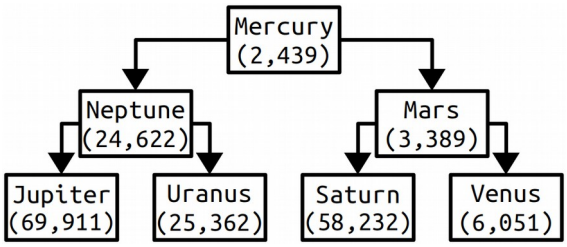
Let's insert more values. If we call `hpq.enqueue({"Chioma", 103})` and then `hpq.dequeue()`, the return value would be the newly-added `{"Chioma", 103}` because her associated weight is lower than all others. And note that Chioma was the most-recently-added TA here; just goes to show you that this is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

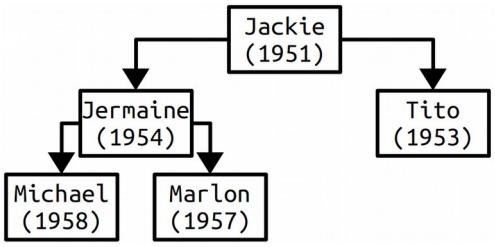
You'll implement `HeapPQueue` using a data structure called a ***binary heap***, hence the name `HeapPQueue`. Binary heaps are best explained by example. To the right is a binary heap containing a collection of current and former TAs, each of whom is associated with a number corresponding to the class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





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.

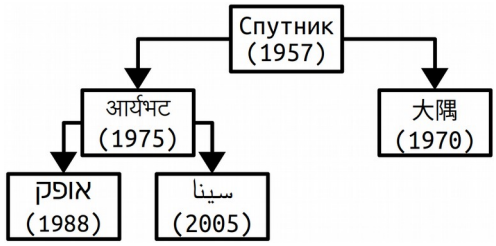
Before moving on, edit the file `res/ShortAnswers.txt` with answers to the following question.

Q3: Draw three different binary heaps containing the `DataPoint`s 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

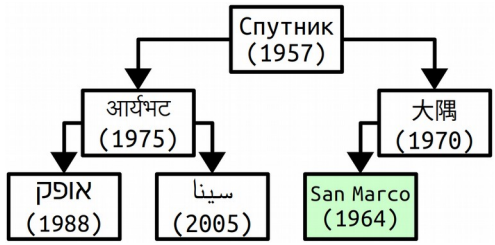
```
          Person A
           Weight
          /      \
   Person B    Person C
    Weight      Weight
   /      \
Person D  Person E
 Weight    Weight
```

Maya Ziv, one of the SLs this quarter, suggests using this website to draw the binary heaps.

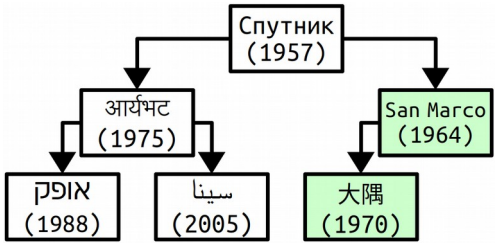
It is easy to read off which element has the smallest weight in a binary heap – it's the one at the top. It is also efficient to insert an element into a binary heap. Suppose, for example, that we have this binary heap containing some famous satellites:



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:

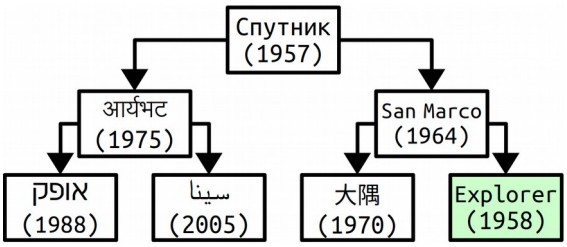


At this point, the binary heap is invalid because San Marco's weight (1964) is less than that of 大隅 (1970). To fix this, we run a ***bubble-up*** step and continuously swap San Marco with its parent node until its weight is at least that of its parent's. This means that we exchange San Marco and 大隅,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

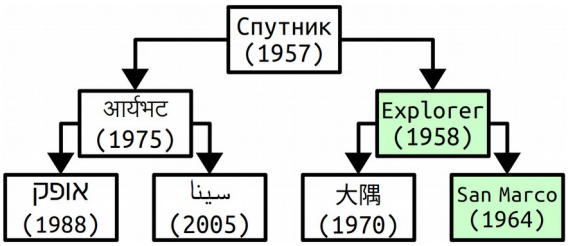


Since San Marco's weight (1964) is greater than its parent's weight (1957), it's now in the right place, and we're done. We now have a binary heap containing all of the original values, plus San Marco.

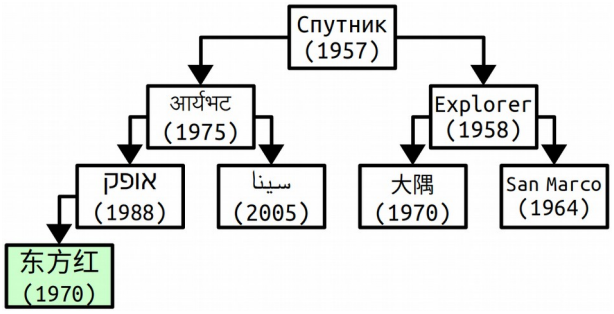
Let's suppose that we now want to insert Explorer, the first US space probe, into the heap with weight 1958. We begin by placing it at the next free location in the last row, as the right child of San Marco:



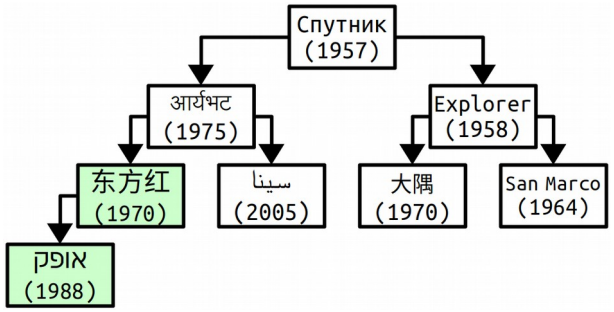
We then bubble Explorer up one level to fix the heap:



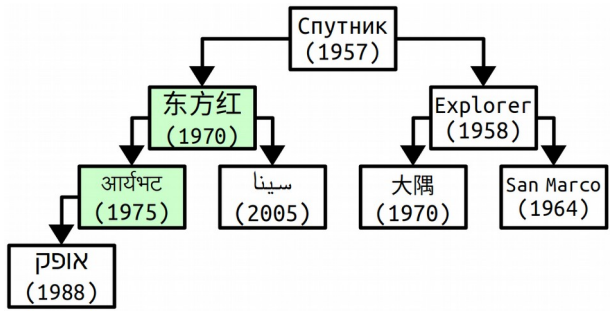
And, again, have a new heap containing these elements. As a final example, suppose that we want to insert 东方红, the first Chinese space probe, into this heap with weight 1970. We begin by putting it into the first free spot in the last row, which in this case is as the left child of the Israeli satellite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

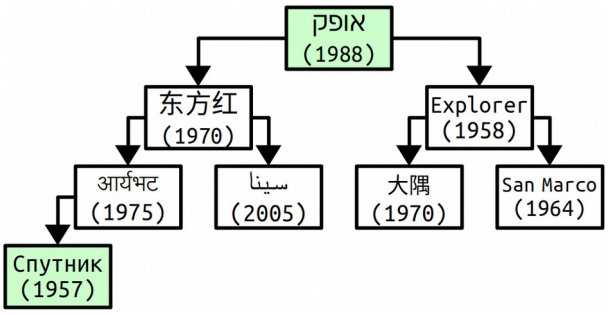


This step runs very quickly. With a bit of math we can show that if there are n nodes in a binary heap, then the height of the heap is at most O(log n), and so we need at most O(log n) swaps to put the new element into its proper place. Thus the enqueue step runs in time O(log n). That's pretty fast!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

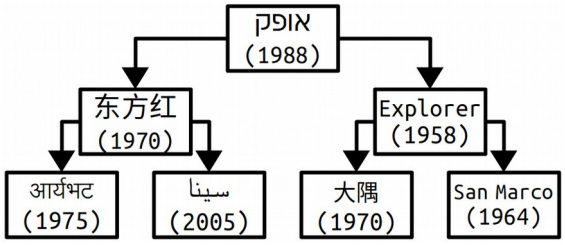
Before moving on, answer the following question in `res/ShortAnswers.txt`:

> Q4: Draw the binary heap formed by inserting these nine `DataPoint`s 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.

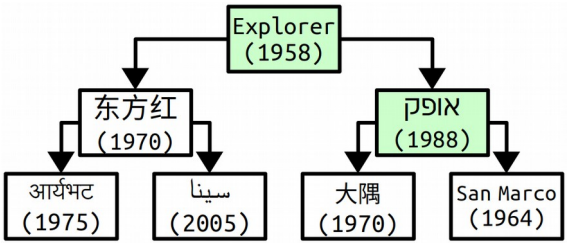
We now know how to insert an element into a binary heap. How do we implement dequeue? We know that the minimum-weight element of the binary heap is atop the heap, but we can't just remove it – that would break the heap into two smaller heaps. Instead, we use a more clever algorithm. First, we swap the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



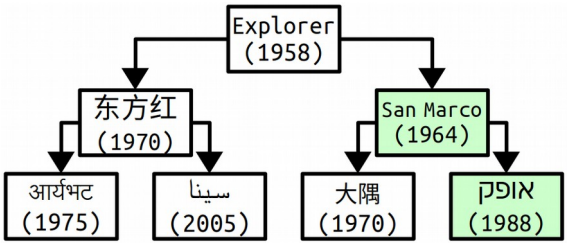
Now, we can remove Спутник from the heap, which is the element we'll return. We now have this:



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 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:

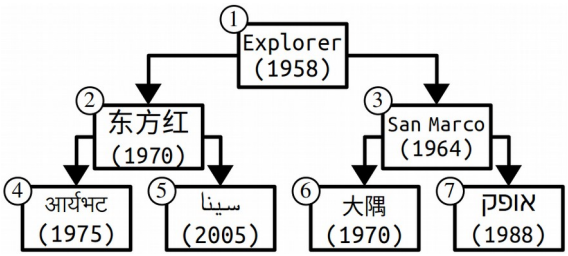


And we're done. That was fast! As with enqueue, this step runs in time O(log *n*), because we make at most O(log *n*) swaps. This means that enqueuing *n* elements into a binary heap and then dequeuing them takes time at most O(*n* log *n*). This method of sorting values is called *heapsort*.

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

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

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. For example, we might number the nodes of the previous heap like this:

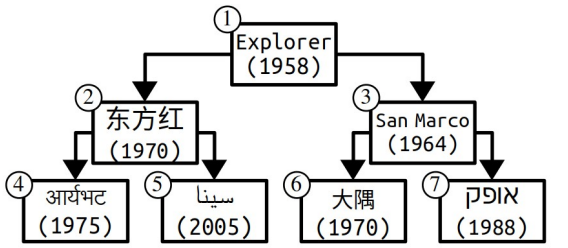


This numbering system has some amazing properties:

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

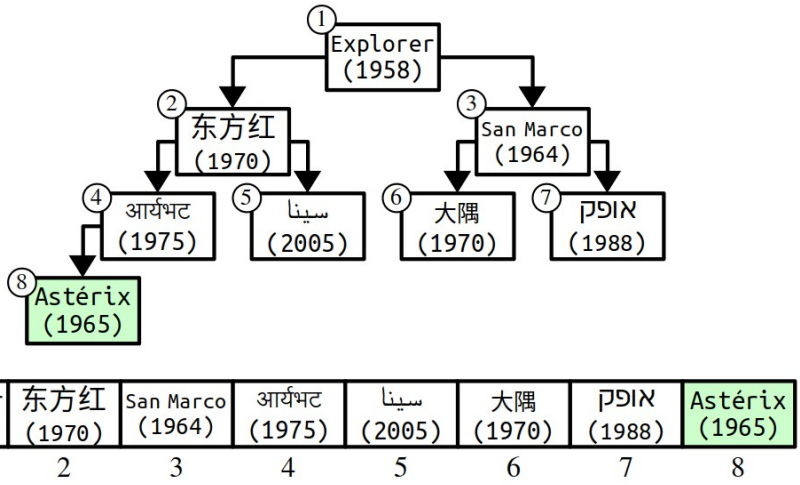
You can check this yourself in the above tree. That's pretty cool, isn't it? The reason that this works is that the heap has a rigid shape – every row must be filled in completely before we start adding any new rows. Without this restriction, our numbering system wouldn't work.

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:



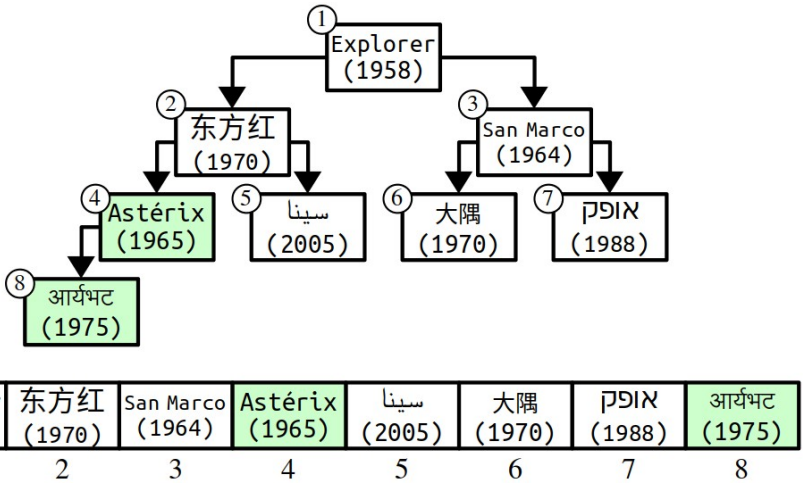
Explorer (1958)	东方红 (1970)	San Marco (1964)	आर्यभट (1975)	سينا (2005)	大隅 (1970)	אופק (1988)
1	2	3	4	5	6	7

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:

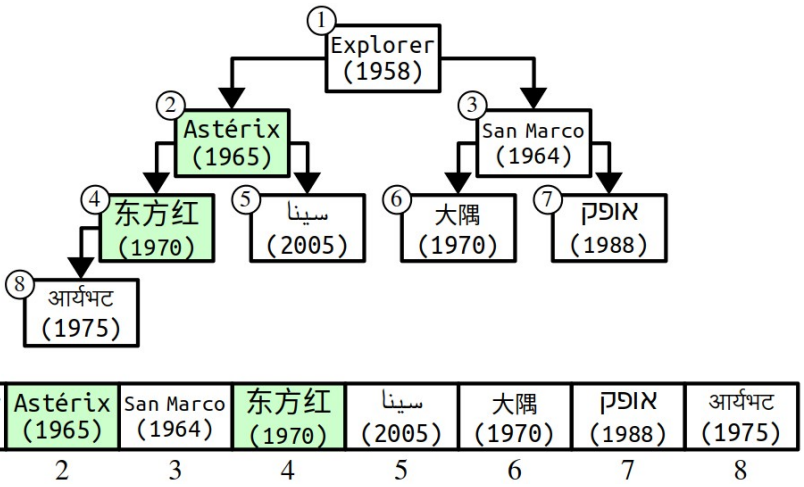


Notice that Astérix is at index 8, which is the last position in the array. This is not a coincidence; whenever you add a node to a binary heap, it always goes at the end of the array. (Do you see why?)

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:



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

> Q6: Draw the array representation of the binary heap that you drew in Q5.

Now, let's talk about the coding assignment.

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.

Here is our recommendation for how to complete this part of the assignment.

1. Start off by adding any private member variables you'll need to `HeapPQueue.h`. You know you'll at a bare minimum need to define something to hold the elements in the heap. We recommend you start there, since without adding member variables you won't be able to store anything.
2. Implement the constructor in a way that allocates a initial amount of space for the elements of the heap. We recommend picking a medium-sized array to begin with (say, around size 100), though you'll probably drop that to something lower later on. Then, implement `size` and `isEmpty`, which should each probably be a single line long.
3. To test whether your code works correctly, choose the "Interactive PQueue" option from the top menu bar. This will let you create and issue commands to a `HeapPQueue` using a graphical interface. Confirm that you can create a `HeapPQueue` and that `size` and `isEmpty` behave as expected (`size` should return `0`, `isEmpty` should return `true`).
4. Implement `enqueue` using the bubble-up algorithm. Then, implement `printDebugInfo` in a way that will let you confirm that your code works correctly. For now, don't worry about what happens if you run out of space in your dynamic array; you'll patch this up later. Head back to the Interactive PQueue option and test out your `enqueue` code by enqueuing a bunch of values, working out with a pencil and paper what the array contents should look like, and then using `printDebugInfo` to confirm things work.
5. Implement `dequeue` using the bubble-down algorithm. Run some initial tests using the Interactive PQueue feature and a pencil and paper, as before. Once that seems to be working, run the full automated test suite and see what happens. If you're passing all the tests, except for the tests that require your `HeapPQueue` to hold thousands of elements and the ones that check for memory leaks, move on to the next step. Otherwise, iterate and debug until things are working.
6. Change your implementation of `enqueue` to grow the dynamic array when more space is needed, and implement the `HeapPQueue` destructor. Change the original size of your array down to something much smaller (say, four or five elements) and use a mix of the automated test suites and the Interactive PQueue to confirm that your code works as intended. Move on to the next step once all the tests pass.
7. Use the Time Tests feature to confirm that the cost of doing n `enqueue`s followed by n `dequeue`s is indeed $O(n \log n)$. If not, look back over your code and see if you can spot the source of the inefficiency.
8. Add in at least one custom test of your own, just to make sure that you've covered all the cases you need to cover.

Some notes on this part of the assignment:

- ***The indices in our array-based heap start at one. C++ arrays are zero-indexed.*** 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.
- You are, of course, 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.
- If multiple data points are tied for the same weight, you can break those ties however you'd like.
- Read over the comments in `HeapPQueue.h`, which detail what all of the member functions need to do. They're there for a reason. 😀
- The C++ standard libraries contain functions `std::push_heap` and `std::pop_heap`. For the purposes of this assignment, please refrain from using those functions.

Problem Four: Streaming Top-*k*

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

In the previous part of this assignment, you implemented the `HeapPQueue` type. Your interactions with the `HeapPQueue` were focused on getting the thing working, not writing code that takes advantage of what the `HeapPQueue` has to offer.

In this last part of the assignment, you'll switch hats and become a *client* of your `HeapPQueue` type. Specifically, you're going to implement an algorithm that uses the `HeapPQueue` as a building block. This will be similar to what you did in Assignment 2, Assignment 3, and Assignment 4 when you interacted with types like `Vector` and `HashMap` – you used those types to solve problems without thinking too much about how those types were actually implemented.

The problem we'd like you to solve here 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 you're working at Google and have a file with 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![1]

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.[2] 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 … */
}
```

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.

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.

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

1. Implement the the `topK` function in `TopK.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.
3. Use the "Time Test" feature to see how fast your code runs, and confirm that the runtimes are consistent with what you'd expect to see in a piece of code whose runtime is O(n log k), where n is the number of elements and k is the input parameter to `topK`.

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 using a `HeapPQueue`? Now imagine you want to find the top two. How would you do that?
- 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 `DataPoint`s in it and don't need to handle the case where the stream contains malformed data.
- Inserting elements into a `Vector` using `Vector::insert` or removing elements from a `Vector` using `Vector::remove` will take time O(n) if you insert or remove elements from the beginning of the `Vector`. Be careful when using these functions, since they can degrade performance.
- Be careful – you can only consume the elements of an `istream` once. This means that, for each test you run, you can only call `topK` on a stream once. If you call `topK` a second time on that stream, the stream will appear empty and `topK` will return a different value than the one you got the first time you called `topK`.

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. 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, since you already saw some curves from an O(n log k)-time algorithm earlier in this assignment.

(Optional) Problem Five: Explore Some Data Sets!

The starter files we've provided you with in this assignment include four demos that use your code – your implementation of `combine`, the `HeapPQueue`, and your implementation of `topK` – to explore large data sets. Once you've finished the previous parts of this assignment, feel free to play around with these demos to see your code in action!

- ***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?

There are no deliverables for this part of the assignment. Just sit back, enjoy, and celebrate having gotten everything working!

(Optional) Problem Six: Extensions

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

- ***Combine:*** 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 heapsort or mergesort on some real data sets. What do you find?
- ***Priority Queue:*** There are so many different ways to implement priority queues, of which the binary heap is only one. 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 assignment builds up to 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:

- `Combine.cpp`
- `res/ShortAnswers.txt`
- `HeapPQueue.cpp` and `HeapPQueue.h`. ***(Don't forget the header file!)***
- `TopK.cpp`.

If you edited any of the other files in the course of adding extensions to the base assignment, please be sure to include them as well.

Good luck, and have fun!