off i flies to new hampshire

the state that screams live free or die here.

No more munchies, or zombie sleepies. I'm free to soar the skies my dear

Code Heaps

In this quest, you get to ride a jolly good heap, and then customize it into a special heap.

Jolly Good Heap

Weellll! *Good* always needs to come before *special*. That's what I always say.

So before you get started on special heaps, you're gonna implement a good ol' heap, which is, like, a jolly good heap.

A jolly good heap is a regular binary min heap, implemented over an array in a pretty much standard way. So let's get it over with first.

Figure 1 shows a fuzzy pic of the `Heap` template class.

```
/ Notes:
/  - T must implement operator<() via get_sentinel<T>() { return min possible
/  - The max size of a heap = capacity - 1 because elem 0 is a sentinel
/  - Capacity is simply elems.size() (= max_size + 1)
emplate <typename T>
lass Heap {
rotected:
   vector<T> _elems;
   size_t _size;
   static const int INIT_HEAP_CAPACITY = 128;

   // private helpers
   bool _percolate_down(size_t pos);
   bool _heapify();

ublic:
   Heap();
   Heap(const vector<T>& vec);

   virtual bool is_empty() const { return _size == 0; }
   virtual bool insert(const T &elem);
   virtual bool delete_min();
   virtual const T &peek_min() const;

   virtual string to_string() con

   friend class Tests; // Don't remove this line
```

Figure 1. The `Heap` Class

Specifics

Implement your binary min-heap by placing a sentinel at location 0 of your data array, `_elems`. This sentinel must be set in the constructor by issuing a call to a client supplied function `get_sentinel<T>()`. You will need to implement it to complete your own testing.

`get_sentinel<T>()` should simply return the lowest value an object of type `T` can have, which, in turn, is guaranteed not to occur elsewhere in the data. Write it and use it to your heart's content. But make sure your submitted code don't have it.

Once you have this sentinel in place, you can avoid a needless comparison of an index to 0 in your `insert()` method as you're bubbling elements into place. That is, traversing a chain of parents from a leaf towards the root. (Why?)

Here are some rewarding things to know about a jolly good heap:

_percolate_down

Let's say that an element that doesn't obey the heap ordering property in the data array is a problem element. Suppose that there is exactly one such element in the heap.

Given the index (in `_elems[]`) of this element, perform a series of moves in which the problem element is repeatedly swapped with its smallest child until it cannot be swapped any more. (Will the algorithm still work if you chose any child rather than the smallest child?)

Note that this can be implemented efficiently. You can avoid needless swaps by creating a *hole* and moving the hole to its final location before filling it with the problem element. In this location, the problem element will not be a problem element any more. (Why?)

_heapify

You call it when you want your heap's vector of elements to be reordered according to the binary heap property. Its running time is linear in the number of elements. (How?)

Done correctly, you will simply need to `percolate_down()` all elements from array index n/2 to 0, where n is the heap size. (Why is this enough?)

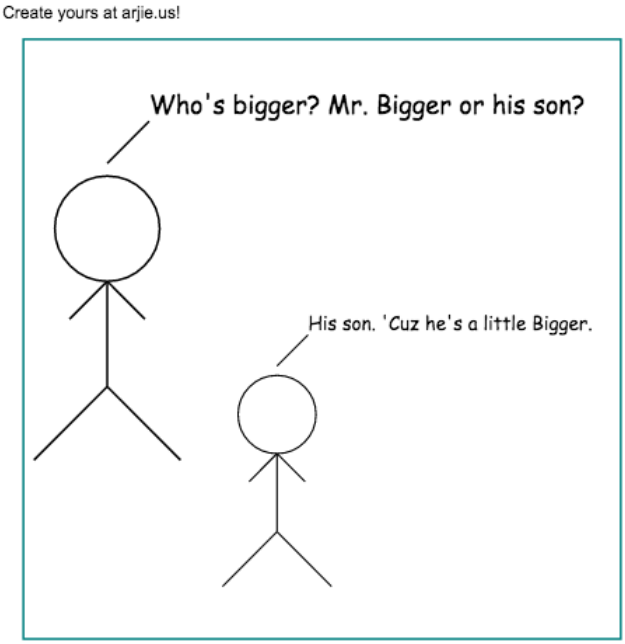
insert

Insert the given elem at the very end of `_elems` (that is, at index `_size+1`). Then bubble it up into its correct place in the heap so that the heap property is restored. Double the vector size if you need to grow.

This method does the opposite of `_percolate_down()`, but it's only a few lines of code, so I didn't bother to separate it out into a private helper method. Feel free to do so if your sense of esthetics says otherwise.

To bubble an element up into its correct place, compare it to its parent. If the child is smaller than its parent, simply swap the parent and child and repeat the process at the child's new location until you can't move it up towards the root any more.

Again, note that this can be implemented efficiently by creating a *hole* and moving the hole to its final location before filling it with the element to be inserted.



I made this comic at Arjie.us, a site created by one of your senior CS2C students (in 2018)

delete_min

Remove the element at the root of the heap. If the heap is empty, it must return false.

Deleting the root (min) element can be done as follows.

- Swap out the min element with the element at the end of the heap (the right-most element in the row of leaves in a pictorial representation of the heap).
- `_percolate_down()` this new root to its correct location in the heap.
- Adjust size accordingly.

peek_min

Return a const reference to the heap's min element in constant time. If the heap is empty, you would return the sentinel, conveniently located *you know where*. (Why do we need this method?)

to_string

No instructions, and no hints. Only an example. If you do it, all you have to do is make your string look like the one in Figure 2.

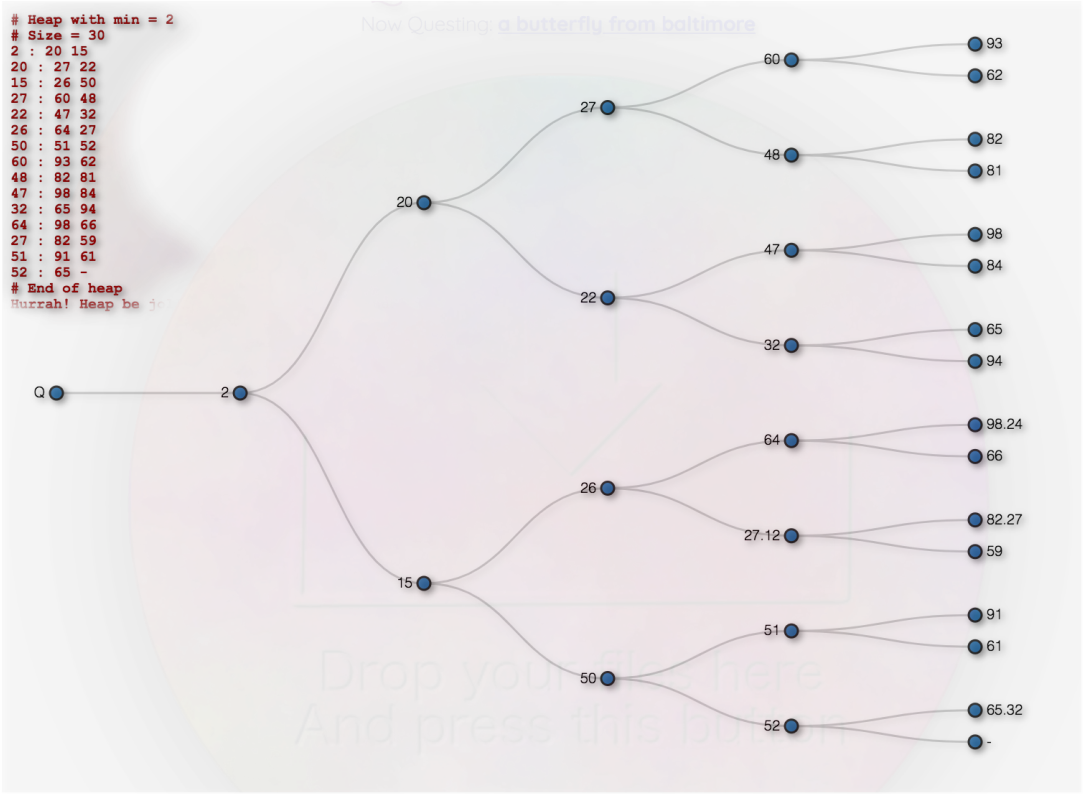


Figure 2. Figure to_string out from this Figure 2 - Ignore the fractional parts of the numbers in the depiction. I stuck 'em in there to help you distinguish between dupes.

If you want to silently skip this MQ, you can return the empty string. If you return anything else, the returned string must be legit. If not, it will result in an "EPIC Fail" which means you can't proceed (and your code/origin info was saved for review).

Remember

Users of your class are not required to furnish any more than the less-than comparison operator for the template parameter class. That is, if I specialize your template like:

```
Heap<Song_Entry> my_heap;
```

you cannot assume that `SongEntry::operator>()` exists. You can, however, assume that `SongEntry::operator<()` will exist.

The Special Heap

The Special_Heap is a subclass of Heap. It should behave just like a regular min heap except that it should support a new method:

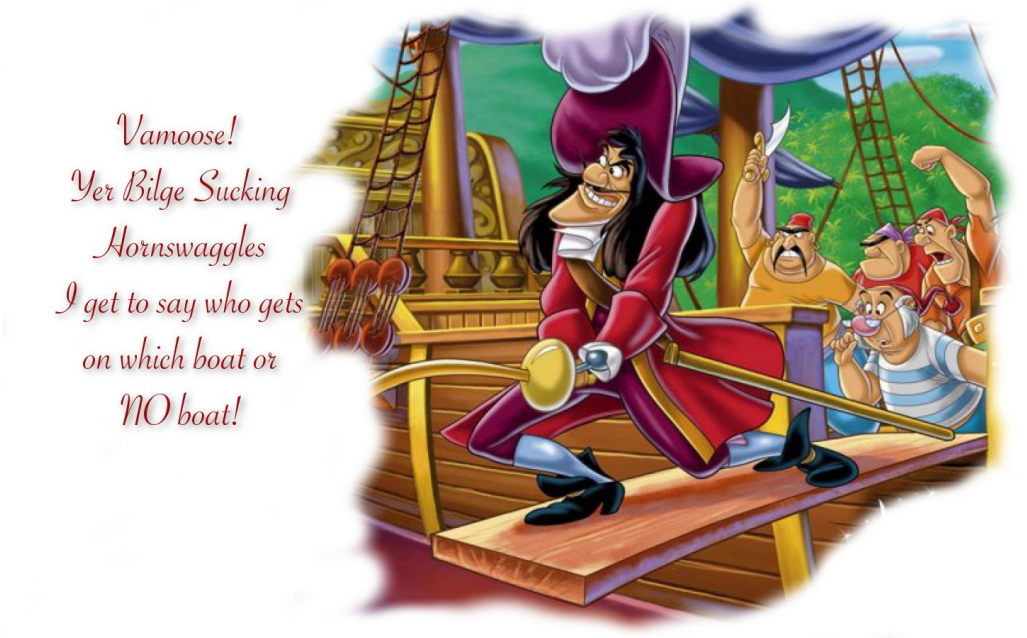
```
get_least_k(size_t k)
```

This method should return a const reference to `_elems` after permuting it such that its k greatest indices contain its `k` least elements - essentially the same as the partition problem from Sharkland with the added requirement that the elements must be in non-ascending (reverse-sorted) order.[1]

Further,

- this new method must have a run-time complexity that is at worst O(k log N). and
- it must work in-place, within the heap's data array (using O(1) space).

To make your task easy, you are allowed to destroy your data array's heapiness in the process. You may mark the heap as empty before returning from this method.



That time when Hook pulled the pecking order before scuttling the ship

[1] Sibei Wan, Fall 2020, pointed this out.

Strategy

Assuming k starts at 0 it's clear that finding the 0-least element would take constant time. That's 0 log N.

So far so good.

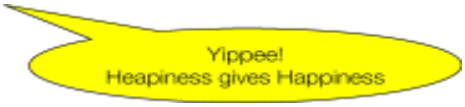
A min-heap (your base class) already satisfies that condition. The challenge is to generalize this property so it works for not just the k=0'th element but for any k.

Since you're allowed to permute the internal data array when the call to this function arrives, you may be thinking that you could simply use the pivot-based partitioning strategy from your shark quest to return the least-k elements. But remember that its complexity is O(N) on average.

Even if its worst-case quadratic complexity was deliberately ignored, it still wouldn't pass our test because we want the k-least elements in O(k log N) time, not O(N).

The key insight here is to see that there is some ordering information already implicit in your heap that Quickselect fails to exploit. There may be several elements in the array that don't have to be examined at all.

Thus, by leveraging the heapiness of the array, you might be able to extract its k-least elements with a worst case of O(k log N). How? The presence of k in O(k log N) is a hint that you're allowed to do k O(logN) operations on the array.



Perform k `delete_min()` operations. Since each `delete_min()` takes at most log(N) time, you can be assured that your worst case behavior will be no worse than O(k log N).

All right. That solves the performance part of the problem. How about the space part? Can we make it happen in-place? (Ponder that before reading ahead).

Do it in-place

Welcome back (I hope).

Leverage another important insight: Every call to `delete_min()` is guaranteed to reduce the size of the heap by 1. Simply move each removed element into the just vacated spot, which is always the spot after the last occupied one.

If you perform k such `delete_min()`'s in a row, each time putting the deleted element into the newly created vacant spot, your k least elements will be in the k greatest indices of the array.

Now you can return a const reference to it.

get_least_k

If you have your min heap working properly this function should be a breeze to implement.

Simply get the min element, then delete it, then copy it into the just vacated cell of the heap which would have shrunk by one element upon the `delete_min()` operation. Repeat this step k times. At the end, mark the heap as empty (set size to 0), and return a const reference to your data array, `_elems`.

Suppose the heap is not large enough to complete the request. Then we have two options. Either we could throw an exception and complain, or you could simply return `_elems` as is, with no changes. Jack Morgan and I had a conversation on this topic in a previous quarter if you're interested. This method should respond to impossibly large requests by returning `_elems` as is.

One way to make sense of this behavior is to imagine that if you ask someone to do an impossible job, they refuse to touch it.

I don't think you need it, but just in case… Figure 3 shows a partial picture of the `Special_heap` class definition.

```
// This is a sub-class of Heap that provides a single extra method called
// get_k_smallest(size_t k), which has a worst-case performance of O(k log N
template <typename T>
class Special_Heap : public Heap<T> {
public:
    Special_Heap() : Heap<T>() {};
    Special_Heap(const vector<T> &vec) : Heap<T>(vec) {}

    const vector<T> &get_least_k(size_t k);

    friend class Tests; // Don't remove this line
};
```

Figure 3. The `Special_Heap` Class

Moar Discussion Points

Here are a few more discussion questions if the rest got picked off by your classmates before you got them:

- Why you would use Quickselect (or Quicksort) at all instead of simply using a special heap (or Heapsort)?
- Why not use a balanced binary search tree for your backing store? If you did, how does that affect the running times of the various public facing heap operations? What is the tradeoff, if any?

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box
3. Drag and drop your source files[2] into the button and press it

Wait for me to complete my tests and report back (usually a minute or less).

Don't worry. Be Heapy.

Heapy Hacking,

&

[2] `Heap.h` & `Special_Heap.h`