Binary Heaps & Heapsort

The Sorting Application

The examples in this document all part of the Java Application `Sorting`. Download the source archive Sorting.zip and install it as a **Java Application with Existing Sources**. See the Using NetBeans document for details.

Like others, the `Sorting` project has multiple Main Classes intended to illustrate various independent features. The simplest way to run the various classes is to right-click the file and select **Run File** either in the file itself or from the listing in the **Projects** window.

The Java files from the project which are referenced in this document are:

```
adapter/
  QueueAdapter
util/
  BinaryHeap
demo/
  PrioQueueDemo
  BinaryHeapDemo
  BuildHeapDemo
  HeapSortDemo
```

Priority Queues

A priority queue is like a queue in that it supports add and remove operations:

- `add(e)`, `offer(e)`: add an element
- `element()`, `peek()`: read the "highest priority" element
- `remove()`, `poll()`: read & remove the "highest priority" element

The usual FIFO queue is like a list in that `add` operations go on one end while `element` and `remove` are from the opposite end. In contrast, a *priority* queue is one with no particular structure; the requirement is that `element` and `remove` always use a **smallest** element with respect to the given order.

By using a suitable comparator, we can make easily change the notion of "smallest" by "largest". Ultimately the notion of the priority queue is quite general in that a FIFO queue can be seen a priority queue with comparison of elements "by time stamp on entering the queue."

Java Priority Queue

The Java priority queue is the appropriately named generic class `PriorityQueue` implementing the `Queue` interface. The key constructors are these:

- `PriorityQueue<E>()`: the "current" element for selection is the smallest element with respect to the natural ordering on `E`, i.e., `E` is assumed to implement the `Comparable` interface.
- `PriorityQueue<E>(int initialCapacity)`: specify the initial capacity.
- `PriorityQueue<E>(int initialCapacity, Comparator<? super E> cmp)`: specify the initial capacity and the desired `Comparator` for `E` objects.

The `initialCapacity` parameter suggests that the underlying implementation is array-based. We will see such an implementation below, the standard implementation called the *Binary Heap*.

List-based implementations

There are actually two potentially viable list-based implementations:

a. An **unordered LinkedList**. In this case we simply add elements to the front of the list (or back), making the `add` operation constant time. It is the others which have a significant cost. Both `element` and `remove` must search the entire list for a smallest element, making them linear time operations.
b. An **reverse-ordered ArrayList**. In this case the `element` and `remove` operations are constant time, since the smallest element is always the last one. It is the `add` which has significant cost, since we must find the correct position to add (logarithmic), plus shift all smaller elements up 1 (linear time).

	comparisons	data moves	total
element	(a) O(n) (b) O(1)	(a) O(1) (b) O(1)	(a) O(n) (b) O(1)
add	(a) O(1) (b) O(log n)	(a) O(1) (b) O(n)	(a) O(1) (b) O(n)
remove	(a) O(n) (b) O(1)	(a) O(1) (b) O(1)	(a) O(n) (b) O(1)

We might conclude that (b) is the "winner", but both are subject to linear time operational costs when all three operations are employed. It is the goal of various heap structures to produce either constant or logarithmic time for all three operations.

Heaps

A *heap*, or more strictly speaking, a *min heap*, is an implementation of a priority queue in the form of a hierarchical tree-like structure where, at each node, the element is smaller than or equal to the elements at the child nodes. In particular, the root must be holding the least element. A *max heap* is one in which the notion of ordering of elements is reversed. In practice, a max heap is simply a min heap in which we use the "reverse comparator."

Test Programs/NetBeans

The project we're using is `PrioQueueDemo`. The following test program illustrates several standard usages of Java's `PriorityQueue` class.

demo.PrioQueueDemo

Note the output of the statements which print the queues:

```
System.out.println("queue:     " + queue);
System.out.println("queue_rev: " + queue_rev);
```

It may not make sense what this output reflects, but it reflects the underlying structure which we'll soon see.

BinaryHeap Implementation

Like our other user-defined data structures, extensions of the `QueueAdapter` class below give us the ability to use our user-defined `BinaryHeap` class as we would a `PriorityQueue`.

adapter.QueueAdapter

and

util.BinaryHeap

A *Binary Heap* is a heap which is maintained as a complete binary tree, meaning all levels are full except possibly the last level, which is filled out from the left. A binary heap is most easily implemented by an array using data members similar to an ArrayList:

```
private int size = 0;
private E[] data;
private Comparator<? super E> cmp = null;
```

Like SearchTreeSets, we need a comparator to establish the heap and also use a `myCompare` function in an analogous way.

Suppose we use the array `data` in positions 1 through `size`, then

- `data[1]` is the root
- `data[2*n]` and `data[2*n+1]` are the left and right children of `data[n]`, respectively
- `data[n]` has only a left child if `2*n == size`
- `data[n]` is a leaf if `2*n > size`

A node `n != 1` can always determine its parent node by the operation:

```
parent = n/2
```

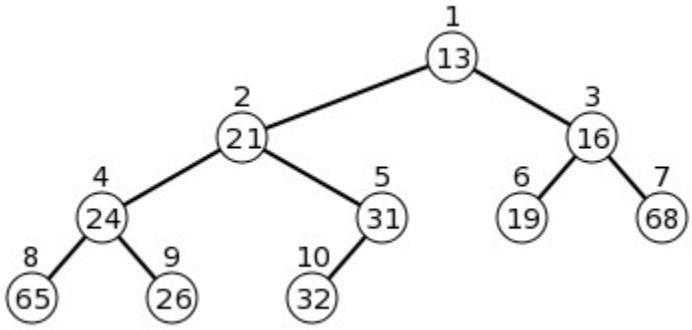
Saying that a complete binary tree satisfies the heap property means that at each node `n`,

```
data[n/2] ≤ data[n]          (n != 1)
```

or, looking at it the other way:

```
data[n] ≤ data[2*n]          (when data[2*n] exists)
data[n] ≤ data[2*n+1]        (when data[2*n+1] exists)
```

Here is an example:



The coding issue involved in using a complete binary tree as a heap is in maintaining the heap property with insertions and removals.

Adding: percolate up

Inserting an element, `elt`, means creating a new position in the array. This position is called a "hole" whose initial value is:

```
hole = ++size;
```

The operation we do is called "percolate up" in which the hole is moved to the correct position in the tree so that we can do the insertion:

```
data[hole] = elt
```

and know that the heap property is maintained:

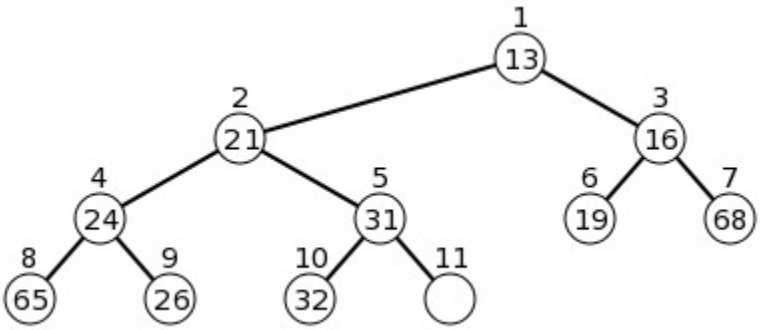
```
data[hole/2] ≤ data[hole]
```

And so, if necessary, the `hole` moves up the tree from the leaf while the data in the parent is shifted down with an operation like this:

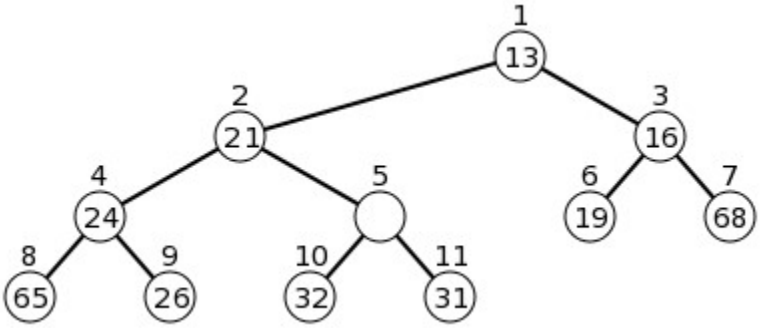
```
data[hole] = data[hole/2];
hole = hole/2;
```

For example, the steps to do the insertion of the value 17 in this tree would be this:

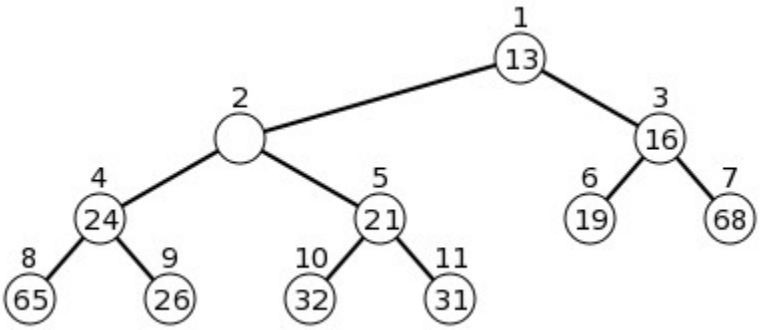
a. Increase the size to 11 and create a hole at position 11 like this:



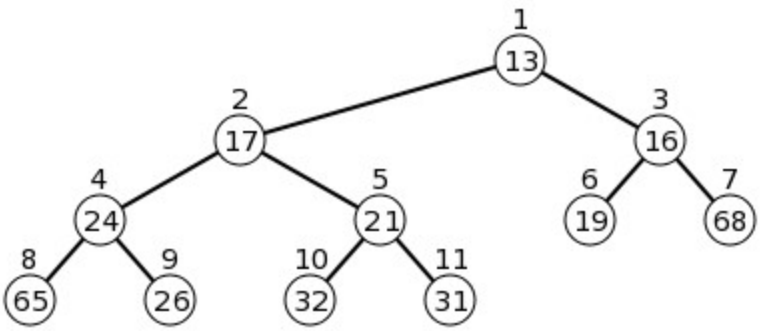
b. Comparing 17 with the hole's parent would make the parent value shift down:



c. Again, comparing 17 with the hole's parent would make the next parent value shift down:



d. Finally, 17 is larger than the parent value at 1, so it gets inserted into the hole:



Here is the code:

```
  private int percolateUp(int hole, E elt) {
    while (hole != 1 && myCompare(data[hole/2], elt) > 0) {
      data[hole] = data[hole/2];         // shift parent down
      hole = hole/2;
    }
    return hole;
  }

  public boolean add(E elt) {
    if (size == data.length) {
      resize(2*data.length);
    }
    int hole = percolateUp( ++size, elt );
    data[hole] = elt;
    return true;
  }
```

Removal: percolate down

Removal means deleting the node at the root, i.e., at position 1. This creates a hole into which we want to eventually insert the last element in the array which must give up its position:

```
elt = data[size--];
hole = 1;
```

The hole must move down the tree (percolate down) with an operation like:

```
hole = 2 * hole;        or        hole = 2 * hole + 1;
```

until we can set

```
data[hole] = elt
```

satisfied that the heap property is maintained:

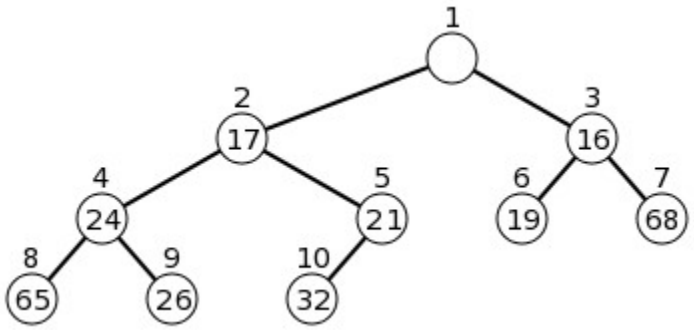
```
elt ≤ data[2*hole]
elt ≤ data[2*hole+1]
```

When a hole goes down from parent to child, always choose a child with a **smaller** value.

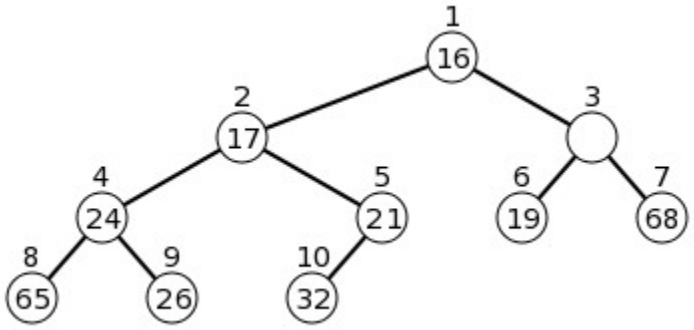
Programmatically, percolate down is a more complicated operation due to having to choose between two children and determining how many children do deal with.

If we do a removal on the tree above (after the insert of 17 is complete) these are the steps;

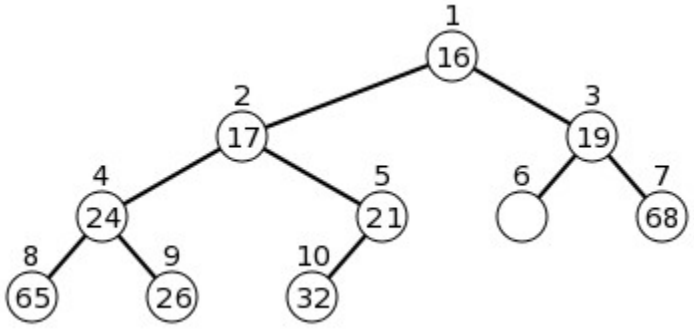
a. Return the value and create a hole at position 1. Pull out the value at node 11 (`elt=31`) and decrease the size to 10:



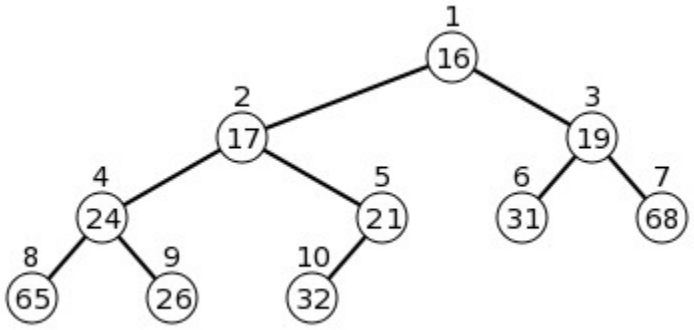
b. 31 is bigger than the smallest child, 16, so move the hole down to position 3 and move 16 up:



c. Again, seeing that 31 is bigger than the smallest child, 19, move the hole down to its position 6 and move 19 up:



d. Finally, since the hole has no children we move 31 into position 6:



```
  private int percolateDown(int hole, E elt) {
    while (true) {
      int left = 2 * hole, right = 2 * hole + 1;

      if (left > size) { // hole is a leaf
        break;
      }

      // so hole must have at least a left child

      int child;   // child with the smallest value

      if (left == size) {
        // hole has only a left child
        child = left;
      }
      else if (myCompare(data[left], data[right]) <= 0) {
        // 2 children with smaller left
        child = left;
      }
      else {
        // 2 children with smaller right
        child = right;
      }

      if (myCompare(elt, data[child]) <= 0) {
        break;    // elt smaller than least child: correct position
      }
      else {
        data[hole] = data[child];  // shift child up
        hole = child;              // move hole down
      }
    }
    return hole;
  }

  @Override
  public E remove() {
    if (size == 0) {
      throw new NoSuchElementException("remove");
    }
    E top = data[1];
    E insert = data[size--];   // get element to reinsert from root
    int hole = percolateDown(1, insert);
    data[hole] = insert;
    return top;
  }
```

BinaryHeap Test Programs

To test the effectiveness of this new class, in `PrioQueueDemo`, you can uncomment the import line:

```
import heap.BinaryHeap;
```

and switch the commented and uncommented sections:

```
//    Queue<Integer> queue = new PriorityQueue<>();
//    Queue<Integer> queue_rev = new PriorityQueue<>(20, rev_cmp);

    Queue<Integer> queue = new BinaryHeap<>();
    Queue<Integer> queue_rev = new BinaryHeap<>(20, rev_cmp);
```

A second test program is better for showing the underlying structure which provided by the `BinaryHeap` class:

demo.BinaryHeapDemo

Analysis

We see that, since the tree maintained is always a complete binary tree, the height (when non-empty) is `flr(log n)`. This becomes, up to a constant factor, the maximum number of comparisons + data movements for either an element insertion or deletion.

Heapsort concept

The idea of heapsort is that it is conceptually similar to selectionSort with a time improvement on the selection operation. An oversimplified version is as follows, starting from an array, `A` to be sorted, using an auxiliary binary heap `H`:

```
// create the heap
int i = 0;
for( i = 0; i < A.length; ++i ) {
  H.add(A[i]);
}
// remove one-by-one, putting back into array
i = 0;
while (! H.isEmpty()) {
  A[i++] = H.remove();
}
```

Since both `add` and `remove` are `O(log(n))` operations, the entire worst-case time is `O(n*log(n))`. In particular, the `O(n)` smallest element selection process used by selection sort is replaced by the efficient `O(log(n))` heap remove operation.

Improvements

Although we cannot do better than an overall `O(n*log(n))` time, we can do make improvements with respect to several points:

1. The "create the heap" portion of the code can actually be done in linear time, `O(n)`, instead of `O(n*log(n))` as currently written. The replacement code is the `buildHeap` operation described in the next section.
2. The `buildHeap` operation which replaces repeated adds is done **in place** within the array itself, obviating the necessity of an external heap.
3. Removals and placement within the array can also be done in place in the array. The outcome is that the auxiliary heap, `H`, is not necessary, thus reducing the `O(n)` extra memory requirement to `O(1)`.

Dealing with actual array ranges

Arrays are 0-based by default, and when we deal with sorting, we have to work with a range between `from` and `to`. To realize this in terms of being an "in-place" heap, we must constantly translate between the actual array range and the 1-based range required by the binary heap.

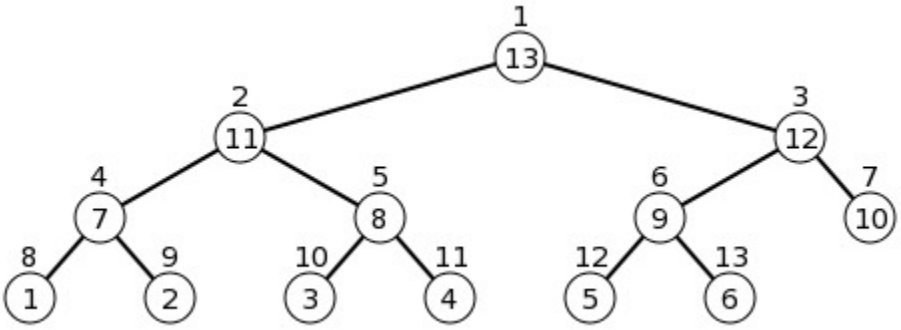
The buildHeap operation

The `buildHeap` process effectively goes row-by-row, starting from the second-to-last row, doing percolateDown operations, thereby making the subtree a heap. Here is a depiction of the process (a "worst-case" situation):

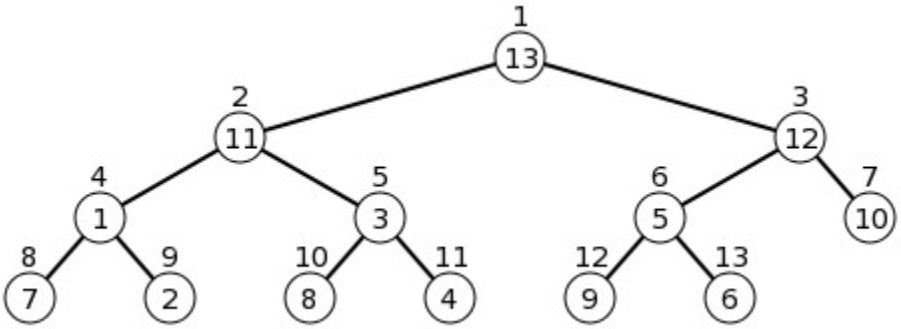
a. The raw array of `size` 13 is

```
[13, 11, 12, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6]
```

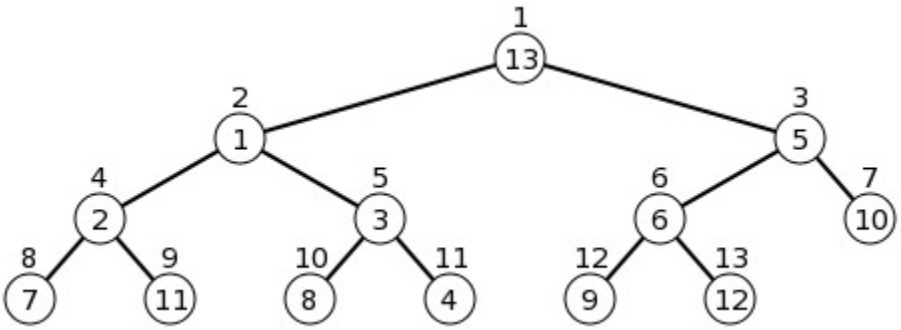
Seen as a tree, we have:



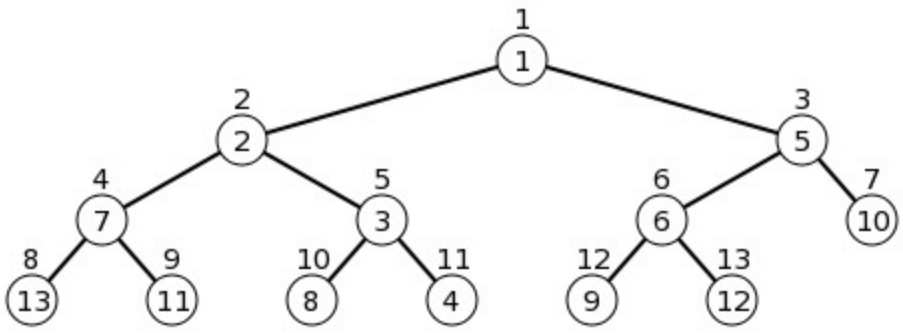
b. The process starts at node 6 (`size/2`) and works backward. After nodes 6, 5, 4 are done we have:



c. After nodes 3, 2 are done we have:



d. Finally, after node #1 is done:



Here is the code from `BinaryHeap`:

```
public void buildHeap(E[] a) {
  size = a.length;
  if (size > data.length) {
    data = (E[]) new Object[size+1];
  }
  for (int i = 0; i < a.length; ++i) {
    data[i+1] = a[i];
  }
  for (int i = size/2; i > 0; --i) {
    int hole = i;
    E elt = data[hole];
    hole = percolateDown(hole, elt);
    data[hole] = elt;
  }
}
```

As you will note, this version of the buildHeap operation uses a spare array for the sole purpose of translating the array index range from 0-based to the 1-based index range needed by the heap operations.

Here is a test program you can use to see the `buildHeap` operation isolated:

demo.BuildHeapDemo

Cost of buildHeap

As discussed in the textbook, this operation runs in **linear** time on the number of nodes. This can be proved from two facts about a **perfect** binary tree (all levels full):

- the total number of nodes is $2^{h+1} - 1$
- the sum of the heights of all the nodes is $2^{h+1} - 1 - (h + 1)$

Roughly, with n nodes, the sum of heights is also n.

Counting comparisons, each step in percolate_down costs 2 and the total number of percolate_down operations is no more than the sum of the heights. Conclude that:

Number of comparisons in buildHeap ≤ 2*n

A binary heap need not be a perfect tree, but the analysis comes out about the same.

HeapSort

The heapsort algorithm uses a binary heap to do its work. The heap is built as a **max** heap, using a reverse comparator. The program below indicates the heapsort behavior which works in two phases:

a. **convert the array into a MAX heap**: run the `buildHeap` operation on the original array
b. **remove (maximum) and put into position**: one at a time, remove from the heap, re-inserting the elements into the array from last (largest) to first (smallest)

You can see total effect by running the `ShowSorts` program after setting:

```
choice = "heap";
```

Worst-case time

The time per phase for an array of size `n` is:

a. buildHeap: `O(n)`
b. removals and placement: `O(n*log(n))` since there are `n` removals, and a removal counts `O(log(n))`

Thus the total time is the optimal `O(n*log(n))`.

Heapsort has O(n) best case time

If the array to be sorted consists of all equals, then each removal requires only two comparisons, because the replacement of the root will be discovered to be "OK where it is" immediately, and need not percolate down the tree. Thus, for `n` removals, the total cost will be `O(n)`.

Heapsort uses O(1) extra space

A key point is that the `heapSort` algorithm uses the array itself as a MAX Heap and does not need a heap structure separate from the array. We need a MAX heap so that the vacated positions, coming down from the right end of the array, can be correctly replaced by the largest values.

As mentioned above, the technical difficulty in writing the real `heapSort` algorithm has to do with translating the array index range `fromIndex:toIndex`" into the range "`1:size+1`" which is natural for a complete binary tree.

Simplified Demo

This demo program uses a simplified version of the `heapSort` algorithm which employs a separate heap. The code is dedicated to creating the output depicted in a way which suggests what happens in the actual `heapSort` algorithm.

demo.HeapSortDemo

A sample run (with annotations) indicates the correct ideas:

```
to sort: [5, 3, 7, 1, 4, 8, 2]

********************** buildHeap restructures the array:
[8, 4, 7, 1, 3, 5, 2]
|         2              this output
|      7                 indicates the
|         5              structure
|   8                    of the heap
|         3
|      4
```

```
|         1
----------------------- remove() = 8
[7, 4, 5, 1, 3, 2][8]
|      5                the removed element (max)
|         2             goes into the top of
|   7                   array, which is precisely
|         3             the position vacated
|      4                by the remove operation
|         1
----------------------- remove() = 7
[5, 4, 2, 1, 3][7, 8]
|      2
|   5
|         3
|      4
|         1
----------------------- remove() = 5
[4, 3, 2, 1][5, 7, 8]
|      2
|   4
|      3
|         1
----------------------- remove() = 4
[3, 1, 2][4, 5, 7, 8]
|      2
|   3
|      1
----------------------- remove() = 3
[2, 1][3, 4, 5, 7, 8]
|   2
|      1
----------------------- remove() = 2
[1][2, 3, 4, 5, 7, 8]
|   1                   no more removals necessary

sorted: [1, 2, 3, 4, 5, 7, 8]
```
