Heapsort

A heap also can be used to sort a set of elements. Suppose that we have a heap, represented as an array. (We have seen two algorithms for building a heap.) Recall that the 0-th element of the array is empty, so that indexing the parent-child can be done with a simple formula (which you should memorize). Here is an algorithm for sorting the n elements in the array.

```
INPUT:  heap  a[1 .. n]  (minimum at root)
OUTPUT: a sorted array a[1 .. n]   (sorted from maximum to minimum)

for i = 1 to n{
    swap( a[1], a[n+1 - i])
    downHeap( a, 1, n-i)          //  overloaded downHeap (see below)
}
```

The `downHeap` method here has three parameters, not two. The third parameter specifies the maximum index that we can `downHeap` to. The idea here is that we are repeatedly removing the minimum element from the heap and storing it in the array (i.e. see the `swap`). After swap i, the heap is of size $n-i$ only. Thus we `downHeap` the root to at most index $n-i$. The last i elements in the array stored the smallest i elements in the set. The result is that the array will be sorted from largest to smallest.

`Heapsort` is $O(n \lg n)$ since we loop n times and at the $i-th$ iteration, we need to perform up to $\lg(n-i)$ swaps. Thus, in the worst case, the number of swaps is $\sum_{i=0}^{n-1} \lg(n-i)$ which is identical to $\sum_{i=0}^{n-1} \lg i$ which we saw earlier is $O(n)$.

One detail worth mentioning is that the resulting elements are ordered from max to min, not min to max. This is not a problem since we can reverse the elements in an array in $O(n)$ time, just by swapping i and $n+1-i$ for $i = 1$ to $n/2$. Or, if our goal in the first place was to order in min to max order, we could construct the heap using a comparison method that is opposite order.

Example

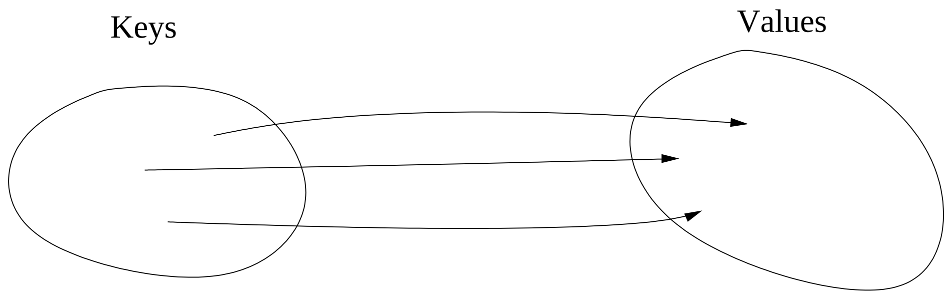
```
  1   2   3   4   5   6   7   8   9
-------------------------------------
  a   d   b   e   l   u   k   f   w |
  b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
  d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
  e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
  f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
  k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
  l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
  u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
  w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
  w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Maps

We have talking about collections of objects and we have looked at operations such as finding, adding and removing objects from the collections. In order to gain efficiency in these operations, we have been assuming that the objects can be compared to each other. This defines an ordering of the objects in the collection. We have seen several data structures and algorithms that took advantage of this ordering: for example, arrays, binary search trees (and to some extent heaps).

Having an ordering is convenient because we can focus our attention on algorithms, rather than the particulars of how the ordering is defined (e.g. in Java, the details may be hidden in the `compareTo` implementation). Often, though, we do want to be explicit about how the ordering is defined. For example, entries in telephone books and personal address books are ordered by names. The entry contains the name, as well as the address and telephone number. For employee records, you might want to search by employee name or by social insurance number. Thus, in many situations it is natural to distinguish the data we are using for the search (called the *key*) from the data we are searching for (called the *value*).

A *map* as a set of *entries* which are ordered pairs $\{(k, v) : k \text{ is a key }, v \text{ is a value}\}$, such that, for any key k, there is at most one value v. I say "at most" because there typically are keys for which there is no corresponding value. (The keys and values don't necessarily need to be comparable i.e. we don't require that there is a natural ordering on them).



In Java, there is a `Map<K,V>` interface whose key and value types are generic. Any class that implements this interface has methods such as:

- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`
- `v get(Object key)`
- `v remove(Object key)`
- `int size()` - the number of entries in the map
- `V put(K key, V value)`

The `Map` interface is implemented by class `TreeMap` which implements a binary search tree on the keys. (There is no specification for how the values are organized, however.) Here is an example:

```
Map<String,Integer>  numbers = new  TreeMap<String,Integer>();
numbers.put("five", 5);
numbers.put("a dozen", 12);
numbers.put("twelve",12);
numbers.get("twelve");     // returns 12
numbers.get("a dozen");    // returns 12
```

The numbers "5" and "12" are automatically wrapped into objects of type `Integer`.

In the example above, which used the `TreeMap` class, the keys are stored in a binary search tree. Binary search trees are relatively fast ways to search through a collection. However, they still require $O(\lg n)$ time, where n is the size of the collection. There are many situations where this is just too slow. If you have millions of items in your collection, it will take you twenty time steps or more (2^{20} = one million) to find an item.

Is there an alternative? Let's take an extreme case that the keys are integers, for example, social insurance numbers. Social insurance numbers have 9 digits, so you could define an array of size 10^9 which stores references to objects of the class `Employee`. This is called *direct addressing.* You provide the key, and in time $O(1)$ – namely an array access – you have the address of the employee with that social insurance number (if such an employee exists. Otherwise the entry in the array is `null`). We will return to this idea next lecture.