CS 228: Introduction to Data Structures
Lecture 32
Monday, April 13, 2015

Hashing (continued)

Rehashing and the Load Factor

When a hash table becomes too full, collisions increase, and performance deteriorates. To avoid this, it helps to, from time to time, expand the array and ***rehash*** the elements. More precisely, we expand and rehash when

$$n / M = \text{(number of entries)} / \text{(number of buckets)} > L,$$

where L is a predetermined value called the ***load factor***. If the load factor is exceeded, the table size is rehashed so that the hash table has approximately twice the number of buckets. (Rehashing, by the way, is the reason why the iteration order may change.) The default load factor for `HashMap` is 0.75, a number that offers a good tradeoff between time and space costs. A higher load factor decreases the space overhead, but increases the time to look up an entry.

Besides the load factor, another parameter that affects the performance of hashing is the initial table capacity; i.e., the initial number of buckets allocated to the table. The initial capacity should be chosen so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, rehashing will never be needed. `HashMap`'s default initial capacity is 16.

Hashing and Sets

Hashing can be used to represent sets: rather than storing key-value pairs, just store keys. Java's `HashSet<E>` implements the `Set` interface using a `HashMap` as the backing store. Under the uniform hashing assumption, a `HashSet` offers constant time performance for the basic operations: `add()`, `remove()`, `contains()`, and `size()`. Iterating over this set requires time proportional to the number of elements in the `HashSet` plus the number of buckets of the backing `HashMap` instance. There is no guarantee as to the iteration order of a `HashSet`; in fact, there is no guarantee that the order will remain constant over time.

Priority Queues

A ***priority queue*** is used to *prioritize* a collection of key-value pairs, based on a total order on the keys (e.g., numerical or alphabetical order). For instance, the values could be airline flights, and the keys could be (arrival or departure) times, as in

(key, value) = (11:16 am, DL3347).

An entry with any key may be inserted at any time. However, you may *only* examine or remove the entry whose key is the lowest. This limitation helps to make priority queues fast.

Priority queues are often used as “event queues” in simulations (e.g., airport simulations). Each value on the queue is an event that is expected to take place, and each key is the time the event takes place. A simulation operates by removing successive events from the queue and simulating them. This is why most priority queues return the minimum, rather than maximum, key: the next event to simulate is the one that occurs earliest.
The basic priority queue methods are

- **insert:** Add a new element to the queue.
- **min / max:** Return the element with the smallest / largest key in the queue.

- **removeMin / removeMax:** Return and remove the element with the smallest / largest key in the queue.

To be consistent with the Java `Queue` interface, we will refer to these methods by the following names:

- `add()` = insert

- `peek()` = min / max

- `remove()` = removeMin / removeMax

For simplicity, our illustrations will only show keys, not values.

Simple Implementations

It is easy to implement a priority queue using a list or array, sorted or unsorted. The following table shows the worst-case running times; n is the number of entries in the queue. We leave the implementation details to you.

	List/Array Sorted	List/Array Unsorted
`peek()`	O(1)	O(n)
`add()`	O(n)	O(1)
`remove()`	O(1)	O(n)

Notes

- If we're using an array-based data structure, these running times assume that we don't run out of room. If we do, it will take O(n) time to allocate a larger array and copy the entries into it. However, if we double the array size each time, the *amortized* running time will still be as indicated.

- Removing the minimum from a sorted array in constant time is most easily done by keeping the array in largest-to-smallest order, whereas for a list smallest-to-largest is better.

Implementing Priority Queues via Binary Heaps

By combining arrays and trees, we can obtain a priority queue where both `add()` and `remove()` take O(log n) time. We first need a new concept.

A ***complete binary tree*** is a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right as in the illustration below. Just the keys are shown; the associated values are omitted.

```
                 2
                / \
               /   \
              5     3
             / \   / \
            9   6 11  4
           / \ /
         17 10 8

   index:  0  1  2  3  4  5  6  7  8  9
          +--+--+--+--+--+--+--+--+--+--+
          | 2| 5| 3| 9| 6|11| 4|17|10| 8|
          +--+--+--+--+--+--+--+--+--+--+
```

A ***binary heap*** is a complete binary tree whose entries satisfy the following:

Heap-order property: No child has a key that is smaller than its parent's key.

Observe that every subtree of a binary heap is also a binary heap, because every subtree is complete and satisfies the heap-order property.

Because they are complete, binary heaps are often stored as arrays of entries, ordered by a level-order traversal of the tree, with the root at index 0. This mapping of tree nodes to array indices is called ***level numbering***.

Fact. If a node's index is i, its children's indices are 2i+1 and 2i+2, and its parent's index is floor((i – 1)/2).

Hence, no node needs to store explicit references to its parent or children[1].

We can use either an array-based or a node-and reference-based tree data structure, but the array representation tends to be faster (by a large constant factor) because there is no need to read and write node references, cache performance is better, and finding the last node in the level order is easier.

Do not confuse binary heaps with binary search trees! Unlike a BST, a heap can have duplicate keys. Further, in a heap we do not require the keys in the left subtree to be smaller than the key at the root or the keys in the right subtree. For instance, the tree in the previous page is a heap, but not a BST.

[1] This method of representing binary trees was introduced by the Austrian historian Michaël Eytzinger (1530-1598) more than 400 years ago. Eytzinger's wanted to represent genealogies without the need for a diagram such as a family tree. In his system — called an *ahnentafel* in German — the subject is listed as No. 1, the subject's father as No. 2 and the mother as No. 3, the paternal grandparents as No. 4 and No. 5 and the maternal grandparents as No. 6 and No. 7, and so on.

Implementing the Methods

We now show how to implement a priority queue with a binary heap. For simplicity, we assume that the queue contains only keys, which are of some `Comparable` type (full details are on Blackboard):

```
public class
BinaryHeap<E extends Comparable<? super E>>
```

`E peek()`

The heap-order property ensures that the entry with the minimum key is always at the top of the heap. Hence, we simply return the entry at the root node. If the heap is empty, return `null` or throw an exception.

`boolean add(E x)`

We place the new entry `x` in the bottom level of the tree, at the first free spot from the left. (If the bottom level is full, start a new level with `x` at the far left.) In an array-based implementation, we place `x` in the first free location in the array[2].

---

[2] As usual, if we run out of space, we double the size of the array. This only adds O(1) amortized time to the `add()`.

The new entry's key may violate the heap-order property. If so, we correct this by having the entry ***percolate up*** the tree until the heap-order property is satisfied. That is, we compare x's key with its parent's key; if x's key is less, we exchange x with its parent, then repeat the procedure with x's new parent.[3]

Example. Suppose we insert 2 into our previous heap.

```
        2                  2
       / \                / \
      /   \              /   \
     5     3            5     3
    / \   / \    =>    / \   / \    =>
   9   6 11  4        9   6 11  4
  / \ /              / \ / \
17 10 8            17 10 8  2


        2                  2
       / \                / \
      /   \              /   \
     5     3            2     3
    / \   / \    =>    / \   / \
   9   2 11  4        9   5 11  4
  / \ / \            / \ / \
17 10 8  6         17 10 8  6
```

[3] See the pseudocode in the appendix.

As this example illustrates, a heap can contain multiple entries with the same key, so its keys do not constitute a set. (After all, in a typical simulation, we can't very well outlaw multiple events happening at the same time.)

Fact. Percolate-up preserves the heap-order property. Therefore, so does `add()`.

To convince ourselves of this fact, let's look at a typical exchange of `x` with a parent p during percolate-up. Suppose that, as shown below, s is `x`'s sibling and `l` and `r` are `x`'s children.

```
          p                x
         / \              / \
        s   x     =>     s   p
       /\   /\          /\   /\
           l  r             l  r
```

Since the heap-order property was satisfied before the insertion, we know that $p \leq s$, $p \leq l$, and $p \leq r$. We only swap if $x < p$, which implies that $x < s$. After the swap, x is the parent of `s` and p is the parent of `l` and `r`. All other relationships in the subtree rooted at x are maintained, so after the swap, the tree rooted at `x` has the heap-order property.

Note that the new key does not necessarily have to percolate all the way up to the root, as in the previous example. To convince yourselves of this, try inserting 5 into the final tree in that example.

For speed, don't put x at the bottom of the tree and bubble it up. Instead, percolate a hole up the tree, then fill in x. This modification saves the time that would be spent setting a sequence of references to x that are going to change anyway.

`E remove()`

If the heap is empty, return `null` or throw an exception. Otherwise, begin by removing the entry at the root node and saving it for the return value. This leaves a hole at the root. We fill the hole with the last entry in the tree, which we call x, so that the tree remains complete.

It is unlikely that x has the minimum key. Fortunately, both subtrees rooted at the root's children are heaps; thus, the new minimum key is one of these two children. We percolate x down the heap as follows:

- If x has a child whose key is smaller, swap x with the child having the minimum key.

- Next, compare x with its new children; if x still violates the heap-order property, again swap x with the child with the minimum key.

- Continue until x is less than or equal to its children, or reaches a leaf.[4]

Example. Here's what happens when we apply `remove()` to our original heap.

```
        2                  8
       / \                / \
      /   \              /   \
     5     3            5     3
    / \   / \  =>   / \   / \  =>
   9   6 11  4     9   6 11  4
  / \ /            / \
17 10 8            17 10
```

```
        3                  3
       / \                / \
      /   \              /   \
     5     8            5     4
    / \   / \  =>   / \   / \
   9   6 11  4     9   6 11  8
  / \                / \
 17 10              17 10
```

[4] See the pseudocode in the appendix.

Note. Analogously to `add()`, percolating a hole down the tree and then filling in x is faster in practice than repeated swapping. Nevertheless, for clarity, the pictures are drawn as if swaps took place.

Example. In the preceding example, the entry bubbled all the way to a leaf. This is not always the case, as the next figure shows.

```
       1                   4
      / \                 / \
     /   \               /   \
    2     3      =>     2     3
   / \   / \           / \   /
  9   6 11  4         9   6 11

               2
              / \
             /   \
     =>     4     3
           / \   /
          9   6 11
```

Running Times of the Binary Heap Methods

The running time of `peek` is clearly O(1). The running times of `add` and `remove` depend on the height of the tree.

- `add` puts an entry `x` at the bottom of the tree and percolates it up. At each level of the tree, it takes O(1) time to compare `x` with its parent and swap, if needed. In the worst case, `x` will bubble all the way to the top, and the time is O(height).

- `remove` may cause an entry to percolate all the way down the heap, taking O(height) worst-case time.

On Wednesday, we will sketch the proof of the following.

Height Bound. The height of an n-node heap is at most $\log_2 n$.

The Height Bound implies that `add` and `remove` take O(log n) time in the worst case.

Appendix: Percolation Pseudocode

In the following pseudocode

- `data` is an array that contains the elements of the heap,
- `size` is the number of elements in the heap, and
- `current` is the index of the element to be percolated up or down.

For simplicity, we assume that the heap consists of integer keys. The Java code posted on Blackboard allows more general objects.

```
percolateUp(data,current):
    parent = (current - 1) / 2
    while (current > 0 &&
            data[current] < data[parent])
        swap data[current] and data[parent]
        current = parent
        parent = (current - 1) / 2
```

```
percolateDown(data,current):
    // Find left child of current
    child = 2 * current + 1

    while (child < size)
        if child + 1 < size
            // Find smaller of two children
            if data[child] > data[child + 1]
                child = child + 1
        if data[current] > data[child]
            swap data[current] and
                 data[child]
        current = child
        child = 2 * current + 1
```