

Heaps

The Many Meanings of Heap

The word *heap* has several different common meanings in computer science:

1. It is used as a synonym for an *abstract priority queue*.
2. It is used to refer to a *particular class of concrete implementations for a priority queue*. In this interpretation, some priority queues are heaps and some are not, which is at odds with meaning #1.
3. In a very different interpretation, *heap* is commonly used to refer to the region of memory used for storing entities whose lifetime exceeds that of the execution frame in which they were created (a.k.a. heap = *Object Land*). This is in contrast to the *stack*, which is the region of memory used for storing entities whose lifetime is less than or equal to that of the execution frame in which they were created (a.k.a. stack = *Execution Land*).

Here we shall focus on interpretation #2, though #1 shall be implied in the term *heapsort*.

Heapsort

Heapsort is a sorting algorithm based on priority queues. Here is a generic implementation:

```
// Modify vector to contain elements sorted from low to high
public static <T> void heapSort (Vector<T> v, Comparable<T> comp) {
    // (1) Create an empty priority queue (any implementation will do)
    PQueue pq = new PQueueZZZ(comp);
    // (2) Insert all elements from vector into priority queue:
    for (int i = 0; i < v.size(); i++) {
        pq.enq(v.get(i));
    }
    // Store all elements in sorted order from priority queue into vector
    for (int i = v.size()-1; i >= 0; i--) { // must go from highest index down!
        v.set(i, pq.deq());
    }
}
```

Notes:

- We can supply any `Comparator<T>` to compare elements of type `T`. By the `PQueue` contract, supplying `null` for the comparator will use the `Comparable<T> compareTo()` method for comparisons (if type `T` has one).
- Later, we shall see that heapsort is particularly efficient if a certain implementation of priority queues is used. Some people reserve the term *heapsort* for this particularly efficient version.

Efficiency of Max PQ Operations

What is the *worst-case* asymptotic running times of the following priority queue operations for the specified implementations? Assume `enq()`, `deq()`, and `front()` are invoked on an n -element heap, and `heapSort()` is invoked on an n -element vector.

Implementation	<code>enq()</code>	<code>deq()</code>	<code>front()</code>	<code>heapSort()</code>
Vector sorted low to high				
Vector sorted high to low				
Binary search tree				
Complete heap (this handout)				

The Heap Condition

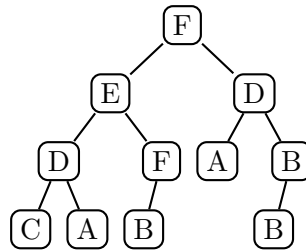
For meaning #2 of heap, a binary tree is a *max heap* if it satisfies the following condition at *every* node in the tree.

Max Heap Condition: The value of a node is \geq the values of all nodes in *both* subtrees.

Similarly, a *min heap* satisfies at every node a min heap condition in which the node value is \leq the values of all values in both subtrees.

Max Heap Example

Consider the following string tree T:



Tree	Max heap?
T	
<code>left(T)</code>	
<code>right(T)</code>	
<code>left(left(T))</code>	
<code>right(left(T))</code>	
<code>left(right(T))</code>	
<code>right(right(T))</code>	

Running Times of Heap Operations

What is the worst-case running time of `front()` on a heap with n nodes?

It turns out that we can make the worst-case time for `enq()` and `deq()` proportional to the height of a heap. What is the worst-case height of a heap with n nodes?

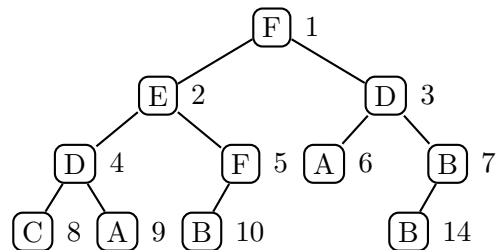
We would like to guarantee the worst-case height of an n -node heap is $\Theta(\log(n))$. In this lecture, we study a restricted form of heap satisfying this condition: the **complete heap**.

Binary Addresses

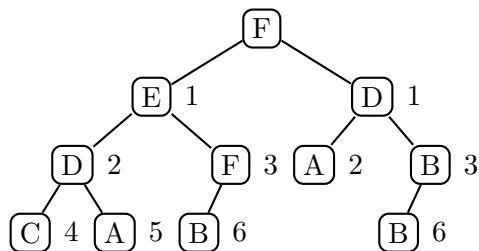
We want to define a notion of *completeness* for a binary tree. To do this, it is first helpful to define the **binary address** of a node in a binary tree:

- The address of the root of the tree is 1.
- The root of the left subtree of node with address a has address $2 \cdot a$.
- The root of the right subtree of node with address a has address $(2 \cdot a) + 1$.

Here are the nodes of T annotated with binary addresses:



Binary addresses are relative to the chosen root. Here are the nodes of the two subtrees of T annotated with binary addresses:



Complete Trees

An n -element binary tree is **complete** if the set of binary addresses of its nodes is the n -element set $\{1, \dots, n\}$.

Example: Reconsider the tree T:

Tree	Complete?
T	
left(T)	
right(T)	
left(left(T))	
right(left(T))	
left(right(T))	
right(right(T))	
left(left(left(T)))	

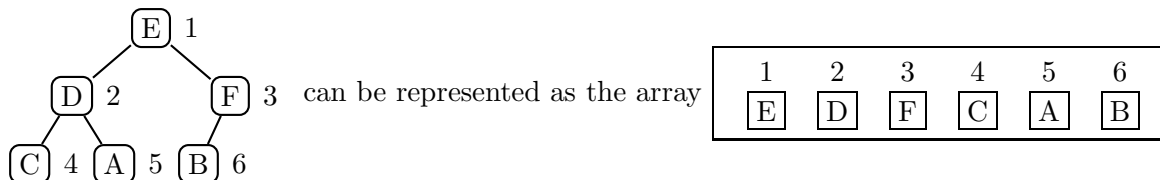
Note:

An n -element binary tree is **full** if it is a complete tree of height h with $2^h - 1$ nodes. Which subtrees of T are full?

What's So Great About Complete Trees?

Complete trees have two important features:

1. The height of an n -node complete tree is $\Theta(\log(n))$ in the worst case.
2. Because the binary addresses cover the range 1 to n , complete trees are easily represented as arrays/vectors. E.g., in languages with 1-based array indexing:



Note: In languages with 0-based array indexing (e.g., Java and C), the index is one less than the binary address:

0	1	2	3	4	5
E	D	F	C	A	B

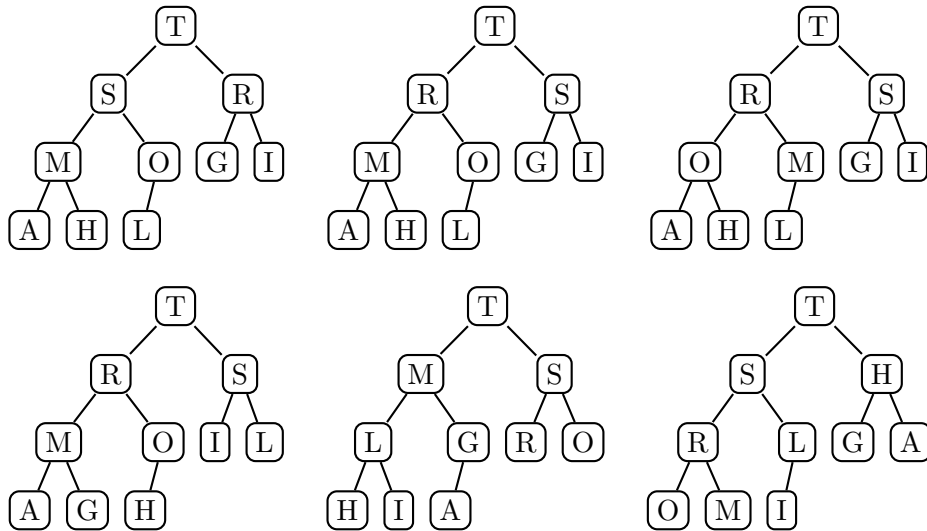
Address Arithmetic For Complete Trees

Operation	Name	Defn. (1-based indexing)	Defn. (0-based indexing)
left address	<code>laddr(index)</code>	$2 * \text{index}$	$(2 * \text{index}) + 1$
right address	<code>raddr(index)</code>	$(2 * \text{index}) + 1$	$(2 * \text{index}) + 2$
parent address	<code>paddr(index)</code>	$\text{index} / 2$	$(\text{index} - 1) / 2$

Complete Heaps

A **complete (max) heap** is a binary tree that is both complete and a max heap.

Example: Below are some sample complete max heaps containing the letters A L G O R I T H M S. These are just some of the many possible complete heaps possible with this set of elements.



Complete Heap Enqueuing

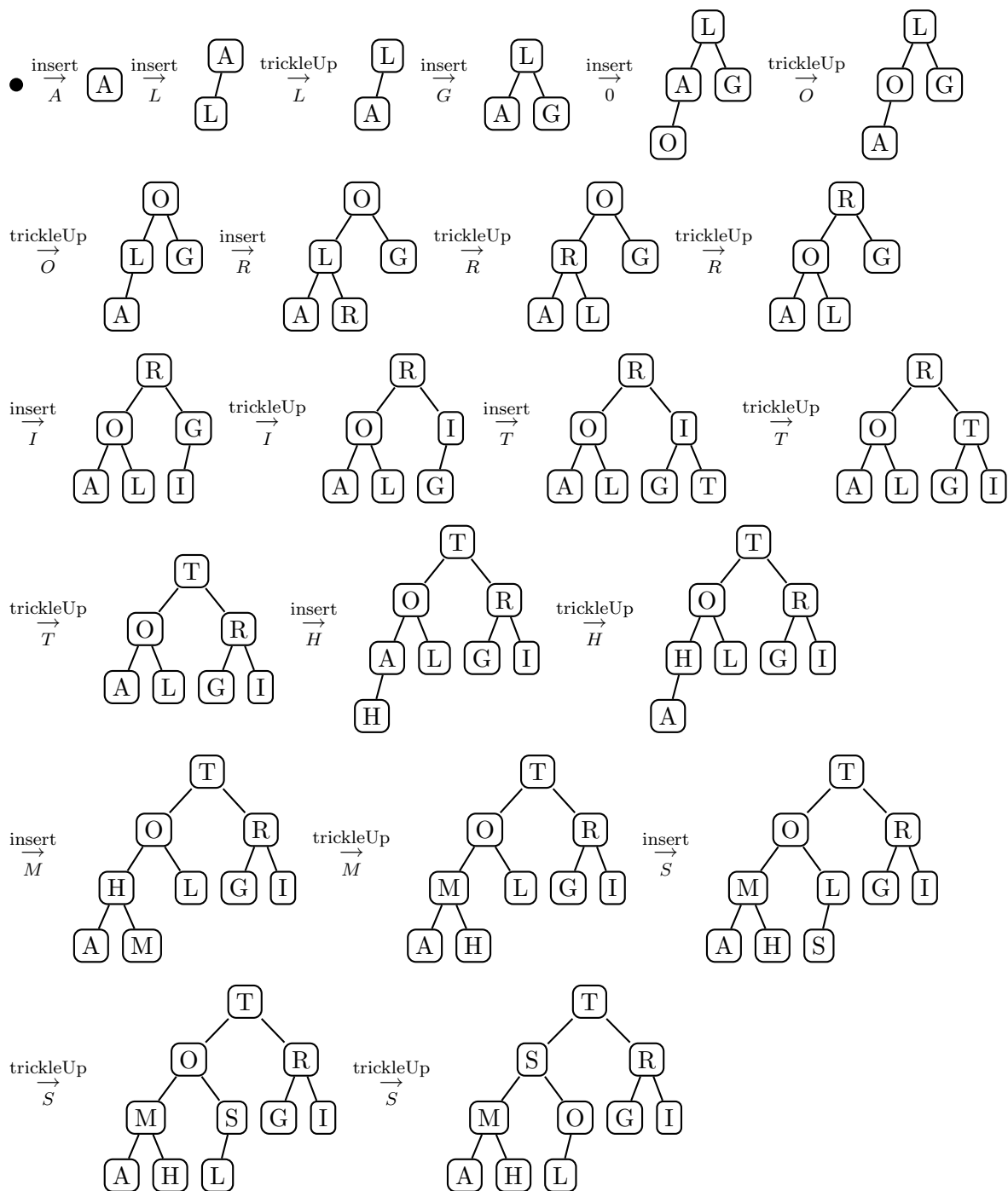
We can enqueue an element x into a complete heap in time proportional to its height by:

1. inserting x in the next “free” complete tree slot. This maintains completeness of the tree but can violate the heap condition.
2. “trickling x up” towards the root until the heap condition is restored. The result is a complete heap with one more element.

Step #1 takes constant time and step #2 takes worst-case time proportional to the height of the tree, which is $\Theta(\log(n))$.

Complete Heap Enqueuing Example

Enqueue the letters A L G O R I T H M S into an initially empty complete max heap.



Java Code for Complete Heap Enqueuing

Here is Java code for enqueuing into a complete heap represented as a vector instance variable `elts` and comparator `comp`.

```
public void enq (T x) {
    elts.add(x); // Add x at next binary address
    trickleUp(elts.size() - 1); // Trickle up from last binary address
}

private void trickleUp (int addr) {
    while ((addr > 0) && greaterThan(addr, paddr(addr))) {
        swap(addr, paddr(addr));
        addr = paddr(addr);
    }
}

private boolean greaterThan (int addr1, int addr2) {
    return compare(elts.get(addr1), elts.get(addr2)) > 0;
}

private void swap (int addr1, int addr2) {
    T temp = elts.get(addr1);
    elts.set(addr1, elts.get(addr2));
    elts.set(addr2, temp);
}
```

Complete Heap Dequeuing

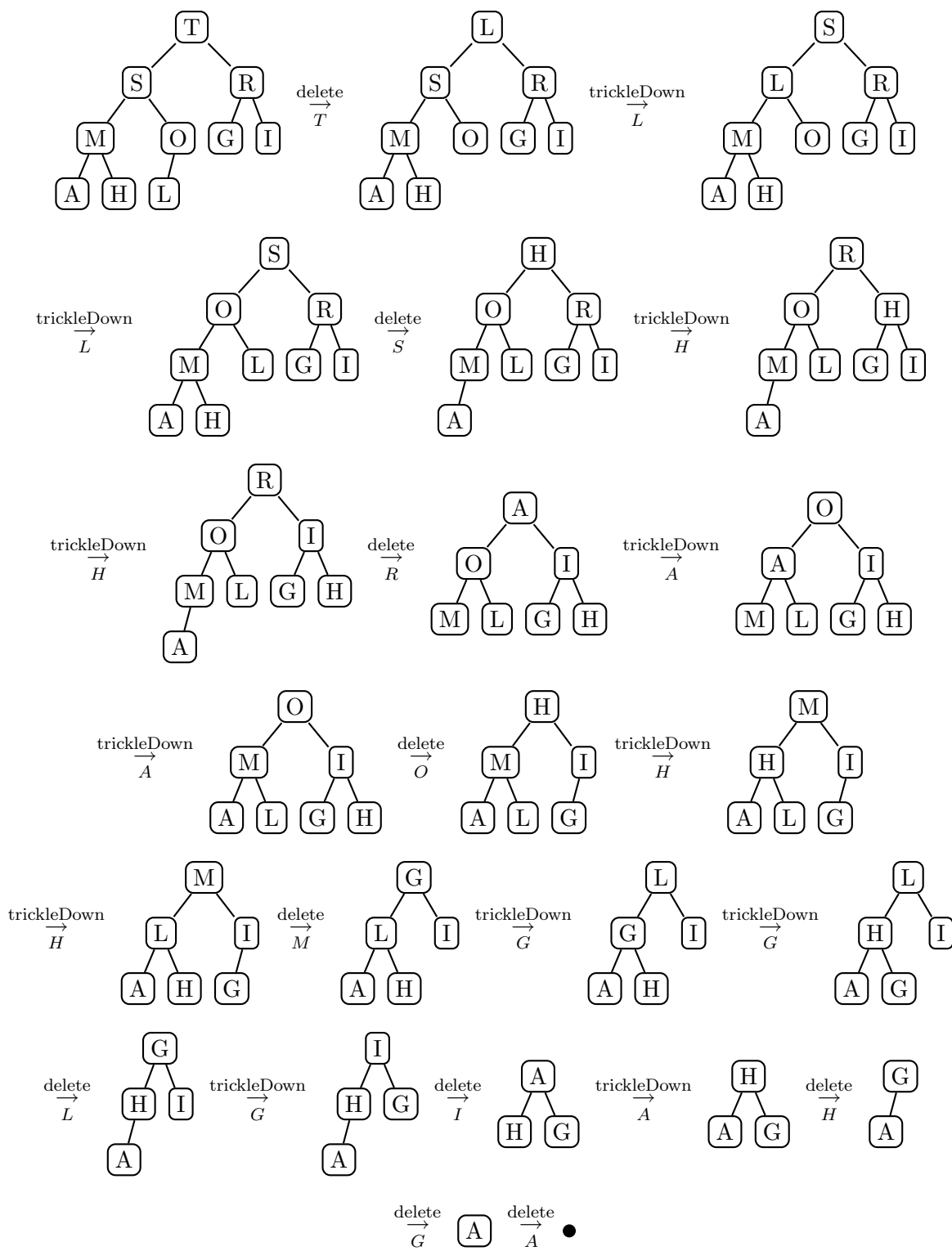
We can dequeue the next element from a complete heap in time proportional to its height by:

1. remembering the top node value `first` (which will be returned later);
2. moving the value `last` from the last complete tree slot into the top node. This maintains completeness of the tree but can violate the heap condition;
3. “trickling `last` down” by swapping it with the larger of its two children (if one is greater than `last`) until the heap condition is restored. The result is a complete heap with one less element;
4. returning the remembered value `first`.

Steps #1, #2, #4 take constant time and step #3 takes worst-case time proportional to the height of the tree, which is $\Theta(\log(n))$.

Complete Heap Dequeuing Example

We will use the dequeuing algorithm specified above to dequeue all the elements from the complete heap that resulted from the enqueueing process:



Java Code for Complete Heap Dequeuing

Here is Java code for dequeuing from a complete heap represented as a vector instance variable `elts` and comparator `comp`.

```
public T deq () {
    if (elts.size() == 0) {
        throw new RuntimeException("Attempt to deq() empty queue.");
    } else {
        T first = elts.get(0); // Remember top value in heap
        T last = elts.remove(elts.size() - 1); // Name last value in heap
        if (elts.size() > 0) {
            elts.set(0, last); // Move last value to top of heap
            trickleDown(0); // Trickle down from top of heap
        }
        return first;
    }
}

private void trickleDown (int addr) {
    int largest = addrOfLargest(laddr(addr), addrOfLargest(raddr(addr), addr));
    if (largest != addr) {
        swap(addr, largest);
        trickleDown(largest);
    }
}

// Given a known-valid complete heap index addr2 and a possibly invalid
// index addr1, returns the index associated with the largest value
// (if addr1 is valid) or addr2 (if addr1 is invalid).
private int addrOfLargest(int addr1, int addr2) {
    if ((addr1 < elts.size()) && greaterThan(addr1, addr2)) {
        return addr1;
    } else {
        return addr2;
    }
}
```

Building a Complete Heap From a Vector

It is often necessary to build a complete heap from a given collection of objects, such as a vector, array, list, etc. For concreteness, we will focus on building a complete heap from a vector. What's an efficient way to do this?

A $\Theta(n \cdot \log(n))$ approach

An obvious approach is to simply enqueue all the elements from the vector one-by-one into an initially empty complete heap. Here's the code for such an approach:

```
public static <T> PQueueCompleteHeap <T> fromVectorInefficient (Vector<T> v) {
    PQueueCompleteHeap<T> pq = new PQueueCompleteHeap<T>();
    for (int i = 0; i < v.size(); i++) {
        pq.enq(v.get(i));
    }
    return pq;
}
```

You should convince yourself that this takes time $\Theta(n \cdot \log(n))$ for an n -element vector.

A $\Theta(n)$ approach

An alternative approach is to use a copy of the given vector as the vector representing the complete heap, and to “trickle down” elements starting at the next to last row of the heap and working up to the top row of the heap. This ordering guarantees that by the time the trickling-down process is invoked at a node, both subtrees are already guaranteed to be heaps. So after the final “trickle down” is performed at the root of the tree, the tree is guaranteed to be a heap.

Here is the code for this approach:

```
public static <T> PQueueCompleteHeap<T> fromVector (Vector<T> v, Comparator<T> comp) {
    PQueueCompleteHeap<T> pq = new PQueueCompleteHeap<T>(comp);
    pq.elts = (Vector<T>) v.clone();
    pq.buildHeap();
    return pq;
}

private void buildHeap () {
    // Note: (v.size() - 2) is the largest index of an
    // element in the next to last level of the tree.
    for (int i = ((elts.size() - 2) / 2); i >= 0; i--) {
        trickleDown(i);
    }
}
```

It turns out that the running time of this algorithm is $\Theta(n)$, which is asymptotically faster than the “obvious” algorithm explored above. See CS231 for the details.

Sometimes we want to re-use the supplied vector for holding the elements of the priority queue:

```
public static <T> PQueueCompleteHeap<T>
    fromVectorShared (Vector<T> v, Comparator<T> comp) {
    PQueueCompleteHeap<T> pq = new PQueueCompleteHeap<T>(comp);
    pq.elts = v;
    pq.buildHeap();
    return pq;
}
```

Heapsort Revisited

When using `heapSort` to sort a vector `v`, it is possible to use `v` itself for the elements of the complete heap rather than using a *copy* of `v`. This leads to a version of `heapSort` that is *in-place*, which means it uses only constant extra storage space in addition to the vector being sorted.

```
public static <T> void heapSortEfficient (Vector<T> v, Comparator<T> comp) {
    PQueueCompleteHeap<T> pq = fromVectorShared(v, comp);
    for (int i = v.size()-1; i >= 0; i--) {
        v.set(i, pq.deq());
    }
}
```

This idea will only work with max complete heaps. Why won't it work for min complete heaps?