Heaps

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

June 3, 2017

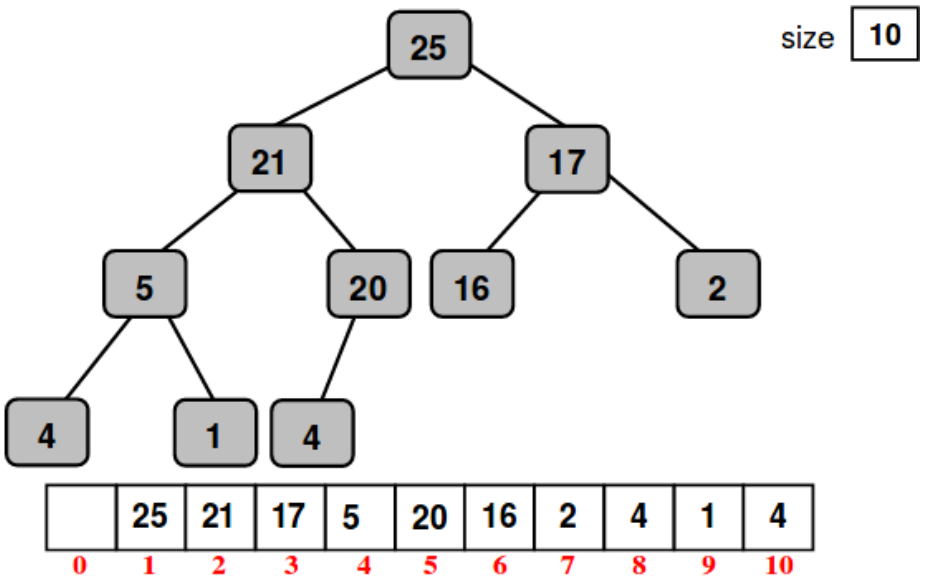
Overview I

Intro

- It is a complete binary tree.
- Value at each node in a (max) heap is at least as large as the value at its children nodes.
- Let X be a totally ordered set. A heap on X is either empty or it is a complete binary tree, t, comprising $n_t \geq 1$ nodes to each node of which a value of X is assignes such that:
  value of node $i \leq$ value of parent of node $i, i = 2, 3, ...n_t$.
- The size of a heap is the number of nodes in the tree.

Array implementation

- The root is stored in `heapArray[1]`.
- If the value of a node, s, is stored in `heapArray[`i`]`, then the value of s is stored in `heapArray[`$2 \times i$`]` and the value of the right child of s is stored in `heapArray[`$2 \times i + 1$`]`.



Sift up

```
void siftUp (Comparable [] a, int k, int m) {
// Value just added
   Comparable v = a[k];
// Index to last element
   int j = k;
// Index to parent of added element
   int i = j/2;
// While parent is greater than the element added
   while (i > 0 && a[i].compareTo(v) < 0) {
// Swap elemets around
      a[j] = a[i];
      j = i;
      i = j/2;
   }
   a[j] = v;
}
```

Sift Down

```
void siftDown (Comparable [] a, int k, int m) {
// m is the size of the heap
/ Start at 1
  Comparable v = a[k];
  int max_index;
  Comparable maxCh;
// i starts at 1
  int i = k;
// Determine that the node has at least on child
  boolean more = m >= 2*k;
  while(more) {
// Get position of the child that's bigger
    max_index = maxChild(a,i,m);
    maxCh = a[max_index];
// If the item is smaller than the max child, swap, otherwise stop.
    if (v.compareTo(maxCh) < 0) {
// Swap
      a[i] = maxCh;
      i = max_index;
// Determine that the node has children
      more = m >= 2*i;
    }
    else
      more = false
  }
  a[i] = v;
}
```

Methods

- delete:

```
void deleteMax() {
// Replace with last element added, reduce size
  heapArray[1] = heapArray[size--];
// Sift that element down to replace max to the top
  siftDown(heapArray,1,size);
}
```

- insert:

```
void insert(Comparable item) {
  if(size == maxSize)
    // Double array size
// increase size and add to the end of the array
  heapArray[++size] = item;
// Sift up to restore heap quality
  siftUp(heapArray,size,size);
}
```

Heapify

- This is taking an array and turning it into a heap
- $O(nlogn)$:
  - Simply add all of the array elements to the `heapArr` and then iterate through calling `siftUp()` on all of them elements.
- $O(n)$:
  - Add all of the array elements to the `heapArr` with height of underlying tree being l.
  - Make each of the subtrees whose roots are at level $l - 1$ into a heap, using `siftDown()`.
  - Repeat this pocess for every subtree rooted at each previous level down to and including 0.

Heap Sort

- To sort n values using `HeapSort`:
  1. Construct a heap from the n values using the $O(n)$ heapify method.
  2. Output the value at the top (largest value)
  3. Then `deleteMax()` to remove it.
  4. Repeat step 2 until all are outputted.
- Run time complexity is:

$$c_1 n + \sum_{i=1}^{n} (c_2 + c_3 logi)$$

Which is $O(nlogn)$

The End