Heapsort

Based off slides by: David Matuszek
http://www.cis.upenn.edu/~matuszek/cit594-2008/

Presented by: Matt Boggus

Previous sorting algorithms

- Insertion Sort
  - $O(n^2)$ time
- Merge Sort
  - $O(n)$ space

Heap data structure

- Binary tree
- Balanced
- Left-justified

(Max) Heap property: no node has a value greater than the value in its parent

Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
  - A binary tree of depth $n$ is balanced if all the nodes at depths 0 through $n-2$ have two children
Left-justified binary trees

- A balanced binary tree of depth \( n \) is left-justified if:
  - it has \( 2^n \) nodes at depth \( n \) (the tree is “full”), or
  - it has \( 2^k \) nodes at depth \( k \), for all \( k < n \), and all the leaves at depth \( n \) are as far left as possible

![Left-justified tree vs. not left-justified tree](image)

Building up to heap sort

- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children

![Examples of heap properties](image)

- All leaf nodes automatically have the heap property
- A binary tree is a heap if all nodes in it have the heap property

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child

![Example of siftUp](image)

- This is sometimes called sifting up
A tree consisting of a single node is automatically a heap.

We construct a heap by adding nodes one at a time:

- Add the node just to the right of the rightmost node in the deepest level.
- If the deepest level is full, start a new level.

Examples:

- Each time we add a node, we may destroy the heap property of its parent node.
- To fix this, we sift up.
- But each time we sift up, the value of the topmost node in the sift may increase, and this may destroy the heap property of its parent node.
- We repeat the sifting up process, moving up in the tree, until either:
  - We reach nodes whose values don’t need to be swapped (because the parent is still larger than both children), or
  - We reach the root.

Other children are not affected.

- The node containing 8 is not affected because its parent gets larger, not smaller.
- The node containing 5 is not affected because its parent gets larger, not smaller.
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally.
A sample heap

- Here’s a sample binary tree after it has been heapified

- Notice that heapified does not mean sorted
- Heapifying does not change the shape of the binary tree; this binary tree is balanced and left-justified because it started out that way

Removing the root (animated)

- Notice that the largest number is now in the root
- Suppose we discard the root:
  - How can we fix the binary tree so it is once again balanced and left-justified?
  - Solution: remove the rightmost leaf at the deepest level and use it for the new root

The reHeap method I

- Our tree is balanced and left-justified, but no longer a heap
- However, only the root lacks the heap property

- We can siftUp() the root
- After doing this, one and only one of its children may have lost the heap property

The reHeap method II

- Now the left child of the root (still the number 11) lacks the heap property

- We can siftUp() this node
- After doing this, one and only one of its children may have lost the heap property
The reHeap method III

- Now the right child of the left child of the root (still the number 11) lacks the heap property:

```
  22
 /   \
19   11
 /   /   \
18  21  3
```

- We can `siftUp()` this node
- After doing this, one and only one of its children may have lost the heap property — but it doesn’t, because it’s a leaf

The reHeap method IV

- Our tree is once again a heap, because every node in it has the heap property

```
  22
 /   \
19   17
 /   /   \
18  14  21
```

- Once again, the largest (or a largest) value is in the root
- We can repeat this process until the tree becomes empty
- This produces a sequence of values in order largest to smallest

Sorting

- What do heaps have to do with sorting an array?
- Here’s the neat part:
  - Because the binary tree is balanced and left justified, it can be represented as an array
  - *Danger Will Robinson:* This representation works well only with balanced, left-justified binary trees
  - All our operations on binary trees can be represented as operations on arrays
  - To sort:
    - heapify the array;
    - while the array isn’t empty {
      remove and replace the root;
      reheap the new root node;
    }

Key properties

- Determining location of root and “last node” take constant time
- Remove n elements, re-heap each time
To reheap the root node, we have to follow one path from the root to a leaf node (and we might stop before we reach a leaf).

- The binary tree is perfectly balanced.
- Therefore, this path is $O(\log n)$ long.
- And we only do $O(1)$ operations at each node.
- Therefore, reheap takes $O(\log n)$ times.
- Since we reheap inside a while loop that we do $n$ times, the total time for the while loop is $n*O(\log n)$, or $O(n \log n)$.

Construct the heap $O(n \log n)$

Remove and re-heap $O(n \log n)$

Total time $O(n \log n) + O(n \log n)$

Continue to priority queues?

Queue – only access element in front

Queue elements sorted by order of importance

Implement as a heap where nodes store priority values.
Extract Max

- Remove root
- Swap with last node
- Re-heapify

Increase Key

- Change node value
- Re-heapify

Insert

- Add new node, priority is minimum possible value
- Increase priority

The End