Today …

- Quiz 7
- Assignments
  - Assignment 7 due Thursday!
- Note on passing pointers by-reference
- Heapsort [Sect. 11.2]
- Heapsort exercise
Heapsort

• The basic idea:
  – Build a heap out of the unsorted list
  – Create a temporary array
  – Find max value and delete
  – Put max value in last position of temp array
  – Find next max value and delete
  – Put next max value in the next to last position of temp array
  – Repeat until the heap is empty!

• What is the cost of doing this?
  – $O(n \log n)$ for building the heap (repeated insertions)
  – $O(n \log n)$ for repeatedly deleting elements
  – So, $O(n \log n)$
Heapsort

• **Building the Heap**
  
  – Instead of building a Heap by repeated insertions …
  
  – We can improve efficiency by:
    
    • Treating the input array as a heap
    
    • Incrementally rebuild the heap “bottom up”
    
    • Starting at the (approx.) last node of level \( h - 1 \) (i.e., element \( n/2 \) )

Heapsort

• Building the Heap more efficiently

  – Recall that a “Semiheap” is a tree whose subtrees are heaps but the root node may be out of place

  – We can rebuild a semiheap via “trickle down”

  – Here we assume a variant to heapRebuild:

    ```
    // rebuild semiheap stored in array at index
    void HeapRebuild(int theArray[], int size, int index);
    ```
Heapsort

• Building the Heap more efficiently

array = [6 3 5 9 2 10]

HeapRebuild(array, 6, 2);

array = [6 3 10 9 2 5]

Heapsort

• Building the Heap more efficiently

array = [6 3 10 9 2 5]

HeapRebuild(array, 6, 1);

array = [6 9 10 3 2 5]
Heapsort

• Building the Heap more efficiently

// building the heap
for(int i = (size-1)/2; i >= 0; i--)
    HeapRebuild(array, size, i);
Heapsort

• Sorting from the Heap more efficiently
  – Notice in the simple approach we use a temporary array
  – We can avoid this by performing heapsort “in place”
  – After each deletion, maintain two partitions of the array:
    • The sorted region (the end of the array)
    • The Heap region (the beginning of the array)

Array-Based Heap Implementation

```c
void heapRebuild(int theArray[], int size, int index) {
    int child = 2*index + 1;
    if(child < size) {
        int right = child + 1;
        if(right < size && theArray[right] > theArray[child])
            child = right;
        if(theArray[index] < theArray[child]) {
            swap(theArray[index], theArray[child]);
            heapRebuild(theArray, size, child);
        }
    }
}
```
void heapsort(int theArray[], int size) {
    for(int i = (size-1)/2; i >= 0; i--)  // build
        heapRebuild(theArray, size, i);
    int last = size-1;
    for(int j = 1; j < size; j++) {        // delete
        swap(theArray[0], theArray[last]);
        heapRebuild(theArray, last, 0);
        last--;
    }
}

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
<td>$O(n\log n)$</td>
</tr>
</tbody>
</table>