The Heap

Topics for this week:
- Heap
  - a new data structure
  - can be used to implement a priority queue, i.e. can efficiently support these operations:
    - insert, removeMin (and update)
- HeapSort - another $O(n \log n)$ sort, non-recursive

Announcements:
- lab from week 8 due Fri, Feb 12
- projects: this week’s lab, final submission and regrade
Problem: A Traffic Simulation

We are studying traffic patterns: our simulation software simulates the trajectories of individual cars and measures the quantities of interest (congestion, speed, etc). One part of the problem studies toll booths and in order for the simulation to proceed, we need to be able to figure out which car arrives at the toll booth next.

Event queue: 

Need to be able to:
- insert (a new car with its estimated time of arrival)
- removeMin (which car to process next at the toll booth?)
Priority Queue

A data structure that supports:
- insert a new element (with its associate value)
- removeMin, returns the element with the smallest value
- possibly updateValue of an existing element

Let's look at previous implementation approaches:

- unsorted list
  - insert: $O(1)$
  - removeMin: $O(n)$

- sorted list, resort after insert
  - insert: $O(n \log n)$ or $O(n^3)$
  - removeMin: $O(1)$

- sorted list, insert at proper position
  - insert: $O(n)$
  - removeMin: $O(1)$
Priority Queue via the Heap

The heap:
- a balanced binary tree
- every parent’s value smaller ($\leq$) than both children’s values

Example: a heap for arriving times 3, 6, 5, 9, 7, 4, 1?
How to **insert** to a heap?

- place the element as the last element in the last row (or start a new row if the last row is already full)

- then what?

Pseudo code:

```python
def insert(Heap H, a new elem: x):
    # Create a new node with value x, as the last element in the last row (or start a new row if the last row is full)
    create a new node with value x
    let nodeX be the new node
    while nodeX is not the root and nodeX's parent > nodeX:
        swap the values of nodeX and nodeX's parent
        nodeX = nodeX's parent
    return new Heap (root)
```
Priority Queue via the Heap

How to `removeMin` from a (min-)heap?

Pseudo code:

```python
def removeMin(Heap H):
    y = the value in the last node in the last level
    remove the node with y (delete it)
    minVal = the value of the root
    y = the value of the root
    node? be the root
    while node? has at least one child and
        node? has at least one child and
        node? or both of its children:
        smallChild be the smaller child of node?
        switch the values of node? and smallChild
        node? = smallChild
    return minVal and the updated heap
```

Time complexity: $O(\log n)$, bubbling down through at most the height of the heap, notice some complexity for insert (bubbling up instead of down)
Implementing Heaps

As a linked data structure:

- A Node class contains attributes:
  - value of the node
  - left child node
  - right child node
  - parent node

A Heap class:
- the root node
- # elements (to be able to quickly find the last node at the last level)

Via arrays:

Python lists

read the heap level by level ⇒ get an array of all the values

```
indices: 0 1 2 3 4 5 6 7 8
```

```
def parent(index i):
    return \[(i-1)\] / 2  (integer division)

def left_child(index i):
    return \[2i+1\]

def right_child(index i):
    return \[2i+2\]
```

check the posted implementation
(arrays much easier to implement than using the Node class)

indices as a tree:
Heap Sort

How to sort with heaps?

3, 1, 7, 8, 5, 10, 6

Idea: insert the elements into a heap (n inserts)
  removeMin n times, creating a list
  will be sorted - why?

Time complexity:
  n inserts (creating a heap - often called "heapify")
  n removeMin

O(nlog n) + O(nlog n) = O(nlog n)
  (twice nlog n, O() "removes" constants)

Remark:
  can be also done in O(n)
  but O(nlog n) is enough
  since the removals will need O(nlog n)
  anyway
Does a heap help?

- recall that the Dijkstra’s shortest path algorithm needs:
  - $n$ inserts
  - $n$ removeMins
  - up to $m$ updateValues

(where $n = \#$ vertices, $m = \#$ edges)