CS 240 — Lecture 4

Bartosz Antczak *Instructor: Eric Schost* *January 12, 2017*

Today's plan

- Wrap up loop analysis
- Look at the merge sort algorithm
- Introduce ourselves to heaps

4.1 Wrapping up loop analysis

Example 4.1.1. *Consider the following algorithm.*

```
sum = 0
for (i = 1; i < n)
  j = i
  while (j >= 1)
    sum = sum + (i / j)
    j = floor(j/2)
return sum
```

For an integer n, let's call $b(n)$ the number of *digits* of n written in base 2 (e.g., $n = 19 = [10011]_2$, thus $b(n) = 5$).
We claim that $\log_2(n) \leq b(n) \leq \log_2(n) + 2$ for all n. The proof is shown:
Proof:

$$\lfloor \log_2(n) \rfloor \leq \log_2(n) \leq \lceil \log_2(n) \rceil \tag{4.1}$$
$$2^{\lfloor \log_2(n) \rfloor} \leq n \leq 2^{\lceil \log_2(n) \rceil} \tag{4.2}$$

Now note that $\lfloor \log_2(n) \rfloor + 1 \leq b(2^{\lfloor \log_2(n) \rfloor})$ and $b(2^{\lceil \log_2(n) \rceil}) \leq \lceil \log_2(n) \rceil + 1$

$$\lfloor \log_2(n) \rfloor + 1 \leq b(n) \leq \lceil \log_2(n) \rceil + 1 \tag{4.3}$$
$$(\log_2(n) - 1) + 1 \leq b(n) \leq (\log_2(n) + 1) + 1 \tag{4.4}$$
$$\log_2(n) \leq b(n) \leq \log_2(n) + 2 \tag{4.5}$$

Now let's analyze the algorithm. For $i \geq 1$, let $u_3(i)$ be the number of times we enter the inner loop starting with $j = i$. Let $t_3(n)$ be the total runtime of the algorithm. Then $t_3(n) = \theta\left(\sum_{i=1}^{n} u_3(i)\right)$
By observation, we see that for each pass in the inner loop of the algorithm, it erases the last digit of j in base 2. For instance, if $j = 18 = [10010]_2$, we see that $\lfloor \frac{j}{2} \rfloor = 9 = [1001]_2$.
Thus, $u_3(i) =$ the number of bits of i written in base $2 = b(i)$ (which is why we defined it previously).

Let's calculate it now:

$$t_3(n) = \theta\left(\sum_{i=1}^{n} u_3(i)\right) = \sum_{i=1}^{n} b(i) \tag{4.6}$$

$$\log_2(i) \leq b(i) \leq \log_2(i) + 2 \tag{4.7}$$

$$\sum_{i=1}^{n} \log_2(i) \leq \sum_{i=1}^{n} b(i) \leq \sum_{i=1}^{n} \log_2(i) + \sum_{i=1}^{n} 2 \tag{4.8}$$

Now referring to our cheat sheet on module 1, slide 38:

$$\sum_{i=1}^{n} \log_2(i) = \log_2\left(\prod_{i=1}^{n} i\right) = \log_2(n!) \in \theta(n \log(n))$$

Therefore, $t_3(n) = \sum_{i=1}^{n} b(i) \in \theta(n \log_2(n))$.

4.2 Design of MergeSort

The recursive algorithm of mergesort receives an input an array A of n integers. The algorithm proceeds as follows:

1. Split A into two smaller arrays. Both arrays should contain the same number of elements or should differ by at most one (aka, split the array evenly or have at most one extra element in one subarray)
2. Recursively run MergeSort on both subarrays
3. After both subarrays have been sorted, use a function `Merge()` to merge them into a single sorted array

Mergesort has a runtime of $\theta(n)$. The simplified algorithm structure of mergesort (`A, S, aL, aR` are arrays, and `n, nL, nR` are integer values):

```
MergeSort(A, n) {
  if (n == 1) S = A
  else {
    nL = ceiling(n/2) // size of subarray 1
    nR = floor(n/2) // size of subarray 2
    aL = {A[1], ..., A[nL]} // subarray 1
    aR = {A[nL + 1], ..., A[n]} //subarray 2
    sL = MergeSort(aL, nL) // the sorted subarray 1
    sR = MergeSort(aR, nR) // the sorted subarray 2
    S = Merge(sL, nL, sR, nR) // The entire sorted array
  }
  return S
}
```

The

— And this concludes Module 1: Order Notation —.

4.3 Abstract Data Types

An **Abstract Data Type (ADT)** is a description of information and a collection of operations on that information. The information is accessed only through the operations. We have various realizations (methods of implementation) of an ADT, which specify

- How the information is stored (Data structure)
- How operations are performed (algorithms)

4.3.1 Priority Queue ADT

A **priority queue** is an ADT consisting of a collection of items, where each item has a priority. An typical application of this is a "todo" list. Two operations in a Priority Queue are

- `insert`: insert an item tagged with a priority
- `deleteMax`: remove the item with the highest priority

Possible Implementations of Priority Queues

1. Use an **unsorted array**. We see that the two operations have a runtime of
   - `insert`: $O(1)$
   - `deleteMax`: $O(n)$
2. Use a **sorted array**. We see that the two operations have a runtime of
   - `insert`: $O(n)$
   - `deleteMax`: $O(1)$
3. Use a *heap* (defined in next subsection)

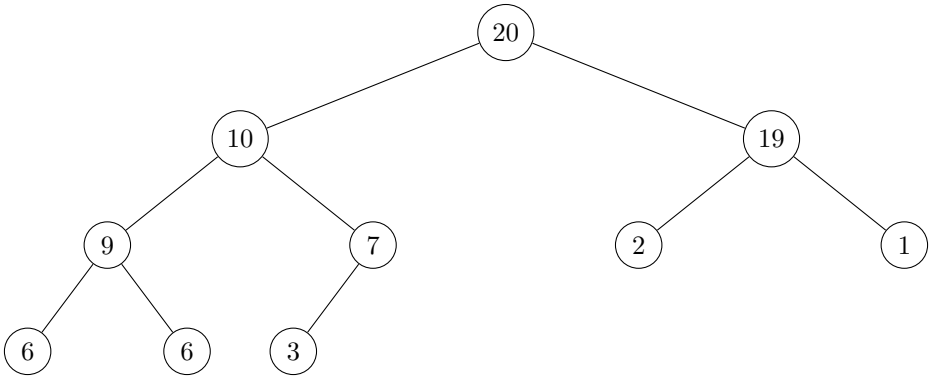
4.3.2 Heaps

A **heap** is a certain type of binary tree (recall that a binary tree is either empty or consists of three parts: a node, a left binary subtree and a right binary subtree).
Definition: a **max-heap** is a binary tree with the following two properties:

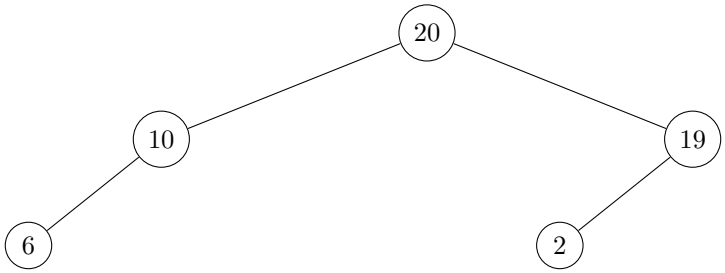
- **Structural Property**: all levels of a heap are completely filled, except maybe the last level. The filled items in the last level are *left-justified* (for every node, their child on the left has their nodes filled with both children)
- **Heap-order Property**: For any node i, the value of the parent of i is larger than or equal to the value of i

Every node contains a value (or key).

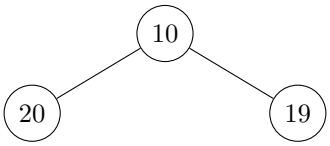
Example 4.3.1. *Various trees that are and aren't heaps.*



Tree # 1: A proper heap



Tree # 2: Not a heap (it's not left justified)



Tree # 3: Not a heap (it doesn't follow the heap-order property)

Insertion in Heaps

To insert a new node in a heap, we first place the new node at the first free leaf. However, this may violate the heap-order property, so we perform a *bubble-up*.
A **bubble-up** is an algorithm that operates in the following way:

- While the parent of the new node exists and the value of the parent is less than the value of the new node, swap both nodes
- The new node is now in its parent's position. Now repeat this algorithm

This algorithm has a runtime of $O(\log n)$

Deleting in Heaps

To delete the maximum item (node with the largest value), we replace the root with the last leaf (the rightmost child leaf in the lowest level of the tree). Again, this may violate the heap-order property, so we perform a *bubble-down*. A **bubble-down** is an algorithm that operates in the following way:

- While the replaced node is not a leaf, find the child node which has the largest value. If the value of the child is greater than the value of the replaced node, swap them. Otherwise, you're done.
- If you swapped them, the replaced node is now in the position of its child node and the child node is now the parent node
- Repeat this algorithm on the replaced node in its new position

This algorithm has a runtime of $O(\log n)$.