COMP20007 Design of Algorithms

Sorting - Part 3

Daniel Beck

Lecture 13

Semester 1, 2020

Priority Queues

- A *set* of elements, each one with a *priority key*.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.
- Used as part of an algorithm (ex: Dijkstra)...

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.
- Used as part of an algorithm (ex: Dijkstra)...
- ...or on its own (ex: OS job scheduling).

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.
- If using an *unsorted array/list*, we obtain Selection Sort.

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.
- If using an *unsorted array/list*, we obtain Selection Sort.
- If using an *heap*, we obtain Heapsort.

The Heap

It's a tree with a set properties:

The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)

The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)

The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)
- Complete (all levels are full except for the last, where only rightmost leaves can be missing) (implies Balanced)

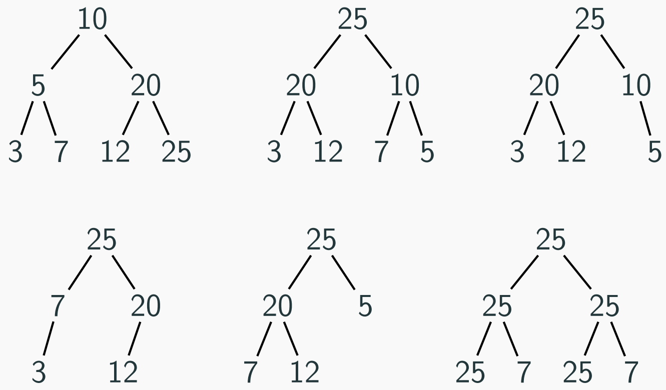
The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)
- Complete (all levels are full except for the last, where only rightmost leaves can be missing) (implies Balanced)
- Parental dominance (the key of a parent node is always higher than the key of its children)

Heaps and Non-Heaps

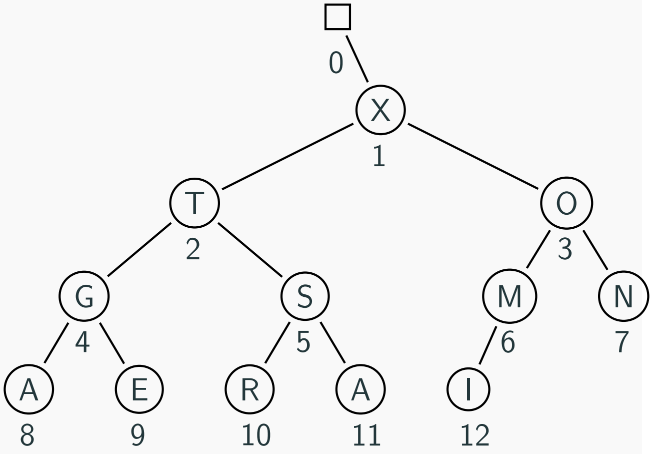
Which of these are heaps?



Heaps as Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array A.

Note that the children of node i will be nodes $2i$ and $2i + 1$.



A:

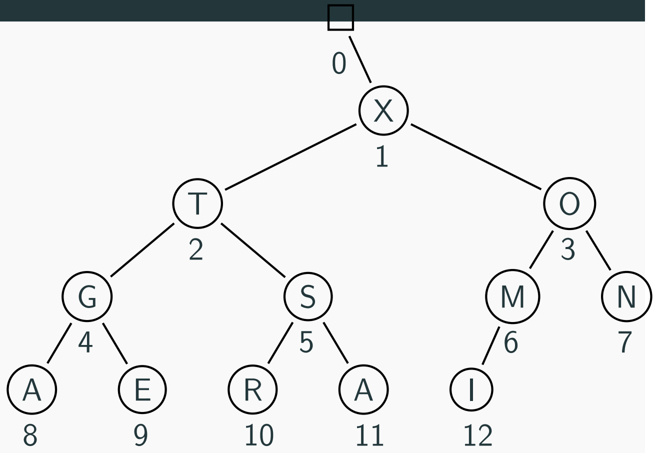
0	1	2	3	4	5	6	7	8	9	10	11	12
	X	T	O	G	S	M	N	A	E	R	A	I

Heaps as Arrays

This way, the heap condition is simple:

$\forall i \in \{0, 1, \ldots, n\}$, we must have $A[i] \leq A[\lfloor i/2 \rfloor]$.

$A[i] \geq A[2i]$
$A[i] \geq A[2i+1]$



A:

0	1	2	3	4	5	6	7	8	9	10	11	12
	X	T	O	G	S	M	N	A	E	R	A	I

Heapsort

function $\textsc{Heapsort}(A[1..n])$ $\triangleright$ Assume $A[0]$ as a sentinel

    $\textsc{Heapify}(A[1..n])$

    for $i \leftarrow n$ **to** 0 **do**

        $\textsc{Eject}(A[1..i])$

Heapify

function $\textsc{BottomUpHeapify}(A[1..n])$

  for $i \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**

    $k \leftarrow i$

    $v \leftarrow A[k]$

    $heap \leftarrow False$

    while not $heap$ **and** $2 \times k \leq n$ **do**

      $j \leftarrow 2 \times k$

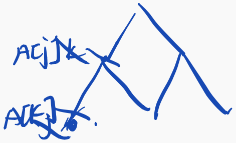
      if $j < n$ **then** ▷ two children

        if $A[j] < A[j+1]$ **then** $j \leftarrow j + 1$

      if $v \geq A[j]$ **then** $heap \leftarrow True$

      else $A[k] \leftarrow A[j]$; $k \leftarrow j$

    $A[k] \leftarrow v$

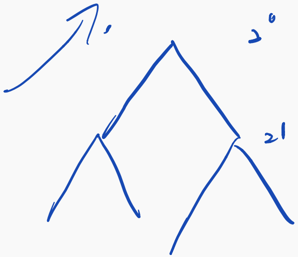


Analysis of Bottom-Up Heapify

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

$2^0 + 2^1 + 2^{\cdots} + 2^h$

$\frac{(1-2^{h+1})}{1-2}$



Analysis of Bottom-Up Heapify

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

Here is an upper bound on the number of "down-sifts" needed

$$\sum_{i=0}^{h-1} \sum_{\text{nodes at level } i} 2(h-i) \quad = \quad \sum_{i=0}^{h-1} 2(h-i)2^i \quad = \quad 2(n-\log_2(n+1))$$

The last equation can be proved by mathematical induction.

Analysis of Bottom-Up Heapify

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

Here is an upper bound on the number of "down-sifts" needed

$$\sum_{i=0}^{h-1} \sum_{\text{nodes at level } i} 2(h-i) \quad = \quad \sum_{i=0}^{h-1} 2(h-i)2^i \quad = \quad 2(n - \log_2(n+1))$$

The last equation can be proved by mathematical induction.

Note that $2(n - \log_2(n+1)) \in \Theta(n)$, hence we have a linear-time algorithm for heap creation.

Eject

```
function EJECT(A[1..i])
    SWAP(A[i], A[1])
    k ← 1
    v ← A[k]
    heap ← False
    while not heap and 2 × k ≤ i − 1 do
        j ← 2 × k
        if j < i − 1 then                              ▷ two children
            if A[j] < A[j + 1] then j ← j + 1
        if v ≥ A[j] then heap ← True
        elseA[k] ← A[j]; k ← j
    A[k] ← v
```

Heapsort - Properties

Transform-and-Conquer paradigm

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

- In-place?

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

- In-place?
- Stable?

Heapsort - Properties

- In-place?

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.
- Stable?

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.
- Stable? **No.** Non-local swaps break stability.

Heapsort - Complexity

- We already saw in the videos that the complexity of Heapsort in the worst case is $\Theta(n \log n)$.

Heapsort - Complexity

- We already saw in the videos that the complexity of Heapsort in the worst case is $\Theta(n \log n)$.
- What's the complexity in the best case?

Heapsort - Complexity

- Heapify is $\Theta(n)$ in the worst case.

Heapsort - Complexity

- Heapify is $\Theta(n)$ in the worst case.
- Eject is $\Theta(\log n)$ in the worst case.

Heapsort - Complexity

- Heapify is $\Theta(n)$ in the worst case.
- Eject is $\Theta(\log n)$ in the worst case.
- In the worst case, heapsort is $\Theta(n) + n \times \Theta(\log n) \in \Theta(n \log n)$.

Heapsort - Complexity (best case)

Heapsort - In practice

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.
- Used in the Linux kernel.

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.
- Used in the Linux kernel.

Take-home message: Heapsort is the best choice when low-memory footprint is required and guaranteed $\Theta(n \log n)$ performance is needed (for example, security reasons).

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.
- Quicksort: best for more general cases and large amounts of data. Not stable.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.
- Quicksort: best for more general cases and large amounts of data. Not stable.
- Heapsort: slower in practice but low-memory and guaranteed $\Theta(n \log n)$ performance.

Sorting - Paradigm Summary

- Selection Sort: *brute force*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

There are all *comparison-based* sorting algorithms.

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

There are all *comparison-based* sorting algorithms.

Next lecture: Ok, my data is sorted. Now how do I keep it sorted?