# Algorithm and Data Structure Analysis (ADSA)

## Lecture 9: Priority Queues/Heapsort

# Heap Operations: BuildHeap

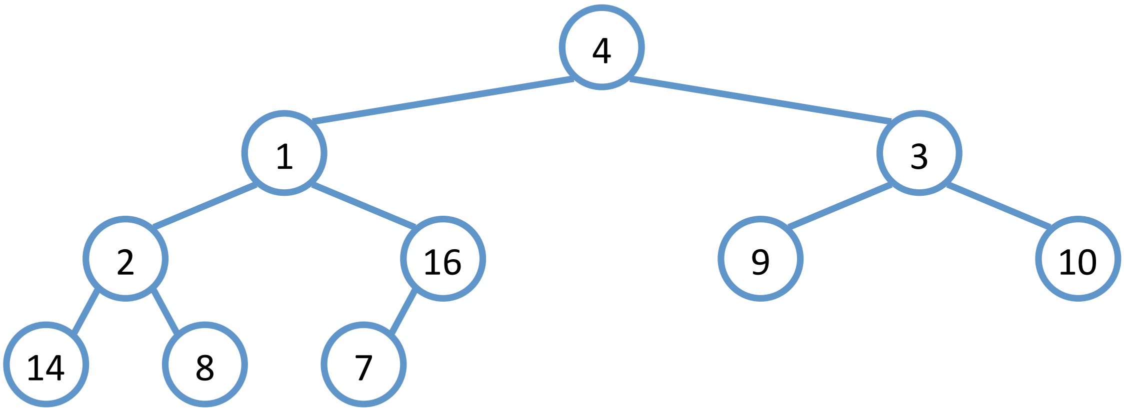
- We can build a heap in a bottom-up manner by running **`Heapify`** on successive subarrays
  - – Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - – So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

# BuildHeap

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
   Heapify(A, i);
}
```

# BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



# Analyzing BuildHeap

- Each call to **`Heapify`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O*(n*)
  - *How can this be? Is there a flaw in the above reasoning?*

# Analyzing BuildHeap: Tight

Theorem: The heap implementation realizes build in time O(n).

Proof:

- There are at most $2^l$ nodes of depth l (from root).
- A call of Heapify for each of these nodes takes time O(k-l), k depth of the tree.
- Get total runtime by summing up l=0, …, k-1

Theorem: The heap implementation realizes BuildHeap in time O(n).

Proof:

- There are at most $2^l$ nodes of depth l.
- A call of sift down for each of these nodes takes time O(k-l), k height of the tree.
- Get total runtime by summing up l=0, …, k-1

# Proof Runtime Build

- Total runtime:

$$
\begin{aligned}
& O\left(\sum_{l=0}^{k-1} 2^l \cdot (k-l)\right) \\
&= O\left(2^k \sum_{l=0}^{k-1} 2^{-k+l} \cdot (k-l)\right) \\
&= O\left(2^k \sum_{j=1}^{k} 2^{-j} \cdot j\right) \\
&= O(n) \qquad \square
\end{aligned}
$$

Explanation:

$$2^{\lfloor \log n \rfloor} \leq n \text{ and } \sum_{j=1}^{k} 2^{-j} \cdot j < 2$$

# Heapsort

Want to have a Sorting algorithm based on heaps that runs in time O(n log n).

Idea:

- Build the heap for n elements in time O(n).
- Pick in each step the maximum element (root) and delete it. (Time O(log n))
- Iterate until heap is empty.

In total n iterations implies total runtime O(n log n)

# Heapsort

- Given **`BuildHeap`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

# Heapsort

```
Heapsort(A)
{
   BuildHeap(A);
   for (i = length(A) downto 2)
   {
      Swap(A[1], A[i]);
      heap_size(A) -= 1;
      Heapify(A, 1);
   }
}
```

# Analyzing Heapsort

- The call to **`BuildHeap`** takes $O(n)$ time
- Each of the $n - 1$ calls to **`Heapify`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

# Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **`Insert`**, **`Maximum`**, and **`ExtractMax`**
  - ***What might a priority queue be useful for?***

# Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*

# Implementing Priority Queues

```
HeapInsert(A, key)    // what’s running
  time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

# Implementing Priority Queues

```
HeapMaximum(A)
{
    // This one is really tricky:

    return A[i];
}
```

# Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```