# 6331 - Algorithms, CSE, OSU

## Heapsort

Instructor: Anastasios Sidiropoulos

# Sorting

Given an array of integers $A[1 \ldots n]$, rearrange its elements so that

$$A[1] \leq A[2] \leq \ldots \leq A[n].$$

# A simple sorting algorithm

Bubble-Sort

```
repeat
  swapped = false
  for i = 1 to n − 1 do
    if A[i] > A[i + 1] then
      swap(A[i], A[i + 1])
      swapped = true
    end if
  end for
until not swapped
```

# A simple sorting algorithm

Bubble-Sort

```
repeat
  swapped = false
  for i = 1 to n − 1 do
    if A[i] > A[i + 1] then
      swap(A[i], A[i + 1])
      swapped = true
    end if
  end for
until not swapped
```

What is the worst-case time complexity of this algorithm?

# A simple sorting algorithm

Bubble-Sort

```
repeat
  swapped = false
  for i = 1 to n − 1 do
    if A[i] > A[i + 1] then
      swap(A[i], A[i + 1])
      swapped = true
    end if
  end for
until not swapped
```

What is the worst-case time complexity of this algorithm?

We can do much better!

# Heaps

A *Heap* is a data structure representing a full binary tree.

# Heaps

A *Heap* is a data structure representing a full binary tree.

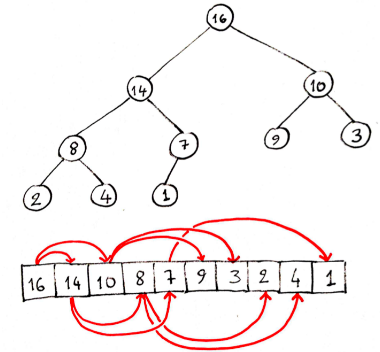
- A heap is stored in an array $A[1 \ldots n]$.
- The root is $A[1]$.
- $\text{parent}(i) = i/2$.
- $\text{left-child}(i) = 2i$.
- $\text{right-child}(i) = 2i + 1$.

# Max-Heaps

For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$

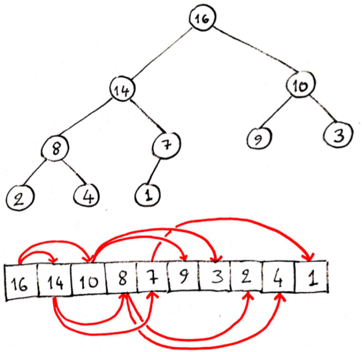
# Max-Heaps

For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$



# Max-Heaps

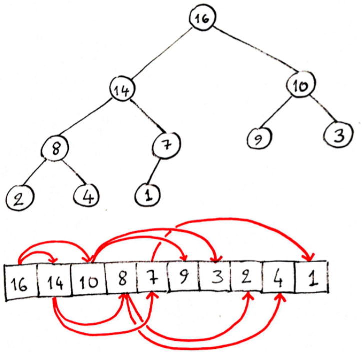
For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$




- Where is the maximum element in the tree?

# Max-Heaps

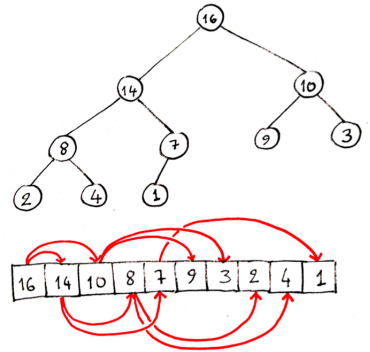
For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$




- Where is the maximum element in the tree?
- Where is the maximum element in the array?

# Max-Heaps

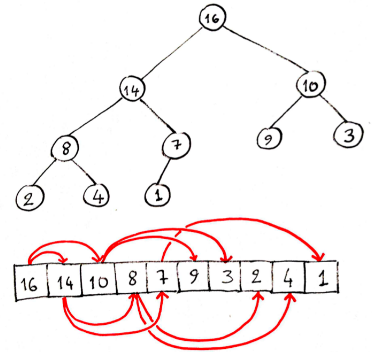
For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$




- Where is the maximum element in the tree?
- Where is the maximum element in the array?
- Where is the minimum element in the tree?

# Max-Heaps

For all nodes other than the root, we have $A[\text{parent}(i)] \geq A[i]$




- Where is the maximum element in the tree?
- Where is the maximum element in the array?
- Where is the minimum element in the tree?
- Where is are the leaves in the array?

# Height

The height of a node $i$ is the maximum number of edges on a path from $i$ to a leaf.

# Height

The height of a node $i$ is the maximum number of edges on a path from $i$ to a leaf.

The height of a tree is the height of its root.

# Height

The height of a node $i$ is the maximum number of edges on a path from $i$ to a leaf.

The height of a tree is the height of its root.

What is the height of a heap?

# Building and using heaps

- Procedure Max-Heapify (auxiliary procedure)
- Procedure Build-Max-Heap (building a max-heap)
- Procedure Heap-Sort (sorting using a heap)

# Maintaining the max-heap property

Suppose that the subtrees rooted at left-child($i$) and right-child($i$) are max-heaps.

However, $i$ might violate the max-heap property.
E.g., $A[i] < A[\text{left-child}(i)]$.

# Maintaining the max-heap property

Suppose that the subtrees rooted at left-child($i$) and right-child($i$) are max-heaps.

However, $i$ might violate the max-heap property.
E.g., $A[i] < A[\text{left-child}(i)]$.

How can we enforce the max-heap property?

# Maintaining the max-heap property

```
Procedure Max-Heapify(A, i)
  l = left-child(i)
  r = right-child(i)
  if l ≤ n and A[l] > A[i]
    largest = l
  else largest = i
  if r ≤ n and A[r] > largest
    largest = r
  if largest ≠ i
    exchange A[i] with A[largest]
    Max-Heapify(A, largest)
```

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify$(A, \text{largest})$.

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify$(A, \text{largest})$.
- Total running time of the recursion?

# Running time of Max-Heapify

- What is the running time of Max-Heapify($A, i$)?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify($A$, largest).
- Total running time of the recursion?
- What is the depth of the recursion?

# Running time of Max-Heapify

- What is the running time of Max-Heapify($A, i$)?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify($A$, largest).
- Total running time of the recursion?
- What is the depth of the recursion?
- Worst-case depth of recursion = height of $i$.

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify$(A, \text{largest})$.
- Total running time of the recursion?
- What is the depth of the recursion?
- Worst-case depth of recursion = height of $i$.
- Worst-case running time is $O(\text{height}(i))$.

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify$(A, \text{largest})$.
- Total running time of the recursion?
- What is the depth of the recursion?
- Worst-case depth of recursion = height of $i$.
- Worst-case running time is $O(\text{height}(i))$.
- Worst-case running time is $O(\log(n))$.

# Running time of Max-Heapify

- What is the running time of Max-Heapify$(A, i)$?
- Total time spent in Max-Heapify is at most $O(1)$ + the time spent in the recursive call Max-Heapify$(A, \text{largest})$.
- Total running time of the recursion?
- What is the depth of the recursion?
- Worst-case depth of recursion = height of $i$.
- Worst-case running time is $O(\text{height}(i))$.
- Worst-case running time is $O(\log(n))$.
- Is this tight?

# Building a heap

Procedure Build-Max-Heap($A$)
  for $i = \lfloor n/2 \rfloor$ downto 1
    Max-Heapify($A, i$)

# Building a heap

Procedure Build-Max-Heap($A$)
for $i = \lfloor n/2 \rfloor$ downto 1
Max-Heapify($A, i$)

**Loop invariant:**
At the start of each iteration, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

# Building a heap

Procedure Build-Max-Heap($A$)
  for $i = \lfloor n/2 \rfloor$ downto 1
    Max-Heapify($A, i$)

**Loop invariant:**
At the start of each iteration, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

- **Initialization:** $i = \lfloor n/2 \rfloor$. The nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and so they are max-heaps.

# Building a heap

Procedure Build-Max-Heap($A$)
  for $i = \lfloor n/2 \rfloor$ downto 1
    Max-Heapify($A, i$)

**Loop invariant:**
At the start of each iteration, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

- **Initialization:** $i = \lfloor n/2 \rfloor$. The nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and so they are max-heaps.
- **Maintenance:** By the loop invariant, the children of $i$ are roots of max-heaps. Therefore, running Max-Heapify makes $i$ the root of a max-heap.

# Building a heap

Procedure Build-Max-Heap($A$)
  for $i = \lfloor n/2 \rfloor$ downto 1
    Max-Heapify($A, i$)

**Loop invariant:**
At the start of each iteration, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

- **Initialization:** $i = \lfloor n/2 \rfloor$. The nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, and so they are max-heaps.
- **Maintenance:** By the loop invariant, the children of $i$ are roots of max-heaps. Therefore, running Max-Heapify makes $i$ the root of a max-heap.
- **Termination:** $i = 1$. By the loop invariant, 1 is the root of a heap.

# Running time of Max-Heapify

- Each call to Max-Heapify takes time $O(\log n)$.

# Running time of Max-Heapify

- Each call to Max-Heapify takes time $O(\log n)$.
- There are $O(n)$ calls to Max-Heapify.

# Running time of Max-Heapify

- Each call to Max-Heapify takes time $O(\log n)$.
- There are $O(n)$ calls to Max-Heapify.
- Total running time $O(n \cdot \log(n))$.

# Running time of Max-Heapify

- Each call to Max-Heapify takes time $O(\log n)$.
- There are $O(n)$ calls to Max-Heapify.
- Total running time $O(n \cdot \log(n))$.
- This is not asymptotically tight!

# Running time of Max-Heapify: Better analysis

- Each call Max-Heapify$(A, i)$ takes time $O(\text{height}(i))$.

# Running time of Max-Heapify: Better analysis

- Each call Max-Heapify$(A, i)$ takes time $O(\text{height}(i))$.
- A heap has height $\lfloor \log(n) \rfloor$.

# Running time of Max-Heapify: Better analysis

- Each call Max-Heapify$(A, i)$ takes time $O(\text{height}(i))$.
- A heap has height $\lfloor \log(n) \rfloor$.
- There are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$.

# Running time of Max-Heapify: Better analysis

- Each call Max-Heapify$(A, i)$ takes time $O(\text{height}(i))$.
- A heap has height $\lfloor \log(n) \rfloor$.
- There are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$.
- Total running time:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).$$

# Sorting using a heap

```
Procedure Heapsort(A)
  Build-Max-Heap(A)
  for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

# Sorting using a heap

Procedure Heapsort($A$)
  Build-Max-Heap($A$)
  for $i = A$.length downto 2
    exchange $A[1]$ with $A[i]$
    $A$.heap-size $= A$.heap-size $- 1$
    Max-Heapify($A, 1$)

- Build-Max-Heap takes time $O(n)$.

# Sorting using a heap

Procedure Heapsort($A$)
  Build-Max-Heap($A$)
  for $i = A$.length downto 2
    exchange $A[1]$ with $A[i]$
    $A$.heap-size $= A$.heap-size $- 1$
    Max-Heapify($A, 1$)

- Build-Max-Heap takes time $O(n)$.
- There are $n - 1$ calls to Max-Heapify.

# Sorting using a heap

Procedure Heapsort($A$)
  Build-Max-Heap($A$)
  for $i$ = $A$.length downto 2
    exchange $A[1]$ with $A[i]$
    $A$.heap-size = $A$.heap-size − 1
    Max-Heapify($A, 1$)

- Build-Max-Heap takes time $O(n)$.
- There are $n - 1$ calls to Max-Heapify.
- Each call to Max-Heapify takes time $O(\log n)$.

# Sorting using a heap

```
Procedure Heapsort(A)
  Build-Max-Heap(A)
  for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    Max-Heapify(A, 1)
```

- Build-Max-Heap takes time $O(n)$.
- There are $n-1$ calls to Max-Heapify.
- Each call to Max-Heapify takes time $O(\log n)$.
- Total running time $O(n\log(n))$.

# Priority queues

A *priority queue* is a data structure for maintaining a set $S$ of elements, each having a *key*.

# Priority queues

A *priority queue* is a data structure for maintaining a set $S$ of elements, each having a *key*.

There are max-priority queues, and min-priority queues.

# Priority queues

A *priority queue* is a data structure for maintaining a set $S$ of elements, each having a *key*.

There are max-priority queues, and min-priority queues.

Operations of a max-priority queue:

- Insert$(S, x)$: $S = S \cup \{x\}$.
- Maximum$(S)$: Return the element in $S$ with the maximum key.
- Extract-Max$(S)$: Removes and returns the element in $S$ with the maximum key.
- Increase-Key$(S, x, k)$: Increases the value of the key of $x$ to $k$, assuming that $k$ is larger than the current value.

# Implementing a max-priority queue using a max-heap

Procedure Heap-Maximum($A$)

# Implementing a max-priority queue using a max-heap

Procedure Heap-Maximum($A$)
  return $A[1]$

# Implementing a max-priority queue using a max-heap

Procedure Heap-Extract-Max($A$)

# Implementing a max-priority queue using a max-heap

Procedure Heap-Extract-Max($A$)
  if $n < 1$
    error "empty heap"
  max $= A[1]$
  $A[1] = A[n]$
  $n = n - 1$
  Max-Heapify($A, 1$)
  return max

# Implementing a max-priority queue using a max-heap

Procedure Heap-Increase-Key($A, i, \text{key}$)

# Implementing a max-priority queue using a max-heap

```
Procedure Heap-Increase-Key(A, i, key)
  if key < A[i]
    error
  A[i] = key
  while i > 1 and A[parent(i)] < A[i]
    exchange A[i] with A[parent(i)]
    i = parent(i)
```

# Implementing a max-priority queue using a max-heap

Procedure Max-Heap-Insert($A$, key)

# Implementing a max-priority queue using a max-heap

Procedure Max-Heap-Insert($A$, key)

$n = n + 1$

$A[n] = -\infty$

Heap-Increase-Key($A$, $n$, key)