Binary heaps

- A binary heap is an array-based data structure that may be viewed as a (nearly) complete binary tree
  - Each node corresponds to an array element
  - Tree is filled on all levels except possibly the lowest level which is filled from left to right
- An array $A$ that represents a heap has two attributes:
  - $\text{length}[A] =$ the number of elements in the array
  - $\text{heap-size}[A] =$ the number of elements in the heap

![Binary heap example](image)

Heap properties

- A **max-heap** is a binary heap that maintains the **max-heap property**: $A[\text{Parent}(i)] \geq A[i]$
  - Largest element is stored at the root
- A **min-heap** is a binary heap that maintains the **min-heap property**: $A[\text{Parent}(i)] \leq A[i]$
  - Smallest element is stored at the root
- The **height** of a node is the number of edges on the longest simple downward path from the node to a leaf
- The height of the heap is the height of the root node
  - Since the heap is a nearly complete binary tree, its height is $\Theta(\log n)$
Maintaining the heap property

- Max-Heapify assumes the binary trees rooted at Right(i) and Left(i) are max-heaps, but A[i] may be smaller than its children
- What is the running time?
  - T(n) = (time to adjust i, Left(i), Right(i)) + (time to run Max-Heapify on one of children)
  - Maximum size of a child subtree is 2n/3 (occurs when lowest level is half full)
- T(n) ≤ T(2n/3) + Θ(1)
- T(n) = Θ(lg n)

<table>
<thead>
<tr>
<th>MAX-HEAPIFY(A, i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 i ← LEFT(i)</td>
</tr>
<tr>
<td>2 r ← RIGHT(i)</td>
</tr>
<tr>
<td>3 if i ≤ heap-size[A] and A[i] &gt; A[r]</td>
</tr>
<tr>
<td>4 then largest ← i</td>
</tr>
<tr>
<td>5 else largest ← r</td>
</tr>
<tr>
<td>7 then largest ← r</td>
</tr>
<tr>
<td>8 if largest ≠ i</td>
</tr>
<tr>
<td>9 then exchange A[i] ← A[largest]</td>
</tr>
<tr>
<td>10 MAX-HEAPIFY(A, largest)</td>
</tr>
</tbody>
</table>

Building a heap

- Use Max-Heapify in a bottom-up manner to convert an array into a max-heap
  - Elements A[⌊n/2⌋ + 1] ... n are all leaves of the tree
- Loop invariant: at the start of each iteration of the for loop of lines 2-3, each node i+1, i+2, ..., n is the root of a max-heap
- Initialization: prior to the first iteration of the loop, i = ⌊n/2⌋
  - Each node ⌈n/2⌉+1, ⌈n/2⌉+2, ..., n is a leaf and is thus the root of a trivial max-heap
- Maintenance: by the loop invariant Left(i) and Right(i) are both roots of max-heaps
  - This is the condition required for Max-Heapify to make node i a max-heap root
  - Max-Heapify preserves property that nodes i+1, ..., n are max-heap roots
  - Following Max-Heapify, decrementing i reestablishes the loop invariant
- Termination: at termination, i=0; by the loop invariant each node 1, 2, ..., n is a max-heap root
  - Thus, node 1 is a max-heap root

<table>
<thead>
<tr>
<th>BUILD-MAX-HEAP(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 heap-size[A] ← length[A]</td>
</tr>
<tr>
<td>2 for i ← [length[A]/2] downto 1</td>
</tr>
<tr>
<td>3 do MAX-HEAPIFY(A, i)</td>
</tr>
</tbody>
</table>

Running time of Build-Max-Heap

- Easy to derive an O(n lg n) bound
  - Upper bound for Max-Heapify is O(lg n)
  - Build-Max-Heap calls Max-Heapify ⌈n/2⌉ times
- Can we derive a tighter bound?
  - Time to execute Max-Heapify varies with the height of the node in the tree
    - Execution time for node of height h is O(h)
  - An n-element heap has height ⌈lg n⌉
  - An n-element heap has at most ⌈n/2h+1⌉ nodes of any height h
    \[ \sum_{h=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor} h) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n) \]
  - We can show that Build-Max-Heap is O(n)

Heapsort

<table>
<thead>
<tr>
<th>HEAPSORT(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BUILD-MAX-HEAP(A)</td>
</tr>
<tr>
<td>2 for i ← length[A] downto 2</td>
</tr>
<tr>
<td>4 heap-size[A] ← heap-size[A] − 1</td>
</tr>
<tr>
<td>5 MAX-HEAPIFY(A, 1)</td>
</tr>
</tbody>
</table>

- Heapsort sorts array A in place, back to front (max to min)
- At the completion of each iteration of Max-Heapify, A[1] is the maximum element left in the heap
- Each iteration of the for loop places the maximum element in its sorted position and reduces the heap-size (removes it from the heap), then runs Max-Heapify to find the maximum element left in the heap
- T(n) = O(n) + (n-1)O(lg n) = O(n lg n)
Priority queues

- A priority queue is a data structure for maintaining a set of $S$ elements with associated key values.
- Elements may be inserted into a priority queue at any time.
- In max-priority queues only the maximum element may be removed from the queue.
  - The increase-key operation may be used to increase the value of an element’s key and possibly change the relative ordering of elements.
- Min-priority queues implement symmetric operations.
- Priority queues are useful for a multitude of scheduling tasks where the highest priority element/job/packet should be scheduled first.
- Max-priority queues leverage the constant time Heap-Maximum operation.

```
HEAP-MAXIMUM(A)
1    return A[1]
```

Extract-Max

- For priority queues implemented with max-heaps, Heap-Extract-Max can be used to remove the maximum element.
- $T(n) = O(lg n)$

```
HEAP-EXTRACT-MAX(A)
1    if heap-size[A] < 1
2        then error “heap underflow”
3        max ← A[1]
5        heap-size[A] ← heap-size[A] − 1
6        MAX-HEAPIFY(A, 1)
7        return max
```

Increase-Key

- Given the index $i$ of an element and a new key value (greater than the current key value), Heap-Increase-Key increases the key value of the element and ensures max-heap property holds.
- After changing key value, Heap-Increase-Key traverses a path from $A[i]$ toward the root.
  - Compares keys of an element and its parent and exchanges if the element’s key is greater than the parent’s key.
- $T(n) = O(lg n)$

```
HEAP-INCREASE-KEY(A, i, key)
1    if key < A[i]
2        then error “new key is smaller than current key”
3        A[i] ← key
4        while i > 1 and A[Parent(i)] < A[i]
5            do exchange A[i] ← A[Parent(i)]
6            i ← Parent(i)
```

Insert

- Given a new key value, Max-Heap-Insert creates a new heap element with the given key value and ensures max-heap property is maintained.
- Increases heap-size by 1.
- Assigns new element the minimum key value.
- Calls Heap-Increase-Key using new element’s index and new key value.
- $T(n) = O(lg n)$

```
MAX-HEAP-INSERT(A, key)
1    heap-size[A] ← heap-size[A] + 1
2    A[heap-size[A]] ← −∞
3    HEAP-INCREASE-KEY(A, heap-size[A], key)
```