Computer Science 331
Heap Sort

Mike Jacobson

Department of Computer Science
University of Calgary

Lectures #26-28

Outline

Binary Heaps

Definition: A *binary heap* is

- a binary tree whose nodes store elements of a multiset (possibly including multiple copies of the same value)
- every heap of size n has the same *shape*
- values at nodes are arranged in *heap order*

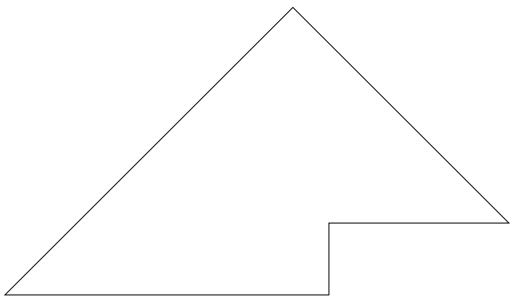
Applications:

- Used to implement another efficient sorting algorithm (Heap Sort)
- One of the data structures commonly used to implement another useful abstract data type (Priority Queue)

Heap Shape

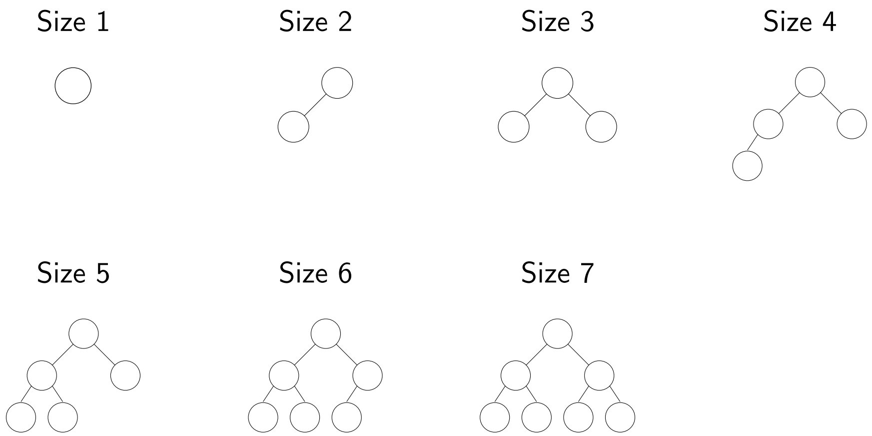
A heap is a *complete* binary tree:

- As the size of a heap increases, nodes are added on each level, from left to right, as long as room at that level is available.



Heap Shape: Examples

Shapes of Heaps with Sizes 1–7:



Height

The *height* of a node, and of a heap, are defined as follows.

- **Height of a Node in a Heap:** Number of edges on the longest path from the node down to a leaf
- **Height of a Heap:** Height of the root of the heap

Note: same as the node's height as a binary tree

Theorem 1

If a heap has size n then its height $h \in \Theta(\log n)$.

Proof: use the fact that a heap is a *complete* tree — every level contains as many nodes as possible.

Proof of Height Bound

Proof.

Lower bound:

- $n \leq 2^{h+1} - 1$ (equal if tree is full)
- thus $h \geq \log_2(n+1) - 1 > \log_2 n - 1$ if $n \geq 1$ and $h \in \Omega(\log n)$

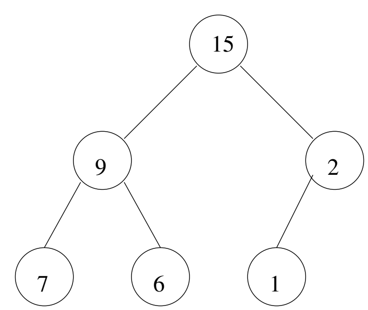
Upper bound:

- 2^i keys at depth $i = 0, \ldots, h-1$
- at least 1 key at depth h
- $n \geq 1 + 2 + 4 + \cdots + 2^{h-1} + 1$
- thus, $n \geq 2^h - 1 + 1$, i.e., $h \leq \log_2 n$ and $h \in O(\log n)$

Conclusion: Therefore $h \in \Theta(\log n)$ (can show that $h = \lfloor \log_2 n \rfloor$) □

Max-Heaps

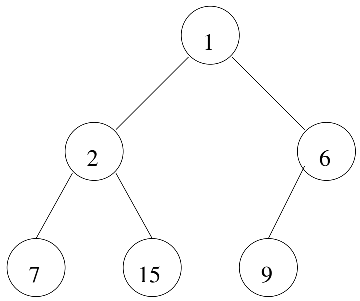
Max-Heaps satisfy the *Max-Heap Property:* The value at each node is *greater than or equal* to values at any children of the node.



Application: The Heap Sort algorithm

Min-Heaps

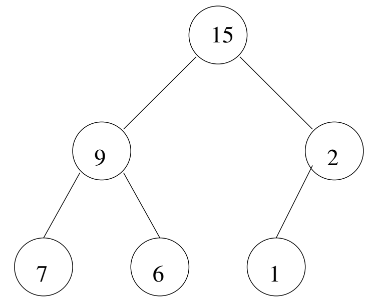
Min-Heaps satisfy the *Min-Heap Property:* The value at each node is *less than or equal* to the values at any children of the node.



Application: Used for Priority Queues

Representation Using an Array

A heap with size n can be represented using an array with size $m \geq n$



0	1	2	3	4	5
15	9	2	7	6	1

Index of Root: 0

For $i \geq 0$

- parent$(i) = \lfloor (i-1)/2 \rfloor$
- left$(i) = 2i + 1$
- right$(i) = 2i + 2$

Representation Using an Array

Suppose A is an array used to represent a binary heap.

Notation:

- $A[i]$: value stored at the node whose index is i
- heap-size(A): size of the heap represented using A

Properties:

- heap-size$(A) \leq A.length$
- The entries
$$A[0], A[1], \ldots, A[\text{heap-size}(A) - 1]$$
are used to store the entries in the heap.

Overview

Operations on Binary Heaps:

- Insertion into a Max-Heap
- Deletion of the Largest Element from a Max-Heap

Like red-black tree operations each has two stages:

a) A simple change determines the output and the set of values stored, but destroys the Max-Heap property

b) A sequence of local adjustments restores the Max-Heap property.

The corresponding Min-Heap operations replace the comparisons used and are otherwise the same.

Insertion: Specification of Problem

Signature: `void` **insert**(`T[] A, T key`)

Precondition 1:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) `key` is a value of type `T`
c) $\texttt{heap-size(A)} < \texttt{A.length}$

Postcondition 1:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) The given `key` has been added to the multiset of values stored in this Max-Heap, which has otherwise been unchanged

Insertion: Specification of Problem

Precondition 2:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) `key` is a value of type `T`
c) $\texttt{heap-size(A)} = \texttt{A.length}$

Postcondition 2:

a) A `FullHeapException` is thrown
b) `A` (and the Max-Heap it represents) has not been changed

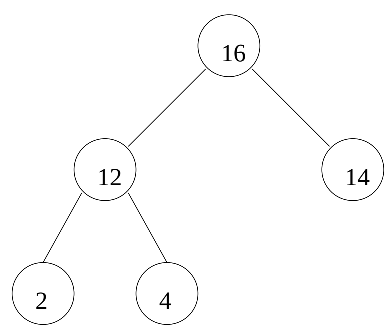
Step 1: Adding the Element

Pseudocode:

```
void insert(T[] A, T key)
  if heap-size(A) < A.length then
    A[heap-size(A)] = key
    heap-size(A) = heap-size(A) + 1
    The rest of this operation will be described in Step 2
  else
    throw new FullHeapException
  end if
```


Example: Insertion, Step 1

Suppose that `A` is as follows.

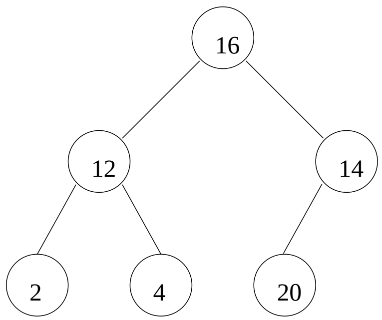


0	1	2	3	4	5	6	7
16	12	14	2	4	1	9	3

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 5$

Example: Insertion, Step 1

Step 1 of the insertion of the key 20 produces the following:



0	1	2	3	4	5	6	7
16	12	14	2	4	20	9	3

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 6$

Step 2: Restoring the Max-Heap Property

Situation After Step 1:

- The given key has been added to the Max-Heap and stored in some position j in A
- If this value is at the root (because the heap was empty, before this) or is less than or equal to the value at its parent, then we have a produced a Max-Heap
- Otherwise we will move the value closer to the root until the Max-Heap property is restored

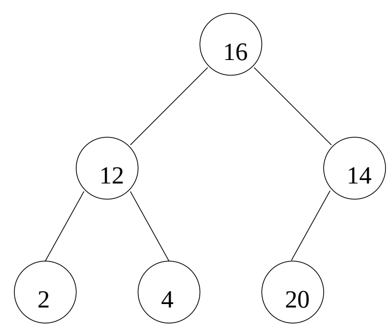
Step 2: Restoring the Max-Heap Property

Pseudocode for Step 2:

```
j = heap-size(A) − 1
while j > 0 and A[j] > A[parent(j)] do
  tmp = A[j]
  A[j] = A[parent(j)]
  A[parent(j)] = tmp
  j = parent(j)
end while
```


Example: Execution of Step 2

Consider the following heap, which was produced using our ongoing example at the end of Step 1:



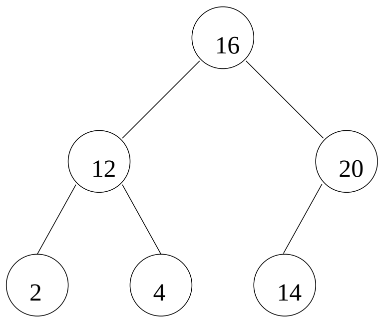
0	1	2	3	4	5	6	7
16	12	14	2	4	20	9	3

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 6$

Initial value of j: 5

Example: Execution of Step 2

A and j are as follows after the *first* execution of the body of the loop in Step 2:



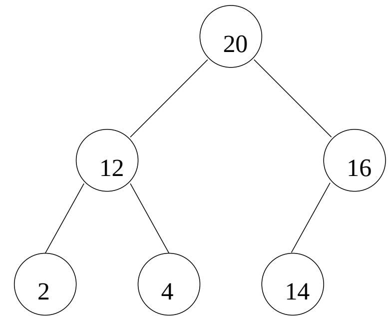
0	1	2	3	4	5	6	7
16	12	20	2	4	14	9	3

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 6$

Current value of j: 2

Example: Execution of Step 2

A and j are as follows after the *second* execution of the body of this loop:



0	1	2	3	4	5	6	7
20	12	16	2	4	14	9	3

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 6$

Current value of j: 0

The loop terminates at this point.

Step 2: Partial Correctness

The following properties are satisfied at the beginning of each execution of the body of the loop:

a) The first heap-size(A) entries of A are the multiset obtained from the original contents of the heap by inserting a copy of the given key

b) j is an integer such that $0 \leq \texttt{j} < \texttt{heap-size(A)}$

c) For every integer h such that $1 \leq \texttt{h} < \texttt{heap-size(A)}$, if $\texttt{h} \neq \texttt{j}$ then $\texttt{A[h]} \leq \texttt{A[parent(h)]}$

d) If $\texttt{j} > 0$ and $\texttt{left(j)} < \texttt{heap-size(A)}$ then $\texttt{A[left(j)]} \leq \texttt{A[parent(j)]}$

e) If $\texttt{j} > 0$ and $\texttt{right(j)} < \texttt{heap-size(A)}$ then $\texttt{A[right(j)]} \leq \texttt{A[parent(j)]}$

Step 2: Partial Correctness

If the loop invariant holds and the loop guard is *true*, then

- $0 < \texttt{j} < \texttt{heap-size(A)}$ and $\texttt{A[j]} > \texttt{A[parent(j)]}$
- Both children of A[j] (if they exist) are $\leq$ A[parent(j)].

After the loop body executes:

- $\texttt{j}_{new} = \texttt{parent}(\texttt{j}_{old})$, $\texttt{A}[\texttt{j}_{old}]$ and $\texttt{A}[\texttt{parent}(\texttt{j}_{old})]$ are swapped.
- $\texttt{A}[\texttt{j}_{old}]$ is $\geq$ both of its children
- Properties (a), (c), (d) and (e) of the loop invariant are satisfied.

If the loop invariant holds but the loop guard *fails*:

- $\texttt{j} = 0$, or $0 < \texttt{j} < \texttt{heap-size(A)}$ and $\texttt{A[j]} \leq \texttt{A[parent(j)]}$
- Properties (a), (c), (d) and (e) of the loop invariant are satisfied.

Exercises:

1. Sketch proofs of the above claims.
2. Use these to prove the partial correctness of this algorithm.

Step 2: Termination and Efficiency

Loop Variant: $f(\mathtt{A}, \mathtt{j}) = \lfloor \log_2(\mathtt{j}+1) \rfloor$

Justification:

- integer value function
- decreases by 1 after each iteration, because j is replaced with $(j-1)/2$
- $f(\mathtt{A}, j) = 0$ implies that $j = 0$, in which case the loop terminates

Application of Loop Variant:

- inital value, and thus upper bound on the number of iterations, is $f(\mathtt{A}, \mathtt{heap\text{-}size(A)} - 1) = \lfloor \log_2 \mathtt{heap\text{-}size(A)} \rfloor$
- loop body and all other steps require constant time
- worst-case running time is in $O(\log \mathtt{heap\text{-}size(A)})$.

Step 2: Termination and Efficiency

Suppose that the given `key` is greater than the largest value stored in the Max-Heap represented by `A` when this operation is performed.

Lower Bound for Number of Steps Executed:

$$\Omega(\log \mathtt{heap\text{-}size(A)})$$

Conclusion: The worst-case cost of this operation is

$$\Theta(\log \mathtt{heap\text{-}size(A)})$$

DeleteMax: Specification of a Problem

Signature: `T` **deleteMax**(`T[] A`)

Preconditon 1:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) $\mathtt{heap\text{-}size(A)} > 0$

Postcondition 1:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) The value returned, `max`, is the largest value that was stored in this Max-Heap immediately before this operation
c) A copy of `max` has been removed from the multiset of values stored in this Max-Heap, which has otherwise been unchanged

DeleteMax: Specification of Problem

Precondition 2:

a) `A` is an array representing a Max-Heap that contains values of type `T`
b) $\mathtt{heap\text{-}size(A)} = 0$

Postcondition 2:

a) An `EmptyHeapException` is thrown
b) `A` (and the Max-Heap it represents) has not been changed

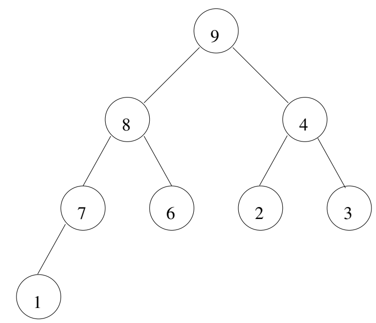
Deletion, Step 1

Pseudocode:

T **deleteMax**(T[] A)
  if heap-size(A) $>$ 0 **then**
    max = A[0]
    A[0] = A[heap-size(A)-1]
    heap-size(A) = heap-size(A) $-$ 1
    The rest of this operation will be described in Step 2
    return max
  else
    throw new EmptyHeapException
  end if

Example: Deletion, Step 1

Suppose that A is as follows.

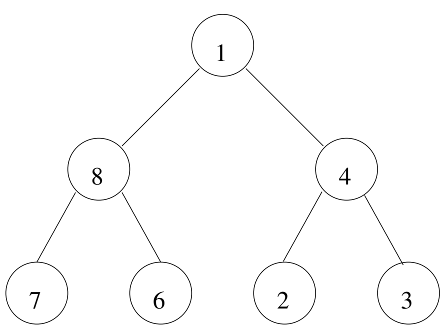


0	1	2	3	4	5	6	7
9	8	4	7	6	2	3	1

A.length = 8, heap-size(A) = 8

Example: Deletion, Step 1

After Step 1, max=9 and A is as follows:



0	1	2	3	4	5	6	7
1	8	4	7	6	2	3	1

A.length = 8, heap-size(A) = 7

Step 2: Restoring the Max-Heap Property

Situation After Step 1:

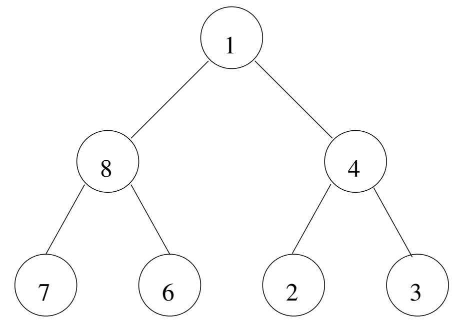
- A copy of the maximum element has been removed from the multiset stored in the heap, as required
- If the heap is still nonempty then a value has been moved from the deleted node to the root
- If the heap now has size at most one, or its size is at least two and the value at the root is larger than the value(s) at its children, then we have produced a Max-Heap
- Otherwise we should move the value at the root *down* in the heap by repeatedly exchanging it with the largest value at a child, until the Max-Heap property has been restored

Step 2: Restoring the Max-Heap Property

```
j = 0
while j < heap-size(A) do
  ℓ = left(j); r = right(j); largest = j
  if ℓ < heap-size(A) and A[ℓ] > A[largest] then
    largest = ℓ
  end if
  if r < heap-size(A) and A[r] > A[largest] then
    largest = r
  end if
  if largest ≠ j then
    tmp = A[j]; A[j] = A[largest]; A[largest] = tmp;
    j = largest
  else
    j = heap-size(A)
  end if
end while
```


Example: Execution of Step 2

Consider the following heap, which is produced using our ongoing example at the end of Step 1:



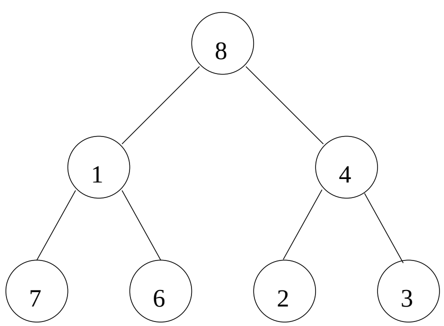
0	1	2	3	4	5	6	7
1	8	4	7	6	2	3	1

A.length = 8, heap-size(A) = 7

Initial value of j: 0

Example: Execution of Step 2

A and j are as follows after the *first* execution of the body of this loop:



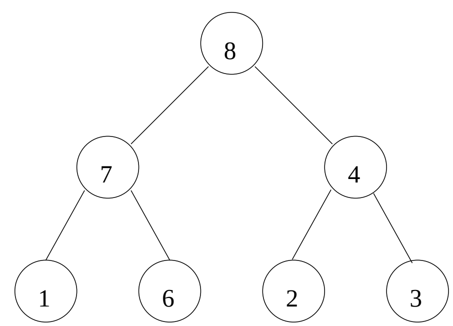
0	1	2	3	4	5	6	7
8	1	4	7	6	2	3	1

A.length = 8, heap-size(A) = 7

Current value of j: 1

Example: Execution of Step 2

A and j are as follows after the *second* execution of the body of this loop:



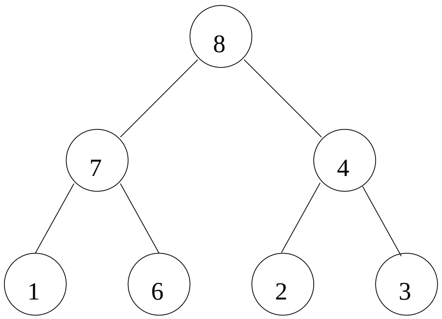
0	1	2	3	4	5	6	7
8	7	4	1	6	2	3	1

A.length = 8, heap-size(A) = 7

Current value of j: 3

Example: Execution of Step 2

A and j are as follows after the *third* execution of the body of this loop:



0	1	2	3	4	5	6	7
8	7	4	1	6	2	3	1

$\texttt{A.length} = 8$, $\texttt{heap-size(A)} = 7$

Current value of $\texttt{j}$: 7

The loop terminates at this point.

Step 2: Partial Correctness

The following properties are satisfied at the beginning of each execution of the body of the loop:

a) The first $\texttt{heap-size(A)}$ entries of $\texttt{A}$ are the multiset obtained from the original contents of the heap by deleting a copy of its largest value

b) $\texttt{j}$ is an integer such that $0 \leq \texttt{j} < \texttt{heap-size(A)}$

c) For every integer $\texttt{h}$ such that $0 \leq \texttt{h} < \texttt{heap-size(A)}$ and $\texttt{h} \neq \texttt{j}$,
   - if $\texttt{left(h)} < \texttt{heap-size(A)}$ then $\texttt{A[left(h)]} \leq \texttt{A[h]}$
   - if $\texttt{right(h)} < \texttt{heap-size(A)}$ then $\texttt{A[right(h)]} \leq \texttt{A[h]}$

d) If $\texttt{j} > 0$ and $\texttt{left(j)} < \texttt{heap-size(A)}$ then $\texttt{A[left(j)]} \leq \texttt{A[parent(j)]}$

e) If $\texttt{j} > 0$ and $\texttt{right(j)} < \texttt{heap-size(A)}$ then $\texttt{A[right(j)]} \leq \texttt{A[parent(j)]}$

Step 2: Partial Correctness

The following properties are satisfied at the *end* of every execution of the body of this loop.

- $\texttt{j}$ is an integer such that $0 \leq \texttt{j} \leq \texttt{heap-size(A)}$
- Properties (a), (c), (d) and (e) of the loop invariant are satisfied

On *termination* of this loop,

- $\texttt{j} = \texttt{heap-size(A)}$
- Properties (a), (c), (d) and (e) of the loop invariant are satisfied

Exercises:

1. Sketch proofs of the above claims.
2. Use these to prove the partial correctness of this algorithm.

Step 2: Termination and Efficiency

Loop Variant:

$$f(\texttt{A},\texttt{j}) = \begin{cases} 1 + \text{height}(\texttt{j}) & \text{if } 0 \leq \texttt{j} < \texttt{heap-size(A)} \\ 0 & \text{if } \texttt{j} = \texttt{heap-size(A)} \end{cases}$$

Justification:

- integer valued, decreases by 1 after each iteration (j replaced by root of a sub-heap)
- $f(\texttt{A}, j) = 0$ implies that $j = \texttt{heap-size(A)}$ (loop terminates)

Application of Loop Variant:

- inital value, and thus upper bound on the number of iterations, is $f(\texttt{A}, 0) = 1 + \textit{height}(0) = \lfloor \log \texttt{heap-size(A)} \rfloor$
- loop body and all other steps require constant time
- worst-case running time is in $O(\log \texttt{heap-size(A)})$.

Step 2: Termination and Efficiency

Suppose that the value moved to the root, at the end of step 1, is the smallest value in the heap.

Lower Bound for Number of Steps Executed:

$$\Omega(\log \texttt{heap-size(A)})$$

Conclusion: The worst-case cost of this operation is

$$\Theta(\log \texttt{heap-size(A)})$$

HeapSort

A deterministic sorting algorithm that can be used to sort an array of length n using $\Theta(n \log n)$ operations in the worst case

Unlike `MergeSort` (which has the same asymptotic worst-case performance) this algorithm can be used to sort "in place," overwriting the input array with the output array, and using only a constant number of additional registers for storage

A disadvantage of this algorithm is that it is a little bit more complicated than the other asymptotically fast sorting algorithms we are studying (and seems to be a bit slower in practice)

HeapSort

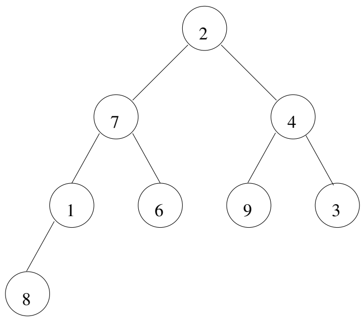
Idea:

- An array `A` of positive length, storing values from some ordered type `T`, can be turned into a Max-Heap of size 1 simply by setting `heap-size(A)` to be 1
- Inserting `A[1]`, `A[2]`, ..., `A[A.length-1]` produces a Max-Heap while reordering the entries of `A` (without changing them, otherwise)
- Repeated calls to `deleteMax` will then return the entries, listed in decreasing order, while freeing up the space in `A` where they should be located when sorting the array.

HeapSort

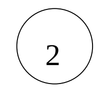
```
void heapSort(T[] A)
  heap-size(A) = 1
  i = 1
  while i < A.length do
    insert(A, A[i])
    i = i + 1
  end while
  i = A.length − 1
  while i > 0 do
    largest = deleteMax(A)
    A[i] = largest
    i = i − 1
  end while
```

Example (Input)



0	1	2	3	4	5	6	7
2	7	4	1	6	9	3	8

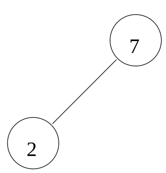
Example: Before First Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
2	7	4	1	6	9	3	8

heap-size(A) $= 1$

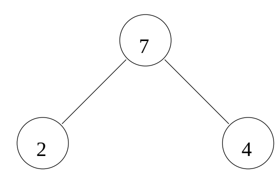
Example: Before Second Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

heap-size(A) $= 2$

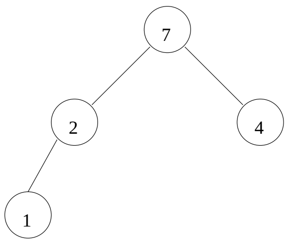
Example: Before Third Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

heap-size(A) $= 3$

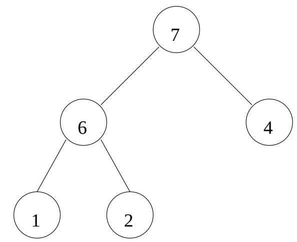
Example: Before Fourth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	2	4	1	6	9	3	8

$\texttt{heap-size(A)} = 4$

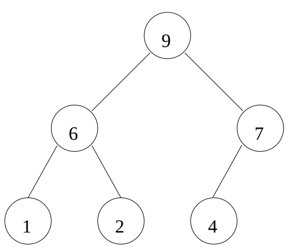
Example: Before Fifth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
7	6	4	1	2	9	3	8

$\texttt{heap-size(A)} = 5$

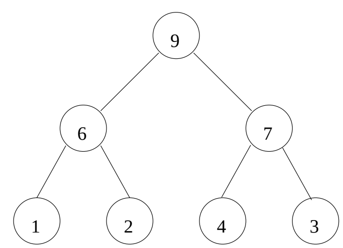
Example: Before Sixth Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
9	6	7	1	2	4	3	8

$\texttt{heap-size(A)} = 6$

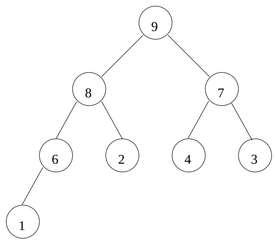
Example: Before Seventh Execution, Loop Body, First Loop



0	1	2	3	4	5	6	7
9	6	7	1	2	4	3	8

$\texttt{heap-size(A)} = 7$

Example: After Seventh Execution, Loop Body, First Loop

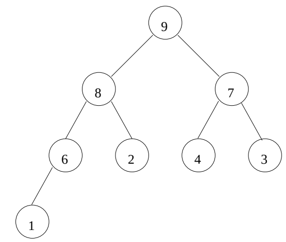


0	1	2	3	4	5	6	7
9	8	7	6	2	4	3	1

heap-size(A) $= 8$

Example: Before First Execution, Loop Body, Second Loop

i $= 7$

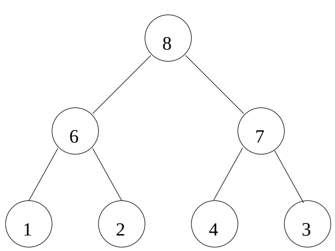


0	1	2	3	4	5	6	7
9	8	7	6	2	4	3	1

heap-size(A) $= 8$

Example: Before Second Execution, Loop Body, Second Loop

i $= 6$

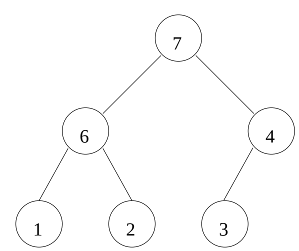


0	1	2	3	4	5	6	7
8	6	7	1	2	4	3	9

heap-size(A) $= 7$

Example: Before Third Execution, Loop Body, Second Loop

i $= 5$

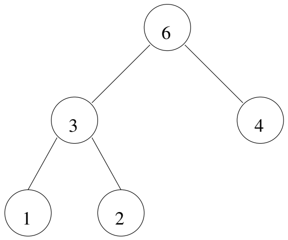


0	1	2	3	4	5	6	7
7	6	4	1	2	3	8	9

heap-size(A) $= 6$

Example: Before Fourth Execution, Loop Body, Second Loop

i = 4

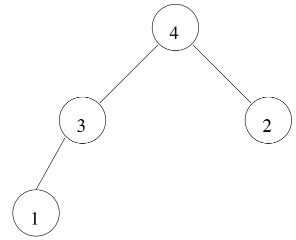


0	1	2	3	4	5	6	7
6	3	4	1	2	7	8	9

heap-size(A) = 5

Example: Before Fifth Execution, Loop Body, Second Loop

i = 3

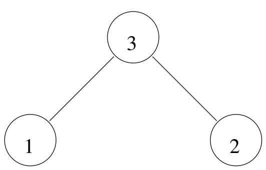


0	1	2	3	4	5	6	7
4	3	2	1	6	7	8	9

heap-size(A) = 4

Example: Before Sixth Execution, Loop Body, Second Loop

i = 2

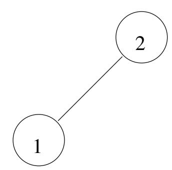


0	1	2	3	4	5	6	7
3	1	2	4	6	7	8	9

heap-size(A) = 3

Example: Before Seventh Execution, Loop Body, Second Loop

i = 1

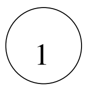


0	1	2	3	4	5	6	7
2	1	3	4	6	7	8	9

heap-size(A) = 2

Example: After Seventh Execution, Loop Body, Second Loop

$\mathtt{i} = 0$



0	1	2	3	4	5	6	7
1	2	3	4	6	7	8	9

$\mathtt{heap\text{-}size(A)} = 1$

Stop — array is sorted!

First Loop — Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the body of the first loop.

a) i is an integer such that $1 \leq \mathtt{i} < \mathtt{A.length}$
b) A represents a heap with size i
c) The entries of the array A have been reordered but are otherwise unchanged

At the *end* of each execution of the body of the first loop, the following properties are satisfied.

- i is an integer such that $1 \leq \mathtt{i} \leq \mathtt{A.length}$
- Parts (b) and (c) of the loop invariant are satisfied

On *termination* of this loop i = A.length, so A represents a heap with size A.length, and the entries of A have been reordered but are otherwise unchanged.

First Loop — Termination and Efficiency

Loop Variant: $\mathtt{A.length} - \mathtt{i}$

Application:

- Number of executions of the body of this loop is at most:

$$\mathtt{A.length} - 1$$

- The cost of a single execution of the body of this loop is at most: k

$$O(\log n), \text{ where } n = \mathtt{A.length}$$

- *Conclusion:* The number of steps used by this loop in the worst case is at most:

$$O(n \log n)$$

Second Loop — Partial Correctness

Loop Invariant: The following properties are satisfied at the beginning of each execution of the body of the second loop.

a) i is an integer such that $1 \leq \mathtt{i} < \mathtt{A.length}$
b) A represents a heap with size $\mathtt{i} + 1$
c) if $\mathtt{i} < \mathtt{A.length} - 1$ then $\mathtt{A[j]} \leq \mathtt{A[i+1]}$ for every integer j such that $0 \leq \mathtt{j} \leq \mathtt{i}$
d) $\mathtt{A[j]} \leq \mathtt{A[j+1]}$ for every integer j such that $\mathtt{i} + 1 \leq \mathtt{j} < \mathtt{A.length} - 1$
e) the entries of A have been reordered but are otherwise unchanged

Second Loop — Partial Correctness

At the *end* of each execution of the body of the second loop, the following properties are satisfied.

- `i` is an integer such that $0 \leq \mathtt{i} < \mathtt{A.length}$
- Parts (b), (c), (d) and (e) of the loop invariant are satisfied

On *termination* $\mathtt{i} = 0$ and parts (b), (c), (d) and (e) of the loop invariant are satisfied. Notes that, when $\mathtt{i} = 0$, parts (c) and (d) imply that the array is sorted, as required.

Second Loop — Termination and Efficiency

Loop Variant: `i`

Application:

- Number of executions of the body of this loop is at most:

$$\mathtt{A.length} - 1$$

- The cost of a single execution of the body of this loop is at most:

$$O(\log n), \text{ where } n = \mathtt{A.length}$$

- *Conclusion:* The number of steps used by this loop in the worst case is at most:

$$O(n \log n)$$

Analysis of Worst-Case Running Time, Concluded

Exercise: Show that if `A` is an array with length n, containing n *distinct* entries that already sorted in increasing order, then this `HeapSort` algorithm uses $\Omega(n \log n)$ steps on input `A`.

Conclusion: The worst-case running time of `HeapSort` (when given an input array of length n) is in $\Theta(n \log n)$.

Priority Queues

Definition: A *priority queue* is a data structure for maintaining a multiset `S` of elements, of some type `V`, each with an associated value (of some ordered type `P`) called a *priority*.

A class that implements *max-priority queue* provides the following operations (not, necessarily, with these names):

- `void insert(V value, P priority)`: Insert the given `value` into `S`, using the given `priority` as its priority in this priority queue
- `V maximum()`: Report an element of `S` stored in this priority that has highest priority, without changing the priority queue (or `S`)
- `V extract-max()`: Remove an element of `S` with highest priority from the priority queue (and from `S`) and return this value as output

Priority Queues

Priority Queues in Java:

- Class `PriorityQueue` in the Java Collections framework implements a "min-priority queue" — which would provide methods `minimum` and `extract-min` to replace `maximum` and `extract-max`, respectively
- Also implements the `Queue` interface, so the names `insert`, `minimum`, and `extract-min` of methods are replaced by the names `add`, `peek`, and `remove`, respectively.
- Furthermore, the signature of `insert` is a little different — no `priority` is provided — because the values themselves are used *as* their priorities (according to their "natural order")

Priority Queues

Dealing With This Restriction:

- In order to provide more general priorities, one can simply write a class, each of whose objects "has" a value of type `V` (that is, the element of `S` it represents) and that also "has" a value of type `P` (that is, the priority). The class should implement the `Comparable` interface, and `compareTo` should be implemented using the ordering for priorities

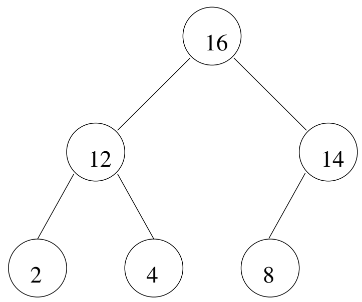
Applications:

- Scheduling: Priorities reflect the order of requests and determine the order in which they should be served

Implementation

Binary Heaps are often used to implement priority queues.

Example: One representation of a max-priority queue including keys $S = \{2, 4, 8, 12, 14, 16\}$ is as follows:



0	1	2	3	4	5	6	7
16	12	14	2	4	8	9	3

`A.length` $= 8$;
`heap-size(A)` $= 6$

Implementation of Operations

A "max-priority queue" can be implemented, in a straightforward way, using a Max-Heap.

- `insert:` Use the `insert` method for the binary heap that is being used to implement this priority queue
- `maximum:` Throw an exception if the binary heap has size zero; return data stored at position 0 if the array that represents the heap, otherwise
- `extract-min`: Use the `deleteMax` method for the binary heap that implements this priority queue

Consequence: If the priority queue has size n then `insert` and `extract-min` use $\Theta(\log n)$ operations in the worst case, while `maximum` uses $\Theta(1)$ operations in the worst case.

Binomial and Fibonacci Heaps

Introduction to Algorithms, Chapter 19 and 20

Better than binary heaps if **Union** operation must be supported:

- creates a new heap consisting of all nodes in two input heaps

Function	Binary Heap (worst-case)	Binomial Heap (worst-case)	Fib. Heap (amortized)
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
Maximum	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Extract-Max	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
Increase-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Union	$\Theta(n)$	$O(\log n)$	$\Theta(1)$

References

Additional Reading:

- **Introduction to Algorithms**, Chapter 6