Computer Science 331

Operations on Binary Heaps

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #25

Outline

Overview

To be considered today:

- 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) `heap-size(A)` $<$ `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) `heap-size(A)` $=$ `A.length`

Postcondition 2:

a) A `FullHeapException` is thrown

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

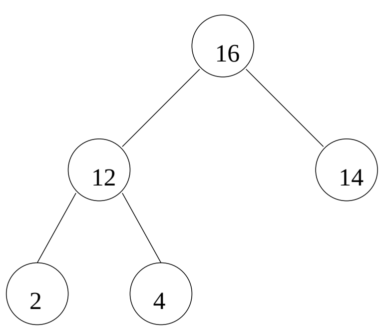
Step 1: Adding the Element

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.

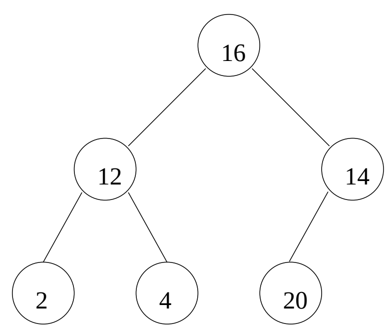


0	1	2	3	4	5	6	7
16	12	14	2	4	1	9	3

`A.length` $= 8$, `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

`A.length` $= 8$, `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 produced a Max-Heap
- Otherwise we will move the value closer to the root until the Max-Heap property is restored

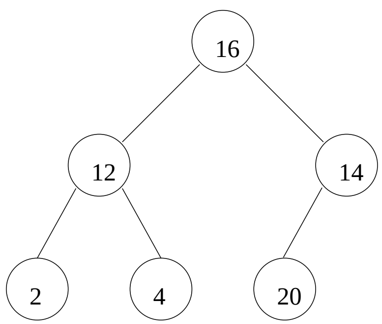
Step 2: Restoring the Max-Heap Property

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:



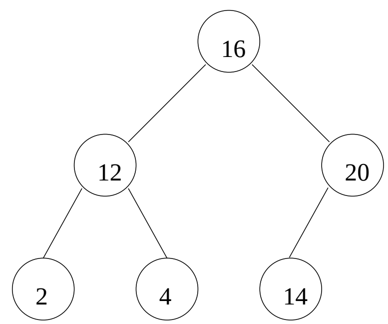
0	1	2	3	4	5	6	7
16	12	14	2	4	20	9	3

`A.length` = 8, `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:



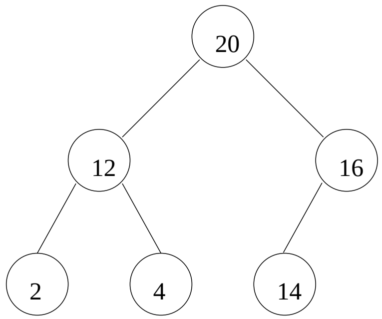
0	1	2	3	4	5	6	7
16	12	20	2	4	14	9	3

`A.length` = 8, `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 $0 < \texttt{j} < \texttt{heap-size(A)}$
c) For every integer h such that $1 \le \texttt{h} < \texttt{heap-size(A)}$, if $\texttt{h} \neq \texttt{j}$ then $\texttt{A[h]} \le \texttt{A[parent(h)]}$
d) $\texttt{A[j]} > \texttt{A[parent(j)]}$
e) If $\texttt{j} > 0$ and $\texttt{left(j)} < \texttt{heap-size(A)}$ then $\texttt{A[left(j)]} \le \texttt{A[parent(j)]}$
f) If $\texttt{j} > 0$ and $\texttt{right(j)} < \texttt{heap-size(A)}$ then $\texttt{A[right(j)]} \le \texttt{A[parent(j)]}$

Step 2: Partial Correctness

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

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

On *termination* of this loop,

- Either $\texttt{j} = 0$, or j is an integer such that $0 < \texttt{j} < \texttt{heap-size(A)}$ and $\texttt{A[j]} \le \texttt{A[parent(j)]}$
- Properties (a), (c), (e) and (f) 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: $f(\texttt{A}, \texttt{j}) = \lfloor \log_2(\texttt{j} + 1) \rfloor$

Justification:

- integer value function
- decreases by 1 after each iteration, because j is replaced with $(j-1)/2$
- $f(\texttt{A}, j) = 0$ 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 $f(\texttt{A}, \texttt{heap-size(A)} - 1) = \lfloor \log_2 \texttt{heap-size(A)} \rfloor$
- 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 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:

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

Conclusion: The worst-case cost of this operation is

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

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) `heap-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) `heap-size(A)` $= 0$

Postcondition 2:

a) An `EmptyHeapException` is thrown

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

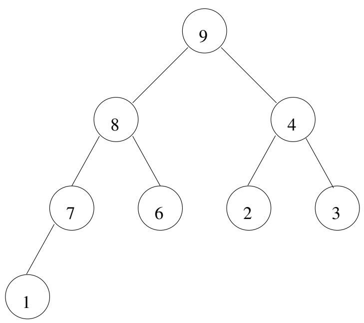
Deletion, Step 1

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.

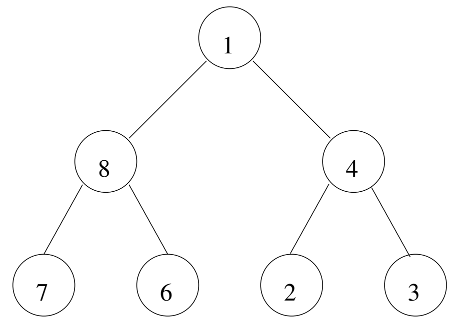


0	1	2	3	4	5	6	7
9	8	4	7	6	2	3	1

`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:

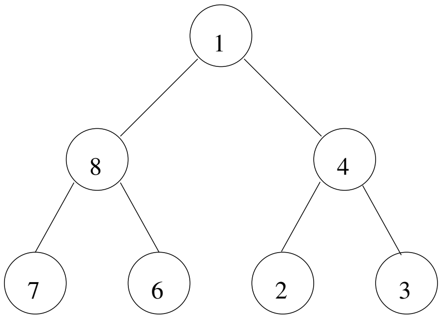
- A copy of the maximum element has been removed from the multiset stored in the heap, as required
- If the heap is still nonempty then a value has been moved from the deleted node to the root
- If the heap now has size at most one, or its size is at least two and the value at the root is larger than 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:



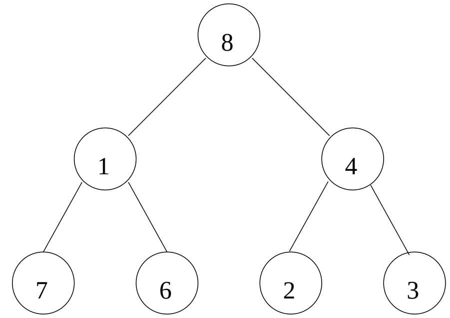
0	1	2	3	4	5	6	7
1	8	4	7	6	2	3	1

`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:



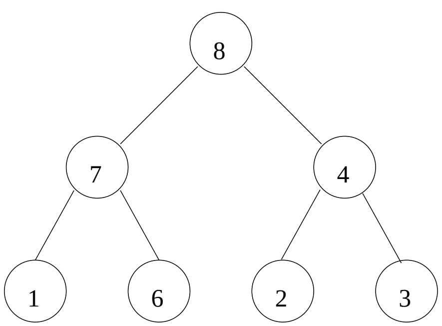
0	1	2	3	4	5	6	7
8	1	4	7	6	2	3	1

`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:



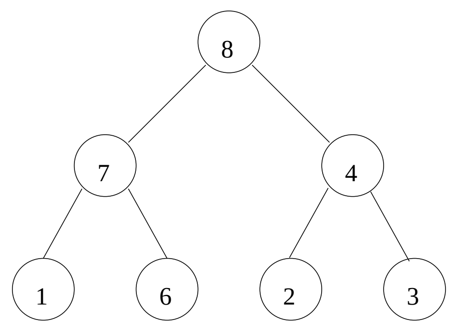
0	1	2	3	4	5	6	7
8	7	4	1	6	2	3	1

`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

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

Current value of `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 `heap-size(A)` entries of `A` are the multiset obtained from the original contents of the heap by deleting a copy of its largest value

b) `j` is an integer such that $0 \le \mathtt{j} < \mathtt{heap\text{-}size(A)}$

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

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

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

Step 2: Partial Correctness

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

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

On *termination* of this loop,

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

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

Justification:

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

Application of Loop Variant:

- inital value, and thus upper bound on the number of iterations, is $f(\mathtt{A},0) = 1 + \mathit{height}(0) = \lfloor \log \mathtt{heap\text{-}size(A)} \rfloor$
- 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 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:

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

Conclusion: The worst-case cost of this operation is

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