Math 182 Lecture 9

Chapter 5 Data Structures

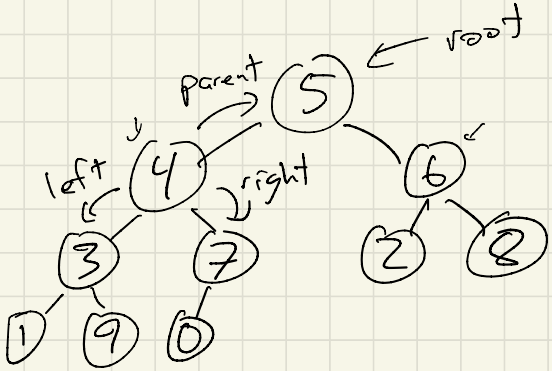
§5.1 Heaps

Recall an array is

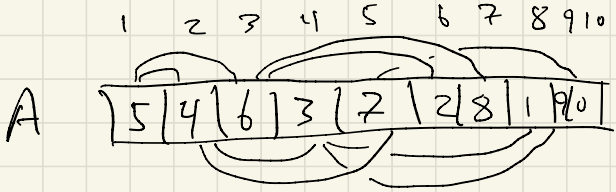
$A[1..n] = \langle a_1, a_2, \ldots, a_n \rangle$

arrays have an attribute A.length

A (binary) heap data structure is an array object which can be viewed as a nearly-complete binary tree.



Heap example as binary tree



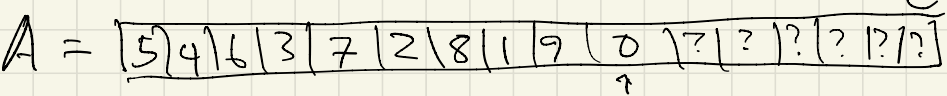
Heap as an array

Heaps have two attributes:

A.length

A.heap-size

A.length.



A.heap-size

```
PARENT(i)
1  return ⌊i/2⌋
LEFT(i)
1  return 2i
RIGHT(i)
1  return 2i + 1
```

In general
$0 \leq$ A.heap-size $\leq$ A.length.

These all run in $\Theta(1)$ constant time.

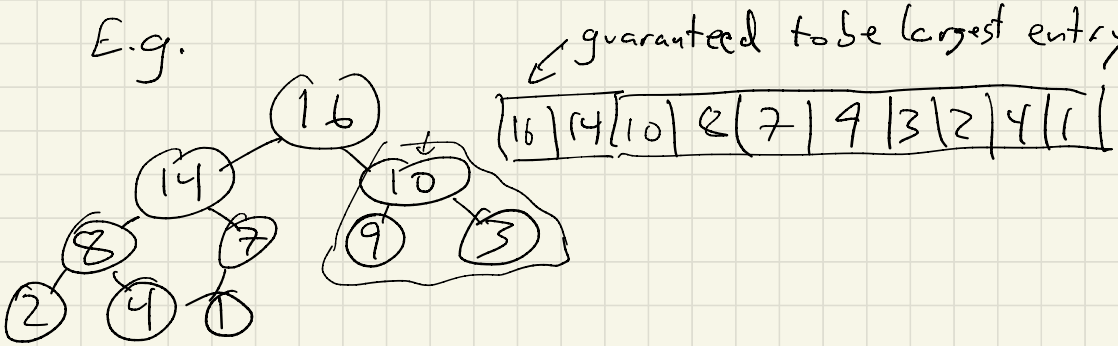
Max-heaps and min-heaps

Interested in heaps which satisfy a heap property

a max-heap is a heap w/ the max-heap property: for every node i which is not the root

$$A[\text{Parent}(i)] \geq A[i]$$

E.g.



is a max heap

a min-heap is a heap w/ the min-heap property: for every node i which is not the root

$$A[\text{Parent}(i)] \leq A[i]$$

Define height of a node in a heap to be longest path downward from node to some leaf. Define height of heap to be height of root. Height of heap of size n is $\Theta(\lg n)$

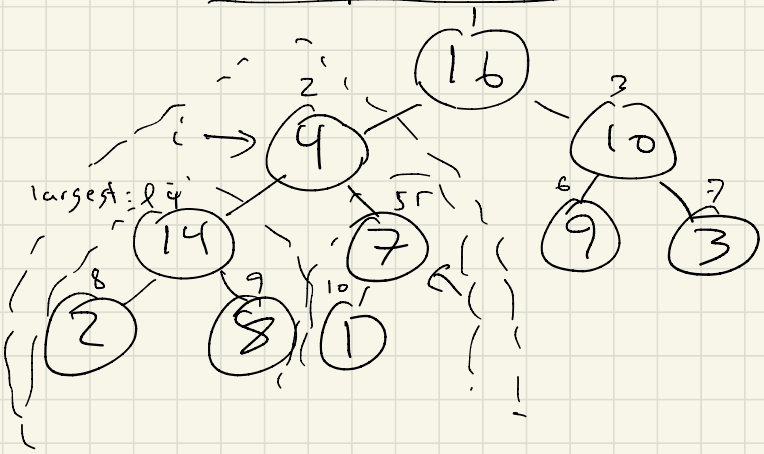
Maintaining the heap property

Sps A heap i node and Left(i) and Right(i) are roots of max-heaps

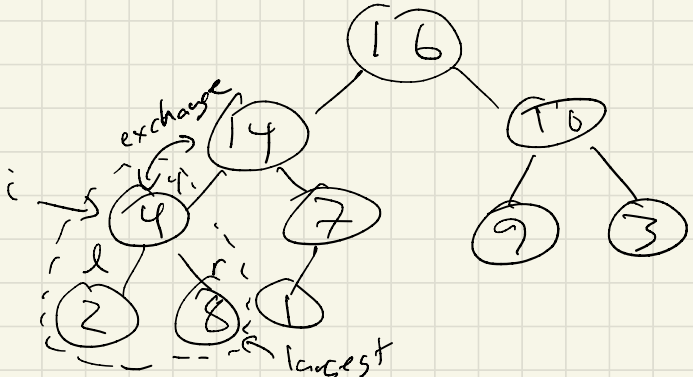
```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        exchange A[i] with A[largest]
10        MAX-HEAPIFY(A, largest)
```

1. In lines 1-7 we determine which of A[i], A[Left(i)], A[Right(i)] should take the place A[i].
2. In lines 8-9 swith A[i] w/ appropriate child if necessary
3. In Line 10, recursively call Max-Heapify on child which has original value of parent

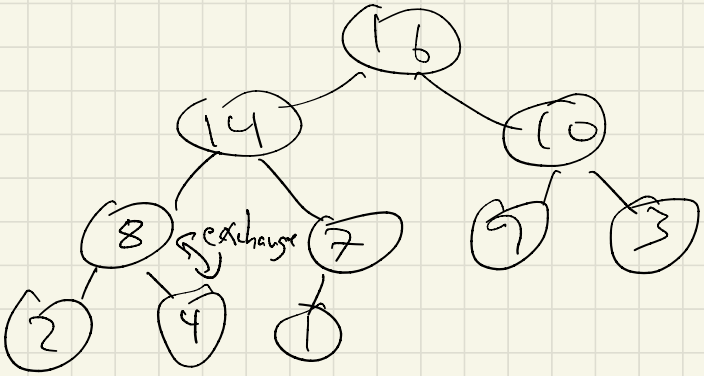
Example of Max-Heapify



Max-Heapify(A, 2)



Max-Heapify(A, 4)



Done ✓

Running time of Max-Heapify

Prop The running time of Max-Heapify on a heap of size n is $O(\lg n)$. Alternatively on a heap of height h is $O(h)$.

Proof: Lines 1-9 are $\Theta(1)$

Line 10 costs at most the T(size of bigger child's subtree)

T(left-child's subtree)

left child's subtree is as bad as possible when bottom row is half full $\Rightarrow$ size of left subtree $\leq 2n/3$

$\leq T(2n/3)$ so

$$T(n) \leq T(2n/3) + \Theta(1)$$

Master Theorem $\Rightarrow$

$$T(n) = O(\lg n).$$

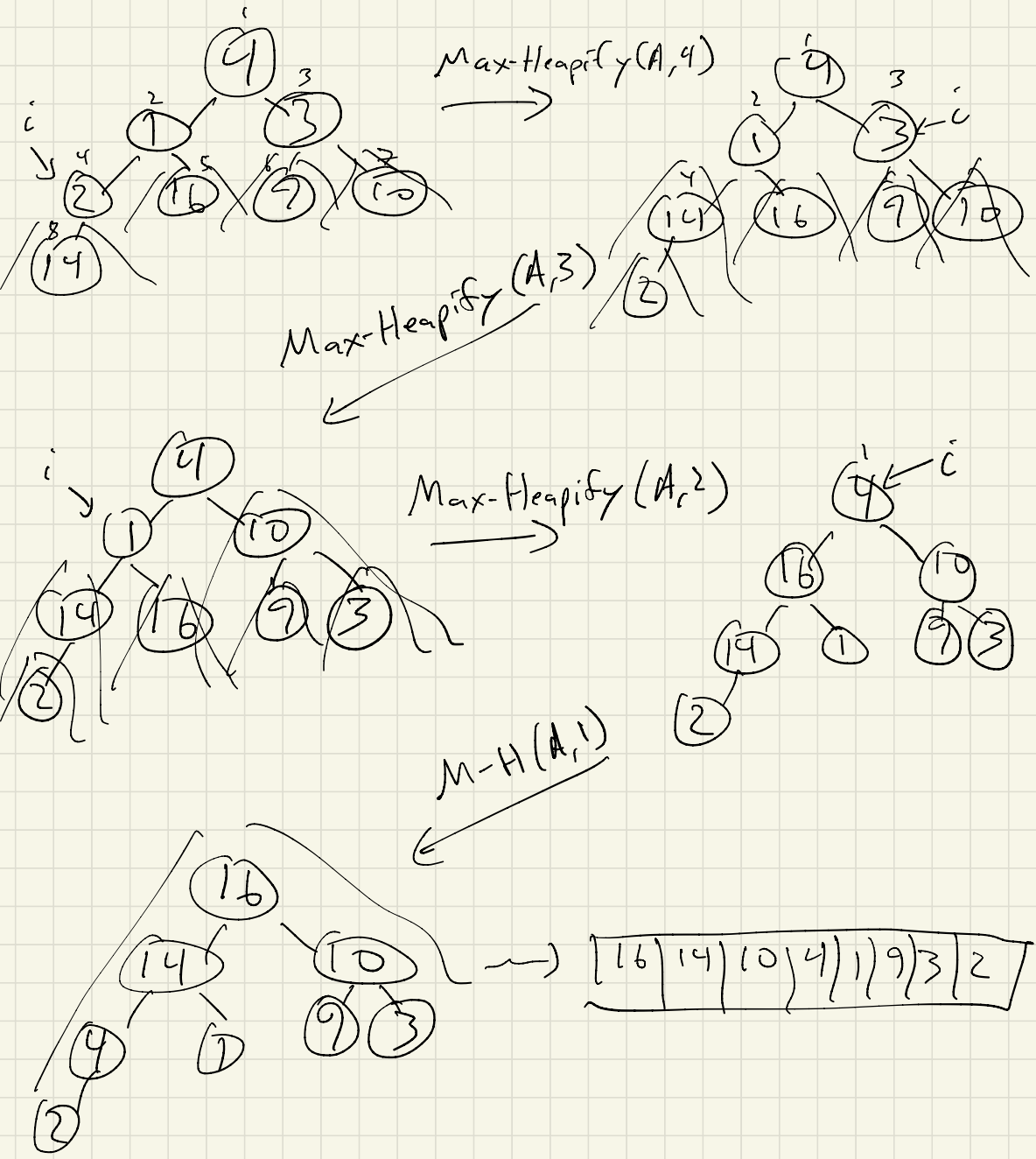
Building a heap

To make max-heap, call Build-Max-Heap(A)

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

1. Assumes we want to make all of A into a max-heap.
2. Fact: the nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ are leaves, thus already the roots of max-heaps.
3. In lines 2-3 iterate through non-leaves from bottom to top $\Rightarrow$ guarantees assumption that left/right children are roots of max-heaps always satisfied.

Example of Build-Max-Heap



Correctness of Build-Max-Heap

Claim: Given an array A, after Build-Max-Heap(A) runs, A is a max-heap.

Pf:

(Loop Invariant) At the end of each time line 2 runs, each node $i+1, i+2, \ldots, n$ is the root of a max heap.

(Init) Clear

(Maintenance) Uses statement of correctness of Max-Heapify

(Term.) Last value of i is $i = 0$ so nodes $1, \ldots, n$ are roots of max-heaps in particular 1 is root of max-heap. ∎

Running time of Build-Max-Heap

Loose upper bound:

Process $O(n)$ non-leaves

Each call to Max-Heapify takes $O(\lg n)$ time, so

running time is $O(n \lg n)$.

Thm: Running time is $T(n) = \Theta(n)$.

Proof: $\Omega(n)$ is clear from for loop.

For upper bound:

Fact: (HW4) ~~on a level of height h~~ there are almost $\lceil n/2^{h+1} \rceil$ many nodes of height h.

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

From HW1: $\sum_{j=0}^{m} j x^j = \frac{m x^{m+2} - (m+1) x^{m+1} + x}{(x-1)^2}$

w/ $x := 1/2$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} h (1/2)^h \le \sum_{h=0}^{\infty} h (1/2)^h = \frac{1/2}{(1/2 - 1)^2} = 2$$

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O(2n) = O(n). \quad \blacksquare$$

§5.2 Heapsort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Total $T(n)$: $O(n \lg n)$

1. First make A into a max-heap in line 1.
2. Since $A[1]$ now biggest element, put $A[1]$ at end by switching w/ $A[n]$.
3. Decrease $A.heap\text{-}size$ by 1 so we never move $A[n]$ again.
4. Call Max-Heapify$(A, 1)$ to arrange $A[1..n-1]$ max-heap
5. $A[1]$ now largest out of $A[1..n-1]$ so switch $A[1] \leftrightarrow A[n-1]$ thus $A[n-1..n]$ now in correct spots.

Example of Heapsort

Thm.

Remarks

1. Heapsort runs in $O(n \lg n)$ time just like Merge-Sort.
2. Heapsort sorts in-place just like Insertion-Sort, i.e., it takes only $\Theta(1)$ amount of extra memory.