CIS 313:
Intermediate Data Structure

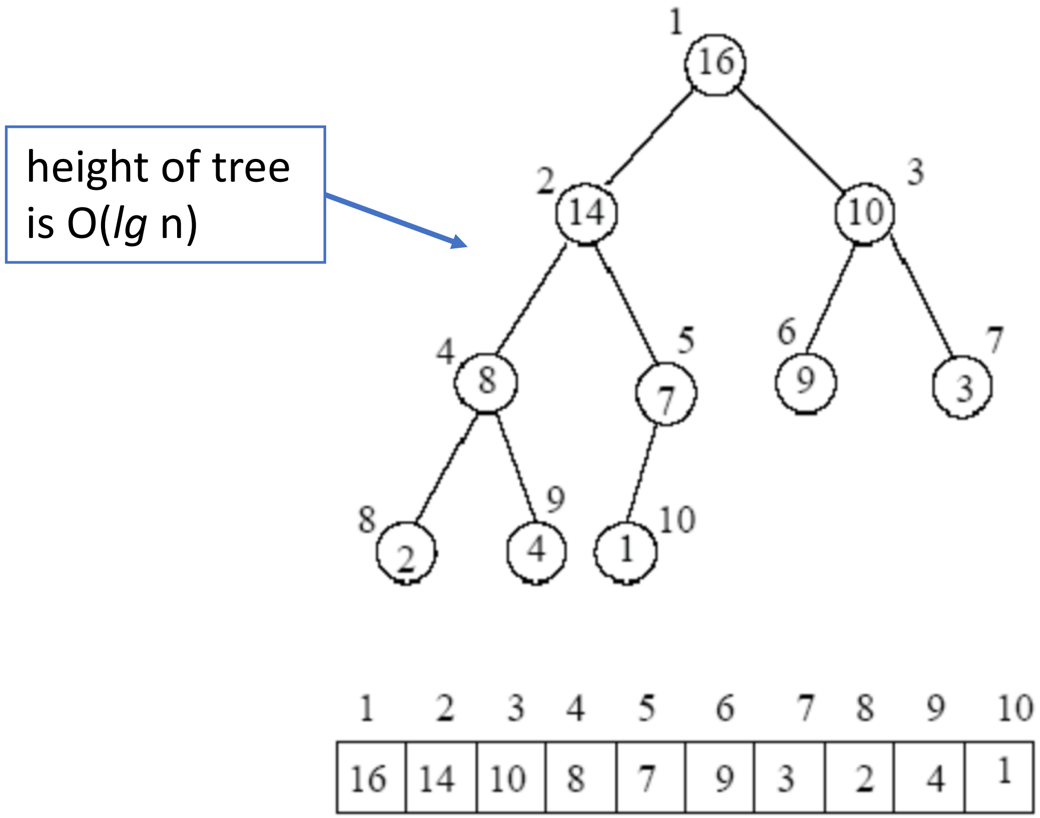
third slide

binary heap implementation of PQ

- most common implementation
- operations are O(*log* n)
- uses a binary tree structure
- except that the tree is stored in an array with no pointers
- it is an *implicit* tree, children and parents inferred from location in array

- PQSort becomes *heapsort*

binary heap



- stored in array
- item located in postion i
  - parent in location $\lfloor i/2 \rfloor$
  - left child in position $2i$
  - right child in postion $2i + 1$
- tree is complete
  - all nodes have two children, except maybe parent of “last” one
- tree maintains heap property
  - value stored at location i is greater than or equal to values stored in both its children
- fact: a binary heap with n elements has the height of $\lfloor \lg n \rfloor$ (why?)

binary heap insertion

- put new value x at end of array, extending its size by 1
- value x is now viewed as being at the bottom of the tree
- if x violates heap property (if larger than parent), swap with parent
- repeat until no violation
- time is proportional to height of tree, which is O(*lg* n)

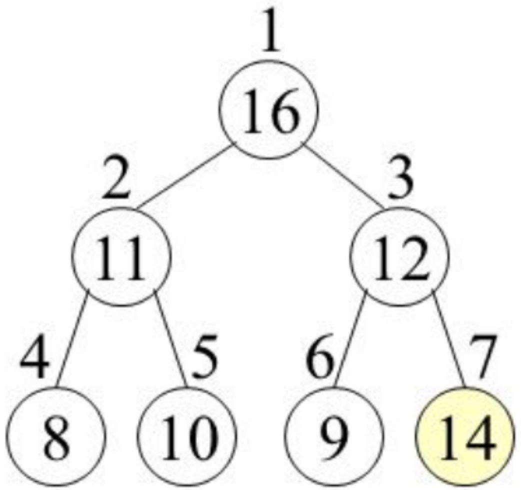
- text handles this differently, they insert $-\infty$ and then use heap-increase-key to the new value

pseudo-code for insert

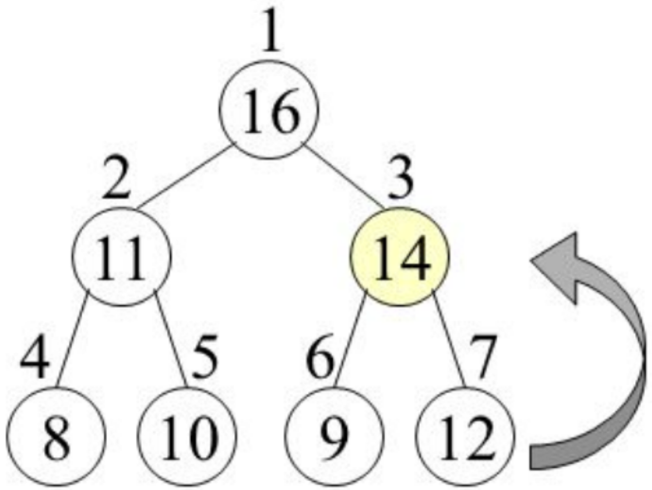
```
insert(x):

heapsize++
A[heapsize]=x

i = heapsize
while i>1 and A[i]>A[parent(i)]
     swap A[i] and A[parent(i)]
     i = parent(i)
```

sometimes called “sift-up” or “bubble-up”

Binary Heap : Insert Operation



viewed as a binary tree



viewed as a binary tree

1	2	3	4	5	6	7
16	11	12	8	10	9	14

viewed as an array

1	2	3	4	5	6	7
16	11	14	8	10	9	12

viewed as an array

heap extract-max (deletion)

- similar but element moves down
- idea: remove and return root (in location 1 of the tree)
- move rightmost element into that empty location …
- … and reduce the heapsize
- tree shape is maintained but root location may violate heap property
- note: rest of tree still has heap property
- swap node with *larger* (why) of it’s children
- repeat while heap property violated until leaf hit
- called “sift-down” or “bubble-down”

text algorithm

```
MAX-HEAPIFY(A, i)
    // Input: A: an array where the left and right children of i root heaps (but i may not), i: an array index
    // Output: A modified so that i roots a heap
    // Running Time: O(log n) where n = heap-size[A] − i
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

first attempt at sorting

1. for each element x, *insert* x into a heap
   - time per insert O(*lg* n), total O(n *lg* n)
   - this can be made much faster

`BUILDHEAP` uses deletion idea to get linear overall time

2. while the heap is not empty, *extract-max*
   - output is a sorted list (reversed)
   - each extract-max is O(*lg* n), total O(n *lg* n)
   - cannot be made faster

buildheap code

```
BUILD-MAX-HEAP(A)
    // Input: A: an (unsorted) array
    // Output: A modified to represent a heap.
    // Running Time: O(n) where n = length[A]
    heap-size[A] ← length[A]
    for i ← ⌊length[A]/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

correctness
- idea sort of clear, build heaps bottom up
- text uses loop invariant!!

time analysis
if tree has height H=lgn
- all nodes at level k take time H-k to sift down
- there are 2^k nodes at level k
- total time is $\sum_0^H 2^k (H-k)$
- can show this is at most $2n$

grinding through the time bound

$$\sum_{k=0}^{H} 2^k (H-k) = 2^H \sum_{k=0}^{H} (2^k/2^H)(H-k)$$

$$= n \cdot \sum_{k=0}^{H} \frac{1}{2^{H-k}} (H-k)$$

$$= n \cdot \sum_{i=0}^{H} \frac{i}{2^i} \le n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} = 2 \cdot n$$

$2^H \approx 2^{\log_2 n} = n$

re-index

why just 2?

- mentioned but not proved in appendix
- “fun” to derive
- can also take derivative of $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$

now heapsort

```
HEAP-SORT(A)
   // Input: A: an (unsorted) array
   // Output: A modified to be sorted from smallest to largest
   // Running Time: O(n log n) where n = length[A]
1  BUILD-MAX-HEAP(A)
2  for i = length[A] downto 2
3        exchange A[1] and A[i]
4        heap-size[A] ← heap-size[A] − 1
5        MAX-HEAPIFY(A, 1)
```

step 1: $\Theta(n)$ time

steps 2-5: $\Theta(n \log n)$ time

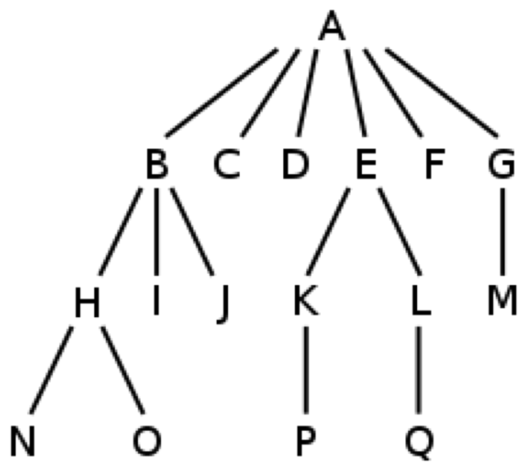
other heap operation: increase-key

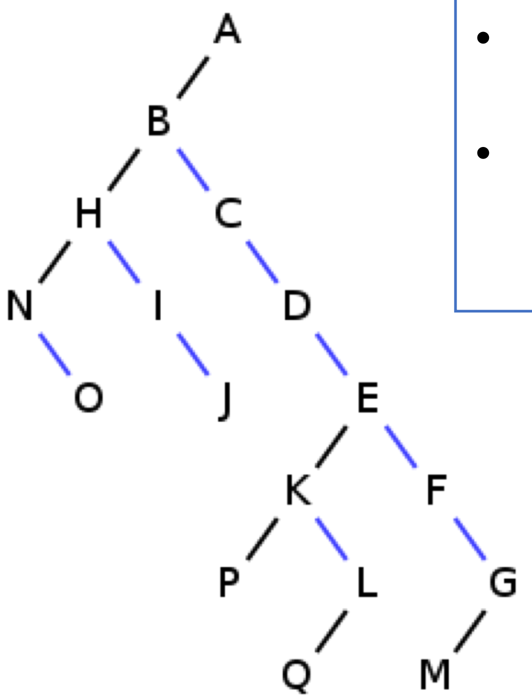
- an item can be increased in O($\lg n$) time
- after the increase, it would need to be sifted up as in the insert method
- the same applies to the decrease-key operation in a min heap
- this operation is a crucial step in Dijkstra’s method (shortest path) and Prim’s method (minimum spanning tree)
- it can be implemented in O(1) amortized time using Fibonacci heaps

summary

Procedure	Binary heap (worst-case)
MAKE-HEAP	$\Theta(1)$
INSERT	$\Theta(\lg n)$
MINIMUM	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$
UNION	$\Theta(n)$
DECREASE-KEY	$\Theta(\lg n)$
DELETE	$\Theta(\lg n)$

small digression: ordered trees





ordered tree:

- tree has designated root
- a node can have any number of children
- if a node has k children, they are ordered
  - 1st child, 2nd child, …, kth child
- good representation involves two pointers per node:
  - first- child and next-sibling
  - so the children of a node are in a linked list