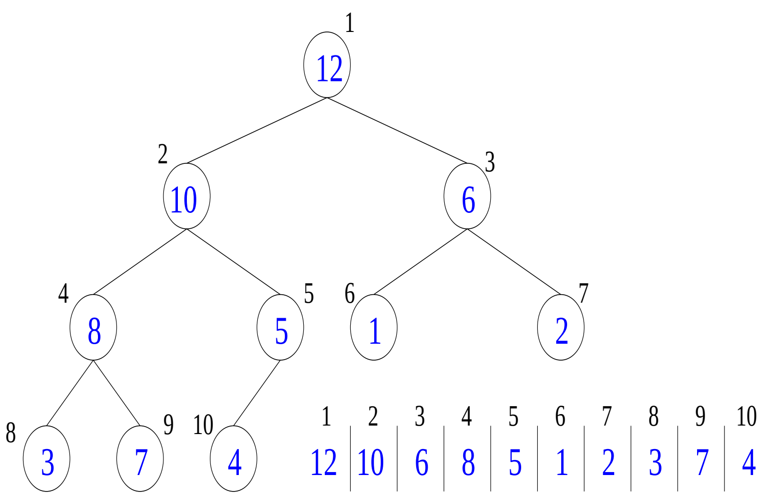
Heapsort

In-place sort.

Running time: O(n lg n)

Heaps



Heap: An array A representing a complete binary tree for HeapSize(A) elements satisfying the heap property:

____

for every node i except the root node.

Because the array may hold more numbers than are contained in the heap, HeapSize(A) $\leq$ length(A) (heap insertion and deletion)

- **parent(i)** $= \lfloor i/2 \rfloor$

- **left(i)** = ___ ; left child

- **right(i)** = _________ ; right child

Running Times

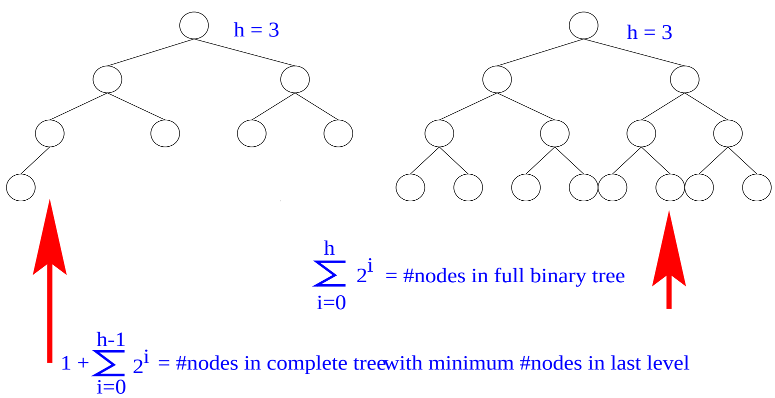
Running times depend on height of the tree — the height of a node in a tree is the number of edges in the longest simple path from the node to a leaf.

- height(node 8) = _
- height(node 3) = _
- height(node 1) = _

The height of the tree is the ___________________________________.
Operations on heap proportional to height.

Exercise 7.1-2

Show that an n-element heap has height $\lfloor \lg n \rfloor$



$$1 + \sum_{i=0}^{h-1} 2^i \leq n \leq \sum_{i=0}^{h} 2^i < \sum_{i=0}^{h} 2^i + 1$$

$$1 + \frac{2^h - 1}{2 - 1} \leq n < \frac{2^{h+1} - 1}{2 - 1} + 1$$

Simplifying a geometric series ($\Sigma_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$)

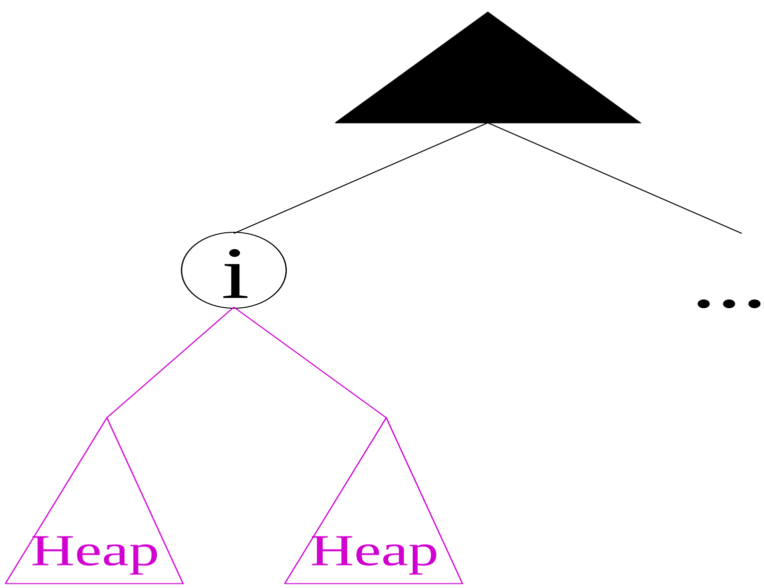
$$2^h \leq n < 2^{h+1}$$

$$h \leq \lg n < h + 1$$

Since h is an integer, h = $\lfloor$ lg n $\rfloor$.

Maintaining the Heap Property

Heapify(A,i)



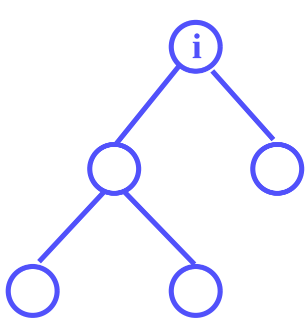
Notice that if A[left(i)] > A[i] or if A[right(i)] > A[i] or both (take largest) then swap and recurse until recursion bottoms out at bottom of heap.

Pseudocode

```
Heapify(A,i)
    l = left(i)
    r = right(i)
    if l ≤ HeapSize(A) and A[l] > A[i]
    then largest = l
    else largest = i
    if r ≤ HeapSize(A) and A[r] > A[largest]
    then largest = r
    if largest ≠ i
    then swap(A[largest], A[i])
         Heapify(A, largest)
```

If the parent is smaller than either of its children, does it matter whether we swap the parent with the larger or the smaller of the children? ______________

Recursive Analysis



$$n = 5,\ n_{subtree} = \lfloor 2n/3 \rfloor = 3$$

Note that this expression has the maximum value when the lowest level of the heap is exactly half full.

$$T(n) = T(\lfloor 2n/3 \rfloor) + \Theta(1)$$

Master: a = 1, b = 3/2, f(n) = $\Theta(1) = \Theta(n^{log_{3/2}1}) = \Theta(n^0) = \Theta(1)$, Case _

T(n) = $\Theta(n^{log_{3/2}1} lgn) = \Theta(lgn)$.

BuildHeap

```
BuildHeap(A)
    HeapSize(A) = length(A)
    for i = ⌊ length(A)/2 ⌋ downto 1
        Heapify(A,i)
```

A[($\lfloor$ n/2 $\rfloor$ + 1) ... n] are leaves and heaps.
Click on mouse to advance to next frame.

Analysis

There are O(n) calls to Heapify, thus BuildHeap is O(nlgn), which is an upper bound but not a tight bound (o(nlgn)).

The tight upper bound is O(nh).

Notice that the height h changes as the heap is being built.

Note: There can be at most $\lceil n/2^{h+1} \rceil$ nodes of height h in an n-element heap.

Thus Heapify = ______ for nodes of height h.

Analysis

From this result we can analyze the run time of BuildHeap.

$$\sum_{h=0}^{\lfloor lgn \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) \; = \; O(n * 1/2 \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h})$$

Note that $\Sigma_{k=0}^{\infty} x^k = \frac{1}{1-x}$, for $\mid x \mid < 1$. The derivative of both sides, $\frac{d}{dx}(\Sigma_{k=0}^{\infty} x^k) = \frac{d}{dx}(\frac{1}{1-x})$, is equal to $\Sigma_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$.

Multiplying both sides of the equivalence by x we get
$\Sigma_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$.
In our case k = h and x = 1/2.
Thus $\Sigma_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$, x = 1/2.

Thus the run time is ______________________________.

Heapsort

Heapsort(A)
1 BuildHeap(A)

```
    for i = length(A) downto 2
        swap(A[1], A[i])
        HeapSize(A) = HeapSize(A) - 1
        Heapify(A, 1)
```

Click on mouse to advance to next frame.

Summary

BuildHeap: _____

Heapify: ____

Heapsort: _________

Priority Queues

A **priority queue** is a data structure for maintaining a set S of elements, each with an associated key value.

Operations:

Max(S): returns element of S with largest key

ExtractMax(S): removes and returns element with largest key from S

Insert(S,x): inserts x into S (S = S $\cup$ $\{x\}$)

Application: Job Scheduling

Support insertion of prioritized jobs in queue.

Support extraction of highest priority job from queue.

Using heaps:

Max(S) = A[1], $\Theta(1)$

ExtractMax O(lgn), First 3 lines $\Theta(1)$, fourth line O(lgn)

```
ExtractMax(S)
1       max = A[1]
2       A[1] = A[HeapSize(A)]
3       HeapSize(A) = HeapSize(A) - 1
4       Heapify(A,1)
5       return max
```

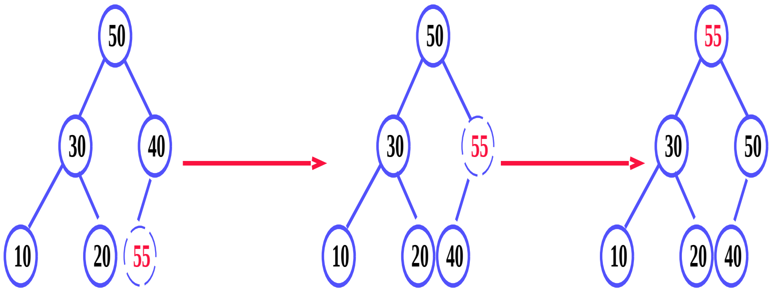
This corresponds roughly to one iteration of HeapSort.

Insert

While loop iterates through height of heap and is thus O(lgn)

```
Insert(A, key)
1       HeapSize(A) = HeapSize(A) + 1
2       i = HeapSize(A)
3       while i > 1 and A[Parent(i)] < key
4           A[i] = A[Parent(i)]
```

```
        i = Parent(i)
    A[i] = key
```



Quicksort

In-place, $\Theta(n^2)$ worst case.
O(n lg n) average case with small constant factors.

Description

Divide-and-Conquer

__________ Partition A[p..r] into two non-empty subsequences A[p..q] and A[q+1..r] such that each element of A[p..q] is $\leq$ each element of A[q+1..r].

____________ Sort A[p..q] and A[q+1..r] by recursive calls to Quicksort.

____________ Trivial, arrays are sorted in place.

Pseudocode

```
Quicksort(A,p,r)
    if p < r
    then q = Partition(A,p,r)
         Quicksort(A,p,q)
         Quicksort(A,q+1,r)
```

Initial Call: Quicksort(A,1,length(A))

Partitioning the Array

1. Pick ______ as the pivot
2. Move from _______ to _____________ looking for an element $\leq$ pivot
3. Move from ______ to _____________ looking for an element $\geq$ pivot
4. Swap the two elements
5. Repeat until pointers cross

Partition

Click on mouse to advance to next frame.

```
Partition(A,p,r)
    x = A[p]
```

```
i = p - 1
j = r + 1
while TRUE
    repeat
        j = j - 1
    until A[j] ≤ x
    repeat
        i = i + 1
     until A[i] ≥ x
 if i < j
 then swap(A[i], A[j])
 else return j
```

Partition

________________________________ (pivot is the median of the sequence) leads to $\Theta(nlgn)$ like Merge Sort.

___________________________________ (maximally bad when array already sorted) leads to $\Theta(n^2)$ like Insertion Sort.

Worst Case

Partition always yields subarrays of size n-1 and 1.

T(n) = T(n-1) + $\Theta(n)$

$$= T(n-2) + \Theta(n-1) + \Theta(n), \quad T(1) = \Theta(1)$$

$$= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n)$$

The ith term is T(n-i) and the boundary case is i=n-1. The summation is

$$= \sum_{k=1}^{n} \Theta(k)$$

We know that $\Sigma_{k=1}^{n} \Theta(f(k)) = \Theta(\Sigma_{k=1}^{n} f(k))$. Thus summation is then $= \Theta(\Sigma_{k=1}^{n} k) = \Theta(n^2)$.

Note that Insertion Sort is O(n) in this same case (already sorted).

Best Case

Partition yields subarrays of size n/2 each.

T(n) = 2T(n/2) + $\Theta(n)$

a=2, b=2, f(n) = $\Theta(n) = \Theta(n^{log_2 2}) = \Theta(n^1)$, Case 2

T(n) = $\Theta(nlgn)$

Average Case

$\Theta(nlgn)$

Worst Case Revisited

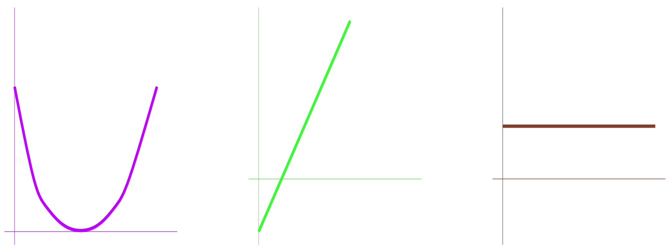
Assume we do not know what the worst partition is.

T(n) = $\max_{1 \leq q \leq n-1}$(T(q) + T(n-q)) + $\Theta(n)$.

By the substitution method (since we know the answer), try T(n) $\leq cn^2$.

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n)$$
$$= c * \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$$

$\frac{d}{dq}(q^2 + (n-q)^2)$ = 2q - 2(n-q) = 2q - 2n + 2q = 4q - 2n
$\frac{d}{dq}(4q - 2n) = 4.$
For q=1: $1^2 + (n-1)^2 = n^2 - 2n + 2.$
For q=n-1: $(n-1)^2 + 1^2 = n^2 - 2n + 2.$



Original Equation   First Derivative   Second Derivative

T(n) $\le$ cn^2 - 2c(n - 1) + $\Theta(n) \le cn^2$. Picking a large enough c,
T(n) = $\Theta(n^2)$

Summary of Comparison Sorts

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Merge Sort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$	Requires Memor
Heapsort	$\Theta(nlgn)$	$\Theta(nlgn)$	$\Theta(nlgn)$	Large constants
Quicksort	$\Theta(n^2)$	$\Theta(nlgn)$	$\Theta(nlgn)$	Small constants

---

Applications