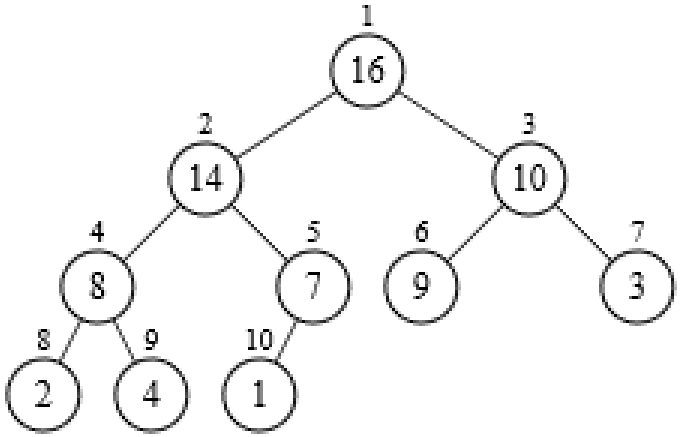
# CSE 5311 Fall 2005 Week 4

Devendra Patel
Subhesh Pradhan

# Heaps

- Heap $A$ (*not* garbage-collected storage) is a nearly complete binary tree.
  - ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
  - ***Height*** of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array $A$.
  - Root of tree is $A[1]$.
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
  - Left child of $A[i] = A[2i]$.
  - Right child of $A[i] = A[2i + 1]$.
  - Computing is fast with binary representation implementation.



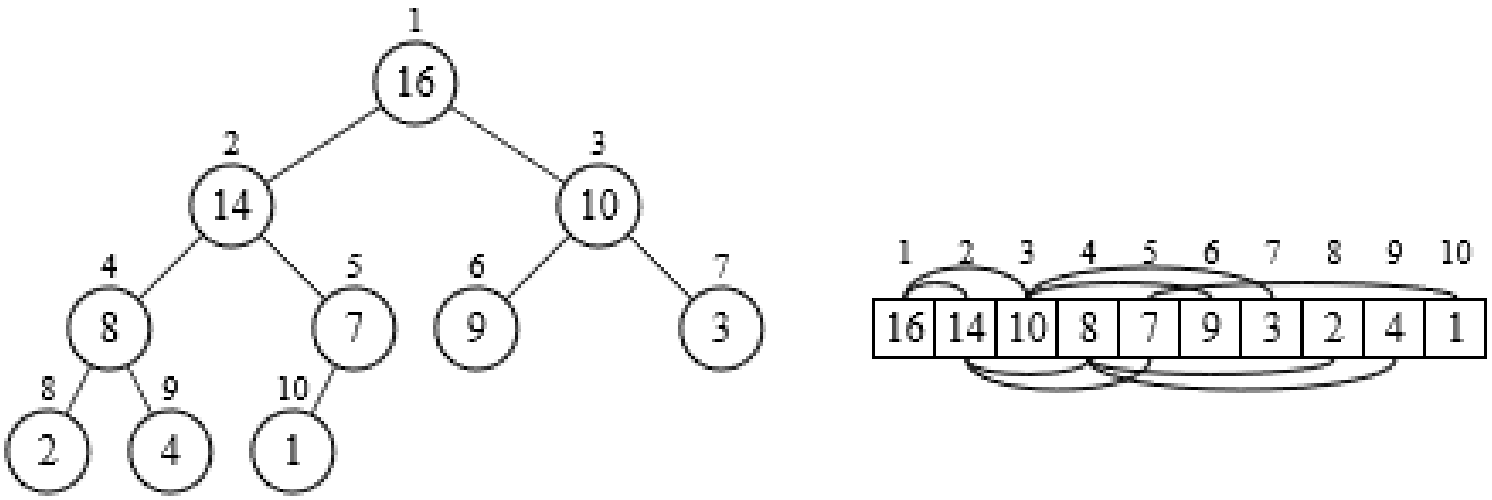
# Caution!

Do not confuse the **height** of a node (longest distance from a leaf) with its **depth** (distance from the root)

# Heap Property

- A max-heap is a nearly complete binary tree with the condition that every node (except the root) must have a value greater than or equal to its children.

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes $i$, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes $i$, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.




Max-heap Example

# HEAPIFY (Keeping the Heap Property)

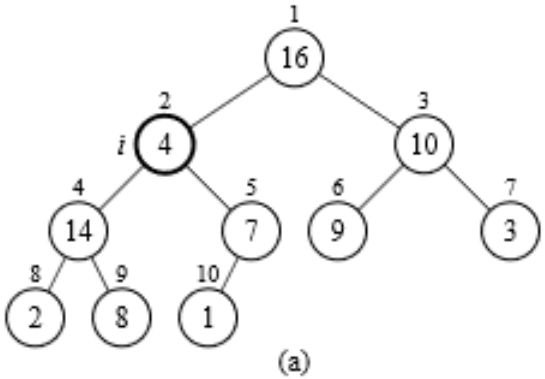
MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of $i$ are max-heaps.
- After MAX-HEAPIFY, subtree rooted at $i$ is a max-heap.

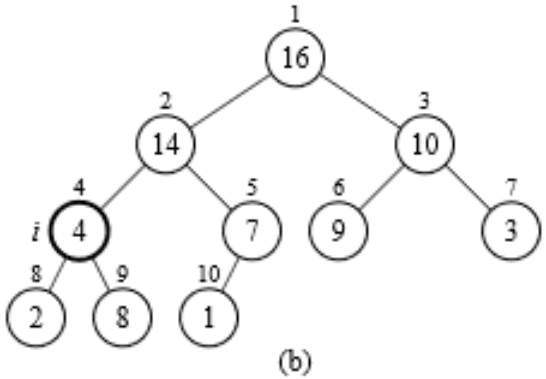
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
       largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
       largest = r
  if largest ≠ i
       exchange A[i] with A[largest]
       MAX-HEAPIFY(A, largest, n)
```

# HEAPIFY (How it works)

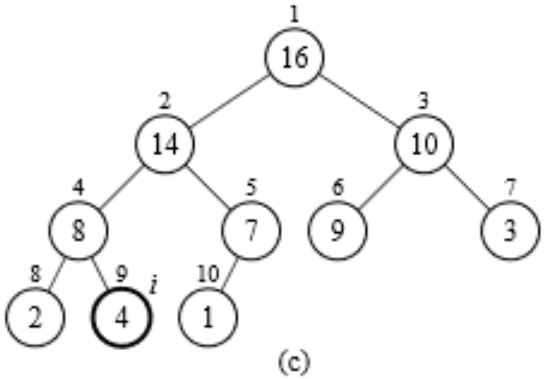
- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at $i$ is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.



(a)

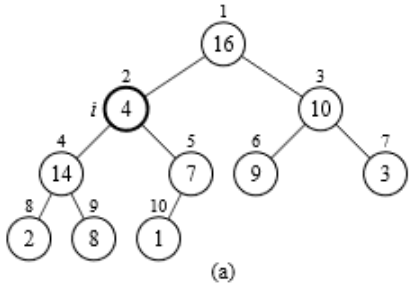


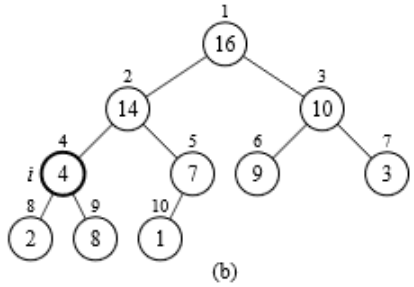
(b)

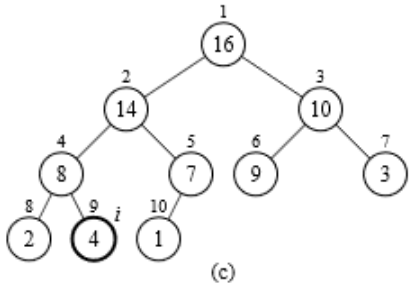


(c)

# HEAPIFY (Time)




- Time
  $O(\lg n)$

- Analysis
  Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

# Building a Heap

The following procedure, given an unordered array, will produce a max-heap.

BUILD-MAX-HEAP$(A, n)$

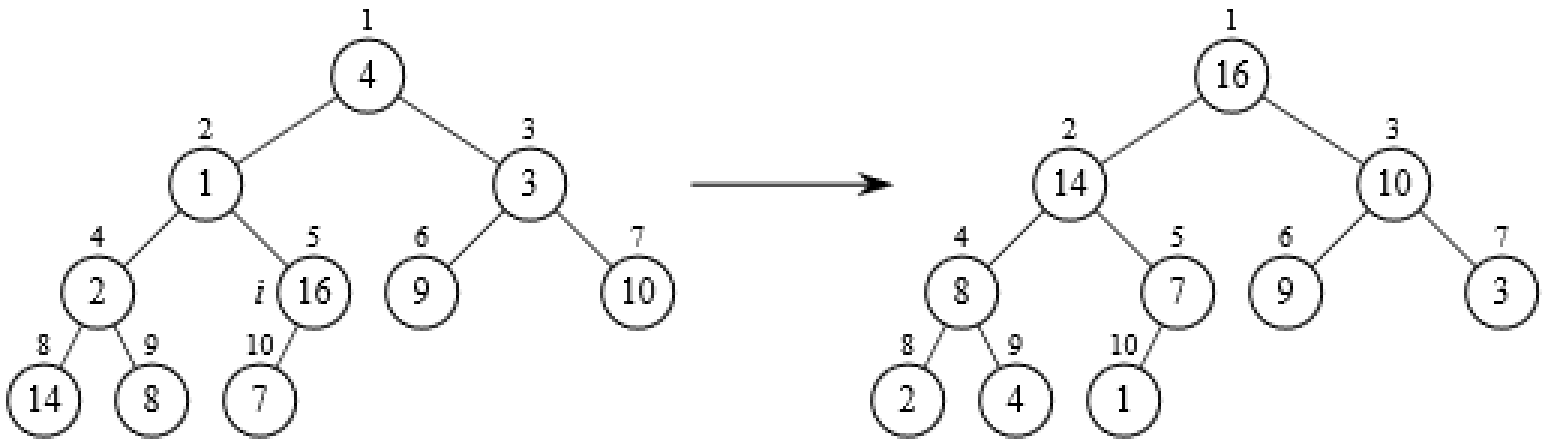
**for** $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY$(A, i, n)$

***Example***

- $i$ starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



# Analysis of BUILD-MAX-HEAP

- Simple Bound:
  - $O(n)$ calls to MAX-HEAPIFY
  - MAX-HEAPIFY takes $O(\lg n)$ time
  - Thus, $T(n) = O(n \lg n)$
- Tight Bound
  The running time of BUILD-MAX-HEAP is $O(n)$ (long proof, see the end).

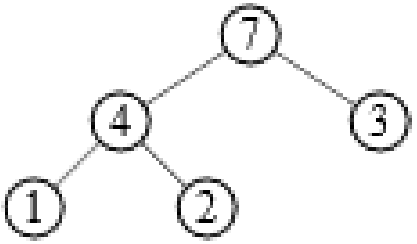
# The Heapsort Algorithm

Given an input array, the heapsort algorithm acts as follows:

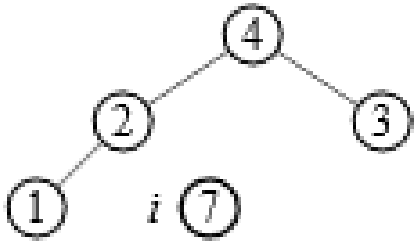
- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

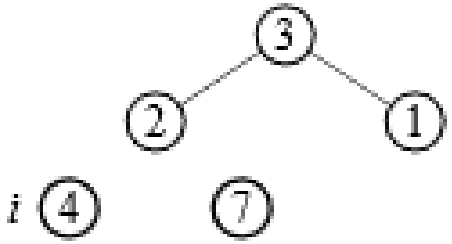
# Example for Heapsort Algorithm



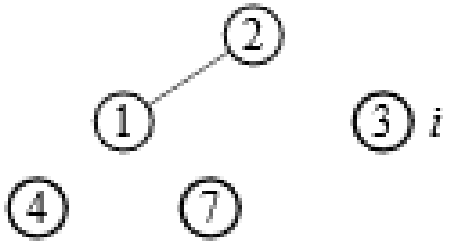

(a)



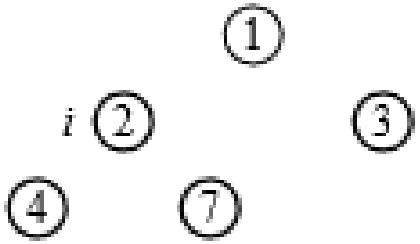

(b)




(c)




(d)




(e)

| $A$ | 1 | 2 | 3 | 4 | 7 |
|---|---|---|---|---|---|

# Analysis of Heapsort Algorithm

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

# Priority Queue

- Maintains a dynamic set $S$ of elements.
- Each set element has a *key*—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT$(S, x)$: inserts element $x$ into set $S$.
  - MAXIMUM$(S)$: returns element of $S$ with largest key.
  - EXTRACT-MAX$(S)$: removes and returns element of $S$ with largest key.
  - INCREASE-KEY$(S, x, k)$: increases value of element $x$'s key to $k$. Assume $k \geq x$'s current key value.
- Min-priority queue supports similar operations:
  - INSERT$(S, x)$: inserts element $x$ into set $S$.
  - MINIMUM$(S)$: returns element of $S$ with smallest key.
  - EXTRACT-MIN$(S)$: removes and returns element of $S$ with smallest key.
  - DECREASE-KEY$(S, x, k)$: decreases value of element $x$'s key to $k$. Assume $k \leq x$'s current key value.

# Finding the Maximum Element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM($A$)
  **return** $A[1]$

***Time***

$\Theta(1)$.

# Extracting Maximum Element (or Delete-Max)

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

```
HEAP-EXTRACT-MAX(A, n)
  if n < 1
        error "heap underflow"
  max = A[1]
  A[1] = A[n]
  n = n - 1
  MAX-HEAPIFY(A, 1, n)        // remakes heap
  return max
```

***Time***

$O(\lg n)$.

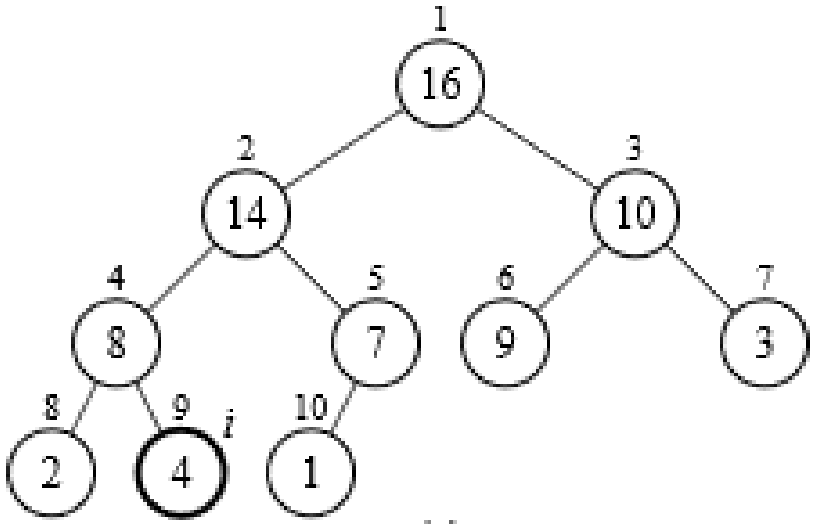
***Analysis***

Constant-time assignments plus time for MAX-HEAPIFY.

# Extracting Maximum Elem (Ex.)

Run HEAP-EXTRACT-MAX on first heap example.

- Take 16 out of node 1.
- Move 1 from node 10 to node 1.
- Erase node 10.
- MAX-HEAPIFY from the root to preserve max-heap property.
- Note that successive extractions will remove items in reverse sorted order.



# Increasing Key Value

Given set $S$, element $x$, and new key value $k$:

- Make sure $k \geq x$'s current key.
- Update $x$'s key value to $k$.
- Traverse the tree upward comparing $x$ to its parent and swapping keys if necessary, until $x$'s key is smaller than its parent's key.

HEAP-INCREASE-KEY$(A, i, key)$

**if** $key < A[i]$
  **error** "new key is smaller than current key"
$A[i] = key$
**while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$

***Time***

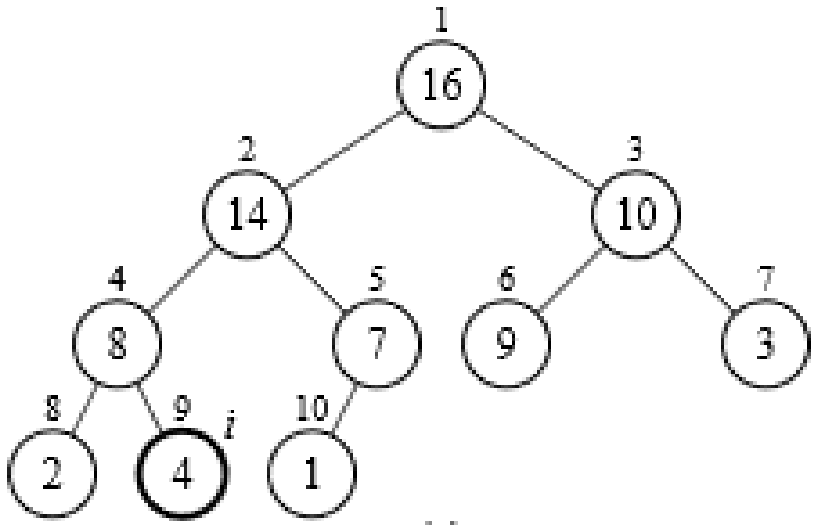
$O(\lg n)$.

***Analysis***

Upward path from node $i$ has length $O(\lg n)$ in an $n$-element heap.

# Increasing Key Value (Ex.)

Increase key of node 9 in first heap example to have value 15. Exchange keys of nodes 4 and 9, then of nodes 2 and 4.



# Inserting into the Heap

Given a key $k$ to insert into the heap:

- Increment the heap size.
- Insert a new node in the last position in the heap, with key $-\infty$.
- Increase the $-\infty$ key to $k$ using the HEAP-INCREASE-KEY procedure defined above.

Note: Again, the parameter $n$ is passed by reference, not by value.

MAX-HEAP-INSERT$(A, key, n)$

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY$(A, n, key)$

***Time***

$O(\lg n)$.

***Analysis***

Constant time assignments + time for HEAP-INCREASE-KEY.

# Building a Heap

The following procedure, given an unordered array, will produce a max-heap.

BUILD-MAX-HEAP$(A, n)$

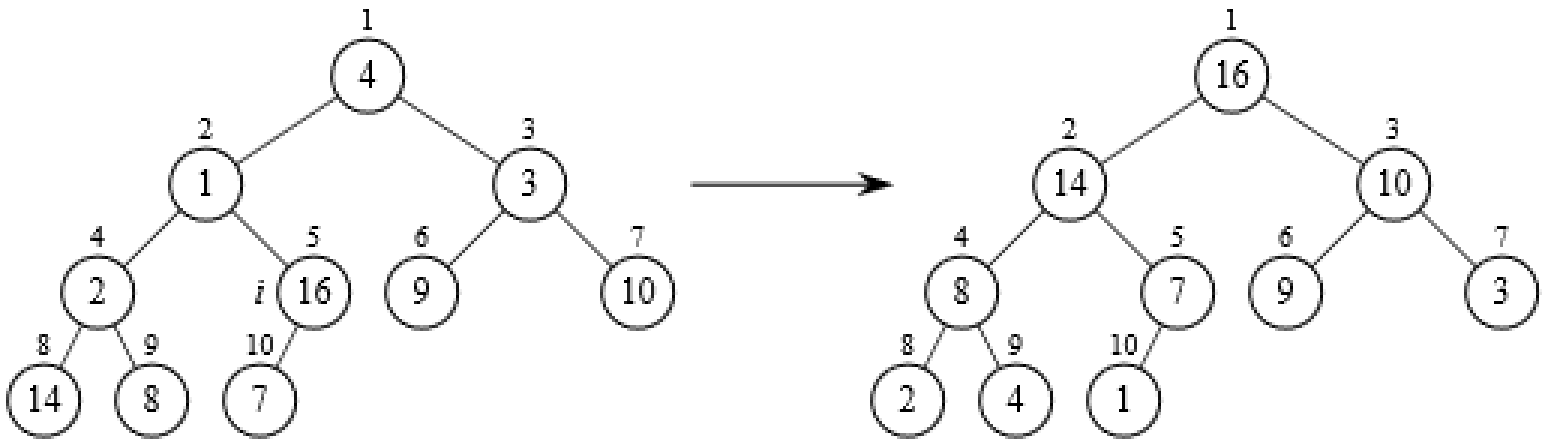
**for** $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY$(A, i, n)$

***Example***

- $i$ starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

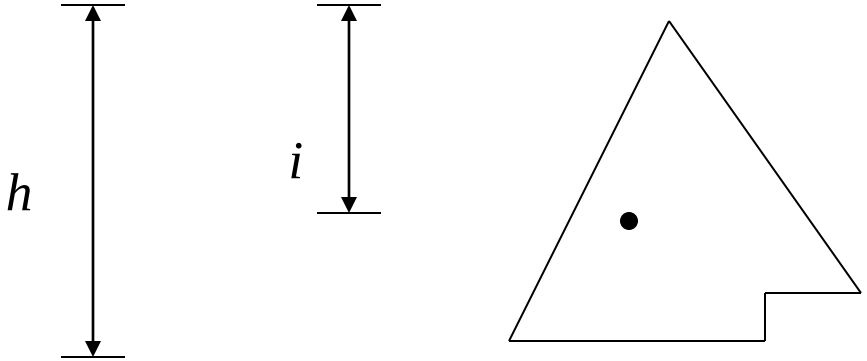


# Analysis of BUILD-MAX-HEAP

- Simple Bound:
  - $O(n)$ calls to MAX-HEAPIFY
  - MAX-HEAPIFY takes $O(\lg n)$ time
  - Thus, $T(n) = O(n \lg n)$
- Tight Bound
  The running time of BUILD-MAX-HEAP is $O(n)$ (long proof, see the book).

# The Tight Bound

- We represent heap in the following manner




For nodes at level $i$ , there are $2^i$ nodes . And the work is done for $h$-$i$ levels

Total work done = $\sum_{i=0}^{h = \lg n} 2^i \text{ x } (h - i)$

$= \sum_{i=0}^{h = \lg n} 2i \text{ x } (\lg n - i)$ (taking $h = \lg n$)

# Tight Bound of Build_Heap

Substituting $j = \lg n - i$ we get,

Total work done $= \sum_{j=\lg n}^{0} (2^{\lg n - j})\, j$

$$= \sum_{j=0}^{\lg n} (2^{\lg n}/2^{j}) j$$

$$= n \sum_{j=0}^{\lg n} \frac{j}{2^{j}}$$

$$= \Theta(n)$$

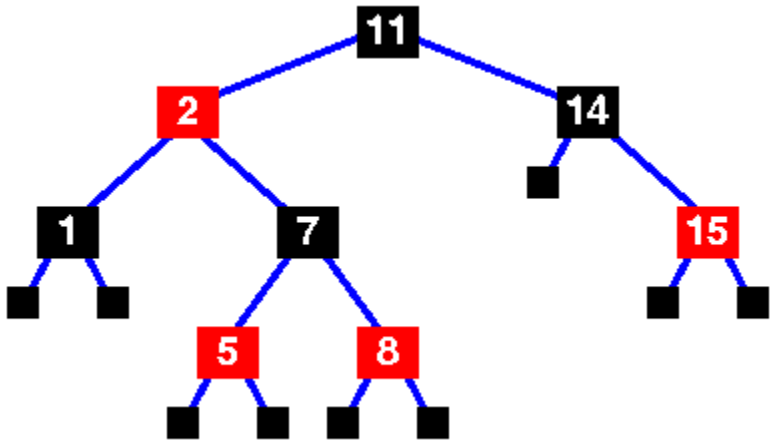
Thus running time of $\text{Build\_Heap} = \Theta(n)$

Note: Here we use the following series result by setting $x = 1/2$

$\sum_{k=0}^{\infty} k x^{k} = \frac{x}{(1-x)^{2}}$ for $|x| < 1$

# Red Black Tree

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



# Home Work

- 6.1-1, 6.1-2, 6.1-4, 6.1-5
- 6.2-1, 6.2-2, 6.2-3
- 6.3-1, 6.3-2
- 6.4-1
- 6.5-1, 6.5-2, 6.5-3