COT 6405 Introduction to Theory of Algorithms

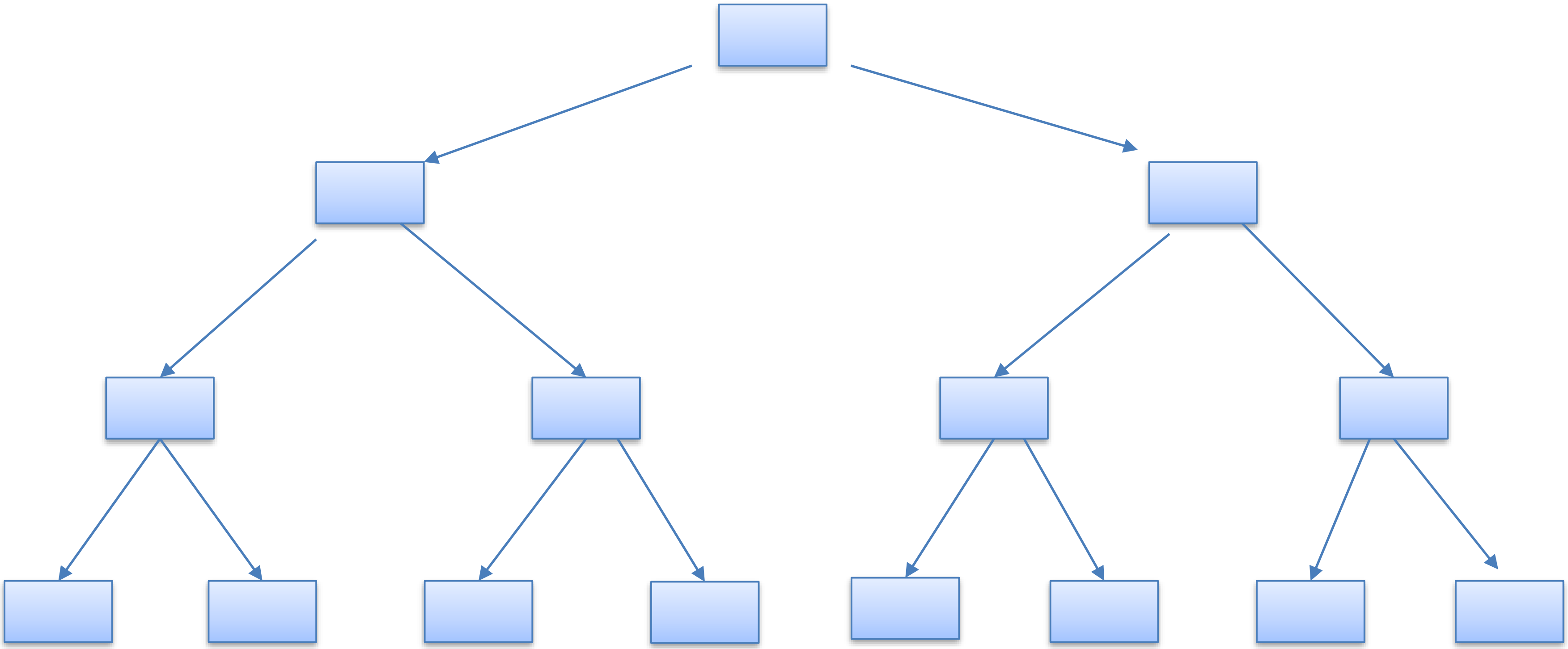
Topic 6. Heapsort

Merge Sort v.s. Insertion Sort

- The number of comparisons in merge sort
  - $\Theta(nlgn)$
- The number of comparisons in insertion sort
  - $\Theta(n^2)$
- Merge sort requires the allocation of new memory to complete the “Merge” procedure
- Insertion sort is in place
  - No need to request additional space

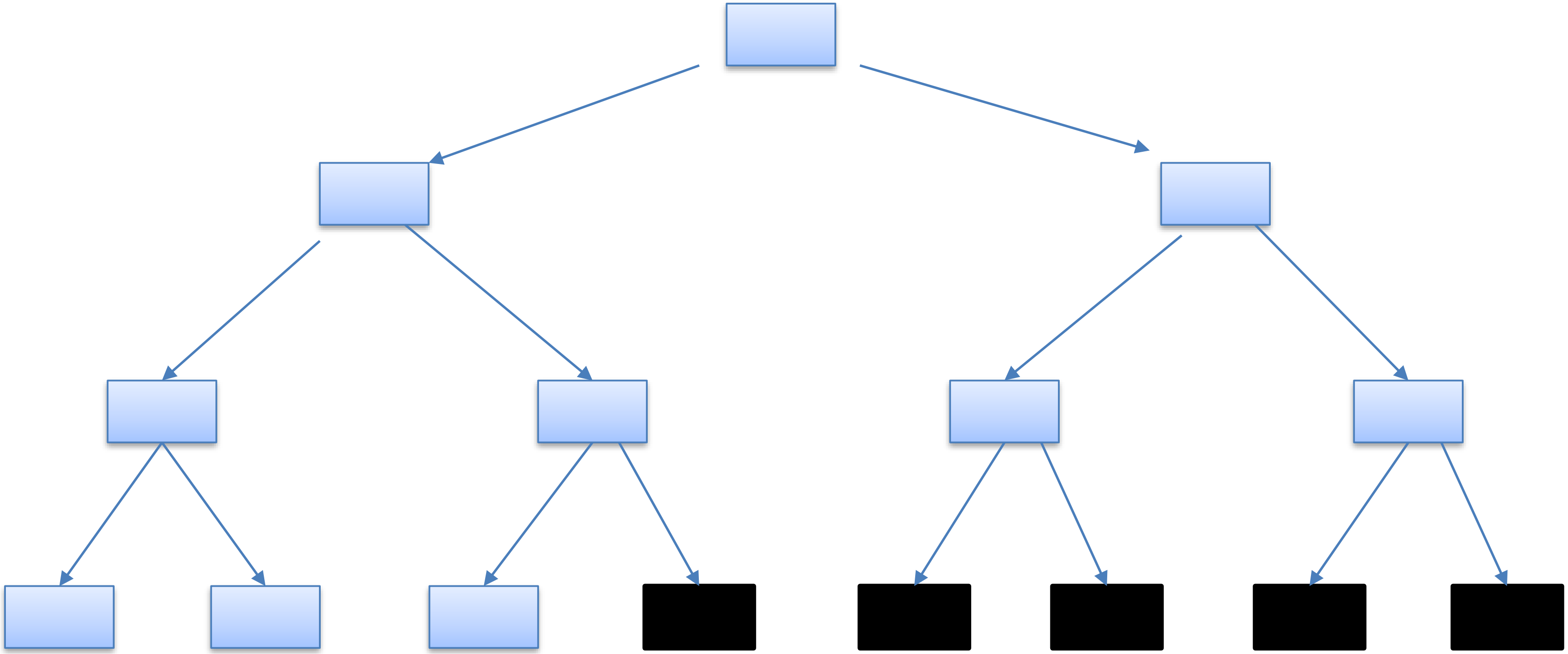
Heaps

- A heap is a complete binary tree



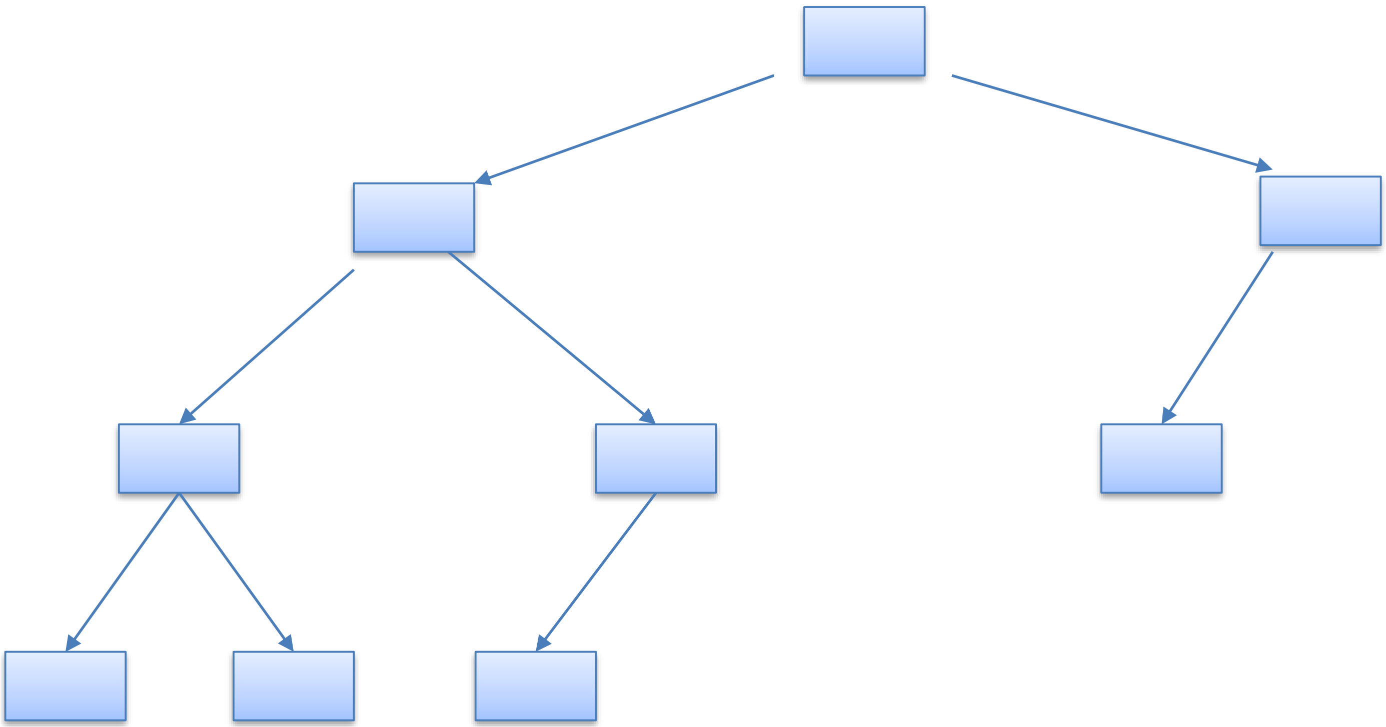
Heaps (cont'd)

- We can think of unfilled leaves as null pointers

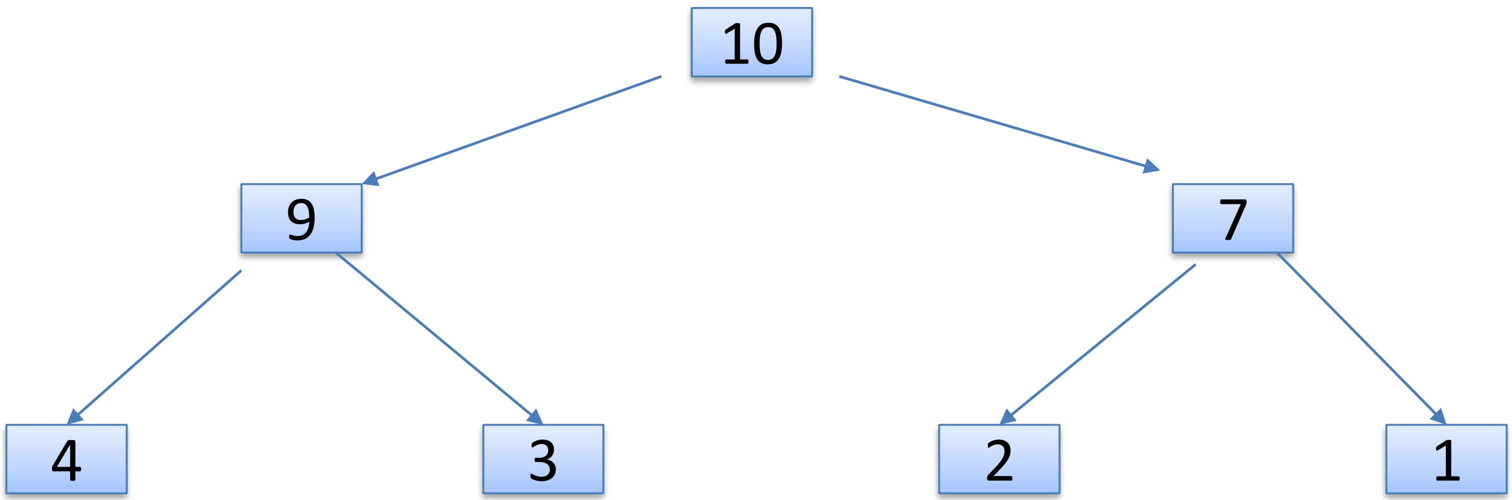


Heaps (cont'd)

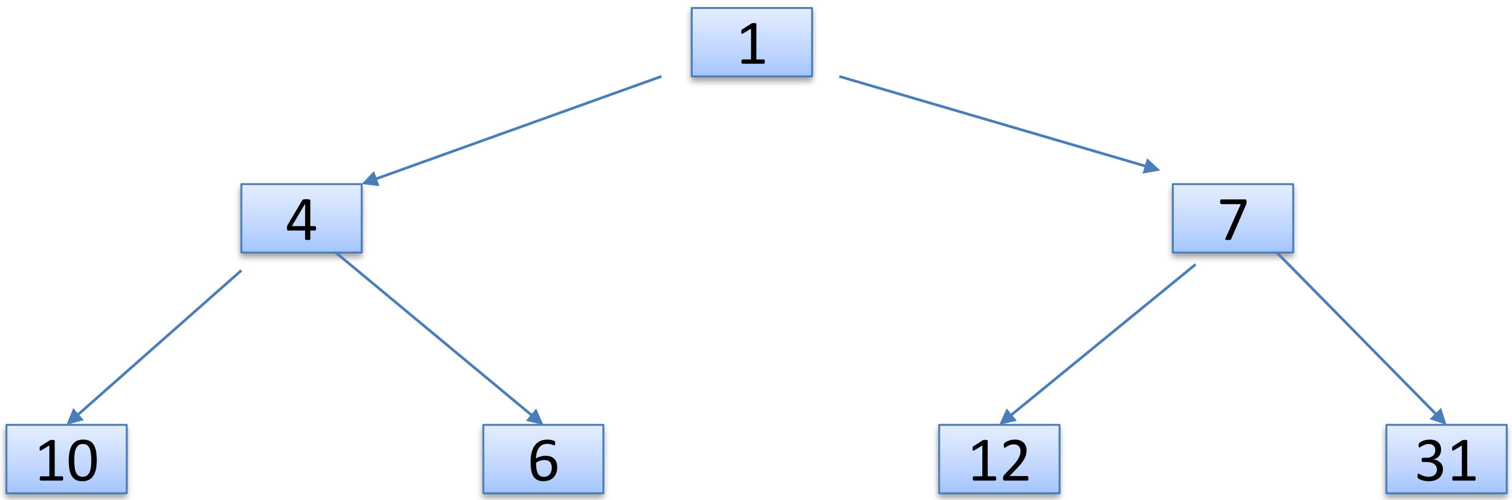
- Not a heap



Max-heap

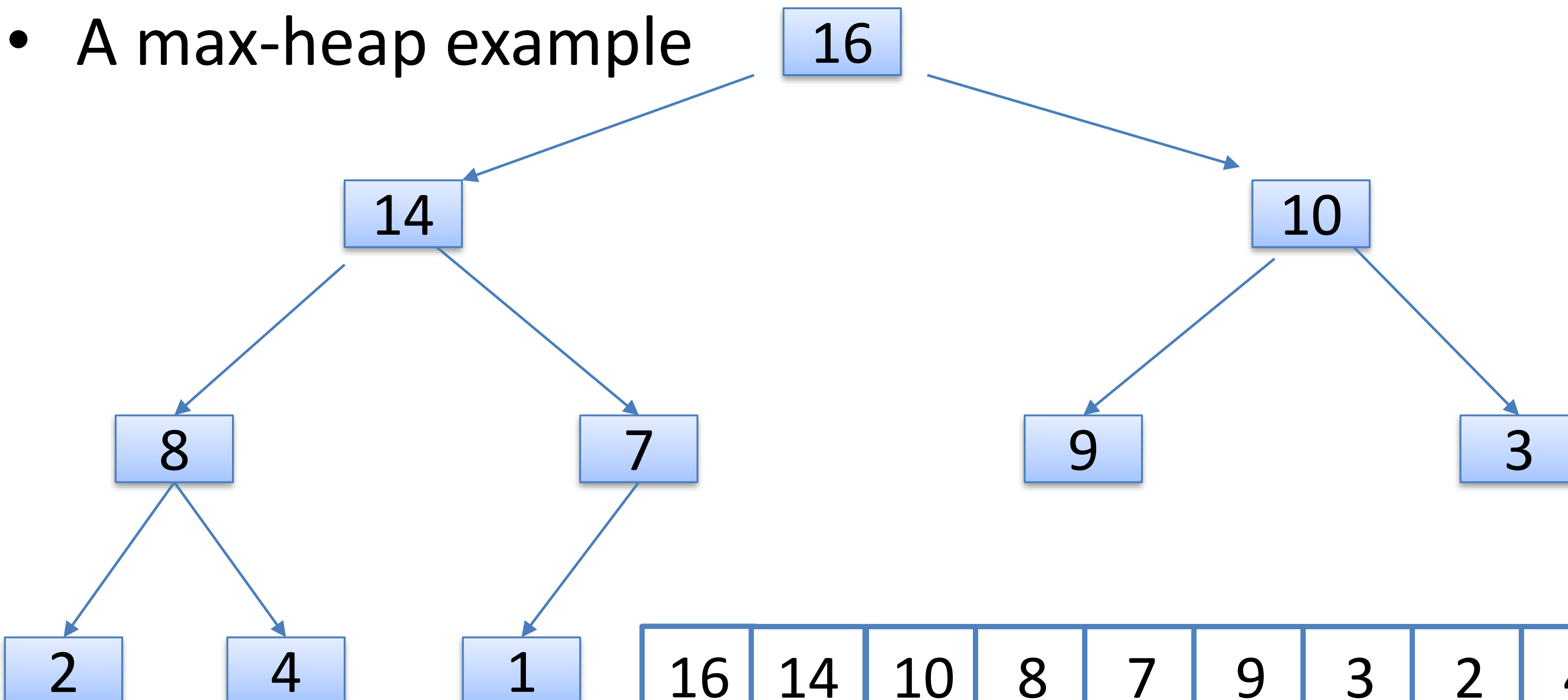


Min-heap



The implementation of heap

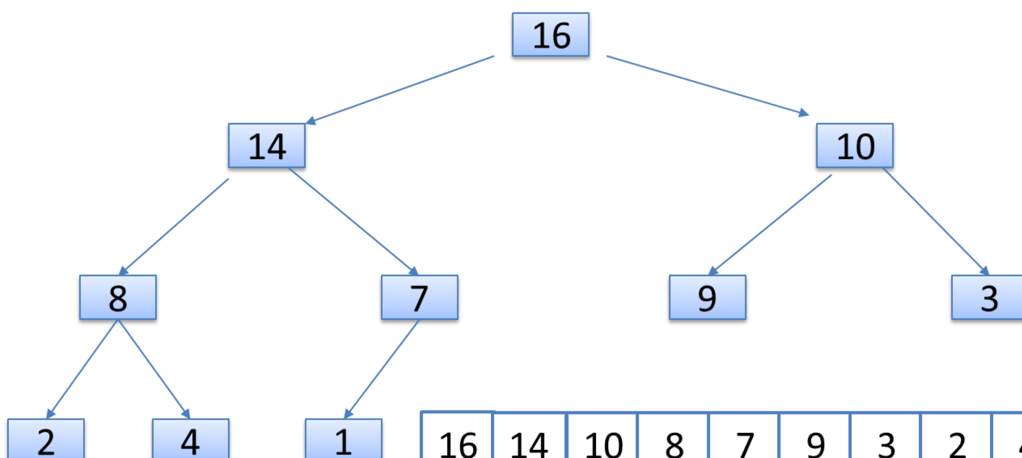
- Heaps are usually implemented as arrays (element index starts from 1)
- A max-heap example



16	14	10	8	7	9	3	2	4	1

Cont'd

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]
  - The parent of node *i* is A[$\lfloor i/2 \rfloor$]



16	14	10	8	7	9	3	2	4	1

Referencing heap elements

- So, we have

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Bit shift operations

- We can use bit shift operations to improve the efficiency
- **`2*`*`i`*** - > left shift *i* by 1 bit
  - E.g., (2*11 = 22) 00001011 << 1 = 00010110
- $\lfloor i/2 \rfloor$ - > right shift *i* by 1 bit
  - E.g., ($\lfloor 3 / 2 \rfloor = 1$) 00000011 >> 1= 000000001

Summary of heaps

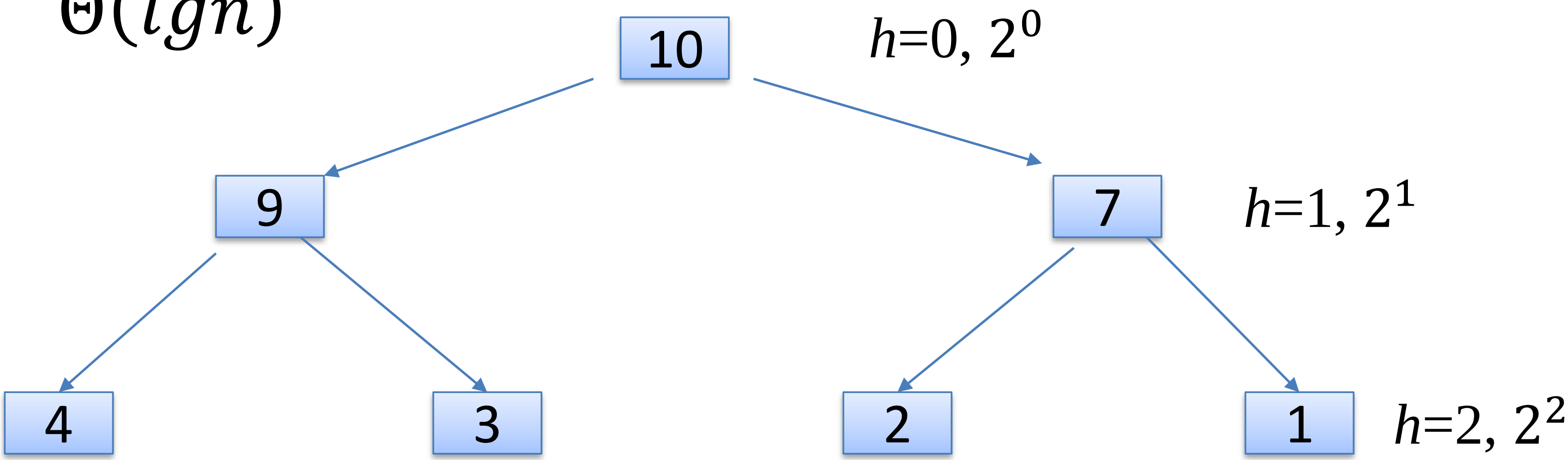
- A heap is a complete binary tree
- A heap can be represented as an array A
  - Root is A[1]
  - Parent of A[i] is A[$\lfloor i/2 \rfloor$]
  - Left child of A[i] is A[2*i]
  - Right child of A[i] is A[2*i+1]
- Bit manipulations can be used to improve the efficiency

Heap height

- Height of a node
  - Number of edges on a longest simple path from the node down to a leaf.
- Height of a tree = height of the root
- Height of a heap
  - Height of the root = lg n
- why?

Heap height (cont'd)

- Show a heap with *n* nodes has a height of $\Theta(lgn)$



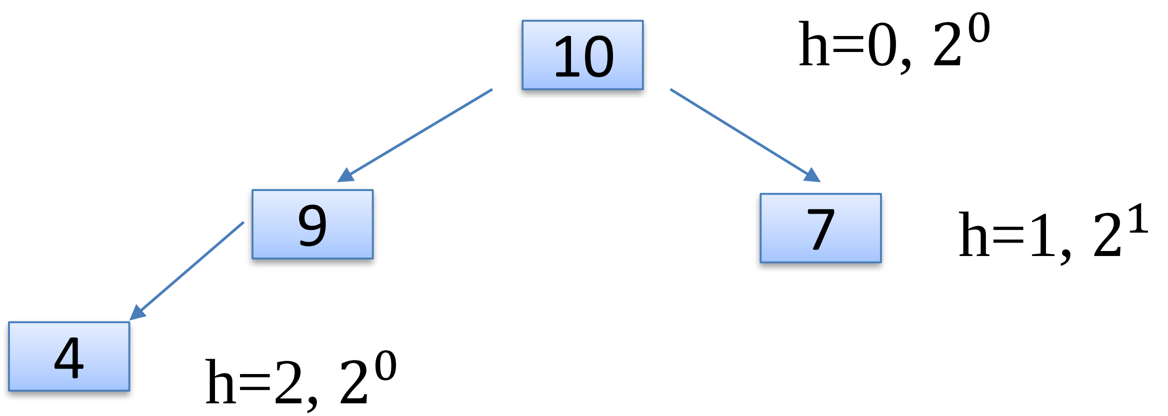
$$n = 2^0 + 2^1 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\Leftrightarrow\ h = \lg(n+1) - 1 = \Theta\,(\lg n)$$

Assume all leaves are filled

Heap height (cont'd)

- What if not all leaves are filled?



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

$$\Leftrightarrow \; h \geq \lg(n+1) - 1 \qquad h \in \Omega(\lg n)$$

$$n \geq \; 2^0 + 2^1 + \cdots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

$$\Leftrightarrow \; h \leq \lg(n+1) \qquad h \in O(\lg n)$$

Exercise

- Suppose you are given the following data structure to represent a binary Tree

```
Struct BinaryTree{
    int data;
    *BinaryTree left;
    *BinaryTree right;
}
```

- Write a function to return the height of a binary tree. You may declare your function like this
  - int maxHeight(BinaryTree *p)

Exercise (cont'd)

- Write a function in C to compute the height of a binary tree

$$h(root) = 1 + \max(h(left), h(right))$$

```
int maxHeight(BinaryTree *p) {
  if (!p) return -1;
  int left_height = maxHeight(p->left);
  int right_height = maxHeight(p->right);
  return (left_height > right_height) ? left_height + 1 : right_height + 1;
}
```

The property of a heap

- Heaps must satisfy the heap property
- Max-heap:
  - A[parent(i)] $\geq$ A[i] for all nodes $i > 1$
  - In other words, the value of a node is at most the value of its parent
  - Where is the largest element in a max-heap stored?

The property of a heap (cont'd)

- Min-heap:
  - $A[\text{parent}(i)] \leq A[i]$ for all nodes $i > 1$
  - In other words, the value of a node is at least the value of its parent
  - Where is the smallest element in a min-heap stored?
- In this course, we focus our discussions on max-heap

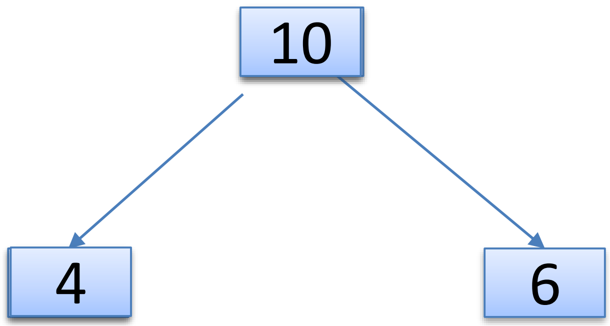
Maintaining the heap property

- How?
- We use HEAPIFY to maintain the property
- Before HEAPIFY, $A[i\ ]$ may violate the property
  - Two subtrees rooted at the left and right children of i, assumed to be heaps
- After HEAPIFY, subtree rooted at *i* is a max-heap.

Heap Operations: MAX-Heapify()

- Given a node *i* in the heap
  - with children *l* and *r.*
  - two subtrees rooted at l and r, assumed to be heaps
- Problem: The subtree rooted at *i* may violate the heap property
- Action: let the value of the parent node "float down"

MAX-Heapify () (cont'd)



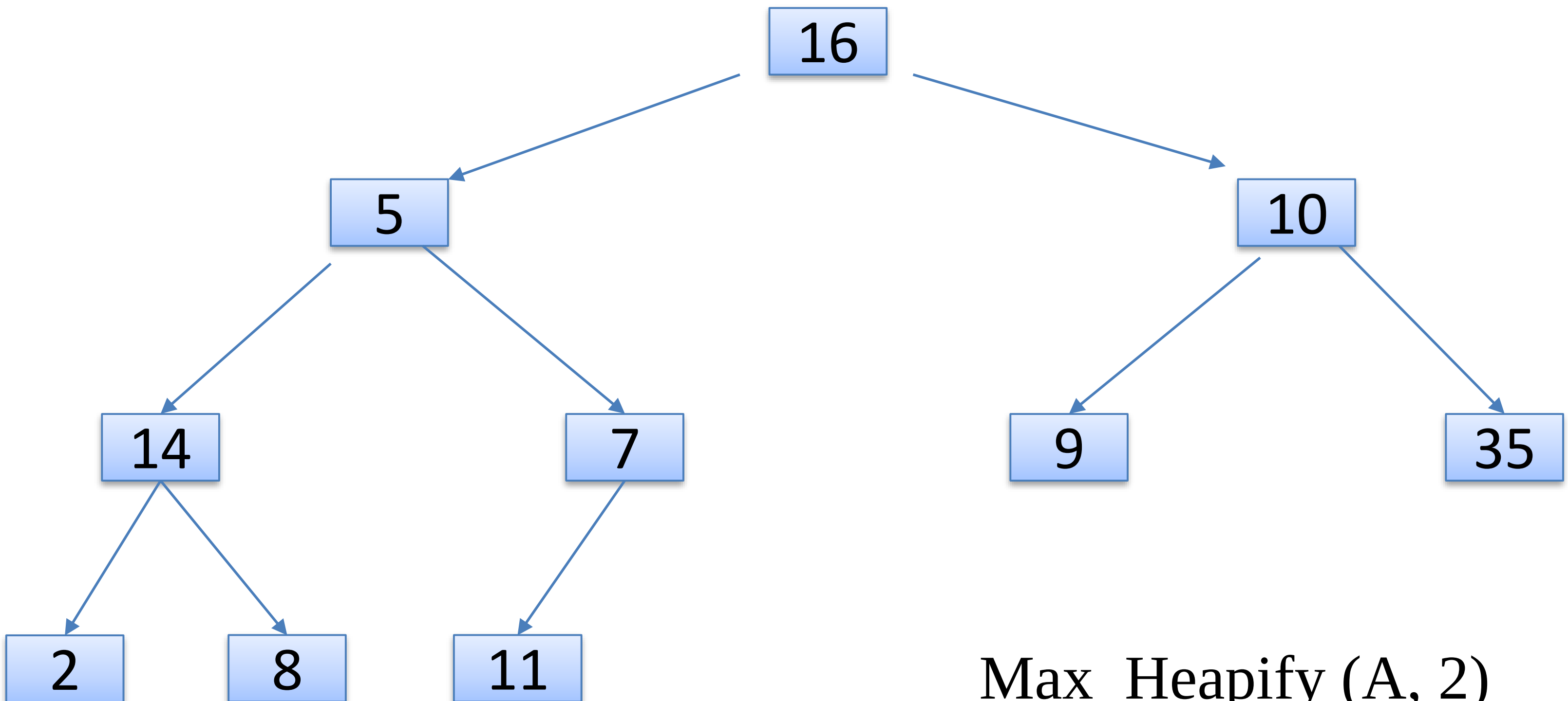
MAX-Heapify () (cont’d)

```
Max_Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= A.heap_size && A[l] > A[i])
        largest = l;
   else
        largest = i;
   if (r <= A.heap_size && A[r] > A[largest])
        largest = r;
   if (largest != i)
        Swap(A, i, largest);
        Max_Heapify(A, largest);//why this works?
}
```

How MAX-HEAPIFY works

- *heap-size is the* *<u>current heap size</u>*
- 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.
  - If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

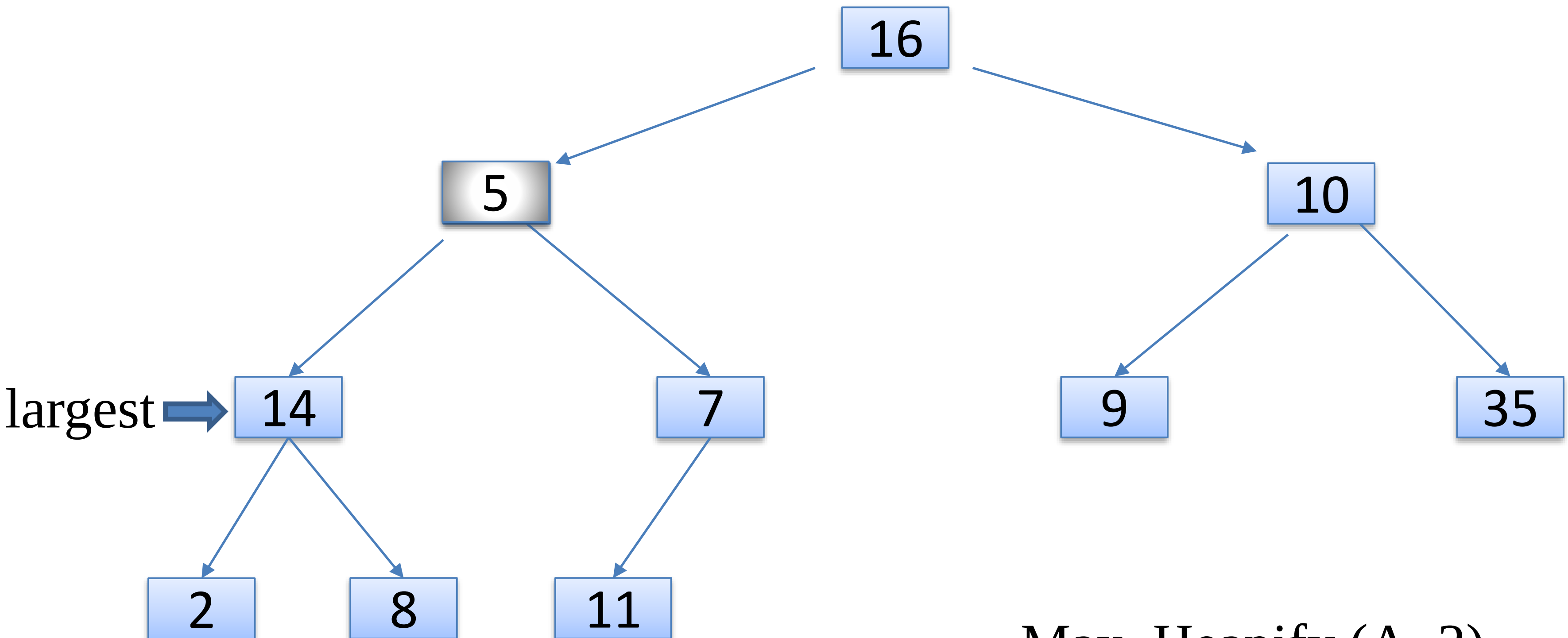
MAX-HEAPIFY example



Max_Heapify (A, 2)

16	5	10	14	7	9	35	2	8	11

MAX-HEAPIFY example

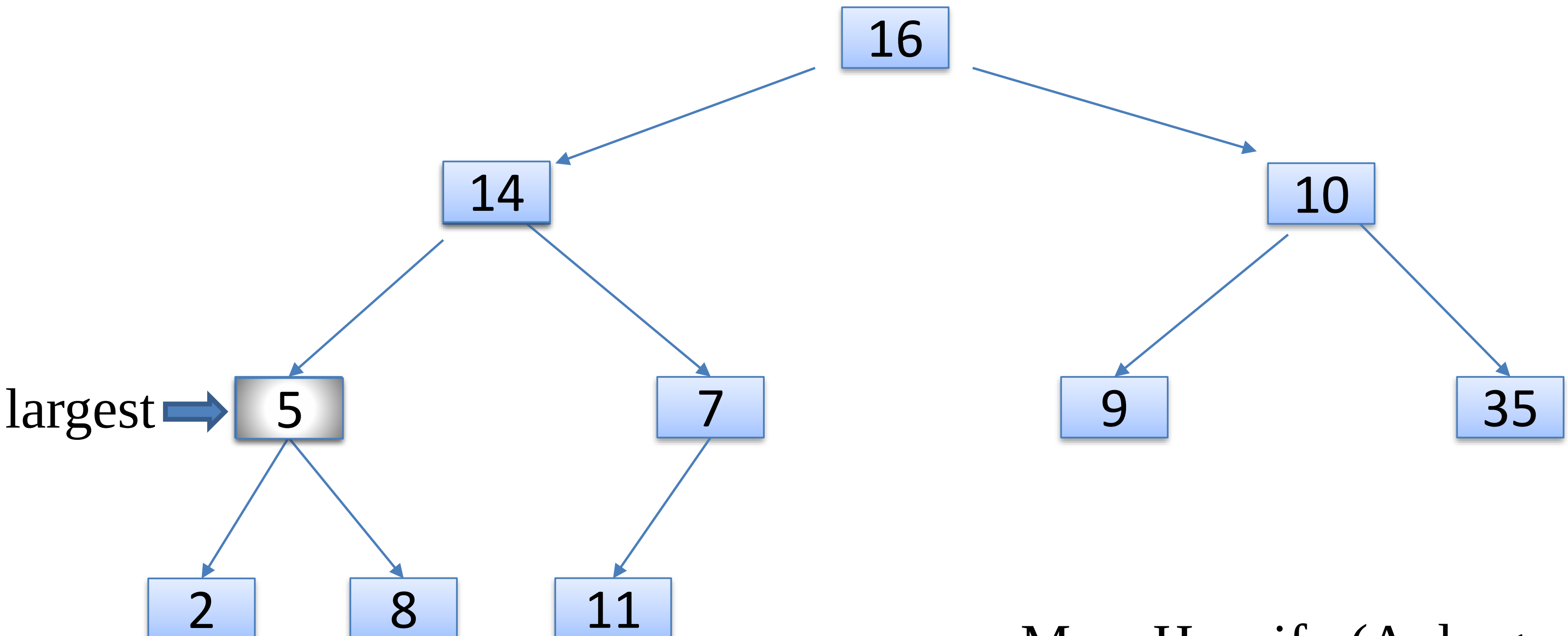


Max_Heapify (A, 2)

16	5	10	14	7	9	35	2	8	11

largest

MAX-HEAPIFY example

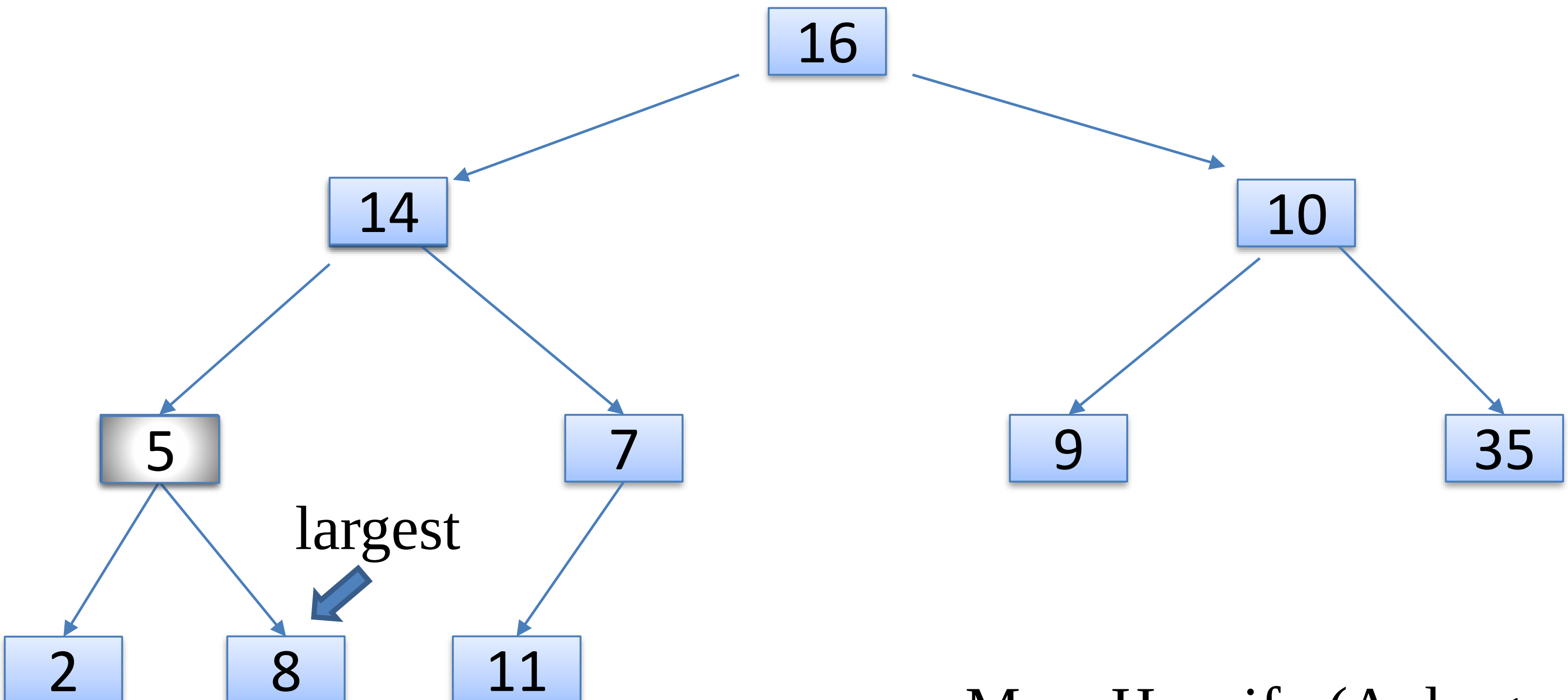


Max_Heapify (A, largest)

16	14	10	5	7	9	35	2	8	11

largest

MAX-HEAPIFY example

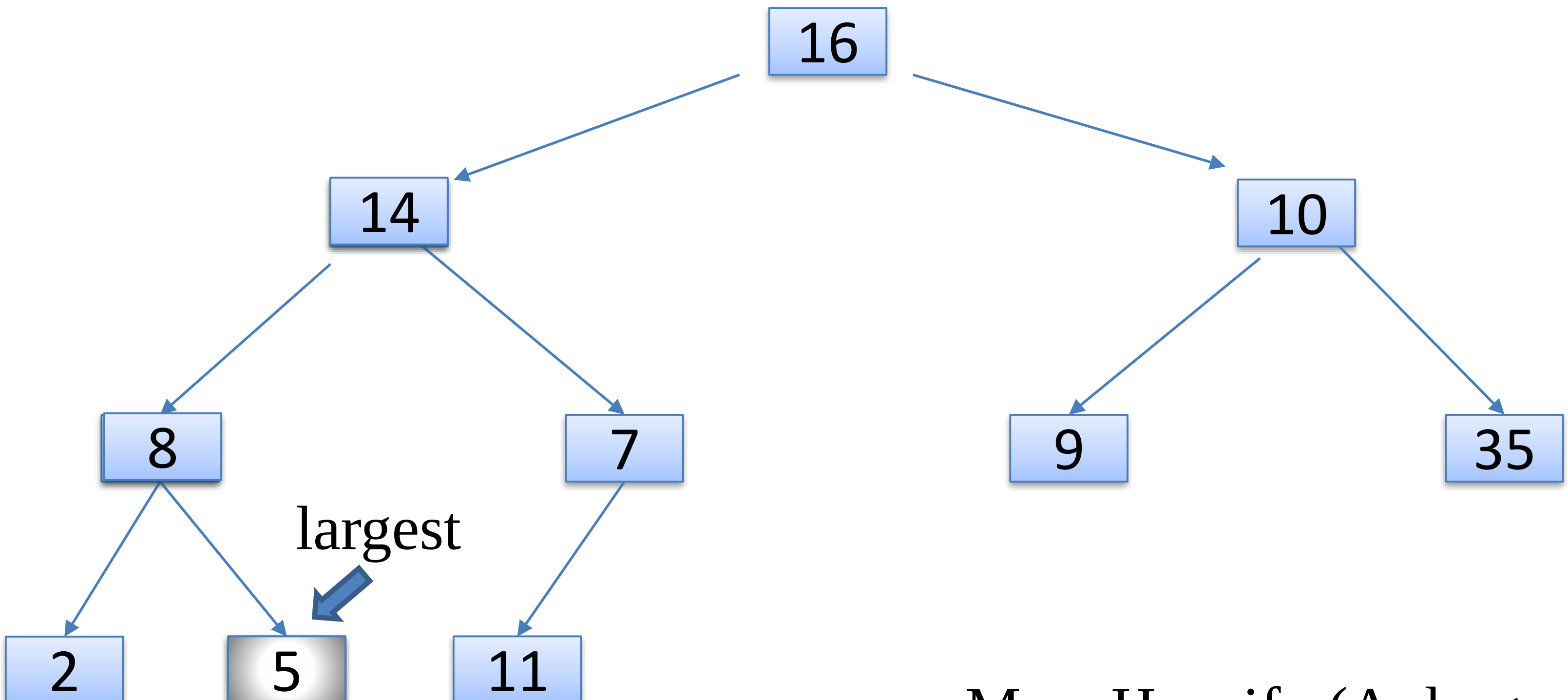


Max_Heapify (A, largest)

16	14	10	5	7	9	35	2	8	11

largest

MAX-HEAPIFY example

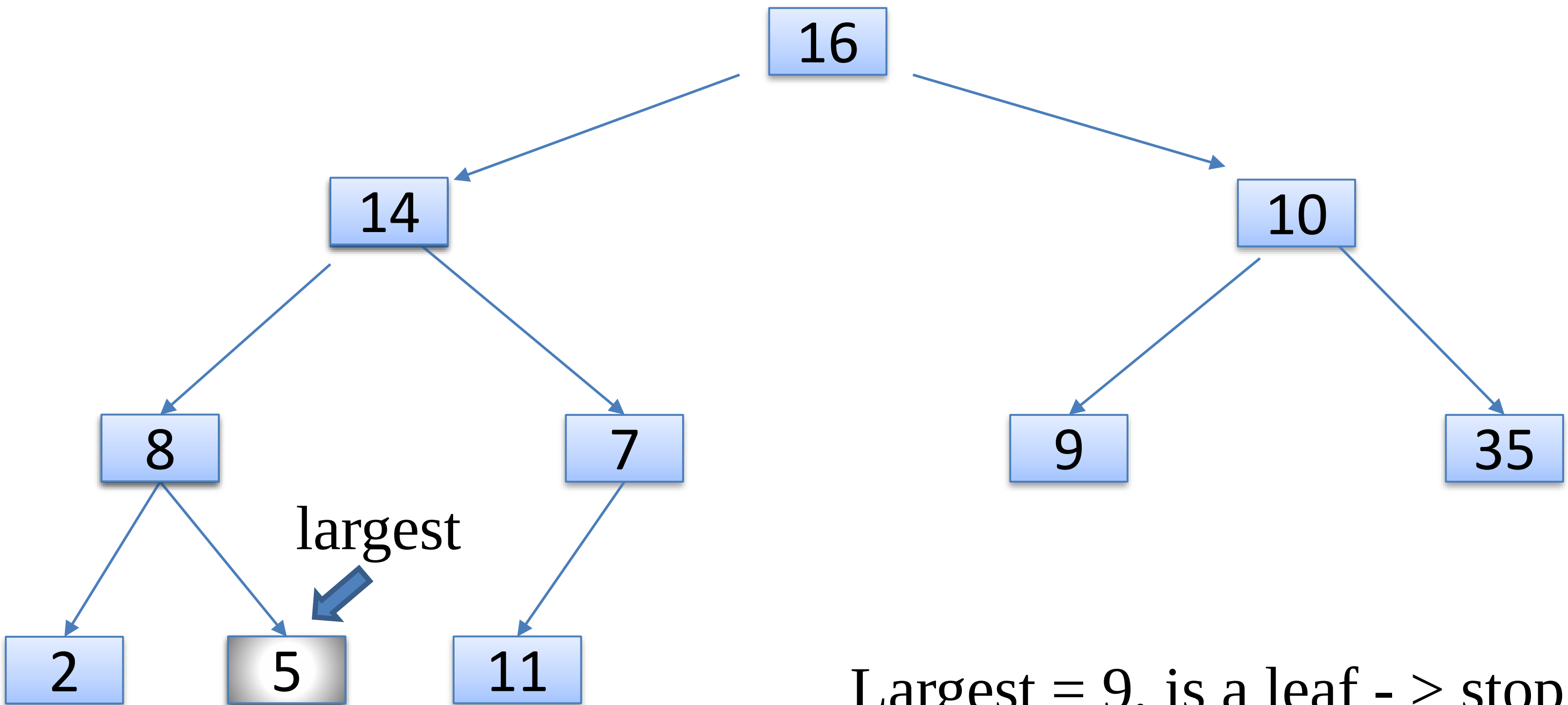


Max_Heapify (A, largest)

16	14	10	8	7	9	35	2	5	11

largest

MAX-HEAPIFY example



Largest = 9, is a leaf - > stop

16	14	10	8	7	9	35	2	5	11

largest

Swap function

```
void Swap (A, i, j)
 {
            int t = 0;
            t = A[i];
            A[i] = A[j];
            A[j] = t;
  }
```

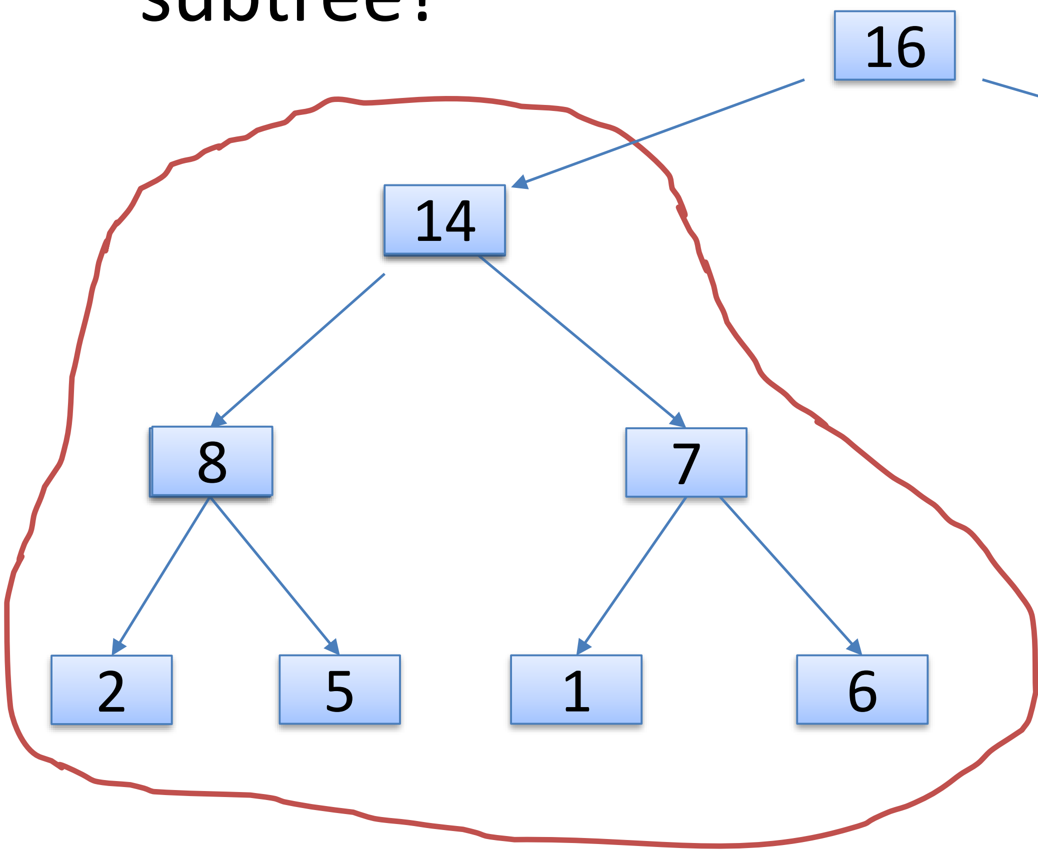
Swap function (cont'd)

- Swapping without using extra variable
- Bit operation: exclusive or
- void Swap (A, i, j)

```
{
        A[i] = A[i]^A[j];
        A[j] = A[j]^A[i];
(% Substituting A[i] in step 1 into A[j] -> A[j] = A[j]^A[i]^A[j] = A[i] )
        A[i] = A[i]^A[j];
(% Substituting A[i] in step 1 and A[j] in step 2 into A[i] -> A[i] = A[i]^A[j]^A[i] = A[j]
}
```

Analyzing MAX-HEAPIFY

- What is the maximum possible size of a subtree?



Analyzing MAX-HEAPIFY (cont'd)

- For a heap with *n* nodes, a subtree has the maximum size when
  - Its root is the left child of the root of the heap
  - **and** It is a complete binary tree with all leaves filled
  - **and** the subtree rooted at the right child lacks the bottom level
  - **and** the bottom level of the entire tree is exactly half full

Analyzing MAX-HEAPIFY (cont'd)

- For a heap of **n** nodes and height **x**, suppose the left tree has the maximum size
- The size of the left tree is

$$2^0 + 2^1 + \cdots + 2^{x-1} = \sum_{i=0}^{x-1} 2^i = 2^x - 1$$

- The size of the right tree is

$$2^0 + 2^1 + \cdots + 2^{x-2} = \sum_{i=0}^{x-2} 2^i = 2^{x-1} - 1$$

- The size of the entire tree is (size of the left tree) + (size of the right tree) + 1

$$(2^x - 1) + (2^{x-1} - 1) + 1 = n$$

Analyzing MAX-HEAPIFY (cont'd)

- Size of the entire tree

  $(2^{x}-1) + (2^{x-1} - 1) + 1 = n \Rightarrow 2^{x} = \frac{2}{3}(n+1)$

- The size of the left tree is

  $2^{x} - 1 = \frac{2(n+1)}{3} - 1 = \frac{2n}{3} - \frac{1}{3} \approx \frac{2n}{3}$

MAX-Heapify () (cont’d)

```
Max_Heapify(A, i)
{
  l = Left(i); r = Right(i);
  if (l <= A.heap_size && A[l] > A[i])
      largest = l;
  else
      largest = i;
  if (r <= A.heap_size && A[r] > A[largest])
      largest = r;
  if (largest != i)
      Swap(A, i, largest);
      Max_Heapify(A, largest);//why this works?
}
```

Analyzing MAX-HEAPIFY (cont'd)

- Fixing up relationships between i, *l*, and *r* takes $\Theta(1)$ time
- The subtree at *l* has at most 2*n*/3 nodes (worst case: bottom row 1/2 full)
- So time taken by MAX-Heapify() is given by
- $T(n) \leq T(2n/3) + \Theta(1)$
- By using master theorem (case 2), we have
- $T(n) = O(\lg n)$