COT 6405 Introduction to Theory of Algorithms

Topic 6. Heapsort (cont'd)

Heap operations: BuildHeap

- We can build a max-heap in a bottom-up manner by running **`MAX-Heapify(x)`** as *x* runs through all nodes
  - **for** $i \leftarrow n$ **downto** 1 **do** MAX-Heapify(i)
- Order of processing guarantees that the children of node i are heaps when i is processed
- A better upper bound?

BuildHeap

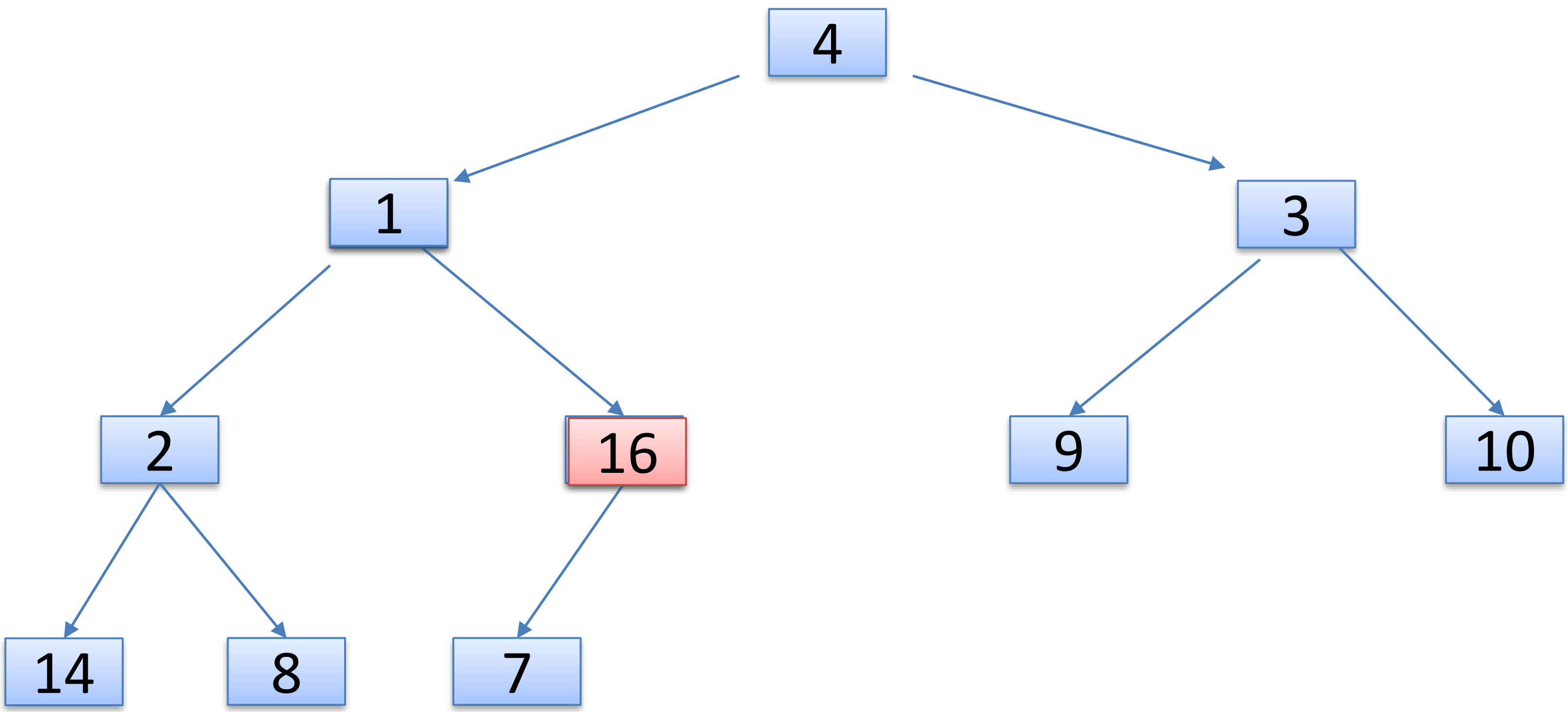
- For an array of length n, all elements in range A[$\lfloor n/2 \rfloor + 1...$ n] are heaps (Why?)
- Walk backwards through the array from $\lfloor n/2 \rfloor$ to 1, calling MAX-Heapify() on each node.

Build-MAX-Heap()

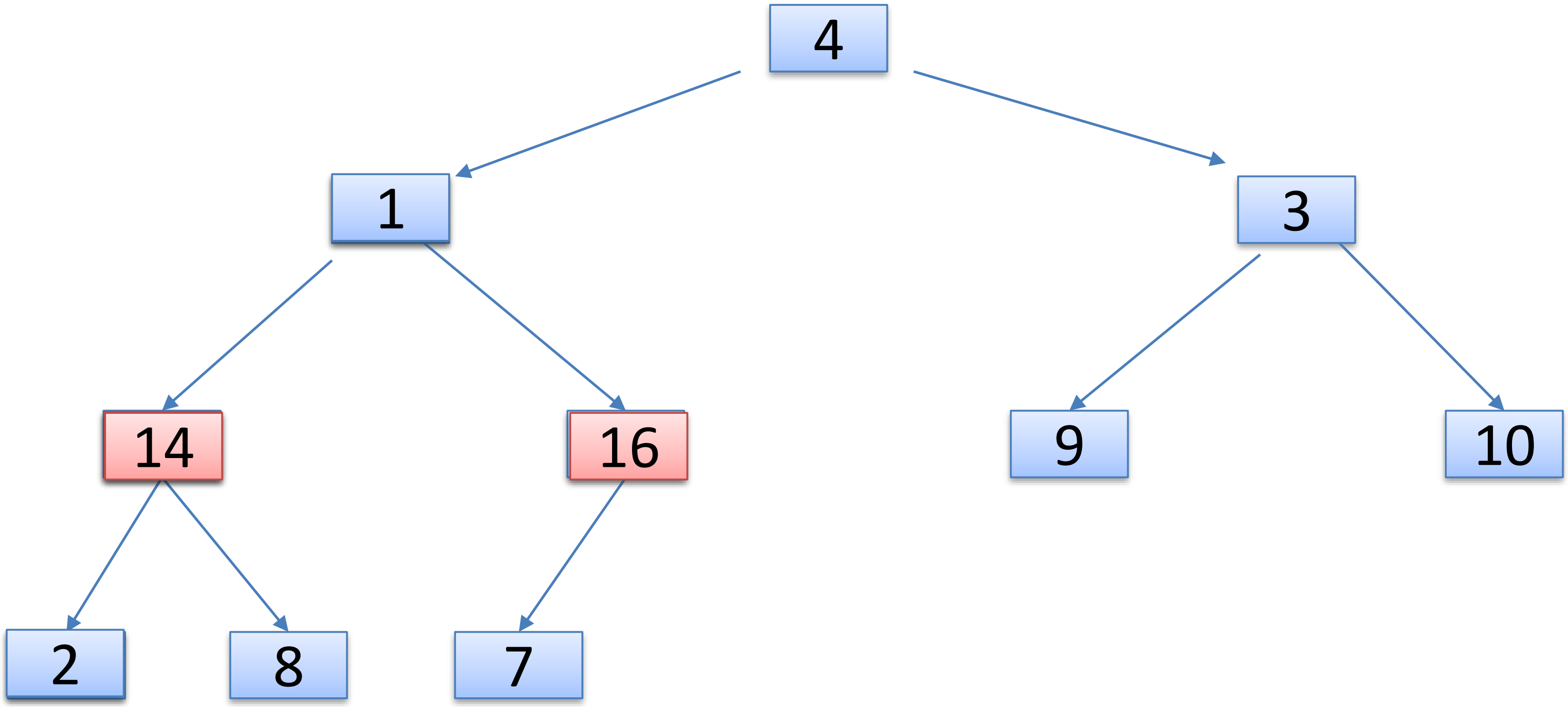
```
// given an unsorted array A, make A a heap
Build-MAX-Heap(A)
{
  A.heap_size = A.length;
  for (i = ⌊A.length/2⌋ downto 1)
    MAX-Heapify(A, i);
}
```

Build-MAX-Heap() Example

- A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7} (10 elements)
- We started with i = A.length/2 = 5

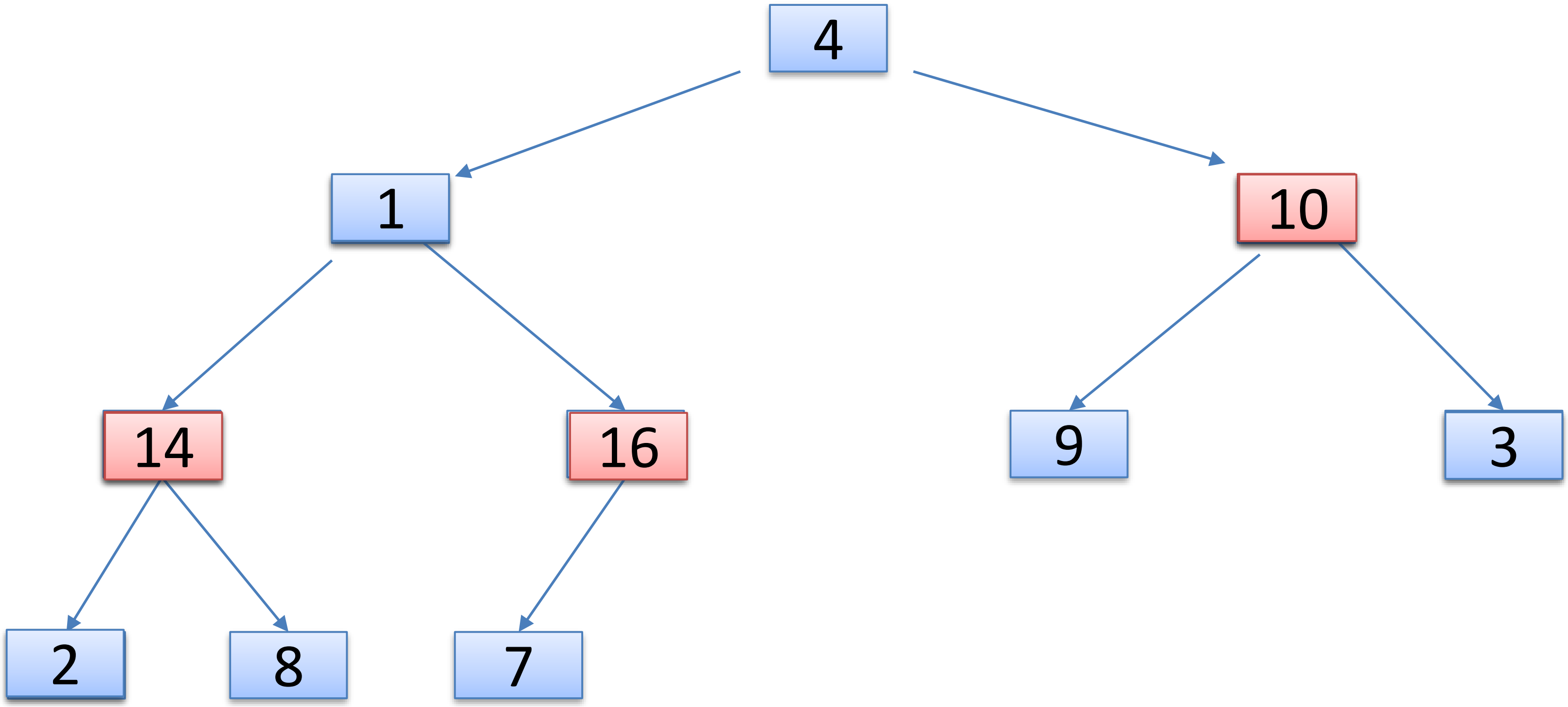


Example (cont'd)



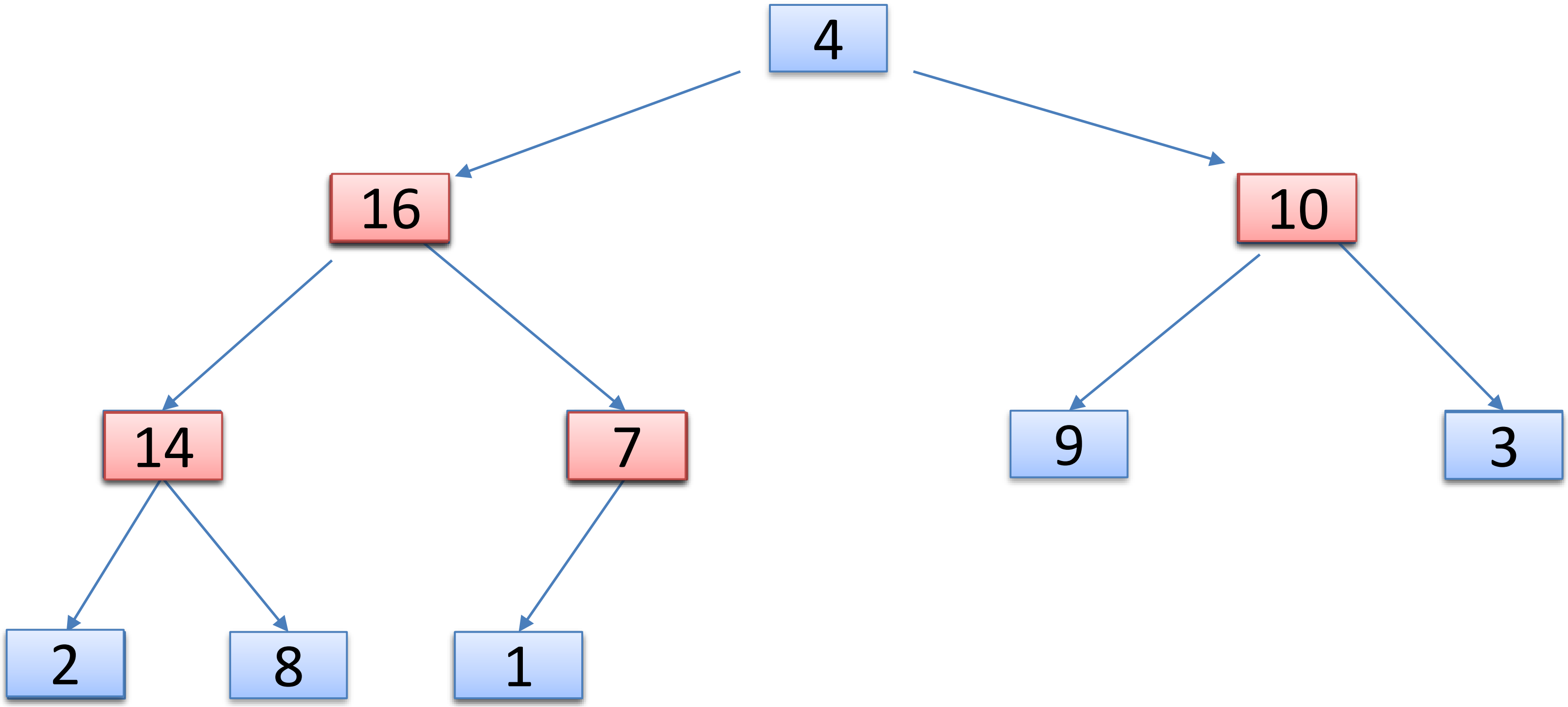
i = 4, A = {4, 1, 3, 14, 16, 9, 10, 2, 8, 7}

Example (cont'd)



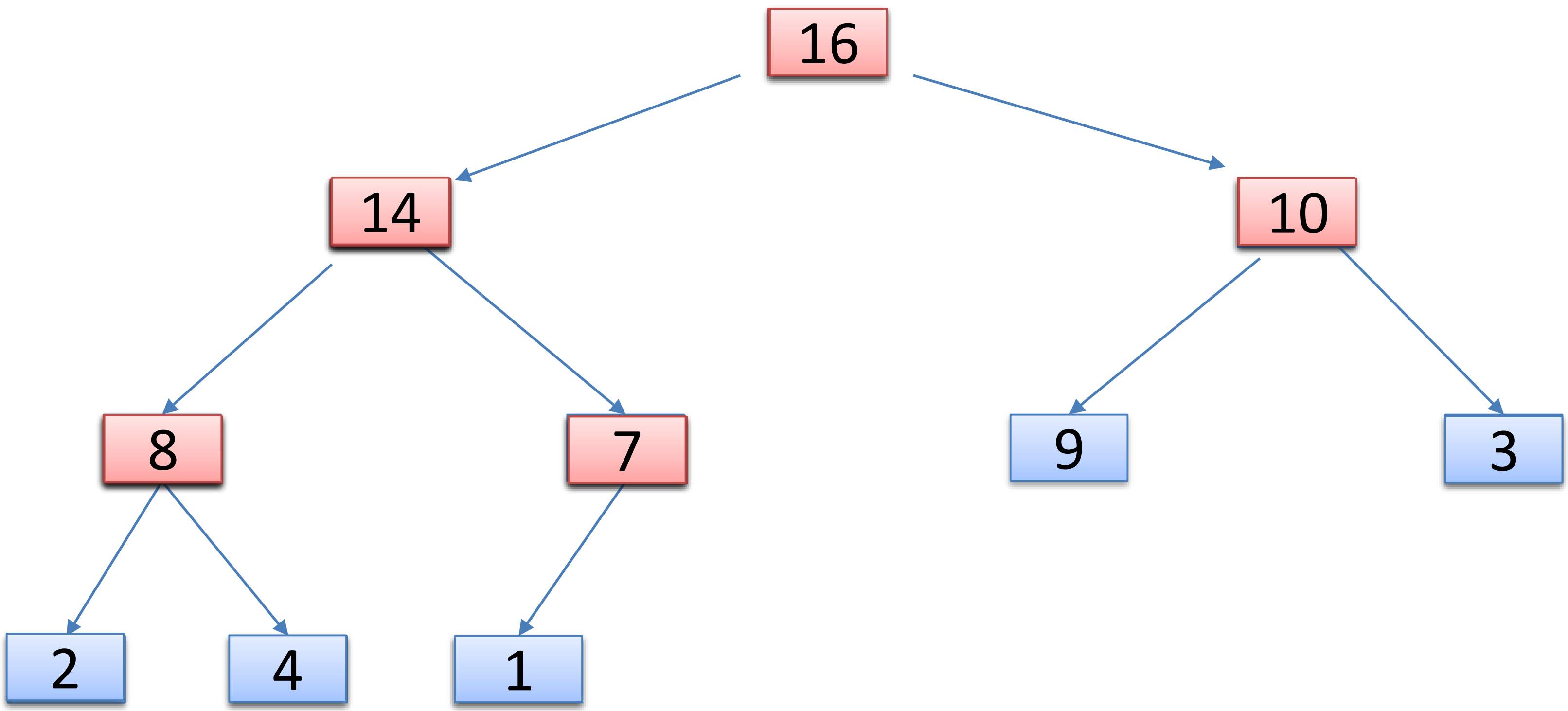
i = 3, A = {4, 1, 10, 14, 16, 9, 3, 2, 8, 7}

Example (cont'd)



i = 2, A = {4, 16, 10, 14, 7, 9, 3, 2, 8, 1}

Example (cont'd)



i = 1, A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}

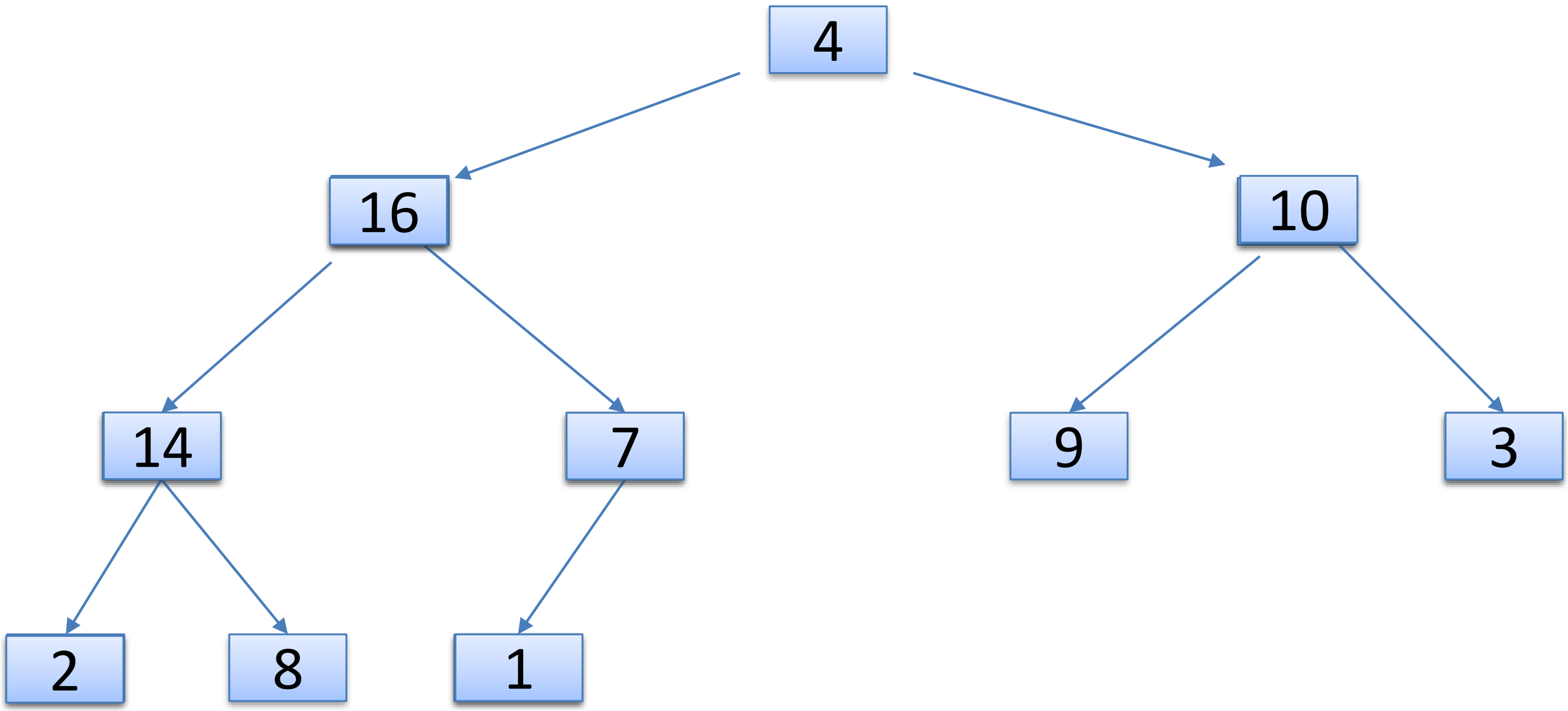
Example

- What happens if we walk forwards through the array?

```
Build-MAX-Heap(A)
{
  A.heap_size = A.length;
  for (i = 1 to ⌊A.length/2⌋)
    MAX-Heapify(A, i);
}
```

Then, we will never build a heap !!

Example (cont'd)



BUILD_MAX_HEAP correctness

Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i+1$, $i+2, \ldots, n$ is root of a max-heap.

Initialization: we know that each node $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$, $\ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

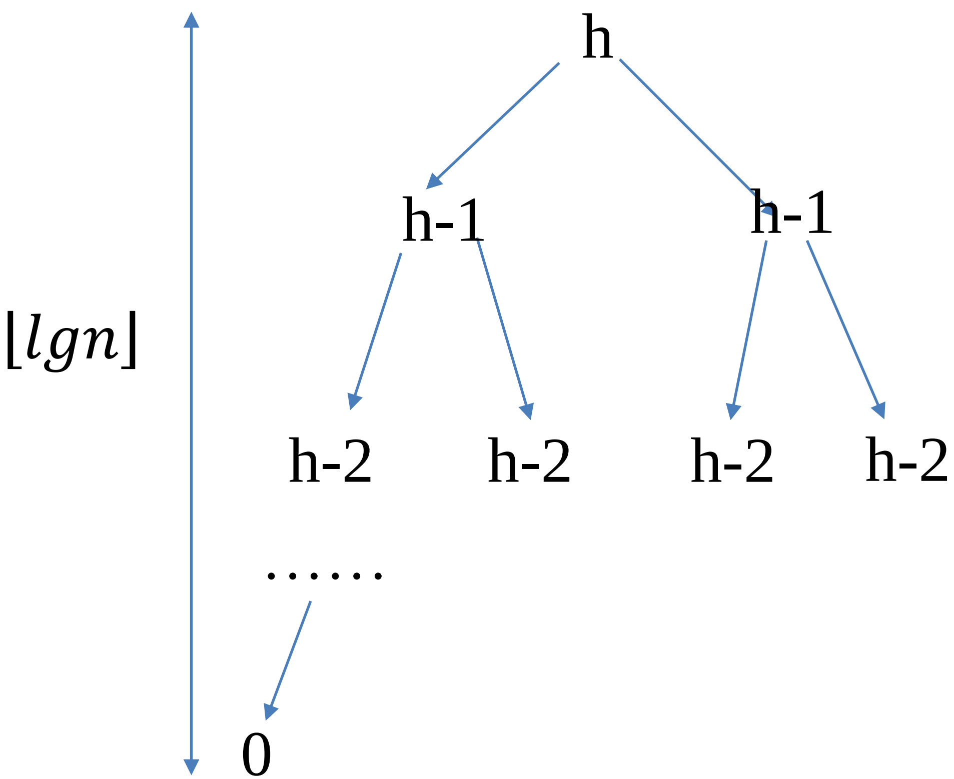
Maintenance: Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analyzing Build-MAX-Heap

- Each call to **`MAX-Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
- A tighter bound of Build-MAX-Heap is O(*n*)
  - How could this be possible?

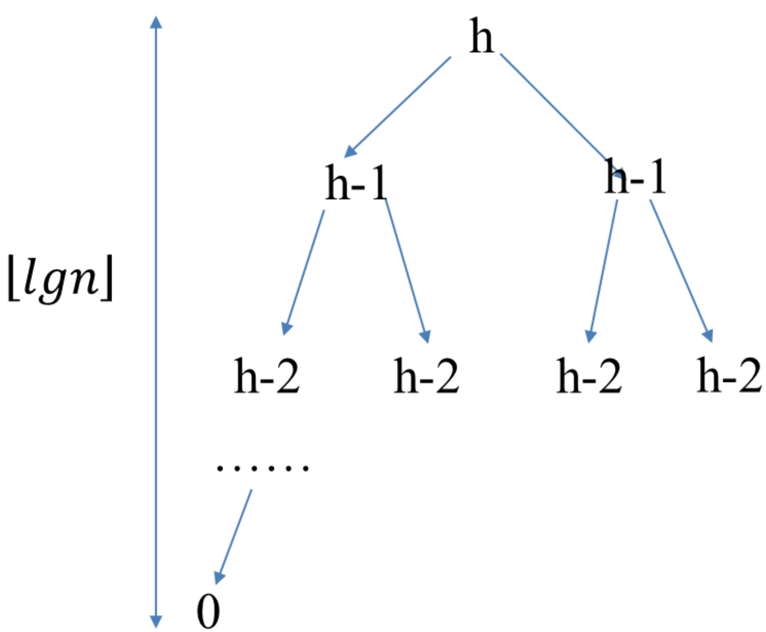
Analyzing Build-MAX-Heap (cont'd)



height	# nodes	cost
h	2^0	$h * 2^0$
h-1	2^1	$(h-1) * 2^1$
h-2	2^2	$(h-2) * 2^2$
0	2^h	$0 * 2^h$

Analyzing Build-MAX-Heap (cont'd)

- Adding up the costs of each level together
- T(n) = $\sum_{x=0}^{h} x\, 2^{h-x} = \sum_{x=0}^{\lfloor lgn \rfloor} x\, 2^{\lfloor lgn \rfloor - x}$



height	# nodes	cost
h	2^0	$h * 2^0$
h-1	2^1	$(h-1) * 2^1$
h-2	2^2	$(h-2) * 2^2$
0	2^h	$0 * 2^h$

Analyzing Build-MAX-Heap (cont'd)

- T(n) = $\sum_{x=0}^{\lfloor lgn \rfloor} x\, 2^{\lfloor lgn \rfloor - x}$ = $\sum_{x=0}^{\lfloor lgn \rfloor} x\, \frac{2^{\lfloor lgn \rfloor}}{2^x}$

  =$\sum_{x=0}^{\lfloor lgn \rfloor} x\, \frac{n}{2^x}$ = n$\sum_{x=0}^{\lfloor lgn \rfloor} \frac{x}{2^x}$

  $\leq$ n$\sum_{x=0}^{\infty} \frac{x}{2^x}$ = 2n = O(n)

$$\boxed{\sum_{x=0}^{\infty} \frac{x}{2^x} = \sum_{x=0}^{\infty} x\left(\frac{1}{2}\right)^x = \frac{1/2}{(1-1/2)^2} = 2}$$

$$\sum_{k=0}^{\infty} k\, y^k = \frac{y}{(1-y)^2}$$

Heapsort

- Given **`Build-MAX-Heap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping it with element at A[n]
    - Decrement A.heap_size
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`MAX-Heapify()`**
  - Repeat, always swapping A[1] for A[A.heap_size]

Heapsort (cont'd)

```
Heapsort(A)
{
    Build-MAX-Heap(A);
    for (i = A.length downto 2)
    {
        Swap(A[1], A[i]);
        A.heap_size= A.heap_size - 1;
        MAX-Heapify(A, 1);
    }
}
```

Heapsort (cont'd)

- Can we call MAX-Heapify(A,1) instead of Build-MAX-Heap(A) before the loop?

```
Heapsort(A)
{
     Build-MAX-Heap(A);
     for (i = A.length downto 2)
     {
          Swap(A[1], A[i]);
          A.heap_size= A.heap_size - 1;
          MAX-Heapify(A, 1);
     }
}
```

Heapsort (cont'd)

- Can we call Build-MAX-Heap(A) instead of MAX-Heapify(A,1) inside of the loop?

```
Heapsort(A)
{
    Build-MAX-Heap(A);
    for (i = A.length downto 2)
    {
        Swap(A[1], A[i]);
        A.heap_size= A.heap_size - 1;
        MAX-Heapify(A, 1);
    }
}
```

Analyzing Heapsort

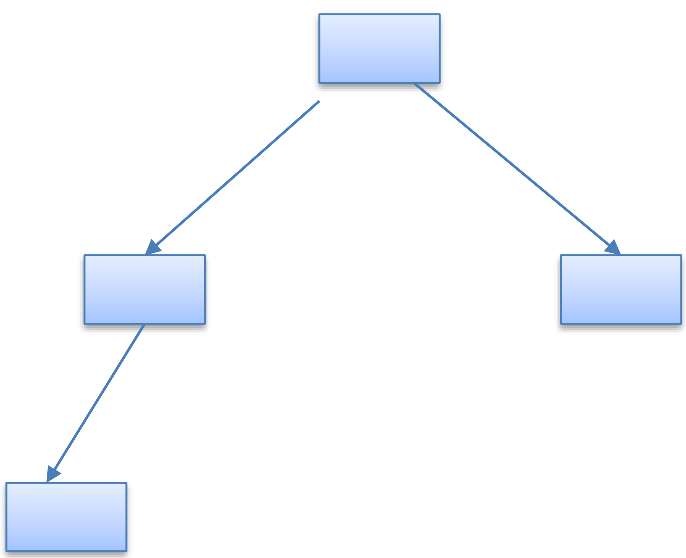
- The call to **`Build-MAX-Heap()`** takes O(*n*) time
- Each of the (*n* – 1) calls to **`MAX-Heapify()`** takes O(lg *n*) time
- Thus the total time taken by **`HeapSort()`** = O(*n*) + (*n* - 1) O(lg *n*)

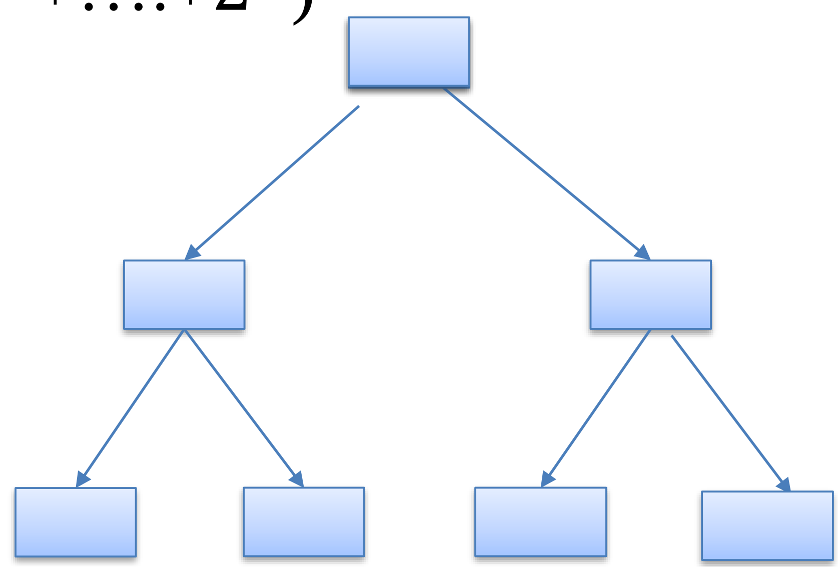
Exercise

- What are the minimum and maximum number of elements in a heap of height h?

Exercise (cont'd)

- A heap is a complete binary tree, so the minimum number of elements in a heap of height h is 2^h $(= 2^0+2^1+\ldots.+2^{h-1} + 1)$
- The maximum number of elements in a heap of height h is 2^{h+1}-1 $(= 2^0+2^1+\ldots.+2^h)$





COT 6405 Introduction to Theory of Algorithms

Topic 7. Priority queues

Priority Queues

- The heap data structure is incredibly useful for implementing (max-/min-) *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or key
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- How could we implement these operations using a heap?

Implementing Priority Queues

```
Heap-Maximum(A)
{
     return A[1];
}
```

Implementing Priority Queues

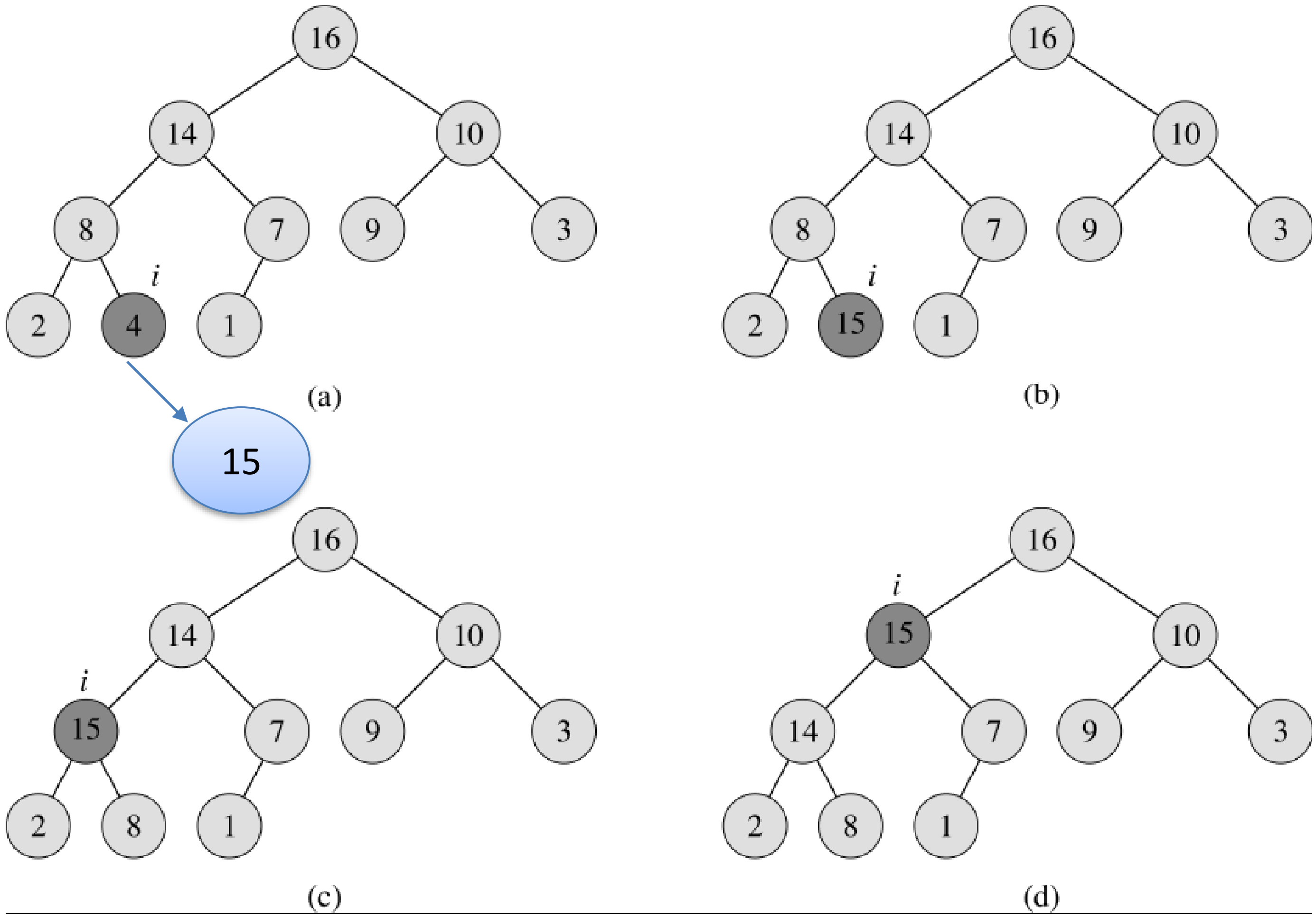
```
Heap-Extract-Max(A)
{
    if(A.heap_size < 1) { error; }
    max = A[1];
    A[1] = A[A.heap_size];
    A.heap_size = A.heap_size - 1;
    MAX-Heapify(A, 1);
    return max;
}
```

Implementing Priority Queues

```
Heap-INCREASE-KEY(A, i, key)

   if key < A[i] {error;}
   A[i]= key;
   while (i>1 and A[PARENT(i)]< A[i])
       exchange(A[i], A[PARENT(i)];
       i= PARENT(i);
} what's running time?
```

HEAP-INCREASE-KEY



Implementing Priority Queues

```
Max-Heap-Insert(A, key)
{
  A.heap_size = A.heap_size + 1;
  A[A.heap_size]= -∞;
  Heap-INCREASE-KEY(A,A.heap_size,key);
}
//what's running time?
```

Building a heap by insertions

- A heap could be built by successive insertions
- How about the cost (the number of swaps)?
- $\lfloor \lg 1 \rfloor + \lfloor \lg 2 \rfloor + \lfloor \lg 3 \rfloor + \cdots + \lfloor \lg n \rfloor$

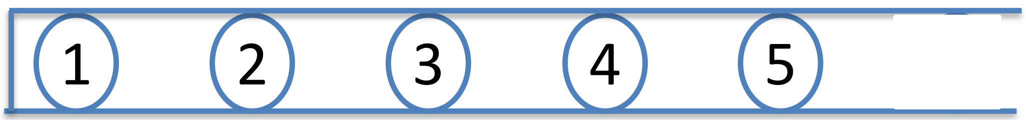
  $\leq \lg 1 + \lg 2 + \ldots + \lg n =$ lgn! = O(nlgn)
  - By Stirling's approximation
    - $\lg n! \approx nlgn - n + O(lgn)$ = O(nlgn)
- This is not the optimal way to construct a heap
- Build-MAX-Heap requires O(n) swaps

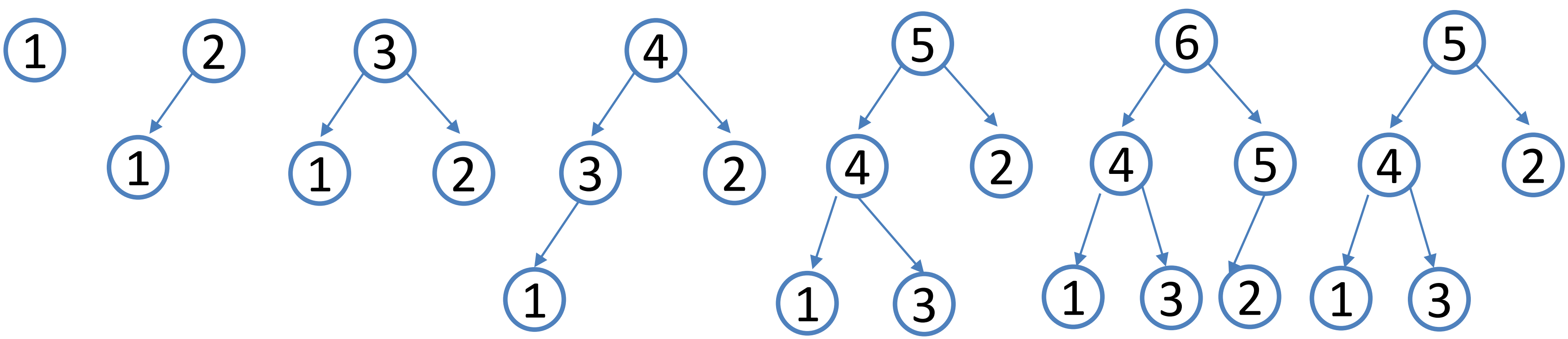
Common mistakes

- Not updating the heap when the key of a node changes.
- After extracting the maximum node, not building the heap again.

Exercise

- How to implement a stack by using a priority queue?





Stack.push(): Max-Heap-Insert()

Stack.pop:Heap-Extract-Max()

Exercise (cont’d)

```
class Stack
{
        private int c = 0;
        private PriorityQueue pq;
        public void Push(int x)
        {
            c++;
            pq.Insert(x, c); }
        public int Pop()
        {
            c--;
            return pq.Remove(); }
}
```

About midterm 1

- Midterm I will cover
  - From Intro lecture to Lecture 7 (inclusive)
  - Function growth rate analysis, divide and conquer, recurrence, recursion tree and the Master Theorem, heaps, basic heap operations, heapsort
  - 3:30pm to 4:45pm Sep 26th
  - Please be familiar with the basic concepts
  - Traditional Calculators are allowed (smart phones, smart tablets, and laptops are not allowed)