Data Structures

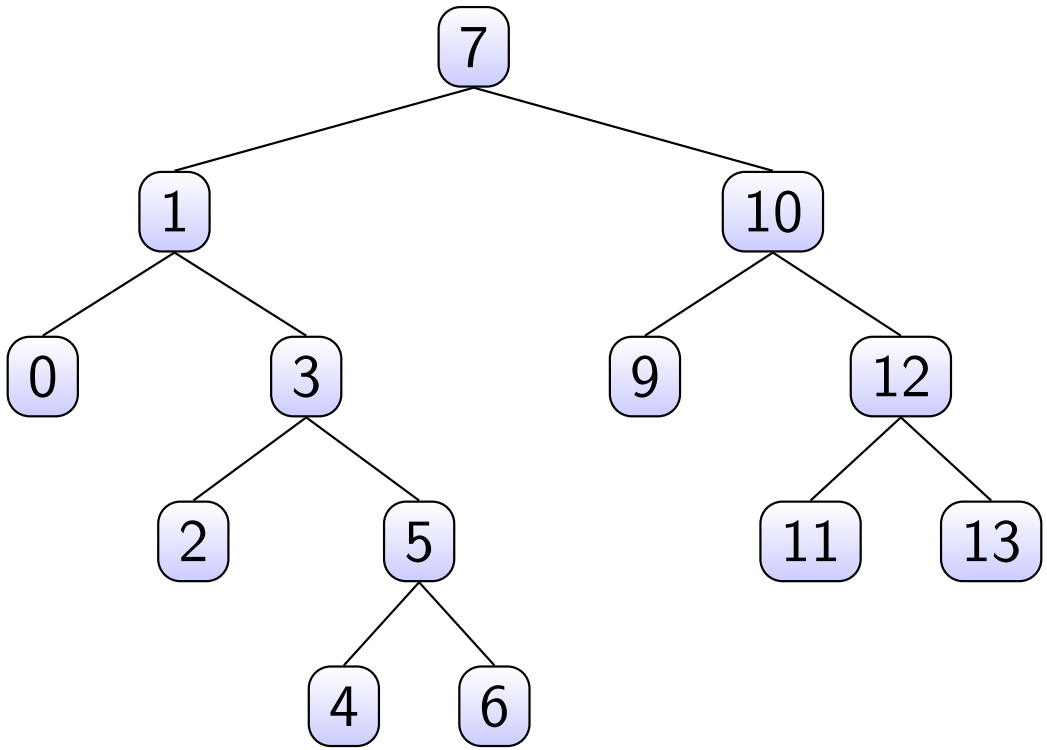
Heaps and Priority Queues

CS284

Heaps

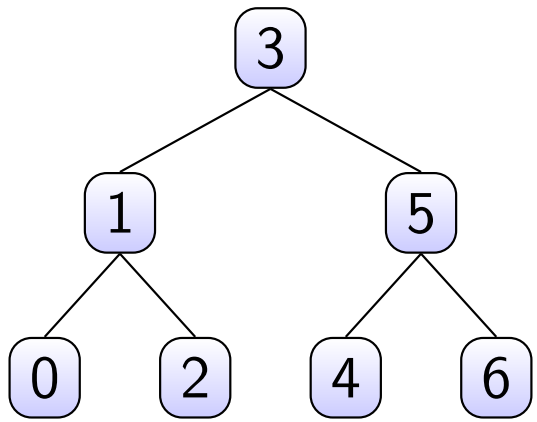
Full, Perfect, and Complete Binary Trees (cont.)

A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes)



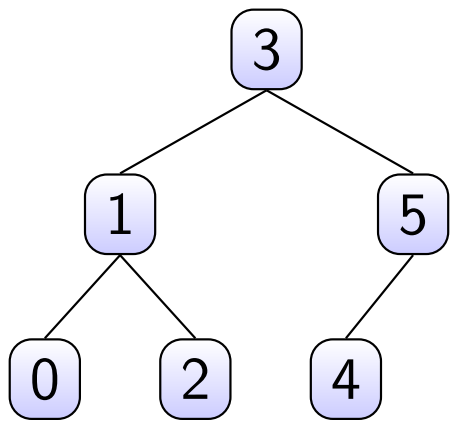
Full, Perfect, and Complete Binary Trees (cont.)

- A perfect binary tree is
  1. a full binary tree of height n
  2. all leaves have the same depth
- Item 2 is equivalent to requiring that the tree have exactly $2^n - 1$ nodes
- In this case, $n = 3$ and $2^n - 1 = 7$



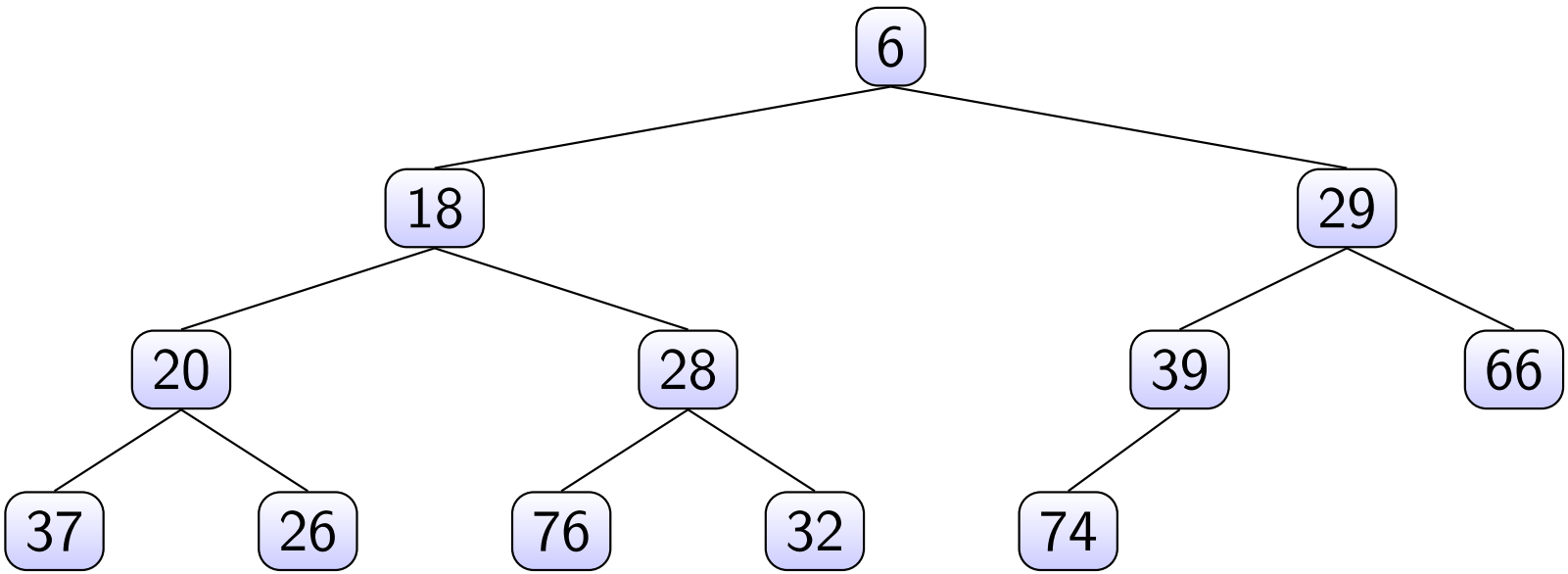
Full, Perfect, and Complete Binary Trees (cont.)

- A complete binary tree is a perfect binary tree through level $n-1$ with some extra leaf nodes at level n (the tree height), all toward the left



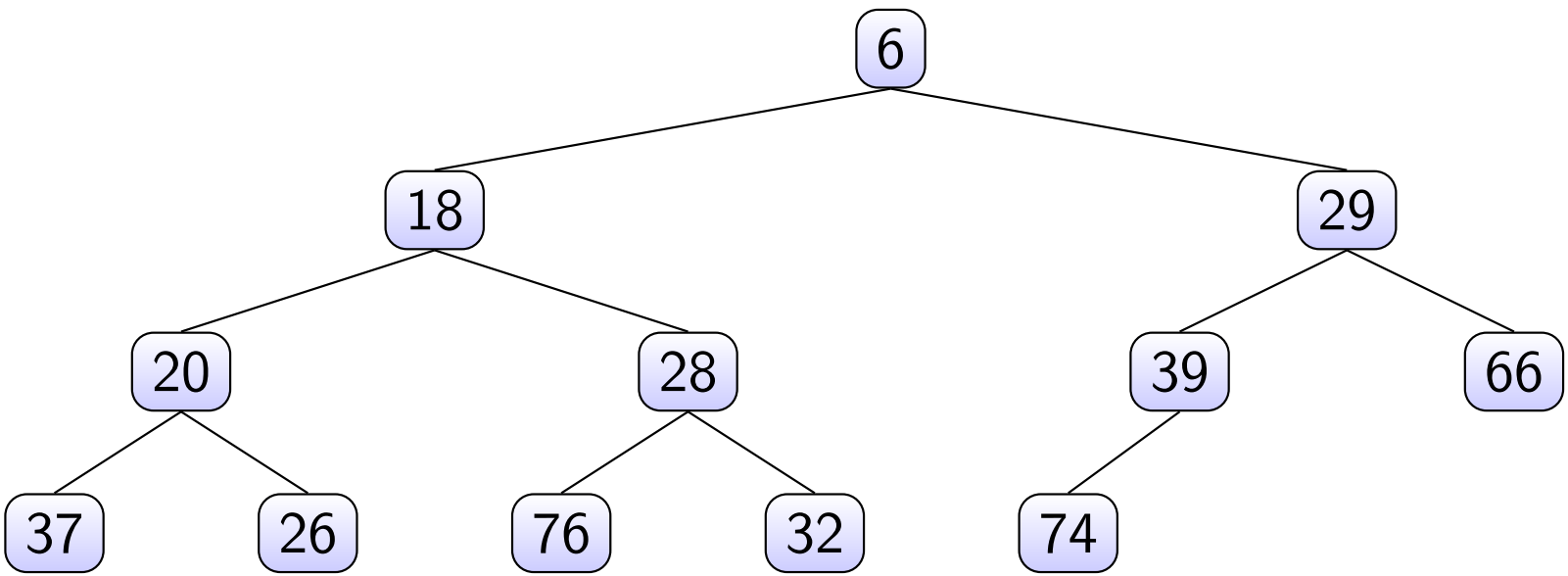
Heaps and Priority Queues

- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap

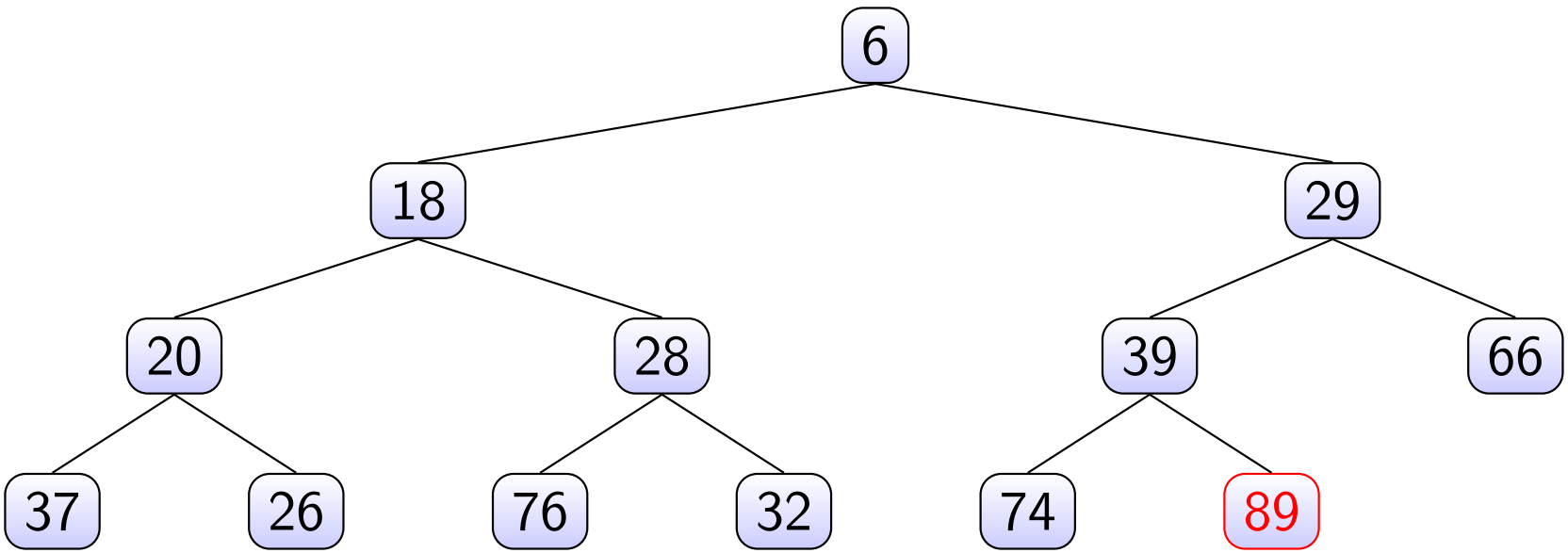


Inserting an Item e into a Heap

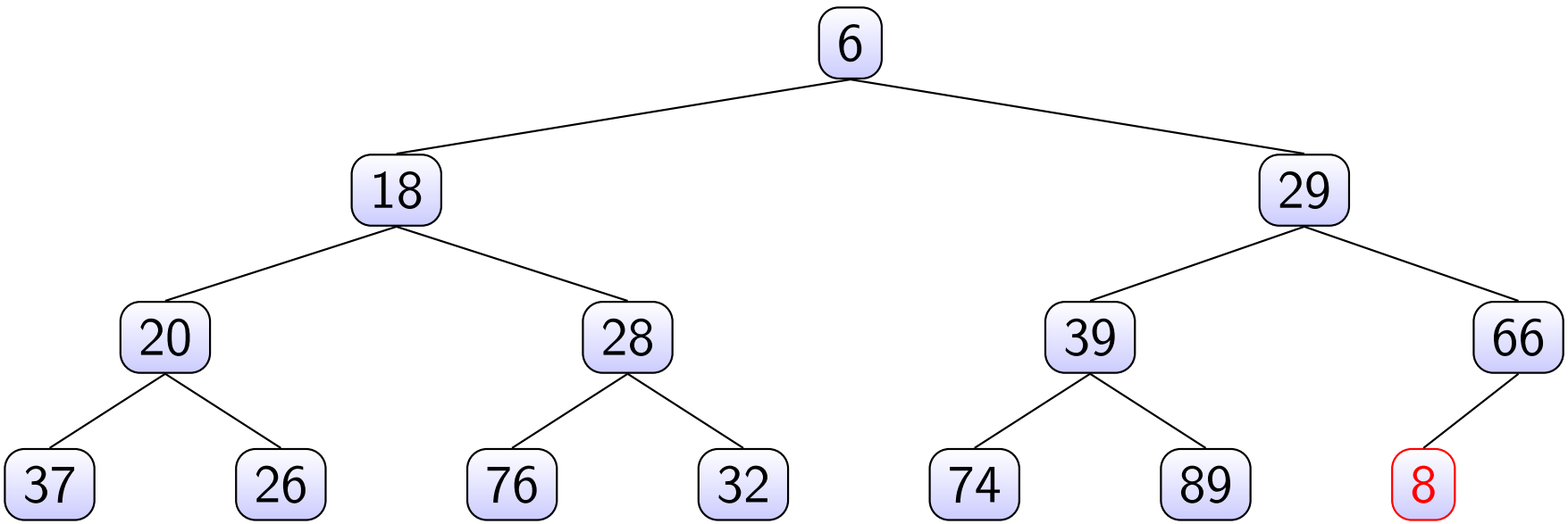
```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```



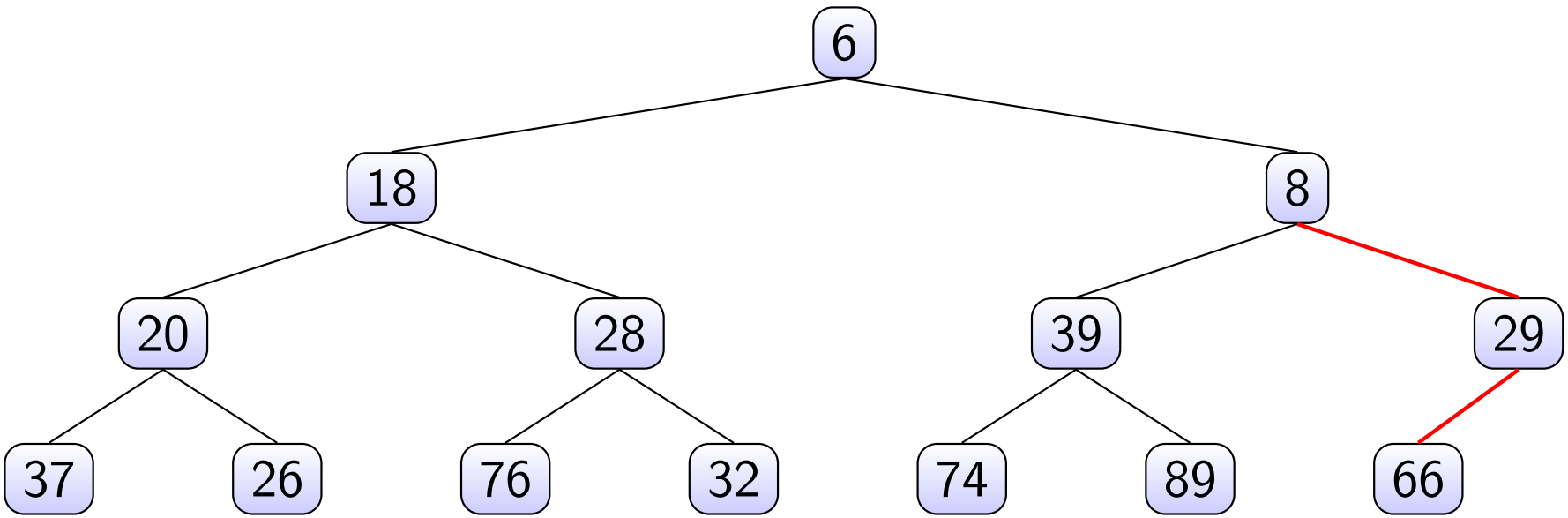
Example 1 – Insert 89



Example 2 – Insert 8



Example 2 – Insert 8 (cont.)



Performance

```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```

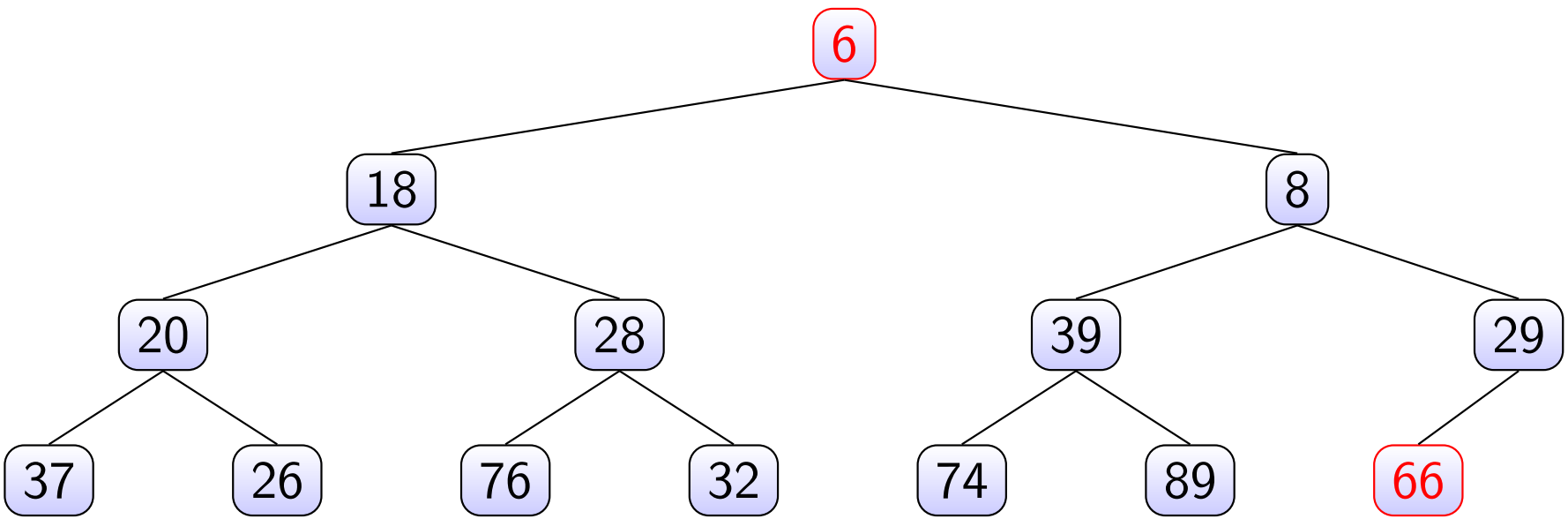
Performance

```
Insert e in the next position at the bottom of the Heap
while (e not at the root and is smaller than its root){
     Swap e with its parent, moving e up the heap
}
```

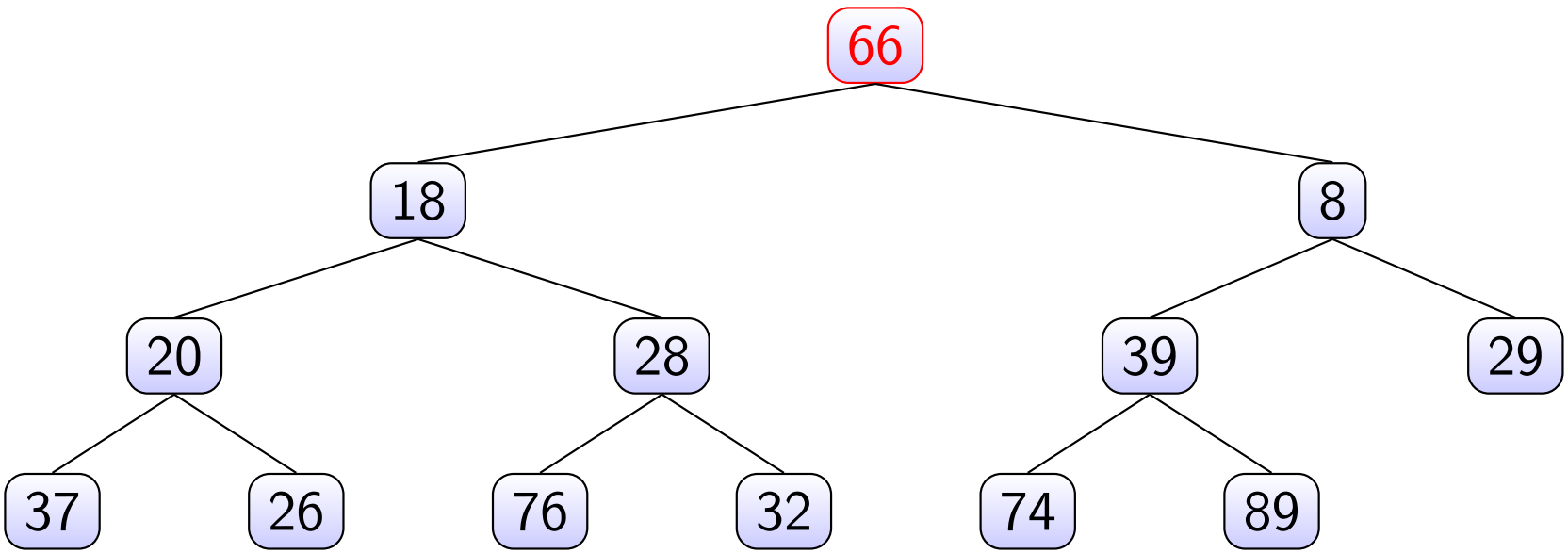
- Insertion is $\mathcal{O}(\log n)$

Removing the Item in the Root of a Heap

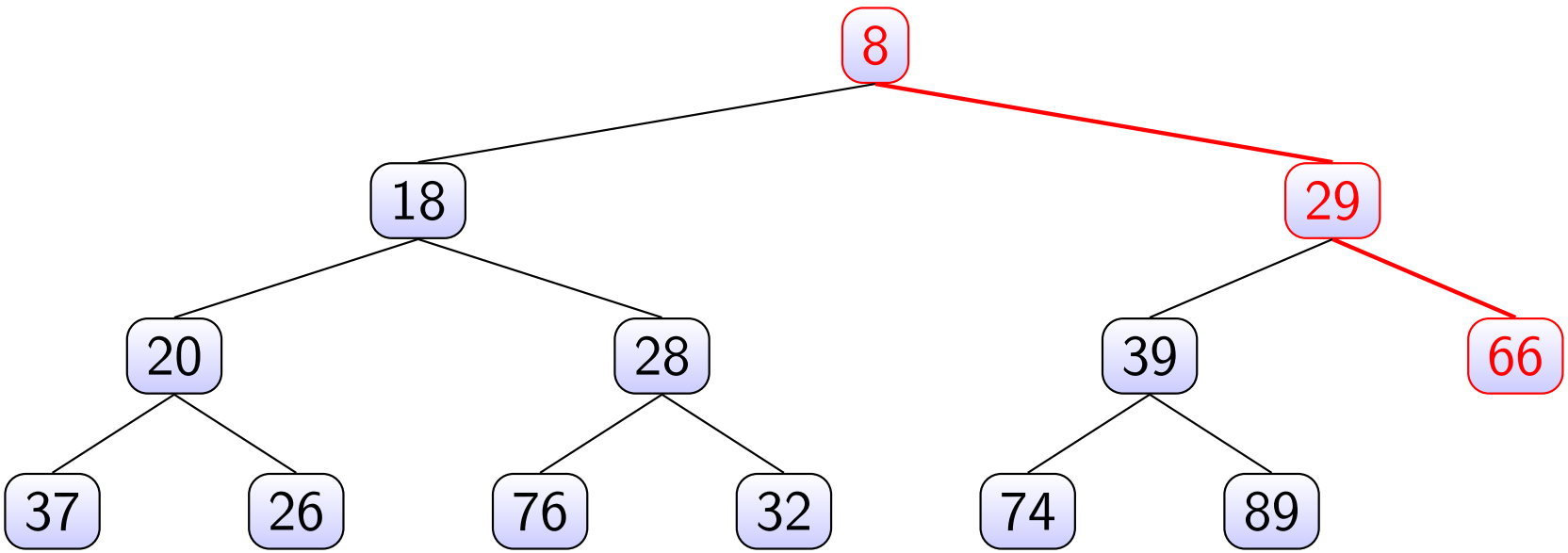
```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```



Removing the Item in the Root of a Heap



Removing the Item in the Root of a Heap



Performance

```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```

Performance

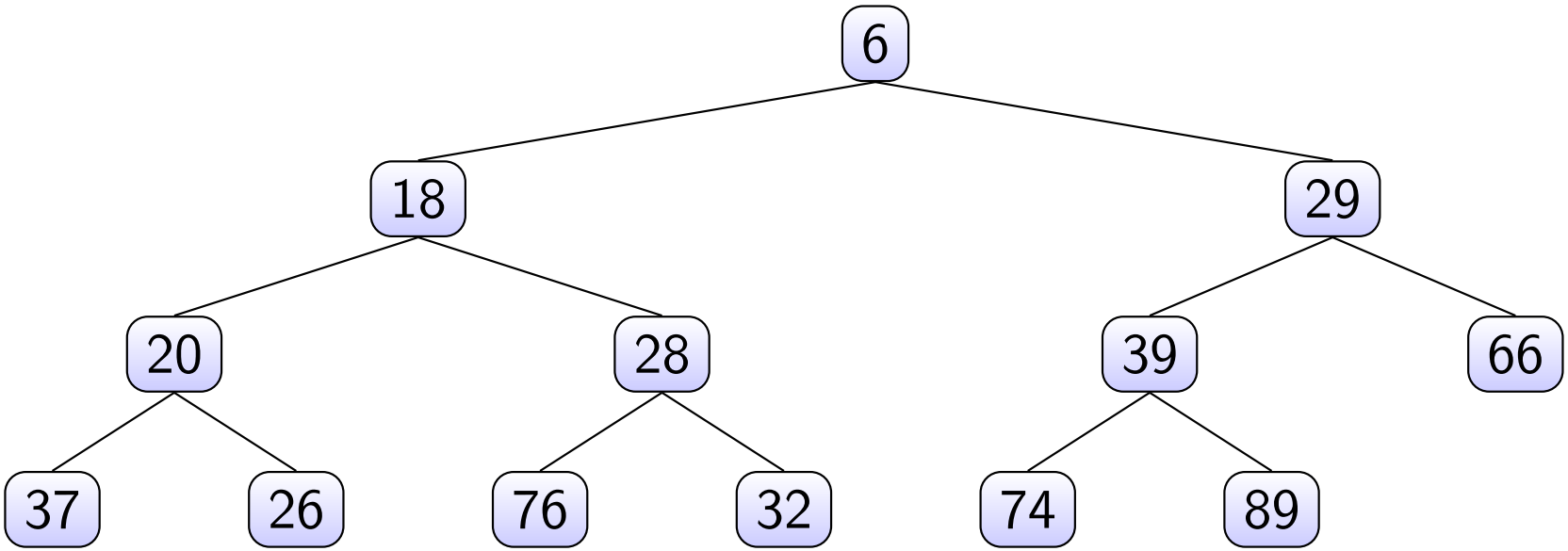
```
Remove the item in the root by replacing it with the last item
while (item LIH has children and it is larger than than either
    Swap item LIH with its smaller child, moving LIH down the
}
```

- Insertion is $\mathcal{O}(\log n)$

Implementing a Heap

- Because a heap is a complete binary tree, it can be implemented efficiently using an array rather than a linked data structure

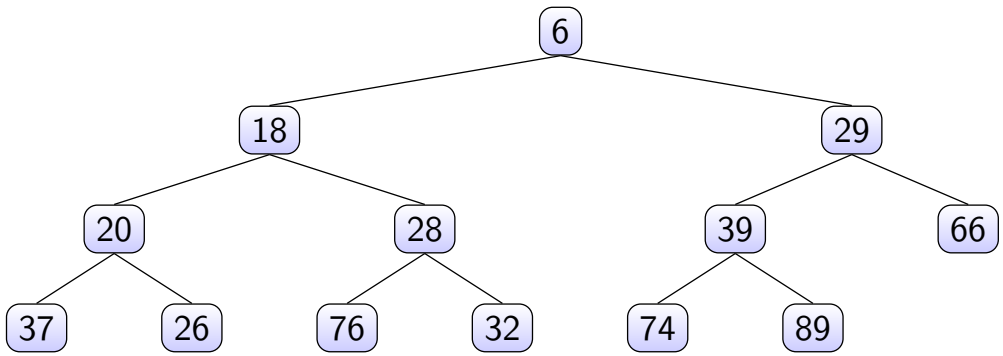
Implementing a Heap (cont.)



0	1	2	3	4	5	6	7	8	9	10	11	12
6	18	29	20	28	39	66	37	26	76	32	74	89

Node at p: left child at $2p + 1$, right child at $2p + 2$ and parent at $(p - 1)/2$.

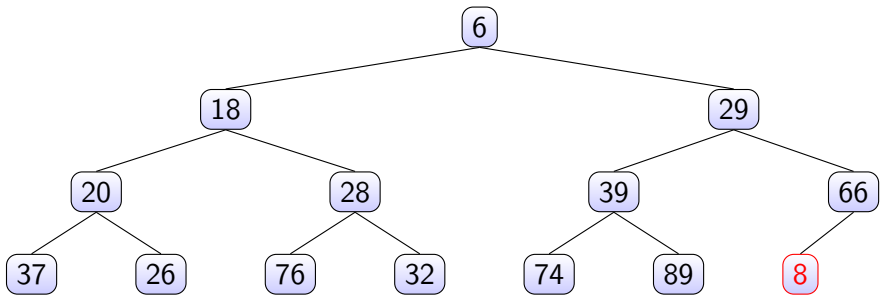
Inserting e into a Heap Implemented as an ArrayList




```
Insert e at the end of Array
child = table.size()
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	

Inserting e into a Heap Implemented as an ArrayList

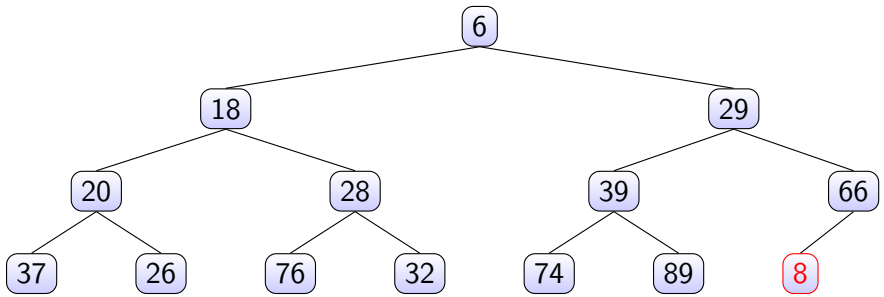



```
Insert e at the end of the Arr
child = table.size()
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

```
child=13
```

Inserting `e` into a Heap Implemented as an ArrayList

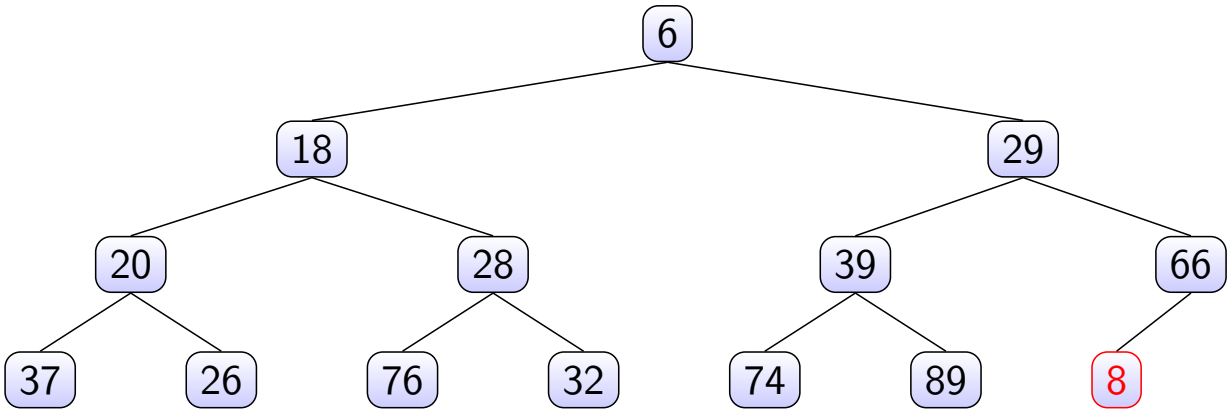



```
parent = (child-1)/2
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

```
child=13
parent=6
```

Inserting into a Heap Implemented as an ArrayList

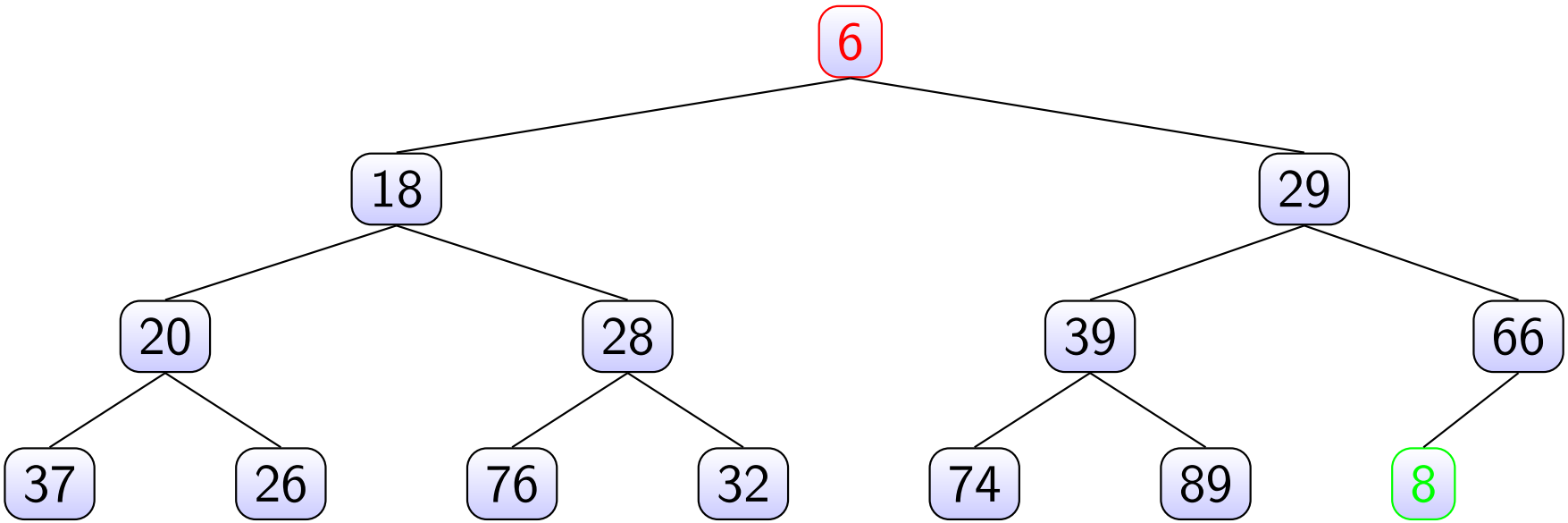



```
while  (parent>=0 and table[parent]>table[child]) {
  swap(table[parent],table[child])
  child  = parent
  parent = (child-1)/2
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
6	18	29	20	28	39	66	37	26	76	32	74	89	8

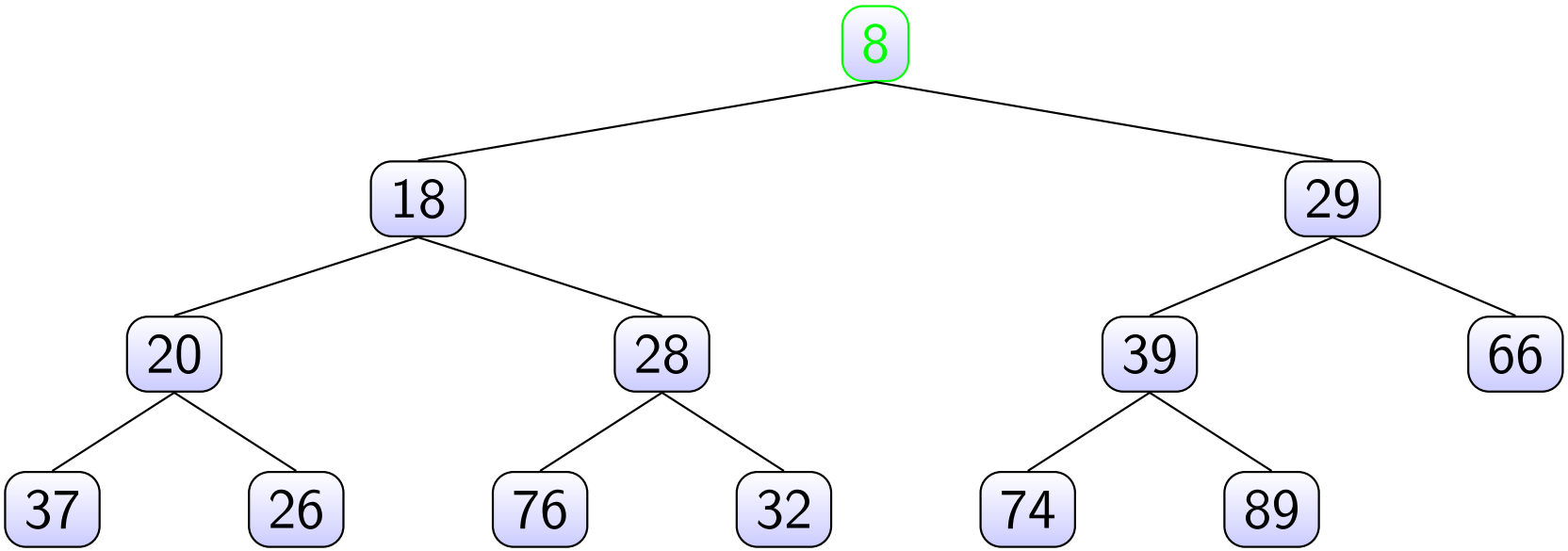
parent=6

Removal from a Heap



Remove and place last element in root

Removal from a Heap



"Trickle" root down

Heap code - Insertion

```
/**
* Insertion an item in the heap
* (1) Attach the item to the last node
* (2) Heapify it up, so it meets the
* requirement of a heap
* Time complexity: O(log n)
*/
public void add(int item) {
ensureCapacity();
heap[size] = item;
size++;
heapifyUp();
}
```

Heap code - Insertion

```
/** heapify up: swap a leaf element up
*  if it is smaller than parent, until
*  the tree is heapified
*/
private void heapifyUp() {
  int index = size - 1;

  while (hasParent(index) && parent(index) > heap[index]) {
    swap(getParentIndex(index), index);
    index = getParentIndex(index);
}
}
```

Heap code - Removal

```
/**
* Removal of the root node at the heap
* (1) Remove and place last element at root
* (2) Heapify down the root element, so it
* meets the requirement of a heap
* Time complexity: O(log n)
*/
public void poll() {
    if (size == 0) {
      throw new NoSuchElementException();
    }
    int element = heap[0];

    heap[0] = heap[size - 1];
    size--;
    heapifyDown();
}
```

Heap code - Removal

```
/** heapify down: "trickle" elements down,
* every time, swap it with the smaller child,
* until the tree is heapified
*
*/
private void heapifyDown() {
int index = 0;

while (hasLeftChild(index)) {
  int smallerChildIndex = getLeftChildIndex(index);

  if (hasRightChild(index) && rightChild(index)
  < leftChild(index)) {
    smallerChildIndex = getRightChildIndex(index);
  }
  if (heap[index] > heap[smallerChildIndex]) {
    swap(index, smallerChildIndex);
  } else {
    break;
  }
  index = smallerChildIndex;
}
}
```

Summary of Performance of the Heap

- `remove` traces a path from the root to a leaf
- `insert` traces a path from a leaf to the root
- This requires at most h steps where h is the height of the tree
- The largest full tree of height h has $2^h - 1$ nodes
- The smallest complete tree of height h has $2^{(h-1)}$ nodes
- Both `insert` and `remove` are $\mathcal{O}(\log n)$