Data Structures & Algorithms

Week 8 - Priority Queues (Binary Heaps, Skew Heaps, Applications)

Subodh Sharma, Rahul Garg

{svs,rahulgarg}@iitd.ac.in

Priority Queues

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
    - Each element has an associated **priority**

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
    - Each element has an associated **priority**
- **Motivation:**

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
  - Each element has an associated **priority**
- **Motivation:**
  - Many application tasks running on OS, and you press ESC (or Ctrl-C)! What would you expect?

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
  - Each element has an associated **priority**
- **Motivation:**
  - Many application tasks running on OS, and you press ESC (or Ctrl-C)! What would you expect?
  - What would have happened if every task had the same priority?

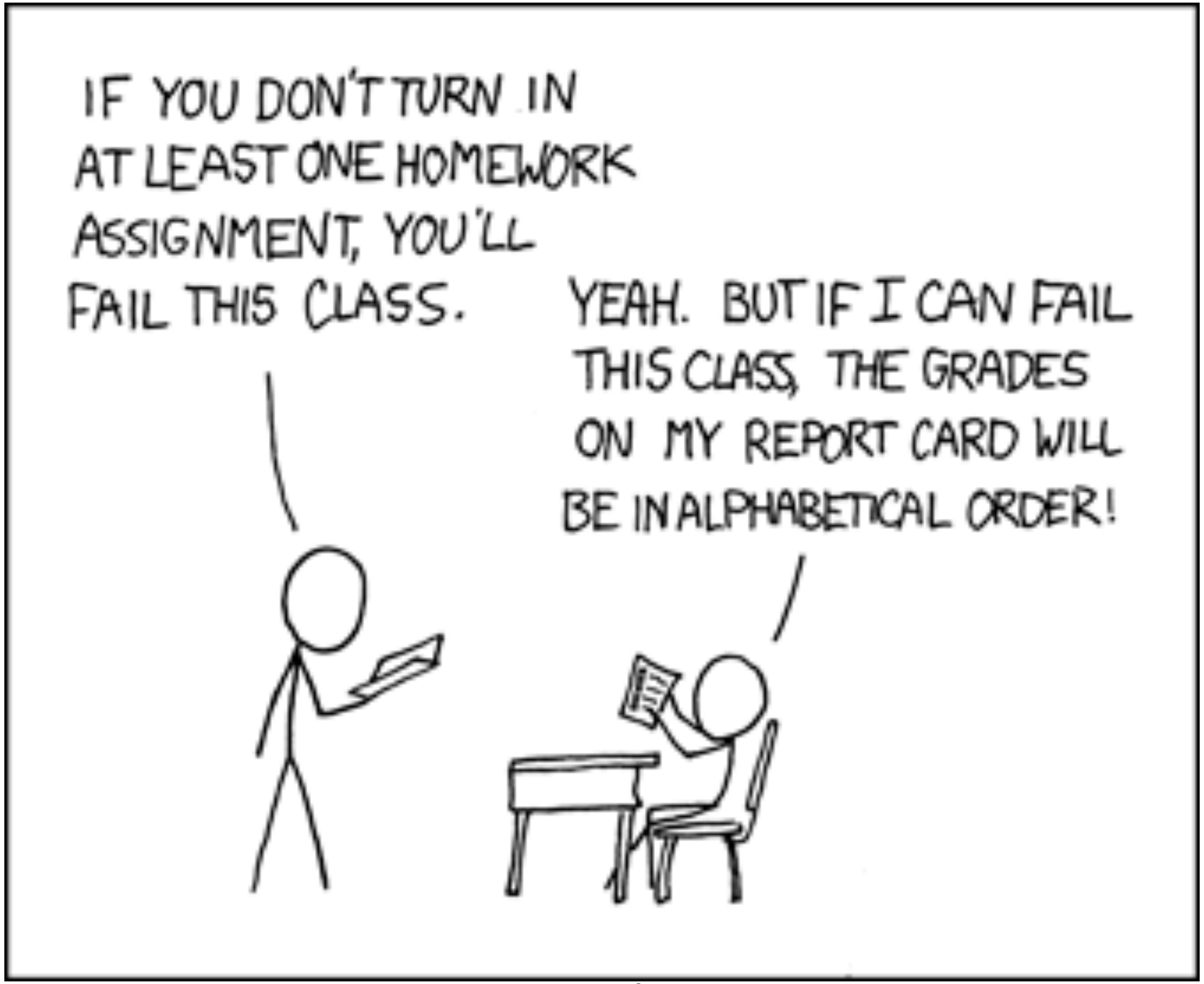
Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
    - Each element has an associated **priority**
- **Motivation:**
    - Many application tasks running on OS, and you press ESC (or Ctrl-C)! What would you expect?
    - What would have happened if every task had the same priority?
- **Applications:**

Priority Queues

- An **ADT** similar to a Queue or Stack but with a caveat
  - Each element has an associated **priority**
- **Motivation:**
  - Many application tasks running on OS, and you press ESC (or Ctrl-C)! What would you expect?
  - What would have happened if every task had the same priority?
- **Applications:**
  - Scheduling, Algorithmic efficiency (Spanning Trees, Shortest Paths etc.), Simulation Systems (Discrete Event Simulation, etc.), Network Traffic Mgmt. (routing pkts with different service reqs.), E-commerce, Load balancing, etc.

On Priorities



Courtesy: XKCD Comics

On Priorities

On Priorities

- Priorities help rank the elements in a Priority Queue with a **total order relation**

On Priorities

- Priorities help rank the elements in a Priority Queue with a **total order relation**

- **Total order relation:**

On Priorities

- Priorities help rank the elements in a Priority Queue with a **total order relation**

- **Total order relation:**

  - **Reflexive:** $a \leq a$

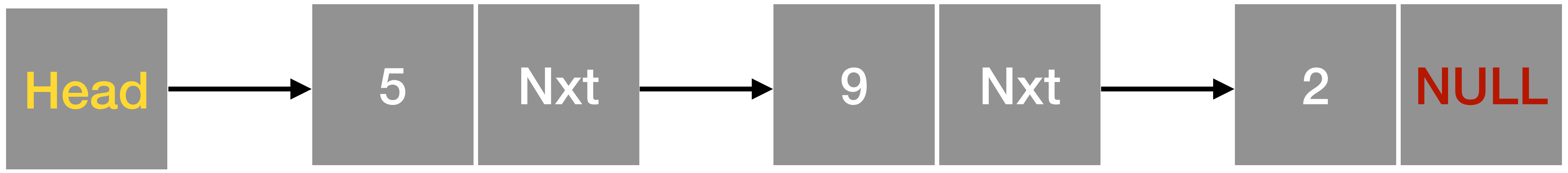
On Priorities

- Priorities help rank the elements in a Priority Queue with a **total order relation**
- **Total order relation:**
  - **Reflexive:** $a \leq a$
  - **Antisymmetric:** if $a_1 \leq a_2$ and $a_2 \leq a_1$, then $a_1 = a_2$

On Priorities

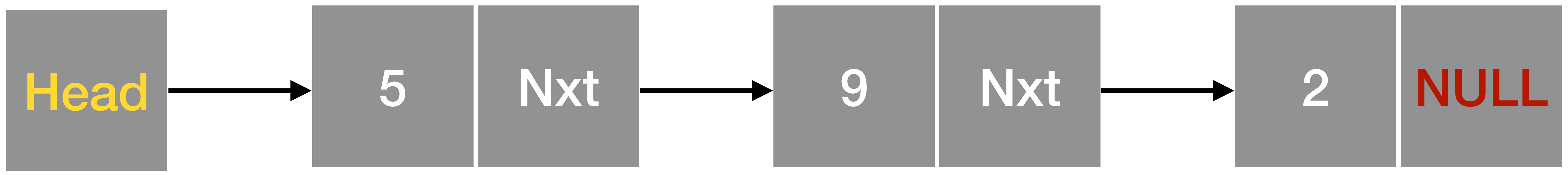
- Priorities help rank the elements in a Priority Queue with a **total order relation**
- **Total order relation:**
  - **Reflexive:** $a \leq a$
  - **Antisymmetric:** if $a_1 \leq a_2$ and $a_2 \leq a_1$, then $a_1 = a_2$
  - **Transitive:** if $a_1 \leq a_2$ and $a_2 \leq a_3$, then $a_1 \leq a_3$

Priority Queue: Model



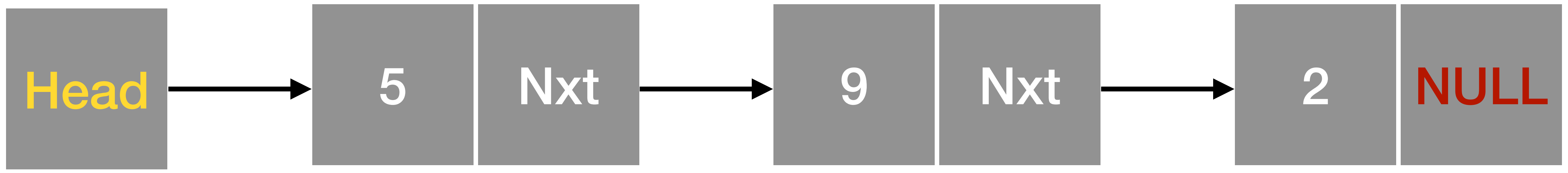
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`



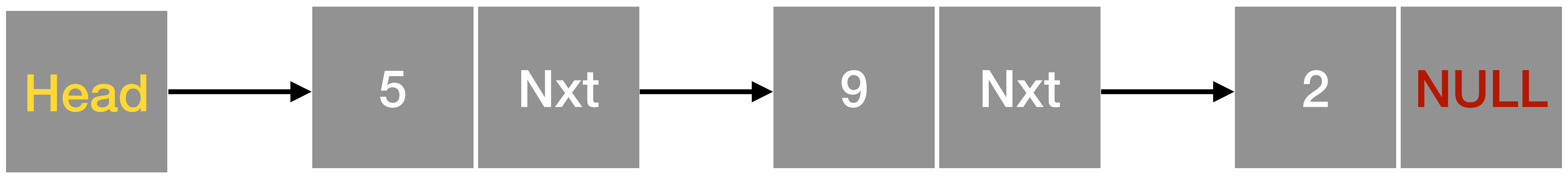
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:



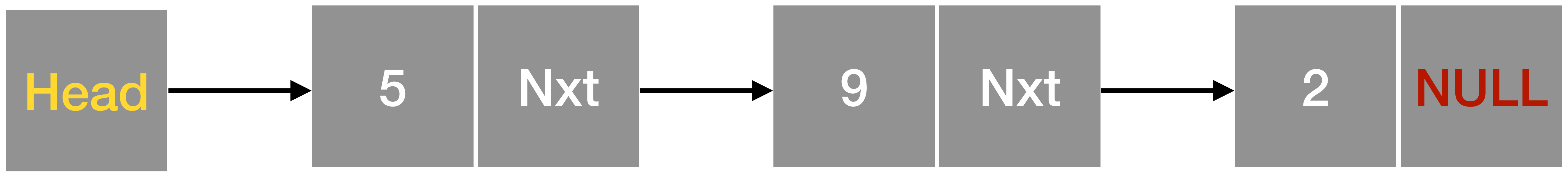
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:



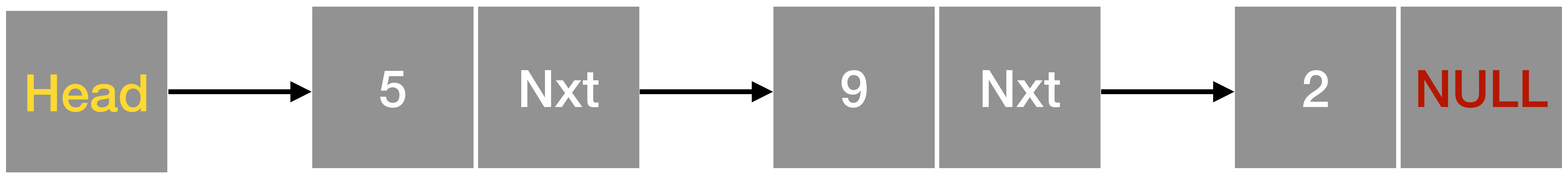
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
    - Using simple **linked lists**:
        - Insertions at the front — O(1)



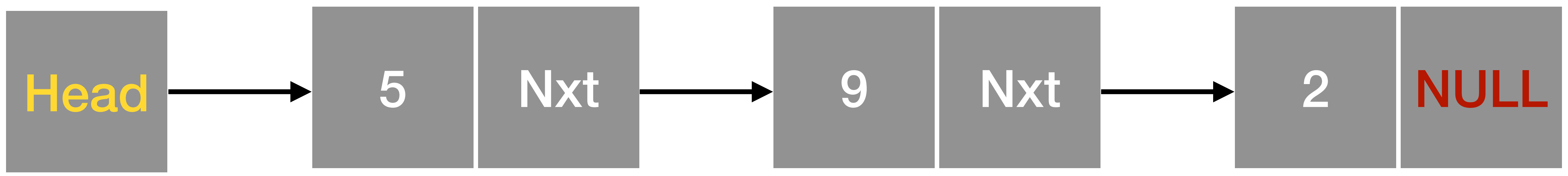
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
    - Using simple **linked lists**:
        - Insertions at the front — O(1)
        - Deleting the minimum — O(N)



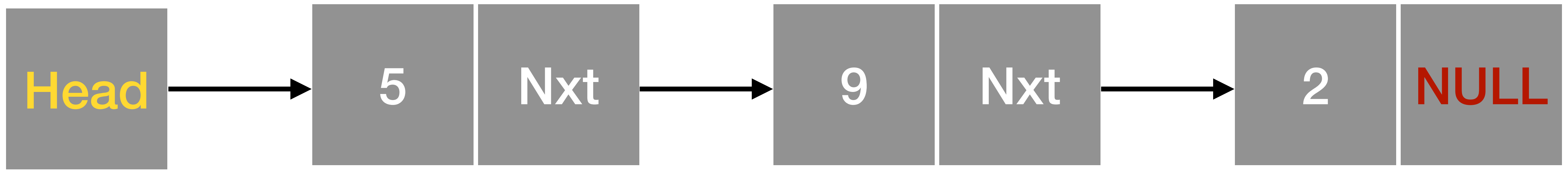
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)
  - Using **linked lists** that **remain sorted:**



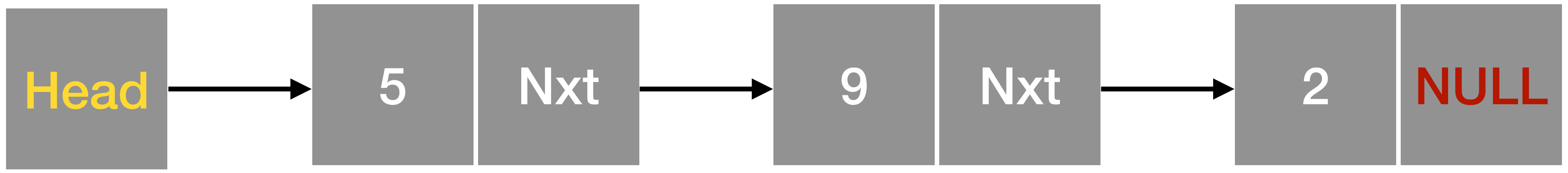
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)
  - Using **linked lists** that **remain sorted:**
    - Insertions — O(N)

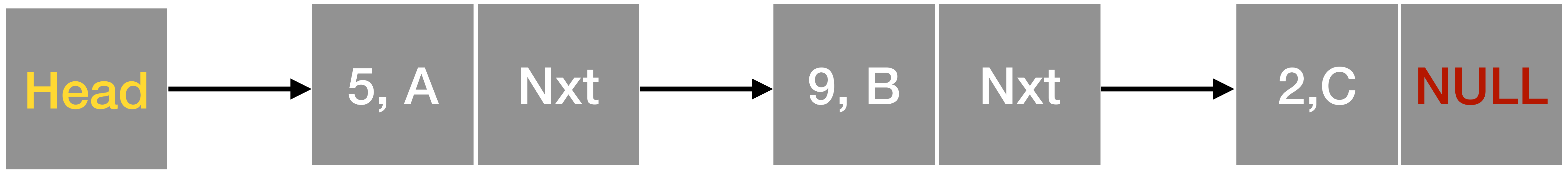


Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)
  - Using **linked lists** that **remain sorted:**
    - Insertions — O(N)
    - DeleteMin — O(1)

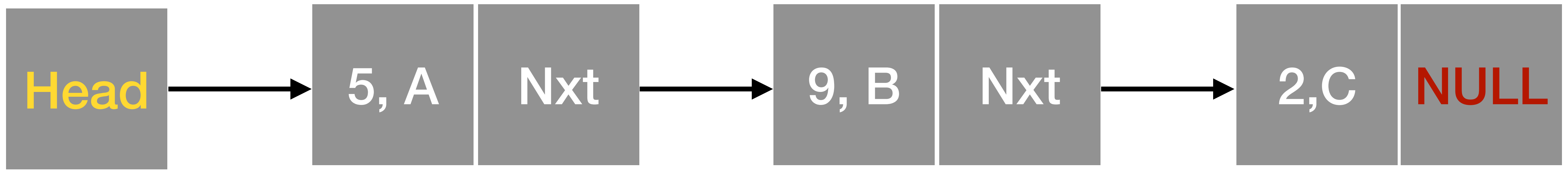


Priority Queue: Model



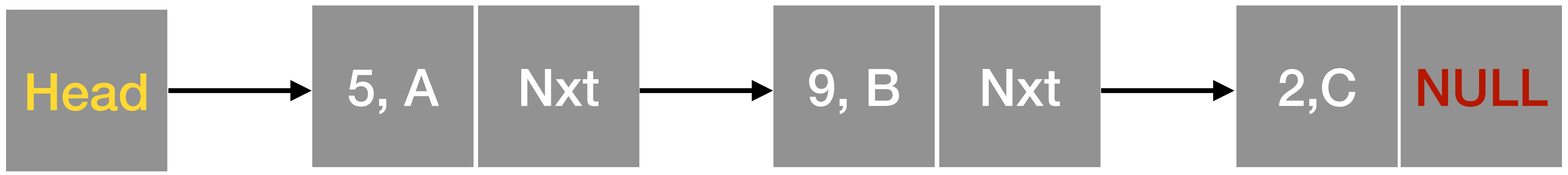
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`



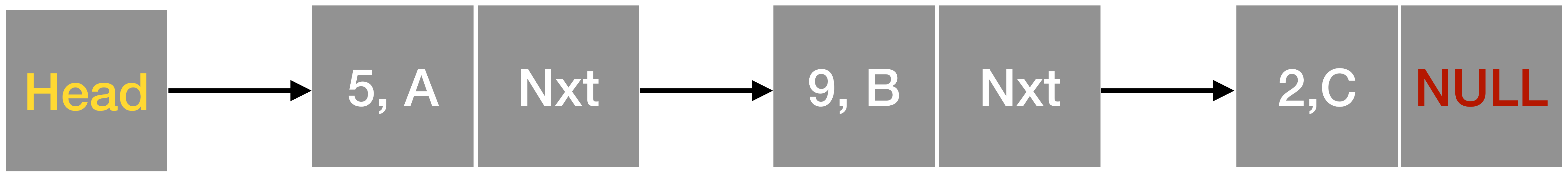
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:



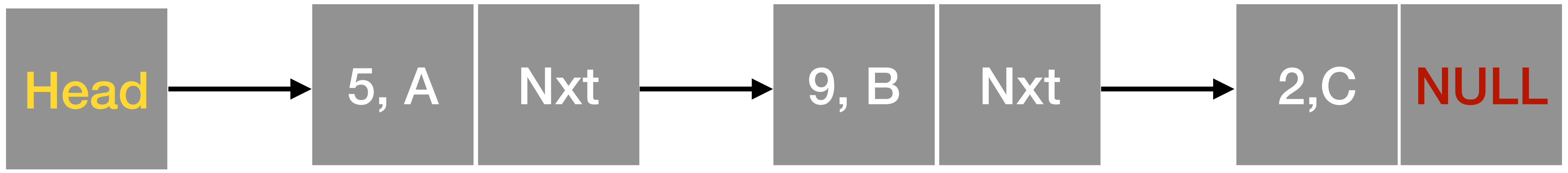
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:



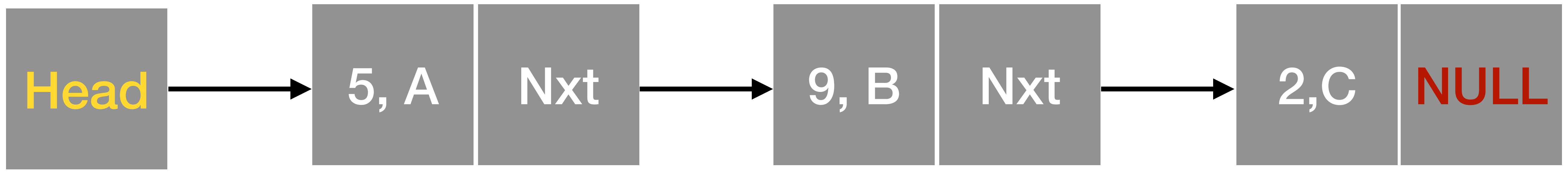
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)



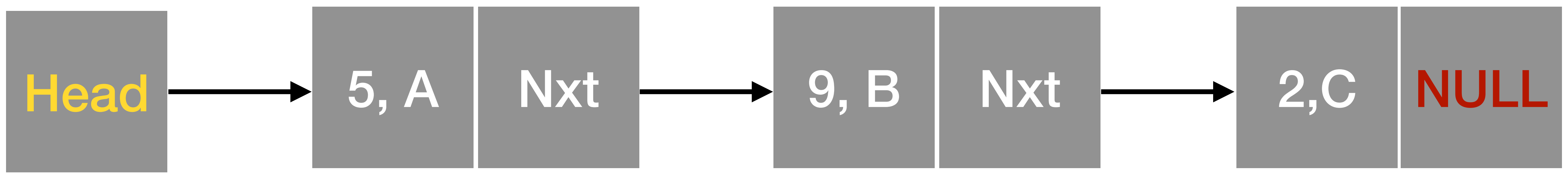
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)



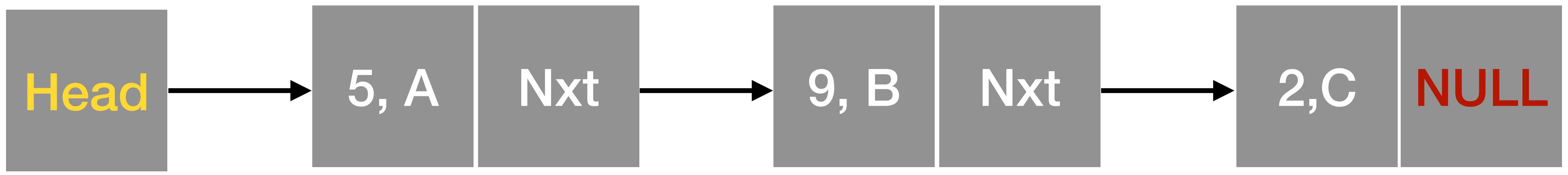
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
    - Using simple **linked lists**:
        - Insertions at the front — O(1)
        - Deleting the minimum — O(N)
    - Using **linked lists** that **remain sorted:**



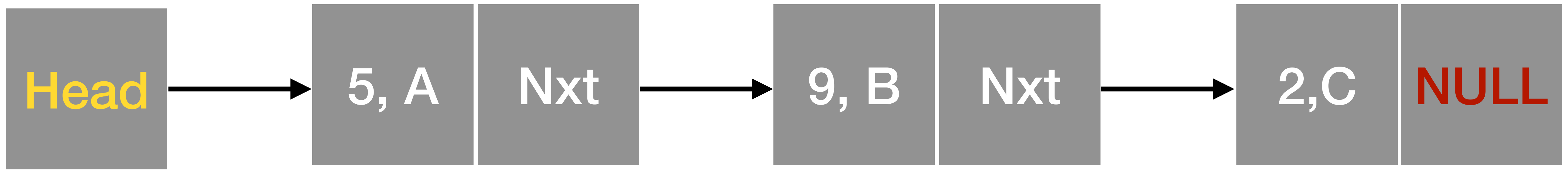
Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)
  - Using **linked lists** that **remain sorted:**
    - Insertions — O(N)



Priority Queue: Model

- Supported operations: `Insert` and `DeleteMin`
- Various implementation of PQ:
  - Using simple **linked lists**:
    - Insertions at the front — O(1)
    - Deleting the minimum — O(N)
  - Using **linked lists** that **remain sorted:**
    - Insertions — O(N)
    - DeleteMin (from front)— O(1)



(Binary) Heap

(Binary) Heap

- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes

(Binary) Heap

- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes
- Heaps have two important properties:

(Binary) Heap

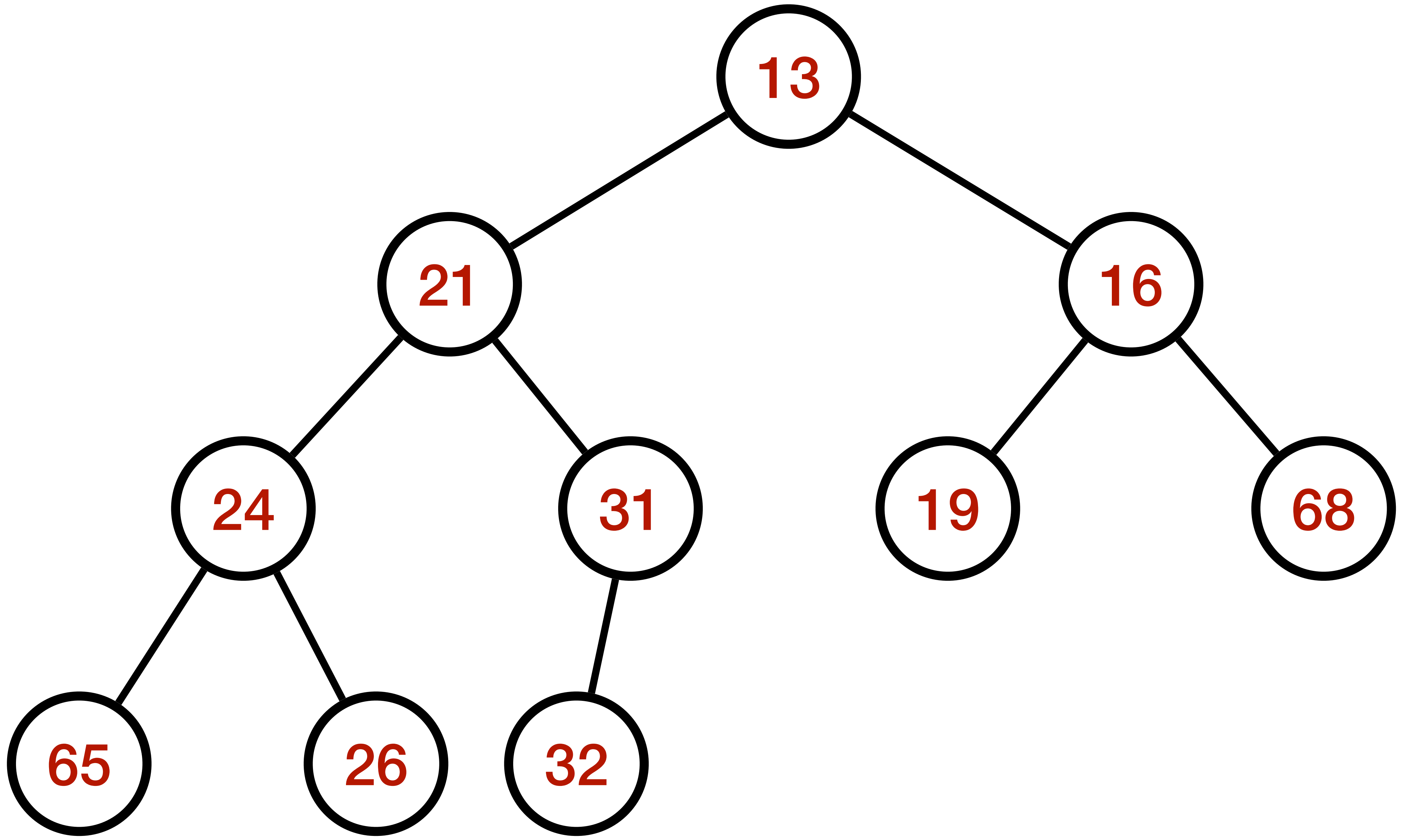
- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes
- Heaps have two important properties:
  - **Structure Property:** Heap is completely filled with the exception of the last level.

(Binary) Heap

- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes
- Heaps have two important properties:
  - **Structure Property:** Heap is completely filled with the exception of the last level.
    - The last level is left-filled.

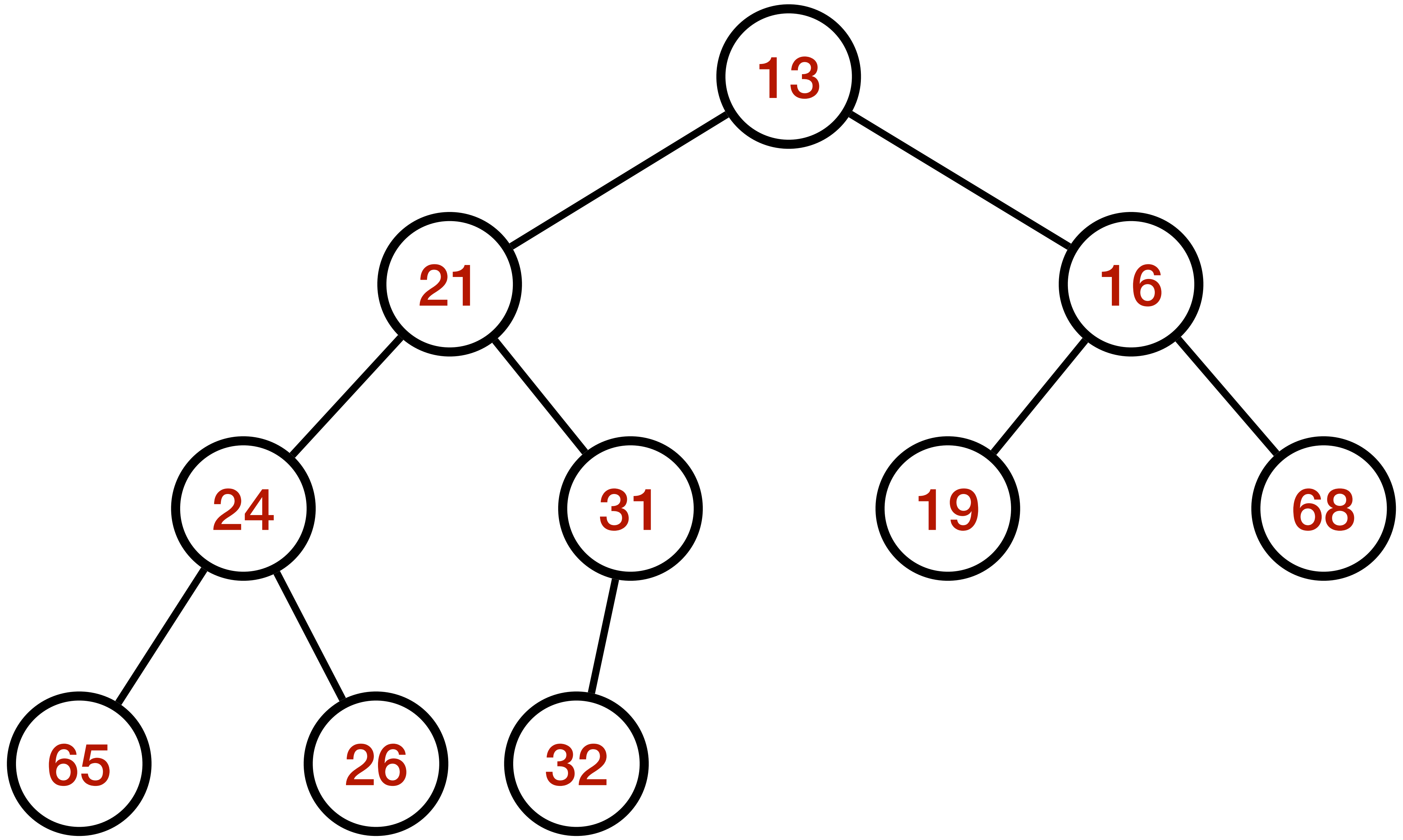
(Binary) Heap

- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes
- Heaps have two important properties:
    - **Structure Property:** Heap is completely filled with the exception of the last level.
        - The last level is left-filled.



(Binary) Heap

- Heap is a **binary tree** that stores priority or priority-value pairs at its nodes
- Heaps have two important properties:
  - **Structure Property:** Heap is completely filled with the exception of the last level.
    - The last level is left-filled.
  - **Order Property**: Every node should be smaller than all of its descendants



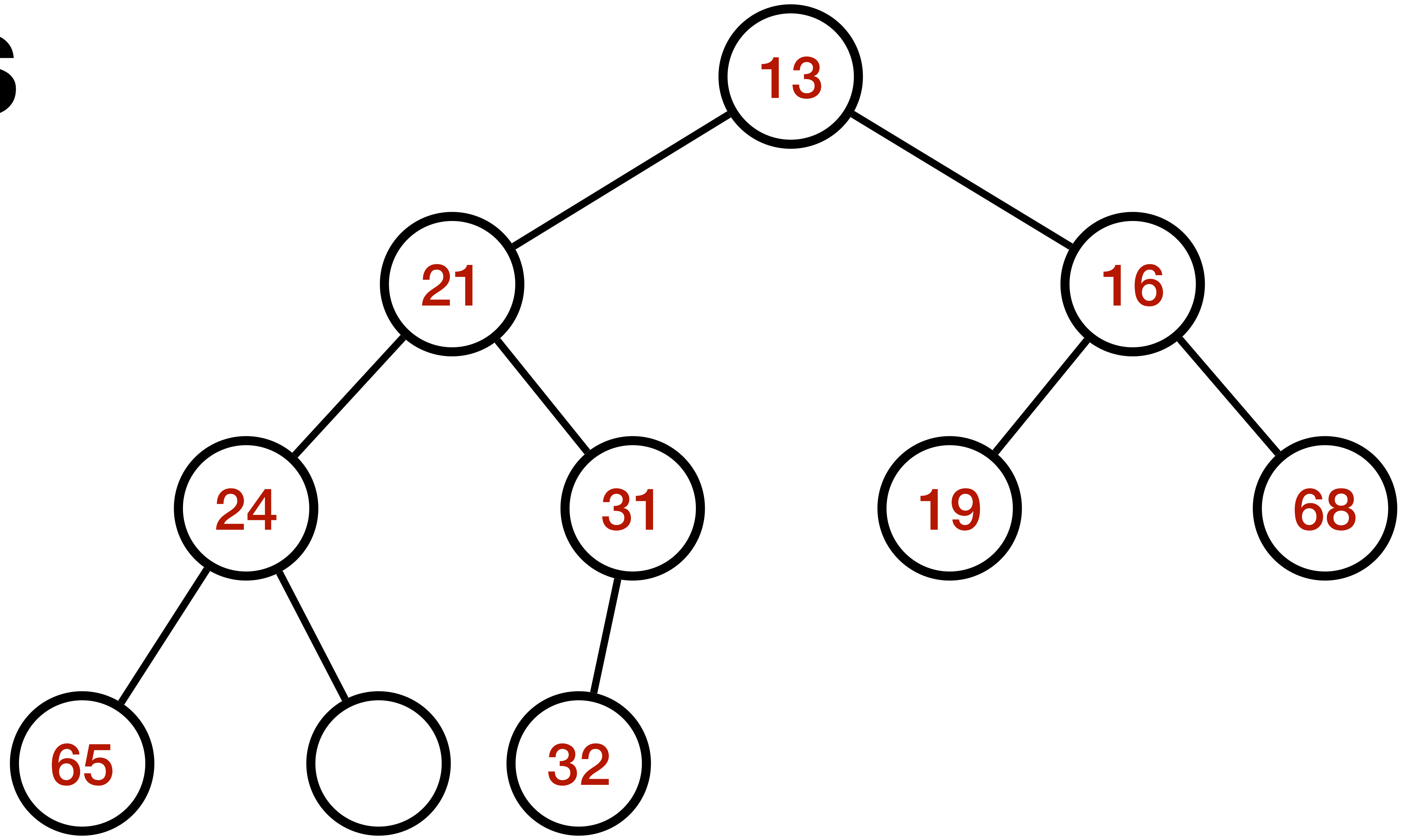
Examples of Non Heaps

Examples of Non Heaps

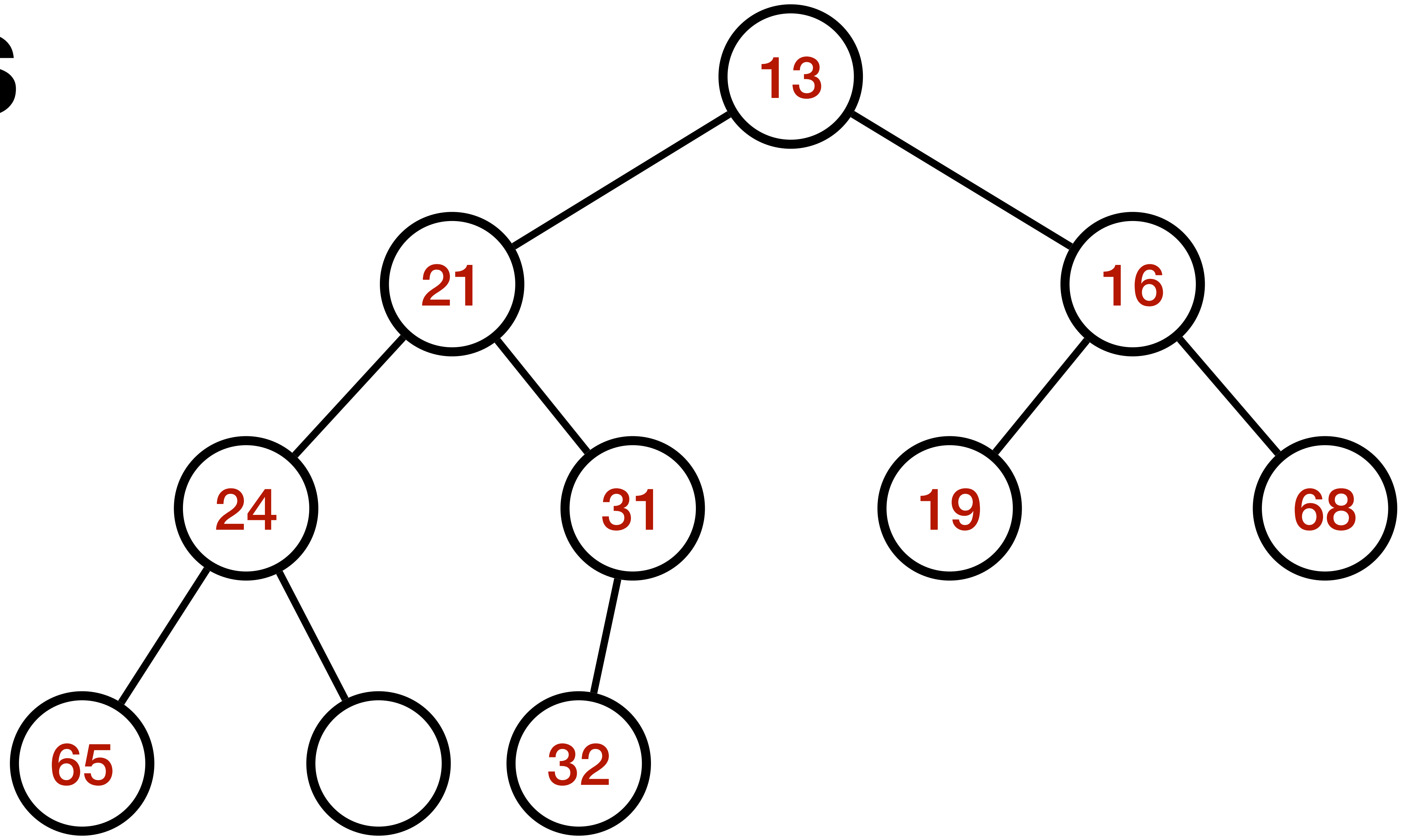
- Structure property violation

Examples of Non Heaps

- Structure property violation



Examples of Non Heaps

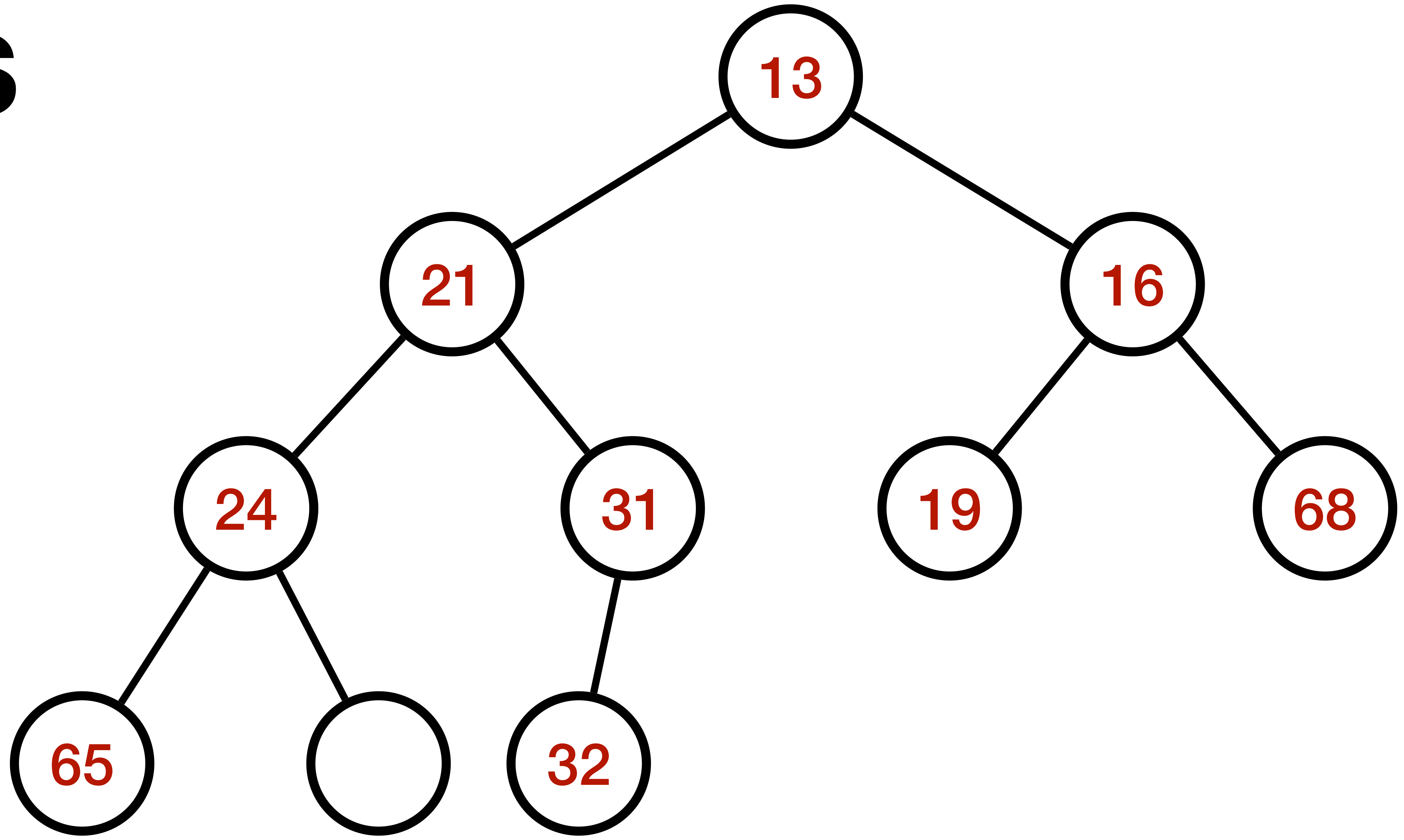


- Structure property violation

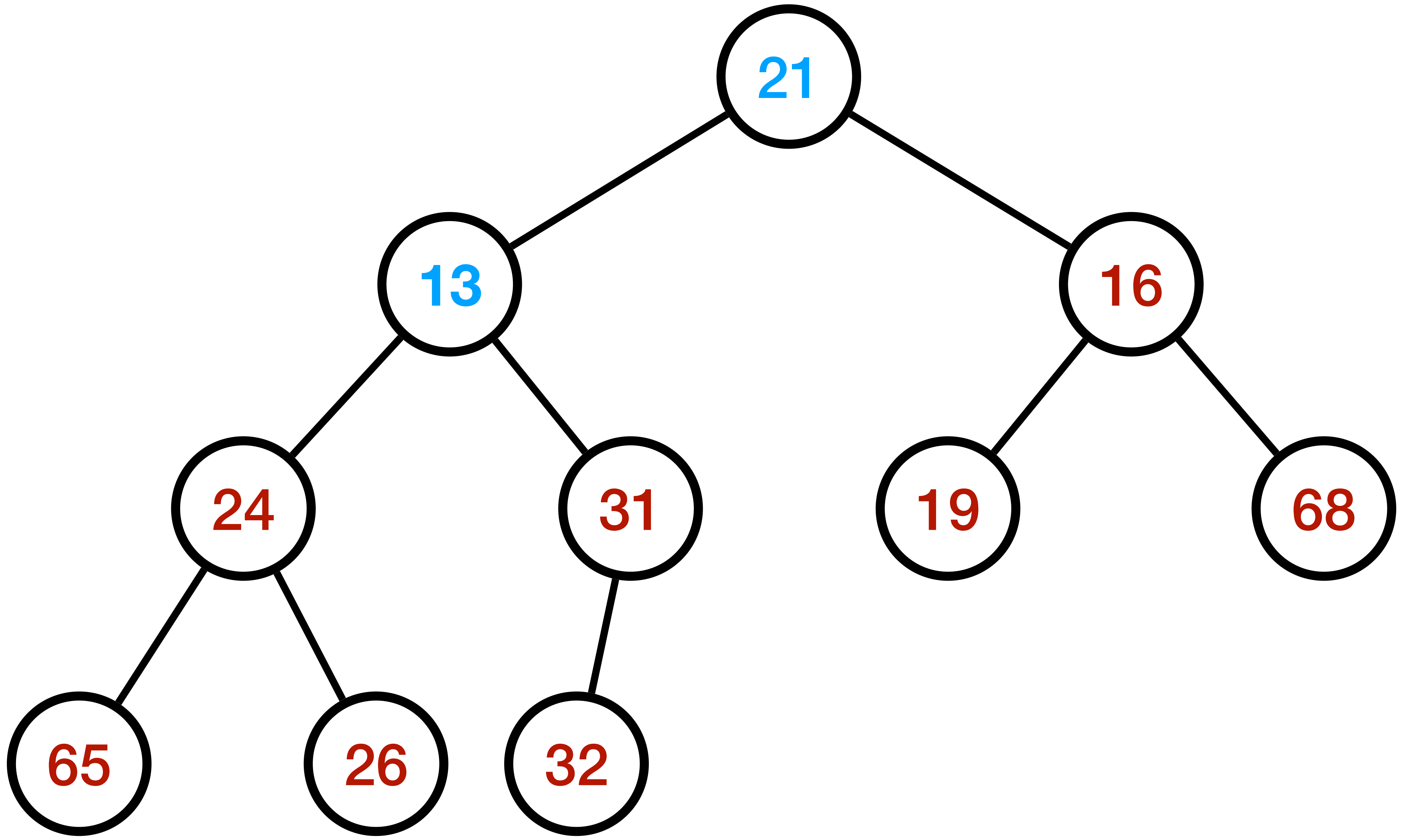
- Order property violation

Examples of Non Heaps

- Structure property violation



- Order property violation



Where is the minimum element?

- Minimum element sits at the top of the heap — root of the heap!
- Why?
  - Otherwise the heap order property would be violated.
- Thus, finding the minimum element can be accomplished in O(1) time.

Height of the Heap

Height of the Heap

- Suppose a heap of n nodes has height h

Height of the Heap

- Suppose a heap of n nodes has height h
- Complete binary tree of height h has $2^{h+1} - 1$nodes

Height of the Heap

- Suppose a heap of n nodes has height h
- Complete binary tree of height h has $2^{h+1} - 1$ nodes
- Hence $2^h - 1 < n \leq 2^{h+1} - 1$

Height of the Heap

- Suppose a heap of n nodes has height h
- Complete binary tree of height h has $2^{h+1} - 1$nodes
- Hence $2^h - 1 < n \leq 2^{h+1} - 1$
- Thus, $h = \lfloor log_2 n \rfloor$

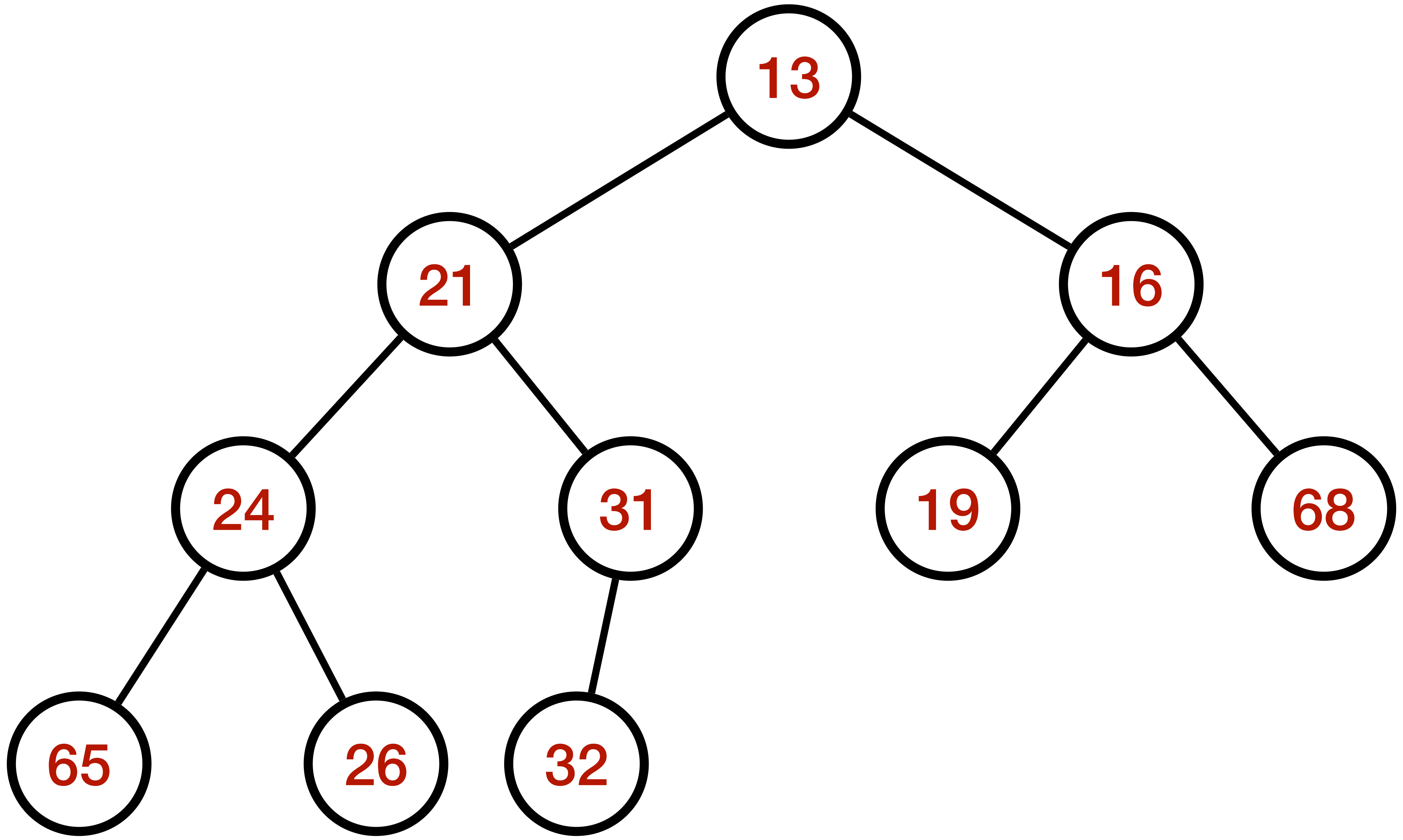
Implementing Heaps

Implementing Heaps

- **Observation:** Complete binary tree is so regular that it can be represented by arrays instead of pointers

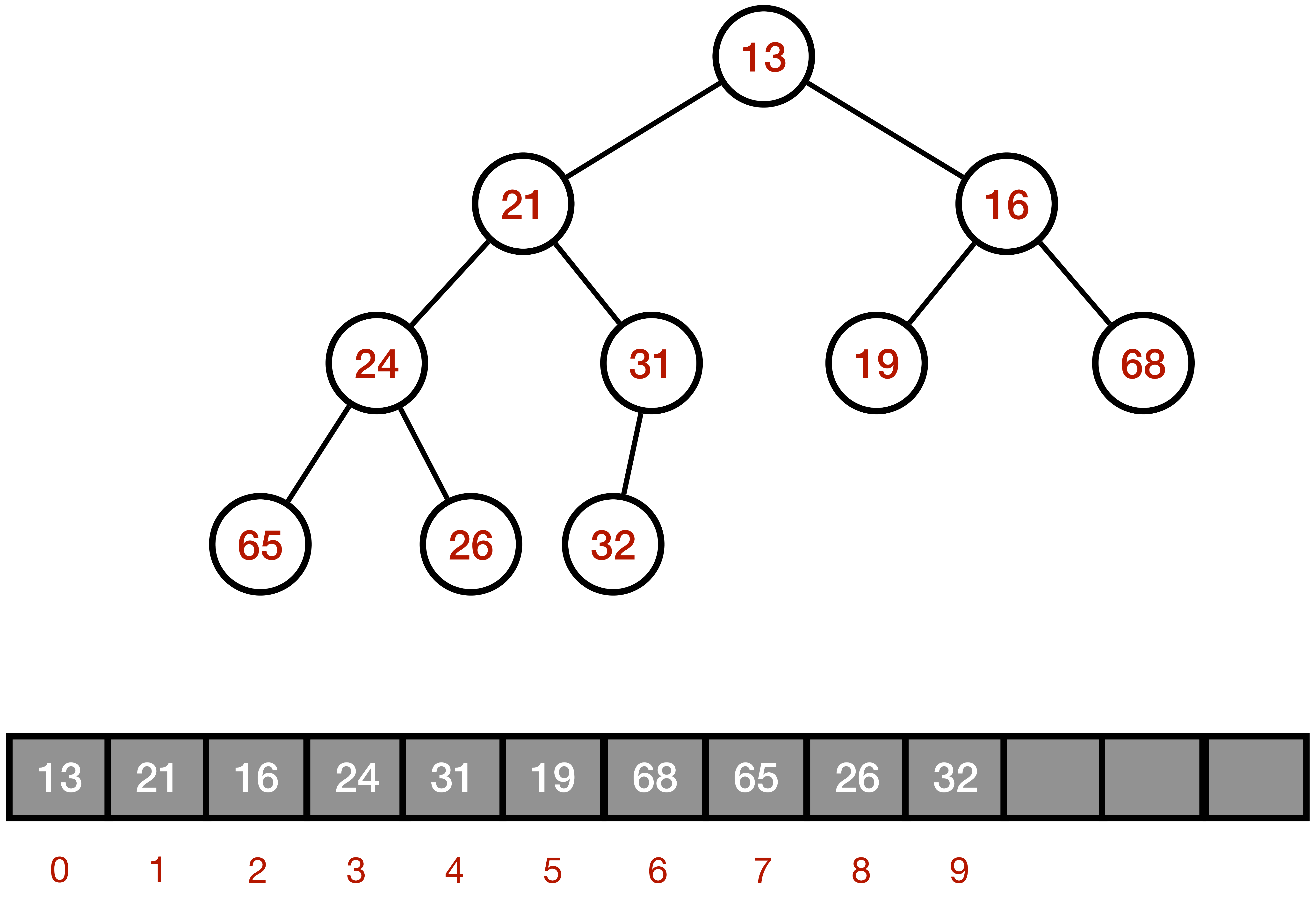
Implementing Heaps

- **Observation:** Complete binary tree is so regular that it can be represented by arrays instead of pointers



Implementing Heaps

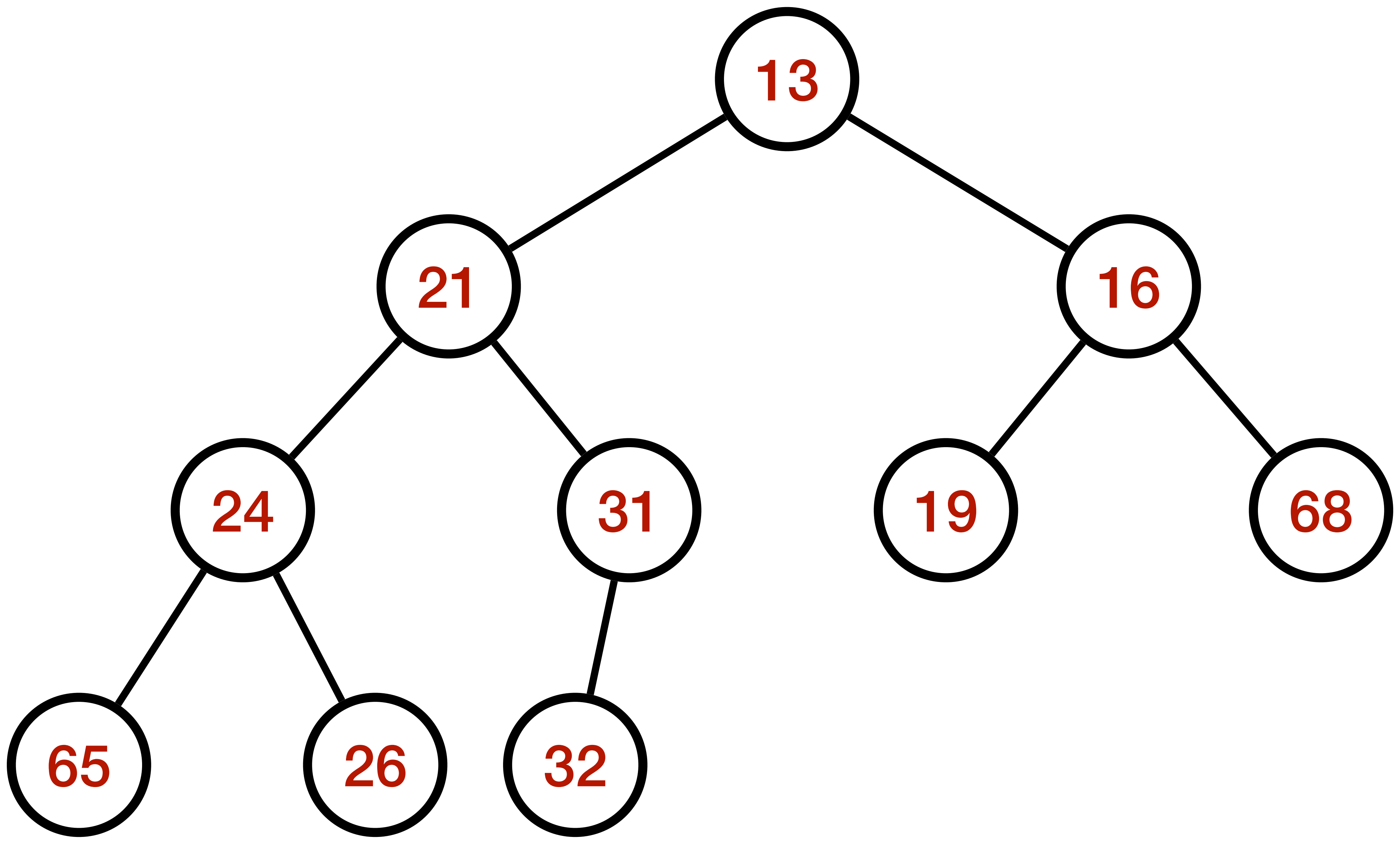
- **Observation:** Complete binary tree is so regular that it can be represented by arrays instead of pointers

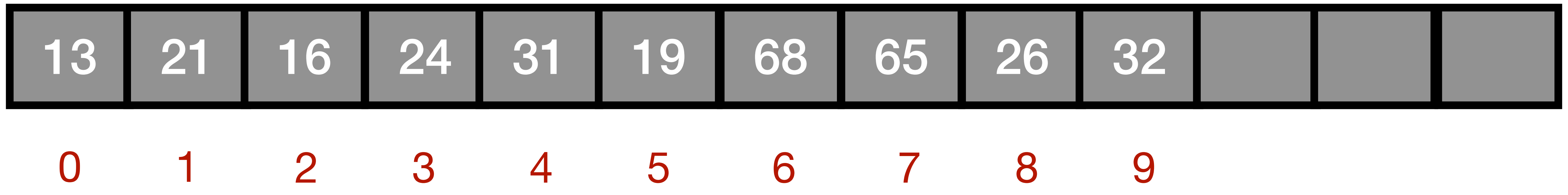


Implementing Heaps

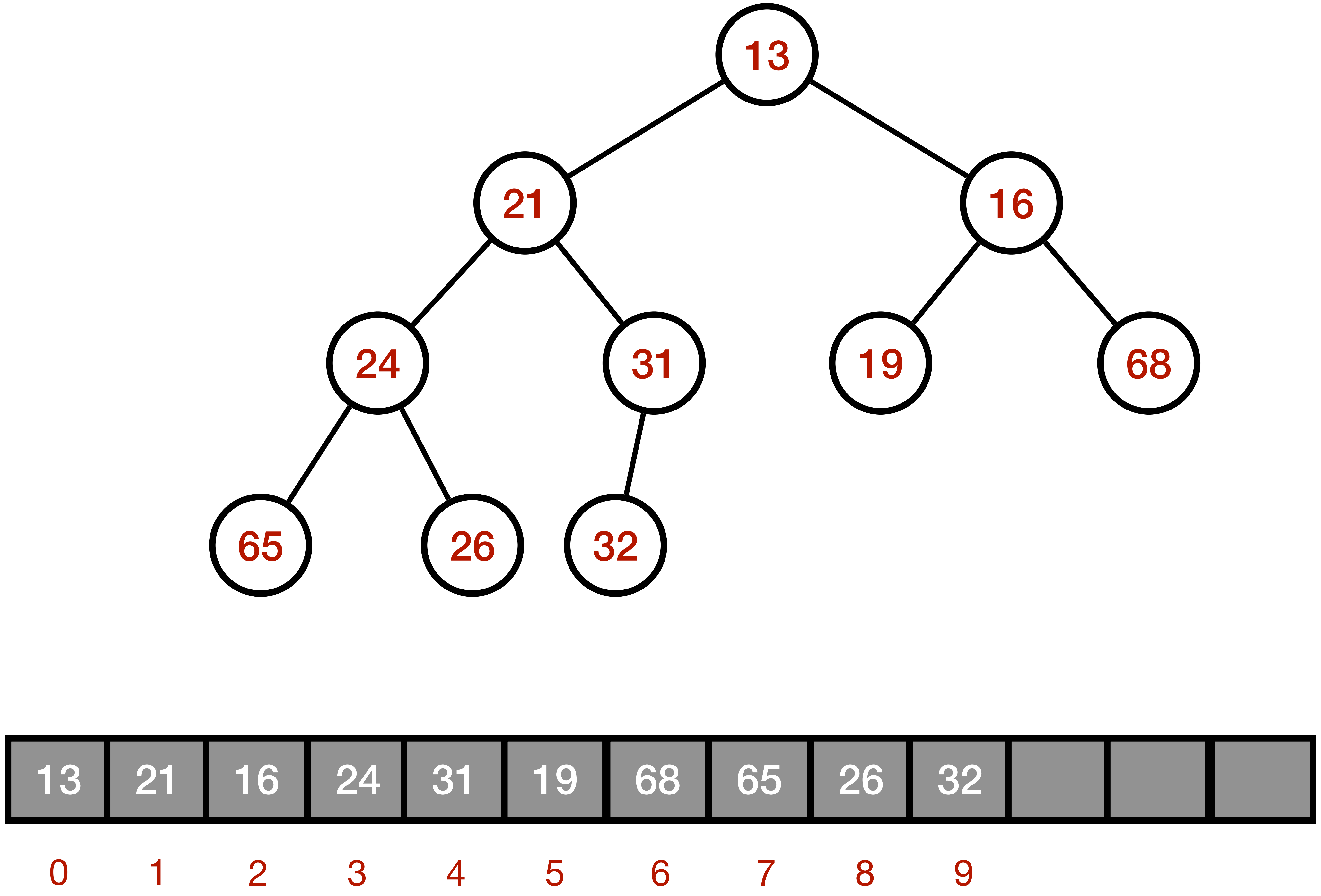
- **Observation:** Complete binary tree is so regular that it can be represented by arrays instead of pointers

```
int getParentIndex(int i) {
    return (i - 1) / 2;
}

int getLeftChildIndex(int i) {
    return 2 * i + 1;
}

int getRightChildIndex(int i) {
    return 2 * i + 2;
}
```



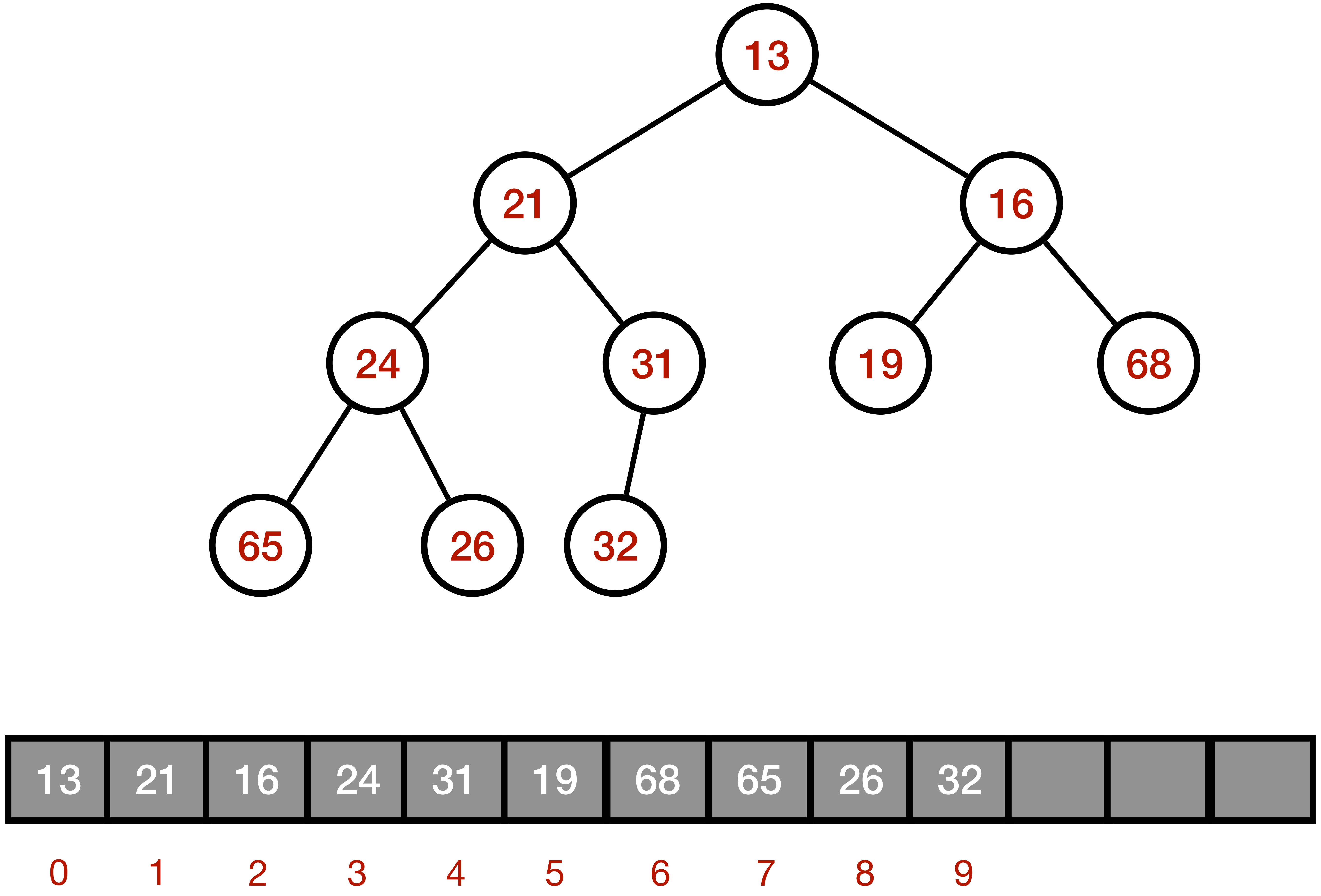


Implementing Heaps: Efficiency



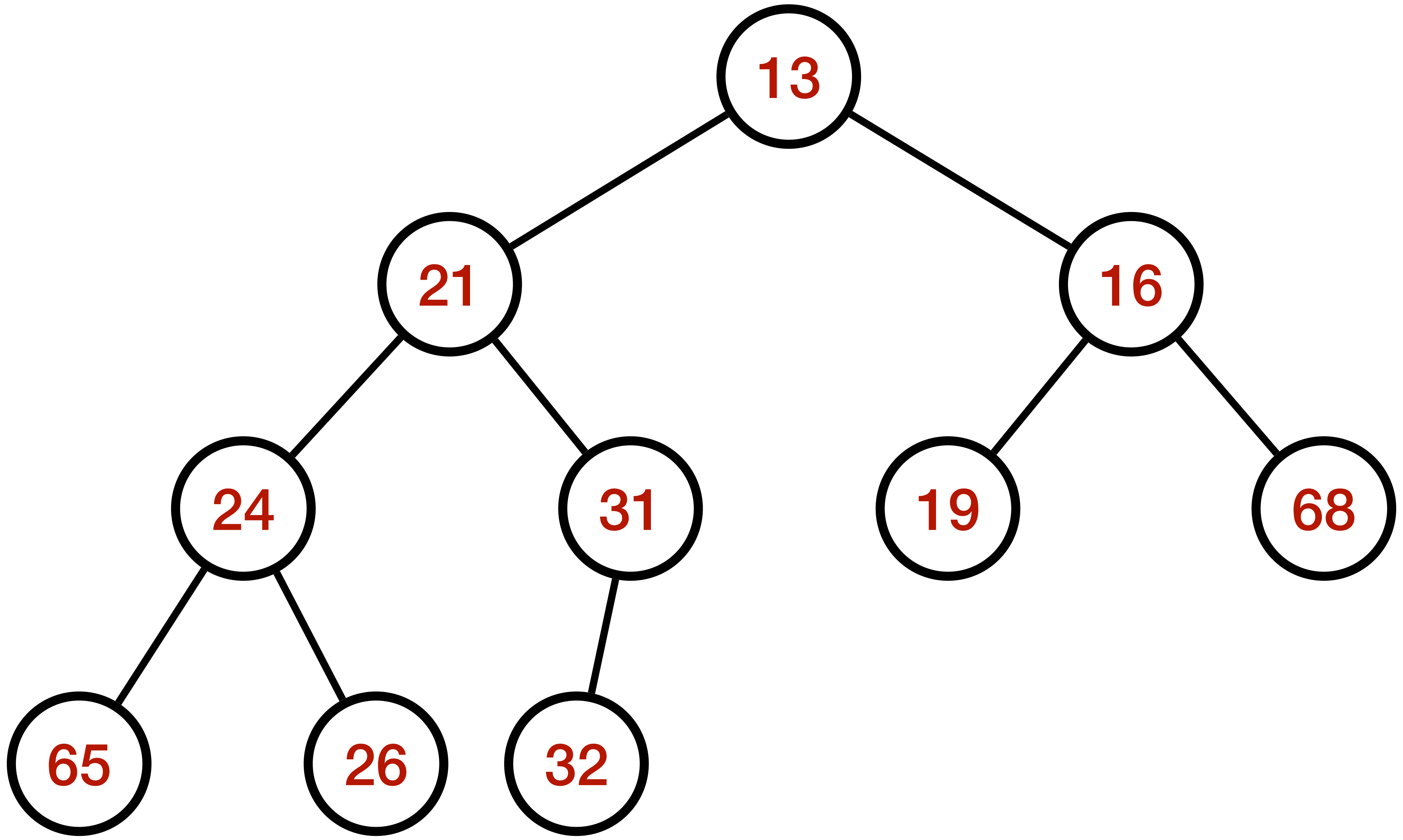
Implementing Heaps: Efficiency

- **Observation:** In binary representation, **multiplication by 2 is a left shift** and FMA instructions to multiply and add (adding 1 to the lowest bit)



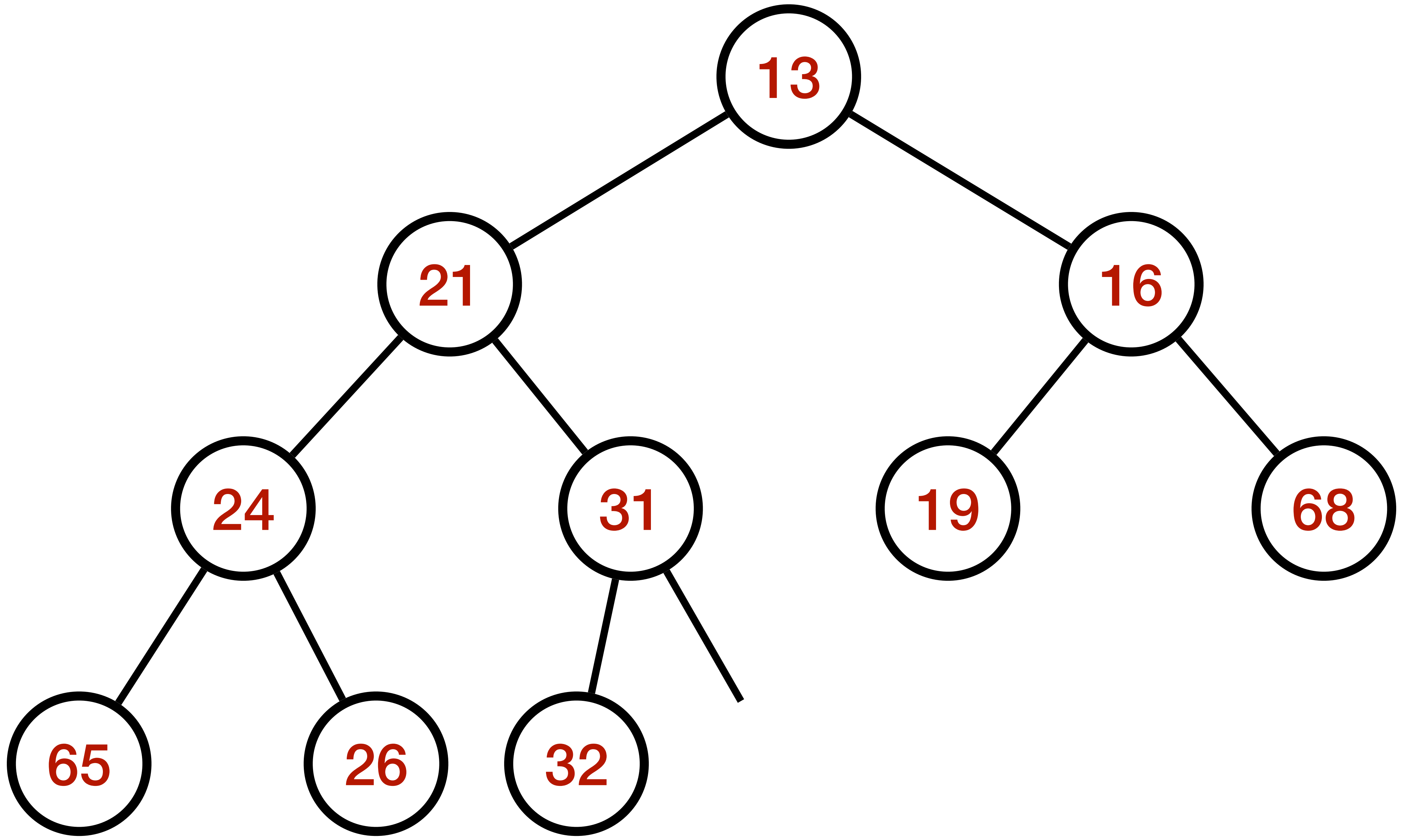
Heap Insertion

- Insert 12



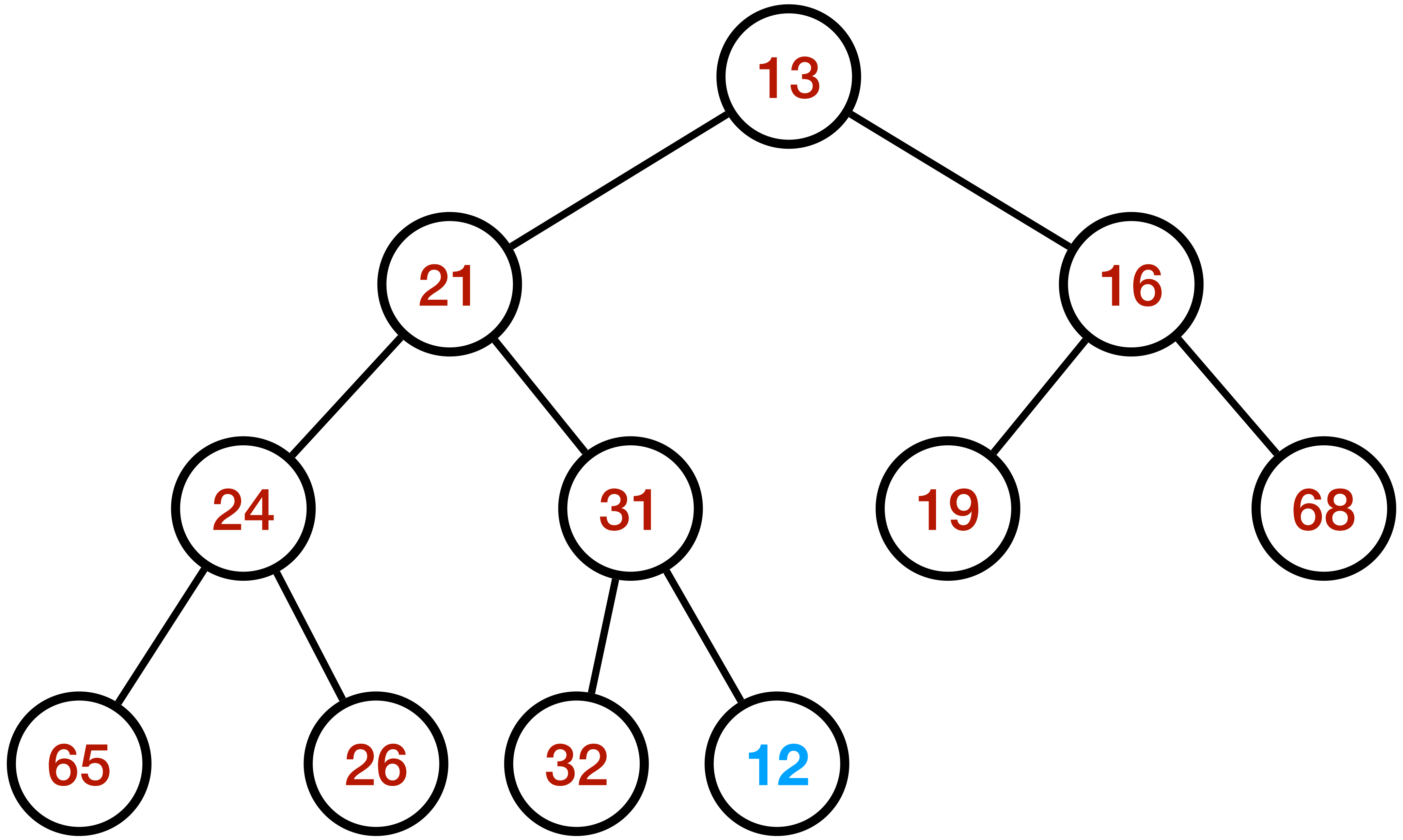
Heap Insertion

- Insert 12



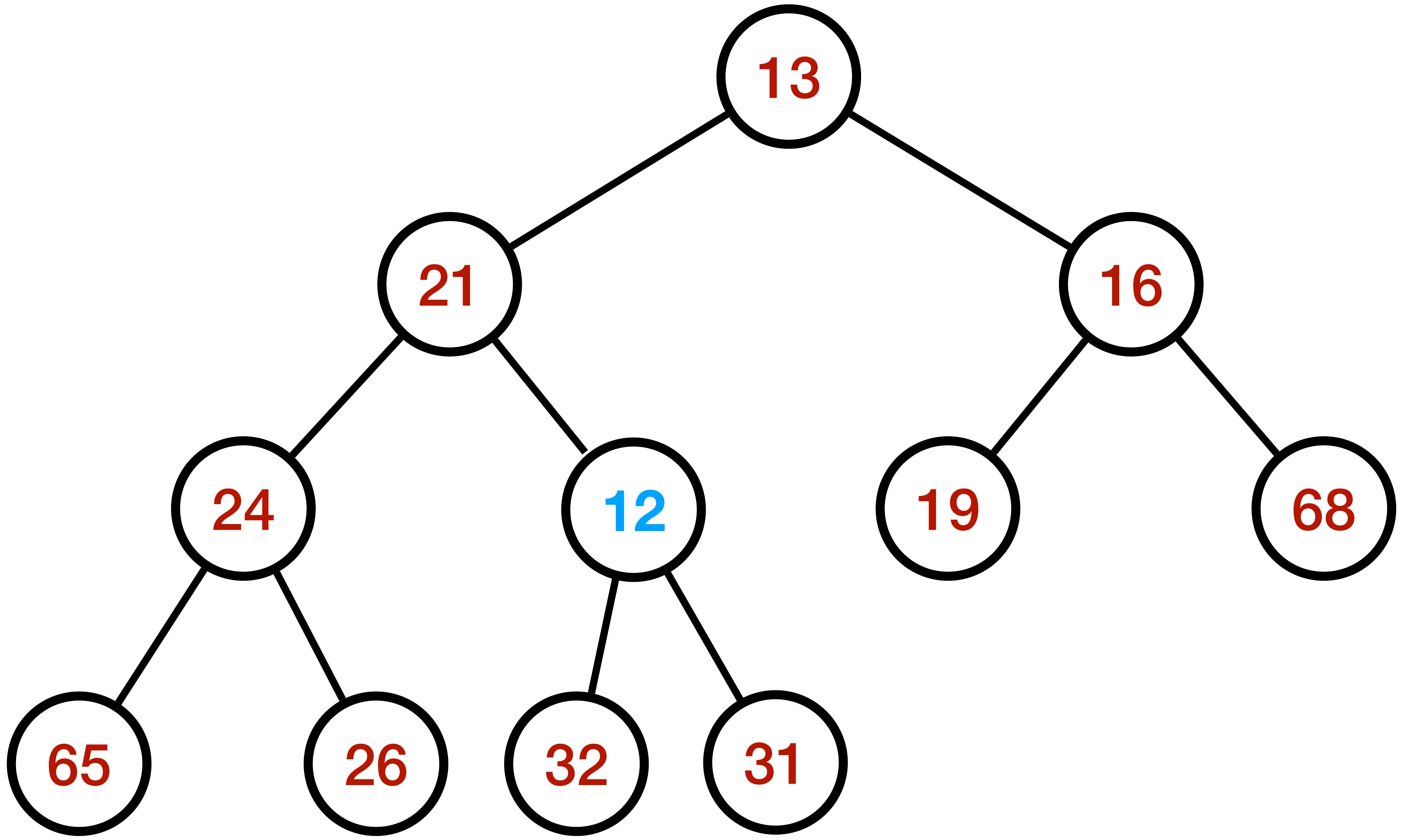
Heap Insertion

- Insert 12



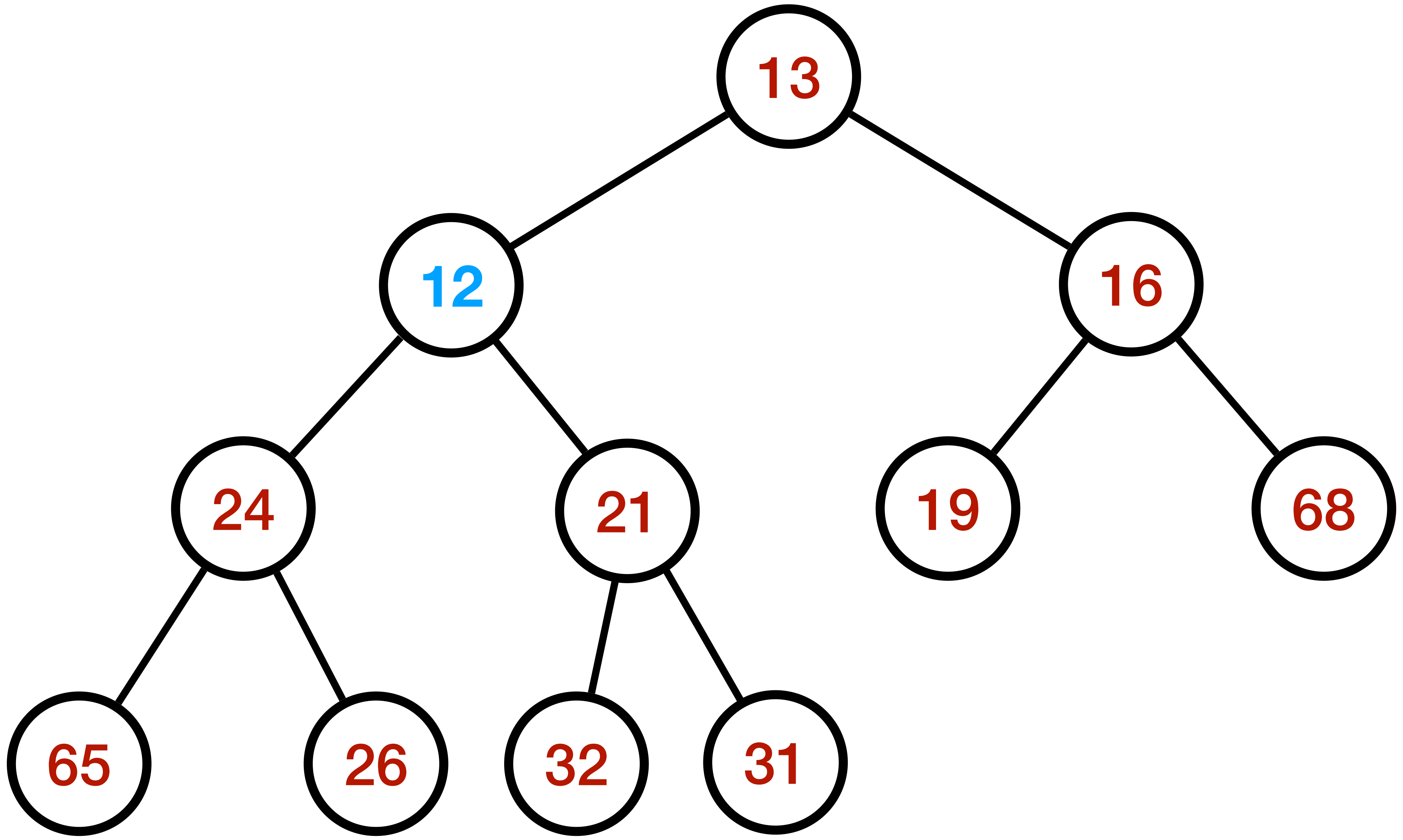
Heap Insertion

- Insert 12



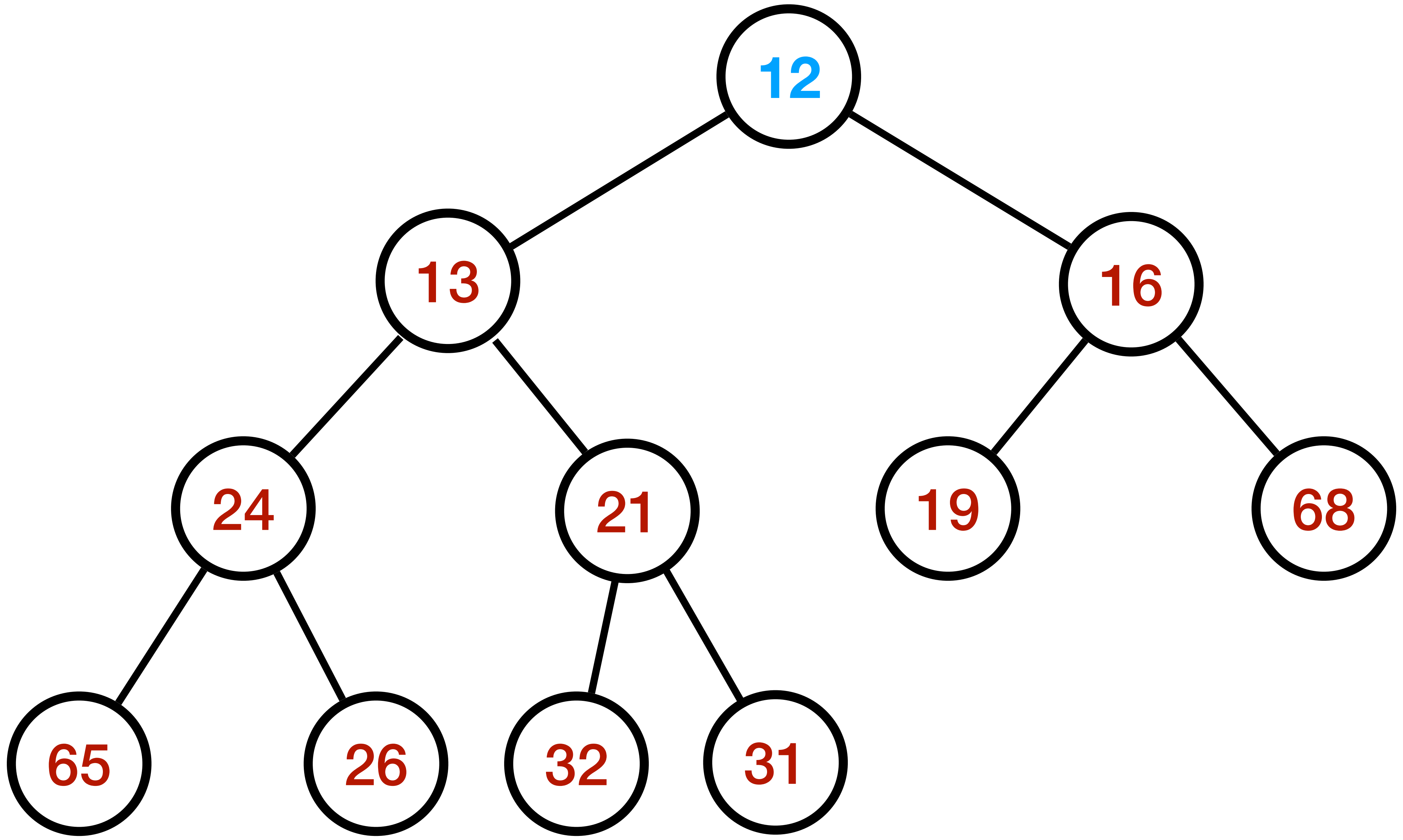
Heap Insertion

- Insert 12



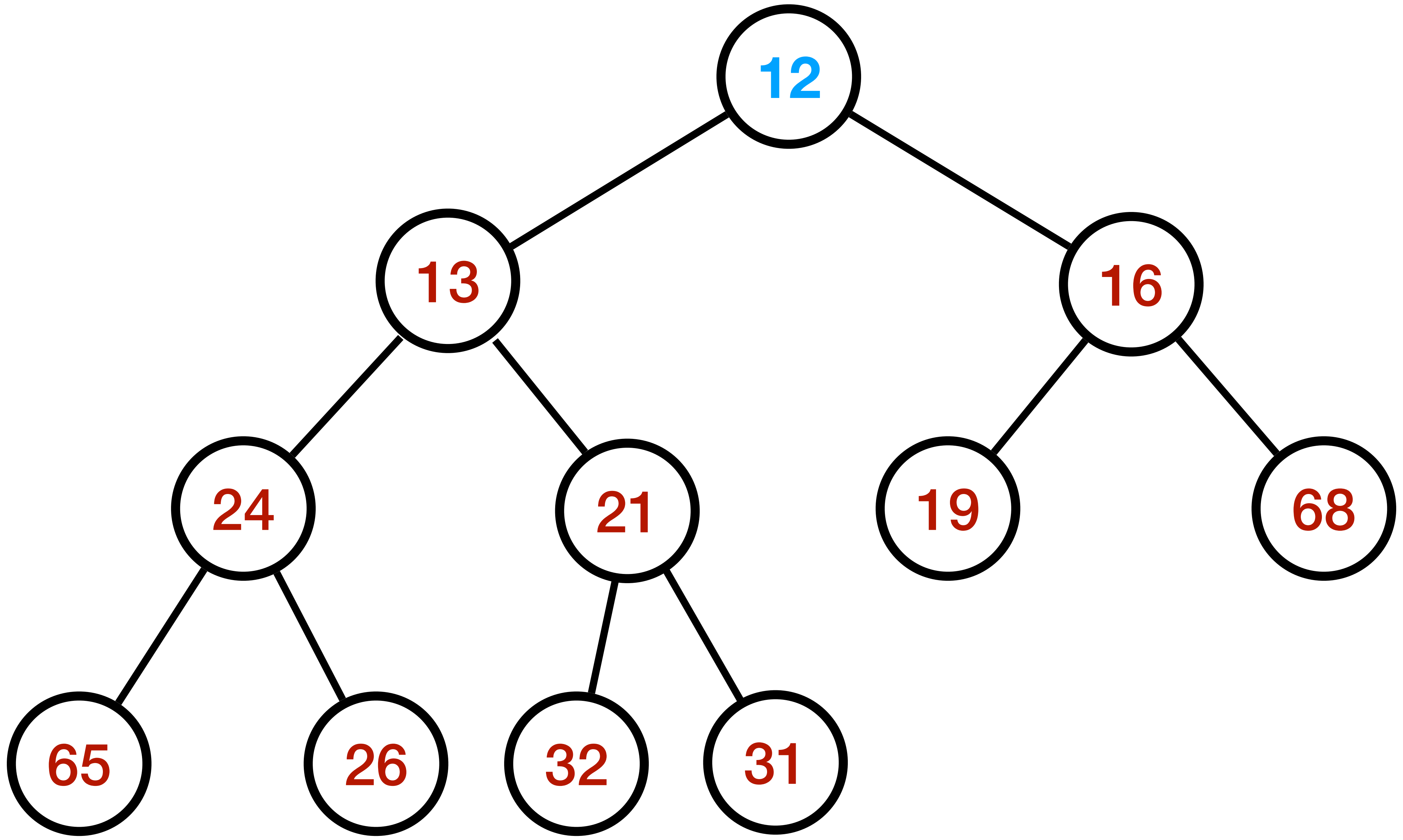
Heap Insertion

- Insert 12



Heap Insertion

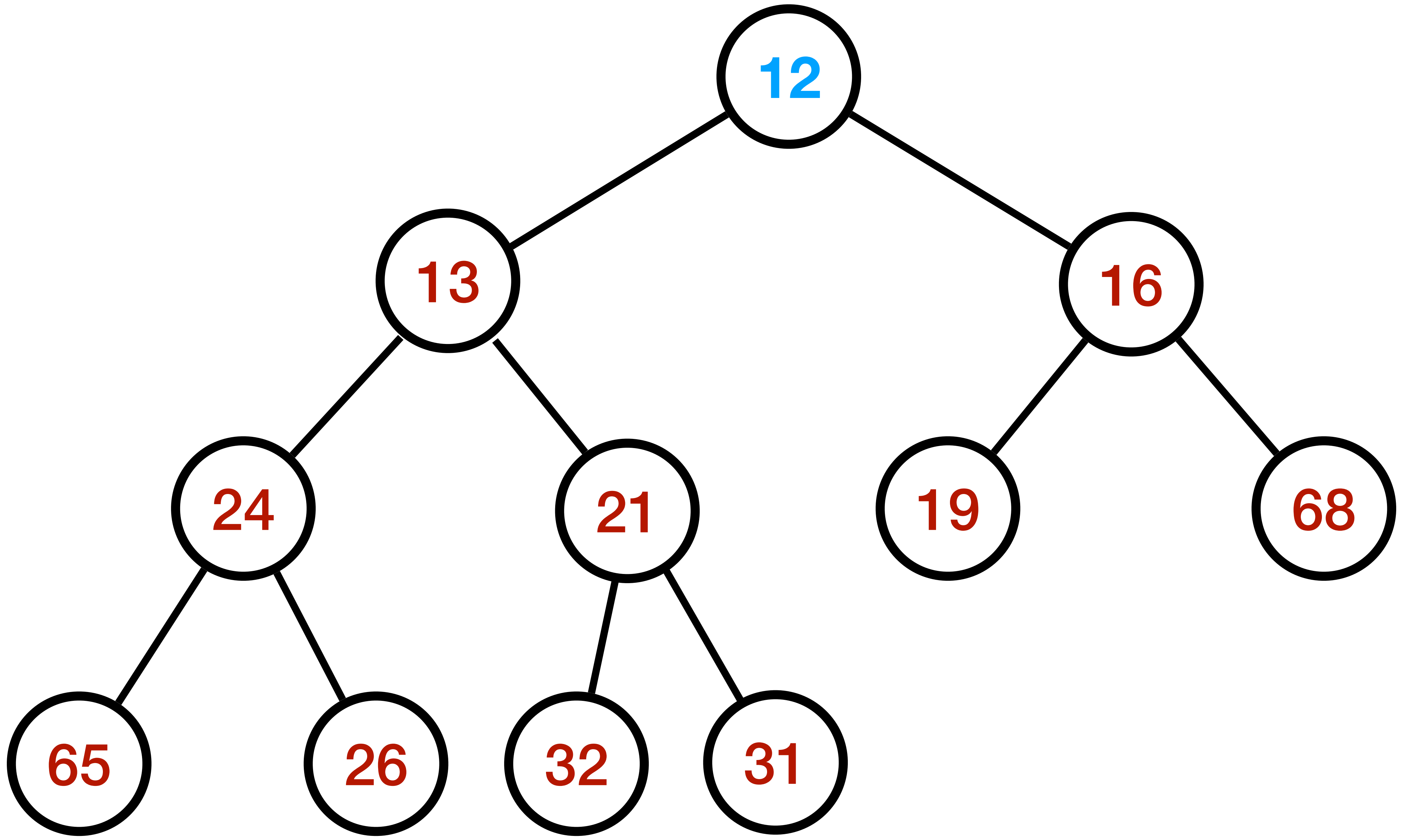
- Insert 12
- The process of restoring order is called **Heapify**



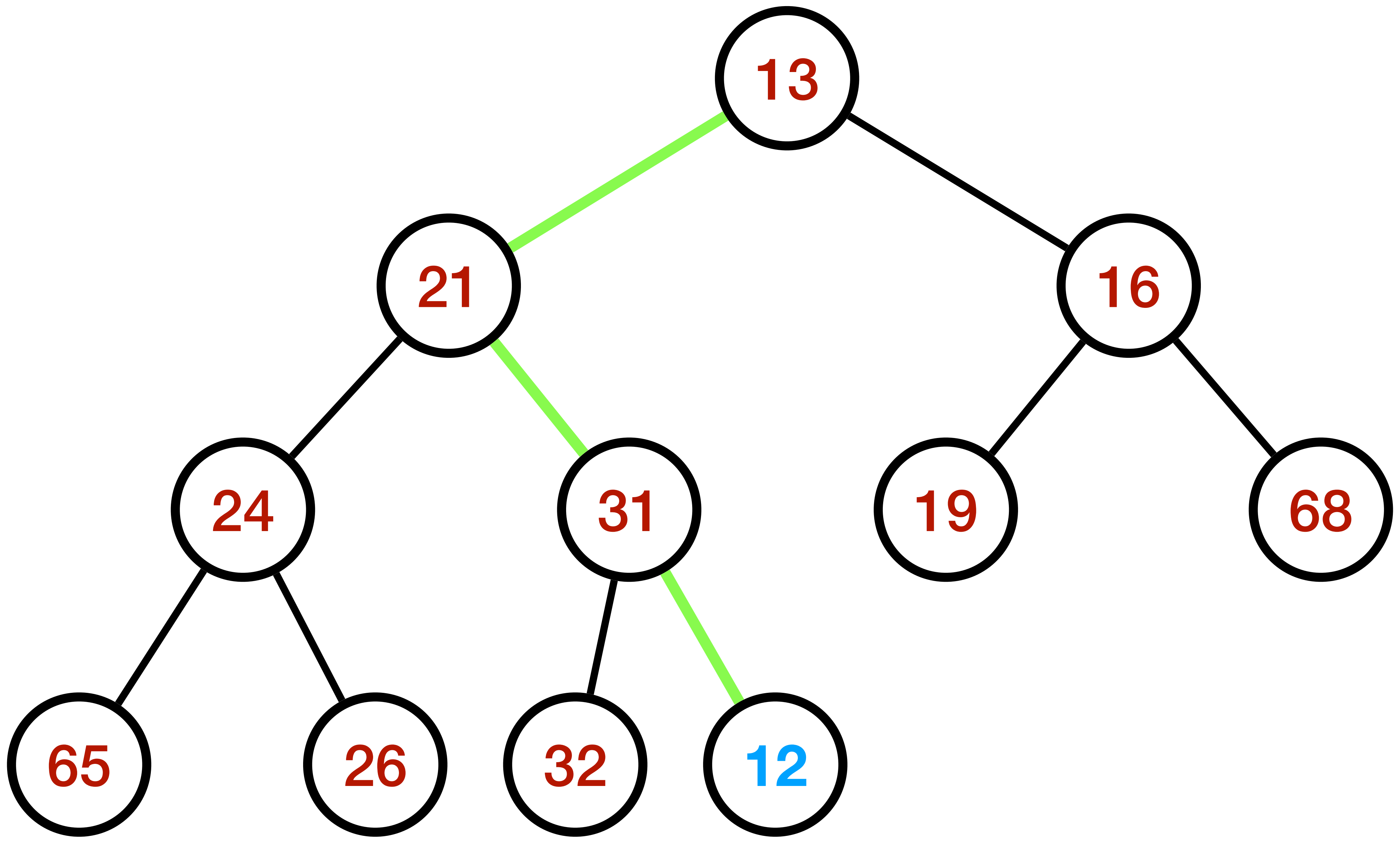
Heap Insertion

- Insert 12
- The process of restoring order is called **Heapify**

```
void insert(int val) {
  heap.push_back(val);
  heapifyUp(heap.size() - 1);
}

void heapifyUp(int index) {
  if (index == 0) return;

  int parentIndex = getParentIndex(index);

  if (heap[parentIndex] > heap[index]) {
    swap(heap[parentIndex], heap[index]);
    heapifyUp(parentIndex);
  }
}
```

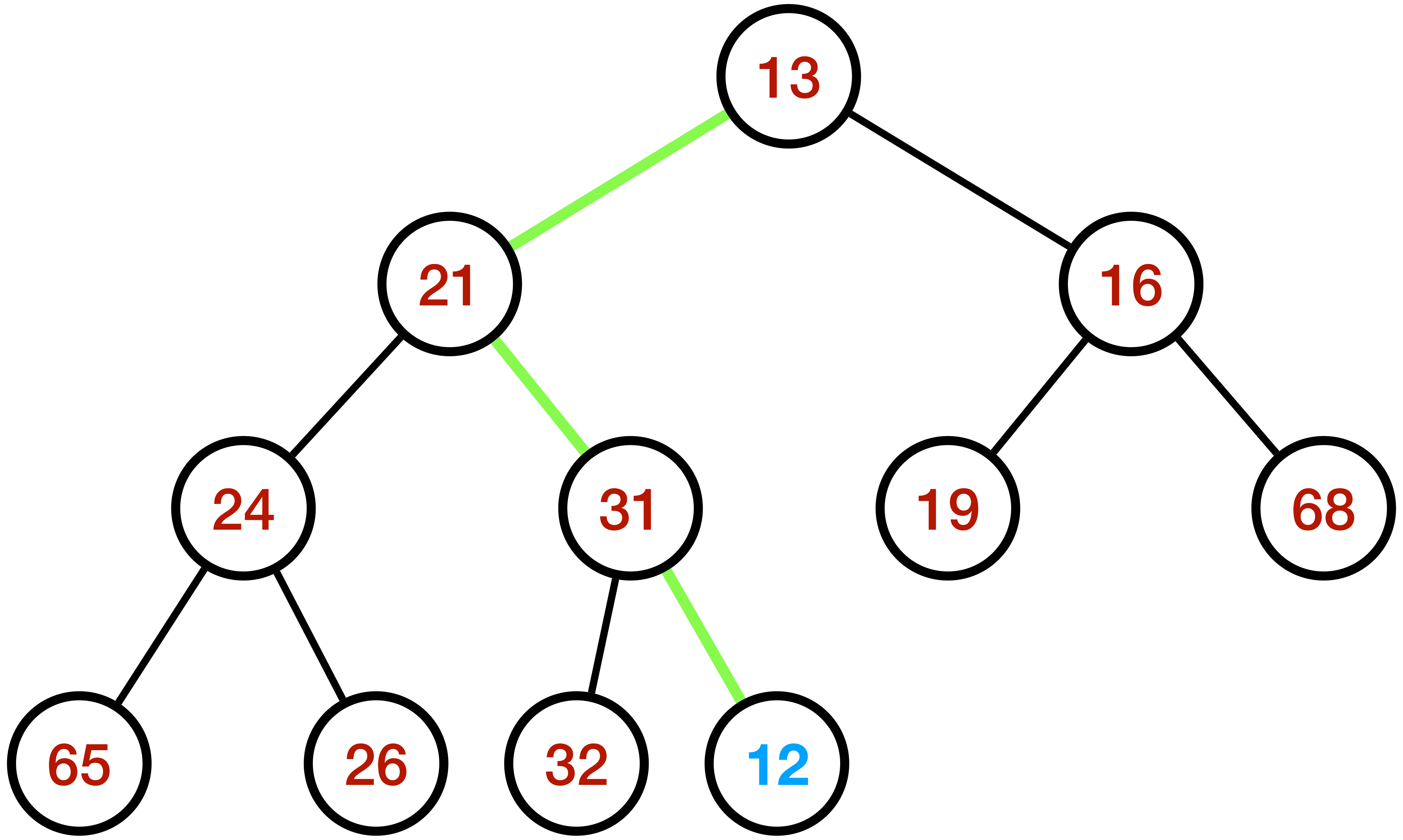


Why is Insertion Correct?



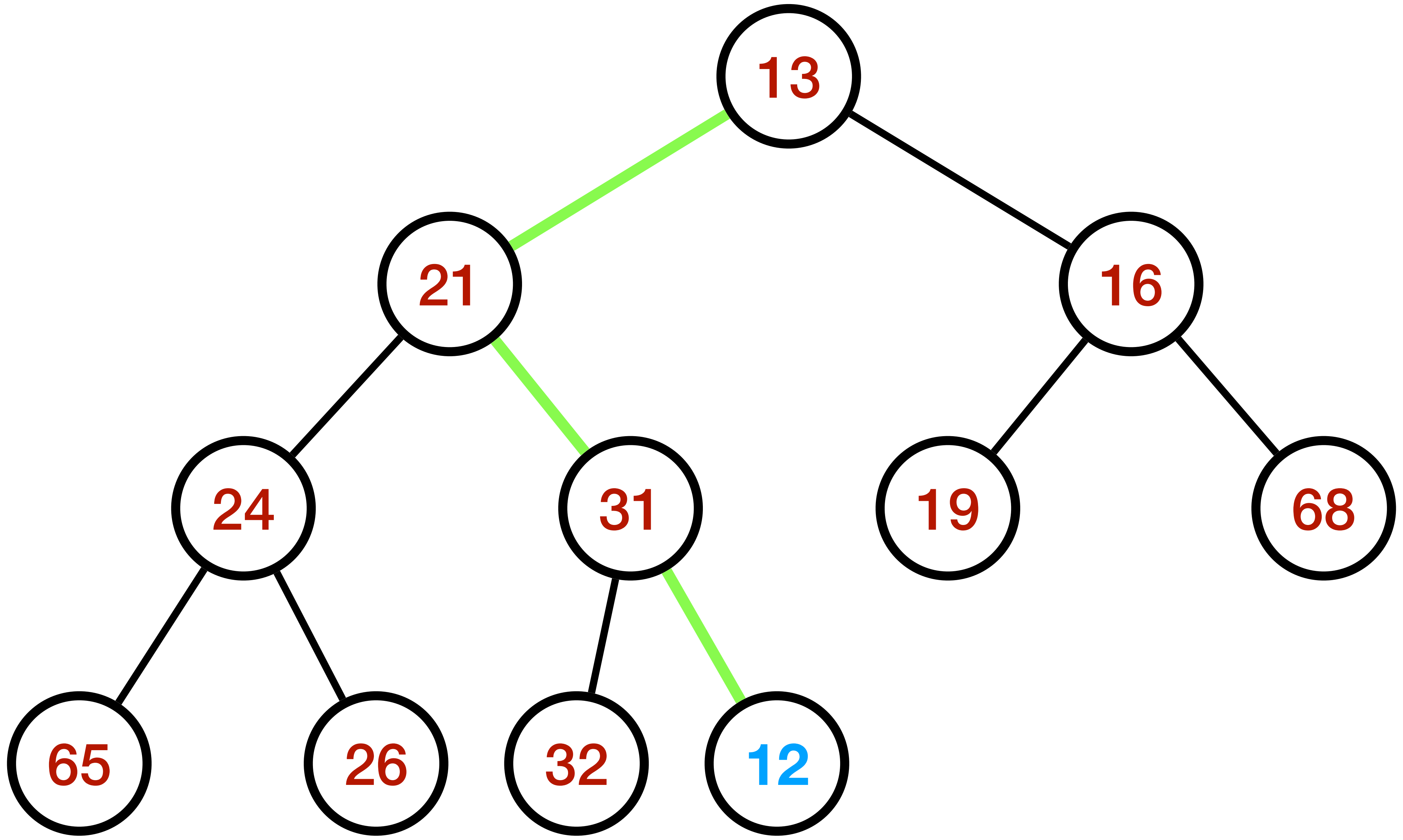
Why is Insertion Correct?

- The only nodes whose contents change are the ones on the path



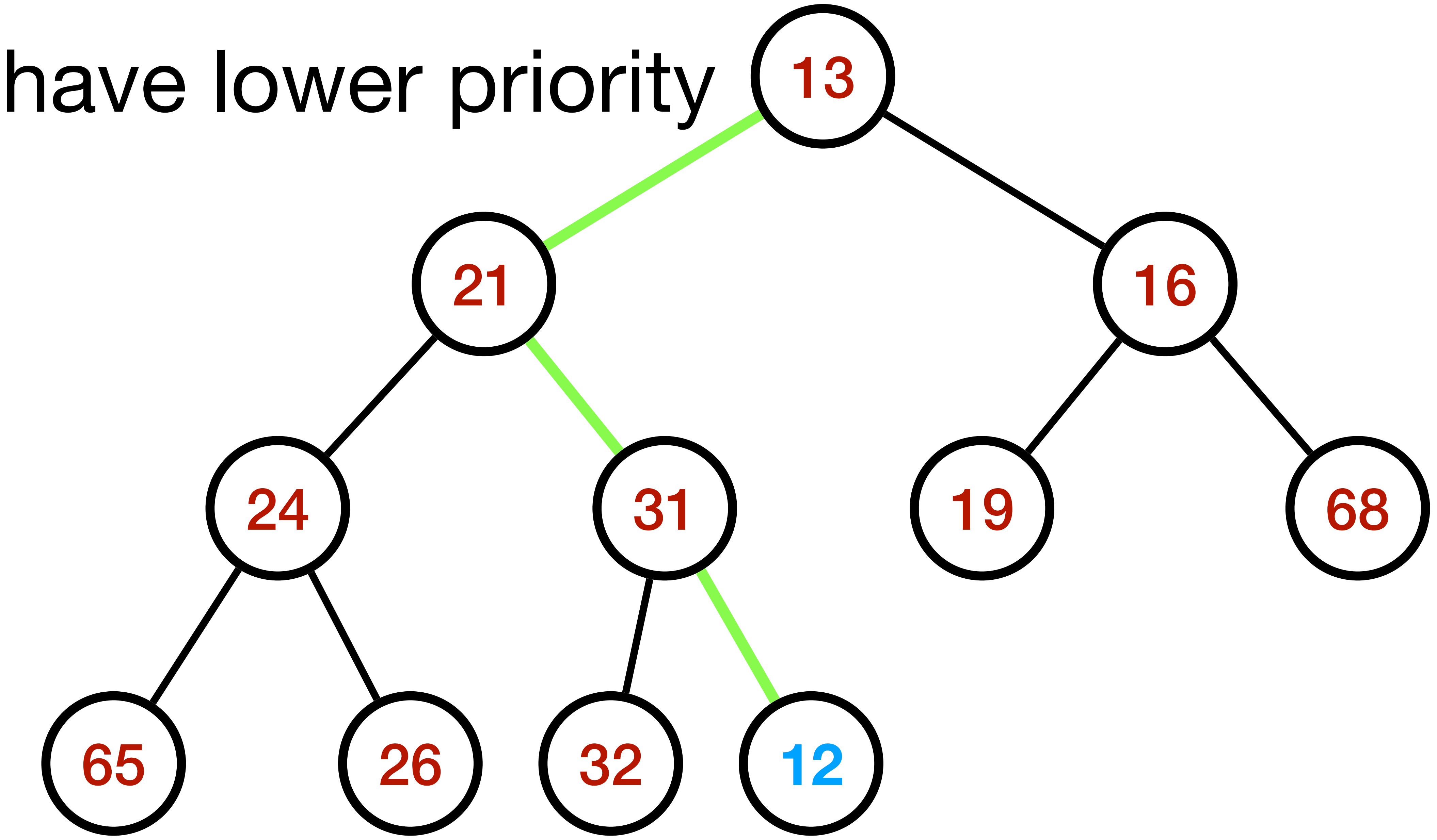
Why is Insertion Correct?

- The only nodes whose contents change are the ones on the path
- Heap property may violate only for children of these nodes



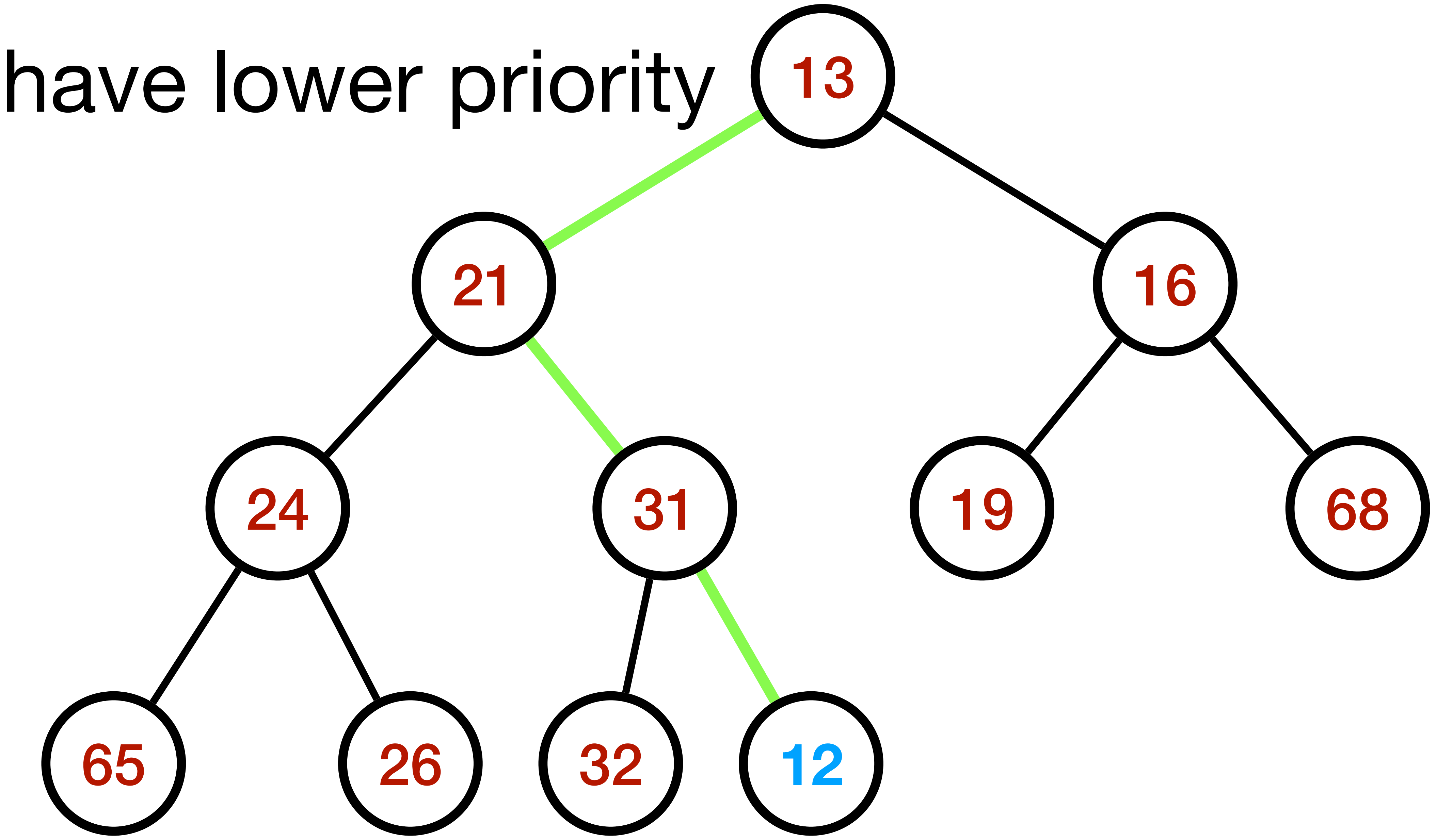
Why is Insertion Correct?

- The only nodes whose contents change are the ones on the path
- Heap property may violate only for children of these nodes
- But the new contents of these nodes only have lower priority



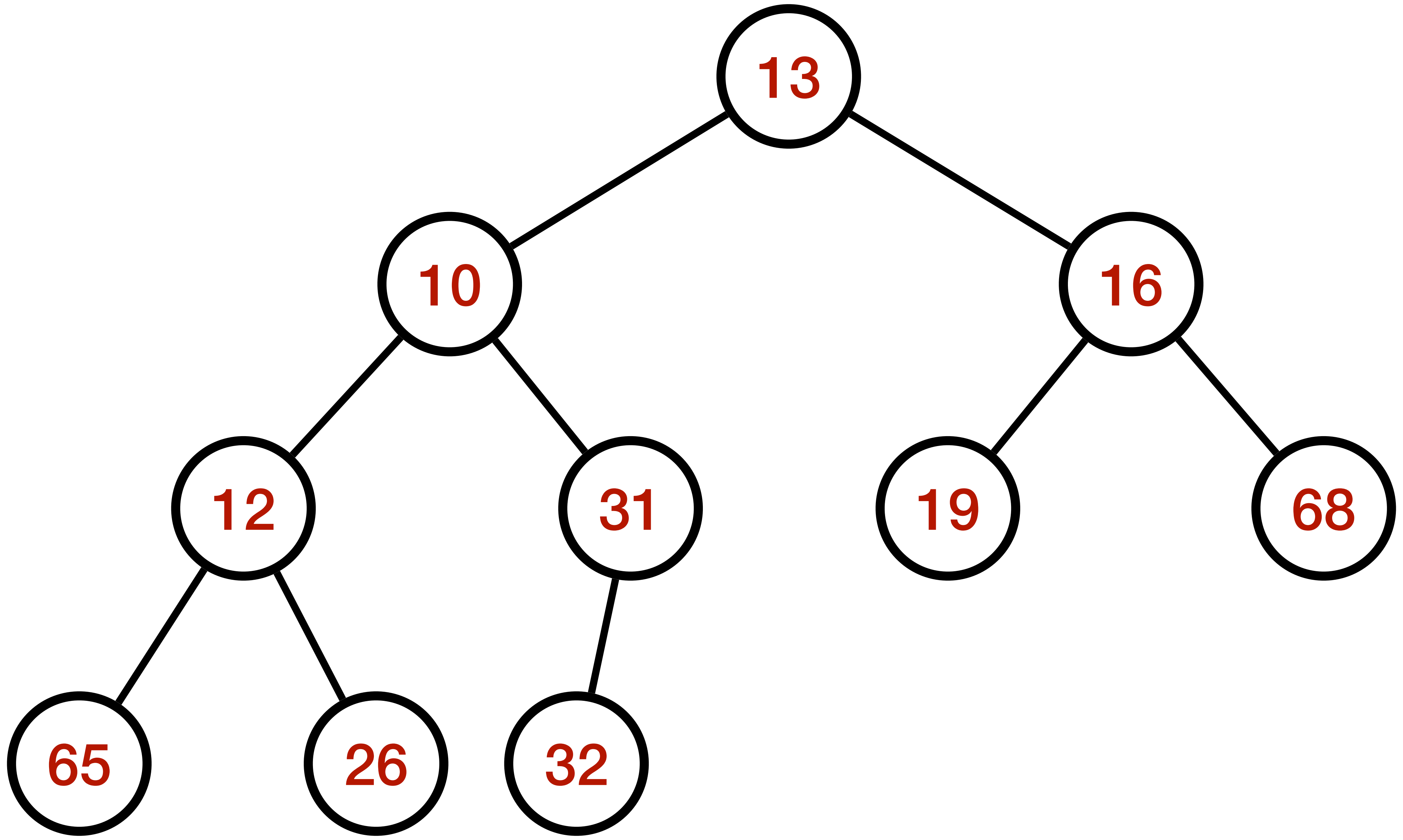
Why is Insertion Correct?

- The only nodes whose contents change are the ones on the path
- Heap property may violate only for children of these nodes
- But the new contents of these nodes only have lower priority
- Thus, it is correct!



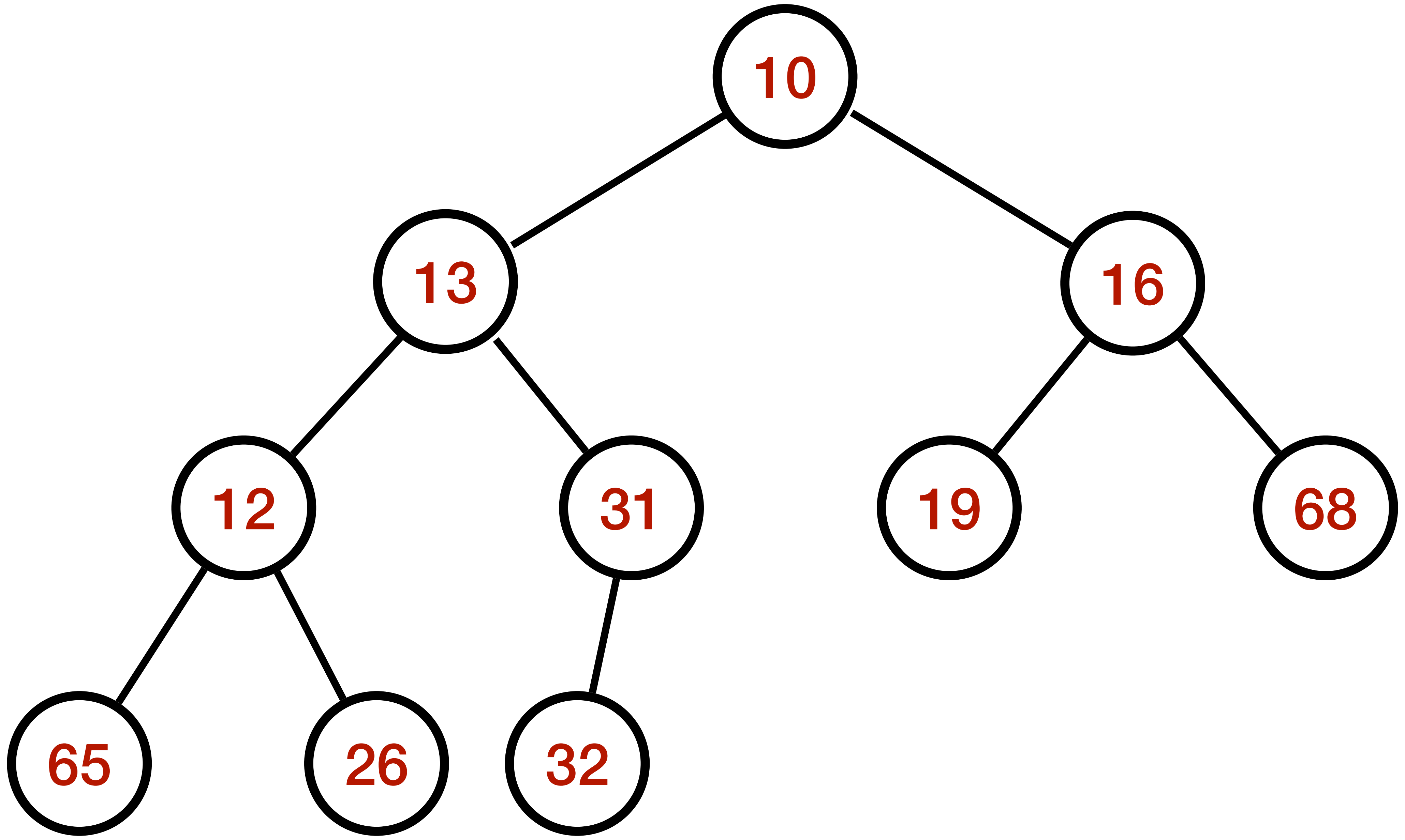
Another View of Heapify

- Heap order property violated at index 0
- The subtrees rooted at index 1 and 2 are valid heaps
  - This is an important point — Heapify would work only when this observation holds
- heapifyDown(0)
- **ToDo — Prove the correctness of HeapifyDown**



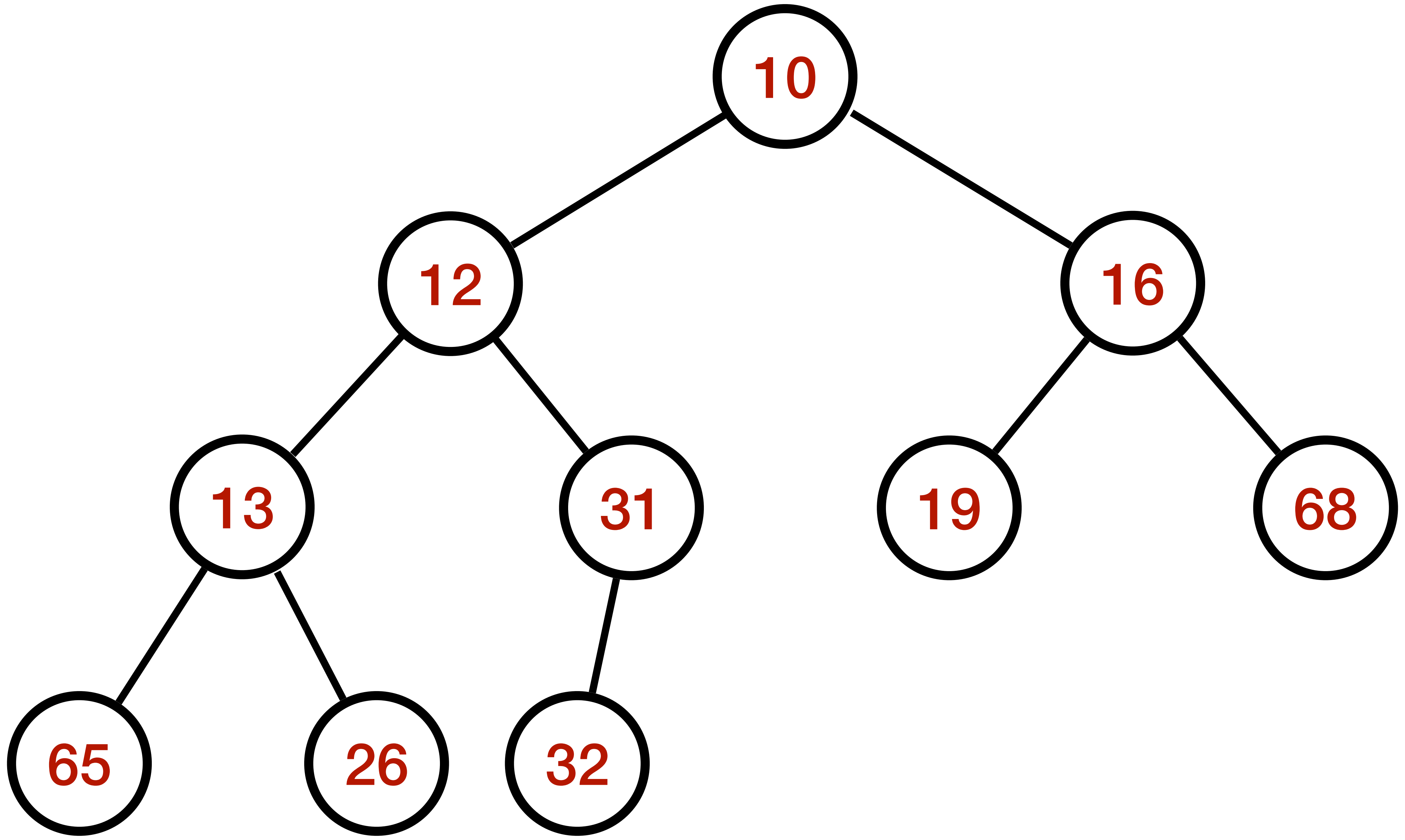
Another View of Heapify

- Heap order property violated at index 0
- The subtrees rooted at index 1 and 2 are valid heaps
    - This is an important point — Heapify would work only when this observation holds
- heapifyDown(0)
- **ToDo — Prove the correctness of HeapifyDown**



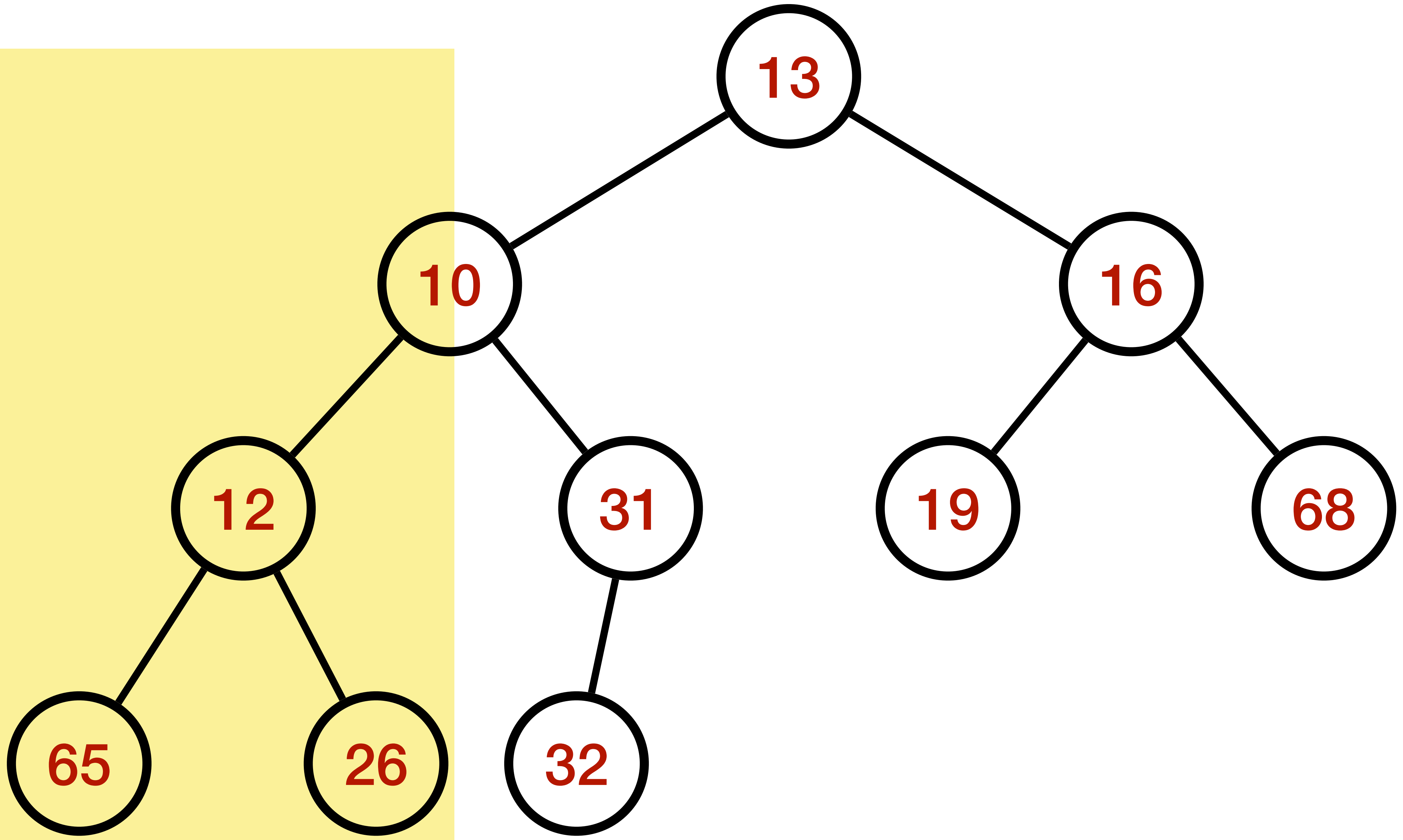
Another View of Heapify

- Heap order property violated at index 0
- The subtrees rooted at index 1 and 2 are valid heaps
  - This is an important point — Heapify would work only when this observation holds
- heapifyDown(0)
- **ToDo — Prove the correctness of HeapifyDown**



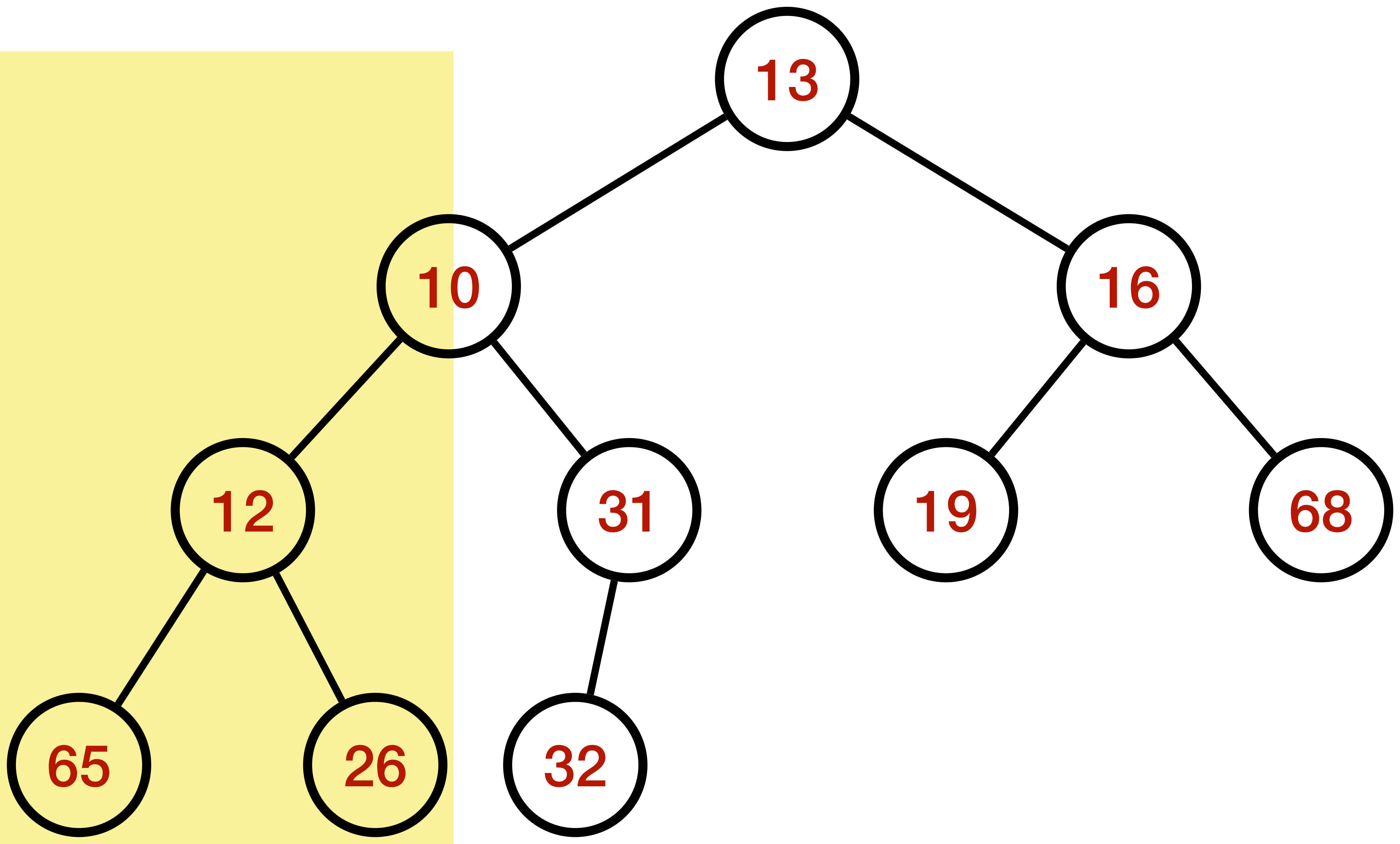
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



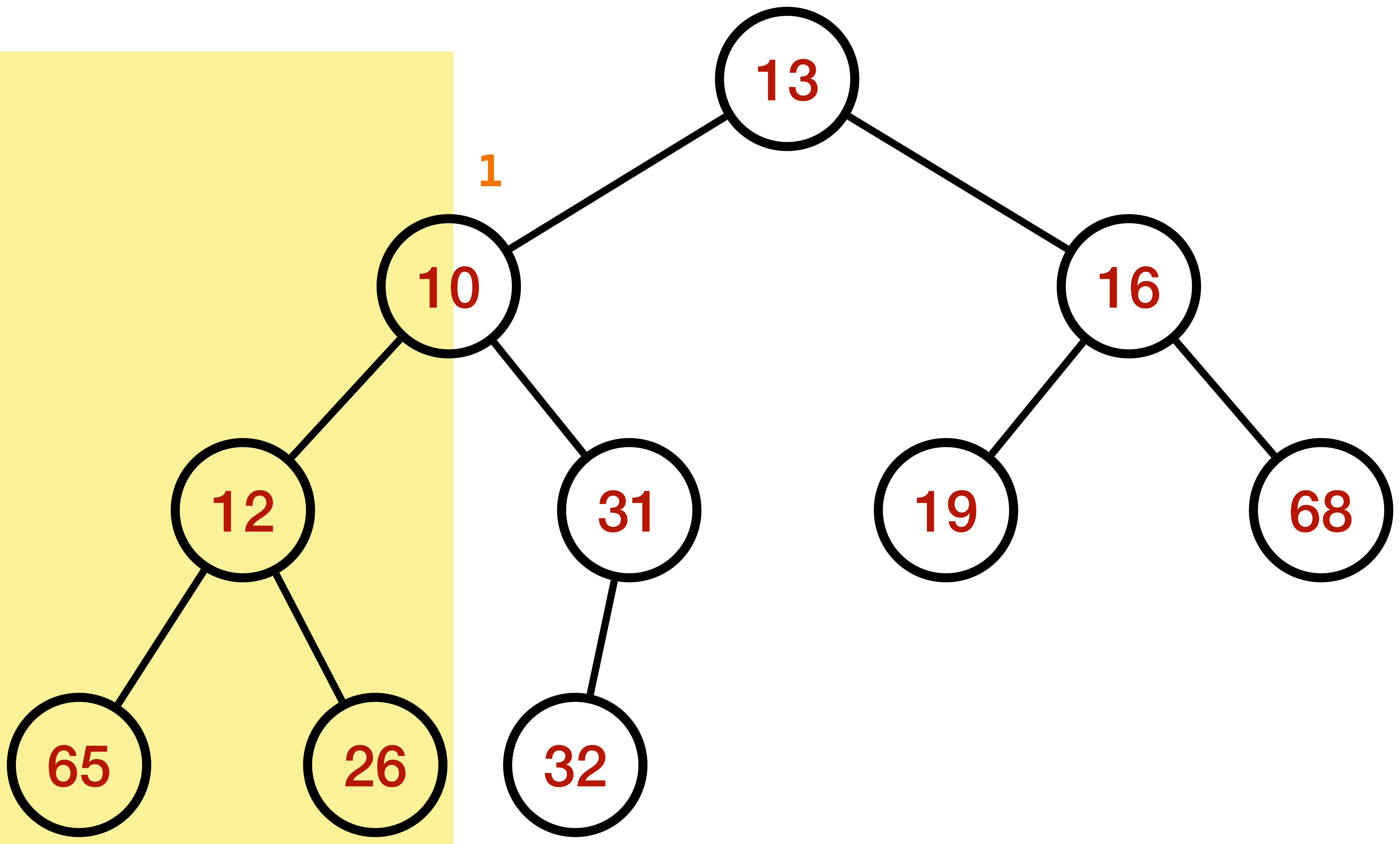
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



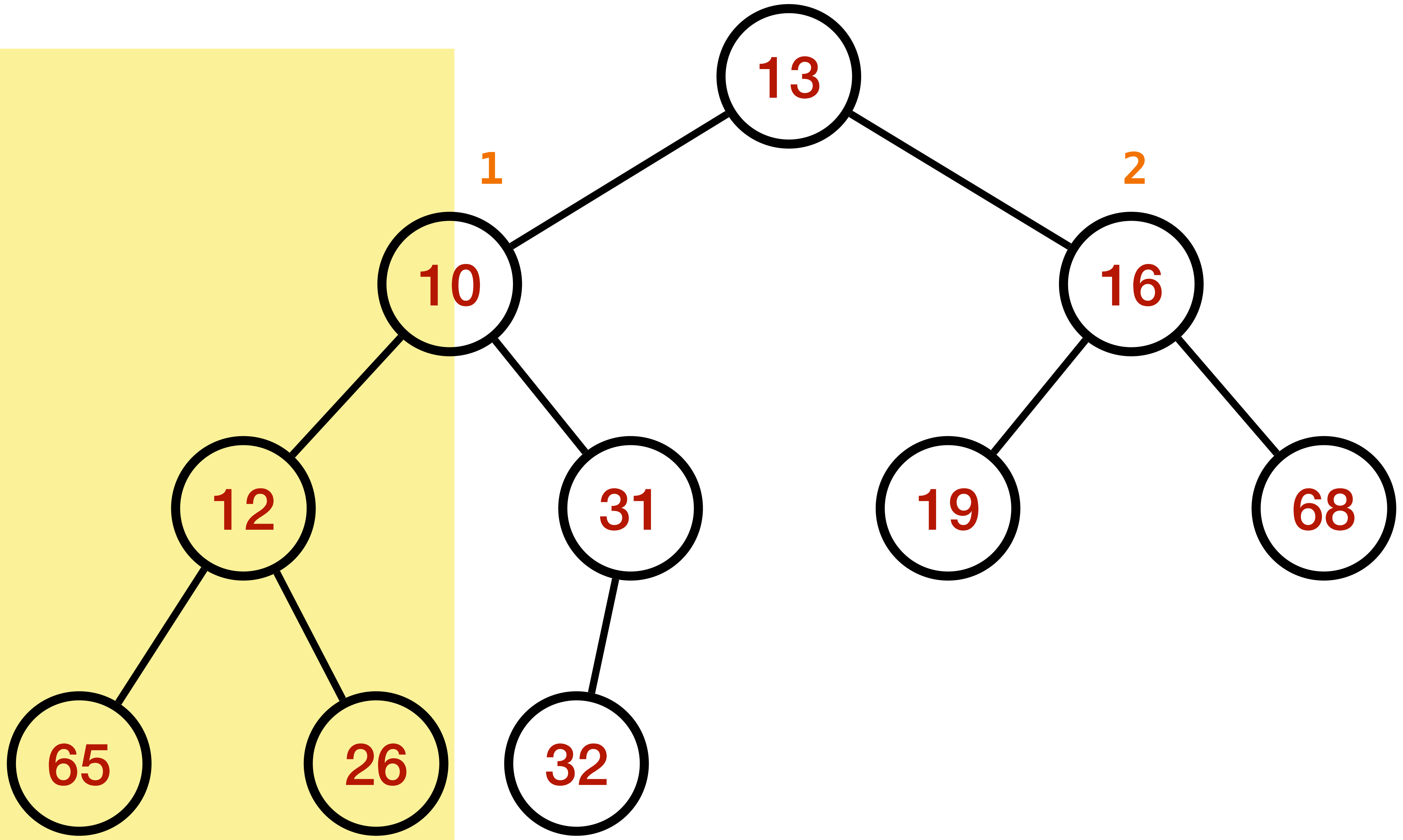
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



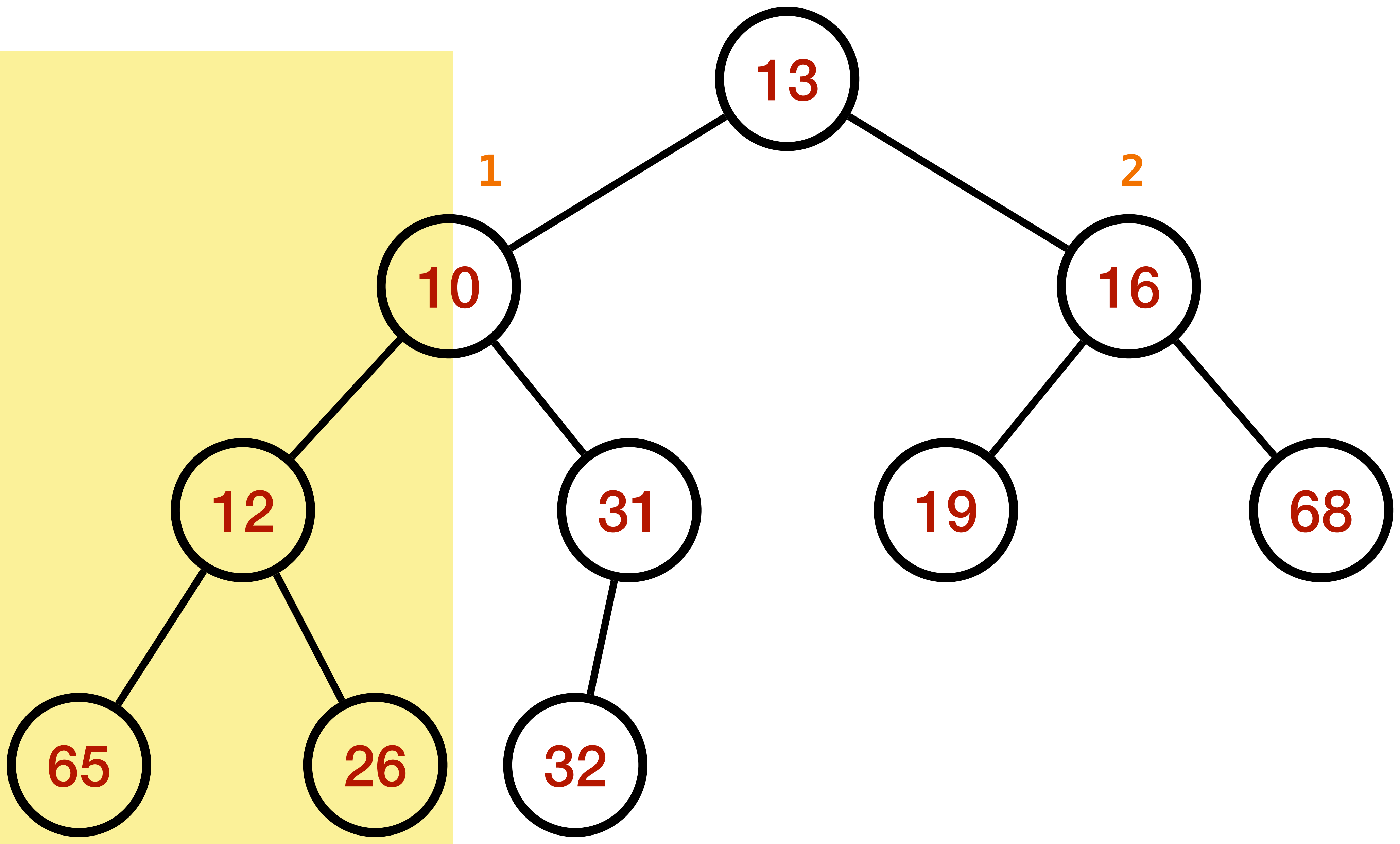
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



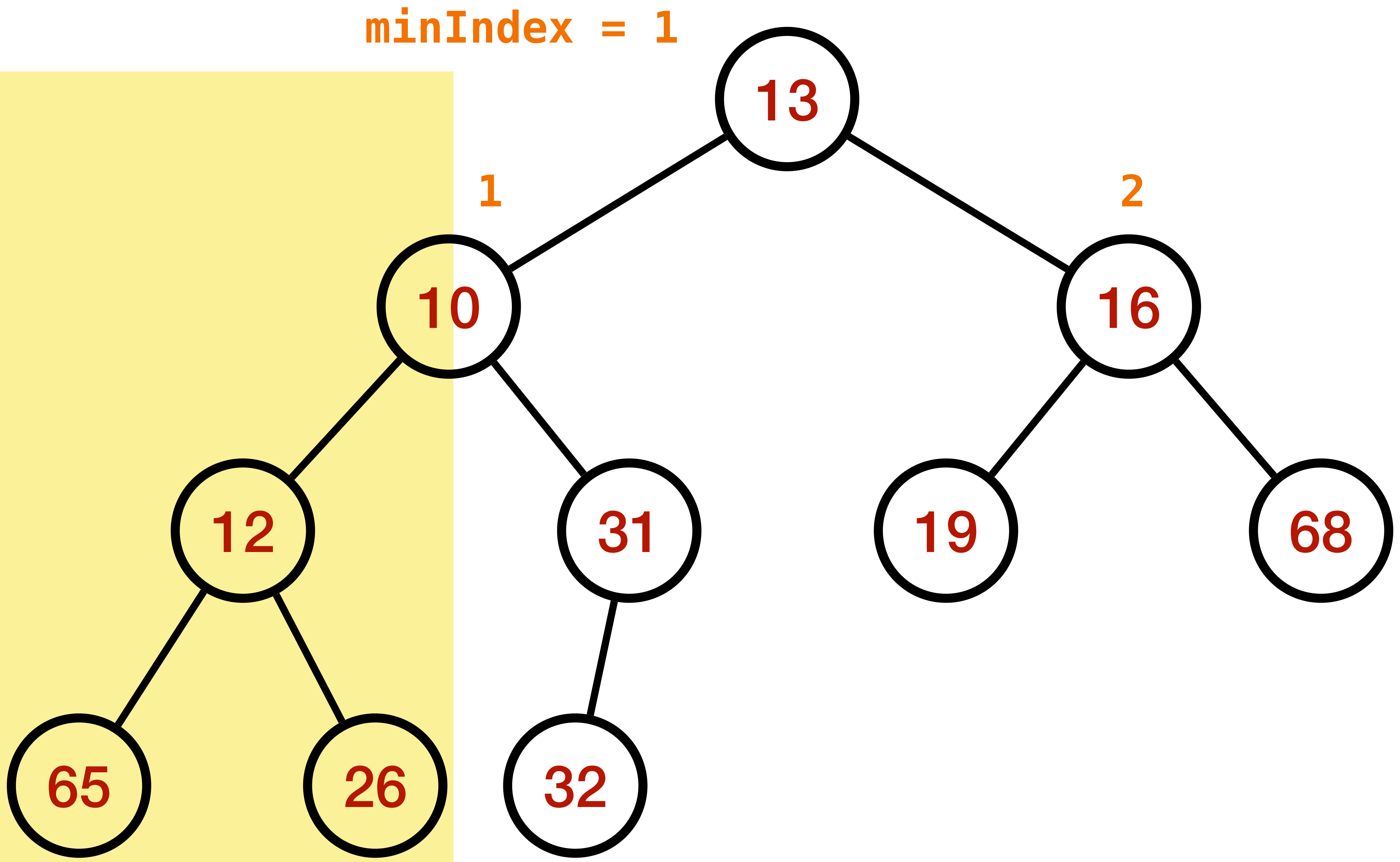
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



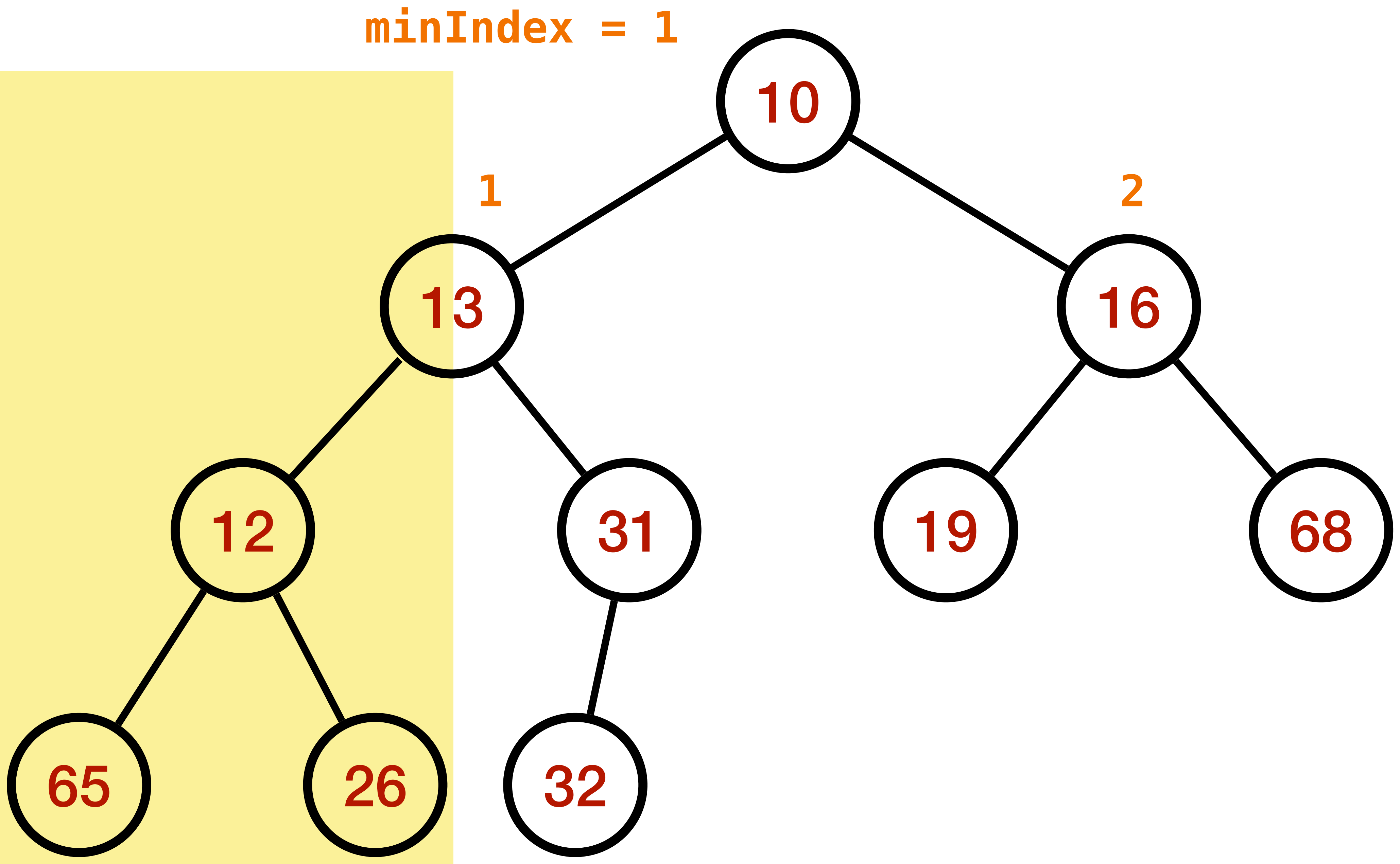
HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```



HeapifyDown

```
void heapifyDown(int index) {
  int leftChild = getLeftChildIndex(index);
  int rightChild = getRightChildIndex(index);

  if (leftChild >= heap.size()) return; // No children

  int minIndex = index;

  if (heap[minIndex] > heap[leftChild]) {
    minIndex = leftChild;
  }

  if (rightChild < heap.size() && heap[minIndex] > heap[rightChild]) {
    minIndex = rightChild;
  }

  if (minIndex != index) {
    swap(heap[minIndex], heap[index]);
    heapifyDown(minIndex);
  }
}
```

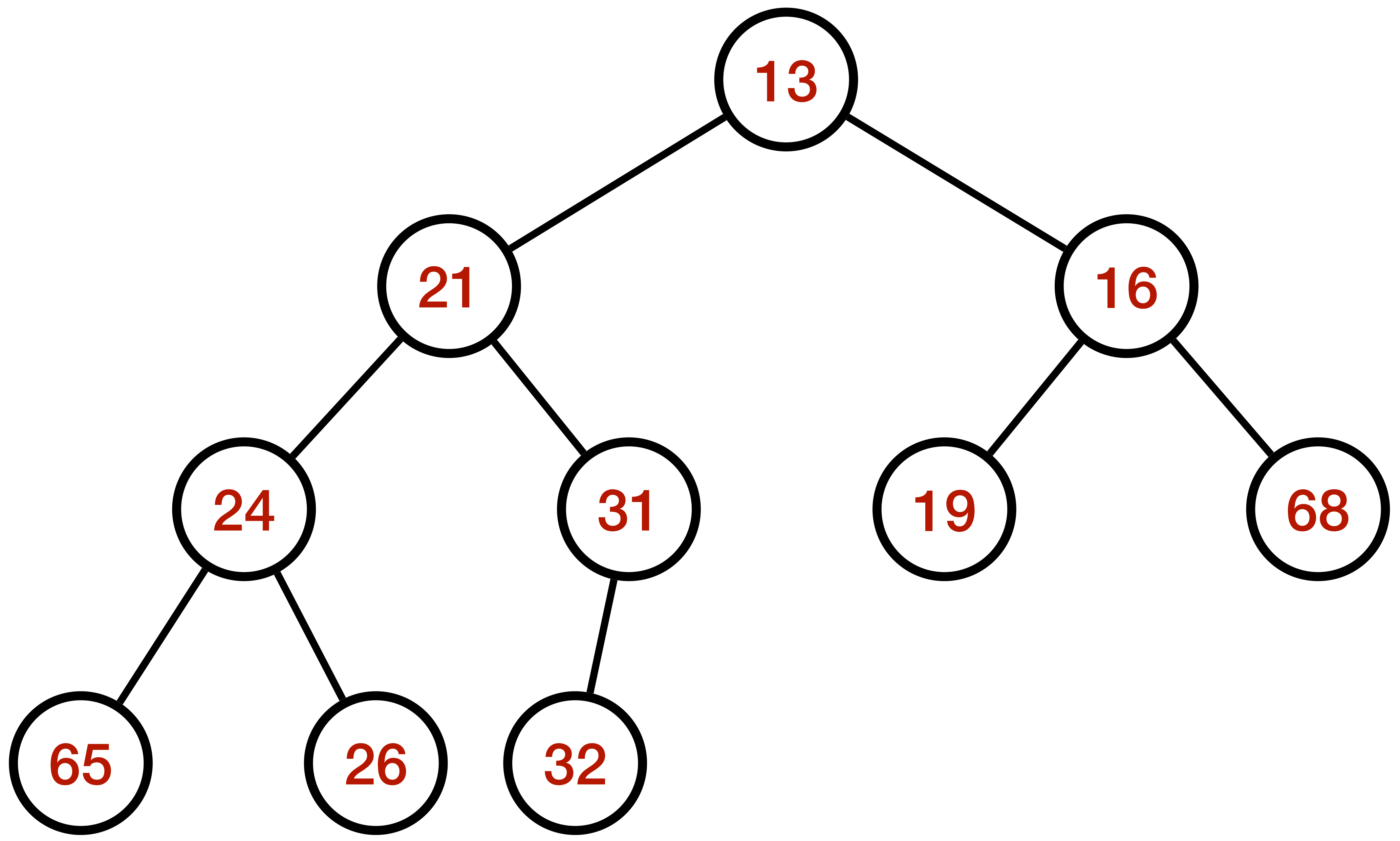


DeleteMin Operation

- Remember that the minimum element is at the root of the heap
  - We can delete this and move one of its children to fill the space!
    - Empty location moves down the tree
    - Resulting tree may not be **left-filled**

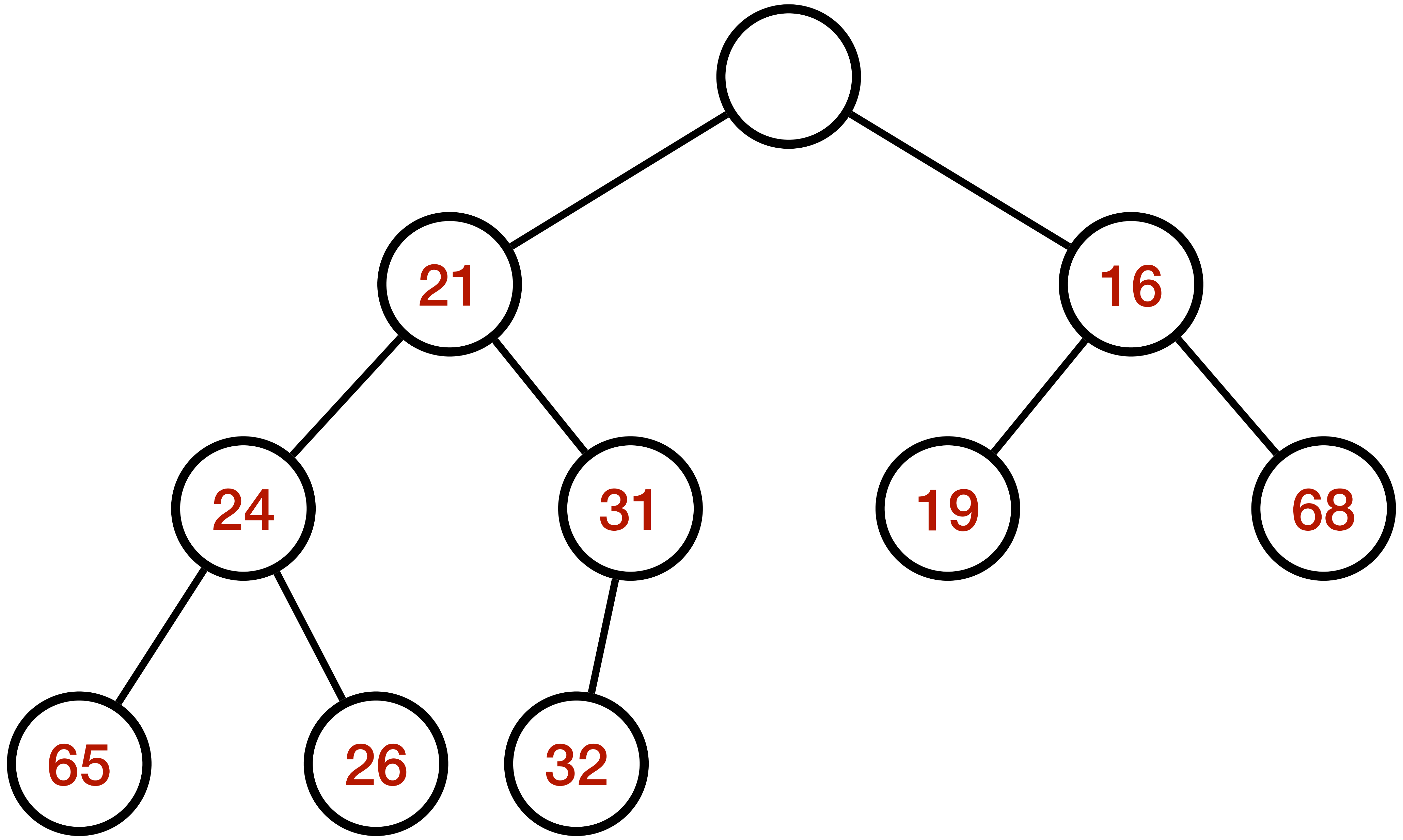
DeleteMin Operation: Attempt 1

- Delete 13
- 16 moves up
- 19 moves up
- Not left-filled



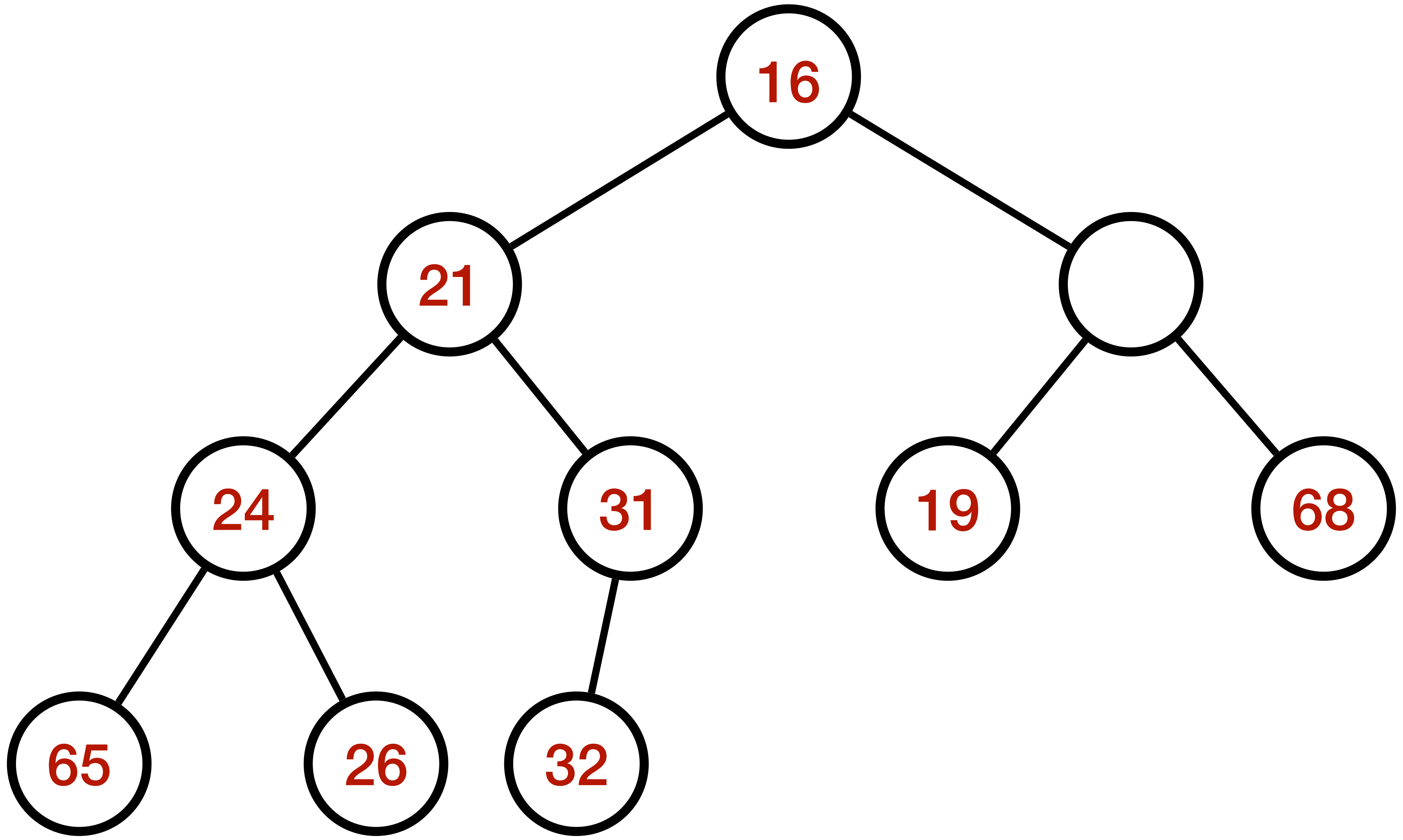
DeleteMin Operation: Attempt 1

- Delete 13
- 16 moves up
- 19 moves up
- Not left-filled



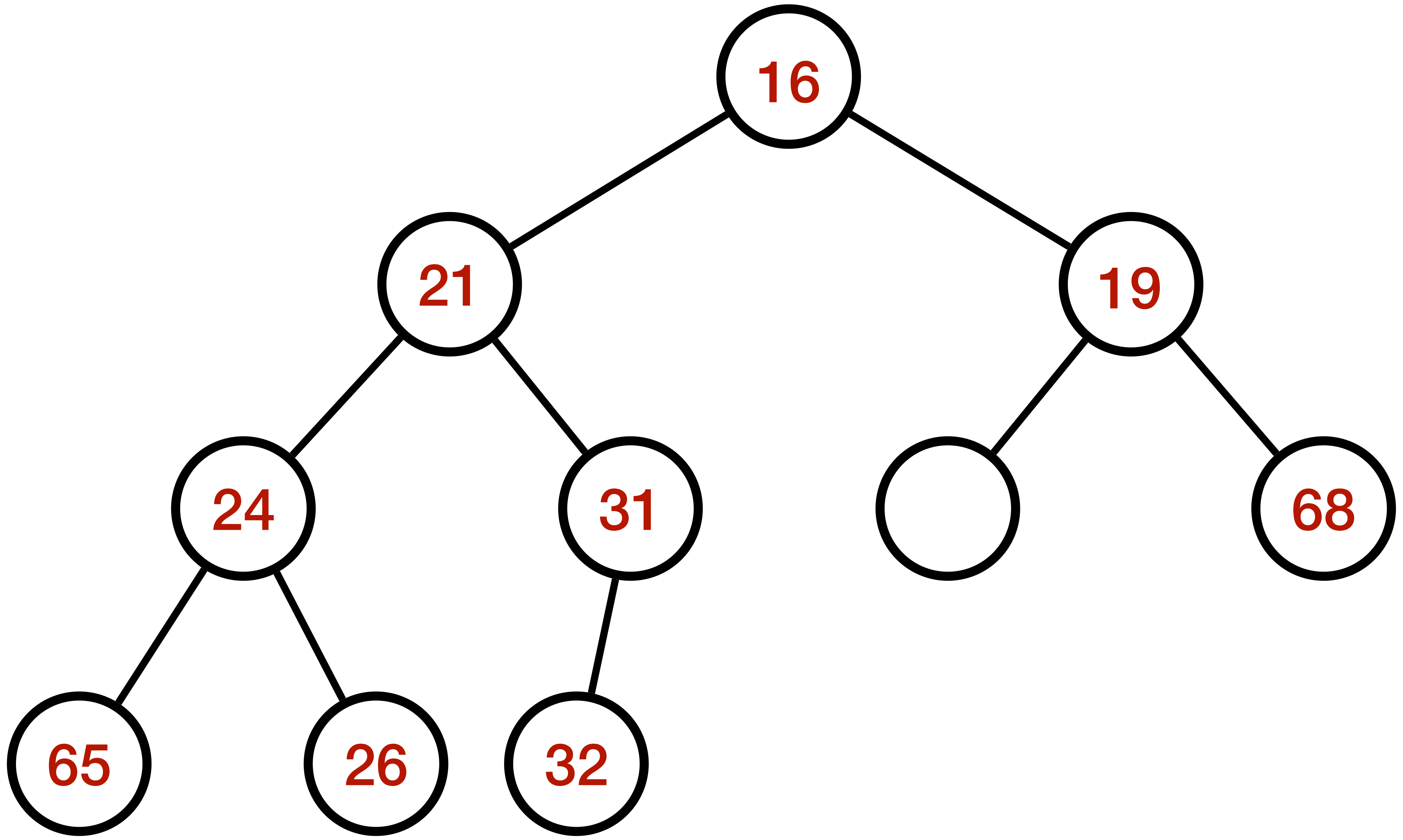
DeleteMin Operation: Attempt 1

- Delete 13
- 16 moves up
- 19 moves up
- Not left-filled

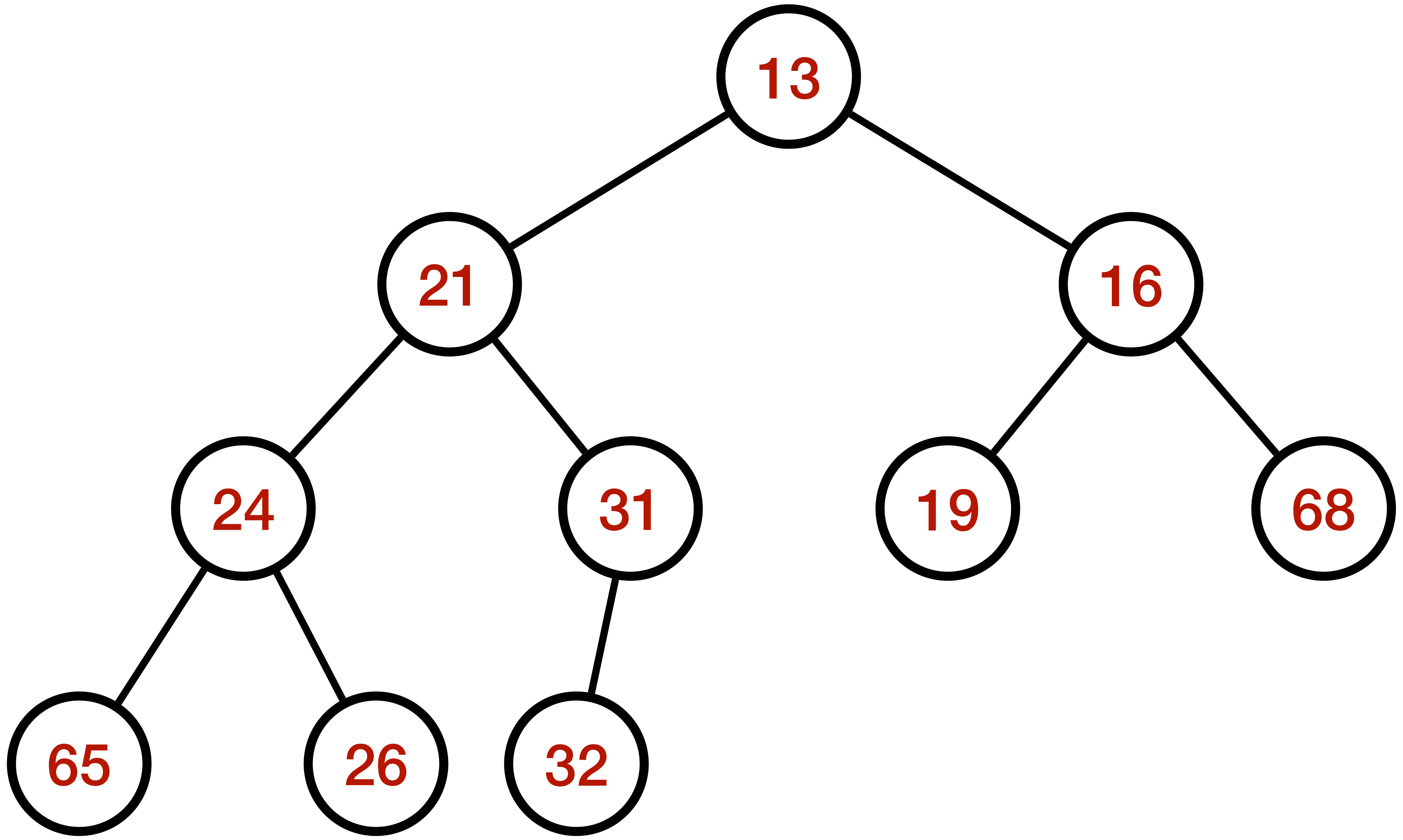


DeleteMin Operation: Attempt 1

- Delete 13
- 16 moves up
- 19 moves up
- Not left-filled

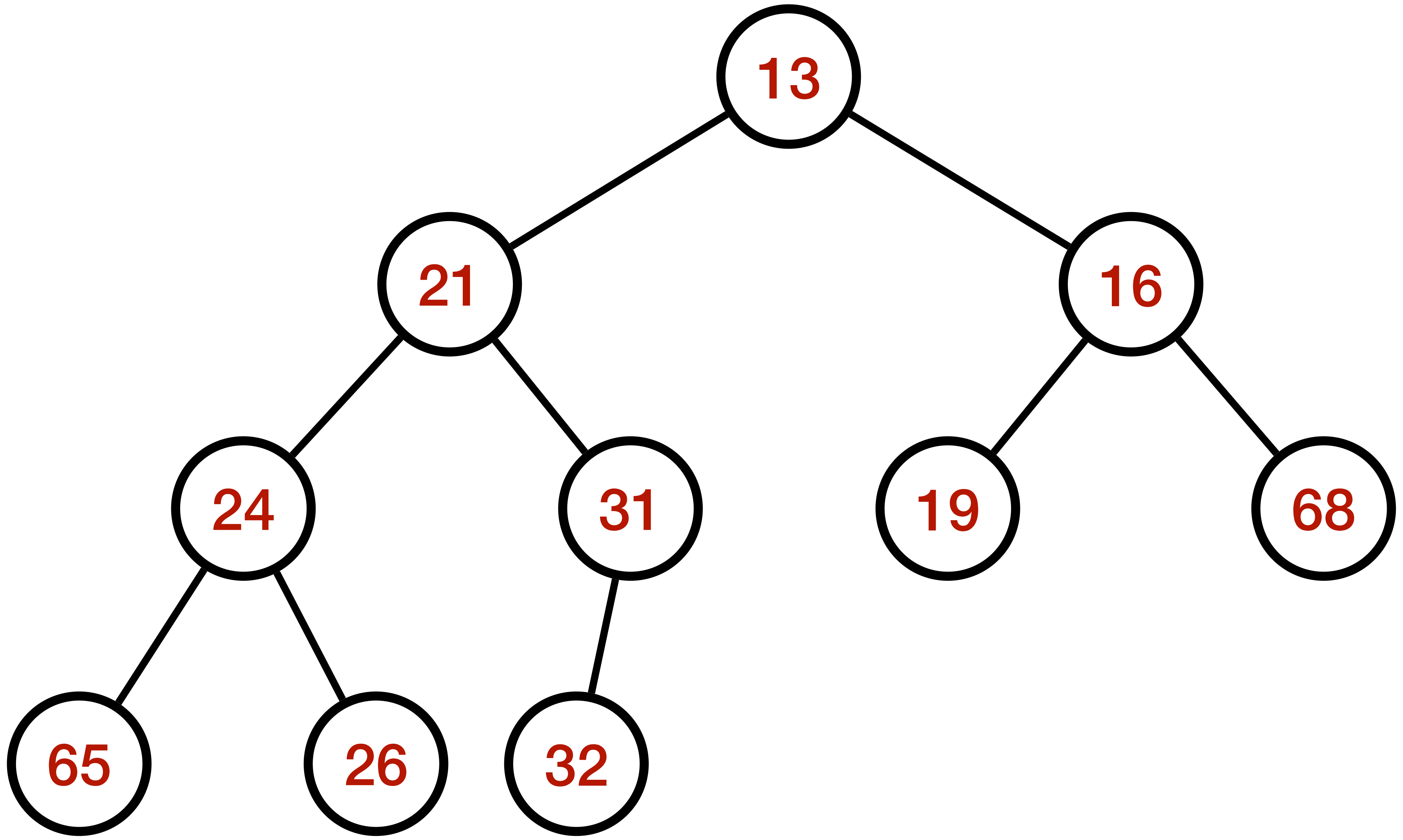


DeleteMin Operation: Attempt 2



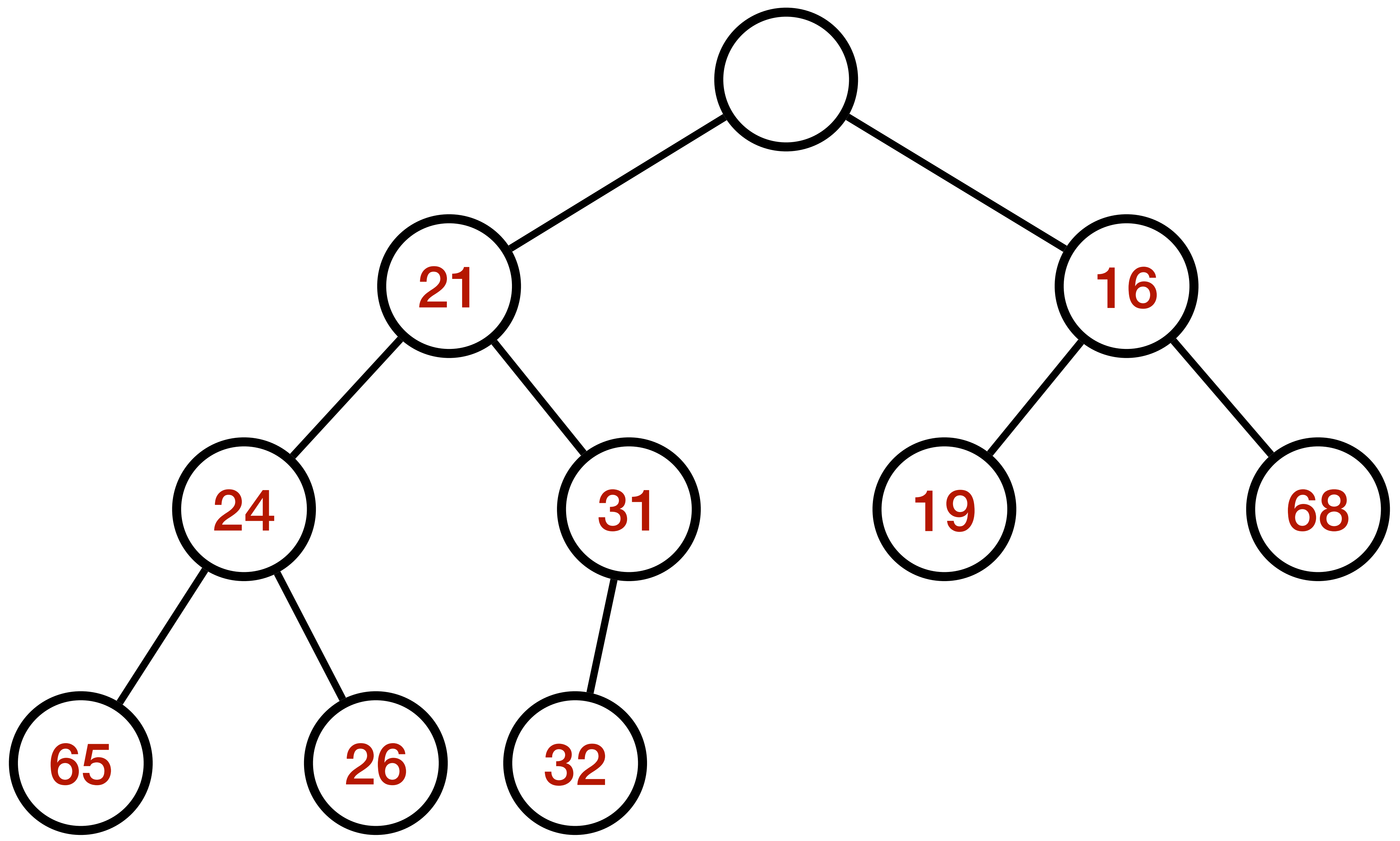
DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap



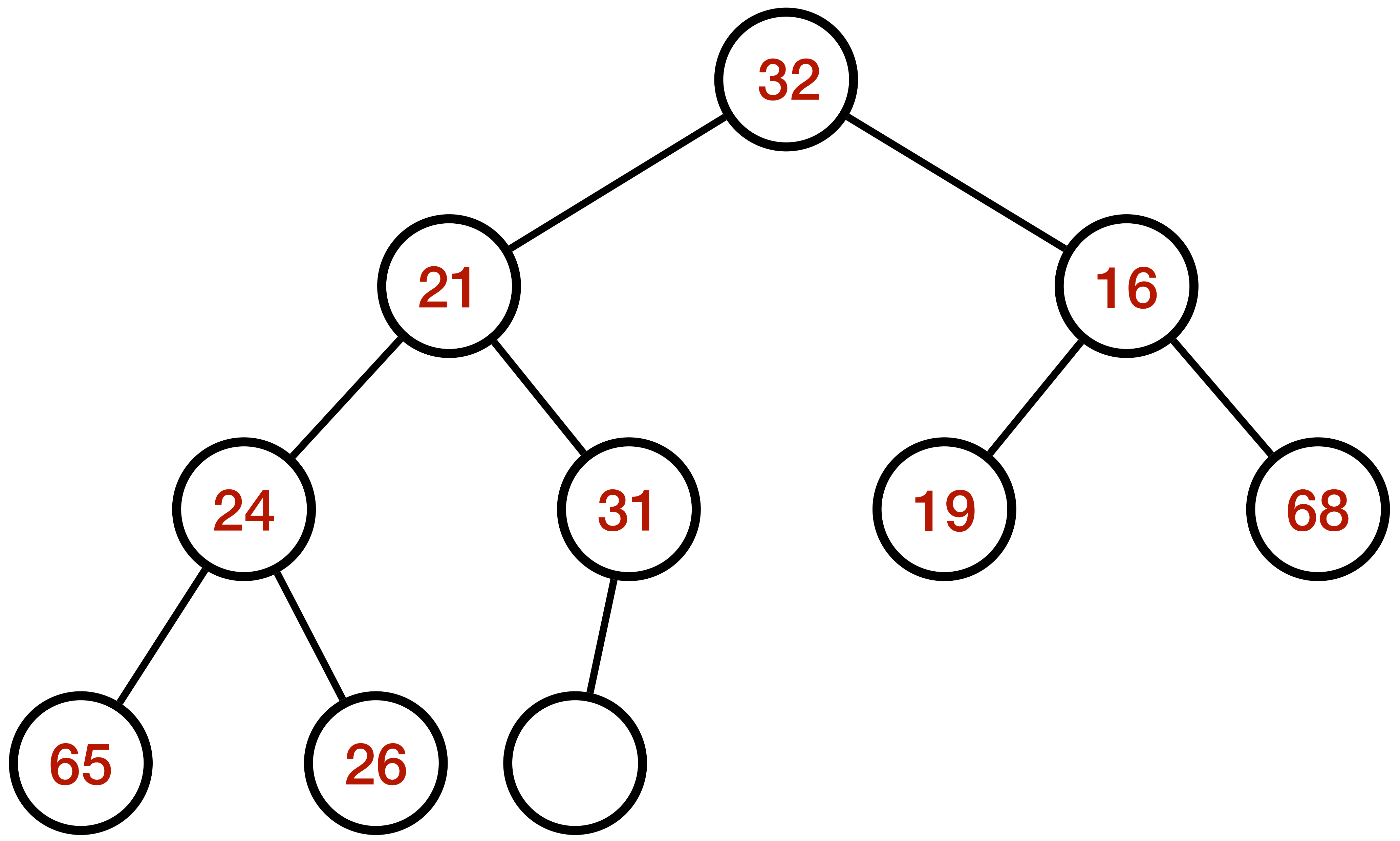
DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap



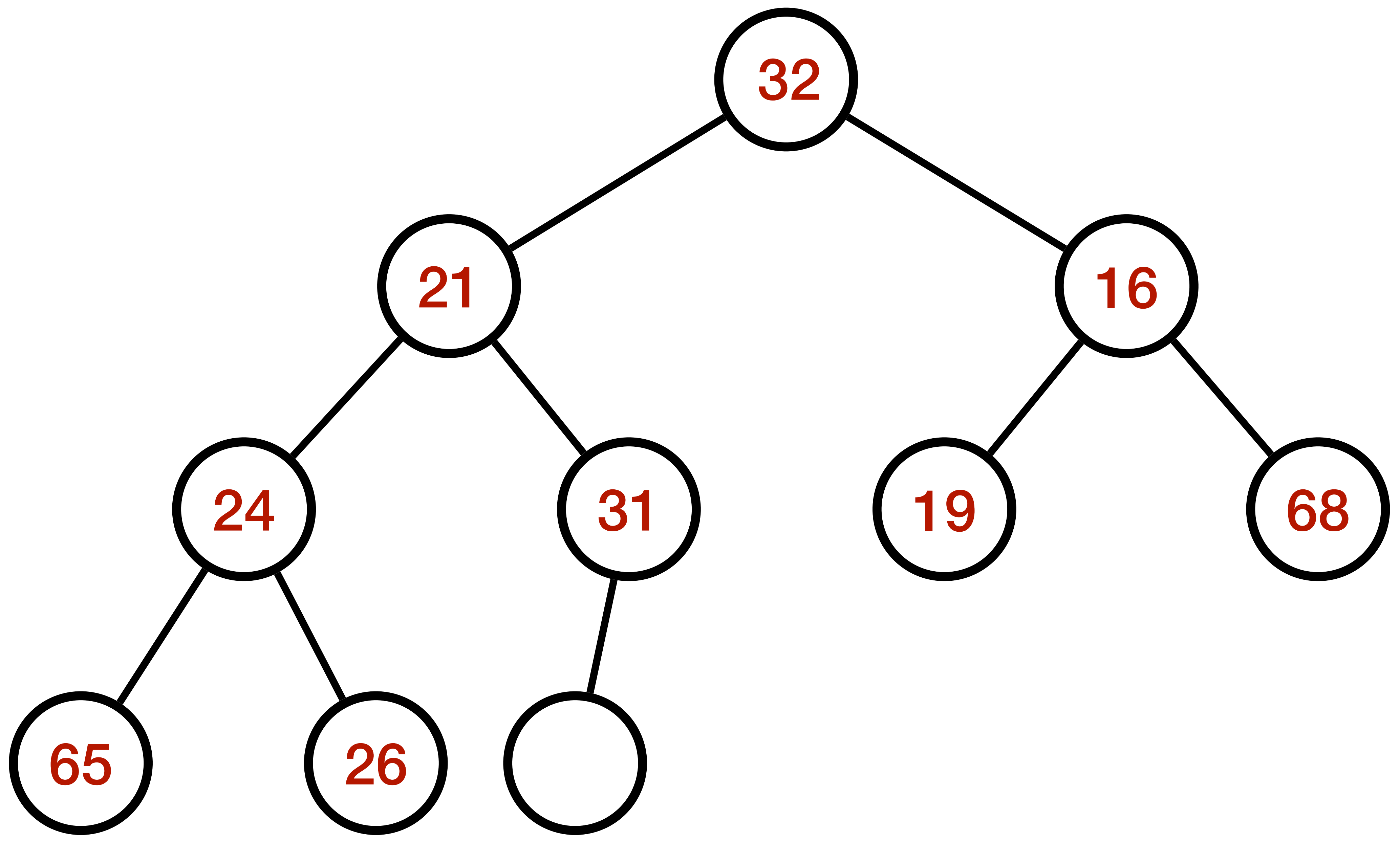
DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap



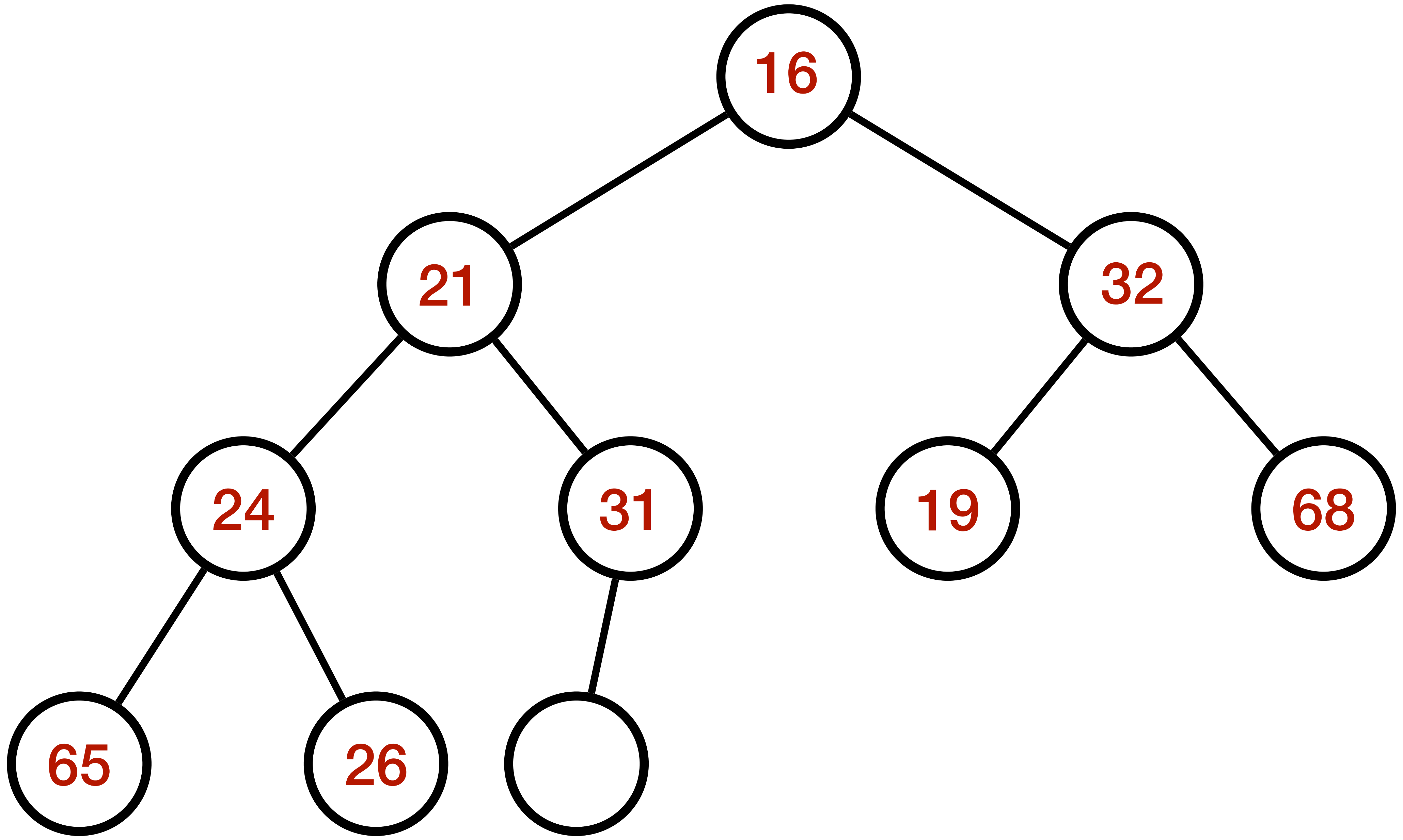
DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap
- HeapifyDown(rootIndex)



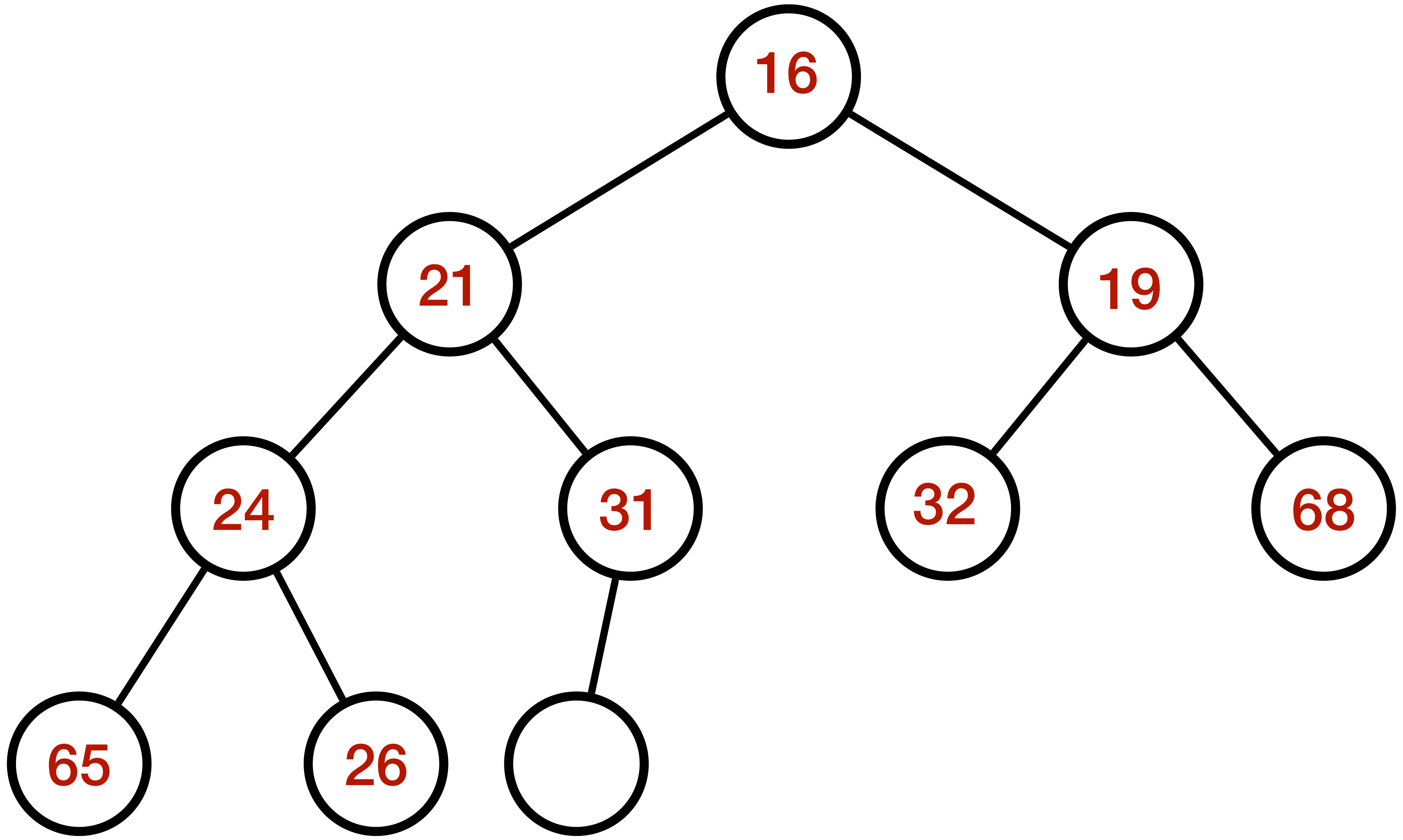
DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap
- HeapifyDown(rootIndex)



DeleteMin Operation: Attempt 2

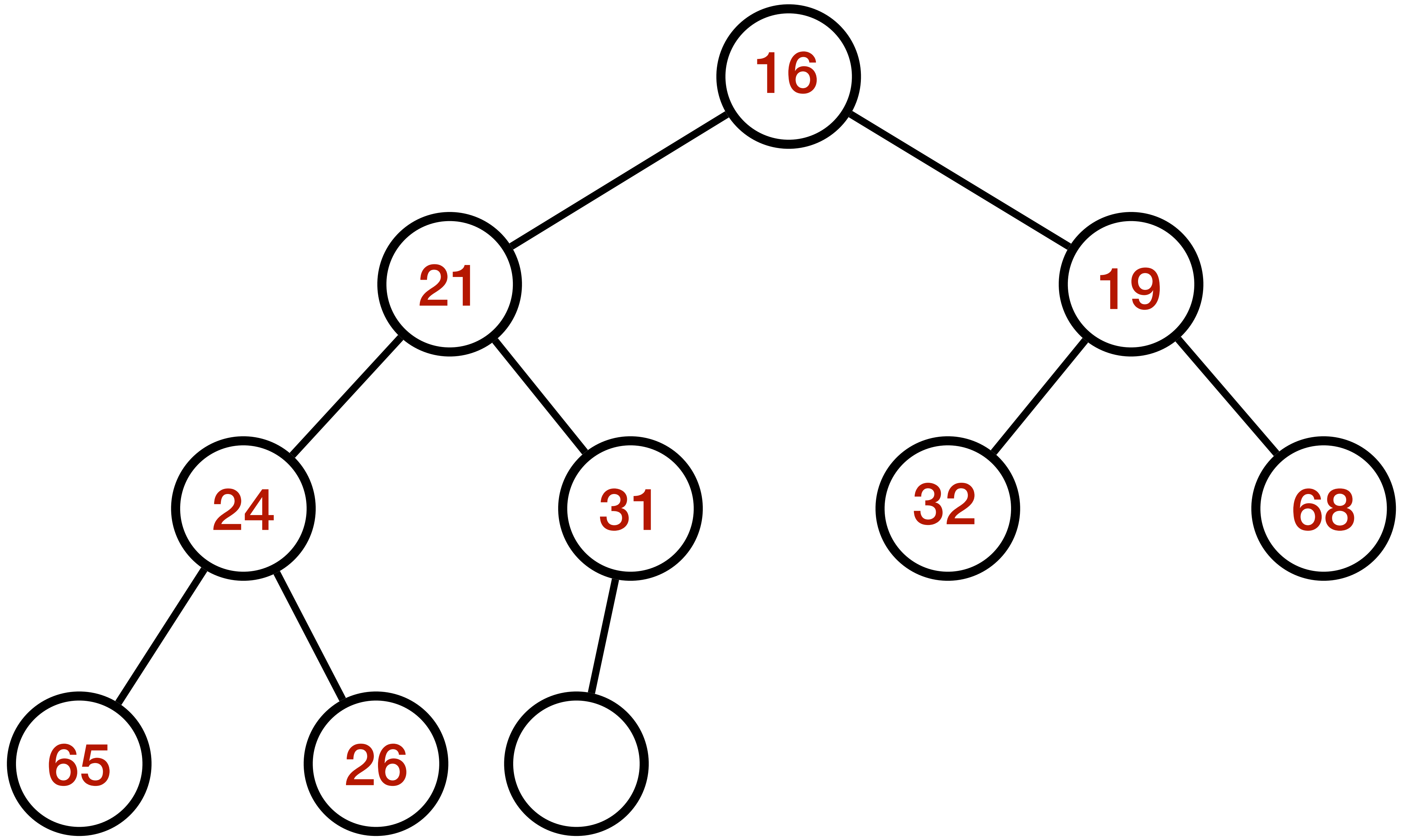
- Replace root element with the last element of the heap
- HeapifyDown(rootIndex)



DeleteMin Operation: Attempt 2

- Replace root element with the last element of the heap
- HeapifyDown(rootIndex)

```
void deleteMin() {
  if (heap.empty()) {
    std::cout << "Heap is empty!" << std::endl;
    return;
  }

  heap[0] = heap.back();
  heap.pop_back();

  heapifyDown(0);
}
```



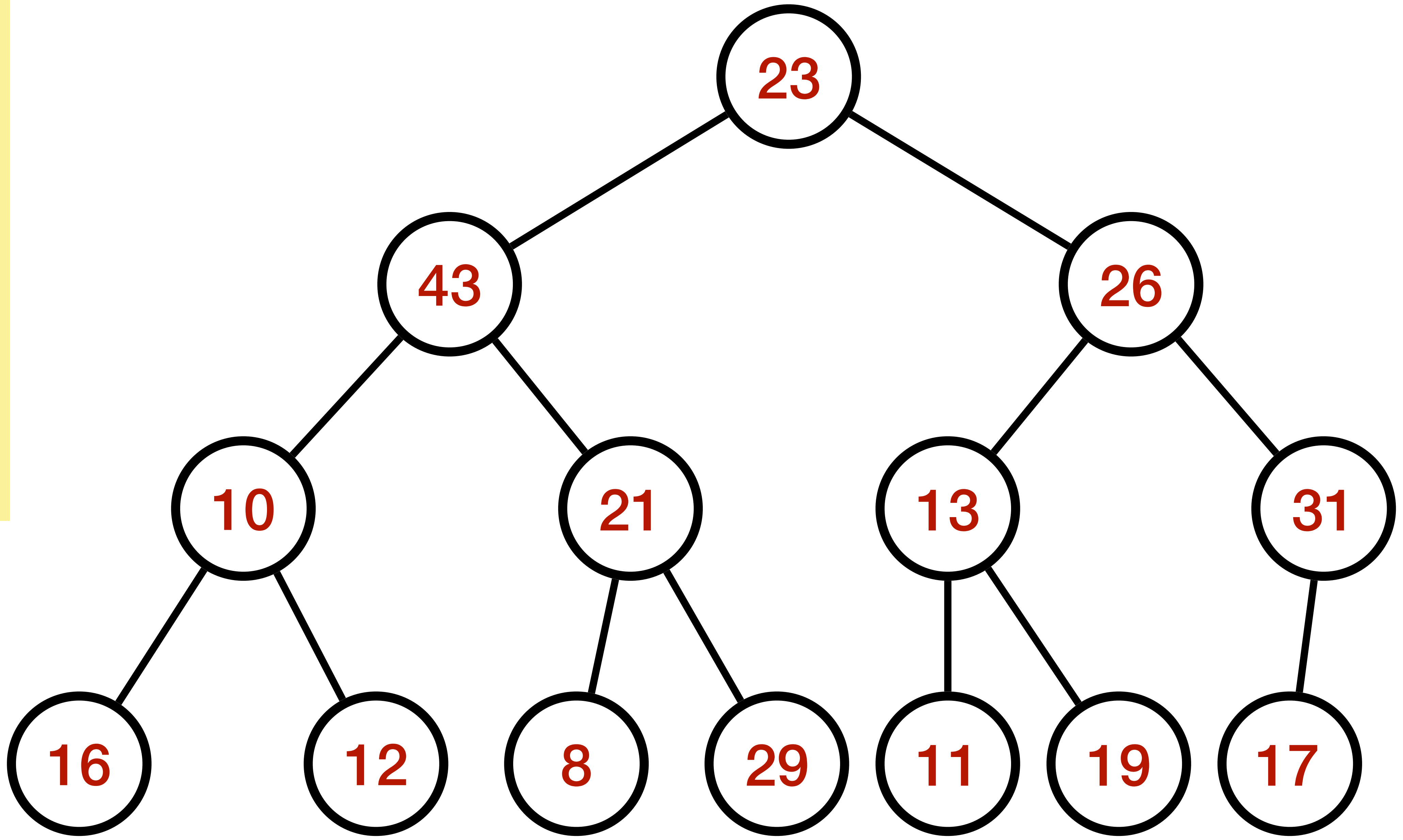
Building Heap

- Simple method — Repeatedly call insert method
  - Time complexity: $\Sigma_{i=1}^{n} log\, i = O(log\, n!) = O(nlog\, n)$
- Better solution: We start from the bottom and move up
- All leaves are heaps (inductive construction)

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```

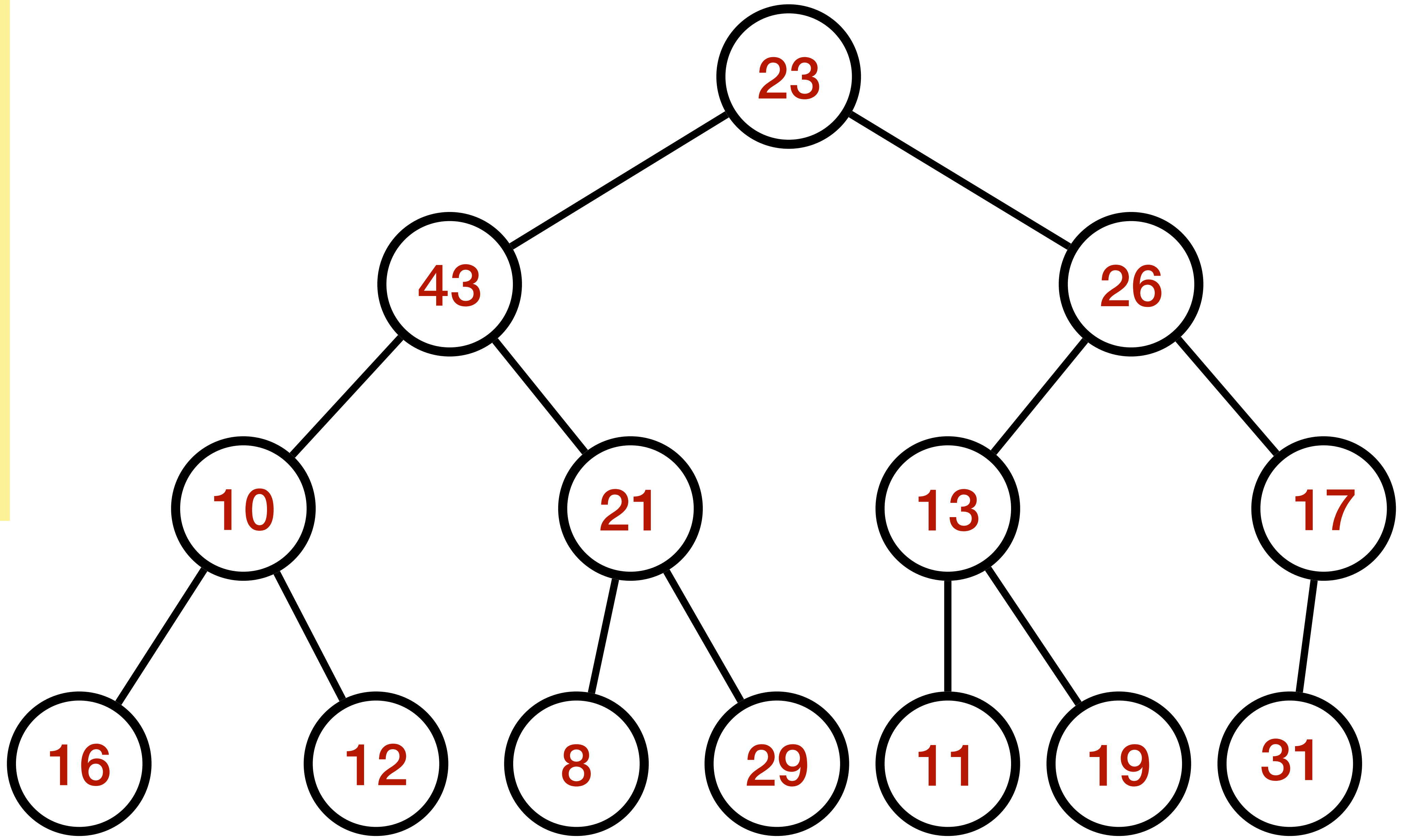
Building Heap

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```



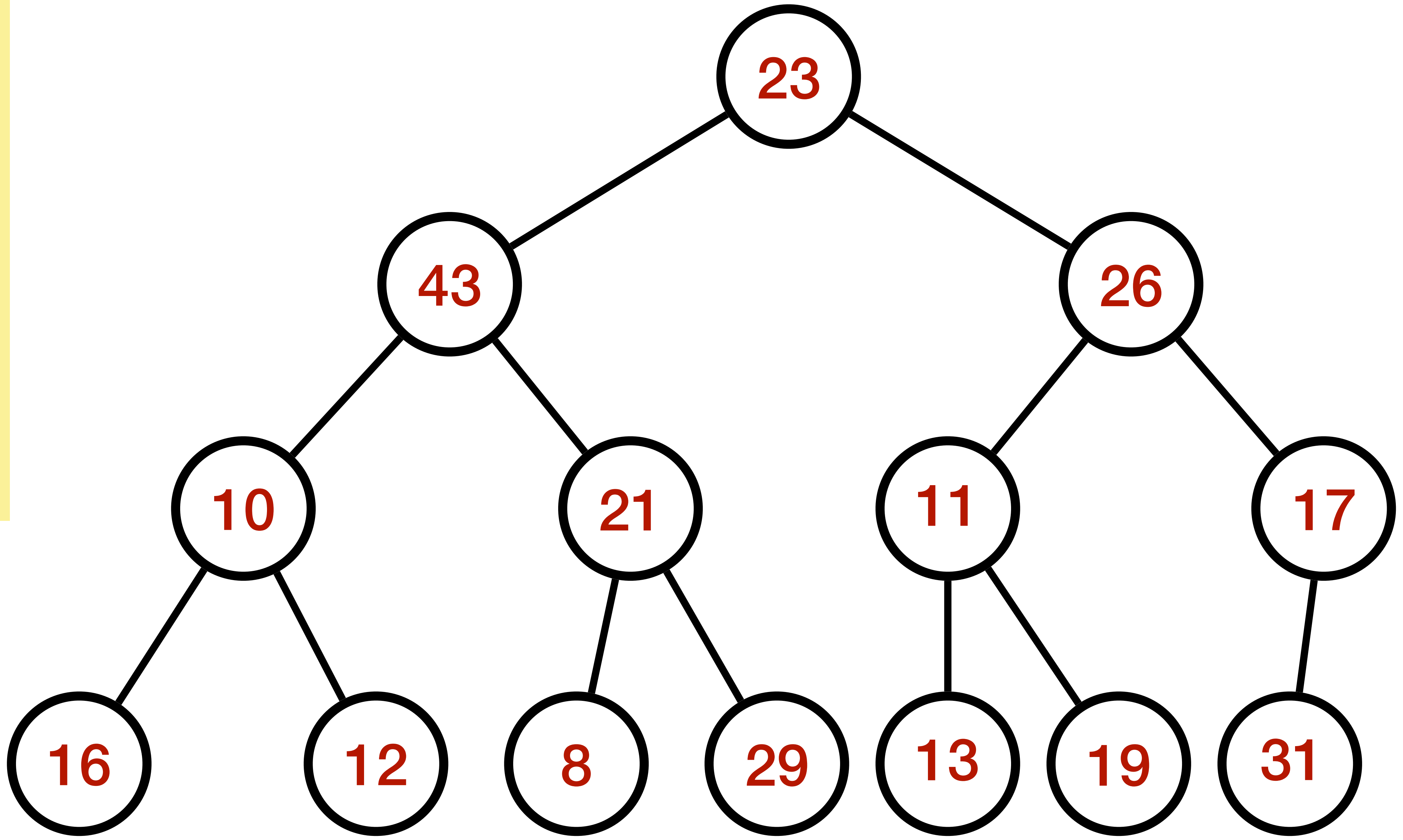
Building Heap

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```



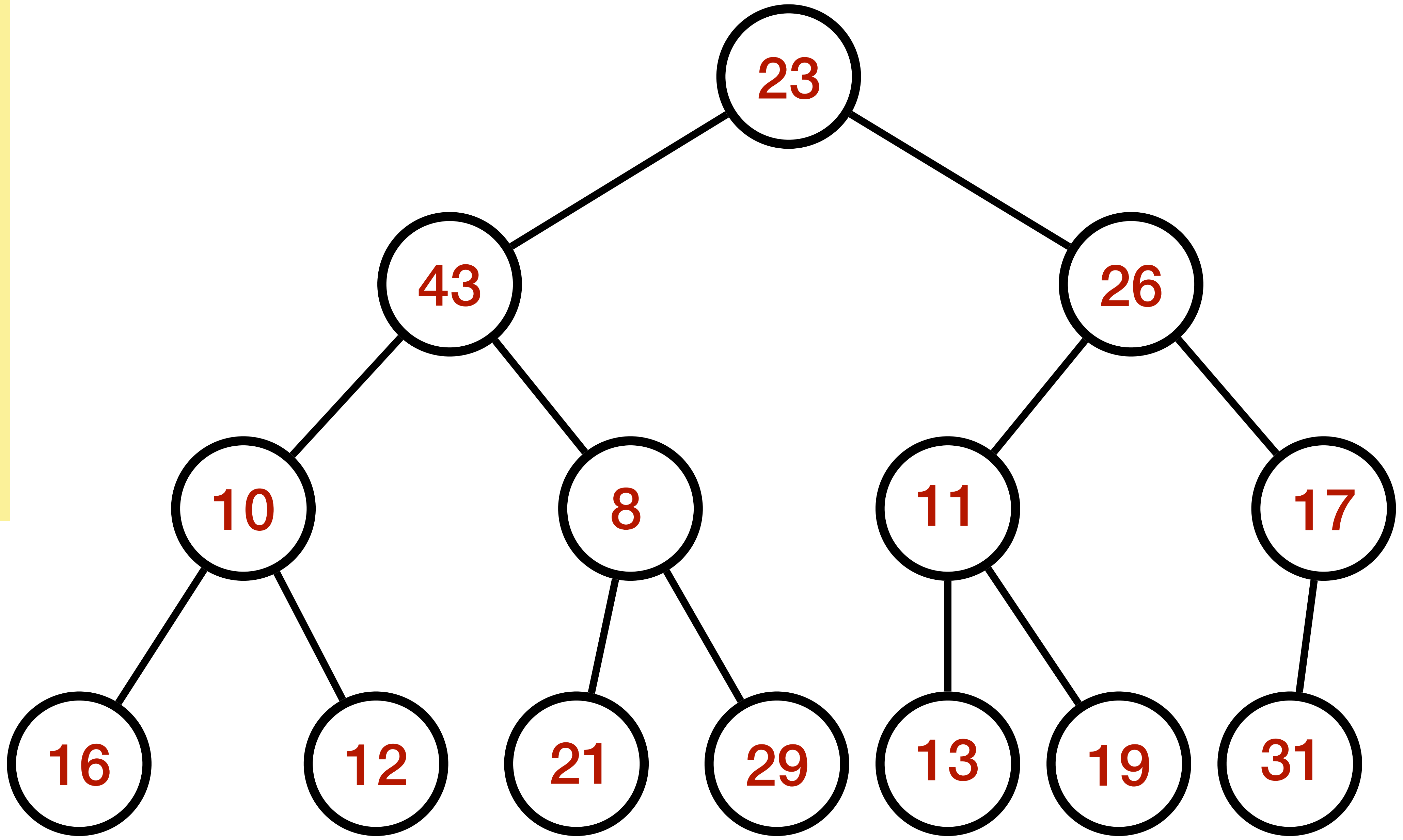
Building Heap

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```



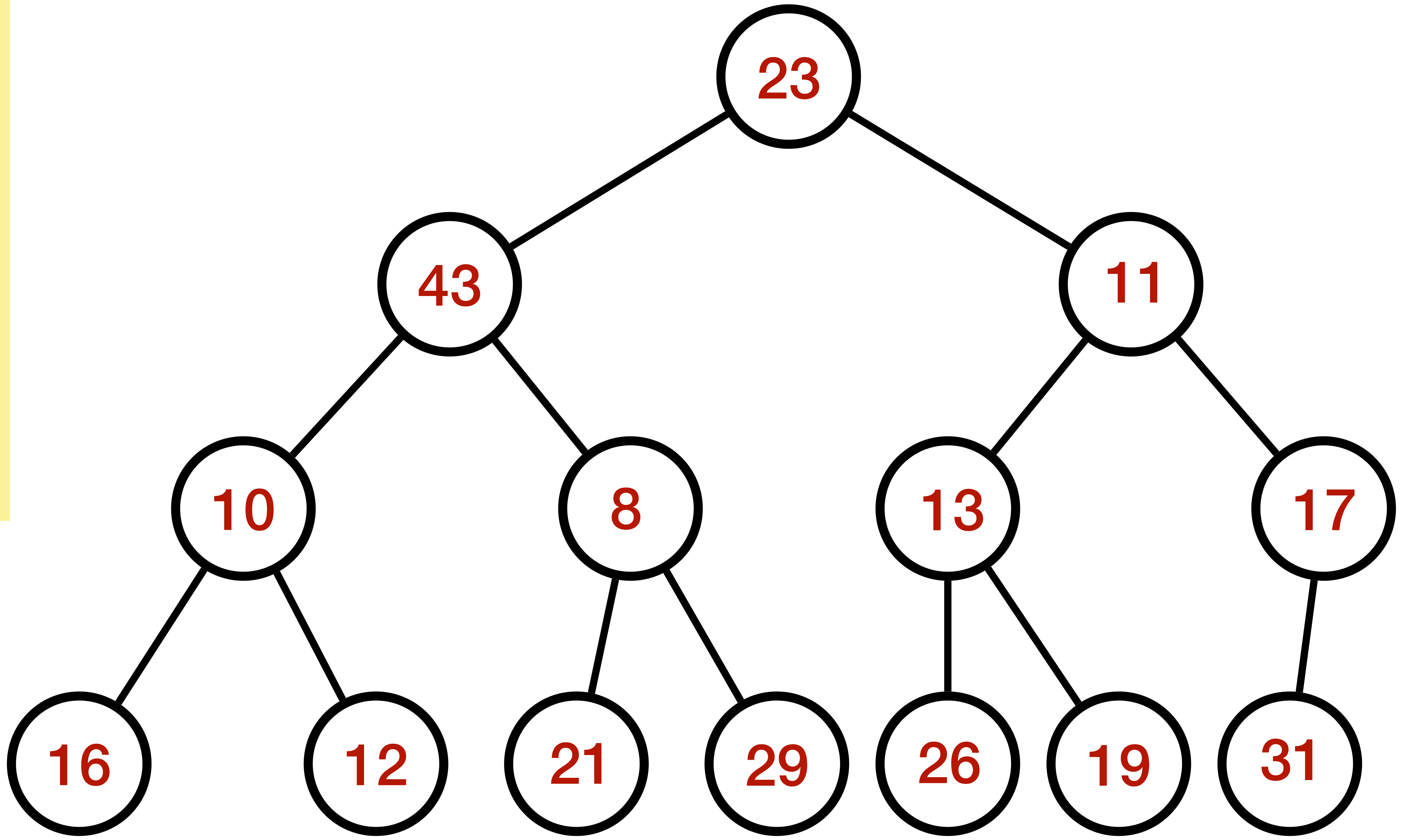
Building Heap

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```



Building Heap

```
void buildHeap(const std::vector<int> &arr) {
  heap = arr;
  int n = heap.size();

  for (int i = n / 2 - 1; i >= 0; i--) {
    heapifyDown(i);
  }
}
```



Building Heap: Analysis

Building Heap: Analysis

- Height of node : length of longest path from the node to leaf

Building Heap: Analysis

- Height of node : length of longest path from the node to leaf
- Height of tree: height of root

Building Heap: Analysis

- Height of node : length of longest path from the node to leaf
- Height of tree: height of root
- Time for HeapifyX(i): O(height of the subtree rooted at i)

Building Heap: Analysis

- Height of node : length of longest path from the node to leaf
- Height of tree: height of root
- Time for HeapifyX(i): O(height of the subtree rooted at i)
- Assume: $n = 2^k - 1$ (a complete binary tree — only help us simplify the analysis)

Building Heap: Analysis

Building Heap: Analysis

- For the n/2 nodes of height 1: Heapify requires at most 1 swap

Building Heap: Analysis

- For the n/2 nodes of height 1: Heapify requires at most 1 swap
- For the n/4 nodes of height 2: Heapify requires at most 2 swaps

Building Heap: Analysis

- For the n/2 nodes of height 1: Heapify requires at most 1 swap
- For the n/4 nodes of height 2: Heapify requires at most 2 swaps
- For the $n/2^i$ nodes of height i: Heapify requires at most i swaps

Building Heap: Analysis

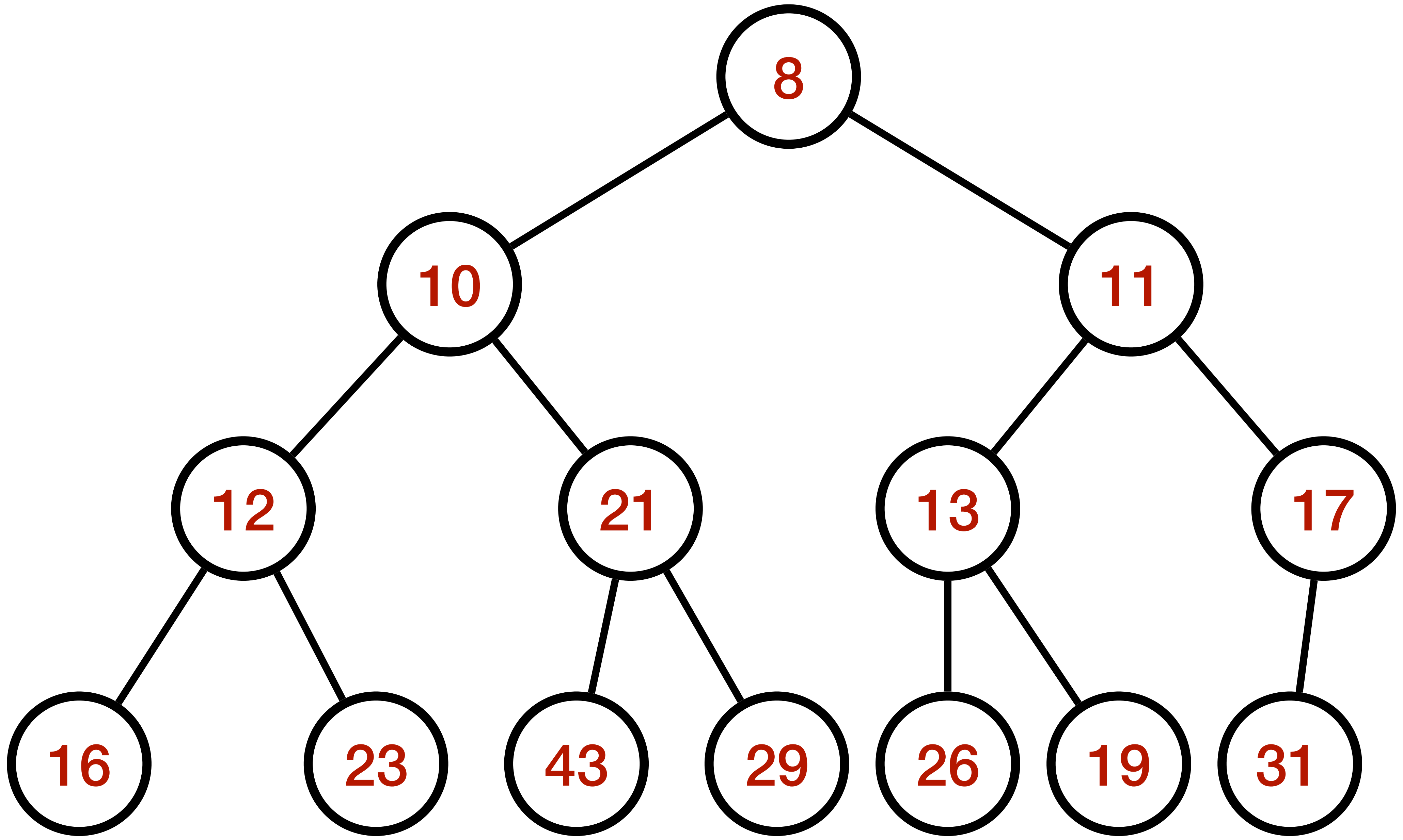
- For the n/2 nodes of height 1: Heapify requires at most 1 swap
- For the n/4 nodes of height 2: Heapify requires at most 2 swaps
- For the $n/2^i$ nodes of height i: Heapify requires at most i swaps
- Total number of swaps: $\Sigma_{i=1}^{log\, n} n\,.\, i/2^i = O(n)$

Heap Sort

- Create a heap: $T(n) = O(n)$
- Do **DeleteMin** repeatedly till the heap becomes empty: $T(n) = O(n \log n)$
- Alternative strategy: No other space constraint, i.e., **in-place sort**
    - Do **DeleteMin** and move the deleted element to the end of the heap
    - Heapify the rest

In-Place Heap Sort

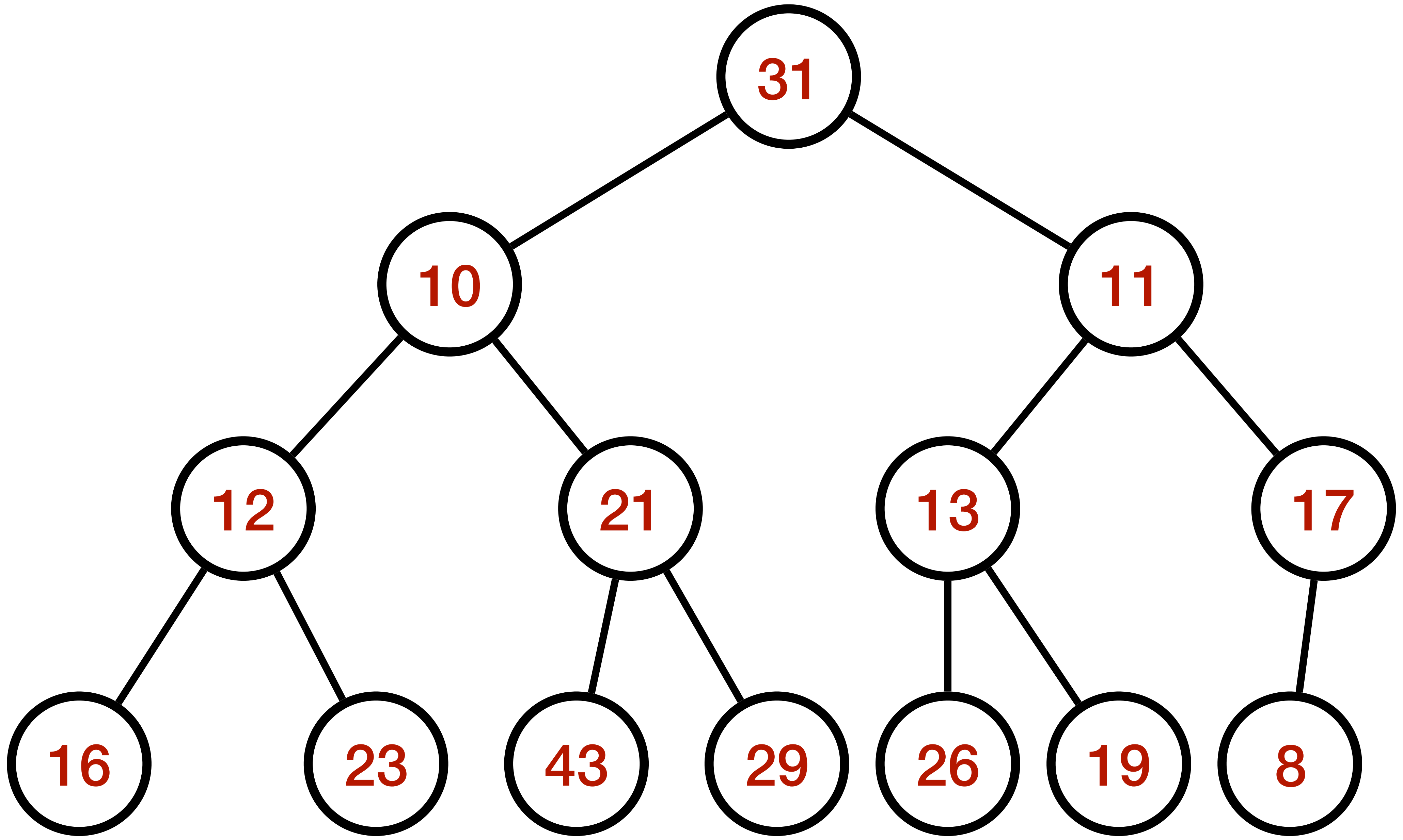
- NOTE: **Heap size is reduced by 1** after each such operation



- Time complexity: O(n log n)

In-Place Heap Sort

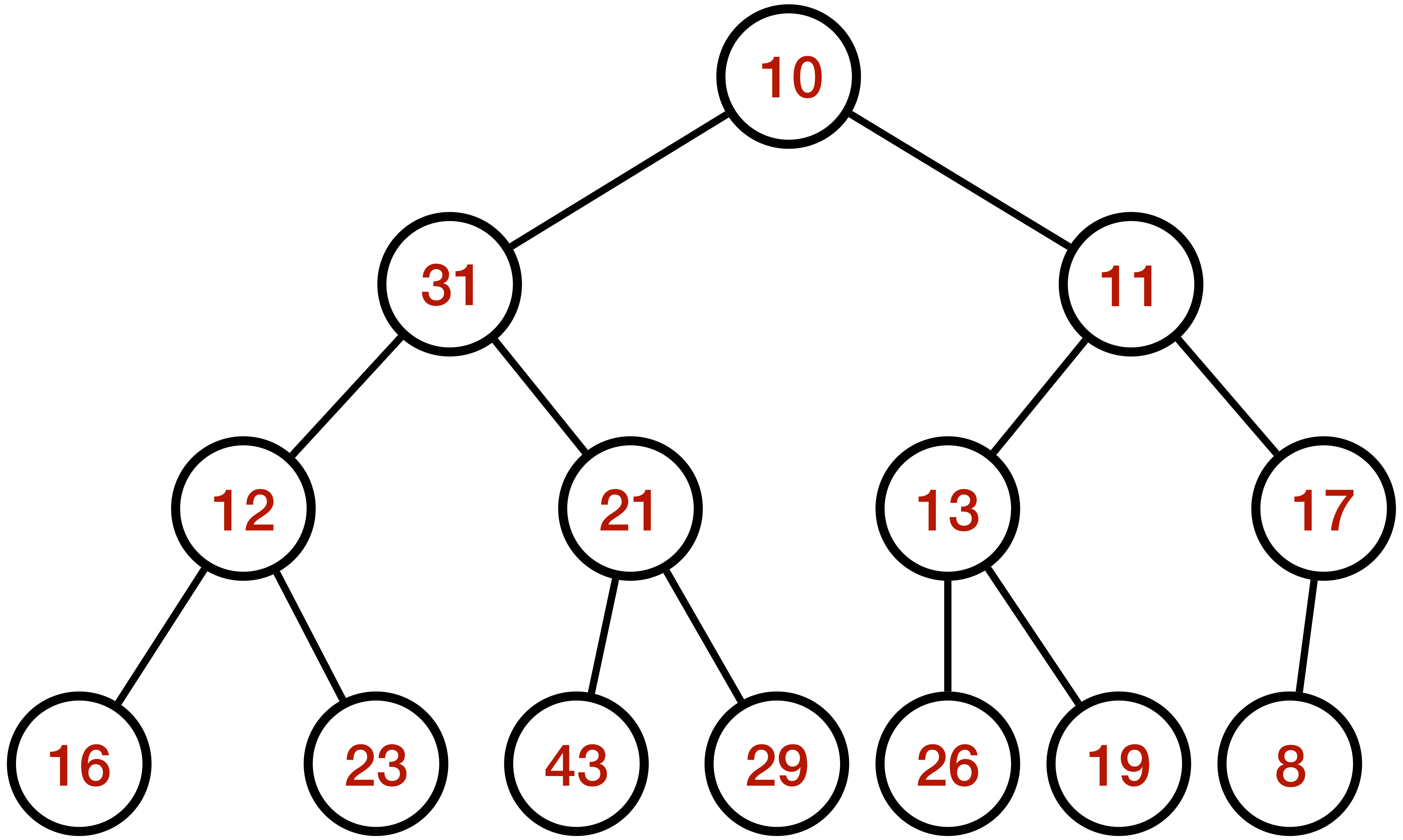
- NOTE: **Heap size is reduced by 1** after each such operation



- Time complexity: O(n log n)

In-Place Heap Sort

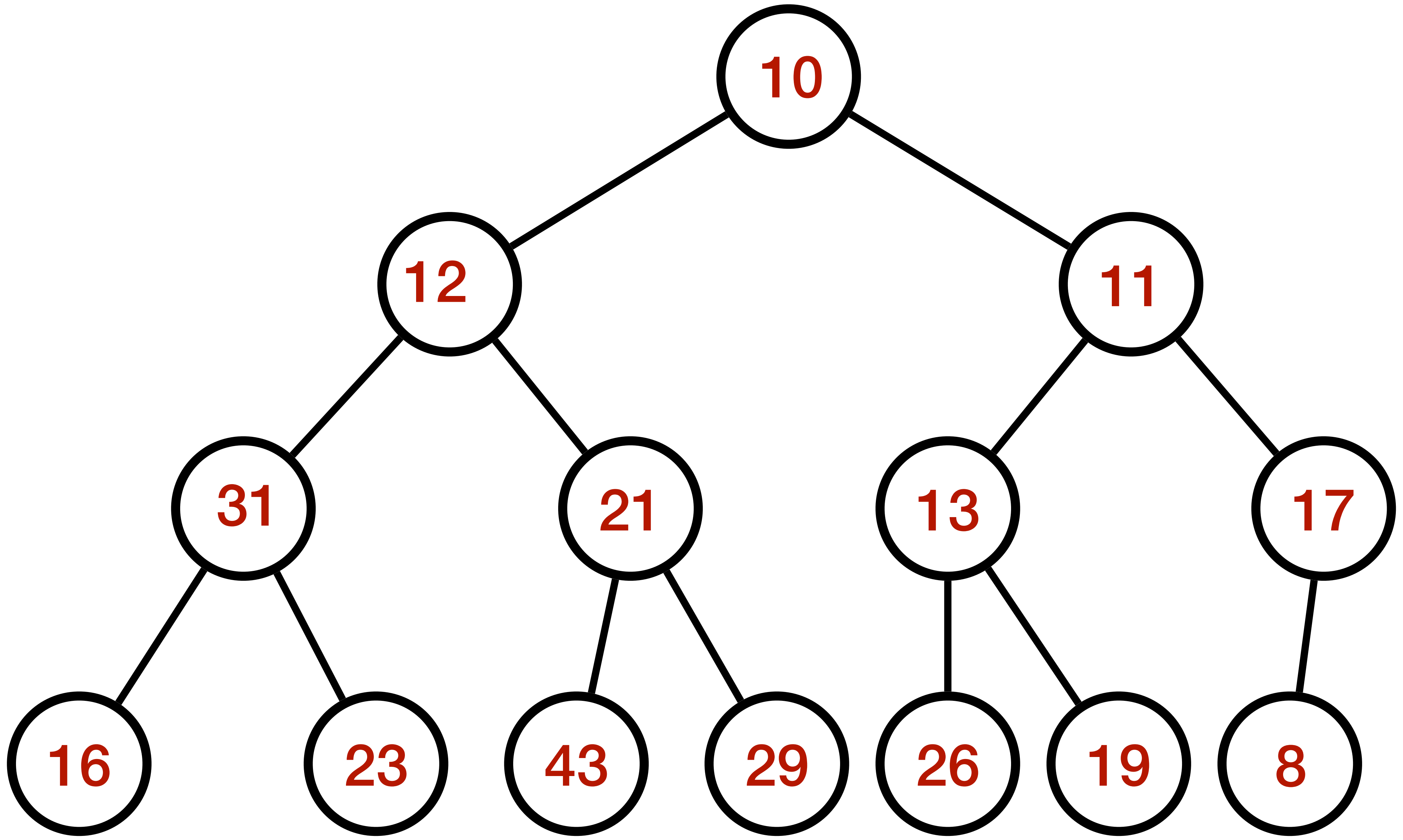
- NOTE: **Heap size is reduced by 1** after each such operation



- Time complexity: O(n log n)

In-Place Heap Sort

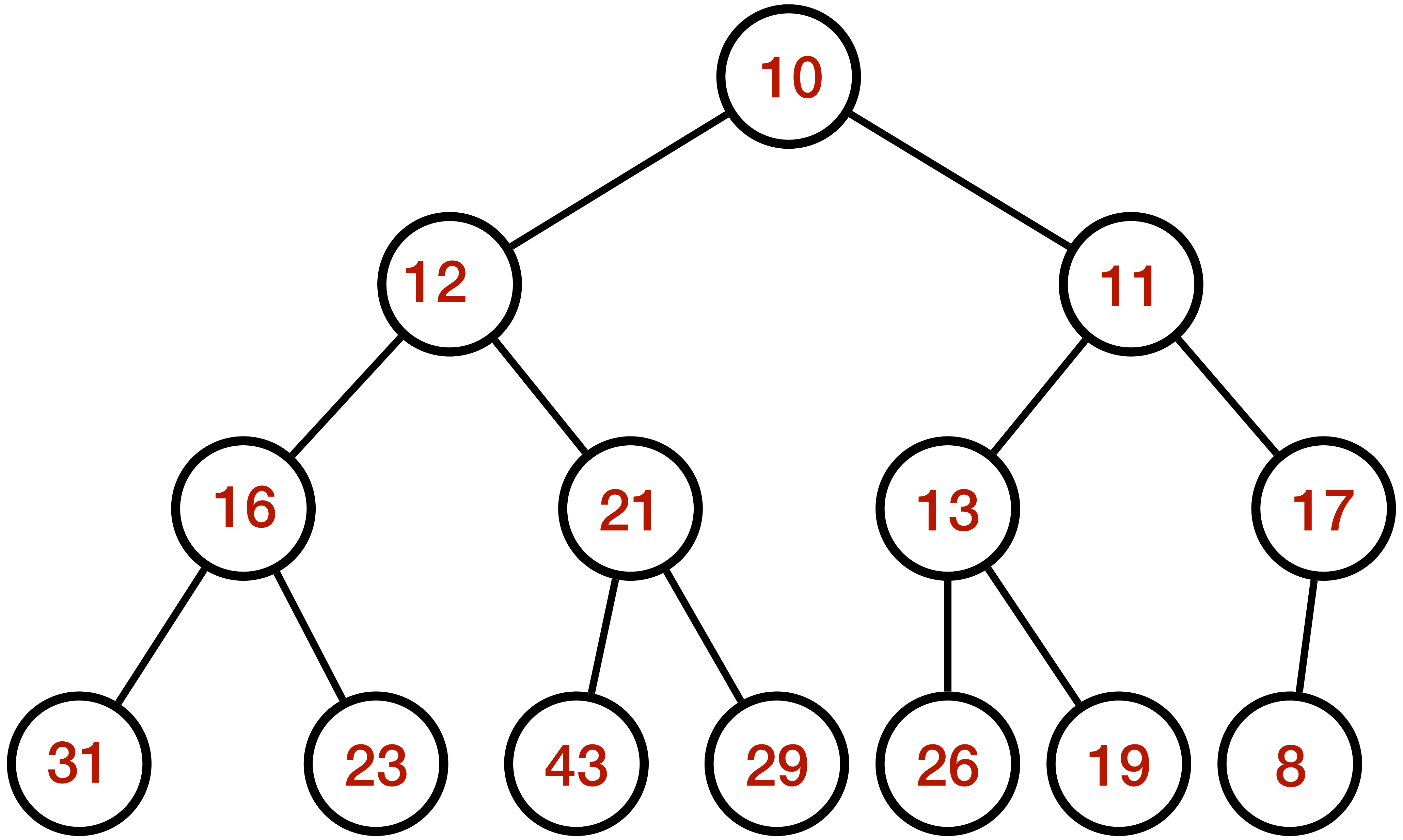
- NOTE: **Heap size is reduced by 1** after each such operation



- Time complexity: O(n log n)

In-Place Heap Sort

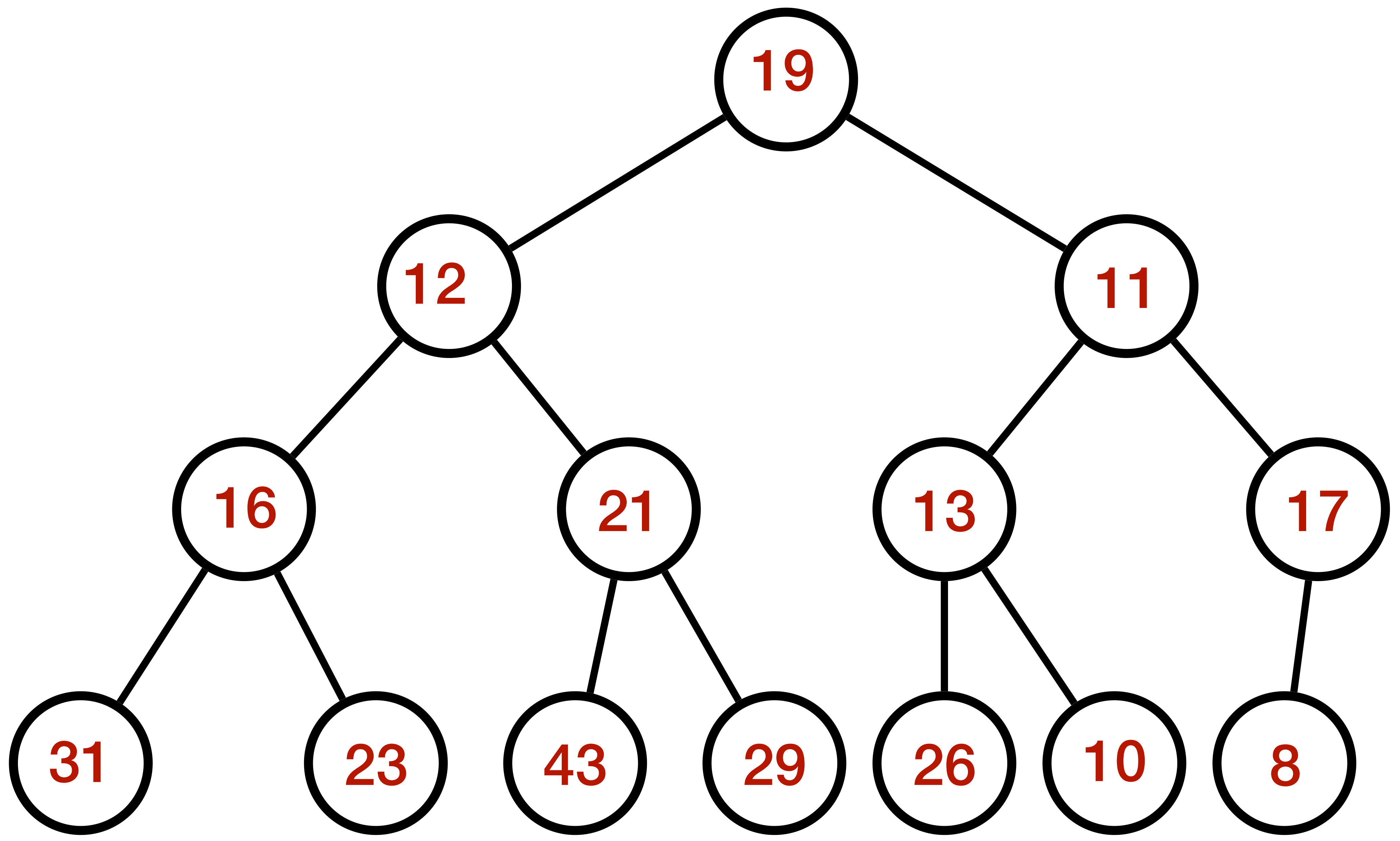
- NOTE: **Heap size is reduced by 1** after each such operation



- Time complexity: O(n log n)

In-Place Heap Sort

- NOTE: **Heap size is reduced by 1** after each such operation

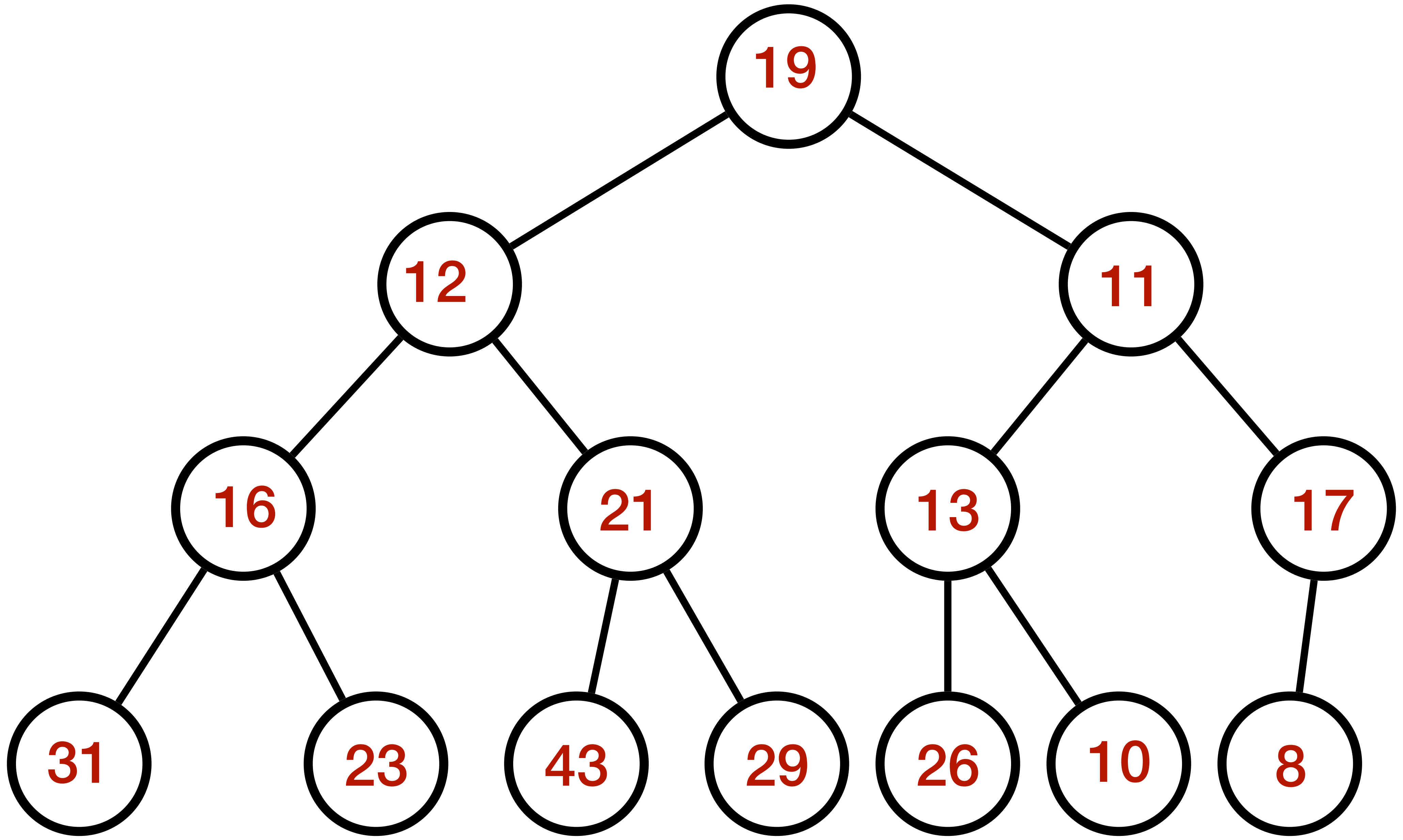


- Time complexity: O(n log n)

In-Place Heap Sort

- NOTE: **Heap size is reduced by 1** after each such operation

```
void heapSort() {
  int n = heap.size();
  // Extract elements from heap one by one
  for (int i = n - 1; i > 0; i--) {
    // Move the root to the end
    std::swap(heap[0], heap[i]);

    // Heapify the reduced heap
    heapifyDown(i,0);
  }
}
```



- Time complexity: O(n log n)

Runtime Analysis

- A heap of n nodes has height O(log n)
- Insertion (heapifyUp along a path) — at most O(log n) steps
- HeapifyDown — O(log n)
  - An element may be moved all the way to the last level
- DeleteMin — O(log n)
- BuildHeap — O(n)
- HeapSort — O(n log n)

Applications: The Selection Problem

Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements

Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1A:

Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1A:
  - Read the elements in an array

Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1A:
  - Read the elements in an array
  - Sort the array

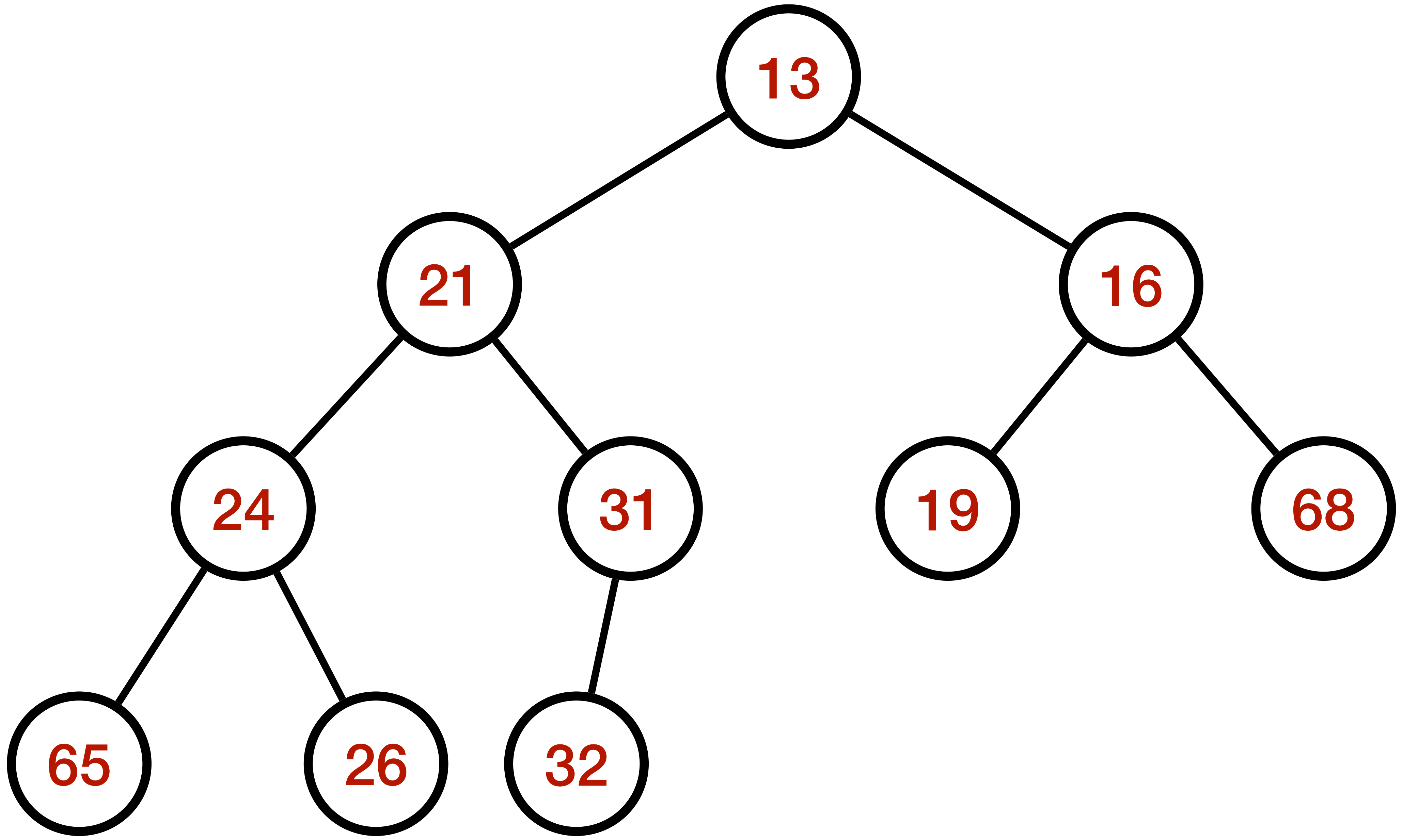
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1A:
  - Read the elements in an array
  - Sort the array
  - Return the k^{th} indexed element from the sorted array

Applications: The Selection Problem

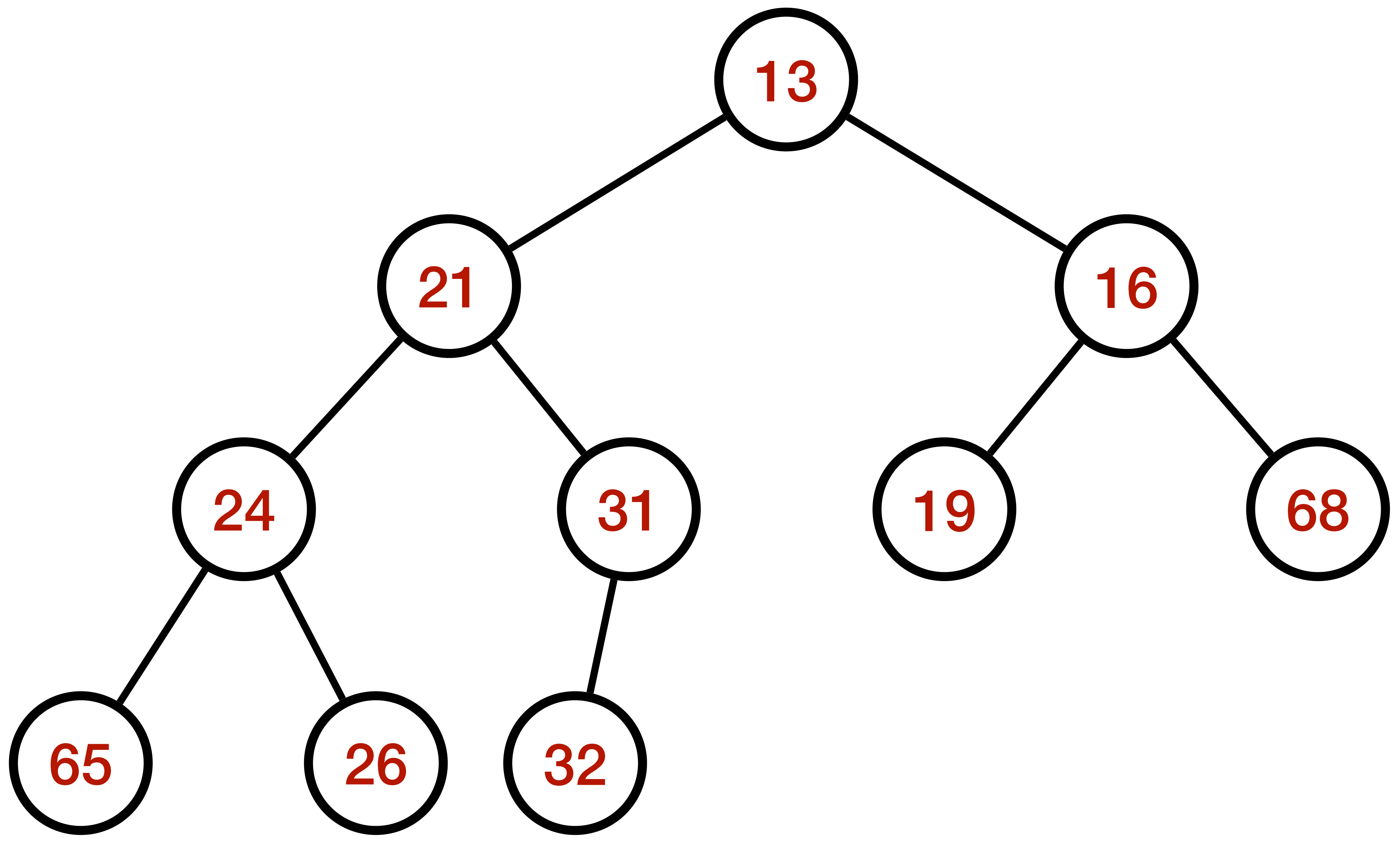
- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1A:
  - Read the elements in an array
  - Sort the array
  - Return the k^{th} indexed element from the sorted array
  - Time complexity: $O(n^2)$ with simple sorting; $O(n\,log\,n)$ otherwise.

Applications: The Selection Problem



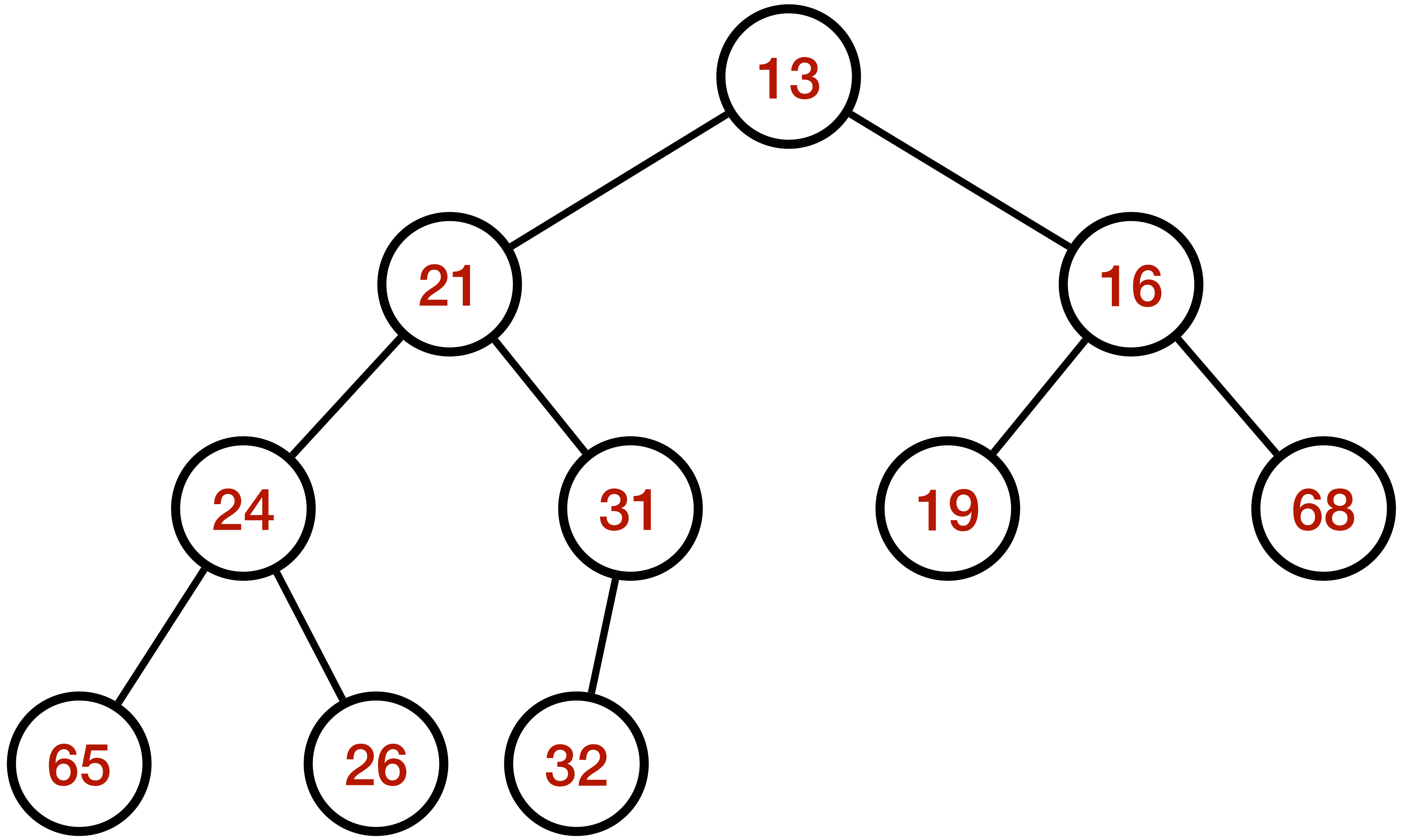
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements



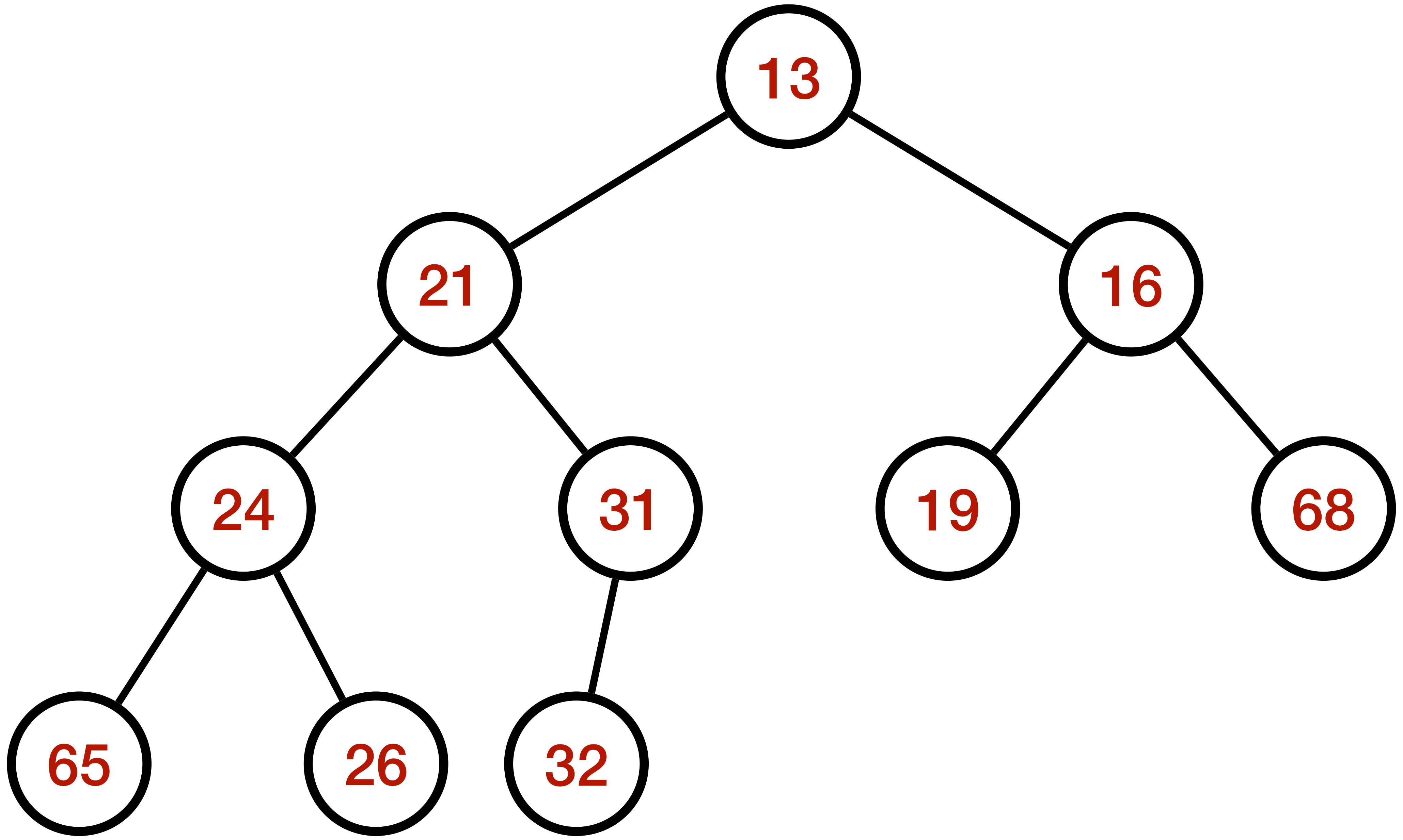
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:



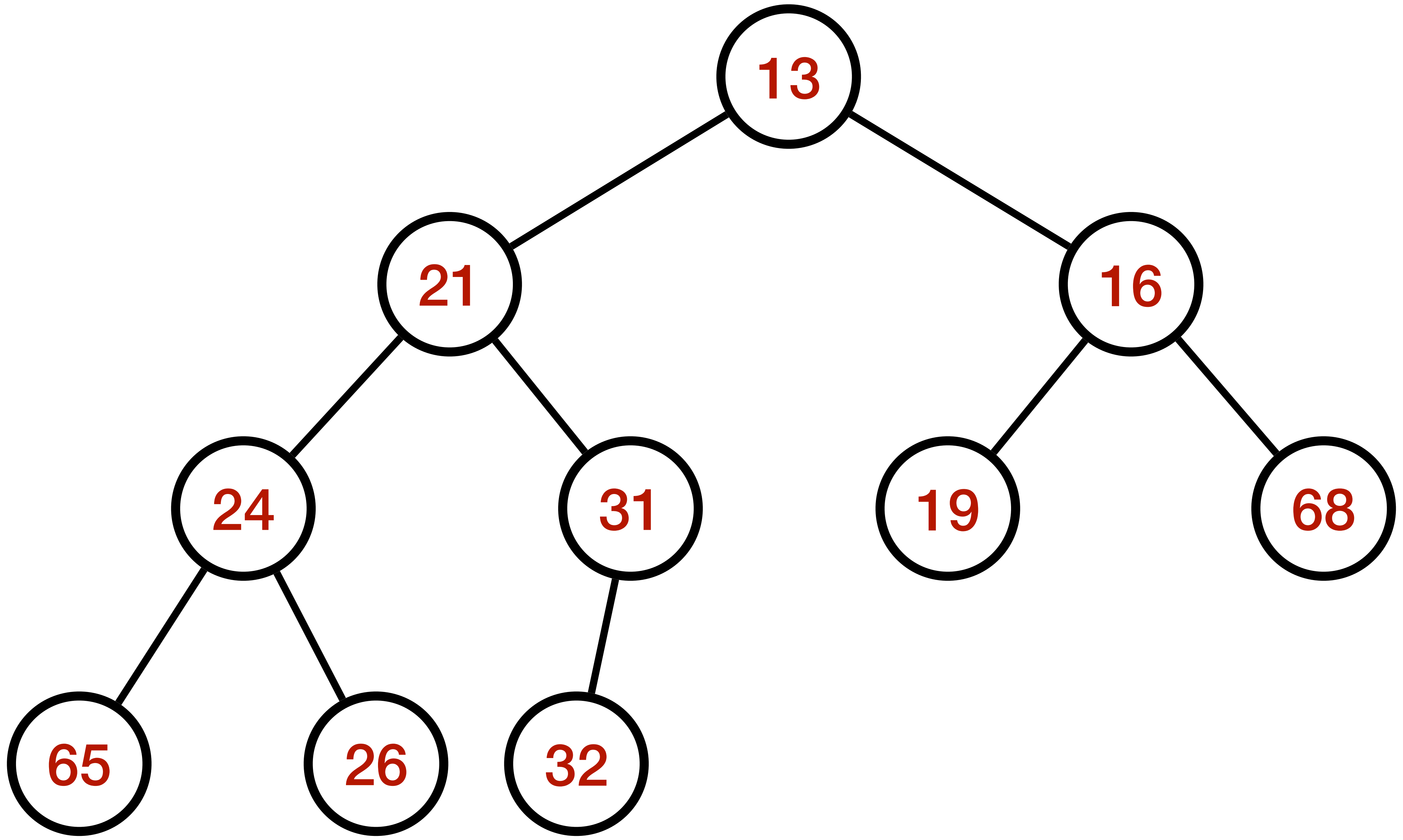
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:
  - Read **only** k elements in an array



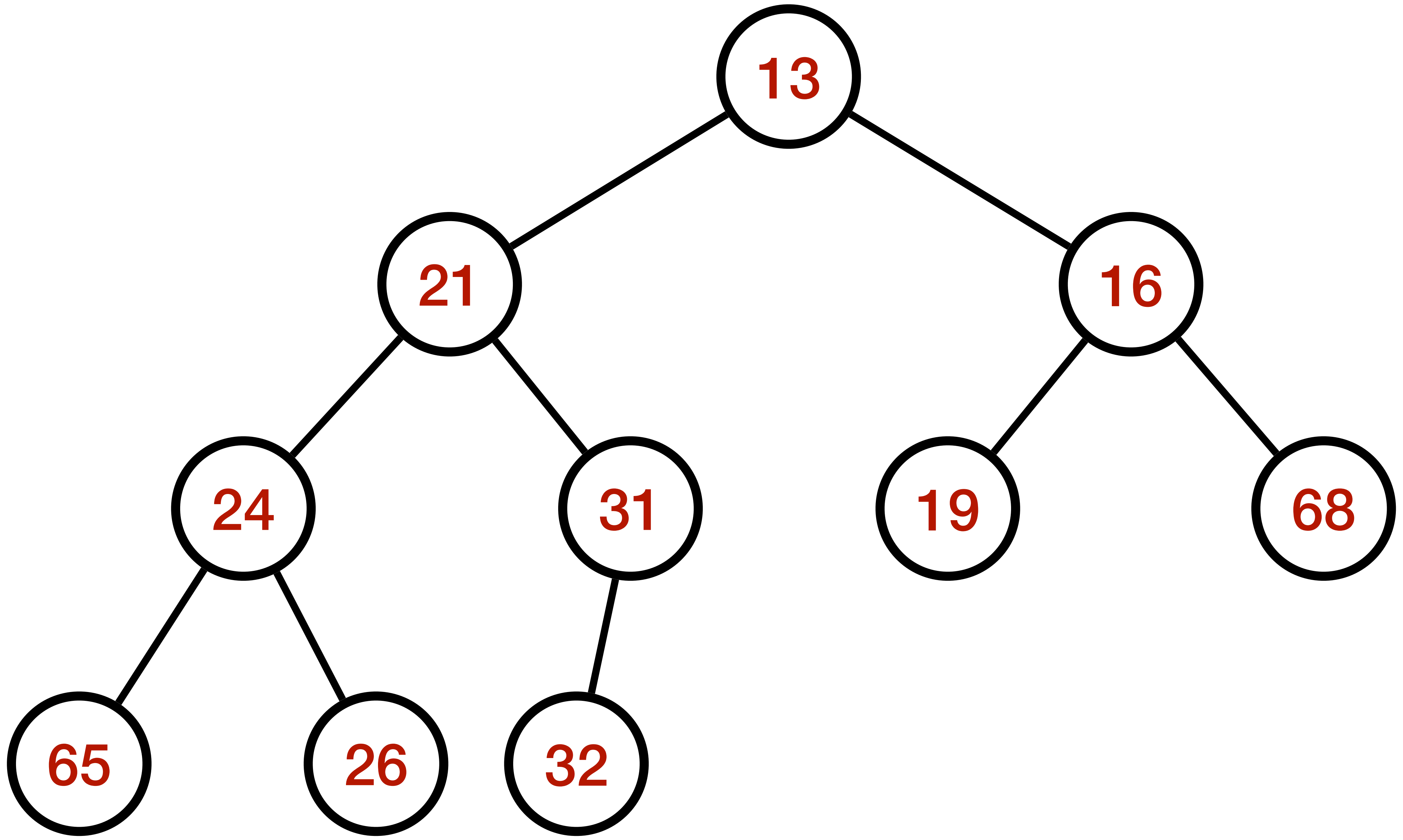
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:
  - Read **only** k elements in an array
  - Sort the array



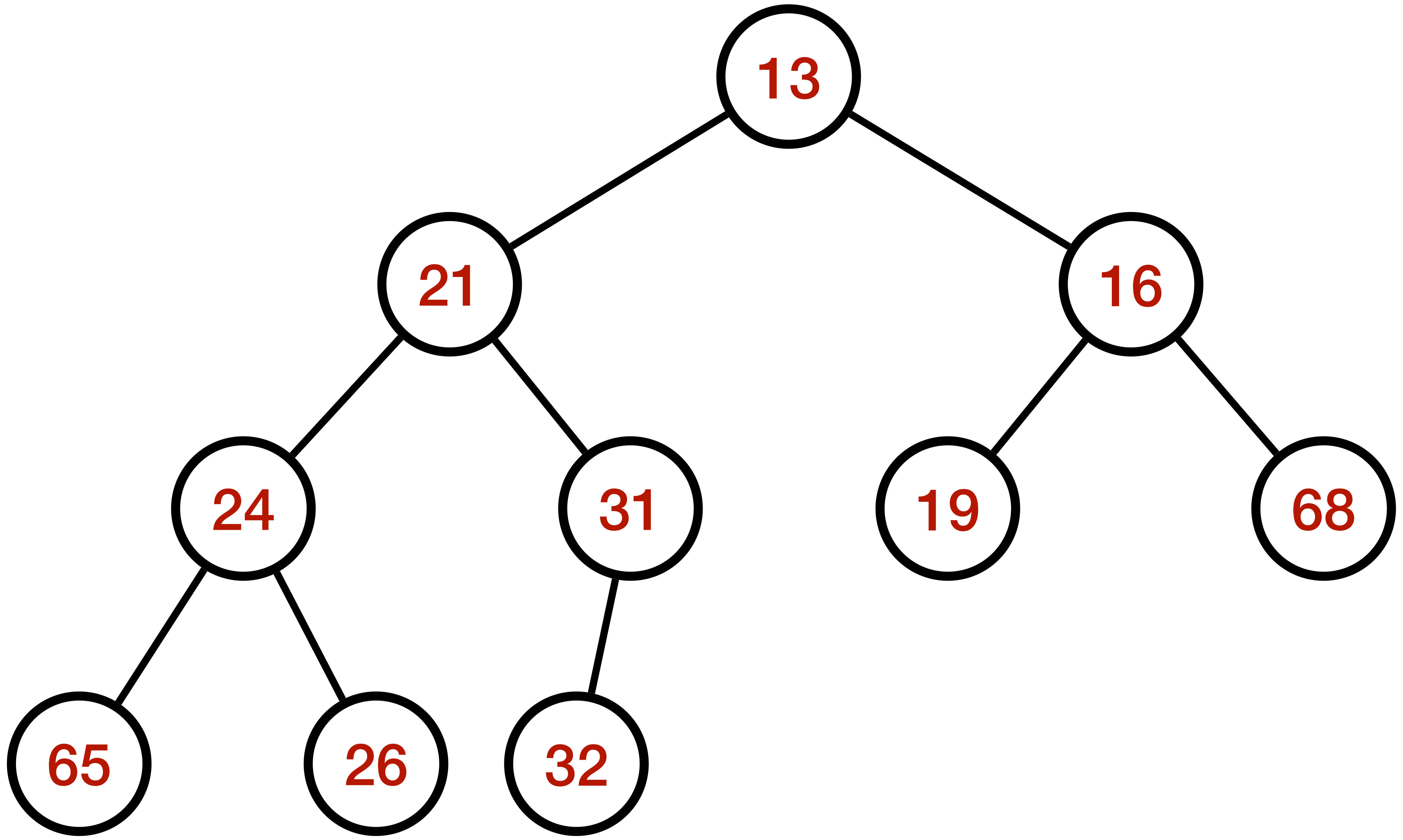
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:
  - Read **only** k elements in an array
  - Sort the array
  - The smallest is at k^{th} position. For the remaining elements:



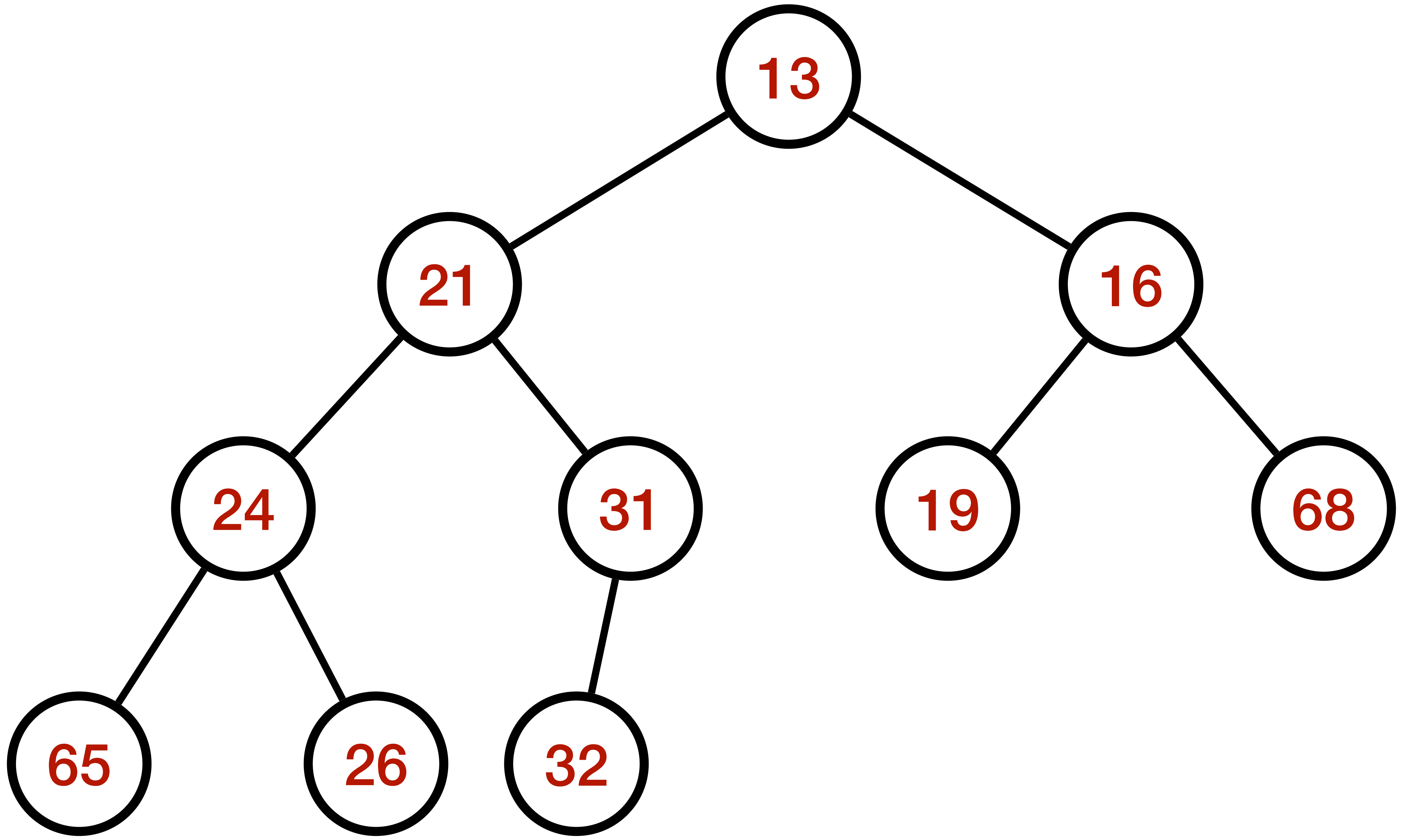
Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:
  - Read **only** k elements in an array
  - Sort the array
  - The smallest is at k^{th} position. For the remaining elements:
    - Compare with the k^{th} element —> if the incoming element is larger then replace it with the k^{th} element

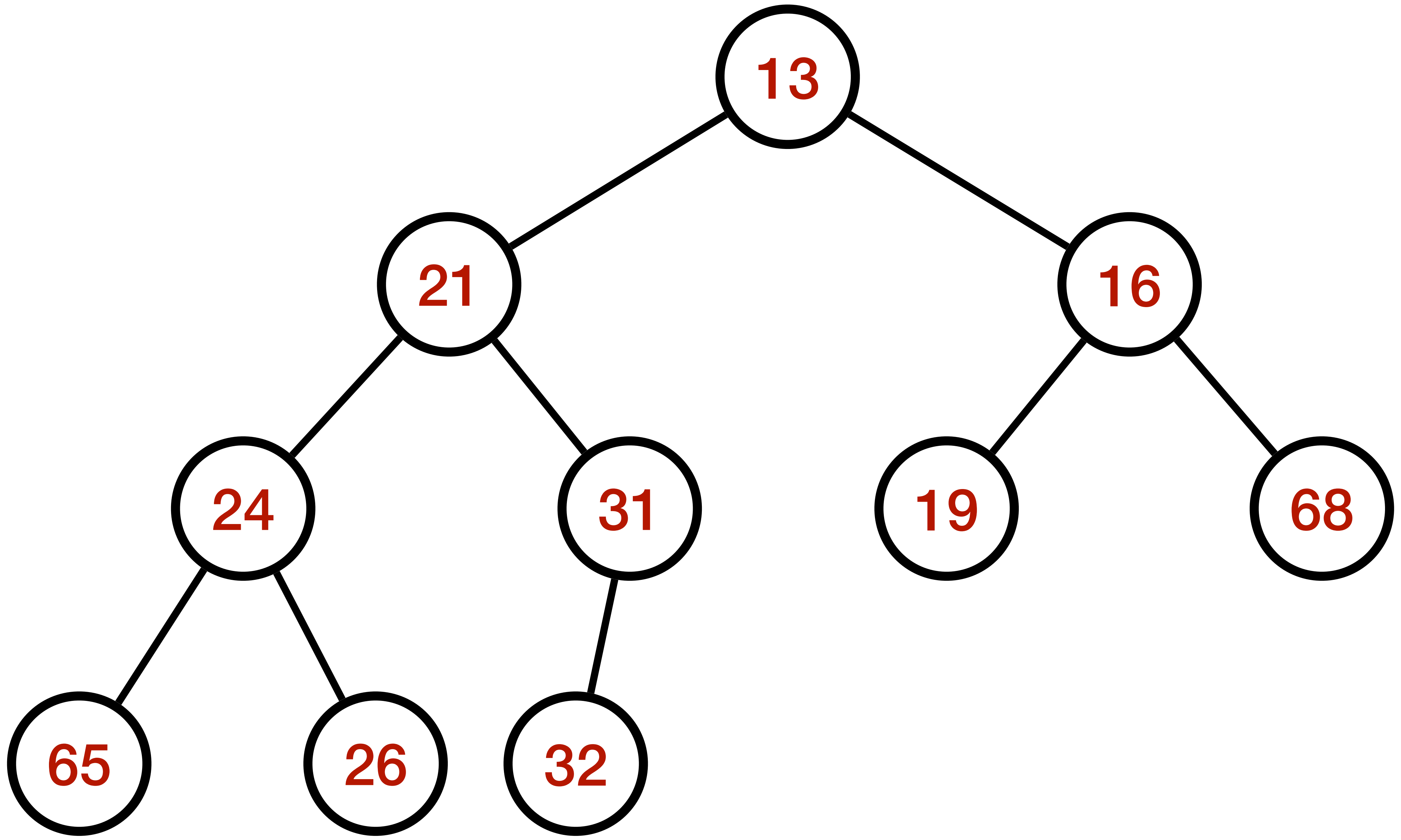


Applications: The Selection Problem

- Problem: Find the k^{th} largest element in a list of n elements
- Algorithm1B:
  - Read **only** k elements in an array
  - Sort the array
  - The smallest is at k^{th} position. For the remaining elements:
    - Compare with the k^{th} element —> if the incoming element is larger then replace it with the k^{th} element
  - Time complexity: ?

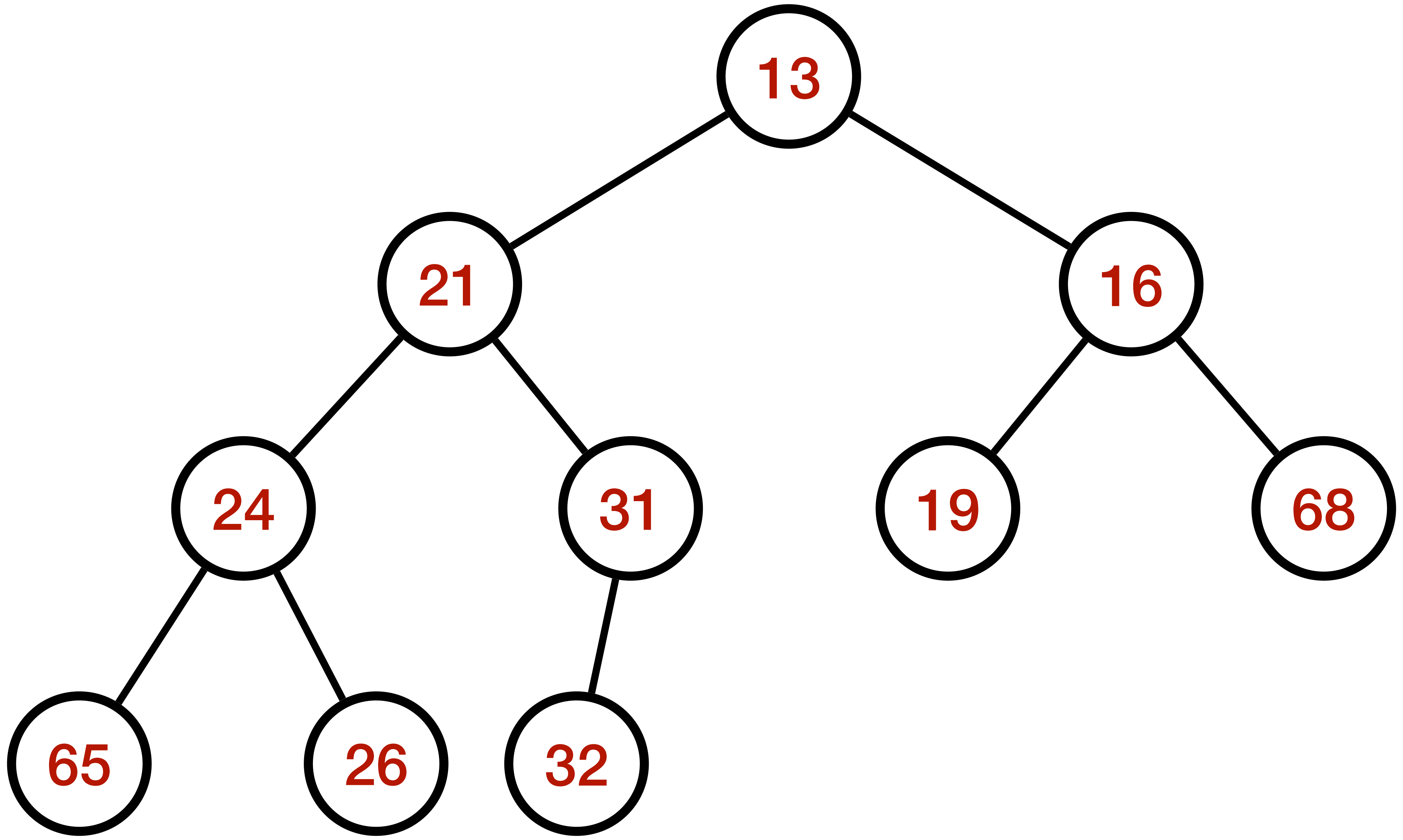


Applications: The Selection Problem



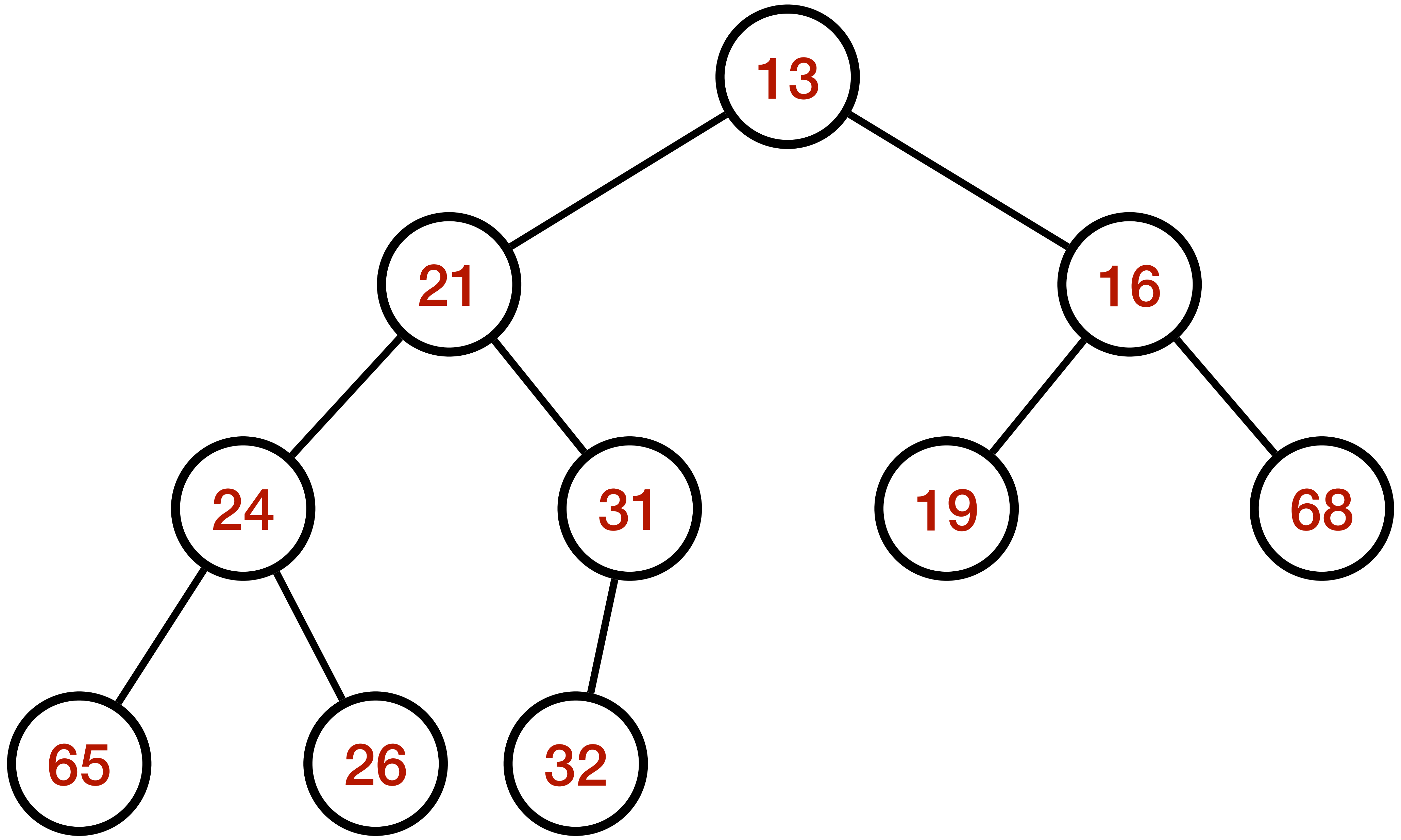
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements



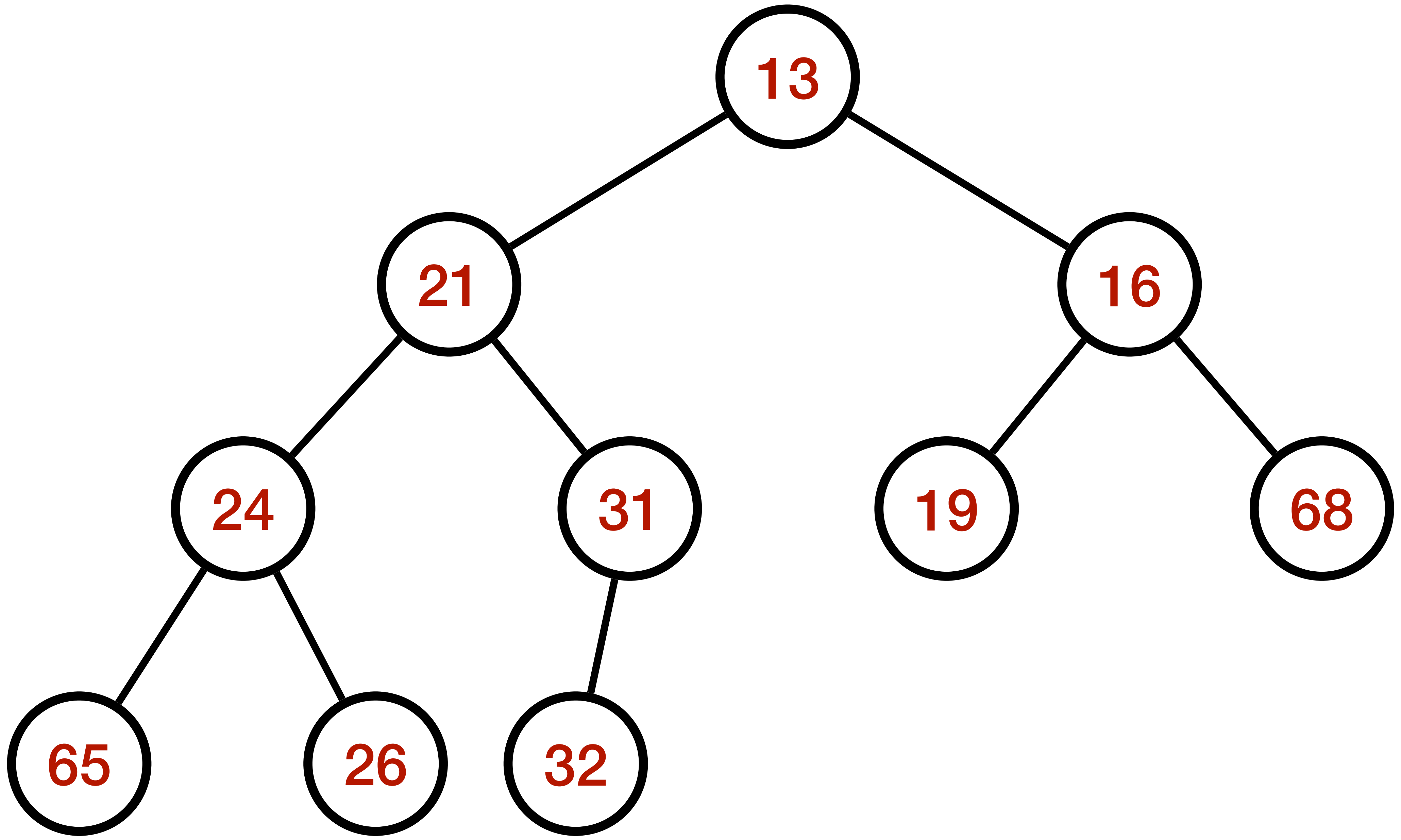
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:



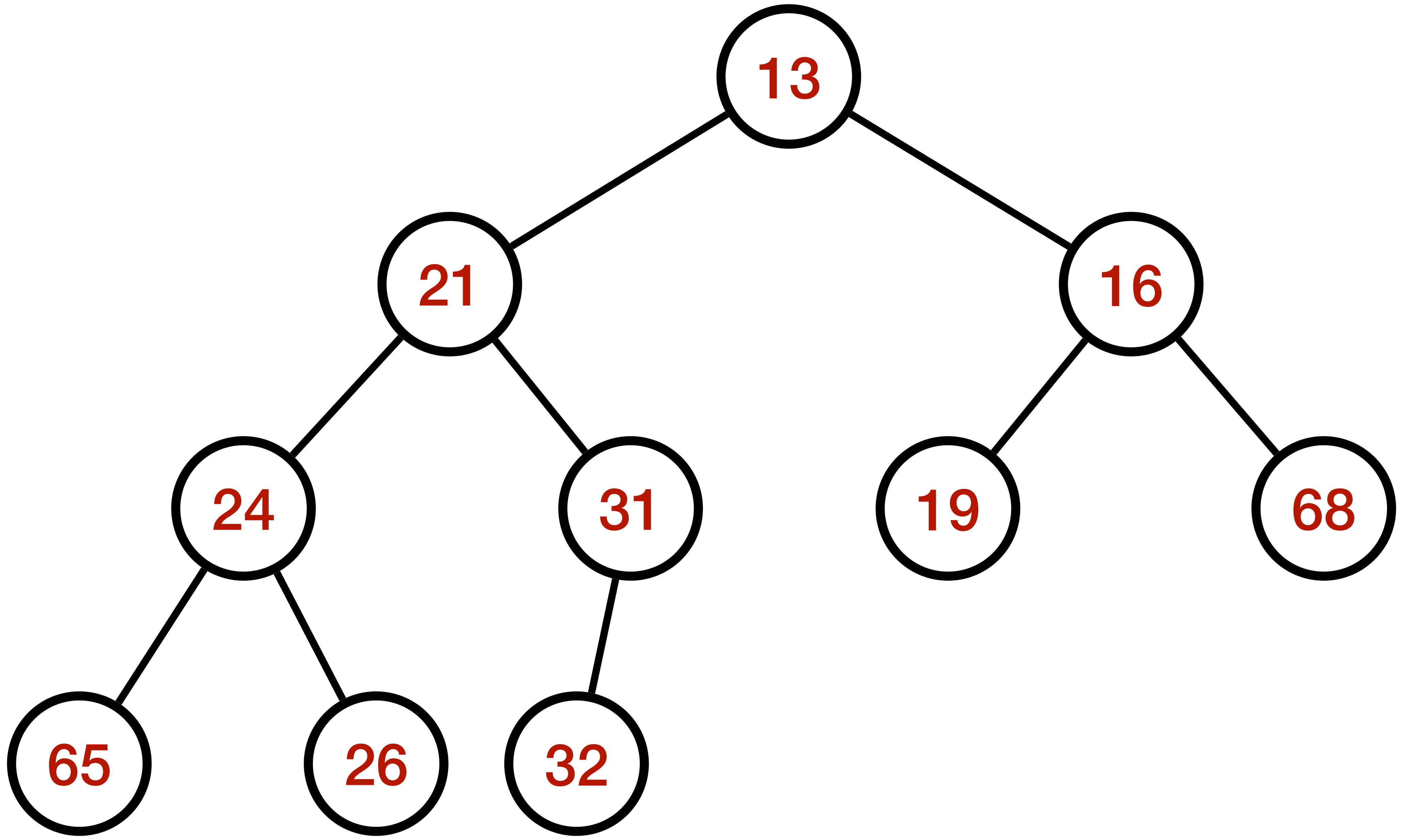
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array



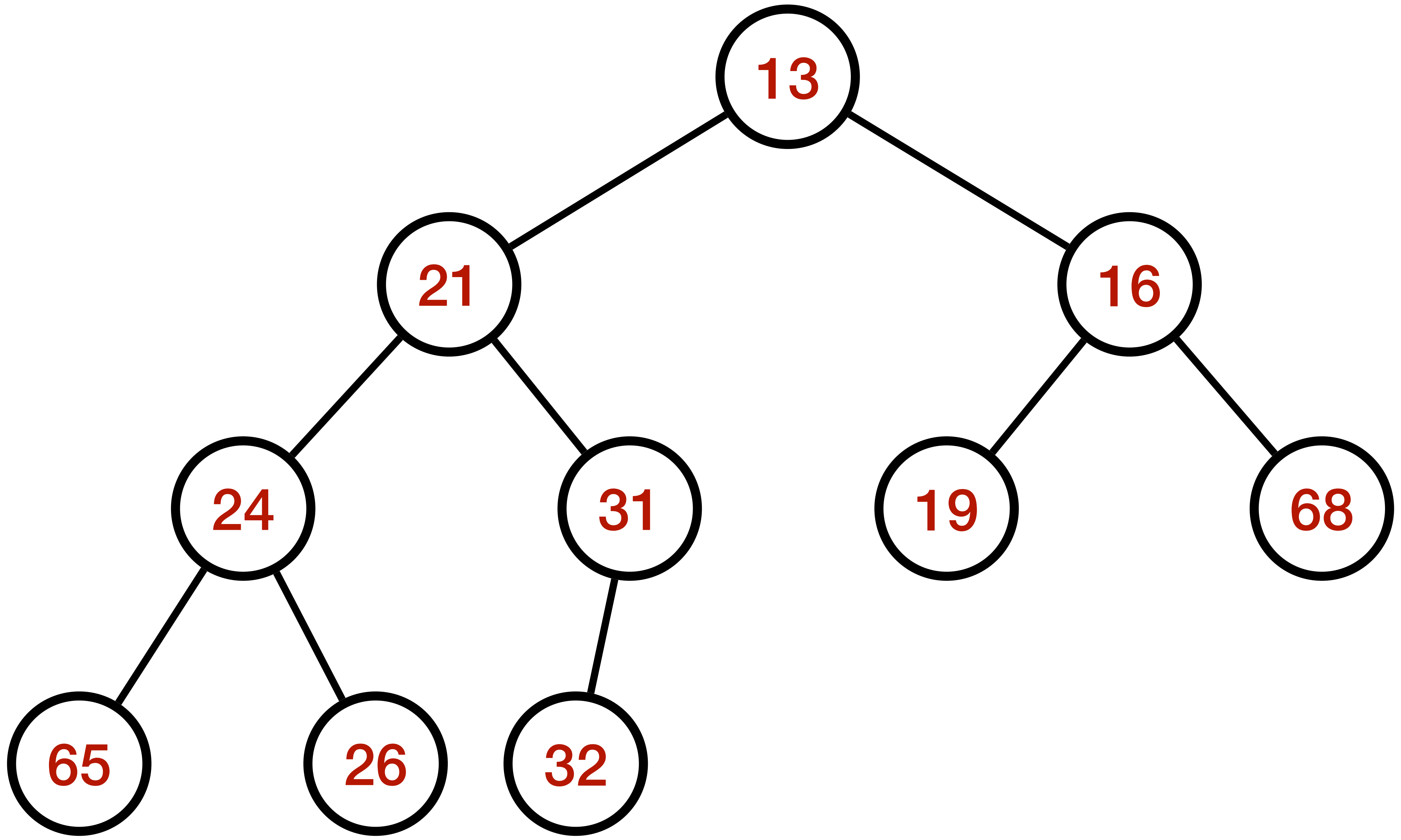
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array
  - Apply BuildHeap



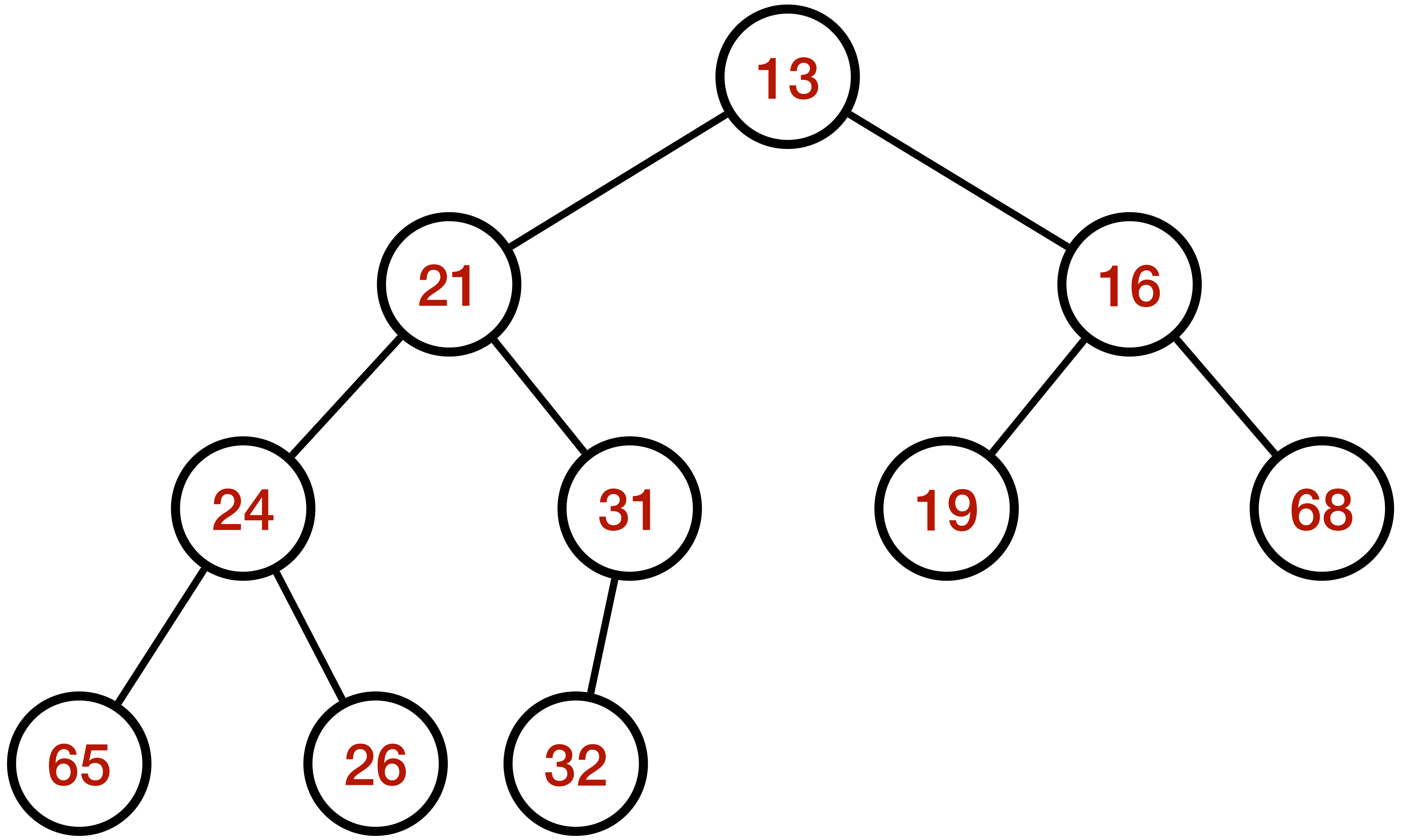
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array
  - Apply BuildHeap
  - Apply k DeleteMin operations



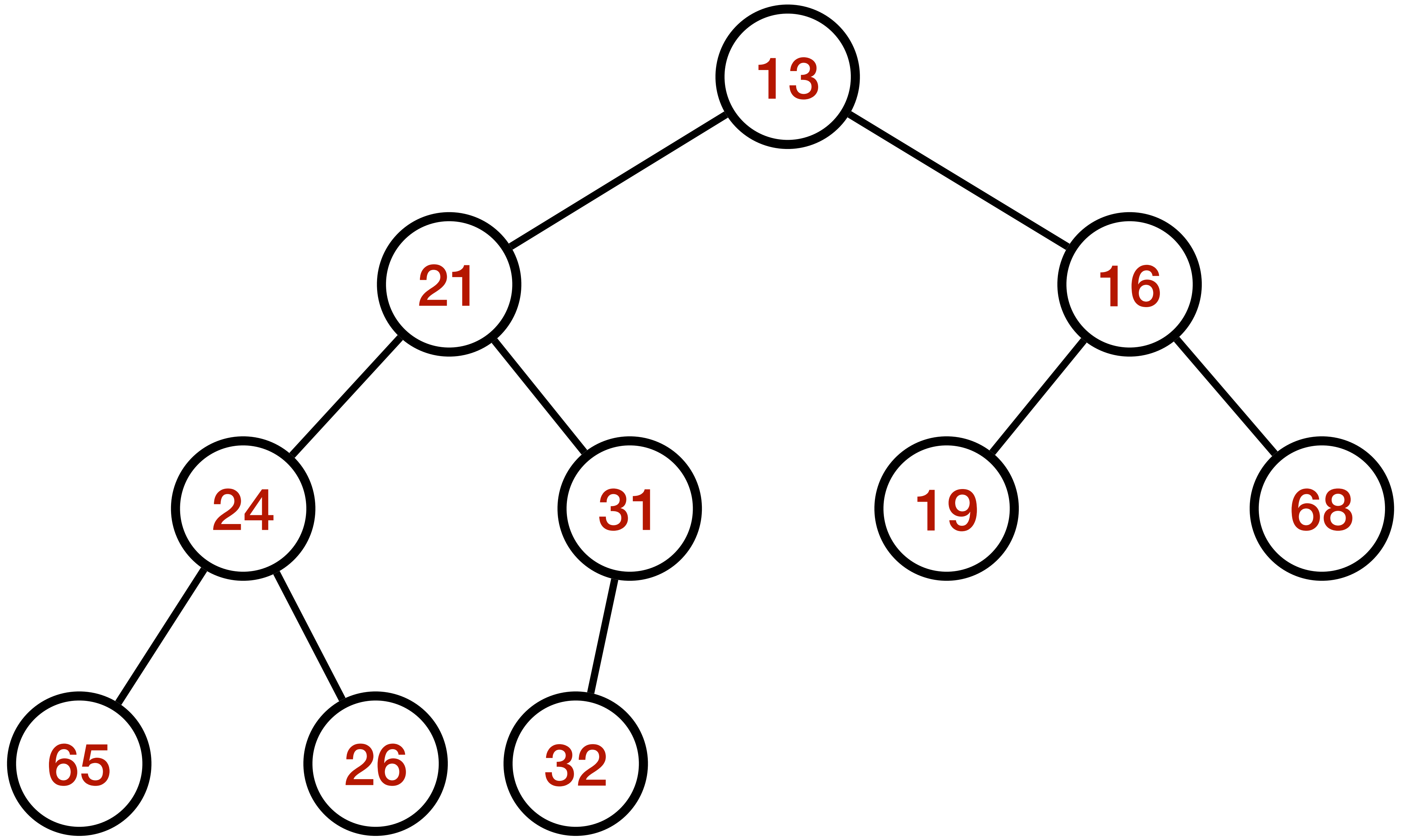
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array
  - Apply BuildHeap
  - Apply k DeleteMin operations
    - The last extracted element is our answer



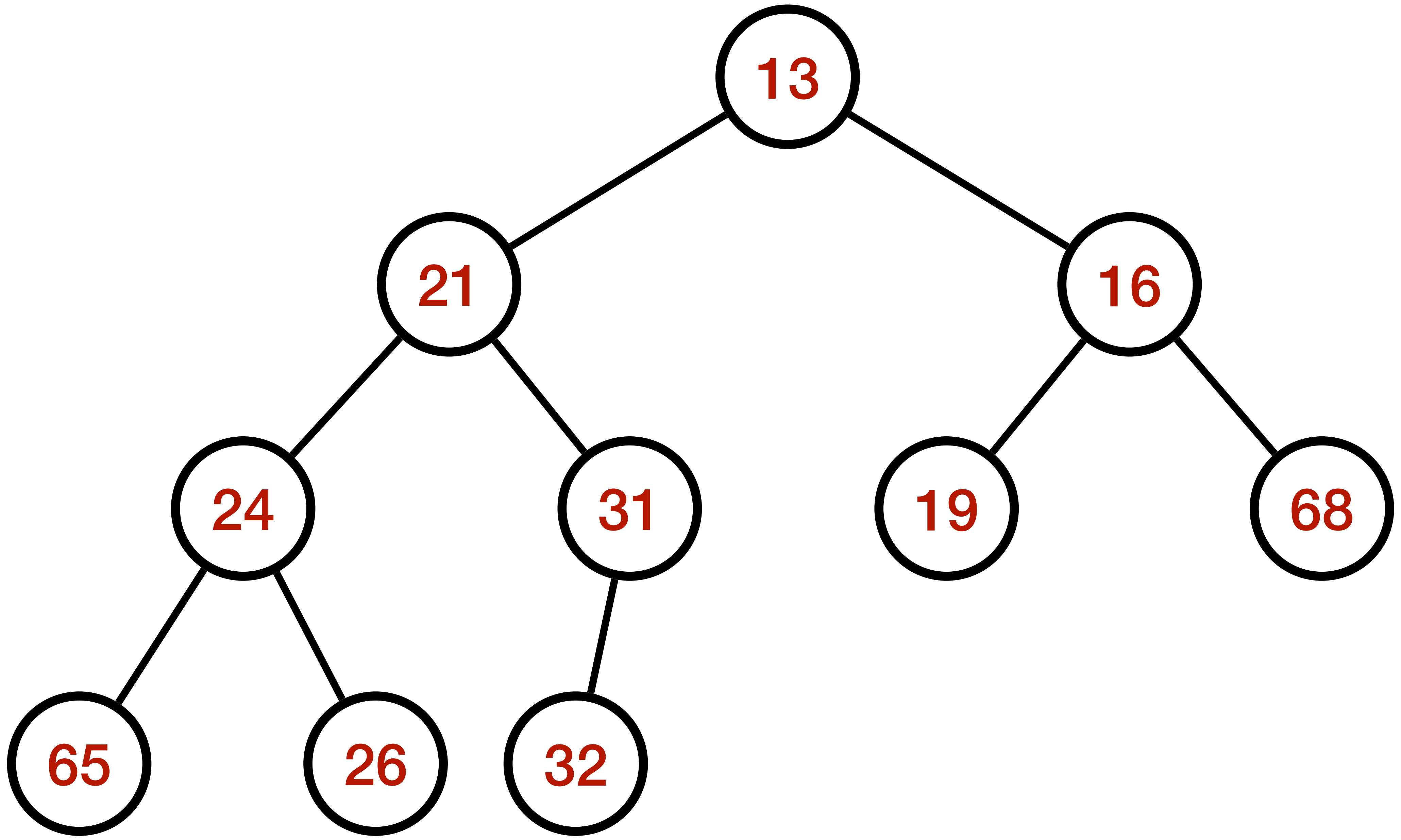
Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array
  - Apply BuildHeap
  - Apply k DeleteMin operations
    - The last extracted element is our answer
  - Time complexity: O(n + k log n). Why?

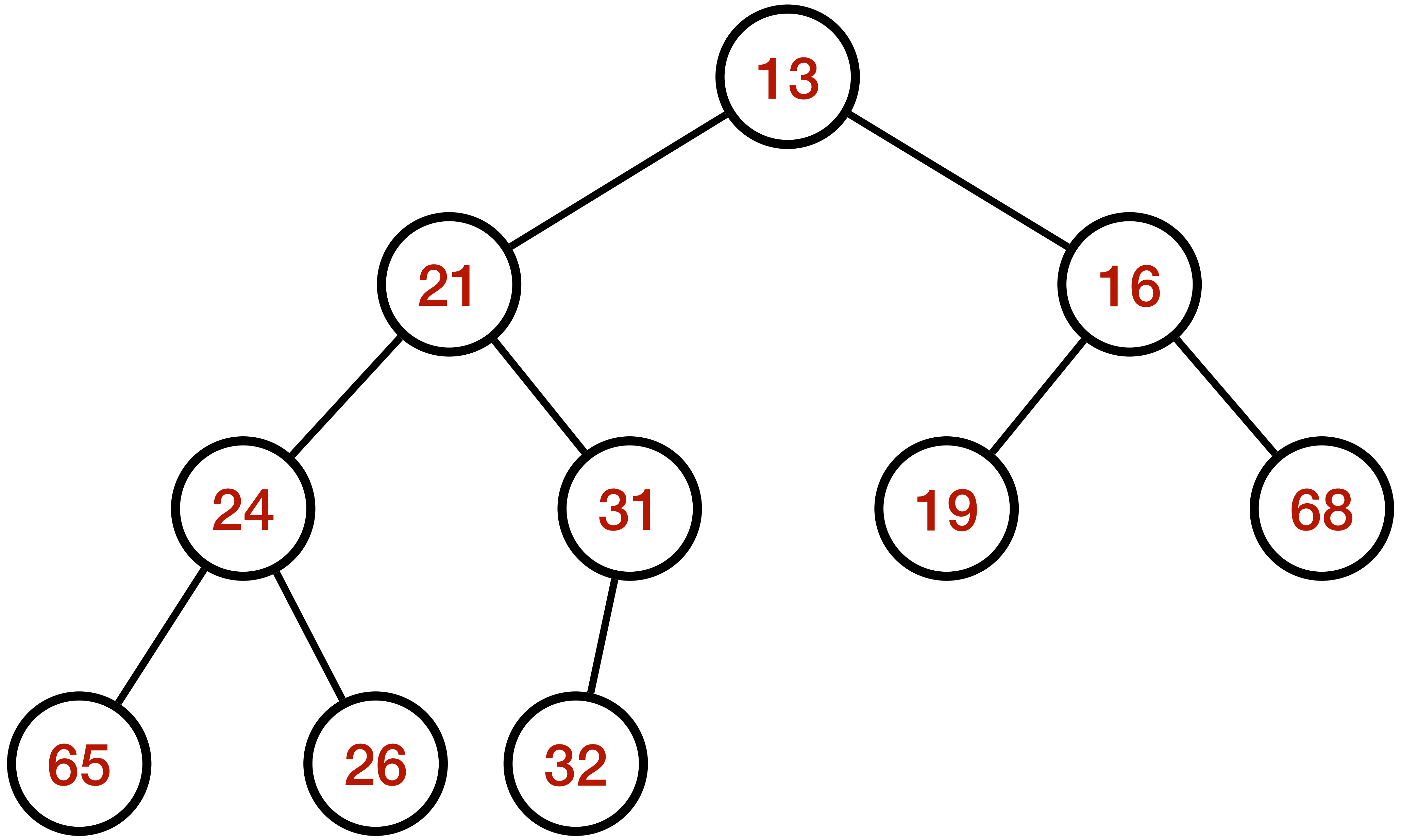


Applications: The Selection Problem

- Changed Problem: Find the k^{th} **smallest** element in a list of n elements
- Algorithm2A:
  - Read elements in an array
  - Apply BuildHeap
  - Apply k DeleteMin operations
    - The last extracted element is our answer
  - Time complexity: O(n + k log n). Why?
    - What happens when $k = \lceil n/2 \rceil$ or when $k = O(n/log\, n)$?

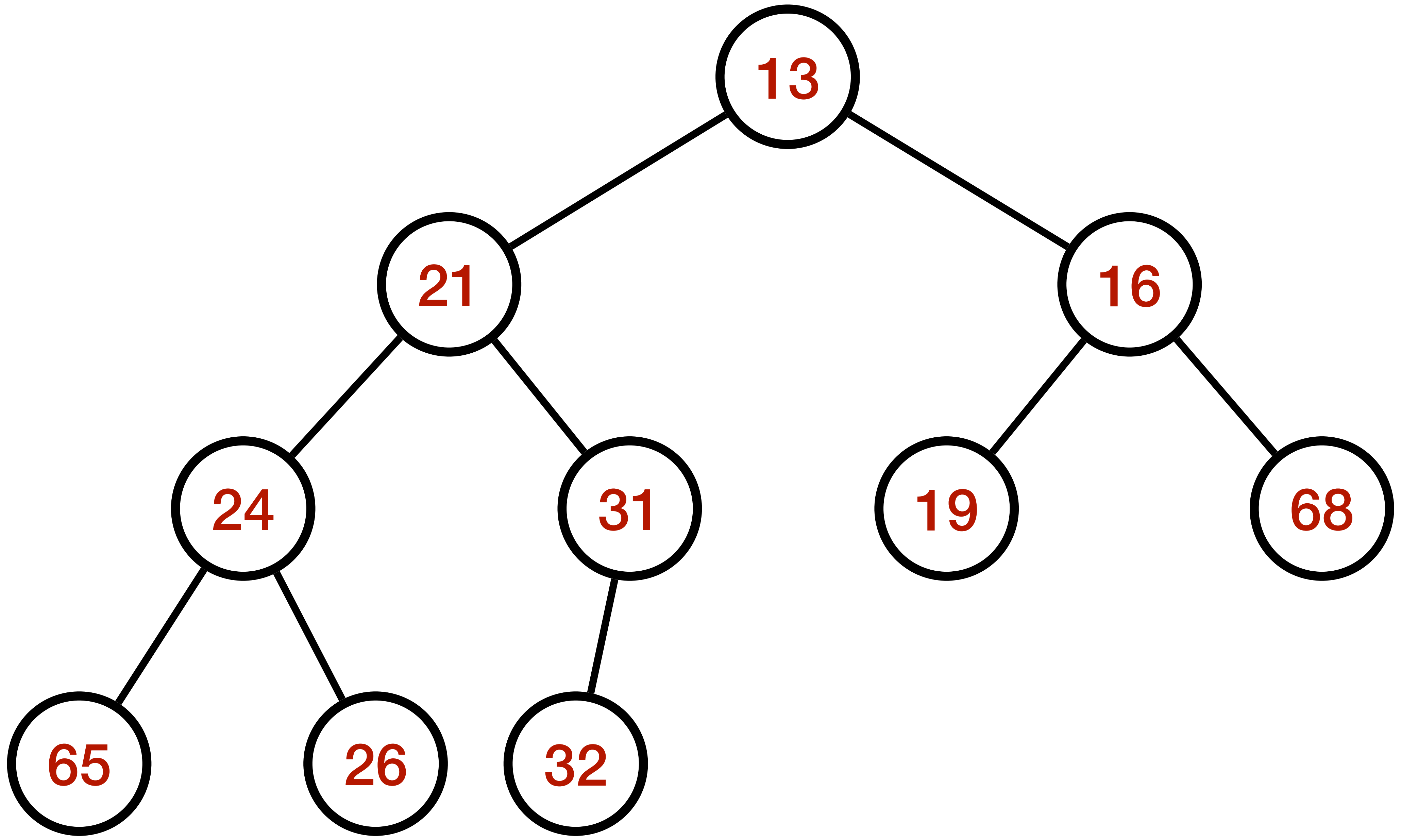


Applications: The Selection Problem



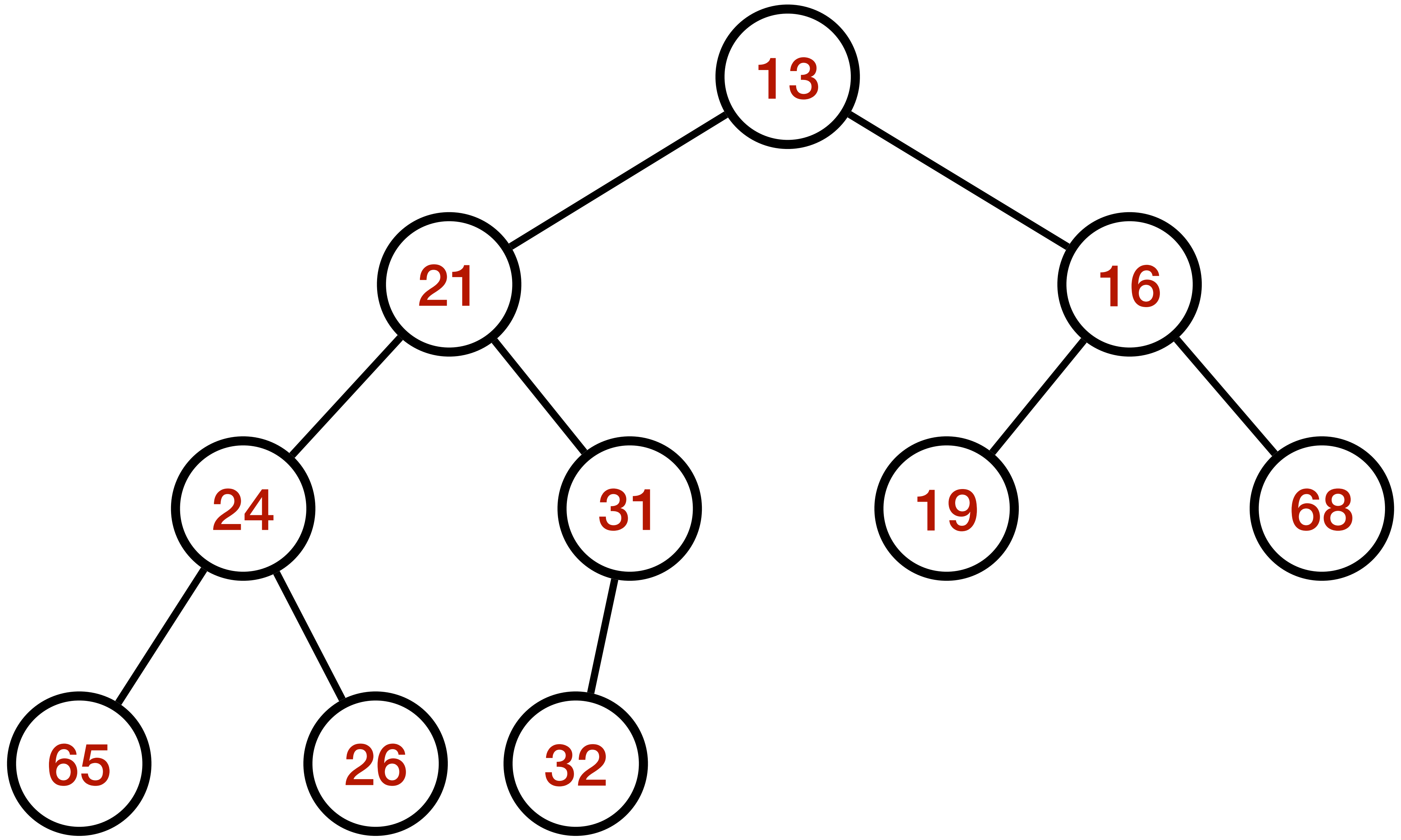
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements



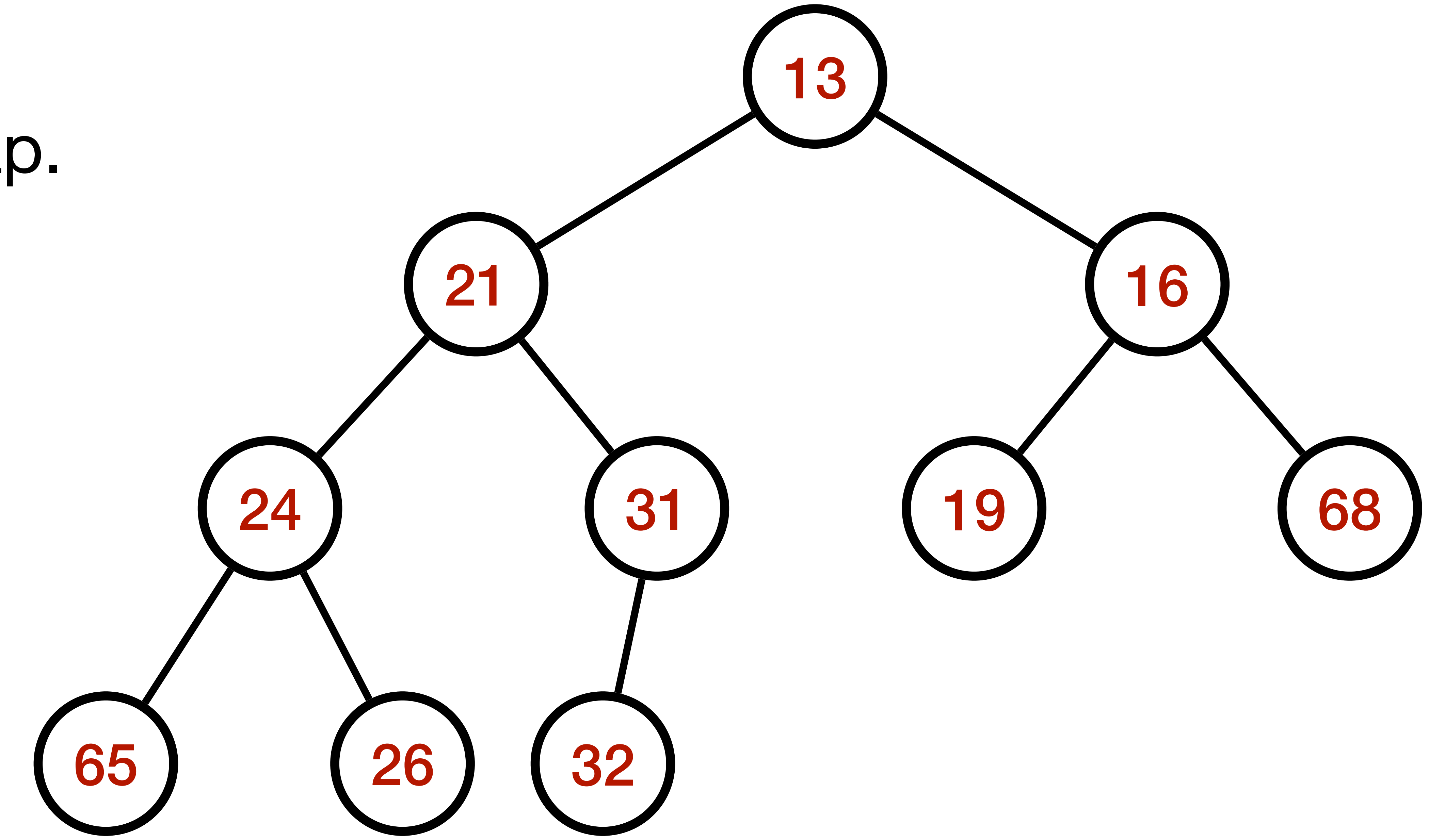
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:



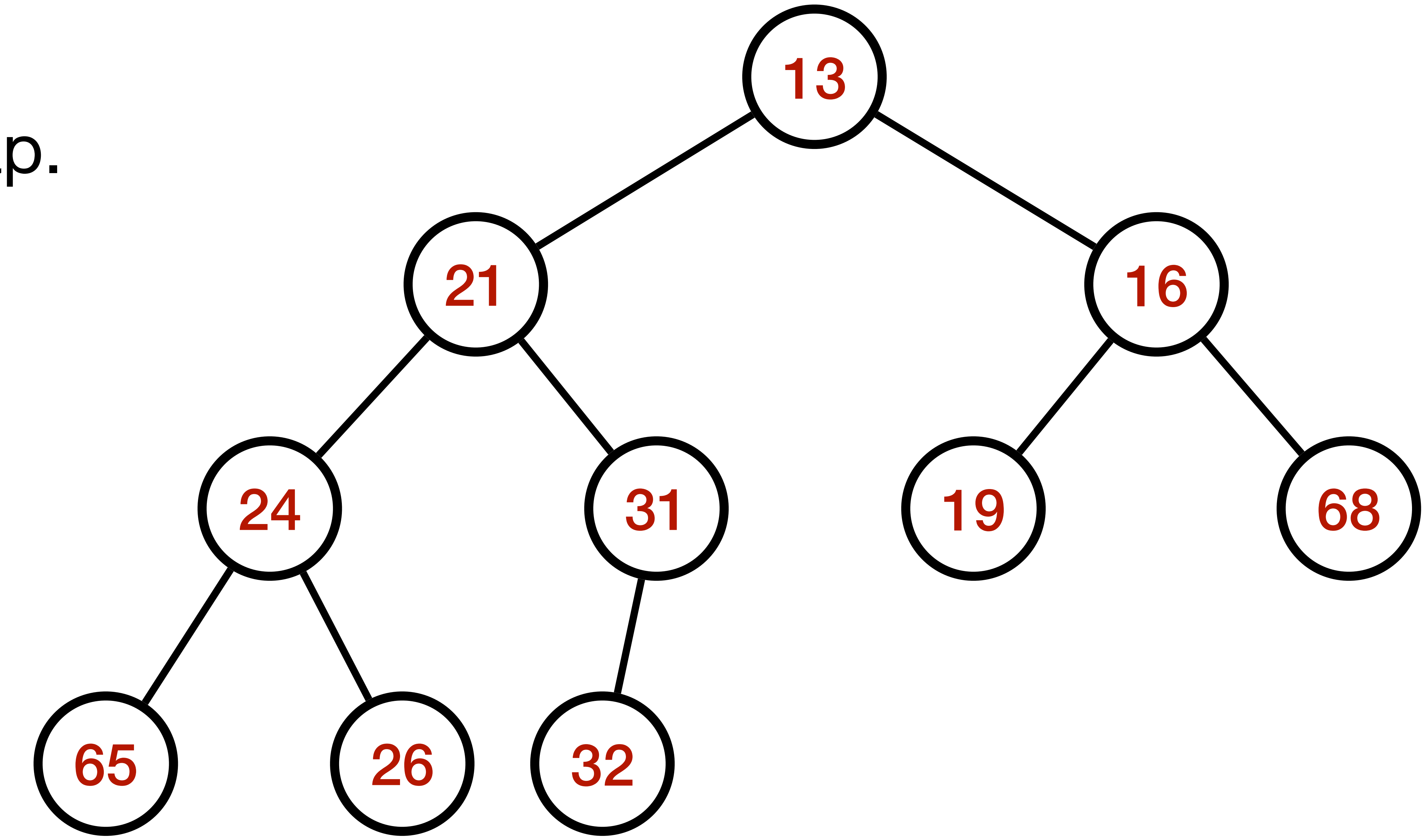
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.



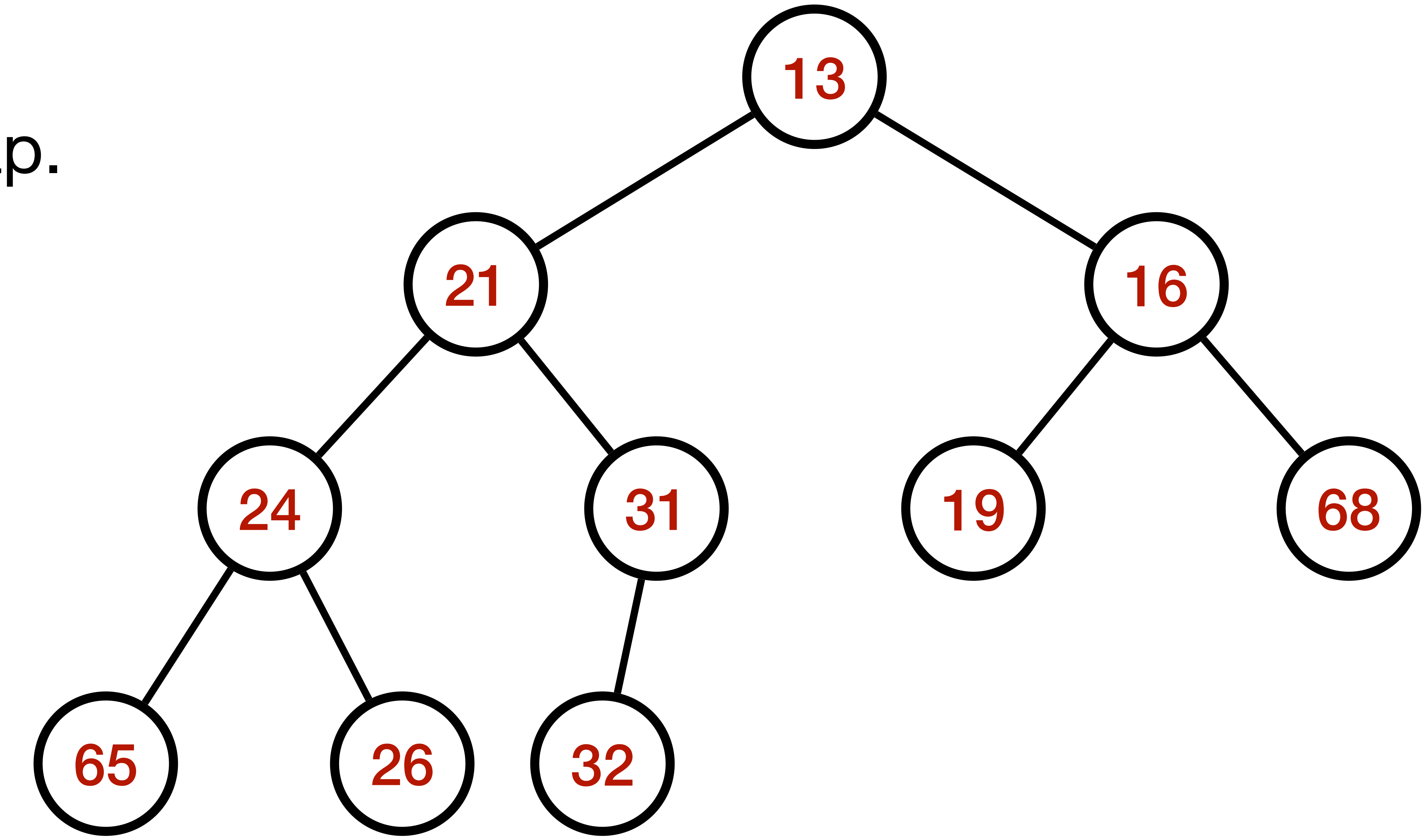
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.
  - New element is compared with the k^{th} largest



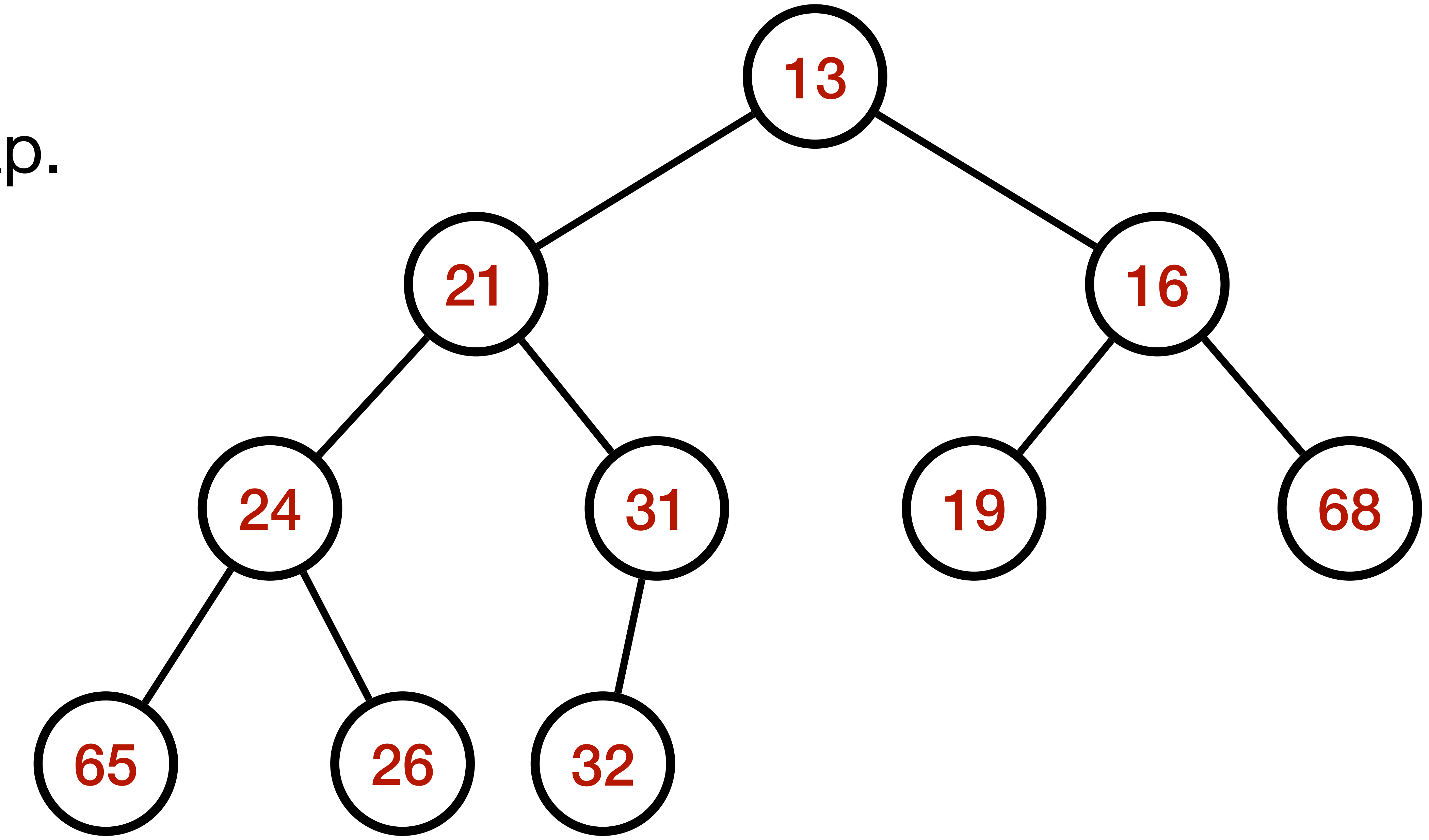
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.
  - New element is compared with the k^{th} largest
  - If the new element is larger, it replaces the root



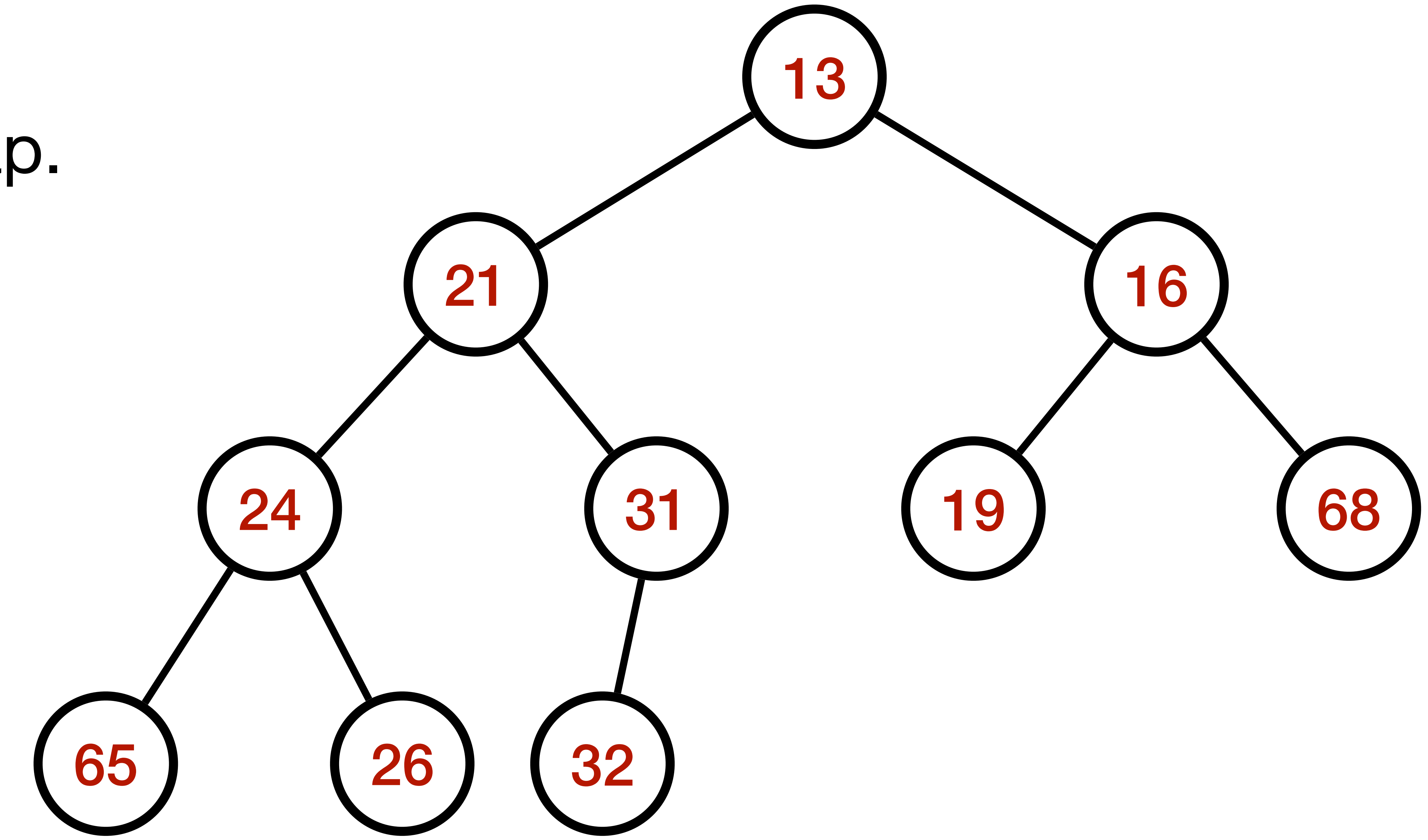
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.
  - New element is compared with the k^{th} largest
  - If the new element is larger, it replaces the root
  - At the end of the input, we return the root.



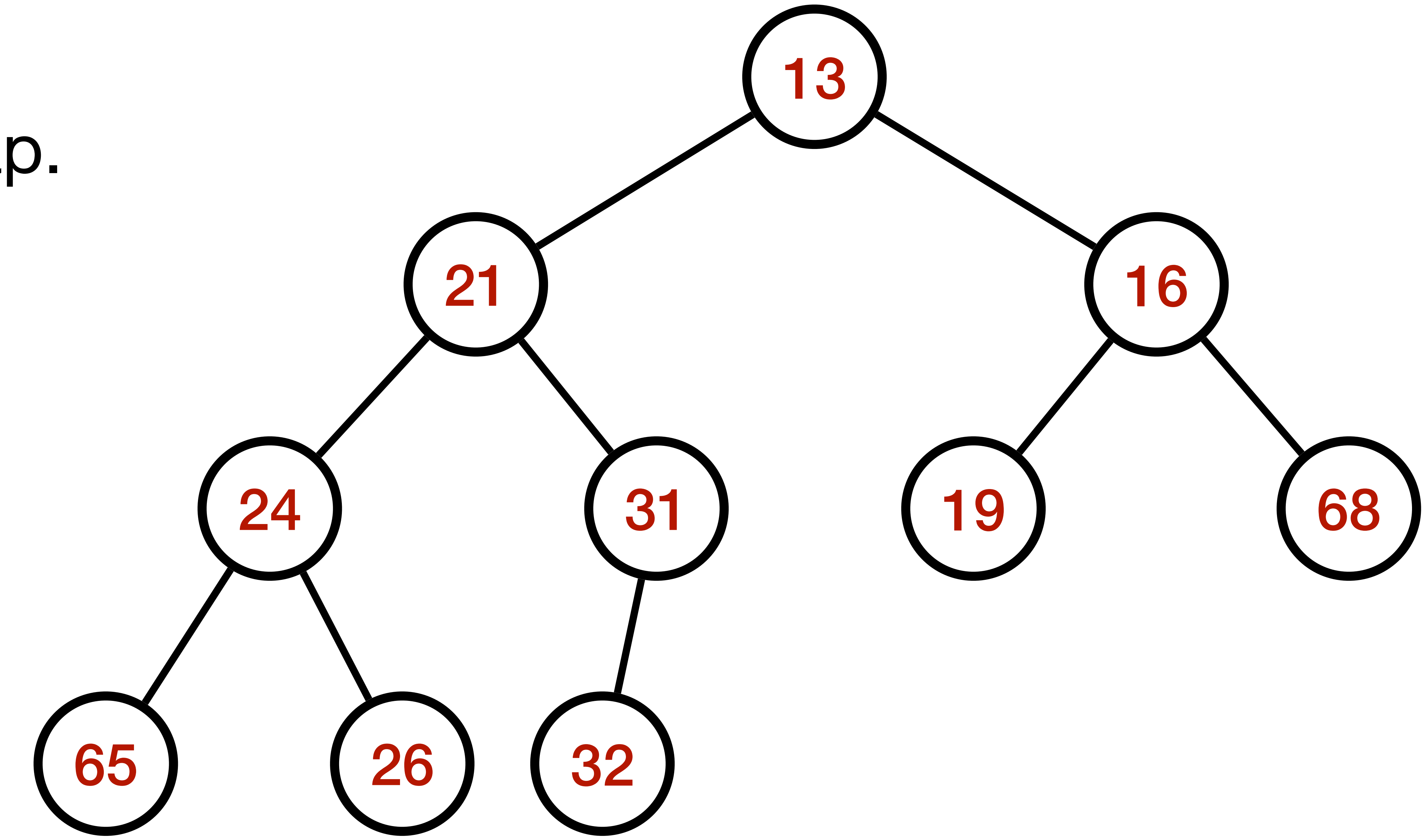
Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.
  - New element is compared with the k^{th} largest
  - If the new element is larger, it replaces the root
  - At the end of the input, we return the root.
  - Time complexity: O(k + (n-k).log k)



Applications: The Selection Problem

- Problem: Find the k^{th} **largest** element in a list of n elements
- Algorithm2B:
  - Read **only** k elements in an array and build a minheap.
  - New element is compared with the k^{th} largest
  - If the new element is larger, it replaces the root
  - At the end of the input, we return the root.
  - Time complexity: O(k + (n-k).log k)
  - Can we do better? Quickselect O(n) average time!



Extra Reading

For those who want to challenge themselves

- Skew Heaps (efficient merge operations)
- Binomial Queues
- Fibonacci Heaps
- Find the median of an array efficiently
- Understand Quick-select