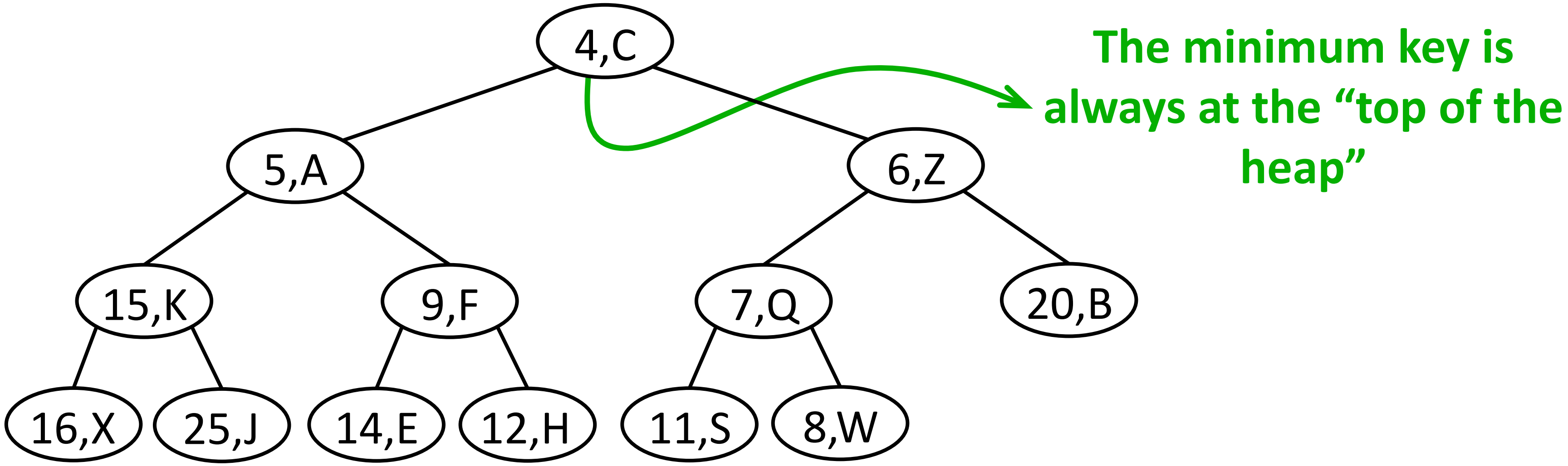
34

HEAPS

THE HEAP DATA STRUCTURE

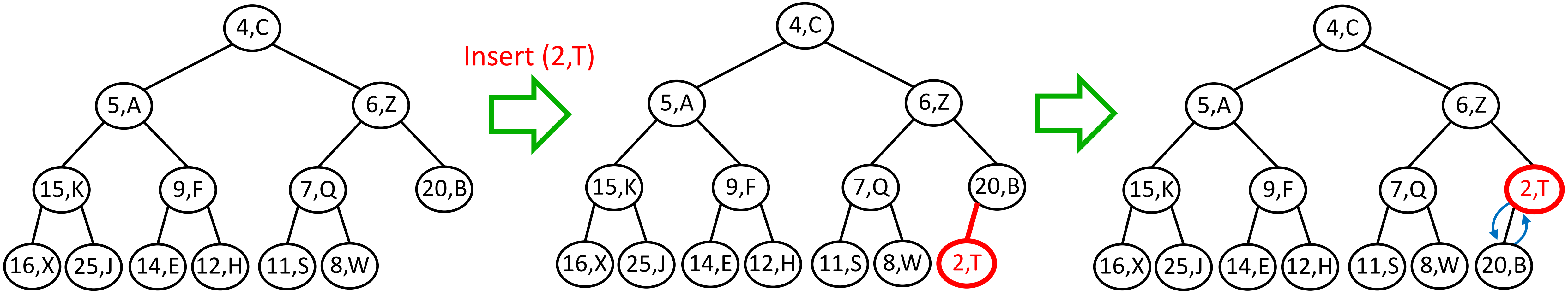
- A **heap** is a **complete binary tree** that stores a collection of elements with their associated keys at its nodes, and that satisfies the following property:
  - **Heap-Order Property:** For every node v other than the root:
    (key associated with v) $\geq$ (key associated with v's parent)

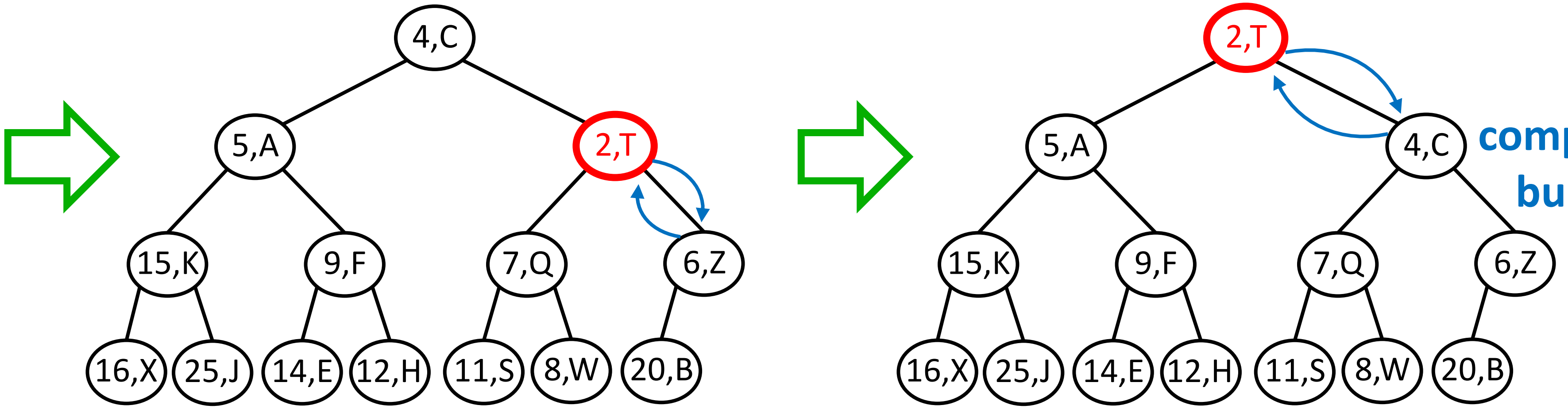


- To implement a priority queue using a heap, we'll need a **comparator**.

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- To **insert** an element to a heap:(1) add the element **at the end** (2) perform "**up-heap bubbling**"
  We need to ensure that the Heap-Order property is preserved!



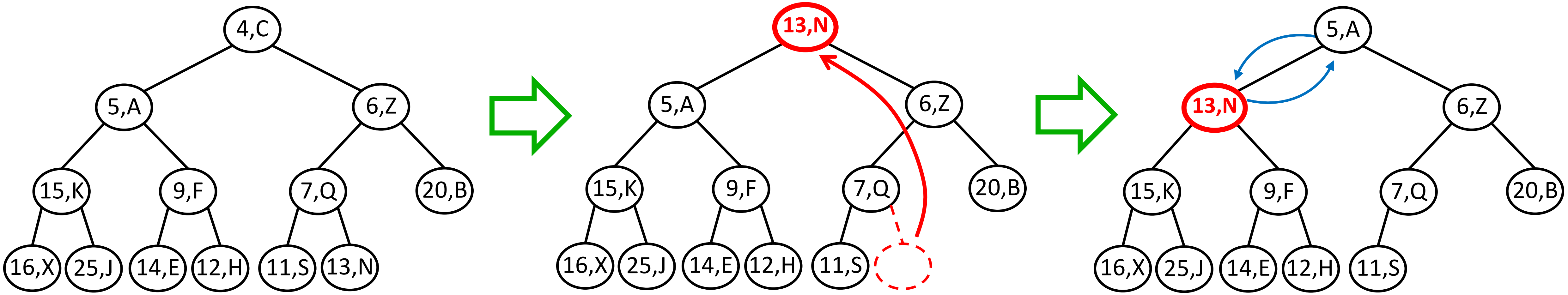


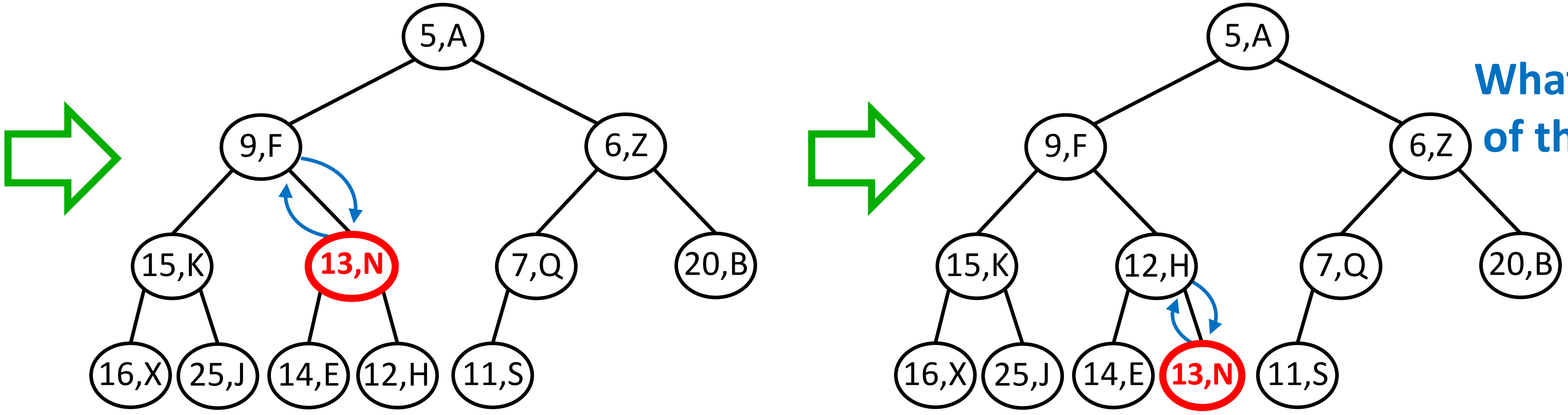
What is the time complexity of the up-heap bubbling of one node?

O(h) -> O(log n)

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- To **remove** (removeMin) an element (with the minimum key): (1) remove the **root** (2) set the **last node as the new root** (3) perform "**down-heap bubbling**"





What is the time complexity of the down-heap bubbling of one node?

O(h) -> O(log n)

HEAP: C++ IMPLEMENTATION

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
    int size() const; // number of elements
    bool empty() const; // is the heap empty?
    void insert(const E& e); // insert element
    const E& min(); // minimum element
    void removeMin();  // remove minimum
private:
    VectorCompleteTree<E> T;
    C isLess; // less-than comparator
    typedef typename VectorCompleteTree<E>::Position Position;
    /* This way, we can simply write "Position" instead of writing
       the long definition */
};
```

```
template <typename E, typename C>
void HeapPriorityQueue<E,C>::insert(const E& e) {
    T.add(e); // add e to heap
    Position v = T.last(); // v is now the position of e
    while (!T.isRoot(v)) {  // up-heap bubbling
        Position u = T.parent(v);
        if (!isLess(*v, *u)) break; // if v in order, we're done
        T.swap(v, u); // . . . else, swap with parent
        v = u; // v is now the NEW position of e
    }
}
```

HEAP: C++ IMPLEMENTATION

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
    int size() const; // number of elements
    bool empty() const; // is the heap empty?
    void insert (const E& e); // insert element
    const E& min(); // minimum element
    void removeMin();  // remove minimum
private:
    VectorCompleteTree<E> T; // complete binary tree
    C isLess; // less-than comparator
    typedef typename VectorCompleteTree<E>::Position Position;
    /* This way, we can simply write “Position” instead of writing
       the long definition */
};
```

```
template <typename E, typename C>
const E& HeapPriorityQueue<E,C>::min()
{  return *(T.root());  }
```

HEAP: C++ IMPLEMENTATION

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
    int size() const; // number of elements
    bool empty() const; // is the heap empty?
    void insert (const E& e); // insert element
    const E& min(); // minimum element
    void removeMin(); // remove minimum
private:
    VectorCompleteTree<E> T;
    C isLess; // less-than comparator
    typedef typename VectorCompleteTree<E>::Position Position;
    /* This way, we can simply write "Position" instead of writing
       the long definition */
};
```

```
template <typename E, typename C>
void HeapPriorityQueue<E,C>::removeMin() {
    if (size() == 1) // If we have only one node
        T.remove();   // remove it
    else {
        Position u = T.root();
        T.swap(u, T.last()); // swap last with root
        T.remove(); // . . .and remove last
        while (T.hasLeft(u)) {  // down-heap bubbling
            Position v = T.left(u);
            if (T.hasRight(u) && isLess(*(T.right(u)), *v))
                v = T.right(u); // v is u's smaller child
            if (isLess(*v, *u)) { // is u out of order?
                T.swap(u, v); // . . .then swap
                u = v;
            }else break; // else we're done
        }
    }
}
```

HEAP SORT

- If you recall, given a list L of n elements, we can sort L using a priority queue Q as follows:
  - **Phase 1:** Put the elements of L into a priority queue P using n `insert(e)` operations
  - **Phase 2:** Extract the elements from P in **an ascending order** using n combinations of `min()` and `removeMin()` operations
- We said that this scheme depends on how `insert(e)`, `min()` and `removeMin()` are implemented. Now, using a heap:
  - Every insertion operation with **up-heap bubbling** takes $O(\log n)$ time
  - Every removal operation with **down-heap bubbling** takes $O(\log n)$ time
- Thus, each of the above takes $O(\log n)$, implying that the entire process of sorting n elements takes $O(n \log n)$ time. This algorithm, called **Heap sort**, is faster than **Selection sort** and **Insertion sort** (they run in $O(n^2)$ time)

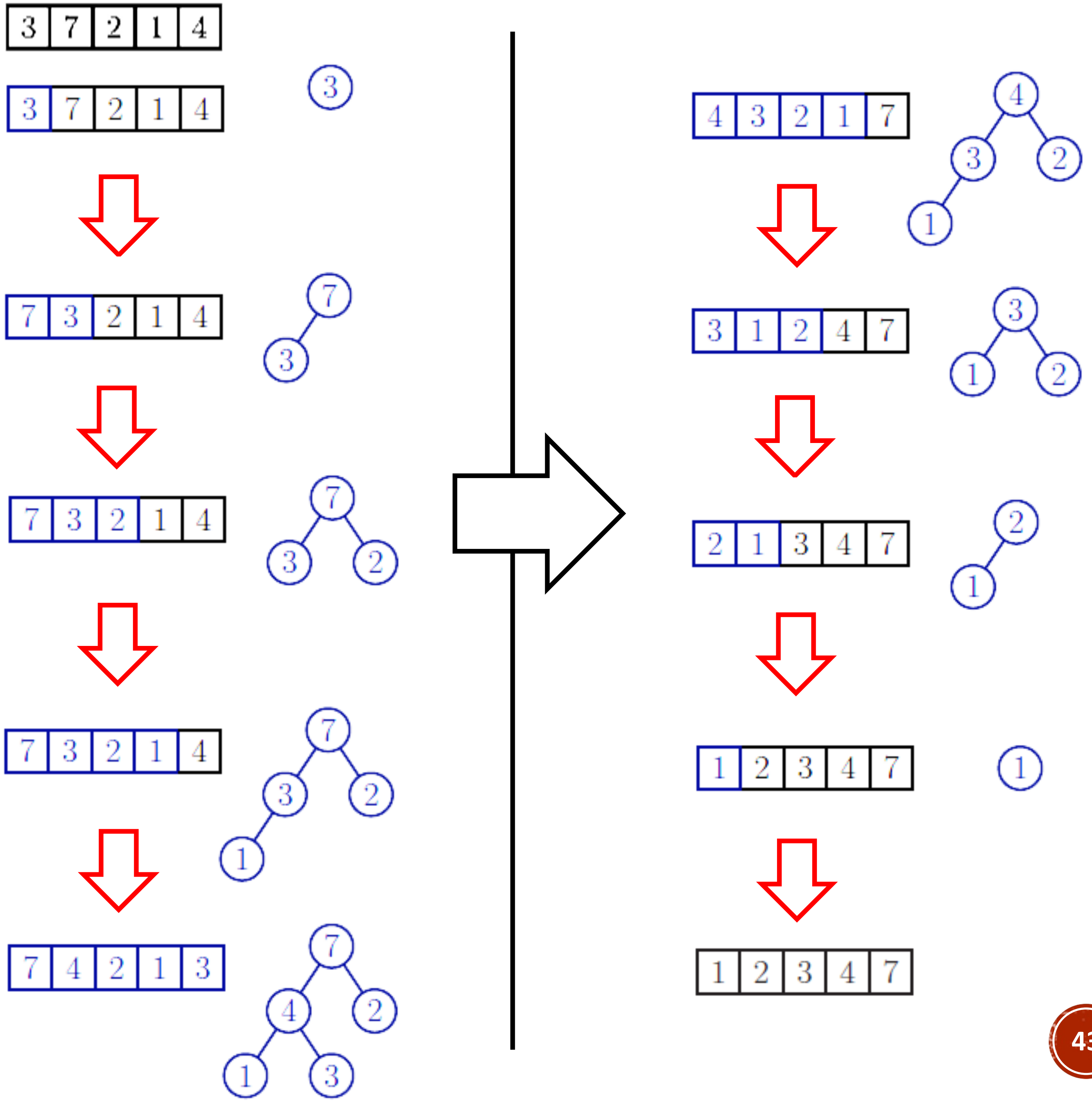
IMPLEMENTING HEAP-SORT "IN-PLACE"

- If you recall, given a list L of n elements, we can sort L using a priority queue Q
- Note that the above scheme requires both L and Q. *To save memory space, we will show how to apply the same scheme, but using only L without Q*
- Such a technique is called "**in-place**", since it does not require the use of a temporary memory space (such as Q)

  → All the rearrangements are done in the list itself, i.e., "**in-place**"
- More specifically, if L is implemented as an array, we can reduce its space requirement *using a portion of L itself to store the heap*, thus avoiding the use of an external heap data structure

EXAMPLE

- Here, we use a reverse comparator, which results in the **largest** key being at the top of the heap.
- First, we scan the array **from left to right,** and insert the elements into the heap (*the blue part represents the heap, while the black part represents the elements that are not yet scanned*)
- Then, we repeat the process of removing the top of the heap and placing it in the array **from right to left**



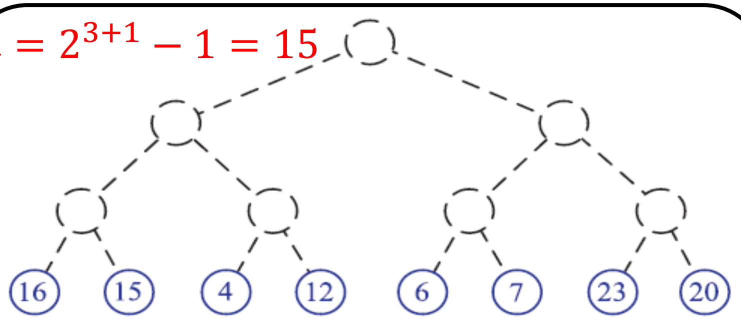
BOTTOM-UP HEAP CONSTRUCTION

- As mentioned earlier, **inserting a single element** into a heap takes $O(\log n)$
  - Thus, **inserting** n **elements** takes $O(n \log n)$
- However, it turns out that inserting n elements **can be done in just** $O(n)$! ***How?***
- This is done by "***bottom-up heap construction***", where:
  - we start by **filling the bottom level** of the heap,
  - then we **fill the level above it**
  - . . .
  - **Until we reach the top** of the heap (i.e., the root)

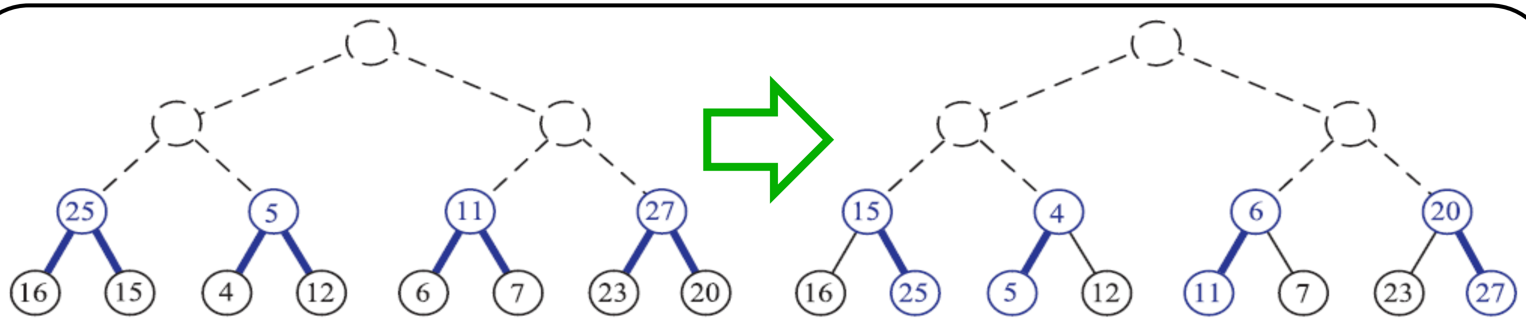
If all the elements to be stored in the heap are given in advance!

BOTTOM-UP HEAP CONSTRUCTION

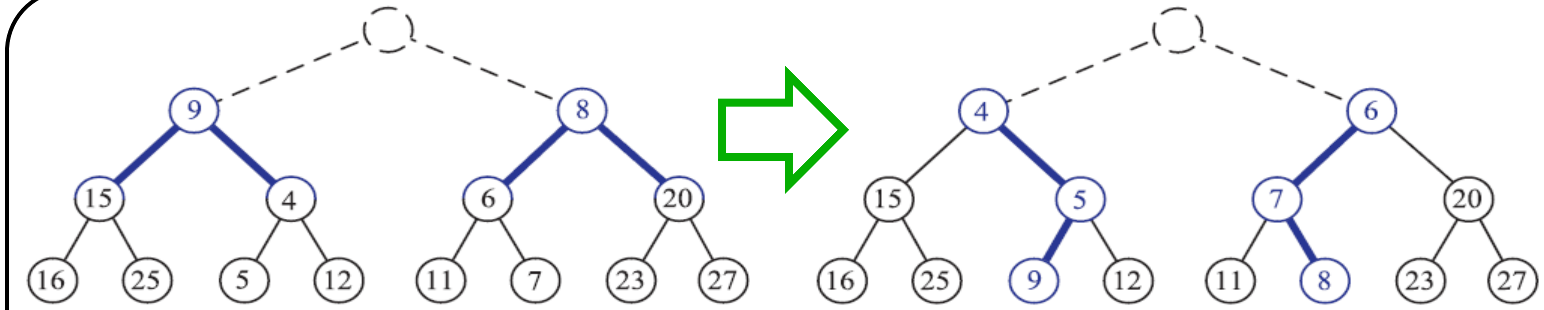
$n = 2^{3+1} - 1 = 15$



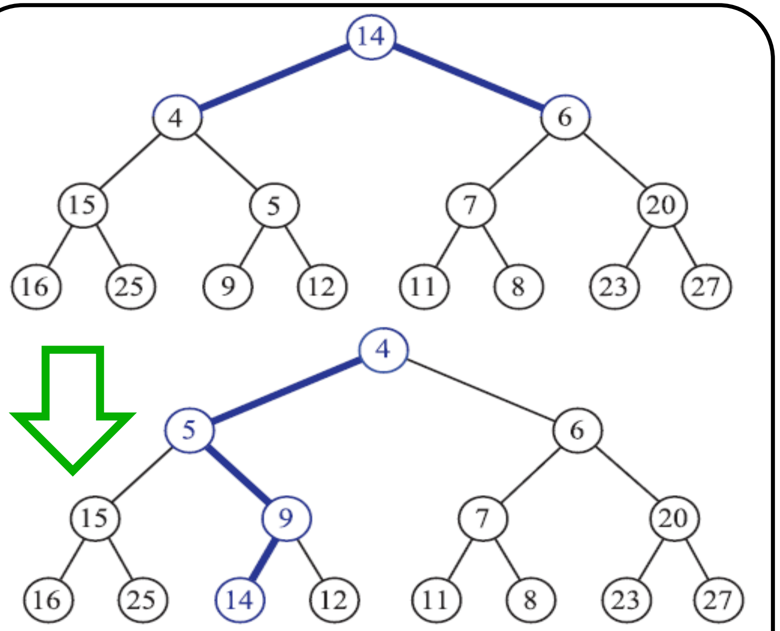
Step 1: Construct $(n+1)/2$ heaps containing a single element each



Step 2: For each pair of heaps, add an element that would join them into a single heap, then perform "down-heap bubbling"



Step 3: For each pair of heaps, add an element that would join them into a single heap, then perform "down-heap bubbling"



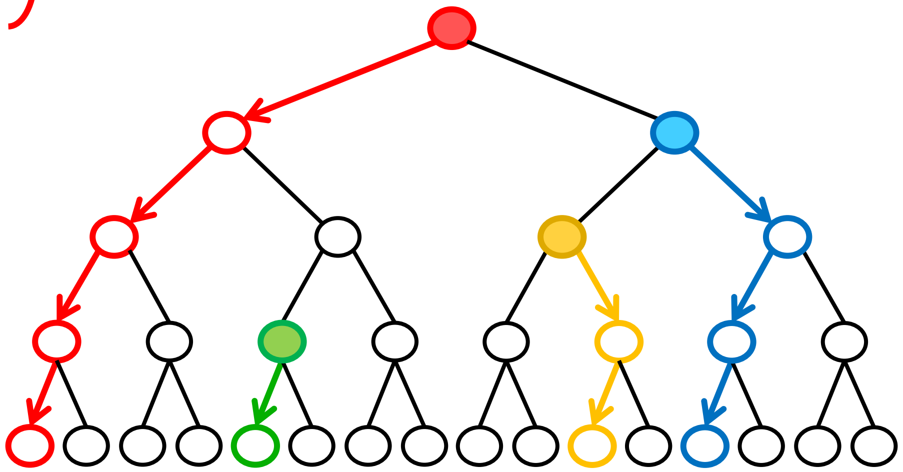
Step 4: Construct the final heap

Bottom-up heap construction with n entries takes **O(n)** time, assuming two keys can be compared in O(1) time!

COMPLEXITY OF BOTTOM-UP HEAP CONSTRUCTION

- If we insert all n elements in the heap, e.g., given n = 31, the **max No. of operations** for down-heap bubbling is:
  - 0 for each of the 16 node in level 4
  - 1 for each of the 8 nodes in level 3
  - 2 for each of the 4 nodes in level 2
  - 3 for each of the 2 nodes in level 1
  - 4 for the root

Therefore, the total is O(*n*); we are inserting *n* elements and only one of them is the root!



A RECURSIVE ALGORITHM FOR BOTTOM-UP HEAP CONSTRUCTION

- Here is a recursive algorithm given $n = 2^{h+1} - 1$ for some positive integer h:

Algorithm BottomUpHeap(L):
  Input: An STL list L storing $n = 2^{h+1} - 1$ entries
  Output: A heap T storing the entries of L.
  if L.empty() **then**
    return an empty heap
  $e \leftarrow L$.front() {e is now the first element in L}
  L.pop_front() {remove the first element in L}
  Split L into two lists, L_1 and L_2, each of size $(n-1)/2$
  $T_1 \leftarrow$ BottomUpHeap(L_1) {recursive call on the first half of L}
  $T_2 \leftarrow$ BottomUpHeap(L_2) {recursive call on the second half of L}
  Create binary tree T with root r storing e, left subtree T_1, and right subtree T_2
  Perform a down-heap bubbling from the root r of T, if necessary
  return T