IMPLEMENTING A PRIORITY QUEUE

As a **sorted** Doubly-linked list:

Elements **are inserted based on their key** to ensure the priority queue is always sorted

What is the complexity of these methods?

- `size()`: $O(1)$
- `empty()`: $O(1)$
- `insert(e)`: $O(n)$
- `min()`: $O(1)$
- `removeMin()`: $O(1)$

As an **unsorted** Doubly-linked list:

This means that elements are **inserted at the end of the list**

What is the complexity of these methods?

- `size()`: $O(1)$
- `empty()`: $O(1)$
- `insert(e)`: $O(1)$
- `min()`: $O(n)$
- `removeMin()`: $O(n)$

PRIORITY QUEUE IMPLEMENTATION

- Let us look at a possible implementation of a priority queue as a **sorted linked list**:

```
template <typename E, typename C>
class ListPriorityQueue {
public:
    int size( ) const;
    bool empty( ) const;
    void insert(const E& e);
    const E& min( ) const;
    void removeMin( );
private:
    std::list<E> L;
    C isLess;
};
```

```
template <typename E, typename C>
const E& ListPriorityQueue<E,C>::min( ) const
{   return L.front( );   }
```

```
template <typename E, typename C>
void ListPriorityQueue<E,C>::removeMin( )
{   L.pop_front( );   }
```

PRIORITY QUEUE IMPLEMENTATION

- Let us look at a possible implementation of a priority queue as a **sorted linked list**:

```
template <typename E, typename C>
class ListPriorityQueue {
public:
    int size( ) const;
    bool empty( ) const;
    void insert(const E& e);
    const E& min( ) const;
    void removeMin( );
private:
    std::list<E> L;
    C isLess;
};
```

```
template <typename E, typename C>
void ListPriorityQueue<E,C>::insert(const E& e) {
    typename std::list<E>::iterator p;
    p = L.begin( );
    while (p != L.end() && !isLess(e, *p)) ++p;
    L.insert(p, e);
}
```

This implies that the list is sorted in an ascending order (i.e., the first element has the smallest key, while the last element has the largest)

SORTING WITH A PRIORITY QUEUE

- Given an **unsorted 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

Phase 1

Phase 2

Algorithm PriorityQueueSort(L, P):
    while !L.empty() **do**
        $e \leftarrow L$.front
        L.pop_front() {remove element e from the list}
        P.insert(e) {add element e to the priority queue}
    while !P.empty() **do**
        $e \leftarrow P$.min()
        P.removeMin() {remove the smallest element e from the queue}
        L.push_back(e) {append element e to the back of L}

SORTING WITH A PRIORITY QUEUE

- Given an **unsorted 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
- This approach depends on how `insert(e)`, `min()` and `removeMin()` are implemented:
  - **Option 1:** If `insert(e)` inserts e at the **end of the queue** (which can be done instantly), then `min()` and `removeMin()` must **search the queue** to find the minimum element!
  - **Option 2:** If `insert(e)` inserts e **based on its key** (so that, after the insert, the queue is always sorted), then `min()` and `removeMin()` can **instantly retrieve the minimum** element (this would be the first element in the queue)!
- If we go with **Option 1**, we end up with an algorithm called "**selection sort**", while if we go with **Option 2** we end up with an algorithm called "**insertion sort**"

SORTING WITH A PRIORITY QUEUE

	List L	*Priority Queue P*
Input	(7,4,8,2,5,3,9)	()
Phase 1	(4,8,2,5,3,9)	(7)
	(8,2,5,3,9)	(7,4)
	⋮	⋮
	()	(7,4,8,2,5,3,9)
Phase 2	(2)	(7,4,8,5,3,9)
	(2,3)	(7,4,8,5,9)
	(2,3,4)	(7,8,5,9)
	(2,3,4,5)	(7,8,9)
	(2,3,4,5,7)	(8,9)
	(2,3,4,5,7,8)	(9)
	(2,3,4,5,7,8,9)	()

Example of **selection sort**

	List L	*Priority Queue P*
Input	(7,4,8,2,5,3,9)	()
Phase 1	(4,8,2,5,3,9)	(7)
	(8,2,5,3,9)	(4,7)
	(2,5,3,9)	(4,7,8)
	(5,3,9)	(2,4,7,8)
	(3,9)	(2,4,5,7,8)
	(9)	(2,3,4,5,7,8)
	()	(2,3,4,5,7,8,9)
Phase 2	(2)	(3,4,5,7,8,9)
	(2,3)	(4,5,7,8,9)
	⋮	⋮
	(2,3,4,5,7,8,9)	()

Example of **insertion sort**

COMPLEXITY ANALYSIS

- Let’s compare the **complexity**. Remember:
  - **Selection sort:** `insert(e)` inserts `e` at the **end of the queue**, while `removeMin()` and `min()` **search the queue** to find the minimum element.
  - **Insertion sort:** `insert(e)` inserts `e` **based on its key** (so that, after the insert, the queue is always sorted), while `min()` and `removeMin()` **retrieve the minimum** element
- With **selection-sort**:
  - The total time needed for the *first phase* → O(n)
  - The total time needed for the *second phase* → $O(n^2)$

  The first `removeMin()` operation takes O(n) time, the second one takes O(n – 1) time, etc. Thus, inserting n elements back to L takes:

  $$n + n - 1 + \cdots + 2 + 1 = \sum_{x=1}^{n} x = \frac{n(n+1)}{2} \quad \text{which is } O(n^2)$$

- With **insertion sort**, the opposite is true; the first phase takes $O(n^2)$ time, while the second takes O(n) time.

SORTING WITH A PRIORITY QUEUE

- To summarize, given a list L of n elements, we can sort L using a priority queue Q:
  - **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
- Depending on how the elements are inserted in P, we have **two options**:
  - **Selection sort:** Each `insert(e)` takes O(1), while each `removeMin()`/`min()` takes O(n)
  - **Insertion sort:** Each `insert(e)` takes O(n), while each `removeMin()`/`min()` takes O(1)
- Both of these algorithms run in $O(n^2)$ time. However, there is a **third option:**
  - **Heap sort:** `insert(e)`, `removeMin()` and `min()` take O(n log n) time!
- Before explaining how it works, we need to discuss “**complete binary tree**” and “**heaps**”.

PRIORITY QUEUE STL

- C++ provides a readily-available priority queue as part of the queue STL in C++. The `priority_queue` class is **templated** with three parameters:

  1. The **base type** of the elements; you must specify this argument.
  2. The **STL container** in which the elements are stored; if you don't specify this argument, an STL vector is used by default.
  3. The **comparator object**. If you don't specify this argument, the standard C++ less-than operator ("<") is used by default.

- Here are example's of how to define a priority queue:

```
#include <queue>
using namespace std;
priority_queue<int> p1;
priority_queue<Point2D, vector<Point2D>, LeftRight> p2;
```

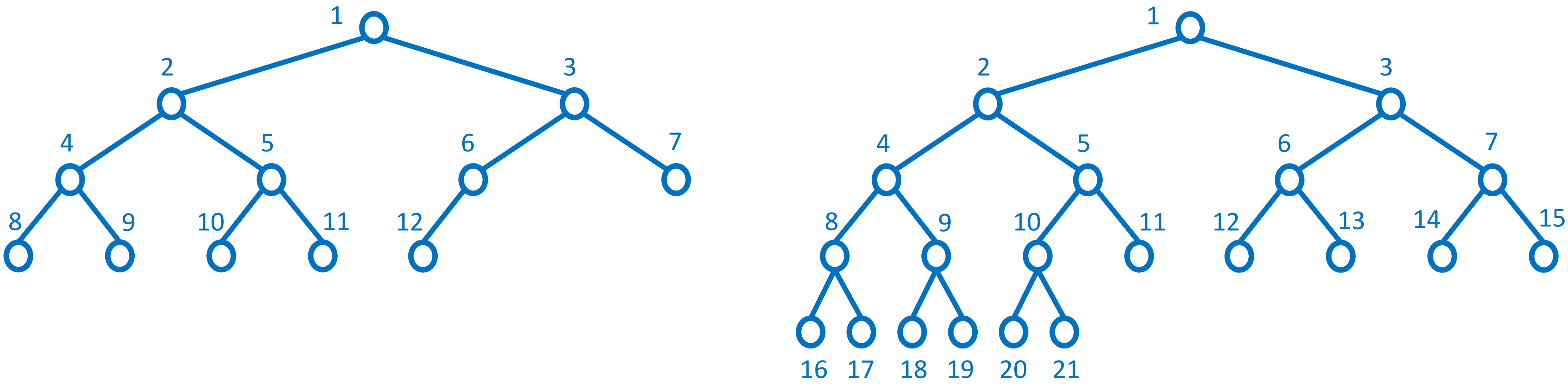
PRIORITY QUEUE ADT VS. STL

- In the case of **priority queue**, the differences are:
  - Instead of insert(e) which can be *implemented either by inserting e at the end or based on its key*, we have push(e) which **inserts** e **based on its key**.
  - Instead of `min()` and `removeMin()` which return and remove an element with the *smallest key*, we have `top()` and `pop()` which return and remove an element with the **largest key**.

24 COMPLETE BINARY TREES

COMPLETE BINARY TREE

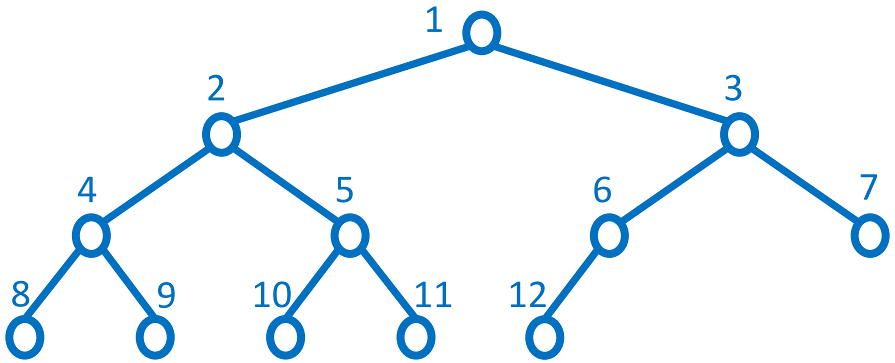
- A binary tree T with height h is **complete** if:
  - Levels 0, 1, 2, . . . , h−1 of T have the **maximum number of nodes** possible
  - The remaining nodes fill level h **from left to right**
- Examples:



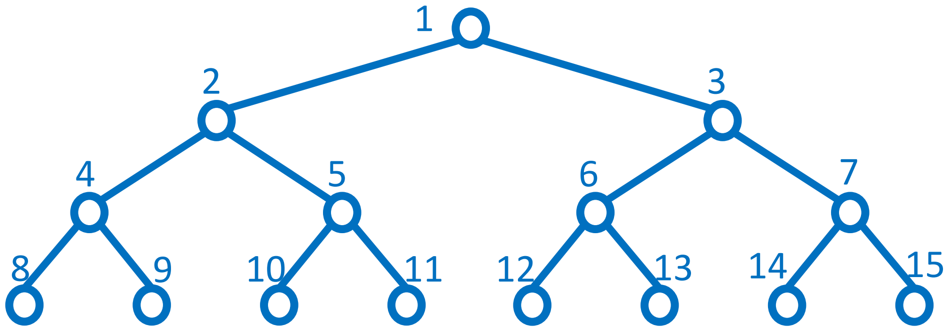
COMPLETE BINARY TREE

Proposition: A complete binary tree T with n nodes has $h = \lfloor \log n \rfloor$

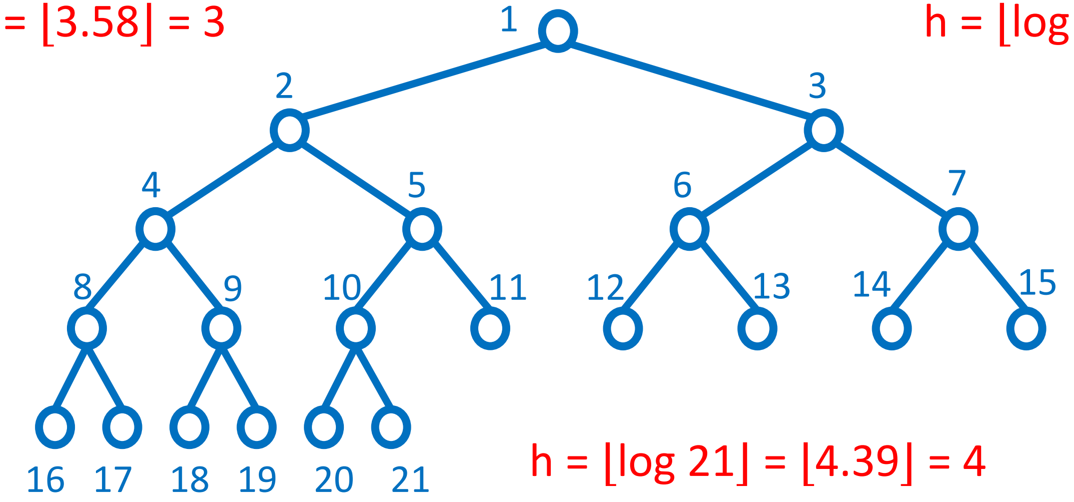
Experimental Justification:



$h = \lfloor \log 12 \rfloor = \lfloor 3.58 \rfloor = 3$



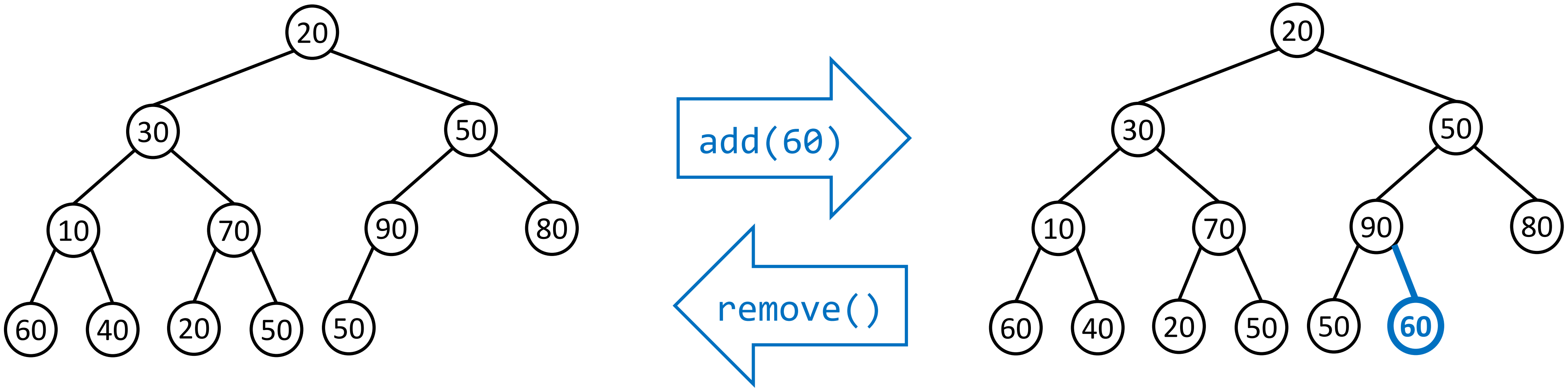
$h = \lfloor \log 15 \rfloor = \lfloor 3.90 \rfloor = 3$



$h = \lfloor \log 21 \rfloor = \lfloor 4.39 \rfloor = 4$

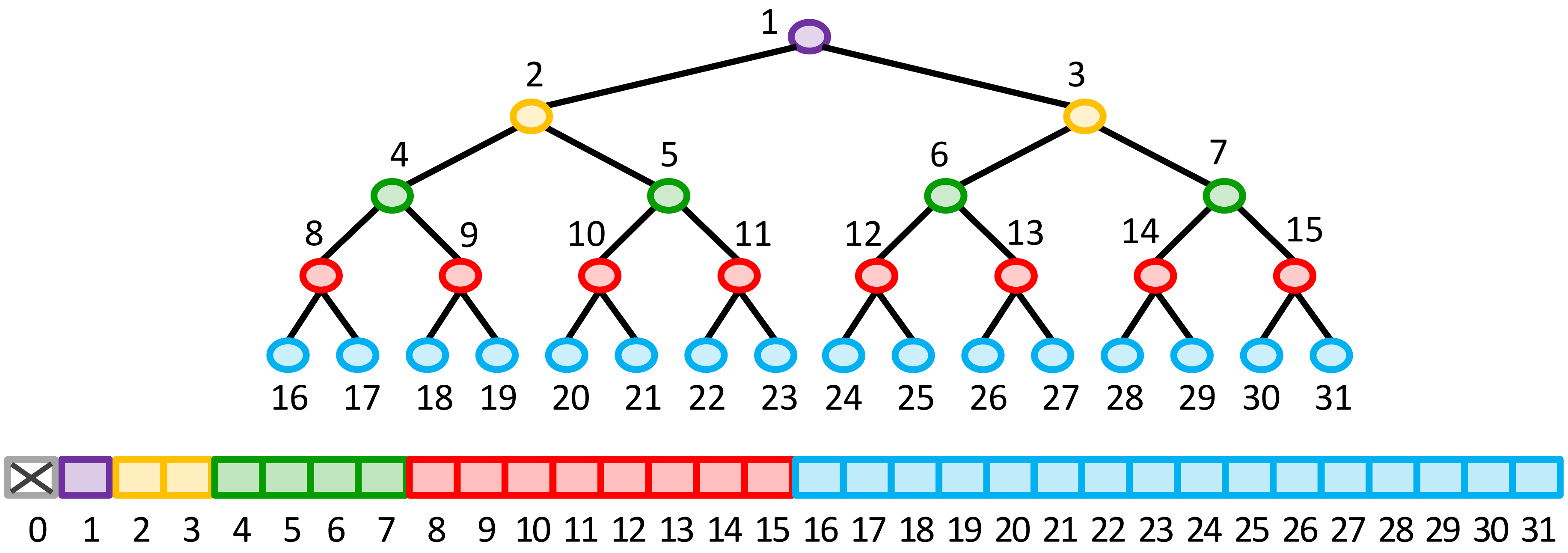
COMPLETE BINARY TREE ADT

- A **complete binary tree** ADT supports all the functions of a **binary tree**, in addition to the following:
  - `add(e)`: Add e to T (and return a new external node v storing element e) such that the resulting tree is a complete binary tree with last node v.
  - `remove()`: Remove the last node of T and return its element.
- By using only these update operations, the binary tree is guaranteed to be complete



VECTOR-BASED IMPLEMENTATION

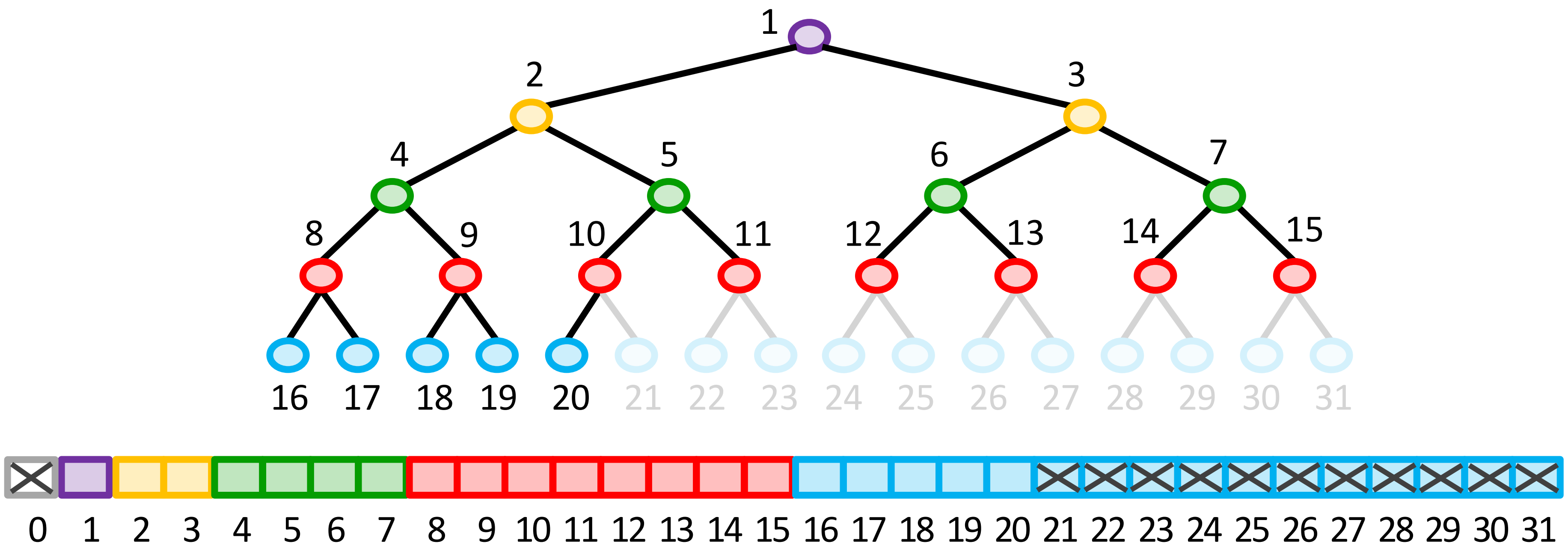
- Remember how we said we can represent a binary tree as a vector



- This representation is especially suitable for complete binary trees, because we always fill the last level of the tree from left to right, which corresponds to **filling the vector from left to right!**

VECTOR-BASED IMPLEMENTATION

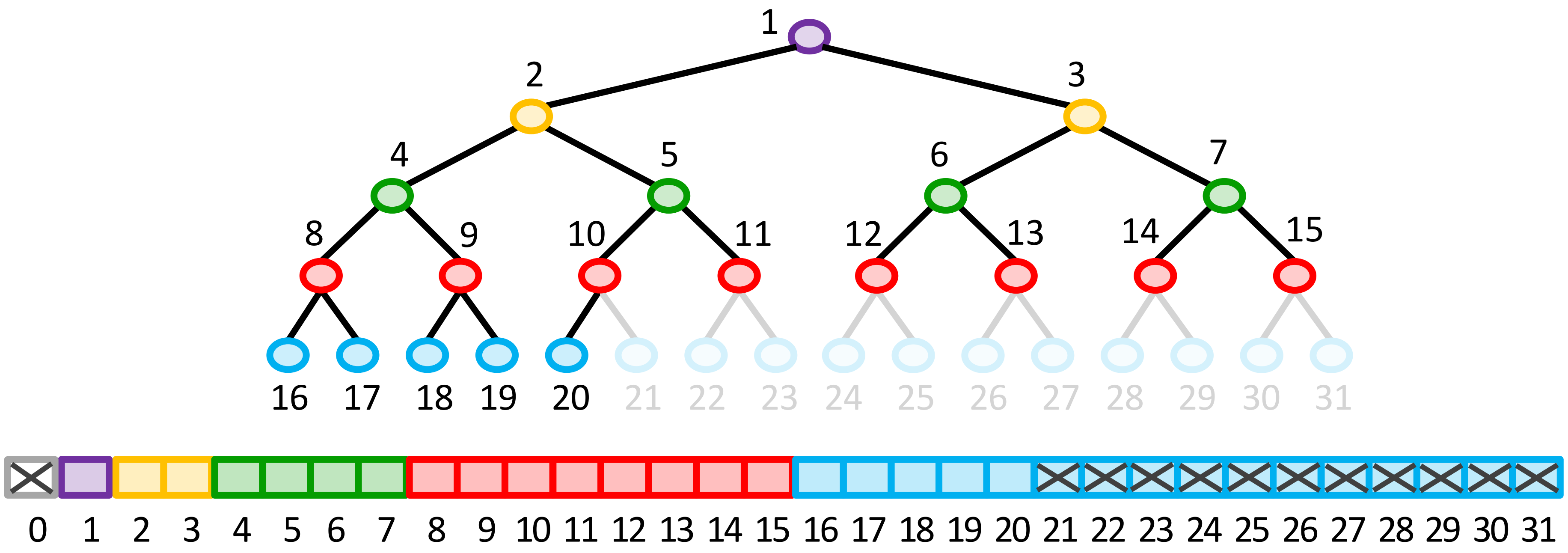
- Remember how we said we can represent a binary tree as a vector



- This representation is especially suitable for complete binary trees, because we always fill the last level of the tree from left to right, which corresponds to **filling the vector from left to right!**

VECTOR-BASED IMPLEMENTATION

- Remember how we said we can represent a binary tree as a vector



- What is the complexity of `add(e)` and `remove()`?

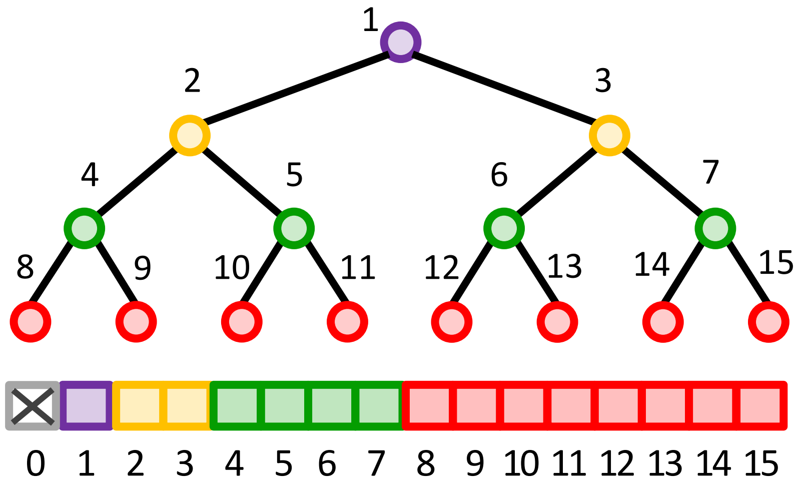
If we use an extendable array these methods would take O(1) amortized time!

C++ VECTOR-BASED IMPLEMENTATION

```
template <typename E>
class VectorCompleteTree {
private:
    std::vector<E> V; // the vector in which the tree will be stored
protected:
    Position pos(int i)  { return V.begin() + i; }
    int idx(const Position& p) const  {  return p – V.begin();  }
public:
    typedef typename std::vector<E>::iterator Position; // a position in the tree
    VectorCompleteTree() : V(1) { } // constructor
    int size() const
            Position left(const Position& p)
            Position right(const Position& p)
getters     Position parent(const Position& p)
            Position root()
            Position last()
            bool hasLeft(const Position& p) const
checkers    bool hasRight(const Position& p) const
            bool isRoot(const Position& p) const
            void add(const E& e)
modifiers   void remove()
            void swap(const Position& p, const Position& q)
};
```

This method maps an index i to a position

It returns the position of the 1^{st} element + i



C++ VECTOR-BASED IMPLEMENTATION

```
template <typename E>
class VectorCompleteTree {
private:
    std::vector<E> V; // the vector in which the tree will be stored
protected:
    Position pos(int i)  {  return V.begin() + i;  }
    int idx(const Position& p) const  { return p – V.begin(); }
public:
    typedef typename std::vector<E>::iterator Position; // a position in the tree
    VectorCompleteTree() : V(1) { } // constructor
    int size() const
    Position left(const Position& p)
    Position right(const Position& p)
    Position parent(const Position& p)
    Position root()
    Position last()
    bool hasLeft(const Position& p) const
    bool hasRight(const Position& p) const
    bool isRoot(const Position& p) const
    void add(const E& e)
    void remove()
    void swap(const Position& p, const Position& q)
};
```

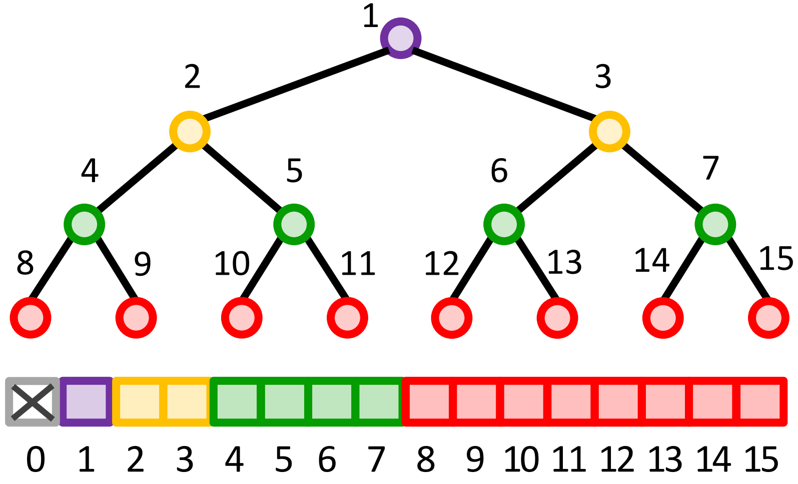
getters

checkers

modifiers

This method returns the index of the element at position *p*

It returns the difference between the position of this element and the position of the 1st element



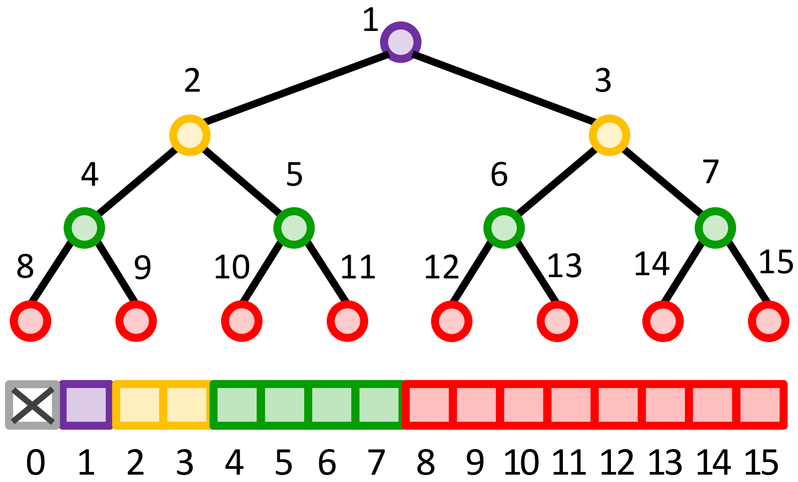
C++ VECTOR-BASED IMPLEMENTATION

```
template <typename E>
class VectorCompleteTree {
private:
    std::vector<E> V; // the vector in which the tree will be stored
protected:
    Position pos(int i)  {  return V.begin() + i;  }
    int idx(const Position& p) const  {  return p – V.begin();  }
public:
    typedef typename std::vector<E>::iterator Position; // a position in the tree
    VectorCompleteTree() : V(1) { } // constructor
    int size() const                          { return V.size() – 1; }
    Position left(const Position& p)          { return pos(2*idx(p)); }
    Position right(const Position& p)         { return pos(2*idx(p) + 1); }
    Position parent(const Position& p)        { return pos(idx(p)/2); }
    Position root()                           { return pos(1); }
    Position last()                           { return pos(size()); }
    bool hasLeft(const Position& p) const     { return 2*idx(p) <= size(); }
    bool hasRight(const Position& p) const    { return 2*idx(p) + 1 <= size(); }
    bool isRoot(const Position& p) const      { return idx(p) == 1; }
    void add(const E& e)                      { V.push_back(e); }
    void remove()                             { V.pop_back(); }
    void swap(const Position& p, const Position& q)  { E e = *q; *q = *p; *p = e; }
};
```

getters

checkers

modifiers



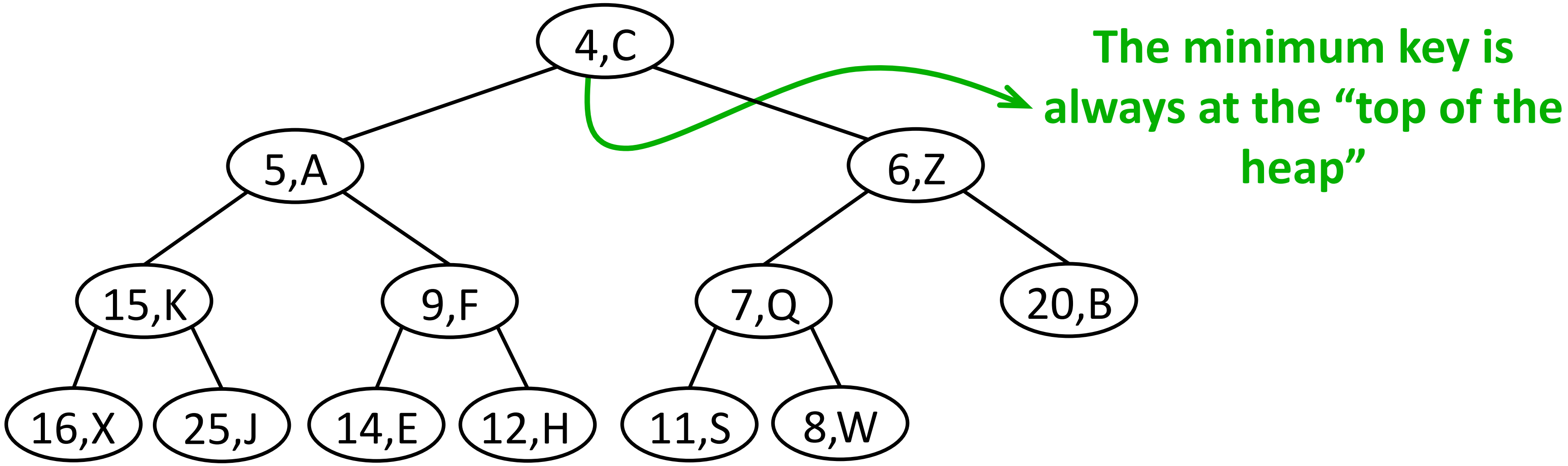
34

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**.