

A very brief introduction to

The Standard Template Library

Standard Template Library

- A collection of common data structures and algorithms
- Compile-time polymorphism (templates)

```
std::vector<int>  
std::list<double>  
std::hash_map<std::string, std::pair<int, int>>
```

- Efficient implementations



```
std::sort(...); //O(n log n)
```

- Correct implementations!



```
std::binary_search(...);
```

Raises the level of abstraction

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the other side of another panel.
    const int center_x = fixed_panel->cur_panel_center();
    for (size_t i = 0; i < expanded_panels_.size(); ++i) {
        Panel* panel = expanded_panels_[i].get();
        if (center_x <= panel->cur_panel_center() ||
            i == expanded_panels_.size() - 1) {
            if (panel != fixed_panel) {
                // If it has, then we reorder the panels.
                ref_ptr<Panel> ref = expanded_panels_[fixed_index];
                expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
                if (i < expanded_panels_.size()) {
                    expanded_panels_.insert(expanded_panels_.begin() + i, ref);
                } else {
                    expanded_panels_.push_back(ref);
                }
            }
        }
        break;
    }
}
```

1/4

Actual code!



Raises the level of abstraction

```
// Find the total width of the panels to the left of the fixed panel.
int total_width = 0;
fixed_index = -1;
for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
        fixed_index = i;
        break;
    }
    total_width += panel->panel_width();
}

CHECK_NE(fixed_index, -1);
int new_fixed_index = fixed_index;

// Move panels over to the right of the fixed panel until all of the ones
// on the left will fit.
int avail_width = max(fixed_panel->cur_panel_left() - kBarPadding, 0);
while (total_width > avail_width) {
    new_fixed_index--;
    CHECK_GE(new_fixed_index, 0);
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Reorder the fixed panel if its index changed.
if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
```

2/4

Actual code!



⋮

Raises the level of abstraction

3/4

```
expanded_panels_.erase(it);
expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
fixed_index = new_fixed_index;
}

// Now find the width of the panels to the right, and move them to the
// left as needed.
total_width = 0;
for (Panels::iterator it = expanded_panels_.begin() + fixed_index + 1;
     it != expanded_panels_.end(); ++it) {
    total_width += (*it)->panel_width();
}
avail_width = max(wm_->width() - (fixed_panel->cur_right() + kBarPadding), 0);

while (total_width > avail_width) {
    new_fixed_index++;
    CHECK_LT(new_fixed_index, expanded_panels_.size());
    total_width -= expanded_panels_[new_fixed_index]->panel_width();
}

// Do the reordering again.
if (new_fixed_index != fixed_index) {
    Panels::iterator it = expanded_panels_.begin() + fixed_index;
    ref_ptr<Panel> ref = *it;
    expanded_panels_.erase(it);
    expanded_panels_.insert(expanded_panels_.begin() + new_fixed_index, ref);
    fixed_index = new_fixed_index;
}
```

Actual code!



Raises the level of abstraction

```
// Finally, push panels to the left and the right so they don't overlap.
int boundary = expanded_panels_[fixed_index]->cur_panel_left() - kBarPadding;
for (Panels::reverse_iterator it =
    // Start at the panel to the left of 'new_fixed_index'.
    expanded_panels_.rbegin() + (expanded_panels_.size() - new_fixed_index);
    it != expanded_panels_.rend(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_right() > boundary) {
        panel->Move(boundary, kAnimMs);
    } else if (panel->cur_panel_left() < 0) {
        panel->Move(min(boundary, panel->panel_width() + kBarPadding), kAnimMs);
    }
    boundary = panel->cur_panel_left() - kBarPadding;
}
boundary = expanded_panels_[fixed_index]->cur_right() + kBarPadding;

for (Panels::iterator it = expanded_panels_.begin() + new_fixed_index + 1;
    it != expanded_panels_.end(); ++it) {
    Panel* panel = it->get();
    if (panel->cur_panel_left() < boundary) {
        panel->Move(boundary + panel->panel_width(), kAnimMs);
    } else if (panel->cur_right() > wm_->width()) {
        panel->Move(max(boundary + panel->panel_width(),
            wm_->width() - kBarPadding),
            kAnimMs);
    }
    boundary = panel->cur_right() + kBarPadding;
}
}
```

4/4

Actual code!



Raises the level of abstraction

Using STL:

```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
    CHECK(fixed_panel);

    // First, find the index of the fixed panel.
    int fixed_index = GetPanelIndex(expanded_panels_, *fixed_panel);
    CHECK_LT(fixed_index, expanded_panels_.size());

    // Next, check if the panel has moved to the left side of another panel.
    auto f = begin(expanded_panels_) + fixed_index;
    auto p = lower_bound(begin(expanded_panels_), f, center_x,
        [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

    // If it has, then we reorder the panels.
    rotate(p, f, f + 1);
}
```

Credit: Sean Parent

Collections & Iterators

Pairs

- `std::pair<T,U>` represents a pair of objects of types `T` and `U`, respectively.

```
std::pair<int, char> p(5, 'a');  
std::cout << p.first << "\n"; //5  
std::cout << p.second << "\n"; //a
```

- `std::make_pair(x,y)` creates a pair from its arguments `x` and `y`.

```
std::pair<int, char> p = std::make_pair(5, 'a');
```

- See also `std::tuple<T, U, V ...>`

Arrays

- A collection of **fixed** length
- Contains elements of the same type

```
std::array<int, 3> A; //Uninitialized contents  
std::array<int, 3> B = {}; //Default initialized  
std::array<int, 3> C = {1,2,3}; //Initialized
```

- Essentially a wrapper for C arrays.

```
C[0] = 4; //C now contains 4,2,3
```

- Better value semantics

```
std::array<int, 3> D;  
D = C; //D's elements are now a copy of C's elements
```

- Supports additional operations, e.g., `size()` or `at()`

Vectors

- A collection of **variable** length

```
std::vector<int> A; //Empty vector
std::vector<int> B(10); //10 default initialized ints
std::vector<int> C(10, 42); //10 ints initialized to 42
std::vector<int> D {1, 2, 3};
```

- Random access:

```
D[0] = 4; // D now contains: 4, 2, 3
```

- Insert an element at the end

```
D.push_back(10); // D now contains: 4, 2, 3, 10
```

- Delete the last element

```
D.pop_back();
```

Vectors

- A collection of **variable** length

```
std::vector<int> A; //Empty vector
std::vector<int> B(10); //10 default initialized ints
std::vector<int> C(10, 42); //10 ints initialized to 42
std::vector<int> D {1, 2, 3};
```

- Random access:

$O(1)$

```
D[0] = 4; // D now contains: 4, 2, 3
```

- Insert an element at the end

$O(1)$ amortized

```
D.push_back(10); // D now contains: 4, 2, 3, 10
```

- Delete the last element

$O(1)$

```
D.pop_back();
```

Vectors

- Insert an element at position i :

```
int i=2;  
std::vector<int> D {1, 2, 3, 4, 5};  
D.insert(D.begin()+i, 10);  
// D now contains: 1, 2, 10, 3, 4, 5
```

- Delete the element at position i :

```
int i=2;  
std::vector<int> D {1, 2, 3, 4, 5};  
D.erase(D.begin()+i);  
// D now contains: 1, 2, 4, 5
```

Vectors

- Insert an element at position i :
 $O(1)$ amortized + $O(\text{\#elements} - i)$

```
int i=2;  
std::vector<int> D {1, 2, 3, 4, 5};  
D.insert(D.begin()+i, 10);  
// D now contains: 1, 2, 10, 3, 4, 5
```

- Delete the element at position i : $O(\text{\#elements} - i)$

```
int i=2;  
std::vector<int> D {1, 2, 3, 4, 5};  
D.erase(D.begin()+i);  
// D now contains: 1, 2, 4, 5
```

Deque

- A vector-like collection that supports fast insertions and deletions from both ends

```
std::deque<int> D {1, 2, 3};
```

- Access to elements

```
std::cout << D[1] << "\n"; //Prints 2
```

- Insert an element at the beginning/end

```
D.push_front(10); //10, 1, 2, 3  
D.push_back(20); //10, 1, 2, 3, 20
```

- Delete an element from the beginning/end

```
D.pop_front(); //2,3  
D.pop_back(); //2
```

(See also `std::queue` and `std::stack`)

Deque

- A vector-like collection that supports fast insertions and deletions from both ends

```
std::deque<int> D {1, 2, 3};
```

- Access to elements

$O(1)$

```
std::cout << D[1] << "\n"; //Prints 2
```

- Insert an element at the beginning/end

$O(1)$ amortized

```
D.push_front(10); //10, 1, 2, 3
```

```
D.push_back(20); //10, 1, 2, 3, 20
```

- Delete an element from the beginning/end

$O(1)$ amortized

```
D.pop_front(); //2,3
```

```
D.pop_back(); //2
```

(See also `std::queue` and `std::stack`)

Iterators

- Iterators provide access to elements in a collection
- An iterator object points to an element of a collection
- Iterators can be dereferenced (* operator) to access the pointed element
- Iterators can be advanced with the ++ operator.
- Some iterators can moved backwards with the -- operator.
- Some iterators support random access via addition or the [] operator.

Iterators

- Collections have a `begin()` method that returns an iterator to their first element.
- ... and an `end()` method that returns an iterator pointing *one past* the last element.
- Use `std::begin()` and `std::end()` to get iterators for C arrays.

```
std::vector<int> V {1,2,3,4,5};  
for(std::vector<int>::iterator it=V.begin(); it<V.end(); it++)  
    std::cout << *it << "□";  
std::cout << "\n";
```

```
int V[] = {1,2,3,4,5};  
auto it = std::begin(V);  
std::cout << *(it+2) << "\n";
```

Lists

- A collection of **variable** length. Implemented as a doubly-linked list.

```
std::list<int> A; //Empty list
std::list<int> B(10); //10 default initialized ints
std::list<int> C(10, 42); //10 ints initialized to 42
std::list<int> D {1, 2, 3};
```

- No random access
- Insert an element at the beginning/end

```
D.push_front(10); //10, 1, 2, 3
D.push_back(20); //10, 1, 2, 3, 20
```

- Delete an element from the beginning/end

```
D.pop_front(); //2,3
D.pop_back(); //2
```

Lists

- A collection of **variable** length. Implemented as a doubly-linked list.

```
std::list<int> A; //Empty list
std::list<int> B(10); //10 default initialized ints
std::list<int> C(10, 42); //10 ints initialized to 42
std::list<int> D {1, 2, 3};
```

- No random access
- Insert an element at the beginning/end

$O(1)$

```
D.push_front(10); //10, 1, 2, 3
D.push_back(20); //10, 1, 2, 3, 20
```

- Delete an element from the beginning/end

$O(1)$

```
D.pop_front(); //2,3
D.pop_back(); //2
```

Lists

- Get an iterator to position i

```
int i=2;  
std::list<int> L {1, 2, 3, 4, 5};  
std::list<int>::it = L.begin();  
std::advance(it, i); //*it = 3
```

- Insert an the element at a given position:

```
L.insert(it, 10); // L now contains: 1, 2, 10, 3, 4, 5
```

- Delete the element at a given position:

```
L.erase(it);
```

Lists

- Get an iterator to position i $O(1 + i)$

```
int i=2;  
std::list<int> L {1, 2, 3, 4, 5};  
std::list<int>::it = L.begin();  
std::advance(it, i); //*it = 3
```

- Insert an the element at a given position: $O(1)$

```
L.insert(it, 10); // L now contains: 1, 2, 10, 3, 4, 5
```

- Delete the element at a given position: $O(1)$

```
L.erase(it);
```

Sets

- A collection that maintains a *sorted* set of unique keys

```
std::set<int> S {1, 2, 5};
```

- Insertion

```
S.insert(10); //S now represents {1,2,5,10}  
S.insert(10); //S represents the same set
```

- Lookup

```
S.find(5); //Returns an iterator to element 5  
S.find(20); //No such element. Returns S.end()
```

- Deletion

```
S.erase(5);
```

Sets

- A collection that maintains a *sorted* set of unique keys

```
std::set<int> S {1, 2, 5};
```

- Insertion

$O(\log n)$

```
S.insert(10); //S now represents {1,2,5,10}
```

```
S.insert(10); //S represents the same set
```

- Lookup

$O(\log n)$

```
S.find(5); //Returns an iterator to element 5
```

```
S.find(20); //No such element. Returns S.end()
```

- Deletion

$O(\log n)$

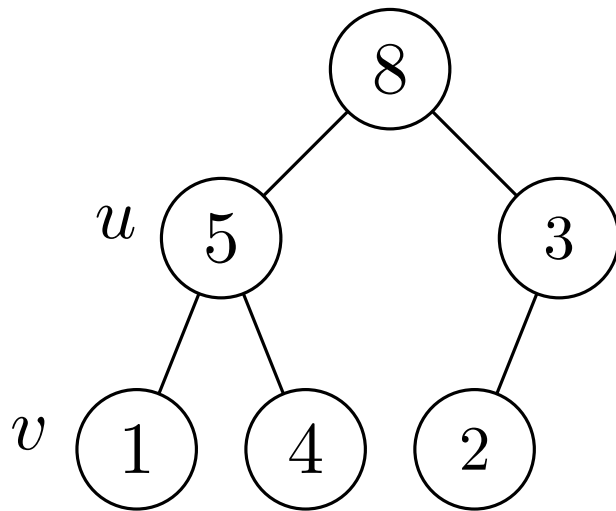
```
S.erase(5);
```

(See also `std::unordered_set`, `std::multiset`,
and `std::unordered_multiset`)

Algorithms

(max-)Heaps

- Binary tree with keys attached to vertices
- Nearly complete
(Complete except possibly for the last level. All leaves on the left.)
- Heap property: if u is v 's parent $\Rightarrow \text{key}(u) \geq \text{key}(v)$



0	1	2	3	4	5
8	5	3	1	4	2

$$\text{left}(u) = 2u + 1$$

$$\text{right}(u) = 2u + 2$$

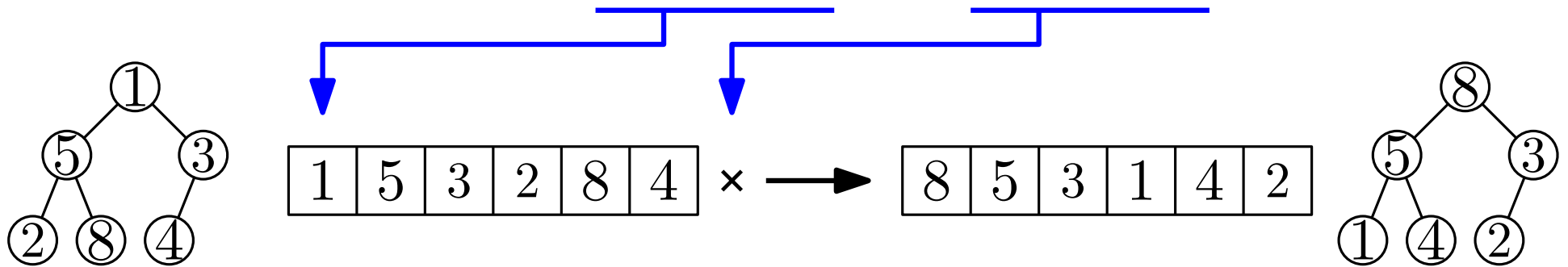
$$\text{parent}(u) = \lfloor \frac{u-1}{2} \rfloor$$

(max-)Heaps

- Building a Heap

$O(n)$

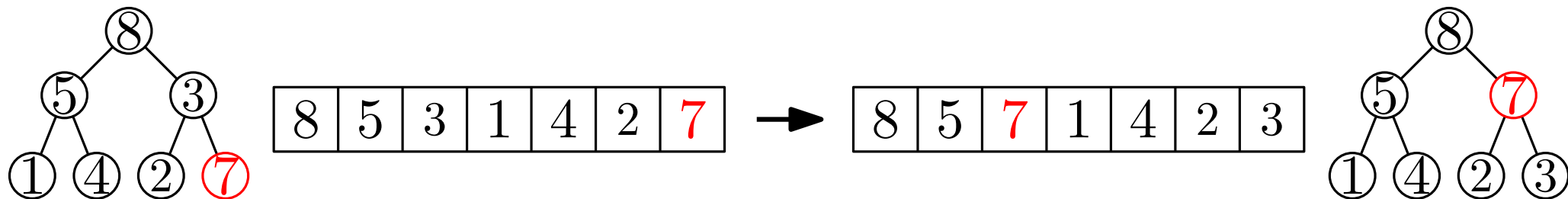
```
std::make_heap(vec.begin(), vec.end());
```



- Inserting a new key

$O(\log n)$

```
std::push_heap(vec.begin(), vec.end());
```

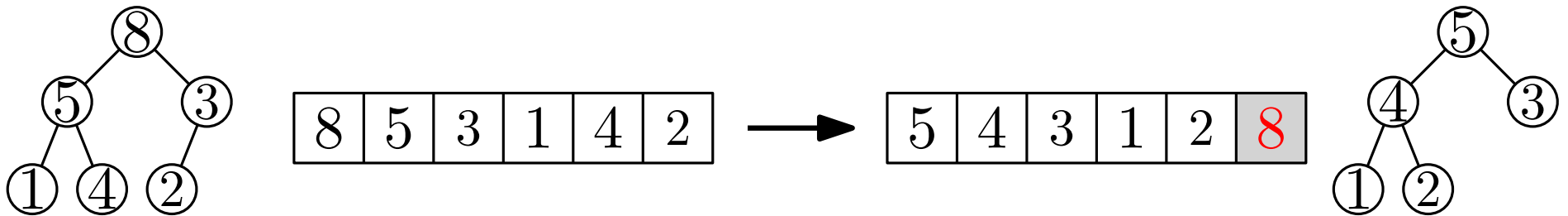


(max-)Heaps

- Extracting the maximum key

$O(\log n)$

```
std::pop_heap(vec.begin(), vec.end());
```

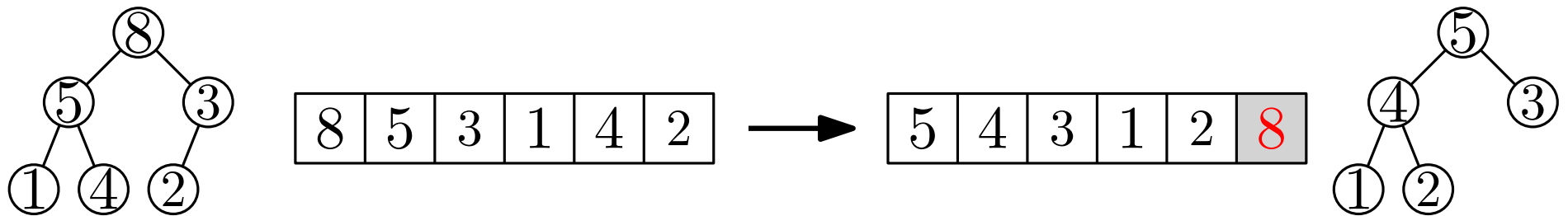


(max-)Heaps

- Extracting the maximum key

$O(\log n)$

```
std::pop_heap(vec.begin(), vec.end());
```



Example: Heapsort

```
std::vector<int> vec { 4, 1, 6, 2, 5, 3 };

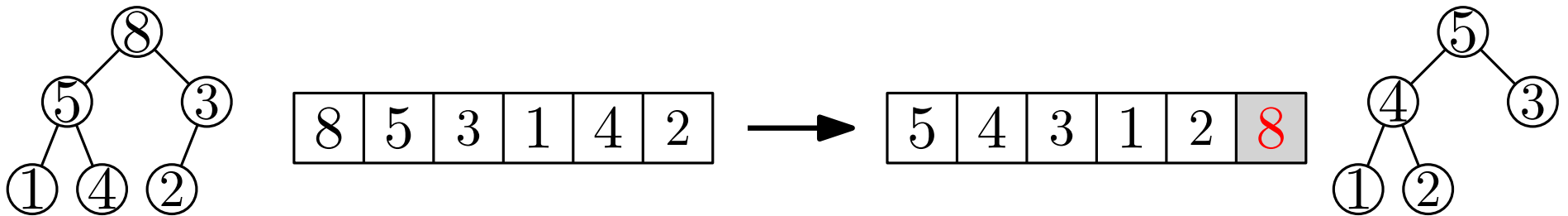
std::make_heap(vec.begin(), vec.end());
for(int i=vec.size(); i>1; i--)
    std::pop_heap(vec.begin(), vec.begin()+i);
```

(max-)Heaps

- Extracting the maximum key

$O(\log n)$

```
std::pop_heap(vec.begin(), vec.end());
```



Example: Heapsort

```
std::vector<int> vec { 4, 1, 6, 2, 5, 3 };
```

```
std::make_heap(vec.begin(), vec.end());
```

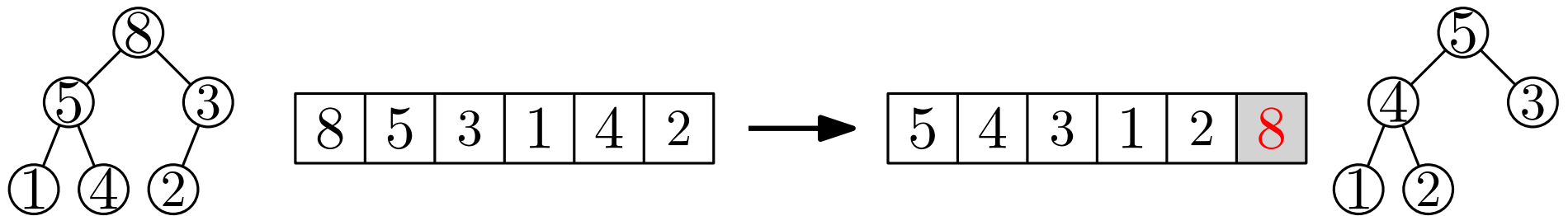
```
std::sort_heap(vec.begin(), vec.end());
```

(max-)Heaps

- Extracting the maximum key

$O(\log n)$

```
std::pop_heap(vec.begin(), vec.end());
```



Example: Heapsort

```
std::vector<int> vec { 4, 1, 6, 2, 5, 3 };
```

```
std::make_heap(vec.begin(), vec.end());
```

```
std::sort_heap(vec.begin(), vec.end());
```

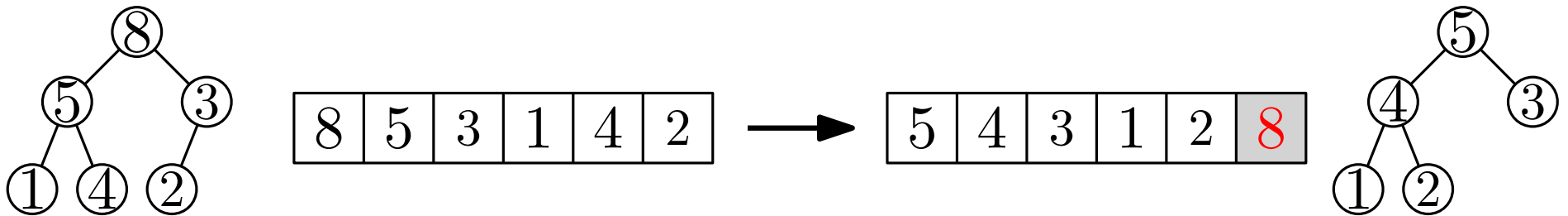
Min-Heaps?

(max-)Heaps

- Extracting the maximum key

$O(\log n)$

```
std::pop_heap(vec.begin(), vec.end());
```



Example: Heapsort

```
std::vector<int> vec { 4, 1, 6, 2, 5, 3 };  
  
std::make_heap(vec.begin(), vec.end());  
std::sort_heap(vec.begin(), vec.end());
```

Min-Heaps?

See `std::priority_queue` for a wrapper.

Sorting

- The whole container

$O(n \log n)$ comparisons

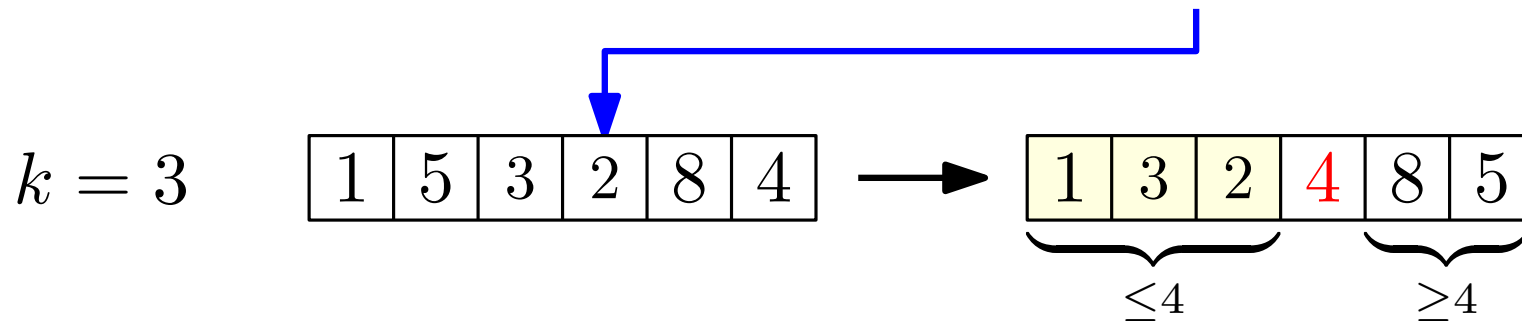
```
std::sort(vec.begin(), vec.end());
```



- k -th smallest element

$O(n)$ comparisons

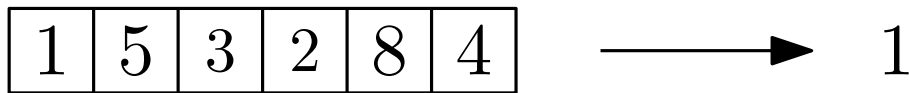
```
std::nth_element(vec.begin(), vec.begin()+k, vec.end());
```



Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

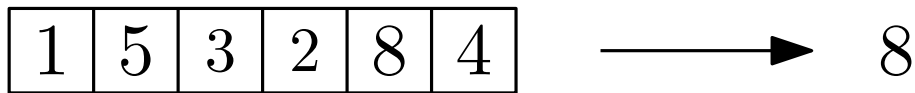


Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```



Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```

1	5	3	2	8	4
---	---	---	---	---	---

 $\longrightarrow$ std::pair(1, 8)

Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```

1	5	3	2	8	4
---	---	---	---	---	---

 $\longrightarrow$ std::pair(1, 8)

- Searching for an element

```
std::find(vec.begin(), vec.end(), 9);
```

6	9	3	5	9	4
---	---	---	---	---	---



Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```

1	5	3	2	8	4
---	---	---	---	---	---

 $\longrightarrow$ std::pair(1, 8)

- Searching for an element

```
std::find(vec.begin(), vec.end(), 2);
```

6	9	3	5	9	4
---	---	---	---	---	---

 ×



Linear Search

- Minimum/Maximum of a collection $O(n)$ comparisons

```
std::min_element(vec.begin(), vec.end());
```

```
std::max_element(vec.begin(), vec.end());
```

```
std::minmax_element(vec.begin(), vec.end());
```

1	5	3	2	8	4
---	---	---	---	---	---

 $\longrightarrow$ std::pair(1, 8)

- Searching for an element

```
std::find(vec.begin(), vec.end(), 2);
```

```
std::find_if(vec.begin(), vec.end(), [](int x) { return x%3; });
```

6	9	3	5	9	4
---	---	---	---	---	---



Binary Search

- Is 6 in the (sorted) collection? $O(\log n)$ comparisons

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```



Binary Search

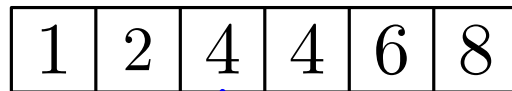
- Is 6 in the (sorted) collection? $O(\log n)$ comparisons

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```



- First index i s.t. $v[i] \geq 3$ (Where should 3 be inserted?)

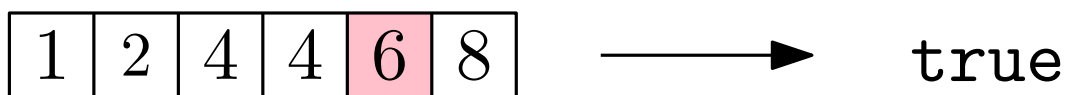
```
std::lower_bound(vec.begin(), vec.end(), 3);
```



Binary Search

- Is 6 in the (sorted) collection? $O(\log n)$ comparisons

```
bool std::binary_search(vec.begin(), vec.end(), 6);
```



- First index i s.t. $v[i] \geq 3$ (Where should 3 be inserted?)

```
std::lower_bound(vec.begin(), vec.end(), 3);
```

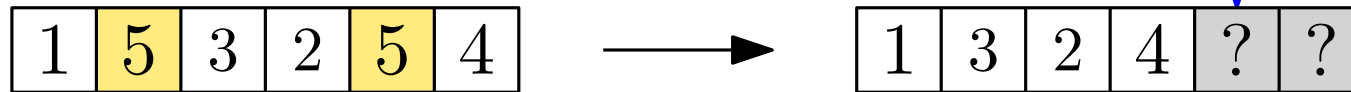
- First index i s.t. $v[i] > 4$

```
std::upper_bound(vec.begin(), vec.end(), 4);
```

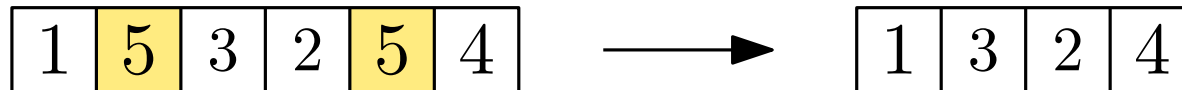


Deleting and replacing elements

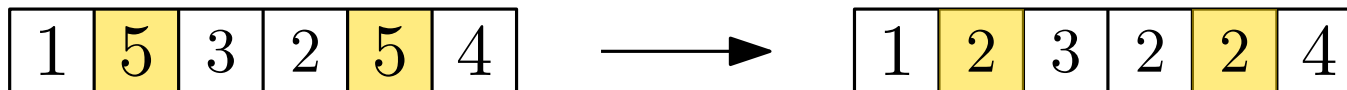
```
std::remove(vec.begin(), vec.end(), 5);
```



```
vec.erase(std::remove(vec.begin(), vec.end(), 5), vec.end());
```



```
std::replace(vec.begin(), vec.end(), 5, 2);
```



See also `std::remove_if` and `std::replace_if`.

(Partial) Sums

- Accumulate

$O(n)$ sums

```
std::accumulate(vec.begin(), vec.end());
```

$$\boxed{1} \boxed{5} \boxed{3} \longrightarrow \sum_{i=0}^2 \text{vec}[i] = 1 + 5 + 3 = 9$$

(Partial) Sums

- Accumulate

$O(n)$ sums

```
std::accumulate(vec.begin(), vec.end());
```

$$\boxed{1} \boxed{5} \boxed{3} \longrightarrow \sum_{i=0}^2 \text{vec}[i] = 1 + 5 + 3 = 9$$



```
std::accumulate(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\text{init} \oplus \left(\bigoplus_{i=0}^{n-1} \text{vec}[i] \right)$

(Partial) Sums

- Accumulate

```
std::accumulate(vec.begin(), vec.end(), 2, std::multiplies<int>);
```



```
std::accumulate(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\text{init} \oplus \left(\bigoplus_{i=0}^{n-1} \text{vec}[i] \right)$

(Partial) Sums

- Accumulate

```
std::accumulate(vec.begin(), vec.end(), 2, std::multiplies<int>);
```

$$\boxed{1} \boxed{5} \boxed{3} \longrightarrow 2 \cdot \prod_{i=0}^2 \text{vec}[i] = 2 \cdot 1 \cdot 5 \cdot 3 = 30$$



```
std::accumulate(vec.begin(), vec.end(), init, binary_op);
```

Note: `binary_op` needs to be commutative and associative.

Returns: $\text{init} \oplus \left(\bigoplus_{i=0}^{n-1} \text{vec}[i] \right)$

(Partial) Sums

- Accumulate

```
std::accumulate(vec.begin(), vec.end(), 2, std::multiplies<int>);
```

$$\boxed{1 \mid 5 \mid 3} \longrightarrow 2 \cdot \prod_{i=0}^2 \text{vec}[i] = 2 \cdot 1 \cdot 5 \cdot 3 = 30$$



```
std::accumulate(vec.begin(), vec.end(), init, binary_op);
```

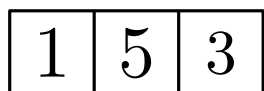
Note: `binary_op` needs to be commutative and associative.

Returns: $\text{init} \oplus \left(\bigoplus_{i=0}^{n-1} \text{vec}[i] \right)$

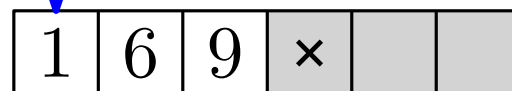
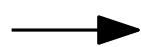
- Partial sums

$O(n)$ sums

```
std::partial_sum(vec.begin(), vec.end(), out.begin());
```

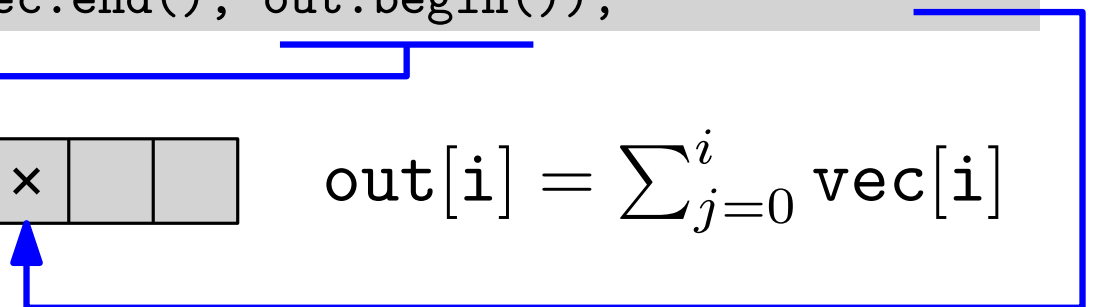


vec



out

$$\text{out}[i] = \sum_{j=0}^i \text{vec}[j]$$



... and much more.

See: <https://www.boost.org/sgi/stl/>