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::hasp_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

1/4

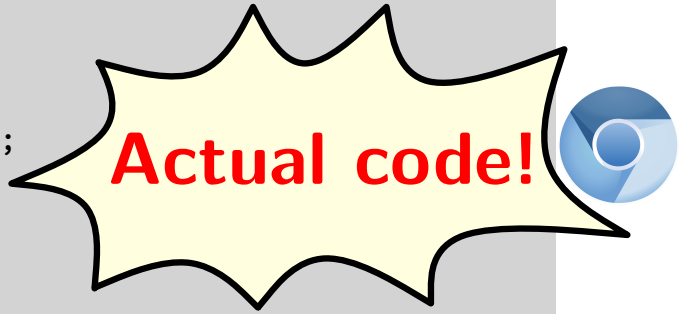
```
void PanelBar::RepositionExpandedPanels(Panel* fixed_panel) {
  CHECK(fixed_panel);

  // 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;
    }
  }
```

Actual code!

⋮

Raises the level of abstraction

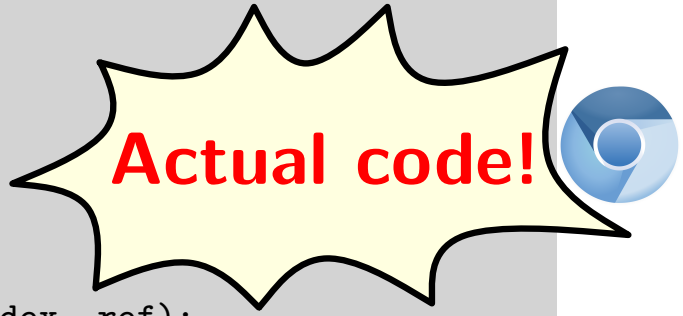

```
// Find the total width of the panels to the left of the fixed panel.
  int total_width = 0;
  fixed_index = -1;
  for (int i = 0; i < static_cast<int>(expanded_panels_.size()); ++i) {
    Panel* panel = expanded_panels_[i].get();
    if (panel == fixed_panel) {
      fixed_index = i;
      break;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



⋮

Raises the level of abstraction


```
  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;
}
```

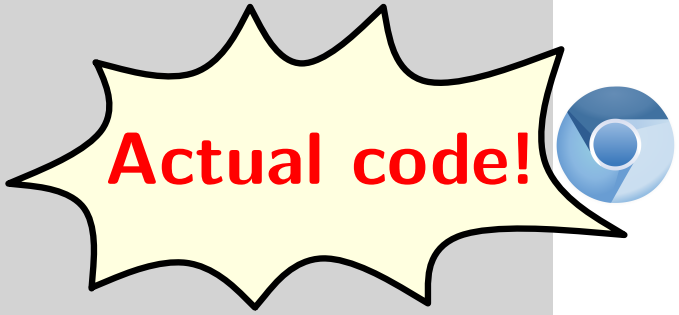


⋮

Raises the level of abstraction

4/4

```
// 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;
 }
}
```



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

Basics

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');
```

Tuples

- A generalization of `std::pair`. Represents a fixed size collection of values.
- Represents a fixed size collection of values.

```
std::tuple<int, char, bool> p(5, 'a', true);
p = std::make_tuple(6, 'b', false);
```

- To access tuple elements use `std::get`

```
std::tuple<int, char, int> p(5, 'a', 8);

std::cout << std::get<0>(p)
          << "␣" << std::get<2>(p) << "\n";
```

Strings

- Represents a *mutable* sequence of characters.

```
std::string s1; //An empty string
std:: string s2(10, 'a') //A string of 10 'a's
std::string s3 = "Hello␣world!"
```

- Accessing chracters

```
std::cout << s3[0] << "\n"; //Prints 'H'
s3[11] = '?'; //Hello world?
```

- Can be written to streams

```
std::cout << s3 << "\n";
```

(see STL documentation for more)

Collections & Iterators

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.

```
std::array<int, 3> D;
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()+2);
// 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()+2);
// 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
```

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
```

Queues

- Queue: FIFO data structure. A wrapper to deque.

```
std::queue<int> Q;
```

- push(x): inserts x at the end of Q.

```
Q.push(1); //Q contains 1
Q.push(2); //Q contains 1, 2
Q.push(3); //Q contains 1, 2, 3
```

- pop(): removes the first element from Q.

```
Q.pop(); //Q contains 2,3
```

- Q.front() returns the first element in Q

```
std::cout << Q.front() << "\n"; //Prints 2
```

Stacks

- Queue: LIFO data structure. A wrapper to deque.

```
std::stack<int> S;
```

- push(x): inserts x at the beginning of S.

```
S.push(1); //S contains 1
S.push(2); //S contains 2, 1
S.push(3); //S contains 3, 2, 1
```

- pop(): removes the first element from S.

```
S.pop(); //Q contains 2,3
```

- S.front() returns the first element in S

```
std::cout << S.top() << "\n"; //Prints 2
```

Iterators

- Iterators provide access to elements in a collection
- An iterator object points to an element of a collection
- Iterators can be dereferenced 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

- Containers 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 andom 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 andom 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);
```

- 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);
```

- 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 mantains 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
```

- Returns a pair `<iterator, bool>`
  - `iterator` points to the inserted element (if any)
  - `bool` is true iff a new element has been inserted

Sets

- Lookup

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

- Deletion

```
std::set<int>::iterator it = S.find(5);
if(it != S.end())
    S.erase(it);
```

Sets

- Lookup

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

- Deletion $O(1)$ amortized

```
std::set<int>::iterator it = S.find(5);
if(it != S.end())
    S.erase(it);
```

All other operations are supported in $O(\log n)$ time.

Multi-Sets & Unsorted Sets

- `std::multiset` mantains a *sorted* set of *not necessarily unique* keys

  Deletions: $O(1)$ amortized. All other operations $O(\log n)$.

- `std::unordered_set` mantains an *unsorted* set of *unique* keys

  All operations are supported in $O(1)$ average and linear worst-case time.

- `std::unordered_multiset` mantains an *unsorted* set of *not necessarily unique* keys

  All operations are supported in $O(1)$ average and linear worst-case time.

Maps

- A collection of key-value pairs, where keys are sorted and unique.

```
std::map<int, std::string> M;
```

- The operator [] returns a reference to the value associated with the given key. If the key does not exist it is created with default value.

```
M[2] = "Hello"
M[5] = "World"

std:: cout << M[2] << "\n";
std:: cout << M[3] << "\n";
```

Maps

- Lookup: returns an iterator to the element with key `key` if it exists, otherwise returns M.end().

```
int key = 5;
M.find(key);
```

- Deletion

```
std::map<int,string>::iterator it = M.find(5);
if(it != M.end())
    M.erase(it);
```

Maps

- Lookup: returns an iterator to the element with key `key` if it exists, otherwise returns M.end().

```
int key = 5;
M.find(key);
```

- Deletion $O(1)$ amortized

```
std::map<int,string>::iterator it = M.find(5);
if(it != M.end())
    M.erase(it);
```

All other operations are supported in $O(\log n)$ time.

Maps

- Lookup: returns an iterator to the element with key `key` if it exists, otherwise returns M.end().

```
int key = 5;
M.find(key);
```

- Deletion $O(1)$ amortized

```
std::map<int,string>::iterator it = M.find(5);
if(it != M.end())
    M.erase(it);
```

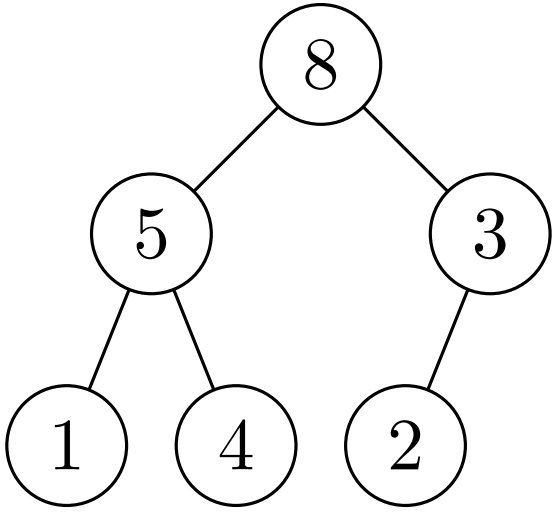
All other operations are supported in $O(\log n)$ time.

See also: `std::multimap`, `std::unordered_map`, `std::unordered_multimap`.

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 $\Rightarrow$ key$(u) \geq$ key(v)



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

left$(u) = 2u + 1$

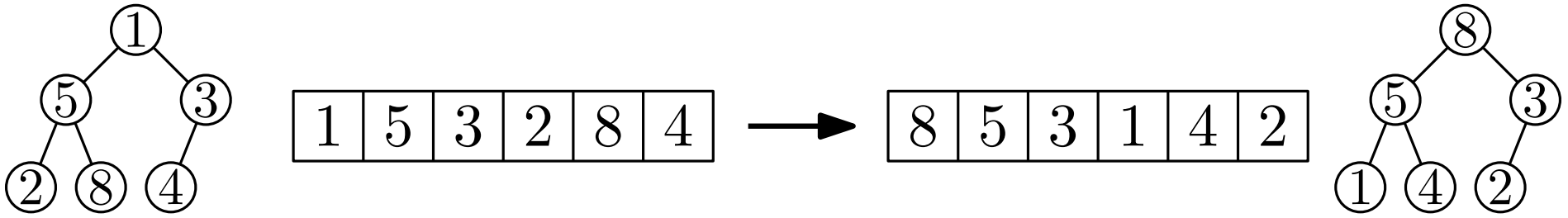
right$(u) = 2u + 2$

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

(max-)Heaps

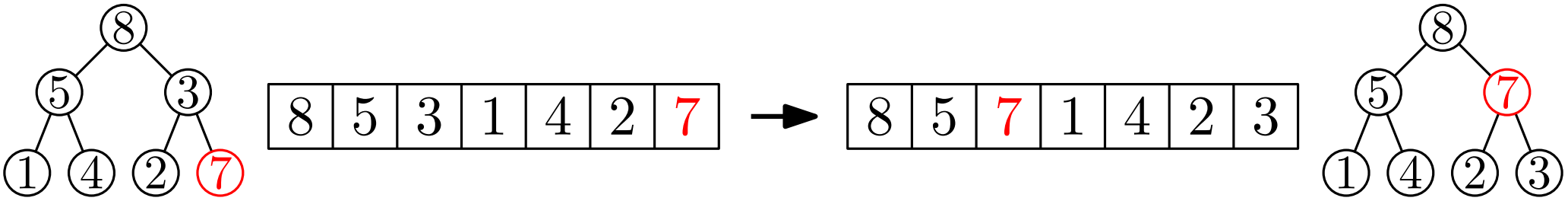
- Building a Heap $O(n)$

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



- Insering a new key $O(\log n)$

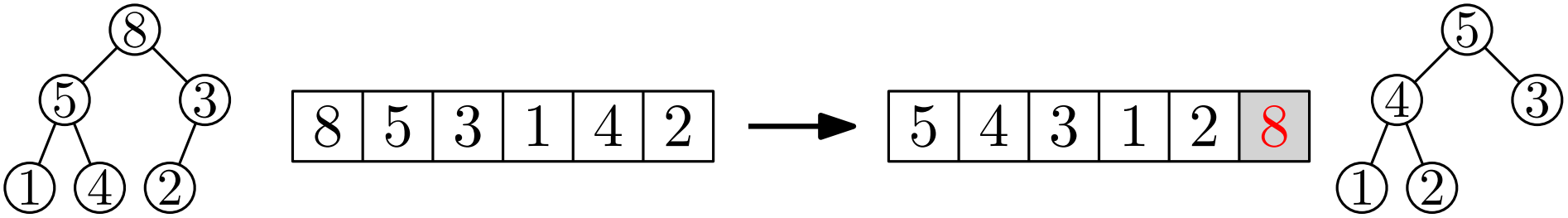
```
std::push_heap(vec.begin(), vec.end());
```



(max-)Heaps

- Extracting the maximum key $O(\log n)$

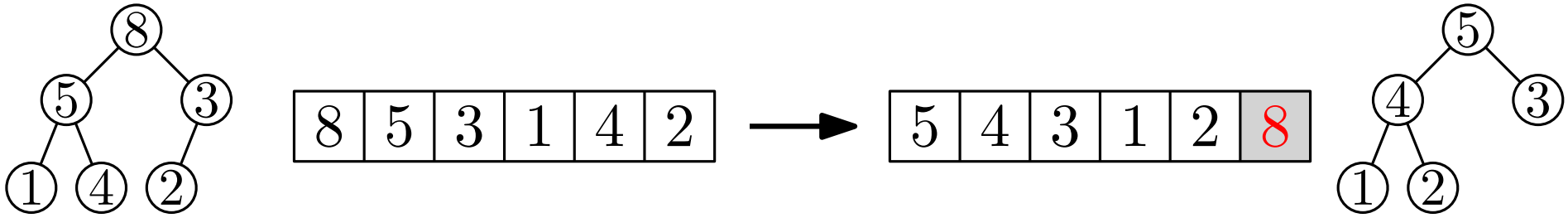
```
std::pop_heap(vec.begin(), vec.end());
```



(max-)Heaps

- Extracting the maximum key $O(\log n)$

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



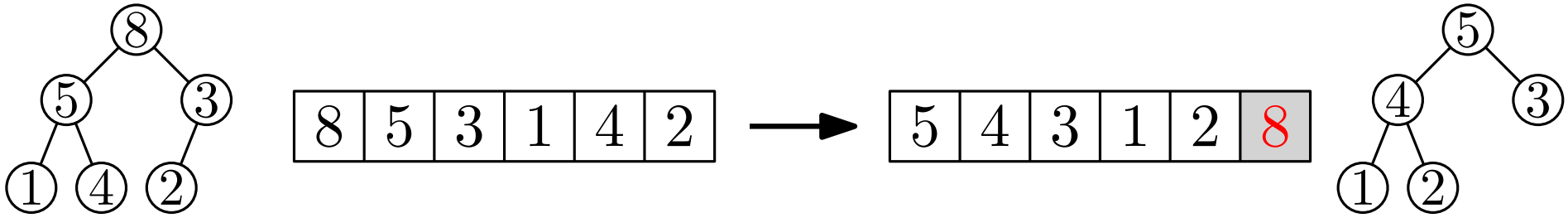
Example: Heapsort

```
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());
```



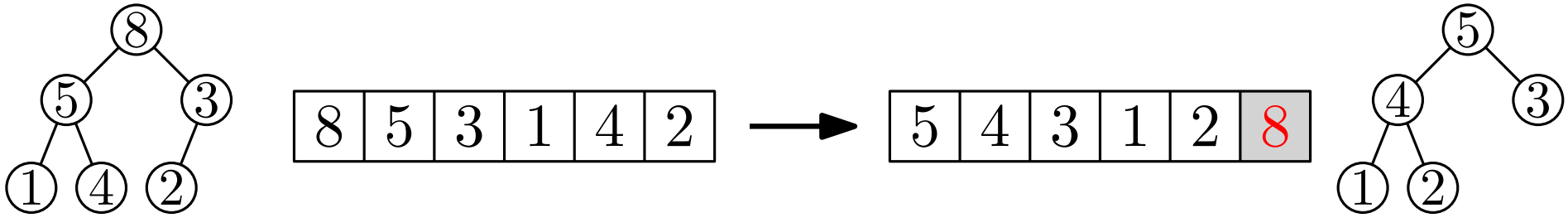
Example: Heapsort

```
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

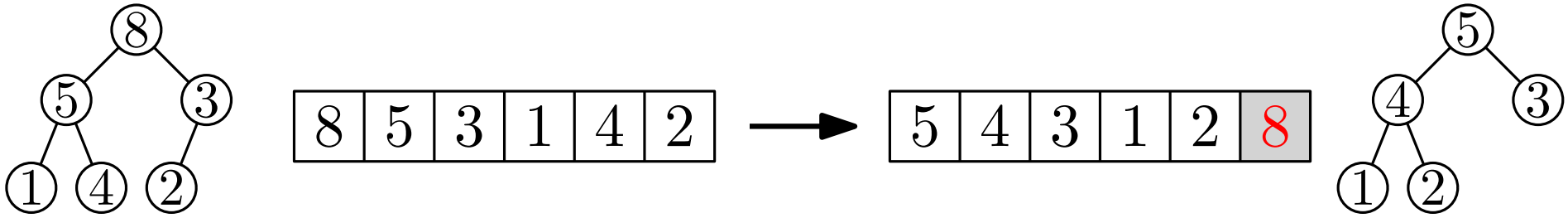
```
std::vector<int> vec { 4, 1, 6, 2, 5, 3};

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

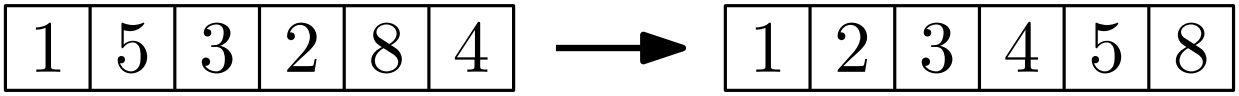
```
std::vector<int> vec { 4, 1, 6, 2, 5, 3};

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());
```



- Stable sorting $O(n \log^2 n)$ comparisons

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

- First k elements $\approx O(n \log k)$ comparisons

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