Chapter 10
Priority Queues and Heaps

10.1 Introduction to Priority Queues

So far, we have covered two types of container adaptors, stacks and queues, that allow data to be accessed based on the order of insertion. In a stack, elements are accessed in last-in, first-out (LIFO) order. In a queue, elements are accessed in first-in, first-out (FIFO) order. However, there are situations where retrieving data isn't as simple as taking out the oldest or newest element in a container. For example, if you were implementing a program that simulated an emergency call center, you would want to dispatch the calls in order of how urgent they are, not by the order in which the calls were received. Similarly, if you wanted to implement a matching algorithm for a stock exchange, you would want to match the highest price a buyer is willing to pay for an asset with the lowest price a seller is willing to sell it for. The trades that are conducted are not determined by the order in which buyers and sellers enter the market, but rather the prices they are willing to buy or sell for.

In these examples, the order of data access is based on some priority value. For the call center, the priority of a call is determined by its urgency. For the stock exchange, the priority of a trader is determined by the price at which they are willing to buy or sell stock.

To access data based on a priority value, we would need a container that stores elements in a way such that the element with the highest priority is always easily accessible. A container type that provides this functionality is the priority queue. A **priority queue** is a data structure that allows the user to extract or pop the element with the highest priority in the container, where priority is determined by a value that can be compared (e.g., using `operator<`). The interface of a priority queue is as follows:

Function	Behavior
`.push(val)`	Pushes the element `val` into the priority queue
`.pop()`	Removes the element with the highest priority from the priority queue
`.top()`	Returns a reference to the element with the highest priority
`.size()`	Returns the number of elements in the priority queue
`.empty()`	Returns whether the priority queue is empty

For instance, if you pushed the elements 1, 5, 2, 8, and 4 into a priority queue, where priority is determined by an element's value (with larger numbers have higher priorities), calling `.pop()` on this priority queue would remove the element 8, since it has the highest value.

How would you go about implementing a priority queue in memory? There are several different ways we can go about this, which we will discuss in this chapter. A summary of four priority queue implementation methods is shown in the table below:

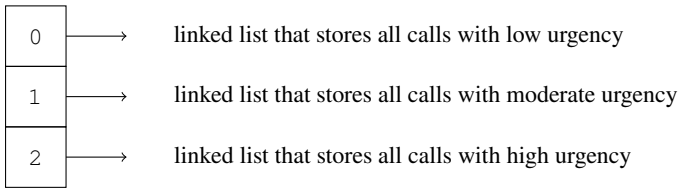
	Time Complexity of `.push()`	Time Complexity of `.pop()`
Array of Linked Lists (for priorities of small integers)	$\Theta(1)$	$\Theta(1)$
Unsorted Sequence Container	$\Theta(1)$	$\Theta(n)$
Sorted Sequence Container	$\Theta(n)$	$\Theta(1)$
Heaps	$\Theta(\log(n))$	$\Theta(\log(n))$

10.2 List and Sequence Container Implementations

※ 10.2.1 *Array of Linked Lists Implementation*

If your data can only take on a small number of priority values, you can use an array of linked lists to implement a priority queue that supports $\Theta(1)$ `.push()` and $\Theta(1)$ `.pop()`. To do so, set the array's size to the number of possible priority values and assign a linked list that holds all values of a certain priority to each index of the array.

For instance, suppose the calls in our emergency call center example can only take on one of three severity levels: 0 (not urgent), 1 (moderately urgent), and 2 (very urgent). We could implement a priority queue by using an array of linked lists that represent each severity level:



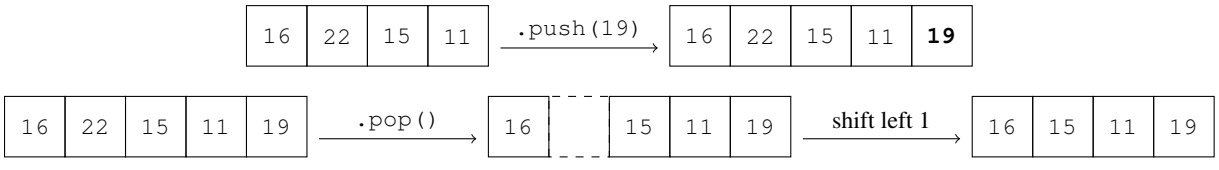
When an element is pushed into this priority queue, its priority is checked, and it is added to the correct linked list in $\Theta(1)$ time. When an element is popped from the priority queue, it is removed from the linked list associated with the highest priority (in this case, the "high urgency" linked list at index 2). However, this implementation only works if there are a fixed number of priorities that exist in your data (e.g., low, moderate, and high urgency). As an example, if we wanted to find the buyer with the highest price in our electronic stock exchange example, the priority value of an order can take on an infinite number of values (e.g., …, \$5.00, \$5.01, \$5.02, \$5.03, …) . For our linked list approach to work in this situation, we would need a separate linked list for every possible price we could encounter, which is impractical.

※ 10.2.2 *Unordered Sequence Container Implementation*

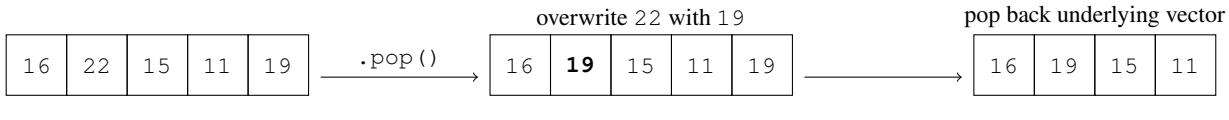
In most cases, an array of linked lists will not work since it cannot support data with many possible priority values. An alternative is to implement the priority queue using an *unordered sequence container*, or a container that stores its elements in no defined order. This is a commonly done using a vector as the underlying container for the data in a priority queue.

To `.push()` an element into a priority queue that is implemented using an unordered sequence container, the element is simply appended to the end of the container. This takes $\Theta(1)$ time.

To `.pop()` an element from a priority queue that is implemented using an unordered sequence container, a linear search is completed to find the element with the highest priority. Since every element in the underlying container must be visited to identify this highest priority element, this search takes worst-case $\Theta(n)$ time. Then, once the highest priority element is found, it is removed. If you are using a vector as the underlying container, this requires all elements after the deleted element to shift to the left by one, which also takes $\Theta(n)$ time on average.



Because the container is unordered, the `.pop()` operation can be further optimized. Instead of removing the element from the middle, which forces the container to shift all elements after the one that was removed, you can overwrite the element you want to remove with the last element, and then `.pop_back()` on the underlying vector (which, unlike erasing at an arbitrary position, always takes constant time).

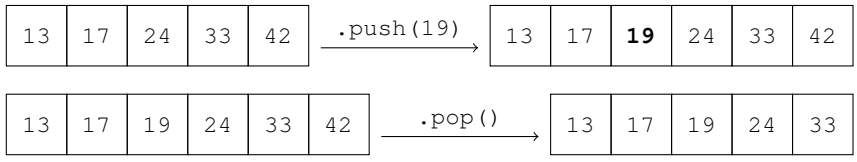


※ 10.2.3 *Sorted Sequence Container Implementation*

Another option for implementing a priority queue is to use a *sorted sequence container* to hold the data. Unlike the unordered sequence container, the sorted sequence container must be sorted at all times. This guarantees that the element with the highest priority is located at the end of the container, where removal is most efficient for a vector.

When `.push()` is called, the element being pushed in must be added at the correct position to maintain the sorted invariant. Because the underlying vector is sorted, this position can be found using *binary search* (a helpful STL function for this task is `std::lower_bound()`, which can be used to return an iterator to the correct point of insertion — this function will be covered in the next chapter). However, since elements after the point of insertion have to be shifted to make room for the new element, the time complexity of `.push()` for a sorted sequence container is $\Theta(n)$ on average.

Whenever `.pop()` is called, the element at the back of the vector is removed using the `.pop_back()` method (since the element at the back of the sorted vector must have the highest priority). This can be done in $\Theta(1)$ time.



To summarize, if you were to implement a priority queue using an *unsorted* sequence container, insertion would take $\Theta(1)$ time, but removal would take $\Theta(n)$ time. On the other hand, if you were to implement a priority queue using a *sorted* sequence container, insertion would take $\Theta(n)$ time, but removal would take $\Theta(1)$ time.

Is it better to have `.push()` or `.pop()` be more efficient? In the end, it does not matter. If you think about the life cycle of a single element in a priority queue, the total cost involved with inserting and removing that element is $\Theta(n)$ regardless of whether you use an unsorted or sorted sequence container. This is because, in most cases, elements that are added to a priority queue will eventually be removed. The only difference between the two implementations is *when* the $\Theta(n)$ operation takes place.

That being said, a $\Theta(n)$ complexity for insertion or removal is not ideal, and there are ways to do better. In the next section, we will introduce the concept of a *heap*, which can be used to improve the time complexities of these priority queue operations.

10.3 Binary Heaps

※ 10.3.1 *Heap Definitions*

A **binary heap** is a data structure that can be used to implement a priority queue that supports worst-case $\Theta(\log(n))$ insertion and removal. Since both insertion and removal can be done in $\Theta(\log(n))$ time, the entire life cycle of any element going in and out of a binary heap priority queue is $2 \times \Theta(\log(n))$, which is also $\Theta(\log(n))$. This makes the binary heap implementation of a priority queue more efficient than both sequence container implementations we covered earlier (where pushing and popping takes $\Theta(n)$ time).

Before we go over how heaps work, we will need to introduce the concept of a **tree**. A tree is a graph (i.e., a set of nodes connected by edges) that is connected without cycles (i.e., there are no unreachable vertices or circular paths that begin and end at the same vertex). In a tree, any two nodes are connected by single, *unique* shortest path. The following terminology can be used to describe different components of a tree:

- A **parent** is an immediate ancestor of a given node (i.e., the node one level above a given node).
- A **child** is an immediate descendant of a given node (i.e., the node(s) one level below a given node).
- An **ancestor** is any node closer to the root that is along a connected path upwards from a given node.
- A **descendant** is any node further from the root that is along a connected path downwards from a given node. If you want to find the descendants of any node, simply treat the given node as the root of a subtree — the subtree would contain all the descendants of that node.
- A **root node** is the "topmost" node in a tree. The root is the common ancestor of all nodes in the tree.
- A **leaf node** is a node *without* children.
- An **internal node** is a node *with* children (i.e., not a leaf node).

The **size** of a tree is the number of nodes in the tree. For an empty tree, the size is 0. For a non-empty tree, the size of the tree is the size of the root's left child + the size of the root's right child + 1 (for the root itself).

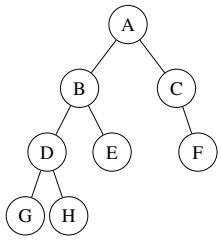
The **height** of a node is defined recursively as the maximum height of the node's children, plus 1. By definition, the height of a leaf node is 1. An alternative definition of height is the maximum distance needed to get from a node to a `nullptr` by moving downwards through the tree.

The **depth** of a node is the opposite of its height. Unlike height, which measures the distance between a node and a `nullptr` at the bottom of the tree, the depth measures the distance between a node and the root. The depth of a node is the depth of its parent, plus 1.

A **binary tree** is a special type of tree where each node has at most two children. Binary trees can be further split into the following categories:

- A **proper** or **full** binary tree is a binary tree where every node either has 0 or 2 children. In other words, all non-leaf nodes in a proper binary tree have 2 children.
- A **complete** binary tree is a tree in which every depth, except possibly the last, is completely filled. If the last depth (the bottom row) is not completely filled, then all nodes that do exist at that depth must be filled from left to right with no gaps.

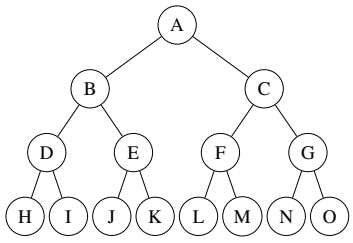
Consider the following tree below. This tree is a binary tree because each node has at most two children.



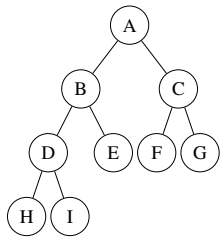
The following statements are true regarding this tree:

- Node A is the root node of the tree.
- Nodes E, F, G, and H are leaf nodes because they have no children.
- Nodes A, B, C, and D are internal nodes because they do have children.
- Nodes G and H are the children of node D.
- Node B is the parent of nodes D and E.
- Nodes A and B are the ancestors of node D.
- Nodes D, E, G, and H are the descendants of node B.
- There are 8 nodes in the tree, so the size of the tree is 8.
- The heights of all the leaf nodes (E, F, G, and H) are each 1. The heights of nodes C and D are 2, the height of node B is 3, and the height of node A is 4.
- The depth of the root node A is 1, and the depths of the other nodes increase by one for each level. Nodes B and C have depths of 2. Nodes D, E, and F have depths of 3. Nodes G and H have depths of 4.

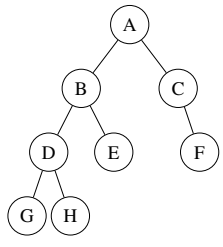
Now, let's look at a few binary trees that depict the property of completeness. The tree below is complete, since every depth is completely filled.



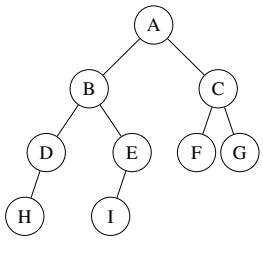
The following binary is also complete. Notice that the lowest level of the tree is not completely filled to capacity. This is okay; per the definition of a complete binary tree, the lowest level may be unfilled as long as all the elements on that level are located as far left as possible. Since this is true for the tree below, the completeness property still holds.



The following tree is *not* complete. This is because the second-to-last level is not completely filled (there is a gap between E and F, since C does not have a left child). A complete tree only allows the final depth of the tree to be unfilled, so this tree violates the completeness property.



The following tree is also *not* complete. Even though all but the last level is completely filled, the nodes in the last level are *not* located as far left as possible, which violates the completeness property. There is a gap between H and I, as D has no right child.

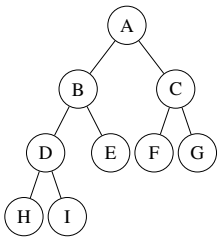


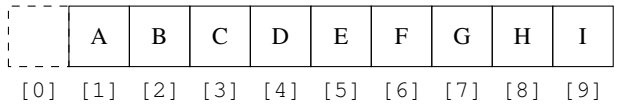
How would a binary tree be represented in memory? One method is to use a node and pointer-based approach, where each `Node` object stores data along with pointers to the nodes of its children:

```
1  template <typename T>
2  struct Node {
3    T data;              // data of node
4    Node* left;          // pointer to left child
5    Node* right;         // pointer to right child
6  };
```

The pointer-based approach is efficient for moving down a tree from parent to child. However, this approach also uses a bit of additional memory, since each node will have to store pointers to its children in addition to its data.

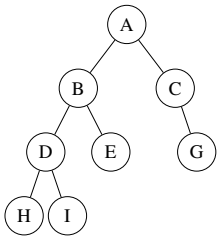
The pointer-based approach is not the only possible approach. It turns out that there is another method of storage that works well for *complete* binary trees. The completeness property allows these trees to be elegantly collapsed into a form that can be efficiently stored in a vector. To convert a complete binary tree into a vector, we append the first row of the tree to the vector, followed by the second row, followed by the third, and so on. The figure below depicts what a complete binary tree would look like after it is flattened into a vector.

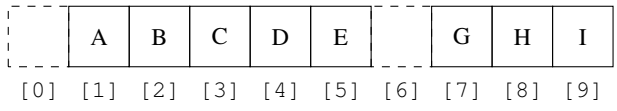




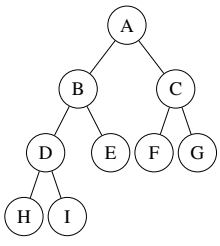
Note: There is a reason why index 0 is empty... this will be explained soon.

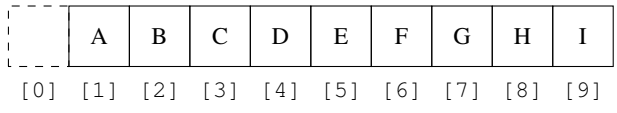
The completeness property ensures efficient vector storage, since it guarantees that there are no gaps in the array. For instance, if node F were missing from the tree (i.e., the tree would not longer be complete), then there would be nothing at index 6. This is inefficient, since the presence of empty indices can make the size of a vector much larger than the number of nodes, wasting memory.





Why is index 0 of the array empty? This is because having the root at index 1 simplifies the math we would need to do to identify the children and parent of a node when given its index. Let's analyze how this works. Consider the following tree, as reproduced from the previous example:





If we use 1-indexing and store the root at index 1, the following properties are true:

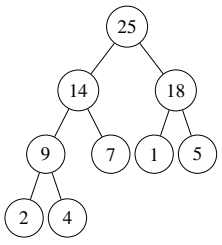
- Given an index i of a node in the tree, the index of that node's parent is $i/2$.
  - In the tree above, the parent of node F (index 6) is node C (index $6/2 = 3$).
  - In the tree above, the parent of node G (index 7) is node C (index $7/2 = 3$, using integer division).
- Given an index i of a node in the tree, the index of its left child is $2i$.
  - In the tree above, the left child of node B (index 2) is node D (index $2 \times 2 = 4$).
- Given an index i of a node in the tree, the index of its right child is $2i + 1$.
  - In the tree above, the right child of node B (index 2) is node E (index $2 \times 2 + 1 = 5$).
- A node is a leaf node if its index i is greater than $n/2$, where n is the size of the tree.
  - In the tree above, node E is a leaf node because its index, 5, is greater than $n/2 = 9/2 = 4$.
  - In the tree above, node D is not a leaf node because its index, 4, is not greater than $n/2 = 9/2 = 4$.
- A node is an internal node if its index i is less than or equal to $n/2$, where n is the size of the tree.
  - In the tree above, node D is an internal node because its index, 4, is less than or equal to $n/2 = 9/2 = 4$.
  - In the tree above, node E is not an internal node because its index, 5, is not less than $n/2 = 9/2 = 4$.

Remark: Even though we store a dummy element at index 0 in these examples, this actually isn't the best way to do things (as it wastes space and makes certain invariants a bit more confusing; e.g., `pq.size() == pq.data.size() - 1`). A better option would be to write a few `private` member functions that can be used to translate 1-indexing into a 0-indexed vector, as shown below. For simplicity, however, the remaining code in this section will assume a dummy element at index 0.

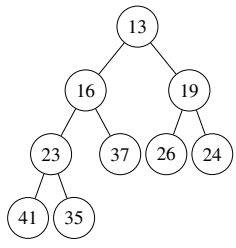
```
1  T& get_element(size_t idx) {
2    return data[idx - 1];
3  } // get_element()
4
5  const T& get_element(size_t idx) const {
6    return data[idx - 1];
7  } // get_element()
```

A tree is considered to be **heap-ordered** if, for every node in the tree, each of its children has a priority that is *less than or equal* to its parent. In a heap-ordered tree, no node has a priority greater than the root.

Heap-ordered binary trees can be separated into two categories. A max heap-ordered tree, or a **max-heap**, is a binary tree where the value of each node is greater than or equal to the values of its children (i.e., larger values have higher priority). A min heap-ordered tree, or a **min-heap**, is a tree where the value of each node is less than or equal to the values of its children (i.e., smaller values have higher priority). The following are examples of max and min heap-ordered trees:



Max-Heap



Min-Heap

This leads us to the formal definition of a binary heap. A binary heap is a binary tree-based data structure that has two properties. *Both properties must be satisfied for a binary tree to be a binary heap!*

1. The tree must be *complete*.
2. The contents of the tree must be *heap-ordered*.

A binary heap can be used to efficiently retrieve the highest priority element in a collection of data. Max-heaps allow for easy access to the largest value, while min-heaps allow for easy access to the smallest value. Because of this, binary heaps can be efficiently used as the underlying structure of a priority queue.

In this class, binary heaps will be implemented using a vector. The vector approach is preferable to the pointer-based approach in that it uses less memory (i.e., no need to store pointers to children) without sacrificing any of the functionality or performance of the heap.

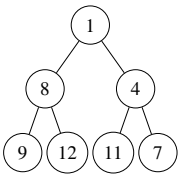
Example 10.1 Consider the following vectors of data. Which of the following are min-heaps? Which of the following are max-heaps? Assume that the root is the first element in the vector.

a. `[1, 8, 4, 9, 12, 11, 7]`
b. `[3, 4, 5, 7, 12, 11, 8, 6, 13]`
c. `[13, 10, 6, 8, 7, 4, 2, 8, 1]`

To approach problems like these, draw out the tree first. If the root element is the smallest in the vector, check if the remaining elements satisfy a min-heap by making sure that no child has a value smaller than that of its parent. On the other hand, if the root element is the largest in the vector, check if the remaining elements satisfy a max-heap by making sure that no child has a value larger than that of its parent.

---

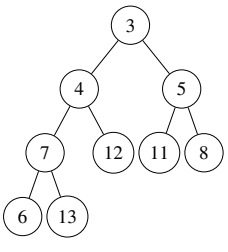
a. The vector `[1, 8, 4, 9, 12, 11, 7]` can be converted to the following tree:



This is a min-heap, since the value of each parent is less than or equal to the values of its children. 1 is less than or equal to 8 and 4, 8 is less than or equal to 9 and 12, and 4 is less than or equal to 11 and 7.

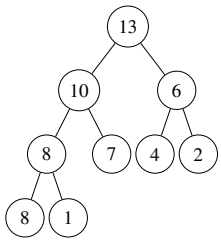
---

b. The vector `[3, 4, 5, 7, 12, 11, 8, 6, 13]` can be converted to the following tree:



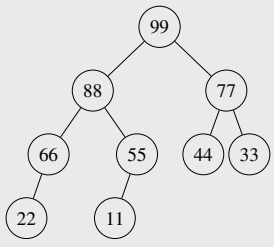
This is neither a min-heap nor a max-heap. Notice that 7 has children 6 and 13. Although the rest of the elements follow the min-heap ordering, this heap is not a min-heap because 7 is not smaller than or equal to 6.

c. The vector `[13, 10, 6, 8, 7, 4, 2, 8, 1]` can be converted to the following tree:



This is a max-heap, since the value of each parent is greater than or equal to the values of its children. 13 is greater than or equal to 10 and 6, 10 is greater than or equal to 8 and 7, 6 is greater than or equal to 4 and 2, and 8 is greater than or equal to 8 and 1.

Example 10.2 Is the following a valid max-heap?



The answer is no. Even though this tree is heap-ordered, it is not complete. Remember that completeness is one of the necessary requirements of a binary heap!

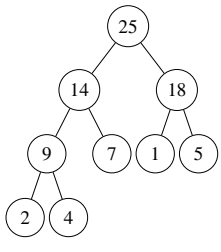
※ 10.3.2 *Fix Up*

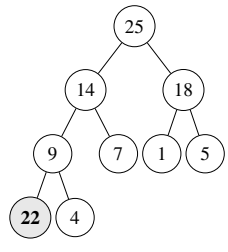
What if the priority of an element in the heap is modified? When this happens, the contents of the heap may need to be reordered to ensure that the heap-ordered criteria is still met.

There are two ways an element in the heap can be modified: the priority of the element can either be *increased* or *decreased*. If the priority of an element is increased, you will need to use the **fix up** approach to bring the element closer to the top of the heap. To do this, continuously swap the modified element with its parent until either:

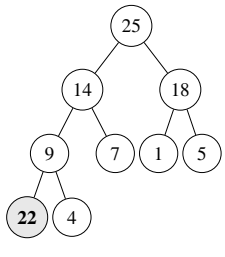
- you reach the root (this happens if the modified element ends up with a priority that is the highest in the heap).
- you reach a parent with a higher or equal priority.

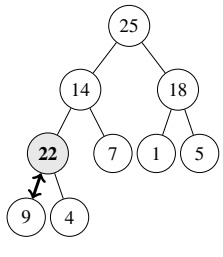
The following process illustrates what happens when the element 2 in the following max-heap gets updated to 22.

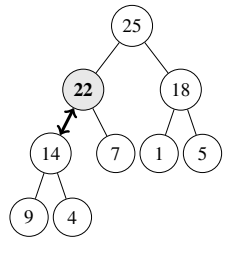




To fix the binary heap, the fix up method is used. The new node 22 is continuously swapped with its parent until it either reaches the root or a parent that is greater than or equal to 22, whichever comes first.







Recall that, in the vector-based implementation of a binary heap, the parent of an element at index i is located at index $i/2$. The following function uses this information to implement fix up, given a vector of data and the index of the element whose priority was increased.

```
1  // this function returns whether val1 has a lower priority than val2
2  template <typename T>
3  bool comp_priority_less(const T& val1, const T& val2);
4
5  template <typename T>
6  void fix_up(std::vector<T>& data, size_t index) {
7    // swap modified element with parent until root is reached or a parent
8    // with a greater than or equal priority is found
9    while (index > 1 && comp_priority_less(data[index / 2], data[index])) {
10     swap(data[index], data[index / 2]);
11     index /= 2;
12   } // while
13 } // fix_up()
```

What is the worst-case time complexity of the fix up operation? In the worst case, the number of swaps you need to perform is equal to the number of levels in the tree (which happens if an element is swapped from the bottom-most level to the root). The number of levels in the tree is approximately $\log(n)$ if n is the size of the heap, so the time complexity of fix up is also $\Theta(\log(n))$.

Why is the number of levels in a heap $\Theta(\log(n))$ rather than $\Theta(n)$? This is because heaps enforce the completeness property. A complete tree with n nodes must have $\Theta(\log(n))$ levels, as every level but the last must be completely filled. The only way to have $\Theta(n)$ levels is to have a sticklike tree where certain nodes only have one child, which would not be complete.

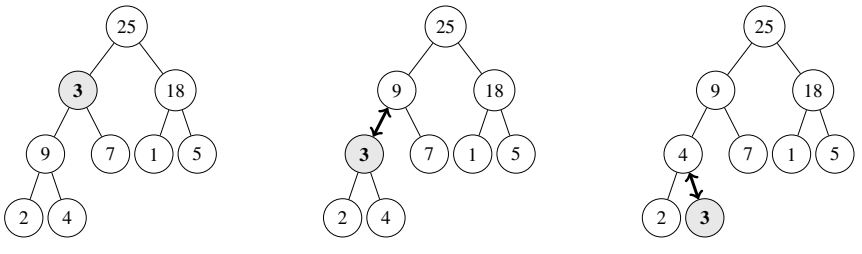
※ 10.3.3 *Fix Down*

If the priority of an element in the heap is decreased, you will need to use the **fix down** approach to lower that element's position in the heap. To do this, continuously swap the modified element with the child that has the highest priority until either:

- you reach the bottom of the heap.
- you reach a position where no child has a higher priority.

The following process illustrates what happens when the element 14 in the following max-heap gets updated to 3:

To fix the binary heap, the fix down method is used. The new node 3 is continuously swapped with the child of higher priority until it either reaches the bottom level of the heap or a position where no child has a higher priority.



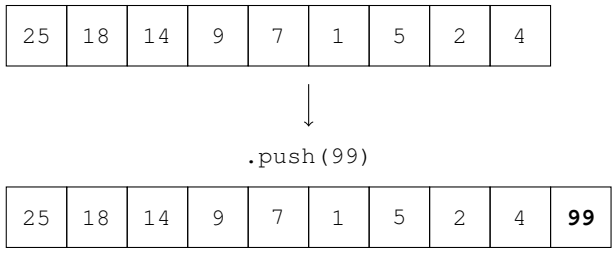
The code involved with the fix down approach using a vector is shown below. This implementation is slightly more complicated than the fix up approach since it requires checks for whether a node actually has a left or right child.

```
1  // this function returns whether val1 has a lower priority than val2
2  template <typename T>
3  bool comp_priority_less(const T& val1, const T& val2);
4
5  template <typename T>
6  void fix_down(std::vector<T>& data, size_t index) {
7    size_t heap_size = data.size() - 1;
8    while (2 * index <= heap_size) {
9      // initialize highest priority child to left child
10     size_t larger_child = 2 * index;
11     // if right child has higher priority, set larger child to right child
12     if (larger_child < heap_size &&
13         comp_priority_less(data[larger_child], data[larger_child + 1])) {
14       ++larger_child;
15     } // if
16     // if children all have lower priority, the heap is restored so break out of the loop
17     if (comp_priority_less(data[larger_child], data[index])) {
18       break;
19     } // if
20     // otherwise, swap the modified element with the largest child
21     std::swap(data[index], data[larger_child]);
22     // set new index value for next iteration of loop
23     index = larger_child;
24   } // while
25 } // fix_down()
```

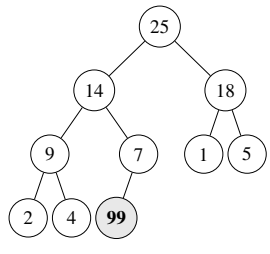
Once again, the time complexity of fix down depends on the number of swaps that need to be done. Since there are $\Theta(\log(n))$ levels in a complete binary tree, the number of swaps required is $\Theta(\log(n))$ at worst. Thus, the worst-case time complexity of fix down is $\Theta(\log(n))$.

※ 10.3.4 *Inserting and Removing Elements*

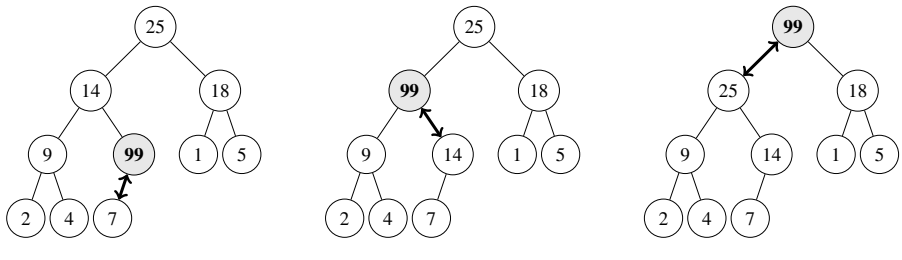
Now, let's consider what happens when we try to insert an element into a binary heap. Since a heap's underlying vector is not guaranteed to be sorted, we cannot do a binary search to find the correct position of insertion (much like with the sorted sequence container). Since we do not immediately know where to insert the element, we will first push the element to the back of the vector using `.push_back()` (as vectors support efficient insertion at the back). An example is shown below:



If we look at the contents of the modified heap, adding an element to the back of the underlying vector essentially adds the element to the next available position in the complete binary tree (i.e., the next open position on the bottom level when filling out from left to right).



After inserting the new element, we will have to move it to the correct position to ensure that the tree remains heap-ordered. Because the new element was added to the bottom of the heap, we want to swap it upwards to the correct position. This can be done using the fix up approach covered earlier: the newly added node is repeatedly swapped with its current parent until it is in its correct position.



In summary, to push an element into a priority queue that is implemented using a vector-based binary heap:

1. Push the element to the back of the underlying vector that stores the data of the binary heap.
2. Call fix up on the element that was added (to swap it up to the correct position).

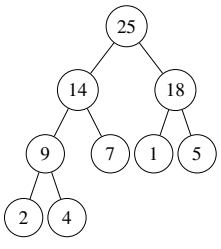
```
1  template <typename T>
2  void push(std::vector<T>& data, const T& val) {
3    data.push_back(val);
4    fix_up(data, data.size() - 1);
5  } // push()
```

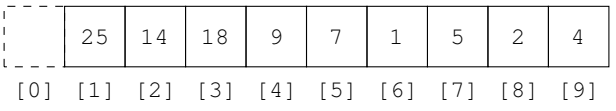
What is the time complexity of pushing an element into this priority queue? Since `.push()` inserts an element at the back of the vector (which takes $\Theta(1)$ time) and then calls `fix_up()` (which takes $\Theta(\log(n))$ time), the time complexity of `.push()` is $\Theta(1+\log(n))$, or $\Theta(\log(n))$.

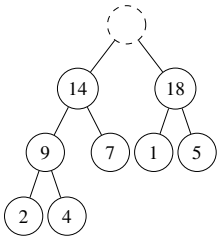
The process of removing an element works similarly. To pop an element from the heap, you would remove the element with the highest priority. This element is located at the root of the binary heap, or index 1 of its underlying vector (assuming a dummy at index 0). The intuitive approach would be to simply erase the element at index 1 of the vector when `.pop()` is called:

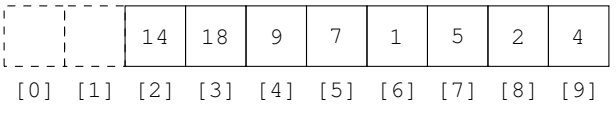
```
1  template <typename T>
2  void pop(std::vector<T>& data) {
3    data.erase(data.begin() + 1);   // NOTE: this implementation of pop is incorrect!
4  } // pop()
```

However, this approach does not work! This is because the rest of the heap may be invalidated. Consider the example below: suppose we wanted to pop the highest priority element off the following max-heap, and we did so by just erasing the first element in the heap's underlying vector:

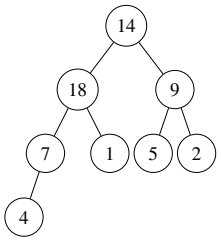


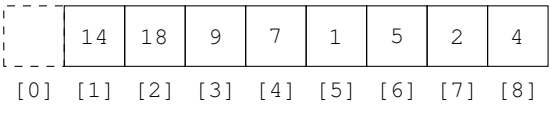




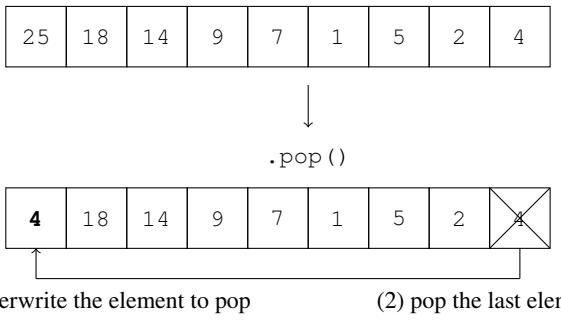


After erasing the element at index 1, all of the remaining elements will need to shift one position to the left to ensure that the new root node is positioned at index 1. This ends up being a $\Theta(n)$ operation, which is not what we want! Furthermore, the heap ends up looking like this after the shifting process is complete:

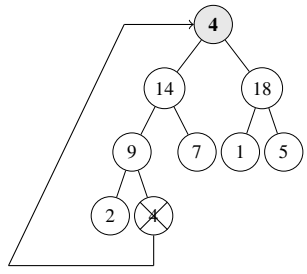




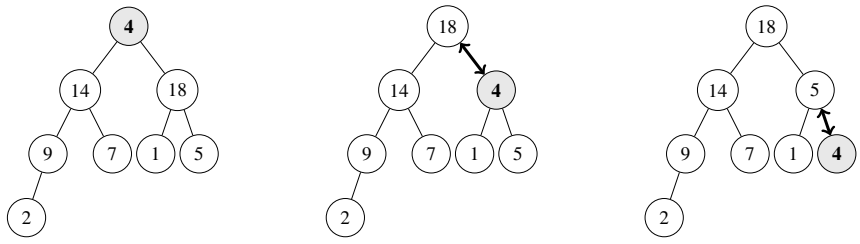
This is not a valid heap! Notice that 14 ends up at the root, but it is not the element with the highest priority (that would be 18). Thus, simply erasing the first element does not guarantee that the remaining elements will form a valid heap. A better approach would be to *overwrite* the root with the last element in the heap, and then pop off the last element. Let's look at how this works:



Looking at this from a tree perspective, this procedure takes the last element in a complete tree (the rightmost node on the bottom level of the tree) and moves it to the root position:



Once again, the heap-ordered invariant is broken, since 4 was moved to the root. However, unlike the previous approach of just erasing the value at index 1, this method ensures that the remaining elements are still heap-ordered (since 4 was the only value that changed its position in the heap). This allows us to fix the heap by simply calling fix down on the node we swapped to the root, repeatedly swapping it with its highest priority child until it is in its correct position.



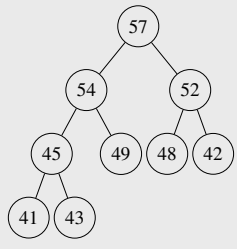
In summary, to pop an element off a priority queue that is implemented using a vector-based binary heap.

1. Overwrite the root element with the element at the back of the underlying vector.
2. Pop back the last element of the vector.
3. Call fix down on the new root element to bring it down to the correct position.

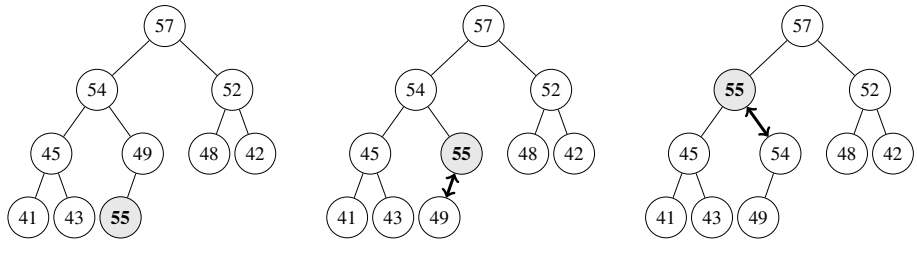
```
1  template <typename T>
2  void pop(std::vector<T>& data) {
3    data[1] = data.back();
4    data.pop_back();
5    fix_down(data, 1);
6  } // pop()
```

Since `.pop()` overwrites an element and calls `.pop_back()` on the underlying vector (both of which take $\Theta(1)$ time) and then calls `fix_down()` (which takes $\Theta(\log(n))$ time), the overall time complexity of `.pop()` is $\Theta(1+\log(n))$, or just $\Theta(\log(n))$.

Example 10.3 Consider the following max-heap. After inserting the value 55 into this max-heap and fixing the heap invariant, what is the final array representation of this max-heap?

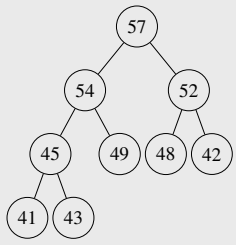


Since this question involves adding an element to the heap, the new element is inserted at the very back, and fix up is continuously called on that element until it is in the correct position. In this case, 55 is added as the left child of 49 and then fixed up to its position.

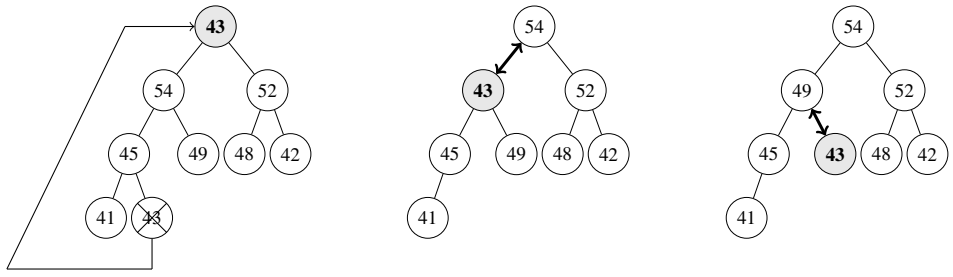


The final array representation of the max-heap is simply the elements of the heap in level order (i.e., left to right across each level of the tree). In this case, the array representation of the heap is `[57, 55, 52, 45, 54, 48, 42, 41, 43, 49]`.

Example 10.4 Consider the following max-heap. After popping off the largest element, 57, and fixing the heap invariant, what is the final array representation of this max-heap?



Since this question involves removing an element from the heap, you should overwrite 57 with the last element in the heap, which is 43. Then, you would pop the element at the back and call fix down continuously to move 43 down to the correct position. In this case 43, is swapped with its largest child until none of its children have a higher priority than it.



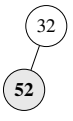
The final array representation of the max-heap is a simply the elements of the heap in level order. In this case, the array representation of the heap is `[54, 49, 52, 45, 43, 48, 42, 41]`.

Example 10.5 You are given an empty min-heap priority queue, and you push the following values into the priority queue in this order: 32, 52, 11, 23, 59, 10. What does the underlying array structure look like after all of these values are inserted?

Remember that inserting an element into a heap involves adding the element to the back of the heap and continuously calling fix up on that element until it is in the correct position. This process for the given sequence of insertions is detailed below. Note that this problem deals with a min-heap, and not a max-heap!

1. Insert 32 into the heap. Since the heap is currently empty, this insertion is trivial.

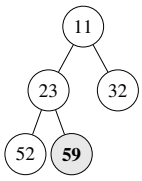
2. Insert 52 into the heap. In this case, 52 is added in the next open spot, which is the left child of 32. Since 52 has a lower priority than 32, the min-heap is still valid.



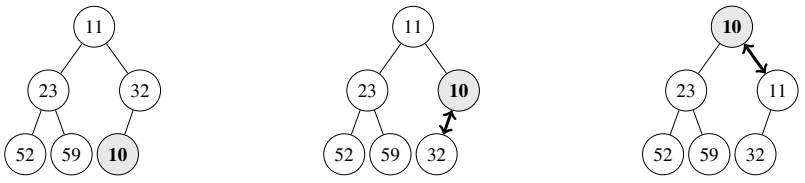
3. Insert 11 into the heap. In this case, 11 is added in the next open spot, which is the right child of 32. Since 11 has a higher priority than 32, the min-heap is no longer valid, and we need to fix it. This is done by calling fix up on 11, which causes it to swap with 32.

4. Insert 23 into the heap. In this case, 23 is added in the next open spot, which is the left child of 52. Since 23 has a higher priority than 52, the min-heap is no longer valid, and we need to fix it. This is done by calling fix up on 23, which causes it to swap with 52.

5. Insert 59 into the heap. In this case, 59 is added in the next open spot, which is the right child of 23. Since 59 has a lower priority than 23, the min-heap is still valid.



6. Insert 10 into the heap. In this case, 10 is added in the next open spot, which is the left child of 32. Since 10 has a lower priority than 32, the min-heap is no longer valid, and we need to fix it. This is done by calling fix up on 10, which causes it to swap with 32 and 11.



We are now done. The final array representation of the min-heap is `[10, 23, 11, 52, 59, 32]`.

10.4 Heapify

※ 10.4.1 *Top-Down and Bottom-Up Heapify*

If you were given an array of values in an arbitrary order, how can you turn it into a heap? The simplest approach would be to push the values into a binary heap one at a time, fixing the heap with every operation (like in the previous example). However, there are several inefficiencies with this method. First, the time complexity of this process is $\Theta(n\log(n))$, since n pushes are needed, each taking $\Theta(\log(n))$ time. Second, this process uses additional $\Theta(n)$ memory, since you would have to push the contents of the original container into a new container for the heap.

It turns out that there is a better way to turn a container of values into a heap using $\Theta(n)$ time and $\Theta(1)$ auxiliary space — this process is known as **heapify**. Heapify relies on the idea that, if fix up and fix down can be used to fix the position of any element in a heap, you should be able to turn any arbitrary container of data into a heap by just iterating through its contents and calling fix up or fix down on each element.

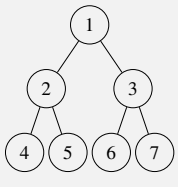
Intuitively, this leads us to four possibilities, each of which is summarized below. We can choose whether we want to call fix up or fix down on each element, and we can choose the direction of iteration. In the explanations below, forward iteration means starting from index 1 of the heap's underlying vector (the root) and iterating to the end; backward iteration means starting at the back of the heap's underlying vector (the rightmost leaf on the lowest level) and iterating to the front.

- **Heapify Idea #1:** Traverse in the *forward* direction, from the top of the heap to the bottom, calling *fix up* on each element encountered.
- **Heapify Idea #2:** Traverse in the *forward* direction, from the top of the heap to the bottom, calling *fix down* on each element encountered.
- **Heapify Idea #3:** Traverse in the *backward* direction, from the bottom of the heap to the top, calling *fix up* on each element encountered.
- **Heapify Idea #4:** Traverse in the *backward* direction, from the bottom of the heap to the top, calling *fix down* on each element encountered.

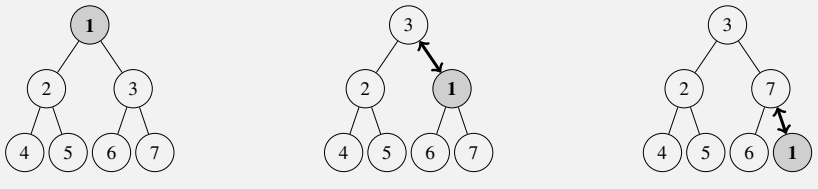
Upon first glance, all four methods seem to work equally well. However, things are not as simple as they seem! It turns out that only two of the four implementations work, and of these two, only one can be done in worst-case $\Theta(n)$ time.

Let's first get rid of the two that do not work, ideas #2 and #3. These two methods may actually produce invalid heaps. Let's look at examples that break these two methods.

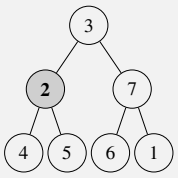
Proof: Traversing a heap in the forward direction (from top to bottom) and calling fix down on each element does *not* always produce a valid heap. Suppose you are given the array `[1, 2, 3, 4, 5, 6, 7]`, and you wanted to create a max-heap. Moving from the top of this heap (1) to the bottom (7) and calling fix down on each element produces an invalid heap.

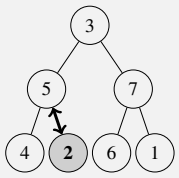


First, we call fix down on the element at index 1, which in this case is 1.

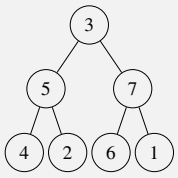


Next, we call fix down on the element at index 2, which in this case is 2.



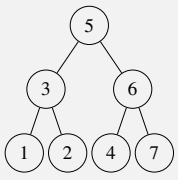


Next, we call fix down on the element at index 3, which in this case is 7. 7 is already larger than its children, so it doesn't change its position. Similarly, calling fix down on the elements 4, 2, 6, and 1 does not do anything either, as those nodes are leaves, which do not have any children. Thus, our final heap looks like this:

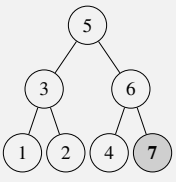


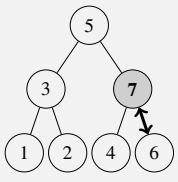
This is *not* a valid max-heap, since 3 is smaller than 5 and 7. The problem with this approach is that the smallest element, 1, was swapped with 3, forcing 3 to the top of the heap. However, since we are iterating in the forward direction while only fixing downwards, the 3 is ignored and ends up stuck at the top, in the incorrect position.

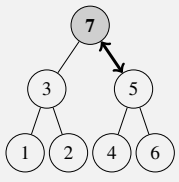
Proof: Traversing a heap in the backward direction (from bottom to top) and calling fix up on each element does *not* always produce a valid heap. To show this, suppose you are given the array `[5, 3, 6, 1, 2, 4, 7]`, and you wanted to create a max-heap. Moving from the bottom of this heap (7) to the top (5) and calling fix up on each element produces an invalid heap.



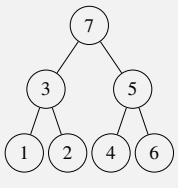
First, we call fix up on the last element, which in this case is 7.







Next, we call fix up on the elements 4, 2, 1, 5, 3, and 7, in this order. You can work through this process on your own, but none of these calls actually change the position of any nodes in the tree (since they are all smaller than their parent's value). Thus, our final heap looks like this:



This is *not* a valid max-heap, since 6 is larger than 5. The problem with this approach is that the largest element, 7, was swapped with 6, forcing 6 to the back of the heap. However, since we are moving toward the top while only fixing upwards, the 6 is ignored and ends up stuck at the bottom, in the incorrect position.

As a result, to correctly heapify a container of values, you must fix in the direction you come from! If you are traversing the heap from top to bottom, you must call fix *up* on every element in the heap. If you are traversing the heap from bottom to top, you must call fix *down* on every element in the heap.[1]

[1] A good analogy to think of is a layer of fresh snow on the ground. As you walk through the snow, you leave footprints *behind* you, while the snow in front of you remains untouched. If you want to "fix" the snow, you would need to clean up the footprints behind you. The same applies to heapify; if you are moving through the heap in one direction, you must fix the elements in the opposing direction, since that is the direction in which a mess could have been made!

To summarize the section up to this point, we discussed the heapify process, which can be used to turn a container of data into a valid heap in-place. Even though we introduced four possible implementations, only two of the four can be used to heapify a container and guarantee a valid heap. These two methods are defined as follows:

- **Top-Down Heapify:** Traverse from the top of the heap (the root) to the bottom (the leaves), calling fix up on each element encountered.
- **Bottom-Up Heapify:** Traverse from the bottom of the heap (the leaves) to the top (the root), calling fix down on each element encountered.

What is the time complexity of the heapify process? As alluded to earlier, these two heapify methods actually have *different* time complexities in the worst case. Given a vector of size n, the top-down heapify method has a worst-case time complexity of $\Theta(n\log(n))$. The explanation for this is straightforward: fix up is called n times, and since each call to fix up takes worst-case $\Theta(\log(n))$ time, the overall worst-case complexity of top-down heapify is $\Theta(n \times \log(n))$. However, things get a bit bizarre with the bottom-up heapify method. Even though the bottom-up method makes n calls to fix down, and each fix down takes worst-case $\Theta(\log(n))$ time, the worst-case time complexity of the entire bottom-up process is actually $\Theta(n)$. Why is this the case?

The reason is that *not every call to fix up or fix down does the same amount of work*. Fix up, for example, continuously swaps an element with its parent until it is in the correct position. Thus, the amount of work required for a single fix up call is greatest when the element to fix up is at the bottom of the heap, since this may potentially result in $\Theta(\log(n))$ swaps (i.e., one for each level of the tree if the element at the bottom is swapped all the way to the top). However, the work involved in a fix up call decreases as you move toward the top of the heap, since there are fewer levels that an element can potentially move up. Calling fix up on the root is essentially a $\Theta(1)$ operation, since the root cannot be fixed up any higher (as it is already at the top of the heap).

Fix down, on the other hand, continuously swaps an element with its highest priority child until it reaches the correct position. Thus, the amount of work required for a single fix down call is greatest when the element to fix down is at the top of the heap, since this may result in $\Theta(\log(n))$ swaps (which happens if the top node is swapped all the way to the bottom). However, the work involved in a fix down call decreases as you move toward the bottom of the heap, as there are fewer levels that an element can potentially move down. Calling fix down on a leaf node is essentially a $\Theta(1)$ time operation, since a leaf cannot be fixed down any lower (as leaves do not have any children).

This distinction is important because *there are more elements near the bottom of the heap than near the top.* In fact, given a binary heap with n nodes, at least half of them must be leaf nodes! As a result, when you heapify a heap, half of the nodes you need to fix up or down are leaf nodes. Since fixing up a leaf node is much more expensive than fixing down a leaf node, the bottom-up approach is more efficient. In fact, since calling fix down on a leaf is guaranteed to do nothing, the bottom-up heapify approach can skip half of the tree!

From a mathematical perspective, let's show that the complexity of the bottom-up approach is $\Theta(n)$. First, note that half of the nodes in a binary heap are leaf nodes. Since fix down does nothing (as leaves do not have any children), no work needs to be done for these nodes:

Number of Nodes	Max Levels to Fix (Distance to Leaf)	Max Swaps Needed
$n/2$	0	$n/2 \times 0 = 0$

Now, let's consider nodes in the heap that are directly above a leaf node. Because a parent has at most two children and binary heaps are complete, there are approximately $n/4$ nodes that are exactly one level above a leaf. If fix down were called on any of these nodes, the maximum number of levels that they could be swapped down is 1.

Number of Nodes	Max Levels to Fix (Distance to Leaf)	Max Swaps Needed
$n/4$	1	$n/4 \times 1 = n/4$

Using this same process, we know that there must be $n/8$ nodes that are two levels above a leaf, $n/16$ nodes that are three levels above a leaf, and so on. Eventually we will reach the root, which is approximately $\log(n)$ levels above a leaf. This allows us to complete the following table:

Number of Nodes	Max Levels to Fix (Distance to Leaf)	Max Swaps Needed
$n/2$	0	$n/2 \times 0 = 0$
$n/4$	1	$n/4 \times 1 = n/4$
$n/8$	2	$n/8 \times 2 = 2n/8$
$n/16$	3	$n/16 \times 3 = 3n/16$
$n/32$	4	$n/32 \times 4 = 4n/32$
…	…	…
1 (the root)	$\log(n)$	$\log(n)$

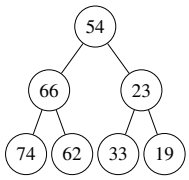
The total work completed during the entire heapify process is equal to the sum of the number of swaps needed at each level. This is equal to $n(0 + \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \frac{4}{32} + \ldots)$, which ends up simplifying to $\Theta(n)$. This is because the multiplicative term $(0 + \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \frac{4}{32} + \ldots)$ actually sums up to a constant.[2]

[2]This simplification won't be explicitly proved in this chapter since it involves some complex mathematics outside the scope of this class. You don't need to know how to get the $\Theta(n)$ — you just need to know that bottom-up heapify takes $\Theta(n)$ time.

Example 10.6 Consider the following unsorted array. Perform a $\Theta(n)$ bottom-up heapify operation to turn this array into a valid min-heap. What are the contents of the array after the heapify operation?

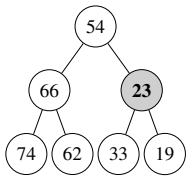
```
[54, 66, 23, 74, 62, 33, 19]
```

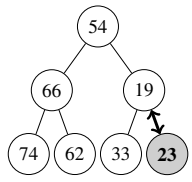
First, draw out this unsorted array as a binary tree:



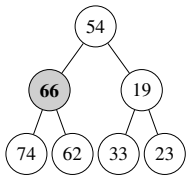
To perform a bottom-up heapify, we should start at the last element in the array and iterate to the front, calling fix down on each element we encounter. Because fix down has no effect on leaf nodes, we can skip the entire bottom row of this tree.

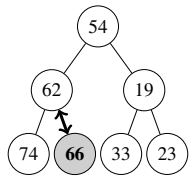
We begin calling fix down on the first node with a child at the back of the heap (index 3), which is 23.



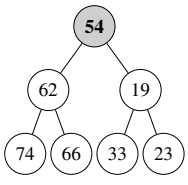


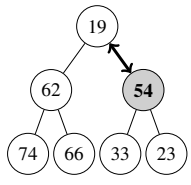
Next, we call fix down on the node at index 2, which is 66.

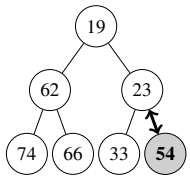




Lastly, we call fix down on the root, which is 54.







The final contents of the array are `[19, 62, 23, 74, 66, 33, 54]`.

※ 10.4.2 *STL Heapify* (✻)

It turns out that, to no surprise, the heap operations we have covered so far are already implemented in the STL. The `<algorithm>` library provides the `std::make_heap()` function, which can be used to heapify a container.

```
template <typename RandomAccessIterator, typename Compare>
void std::make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Rearranges the elements in the iterator range `[first, last)` to create a heap. The comparator `comp` is optional, and without it, a max-heap is created by default. The optional comparator can be used to create a heap whose priority is based on some other condition, where `comp(a, b)` returns `true` if `a` has a lesser priority than `b`.

The comparator is optional, and without it, `std::make_heap()` will by default heapify the given container into a max-heap. However, if a comparator `comp` is specified, the priorities of elements in the heap are determined using the following rule: for any two elements `a` and `b`, `a` has a lower priority than `b` if `comp(a, b)` returns `true`.

For instance, passing the `std::less<>` comparator into `std::make_heap()` would produce a max-heap: using the rules of determining priority, we know that if `a < b`, `a` must have a lower priority than `b` because `std::less<T>(a, b)` returns `true` if `a < b`. In contrast, passing in the `std::greater<>` comparator would create a min-heap: if `a > b`, `a` must have a lower priority than `b` because `std::greater<T>(a, b)` returns `true`.

```
1  std::vector<int32_t> v1 = {4, 6, 2, 9, 7, 3, 8, 1, 5};
2  std::make_heap(v1.begin(), v1.end());                    // creates max-heap
3  for (int32_t val : v1) {
4    std::cout << val << ' ';                               // 9 7 8 6 4 3 2 1 5
5  } // for val
```

```
std::vector<int32_t> v2 = {4, 6, 2, 9, 7, 3, 8, 1, 5};
std::make_heap(v2.begin(), v2.end(), std::greater<int>());    // creates min-heap
for (int32_t val : v2)
  std::cout << val << ' ';                                    // 1 4 2 5 7 3 8 9 6
} // for val
```

The STL algorithm library also provides the `std::push_heap()` and `std::pop_heap()` methods, which can be used to insert and remove values from a heap. The `std::push_heap()` method takes the value at the end of a given iterator range and fixes it into the correct position in the heap. The `std::pop_heap()` method takes the value at the front of an iterator range, swaps it with the value at the back, and then fixes the element that was swapped to the front into its correct heap-ordered position.

```
template <typename RandomAccessIterator, typename Compare>
void std::push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
Inserts the value at the position `last - 1` into the heap defined by the range `[first, last - 1)`. The comparator is optional, and the method defaults to a max-heap if it is not provided.

```
std::vector<int32_t> v3 = {4, 6, 2, 9, 7, 3, 8, 1};
std::make_heap(v3.begin(), v3.end());                         // 9 7 8 6 4 3 2 1 5
v3.push_back(10);                                             // 9 7 8 6 4 3 2 1 5 10
std::push_heap(v3.begin(), v3.end());                         // 10 9 8 6 7 3 2 1 5 4
```

```
template <typename RandomAccessIterator, typename Compare>
void std::pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
Swaps the value at the position `first` with the value at `last -` and turns the subrange `[first, last - 1)` into a heap. The comparator is optional, and the method defaults to a max-heap if it is not provided.

```
std::vector<int32_t> v4 = {4, 6, 2, 9, 7, 3, 8, 1};
std::make_heap(v4.begin(), v4.end());                         // 9 7 8 6 4 3 2 1 5
std::pop_heap(v4.begin(), v4.end());                          // 8 7 3 6 4 1 2 9
v4.pop_back();  // actually removes 9 from the vector         // 8 7 3 6 4 1 2
```

Since `std::make_heap()` heapifies a given range of elements, its time complexity is $\Theta(n)$, where n is the number of elements in the range. However, the other two methods assume that the given iterator range is already in the form of a valid heap, so they only need to fix one value to its correct position (the value at the back needs to be fixed up for `std::push_heap()`, and the value that was newly swapped to the front needs to be fixed down for `std::pop_heap()`). Thus, the time complexity of these methods is the same as the time complexity of performing a single fix up or fix down, or $\Theta(\log(n))$.

※ 10.4.3 *Summary of Binary Heaps*

In summary, a heap is a structure than can be used to implement a priority queue. Because heaps support both $\Theta(\log(n))$ push and pop operations, they are more efficient than the standard sequence container implementations. There are many different types of heaps. The most common heap is the binary heap, where each node only supports at most two children. Although a binary heap can be visualized using a tree, it is common to store a heap's underlying data in a vector. For a tree to be considered a heap, it must be both complete and heap-ordered.

The binary heap relies on two important operations: fix up and fix down. If fix up is called on an element, the element is continuously swapped with its parent until it is in the correct position. If fix down is called on an element, the element is continuously swapped with its largest-priority child until it is in the correct position. Both operations take $\Theta(\log(n))$ time.

To push an element into a priority queue that is implemented using a binary heap, the element should be added to the back of the heap's underlying vector, and then fix up should be called on that element.

To pop an element from a priority queue that is implemented using a binary heap, the element at the front of the heap's underlying vector should be overwritten with the element at the back of the vector. The last element should then be removed from the vector, and fix down should be called on the element that was moved to the front.

To build a binary heap from a container of data, a bottom-up heapify should be used. To do this, start at the last internal node of the container and iterate to the front, calling fix down on each element you encounter. Because fix down is cheaper for leaf nodes, the bottom-up heapify approach is more efficient than the top-down approach, as there are more nodes closer to the leaf nodes than there are closer to the root.

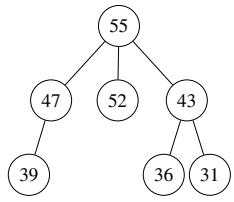
10.5 Pairing Heaps

※ 10.5.1 *Pairing Heap Structure*

A binary heap is a heap where each node has at most two children. However, binary heaps are not the only type of heaps that can be used to implement a priority queue. In this section, we will discuss a type of heap known as a **pairing heap**. Pairing heaps are similar to binary heaps in that the priority of any node cannot be higher than that of its parent. However, unlike a binary heap, each node in a pairing heap can have more than two children, and the structure of the tree does not need to be complete.

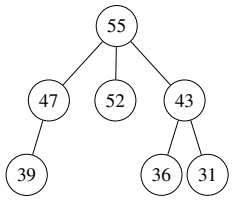
Because a node in a pairing heap can have many children, the array-based implementation is no longer feasible (there is no easy way to traverse the branches of the tree using indexing). As a result, pairing heaps are implemented using a pointer-based approach. However, instead of having each parent `Node` store pointers to each of their children (as was the case in a binary heap), each parent stores a *single* pointer to one of its children, and this child is linked to all of the other children using a `sibling` pointer. This is done for efficiency purposes: if a node has 281 children, for example, it would be impractical for that node to keep track of 281 pointers.

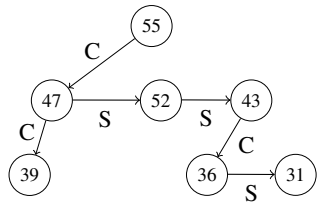
Pairing heaps come in two varieties: max pairing heaps and min pairing heaps. Much like binary heaps, the max pairing heap provides efficient access to the element with the largest value, and the min pairing heap provides efficient access to the element with the smallest value. How would you represent a pairing heap? Consider the following max pairing heap:



Behind the scenes, each `Node` keeps track of a `child` and `sibling` pointer, in addition to the data it stores. A node's `child` pointer points to the leftmost child of the node, and the `sibling` pointer links together all nodes that share a common parent.

Here, the root node 55 has three children. However, instead of keeping track of three pointers, 55 only keeps track of 47 as its `child`. The remaining children of 55 are linked together using `sibling` pointers: 47's `sibling` is 52, and 52's `sibling` is 43. The rest of the connections in the tree are labeled below (C for child, S for sibling):



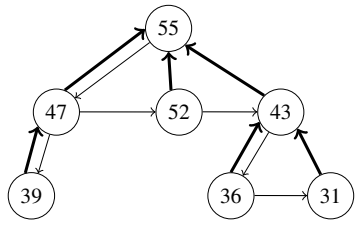


For our pairing heap to be fully efficient, each `Node` will need to store an additional pointer. This pointer can either be a `parent` or a `previous` pointer. Only one of these pointers is needed to complete all of the functionality of a pairing heap.

If each `Node` in the pairing heap stores a `parent` pointer, this pointer must point to the node's parent. An example is shown below (the bolded arrows represent the `parent` pointers):

```
1   template <typename T>
2   class Node {
3     T data;
4     Node* child;
5     Node* sibling;
6     Node* parent;
7   public:
8     explicit Node(const T& val)
9       : data{val}, child{nullptr},
10        sibling{nullptr},
11        parent{nullptr} {}
12  };
```

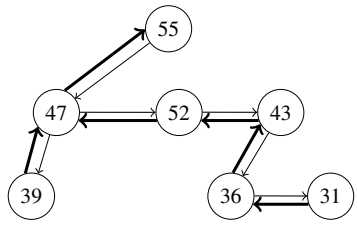
Pairing Heap Using Parent Pointers



If each `Node` in the pairing heap stores a `previous` pointer, this pointer must point to the sibling directly to the left of it. If the node is already the leftmost node among its siblings, its `previous` pointer should point to its parent. An example is shown below (the bolded arrows represent the `previous` pointers):

```
1   template <typename T>
2   class Node {
3     T data;
4     Node* child;
5     Node* sibling;
6     Node* previous;
7   public:
8     explicit Node(const T& val)
9       : data{val}, child{nullptr},
10        sibling{nullptr},
11        previous{nullptr} {}
12  };
```

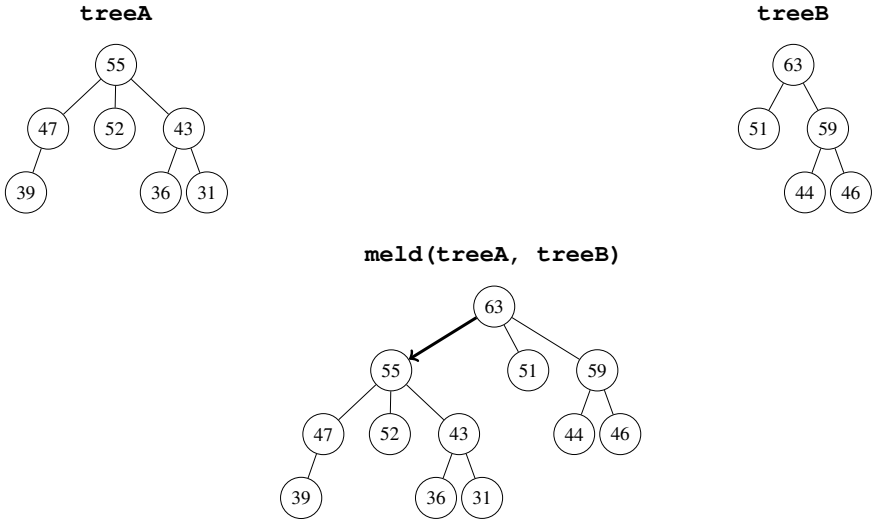
Pairing Heap Using Previous Pointers



The `parent` or `previous` pointer will become useful when we modify elements in the pairing heap (which will be discussed later in this section). Just remember that only *one* of these pointers is enough to build a fully functional priority queue with optimal time complexities. Even though having both may speed up things a little bit, the additional memory overhead involved with storing an extra pointer for every node in the pairing heap is not a worthwhile tradeoff for the small increase in performance.

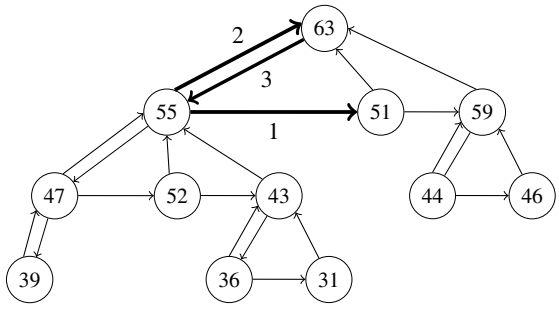
※ 10.5.2 *Meld*

The fundamental operation of a pairing heap is a method known as `meld()`, which can be used to combine two pairing heaps into a single pairing heap. When `meld()` is called on two pairing heaps, the function checks the roots of the two trees. The tree with the lower priority root is then attached as the leftmost child of the other pairing heap (ties are broken arbitrarily). In the following example, the root of tree A is less than the root of tree B, so melding tree A with tree B would cause tree A to be attached the leftmost child of tree B:



The implementation of `meld()` differs based on whether each node stores a `parent` pointer or a `previous` pointer. The following illustrates the `parent` pointer view:

Melding Pairing Heaps Using Parent Pointers

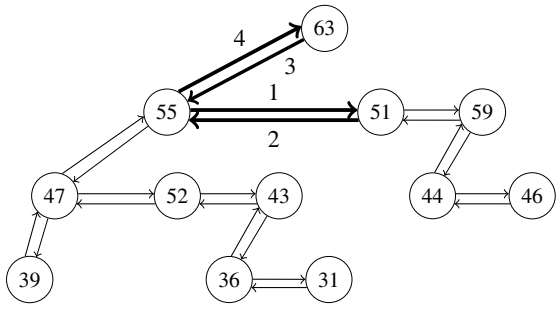


Here, there are three pointers that may need adjustments when melding. These are bolded in the previous diagram:

1. The `sibling` of the lower priority root should be set to the leftmost child of the higher priority root.
2. The `parent` of the lower priority root should be set to the higher priority root.
3. The `child` of the higher priority root should be set to the lower priority root.

Alternatively, the following illustrates the `previous` pointer view:

Melding Pairing Heaps Using Previous Pointers



Here, there are four pointers that may need adjustments when melding. These are bolded in the previous diagram:

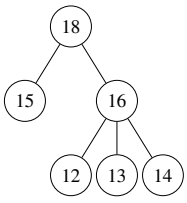
1. The `sibling` of the lower priority root should be set to the leftmost child of the higher priority root.
2. The `previous` of the higher priority root's leftmost child should be set to the lower priority root.
3. The `child` of the higher priority root should be set to the lower priority root.
4. The `previous` of the lower priority root should be set to the higher priority root.

Because `meld()` only moves a constant number of pointers around, its runtime is not affected by the size of the heap. Thus, each call to `meld()` takes $\Theta(1)$ time.

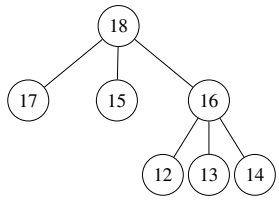
※ 10.5.3 *Inserting and Removing Elements*

To insert an element into the pairing heap, you can meld the newly created element with the rest of the pairing heap. If the new element has a lower priority than the root of the current pairing heap, add the new element as a child of the pairing heap. Otherwise, if the new element has a higher priority than the current root, add the entire current pairing heap as a child of the new element. This process takes $\Theta(1)$ time.

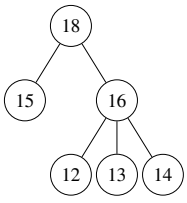
`.push(17)`



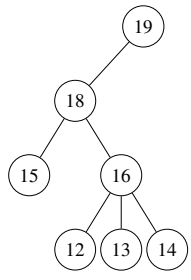
17 < 18, so add 17 as leftmost child



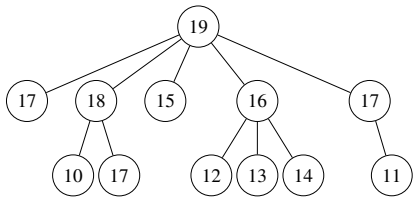
`.push(19)`



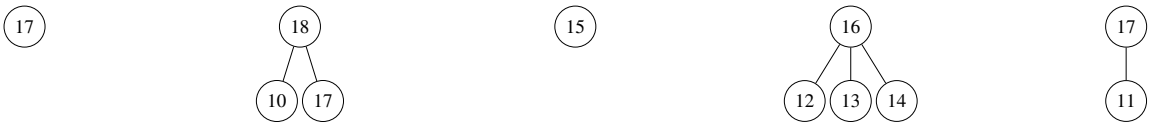
19 > 18, so add entire heap as leftmost child of 19



However, removing an element is slightly more complicated. Once the root is removed, there may be multiple children available; which one becomes the new root? Furthermore, how should these children be melded together? Consider the following pairing heap:



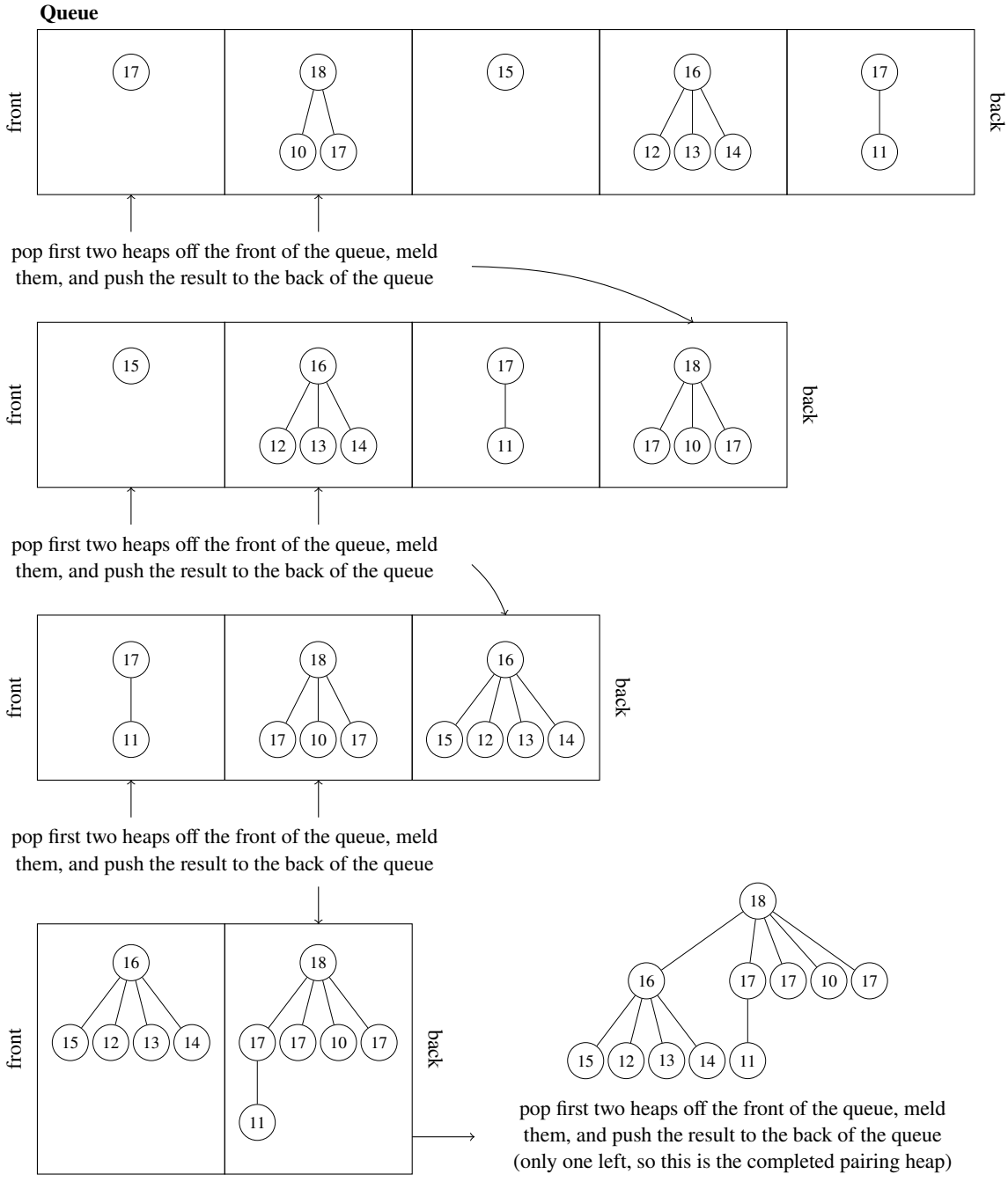
If we were to pop 19 off the pairing heap, we would end up with five separate children:



A key detail to note is that all of the children are also pairing heaps themselves! As a result, we can rebuild our pairing heap by simply melding the children back together. There are two approaches that can be efficiently used to do this: the **multi-pass** approach and the **two-pass** approach. In the multi-pass approach, we complete the following:

1. Take all the children and break all of their existing sibling connections.
2. Push all of the children into a queue (this is done concurrently with step 1 while breaking sibling connections).
3. Take two heaps from the front of the queue, meld them, and push the result to the back of the queue.
4. Repeat step 3 until only one pairing heap remains.

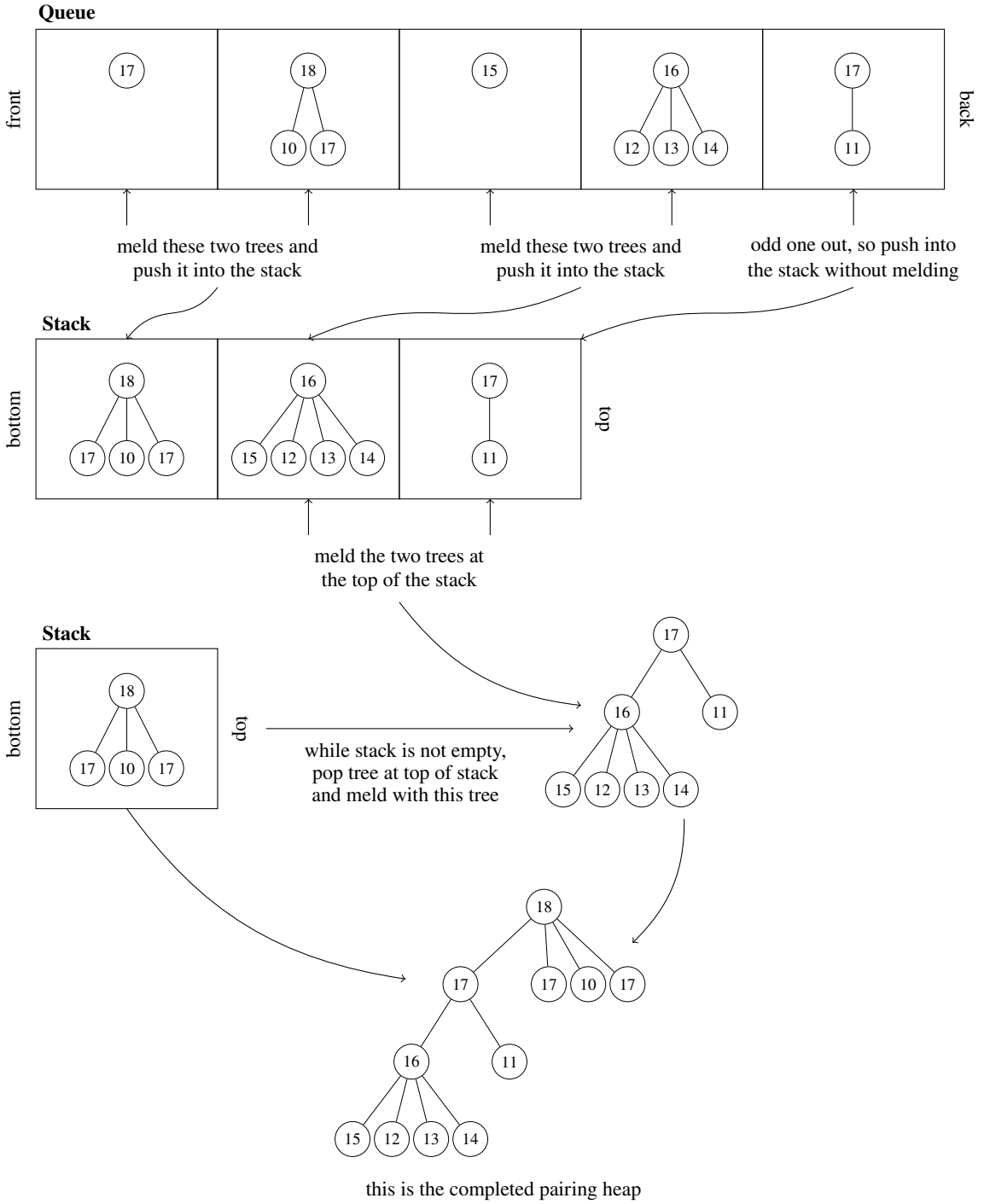
Using the previous example, we would place the five children into a queue. Then, we would continuously remove two children from the front, meld them together, and push them back into the queue. This is repeated until only one heap is left. The time complexity of this pop operation is amortized $O(\log(n))$.



Alternatively, in the two-pass approach, we complete the following:

1. Take all the children and break all of their existing sibling connections.
2. Initialize both a queue and a stack (or alternatively, two deques). Push all of the children into the queue (this is done concurrently with step 1 while breaking sibling connections).
3. Make a left-to-right pass over the trees in the queue and meld them in pairs. Push the resultant trees into the stack. If there are an odd number of children, the last tree gets placed in the stack without being melded.
4. Start with the right-most tree and meld the remaining trees into this tree one at a time. This can be done by taking out the tree at the top of the stack and continuously melding it with the remaining trees in the stack.

Like with the multi-pass approach, the two-pass approach can also be done in amortized $O(\log(n))$ time. (Note that the stack in the following illustration is inverted and flipped on its side.)



Remark: At this point, you might ask: why do we have to perform multiple passes when reconstructing the pairing heap? Can't we just list out the children and iterate over them once, melding the children together into an accumulated pairing heap?

It turns out that such an approach is not as efficient for repeated operations. When you make multiple passes and continuously meld pairs of children with each pop, over time you will end up with more ideal trees with fewer intermediate children of the root. This allows the pop operation to support an amortized $O(\log(n))$ time complexity, as mentioned (we will go over amortization in chapter 12). On the contrary, if you simply make a single pass over the children and meld them one-by-one into an accumulated heap, you may eventually end up with a pairing heap with many immediate children after subsequent pops, which would make the pop operation more expensive (since there are more immediate children to meld together).

※ 10.5.4 *Implementing the Pairing Heap Copy Constructor*

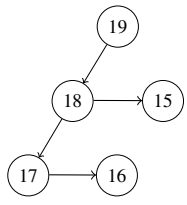
Because the pairing heap uses dynamic memory, we will need to implement a custom copy constructor, destructor, and assignment operator. The copy-swap method can be used to implement the assignment operator (make a temporary copy of the other pairing heap and swap the contents of the current pairing heap with the temporary one).

When implementing the copy constructor, we cannot just copy over the root node's pointer — we must walk through the entire tree and copy over every node. Note that the structure of the heap does not need to be the same between the original and the copy. As long as the copy is still a valid pairing heap that holds the same elements, the copy is valid.

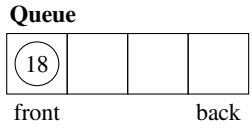
To walk through all the elements of a pairing heap, we will utilize a queue to store the elements we have yet to visit. Whenever we visit a node in the pairing heap, we push a copy to our new pairing heap and push all of the node's direct connections (`child` and `sibling`) into the queue — this ensures that all nodes in the heap will be pushed into the queue at some point during our algorithm. Then, we use the following process to make a copy of the pairing heap:

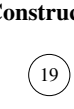
1. Push the root node of the pairing heap into a queue.
2. Retrieve the node at the front of the queue and pop it off.
3. Add all the direct connections of the node into the queue. If the node that was taken out has a child, push the child into the queue. If the node that was taken out has a sibling, push the sibling into the queue.
4. Push the value of the node that was taken out of the queue into the new pairing heap that is being copy constructed.
5. Repeat steps 2-5 until the queue is empty.

For instance, suppose we wanted to copy the following pairing heap:

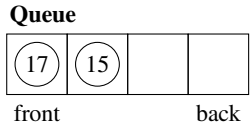


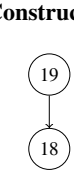
First, we will copy over the root node, 19. Before we push 19 into our new pairing heap, we must first push its `child` and `sibling` into our queue (if they aren't `nullptr`). In this case, 19 has a `child` of 18, so 18 gets pushed into the queue. After its connections are pushed into the queue, 19 is pushed into the new pairing heap.



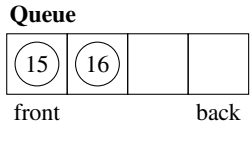


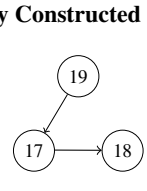
Then, we pop the element at the front of the queue (18). We push 18's `child` (17) and `sibling` (15) into the queue and push 18 into our heap.



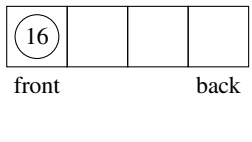


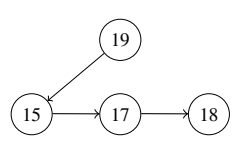
Continuing this process, we then take out 17, push its `sibling` (16) into the queue, and push 17 into our heap. 17 has no `child`, so only the `sibling` is pushed into the queue.





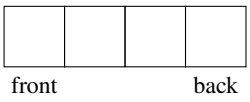
We then take out 15. Since 15 has no children or siblings, nothing is added to the queue. 15 is pushed into the heap.



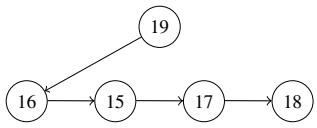


We then take out 16. Since 16 has no children or siblings, nothing is added to the queue. 16 is pushed into the heap.

Queue



Copy Constructed Heap



The queue is empty, so all of the elements have been successfully copied over. Even though the structure of the new, copy constructed pairing heap is different from the original heap, this is okay because the new heap is still a valid pairing heap with the same elements as the copy. The internal structure of the new pairing heap does not matter as long as we can still efficiently retrieve the element with the highest priority.

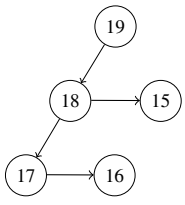
Because the copy constructor iterates over all the nodes in a pairing heap and performs a copy, the time complexity of the copy constructor is $\Theta(n)$, where n is the number of nodes in the pairing heap that is being copied.

※ 10.5.5 *Implementing the Pairing Heap Destructor*

The destructor works similarly to the copy constructor. The only difference is that, instead of pushing the nodes we visit into a new pairing heap, each node is deleted after we take it out of the queue. This ensures that all of the nodes are cleaned up when the heap is destroyed. To destruct a pairing heap, complete the following process:

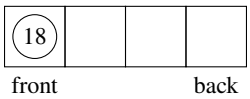
1. Push the root node of the pairing heap into a queue.
2. Retrieve the element at the front of the queue and pop it off.
3. Add all the direct connections of the node into the queue. If the node that was taken out has a child, push the child into the queue. If the node that was taken out has a sibling, push the sibling into the queue.
4. Delete the node that was taken out.
5. Repeat steps 2-5 until the queue is empty.

For instance, suppose we wanted to destruct the following pairing heap:

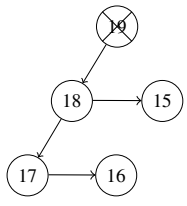


First, we add all of 19's direct connections into the queue (here, 18 is added since this is 19's child). The dynamic memory of 19 is then freed.

Queue

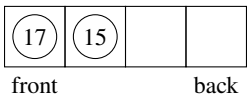


Heap to Destruct

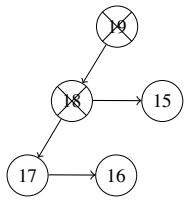


We then pop the element at the front of the queue, or 18. We add 18's direct connections (17 and 15) into the queue before freeing 18's memory.

Queue

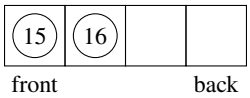


Heap to Destruct

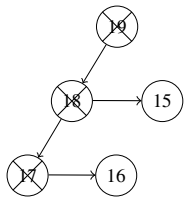


We then take out the element at the front of the queue, or 17. We add 17's direct connections (16) into the queue before freeing 18's memory.

Queue



Heap to Destruct



We then take out 15. Since 15 has no children or siblings, nothing is added to the queue before it is deleted.

We then take out 16. Since 16 has no children or siblings, nothing is added to the queue before it is deleted.

The queue is empty, which means all the nodes have been cleaned up. Similar to the copy destructor, the destructor iterates over all the nodes in a pairing heap to delete them; the time complexity of the destructor is thus also $\Theta(n)$, where n is the initial size of the pairing heap to be destroyed.

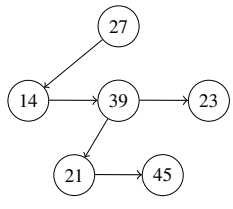
※ 10.5.6 *Updating Priorities*

So far, we have covered the basics of constructing and destructing a pairing heap. However, suppose someone went behind the scenes and messed up all the data in your pairing heap. Now your pairing heap is invalid! How would you go about fixing it?

One problem with this situation is that you do not know where the invalid elements could be. Maybe one element is in the wrong place, or maybe all the elements are in the wrong place. As a result, the best way to go about this is to break apart the entire heap and rebuild it. To do so, you will need to walk through the entire pairing heap. Like before, this can be done using a queue, which guarantees that each element in the pairing heap is visited once. First, push the root of the invalid pairing heap into a queue and reset the root of the current pairing heap to `nullptr`. Then, repeat the following process until the queue is empty:

1. Retrieve the node at the front of the queue and pop it off. This node is the root of a subheap of the initial pairing heap, so it may have additional connections (i.e., children and siblings) that will need to be broken (this is different from the copy constructor and destructor since we are rearranging the nodes of an existing pairing heap).
2. Add all the direct connections of this node (i.e., the root of the removed subheap) into the queue. If the node has a child, push the child into the queue. If the node has a sibling, push the sibling into the queue.
3. Break all of the subheap's existing connections by setting `child`, `sibling` and `parent`/`previous` to `nullptr`.
4. After breaking all of its connections, meld the removed subheap with the current pairing heap.

To illustrate this, suppose you wanted to fix the following invalid pairing heap:



First, push the root (27) into a queue and set the current pairing heap pointer to `nullptr`.

Then, we take out the node at the front (27) and push in its `child` and `sibling`, if there are any. In this case, 27 has a `child` of 14, so 14 gets pushed into the queue. Then, we break all of 27's connections and meld it with the new pairing heap (which is currently `nullptr`).

We then take 14 out of the queue, push its `sibling` (39) into the queue (there is no `child` to push in), break all of 14's connections, and meld it with the new pairing heap.

We then take 39 out of the queue, push its `child` (21) and `sibling` (23) into the queue, break all of 39's connections, and meld it with the new pairing heap.

We then take 21 out of the queue, push its `sibling` (45) into the queue, break all of 21's connections, and meld it with the new pairing heap.

We then take 23 out of the queue. Since 23 has no children or siblings, nothing is pushed into the queue. We break all of 23's connections and meld it with the new pairing heap.

We then take 45 out of the queue. Since 45 has no children or siblings, nothing is pushed into the queue. We break all of 45's connections and meld it with the new pairing heap.

Since the queue is now empty, we have successfully fixed the pairing heap. The new pairing heap points to the same data as before, but the relationships between elements are now valid. Because we are iterating over all the nodes of the pairing heap to rebuild it, the time complexity of this procedure is $\Theta(n)$, where n is the number of nodes in the pairing heap.

※ 10.5.7 *Increasing the Priority of an Existing Element*

Now, suppose someone *increased* the priority of a single element in the heap, and you are given the node that is modified. In this case, breaking and rebuilding the entire heap is no longer an efficient way to fix the heap. This is because the rest of the heap is still valid, and tearing it apart and rebuilding it would be a waste of time. Instead, if the priority of a single element is increased, all you need to do is compare the updated node with the value of its parent (if there is one). This is where the `parent` or `previous` pointer comes in.

If the modified value still has a lower priority than its parent after the update, then nothing needs to be done since the heap would still be valid. However, if the modified value has a higher priority than its parent, it must be broken off from its siblings and remelded with the root. Since we are only considering an increase in priority for this problem, the children of the modified node do not need to be detached, as they are guaranteed to remain in a valid position. Had the element's priority been decreased instead (which we will not deal with in this class), its children would also need to be detached and remelded back into the tree. Consider the following example, where 38 is updated to 83:

Because the updated value of 83 is larger than the value of its parent, 79, the entire subtree rooted at 83 must be broken off and remelded with the original root:

The implementation of this operation depends on whether a `parent` or `previous` pointer is used. If a `parent` pointer is used, comparing the modified element to its parent is fast, since the `parent` pointer provides direct access to the parent. However, in order to break the sibling connections of the nodes adjacent to the modified node, you must traverse rightward through the siblings on its level to find the element directly to the left of the modified node. On the other hand, if a `previous` pointer is used, comparing the modified node with its parent takes longer, since you would need to traverse leftward through the siblings on the same level to find the parent. However, breaking the sibling connections is faster with a `previous` pointer, since you have direct access to the sibling directly to the left of the modified node.

Remark: If you have a pairing heap that is implemented with a `previous` pointer, it may be worthwhile to skip the parent check entirely and *always* break and meld with every update. This is because the updated node's parent is not easily accessible if you only have a `previous` pointer, and the cost of breaking and melding even when it is not necessary may still be preferable to the cost of iterating over all the siblings to find the parent. That being said, if you do have a `parent` pointer, it is still worthwhile to check if corrective action is needed, since you have direct access to the parent, and it can save you the cost of doing unnecessary work.

10.6 The STL Priority Queue Container

Like many of the other containers we have discussed, the STL provides a priority queue data structure that is implemented for you. Behind the scenes, the STL's priority queue is typically implemented using a binary heap.[3] To use an STL priority queue, include the `<queue>` library and declare a `std::priority_queue<>` using the following syntax:

```
std::priority_queue<TYPE, CONTAINER, COMPARATOR> name_of_pq;
```

Similar to the `std::stack<>` and `std::queue<>`, the `std::priority_queue<>` is a container adaptor that is built on top of another container that stores its data. This container can be specified as the second term in the template definition (`CONTAINER`). There are limitations to what this underlying container can be — a priority queue's underlying container must be sequential and support random access, since binary heaps rely on efficient indexing to access a node's children.

The third term of the template definition is a comparator that can be used to determine the priority of elements in the priority queue. Given a comparator, the rules for determining priority are the same as it was in the case of `std::make_heap()` — if `comp(a, b)` returns `true` for any two elements `a` and `b` in the priority queue, `a` has a lower priority than `b`.

[3] Why not a pairing heap or any other type of heap? Even though pairing heaps are theoretically more efficient, binary heaps often outperform more complicated heaps due to caching, since memory closer together in memory can be sequentially accessed faster. Also, it should be noted that `std::priority_queue<>` is *not guaranteed* to be implemented using a binary heap, as long it conforms to the interface specified by the standard. As always, STL containers are implementation defined, and different libraries may use different implementations (although many use a binary heap of some form).

Only the first term (the `TYPE`) is required when declaring an STL priority queue. If the `CONTAINER` is not specified, the underlying container defaults to a `std::vector<>`. If the `COMPARATOR` is not specified, the priority queue defaults to a max priority queue using the `std::less<>` comparator. For example, the following initializes a max priority queue of integers that uses a vector for its underlying data:

```
std::priority_queue<int32_t> max_pq;
```

Unfortunately, the `COMPARATOR` must be the third term in the template definition; if you want to declare anything other than a max priority queue, you must also specify the underlying container, even if you want to use the default. For instance, the following line of code constructs a min priority queue using the `std::greater<>` comparator:

```
std::priority_queue<int32_t, std::vector<int32_t>, std::greater<int32_t>> min_pq;
```

Here, the `std::vector<int32_t>` term *must* be included if you want to specify a custom comparator. The following line of code would *not* compile, since the `COMPARATOR` cannot be the second term in the template definition.

```
std::priority_queue<int, std::greater<int>> min_pq;  // does not compile!
```

The `std::priority_queue<>` supports the following operations:

Function	Behavior
`.push(val)`	Adds `val` to the priority queue
`.emplace(args)`	Constructs a new element in-place (i.e., no copies or moves) using the provided constructor arguments
`.pop()`	Removes the element with the highest priority in the priority queue
`.top()`	Returns a **const** reference to the element with the highest priority in the priority queue
`.size()`	Returns the number of elements in the priority queue
`.empty()`	Checks if the priority queue is empty

The `.top()` function returns a `const` reference to the top element in the priority queue because you are *not* allowed to modify an element once it is in the priority queue. This is because the `std::priority_queue<>` does not automatically fix itself if an element's priority is invalidated. If you want to modify the top element of a priority queue, the safest way would be to pop the element out and repush it in with a new value. For larger structs or classes, you can use the `mutable` keyword to denote member variables that can be modified, but you must be careful not to invalidate the priority queue in the process.

The STL priority queue also supports a *range constructor* that can be used to construct a priority queue out of an existing container of data. As a result, if you have a container of data values that you want to insert into a priority queue, you do not need to push the elements in one by one. Instead, you can pass in an iterator range when constructing the priority queue, and the range constructor will construct the priority queue using the elements in the range. An example is shown below:

```
std::vector<int32_t> vec = {12, 66, 34, 25, 84, 25, 17, 98, 53};
std::priority_queue<int32_t> pq(vec.begin(), vec.end());
```

The above code constructs a priority queue by heapifying the contents of the vector in $\Theta(n)$ time. This is more efficient than pushing each of the numbers into the priority queue individually, which would take worst-case $\Theta(n\log(n))$ time (n pushes that each take $\Theta(\log(n))$ time).

Example 10.7 Consider the following code, which uses the STL priority queue and a custom comparator:

```
1  struct Comp {
2    bool operator() (const int32_t lhs, const int32_t rhs) const {
3      return abs(lhs - 100) > abs(rhs - 100);
4    } // operator()()
5  };
6
7  int main() {
8    std::vector<int32_t> vec = {112, 166, 314, 251, 184, 25, 117, 98, 153};
9    std::priority_queue<int32_t, std::vector<int32_t>, Comp> pq(vec.begin(), vec.end());
10   std::cout << pq.top() << '\n';
11 } // main()
```

What is the output of this code?

In this example, the priority queue is given a custom comparator. Which element has the highest priority according to this comparator? To start off, let's look at the first two elements in the vector, 112 and 166. If we passed 112 and 166 into the comparator in this order, the comparator would return `false` because `abs(112 - 100)` is not greater than `abs(166 - 100)`. Thus, 112 must *not* have a lower priority than 166 (since `false` = higher priority).

What about the elements 25 and 166 — which one has the higher priority? If we passed 25 and 166 into the comparator in this order, the comparator would return `true` because `abs(25 - 100)` is greater than `abs(166 - 100)`. Thus, 25 must have a lower priority than 166.

In fact, the priority of an element is determined by its positive difference from 100. Given two numbers `a` and `b`, `comp(a, b)` would only return `true` if `a` is farther from 100 than `b`. Thus, elements that are farther from 100 must have a lower priority than elements that are closer to 100. Since the code prints out the element with the highest priority, the output must be the value in the vector that is closest to 100, or 98.

Example 10.8 Consider the following object, which stores information on a stock order:

```
struct StockOrderInfo {
  int32_t price;
  int32_t quantity;
  int64_t timestamp;
  int64_t order_id;
};
```

The `price` member stores the price of an order, the `quantity` member stores the size of an order, the `timestamp` member stores the time at which an order was placed, and the `order_id` member stores the order ID. For the sake of simplicity, assume that all values for these four members will be positive integers (this removes the issue of floating-point comparison precision, etc.).

Given a set of resting BUY orders that need to be executed, a broker-dealer must satisfy orders in the following priority:

- Orders with the highest price must be executed first, regardless of quantity or time of order.
- If there are multiple orders with the same price, the order with the earlier timestamp must be executed first.
- If there are multiple orders with the same price and time of order, the order with the larger quantity must be executed first.
- If there are multiple orders with the same price, time of order, and quantity, the order with the smaller order ID must be executed first. Assume that all orders have a unique ID.

The orders are to be inserted into a priority queue, where orders with higher priority are executed first. Implement a comparator that can be used to define this priority queue such that it follows the provided rules above.

Given two stock orders `a` and `b`, our comparator should return `true` if `a` has a lower priority than `b`. What determines if a stock order has a lower priority?

- Given two orders with the same price, time of order, and quantity, the stock order with the larger order ID has the lower priority. Therefore, we return `true` for larger IDs in this case.
- Given two orders with the same price and time of order, the stock order with the smaller quantity has the lower priority. Therefore, we return `true` for smaller quantities in this case.
- Given two orders with the same price where no variables match, the stock order with the later timestamp has the lower priority. Therefore, we return `true` for larger timestamps in this case.
- Otherwise, if given two orders, the stock order with the lower price has the lower priority. Therefore, we return `true` for lower prices in this case.

The constructor below satifies the rules above (the steps for defining a comparator mirrors the process introduced in section 1.6, so we will not be fully walking through every step here):

```
struct StockOrderComparator {
  bool operator() (const StockOrderInfo& lhs, const StockOrderInfo& rhs) const {
    if (lhs.price == rhs.price) {
      if (lhs.timestamp == rhs.timestamp) {
        if (lhs.quantity == rhs.quantity) {
          return lhs.order_id > rhs.order_id;
        } // if
        return lhs.quantity < rhs.quantity;
      } // if
      return lhs.timestamp > rhs.timestamp;
    } // if
    return lhs.price < rhs.price;
  } // operator()()
};
```

We can then define our priority queue as follows:

```
std::priority_queue<StockOrderInfo, std::vector<StockOrderInfo>, StockOrderComparator> stock_pq;
```

Some example code is provided below:

```
int main() {
  std::priority_queue<StockOrderInfo, std::vector<StockOrderInfo>, StockOrderComparator> stock_pq;
  stock_pq.push(StockOrderInfo{.price = 50, .quantity = 30, .timestamp = 12343, .order_id = 1023});
  stock_pq.push(StockOrderInfo{.price = 50, .quantity = 35, .timestamp = 12343, .order_id = 1024});
  stock_pq.push(StockOrderInfo{.price = 50, .quantity = 45, .timestamp = 12344, .order_id = 1025});
  stock_pq.push(StockOrderInfo{.price = 50, .quantity = 45, .timestamp = 12344, .order_id = 1026});
  stock_pq.push(StockOrderInfo{.price = 60, .quantity = 25, .timestamp = 12345, .order_id = 1027});

  std::cout << "The orders are executed in the following order: ";
  while (!stock_pq.empty()) {
    std::cout << stock_pq.top().order_id << " ";
    stock_pq.pop();
  } // while
} // main()
```

The output of this code is:

```
The orders are executed in the following order: 1027 1024 1023 1025 1026
```

10.7 Solving Problems Using Priority Queues

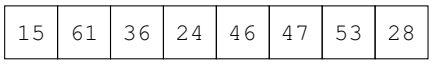
In this section, we will explore some problems that can be solved using priority queues.

※ 10.7.1 *k Largest Elements in an Array*

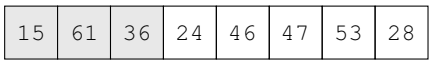
Example 10.9 Suppose you are given an array of n elements, and you want to write a program that prints the k largest elements in the array, in any order. How would you approach this problem?

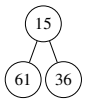
The simplest solution would be to sort the array and print out the k largest elements. However, this is not the most efficient approach, since sorting takes $\Theta(n\log(n))$ time (we will go over more detail on sorting algorithms in chapter 14). A better solution would be to heapify our data into a *max-heap* and extract out the k largest elements. Heapifying an array takes worst-case $\Theta(n)$ time, and popping out the largest element takes worst-case $\Theta(\log(n))$ time. Since this algorithm pops out the largest element k times after heapifying, the overall worst-case time complexity of this algorithm is $\Theta(n + k\log(n))$ — $\Theta(n)$ from the heapify step, and $\Theta(k\log(n))$ from popping k times.

However, there is a better way to solve this problem. This process is a less intuitive, since it uses a *min-heap* to find the k largest elements rather than a max-heap. In this procedure, the first k elements of the array are added to the min-heap. Then, the remaining $n-k$ elements are compared with the element at the top of the min-heap. If any of the elements encountered is larger than the top of the min-heap, pop off the top value and push in the new, larger value. This ensures that the min-heap will always contain the k highest priority elements that have been seen so far. The worst-case time complexity of this process is $\Theta(n\log(k))$; this is because n pushes and pops may be needed, which each take worst-case $\Theta(\log(k))$ time. The auxiliary space used is $\Theta(k)$, since the priority queue is capped at a size of k. This algorithm is illustrated below:

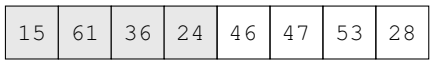


If we wanted to print the 3 largest elements in this array, we would first push the first 3 elements into a min-heap.

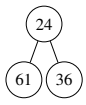


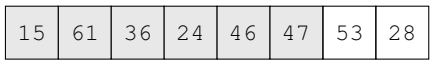


Then, we would iterate through the remaining elements and compare them with the top element. If an element has a larger priority than the top element of the min-heap, pop off the element at the top and push the current value in.

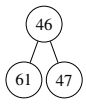


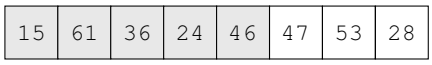
24 > 15, so pop 15 out and push 24 in



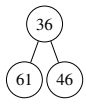


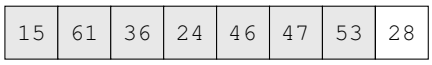
47 > 36, so pop 36 out and push 47 in



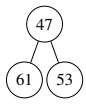


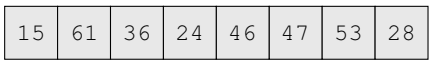
46 > 24, so pop 24 out and push 46 in



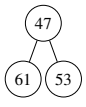


53 > 46, so pop 46 out and push 53 in





28 < 47, so it is not pushed into the min-heap



The three largest elements in the array are thus 47, 61, and 53, since these are the elements remaining in the min-heap. The worst-case time complexity of this algorithm can be further improved from $\Theta(n\log(k))$ to $\Theta(k + (n-k)\log(k))$ if the first k elements were heapified rather than being pushed in one by one.

※ 10.7.2 *Streaming Median Algorithm*

In this section, we will discuss how heaps can be used to efficiently determine the median of a *stream* of data. Unlike an array, the data in the stream must be accessed sequentially, and there is no way to predetermine the number of elements in a stream nor identify the values of elements that are yet to be extracted.

Example 10.10 Devise an algorithm that can be used to determine the median value of all values that you have seen at any point in a given stream. For example, consider the following stream of data:

```
12 55 74 35 96 52 54 53 63 23 37 ...
```

If you were asked to calculate the median after the first seven numbers in the stream are extracted (12, 55, 74, 35, 96, 52, and 54), you would return 54, since that is the median of the first seven numbers. If you were asked again to calculate the median after extracting 53, the new median would be 53.5. How can you accomplish this task efficiently?

A naïve approach would be to store all the values you have extracted so far in a vector and sort the vector whenever you need to calculate the median. For example, if you were asked to determine the median after extracting the first seven elements, you would sort the first seven elements and retrieve the middle element, as shown:

12	35	52	54	55	74	96

However, sorting is an $\Theta(n \log(n))$ process, which can quickly blow up the runtime if you are asked to query the median multiple times. Instead, a better method would be to use a heap to keep track of the streaming median.

If you think back to the previous problem of finding the k largest elements in an array, we used a min-heap of size k to keep track of the largest k elements seen so far. A similar approach can be used to help us find the streaming median: we will instantiate a max-heap *and* a min-heap, where the former keeps track of the smallest $n/2$ elements, and the latter keeps track of the largest $n/2$ elements. By doing so, we ensure that the values needed to calculate the median are always accessible at the top of the two heaps. This process can be summarized as follows:

1. Every time you extract an element, determine whether it should be pushed into the max-heap (which stores the smaller half of elements seen so far) or the min-heap (which stores the larger half of elements seen so far).
   - This is done by checking the values at the top of the heaps: if the new element is larger than the top of the max-heap, push it into the min-heap; otherwise, push it into the max-heap. If both heaps are empty, you can push the new element into either heap.
2. If the sizes of the two heaps differ by more than one, you will have to rebalance the heaps to ensure that the median value is still on top.
   - This is done by popping the top element off the larger heap and pushing it into the smaller one.
3. To calculate the median at any point in time, first check the sizes of the two heaps. If they have the same size, there must be an even number of elements, so the median would be the average of the two values at the top of the heaps. Otherwise, there is an odd number of elements, so the median would be the value at the top of the heap with the larger size.

Consider the same stream as above, but this time using the heap approach:

```
12 55 74 35 96 52 54 53 63 23 37 ...
```

First, we declare two heaps: a max-heap that stores the smaller half of values seen, and a min-heap that stores the larger half of values seen. The first value we extract, 12, can go in any heap, so we will arbitrarily add it to the left one.

Left Max-Heap **Right Min-Heap**

The next element in the stream, 55, is larger than the top of the left heap, so it gets added to the right heap.

Left Max-Heap **Right Min-Heap**

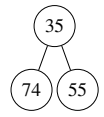
At this point, the median is the average of the two top values, or (12 + 55) / 2 = 33.5, since both the left and right heaps are the same size. The next element in the stream, 74, is larger than the value at the top of the left heap, so it gets added to the right heap.

Left Max-Heap **Right Min-Heap**

At this point, the median is the value at the top of the right heap, or 55, since the right heap has a larger size. The next element in the stream, 35, is larger than the value at the top of the left heap, so it gets added to the right heap.

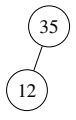
Left Max-Heap

Right Min-Heap

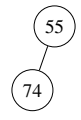


We now have an issue — the two heaps differ in size by more than one. When this happens, we rebalance the heaps by moving the top element from the larger heap (35) into the smaller heap.

Left Max-Heap

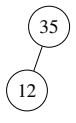


Right Min-Heap

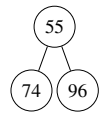


At this point, the median is the average of the two top values, or (35 + 55) / 2 = 45, since both the left and right heaps are the same size. The next element in the stream, 96, is larger than the value at the top of the left heap, so it gets added to the right heap.

Left Max-Heap

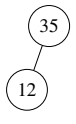


Right Min-Heap

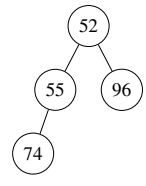


At this point, the median is the value at the top of the right heap, or 55, since the right heap has a larger size. The next element in the stream, 52, is larger than the value at the top of the left heap, so it gets added to the right heap.

Left Max-Heap

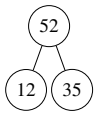


Right Min-Heap

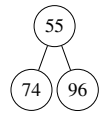


Once again, the two heaps differ in size by more than one, so we rebalance the heaps by moving the top element of from the larger heap (52) into the smaller heap.

Left Max-Heap



Right Min-Heap



At this point, the median is the value at the average of the two top values, or (52 + 55) / 2 = 53.5, since both the left and right heaps are the same size. We can continue this process for the remaining elements, adding each value to the correct heap, rebalancing if necessary, and then retrieving the median by either taking the top value of the larger heap, or an average of the top values if both heaps have the same size.

※ 10.7.3 *Merging Sorted Arrays*

Example 10.11 Suppose you are given k sorted arrays of size n each. Devise an time-efficient algorithm that merges the arrays into a single larger, sorted array.

The naïve approach would be to insert all the elements into an array of size nk, and then run a sorting algorithm on the entire array. However, this approach is inefficient, since sorting an array of size nk takes $\Theta(nk \log(nk))$ time. Instead, we will look at two better solutions that can be used to solve this problem, one that uses heaps and one that uses recursion.

In the heap solution, we first create a min-heap and insert the first element of each of the k sorted arrays. Then, while the min-heap has a size greater than zero, remove the element at the top of the heap, push it to the output array, and insert the next element in the array that the newly pushed in element originated from. To illustrate this process, suppose you are given the following four sorted arrays:

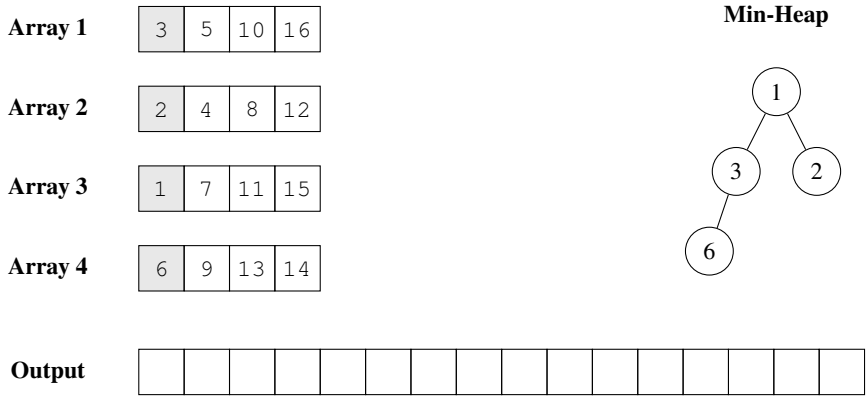
3	5	10	16

1	7	11	15

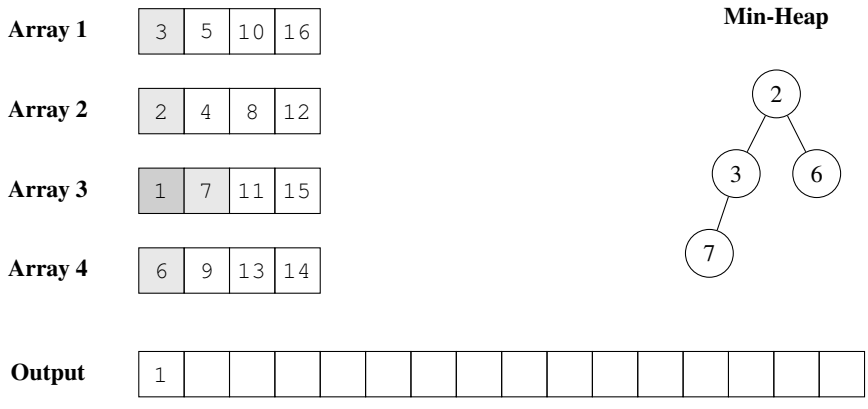
2	4	8	12

6	9	13	14

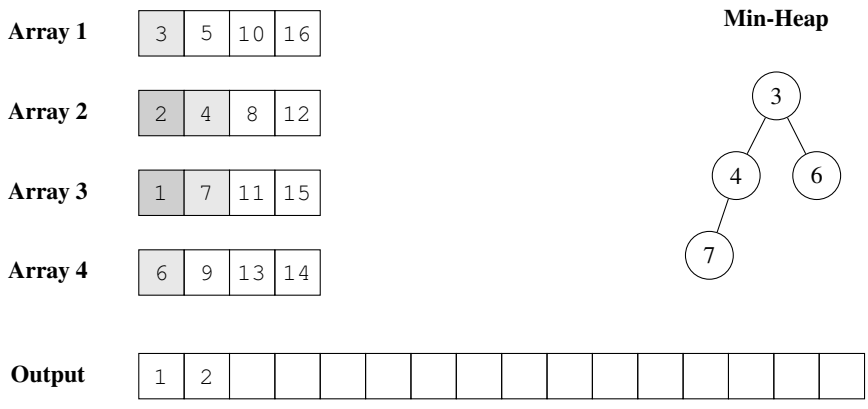
First, we will declare a min-heap and push in the first value of each of the k sorted arrays:



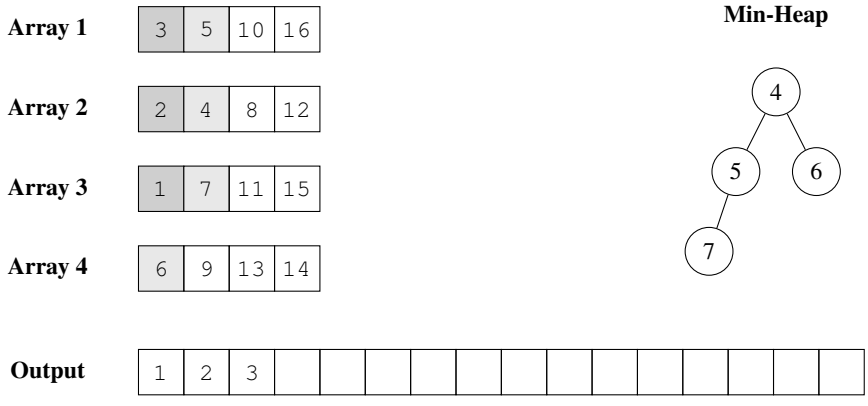
We then pop the top element off the heap and push it into our output array. Since the element we pushed to the output array belongs to array 3, we push the next element in array 3 into the min-heap.



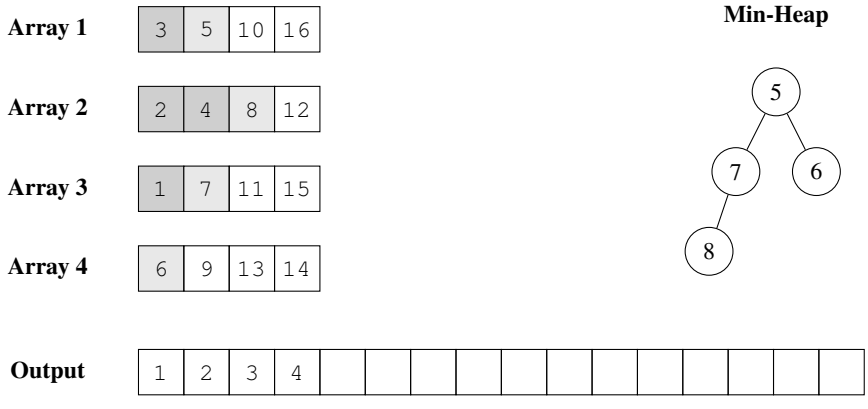
The top element is now 2, which belongs to array 2. Thus, we push 2 into the output vector and push the next value in array 2 into the min-heap.



The top element is now 3, which belongs to array 1. Thus, we push 3 into the output vector and push the next value in array 1 into the min-heap.

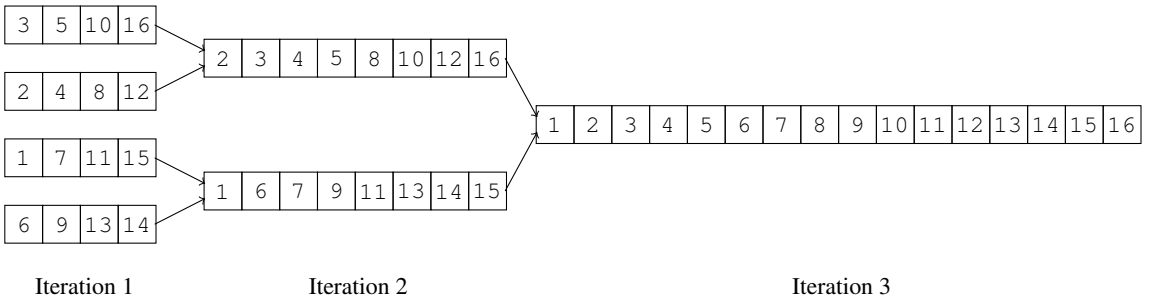


The top element is now 4, which belongs to array 2. Thus, we push 4 into the output vector and push the next value in array 2 into the min-heap.



If we continue pushing and popping, we will eventually push all of the elements into the output array in sorted order. Since the worst-case complexity of a single push or pop from the heap is $\Theta(\log(k))$ (since k is the size of the heap), and a total of nk elements are pushed and popped from the heap, the worst-case complexity of the entire algorithm is $\Theta(nk\log(k))$.

An alternative solution would be to recursively merge the arrays two at a time until there is only one array remaining. Merging two sorted arrays is a $\Theta(n)$ process (we will cover the details of the merging process in chapter 13).



What is the time complexity of this recursive approach? On the first iteration, we merge k arrays of size n together, which takes $\Theta(nk)$ time. On the second iteration, we merge $k/2$ arrays of size $2n$ together, which also takes $(k/2)\times 2n = \Theta(nk)$ time. In fact, each iteration of merging takes $\Theta(nk)$ time, and since we are halving the number of arrays we have to merge at each iteration (from k down to 1), a total of $\Theta(\log(k))$ iterations are needed. The total time complexity is thus $\Theta(\log(k))$ iterations $\times$ $\Theta(nk)$ work per iteration, or $\Theta(nk\log(k))$, the same as the heap approach! From an asymptotic standpoint, both algorithms perform equally well.