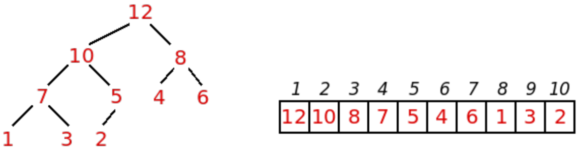
# Priority Queues and Heaps

L.EIC

Algorithms and Data Structures

2025/2026




**P Ribeiro**, AP Tomas

# Priority Queues - Motivation

- Hospital emergency departments often operate with the **Manchester Triage System**, which allows classifying the severity of each patient's situation, assigning them one of the following colors:

| Category | Classification | Time to be seen |
|---|---|---|
| 1 | IMMEDIATE | STRAIGHT AWAY |
| 2 | VERY URGENT | WITHIN 10 MIN |
| 3 | URGENT | WITHIN 60 MIN |
| 4 | STANDARD | WITHIN 120 MIN |
| 5 | NON-URGENT | WITHIN 240 MIN |

- The order in which patients are attended depends on their **priority**. For example, a red patient who arrives later is always attended to before any green or blue patient, even if they have been waiting in the emergency room for a long time.

# Priority Queues - Definition

- The ADTs (*abstract data structures*) we know depend only on the **order of arrival** and do not (directly) adjust to a process like this:
  - A **stack** is always LIFO (Last In, First Out)
  - A **queue** is always FIFO (First In, First Out)
  - A **deque** only allows access to the first or last element

- We need an ADT that takes into account **priorities**.
  - If there were always only the 5 priorities of the triage system, we could use queues. But what happens if there are more?
  - And in a more general case where the number of different priorities is not limited? (e.g., the priority is any number, a *double*)

- A **priority queue** (**Priority Queue**) is an ADT for storing a collection of elements that supports three main operations:
  - **insert(**$x$**)** which adds an element $x$ to the collection
  - **peek()** which returns (without removing) the **higher priority** element
  - **remove()** which returns and removes the **higher priority** element

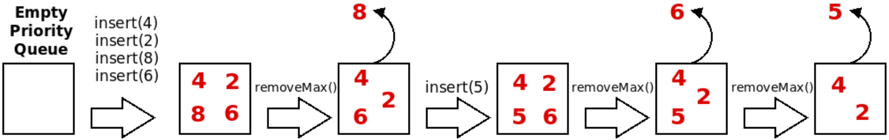
# Priority Queues - Applications

- Priority queues are useful in many other scenarios. Here are some **examples** where they can be applied:
  - A queue (e.g., at the post office) with **priority service** (e.g., for pregnant women)
  - A router with **priority traffic** (e.g., VoIP calls)
  - Simulation of **discrete events**: imagine several events starting at different times. We can use priority queues to determine the next event to occur (start time as priority)

  - There are many **algorithms** that use priority queues as a basic building block. Some examples:
    - **Dijkstra's Algorithm** (shortest paths): to determine the next closest unprocessed node
    - **Prim's Algorithm** (for minimum spanning trees): to determine the next closest node to the tree that has not yet been added
    - **A* Algorithm** (*best-first* search): to determine the next node to visit with the best heuristic value

# Priority Queues - What is higher priority?

- To think about implementation, we need to **define** what it means to have **"higher priority"**
- Without loss of generality, we will assume we are working with **comparable** elements and that the higher priority one is the **largest**.
  - For example, if we have the integers {4,8,5}, the largest is 8.
  - In a case like triage, it would be enough to associate larger numbers to more prioritized colors (e.g., red=5, orange=4, yellow=3, ..)
  - If the smallest rather than the largest was useful, simply adjust the priorities correspondingly (e.g., store the negatives of the numbers so the "largest" is originally the smallest)
- Thus, the operation **remove()** can also be thought of as **removeMax()** (and we will call the operation **peek()** **max()**)



# Priority Queues - Implementations

- How can we implement a priority queue using the data structures we already know?
  - **Unordered List:** an array or linked list without any order. Insertion is easy, but returning the maximum requires a linear search
  - **Ordered List:** an array or linked list in ascending order. Returning the maximum is easy (at the end), but insertion requires maintaining order
  - **Binary Search Tree:** insertion and removal (rightmost node) are associated with the tree height

| | **insert** | **max** | **removeMax** |
|---|---|---|---|
| **Unordered List** | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| **Ordered List** | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| **Binary Search Tree** *(if balanced)* | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |

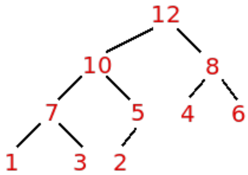
*Note: if the search tree is unbalanced, insertion and removal can cost $\mathcal{O}(n)$. We could also create a dedicated variable to store the maximum and respond to max() in $\mathcal{O}(1)$.*

# Heaps - Invariant

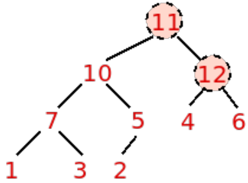
- Let us look at another **specialized** and very **efficient** solution.
- A **heap** is a tree that obeys the following restriction:

**(max)Heap Invariant**

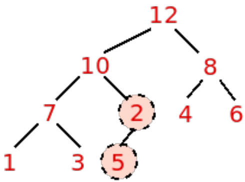
The parent of any node always has higher priority than the child. In a **maxHeap**, the parent is always *larger* than its children.




✓ It is a Heap




× It is not a Heap




× It is not a Heap

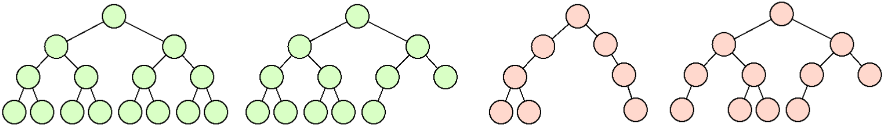
Note: in a **minHeap**, a node would always be smaller than its children.

# Heaps - Height $\mathcal{O}(\log n)$

- To ensure the efficiency of associated operations, a heap must also be a **complete binary tree**:

**Complete Tree**

A tree where all levels (except potentially the last) are fully filled with nodes, and all nodes are as far left as possible.



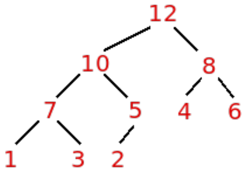
2 examples of complete trees

2 examples of incomplete trees

- In a complete tree with $n$ nodes, the height is $\mathcal{O}(\log n)$.
  - It is a highly balanced tree, and we've discussed this before. Intuitively, think that to *increase the height by 1*, it is necessary to *double* the number of elements.

# Heaps - Mapping to an Array

- The easiest and most compact way to implement a heap is to use an **array** that *implicitly* represents the tree.
  - The elements appear in the array in *level order* (from top to bottom, left to right).
  - If we place the root at position 1, then:
    - The children of the node at pos. $i$ are at pos. $i \times 2$ and $i \times 2 + 1$.
    - The parent of a node $i$ is at position $/2$ (integer division).
- Let's look at an example:



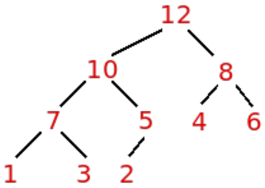

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 10 | 8 | 7 | 5 | 4 | 6 | 1 | 3 | 2 |

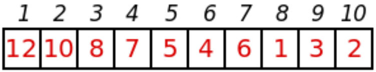
E.g.: children of pos. 3 (node 8) are at pos. $3 \times 2 = 6$ (node 4) and $3 \times 2 + 1 = 7$ (node 6). The parent of pos. 3 is the node at pos. $3/2 = 1$ (node 12).

- Since the tree is complete, this means the array has consecutively filled positions.

# Heaps - Operation max()

- Since each node is larger than its children, the largest node of all is guaranteed to be at the root of the heap (the first element of the array):



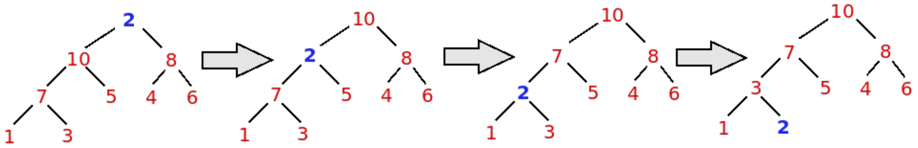




- **max():** time $\mathcal{O}(1)$

# Heaps - Operation removeMax()

- After removing the root, it is necessary to restore the heap properties. To do this:
  - Take the last element of the array and place it in the root's position (the tree remains complete).
  - The element "sinks" (**downHeap**), swapping with the largest of its children, until the heap invariant is restored.
  - At most, this traverses the entire height of the tree, which is $\mathcal{O}(\log n)$.

An example for the previous heap after removing the maximum (12) and placing the last element (2) in its position (the root):

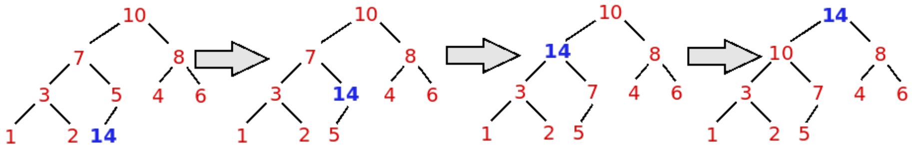


- **removeMax():** time $\mathcal{O}(\log n)$

# Heaps - Operation insert($x$)

- To **insert** an element:
  - Place it immediately after the last occupied position, the first free position in the array (the tree remains complete).
  - The element "rises" (**upHeap**), swapping with its parent, until the heap invariant is restored.
  - At most, this traverses the height of the tree, which is $\mathcal{O}(\log n)$.

An example for the insertion of 14 into the heap from the previous slide:

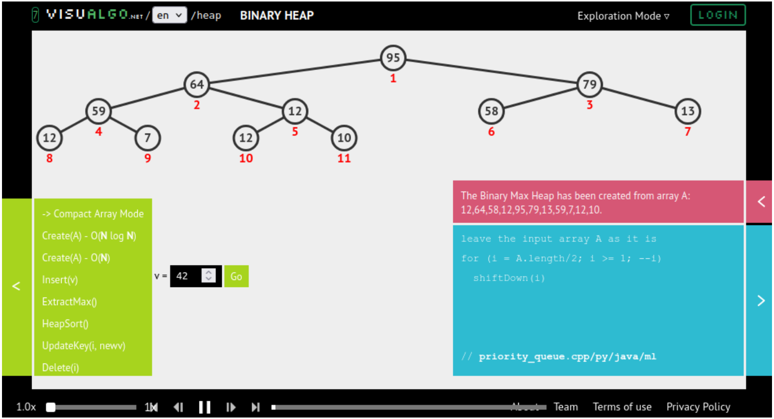



- **insert($x$):** time $\mathcal{O}(\log n)$

# Heaps - Visualization

- You can visualize insertion and removal in heaps (try the indicated URL):

https://visualgo.net/en/heap



# Building a Heap

- How can we **build a heap** from an initially general unsorted array with $n$ elements?

- A first answer could be: **insert all elements into a new heap**
  - This would cost $n$ insertions.
    Since each one costs $\mathcal{O}(\log n)$ the total cost would be $\mathcal{O}(n \log n)$

- But what if we want reuse the same initial array and just *"heapify"* it? We could now have two options:
  1. Start at the beginning of the array (top of the heap) and move forward towards the end. At each iteration, call **upHeap** on each item.

     At each step, the the items before the current item in the array form a valid heap, and moving the next item up places it into a valid position in the heap. After moving up each node, all items satisfy the heap property.

  2. Start at the end of the array and move backwards towards the front. At each iteration, call **downHeap** on each item.

     Can you see why it works? At each step, what can we say about the the items after the current item?

- Both approaches produce a valid heap. **Which one is faster?**

# Building a Heap

- Both **upHeap** and **downHeap** depend on the height of the tree
  - At most they will go up or down $\mathcal{O}(\log n)$ times
- A first examination of the cost would therefore give us the bounds:
  1. Calling $n$ times **upHeap** costs $\mathcal{O}(n \log n)$
  2. Calling $n$ times **downHeap** costs $\mathcal{O}(n \log n)$
- While correct ($\mathcal{O}$ provides an upper bound), one of these bounds is **not asymptotically tight** (and in fact its cost is $\mathcal{O}(n)$)
- Why? We will apply the operation to all nodes in the array, but:
  1. For **upHeap** the cost is proportional to the distance to the top (expensive for nodes at the bottom of the tree)
  2. For **downHeap** the cost is proportional to the distance to the bottom (expensive for nodes at the top of the tree)
- Both operations are $\mathcal{O}(\log n)$ in the worst case but **only one node is at the top whereas half the nodes lie in the bottom layer.**

  It should therefore be no surprise that if we have to apply an operation to every node, we would **prefer downHeap over upHeap**

# Building a Heap

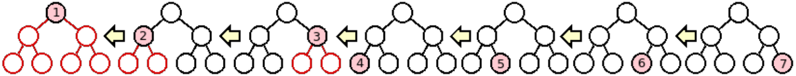
- Let's see this more visually on a heap with 7 nodes:

1. Calling **upHeap** on all nodes will do it from 1 to 7


At most this will cost:

- ★ 1 will **never** move up (max cost = 0)
- ★ 2 and 3 will move up **at most once** (max cost = 1+1 = 2)
- ★ 4, 5, 6 and 7 move up **at most twice** (max cost = 2+2+2+2 = 8)
- ★ **Total cost = 0+2+8 = 10**

2. Calling **downHeap** on all nodes will do it from 7 to 1




At most this will cost:

- ★ 4, 5, 6 and 7 will **never** move down (max cost = 0)
- ★ 2 and 3 will move down **at most once** (max cost = 1+1 = 2)
- ★ 1 will move down **at most twice** (max cost = 2)
- ★ **Total cost = 0+2+2 = 4** **(less than upHeap)**

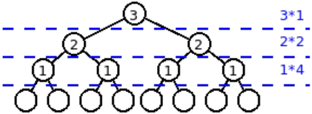
# Building a Heap

- Our algorithm to **build an heap** for any array of size $n$ could be:

```
// No need to call on bottom level (start on second to last level)
for (int i=n/2; i>=1; i--)
   downHeap(i)
```

You can visualize it: https://visualgo.net/en/heap [create(A) - O(N)]

- This has **linear cost** on $n$, that is, $\mathcal{O}(n)$
- Let's sketch a proof. For simplicity of analysis let's consider a **complete tree**, that is, an array with $n = 2^{h+1} - 1$ items
  - Level 0 has 1 node, level 1 has 2 nodes, ..., level $h$ has $2^h$ nodes
  - Second to last level: $2^{h-1}$ nodes cost at most 1
  - Third to last level: $2^{h-2}$ nodes cost at most 2
  - ...
  - First level: 1 node costs at most $h$



# Building a Heap

- Let's call the total cost $T(n) = T(2^{h+1} - 1)$. Then:

$$T(n) \leq \sum_{i=1}^{h} i \times 2^{h-i} = \sum_{i=1}^{h} i \times \frac{2^h}{2^i} = 2^h \sum_{i=1}^{h} \frac{i}{2^i} \leq 2^h \sum_{i=1}^{\infty} \frac{i}{2^i}$$

- Let's focus on the summation of $S = \sum_{i=1}^{\infty} \frac{i}{2^i}$

$$\textbf{(1)}\ S = 1/2 + 2/4 + 3/8 + 4/16 + \dots$$

$$\textbf{(2)}\ S/2 = 1/4 + 2/8 + 3/16 + 4/32 + \dots$$

$$\textbf{(1)-(2)}\ S/2 = 1/2 + 1/4 + 1/8 + 1/16 + \dots$$

This is a geometric series $a + ar + ar^2 + ar^3 + \dots$ with $a = 1/2$ and $r = 1/2$
When $|r| < 1$ we know that this sum converges to $\frac{a}{1-r}$, so $S/2 = \frac{1/2}{1-1/2} = 1$

- $S = 2$, so finally:

$$T(n) \leq 2^h \sum_{i=1}^{\infty} \frac{i}{2^i} = 2^h \times 2 = 2^{h+1} = n + 1 \in \mathcal{O}(n) \quad \square$$

# HeapSort

- Heaps suggest an **obvious sorting algorithm**. To sort $n$ elements, simply do the following:
  1. Create a heap with the $n$ elements
  2. Remove the $n$ elements from the heap one by one
- Since the elements are removed in order of priority, they will appear... in **descending order**!
- What is the cost of this process?
  1. Building the heap costs $\mathcal{O}(n)$
  2. Removing one by one costs $\mathcal{O}(n \log n)$ [downHeap from root]
- The cost will be dominated by (2) and the sort will cost $\mathcal{O}(n \log n)$
- This algorithm (in its essence) is known as **HeapSort**.
- You can visualize it here: https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

# Heaps - Implementation

- Let's see a simple implementation of heaps in C++
- We will implement a generic `maxHeap`<T> supporting any type using the natural comparator with 4 methods:
  - `bool isEmpty()`: is the heap empty?
  - `T max()`: return max vale
  - `T removeMax()`: return and remove max value
  - `bool insert(T value)`: insert a new value

# Heaps - Implementation

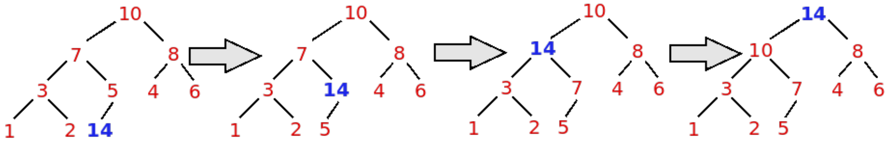
- The attributes, constructor and easy methods:

```
template <class T> class MaxHeap {
private:
  std::vector<T> data; // Items between 1 and size
  int size;            // Number of items
  T none;              // What to return when there are no items

public:
  // Capacity cap and value N used for indicating no element
  MaxHeap(int cap, T n) : data(cap+1), size(0), none(n) {}

  // Empty heap?
  bool isEmpty() {
    return (size==0);
  }

  // Return (without removing) maximum element
  T max() {
    if (isEmpty()) return none;
    return data[1];
  }
  //..
}
```

# Heaps - Implementation

- **Insert** an element into the heap: place it at the end of the array and call *upHeap* until the element reaches its position:

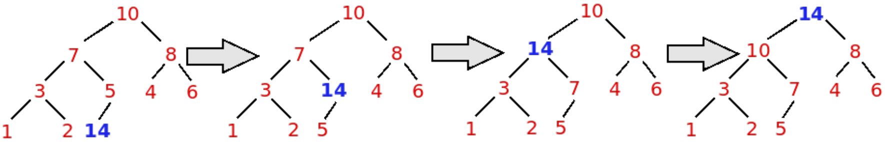
```
// Insert an element in the heap (true on sucess)
bool insert(T value) {
  if (size+1 >= (int)data.size()) return false;
  size++;
  data[size] = value;
  upHeap(size);
  return true;
}
```



# Heaps - Implementation

- **upHeap:** move the element up while it is larger than the parent

```
// Make an element go up until its position
void upHeap(int i) {
  // While the element is greater than parent and not on root
  while (i>1 && data[i] > data[i/2]) {
    std::swap(data[i], data[i/2]);
    i = i/2;
  }
}
```

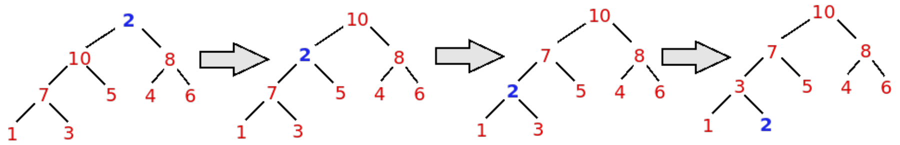



- std::swap is an existing C++ function

# Heaps - Implementation

- **Remove** an element from the heap: return the root; place the last element at the root and call *downHeap* until the element reaches its position:

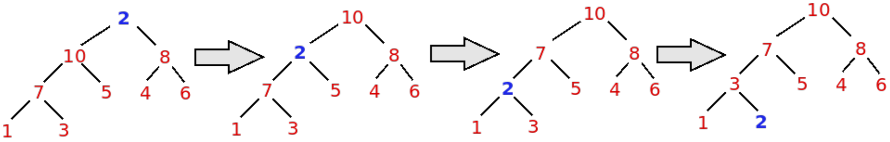
```
// Remove and return maximum
T removeMax() {
  if (isEmpty()) return none;
  T max = data[1];
  data[1] = data[size];
  size--;
  downHeap(1);
  return max;
}
```



# Heaps - Implementation

- **downHeap:** move the element down while one of the children is larger (and swap with the larger child)

```
// Make an element go down until its position
void downHeap(int i) {
  while (2*i <= size) { // While still on the limits of the heap
    int j = i*2;
    // Choose largest child (position i*2 or i*2+1)
    if (j<size && data[j+1] > data[j]) j++;
    // If node is already larger or equal than largest child, stop
    if (data[i] >= data[j]) break;
    // Otherwise, swap with largest child
    std::swap(data[i], data[j]);
    i = j;
  }
}
```



# Heaps - Usage Example

- Let's now see some usage examples.

```cpp
#include <iostream>
#include "maxHeap.h"

int main() {
  // Create a heap with capacity for 10 integers (-1 for no item)
  MaxHeap<int> h(10, -1);

  // Create array for 10 integers
  std::vector<int> v = {10, 4, 3, 12, 9, 1, 7, 11, 5, 8};

  // Insert on heap all elements of array
  for (auto x : v) h.insert(x);

  // Remove elements one by one and print
  for (int i=0; i<10; i++) std::cout << h.removeMax() << " ";
  std::cout << std::endl;

  return 0;
}
```

```
12 11 10 9 8 7 5 4 3 1
```

# Heaps - Usage Example

- Any comparable type can be used, not just integers.
- Here is an example with strings:

```
// Create a heap h (for strings)
MaxHeap<std::string> h(5, "---");

// Create an array of 5 strings
auto v = {"heap", "tree", "stack", "queue", "deque"};

// Insert on heap all elements of array
for (auto s : v) h.insert(s);

// Remove elements one by one and print
for (int i=0; i<5; i++) std::cout << h.removeMax() << " ";
std::cout << std::endl;
```

```
tree stack queue heap deque
```

# Priority Queues in C++ Standard

- C++ standard directly implement priority queues as max heaps
- The name of the container is priority_queue
- To use it you should use #include <queue>

```
// Create a priority_queue as a heap
std::priority_queue<int> h;

// Create array for 10 integers
std::vector<int> v = {10, 4, 3, 12, 9, 1, 7, 11, 5, 8};

// Insert on heap all elements of array
for (auto x : v) h.push(x);

// Remove elements one by one and print
for (int i=0; i<10; i++) {
  std::cout << h.top() << " "; // returns, but does not remove
  h.pop();                     // removes, but does not return
}
std::cout << std::endl;
```

```
12 11 10 9 8 7 5 4 3 1
```

# Priority Queues in C++ Standard

- Any comparable type can be used, not just integers.
- Here is an example with strings:

```
// Create a priority_queue as a heap h (for strings)
std::priority_queue<std::string> h;

// Create an array of 5 strings
auto v = {"heap", "tree", "stack", "queue", "deque"};

// Insert on heap all elements of array
for (auto s : v) h.push(s);

// Remove elements one by one and print
for (int i=0; i<5; i++) {
  std::cout << h.top() << " ";
  h.pop();
}
std::cout << std::endl;
```

```
tree stack queue heap deque
```

# Priority Queues in C++ Standard

- We can for instance use a custom comparator
- The template is `priority_queue<type, container, comparator>` (code here with `using namespace std` for spacing reasons)

```
// Custom comparator (using lambda function)
auto comp = [](string a, string b) {return b.size() > a.size();};

// Create a heap h (for strings)
priority_queue<string, vector<string>, decltype(comp)> h(comp);

// Create an array of 5 strings
auto v = {"aa", "bbbbb", "ccc", "dddd", "e"};

// Insert on heap all elements of array
for (auto s : v) h.push(s);

// Remove elements one by one and print
for (int i=0; i<5; i++) {cout << h.top() << " "; h.pop();}

cout << endl;
```

```
bbbbb dddd ccc aa e
```

# Heaps in C++ Standard

- C++ also allows to directly use a container as a heap
- You can call make_heap to make a heap in linear time

```cpp
std::vector<int> v = {10, 4, 3, 12, 9, 1, 7, 11, 5, 8};

// Creates heap from the vector (needs a range as as argument)
std::make_heap(v.begin(), v.end());
for (auto x : v) std::cout << x << " "; std::cout << std::endl;

// Pops the maximum (max is pushed to the end of the range)
// Effectively range that stays a heap decreases in size
std::pop_heap(v.begin(), v.end());
for (auto x : v) std::cout << x << " "; std::cout << std::endl;

// Pushes the last element to its heap position
std::push_heap(v.begin(), v.end());
for (auto x : v) std::cout << x << " ";  std::cout << std::endl;
```

```
12 11 7 10 9 1 3 4 5 8
11 10 7 8 9 1 3 4 5 12
12 11 7 8 10 1 3 4 5 9
```

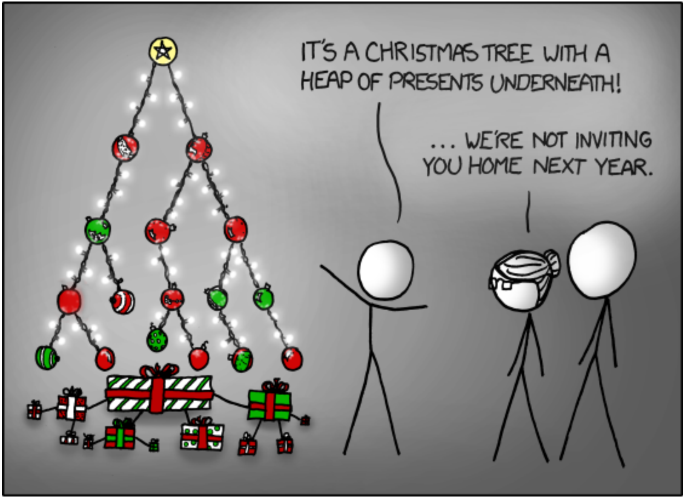
# Heaps in C++ Standard

- For instance, with this API, doing an **HeapSort** as described is just:
  - Call `make_heap` on a container with *n* items
  - Call `pop_heap` *n* times on the container
    *(each time on a smaller range)*

```
std::vector<int> v = {10, 4, 3, 12, 9, 1, 7, 11, 5, 8};

// Creates heap from the vector [in O(n)]
std::make_heap(v.begin(), v.end());

// Pops max on consecutively smaller ranges (in O(n log n))
for (int i=0; i<(int)v.size(); i++)
  std::pop_heap(v.begin(), v.end()-i);

// Show vector at the end
for (auto x : v) std::cout << x << " ";
std::cout << std::endl;
```

```
1 3 4 5 7 8 9 10 11 12
```

# Heaps




(image from https://xkcd.com/835/)

"Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt."