# CSC 212: Data Structures and Abstractions

## 09: Priority Queues

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

# Announcements

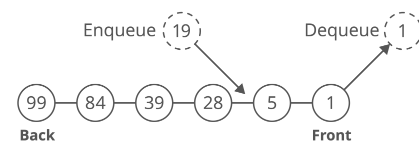
- Assignment 2
  - ✓ purpose of the assignment is to learn how to solve problems with stacks, queues, and deques
    - at least one of those data structures should be used on each of the problems
  - ✓ designing a good algorithm is more important than coding
    - try designing your own solution first, test it on paper
- Spring break plans?
  - ✓ ideas for further developing your C++ skills
    - **OOP, templates, pointers:** implement your own templated versions of stack, queue, deque, and priority queues (use rigorous testing)
    - **problem solving:** solve as many Kattis/LeetCode problems as possible



# Priority queues

# Priority queues

- Definition
  - ✓ a **<u>priority queue</u>** is a linear data structure that functions like a queue but with priorities assigned to elements
  - ✓ elements with **higher priority** are dequeued before elements with lower priority
- Main Operations
  - ✓ **enqueue**: add an element with an associated priority
  - ✓ **dequeue**: remove and return the highest priority element
- Applications
  - ✓ algorithms for graphs
  - ✓ event-driven simulation
  - ✓ search methods in artificial intelligence
  - ✓ job scheduling in operating systems, etc.



# Implementation

- Key-value pairs
  - ✓ elements in a priority queue can be implemented as a collection of **<key,value>** pairs
    - **key**: determines priority, **value**: associated data

| Operations (min-pq) | Return value |
|---|---|
| enqueue(5, A) | |
| enqueue(10, D) | |
| enqueue(3, B) | |
| dequeue() | (3, B) |
| enqueue(7, C) | |
| dequeue() | (5, A) |
| dequeue() | (7, C) |
| size() | 1 |
| isEmpty() | FALSE |



# Implementation

- Using arrays
  - ✓ ensure enqueue and dequeue work efficiently
  - ✓ array can be fixed-length or a dynamic array (additional cost)
- Considerations
  - ✓ highest priority can be defined in different ways
    - in a **max-priority queue**, the highest priority is the largest priority
    - in a **min-priority queue**, the highest priority is the smallest priority
  - ✓ for equal priorities, the order of elements is determined by the underlying implementation
    - in some implementations, equal priority elements are served following FIFO order
    - in other implementations, the order of elements with the same priority is undefined
  - ✓ underflow: throw an error when calling dequeue on an empty queue
  - ✓ overflow: throw an error when calling enqueue on a full queue



# Implementation

- Array-based (unsorted array)
  - ✓ enqueue at the end — $O(1)$ cost (amortized cost if using a dynamic array)
  - ✓ dequeue (extract max/min) — $O(n)$ cost
    - requires searching the entire array
- Array-based (sorted array)
  - ✓ enqueue at position — $O(n)$ cost
    - requires finding position for insertion and shifting elements
  - ✓ dequeue (extract max/min) — $O(1)$ cost
- Binary heap (array)
  - ✓ most common and efficient
  - ✓ enqueue — $O(\log n)$ cost
  - ✓ dequeue (extract max/min) — $O(\log n)$ cost
  - ✓ can also build a binary heap from an array in $O(n)$ cost



## std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

The priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided `Compare` can be supplied to change the ordering, e.g. using `std::greater<T>` would cause the smallest element to appear as the `top()`.

Working with a `priority_queue` is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

**Member functions**

| | |
|---|---|
| (constructor) | constructs the `priority_queue` (public member function) |
| (destructor) | destructs the `priority_queue` (public member function) |
| **operator=** | assigns values to the container adaptor (public member function) |
| **Element access** | |
| **top** | accesses the top element (public member function) |
| **Capacity** | |
| **empty** | checks whether the container adaptor is empty (public member function) |
| **size** | returns the number of elements (public member function) |
| **Modifiers** | |
| **push** | inserts element and sorts the underlying container (public member function) |
| **push_range** (C++23) | inserts a range of elements and sorts the underlying container (public member function) |
| **emplace** (C++11) | constructs element in-place and sorts the underlying container (public member function) |
| **pop** | removes the top element (public member function) |
| **swap** (C++11) | swaps the contents (public member function) |

```
#include <iostream>
#include <queue>

int main()
{
    std::priority_queue<int> pq1;
    pq1.push(5);
    std::cout << "pq1.size() = " << pq1.size() << '\n';

    std::priority_queue<int> pq2 {pq1};
    std::cout << "pq2.size() = " << pq2.size() << '\n';

    std::vector<int> vec {3, 1, 4, 1, 5};
    std::priority_queue<int> pq3 {std::less<int>(), vec};
    std::cout << "pq3.size() = " << pq3.size() << '\n';

    for (std::cout << "pq3 : "; !pq3.empty(); pq3.pop())
        std::cout << pq3.top() << ' ';
    std::cout << '\n';
}
```

https://en.cppreference.com/w/cpp/container/priority_queue/priority_queue

# Practice

- What is the output of this code?

```cpp
#include <iostream>
#include <queue>
#include <utility> // for std::pair

int main() {
    // default priority_queue - max-heap behavior
    std::priority_queue<std::pair<int, std::string>> pq;

    pq.push(std::make_pair(3, "Job 1"));
    pq.push(std::make_pair(1, "Job 2"));
    pq.push(std::make_pair(5, "Job 3"));
    pq.pop();
    pq.push(std::make_pair(2, "Job 4"));
    pq.pop();
    pq.push(std::make_pair(7, "Job 5"));
    pq.pop();
    pq.pop();
    pq.push(std::make_pair(7, "Job 6"));
    pq.push(std::make_pair(7, "Job 7"));

    while (! pq.empty()) {
        std::pair<int, std::string> top = pq.top();
        std::cout << top.second << std::endl;
        pq.pop();
    }

    return 0;
}
```



# Practice

- What is this function doing?
  - ✓ what is the time complexity?
    - depends on the running time of the priority queue operations

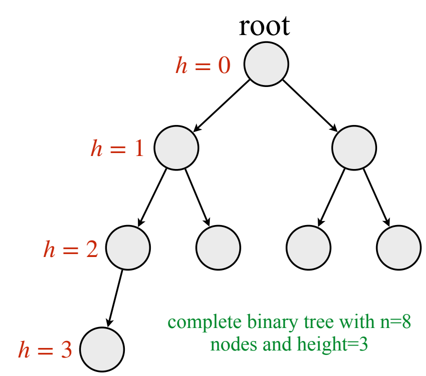
```cpp
void foo(std::vector<int>& vec) {
    int n = vec.size();
    std::priority_queue<int> pq;

    for (auto& elem : vec)
        pq.push(elem);

    while (!pq.empty()) {
        vec[--n] = pq.top();
        pq.pop();
    }
}
```



# Binary heaps

# Complete binary tree

- Binary tree
  - ✓ tree data structure in which each **node** has at most two children, referred to as the left child and the right child
- Complete binary tree
  - ✓ binary tree in which every level, except possibly the last, is completely filled
    - all nodes in the last level are as far left as possible
    - it can have between 1 and $2^h$ nodes at the last level $h$




The height of a complete binary tree with $n$ nodes is $\lfloor \log_2 n \rfloor$

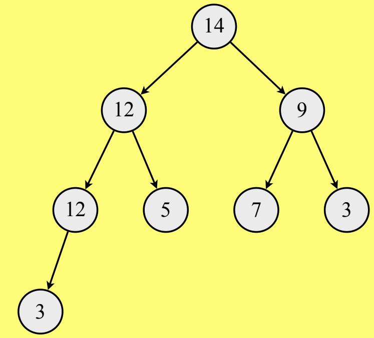
# Binary heap

- Definition
  - ✓ **structure property**: a binary heap is a **complete binary tree**
  - ✓ **heap property**: each node's value is greater/smaller than or equal to its children's
    - a binary heap can be a **max-heap** (greater or equal) or a **min-heap** (smaller or equal)
- Considerations
  - ✓ the height of a binary heap is $\lfloor \log_2 n \rfloor$
  - ✓ the number of nodes at each level $h$ is at most $2^h$
  - ✓ the number of nodes in a heap is at most: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$



# Max-heap example

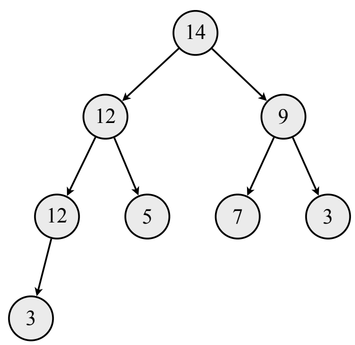
- Check:
  - ✓ structure property
  - ✓ heap-order property
- Add 3 elements
  - ✓ without violating properties
- Change 2 values
  - ✓ that violate the heap property






# Array representation

- A binary heap can be represented as an array
  - ✓ **root** is at index 0
  - ✓ **last element** is at index $n-1$
- For any node at index $i$:
  - ✓ **left child** is at index $2i+1$
  - ✓ **right child** is at index $2i+2$
  - ✓ **parent** is at index $(i-1)//2$




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 12 | 9 | 12 | 5 | 7 | 3 | 3 | | | | | | | | |

`n=8, capacity=16`



# Enqueue (max-heap)

- Algorithm (min-heap is analogous)
  1. append the element to the end of the array
  2. compare the element with its parent
  3. if the element is greater than its parent, swap them
  4. repeat 2-3 until the element is in the correct position (heap-order restored)

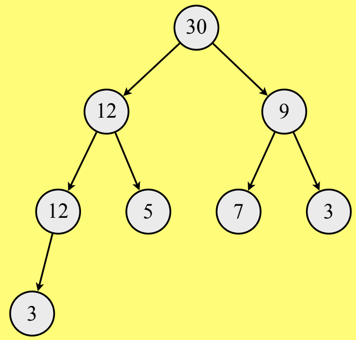
steps 2-3-4 can be implemented as a function called **up-heap**

- Time complexity
  - ✓ how many swaps are necessary? $O(\log n)$

https://visualgo.net/en/heap

# Practice (max-heap)

- Enqueue 20
  - ✓ show resulting array
- Enqueue 1
  - ✓ show resulting array
- Enqueue 50
  - ✓ show resulting array



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 | 12 | 9 | 12 | 5 | 7 | 3 | 3 | | | | | | | | |



# Dequeue (max-heap)

- Algorithm (min-heap is analogous)
  1. replace the root with the last element
  2. remove the last element from the array
  3. compare the root with its children
  4. if the root is less than either child, swap it with the larger child
  5. repeat 3-4 until the root is in the correct position (heap-order restored)

steps 3-4-5 can be implemented as a function called **down-heap**

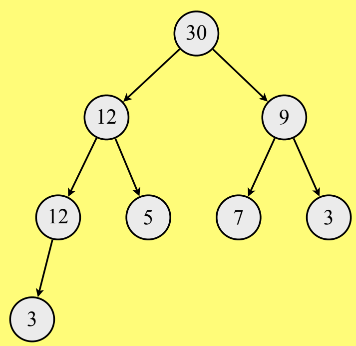
- Time complexity
  - ✓ how many swaps are necessary? $O(\log n)$

https://visualgo.net/en/heap



# Practice

- Dequeue
  - ✓ show resulting array
- Dequeue
  - ✓ show resulting array
- Dequeue
  - ✓ show resulting array



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 | 12 | 9 | 12 | 5 | 7 | 3 | 3 | | | | | | | | |



# Performance

| Method | Unsorted Array | Sorted Array | Binary Heap |
|---|---|---|---|
| **Enqueue** | `O(1)` | `O(n)` | `O(log n)` |
| **Dequeue** | `O(n)` | `O(1)` | `O(log n)` |
| **Max** | `O(n)` | `O(1)` | `O(1)` |
| **Size** | `O(1)` | `O(1)` | `O(1)` |
| **IsEmpty** | `O(1)` | `O(1)` | `O(1)` |