CSC 212: Data Structures and Abstractions

09: Priority Queues

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

THINK BIG WE DO℠

Practice

- A server runs tasks in order. Each day, the server can run for at most T minutes. Each task has a duration. Given n tasks, write a program that outputs how many tasks can the server finish before exceeding T?
- Input:
  - ✓ first line: $n\ T$, $\quad n \leq 50$, $T \leq 500$
  - ✓ second line: n task times
- Output:
  - ✓ number of tasks that can be completed

```
sample input:
6 180
45 30 55 20 90 20

sample output:
4
```


Practice

- A server runs tasks in a specific order. Each day, the server can run for at most T minutes. Each task has a duration. The server runs tasks by alternating between the first and last remaining tasks. Given n tasks, write a program that outputs how many tasks can the server finish before exceeding T?
- Input:
  - ✓ first line: $n\ T$, $\quad n \leq 50$, $T \leq 500$
  - ✓ second line: n task times
- Output:
  - ✓ number of tasks that can be completed

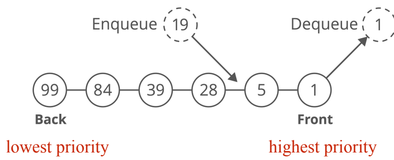
```
sample input:
6 180
45 30 55 20 90 20

sample output:
3
```


Priority queues

Priority queues

- Definition
  - a **priority queue** is a data structure similar to a queue, but where each element has an associated priority
  - elements with **higher priority** are removed before elements with lower priority
- Main Operations
  - **enqueue**: add element with an associated priority
  - **dequeue**: remove element with highest priority
- Applications
  - algorithms for graphs
  - event-driven simulation
  - search methods in artificial intelligence
  - job scheduling in operating systems, etc.



Implementation

- Representation
  - elements in a priority queue can be implemented as a collection of **<key,value>** pairs
    - **key**: determines the priority, used for comparison
    - **value**: actual data/payload associated with that priority

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

Practice

- What is the output of the following code?

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

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

    pq.push({3, "Low priority"});
    pq.push({9, "High priority"});
    pq.push({5, "Medium priority"});

    while (!pq.empty()) {
        std::cout << pq.top().first << ": " << pq.top().second << "\n";
        pq.pop();
    }

    return 0;
}
```


Practice

- What is the output of this code?

```
#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;
}
```

Implementation

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

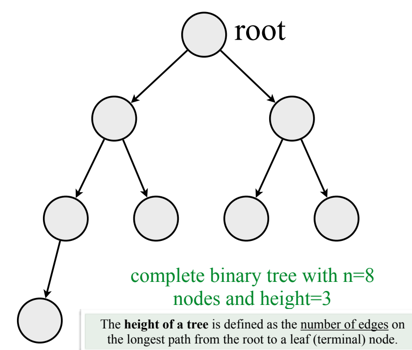
Implementation

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

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



complete binary tree with n=8 nodes and height=3

The **height of a tree** is defined as the number of edges on the longest path from the root to a leaf (terminal) node.

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

Practice

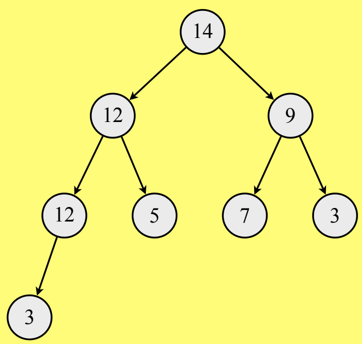
- Consider a complete binary tree of height h
  - what is n_{max}, the max number of nodes in the tree as a function of h?
    - hint: use a summation formula
  - what is n_{min}, the min number of nodes in the tree as a function of h?
- For a complete binary tree the following inequality holds: $n_{min} \leq n < n_{max} + 1$
  - take the logarithm (base 2) of this inequality and express h in terms of n

Binary heap

- Definition
  - **structure property**: a binary heap is a **complete binary tree**
  - **heap property**: a binary heap can be:
    - **max-heap**: each node's value is greater than or equal to its children's values
    - **min-heap**: each node's value is smaller than or equal to its children's values
- 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: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$

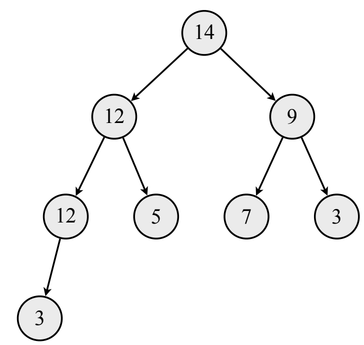
Max-heap example

- 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`

```
template <typename T>
class PriorityQueue {
    private:
        T *arr;
        size_t capacity;
        size_t size;

        size_t parent(size_t i) { return (i-1) / 2; }
        size_t left(size_t i) { return 2*i + 1; }
        size_t right(size_t i) { return 2*i + 2; }

        void upHeap(size_t i);
        void downHeap(size_t i);

    public:
        PriorityQueue(size_t cap);
        ~PriorityQueue();

        void enqueue(const T& val);
        void dequeue();

        T& front();
        size_t get_size() { return size; }
        size_t get_capacity() { return capacity; }
        bool empty() { return size == 0; }
};
```


Enqueue (max-heap)

- Algorithm (min-heap is analogous)
  1. append the element to the end of the array
  2. for each node from parent(n-1) to the root
  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)

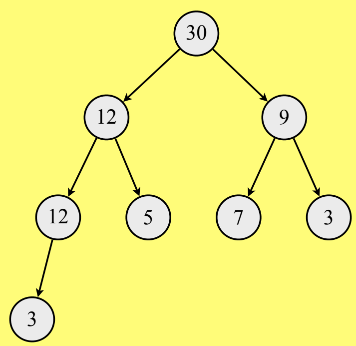
steps 2-3-4 can be implemented as a function called **upHeap**

- Time complexity
  - ✓ how many swaps are necessary in the worst case? $\Theta(\log n)$

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								

Enqueue

```
template <typename T>
void PriorityQueue<T>::enqueue(const T& val) {
    if (size == capacity) {
        throw std::out_of_range("PriorityQueue is full");
    }
    arr[size] = val;
    size ++;
    upHeap(size-1);
}


template <typename T>
void PriorityQueue<T>::upHeap(size_t idx) {
    while (idx > 0) {
        size_t p = parent(idx);
        if (arr[idx] > arr[p]) {
            std::swap(arr[idx], arr[p]);
            idx = p;
        } else {
            break;
        }
    }
}
```

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)

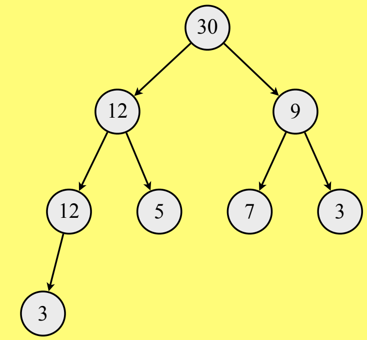
steps 3-4-5 can be implemented as a function called **downHeap**

- Time complexity
  - ✓ how many swaps are necessary in the worst case? $\Theta(\log n)$

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								

Dequeue

```cpp
template <typename T>
void PriorityQueue<T>::dequeue() {
    if (size == 0) {
        throw std::out_of_range("PriorityQueue is empty");
    }
    arr[0] = arr[size-1];
    size--;
    downHeap(0);
}

template <typename T>
void PriorityQueue<T>::downHeap(size_t i) {
    while (true) {
        size_t largest = i;
        size_t l = left(i);
        size_t r = right(i);
        if (l < size && arr[l] > arr[largest]) {
            largest = l;
        }
        if (r < size && arr[r] > arr[largest]) {
            largest = r;
        }
        if (largest != i) {
            std::swap(arr[i], arr[largest]);
            i = largest; // move down to largest
        } else {
            break;
        }
    }
}
```


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)