Containers: Priority Queues

Jordi Cortadella and Jordi Petit
Department of Computer Science
A priority queue

• A priority queue is a queue in which each element has a priority.

• Elements with higher priority are served before elements with lower priority.

• It can be implemented as a vector or a linked list. For a queue with \( n \) elements:
  – Insertion is \( O(n) \).
  – Extraction is \( O(1) \).

• A more efficient implementation can be proposed in which insertion is \( O(\log n) \): binary heap.
• Complete binary tree (except at the bottom level).
• Height \( h \): between \( 2^h \) and \( 2^{h+1} - 1 \) nodes.
• For \( N \) nodes, the height is \( O(\log N) \).
• It can be represented in a vector.
Binary Heap

Locations in the vector:

Heap-Order Property: the key of the parent of X is smaller than (or equal to) the key in X.
Binary Heap

Two main operations on a binary heap:
- Insert a new element
- Remove the min element

Both operations must preserve the properties of the binary heap:
- Completeness
- Heap-Order property
Binary Heap: insert 14

Insert in the last location

... and bubble up ...

done!
Binary Heap: remove min

Extract the min element and move the last one to the root of the heap

... and bubble down ...
Binary Heap: remove min

Containers: Priority Queues

© Dept. CS, UPC
// Elem must be a comparable type
template <typename Elem>
class PriorityQueue {
private:
    vector<Elem> v;  // Table for the heap (location 0 not used)

public:

    // Constructor (one fake element in the vector)
    PriorityQueue() {
        v.push_back(Elem());
    }

    int size() {
        return v.size() - 1;  // The 0 location does not count
    }

    bool empty() {
        return size() == 0;
    }
public:

Elem minimum() {
    assert(not empty());
    return v[1];
}

void insert(const Elem& x) {
    v.push_back(x); // Put element at the bottom
    bubble_up(size()); // ... and bubble up
}

Elem remove_min () {
    assert(not empty());
    Elem x = v[1]; // Store the element at the root
    v[1] = v.back(); // Move the last element to the root
    v.pop_back();
    bubble_down(1); // ... and bubble down
    return x;
}
private:

void bubble_up(int i) {
    if (i != 1 and v[i/2] > v[i]) {
        swap(v[i], v[i/2]);
        bubble_up(i/2);
    }
}

void bubble_down(int i) {
    int n = size();
    int c = 2*i;
    if (c <= n) {
        if (c+1 <= n and v[c+1] < v[c]) c++;
        if (v[i] > v[c]) {
            swap(v[i], v[c]);
            bubble_down(c);
        }
    }
}
Binary Heap: complexity

• Bubble up/down operations do at most $h$ swaps, where $h$ is the height of the tree and

$$h = \lfloor \log_2 N \rfloor$$

• Therefore:
  – Getting the min element is $O(1)$
  – Inserting a new element is $O(\log N)$
  – Removing the min element is $O(\log N)$
Binary Heap: other operations

• Let us assume that we have a method to know the location of every key in the heap.

• Increase/decrease key:
  – Modify the value of one element in the middle of the heap.
  – If decreased \(\rightarrow\) bubble up.
  – If increased \(\rightarrow\) bubble down.

• Remove one element:
  – Set value to \(-\infty\), bubble up and remove min element.
Heaps are sometimes constructed from an initial collection of $N$ elements. How much does it cost to create the heap?

- Obvious method: do $N$ insert operations.
- Complexity: $O(N \log N)$

Can it be done more efficiently?
Building a heap from a set of elements

Containers: Priority Queues

© Dept. CS, UPC
Building a heap: implementation

```cpp
// Constructor from a collection of items
PriorityQueue(const vector<Elem>& items) {
    v.push_back(Elem());
    for (auto& e: items) v.push_back(e);
    for (int i = size()/2; i > 0; --i) bubble_down(i);
}
```

Sum of the heights of all nodes:
- 1 node with height $h$
- 2 nodes with height $h - 1$
- 4 nodes with height $h - 2$
- $2^i$ nodes with height $h - i$

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

$$S = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \cdots + 2^{h-1}(1)$$

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \cdots + 2^h(1)$$

Subtract the two equations:

$$S = -h + 2 + 4 + 8 + \cdots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1) = O(N)$$

A heap can be built from a collection of items in linear time.
Heap sort

```
template <typename T>
void heapSort(vector<T>& v) {
    PriorityQueue<T> heap(v);
    for (T& e: v) e = heap.remove_min();
}
```

- Complexity: $O(n \log n)$
  - Building the heap: $O(n)$
  - Each removal is $O(\log n)$, executed $n$ times.