Priority Queues and Heaps
Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - `insertItem(k, o)` inserts an item with key k and element o
  - `removeMin()` removes the item with the smallest key
- Additional methods
  - `minKey(k, o)` returns, but does not remove, the smallest key of an item
  - `minElement()` returns, but does not remove, the element of an item with smallest key
  - `size()`, `isEmpty()`
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
Total Order Relation

- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Two distinct items in a priority queue can have the same key.

Mathematical concept of total order relation ≤

- **Reflexive** property: $x \leq x$
- **Antisymmetric** property: $x \leq y \land y \leq x \Rightarrow x = y$
- **Transitive** property: $x \leq y \land y \leq z \Rightarrow x \leq z$
Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation.

- A generic priority queue uses a comparator as a template argument, to define the comparison function ($<$, $=$, $>$).

- The comparator is external to the keys being compared. Thus, the same objects can be sorted in different ways by using different comparators.

- When the priority queue needs to compare two keys, it uses its comparator.
Using Comparators in C++

A comparator class overloads the "()" operator with a comparison function.

Example: Compare two points in the plane lexicographically.

```cpp
class LexCompare {
    public:
        int operator()(Point a, Point b) {
            if (a.x < b.x) return -1;
            else if (a.x > b.x) return +1;
            else if (a.y < b.y) return -1;
            else if (a.y > b.y) return +1;
            else return 0;
        }
};
```

To use the comparator, define an object of this type, and invoke it using its "()" operator:

Example of usage:

```cpp
Point p(2.3, 4.5);
Point q(1.7, 7.3);
LexCompare lexCompare;
if (lexCompare(p, q) < 0)
    cout << "p less than q";
else if (lexCompare(p, q) == 0)
    cout << "p equals q";
else if (lexCompare(p, q) > 0)
    cout << "p greater than q";
```
We can use a priority queue to sort a set of comparable elements:

- Insert the elements one by one with a series of \texttt{insertItem}(e, e) operations.
- Remove the elements in sorted order with a series of \texttt{removeMin}() operations.

The running time of this sorting method depends on the priority queue implementation.

Algorithm \texttt{PQ-Sort}(S, C)

\begin{itemize}
  \item \textbf{Input} sequence \textit{S}, comparator \textit{C} for the elements of \textit{S}
  \item \textbf{Output} sequence \textit{S} sorted in increasing order according to \textit{C}
\end{itemize}

\begin{verbatim}
P \gets \text{priority queue with comparator } C
\textbf{while} \! S.\text{isEmpty} ()
  \begin{itemize}
    \item \textit{e} \gets S.\text{remove} (S.\text{first} ())
    \item \textit{P}.\text{insertItem}(\textit{e}, \textit{e})
  \end{itemize}
\textbf{while} \! P.\text{isEmpty} ()
  \begin{itemize}
    \item \textit{e} \gets P.\text{minElement}()
    \item P.\text{removeMin}()
    \item S.\text{insertLast} (\textit{e})
  \end{itemize}
\end{verbatim}
Sequence-based Priority Queue

Implementation with an unsorted list

Performance:
- **insertItem**
  - takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
- **removeMin, minKey and minElement**
  - take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

Implementation with a sorted list

Performance:
- **insertItem**
  - takes $O(n)$ time since we have to find the place where to insert the item
- **removeMin, minKey and minElement**
  - take $O(1)$ time since the smallest key is at the beginning of the sequence
Selection-Sort

Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence.

Running time of Selection-sort:
- Inserting the elements into the priority queue with $n$ `insertItem` operations takes $O(n)$ time.
- Removing the elements in sorted order from the priority queue with $n$ `removeMin` operations takes time proportional to $1 + 2 + \ldots + n$

Selection-sort runs in $O(n^2)$ time.
**Insertion-Sort**

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence.

- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with $n$ `insertItem` operations takes time proportional to $1 + 2 + \ldots + n$.
  - Removing the elements in sorted order from the priority queue with a series of $n$ `removeMin` operations takes $O(n)$ time.

- Insertion-sort runs in $O(n^2)$ time.
What is a heap?

A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:

- **Heap-Order:**
  - for every internal node \( v \) other than the root, \( \text{key}(v) \geq \text{key(parent}(v)) \)

- **Complete Binary Tree:**
  - let \( h \) be the height of the heap
  - for \( i = 0, \ldots, h - 1 \), there are \( 2^i \) nodes of depth \( i \)
  - at depth \( h - 1 \), the internal nodes are to the left of the external nodes

- The last node of a heap is the rightmost internal node of depth \( h - 1 \)

![Heap Example](image)
Height of a Heap

**Theorem:** A heap storing $n$ keys has height $O(\log n)$

**Proof:** (we apply the complete binary tree property)

- Let $h$ be the height of a heap storing $n$ keys
- Since there are $2^i$ keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
- Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$
Heaps and Priority Queues

- We can use a heap to implement a priority queue.
- We store a (key, element) item at each internal node.
- We keep track of the position of the last node.
- For simplicity, we show only the keys in the pictures.

The heap is constructed such that the keys are ordered: 2 < 5 < 6 < 7 < 9. The priority queue contains the elements (2, Sue), (5, Pat), (6, Mark), (7, Anna), and (9, Jeff).
Insertion into a Heap

- Method `insertItem` of the priority queue ADT corresponds to the insertion of a key $k$ to the heap.
- The insertion algorithm consists of three steps:
  - Find the insertion node $z$ (the new last node).
  - Store $k$ at $z$ and expand $z$ into an internal node.
  - Restore the heap-order property (discussed next).
Upheap

- After the insertion of a new key $k$, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping $k$ along an upward path from the insertion node.
- Upheap terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$.

Performance
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time.
Removal from a Heap

- Method `removeMin` of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps:
  - Replace the root key with the key of the last node $w$.
  - Compress $w$ and its children into a leaf.
  - Restore the heap-order property (discussed next).
Downheap

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated.
- Algorithm downheap restores the heap-order property by swapping key $k$ along a downward path from the root.
- Upheap terminates when key $k$ reaches a leaf or a node whose children have keys greater than or equal to $k$.

Performance
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time.
Example
Example
Example
Example
Example

Heaps and Priority Queues
Example
Accessing the Queue

- In a regular queue, you can explicitly keep
  - the head-index and tail-index, or
  - the head-index and the size

- In a priority queue, you can explicitly keep
  - the head-pointer (root) and the tail-pointer (last node), or
  - the head-pointer and the size
Question:

How do you update the last node ("tail") pointer or get it from the queue size?

Two answers...
1. Updating Last Node Pointer

The insertion node can be found by traversing a path:
- Go up until a left child or the root is reached
- If a left child is reached, go to the right child
- Go down left until a leaf is reached

Similar algorithm for updating the last node after a removal

Performance:
- $O(\log n)$
Finding Last Node Pointer

The insertion node can be found by traversing a path without needing an explicit tail pointer:

- Start at the root and using the binary number equivalent of the new number of nodes
  - Assume the root to be the right-child of an imaginary parent
  - Starting with MSB, traverse using 0=left and 1=right
- Prevents the need to keep a last node pointer around
- Asymptotically same performance, but half the cost

Similar algorithm for updating the last node after a removal

Performance:
- $O(\log n)$
Examples
Heap-Sort

- Consider a priority queue with $n$ items implemented by means of a heap
  - the space used is $O(n)$
  - methods `insertItem` and `removeMin` take time $O(\log n)$
  - methods `size`, `isEmpty`, `minKey`, and `minElement` take time $O(1)$

- Using a heap-based priority queue, we can sort a sequence of $n$ elements in time $O(n \log n)$

- The resulting algorithm is called heap-sort

- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort
Heap-Sort

More explicitly, how much time does it take to construct a heap?

- \( n \) items, each requiring up to \( \log n \) swaps during “up-heap” operations
  - \( O(n \log n) \)

How much time does it take to “destruct” a heap (or remove items in sorted order)?

- \( n \) items, each requiring up to \( \log n \) swaps during “down-heap” operations
  - \( O(n \log n) \)

Thus Heap-Sort is

- \( n \log n + n \log n = O(n \log n) \)
Heap Construction

Can you do better than $O(n \log n)$?

How?

Why do we care?

- We only want to find the few smallest keys among many items
- We want to quickly start “using the items” in sorted order but the sorting can continue while I start using the first items, e.g.: real-time OS, games, simulations, etc.
First: Merging Two Heaps

- We are given two heaps and a key $k$.
- We create a new heap with the root node storing $k$ and with the two heaps as subtrees.
- We perform downheap to restore the heap-order property.
Then: Bottom-up Heap Construction

We can construct a heap storing \( n \) given keys using a bottom-up construction with \( \log n \) phases.

In phase \( i \), pairs of heaps with \( 2^i - 1 \) keys are merged into heaps with \( 2^{i+1} - 1 \) keys.
Example

Goal: to create a heap of $N$ elements

Assume $N = 2^H - 1$ for some integer $H$ and thus the heap (tree) is “full”

In a first step, we construct $(N+1)/2$ basic heap structures

- One key and two empty children pointers
Example
Example (contd.)

Heaps and Priority Queues
Example (contd.)

Heaps and Priority Queues
Example (end)
Analysis: What is the performance?

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path).
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$.
  - Or, similarly, each edge of the tree is visited once and since the total number of edges is $(2n-1)$, then $O(n)$.
- Thus, bottom-up heap construction runs in $O(n)$ time.
  - Bottom-up heap construction is faster than $n$ successive insertions and speeds up the first phase of heap-sort.
Analysis: What is the performance?

Thus, we can start using the first results of sorting after $O(n)$ time and using $O(n)$ space.

- Groovy!
Analysis: Why is this important again?

Consider the Internet

- You have $N=10^9$ pages you want to sort and know the top results as soon as possible
- Waiting $O(N^2) = 10^{18}$ before knowing the top results
  - is too long...
  - Even if you choose a very small time unit, e.g.:
    - you assume a 1-GigaHz computer to do 1-Giga operations per second, you will take $10^9$ seconds, or 31 years!
- Waiting $O(N\log N) = 30 \times 10^9$
  - is doable, maybe it means 30 seconds
- Waiting $O(N) = 10^9$
  - is more doable, maybe meaning 1 second!!!
Vector-based Heap Implementation

We can represent a heap with $n$ keys by means of a vector of length $n + 1$

For the node at rank $i$
- the left child is at rank $2i$
- the right child is at rank $2i + 1$

Links between nodes are not explicitly stored

The leaves are not represented

Operation insertItem corresponds to inserting at rank $n + 1$

Operation removeMin corresponds to removing at rank $n$

**Yields in-place heap-sort!**
Let's look at this again

Consider the Internet

- We looked at sorting the pages
  - $O(N^2)$, $O(N \log N)$, $O(N)$ (for first keys of the sort and $O(N \log N)$ to complete it)
- How fast can we find any particular page we want in an initially unsorted set?
  - $O(N^2)$?
  - $O(N \log N)$?
  - $O(N)$?
  - $O(1)$? It is possible! (kinda)
Coming next!