

Priority Queue

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

©Hans de Nivelles, 2025

A **priority queue** is multiset-like data structure that supports the following operations:

- Insertion of an element.
- Selection of a maximal element under a given order.
- Removal of a maximal element under a given order.

(A multiset is a set that is able to distinguish how often an element occurs in it.)

Priority Queue

```
#include <queue>
std::priority_queue<int> q;
    // No order is specified. This means that
    // std::less<T> will be used, which is usually
    // equal to <.

q. push(4);
q. push(5);
q. push(2);
q. push(4);
while( q. size( ))
{
    std::cout << q. top( ) << "\n";
    q. pop( );
}
```

Priority Queue (2)

Adding and removing can be freely mixed:

```
q. push(4);  
q. push(3);  
q. pop( ); // Removes 4.  
q. push(5); // 5 will be on front.  
q. pop( ); // Removes 5.  
q. push(2); // Leaves 3 on front.
```

Some Theory: What is an Order?

A **total order on** D is a binary relation on D that completely defines how a set should be sorted. For all x, y, z in D , we have the following:

irreflexivity: Not $x < x$.

transitivity: If $x < y$, and $y < z$, then $x < z$.

totality: Either $x < y$, $y < x$ or $x = y$.

Think of $<$ on the natural numbers, or the reals. Another example is $<$ on strings.

Orders are essential for computer science. They are used for look up (search trees), for sorting, for expressing preference (priority queue).

Weak Orders

In many cases, not all elements need to be comparable.

Different solutions:

1. Define a function f to some set D which has a total order, and define $x \prec y \Leftrightarrow f(x) < f(y)$.

2. Redefine the conditions on $<$, e.g. \preceq is **weak order on** a set X if for all $x, y, z \in X$,

reflexive: $x \preceq x$.

transitive: If $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

total: Either $x \preceq y$, or $y \preceq x$.

The different approaches are equivalent, one can also try to define the conditions on \prec directly. I think **(1)** is the easiest way.

Equivalence of the Two Definitions

If you prefer **(1)**, you can define $x \preceq y$ if $f(x) < f(y)$ or $f(x) = f(y)$ to get **(2)**.

If you like **(2)** more, you have to define the **equivalence classes** first:

In set X , define $x \equiv y$ if $x \preceq y$ and $y \preceq x$.

Define f as the function that maps x to its **equivalence class**:

$$f(x) = \{x' \in X \mid x \equiv x'\}.$$

(We did not define equivalence classes, but it is every set that can occur as $f(x)$ for an $x \in X$.)

Let D be the set of equivalence classes, and define $E_1 < E_2$ if for all $e_1 \in E_1$ and $e_2 \in E_2$, we have $e_1 \preceq e_2$ and $E_1 \neq E_2$.

It is surprising how such an intuitive concept can induce so much technicality. (and there is a third definition).

Examples of weak orders:

X is set of pairs of integers. Take $(m_1, n_1) \preceq (m_2, n_2)$ if $\frac{m_1}{n_1} \leq \frac{m_2}{n_2}$.

Or take $(m_1, n_1) \preceq (m_2, n_2)$ if $n_1 \leq n_2$.

Implementation: Heap

`priority_queue` uses a data structure called **heap**. It is a vector that is sorted just enough to know the maximal element.

Assume that N is the size of vector v :

- If $2i + 1 < N$, then $v[2i + 1] \preceq v[i]$.
- If $2i + 2 < N$, then $v[2i + 2] \preceq v[i]$.

Adding to a heap, and removing the top element (while preserving heap structure) can be done in $O(\log N)$.

Providing the Order

Priority queue has definition

```
template< class T, class C = vector< T >,
          class Cmp = less< C :: value_type >>
```

- T is the type variable.
- C is the container type that the priority queue uses to store its elements. It is `std::vector<T>` by default.
- Cmp specifies the order. It works in the same way as with `std::map< >`. The default is `less< C :: value_type >`. Since `less< C :: value_type >` is usually T, the default is `less<T>`, which is by default `<` on T.

Providing the Order (2)

There are two things to observe:

1. If you want to provide an order, you have to provide a container. Just use `std::vector<T>`.
2. As usual, `Cmp` is a type, that must have a default constructor and a method
`bool operator() (const T& t1, const T& t2) const.`
This method must return `true` if `t2` is more preferred than `t1`.
If you forget to make `operator()` **const**, you will see horrible error messages.

Write

```
struct Cmp
{
    bool operator( ) ( int i1, int i2 ) const
    {
        if( i1 < 0 ) i1 = - i1;
        if( i2 < 0 ) i2 = - i2;
        return i1 < i2;
    }
};
```

if you want to compare **int** by absolute value, instead of value.

Non-Total Order

When the order is not total, (does not always decide a priority between all elements of T), function `top()` `const` will non-deterministically pick an element from the best.

‘Non-deterministically’ means that one should not try to understand which element is selected. Your program should be written in such a way that this doesn’t matter. Non-determinism is an essential aspect of high-level programming.

`pop()` is guaranteed to delete the element returned by `top()`.

When to use `<` or a comparator?

Don't create an order `<` on a type `T`, when there is no natural choice that will be evident to readers of your code.

If you define a dedicated class `struct` or `class` for the priority queue, you can name the class in such a way that `<` is the natural choice.

Using Priority Queue and Map in Search

Let $\mathcal{G} = (V, E)$ be a directed graph. For simplicity, assume that all edges $(v_1, v_2) \in E$ have equal weight.

We want to find a path from v_s to v_e in \mathcal{G} .

Searching the Path to an Element

Let F be a partial function from V to \mathcal{N} , denoting the length of the best known path to V , if we have found one.

Let $U \subseteq \text{Dom}(F)$ be the set of nodes whose neighbours have not been checked.

Start with:

```
f[vs] = 0;           // A map. f[v] is distance to v.  
u. push( vs );      // A priority queue, nodes whose  
                    // neighbours were not checked.
```

```

while( !u.isempty( ) && f[ve] is undefined )
{
    v = u. top( ); u. pop( );
    // v is most promising unchecked vertex.
    for every direct neighbour v' of v do
    {
        if( f[v'] is undefined or f[v'] > f[v] + 1 )
        {
            f[v'] = f[v] + 1;
            u. push( v' );
        }
    }
}
if( f[ve] is defined ) std::cout << "found a solution";
else std::cout << "found no solution";

```

Printing the Solution

The easiest way to print the solution is by backward recursion. We know that v_e is reachable in $n = f[v_e]$ steps. This means that there must exist a solution with a path in which one of the neighbours of v_e is reachable in $< n$ steps.

```
printpath( f, v, n )
if( n > 0 )
{
    find a neighbour v' of v with f[v'] < n.
    // There is guaranteed to exist one.

    printsolution( f, v', f[v'] );
    print v and the vertex ( v', v );
}
```