**Priority queue**

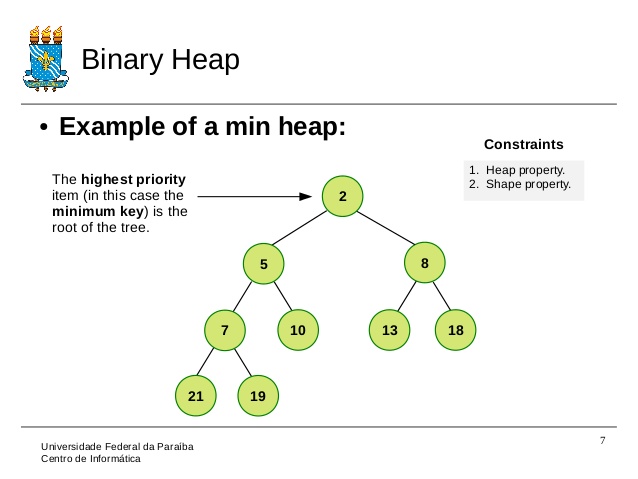
While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map" - just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.

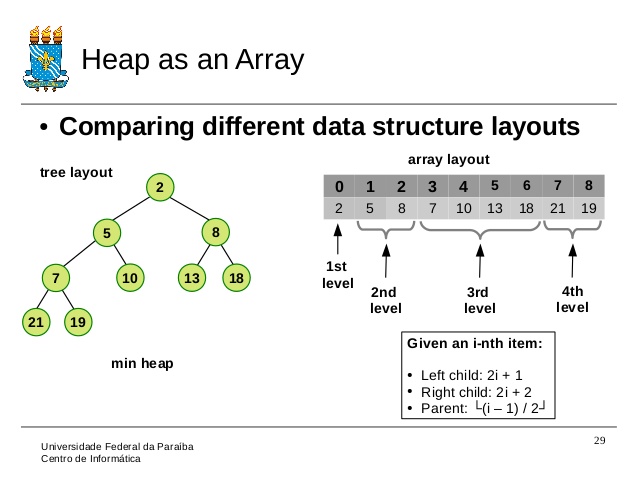
Because most priority queue implementations use heap, we will focus on this data structure.

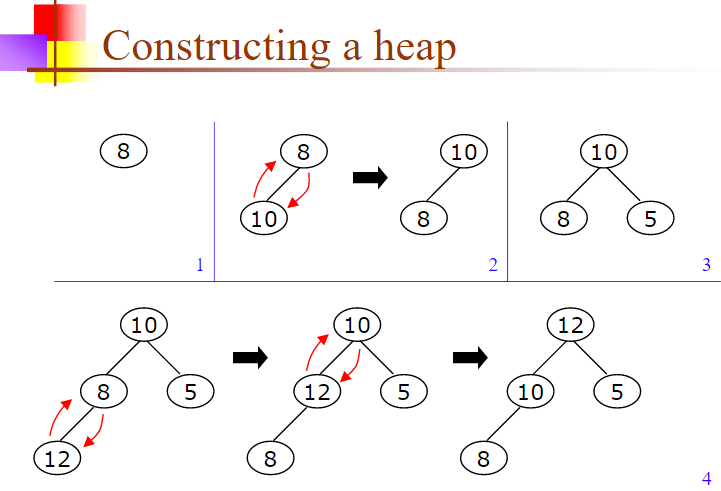
**Heap**

In a heap, for every node *i*other than the root, the value of a node is greater \ smaller than or equal (at most) to the value of its parent.  
Thus, the largest \ smallest element in a heap is stored at the root. This is the most important property of a heap and priority queue.

Heaps are more efficient than binary trees if you only need the extreme element. Heaps only do the comparisons (lazily) necessary to determine the extreme element.



Heaps are usually implemented in an array (fixed size or dynamic array):



Difference of priority\_queue vs queue:

priority queues are based on priority instead of a pure FIFO behavior.

In C++:

A std::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.

Working with a std::priority\_queue is similar to managing a heap in some random access

Container (like std::vector), with the benefit of not being able to accidentally invalidate the heap and improved readability of intent.

Alternative of using std::priority\_queue would be to use std::vector \ std::deque and manage heap manually by functions:

std::make\_heap, std::push\_heap, std::pop\_heap, std::sort\_heap.

Advanced: example implementation of std::priority\_queue:

template <class T, class Container = std::vector<T>,

class Compare = std::less<T> >

class priority\_queue

{

protected:

Container c;

Compare comp;

public:

explicit priority\_queue(const Container& c\_ = Container(),

const Compare& comp\_ = Compare())

: c(c\_), comp(comp\_)

{

std::make\_heap(c.begin(), c.end(), comp);

}

bool empty() const { return c.empty(); }

std::size\_t size() const { return c.size(); }

const T& top() const { return c.front(); }

void push(const T& x)

{

c.push\_back(x);

std::push\_heap(c.begin(), c.end(), comp);

}

void pop()

{

std::pop\_heap(c.begin(), c.end(), comp);

c.pop\_back();

}

};