Graphs

Graphs

For the next 5 weeks we will be talking about data structures and problems related to graphs.

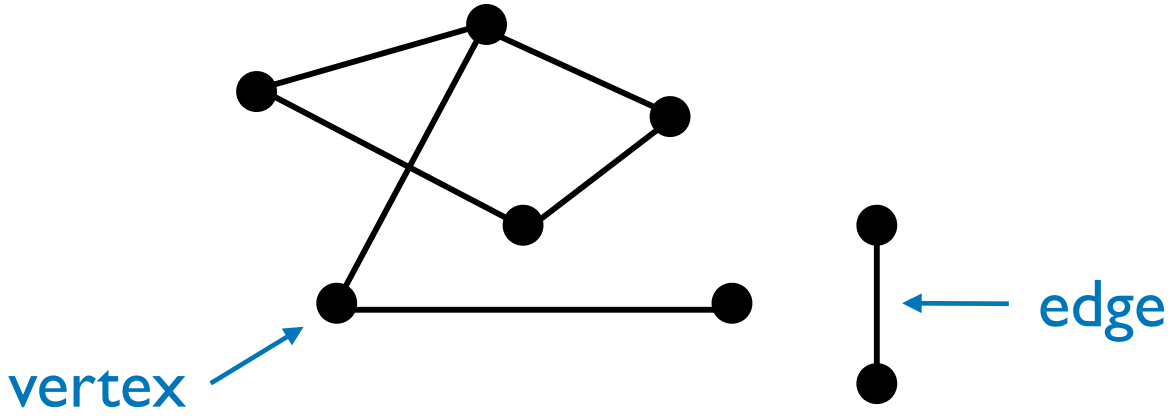
This week and next we talk about data structures that make use of graphs, specifically binary trees.

Then we talk about solving computational problems on graphs themselves.

We begin with an introduction to graphs and trees in particular.

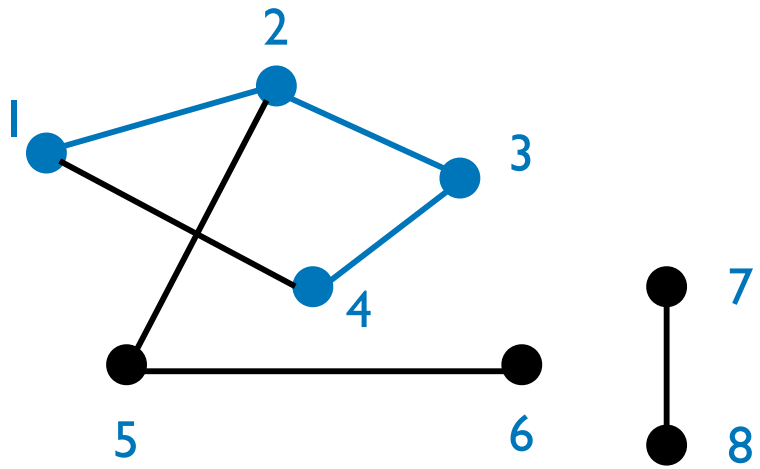
Graphs

A graph consists of vertices and edges connecting pairs of vertices.



Two vertices connected by an edge are called neighbours.

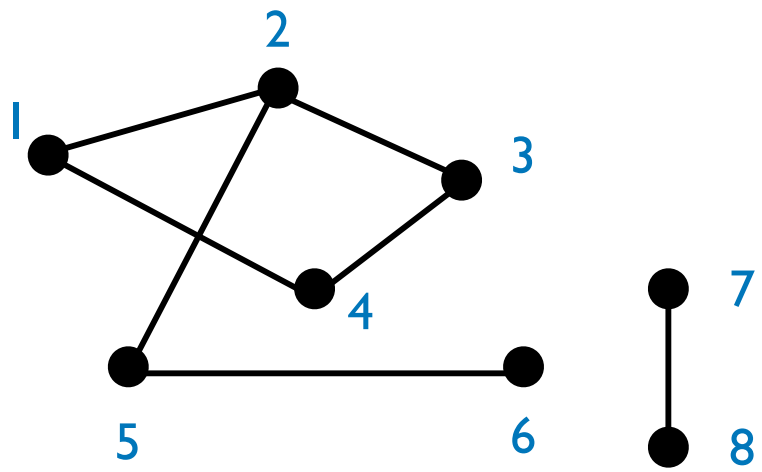
Graphs



A path is a sequence of edges which joins a sequence of vertices.

The blue edges are a path connecting vertex 1 to vertex 4.

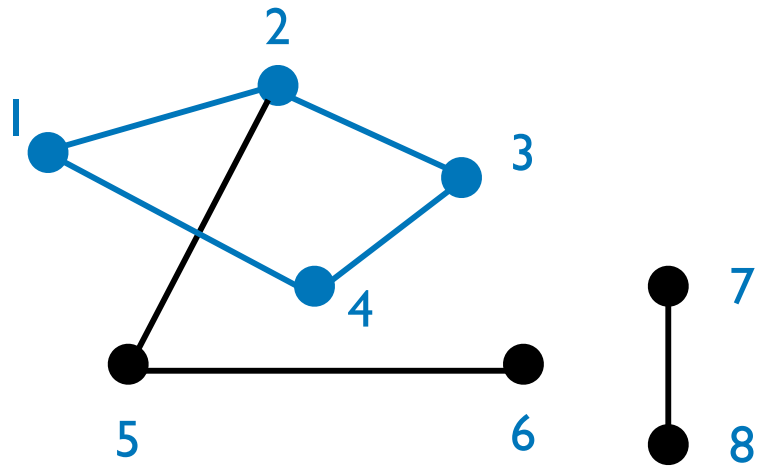
A graph is connected if and only if there is a path between every pair of vertices.



Is this graph connected?

Cycles

A cycle is a path that starts and ends at the same vertex.

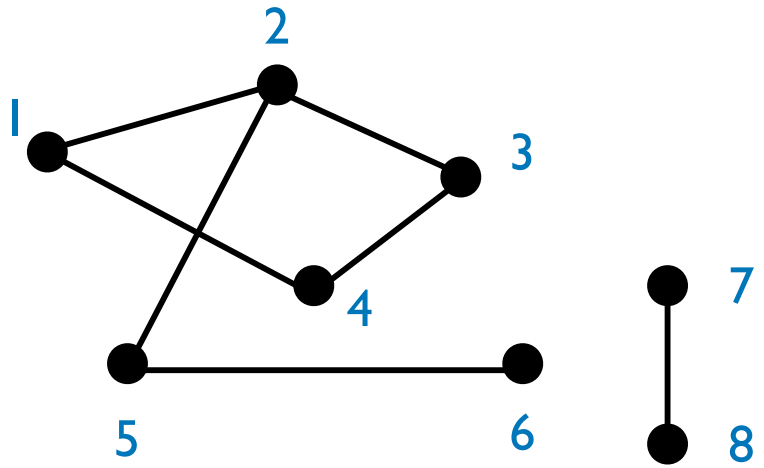


Now the blue edges form a cycle.

Trees

One of the simplest class of graphs are trees.

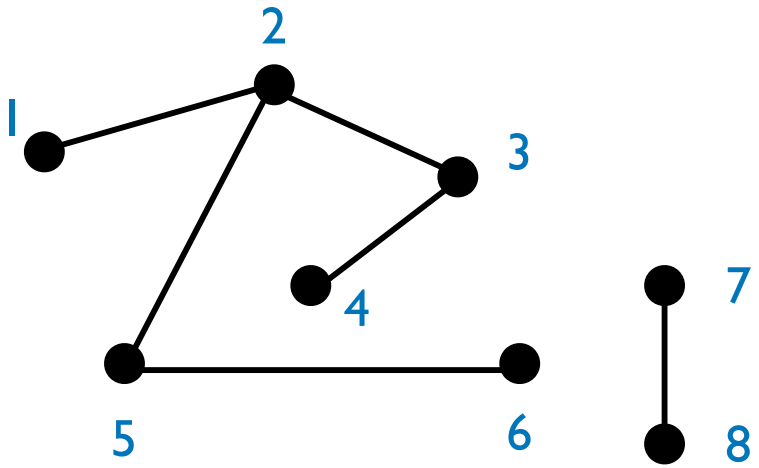
A tree is a connected graph with no cycles.



Is this graph a tree?

Trees

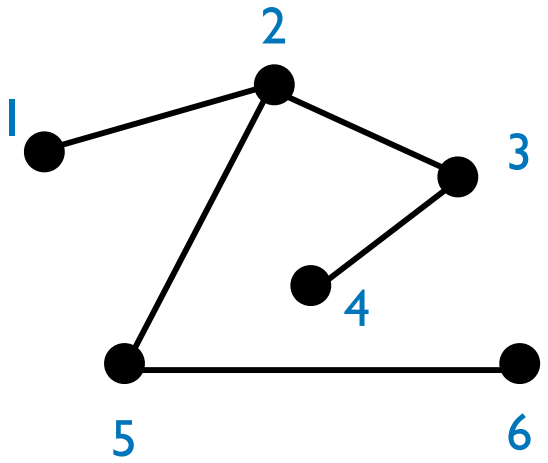
A tree is a connected graph with no cycles.



Is it a tree now?

Trees

A tree is a connected graph with no cycles.



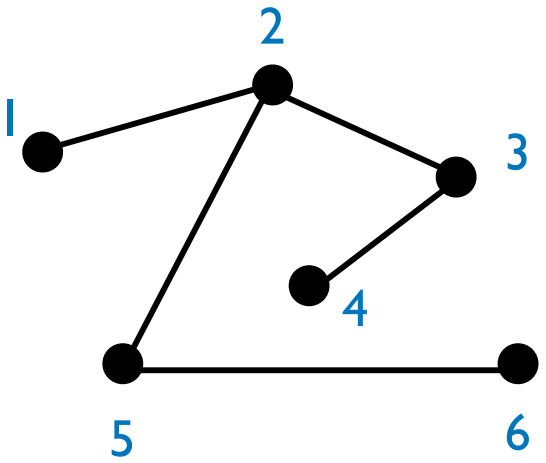
How about now?

Trees

A tree is a connected graph with no cycles.

A tree on n vertices will always have exactly $n-1$ edges.

In fact, a connected graph on n vertices with $n-1$ edges will always be a tree.



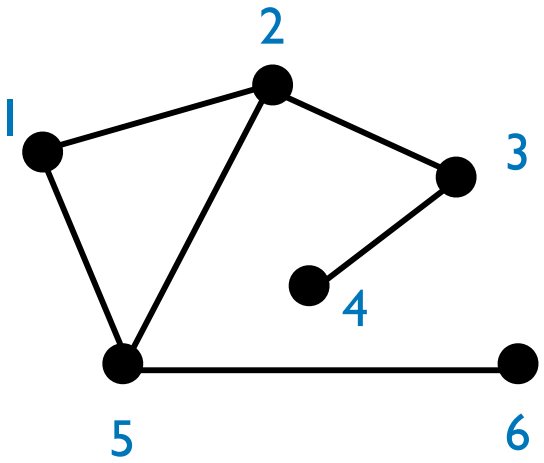
Trees

A tree is a connected graph with no cycles.

In fact, a connected graph on n vertices with $n-1$ edges will always be a tree.

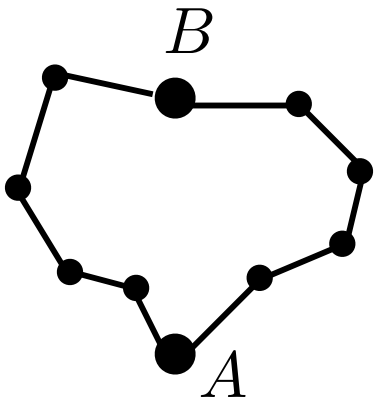
If the graph had a cycle, we could remove an edge of the cycle and the graph would still be connected.

Then it would have $n-2$ edges and be connected, which is impossible.



Trees

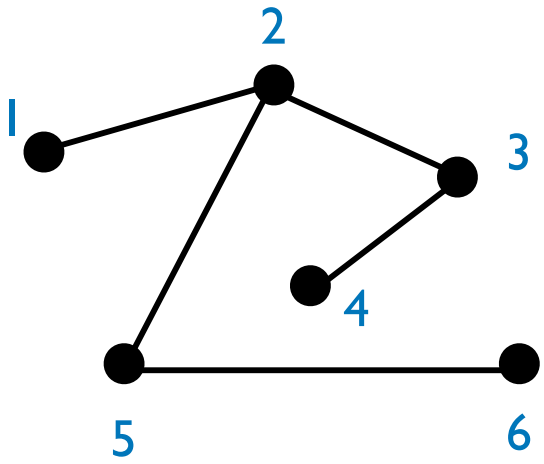
Nice fact: In a tree there is always a unique path (not reusing edges) that connects any two vertices.

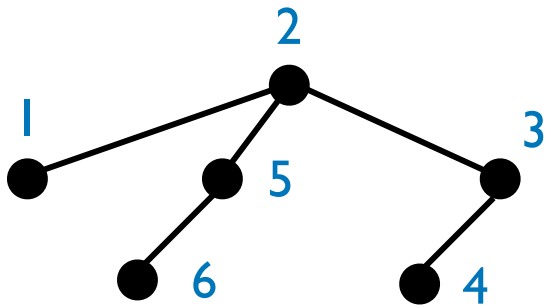


If there were two distinct paths this would create a cycle.

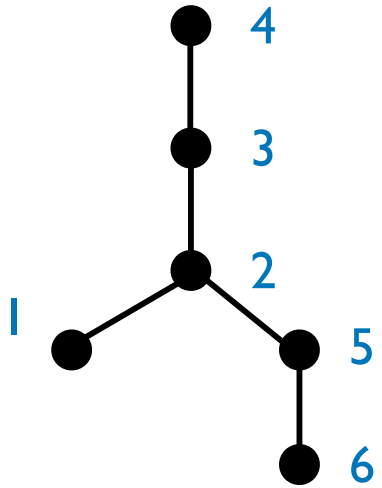
The distance between two vertices in a tree is the length of the path between them.

Rooted trees





making 2 the root



making 4 the root

We often think of a tree as having a root.

We can root a tree at any vertex.

We draw the root at the top, and layer the vertices by distance from the root.

Depth

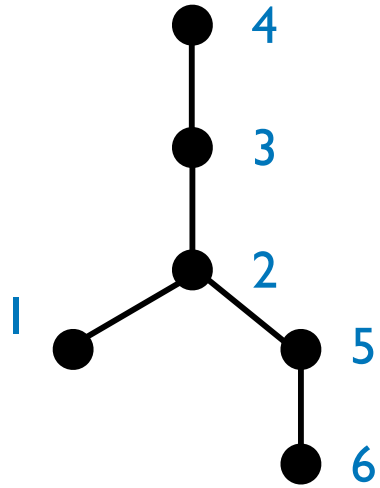
The depth of a vertex is its distance from the root.

The depth of a tree is the maximum depth of a vertex.

Vertex 4 has depth 0.

Vertex 1 has depth 3.

The tree has depth 4.



making 4 the root

Children

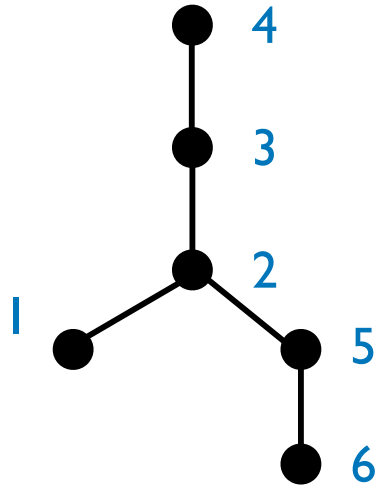
The children of a vertex at depth d are its neighbors at depth $d+1$.

Vertex 3 is the child of vertex 4.

Vertex 1 and 5 are the children of vertex 2.

Vertex 6 has no children.

A vertex with no children is called a leaf.



making 4 the root

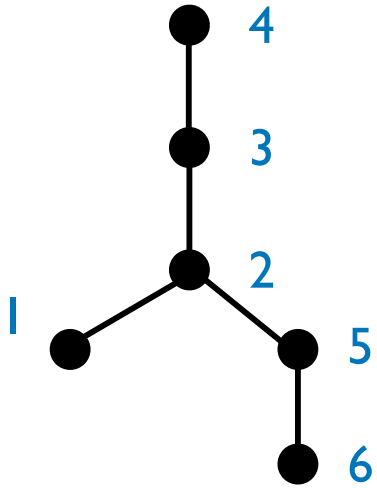
Parents

The parent of a vertex at depth d is its neighbor at depth $d-1$.

A vertex has at most one parent.

The root (vertex 4) has no parent.

The parent of vertex 5 is vertex 2.



making 4 the root

Number of edges

A tree on n vertices will always have exactly $n-1$ edges.

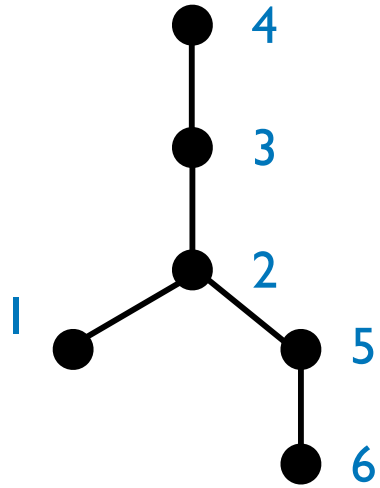
Every vertex except the root can be identified with a unique edge, the edge to its parent.

Height

The height of a vertex is the length of the longest path from the vertex to a leaf.

The height of vertex 3 is 3.

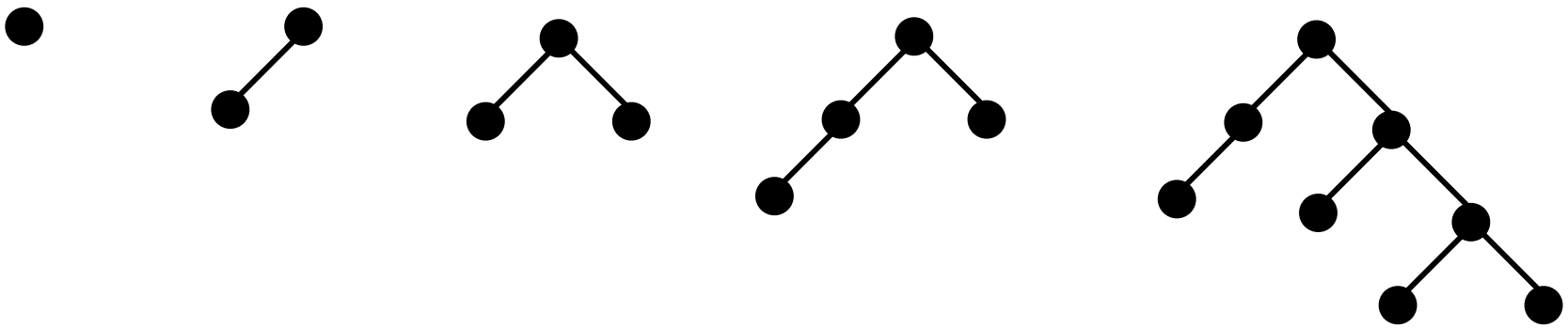
The height of a leaf is 0.



making 4 the root

Binary Tree

A binary tree is a rooted tree where every vertex has at most 2 children.



These are all examples of binary trees.

Priority Queue

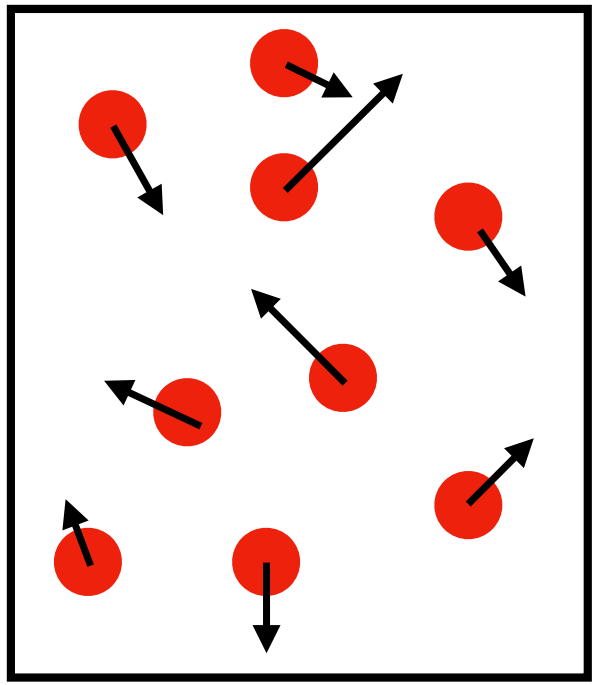
Particle Collision

Suppose we want to simulate the dynamics of n billiard balls.

We assume there is no friction and collisions are elastic.

This is known as the hard disc model in statistical physics.

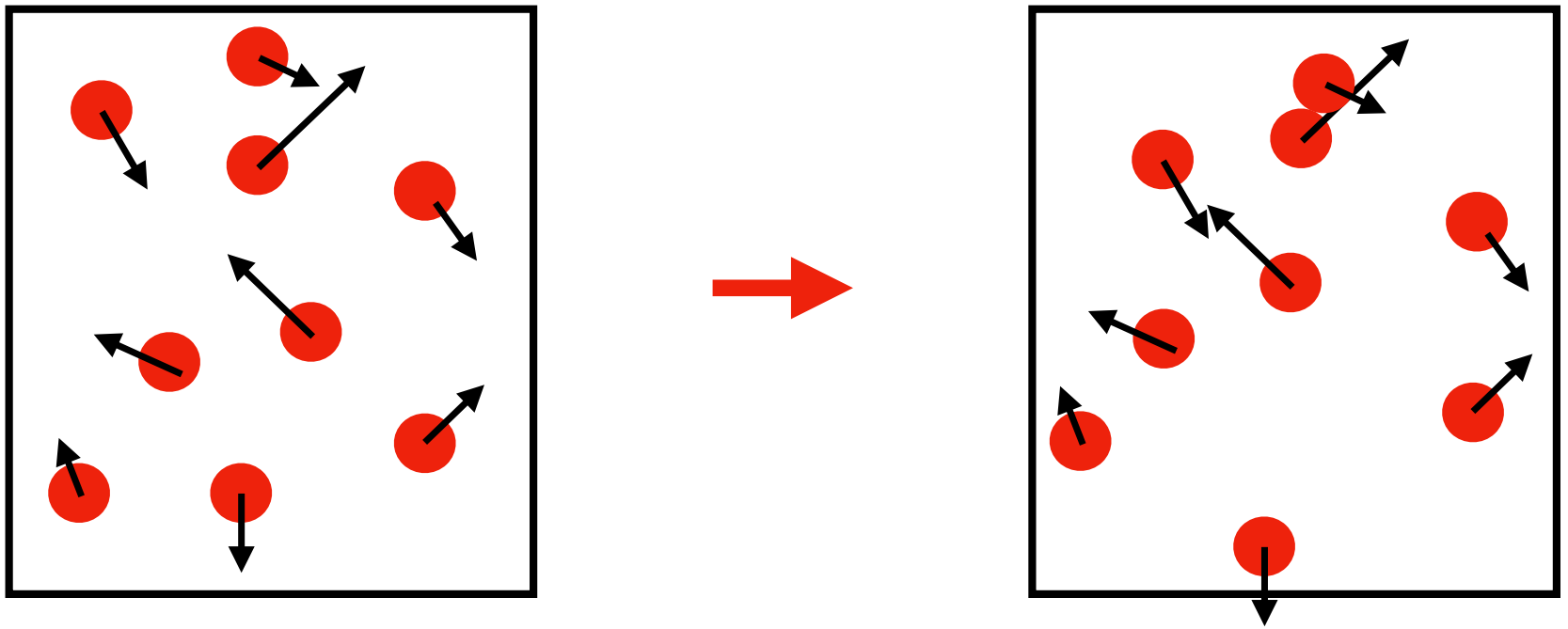
How can we evolve the system computationally?



Event-driven simulation

In between collisions velocities don't change.

We can "fast forward" the system to the next collision.



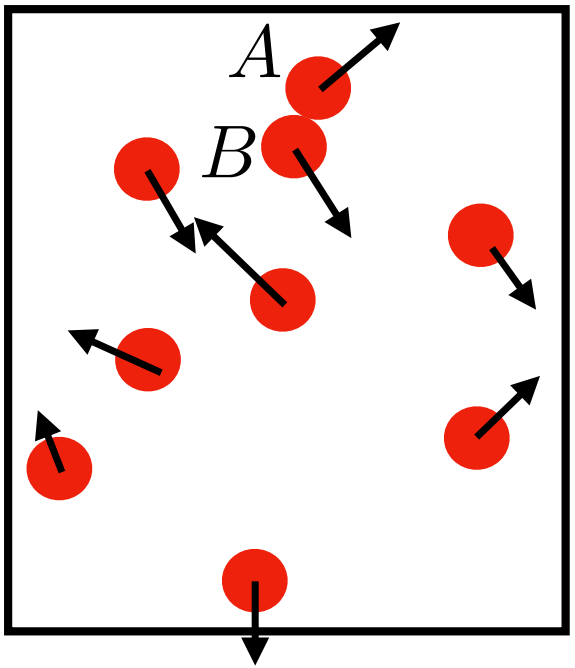
Update with collision

We update the velocities of the balls A, B in the collision.

We then calculate the next collision for balls A, B .

This can be done in $O(n)$ time.

We add these next collision times to our database.



Desired database

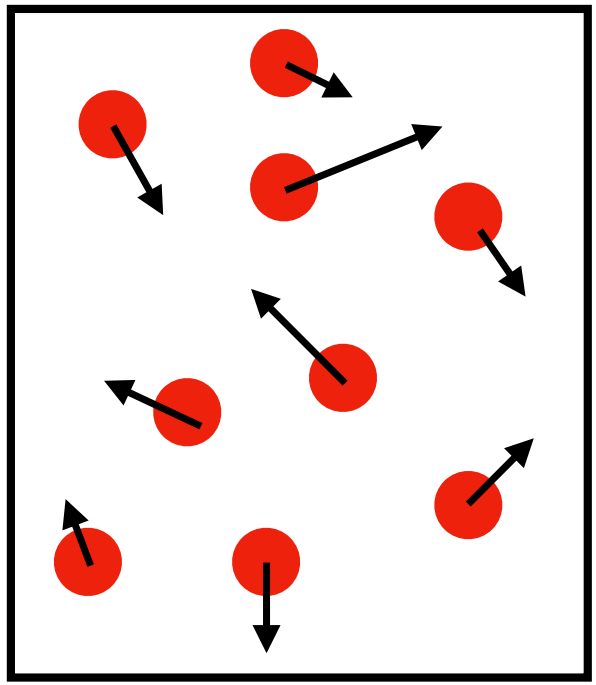
We want a database of collision times.

We want to easily extract the next collision.

We then check that this event is still relevant (the balls' velocities may have changed in the meantime).

If it is relevant, we update velocities.

We then add to the database the next collision time of the balls that just collided.



Abstract Data Type

Let's now extract the features we need for the database.

We want to maintain a (multi)-set of keys, with associated data, and allow operations of:

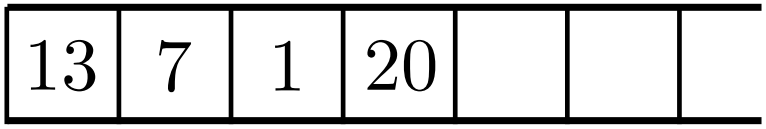
`insert`	put a new key in the set.
`remove_min`	delete the minimum key.
`peek`	look at the minimum key.

This ADT is known as a minimum priority queue.

Unordered Array?

How could we implement a priority queue? Let's go through some data structures we have seen.

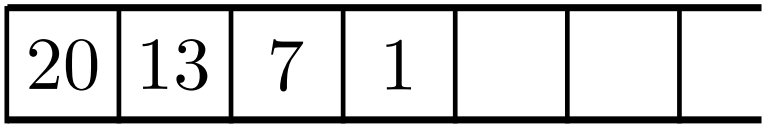
Unordered Array:



Insertion is $O(1)$.

What about finding the minimum element?

Ordered Array?



Let's say we keep the array in reverse order.

Now `peek` and `remove_min` are $O(1)$.

What about `insert`?

Binary Heap

We will now see a way to implement a minimum priority queue using a binary heap.

This uses space $O(n)$ and implements `insert` and `remove_min` in $O(\log n)$ time, and `peek` in $O(1)$ time.*

Can you hope to do `insert` and `remove_min` in $O(1)$ time?

*Assuming we know the number of elements n in advance.

Binary Heap

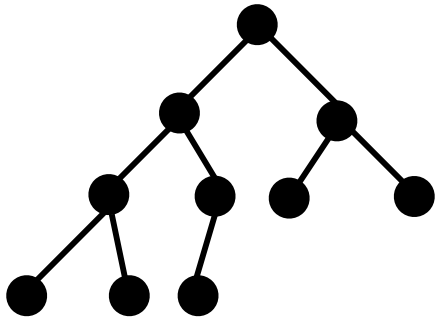
We will now see a way to implement a minimum priority queue using a binary heap.

A binary heap is a binary tree with nodes storing keys and additionally:

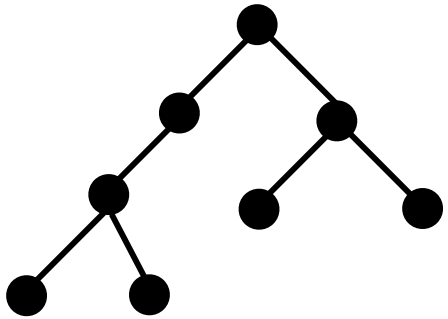
- We maintain that it is a complete binary tree.
- We maintain that it has the heap property.

Complete binary tree

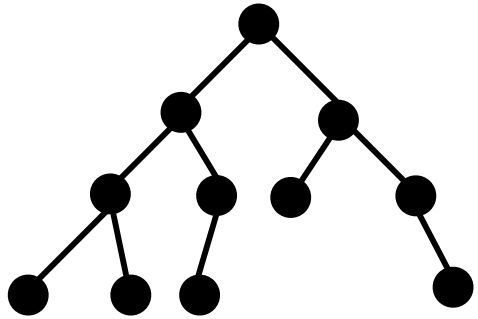
In a complete binary tree, every layer is totally filled except possibly the bottom one, which is filled from the left.



complete



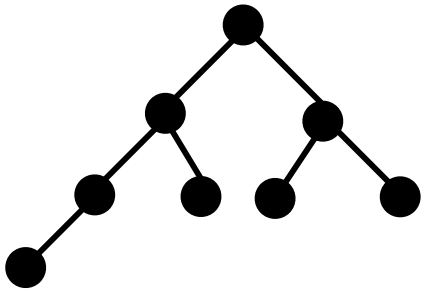
not complete



not complete

Complete binary tree

What is the minimum number of nodes in a binary tree of height h?



The minimum is when there is just a single node on the bottom level.

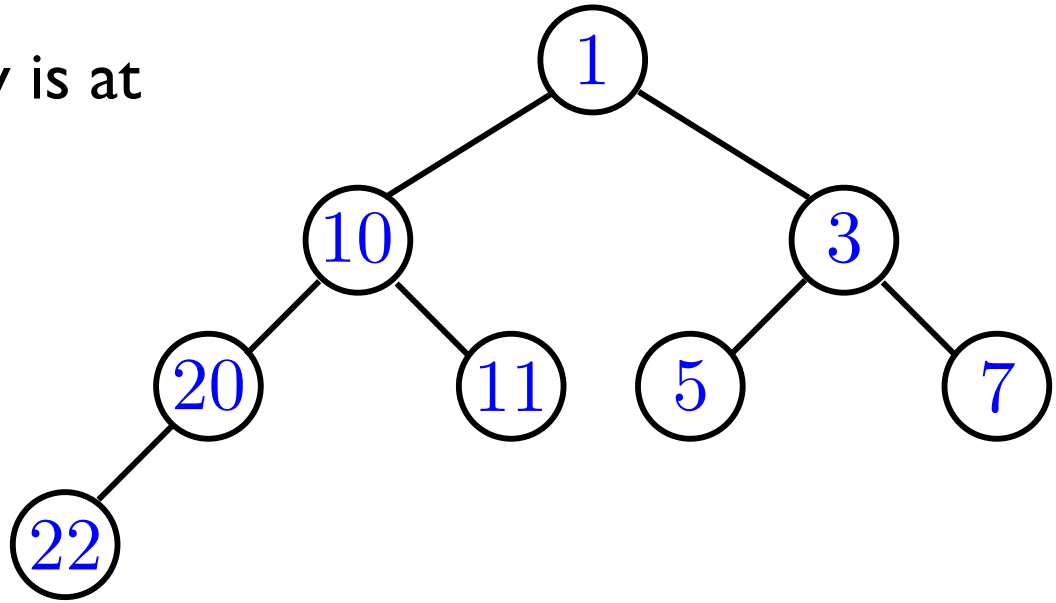
The number of nodes is $1 + \sum_{i=0}^{h-1} 2^i = 2^h$.

A complete binary tree with n nodes has height $\lfloor \log n \rfloor$.

Heap property

Heap property: The key stored at each node is at most the keys of its children.

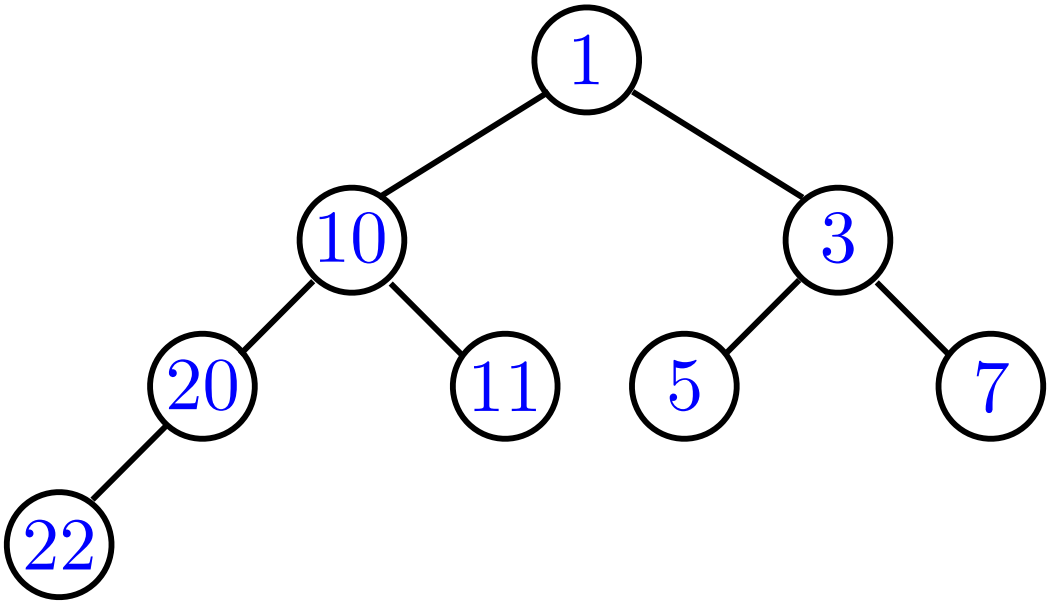
This guarantees that the minimum key is at the root.



Peek

We can immediately see how the peek function works in a binary heap.

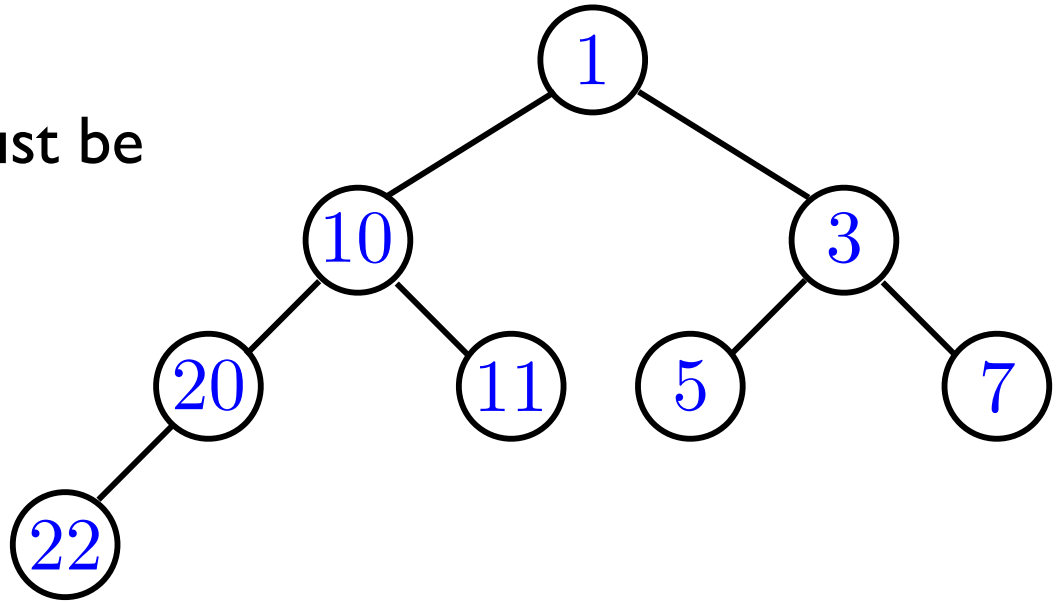
Just observe the root!



3rd smallest element

What can we say about the location of the 3rd smallest element in a min binary heap?

If the depth of a vertex is k there must be at least k keys that are less than or equal to the key at the vertex.



Implementing a binary heap

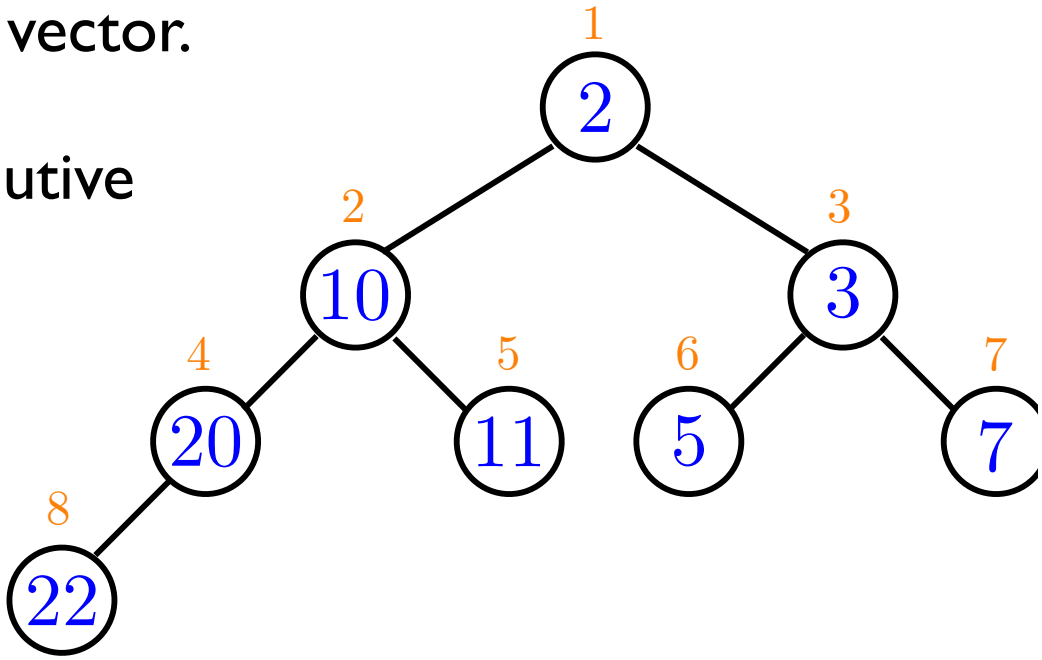
Representing a binary tree

Binary heaps can be implemented with a vector.

We map the nodes of the tree to consecutive integers ordered by levels.

The key of a node is stored at the corresponding index of the vector.

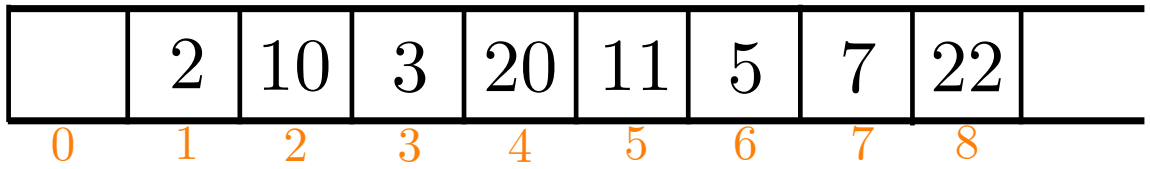
We start from 1 for cleaner indexing.



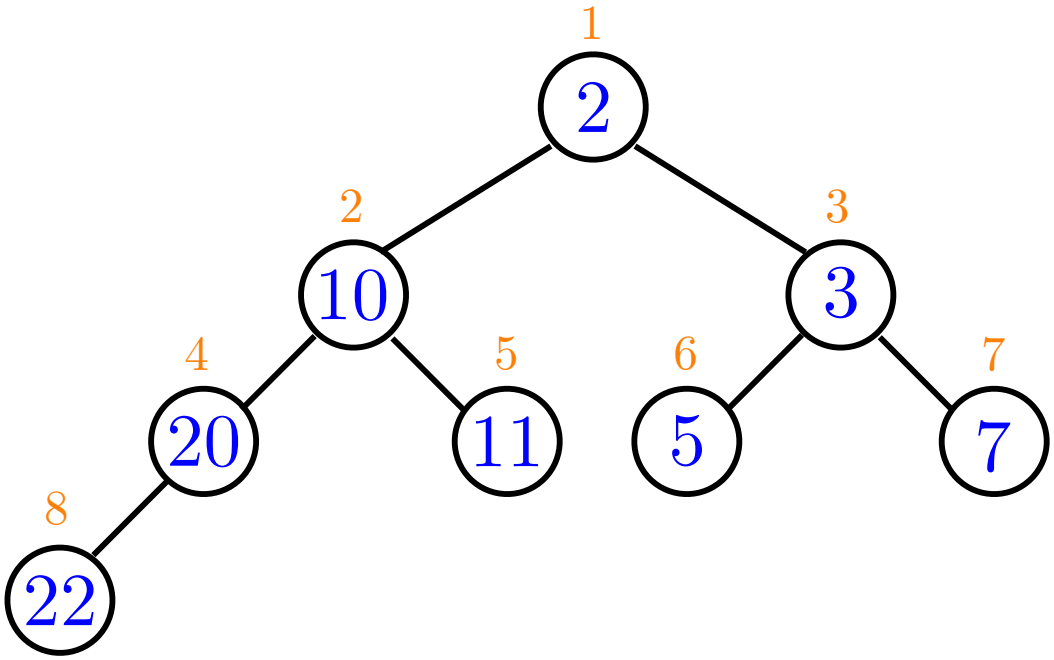
0	1	2	3	4	5	6	7	8	
	2	10	3	20	11	5	7	22	

Heap class

```
class Heap
{
private:
    std::vector<int> vec;
    unsigned num_elements;

public:
    Heap();
    ~Heap();

    void insert(int key);
    void remove_min();
    int peek();
};
```




```
unsigned left(unsigned i){
    return 2*i;
}

unsigned right(unsigned i){
    return 2*i+1;
}

unsigned parent(unsigned i){
    // floor(i/2)
    return i >> 1;
}
```



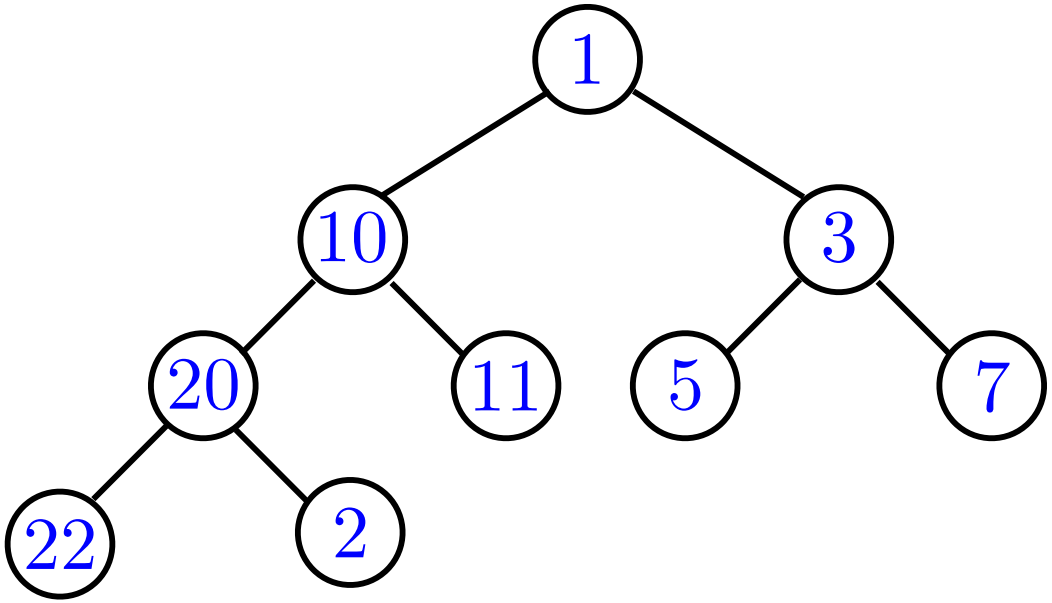
Insert

Now let's see how to insert a key.

Say we want to insert 2.

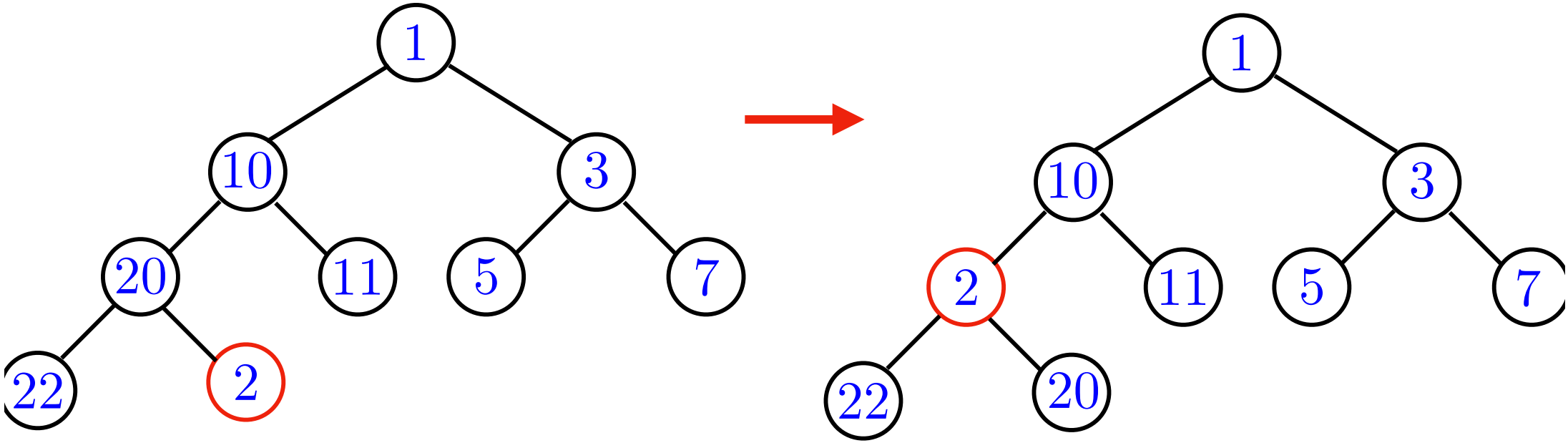
To maintain a complete binary tree, we insert at the leftmost free slot in the bottom row.

This might destroy the heap property.



Swim

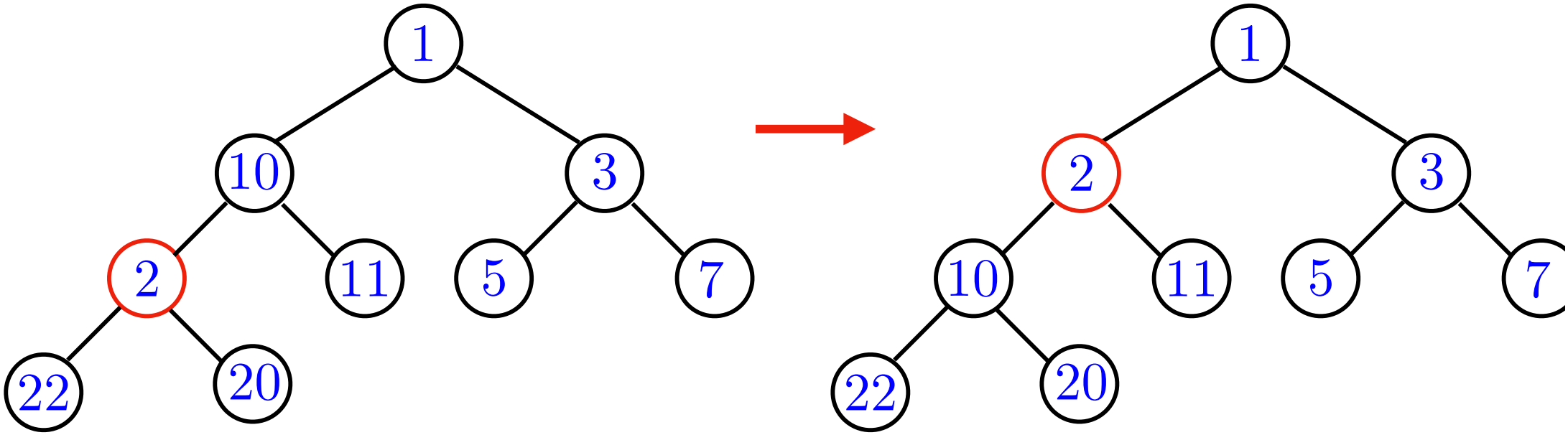
To fix the heap property, we the inserted key "swims" up until it is at least its parent.



Are we done?

Swim

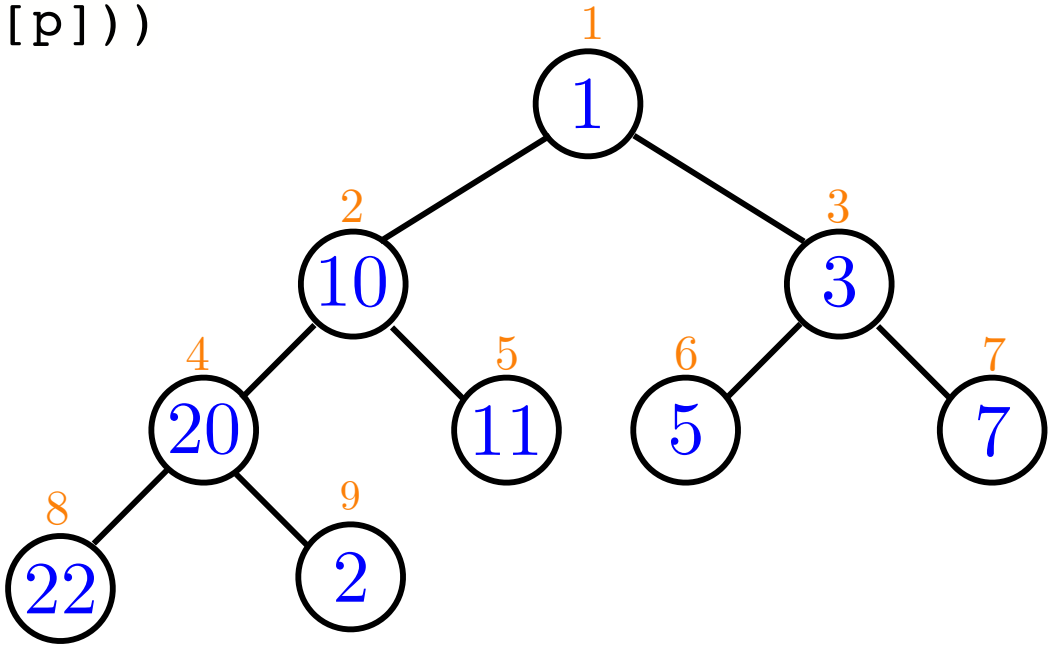
To fix the heap property, we bubble the inserted key up until it is at least its parent.



The heap property is now restored.

Swim: code

```
void swim(unsigned i)
{
    unsigned p = parent(i);
    while((p > 0) && (vec[i] < vec[p]))
    {
        std::swap(vec[i],vec[p]);
        // increment i and p
        i = p;
        p = parent(i);
    }
}
```



https://godbolt.org/z/cnc5P15a5

Insert: complexity

In insert we travel up the tree comparing a node with its parent, and exchanging them if the key of the node is smaller.

The body of the while loop takes constant time.

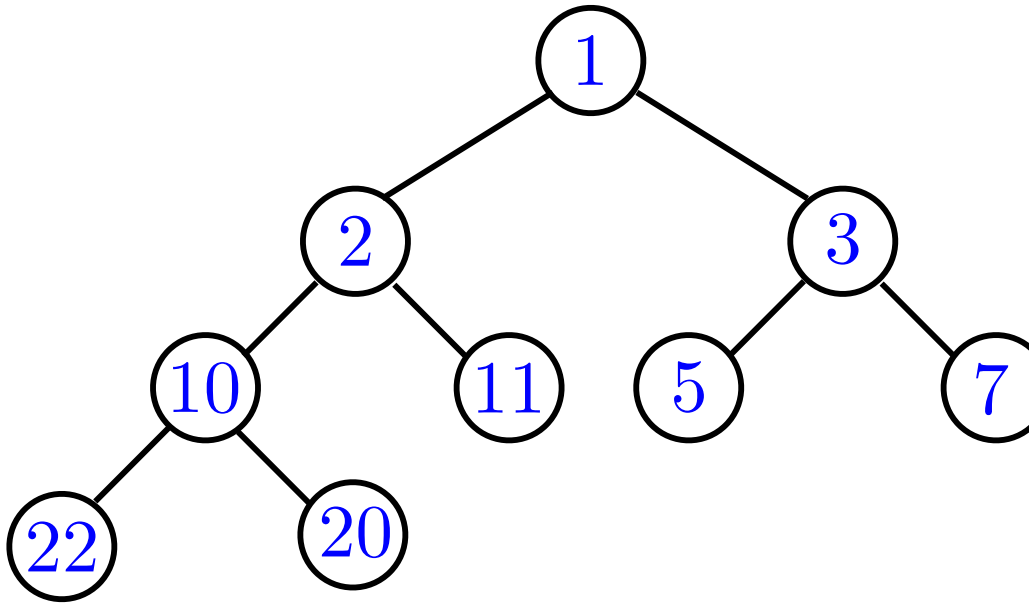
The number of iterations of the while loop is at most the height of the tree.

This is the value of using a complete binary tree: the height is $O(\log n)$.

remove_min

To remove the root from the tree, we replace it with the rightmost element of the bottom row.

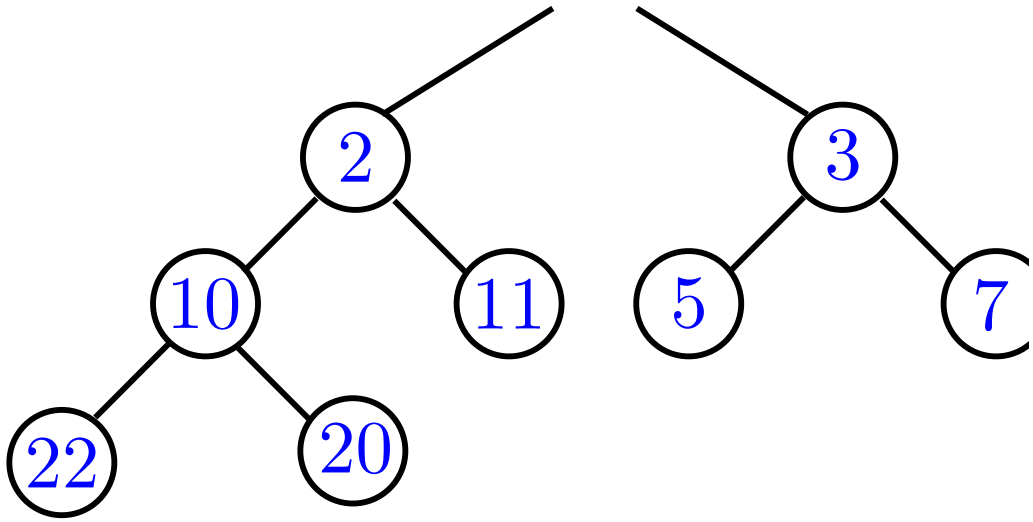
This preserves the complete binary tree property, but again may destroy the heap property.



remove_min

To remove the root from the tree, we replace it with the rightmost element of the bottom row.

This preserves the complete binary tree property, but again may destroy the heap property.

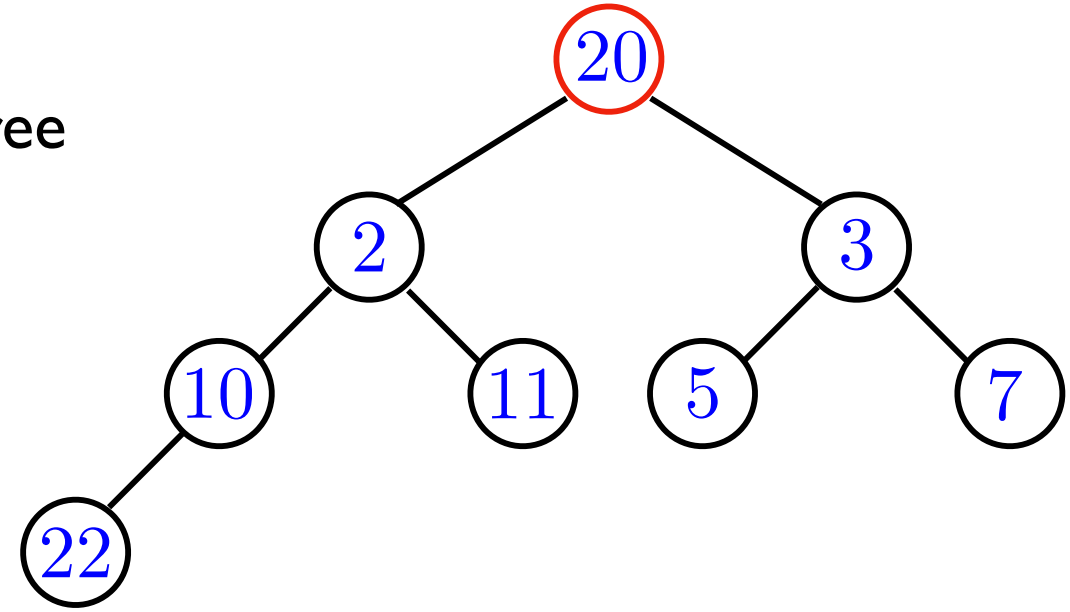


remove_min

To remove the root from the tree, we replace it with the rightmost element of the bottom row.

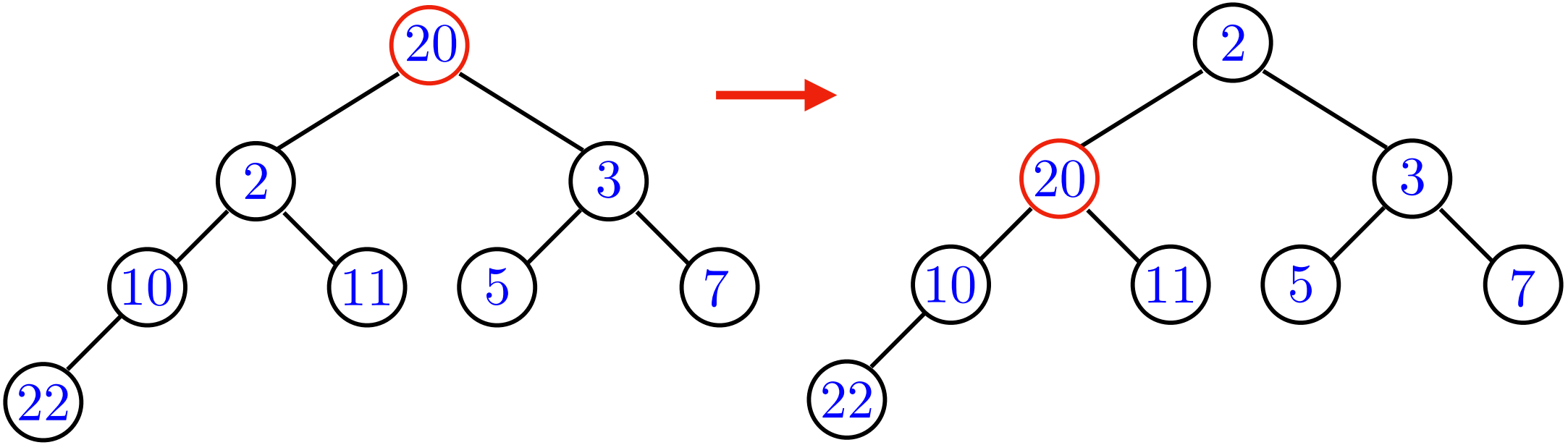
This preserves the complete binary tree property, but again may destroy the heap property.

Now we need to restore the heap property.



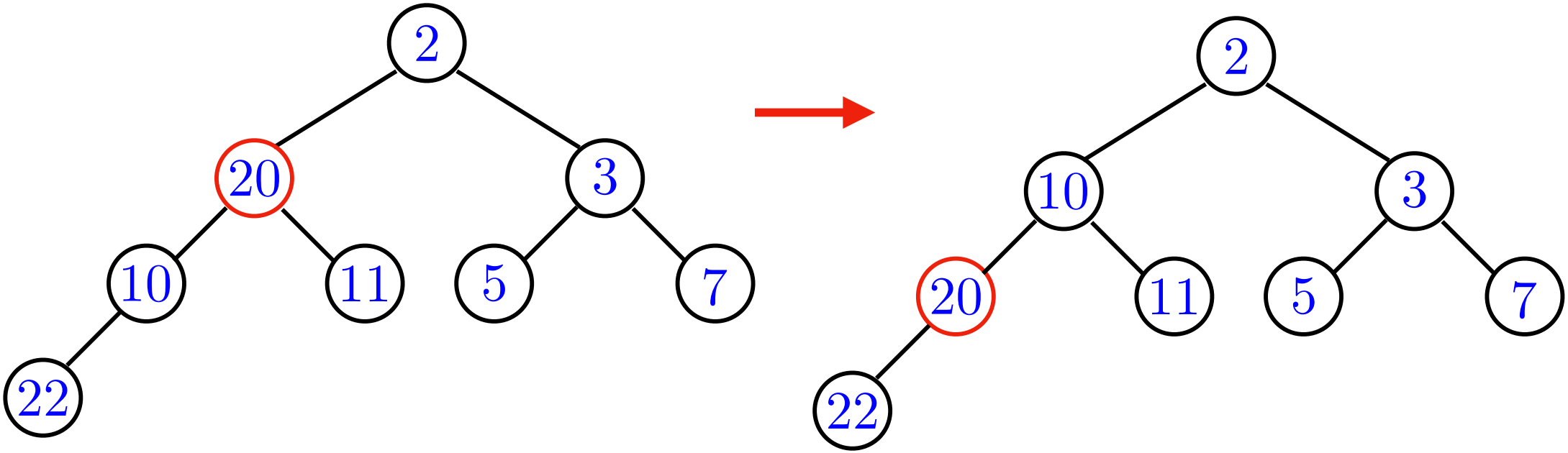
Sink

While the key of the node is bigger than the minimum of the keys of its children, exchange it with the smaller child.



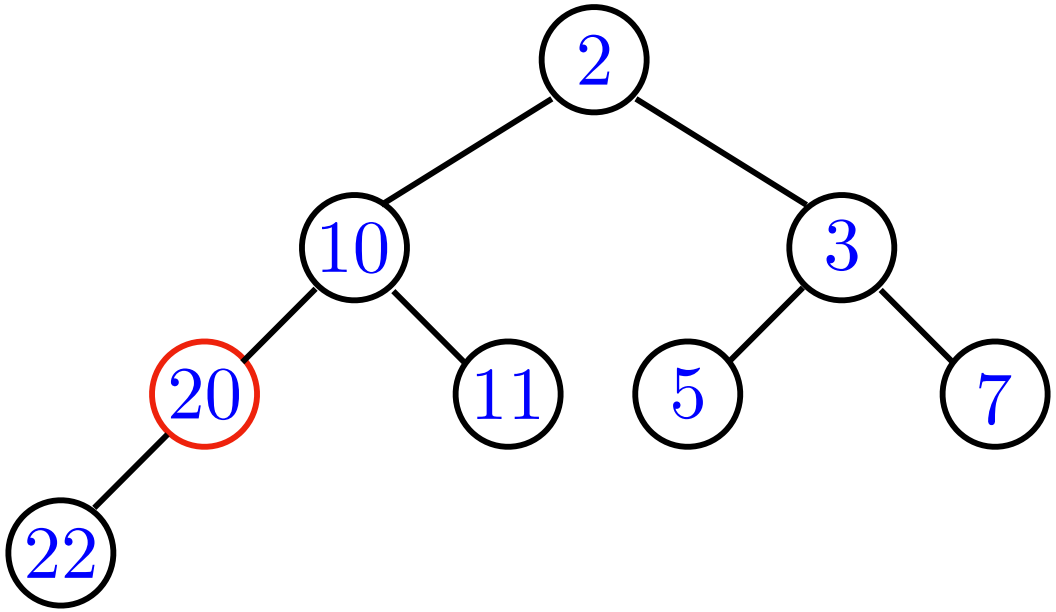
Sink

We keep bubbling down until the key of the node is at most that of its children.



Sink

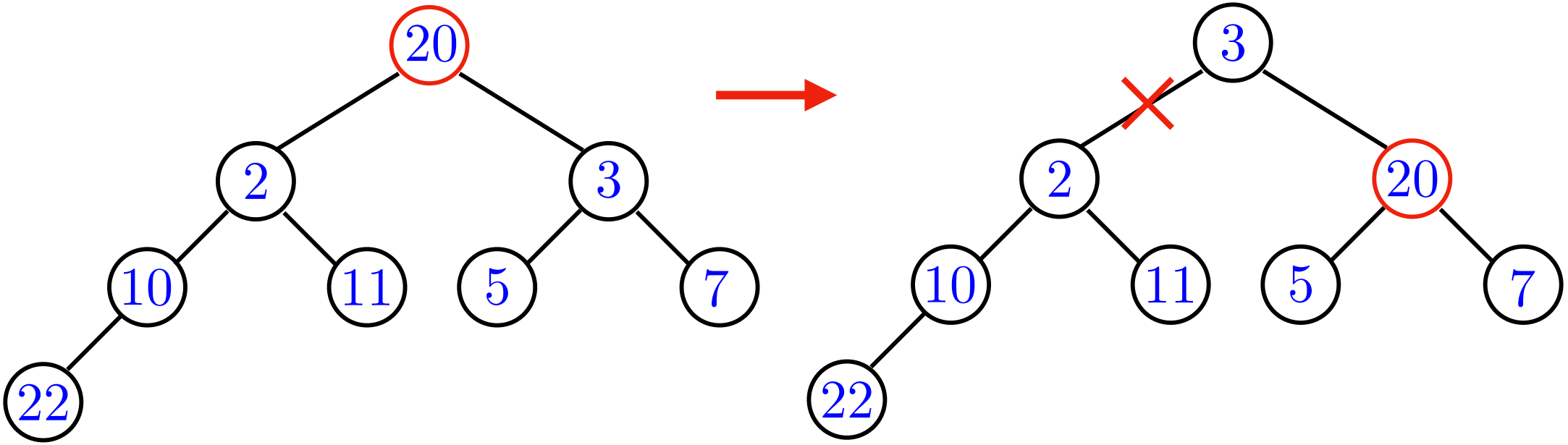
We keep bubbling down until the key of the node is at most that of its children.



Now we have restored the heap property.

Sink

Is it important that we exchange with the smaller child?



remove_min: complexity

The complexity of `remove_min` is similar to that of `insert`.

We do constant work comparing a node to its children and deciding if and with whom to swap.

The number of iterations is at most the height of the tree, which is $O(\log n)$.

Using std::vector

Using std::vector makes inserting into the heap easy with via `push_back`

For our complexity analysis we have to remember that `push_back` is not always a constant time operation.

When the vector reaches its capacity, the next push back is expensive as we have to allocate a new, bigger block of memory and transfer the elements into the new location.

Using std::vector

To do n push back operations still only takes time $O(n)$.

We say that `push_back` has amortised complexity $O(1)$.

This means that we cannot guarantee the insert operation takes $O(\log n)$ time, but will have $O(\log n)$ amortised complexity.

Heap: summary

operation	cost
`insert`	$O(\log n)$ amortised
`remove_min`	$O(\log n)$ worst-case
`peek`	$O(1)$ worst-case