Graphs

Graphs

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

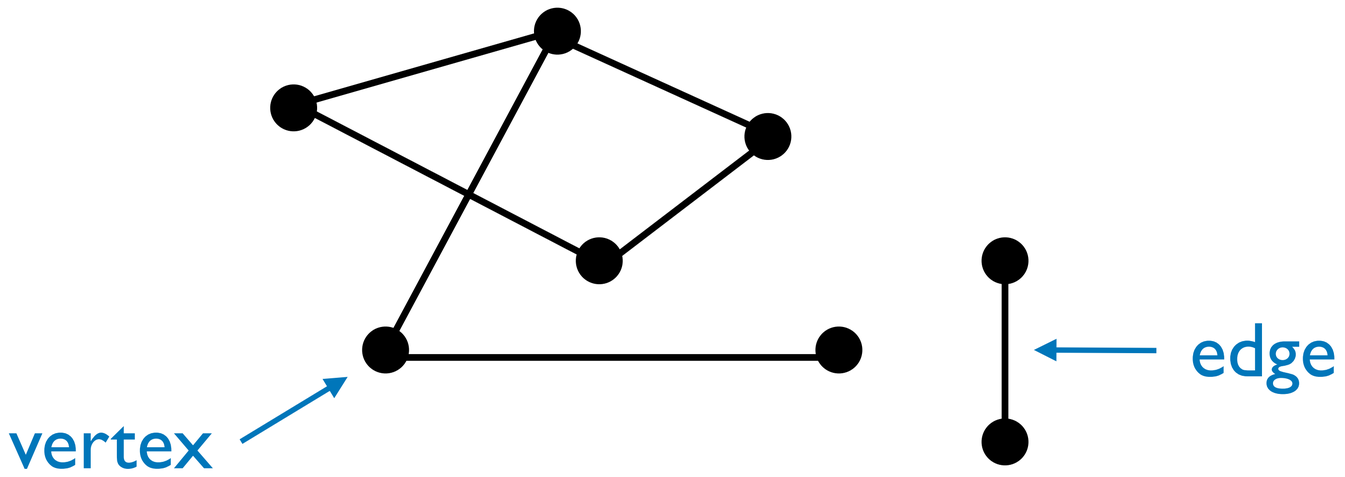
This week and next we talk about data structures that make use of graphs, specifically binary trees.

Then we talk about solving computational problems on graphs themselves.

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

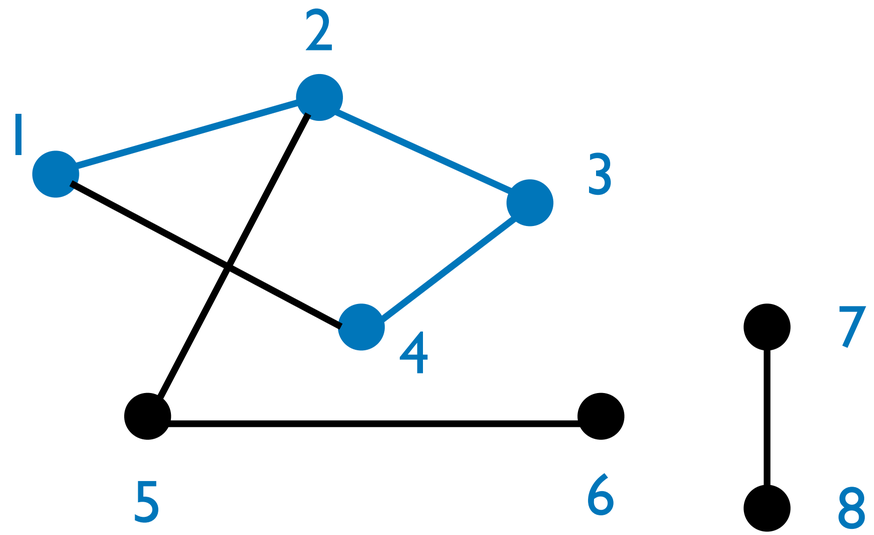
Graphs

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



Two vertices connected by an edge are called neighbours.

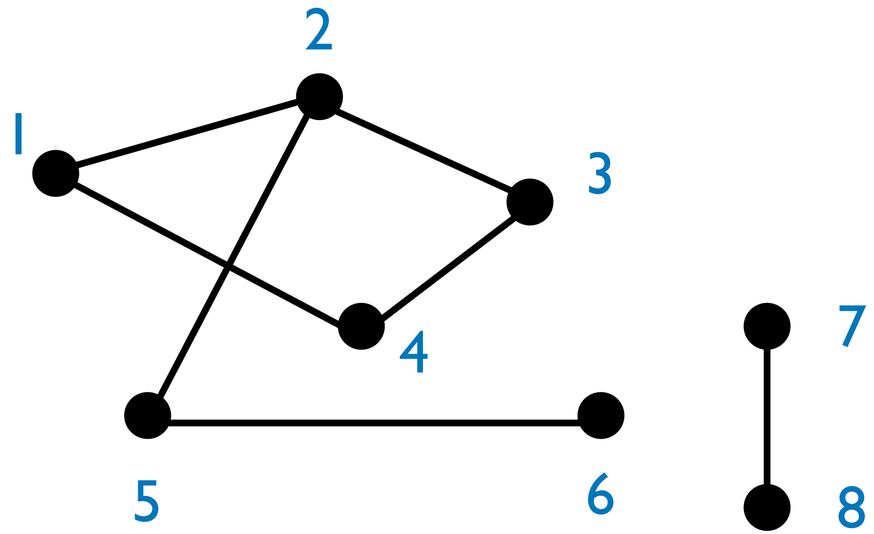
Graphs



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.

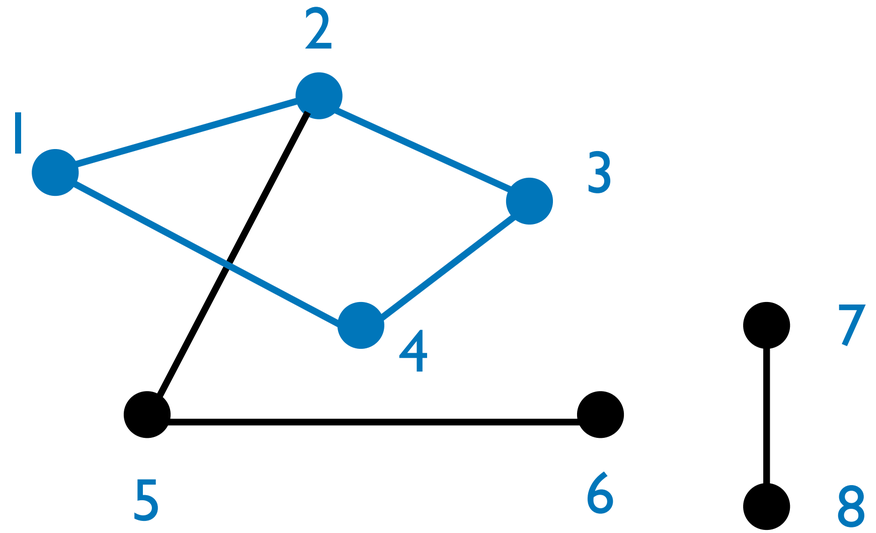
A graph is connected if and only if there is a path between every pair of vertices.



Is this graph connected?

Cycles

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

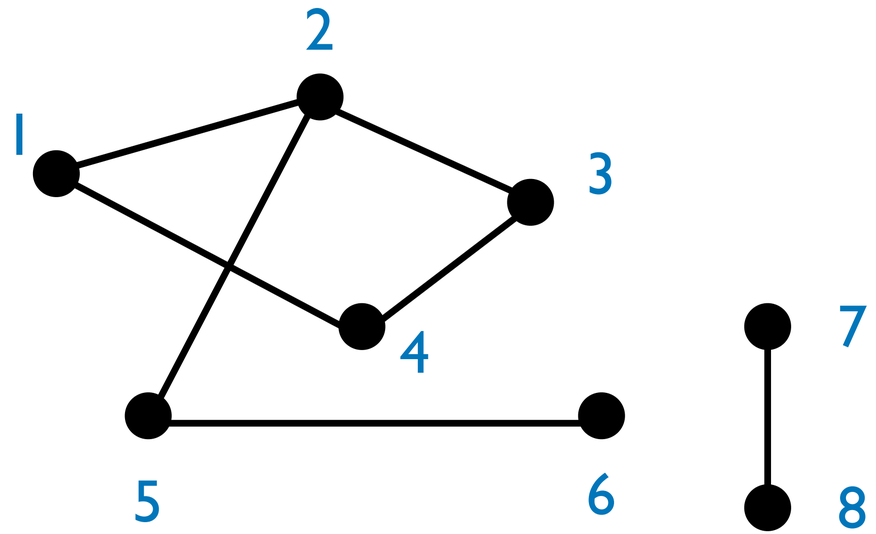


Now the blue edges form a cycle.

Trees

One of the simplest class of graphs are trees.

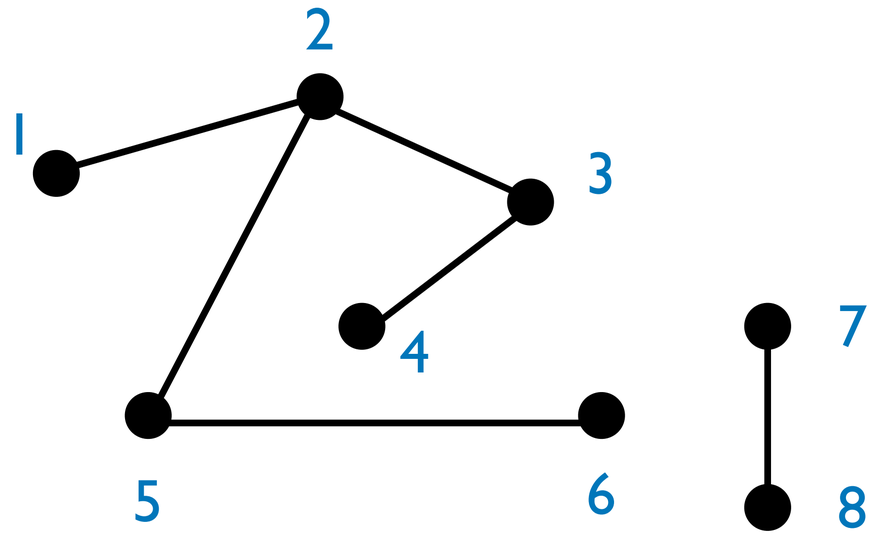
A tree is a connected graph with no cycles.



Is this graph a tree?

Trees

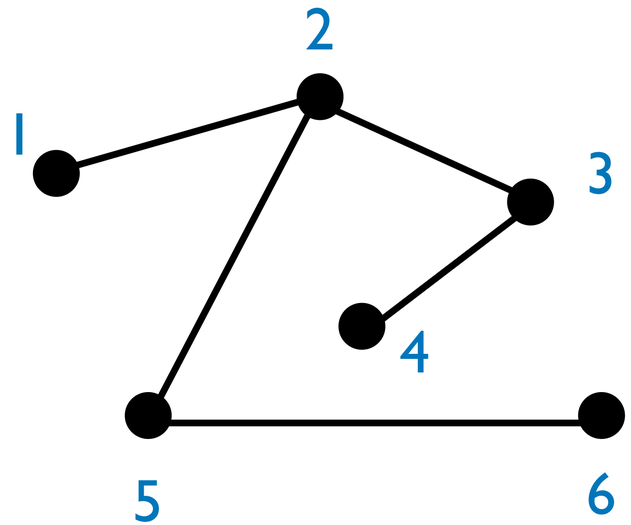
A tree is a connected graph with no cycles.



Is it a tree now?

Trees

A tree is a connected graph with no cycles.



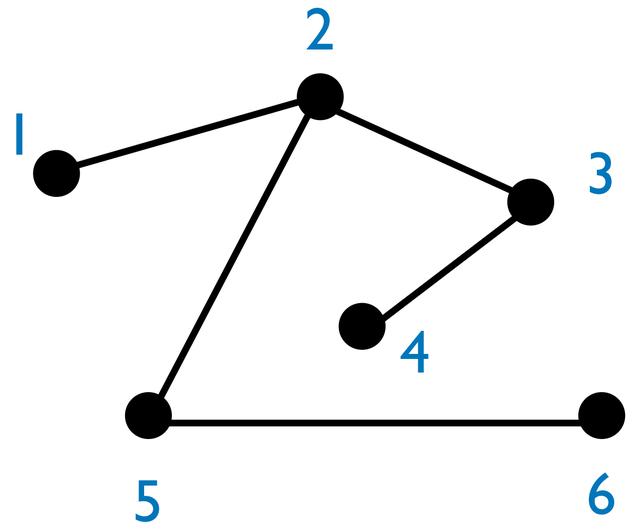
How about now?

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.



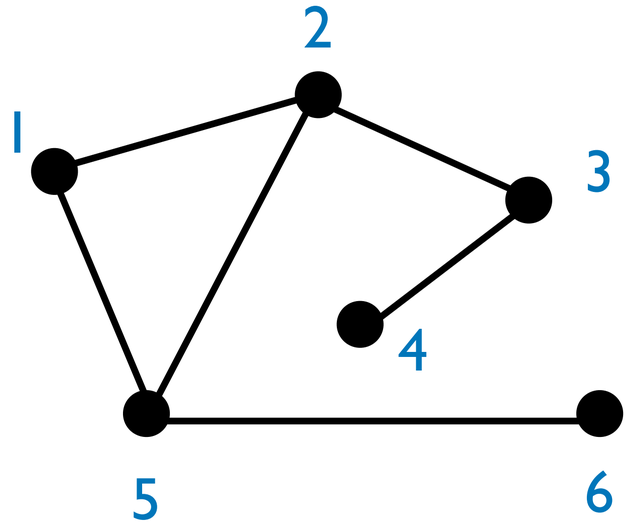
Trees

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.

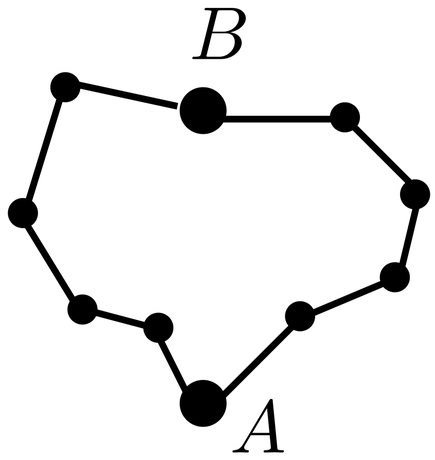
If the graph had a cycle, we could remove an edge of the cycle and the graph would still be connected.

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



Trees

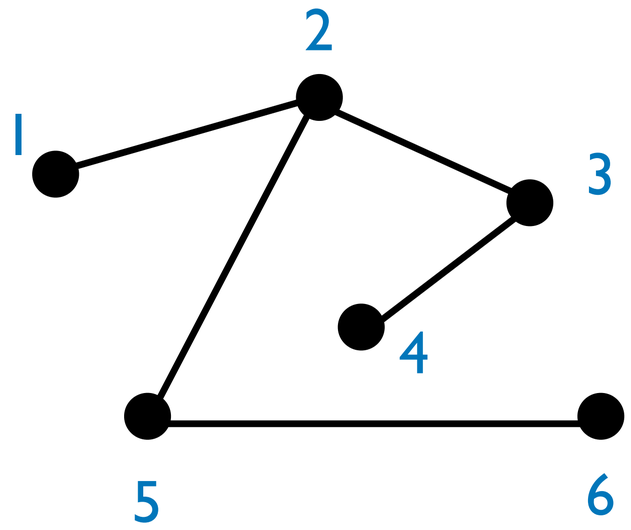
Nice fact: In a tree there is always a unique path (not reusing edges) that connects any two vertices.

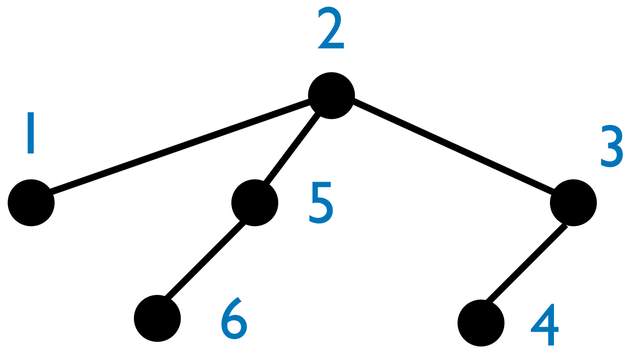


If there were two distinct paths this would create a cycle.

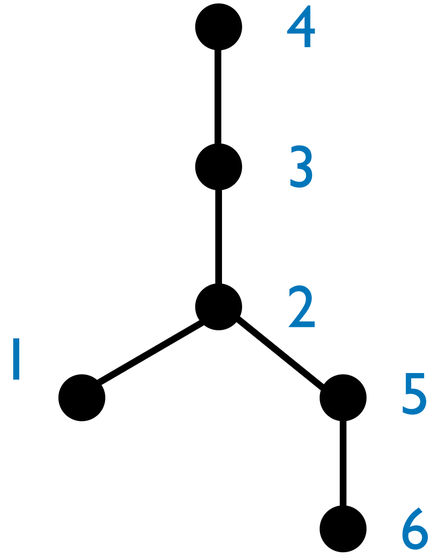
The distance between two vertices in a tree is the length of the path between them.

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

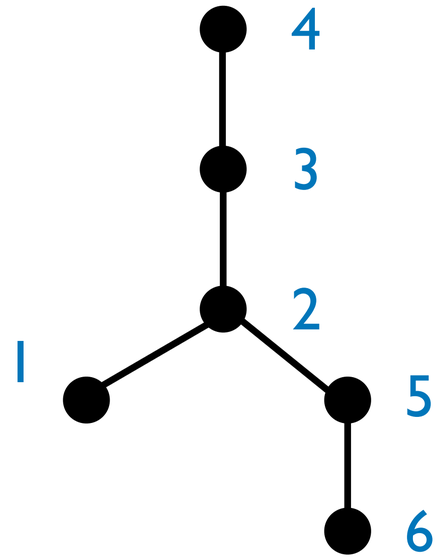
The depth of a vertex is its distance from the root.

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

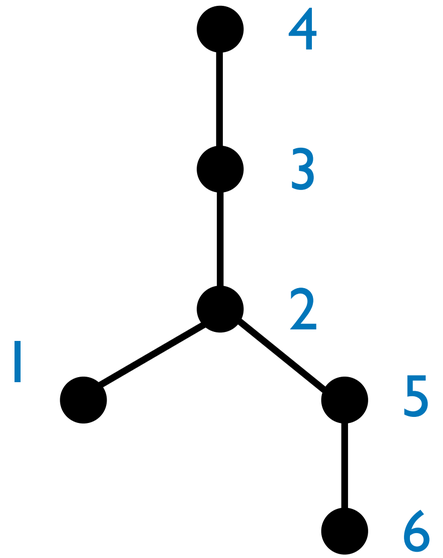
The children of a vertex at depth d are its neighbors at depth $d+1$.

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

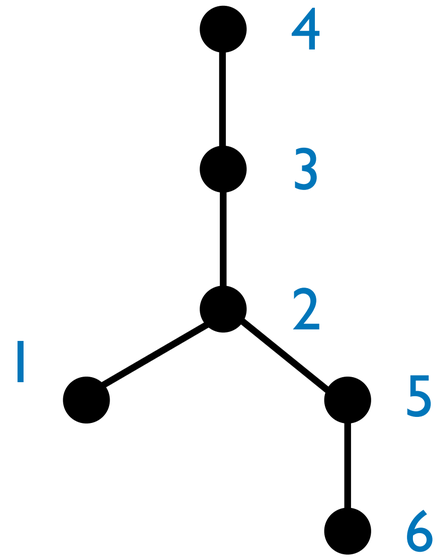
Parents

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.

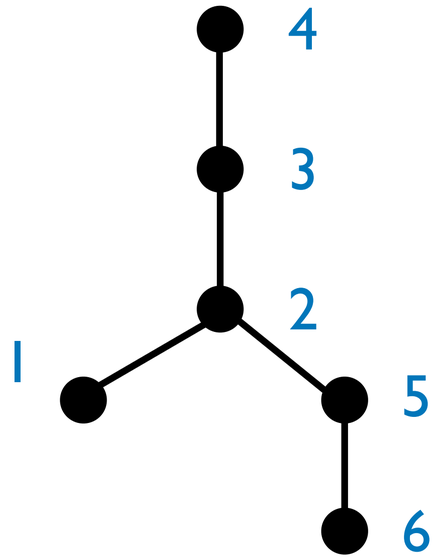
Every vertex except the root can be identified with a unique edge, the edge to its parent.

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.

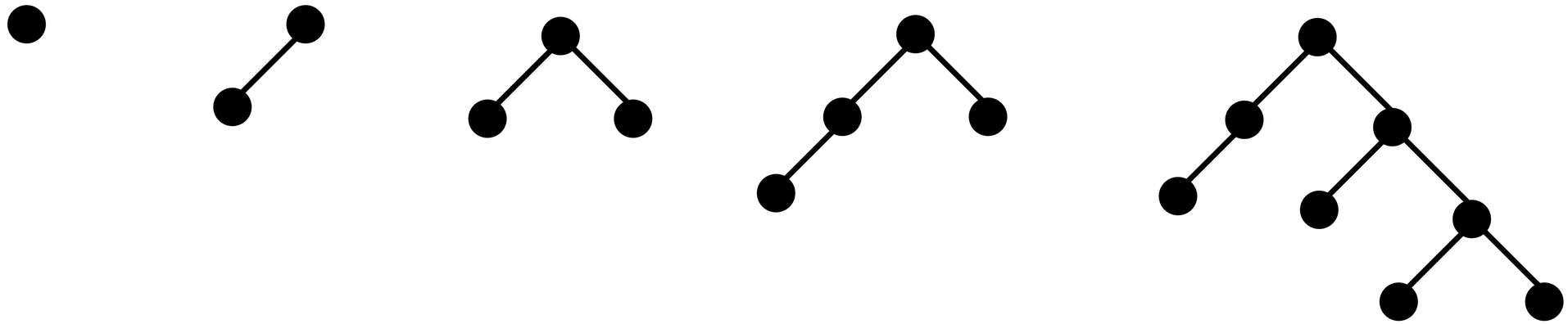
The height of a leaf is 0.



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

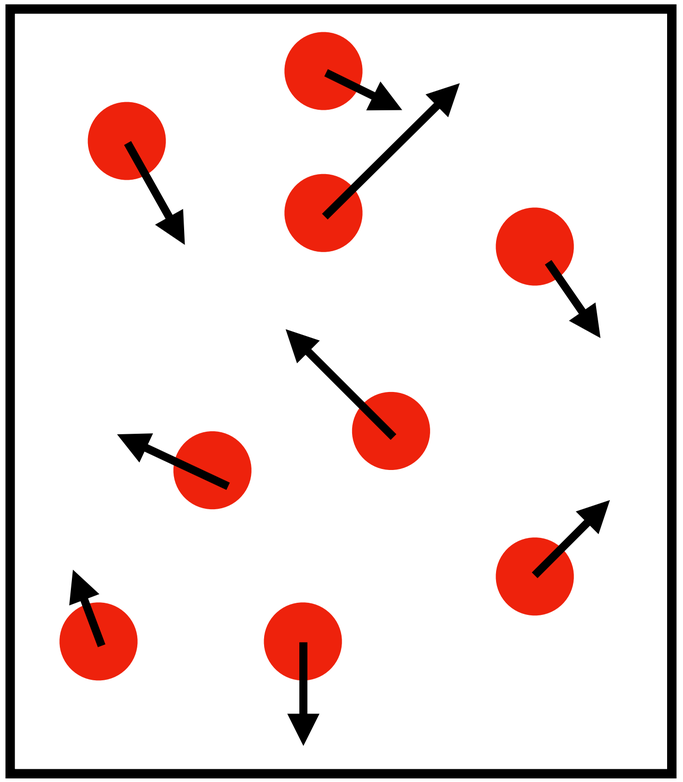
Particle Collision

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.

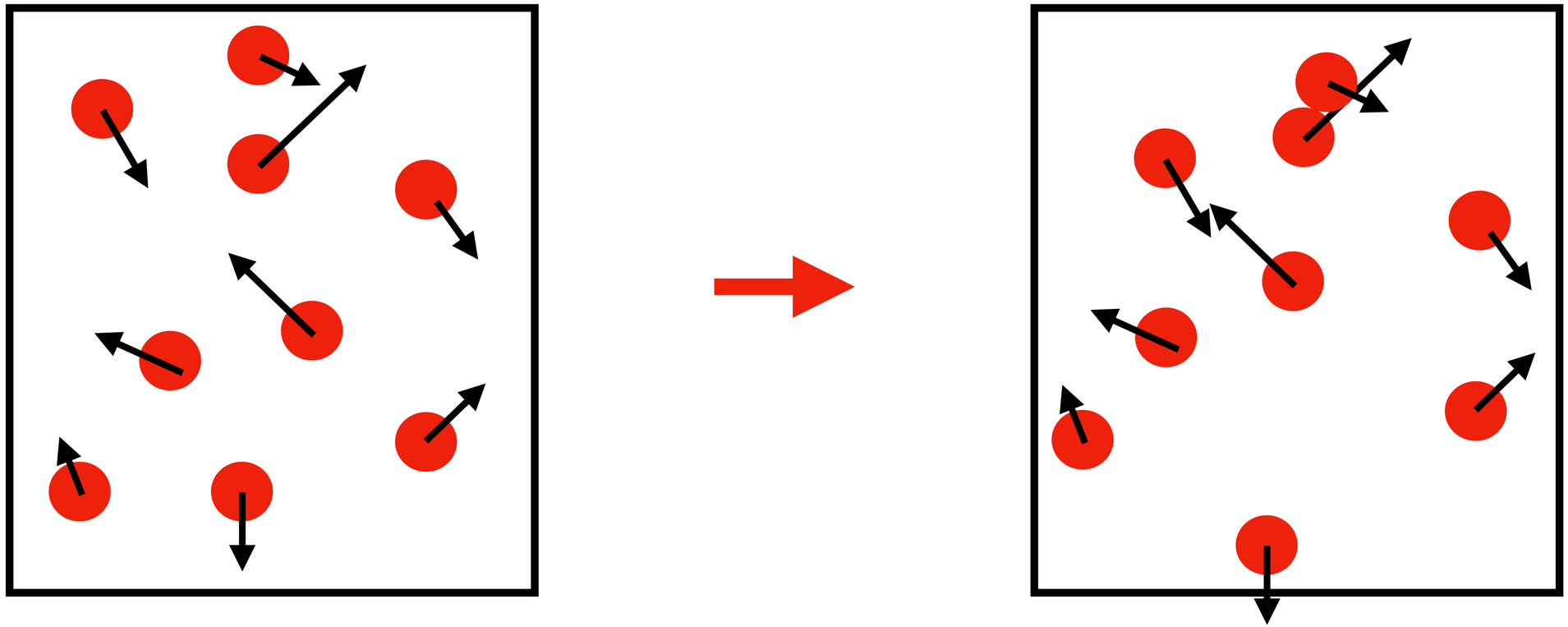
How can we evolve the system computationally?



Event-driven simulation

In between collisions velocities don't change.

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



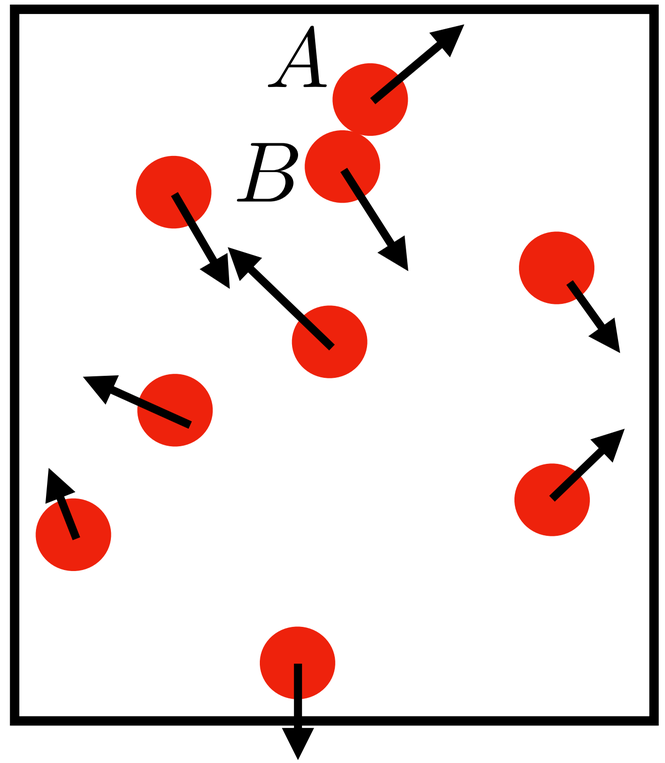
Update with collision

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

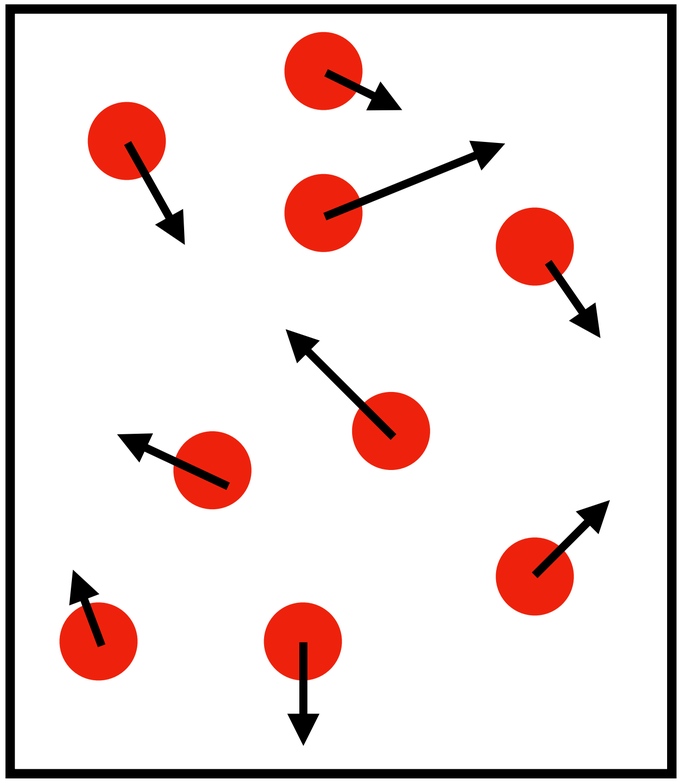
We want a database of collision times.

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:

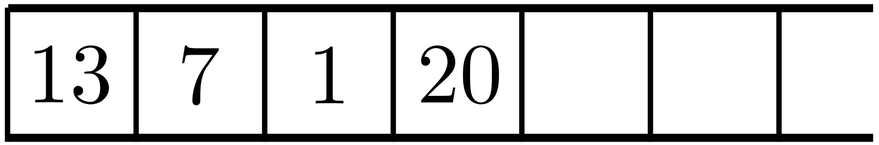
`insert`	put a new key in the set.
`remove_min`	delete the minimum key.
`peek`	look at the minimum key.

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.

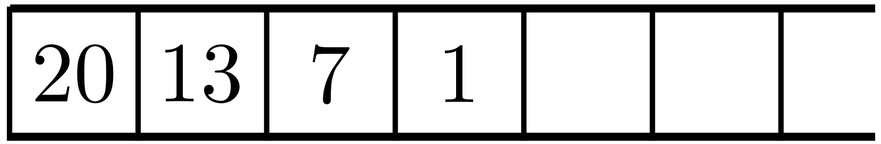
Unordered Array:



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

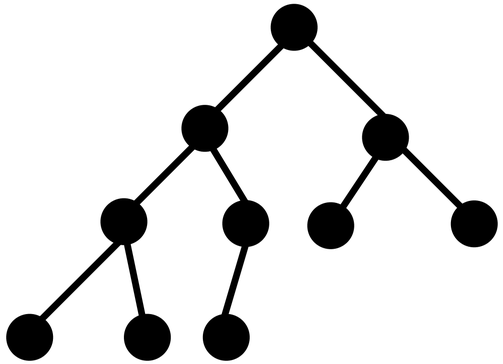
We will now see a way to implement a minimum priority queue using a binary heap.

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

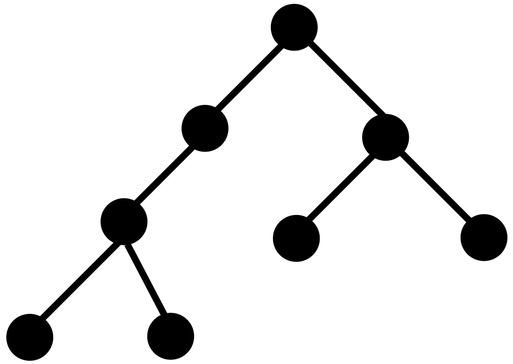
- We maintain that it is a complete binary tree.
- We maintain that it has the heap property.

Complete binary tree

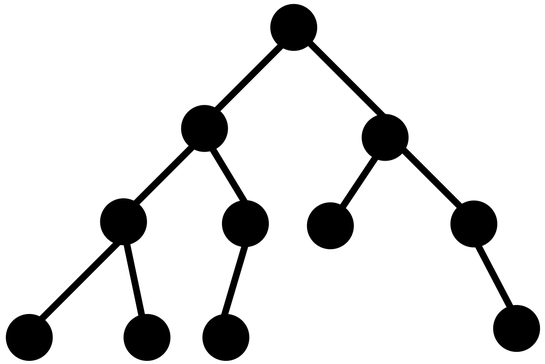
In a complete binary tree, every layer is totally filled except possibly the bottom one, which is filled from the left.



complete



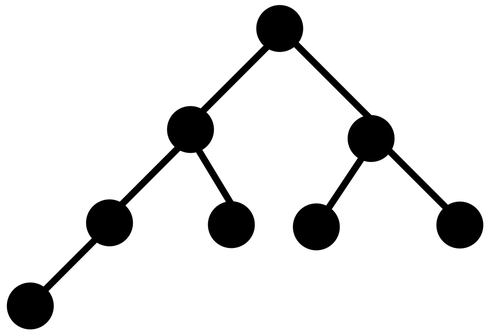
not complete



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.

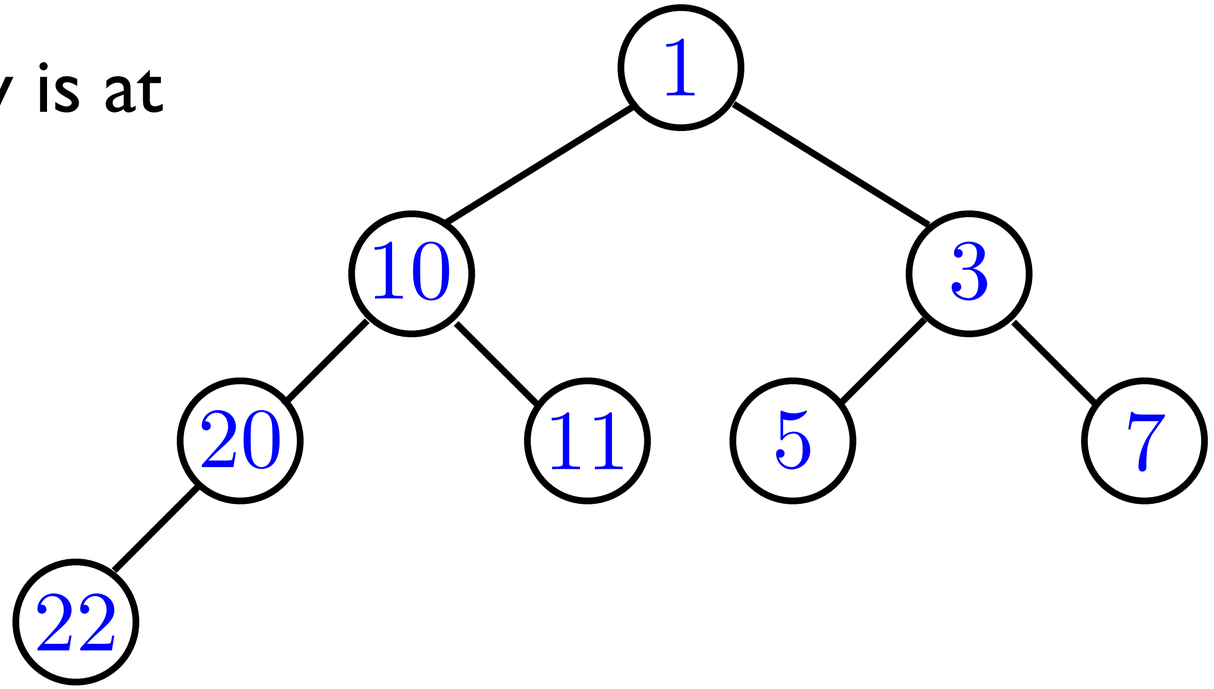
The number of nodes is $1 + \sum_{i=0}^{h-1} 2^i = 2^h$.

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.

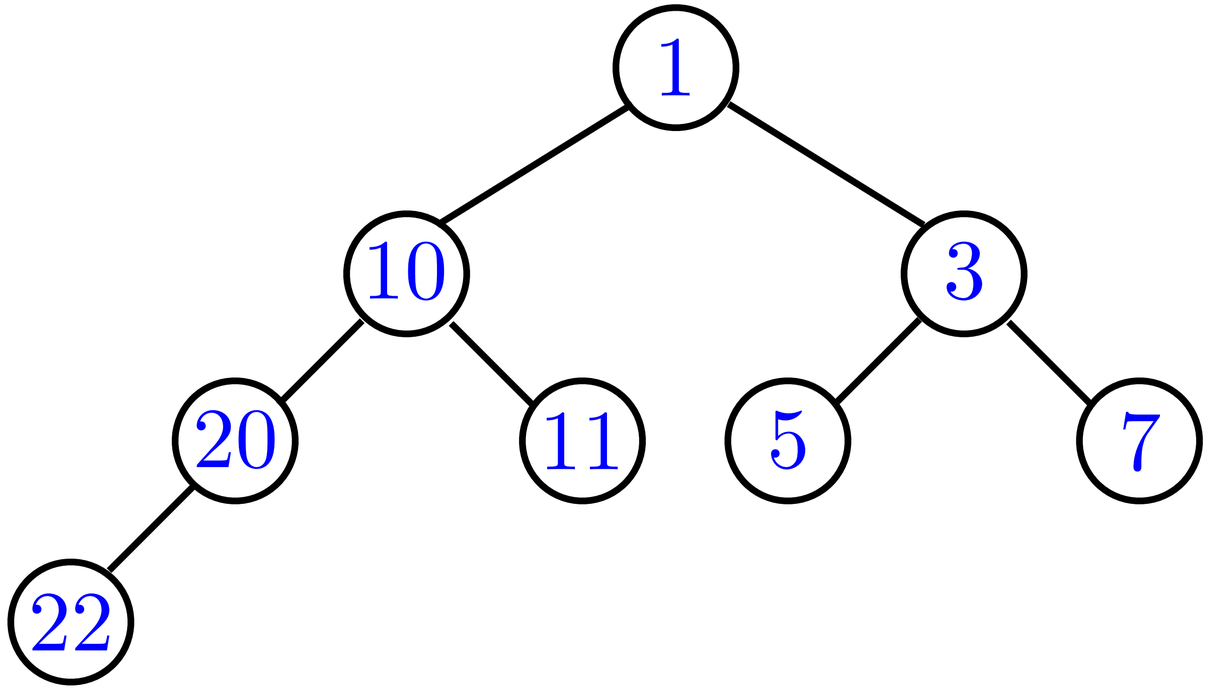
This guarantees that the minimum key is at the root.



Peek

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

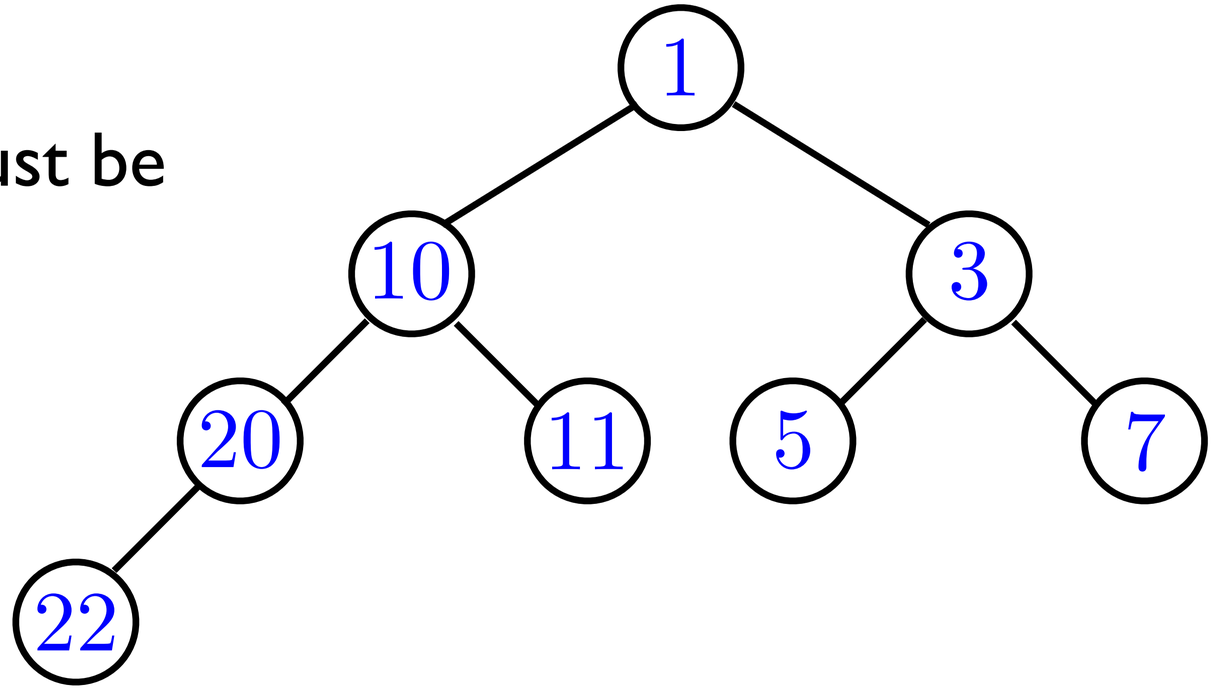
Just observe the root!



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

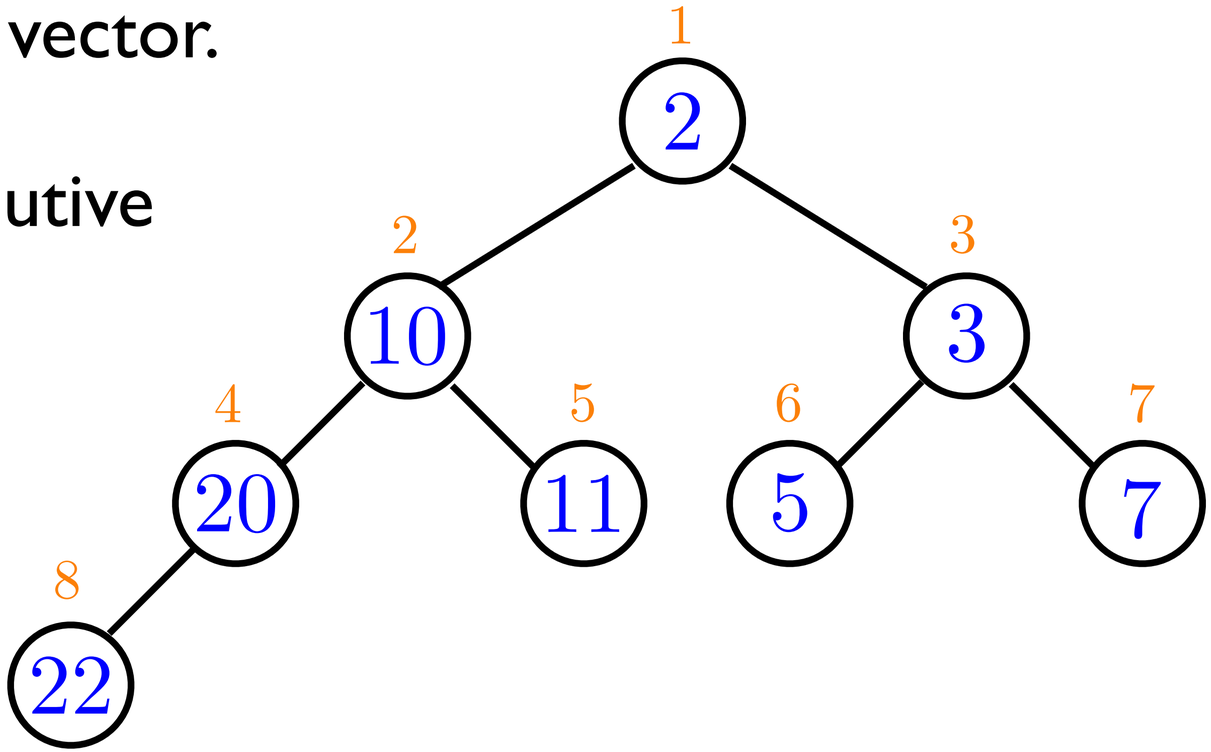
Representing a binary tree

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.

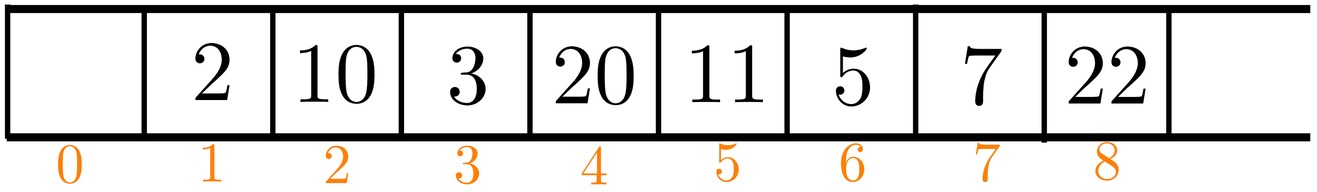
We start from 1 for cleaner indexing.



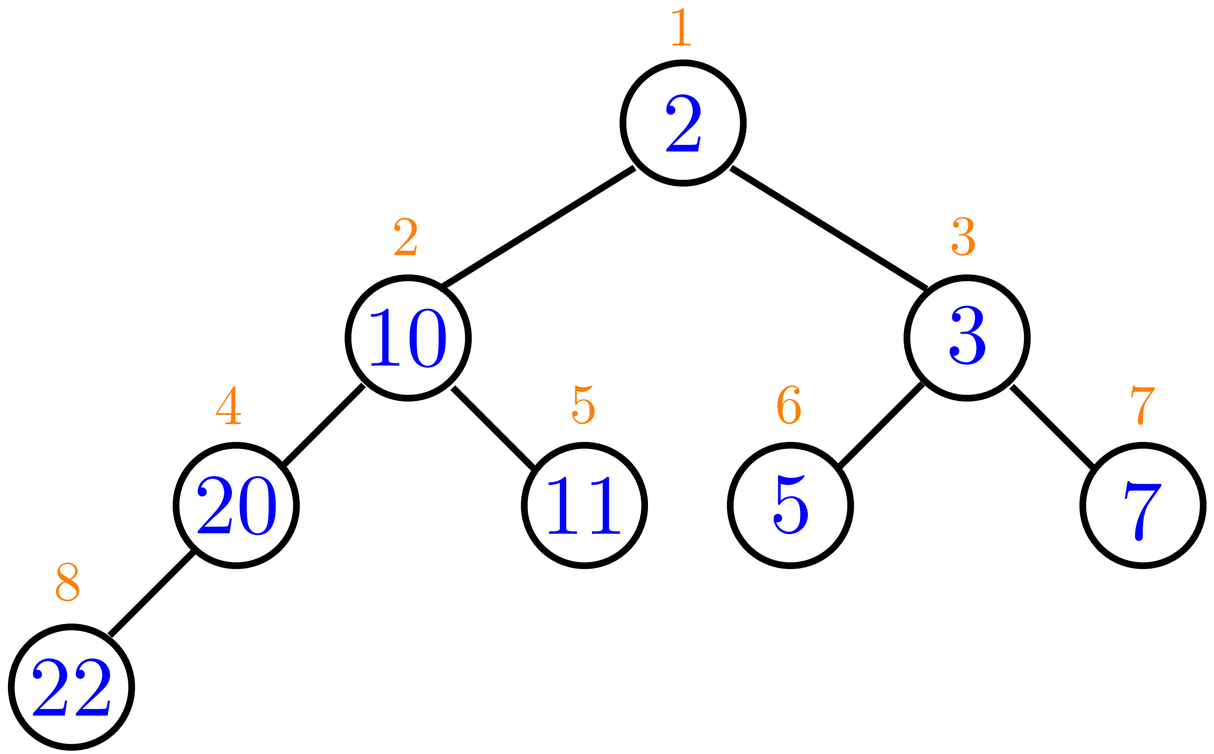
0	1	2	3	4	5	6	7	8	
	2	10	3	20	11	5	7	22	

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;
}
```



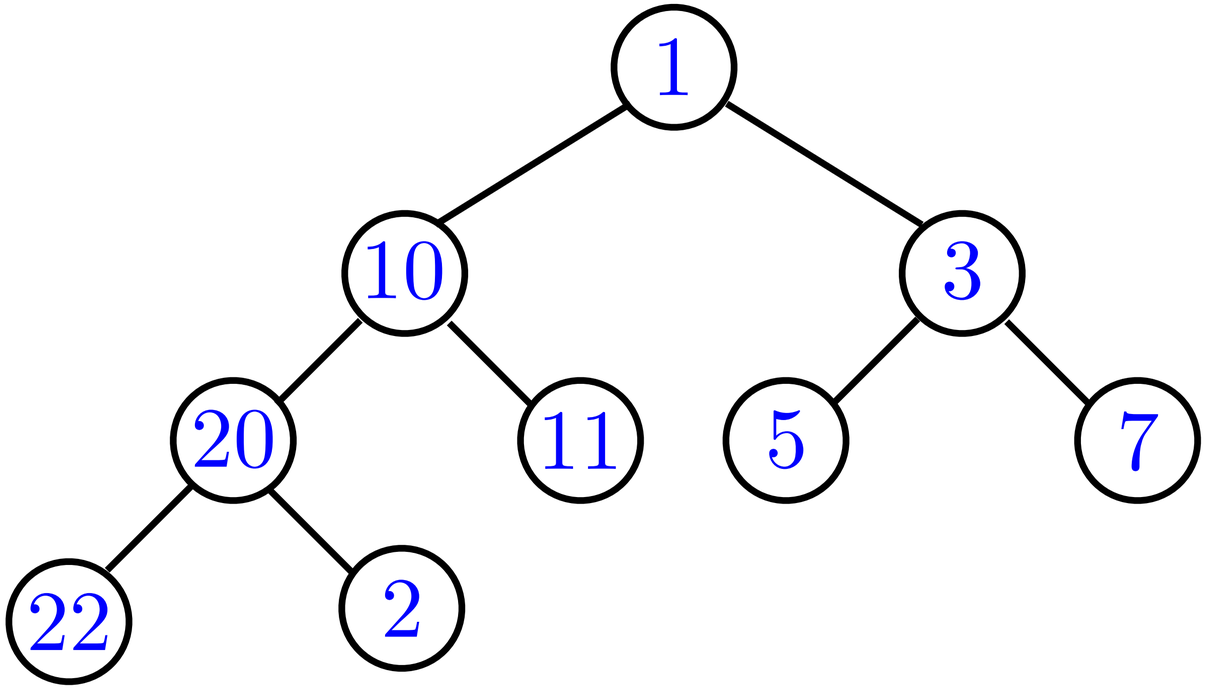
Insert

Now let's see how to insert a key.

Say we want to insert 2.

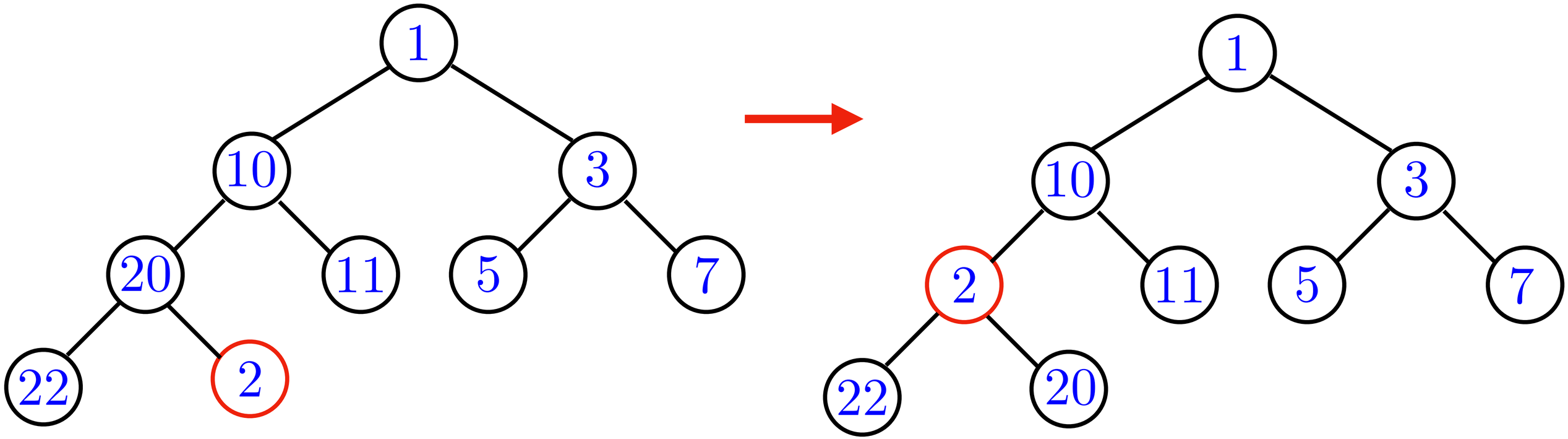
To maintain a complete binary tree, we insert at the leftmost free slot in the bottom row.

This might destroy the heap property.



Swim

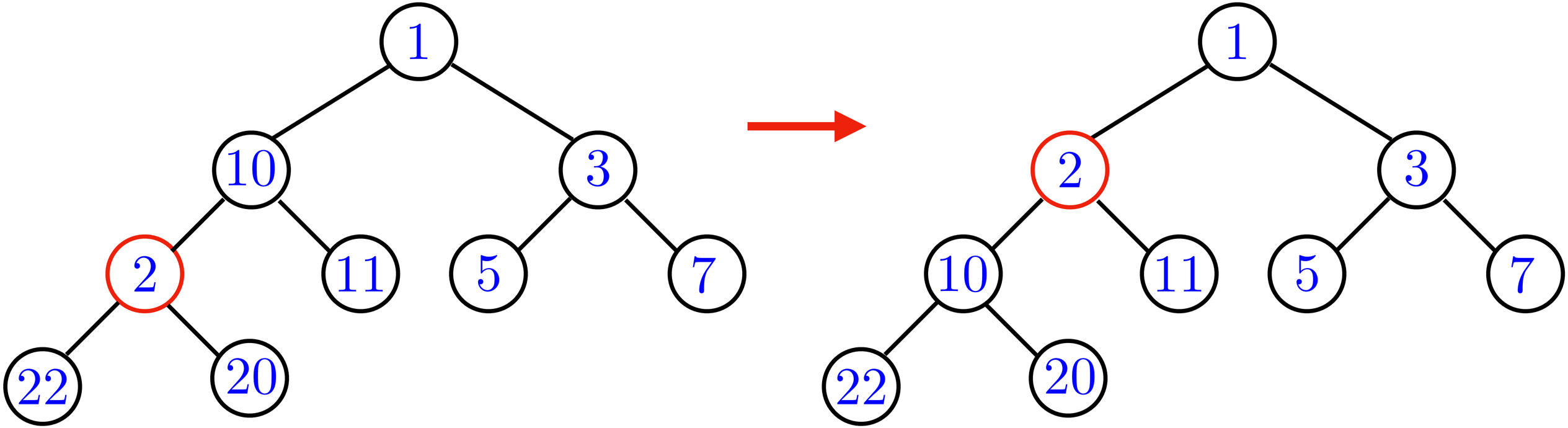
To fix the heap property, we the inserted key "swims" up until it is at least its parent.



Are we done?

Swim

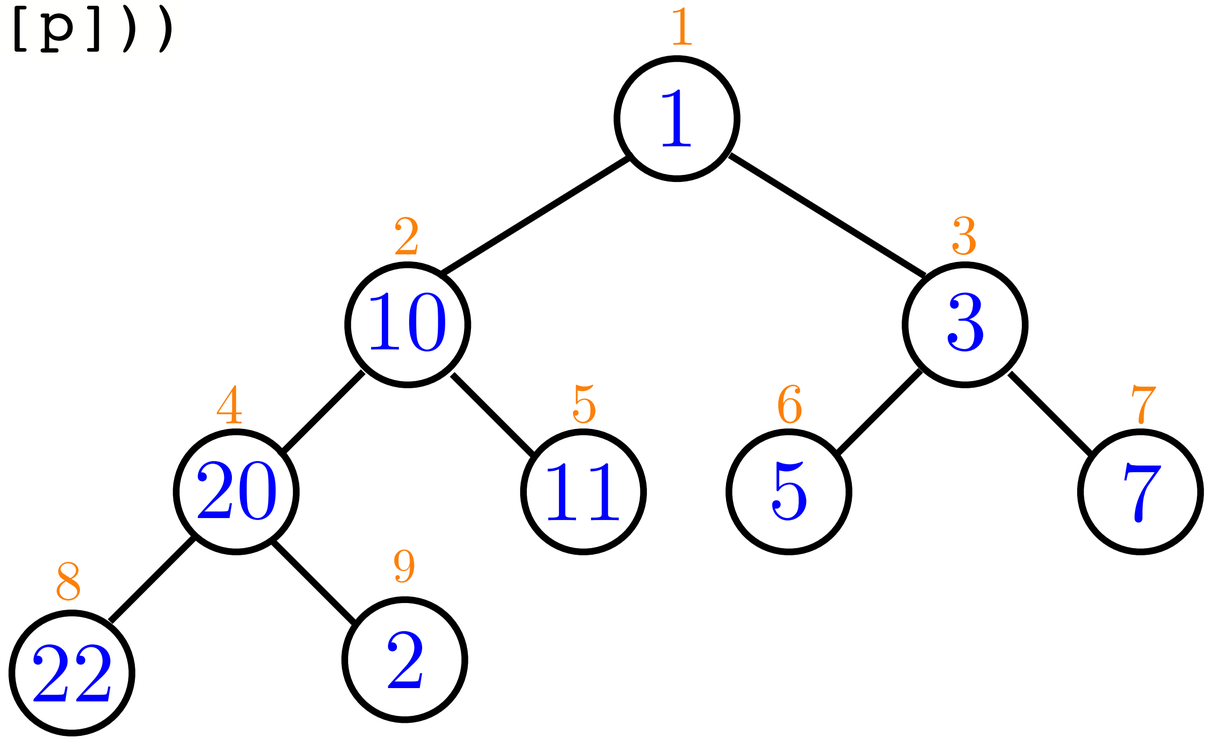
To fix the heap property, we bubble the inserted key up until it is at least its parent.



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.

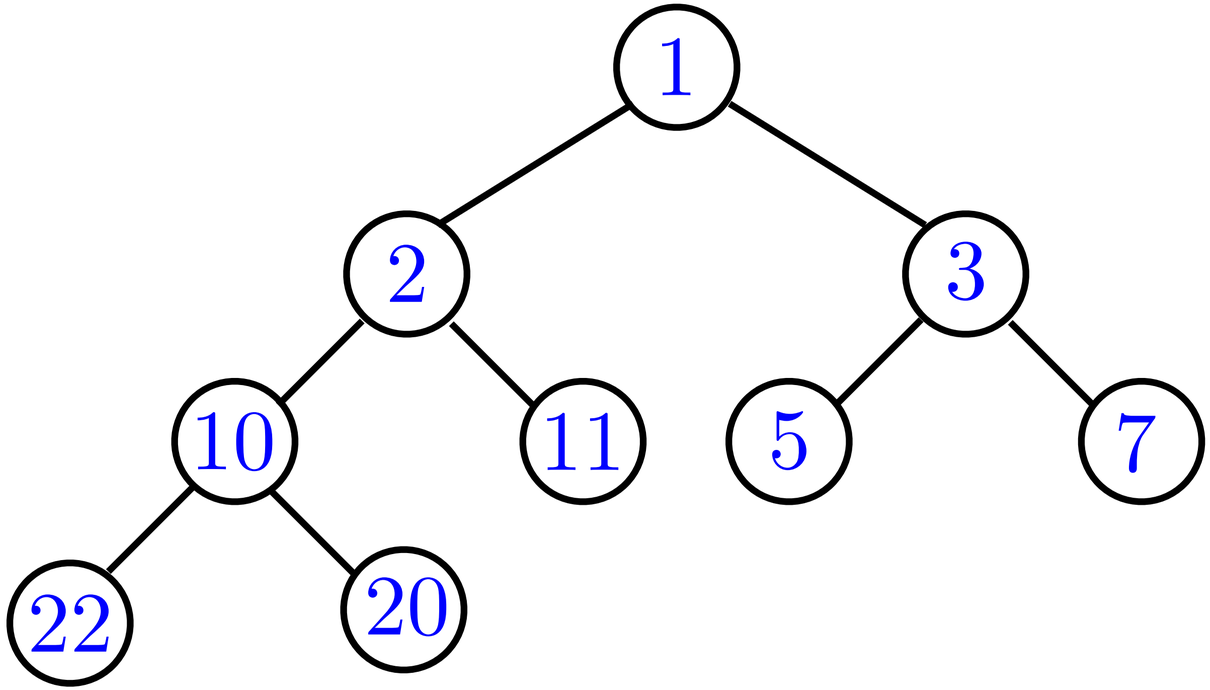
The number of iterations of the while loop is at most the height of the tree.

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.

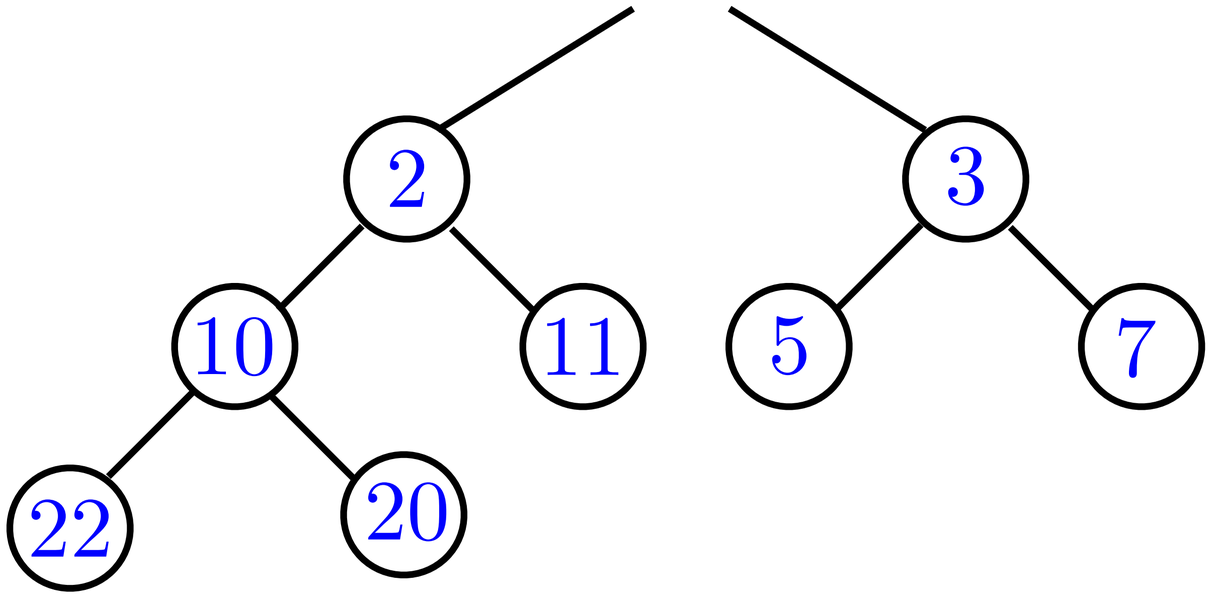
This preserves the complete binary tree property, but again may destroy the heap property.



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.

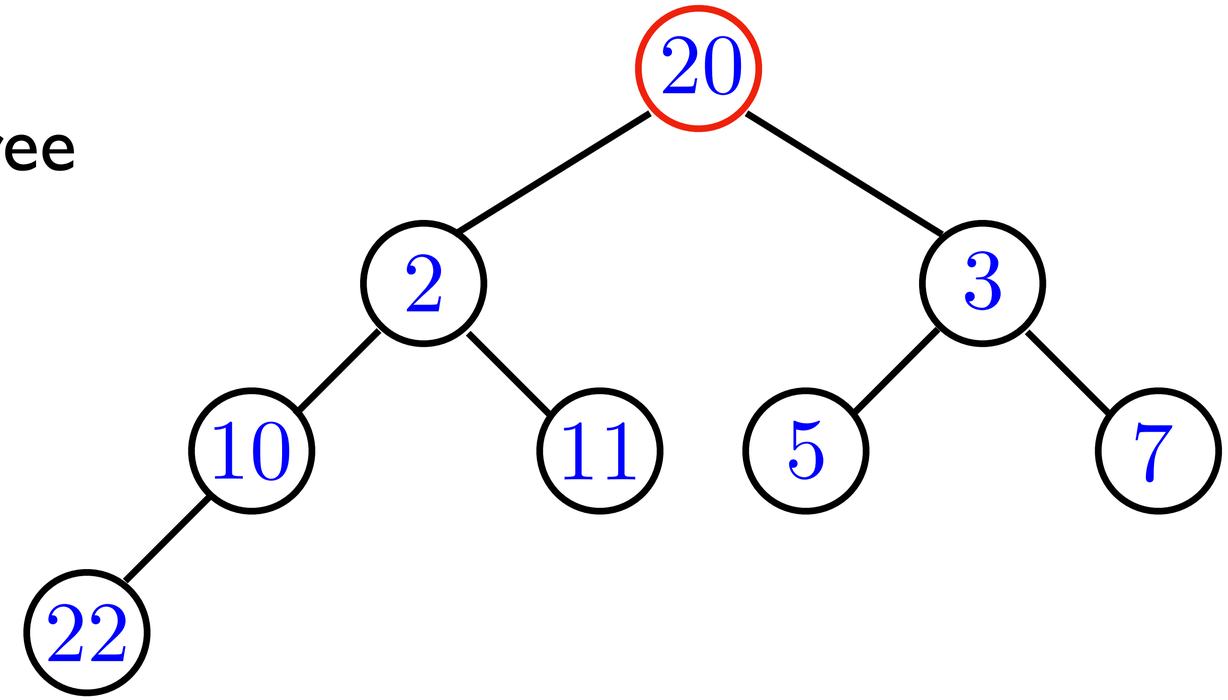


remove_min

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

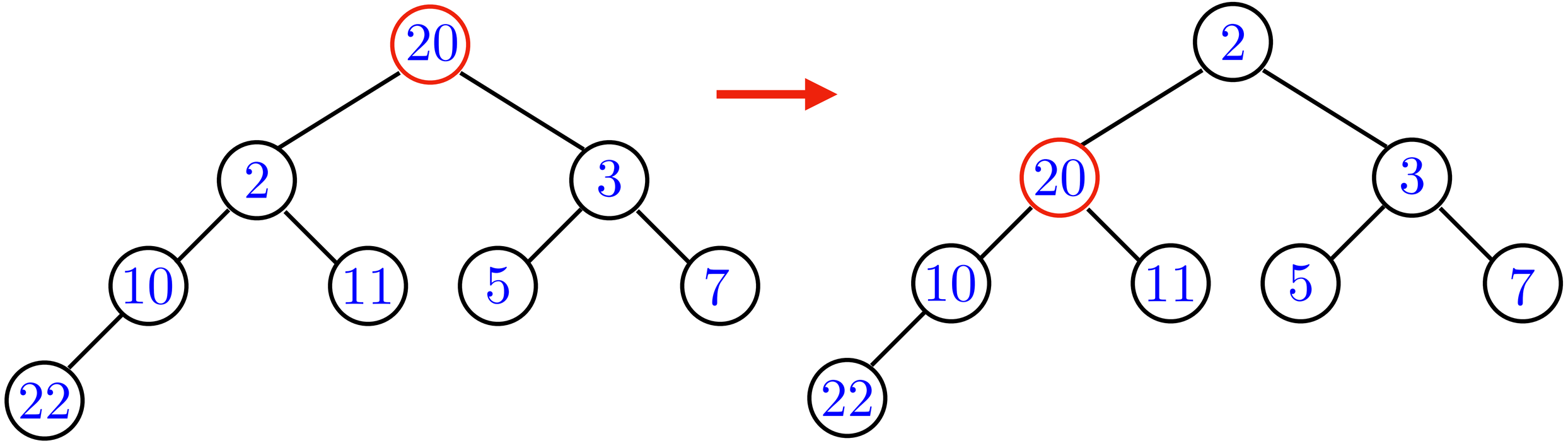
This preserves the complete binary tree property, but again may destroy the heap property.

Now we need to restore the heap property.



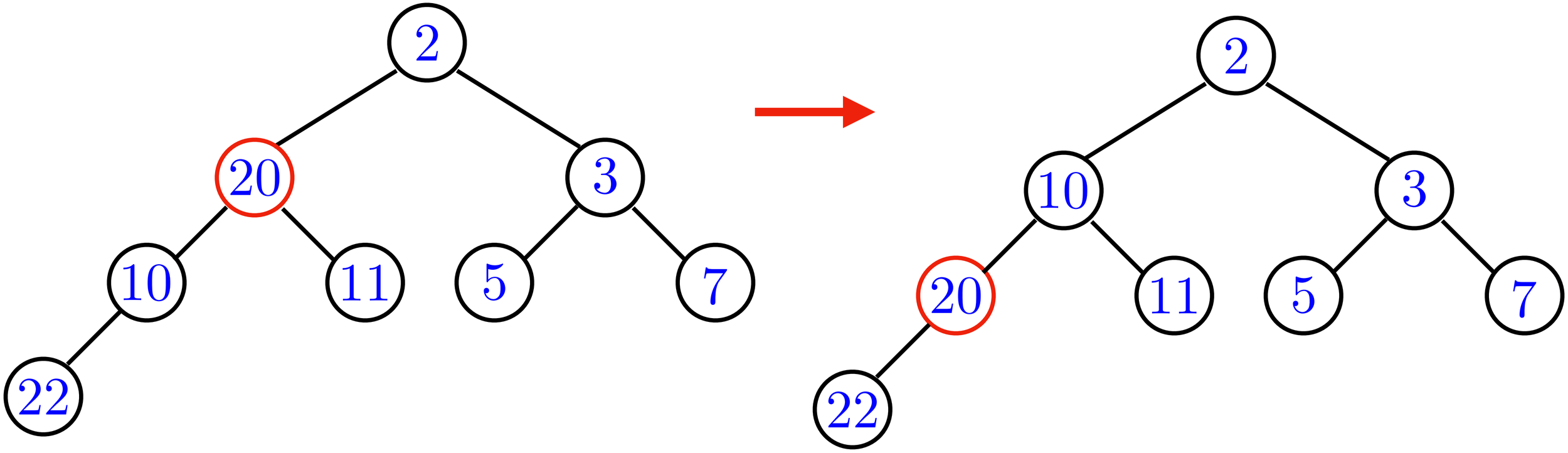
Sink

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



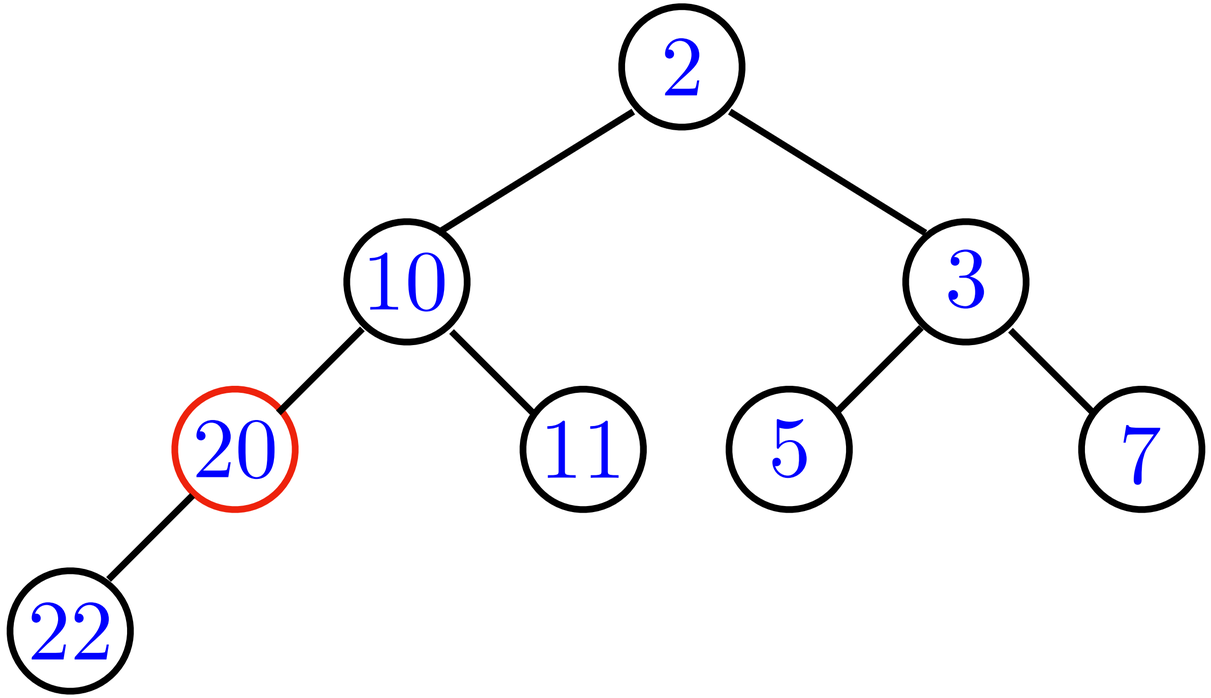
Sink

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



Sink

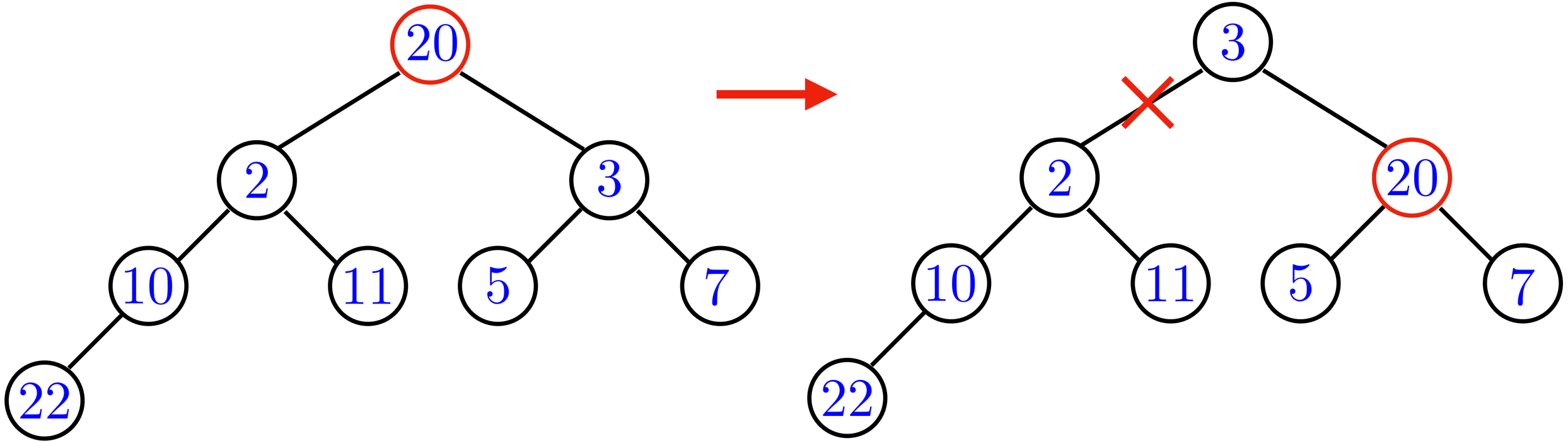
We keep bubbling down until the key of the node is at most that of its children.



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

Binary Search Trees: Motivation

Runway Reservations

Problem: You are in charge of an airport with a single runway. Planes radio in to request a landing time. You need to schedule the landing times so that no planes land $< k$ minutes of each other.

Example with $k = 3$:

current set	request	decision
$\emptyset$	13	scheduled
$\{13\}$	7	scheduled
$\{7, 13\}$	9	denied
$\{7, 13\}$	22	scheduled

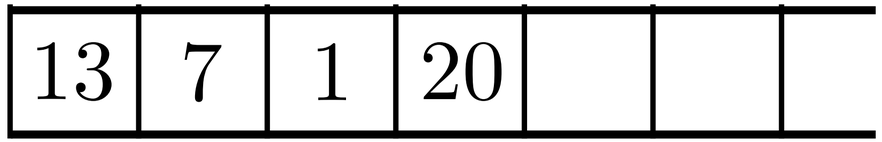
Abstracting the problem

We want to maintain a set of keys (landing times). We want to check if a key satisfies the time constraint, and if so insert it into the database. We also want to be able to remove keys.

What data structure is good for this problem?

Unordered Array

We check if a request is valid and if so insert it at the end of an array.



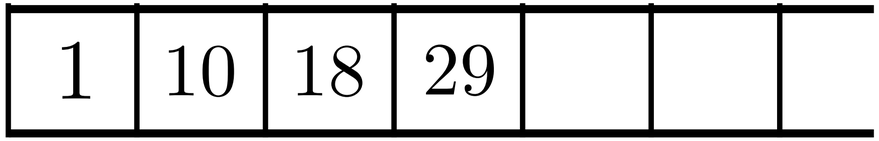
What is the problem with this solution?

insertion is $\Theta(1)$.

checking the constraint is $\Theta(n)$.

Sorted Array

How about if we maintain a sorted array?

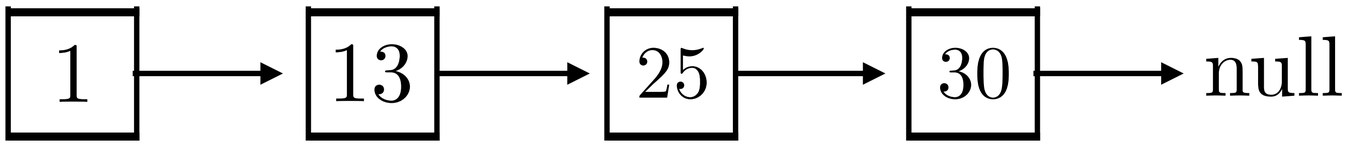


Now we can check if a request is valid via binary search.

Checking the constraint is $\Theta(\log n)$.

What is the problem with this solution?

Sorted Linked List



What is the problem with this solution?

Checking the constraint is $\Theta(n)$.

Set/Map

If you thought of std::set or std::map, you are exactly right!

A map provides the functionality needed for the runway reservation problem.

Map is typically implemented with a (balanced) binary search tree, the data structure we will see today.

Binary search trees combine the benefits of the sorted array and linked list approaches.

Formalising Runway Reservations

Abstract Data Type

Let's develop a set of operations to solve the runway reservation problem.

We want to maintain a set of keys. Each key can be associated with some data. We think of them as $\{\text{key}, \text{record}\}$ pairs.

We have two dynamic operations that modify the database:

$\texttt{insert}(\text{key}, \text{record})$

$\texttt{remove}(\text{key})$

Abstract Data Type

We want to maintain a set of keys. Each key can be associated with some data. We think of them as $(\text{key}, \text{record})$ pairs.

Operations to extract information from the database:

`contains`(key)	check if a key is in the database
`successor`(key)	find the next largest key in the database
`predecessor`(key)	find the next smallest key in the database

For successor and predecessor we will assume the argument is already in the database.

Abstract Data Type

A somewhat roundabout solution to the runway reservation problem.

Request for landing time t comes in with plane info in data.

Run $\texttt{contains}(t)$. If t is already in the database, reject. Otherwise do:

$\texttt{insert}(t, \mathrm{data})$

$\mathrm{prev} = \texttt{predecessor}(t)$

$\mathrm{next} = \texttt{successor}(t)$

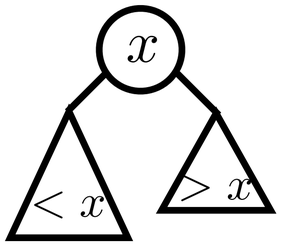
If $t - \mathrm{prev} \geq k$ and $\mathrm{next} - t \geq k$ then we are done. Otherwise reject and

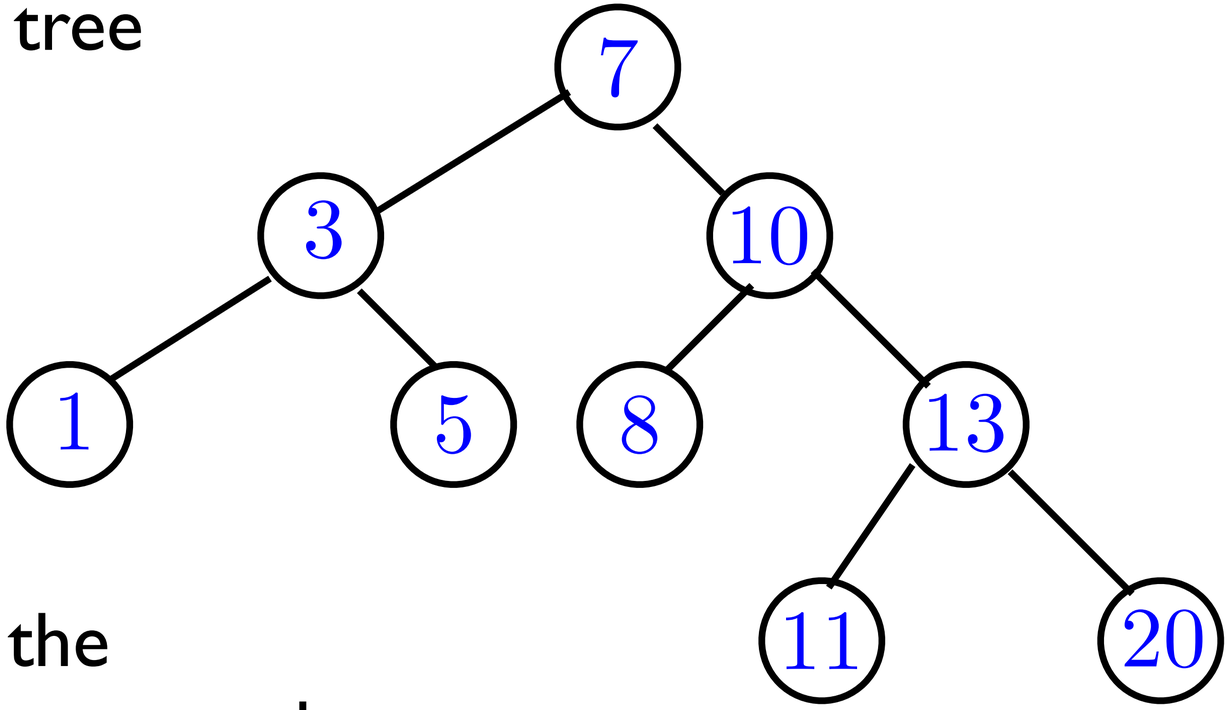
$\texttt{remove}(t)$

Binary Search Trees

Binary Search Tree

A binary search tree (BST) is a binary tree with keys labelling the vertices.





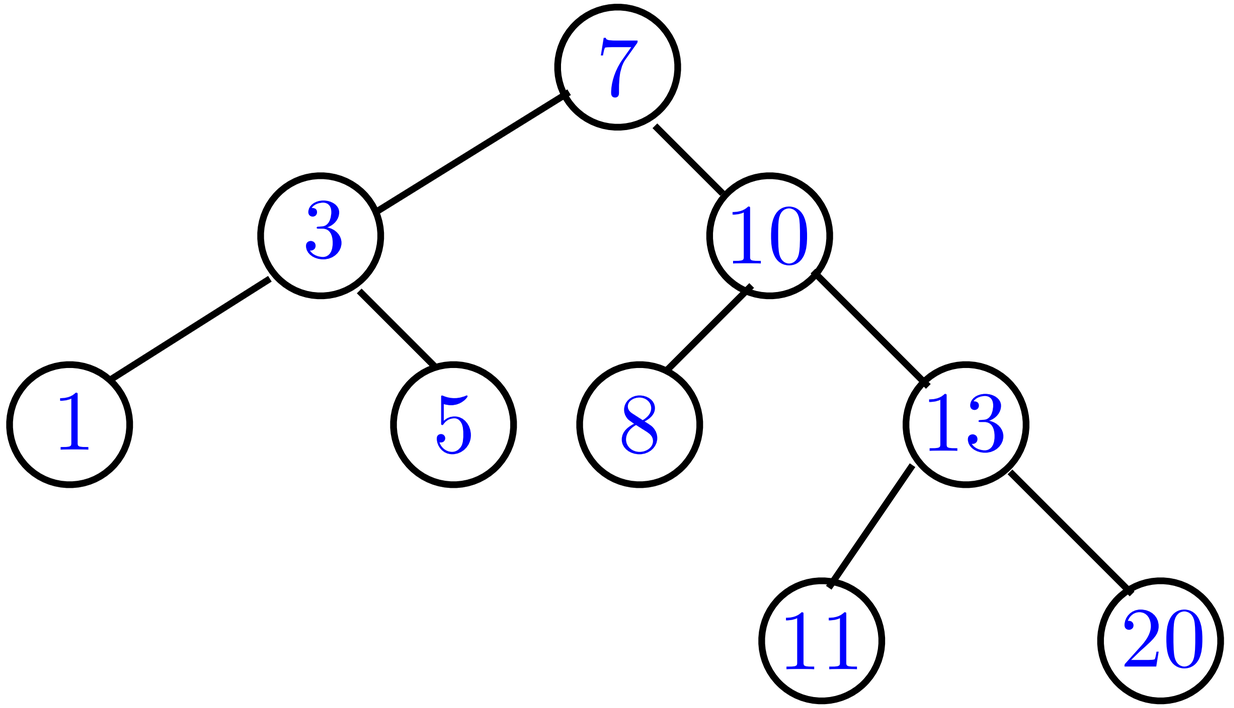
BST property: for any vertex v,

1) all keys in the subtree rooted at the left child of v are less than the key at v, and

2) all keys in the subtree rooted at the right child of v are greater than the key at v.

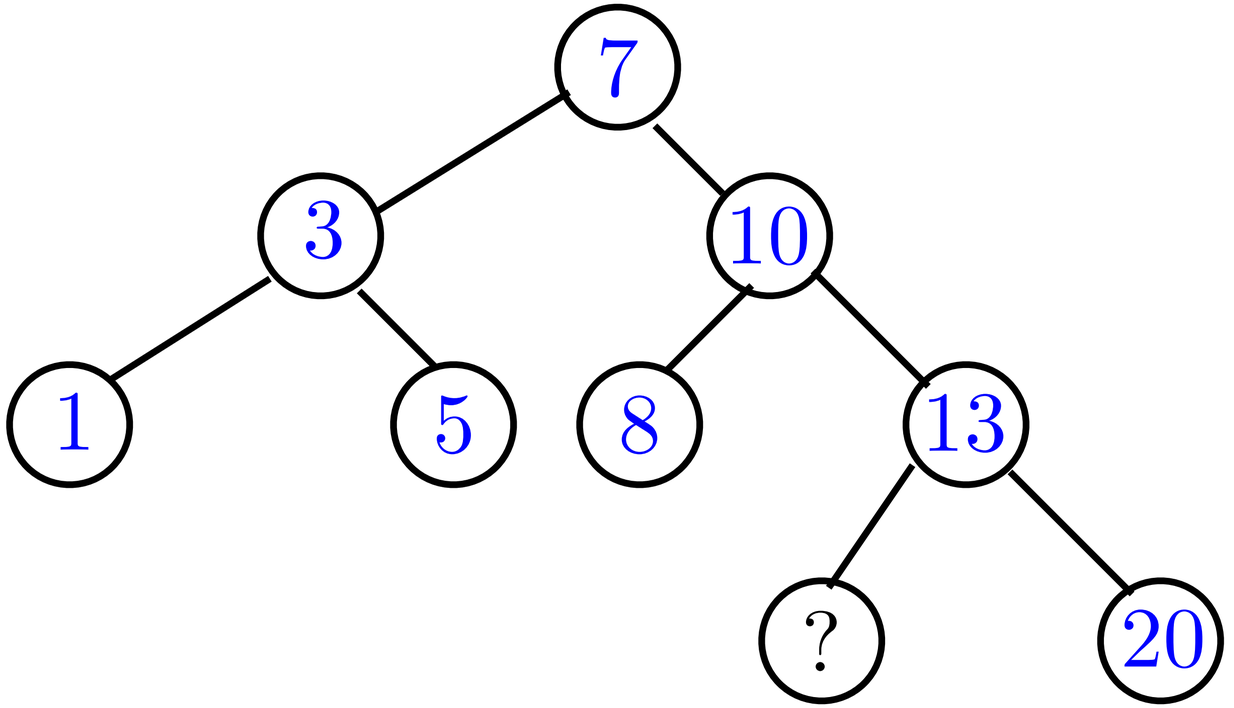
Question

Where is the minimum key in a BST?



Question

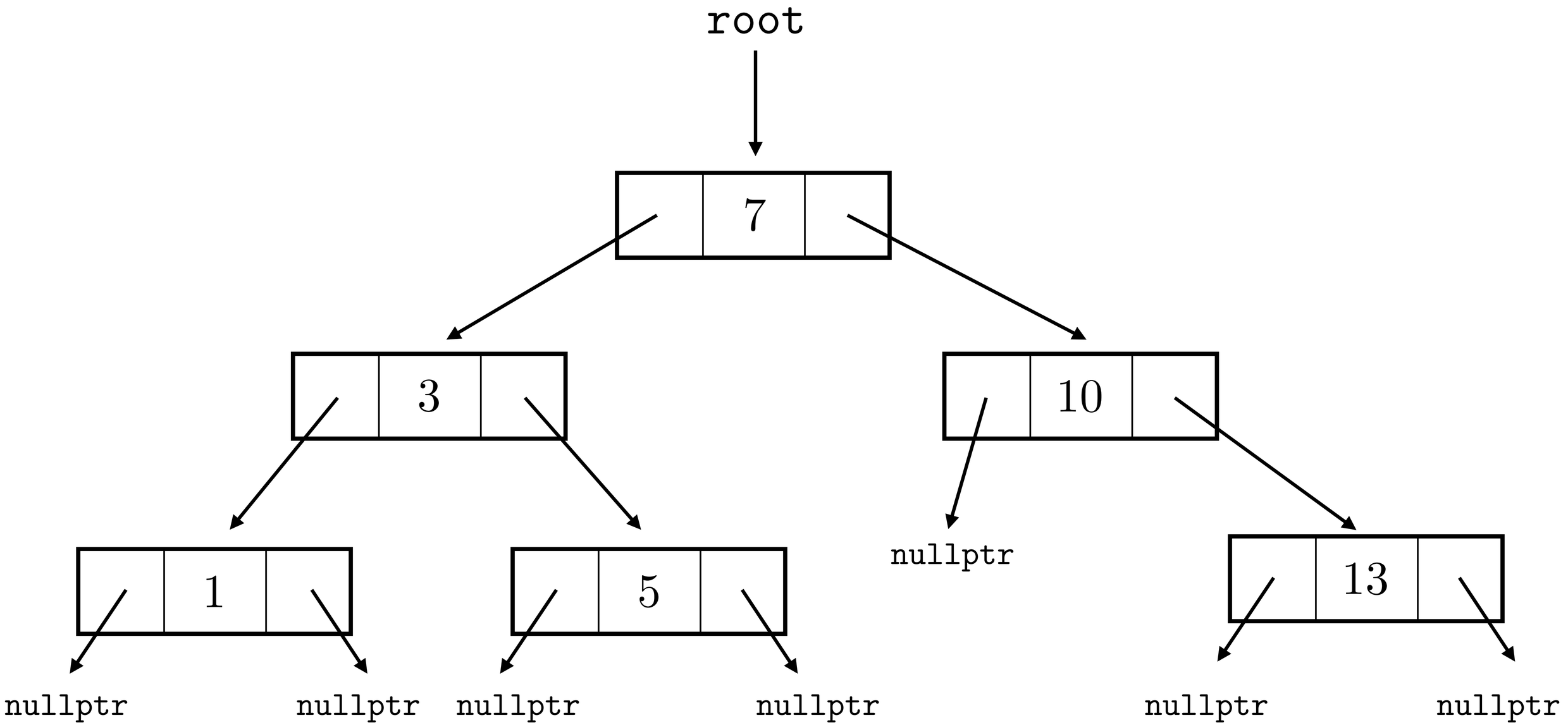
What integers could be placed at the question mark?



Representation of a BST

```
class Node {
public:
  int key;
  Node* left;
  Node* right;
};
```

```
class BST {
  private:
    Node* root;

  public:
    BST();
    ~BST();

    void insert(int k);
    void remove(int k);
    Node* contains(int k);
    Node* successor(int k);
    Node* predecessor(int k);
};
```


root
7
3
10
1
5
nullptr
13
nullptr
nullptr
nullptr
nullptr
nullptr
nullptr

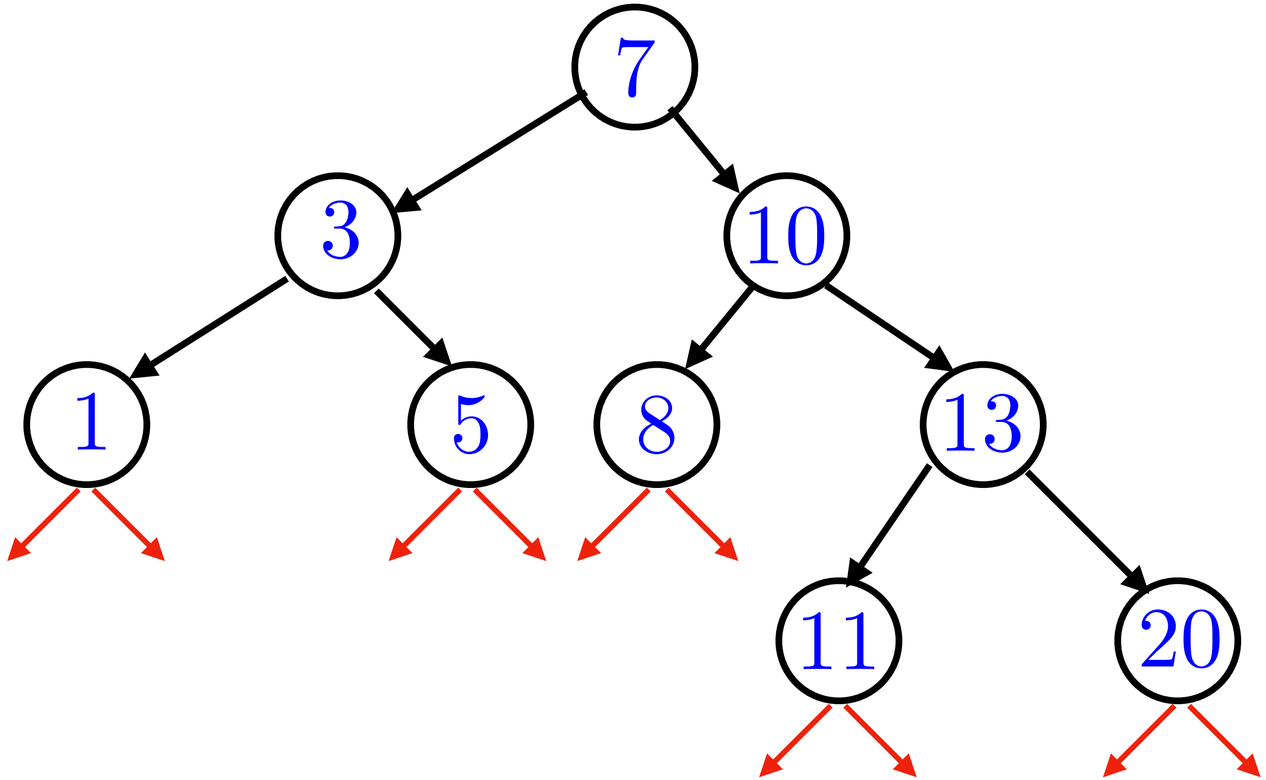
Contains

Let us first see how to check if a key is in a BST.

`contains(11)`

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



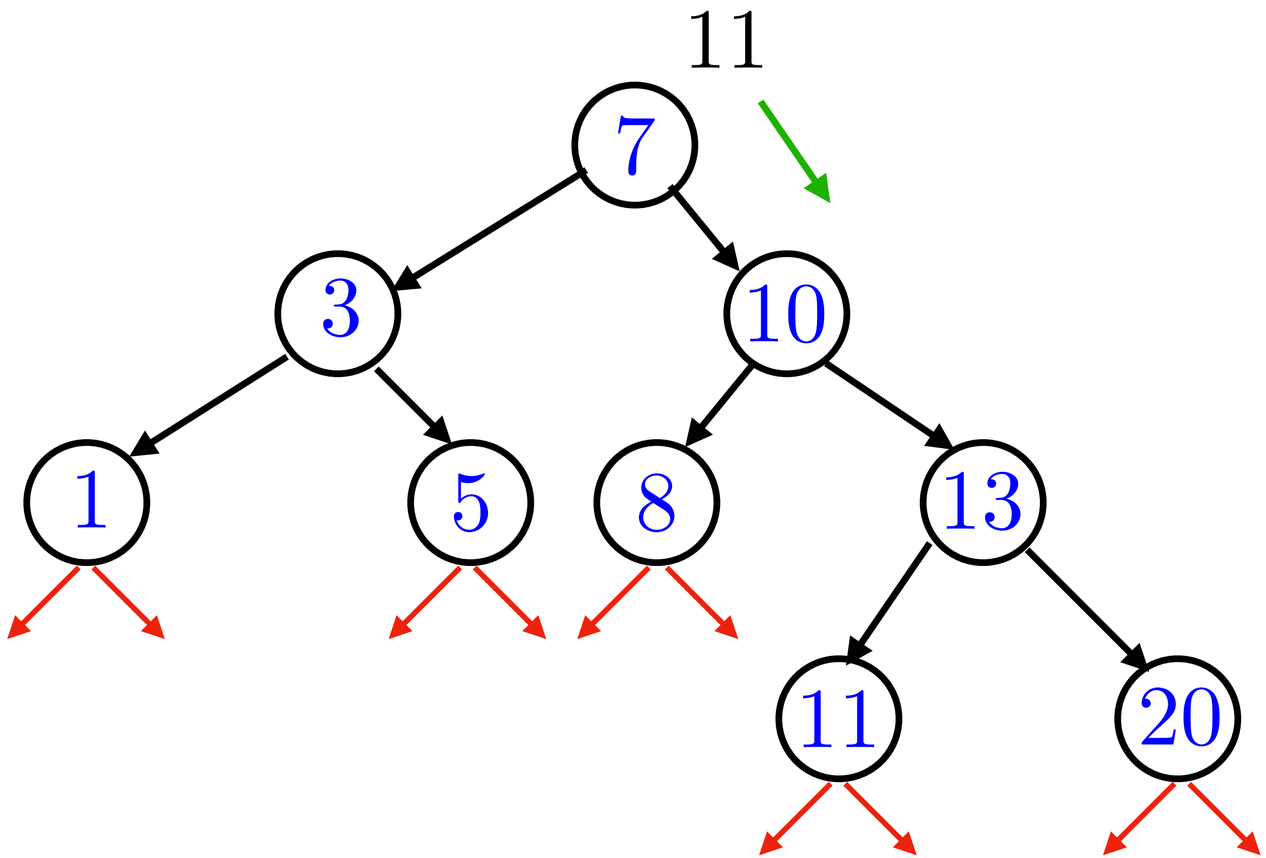
Contains

Let us first see how to check if a key is in a BST.

`contains(11)`

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



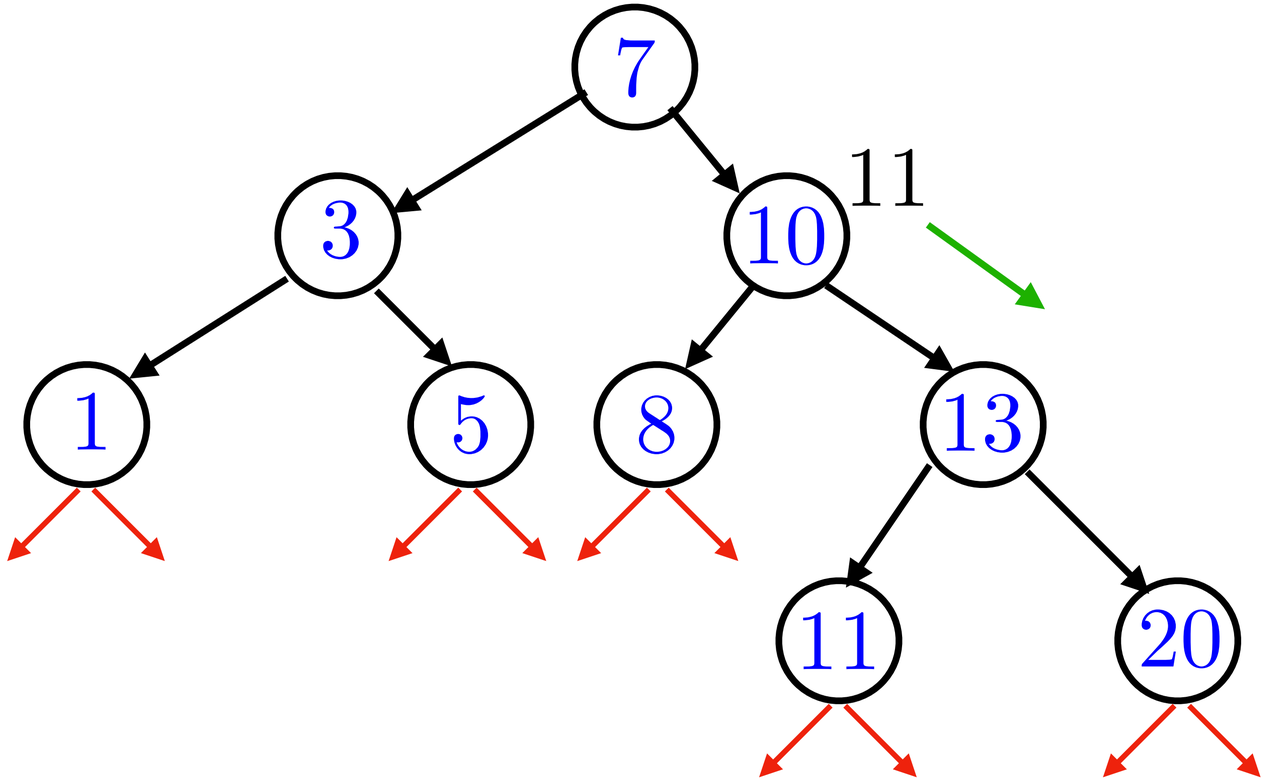
Contains

Let us first see how to check if a key is in a BST.

`contains(11)`

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



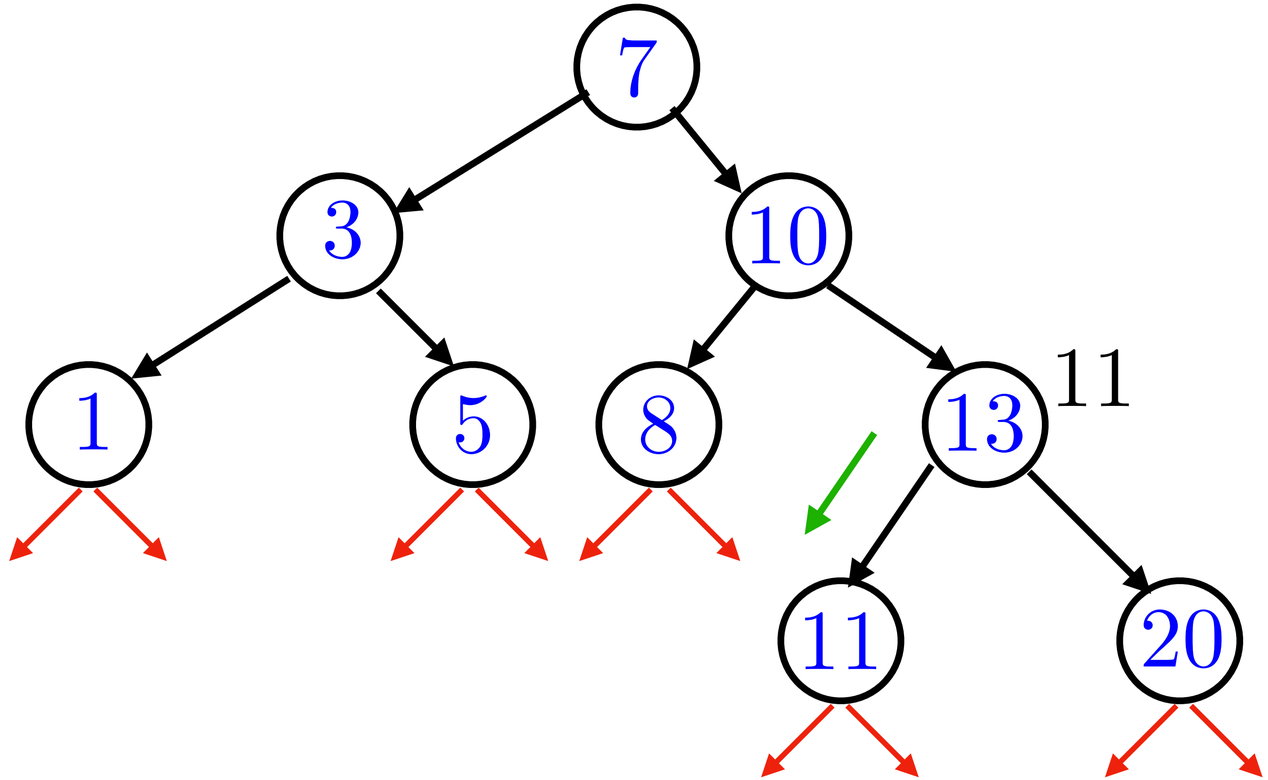
Contains

Let us first see how to check if a key is in a BST.

`contains`(11)

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



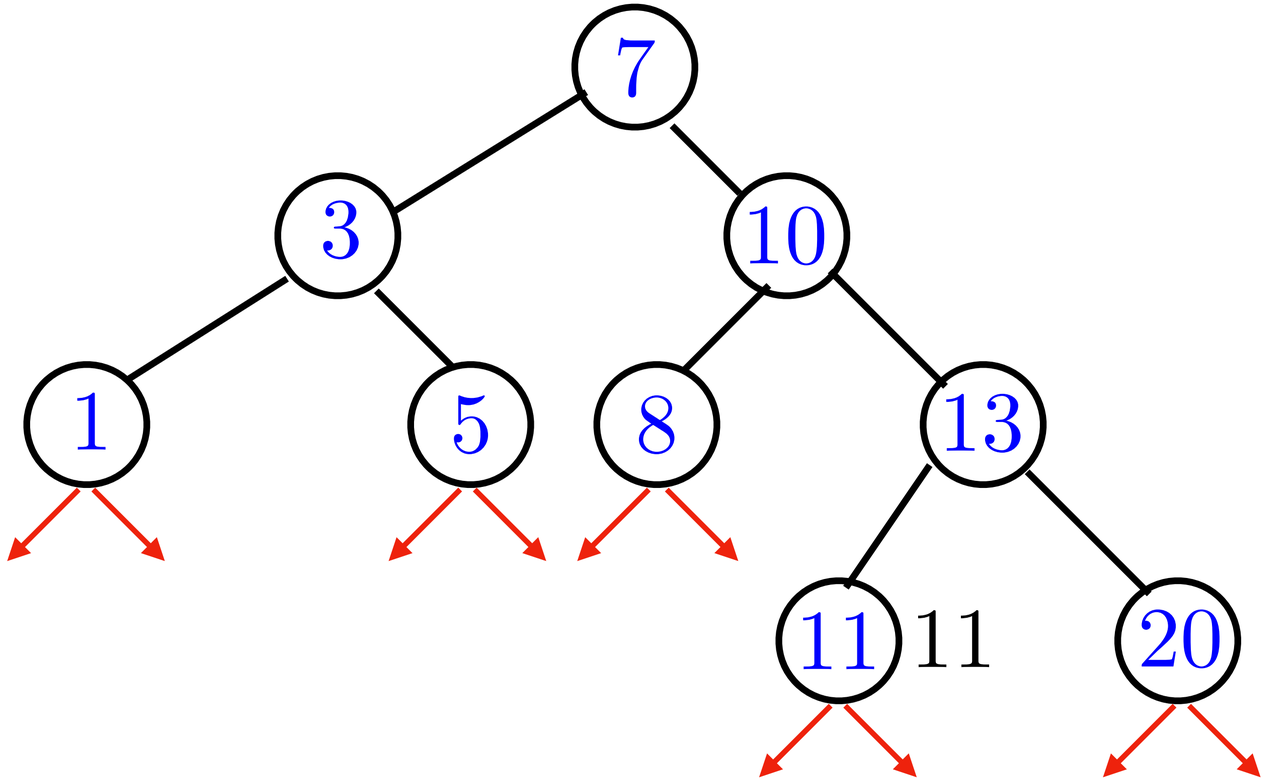
Contains

Let us first see how to check if a key is in a BST.

`contains(11)`

We start at the root and compare 11 to the key at the current vertex.

If it is larger, we go right, if it is smaller we go left.



Contains

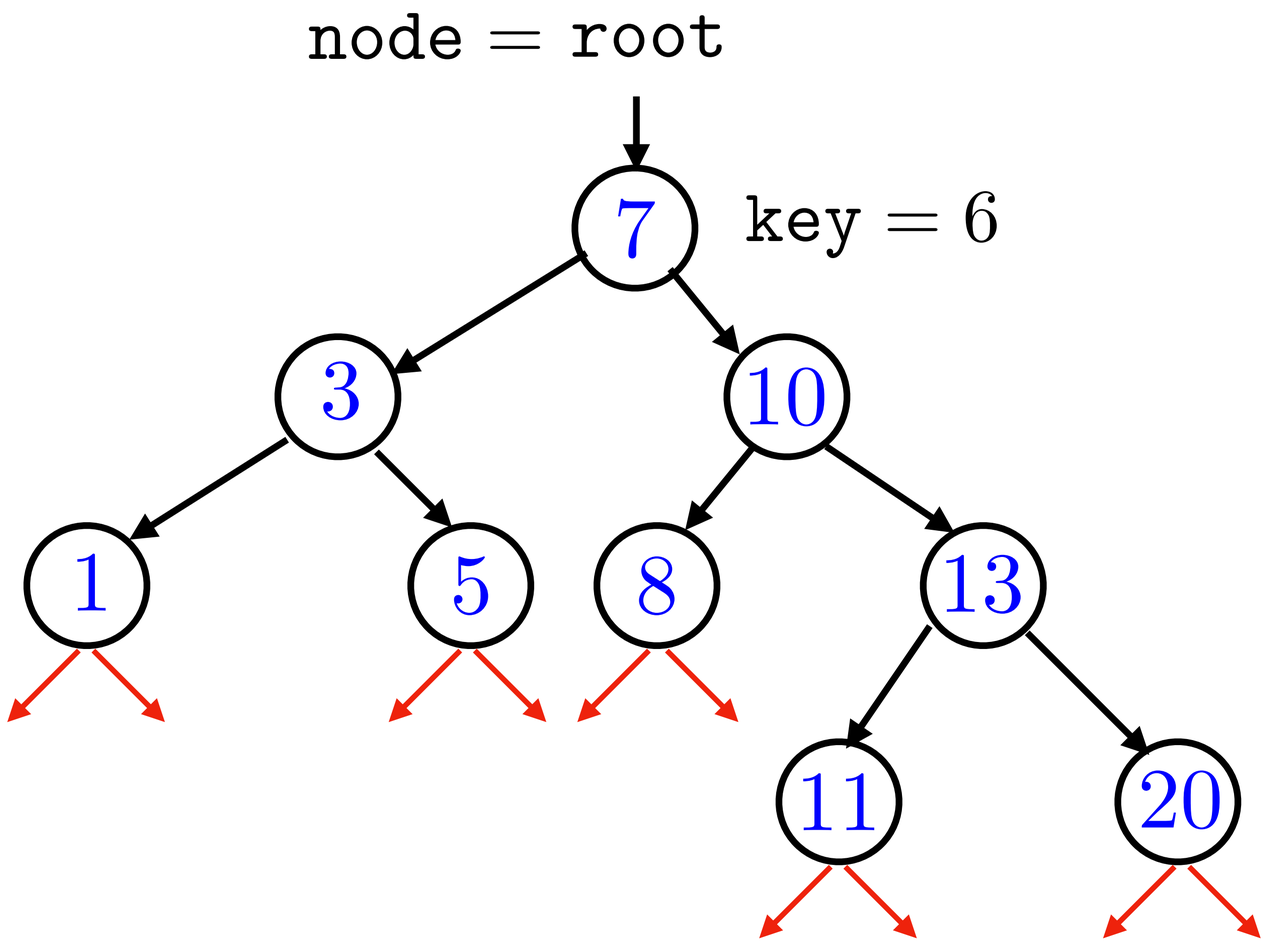
```
Node* contains(int k)
{
    Node* node = root;
    while(node != nullptr && node->key != k)
    {
        if(k < node->key)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    return node;
}
```

Contains (Miss)

Let's look at another example with a key not in the database.

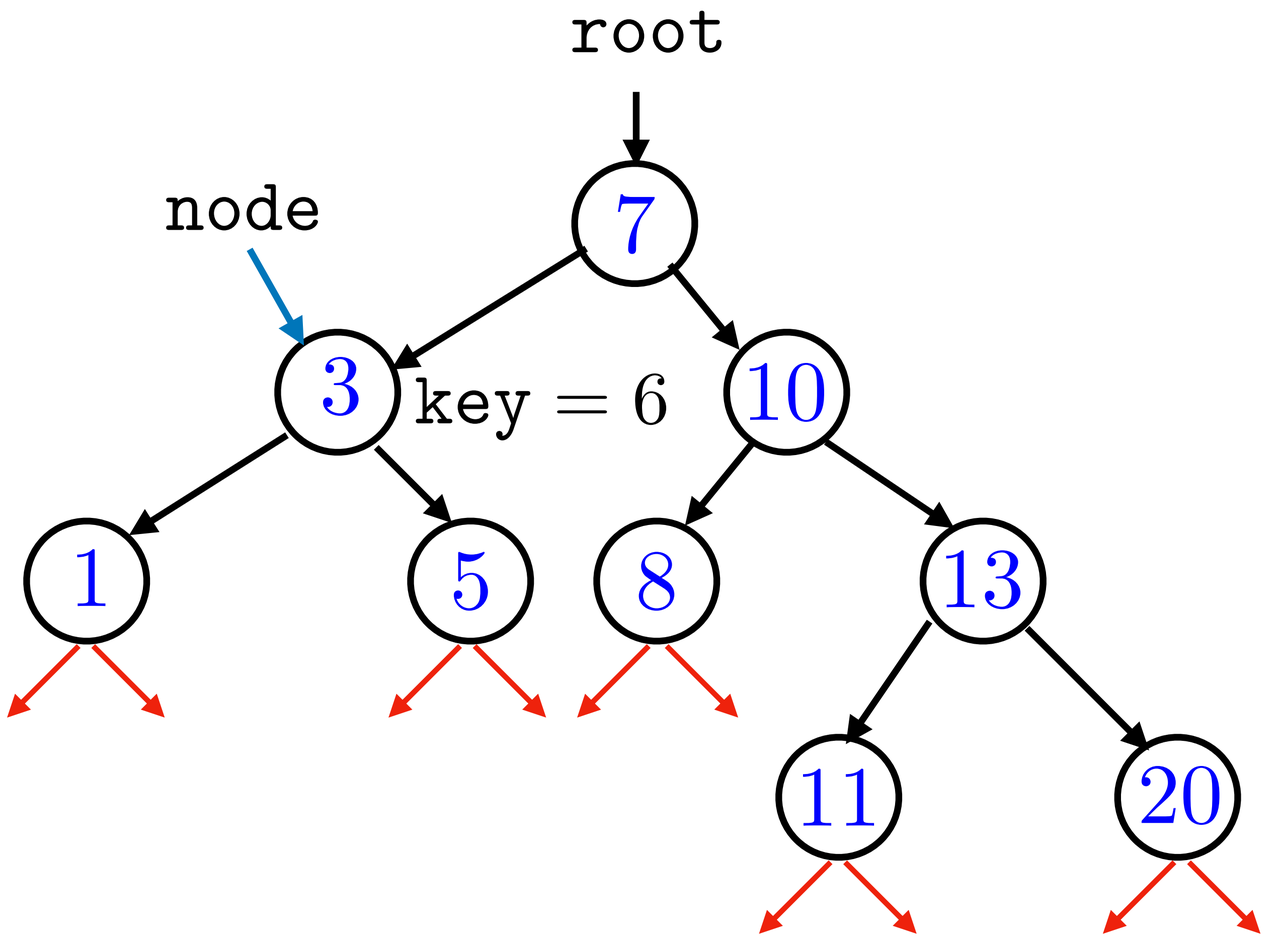
$$\texttt{contains}(6)$$

```
Node* contains(int k)
{
    Node* node = root;
    while(node != nullptr && node->key != k)
    {
        if(k < node->key)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    return node;
}
```



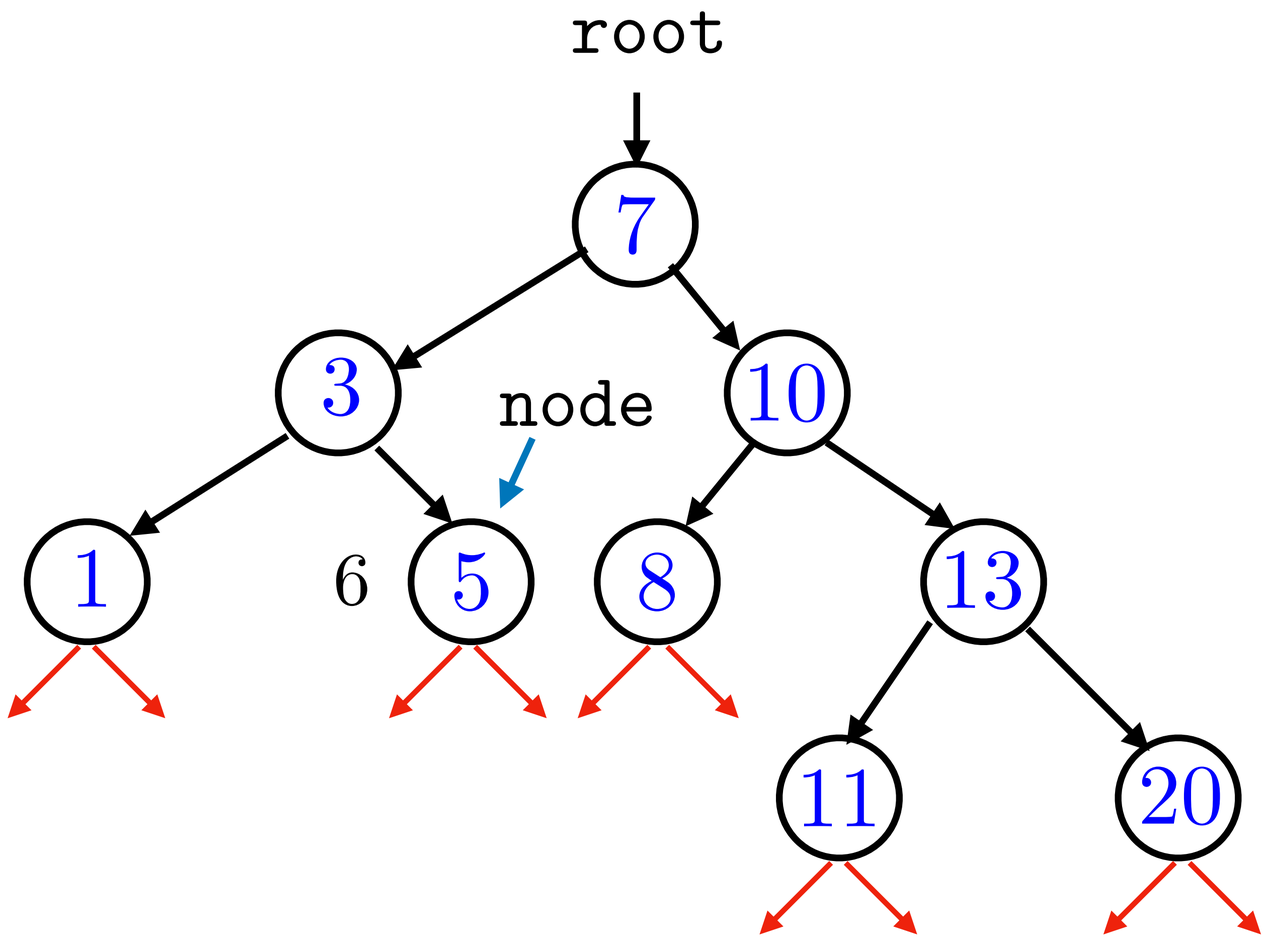
contains(6)

```
Node* contains(int k)
{
    Node* node = root;
    while(node != nullptr && node->key != k)
    {
        if(k < node->key)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    return node;
}
```



contains(6)

```
Node* contains(int k)
{
    Node* node = root;
    while(node != nullptr && node->key != k)
    {
        if(k < node->key)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    return node;
}
```

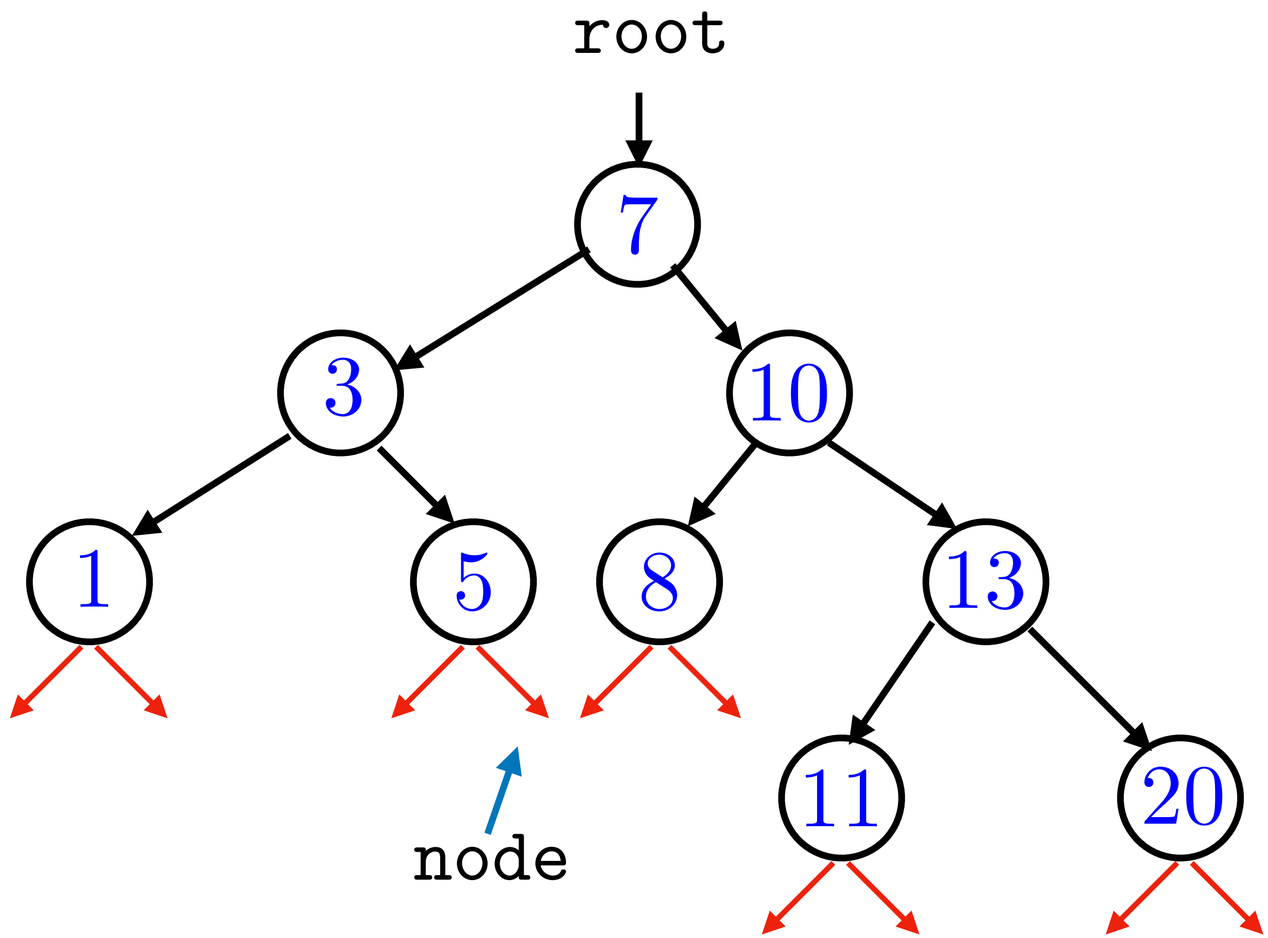


contains(6)

Now node = nullptr and so the while loop terminates.

It returns nullptr .

```
Node* contains(int k)
{
    Node* node = root;
    while(node != nullptr && node->key != k)
    {
        if(k < node->key)
        {
            node = node->left;
        }
        else
        {
            node = node->right;
        }
    }
    return node;
}
```

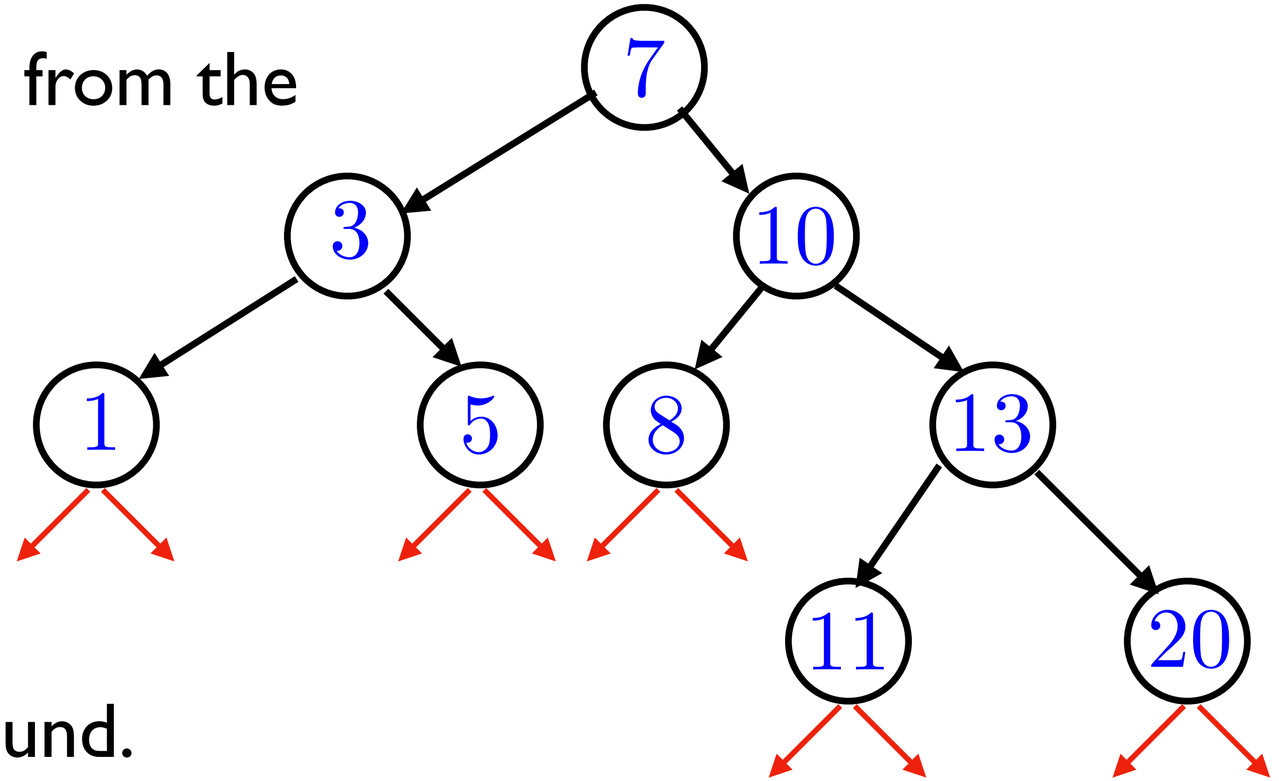


Contains: complexity

What is the complexity of `contains` ?

In the worst case, we visit every node from the root to the deepest leaf.

This takes time $\Omega(h)$ where h is the height of the tree.



The algorithm spends constant time at each node so $O(h)$ is an upper bound.

The complexity is $\Theta(h)$.

Insert

Insert is similar to a contains miss.

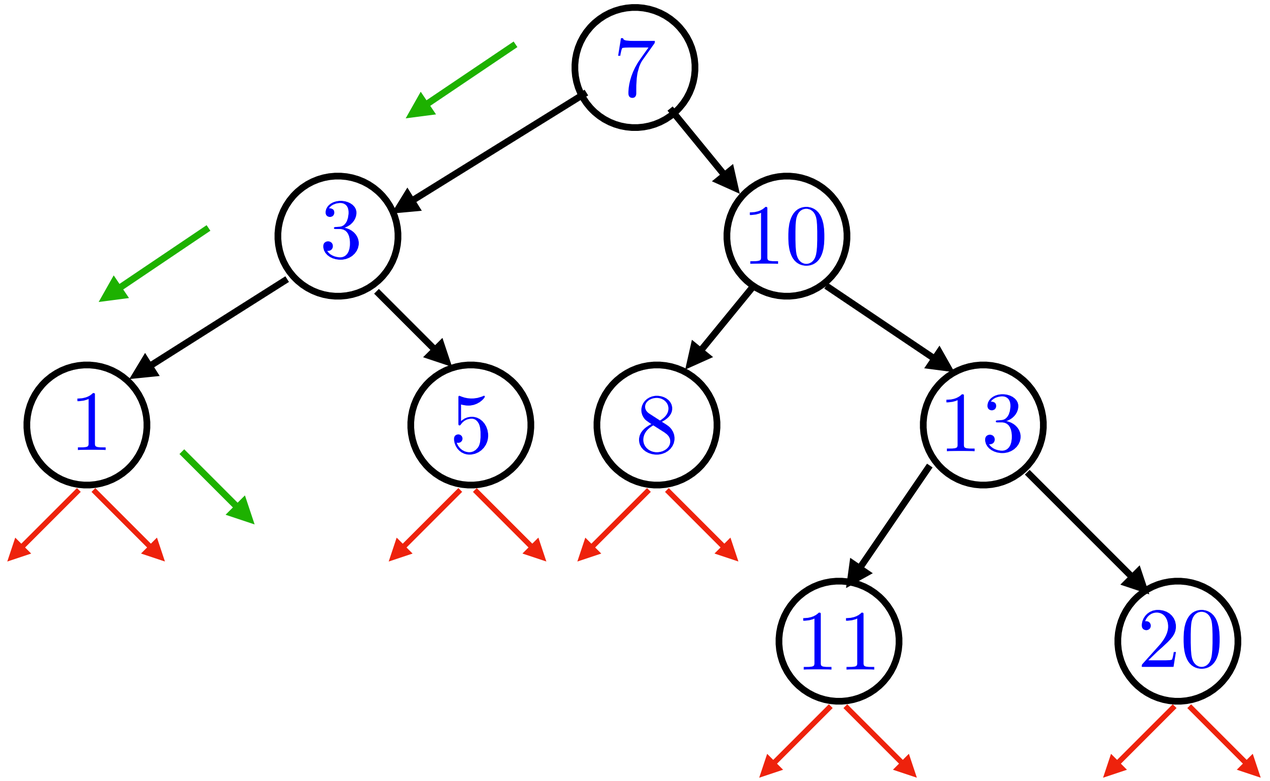
If a key is not in the set, `contains` ends up at a `nullptr` coming out of a leaf.

We instead make this `nullptr` point to a new node with the key to be inserted.

In particular, inserted nodes are always leaves.

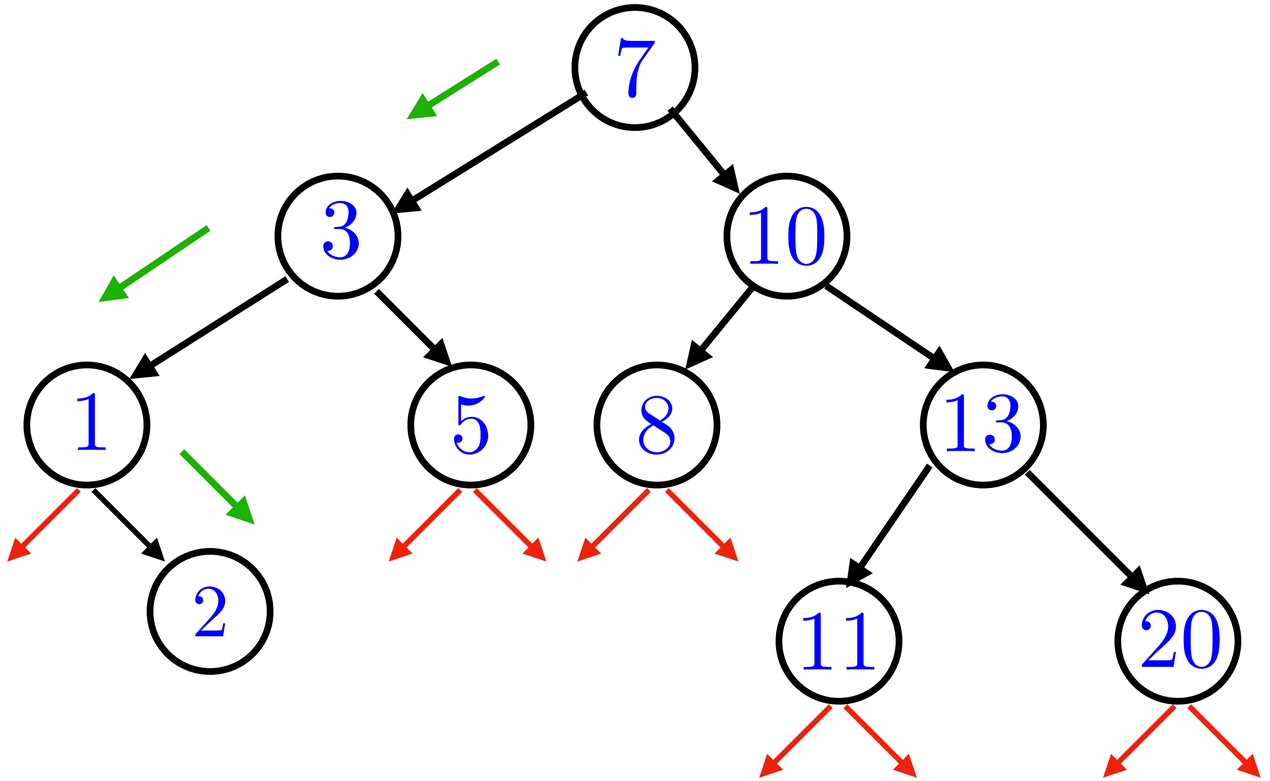
Insert: example

`insert(2)`



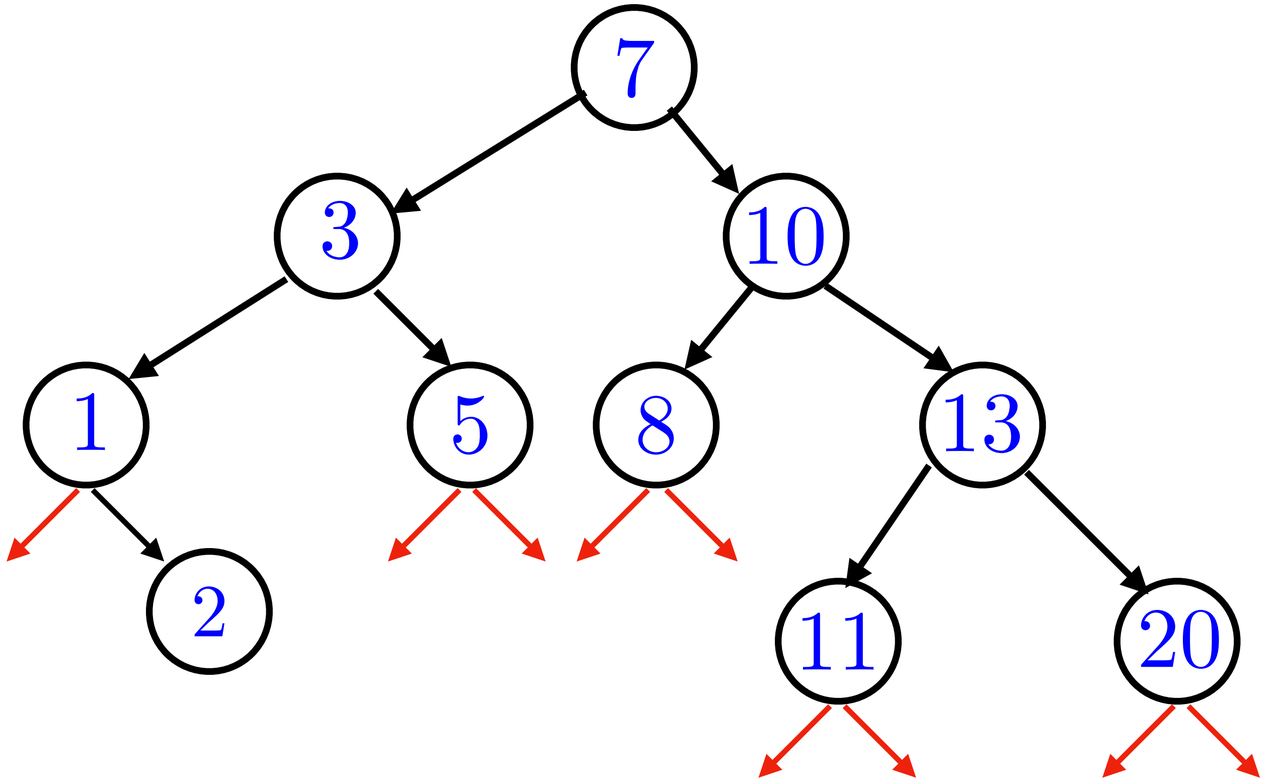
Insert: example

`insert(2)`



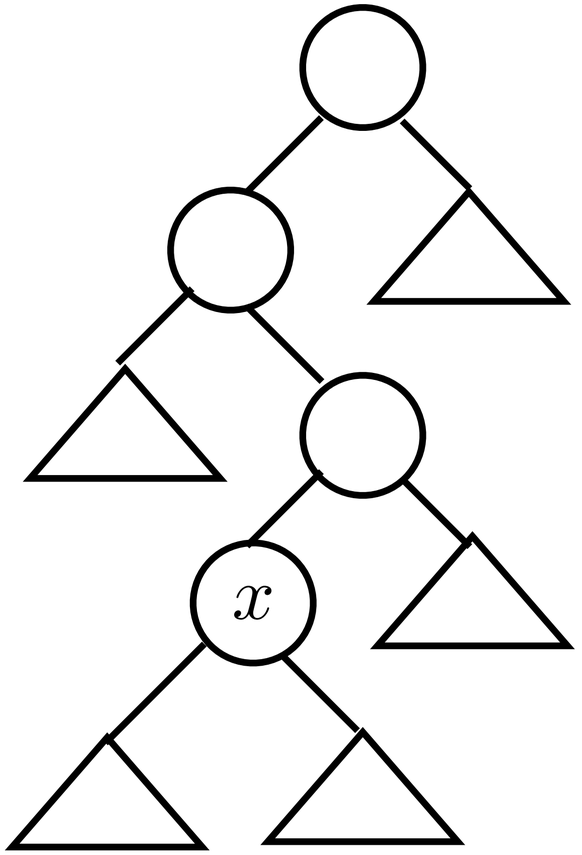
Insert: complexity

The worst case time for insertion is also $\Theta(h)$.



Successor

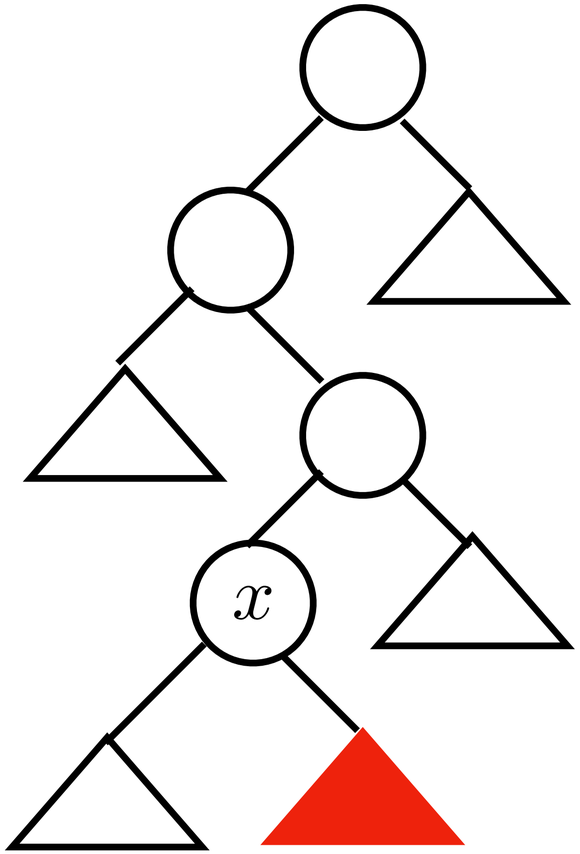
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

Successor

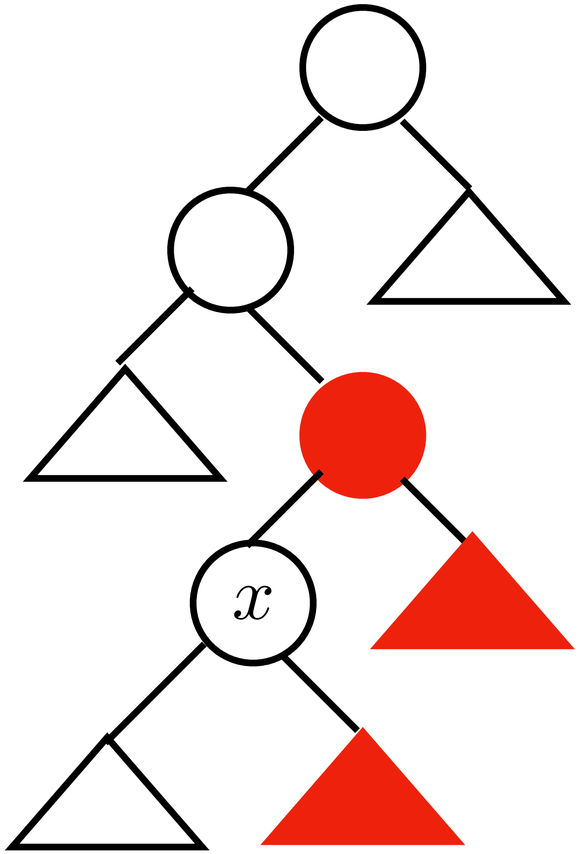
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

Successor

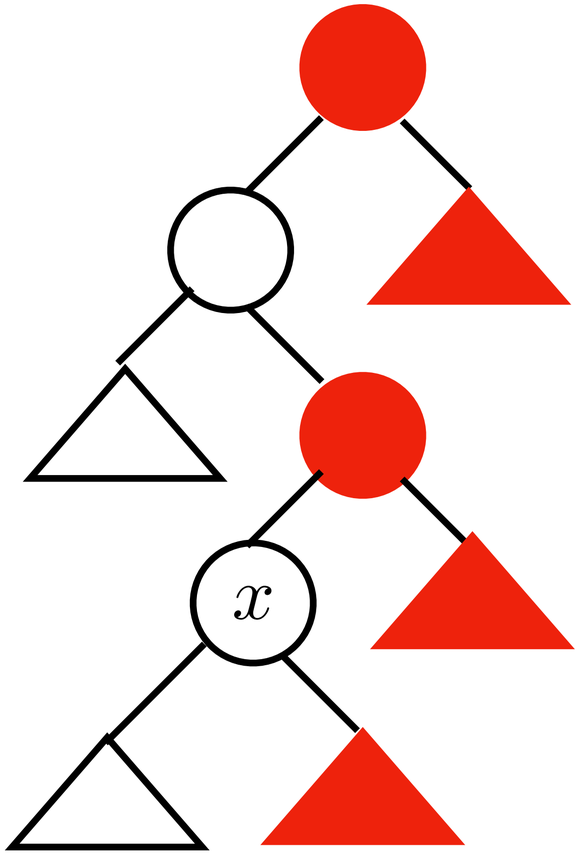
Before we get into successor let's think about the BST property some more.



Where are keys larger than x ?

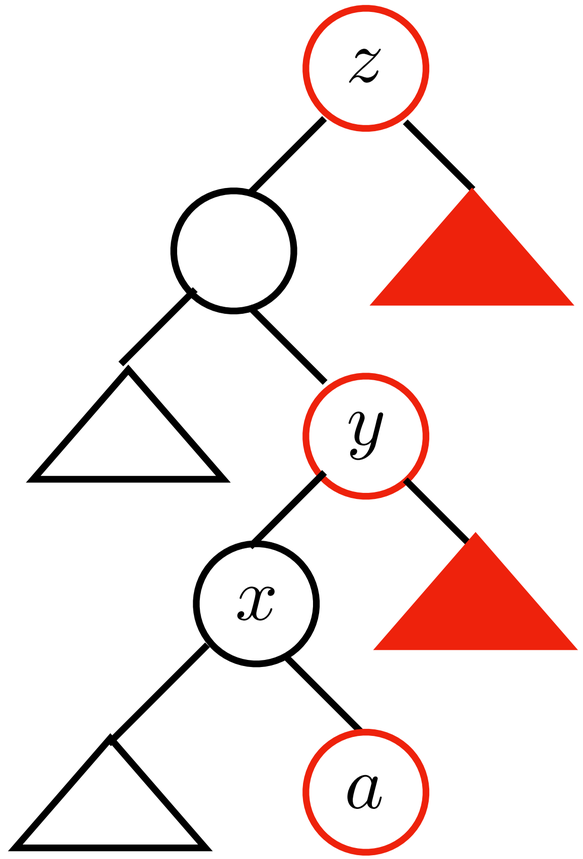
Successor

Before we get into successor let's think about the BST property some more.



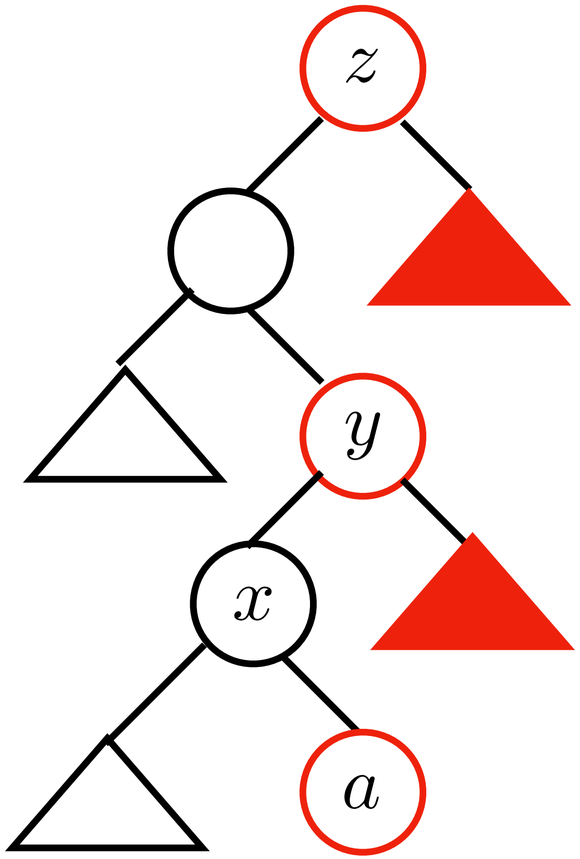
Where are keys larger than x ?

Now that we understand where the keys larger than x are, we move to find the successor of x .



What is the ordering of z, y, a ?

Now that we understand where the keys larger than x are, we move to find the successor of x .

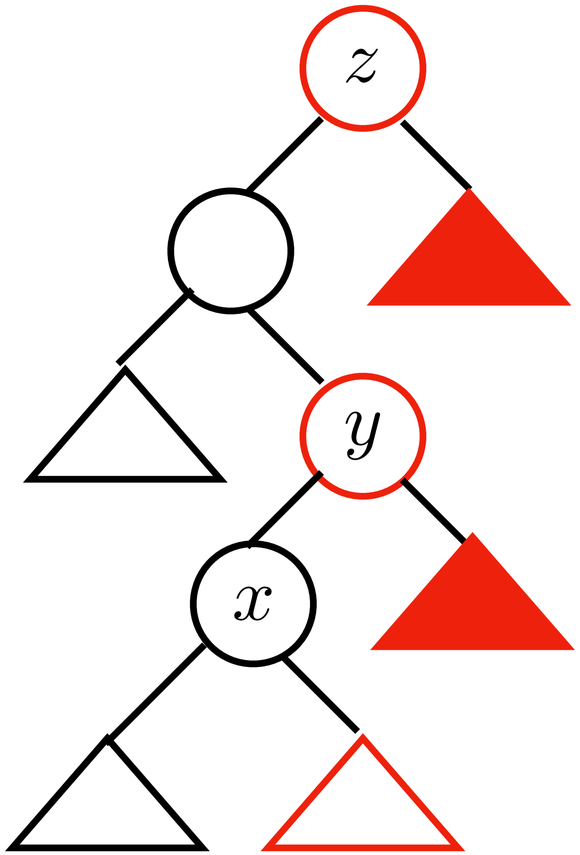


What is the ordering of z, y, a ?

We know that $a < y < z$.

To find the successor of x we should look in its right subtree first.

Successor: Case 1

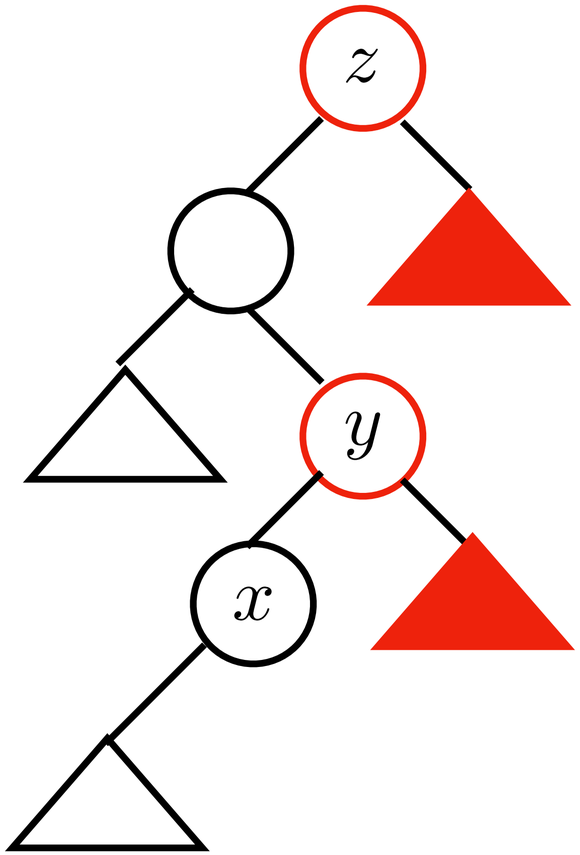


To find the successor of x we should look in its right subtree first.

Fact: If x has a right child, the successor of x is the minimum element of the right subtree of x.

The successor is found by always going left in the right subtree of x.

Case 2: no right child



What if x has no right child?

Fact: If x has no right child, the successor of x is the node where you last go left on the path from the root to x .

To implement successor we can remember the last left turn as we search for x in the tree.

Successor: complexity

To find the successor of x, we first find x in the tree.

The complexity is at least $\Omega(h)$, that of `contains`.

What work do we do beyond that of `contains`?

In the case x has a right child, we find the minimum in the right subtree. This can also be done in $O(h)$ time.

`successor` also has complexity $\Theta(h)$.

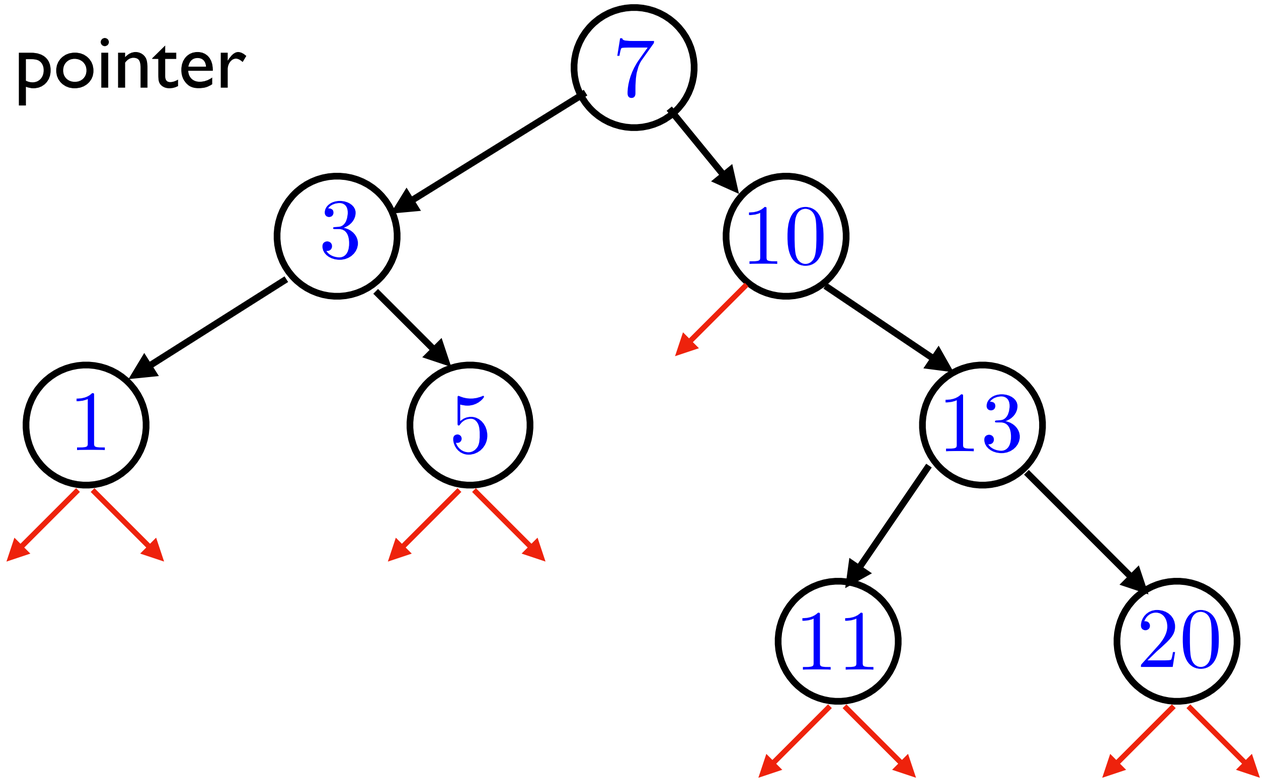
Remove

Let's now see how to remove a node. This is trickier than the others.

First some easy cases. Say the node to delete is a leaf.

We delete the leaf, and change the pointer from its parent to a `nullptr` .

Example: delete key 5 .



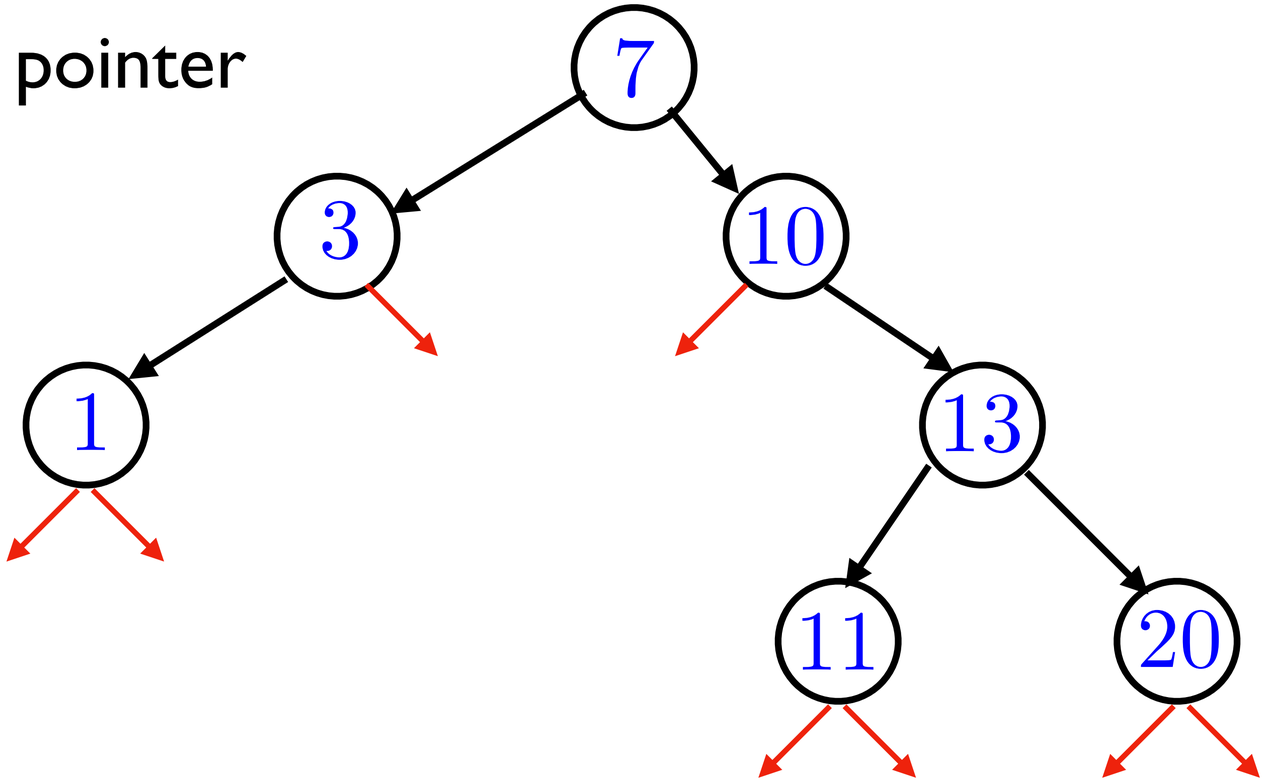
Remove: leaf

Let's now see how to remove a node. This is trickier than the others.

First some easy cases. Say the node to delete is a leaf.

We delete the leaf, and change the pointer from its parent to a `nullptr` .

Example: delete key 5 .

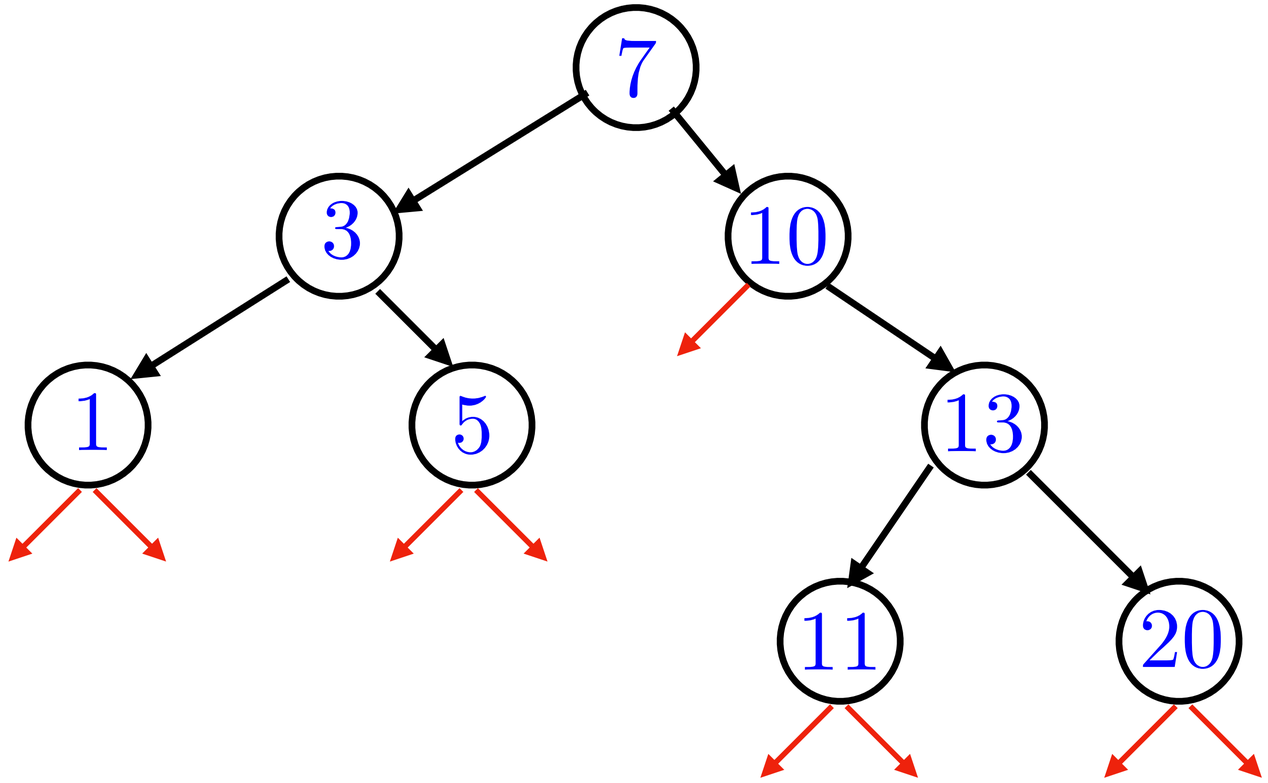


Remove: one child

Similar easy case: Node to delete just has one child.

Then the child takes the place of the node to delete.

Example: delete key 10.



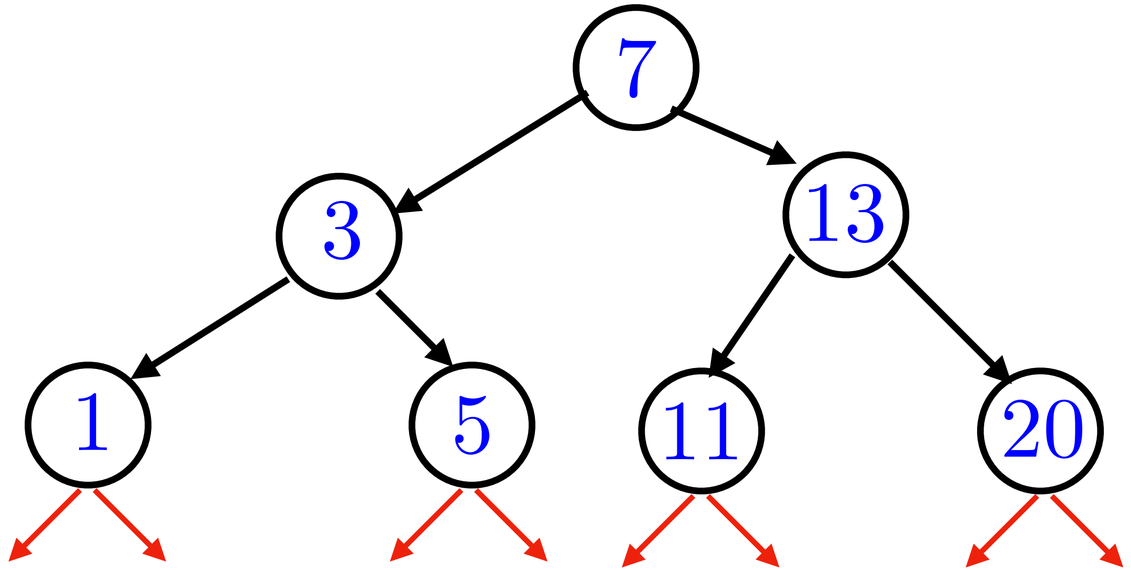
Remove: one child

Similar easy case: Node to delete just has one child.

Then the child takes the place of the node to delete.

Example: delete key 10.

Note this preserves the BST property.



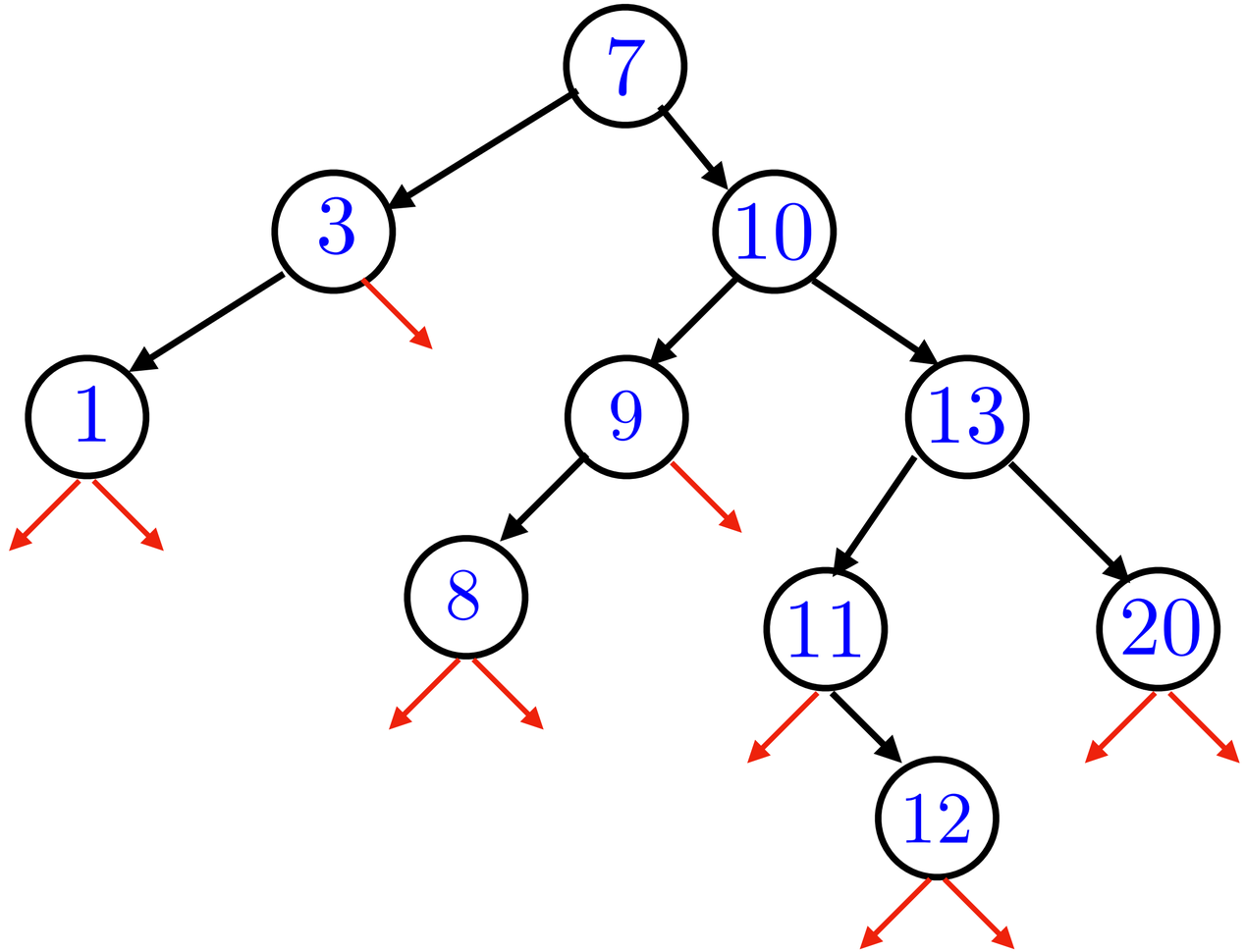
Remove: both children

The interesting case is where the node to delete has both children.

Let's say we want to delete key 10.

We again want to replace 10 by some other key in the set.

What key can we replace it with that causes minimal change?

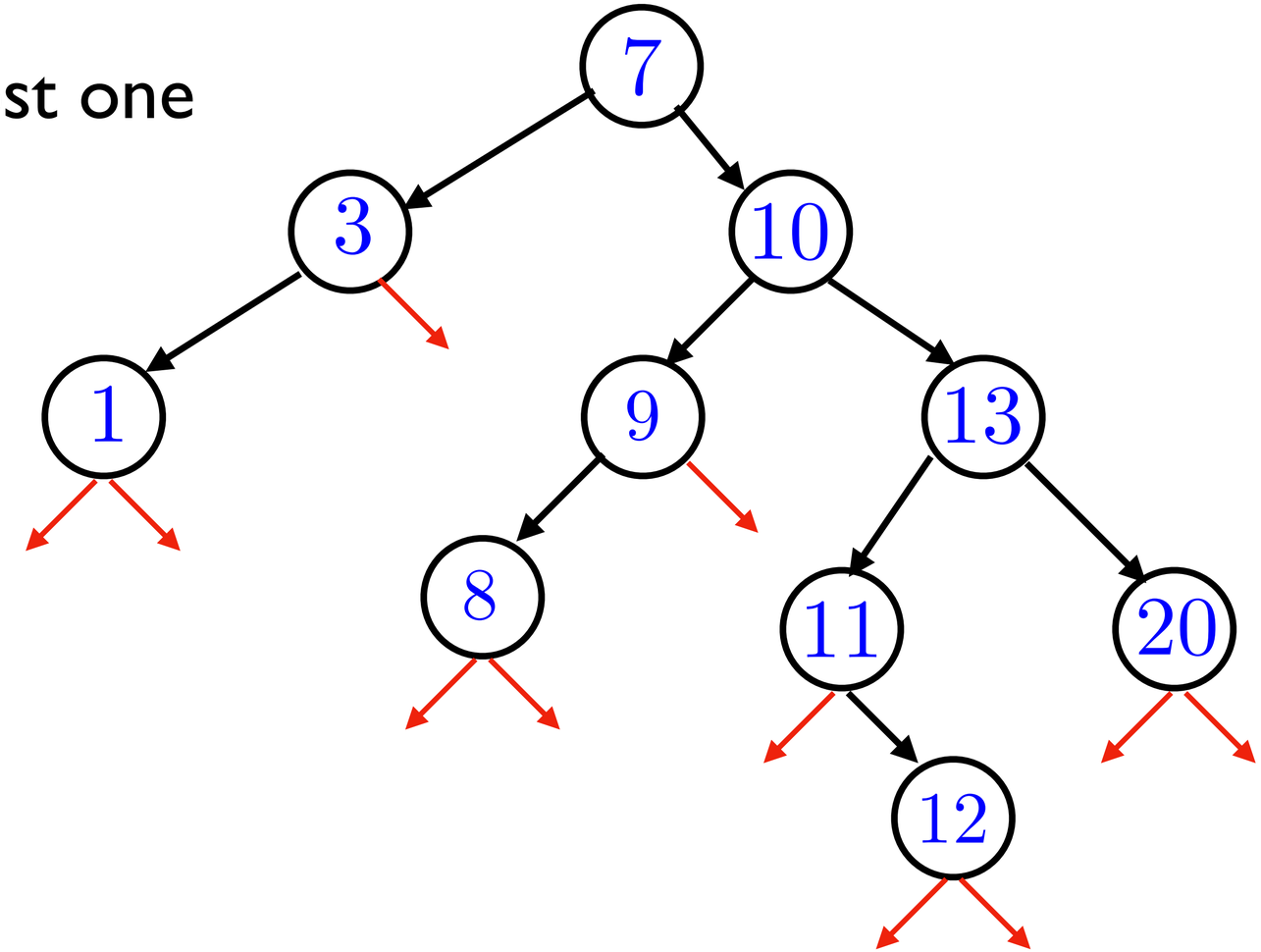


Remove: both children

What key can replace 10 and cause minimal change?

Idea: Replace by a node with at most one child.

What is a node with at most one child whose key fits in the position of 10?



Remove: both children

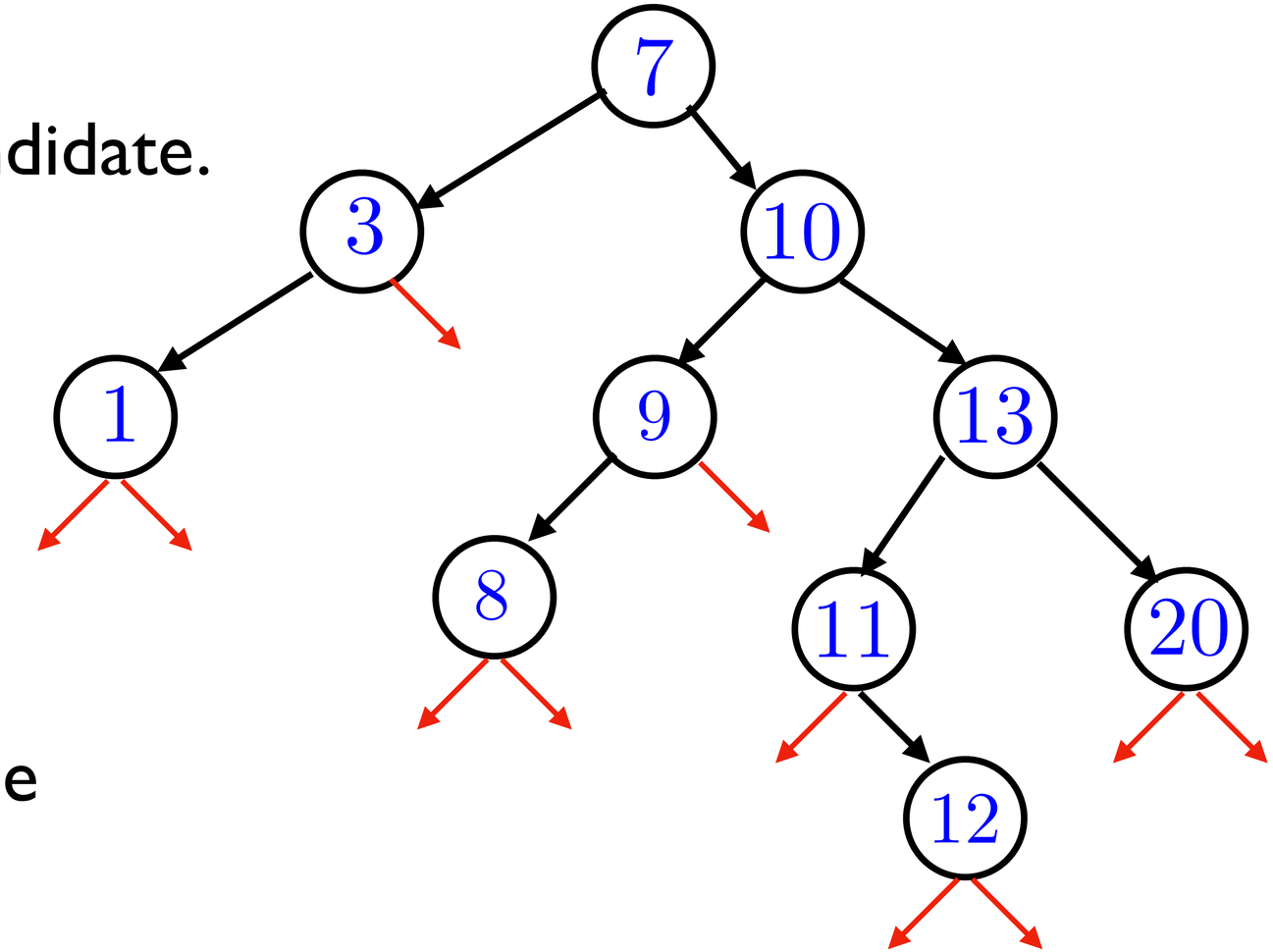
What is a node with at most one child whose key fits in the position of 10?

The successor of 10 is a good candidate.

It has no left child.

It is bigger than everything in the left subtree of 10.

It is less than everything else in the right subtree of 10.



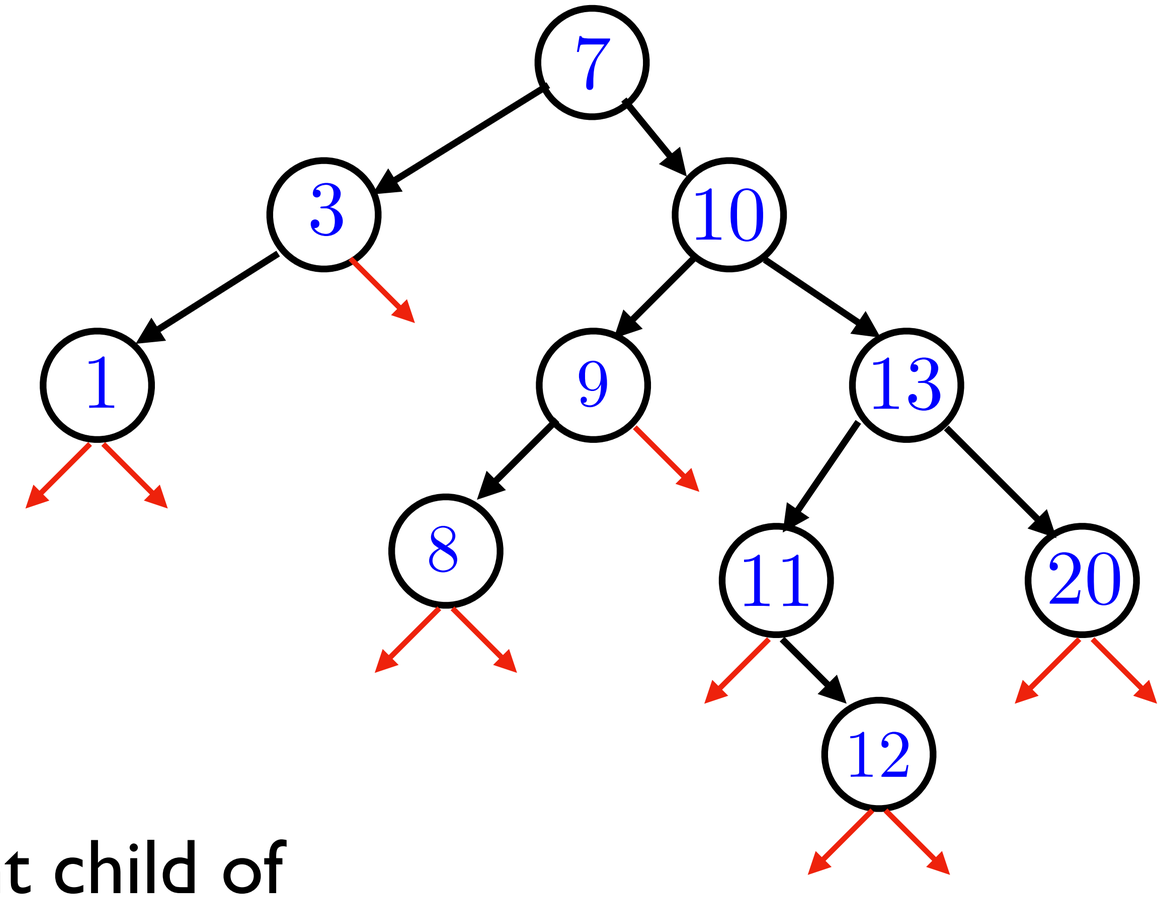
Remove: both children

Algorithm idea:

1) Find successor s of key to delete.

2) Make parent of s point to right child of s .

3) Make parent of node to delete point to s , and update children of s with children of node to delete.



Careful with special case: successor is right child of node to delete.

Complexity of remove

Complexity of remove is essentially:

1) Finding node to delete (`contains`).

2) Successor operation.

3) Constant number of pointer changes.

Complexity is again $\Theta(h)$, where h is the height of the tree.

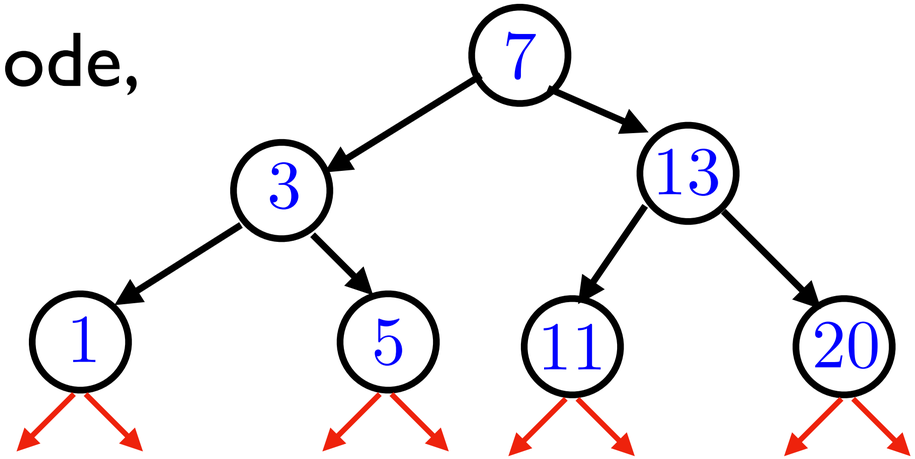
Inorder traversal

Inorder traversal

Another natural operation we may want from a BST is to extract all the keys in order.

This can be done by an inorder traversal of the tree.

To print keys in order, we want to first print
all keys in a node's left subtree, then print the node,
then print all keys in the trees right subtree.



Inorder traversal

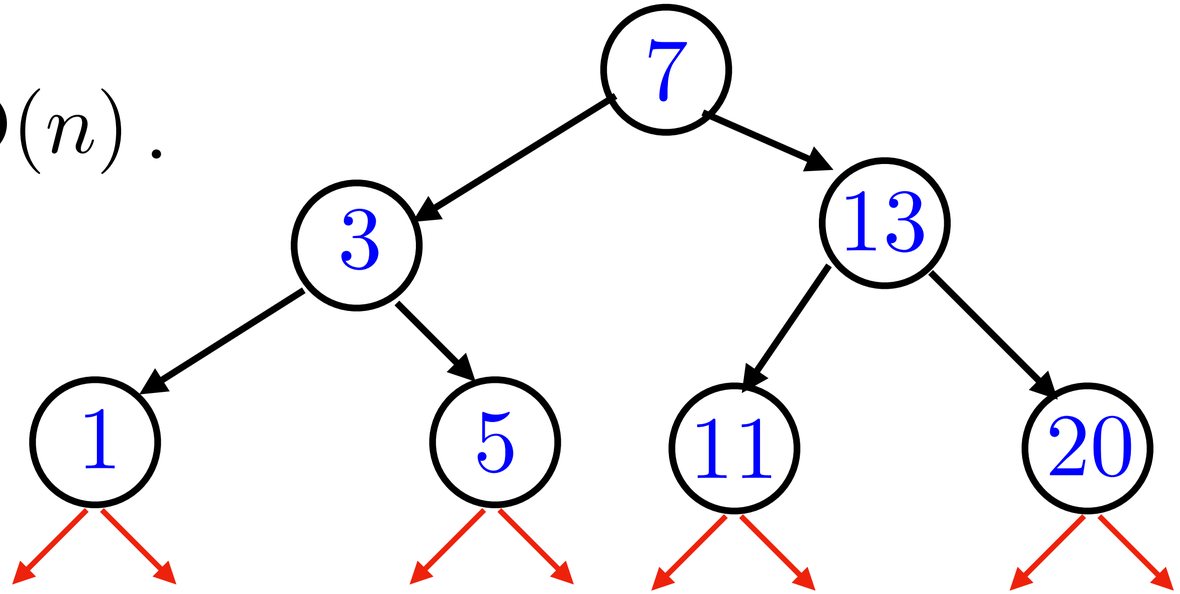
To print keys in order, we want to first print all keys in a node's left subtree, then print the node, then print all keys in the trees right subtree.

This gives a simple recursive implementation of inorder traversal.

```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

```
void print(Node* node){
    if(node == nullptr)
        return;
    print(node->left);
    std::cout << node->key << '\n';
    print(node->right);
}
```

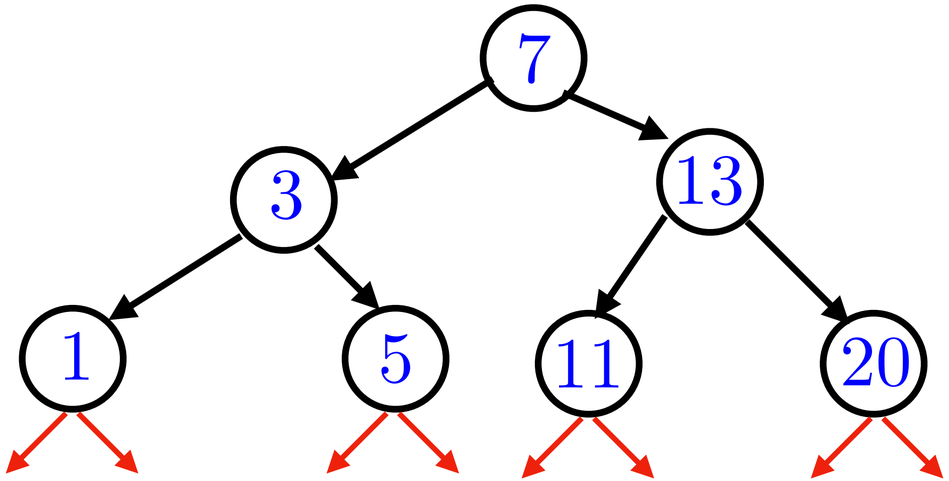
The complexity is $\Theta(n)$.



Preorder and Postorder traversal

```
void preorder(Node* node){
    if(node == nullptr)
        return;
    std::cout << node->key << '\n';
    preorder(node->left);
    preorder(node->right);
}
```

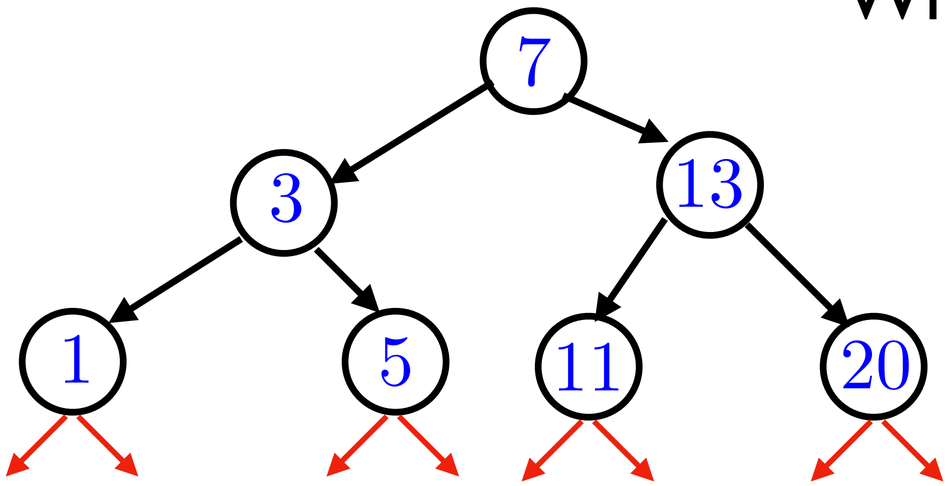
```
void postorder(Node* node){
    if(node == nullptr)
        return;
    postorder(node->left);
    postorder(node->right);
    std::cout << node->key << '\n';
}
```



BST Destructor

In the destructor for a BST we want to free the memory allocated to each node.

We traverse through the tree deleting pointers to the nodes.



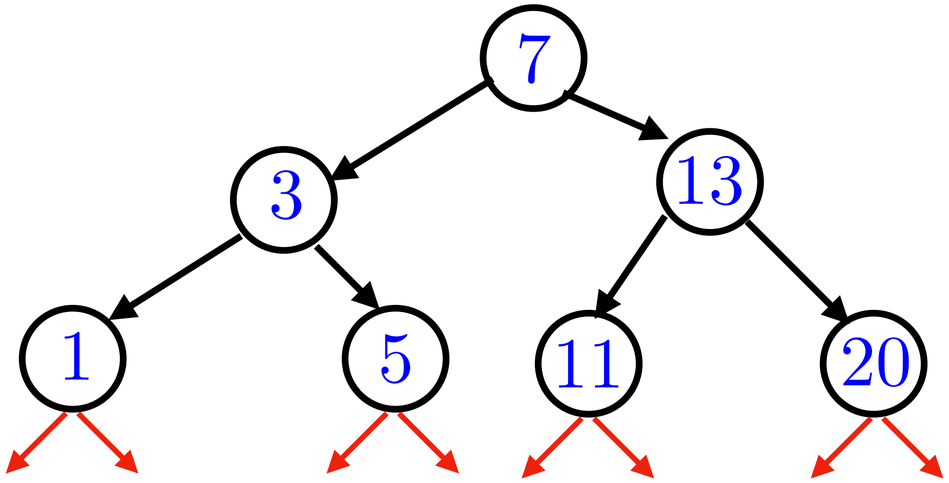
What kind of traversal should we use for this?

We don't want to delete the node with key 13 before deleting its children.

BST Destructor

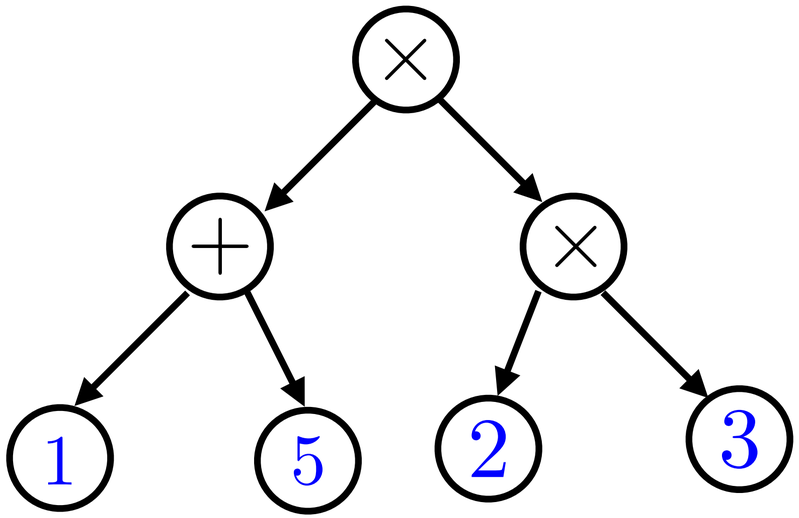
In the destructor for a BST we want to free the memory allocated to each node.

We traverse through the tree deleting pointers to the nodes.




```
void postorder(Node* node){
    if(node == nullptr)
        return;
    postorder(node->left);
    postorder(node->right);
    std::cout << node->key << '\n';
}
```

Expression Tree



inorder: $(1 + 5) \times (2 \times 3)$

preorder: $\times + 15 \times 23$

Polish notation. No parens needed.

postorder: $15 + 23 \times \times$

Good for stack calculator.

Height of a BST

Height of a BST

All of our operations have complexity $\Theta(h)$, where h is the height of the tree.

It is key to the performance to understand the height of the tree.

When we insert n elements what is the maximum height of the tree?

Worst case

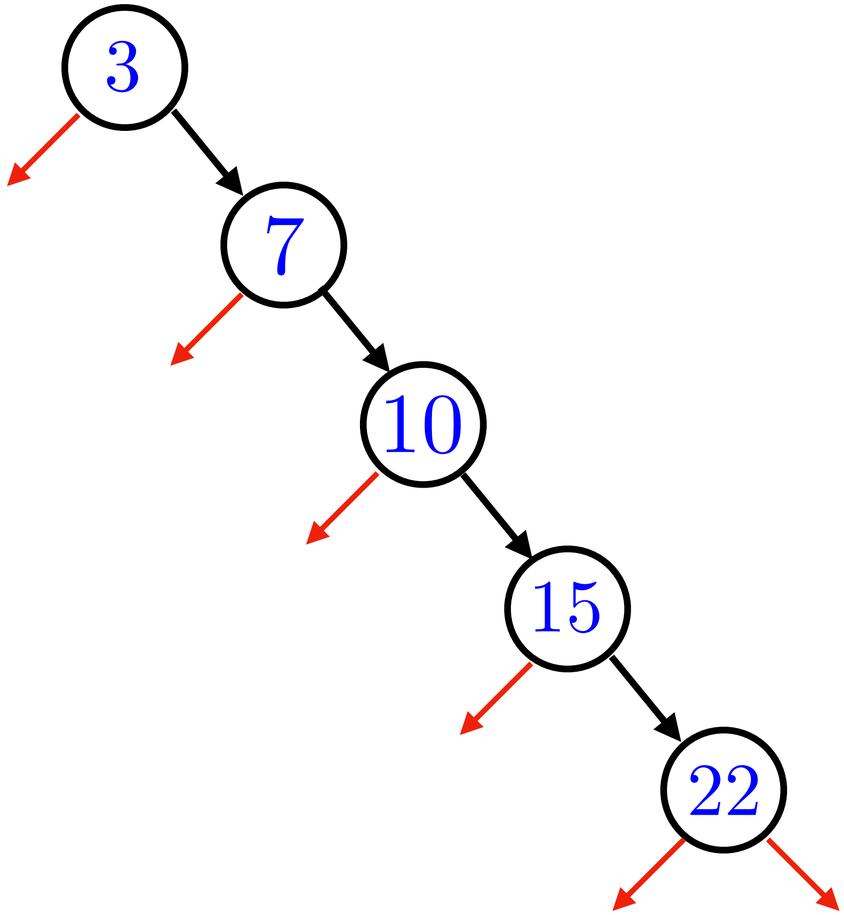
Say that we insert elements in the order 3, 7, 10, 15, 22.

In this case every vertex has at most one child.

The height is $n - 1$ which is as large as possible.

The worst case is when the elements are inserted in sorted order!

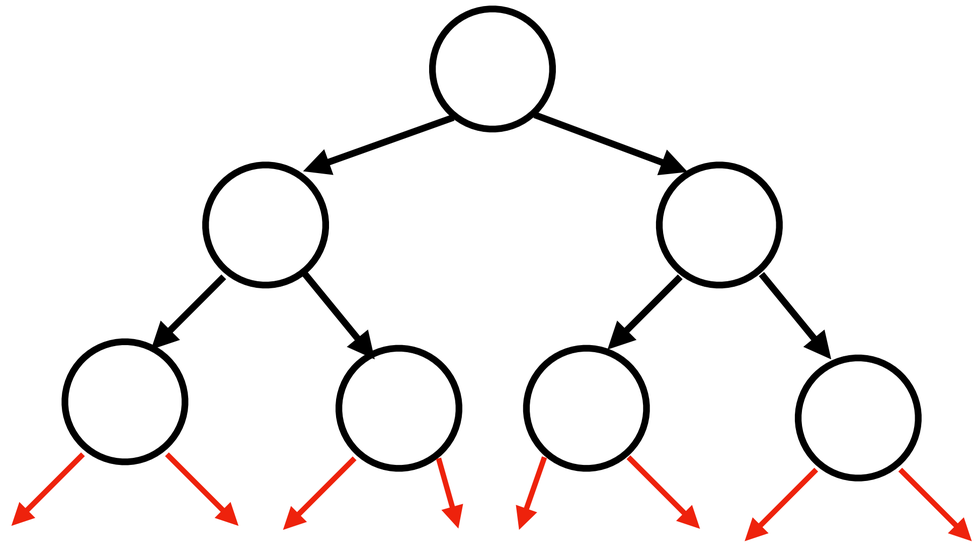
BSTs do not perform well in this scenario.



Best case

What is the best case height of the tree?

In the best case, for every vertex the height of its left and right subtrees is the same.



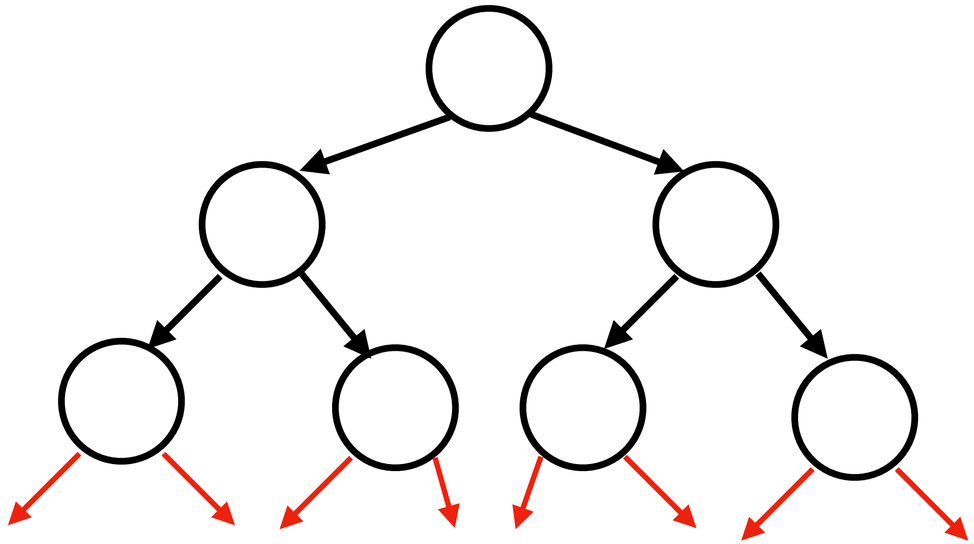
This is known as a full binary tree and is only possible if $n = 2^d - 1$.

We always have $n \leq 2^{h+1} - 1$ and so $h \geq \log(n+1) - 1$.

Our operations will take time at least $\Omega(\log n)$.

Best case

Say that we have keys 3, 7, 9, 10, 12, 15, 22.



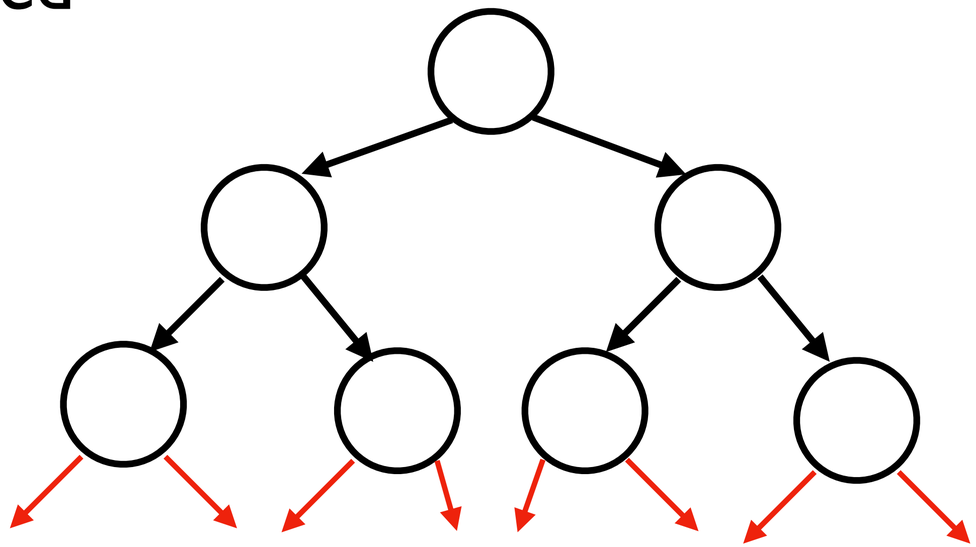
In what order should we insert these keys in to obtain a full binary tree?

In a full binary tree, for any vertex the number of nodes in the left subtree and right subtree is the same.

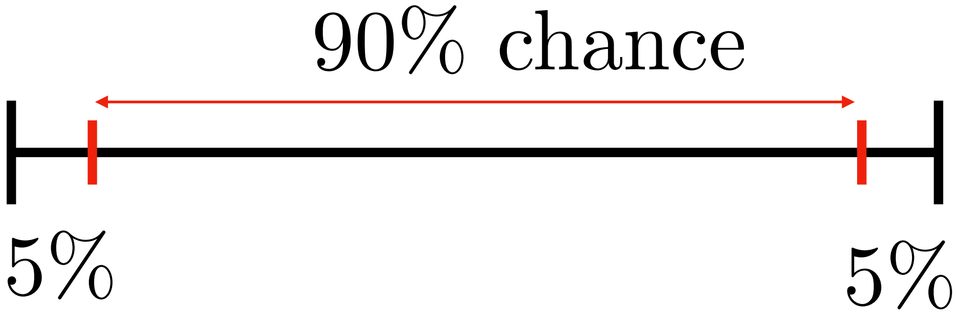
If the root has key x how many nodes will be in its left subtree?

Random case

If the keys come in a random order, the expected height of a BST is $O(\log n)$.



Intuition: we do not expect the first key to be the median, but with good probability it will be near the middle.



See Theorem 12.4 of [CLRS] for a proof.

AVL Trees

Balanced Binary Trees

It is desirable to maintain a tree height of $O(\log n)$ when the tree has n keys.

We now look at a way of actively ensuring this property.

Whenever we insert or remove a key, if the tree becomes too unbalanced we change the structure to fix it.

There are several (related) techniques to do this: AVL trees, 2-3 trees, AA trees, red-black trees.

These achieve $O(\log n)$ worst-case time for all our operations.

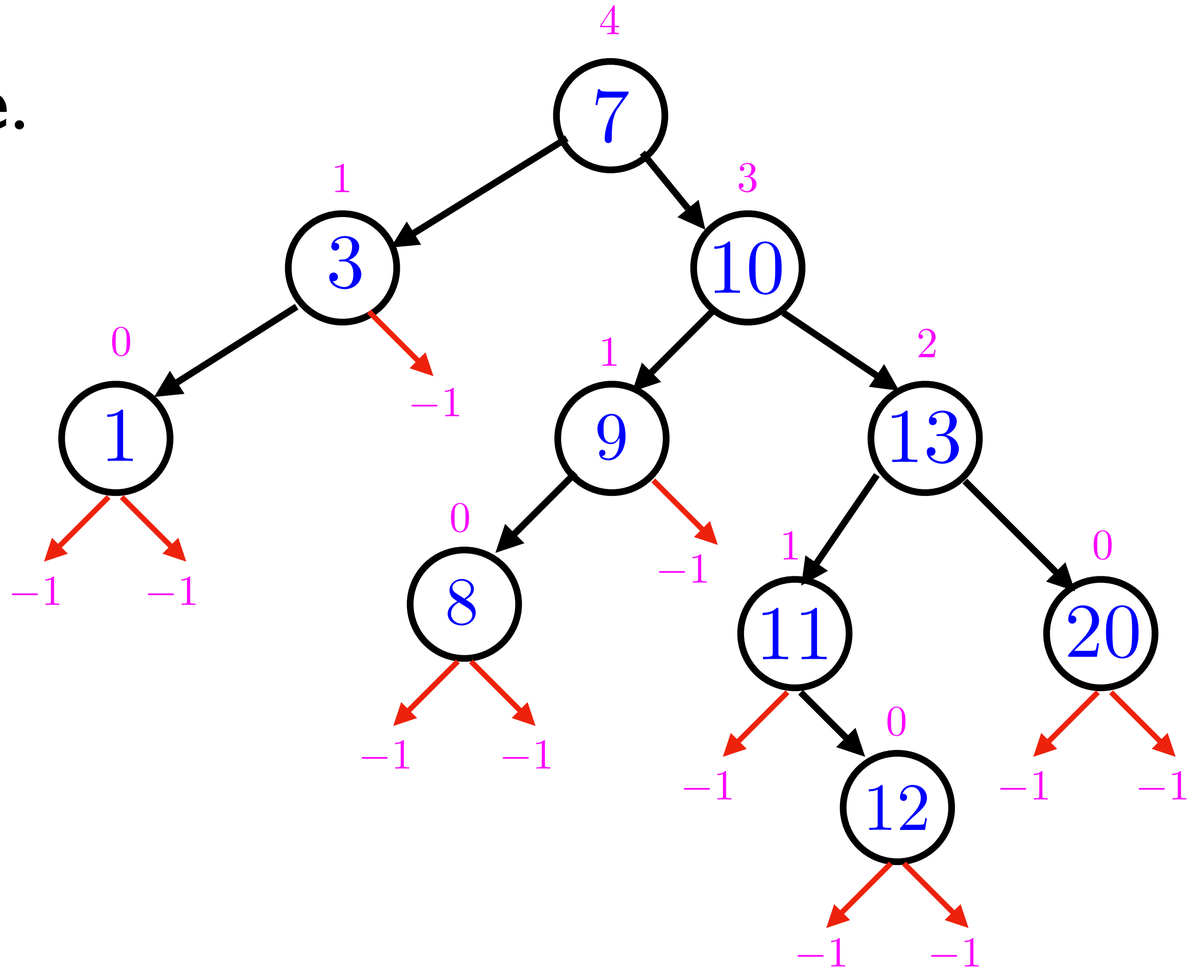
AVL trees

We will discuss AVL trees, named after Adelson-Velsky and Landis.

We keep track of the height of each node.

The height of a node is the maximum height of its children plus one.

To avoid a special case, `nullptr` points to a node of height -1.



AVL trees

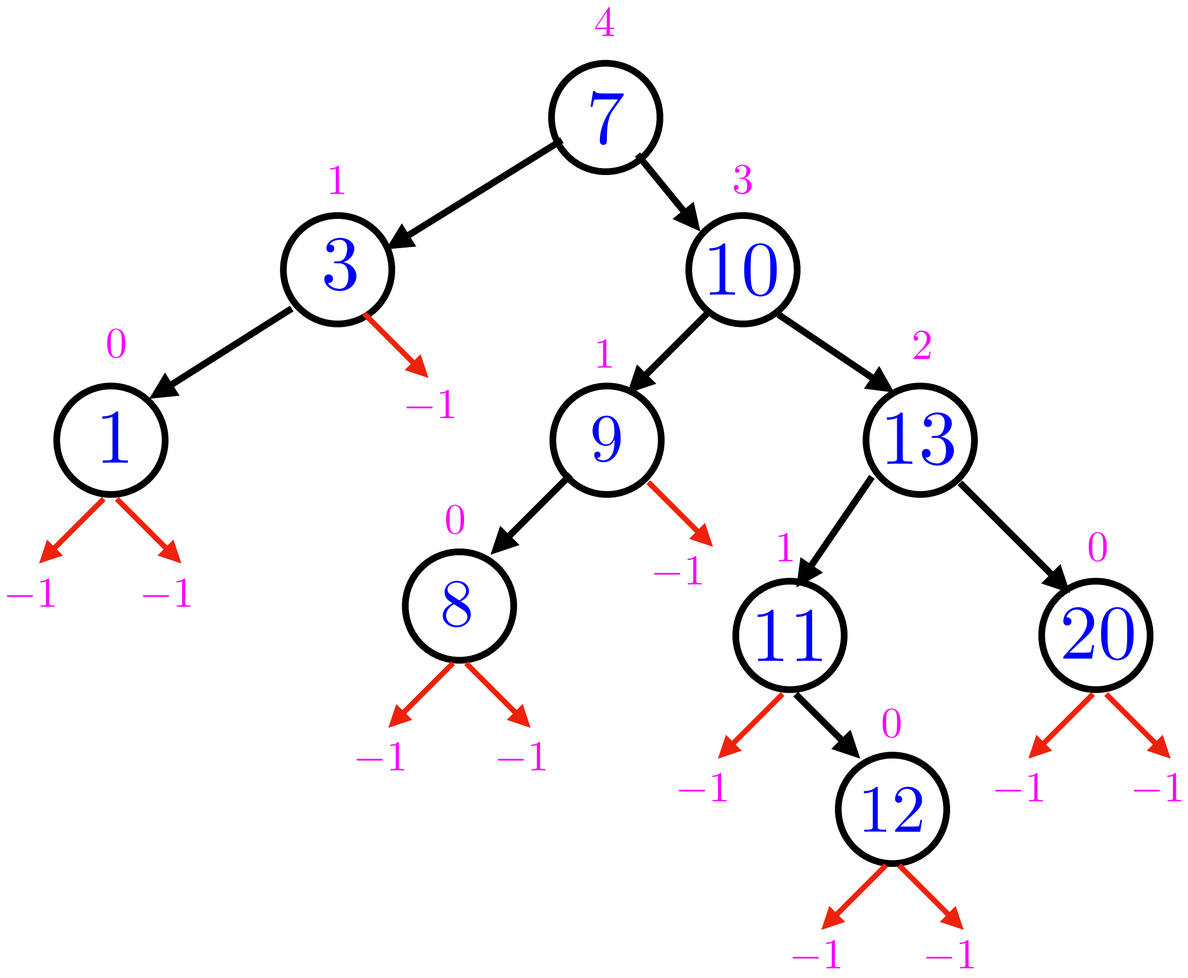
An AVL tree maintains the property that left and right children differ in height by at most one.

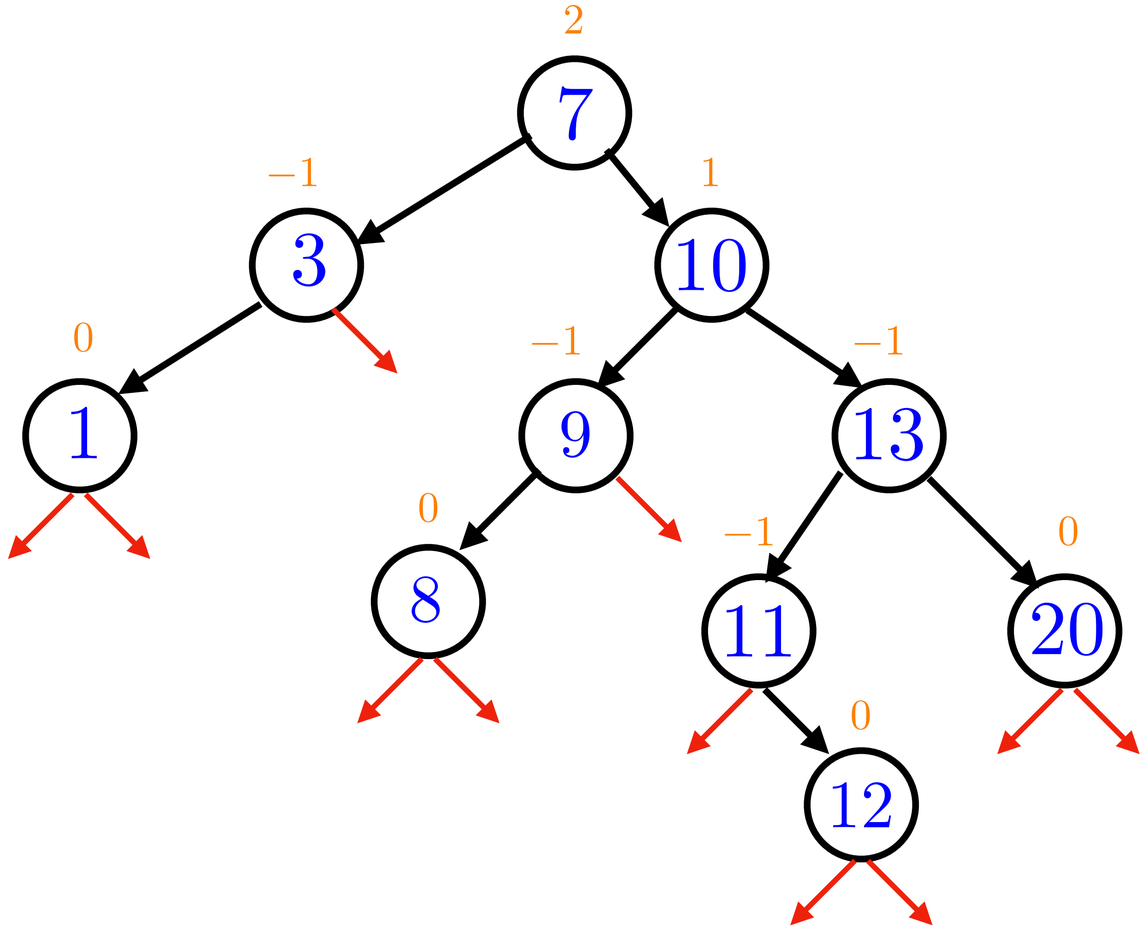
Call the balance factor of a node to be the height of its right subtree minus the height of its left subtree.

We say a node has the AVL property if it balance factor is in $\{-1, 0, 1\}$.

In an AVL tree every node has the AVL property.

Balance Factor





Is this an AVL tree?

Height of an AVL tree

AVL trees

In a moment we will see how to insert keys and maintain the AVL property.

First let's see why we would want to do this.

Key fact: an AVL tree with n nodes has height at most $2 \log n$.

How do we maximise the height of an AVL tree with n nodes?

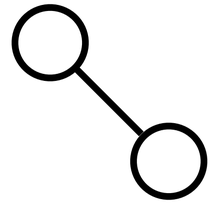
How do we minimise the number of nodes in an AVL tree with height h?

Small examples

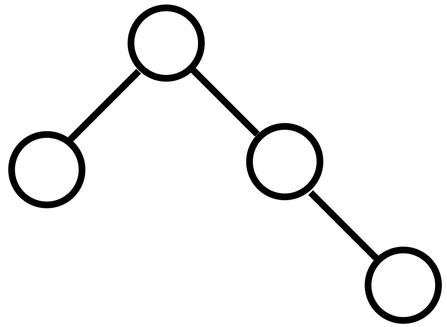
Let $T(h)$ be the minimum number of nodes in an AVL tree of height h.

$T(0) = 1$

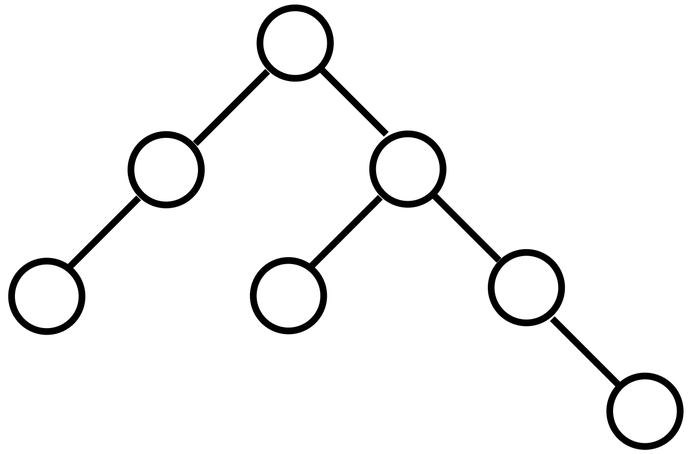
$T(1) = 2$



$T(2) = 4$



$T(3) = 7$

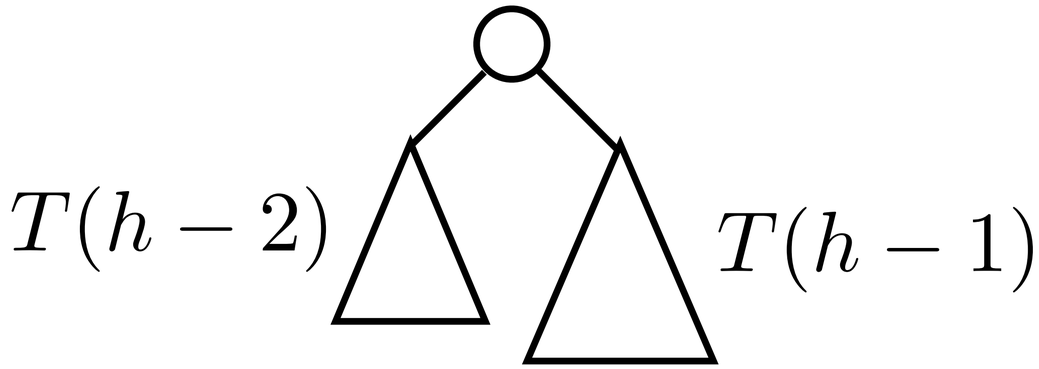


General Case

Let $T(h)$ be the minimum number of nodes in an AVL tree of height h.

Clearly $T(h)$ is an increasing function.

One child of the root should have height $h-1$ and the other $h-2$.



$$T(h) = 1 + T(h-1) + T(h-2)$$

Simple Bound

$$T(h) = 1 + T(h-1) + T(h-2)$$
$$\geq 2T(h-2)$$

$T(0) = 1$

$T(2) \geq 2$

$T(4) \geq 4$

$T(h) \geq 2^{h/2}$

An AVL tree with n nodes has height $\leq 2 \log n$.

Fibonacci numbers

$$T(h) = 1 + T(h-1) + T(h-2)$$

This looks a lot like the recurrence for Fibonacci numbers!

	0	1	2	3	4	5	6	7	8
Fibonacci	0	1	1	2	3	5	8	13	21
$T(h)$	1	2	4	7	12	20	33	54	88

$$T(h) = F(h+3) - 1$$

This gives a better upper bound on the depth of roughly $1.44 \log n$.

AVL Insert

AVL insert

Say that we have an AVL tree. We want to insert a key and keep the AVL property.

We first insert the key in the usual way.

The insertion changes the height of a node by at most one.

After usual BST insertion, each node has balance factor in $\{-2, -1, 0, 1, 2\}$.

We fix the tree from the bottom up to restore the AVL property.

Rotation

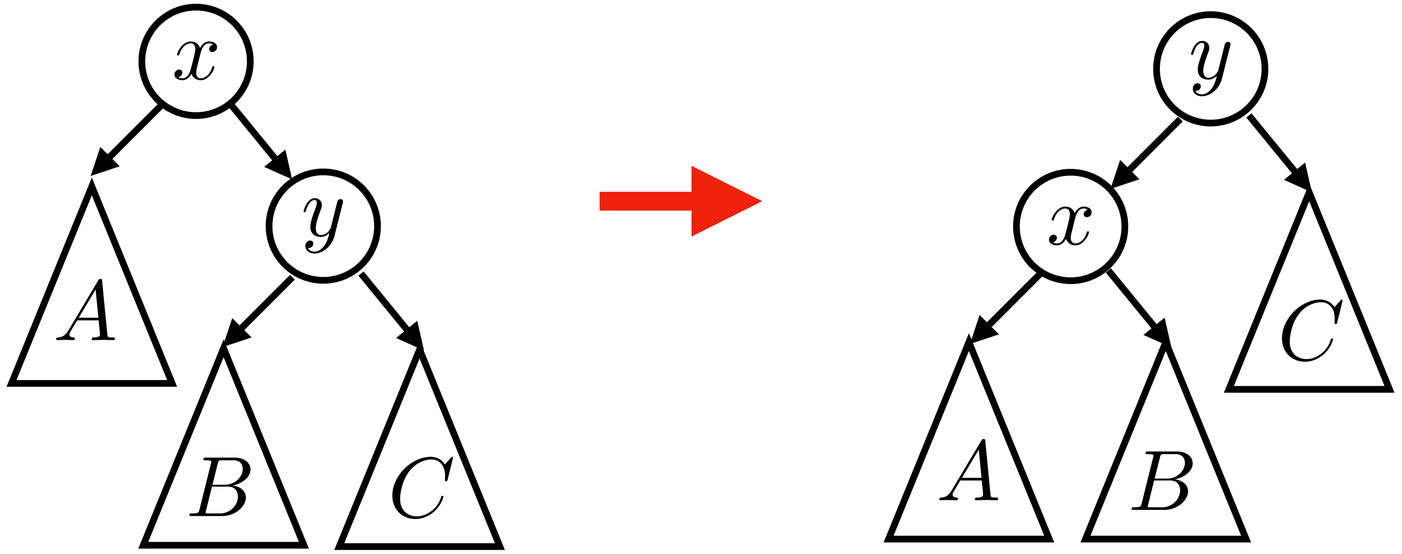
After usual BST insertion, each node has balance factor in $\{-2, -1, 0, 1, 2\}$.

We fix the tree from the bottom up to restore the AVL property.

Let's see how to fix a minimal violating node---all its children satisfy the AVL property.

The key to this fix is an operation on BSTs called rotation.

Left Rotate

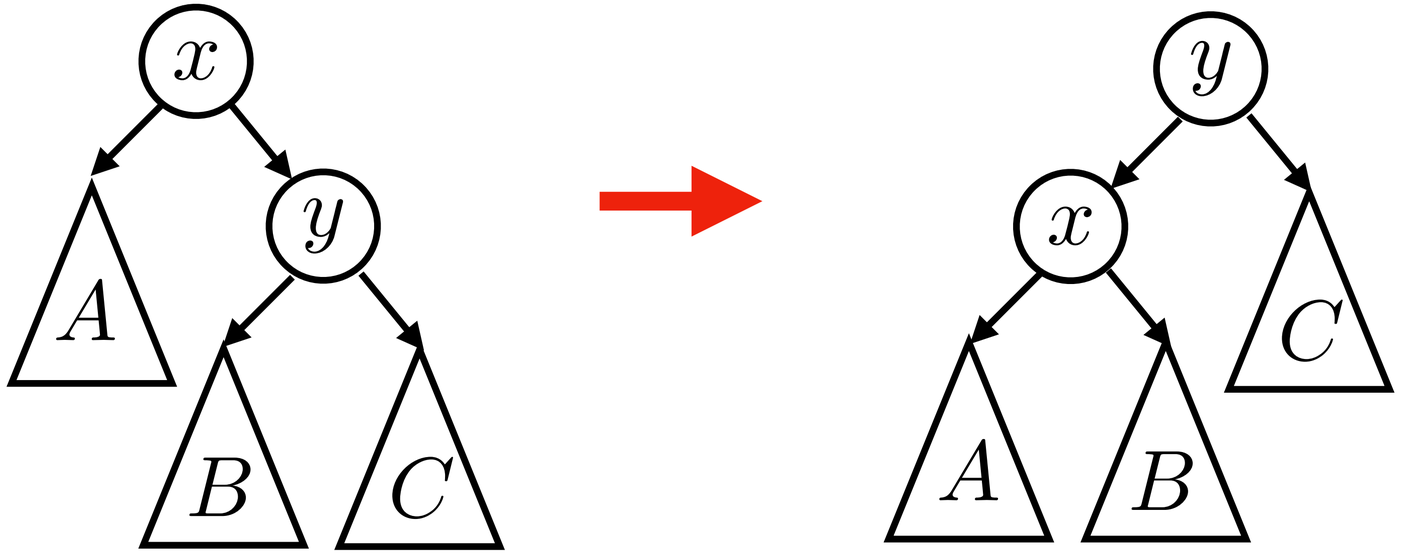


This is a left rotation of x .

Key fact: Left rotation preserves the BST property.

Everything in B is greater than x and less than y .

Left Rotate

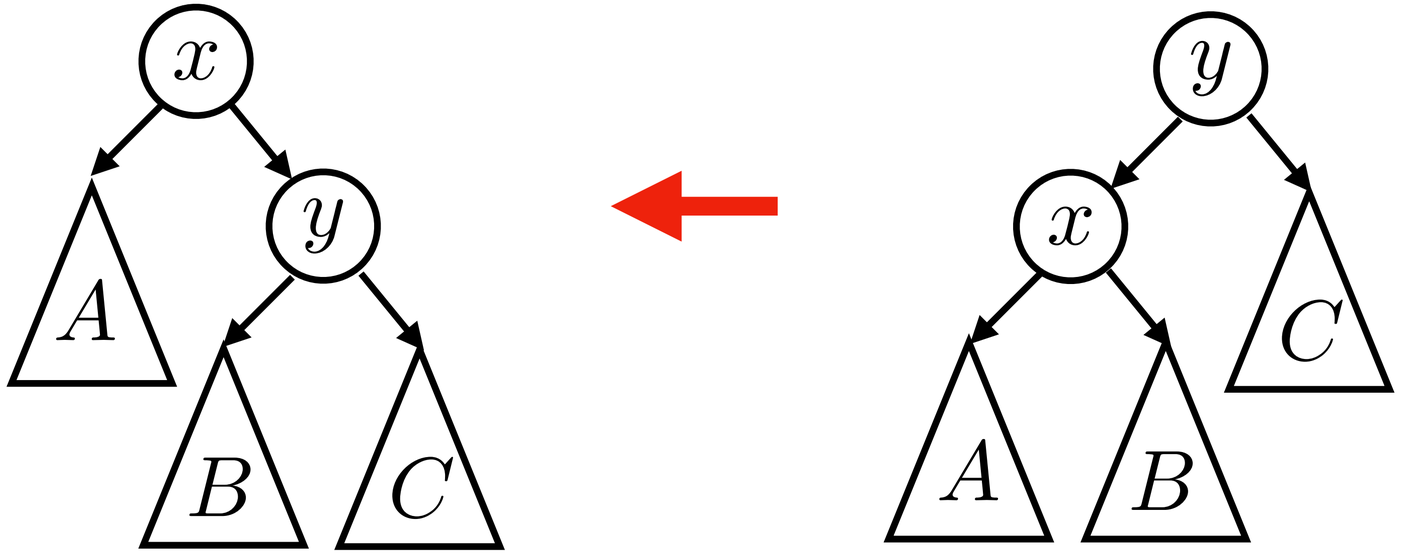


This is a left rotation of x .

Key fact: Left rotation preserves the BST property.

Left rotate can be done with a constant number of pointer changes.

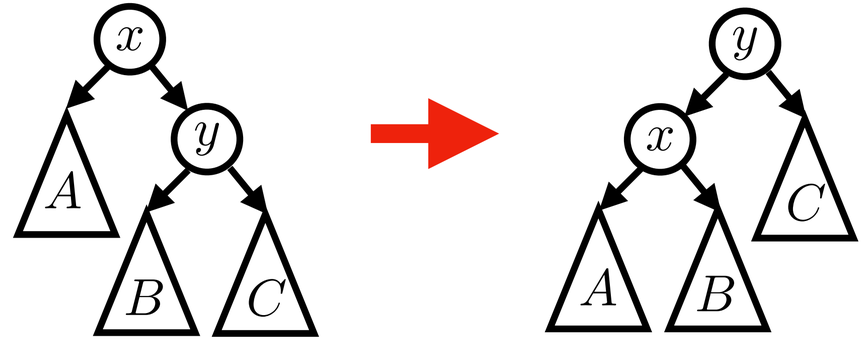
Right Rotate



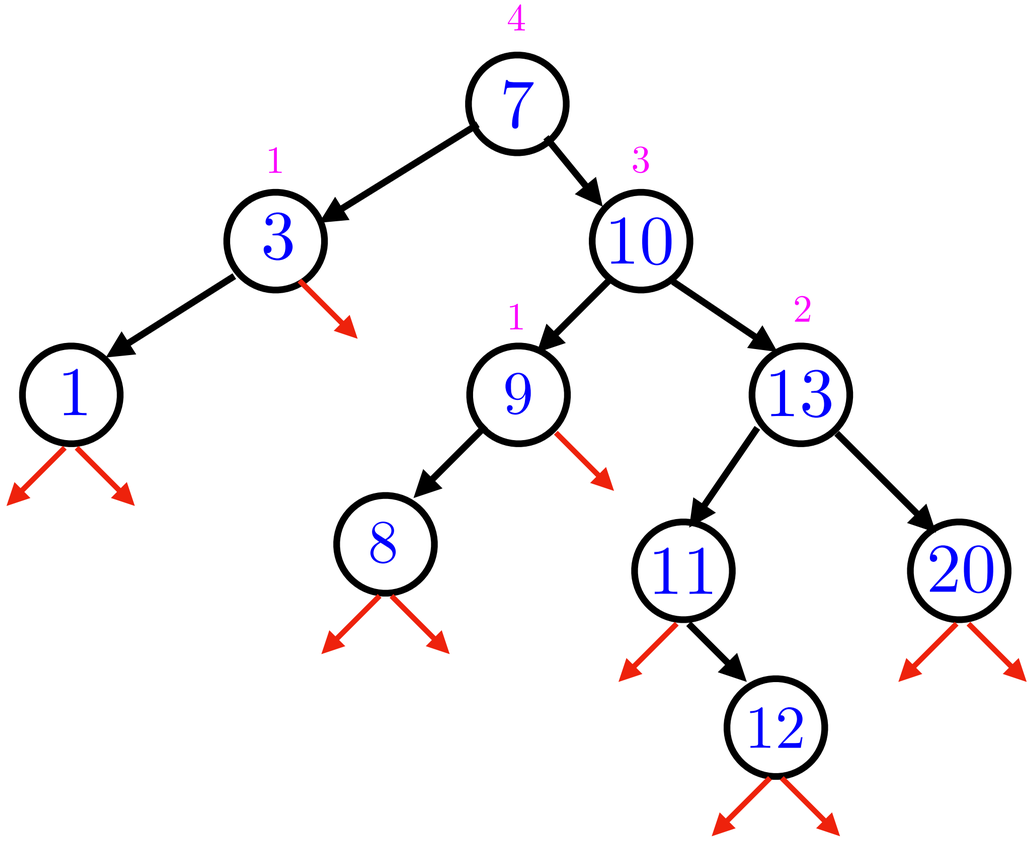
Right rotation is the inverse of left rotation. This is a right rotation of y .

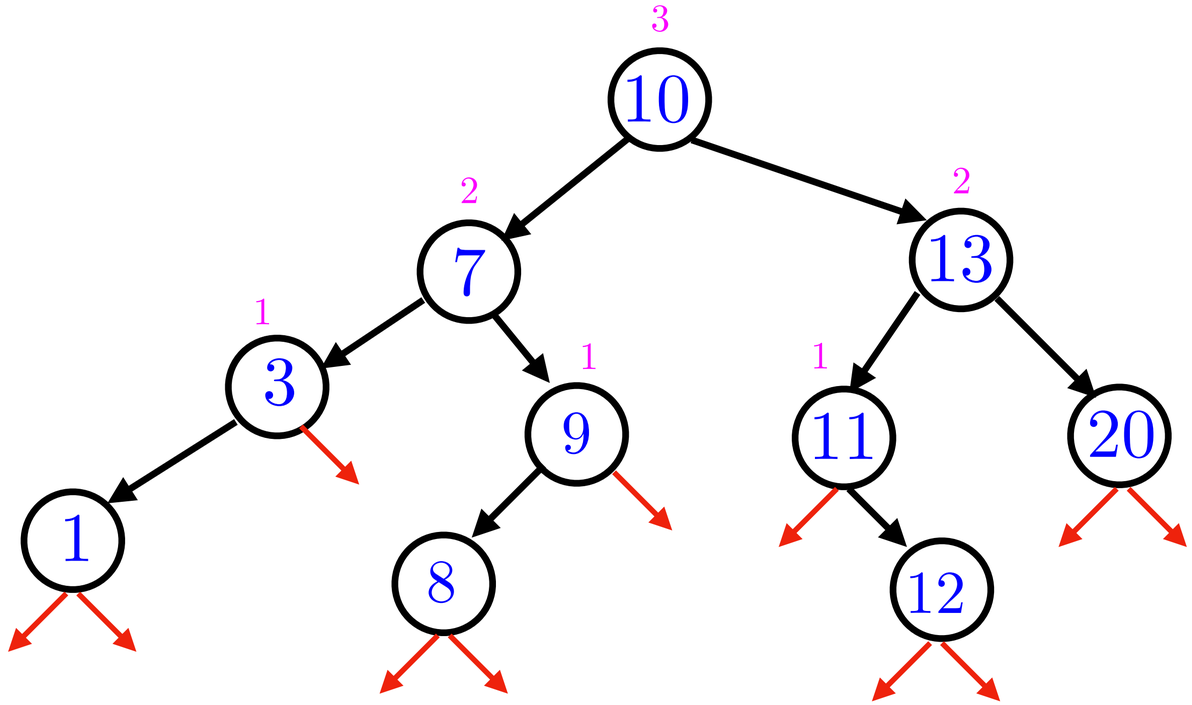
It also preserves the BST property.

Example: Left Rotate



This restores the AVL property!



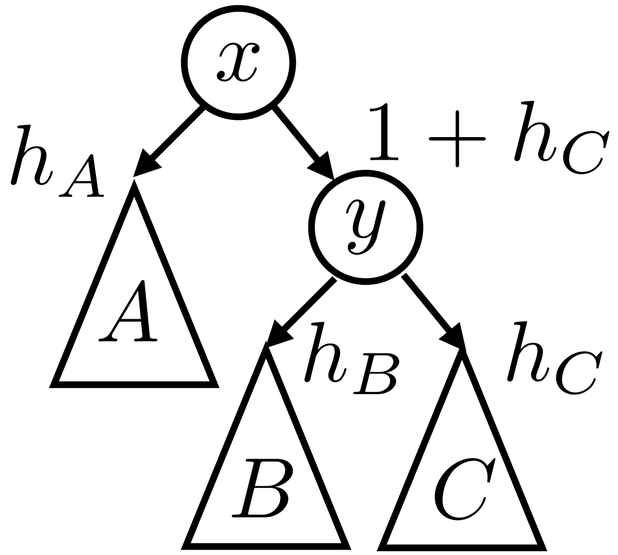


General Case 1

Assume x is a minimal violating node with balance factor 2.

For case 1 we further assume $h_C \geq h_B$.

In this case a left rotation restores the AVL property.



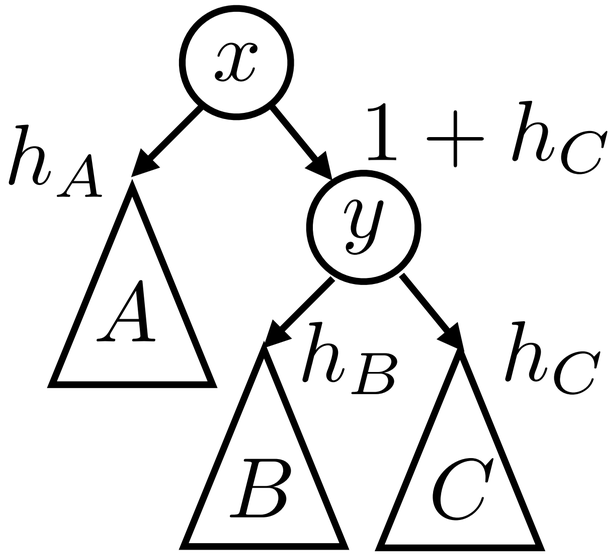
We know that $1 + h_C = h_A + 2$ so $h_C = h_A + 1$.

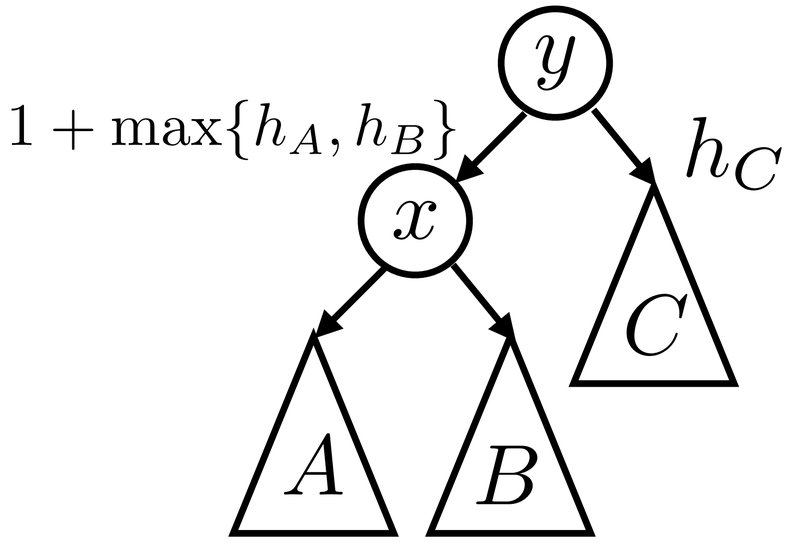
General Case 1

Assume x is a minimal violating node with balance factor 2.

For case 1 we further assume $h_C \geq h_B$.

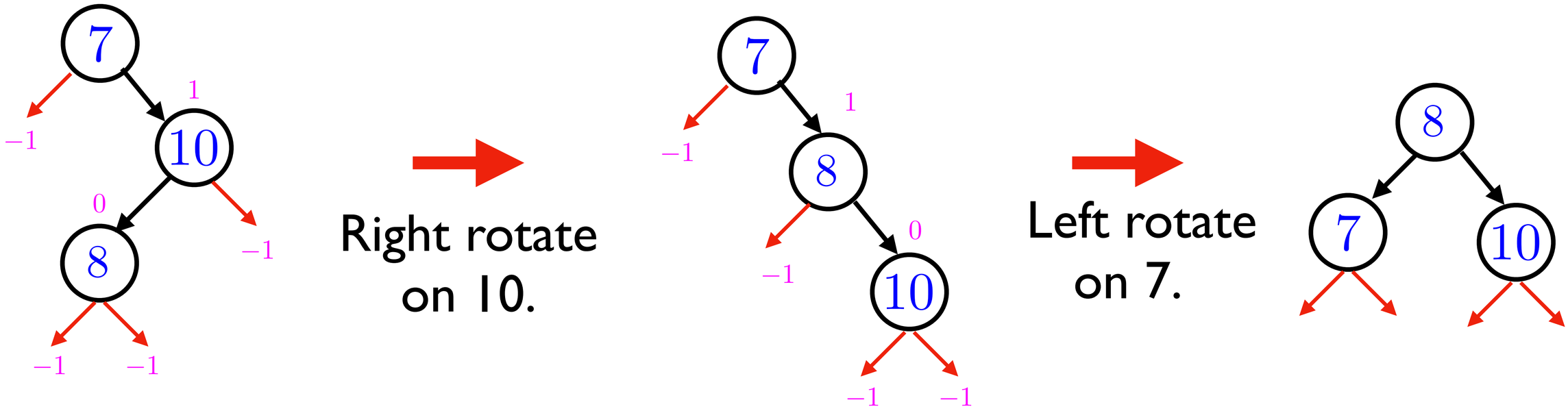
This means $h_A = h_C - 1$ and so $\max\{h_A, h_B\} \in \{h_C - 1, h_C\}$.





Case 2: Example

The case we haven't handled is where $h_B > h_C$.



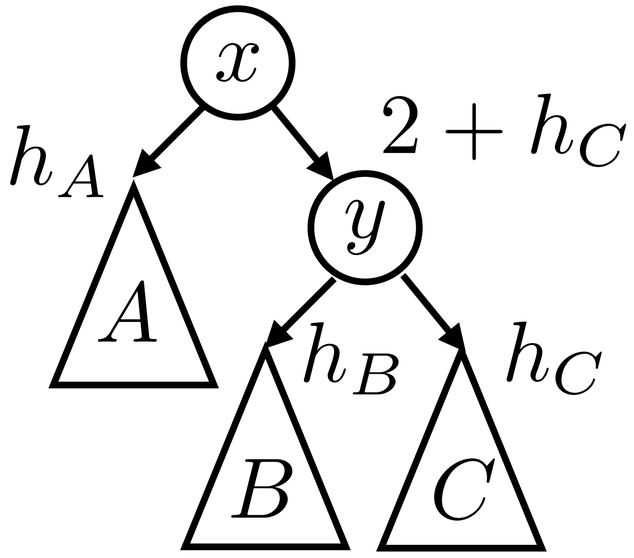
Wait, the tree is still not AVL!

But now we have reduced it to Case 1.

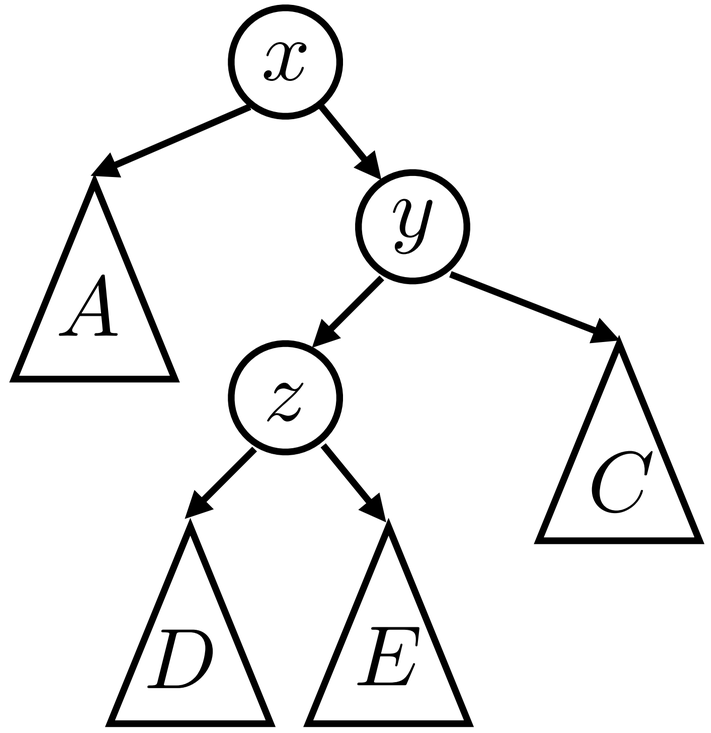
General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



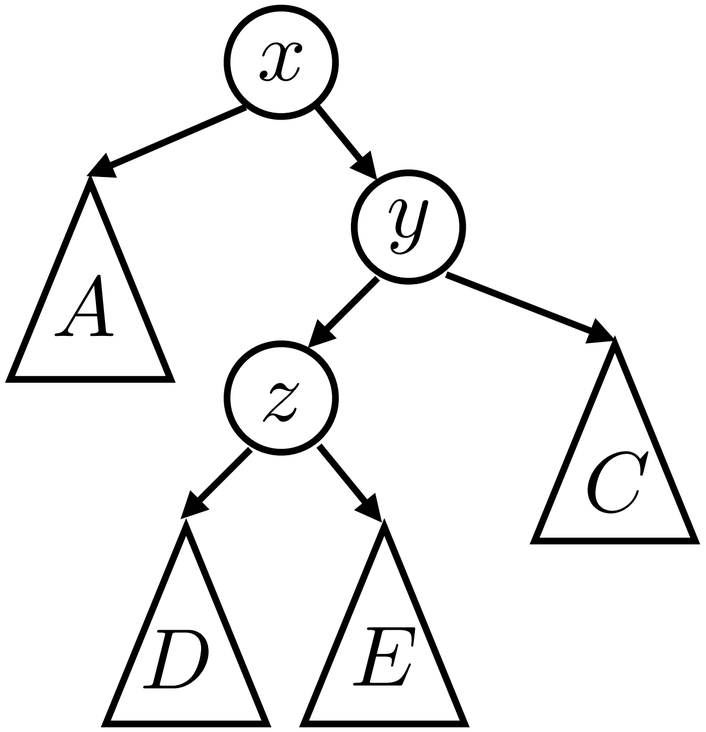
We need to look inside the B tree now.



General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.



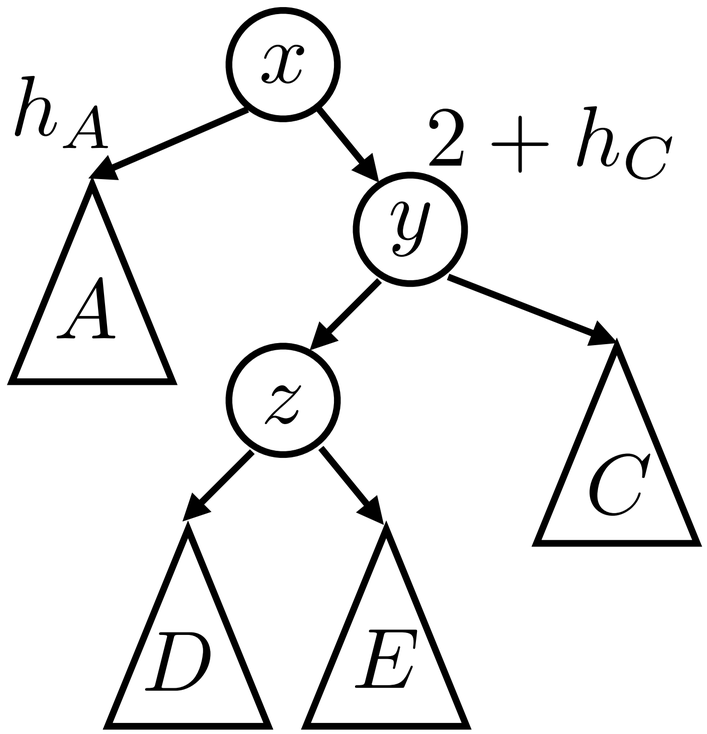
Both of D, E have height $\leq h_C$.

One of them has height h_C.

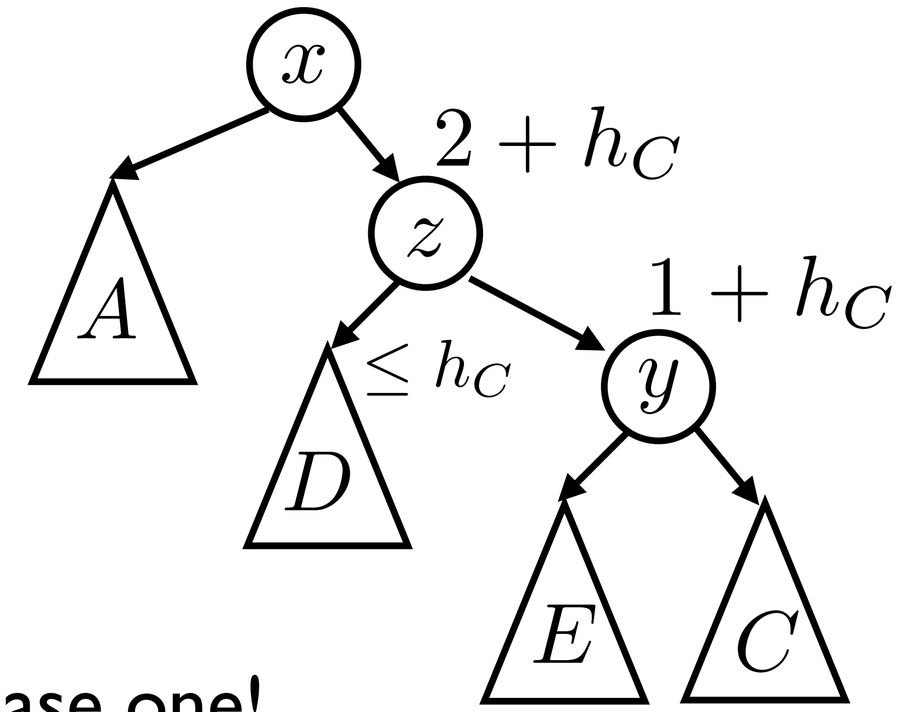
General Case 2

Assume x is a minimal violating node with balance factor 2.

In case 2 $h_B = h_C + 1$ which means $h_A = h_C$.

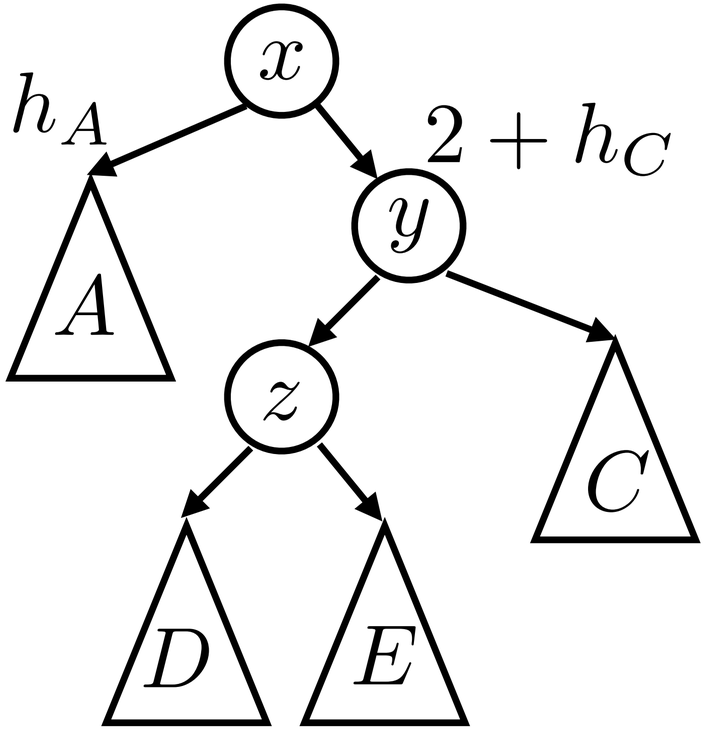


Right rotate on y

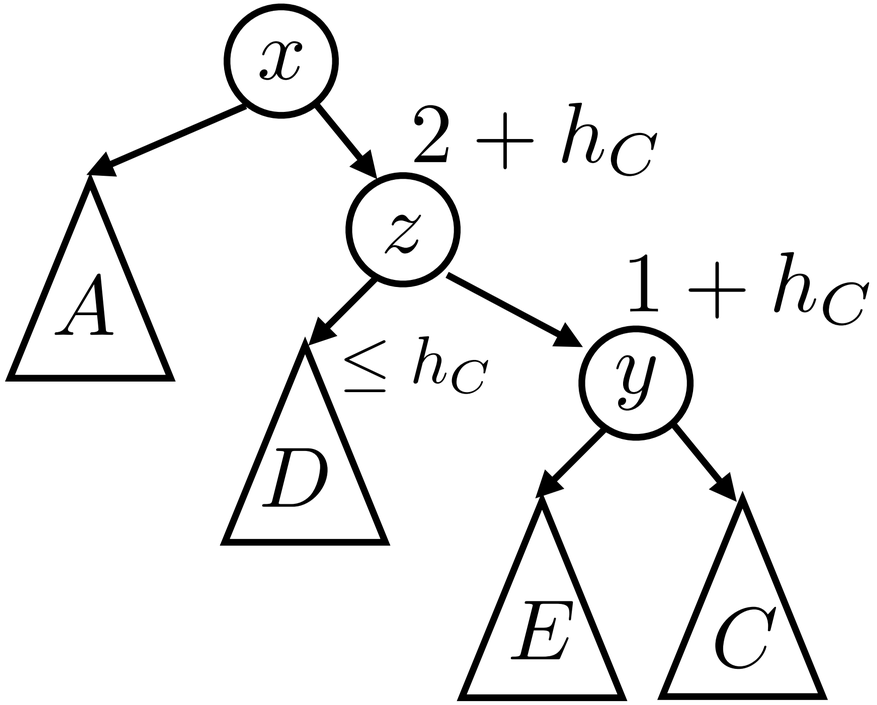


Now we are back in case one!

Case 2: Summary



Right rotate on y



Case 2 can be solved with 2 rotations.

We right rotate on y, and then left rotate on x .

AVL insertion: summary

We have now seen how to repair the balance factor of a single node with a constant number of rotations.

Inserting a node can upset the balance factor of any node on the path from the insertion point to the root.

We may have to do this repair $\Theta(h)$ times.

This still gives us $O(\log n)$ insertion time in an AVL tree.

AVL tree: summary

An AVL tree gives $O(\log n)$ worst case time for all our operations.

operation	worst case
`insert`	$O(\log n)$
`remove`	$O(\log n)$
`contains`	$O(\log n)$
`successor`	$O(\log n)$

AVL sort

AVL trees provide another comparison-based $O(n \log n)$ sorting algorithm.

Insert all n items into an AVL tree.

Takes $O(n \log n)$ time.

Do in-order traversal to write the sorted order.

Takes $O(n)$ time.

This is nice as we can support all the other operations as well.