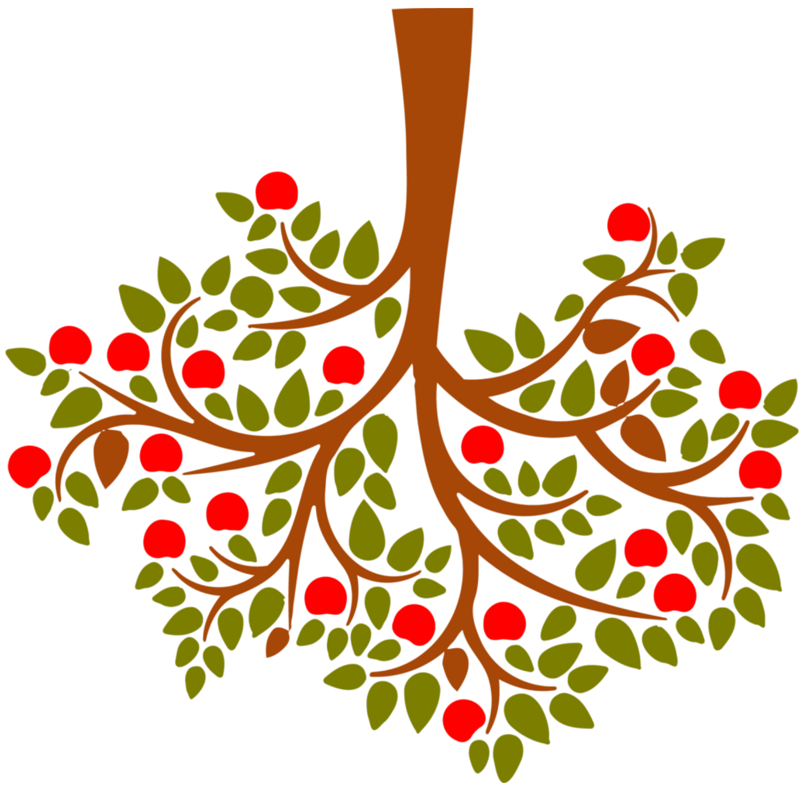
CS11212 - **Spring** 2022

Data Structures & Introduction to **Algorithms**

Data Structures

Trees: Basic Operations

Ibrahim Albluwi



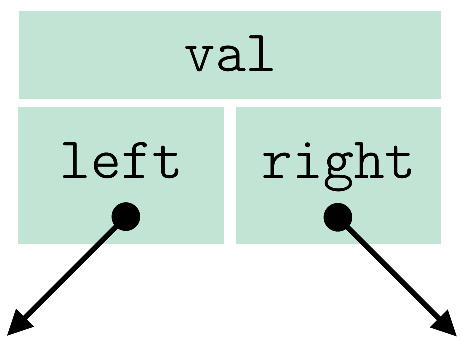
Tree Data Structures

- Definitions and properties
- Basic Operations
- Balanced binary search trees
- Tree traversals

Implementation

Class `Node`. Both a BST node and a DLL node store a `value` and two pointers to other nodes.

In a BST node, the two pointers represent links to the `left` and `right` children.

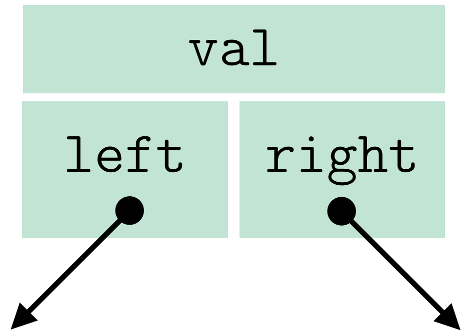


In a DLL node, the two pointers represent links to the `next` and `previous` nodes.

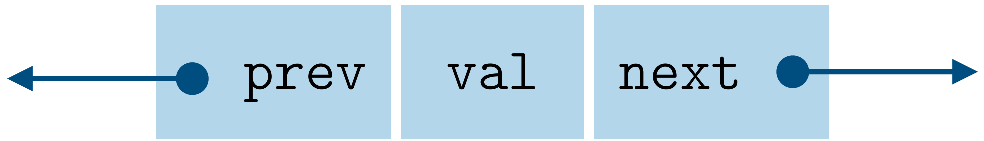
Implementation

Class Node. Both a BST node and a DLL node store a `value` and two pointers to other nodes.

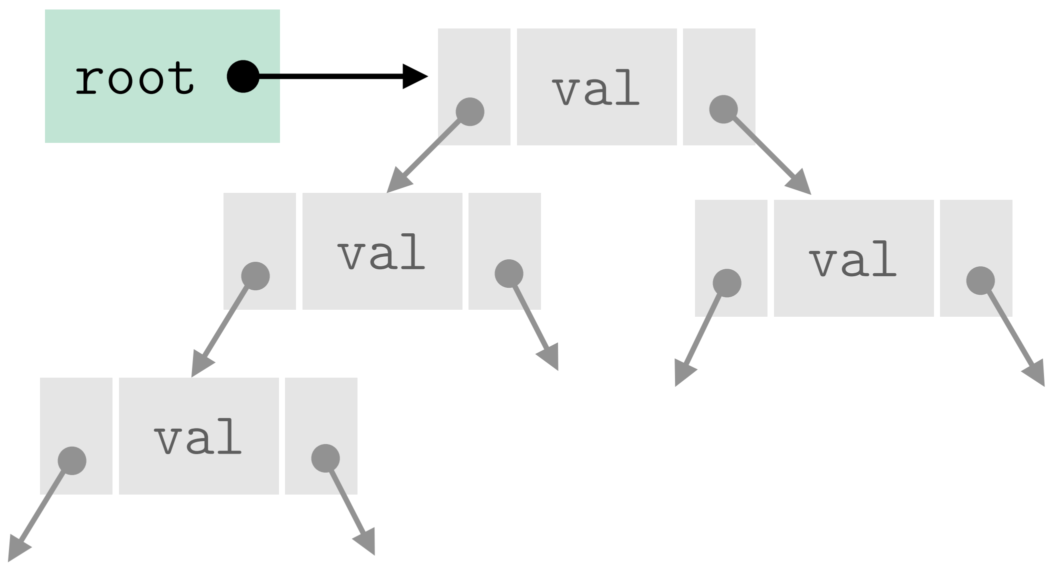
In a BST node, the two pointers represent links to the `left` and `right` children.

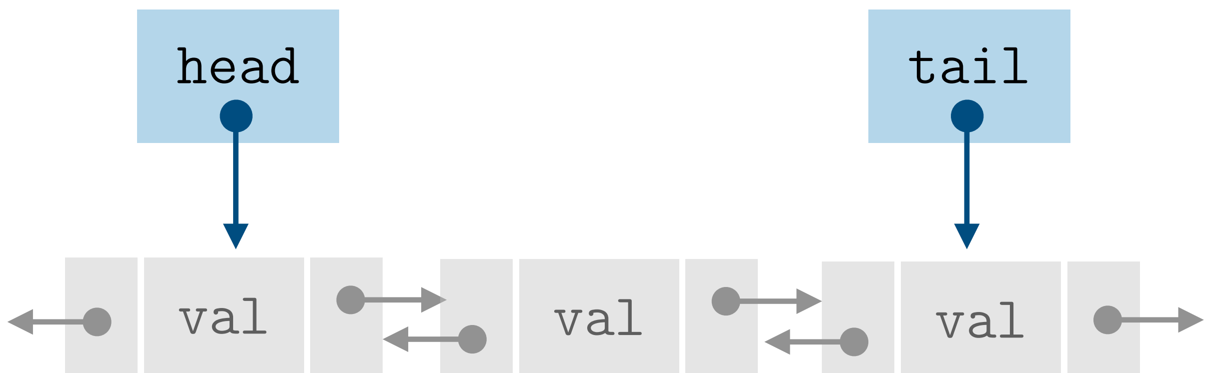


In a DLL node, the two pointers represent links to the `next` and `previous` nodes.



Class BST. Stores a pointer to the `root` of the tree.
A DLL class stores a pointer to the `head` of the list (and the `tail` of the list).





Recursive Search

```
template <class T>
bool BST<T>::contains(const T& val) {
     return contains(val, root);
}
```

public function used by the user calls the `private` recursive version

```
template <class T>
bool BST<T>::contains(const T& val, Node<T>* n)
{
    if (n == nullptr)
        return false;

    if (n->val == val)
        return true;

    if (val > n->val)
        return contains(val, n->right);
    else
        return contains(val, n->left);
}
```

Recursive Search

```
template <class T>
bool BST<T>::contains(const T& val) {
     return contains(val, root);
}
```

```
template <class T>
bool BST<T>::contains(const T& val, Node<T>* n)
{
    if (n == nullptr)
        return false;

    if (n->val == val)
        return true;

    if (val > n->val)
        return contains(val, n->right);
    else
        return contains(val, n->left);
}
```

`private` recursive function.

Requires a pointer to the `root` of the tree (or subtree) where the search will be performed

Recursive Search

```
template <class T>
bool BST<T>::contains(const T& val) {
     return contains(val, root);
}

template <class T>
bool BST<T>::contains(const T& val, Node<T>* n)
{
    if (n == nullptr)
        return false;

    if (n->val == val)
        return true;

    if (val > n->val)
        return contains(val, n->right);
    else
        return contains(val, n->left);
}
```

Base case.

`val` can't be present in an empty tree!

Recursive Search

```
template <class T>
bool BST<T>::contains(const T& val) {
     return contains(val, root);
}

template <class T>
bool BST<T>::contains(const T& val, Node<T>* n)
{
    if (n == nullptr)
        return false;
    if (n->val == val)
        return true;
    if (val > n->val)
        return contains(val, n->right);
    else
        return contains(val, n->left);
}
```

val found in the current node

Recursive Search

```
template <class T>
bool BST<T>::contains(const T& val) {
     return contains(val, root);
}

template <class T>
bool BST<T>::contains(const T& val, Node<T>* n)
{
    if (n == nullptr)
        return false;

    if (n->val == val)
        return true;

    if (val > n->val)
        return contains(val, n->right);
    else
        return contains(val, n->left);
}
```

Search recursively in the `left` subtree or in the `right` subtree

Iterative Search

```
template <class T>
bool BST<T>::contains(const T& val) {
    Node<T>* curr = root;

    while (curr != nullptr) {
        if (curr->val == val)
            return true;

        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    return false;
}
```

Search Running Time

```
template <class T>
bool BST<T>::contains(const T& val) {
    Node<T>* curr = root;

    while (curr != nullptr) {
        if (curr->val == val)
            return true;

        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    return false;
}
```

Best Case. $O(1)$
If `val` is found at the root of the tree.

It does not matter if the tree is balanced or not.

Search Running Time

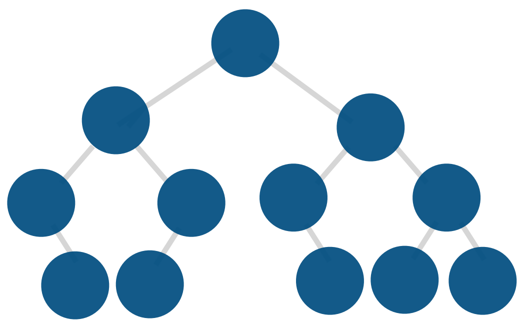
```
template <class T>
bool BST<T>::contains(const T& val) {
    Node<T>* curr = root;

    while (curr != nullptr) {
        if (curr->val == val)
            return true;

        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    return false;
}
```

Best Case. $O(1)$
If val is found at the root of the tree.

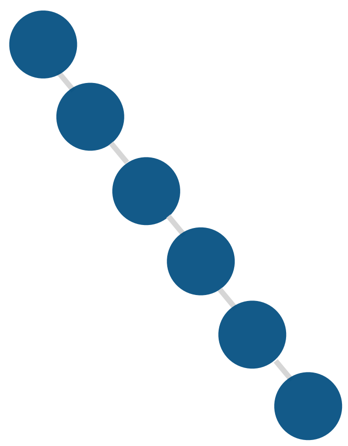
It does not matter if the tree is balanced or not.

Worst Case. $O(\text{height})$
If the search proceeds to the last level in the tree.

$O(\log n)$ if the tree is balanced.
$O(n)$ if the tree is unbalanced.



height = $O(\log n)$

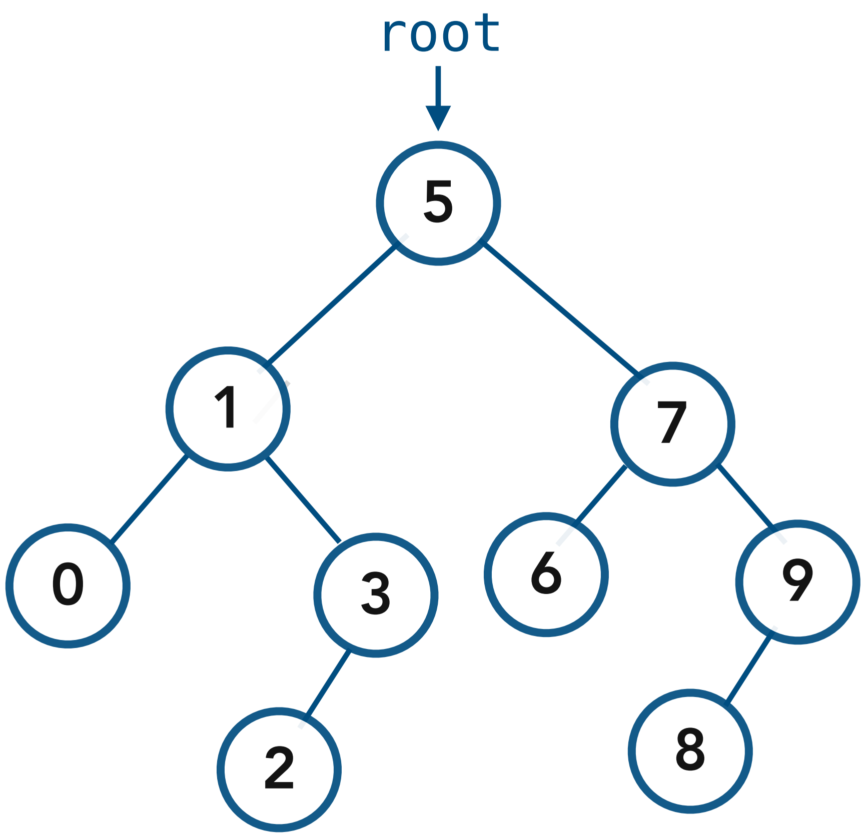


height = $O(n)$

Insert

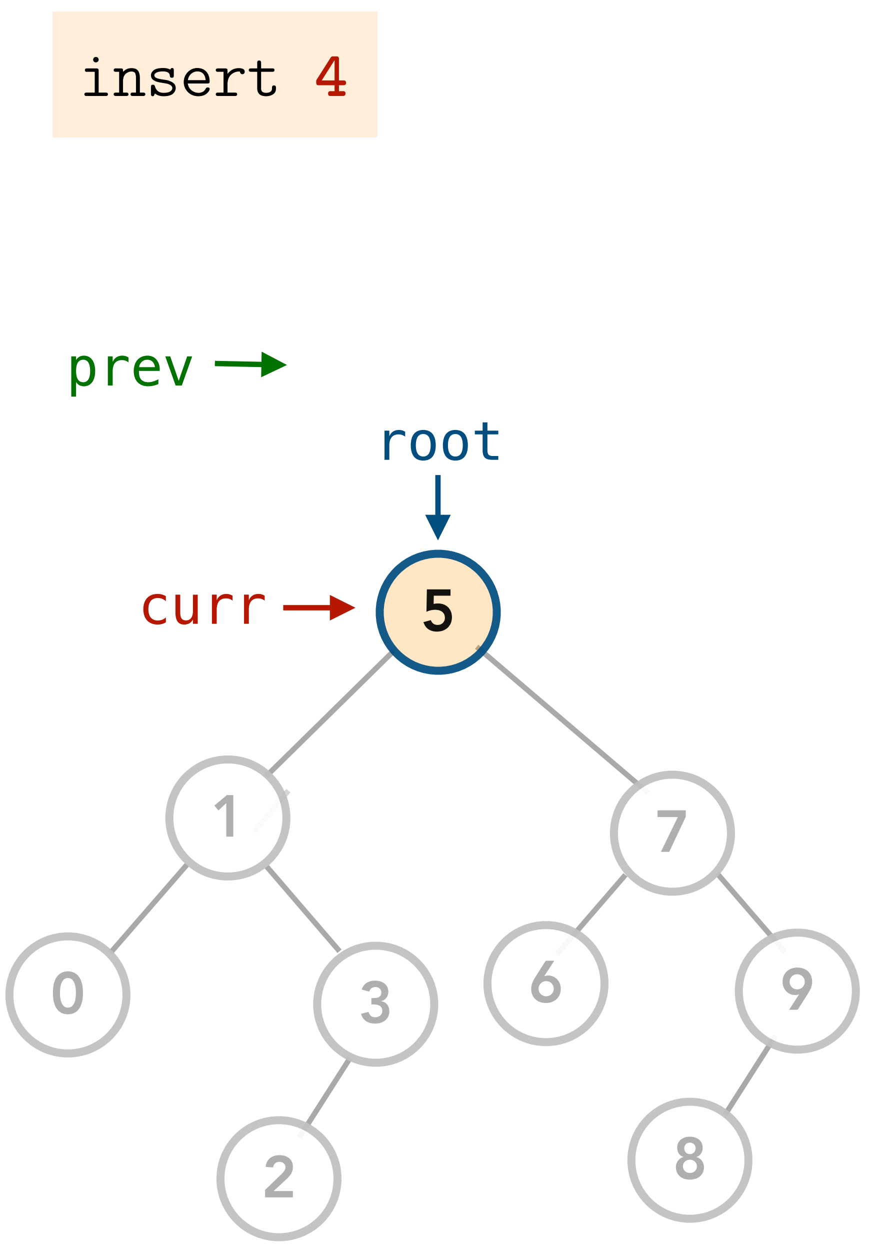
```
template <class T>
void BST<T>::insert(const T& val) {
     Node<T>* prev;
     Node<T>* curr = root;

     while (curr != nullptr) {
         prev = curr;
         if (val == curr->val) return;
         if (val  < curr->val) curr = curr->left;
         else                  curr = curr->right;
     }

     Node<T>* node =
             new Node<T>(val, nullptr, nullptr);

     if      (root == nullptr) root        = node;
     else if (val < prev->val) prev->left  = node;
     else                      prev->right = node;
}
```

`insert 4`



Insert

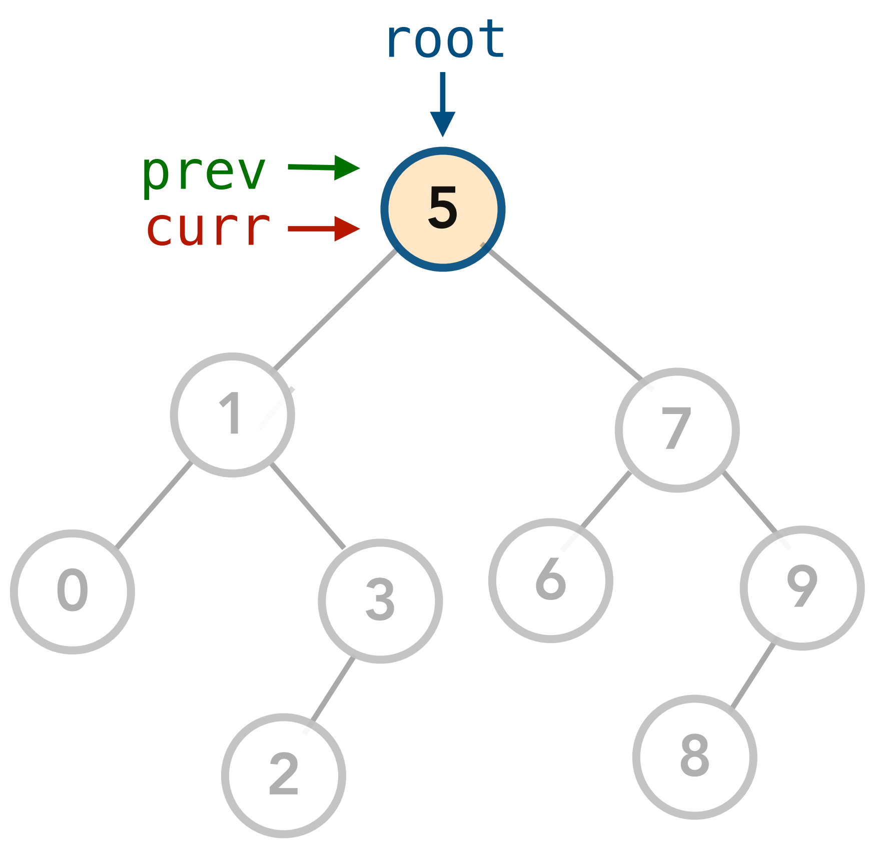
```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
        new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

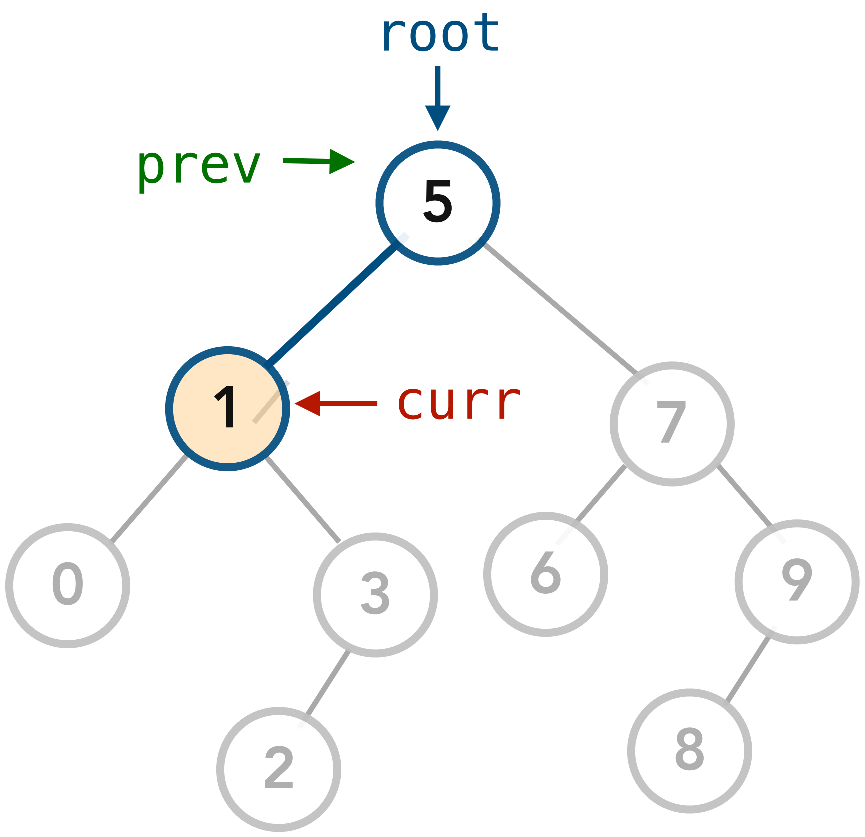
insert 4



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
     Node<T>* prev;
     Node<T>* curr = root;

     while (curr != nullptr) {
         prev = curr;
         if (val == curr->val) return;
       ● if (val  < curr->val) curr = curr->left;
         else                  curr = curr->right;
     }

     Node<T>* node =
             new Node<T>(val, nullptr, nullptr);

     if       (root == nullptr) root        = node;
     else if (val < prev->val) prev->left  = node;
     else                      prev->right = node;
}
```

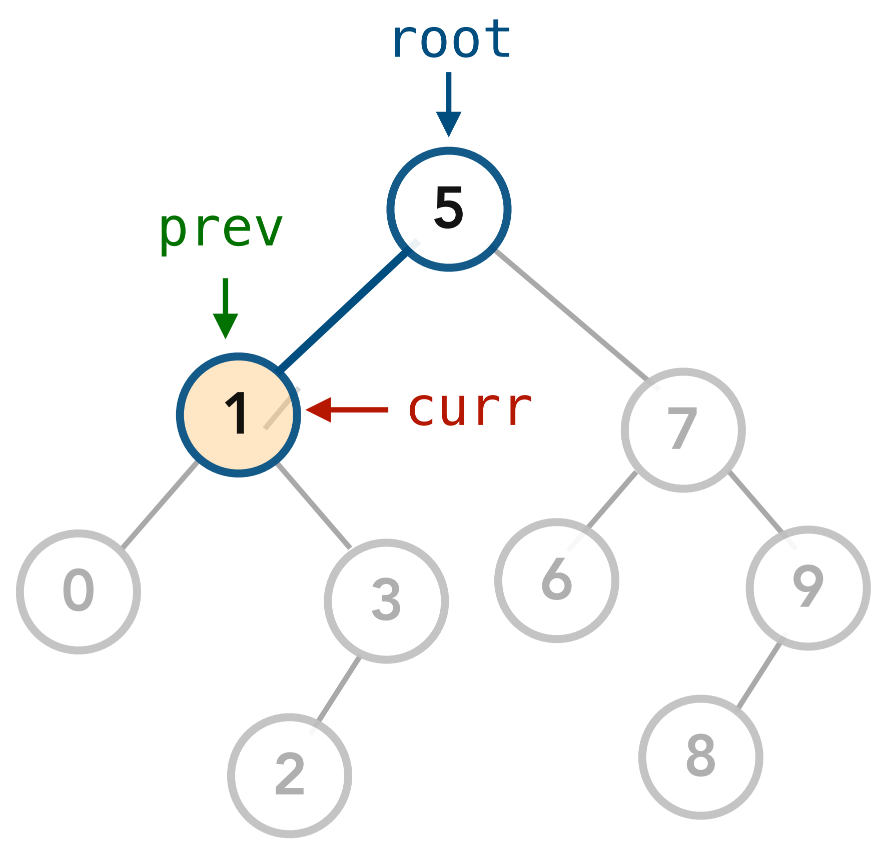
insert 4



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
     Node<T>* prev;
     Node<T>* curr = root;

     while (curr != nullptr) {
         prev = curr;
         if (val == curr->val) return;
         if (val  < curr->val) curr = curr->left;
         else                  curr = curr->right;
     }

     Node<T>* node =
             new Node<T>(val, nullptr, nullptr);

     if      (root == nullptr) root        = node;
     else if (val < prev->val) prev->left  = node;
     else                      prev->right = node;
}
```

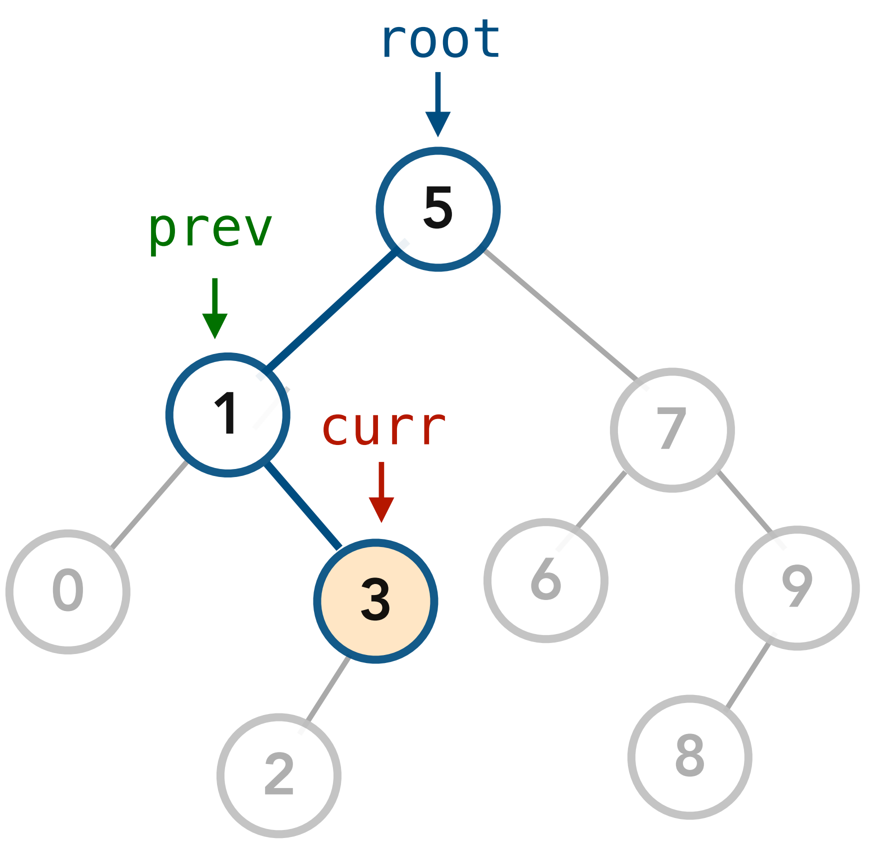
insert 4



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

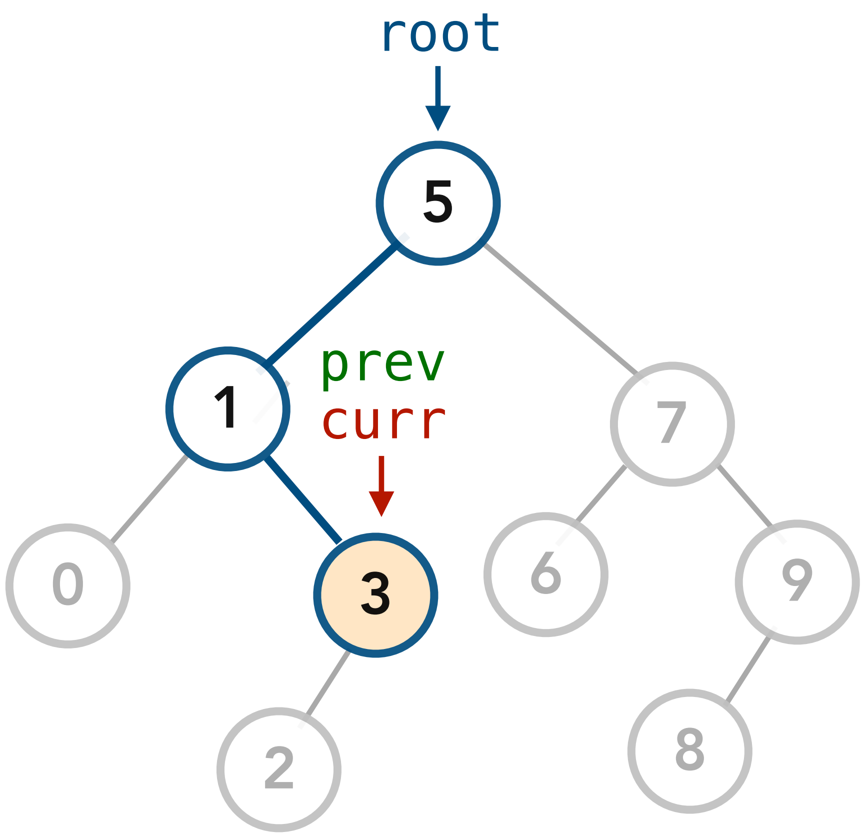
insert 4



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

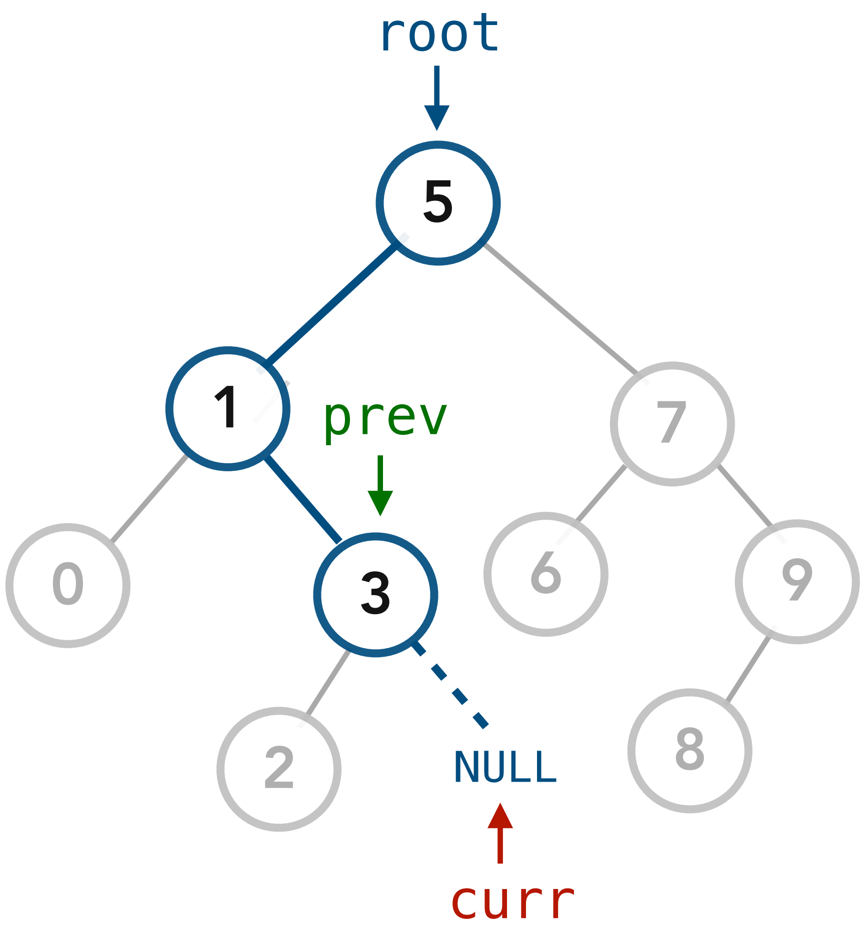
insert 4



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

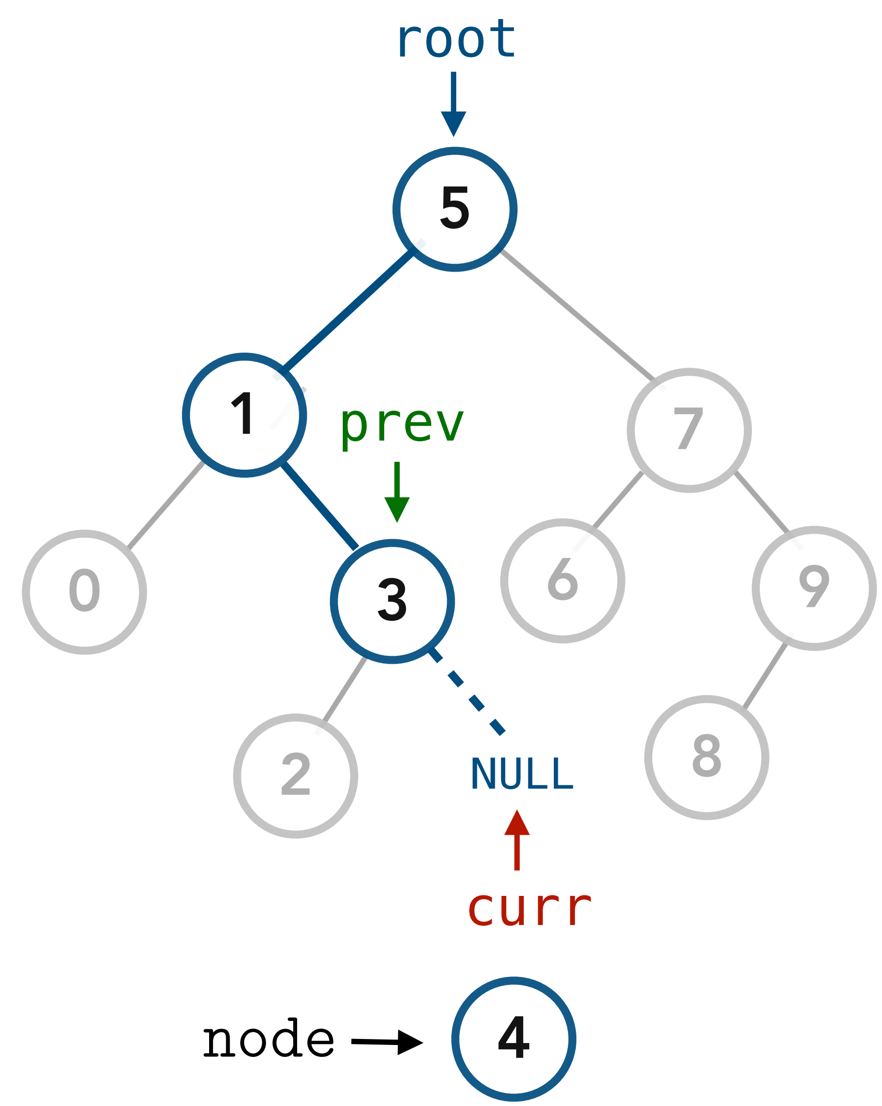
```
insert 4
```



Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
          new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

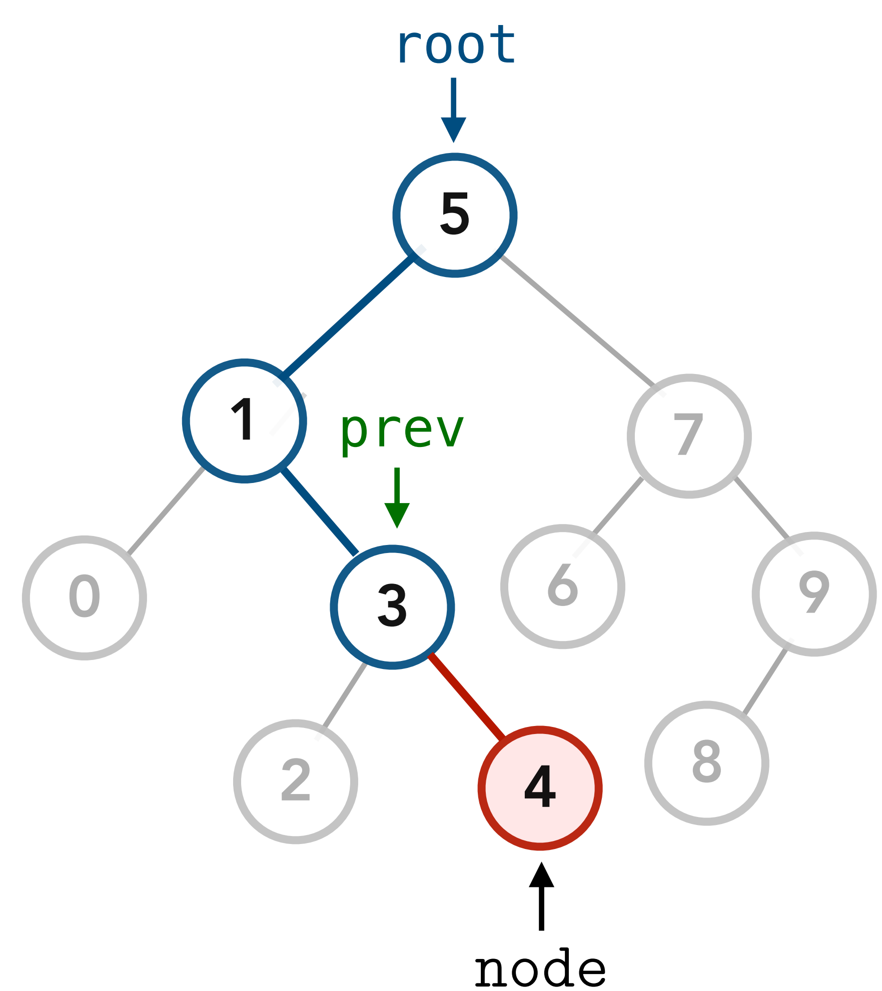
insert 4



Iterative Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {
        prev = curr;
        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
●   else                      prev->right = node;
}
```

insert 4



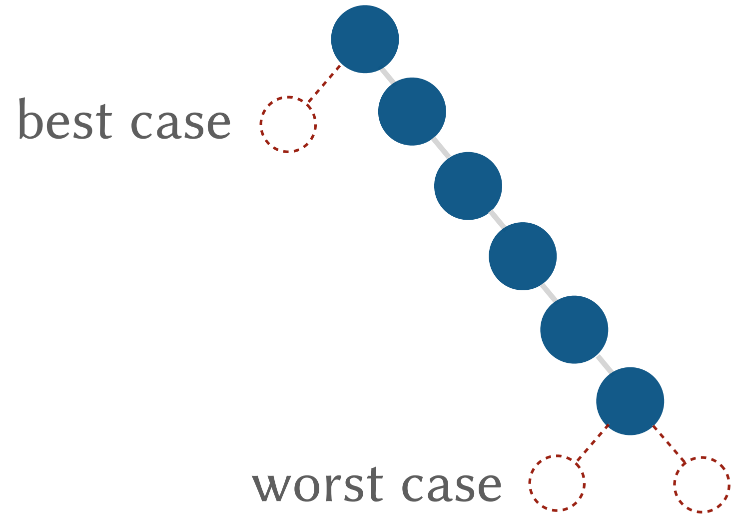
Iterative Insert

```
template <class T>
void BST<T>::insert(const T& val) {
     Node<T>* prev;
     Node<T>* curr = root;

     while (curr != nullptr) {
          prev = curr;
          if (val == curr->val) return;
          if (val  < curr->val) curr = curr->left;
          else                  curr = curr->right;
     }

     Node<T>* node =
          new Node<T>(val, nullptr, nullptr);

     if      (root == nullptr) root        = node;
     else if (val < prev->val) prev->left  = node;
     else                      prev->right = node;
}
```

Unbalanced Tree.

Best Case: $O(1)$

Worst Case: $O(n)$



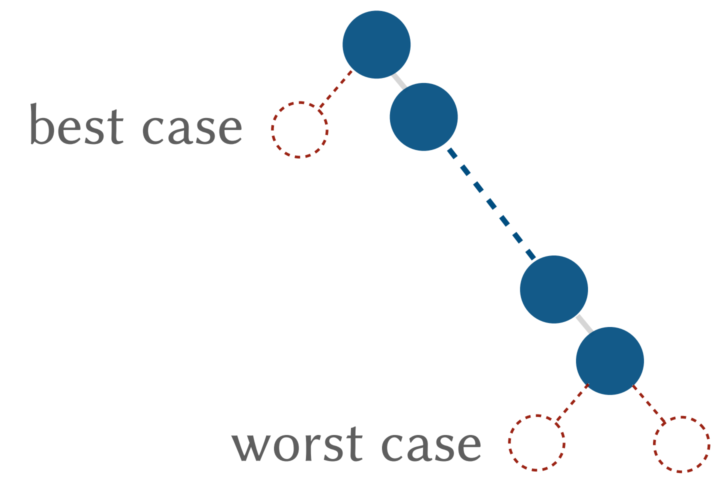
Iterative Insert

```
template <class T>
void BST<T>::insert(const T& val) {
    Node<T>* prev;
    Node<T>* curr = root;

    while (curr != nullptr) {

        prev = curr;

        if (val == curr->val) return;
        if (val  < curr->val) curr = curr->left;
        else                  curr = curr->right;
    }

    Node<T>* node =
            new Node<T>(val, nullptr, nullptr);

    if      (root == nullptr) root        = node;
    else if (val < prev->val) prev->left  = node;
    else                      prev->right = node;
}
```

Unbalanced Tree.

Best Case: $O(1)$

Worst Case: $O(n)$

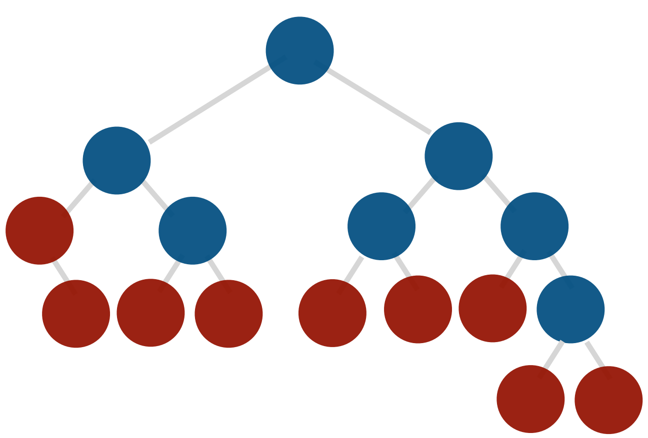


Balanced Tree.

Best Case: $O(1)$

if the value is already at the root

Worst Case: $O(\log n)$



Insertion always happens at the lower levels

Recursive Insert

Optional

```
template <class T>
void BST<T>::insert(T& val) {
     root = insert(val, root);
}

template <class T>
Node<T>* BST<T>::insert(T& val, Node<T>* node) {
    if (node == nullptr)
        return new Node<T>(val, nullptr, nullptr);
    if (val > node->val)
        node->right = insert(val, node->right);
    else if (val < node->val)
        node->left  = insert(val, node->left);

    return node;
}
```

Convince yourself that this code works!

Finding the Max and Min

```
template <class T>
T BST<T>::get_max() const {
    if (is_empty())
        throw string("No max in an empty tree");

    Node<T>* curr = root;
    while (curr->right != nullptr)
        curr = curr->right;

    return curr->val;
}
```

```
template <class T>
T BST<T>::get_min() const {
    if (is_empty())
        throw string("No min in an empty tree");

    Node<T>* curr = root;
    while (curr->left != nullptr)
        curr = curr->left;

    return curr->val;
}
```

Finding the Max and Min

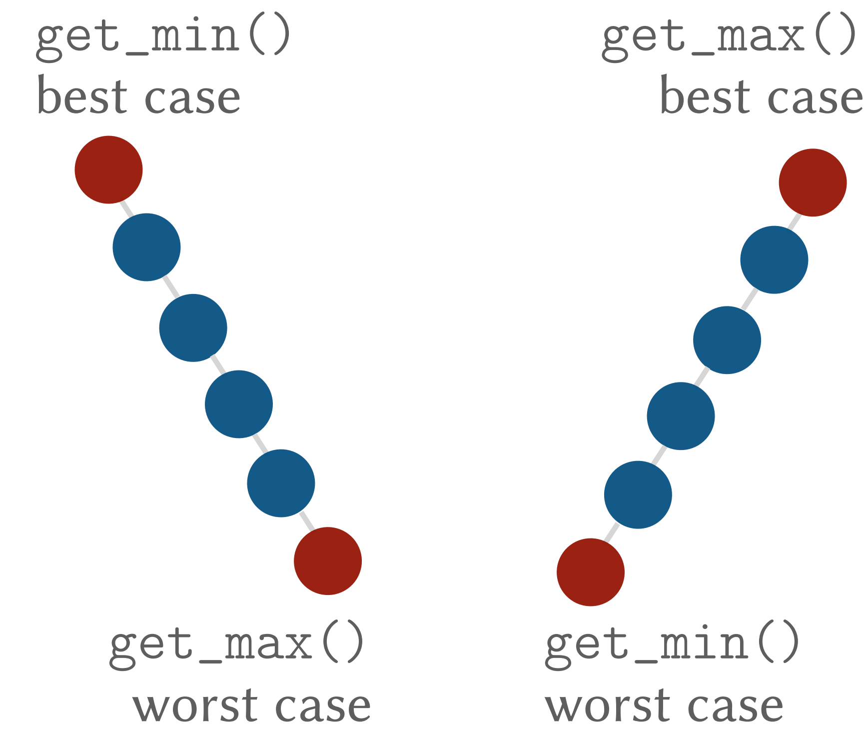
```
template <class T>
T BST<T>::get_max() const {
    if (is_empty())
        throw string("No max in an empty tree");

    Node<T>* curr = root;
    while (curr->right != nullptr)
        curr = curr->right;

    return curr->val;
}
```

```
template <class T>
T BST<T>::get_min() const {
    if (is_empty())
        throw string("No min in an empty tree");

    Node<T>* curr = root;
    while (curr->left != nullptr)
        curr = curr->left;

    return curr->val;
}
```

Unbalanced Tree.

Best Case: $O(1)$

Worst Case: $O(n)$



Finding the Max and Min

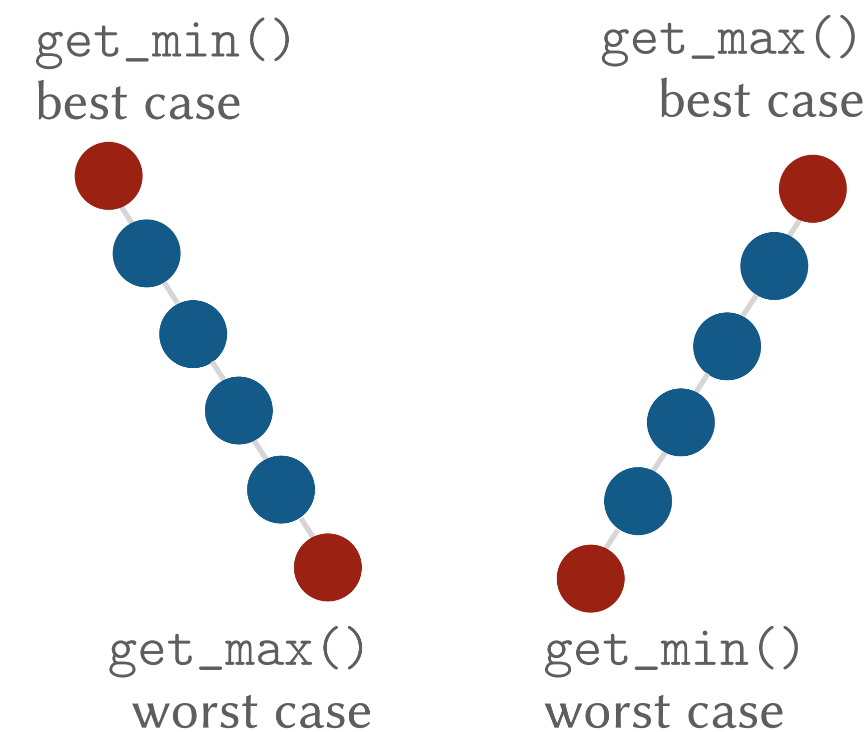
```cpp
template <class T>
T BST<T>::get_max() const {
    if (is_empty())
        throw string("No max in an empty tree");

    Node<T>* curr = root;
    while (curr->right != nullptr)
        curr = curr->right;

    return curr->val;
}
```

```cpp
template <class T>
T BST<T>::get_min() const {
    if (is_empty())
        throw string("No min in an empty tree");

    Node<T>* curr = root;
    while (curr->left != nullptr)
        curr = curr->left;

    return curr->val;
}
```

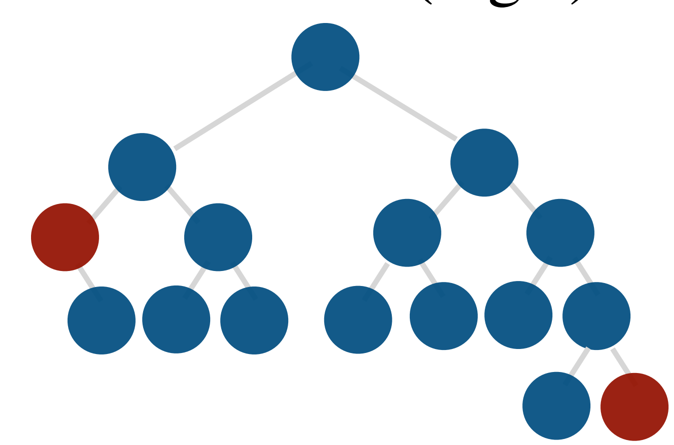
Unbalanced Tree.

Best Case: $O(1)$

Worst Case: $O(n)$



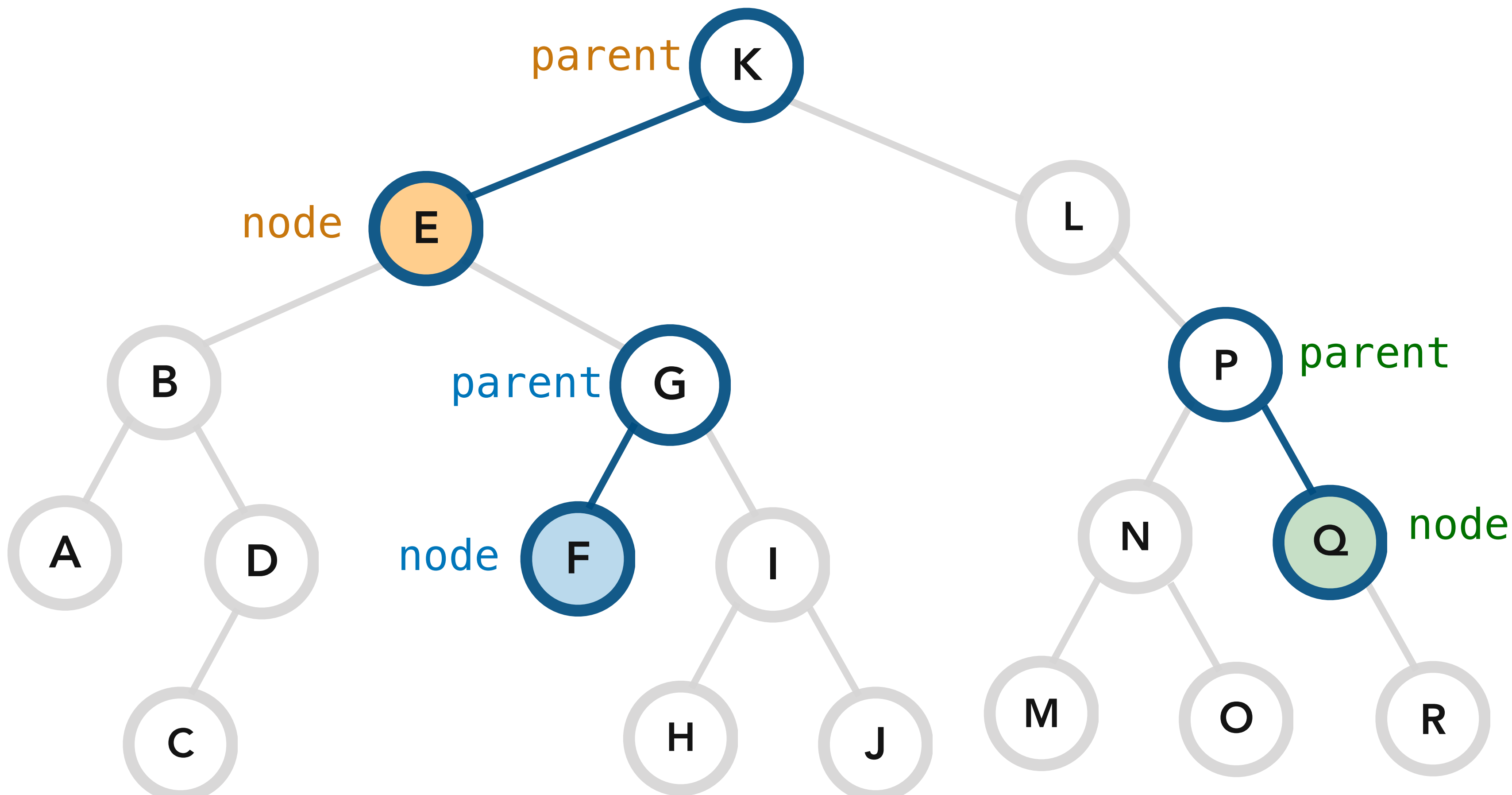
Balanced Tree. $O(\log n)$



The max and min are always at the lower levels.

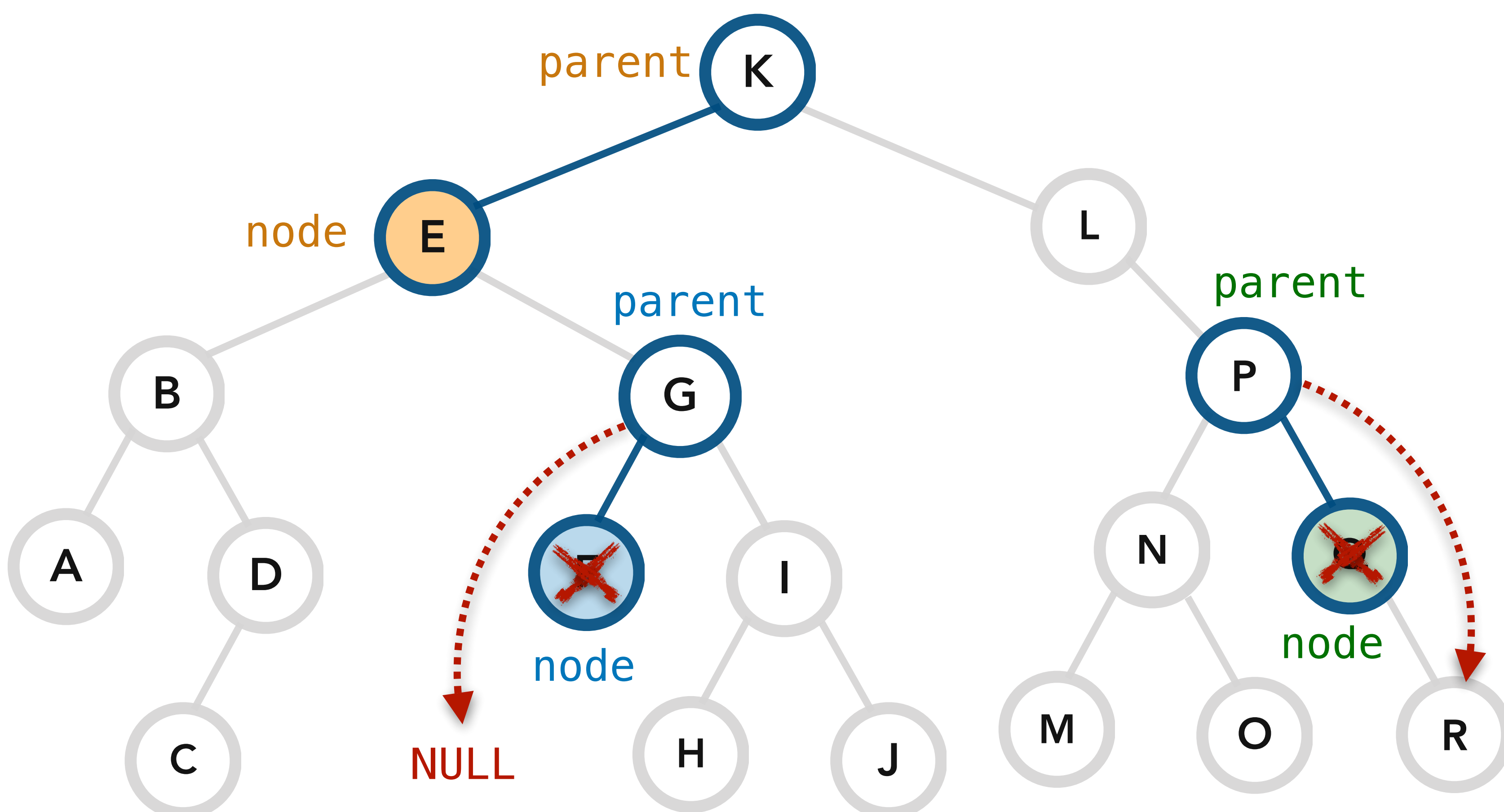
Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.



Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.



Easy cases to deal with!

Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right to `NULL`.

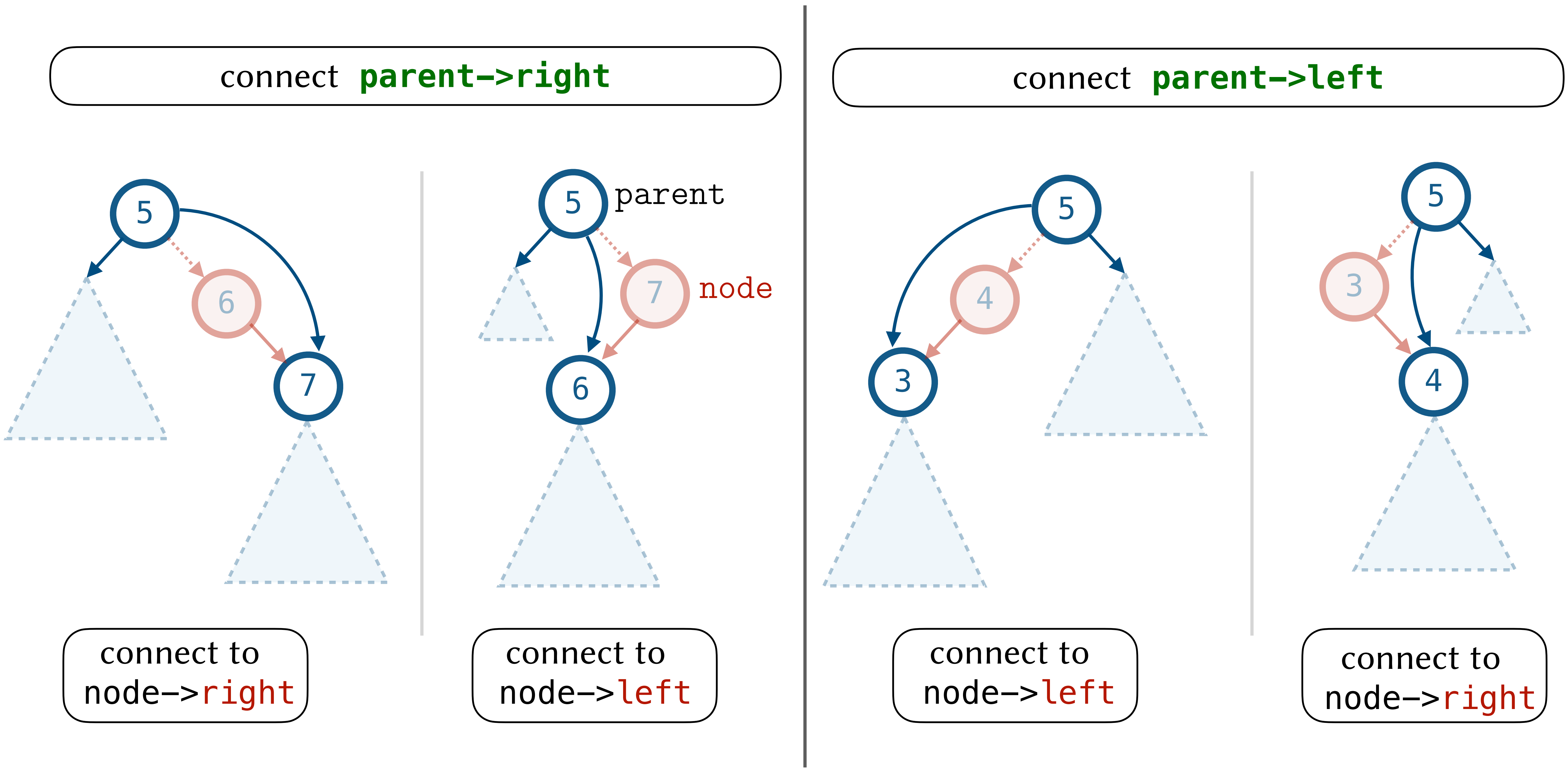
Case 2. If the `node` has one child: connect its `parent`'s left or right to this child.

Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right to this child.



Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

if the deleted node is the `root`, update the `root` pointer, not the `parent` pointer

Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

if the `node` to be deleted is to the `right` of its `parent`.

Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

if the `node` to be deleted is to the `left` of its `parent`.

Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

delete the `node` once the `parent` and child have been connected

Deleting a Node With 0 or 1 Children

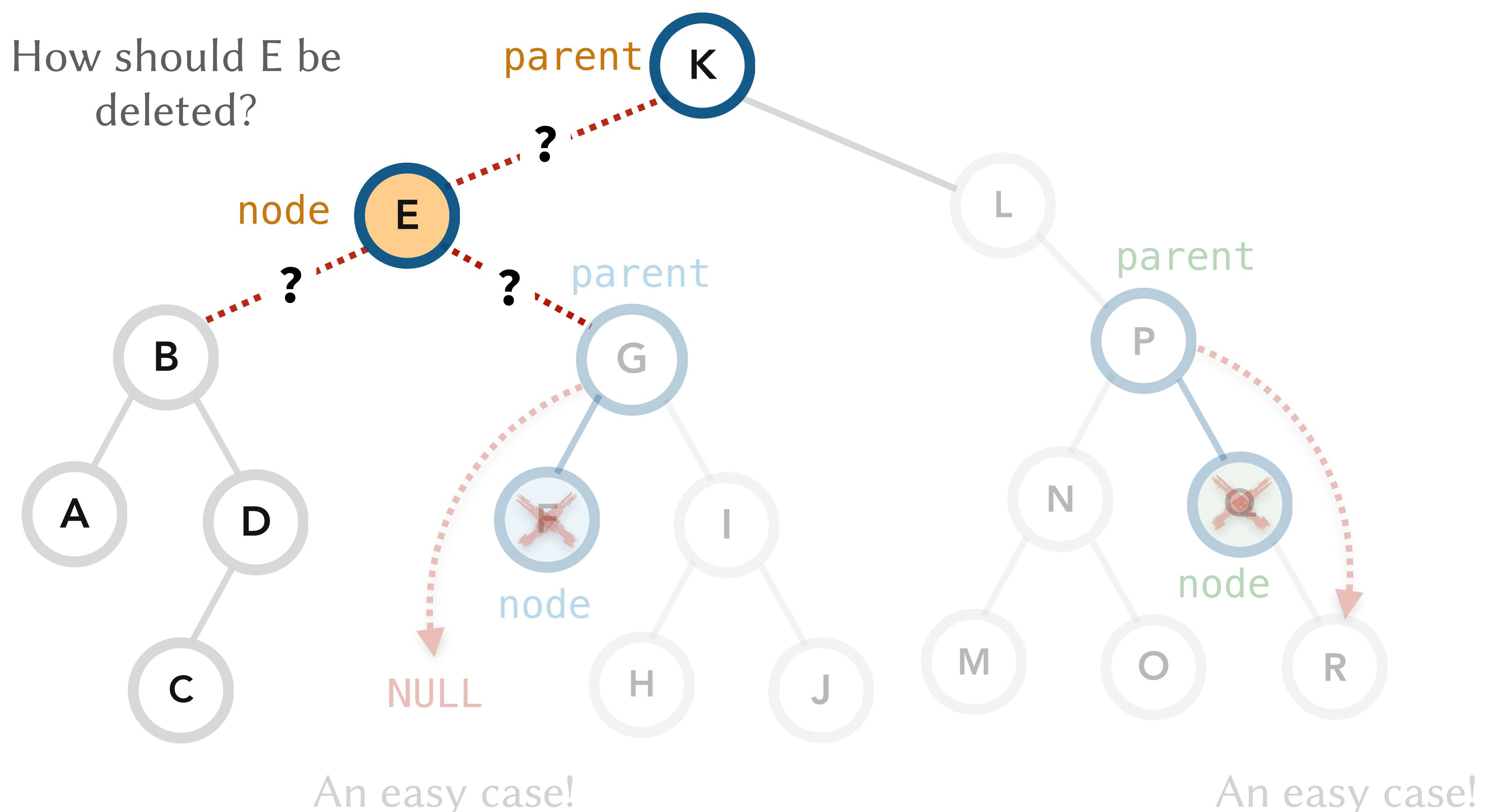
```
template<class T>
void BST<T>::remove_1(Node<T>* node, Node<T>* parent) {
    if (node == root) {
        if (node->left == nullptr)
            root = root->right;
        else
            root = root->left;
    }
    else if (node == parent->right) {
        if (node->right != nullptr)
            parent->right = node->right;
        else
            parent->right = node->left;
    }
    else {
        if (node->right != nullptr)
            parent->left = node->right;
        else
            parent->left = node->left;
    }

    delete node;
}
```

! Convince yourself.
This code handles correctly the case of `node` being a *leaf*.

! Running Time.
This code runs in O(1)

Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.



What if the node has *two* children?

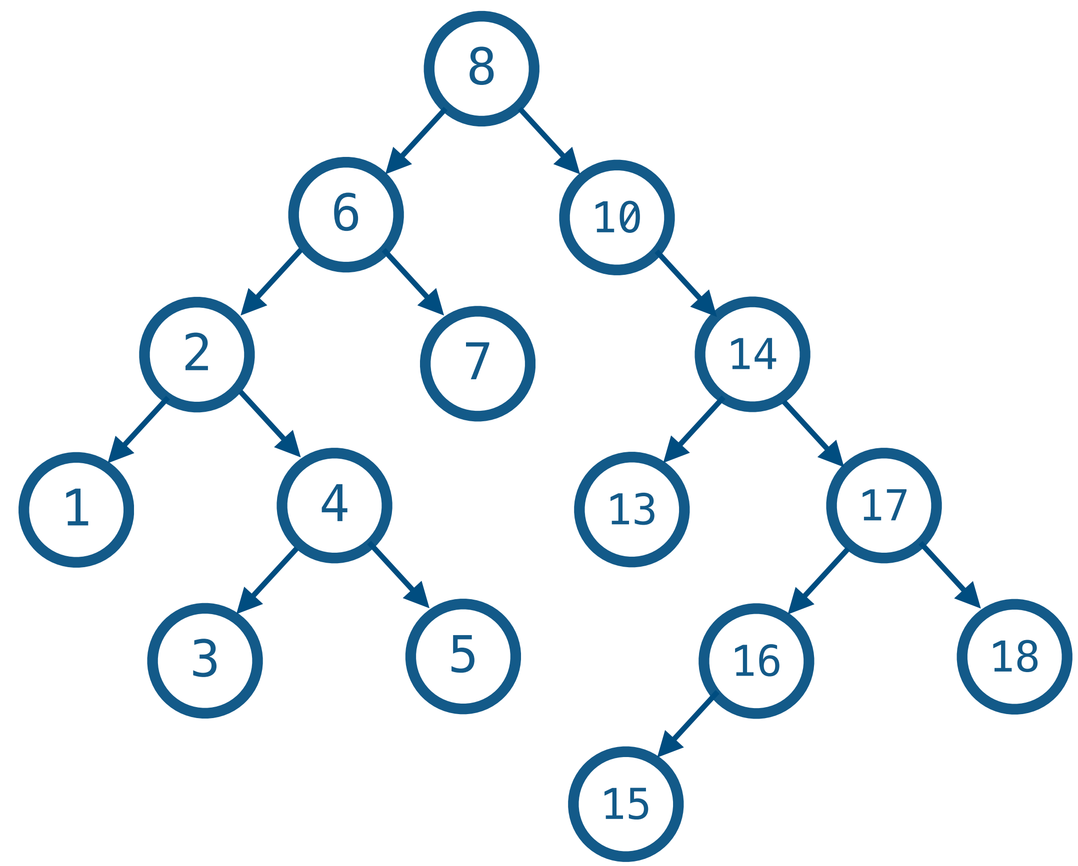
Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.



Deleting a Node

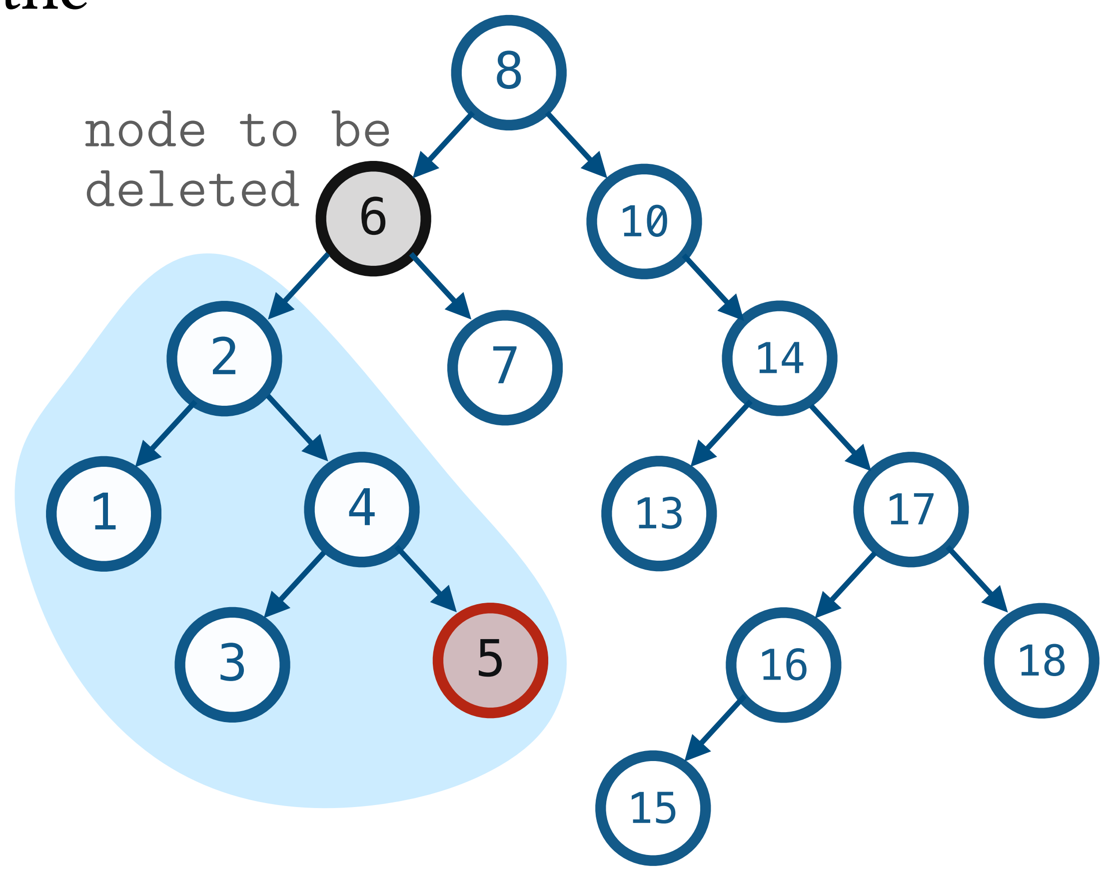
Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.

Idea. (1) Replace the `node` to be deleted with the *max* in its *left* subtree



5 is the max in 6's left subtree.

5 can replace 6 and the tree would remain a BST

Deleting a Node

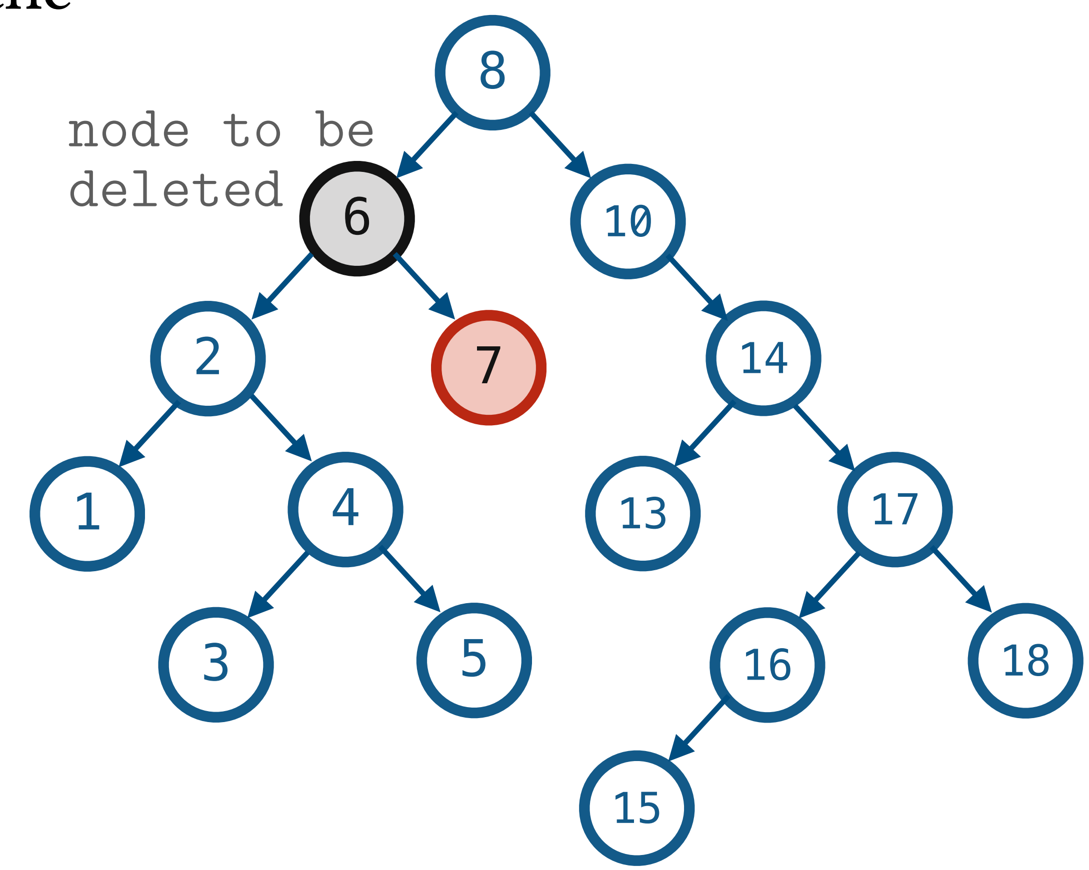
Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.

Idea. (1) Replace the `node` to be deleted with the *max* in its *left* subtree or with the *min* in its *right* subtree.



7 is the min in 6's right subtree.

7 can also replace 6 .

Deleting a Node

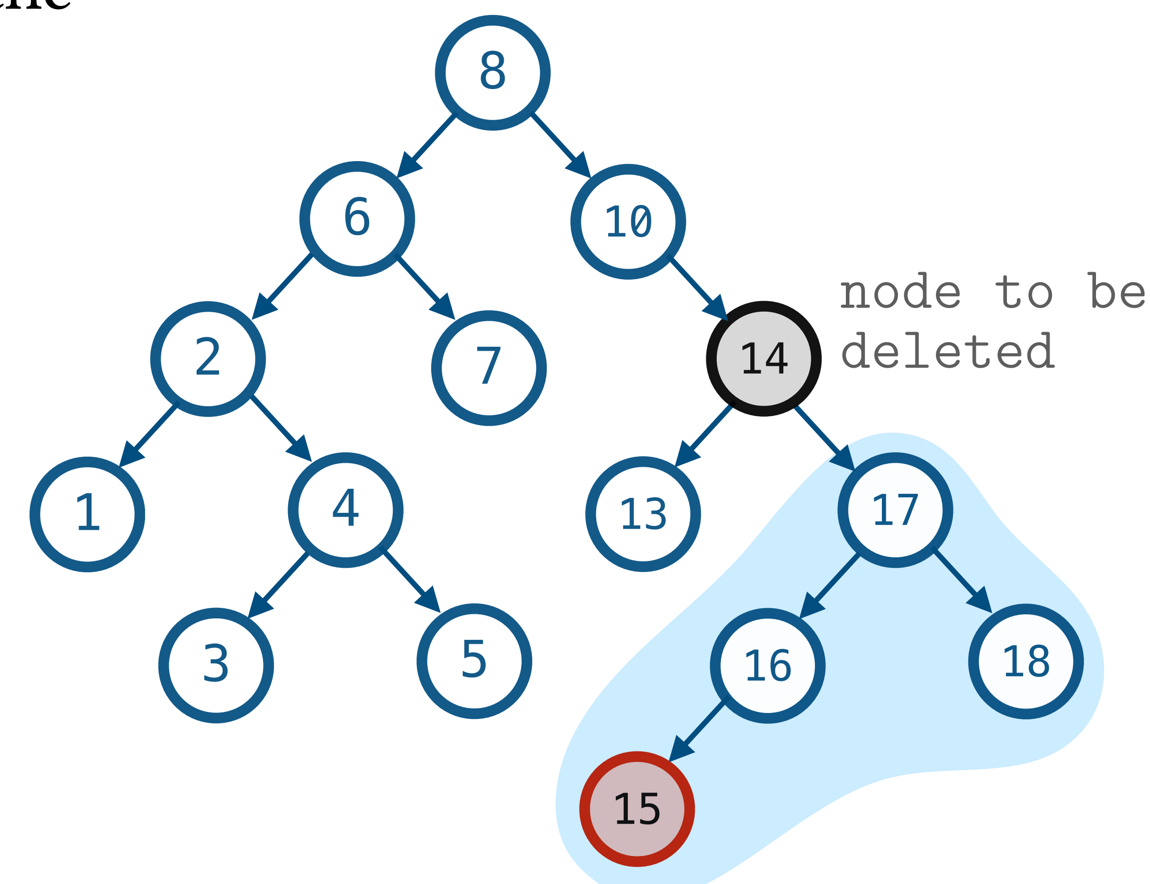
Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.

Idea. (1) Replace the `node` to be deleted with the *max* in its *left* subtree or with the *min* in its *right* subtree.



15 is the min in 14's right subtree.
15 can replace 14 and the tree would remain a BST

Deleting a Node

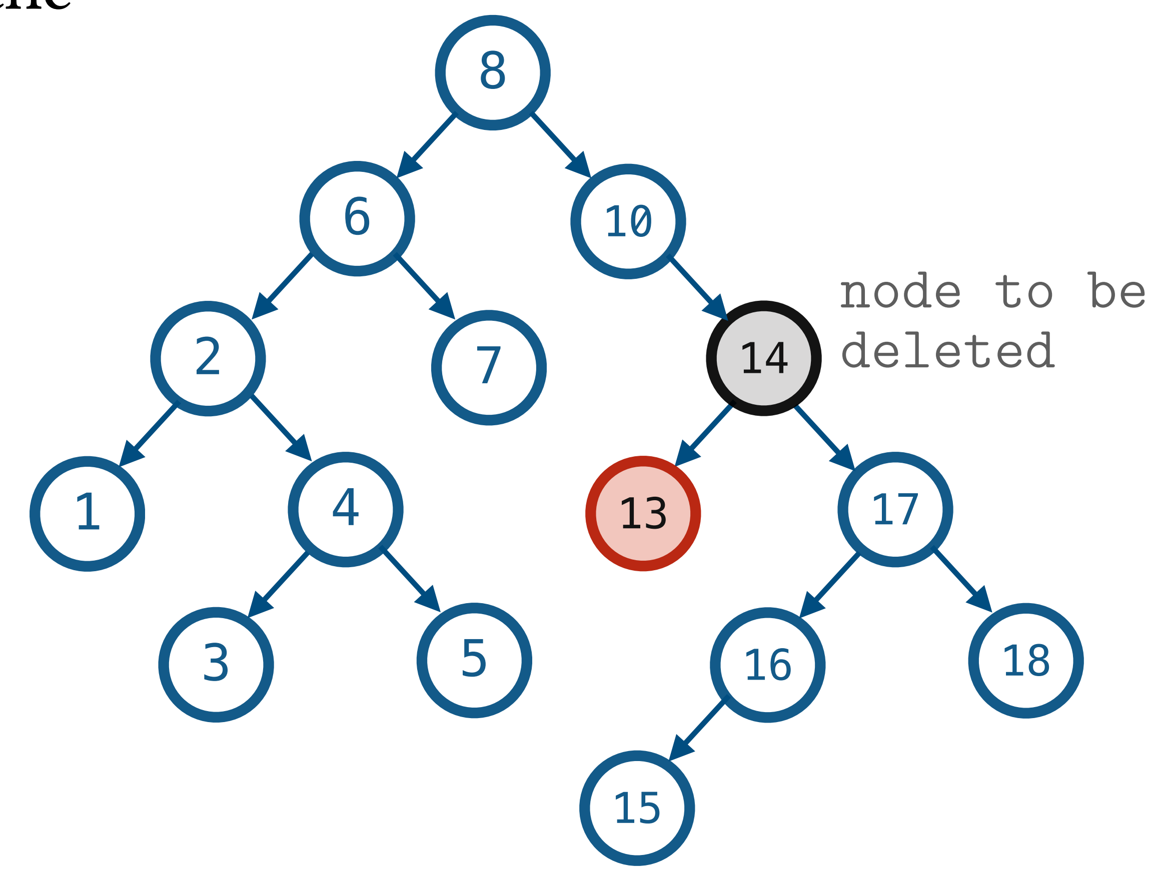
Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.

Idea. (1) Replace the `node` to be deleted with the *max* in its *left* subtree or with the *min* in its *right* subtree.



13 is the max in 14's left subtree.
13 can also replace 14.

Deleting a Node

Problem. Given a pointer to a `node` and a pointer to its `parent`, delete the `node`.

Case 1. If the `node` is a leaf node: connect its `parent`'s left or right link to `NULL`.

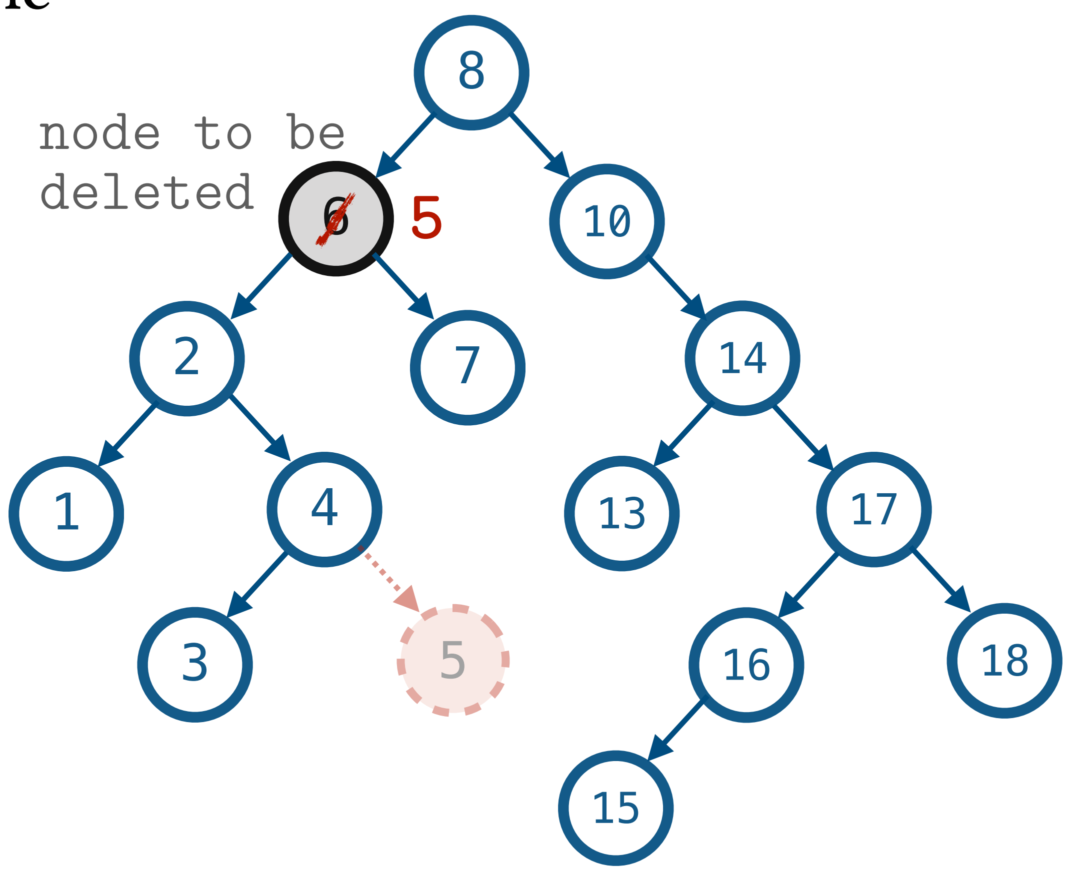
Case 2. If the `node` has one child: connect its `parent`'s left or right link to this child.

Case 3. If the `node` has 2 children: convert the problem to Case 1 or Case 2.

Idea. (1) Replace the `node` to be deleted with the *max* in its *left* subtree or with the *min* in its *right* subtree.

(2) Delete the replacement `node`.

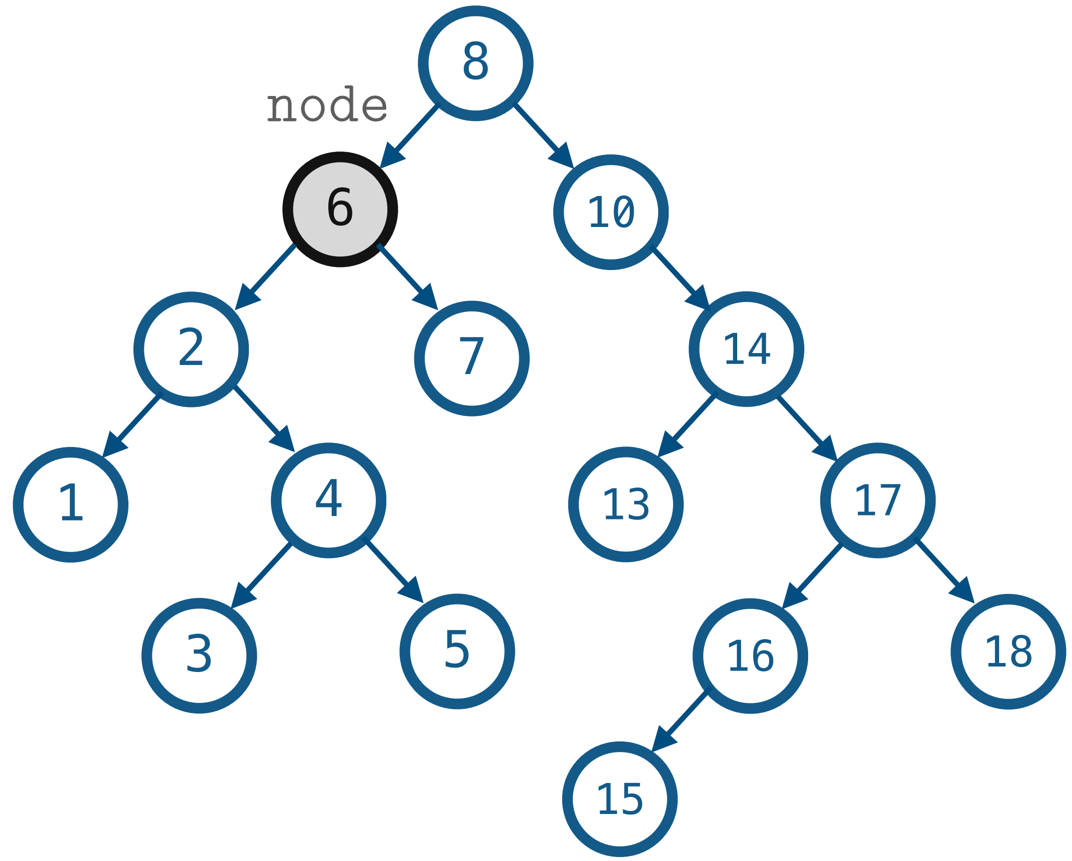
(guaranteed to have 0 or 1 children!)



replace 6 with 5 and delete 5

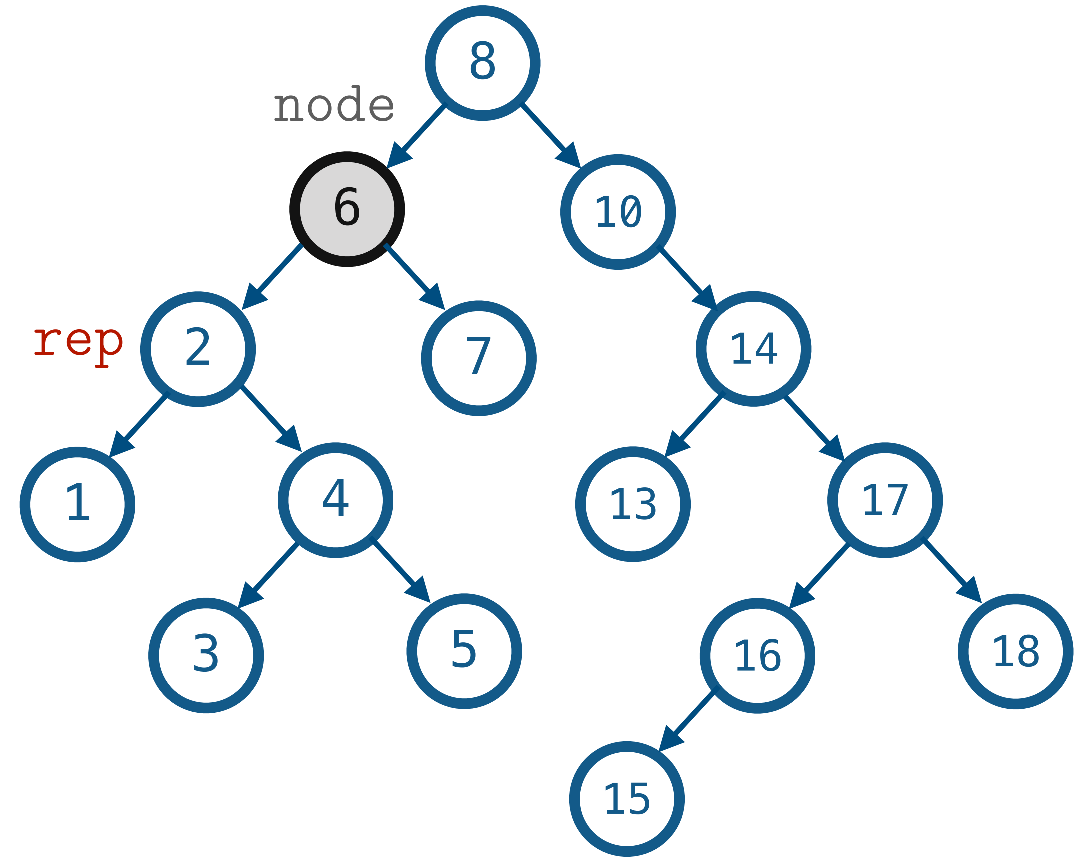
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



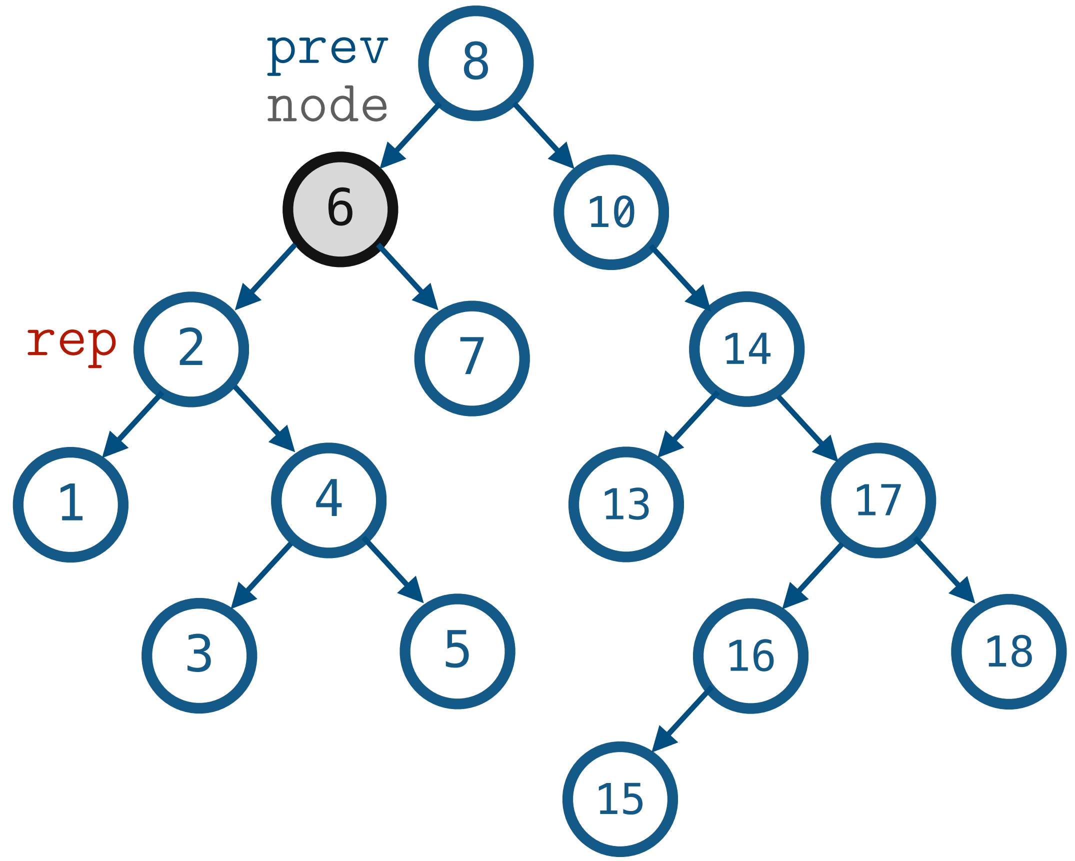
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
  • Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



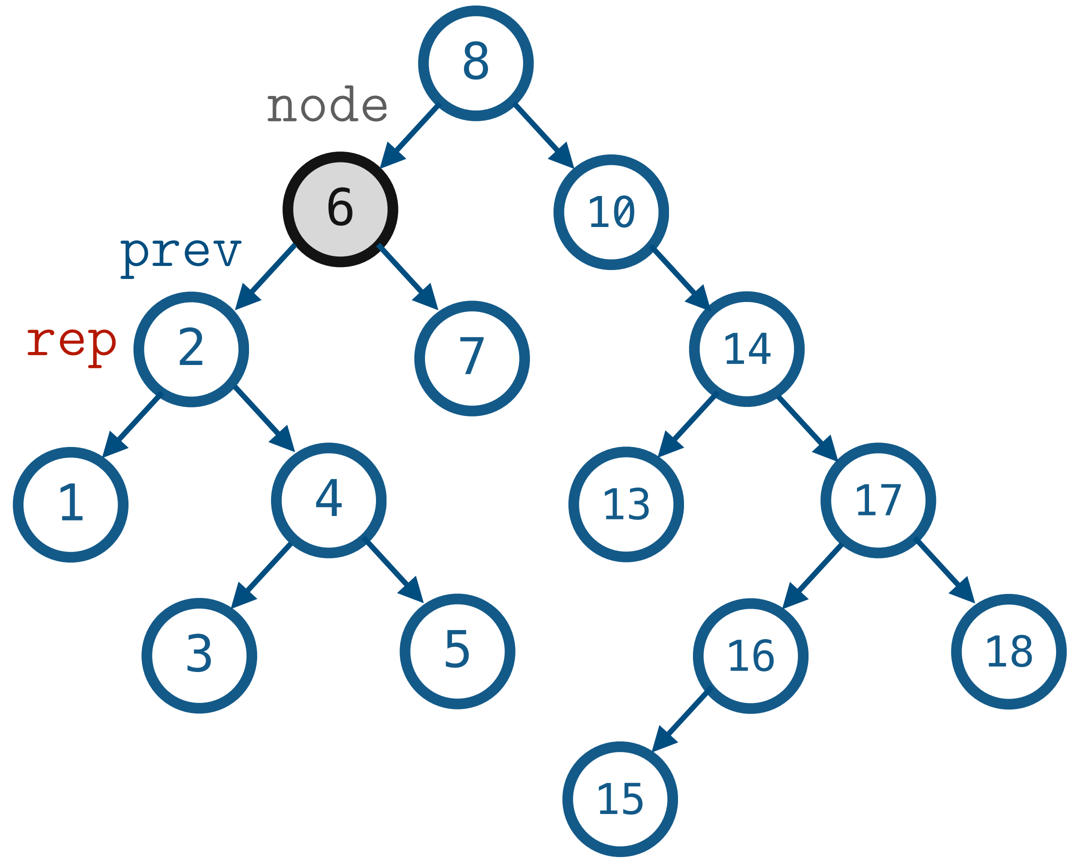
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



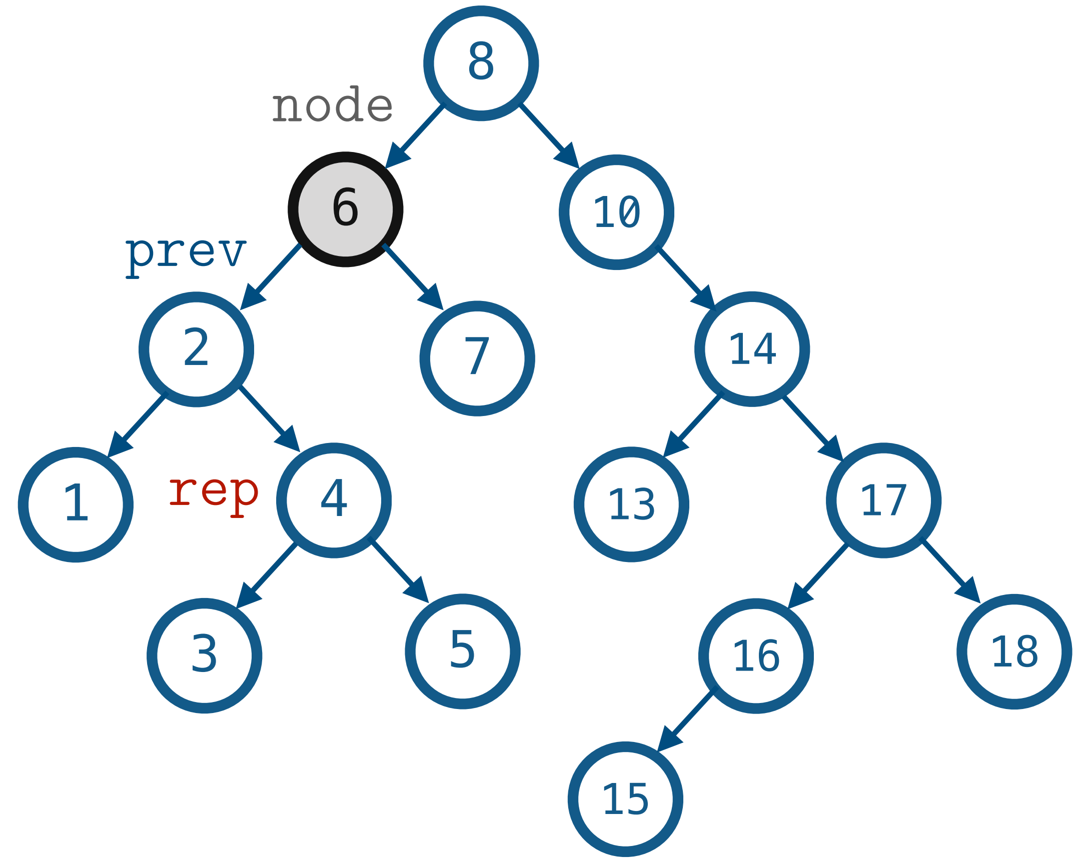
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



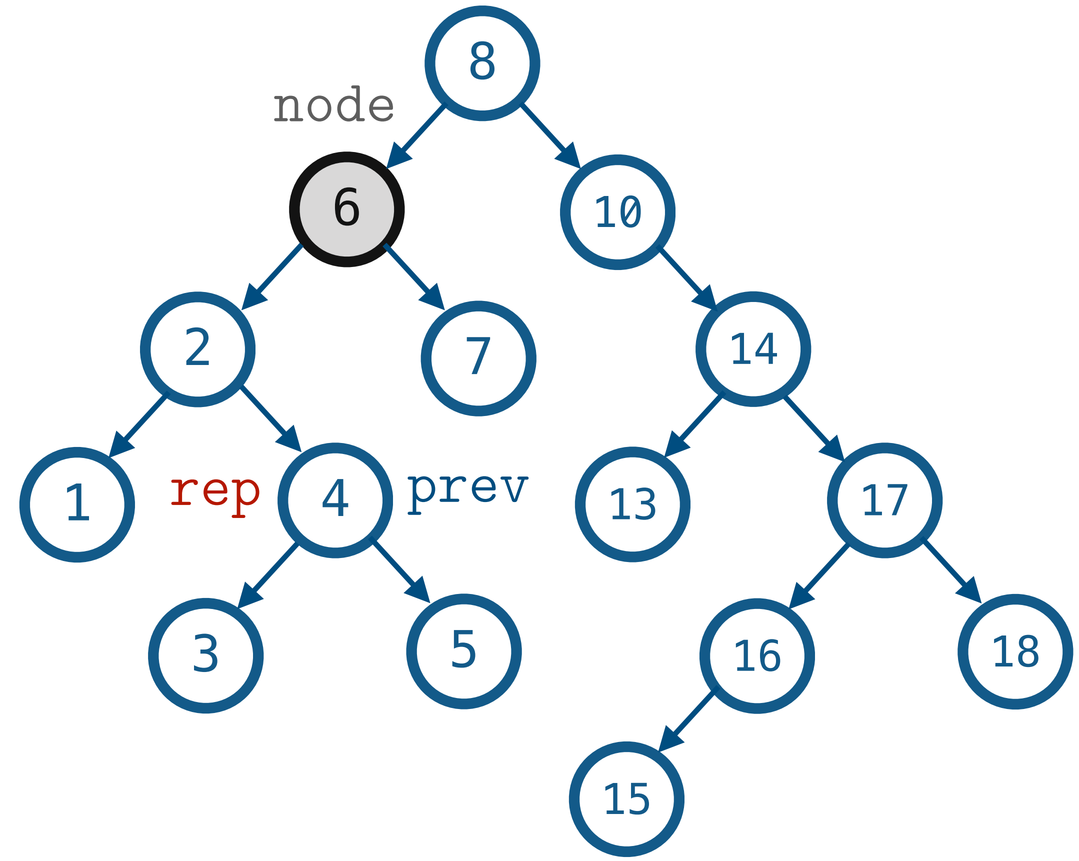
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



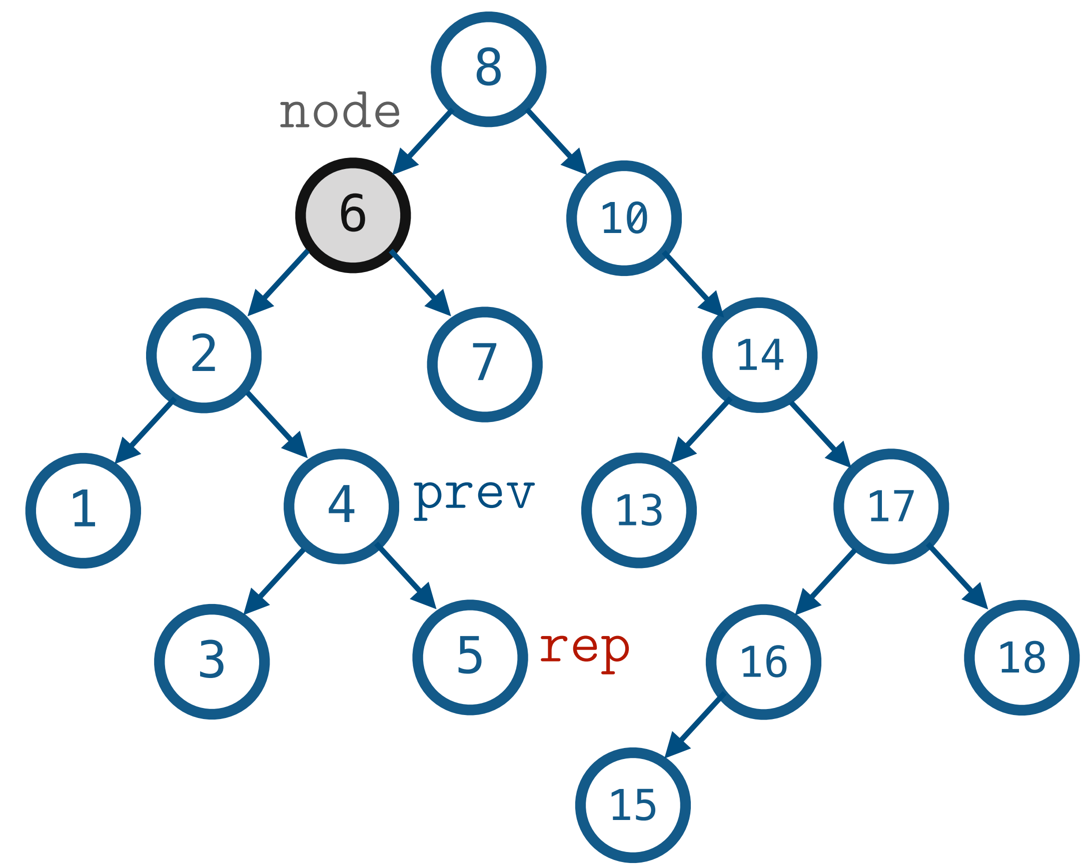
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



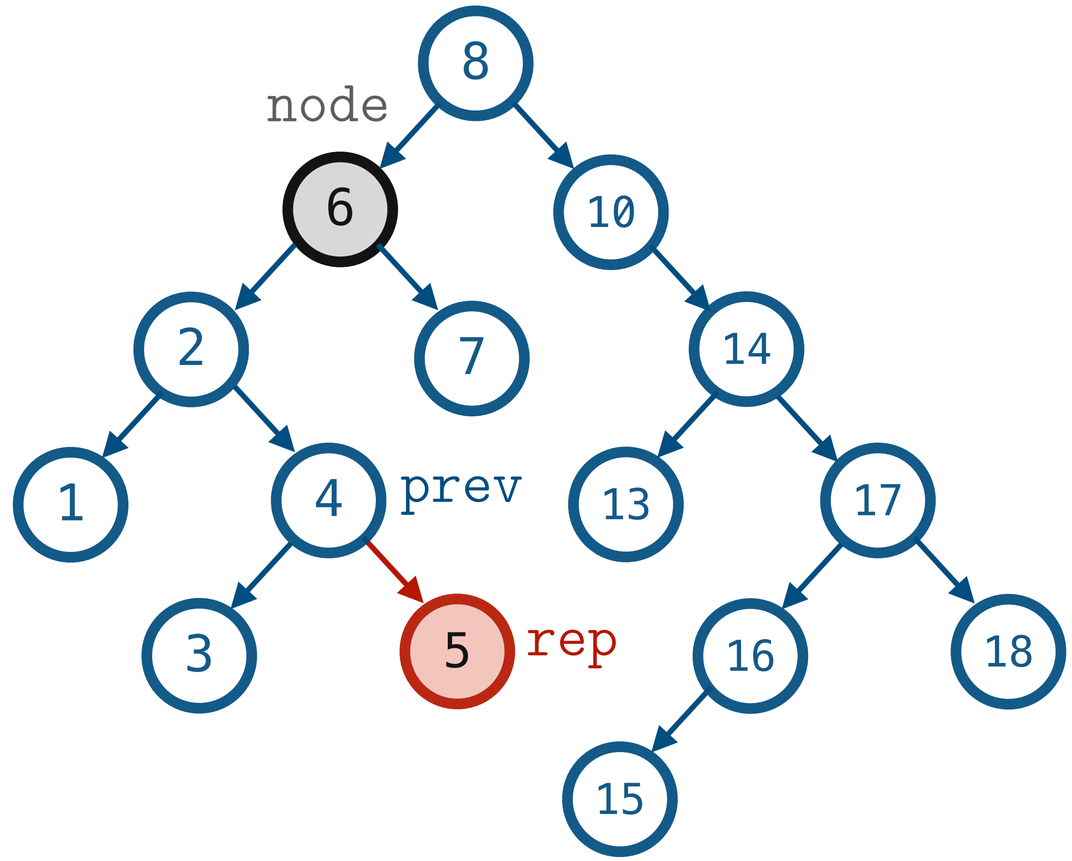
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



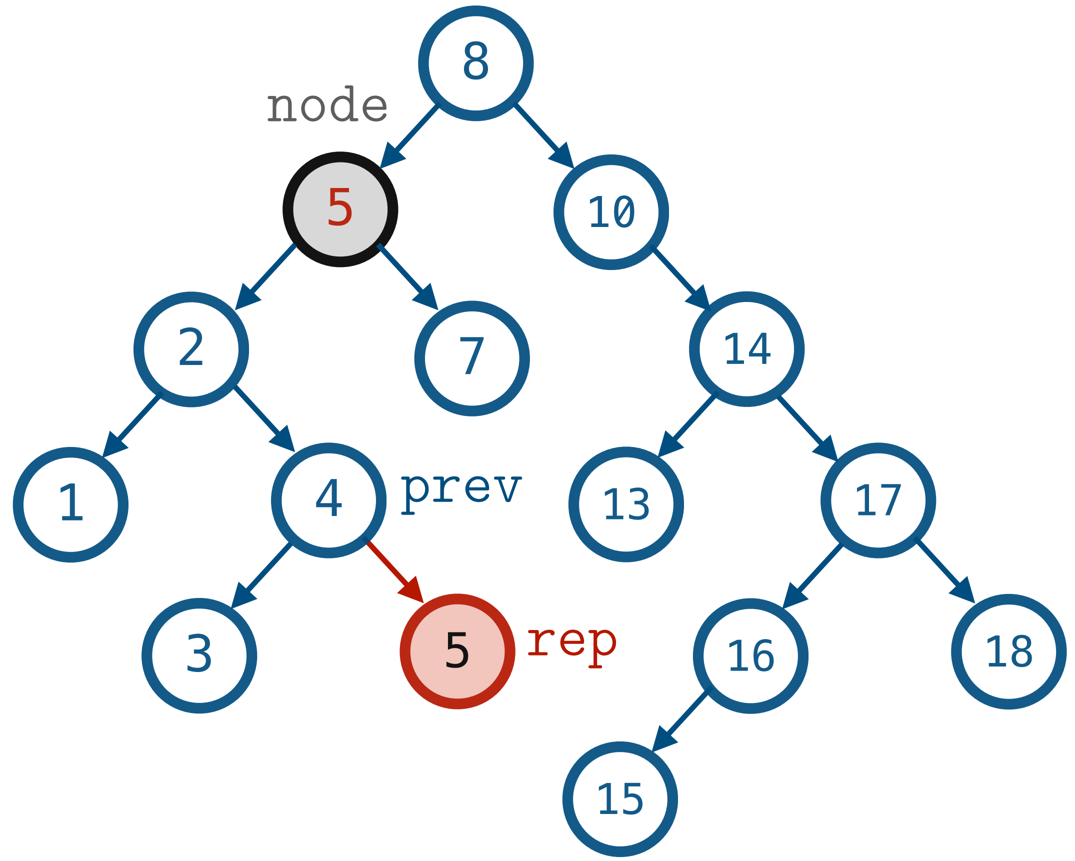
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



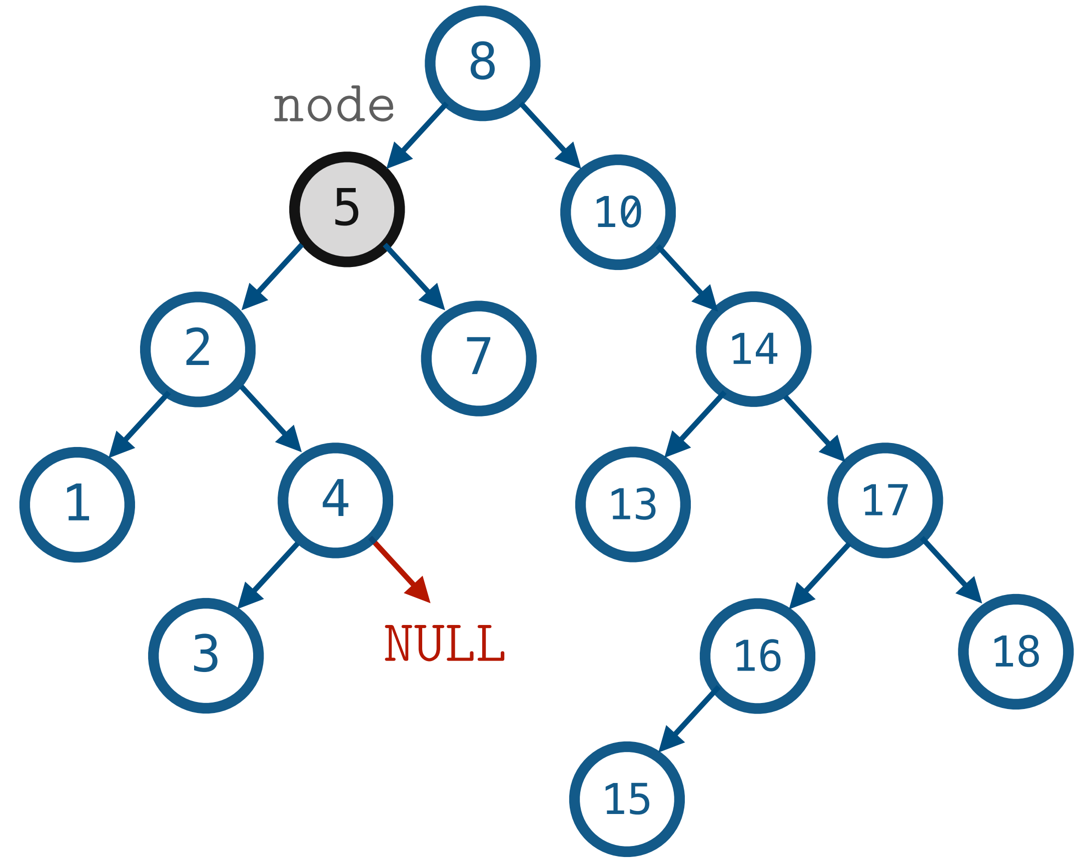
Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



Deleting a Node With 0 or 1 Children

```
template<class T>
void BST<T>::remove_2(Node<T>* node) {
    Node<T>* rep = node->left;
    Node<T>* prev = node;
    while (rep->right != nullptr) {
        prev = rep;
        rep = rep->right;
    }

    node->val = rep->val;
    remove_1(rep, prev);
}
```



Deleting a Node

```
template<class T>
bool BST<T>::remove(const T& val) {
    Node<T>* prev = nullptr;
    Node<T>* curr = root;

    while (curr != nullptr && curr->val != val) {
        prev = curr;
        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    if (curr == nullptr)
        return false;

    if (curr->left != nullptr && curr->right != nullptr)
        remove_2(curr);
    else
        remove_1(curr, prev);

    return true;
}
```

finds the node with the given `val` and deletes it.

Deleting a Node

```
template<class T>
bool BST<T>::remove(const T& val) {
    Node<T>* prev = nullptr;
    Node<T>* curr = root;

    while (curr != nullptr && curr->val != val) {
        prev = curr;
        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    if (curr == nullptr)
        return false;

    if (curr->left != nullptr && curr->right != nullptr)
        remove_2(curr);
    else
        remove_1(curr, prev);

    return true;
}
```

search for the `node` with the given `val`

Deleting a Node

```
template<class T>
bool BST<T>::remove(const T& val) {
    Node<T>* prev = nullptr;
    Node<T>* curr = root;

    while (curr != nullptr && curr->val != val) {
        prev = curr;
        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    if (curr == nullptr)
        return false;

    if (curr->left != nullptr && curr->right != nullptr)
        remove_2(curr);
    else
        remove_1(curr, prev);

    return true;
}
```

no node in the tree contains `val`

Deleting a Node

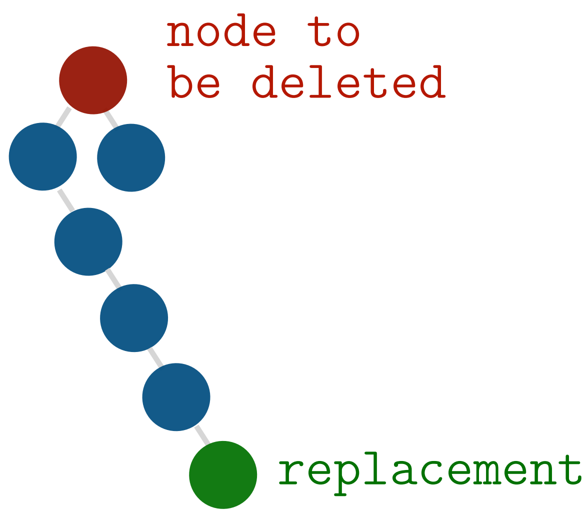
```
template<class T>
bool BST<T>::remove(const T& val) {
    Node<T>* prev = nullptr;
    Node<T>* curr = root;

    while (curr != nullptr && curr->val != val) {
        prev = curr;
        if (val > curr->val)
            curr = curr->right;
        else
            curr = curr->left;
    }

    if (curr == nullptr)
        return false;

    if (curr->left != nullptr && curr->right != nullptr)
        remove_2(curr);
    else
        remove_1(curr, prev);

    return true;
}
```

handle the case of 0/1 children and the case of 2 children separately

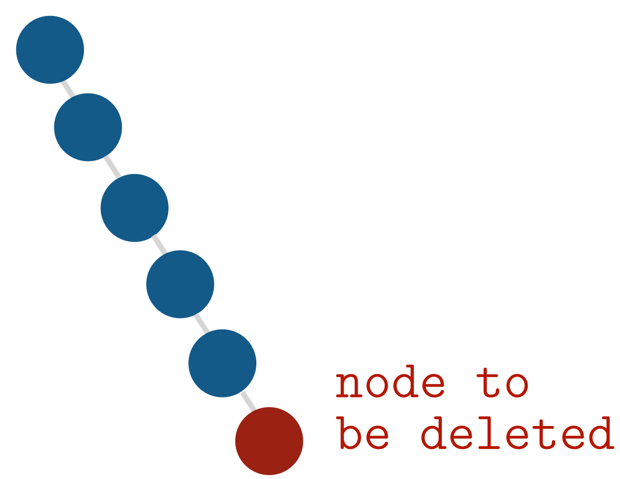
Running Time of Deleting a Node

Best Case:
$O(1)$ to *find* the node to be deleted + $O(1)$ to *delete* = $O(1)$

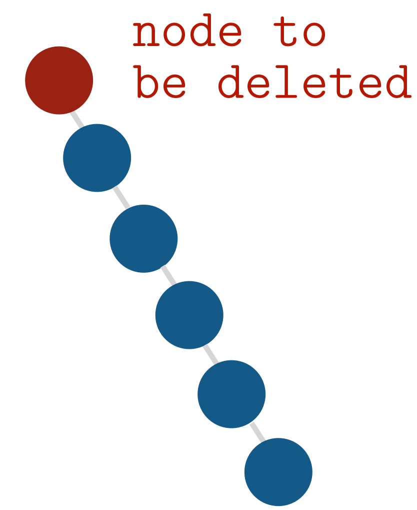
Worst Case:
$O(h)$ to *find* the node to be deleted or
$O(h)$ to *delete* or both.



Example of a worst case



Example of a worst case



Example of a best case

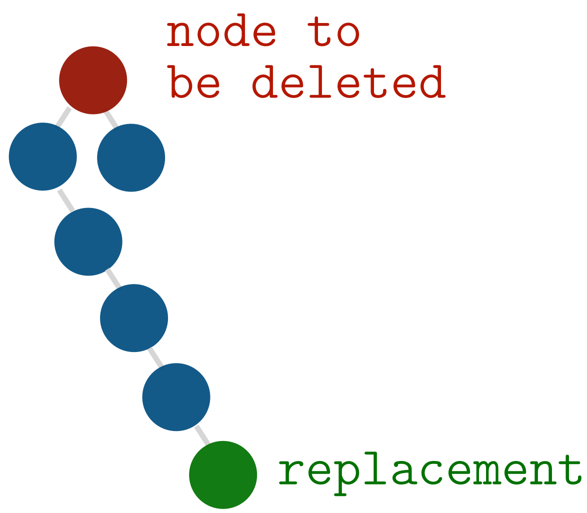
Running Time of Deleting a Node

Best Case:

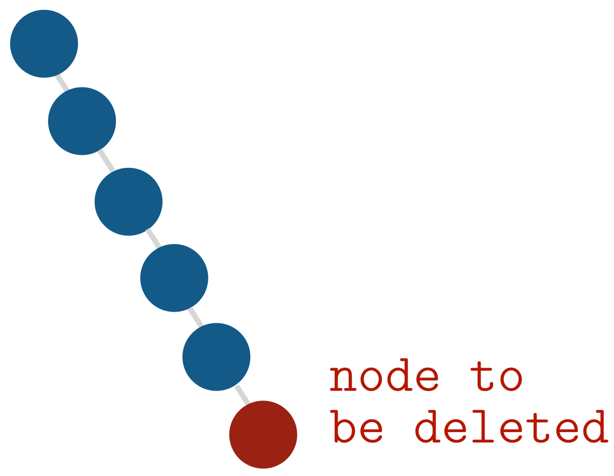
$O(1)$ to *find* the node to be deleted + $O(1)$ to *delete* = $O(1)$

Worst Case:

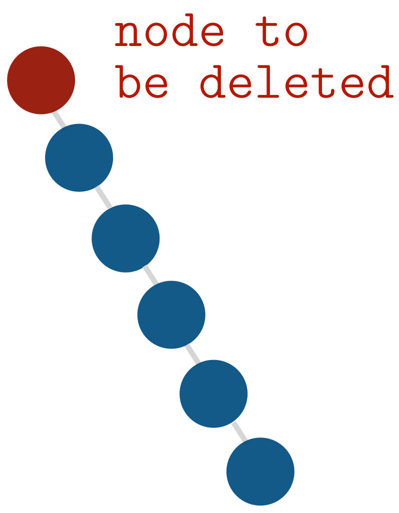
$O(h)$ to *find* the node to be deleted or
$O(h)$ to *delete* or both.



Example of a worst case



Example of a worst case



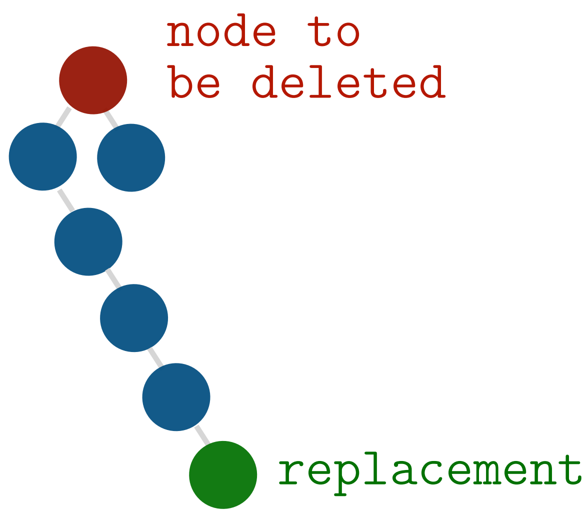
Example of a best case

Note that this best case does not happen if the tree is balanced.

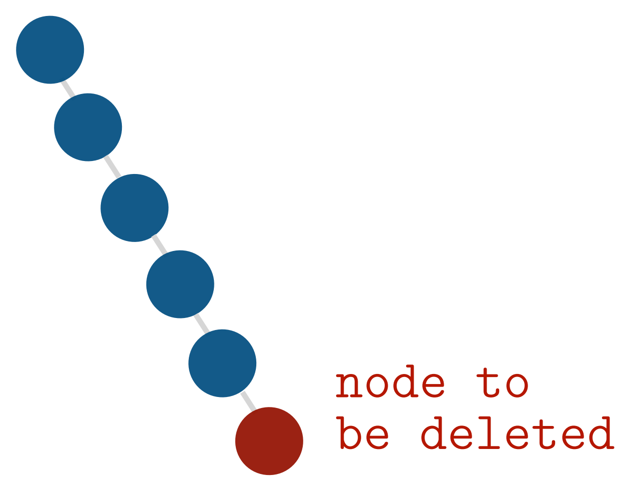
Running Time of Deleting a Node

Best Case:
$O(1)$ to *find* the node to be deleted + $O(1)$ to *delete* = $O(1)$

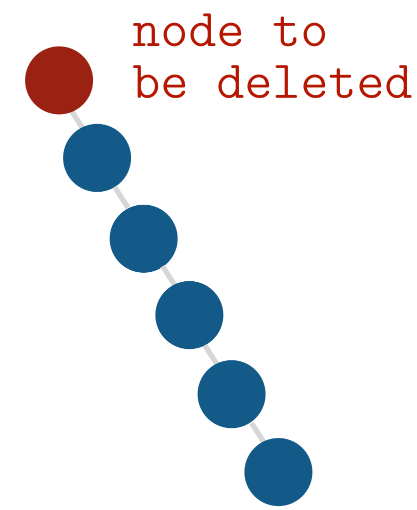
Worst Case:
$O(h)$ to *find* the node to be deleted or
$O(h)$ to *delete* or both.



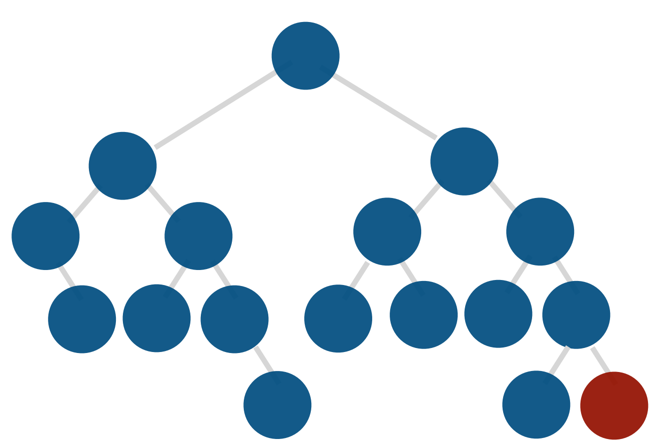
Example of a worst case



Example of a worst case



Example of a best case



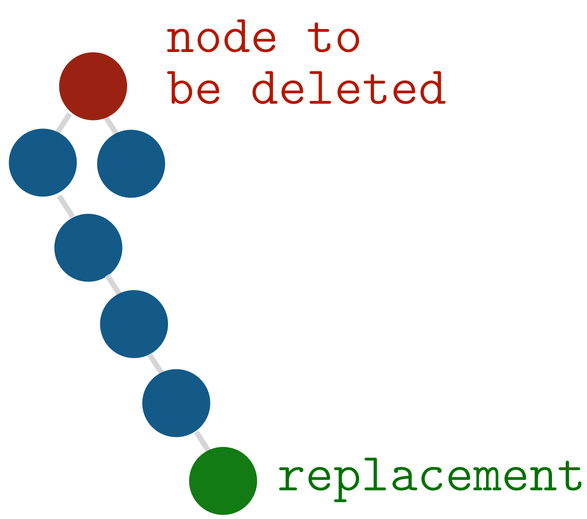
Note that this best case does not happen if the tree is balanced.

If the tree is balanced:
if $O(h)$ to search, then $O(1)$ to delete

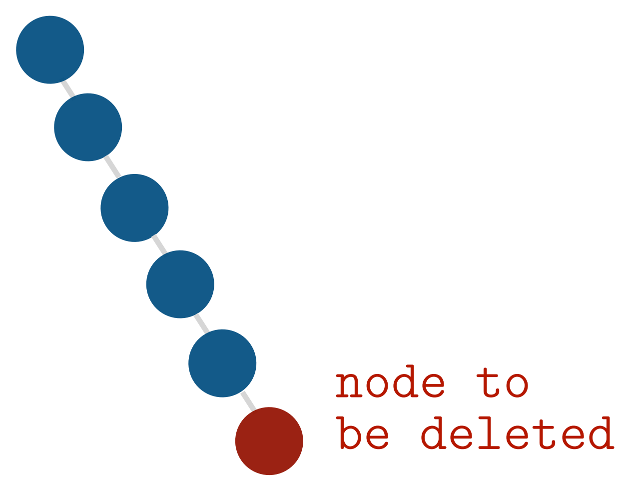
Running Time of Deleting a Node

Best Case:
$O(1)$ to *find* the node to be deleted + $O(1)$ to *delete* = $O(1)$

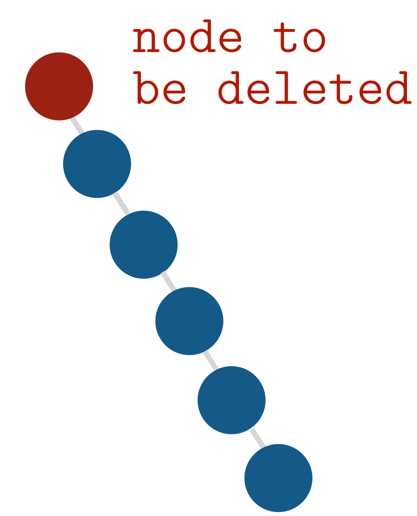
Worst Case:
$O(h)$ to *find* the node to be deleted or
$O(h)$ to *delete* or both.



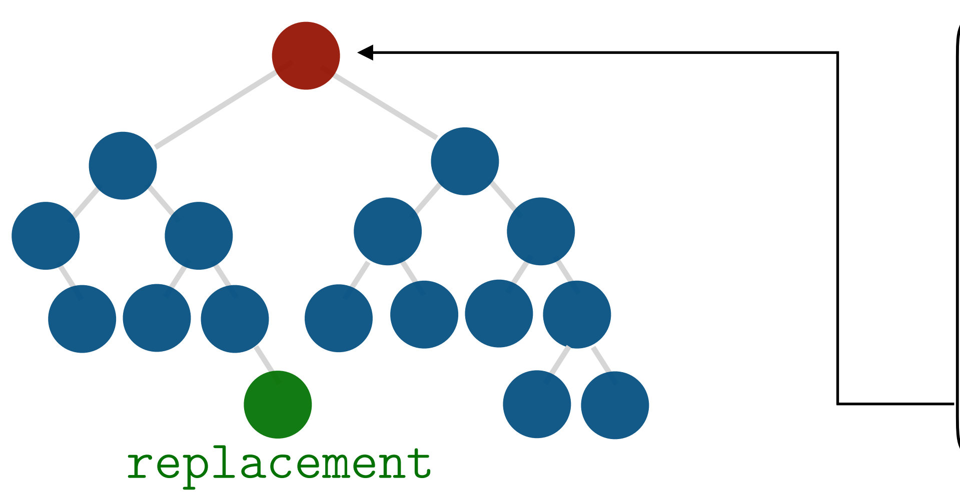
Example of a worst case



Example of a worst case

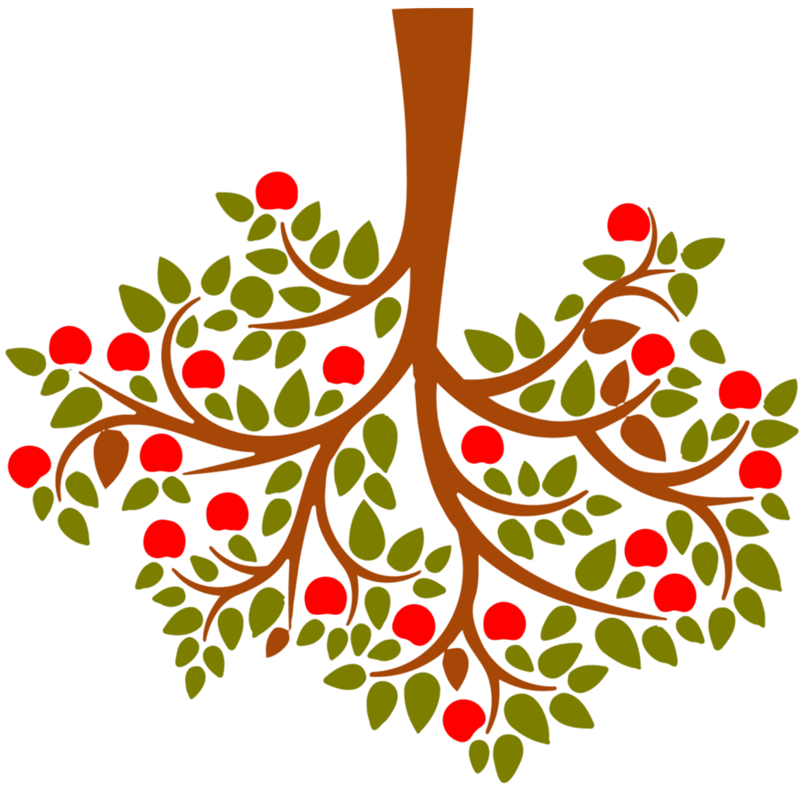


Example of a best case



Note that this best case does not happen if the tree is balanced.

If the tree is balanced:
if $O(h)$ to search, then $O(1)$ to delete
if $O(1)$ to search, then $O(h)$ to delete



Tree Data Structures

- Definitions and properties
- Basic operations
- • Balanced binary search trees
- Tree traversals

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Balanced Binary Search Trees

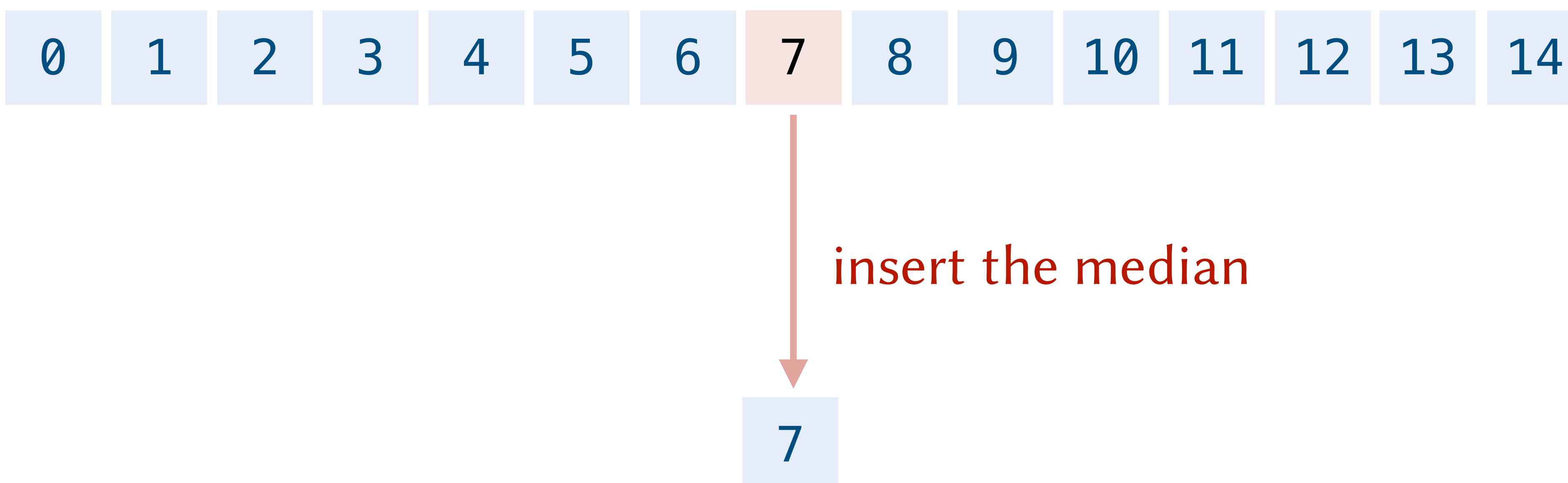
Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

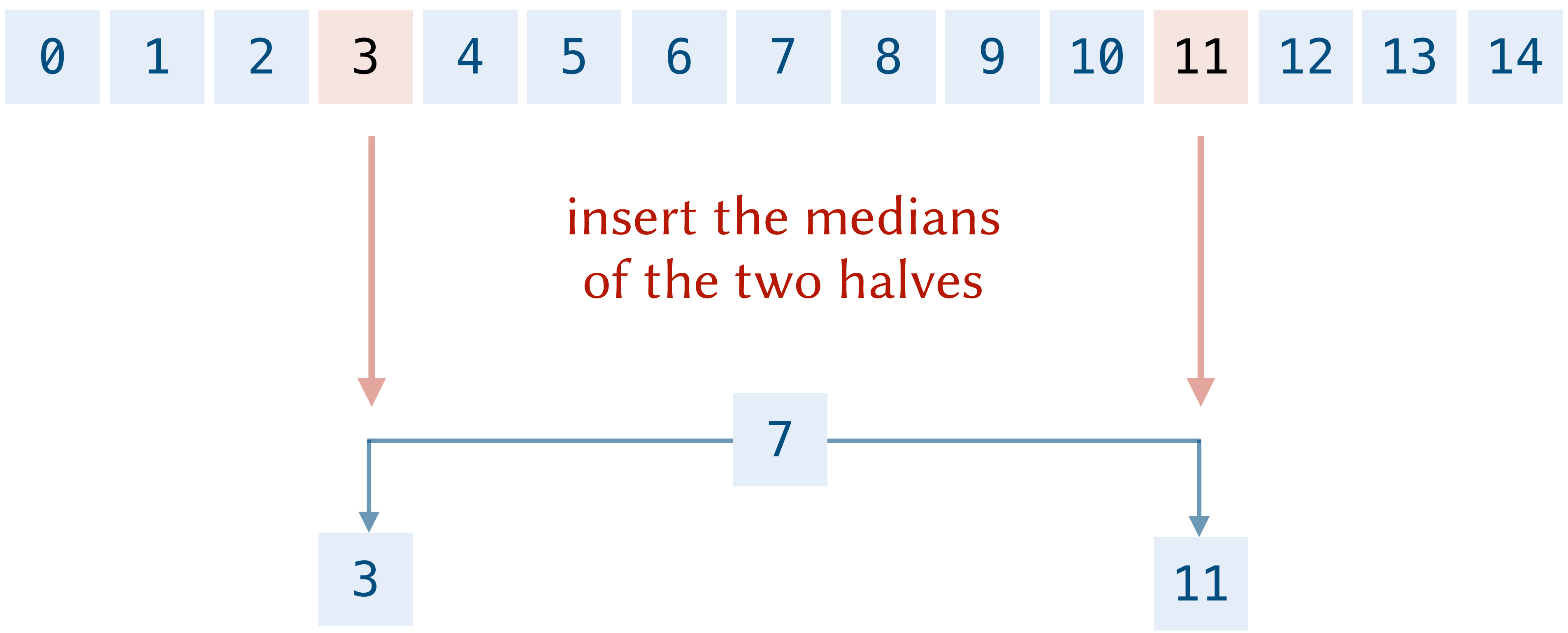
Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.



Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

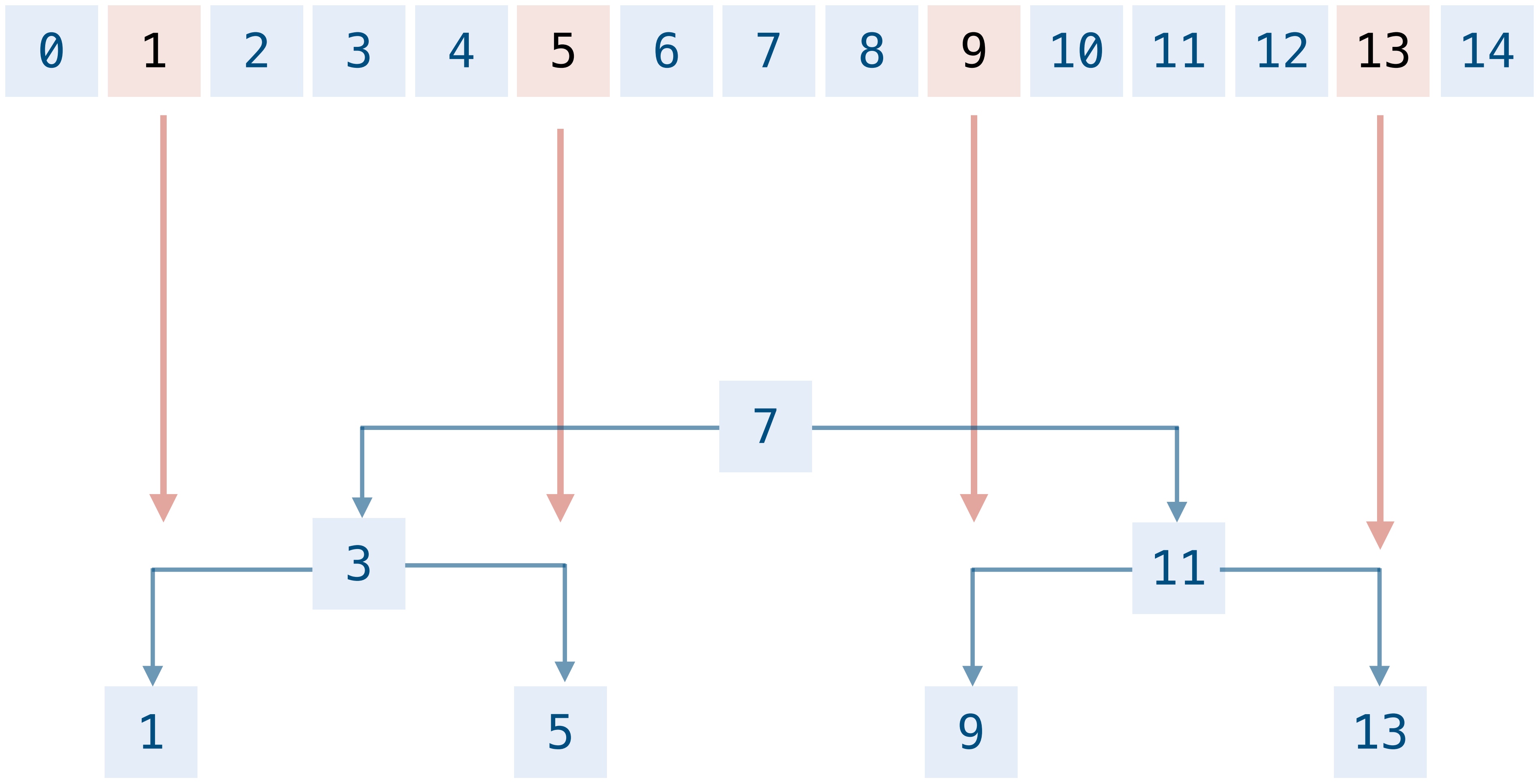
Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.



Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.



etc.

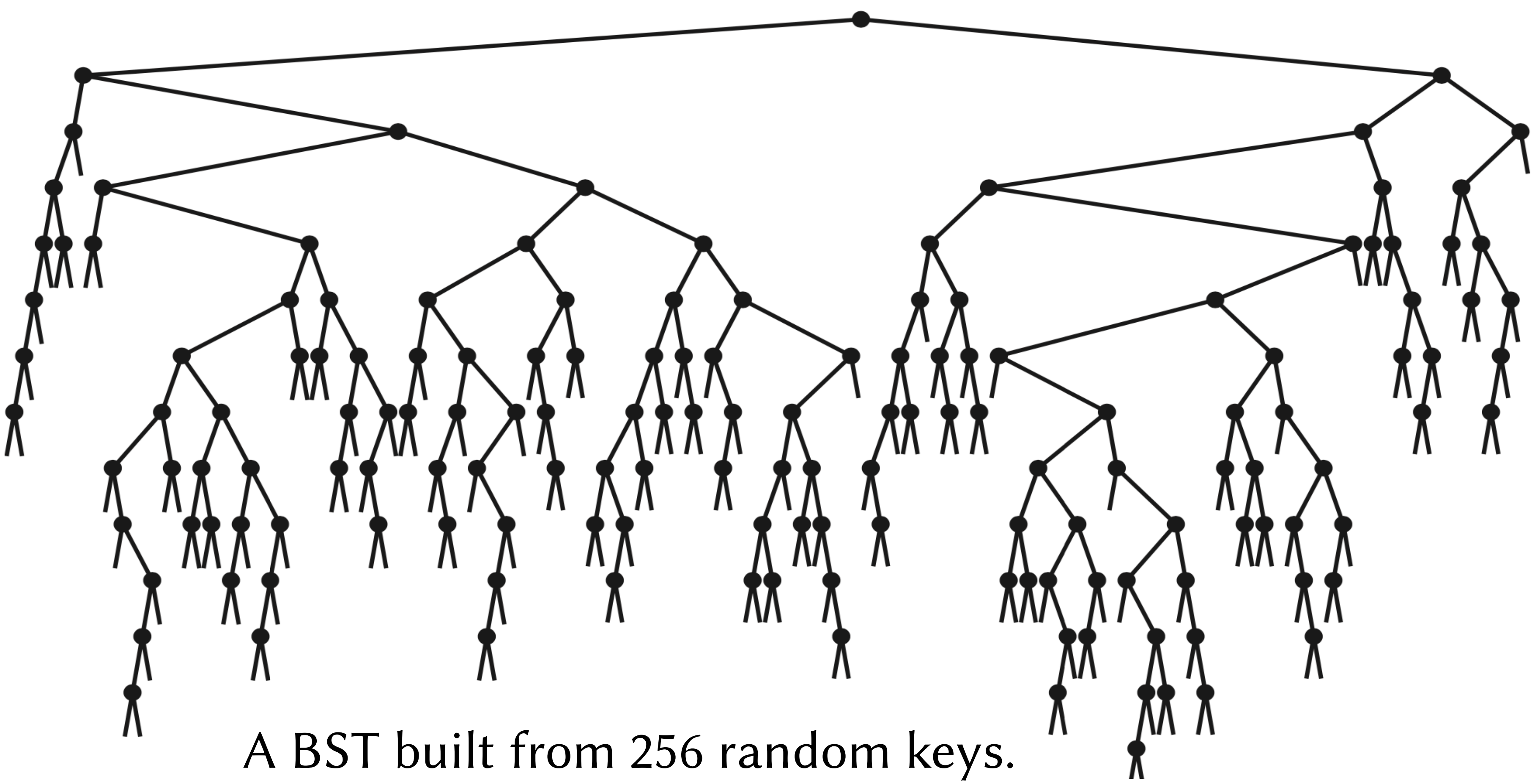
See the exercises for the code

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them!
The expected hight of the tree is $O(\log n)$



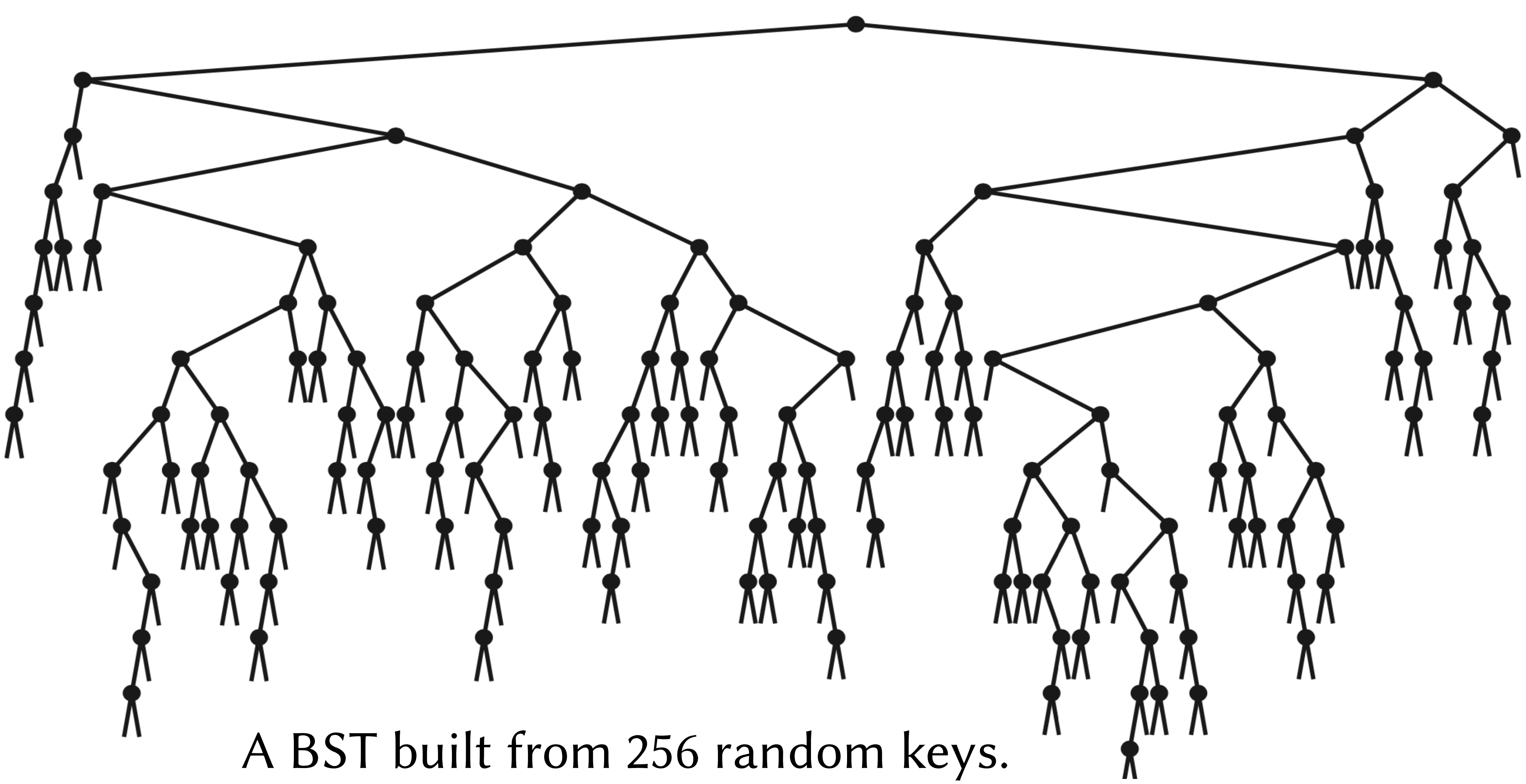
A BST built from 256 random keys.
(image by Sedgwick and Wayne)

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them!
The expected hight of the tree is $O(\log n)$ → Math skipped! See the Design & Analysis of Algorithms course!



A BST built from 256 random keys.
(image by Sedgwick and Wayne)

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them!
The expected hight of the tree is $O(\log n)$

Problem. How can we guarantee the tree is balanced after any insertion or deletion?

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them!
The expected hight of the tree is $O(\log n)$

Problem. How can we guarantee the tree is balanced after any insertion or deletion?

Solution 1. If the tree becomes misbalanced after an insert or delete operation, rebuild the whole tree using one of the above solutions!

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them!
The expected hight of the tree is $O(\log n)$

Problem. How can we guarantee the tree is balanced after any insertion or deletion?

Solution 1. If the tree becomes misbalanced after an insert or delete operation, rebuild the whole tree using one of the above solutions!

Waste of Time!

Balanced Binary Search Trees

Problem. Given n elements, how can we construct a balanced BST?

Solution 1. Sort the elements, and insert the median, then recursively do the same in the left and right halves.

Solution 2. Shuffle the elements randomly and then insert them! The expected hight of the tree is $O(\log n)$

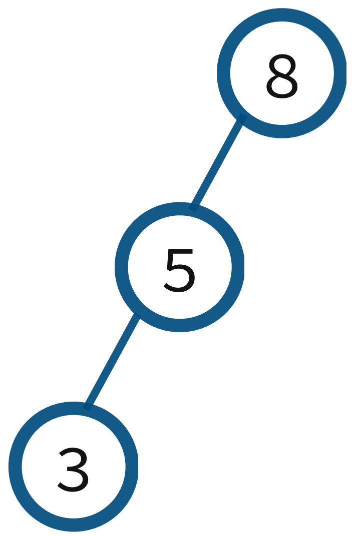
Problem. How can we guarantee the tree is balanced after any insertion or deletion?

Solution 1. If the tree becomes misbalanced after an insert or delete operation, rebuild the whole tree using one of the above solutions!

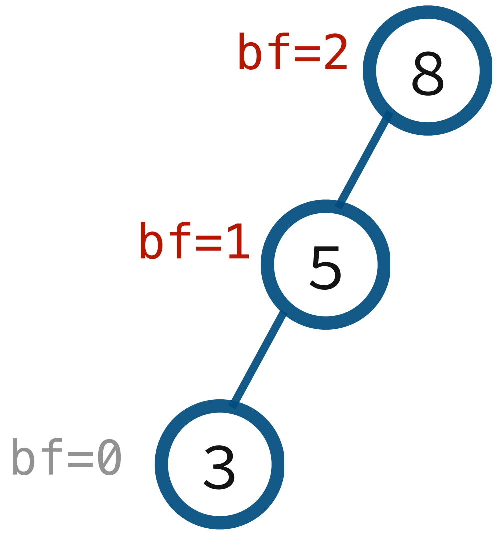
Waste of Time!

Solution 2. Rebalance only the affected part of the tree using rotations!

Self-Balancing BSTs: Tree Rotations

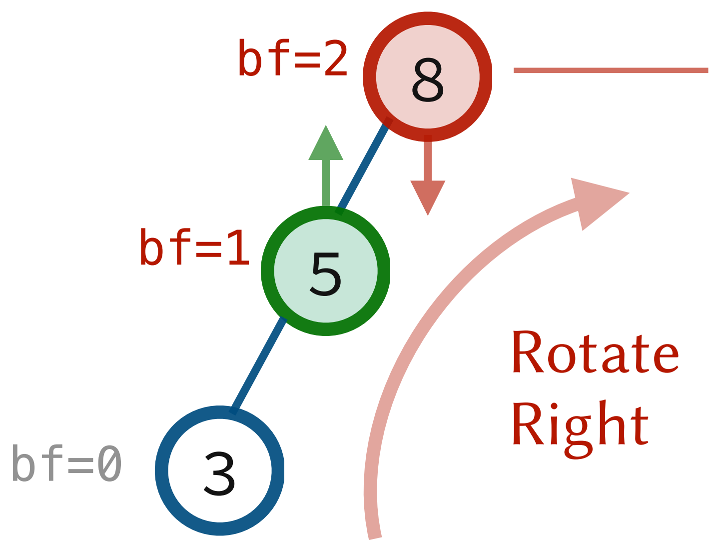


Self-Balancing BSTs: Tree Rotations



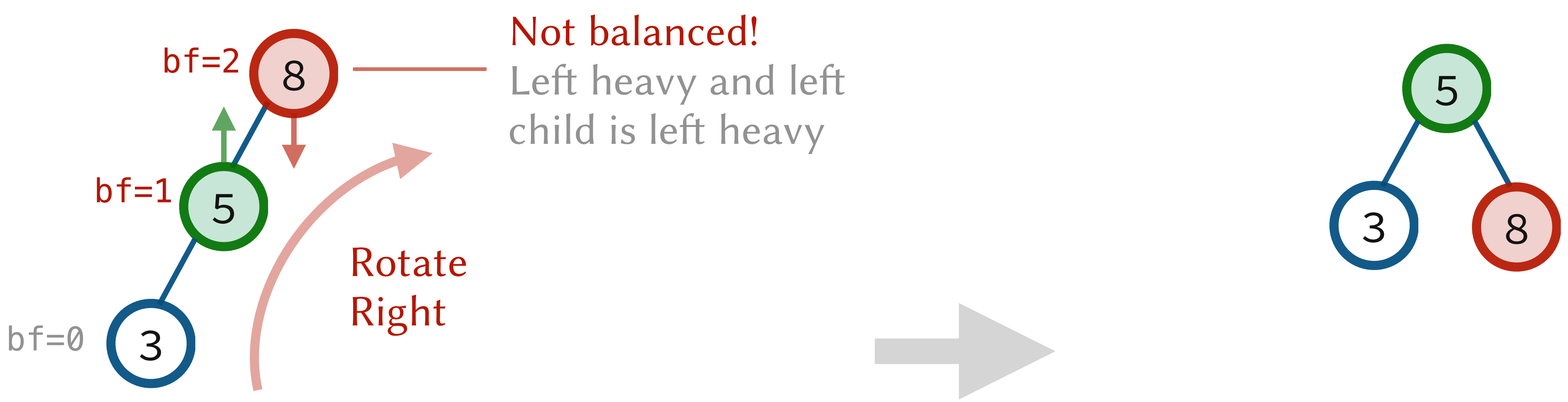
Not balanced!
Left heavy and left child is left heavy

Self-Balancing BSTs: Tree Rotations

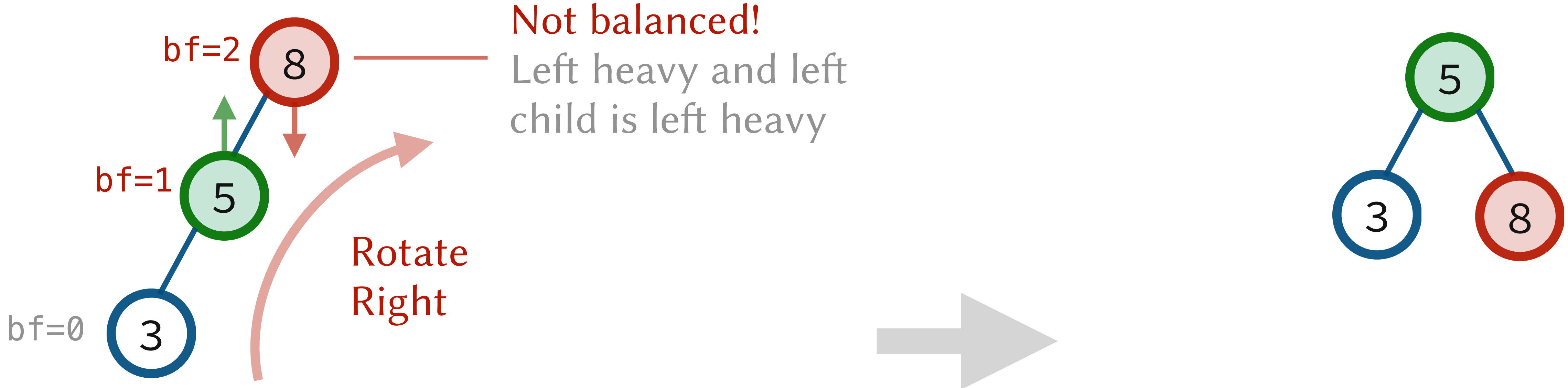


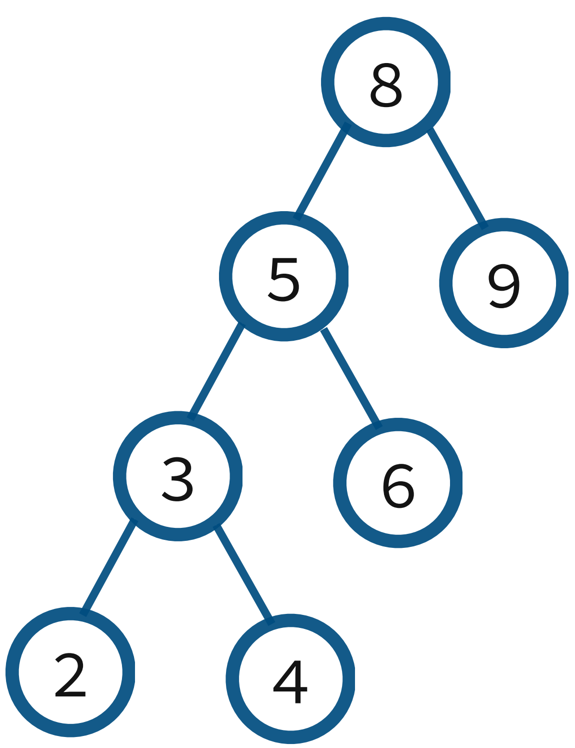
Not balanced!
Left heavy and left
child is left heavy

Self-Balancing BSTs: Tree Rotations

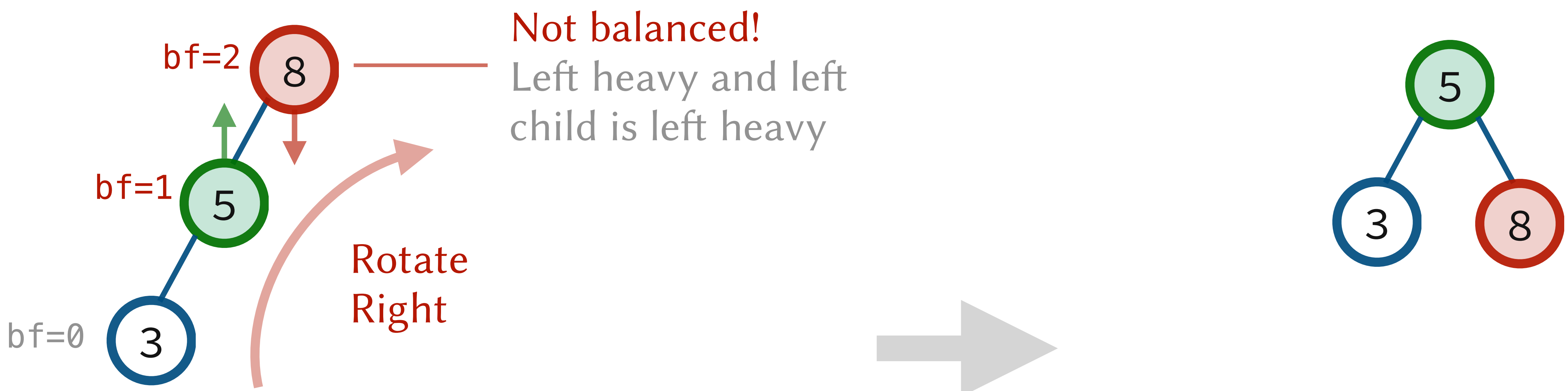


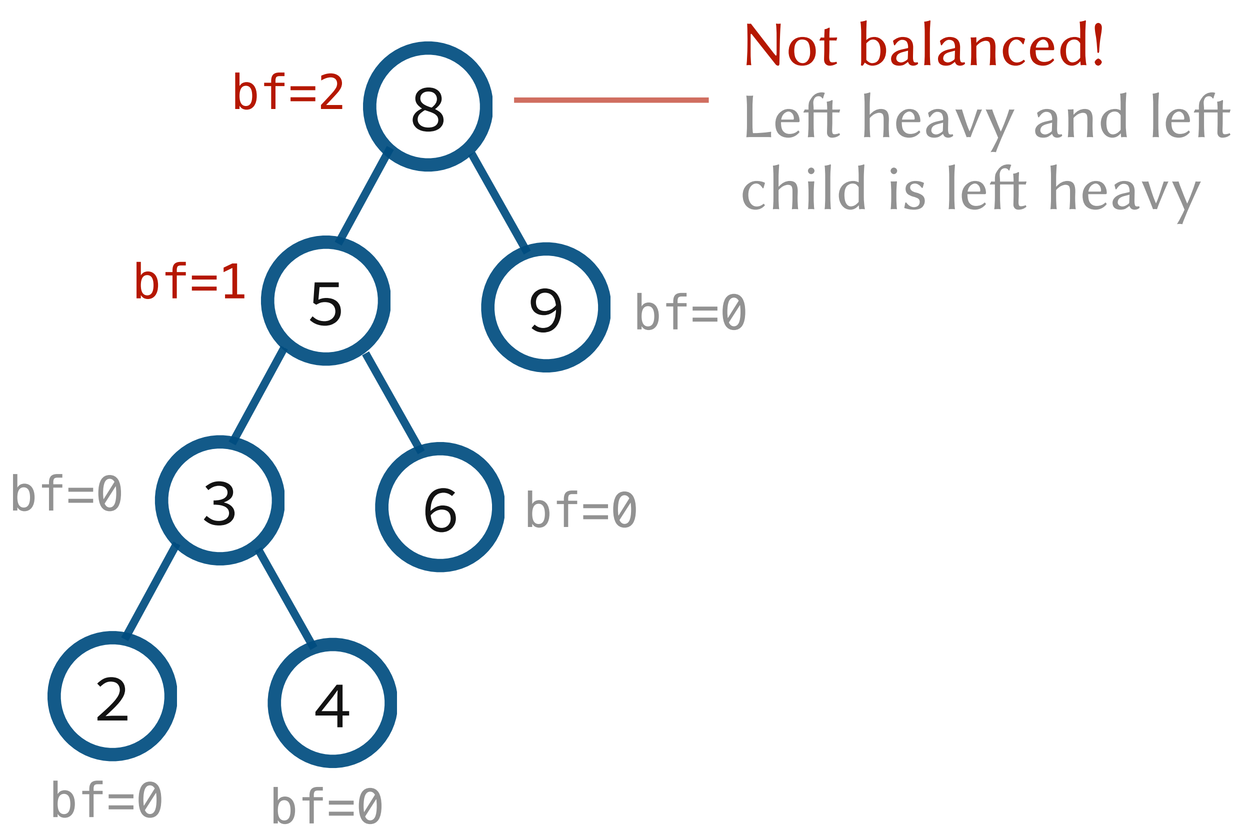
Self-Balancing BSTs: Tree Rotations



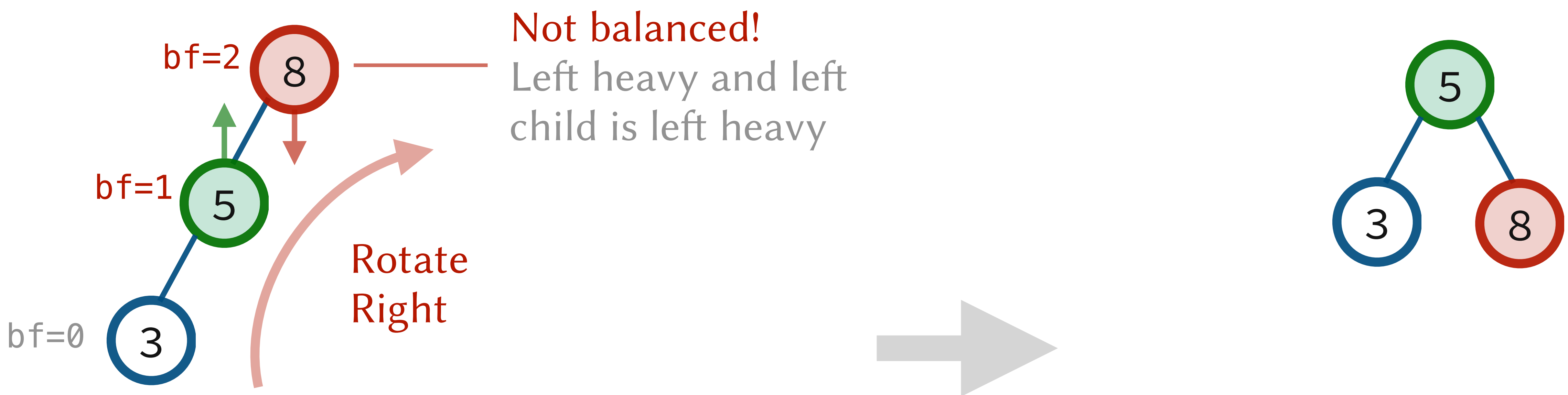


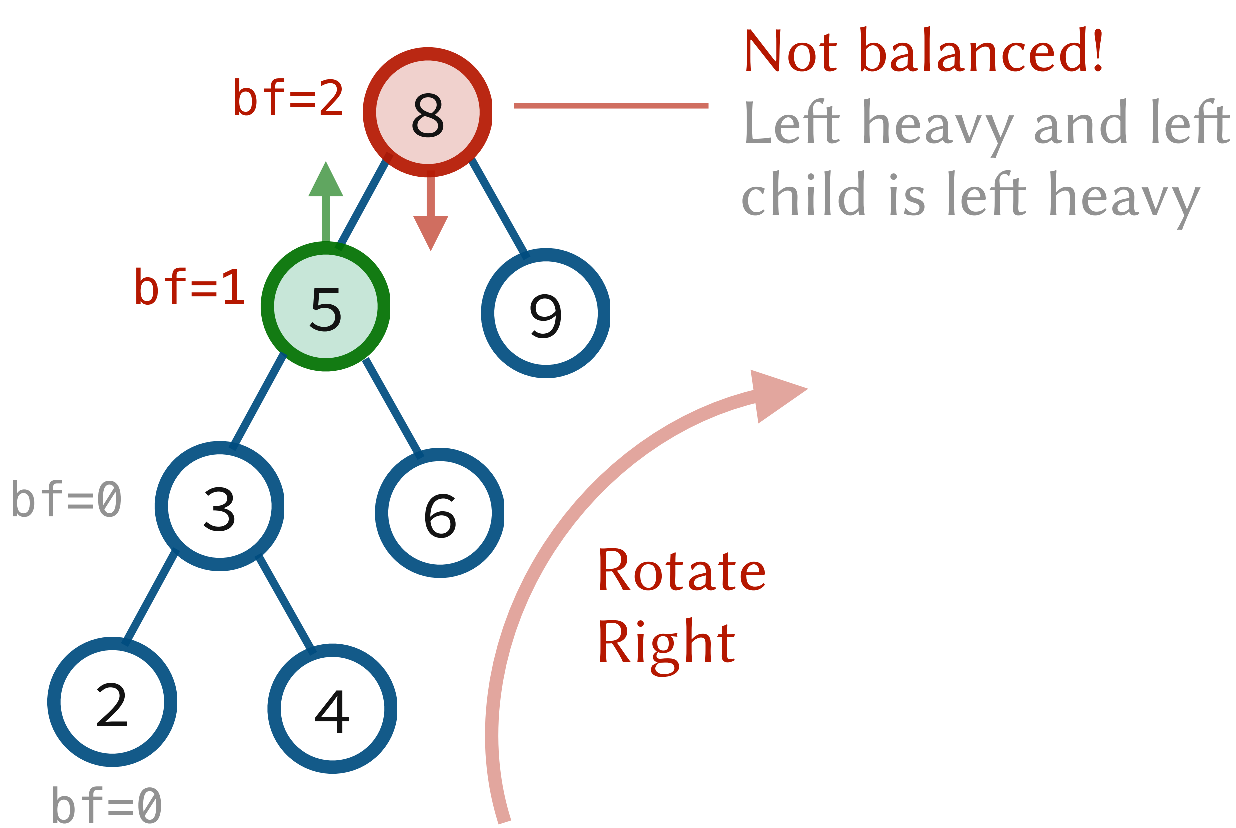
Self-Balancing BSTs: Tree Rotations



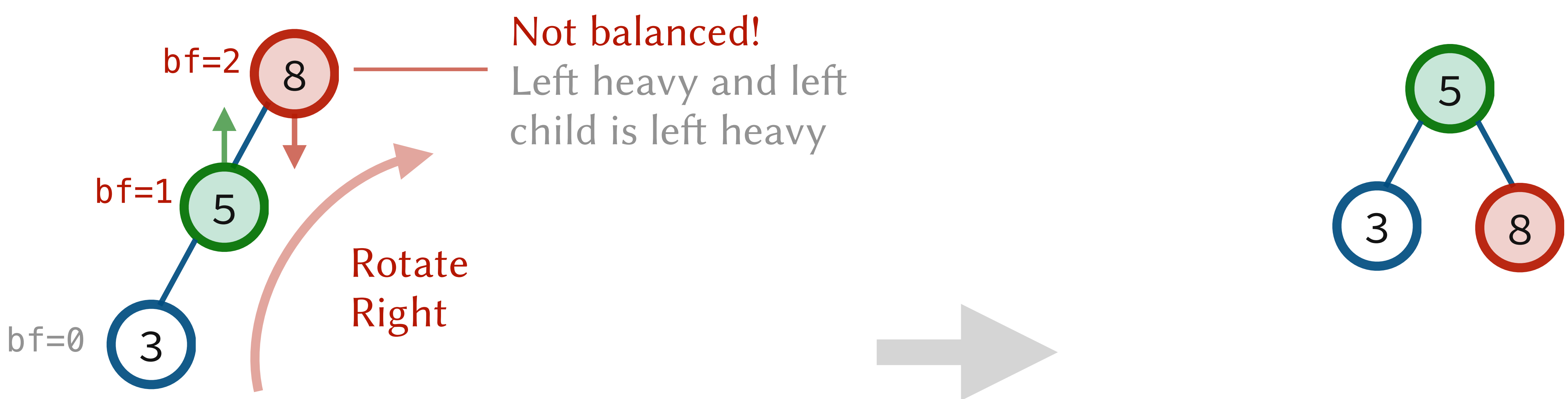


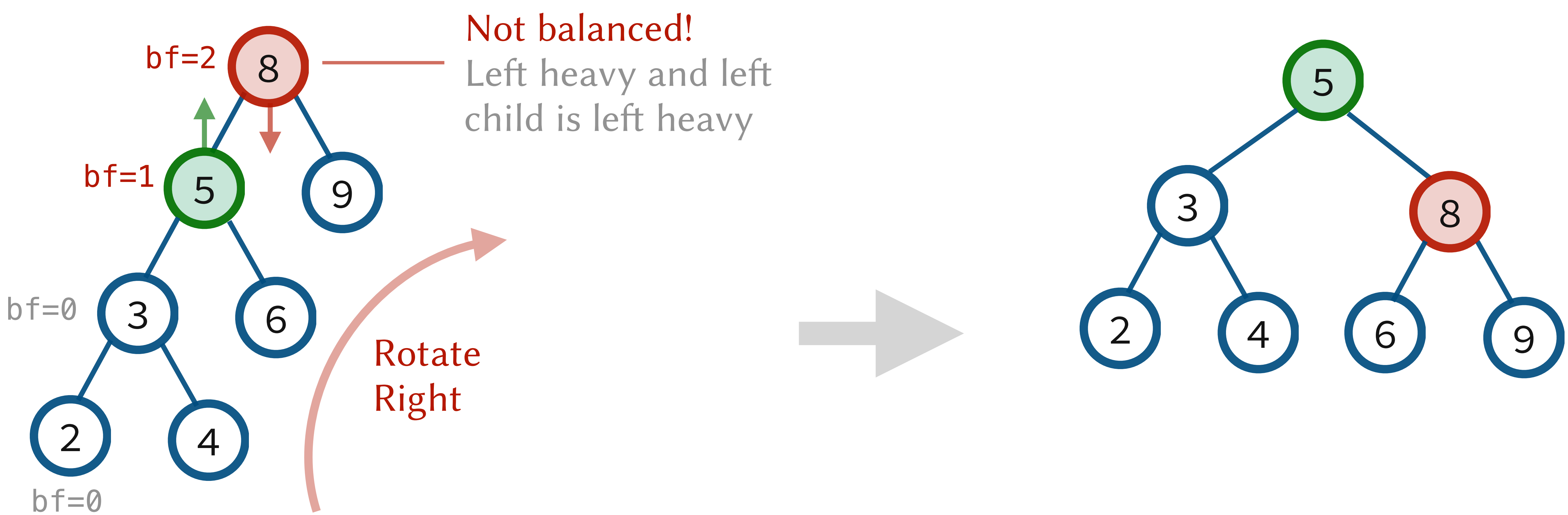
Self-Balancing BSTs: Tree Rotations



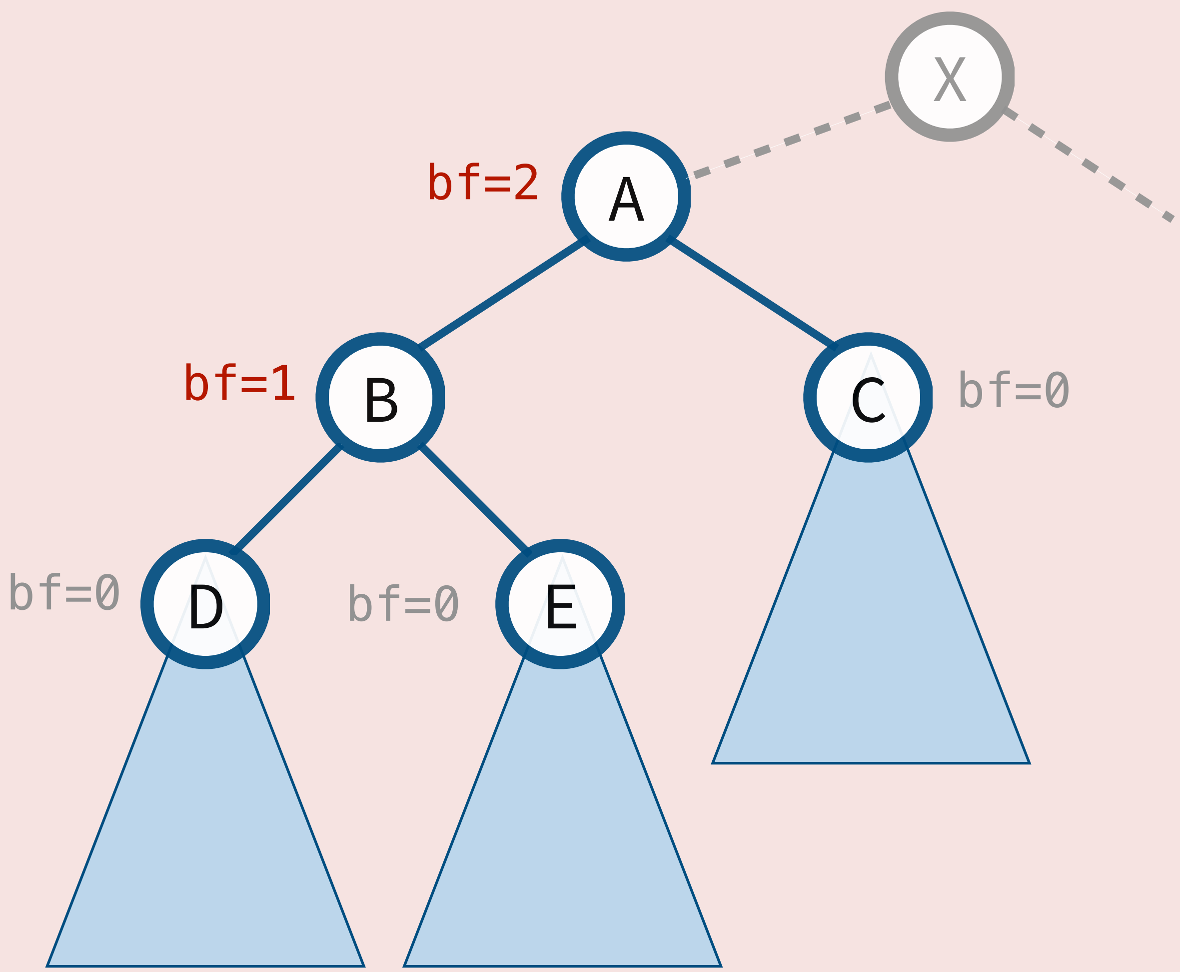


Self-Balancing BSTs: Tree Rotations





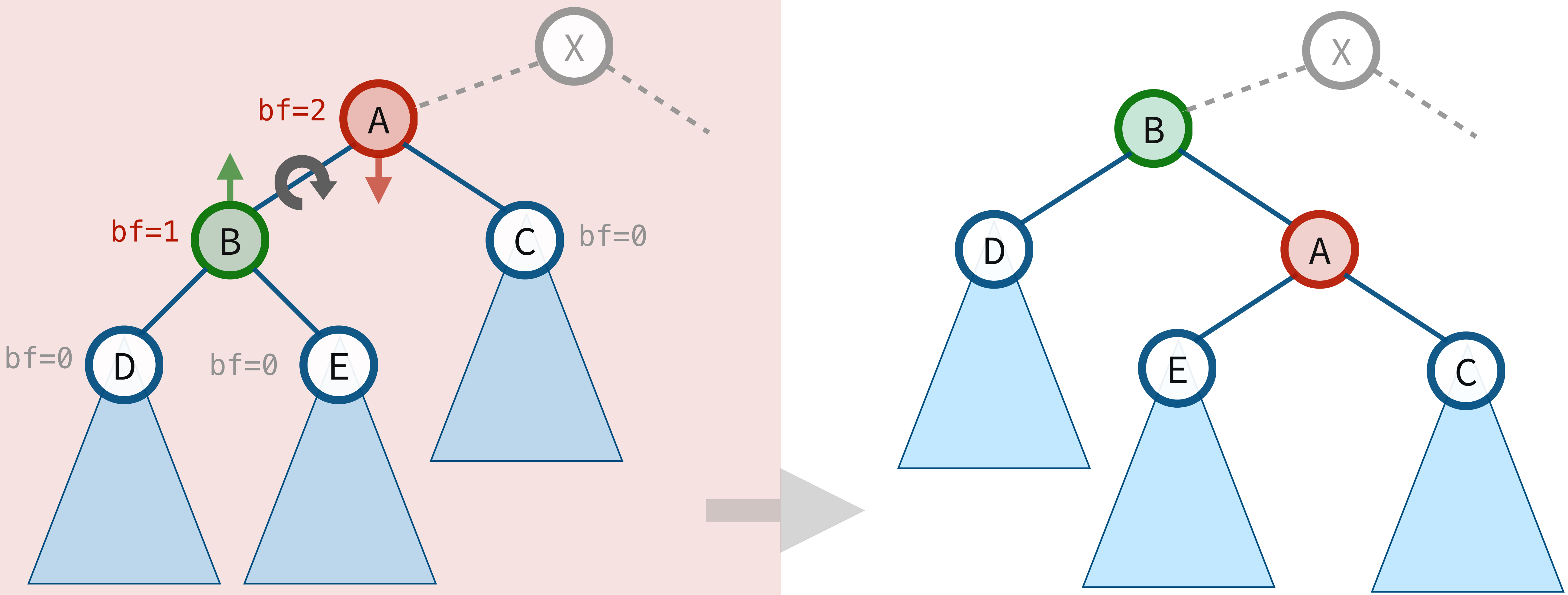
Exercise: Right Rotation



Not balanced!
Left heavy and left child is left heavy

Task: Perform a right rotation.

Exercise: Right Rotation

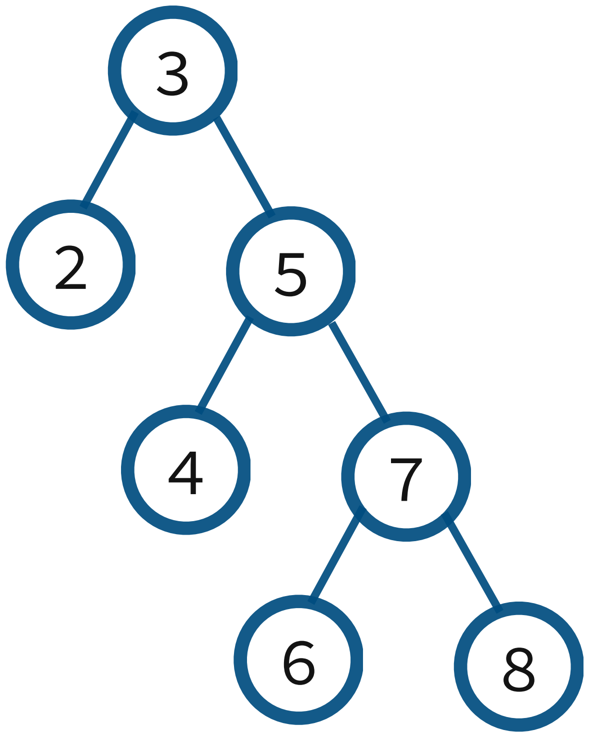


Not balanced!
Left heavy and left child is left heavy

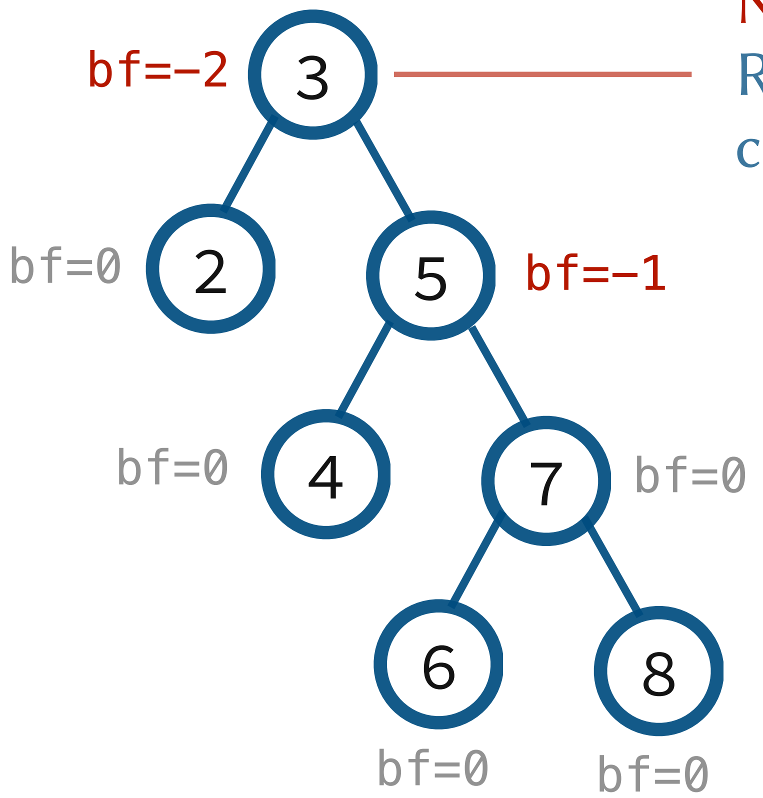
Task: Perform a right rotation.

```
X->left = B;
A->left = B->right;
B->right = A;
```

Self-Balancing BSTs: Left Rotations

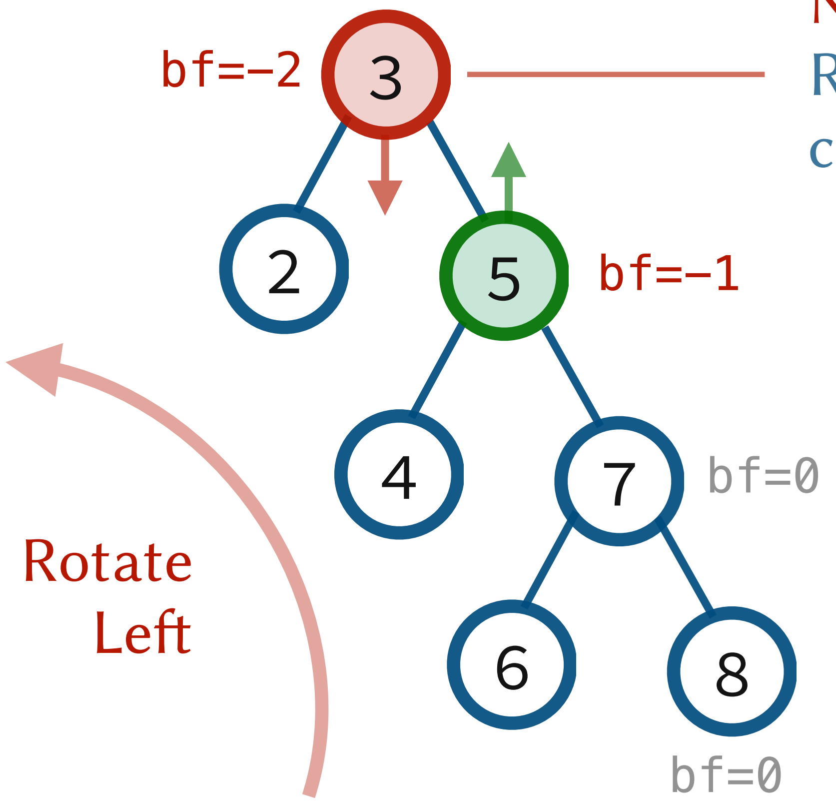


Self-Balancing BSTs: Left Rotations



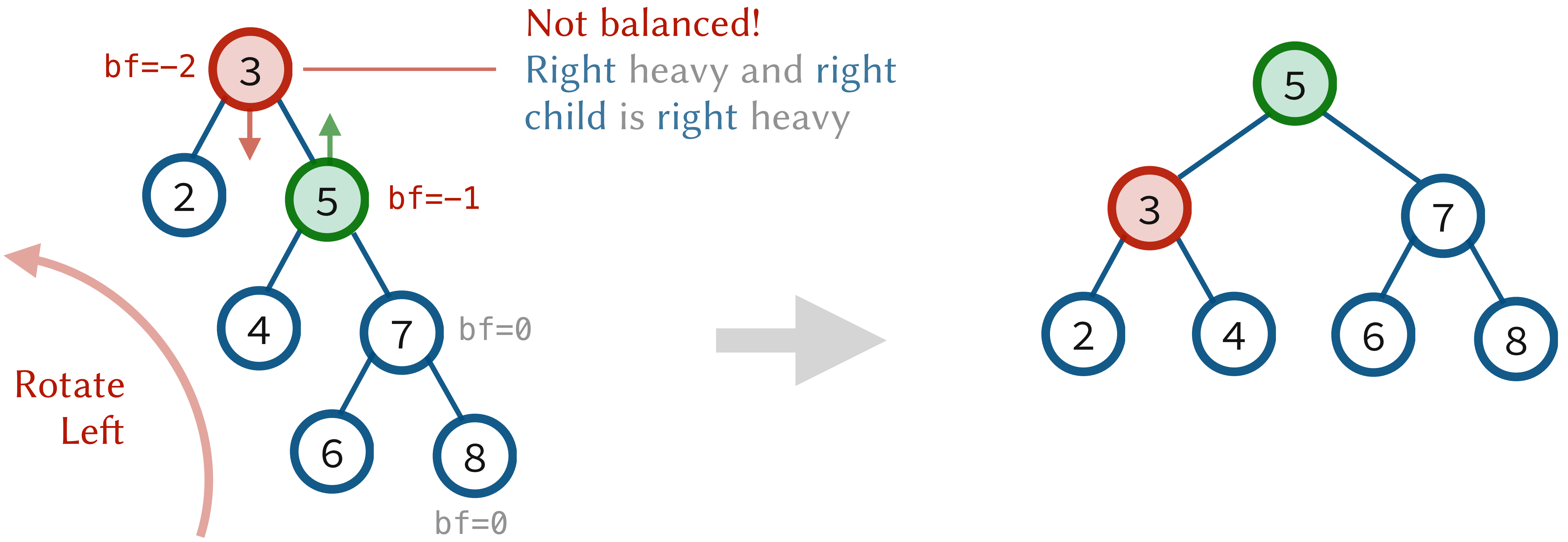
Not balanced!
Right heavy and right child is right heavy

Self-Balancing BSTs: Left Rotations

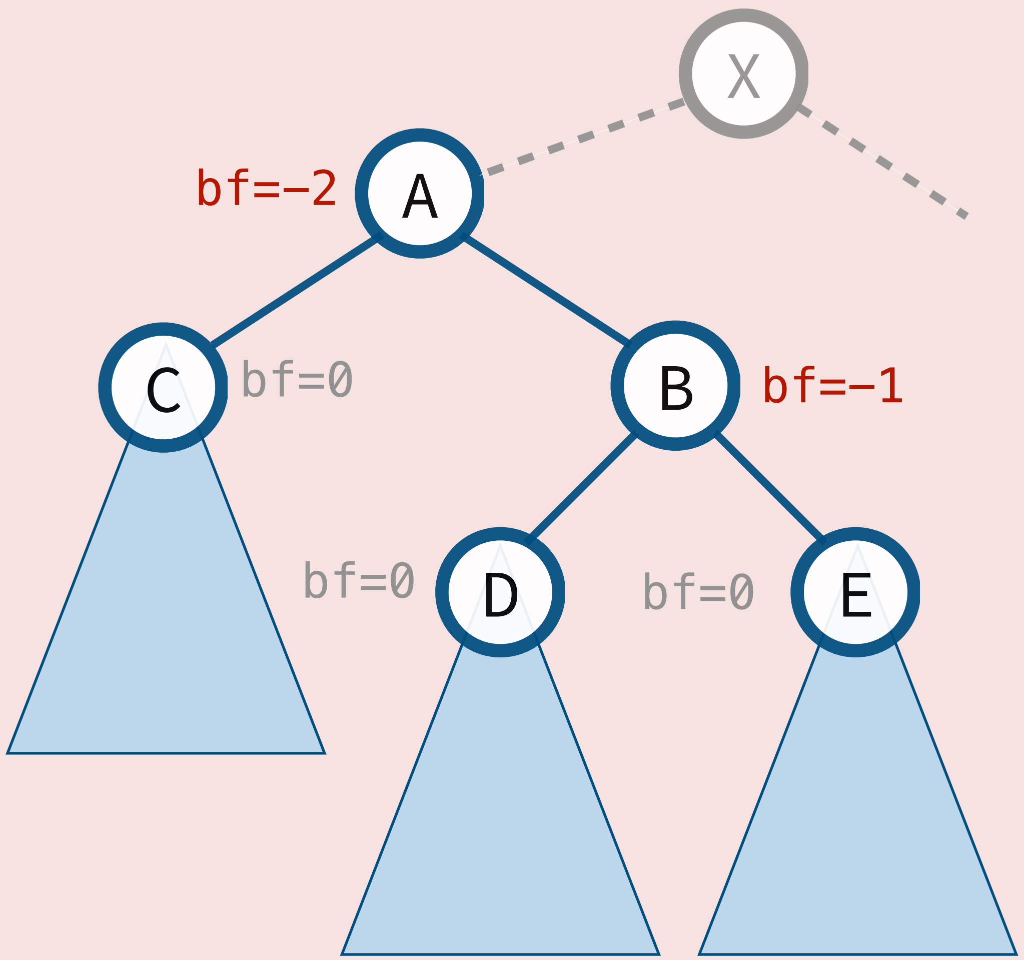


Not balanced!
Right heavy and right child is right heavy

Self-Balancing BSTs: Left Rotations



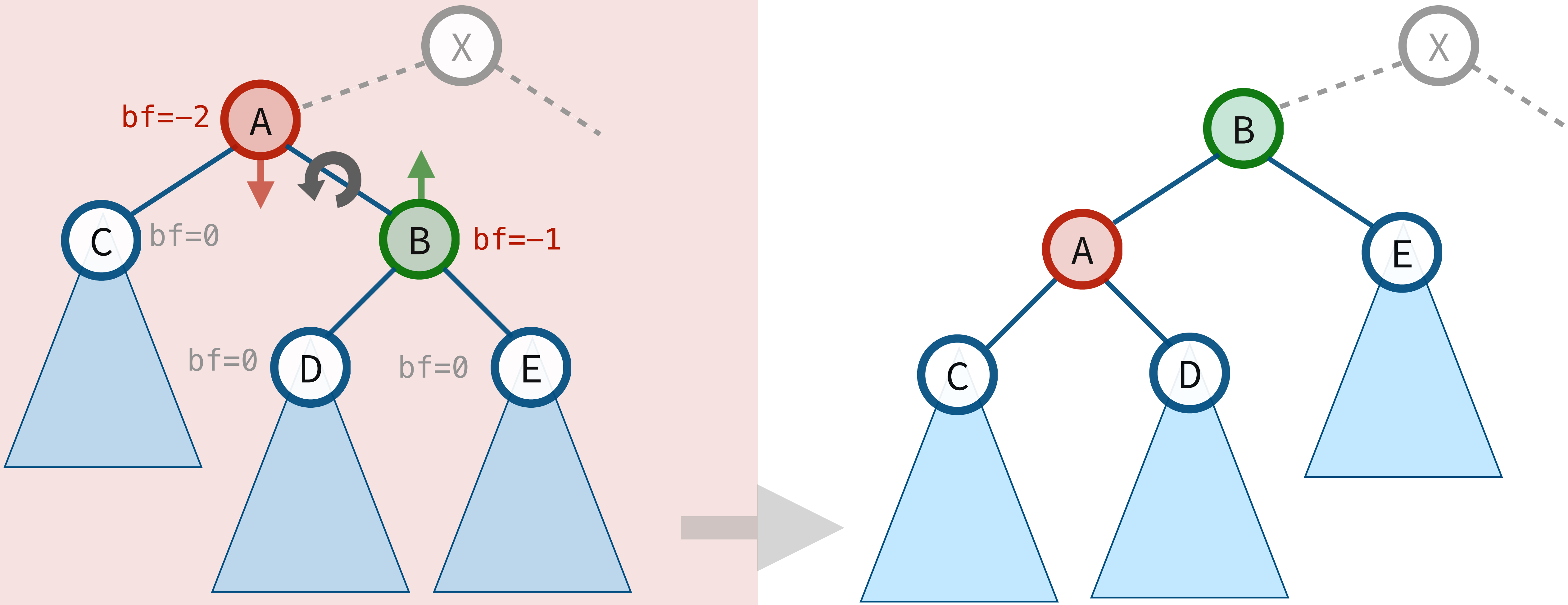
Exercise: Left Rotation



Not balanced!
Right heavy and right
child is right heavy

Task: Perform a left rotation.

Exercise: Left Rotation

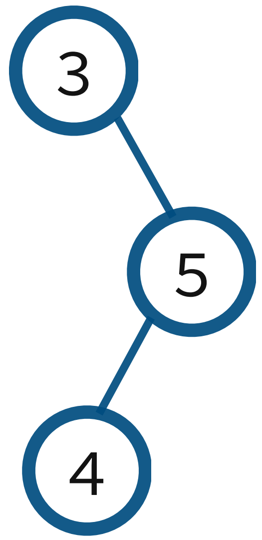


Not balanced!
Right heavy and right child is right heavy

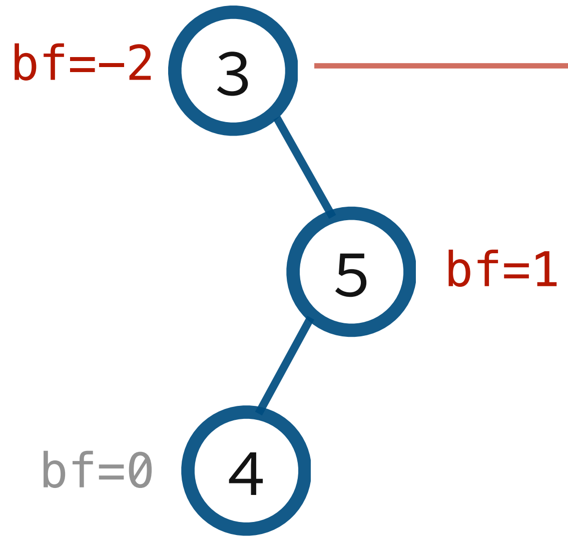
Task: Perform a left rotation.

```
X->left = B;
A->right = B->left;
B->left = A;
```

Self-Balancing BSTs: Double Rotations

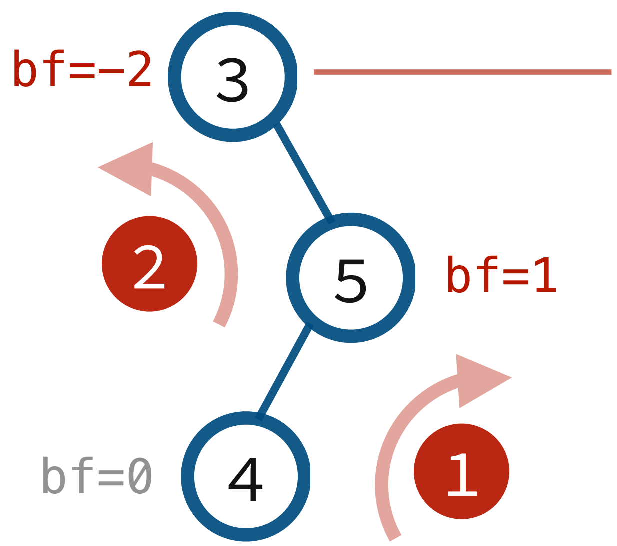


Self-Balancing BSTs: Double Rotations



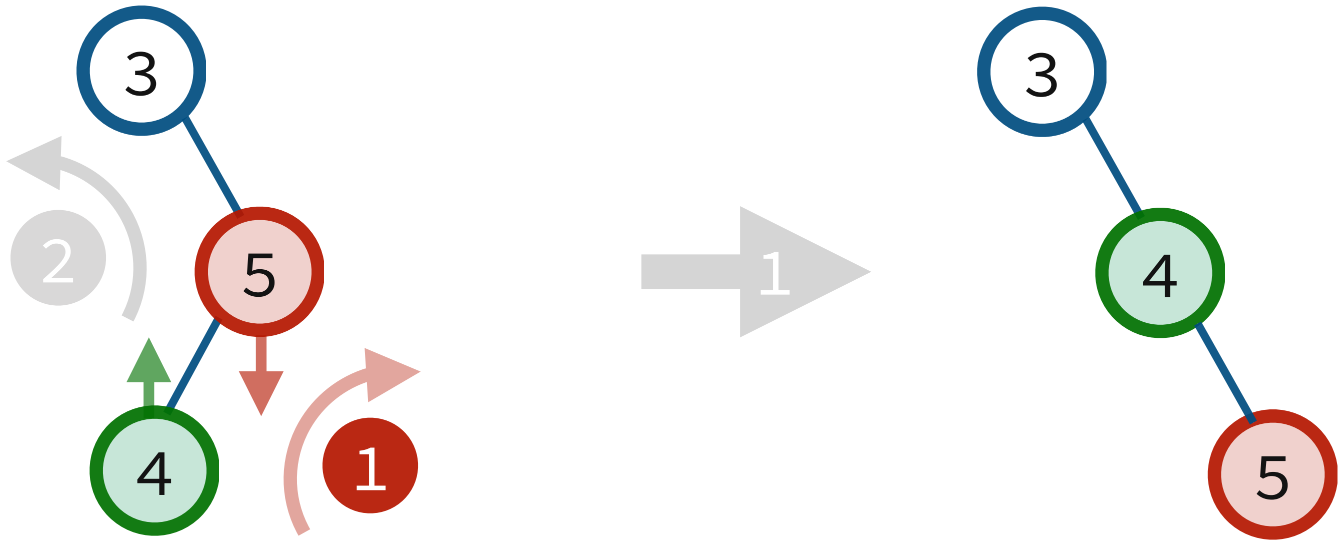
Not balanced!
Right heavy and right child is left heavy

Self-Balancing BSTs: Double Rotations

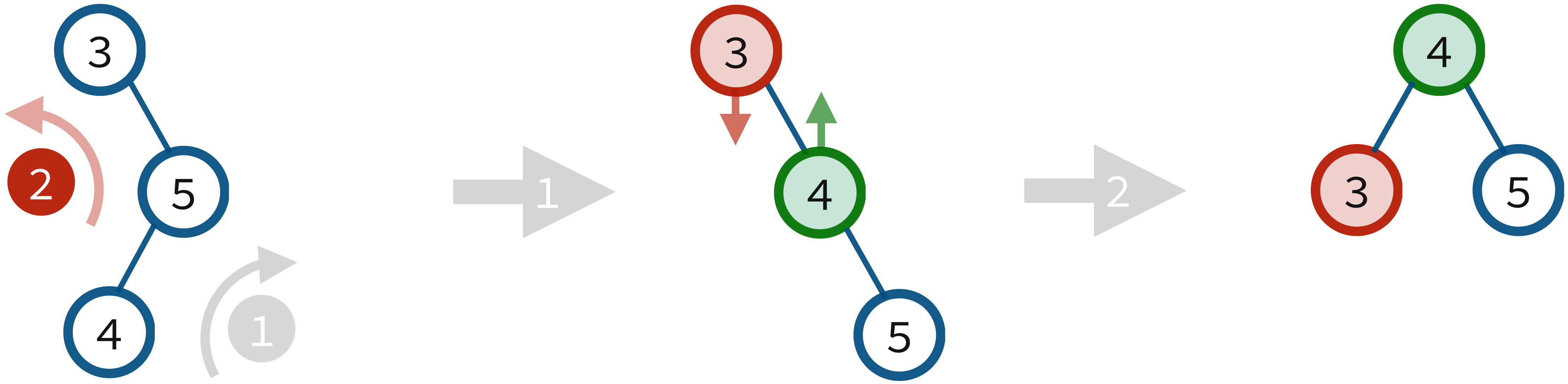


Not balanced!
Right heavy and right child is left heavy

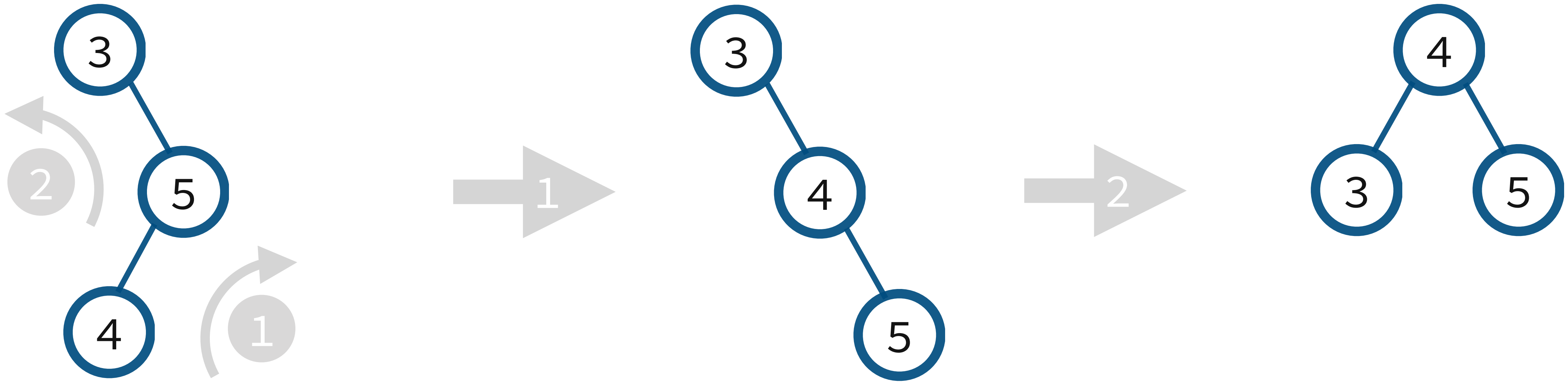
Self-Balancing BSTs: Double Rotations

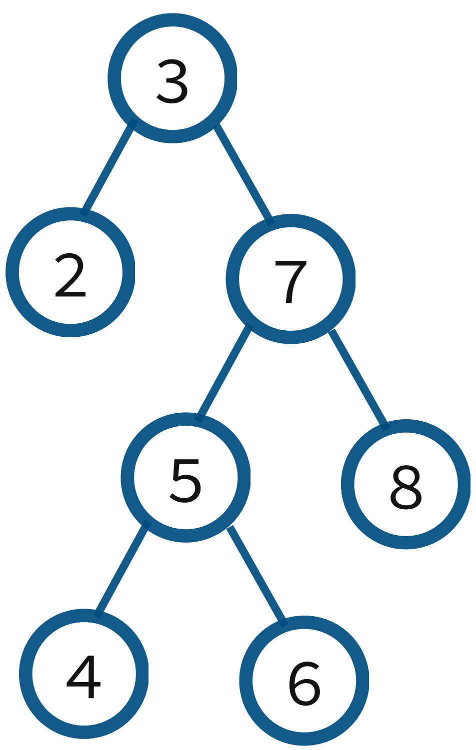


Self-Balancing BSTs: Double Rotations

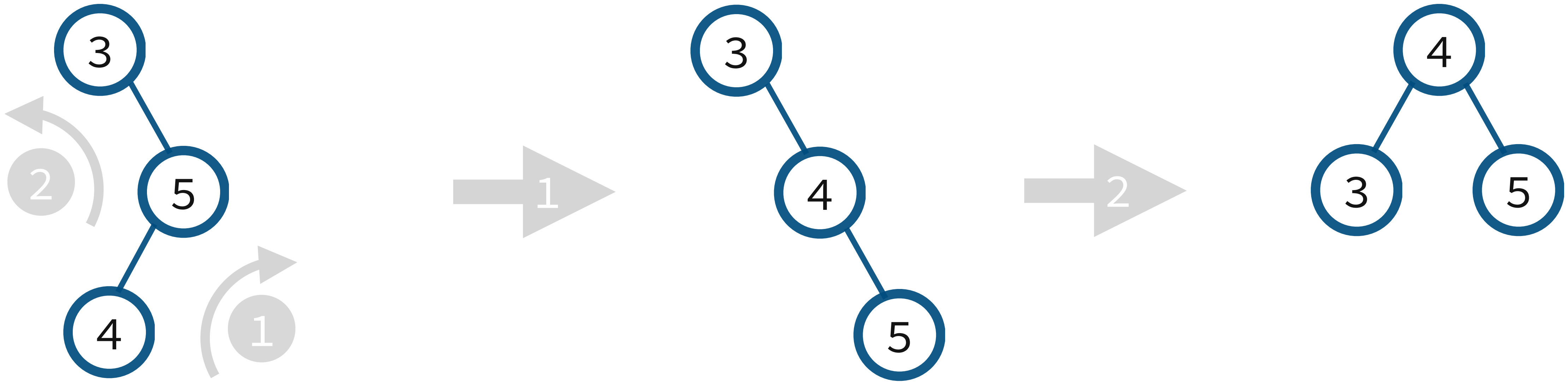


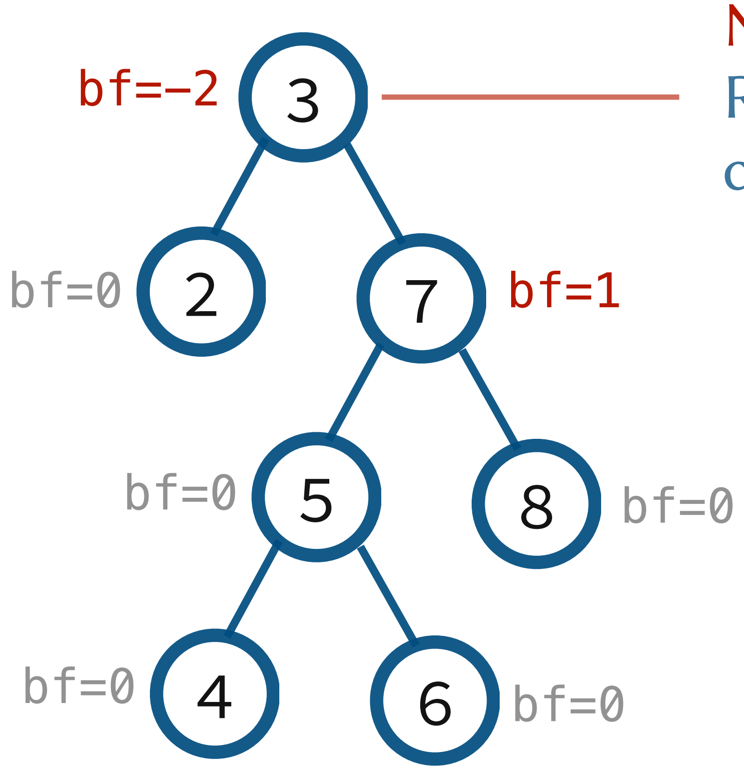
Self-Balancing BSTs: Double Rotations





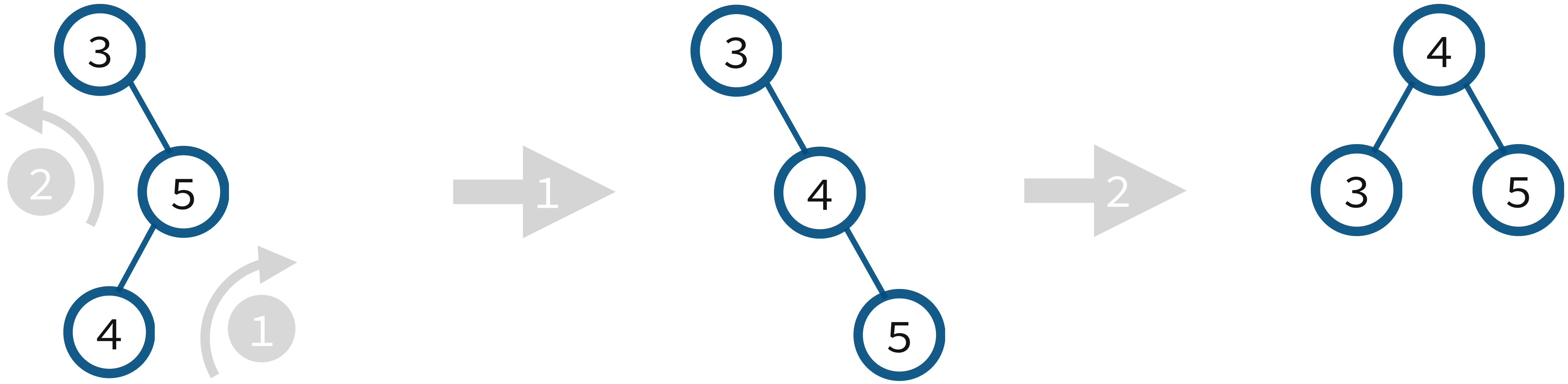
Self-Balancing BSTs: Double Rotations

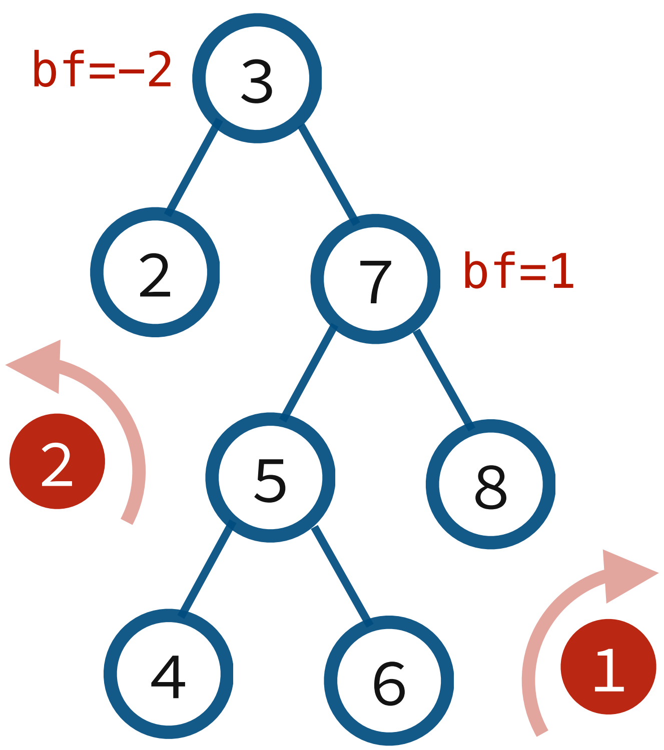




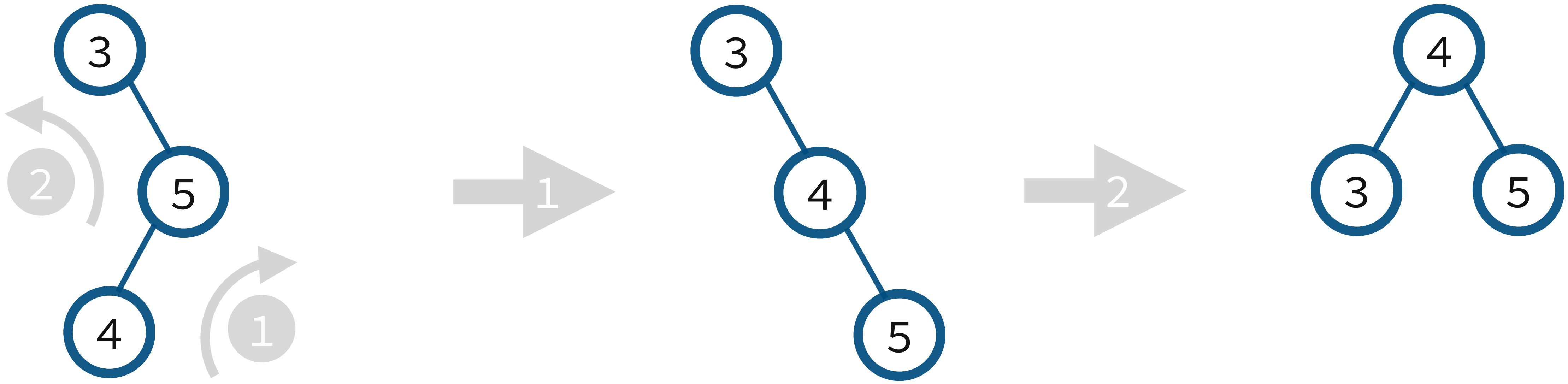
Not balanced!
Right heavy and right child is left heavy

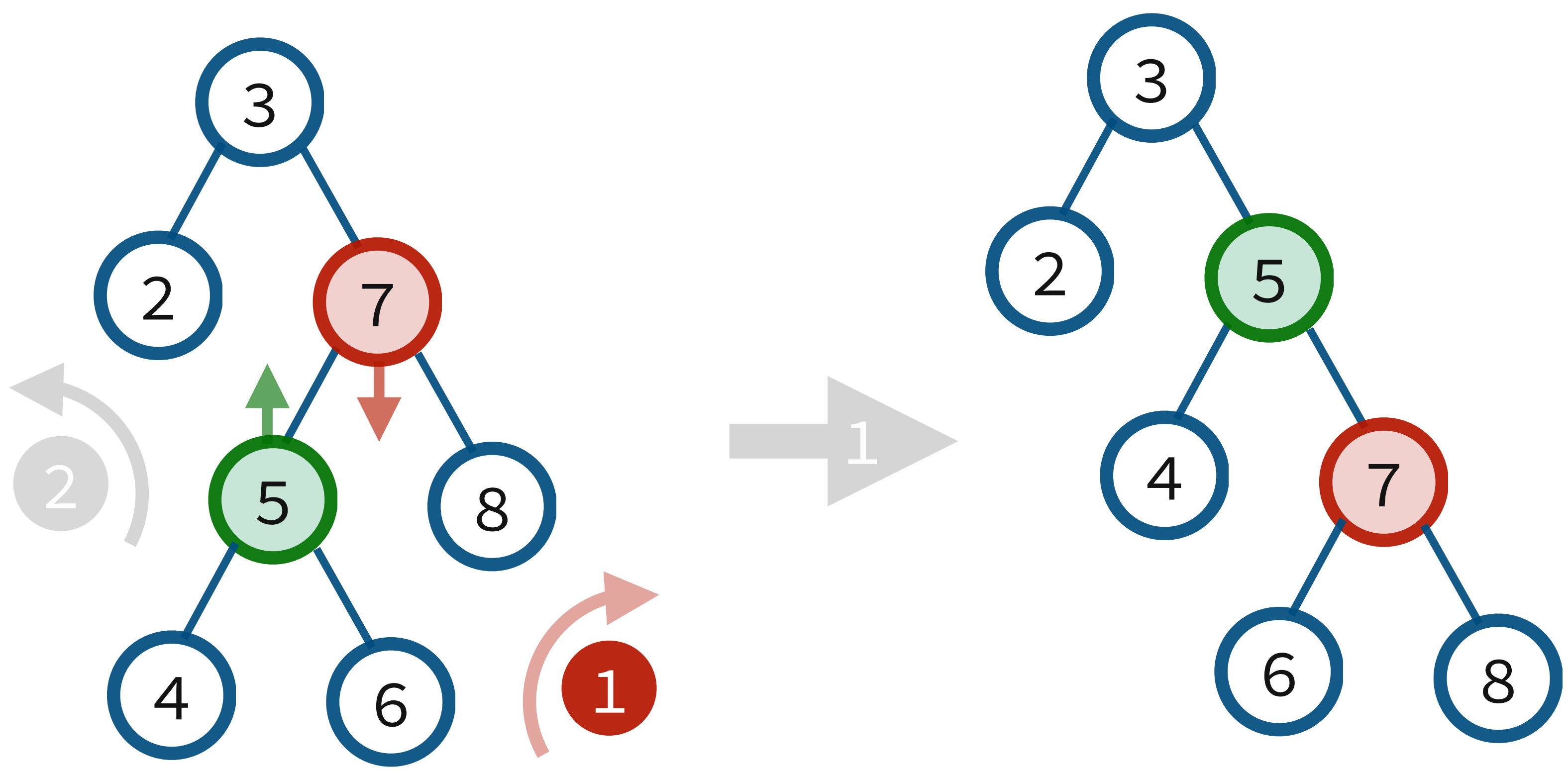
Self-Balancing BSTs: Double Rotations



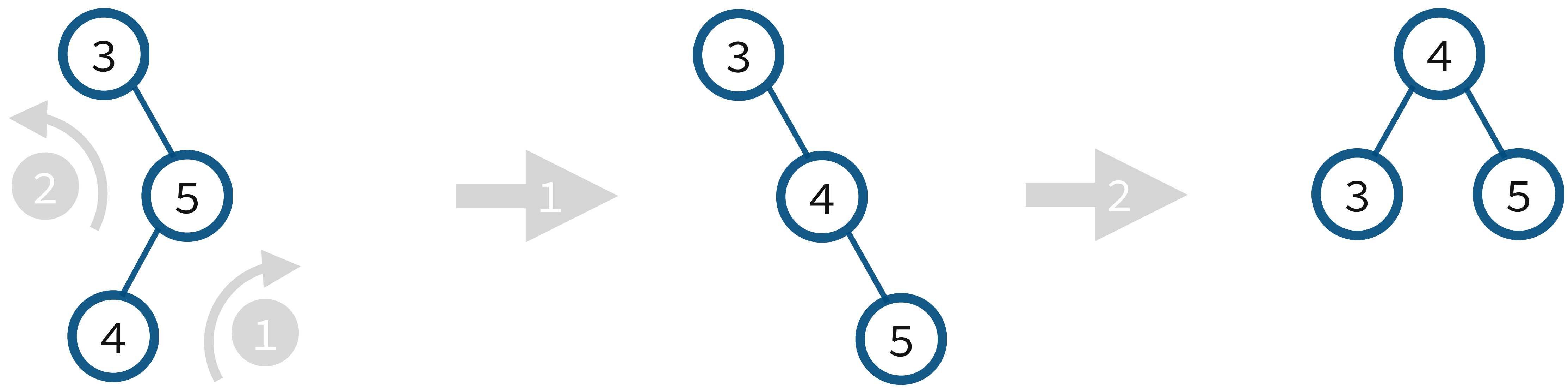


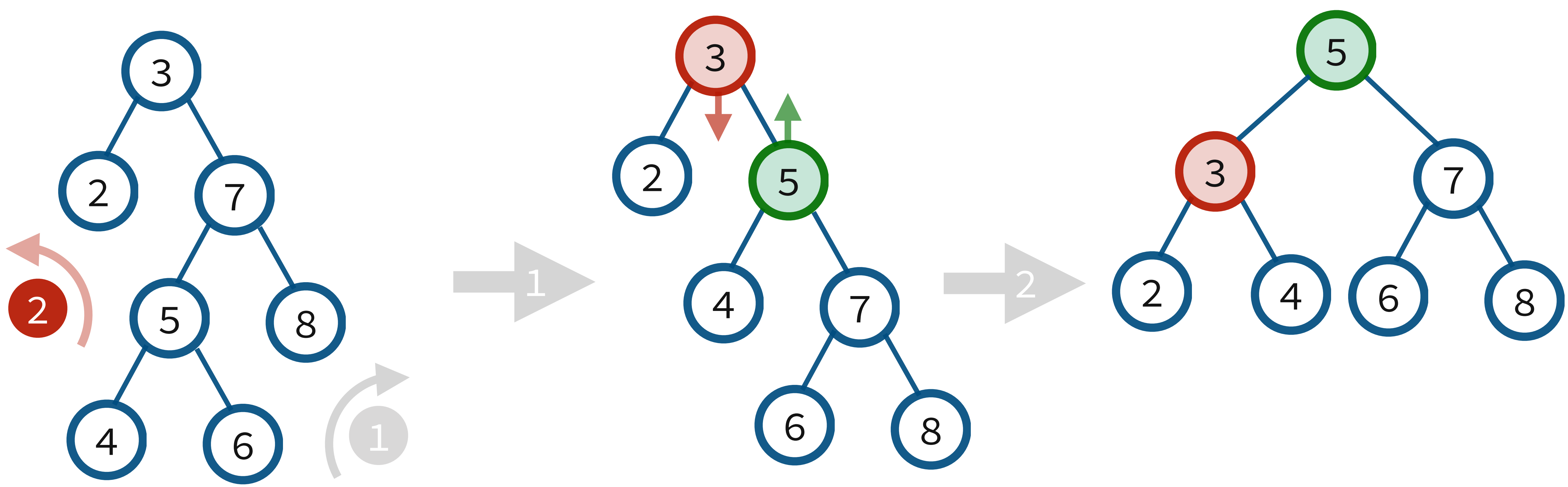
Self-Balancing BSTs: Double Rotations



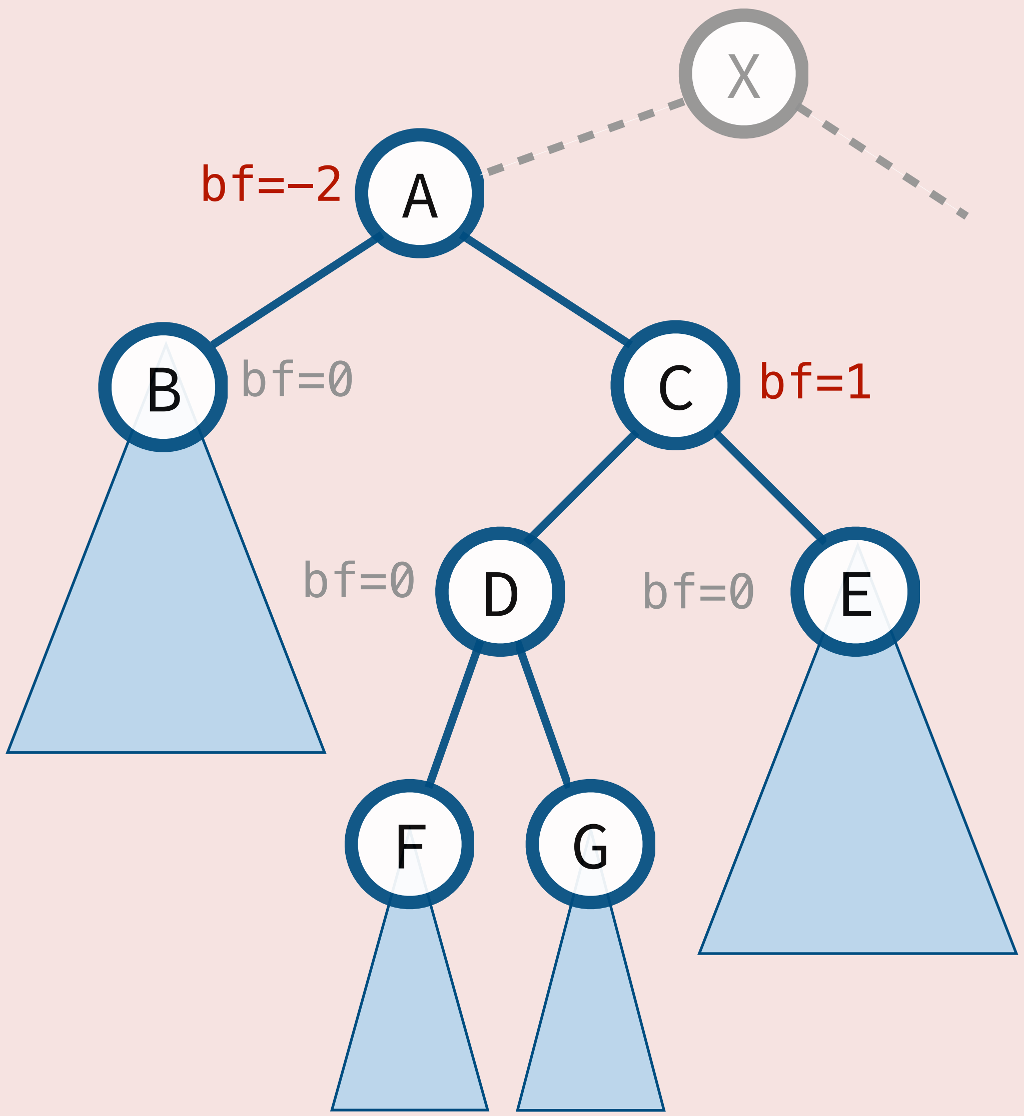


Self-Balancing BSTs: Double Rotations





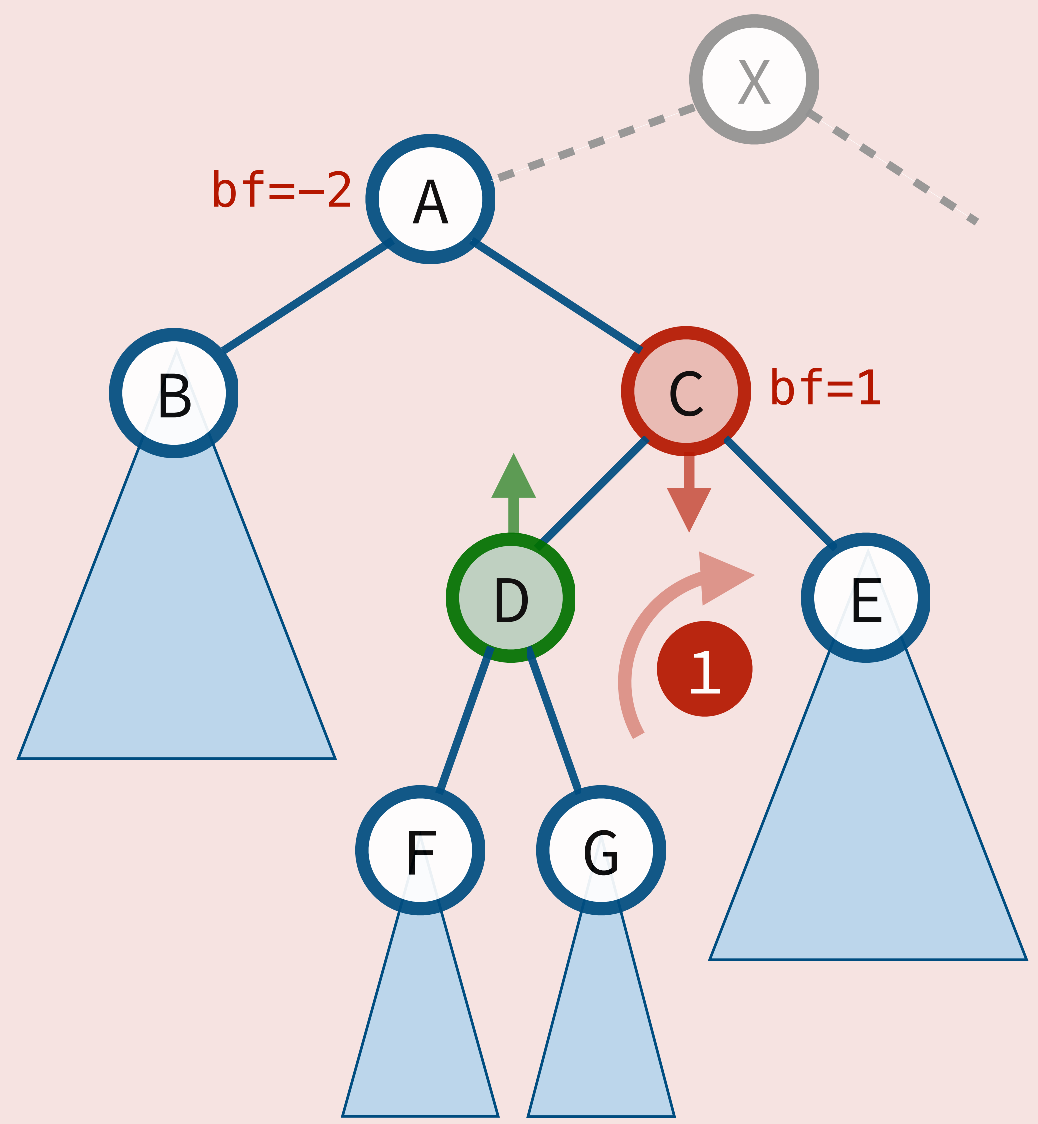
Exercise: Double Rotation

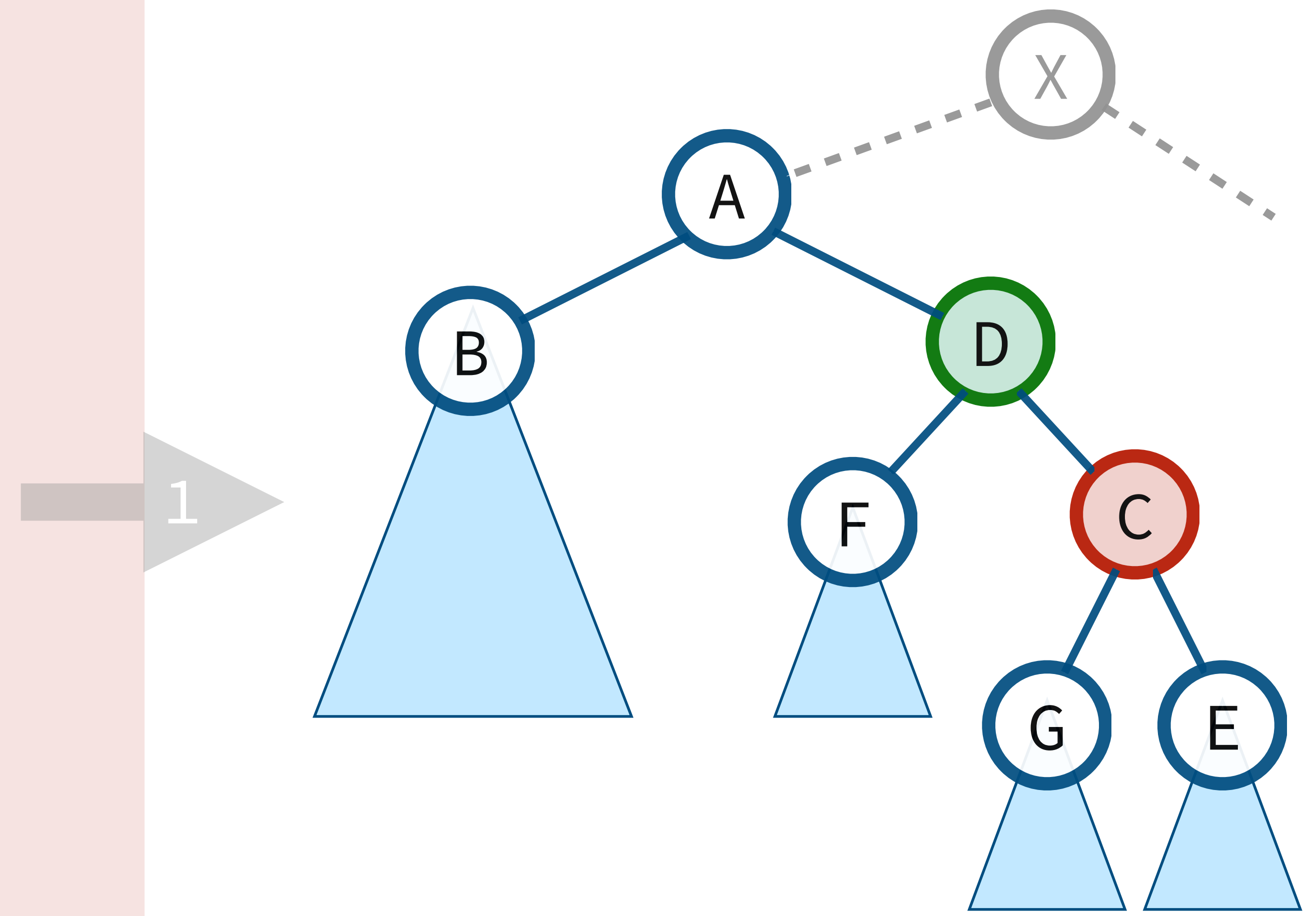


Not balanced!
Right heavy and right child is left heavy

Task: Perform a double rotation.

Exercise: Double Rotation

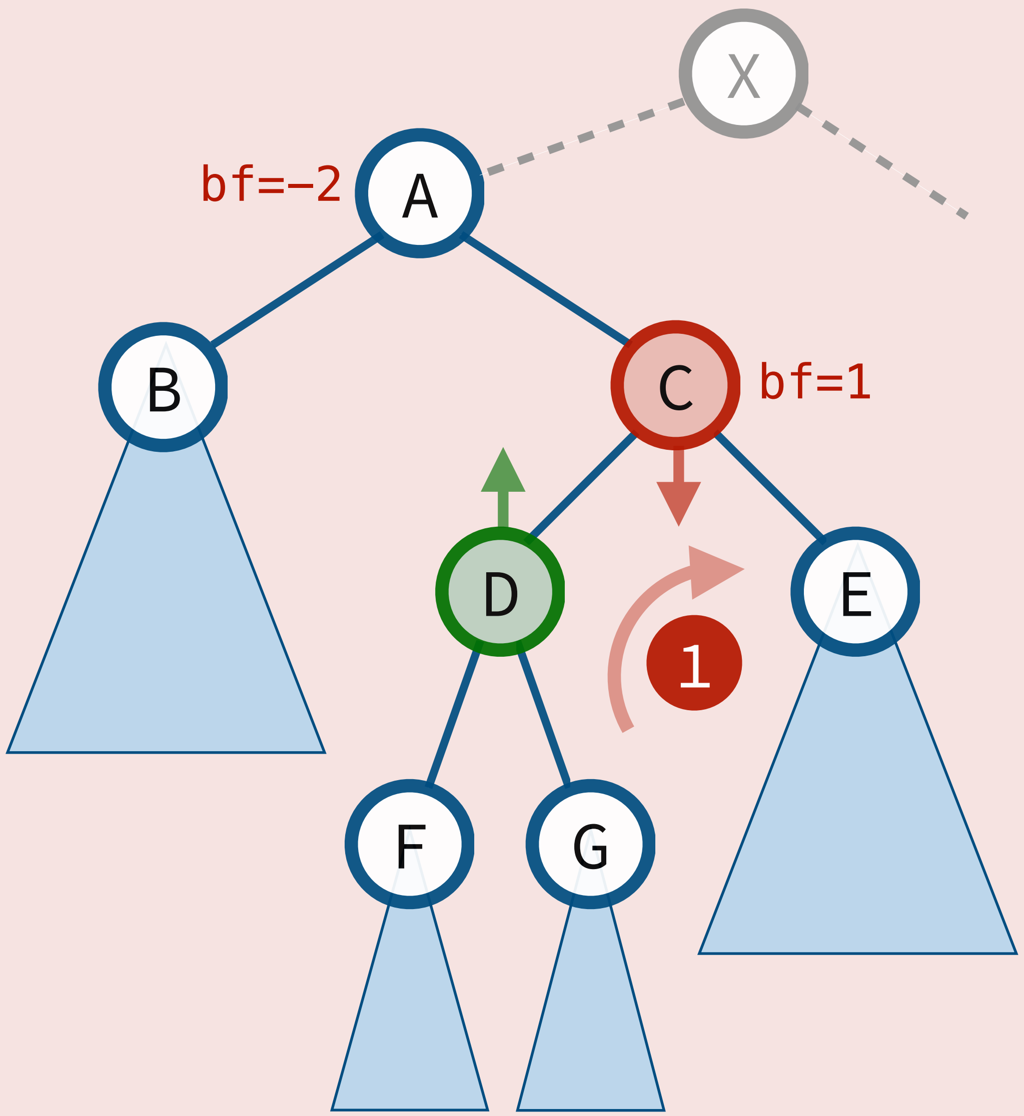


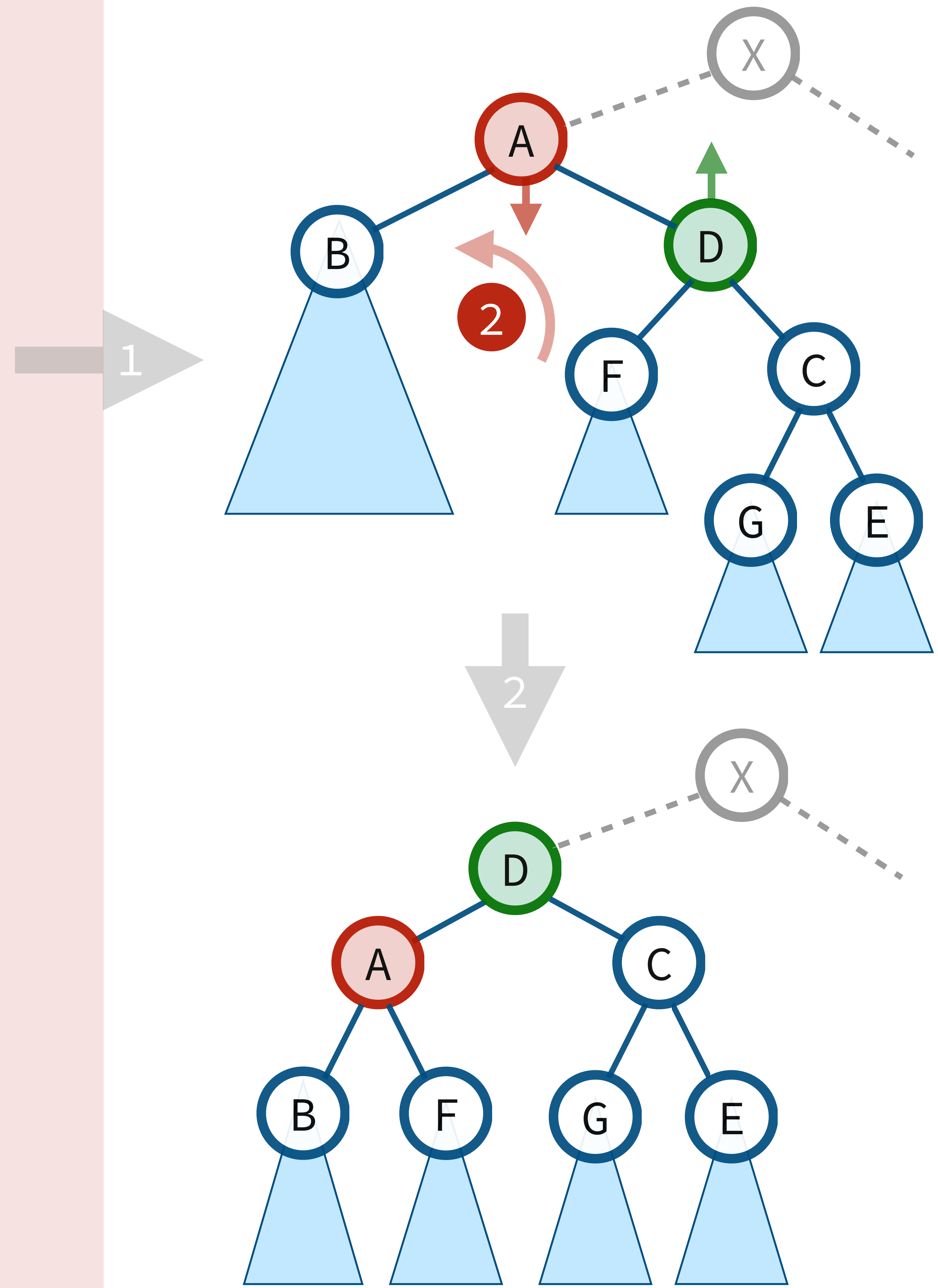


Not balanced!
Right heavy and right child is left heavy

Task: Perform a double rotation.

Exercise: Double Rotation

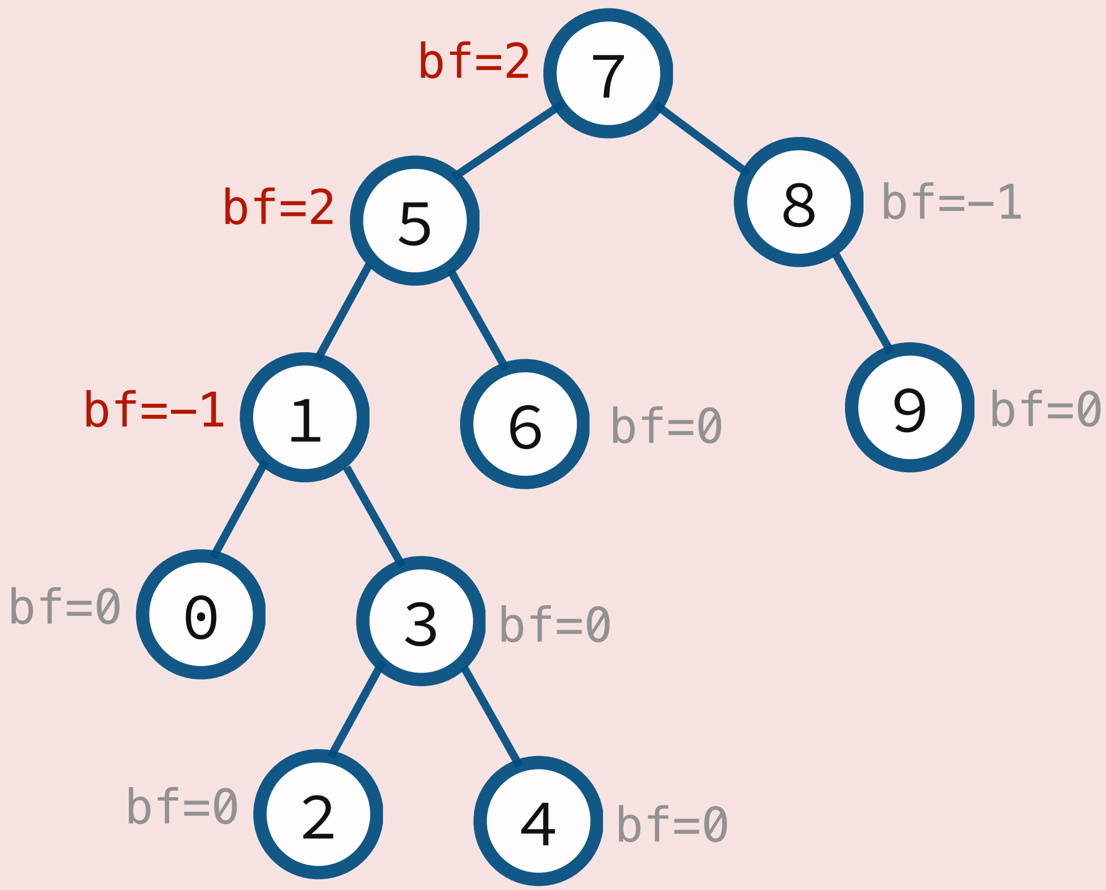




Not balanced!
Right heavy and right child is left heavy

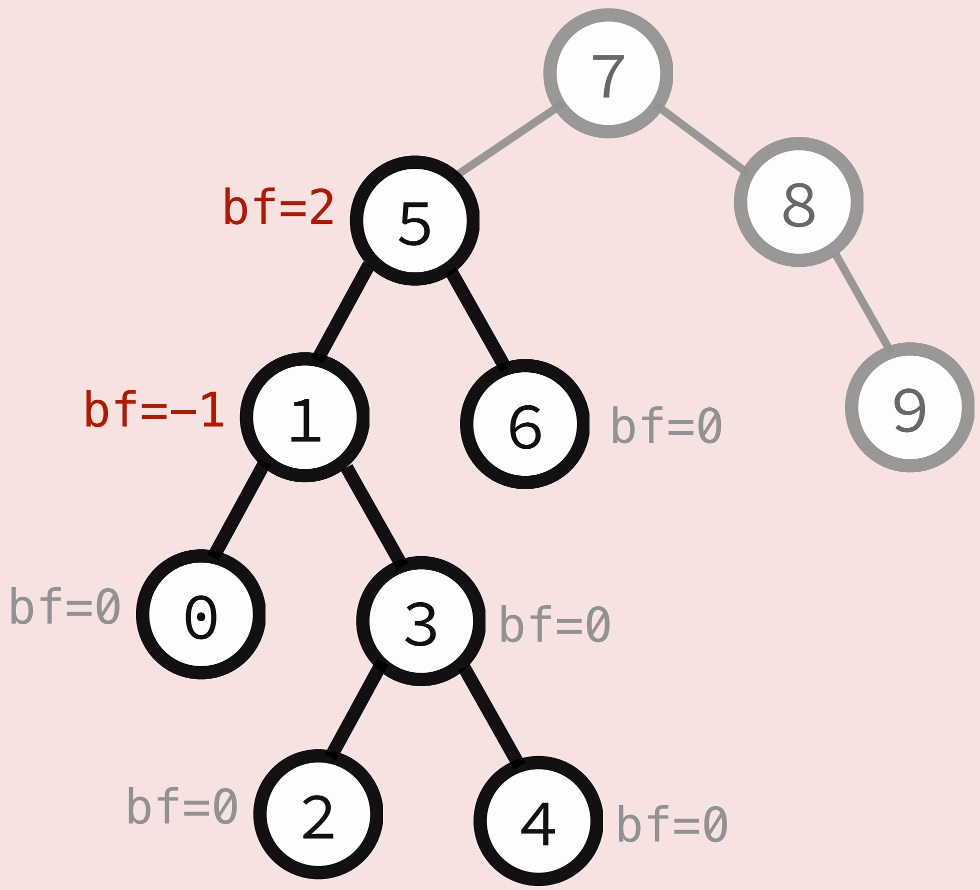
Task: Perform a double rotation.

Exercise: Double Rotation



Task: Balance the tree

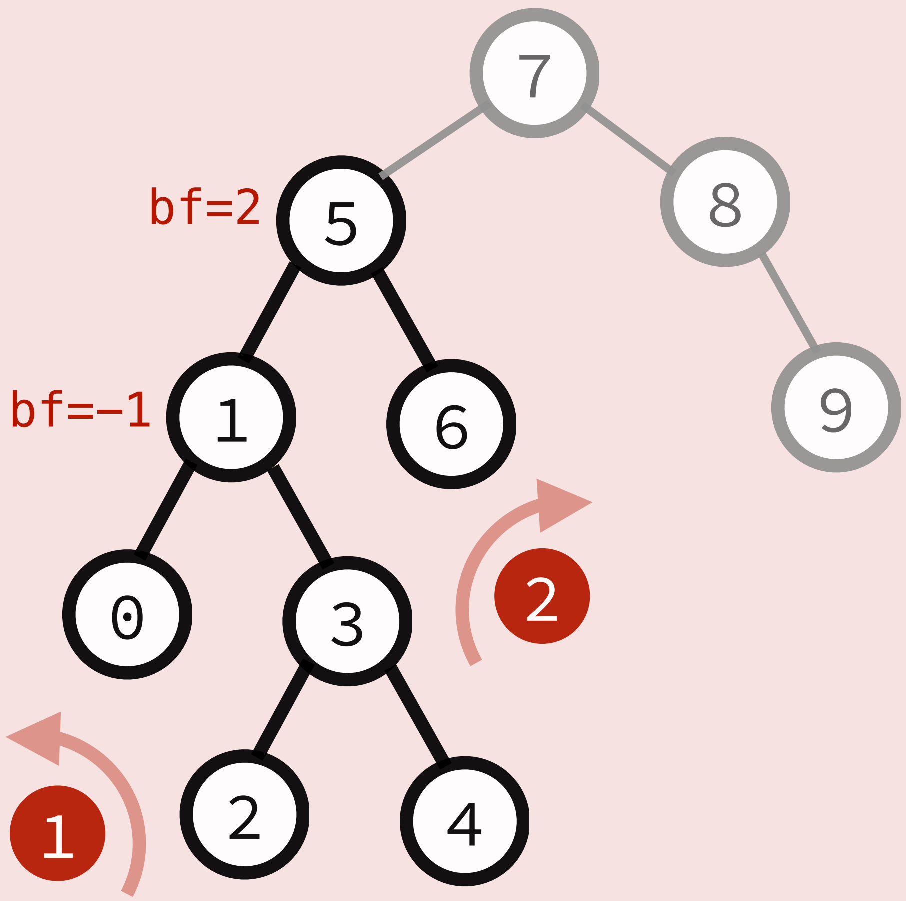
Exercise: Double Rotation



Task: Balance the tree

Start with *lowest* misbalanced subtree

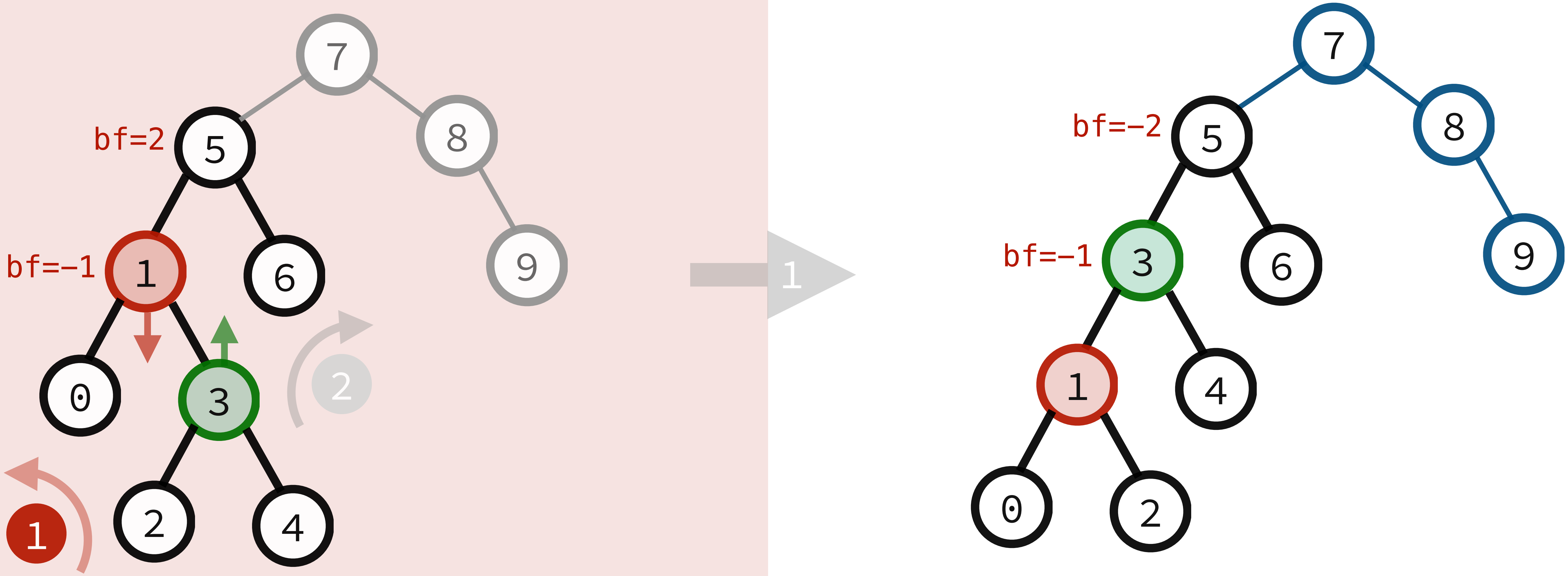
Exercise: Double Rotation



Task: Balance the tree

Start with *lowest*
misbalanced subtree

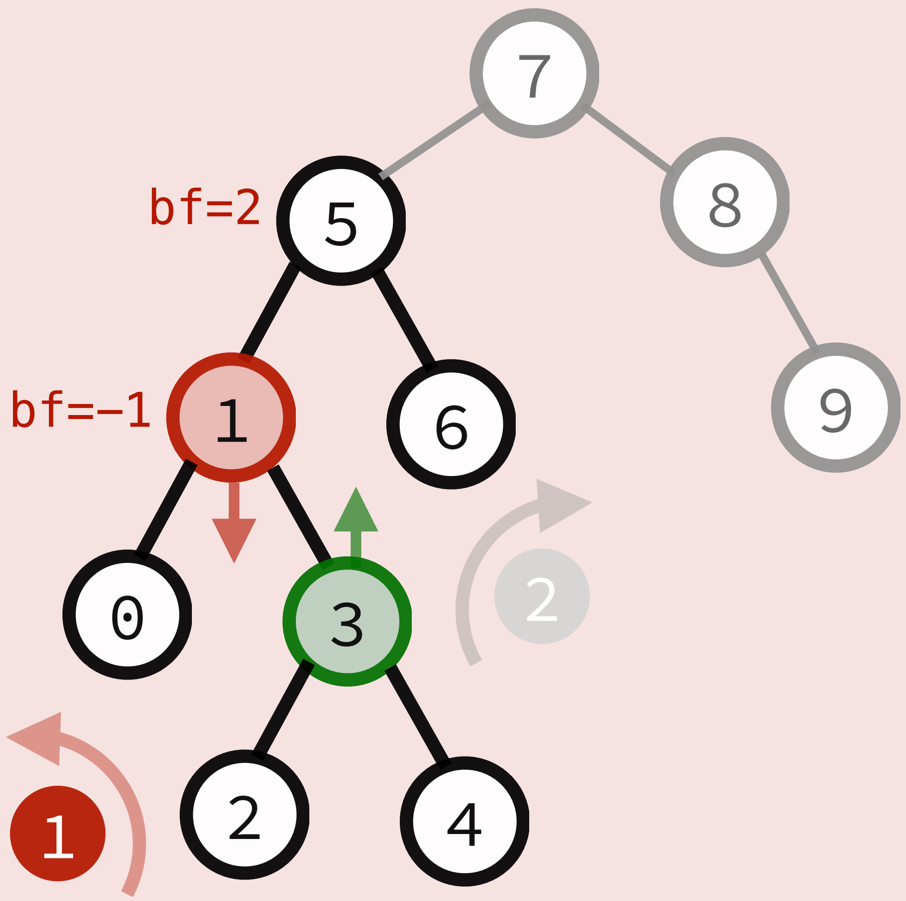
Exercise: Double Rotation

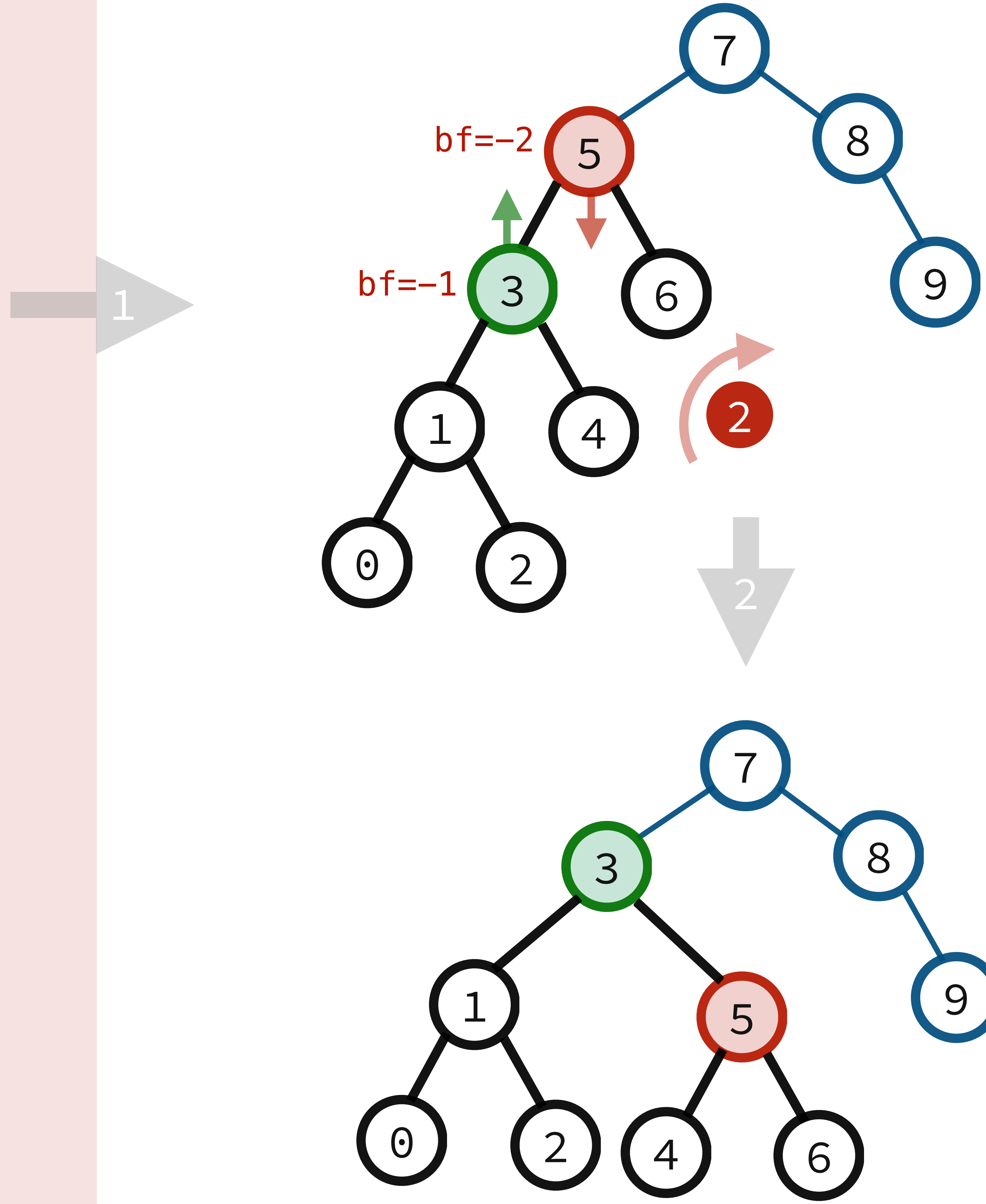


Task: Balance the tree

Start with *lowest* misbalanced subtree

Exercise: Double Rotation





Task: Balance the tree

Start with *lowest* misbalanced subtree

AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example. **3** 5 6 7 8 9 4 1 2

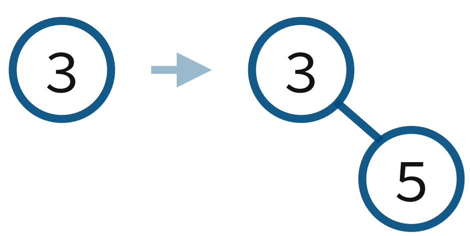
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	6	7	8	9	4	1	2



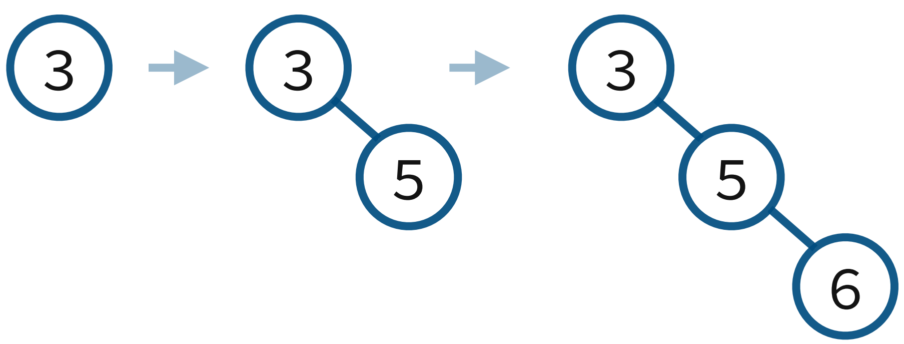
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	7	8	9	4	1	2



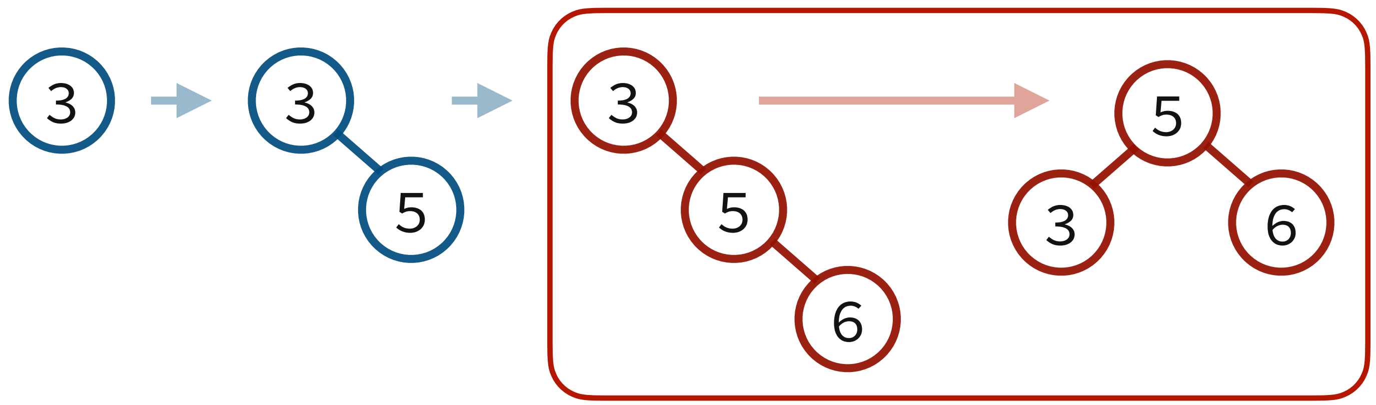
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	7	8	9	4	1	2



left rotation

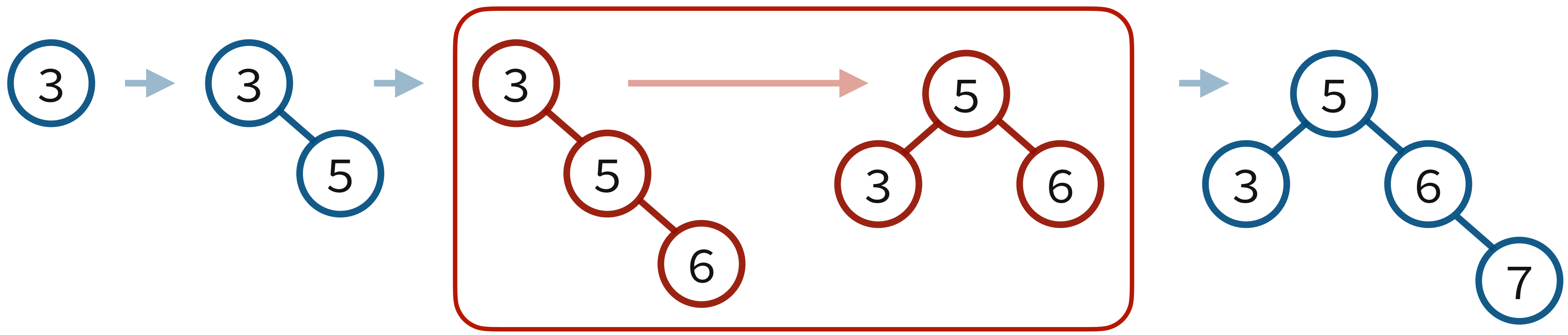
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	**7**	8	9	4	1	2



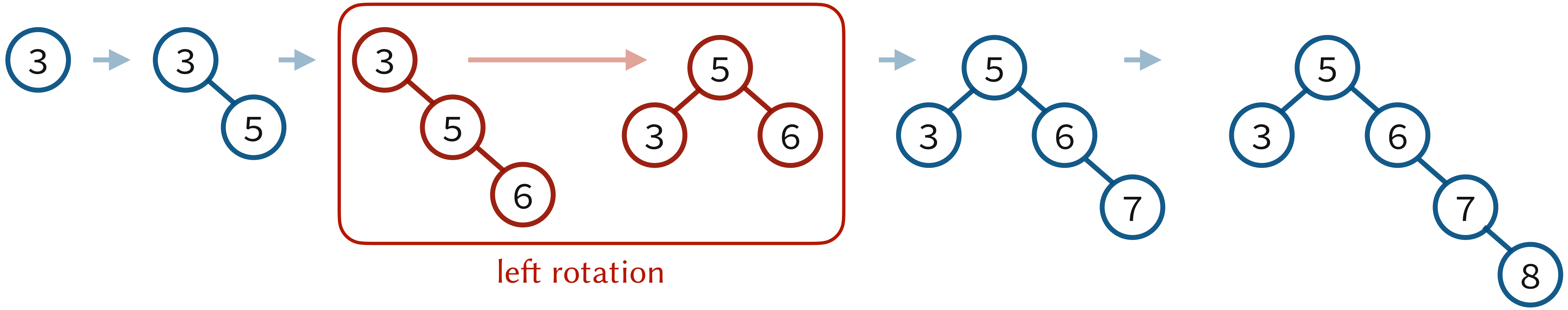
left rotation

AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example. **3** **5** **6** **7** **8** 9 4 1 2

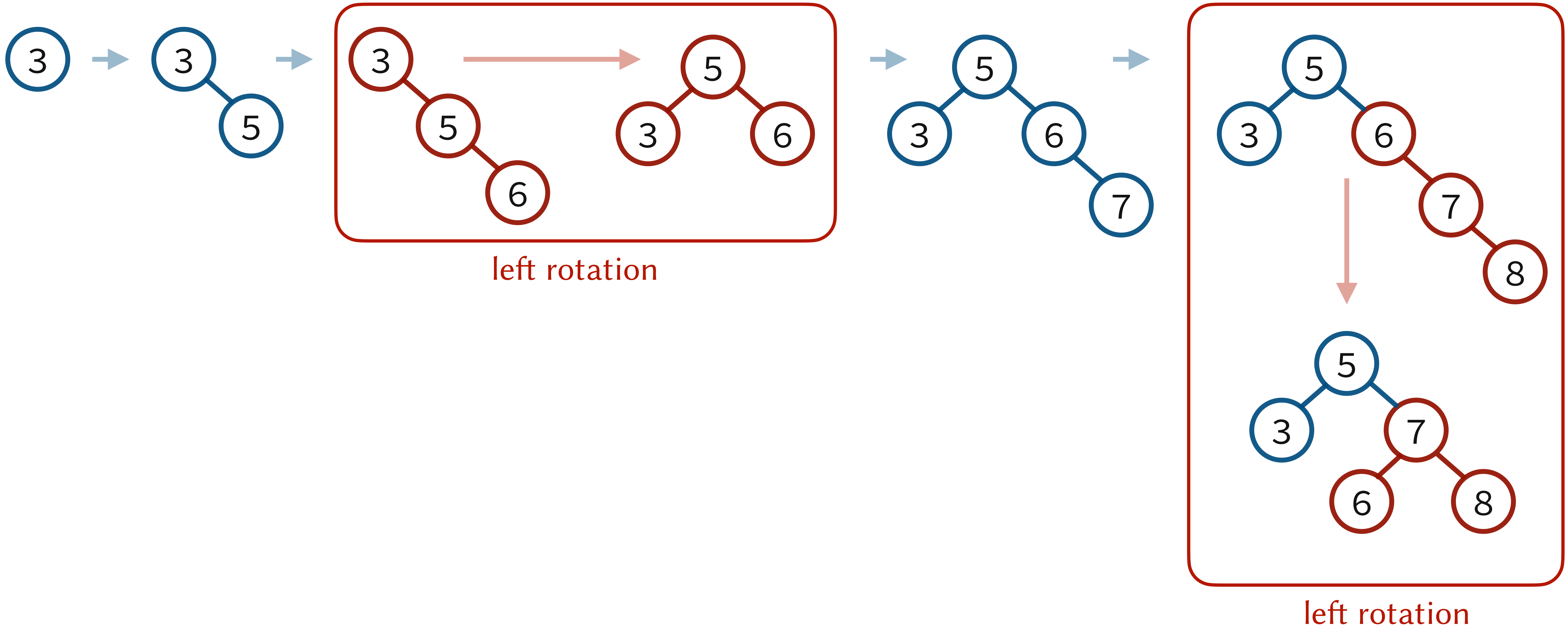


AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example. 3 5 6 7 8 9 4 1 2



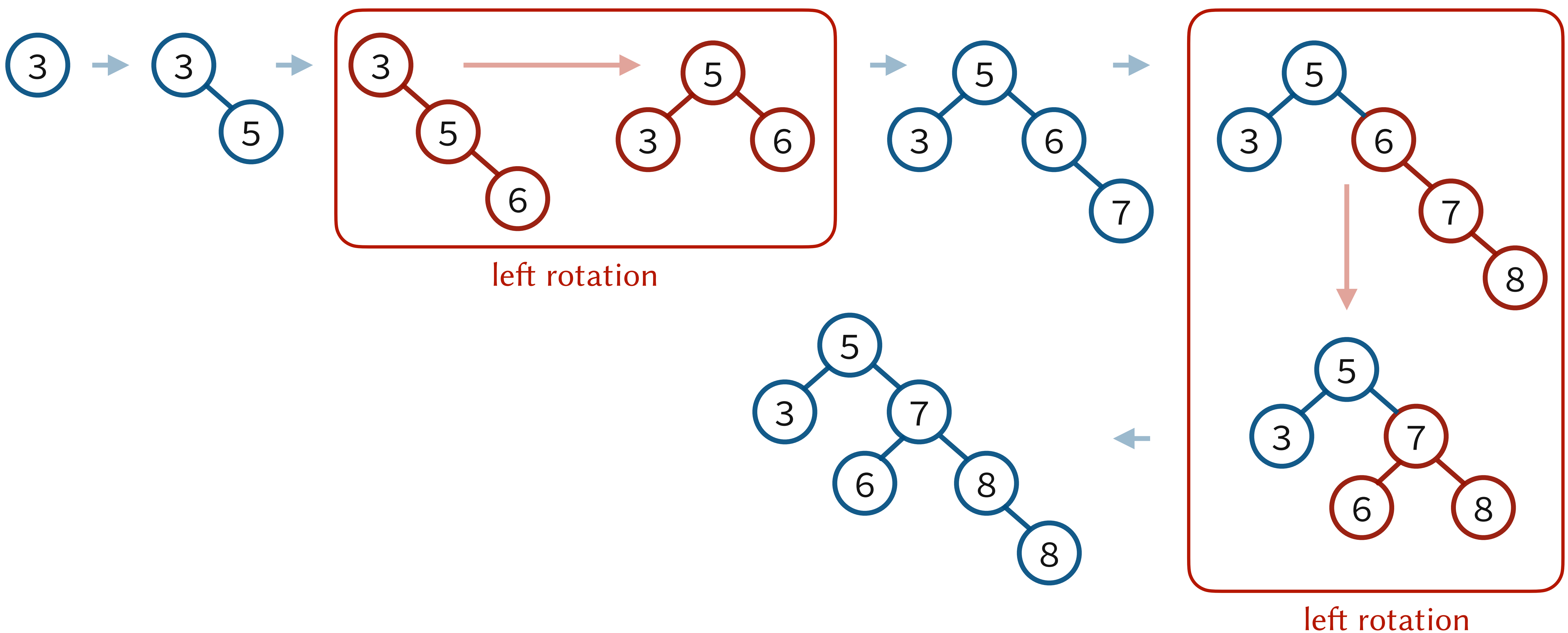
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	**7**	**8**	**9**	4	1	2



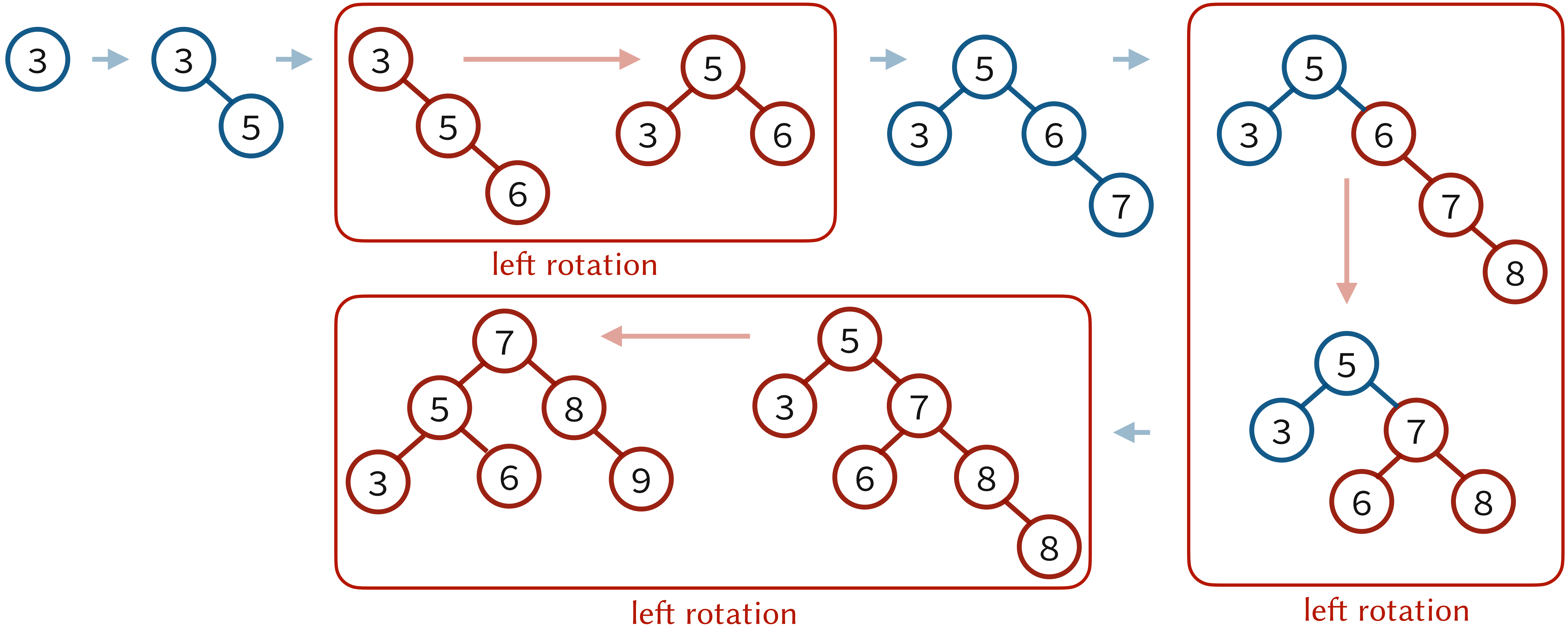
AVL Trees

After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	**7**	**8**	**9**	4	1	2



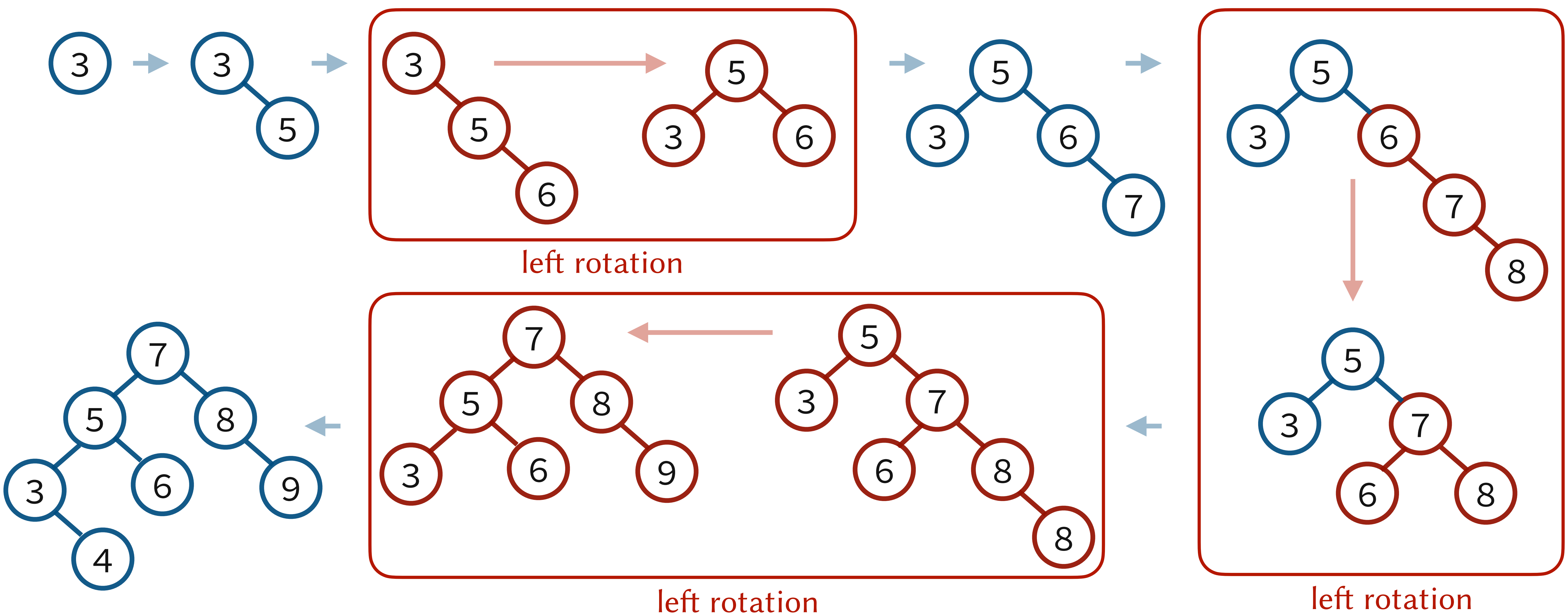
AVL Trees

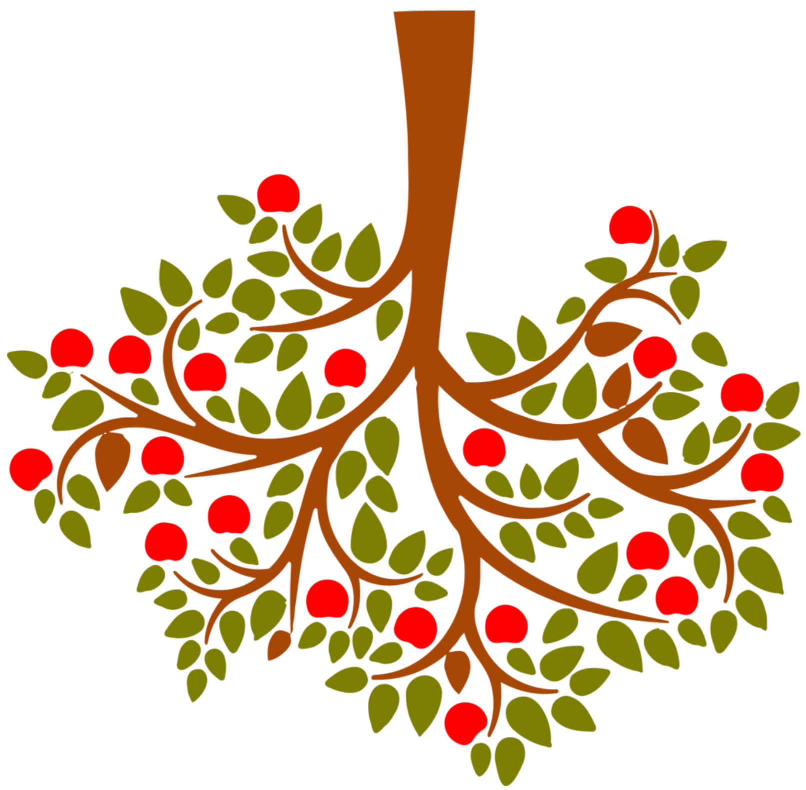
After every insertion (or deletion):

- Check the balance factors of the nodes on the insertion (or deletion) path from the lowest to the highest in the tree.
- Perform the appropriate rotation on every subtree that is not balanced.

Example.

3	**5**	**6**	**7**	**8**	**9**	**4**	1	2





Tree Data Structures

- Definitions and properties
- Basic operations
- **Balanced binary search trees**
- Tree traversals