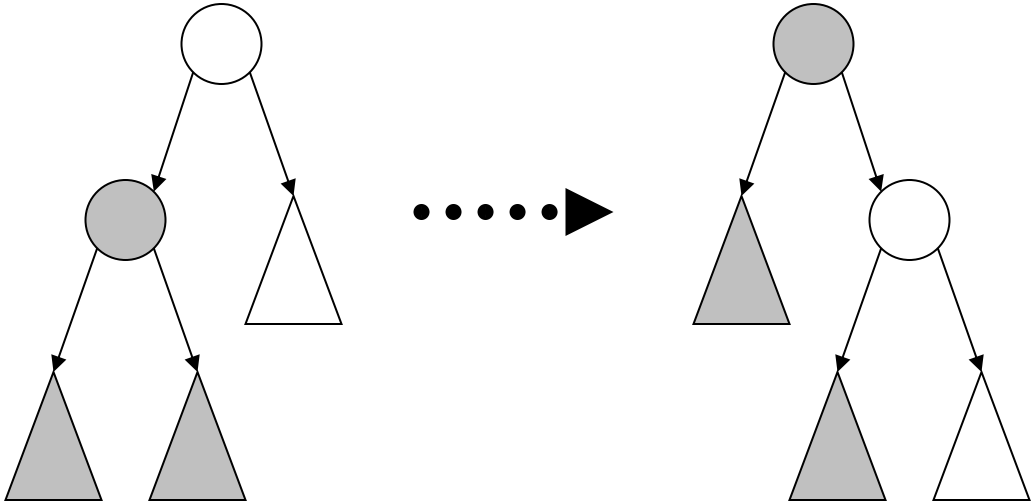
Tree Rotations

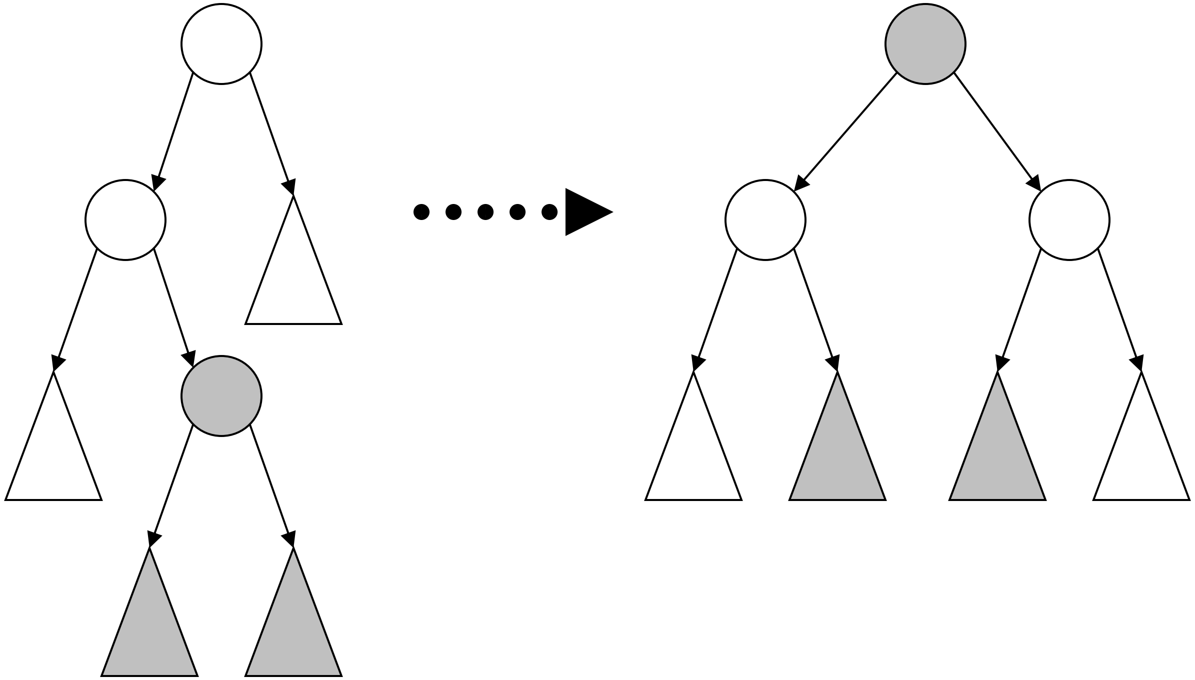
Use tree rotations to balance binary trees

Zig



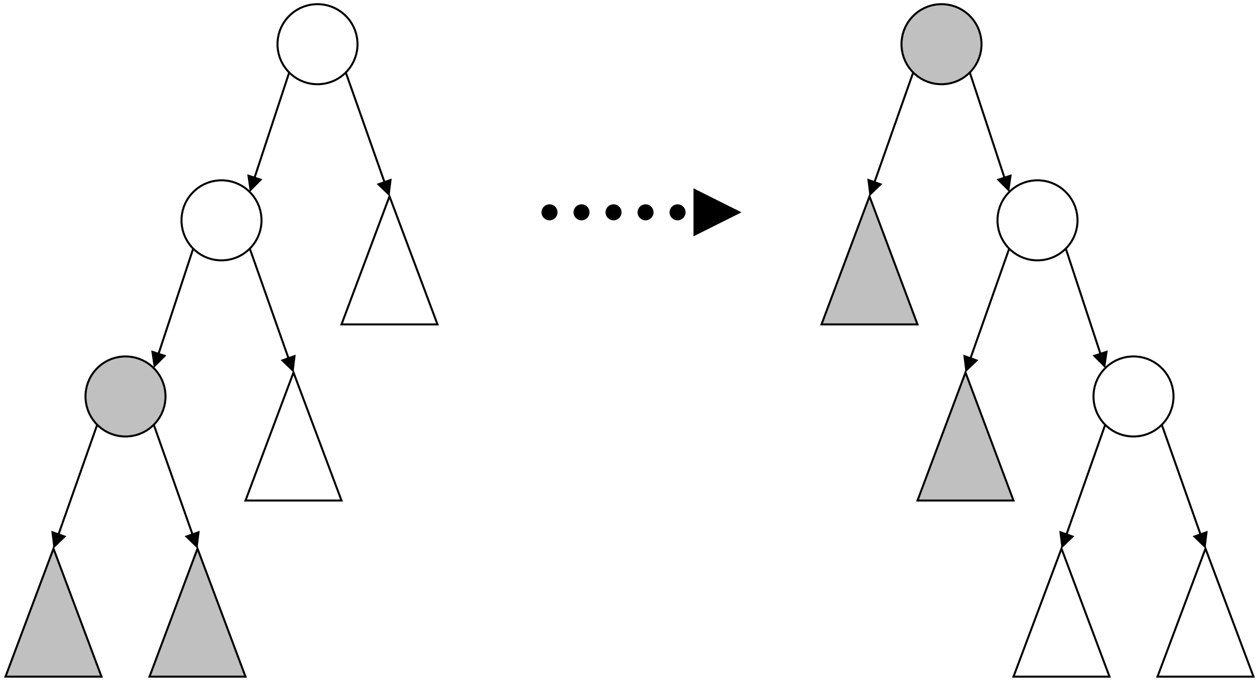
```
function method Zig( t: BST ): BST
    requires t != BSTEmpty
    requires Left(t) != BSTEmpty
    ensures TreeSeq(Zig(t)) == TreeSeq(t)
    ensures RootValue(Zig(t)) == RootValue(Left(t))
{
    match t
    case BSTNode(BSTNode(A,x,B),y,C) => BSTNode(A,x,BSTNode(B,y,C))
}
```

ZigZag



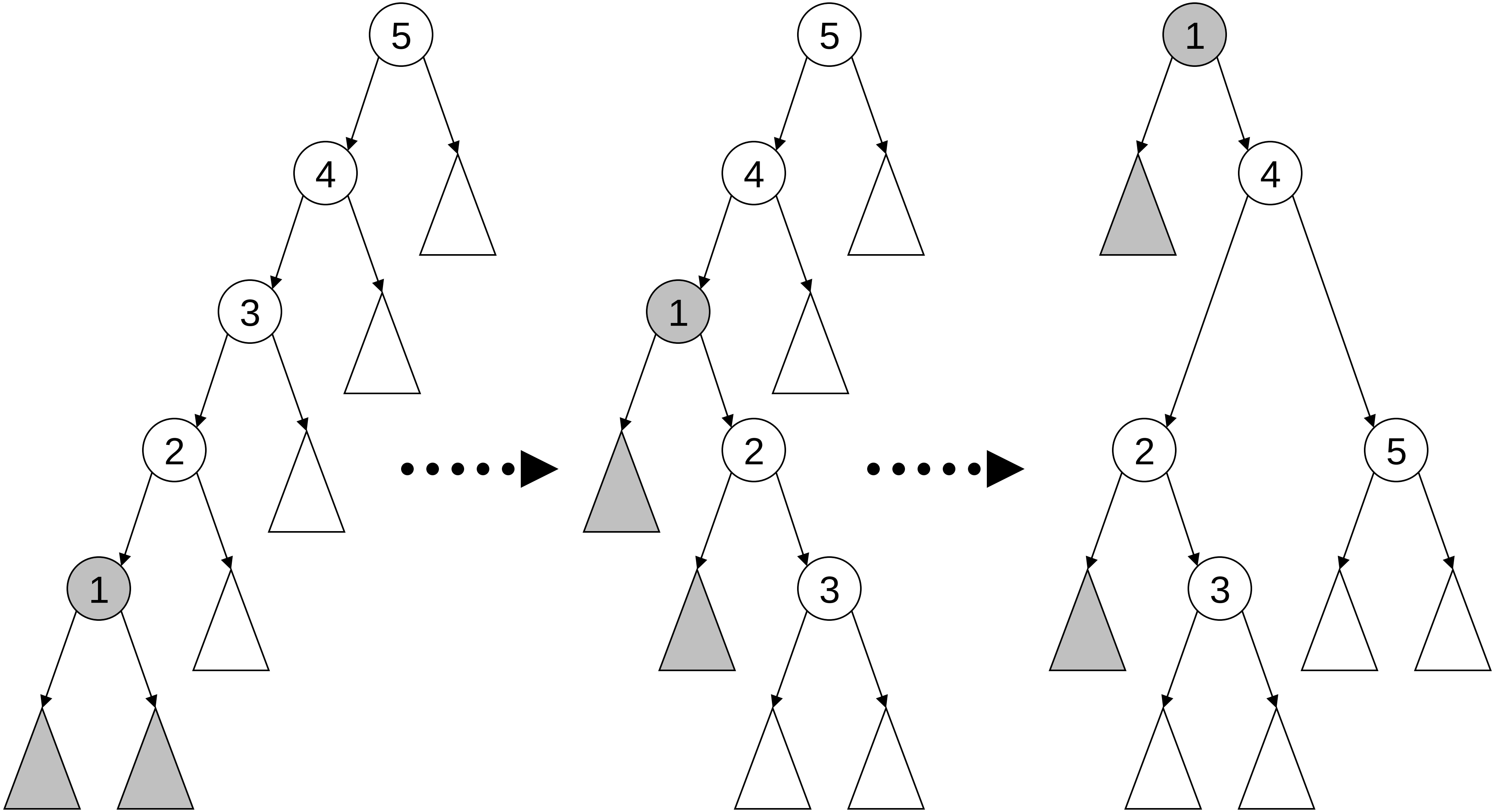
```
function method ZigZag( t: BST ): BST
    requires t != BSTEmpty
    requires Left(t) != BSTEmpty
    requires Right(Left(t)) != BSTEmpty
    ensures TreeSeq(t) == TreeSeq(ZigZag(t))
    ensures RootValue(Zig(t)) == RootValue(Right(Left(t)))
{
    match t
    case BSTNode(A,x,B) => Zig(BSTNode(Zag(A),x,B))
}
```

ZigZig



```
function method ZigZig( t: BST ): BST
    requires t != BSTEmpty
    requires Left(t) != BSTEmpty
    requires Left(Left(t)) != BSTEmpty
    ensures TreeSeq(t) == TreeSeq(ZigZig(t))
    ensures RootValue(ZigZig(t)) == RootValue(Left(Left(t)))
{
    Zig(Zig(t))
}
```

Splay

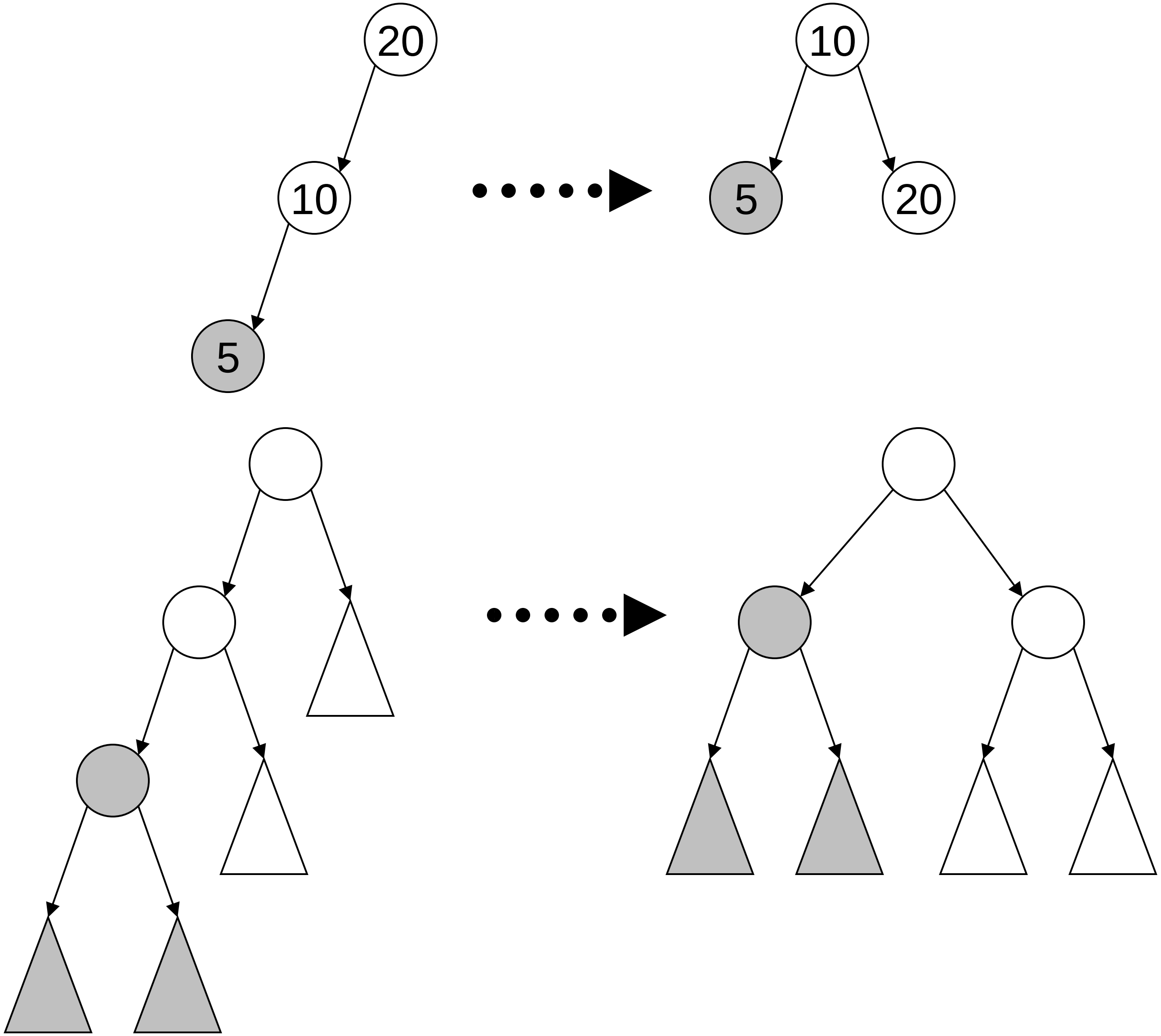


```
method SplayMin( t: BST ) returns( s: BST )
    decreases t
    ensures TreeSeq(s) == TreeSeq(t)
    ensures s == BSTEmpty || Left(s) == BSTEmpty
{
    match t
    case BSTEmpty => { return BSTEmpty; }
    case BSTNode(left,x,right) =>
    {
        match left
        case BSTEmpty =>              { s := t;    }
        case BSTNode(BSTEmpty,_,_) => { s:=Zig(t); }
        case BSTNode(left2,y,right2) =>
        {
            var newleft2 := SplayMin(left2);
            assert TreeSeq(BSTNode(newleft2,y,right2))
                   == TreeSeq(BSTNode(left2,y,right2));
            s := BSTNode(BSTNode(newleft2,y,right2),x,right);
            s := ZigZig(s);
        }
    }
}
```

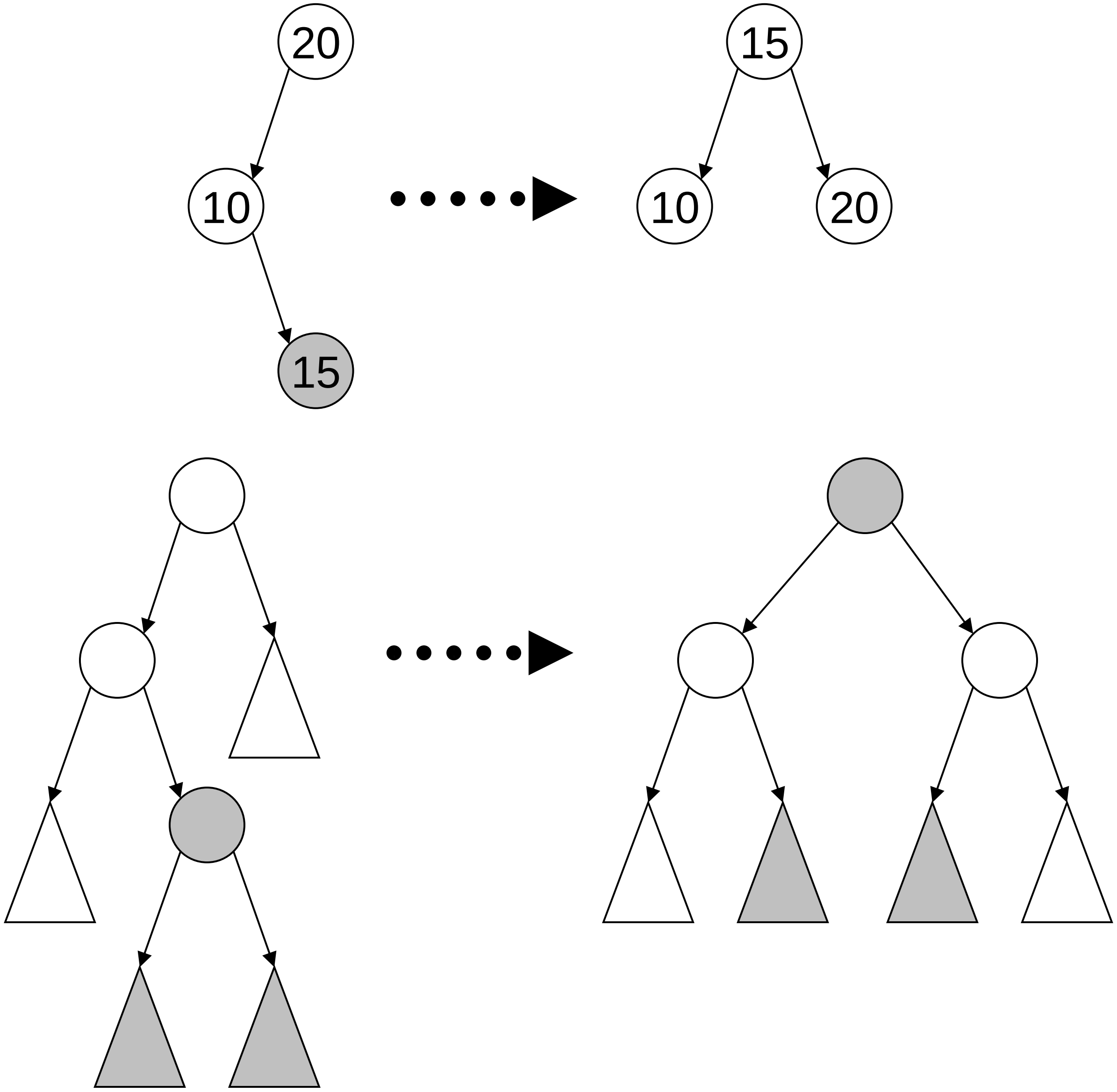
Simple Insertion

```
method Insert( t: BST, x: int ) returns ( r: BST )
    decreases t
    requires TreeIsSorted(t)
    ensures TreeIsSorted(r)
    ensures multiset(TreeSeq(r)) == multiset(TreeSeq(t))+multiset{x}
    ensures forall z | z in TreeSeq(r) :: z == x || z in TreeSeq(t)
{
    if t == BSTEmpty
    {
        r := BSTNode(BSTEmpty,x,BSTEmpty);
    }
    else if x < RootValue(t)
    {
        r := Insert(Left(t),x);
        r := BSTNode(r,RootValue(t),Right(t));
    }
    else
    {
        r := Insert(Right(t),x);
        r := BSTNode(Left(t),RootValue(t),r);
    }
}
```

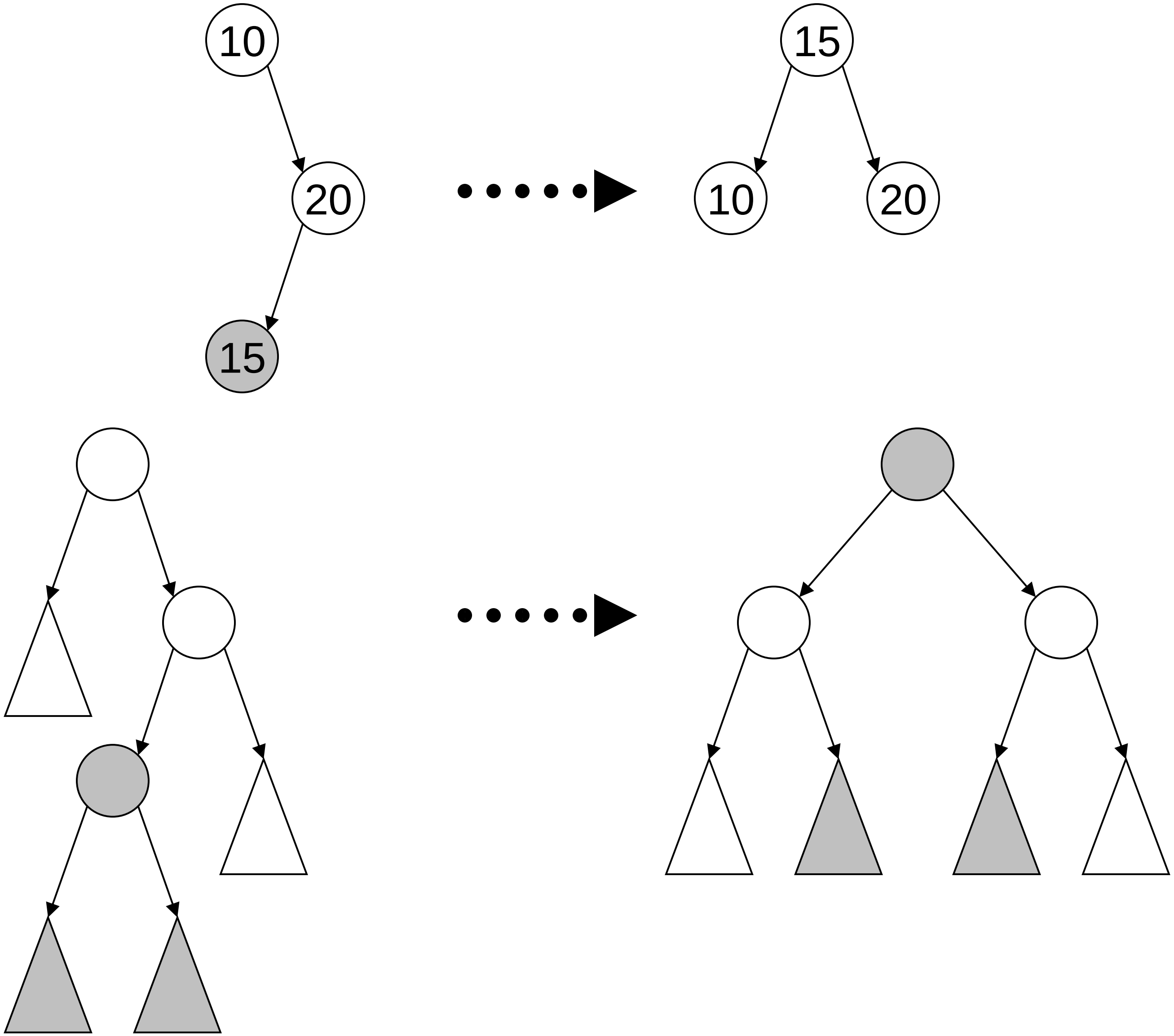
Inserting and Rotating



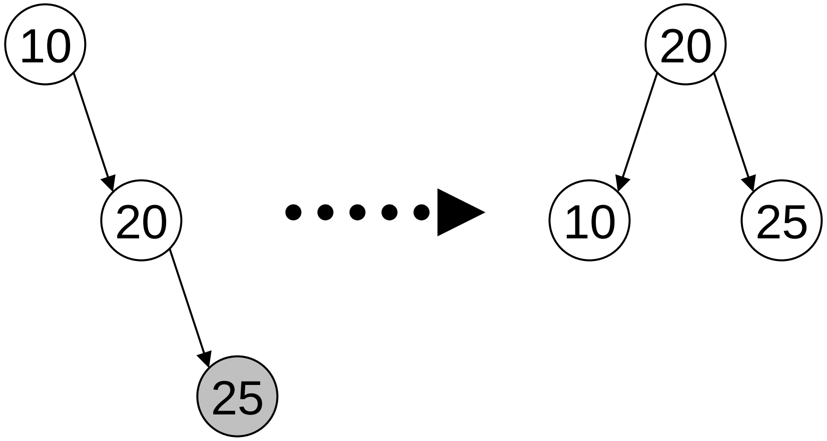
Inserting and Rotating

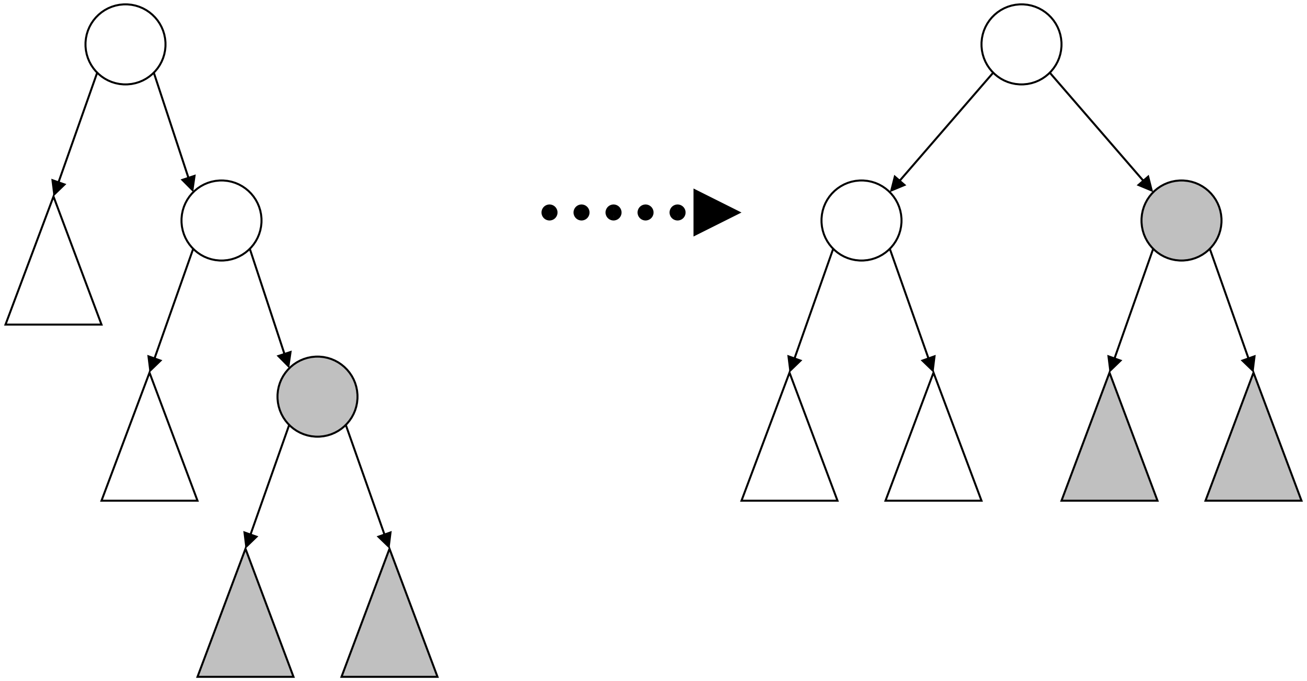


Inserting and Rotating



Inserting and Rotating

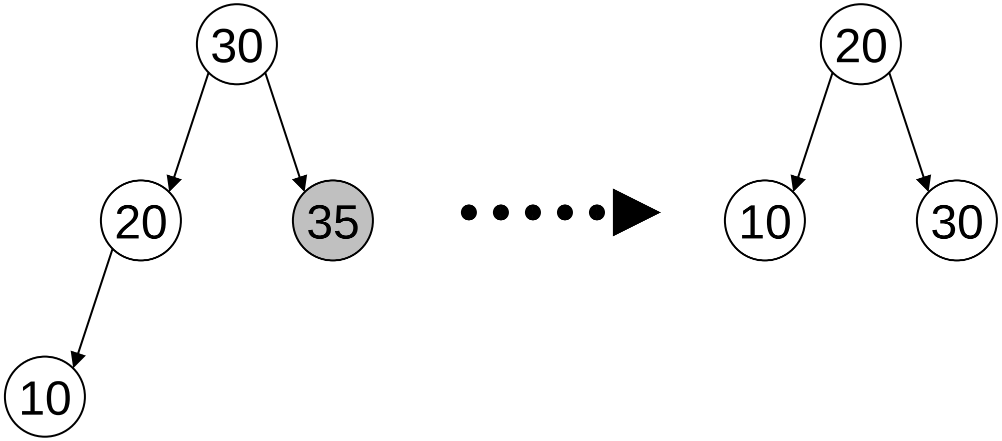


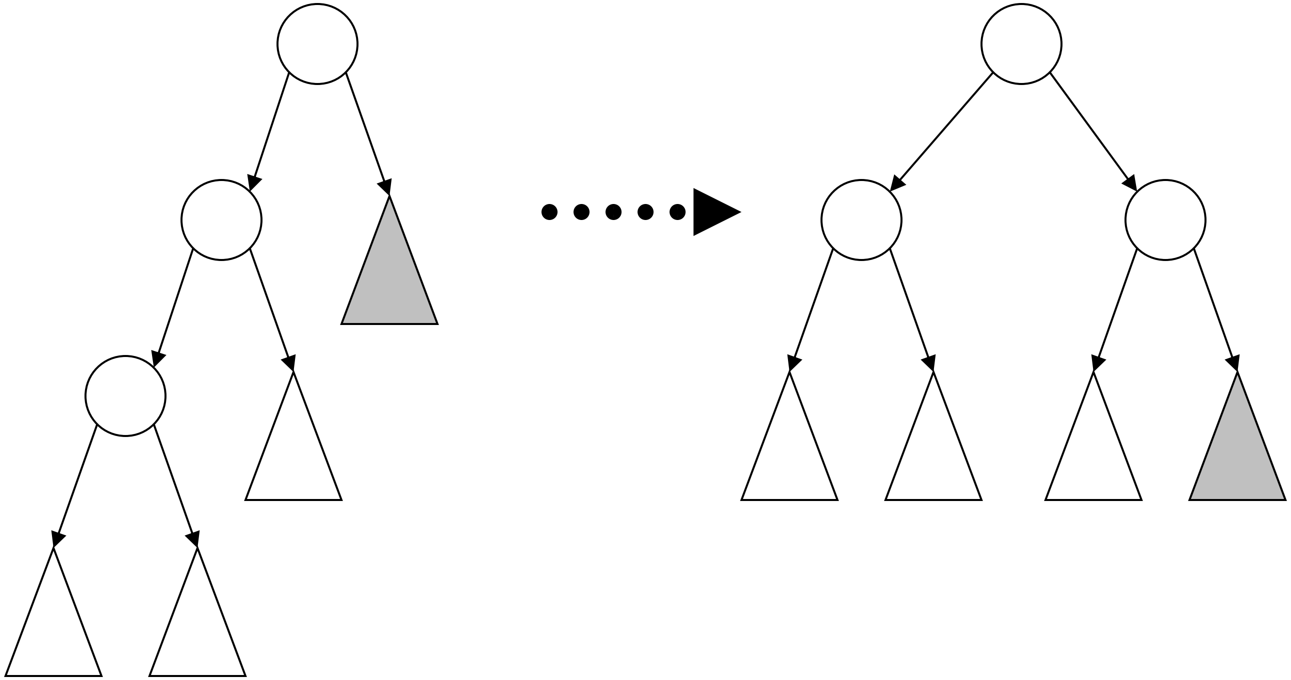


Simple Deletion

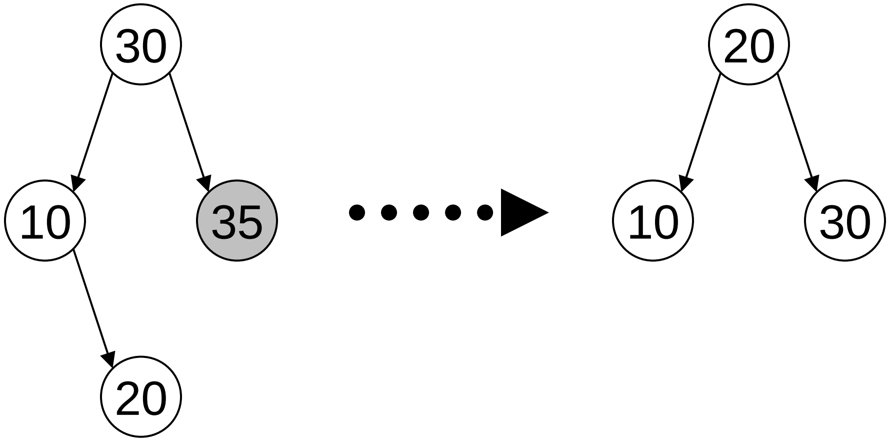
```
method Delete( t: BST, x: int ) returns ( r: BST )
    decreases t
    requires TreeIsSorted(t)
    ensures TreeIsSorted(r)
    ensures multiset(TreeSeq(r)) == multiset(TreeSeq(t))-multiset{x}
    ensures x !in TreeSeq(t) ==> r == t
    ensures forall z | z in TreeSeq(r) :: z in TreeSeq(t)
{
    if t == BSTEmpty { return BSTEmpty; }
    else if RootValue(t) == x
    {
        if Left(t) == BSTEmpty  { return Right(t); }
        if Right(t) == BSTEmpty { return Left(t);  }
        var newright, min := DeleteMin(Right(t));
        r := BSTNode(Left(t),min,newright);
    }
    else if x < RootValue(t)
    {
        assert forall z | z in multiset(TreeSeq(Right(t))) ::
               z in TreeSeq(Right(t));
        var newleft := Delete(Left(t),x);
        r := BSTNode(newleft,RootValue(t),Right(t));
    }
    else
    {
        var newright := Delete(Right(t),x);
        r := BSTNode(Left(t),RootValue(t),newright);
        assert forall z | z in multiset(TreeSeq(Left(t))) :: z in TreeSeq(Left(t));
    }
}
```

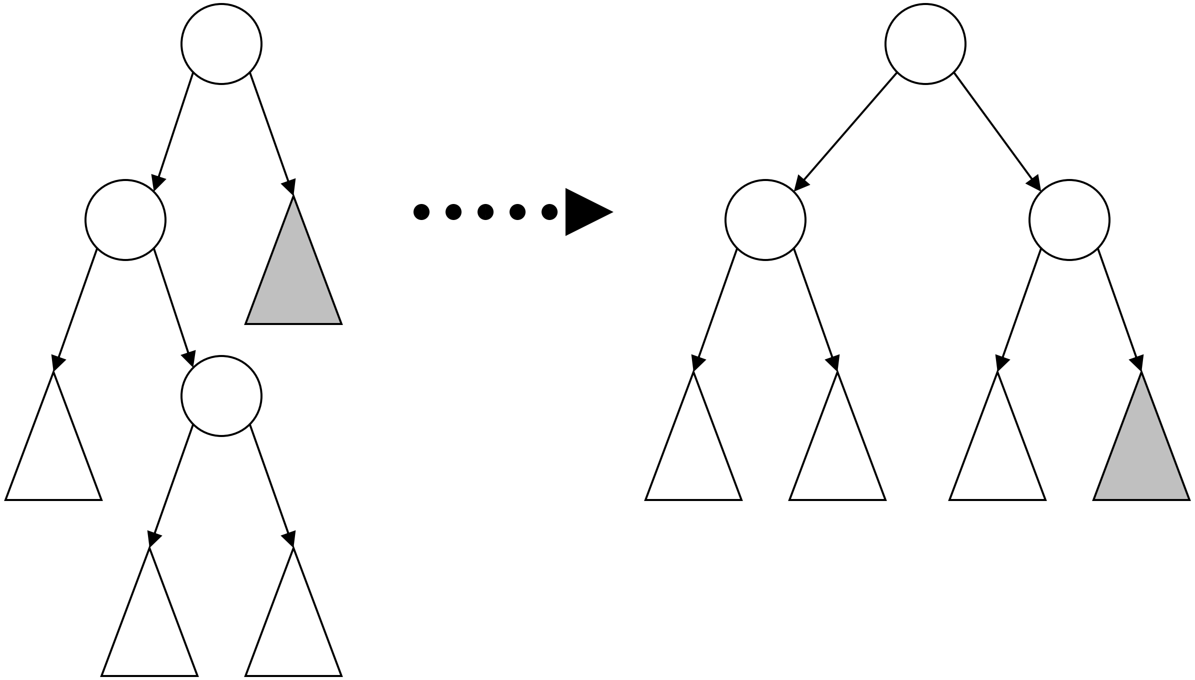
Deleting and Rotating



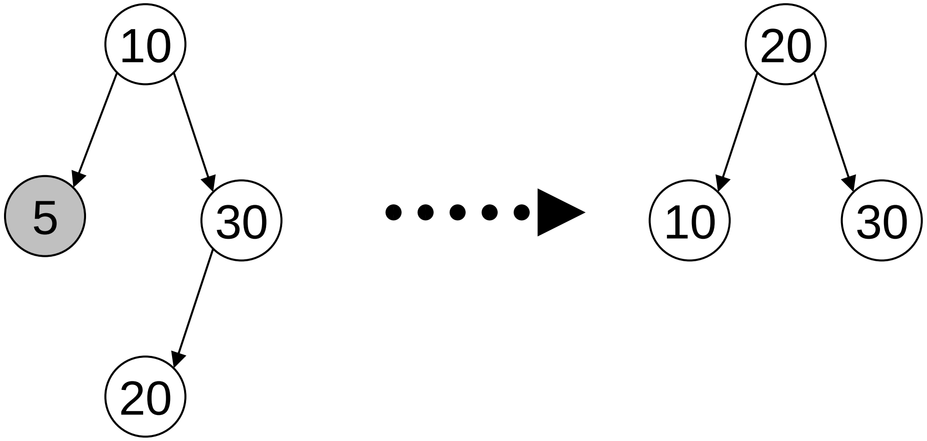


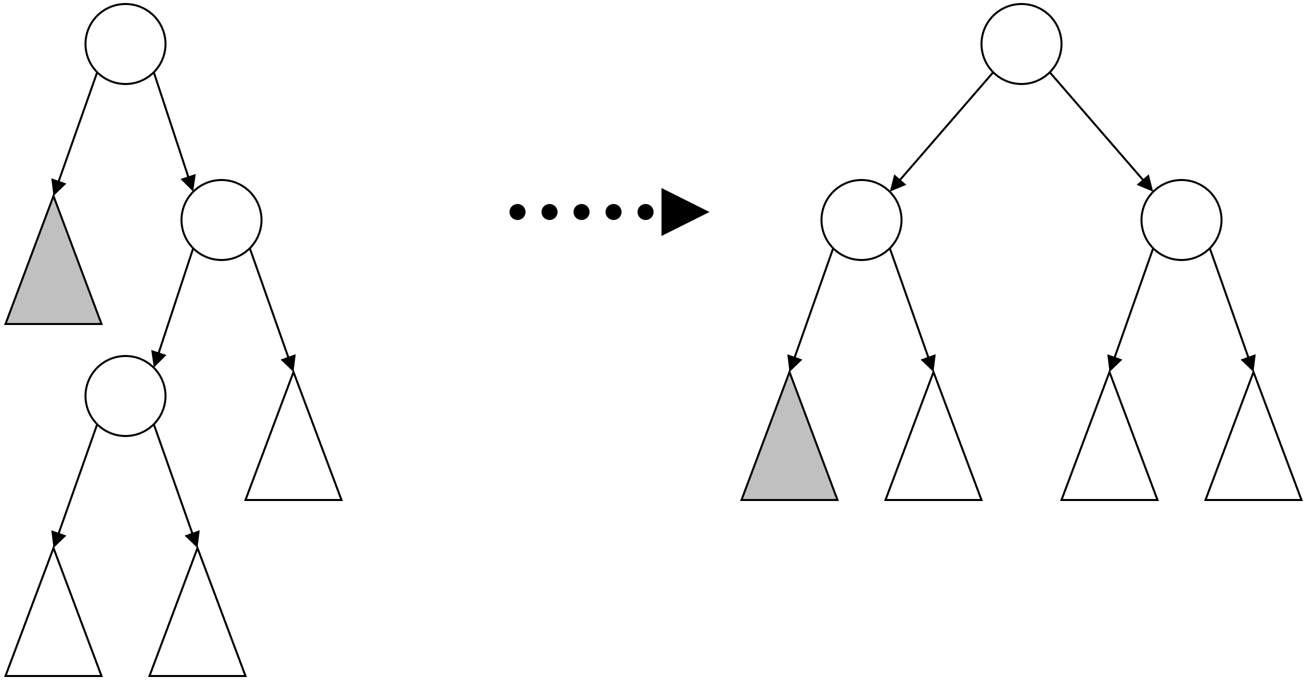
Deleting and Rotating



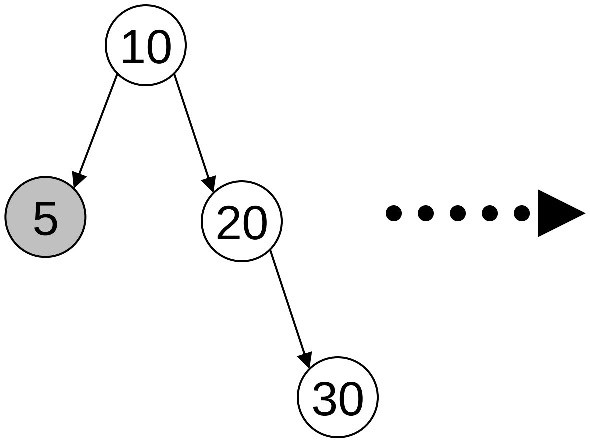


Deleting and Rotating





Deleting and Rotating


10
5
20
30
20
10
30