Binary Search Trees

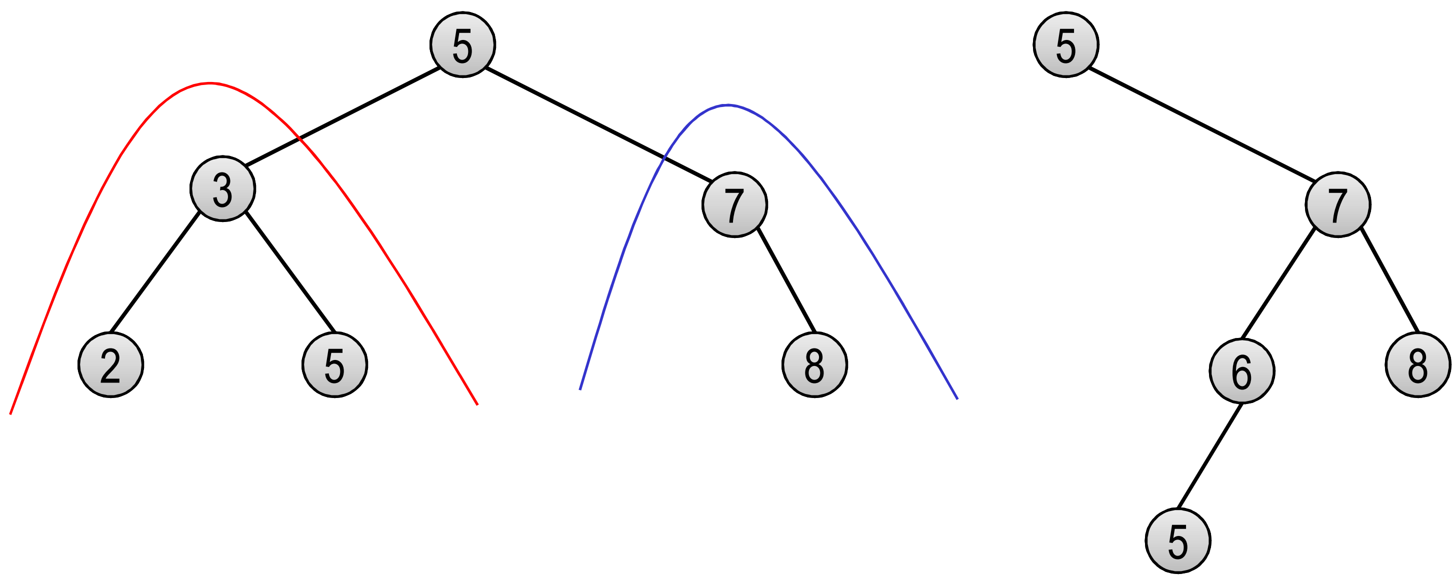
Data Structures and Algorithms
Andrei Bulatov

Binary Search Tree

A binary search tree is a binary rooted tree, in which keys satisfy the Binary Search Tree property:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key[y] ≤ key[x].

If y is a node in the right subtree of x, then key[x] ≤ key[y]



Inorder Tree Walk

Having a binary search tree one can print its content in sorted order

Inorder-Tree-Walk(x)

```
if x≠Nil then do
     Inorder-Tree-Walk(left[x])
     print key[x]
     Inorder-Tree-Walk(right[x])
```

To print the entire tree just call Inorder-Tree-Walk(root[T])

It is called inorder because the root is printed between the subtrees

A tree walk can also be preorder and postorder

Inorder Tree Walk (cntd)

Lemma

If x is the root of an n-node subtree, then the call Inorder-Tree-Walk(x) takes $\Theta(n)$ time

Proof

Let $T(n)$ denote the running time

If $x = Nil$ then $T(0) = c$, a constant

If $n > 0$ then $T(n) = T(k) + T(n - k - 1) + d$ where k is the number of nodes in the left subtree

We prove that $T(n) = (c+d)\, n + c$

For $n = 0$ we have $(c + d) \cdot 0 + c = c = T(0)$

Inorder Tree Walk (cntd)

Proof

For $n > 0$ we have

$$
\begin{aligned}
T(n) &= T(k) + T(n - k - 1) + d \\
&= ((c + d)\,k + c) + ((c + d)\,(n - k - 1) + c) + d \\
&= (c + d)\,n + c - (c + d) + c + d \\
&= (c + d)\,n + c
\end{aligned}
$$

QED

Searching

Elements of a binary search tree can be found efficiently

Tree-Search(x,k)

```
if x=Nil or k=key[x] then
   return x
if  k<key[x] then
   return Tree-Search(left[x],k)
else
   return Tree-Search(right[x],k)
```

Minimum and Maximum

We can find minimum and maximum keys in the tree

Tree-Minimum(x)

```
while left[x]≠Nil do
   set x:=left[x]
endwhile
return x
```

Tree-Maximum(x)

```
while right[x]≠Nil do
   set x:=right[x]
endwhile
return x
```

Successor

We can find the successor of a key in the tree

Tree-Successor(x)

```
if right[x]≠Nil then
   return Tree-Minimum(right[x])
endif
set y:=parent[x]
while y≠Nil and x=right[y] do
   set x:=y
   set y:=parent[y]
endwhile
return y
```

Running Time

Theorem

The operations Search, Minimum, Maximum, Successor, and Predecessor can be made to run in O(h) time on a binary search tree of height h.

Insertion

We insert a new value v into a binary search tree T.

The procedure is passed a node z for which key[z] = v, left[z] = Nil, and right[z] = Nil.

Tree-Insert(T,z)

```
set y:=Nil,   x:=root[T]
while x≠Nil do
   set y:=x
   if key[z]<key[x] then set x:=left[x]
                    else set x:=right[x]
endwhile
set parent[z]:=y
if y=Nil then set root[T]:=z
         else if key[z]<key[y] then set left[y]:=z
                               else set right[y]:=z
```

Deletion

We delete a given node z
Consider 3 cases:

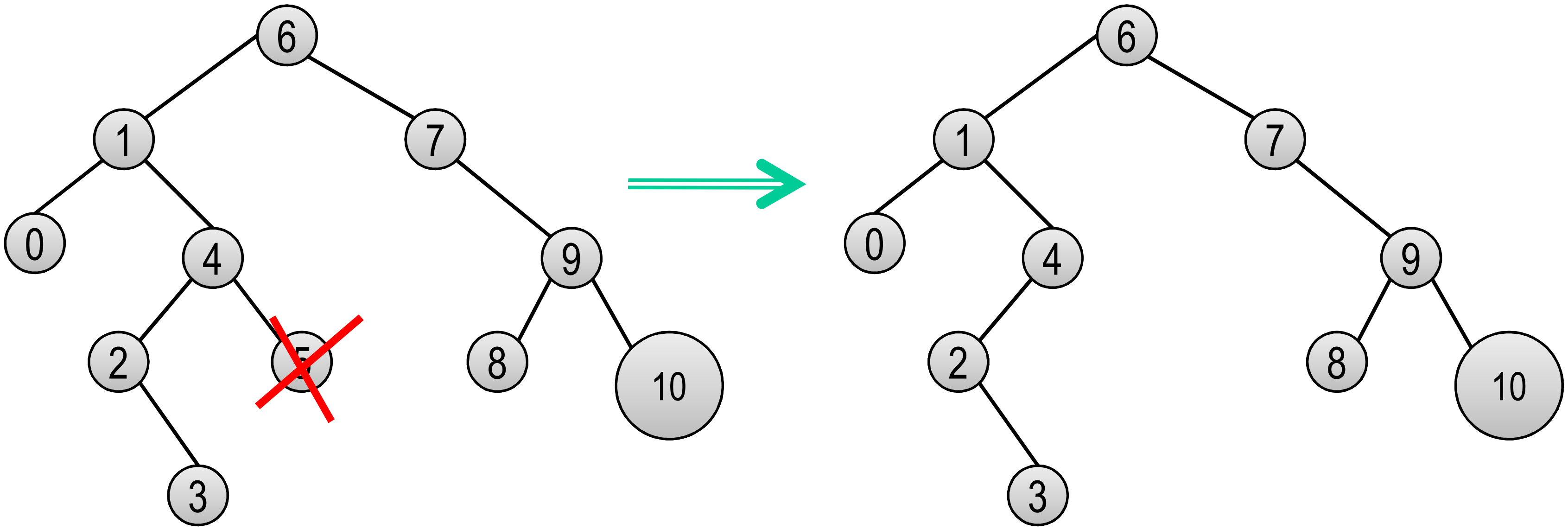
(i) z has no children

(ii) z has one child

(iii) z has 2 children

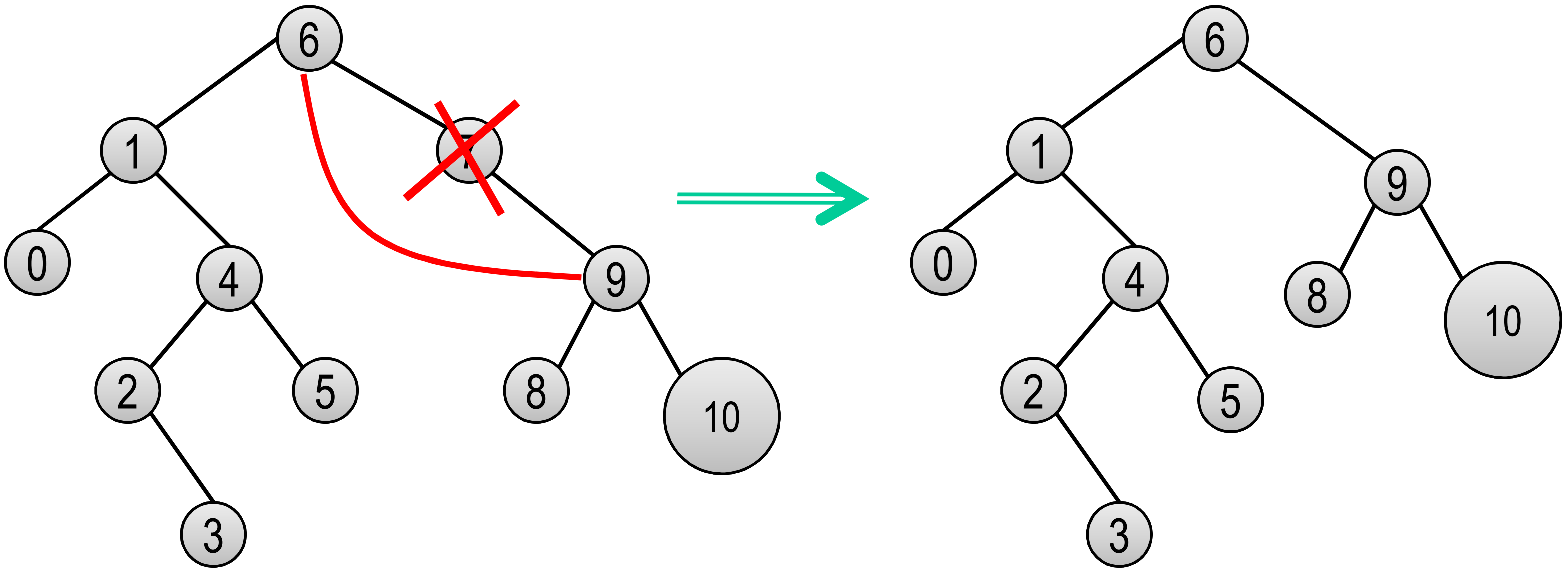
Deletion (cntd)

(i) z has no children



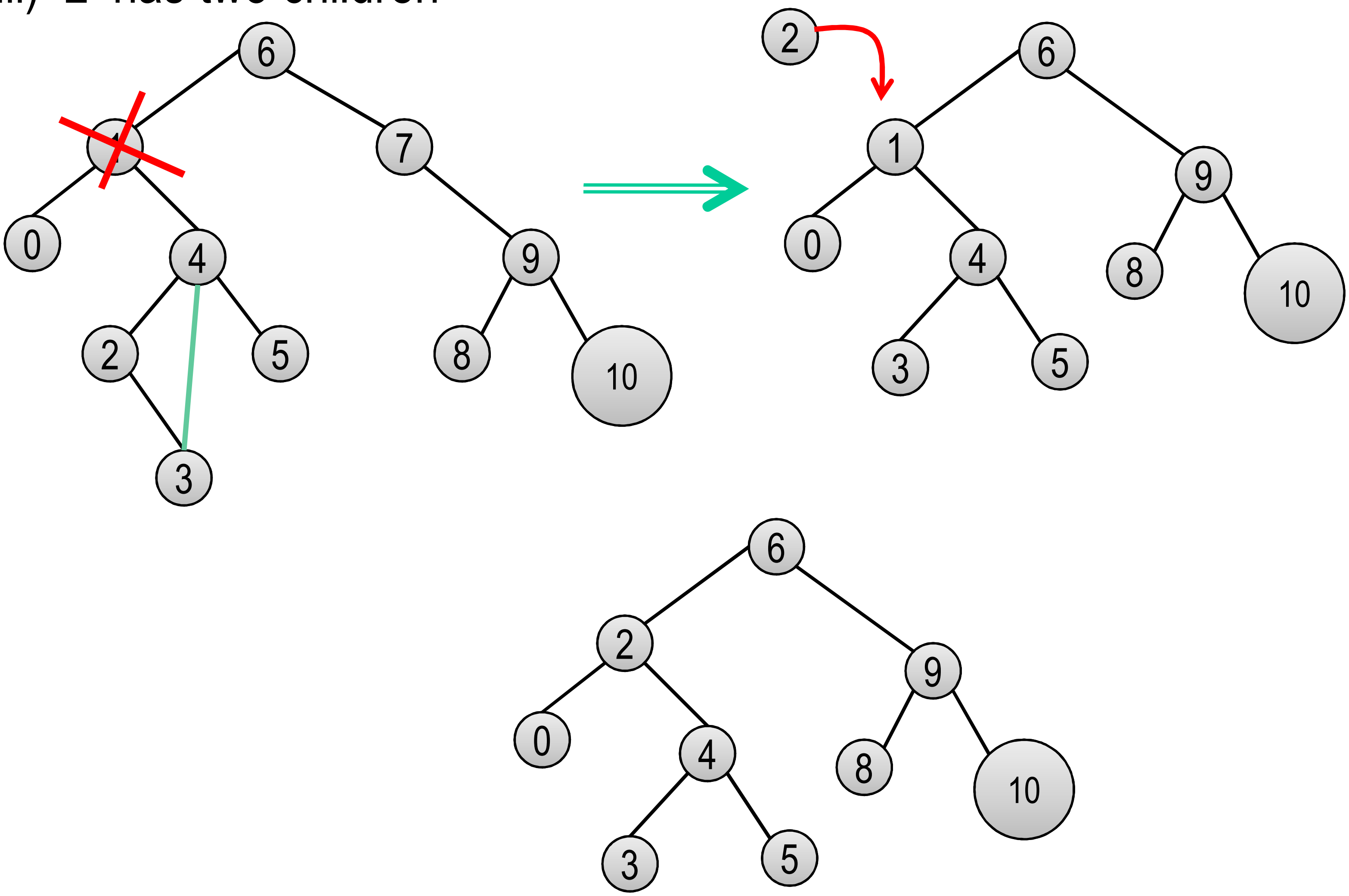
Deletion (cntd)

(ii) z has one child



Deletion (cntd)

(iii) z has two children



Deletion

Tree-Delete(T,z)

```
if left[z]=Nil or right[z]=Nil then set y:=z
                              else set y:=Tree-Successor(z)
if left[y]≠Nil then set x:=left[y]
               else set x:=right[y]
if x≠Nil then set parent[x]:=parent[y]
                    else set x:=right[x]
if parent[y]=Nil then set root[T]:=x
 else if y=left[parent[y]]:=x then left[parent[y]:=x
                            else right[parent[y]]:=x
if y≠z then do
  set key[z]:=key[y]
  copy y's data into z
return y
```

Red-Black Trees

All binary search tree operations take O(h) time, where h is the height of the tree

Therefore, it is important to `balance' the tree so that its height is as small as possible

There are many ways to achieve this

One of them: Red-Black trees

Every node of such a tree contains one extra bit, its color

Another agreement: the Nil pointer is treated as a leaf, an extra node

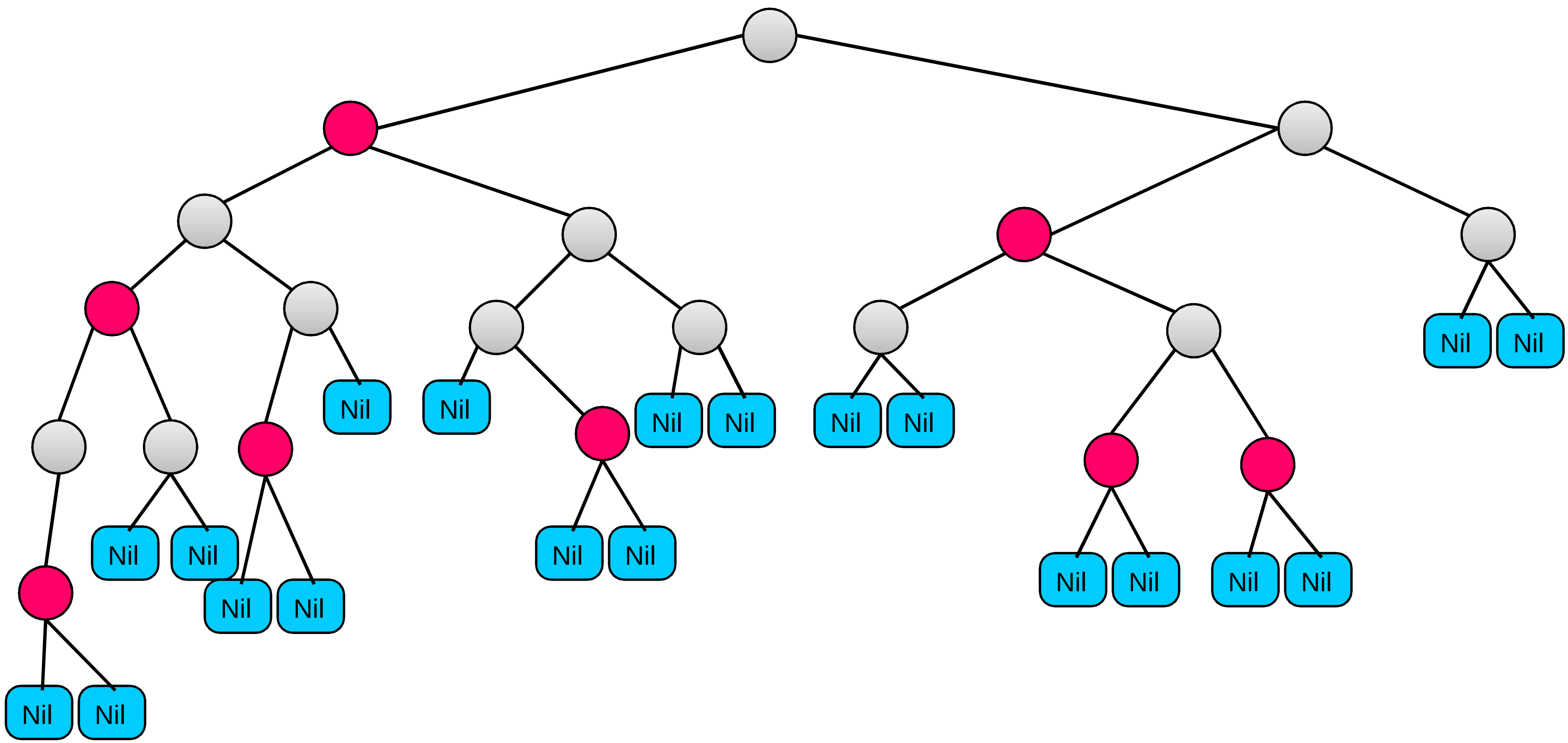
The rest of the nodes are called internal

Red-Black Properties

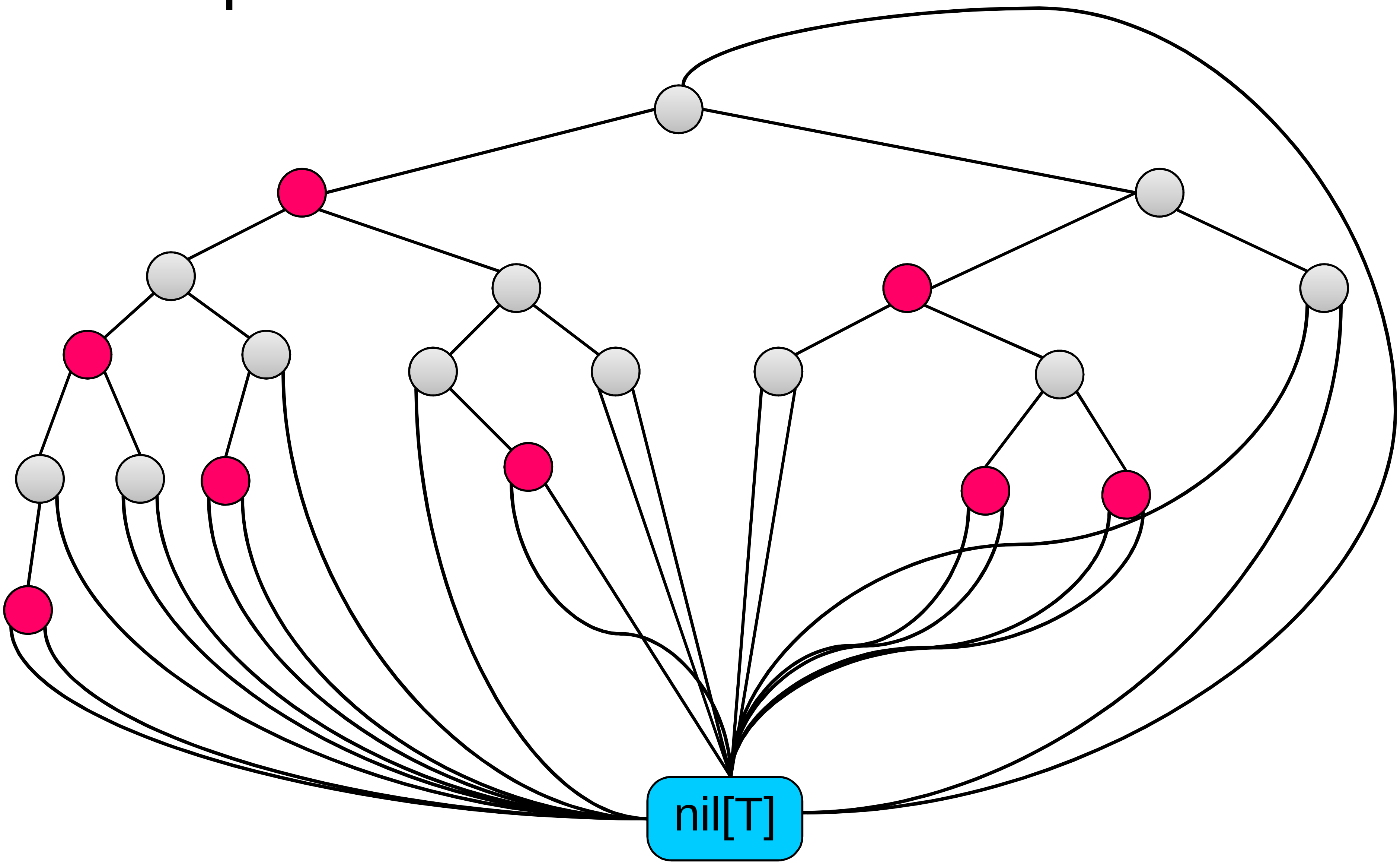
A binary search tree is a red-black tree if it satisfies the following red-black properties:

- Every node is either red or black
- The root is black
- Every leaf (Nil) is black
- If a node is red, then both its children are black
- For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example


Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil

Example



Black Height

The number of black nodes on paths from node x to its descendant leaves in a red-black tree is called its black height, denoted bh(T)

Lemma

A red-black tree with n internal nodes has height at most 2·log(n + 1)

Proof

We show first that the subtree rooted at x contains at least $2^{bh(x)} - 1$ nodes

Induction on bh(x)

Base Case: If bh(x) = 0, then x is a leaf, nil[T]

In this case, $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

Black Height (cntd)

Inductive hypothesis: the claim is true for any y with height less than that of x

Inductive Case: Let bh(x) > 0 and has two children

The black height of the children is either bh(x) or bh(x) – 1, depending on its color

Since the children have smaller height, we can apply the induction hypothesis

Thus the subtree rooted at x contains at least

$$(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$$

nodes

Black Height (cntd)

Suppose h is the height of the tree

By the red-black property at least half of nodes on every root-to-leaf path are black (not including the root)

Therefore the black height of the root is at least h/2

Thus

$$n \geq 2^{h/2} - 1$$
$$\log(n+1) \geq h/2$$

QED