cis112
Binary Search Trees

Haluk O. Bingol

Faculty of Computer and Information Sciences
Yeditepe University

v2024-04-25

Content

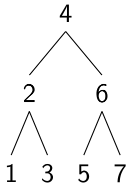
Motivation

Game of secret number

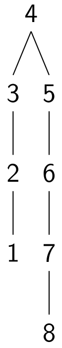
Game.

- Player-A selects a secret number in the range of $\{1, 2, \ldots, 8\}$.
- Game repeats till the secret is found.
  - Player-B tries a number.
  - Player-A should answer as "found", or "smaller" or "larger".

Q. What is the best strategy if the range becomes large such as $\{1, 2, \ldots, 1000, 000\}$.



This strategy finds the secret faster.



This strategy is slower.

Definitions

Binary Search Tree

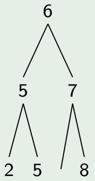
Definition

Binary search tree property :
Let x be a node in a binary tree.

1. If y is a node in the left subtree of x, then $y.key \leq x.key$.
2. If y is a node in the right subtree of x, then $y.key \geq x.key$.

A binary tree with the binary-search-tree property is called a binary search tree.

Example



2 is in the left subtree of 5 since $2 \leq 5$.
2 is in the left subtree of 6 since $2 \leq 6$.
8 is in the right subtree of 7 since $8 \geq 7$.
8 is in the right subtree of 6 since $8 \geq 6$.
[[2]]

[[1], [2], [3], [4], [5], [6], [7]]

Algorithms

Node

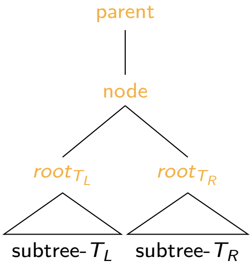
Algorithm 1: Node for Binary Search Tree

```
begin
    data
    parent                          // reference to the parent
    left                        // reference to the left child
    right                      // reference to the right child
```

[[2]]

Warning. `key` is comparable:
Comparison of `a.data` and `b.data` for Nodes, `a` and `b`, is defined as

$$\texttt{a.key.comparedTo(b.key)} = \begin{cases} < 0, & \texttt{a.data < b.data}, \\ 0, & \texttt{a.data == b.data}, \\ > 0, & \texttt{a.data > b.data}. \end{cases}$$



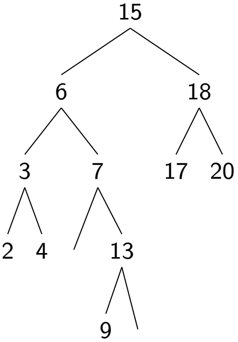
Algorithms

Tree Traversal

Inorder Tree Traversal

Function InorderTreeWalk(x) [[2]]

```
if x ≠ NIL then
    InorderTreeWalk(x.left)
    print x.data
    InorderTreeWalk(x.right)
```



Output of `InorderTreeWalk` (15)
2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20.

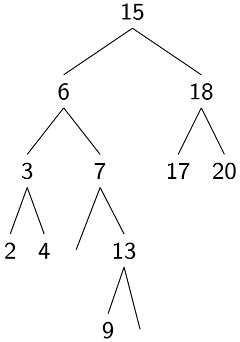
Algorithms

Searching

Tree Search

Function TreeSearch(x, k) [[2]]

```
if x == NIL or key == x.key then
    return x
if  k < x.key then
    return TreeSearch( x.left, k)
else
    return TreeSearch( x.right, k)
```



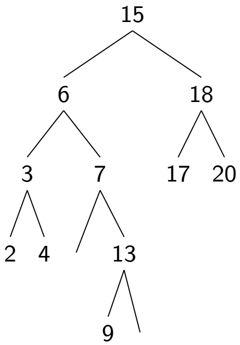
Trace of `TreeSearch` (15, 14)
`TreeSearch` (15,14)
`TreeSearch` (6,14)
`TreeSearch` (7,14)
`TreeSearch` (13,14)
`TreeSearch` (NIL,14)

Trace of `TreeSearch` (15, 13)
`TreeSearch` (15,13)
`TreeSearch` (6,13)
`TreeSearch` (7,13)
`TreeSearch` (13,13)

Iterative Tree Search

Function IterativeTreeSearch(x, k) [[2]]

```
1 while x ≠ NIL and k ≠ x.key do
2     if k < x.key then
3         x = x.left
4     else
5         x = x.right
6 return x
```



Trace of IterativeTreeSearch (15, 13)
x: 15 // lines: 1-5
x: 6 // lines: 1-5
x: 7 // lines: 1-5
x: 13 // lines: 1-5

Algorithms

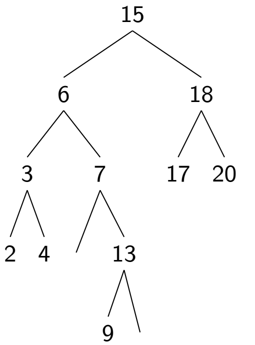
Minimum, Maximum, Successor, Predecessor

Tree Minimum

Function TreeMinimum(x) [[2]]

```
1 while x.left ≠ NIL do
2     x = x.left
3 return x
```

Q. `TreeMinimum` (6)?
Q. `TreeMinimum` (7)?
Q. `TreeMinimum` (18)?

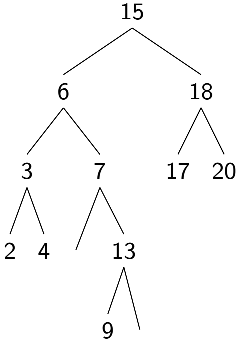


Trace of `TreeMinimum` (15)
15 // lines: 1-2
6 // lines: 1-2
3 // lines: 1-2
2 // lines: 1-2

Tree Maximum

Function TreeMaximum(x) [[2]]

```
1 while x.right ≠ NIL do
2     x = x.right
3 return x
```



Q. `TreeMaximum` (3)?
Q. `TreeMaximum` (4)?

Trace of `TreeMaximum` (6)
6 // lines: 1-2
7 // lines: 1-2
13 // lines: 1-2

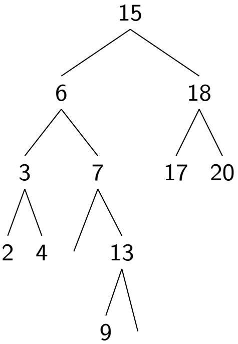
Trace of `TreeMaximum` (15)
15 // lines: 1-2
18 // lines: 1-2
20 // lines: 1-2

Tree Successor

Function TreeSuccessor(x) [[2]]

```
1 if x.right ≠ NIL then
                      // leftmost node in right subtree
2     return TreeMinimum (x.right)
3 else
                      // find the lowest ancestor of x,
                  // whose left child is an ancestor of x
4     y = x.parent
5     while y ≠ NIL and x == y.right do
6         x = y
7         y = y.parent
8 return y
```

Trace of TreeSuccessor(15)
15 // line: 2
18 // line: 2
17 // line: 2



Trace of TreeSuccessor (13)
x:13, y:7 // lines: 4-7
x:7, y:6 // lines: 4-7
x:6, y:15 // lines: 4-7

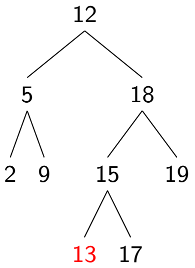
Algorithms

Insertion, Deletion

Tree Insert

Function TreeInsert(T, z) [[2]]

```
1  x = T.root                              // node being compared with z
2  y = NIL                                 // will be parent of z
3  while x ≠ NIL do
       // descend until reaching a leaf
4      y = x
5      if z.key < x.key then
6          x = x.left
7      else
8          x = x.right
9  z.parent = y   // found the location- insert z with parent y
10 if y == NIL then
11     T.root = z                          // tree was empty
12 else if z.key < y.key then
13     y.left = z
14 else
15     y.right = z
```



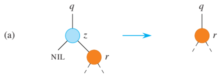
Trace of TreeInsert (T, 13)
x:12, y:NIL // lines: 1-2
x:18, y:12 // lines: 3-8
x:15, y:18 // lines 3-8
x:NIL, y:15 // lines: 3-8
15.left=13 // lines: 10-15

Deleting node z

Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange.

a) Node z has no left child.

a)



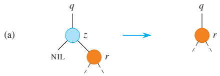
Deleting node z

Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange.

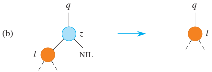
a) Node z has no left child.

b) Node z has a left child ℓ but no right child.

a)



b)



Deleting node z

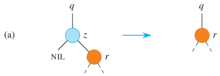
Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange.

a) Node z has no left child.

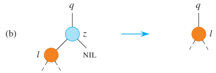
b) Node z has a left child ℓ but no right child.

c) Node z has two children. Its left child is node ℓ, its right child is its successor y (which has no left child), and y's right child is node x.

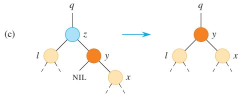
a)



b)



c)



Deleting node z

Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange.

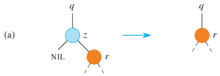
a) Node z has no left child.

b) Node z has a left child ℓ but no right child.

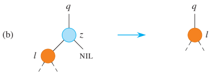
c) Node z has two children. Its left child is node ℓ, its right child is its successor y (which has no left child), and y's right child is node x.

d) Node has two children (left child ℓ and right child r), and its successor y $\neq$ r lies within the subtree rooted at r.

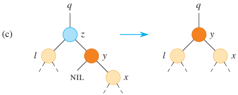
a)



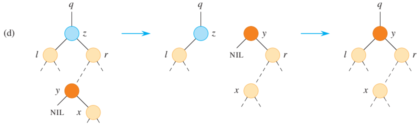
b)



c)



d)

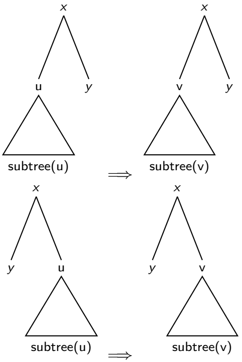


Transplant

Function Transplant(T, u, v) [[2]]

```
if u.p == NIL then
    T.root = v
else if u == u.p.left then
    u.p.left = v
else
    u.p.right = v
if v ≠ NIL then
    v.p = u.p
```

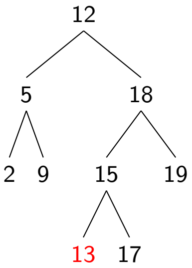
TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.



Tree Delete

Function TreeDelete(T, z) [[2]]

```
if z.left == NIL then
    Transplant (T, z, z.right)        // replace z by its right child
else if z.right == NIL then
    Transplant (T, z, z.left)         // replace z by its left child
else
    y = TreeMinimum (z.right)         // y is z's successor
    if y ≠ z.right then               // is farther down the tree?
        Transplant (T, y, y.right)    // replace y by its right child
        y.right = z.right             // z's right child becomes
        y.right.parent = y            // y's right child
    Transplant (T, z, y)              // replace z by its successor y
    y.left = z.left                   // and give z's left child to y,
    y.left.parent = y                 // which had no left child
```



Trace of TreeDelete (T, 13)

Implementation

Node for Binary Search Tree

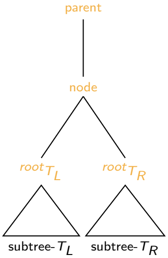
Algorithm 2: Node for Binary Search Tree

```
begin
    data
    parent                  // reference to the parent
    left                    // reference to the left child
    right                   // reference to the right child
```

[[2]]

In binary search tree, each node has 0, 1 or 2 subtrees.
Use node with three pointers:

- `parent`: points the parent
- `left`: points the left child
- `right`: points the right child



Binary Search Tree property:

- Every node in subtree T_L is smaller than `node`
- Every node in subtree T_R is larger than `node`

Q. Is `node.data` smaller than `parent.data`?
Q. Is `node.data` larger than `parent.data`?

Assume that data at each node is different.

MyNodeBST

Algorithm 3: Node for Binary Search Tree

```
begin
    data
    parent          // reference to the parent
    left            // reference to the left child
    right           // reference to the right child
```

[[2]]

`MyNodeBST` implements interface `NodeBSTInterface`.

- `T` implements `Comparable`

$$\texttt{y.data.compareTo(x.data)} = \begin{cases} < 0, & \texttt{y.data < x.data}, \\ 0, & \texttt{y.data = x.data}, \\ > 0, & \texttt{y.data > x.data}. \end{cases}$$

```java
MyNodeBST<T> x = ...;
MyNodeBST<T> y = ...;

// compare x and y
if (y.data.compareTo(x.data) < 0) {
  // y is smaller than x
} else {
  // y is larger than x
}
```

```java
public class MyNodeBST<T extends Comparable<T>>
    implements MyNodeBSTInterface<T> {
  T data;
  MyNodeBST<T> parent;
  MyNodeBST<T> left;
  MyNodeBST<T> right;

  public MyNodeBST() {
    this(null);
  }
  public MyNodeBST(T data) {
    this.data = data;
    parent = null;
    left = null;
    right = null;
  }
  ...
}
```

```java
public interface NodeBSTInterface<T> {

  T data();

  MyNodeBSTInterface<T> parent();

  MyNodeBSTInterface<T> left();

  MyNodeBSTInterface<T> right();

  T canonical();

}
```

MyBST, constructor

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;

  public MyBST() {
    root = null;
  }
  public void insert(T data) {
    if (data == null) {
      return;
    }
    MyNodeBST<T> x = root;
    MyNodeBST<T> y = null;
    MyNodeBST<T> z = new MyNodeBST<>(data);
    while (x != null) {
      y = x;
      if (z.data.compareTo(x.data) < 0) {
        x = x.left;
      } else {
        x = x.right;
      }
    }
    // location is found.
    z.parent = y;
    if (y == null) {
      root = z;
    } else if (z.data.compareTo(y.data) < 0) {
      y.left = z;
    } else {
      y.right = z;
    }
  }
  ...
}
```

```java
public class MyNodeBST<T extends Comparable<T>>
    implements MyNodeBSTInterface<T> {
  T data;
  MyNodeBST<T> parent;
  MyNodeBST<T> left;
  MyNodeBST<T> right;

  public MyNodeBST() {
    this(null);
  }
  public MyNodeBST(T data) {
    this.data = data;
    parent = null;
    left = null;
    right = null;
  }
  ...
}
```

```java
public class MyBSTConstructor {

  public static MyBST<String>
      constructBST_S_412_nodeByNode() {
    MyBST<String> tree = new MyBST<>();
    tree.insert("4");
    tree.insert("1");
    tree.insert("2");
    return tree;
  }
  ...
}
```

MyBST, `insert`

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;

  public MyBST() {
    root = null;
  }
  public void insert(T data) {
    if (data == null) {
      return;
    }
    MyNodeBST<T> x = root;
    MyNodeBST<T> y = null;
    MyNodeBST<T> z = new MyNodeBST<>(data);
    while (x != null) {
      y = x;
      if (z.data.compareTo(x.data) < 0) {
        x = x.left;
      } else {
        x = x.right;
      }
    }
    // location is found.
    z.parent = y;
    if (y == null) {
      root = z;
    } else if (z.data.compareTo(y.data) < 0) {
      y.left = z;
    } else {
      y.right = z;
    }
  }
  ...
}
```

```java
public class MyBSTConstructor_Test {

  private static final boolean DEBUG = true;

  public static void main(String[] args) {
    MyBST<String> tree;

    tree = MyBSTConstructor.
      constructBST_S_412_nodeByNode();
    //
    if (DEBUG) {
      tree.plot();
      System.out.println(tree.canonical());
    }
  }

}
```

```java
public class MyBSTConstructor {

  public static MyBST<String>
      constructBST_S_412_nodeByNode() {
    MyBST<String> tree = new MyBST<>();
    tree.insert("4");
    tree.insert("1");
    tree.insert("2");
    return tree;
  }
  ...
}
```

```
>4
  /2
 \1

/[1]/[2]\[4]\
```

MyBST, insert

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;

  public MyBST() {
    root = null;
  }
  public void insert(T data) {
    if (data == null) {
      return;
    }
    MyNodeBST<T> x = root;
    MyNodeBST<T> y = null;
    MyNodeBST<T> z = new MyNodeBST<>(data);
    while (x != null) {
      y = x;
      if (z.data.compareTo(x.data) < 0) {
        x = x.left;
      } else {
        x = x.right;
      }
    }
    // location is found.
    z.parent = y;
    if (y == null) {
      root = z;
    } else if (z.data.compareTo(y.data) < 0) {
      y.left = z;
    } else {
      y.right = z;
    }
  }
  ...
}
```

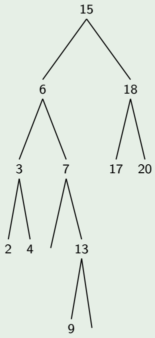
```java
public class MyBSTConstructor_Test {

  private static final boolean DEBUG = true;

  public static void main(String[] args) {
    MyBST<String> tree;

    tree = MyBSTConstructor.
      constructBST_S_412_nodeByNode();
    //
    if (DEBUG) {
      tree.plot();
      System.out.println(tree.canonical());
    }
  }

}
```

```java
public class MyBSTConstructor {

  public static MyBST<String>
      constructBST_S_412_nodeByNode() {
    MyBST<String> tree = new MyBST<>();
    tree.insert("4");
    tree.insert("1");
    tree.insert("2");
    return tree;
  }
  ...
}
```

MyBST, `search`

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> successor() {
    return (MyNodeBST<T>) LibTreeBS.successor(
       root);
  }

  public MyNodeBST<T> successor(MyNodeBST<T> x)
       {
    return (MyNodeBST<T>) LibTreeBS.successor(x)
       ;
  }
  ...
}
```

Example



`tree.search(13)` ?

`tree.search(4)` ?

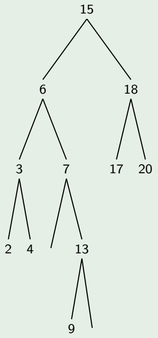
`tree.search(5)` ?

MyBST, `search`

```
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> successor() {
    return (MyNodeBST<T>) LibTreeBS.successor(root);
  }

  public MyNodeBST<T> successor(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.successor(x);
  }
  ...
}
```

```
search(13): 15 -> 6 -> 7 -> 13
search(4): 15 -> 6 -> 3 -> 4
search(5): 15 -> 6 -> 3 -> 4 -> not found
```

Example



`tree.search(13)` ?

`tree.search(4)` ?

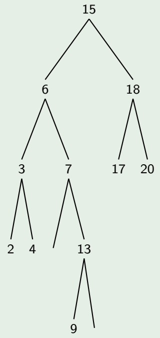
`tree.search(5)` ?

MyBST, `minimum`

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...

  public MyNodeBST<T> minimum() {
    return (MyNodeBST<T>) LibTreeBS.minimum(root);
  }

  public MyNodeBST<T> minimum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.minimum(x);
  }
  ...
}
```

```java
public class LibTreeBS<T extends Comparable<T>> {
  ...
  public static <T> NodeBSTInterface<T> minimum(
      NodeBSTInterface<T> x
    ) {
    while (x.left() != null) {
      x = x.left();
    }
    return x;
  }
  ...
}
```

Example



`tree.minimum()` ?

`tree.minimum(tree.root.right)` ?

`tree.minimum(tree.root.right.right)` ?

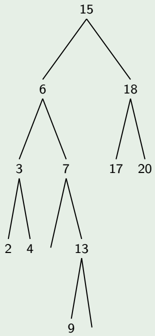
MyBST, `minimum`

```
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...

  public MyNodeBST<T> minimum() {
    return (MyNodeBST<T>) LibTreeBS.minimum(root);
  }

  public MyNodeBST<T> minimum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.minimum(x);
  }
  ...
}
```

```
public class LibTreeBS<T extends Comparable<T>> {
  ...
  public static <T> NodeBSTInterface<T> minimum(
      NodeBSTInterface<T> x
    ) {
    while (x.left() != null) {
      x = x.left();
    }
    return x;
  }
  ...
}
```

```
minimum(15): 2
minimum(18): 17
minimum(20): 20
```

Example



`tree.minimum()` ?

`tree.minimum(tree.root.right)` ?

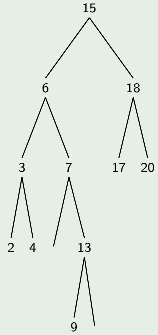
`tree.minimum(tree.root.right.right)` ?

MyBST, `maximum`

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> maximum() {
    return (MyNodeBST<T>) LibTreeBS.maximum(root);
  }

  public MyNodeBST<T> maximum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.maximum(x);
  }
  ...
}
```

```java
public class LibTreeBS<T extends Comparable<T>> {
  ...
  public static <T> NodeBSTInterface<T> maximum(
      NodeBSTInterface<T> x
    ) {
    while (x.right() != null) {
      x = x.right();
    }
    return x;
  }
  ...
}
```

Example



`tree.maximum()` ?

`tree.maximum(tree.root.left)` ?

`tree.maximum(tree.root.left.left)` ?

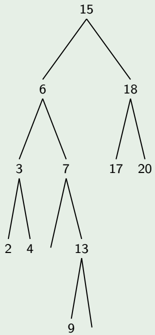
MyBST, `maximum`

```
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> maximum() {
    return (MyNodeBST<T>) LibTreeBS.maximum(root
       );
  }

  public MyNodeBST<T> maximum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.maximum(x);
  }
  ...
}
```

```
public class LibTreeBS<T extends Comparable<T>>
     {
  ...
  public static <T> NodeBSTInterface<T> maximum(
       NodeBSTInterface<T> x
    ) {
    while (x.right() != null) {
      x = x.right();
    }
    return x;
  }
  ...
}
```

```
maximum(15):  20
maximum(6):  13
maximum(3):  4
```

Example



`tree.maximum()` ?

`tree.maximum(tree.root.left)` ?

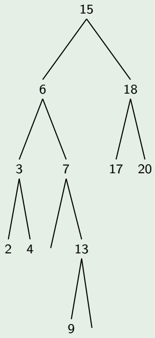
`tree.maximum(tree.root.left.left)` ?

MyBST, `successor`

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> maximum() {
    return (MyNodeBST<T>) LibTreeBS.maximum(root);
  }

  public MyNodeBST<T> maximum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.maximum(x);
  }
  ...
}
```

```java
public class LibTreeBS<T extends Comparable<T>> {
  public static <T> NodeBSTInterface<T> successor(NodeBSTInterface<T> x) {
    if (x.right() != null) {
      // leftmost node in the right subtree
      return minimum(x.right());
    } else {
      // find the lowest ancestor of x whose left child is an ancestor of x
      NodeBSTInterface<T> y = null;
      y = x.parent();
      while (y != null && x == y.right()) {
        x = y;
        y = y.parent();
      }
      return y;
    }
  }
}
```

Example



`tree.successor()` ?

`tree.successor(tree.root.left)` ?

`tree.successor(tree.root.left.right.right)` ?

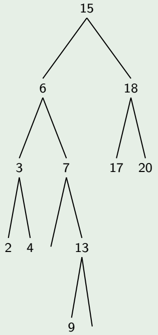
MyBST, `successor`

```
successor(15): 17
successor(6): 7
successor(13): 15
```

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public MyNodeBST<T> maximum() {
    return (MyNodeBST<T>) LibTreeBS.maximum(root);
  }

  public MyNodeBST<T> maximum(MyNodeBST<T> x) {
    return (MyNodeBST<T>) LibTreeBS.maximum(x);
  }
  ...
}
```

```java
public class LibTreeBS<T extends Comparable<T>> {
  public static <T> NodeBSTInterface<T> successor(NodeBSTInterface<T> x) {
    if (x.right() != null) {
      // leftmost node in the right subtree
      return minimum(x.right());
    } else {
      // find the lowest ancestor of x whose left child is an ancestor of x
      NodeBSTInterface<T> y = null;
      y = x.parent();
      while (y != null && x == y.right()) {
        x = y;
        y = y.parent();
      }
      return y;
    }
  }
}
```

Example



`tree.successor()` ?

`tree.successor(tree.root.left)` ?

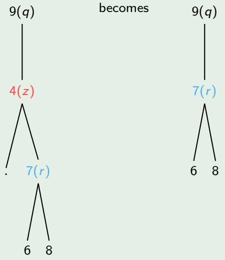
`tree.successor(tree.root.left.right.right)` ?

MyBST, `delete` (i)

```
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public void delete(MyNodeBST<T> z) {
    MyNodeBST<T> y = null;
    if (z.left == null) {
      // replace z by its right child
      transplant(z, z.right);
    } else if (z.right == null) {
      // replace z by its left child
      transplant(z, z.left);
    } else {
      // y is z's successor
      y = minimum(z.right);
      if (y != z.right) {
        // y is farther down in the tree
        // replace y by its right child
        transplant(y, y.right);
        // z's right child becomes y's right child
        y.right = z.right;
        y.right.parent = y;
      }
      // replace z by its successor y
      transplant(z, y);
      // z's right child, which has no let child, becomes y's right child
      y.left = z.left;
      y.left.parent = y;
    }
  }
}
```

Example

Node `z` has no left child.
We have `z` < `r`.

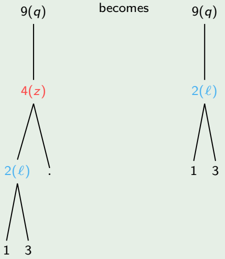


MyBST, `delete` (ii)

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public void delete(MyNodeBST<T> z) {
    MyNodeBST<T> y = null;
    if (z.left == null) {
      // replace z by its right child
      transplant(z, z.right);
    } else if (z.right == null) {
      // replace z by its left child
      transplant(z, z.left);
    } else {
      // y is z's successor
      y = minimum(z.right);
      if (y != z.right) {
        // y is farther down in the tree
        // replace y by its right child
        transplant(y, y.right);
        // z's right child becomes y's right child
        y.right = z.right;
        y.right.parent = y;
      }
      // replace z by its successor y
      transplant(z, y);
      // z's right child, which has no let child, becomes y's right child
      y.left = z.left;
      y.left.parent = y;
    }
  }
}
```

Example

Node z has a left child ℓ but no right child.
We have $\ell < z$.

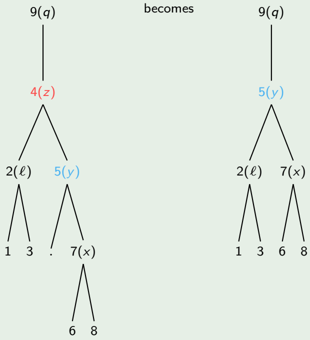


MyBST, `delete` (iii)

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public void delete(MyNodeBST<T> z) {
    MyNodeBST<T> y = null;
    if (z.left == null) {
      // replace z by its right child
      transplant(z, z.right);
    } else if (z.right == null) {
      // replace z by its left child
      transplant(z, z.left);
    } else {
      // y is z's successor
      y = minimum(z.right);
      if (y != z.right) {
        // y is farther down in the tree
        // replace y by its right child
        transplant(y, y.right);
        // z's right child becomes y's right child
        y.right = z.right;
        y.right.parent = y;
      }
      // replace z by its successor y
      transplant(z, y);
      // z's right child, which has no let child, becomes y's right child
      y.left = z.left;
      y.left.parent = y;
    }
  }
}
```

Example

Node `z` has two children (ℓ and `y`). Node `y` is the successor of `z`, that is, `y` has no left child. Node `x` is the right child of `y`. Hence, `y = successor(z)` and ℓ < `z` < `y` < `x`.

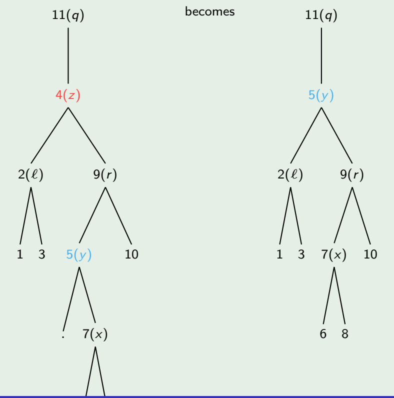


MyBST, `delete` (iv)

```java
public class MyBST<T extends Comparable<T>> {
  MyNodeBST<T> root;
  ...
  public void delete(MyNodeBST<T> z) {
    MyNodeBST<T> y = null;
    if (z.left == null) {
      // replace z by its right child
      transplant(z, z.right);
    } else if (z.right == null) {
      // replace z by its left child
      transplant(z, z.left);
    } else {
      // y is z's successor
      y = minimum(z.right);
      if (y != z.right) {
        // y is farther down in the tree
        // replace y by its right child
        transplant(y, y.right);
        // z's right child becomes y's right child
        y.right = z.right;
        y.right.parent = y;
      }
      // replace z by its successor y
      transplant(z, y);
      // z's right child, which has no let child, becomes y's right child
      y.left = z.left;
      y.left.parent = y;
    }
  }
}
```

Example

Node `z` has two children (ℓ and `r`) and its successor `y` $\neq$ `r` lies within the subtree rooted at `r`.
Hence, `y = successor(z)` and `y` $\neq$ `r`.



Applications

Some Applications of BST

Implement a set in math as a BST.

- Modify `insert` so that no duplication is allowed.
- If the BST is balanced, it would be fast to search, to insert and to delete in $O(\log n)$.

Database Management Systems (DBMS)

- DBMS keeps data in a system similar to BSTs.

References I

[1] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms, volume 1*, 2nd ed. Addison-Wesley Professional, 1973.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (CLRS)*, 4th ed. MIT Press, 2022.

[3] B. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 1999.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[5] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Pitman, 1982.

[6] C. S. Horstmann, *Big Java: Early Objects*, 7th ed. John Wiley & Sons, 2019.

[7] R. P. Grimaldi, *Discrete and Combinatorial Mathematics*, 5th ed. Pearson Education India, 2006.