ITS202: Algorithms and Data Structures

Advanced Data Structures

Ms. Sonam Wangmo

Gyalpozhing College of Information Technology
Royal University of Bhutan

November 13, 2020

Search Trees

1. Binary Search trees
2. Balanaced Binary Trees
    1. Red Black Trees
    2. AVL trees
    3. B trees

Binary Search Trees

1. Binary Search Trees
2. Ordered Operations
3. Deletion

Binary Search Trees

Definition

Binary Search trees is a binary tree T with each position p storing a key-value pair(k,v) such that :

- Keys stored in the left subtree of p are less than k.
- Keys stored in the right subtree of p are greater than k.

Binary Search Trees

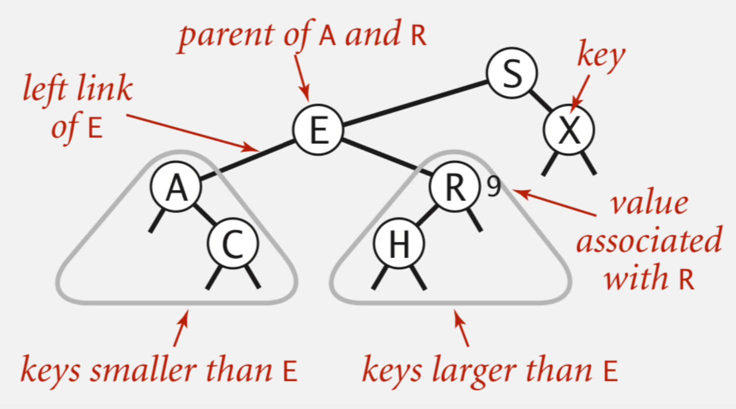

Figure 1: Binary Tree example

Binary Search Trees

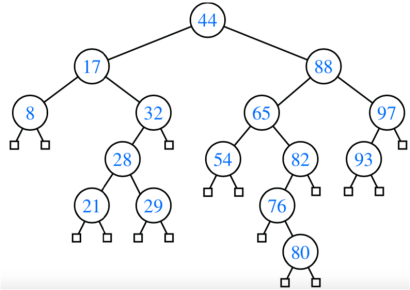

Figure 2: Binary Search Tree

Note

As a matter of convenience, we will not diagram the values associated with keys, since those values do not affect the placement of items within a search tree.

BST representation in Java

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

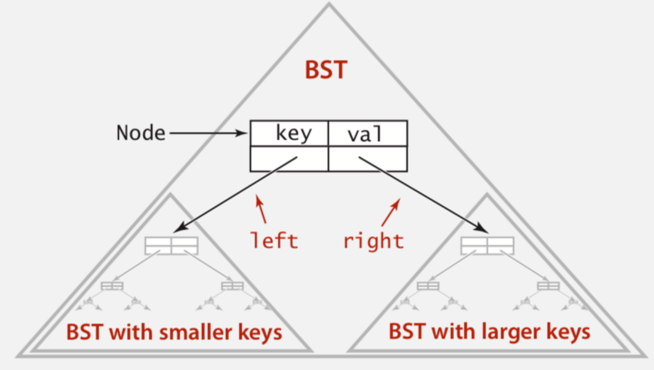

Figure 3: BST representation

BST representation in Java

Node count N

The instance variable N gives the node count in the subtree rooted at the node.

$\text{size}(x) = \text{size}(x.\text{left}) + \text{size}(x.\text{right}) + 1$

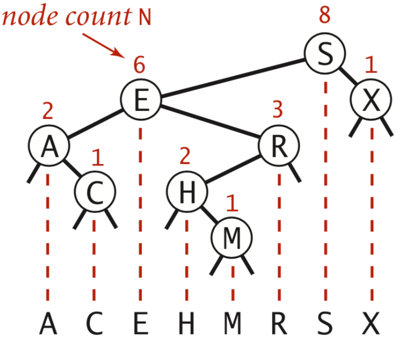

Figure 4: BST

BST representation in Java

Node class in BST

```
private class Node
    {
            private Key key;
            private Value val;
            private Node left, right;
            public Node(Key key, Value val)
            {
                this.key = key;
                this.val = val;
            }
    }
```

Key and Value are generic types; Key is Comparable

BST Implementation(Skeleton)

```
public class BST<Key extends Comparable<Key
    >, Value>
{
    private Node root;
    private class Node
   {  /* see previous slide */   }
    public void put(Key key, Value val)
     {  /* see next slides */   }
    public Value get(Key key)
     {  /* see next slides */   }
    public void delete(Key key)
     {  /* see next slides */   }
    public Iterable<Key> iterator()
     {  /* see next slides */   }
}
```

Binary Search Tree Demo

Note

If a node containing the key is in the table, we have a search hit, so we return the associated value. Otherwise, we have a search miss (and return null).

Search: If less, go left; if greater, go right; if equal, search hit.

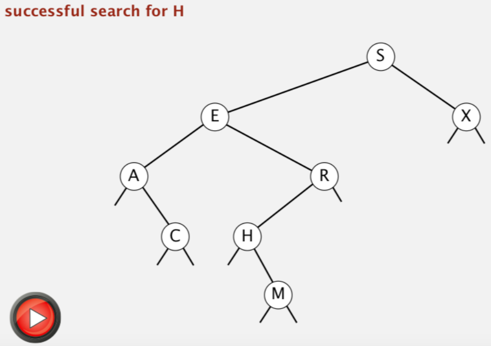

Figure 5: BST Search demo

Binary Search Tree Demo

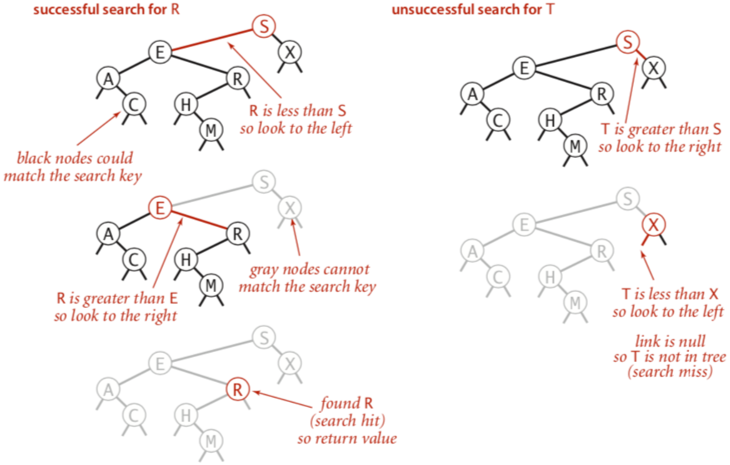

Figure 6: Search hit (left) and search miss (right) in a BST

Binary Search Tree Demo

Insert: If less, go left; if greater, go right; if null, insert.

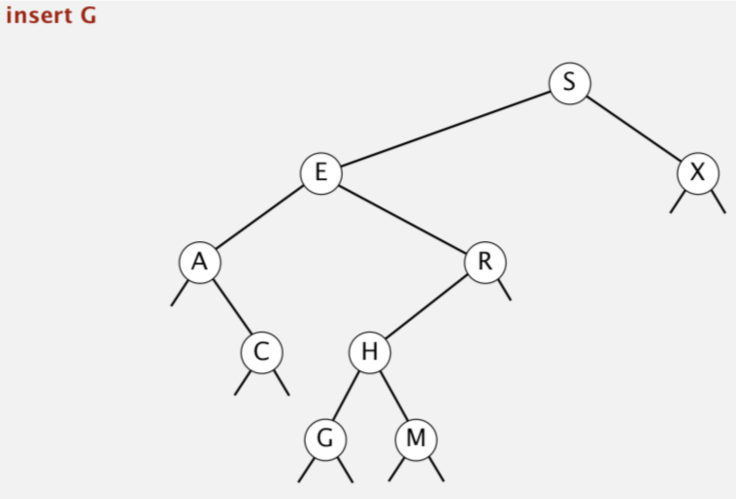

Figure 7: BST Insert demo

BST Search: Java Implementation

Get: Return value corresponding to given key, or null if no such key.

```
public Value get(Key key)
 {
     Node x = root;
     while (x != null)
     {
         int cmp = key.compareTo(x.key);
         if      (cmp  < 0)  x = x.left;
         else if (cmp  > 0)  x = x.right;
         else return x.val;
     }
     return null;
 }
```

Cost: Number of compares is equal to 1 + depth of node.

BST Insert

Put: Associate value with key.

Search for key, then two cases:

Key in tree : reset value.
Key not in tree : add new node.

BST Insert

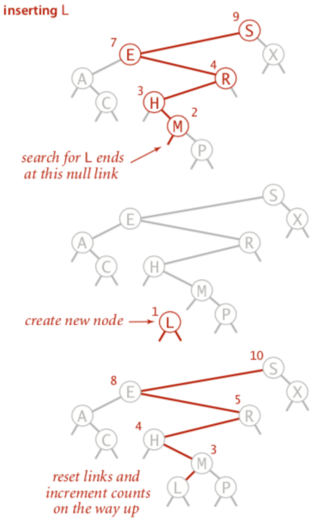

Figure 8: Insertion into a BST

BST Insert

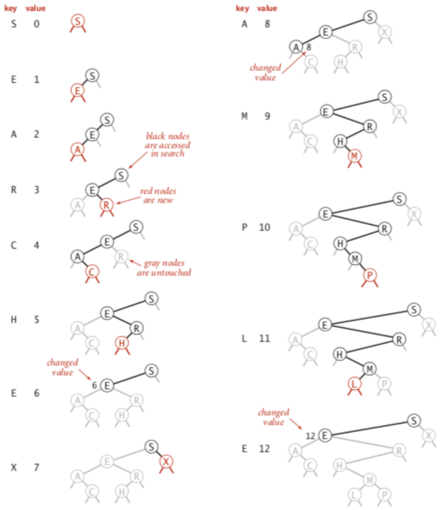

Figure 9: BST trace for standard indexing client

BST Insert: Implementation

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0)
      x.left  = put(x.left,  key, val);
   else if (cmp  > 0)
      x.right = put(x.right, key, val);
   else if (cmp == 0)
      x.val = val;
   return x;
}
```

concise, but tricky, recursive code; read carefully!

Figure 10

BST Running Time Analysis

Search and Insert Operation in BST

Best Case: ω(log n) or $\omega(1)$
Worst Case: O(n)

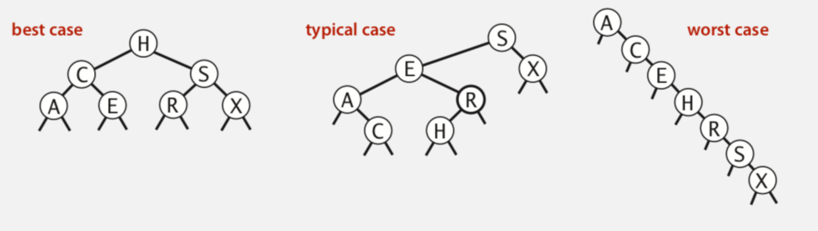

Figure 11: Tree Shape

The running times of algorithms on binary search trees depend on the shapes of the trees, which, in turn, depend on the order in which keys are inserted.

BST Ordered Operations

Minimum

If the left link of the root is null, the smallest key in a BST is the key at the root; if the left link is not null, the smallest key in the BST is the smallest key in the subtree rooted at the node referenced by the left link.

Maximum

Finding the maximum key is similar, moving to the right instead of to the left.

BST Ordered Operations

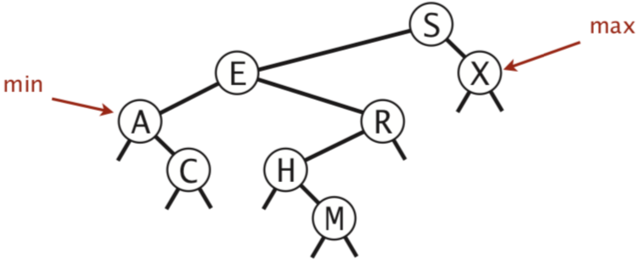

Figure 12

```java
public Key min()
 {
    return min(root).key;
 }
 private Node min(Node x)
 {
    if (x.left == null) return x;
    return min(x.left);
 }
```

Figure 13: Code for Min()

BST Ordered Operations: Floor and ceiling

Floor and ceiling

Floor: Largest key <= a given key.
Ceiling: Smallest key >= a given key.

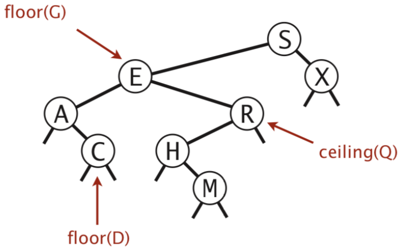

Figure 14

Q. How to find the floor / ceiling?

BST Ordered Operations: Computing the floor

Case 1. [k equals the key in the node]

The floor of k is k.

Case 2. [k is less than the key in the node]

The floor of k is in the left subtree.

Case 3. [k is greater than the key in the node]

The floor of k is in the right subtree

(if there is any key $\leq k$ in right subtree);

otherwise it is the key in the node.

BST Ordered Operations: Computing the floor

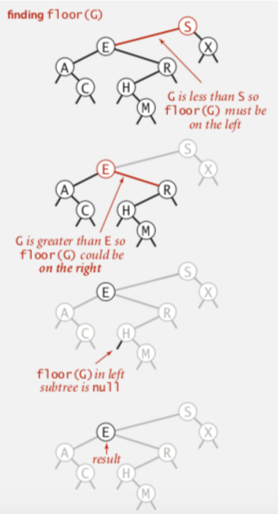

Figure 15: Computing the floor function

BST Ordered Operations: Computing the floor

Example.java

```java
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0)  return floor(x.left, key);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}
```

Line 15, Column 4 Tab Size: 4 Java

Figure 16: Code for floor function

BST Ordered Operations: Rank and Select

Q. How to implement rank() and select() efficiently?

A. In each node, we store the number of nodes in the subtree rooted at that node; to implement size(), return the count at the root.

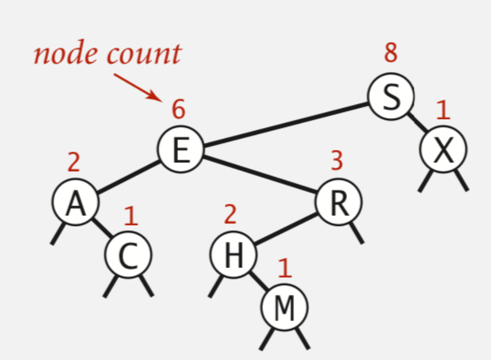

Figure 17: BST

BST Ordered implementation: subtree counts

```java
private class Node
{
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int count;//<----------See this
}
```

Figure 18: Node class

```java
public int size()
{  return size(root);  }

private int size(Node x)
{
    if (x == null) return 0;
    return x.count;
}
```

Figure 19: Size Method

BST Ordered implementation: subtree counts

```java
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);//initialize subtree count
    int cmp = key.compareTo(x.key);
    if      (cmp  < 0) x.left  = put(x.left,  key, val);
    else if (cmp  > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val = val;
    x.count = 1 + size(x.left) + size(x.right);
    return x;

 }
```

Figure 20: Put Method

BST Ordered implementation: Rank

Rank. How many keys < k ?

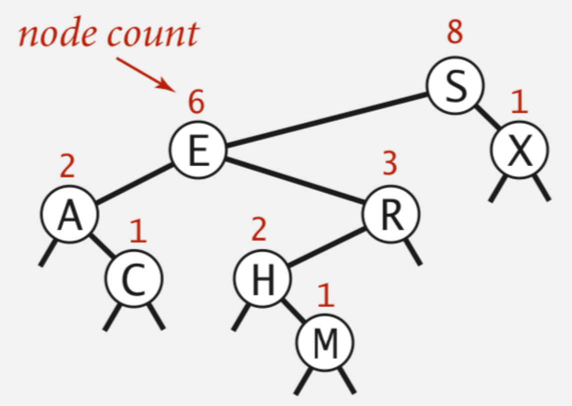

Figure 21: BST

BST Ordered implementation: Rank

```java
public int rank(Key key)
{  return rank(key, root);  }
private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if      (cmp  < 0) return rank(key, x.left);
    else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right)
    else if (cmp == 0) return size(x.left);
}
```

Figure 22: rank method