Binary Search Trees

Outline

What is a Binary Search Tree (BST)?

What is a Binary Search Tree (BST)?

A binary tree is either empty or a node with a key (and associated value) and links (left and right) to two disjoint binary subtrees

What is a Binary Search Tree (BST)?

A binary tree is either empty or a node with a key (and associated value) and links (left and right) to two disjoint binary subtrees

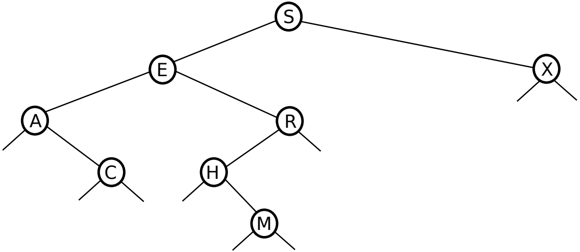
A binary tree is in symmetric order if each node's key is larger than all keys in its left subtree and smaller than all keys in its right subtree

What is a Binary Search Tree (BST)?

A binary tree is either empty or a node with a key (and associated value) and links (left and right) to two disjoint binary subtrees

A binary tree is in symmetric order if each node's key is larger than all keys in its left subtree and smaller than all keys in its right subtree

A binary search tree (BST) is a binary tree in symmetric order



What is a Binary Search Tree (BST)?

What is a Binary Search Tree (BST)?

A BST representation in Java is a reference to a root node, which is composed of five fields: a key, a value, a reference to the left subtree, a reference to the right subtree, and the number of nodes in the subtree

What is a Binary Search Tree (BST)?

A BST representation in Java is a reference to a root node, which is composed of five fields: a key, a value, a reference to the left subtree, a reference to the right subtree, and the number of nodes in the subtree

```java
private class Node {
    private Key key;
    private Value val;
    private int size;
    private Node left, right;

    public Node(Key key, Value value) {
        this.key = key;
        this.val = value;
        this.size = 1;
    }
}
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

BinarySearchTreeST.java

```java
package dsa;

import java.util.NoSuchElementException;
import stdlib.StdIn;
import stdlib.StdOut;

public class BinarySearchTreeST<Key extends Comparable<Key>, Value>
        implements OrderedST<Key, Value> {
    private Node root;

    public BinarySearchTreeST() {
        root = null;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

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

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
            throw new IllegalArgumentException("value is null");
        }
        root = put(root, key, value);
    }

    public Value get(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

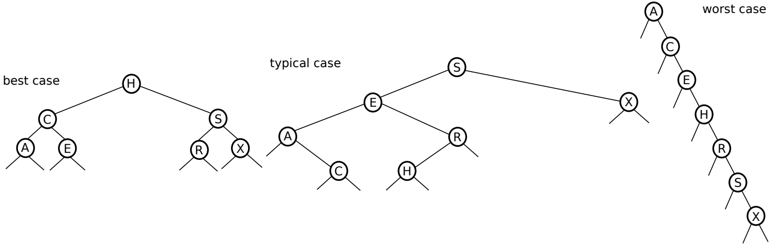
BinarySearchTreeST.java

```java
    public boolean contains(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        return get(key) != null;
    }

    private Node put(Node x, Key key, Value value) {
        if (x == null) {
            return new Node(key, value);
        }
        int cmp = key.compareTo(x.key);
        if (cmp < 0) {
            x.left = put(x.left, key, value);
        } else if (cmp > 0) {
            x.right = put(x.right, key, value);
        } else {
            x.val = value;
        }
        x.size = size(x.left) + size(x.right) + 1;
        return x;
    }

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

Implementation of the Ordered Symbol Table API Using a BST

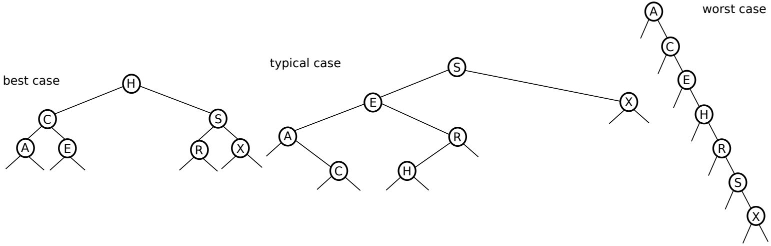
Implementation of the Ordered Symbol Table API Using a BST

Tree shape depends on order of insertion



Implementation of the Ordered Symbol Table API Using a BST

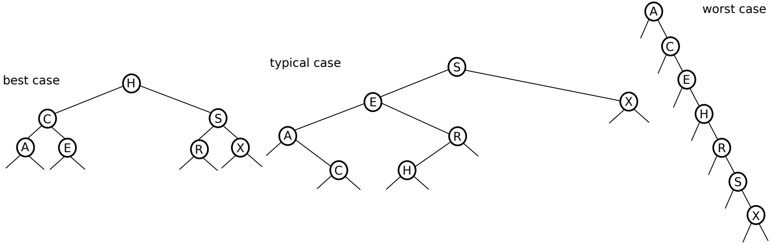
Tree shape depends on order of insertion



Number of comparisons for search/insert is $\sim$ depth of node

Implementation of the Ordered Symbol Table API Using a BST

Tree shape depends on order of insertion



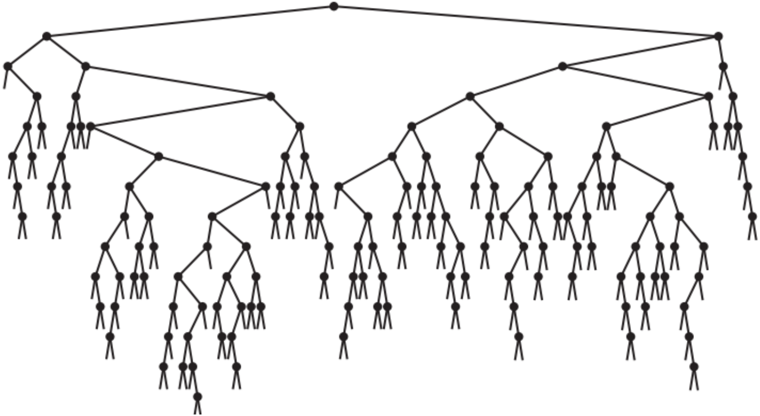
Number of comparisons for search/insert is $\sim$ depth of node

If n distinct keys are inserted into a BST in random order, the expected number of comparisons for a search/insert is $\sim \lg n$

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

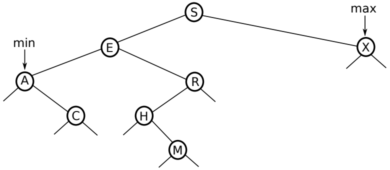
Typical BST, built from 256 random keys



Implementation of the Ordered Symbol Table API Using a BST

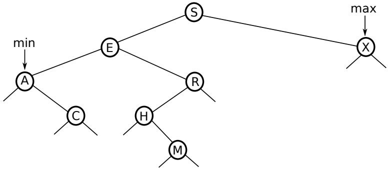
Implementation of the Ordered Symbol Table API Using a BST

Minimum and maximum



Implementation of the Ordered Symbol Table API Using a BST

Minimum and maximum



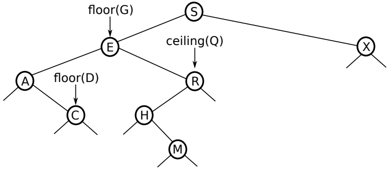
BinarySearchTreeST.java

```java
public Key min() {
    if (isEmpty()) { return null; }
    return min(root).key;
}

private Node min(Node x) {
    if (x.left == null) { return x; }
    else                { return min(x.left); }
}

public Key max() {
    if (isEmpty()) { return null; }
    return max(root).key;
}

private Node max(Node x) {
    if (x.right == null) { return x; }
    else                 { return max(x.right); }
}
```

Implementation of the Ordered Symbol Table API Using a BST

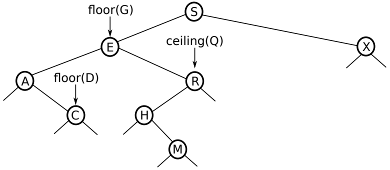
Implementation of the Ordered Symbol Table API Using a BST

Floor and ceiling



Implementation of the Ordered Symbol Table API Using a BST

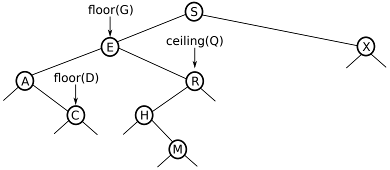
Floor and ceiling



Computing the floor of key k

Implementation of the Ordered Symbol Table API Using a BST

Floor and ceiling

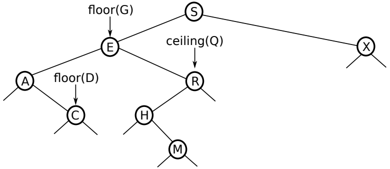


Computing the floor of key k

- Case 1 (k equals the key in the node): the floor of k is k

Implementation of the Ordered Symbol Table API Using a BST

Floor and ceiling

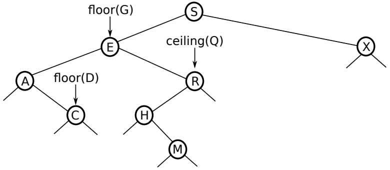


Computing the floor of key k

- 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

Implementation of the Ordered Symbol Table API Using a BST

Floor and ceiling

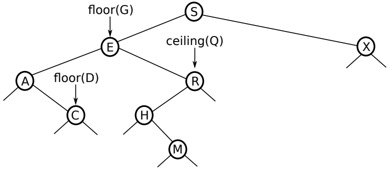


Computing the floor of key k

- 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 there; otherwise, it is the key in the node

Implementation of the Ordered Symbol Table API Using a BST

Floor and ceiling



Computing the floor of key k

- 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 there; otherwise, it is the key in the node

BinarySearchTreeST.java

```java
    public Key floor(Key key) {
        Node x = floor(root, key);
        if (x == null) { return null; }
        else           { 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; }
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

Computing the ceiling of key k

Implementation of the Ordered Symbol Table API Using a BST

Computing the ceiling of key k

- Case 1 (k equals the key in the node): the ceiling of k is k

Computing the ceiling of key k

- Case 1 (k equals the key in the node): the ceiling of k is k
- Case 2 (k is greater than the key in the node): the ceiling of k is in the right subtree

Implementation of the Ordered Symbol Table API Using a BST

Computing the ceiling of key k

- Case 1 (k equals the key in the node): the ceiling of k is k
- Case 2 (k is greater than the key in the node): the ceiling of k is in the right subtree
- Case 3 (k is less than the key in the node): the ceiling of k is in the left subtree if there is any key $\geq k$ in there; otherwise, it is the key in the node

Implementation of the Ordered Symbol Table API Using a BST

Computing the ceiling of key k

- Case 1 (k equals the key in the node): the ceiling of k is k
- Case 2 (k is greater than the key in the node): the ceiling of k is in the right subtree
- Case 3 (k is less than the key in the node): the ceiling of k is in the left subtree if there is any key $\geq k$ in there; otherwise, it is the key in the node

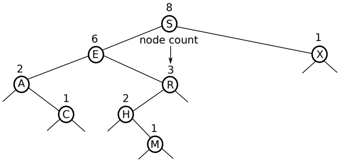
BinarySearchTreeST.java

```java
    public Key ceiling(Key key) {
        Node x = ceiling(root, key);
        if (x == null) { return null; }
        else           { return x.key; }
    }

    private Node ceiling(Node x, Key key) {
        if (x == null) { return null; }
        int cmp = key.compareTo(x.key);
        if (cmp == 0) { return x; }
        if (cmp > 0)  { return ceiling(x.right, key); }
        Node t = ceiling(x.left, key);
        if (t != null) { return t; }
        else           { return x; }
    }
```

Implementation of the Ordered Symbol Table API Using a BST

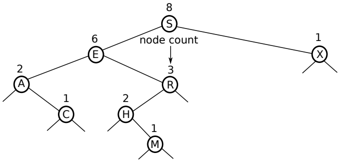
Implementation of the Ordered Symbol Table API Using a BST

Rank and selection



Implementation of the Ordered Symbol Table API Using a BST

Rank and selection



BinarySearchTreeST.java

```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              { return size(x.left); }
}

public Key select(int k) {
    if (k < 0 || k >= size()) { return null; }
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k) {
    if (x == null) { return null; }
    int t = size(x.left);
    if      (t > k) { return select(x.left,  k); }
    else if (t < k) { return select(x.right, k - t - 1); }
    else            { return x; }
}
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

Range count and range search

BinarySearchTreeST.java

```java
public int size(Key lo, Key hi) {
    if (lo.compareTo(hi) > 0) { return 0; }
    if (contains(hi)) { return rank(hi) - rank(lo) + 1; }
    else              { return rank(hi) - rank(lo); }
}

public Iterable<Key> keys() {
    return keys(min(), max());
}

public Iterable<Key> keys(Key lo, Key hi) {
    Queue<Key> queue = new Queue<Key>();
    keys(root, queue, lo, hi);
    return queue;
}

private void keys(Node x, Queue<Key> queue, Key lo, Key hi) {
    if (x == null) { return; }
    int cmplo = lo.compareTo(x.key);
    int cmphi = hi.compareTo(x.key);
    if (cmplo < 0) { keys(x.left, queue, lo, hi); }
    if (cmplo <= 0 && cmphi >= 0) { queue.enqueue(x.key); }
    if (cmphi > 0) { keys(x.right, queue, lo, hi); }
}
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

- Go left (right) until you find a node with null left (right) link

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

- Go left (right) until you find a node with null left (right) link
- Replace that node by its right (left) link

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

- Go left (right) until you find a node with null left (right) link
- Replace that node by its right (left) link
- Update subtree counts

Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete the minimum (maximum) key

- Go left (right) until you find a node with null left (right) link
- Replace that node by its right (left) link
- Update subtree counts

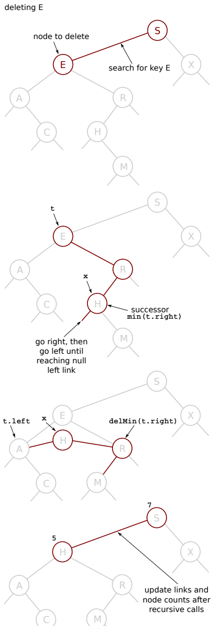
BinarySearchTreeST.java

```java
    public void deleteMin() {
        if (isEmpty()) { throw new NoSuchElementException(); }
        root = deleteMin(root);
    }

    private Node deleteMin(Node x) {
        if (x.left == null) { return x.right; }
        x.left = deleteMin(x.left);
        x.N = size(x.left) + size(x.right) + 1;
        return x;
    }

    public void deleteMax() {
        if (isEmpty()) { throw new NoSuchElementException(); }
        root = deleteMax(root);
    }

    private Node deleteMax(Node x) {
        if (x.right == null) { return x.left; }
        x.right = deleteMax(x.right);
        x.N = size(x.left) + size(x.right) + 1;
        return x;
    }
```

Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

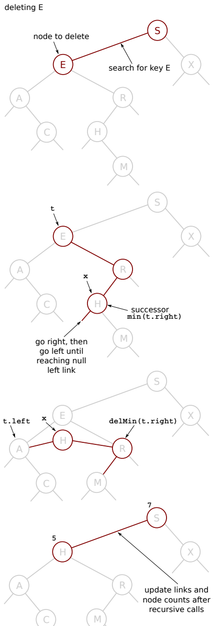
Deletion: to delete a node with key k (Hibbard deletion), search for the node t containing key k



Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete a node with key k (Hibbard deletion), search for the node t containing key k

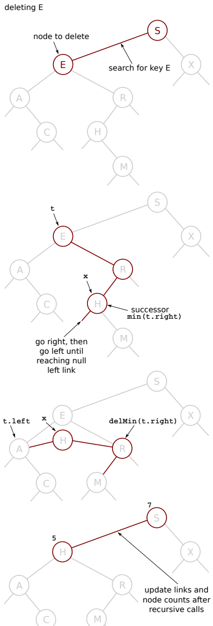
- Case 1 (0 children): delete t by setting parent link to null



Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete a node with key k (Hibbard deletion), search for the node t containing key k

- Case 1 (0 children): delete t by setting parent link to null
- Case 2 (1 child): delete t by replacing parent link

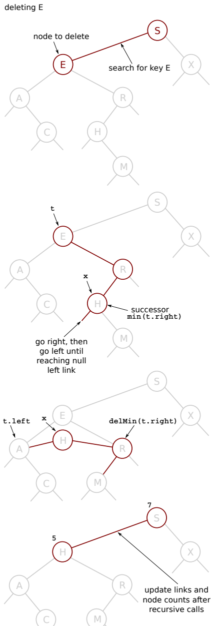


Implementation of the Ordered Symbol Table API Using a BST

Deletion: to delete a node with key k (Hibbard deletion), search for the node t containing key k

- Case 1 (0 children): delete t by setting parent link to null
- Case 2 (1 child): delete t by replacing parent link
- Case 3 (2 children): find successor x of t (x has no left child); delete the minimum in t's right subtree; and put x in t's spot

and update subtree counts



Implementation of the Ordered Symbol Table API Using a BST

Implementation of the Ordered Symbol Table API Using a BST

BinarySearchTreeST.java

```java
    public void delete(Key key) {
        root = delete(root, key);
    }

    private Node delete(Node x, Key key) {
        if (x == null) { return null; }
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) { x.left  = delete(x.left,  key); }
        else if (cmp > 0) { x.right = delete(x.right, key); }
        else {
            if (x.right == null) { return x.left; }
            if (x.left  == null) { return x.right; }
            Node t = x;
            x = min(t.right);
            x.right = deleteMin(t.right);
            x.left = t.left;
        }
        x.N = size(x.left) + size(x.right) + 1;
        return x;
    }
```

Binary Tree Traversal

Binary Tree Traversal

Pre-order traversal

```
public void preorder() { preorder(root); }

private void preorder(Node x) {
    if (x == null) { return; }
    process(x);
    preorder(x.left);
    preorder(x.right);
}
```

Binary Tree Traversal

Pre-order traversal

```
public void preorder() { preorder(root); }

private void preorder(Node x) {
    if (x == null) { return; }
    process(x);
    preorder(x.left);
    preorder(x.right);
}
```

In-order traversal

```
public void inorder() { inorder(root); }

private void inorder(Node x) {
    if (x == null) { return; }
    inorder(x.left);
    process(x);
    inorder(x.right);
}
```

Binary Tree Traversal

Pre-order traversal

```
public void preorder() { preorder(root); }

private void preorder(Node x) {
    if (x == null) { return; }
    process(x);
    preorder(x.left);
    preorder(x.right);
}
```

In-order traversal

```
public void inorder() { inorder(root); }

private void inorder(Node x) {
    if (x == null) { return; }
    inorder(x.left);
    process(x);
    inorder(x.right);
}
```

Post-order traversal

```
public void postorder() { postorder(root); }

private void postorder(Node x) {
    if (x == null) { return; }
    postorder(x.left);
    postorder(x.right);
    process(x);
}
```

Binary Tree Traversal

Pre-order traversal

```
public void preorder() { preorder(root); }

private void preorder(Node x) {
    if (x == null) { return; }
    process(x);
    preorder(x.left);
    preorder(x.right);
}
```

In-order traversal

```
public void inorder() { inorder(root); }

private void inorder(Node x) {
    if (x == null) { return; }
    inorder(x.left);
    process(x);
    inorder(x.right);
}
```

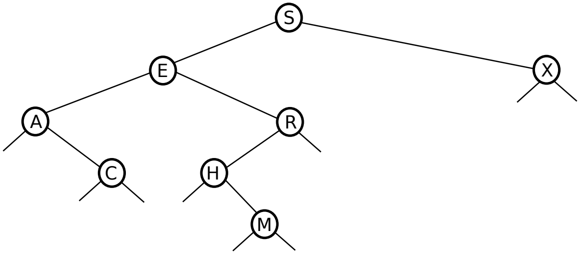
Post-order traversal

```
public void postorder() { postorder(root); }

private void postorder(Node x) {
    if (x == null) { return; }
    postorder(x.left);
    postorder(x.right);
    process(x);
}
```

Binary Tree Traversal

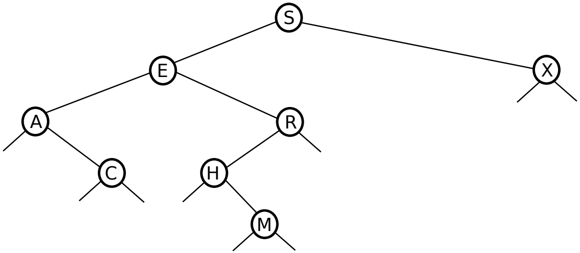
Binary Tree Traversal

For example, let `root` denote the following BST



Binary Tree Traversal

For example, let `root` denote the following BST



The calls `preorder()`, `inorder()`, and `postorder()` will process the tree as follows

`S E A C R H M X` (preorder)

`A C E H M R S X` (inorder)

`C A M H R E X S` (postorder)

Performance Characteristics

Performance Characteristics

Symbol table operations summary

Operation	Unordered Linked List	Ordered Array	BST
search	n	$\lg n$	$h^{\dagger}$
insert	n	n	h
delete	n	n	$\sqrt{n}^{\dagger\dagger}$
min/max	-	1	h
floor/ceiling	-	$\lg n$	h
rank	-	$\lg n$	h
select	-	1	h
ordered iteration	-	n	n

† h is the height of BST, proportional to $\lg n$ if keys inserted in random order

†† other operations also become $\sqrt{n}$ if deletions are allowed