COMP 2140 - Data Structures

Shahin Kamali

Topic 9 - Binary Search Trees

University of Manitoba

Partially based on notes by S. Durocher.
Overview

- binary search trees for dictionaries
- implementing operations on binary search trees

(see Open Data Structures, Chapter 6 for further reading)
Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.
- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.

- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).

- Binary search trees are another method for implementing dictionaries that support additional operations.
  - **minimum()** and **maximum()**: return the smallest/largest item in the dictionary (the items with smallest/largest keys).
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.

- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).

- Binary search trees are another method for implementing dictionaries that support additional operations.
  - minimum() and maximum(): return the smallest/largest item in the dictionary (the items with smallest/largest keys).
  - predecessor(key): return the largest item smaller than key.
Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.

We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).

Binary search trees are another method for implementing dictionaries that support additional operations.

- **minimum()** and **maximum()**: return the smallest/largest item in the dictionary (the items with smallest/largest keys).
- **predecessor(key)**: return the largest item smaller than key
- **successor(key)**: return the smallest item larger than key
Recall that a dictionary is an abstract data type with **search**, **insert**, and **delete** operations for a set of key-value pairs.

We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).

Binary search trees are another method for implementing dictionaries that support additional operations.

- **minimum()** and **maximum()**: return the smallest/largest item in the dictionary (the items with smallest/largest keys).
- **predecessor(key)**: return the largest item smaller than key
- **successor(key)**: return the smallest item larger than key

E.g., \( S = \{(100, v_1), (150, v_2), (56, v_3), (12, v_4)\} \)
- predecessor(100) = ?
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.
- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).
- Binary search trees are another method for implementing dictionaries that support additional operations.
  - minimum() and maximum(): return the smallest/largest item in the dictionary (the items with smallest/largest keys).
  - predecessor(key): return the largest item smaller than key
  - successor(key): return the smallest item larger than key
  - E.g., $S = \{(100, v_1), (150, v_2), (56, v_3), (12, v_4)\}$
    - predecessor(100) = ? 56
    - successor(100) = ?
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.
- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).
- Binary search trees are another method for implementing dictionaries that support additional operations.

- **minimum()** and **maximum()**: return the smallest/largest item in the dictionary (the items with smallest/largest keys).
- **predecessor(key)**: return the largest item smaller than key.
- **successor(key)**: return the smallest item larger than key.
- E.g., \( S = \{(100, v_1), (150, v_2), (56, v_3), (12, v_4)\}\)
  - predecessor(100) = ? 56
  - successor(100) = ? 150
  - predecessor(12) = ?
Motivation

- Recall that a dictionary is an abstract data type with search, insert, and delete operations for a set of key-value pairs.
- We saw how to implement a dictionary using arrays (bad), linked-lists (bad), and hash tables (good).
- Binary search trees are another method for implementing dictionaries that support additional operations.
  - `minimum()` and `maximum()`: return the smallest/largest item in the dictionary (the items with smallest/largest keys).
  - `predecessor(key)`: return the largest item smaller than key.
  - `successor(key)`: return the smallest item larger than key.
  - E.g., \( S = \{(100, v_1), (150, v_2), (56, v_3), (12, v_4)\}\)
    - `predecessor(100) = ?` 56
    - `successor(100) = ?` 150
    - `predecessor(12) = ?` null
    - `successor(12) = 56`
public interface DictionaryADT<E> {
    public E search(int key);
    public void insert(int key, E item);
    public void delete(int key);
    public int minimum();
    public int maximum();
    public int predecessor(int key);
    public int successor(int key);
    public boolean isEmpty();
    public int size();
}
A **binary search tree (BST)** is a binary tree such that for every node \( v \) in the tree:

1. for all nodes \( u \) in the left subtree of \( v \), \( \text{key}(u) < \text{key}(v) \), and
2. for all nodes \( u \) in the right subtree of \( v \), \( \text{key}(u) > \text{key}(v) \).
Implementation of Binary Search Trees

```java
public class TreeNode<E> {
    public int key;
    public E data;
    public TreeNode<E> left;
    public TreeNode<E> right;
    public TreeNode<E> parent;
}

public class BST<E> implements DictionaryADT<E> {
    private TreeNode<E> treeRoot;
    public BST() { treeRoot = null; }
    ...
    // more to add
}
```
Search in a BST

How to search for search key 23 in the following tree?
Search in a BST

Search for a key (called a search key) in a BST:

- If the tree is empty, then the search key is not present in the tree.
Search in a BST

Search for a key (called a search key) in a BST:

- If the tree is empty, then the search key is not present in the tree.
- If the search key matches the root’s key, then the item has been found.
Search in a BST

Search for a key (called a **search key**) in a BST:

- If the tree is empty, then the search key is not present in the tree.
- If the search key matches the root’s key, then the item has been found.
- If the search key is less than to the root’s key, then recursively search in the left subtree of the root.
Search in a BST

- Search for a key (called a search key) in a BST:
  - If the tree is empty, then the search key is not present in the tree.
  - If the search key matches the root’s key, then the item has been found.
  - If the search key is less than to the root’s key, then recursively search in the left subtree of the root.
  - If the search key is greater than the root’s key, then recursively search in the right subtree of the root.

- Unlike the search algorithm for arbitrary binary trees, searching a BST does not require a tree traversal.
Example: $\text{search}(10)$
Example: \textit{search}(10)
Example: \textit{search}(10)
Search Example

Example: search(24)

A search corresponds to a single descent from the root to a node, requiring time proportional to the height of the tree in the worst case.
Search Example

Example: $\textit{search}(24)$
Search Example

Example: \textit{search}(24)
A search corresponds to a single descent from the root to a node, requiring time proportional to the height of the tree in the worst case.
public E search(int searchKey) {
    return searchSubtree(searchKey, treeRoot);
}

private E searchSubtree(int searchKey, TreeNode<E> node) {
    E result = null; // if the tree is null, we should return null
    if (node != null && searchKey == node.key)
        result = node.data; // key is found at root
    else if (node != null && searchKey < node.key)
        result = searchSubtree(searchKey, node.left);
    else if (node != null && searchKey > node.key)
        result = searchSubtree(searchKey, node.right);
    return result;
}
public E search(int searchKey) {
    return searchSubtree(searchKey, treeRoot);
}

private E searchSubtree(int searchKey, TreeNode<E> node) {
    E result = null; // if the tree is null, we should return null
    if (node != null && searchKey == node.key)
        result = node.data; // key is found at root
    else if (node != null && searchKey < node.key)
        result = searchSubtree(searchKey, node.left);
    else if (node != null && searchKey > node.key)
        result = searchSubtree(searchKey, node.right);
    return result;
}

Like any other recursive, search can be implemented sequentially (you will see it in your next assignment).
Insertion in a BST

- **Insert** a new key-value pair into a BST:
  - If the tree is **empty**, then insert the new node as the root.
Insertion in a BST

- **Insert** a new key-value pair into a BST:
  - If the tree is **empty**, then insert the new node as the root.
  - If the new key is **less** than the root’s key, then recursively insert the new node in the **left subtree** of the root.
Insertion in a BST

- **Insert** a new key-value pair into a BST:
  - If the tree is **empty**, then insert the new node as the root.
  - If the new key is **less** than the root’s key, then recursively insert the new node in the **left subtree** of the root.
  - If the new key is **greater** than the root’s key, then recursively insert the new node in the **right subtree** of the root.
Example: $search(24)$
Example: \textit{search}(24)
Example: \textit{search}(24)
Example: \textit{search(24)}
Example: $\text{insert}(24, \ldots)$
Insertion Example

- Insert the following sequence of keys into a BST. Observe that both sets of keys are identical, but the ordering is different.
- Case 1: 7, 4, 3, 6, 9, 8
Insertion Example

• Insert the following sequence of keys into a BST. Observe that both sets of keys are identical, but the ordering is different.

  Case 1: 7, 4, 3, 6, 9, 8

  • The resulting tree is a **complete binary tree** (complete = optimally balanced).
  • Searching in a balanced (or nearly-balanced) binary search tree takes $O(\log n)$ time.
Insertion Example

- Insert the following sequence of keys into a BST. Observe that both sets of keys are identical, but the ordering is different.

Case 1: 7, 4, 3, 6, 9, 8
  - The resulting tree is a **complete binary tree** (complete = optimally balanced).
  - Searching in a balanced (or nearly-balanced) binary search tree takes $O(\log n)$ time.

Case 2: 3, 4, 6, 7, 8, 9
  - The resulting tree is unbalanced.
  - Searching in an **unbalanced binary search tree** takes $O(\log n)$ time in the worst case.
Insertion Example

- Insert the following sequence of keys into a BST. Observe that both sets of keys are identical, but the ordering is different.

- Case 1: 7, 4, 3, 6, 9, 8
  - The resulting tree is a complete binary tree (complete = optimally balanced).
  - Searching in a balanced (or nearly-balanced) binary search tree takes $O(\log n)$ time.

- Case 2: 3, 4, 6, 7, 8, 9
  - The resulting tree is unbalanced.
  - Searching in an unbalanced binary search tree takes $O(\log n)$ time in the worst case.
  - It is desirable to keep the BST balanced after each insertion.
Balancing a BST

- Examples of balanced search trees include AVL trees, red-black trees, and b-trees.
Balancing a BST

- Examples of balanced search trees include AVL trees, red-black trees, and b-trees.
- A common tree-balancing operation is a **tree rotation**.
  - Assume subtree $a$ is too tall compared to $b$ and $c$.
  - We update the pointers of nodes 2 and 4 too ‘shorten $a$’.
  - Note the result is still a BST (node ordering is preserved)
Examples of balanced search trees include AVL trees, red-black trees, and b-trees.

A common tree-balancing operation is a **tree rotation**.
- Assume subtree a is too tall compared to b and c.
- We update the pointers of nodes 2 and 4 too ‘shorten a’.
- Note the result is still a BST (node ordering is preserved)

Each rotation involves updating at most 6 pointers and takes \(O(1)\) time.
Examples of balanced search trees include AVL trees, red-black trees, and b-trees.

A common tree-balancing operation is a **tree rotation**.

- Assume subtree $a$ is too tall compared to $b$ and $c$.
- We update the pointers of nodes 2 and 4 too ‘shorten $a$’.
- Note the result is still a BST (node ordering is preserved)

Each rotation involves updating at most 6 pointers and takes $O(1)$ time

original height: $\max(\text{height}(a) + 2; \text{height}(b) + 2; \text{height}(c) + 1)$

new height: $\max(\text{height}(a) + 1; \text{height}(b) + 2; \text{height}(c) + 2)$
Insert with Balancing

• AVL tree idea: After descending the tree to insert the new node, the descent path is reversed back up to the root. At each node, compare the heights of the subtrees and check whether a tree rotation is required.
Insert with Balancing

- AVL tree idea: After descending the tree to insert the new node, the descent path is reversed back up to the root. At each node, compare the heights of the subtrees and check whether a tree rotation is required.

- In the worst case, rotations are performed back up to the root: this requires $O(\log n)$ time.
Insert with Balancing

- AVL tree idea: After descending the tree to insert the new node, the descent path is reversed back up to the root. At each node, compare the heights of the subtrees and check whether a tree rotation is required.

- In the worst case, rotations are performed back up to the root: this requires $O(\log n)$ time.

- Tree rotations allow insertion and deletion in a BST
  - in $O(\log n)$ worst-case time,
  - while maintaining the BST property, and
  - Ensuring that the tree remains balanced (has height $O(\log n)$).
Finding Minimum/Maximum in a BST

- How do we find the node with the smallest (or largest) key?

![BST Diagram]

- To find minimum: Find the left-most node, i.e., follow the left pointers to find a node without a left child.
- To find maximum: Find the right-most node, i.e., follow the right pointers to find a node without a right child.
Finding Minimum/Maximum in a BST

- How do we find the node with the smallest (or largest) key?

To find minimum:
- Find the left-most node, i.e., follow the left pointers to find a node without left child.

To find maximum:
- Find the right-most node, i.e., follow the right pointers to find a node without right child.
public int minimum () {
    int result = -1;
    TreeNode<E> node = treeRoot;
    while ( node != null && node.left != null )
        node = node.left;
    if ( node != null )
        result = node.key;
    return result;
}
public int minimum() {
    int result = -1;
    TreeNode<E> node = treeRoot;
    while (node != null && node.left != null) {
        node = node.left;
    }
    if (node != null) {
        result = node.key;
    }
    return result;
}

maximum is similar: replace "left" by "right."
Deletion in BSTs

- Deleting a node involves a few cases.
- First we define some helper functions:
  - counting the children of a node
  - swapping the keys of two nodes
  - finding the predecessor or successor of a node
Finding The Number of Children

- Recall that each node has 0, 1, or 2 children.
- We use the following code to find the exact number.

```java
private int numChildren(TreeNode<E> node) {
    int result = -1;
    if (node != null)
        result = 0;
    if (node != null && node.left != null)
        result ++; // has a left child
    if (node != null && node.right != null)
        result ++; // has a right child
    return result;
}
```

The number of children of a node can be reported in constant time \( O(1) \).
Finding The Number of Children

- Recall that each node has 0, 1, or 2 children.
- We use the following code to find the exact number.

```java
private int numChildren(TreeNode<E> node) {
    int result = -1;
    if (node != null)
        result = 0;
    if (node != null && node.left != null)
        result++;// has a left child
    if (node != null && node.right != null)
        result++;// has a right child
    return result;
}
```

- The number of children of a node can be reported in constant time $O(1)$. 
Swapping Two Nodes

- Swap the contents of two given nodes; the structure of the remaining tree remains unchanged.
- Only key/data fields are swapped; children/parent pointers remain unchanged.

```java
private void swap(TreeNode<E> node1, TreeNode<E> node2) {
    if (node1 != null && node2 != null) {
        TreeNode<E> temp = new TreeNode<E>(node1.key, node1.data);
        node1.key = node2.key;
        node1.data = node2.data;
        node2.key = temp.key;
        node2.data = temp.data;
    }
}
```
Swapping Two Nodes

- Swap the contents of two given nodes; the structure of the remaining tree remains unchanged.
- Only key/data fields are swapped; children/parent pointers remain unchanged.

```java
private void swap(TreeNode<E> node1, TreeNode<E> node2) {
    if (node1 != null && node2 != null) {
        TreeNode<E> temp = new TreeNode<E>(node1.key, node1.data);
        node1.key = node2.key;
        node1.data = node2.data;
        node2.key = temp.key;
        node2.data = temp.data;
    }
}
```

- Swapping two nodes takes constant time $O(1)$. 
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40
- `myTree.predecessor(30);`
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

```
myTree.predecessor(30); // 25
myTree.predecessor(9);  // 6
myTree.predecessor(2);  // undefined
myTree.successor(9);    // 22
```

- sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40
  - myTree.predecessor(30); 25
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

- sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40
  - myTree.predecessor(30); 25
  - myTree.predecessor(9);
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

```
myTree.predecessor(30); 25
myTree.successor(9); 22
```

- sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

**sequence of keys in tree:** 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40

- `myTree.predecessor(30);` 25
- `myTree.predecessor(9);` 6
- `myTree.predecessor(2);`
Predecessor/Successor in a BST

Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40

- myTree.predecessor(30); 25
- myTree.predecessor(9); 6
- myTree.predecessor(2); undefined
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

```
30
  9  34
 / \
4   22
  / \
2  6   25
  /   /   \
2   31   39
   / \
 30 40
```

- sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40
  - myTree.predecessor(30); 25
  - myTree.predecessor(9); 6
  - myTree.predecessor(2); undefined
  - myTree.successor(9);
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

Sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40

- myTree.predecessor(30); 25
- myTree.predecessor(9); 6
- myTree.predecessor(2); undefined
- myTree.successor(9); 22
Predecessor/Successor in a BST

- Given a search key, find the preceding/next key in the sequence of keys stored in the BST.

sequence of keys in tree: 2, 4, 6, 9, 22, 25, 30, 31, 34, 39, 40

- myTree.predecessor(30); 25
- myTree.predecessor(9); 6
- myTree.predecessor(2); undefined
- myTree.successor(9); 22
Suppose we want to find the predecessor of a node $x$.

- If $x$ has a left child, then the predecessor must be the maximum node in the left subtree.

- If $x$ has no left child, then the nearest ancestor whose right subtree contains $x$ must be the predecessor.

If no such ancestor exists, then $x$ must be the minimum.
Suppose we want to find the predecessor of a node $x$.

- If $x$ has a left child, then the predecessor must be the maximum node in the left subtree. predecessor(9)?

- Otherwise, the predecessor is the nearest ancestor of $x$ whose right subtree contains $x$. Follow the parent pointer until node = node.parent.right. predecessor(31)?

- If no such ancestor exists, then $x$ must be the minimum. predecessor(2)?
Suppose we want to find the predecessor of a node \( x \).

- If \( x \) has a left child, then the predecessor must be the maximum node in the left subtree. \( \text{predecessor}(9) = 6 \)

- Otherwise, the predecessor is the nearest ancestor of \( x \) whose right subtree contains \( x \).

If no such ancestor exists, then \( x \) must be the minimum. \( \text{predecessor}(2) \) is not shown in the diagram.
Predecessor in a BST

Suppose we want to find the predecessor of a node $x$.

- If $x$ has a left child, then the predecessor must be the maximum node in the left subtree. `predecessor(9)`? 6
- Otherwise, the predecessor is the nearest ancestor of $x$ whose right subtree contains $x$. → follow the parent pointer until `node = node.parent.right` →
Suppose we want to find the predecessor of a node $x$.

- If $x$ has a left child, then the predecessor must be the maximum node in the left subtree. $\text{predecessor}(9)$? 6
- Otherwise, the predecessor is the nearest ancestor of $x$ whose right subtree contains $x$. → follow the parent pointer until $\text{node} = \text{node}.\text{parent}.\text{right}$ → $\text{predecessor}(31)$?
Suppose we want to find the predecessor of a node \( x \).

- If \( x \) has a left child, then the predecessor must be the maximum node in the left subtree. \( \text{predecessor}(9)? 6 \)
- Otherwise, the predecessor is the nearest ancestor of \( x \) whose right subtree contains \( x \). Follow the parent pointer until \( \text{node} = \text{node}.parent.right \to \text{predecessor}(31)? 30 \)
Predecessor in a BST

- Suppose we want to find the predecessor of a node \( x \).

- If \( x \) has a left child, then the predecessor must be the maximum node in the left subtree. \( \text{predecessor}(9) \)? 6

- Otherwise, the predecessor is the nearest ancestor of \( x \) whose right subtree contains \( x \). → follow the parent pointer until \( \text{node} = \text{node}.\text{parent}.\text{right} \) → \( \text{predecessor}(31) \)? 30

- If no such ancestor exists, then \( x \) must be the minimum.
Suppose we want to find the predecessor of a node \( x \).

- If \( x \) has a left child, then the predecessor must be the maximum node in the left subtree. \( \text{predecessor}(9) \)? 6
- Otherwise, the predecessor is the nearest ancestor of \( x \) whose right subtree contains \( x \). → follow the parent pointer until node = node.parent.right → \( \text{predecessor}(31) \)? 30
- If no such ancestor exists, then \( x \) must be the minimum. \( \text{predecessor}(2) \)?
Finding the predecessor of node with a given key

```java
public int predecessor(int key) {
    int result = -1;
    TreeNode<E> node = findNode(key);
    if (node != null && node.left != null)
        result = maximumSubtree(node.left);
    else if (node != null) {
        while (node.parent != null && node == node.parent.left)
            node = node.parent;
        if (node.parent != null)
            result = node.parent.key;
    }
    return result;
}
```
Predecessor/Successor in a BST

Finding the predecessor of node with a given key

```java
public int predecessor(int key) {
    int result = -1;
    TreeNode<E> node = findNode(key);
    if (node != null && node.left != null)
        result = maximumSubtree(node.left);
    else if (node != null) {
        while (node.parent != null && node == node.parent.left)
            node = node.parent;
        if (node.parent != null)
            result = node.parent.key;
    }
    return result;
}
```

successor is similar: replace ".left" by ".right" and maximumSubtree by minimumSubtree.
Finding the predecessor of node with a given key

```java
public int predecessor(int key) {
    int result = -1;
    TreeNode<E> node = findNode(key);
    if (node != null && node.left != null)
        result = maximumSubtree(node.left);
    else if (node != null) {
        while (node.parent != null && node == node.parent.left)
            node = node.parent;
        if (node.parent != null)
            result = node.parent.key;
    }
    return result;
}
```

successor is similar: replace “.left” by “.right” and maximumSubtree by minimumSubtree.

This implementation of predecessor/successor requires that the search key be contained in the BST.
Deletion in BSTs

There are four cases when deleting a node:

1. The node to be deleted is a leaf node.
2. The node to be deleted only has a left child.
3. The node to be deleted only has a right child.
4. The node to be deleted has both a left child and a right child.
Deletion Algorithm

- Find the node $a$ containing the search key $k$. 
Deletion in BSTs

**Deletion Algorithm**

- Find the node \( a \) containing the search key \( k \).
- If \( k \) is not found, then there is no node to delete.
Deletion in BSTs

Deletion Algorithm

- Find the node $a$ containing the search key $k$.
- If $k$ is not found, then there is no node to delete.
- If $a$ is a leaf, delete $a$ (update the parent’s pointer with with null).
Deletion in BSTs

**Deletion Algorithm**

- Find the node $a$ containing the search key $k$.
- If $k$ is not found, then there is no node to delete.
- If $a$ is a leaf, delete $a$ (update the parent’s pointer with null).
- Else if the left subtree of $a$ is empty, replace $a$ with its right child.
- Else swap $a$ with its predecessor (or its successor) and delete $a$. 
Deletion in BSTs

Deletion Algorithm

- Find the node \( a \) containing the search key \( k \).
- If \( k \) is not found, then there is no node to delete.
- If \( a \) is a leaf, delete \( a \) (update the parent's pointer with null).
- Else if the left subtree of \( a \) is empty, replace \( a \) with its right child.
- Else if the right subtree of \( a \) is empty, replace \( a \) with its left child.
Deletion in BSTs

**Deletion Algorithm**

- Find the node \( a \) containing the search key \( k \).
- If \( k \) is not found, then there is no node to delete.
- If \( a \) is a leaf, delete \( a \) (update the parent’s pointer with null).
- Else if the left subtree of \( a \) is empty, replace \( a \) with its right child.
- Else if the right subtree of \( a \) is empty, replace \( a \) with its left child.
- Else swap \( a \) with its predecessor (or its successor) and delete \( a \).
Deleting a node: case 1

- The node has a parent → update parent’s pointer to null
- The node has no parent → the deleted node was the only node in the tree (the tree becomes empty after deletion)
Deletion Case 1

```java
public void delete(int deleteKey) {
    deleteNode(findNode(deleteKey));
}

private void deleteNode(TreeNode<E> deleteNode) {
    TreeNode<E> parent = null;
    if (deleteNode != null)
        parent = deleteNode.parent;
    if (numChildren(deleteNode) == 0) {
        // deleteNode is a leaf
        if (parent == null) // tree has size one
            treeRoot = null;
        else
            if (parent.left == deleteNode)
                parent.left = null
            else
                parent.right = null;
    }
```
public void delete(int deleteKey) {
    deleteNode(findNode(deleteKey));
}

private void deleteNode(TreeNode<E> deleteNode) {
    TreeNode<E> parent = null;
    if (deleteNode != null)
        parent = deleteNode.parent;
    if (numChildren(deleteNode) == 0) {  // deleteNode is a leaf
        if (parent == null)  // tree has size one
            treeRoot = null;
        else
            if (parent.left == deleteNode)
                parent.left = null
            else
                parent.right = null;
    }
}

Deletion takes $O(1)$ in this case (given the pointer to deleteNode).
Deleting a node: case 2

- We need to update the parent’s pointer to point to the single child of the node (similar to deletion in linked-lists)
- The node has no parent (it was the root) → the child’s parent to null (it becomes the new root)
Deletion Case 2

```java
else if (numChildren(deleteNode) == 1 &&
         deleteNode.left != null) {
    deleteNode.left.parent = parent;
    // left child’s parent pointer skips
    the deleted node

    if (parent == null) // deleteNode is
                        // the root
        treeRoot = deleteNode.left;
    else if (parent.left == deleteNode)
        parent.left = deleteNode.left;
    else // parent.right == deleteNode
        parent.right = deleteNode.left;
}
```

Deletion Case 2

```java
else if (numChildren(deleteNode) == 1 && deleteNode.left != null) {
    deleteNode.left.parent = parent;
    // left child’s parent pointer skips the deleted node

    if (parent == null) // deleteNode is the root
        treeRoot = deleteNode.left;
    else if (parent.left == deleteNode)
        parent.left = deleteNode.left;
    else // parent.right == deleteNode
        parent.right = deleteNode.left;
}
```

when deleteNode.right != null is analogous.
Deletion Case 2

```java
else if (numChildren(deleteNode) == 1 && deleteNode.left != null) {
    deleteNode.left.parent = parent;
    // left child's parent pointer skips the deleted node

    if (parent == null) // deleteNode is the root
        treeRoot = deleteNode.left;
    else if (parent.left == deleteNode)
        parent.left = deleteNode.left;
    else // parent.right == deleteNode
        parent.right = deleteNode.left;
}

- when deleteNode.right != null is analogous.
- Deletion takes $O(1)$ in this case (given the pointer to deleteNode).
```
Deletion Case 2

```java
else if (numChildren(deleteNode) == 1 && deleteNode.left != null) {
    deleteNode.left.parent = parent;
    // left child’s parent pointer skips the deleted node

    if (parent == null) // deleteNode is the root
        treeRoot = deleteNode.left;
    else if (parent.left == deleteNode)
        parent.left = deleteNode.left;
    else // parent.right == deleteNode
        parent.right = deleteNode.left;
}
```

- when `deleteNode.right` != null is analogous.
- Deletion takes $O(1)$ in this case (given the pointer to `deleteNode`).
- Case 3 is analogous to Case 2, with references to left and right reversed.
Deleting a node: case 4

In Case 4 our algorithm does the following:

1. find the predecessor or successor of the node
2. swap the keys of the two nodes
3. delete the node (from its new position)

Predecessor has no right child (and successor has no left child) →
After the swap, deleting the node will be in case 1, 2, or 3.

```javascript
myTree.delete(22);
```
Deletion Case 4

```java
else if (numChildren(deleteNode) == 2) {
    TreeNode<E> predNode = predecessorNode(deleteNode);
    swap(deleteNode, predNode);
    deleteNode(predNode);
}
```

- How long does it take to delete a node?
  - It takes $O(h)$ where $h$ is the height of the tree
  - It could be as good as $O(\log n)$ if the tree is balanced
Deletion Case 4

```java
else if (numChildren(deleteNode) == 2) {
    TreeNode<E> predNode = predecessorNode(deleteNode);
    swap(deleteNode, predNode);
    deleteNode(predNode);
}
```

How long does it take to delete a node?

- It takes $O(h)$ where $h$ is the height of the tree
- It could be as good as $O(\log n)$ if the tree is balanced
BST Delete More Example

- If node is a leaf, just delete it.
If node is a leaf, just delete it.
BST Delete More Example

- If node is a leaf, just delete it.
- If node has one child, move child up
BST Delete More Example

- If node is a leaf, just delete it.
- If node has one child, move child up
BST Delete More Example

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with predecessor or successor node and then delete
BST Delete More Example

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with **predecessor** or **successor** node and then delete
BST Delete More Example

- If node is a leaf, just delete it.
- If node has one child, move child up
- Else, swap with predecessor or successor node and then delete

```
  23
 /   \
10    25
 / \
8   14 24
    / \
     29 50
```
Summary of BST Operations

<table>
<thead>
<tr>
<th>operation</th>
<th>worst-case running time</th>
<th>balanced</th>
<th>unbalanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST search</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST insert</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST delete</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>tree rotation</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST minimum/maximum</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST predecessor/successor</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>swap</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numChildren</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Summary of BST Operations

<table>
<thead>
<tr>
<th>operation</th>
<th>worst-case running time</th>
<th>balanced</th>
<th>unbalanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>height</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST search</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST insert</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST delete</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>tree rotation</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST minimum/maximum</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST predecessor/successor</td>
<td>$O(h)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>swap</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numChildren</td>
<td>$O(1)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In practice, balanced BSTs are **augmented** such that height/predecessor/successor/minimum/maximum operations take $O(1)$.  

\[  
\text{In practice, balanced BSTs are augmented such that}  
\text{height/predecessor/successor/minimum/maximum operations take}  
\text{$O(1)$.}  
\]
Given a set of keys, how can we sort them using a binary search trees of height $h$?

- Insert all items (one by one) into the binary search tree
Given a set of keys, how can we sort them using a binary search trees of height $h$?

- Insert all items (one by one) into the binary search tree $\rightarrow$
  
  $n \times h = O(n \cdot h)$. 
Given a set of keys, how can we sort them using a binary search trees of height $h$?

- Insert all items (one by one) into the binary search tree $\rightarrow n \times h = O(n \cdot h)$.
- Traverse the tree in in-order traversal:
  - All smaller items are reported first (in the left subtree) then root is reported and then large items (in the right subtree) $\rightarrow$ the output will be sorted.
**Sorting Using BSTs**

Given a set of keys, how can we sort them using a binary search trees of height $h$?

- Insert all items (one by one) into the binary search tree → $n \times h = O(n \cdot h)$.
- Traverse the tree in in-order traversal:
  - All smaller items are reported first (in the left subtree) then root is reported and then large items (in the right subtree) → the output will be sorted.
  - It takes $O(n)$. 
Given a set of keys, how can we sort them using a binary search trees of height $h$?

- Insert all items (one by one) into the binary search tree $\rightarrow n \times h = O(n \cdot h)$.
- Traverse the tree in in-order traversal:
  - All smaller items are reported first (in the left subtree) then root is reported and then large items (in the right subtree) $\rightarrow$ the output will be sorted.
  - It takes $O(n)$.
- It is possible to sort $n$ items in $O(n \cdot h + n) = O(n \cdot h)$.
- If the tree is balanced, the time complexity will be $O(n \log n)$. 