Binary Search Trees

Contents

Abstract Binary Node

Source: `AbstractBinaryNode.java`.

```java
public abstract class AbstractBinaryNode<T extends Comparable<T>,
                                         N extends AbstractBinaryNode<T, N>> {
    protected T key;
    protected N parent;
    protected N left_child;
    protected N right_child;

    public AbstractBinaryNode(T key) {
        setKey(key);
    }
```

- Generics with two types, `T` and `N`.
- Recursive generics: `AbstractBinaryNode`.

Getters

```
public N getParent() { return parent; }

public N getLeftChild() { return left_child; }

public N getRightChild() { return right_child; }

public N getGrandParent() {
    return hasParent() ? parent.getParent() : null;
}

public N getUncle() {
    return hasParent() ? parent.getSibling() : null;
}

public N getSibling() {
    if (hasParent()) {
        N parent = getParent();
        return parent.isLeftChild((N)this)
             ? parent.getRightChild() : parent.getLeftChild();
    }

    return null;
}
```

- Node type: `this` vs. `(N)this`.

Setters

```java
public void setParent(N node) {
    parent = node;
}

public void setLeftChild(N node) {
    replaceParent(node);
    left_child = node;
}

public void setRightChild(N node) {
    replaceParent(node);
    right_child = node;
}

protected void replaceParent(N node) {
    if (node != null) {
        if (node.hasParent()) node.getParent().replaceChild(node, null);
        node.setParent((N)this);
    }
}

public void replaceChild(N oldChild, N newChild) {
    if (isLeftChild(oldChild)) setLeftChild(newChild);
    else if (isRightChild(oldChild)) setRightChild(newChild);
}
```

Binary Node

Source: `AbstractBinaryNode.java`.

```java
public class BinaryNode<T extends Comparable<T>>
             extends AbstractBinaryNode<T, BinaryNode<T>> {

    public BinaryNode(T key) {
        super(key);
    }
}
```

- Any abstract method that needs to be defined?

Abstract Binary Search Tree

```java
public abstract class AbstractBinarySearchTree<T extends Comparable<T>,
                                               N extends AbstractBinaryNode<T,N>>
{
    protected N root;

    public AbstractBinarySearchTree()
    {
        setRoot(null);
    }

    /** @return a new node with the specific key. */
    abstract public N createNode(T key);

    /** @return the root of this tree. */
    public N getRoot() {
        return root;
    }

    /** Sets the root of this tree to the specific node. */
    public void setRoot(N node)
    {
        if (node != null) node.setParent(null);
        root = node;
    }
```

- How to create a node type `N` : `createNode()` .

Search

```
protected N findNode(N node, T key) {
    if (node == null) return null;
    int diff = key.compareTo(node.getKey());

    if (diff < 0)
        return findNode(node.getLeftChild(), key);
    else if (diff > 0)
        return findNode(node.getRightChild(), key);
    else
        return node;
}

public N get(T key) {
    return findNode(root, key);
}

public boolean contains(T key) {
    return get(key) != null;
}
```

- Complexity: `findNode()`.

Add

```
public N add(T key) {
    N node = null;

    if (root == null)  setRoot(node = createNode(key));
    else               node = addAux(root, key);

    return node;
}

private N addAux(N node, T key) {
    int diff = key.compareTo(node.getKey());
    N child, newNode = null;

    if (diff < 0) {
        if ((child = node.getLeftChild()) == null)
            node.setLeftChild(newNode = createNode(key));
        else
            newNode = addAux(child, key);
    } else if (diff > 0) {
        if ((child = node.getRightChild()) == null)
            node.setRightChild(newNode = createNode(key));
        else
            newNode = addAux(child, key);
    }

    return newNode;
}
```

- Complexity: `add()` .

Remove

```
public N remove(T key) {
    N node = findNode(root, key);

    if (node != null) {
        if (node.hasBothChildren()) removeHibbard(node);
        else removeSelf(node);
    }

    return node;
}

protected N removeSelf(N node) {
    N parent = node.getParent();
    N child = null;

    if (node.hasLeftChild()) child = node.getLeftChild();
    else if (node.hasRightChild()) child = node.getRightChild();
    replaceChild(node, child);

    return parent;
}

private void replaceChild(N oldNode, N newNode) {
    if (isRoot(oldNode))
        setRoot(newNode);
    else
        oldNode.getParent().replaceChild(oldNode, newNode);
}
```

- Handle cases when the node to be removed has one or two children.
- What does `removeSelf()` return?

```
protected N removeHibbard(N node) {
    N successor = node.getRightChild();
    N min = findMinNode(successor);
    N parent = min.getParent();

    min.setLeftChild(node.getLeftChild());

    if (min != successor) {
        parent.setLeftChild(min.getRightChild());
        min.setRightChild(successor);
    }

    replaceChild(node, min);
    return parent;
}
```

References

- Binary Search Trees.