public class BTree implements Iterable<String> {
    Comparator<String> comp;
    int active = 0;
    RBTree rbt;

    BTree(Comparator<String> comp) {
        this.comp = comp;
    }

    public static BTree binTree(Comparator<String> comp) {
        return new BTree(comp);
    }

    public void build(Iterable<String> iter) {
        for (String st: iter) {
            if (active == 0) {
                this.rbt.ins(st);
            } else {
                throw new ConcurrentModificationException();
            }
        }
    }
}
Liskov Chapter 8: Polymorphic Abstraction
Polymorphic Abstraction

[Liskov]

“An abstraction that works for many types. A procedure or iterator can be polymorphic with respect to the types of one or more arguments. A data abstraction can be polymorphic with respect to the type of elements its objects contain.”
Liskov Chapter 7: Type Hierarchy
Terminology

[Liskov]

- Type hierarchy
- Subtype
- Supertype
- Multiple implementations
- Apparent Type vs. Actual Type
- Inheritance
- Concrete Class vs. Abstract Class
- Template Pattern
- Singleton pattern
Reasoning about the Substitution Principle

[Liskov]

• The *signature rule* ensures that if a program is type-correct based on the supertype specification, it is also type-correct with respect to the subtype specification.

• The *methods rule* ensures that reasoning about calls of supertype methods is valid even though the calls actually go to code that implements a subtype.

• The *properties rule* ensures that reasoning about properties of objects based on the supertype specification is still valid when objects belong to a subtype. The properties must be stated in the overview section of the supertype specification.
Sum of Products
Sum of Products

• Each *product* is represented by a separate class.

• Each of these classes is a subclass of a single parent class that represents the *sum*.
public static boolean isEmpty(StackInt s) {
    return s.isEmptyMethod();
}

public static boolean isEmpty(StackInt s) {
    if (s instanceof Empty) {
        return true;
    } else if (s instanceof Push) {
        return false;
    } else {
        throw new RuntimeException("Bug in isEmpty");
    }
}
Sum of Products

\[ T = A + B + C + D \]

A = \((\text{int} \times \text{String})\)
Sum of Products

StackInt = Empty + Push

Empty = ()

Push = StackInt x int

FSetString = EmptySet + Insert

EmptySet = ()

Insert = FSetString x String

FListInteger = EmptyList + Add

EmptyList = ()

Add = FListInteger x Integer
Polymorphism

• Parametric polymorphism

• Inheritance
  - Ad hoc polymorphism
  - Inclusion polymorphism
Parametric Polymorphism

• Generic types
Inheritance

• Extending a type in one of two ways:
  - Ad hoc polymorphism - extending a sum
  - Inclusion polymorphism - extending a product
Ad hoc polymorphism

- Extending a sum
- “A function or operator exhibits *ad hoc polymorphism* if it has at least two but only finitely many possible types.” [Web03]
- Abstract classes
- Interfaces
- Consider a recipe implementation of the FSetString ADT.

\[
\text{FSetString.size(FSetString.empty()) returns 0, but } \\
\text{FSetString.size(FSetString.insert(FSetString.empty(), "B")) returns 1}
\]
Inclusion polymorphism

- Extending product
- Adding a field