

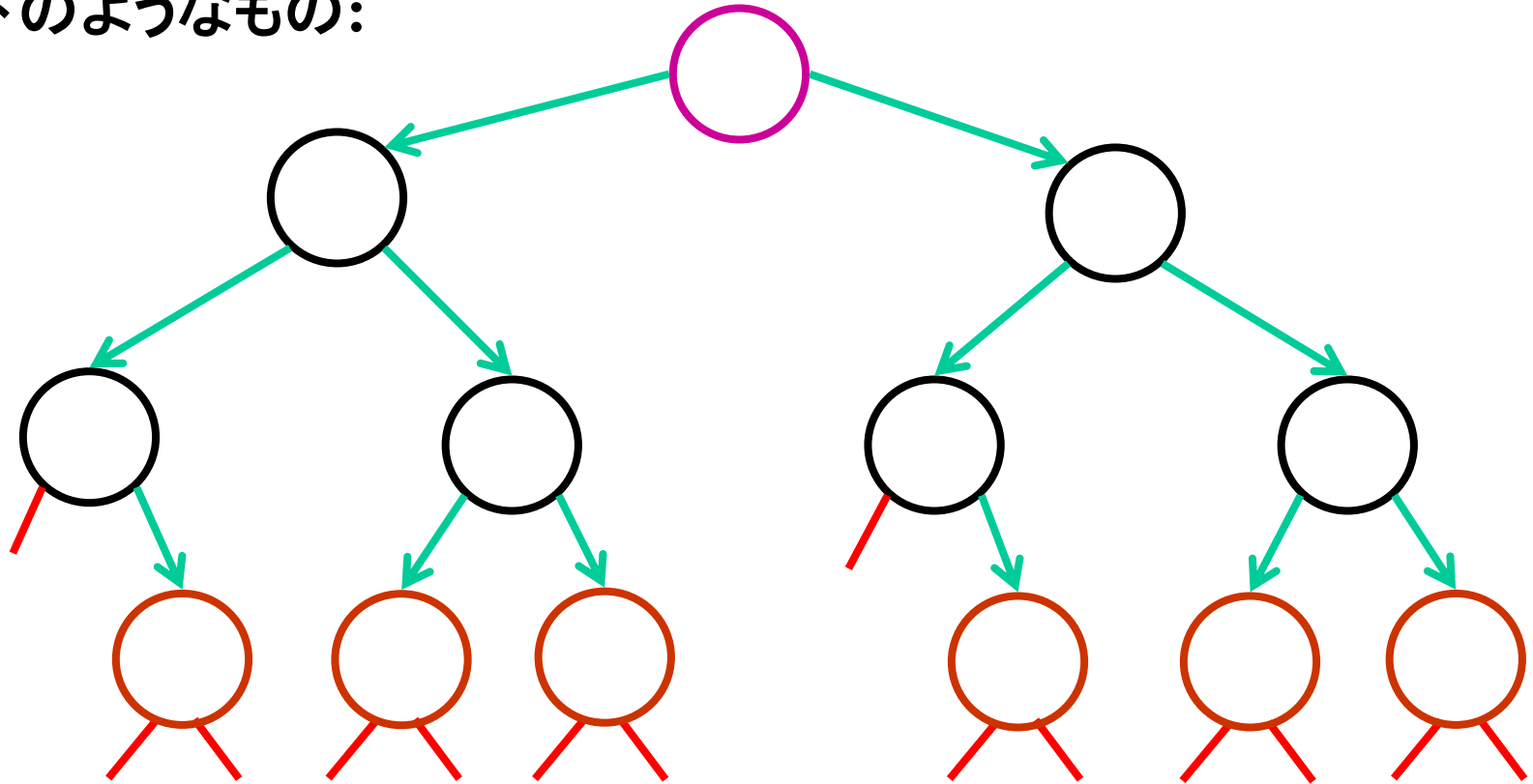
応用Javaプログラミング

第2回

- Generic Searching --
- Iteratorデザインパターン --

2分木 (binary tree)

• 以下のようなもの:

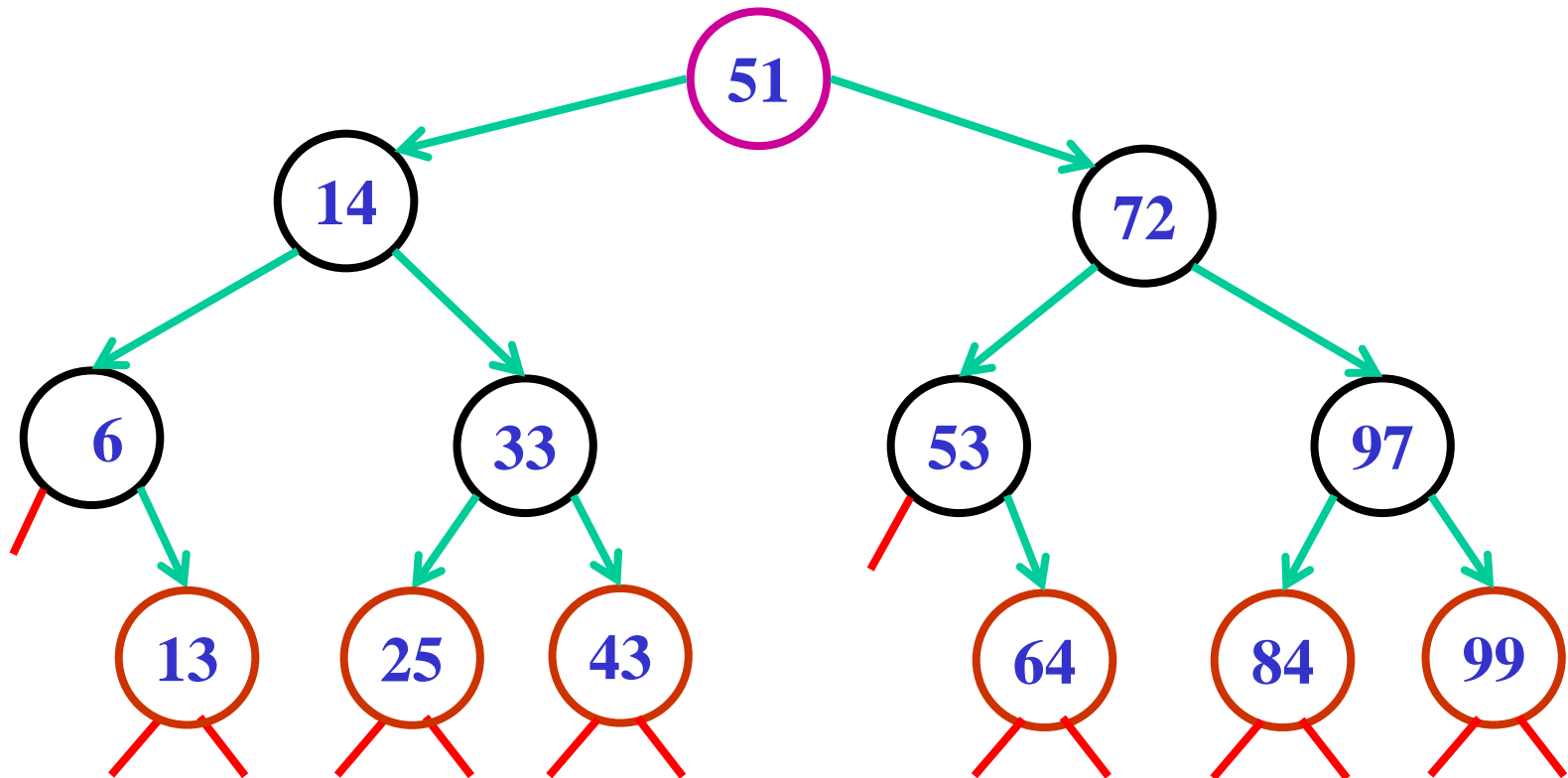


- 一番上のノード (node) を **根** と呼ぶ.
- 各ノードが高々二つの子供を持つ. 子供は左右に区別される.
どの2ノードも子供を共有しない.
- 子供が全部抜けているノードを **葉** と呼ぶ.

2分探索木 (binary search tree)

• 次の条件を満たすように2分木の各ノードにデータを置いた後の木:

条件: 葉以外の各ノード v に置かれたデータのキーは,
 v の左の子に置かれたデータのキー以上, かつ
 v の右の子に置かれたデータのキー以下.



∴ データは左から右へキーの小さい順に並んでいる.

オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである。

例：検索(searching)問題。

問：変更されないであろう箇所は何か？

答：処理順序：データの準備，データの挿入・検索・削除など，
結果の表示。

オブジェクト指向によるアプリケーション開発とは

変更されない箇所を軸に、頻繁に変更されるであろう箇所を
クラスやインターフェースに抽出するプログラミングスタイルである。

例: 検索 (searching) 問題.

問: 頻繁に変更されるであろう箇所は何か？

答: ① 検索の対象データ

② データの記憶方法 (i.e., データ構造)

③ データの大小を比べるための基準
(たとえば, 学生データから指定の
学籍番号を持つ学生を検索する際,
学籍番号の大小を比べる方法が
必要)

Integer,
Double,
String,
StudentData
など

Comparator,
など

配列,
LinkedList,
HashMap,
TreeMap
など

Generic Searching

ポイント: データがキー(**key**)と値(**value**)からなることを強制したい。
そのために、下記のインターフェースが用意されている。

```
//Javaで用意されているインターフェース
// Kはキーの型, Vは値の型
interface Map.Entry<K,V> {
    K getKey( ) ; //データのキーを取得するためのメソッド
    V getValue( ) ; //データの値を取得するためのメソッド
    V setValue(K value) ; //データの値をセットするためのメソッド
    ...
}
```

データがキー(**key**)と値(**value**)からなることを強制するには、
たとえば、データ構造にデータを挿入するためのメソッドの引数を
Map.Entry<K,V>型に限定すればよい。

また、データの大小を比べるための基準を指定するには、データ構造
のコンストラクタに**Comparator<K>**型の引数を用意すればよい。

Generic Searching (続)

Javaには探索のためのデータ構造として、

- `TreeMap<K,V>`, `HashMap<K,V>`, ... が用意されている

平衡2分探索木
を実装したもの.

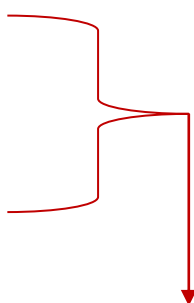
これらのデータ構造を使って整数や学生データの検索を行う
プログラムは `InegerSearchTree.java`, `InegerSearchHash.java`,
...にある.

次に、汎用2分探索木を自力で作ってみよう.

2分探索木のノード

問: 2分探索木のノードに記憶したい情報は何か？

答: ① 1つのデータ,
② 左の子ノードを指すポインタ,
③ 右の子ノードを指すポインタ.



これらを,
「2分探索木のノード」を表すクラス
のフィールドにすればよい.

問: 「2分探索木のノード」を表すクラスの持つべきメソッドは何か？

答: ① **コンストラクタ** (1つのノードのメモリを確保して初期化するため)
② **そのノードが葉** (子供を持たないノード) **か否かを判定する**
ためのメソッド. このメソッドは必須ではないが, あったほうが
便利そう)

2分探索木のノードを表すクラス

```
package genericSearching ;
```

```
//Kはデータのキーの型, Vはデータの値の型
```

```
public class BinSearchTreeNode<K,V> {
```

```
    private Map.Entry<K,V> data ; //このノードに置くデータ
```

```
    private BinSearchTreeNode<K,V> left, right ; //左の子と右の子の参照
```

```
    public BinSearchTreeNode( Map.Entry<K,V> data,    //コンストラクタ  
                               BinSearchTreeNode<K,V> left, BinSearchTreeNode<K,V> right) {
```

```
        this.data = data ;    this.left = left ;    this.right = right ;
```

```
    }
```

```
//葉を作るためのコンストラクタ
```

```
public BinSearchTreeNode( Map.Entry<K,V> data ) {
```

```
    this(data, null, null) ;
```

```
}
```

```
//葉か否かを判定するメソッド
```

```
public boolean isLeaf( ) {
```

```
    return left == null && right == null ;
```

```
}
```

```
//accessorメソッド(省略)
```

```
}
```

2分探索木

問: 2分探索木に記憶したい情報は何か？

答: ① 根ノードを指すポインタ(根ノードへの参照という),
② データの大小を比べるための基準(**Comparator<K>**型).

これらを,
「**2分探索木**」を表すクラス
のフィールドにすればよい.

問: 「**2分探索木**」を表すクラスの持つべきメソッドは何か？

答: ① データの**挿入** ② データの**削除** ③ データの**検索**
④ 木の**走査** ⑤ 木の**空判定** ⑥ **コンストラクタ**

木にあるデータを特定の順にスキャン

Comparator<K>を渡す

2分探索木を表すクラス

```
package genericSearching ;
```

```
//Kはデータのキーの型, Vはデータの値の型
```

```
public class BinSearchTree<K,V> {
```

```
    private BinSearchTreeNode<K,V> root ; //木の根
```

```
    private Comparator<K> comparator ; //データのキーの大小比較のため
```

```
//引数なしのコンストラクタがprivateなので, 他のクラスで呼び出せない
```

```
    private BinSearchTree( ) { }
```

```
//引数ありのコンストラクタ
```

```
    public BinSearchTree(Comparator<K> comparator) { //引数あり
```

```
        this.comparator = comparator ;
```

```
}
```

```
//空の木か否かを判定するメソッド
```

```
    public boolean isEmpty( ) {
```

```
        return root == null ;
```

```
}
```

```
//この木において指定キーを持つデータを検索するメソッド
```

```
    public V search(K key) {
```

```
        return search(root, key); //次ページの自作メソッドを呼び出す
```

```
}
```

探索を行うメソッド

//startを根とする部分木において指定キーを持つデータを検索するメソッド

```
private V search(BinSearchTreeNode<K,V> start, K key) {  
    if (start == null) return null ; //木が空の場合  
  
    //見つかった場合  
    if (comparator.compare(key, start.getData().getKey()) == 0)  
        return start.getData().getValue();  
  
    if (comparator.compare(key, start.getData().getKey()) < 0)  
        //左部分木で再帰的に検索  
        return search( start.getLeft(), key ) ;  
    else  
        //右部分木で再帰的に検索  
        return search(start.getRight(), key) ;  
}
```

挿入を行うメソッド

//startを根とする部分木に指定データを挿入するメソッド

```
private BinSearchTreeNode<K,V> insert(BinSearchTreeNode<K,V> start,
    Map.Entry<K,V> data) {
    if (start == null) //木が空の場合
        return new BinSearchTreeNode<K,V>(data) ;

    if (comparator.compare(data.getKey(), start.getData().getKey()) < 0)
        //左の子を根とする部分木に挿入すべきとき
        start.setLeft( insert(start.getLeft() , data) );
    else //右の子を根とする部分木に挿入すべきとき
        start.setRight( insert(start.getRight() , data) );
    return start ; //挿入した後の部分木を返す
}
```

//この木に指定データを挿入するメソッド

```
public void insert(Map.Entry<K,V> data) {
    root = insert(root, data) ; //上記の自作メソッドを呼び出す
}
```

//この木から指定キーを持つデータを削除するメソッド

```
public void delete(K key) {
    root = delete(root, key) ; //次頁の自作メソッドを呼び出す
}
```

//startを根とする部分木から指定キーのデータを削除して、結果の木の根の参照を返すメソッド

```
private BinSearchTreeNode<K,V> delete(BinSearchTreeNode<K,V> start, K key) {  
    if (start == null) return null ; //木が空の場合  
  
    if (comparator.compare(key, start.getData().getKey()) < 0)  
        //左の子を根とする部分木から削除すべきとき  
    { start.setLeft( delete(start.getLeft() , key) ); return start ; }  
    else //右の子を根とする部分木に挿入すべきとき  
    { start.setRight( delete(start.getRight() , key) ); return start ; }  
    else { //このstartノードを削除すべきとき  
        if ( start.isLeaf() ) return null ; //このstartノードが葉のとき、削除後の部分木が空。  
        else if (start.getLeft() == null) //このstartノードに左の子がないとき  
            return start.getRight() ; //削除後の部分木の根がstartの右の子。  
        else if (start.getRight() == null) //このstartノードに右の子がないとき  
            return start.getLeft() ; //削除後の部分木の根がstartの左の子。  
        else { //左の子も右の子がいるとき、まず次に大きいキーを持つデータを探す  
            BinSearchTreeNode<K,V> p = start.getRight() ;  
            while ( p.getLeft() != null ) p = p.getLeft() ;  
            Map.Entry<K,V> nextBigData = p.getData();  
  
            //右の木から次に大きいデータを削除  
            start.setRight( delete(start.getRight(), nextBigData.getKey()) );  
  
            //次に大きいデータでこのノードにあるデータを上書きする  
            start.setData(nextBigData) ; return start ;  
        }  
    }  
}
```

削除を行うメソッド

2分探索木の走査方法

- ① **preorder走査**: どの部分木 Γ の走査においても, 下記の順に走査:
 - (a) Γ の根を走査
 - (b) Γ の左部分木を走査
 - (c) Γ の右部分木を走査

- ② **inorder走査**: どの部分木 Γ の走査においても, 下記の順に走査:
 - (a) Γ の左部分木を走査
 - (b) Γ の根を走査
 - (c) Γ の右部分木を走査

- ③ **postorder走査**: どの部分木 Γ の走査においても, 下記の順に走査:
 - (a) Γ の左部分木を走査
 - (b) Γ の右部分木を走査
 - (c) Γ の根を走査

2分探索木の走査結果を覚える方法

どの走査結果も一列のデータ

(すなわち、全データが先頭から末尾まで順に一列に並んでいる感じ)。

ポイント: 一列のデータを先頭から末尾までスキャンしていきたいとき、その一列のデータを覚えるデータ構造に提供してほしいメソッド:

- ① まだスキャンしていない**データが残っているかを判定**するメソッド。
- ② まだスキャンしていない**次のデータを返す**メソッド。

考察: **配列と線形リスト**のどれも上記の①と②をサポートしている。

しかし、**配列と線形リスト**のような具体的なデータ構造を利用するよりも、上記の①と②をサポートしている抽象的なデータ構造を使ったほうがプログラムに柔軟性をもたらすのでよい。

上記の①と②をサポートしている抽象的なデータ構造として、Javaにおいて **Iterator<E>** が用意されている。

木にあるデータをinorder順に見ていくためのiteratorを返すメソッド

//startを根とする部分木に指定データを挿入するメソッド

```
public Iterator<Map.Entry<K,V>> inorderIterator() {  
    return inorder(root).iterator();  
}
```

この **public** と **private** に注目！

//startを根とする部分木にあるデータをinorder順にリストに挿入して返すメソッド

```
private List<Map.Entry<K,V>> inorder(BinSearchTreeNode<K,V> start) {  
    List<Map.Entry<K,V>> list = new LinkedList<Map.Entry<K,V>>();
```

```
    if (start == null) //木が空の場合  
        return list;
```

```
    //左の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll(inorder(start.getLeft()));
```

```
    //startにあるデータをリストに挿入  
    list.add(start.getData());
```

```
    //右の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll(inorder(start.getRight()));
```

```
    return list; //結果のリストを返す
```

```
}
```

情報の隠蔽を達成

private なので、走査結果を使う側から、線形リストが使われていることが見えない

Iteratorデザインパターン

ポイント: データ構造の各要素に対する繰り返し処理を抽象化することで、データ構造がどう実装されているかは処理と関係がないようにする.

Iterator役 (反復子役): データ構造の要素をスキャンしていくインターフェース(API)を定める役. 例では, **Iterator**インターフェースがこの役をつとめている.

ConcreteIterator役 (具体的反復子役): **Iterator**のAPIを実装した役. この例では, **preorderIterator**や**inorderIterator**や**postorderIterator**がこの役をつとめている.

Aggregate役 (集合体役): **Iterator役**を作り出すインターフェース(API)を定める役. この例では, **BinSearchTree**クラスがこの役を担っている.

木にあるデータをpreorder順に見ていくためのiteratorを返すメソッド

//startを根とする部分木に指定データを挿入するメソッド

```
public Iterator<Map.Entry<K,V>> preorderIterator( ) {  
    return preorder(root).iterator( ) ;  
}
```

//startを根とする部分木にあるデータをpreorder順にリストに挿入して返すメソッド

```
private List<Map.Entry<K,V>> preorder(BinSearchTreeNode<K,V> start) {  
    List<Map.Entry<K,V>> list = new LinkedList<Map.Entry<K,V>>( ) ;  
    if (start == null) //木が空の場合  
        return list ;  
  
    //startにあるデータをリストに挿入  
    list.add( start.getData( ) ) ;  
  
    //左の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll( preorder(start.getLeft( )) ) ;  
  
    //右の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll( preorder(start.getRight( )) ) ;  
  
    return list ; //結果のリストを返す  
}
```

木にあるデータをpostorder順に見ていくためのiteratorを返すメソッド

//startを根とする部分木に指定データを挿入するメソッド

```
public Iterator<Map.Entry<K,V>> postorderIterator( ) {  
    return postorder(root).iterator( ) ;  
}
```

//startを根とする部分木にあるデータをpreorder順にリストに挿入して返すメソッド

```
private List<Map.Entry<K,V>> postorder(BinSearchTreeNode<K,V> start) {  
    List<Map.Entry<K,V>> list = new LinkedList<Map.Entry<K,V>>( ) ;  
    if (start == null) //木が空の場合  
        return list ;  
  
    //左の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll( postorder(start.getLeft( )) ) ;  
  
    //右の子を根とする部分木にあるデータをinorder順にリストに挿入  
    list.addAll( postorder(start.getRight( )) ) ;  
  
    //startにあるデータをリストに挿入  
    list.add( start.getData( )) ;  
  
    return list ; //結果のリストを返す  
}
```

演習課題

課題: 自作`search, insert, delete`メソッドでは再帰的呼び出しを使っているが、再帰的呼び出しを使わないバージョンを書いて動作を確認せよ.