数据流分析报告

数据流测试

`BPlusTree`有5个方法：

1. `insert`：插入
2. `search`：查找
3. `searchRange`：范围查找
4. `delete`：删除
5. `toString`：输出遍历得到的字符串，按层

`insert`

插入操作涉及到3个方法：

```
// BPlusTree
public void insert(K key, V value) {
    root.insertValue(key, value);
    if (root.isOverflow()) {
        Node sibling = root.split();
        InternalNode newRoot = new InternalNode();
        newRoot.keys.add(sibling.getFirstLeafKey());
        newRoot.children.add(root);
        newRoot.children.add(sibling);
        root = newRoot;
    }
}

// Internal Node
void insertValue(K key, V value) {
    Node child = getChild(key);
    child.insertValue(key, value);
    if (child.isOverflow()) {
        Node sibling = child.split();
        insertChild(sibling.getFirstLeafKey(), sibling);
    }
}

// Leaf Node
void insertValue(K key, V value) {
    int loc = Collections.binarySearch(keys, key);
    int valueIndex = loc >= 0 ? loc : -loc - 1;
    if (loc >= 0) {
        values.set(valueIndex, value);
    } else {
        keys.add(valueIndex, key);
        values.add(valueIndex, value);
    }
}
```

- `key` 路径:

  `p1 = <invoke, 83, 327>`

  `p2 = <invoke, 83, 193, 327>`

- `value` 路径:

  `p1 = <invoke, 83, 325>` (插入相同的key)

  `p2 = <invoke, 83, 328>`

  `p3 = <invoke, 83, 193, 325>`

  `p4 = <invoke, 83, 193, 328>`

search

```
public V search(K key) {
    return root.getValue(key);
}

// Internal Node
V getValue(K key) {
    return getChild(key).getValue(key);
}

// Leaf Node
V getValue(K key) {
    int loc = Collections.binarySearch(keys, key);
    return loc >= 0 ? values.get(loc) : null;
}
```

`search()` 中只涉及变量 `key`，并且在整个过程中没有重新赋值。
因此，从函数调用开始到任意一个使用 `key` 的语句 `(53, 166, 253, 307)`，均为定义清除路径。

`p1 = <invoke, 53, 307>` (root为叶子节点)
`p2 = <invoke, 53, 166, 253, 307>` (root不为叶子节点)
`p3 = <invoke, 53, 198, 253, 166, 153, 307>` (一次环路, 即B+树需要找到child的child)

searchRange

```
public List<V> searchRange(K key1, RangePolicy policy1, K key2, RangePolicy
policy2) {
    return root.getRange(key1, policy1, key2, policy2);
}
```

`searchRange()` 中传入了变量 `key1`、`key2`、`policy1` 和 `policy2`。
和 `search()` 类似，这些传入的变量都不会被重新赋值。
`policy1` 和 `policy2` 共享路径:

`p1 = <invoke, 71, 358>` (root为叶子节点)
`p2 = <invoke, 71, 216, 358>` (root不为叶子节点)

`key1` 路径:

`p1 = <invoke, 71, 356>`
`p2 = <invoke, 71, 216, 253, 356>`
`p3 = <invoke, 71, 216, 253, 216, 253, 356>`

`key2` 路径:

`p1 = <invoke, 71, 357>`
`p2 = <invoke, 71, 216, 357>`

和 `search()` 类似，只需要构造一个深度为3的B+树即可覆盖所有路径。
在 `LeafNode.getRange` 中，有临时变量 `kIt` / `vIt` / `result` 存在分支情况（其他均为顺序执行，不需要过多考虑），因此需要对两个while循环进行分支测试以覆盖这三个变量的定义清除路径。

`delete`

```
void deleteValue(K key) {
    int loc = Collections.binarySearch(keys, key);
    if (loc >= 0) {
        keys.remove(loc);
        values.remove(loc);
    }
}
```

`key` 路径:

`p1 = <92, 313>`
`p2 = <92, 171, 313>` (root是Internal Node)
`p3 = <92, 171, 313, 173, 278, 192, 287>` (merge 以后子节点 overflow)
`p4 = <92, 171, 313, 120, 278, 259>` (root被删除到只有一个分支)