Design Patterns for Self-Balancing Trees

Dung “Zung” Nguyen
Stephen Wong

Rice University
Motivations

- A binary search tree of n elements can be skewed resulting in O(n) search.

Need a way to maintain the tree’s balance in order to guarantee O(log n) search.
Balanced Trees

- Classic balanced tree structures
  - 2-3-4 tree (see next slide)
  - red-black tree (binary tree equivalent of 2-3-4 tree)
  - B-tree (generalized 2-3-4 tree)
  - Difficult and complex. Where’s the code?

What’s the proper abstraction?
  - Need to decouple algorithms from data structures.
A 2-3-4 Tree is...

- **Empty**
  - 0-State: no data element + no sub-trees.

- **Non-Empty**, in 3 possible states:
  - 1-State: 1 data element + 2 sub-trees.
  - 2-State: 2 data elements + 3 sub-trees.
  - 3-State: 3 data elements + 4 sub-trees.
Variant vs. Invariant Operations

- Self-balancing insertion is **not** an intrinsic (invariant) operation of a tree.

- What are the invariant operations?
  - Gettors & Constructors.
  - Constructive and Destructive operations:
    - **Constructive**: Splice a tree into another.
    - **Destructive**: Split a tree into a 2-state.
Splittin’ and Splicin’

**Splice:**

**Split Up:**

*Intrinsic operations on the tree STRUCTURE, not the data!*
Structural Operations

\[ t1 \]

\[ t1.\text{splitUpAt}(1) \]

\[ t1.\text{splitDownAt}(1) \]

\[ t2 \]

\[ t1.\text{spliceAt}(1, t2) \]
Con/De-struction

t1

10

t1.splitUpAt(0)

10

t1.splitDownAt(0)

10

t2.spliceAt(0, t1)

10

t2
Visitor Design Pattern

Invariant: Host\textsubscript{i} calls case\textsubscript{i} of the visitor.

Fixed # of methods $\Rightarrow$ fixed # of hosts

Non-Extensible!
Generalized Visitors

Invariant: Hostᵢ calls caseAt(i) of the visitor.

Unbounded # of hosts!
Visitor Design Pattern

Composite Design Pattern: A non-empty TreeN has sub-trees that are TreeN.
public class ToStringAlgo implements ITreeNAlgo {

    // Constructors omitted

    public Object caseAt(int idx, TreeN host) {
        switch(idx) {
            case 0: { return "[ ]"; }  
            default: {
                String sData= "", sTrees="";
                for(int i = 0;i<idx;i++) {
                    sData += host.getDat(i)+" ";
                    sTrees += host.getChild(i).execute(toStringHelp,"\n")+
                }
                sTrees += host.getChild(idx).execute(toStringHelp,"\n")+
                return sData +"\n"+sTrees;
            }
        }
    }

    ITreeNAlgo toStringHelp = ...see next slide....
}
private final static ITreeNAlgo toStringHelp = new ITreeNAlgo() {
    public Object caseAt(int idx, TreeN host, 
        switch(idx) {
            case 0: {  return "|_[ ]"; } 
            default: {
                String sData= "", sTrees="";
                for(int i = 0;i<idx;i++) {
                    sData += host.getDat(i)+" ";
                    sTrees += prefix
                        + (String) host.getChild(i).execute(this, prefix+"| ")+"\n";
                }
                sTrees += prefix
                    + host.getChild(idx).execute(this, prefix+"| ").toString();
                return "|_ "+sData +"\n"+sTrees;
            }
        }
    
};
Vertical Data Transport

No net height change except at root and leaves!
Command Design Pattern

**ICommand**
+Object apply(Object inp)

Invokes

Performs a task.

No specified semantic!

Well-defined, but unrelated semantics.

Commands = Lambda Functions

Anonymous inner classes provide closures for lambdas!
Insertion Heuristics

- Insertion must take place at the leaf.
- Tree must grow only at the root.

Must transport data from the leaves to the root without affecting the height balance.
Problem: If a child node is too wide, it needs to split up and splice into its parent, but...

- The child node does not know *where* to splice into its parent.
- The child does not even have a reference to its parent.

Solution: Pass a command *(lambda)* forward from the parent to the child during the recursive call.
class SplitUpAndApply implements ITreeNAlgo {
    int _order;
    public SplitUpAndApply(int order) { _order = order; }

    public Object caseAt(int i, TreeN host, Object param) {
        if(i <= _order) return host;
        else {
            host.splitUpAt(i / 2);
            return ((ILambda)param).apply(host);
        }
    }
}

Lambda/commands enable decoupled communication!
Insertion Algorithm

- Find insertion point at the leaf and splice new data in.
- Use Split-and-Apply visitor to transport excess data upwards.
  - Visitor passed as parameter to recursive call.
  - Non-root: split-and-splice
  - Root node: split-and-no-op will cause entire tree to grow in height.
  - Abstract the splice/no-op as a command passed to the visitor!

Lambdas simplify code!
Insertion Dynamics

2 elements/node max

\( \lambda_0 \) no-op

\( \lambda_1 \) Splice into parent!

\( \lambda_2 \) Split up

Insert here!
public Object caseAt(int s, final TreeN host, final Object key) {
    switch(s) {
        case 0: { return host.spliceAt(0, new TreeN((Integer) key)); } 
        default: { host.execute(new ITreeNAlgo() {
            public Object caseAt(int s_help, final TreeN h, final Object cmd) {
                switch(s_help) {
                    case 0: { return ((ILambda)cmd).apply(new TreeN((Integer)key)) ;} 
                    default: {
                        final int[] x={0}; // hack to get around final
                        for(; x[0] < s_help; x[0]++) {
                            int d = h.getData(x[0]);
                            if (d >= (Integer)key) {
                                if (d == (Integer)key)) return h; // no duplicate keys
                                else break; } }
                        h.getChild(x[0]).execute(this, new ILambda() {
                            public Object apply(Object child) {
                                return h.spliceAt(x[0], (TreeN) child); } })
                        return h.execute(splitUpAndSplice, cmd); } } },
                    new ILambda() {
                        public Object apply(Object child){ return host; } })
            return host; } } }
}
Deletion Heuristics

- Deletion only well-defined at leaf.
- Data might exist anywhere in the tree.
- Tree can only shorten at root.

→ Push “candidate” data down from the root to the leaves.
→ Bubble “excess” data back to the root.

Must transport data from the root to the leaves and from the leaves to the root without affecting the height balance.
**Deletion Algorithm**

- Identify candidate data
  - split down at candidate and collapse with children.
  - If root is a 2-node, then tree will shorten.
- Data to delete will appear as 2-node below leaves.
- Use Split-and-Apply to transport excess data upwards.

*No Rotations!*
Deletion Dynamics

2 elements/node max

Remove “30”

Split-down candidate element and collapse with children as needed

Delete it!
Conclusions

- Proper abstraction leads to
  - Decoupling
  - Simplicity
  - Flexibility & extensibility
- Generalized Visitors open up new possibilities.
- Self-balancing trees illustrate
  - Abstract decomposition
  - Design patterns
  - Component-frameworks
  - Lambda calculus
  - Proof-of-correctness & complexity analysis