Non-linear containers

Trees

Definition of tree container

- We define a tree as a finite set of nodes, consisting of:
  - a distinct root node, and
  - a possibly empty list of disjoint trees, called the subtrees of the root.
- the root node is not an element of any of its subtrees,
- two distinct subtrees of a root do not share any nodes.

Tree definition

- tree defined recursively.
- tree must contain at least one node.
- trees defined are called “rooted” and “ordered,”
  - there is a distinguished root
  - subtrees are ordered by position on a list.
Parents, children

- The root node is the parent of the root nodes of its subtrees, which are its children.
- Children of the same parent are siblings.
- In Figure, node A is parent of nodes B and C; C is parent of D, E, and F, and so forth. D, E, and F are siblings and children of C.

Some definitions

- The number of subtrees a node has is called its degree. Node A has degree 2, node C degree 3, and node E degree 0.
- A node of degree 0 is a leaf.
- Path: A non-empty list of nodes $N_0, \ldots, N_k$ in which each node $N_i$ is parent of the next node $N_{i+1}$, for $0 \leq i < k$. For instance, A, C, D is a path in the tree shown in Figure 26.1.
- If $N_0, \ldots, N_k$ is a path, its length is $k-1$: number of edges traversed in going from $N_0$ to $N_k$.
- Note that there is a unique path from the root to every node in the tree.
- The depth of a node is the length of the path from the root to the node, i.e. number of ancestors excluding itself.
- The height of a node is the length of a longest path from the node to a leaf. Thus the depth of node C is 1, and its height is 2. The root has depth 0, and a leaf has height 0. The height of the tree is the height of the root.

Binary trees

- A binary tree is a finite set of nodes that is either empty, or consists of a root node and two disjoint binary trees, called the left and right subtrees of the root.
- Binary trees are not a subcategory of trees.
  - allow empty binary trees but not empty trees.
  - since we distinguish left and right subtrees, there are two distinct binary tree structures having two nodes: one in which root has an empty left subtree, and one in which root has an empty right subtree.

Two distinct binary trees

- A binary tree is a finite set of nodes that is either empty, or consists of a root node and two disjoint binary trees, called the left and right subtrees of the root.
- Binary trees are not a subcategory of trees.
  - allow empty binary trees but not empty trees.
  - since we distinguish left and right subtrees, there are two distinct binary tree structures having two nodes: one in which root has an empty left subtree, and one in which root has an empty right subtree.
Definitions

- binary tree can have at most $2^n$ nodes at level $n$.
- A binary tree of height $n$ can have as many as $2^{n-1}$ nodes, and as few as $n+1$.
- We call a binary tree full or complete if every level, except possibly the last, has as many nodes as possible. The last level node’s are found from left to right in sequence.
- length of the longest path of a binary trees containing $n$ nodes, is bounded by $\log_2 n$.

Binary tree API

```java
/**
 * Construct an empty BinaryTree.
 */
public BinaryTree () { ... }

/**
 * Construct a BinaryTree with the given
 * root item and left and
 * right subtrees.
 */
public BinaryTree ( Element rootItem,
                    BinaryTree<Element> left,
                    BinaryTree<Element> right) { ... }
```

Binary tree API

- queries for empty, and for accessing parts of a non-empty binary tree:
  - `public boolean isEmpty () { ... }`
  - `// require: !isEmpty()`
  - `public Element root () { ... }`
  - `// require: !isEmpty()`
  - `public BinaryTree<Element> left () { ... }`
  - `// require: !isEmpty()`
  - `public BinaryTree<Element> right () { ... }`

- Need iterator, and methods for modifying the binary tree: adding and deleting nodes, changing the item at a node, and so on.
Most direct implementation

- Consider an implementation based directly on the definition. Subclass BinaryTree to capture the two flavors, empty and non-empty:
  - class EmptyBinaryTree<Element> implements BinaryTree...
  - class NonEmptyBinaryTree<Element> implements BinaryTree...
- A NonEmptyBinaryTree has a root and two subtrees:
  
  ```java
  public class NonEmptyBinaryTree <Element> implements BinaryTree
  {
  ...
  private <Element> root; -- item at the root
  private BinaryTree left;  -- left subtree
  private BinaryTree right; -- right subtree
  }
  ```

Implementation difficulties

- Straight forward implementation difficulty: it will be natural to build the implementation where binary trees share node structure.
- If t1 is a binary tree that has already been constructed, the statement
  ```java
  BinaryTree t2 = t1.left();
  ```
  will create a “binary tree” with identical left and right subtrees:

![Diagram of binary tree with identical left and right subtrees]

- The situation becomes even worse if we include commands that allow a subtree of an existing tree to be replace with another existing tree, such as:
  ```java
  /**
   * Replace the left subtree of this BinaryTree with the specified
   * BinaryTree.
   * *
   * @param newLeft the new left subtree
   * @param newRight the new right subtree
   */
  public void setLeft (BinaryTree<Element> newLeft) {
  ...
  ```
  With this command, we can create cycles in the tree structure.
Implementation difficulties

- For example, with \( t_1 \) and \( t_2 \) as shown above, if we write:
  - \( t_2.setLeft(t_1) \);
- we end up with a structure like this

  ![Binary tree structure](image)

How can we address this situation?

- No method that modify the structure.
  - binary tree and iterator classes would not include methods for inserting or deleting nodes, or methods such as setLeft for modifying tree structure. developers find such restrictions unacceptable in a library class.
- Duplicate the subtree structure as necessary to avoid sharing nodes. Overhead may be prohibitive.
- Pass responsibility for maintaining structural coherence to the client.
- Most commonly adopted approach. straightforward, on implementor. Not as dangerous as it seems. Applications tend to use them as private components of some more abstract construct, and structural manipulation of the trees is localized. Thus there is little actual danger of an unanticipated client destroying the coherence of a tree structure.

Binary trees

- we guarantee that distinct BinaryTree instances do not share nodes. This is accomplished by either copying nodes, or by removing nodes from one tree if they are used to build another.
- For instance, one pair of methods for obtaining left and right subtrees remove the subtrees from the original tree:

```java
/* Remove and return the left subtree of this BinaryTree; not a proper query since it changes the state of this BinaryTree to insure that two BinaryTree instances do not share nodes. require: !this.isEmpty(); ensure: this.emptyLeft();
   this.setLeft(result).equals(old) */
public BinaryTree<ElementType> getLeft () {...}
```

Binary trees

- For example, the statement
  - \( \text{BinaryTree} \ t_2 = t_1 \text{getLeft}(); \)

  ![Binary tree transformation](image)
Binary trees

- public BinaryTree<Element> copyLeft () { ... }
- One pair of methods for setting left and right subtrees “steal” the nodes of the argument:
  - public void setLeft (BinaryTree <Element> left) { ... }
  - public void copyToLeft (BinaryTree <Element> left) { ... }

```
t1.setLeft(t2)
```

```
t1.copyToLeft(t2)
```

Binary Tree Implementations

- An old story again:
  - There are several implementations.
    - Array based
    - Link based
    - Linked based and optimized for traversals
  - They should be organized so that client select one if desired.
Array Implementation

- The root of the binary tree is stored in array element with index 0.
- If a node is at index $i$, left child at $2i+1$, right child at $2i+2$.
- An empty subtree can be denoted by null array entry.
- Advantages:
  - Given a parent node at location $k$, its children will be at $2k + 1, 2k + 2$.
  - Given a child, at location $k$, its parent is at location $(k-1)/2$.
  - Easy to write traversals for this structure.
- Disadvantages:
  - There is an entry per every level.
  - What is the size of the needed array, for a binary tree with $n$ nodes?
- This implementation is best used for complete or almost complete trees.

Linked implementations

- Define an implementation class `Node` that contains the Node element and references to the children:

```java
private class Node {
    Object element;
    Node left;
    Node right;
}
```

Iterators: preOrder

- For a non-empty binary tree:
  - visits the root
  - preorder traverses the left subtree
  - preorder traverses the right subtree.
Iterators: PostOrder

• A *postorder* traversal of a non-empty binary tree
  • postorder traverses the left subtree;
  • postorder traverses the right subtree;
  • visits the root.

Iterators: InOrder

• A *inorder* traversal of a non-empty binary tree
  • traverses the left subtree
  • visits the root
  • inorder traverses the right subtree.

Iterators: breath-first

• A *non-recursive traversal*

• *breadth-first or level-order* traversal
  – Visit the nodes of each level completely starting at the root and ending with last node of last level.

Inorder traversal

• traverses the left subtree
• visits the root
• inorder traverses the right subtree.

To find the first node, we move left down the binary tree as far as possible
Inorder traversal
At any point during the traversal:
• the left subtree of the current node will have been visited.
• In fact, the left subtree of any visited node has been visited.
• any ancestor whose right subtree the current node is in will have been visited

Preorder traversal
• visit the root;
• traverse the left subtree;
• traverse the right subtree.
• Finding the first preorder node: the root of the tree.
• At any point in the traversal,
  – all ancestors of the current node have been visited.
  – Also, if the current node is in the right subtree of an ancestor node, the left subtree of that node has been visited

How to find the next node after the current?
• If the current node has a non-empty right subtree, the next node is the first node in that subtree

find next node in a preorder
• There are three cases to consider.
  – If the current node has a non-empty left subtree, the next node is the root of the left subtree.
  – If the current node has an empty left subtree, but a non-empty right subtree, it is the root of the right subtree.
  – If the current node is a leaf, find the closest ancestor whose left subtree the current node is in, and that has a non-empty right subtree. The next node is the root of this right subtree.
Postorder traversal

- traverse the left subtree;
- traverse the right subtree;
- visit the root.

To find the first node
- move down the tree from the root, moving left whenever possible, until a leaf is reached.

At any point in the traversal
- all descendants of the current node have been visited.
- but none of the ancestors.
- If the current node is in the right subtree of an ancestor node, the left subtree of that node has been visited.

Implementations: Threaded binary trees

- In a linked implementation of a binary tree, with n nodes:
  - there are 2n child-referencing components.
  - Only n-1 of these are non-null
- Thus: more than half of the child-referencing components (right and left) contain null values.
Using null references

- replace null references with references that point back up into the tree – but to what?
- A node’s right child is null exactly when the next inorder node (the inorder successor) is an ancestor.
- A node’s left child is null exactly when the previous inorder node (the inorder predecessor) is an ancestor.
- Thus replace
  - null left references with references to a node’s inorder predecessor
  - null right references with references to a node’s inorder successor.

Array implementation application

Heaps

- Heap is a complete binary tree where the nodes must meet the following property:
  - order.lessThan(tree[i], tree[2*i + 1]) == true
  - And
  - order.lessThan(tree[i], tree[2*i + 2]) == true
- This is true for every i>=0, i<=tree.size()/2
- In other words, child nodes are less or equal than parent nodes.
Heap applications

- To write very efficient sorts, in situ.
- To implement efficient priority queues.

Heap sort

- Given an array:
  1. Build a heap from it.
  2. Swap top entry with last entry.
     - It breaks heap property.
  3. Reduce heap size by 1.
  4. Percolate top element to regain heap property.
  5. Do this until size is zero.

```java
public heapSort(IndexContainer list, Order o){
    Heap heap = buildHeap(list, o);
    for(int i = heap.size() - 1; i > 0; i--){
        // swap top with i entry
        swap(heap, 0, i);
        heapify(heap, i - 1, 0);
    }
}
```