1 Overview

At the end of this topic we would like you to understand the purpose and benefits of abstract data types, when you might use them, and how to implement one. We also hope to familiarize you with two basic abstract data types: binary trees and binary search trees.

2 Motivation

The built in type `Integer` is a data type, but on its own an integer is kinda worthless. Integers are useful only because there are operations defined on integers. For example, one can read integers from an input stream, add two integers to get a new one, increment an integer, or print an integer. It is the *combination* of a type and a collection of operations on that type that is useful.

All programming languages provide ways to build new data types from existing ones. For example, in most languages we can create sets, records, and lists. We can manipulate these types using built-in procedures, and we can write our own procedures to manipulate them, as we did for our `List` class. As they stand however, such data types are not *abstract* in the same way `Integer` is. The difference is that we can easily see the internal details of our constructed types, as we could see our `first` and `rest` of our `List`. Integers can be manipulated only through built-in operations; we do not (ordinarily) "see inside" an integer and look at its representation. However, if we define a new data type, there isn't a specific set of operations that apply to these objects. For example, if the new data type is a record type, then the fields of the record are visible to everyone, and any procedure can manipulate the internal details.

Exposing the details of data types is bad for at least two reasons (and these should sound familiar!):

1. Adding new data types adds to the complexity of the program, and adds to the difficulty of the programmer's task. A programmer who is working on any part of the program must understand the details of the new data type. For example, for someone to use our `List` type they would have to keep in mind our definition of `List`. It would be important to remember that there are `first` and `rest` fields, and their default values. This isn't too bad for a simple type such as `List`, but it becomes unmanageable for more complicated types. The result is an explosion in complexity; the only way to understand any one part of the program is to understand almost all parts simultaneously. Imagine the difficulty of changing type `List`: any code that uses `List` has to be examined to see if the change affects that piece of code. What a pain.

2. Even worse than having to remember the names of fields, default values, types, etc, is having to remember *how* the data type is supposed to be used. In the case of `List`, suppose that we write a procedure that counts the number of elements in the list. Now suppose someone else comes along and changes the implementation of our `List` class so that our procedure no longer works. Also a pain.

To make my long story short, we need a way to implement abstract data types: types together with operations that manipulate objects of that type. By bundling the data type and its related operations together into a package that other programmers can't look into, we protect the object from abusive users of the object and we can clarify and formalize the responsibilities of both the user and the implementor.

3 What is an ADT?

An *abstract data type*, or ADT, is a set of elements and a finite number of well-defined operations (methods) on these elements. The specification of an ADT does not have any design or implementation information associated with it, in other words, the user of an ADT should not know how the ADT's elements and methods are designed or implemented. The ADT is a "black box" whose internal representation is ignored.

3.1 Advantages of ADTs

There are many advantages to ADT's, some of which are:

1. they create natural modular decompositions (think "chunks")
2. they separate the design of programs from the implementation of data structures
3. they hide unnecessary information from the user
4. they allow the programmer to delay final implementation decisions (yay for procrastination!)

4 Trees

The best way to get the flavour of ADTs is by example. The first ADT we are going to look at is not the easiest one, but is very useful (and interesting!). It's called the tree.

4.1 Definition and Properties

A *tree* is a data structure that consists of a set of nodes and a set of edges that connect pairs of nodes such that there are no cycles. We are going to concern ourselves with *rooted trees*, which are trees with the additional propery that one node is distinguised as the root. Some properties of trees are:

1. every node except the root is connected by an edge from exactly one other node p. We say p is c's *parent* and c is p's *child*.
2. there is a unique path from the root to each node.
3. a tree is *connected* in the sense that if we start at any node n other than the root, move to the parent of n, the parent and parent of n, and so on, we eventually reach the root of the tree.
4. every tree with n nodes has $n - 1$ edges since each node, except the root, is the head of exactly one edge.

Example: Rooted Trees

Example: Not Rooted Trees

4.2 Parts of a Tree

Nodes that have no children are called *leaves*. An *internal node* is any node that is not a leaf. The terms *ancestor*, *descendant*, and *sibling* are defined as you would expect.

The *depth* of a node n is the number of edges you must traverse to get from the root to the node n. The *height* of a tree is the maximum depth of all the nodes. The *length* of a path between node n and m is the number of edges that we must follow to get from n to m.

Example: Parts of a Tree

4.3 Recursive Definition

We can also write a recursive definition: A tree is either

- empty (no nodes or edges), or
- a root node r with zero or more nonempty subtrees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by an edge from the root r.

4.4 Special Trees

Trees can be classified in many ways, but for now we're only going to discuss two particular types of trees:

ordered: the children of each node in an ordered tree have a designated order. Therefore you can refer unambiguously to the $1^{st}, 2^{nd}, \ldots, k^{th}$ child of a node with k children.

binary: a binary tree is an ordered tree with at most 2 children per node:

- the first child is called the left child
- the second child is called the right child
- a binary tree is different than an ordered tree with at most two children! Either child might be missing. There are two different such binary trees:

4.5 Our Tree So Far

So far, we may try to implement a binary tree as follows:

```
public class Node {
    Object      item;       // contains the node's information
    BinaryTree  left;       // the left child
    BinaryTree  right;      // the right child

    public Node( Object root, Node left, Node right ) {
        this.item  = root;
        this.left  = left;
        this.right = right;
    }
}
```

4.6 What operations do we do on a tree?

Now we know *what* a binary tree is, and we have a skeleton implementation for one, so our next question should be what do we do with it? What is it used for? Well, given the structure of a binary tree, you can imagine we may like to have the following operations:

makeEmptyTree( ) creates a new empty binary tree

leftChild(N) returns the left child of node N

rightChild(N) returns the right child of node N

sibling(N) returns the sibling of node N

isLeaf(N) returns whether node N is a leaf node

depth(N) returns the depth of node N

height(N) returns the height of the tree rooted at N

setLabel(N, l) sets the label/object of node N to l

How would I implement some of the above operations? Well a `BinaryTree` is very similar to our `List` class, and a `BinaryTree` is either an `Empty` (an empty binary tree) or a `Node` (a root node with a right and left subtree, both `BinaryTree`s themselves).

```
public abstract class BinaryTree {
  public abstract BinaryTree makeEmpty( );
  public abstract BinaryTree leftChild( );
  public abstract BinaryTree rightChild( );
  public abstract Object     getItem( );
  public abstract boolean    isLeaf( );
  public abstract int        height( );
}

class Empty extends BinaryTree {
  public BinaryTree makeEmpty( )    { return new Empty(); }
  public BinaryTree leftChild( )    { return null; }
  public BinaryTree rightChild( )   { return null; }
  public Object     getItem( )      { return null; }
  public boolean    isLeaf( )       { return true; }
  public int        height( )       { return 0; }
}
```

```
class Node extends BinaryTree {
  Object     item;
  BinaryTree left;
  BinaryTree right;

  public Node( )                    { this( null ); }

  public Node( Object item )        { this( item, new Empty(), new Empty() ); }

  public Node( Object item, BinaryTree left, BinaryTree right ) {
    this.item  = item;
    this.left  = left;
    this.right = right;
  }

  public BinaryTree makeEmpty( )    { return new Empty(); }
  public BinaryTree leftChild( )    { return left; }
  public BinaryTree rightChild( )   { return right; }
  public Object     getItem( )      { return item; }

  public boolean    isLeaf( ) {
    return ( left instanceof Empty ) && ( right instanceof Empty );
  }

  public int        height( ) {
    return 1 + Math.max( left.height( ), right.height( ) );
  }
}
```

Exercise Write a `sibling()` method that returns the sibling of a node of a binary tree.

Exercise Write a `duplicate()` method that returns a copy of a binary tree.

Exercise Write a `merge( BinaryTree t )` that merges a the tree `t` with the current tree.

Exercise Think about how you would implement a `depth()` method. What problems do you encounter? Do you have ideas as to how to fix these problems?

4.7 Array Representation

Although the above implementation of a binary tree works, we could also use an array to represent the structure. For example,

We can define useful operations in terms of index manipulation:

```
public int parent( int v ) {
    if( v == 0 )          { return -1; }
    else                  { return Math.floor( (v - 1) / 2); }
}

public int leftChild( int v ) {
    if( 2 * v + 1 >= n )  { return -1; }
    else                  { return ( 2 * v + 1 ); }
}

public int rightChild( int v ) {
    if( 2 * v + 2 >= n )  { return -1; }
    else                  { return ( 2 * v + 2 ); }
}
```

Exercise Try to implement the `height( int v )` method, in a similar manner.

Exercise Try to implement the `depth( int v )` method.

But why would you use an array implementation over our recursive one? Well, both implementations have their advantages. The array implementation uses less memory and makes extending the tree easy (new nodes are always added at the left-most spot in the array). Also, finding the depth of a node in the array implementation is easy, but finding the depth in our first implementation is really tricky! On the other hand, our recursive implementation is easy to understand, which makes implementing some methods easier. It's a toss-up.

4.8 Tree Traversals

Another operation we often like to perform on a binary tree is a traversal. The goal of a tree traversal is to visit each node of a binary tree in some specified order. We'll use recursive routines to do this, of course!

The three traversals we'll talk about aree called preorder, postorder, and inorder traversal.

4.8.1 Preorder

Tagline: Process a node before its child trees. The algorithm is

```
void visit( BinaryTree t ) {
    if( t instanceof Empty )              { return; }
    else {
        doSomethingWith( t.getItem( ) );  // Process the node n
        visit( t.LeftChild( ) );          // Process the left child
        visit( t.RightChild( ) );         // Process the right child
    }
}
```

Example: Tree Traversal

The traversal of the above tree processes the nodes in the order ABDECFHIG.

4.8.2 Postorder

Tagline: Process a node after both its children. For example, the traversal of the tree above would be in the order DEBHIFGCA.

Exercise: Write the code for a postorder traversal.

4.8.3 Inorder

Tagline: Process a node "between its children", that is, after its left child but before its right child. In the example above we would have traversed in the order DBEAHFICG.

Exercise: Write the code for an inorder traversal.

5 Binary Search Trees

A binary tree is a tree that contains integers in some order. A binary search tree is a special case of the binary tree. It has the following properties:

- all integers smaller than the integer at node n are stored in the left subtree of n
- all integers larger than the integer at node n are stored in the right subtree of n

Example: Binary Search Trees

Example: Non-Binary Search Trees

5.1 BST Operations

What are the operations we would like to execute on a binary search tree?

insert(B, v) inserts value v into the BST B

remove(B, v) removes value v from the BST B, if it exists

find(B, v) determines whether value v is in the BST B

findMin(B) returns the minimum value of the BST B

findMax(B) returns the maximum value of the BST B

printTree(B) prints the values of the BST B, in order

5.2 Searching a Binary Search Tree

In order to accomplish many of the operations above, we will first need an efficient way to search our binary search tree. For example, in order to remove an element from our tree, we will first have to find the element. The algorithm for searching a binary search tree is quite easy. Notice that it uses our favourite technique, recursion. Yay!

```
if tree is empty, return false
if ((object at root) == (search object)) return true
if ((object at root) <  (search object)) search in right subtree
if ((object at root  >  (search object)) search in left subtree
```

Exercise How would you write this algorithm in Java?

5.3 Finding the Minimum Value

To find the minimum value in a binary search tree, you need to traverse to the left-most leaf of the tree. It is also possible for the minimum value to be an internal node, but this node would have to have no left child. You can find this minimum using the following recursive method:

```
public static Object getMin( BinaryTree t )
{
    if( t.isEmpty() )                    // this is one base case
        return null;
    if( t.LeftChild() instanceof Empty )  // t is the min
        return t.getItem();               // this is the other base case
    else
        return getMin( t.LeftChild() );   // this is the recursive case
}
```

5.4 Insertion into a BST

The algorithm to insert a value v into a BST is:

- search for v in the data structure
- if v is not there, you will drop out of the BST at some node n
- create a new tree t containing v; if v is less than the contents of node n, make t the left child of n; otherwise, make it the right child of n.

Exercise Write a `insert( int v )` method.

5.5 Deletion from a BST

Deleting a node n from a BST is easy if:

- n is a leaf: change the reference in the parent node of n to an `Empty()`
- n has only one child c: change reference in parent to point to c, rather than n
- n has 2 children: tougher stuff here

6 Implementing ADTs

The implementation of an ADT is the programming in some language of

- what it means for a variable to be in our set, plus
- a procedure for each method of the ADT.

So first we write the specification of the ADT, i.e. we decide the elements of the set and the operations we want defined on this set. *Then* we choose a data structure to represent the ADT in order to implement these operations effectively. For example, we may choose a list or an array depending on whether the specification includes a delete operation or not (an array is bad at deletion, whereas a list is good).

It is important to choose your data structures carefully. A good data structure will

- make programming easier
- make your programs more readable (due to being more understandable)
- make your program more efficient

We will be exploring implementations of other common ADTs in the next few classes.