Abstract Data Type (ADT)

An Abstract Data Type, or ADT:
A type (set of values together with operations on them), where:
- We state in some fashion what the operations do
- We may give constraints on the operations, such as how much they cost (how much time or space they must take)

We use ADTs to help describe and implement many important data structures used in computer science, e.g.:
- set, bag, tree, binary tree, BST
- list or sequence, stack, queue, graph
- map, dictionary

ADT example: Set (bunch of different values)

Set of values: Values of some type E (e.g. int)
Typical operations:
1. Create an empty set (using a new-expression)
2. size() = size of the set
3. add(v) = add value v to the set (if it is not in)
4. delete(v) = delete v from the set (if it is in)
5. isIn(v) = "v is in the set"

Constraints: size takes constant time. add, delete, isIn take expected (average) constant time but may take time proportional to the size of the set.

We learn about hashing later on, it gives us such an implementation

Java Collections Framework

Java comes with a bunch of interfaces and classes for implementing some ADTs like sets, lists, trees. Makes it EASY to use these things. Defined in package java.util.

Homework: Peruse these two classes in the API package:
- ArrayList<E>: Implement a list or sequence — some methods:
  - add(e) add(i, e) remove(i) remove(e)
  - indexOf(e) lastIndexOf(e) contains(e)
  - get(i) set(i, e) size() isEmpty()

They use an array to implement the list!

i: a position. First is 0
e: an object of class E

References and Homework

- Text:
  - Chapters 10, 11 and 12

- Homework: Learn these List methods, from http://docs.oracle.com/javase/7/docs/api/java/util/List.html
  - add, addAll, contains, containsAll, get, indexOf, isEmpty, lastIndexOf, remove, size, toArray
  - myList = new List(someOtherList)
  - myList = new List(Collection<T>)
  - Also useful: Arrays.asList()
Stack\(<E>\) in java.util

Stack\(<E>\):
- `size()` (size)
- `isEmpty()` (isEmpty)
- `push(e)` (push)
- `pop()` (pop)
- `peek()` (peek)

Queue implements a queue:
- `size()` (size)
- `isEmpty()` (isEmpty)
- `push(e)` (push)
- `pop()` (pop)
- `peek()` (peek)

**Stack**:
- Last in first out (LIFO)
- Stack is actually a subclass of Vector, so you can use all of Vector’s methods

**Queue**:
- First in first out (FIFO)
- Peek: get top or first value but don’t remove it

---

Tree Overview

- Tree: recursive data structure (similar to list)
  - Each node may have zero or more successors (children)
  - Each node has exactly one predecessor (parent) except the root, which has none
  - All nodes are reachable from root

**Binary tree**:
- Tree in which each node can have at most two children: a left child and a right child

**General tree**

**Not a tree**

**List-like tree**

---

Readings & Homework on Trees

- **Textbook**:
  - Chapter 23 “Trees”
  - Chapter 24 “Tree Implementations”

- **Assignment #4**
  - “Collision Detection”
  - Based on “bounding box” trees

---

Tree Terminology

- **M**: root of this tree
- **G**: root of the left subtree of M
- **B, H, J, N, S**: leaves
- **N**: left child of P; **S**: right child
- **P**: parent of N
- **M and G**: ancestors of D
- **P, N, S**: descendants of W
- **J** is at depth 2 (i.e. length of path from root = no. of edges)
- **W** is at height 2 (i.e. length of longest path to a leaf)
- A collection of several trees is called a...

---

Binary Trees were in A1!

- You have seen a binary tree in A1.
- An elephant has a mom and pop. There is an ancestral tree!

```
  elephant
     /
    /   \
   mom   pop
    /     \
   mom   pop
```

---
Class for Binary Tree Node

```java
class TreeNode<T> {
    private T datum;
    private TreeNode<T> left, right;

    /** Constructor: one node tree with datum x */
    public TreeNode(T x) { datum = x; }

    /** Constr: Tree with root value x, left tree lft, right tree rgt */
    public TreeNode(T x, TreeNode<T> lft, TreeNode<T> rgt) {
        datum = x; left = lft; right = rgt;
    }
}
```

Points to left subtree

Points to right subtree

Class for General Tree nodes

```java
class GTreeNode {
    private Object datum;
    private GTreeCell left;
    private GTreeCell sibling;

    appropriate getters/setters
}
```

5
4
7
8
9
2
7
3
1

Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is implicit in ordinary textual representation
- Recursive structure can be made explicit by representing sentences in the language as trees: Abstract Syntax Trees (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- A parser converts textual representations to AST

Example

Expression grammar:

```
E → integer
E → (E + E)
```

In textual representation

```
(2 + 3)
```

In tree representation

```
E
  +
  |
  2
  |
  3
```

Recursion on Trees

Recursive methods can be written to operate on trees in an obvious way

Base case

- empty tree
- leaf node

Recursive case

- solve problem on left and right subtrees
- put solutions together to get solution for full tree
Searching in a Binary Tree

**Return true if x is the datum in a node of tree t.**

```java
public static boolean treeSearch(Object x, TreeNode t) {
    if (t == null) return false;
    if (t.datum.equals(x)) return true;
    return treeSearch(x, t.left) || treeSearch(x, t.right);
}
```

- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively

Binary Search Tree (BST)

If the tree data are ordered: in every subtree,
- All left descendents of node come before node
- All right descendents of node come after node

Search is MUCH faster

```java
/** Return true if x if the datum in a node of tree t.  
 Precondition: node is a BST */
public static boolean treeSearch (Object x, TreeNode t) {
    if (t == null) return false;
    if (t.datum.equals(x)) return true;
    if (t.datum.compareTo(x) > 0) return treeSearch(x, t.left);
    else return treeSearch(x, t.right);
}
```

Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses alphabetical order
  - Months appear for insertion in calendar order

What Can Go Wrong?

- A BST makes searches very fast, unless...
  - Nodes are inserted in alphabetical order
  - In this case, we’re basically building a linked list (with some extra wasted space for the left fields that aren’t being used)
- BST works great if data arrives in random order

Printing Contents of BST

Because of ordering rules for a BST, it’s easy to print the items in alphabetical order
- Recursively print left subtree
- Print the node
- Recursively print right subtree

```java
/** Print the BST in alpha. order. */
public void show () {
    show(root);
    System.out.println();
}
```

```java
/** Print BST t in alpha order */
private static void show(TreeNode t) {
    if (t == null) return;
    show(t.lchild);
    System.out.print(t.datum);
    show(t.rchild);
}
```
Tree Traversals

- "Walking" over whole tree is a tree traversal
  - Done often enough that there are standard names
  - Previous example: inorder traversal
    - Process left subtree
    - Process node
    - Process right subtree
  - Note: Can do other processing besides printing

Other standard kinds of traversals
  - Preorder traversal
  - Process node
  - Process left subtree
  - Process right subtree
  - Postorder traversal
    - Process left subtree
    - Process right subtree
    - Process node
  - Level-order traversal
    - Not recursive uses a queue

Useful Facts about Binary Trees

Max number of nodes at depth d: \(2^d\)

If height of tree is h:
  - Minimum number of nodes in tree: \(h + 1\)
  - Maximum number of nodes in tree:
    \[2^0 + \ldots + 2^h = 2^{h+1} - 1\]

Complete binary tree:
  - All levels of tree down to a certain depth are completely filled

Some Useful Methods

```java
/* Return true iff node t is a leaf */
public static boolean isLeaf(TreeNode t) {
    return t!=null &
    t.left == null &
    t.right == null;
}

/** Return height of node t using postorder traversal */
public static int height(TreeNode t) {
    if (t==null) return -1;
    //empty tree
    if (isLeaf(t)) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}

/** Return number of nodes in t using postorder traversal */
public static int nNodes(TreeNode t) {
    if (t==null) return 0;
    return 1 + nNodes(t.left) + nNodes(t.right);
}
```

Tree with Parent Pointers

- In some applications, it is useful to have trees in which nodes can reference their parents
- Analog of doubly-linked lists

```java
In some applications, it is useful to have trees in which nodes can reference their parents
```

Things to Think About

What if we want to delete data from a BST?

A BST works great as long as it's balanced

How can we keep it balanced? This turns out to be hard enough to motivate us to create other kinds of trees

Suffix Trees

- Given a string s, a suffix tree for s is a tree such that
  - each edge has a unique label, which is a nonnull substring of s
  - any two edges out of the same node have labels beginning with different characters
  - the labels along any path from the root to a leaf concatenate together to give a suffix of s
  - all suffixes are represented by some path
  - the leaf of the path is labeled with the index of the first character of the suffix in s
  - Suffix trees can be constructed in linear time

```
```
**Suffix Trees**

- Useful in string matching algorithms (e.g., longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)

**Huffman Trees**

- Fixed length encoding
  \[197^2 + 65^2 + 40^2 + 26^2 = 652\]
- Huffman encoding
  \[197^1 + 63^2 + 40^3 + 26^3 = 521\]

**Huffman Compression of “Ulysses”**

- Original size: 11904320
- Compressed size: 6822151
- 42.7% compression

**BSP Trees (c.f. k-d trees)**

- BSP = Binary Space Partition (not related to BST!)
- Used to render 3D images of polygons, e.g., Doom engine
- Example: Axis-aligned BSP Tree
  - Each non-leaf node \( n \) represents a region & splitting plane \( p \)
  - Left subtree of \( n \) contains all sub-regions on one side of \( p \)
  - Right subtree of \( n \) contains all sub-regions on the other side of \( p \)
  - Leaf nodes represent regions with associated data (e.g., geometry)
Tree Summary

- A tree is a recursive data structure
  - Each cell has 0 or more successors (children)
  - Each cell except the root has at exactly one predecessor (parent)
  - All cells are reachable from the root
  - A cell with no children is called a leaf
- Special case: binary tree
  - Binary tree cells have a left and a right child
  - Either or both children can be null
- Trees are useful for exposing the recursive structure of natural language and computer programs

A4: Collision Detection

- Axis-aligned Bounding Box Trees (AABB-Trees)
  - Object partitioning
  - Build one on each shape
  - Do tree-tree queries to detect overlapping shapes
- Some GUI material
- Available on CMS
- Due October 22, 11:59 pm
- Demo!