ABSTRACT DATA TYPES; LISTS & TREES

Lecture 10
CS2110 – Fall 2014

References and Homework

2

- 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()

Abstract Data Type (ADT)

4

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
list or sequence, stack, queue
map, dictionary
tree, binary tree, BST
graph

ADT example: Set (bunch of *different* values)

6

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

7

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

Java Collections Framework

9

Homework: Peruse following in the API package:
LinkedList<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()
getFirst()	getLast()		

Uses a doubly linked list to implement the list or sequence of values

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

Stack<E> in java.util   Queue not in java.util

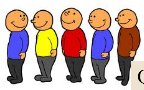
10

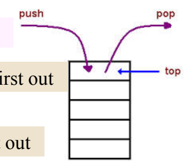
Stack<E>: Implements a stack:
- size() isEmpty()
- push(e) pop() peek()

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

Stack is actually a subclass of Vector, So you can use all of Vector's methods

peek: get top or first value but don't remove it





Stack LIFO last in first out

Queue: FIFO first in first out

TREES

Lecture 10
CS2110 – Fall 2013

Readings & Homework on Trees

18

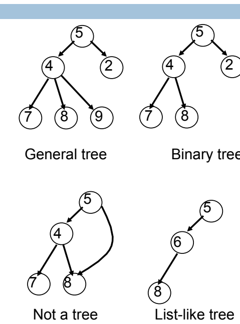
- Textbook:
  - Chapter 23 "Trees"
  - Chapter 24 "Tree Implementations"
- Assignment #4
  - "Collision Detection"
  - Based on "bounding box" trees

Tree Overview

20

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

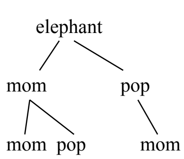


Binary Trees were in A1!

21

You have seen a binary tree in A1.

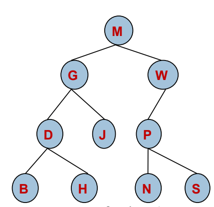
An elephant has a mom and pop. There is an ancestral tree!



Tree Terminology

22

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: *descendents* 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 ...?



Class for Binary Tree Node

23

```
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

more methods: getDatum, setDatum, getLeft, setLeft, etc.

Binary versus general tree

24

In a binary tree each node has exactly two pointers: to the left subtree and to the right subtree

- Of course one or both could be *null*

In a general tree, a node can have any number of child nodes

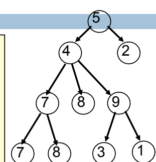
- Very useful in some situations ...

Class for General Tree nodes

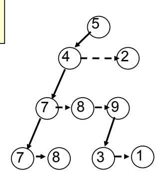
25

```
class GTreeNode {
1.   private Object datum;
2.   private GTreeCell left;
3.   private GTreeCell sibling;
4.   appropriate getters/setters
}
```

- Parent node points directly only to its leftmost child
- Leftmost child has pointer to next sibling, which points to next sibling, etc.



General tree



Tree represented using GTreeCell

Applications of Trees

26

- 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

27

Expression grammar:

- E → integer
- E → (E + E)

In textual representation

- Parentheses show hierarchical structure

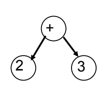
In tree representation

- Hierarchy is explicit in the structure of the tree

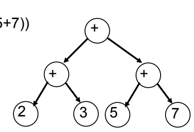
Text | AST Representation

-34

(2 + 3)



((2+3) + (5+7))



Recursion on Trees

28

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

Base case

- empty tree
- leaf node

Recursive case

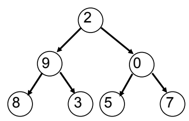
- solve problem on left and right subtrees
- put solutions together to get solution for full tree

Searching in a Binary Tree

29

```
/** Return true iff x is the datum in a node of tree  t*/
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



Searching in a Binary Tree

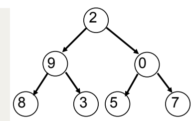
30

```
/** Return true iff x is the datum in a node of tree  t*/
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);
}
```

Important point about t. We can think of it either as

(1) One node of the tree OR

(2) The subtree that is rooted at t



Binary Search Tree (BST)

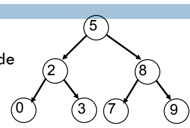
31

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

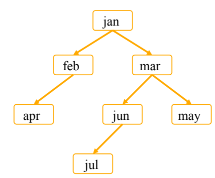



```
/** Return true iff x if the datum in a node of tree t.
    Precondition: node is a BST */
public static boolean treeSearch (Object x, TreeNode t) {
    if (t== null) return false;
    if (t.datum.equals(x)) return true;
    if (t.datum.compareTo(x) > 0)
          return treeSearch(x, t.left);
 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

Building a BST

32

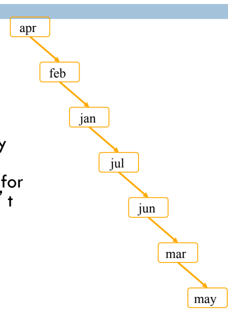
- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree
- This can be done using either recursion or iteration
- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*



What Can Go Wrong?

33

- 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

34

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

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

35

- "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

Some Useful Methods

36

```
/** 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);
}
```

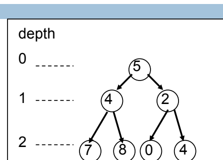
Useful Facts about Binary Trees

37

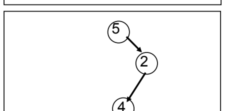
Max number of nodes at depth d: 2^d

If height of tree is h
- min number of nodes in tree: $h + 1$
- Max 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



Height 2,
maximum number of nodes

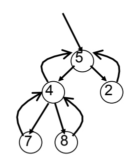


Height 2,
minimum number of nodes

Tree with Parent Pointers

38

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



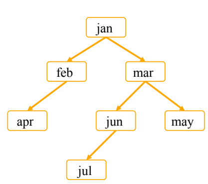
Things to Think About

39

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*



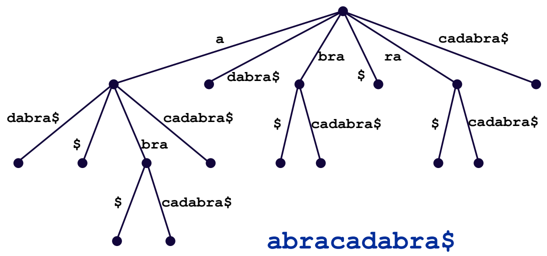
Suffix Trees

40

- 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

41



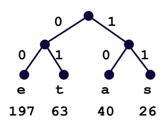
Suffix Trees

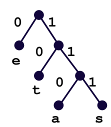
42

- 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

43





Fixed length encoding
197*2 + 63*2 + 40*2 + 26*2 = 652

Huffman encoding
197*1 + 63*2 + 40*3 + 26*3 = 521

Huffman Compression of "Ulysses"

44

' '	242125	00100000	3	110
'e'	139496	01100101	3	000
't'	95660	01110100	4	1010
'a'	89651	01100001	4	1000
'o'	88884	01101111	4	0111
'n'	78465	01101110	4	0101
'i'	76505	01101001	4	0100
's'	73186	01110011	4	0011
'h'	68625	01101000	5	11111
'r'	68320	01110010	5	11110
'l'	52657	01101100	5	10111
'u'	32942	01110101	6	111011
'g'	26201	01100111	6	101101
'f'	25248	01100110	6	101100
'.'	21361	00101110	6	011010
'p'	20661	01110000	6	011001

44

Huffman Compression of "Ulysses"

45

...

'7'	68	00110111	15	111010101001111
'/'	58	00101111	15	111010101001110
'X'	19	01011000	16	0110000000100011
'&'	3	00100110	18	011000000010001010
'%'	3	00100101	19	0110000000100010111
'+'	2	00101011	19	0110000000100010110

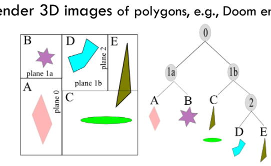
- original size 11904320
- compressed size 6822151
- 42.7% compression

45

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

47

- BSP = Binary Space Partition (not related to BST!)
  - Used to render 3D images of polygons, e.g., Doom engine



http://www.bogotobogo.com/Games/spatialdatastructure.php

- 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

48

- 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

49

- 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!

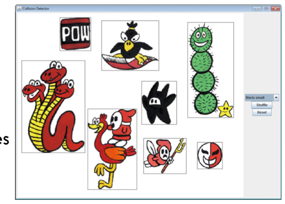


Figure 5: Shapes drawn with their bounding boxes (bounding rectangles).