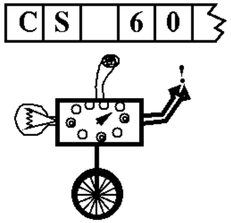


Trees and Rex Functions

September 10, 2001

Topics for Today

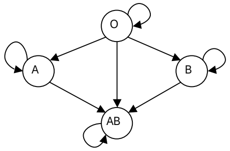
- More examples of representing data as lists
  - In particular, trees
- More ways to actually do things in rex
  - Also known as: introduction to high-level functional programming.

Review

- In the previous lecture we talked about how one might represent a number of different information structures
  - Sets
  - Dictionaries
  - Directed and Undirected Graphs
  - Binary Relations

Warmup

- Directed graph as a list of pairs:

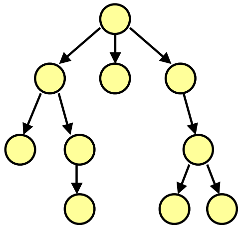


```
[["A", "A"],  ["A", "AB"],  ["B", "B"],
 ["B", "AB"], ["AB", "AB"], ["O", "A"],
 ["O", "AB"], ["O", "B"],   ["O", "O"]]
```

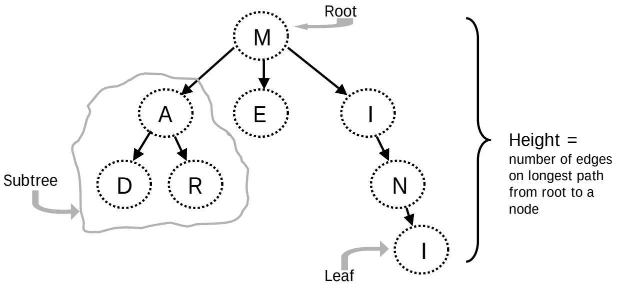
Alternate Representation?

Trees

- Trees are a particular form of directed graph

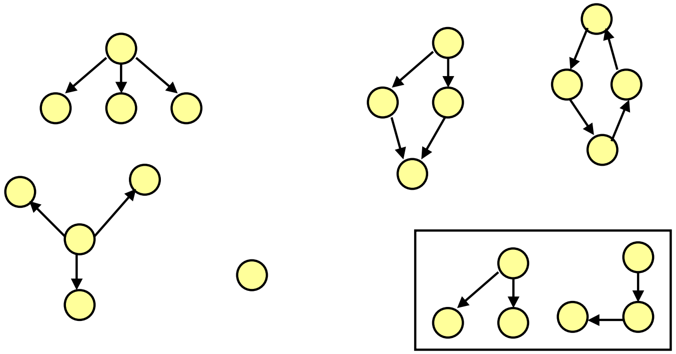


Some Tree Terminology



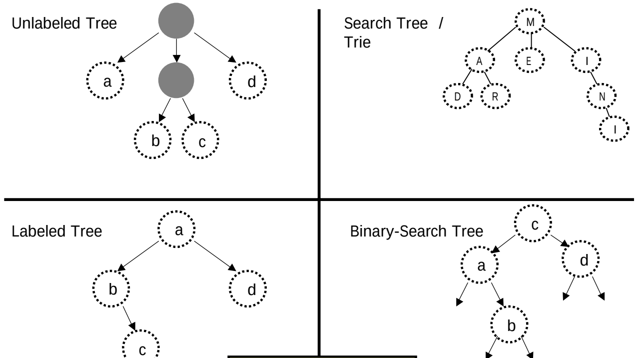
Height =
number of edges on longest path from root to a node

When is a Graph a Tree?



When is a Graph a Tree?

Many Different Types Of Trees



Read Chapter 2!

Encoding Labeled Trees

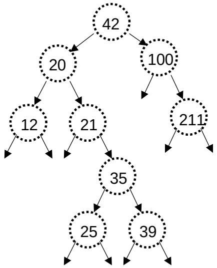
- Every tree is a graph, so we could always use the graph encoding.
  - But can we take advantage of the fact that the graph is a tree?

Encoding Labeled Trees

Ordered Graphs

- A graph is said to be ordered if the order in which arcs leave a node matters.
  - When using representations, an ordering is often implicit in the representation.
  - We can choose to ignore this implicit order, or not.
- Example: Binary Trees
  - A binary tree is a tree where every node is either a leaf or has exactly two children.
  - We specify one of these children as the "left" child and one as the "right" child.

Binary Search Trees



Identifying features:

- Every node has distinguished left and right subtrees
- Each node has a value or “key”
- A root’s value is always **greater than** all nodes in a left subtree
- A root’s value is always **less than** all nodes in a right subtree
- Each node has a distinct key

Part 2: Functions in rex

Running rex

- When you type `rex` at the command prompt, you enter the rex interpreter.
- You can then type in either a new definition or an expression to be evaluated.

```
rex > 3+5;
8
rex > x=5;
1
rex > 3+x;
8
```

Defining Functions in rex

- One way to define a function is to directly specify its definition, as commonly done in mathematics.

```
f(x)   = x*5;
g(x,y) = f(x)+y;
h(x)   = g(f(x),9);
```

Defining Functions in rex

- Sample transcript:

```
rex > f(x) = x+1;
1
rex > f(6);
7
rex > f(f(2));
4
```

Built-in Functions in rex

To make programming more interesting, there are a number of functions that have been pre-defined for you in rex.

`length`

- Returns the length of a list.
- Examples:

```
length([2, 3, 5, 7, 11]) ==> 5
length([ ]) ==> 0
length(5) ==> error
```

[Note: Book uses ==> to mean
"ultimately computes to"]

`append`

- Used to append together two lists:

```
append([1, 2, 3], [4, 5])
          ==>
    [1, 2, 3, 4, 5]
```

```
append([1, 2], []) ==> [1, 2]
```

Simple rex Code

```
rex > f(x,y) = length(append(x,y));
1

rex > f([1,2],[3,4,5]);
5

rex > f([],[]);
0
```

`reverse`

- Used to reverse the elements of a list:

```
reverse([1, 2, 3]) ==> [3, 2, 1]
```

- It applies to the top-level elements only:

```
reverse([ [1,2], [3,4] ])
          ==>
    [ [3,4], [1,2] ]
```

`member`

- Checks whether an item is an element of a list, returning `1` if so and `0` otherwise.
- Examples:

```
member(7, [2, 3, 5, 7, 11]) ==> 1

member(9, [2, 3, 5, 7, 11]) ==> 0

member("foo", [ ["foo","bar"] ] ) ==> 0
```

assoc

- Used to look up values in an association list.
- Given a key and an association list, returns the first pair in that list with the given key
  - It returns `[ ]` if there no such pair exists.

```
assoc("octahedron", polyhedrondict)
==>
["octahedron", [8, 3]]

assoc("cubeahedron", polyhedrondict) ==> []
```

cons

- Used to construct a list out of a first element and another list (short for "construct").
- Examples:

```
cons(5, [10, 15, 20])
==>
[5, 10, 15, 20]
```

cons is not append

- The distinction between `cons` and `append` can be confusing and should be noted carefully:

```
cons(1, [2, 3]) ==> [1, 2, 3]

append(1, [2, 3]) ==> error

cons([1], [2, 3]) ==> [[1], 2, 3]

append([1], [2, 3]) ==> [1, 2, 3]
```

The Difference

- The first argument to `cons` becomes an *element* of the resulting list.

```
cons([1, 2, 3], [4, 5])
==>
[ [1, 2, 3], 4, 5]
```

- The first argument to `append` becomes a *prefix* of the resulting list.

```
append([1, 2, 3], [4, 5])
==>
[1, 2, 3, 4, 5]
```

`first` and `rest`

- Returns the first element or all but the first element of a non-empty list:

```
  first([1, 2, 3, 4]) ==> 1
rest([1, 2, 3, 4]) ==> [2, 3, 4]
```

Relationships

- In rex, == is the equality-testing function, but we'll also use it to express relationships:

```
cons(F, R) == append([F], R)
length(append(L,M)) == length(L) + length(M)
length(cons(F,R)) ==
first(cons(F,R)) ==
rest(cons(F,R)) ==
length(reverse(L)) ==
reverse(append(L,M)) == ?
reverse(cons(F,R)) == ?
```

`sort`

- Sorts a list into "increasing" order

```
sort([3, 1, 4, 2]) ==> [1, 2, 3, 4]
```

- Numbers are ordered numerically
- Strings are ordered alphabetically
- Lists are ordered lexicographically

Relationships for sort

```
length(sort(L)) ==


sort(reverse(L)) ==


sort(append(sort(L), sort(M))) ==
```