CS 310: HW3 Ackcell Spreadsheet

Chris Kauffman

Week 11-1

Logistics

Reading

- 21: Priority Queue/Binary Heap
- 6.9: Priority Queue Interface

Today's Menu

Priority Queues

HW3

- Milestones due tonight
- Final due Tuesday

End Game

7/13 Thu	BST Removal, AVL Trees
7/18 Tue	AVL / Red-Black Trees
7/20 Thu	Priority Queues
	Binary Heaps
	HW3 Milestones Due
7/25 Tue	HeapSort
	Review / Evals
	HW3 Final Due
7/27	Final Exam

Final Exam in 1 week

Will post Java Jeopardy review later today, play Tue

HW3: AckCell

- Implement a spreadsheet *model*
- Cells contain data: Numbers, Strings, Formulas
- Formulas are parsed into trees of `FNodes`
- DAGs track dependencies between things, prevent cycles, discuss next time
- Spreadsheet maps IDs like `A17` to cells, notifies cells of changes in their dependencies

- Milestones Concern only the `Cell` class, due next Thu
- Project designed write classes in this order
  - First `Cell.java`
  - Second `DAG.java`
  - Lastly tie them together in `SpreadSheet.java`
- Final deadline Tue before final

Cell Formulas

- Cell formulas are the first hurdle
- Provided `FNode.parseFormulaString(str)` parses formulas

```
FNode root = FNode.parseFormulaString("=(100 + A2) - 10 / (CX5 * BB8)");
```

- Requires `formula.jar` library; experiment on command line

```
> javac -cp formula.jar:. FNode.java
> java -cp formula.jar:. FNode
usage:   java -jar formula.jar 'formula to interpret'
Example: java -jar formula.jar '=A1 + -5.23 *(2+3+A4) / ZD11'
> java -cp formula.jar:. FNode '=1 + 2*A4 / (7+BB8) - Z2'
-
  +
    1
    /
      *
        2
        A4
      +
        7
        BB8
  Z2
```

- Discuss basic strategy for walking/evaluating `FNode` trees
- Required for `cell.evalFormulaTree(str,cellMap)` and `cell.getUpstreamIDs()`

Cell: Subclass vs Single Class

```
abstract class Cell{
  public abstract String kind();
  public static Cell make(String s){
    if(s is a formula){
      return new FormulaCell(s);
    } else if(s is a number){
      return new NumberCell(s);
    }
    ...
  }
}
class StringCell extends Cell{
  @Override public String kind(){
    return "string";
  }
}
class FormulaCell extends Cell{
  private FNode formulaRoot;
  @Override public String kind(){
    return "formula";
  }
}
class NumberCell extends Cell{
  @Override public String kind(){
    return "number";
  }
```

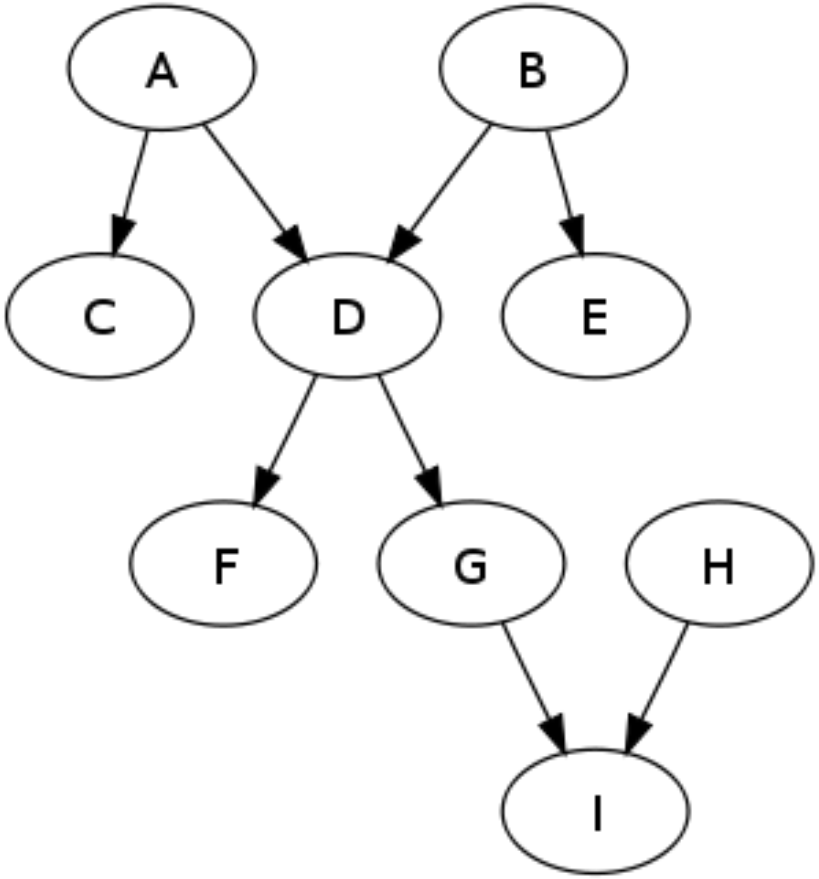
```
public class Cell{
  private String myKind;
  private FNode root;
  public static Cell make(String s){
    Cell c = new Cell();
    if(s is a formula){
      c.kind = "formula";
      c.root = set up tree;
    } else if(s is a number){
      c.kind = "number";
      c.root = null;
    } else {
      c.kind = "string";
      c.root = null;
    }
    return c;
  }
  public String kind(){
    return this.kind;
  }
}
```

Neither of these are "right", just tradeoff design differently

Structure of Code for evalFormulaTree()

```
public static Double eval(node, cellMap){
  if(node.type == TokenType.Plus){
    Double leftVal  = eval(node.left);
    Double rightVal = eval(node.right);
    return leftVal + rightVal;
  }
  else if(node.type == TokenType.Minus){
    Double leftVal  = eval(node.left);
    Double rightVal = eval(node.right);
    return leftVal - rightVal;
  }
  // Cases for multiply, divide, negate
  else if(node.type == TokenType.Number){
    // node.data contains a string of a number
    // converts it to a double and return
  }
  else if(node.type == TokenType.CellID){
    // node.data contains a string of a cell ref like C12
    // look it up in cellMap and return its number
    // throw evalForumlaException if the cell has no number value
  }
  else{
    throw new RuntimeException("Error with TokenType '"+node.type+"'");
  }
}
```

DAGs: Directed Acyclic Graphs



- Directed Acyclic Graph
- Graph: Nodes connected by links (vertices connected by edges)
- Directed: Links between Nodes have a direction (arrow head)
- Acyclic: No cycles, can't go in circles

HW3 and DAGs

- `DAG.java` is an independent class, doesn't know anything about `Cell` or `Spreadsheet`
- Create an empty `DAG` and start adding *upstream links* to it with `add(id,links)`

```
DAG dag = new DAG();
dag.add("A1",DAGDemo.toSet("B1","C1","D1"));
dag.add("B1",DAGDemo.toSet("C1","D1"));
```

- Keeps track of upstream links and downstream links
- Useful in spreadsheet context

```
spreadsheet.setCell("A1","=B1 + C1 * D1");
```

  - `A1` depends on `B1 C1 D1`: they are *upstream*
  - Whenever `B1 C1 D1` are changed, notify `A1` as it is *downstream* from them

- Play with this in DrJava: detect cycles

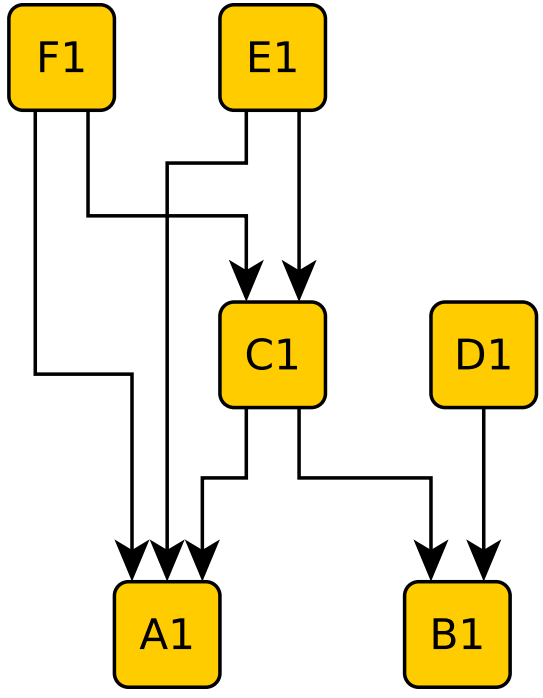
Exercise: Draw this DAG

- `DAGDemo.java` constructs this DAG with repeated `add(id,upstream)` calls
- Draw the DAG based on downstream links

```
Upstream Links:
  A1 : [E1, F1, C1]
  C1 : [E1, F1]
  B1 : [D1, C1]
Downstream Links:
  E1 : [A1, C1]
  F1 : [A1, C1]
  D1 : [B1]
  C1 : [A1, B1]
```

Answer: Draw this DAG

```
Upstream Links:
  A1 : [E1, F1, C1]
  C1 : [E1, F1]
  B1 : [D1, C1]
Downstream Links:
  E1 : [A1, C1]
  F1 : [A1, C1]
  D1 : [B1]
  C1 : [A1, B1]
```



Consider the following DAG operation

```
dag.add("F1",toSet("G1","B1")); // allowed or not?
```

Demo of Depth First Search to Detect Cycles

```
 1 boolean checkForCycles(Map LINKS, List PATH)
 2   LASTNODE = get last element from PATH
 3   NEIGHBORS = get neighbors of LASTNODE from LINKS
 4
 5   if NEIGHBORS is empty or null then
 6     return false as this path has reached a dead end
 7   otherwise continue
 8   for every NID in NEIGHBORS {
 9     append NID to the end of PATH
10     if the first element in PATH equals NID then
11       return true because PATH now contains a cycle
12     otherwise continue
13     RESULT = checkForCycles(LINKS,PATH) // recursive
14     if RESULT is true then
15       return true because PATH contains a cycle
16     otherwise continue
17     remove the last element from PATH which should be NID
18   }
19   after exploring all NEIGHBORS, no cycles were found so
20   return false
```