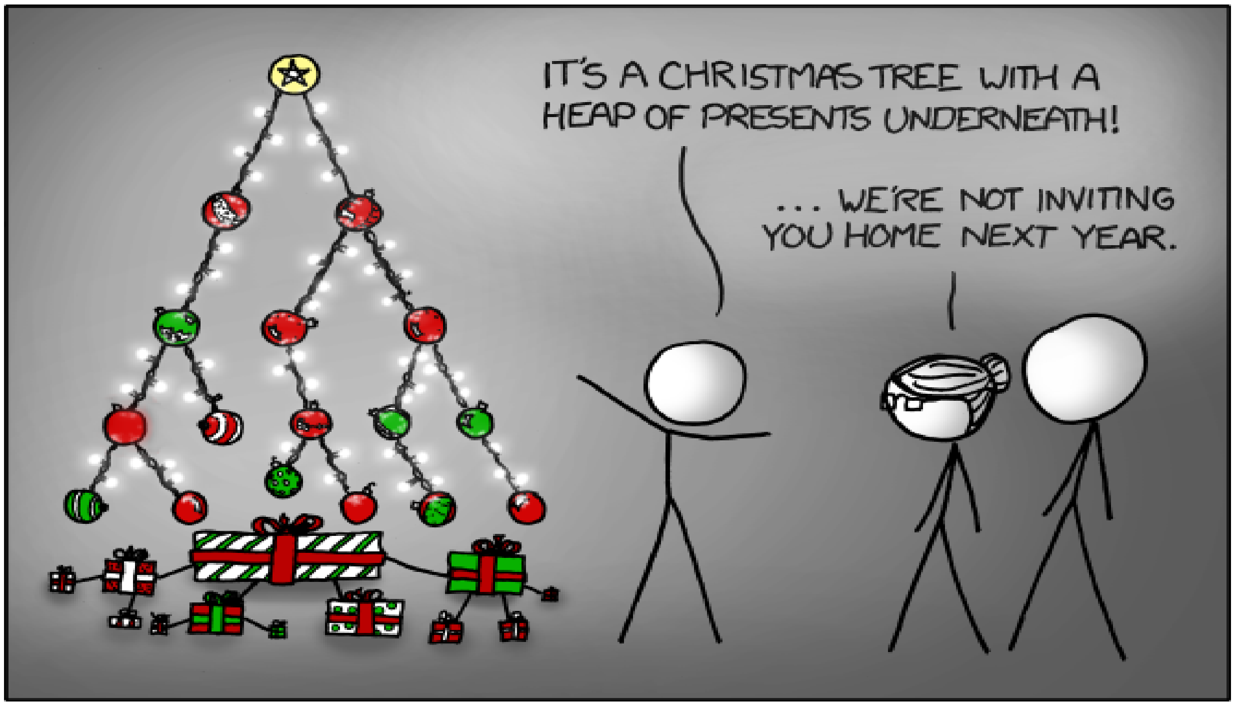
AGRAJAG

A GRAph JArGon

by:

Dongyang Jiang (dj2322)
Zachary Salzbank (zis2102)
Erica Sponsler (es3094)
Nate Weiss (ndw2114)


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

Introduction/Description

Today we live in a social world, where knowing how different things connect to each other is increasingly important. To really take advantage of these connections, a programming language with relational capabilities can be useful. The purpose of our programming language, AGRAJAG, is to simplify the process of creating graphs and trees which can easily reflect connections between multiple objects.

Purpose

The purpose of our language is to simplify the process of connecting different objects together. We do this by creating a new data type called Node which can hold a value and references to objects that are related to it. The Node type can have as many or as few related objects as required. This language will be optimal for representing graphs and trees. Graphs and trees are very important data structures to both computer scientists and mathematicians. Graph theory and combinatorics use these structures to help solve complex problems and increase efficiency in already written programs.

Data Types

- Node<Type> - a node on a graph or tree, containing children and a data value of type <Type>. <Type> can be any data type.
- int - a standard non-floating point number.
- float - a floating point number.
- char - any single standard ASCII character. A string would be a list of chars.
- bool - true or false.

Language Capabilities

Our language will contain the following programming components:

- while
- if/then/else
- basic mathematical functions (+,-,/,*)
- boolean logic (&&, ||, !)
- comparisons (==, <, >, <=, >=, !=)
- null value
- printing capabilities
- function declaration

Example Code

```
Node binSearch ( Node<int> sNode, int searchFor ) {
      while (sNode != null) {
            if (searchFor < sNode.value) {
                  sNode = sNode[0];
            }
```

```
		else if(searchFor > sNode.value) {
			sNode = sNode[1];
		}
		else {
			return sNode;
		}
	}
	return null;
}

Node<int> bfs(Node<int> n, int target)
{
	Node<Node<int>> toVisit = {n, 1};
	endToVisit = toVisit;
	return bfsHelper(n, target, toVisit, endToVisit);
}

Node<int> bfsHelper(Node<int> n, int target, Node<Node<int>> toVisit,
Node<Node<int>> endToVisit)
{
	if (n[value] == target)
	{
		return n;
	}

	while (n[i] != null)
	{
		if(existsInGraph(n[i], toVisit)
		{
			 next = {n[i], 1};
			 endToVisit[0] = next;
			 endToVisit = endToVisit[0];
		}
		i = i + 1;
	}
	return bfsHelper(toVisit, target, toVisit[0], entToVisit);
}

boolean existsInGraph(Node n, Node graph)
{
	int i = 0;
	while (graph[i] != null)
	{
		if (graph[i] == n)
		{
			return true;
		}
		i = i + 1;
	}
```

```
        return false;
    }
```