CMPSCI 187: Programming With Data Structures

Lecture 6: Collections and Abstract Stacks
19 September 2011

Collections and Abstract Stacks

- Catchup: Big-O time for three kinds of searches
- The Collections Idea
- Terminology: Data Types, Abstract Data Types, Data Structures
- The Stack ADT in Java Terms, L&C's `StackADT` Interface
- Using a Stack: Searching for a Path as in Project #2
- Why Does the Search Algorithm Work?

The Collections Idea

- As we said last time, a Collection is a set of objects of a common type.
- Different kinds of Collection will support different operations to add objects, remove objects, or look at particular aspects of the objects.
- In Discussion #1 the objects were containers and we kept them in two Collections, the stack and the buffer.
- The key to defining a kind of Collection is the **interface** between the user and the Collection -- what operations can the user request, and what will be their result, depending on the contents of the Collection?

Terminology: Data Types, ADT's, and Data Structures

- A **data type** is a set of values and accompanying operations, such as the eight primitive types in Java. The type of a variable that represents an object is the set of object classes that could legally be the value of that variable.

- An **abstract data type** or **ADT** is a data type whose values and operations are given not by the language definition but by code. Here "abstract" means that the details of the implementation are hidden from the user.

- An **application programming interface** or **API** is a formal specification of an ADT, with a list of its methods, their signatures, and their intended behaviors.

- A **data structure** is the set of programming constructs used to implement a collection. For example, we will soon use an array data structure to implement the `Stack` ADT.

The Stack ADT in Java Terms

- An implementation of Stack must contain the following methods:

```
public void push (T newElement);
// makes newElement the new top element

public T pop ( );
// returns and removes the top element

public T peek ( );
// returns top element but does not remove it

public boolean isEmpty( );
// returns true if stack has no elements

public int size ( );
// returns number of elements in stack

public String toString ( );
// return a string describing the stack
```

Stack versus StackADT in L&C

- On page 42 L&C give a Java interface called `StackADT` with the methods on the previous slide. As a generic, it defines a type `StackADT<T>` for any possible type T.

- They will give two implementations for this interface -- they will be Java classes that implement the interface: `ArrayStack` and `LinkedStack`.

- Java's `Stack<T>` class also implements this interface, but because it extends another Java class called `Vector` it has additional methods that shouldn't be allowed for a stack. L&C also want to illustrate the separation between ADT and implementation with this data type.

- We'll just use `Stack` in Project #2, though our code should work with any implementation of `StackADT`.

Using a Stack: Searching for a Path (Project #2)

- A **path** in a maze is a sequence of moves from one cell to the next, where each move is one step up, down, right, or left. A path goes from one node (the **source**) to another (the **destination**).

- Given a Maze, a source, and a destination, we'd like to know whether a path exists (using open Cells only) and find a path if it does.

- We'll do this with a `Stack<Cell>` object. Our stack at any time will represent a *trial path* starting at the source. We can add cells to this path at the end, and remove them from the end. If we ever put the destination on the stack, we know we have found a path. We'll take the nodes off the stack and put them in an array.

- If we give up, we will return an array of length 0, or a false boolean.

Depth-First or Backtrack Search

- Here's the idea. Whenever we put a cell on the stack, it will stay there until we have finished with it, meaning that we have looked at and finished all of its open neighbors.
- If we finish the source without finding the destination, then the destination must be unreachable. (Think about it -- we'll give an argument soon.)
- If we find the destination, we've got a path and thus know that a path exists.
- We mark a cell as **seen** if it is on the stack or we have finished with it. There is never a need to revisit a node that has been seen.
- To mark cells as seen we must create a new type `SCell` extending `Cell`.

More on DFS/Backtrack Search

- We call the search **depth-first** because if we arranged all of our visits to nodes in a tree, we explore nodes that are deeper in the tree first -- we make the trial path as long as we can before we finish with any nodes.

- In 250 and 311 you'll also see **breadth-first search**, where we explore all short paths before we look at any longer ones. This is implemented with a queue rather than with a stack.

- This method is called **backtrack search**, especially in AI, because we remove cells from the trial path if we discover that they don't lead to success, and then consider alternative choices to these nodes.

- It's crucial that we can mark nodes that have been processed -- otherwise our search could visit the same node far too many times, or even forever.

Why Does the Search Always Work?

- This is more of a 250 question, but how do we know that this search will always find a path if one exists?
- If there is a path, it has a finite number of cells on it: c0, c1,..., ck where c0 is the source and ck the destination.
- We start by putting c0 on the stack. At some time before we take c0 off, we put c1 on because c1 is a neighbor. We put c2 on before taking c1 off, put c3 on before taking c2 off, and so on, so eventually we put ck on (and declare victory) before taking c0 off.
- In 250 we'll prove this by mathematical induction, but this is a pretty solid argument that we won't ever give up on the search if the path exists.

What Happens if There Isn't a Path?

- Of course if no path exists, we will not find one. But we should satisfy ourselves that we won't get stuck in a loop.
- This is easy to see because there are only finitely many nodes, each node has only finitely many neighbors, and we never look at a node after it is finished.
- In 311 you'll prove that DFS takes time proportional to the number of nodes times the number of neighbors per node.