Strux

Joshua Bartlett
jcb2254
Language Guru

Sophie Stadler
srs2231
Manager

Fredrick Kofi Tam
fkt2105
System Architect

Millie Yang
my2440
Tester

26 September 2017

1 Introduction

Data structures are one of the most important concepts in computer science for beginners and seasoned developers alike. For many students, there is a certain hurdle associated with visualizing data structures—that is, connecting the drawings in a textbook to the Java or C++ they are writing. A major problem with drawings is their static nature; there is no way to see how they are affected by code. Strux hopes to tackle this issue by providing a link between code and data structures in the form of visualizations. We use this term to refer to an "ASCII art" rendering of a stack, queue, linked list, or array that is output by Strux. These visualizations, when called via `visualize(dataStructure)`, are printed to the console to help programmers become familiar with the key features of each structure, and illuminate the data their objects currently contain.

Why printing to the console versus, say, generating an image? The primary reasons are ease of use and efficient visualization of modifications. Users can simply scroll up through the console to see how their stack has changed, rather than sift through a series of images. Strux doesn't require leaving the command line to be useful.

More generally, Strux is an object-oriented language that implements a simplified Java syntax. Additionally, it enforces types, uses the ASCII alphabet, and compiles into LLVM. These characteristics, along with its built in data structures, make it approachable and effective in its goal to increase understanding of data structures.

2 Language Features

2.1 Data Types

Primitives

- `num`: number represented in decimal format
- `string`: an array of ASCII characters, presented in double quotes (")
- `bool`: true or false value

Builtins

- `Array`: fixed-length Java-style array, with 0 indexing. All elements must be of the same type.
- `ListNode`: a node containing a `ListNode` next, and `data` of type `string`, `num`, or `bool`

Data Structures

Stack A class that represents LIFO (last-in-first-out) operations on stack of objects.

Constructors	
`Stack()`	Constructs an empty stack.
`Stack(num[] or string[])`	Constructs a stack containing the array's elements.
Library functions	
`stack.peek()`	Retrieves value of top-most element of stack without removing it from stack. If element does not exist, method returns null.
`stack.pop()`	Retrieves value of top-most element of stack by removing it from stack and returns it. If element does not exist, method returns null.
`stack.push(e)`	Pushes an item `e` to the top of the stack.
`stack.isEmpty()`	Returns boolean variable to indicate whether stack is empty.
`stack.size()`	Returns number of elements in stack. Returns 0 if stack is empty.

Queue A class that represents FIFO (first-in-first-out) operations on stack of objects.

Constructors	
`Queue()`	Constructs an empty queue.
`Queue(num[] or string[])`	Constructs a queue containing the array's elements.
Library functions	
`queue.peek()`	Retrieves but does not remove the head of the queue. Returns null if queue is empty.
`queue.enqueue(e)`	Inserts element `e` into the rear of the queue if it does not violate capacity restrictions.
`queue.dequeue()`	Removes element from the head of the queue. If head does not exist, return null.
`queue.isEmpty()`	Returns boolean variable to indicate whether queue is empty.
`queue.size()`	Returns number of elements in queue. Returns 0 if queue is empty.

LinkedList A `LinkedList` is comprised of `ListNodes`, which contain `data` (either a `num` or `string`), and the next `ListNode`.

Constructors	
`LinkedList()`	Constructs an empty list.
`LinkedList(num[] or string[])`	Constructs a linked list containing the array's elements.
Library functions	
`list.add(e)`	Adds item to tail of list
`list.remove(num or string data)`	Removes and returns list item that contains specified data. If multiple nodes with the same data are present, remove the first node found starting from head.
`list.isEmpty()`	Returns boolean variable to indicate whether list is empty.
`list.size()`	Returns number of elements in list. Returns 0 if list is empty.

Array An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. Two-dimensional arrays will be printed out visually as Matrices by calling `visualize()` on the array.

Constructors	
`num[]e1,e2,e3,...` or `string[]e1,e2,e3,...`	Constructs a num or string array with set size that is determined by the number of elements in the brackets. For example, if there are e1,e2,e3, the size of the array is 3.
Library functions	
`array.length`	Returns the number of items in array
`array.find(x)`	Returns smallest index `i`, where `i` is the first occurrence of `x`.

2.2 Operators

Basic Operators As follows:

`=`	Assignment
`+, -, *, /`	Arithmetic operators. In order: addition, subtraction, multiplication, division.
`%`	Modulo
`++, --`	Increment, decrement
`||, &&, !`	Logical OR, AND, NOT
`<, >, >=, <=, ==, !=`	Traditional value comparators
`[ ]`	Array indexing
`length`	Access array length

Control Flow Control flow mostly follows Java conventions.

- `if (condition)`/`elif (condition)`/`else`: conditional statements
- `for (initialization; termination; increment)`: standard for-loop
- `forEach` *`item`* `in` *`iterable`*: replaces Java's enhanced for loop. Used to iterate over something like elements in an array.
- `while`: standard while-loop
- `break, continue, return`: exit a loop or function.

Function Signature

```
returnType functionName(argType argument) {
    :( function body ):
    return;
}
```

Logging to Console

- `print("your output here")`: prints to console
- `visualize(dataStructure)`: prints data structure visualization to console

Comments Notation for single- and multi-line comments will be consistent. Symbols to signify a commented portion of code will resemble reflective "frowny faces"

```
:( This is a comment. ):
:(
    So
    is
    this.
):
```

2.3 Conventions

- Semicolons occur at the end of a line.
- Indentation (4 spaces) is used for readability, but not enforced by the compiler.
- Braces (`{}`) are required to delimit loops, conditionals, and functions. They are necessary even for single line statements.

3 Sample Programs

3.1 Stack

Program

```
void main() {
    Stack stack = new Stack(new num[]{1, 2, 3});
    visualize(stack);

    stack.push(4);
    visualize(stack);

    stack.pop();
    visualize(stack);
}
```

Output

```
+---+
| 3 | <- Top
+---+
| 2 |
+---+
| 1 |
+---+

+---+
| 4 | <- Top
+---+
| 3 |
+---+
| 2 |
+---+
| 1 |
+---+

+---+
| 3 | <- Top
+---+
| 2 |
+---+
| 1 |
+---+
```

3.2 Queue

Program

```
void main() {
    Queue queue = new Queue(new num[]{4,5,6});
    visualize(queue);

    queue.enqueue(1);
    visualize(queue);

    queue.dequeue();
    visualize(queue);
}
```

Output

```
Head    Tail
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+

Head        Tail
+---+---+---+---+
| 4 | 5 | 6 | 1 |
+---+---+---+---+

Head    Tail
+---+---+---+
| 5 | 6 | 1 |
+---+---+---+
```

3.3 LinkedList

Program

```
void main() {
    LinkedList list = new LinkedList(new num[]{0, 1, 2, 3, 4, 5});

    visualize(list);
    print(list.isEmpty());

    for (num i = 6; i < 10; i++) {
        list.add(i);
    }

    visualize(list);

    list.remove(4);
    visualize(list);

    return list.size();
}
```

Output

```
Head                                Tail
+---+  +---+  +---+  +---+  +---+  +---+  +------+
| 0 |->| 1 |->| 2 |->| 3 |->| 4 |->| 5 |->| null |
+---+  +---+  +---+  +---+  +---+  +---+  +------+

false

Head                                                  Tail
+---+  +---+  +---+  +---+  +---+  +---+  +-----+  +---+  +------+
| 0 |->| 1 |->| 2 |->| 3 |->| 4 |->| 5 |->| ... |->| 9 |->| null |
+---+  +---+  +---+  +---+  +---+  +---+  +-----+  +---+  +------+

Head                                          Tail
+---+  +---+  +---+  +---+  +---+  +-----+  +---+  +------+
| 0 |->| 1 |->| 2 |->| 3 |->| 5 |->| ... |->| 9 |->| null |
+---+  +---+  +---+  +---+  +---+  +-----+  +---+  +------+
```

3.4 Array

Program

```
void main() {
    num[] list = new num[]{0, 1, 2, 3, 4, 5};
    print(list.length);
    visualize(list);
    list[2] = 6;
    visualize(list);
}
```

Output

```
6
[0, 1, 2, 3, 4, 5]
[0, 1, 6, 3, 4, 5]
```