Discrete Structures

Lists and Tuples

Introduction

- Some techniques are a convenient way to store collections of items, and how loops and iteration allow us to sequentially process those items. However, these techniques like arrays are not always the most efficient way to store collections of items. lists may be a better way to store collections of items, and how recursion may be used to process them.
- As we explore the details of storing collections as lists, the advantages and disadvantages of doing so for different situations will become apparent.

Introduction

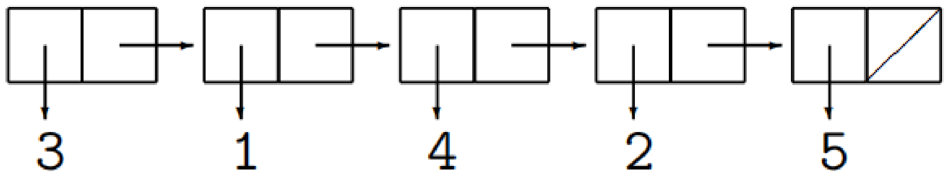
- A list can involve virtually anything, for example, a list of integers [3, 2, 4, 2, 5], a shopping list [apples, butter, bread, cheese], or a list of web pages each containing a picture and a
- link to the next web page. When considering lists, we can speak about-them on different levels on a level on which we can depict lists and communicate as humans about them, on a level on which computers can communicate, or on a machine level in which they can be implemented.

Introduction

- In computer science, a list is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence; the (potentially) infinite analog of a list is a stream.
- If the same value occurs multiple times, each occurrence is considered a distinct item.

Graphical representation of Lists

- Non-empty lists can be represented by two-cells, in each of which the first cell contains a
- pointer to a list element and the second cell contains a pointer to either the empty list or
- another two-cell. We can depict a pointer to the empty list by a diagonal bar or cross through
- the cell. For instance, the list [3, 1, 4, 2, 5] can be represented as:



Lists

- It is obviously also important to be able to get back the elements of a list, and we no longer have an item index to use like we have with an array. The way to proceed is to note that a list is always constructed from the first element and the rest of the list. So, conversely, from a non-empty list it must always be possible to get the first element and the rest. This can be done using the two selectors, also called accessor methods: ***first(list)***, and

rest(list)

Lists

- The selectors will only work for non-empty lists (and give an error or exception on the empty list), so we need a condition which tells us whether a given list is empty:

  isEmpty(list)

This will need to be used to check every list before passing it to a selector.

Lists

- In addition to constructing and getting back the components of lists, one may also wish to destructively change lists. This would be done by so-called mutators which change either the first element or the rest of a non-empty list
- For instance, with l = [3, 1, 4, 2, 5], applying replaceFirst (9, l) changes l to [9, 1, 4, 2, 5].

Storing and updating lists

- In programming, variables give us a way to remember a piece of data and give it a name, so that we can access and even change it later. Oftentimes, that data is a single piece of information, like a number or a string.
- Sometimes that data is a collection of related information, like a listing of high scores or student names. To store collections like those, computer programs can use the list data type, also known as array or sequence.

Initializing a list

- To get started using lists, we need to initialize a variable to store a list.
- In the JavaScript language, we call a list an array and use square brackets to store an array:

```
var myChores = [ ];
```

That list is completely empty, since there are no values inside the brackets.

Initializing a list

- Here's a list that starts off with 5 numbers:

```
var Num = [8, 9, 32, 37, 39];
```

- Notice that we separate each value by a comma.
- We can also store a list of strings, like this example:

```
var rainbowColors = ["red", "orange", "yellow", "green",
"blue", "indigo", "violet"];
```

Accessing list values

- Now that we know how to store a list, we need a way to retrieve each item inside the list.
- The first step is to figure out the index of the desired item. The computer assigns an index to each item of the list, so that it can keep track of where it stored that item in its actual memory.
- the first item in a list is at index 0, not at index 1.

Accessing list values

- Here are the indices for the rainbowColors array:
  - 0 1 2 3 4 5 6
  - "red" "orange" "yellow" "green" "blue" "indigo" "violet"
- Now we can reference any item in the array using "bracket notation":

var firstColor = rainbowColors[0];

var lastColor = rainbowColors[6];

Updating list values

- We can update an item in a list as long as we know its index.
- In JavaScript, we use bracket notation to update a value. For example, let's update the colors of the rainbow based on more modern interpretations.

rainbowColors[4] = "cyan";

rainbowColors[5] = "blue";

Now the rainbowColors array will store "red", "orange","yellow", "green", "cyan", "blue", and "violet".

List operations

- There are many, many ways we can modify a list, besides just updating a singular value, and programming languages often provide built-in procedures for list modification. Let's try a few of them:

List operations

1- ***Appending an item:***

- We often want to append an item to a list, which means adding a new item to the end of it.
- In JavaScript, we can call the push() method on an array, and pass the new item as a parameter.
- Remember my empty myChores array from earlier?

List operations

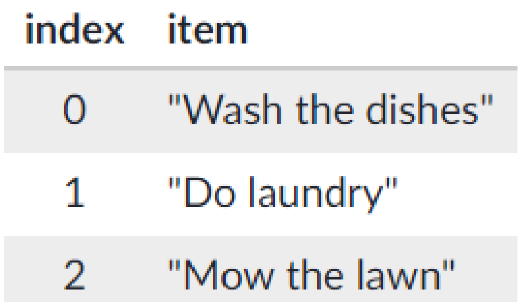
```
myChores.push("Wash the dishes");
```

Now the list stores a single item, "Wash the dishes". We can keep adding items as needed, and the list will get longer each time.

```
myChores.push("Do laundry");
myChores.push("Mow the lawn");
```

List operations

The list now has 3 items:



index	item
0	"Wash the dishes"
1	"Do laundry"
2	"Mow the lawn"

2- Inserting an item

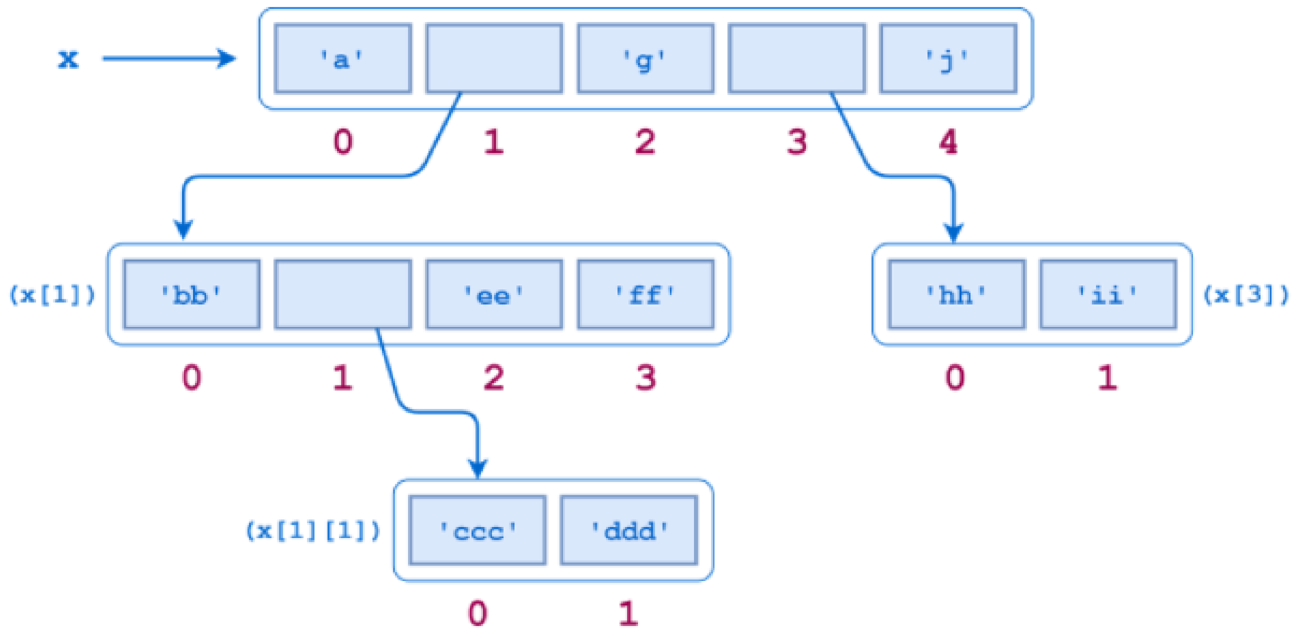
Sometimes we want to add an item earlier in the list, like at the beginning or between two existing items.

Lists Can Be Nested

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

Lists Can Be Nested



Definition

- **Tuples Definition**: When we write doen sentence, it always has a sequential nature. Foe example, in the previous sentence, the word “When” is the first woed, the word “we” is the second word,and so on. Informally, a tuple is a collection of things, caleed ***elements***, where there is a first element, second element and so on.

Definition

- The elements of a tuple also called members, objects or components. For example, the tuple (12,R,9) has three elements. The begining sentence of the paragraph can be represented by the following tuple:
- (When, we, write, down, ... , sequential, nature).

Tuples are immutable

Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
• >>> x = [9, 8, 7]
• >>> x[2] = 6
• >>> print x[9, 8, 6]
• >>> y = 'ABC'
• >>> y[2] = 'D'
• Traceback:'str' object does
• not support item
• Assignment
```

Many methods not work with tuples

- Tuples are more efficient
  - they are simpler and more efficient in terms of memory use and performance than lists
  - So in our program when we are making "temporary variables" we prefer tuples over lists.
  - We can also put a tuple on the left hand side of an assignment statement
  - We can even omit the parenthesis

Tuple Assignments

- `>>> (x, y) = (4, 'fred')`
- `>>> print y`
- `Fred`
- `>>> (a, b) = (99, 98)`
- `>>> print a`
- `99`

Tuples are Comparable

The comparison operators work with tuples. If the first item is equal, then go on to the next element, and so on, until it finds elements that differ.

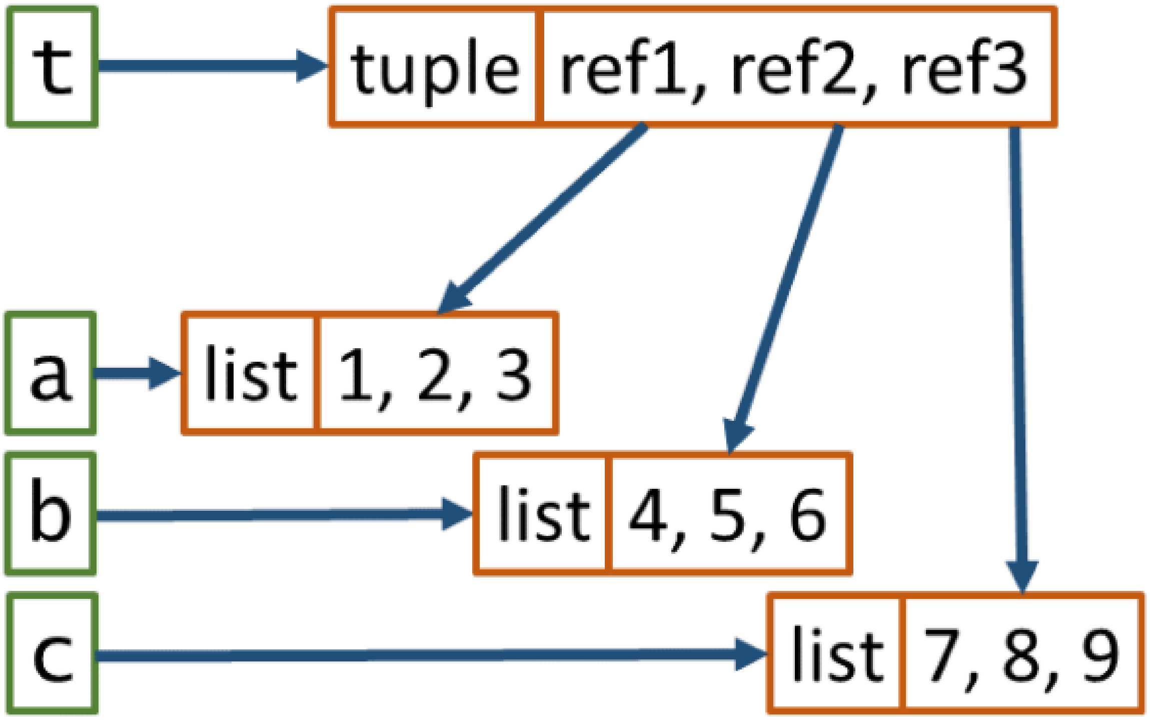
We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary

Lists inside tuples

When we say that a tuple is immutable, it is important to understand exactly what we mean.

As we said, we cannot apply editting on the members or values that the tuple contains, so we can apply edit (inserting or deleting) values through another approach as explained in the diagram below:

Lists inside tuples



differences between list and tuple

List	Tuple
It is mutable	It is immutable
The implication of iterations is time-consuming in the list.	Implications of iterations are much faster in tuples.
Operations like insertion and deletion are better performed.	Elements can be accessed better.
Consumes more memory.	Consumes less memory.
Many built-in methods are available.	Does not have many built-in methods.
Unexpected errors and changes can easily occur in lists.	Unexpected errors and changes rarely occur in tuples.