Overview of List Operations

MARCH 12TH, 2014

Data Structures

- As you study computer science more, you will study lots of different *data structures* (e.g., hash tables, red-black trees, Fibonacci heaps, B-trees, Bloom Filters).
- How data is structured plays a crucial role in the efficiency (both time and space) of your algorithms an programs.
- You should think of lists as your first data structure example.
- Typically data structures support *insert*, *search*, and *delete* operations. And data structures are evaluated by how efficient these operations are.

Organization of the Rest of the Lecture

- First, let us review the most important list operations, categorized into insert, delete, search, and miscellaneous operations.

- Then, let us study the efficiency of some of these operations.

Insert operations on Lists

1. L.append(e)
2. L.insert(i, e)
   - Insert element e at position i of list L. Moves elements originally in positions i..len(L)-1 to the right by one location.
3. L[i:j] = M

   Example:

```
L = range(6)
L
[0, 1, 2, 3, 4, 5]
L[2:4] = [1, 2, 3]
L
[0, 1, 1, 2, 3, 4, 5]
```

Delete operations on Lists

1. L.remove(e)
   - removes the first occurrence of element e from the list L. Elements after e are shifted left one slot.
2. del L[i]
   - removes the element at position *i*. Elements in positions i+1 through *len(L)-1* are moved one slot to the left.
3. del L[i:j]
   - removes the slice of list *L* starting at position *i* and ending at position *j-1*.
4. L[i:j] = M
   - slice assignment can be viewed as deletion if we assign a list *M* smaller than the slice being assigned to.

Search operations on Lists

1. `L[i]`
   - Accessing an element in a list, given its position, can be viewed as a type of search operation. This search operation is very fast and takes *constant* time, i.e., time that is independent of the index `i` and of the length of the list `L`.
2. `L[i:j]`
   - Accessing a list slice.
3. `L.index(e)`
   - returns the index of the first occurrence of element `e` in `L`. Causes an error if `e` is not in `L`.
4. `L.count(e)`
   - returns the number of occurrences of an element `e` in `L`.

Miscellaneous operations on Lists

These function calls return a quantity computed using the list elements.

- sum(L)
- min(L)
- max(L)
- len(L)

These functions reorder the list elements in-place.

- L.sort()
- L.reverse()

Some ways of constructing lists are faster than others…

- Consider this code snippet:

```
L = []
for i in range(100000):
        L.insert(0, i)
```

This constructs a list of one hundred thousand integers:

99999, 99998, 99997, … , 3, 2, 1, 0.

How does this compare in speed to the other ways one can do this in Python?

Other ways of doing the same thing…

```
L = []
for i in range(100000-1, 0, -1):
    L.append(i)
```

```
L = []
for i in range(100000):
    L = [i] + L
```

Here is a puzzle

- When I ran these different ways and measured the running time, here is what I got (in seconds):

  0.031, 5.063, 34.55.

Can you match the running times with the code snippets?

- The medium-speed code is more than 150 times slower than the fastest code. The slowest code is more than 1000 times slower than the fastest code!

Mental model of how lists are implemented

- Suppose we execute `L = [12, 15, 11, 4]`.

- A block of memory is allocated and the items 12, 15, 11, and 4 are stored consecutively at the beginning of this block.

- This allows efficient access to all elements of the list. The location of `L[i]` in memory is simply i + starting location of `L`.

- This guarantees that every element in the list, no matter what its index is, can be accessed equally quickly. This kind of access is called *random access*.

Consequences of this implementation

- append is fast. Consider L.append(e). The length of L is known and hence the location of the first empty slot following L is also known. The element e is stored in this slot.

- Notice that the running time of the append operation is *independent* of the size of L. append takes the same amount of time, no matter how large L is.

- We say that the running time of append is *constant*. (This does not mean that it is the same across different machines.)

Consequences of this implementation

- `insert` and `remove` can be slow because these might cause a large portion of the list to "shift."
- For example, `L.insert(0, e)` causes every element in the list to move one slot. This creates a "hole" at the beginning of the list for element `e`.
- This also means that insert operations towards the end of the list are cheaper than those at the beginning of the list.
- In the *worst case* insert takes time that is proportional to the length of L.
- In other words, insert is said to take *linear* time in the worst case.

Analyzing the code snippets

```
L = []
for i in range(n-1, 0, -1):
    L.append(i)
```

- Assume that append takes time *c*, a constant that has nothing to do with *n*.
- Since the for-loop executes n times, the running time of this code snippet is *c n*.
- Since *c* is a constant this is a linear function in *n*.

Analyzing the code snippets

```
L = []
for i in range(n):
    L.insert(0, i)
```

- After the for-loop has executed *i* times, we have a list of length *i*. We know that `insert` takes time *c i* on this list.

- Therefore the total running time is

  $$c\,(1 + 2 + 3 + \ldots + n\text{-}1) = c\,n\,(n - 1)/2.$$

- Since *c* is a constant this is a *quadratic* function in *n*.

This is the slowest code snippet!

```
L = []
  for i in range(100000):
       L = [i] + L
```

- Whenever the right-hand side is evaluated a new copy of the entire list is made.
- So this code snippet also has quadratic running time.
- However, this slower than the previous code snippet because copying an entire list seems costlier than shifting a list.