Algorithms in Systems Engineering ISE 172

Lecture 5

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Chapter 3

Delving into the Lists

- As we have seen by now, the list data structure is fundamental in Python.
- How is the list data structure implemented?
- How efficient is it?
- What can it be used for?

Example: List Data Type

- Suppose we wanted to design a new data type for storing a list of “objects” similar to the Python list data structure.
- What operations might we want to perform?
  - Create a list.
  - Get/set the value of element j.
  - Delete element j from the list.
  - Add/remove something to the list just before element j.
  - Add/remove an item from the bginning/end of the list.
  - Loop through the elements in sequence.
  - Concatenate two lists.
  - Make a copy of a list.
  - Find/remove an element in the list.
- This data structure is usually implemented in one of two different ways:
  - using an array, or
  - using a linked list.

List Class API (Python

```
class List:
    # creating the list
    def __init__(self):

    # adding
    def insert(self, index, item):
    def append(self, item):
    def extend(self, other):
    def __add__(self, other):

    # deleting
    def remove(self, item):
    def __delitem__(self, index = None):

    # list queries
    def __contains__(self, item):
    def index(self, item):
    def __len__(self):
    def __getitem__(self, index):

    # miscellaneous
    def __iter__(self):
    def __repr__(self):
```

Implementing with Arrays

- In most programming languages, an array is a set of contiguous memory locations in which values can be stored.
- Storing list items in an array allows us to easily find the i^{th} item if we know where the first item is.
- You cannot create an array without a special package Python, but the list class is implemented using arrays in C.

Some Details

- The specific requirements of the API make subtle differences in how we implement the class.
- An important requirement is that we be able to loop through the items in (some) order.
- This means that we can meaningfully refer to an item's position in the list.
- The textbook calls this kind of list an "unordered list," but this is a bit misleading.
- The meaning is that the items do not have an order other than the order determined by the list itself.

A Basic Implementation

- A basic implementation of a list class with arrays would require us to store
  - The underlying array (which may have more slots than necessary)
  - The size of the array
  - The number of elements in the list (could be less than the size)
- For now, we'll assume that the items on the list are stored in the first available positions in the array.
- This storage scheme affects the efficiency of certain operations.

Making an Empty List

- To make an empty list, what do you have to do?
  - Allocate an array of a specified size.
  - How big?
- The best size for the allocated array depends on what will be done with the list.
  - How many items will be added to the list?
  - How much will its size changeover time?
  - Is there a fixed maximum size?

Python's List Data Type

- In Python, all of these questions are hidden from the user, but must be answered when you execute a command such as

```
list = []
```

- Note that even Python itself can and does have different implementations.
- The Python language is also specified by an API of sorts, leaving room for different implementations.

Implementing with Arrays in C++

```
class list {
 private:
   // Here is the implementation-dependent code.
   // We'll store the data in this array.
   int* array;
   // Here is the size of the array.
   int size;
   // Here is the number of items in the list.
   int numItems;
 public:
   list();
   ~list();
   int getNumItems() const;
   bool getValue(const int j, int& value) const;
   bool setValue(const j, const int value);
   bool addItem(const int j, const int value);
   bool delItem(const int j);
}
```

Constructing and Destructing in C++

```
#include "list.h"

list::list() :
   array(new int[MAXSIZE]),
   size(MAXSIZE),
   numItems(0)
{}

list::~list() {
   delete array;
   size = 0;
}
```

Implementing List Query Operations

- Returning the item in the i^{th} position is easy with this implementation.
- Determining whether an item is in the list is time-consuming in general.
- Finding the position of a given item in the list is similarly difficult.
- We don't have much choice but to search through the list linearly.
- This is the nature of an "unordered" list.

Implementing List Query Operations in C++

```
int list::getNumItems() const {
   return numItems;
}

const bool list::getItem(const int j, int& value) {
   if (j > 0 && j < size){
      value = array[j];
      return true;
   }else{
      return false;
   }
}
```

Implementing List Modification Operations

- Appending to a list
  - Generally easy—we just put the item in the last open slot.
  - However, if the array is full, we have to allocate more memory.
- Inserting in the middle of the list requires moving some list items aside (and perhaps also allocating more memory).
- Deleting the item with a specified index from a list also requires moving some elements to close the gap.
- Removing an item whose index is unknown requires first searching the list and then removing the item once found.

Implementing List Modification Operations

```
bool list::addItem(const int j, const int value){
   if (numItems == size || j < 0 || j > size){
      return false;
   }else{
      for (int i = size; i > j; i--)
         array[i] = array[i-1];
      array[j] = value;
      numItems++;
   }
}

bool list::delItem(const int j){
   if (j < 0 || j > size - 1){
      return false;
   }else{
      for (int i = j; i < size - 1; i++)
         array[i] = array[i+1];
      numItems--;
   }
}
```

Implementing with Linked Lists

- For a linked list implementation, we would replace the array with a linked list.
- To the client, the class could function exactly as before, but with a different implementation.
- With a linked list, the items to be stored in the list are stored within separate objects called "nodes".
- The nodes are linked to each other through a variable `next` that tracks which node is next in the list.
- In addition, we must also keep track of which node is the first or "head node".

Linked List Implementation: Node Class

Here is the definition of a node class for a linked list.

```
class Node:
    def __init__(self, initdata, nextNode = None):
        self.data = initdata
        self.nextNode = nextNode
    def getData(self):
        return sefl.data
    def getNext(self):
        return self.next
    def setData(self, newdata):
        self.data = newdata
    def setNext(self, newnext):
        self.next = newnext
```

Linked List Implementation: List Class

- In the list class, we need to store
  - The head node.
  - The number of items on the list.

```
class List:
    def __init__(self, Node = None, length = 0):
        self.head = Node
        self.length = length
    def append(self, item):
        current = self.head
        self.head = Node(item)
        self.head.nextNode = current
        self.length += 1
```

- Note that to make append efficient for the linked list, we must store the list in "reverse" order (the head node is the last item).
- For this reason, it's only efficient to iterate through a singly linked list in reverse order.

Linked List Implementation: Getting an Item

```
def __getitem__ (self, index):
    current = self.head
    if index < 0:
        if -index > self.length:
            raise IndexError
        return self[self.length+index]
    else:
        i = self.length - 1
        while True:
            if current == None:
                raise IndexError
            if i <= index:
                break
            current = current.getNext()
            i -= 1
        return current.getData()
```

Linked List Implementation: Removing Item

```
def remove(self, item):
    current = self.head
    previous = None
    found = False
    while not found and current != None:
        if current.getData() == item:
            found = True
        else:
            previous = current
            current = current.getNext()

    if not found:
        return False
    elif previous == None:
        self.head = current.getNext()
    else:
        previous.setNext(current.getNext())
    self.length -= 1
    return True
```

Linked List Implementation: Removing Index

```
def __delitem__(self, index = None):
    previous = None
    current = self.head
    if index == None:
        current = self.head
        self.head = self.head.getNext()
    else:
        for i in range(self.length - index - 1):
            previous = current
            current = current.getNext()
        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
    self.length -= 1
    return current.data
```

Linked List Implementation: Other Methods

- `insert()`
- `append()`
- `extend()`
- `__add__()`
- `__contains__()`
- `index()`
- `__len__()`
- `peek()`
- `pop()`
- `__iter__()`

Comparing List Implementations

- Consider the two implementations we have just discussed.
  - An *array* is a simple data structure that allows us to store a sequence of numbers.
  - A *linked list* does the same thing.
- What is the difference?

Comparing List Implementations: Theoretical Running Times

- To compare the two data structures, we must analyze the running time of each operation.
- This table compares the running times of the operations.

	Array	Linked List
`insert()`		
`append()`		
`extend()`		
`__add__()`		
`remove()`		
`__delitem__()`		
`index()`		
`__len__()`		
`__getitem__()`		
`__iter__()`		

Comparing List Implementations: Empirical Running Times

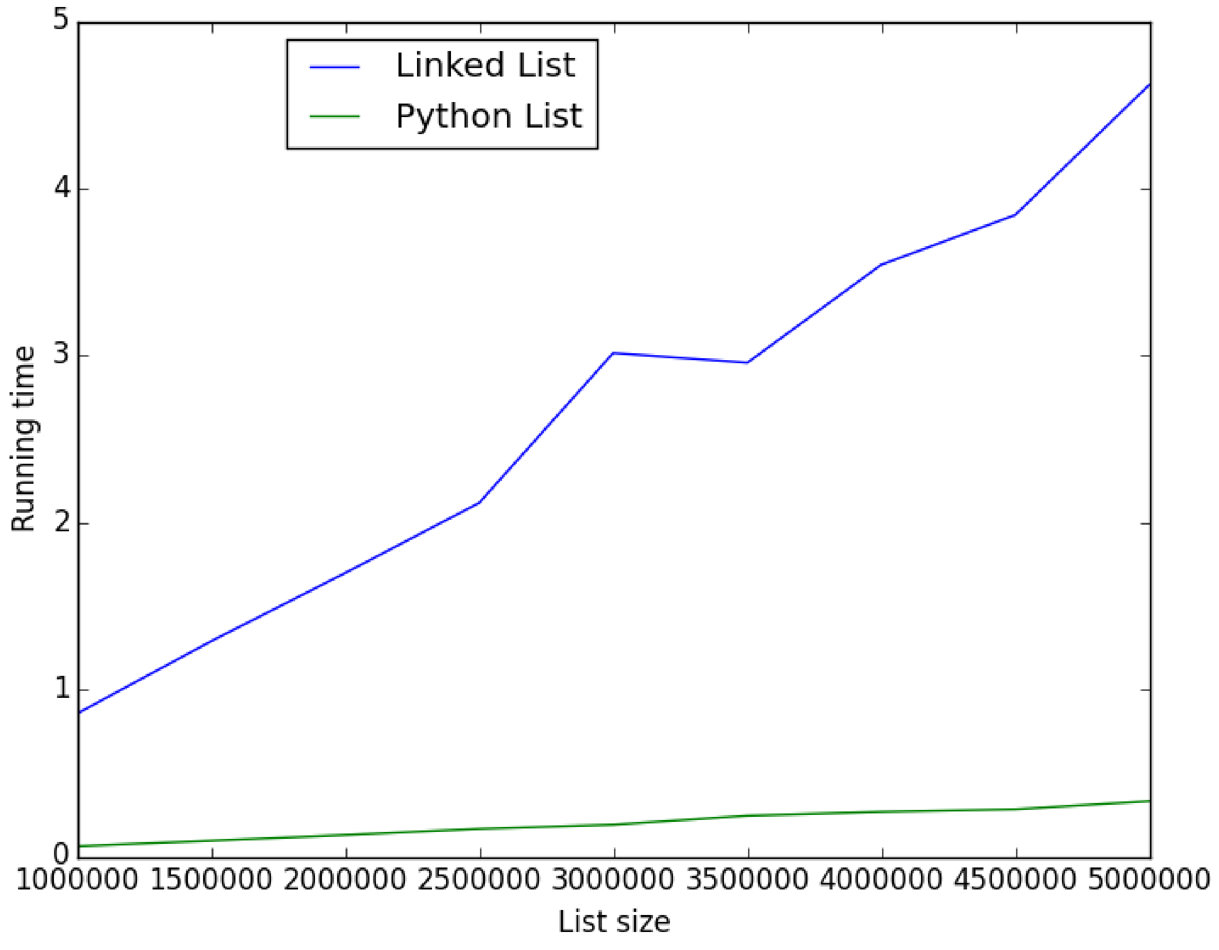

Figure 1: Iterating Over Linked List and Python List

Comparing List Implementations: Empirical Running Times

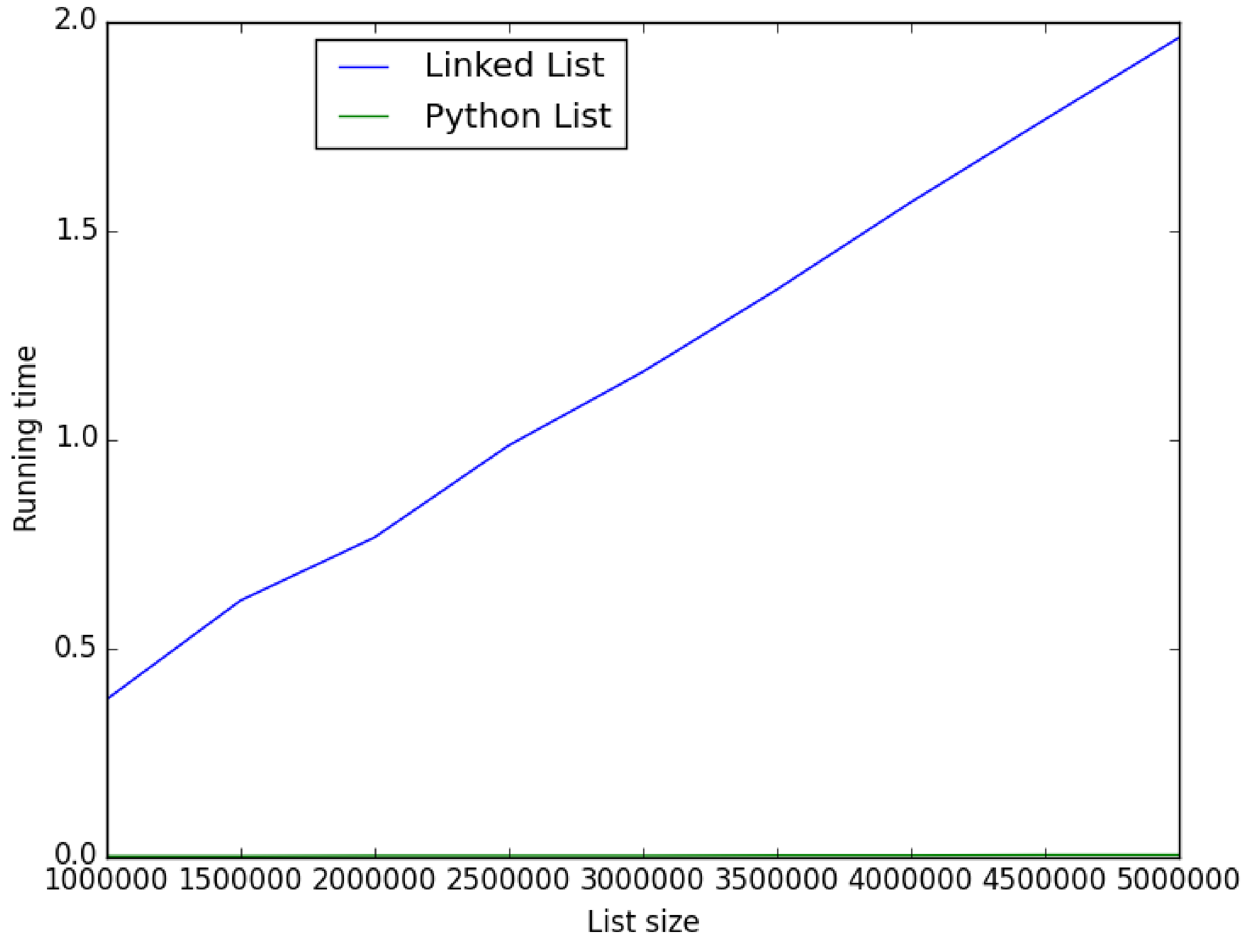

Figure 2: Deleting From Beginning of Python List and Linked List

Comparing List Implementations: Memory Usage

- How do these data structures compare in terms of the amount of memory required?
- It depends...
- The nodes take twice as much memory as an entry in an array.
- However, we only need to have exactly the number of nodes that we have list items with a linked list.
- With an array, we generally need to have more slots available than there are items.
- In the end, the choice depends on what we expect to do with the list in a particular application.

Variations

- Doubly linked list
- Circular list
- Ordered linked list