Computational Optimization
ISE 407

Lecture 15

Dr. Ted Ralphs

Reading for This Lecture

- Horowitz and Sahni, Chapter 2
- Aho, Hopcroft, and Ullman, Chapter 2

What is a Data Structure?

- We will define data structures to be schemes for organizing and storing sets, though this is a slightly limiting definition.
- Examples of set operations.
  - `add`
  - `delete`
  - `find`
  - `union`
  - `sort`
- We also want to be able to efficiently enumerate the items in a set.

Choosing the Right Data Structure

- Data structures consist of
  - a scheme for storing the set(s), and
  - algorithms for performing the desired operations
- Hence, each set operation has an associated complexity
- To choose a data structure, you should know
  - something about the elements of the set, and
  - what operations you will want to perform on the set.

Data Structures and Algorithms

- Typically, data structures are part of a larger algorithm.
- In order to choose a data structure, you should also know something about the algorithm.
- The data structure should be efficient for the operations that will be performed most often.
- The same algorithm can have different running times using different data structures.
- Alternatively, the same data structures can perform differently in different algorithms.

Data Structures and Data Types

- A *data structure* is an abstraction typically specified independent of any particular programming environment.
  - We analyze data structures in the context of a particular model of computation, just as we do algorithms.
  - The term is typically used to indicate a complete scheme, including implementation details.
- A *data type* is the analogue of a data structure in the context of a particular computing system.
- However, in object-oriented languages, a data type is typically defined independent of the implementation.

Object Oriented Programming

- Object-oriented programming is a paradigm that empasizes
  - Data rather than methods
  - Code reuse
  - Separation of interface from implementation
- Defining new data types is the mechanism by which object-oriented languages incorporate *data structures* into the language.
- Defining a new data type requires both
  - an Application Program Interface (API): the set of data *values* that are to be stored and a set of supported *operations* on those values and
  - one or more mplementation of the API: programmatically enabling the capabilities laid out in the API.
- *Classes* and *Inheritance* allow us to separate the interface from the implementation.
- It also allows us to define hierarchies of data types.
- This is useful in allowing concrete types to be interchangeable in well-defined ways, even when not entirely compatible.

Classes

- Classes are the mechanisms by which new data types can be defined.
- A class is composed of
  - Data are the values to be stored (data members in C++; data attributes in Python).
  - Functions are the operations to be performed on the data (methods in C++; method attributes in Python).
- There are also constructors and destructors by which objects of the new type can be created and destroyed.
- Ideally, the definition of the class is independent from the implementation.
  - The *definition* (API) specifies what the data values are and what operations we would like to perform on them.
  - The API informs the user of the class how to use it within another program.
  - The *implementation* specifies the algorithms used to perform those operations and is hidden from the user.
- Julia's structs are a simplified version of classes.

Inheritance

- Inheritance is a mechanism by which we can either
  - Define a grouping of data types with a common (sub-)API (a data type might belong to more than one grouping).
  - Define an API for which it is expected there will be multiple alternative implementations.
- A *base class* is a class defined for the above purposes that may be incomplete.
  - Classes *derived* from the base class inherit its structure and may complete/extend it.
  - A base class is *abstract/virtual* if the implementation of the API is missing or incomplete.
- In C++, functions in a base class may be *virtual*, which means they can be re-implemented in a derived class.
- In Python, any method can be re-implemented.
- Julia has a similar notion of type hierarchies, but it is implemented using multiple dispatch and structs.

Classes in C++

- In C++, the definition is contained in a *header file* that must be included in any source file that uses the data type.
- Members can be either public or private.
  - The interface consists of the public members of the class.
  - The private members of the class along with the function implementations are the implementation.
- It is good programming style to keep all data members private.
- Data members define how the data is stored, which is implementation-dependent
- Access to data values can be provided through query methods.
- This allows changing the implementation without affecting clients.
- Defining operators, such as `+` and `[]` within the classes can allow new data types to work with in-built operators.

Classes in Python

- In Python, the class mechanism is much simpler.
- Methods implemented in a single file are part of an implicitly defined *module* that is like a class with no data.
- Modules can also be grouped together into *packages*.
- There is a proper class mechanism, but all attributes of a class are public.
- If necessary, attributes that are meant to be "private" can be given names that are affixed with an "_".
- It is possible for the interface to be separated from the implementation, but this is not usually done in Python.
- There are also *initializers* and other "magic methods" that allow new data types to behave as expected with in-built operators.

Multiple Dispatch in Julia

- Julia does not have classes in the same sense that C++ and Python do,
- Rather, Julia only has structs (like those in C).
- The methods that operate on those structs are independent and simply take the struct as an argument.
- This is actually almost identical to Python in which class methods are simply functions that take a class object as the first argument.
- The difference is only in the high-level syntax.
- In Julia, standard operations can be extended to work with new data types using multiple dispatch.

Abstract Base Classes

- In Python and in the STL of C++, we group the data types according to the APIs they support by defining *abstract base classes*.
- In C++, an abstract base class is a base class with *pure virtual* functions.
  - Virtual functions may or may not be implemented in the base class.
  - A pure virtual function is one that has no definition and thus prevents the class from being instantiated.
  - In order to be used, a derived class must define all of the functions from the base class that are pure virtual.
- Python has a concept called *virtual base class*, but the philosophy and usage are different than in C++.
- Julia's type system also has abstract types, but there is no mechanism for defining an interface independent of implementation.

Built-in Data Tyes

- Python has a number of very useful data types built into the language, but it is easy to obtain new ones by installing packages.
- In C++, the *Standard Template Library* or *STL* is a library of commonly used data types and algorithms.
- Technically, the STL is not part of the language.
- Both Python built-in data structures and the STL are highly optimized.
- Julia has a collection of built-ins that is a bit broader than C/C++, but in Julia, it is also easy to etend the language with packages.

Python's Collections ABCs

ABC	Inherits from	Abstract Methods	Mixin Methods
`Container`		`__contains__`	
`Hashable`		`__hash__`	
`Iterable`		`__iter__`	
`Iterator`	`Iterable`	`next`	`__iter__`
`Sized`		`__len__`	
`Callable`		`__call__`	
`Sequence`	`Sized`, `Iterable`, `Container`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
`MutableSequence`	`Sequence`	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	Inherited `Sequence` methods and `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`
`Set`	`Sized`, `Iterable`, `Container`	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
`MutableSet`	`Set`	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	Inherited `Set` methods and `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`, and `__isub__`
`Mapping`	`Sized`, `Iterable`, `Container`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
`MutableMapping`	`Mapping`	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	Inherited `Mapping` methods and `pop`, `popitem`, `clear`, `update`, and `setdefault`
`MappingView`	`Sized`		`__len__`
`ItemsView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`KeysView`	`MappingView`, `Set`		`__contains__`, `__iter__`
`ValuesView`	`MappingView`		`__contains__`, `__iter__`

Figure 1: Source: http://docs.python.org/2/library/collections.html

Python's Built-in Types and Other ABCs

- Numeric
  - Real: `float`, `long`, `complex`
  - Integral: `int`
- Sequence
  - String: `str`, `unicode`
  - Immutable: `tuple`, `xrange`
  - Mutable: `list`, `bytearray`
- Set: `set`, `frozenset`
- Map: `dict`
- File
- Memoryview
- Context Manager

Delving into Lists

- The "list" data type is fundamental in all programming languages.
- As opposed to the array data type in C, which has a fixed length, the list data type is meant to support dynamic re-sizing.
- Such a data structure might also be called a "dynamic array."
  - Python provides a built-in list data type with a full-featured API.
  - C++ provides something similar in the `vector` class of the *standard template library* (STL)
  - Julia provides the `Array`, which can function either as a C-style array or a Python-style list in different circumstances.
- How is the list data type implemented?
- How efficient is it in each language?
- 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 type.
- 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 type is usually implemented in one of two different ways:
  - using an array, or
  - using a linked list.

A List Class in Python

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

    # adding items
    def insert(self, pos, item)
    def append(self, item)
    def extend(self, list)

    # deleting items
    def remove(self, item)
    def pop(self, pos)

    # list queries
    def __contains__(self, item)
    def index(self, item)
    def peek(self, pos)
    def __len__(self)
    def __getitem__(self, index)
```

A List Class in C++

```
class list {
 private:
   // Here is the implementation-dependent code
   // that defines exactly how the list is stored.
 public:
   // Here is the list of operations to be implemented.
   // Create and destroy a list
   list();
   ~list();
   // Get the number of items in the list
   int getNumItems() const;
   // Get the value of item j
   bool getItem(const int j, int& value) const;
   // Change the value of item j
   bool setItem(const int j, const int value);
   // Add an item before item j
   bool addItem(const int j, const int value);
   // Delete item j
   bool delItem(const int j);
}
```

In practice, `getItem` would be implemented by defining the `[]` operator.

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 in Python, but the Python list class is implemented using the arrays provided 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 order.
- This means that we can meaningfully refer to an item's position in the list.
- The list is not really "ordered," the ordering is determined by how the items are added to the list originally.
- Of course, a given implementation may support the sorting of the list.

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.

Implementing with Arrays in C++

This source would be put in a file called `list.h`.

```
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 getItem(const int j, int& value) const;
   bool setItem(const j, const int value);
   bool addItem(const int j, const int value);
   bool delItem(const int j);
}
```

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?

Constructing and Destructing in C++

This source would be put in a file called `list.cpp`.

```
#include "list.h"

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

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

Python's List Data Type

- With the Python built-in list data type, the details are hidden from the user, but construction occurs upon executing the command

```
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 List Query Operations

- Returning the item in the $i^{\mathrm{t}h}$ 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 in Python

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

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

Linked List Implementation: List Class in Python

- 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 we append to the "beginning" of a linked list and the "end" of an array.
- For this reason, it's only efficient to iterate through a singly linked list backward.

Linked List Implementation: Search

```
def __contains__(self, item):
    current = self.head
    while current != None:
        if current.getData() == item:
            return True
        else:
            current = current.getNext()
    return False
```

Linked List Implementation: Removing

```
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: Other Methods

- `__len__()`
- `insert()`
- `peek()`
- `pop()`
- `extend()`
- `index()`

Comparing List Implementations

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

Comparing List Implementations: Efficiency

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

	Array	Linked List
`length()`		
`insert()`		
`peek()`		
`pop()`		
`extend()`		
`index()`		
`append()`		
`remove()`		

Comparing List Implementations: Memory Usage

- How do these data types 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

Using lists

- Insertion sort
- Merge sort/quick sort
- Binary search
- Circular lists
- Doubly linked lists

Stacks

- A *stack* is a special kind of list in which items can only be removed in "last-in, first-out" (LIFO) order.
- The basic operations on a stack are
  - `push()`: Put a new item on the stack.
  - `pop()`: Take the most recently added item off the stack.
  - `peek()`: Get a copy of the most recently added item.
  - `isEmpty()`: Determine whether the stack is empty.
  - `remove()`: Remove a particular item from the stack.
- Stack data structures
  - Array
  - Linked list
- In Python, the `list` API includes the methods to support its use as a stack.

Queues

- A *queue* is a list in which items can only be removed in "first-in, first-out" (FIFO) order.
- The basic operations on a queue are
  - `enqueue()`: put a new item in the queue.
  - `dequeue()`: remove the most recently added item from the queue.
  - `peek()`: Get a copy of the most recently added item.
  - `isEmpty()`: Determine whether the stack is empty.
  - `remove()`: Remove a particular item from the stack.
- Queue data structures
  - Array
  - Circular array
  - Linked list