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 compete scheme, including implementation details.

• A *data type* is the analogue of a data structure in the context of a particular computing system.
  – In object-oriented languages, a data type is typically defined independent of the implementation.
  – *Inheritance* allows us to separate the definitions 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.
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.

- An 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.
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++.
## Python’s Collections ABCs

<table>
<thead>
<tr>
<th>ABC</th>
<th>Inherits from</th>
<th>Abstract Methods</th>
<th>Mixin Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td></td>
<td><strong>contains</strong></td>
<td></td>
</tr>
<tr>
<td>Hashable</td>
<td></td>
<td><strong>hash</strong></td>
<td></td>
</tr>
<tr>
<td>Iterable</td>
<td></td>
<td><strong>iter</strong></td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Iterable</td>
<td>next</td>
<td><strong>iter</strong></td>
</tr>
<tr>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
<td></td>
</tr>
<tr>
<td>Callable</td>
<td></td>
<td><strong>call</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence</td>
<td>Sized, Iterable, Container</td>
<td><strong>getitem</strong>, <strong>len</strong></td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>reversed</strong>, index, and count</td>
</tr>
<tr>
<td>MutableSequence</td>
<td>Sequence</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>len</strong>, insert</td>
<td>Inherited Sequence methods and append, reverse, extend, pop, remove, and <strong>iadd</strong></td>
</tr>
<tr>
<td>Set</td>
<td>Sized, Iterable, Container</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td><strong>le</strong>, <strong>lt</strong>, <strong>eq</strong>, <strong>ne</strong>, <strong>gt</strong>, <strong>ge</strong>, <strong>and</strong>, <strong>or</strong>, <strong>sub</strong>, <strong>xor</strong>, and <strong>isub</strong></td>
</tr>
<tr>
<td>MutableSet</td>
<td>Set</td>
<td><strong>contains</strong>, <strong>iter</strong>, <strong>len</strong>, add, discard</td>
<td>Inherited Set methods and clear, pop, remove, <strong>ior</strong>, <strong>iand</strong>, <strong>ixor</strong>, and <strong>isub</strong></td>
</tr>
<tr>
<td>Mapping</td>
<td>Sized, Iterable, Container</td>
<td><strong>getitem</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td><strong>contains</strong>, keys, items, values, get, <strong>eq</strong>, and <strong>ne</strong></td>
</tr>
<tr>
<td>MutableMapping</td>
<td>Mapping</td>
<td><strong>getitem</strong>, <strong>setitem</strong>, <strong>delitem</strong>, <strong>iter</strong>, <strong>len</strong></td>
<td>Inherited Mapping methods and pop, popitem, clear, update, and setdefault</td>
</tr>
<tr>
<td>MappingView</td>
<td>Sized</td>
<td></td>
<td><strong>len</strong></td>
</tr>
<tr>
<td>ItemsView</td>
<td>MappingView, Set</td>
<td></td>
<td><strong>contains</strong>, <strong>iter</strong></td>
</tr>
<tr>
<td>KeysView</td>
<td>MappingView, Set</td>
<td></td>
<td><strong>contains</strong>, <strong>iter</strong></td>
</tr>
<tr>
<td>ValuesView</td>
<td>MappingView</td>
<td></td>
<td><strong>contains</strong>, <strong>iter</strong></td>
</tr>
</tbody>
</table>

**Figure 1:** Source: [http://docs.python.org/2/library/collections.html](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
Example: Back to Lists

• A list is a data structure for storing a finite sequence of elements.

• The items in a list have an order that is imposed by the sequence in which they are given, but are otherwise unordered.

• Example List operations
  – Create a list.
  – Get the number of items.
  – Get the value of the $j^{th}$ item in the sequence.
  – Set the value of the $j^{th}$ item in the sequence.
  – Add something to the list before the $j^{th}$ item in the sequence.
  – Delete $j^{th}$ item in the sequence from the list.
  – Enumerate the items in sequence.

• As we have seen, lists have two basic implementation schemes.
  – Arrays
  – Linked Lists
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.
Aside: Stack Frames

- Local data for each function call is stored on the stack
- Each function gets a stack frame within which to store data
  - space for local variables
  - pointers to the parameters the function was called with
  - pointers to the instruction to return to in the calling function
  - pointer to the location to store the return value
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
Symbol Tables and Dictionaries

• We have now discussed in detail the ADT for a list and have seen two implementations, as well as other data structures built on lists.

• We now consider a different kind of list data structure that supports different kinds of operations.

• A symbol table is a data structure for storing a list of items, each with a key and satellite data.

• This data structure supports the following basic operations.
  – Construct a symbol table.
  – Search for an item (or items) having a specified key.
  – Insert an item.
  – Remove a specified item.
  – Count the number of items.
  – Print the list of items.

• Symbol tables are also called dictionaries because of the obvious comparison with looking up entries in a dictionary.

• Note that the keys may not have an ordering.
Additional Operations on Symbol Tables

• If the items can be ordered, e.g., by `__lt__` and `__eq__`, we may support the following additional operations.

  – Sort the items (print them in sorted order).
  – Return the maximum or minimum item.
  – Select the $k^{th}$ item.
  – Return the successor or predecessor of a given item.

• We may also want to be able to join two symbol tables into one.

• These operations may or may not be supported in various implementations.
Applications of Symbol Tables

- What are some applications of symbol tables?
Dictionary ADT

class Dictionary:
    def __init__():
    def getNumItems():
    def __contains__(key):
    def get(k):
    def append(key, value):
    def remove(key):
    def keys():
    def values():
    def select(key):
    def sort():
Naive Implementation

• Consider a list of items whose keys are small positive integers.

• Assuming no duplicate keys, we can implement such a symbol table using an unordered list.

class Dictionary:
    def __init__(self, maxKey):
        self.list = [None for i in range(maxKey)]
        self.maxKey = maxKey
    def append(self, key, value):
        self.list[key] = value
    def remove(key):
        self.list[key] = None
    def __contains__(key):
        if self.list[key] = None:
            return False
        else:
            return True
Naive Implementation (cont.)

def select(self, key):
    for i in self.list:
        if i != None:
            key--
            if key == 0:
                return self.list[i]

def sort(self):
    pass

def getNumItems(self):
    count = 0
    for i in self.list:
        if i != None:
            count++
    return count
Arbitrary Keys

- Note that with arrays, most operations are constant time.
- What if the keys are not integers or have arbitrary value?
- We could still use an array or a linear linked list to store the items.
- However, some of the operations would become inefficient.
- Recall Lab 2
  - If we keep the items in order, searching would be efficient (binary search), but inserting would be inefficient.
  - If we kept the items unordered, inserting would be efficient, but searching would be inefficient (sequential search).
- A hash table is a more efficient data structure for implementing symbol tables where the keys are an arbitrary data type.
- Hash tables are the subject of the next lecture.