CSC 172– Data Structures and Algorithms

Lecture #7

Spring 2018

Abstract Data Type

ADT

Outline

- Accessing the data
- Data types
- Abstract data types
- Collection
- List
- ArrayList and LinkedList

Accessing the data

- Any form of information processing or communication requires that data must be stored in and accessed from either main or secondary memory

- There are two questions we should ask:
  - What do we want to do?
  - How can we do it?

- This topic will cover Abstract Data Types:
  - Models of the storage and access of information

- The next topic will cover data structures and algorithms:
  - The concrete methods for organizing and accessing data in the computer

Data Type

- A data type is a type together with a collection of operations to manipulate the type.
  - Example:
    - An integer variable is a member of the integer data type.
    - Addition is an example of an operation on the integer data type.

Logical vs Physical

- A distinction should be made between:
  - The logical concept of a data type and its physical implementation in a computer program.
  - For example:
    - Two traditional implementations for the list data type: the linked list and the array-based list.

- The list data type can therefore be implemented using a linked list or an array.

Arrays

- The most primitive data structure
- "Array" is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item.
- By this meaning, an array is a physical data structure.

Abstract data type (ADT)

- An abstract data type (ADT) is the realization of a data type as a software component.
  - The interface of the ADT is defined in terms of
    - A type and
    - A set of operations on that type.
  - The behavior of each operation is determined by its inputs and outputs.
  - An ADT does not specify how the data type is implemented.
  - Encapsulation:
    - These implementation details are hidden from the user of the ADT and protected from outside access

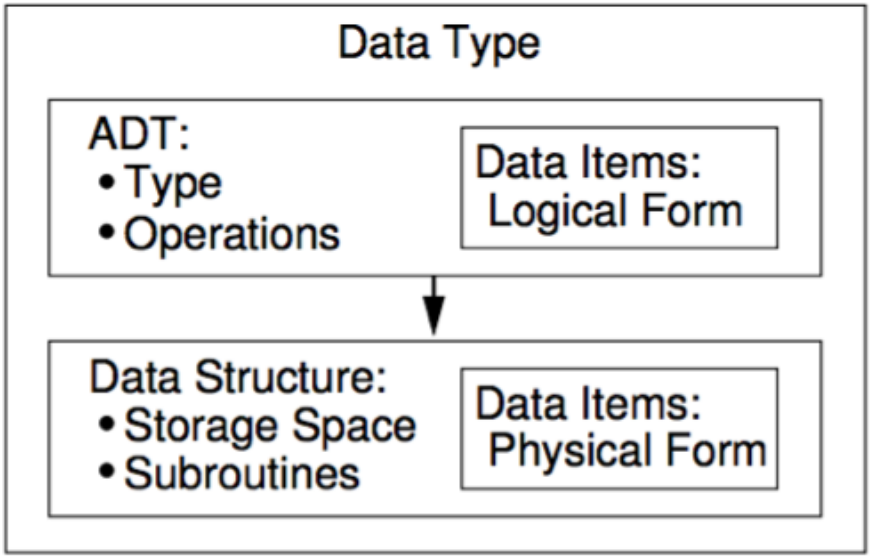
Data structure

- A data structure is the implementation for an ADT.
  - In Java, an ADT and its implementation together make up a class.
  - Each operation associated with the ADT is implemented by a member function or method.
  - The variables that define the space required by a data item are referred to as data members.
  - An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

Data structure often refers to data stored in a computer's main memory.

File structure often refers to the organization of data on secondary memory

ADT vs Data Structures



The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

COLLECTION AND LIST

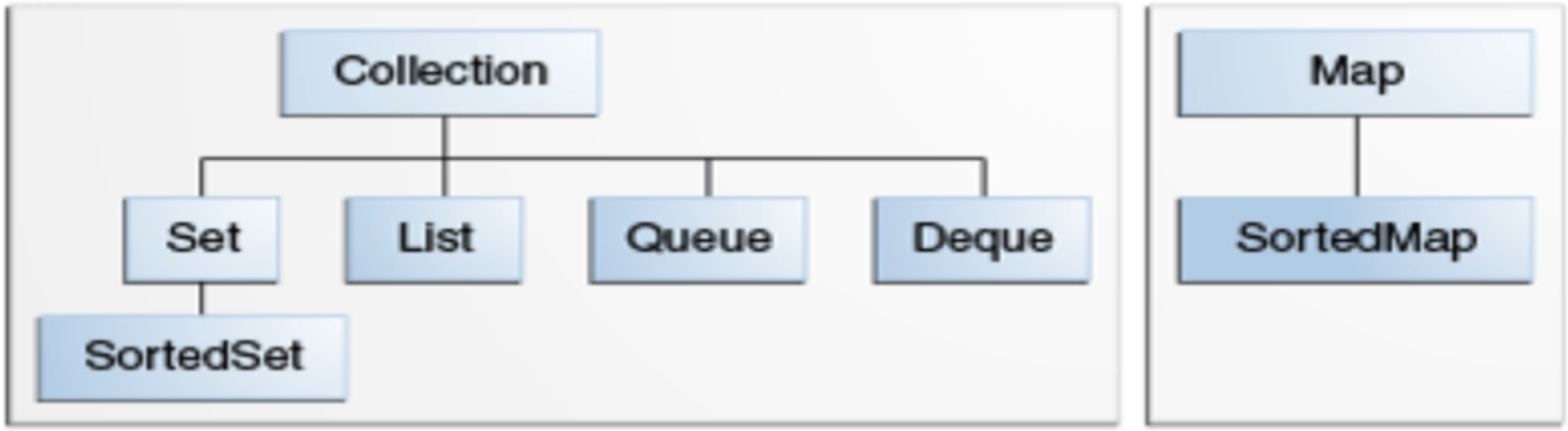
Collection

2.1.1

The most general Abstract Data Type (ADT) is that of a *collection*

A collection describes structures that store and give access to objects

The core collection interfaces



Note that the hierarchy consists of two distinct trees — a Map is not a true Collection.

https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Collection<E>

- public interface **Collection<E>** extends Iterable<E>

The root interface in the *collection hierarchy*.
A collection represents a group of objects, known as its *elements*.
Some collections allow duplicate elements and others do not.
Some are ordered and others unordered.

Iterable<T>

- public interface **Iterable<T>**

Modifier and Type	Method and Description
`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type T.

Implementing this interface allows an object to be the target of the "foreach" statement.

Review Lab 3 and learn how you can iterate an ArrayList using iterator.

public interface **Iterator<E>**

Modifier and Type	Method and Description
boolean	**hasNext()** Returns `true` if the iteration has more elements.
E	**next()** Returns the next element in the iteration.
void	**remove()** Removes from the underlying collection the last element returned by this iterator (optional operation).

Interface Collection<E>

Modifier and Type	Method and Description
boolean	**add(E e)** Ensures that this collection contains the specified element (optional operation).
boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns `true` if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.
int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns `true` if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size()** Returns the number of elements in this collection.
Object[]	**toArray()** Returns an array containing all of the elements in this collection.
<T> T[]	**toArray(T[] a)** Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

List<E>

- public interface **List<E>** extends Collection<E>

List<E> Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.

List<E> Methods (continued)

int	**indexOf(Object o)** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty()** Returns `true` if this list contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf(Object o)** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator()** Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator(int index)** Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove(int index)** Removes the element at the specified position in this list (optional operation).
boolean	**remove(Object o)** Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll(Collection<?> c)** Retains only the elements in this list that are contained in the specified collection (optional operation).
E	**set(int index, E element)** Replaces the element at the specified position in this list with the specified element (optional operation).
int	**size()** Returns the number of elements in this list.
List<E>	**subList(int fromIndex, int toIndex)** Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
Object[]	**toArray()** Returns an array containing all of the elements in this list in proper sequence (from first to last element).
<T> T[]	**toArray(T[] a)** Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

ArrayList and LinkedList

- The Java platform contains two general-purpose List implementations
- Note: ArrayList and LinkedList are both classes, not interfaces
- ArrayList, which is usually the better-performing implementation, and
- LinkedList which offers better performance under certain circumstances.
-

class ArrayList<E>

- public class **ArrayList<E>**
- extends AbstractList<E>
- implements
- List<E>,
- RandomAccess,
- Cloneable,
- Serializable

class LinkedList<E>

- public class **LinkedList<E>**
- extends AbstractSequentialList<E>
- implements
- List<E>,
- Deque<E>,
- Cloneable,
- Serializable

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.