Dynamic Data Structures

Outline

- Introduction
- Linked Lists
- **ArrayList** Class
- Generics
- **LinkedList** Class
- **Collections** Class
- **Stack** Class
- **PriorityQueue** Class
- **HashSet** Class
- References
Introduction
Introduction

- **Data Structure**
  - A data structure is a collection of data organized in some fashion.
  - A data structure not only stores data, but also supports the operations for accessing and manipulating data in the structure.

- **Types of data structures:**
  - **Fixed-size data structures**
    - such as one-dimensional and multidimensional arrays.
  - **Dynamic data structures**
    - that grow and shrink at execution time.
Arrays

- An array is a data structure that holds a collection of data in sequential order.
- You can find the size of the array, and store, retrieve, and modify data in the array.
- Arrays are simple and easy to use, but they have two limitations:
  - (1) once an array is created, its size cannot be altered;
  - (2) an array does not provide adequate support for insertion and deletion operations.
Classic dynamic data structures:
- Linked Lists
- Stacks
- Queues
- Trees
Dynamic Data Structures

Classic Dynamic Data Structures

- **Linked lists**
  - are collections of data items "linked up in a chain; insertions and deletions can be made anywhere in a linked list.

- **Stacks**
  - are important in compilers and operating systems; insertions and deletions are made only at one end of a stacks top.
Dynamic Data Structures

Classic Dynamic Data Structures

- **Queues**
  - represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and deletions are made from the front (also referred to as the head).

- **Trees**
  - is a data structure that supports searching, sorting, inserting, and deleting data efficiently.
In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements.

Some people refer to data structures as container objects or collection objects.

To define a data structure is essentially to declare a class.

The class for a data structure should use data fields to store data and provide methods to support such operations as insertion and deletion.
To create a data structure is therefore to create an instance from the class.

You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into the data structure or deleting an element from the data structure.
Introduction

- Java Collections Framework
  - Contain prepackaged data structures, interfaces, algorithms for manipulating those data structures
  - With collections, programmers use existing data structures, without concern for how they are implemented.
  - This is a example of code reuse.
  - Programmers can code faster and can expect excellent performance, maximizing execution speed and minimizing memory consumption.
Introduction

- **Collection**
  - Data structure (object) that can hold references to other objects

- **Collections Framework Interfaces**
  - declare operations for various collection types
  - Provide high-performance, high-quality implementations of common data structures
  - Enable software reuse
Collections Framework Interfaces

- **Collection**
  - The root interface in the collections hierarchy from which interfaces Set, Queue and List are derived.

- **List**
  - An ordered collection that can contain duplicate elements.

- **Queue**
  - Typically a first-in, first-out collection that models a waiting line; other orders can be specified.

- **Set**
  - A collection that does not contain duplicates.

- **Map**
  - Associates keys to values and cannot contain duplicate keys.
Set and List are subinterfaces of Collection
## Dynamic Data Structures

### Interfaces and Classes

<table>
<thead>
<tr>
<th>Interface</th>
<th>implemented by class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>ArrayList</td>
<td>a linked list data structure</td>
</tr>
<tr>
<td>List</td>
<td>LinkedList</td>
<td>a linked list data structure</td>
</tr>
<tr>
<td>List</td>
<td>Stack</td>
<td>a stack data structure</td>
</tr>
<tr>
<td>Queue</td>
<td>PriorityQueue</td>
<td>A queue data structure</td>
</tr>
<tr>
<td>Set</td>
<td>HashSet</td>
<td>Stores unique elements in a hash table</td>
</tr>
</tbody>
</table>
ArrayList Class
A list is a popular data structure for storing data in sequential order.

For example, a list of students, a list of available rooms, a list of cities, and a list of books can all be stored using lists.

The operations listed below are typical of most lists:

- Retrieve an element from a list.
- Insert a new element to a list.
- Delete an element from a list.
- Find how many elements are in a list.
- Find whether an element is in a list.
- Find whether a list is empty.
Lists

- **List**
  - A list is can contain duplicate elements
  - Sometimes called a *sequence*
  - List indices are zero based (i.e., the first element's index is zero)
  - Classes
    - *ArrayList*: is resizable-array
    - *LinkedList*: is resizable-array
You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the **ArrayList** class that can be used to store an unlimited number of objects. **ArrayList** is a class of **java.util**. **Autoboxing** occurs when you add a primitive type to a **ArrayList**.
**ArrayList Class**

- **+ArrayList()**
  - Creates an empty list.

- **+add(o: Object) : void**
  - Appends a new element `o` at the end of this list.

- **+add(index: int, o: Object) : void**
  - Adds a new element `o` at the specified index in this list.

- **+clear(): void**
  - Removes all the elements from this list.

- **+contains(o: Object): boolean**
  - Returns true if this list contains the element `o`. 
Array List Class

- `+get(index: int) : Object`
  - Returns the element from this list at the specified index.
- `+indexOf(o: Object) : int`
  - Returns the index of the first matching element in this list.
- `+isEmpty(): boolean`
  - Returns true if this list contains no elements.
- `+lastIndexOf(o: Object) : int`
  - Returns the index of the last matching element in this list.
ArrayList Class

- **+remove(o: Object): boolean**
  - Removes the element `o` from this list.

- **+size(): int**
  - Returns the number of elements in this list.

- **+remove(index: int) : Object**
  - Removes the element at the specified index.

- **+set(index: int, o: Object) : Object**
  - Sets the element at the specified index.
Dynamic Data Structures

ArrayList Class

- Example 1:
  - TestArrayList.java
  - GeometricObject.java
  - Circle.java

- You will get a compilation warning “unchecked operation” Ignore it.
ArrayList Class

- The output:
  - List size? 6
  - Is Toronto in the list? true
  - The location of New York in the list? 1
  - Is the list empty? false
  - London Beijing Paris Hong Kong Singapore
  - The area of the circle? 12.566370614359172
Dynamic Data Structures

ArrayList Class

- Example 2:
  - TestArrayList2.java

- ArrayList:
  - MAGENTA RED WHITE BLUE CYAN

- ArrayList after calling removeColors:
  - MAGENTA CYAN
ArrayList Class

● Example 3:
  - TestArrayList3.java

● The output:
  List is: 7
  List is: 7 11
  List is: 12 7 11
  List is: 12 11
An array can be declared to contain more elements than the number of items expected, but this wastes memory.

Linked lists provide better memory utilization in these situations.

Linked lists allow the program to adapt to storage needs at runtime.
Performance Tip

- Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point).
- All existing node objects remain at their current locations in memory.
Performance Tip

- Insertion and deletion in a sorted array can be time consuming.
- All the elements following the inserted or deleted element must be shifted appropriately.
Performance Tip

- Normally, the elements of an array are contiguous in memory.
- This allows immediate access to any array element, because its address can be calculated directly as its offset from the beginning of the array.
- Linked lists do not afford such immediate access to their elements.
- An element can be accessed only by traversing the list from the front (or from the back in a doubly linked list).
Dynamic Data Structures

Linked Lists

- Linked list graphical representation
Generics
Generics

- It would be nice if we could write a single sort method that could sort the elements in an `Integer` array, a `String` array or an array of any type that supports ordering (i.e., its elements can be compared).

- It would also be nice if we could write a single `Stack` class that could be used as a `Stack` of integers, a `Stack` of floating-point numbers, a `Stack` of Strings or a `Stack` of any other type.
Generics

- It would be even nicer if we could detect type mismatches at compile time known as compile-time type safety.
- For example, if a Stack stores only integers, attempting to push a String on to that Stack should issue a compile-time error.
- Generics provides the means to create the general models mentioned above.
Generics

- Provide compile-time type safety
  - Catch invalid types at compile time
- Generic methods
  - A single method declaration
- Generic classes
  - A single class declaration
Generics

- Note that ArrayList is a generic class, so we are able to specify a type argument (String in this case) to indicate the type of the elements in each list.

- Example:
  - TestArrayList4.java
LinkedList Class
Linked List Class

- **LinkedLists** can be used to create stacks, queues, trees and deques (double-ended queues, pronounced “decks”).
- The collections framework provides implementations of some of these data structures.
Dynamic Data Structures

**Iterator**

- It is common in object-oriented programming to declare an `iterator` class that can traverse all the objects in a collection, such as an array or an `ArrayList` or `LinkedList`.
- For example, a program can print an `LinkedList` of objects by creating an `iterator` object and using it to obtain the next list element each time the `iterator` is called.
- `Iterators` often are used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy.
Dynamic Data Structures

Linked List Class

- **ListTest.java**
  - The program creates two LinkedLists that contain Strings.
  - The elements of one List are added to the other.
  - Then all the Strings are converted to uppercase, and
  - a range of elements is deleted
LinkedList Class

- The program output:
  list1:
  black yellow green blue violet silver gold white brown blue gray silver

  list1:
  BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

  Deleting elements 4 to 6...
  list1:
  BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

  Reversed List:
  SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
Common Programming Error

- If a collection is modified by one of its methods after an iterator is created for that collection, the iterator immediately becomes invalid—any operations performed with the iterator after this point throw `ConcurrentModificationExceptions`.
- For this reason, iterators are said to be “fail fast.”
LinkedList Class

- **static method asList** of class Arrays
  - Allow programmer to manipulate the array as if it were a list
  - Any modification made through the List view change the array
  - Any modification made to the array change the List view
Dynamic Data Structures

**LinkedList Class**

- **UsingToArray.java**
  - The program calls method `asList` to create a `List` view of an array, which is then used for creating a `LinkedList` object,
  - adds a series of strings to a `LinkedList` and
  - calls method `toArray` to obtain an array containing references to the strings.
  - Notice that the instantiation of `LinkedList` indicates that `LinkedList` is a generic class that accepts one type argument `String`, in this example.
LinkedLIst Class

- The program output:

  colors:
  cyan
  black
  blue
  yellow
  green
  red
  pink
Common Programming Error

- If the number of elements in the array is smaller than the number of elements in the list on which `toArray` is called, a new array is allocated to store the list’s elements.

- If the number of elements in the array is greater than the number of elements in the list, the elements of the array (starting at index zero) are overwritten with the list’s elements. Array elements that are not overwritten retain their values.
Collections Class
**Collections Class**

- Collections class provides set of algorithms, implemented as static methods, include:
  - sort
    - Sorts the elements of a List.
  - binarySearch
    - Locates an object in a List.
  - reverse
    - Reverses the elements of a List.
  - shuffle
    - Randomly orders a List's elements.
  - fill
    - Sets every List element to refer to a specified object.
  - copy
    - Copies references from one List into another.
Dynamic Data Structures

Collections Class

- min
  - Returns the smallest element in a Collection.

- max
  - Returns the largest element in a Collection.

- addAll
  - Appends all elements in an array to a collection.

- frequency
  - Calculates how many elements in the collection are equal to the specified element.

- disjoint
  - Determines whether two collections have no elements in common.
Algorithm sort

- **sort**
  - Sorts List elements
    - Order is determined by natural order of elements’ type

- **Sorting in ascending order**
  - Collections method **sort**

- **Sorting in descending order**
  - Collections static method **reverseOrder**
Algorithm sort

- **Sort1.java**
  - uses algorithm `sort` to order the elements of a `List` in ascending order.
  - Recall that `List` is a generic type and accepts one type argument that specifies the list element type

- The program output:
  
  Unsorted array elements:
  [Hearts, Diamonds, Clubs, Spades]
  Sorted array elements:
  [Clubs, Diamonds, Hearts, Spades]
Sorting in Descending Order

- **Sort2.java**
  - sorts the same list of strings in descending order.
  - The static `Collections` method `reverseOrder` returns a object that orders the collection's elements in reverse order. It is a parameter for `sort` method.

- The program output:
  
  Unsorted array elements:
  [Hearts, Diamonds, Clubs, Spades]
  
  Sorted list elements:
  [Spades, Hearts, Diamonds, Clubs]
Algorithm shuffle

- **ShuffleTest.java**
  - In this program we use algorithm `shuffle` to shuffle a deck of `Card` objects that might be used in a card game simulator.

- The program output:
  
  Array elements:
  
  [Hearts, Diamonds, Clubs, Spades]

  Shuffled list elements:
  
  [Spades, Clubs, Diamonds, Hearts]
Algorithm reverse, fill, copy, max and min

- **reverse**
  - Reverses the order of List elements

- **fill**
  - Overwrites elements in a List with a specified value.
  - The fill operation is useful for reinitializing a List.

- **copy**
  - Creates copy of a List
  - takes two arguments a destination List and a source List
  - Each source List element is copied to the destination List
  - The destination List must be at least as long as the source List; otherwise, an IndexOutOfBoundsException occurs.
  - If the destination List is longer, the elements not overwritten
Dynamic Data Structures

Algorithm reverse, fill, copy, max and min

- **max**
  - Returns largest element in List
  - Operate on any Collection

- **min**
  - Returns smallest element in List
  - Operate on any Collection
Algorithm1.java

- **Algorithm1.java** demonstrates the use of algorithms reverse, fill, copy, min and max.

- The program output:

  Initial list:
  The list is: P C M
  Max: P  Min: C

  After calling reverse:
  The list is: M C P
  Max: P  Min: C

  After copying:
  The list is: M C P
  Max: P  Min: C

  After calling fill:
  The list is: R R R
  Max: R  Min: R
The **binarySearch** algorithm locates an object in a List (i.e., a LinkedList or an ArrayList).

If the object is found, its index is returned. If the object is not found, **binarySearch** returns a negative value.

Algorithm **binarySearch** determines this negative value by first calculating the insertion point and making its sign negative.

Then, **binarySearch** subtracts 1 from the insertion point to obtain the return value, which guarantees that method **binarySearch** returns positive numbers (>=0) if and only if the object is found.
Algorithm binarySearch

- If multiple elements in the list match the search key, there is no guarantee which one will be located first.

- **BinarySearchTest.java**
  - uses the `binarySearch` algorithm to search for a series of strings in an `ArrayList`. 
Algorithm `binarySearch`

- The program output:
  
  Sorted list: [black, blue, pink, purple, red, tan, white, yellow]

  Searching for: black
  Found at index 0

  Searching for: red
  Found at index 4

  Searching for: pink
  Found at index 2

  Searching for: aqua
  Not Found (-1)

  Searching for: gray
  Not Found (-3)

  Searching for: teal
  Not Found (-7)
Algorithms **addAll, frequency and disjoint**

- **addAll**
  - Insert all elements of an **array** into a collection
  - Takes two arguments, a **Collection** into which to insert the new element(s) and an array that provides elements to be inserted

- **frequency**
  - Calculate the number of times a specific element appear in the collection
  - Takes two arguments a **Collection** to be searched and an **Object** to be searched for in the collection

- **disjoint**
  - Algorithm disjoint takes two **Collections** and returns true if they have no elements in common
Algorithms `addAll`, `frequency` and `disjoint`

- **Algorithms2.java**
  - demonstrates the use of algorithms `addAll`, `frequency` and `disjoint`.

- The program output:
  - Before `addAll`, list2 contains:
    - black red green
  - After `addAll`, list2 contains:
    - black red green red white yellow blue
  - Frequency of red in list2: 2
  - list1 and list2 have elements in common
Stack Class
Stacks

A stack is a constrained version of a linked list.

The link member in the bottom (i.e., last) node of the stack is set to null to indicate the bottom of the stack.

Last-In, First-Out (LIFO) data structure:
- Method push adds a new node to the top of the stack.
- Method pop removes a node from the top of the stack and returns the data from the popped node.

Program execution stack:
- Holds the return addresses of calling methods.
- Also contains the local variables for called methods.
Stack Class

- **Stack** class in the Java utilities package `java.util` implements stack data structure
- Class **Stack** stores references to objects
- **Autoboxing** occurs when you add a primitive type to a **Stack**
- Class **Stack** extends class **Vector** class to implement a stack data structure.
Stack Class

- Example 1:
  - TestStack.java

- The output:
  Stack is: -1
  Stack is: -1 0
  Stack is: -1 0 1
  Stack is: -1 0 1 5
  5 popped
  Stack is: -1 0 1
  1 popped
  Stack is: -1 0
Stack Class

● Example 2:
  - TestStack2.java

● The output:

  stack contains: 12 (top)
  stack contains: 12 34567 (top)
  stack contains: 12 34567 1.0 (top)
  stack contains: 12 34567 1.0 1234.5678 (top)
  1234.5678 popped
  stack contains: 12 34567 1.0 (top)
  1.0 popped
  stack contains: 12 34567 (top)
The constructor creates an empty Stack of type Number.

Class Number (in package java.lang) is the superclass of most wrapper classes (e.g., Integer, Double) for the primitive types.

By creating a Stack of Number, objects of any class that extends the Number class can be pushed onto the stack.
Stack Class

- Any integer literal that has the suffix L is a long value.
- An integer literal without a suffix is an int value.
- Similarly, any floating-point literal that has the suffix F is a float value.
- A floating-point literal without a suffix is a double value.
Stack Class

- Because Stack extends Vector, all public Vector methods can be called on Stack objects, even if the methods do not represent conventional stack operations.
- For example, Vector method add can be used to insert an element anywhere in a stack—an operation that could “corrupt” the stack.
- When manipulating a Stack, only methods push and pop should be used to add elements to and remove elements from the Stack, respectively.
PriorityQueue Class
Dynamic Data Structures

PriorityQueue Class

- Queue
  - Similar to a checkout line in a supermarket
  - **First-In, First-Out (FIFO)** data structure
    - **Enqueue** inserts nodes at the tail (or end)
    - **Dequeue** removes nodes from the head (or front)
  - Used to support print spooling
    - A spooler program manages the queue of printing jobs
PriorityQueue Class

- Interface Queue,
  - extends interface Collection and provides additional operations for inserting, removing and inspecting elements in a queue.

- Class PriorityQueue,
  - one of the classes that implements the Queue interface, orders elements by their natural ordering
  - When adding elements to a PriorityQueue, the elements are inserted in priority order such that the highest-priority element (i.e., the largest value) will be the first element removed from the PriorityQueue.
PriorityQueue Class

- The common PriorityQueue operations are
  - **offer** to insert an element at the appropriate location based on priority order
    - Method offer throws a NullPointerException if the program attempts to add a null object to the queue.
  - **poll** to remove the highest-priority element of the priority queue (i.e., the head of the queue),
  - **peek** to get a reference to the highest-priority element of the priority queue (without removing that element),
  - **clear** to remove all elements in the priority queue and
  - **size** to get the number of elements in the priority queue.
Dynamic Data Structures

PriorityQueue Class

- PriorityQueueTest.java
  - demonstrates the PriorityQueue class.

- The program output:
  Polling from queue: 3.2 5.4 9.8
HashSet Class
A **Set** is a Collection that contains unique elements (i.e., no duplicate elements), including:

- **HashSet**
  - Stores elements in hash table
- **TreeSet**
  - Stores elements in tree
HashSet Class

- HashSetTest.java
  - Recall that both List and Collection are generic types, so this program creates a List that contains String objects, and
  - It passes a Collection of Strings to method printNonDuplicates.

- The program output:
  - ArrayList: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]
  - Nonduplicates are: orange green white peach gray cyan red blue tan
References


The End