Data Structure

Recitation VIII
Topic

- Queue: Link-based Implementation
- Two applications: Palindromes & Producer Consumer
- Object comparing
Linked base Implementation

- Unbounded Queue
package ch05.queues;

import support.LLNode;

public class LinkedUnbndQueue<T> implements UnboundedQueueInterface<T> {
    private LLNode<T> front = null, rear = null;
    public boolean isEmpty() { return front == null; }
    public void enqueue(T data) { // shown above }
    public T dequeue() { // shown above }
}
Enqueue

- Enqueue
  - Create a new node using an LLNode<T> constructor.
  - Make the current last node point to this new node.
  - Make rear point to this new node.

- When inserting into an empty queue:
  - Rear does not point to the last node; instead rear==null.
  - rear.setLink(newNode) error!
Enqueue

public void enqueue(T data) {
    LLNode<T> newNode = new LLNode<T>(data);
    if (isEmpty())
        front = newNode;
    else
        rear.setLink(newNode);
    rear = newNode;
}
Dequeue

- Dequeue
  - Save the value of front.
  - Move front one node forward.
  - Return the value saved.

- Clearly we can't dequeue from an empty queue. Instead we raise an exception.

- 1-element queue
  - Front and rear point to the node we are removing.
  - Empty queue, Set rear=null
public T dequeue() {
    if (isEmpty())
        throw new QueueUnderflowException("Helpful msg");
    T ans = front.getInfo();
    front = front.getLink();
    if (front == null)
        rear = null;
    return ans;
}
Circular Linked Queue

- Have only one instance variable: rear, rather than two.

- To access front: Front = rear.getlink();
Palindromes

- A string that reads the same forward and backward
- isPalindrome(): Using both queue and stack
  - Obtain the character from the string charAt().
  - Enqueue it.
  - Stack it.
  - Dequeue it.
  - Pop it (including top).
Palindromes

- Only use stack
- First the characters are extracted one by one from the input string and pushed onto the stack.
- Then they are popped from the stack and compared with each character of the given string.
- It is enough to compare with first half of the string. Because of its last-in-first-out characteristic, the stack reverses the order of the characters.
static boolean isPalindrome(String str)
{
    Stack stk = new Stack();
    for( int i=0; i < str.length(); i++ )
        stk.push(str.charAt(i));
    for( int i=0; i < str.length()/2; i++ )
        if( str.charAt(i) != stk.pop() ) return false;
    return true;
}
}
Palindromes

- Only use queue
- First the characters are extracted one by one from the input string and inserted into the queue.
- Then they are removed from the queue and compared with each character of the given string (in the reverse order of the string).
- It is enough to compare with second half of the string (of course in the reverse order). Because of its first-in-first-out characteristic, the characters are deleted from the queue in the order of the characters of the string.
static boolean isPalindrome(String str) {
    LinkedList<Character> que = new LinkedList<Character>();
    int n = str.length();
    
    for (int i=0; i < n; i++)
        que.addLast(str.charAt(i));
    
    for (int i=n-1; i > n/2; i--)
        if (str.charAt(i) != que.removeFirst()) return false;
    return true;
}
The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate a piece of data, put it into the buffer and start again.

At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.
procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) {
            sleep();
        }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
    }
}

procedure consumer() {
    while (true) {
        if (itemCount == 0) {
            sleep();
        }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        consumeItem(item);
    }
}
Object Comparing

- ==, .equals(), compareTo(), and compare()
==

- Compares references, not values. The use of `==` with object references is generally limited to the following:
  - Comparing to see if a reference is null.
  - You want to know if two references are to the same object.
  - Used for both Objects and Primitives.
equals()

- Compares values for equality. Because this method is defined in the Object class, from which all other classes are derived, it's automatically defined for every class.

- However, it doesn't perform an intelligent comparison for most classes unless the class overrides it. It has been defined in a meaningful way for most Java core classes. If it's not defined for a (user) class, it behaves the same as ==.

- It turns out that defining equals() isn't trivial; in fact it's moderately hard to get it right, especially in the case of subclasses. Be sure equals have the same signature as the super class.
equals()

- If you override equals, you should also override hashCode().

- Overriding hashCode(). The hashCode() method of a class is used for hashing in library data structures such as HashSet and HashMap. If you override equals(), you should override hashCode() or your class will not work correctly in these (and some other) data structures.
public boolean equals (Object circle) {
    if (circle instanceof Circle)
        return this.radius == ((Circle) circle).radius;
    return false;
}

a.compareTo(b)

- Comparable interface. Compares values and returns an int which tells if the values compare less than, equal, or greater than.

- If your class objects have a natural order, implement the Comparable<T> interface and define this method. All Java classes that have a natural ordering implement this (String, Double, BigInteger, ...).
public class User implements Comparable {

    private String id;
    private int age;

    public User(String id, int age) {
        this.id = id;
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public int compareTo(Object o) {
        return this.age - ((User) o).getAge();
    }
}
Compare(a,b)

- Comparator interface. Compares values of two objects. This is implemented as part of the Comparator<T> interface, and the typical use is to define one or more small utility classes that implement this.

- Multiple comparisons. To provide several different ways to sort something. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort() method.

- System class. To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length.

- If your class objects have one natural sorting order, you may not need this.
public class UserComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        return ((User) o1).getAge() - ((User) o2).getAge();
    }
}
public class Fruit implements Comparable<Fruit> {
    private String fruitName;
    private String fruitDesc;
    private int quantity;

    public int compareTo(Fruit compareFruit) {
        int compareQuantity = ((Fruit) compareFruit).getQuantity();

        // ascending order
        return this.quantity - compareQuantity;

        // descending order
        // return compareQuantity - this.quantity;
    }
}
Sort Example(I)

```java
public class Fruit implements Comparable<Fruit>{
    private String fruitName;
    private String fruitDesc;
    private int quantity;

    public int compareTo(Fruit compareFruit) {
        int compareQuantity = ((Fruit) compareFruit).getQuantity();
        return this.quantity - compareQuantity;
    }
}
```
public static void main(String[] args) {

    Fruit[] fruits = new Fruit[4];
    Fruit pineapple = new Fruit("Pineapple", "Pineapple description", 70);
    Fruit apple = new Fruit("Apple", "Apple description", 100);
    Fruit orange = new Fruit("Orange", "Orange description", 80);
    Fruit banana = new Fruit("Banana", "Banana description", 90);
    fruits[0] = pineapple;
    fruits[1] = apple;
    fruits[2] = orange;
    fruits[3] = banana;
    Arrays.sort(fruits);
    int i = 0;
    for (Fruit temp : fruits) {
        System.out.println("fruits " + ++i + " : " + temp.getFruitName() + ", Quantity : " + temp.getQuantity());
    }
}
Public class FruitNameComparator implements Comparator {
    public int compare(Fruit fruit1, Fruit fruit2) {
        String fruitName1 = fruit1.getFruitName().toUpperCase();
        String fruitName2 = fruit2.getFruitName().toUpperCase();
        //ascending order
        return fruitName1.compareTo(fruitName2);
        //descending order
        //return fruitName2.compareTo(fruitName1);
    }
};
Arrays.sort(fruits, new FruitNameComparator());