COMP 1020 - Abstract Data Types (ADTs)

UNIT 10

Abstraction

- In the previous unit, we learned about interfaces, which describe what we *can* do with a class without knowing *how* it does it.
- Interfaces are one tool that allow us to focus on what's important in a context and ignore the details.
- In Computer Science this strategy of ignoring the details is called **abstraction**.
- It allows us to build systems out of complex components without getting lost.

Abstraction

- Our ability to write programs already relies on abstractions.

- When we write code like: System.out.println("hello"); we know what will happen but we don't necessarily know how it happens.

  - We think that maybe the computer uses electricity and ones and zeroes and ??? (*magic*?)

  - In the end, we see the output on our display.

Abstraction

- An example of an abstraction is an “opaque box”, where data comes in, is processed, and output is produced, somehow.

- We can describe the inputs and generated outputs without having to say how the box produces them.

- Example: we can know what a sort() function does from its name, without seeing the implementation.

Data Abstraction

- We can apply this concept of abstraction to the techniques our program uses to store data.

- A **data abstraction** is a description of data storage according to the organization of the data, and the operations we can perform on it.

  - Basically, a name that describes how the data is organized, and an interface.

Abstract Data Type

- Then an **abstract data type** or **ADT** is a particular example of a data abstraction.

- The ADT List is one that we have been using throughout this course.

  - The data is in a List is organized sequentially; that is, there is a first, second, third, etc. element.

  - Operations could include add, get, set, remove, size, and find.

ADT vs Data Structure

- The name and operations of the ADT form the public interface.
- The private implementation uses a particular **data structure**.
  - Examples of data structures that can be used to implement the ADT List are arrays or linked lists.

Another ADT Example

- A variation of the ADT List is the ADT Ordered List.

- An Ordered List contains data organized sequentially, just like a List. However, the data is guaranteed to be sorted in some defined order (numerically, alphabetically, etc., depending on the data stored).

- The operations are slightly different from List. We can still have get, remove, size, and find. There is no add, but the insert operation will insert into a unique (ordered) position. There is no set operation, because that could violate the list's ordering.

ADTs and Formal Interfaces

- A programming language like Java that has formal interfaces can use them to define an ADT.

- For example, this is (slightly edited, incomplete) Java's built-in ADT List interface:

```
interface List<E> {
    void add(E obj);
    E get(int index)
    void set(int index, E obj);
    int size();
    …
```

E? No, Object!

- Java uses a technique known as “generics” to allow a List to store any type of data. That’s what `E` meant in the previous example.
  - Unfortunately these can be tricky to implement.
- Instead, we will use a simpler strategy known as the Java `Object`.

Object

- An Object stands in for any kind of object; that is, an instance of any class.

- If we define our List interface to store Object, we can put any type of object in it we like.

```
interface ObjectList {
    void add(Object obj);
    Object get(int index)
    int size();
    …
```

Object

- We can then add anything to our generalized list.

- For example:

```
ObjectList list;
// construct a list here (not shown)

list.add("hello, world");
list.add(new Student(...));
list.add(3); // adds an Integer containing 3
System.out.println(list.get(list.size() - 1);
```

Using Object

- When we get an object out of our list, we get an Object, but its actual class is unknown.

```
Object obj = list.get(0);
```

- We can't call class-specific methods on obj.
  - For example, even if we are certain obj is a String, we can't call obj.length()

Casting Object

- Instead, we can cast an Object to a more specific type:

```
String str = (String)list.get(0);
```

- Then we can use str like any other String.
  - But we have to be certain that the object we got from the list is the expected type, or else this will crash with a ClassCastException

Safely casting an Object

- One way to safely cast an object is to check using the instanceof operator, which gives a boolean result:

```
Object obj = list.get(0);
if (obj instanceof String) {
    String str = (String)obj;
}
```

- The instanceof operator produces a true result if the object is actually an instance of that class.

Easier and safer

- A generally better strategy is to use a generalized List of objects to build a type-specific collection class:

```
class StringList {
    private ObjectList objList;
    // constructor and other methods not shown
    public void add(String str) {
        objList.add(str);
    }
    public String get(int index) {
        return (String)(objList.get(index));
    }
}
```

List implementation

- The implementation of the List can be an array:

```
class ListWithArray implements ObjectList {
    private Object[] array;
    private int size; // partially-filled array

    ...
}
```

List implementation

- Or a linked list:

```
class ListWithLL implements ObjectList {
    private ObjectNode top;

    ...
}

class ObjectNode {
    public Object data;
    public ObjectNode next;
}
```

ADT OrderedList

- We can't easily make an OrderedList out of Object, but we can make it out of Comparable:

```
interface OrderedList {
    void insert(Comparable obj);
    Comparable get(int index);
    void remove(int index);
    int find(Comparable obj);
    int size();
    …
}
```

A general Comparable

- Because we don't use generics, Comparable has dropped the <Classname> part at the end of it.
- Like Object, this Comparable needs to be cast to a more specific type when we take it out of the list.
- An implementation of the insert method will call compareTo(Object other) to determine order.
- And the OrderedList can be implemented by an array or linked list of Comparable objects.

More ADTs

- There are many other ADTs used to solve problems in Computer Science.

- We will look at two in particular: the **stack** and the **queue**.

Stack

- A **stack** of data is a sequence of values where only the most recently added value, of all the values currently in the sequence, can be accessed.

- The value on "top" of the stack is the only one we can see or remove. The top is the only spot we can add a new value.

Stack

- You can place an item onto the top of the stack.
- You can examine the item on the top of the stack.
- You can remove the item from the top of the stack.
  - The item beneath it is now the top. It can be removed, or more items placed on it.
- We can’t remove items when none are left.
  - Until we add more.

Stack

- You can place an item onto the top of the stack.
  - This is called a **push** operation.
- You can examine the item on the top of the stack.
  - This is called a **peek** operation.
- You can remove the item from the top of the stack.
  - The item beneath it is now the top. It can be removed, or more items placed on it.
  - This is called a **pop** operation.
- We can't remove items when none are left.
  - Check this with an **isEmpty** operation.
  - Until we push more.

ADT Stack

- An ADT Stack is defined by this restricted set of operations.
  - Just like the Ordered List, where the ADT cannot allow operations that violate its ordering constraint.
- If you allow it to have (for example) a get for an arbitrary element, it's not a Stack any more!

ADT Stack

- We can define a stack using the following interface. It can be a stack of Objects, or a more specific type.

```
interface Stack {
    void push(Object obj);
    Object pop(); // returns the object popped
    boolean isEmpty();
    // this is optional because we can also access the
    // top object using pop() :
    Object peek();
}
```

Implementing a stack

- A stack can be implemented using a partially-filled array: we can easily push items to the end of the array, and pop them from the end.
  - The end of the array is the “top” of the stack.
  - We could also use an ArrayList, only allowing access at the end.
- A stack can be implemented using a linked list, where we push/pop from the top of the linked list.

Applications of stacks

- Some places where stacks are used:
  - To reverse items: push n items onto a stack, then pop them all to get them in the reverse order.
  - Return path finding ("backtracking"): go from *a* to *b* to *c* to *d*, push each location onto a stack, and pop them to go back.
    - We can try following different paths (branches) when we go back to an earlier location.
  - The run-time stack that makes recursion possible.

Replacing recursion

- The last application is particularly interesting.
- The run-time stack is part of every program's execution environment, and allows functional independence and recursion.
- By defining our own stack, we can simulate the use of the run-time stack in a recursive function.
  - That means we can replace any recursive function with a non-recursive function and our own stack!

Queue

- A **queue** of data is a sequence of values where only the least recently added value, of all the values currently in the sequence, can be accessed.

- We add new values to one end of the queue, but we can only access values at the other end.

Queue

- You can place an item at the back/**tail** of the queue.
- You can examine the item at front/**head** of the queue.
  - Head and tail are at the opposite ends!
- You can remove the item from the head of the queue.
- We can't remove items when none are left.
  - Until we add more.

Queue

- You can place an item at the back/**tail** of the queue.
  - This is called an **enqueue** operation.
- You can examine the item at front/**head** of the queue.
  - Head and tail are at the opposite ends!
  - This is called an **peek** operation.
- You can remove the item from the head of the queue.
  - This is called a **dequeue** operation.
- We can't remove items when none are left.
  - Check this with an **isEmpty** operation.
  - Until we enqueue more.

ADT Queue

- We can define a queue of Objects using the following interface. Like a Stack, we limit the operations.

```
interface Queue {
    void enqueue(Object obj);
    Object dequeue(); // returns the head object
    boolean isEmpty();
    // this is optional because we can also access the
    // head using dequeue() :
    Object peek();
}
```

Implementing a queue

- A queue can be implemented using a partially-filled array: we could enqueue items at the end (easy) and dequeue them at the start (hard!).
  - It's hard because we need to shift the remaining elements over when we dequeue the head. It's easier but no more efficient with an ArrayList.
  - The start of the array is the head, end is tail.
  - There is a better strategy using arrays, covered in later CS courses.

Queues with a linked list

- A queue can be implemented using a linked list.
  - We can enqueue at the end of the linked list and dequeue at the beginning. That is, the start of the linked list is the head, the end is the tail.
  - It's still a lot of work to enqueue values at the end, because we have to traverse the entire list to get there...
    - … or do we?

Linked lists with a tail

- A simple modification to our typical linked list makes queues much easier and more efficient.

- Add a “bottom” or “tail” reference to keep track of the position of the last node.

```
public class LinkedList {
    private Node head; // A reference to the first Node
    private Node tail; // A reference to the last Node
    …
}
```

Linked lists with a tail

- All the operations that modify our linked list need to be updated to use the new reference.

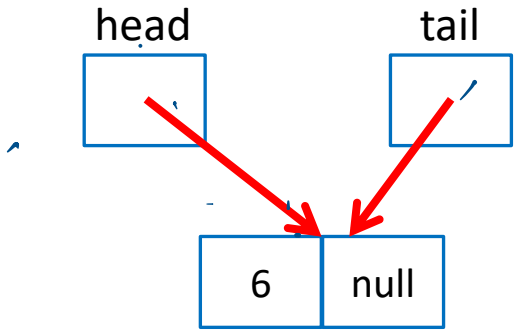
- Fortunately, for a Queue, we only need two such operations: enqueue and dequeue.

```
public class QueueWithLL implements Queue {
    private Node head;
    private Node tail;
    …
}
```

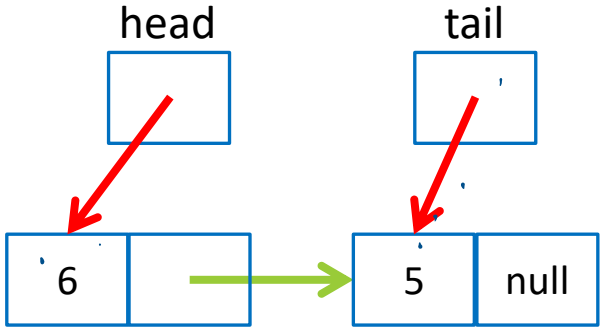
Linked lists with a tail

- In an empty linked list, both head and tail are null.

- When we enqueue the first item, we update both to point to it.

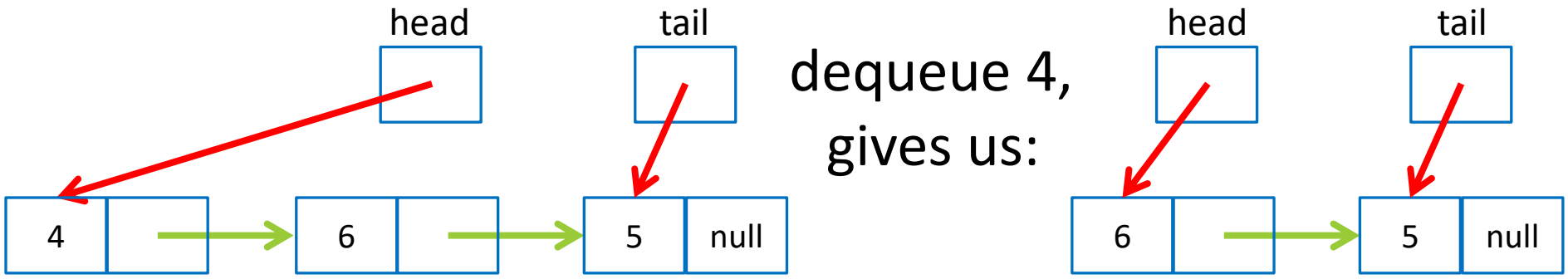


- As we enqueue more items, head still points to the first item, but tail always points to the last one added.



Linked lists with a tail

- To dequeue items, we remove them from the beginning of the list.
  - The head pointer changes, tail does not.



- Until we dequeue the last item, when both head and tail are set to null.

Linked lists with a tail

- Neither enqueuing nor dequeuing need a loop.

- For example:

```
public void enqueue(Object obj) {
    Node newNode = new Node(obj, null);
    if (tail == null) { // same as: head==null
        head = newNode;
    } else {
        tail.next = newNode;
    }
    tail = newNode;
}
```

Applications of queues

- Some places where queues are used:
  - When a program has to keep track of the work it needs to do, it can add the tasks to a queue and execute them in order.
  - As a *buffer*: to store a sequence of data values that need to be processed.
    - Such as sending data on a slow network, where new data has to wait until old data has been transmitted.