Section 1

ADTs, Data Structures, Java Generics

Hello!

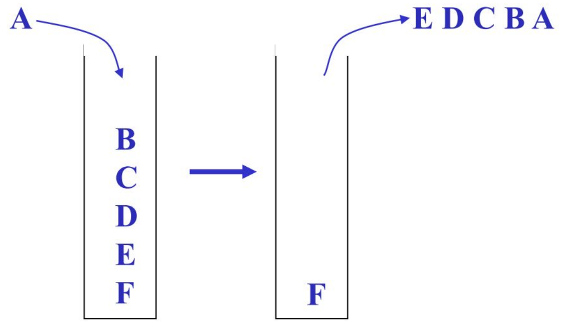
- Name
- Pronouns
- Year
- Hometown
- Hobbies
- Fun fact

- Name
- Pronouns
- Year
- Hometown
- Hobbies
- Fun fact

Icebreaker Activity

- Describe an icebreaker activity here
- Should allow folks to get up and introduce themselves to each other

Abstract Data Type (ADT)



From lecture:

Mathematical description of a “thing” with set of operations on that “thing”

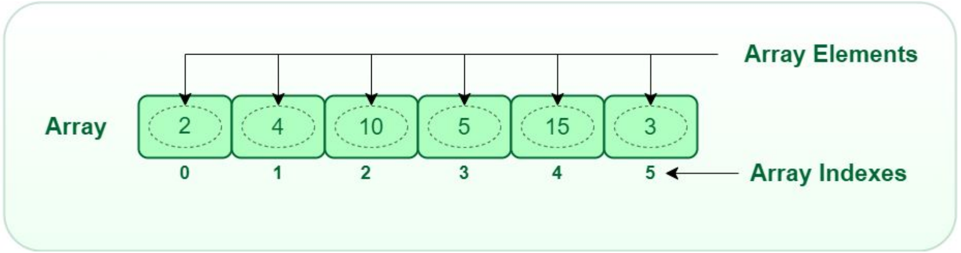
Main Idea:

Based on the point of view **from the USER** (as opposed to the implementer)

3 Descriptors:

1. Operation (e.g. stack has a push ~~method~~ operation)
2. Behaviour (e.g. stack has LIFO behaviour)
3. Data Type (e.g. stack can store any elements)

Data Structure



From lecture:

A specific organization of data and family of algorithms for implementing an ADT

Main Idea:

Based on the point of view **from the IMPLEMENTER** (as opposed to the user)

i.e. it is the **specification (data + algorithms) of an ADT**

e.g. stack can use an underlying **array** data structure and specific **pop/push algorithms**

List Abstract Data Type

Summary/Idea: An indexable collection of items held in a sequence

Identify List operations:

-

Identify List Data Structures:

-

List Abstract Data Type

Summary/Idea: An indexable collection of items held in a sequence

Identify List operations:

- Get
- Add
- Remove

Identify List Data Structures:

- Linked List
- Array List

Stack Abstract Data Type

Summary/Idea: A collection of items accesses in LIFO order

Identify Stack Operations:

-

Identify Stack Data Structures:

-

Stack Abstract Data Type

Summary/Idea: A collection of items accesses in LIFO order

Identify Stack Operations:

- Push
- Pop
- Peek

Identify Stack Data Structures:

- Linked list
- Array List (resizable array)

Queue Abstract Data Type

Summary/Idea: A collection of items accessed in FIFO order

Identify Queue Operations:

-

Identify Queue Data Structures:

-

Queue Abstract Data Type

Summary/Idea: A collection of items accessed in FIFO order

Identify Queue Operations:

- Enqueue
- Dequeue
- Peek

Identify Queue Data Structures:

- Linked List
- Array List (resizable array)
- Circular Array

Set Abstract Data Type

Summary/Idea: An unordered collection of items without duplicates

Identify Set Operations:

-

Identify Set Data Structures:

-

Set Abstract Data Type

Summary/Idea: An unordered collection of items without duplicates

Identify Set Operations:

- Add
- Remove
- Contains

Identify Set Data Structures:

- Linked List
- Array List
- Binary Tree
- Hash Set

Java Generics

- Java is a strongly typed language, meaning everything needs a type
- Java requires it know what type of things every collection holds
- “Generics” is the name of the strategy Java uses to do this
    - Example: ArrayList<String> myArrayList = new ArrayList<>();
    - Name of the type that the array list contains goes inside <>
    - The ArrayList implementation is written to work for any kind of Object
- Since we’re going to be implementing data structures this quarter, we need to know how to use these!
- Note: different programming languages solve this problem in different ways, we’re just presenting on how Java does it.

Pair ADT

Idea: One object which holds an ordered pair of items

Operations:

- Set first
- Get first
- Set second
- Get second

Data Structure:

- An object with a first and second parameter

IntPair Data structure

```
public class IntPair {
    private int first;
    private int second;
    public IntPair(int first, int second){
        this.first = first;
        this.second = second;
    }
    public int getFirst(){
        return first;
    }
    public int getSecond(){
        return second;
    }
    public void setFirst(int newFirst){
        this.first = newFirst;
    }
    public void setSecond(int newSecond){
        this.second = newSecond;
    }
}
```

This Pair data structure is set up to work exclusively with Integers. We cannot easily use this to hold pairs of doubles, strings, nodes, etc.

We would need to re-implement the class for each type!

LikePair Data structure

```java
public class LikePair<T> {
    private T first;
    private T second;
    public LikePair(T first, T second){
        this.first = first;
        this.second = second;
    }
    public T getFirst(){
        return first;
    }
    public T getSecond(){
        return second;
    }
    public void setFirst(T newFirst){
        this.first = newFirst;
    }
    public void setSecond(T newSecond){
        this.second = newSecond;
    }
}
```

Using generics we essentially treat the type itself as a field in the class.

We put <T> to indicate that the type that the pair will contain will be called “T”, and now we can use it just like we might use int or String!

UnlikePair Data structure

```
public class UnlikePair<A,B> {
    private A first;
    private B second;
    public UnlikePair(T first, T second){
        this.first = first;
        this.second = second;
    }
    public T getFirst(){
        return first;
    }
    public T getSecond(){
        return second;
    }
    public void setFirst(T newFirst){
        this.first = newFirst;
    }
    public void setSecond(T newSecond){
        this.second = newSecond;
    }
}
```

We can even have multiple types! Here, the first item will be of type A, the second item will be of type B.

Replace the Ts in the rest of the implementation to finish it!

Tuple ADT

Idea: One object which holds a fixed number of items.

Operations:

- Get ith
  - Return the ith object in the tuple
- Set ith
  - Change the ith object in the tuple

Data Structure:

- An object with an array

Tuple Data structure

```
public class Tuple<T> {
    private final T[] data;
    public Tuple(T[] tuple){
        data = (T[]) new Object[tuple.length]; // this is a weird thing, but is necessary. Java does not allow
"new T[5]", so instead we must do "(T[]) new Object[5]". That is, create an array of Objects and then cast it to
an array of Ts.
        for(int i = 0; i < tuple.length; i++){
            data[i] = tuple[i];
        }
    }
    public T getIth(int i){
        return data[i];
    }
    public void setIth(int i, T value){
        data[i] = value;
    }
}
```

LikeAnimalPair Data structure

```
public class LikeAnimalPair<T extends Animal> {
    private T first;
    private T second;
    public LikeAnimalPair(T first, T second){
        this.first = first;
        this.second = second;
    }
    public T getFirst(){
        return first;
    }
    public T getSecond(){
        return second;
    }
    public void setFirst(T newFirst){
        this.first = newFirst;
    }
    public void setSecond(T newSecond){
        this.second = newSecond;
    }
```

Suppose we wanted to be able to assume that a type inherited from a particular class. We can actually use the “extends” keyword to require a type to have a superclass!

In this example, both items need to be the same animal

Your turn!

Implement the following pair data structures:

1. TypeSubtypePair
   a. The first item is of some type, the second item is of a subtype of the first
2. IntOtherPair
   a. The first item is an integer, the second item is some other type
3. ItemPairPair
   a. The first item is of some type, the second item is a LikePair of things whose type inherits the first item’s
4. ItemArrayPair
   a. The first item is of some type, the second item is an array of objects of another type
5. ComparablePair (challenge!!!)
   a. A pair of items such that both of them implement the comparable interface.
   b. A comparable pair must itself be comparable!