CS 261

Abstract Data Types
Container Abstractions

• Over the years, programmers have identified a small number of ways to organize and operate on collections of data
  – These are sometimes called container abstractions or container classes

• These abstractions are at the heart of the study of data structures

• Examples: Stack, Queue, Set, Map, etc.
Container Abstractions

• Containers are reusable if implemented properly

• Need a clean interface

• Most languages have established container libraries

• Understanding the guts of containers will help you make informed choices
Three Levels of Abstraction

• ADT - Abstract Data Type, language independent – **What are the general properties?**

• Interface - in a particular library of containers -- **How to use the ADT?**

• Implementation - in a particular library – **How it is implemented?**
ADTs versus Data Structures

• Is there a difference between the concepts of container classes and data structures?

• A container class is an abstract concept of data organization and manipulation
  – Stack, Queue, Bag, etc

• Container classes are implemented using data structures (e.g. linked lists)
  – Could implement a container using different data structures to get different trade-offs
Abstract Data Type View

- Data type described in a language-independent way

- Properties are true regardless of the names given to operations in library

- E.g., A Stack is a collection where an item removed is the most recently entered item (LIFO property)
Metaphors

• ADT view are often described by metaphors (e.g., stack of plates). Easy to understand.
The Interface View

• Gives specific names to operations
• Follows conventions of a particular language
• In C, interface is defined by .h files

• E.g., stack interface:
  – Initialize the stack
  – Push = add an element to the stack
  – Pop = read and remove an element from the stack
  – Top = read the top element of the stack
  – Check if the stack is empty
The Interface View

• Gives specific names to operations
• In C, interface is defined by .h files

struct stack;
void initStack (struct stack * stk);
void pushStack (struct stack * stk, double val);
double topStack (struct stack * stk);
void popStack (struct stack * stk);
int isEmptyStack (struct stack * stk);
Additional Information

• The interface gives names of operations

• Should also attach meanings
  – E.g., LIFO properties of stack, etc
  – Comments should explain meaning

• May attach expected execution times
  – E.g., want push and pop to be constant time
  – Comments and documentation should explain
The Implementation View

```c
#define TYPE double
struct Stack {
    TYPE data [100];
    int count;
};

int stackIsEmpty (struct Stack * stk) {
    return stk->count == 0;
};
```
The Classic ADTs

- Bag, Ordered Bag - simple collections
- Stack, Queue, Deque - ordered by insertion
- Set - unique elements, fast test
- Map (Dictionary) - key/value associations
- Priority Queue - ordered by importance
BAGs
Bag as ADT

• **Problem**: Need to maintain an *unordered* collection of elements

• **Operations**:
  – Insert
  – Remove
  – Contains

• **Specific requirements**:
  – E.g., Time of insertion is important
Bag Interface

• Provide functions for operations

addBag (container, value)
testBag (container, value)
removeBag (container, value)
sizeBag (container, value)
Stack as ADT

- **Problem**: Maintain a collection of elements in Last-In, First-Out format

- **Operations**:
  - Add an element to Stack
  - Remove an element from Stack
  - Read the top element of Stack
  - Contains
The Classic Implementations

Common data structures for implementing ADTs

- Arrays and Dynamic Arrays (Vectors)
- Linked Lists
- Binary Trees, Balanced Trees
- Heaps
- Hash Tables
- Etc etc etc
Worksheet 0
Worksheet 0

• Make implementations of a simple Bag and Stack using an array in C

• We will solve only a part of Worksheet 0 in class, while you should try to solve the remainder on your own at home
First look at C - interface file

# define TYPE double
# define EQ(a, b) (a == b)

struct arrayBag {
    TYPE data[100];
    int count;
};

void initBag (struct arrayBag * b);  … etc
First function - initialize

```c
void initBag (struct arrayBag * b) {
}
```
Add to bag

void addBag (struct arrayBag * b, TYPE v)
{

}
Test for contains

int containsBag (struct arrayBag * b, TYPE v)
{
}
Remove from bag

void removeBag (struct arrayBag * b, EleType v)
{

}
Return size of collection

int sizeBag (struct arrayBag * b)
{
}
}
How about stack?

```c
void pushStack (struct arrayBag * b, TYPE v) {
}
```
Test for stack empty

```c
int isStackEmpty (struct arrayBag * b)
{
}
```
Top of stack

eleType topStack (struct arrayBag * b)
{
}
}
void popStack (struct arrayBag * b) {
}