CSCI 104
Abstract Data Types

Mark Redekopp
David Kempe

Revised: 05/2022

Abstract Data Types

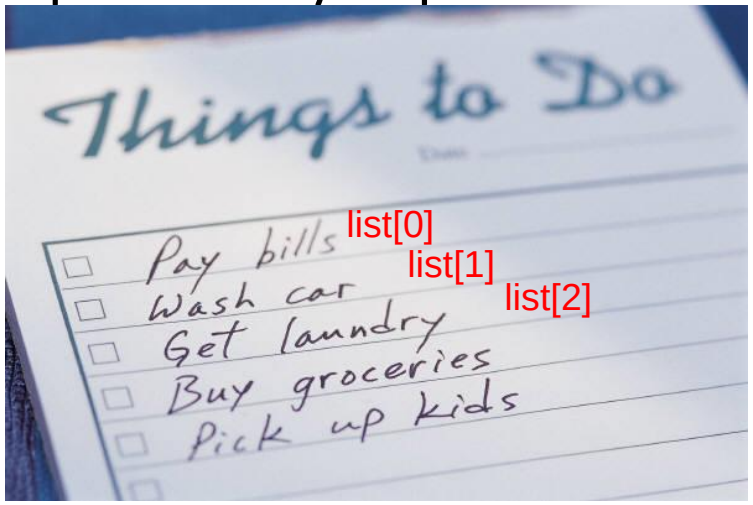
- An **abstract data type**, or ADT, as a specification or model for a group of values/data and the operations on those values
- A **data structure** is a specific implementation of an ADT in a given programming language
- As an analogy think of the ADT as the class declaration (header file) and the data structures are various implementations of a specific ADT (source files)
- Given an application we can quickly identify the ADT and then proceed to choose an appropriate data structure
- Each data structure we will examine in this course has certain:
  - Well defined operations and capabilities that are often useful
  - Time & space advantages
  - Time & space disadvantages
- You need to know those operations, advantages and disadvantages

Popular ADTs

- The "Big 3" ADTs
  - List
    - 3 specialized List ADTs: Queues, Stacks, Deques
  - Set
  - Map (Dictionary)
- Other ADTs
  - Priority Queue
  - Graphs

Lists

- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are some operations you perform on a list?



List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

Queues and Stacks

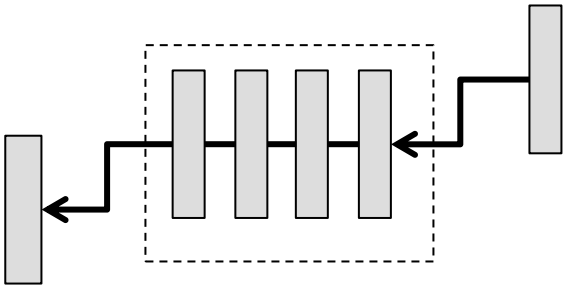
- Two specialized List ADTs

Queue

2 Items leave from the other side (often the front…pop_front)

1 Items enter from one side (often the back…push_back)

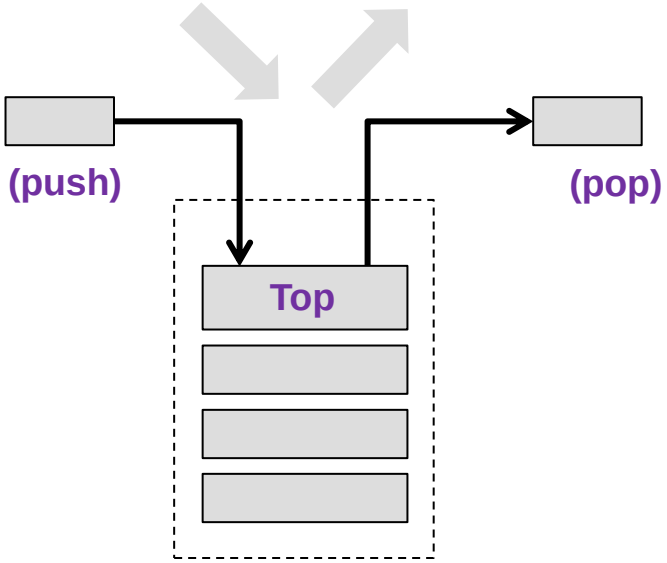
(push_back)



(pop_front)

Stack

Items enter and leave from the same side (i.e. the top)



Queue & Stack Operations

Queues

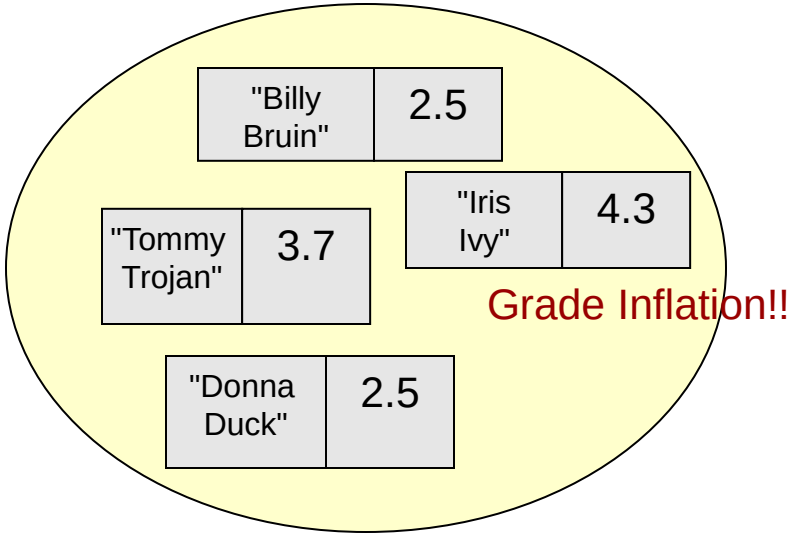
Operations Relative to Lists	Notes
~~insert~~	
~~remove~~	
get (front)	Can only get front item
set	
empty	
size	
push_back	Add to one side
pop_front	Remove from the other

Stacks

Operations Relative to Lists	Notes
~~insert~~	
~~remove~~	
get (top)	Can only get top item
set	
empty	
size	
push	Add to one side
pop	Remove from the same

Maps / Dictionaries

- Stores **key, value pairs**
  - Example: Map student names to their GPA
- Keys must be **unique** (can only occur once in the structure)
- No constraints on the values (can have duplicates)
- What operations do you perform on a map/dictionary?
- No inherent ordering between key,value pairs
  - Can't ask for the 0^{th} item…
- **Primary operations**:
  - **Insert, remove, find/lookup**

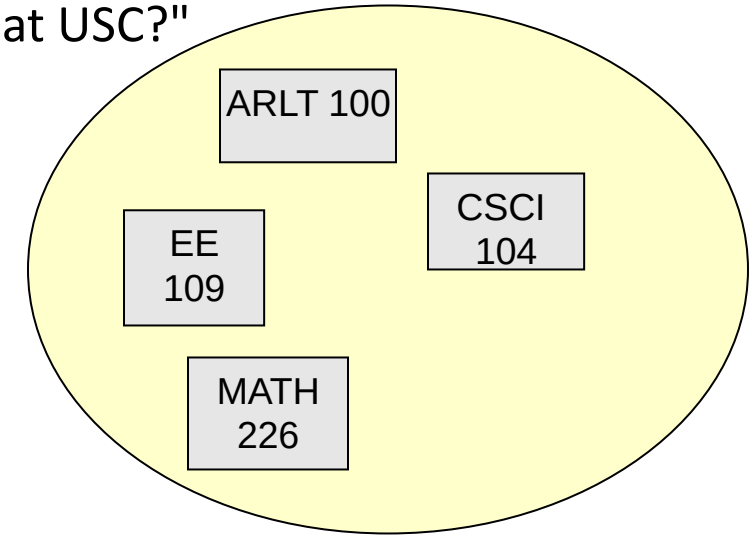


Map / Dictionary Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key,value pair to the dictionary (assuming its not there already)	Key, Value	
Remove	Remove the key,value pair with the given key	Key	
Get / lookup	Lookup the value associated with the given key or indicate the key,value pair doesn't exist	Key	Value associated with the key
In / Find	Check if the given key is present in the map	Key	bool (or ptr to pair/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int

Set

- A set is a dictionary where we only store keys (no associated values)
  - Example: All the courses taught at USC (ARLT 100, …, CSCI 104, MATH 226, …)
- Items (a.k.a. Keys) must be **unique**
  - No duplicate keys (only one occurrence)
- Not accessed based on index but on value
  - We wouldn't say, "What is the 0^{th} course at USC?"
- What operations do we perform on a set?
  - **Similar to a map**
  - **Insert, remove, find/in**

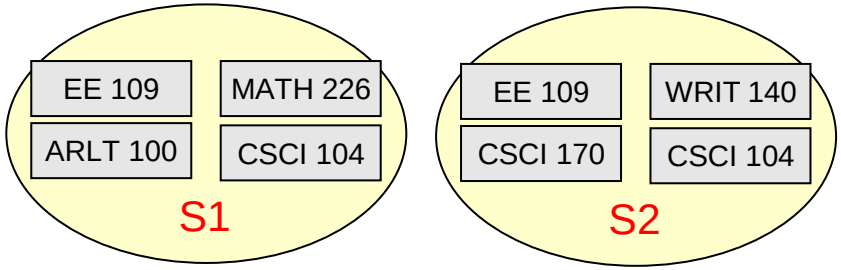


Set Operations

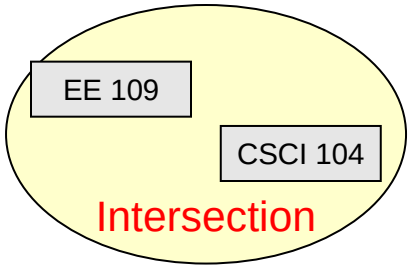
Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key to the set (assuming its not there already)	Key	
Remove	Remove	Key	
In / Find	Check if the given key is present in the map	Key	bool (or ptr to item/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Int
intersection	Returns a new set with the common elements of the two input sets	Set1, Set2	New set with all elements that appear in both set1 and set2
union	Returns a new set with all the items that appear in either set	Set1, Set2	New set with all elements that appear in either set1 and set2
difference	Returns a set with all items that are just in set1 but not set2	Set1, Set2	New set with only the items in set1 that are not in set2

Intersection, Union, Difference

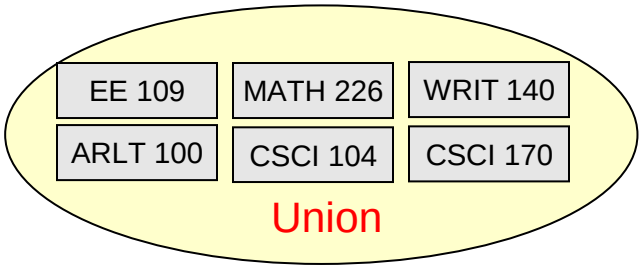
- May be familiar from CS 170



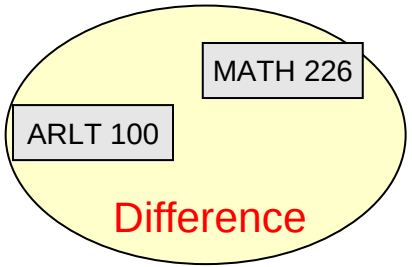
- Set intersection
  - S1 ∩ S2



- Set Union
  - S1 ∪ S2



- Set Difference
  - S1 – S2



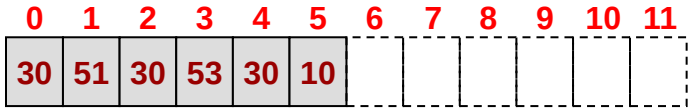
What's Your ADT?

- Scores on a test
- Students in a class
- Courses & their enrollment
- Temperature Reading at a location
- Usernames and password
- Index in a textbook
- Facebook friends

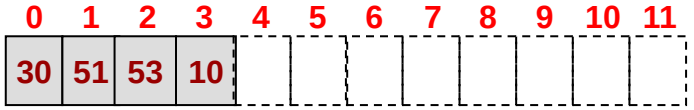
- List
- Set (maybe List)
- Map (Key = course, Value = enrollment)
- List
- Map
- Map
- Set

Some Implementation Details

- List
  - An array acts as a list
  - Index provides ordering
    - First at location 0
    - Last at location n-1



- Set
  - Can use an array
  - Must check for duplicate on insertion
    - O(n) solution
  - Can we do better? Yes…

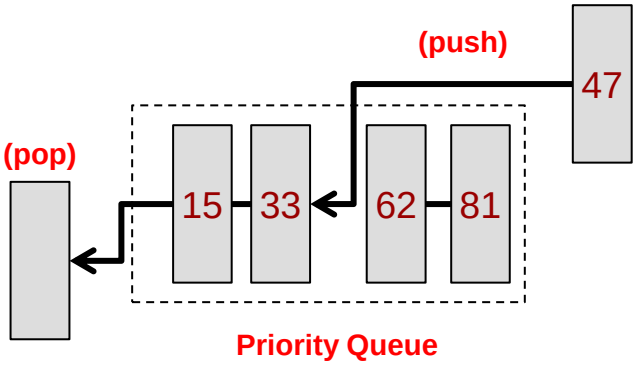


- Map
  - Can also use an array
  - Again check for duplicate key on insertion

```
struct Pair{
  string key;
  double value;;
};
```

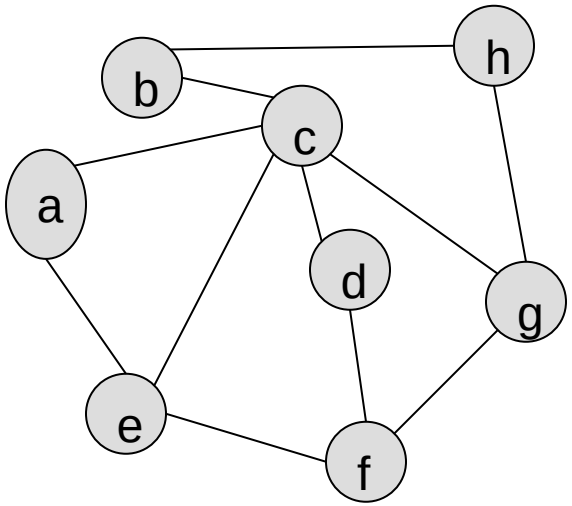
Priority Queue ADT

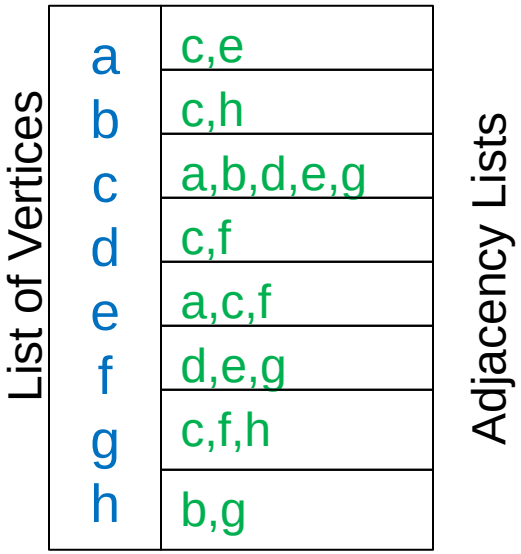
- Operations
  - Can add items in any order
  - Only allows retrieval of the "best/top" priority item (however "best" is defined: smallest, largest, etc.)
  - Only allows removal of the "best/top" item
- Can be stored as a "sorted" list
  - But there are more efficient implementations



Graph ADT

- Stores nodes (aka vertices) and edges between the nodes
  - Edges model relationships between vertices
  - Note: a "tree" is common form of a graph
- Can be stored as a list of lists or a _____





How else would you express this using the ADTs you've just learned?