

Abstractions, Collections & Data Structures



Nelson Padua-Perez
Bill Pugh

Department of Computer Science
University of Maryland, College Park

Abstractions and Collections

- **Programs represent and manipulate abstractions, or chunks of information**
 - a Sudoku board
 - the moves legal for a particular square
 - a media player
 - a deck of cards
- **The most universal of these is a collection**
 - lots of different kinds of collections
 - list, set, ordered set, bag, map
 - For each kind of collection, operations you can perform on them

Collections and Data Structures

- **A collection can typically implemented using several different data structures**
 - **a data structure is a way of implementing an abstraction and operations on it**
- **Different implementations or data structures for an abstraction will provide different efficiencies for different operations**
 - **we will see more of this in a moment**

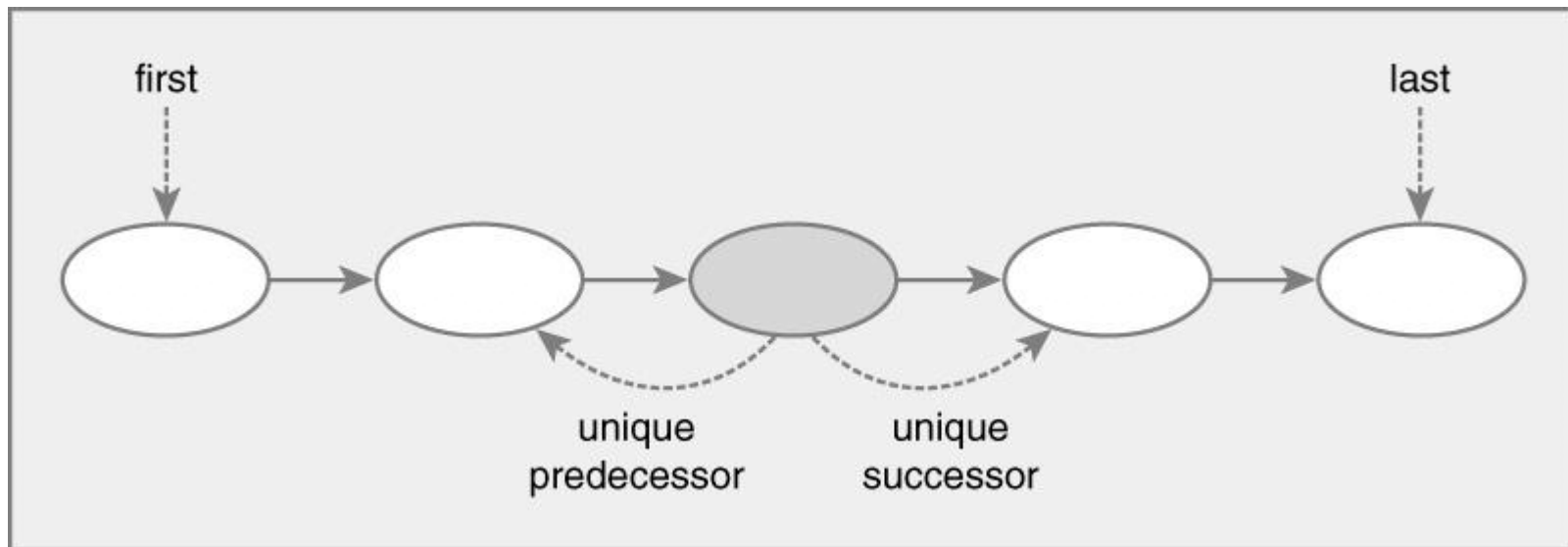
Collection

- **The base interface of most collections**
- **Operations supported:**
 - **add elements**
 - **remove elements**
 - **determine size of collection (# of elements)**
 - **iterate through elements**

Linear Data Structures

■ Total order over elements

- Each element has **unique** predecessor
- Each element has **unique** successor
- For any two distinct elements x and y , either x comes before y or y comes before x



Linear Data Structures

■ Core operations

- Find first element (head)
- Find next element (successor)
- Find last element (tail)

■ Terminology

- Head \Rightarrow no predecessor
- Tail \Rightarrow no successor

■ Duplicates allowed

- the same value can appear at different places in the list

Operations on lists

■ add an element

■ where?

- at beginning/front

- at rear/end

- after a particular element

■ remove an element

- remove first element

- remove last element

- remove the String value “Happy”

- what happens if “Happy” occurs more than once

Accessing elements

- **How do you name an element you want to manipulate**
 - **first/head**
 - **last/tail**
 - **by position (e.g, the 5th element)**
 - **also a bad movie**
 - **by walking/iterating through the next, and talking about relative position**
 - **the next element**
 - **the previous element**

Restricted abstractions

- **Restricting the operations an abstraction supports can be a good thing**
 - efficiently supporting only a few operations efficiently is easier
 - if a limited abstraction suits your needs, easily to reason about the limited abstraction than a more general one
- **Restricted list abstractions:**
 - queue (aka FIFO queue)
 - stack (aka LIFO queue)
 - dequeue (aka double ended queue)

Queue

- Can add items to the end of the queue
- remove them from the front of the queue
- First-in, first-out order (FIFO)
- Represented as a list
 - total order over elements
- But no access to middle of the list

- Example use:
 - songs to be played
 - jobs to be printed
 - customers to be served

Stack

- **Can add an item to the top of the stack**
- **can remove the item on top of the stack**
- **No access to elements other than the top**
- **Just a list where you can add/remove elements only at one end**
 - **whether the top of the stack is the front or beginning of the list doesn't really matter**
- **Examples:**
 - **keep track of pending calculations in a calculator**
 - **parsing**

Double ended queue

- **List/queue where you can add or remove at either end**

General list operations

- **get/set/insert/remove value at a particular index**
- **get/set/insert/remove value at head/tail**
- **iterate through list, and get/set/insert/remove values as you go**
 - **use a `java.util.ListIterator`**

List implementations

- **Two basic implementation techniques for any kind of list**
 - **store elements in an array**
 - **store each element in a separate object, and link them together (a linked list representation)**

Array implementations

■ Advantages:

- Space efficient (just space to hold reference to each element)
- Can efficiently access elements at any position

■ Disadvantages

- Has to grow/shrink in spurts
- Can't efficiently insert/remove elements in the middle
- inserting/removing elements at both ends is tricky

Linked implementation

■ Advantages

- can efficiently insert/remove elements anywhere

■ Disadvantages

- can't efficiently access elements at arbitrary positions
- space overhead (one or two additional pointers per element)