“Computer science is the science of abstraction.”

Abstract Data Type

- Abstraction of a data structure
- Operations on that data structure
- OO programming allows implementation details to be hidden from user/client
Examples Already Discussed

ADT: sorted list
■ Operations
  – op1
  – O( )
  – op2
  – O( )
  – op3
  – O( )

ADT: unsorted list
■ Operations
  – op1
  – O( )
  – op2
  – O( )
  – op3
  – O( )

Other ADTs in 281
■ Stack
■ Queue
■ Priority Queue
■ Tree
■ Graph
■ Hash table
■ ...

Definitions

Abstract Data Type: mathematical model of a data type and operations defined on the model that is accessed only through an interface

■ separate the conceptual transformations that our programs perform on data from any particular data structure representation and algorithm implementation
An abstract data type is realized by a concrete data structure, which is modeled in C++ by a class.
- Abstract class
- Concrete class

Foundational Data Structure
- Most ADTs are built upon some base data structure
- Base data structure is not viewed as an ADT itself, but an alternative implementation
- Foundational, base data structures come in two flavors:
  - arrays (vectors)
  - (singly) linked lists

Foundational Data Structure
- Logical sequential storage
  - a collection of \( n \) objects of some type \( T \)
  - each object in the list is called an "element"
- Must have:
  - insert a new element
  - remove a previously inserted element
- Typically have:
  - construct an empty list
  - count the number of elements
- May have:
  - destroy or copy the list
  - reverse one list, concatenate two lists, etc. etc.
Arrays in C++

- Problems with C++ array implementation
  - no bounds checking
  - 0 to n-1 rather than 1 to n
  - size static and fixed at compile time
  - ...

Better than Arrays: Vectors

- rank: number of elements before e in S
  - 1st ele is rank 0, last ele is rank n-1
- vector: linear sequence that supports access to elements by their ranks
- extendable: ‘automatic’ doubling of storage space when overflow condition is reached

Vector ADT

Operations
- size(): return number of elements in vector
- isempty(): return T if vector is empty, F otherwise
- elemAtRank(r): return the element at rank r
- replaceAtRank(r,e): replace the element at rank r with element e
- insertAtRank(r,e): insert a new element e at rank r
- removeAtRank(r): remove the element at rank r
Vector ADT: Array-based

Operations
- size(): $O(1)$
- isempty(): $O(1)$
- elemAtRank(r): $O(1)$
- replaceAtRank(r,e): $O(1)$
- insertAtRank(r,e): $O(1)$
- removeAtRank(r): $O(1)$

Vector ADT: Extension

When $n = N$ (size = capacity)
- allocate new array $B$ of size $2N$
- copy $A[i]$ to $B[i]$ for $i = 0...N-1$
- deallocate $A$, and let $A = B$

See code fragments 5.2 and 5.3 (esp. `insertAtRank` and `overflow`) for detail

Vector ADT: Extension

*Amortization:* although doubling size of vector seems expensive, cost is ‘paid’ over time so insertion of $n$ items is $O(n)$
Linked Lists in C++

- Sequence of dynamically allocated storage elements
- Each node has:
  - data member (called an element)
  - pointer to its successor
- Many variations
- Book uses pointer to head only

List ADT

Operations

- size(): return number of nodes in list
- isempty(): return T if list is empty, F otherwise
- elemAtPos(p): return the element at position p
- replaceAtPos(p,e): replace the element at position p with element e
- insertAfter(p,e): insert a new element e after position p
- insertBefore(p,e): insert a new element e before position p
- removeAt(p): remove the element at position p

List ADT

Operations

- size(): O( )
- isempty(): O( )
- elemAtPos(p): O( )
- replaceAtPos(p,e): O( )
- insertAfter(p,e): O( )
- insertBefore(p,e): O( )
- removeAt(p): O( )
Summary: Foundational Data Structures

- Arrays (Vectors)
  - `size()`
  - `isempty()`
  - `elemAtRank(r)`
  - `replaceAtRank(r,e)`
  - `insertAtRank(r,e)`
  - `removeAtRank(r)`

- Linked Lists
  - `size()`
  - `isempty()`
  - `elemAtPos(p)`
  - `replaceAtPos(p,e)`
  - `insertAfter(p,e)`
  - `insertBefore(p,e)`
  - `removeAt(p)`

Wrap-up Questions

- Name advantages of using arrays
- Name advantages of using linked lists
- How do you implement a multi-dimensional array?
- How do you implement a sparse matrix?
- What's the advantage of using a tail pointer?
- What's an advantage of using a doubly linked list over singly linked list?

Common Bugs

- Arrays
  - Index variable is not initialized
  - Bounds not checked
  - Memory leak (pointer to array elements not de-allocated)
  - When removed, pointer to array elements not removed

- Linked List
  - Head and tail not initialized to NULL
  - Accessing de-allocated element
  - Memory leak (de-allocation doesn’t walk down list)
  - Extraction or insertion loses pointer, e.g. delete last element, but `prev -> next` not set to null