Introduction to Computer Science

Data Abstractions

Dr. Antonio Siochi

Outline

- Data Structure Fundamentals
- Implementing Data Structures
- Customized Data Structures

Data Structure Overview

- array
- list
- stack
- queue
- tree

Organization Chart

Tree Terminology
Antonio Siochi
Slide 7

**Pointers**

Antonio Siochi
Slide 9

**Outline**

- Data Structure Fundamentals
  - Implementing Data Structures
  - Customized Data Structures

Antonio Siochi
Slide 10

**Arrays**

- group of cells, in which each cell is directly accessible
- conceptual shape of 1 dimensional array is a sequence of cells
- matches the “actual” structure in memory: sequence of cells
- programmer does not need to know the actual shape and it’s easier to deal with the conceptual shape

Antonio Siochi
Slide 11

**Translate conceptual shape**

- let Readings be an array of temperature readings taken each hour for 24 hours
- Readings[4] is the temperature reading at the 4th hour

Antonio Siochi
Slide 12

**2 dimensional Arrays**

- constant size and shape: fixed number of columns and rows
- like a spreadsheet: element is at the intersection of a row and a column
- SALES is a 4 row, 5 column array starting at address 10
- SALES[1,1] is the entry in the first row, first column it is located at address 10 (SALES[0][0] in C++)
**Row Major Order**
- address of $\text{SALES}[i,j] = \text{START} + C \times (i-1) + (j-1)$
- $C$ is number of columns

**Abstraction helps**
- if we did not have languages which let us write $\text{SALES}[3,4]$, we would have to perform the 2d address calculation ourselves
- 2d array is an abstraction in which the actual address calculation is hidden
- we can thus focus on the problem at hand, instead of the implementation DETAILS

**Lists**
- arrays are fixed in size and shape
- some applications (such as add/drop procedure for classes) require data structures which can grow or shrink on demand
- lists are like arrays in that they are a group of elements in a particular sequence
- however, lists can grow or shrink

**Example List**
- represent a list as a sequence of elements, separated by commas
  - shopping-list = (apples, fish, rice, soap)
- or draw a picture, where each element is enclosed in a box, and arrows show which element is next

**Contiguous List**

**Pointers**
- each element in the list points to the next element
- this is the conceptual or abstract level of a list
- how are pointers implemented in memory?
- instead of storing data, store the address of data
- example: PC register holds address of next instruction
- pointer is a memory cell that holds the address of another memory cell
**Pointer Implementation**

<table>
<thead>
<tr>
<th>CONCEPTUAL</th>
<th>ACTUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>0000</td>
</tr>
<tr>
<td>23456</td>
<td>0004</td>
</tr>
<tr>
<td>34567</td>
<td>0002</td>
</tr>
</tbody>
</table>

**Linked Lists**

- blocks of memory, each with a data portion and a pointer to the next block of memory
- need a special pointer to show where the first element of the list is: head pointer
- need a special address which indicates emptiness NIL pointer indicates there is nothing next to this cell (it’s the last cell)

**Structure of a Linked Lists**

- Head pointer
- Name
- Pointer
- Name
- Pointer
- NIL

**Deleting a cell from a list**

- Head pointer
- Name
- Pointer
- Old pointer
- Name
- Pointer
- New pointer
- NIL
- point to next node

**Inserting a cell into a list**

- create the cell and make it point to what it should precede
- make the cell that should come before it point to the new cell

**List Abstraction**

- composed of a sequence of elements
- procedures to manipulate those elements:
  - insert element into list
  - delete element from list
  - search for an element in a list
  - print all elements in the list
**Implemented as Software**

- List – variable used to hold the list (points to it)
- Item – variable used to hold an element of the list
- Programmer can just use these routines and not worry about internal implementation of list
  - insert (item, List)
  - delete (item, List)
  - search (item, List)
  - printList (List)

**Stacks**

- How would you print a list of names in reverse?
- Like a list, except that you can only insert or delete from a single end
- Things you put in first are at the bottom of the stack
- At the top is the LAST thing you put in

**Example**

- Useful when you need to remember a sequence of things in REVERSE order
- Calling procedures and returning from them
- How do you keep track of which procedure to return to?

**Stack of invoked procedures**

```
Procedure returning order

Main program Procedure A Procedure B Procedure C

Call A \rightarrow Call B \rightarrow Call C \rightarrow

Procedure invoking order

Return from C \rightarrow Return from B \rightarrow Return from A \rightarrow

main \rightarrow A \rightarrow main

A \rightarrow main

B \rightarrow A \rightarrow main

C \rightarrow B \rightarrow A \rightarrow main
```

**Stack Implementation**

- Use array to hold the elements of the stack
- Keep a variable called the stack pointer, which holds the index in the array of the top element
- Push: increment the stack pointer to the next vacant array element, then insert the new element at that spot
- Pop: decrement the stack pointer to the next element down on the stack

**Stack in Memory**

```
Stack's base

Reserved block of memory cells

Stack entries

Stack's pointer

Space for growth
```
Stack Abstraction

- use variable to hold the elements in the stack
- access elements using only these procedures
  - push (element, Stack)
  - pop (element, Stack)
  - isEmpty (Stack)

Push and Pop

- PUSH
  - main ➔ A ➔ main
- POP
  - main ➔ main ➔ main

Queues

- like a list, but insert only at one end, and delete from the other
- useful for remembering elements in first-come first-served order
- things put in first are at the FRONT (HEAD) of the queue
- things put in last are at the REAR (TAIL) of the queue
- elements enter the REAR and exit from the FRONT

Implementation of Queue

Queue “crawling” through Memory

Circular Queue
Outline

- Data Structure Fundamentals
- Implementing Data Structures
- Customized Data Structures

Abstract Data Types (ADTs)

- ADT = data structure + valid procedures
- variable declaration:
  - x is a variable of type integer
    - allocate 2 bytes for the variable
    - allow only integer operations on x
- the primitive type integer is an ADT
- we can create new ADTs by combining existing ADTs
ADTs and Instances

- Create a LIST ADT from an array data type and provide a set of access procedures.
- Now you can create an INSTANCE of that LIST ADT by declaring a variable to be of type LIST.
- The primitive type INTEGER is an ADT, and we create an INSTANCE of an integer ADT by declaring a variable to be of type integer.
- ADT is the blueprint, INSTANCE is a building.
- ADT is the cookie cutter, OBJECT is the cookie.

ADTs and Encapsulation

- ADTs hide the implementation mechanism.
- You should not access the data structure of an ADT instance without using the access procedures.
- Some languages don’t prohibit direct access to the data structure of an ADT instance.
- The situation is similar to using the reserve section of a library (shelves = data structure, librarian = access procedure).

Encapsulation and Coupling

- Violation of encapsulation leads to greater data and possibly control coupling.
- Increased coupling means less modifiability.
- Other team members expect encapsulation and so modify the implementation of the access procedures or data structures.
- If your program is dependent on the old implementation or data structure, your program will cease to function correctly.

Encapsulation and Building Blocks

- Properly designed and encapsulated ADTs can extend the functions of a programming language.
- Standard C++ has no built-in string data type.
- If you design a string ADT and implement it as a software package, then anyone can use your software package to write C++ programs that manipulate strings.