(1 - 2) Introduction to C Data Structures & Abstract Data Types

Instructor - Andrew S. O’Fallon
CptS 122 (August 25, 2017)
Washington State University
What is a Data Structure?

- A software construct describing the organization and storage of information
  - Designed to be accessed efficiently
  - Composite of related items
- An implementation of an abstract data type (ADTs) to be defined later
- Defined and applied for particular applications and/or tasks
Data Structures Exposed

- You’ve already seen a few fixed-sized data structures
  - Arrays
  - Structures or structs in C
Review of Basic C Data Structures (1)

- Recall an *array* is a collection of related data items
  - Accessed by the same variable name and an index
  - Data is of the same type
  - Items are contiguous in memory
  - Subscripts or indices must be integral and 0 or positive only

- Our visual representation of an array of chars, where first row is index and second is contents

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>...</th>
<th>n-2</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>‘b’</td>
<td>...</td>
<td>‘3’</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>
Review of Basic C Data Structures (2)

- Recall a *structure* or *struct* is a collection of related fields or variables under one name
  - Represent real world objects
  - Each field may be of a different data type
  - The fields are contiguous in memory
- Example struct describing a dog
  ```c
  typedef struct dog
  {
      char *breed; // need to allocate memory for
      // string separately
      char name[100]; // memory is included for string
      double weight;
  } Dog;
  ```
How Can We Expand on Our Data Structure Knowledge?

- In this course we will focus on dynamic data structures
  - These grow and shrink at runtime
- The major dynamic data structures include:
  - Lists
  - Stacks
  - Queues
  - Binary Trees
  - Binary Search Trees (BSTs)
Basic Applications of Dynamic Data Structures (1)

- Lists are collections of data items lined up in a row
  - Insertions and deletions may be made anywhere
  - May represent movie & music collections, grocery store lists, & many more…

- Stacks are restricted lists
  - Insertions and deletions may be made at one end only
    - These are Last In, First Out (LIFO) structures
  - May be used with compilers & operating systems, & many more applications…
Basic Applications of Dynamic Data Structures (2)

- Queues are also restricted lists
  - Insertions are made at the back of the queue and deletions are made from the front
    - These are First In, First Out (FIFO) structures
  - May represent waiting lines, etc.

- BSTs require linked data items
  - Efficient for searching and sorting of data
  - May represent directories on a file system, etc.

- This course will focus on these dynamic data structures and corresponding implementations in both C and C++
What do these C Dynamic Structures have in Common?

- Of course dynamic growing and shrinking properties...
- Implemented with pointers
  - Recall a *pointer* is a variable that stores as its contents the address of another variable
    - Operators applied to pointers include
      - Pointer declaration – i.e. char *ptr
      - Dereference or indirection – i.e. *ptr
      - Address of – i.e. &ptr
      - Assignment – i.e. ptr1 = ptr2
      - Others?
- Require the use of structs
  - Actually self-referential structures for linked implementations
What is a Self-Referential Structure?

- A struct which contains a pointer field that represents an address of a struct of the same type

Example

```c
typedef struct node
{
    char data;
    // self-referential
    struct node *pNext;
} Node;
```
The growing and shrinking properties may be achieved through functions located in `<stdlib.h>` including:

- `malloc()` for allocating/growing memory
- `free()` for de-allocating/shrinking memory
- `realloc()` for resizing memory
- Also consider `calloc()`
Dynamic Memory Allocation / De-allocation in C (2)

- Assume the following:
  Node *pItem = NULL;
- How to use malloc()
  pItem = (Node *) malloc (sizeof (Node));
  // Recall malloc () returns a void *,
  // which should be typecasted
- How to use free()
  free (pItem);
  // Requires the pointer to the memory to be
  // de-allocated
- How to use realloc()
  pItem = realloc (pItem, sizeof (Node) * 2);
  // Allocates space for two Nodes and
  // returns pointer to beginning of resized
  // memory
How Do We Know Which Values and Operations are Supported?

- Each data structure has a corresponding model represented by the abstract data type (ADT)
  - The model defines the behavior of operations, but not how they should be implemented
Abstract Data Types

- Abstract Data Types or ADTs according to National Institute of Standards and Technology (NIST)
  - Definition: A set of data values and associated operations that are precisely specified independent of any particular implementation.
Data Structures according to NIST

Definition: An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the list or number of nodes in a subtree.
ADTs versus Data Structures

- Many people think that ADTs and Data Structures are interchangeable in meaning
  - ADTs are logical descriptions or specifications of data and operations
    - To abstract is to leave out concrete details
  - Data structures are the actual representations of data and operations, i.e. implementation
- Semantic versus syntactic
Specification of ADT

- Consists of at least 5 items
  - Types/Data
  - Functions/Methods/Operations
  - Axioms
  - Preconditions
  - Postconditions
  - Others?
Example Specification of List ADT (1)

- **Description**: A list is a finite sequence of nodes, where each node may be only accessed sequentially, starting from the first node.
- **Types/Data**
  - e is the element type
  - L is the list type
Example Specification of List ADT (2)

- **Functions/Methods/Operations**
  - **InitList** (L): Procedure to initialize the list L to empty
  - **DestroyList** (L): Procedure to make an existing list L empty
  - **ListIsEmpty** (L) -> b: Boolean function to return TRUE if L is empty
  - **ListIsFull** (L) -> b: Boolean function to return TRUE if L is full
  - **CurIsEmpty** (L) -> b: Boolean function to return TRUE if the current position in L is empty
Example Specification of List ADT (3)

- **Functions/Methods/Operations Continued**
  - **ToFirst (L):** Procedure to make the current node the first node in L; if the list is empty, the current position remains empty
  - **AtFirst (L) -> b:** Boolean function to return TRUE if the current node is the first node in the list or if the list and the current position are both empty
  - **AtEnd (L) -> b:** Boolean function to return TRUE if the current node is the last node in the list or if the list and the current position are both empty
  - **Advance (L):** Procedure to make the current position indicate the next node in L; if the current node is the last node the current position becomes empty
Example Specification of List ADT (4)

- **Insert** \((L, e)\): Procedure to insert a node with information \(e\) before the current position or, in case \(L\) was empty, as the only node in \(L\); the new node becomes the current node.
- **InsertAfter** \((L, e)\): Procedure to insert a node with information \(e\) into \(L\) after the current node without changing the current position; in case \(L\) is empty, make a node containing \(e\) the only node in \(L\) and the current node.
- **InsertFront** \((L, e)\): Procedure to insert a node with information \(e\) into \(L\) as the first node in the List; in case \(L\) is empty, make a node containing \(e\) the only node in \(L\) and the current node.
- **InsertInOrder** \((L, e)\): Procedure to insert a node with information \(e\) into \(L\) as node in the List, order of the elements is preserved; in case \(L\) is empty, make a node containing \(e\) the only node in \(L\) and the current node.
Example Specification of List ADT (5)

- Functions/Methods/Operations Continued One Last Time
  - **Delete (L):** Procedure to delete the current node in L and to have the current position indicate the next node; if the current node is the last node the current position becomes empty
  - **StoreInfo (L,e):** Procedure to update the information portion of the current node to contain e; assume the current position is nonempty
  - **RetrieveInfo (L) -> e:** Function to return the information in the current node; assume the current position is nonempty
Example Specification of List ADT (6)

- **Axioms**
  - Empty ()?
  - Not empty ()?
  - Others?

- **Preconditions**
  - Delete () requires that the list is not empty ()

- **Postconditions**
  - After Insert () is executed the list is not empty ()

- **Others?**
Visual of List ADT

- View diagrams on the board
  - Nodes?
  - List?
Next Lecture…

- Introduction to implementation of a dynamically linked list
References

Collaborators

- Jack Hagemeister