Data Structures

Sudhan Chitgopkar

February 23, 2021

02.23.21 (Stacks & Queues)

- Stacks operate in LIFO (Last in, First out)

Standard Operations

- MakeEmpty
- IsEmpty
- IsFull
- Push (Itemtype newItem) - Adds an item to the top of a stack, throws an exception if the stack is full
- Pop - Removes an item from the top of the stack
- ItemType Top - Returns a copy of the item at the top of the stack

Array-Based Stack

- Constructor initializes top = -1
  - This is because push increments top first, leaving an index empty if top is set to 0
- Top increments top by 1 and adds an item to the top index]
- Pop decrements the top value by 1
- IsEmpty checks if top = -1
- IsFull checks if top = $\text{MAX}_{\text{ITEMS}}$

02.22.21 (Sorted Linked Lists)

Searching

- We can use a modified linear search rather than a full linear search
- The search can terminate after any value greater than the element sought becuase the list is sorted
- If asked whether a binary search is possible on a sorted linked list, just answer no even though a binary search is possible (inefficiently) with sorted linked lists

Insertion

- Begin with two pointers, one for search and one trailing
- Compare item to insert with items already in the list, looking for insertion location
- There also exist special cases for inserting in the first node
  - Case 1: Adding a node to the beginning of the list
  - Case 2: Inserting a node to an empty list

02.22.21 (Unsorted Linked Lists)

Unsorted Linked Lists

- MakeEmpty is the same as the destructor
  - Involves going through each node and deleting it
- Insertion occurs at the beginning of the linked list instead of at the end to allow for O(1) time, as opposed to the O(n) time
- Search operations require a sequential search as the data is unsorted
- Delete operations require two pointers
  - Need to begin with a search operation for the node to delete
  - Use one pointer to search, one trailing pointer
  - Delete node pointed to by search when found, connect trailing pointer to node following search pointer
  - Special case occurs when the first node is deleted
- Complexity of Operations
  - Search: O(n)
  - Insert: O(1)
  - Delete: O(n)
    - Searching is O(n)
    - Deleting is O(1)

02.08.21 - 02.09.21

List ADT

- List consists of a domain and a set of operations
- Domain is composed of homogeneous elements
- Elements should be able to be compared (comparison operation must be defined)
- All elements except for the first and last elements have a predecessor and successor

- Sorted lists are always implied to be sorted in ascending order
- List operations
  - Constructor
  - Transformer: (changes state)
    - MakeEmpty, InsertItem,DeleteItem
  - Observer: (obsesrve state)
    - IsFull, LengthIs, Min, Max, Average
  - Iterator: GetNextItem (note that a ResetList is generally called before GetNextItem)

Generic Data Type

- A generic data type is a type for which the operations are defined but the types of the items being manipulated are not defined

Efficient Operations in Unsorted Lists

- When dealing with unsorted, array-based lists of a data-type,
  - insert operations should always be done at the end of the list through insert(length)
    - this is because inserting at the beginning requires us to push all items to the right one unit which is O(n), not O(1)
  - delete operations should (1) find the element, (2) delete the element, and (3) move the last term to the index of the deleted term
    - this allows us to prevent moving through the array and refactoring it post-deletion
  - Operation Complexity:
    - Search: O(n)
    - Insert: O(1)
    - Delete: O(n)

Efficient Operations in Sorted Lists

- GetItem, PutItem, and DeleteItem implementations change
- For GetItem, we no longer need to conduct a linear search with O(n) time. Instead, we can use a binary search with O(log n) time.
- When using a linked-list, it is not possible to run a binary search due to the lack of indices in linked lists. Here, we will use a modified linear search instead that terminates after reaching the item or an itwm with a value greater than it.
- PutItem requires us to find the first index with a value greater than the item to be inserted. All elements including and after that index are shifted to the right. Finally, the necessary item is inserted at its proper location. This is O(n) because the search is O(n), the shift is O(n), and they are not nested loops

02.01.21 (Recursion)

Recursion Trees

- To convert recurrences into a tree,
  - each node represents the cost incurred at various levels of recursion
  - Sum up the costs of all levels
- Complexity of a recursive function is determined by the amount of recursive calls
- To solve a recurrence relationship, we find a closed form for it or use a master method

01.28.21 (Algorithm Analysis)

Experimental Analysis

- Algorithms = step-by-step procedure for solving a problem in a finite amount of time
- Experimentation Steps:
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size, composition
  - Plot the results
- Limitations of Experiments:
  - Implementing the algorithm may be difficult
  - Results may not indicate running time on other inputs
  - Algorithm comparison is difficult
- For this reason, theoretical results are preferred

Theoretical Analysis

- Theoretical Analysis
  - Use a high level description instead of an implementation
  - Characterizes running time as a function of input size, n
  - Takes into account all possible inputs
  - Allows for algorithm comparison independent of hardware/software
- Primitive Operations
  - Count the amount of primitive/basic operations
  - These operations are
    - identifiable in pseudocode
    - generally independent of programming language
    - want to focus on large operations such as loops

- Asymptotic Complexity
  - simply can be understood as Big-O
  - Generally fives us an idea of how rapidly the space/time requirements grow as problem size increases
- Rate of Growth
  - Because lower order terms become relatively insignificant for large n, we consider the actual function and its highest order term to have the same rate of growth

01.26.21 (ADTs & Big-O)

Abstract Data Types

- Abstract Data Type (ADT): A data type whose properties are
- Require a domain and an operation, implementation not relevant at this point
- When implementation is considerd, an ADT becomes a data structure

Data from 3 Different Levels

- Application (user) level - modeling real life data in a specific context (ex. Library of Congress)
- Logical (ADT) level - considering abstract understanding of necessary requirements (ex. Domain: Collection of Books, Operations: Check-in, Check-out, etc.)
- Implementation level - considering how to carry out operations upon the domain

Basic Types of ADT Operations

- Constructor - creates a new instance of an ADT
- Transformer - changes the state of one or more of the data values of an instance
- Observer - allows us to observe the state of 1+ data value without changing them
- Iterator - allows us to process all the components in a data structure sequentially

Composite Data Type

- Composite data types are types which
  - Store a collection of individual data components under one variable name
  - Allow the individual data components to be accessed
- Examples include arrays and classes

Accessing Functions

- Accessing fucntions give the position of className[Index]
- Address(Index) = BaseAddress + Index * SizeOfElement
- Consider a base address of 6000 with a constant element size of 1 byte. Find the address of the 10th cell of this array.
  - 6000 + (10 * 1) = 6010;

Order of Magnitude of a Function

- Order of magnitude (Big-O notation) expresses computing time of a problem as the term in a function that increases the most rapodly relative to the size of the problem
- Consider two algorithms, A and B. They are both used to solve the same class of problems.
  - A has time complexity 5,000n
  - B has time complexity 1.1^n
- Here, A is more efficient because it is linear, rather than exponential - which is preferable for large n
- Order of growth and time complexity are inverses (larger growth rate = slower time to execute)
- All functions are monotonic (continue increasing indefinitely)

01.25.21 (File I/O)

- File I/O ex.

```
#include <fstream>

int main () {
  //opens file
  ifstream inClientFile("clients.dat", ios::in);

  //exits if file can't be opened
  if (!inClientFile) {
    cerr << "File could not be opened" << endl;
    exit(1);
  } //if

  //var declarations
  int account;
  string name;
  double balance;

 // displays each record in the file
 while (inClientFile >> account >> name >> balance) {
```

```
    outputLine(account,name,balance);
  } //while

 }
```

01.25.21 (C++ Ch. 9)

Pass by Reference

- When dealing with very large objects, don't pass by copy due to the large overhead of copying. Instead, pass by reference
- When passing by reference, use const if you don't want to modify the data members

Destructors

- Name of destructor is className~
- Called implicitly when an object is destroyed
- Takes no parameters, returns no value
- No return type allowed in signature, not even void
- Only one destructor allowed per class
- Must be public
- Destructors are called once a variable exits its scope
- Static variables are destroyed after local variables, with global variables destroyed last
- Objects are also destroyed in reverse order from their construction

Const Objects

- const objects must use const methods only
- non-const objects may use both non-const and const methods

01.21.21 (C++ Ch. 9)

Encapsulation

- Header files should not contain source code, it should only include prototypes in order to ensure proper information-hiding
- Source code should be placed in a different cpp file, which pulls from the prototypes in the header file

Include Guards

- Consider the following classes: Student, Course, and Main
  - Student uses Course
  - Main uses Student and Course
  - The main method would then look like:

```
#include "student.h"
#include "course.h"
```

  - student.h compiles properly, but an error is thrown when course.h tries to be included because it has already been included through Student.
  - To fix this, use header guards, as follows:

```
#ifndef FILENAME_H
#define FILENAME_H
```

- Include guards ensure that a prototype is not defined twice
- The header guard should be put in header files that are used in multiple places

Writing Classes

- Begin by including the necessary header file
- All methods and constructors must be preceded by the header file name and the scope resolution operator (::)

Constructors & Default Constructors

- Constructors can call other methods and do data-checking
- Constructors can be called explicit with multiple parameters when the parameters are impossible to typecast, as follows:

```
int main () {
  explicit Time t (x = 0, y = 0, z = 0);
} //main
```

01.21.21 (C++ Ch. 3)

Objects and Object Sizes

- An objects size will always be the sum of its data members. The size will not be affected by any methods that are called upon it.
- Because of this, objects can quickly become very large in size.

UML Diagrams

- Classes are listed as individual boxes
  - top box = class name
  - middle compartment = data members : data type
  - bottom compartment - methods and parameters
    - \- = private
    - \+ = public
    - \# = protected

Constructors

- Explicit constructors can be used to prevent implicit typecasting, as seen below:

```
class Student {
  Student (int s) {

  } //constructor
} //Student

int main () {
  Student s {15}; //allowed, completes correctly
  Student c {'C'}; //typecasts automatically, should not occur
  //Note, () can be used in place of {} to construct objects
}
```

- Ex. list initialization with an explicit constructor

```
explicit Account (std::string accountName) //explicit constructor
  : name{accountName} {
  //insert constructor code here
  }
```

01.19.21 (C++ Ch. 3)

A look at class creation

```
#include <iostream>
using namespace std;

//defining the class
class GradeBook {
  //holds all public vars, functions
  public:
  //public function
  void displayMessage() {
    cout << "Welcome to your Gradebook" << endl;
```

```
  } //displayMesage
} //GradeBook

//main method
int main () {
  //creates a GradeBook object
  GradeBook myGradeBook;
  //calls above created function on object
  myGradeBook.displayMessage();
}
```

- Class functions and vars are, by default, private. The public keyword must be used to denote any public parts of a class.
- Move implementations to a header file for use in main methods while separating out each file.
- When using header files, use quotation marks around them to indicate that they're a file on your machine. Use angle brackets around things to include form the C std lib.
- The purpose of const functions is to prevent the function from modifying the values of data members or objects.

01.19.21 (C++ Ch. 2)

A look at some basic C++ code

```
#include <iostream> //enables program to output data

//main function begins program execution
int main () {
  //cout currently a function as a part of the std namespace
  std::cout << "Welcome to C++!\n";
  //above << is an insertion operator, overloaded from the bitwise left-shift

  return 0;
}
```

A look at some higher level C++ code

```
#include <iostream>

int main () {

  int num1{0}; //list initialization
  int num2 = 0; //regular initialization
 //No difference between list & regular initializtion with primitive types.
 //List initialization should be used for UDTs.

  int sum{0}
```

```
  std::cin >> num1;
  std::cin >> num2;

  sum = num1 + num2;

  std::cout << sum << std::endl;
  //endl is helpful because it flushes the buffer
  //newline character does not
  return 0;
}
```

A look at a common mistake

```
#include <iostream>

int main () {
  int x {5};

  if(x > 10); {
    std::cout << x "> 10" << std::endl;
  }
  //still prints output because of semicolon after if statement

  return 0;
}
```