CS / IS C363 Data Structures & Algorithms

Course Motivation

Administrivia

Introduction: Data Abstraction

Data Modeling

Abstraction

Data Abstraction, Representation

Abstract Data Types

Courtsey: Sundar B Slides 2013

Course Motivation

- Solving Problems
  - Requires writing Programs (“Concrete solutions”)
  - A program typically solves one specific problem i.e. for a class of inputs
  - Solution may depend on specific platform
- Writing Programs
  - Requires designing Algorithms (“abstract solutions”)
  - An algorithm may solve a class of problems
  - Solution will not depend on specific language/platform
- Designing Algorithms
  - Requires organizing (i.e. structuring) and representing (i.e. storing) data
  - such that algorithms can effectively access and use them

Administrivia – Semester Plan & Evaluation

- 3 lectures and 1 lab (3 hours) per week
- All Evaluation components are open book
- 1 Mid-Term Test (40 marks i.e. 20%)
- Comprehensive Exam (60 marks i.e. 30%)
- Quizzes – 2 (2 x 10 = 20 marks i.e. 10%)
- Lab sessions – (Total 80 marks i.e. 40%)
  - Labs are open book
  - Structured as – in alternate weeks –
  - No Weightage for weekly labs
  - Attending is your duty.

ADMINISTRIVIA - LABS

- Lab Sessions:
  - Focus on implementation of algorithms
  - Implementation Techniques
  - Performance Evaluation of algorithms / implementations
  - Emphasis on completion
  - Marking will depend on executable (parts of the) code:
    - **Advice: Learn incremental development !**

Data Modeling

- Solving Problems <== Writing Programs <== Designing Algorithms <== Structuring of data <== Understanding of data
- What kind of data?
  - Input or Output data
  - e.g. Census Records to (Aadhaar) Unique IDs
    - List/Set, Tuples, Ordering, and Keys
  - Computational data
  - “Undo” operations in a word processor (or editor or game)
    - What kind of data must be remembered?
    - How do your organize the data?
      - Last-in-First Out List of operations

Data Modeling

- Goal:
  - Understand Data (related to the problem being solved) and capture essentials as a model
- Purpose of Modeling:
  - Abstraction (that leads to) Design
  - e.g. architect’s model in clay / wood, or blueprints
- Principles/Techniques:
  - Capture essentials
  - Ignore details (that are irrelevant)
  - E.g. History in command shells (Unix/Linux)
    - Should a list of commands be enough?

Data Modeling

- Model: components and attributes
  - Factors: Shape, Size and Ordering
  - E.g. Census Record per person: What fields should it contain?
  - E.g. Census List: Should it be ordered? By what attribute? Is there a unique attribute i.e. a key?
  - E.g. History List in Shell: Should it remember all the “past” commands?
  - E.g. “Undo” List in Editor: Should I be able to “undo” everything I have done?

Data modeling

- E.g. History in command shells (Unix/Linux)
  - uses: previous command; argument of previous command
  - (Reverse) Chronological order is essential
  - Model as LIFO list of commands
  - Exact time of the command is an irrelevant detail
  - Do not include in the model
  - How the LIFO list is implemented is a(n implementor's) choice
  - Do not include in the model

Data Modeling

- Model: Behavior
  - Factors: State, Change of state, Lifetime
  - E.g. History list in editor:
  - State (memory): past commands
    - Model (getting the) state as a *top* operation
  - Change: a new command that is executed
    - Model this as a *push* operation
  - Lifetime: Decided by a specific configuration
    - e.g. Last 100 commands – limit configured by administrator
      - Model this by including a boundary check (size <= 100)
    - e.g. All commands in current “login” session
      - Model this by a *clear* operation

Data model

- Model the data in examples (History, Undo) as a Stack:
  - A list that is (reverse) chronologically ordered i.e. LIFO list
  - Behavior:
  - Get the last element: Element top(Stack s)
  - Remove the last element: Stack pop(Stack s)
  - Add a new element: Stack push(Stack s, Element e)
  - Find whether the list is empty: isEmpty(Stack s)
  - Create a new stack: Stack newStack()
  - Properties (for unbounded Stack):
  - isEmpty(newStack()) == TRUE
  - isEmpty(push(s,e)) == FALSE
  - top(push(s,e)) == e
  - pop(push(s,e)) == s
  - Exercise: Adapt this for Bounded Stack

Modularity

- Modular Design:
  - Enables separate modules to be “implemented” independently
  - Enables modules to be replaced (i.e. pluggable) independently
  - E.g. Parts of an automobile: Engine, wheels and axle, body/doors, air-conditioner
- Separation of Concerns:
  - Principle underlying “Modular” Product Design:
  - Separate modules should address separate concerns
  - Why?
    - E.g. “engine and air-conditioner” is not a module

Modularity

- Information Hiding:
  - “Separation of concerns” as applied to software design
  - A module should hide information that is not relevant for its use:
  - i.e. users of a module need know only what is required for using (that module)
  - Information hiding separates concerns of user from that of provider (implementor)
  - E.g. What should the designer of a control system for an automobile know about the air-conditioning?
    - Should know: the (temperature) settings and power requirements
    - Need not know: the coolant used

Data abstraction

- An abstraction is a perspective:
  - You (choose to) see some features and ignore (omit) other features of the same entity
  - E.g. a blueprint or a clay model
- Data Abstraction:
  - Modular Design Principle for software – particularly for modules that organize data
  - Separate

model interface

  - user concerns (type of data, observable behavior) from
  - provider concerns (representation of data, implementation of that behavior)
  - How do you achieve this?
  - “Encapsulate” data

Data abstraction

- E.g. Stack
  - Observable behavior (i.e. interface):
  - *isEmpty, push, pop, top* with the properties listed earlier
  - User (in our case, word processor/editor, command shell) should see only these operations.
  - Hence the stack may be represented, say, either
    - as a contiguous chunk of memory (i.e. an array), or
    - as a linked list
    - and the operations are implemented accordingly.

From modularity to data abstraction

- Modular Design has development/maintenance benefits
  - Usually achieved by “separation of concerns”
- Information hiding
  - Principle for modularity in software that says

    a module should hide information irrelevant for its use

    i.e. *separate user information from provider information*
- Data Abstraction
  - Principle specific to software modules for storing/retrieving data that says

expose data model (type) and operations (interface) but hide representation and implementation information

Data representation

- Choice of representation is important.
- Representation should be chosen based on the desired set of operations :
  - Are the operations feasible with a given representation?
  - Can they be easily implemented?
  - Can they be efficiently implemented?
- E.g. Natural numbers:
  - Representation: English, Roman numerals, Arabic numerals

Types

- Types classify values:
  - E.g. Taxonomies in Biology
  - Useful for abstract understanding and reasoning
  - E.g. Given Platypus is a mammal
    - Valid reasoning: a platypus does not lay eggs
- Data types classify data values:
  - ***int***, ***char***, ***bool*** …
  - Useful for reasoning as well as for implementing such reasoning
  - e.g. ***int x; int y;*** …. ***x + y*** ….
  - ***x + y*** is an ***int*** value can be inferred.
  - e.g. ***float x; int y;*** …. ***y = x*** …
  - Compiler can identify/prevent (by type checking) such assignments

Data Types

- A (data) type is a set of values
  - grouped on the basis of a common set of operations and hence, typically,
  - implemented using a common representation
  - E.g.
    - int =def { -2k-1,…,-1, 0, 1,…2k-1-1 }
  - operations: { +,-,/,*,% }
  - representation: k bit 2’s complement

Structured Data Types

- Programming languages allow programmers to create structured data types:
  - e.g. struct in C: sets of tuples (i.e. cartesian products)
  - The common set of operations (e.g. get or set a field) and the common representation (e.g. contiguous locations) are decided by the language designer and/or compiler implementor.