Today's Agenda

1. Data-driven Programming
2. Data Abstraction

Personal Process for Programming

- Design ✓
  - Algorithm Design, Correctness Issues, Efficiency.
- Development ✓
  - Modularity and Reuse
- Testing (Will be discussed along with other topics)

Course Agenda

- Module 1 ✓
  - Personal Process for Programming
- Module 2
  - Data-Driven Programming – Basics:
    - Motivation
    - Data Abstraction
    - Linear Data

Today's Agenda

(1) Data-driven Programming

(2) Data Abstraction

Data Abstraction

- Abstraction
  - Provide a view (i.e abstract) by hiding details
  - Data Abstraction
    - Expose operations (View)
    - Hide representation

Data Representation

- Recall the example of numbers
- First attempt:
  - “One”, “Two” … “Dozen”, “Gross”
  - “Dozen” x “Dozen” = “Gross”
  - Operations were not systematic

Data Representation [2]

- Second attempt:
  - I, II, III, IV, V, …, X, XI …
  - X * X = C
  - XX * XX = CCCC
  - Operations were slightly better.

Data Representation [3]

- Finally:
  - 1, 2, …, 10,11, …
  - Operations are systematic – I.e. algorithmic
  - We can teach them to Martians (syntactically!)

Data Representation [4]

- Lesson:
  - Data Representation must be decided based on operations!
- List Data:
  - Operations: Addition, Deletion, Search (Find).
  - Prior Knowledge?

Data Representation

- Lesson:
  - Data Representation must be decided based on operations!
- Text Data: -List containing characters
  - Operations: Addition, Deletion, Search (Find).
  - Prior Knowledge?

Data Driven Programming-Case Study

- Consider the scenario of Text Editing:

  Requirements
  - What are the requirements?
    - How do we represent Text?
  - Operations
    - create Text? // (gets( ))
    - print Text?// helper functions (puts( ))
    - Finding whether a word is present in text ?
    - Update Text?

  - insertion: add a word at a position (Appending ?)

  - deletion: remove a word

Data Driven Programming [3]

- Design questions
  - Data Representation
    - How do we **design Text**?
    - How do we **represent a word**?
  - Algorithm Design
    - How do we design update operations?
    - How do we design search operation?

List Data - Operations

- How do we perform word insertion ?
  - Insertion is (partly) similar to linear Search(?):

  step1: Locate the position for insertion

  step2: Shift the rest of the contents from position towards right

  step3: Copy the contents of word

List Data - Operations

- How do we perform deletion ?
  - Deletion is (partly) similar to insertion:

  step1: Locate (i.e. search) the word,

  step2: Shift the contents towards Left by length of Word.

List Data - Operations

- How do we perform deletion ?
  - Deletion is (partly) similar to insertion:

  step1: Locate (i.e. search) the word,

  step2: Shift the contents towards Left by length of Word.

Data Abstraction

- What to show user?
  - List of operations:
    - void createString(String text)
    - void printString(String text)
    - Position findWord(String text, String word);
    - Size insertWord(String text, int pos, String newWd);
    - Size deleteWord(String text, String word);
  - What is Size, Position?

Data Type Representation: String.h

- #define LENGTH 80
- typedef char String[LENGTH];
- typedef int Position;
- typedef unsigned int Size;

Operations: StringOps.h

- #include "String.h"

/* Pre cond: memory allocation for text

Post cond: input text is stored in text */

- extern void createString(String text);

/* Pre cond: text should not be empty

Post cond: input text is printed */

- extern void printString(String text)

Operations On : StringOps.h

/* Pre cond: Text should have atleast one word

Post cond: returns index position of word, if present, else returns -1 */

- extern Position findWord(String text, String word);

/* Pre cond: There should be enough room for insertion;

$0 \leq pos < length(text)$

Post cond: returns length of Text after insertion*/

- extern Size insertWord(String text, Position index, String newWd);

- /* Pre condn: ?? Post cond ?? */
- extern Size deleteWord(String text, String word);

Operations On : StringOps.h

List out other operations with pre and post conditions

Implementations for Operations StringOps.c

```
#include “StringOps.h”
#include<stdio.h>
#include<string.h> // from C compiler!

  void   createString(String text)   { // code here}

  void   printString(String text)    { // code here}
```

Generate a linkable

gcc –c StringOps.c ➔ StringOps.o

User code: StringMain.c

```
#include “StringOps.h”
#include<stdio.h>
int main( )
{
// function calls
}
```

Generate a executable

gcc –o exefile StringMain.c StringOps.o

➔executable file is: exefile

➔Execute: **./exefile**

Data Driven Programming

- Example 2
  - Consider the scenario:
    - Librarian wants an electronic list of members.

Data Driven Programming [2]

- Requirements
  - What are the requirements?
    - Adding a new member to the list
    - Removing a member from the list
    - Finding out whether someone is a member
  - Is that all?
    - Program should be “fast”!

Data Driven Programming [3]

- Design questions
  - Data Representation
    - How do we design a list?
    - How do we represent a member?
  - Algorithm Design
    - How do we design addition (of a member to the list)?
    - How do we design deletion (of a member from the list)?
    - How do we design search?
    - How do we make it “fast”?

List Data - Operations

- Consider Search
  - Choices: Ordered List, Unordered List
  - Internal (Computer) Representation – Arrays
  - Ordered List: $O(\log_2 N)$
  - Unordered List: $O(N)$
  - (Local) Decision: Ordered List

List Data - Operations

- Now, consider addition:
  - If ordered list were our choice –
    - Order needs to be maintained on addition
    - Position to add must be found (I.e. searched)
    - Cost: At least O(logN)
  - E.g.
    - List (Ids of Member): 1 4 7 12 15 27
    - New Member: 13
    - New List: 1 4 7 12 13 15 27 (Last two elements moved!)

List Data - Operations

- Addition in Ordered List
  - O(logN) to find the position
  - Shifting elements to the right:
    - Worst case: All elements to be shifted: O(N)
    - Average case: (1 + 2 + … + N) / N is O(N)
  - Total complexity: (Worst and average case):

N + logN is O(N)

List Data - Operations

- Addition – Unordered List
  - Idea: Add it at the end!
  - Time Complexity: O(1)
- Back to Square One:
  - Ordered List or Unordered List?
  - Question to Librarian:
    - How often do you add members?
    - How often do you search for members?

List Data - Operations

- Possibility:
  - Search is done very often but addition is occasionally!
    - Choice: Ordered List
    - Search in O(logN) and Addition in O(N)
    - Suppose N searches to 1 addition is the ratio:

N*logN + N   is   O(N*logN)

Averaged over N+1 operations is O(logN)

List Data - Operations

- Possibility:
  - Search is done now and then but addition is done frequently!
    - Choice: Unordered List
    - Search in O(N) time and addition in O(1) time
    - Suppose N additions to 1 search is the ratio

N*1 + N   is O(N)

Averaged over N+1 operations is O(1)

List Data - Operations

- Does deletion affect the choice?
  - Deletion is (partly) similar to addition:
  - Locate (I.e. search), Delete, Move (Left) the rest
  - But these have to be done irrespective of whether the list is ordered or unordered.

List Data - Operations

- Design for change:
  - What if the mix of operations change tomorrow?
  - Choice of list may change.
  - Ensure separation between Interface and Implementation
  - This ensures Librarian need not know all your efficiency computations!

Data Abstraction

- Abstraction
  - Provide a view (I.e abstract) by hiding details
  - Data Abstraction
    - Expose operations (View)
    - Hide representation

Data Abstraction

- What to show Librarian?
  - List of operations:
    - Size add(Member, List, Size)
    - Size delete(Member, List, Size)
    - Boolean find(Member, List, Size)

Data Driven Programming-Example

- Recall the scenario:
  - Requirement:
    - Librarian wants an electronic list of members.
  - Design Choice:
    - Ordered List if more searches than additions
    - Unordered List if more additions than searches
    - What if data is not available?
      - Prepare for both eventualities

Data Abstraction - Principle

- Abstraction
  - Provide a view (i.e abstract) by hiding details
  - Data Abstraction
    - Expose operations (View)
    - Hide representation

Data Abstraction - Application

- What to show Librarian?
  - List of operations:
    - add(Member, List, ListSize) returns ListSize
    - delete(Member, List, ListSize) returns ListSize
    - isMember(Member, List, ListSize) returns Boolean
- Limitation of the Interface:
    - ListSize needs to be passed to operations: Librarian should not be bothered by such details

Data Abstraction - Application

	Files	Contents	Includes
1	LibraryList.h	typedef unsigned int ID; typedef unsinged int ListSize; #define MAX 1000 typedef unsigned int Member; typedef Member List[MAX];	____
2	boolean.h	typedef enum { TRUE = 1, FALSE=0 } Boolean;	____

3	LibraryOps.h	extern ListSize add(Member m, List ms,ListSize n);
		extern ListSize delete(Member m, List ms, ListSize n);
		extern Boolean isMember(ID i, List ms, ListSize n);
			2

4	LibraryOps.c	Implementation of ListSize add(Member m, List ms,ListSize n); ListSize delete(Member m, List ms, ListSize n); Boolean isMember(ID i, List ms, ListSize n);	3

Data Abstraction – Interface Declarations

```
/* file: LibraryOps.h */

#include "LibraryList.h"

/* Pre-condition:
            - Size of ms is n.
            - ms is a list of Member elements.
   Post-condition:
            - m added to ms if not present and overwritten if present.
            - return the size of the (possibly) updated list
*/
  extern ListSize add(Member m, List ms, ListSize n);
```

Data Abstraction – Interface Declarations

```
/* file: LibraryOps.h  … */

/* Pre-condition:
            - Size of ms is n.
            - ms is a list of Member elements.
   Post-condition:
            - m deleted from ms if present
                ms unchanged if not present.
            - return the size of the possibly updated list
*/

extern ListSize delete(Member m, List ms,
ListSize n);
```

Data Abstraction – Interface Declarations

```
/* file: LibraryOps.h  … */

  #include “boolean.h”

/* Pre-condition:
          - Size of ms is n.
          - Each element of ms can be matched against an ID
   Post-condition:
          -return TRUE if ID matches against an element in ms
                FALSE otherwise
*/
   extern Boolean isMember(Member m, List ms,
   ListSize n);
```

Data Abstraction – Types

```
/* file: LibraryList.h */

typedef unsigned int ID;
typedef unsinged int ListSize;

#define MAX 1000

typedef unsigned int Member;


// List is an array of at most MAX elements of type Member
typedef Member List[MAX];
```

Data Abstraction – Types

- Enumerated Type:
  - Recall that a type is a set of values
  - E.g. Boolean = { TRUE, FALSE };
  - E.g. DaysOfWeek = { Su, Mo, Tu, We, Th, Fr, Sa }
- C Syntax:
  - typedef enum { TRUE = 1, FALSE=0 } Boolean;
  - typedef enum { Su = 1, Mo, Tu, We, Th, Fr, Sa } DaysOfWeek;

Data Abstraction – Types

```
/* file: boolean.h */

typedef enum { TRUE = 1, FALSE=0 } Boolean;
```

Data Abstraction - Implementation

```
/* file: LibraryOps.c */
#include "LibraryOps.h"
ListSize add(Member m, List ms, ListSize n)
{   int pos, newsize = n;
    if (n+1 > MAX) return 0;  else  newsize=n+1;
     // This should lead to a test case.
    for (pos=0; pos < n; pos++)    {// pos iterations
               if (ms[pos] > m) break;
               if (ms[pos] == m) return newsize; // m already present
    }
    for ( ; n > pos; n--)  ms[n] = ms[n-1];  // n – pos –1 iterations
    ms[n] = m;
   return newsize;
}
```

Data Abstraction - Implementation

```
/* file: LibraryOps.c  … */

Boolean isMember(Member m, List ms, ListSize n)
{
    if  (search(m, ms, n) >= 0)  return TRUE;
    else return FALSE;
}
// Exercise: Implement delete
```

Data Abstraction - Summary

- Expose Data Types and Operations (through Interfaces)
  - E.g Member and List data types
  - E.g add, delete, and isMember interfaces
- Hide Representation and Implementation
  - E.g. List is an array of up to MAX elements
  - E.g. isMember is implemented through search.

Data Driven Programming - Observations

- Kinds of Data (used so far)
  - Simple data elements (int, Boolean etc.)
  - Data Collections – Linear collections or Lists
    - Often represented as Arrays
    - Often referred to as Random Access Lists.

For any i in the range (0 to size – 1) A[i] can be accessed.

Lists as Arrays - Observations

- Operations:
  - Access – Random element: A[i]
  - Access each element – Idiom:

for (i=0; i<N; i++)   … A[i] …

  - Cost of (each) Access: O(1)
    - Independent of i
  - Insertion / Deletion : Random position i
    - Movement of adjacent elements
    - Cost: O(N)
  - Limitation: Max size is a hard bound.

Today’s Agenda

Abstract Data Type - Set

Data Driven Programming – Example 2

- Consider Sets and operations on sets.
- Basic operation on sets:
  - Membership testing: Is x an element of S?
- Dependent operations on sets:
  - Equality testing: Is S1 same as S2?
  - Union computation: Let S be union of S1 and S2
  - Intersection computation: Let S be intersection of S1 and S2.
  - Complement computation: Let S be the complement of S1.

Provide (Abstract) Data Type - Set

1. Design a representation.
2. Design operations.
3. Define interfaces for operations over the new data type Set
4. Provide a representation for this type.
5. Implement the operations using this representation.

Representation for Sets

- First attempt: List of values.
  - E.g. S = { 5, 7, 10 } is represented as a list (or array)
  - int S[3] = { 5, 7, 10 }
- Is this a good representation?
  - Recall the criterion:Operations!
- Membership
  - Cost O(n) comparsions – using unsorted lists.
  - Cost log(n) comparisons – using sorted lists.

Provide (Abstract) Data Type - Set

1. Design a representation.
2. Design operations.
3. Define interfaces for operations over the new data type Set
4. Provide a representation for this type.
5. Implement the operations using this representation.

Representation for Sets

- First attempt: List of values.
  - E.g. S = { 5, 7, 10 } is represented as a list (or array)
  - int S[3] = { 5, 7, 10 }
- Membership
  - Cost O(n) comparsions – using unsorted lists.
  - Cost log(n) comparisons – using sorted lists.

Representation for Sets

- Union (ignoring duplicates)
  - Cost: O(max(m,n)) - assuming one list can be appended (by copying elements) to another
  - Cost: O(m+n) – if max. size cannot accommodate the united set.
  - Question: Can we predict the max size?
  - Max Size: Size of the universe (of values).

Representation for Sets

- Union (considering duplicates)
  - For each element x in S1,

  check whether x occurs in S2.
  - $O(mn)$ where $m=|S1|$ and $n=|S2|$ using unsorted lists

  i.e. $O(n^2)$ for same size
  - $O(m * \log(n))$ using sorted lists

  i.e. $O(n*\log n)$ for same size
- Intersection
  - Similar costs.

Representation for Sets

- Alternate Representation:
  - Basic idea: Membership is a boolean function on the universe of values. (i.e. yes or no function).
  - i.e. a set can be represented as a list of bits (0 or 1) corresponding to each element in the universe.
  - Simplifications:
    - Assume elements are integers and universe is finite.
    - So, each set is a binary string of fixed length |U|.

Representation for Sets

- Simplistic Example:
  - Universe U = { 0 .. 31 }
  - Each set is a bit string of length 32.
  - Each position j in the bit string corresponds to membership of j in the set.
  - E.g. S1 = { 1 } can be represented by

00000000000000000000000000000010

  - E.g. S2 = { 2, 4, 31 } can be repd. by

10000000000000000000000000010100

Representation for Sets

- Consider the operations:
  - Union: if $S = S1 \cup S2$,
    then $x \in S \Longleftrightarrow x \in S1$ OR $x \in S2$
  - Union operation is Bit-wise OR operation since membership is 0 or 1.
  - Extend the model:
    - Intersection is Bit-wise AND operation
    - Complement is Bit-wise NOT operation
    - Equality is Bit-wise EQUALITY!!
    - Question: What is S1 <> S2 ?

Sets and Operations on Sets - Steps

1. Design a representation. ✓
2. Design operations. ✓
3. Define interfaces for operations over the new data type Set
4. Provide a representation for this type.
5. Implement the operations using this representation.

Set Data Type

- Suppose we define (in C)

  typedef unsigned int SmallSet;

- Question: How many bits in SmallSet?
  - Answer: 16, or 32 or … // machine dependent
  - C Answer: sizeof(SmallSet) // machine-independent
  - So, what?
  - We can say "testing membership of x in SmallSet is valid if 0 <= x < sizeof(SmallSet)"

Set Data Type – Type Declaration

```
/* file: SmallSet.h */

 /* y is a SmallSet and x in y   →
           0 <= x < sizeof(SmallSet)
 */
  typedef unsigned int SmallSet;
```

Set Data Type - Interfaces

```
/* file: SmallSetOps.h */
#include “SmallSet.h”
#include “boolean.h”
/* Pre-condition:
      x < sizeof(SmallSet)
   Post-condition:
      return TRUE if x is an element of S.
                  FALSE otherwise.
*/
extern Boolean elementOf(unsigned int x, SmallSet S);
```

Set Data Type - Interfaces

```
/* file: SmallSetOps.h   … */

 /* Post-condition:
      return S such that x in S
                 iff  x in S1 OR x in S2
 */
extern SmallSet union(SmallSet S1, SmallSet S2);

  /* Post-condition:
       return S such that x in S
                  iff  x in S1 AND x in S2
  */
 extern SmallSet intersect(SmallSet S1, SmallSet S2);
```

Set Data Type - Interfaces

```
/* file: SmallSetOps.h  …  */
 /* Post-condition:
      return S1 such that x in S1
               iff  NOT ( x in S )
 */
extern SmallSet complement(SmallSet S);

  /* Post-condition:
       return
         TRUE  if ( x in S1 ➔x in S2 ) AND (x in S2 ➔ x in S1)
         FALSE otherwise*/
 extern Boolean equals(SmallSet S1, SmallSet S2);
```

Sets and Operations on Sets - Steps

1. Design a representation. ✓
2. Design operations. ✓
3. Define interfaces for operations over the new data type Set ✓
4. Provide a representation for this type. ✓
5. Implement the operations using this representation.

Set Data Type - Implementation

```
/* file: SmallSetOps.c  */
#include "SmallSetOps.h"
const unsigned int mask = (unsigned int)1;
Boolean elementOf(unsigned int x, SmallSet S)
{
        if (S & (mask << x))      return TRUE;
         else return FALSE;

}

// Exercise: Provide a correctness argument!
```

Set Data Type - Implementation

```
/* file: SmallSetOps.c …  */

 SmallSet union(SmallSet S1, SmallSet S2)
 {
   return S1 | S2;
 }
 SmallSet intersect(SmallSet S1, SmallSet S2)
 {
   return S1 & S2;
 }

  // Exercise: Provide correctness arguments!
```

Set Data Type - Implementation

```
/* file: SmallSetOps.c …  */

SmallSet complement(SmallSet S)
{
 // Exercise: Provide implementation!
}
SmallSet equals(SmallSet S1, SmallSet S2)
{
   // Exercise: Provide implementation!
}

 // Exercise: Provide correctness arguments!
```

Set Data Type - Limitations

- What if the size of universal set is large?
  - i.e. greater than 31 or greater than sizeof(SmallSet)
- Let |U| = N
  - Each set can be represented by a list of SmallSet elements
  - A set S will have N bits
  - Implement the idea:
    - use SmallSet as an abstract type.
    - i.e. SmallSet operations are used not its representation.

Set Data Type – General Representation

- Assume |U| = N and sizeof(SmallSet) = B
- Set S is a list of ceil(N / B) SmallSet elements.
  - E.g. N = 125 and B = 32
  - S is represented by an array of size 4 (=ceil(125 / 32))
  - S[0] is the SmallSet { x in S | 0 <= x < 32 }
  - S[1] is the SmallSet

{ x - 32 | 32 <= x < 64 and x in S } …

Set Data Type – General Representation

- Membership (for 0 <= x < N):
  - x in S iff (x mod B) in S[x / B]
  - Let N=125 and B=32 and S = { 5, 36, 96 }
  - 5 in S iff (5 mod 32) in S[5/32]

    i.e. 5 in S iff 5 in S[0]
  - 36 in S iff (36 mod 32) in S[36/32]

    i.e. 36 in S iff 4 in S[1]
  - 96 in S iff 0 in S[3]

Set Data Type – General Representation

- Membership:
  - Cost: A division, a mod operation, plus membership in a small set.
  - Complexity: Same as Membership in small set
    - i.e. O(sizeof(SmallSet)) which is constant compared to N.
- Added Flexibility
  - We can fix the max size of the array here!

Set Data Type – General Representation

- Algorithm for Union:
  - S = Union(S1, S2) where S1, S2 are arrays of SmallSet elements
  - for each i, 0 <= i <= N/sizeof(SmallSet)

S[i] = union(S1[i], S2[i])

- Complexity
  - Time: O(N) but space O(N) fixed ahead of time!

Set Data Type – General Representation

- Adapt the design from SmallSet for general sets
- Exercise:
  - Provide Data Type Set by going through the rest of steps
    - Type Declaration
    - Interface Declarations and
    - Implementations for operations.

Set Data Type – Comparison of Representations

- Exercise:

  Compare the operation costs (membership, union) for the two different representations:

  list of elements vs. list of bits.

  Under what scenarios would you choose one over the other?