CS 126 Lecture P5: Abstract Data Type

Outline

• Introduction
• Stacks (and queues)
• Stack and queue applications
Data Type and ADT

Data type
- set of values
- collection of operations on those values
  ex: short int
- set of values between -32768 and 32767
- arithmetic operations + - * /

Abstract data type (ADT)
- data type whose representation is hidden

Interface, Implementation, and Client

Separate implementation from specification
INTERFACE: specify the allowed operations
IMPLEMENTATION: provide code for ops
CLIENT: code that uses them
Advantages of ADT

- Advantages
  * different clients can use the same ADT
  * can change ADT without changing clients

- Convenient way to organize large programs
  * decompose into smaller problems
  * substitute alternate solutions
  * separate compilation
  * build libraries

Powerful mechanism for building layers of abstraction

Client can work at higher level of abstraction

“Non-ADTs”

```
interface: #define struct {int p; int q;} Rational;
client:      Rational a; a.p = 3;

Non-ADT

interface: #define struct {int p; int q;} Rational;
    void setRationalP(Rational *r; int x);
implementation: void setRationalP(Rational *r; int x)
                  {r->p = x;}
client:      Rational a; setRationalP(&r, 3);

Rational data type (Assignment 3) is NOT an ADT
representation is in interface

Are C built-in types ADTs?

ALMOST: we generally ignore representation
NO: set of values depends on representation
YES: good programs use <limits.h> to function
      properly independent of representation
```
Outline

• Introduction
• Stacks (and queues)
• Stack and queue applications

Stack and Queue Definitions

Prototypical data types
set of operations on generic data

STACK ("last in, first out" or LIFO)
push: add info to the data structure
pop: remove the info most recently added (initialize, test if empty)

QUEUE ("first in, first out" or FIFO)
put: add info to the data structure
get: remove the info LEAST recently added (initialize, test if empty)

Could use either array or linked list
to implement either stack or queue

Client can work at higher level of abstraction
(stay tuned)
• “Client” needs to know how to use the “interface”

• “Implementation” needs to know what “interface” to implement
Post-increment:
\[ s[N] = \text{item}; \]
\[ N+=1; \]

Pre-decrement:
\[ N-=1; \]
\[ \text{return } s[N]; \]
Demo Linked List Stack
Outline

• Introduction
• Stacks (and queues)
• Stack and queue applications
**Reverse polish (postfix) notation**

- Put operator after operands in expression
- Use stack to evaluate operand: push it on stack
  operator: pop operands, push result
- Systematic way to save intermediate results

**Practical example of use of stack abstraction**

Ex: convert 97531 from hex to decimal

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
</tbody>
</table>

9
7
5
3
1

Ex: alternate implementation

Stack never has more than two numbers on it! HORNER’S METHOD (see lecture Ai)

- number of inputs to main
- array of input strings
- number of characters in string
- for each character in the string
- pop two items, add them up, and push result back
- deal with digits:
  - pop the previous digits, multiply the number by 10, add the new digit, and push the partial answer back.
Demo Postfix Calculator

PostScript

defined language, abstract stack machine

Ex: convert 0x7f from hex to decimal
    \[ 9 \quad 16 \quad \text{mul} \quad 7 \quad \text{add} \quad 16 \quad \text{mul} \quad 5 \quad \text{add} \quad 16 \quad \text{mul} \quad 3 \quad \text{add} \quad 16 \quad \text{mul} \quad 1 \quad \text{add} \]

Stack:
- operands for operators
- arguments for functions
- return values for functions

Coordinate system: rotate, translate, scale, ...
Turtle commands: moveto, lineto, rmoveto, rlineto
Graphics commands: stroke, fill, ...
Arithmetic: add, sub, mul, div, ...
Stack commands: copy, exch, dup, currentpoint, ...
Control constructs: if, ifelse, while, for, ...
Define functions: /XX { ... } def

Everyone’s first program: draw a box
\%
50 50 translate
0 0 moveto 0 512 rlineto
512 0 rlineto 0 -512 rlineto -512 0 rlineto
stroke
showpage
“First Class” ADTs

- So far, only one stack (or queue) per program

- “First Class” ADTs
  - An ADT that is just like a built-in C type
  - Can declare multiple instances of them
  - Pass specific instances of them to the interface functions as inputs

```c
{ ... 
    stackInit();
    ... 
    stackPush(5);
} { 
    Stack s1, s2;
    s1 = stackInit(); s2 = stackInit();
    ... 
    stackPush(s1, 5); stackPush(s2, 8);
}
```

“WAR” using abstract data types

- Need “first class” queue ADT
- Declare variables of type queue
- Use them as arguments to functions
- Hide representation from clients

Interface, implementation?

```
{ //object-oriented programming; stay tuned
    "peace" client code, FYI
    ("QUEUE -> "Q" in identifiers, for brevity)
    int play(Q deck)
    { int Aval, Bval, i, cnt = 0; Q T = Qinit();
      deal(deck);
      while (((Qempty(A)) && (!Qempty(B))))
      { cnt++;
        Aval = Qget(A); Bval = Qget(B);
        if (randT(2))
          { Qput(T, Aval); Qput(T, Bval); }
        else { Qput(T, Bval); Qput(T, Aval); }
        if (Aval % 13 > Bval % 13)
          while (!Qempty(T) Qput(A, Qget(T));
        else 
          while (!Qempty(T) Qput(A, Qget(T));
      }
      return cnt;
    }
}
```

*Take one element from the T queue at a time, add it to the A queue, and repeat until the T queue is empty.*

(Not most efficient.)
Conclusion

• ADT is one of the most important concepts for managing software engineering complexity
• Learn to identify the possible use of ADTs in a program
• Learn the proper decomposition and encapsulation using interface and implementation files