COS 126 Lecture 7: Data structures and types

Data Structure
* mechanism for organizing information
    arrays
    structs
    linked lists
    binary trees
    (compound structures)

Data Type
* set of values
* collection of operations on those values
    int, float, char
    strings
    stacks
    queues

Interface
Implementation
Client

Abstract data type
Object-oriented programming

Array

Fundamental data structure

    store values sequentially in memory
    associate INDEX with each value
    use index to quickly access Nth for any N

Example: symbolic manipulation of polynomials

C representation of x^14 + 3x^5 + 12:

```
int p[15];
for (i = 0; i < 15; i++) p[i] = 0;
p[0] = 12; p[5] = 3; p[14] = 1;
```

TOY representation of x^14 + 3x^5 + 12:

```
22: 000C
23: 0000
24: 0000
25: 0000
26: 0000
27: 0003
28: 0000
29: 0000
2A: 0000
2B: 0000
2C: 0000
2D: 0000
2E: 0000
2F: 0000
30: 0001
```

Advantages:

    can get to each item quickly
    index carries implicit info, takes no space

Disadvantage: Uses up space for unused items

Linked List

Fundamental data structure

store values anywhere in memory

associate LINK with each value

use link to quickly access the NEXT value

TOY representation of x^14 + 3x^5 + 12:

```
22: 000E
23: 0001
24: 004D
...
38: 0000
39: 000C
3A: 0000
...
4D: 0005
4E: 0003
4F: 0038
```

Advantage: space proportional to amount of info

Disadvantage: can only get to *next* item quickly

C representation of x^14 + 3x^5 + 12: ???

Need to know:

how to associate pieces of information

how to specify links

how to reserve memory to be used

how to use links to access information

Primitives for linked lists in C

To associate pieces of info, use C structures

```
typedef struct { int p; int q; } Rational;
float x; Rational t;
x = (1.0)*t.p/t.q
```

To specify links, use "*" (pointers)

```
typedef struct node* link;
struct node
   { int coef; int exp; link next; };
link p;
```

To reserve memory for a structure, use "malloc"
(library routine declared in stdlib.h)

```
p = malloc(sizeof *p);
```

To use a pointer to access information, use "->"

```
p->coef = 1;
```

Build the list for x^14 + 3x^5 + 12

```
p = malloc(sizeof *p)
p->coef = 1; p->exp = 14;
q = malloc(sizeof *p)
q->coef = 3; q->exp = 5;
r = malloc(sizeof *p)
r->coef = 12; r->exp = 0;
p->next = q; q->next = r; r->next = NULL;
```

STUDY THIS: Tip of the iceberg!

Digression: pointers

POINTERS (addresses)

Used to implement BOTH arrays and linked lists

ARRAY

```
.    p         pointer to array (first item)
.   *p         first array item
.   p+1        pointer to second array item
.  *(p+1)      second array item
.
.  *(p+i)      (i+1)st array item
.   p[i]       alternate way to say the same thing
```

LINKED LIST

```
.      p        pointer to structure
.     *p           structure
.  (*p).coef   field in structure
.   p->coef    alternate way to say the same thing
```

Stacks, Queues

Prototypical data types
    data structures
    defined by operations

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

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

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

Client can work at higher level of abstraction
    (stay tuned)

Stack interface and client

Interface (STACK.h)

```
void STACKinit();
int STACKempty();
void STACKpush(int);
int STACKpop();
```

Client (myprog.c)

```
#include "STACK.h"
main()
  { int a, b;
      ...
      STACKinit(N);
      ...
      STACKpush(a);
      ...
      b = STACKpop();
      ...
  }
```

Client uses data type, without regard to how
it is implemented

Array implementation of stack (of ints)

Push and pop at the end of the array

```
#include <stdlib.h>
#include "STACK.h"
int s[1000];
int N;

void STACKinit()
  { N = 0; }
int STACKempty()
  { return N == 0; }
void STACKpush(int item)
  { s[N++] = item; }
int STACKpop()
  { return s[--N]; }
```

Client and implementation both include STACK.h

(could be compiled separately)

can be compiled with one command

```
cc myprog.c stackArray.c
a.out
```

Problem: have to reserve space for max size

Linked-list implementation of stack

Push and pop nodes at the front of the list

```
#include <stdlib.h>
typedef struct STACKnode* link;
struct STACKnode { int item; link next; };
static link head;
link NEW(int item, link next)
  { link x = malloc(sizeof *x);
    x->item = item; x->next = next;
    return x;
  }
void STACKinit(int maxN)
  { head = NULL; }
int STACKempty()
  { return head == NULL; }
STACKpush(int item)
  { head = NEW(item, head); }
int STACKpop()
  { int item = head->item;
    link t = head->next;
    free(head); head = t;
    return item;
  }
```

Switch implementations without changing interface or client

```
  cc myprog.c stackList.c
  a.out
```

Polish notation

Practical example of use of stack abstraction

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

Ex: convert 97531 from hex to decimal

```
9 16 16 16 16 * * * * 7 16 16 16 * * *
     5 16 16 * * 3 16 * 1 + + + +
9
9 16
9 16 16
9 16 16 16
9 16 16 16 16
9 16 16 256
9 16 4096
9 65536
589824
589824 7
589824 7 16
...
589824 28672 1280 48 1
589824 28672 1280 49
589824 28672 1329
589824 30001
619825
```

Note: HORNER's METHOD (see lecture 4)

```
9 16 * 7 + 16 * 5 + 16 * 3 + 16 * 1 +
```

Stack never has more than two numbers on it!

PostScript

Language is postfix
Abstract stack machine

Ex: convert 97531 from hex to decimal

```
9 16 mul 7 add 16 mul 5 add
     16 mul 3 add 16 mul 1 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: 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
```

Abstract data types (ADT)

A data type that is accessed *only*
through the operations in the interface

Example: STACK implementations

Rational data type (Assignment 3) is NOT an ADT
representation is in interface
(possible to fix: see CS217)

Representation is hidden in the implementation
Client program does not need to know
how the implementation works

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 librarys

Powerful mechanism
for building layers of abstraction