CSE 143

Stacks

[Chapter 6]

7/17/00 O-1

Collections

- Collections or containers are ADTs that hold many items of data, usually all of the same type
- We can think of arrays and lists as collections
- Some programming languages (or their libraries) support various types of containers directly
  - C++ has the Standard Template Library or STL

    `includes many generic algorithms and container types`
  - But to use the library you have to understand the container concepts as well as many advanced language features

7/17/00 O-2

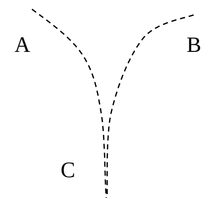
Collections in 143

- We've seens Arrays and Lists (Vectors) already
- We'll continue with Stacks, then Queues, then Trees and perhaps a couple of others
- Other collection types can be programmed and/or invented by you!
  - But knowing what's already been invented can keep you from "reinventing the wheel."

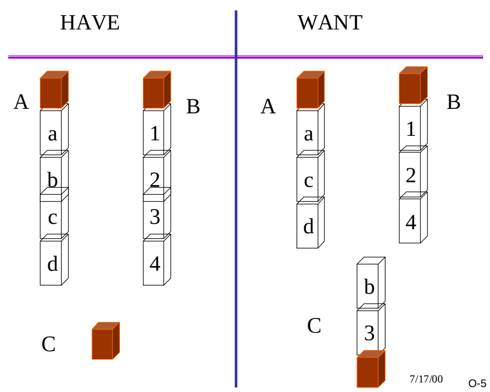
7/17/00 O-3

Problem: Switching Boxcars

- The back of three trains meet at a Y-junction
- Can only add/remove from the caboose-end of each train
- Goal: get the right cars, in the right order, on the trains.



7/17/00 O-4



7/17/00 O-5

Typing and Correcting Chars

- Type characters, use backspace (<) to mean "erase the previous character"
- The most recently typed unerased char is the one erased

7/17/00 O-6

Sample

- Action
- type h
- type e
- type l
- type o
- type <
- type l
- type w
- type <
- type <
- type <
- type <
- type i

- Result
- **h**
- **he**
- **hel**
- **helo**
- **hel**
- **hell**
- **hellw**
- **hell**
- **hel**
- **he**
- **h**
- **hi**

7/17/00 O-7

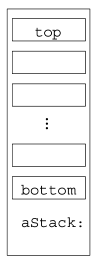
What's common?

- I have data to process
  - boxcars; characters
- The order of adding data is remembered
- *I can only remove or affect what I most recently put in*
- We say the data structure is **LIFO** or *Last In, First Out*, and we call it a **Stack**.
- The point where you can add data is called the **Top**.
  - boxcar train: Top is the end of the train
  - character line: Top is the rightmost character

7/17/00 O-8

Stack as ADT

- *Top*: Uppermost element of stack, first to be removed
- *Bottom*: Lowest element of stack, last to be removed
- *Elements are always inserted and removed from the top (LIFO)*
- Homogeneous collection (items all the same type)
  - Could be ANY type
  - Most of our lecture examples are stacks of ints



7/17/00 O-9

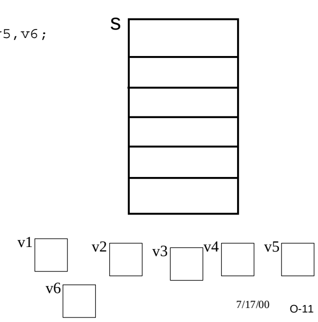
Abstract Stack Operations

- ***push****(item)*: Adds an element to top of stack, increasing stack height by one
- *item* ***pop****()*: Removes topmost element from stack and returns it, decreasing stack height by one
- *item* ***top****()*: Returns a copy of topmost element of stack, leaving stack unchanged
- No "direct access"
  - cannot index to a particular data item
- No way to traverse the collection

7/17/00 O-10

What is the result of...

```
Stack s;
int v1,v2,v3,v4,v5,v6;
s.push(1);
s.push(2);
v1 = s.pop();
s.push(3);
s.push(4);
v2 = s.pop();
s.push(5);
v3 = s.pop();
v4 = s.pop();
v5 = s.pop();
v6 = s.pop();
```



7/17/00 O-11

Stacks Around Us

- Stack of books on a desk
- Trays in the Husky Den
  - Take one from top only
  - Tray on bottom was put there first
- Discard pile in a card game
  - Discard on to top, draw card from top
  - Not allowed to see or draw what's underneath
- Towers of Hanoi

7/17/00 O-12

Stacks in CS

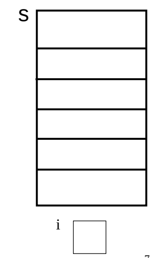
- Implementing function calls
  - Activation records go on a stack
- Evaluating expressions
  - How does a calculator (or compiler) understand (3 +4)/5?

    more later
- "Backtracking" to systematically try all combinations of possibilities
  - e.g., to explore paths through a maze

Stack Example

- Show the changes to the stack in the following example:

```
Stack s;
int i;
s.push(5);
s.push(3);
s.push(9);
i = s.pop();
i = s.top();
s.push(6);
s.push(4);
```



A Stack Class Interface

```
class IntStack {
public:
   IntStack();          //constructor
                //should have a copy constructor, too
   bool isEmpty();      // is the stack empty?
   void push(int item);
                        // add item to top
   int pop();           // remove and return top item
   int top();           // show the top item
private:
   . . .
};
```


A Stack Client

```
// Goal: Read numbers and print in reverse order

void ReverseNumbers() {
 IntStack s;
 int oneNumber;
 while ( cin >> oneNumber ) {
   s.push(oneNumber);
 }
 while ( !s.isEmpty() )
   oneNumber = s.pop();
   cout << oneNumber << endl;
}
```


Ex.: Parenthesis Matching

- Are these legal expressions?

  a + (b * [c + d] / (c – d))

  a +) (b * [c + d]]] / (c – d)]

  a + (b * [c + d) / (c – d)]
- How to tell?
  - Method 1: match "inside out"
    Good "by eye" but hard to program
  - Method 2: Via Stack – push '(', '['; pop & match ')', ']'
    Harder "by eye", but easier to program.

Parenthesis Matching

```
char c;
charStack s;
while (cin >> c) {
 if ( c == '(' || c== '[' )
     s.push( c );
 else if ( c == ')' )
     if ( s.pop() != '(' ) … // error
 else if ( c==']' )
     if ( s.pop() != '[' ) … // error
}
if ( !s.empty() ) … //error
cout << "Parens all match."
//NEEDS TEST FOR PREMATURE EMPTY, TOO!
```

Stack Implementations

- Many possible implementations
  - Array-based
  - Linked list
  - Or even, using already implemented Vector ADT
- As implementer, use other ADTs to make job easier
  - Don't reinvent the wheel for every problem
  - Often simplifies job to reuse pieces when possible
- We'll use stack of ints as an example
  - could have stack of any type of data item

7/17/00 O-19

Stack Via Vector ADT

- We'll use a private Vector variable.

```
#include "Vector.h"

class IntStack {
public:
   IntStack();
   bool isEmpty();            // is the stack empty?
   // etc etc -- all the Stack operations
   ...
private:
   Vector items;
};
```

Note: no Top variable! Always use the front of the list as the Top.

7/17/00 O-20

Review: Vector Interface

class Vector {

public:

```
Vector ( );
bool   isEmpty( );
int    length ( );
void   insert (int newPosition, int newItem);
int    delete (int position);
int    retrieve (int position);
...
}
```

7/17/00 O-21

Stack Via Vector (2)

```
IntStack::IntStack() { }
   // don't need to do anything, why?


bool IntStack::isEmpty() {
   return items.isEmpty();
}
```

7/17/00 O-22

Stack Via Vector (3)

```
void IntStack::push(int item) {
   items.insert(0, item);
}
int IntStack::top() {
   //FILL THIS IN
   //HINT: can be done in one line of code


}
int IntStack::pop() {
   //FILL THIS IN



}
```

7/17/00 O-23

Possible Implementations

- Many possible implementations
  - Array-based
  - Linked list
  - Or even, using already implemented List ADT

```
As implementer, use other ADTs to make job easier
Don't reinvent the wheel for every problem
Often simplifies job to reuse pieces when possible
```

7/17/00 O-24

Stack Via Linked List

- Another implementation technique
- Main idea: keep a linked list, with private "top" pointer to the front of the list
- Add new data as a new link to the beginning of the linked list
- Pop/top: remove/return the beginning of the linked list
- Not the only way -- could have decided to make top be the end of the list
  - Important thing is to choose a way;document it; and stick with it.

7/17/00 O-25

Stack Via Linked List (2)

```
struct Node {
  int data;
  Node* next;
};

class IntStack {
public:  //same as before
   ...
private:
   Node * top;   //points to top
                 //NULL means empty stack
};
```

7/17/00 O-26

Stack Via Linked List (3)

```
// Push item onto top of this stack
void IntStack::push(int item) {
   Node *newNode = new Node;
   assert(newNode != NULL);
   newNode->data = item;
   newNode->next = top;
   top = newNode;
}
// pop an item off the stack
int IntStack::pop() {



}
```

7/17/00 O-27

Possible Implementations

- Many possible implementations
  - Array-based
  - Linked list
  - Or even, using already implemented List ADT

    As implementer, use other ADTs to make job easier
    Don't reinvent the wheel for every problem
    Often simplifies job to reuse pieces when possible

7/17/00 O-28

Stack Via Dynamic Arrays

```
class IntStack {
public:  //same as before
   ...
private:
   int size;          // # items currently in stack
   int capacity;      // amount of space allocated
   int *data;         // Items in stack are stored
                      // in data[0.. size-1].
                //data[0]is the bottom of the stack;
   // data[size-1] is the top item on the stack.
};
```

- **The comments are very important to record how we plan to use the variables**

7/17/00 O-29

Stack Via Arrays (2)

```
// construct empty IntStack
IntStack::IntStack() {
        size = 0;
        capacity = DEFAULT_CAPACITY;
        data = new int[capacity];
        assert(data != NULL);
}

// = "this stack is empty"
bool IntStack::isEmpty() {
   return (size == 0);
}
```

7/17/00 O-30

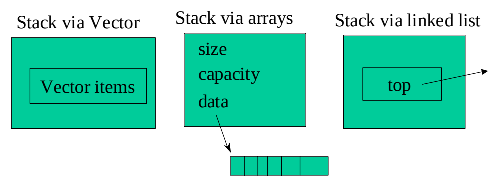
Stack Via Arrays (3)

```
// Push item onto top of this stack
void IntStack::push(int item) {
   if (size == capacity) growArray(capacity * 2);
   else {
          data[size] = item;
          size++;
   }
}
// FILL IN THE CODE
int IntStack::pop() {



}
```

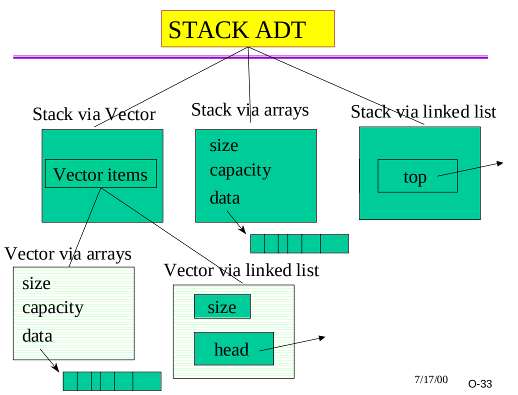
7/17/00 O-31

Picturing the Implementations

All have the same public interface
Picture the private data
- Vector itself allows more than one implementation!



7/17/00 O-32



7/17/00 O-33

Discussion

- Why learn three different ways to implement the same ADT?
- What are the pro's and con's of each way?
  - Programming effort?
  - Speed (efficiency) of execution?
  - Suitability to application?
  - Other factors?

7/17/00 O-34

Stack Application: Evaluating Expressions

- Expressions like "3 * (4 + 5)" have to be evaluated by calculators and compilers
- We'll look first at another form of expression, called "postfix" or "reverse Polish notation"
- Turns out a stack algorithm works like magic to do postfix evaluation
- And... another stack algorithm can be used to convert from infix to postfix!

7/17/00 O-35

Postfix vs. Infix

- Review: Expressions have *operators* (+, -, *, /, etc) and *operands* (numbers, variables)
- In everyday use, we write the binary operators <u>in between</u> the operands
  - "**4 + 5**" means "add 4 and 5"
  - called ***in****fix* notation
- No reason why we couldn't write the two operands first, then the operator
  - "**4 5 +**" would mean "add 4 and 5"
  - called ***post****fix* notation

7/17/00 O-36

More on Postfix

- **3 4 5 * -** means same as (3 (4 5 *) -)
  - infix: **3 - (4 * 5)**
- Parentheses aren't needed!
  - When you see an operator:

```
both operands must already be available.
Stop and apply the operator, then go on
```

- Precedence is implicit
  - Do the operators in the order found, period!
- Practice converting and evaluating:
  - 1 2 + 7 * 2 %
  - (3 + (5 / 3) * 6) - 4

7/17/00 O-37

Why Postfix?

- Does not require parentheses!
- Some calculators make you type in that way
- Easy to process by a program
- The processing algorithm uses a stack for operands (data)
  - simple and efficient

7/17/00 O-38

Postfix Evaluation via a Stack

- Read in the next "token" (operator or data)
  - If data, push it on the data stack
  - If (binary) operator (call it "op"):

```
Pop off the most recent data (B) and next most recent (A)
Perform the operation R = A op B
Push R on the stack
```

- Continue with the next token
- When finished, the answer is the stack top.
- Simple, but works like magic!

7/17/00 O-39

Refinements and Errors

- If data stack is ever empty when data is needed for an operation:
  - Then the original expression was bad
  - Too many operators up to that point
- If the data stack is <u>not</u> empty after the last token has been processed and the stack popped:
  - Then the original expression was bad
  - Too few operators or too many operands

7/17/00 O-40

Example: **3 4 5 - ***

Draw the stack at each step!

- Read 3. Push it (because it's data)
- Read 4. Push it.
- Read 5. Push it.
- Read -. Pop 5, pop 4, perform 4 - 5. Push -1
- Read *. Pop -1, pop 3, perform 3 * -1. Push -3.
- No more tokens. Final answer: pop the -3.
  - note that stack is now empty

7/17/00 O-41

Converting in- to post-

- A different algorithm converts from infix to postfix
  - Uses a stack of operators.
- Algorithm:
  - Read a token
  - If operand, output it immediately
  - If '(', push the '(' on stack
  - If operator:

```
if stack top is an op of => precedence: pop and output
stop when '(' is on top or stack empty
push the new operator
```

  - If ')', pop and output until '(' has been popped
  - Repeat until end of input

```
pop rest of stack
```

7/17/00 O-42

Another Stack Application

- Searching for a path through a maze
- Algorithm: try all possible sequences of locations in the maze until you find one that works (or no more to try)
  - called “exhaustive search”
- A stack helps keep track of the possibilities
  - traces a path of locations
  - just like the recursive activation records in the maze-solver

7/17/00 O-43

Stack Wrapup

- Essence: Last In, First Out
- Various ways to implement
- Numerous uses
  - In Computer Science
  - In modeling the world

7/17/00 O-44