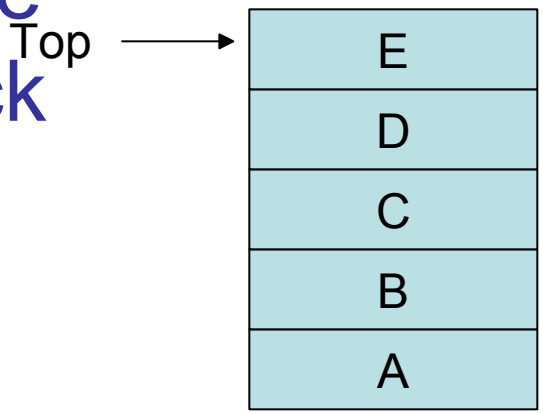
Data Structures

Lecture 4: Stacks

Chapter 2

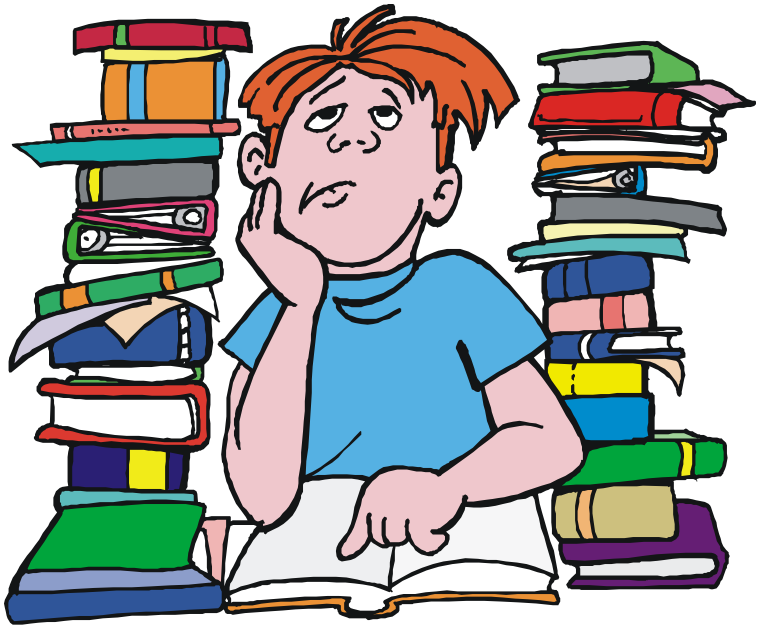
What is a Stack?

- A **stack** is an ordered collection of items in which all insertions and deletions are made at one end, called the ***top*** of the stack
- Either end of the stack can be designated as top of the stack



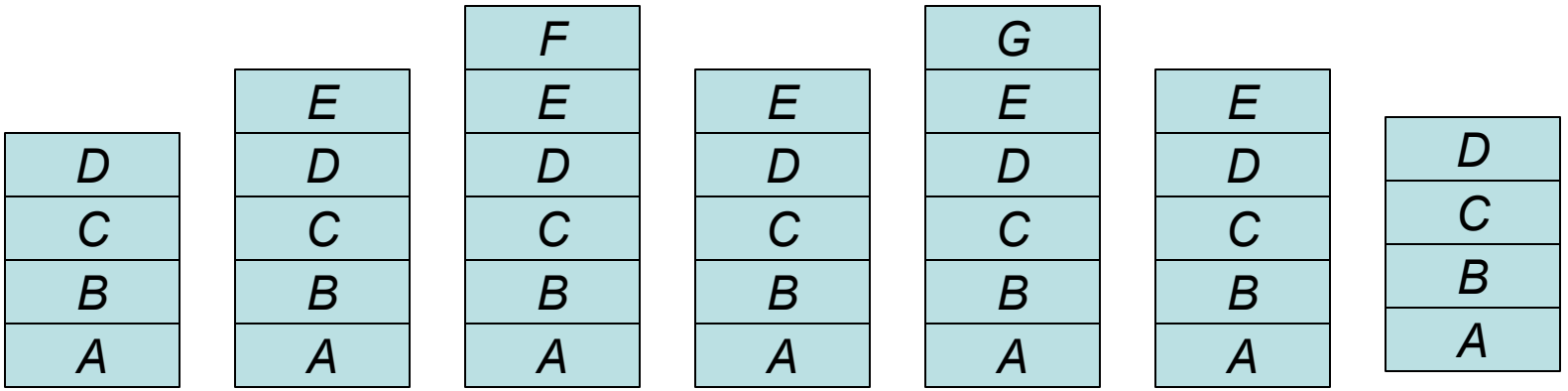
Stack Analogy

- *Intuitively, a stack is like a pile of plates where we can only (conveniently) remove a plate from the top and can only add a new plate on the top*
- *In computer science we commonly place numbers on a stack, or perhaps place records on the stack*



Two stack of books

Stack Frames



Motion picture of a stack

Applications

- **Direct applications**
  - *Page-visited history in a Web browser*
  - *Undo sequence in a text editor*
  - *Chain of function calls in a C program*
- **Indirect applications**
  - *Auxiliary data structure for algorithms*
  - *Component of other data structures*

Primitive Operations: Push and Pop

- **Push:** When we add an item to a stack, we say that we ***push*** it onto the stack
  - The last item put into the stack is at the top
- **Pop:** When we remove an item, we say that we ***pop*** it from the stack
  - When an item popped, it is always the top item which is removed

Other Operations

A stack containing zero items is called an empty stack

- **Empty:** The **Empty** operation tells us if the stack is empty or not
- **MakeEmpty:** Creates an empty stack
- **Top:** This operation returns the current item at the top of the stack, it doesn't remove it

Stack Underflow

- The operations **Pop** and **Top** is not defined for an empty stack
- The result of an illegal attempt to **Pop** or access (**Top**) an item from an empty stack is called ***stack underflow***
  - Underflow can be avoided by ensuring that **Empty** is false before attempting the operation **Pop** or **Top**

Other Names of Stack

- **Since it is always the last item to be put into the stack that is the first item to be removed, a stack is a *last-in, first-out*** or ***LIFO list***
- Because of the push operation which adds elements to a stack, a stack is sometimes called a ***pushdown list***

Example: Reversing a string

- ***Make an empty stack S***
- For each character ***ch*** in the input string
  - **Push *ch*** in ***S***
- While ***S*** is not empty
  - ***ch*** = **Pop** a character from ***S***
  - Print ***ch***

Example: Balancing Symbols

- ***Given a mathematical expression such as***
  7 – ((X * ((X + Y) / (J – 3)) + Y) / (4 – 2.5))
- ***We want to check that***
  - ***There are an equal number of right and left parenthesis***
  - ***Every right parenthesis is preceded by a matching left parenthesis***

Solution

- *Scan the expression from left to right*
- *At any point of the expression, define a parenthesis count (initially 0)*
  - *This values is the no. of left parenthesis minus the no. of right parenthesis that have been encountered in scanning the expression from its left end up to that particular point*
- *Expression is valid, if*
  - *The parenthesis count at the end is 0*
  - *The parenthesis count at each point in the expression is nonnegative*

Illustration

- `7-((X*((X+Y)/(J-3))+Y)/(4-2.5))`
  `00122234444334444322211222222210`
- ***What if you want to introduce three different types of scope delimiters, (, {, [***
  - *Do you need a separate count for each type of delimiter OR use a stack?*

Algorithm for Balancing Symbols

- ***valid*** = true //Assuming the stack is valid
- Make an empty stack ***S***
- For each symbol ***symb*** in the input expression
  - If ***symb*** is an opening symbol, **Push *symb*** into ***S***
  - Else If ***symb*** is a closing symbol
    - If ***S*** is **Empty**, valid = false
    - Else
      - ***i*** = **Pop** a symbol from ***S***
      - If ***i*** is not the matching opener of ***symb***, ***valid*** = false
- If ***S*** is not **Empty**, ***valid*** = false
- If ***valid*** is true, expression is valid otherwise invalid

Trace of Balancing Symbol Algo.

Expression	Current Symbol	Stack Operation	Contents of symbol Stack s
{x + (y – [a + b]) * c – [(d + e)]}		MakeEmpty(s)	
{x + (y – [a + b]) * c – [(d + e)]}	**{**	Push(s, '{')	{
{x + **(**y – [a + b]) * c – [(d + e)]}	**(**	Push(s, '(')	{, (
{x + (y – **[**a + b]) * c – [(d + e)]}	**[**	Push(s, '[')	{, (, [
{x + (y – [a + b**]**) * c – [(d + e)]}	**]**	Pop(s)	{, (
{x + (y – [a + b]**)** * c – [(d + e)]}	**)**	Pop(s)	{
{x + (y – [a + b]) * c – **[**(d + e)]}	**[**	Push(s, '[')	{, [
{x + (y – [a + b]) * c – [**(**d + e)]}	**(**	Push(s, '(')	{, [, (
{x + (y – [a + b]) * c – [(d + e**)**]}	**)**	Pop(s)	{, [
{x + (y – [a + b]) * c – [(d + e)**]**}	**]**	Pop(s)	{
{x + (y – [a + b]) * c – [(d + e)]**}**	**}**	Pop(s)	

The Stack ADT

- A **stack** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations
  - **MakeEmpty(S):** Make stack ***S*** be an empty stack
  - **Empty(S):** Determine if ***S*** is empty or not. Return **true** if ***S*** is an empty stack; return **false** otherwise
  - **Push(S, x):** Insert ***x*** at one end of the stack, called its **top**
  - **Top(S):** If stack ***S*** is not empty; then retrieve the element at its **top**
  - **Pop(S):** If stack ***S*** is not empty; then delete the element at its **top**

The Stack ADT

<ADT definition> Stack
Dataholder[items] //Holds data
Top of stack //top of the stack
<**process**>Process name:<returns none>Push (element)
Pre-condition: the stack must not be full
Method: if the stack is full, return error
else increment the top of stack and push the item
Post-condition: stack = stack + element
top of stack = top of stack + 1
<**end process**>
<**process**>Process name:<returns element, error>Pop
Pre-condition: the stack must not be full
Method: if stack is empty, return error
else pop the item at the top of stack and return
decrement the stack
Post-condition:
stack = stack – element
top of stack = top of stack -1
<**end process**>
<**end definition**>

Problems

- Write an algorithm that uses a stack to find the binary equivalent of a decimal positive integer
- Write an algorithm that uses a stack to print all the prime divisors of a given integer in descending order. For example with the integer 2100 the output should be 7 5 5 3 2 2.
  - *[Hint: The smallest divisor greater than 1 of any integer is guaranteed to be a prime.]*

Implementation of Stacks

- *There are several ways of implementing stacks*
  - *Array Implementation*
  - *Linked List Implementation*
- *Each method has advantages and disadvantages*
  - *how close it comes to mirroring the abstract concept of a stack*
  - *how much effort must be made by the programmer and the computer in using it*

Array Implementation of Stacks (1/2)

- ***As a stack is an ordered collection of items, and the C array is also an ordered collection of items, we can use an array to hold a stack items***
- However, a stack and an array are two different things
  - The number of elements in an array is fixed
  - A stack is a dynamic object whose size changes as items are pushed and popped

Array Implementation of Stacks (2/2)

- We can declare an array large enough for the maximum size of the stack
- During its lifetime, the stack can grow and shrink within the space reserved for it
- One end of the array is the fixed bottom of the stack, while the top of the stack constantly shifts as items are pushed and popped
- Hence, we need another field to keep track of the current position of the top of the stack, at any point

Stack Declaration

- We need two variables
  - an array ***items*** of the type of data to be stored in the stack
  - an integer ***top*** that points to top position of the stack

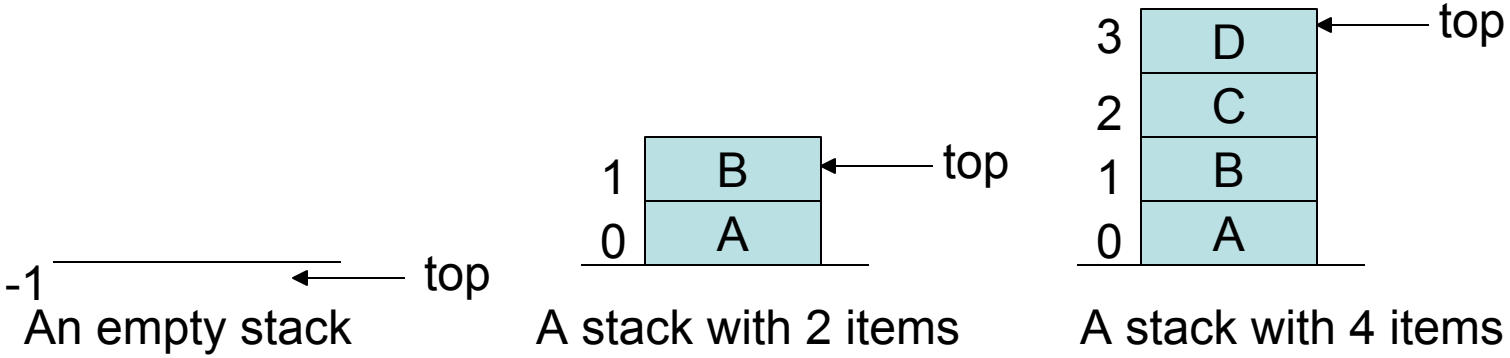
```
#define MAXSTACKSIZE 100

typedef int ItemType;

typedef struct {
   int top;
   ItemType
items[MAXSTACKSIZE];
} StackType;
```

Once this has been done, an actual stack can be created with the declaration

```
StackType st;
```

Making an Empty Stack

- An initially created stack is empty and contains no elements
- We indicate this by ***top*** equalling ***-1***
- To initialize a stack ***st*** to the empty state, we may initially execute ***st.top = -1***



MakeEmpty Operation

- To make a stack empty, we use the **MakeEmpty** function

```
void MakeEmpty(StackType *ps)
{
   ps->top = -1;
}
```

This function should be immediately called, after declaring a stack

```
StackType st;

MakeEmpty(&st);
```

Checking for Empty Stack

- Since we have initialized ***top*** to ***-1*** for empty stack, the condition ***st.top == -1*** may be used to determine if the stack ***st*** is empty or not

```
if (st.top == -1)
{
   /* stack is empty */
}
else
{
  /* stack is not empty */
}
```

Empty Operation

- We define the following **Empty** function for checking if a stack is empty or not

```
int Empty(StackType *ps)
{
   if (ps->top == -1)
      return TRUE;
   else
      return FALSE;
}
```

A check for empty stack can be done as

```
if (Empty(&st))
   /* stack is empty */
else
   /* stack is not empty */
```

NOTE: TRUE and FALSE are symbolic constants having values 1 and 0 respectively

Implementing the Push Operation

- To implement the **Push** operation, we simply increment ***top*** and insert the new item at the position ***top***
- However, the array size is predetermined and we need to check if the stack will outgrow the array or not
  - This can occur when the array is full, that is, when the stack contains as many elements as the array
  - The result to push an item on a full stack is called ***overflow***

The Push Operation

```
void Push(StackType *ps, ItemType newitem)
{
   if (ps->top == MAXSTACKSIZE-1) {
      printf("Stack Overflow\n");
      exit(1);
   }
   ++ps->top
   ps->items[ps->top] = newitem;
}
```

Now to push some items in a stack, we can write

```
StackType st;
MakeEmpty(&st);
Push(&st, 10);
Push(&st, 20);
```

Implement the Pop Operation

- To implement the **Pop** operation, first we check if the stack is empty or not, because we can't pop from an empty stack
- If stack is empty, we print a warning message and halt execution
- Then we remove the top item from the stack, by decrementing ***top***
- A better idea is to return the top item to the calling program

The Pop Operation

```
ItemType Pop(StackType *ps)
{
   if (Empty(ps)) {
      printf("Stack Underflow\n");
      exit(1);
   }
   return ps->items[ps->top];
   ps->top--;
}
```

The Top Operation

```
ItemType Top(StackType *ps)
{
   if (Empty(ps)) {
      printf("Stack Underflow\n");
      exit(1);
   }
   return ps->items[ps->top];
}
```

Checking for FullStack

- Although its not required by the Stack ADT, we define the **Full** operation, that checks if the stack is full or not

```
int Full(StackType *ps)
{
   if (ps->top == MAXSTACKSIZE-1)
      return TRUE;
   else
      return FALSE;
}
```

Problems

- Implement stack using array
- Implement stack using structure
- Implement stack using structure pointer
- Also, Show how to implement a stack of integers in C by using an array ***int s[MAXSTACKSIZE-1]***, where ***s[0]*** is used to contain the index of the top element of the stack, and where ***s[1]*** through ***s[MAXSTACKSIZE-1]*** contain the elements on the stack. Write functions **MakeEmpty**, **Empty**, **Pop**, **Push**, **Full**, and **Top**

Infix Expressions

- **How do you write the sum of *A* and *B*?**
  A + B
- Here '**+**' is an ***operator*** and ***A*** and ***B*** are called ***operands***
- This representation of expressions where we write the operators in between operands is called ***infix expression***

Prefix and Postfix expression

- *In prefix notation, operators precede the operands*
- *In postfix notation, operators follows the operands*
- *Examples*
  - *A + B (Infix)*
  - *+ AB (Prefix)*
  - *AB + (Postfix)*

More on Prefix and Postfix Expressions

- *Both prefix and postfix are parenthesis free expressions*
- *For example*
  - *(A + B) * C Infix form*
  - ** + A B C Prefix form*
  - *A B + C * Postfix form*

Converting an Infix Expression to Postfix

- ***First convert the sub-expression to postfix that is to be evaluated first and repeat this process***
- ***Example***
  - ***A + (B * C)*** *parenthesis for emphasis*
  - A + (BC*) *convert the multiplication*
  - A (BC*) + *convert the addition*
  - ABC*+ *postfix form*

More Examples

- *(A + B) * ((C - D) + E) / F Infix form*
- *(AB+) * ((C – D) + E) / F*
- *(AB+) * ((CD-) + E) / F*
- *(AB+) * (CD-E+) / F*
- *(AB+CD-E+*) / F*
- *AB+CD-E+*F/ Postfix form*

Examples

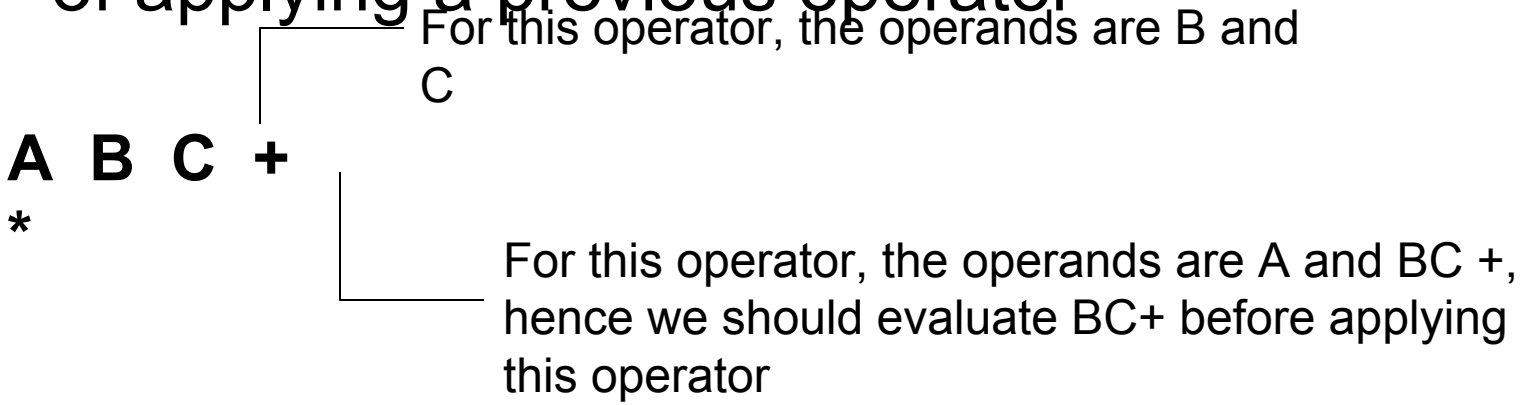
Infix	***Postfix***
A + B	AB +
A + B – C	AB + C -
(A + B) * (C – D)	AB + CD - *
A $ B * C – D + E / F / (G + H)	AB $ C * D – EF / GH + / +
((A + B) * C – (D - E)) $ (F + G)	AB + C * DE - - FG + $
A – B / (C * D $ E)	ABCDE $ * / -

Examples

Infix	***Prefix***
A + B	+ AB
A + B – C	- + ABC
(A + B) * (C – D)	* + AB – CD
A $ B * C – D + E / F / (G + H)	+ - * $ ABCD / / EF + GH
((A + B) * C – (D - E)) $ (F + G)	$ - * + ABC – DE + FG
A – B / (C * D $ E)	- A / B * C $ DE

Evaluating a Postfix Expression

- Each operator in a postfix expression refers to the previous two operands in the expression
  - One of these operands may itself be the result of applying a previous operator



Algorithm for evaluating Postfix expression

- Make an empty stack ***opndstk***
- For each token ***tok*** in the input postfix string
  - If ***tok*** is an operand
    - **Push *tok*** in ***opndstk***
  - Else
    - ***Opnd2*** = **Pop** a operand from ***opndstk***
    - ***Opnd1*** = **Pop** a operand from ***opndstk***
    - ***value*** = Result of applying ***tok*** to ***opnd1*** and ***opnd2***
    - **Push *value*** in ***opndstk***
- **Pop** a value from ***opndstk***, which is the required result

Trace of Evaluation

6 2 3 + - 3 8 2 / + * 2 $ 3 +

tok	*opnd1*	*opnd2*	*value*	*opndstk*
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8

Trace of Evaluation (Cont…)

tok	*opnd1*	*opnd2*	*value*	*opndstk*
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
$	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Converting Infix to Postfix

- Remember, to convert an infix expression to its postfix equivalent, we first convert the innermost parenthesis to postfix, resulting as a new operand
  - In this fashion parenthesis can be successively eliminated until the entire expression is converted
- The last pair of parenthesis to be opened within a group of parenthesis encloses the first expression within the group to be transformed
  - This last in, first-out behavior suggests the use of a stack

Example (1/2)

- Consider A + B * C and (A + B) * C, and their postfix versions ABC * and AB + C *
- In each case the order of operands is same as the order of the operands in the original infix form
  - Hence, while scanning an infix expression, the operands encountered can be immediately placed at the end of the postfix expression

Example (2/2)

- When an operator is encountered it cannot be placed in the postfix expression until its second operand is scanned
  - Therefore, it must be stored away to be retrieved later and inserted in its proper position
- However, even if all the operands of a operator is inserted, it cannot be placed in the postfix expression
  - We need to get the next operator, if it is there, and check for the precedence. The operator with higher precedence should be placed in the postfix expression

Algorithm for Conversion (1/2)

- Make an empty string ***postfix***
- Make an empty stack ***opstk***
- For each token ***tok*** in the input ***infix*** expression
  - If ***tok*** is an operand
    - Add ***tok*** to the end of ***postfix***
  - Else

Continued on next slide….

Algorithm for Conversion (2/2)

- While (!**Empty(*opstk*)** AND Precedence of **Top(*optsk*)** ≥ Precedence of ***tok***
  - ***topopr*** = **Pop(*opstk*)**
  - Add ***topopr*** to the end of ***postfix***
- **Push *tokopr*** at the end of ***postfix***

- While (!**Empty(*opstk*)**)
  - ***toptoken*** = **Pop(*opstk*)**
  - Add ***toptoken*** to the end of ***postfix***
- ***postfix*** is the required postfix expression

Trace of Conversion Algorithm

A + B * C

tok	*postfix*	*opstk*
A	A	
+	A	+
B	AB	+ *
*	AB	+ *
C	ABC	+ *
	ABC *	+
	ABC * +	