Stacks

- The stack ADT
- Stack Implementation
  - using arrays
  - using generic linked lists
  - using List ADT
- Stack Examples

Stacks and Queues

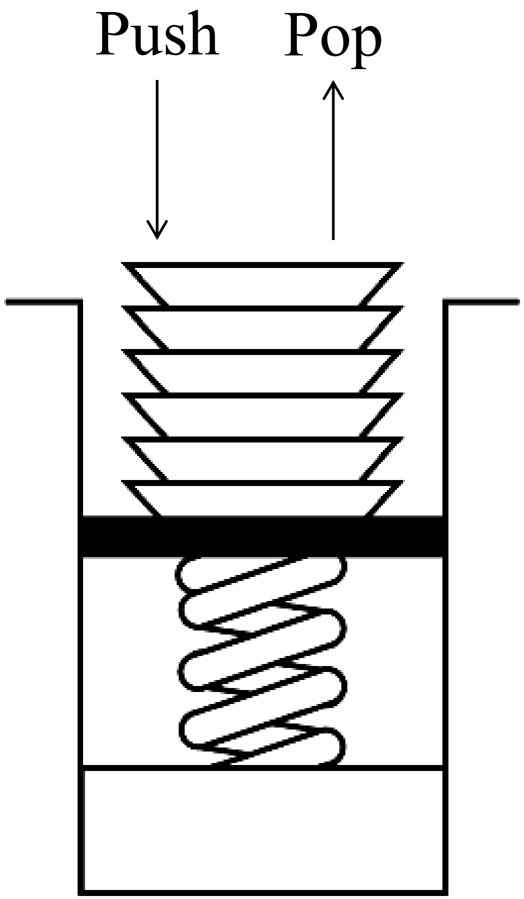
- *Linear* data structures
  - each item has specific *first*, *next*, and *previous* relations with other items in the set
  - examples: arrays, linked lists, vectors, strings
- Stacks and queues are special types of lists with restricted operations
  - restrict how the items are added and removed from the list

Stacks and Queues

- Stacks
  - Last In First Out (LIFO) add/delete semantics
  - *Push()* to add item only to the top or front of the list
  - *Pop()* to remove/delete only the top or front item from the list
- Queues
  - First In First Out (FIFO) add/delete semantics
  - *Enqueue()* to add item to the end of the list
  - *Dequeue()* to remove item from the front of the list
  - will study in next chapter

Stacks

- Analogy of stack
  - stack of dishes in cafeteria
- Several real-world examples
  - subroutine call stack management at runtime
    - implicitly used during recursion
  - language parsing
    - parenthesis matching
  - algebraic expression evaluation



Stack ADT

- Operation Contract for the ADT Stack
  - isEmpty():boolean {query}
  - push(in newItem:StackItemType)
    throw StackException
  - pop() throw StackException
  - pop(out stackTop:StackItemType)
    throw StackException
  - getTop(out stackTop:StackItemType) {query}
    throw StackException

Stack Implementation

- Can use linked lists or arrays for implementing stacks
  - more important is the *interface* that is exposed to the developer!
- A program can use a stack independently of the stack's implementation
- Use axioms to define an ADT stack formally
  - Example: Specify that the last item inserted is the first item to be removed
    - (aStack.push(newItem)).pop()= aStack

Example: Reverse a List

- Traverse and output a list in reverse
- Solution can use either stack or recursion
  - recursion uses the implicit call stack

see C6-reverseList.cpp

Checking for Balanced Braces

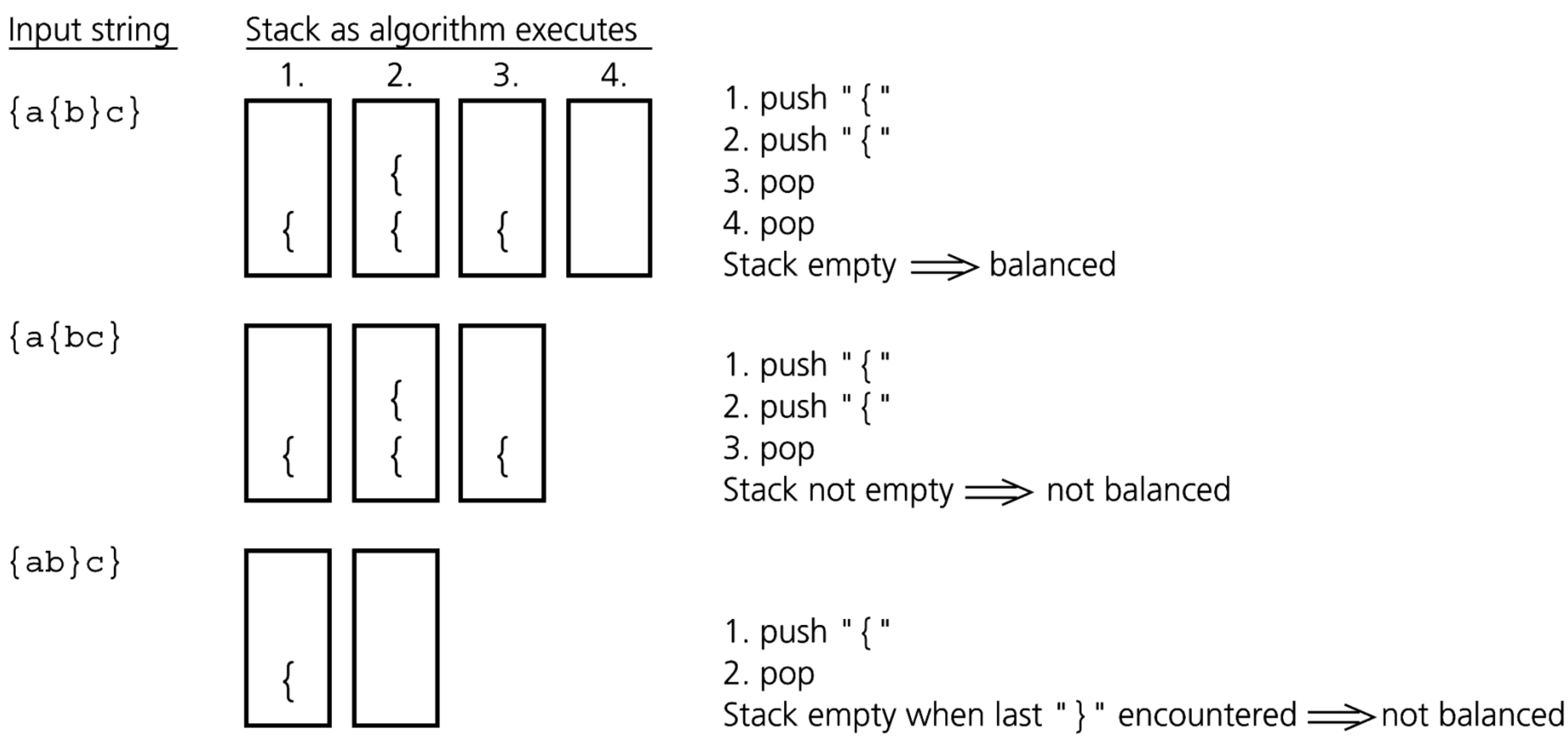
- Problem: Develop an algorithm to read an expression one symbol at a time and check for matching braces
- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces
    - abc{defg{ijk}{l{mn}}op}qr
  - An example of unbalanced braces
    - abc{def}}{ghij{kl}m

Checking for Balanced Braces

- Function performed by parsers/compilers
  - also in several editors, like emacs, vi
- Requirements for balanced braces
  - Each time you encounter a "}", it matches an already encountered "{"
  - When you reach the end of the string, you have matched each "{"
- *Use the stack API as done in the previous problem to develop your own solution.*

Checking for Balanced Braces

- Stepping through the algorithm for 3 expressions



Recognizing Strings in a Language

- L = {w$w' : w is a possibly empty string of characters other than $, w' = reverse(w) }
- A solution using a stack
  - Traverse the first half of the string, pushing each character onto a stack
  - Once you reach the $, for each character in the second half of the string, match a popped character off the stack

see C6-strRecog.cpp

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array – will have size limit
  - A linked list
  - The ADT list
- All three implementations use a `StackException` class to handle possible exceptions

Implementations of the ADT Stack

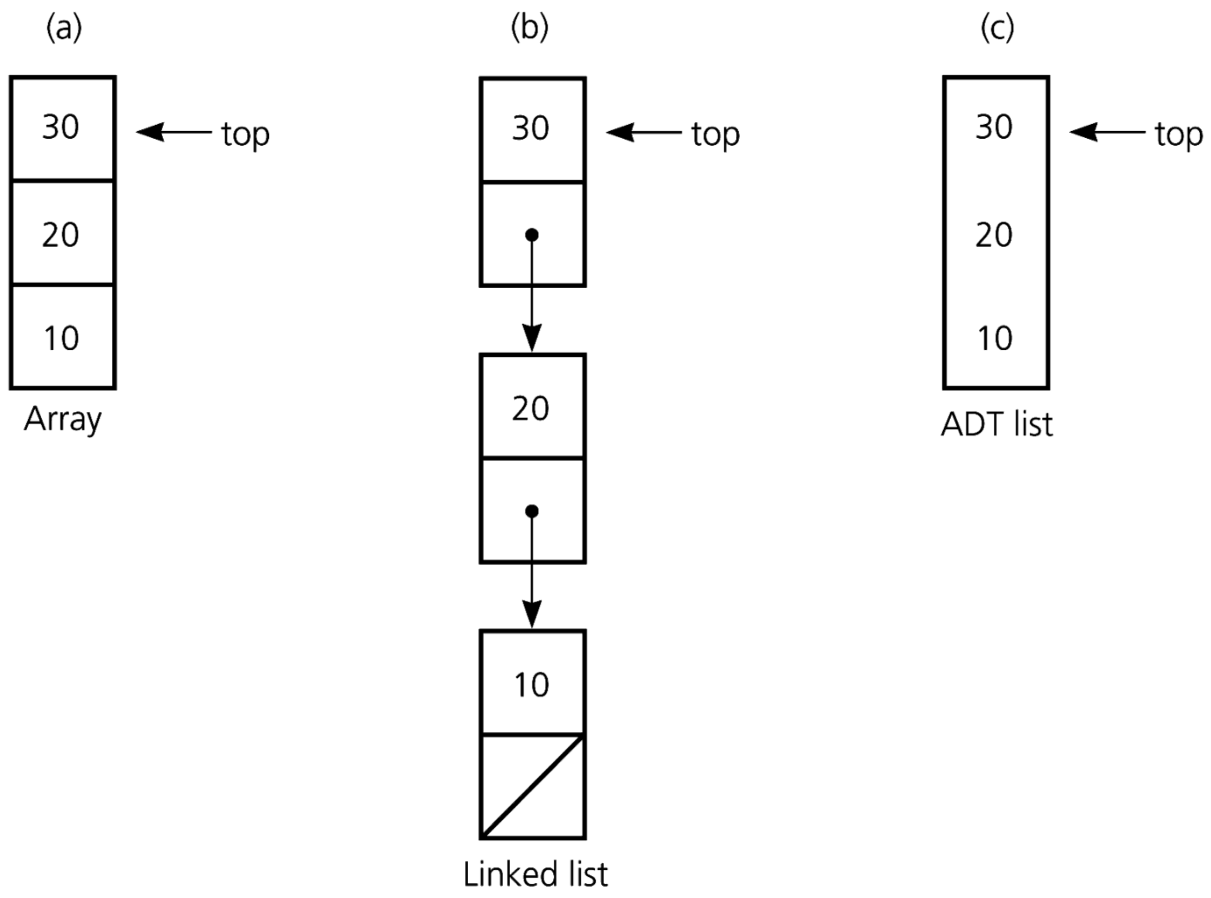


Figure 6-4

Implementations of the ADT stack that use (a) an array; (b) a linked list; (c) an ADT list

An Array-Based Implementation of the ADT Stack

- Private data fields
  - An array of `items` of type `StackItemType`
  - The index `top` to the top item
- Compiler-generated destructor and copy constructor

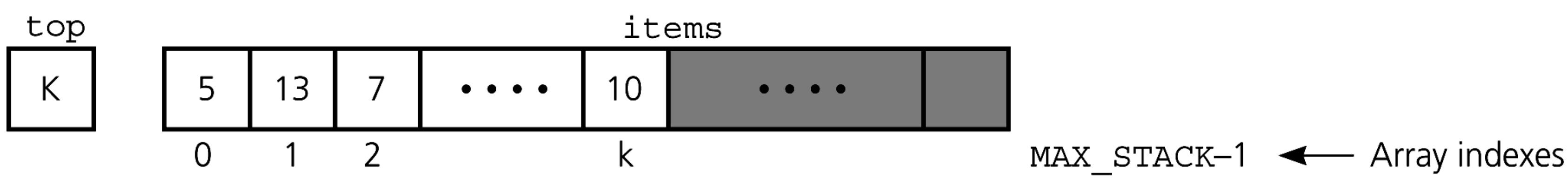
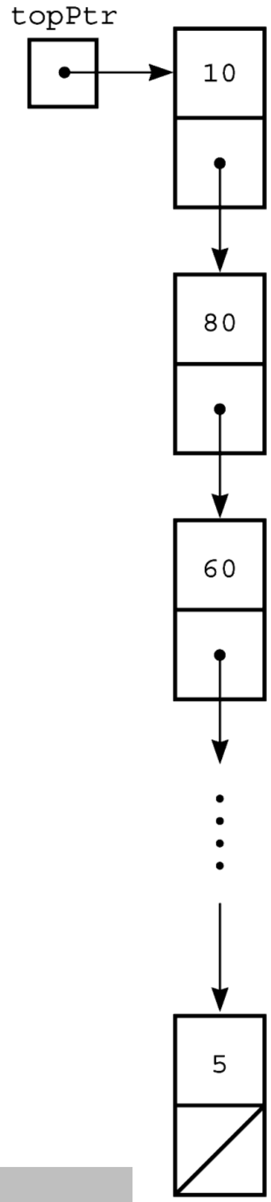


Figure 6-5

An array-based implementation

see C6-StackA.cpp

A Pointer-Based Implementation of the ADT Stack

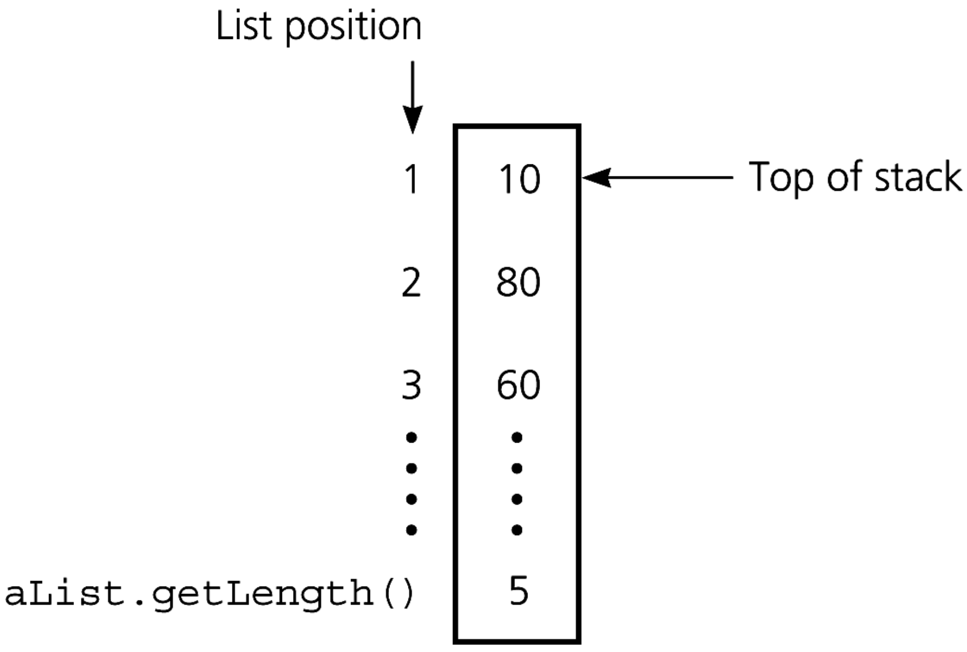


- A pointer-based implementation
  - Enables the stack to grow and shrink dynamically
- `topPtr` is a pointer to the head of a linked list of items
- A copy constructor and destructor must be supplied

see C6-StackP.cpp

An Implementation That Uses the ADT List

- The ADT list can represent the items in a stack
- Let the item in position 1 of the list be the top
  - push(newItem)
    - insert(1, newItem)
  - pop()
    - remove(1)
  - getTop(stackTop)
    - retrieve(1, stackTop)



see C6-StackL.cpp

Comparing Implementations

- Fixed size versus dynamic size
  - A statically allocated array-based implementation
    - Fixed-size stack that can get full
    - Prevents the push operation from adding an item to the stack, if the array is full
  - A dynamically allocated array-based implementation or a pointer-based implementation
    - No size restriction on the stack

Comparing Implementations

- A pointer-based implementation vs. one that uses a pointer-based implementation of the ADT list
  - Pointer-based implementation is more efficient
  - ADT list approach reuses an already implemented class
    - Much simpler to write
    - Saves programming time

Application: Algebraic Expressions

- Infix expressions – most commonly used
  - a*b-c, a+b+c/d
  - need operator precedence rules and parenthesis
- Postfix / prefix expressions
  - definitive, unambiguous grammars
  - no need for precedence rules or parenthesis
- Infix / Prefix / Postfix

Infix	Prefix	Postfix
`a*b-c`	`-*abc`	`ab*c-`
`a*(b-c)`	`*a-bc`	`abc-*`

Application: Algebraic Expressions

- Postfix/prefix expressions are easier to evaluate than infix
- Evaluate an *infix* expression
  – convert the infix expression to *postfix* form
  – evaluate the postfix expression
- We use stack
  – can use either array, pointer, or List ADT based implementation
  – interface of stack ADT is important
  – implementation of the stack ADT is not

Evaluating Postfix Expressions

- When an operand is entered
  - push it onto a stack
- When an operator is entered
  - apply it to the top two operands of the stack
  - pop the operands from the stack
  - push the result of the operation onto the stack
- Simplifying assumptions
  - the string is a syntactically correct postfix expression
  - no unary operators are present
  - no exponentiation operators are present
  - operands are single lowercase letters that represent integer values

Evaluating Postfix Expressions

Key entered	Calculator action		After stack operation: Stack (bottom to top)
2	`push 2`		2
3	`push 3`		2 3
4	`push 4`		2 3 4
+	`operand2 = pop stack`	(4)	2 3
	`operand1 = pop stack`	(3)	2
	`result = operand1 + operand2`	(7)	2
	`push result`		2 7
*	`operand2 = pop stack`	(7)	2
	`operand1 = pop stack`	(2)	
	`result = operand1 * operand2`	(14)	
	`push result`		14

Converting Infix Expressions to Equivalent Postfix Expressions

- Evaluate infix expression by first converting it into an equivalent postfix expression
- Facts about converting from infix to postfix
  - operands always stay in the same order with respect to one another
  - operator will move only "to the right" with respect to the operands
  - all parentheses are removed
- Stack used to hold pending operators until they can be emitted in their right position

Converting Infix Expressions to Equivalent Postfix Expressions

- Steps to process infix expression
  - append an operand to the end of an initially empty string postfixExpr
  - push (onto a stack
  - push an operator onto the stack, if stack is empty; otherwise pop operators and append them to postfixExpr as long as they have a precedence >= that of the operator in the infix expression
  - at), pop operators from stack and append them to postfixExpr until (is popped

see C6-InfixEval.cpp

Infix to Postfix Expressions

- Convert a-(b+c*d)/e to postfix

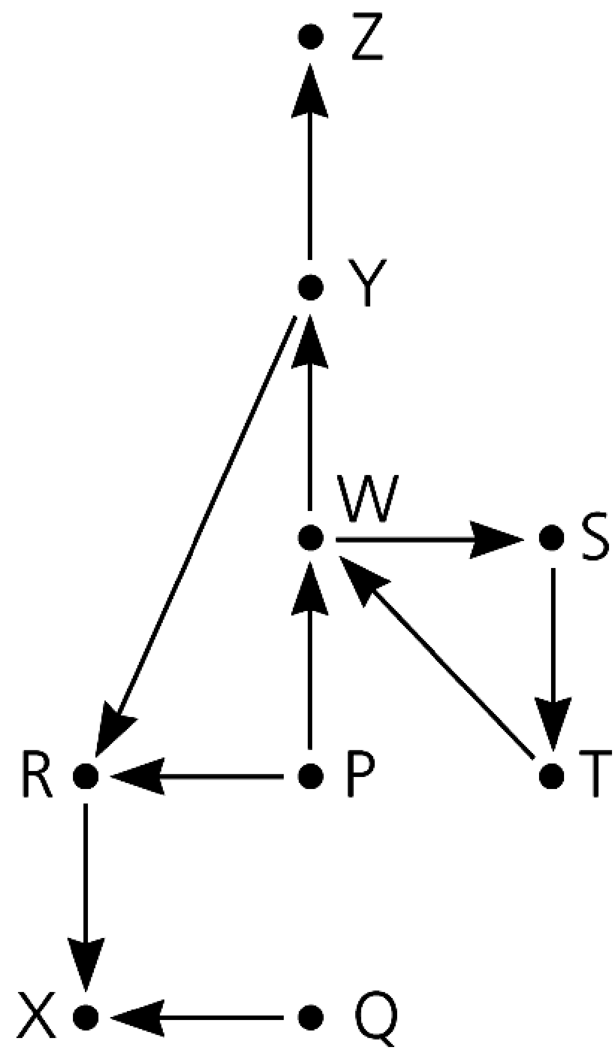
ch	Stack (bottom to top)	postfixExp
a		a
–	–	a
(	– (	a
b	– (	ab
+	– (+	ab
c	– (+	abc
*	– (+ *	abc
d	– (+ *	abcd
)	– (+	abcd*
	– (	abcd*+
	–	abcd*+
/	– /	abcd*+
e	– /	abcd*+e
		abcd*+e/–

Move operators from stack to `postfixExp` until "("

Copy operators from stack to `postfixExp`

Application: A Search Problem

- Indicate whether a sequence of flights exists from the *origin* city to the *destination* city
- The flight map is a graph
  - two adjacent vertices are joined by an edge
  - a directed path is a sequence of directed edges



Stack-Based Nonrecursive Solution

- The solution performs an exhaustive search
  - feasible only for small search spaces
  - beginning at the origin city, try every possible sequence of flights until either
    - we find a sequence that gets to the destination city
    - we determines that no such sequence exists
- Backtracking used to recover from choosing a wrong city

Stack-Based Nonrecursive Solution

Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z

A Recursive Solution

- Possible outcomes of recursive search strategy
  - we eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
  - we reach a city C from which there are no departing flights
  - we go around in circles

A Recursive Solution

- A refined recursive search strategy

```
searchR(in originCity:City,
         in destinationCity:City):boolean
   Mark originCity as visited
   if (originCity is destinationCity)
      Terminate -- the destination is reached
   else
      for (each unvisited city C adjacent to
           originCity)
         searchR(C, destinationCity)
```

The Relationship Between Stacks and Recursion

- Typically, stacks are used by compilers to implement recursive methods
  - during execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm

Summary

- ADT stack operations have a last-in, first-out (LIFO) behavior
- Have a wide range of practical applications
  - algorithms that operate on algebraic expressions
  - flight maps
- A strong relationship exists between recursion and stacks