CMPT 225

Stacks-part2

Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

Let e be the next element in the infix expr

1. if e is an operand append it to the output string.
2. if e is “(“ push e to the stack
3. if e is an operator
    1. If stack is empty push e into the stack
    2. Otherwise, pop operators of greater or equal precedence from the stack and append them to output string-stop when you see a “(“ or an operator of lower precedence.
4. if e is “)” pop operators from the stack and append them to the output string until you see the matching “(“.
5. when you reach the end of the string pop all the operators and attach them to the output.

```
for ( each character ch in the infix expr){
   switch (ch) {
        case operand:
                postfixExp += ch; break;
        case '(':
                aStack.push(ch); break;
        case ')':
                while (top of the stack is not '('
                        postfixExp += aStack.pop()
                aStack.pop();
                break;
        case operator:
                while (!aStack.isEmpty() && top of the stack
                is not '(' && preced(ch) <= preced(top of the
                 stack))
                        postfixExpr += aStack.pop();
                aStack.push(ch);
                break;
   }
   while (!aStack.isEmpty())
        postfixExp += aStack.pop();
}
```

Converting Infix Expressions to Equivalent Postfix Expressions

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	–(	a	
b	–(	ab	
+	–(+	ab	
c	–(+	abc	
*	–(+*	abc	
d	–(+*	abcd	
)	–(+	abcd*	Move operators
	–(	abcd*+	from stack to
	–	abcd*+	**postfixExp** until " ("
/	–/	abcd*+	
e	–/	abcd*+e	Copy operators from
		abcd*+e/–	stack to **postfixExp**

Figure 7-9

A trace of the algorithm that converts the infix expression *a* - (*b* + *c* * *d*)/*e* to postfix form

Recognizing Strings in Languages

- Recognize whether a particular string is in the language:

L={ w$w': w is possibly an empty string of characters other than $ and w'=reverse(w)}

- Example

  ABC$CBA, $, AAB$BAA, …

Application: A Search Problem

- High Planes Airline Company (HPAir)
  - Problem
    - For each customer request, indicate whether a sequence of HPAir flights exists from the origin city to the destination city

Representing the Flight Data

- The flight map for HPAir is a graph
  - Adjacent vertices
    - Two vertices that are joined by an edge
  - Directed path
    - A sequence of directed edges

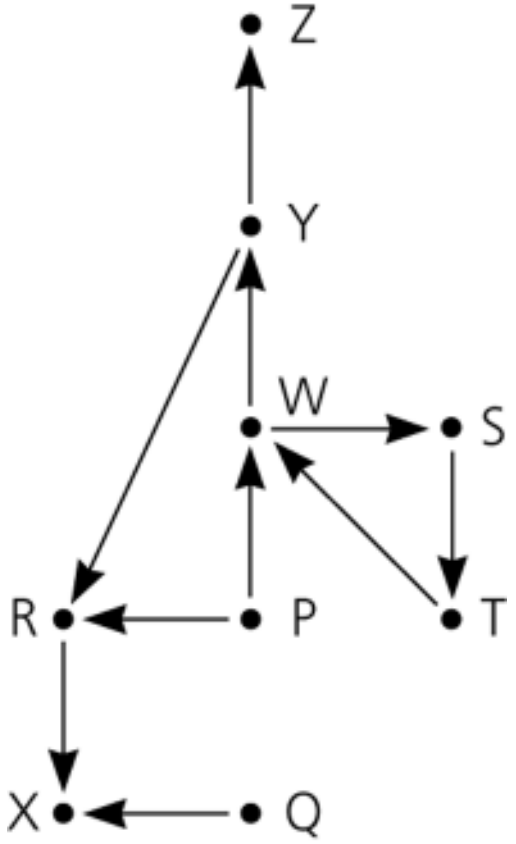

Figure 7-10

Flight map for HPAir

A Nonrecursive Solution that Uses a Stack

- The solution performs an exhaustive search
  - Beginning at the origin city, the solution will try every possible sequence of flights until either
    - It finds a sequence that gets to the destination city
    - It determines that no such sequence exists
- The ADT stack is useful in organizing an exhaustive search
- Backtracking can be used to recover from a wrong choice of a city

1. Start from the origin
2. Select an arbitrary city and fly to it.
3. Repeat step 2 until you reach the destination

Possible outcomes:

1. You eventually reach the destination
2. You reach a city from which there is no departing flight.
3. You go around a circle.

- The first outcome is lucky and only happens if you select a correct flight at each step.
- Since we don’t know whether we are making the right decision at each step, we need to keep track of our decision so that if we made a mistake we could go back and make another decision.
- Using stack can help us keep track of our decisions.

```
aStack.push(originCity)
while (destination is not on top && !aStack.isEmpty()){
   if ( there is no flight out of the city on top of stack){
        aStack.pop();
   }
   else{
        Select a destination city C from the city on top of the stack.
        aStack.push( C );
   }
}

if (aStack.isEmpty())
   return false;
else
   return true;
```

We may end up in a loop with the above algorithm.

To avoid the loop we must make sure we don't visit a city more than once.

```
aStack.push(originCity)
Mark the originCity as visited
while (destination is not on top && !aStack.isEmpty()){
   if ( there is no flight out of the city on top of stack to an
         unvisited city){
        aStack.pop();
   }
   else{
        Select an unvisited destination city C from the city on top of
         the stack.
        aStack.push( C );
        Mark C as visited;
   }
}
if (aStack.isEmpty())
   return false;
else
   return true;
```

A Nonrecursive Solution that Uses a Stack

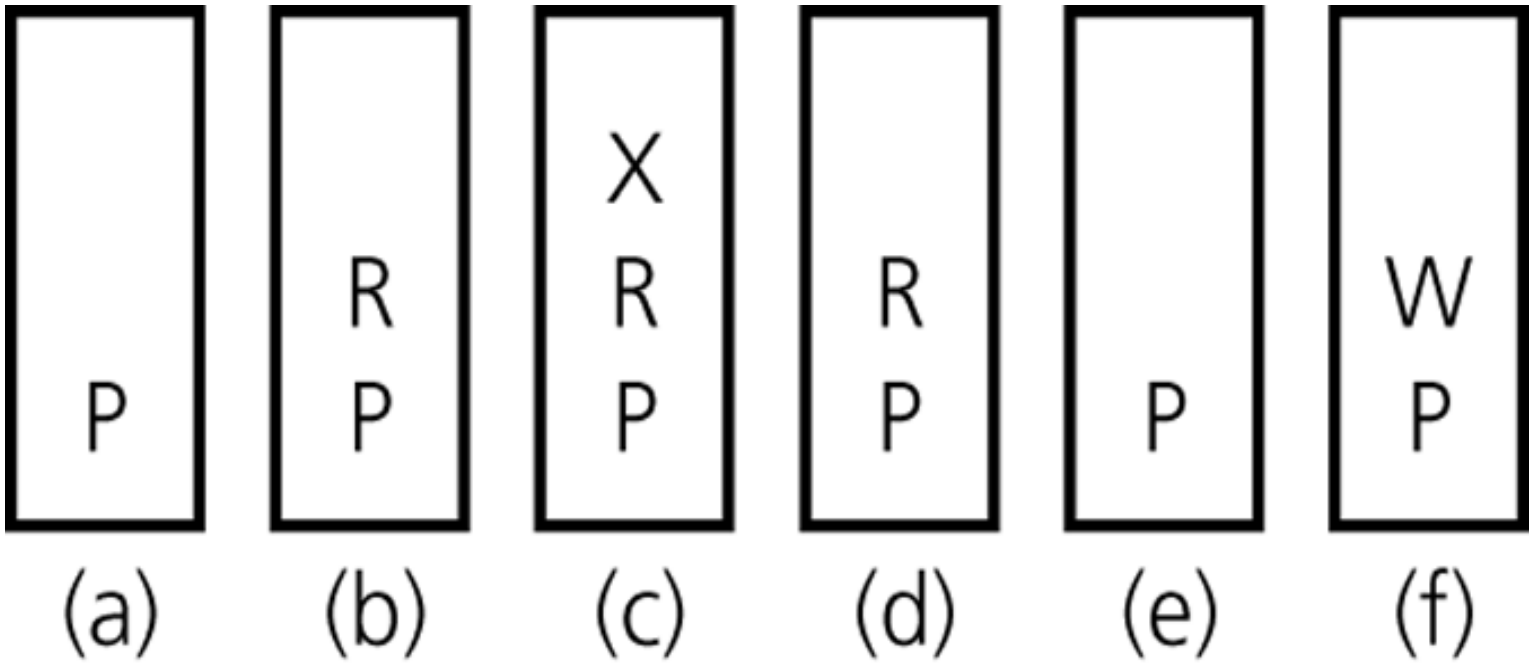

Figure 7-11

The stack of cities as you travel a) from *P*; b) to *R*; c) to *X*; d) back to *R*; e) back to *P*; f) to *W*

A Nonrecursive Solution that Uses a Stack

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

Figure 7-13

A trace of the search algorithm, given the flight map in Figure 6-9

A Recursive Solution

```
boolean searchR(originCity, destinationCity){
     Mark originCity as visited
     if (originCity is destinationCity) {
       return true
     }
     else {
       for (each unvisited city C adjacent to
              originCity) {
         if (searchR(C, destinationCity))
              return true
       }
       return false
     }
   }
```

The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of 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
- Algorithms that operate on algebraic expressions are an important application of stacks
- A stack can be used to determine whether a sequence of flights exists between two cities
- A strong relationship exists between recursion and stacks