Chapter 6-4, Lecture notes

Application: Algebraic Expressions

1. There are two problems we can solve quite nicely using the ADT stack.
   1.1 Use the stack operations but do not assume any particular implementation (do that as a last step).

   1.2 Let's first try using stacks to solve converting infix expressions into postfix expressions (postfix is less ambiguous than infix and easier for a computer to operate on).
      1.2.1 For our problem, we'll keep the binary operators to: *, /, + and - (no exponentiation nor unary operators like a –x).
      1.2.2 Let's develop the algorithm for evaluating a postfix expression then develop the algorithm for transforming an infix operation into postfix.
      1.2.3 Both will let us evaluate an infix expression.

   1.3 Some calculators make you enter in postfix expressions.
      1.3.1 For example, if you wanted to get the value of:

         2 * (3 + 4)

      1.3.2 You would need to enter the values as a postfix expression of:

         2 3 4 + *

         Any binary operation would apply to the two values immediately preceding it.
      1.3.3 The ADT stack would allow popping off the two values, getting an answer and pushing the answer back onto the stack for the next operation.
      1.3.4 The postfix calculator would work like (figure 6-8, page 312):

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) operand1 = pop stack (3)	2 3 2
	result = operand1 + operand2 (7) push result	2 2 7
*	operand2 = pop stack (7) operand1 = pop stack (2)	2
	result = operand1 * operand2 (14) push result	 14

1.3.5 The answer of 14 is on the top of the stack.
1.3.6 And you can create the algorithm to do the postfix calculator.
1.3.6.1 However, we need to formalize what the calculator would look like.
1.3.6.2 To do so, assume that:
- The string is a syntactically correct postfix expression
- No unary operators are present
- No exponentiation operators are present
- Operands are single uppercase letters that represent integer values

1.4 The pseudo code algorithm would be:

```
for (each character ch in the string)
{       if (ch is an operand)
                Push value that operand ch represents onto stack
        else // ch is an operator named op
        {       // evaluate and push the result
                operand2 = top of stack
                Pop the stack
                operand1 = top of stack
                Pop the stack
                result = operand1 op operand 2
                Push result onto stack
        } // end if
} // end for
```

1.5 When the algorithm finishes, the value of the expression will be on top of the stack.

2 Let's try converting infix expressions to the equivalent postfix expressions.
2.1 Infix operations can look like: (a + b) * c/d – e

2.2 They allow parentheses, operator precedence, and left to right associations.

2.3 The compiler translates infix expressions to postfix form so that machine language operations can be done on the expression.

2.3.1 Knowing how to go from infix to postfix will give you an insight into some compiler operations.
2.3.2 To manually convert, the items to keep in mind are:
- The 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; that is, if, in the infix expression, the operand x precedes the operator

op, it is also true that in the postfix expression the operand x precedes the operator op
- All parentheses are removed

2.4 Let's look at the first attempt at the pseudo code to translate from infix to postfix:

```
Initialize postfixExp to the null string
for (each character ch in the infix expression)
{       switch(ch)
        {       case ch is an operand:
                        Append ch to the end of postfixExp
                        Break
                case ch is an operator:
                        Store ch until you know where to place it
                        break
                case ch is '(' or ')':
                        Discard ch
                        Break
        } // end switch
} // end for
```

2.5 The part of just discarding the parentheses does not seem correct.

2.5.1 The parentheses actually play an important part in figuring out the placement of the operators – they indicate an isolated subexpression in the expression.

2.5.2 So we need to evaluate the subexpression or, in other words, the parentheses tell the rest of the expression:

2.5.3 You can have the value of this subexpression after it is evaluated; simply ignore everything inside.

2.6 The parentheses are just only factor in the placement of operators, the other factors are precedence and left-to-right association.

2.6.1 The conversion can be complicated and needs the following to figure out the placement:

2.6.1.1 When you encounter an operand, append it to the output string postfixExp.
- Justification: The order of the operands in the postfix expression is the same as the order in the infix expression, and the operands that appear to the left of an operator in the infix expression also appear to the left in the postfix expression

2.6.1.2 Push each "(" onto the stack

2.6.1.3 When you encounter an operator, if the stack is empty, put the operator onto the stack.
- However, if the stack is not empty, pop operators of greater or equal precedence from the stack and append them to postfixExp.
- You stop when you encounter either a "(" or an operator of lower precedence or when the stack becomes empty.
- You then push the new operator onto the stack.
- Thus, this step orders the operators by precedence and in accordance with left-to-right association.
- Note: you continue popping from the stack until you hit an operator that is strictly lower precedence – equality does NOT count.
- With equality, you do the left-to-right association and the equal operator on the stack would be done before the operator "in hand".

2.6.1.4 When you encounter a ")", pop operators off the stack and append them to the end of postfixExp until you encounter the matching "(".
- Justification: Within a pair of parentheses, precedence and left-to-right association determine the order of the operators, and Step 3 has already ordered the operators in accordance with these rules.

2.6.1.5 When you read the end of the string, you append the remaining contents of the stack to postfixExp.

2.6.2 The revised pseudo code algorithm for converting infix into postfix is:

```
for (each character ch in the infix expression)
{       switch(ch)
        {       case operand:  // append operand to end of PE
                        postfixExp = postfixExp + ch
                        break
                case '(':       // save '(' on stack
                        aStack.push(ch)
                        break
                case ')':       // pop stack until matching '('
                        while (top of stack is not '(')
                        {       postfixExp = postfixExp + (top of aStack)
                                aStack.pop()
                        } // end while

                        aStack.pop()    // remove the open parenthesis
                        break
                case operator:  // process stack operators of greater precedence
                        while (!aStack.isEmpty() and top of stack is not '(' and
                                precedence(ch) <= precedence(top of stack))
                        {       postfixExp = postfixExp + (top of stack)
                                aStack.pop()
                        } // end while
```

```
                    aStack.push(ch)         // save new operator
                    break
        } // end switch
} // end for

// append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty())
{       postfixExp = postfixExp + (top of aStack)
        aStack.pop()
} // end while
```

2.7 Note: the above algorithm assumes that the given infix expression is correct and therefore not worry about the StackException (stack empty).

2.7.1 In reality, you would need to be concerned with it.

2.7.2 Figure 6-9 on page 315 traces the algorithm working on the infix expression of:

a – (b + c * d) / e

ch	Stack (bottom to top)	postfixExp	
a		a	
-	-	a	
(	- (	a	
b	- (	a b	
+	- (+	a b	
c	- (+	a b c	
*	- (+ *	a b c	
d	- (+ *	a b c d	
)	- (+ - (-	a b c d * a b c d * + a b c d * +	Move operators from stack to postfixExp until "("
/	- /	a b c d * +	
e	- /	a b c d * + e a b c d * + e / -	Copy operators from stack to postfixExp

Chapter 6-5, Application: A Search Problem

1 Let's try a general search problem that uses stacks.

1.1 We'll try the non-recursive version first then a recursive version.

1.2 In that way, hopefully, you will see the relationship between stacks and recursion.

1.2.1 The High Planes Airline Company (HPAir) wants a program to process customer requests to fly from some origin city to some destination city.

1.2.2 So, we'll simplify the problem to keep the focus of using stacks in the problem.
1.2.3 For each customer request, just indicate whether a sequence of HPAir flights from origin city to the destination city exists.
1.2.4 We could try a more realistic approach of actually producing an itinerary but that is a programming problem given at the end of the chapter.

1.3 We'll have three input text files that specify all of the flight information for the airline as follows:

- The names of cities that HPAir serves
- Pairs of city names, each pair representing the origin and destination of one of HPAir's flights
- Pairs of city names, each pair representing a request to fly from some origin to some destination

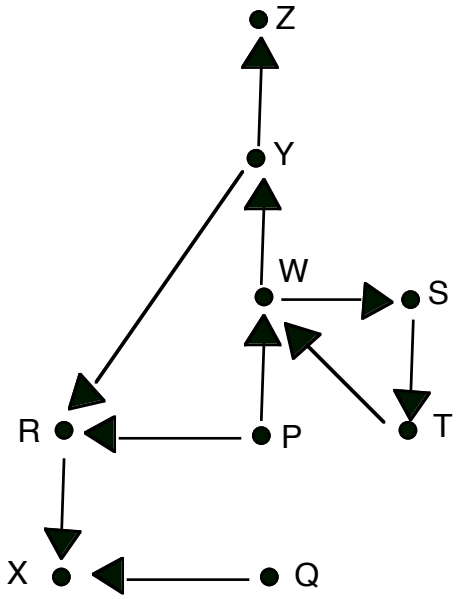
1.4 The program should then produce output such as:

Request is to fly from Providence to San Francisco.
HPAir flies from Providence to San Francisco.

Request is to fly from Philadelphia to Albuquerque.
Sorry. HPAir does not fly from Philadelphia to Albuquerque.

Request is to fly from Salt Lake City to Paris.
Sorry. HPAir does not serve Paris.

1.5 Figure 6-1 on page 318 shows a flight map using single letters to represent cities and their flights:



1.5.1 It is a “directed graph”. An arrow from city A to B indicates a flight from A to B.
1.5.2 That would mean that B is adjacent to A and the path from A to B is called a “directed path”.
1.5.3 However, A is NOT adjacent to B is there is no directed path from B to A.

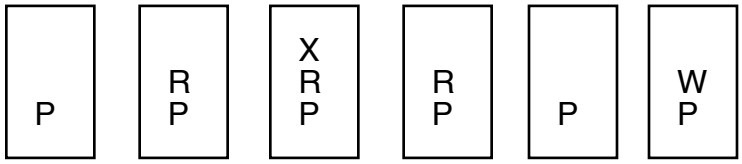
1.6 Looking at the graph, you can see that a customer could fly from city P to city Z by going through cities W and Y. Or P → W, W → Y and Y → Z.
1.6.1 However, that was not an “exhaustive search” which is what we want the solution to be.
1.6.2 With an exhaustive search, we start from A to go to D. If we find it immediately, we are done.
1.6.3 But if we must go from A to B then we must try going from B to D.
1.6.4 If we made it to D, we are done.
1.6.5 However, if you wound up at city C, you must try yet again.

1.7 There are possible outcomes to the above strategy such as:
- You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
- You reach a city C which there are no departing flights
- You go around in circles. With the above graph, you could have gone from W to S to T and back to W to start again.

1.8 If you find the path, then everyone’s happy.
1.8.1 However, since HPAir does not fly between all pairs of cities (i.e. they’re cheap), you cannot always expect to find a path.
1.8.2 For instance, city P cannot be flown TO, just FROM.

2 We need to make a relatively sophisticated algorithm to prevent outcomes of 2 and 3 from happening.
2.1 If you inadvertently start at P, want to go to W but tried R instead, you would go to a city, such as X, with no out bound flights, you need to “back up” by one city and try another path.
2.2 If that did not work, then you would need to “back up” by two cities.
2.3 This time you would find W and you have your destination. The easiest way to “back up” is to store the cities visited onto a stack.
2.4 If you hit a dead end, you could simply “pop off” that city and try another path.

2.5 The above scenario using a stack is like so (figure 6-11 on page 320):



3 The algorithm so far would be:

```
aStack.createStack()

aStack.push(originCity)        // push origin city onto aStack

while (a sequence of flights from the origin to the destination has not been found)
{       if (you need to backtrack from the city on the top of the stack)
                aStack.pop()
        else
        {       Select a destination city C for a flight from the city on the top of the stack
                aStack.push(C)
        } // end if
} // end while
```

3.1 The stack would contain the cities visited with the top of the stack being the latest city visited.

3.2 Or in other words:

The stack contains a directed path from the origin city at the bottom of the stack to the city at the top of the stack.

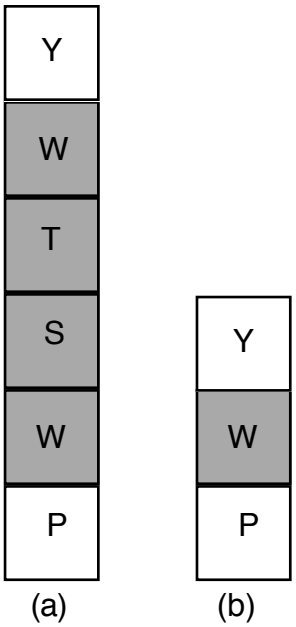
4 You can retrace your steps as far back as you need to.

4.1 However, you do NOT want to visit a city that the search has already visited.

4.2 Therefore, you must backtrack from a city whenever there are no more unvisited cities to fly to.

4.3 Two cases why you never want to visit a city a second time:

- If you visited city C and it is still somewhere in the stack – i.e. part of the sequence of cities that you are currently exploring – you do not want to visit C again.
  - And sequence that would go from C to A to B and back to C would have the A and B cut out.
    An example of this from the graph would be going from P to Y and visiting W.
  - W could go to T then S then W again, thus having a loop.
  - You could eliminate T and S from the stack.
  - In figure 6-12 page 321, part a is the loop of going from W to S to T to W again, part b would be the backtracking that eliminates S and T:



- If you have visited city C but it is no longer in the stack – because you backtracked from it and popped it from the stack – you do not want to visit C again.
  - The reasons would be subtle.
  - Two cases to consider: 1) if you backtracked from C because there were no flights out of C then you most certainly don't want to visit it again 2) if you backtracked from C because all the cities adjacent to it had been visited and you did not get to your destination.
  - You most certainly do not want to go there again and possibly get into a loop.

4.4 To implement the "no visit city twice" rule, you merely mark the cities you have visited.

4.4.1 Thus the algorithm looks like:

```
aStack.createStack()
Clear marks on all cities

aStack.puch(originCity)        // push origin city onto aStack
Mark the origin as visited

while (a sequence of flights from the origin to the destination has not been found)
{       // Loop invariant: The stack contains a directed path from the origin city at the
        // bottom of the stack to the city at the top of the stack
        if (no flights exist from the city on the top of the stack to unvisited cities)
                aStack.pop()   // backtrack
        else
        {       Select an unvisited destination city C for a flight from the city on the
                        top of the stack
```

```
            aStack.push(C)
            Mark C as visited
        } // end if
} // end while
```

4.4.2 The problem with the above algorithm is that it assumes there is a path from a city to another city – not valid.

4.4.3 So we need to handle the case where all cities have been exhausted and you pop off the origin city and thus there is no path.

```
+searchS(in originCity:City, in destinationCIty:City):boolean
// Searches for a sequence of flights from originCity to destinationCity
        aStack.createStack()
        Clear marks on all cities

        aStack.push(originCity)         // push origin onto aStack
        Mark the origin as visited

        while (!aStack.isEmpty() and
                        destinationCity is not at the top of the stack)
        {       // Loop invariant: The stack contains a directed path from the origin city at
                // the bottom of the stack to the city at the top of the stack

                // originCIty to destinationCity
                if (no flights exist from the city on the top of the stack to unvisited cities)
                        aStack.pop()    // backtrack

                else
                {       Select an unvisited destination city C fro a flight from the city
                                on the top of the stack
                        aStack.push(C)
                        Mark C as visited
                } // end if
        } // end while

        if (aStack.isEmpty())
                return false      // no path exists
        else
                return true       // path exists
```

4.5 The algorithm does not specify the order of selection for the unvisited cities.

4.5.1 It really doesn't matter since it does not affect the outcome.

4.5.2 Let's try alphabetic.

4.5.3 The ADT specs would be:

+createFlightMap()
// Creates an empty flight map.

+destroyFLightmap()
// Destroys a flight map.

+readFlightMap(in cityFileName:string, in flightFileName:string)
// Reads flight information into the flight map.

+displayFlightMap()
// Displays flight information.

+displayAllCIties()
// Displays the names of all cities that HPAir serves.

+displayAdjacentCities(in aCity:City)
// Displays all cities that are adjacent to a given city.

+markVisited(in aCity:City)
// Marks a city as visited

+unvisitAll()
// Clears marks on all cities

+isVisted(in aCity:City):boolean
// Determines whether a city was visited.

+insertAdjacent(in aCity:City, in adjCity:City)
// Inserts a city adjacent to another city in a flight map

+getNextCity(in fromCity:City, out nextCity:City):boolean
// Determines the next unvisited city, if any, that is adjacent to a given city. Returns
// true if an unvisited adjacent city was found, false otherwise.

+isPath(in originCity:City, in destinationCity:City):boolean
// Determines whether a sequence of flights exists between two cities.

4.5.4 A trace using alphabetic would be in figure 6-13 on page 324 like so:

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

4.5.5 The following C++ implement of searchS algorithm uses integers to represent cities.

```
/** Determines whether a sequence of flights between two cities exists. Nonrecursive
 *      stack version
 *  @pre originCity and destinationCity both exist in the Map
 *  @post None
 *  @param originCity   Number of the origin city
 *  @param destinationCity  Number of the destination city
 *  @return True is a sequence of flights exists from originCity to destinationCity;
 *              otherwise returns false. Cities visited during the search are marked
 *              as visited in the flight map
 *  @note  Uses a stack for the city numbers of a potential path. Calls unvisitedAll,
 *              markVisited, and getNextCity */
bool: Map::isPath(int originCIty, int destinationCity)
{       Stack   aStack;
        int     topCity, nextCity;
        bool    success;

        unvisitedAll();         // clear marks on all cities

        // push origin city onto aStack, mark it visited
        try
        {       aStack.push(originCity);
                markVisited(originCity);

                aStack.getTop(topCity);
                while (!aStack.isEmpty() && (topCity != destinationCity))
                {       // Loop invariant: The stack contains a directed path from origin
                        // city at  the bottom of the stack to the city at the top of the stack
                        success = getNextCity(topCity, nextCity);

                        if (!success)
                                aStack.pop();  // no city found; backtrack

                        else
                        {       aStack.push(nextCity);
                                markVisited(nextCity);
                        } // end if
```

```
                if (!aStack.isEmpty())
                        aStack.getTop(topCity);
            } // end while

            if (aStack.isEmpty())
                return false;    // no path exists
            else
                return true;     // path exists
            }
            catch (StackException e)
            {       cout << e.what() << endl;
            } // end catch
        } // end try
} // end isPath
```

4.6 The programming problem 11 would give implementation details.

5 What about a recursive solution? One recursive strategy is:

5.1 To fly from the origin to the destination:

```
Select a city C adjacent to the origin
Fly from the origin to city C
If (C is the destination city)
        Terminate – the destination is reached
else
        Fly from city C to the destination
```

5.2 Now the same outcomes would occur with the recursive solution as with the non-recursive one:

- You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination
- You reach a city C from which there are no departing flights
- You go around in circles

5.3 The first outcome would be a case to the recursive algorithm.

5.3.1 But the algorithm does not address what happens with the second and third outcomes.

5.3.2 So, let's do some refinement:

```
+searchR(in originCity:City, in destinationCity:City):boolean
// Searches for a sequence of flights from originCity to destinationCity

        Mark originCity as visited
```

```
If (originCity is destinationCity)
        Terminate – the destination is reached
else
        for (each unvisted city C adjacent to originCity)
                searchR(C, destinationCity)
```

5.4 Now if each city adjacent to a sample city, such as X, is visited and does not lead to the destination, then the routine just returns to city W – i.e. it backtracks to W.

5.4.1 The for loop with city W will continue on its merry way to go to the next city. If the algorithm backtracks all the way to the origin city then it terminates and you can conclude there is no route.

5.4.2 The following C++ method implements the searchR algorithm:

```
bool Map::isPath(int originCity, int destinationCity)
{
        int nextCity;
        bool success, done;

        // mark the current city as visited
        markVisited(originCity);

        // base case: the destination is reached
        if (originCity == destinationCity
                return true;

        else    // try a flight to each unvisited city
        {       done = false;
                success = getNextCity(originCity, nextCity);

                while (success && !done)
                {       done = isPath(nextCity, destinationCity);
                        if (!done)
                                success = getNextCity(originCity, nextCity);
                } // end while

                return done;
        } // end if
} // end isPath
```

5.5 In a sense, the above routine uses a stack – the stack is just not visible.

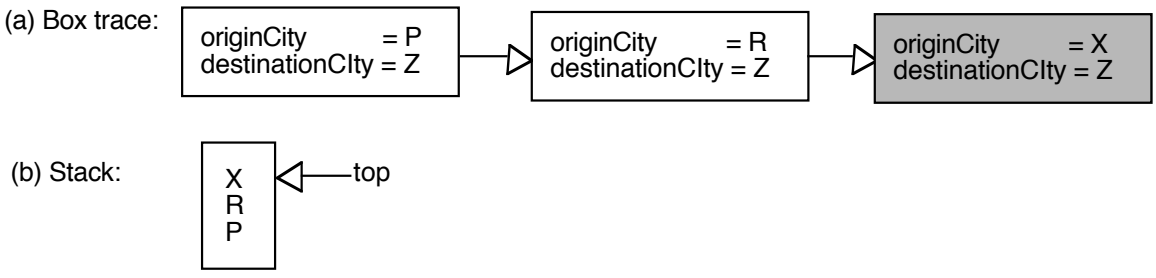
Chapter 6-6, Application: The Relationship Between Stacks and Recursion

1 With the non-recursive algorithm, you used an explicitly defined stack to solve the problem.

1.1 With the recursive algorithm, the stack was hidden but there, nevertheless.

1.2 The two search algorithms have the three key aspects:

- Visiting a new city.
  - The recursive algorithm searchR visits a new city C by calling searchR(C, destinationCity).
  - The algorithm searchS visits a city C by pushing C onto a stack.
  - If you were to do a box trace of searchR, you would see the city being put onto the hidden stack.
  - Figure 6-15 page 320 show the box trace for searchR (part a) and the stack for searchS (part b).
  - It is below:
- Backtracking.
  - Both search algorithms attempt to visit an unvisited city that is adjacent to the current city.
  - Notice that this current is the value associated ith the formal argument originCity in the deepest (rightmost) box of searchR's box trace.
  - Likewise, the current city is on the top of the searchS's stack.
  - The current city is X.
  - The searchR backtracks by returning from the current recursive call.
  - The searchS backtracks by explicitly popping from its stack.
- Termination.
  - The search algorithms terminate either when they reach the destination city or when they exhaust all possibilities.
  - All possibilities are exhausted when, on backtracking, there are no unvisited adjacent cities left.
  - With searchR, that happens when all boxes have been crossed off the box trace and the return occurs in the original call to the routine.
  - With searchS, no unvisited cities are adjacent to the origin when the stack is emptied.



2 Typically, stacks are used to implement recursive functions.

2.1 The compiler will usually use a stack to implement a recursive function.
2.2 When a recursive call is done, the implementation must remember certain information such as the local environment of arguments and local variables and where to return back to (like with the box trace).

2.3 During execution, the complied program must manage these boxes of information (or activation records) just as the box trace does.

2.4 Each recursive call generates an activation record that is pushed onto the stack.

2.5 When the return is made, the activation record is popped off the stack.

3 Instead of recursion, you can use stacks explicitly when implementing a nonrecursive version of a recursive.
3.1 That would be done for efficiency.
3.2 If you take an advanced course, such as compiler writing, you'll see recursion removal as a formal topic.