CHAPTER 7

Stacks

In the previous chapters, we used the Python list and linked list structures to implement a variety of container abstract data types. In this chapter, we introduce the stack, which is a type of container with restricted access that stores a linear collection. Stacks are very common in computer science and are used in many types of problems. Stacks also occur in our everyday lives. Consider a stack of trays in a lunchroom. When a tray is removed from the top, the others shift up. If trays are placed onto the stack, the others are pushed down.

7.1 The Stack ADT

A ***stack*** is used to store data such that the last item inserted is the first item removed. It is used to implement a *last-in first-out* (LIFO) type protocol. The stack is a linear data structure in which new items are added, or existing items are removed from the same end, commonly referred to as the ***top*** of the stack. The opposite end is known as the ***base***. Consider the example in Figure 7.1, which

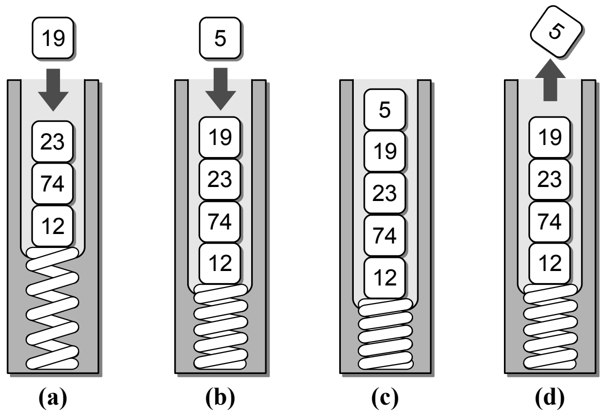

Figure 7.1: Abstract view of a stack: (a) pushing value 19; (b) pushing value 5; (c) resulting stack after 19 and 5 are added; and (d) popping top value.

illustrates new values being added to the top of the stack and one value being removed from the top.

Define **Stack ADT**

A *stack* is a data structure that stores a linear collection of items with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the *top* of the stack. An *empty stack* is one containing no items.

- `Stack()`: Creates a new empty stack.
- `isEmpty()`: Returns a boolean value indicating if the stack is empty.
- *`length`*`()`: Returns the number of items in the stack.
- `pop()`: Removes and returns the top item of the stack, if the stack is not empty. Items cannot be popped from an empty stack. The next item on the stack becomes the new top item.
- `peek()`: Returns a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.
- `push( item )`: Adds the given `item` to the top of the stack.

To illustrate a simple use of the Stack ADT, we apply it to the problem of reversing a list of integer values. The values will be extracted from the user until a negative value is entered, which flags the end of the collection. The values will then be printed in reverse order from how they were entered. We could use a simple list for this problem, but a stack is ideal since the values can be pushed onto the stack as they are entered and then popped one at a time to print them in reverse order. A solution for this problem follows.

```
PROMPT = "Enter an int value (<0 to end):"
myStack = Stack()
value = int(input( PROMPT ))
while value >= 0 :
  myStack.push( value )
  value = int(input( PROMPT ))

while not myStack.isEmpty() :
  value = myStack.pop()
  print( value )
```

Suppose the user enters the following values, one at a time:

```
 7  13  45  19  28  -1
```

When the outer `while` loop terminates after the negative value is extracted, the contents of the stack will be as illustrated in Figure 7.2. Notice the last value entered is at the top and the first is at the base. If we pop the values from the stack, they will be removed in the reverse order from which they were pushed onto the stack, producing a reverse ordering.

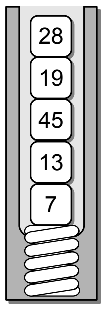

Figure 7.2: Resulting stack after executing the sample application.

7.2 Implementing the Stack

The Stack ADT can be implemented in several ways. The two most common approaches in Python include the use of a Python list and a linked list. The choice depends on the type of application involved.

7.2.1 Using a Python List

The Python list-based implementation of the Stack ADT is the easiest to implement. The first decision we have to make when using the list for the Stack ADT is which end of the list to use as the top and which as the base. For the most efficient ordering, we let the end of the list represent the top of the stack and the front represent the base. As the stack grows, items are appended to the end of the list and when items are popped, they are removed from the same end. Listing 7.1 on the next page provides the complete implementation of the Stack ADT using a Python list.

The `peek()` and `pop()` operations can only be used with a non-empty stack since you cannot remove or peek at something that is not there. To enforce this requirement, we first assert the stack is not empty before performing the given operation. The `peek()` method simply returns a reference to the last item in the list. To implement the `pop()` method, we call the `pop()` method of the list structure, which actually performs the same operation that we are trying to implement. That is, it saves a copy of the last item in the list, removes the item from the list, and then returns the saved copy. The `push()` method simply appends new items to the end of the list since that represents the top of our stack.

Listing 7.1 The `pyliststack.py` module.

```python
# Implementation of the Stack ADT using a Python list.
class Stack :
   # Creates an empty stack.
  def __init__( self ):
    self._theItems = list()

   # Returns True if the stack is empty or False otherwise.
  def isEmpty( self ):
    return len( self ) == 0

   # Returns the number of items in the stack.
  def __len__ ( self ):
    return len( self._theItems )

   # Returns the top item on the stack without removing it.
  def peek( self ):
    assert not self.isEmpty(), "Cannot peek at an empty stack"
    return self._theItems[-1]

   # Removes and returns the top item on the stack.
  def pop( self ):
    assert not self.isEmpty(), "Cannot pop from an empty stack"
    return self._theItems.pop()

   # Push an item onto the top of the stack.
  def push( self, item ):
    self._theItems.append( item )
```

The individual stack operations are easy to evaluate for the Python list-based implementation. `isEmpty()`, `__len__`, and `peek()` only require $O(1)$ time. The `pop()` and `push()` methods both require $O(n)$ time in the worst case since the underlying array used to implement the Python list may have to be reallocated to accommodate the addition or removal of the top stack item. When used in sequence, both operations have an amortized cost of $O(1)$.

7.2.2 Using a Linked List

The Python list-based implementation may not be the best choice for stacks with a large number of push and pop operations. Remember, each `append()` and `pop()` list operation may require a reallocation of the underlying array used to implement the list. A singly linked list can be used to implement the Stack ADT, alleviating the concern over array reallocations.

To use a linked list, we again must decide how to represent the stack structure. With the Python list implementation of the stack, it was most efficient to use the end of the list as the top of the stack. With a linked list, however, the front of the list provides the most efficient representation for the top of the stack. In Chapter 6, we saw how to easily prepend nodes to the linked list as well as remove the first node. The Stack ADT implemented using a linked list is provided in Listing 7.2.

Listing 7.2 The `lliststack.py` module.

```python
# Implementation of the Stack ADT using a singly linked list.
class Stack :
   # Creates an empty stack.
  def __init__( self ):
    self._top = None
    self._size = 0

   # Returns True if the stack is empty or False otherwise.
  def isEmpty( self ):
    return self._top is None

   # Returns the number of items in the stack.
  def __len__( self ):
    return self._size

   # Returns the top item on the stack without removing it.
  def peek( self ):
    assert not self.isEmpty(), "Cannot peek at an empty stack"
    return self._top.item

   # Removes and returns the top item on the stack.
  def pop( self ):
    assert not self.isEmpty(), "Cannot pop from an empty stack"
    node = self._top
    self.top = self._top.next
    self._size -= 1
    return node.item

   # Pushes an item onto the top of the stack.
  def push( self, item ) :
    self._top = _StackNode( item, self._top )
    self._size += 1

# The private storage class for creating stack nodes.
class _StackNode :
  def __init__( self, item, link ) :
    self.item = item
    self.next = link
```

The class constructor creates two instance variables for each `Stack`. The `_top` field is the head reference for maintaining the linked list while `_size` is an integer value for keeping track of the number of items on the stack. The latter has to be adjusted when items are pushed onto or popped off the stack. Figure 7.3 on the next page illustrates a sample `Stack` object for the stack from Figure 7.1(b).

The `_StackNode` class is used to create the linked list nodes. Note the inclusion of the `link` argument in the constructor, which is used to initialize the `_next` field of the new node. By including this argument, we can simplify the prepend operation of the `push()` method. The two steps required to prepend a node to a linked list are combined by passing the head reference `_top` as the second argument of the `_StackNode()` constructor and assigning a reference to the new node back to `_top`.

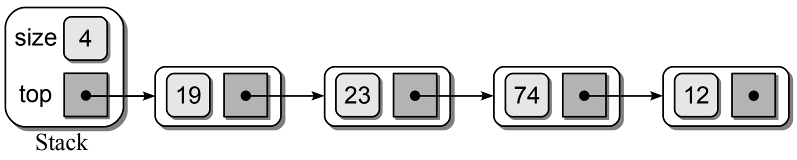

Figure 7.3: Sample object of the Stack ADT implemented as a linked list.

The `peek()` method simply returns a reference to the data item in the first node after verifying the stack is not empty. If the method were used on the stack represented by the linked list in Figure 7.3, a reference to 19 would be returned. The peek operation is only meant to examine the item on top of the stack. It should not be used to modify the top item as this would violate the definition of the Stack ADT.

The `pop()` method always removes the first node in the list. This operation is illustrated in Figure 7.4(a). This is easy to implement and does not require a search to find the node containing a specific item. The result of the linked list after popping the top item from the stack is illustrated in Figure 7.4(b).

The linked list implementation of the Stack ADT is more efficient than the Python-list based implementation. All of the operations are $O(1)$ in the worst case, the proof of which is left as an exercise.

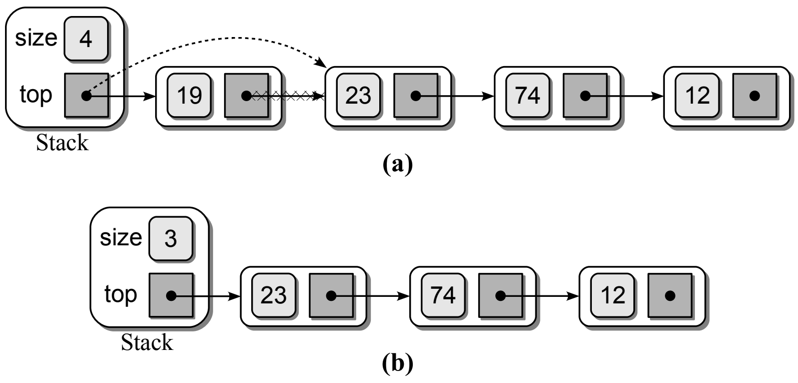

Figure 7.4: Popping an item from the stack: (a) the required link modifications, and (b) the result after popping the top item.

7.3 Stack Applications

The Stack ADT is required by a number of applications encountered in computer science. In this section, we examine several basic applications that traditionally are presented in a data structures course.

7.3.1 Balanced Delimiters

A number of applications use delimiters to group strings of text or simple data into subparts by marking the beginning and end of the group. Some common examples include mathematical expressions, programming languages, and the HTML markup language used by web browsers. There are typically strict rules as to how the delimiters can be used, which includes the requirement of the delimiters being paired and balanced. Parentheses can be used in mathematical expressions to group or override the order of precedence for various operations. To aide in reading complicated expressions, the writer may choose to use different types of symbol pairs, as illustrated here:

```
{A + (B * C) - (D / [E + F])}
```

The delimiters must be used in pairs of corresponding types: `{}`, `[]`, and `()`. They must also be positioned such that an opening delimiter within an outer pair must be closed within the same outer pair. For example, the following expression would be invalid since the pair of braces `[]` begin inside the pair of parentheses `()` but end outside.

```
(A + [B * C)] - {D / E}
```

Another common use of the three types of braces as delimiters is in the C++ programming language. Consider the following code segment, which implements a function to compute and return the sum of integer values contained in an array:

```
int sumList( int theList[], int size )
{
  int sum = 0;
  int i = 0;
  while( i < size ) {
    sum += theList[ i ];
    i += 1;
  }
  return sum;
}
```

As with the arithmetic expression, the delimiters must be paired and balanced. However, there are additional rules of the language that dictate the proper placement and use of the symbol pairs. We can design and implement an algorithm that scans an input text file containing C++ source code and determines if the delimiters are properly paired. The algorithm will need to remember not only the most recent opening delimiter but also all of the preceding ones in order to match them with closing delimiters. In addition, the opening delimiters will need to be remembered in reverse order with the most recent one available first. The Stack ADT is a perfect structure for implementing such an algorithm.

Consider the C++ code segment from earlier. As the file is scanned, we can push each opening delimiter onto the stack. When a closing delimiter is encountered, we pop the opening delimiter from the stack and compare it to the closing delimiter. For properly paired delimiters, the two should match. Thus, if the top of the stack contains a left bracket `[`, then the next closing delimiter should be a right bracket `]`. If the two delimiters match, we know they are properly paired and can continue processing the source code. But if they do not match, then we know the delimiters are not correct and we can stop processing the file. Table 7.1 shows the steps performed by our algorithm and the contents of the stack after each delimiter is encountered in our sample code segment.

Operation	Stack	Current scan line
push (	(	`int sumList(`
push	([	`int sumList( int values[`
pop & match]	(	`int sumList( int values[]`
pop & match)		`int sumList( int values[], int size )`
push {	{	{
	{	`int sum = 0;`
	{	`int i = 0;`
push (	{ (	`while(`
pop & match)	{	`while( i < size )`
push {	{ {	`while( i < size ) {`
push [	{ { [	`sum += theList[`
pop & match]	{ {	`sum += theList[ i ]`
	{ {	`i += 1;`
pop & match }	{	}
	{	`return sum;`
pop & match }	`empty`	}

Table 7.1: The sequence of steps scanning a valid set of delimiters: the operation performed (left column) and the contents of the stack (middle column) as each delimiter is encountered (right column) in the code.

So far, we have assumed the delimiters are balanced with an equal number of opening and closing delimiters occurring in the proper order. But what happens if the delimiters are not balanced and we encounter more opening or closing delimiters than the other? For example, suppose the programmer introduced a typographical error in the function header:

```
int sumList( int theList)], int size )
```

Our algorithm will find the first set of parentheses correct. But what happens when the closing bracket `]` is scanned? The result is illustrated in the top part of Table 7.2. You will notice the stack is empty since the left parenthesis was popped and matched with the preceding right parenthesis. Thus, unbalanced delimiters in which there are more closing delimiters than opening ones can be detected when trying to pop from the stack and we detect the stack is empty.

Operation	**Stack**	**Current point of scan**
push (	`(`	`int sumList(`
pop & match)	`empty`	`int sumList( int values)`
pop & match]	`error`	`int sumList( int values)]`

Scanning: `int sumList( int values)], int size )`

Operation	**Stack**	**Current point of scan**
push (	`(`	`int sumList(`
push (	`( (`	`int sumList( int (`
push [	`( ( [`	`int sumList( int (values[`
pop & match]	`( (`	`int sumList( int values[]`
pop & match)	`(`	`int sumList( int values[], int size )`

Scanning: `int sumList( int (values[], int size )`

Table 7.2: Sequence of steps scanning an invalid set of delimiters. The function header: (top) contains more closing delimiters than opening and (bottom) contains more closing delimiters than opening.

Delimiters can also be out of balance in the reverse case where there are more opening delimiters than closing ones. Consider another version of the function header, again containing a typographical error:

```
int sumList( int (theList[], int size )
```

The result of applying our algorithm to this code fragment is illustrated in the bottom chart in Table 7.2. If this were the complete code segment, you can see we would end up with the stack not being empty since there are opening delimiters yet to be paired with closing ones. Thus, in order to have a complete algorithm, we must check for both of these errors.

A Python implementation for the validation algorithm is provided in Listing 7.3. The function `isValidSource()` accepts a file object, which we assume was previously opened and contains C++ source code. The file is scanned one line at a time and each line is scanned one character at a time to determine if it contains properly paired and balanced delimiters.

A stack is used to store the opening delimiters and either implementation can be used since the implementation is independent of the definition. Here, we have chosen to use the linked list version. As the file is scanned, we need only examine

Listing 7.3 Function for validating a C++ source file.

```python
# Implementation of the algorithm for validating balanced brackets in
# a C++ source file.
from lliststack import Stack

def isValidSource( srcfile ):
  s = Stack()
  for line in srcfile :
    for token in line :
      if token in "{[(" :
        s.push( token )
      elif token in "}])" :
        if s.isEmpty() :
          return False
        else :
          left = s.pop()
          if (token == "}" and left != "{") or \
             (token == "]" and left != "[") or \
             (token == ")" and left != "(") :
            return False

  return s.isEmpty()
```

the characters that correspond to one of the three types of delimiter pairs. All other characters can be ignored. When an opening delimiter is encountered, we push it onto the stack. When a closing delimiter occurs, we first check to make sure the stack is not empty. If it is empty, then the delimiters are not properly paired and balanced and no further processing is needed. We terminate the function and return `False`. When the stack is not empty, the top item is popped and compared to the closing delimiter. The two delimiters do match corresponding opening and closing delimiters; we again terminate the function and return `False`. Finally, after the entire file is processed, the stack should be empty when the delimiters are properly paired and balanced. For the final test, we check to make sure the stack is empty and return either `True` or `False`, accordingly.

7.3.2 Evaluating Postfix Expressions

We work with mathematical expressions on a regular basis and they are rather easy for humans to evaluate. But the task is more difficult in a computer program when an expression is represented as a string. Given the expression

```
A * B + C / D
```

we know `A * B` will be performed first, followed by the division and concluding with addition. When evaluating this expression stored as a string and scanning one character at a time from left to right, how do we know the addition has to wait until after the division? Your first response is probably that we know the order of the precedence for the operators. But how do we represent that in our string scanning process? Suppose we are evaluating a string containing nine non-blank characters and have scanned the first three:

```
A + B
```

At this point, we have no way of knowing if the addition operation is to be performed on the two variables `A` and `B` or if we have to save this information for later. After moving to the the next character

```
A + B /
```

we encounter the division operator and know that the addition is not the first operation to be performed. Is the division the first operation to be performed? It does have higher precedence than the addition, but it may not be the first operation since parentheses can override the order of evaluation. We will have to scan more of the string to determine which operation is the first to be performed.

```
A + B / (C * D)
```

After determining the first operation to be performed, we must then decide how to return to those previously skipped. This can become a tedious process if we have to continuously scanned forward and backward through the string in order to properly evaluate the expression. To simplify the evaluation of a mathematical expression, we need an alternative representation for the expression. A representation in which the order the operators are performed is the order they are specified would allow for a single left-to-right scan of the expression string.

Three different notations can be used to represent a mathematical expression. The most common is the traditional algebraic or ***infix*** notation where the operator is specified between the operands `A+B`. The ***prefix*** notation places the operator immediately preceding the two operands `+AB`, whereas in ***postfix*** notation, the operator follows the two operands `AB+`.

At first glance, the different notations may seem to be nothing more than different operator placement. But the postfix and prefix notations have the advantages that neither uses parentheses to override the order of precedence and both create expressions in unique form. In other words, each expression is unique and produces a specific result unlike infix notation in which the same expression can be written in multiple ways.

Converting from Infix to Postfix

Infix expressions can be easily converted by hand to postfix notation. The expression `A + B - C` would be written as `AB+C-` in postfix form. The evaluation of this expression would involve first adding `A` and `B` and then subtracting `C` from that result. We will examine the evaluation of postfix expressions later; for now we focus on the conversion from infix to postfix.

Short expressions can be easily converted to postfix form, even those using parentheses. Consider the expression `A*(B+C)`, which would be written in postfix as `ABC+*`. Longer expressions, such as the example from earlier, `A*B+C/D`, are a bit more involved. To help in this conversion we can use a simple algorithm:

1. Place parentheses around every group of operators in the correct order of evaluation. There should be one set of parentheses for every operator in the infix expression.

   ```
   ((A * B) + (C / D))
   ```

2. For each set of parentheses, move the operator from the middle to the end preceding the corresponding closing parenthesis.

   ```
   ((A B *) (C D /) +)
   ```

3. Remove all of the parentheses, resulting in the equivalent postfix expression.

   ```
   A B * C D / +
   ```

Compare this result to a modified version of the expression in which parentheses are used to place the addition as the first operation:

```
A * (B + C) / D
```

Using the simple algorithm, we parenthesize the expression:

```
((A * (B + C)) / D)
```

and move the operators to the end of each parentheses pair:

```
((A (B C +) *) D /)
```

Finally, removing the parentheses yields the postfix expression:

```
A B C + * D /
```

A similar algorithm can be used for converting from infix to prefix notation. The difference is the operators are moved to the front of each group.

Postfix Evaluation Algorithm

Parentheses are used with infix expressions to change the order of evaluation. But in postfix notation, the order cannot be altered and thus there is no need for parentheses. Given the unique form or single order of evaluation, postfix notation is a good choice when evaluating a mathematical expression represented as a string. Of course the expression would have to either be given in postfix notation or first converted from infix to postfix. The latter can be easily done with an appropriate algorithm, but we limit our discussion to the evaluation of existing postfix expressions.

Evaluating a postfix expression requires the use of a stack to store the operands or variables at the beginning of the expression until they are needed. Assume we are given a valid postfix expression stored in a string consisting of operators and single-letter variables. We can evaluate the expression by scanning the string, one character or token at a time. For each token, we perform the following steps:

1. If the current item is an operand, push its value onto the stack.
2. If the current item is an operator:
   (a) Pop the top two operands off the stack.
   (b) Perform the operation. (Note the top value is the right operand while the next to the top value is the left operand.)
   (c) Push the result of this operation back onto the stack.

The final result of the expression will be the last value on the stack. To illustrate the use of this algorithm, let's evaluate the postfix expression `A B C + * D /` from our earlier example. Assume the existence of an empty stack and the following variable assignments have been made:

```
A = 8          C = 3
B = 2          D = 4
```

The complete sequence of algorithm steps and the contents of the stack after each operation are illustrated in Table 7.3.

Token	Alg. Step	Stack	Description
ABC+*D/	1	8	push value of A
A**B**C+*D/	1	8 2	push value of B
AB**C**+*D/	1	8 2 3	push value of C
ABC**+***D/	2(a)	8	pop top two values: y = 3, x = 2
	2(b)	8	compute z = x + y or z = 2 + 3
	2(c)	8 5	push result (5) of the addition
ABC+*D/	2(a)		pop top two values: y = 5, x = 8
	2(b)		compute z = x * y or z = 8 * 5
	2(c)		push result (40) of the multiplication
ABC+***D**/	1	40 4	push value of D
ABC+*D**/**	2(a)		pop top two values: y = 4, x = 40
	2(b)		compute z = x / y or z = 40 / 4
	2(c)	10	push result (10) of division

Table 7.3: The stack contents and sequence of algorithm steps required to evaluate the valid postfix expression `A B C + * D`.

The postfix evaluation algorithm assumes a valid expression. But what happens if the expression is invalid? Consider the following invalid expression in which there are more operands than available operators:

```
A B * C D +
```

After applying the algorithm to this expression, there are two values remaining on the stack as illustrated in Table 7.4. What happens if there are too many operators for the given number of operands? Consider such an invalid expression:

```
A B * + C /
```

In this case, there are too few operands on the stack when we encounter the addition operator, as illustrated in Table 7.5. If we attempt to perform two pops from the stack, an assertion error will be thrown since the stack will be empty on the second pop. We can modify the algorithm to detect both types of errors. In step 2(a), we must first verify the stack is not empty before popping an item. If the stack is empty, we can stop the evaluation and flag an error. The second modification occurs after the evaluation of the entire expression. We can pop the result from the stack and then verify the stack is empty. If the stack is not empty, the expression was invalid and we must flag an error.

Token	Alg. Step	Stack	Description
AB*CD+	1	8	push value of A
A**B***CD+	1	8 2	push value of B
AB*CD+	2(a)		pop top two values: y = 2, x = 8
	2(b)		compute z = x * y or z = 8 * 2
	2(c)	16	push result (16) of the multiplication
AB***C**D+	1	16 3	push value of C
AB*C**D**+	1	16 3 4	push value of D
AB*CD**+**	2(a)	16	pop top two values: y = 4, x = 3
	2(b)	16	compute z = x + y or z = 3 + 4
	2(c)	16 7	push result (7) of the addition
Error	***xxxxxx***	***xxxxxx***	***Too many values left on stack.***

Table 7.4: The sequence of algorithm steps when evaluating the invalid postfix expression `A B * C D +`.

7.4 Application: Solving a Maze

A classic example of an application that requires the use of a stack is the problem of finding a path through a maze. When viewing a maze drawn on paper such

Token	Alg. Step	Stack	Description
<u>**A**</u>B*+C/	1	`8`	push value of A
A<u>**B**</u>*+C/	1	`8 2`	push value of B
AB<u>*****</u>+C/	2(a)		pop top two values: y = 2, x = 8
	2(b)		compute z = x * y or z = 8 * 2
	2(c)	`16`	push result (16) of the multiplication
AB*<u>+</u>C/	2(a)		pop top two values: y = 16, x = ?
Error	***xxxxxx***	***xxxxxx***	***Only one value on stack, two needed.***

Table 7.5: The sequence of algorithm steps taken when evaluating the invalid postfix expression `A B * + C /`.

as the one illustrated in Figure 7.5, we can quickly find a path from the starting point to the exit. This usually involves scanning the entire maze and mentally eliminating dead ends. But consider a human size maze in which you are inside the maze and only have a “rat’s-eye” view. You cannot see over the walls and must travel within the maze remembering where you have been and where you need to go. In this situation, it’s not as easy to find the exit as compared to viewing the maze on paper.

An algorithm that can be used to find a path through a maze is likely to employ a technique similar to what you would use if you were inside the maze. In this section, we explore the backtracking technique to solving a maze and design an algorithm to implement our technique.

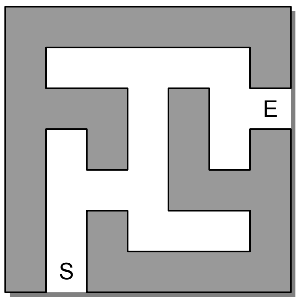

Figure 7.5: A sample maze with the indicated starting (`S`) and exit (`E`) positions.

7.4.1 Backtracking

The most basic problem-solving technique in computer science is the ***brute-force*** method. It involves searching for a solution to a given problem by systematically trying all possible candidates until either a solution is found or it can be determined there is no solution. Brute-force is time-consuming and is generally chosen as a last resort. But some problems can only be solved using this technique.

If applied to the maze problem, the brute-force method would require we start at the beginning and follow a path until we either find the exit or encounter a blocked passage. If we hit a wall instead of the exit, we would start over from the beginning and try a different path. But this would be time consuming since we would likely follow part of the same path from the beginning to some point before we encountered the blocked passage. Instead of going all the way back to the beginning, we could back up along the path we originally took until we find a passage going in a different direction. We could then follow the new passage in hopes of finding the exit. If we again encounter a blocked passage before the exit, we can back up one or more steps and try a different passage.

This process of eliminating possible contenders from the solution and partially backing up to try others is known as ***backtracking*** and is a refinement of the basic brute-force method. There is a broad class of algorithms that employ this technique and are known as ***backtracking algorithms***. All of these algorithms attempt to find a solution to a problem by extending a partial solution one step at a time. If a "dead end" is encountered during this process, the algorithm backtracks one or more steps in an attempt to try other possibilities without having to start over from the beginning.

7.4.2 Designing a Solution

The solution to the maze problem is a classic example of backtracking. In this section, we explore the technique and design a solution to the maze problem.

Problem Details

Given a maze with indicated starting and exit positions, the objectives are (1) determine if there is a path from the starting position to the exit, and (2) specify the path with no circles or loopbacks. In designing an algorithm to solve a maze, it will be easier if we think of the maze as a collection of equal-sized cells laid out in rows and columns, as illustrated in Figure 7.6. The cells will either be filled representing walls of the maze or empty to represent open spaces. In addition, one cell will be indicated as the starting position and another as the exit.

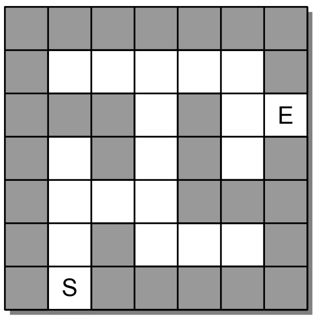

Figure 7.6: Sample maze from Figure 7.5 divided into equal-sized cells.

To further aide in the algorithm development, we place certain restrictions on movement within the maze. First, we can only move one cell at a time and only to open positions, those not blocked by a wall or previously used along the current path. The latter prevents us from reusing a cell as part of the solution since we want to find a path from the start to the exit without ever having to go in circles. Finally, we limit movement between opens cells to the horizontal and vertical directions—up, down, left, and right—as illustrated in Figure 7.7.

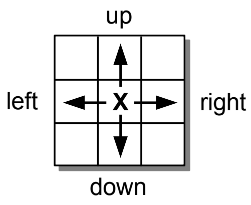

Figure 7.7: The legal moves allowed from a given cell in the maze.

During our search for the exit, we need to remember which cells have been visited. Some will be part of the final path to the exit while others will have led us to dead ends. At the end, we need to know which cells form the path from the start to the exit. But during the search for the exit, we also need to avoid cells that previously led to a dead end. To assist in remembering the cells, we can place a token in each cell visited and distinguish between the two. In our example, we will use an x to represent cells along the path and an o to represent those that led to a dead end.

Describing the Algorithm

We begin at the starting position and attempt to move from cell to cell until we find the exit. As we move between cells, we must consider what actions are available at each cell. Consider the smaller maze in Figure 7.8 in which the rows and columns have been numbered to aide in identifying the cells.

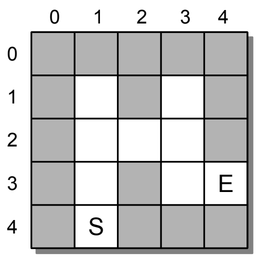

Figure 7.8: A small maze with the rows and columns labeled for easy reference.

Finding the Exit. From the starting position $(4,1)$, we can examine our surroundings or more specifically the four neighboring cells and determine if we can move from this position. We want to use a systematic or well-ordered approach in finding the path. Thus, we always examine the neighboring cells in the same order: up, down, left, and right. In the sample maze, we find the cell above, $(3,1)$, is open and prepare to move up one step. Before moving from the current position, however, we need to lay down a token to indicate the current cell is part of our path. As indicated earlier, we place a lowercase `x` in the cell to indicate it comprises part of the path. The complete set of steps taken to solve our sample maze are illustrated in Figure 7.9.

After placing the token, we move to the open cell above the starting position. The current position in the maze is marked in the illustration using an uppercase `X`. We repeat the process and find the cell above our current position is open. A token is laid in the current cell and we move up one position to cell $(2,1)$. From our vantage point above the matrix, we easily see the solution to the problem, which requires that we move to the right. But from the point of view of a mouse searching for cheese, that specific move would be unknown.

Using our systematic approach, we examine the cell above our current position. We find it open, and move up one position to cell $(1,1)$. From this position, we

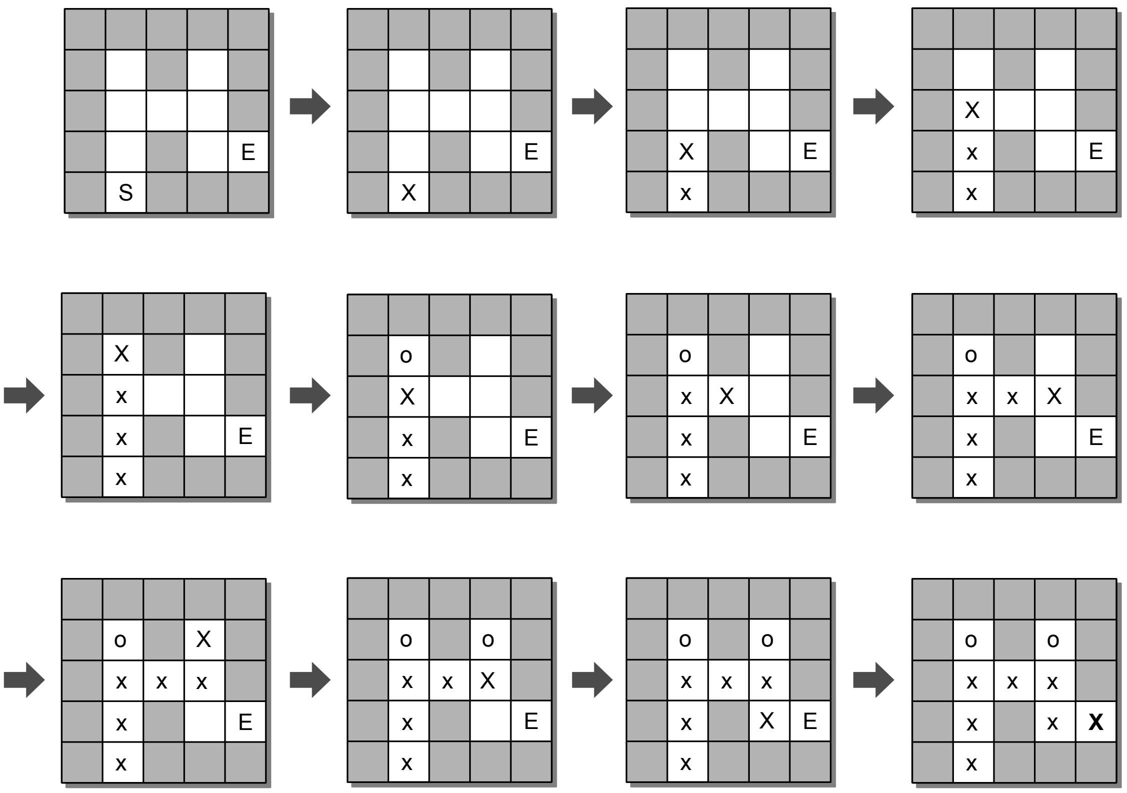

Figure 7.9: The sequence of steps taken through a sample maze. `S` is the starting position and `E` is the exit. The path is marked using the character `x` while the current cell is marked with `X`. Cells we visited but from which we had to backtrack are marked with `o`.

soon discover there are no legal moves since we are blocked by a wall on three sides and a cell comprising part of our path. Since we can go no further from this cell, we have no choice but to go back to our previous position in cell $(2, 1)$. When hitting a dead end, we don't simply turn around and go back over a cell previously visited as if it were part of the path since this would cause a circle. Instead, we mark the cell with a different token indicating a dead end and move back to the previous position. In our example, we use a lowercase o to represent a cell leading to a dead end.

After moving back to cell $(2, 1)$, we examine the other directions and soon find the cell to the right is open and move in that direction, placing us in cell $(2, 2)$. From this position, we find the only legal move is to the right and thus move in that direction, placing us in cell $(2, 3)$. Next, we move up one step since the cell above is open. But this move will result in a dead end, requiring us to once again back up to the previous position. After backing up to our previous position at cell $(2, 3)$, we find the cell below is open and move to position $(3, 3)$. Repeating the process we soon find the exit at position $(3, 4)$ and a path from the starting cell to the exit.

No Path to the Exit. The exit in this example was accessible. But what happens if there is no path between the start and exit cells? Consider a modified version of our sample maze in Figure 7.10 where a wall has been placed in cell $(3, 3)$ closing the path to the exit.

When reaching position $(2, 3)$, as described earlier, we will discover this is a dead end and have to back up. But there are no other legal moves from cell $(2, 2)$ with the positions above and below blocked by a wall, the position to the right leading to a dead end and the position to the left currently part of our path. From position $(2, 2)$, we have to back up and try another direction. Ultimately, we will have to backtrack all the way to the start, having found no legal move from that position.

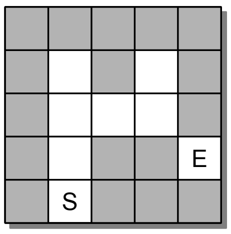

Figure 7.10: A modified version of the sample maze with the exit blocked.

7.4.3 The Maze ADT

Given the description of the maze problem and the backtracking algorithm for finding a path through the maze, we now define the Maze ADT that can be used to construct and solve a maze.

Define **Maze ADT**

A *maze* is a two-dimensional structure divided into rows and columns of equal-sized cells. The individual cells can be filled representing a wall or empty representing an open space. One cell is marked as the starting position and another as the exit.

- `Maze( numRows, numCols )`: Creates a new maze with all of the cells initialized as open cells. The starting and exit positions are undefined.
- `numRows()`: Returns the number of rows in the maze.
- `numCols()`: Returns the number of columns in the maze.
- `setWall( row, col )`: Fills the indicated cell (`row`, `col`) with a wall. The cell indices must be within the valid range of rows and columns.
- `setStart( row, col )`: Sets the indicated cell (`row`, `col`) as the starting position. The cell indices must be within the valid range.
- `setExit( row, col )`: Sets the indicated cell (`row`, `col`) as the exit position. The cell indices must be within the valid range.
- `findPath()`: Attempts to the solve the maze by finding a path from the starting position to the exit. If a solution is found, the path is marked with tokens (`x`) and `True` is returned. For a maze with no solution, `False` is returned and the maze is left in its original state. The maze must contain both the starting and exit position. Cells on the perimeter of the maze can be open and it can be assumed there is an invisible wall surrounding the entire maze.
- `reset()`: Resets the maze to its original state by removing any tokens placed during the find path operation.
- `draw()`: Prints the maze in a readable format using characters to represent the walls and path through the maze, if a path has been found. Both the starting and exit positions are also indicated, if previously set.

Our ADT definition is not meant for a general purpose maze, but instead one that can be used to build a maze and then solve and print the result. A more general purpose ADT would most likely return the solution path as a list of tuples instead of simply marking the cells within the maze as is the case in our definition.

Example Use

We can use this definition of the ADT to construct a program for building and solving a maze as shown in Listing 7.4. The main routine is rather simple since we need only build the maze, determine if a path exists and print the maze if a path does exist.

Listing 7.4 The `solvemaze.py` program.

```python
# Program for building and solving a maze.
from maze import Maze

# The main routine.
def main():
  maze = buildMaze( "mazefile.txt" )
  if maze.findPath() :
    print( "Path found...." )
    maze.draw()
  else :
    print( "Path not found...." )

# Builds a maze based on a text format in the given file.
def buildMaze( filename ):
  infile = open( filename, "r" )

    # Read the size of the maze.
  nrows, ncols = readValuePair( infile )
  maze = Maze( nrows, ncols )

    # Read the starting and exit positions.
  row, col = readValuePair( infile )
  maze.setStart( row, col )
  row, col = readValuePair( infile )
  maze.setExit( row, col )

    # Read the maze itself.
  for row in range( nrows ) :
    line = infile.readline()
    for col in range( len(line) ) :
      if line[col] == "*" :
        maze.setWall( row, col )

   # Close the maze file and return the newly constructed maze.
  infile.close()
  return maze

# Extracts an integer value pair from the given input file.
def readValuePair( infile ):
  line = infile.readline()
  valA, valB = line.split()
  return int(valA), int(valB)

# Call the main routine to execute the program.
main()
```

Maze Text File Format

Before searching for a path through the maze, we must first build a maze. The maze can be constructed directly within the program by calls to `setWall()` with literal indices or we can read a maze specification from a text file. Suppose a maze is represented in a text file using the following format:

```
5 5
4 1
3 4
*****
*.*.*
*...*
*.*..
*.***
```

The first line contains the size of the maze given as the number of rows and columns. The two subsequent lines indicate the row and column indices of the starting and exit positions. The remaining lines of text represent the maze itself, with walls represented using a hash symbol and open cells represented as blank spaces. The maze is constructed from the text file using the `buildMaze()` function as shown in lines 14–42 of Listing 7.4.

7.4.4 Implementation

The implementation of our Maze ADT will require the selection of a data structure to represent the maze and to implement the backtracking operation used to find a path. The most obvious choice of data structure for storing the maze is a 2-D array. The individual elements of the array will represent the cells of the maze. Strings containing a single character can be used to represent the walls and tokens while the open cells are easily represented as null pointers. The array representation of our sample maze is illustrated in Figure 7.11.

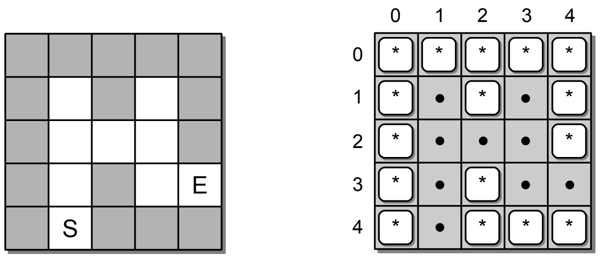

Figure 7.11: The abstract view of a maze physically represented using a 2-D array. Walls are indicated with an asterisk (∗) character, while open cells contain a null reference. The start and exit cells will be identified by cell position stored in separate data fields.

Class Definition

A partial implementation of the Maze ADT is provided in Listing 7.5. Three constant class variables are defined and initialized to store the various symbols used to mark cells within the maze. Remember, class variables are not data fields of the individual objects, but are instead variables of the class, which can be accessed by the individual methods. By using the named constants, the values used to represent the maze wall and tokens could easily be changed if we were so inclined.

Listing 7.5 The `maze.py` module.

```python
# Implements the Maze ADT using a 2-D array.
from array import Array2D
from lliststack import Stack

class Maze :
   # Define constants to represent contents of the maze cells.
  MAZE_WALL = "*"
  PATH_TOKEN = "x"
  TRIED_TOKEN = "o"

   # Creates a maze object with all cells marked as open.
  def __init__( self, numRows, numCols ):
    self._mazeCells = Array2D( numRows, numCols )
    self._startCell = None
    self._exitCell = None

   # Returns the number of rows in the maze.
  def numRows( self ):
    return self._mazeCells.numRows()

   # Returns the number of columns in the maze.
  def numCols( self ):
    return self._mazeCells.numCols()

   # Fills the indicated cell with a "wall" marker.
  def setWall( self, row, col ):
    assert row >= 0 and row < self.numRows() and \
           col >= 0 and col < self.numCols(), "Cell index out of range."
    self._mazeCells.set( row, col, self.MAZE_WALL )

   # Sets the starting cell position.
  def setStart( self, row, col ):
    assert row >= 0 and row < self.numRows() and \
           col >= 0 and col < self.numCols(), "Cell index out of range."
    self._startCell = _CellPosition( row, col )

   # Sets the exit cell position.
  def setExit( self, row, col ):
    assert row >= 0 and row < self.numRows() and \
           col >= 0 and col < self.numCols(), \
           "Cell index out of range."
    self._exitCell = _CellPosition( row, col )

   # Attempts to solve the maze by finding a path from the starting cell
   # to the exit. Returns True if a path is found and False otherwise.
  def findPath( self ):
    ......

```

(Listing Continued)

Listing 7.5 Continued . . .

```
49     # Resets the maze by removing all "path" and "tried" tokens.
50    def reset( self ):
51      ......
52
53     # Prints a text-based representation of the maze.
54    def draw( self ):
55      ......
56
57     # Returns True if the given cell position is a valid move.
58     def _validMove( self, row, col ):
59      return row >= 0 and row < self.numRows() \
60         and col >= 0 and col < self.numCols() \
61         and self._mazeCells[row, col] is None
62
63     # Helper method to determine if the exit was found.
64    def _exitFound( self, row, col ):
65      return row == self._exitCell.row and \
66             col == self._exitCell.col
67
68     # Drops a "tried" token at the given cell.
69    def _markTried( self, row, col ):
70      self._mazeCells.set( row, col, self.TRIED_TOKEN )
71
72     # Drops a "path" token at the given cell.
73    def _markPath( self, row, col ):
74      self._mazeCells.set( row, col, self.PATH_TOKEN )
75
76
77  # Private storage class for holding a cell position.
78  class _CellPosition( object ):
79    def __init__( self, row, col ):
80      self.row = row
81      self.col = col
```

The class constructor, shown in lines 12–15 of Listing 7.5, creates a `MultiArray` object and two fields to store the starting and exit cells. A sample `Maze` object for our maze in Figure 7.11 is illustrated in Figure 7.12. The array is created using the arguments to the constructor. The cells of the maze are automatically initialized to `None`, as specified in the previous section, since this is the default value used when creating a `MultiArray` object. The `_startCell` and `_exitCell` fields are set to `None` since they are initially undefined. Later, specific positions will have to be stored when they are defined by the respective methods. Since a cell is indicated by its position within the array, we can define the `_CellPosition` class to store a specific cell.

Maze Components

Components of the maze are specified using the various set methods, which are shown in lines 26–42 of Listing 7.5. Since the user specifies specific maze elements with each of these methods, they must first validate the cell position to ensure the

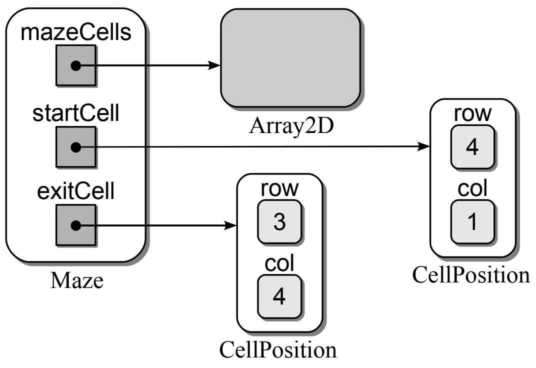

Figure 7.12: A sample Maze ADT object.

indices are within the valid range. The two methods that set the starting and exit positions simply create and store `_CellPosition` objects while the creation of a wall fills the indicated cell using one of the named constants defined earlier.

Helper Methods

During the actual process of finding a solution in the `findPath()` method, we will need to perform several routine operations that access the underlying `MultiArray` object. To aide in this task, we define several helper methods, as shown in lines 58–74. First, we will need to drop or place tokens as we move through the maze. The `_markTried()` and `_markPath()` methods can be used for this task. Note we do not need a "pickup token" method since the task of picking up a path token is immediately followed by dropping a tried token.

The `_exitFound()` method is used to determine if the exit is found based on the contents of the `_exitCell` object and the current position supplied as arguments. Finally, the `_validMove()` helper method is used to determine if we can move to a given cell. A move is valid if the destination cell is open and its indices are not outside the border of the maze. Note an assertion is not used here since we do not want to flag an error. Instead, the backtracking solution will have to try other directions when encountering an invalid move.

Finding the Path

The `findPath()` method implements the actual path finding algorithm described earlier, which searches through the maze for a path from the starting position to the

TIP

Helper Methods. Helper methods are commonly used in implementing classes to aide in subdividing larger problems and for reducing code repetition by defining a method that can be called from within the different methods. But helper methods can also be used to make your code more readable even if they only contain a single line. This use of helper methods is illustrated by the helper methods defined as part of the `Maze` class.

exit. As we move through the maze, we must remember the path we took in order to backtrack when reaching a dead end. A stack provides the ideal structure we need to remember our path. As we move forward in the maze, we can push our current position onto the stack using a `_CellPosition` object before moving forward to the next cell. When reaching a dead end, we can backtrack by popping the previous position from the stack and backing up to that position. The implementation of this method is left as an exercise along with the `reset()` and `draw()` methods.

Exercises

7.1 Hand execute the following code segment and show the contents of the resulting stack.

```
values = Stack()
for i in range( 16 ) :
  if i % 3 == 0 :
    values.push( i )
```

7.2 Hand execute the following code segment and show the contents of the resulting stack.

```
values = Stack()
for i in range( 16 ) :
  if i % 3 == 0 :
    values.push( i )
  elif i % 4 == 0 :
    values.pop()
```

7.3 Translate each of the following infix expressions into postfix.

(a) `(A * B) / C`

(b) `A - (B * C) + D / E`

(c) `(X - Y) + (W * Z) / V`

(d) `V * W * X + Y - Z`

(e) `A / B * C - D + E`

7.4 Translate each of the infix expressions in Exercise 7.3 to prefix notation.

7.5 Translate each of the following postfix expressions into infix.

(a) `A B C - D * +`

(b) `A B + C D - / E +`

(c) `A B C D E * + / +`

(d) `X Y Z + A B - * -`

(e) `A B + C - D E * +`

7.6 Consider our implementation of the Stack ADT using the Python list, and suppose we had used the front of the list as the top of the stack and the end of the list as the base. What impact, if any, would this have on the run time of the various stack operations?

7.7 Show that all of the Stack ADT operations have a constant time in the worst case when implemented as a linked list.

7.8 Would it buy us anything to use a tail reference with the linked list structure used to implement the Stack ADT? Explain your answer.

7.9 Evaluate the run time of the `isValidSource()` function where n is the number of characters in the C++ source file.

Programming Projects

7.1 Write and test a program that extracts postfix expressions from the user, evaluates the expression, and prints the results. You may require that the user enter numeric values along with the operators and that each component of the expression be separated with white space.

7.2 The `isValidSource()` function can be used to evaluate a C++ source file, but it is incomplete. Brackets encountered inside comments and literal strings would not be paired with those found elsewhere in the program.

(a) C++ comments can be specified using `//`, which starts a comment that runs to the end of the current line, and the token pair `/* */`, which encloses a comment that can span multiple lines. Extend the function to skip over brackets found inside C++ comments.

(b) C++ literal strings are denoted by enclosing characters within double quotes (`"string"`) and literal characters are denoted by enclosing a character within single quotes (`'x'`). Extend the function to skip over brackets found inside C++ literal strings and characters.

7.3 Design and implement a function that evaluates a prefix expression stored as a text string.

7.4 Implement the `findPath()`, `reset()`, and `draw()` methods for the `Maze` class.

7.5 Implement a complete maze solving application using the components introduced earlier in the chapter. Modify the `solve()` method to return a vector containing tuples representing the path through the maze.

7.6 We can design and build a postfix calculator that can be used to perform simple arithmetic operations. The calculator consists of a single storage component that consists of an operand stack. The operations performed by the stack always use the top two values of the stack and store the result back on the top of the stack. Implement the operations of the Postfix Calculator ADT as defined here:.

- `PostfixCalculator()`: Creates a new postfix calculator with an empty operand stack.
- `value( x )`: Pushes the given operand `x` onto the top of the stack.
- `result()`: Returns an alias to the value currently on top of the stack. If the stack is empty, `None` is returned.
- `clear()`: Clears the entire contents of the stack.
- `clearLast()`: Removes the top entry from the stack and discards it.
- `compute( op )`: Removes the top two values from the stack and applies the given operation on those values. The first value removed from the stack is the righthand side operand and the second is the lefthand side operand. The result of the operation is pushed back onto the stack. The operation is specified as a string containing one of the operators `+ - * / **`.

7.7 Extend the Postfix Calculator ADT as follows:

(a) To perform several unary operations commonly found on scientific calculators: absolute value, square root, sine, cosine, and tangent. The operations should be specified to the `compute()` method using the following acronyms: `abs, sqrt, sin, cos, tan`.

(b) To use a second stack on which values can be saved. Add the following two operations to the ADT:

- `store()`: Removes the top value from the operand stack and pushes it onto the save stack.
- `recall()`: Removes the top value from the save stack and pushes it onto the operand stack.

7.8 Design and implement a complete program that uses the Postfix Calculator ADT to perform various operations extracted from the user. The user enters text-based commands, one per line, that should be performed by the calculator. For example, to compute `12 * 15`, the user would enter the following sequence of commands:

```
ENTER 12
ENTER 15
MUL
RESULT
```

which would result in 180 being displayed. Your program should respond to the following set of commands: `ENTER, CLR, CLRLAST, RESULT, ADD, SUB, MUL, DIV, POW`.