Lecture 6: ADT Stack

Read: Chpt.6, Carrano

Linear ADT: Based on linear data structures consisting of a collection of objects in which:

(1) There is a specific *first* object in the collection.
(2) Each object in the collection has a well-defined *next*, and *previous*, item in the collection.
(3) Beginning at the first object, each object in the collection will eventually be encountered by successively visiting next items.

Some Examples of Linear ADT:

Array: Fixed length; contiguous in storage.

Vector: Arbitrary length; otherwise pretty much the same as array

String: Essentially a special type of array, although the details vary by language.

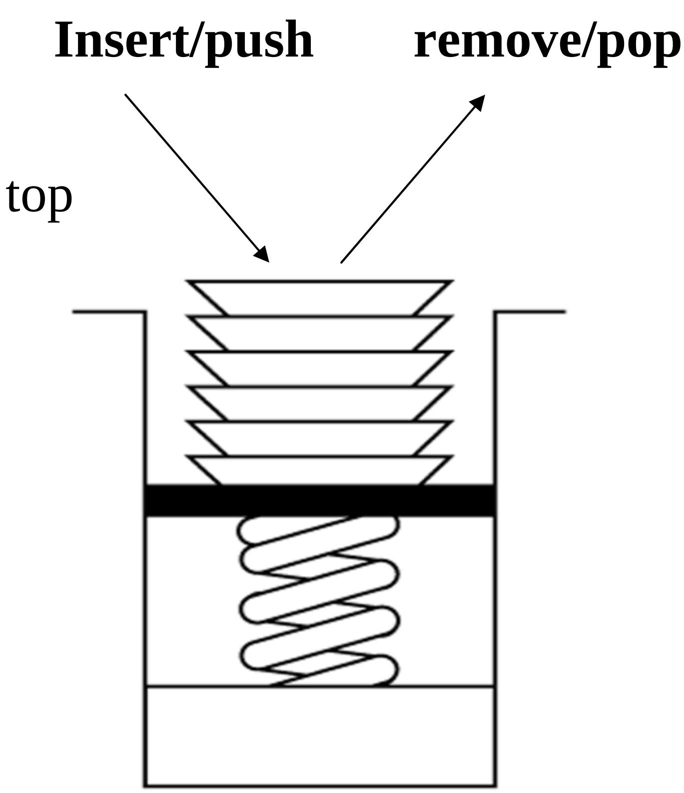
List: *Sorted* and *unsorted*. This is representative of the most general sort of linear ADT. They are not constrained to be contiguous in storage.

Other Important Linear ADT: Stack and Queue.

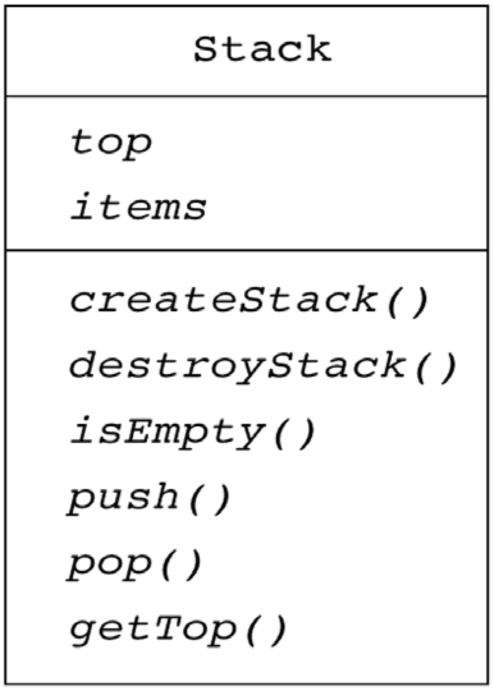
Both are restricted forms of lists in that insertion and deletion are restricted to take place at one "end" of the structure only. (For list, insertion and deletion can occur anywhere!)

Stacks exhibit **LIFO** (**L**ast-**I**n-**F**irst-**O**ut) behavior: Insertions and deletions must take place ***at the same end*****.**
Queues exhibit **FIFO** (**F**irst-**I**n-**F**irst-**O**ut) behavior: Insertions and deletions must take place at ***opposite ends***.

ADT: Stack



UML Diagram for class Stack:



+createStack( )

+destroyStack( )

+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

Common applications of stack:

To retrieve data in the opposite order in which it was saved.

Examples:

(1) Certain type of parsing (e.g., parenthesis matching)

(2) Algebraic expression handling (e.g., infix-postfix conversion; postfix evaluation)

(3) Implementation of function call/return, including recursive call/return mechanisms in programming languages.

Example: Suppose you have been asked to implement a method printReverse in class List, but you have forgotten how recursion works.

Q: How could you do it with a Stack?

```
void List::printReverse(ostream& os)
{
  Stack stk;
  ListNode* p = head;
  while (p != NULL)      // push listItems onto stack
  {
    stk.push(p->item);
    p = p->next;
  }
  int anItem;
  while (!stk.isEmpty()) // pop listItems from stack
  {
    stk.pop(anItem);
    os << anItem;
  }
  os << endl;
}
```

Example: Balancing/Matching parentheses in an algebraic expression.

a * (b + 3 *{d - [c + d] + f} -h) +k   OK

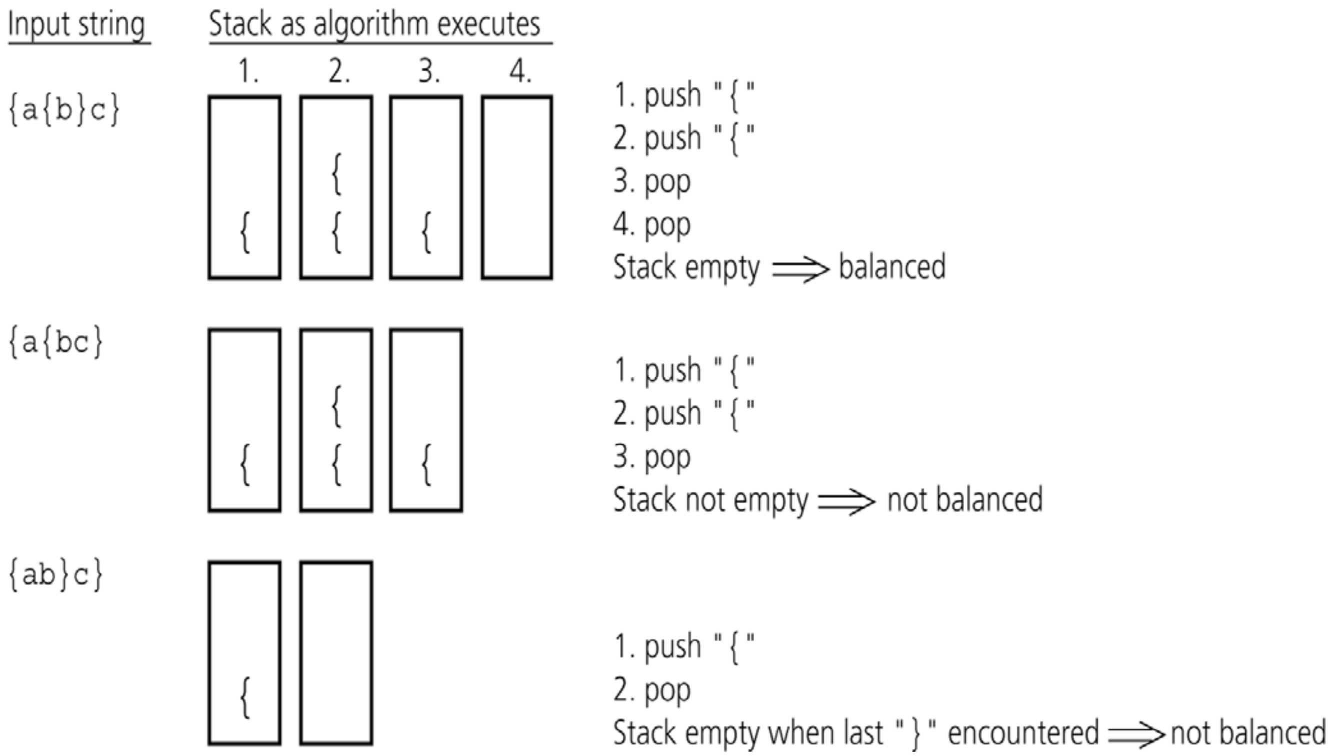
a * (b + 3 *(d - [c + d] + f} -h) +k   Not OK

Q: How do we develop an algorithm to read in an algebraic expression symbol-by-symbol and report whether the parentheses match correctly?

Algorithm:

```
create an empty stack S;
while not end of expression
   read in a symbol;
   if the symbol is an open (left) grouping symbol
      then push it onto the stack S
      else if the symbol is a close (right) grouping symbol
               then if S is empty
                         then return false  //too many right paren
                         else  pop the stack S;
                                 if the popped opening
                                 symbol does not match the
                                 closing one
                                    then    // mismatched parens
                                      return false
                                 endif
                        endif;
            endif;
   endif;
endwhile;
if S is empty        // all aprens matched & balanced
   then return true
   else return false   // too many left parens
endif;
// end algorithm;
```

Example:



HW: Implement this parentheses matching algorithm.

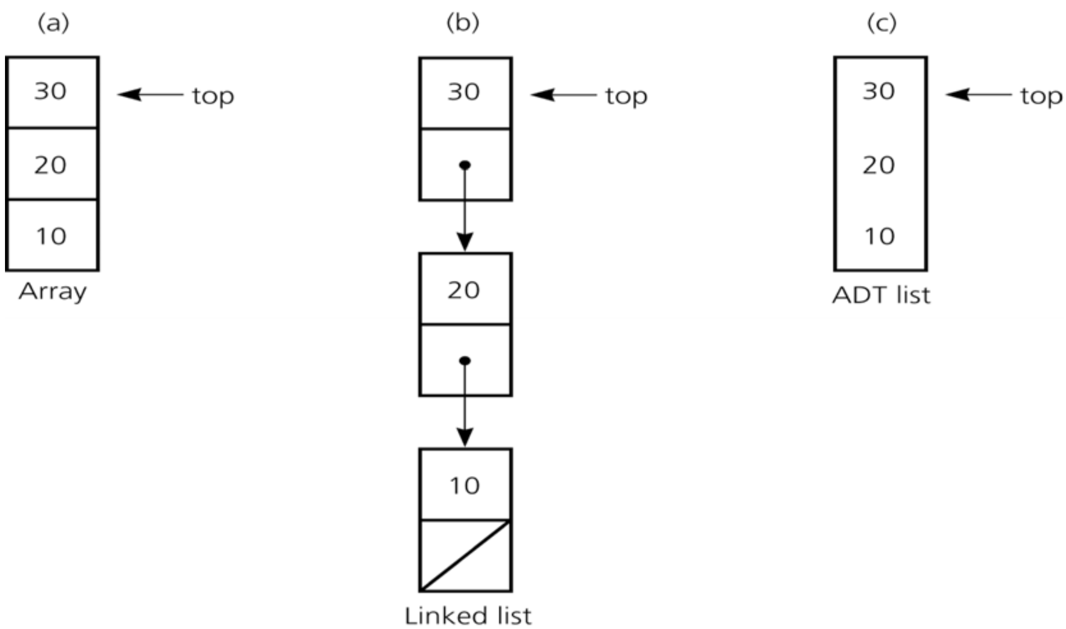
Implementations of Stacks:

(1) Array: Easiest.

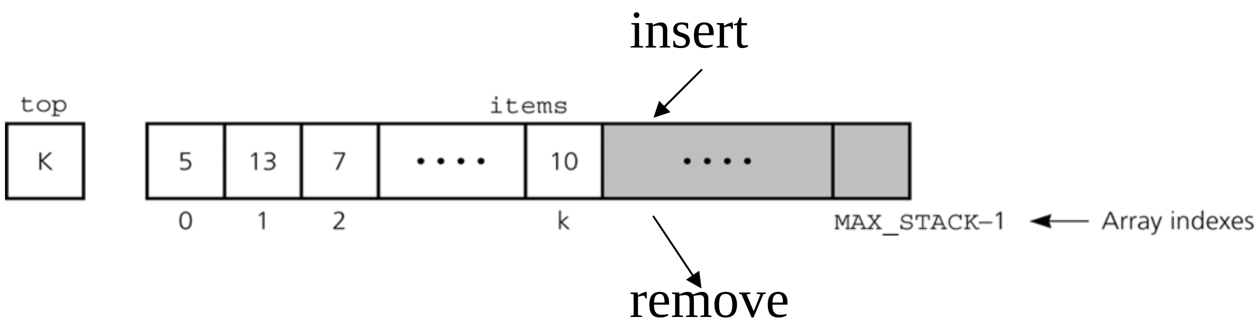
(2) Pointer-based linked list data structure: Very adaptable in terms of widely varying expected stack sizes.

(3) ADT List class based: Maximizes code re-use.

Example:



An Array-Based Implementation of Stack:




```
// Header file StackA.h for the ADT stack on P.293
// Array-based implementation.

#include "StackException.h"
const int MAX_STACK = maximum-size-of-stack;
typedef desired-type-of-stack-item StackItemType;

class Stack
{
public:

// constructors and destructor:
   Stack();       // default constructor
                  // copy constructor and destructor are
                  // supplied by the compiler
```

```
// stack operations:
   bool isEmpty() const;

   void push(StackItemType newItem)
                         throw(StackException);

   void pop() throw(StackException);

   void pop(StackItemType& stackTop)
                         throw(StackException);

   void getTop(StackItemType& stackTop) const
                                        throw(StackException);

private:

   StackItemType items[MAX_STACK];
   int   top;            // index to top of stack

}; // end class
// End of header file.
```

```
// Implementation file StackA.cpp for the ADT stack.
// Array-based implementation.

#include "StackA.h" // Stack class specification file

Stack::Stack(): top(-1)
{
} // end default constructor

bool Stack::isEmpty() const
{
  return ( top < 0 );
} // end isEmpty

void Stack::push(StackItemType newItem)
{
  if (top >= MAX_STACK-1)        // stack full
    throw StackException(
                "StackException: stack full on push");
  else                           // push newItem
  {
     ++top;
     items[top] = newItem;
  } // end if
} // end push
```

```
void Stack::pop()                         // stack is empty
{
   if (isEmpty())
     throw StackException(
                  "StackException: stack empty on pop");
   else                                   // pop stack
     --top;
} // end pop

void Stack::pop(StackItemType& stackTop)
{
   if (isEmpty())
     throw StackException(
                  "StackException: stack empty on pop");
   else                         // retrieve top of stack
   {
      // stack is not empty, retrieve top
     stackTop = items[top];
     --top;                     // pop top off stack
   } // end if
} // end pop
```

```
void Stack::getTop(StackItemType& stackTop) const
{
   if (isEmpty())
       throw StackException(
                      "StackException: stack empty on getTop");
   else                                   // retrieve top of stack
       stackTop = items[top];
} // end getTop
// End of implementation file.
```

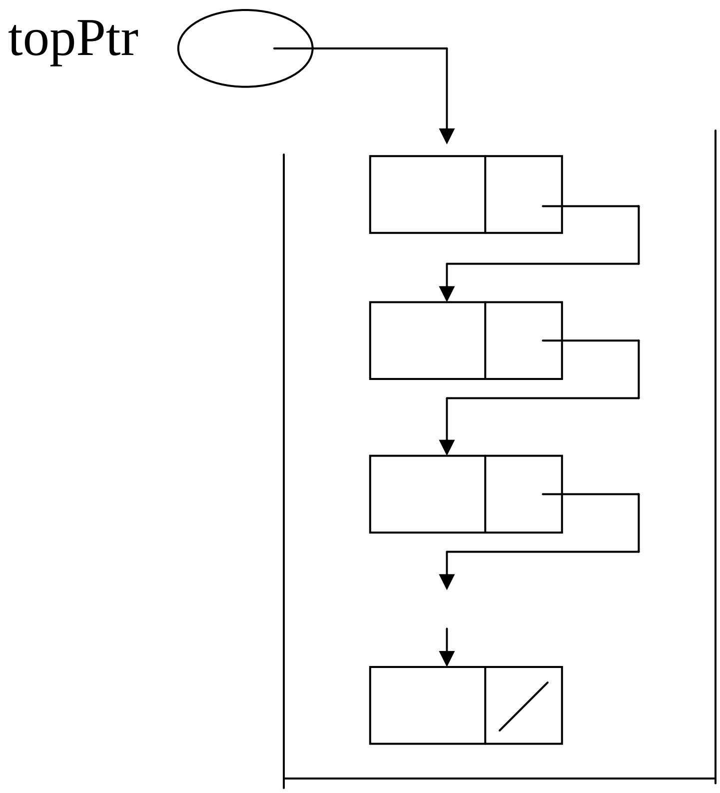
StackException.h File:

```
#include <exception>
#include <string>
using namespace std;

class StackException: public runtime_error
{
public:
   StackException(const string & message="")
              : exception(message.c_str())
   {
    }
}; // end StackException
```

Using the class Stack:

```
#include <iostream>
#include "StackA.h"
using namespace std;

int main()
{
   StackItemType anItem;
   Stack aStack;
       …
   cin >> anItem;       // read an item
   aStack.push(anItem); // push it onto stack
       …
}
```

Pointer-Based Implementation of Stack:



```
// Header file StackP.h for the ADT stack. P. 296
// Pointer-based implementation.

#include "StackException.h"
typedef desired-type-of-stack-item StackItemType;

class Stack
{
public:
// constructors and destructor:
   Stack();                          // default constructor
   Stack(const Stack& aStack);  // copy constructor
   ~Stack();                         // destructor

// stack operations:              // same as in array imp
   bool isEmpty() const;
   void push(StackItemType newItem)
                    throw(StackException);
   void pop() throw(StackException);
   void pop(StackItemType& stackTop)
                    throw(StackException);
   void getTop(StackItemType& stackTop) const
                    throw(StackException);
```

```
private:
   struct StackNode           // a node on the stack
   {
      StackItemType item;   // a data item on the stack
      StackNode *next;       // pointer to next node
   }; // end struct

   StackNode *topPtr;        // pointer to top of stack
}; // end Stack class
// End of header file.
```

```
// Implementation file StackP.cpp for the ADT stack.
// Pointer-based implementation.

#include "StackP.h" // header file
#include <cstddef>  // for NULL
#include <cassert>  // for assert

Stack::Stack() : topPtr(NULL)
{
} // end default constructor
```

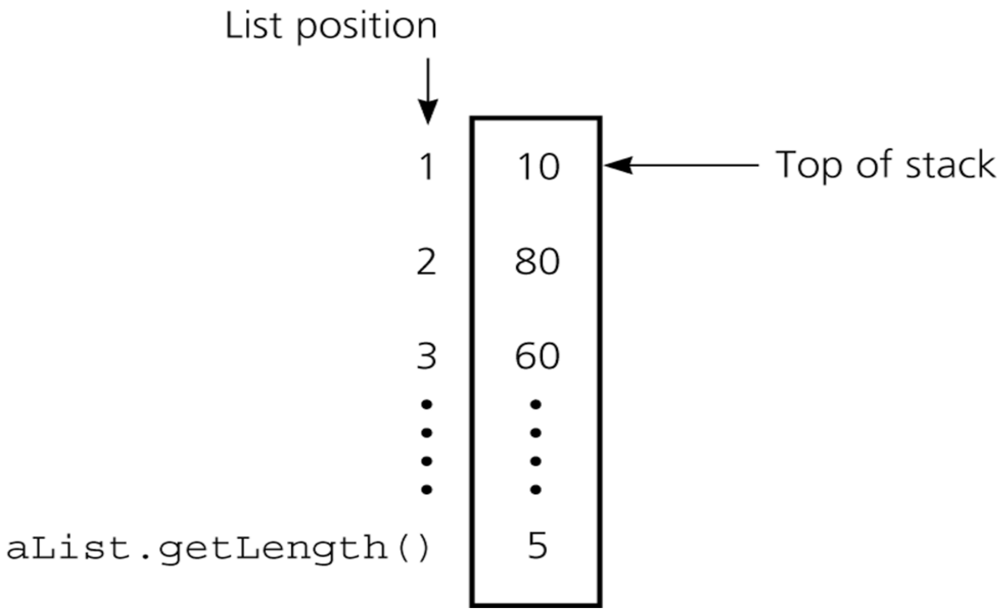
```
Stack::Stack(const Stack& aStack)
{
  if (aStack.topPtr == NULL)
    topPtr = NULL;              // original list is empty

  else
  {
    // copy first node
     topPtr = new StackNode;
     assert(topPtr != NULL);
     topPtr->item = aStack.topPtr->item;

    // copy rest of list
     StackNode *newPtr = topPtr; // new list pointer
     for (StackNode *origPtr = aStack.topPtr->next;
                origPtr != NULL; origPtr = origPtr->next)
    {
       newPtr->next = new StackNode;
      assert(newPtr->next != NULL);
      newPtr = newPtr->next;
      newPtr->item = origPtr->item;
    } // end for

     newPtr->next = NULL;
  } // end else
} // end copy constructor
```

```
Stack::~Stack()
{
  while (!isEmpty())              // pop until stack is empty
    pop();
} // end destructor

bool Stack::isEmpty() const
{
  return ( topPtr == NULL );
} // end isEmpty

void Stack::push(StackItemType newItem)
{
  // create a new node
  StackNode *newPtr = new StackNode;

  if (newPtr == NULL)                    // out of memory
    throw StackException(
   "StackException: stack push cannot allocate memory");
  else
  {  // allocation successful; set data portion of new node
    newPtr->item = newItem;
    // insert the new node
    newPtr->next = topPtr;
    topPtr = newPtr;
  } // end if
} // end push
```

```
void Stack::pop()
{
  if (isEmpty())
    throw StackException(
            "StackException: stack empty on pop");
  else
  {  // stack is not empty; delete top
    StackNode *temp = topPtr;
    topPtr = topPtr->next;
    // return deleted node to system
    temp->next = NULL;             // safeguard
    delete temp;
  }
} // end pop
```

```
void Stack::pop(StackItemType& stackTop)
{
  if (isEmpty())
    throw StackException(
      "StackException: stack empty on pop");
  else
  {  // stack is not empty; retrieve and delete top
    stackTop = topPtr->item;
    StackNode *temp = topPtr;
    topPtr = topPtr->next;

    // return deleted node to system
    temp->next = NULL;
    delete temp;
  } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop) const
{
  if (isEmpty())
    throw StackException(
          "StackException: stack empty on getTop");
  else
    // stack is not empty; retrieve top
    stackTop = topPtr->item;
} // end getTop
// End of implementation file.
```

Implementation Using ADT List:



Maximizing code re-use!

Example:

Stack Op	**List Op**
push(newItem)	insert(1,newItem)
pop( )	remove(1)
getTop(stackTop)	retrieve(1,stackTop)

```
// Header file ListP.h for the ADT list on P.300
// Pointer-based implementation.
#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
typedef desired-type-of-list-item ListItemType;

class List
{
public:
   List();
   List(const List& aList);
   ~List();

// list operations:
   bool isEmpty() const;
   int getLength() const;
   void insert(int index, ListItemType newItem)
      throw(ListIndexOutOfRangeException,ListException);
   void remove(int index)
            throw(ListIndexOutOfRangeException);
    void retrieve(int index, ListItemType& dataItem) const
            throw(ListIndexOutOfRangeException);
```

```
private:
  struct ListNode
  {
    ListItemType item;
    ListNode *next;
  }; // end struct

  int size;
  ListNode *head;         // pointer to linked list of items

  ListNode *find(int index) const;
}; // end class
// End of header file.


// Header file StackL.h for the ADT stack.
// ADT list implementation.
#include "StackException.h"
#include "ListP.h"                    // list operations
typedef ListItemType StackItemType;

class Stack
{
public:
  Stack();                            // default constructor
  Stack(const Stack& aStack);         // copy constructor
  ~Stack();                           // destructor
```

```
// Stack operations:
   bool isEmpty() const;
   void push(StackItemType newItem)
                              throw(StackException);
   void pop() throw (StackException);
   void pop(StackItemType& stackTop)
                              throw(StackException);
   void getTop(StackItemType& stackTop) const
                              throw(StackException);

private:
   List aList; // list of stack items
}; // end class
// End of header file.


// Implementation file StackL.cpp for the ADT stack.
// ADT list implementation.

#include "StackL.h" // header file

Stack::Stack()
{
} // end default constructor

Stack::Stack(const Stack& aStack): aList(aStack.aList)
{
} // end copy constructor
```

```
Stack::~Stack()
{
} // end destructor

bool Stack::isEmpty() const
{
  return aList.isEmpty();
} // end isEmpty

void Stack::push(StackItemType newItem)
{
  try
  {
    aList.insert(1, newItem);
  } // end try
  catch (ListException e)
  {
    throw StackException(
      "StackException: cannot push item");
  } // end catch
} // end push
```

```
void Stack::pop()
{
  try
  {
    aList.remove(1);
  } // end try
  catch (ListIndexOutOfRangeException e)
  {
    throw StackException(
      "StackException: stack empty on pop");
  } // end catch
} // end pop

void Stack::pop(StackItemType& stackTop)
{
  try
  {
  aList.retrieve(1, stackTop);
  aList.remove(1);
  } // end try
  catch (ListIndexOutOfRangeException e)
  {
  throw StackException(
    "StackException: stack empty on pop");
  } // end catch
} // end pop
```

```
void Stack::getTop(StackItemType& stackTop) const
{
   try
   {
      aList.retrieve(1, stackTop);
   } // end try
   catch (ListIndexOutOfRangeException e)
   {
      throw StackException(
         "StackException: stack empty on getTop");
   } // end catch
} // end getTop
// End of implementation file.
```

More on Algebraic Expressions:

Recall that postfix expressions can be defined as:

<postfix> = <identifier>|<postfix><postfix><operator>

<identifier> = *a* | *b* | … | *z*

<operator> = + | - | * | /

Hence, we can use a stack to evaluate a postfix expression by reading in the symbols one at a time. If the symbol is an operand, or the value of a postfix expression, then push it onto the stack. Else if the symbol is an operator, pop the stack twice to get two operands and then apply the operator to them.

Algorithm:

for each symbol in postfix
if operand, push its value;
if operator
pop top two operand values;
apply operator;
push result back on stack;
value at top of stack is result of expression;

Example: Evaluating 2 ∗ (3 + 4).
Postfix: 2 3 4 + ∗

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

Infix (with parentheses) to Postfix Conversion: Observation:

From the examples we see:

- Operands (A, B, etc.) never change order. While reading an infix expression, an operand can be written to the developing postfix string immediately.
- Operators, on the other hand, generally move to the right ***and*** change order
  - "Move to the right" because both operands must have already been output
  - "Switch order" because: I cannot apply an operator (i.e., write it to the postfix string) <u>until I know whether the next operator has higher precedence</u>.)

The algorithm therefore requires a stack to hold a list of "pending operators", ordered from lower to higher precedence. This will require us to design and implement policies for when we push/pop operators so that "lower to higher precedence" *is the same as* "least recently pushed to most recently pushed". When an operator is encountered in the infix expression, if the stack is empty OR if the operator has strictly higher precedence than what is at the top of the stack, then push the operator, else pop and append operators until *either* the stack is empty *or* an operator of lower precedence is encountered.

Algorithm:

```
create an initially empty postfix string
create an initially empty operator stack
for each symbol, S, in the infix string do
   if S is an operand then
            append it to the postfix string
   else if S == ‘(‘ then
            push S
   else if S == ‘)’ then
            pop and append operators until the matching ‘(‘ is
                 encountered
            else // must be some other operator
            {  while operator stack not empty and
                   precedence(tos) = precedence(S) and
                      tos != ‘(‘
                   pop operator & append it to the postfix string
               end while
               push S
            }
end for;
while operator stack is not empty
     pop operator
     append it to the postfix string
endwhile;
```

Example: converts the infix expression a - (b + c * d)/e to postfix form**.**

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