Stacks

Stack

- Study a stack as an ADT
- Build a static-array-based implementation of stacks
- Build a dynamic-array-based implementation of stacks
- Build a linked-implementation of stacks

Objectives

- Study a stack as an ADT
- Build a static-array-based implementation of stacks
- Build a dynamic-array-based implementation of stacks
- Build a linked-implementation of stacks
- Show how a run-time stack is used to store information during function calls
- (Optional) Study postfix notation and see how stacks are used to convert expressions from infix to postfix and how to evaluate postfix expressions

Stack

- This last-in-first-out (LIFO) data structure is called a **Stack**
- Adding an item (push)
  - Referred to as pushing it onto the stack
- Removing an item (pop)
  - Referred to as popping it from the stack

Stack

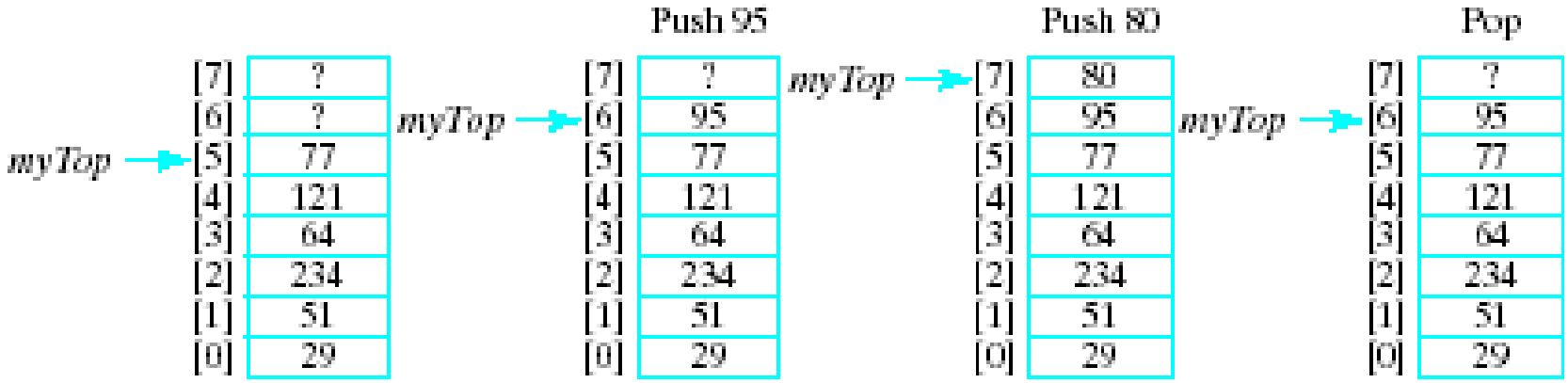
- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)
- Operations:
  - construct a stack (usually empty)
  - check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element (read)
  - Pop: remove the top element

Stack class

- The basic functions are:
  - Constructor: construct an empty stack
  - Empty(): Examines whether the stack is empty or not
  - Push(): Add a value at the top of the stack
  - Top(): Read the value at the top of the stack
  - Pop(): Remove the value at the top of the stack
  - Display(): Displays all the elements in the stack

Storage structures:
Select position 0 as bottom of the stack

- A better approach is to let position 0 be the bottom of the stack



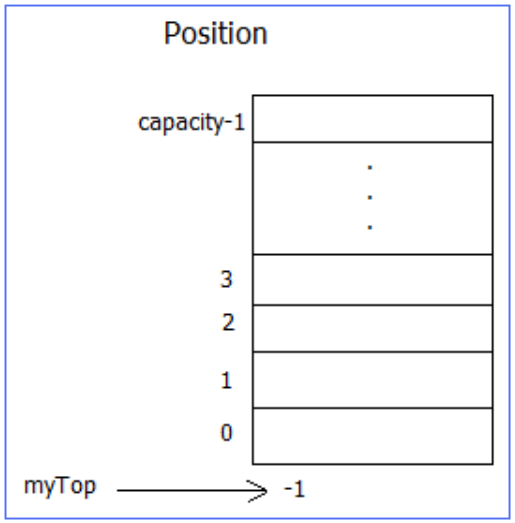
- Thus our design will include
  - An array to hold the stack elements
  - An integer to indicate the top of the stack

Implementation of the Operations

- Constructor:

  Create an array: (int) array[capacity]

  Set myTop = -1



- Empty():

  check if myTop == -1

Implementation of the Operations

- Push(int x):

  if array is not FULL (myTop < capacity-1)

  myTop++

  store the value x in array[myTop]

  else

  output “out of space”

Implementation of the Operations

- Top():

  If the stack is not empty

  return the value in array[myTop]

  else:

  output “no elements in the stack”

Implementation of the Operations

- Pop():

  If the stack is not empty

  myTop −= 1

  else:

  output “no elements in the stack”

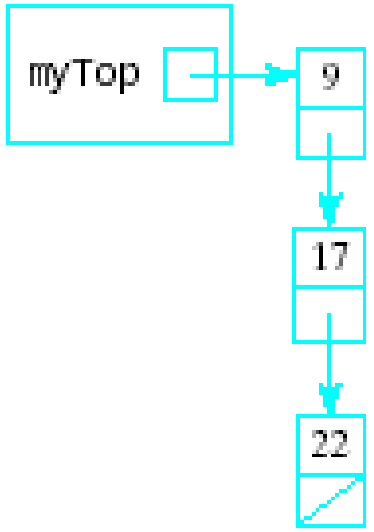
Further Considerations

- What if static array initially allocated for stack is too small?
  - Terminate execution?
  - Replace with larger array!

- Creating a larger array
  - Allocate larger array
  - Use loop to copy elements into new array
  - Delete old array

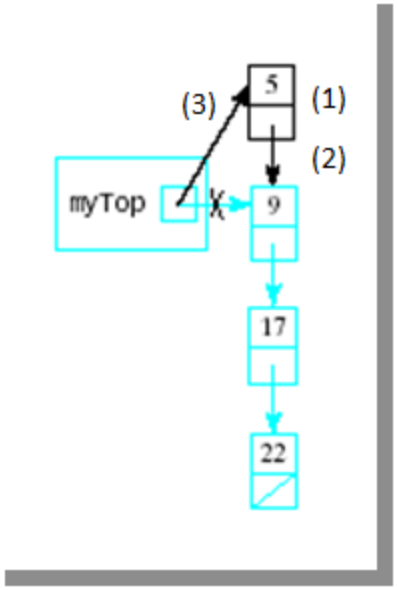
Linked Stacks

- Another alternative to allowing stacks to grow as needed
- Linked list stack needs only one data member
  - Pointer **`myTop`**
  - Nodes allocated (but not part of stack class)



Implementing Linked Stack Operations

- Constructor
  - Simply assign null pointer to `myTop`
- Empty
  - Check for `myTop == null`



- Push
  - Insertion at beginning of list

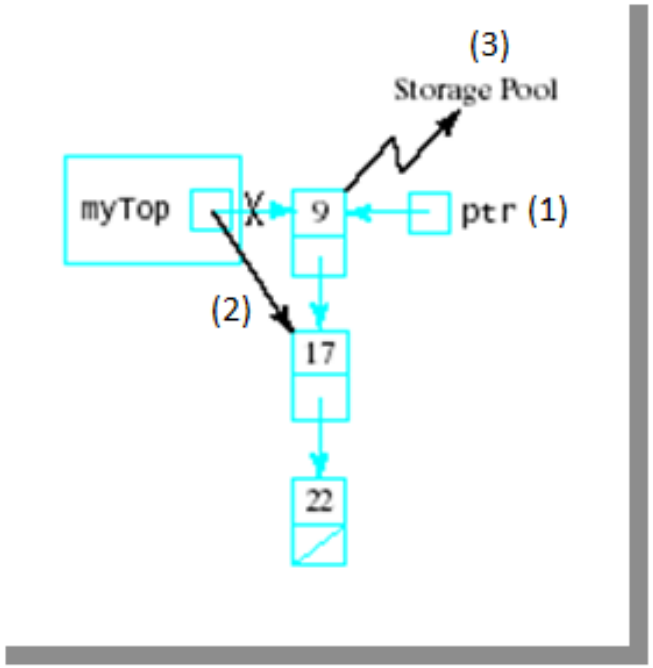
```
myTop == new stack::Node(value, mytop)
```

- Top
  - Return data to which `myTop` points

Implementing Linked Stack Operations

- Pop
  - Delete first node in the linked list

```
ptr = myTop;
myTop = myTop->next;
delete ptr;
```



- Output
  - Traverse the list

```
for (ptr = myTop;
        ptr != 0; ptr = ptr->next)
   out << ptr->data << endl;
```

- Stacks structures are usually implemented using arrays or linked lists.

- For both implementations, the running time is O(n).

Stack Applications

- Reversing Data: We can use stacks to reverse data.
  (example: files, strings)
  Very useful for finding palindromes.

Consider the following pseudocode:

1) read (data)
2) loop (data not EOF and stack not full)
   1) push (data)
   2) read (data)
3) Loop (while stack notEmpty)
   1) pop (data)
   2) print (data)

Stack Applications

- Converting Decimal to Binary: Consider the following pseudocode

  1) Read (number)
  2) Loop (number > 0)
     1) digit = number modulo 2
     2) print (digit)
     3) number = number / 2

  // from Data Structures by Gilbert and Forouzan

  The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.)

  To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Stack Applications

- Postponement: Evaluating arithmetic expressions.

- Prefix: + a b
- Infix: a + b (what we use in grammar school)
- Postfix: a b +

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Infix to Postfix Conversion

Infix to Postfix Example

A + B * C - D / E

	Infix	**Stack(bot->top)**	**Postfix**
a)	A + B * C - D / E		
b)	+ B * C - D / E		A
c)	B * C - D / E	+	A
d)	* C - D / E	+	A B
e)	C - D / E	+ *	A B
f)	- D / E	+ *	A B C
g)	D / E	+ -	A B C *
h)	/ E	+ -	A B C * D
i)	E	+ - /	A B C * D
j)		+ - /	A B C * D E
k)			A B C * D E / - +

Infix to Postfix Example

A + B * C - D / E

	Infix	**Stack(bot->top)**	**Postfix**
a)	A + B * C - D / E		
b)	+ B * C - D / E		A
c)	B * C - D / E	+	A
d)	* C - D / E	+	A B
e)	C - D / E	+ *	A B
f)	- D / E	+ *	A B C
g)	D / E	+ -	A B C *
h)	/ E	+ -	A B C * D
i)	E	+ - /	A B C * D
j)		+ - /	A B C * D E
k)			A B C * D E / - +

Postfix Evaulation

```
Operand: push
Operator: pop 2 operands, do the math, pop result
          back onto stack
```

1 2 3 + *

Postfix	Stack(bot -> top)
a) 1 2 3 + *	
b) 2 3 + *	1
c) 3 + *	1 2
d) + *	1 2 3
e) *	1 5 // 5 from 2 + 3
f)	5 // 5 from 1 * 5

C/C++ Standard library

- The **C standard library** (also known as **libc**) is a now-standardized collection of header files and library routines used to implement common operations, such as input/output and string handling

- For example:

  #include <iostream>

Vector

- Vectors contain contiguous elements stored as an **Dynamic** array.

- All you have to do is include vector from library

  #include <vector>

Vector Functions

Vector constructors	create vectors and initialize them with some data
Vector operators	compare, assign, and access elements of a vector
assign	assign elements to a vector
at	returns an element at a specific location
back	returns a reference to last element of a vector
begin	returns an iterator to the beginning of the vector
capacity	returns the number of elements that the vector can hold
clear	removes all elements from the vector
empty	true if the vector has no elements
end	returns an iterator just past the last element of a vector
erase	removes elements from a vector
front	returns a reference to the first element of a vector
insert	inserts elements into the vector
max_size	returns the maximum number of elements that the vector can hold
pop_back	removes the last element of a vector
push_back	add an element to the end of the vector
rbegin	returns a reverse_iterator to the end of the vector
rend	returns a reverse_iterator to the beginning of the vector
reserve	sets the minimum capacity of the vector
resize	change the size of the vector
size	returns the number of items in the vector
swap	swap the contents of this vector with another

Designing and Building a Stack class

- The basic functions are:
  - Constructor: construct an empty stack
  - Empty(): Examines whether the stack is empty or not
  - Push(): Add a value at the top of the stack
  - Top(): Read the value at the top of the stack
  - Pop(): Remove the value at the top of the stack
  - Display(): Displays all the elements in the stack

Functions related to Stack

- Constructor: vector<int> L;
- Empty(): L.size() == 0?
- Push(): L.push_back(value);
- Top(): L.back();
- Pop(): L.pop_back();
- Display(): Write your own

Example

```
#include <iostream>
#include <vector>
using namespace std;


char name[20];
int i, j, k;

int main()
{

	vector<int> L;
	L.push_back(1);
	L.push_back(2);
	L.push_back(3);
	L.pop_back();

	for(i=0;i<L.size();i++)
		cout << L[i] << " ";

	cout << L.back();

	cin >> name;
}
```

Use of Stack in Function calls

- Whenever a function begins execution, an **activation record** is created to store the **current environment** for that function

- Current environment includes the
  - values of its parameters,
  - contents of registers,
  - the function's return value,
  - local variables
  - address of the instruction to which execution is to **return** when the function finishes execution (If execution is interrupted by a call to another function)

Use of Stack in Function calls

- Functions may call other functions and thus interrupt their own execution, some data structure must be used to store these activation records so they can be recovered and the system can be reset when a function resumes execution
- It is the fact that the last function interrupted is the first one reactivated
- It suggests that a stack can be used to store these activation records
- A stack is the appropriate structure, and since it is manipulated during execution, it is called the **run-time stack**

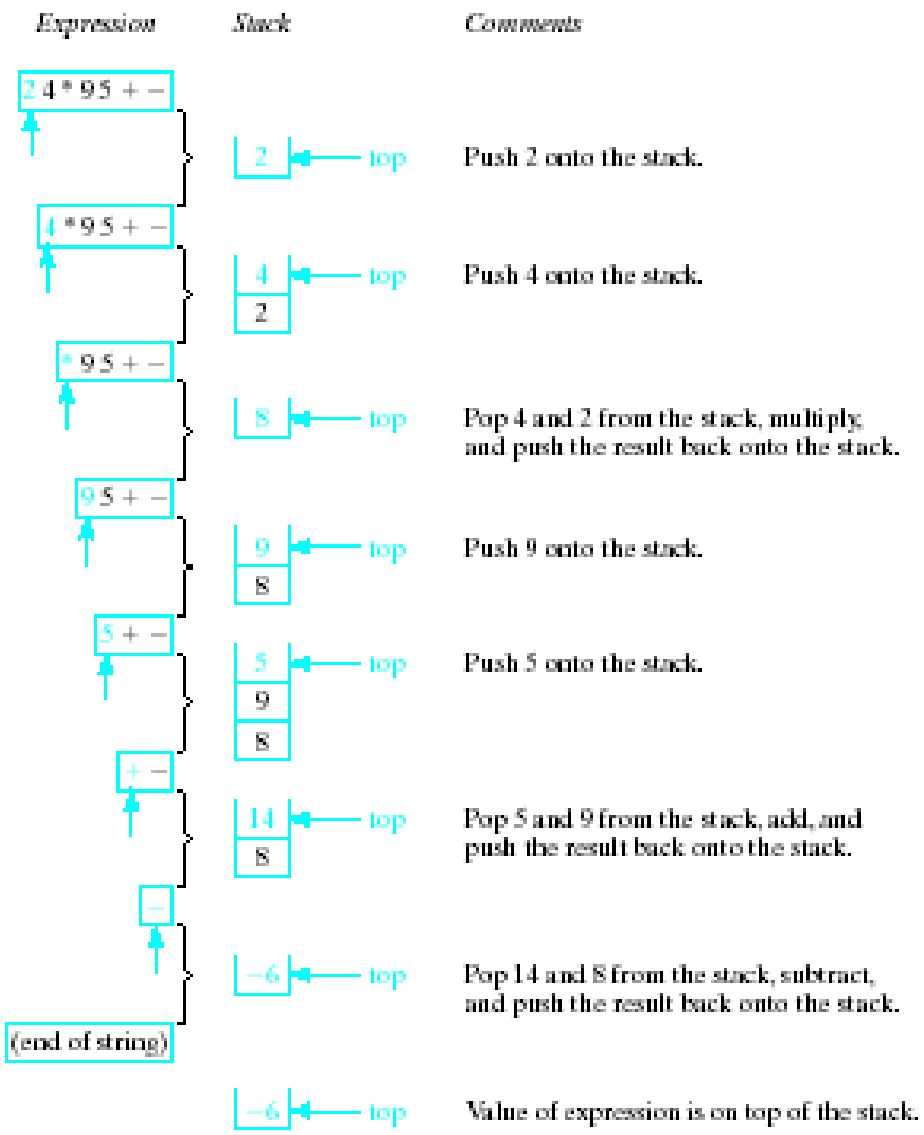
Stack, Queue, Array

- Stack & Queue vs. Array
  - Arrays are data storage structures while stacks and queues are specialized DS and used as programmer's tools.
- Stack – a container that allows push and pop
- Queue - a container that allows enqueue and dequeue
- No concern on implementation details.
- In an array any item can be accessed, while in these data structures access is restricted.
- They are more abstract than arrays.

Infix -> Postfix

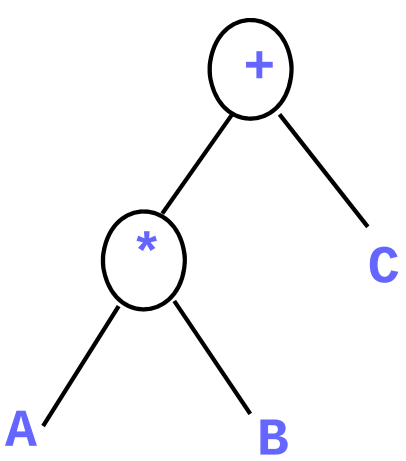
ch	aStack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators from stack to
	– (	abcd*+	postfixExp until "("
	–	abcd*+	
/	– /	abcd*+	Copy operators from
e	– /	abcd*+e	stack to postfixExp
		abcd*+e/–	

Evaluation of Postfix

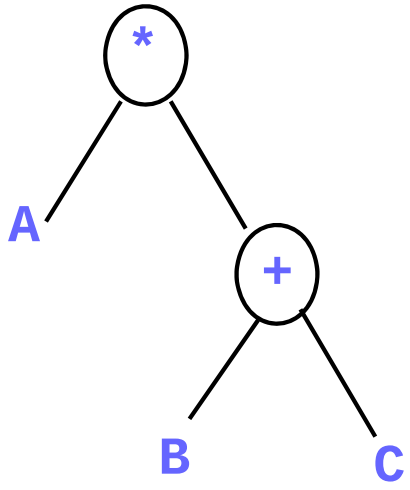


Converting infix to postfix: Represent infix expression as an *expression tree*

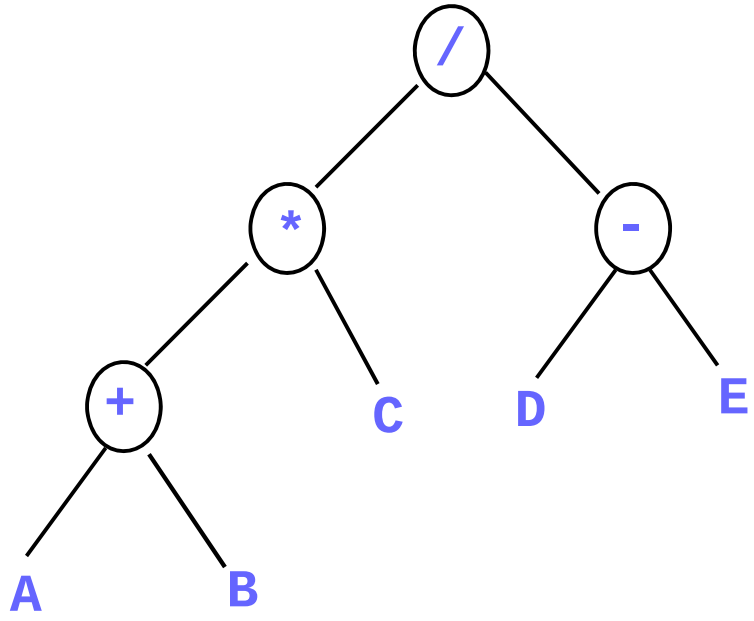
`A * B + C`

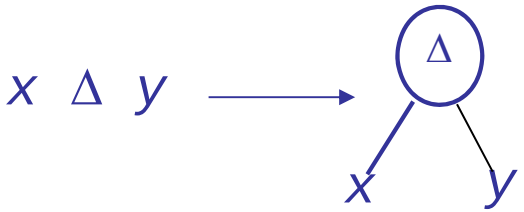


`A * (B + C)`



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





Traverse the tree in *Left-Right-Parent* order (*postorder*) to get Postfix:

A B + C * D E - /

Traverse tree in *Parent-Left-Right* order (*preorder*) to get **prefix:**

/ * + A B C - D E

Traverse tree in *Left-Parent-Right* order (*inorder*) to get **infix:** — must insert ()'s

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