Data Structures and Algorithms

Chapter 3

Stacks

Contents

- Introduction to Stacks
- Designing and Building a Stack Class – Array-Based
- Linked Stacks

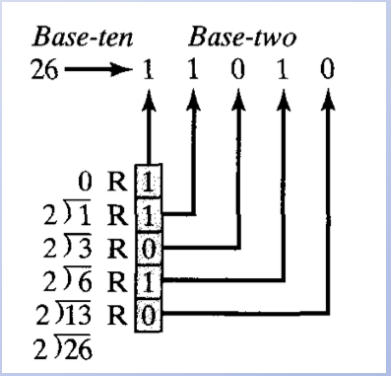
Objectives

- 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

Introduction to Stacks

- Consider a card game with a discard pile
  - Discards always placed on the top of the pile
  - Players may retrieve a card only from the top

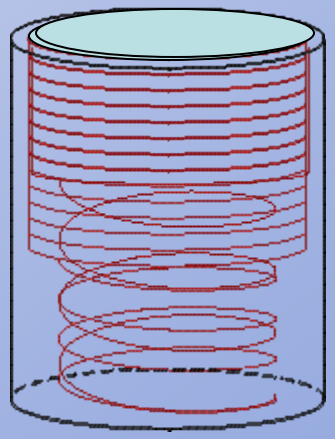
What other examples can you think of that are modeled by a stack?



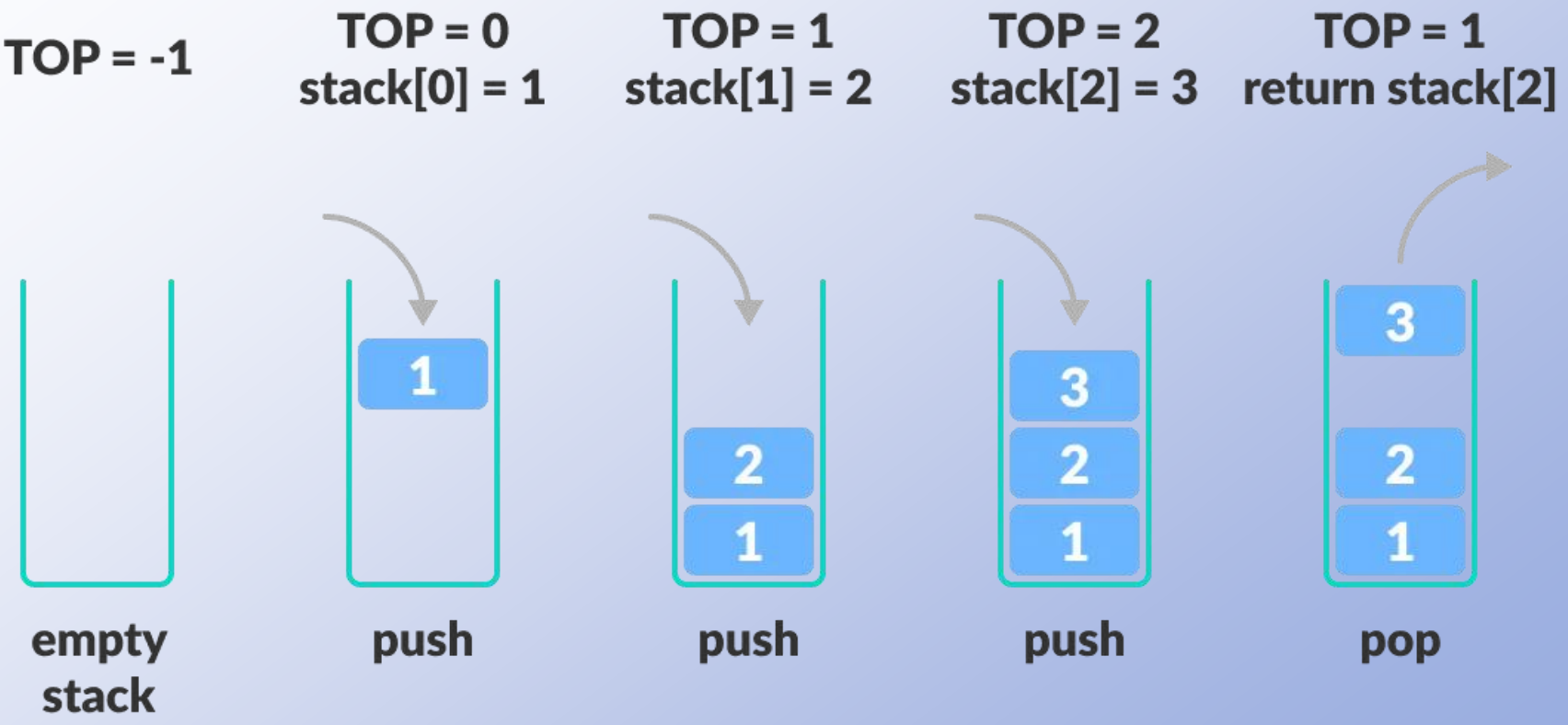
- We seek a way to represent and manipulate this in a computer program
- This is a stack

Introduction to Stacks

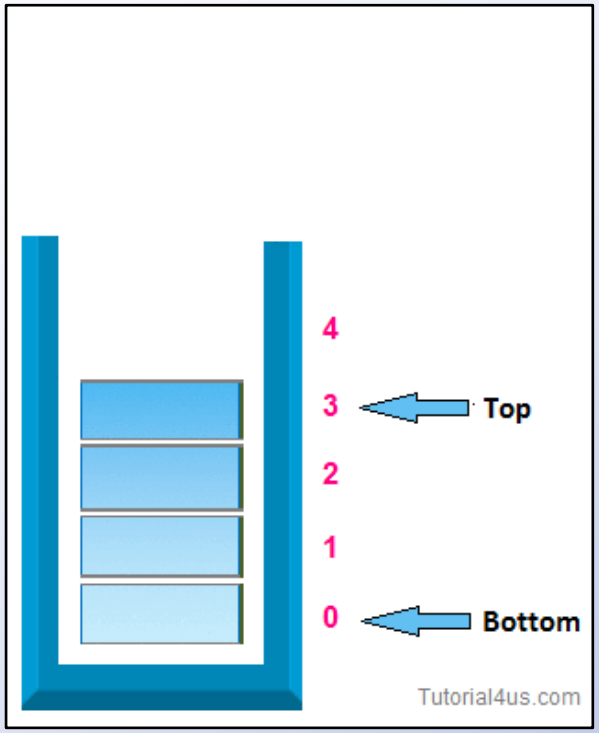
- A stack is a *Last-In-First-Out* (**LIFO**) data structure
- Adding an item
  - Referred to as *pushing* it onto the stack
- Removing an item
  - Referred to as *popping* it from the stack

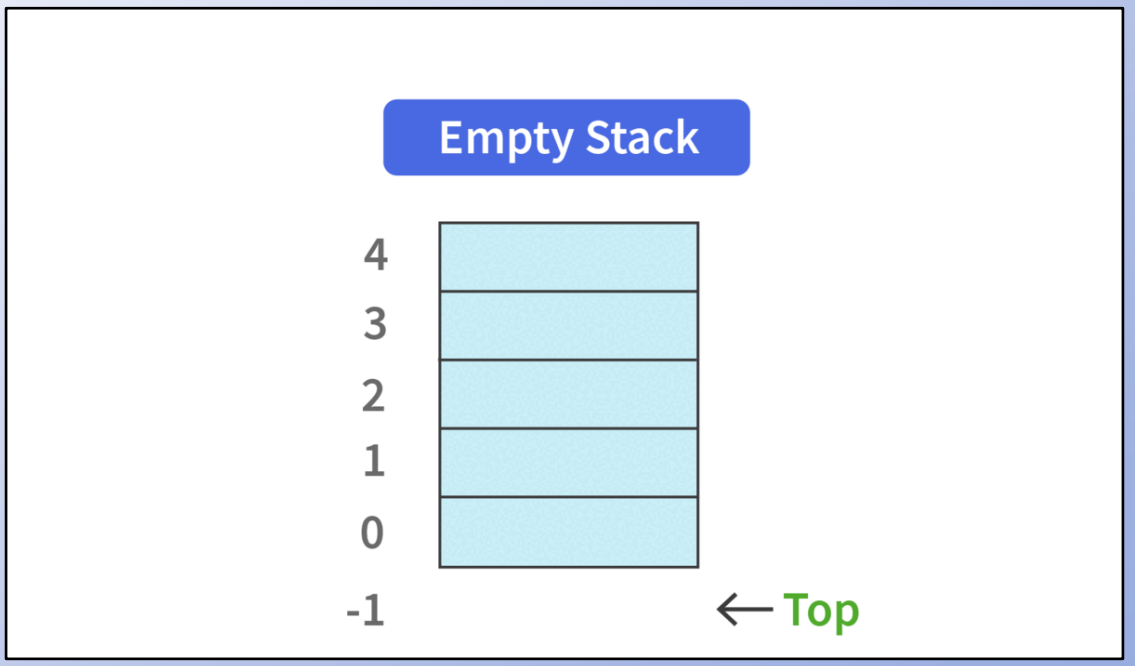


Introduction to Stacks



Introduction to Stacks





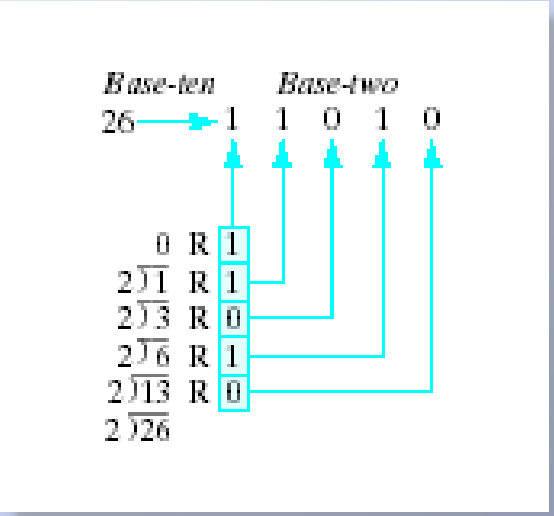
A Stack

- Definition:
  - An ordered collection of data items
  - Can be accessed at only one end (the top)

- Operations:
  - construct a stack (usually empty)
  - Empty: check if it is empty
  - Push: add an element to the top
  - Top: retrieve the top element
  - Pop: remove the top element

Example Program

- Consider a program to do base conversion of a number (Decimal to Binary)



- Note program which assumes existence of a `Stack` class to accomplish this
  - Demonstrates `push`, `pop`, and `top`

Base conversion algorithm

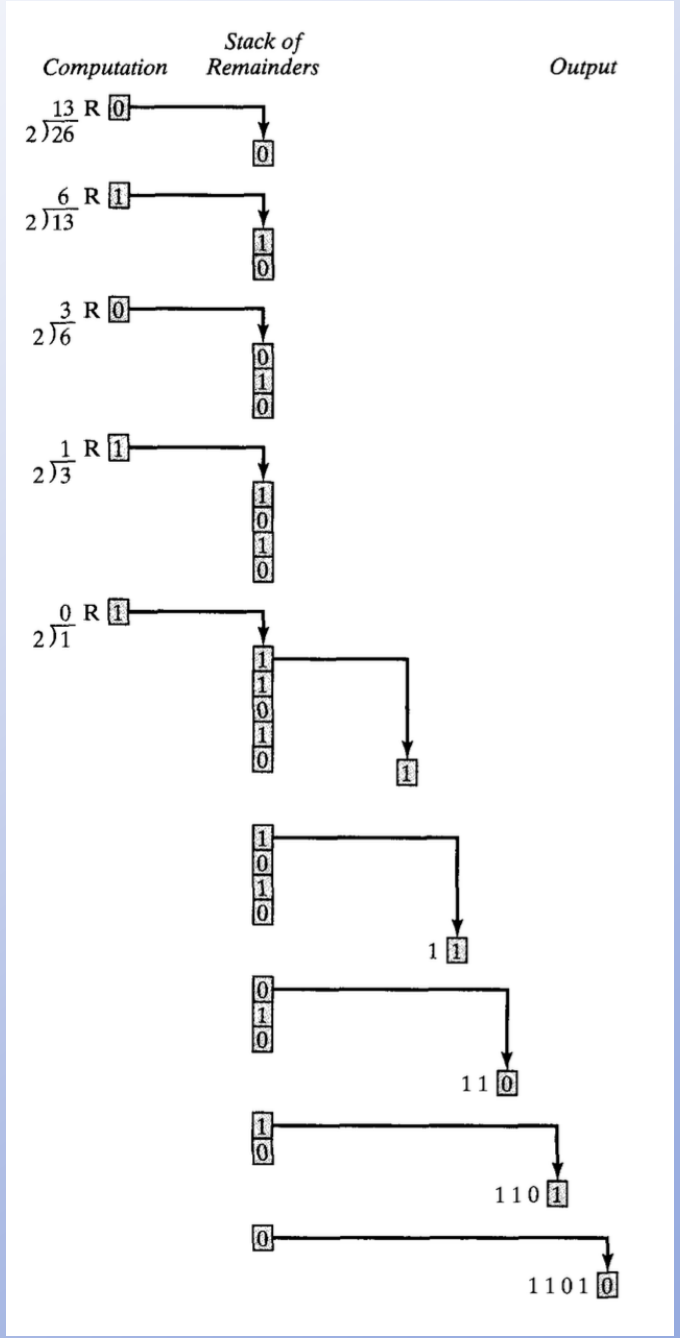
/* Algorithm to display the base-two representation of a base-ten number. */

1. Declare an empty stack to hold the remainders.
2. While *number* ≠ 0:
   a. Calculate the *remainder* that results when *number* is divided by 2.
   b. Put the *remainder* on the top of the stack of remainders.
   c. Replace *number* by the integer quotient of *number* divided by 2.

   End while.
3. While the stack of remainders is not empty,
   a. Retrieve and remove the *remainder* from the top of the stack of remainders.
   b. Append *remainder* to the output already produced.

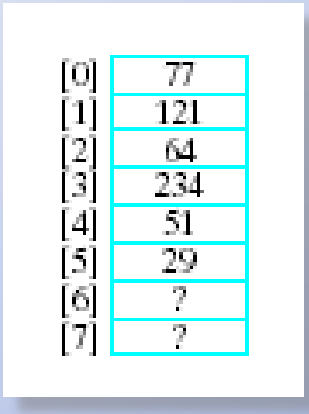
   End while.

Using a Stack to Convert a Number from Base Ten to Base Two

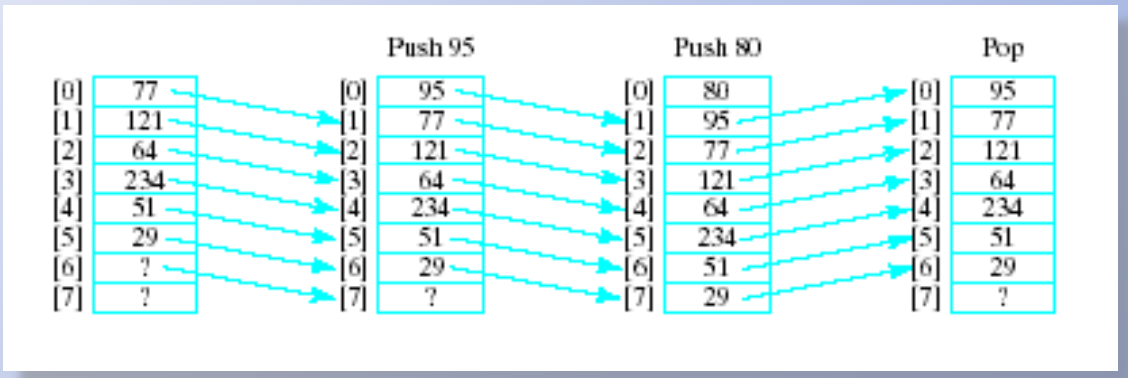


Selecting Storage Structure

- Model with an array
  - Let position 0 be top of stack

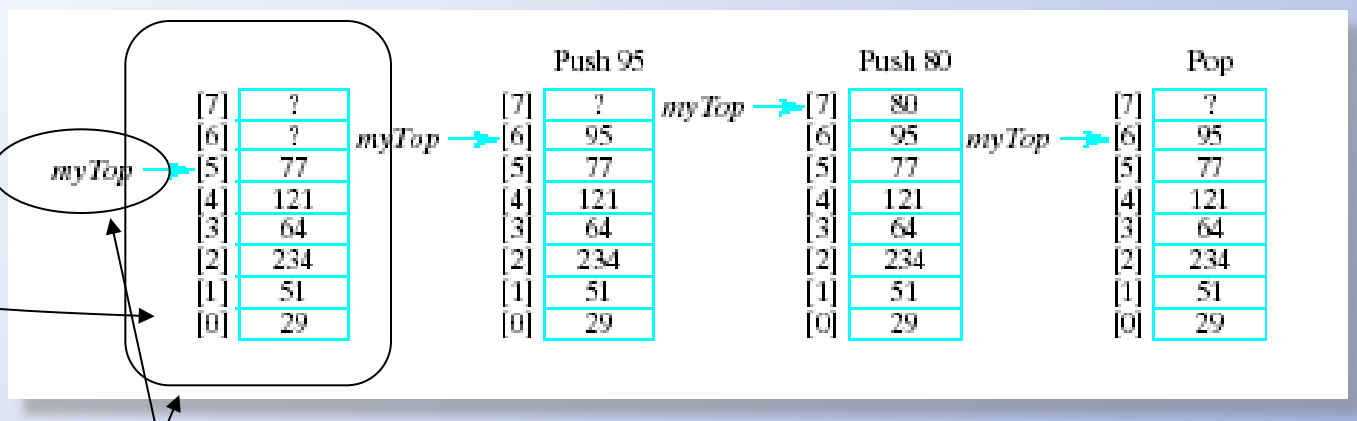


- Problem … consider pushing and popping
  - Requires much shifting



Selecting Storage Structure

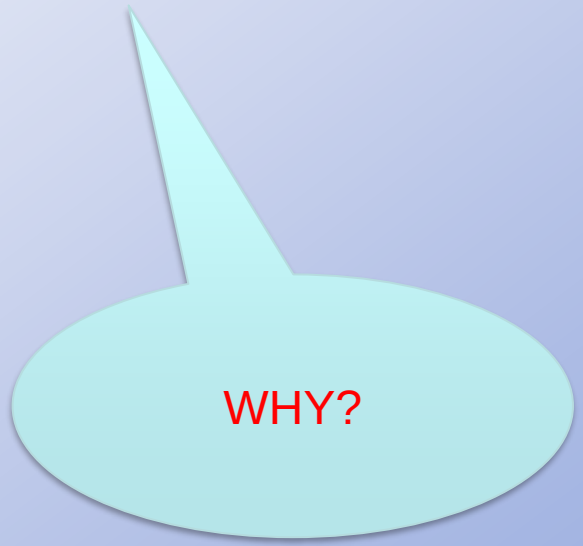
- A better approach is to let position 0 be the bottom of the stack



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

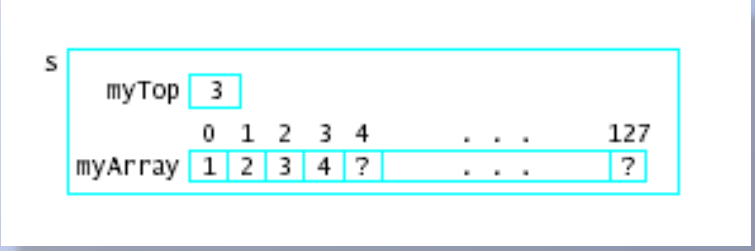
Implementing Operations

- Constructor
  - Compiler will handle allocation of memory, `myTop = -1`
- Empty
  - Check if value of `myTop == -1`
- Push (if `myArray` not full)
  - Increment `myTop` by 1
  - Store value in `myArray [myTop]`
- Top
  - If stack not empty, return `myArray[myTop]`
- Pop
  - If array not empty, decrement `myTop`
- Output routine added for testing



The Stack Class

- The completed `Stack.h` file
  - All functions defined
  - Note use of `typedef` mechanism
- Implementation file, `Stack.cpp`,
- Driver program to test the class,
  - Creates stack of 4 elements
  - Demonstrates error checking for stack full, empty



Dynamic Array to Store Stack Elements

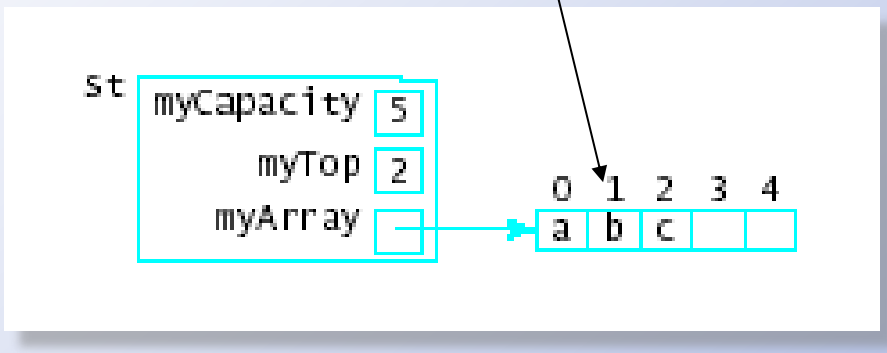
- Same issues regarding static arrays for stacks as for lists
  - Can run out of space if stack set too small
  - Can waste space if stack set too large
- As before, we demonstrate a dynamic array implementation to solve the problems
- Note additional data members required

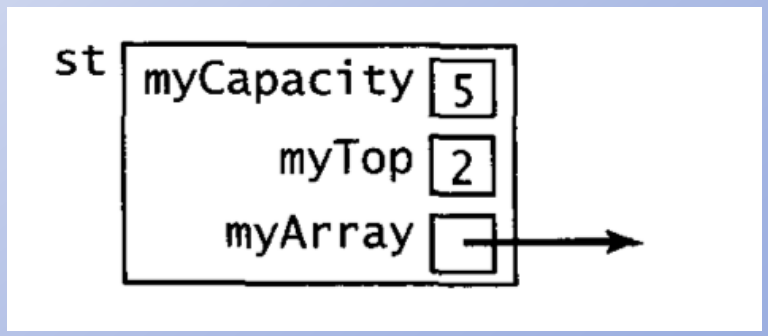
Dynamic Array to Store Stack Elements

- Constructor must
  - Check that specified `numElements > 0`
  - Set capacity to `numElements`
  - Allocate an array pointed to by `myArray` with capacity = `myCapacity`
  - Set `myTop` to -1 if allocation goes OK
- Note implementation of constructor for DStack

Dynamic Array to Store Stack Elements

- Class Destructor needed
  - Avoids memory leak
  - Deallocates array allocated by constructor

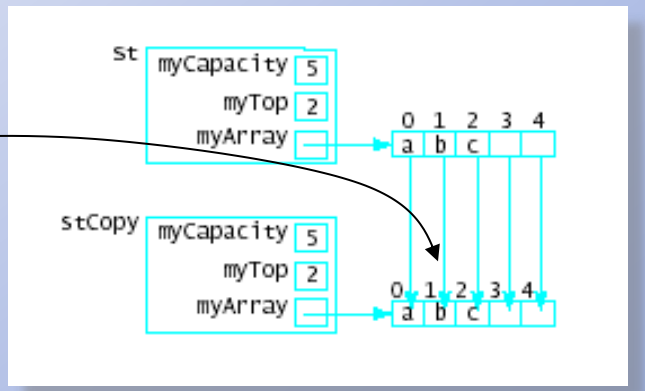




- Note destructor definition

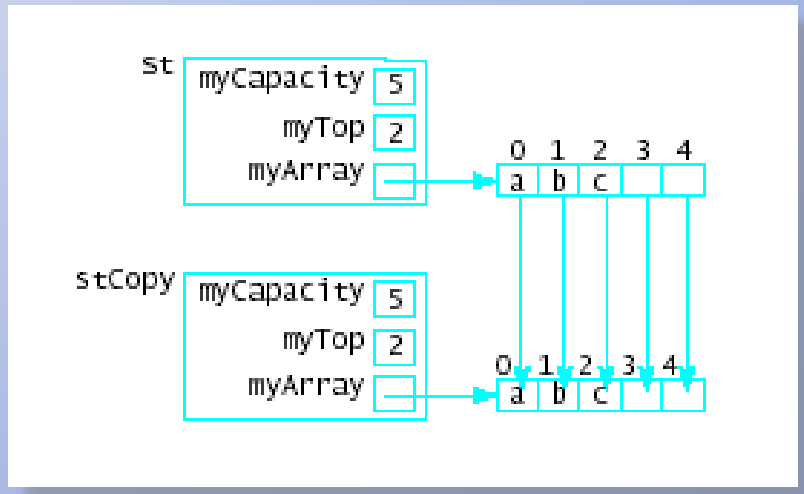
Dynamic Array to Store Stack Elements

- Copy Constructor needed for
  - Initializations
  - Passing value parameter
  - Returning a function value
  - Creating a temporary storage value
- Provides for deep copy
- Note definition



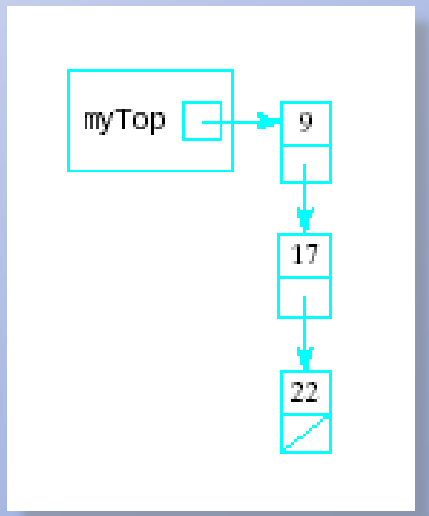
Dynamic Array to Store Stack Elements

- Assignment operator
  - Again, deep copy needed
  - copies member-by-member, not just address
- Note implementation of algorithm in `operator=` definition

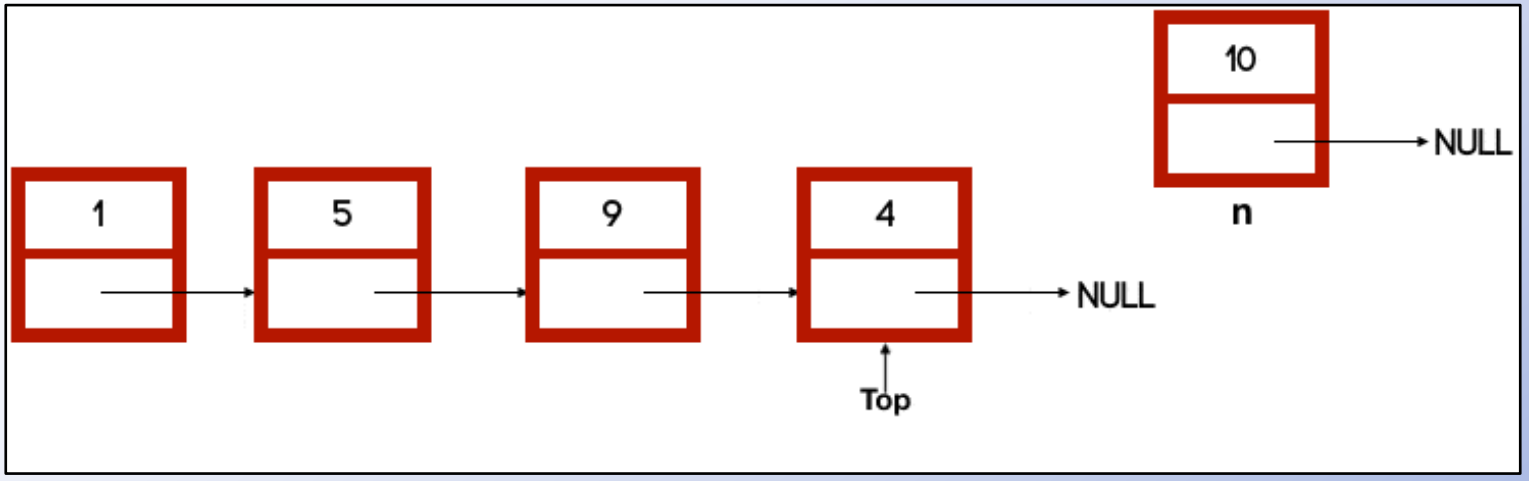


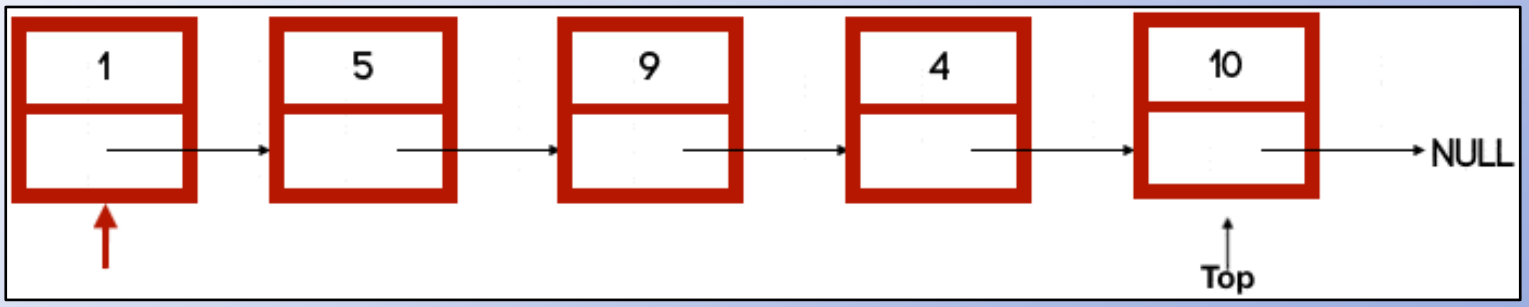
Linked Stacks

- 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)
- Note declaration



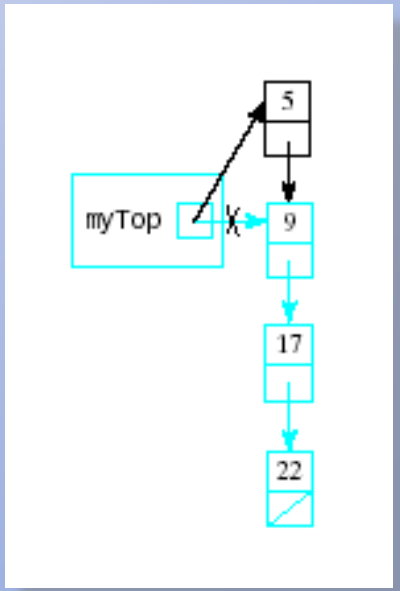
Linked Stacks Graphically





Implementing Linked Stack Operations

- Constructor
  - Simply assign null pointer to `myTop`
- Empty
  - Check for `myTop == null`
- Push
  - Insertion at beginning of list
- Top
  - Return data to which `myTop` points

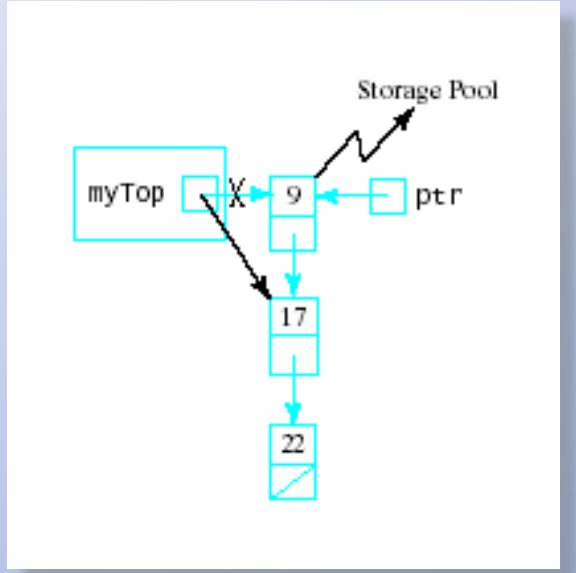


Implementing Linked Stack Operations

Pop

- Delete first node in the linked list

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



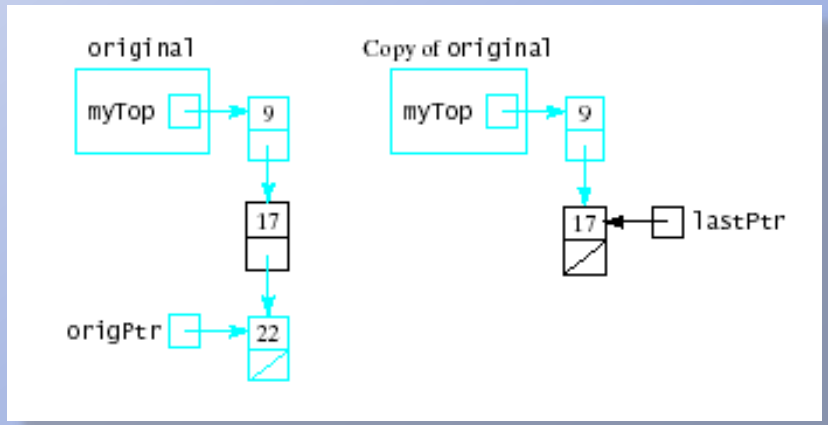
Output

- Traverse the list

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

Implementing Linked Stack Operations

- Destructor
  - Must traverse list and deallocate nodes
  - Note need to keep track of `ptr->next` before calling `delete ptr;`
- Copy Constructor
  - Traverse linked list, copying each into new node
  - Attach new node to copy



Implementing Linked Stack Operations

- Assignment operator
  - Similar to copy constructor
  - Must first rule out self assignment
  - Must destroy list in stack being assigned a new value
- View completed linked list version of stack class,
- Note driver program.