Stacks

CSM 387 – Data Structures

Lecture 5

Data Structure Rules

- Simply using an Array allows for problems
  - Data to be entered anywhere, and removed

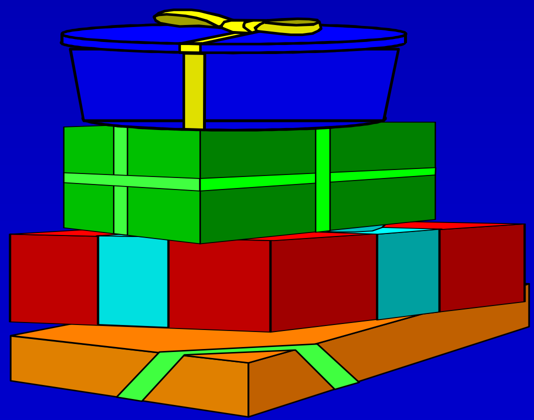
- Apply the constraints of the Stack ADT
  - A stack is an ordered collection of elements into which new items may be inserted and from which elements may be removed, one at a time from the top of the stack only
- Based on First-In-Last-Out - FILO

What is a stack?

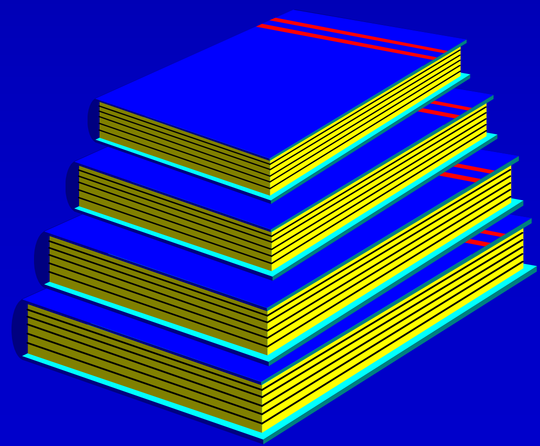
- It is an ordered group of homogeneous items.
- Items are added to and removed from the top of the stack

  LIFO property: Last In, First Out
- The last item added would be the first to be removed

TOP OF THE STACK

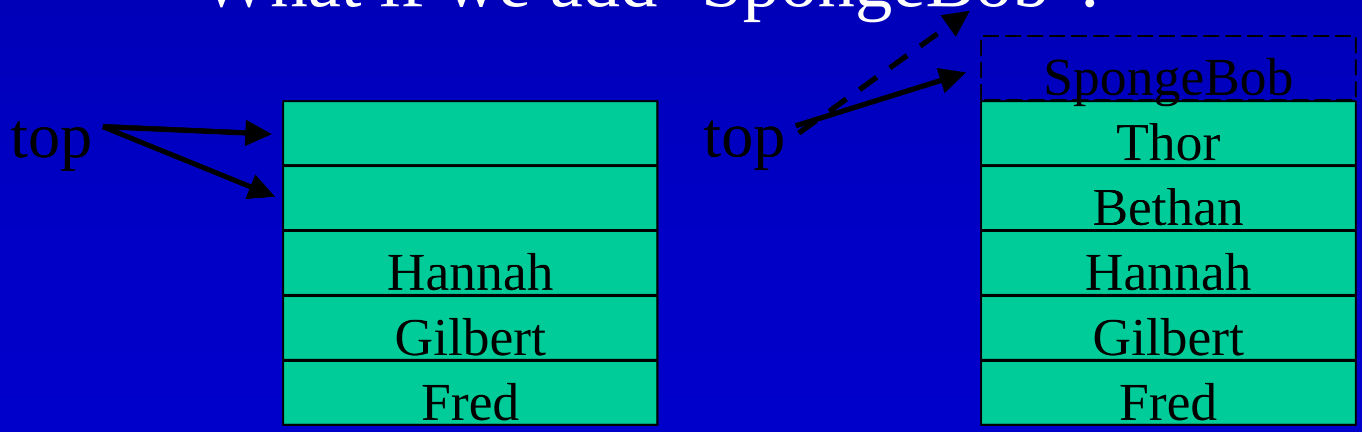


TOP OF THE STACK



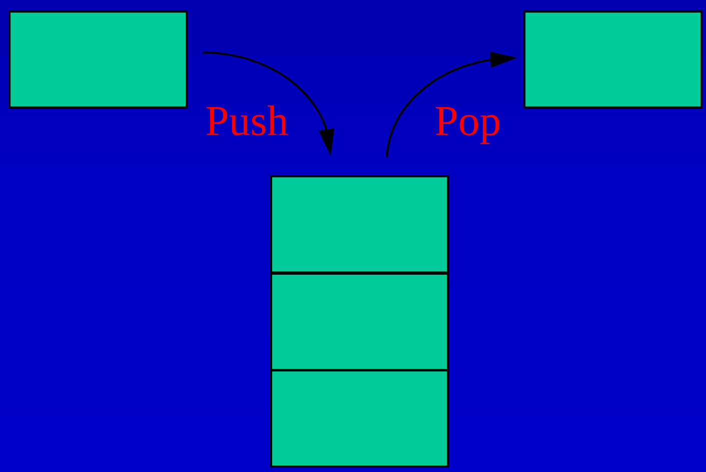
Stack

- 5 *element* stack (array) with 3 values
  - How do we access the data?
  - Control the use of the stack?
- Remember the rules
  - Only access the *top* of the stack
  - Now add ‘Bethan’ & ‘Thor’, we get:
  - What if we add ‘SpongeBob’?



Manipulating the Stack

- We now have some structure to store our data
  - A way to *point* to the _top_ data item
  - Now need to *add* and *retrieve* the data
- Stacks *only* have two mutator methods
- `push()`
  - Push data *onto* the stack
- `pop()`
  - Pop data *off* the stack



push() & pop()

- Insert item *E* on top of the stack

```
public void push(int E)
{
    myStack[topPosition] = E;
    topPosition++;
}
```

- Remove item *E* from the top of the stack

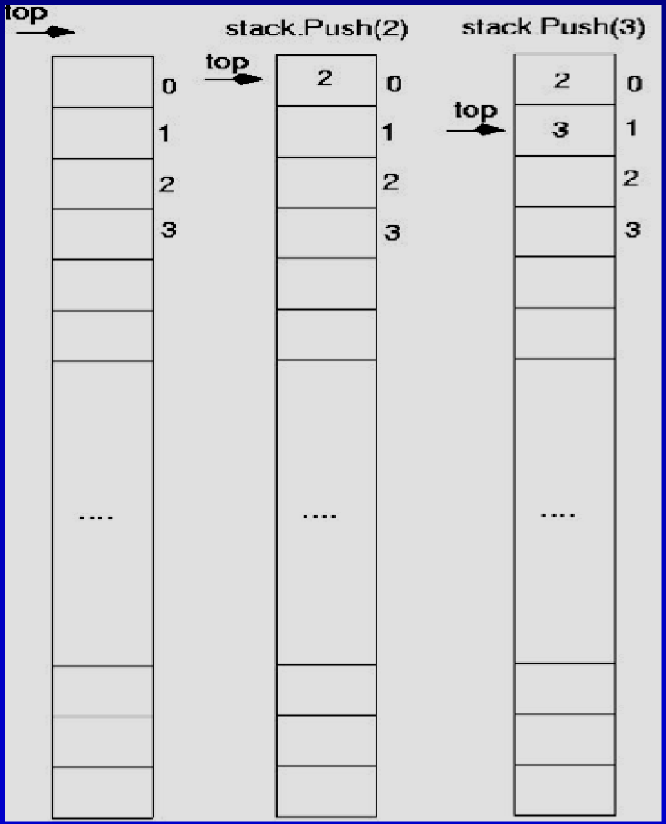
```
public int pop()
{
    int tempE = myStack[topPosition-1];
    myStack[topPosition-1] = 0;
    topPosition--;
    return tempE;
}
```

Stack Overview

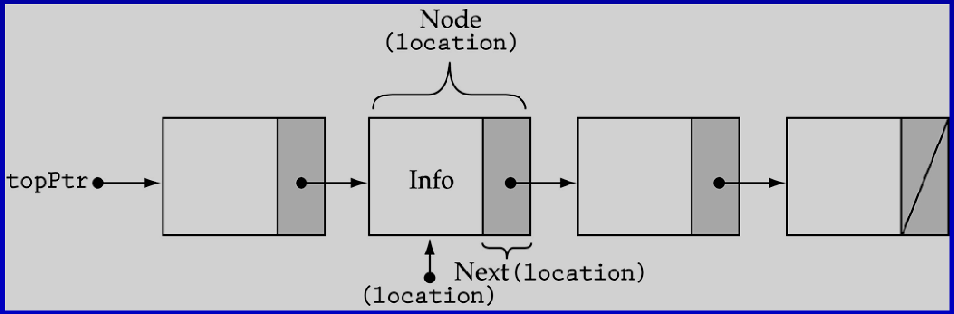
- What are the properties of the Stack Data Structure?
  - First In Last Out – FILO
    - Accessed from the top of the Stack only
- What functions do we have available?
  - `Push()`
    - items on to the Stack
  - `Pop()`
    - items off the Stack
  - Would we need any others?

Stack Implementations

Array-based



Linked-list-based



Stack Operations

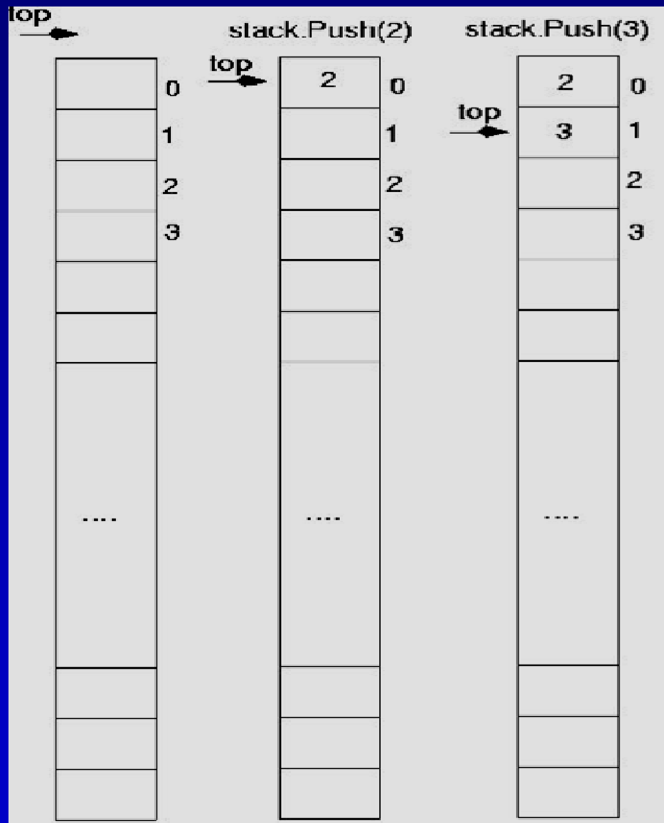
- Push(e): inserts e at the top of the stack
- Pop(0): removes the top element; error occurs when stack is empty
- Top(): returns a reference to the top element
- Size():return the number of elements in the stack
- Empty(): returns true if stack is empty

Example

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	"error"	()
top()	"error"	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

Array-based Stacks

```
template<class ItemType>
class StackType {
 public:
   StackType(int);
   void MakeEmpty();
   bool IsEmpty() const;
   bool IsFull() const;
   void Push(ItemType);
   void Pop(ItemType&);
 private:
    int top, maxStack;
    ItemType *items;
};
```



dynamically allocated array

Stack as an Array

Algorithm size():
  return $t+1$

Algorithm empty():
  return $(t<0)$

Algorithm top():
  if empty() **then**
    throw StackEmpty exception
  return $S[t]$

Algorithm push(e):
  if size() $=N$ **then**
    throw StackFull exception
  $t \leftarrow t+1$
  $S[t] \leftarrow e$

Algorithm pop():
  if empty() **then**
    throw StackEmpty exception
  $t \leftarrow t-1$

Array-based Stacks (cont'd)

```
template<class ItemType>
StackType<ItemType>::StackType(int size)
{
 top = -1;
 maxStack = size;
 items = new ItemType[maxStack];
}
```

O(1)

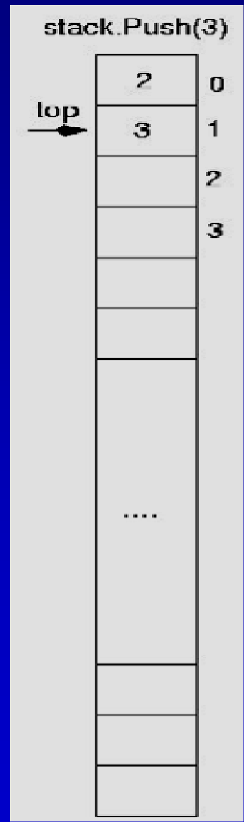
```
template<class ItemType>
StackType<ItemType>::~StackType()
{
 delete [ ] items;
}
```

O(1)

Array-based Stacks (cont’d)

template<class ItemType>
void StackType<ItemType>::**MakeEmpty**()
{
 top = -1;
}

O(1)



Array-based Stacks (cont.)

```
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
 return (top == -1);
}
```

O(1)

```
template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
 return (top == maxStack-1);
}
```

O(1)

Push (ItemType newItem)

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
   stack.Push(item);
```

Array-based Stacks (cont.)

```
template<class ItemType>
void StackType<ItemType>::Push(ItemType newItem)
{
 top++;
 items[top] = newItem;
}
```

O(1)

Pop (ItemType& item)

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
  stack.Pop(item);
```

Array-based Stacks (cont.)

```
template<class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
 item = items[top];
 top--;
}
```

O(1)

Templates

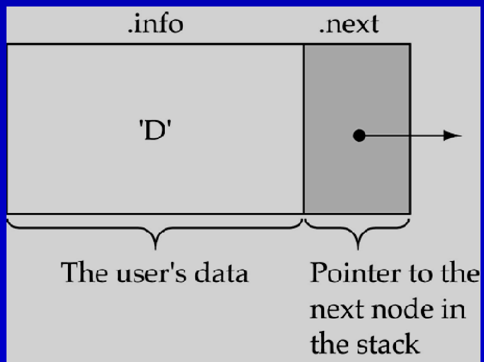
- Templates allow the compiler to generate multiple versions of a class type by allowing parameterized types.
- Compiler generates distinct class types and gives its own internal name to each of the types.

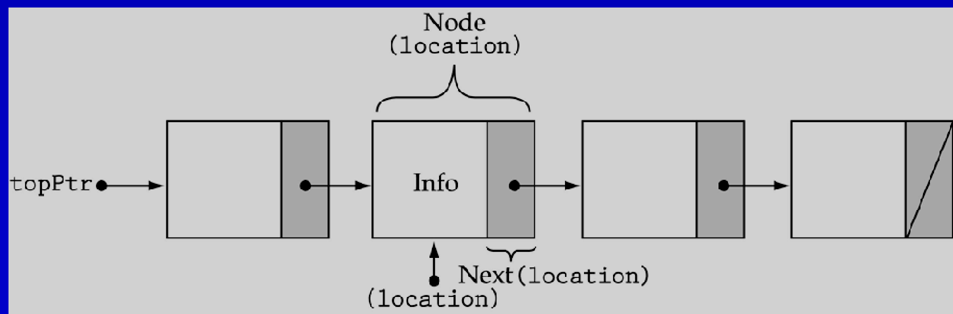
Linked Implementation of Stacks

- Because an array size is fixed
  - Only a fixed number of elements can be pushed onto the stack.
- To avoid pushing onto a full stack use linked list.
- Top is now memory address rather than index.

Linked-list-based Stacks

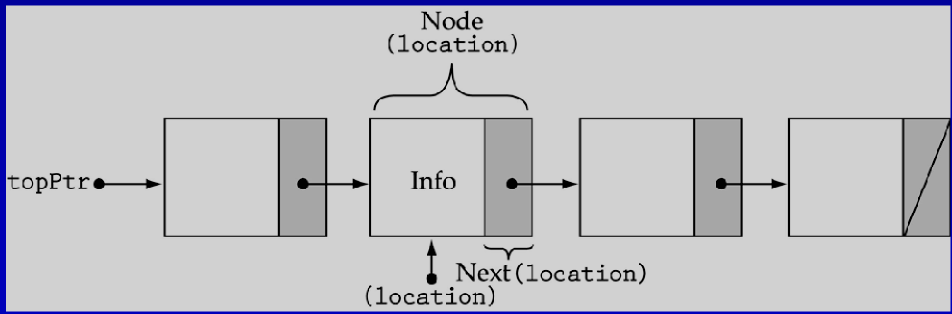
```
template<class ItemType>
struct NodeType<ItemType> {
  ItemType info;
  NodeType<ItemType>* next;
};
```





Linked-list-based Stacks (cont'd)

```
template<class ItemType>
struct NodeType<ItemType>;


template<class ItemType>
class StackType {
 public:
   StackType();
   ~StackType();
   void MakeEmpty();
   bool IsEmpty() const;
   bool IsFull() const;
   void Push(ItemType);
   void Pop(ItemType&);
 private:
   NodeType<ItemType>* topPtr;
};
```



Linked-list-based Stacks (cont'd)

```
template<class ItemType>
StackType<ItemType>::StackType()
{
 topPtr = NULL;
}
```

O(1)

```
template<class ItemType>
void StackType<ItemType>::MakeEmpty()
{
 NodeType<ItemType>* tempPtr;

 while(topPtr != NULL) {
   tempPtr = topPtr;
   topPtr = topPtr->next;
   delete tempPtr;
 }
}
```

O(N)

Linked-list-based Stacks (cont'd)

```
template<class ItemType>
StackType<ItemType>::~StackType()
{
 MakeEmpty();
}
```

O(N)

```
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const
{
 return(topPtr == NULL);
}
```

O(1)

Linked-list-based Stacks (cont'd)

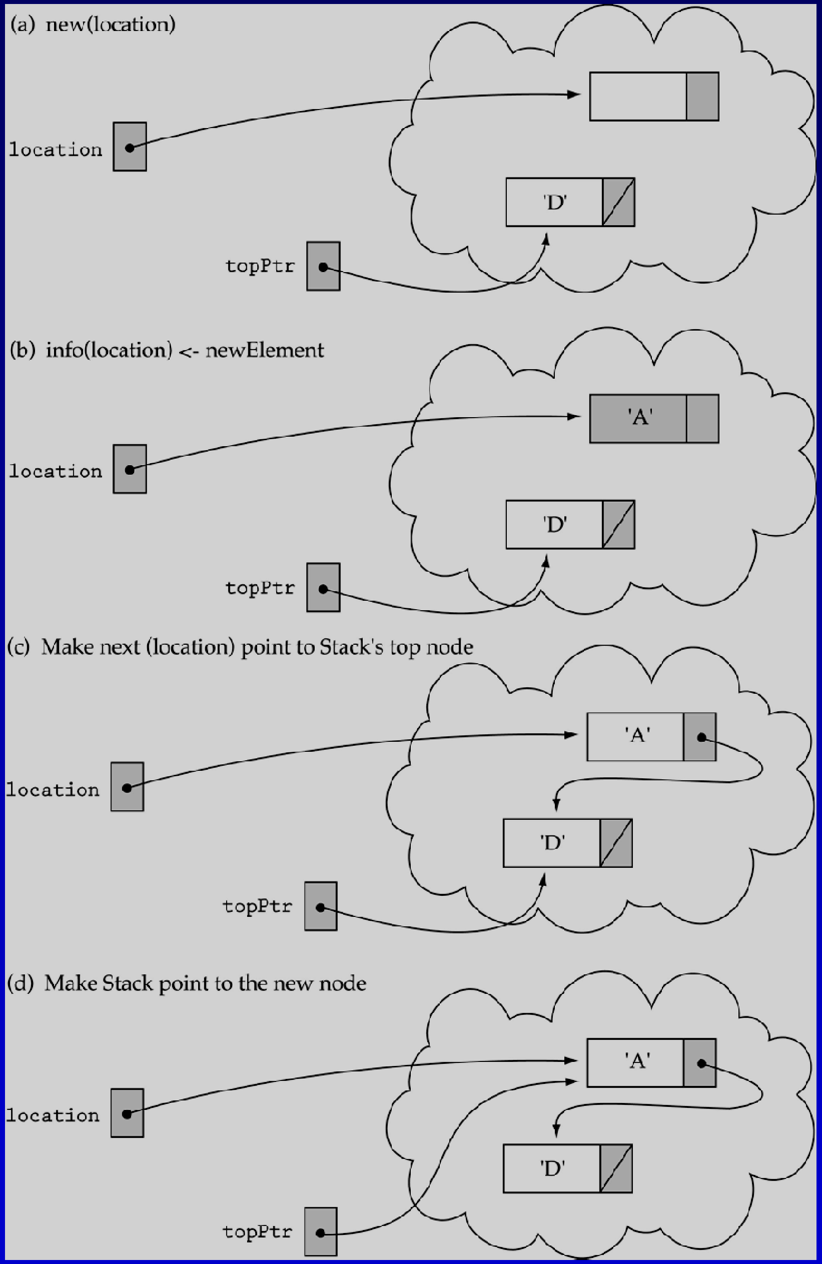
```
template<class ItemType>
bool StackType<ItemType>::IsFull() const
{
 NodeType<ItemType>* location;

 location = new NodeType<ItemType>; // test
 if(location == NULL)
   return true;
 else {
   delete location;
   return false;
 }
}
```

O(1)

Push (ItemType newItem)

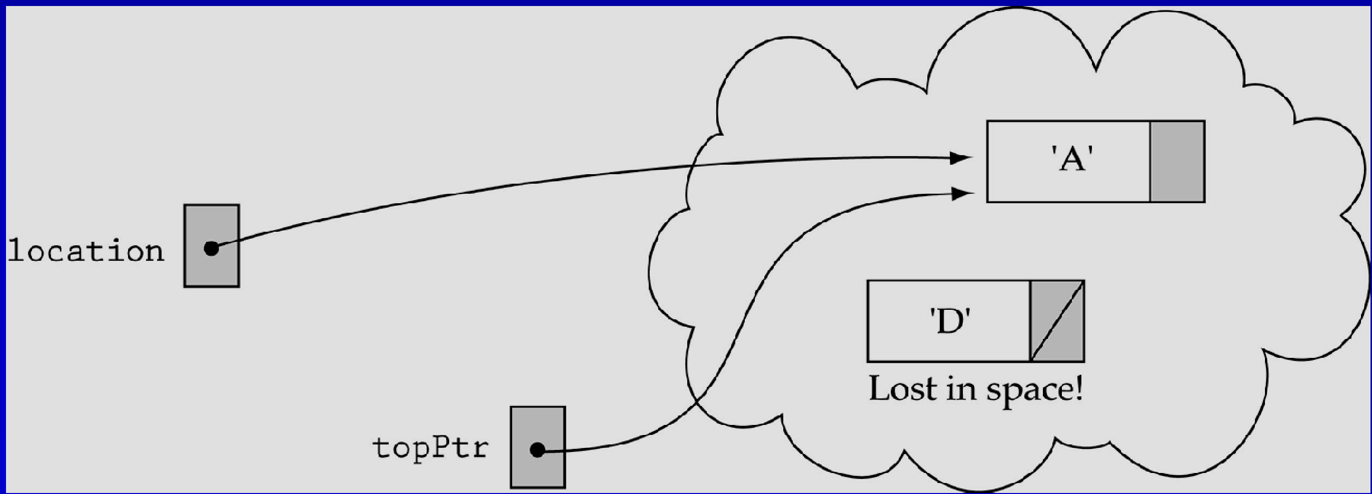
- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Pushing on a non-empty stack

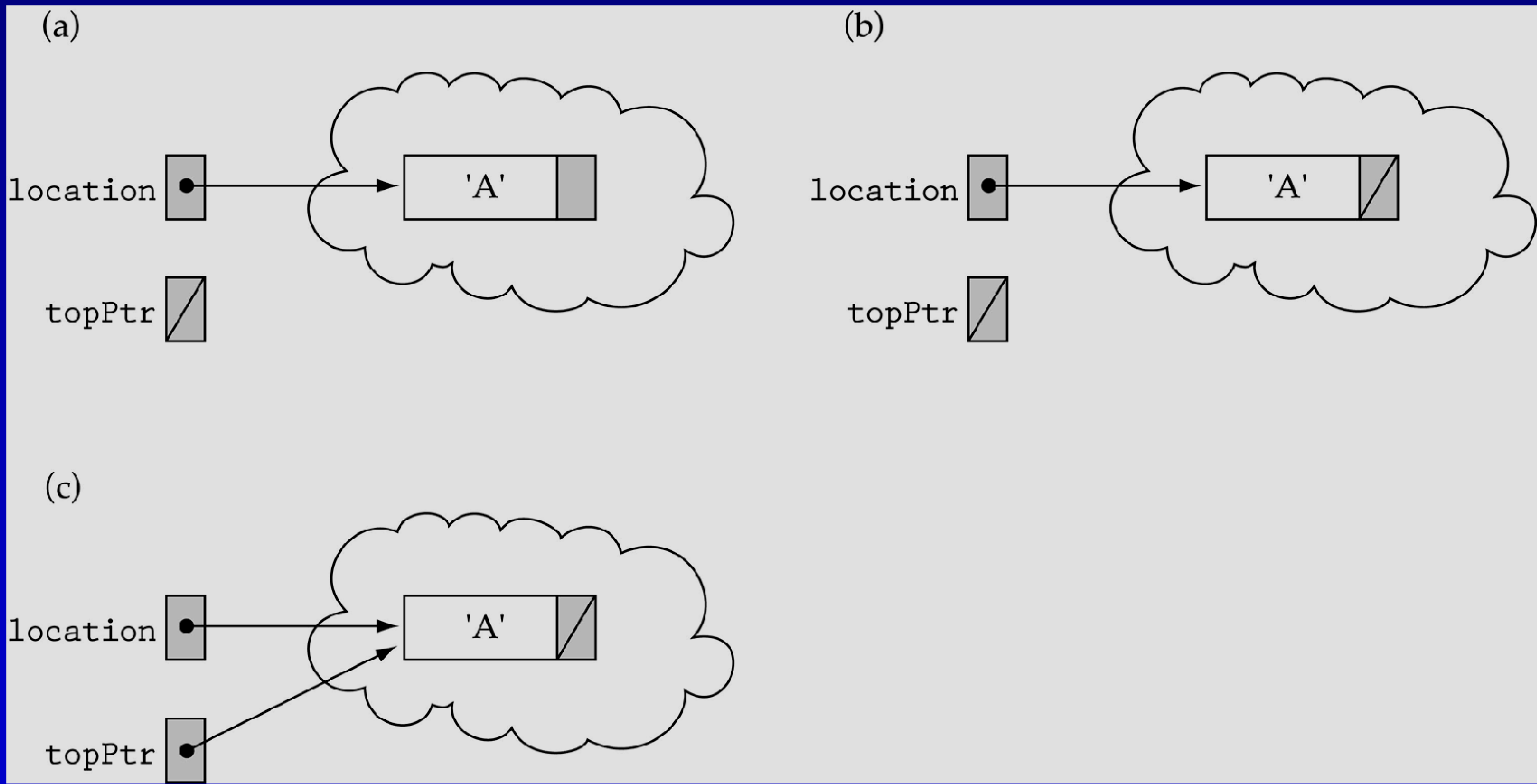


Pushing on a non-empty stack (cont.)

- The order of changing the pointers is important!



Special Case: pushing on an empty stack



Function Push

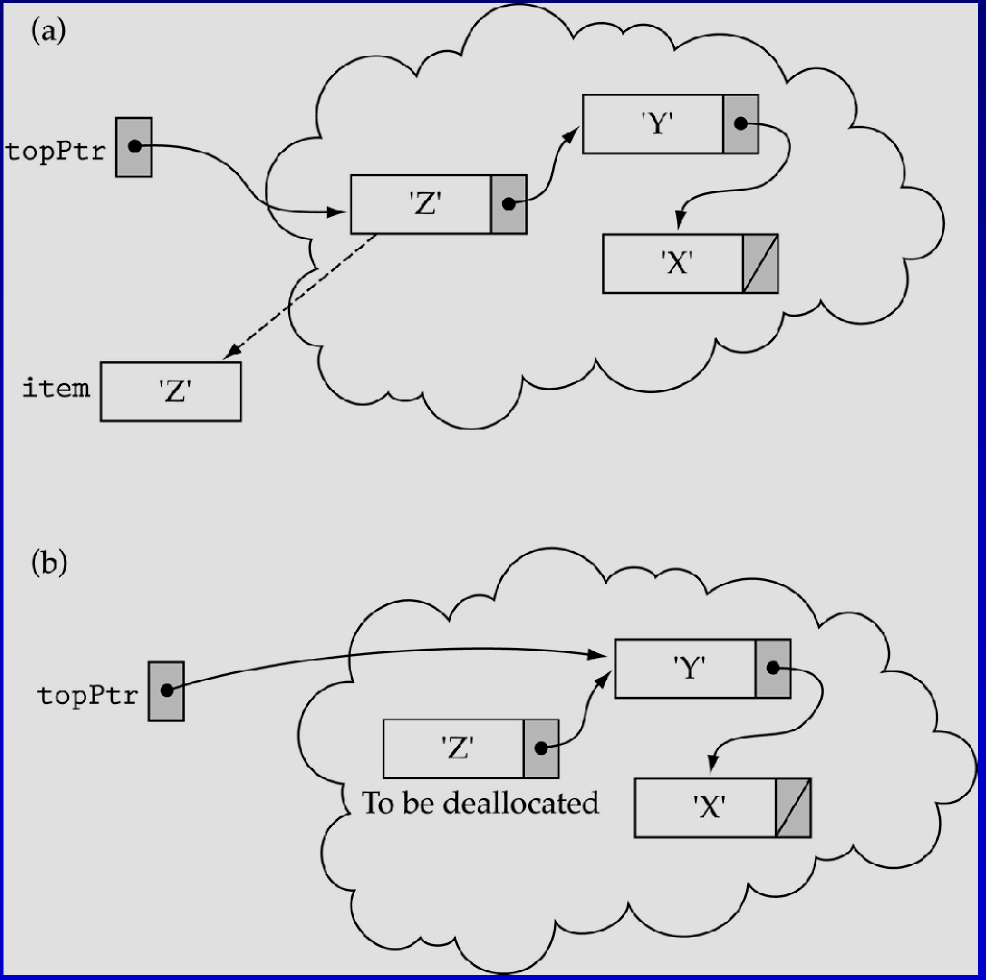
```
template <class ItemType>
void StackType<ItemType>::Push(ItemType
   item)
{
 NodeType<ItemType>* location;

 location = new NodeType<ItemType>;
 location->info = newItem;
 location->next = topPtr;
 topPtr = location;
}
```

O(1)

Pop (ItemType& item)

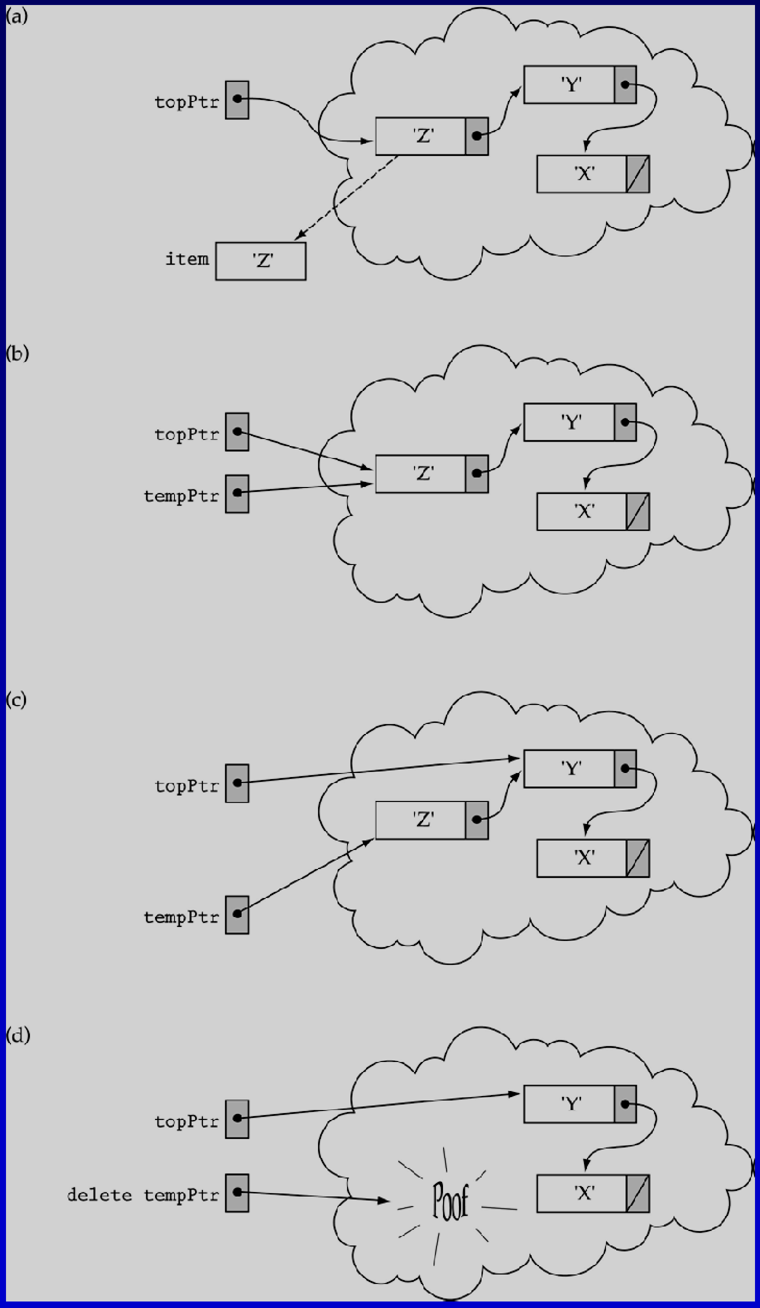
- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.

Popping the top element

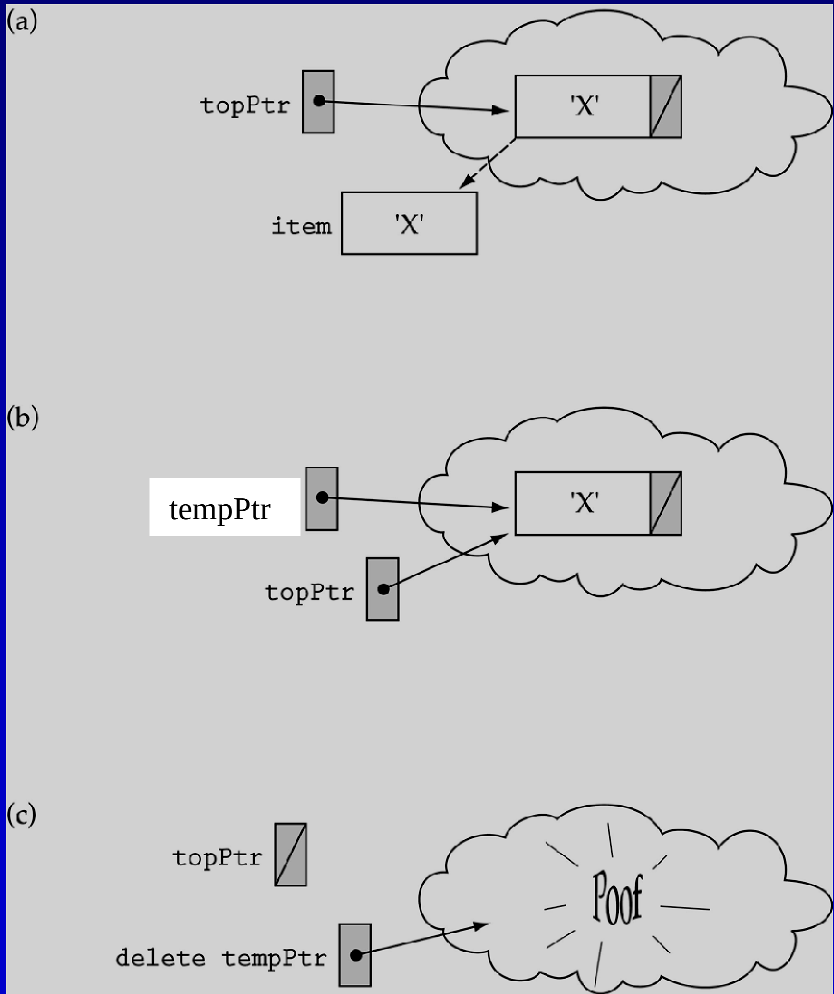


Popping the top element (cont.)

Need a temporary pointer !



Special case: popping the last element on the stack



Function Pop

```
template <class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
 NodeType<ItemType>* tempPtr;

 item = topPtr->info;
 tempPtr = topPtr;
 topPtr = topPtr->next;
 delete tempPtr;
}
```

O(1)

Comparing stack implementations

Big-O Comparison of Stack Operations		
Operation	Array Implementation	Linked Implementation
Constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Destructor	O(1)	O(N)

Array-vs Linked-list-based Stack Implementations

- Array-based implementation is simple but:
  - The size of the stack must be determined when a stack object is declared.
  - Space is wasted if we use less elements.
  - We cannot "enqueue" more elements than the array can hold.
- Linked-list-based implementation alleviates these problems but time requirements might increase.

Example using stacks: evaluate postfix expressions

- Usual arithmetic expressions are *infix* notation.
- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

(1920, Polish Mathematician) (1950, Australian philosopher)

prefix: +2 5 *infix:* 2+5 *postfix:* 2 5 +

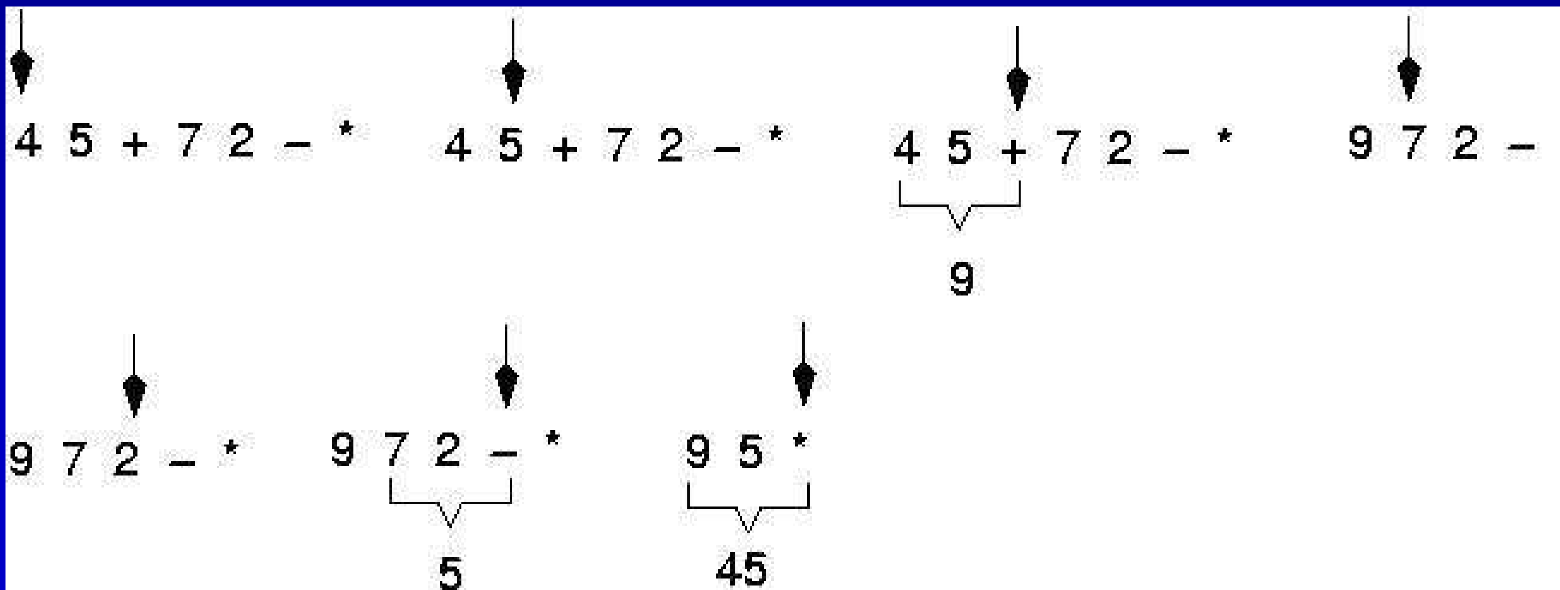
(Jan Lukasiewicz, Polish notation) (Charles Hamblin, Reverse Polish)

- Why using postfix notation?

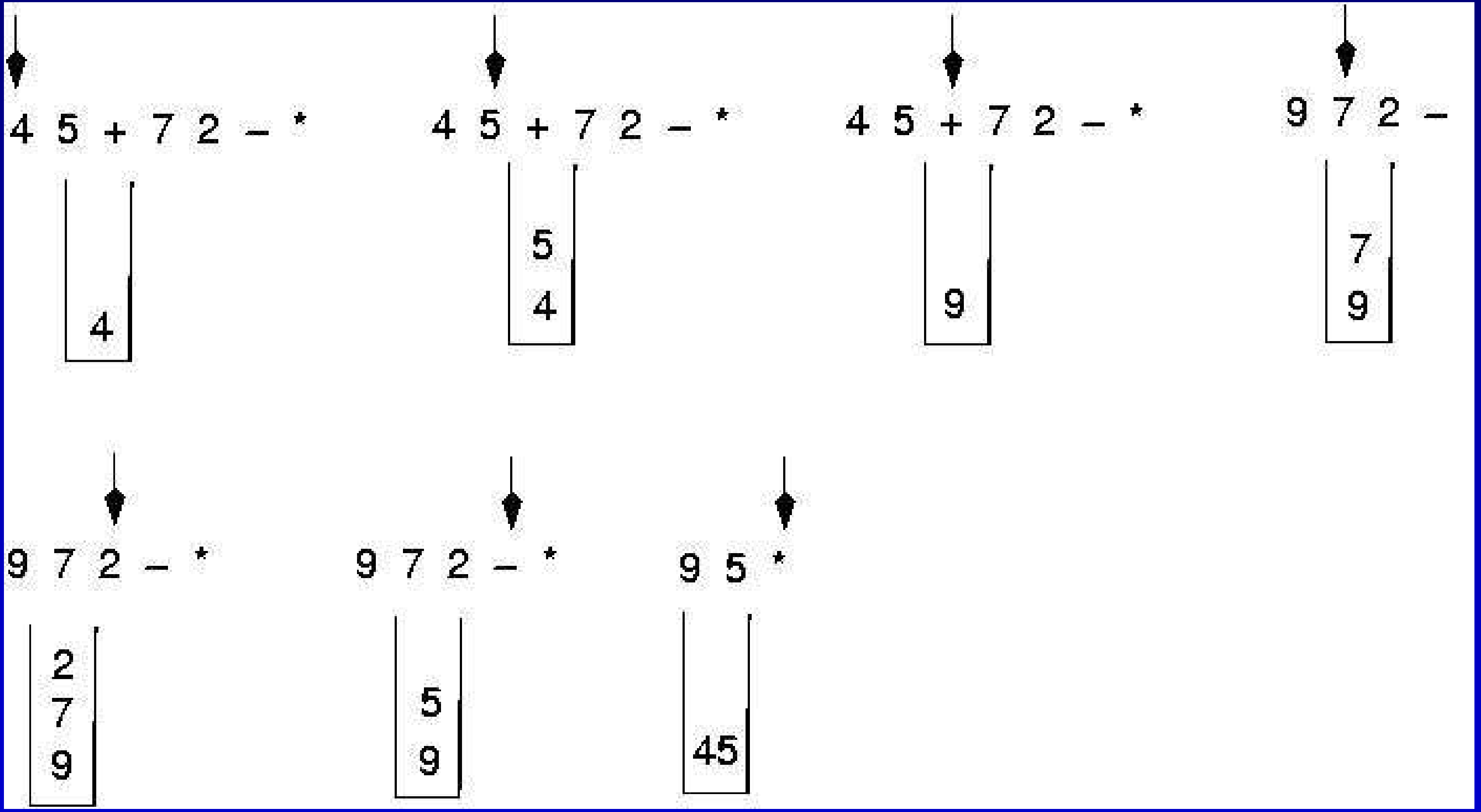
Precedence rules and parentheses are not required!

Example: postfix expressions (cont.)

Expressions are evaluated from left to right.



Postfix expressions: Algorithm using stacks (cont.)



Example

Infix expression	Equivalent postfix expression
$a + b$	
$a + b * c$	
$a * b + c$	
$(a + b) * c$	
$(a - b) * (c + d)$	
$(a + b) * (c - d / e) + f$	

Example

Infix expression	Equivalent postfix expression
$a + b$	$a\ b\ +$
$a + b * c$	$a\ b\ c\ *\ +$
$a * b + c$	$a\ b\ *\ c\ +$
$(a + b) * c$	$a\ b\ +\ c\ *$
$(a - b) * (c + d)$	$a\ b\ -\ c\ d\ +\ *$
$(a + b) * (c - d / e) + f$	$a\ b\ +\ c\ d\ e\ /\ -\ *\ f\ +$

Exercise: Consider the following postfix expression:

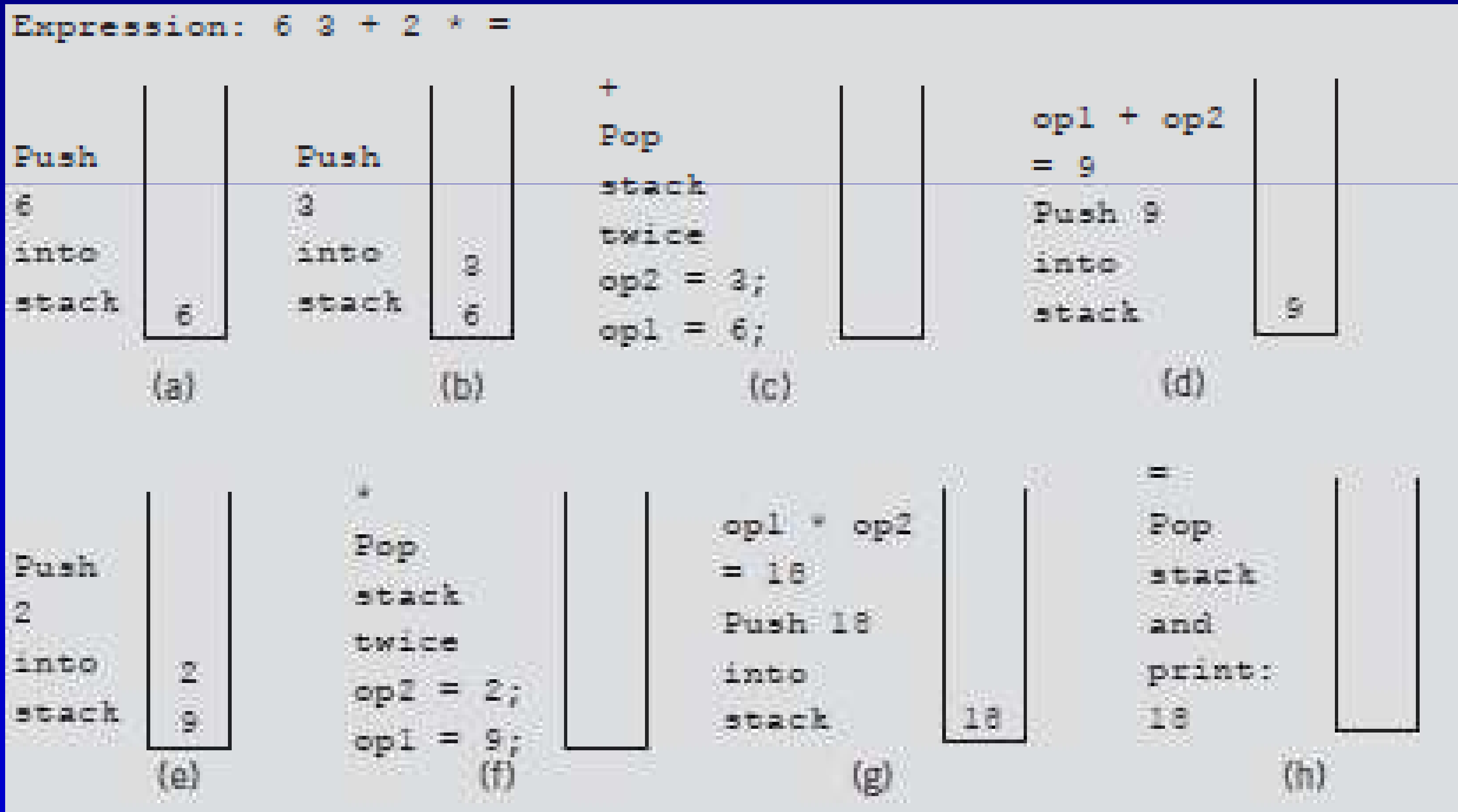
6 3 + 2 * =

Evaluate this expression using a stack

Exercise: Consider the following postfix expression:

6 3 + 2 * =

Evaluate this expression using a stack



More Array Uses

- In a Stack, first in must wait longest....
  - What happened to first come, first served?
- The Queue Structure
  - FIFO – First In First Out
  - Queue ADT Definition:
  - The queue defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the sequence and new elements are added to the back