Stacks

CPE 212 -- Lecture 09

Outline

- Data Structures
- Stack ADT
  - Concepts
  - Implementations
- Summary

Data Structures

- A ***Stack*** is an example of a data structure – a virtual container carved from memory
- Containers have different shapes and access rules which impact the efficiency of operations such as insert, delete, and find
- **Fundamental tradeoff**
  - **Efficiency of container operations**
  - **Memory consumed by container**

Stack ADT - Basic Concepts

- Special type of list
  - All insertions and deletions are ***only from the top of the stack***
  - Last item added is the first item removed
    - ***LIFO***: Last-In, First-Out
- Stack analogy
  - Pile of books

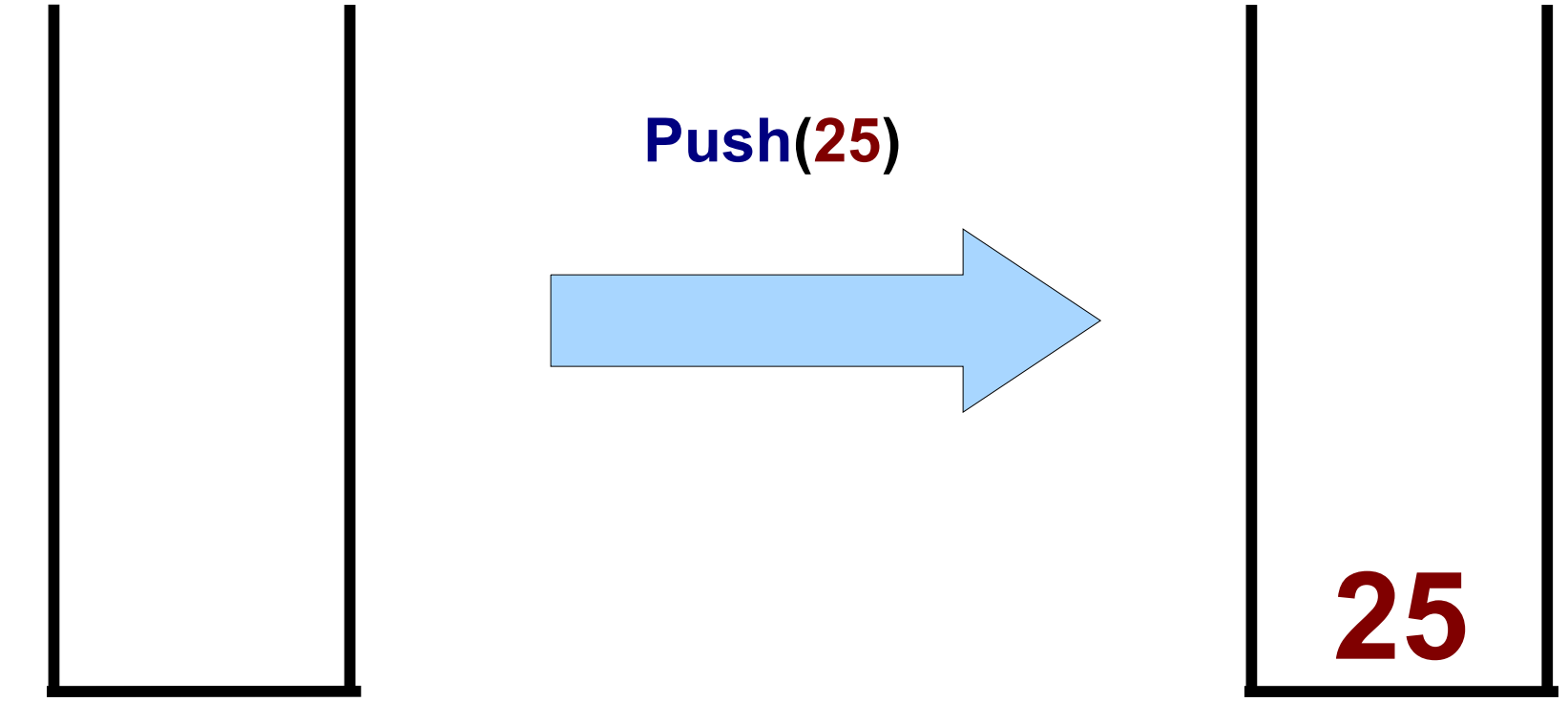
Stack ADT - Applications

- **Compilers**
  - Parsing nested structures within code
- **Operating System**
  - Function activation records track variable values for currently active functions
- **Text Editor**
  - Process a line of text as a stack
- **Text Reversal**

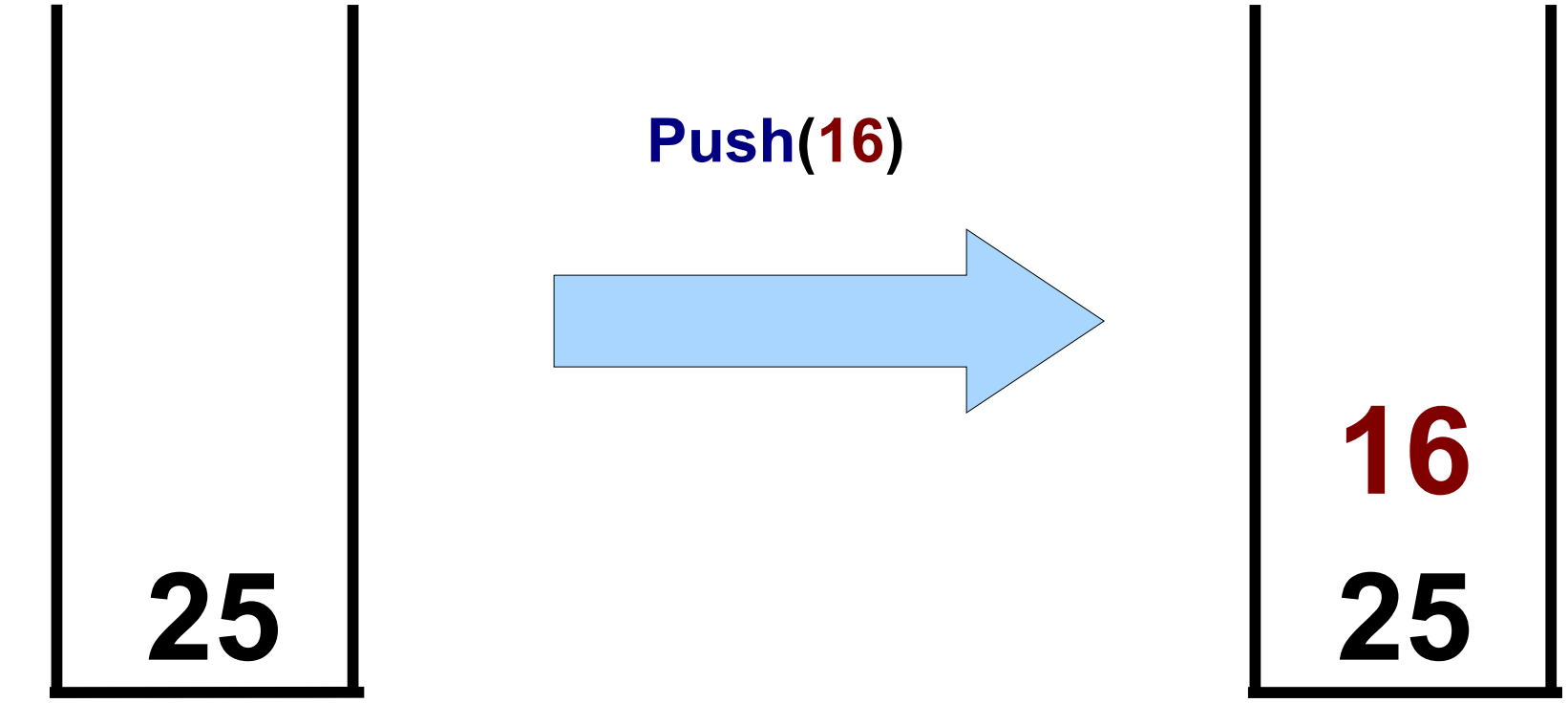
Stack ADT - Basic Operations

- **Push**
  - adds new item to top of stack
- **Pop**
  - removes item from top of stack
- **Top**
  - returns a copy of the top item on the stack
- **IsEmpty**
  - determines whether stack is empty
- **IsFull**
  - determines whether stack is full
- **MakeEmpty**
  - empties the stack

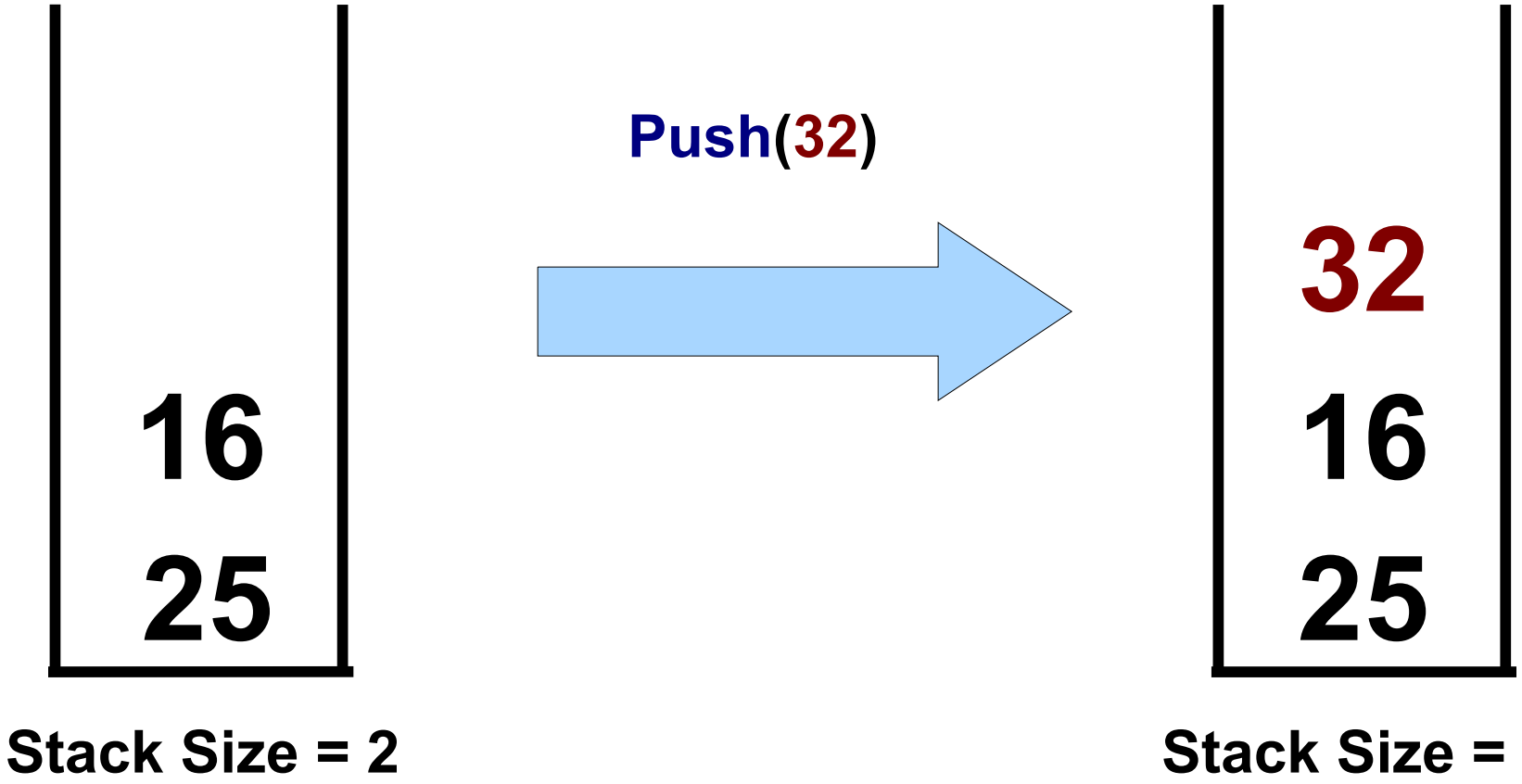
Push Operation - 1



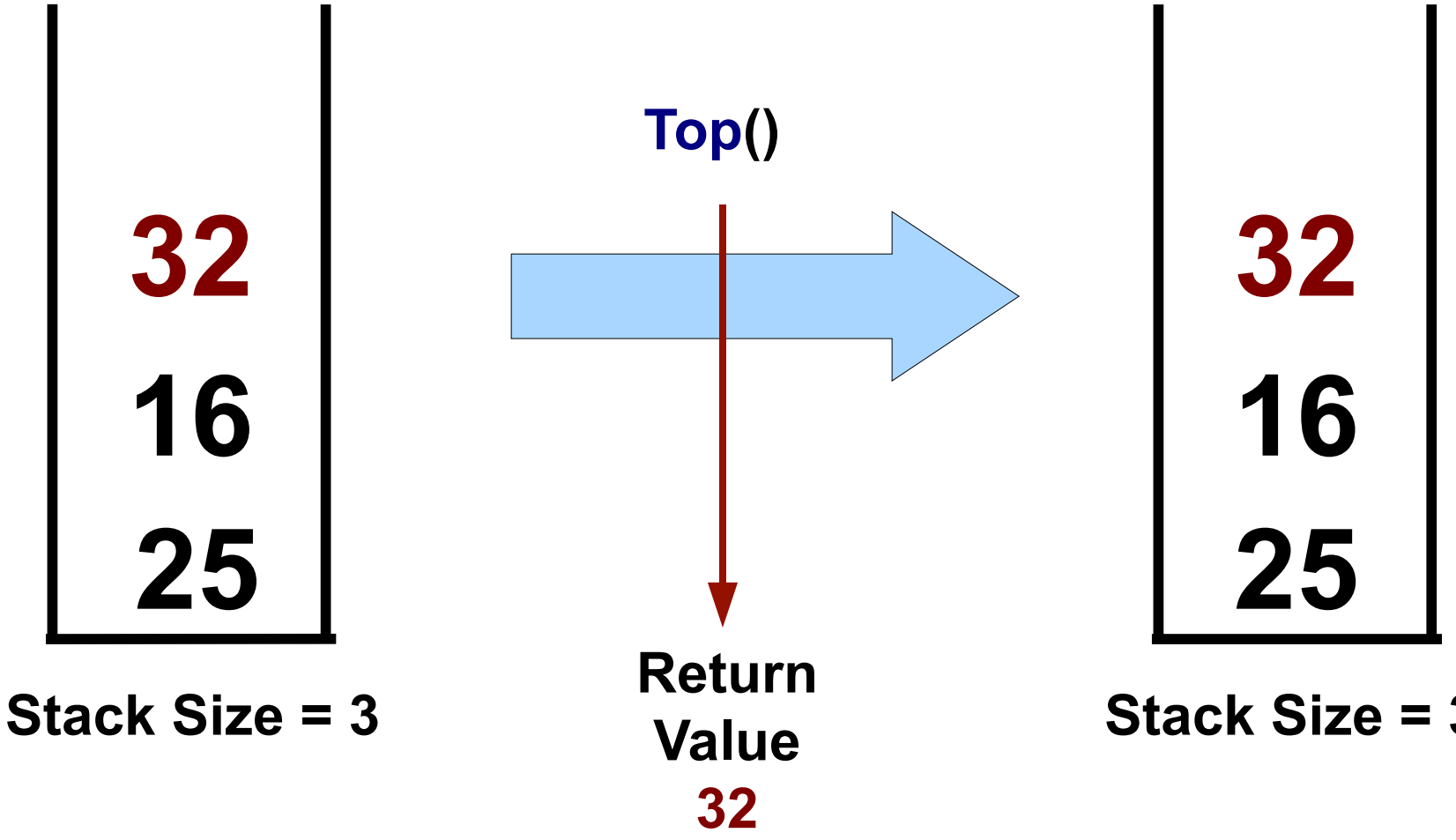
Push Operation - 2



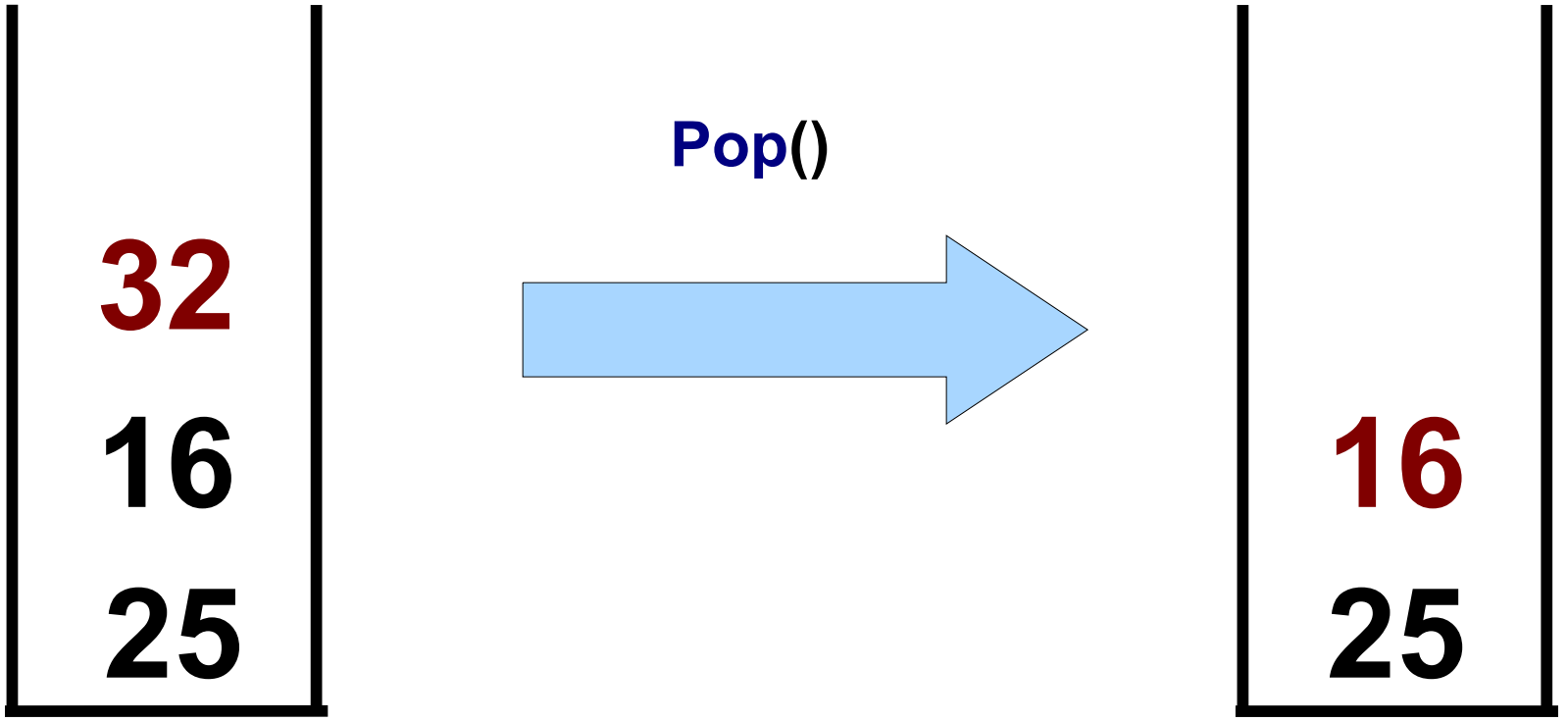
Push Operation - 3



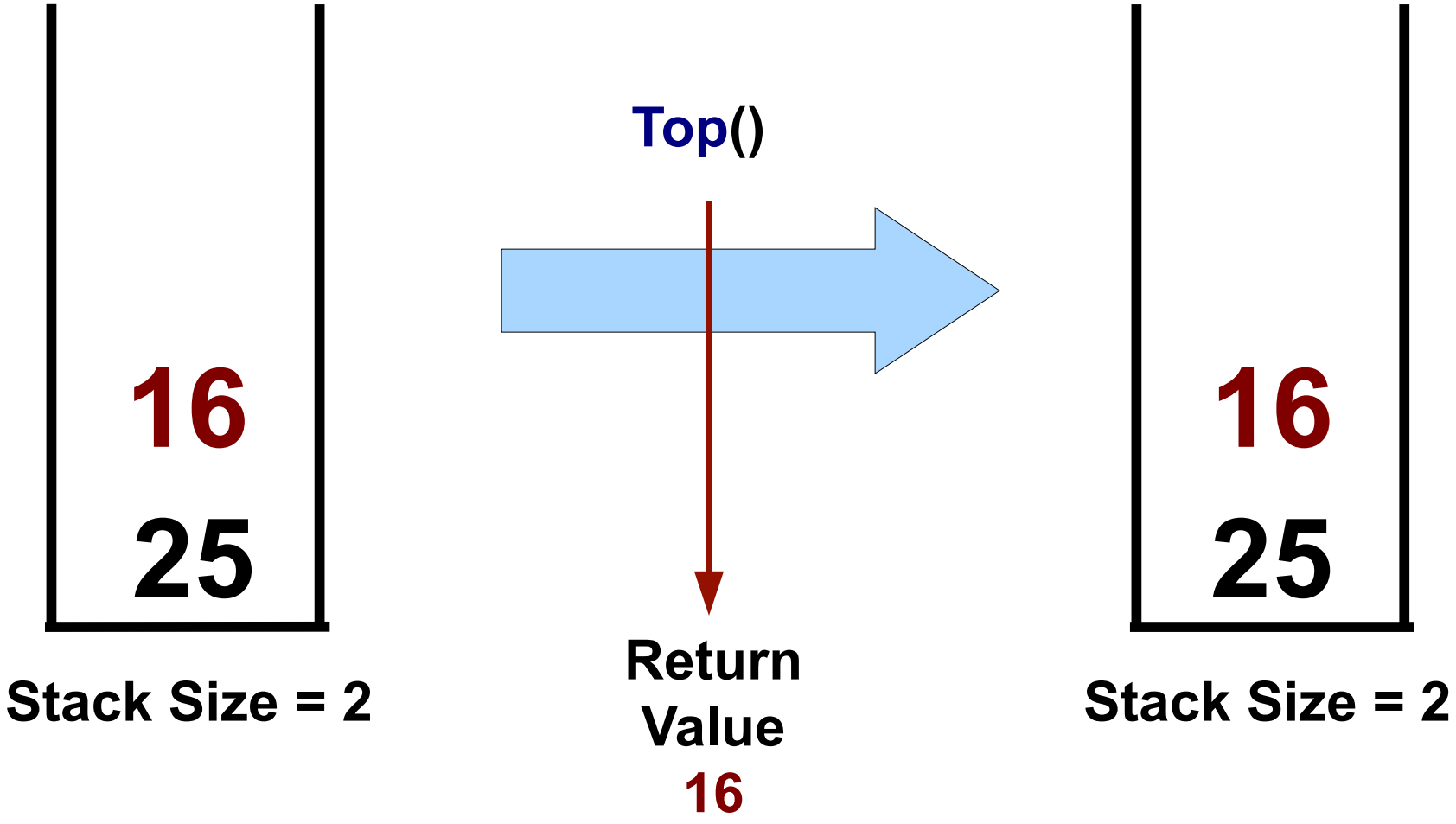
Top Operation - 1



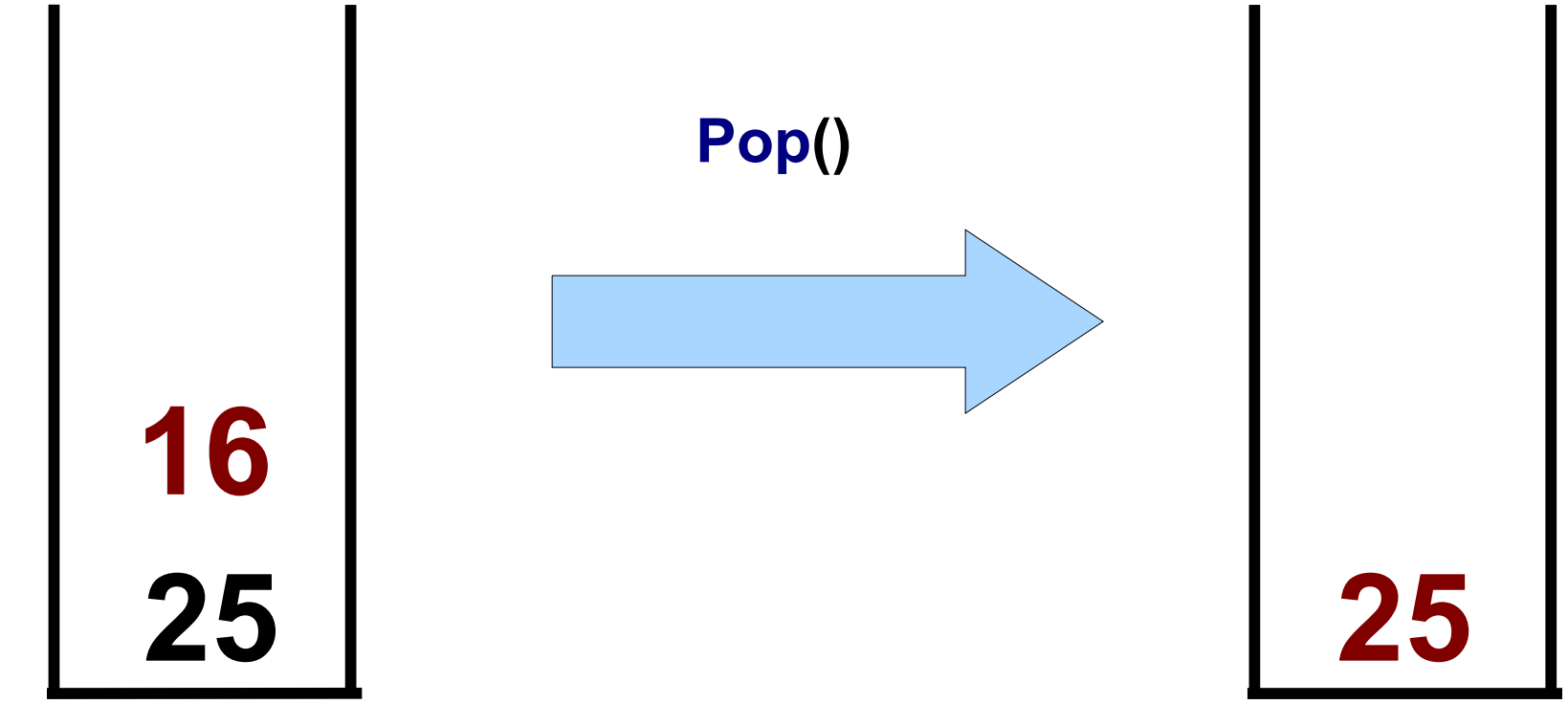
Pop Operation - 1



Top Operation - 2



Pop Operation - 2



Limits of Push, Pop, Top - 1

- Any implementation of the **Stack** container has a finite limit on the number of elements that can be stored
  - What happens if **Push** is called when the stack is already full?

Limits of Push, Pop, Top - 2

- Even when there is room to store values on the **Stack**, there may be no values stored in the container at the time when a particular operation is invoked
  - What happens if **Pop** or **Top** is called when the stack is empty?

Limits of Push, Pop, Top - 3

- Two options for error handling

  Option #1 – Client is responsible

  – Client code must contain **defensive code** to detect and avoid these situations in which the operations of the container class are undefined

  Option #2 – Container is responsible

  – Container class must contain **defensive code** that signals client code if these situations occur so that client can take appropriate action

Option #1
Client is responsible

```
// Somewhere in the client code...

Stack  s;                 // Create a stack called s

if  ( !s.IsEmpty() )      // If stack s is not empty
{
    s.Pop();              //  then remove top value from stack s

    /* Do something useful here */
}
else                      // ...else s is empty so...
{
    /* Process this error condition */
}
```

Option #2
Container is responsible

Within the Pop() function

```
if   ( IsEmpty() )       // If stack is empty
{
    /* ...then throw exception to signal client code */
}
else
{
    /* ...otherwise, remove top item from stack as requested */
}
```

Stack ADT Implementation

Sequential Array Implementation

UAHuntsville

```
//*******  stack.h Standard Header Information Here **********

const int MAX_SIZE = 100;                // Maximum stack size

typedef int ItemType;                    // Data type of each item on stack

class Stack                              // Array-based Stack class
{
 private:
  ItemType data[MAX_SIZE];               // Head of linked list
  int top;                               // Top of stack indicator

 public:
  Stack();                               // Default constructor
                                         // Postcondition: Empty stack created

  bool IsEmpty() const;                  // Checks to see if stack is empty
                                         // Postcondition: Returns TRUE if empty, FALSE otherwise

  bool IsFull() const;                   // Checks to see if stack is full
                                         // Postcondition: Returns TRUE if full, FALSE otherwise

  void Push(ItemType item);              // Adds item to top of stack

  void Pop();                            // Removes top item from stack

  ItemType Top() const;                  // Returns a copy of top item on stack
                                         // Postcondition: item still on stack, copy returned

  void MakeEmpty();                      // Removes all items from stack
};
```

```
//*******  stack.cpp Standard Header Information Here **********
#include "stack.h"

Stack::Stack()                          // Default constructor
{                                       // Postcondition: Empty stack created

}

bool Stack::IsEmpty() const             // Checks to see if stack is empty
{                                       // Postcondition: Returns TRUE if empty, FALSE otherwise

}

bool Stack::IsFull() const              // Checks to see if stack is full
{                                       // Postcondition: Returns TRUE if full, FALSE otherwise

}

void Stack::Push(ItemType item)         // Adds item to top of stack
{                                       // Precondition: stack is not full


}

void Stack::Pop()                       // Removes top item from stack
{                                       // Precondition: stack is not empty

}

ItemType Stack::Top() const             // Returns a copy of top item on stack
{                                       // Precondition: stack is not empty
                                        // Postcondition: item still on stack, copy returned
}

void Stack::MakeEmpty()                 // Removes all items from stack
{

}
```

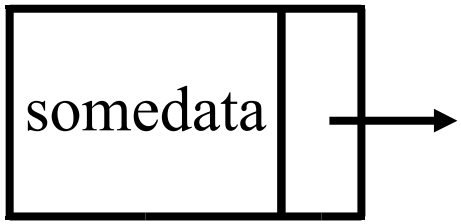
```
//*******  stack.cpp Standard Header Information Here **********
#include "stack.h"

Stack::Stack()                          // Default constructor
{                                       // Postcondition: Empty stack created
  top = -1;
}

bool Stack::IsEmpty() const             // Checks to see if stack is empty
{                                       // Postcondition: Returns TRUE if empty, FALSE otherwise
  return (top == -1);
}

bool Stack::IsFull() const              // Checks to see if stack is full
{                                       // Postcondition: Returns TRUE if full, FALSE otherwise
  return (top == (MAX_SIZE-1));
}

void Stack::Push(ItemType item)         // Adds item to top of stack
{                                       // Precondition: stack is not full
  top++;
  data[top] = item;
}

void Stack::Pop()                       // Removes top item from stack
{                                       // Precondition: stack is not empty
  top--;
}

ItemType Stack::Top() const             // Returns a copy of top item on stack
{                                       // Precondition: stack is not empty
  return data[top];                     // Postcondition: item still on stack, copy returned
}

void Stack::MakeEmpty()                 // Removes all items from stack
{
  top = -1;
}
```

```
//*******  stackclient.cpp Standard Header Information Here **********
#include <iostream>
#include <fstream>
#include "stack.h"

using namespace std;

int main()
{
  Stack temps;
  ifstream datafile;
  ItemType someTemp;

  datafile.open("June05Temps");

  datafile >> someTemp;
  while (datafile)
  {
    if ( !temps.IsFull() )
    {
      temps.push(someTemp);
    }
    datafile >> someTemp;
  }

  while ( !temps.IsEmpty() )
  {
    cout << temps.Top() << endl;
    temps.Pop();
  }

  return 0;
}
```

Stack ADT Implementation

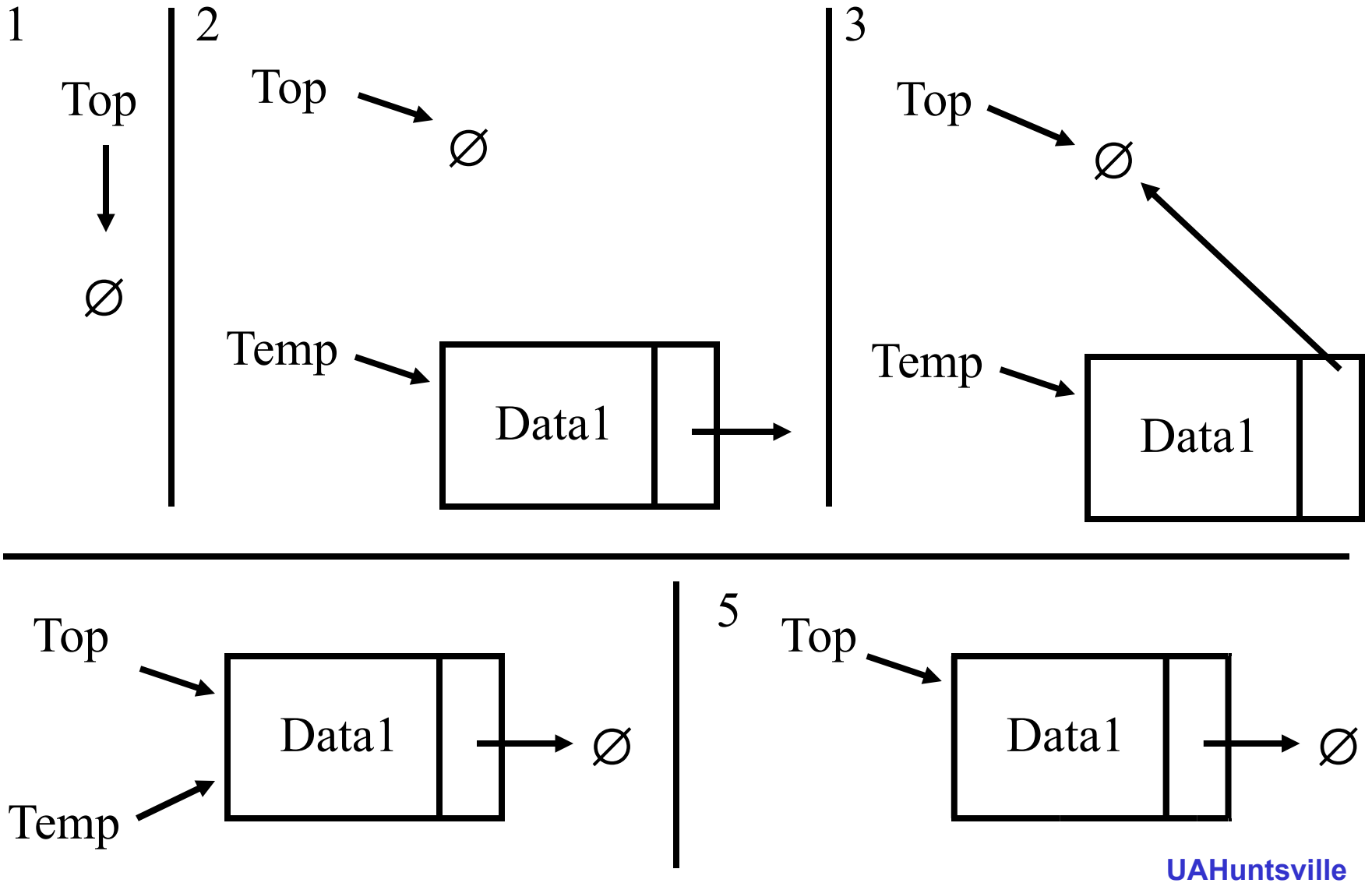
Linked List of
Dynamically-Allocated Nodes

Node Structure

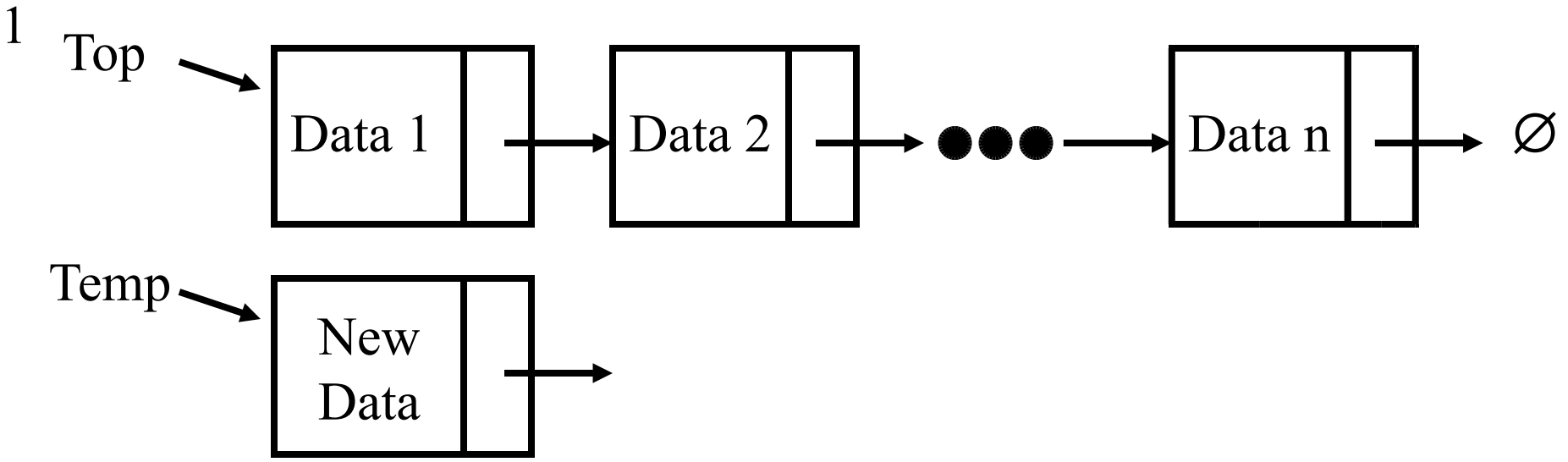


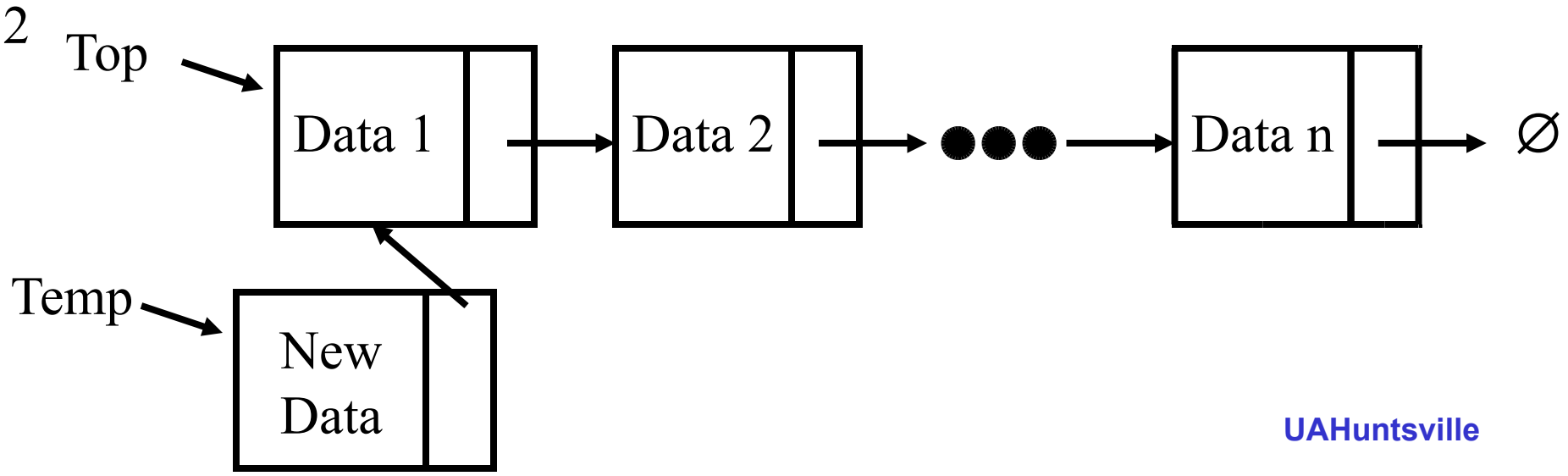

```
struct  NodeType
{
   ItemType        info;
   NodeType*       next;
};
```

Push - Onto Empty Stack

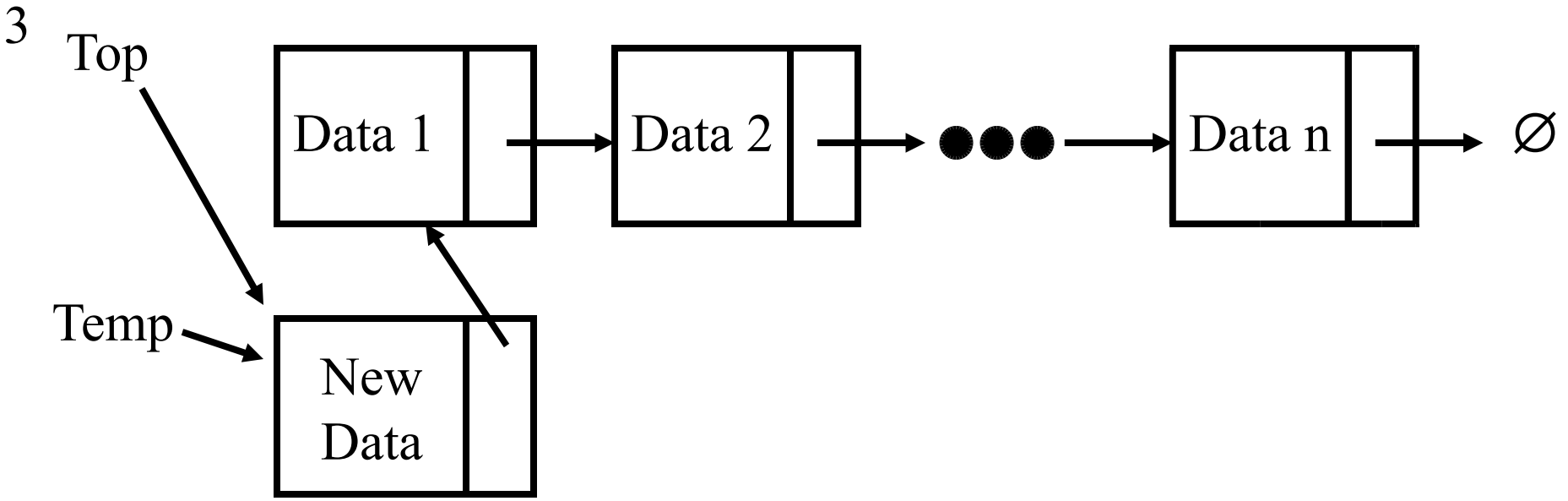


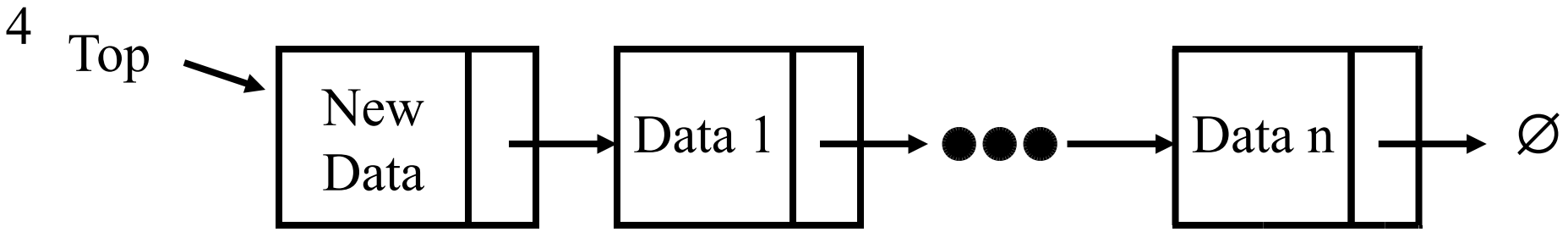
Push - Onto Populated Stack - 1



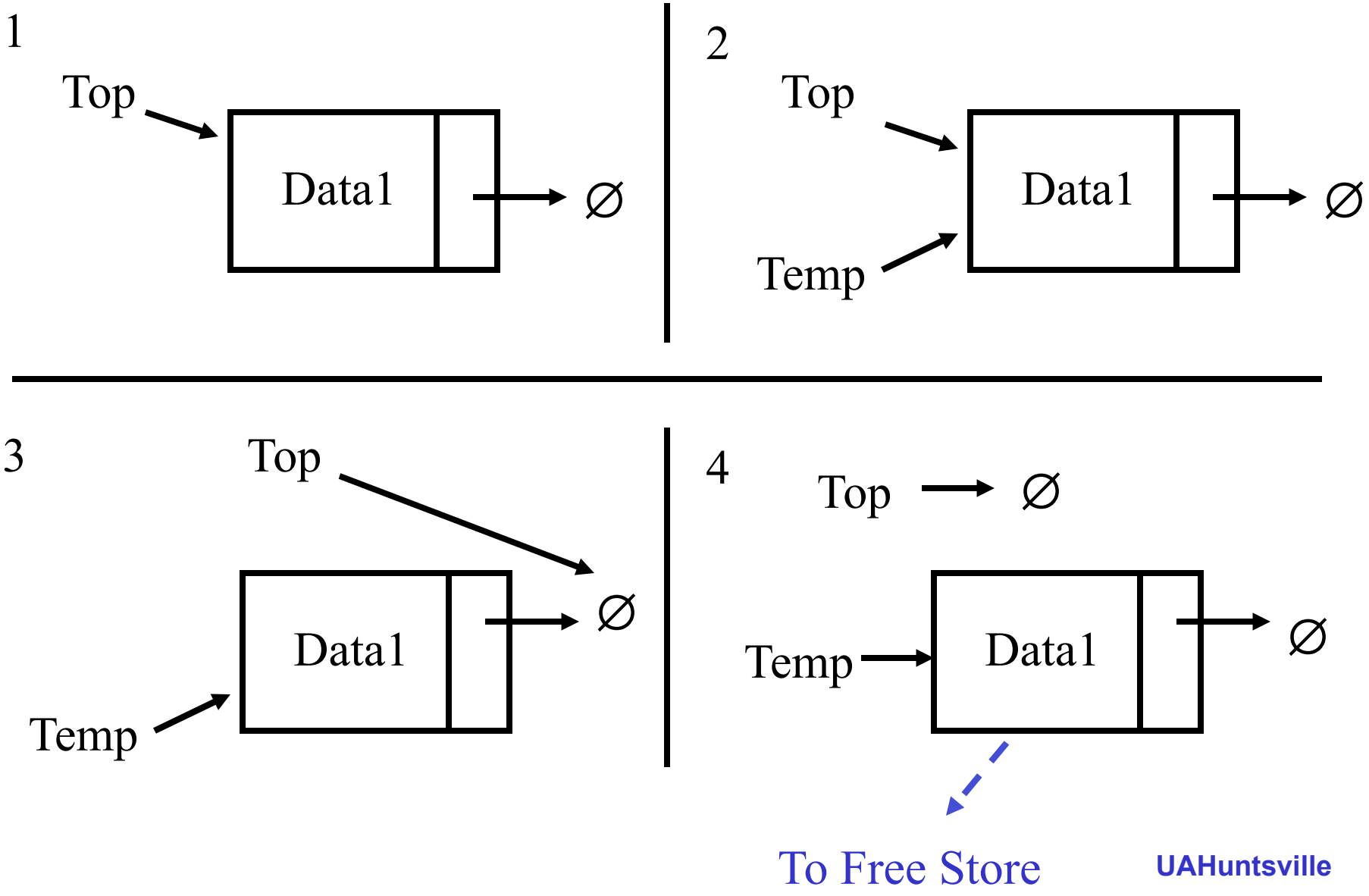


Push - Onto Populated Stack - 2

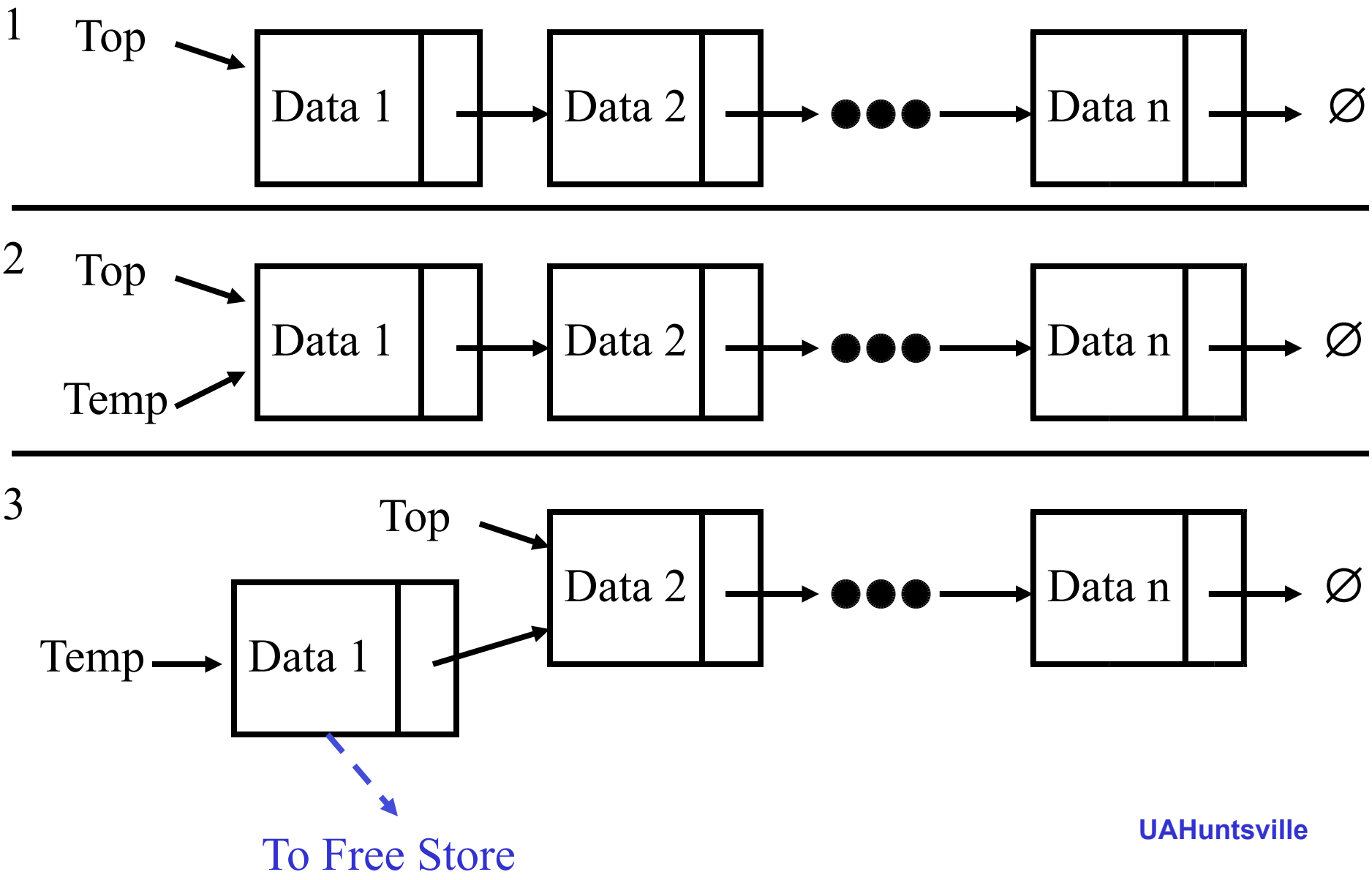




Pop - Leaving Empty Stack



Pop - Leaving Populated Stack



```
//*******  stack.h Standard Header Information Here **********
#ifndef STACK_H
#define STACK_H
typedef int ItemType;                    // Data type of each item on stack

struct NodeType                                          // Declaration of the node stucture
{
  ItemType info;                                         // Field storing the data
  NodeType* next;                                        // Field storing address of next
node in sequence
};

class Stack                              // Linked Node-based Stack class
{
 private:
  NodeType* topPtr;                      // Top of stack pointer

 public:
  Stack();                               // Default constructor creates an empty stack

  bool IsEmpty() const;                  // Returns TRUE if empty, FALSE otherwise

  bool IsFull() const;                   // Returns TRUE if full, FALSE otherwise

  void Push(ItemType item);              // Adds item to top of stack

  void Pop();                            // Removes top item from stack

  ItemType Top() const;                  // Returns copy of top item on stack assuming it exists

  void MakeEmpty();                      // Returns stack to empty state

  ~Stack();                              // Destructor deallocates any nodes
};

#endif
```

```
//*******  stack.cpp Standard Header Information Here **********
#include <cstddef>
#include <new>
#include "stack.h"
using namespace std;
//*****************************************************************

Stack::Stack()                          // Default constructor
{                                       // Postcondition: Empty stack created

}

//*****************************************************************

bool Stack::IsEmpty() const             // Checks to see if stack is empty
{                                       // Postcondition: Returns TRUE if empty, FALSE otherwise

}

//*****************************************************************

bool Stack::IsFull() const              // Returns true if there is no room for another ItemType
{                                       //  on the free store; false otherwise.









}
```

```
//*******  stack.cpp Standard Header Information Here **********
#include <cstddef>
#include <new>
#include "stack.h"
using namespace std;
//*************************************************************

Stack::Stack()                          // Default constructor
{                                       // Postcondition: Empty stack created
  topPtr = NULL;
}

//*************************************************************

bool Stack::IsEmpty() const             // Checks to see if stack is empty
{                                       // Postcondition: Returns TRUE if empty, FALSE otherwise
  return (topPtr == NULL);
}

//*************************************************************

bool Stack::IsFull() const              // Returns true if there is no room for another ItemType
{                                       //  on the free store; false otherwise.
  NodeType* location;
  try
  {
    location = new NodeType;            // new raises an exception if no memory is available
    delete location;
    return false;
  }
  catch(std::bad_alloc)                 // This catch block processes the bad_alloc exception
  {                                     // should it occur
    return true;
  }
}
```

```
//*******  stack.cpp continued above **********

void Stack::Push(ItemType item)          // Adds item to top of stack
{                                        // Precondition: stack is not full







}

//************************************************************

void Stack::Pop()                        // Removes top item from stack
{                                        // Precondition: stack is not empty







}

//************************************************************

ItemType Stack::Top() const              // Returns a copy of top item on stack
{                                        // Precondition: stack is not empty
                                         // Postcondition: item still on stack, copy returned


}

//************************************************************
```

```
//*******  stack.cpp continued above **********

void Stack::Push(ItemType item)          // Adds item to top of stack
{                                        // Precondition: stack is not full
  NodeType* tempPtr = new NodeType;
  tempPtr->info = item;
  tempPtr->next = topPtr;
  topPtr = tempPtr;
}

//************************************************************

void Stack::Pop()                        // Removes top item from stack
{                                        // Precondition: stack is not empty
  NodeType* tempPtr;
  tempPtr = topPtr;
  topPtr = topPtr->next;
  delete tempPtr;
}

//************************************************************

ItemType Stack::Top() const              // Returns a copy of top item on stack
{                                        // Precondition: stack is not empty
  return topPtr->info;                   // Postcondition: item still on stack, copy returned
}

//************************************************************
```

```
//*******  stack.cpp continued above **********

void Stack::MakeEmpty()                    // Returns stack to empty state
{









}

//************************************************************

Stack::~Stack()                            // Destructor deallocates any nodes on the stack
{                                                                    // ==> Must
be done to prevent memory leaks <==









}

//************************************************************
```

```
//*******  stack.cpp continued above **********

void Stack::MakeEmpty()                    // Returns stack to empty state
{
  NodeType* tempPtr;

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

//****************************************************************

Stack::~Stack()                            // Destructor deallocates any nodes on the stack
{                                          // ==> Must be done to prevent memory leaks <==
  NodeType* tempPtr;

  while ( topPtr != NULL )                 // Loops to deallocate all nodes
  {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
  }
}

//****************************************************************
```

```
//*******  stackclient.cpp Standard Header Information Here **********
#include <iostream>
#include <fstream>
#include "stack.h"

using namespace std;

int main()           // Note:  Implementation changed but no change in client program!!
{
  Stack temps;
  ifstream datafile;
  ItemType someTemp;

  datafile.open("June05Temps");

  cout << "Raw Data" << endl;
  datafile >> someTemp;
  while (datafile)
  {
    cout << someTemp << endl;
    if ( !temps.IsFull() )
    {
      temps.Push(someTemp);
    }
    datafile >> someTemp;
  }

  cout << "Stack Values" << endl;
  while ( !temps.IsEmpty() )
  {
    cout << temps.Top() << endl;
    temps.Pop();
  }

  return 0;
}
```

Key Lesson:

As long as the public interface to your Abstract Data Type does not change, you may alter the ADT implementation without having to modify ***Client*** programs that make use of your ADT

This allows you to repair defects in the ADT implementation or to adjust the implementation to improve execution speed or reduce memory usage.

makefile

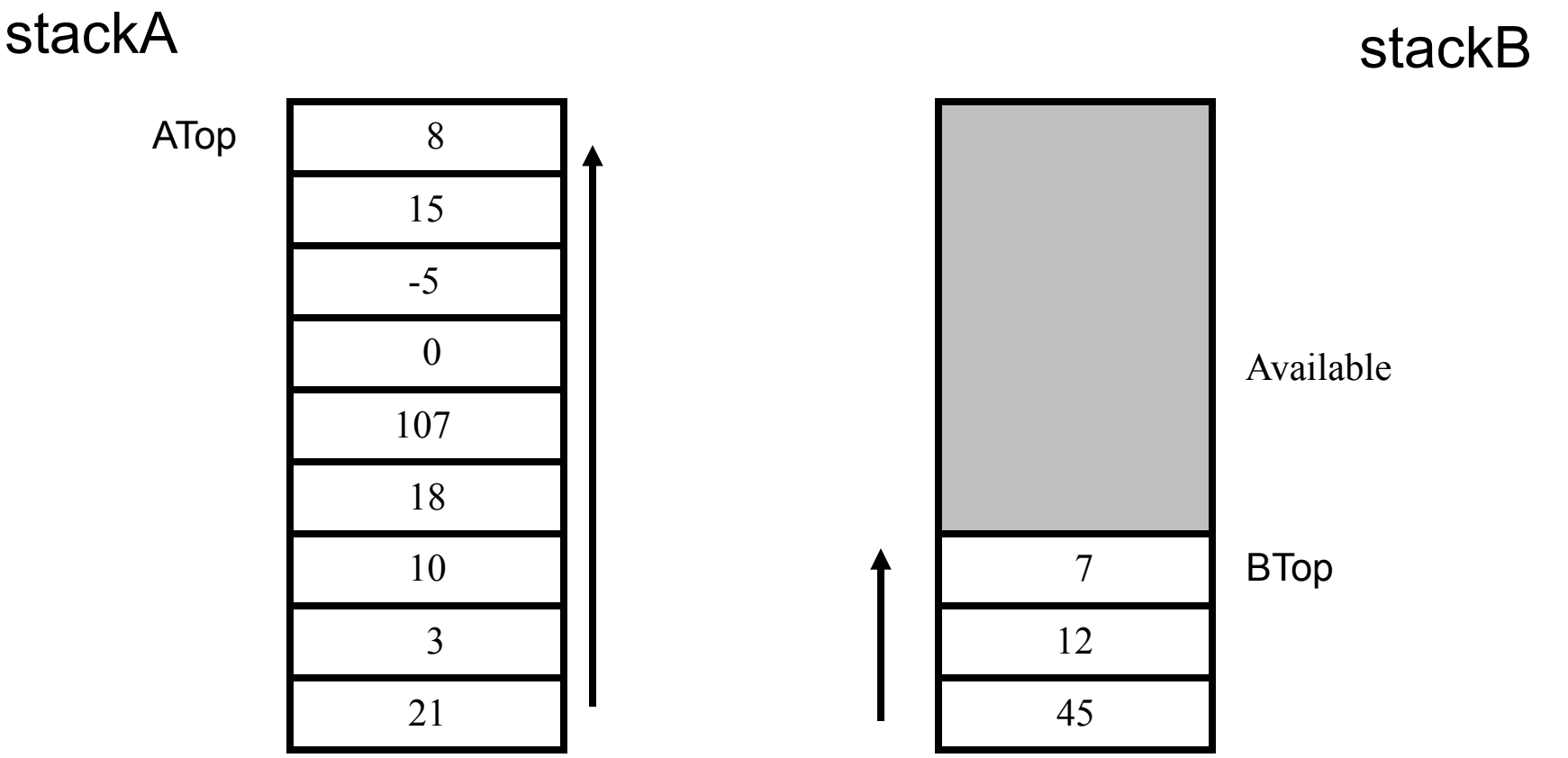
```
stackclient: stackclient.o stack.o
	g++ stackclient.o stack.o -o stackclient

stackclient.o: stackclient.cpp stack.h
	g++ -c stackclient.cpp

stack.o: stack.cpp stack.h
	g++ -c stack.cpp

clean:
	rm *.o
```

Implementation Alternatives

- Sequential Arrays
- Non-sequential Arrays of structs
- Linked List of Nodes
- Multiple Sequential Arrays
- Template Class using Linked Lists of Nodes
  - More on this later

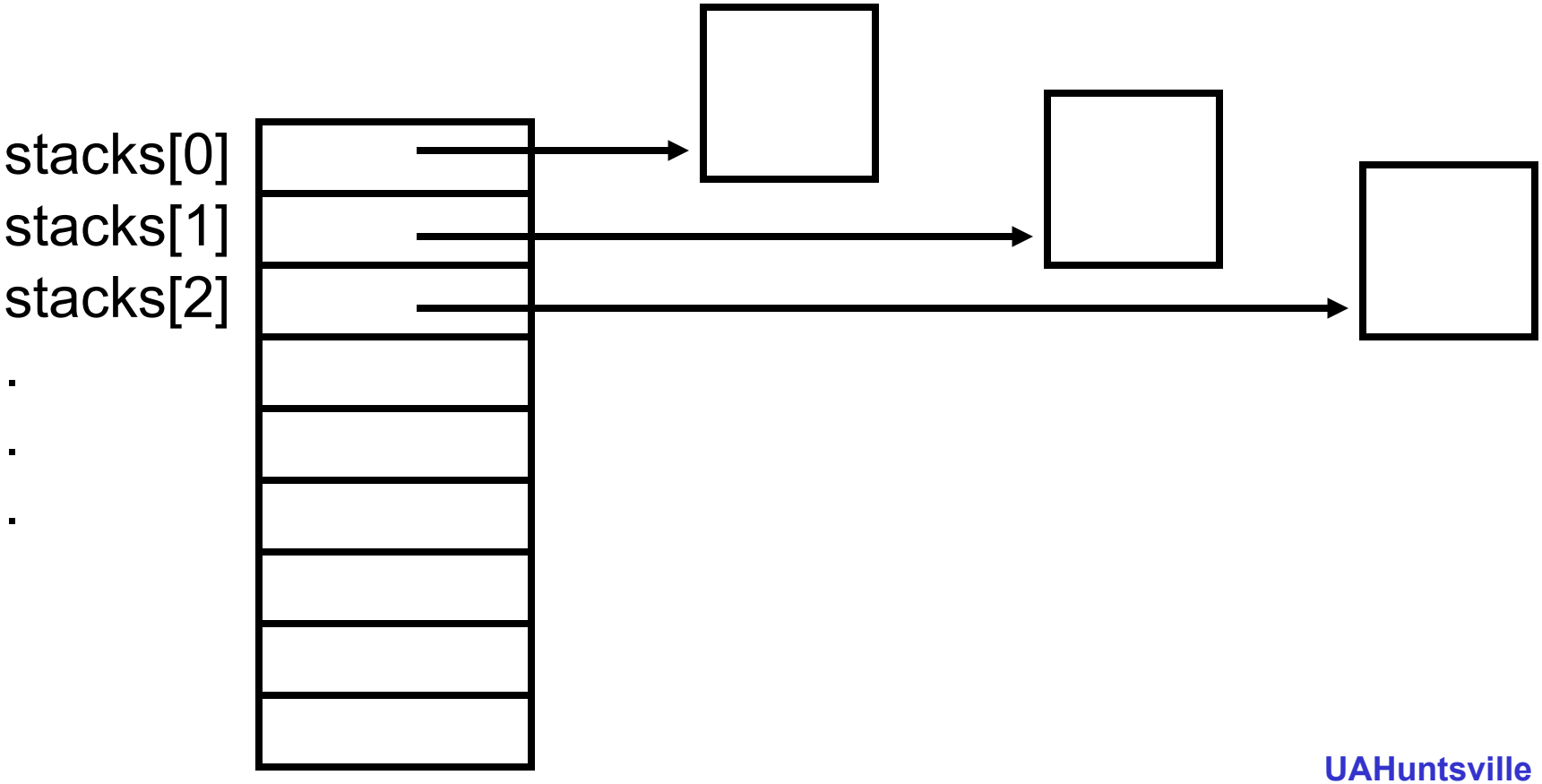
Stacks Using Multiple Sequential Arrays

int* stackA = new int[stacksize];



A Stack of Stacks Using Multiple Sequential Arrays

```
int* stacks[NUMBER_OF_STACKS];
int* newStack;
newStack = new int[STACKSIZE];
stacks[0] = newStack;
```



Summary

- **Stacks** are **Last-In, First-Out** containers
- Several ways to implement a **Stack**
  - Arrays (static or dynamic)
  - Linked, dynamically allocated nodes
  - Tradeoffs:
    - Array implementation is memory efficient but difficult to resize
    - Linked node implementation uses more memory but allows size of container to vary based upon amount of data