CPE 212 - Fundamentals of Software Engineering

Stacks

Project03 is due this Friday by 11:59pm

Outline

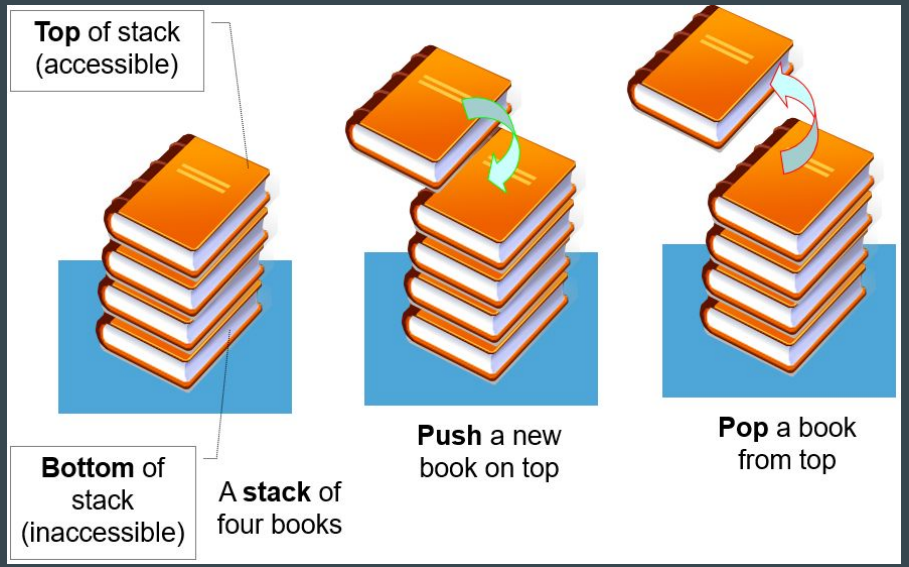
- Data Structures
- Stack Concepts
- Implementations
- Coding Examples

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

- 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 of dishes



Stack- 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
- What else?

Stack - 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

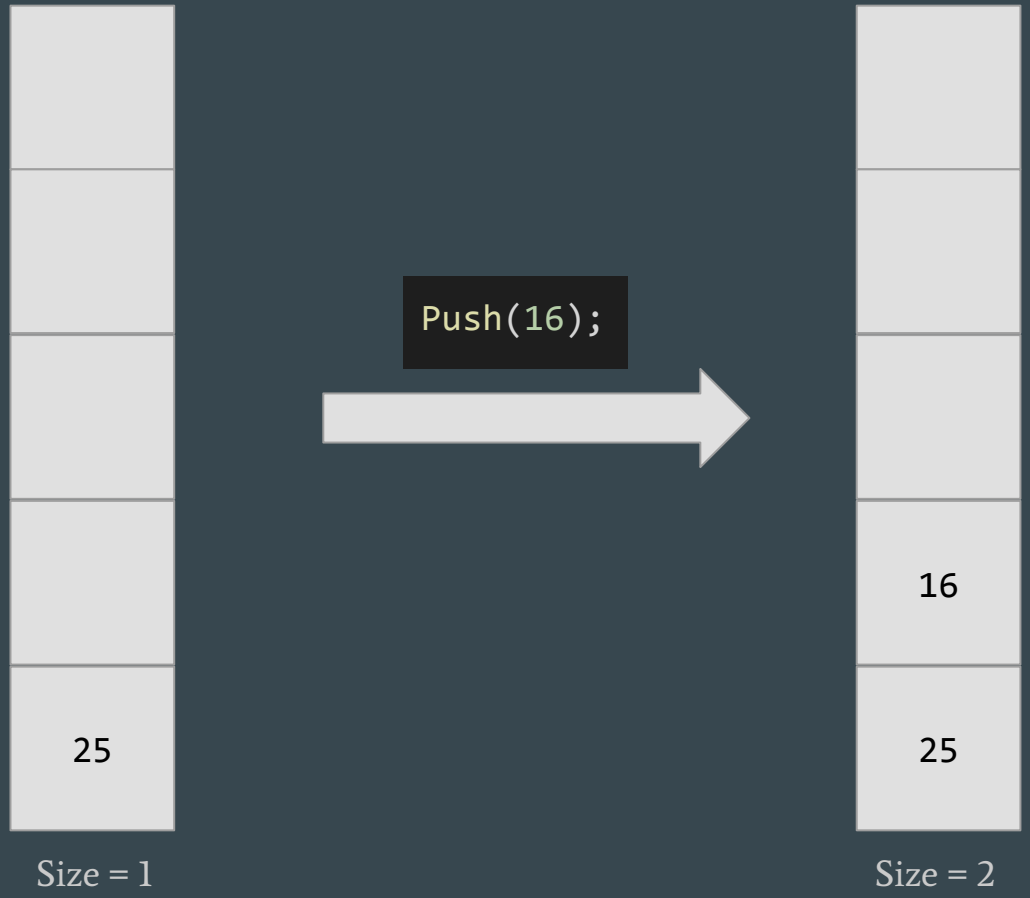
- determines whether stack is full

MakeEmpty

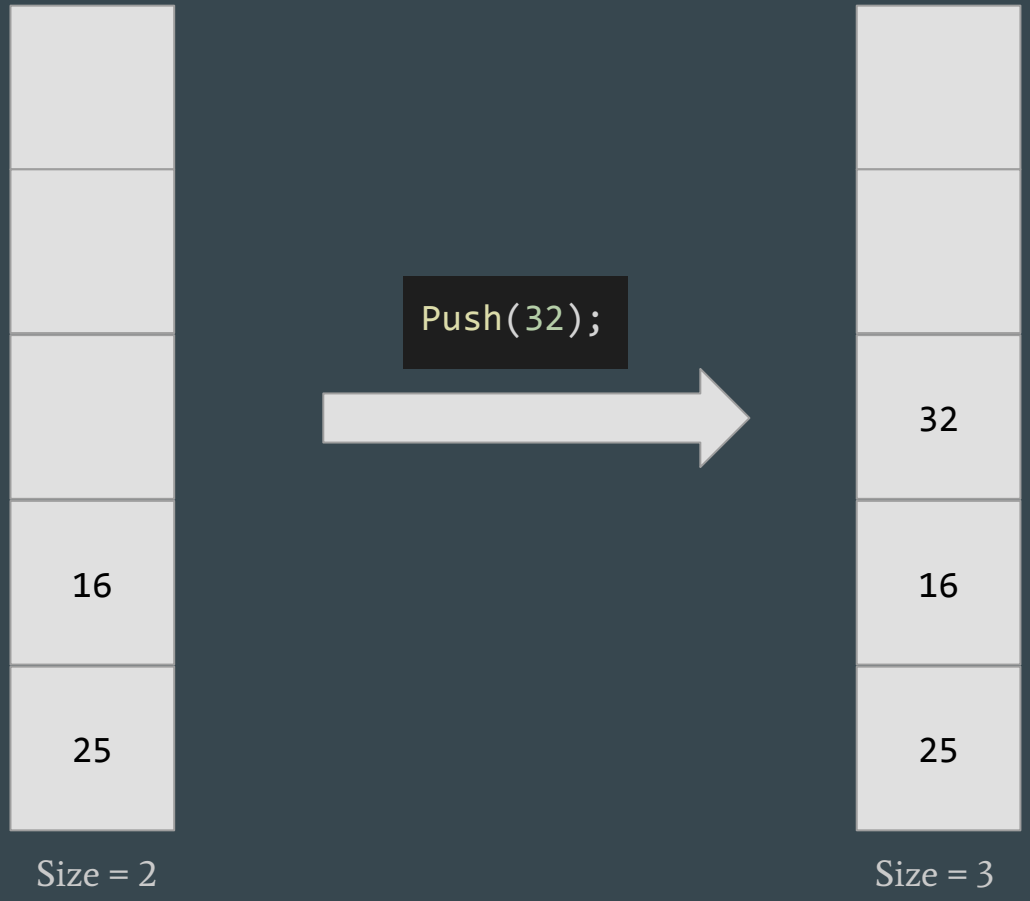
- empties the stack

Push

Push



Push



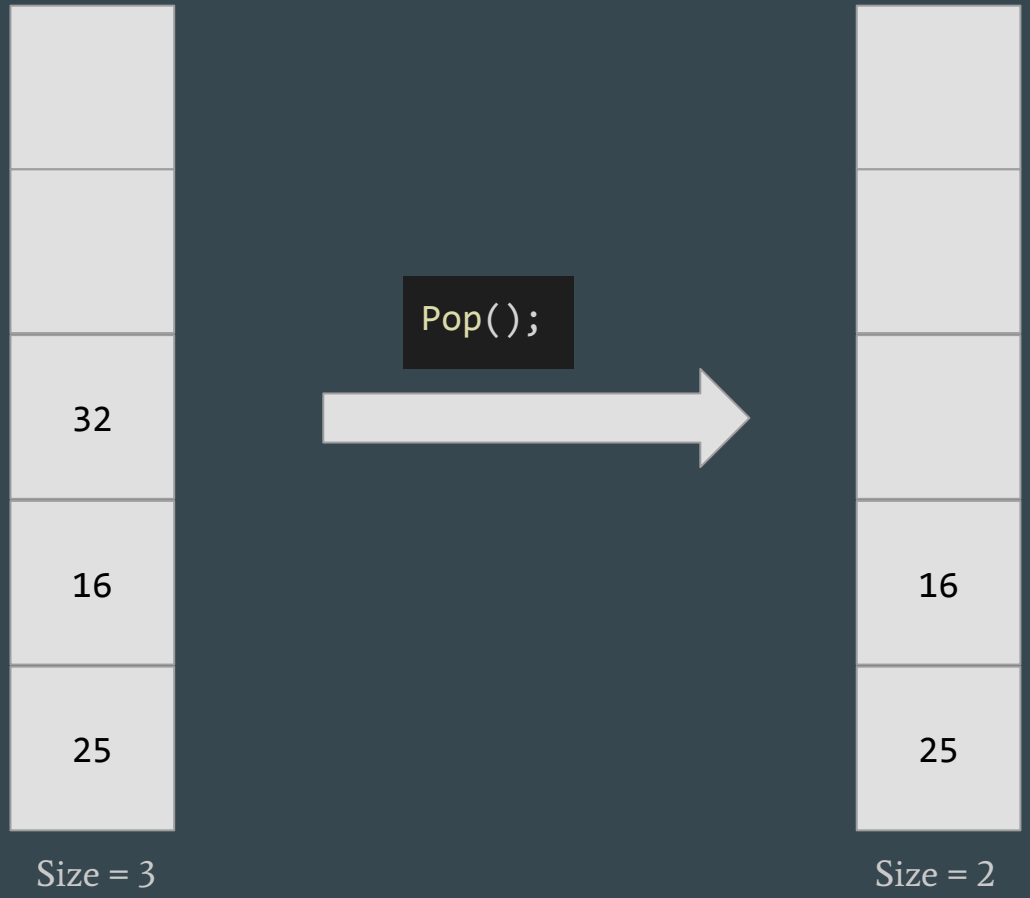
Top

Size = 3

```
Top();
```

```
32
```

Pop



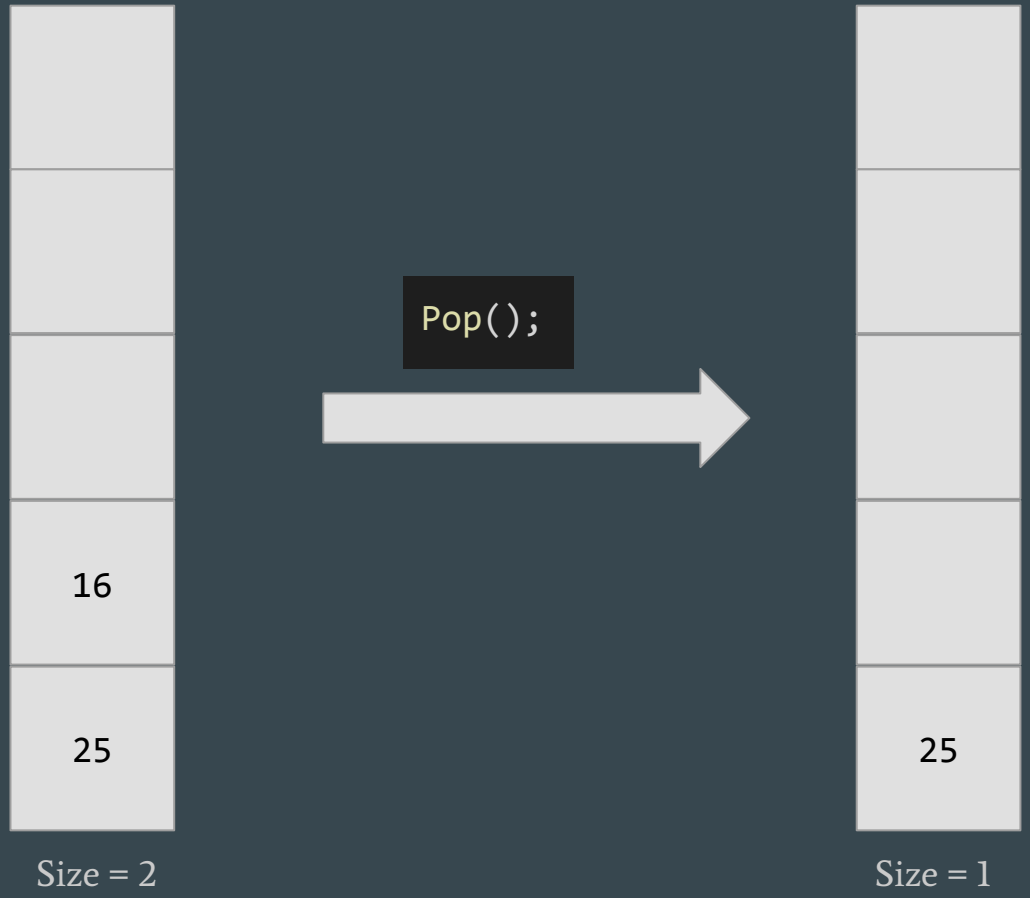
Top

Size = 3

```
Top();
```

```
16
```

Pop



Queue - Operation Limitations

- What happens when Push is invoked when the stack is full?
- What happens when Pop is invoked when the stack is empty?
- What happens when Top is invoked when the stack is empty?
- Same options as with Queue ADT
  - 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 Responsibility

```
// 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 Responsibility

```
// 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 Implementation

Sequential Array - stack.h

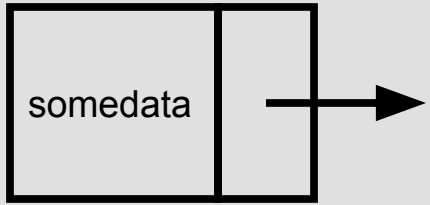
```
//*******  stack.h Standard Header Information Here **********

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];      // Array of stack data
  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 Implementation

Sequential Array - stack.cpp

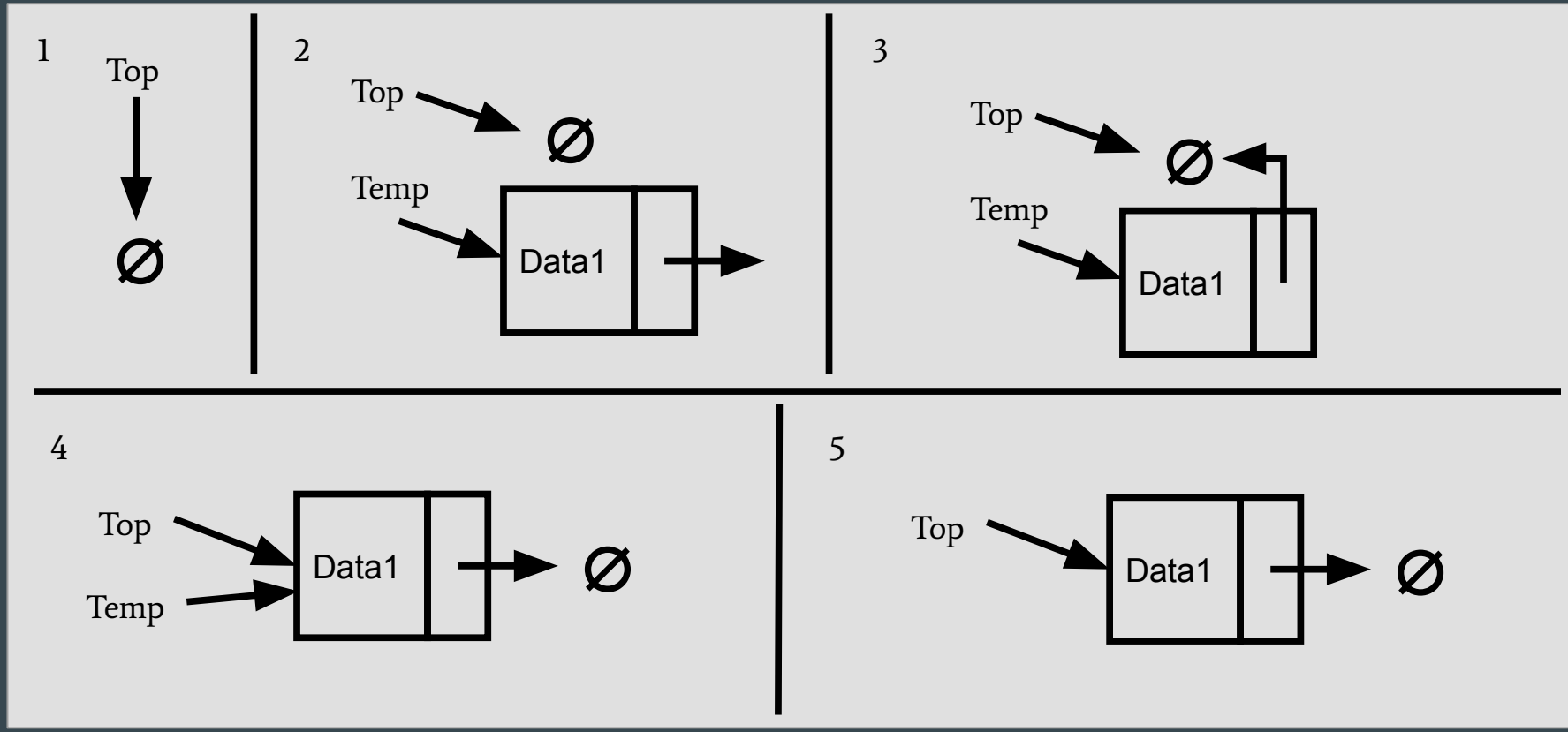
```
//*******  stack.cpp Standard Header Information Here **********
#include “stack.h”

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;
}
```

Stack ADT - Linked List of Dynamically-Allocated Nodes

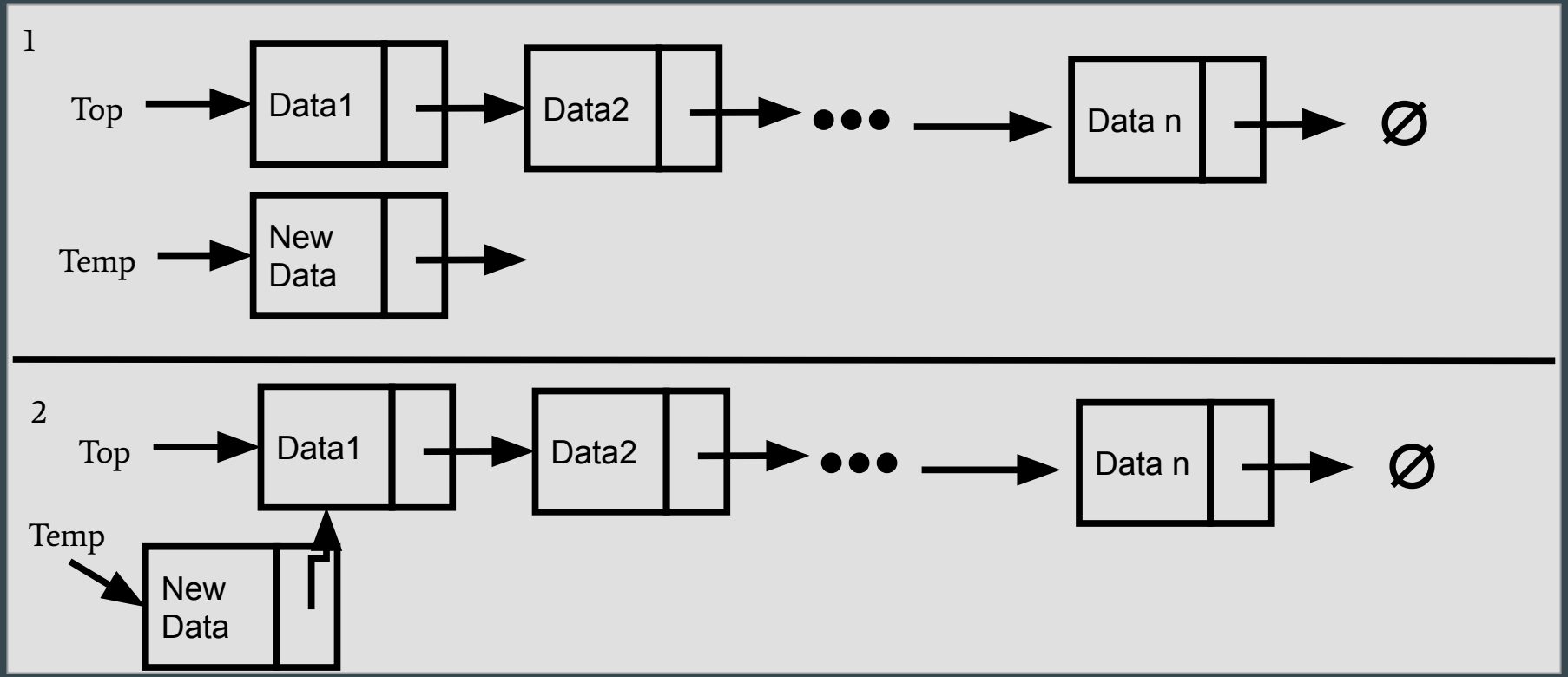



```
struct  NodeType
{
   ItemType        info;
   NodeType*       next;
};
```

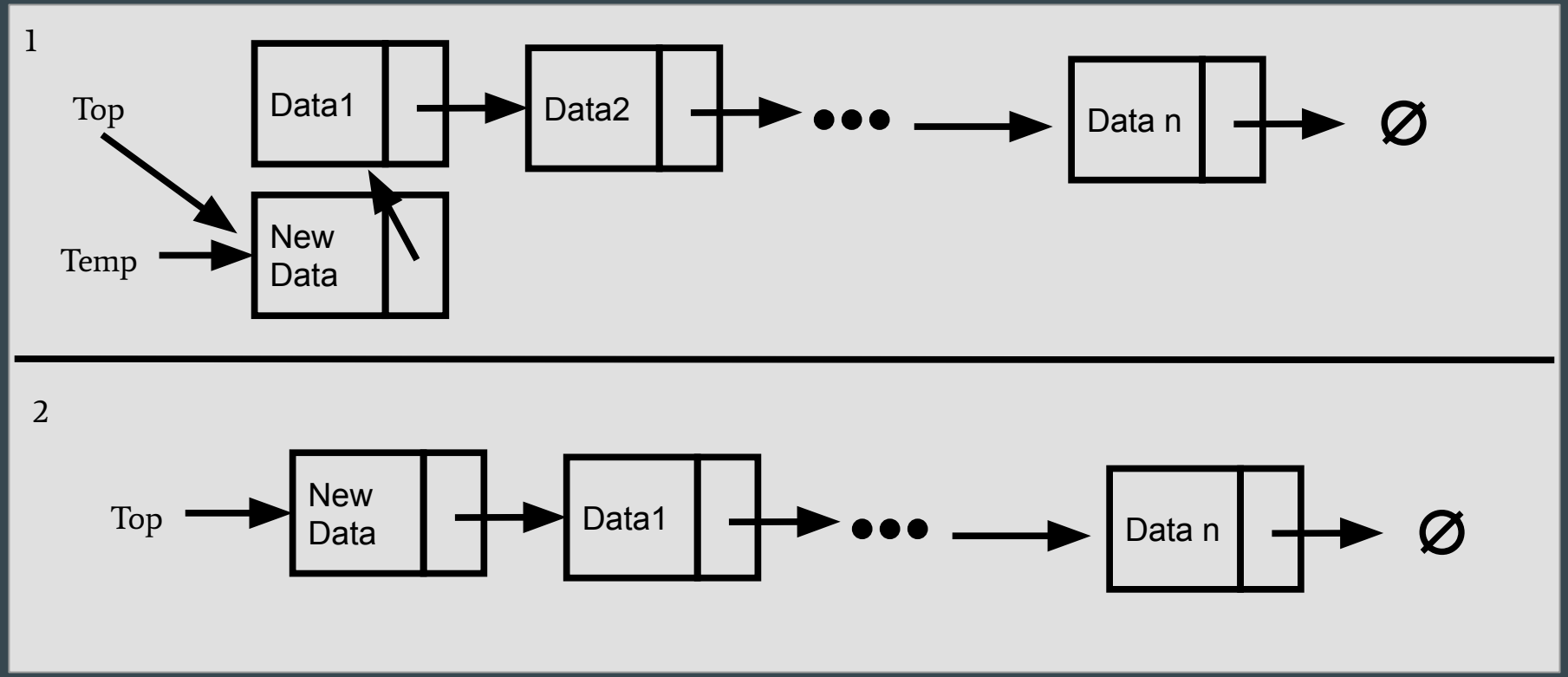
Stack ADT- Push



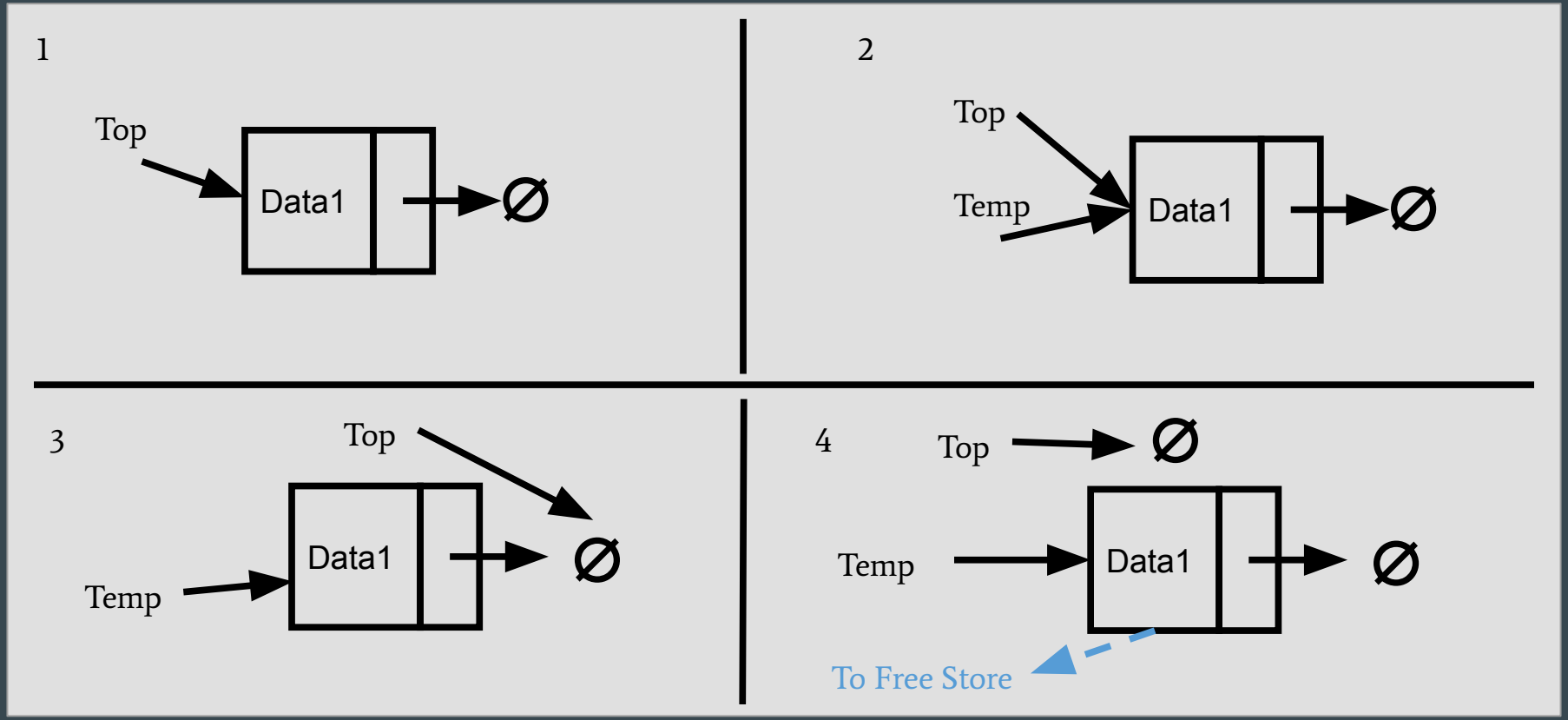
Stack ADT- Push Onto Populated Stack



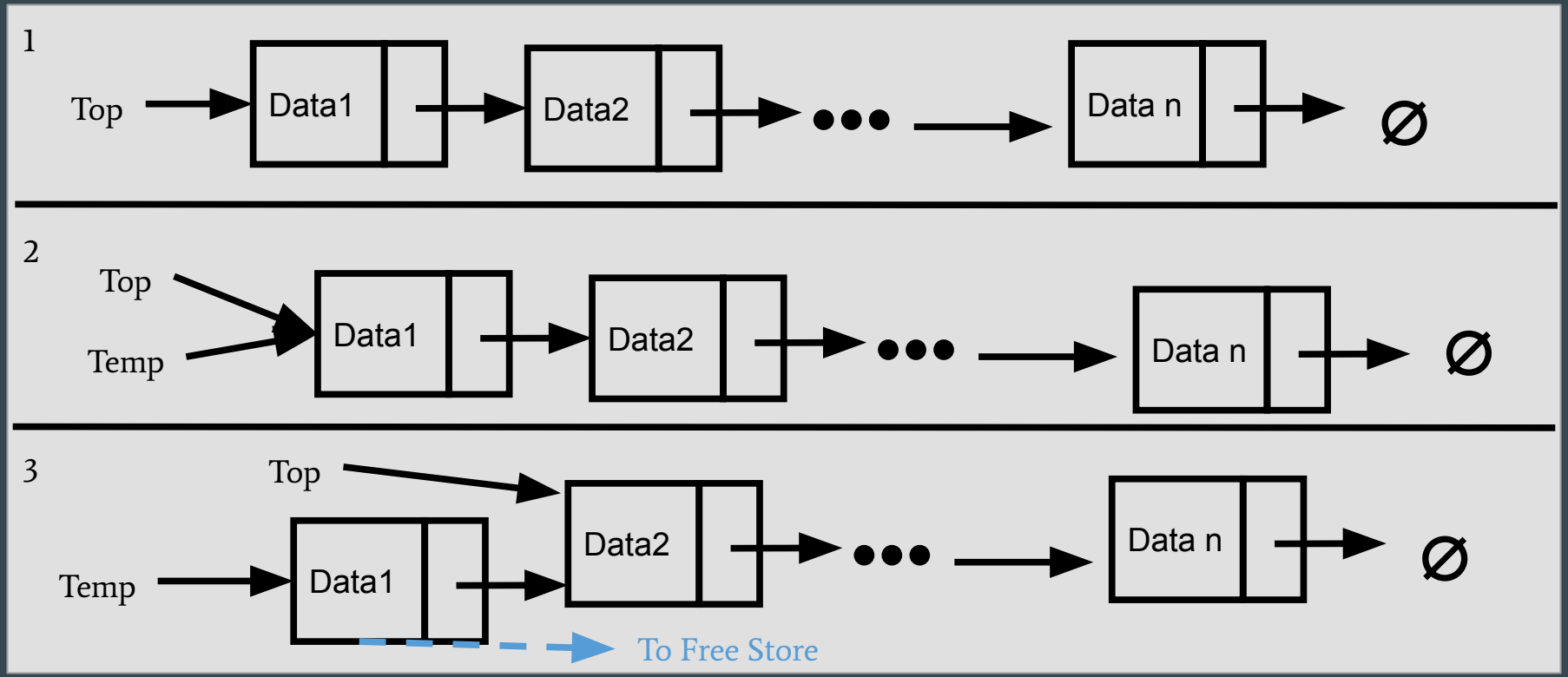
Stack ADT- Push Onto Populated Stack



Stack ADT- Pop



Stack ADT- Pop From a Populated Stack



Stack Implementation

Link List - stack.h

```
//*******  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 structure
{
  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 Implementation

Link List - stack.cpp

```
//*******  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 Implementation

Link List - stack.cpp

```cpp
//*******  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 Implementation

Link List - stack.cpp

```
//************************************************************
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 Implementation

Link List - stack.cpp

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

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 Implementation

Link List - stack.cpp

```
//********************************************************
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 Implementation

Link List - stack.cpp

```
//************************************************************
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;
  }
}


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

Stack Implementation

Link List - main.cpp

```
//*******  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;
}
```

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