# STACKS

NATIONAL UNIVERSITY OF TECHNOLOGY (NUTECH)

DR. SAMAN RIAZ

LECTURE # 12

# STACKS

"A ***Stack*** is a special kind of list in which all insertions and deletions take place at one end, called the ***Top***"

*A Stack is a recursive data structure.*

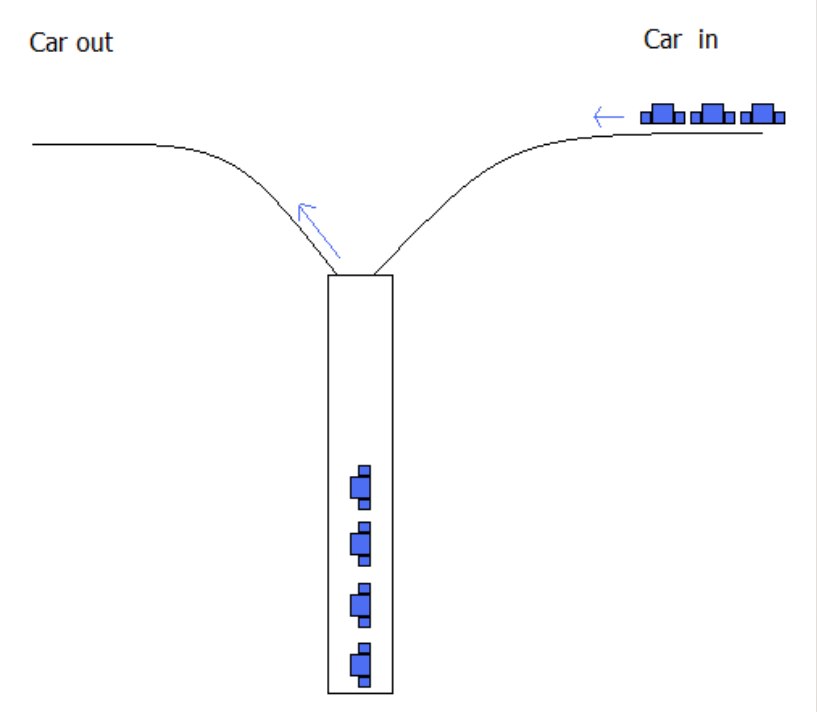
## Other Names

- Pushdown List
- Last In First Out (LIFO)

# STACKS (EXAMPLES)

- Books on a floor
- Dishes on a shelf

# STACKS (EXAMPLES)




Is there an appropriate data type to model this parking lot???

# APPLICATIONS

- Undo and Redo in MS Word
  - Last action would be undone/redone first
- Banking transactions
  - Last transaction would be viewed first
- Mathematical operations
  - Operators precedence

# COMMON OPERATIONS ON STACKS

1. **MAKENULL(*S*):** Make Stack S be an empty stack.
2. **TOP(*S*):** Return the element at the top of stack *S*.

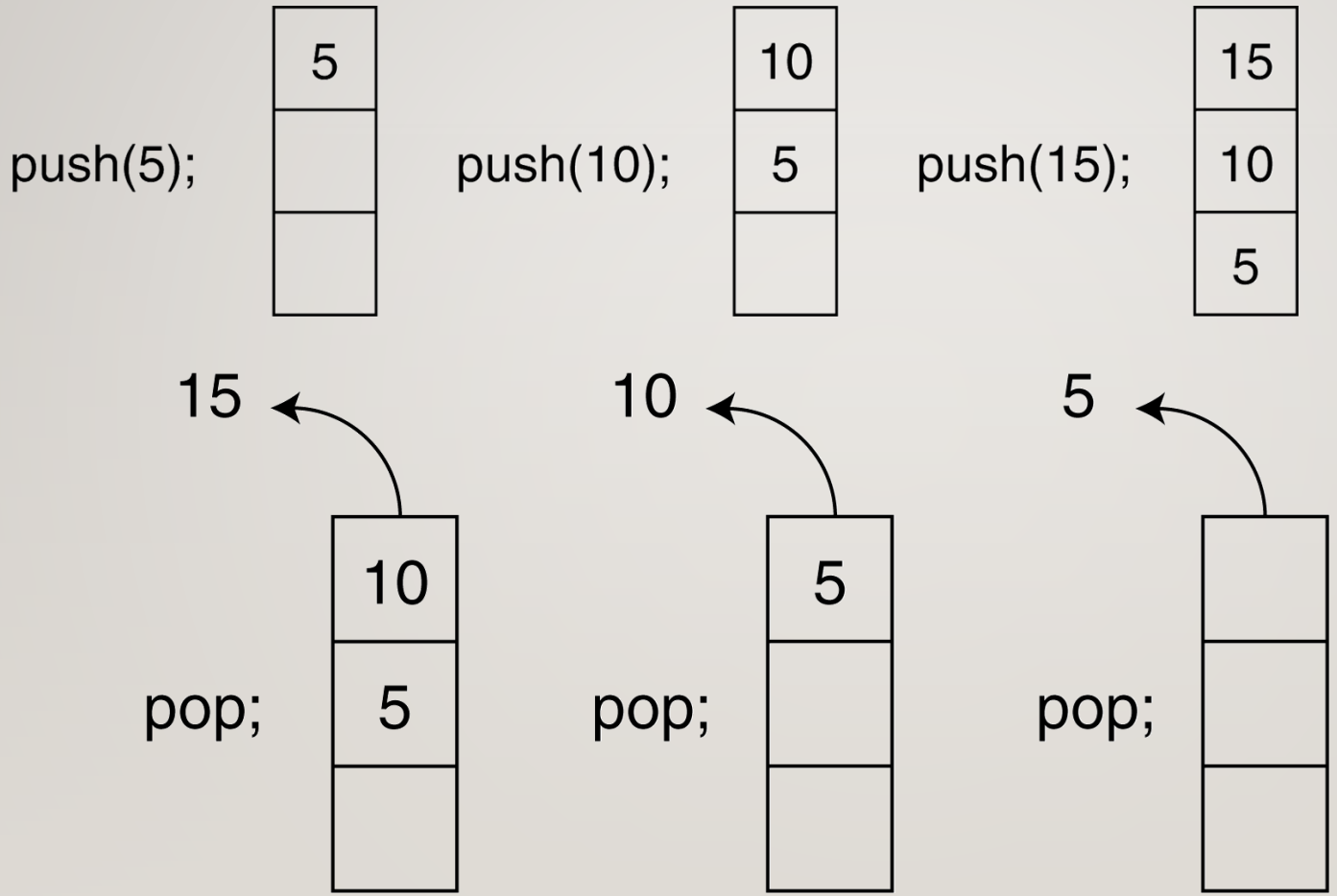
3. **POP(*S*):** Remove the top element of the stack.
4. **PUSH(*S*,*x*):** Insert the element *x* at the top of the stack.

5. **EMPTY(*S*):** Return true if S is an empty stack; return false otherwise.

# STATIC AND DYNAMIC STACKS

- There are two kinds of stack data structure -
  - a) **static**, i.e. they have a **fixed size**, and are *implemented as* **arrays**.
  - b) **dynamic**, i.e. they **grow in size** as needed, and *implemented as* **linked lists**

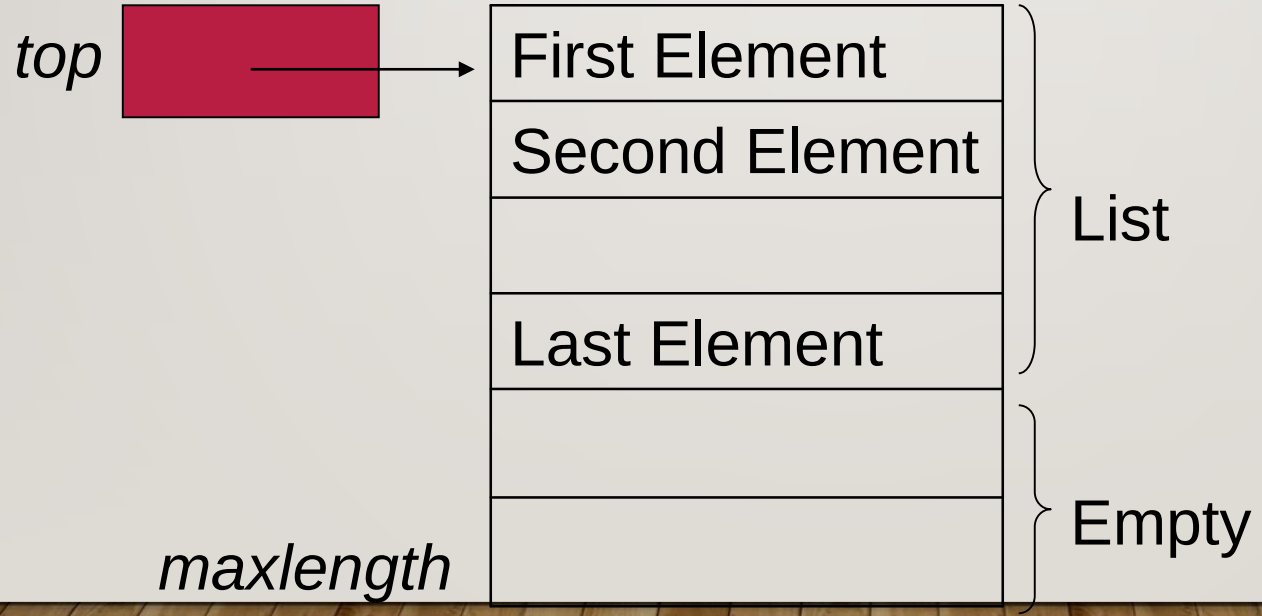
# PUSH AND POP OPERATIONS OF STACK



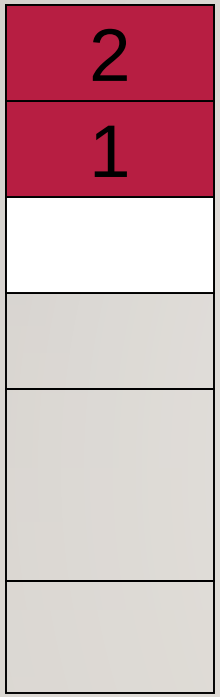
# AN ARRAY IMPLEMENTATION OF STACKS

**First Implementation**

- Elements are stored in contiguous cells of an array.
- New elements can be inserted to the top of the list.



# AN ARRAY IMPLEMENTATION OF STACKS




**Problem with this implementation**

- Every PUSH and POP requires moving the entire array up and down.

# OPERATIONS

- **Push**
  - Stores a value to the top of the stack
- **Pop**
  - Retrieves a value from the top of the stack
- **IsFull**
  - Returns true if the stack is full
- **IsEmpty**
  - Returns true if the stack is empty

# IMPLEMENTATION

- Stack can be implemented using
  - **Array**
    - Static size
      - **Pro**: Best performance
      - **Con**: Fixed size
  - **Linked List**
    - Dynamic size
      - **Pro**: Variable size
      - **Con**: Bigger size can hurt the performance

# IMPLEMENTATION USING ARRAY

- **IntStack class**

```
class IntStack
{
    private:
        int *stackArray;
        int stackSize;
        int top;

    public:
        IntStack(int);   // Constructor
        void Push(int);
        void Pop(void);
        bool isFull(void);
        bool isEmpty(void);
        void Display(void);
};
```

# IMPLEMENTATION USING ARRAY

- **Constructor**

```
IntStack::IntStack(int size)
{
    stackArray = new int[size];
    stackSize  = size;
    top = -1;
}
```

# IMPLEMENTATION USING ARRAY

- **Push()**

```
void IntStack::Push(int num)
{
    if (isFull())
    {
        cout << "The stack is full.\n";
    }

    else
    {
        top++;
        stackArray[top] = num;
    }
}
```

# IMPLEMENTATION USING ARRAY

- **Pop()**

```
void IntStack::Pop(void)
{
    if (isEmpty())
    {
        cout << "The stack is empty.\n";
    }

    else
    {
        top--;
    }
}
```

# IMPLEMENTATION USING ARRAY

- **isFull()**

```
bool IntStack::isFull(void)
{
    bool status;

    if (top == stackSize - 1)

        status = true;
    else

        status = false;

    return status;
}
```

# IMPLEMENTATION USING ARRAY

- **isEmpty()**

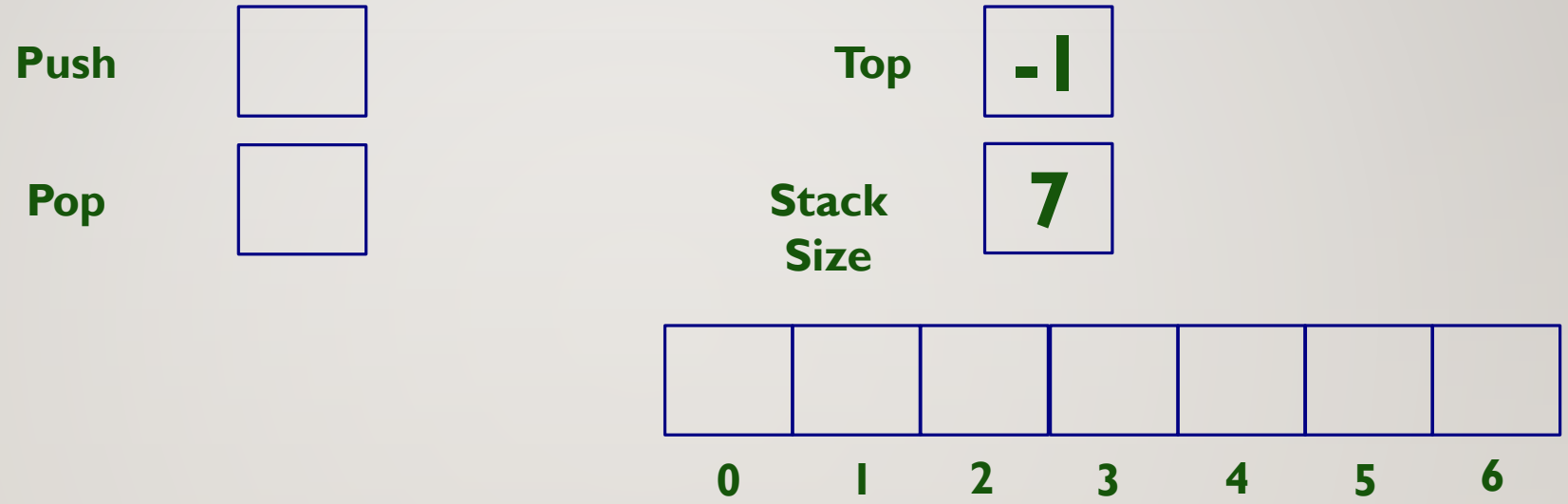
```
bool IntStack::isEmpty(void)
{
    bool status;

    if (top == -1)

        status = true;
    else

        status = false;

    return status;
}
```

# IMPLEMENTATION USING ARRAY

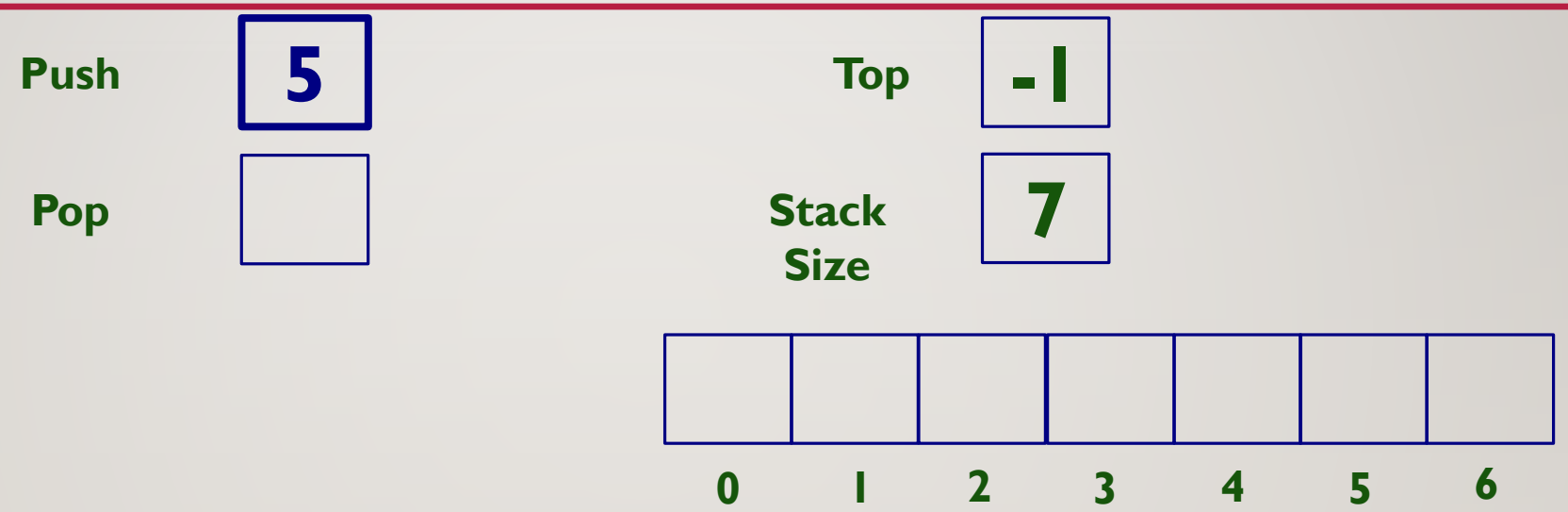
- **Display()**

```
void IntStack::Display(void)
{
    if (isEmpty()) {
        cout << "The stack is empty.\n";
    }
    else {
        int t = top;
        while (t >= 0)
        {
            cout << “stackArray[” << t << “] = ”
            << stackArray[t] << endl;
            t--;
        }
    }
}
```

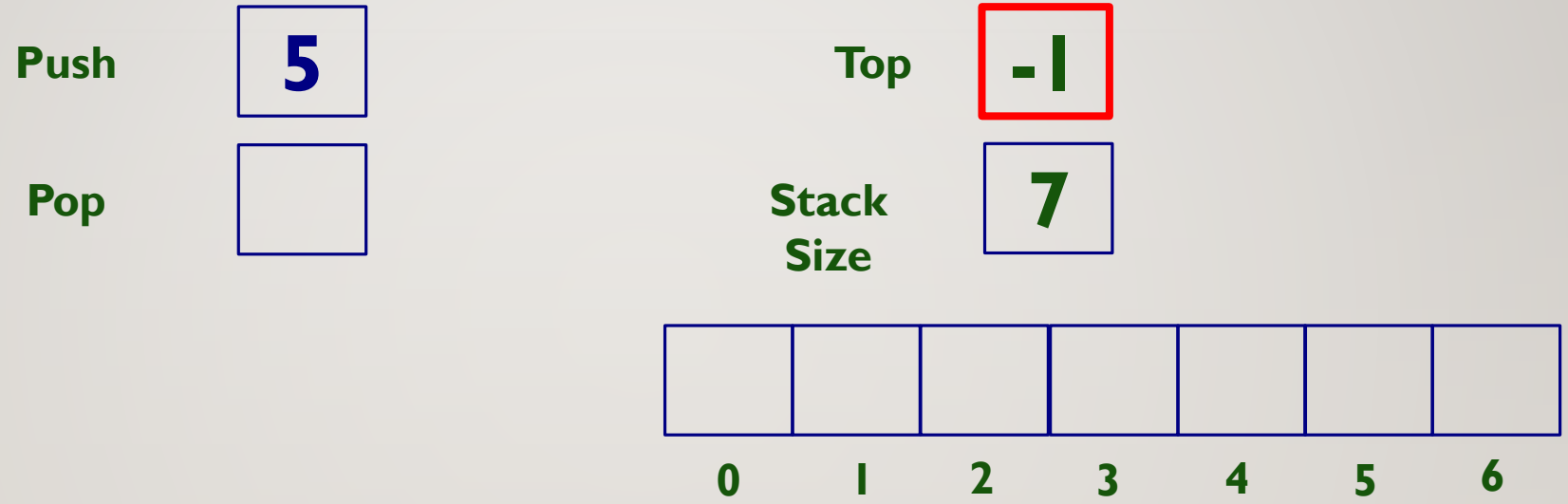
# DEMONSTRATION



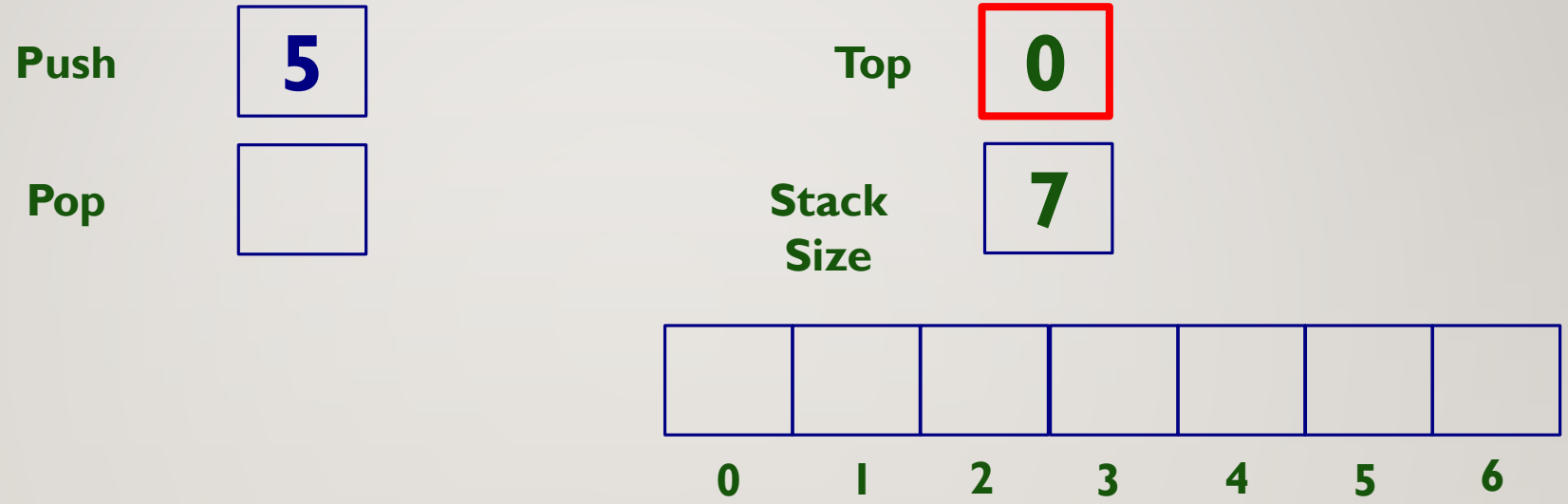
# DEMONSTRATION



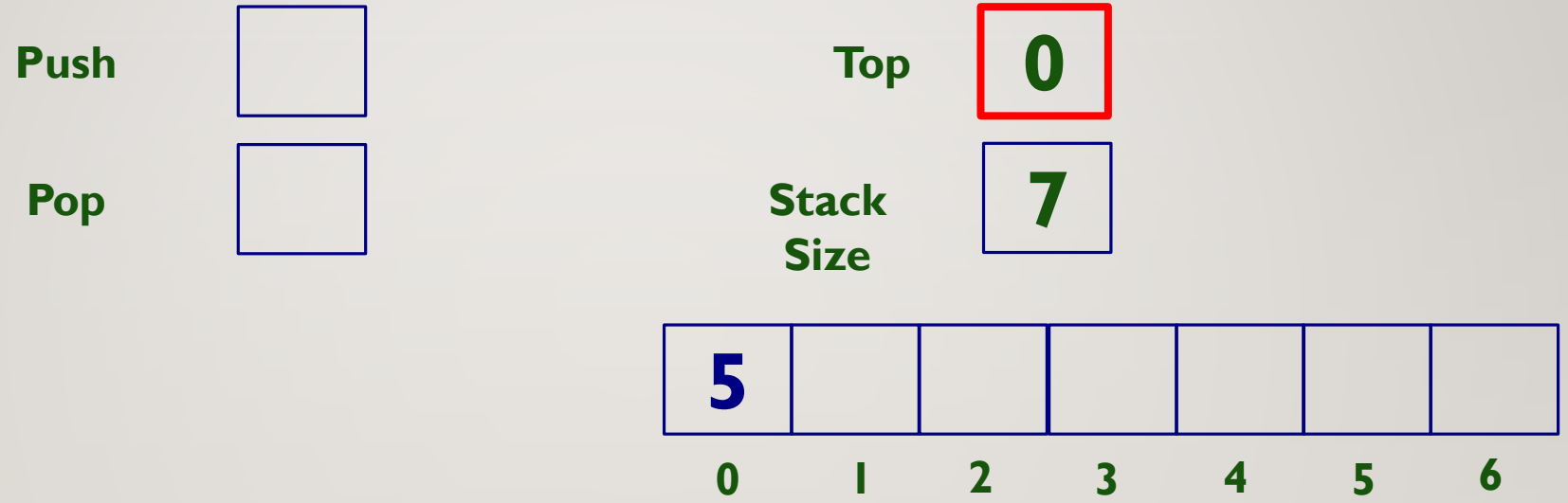
# DEMONSTRATION



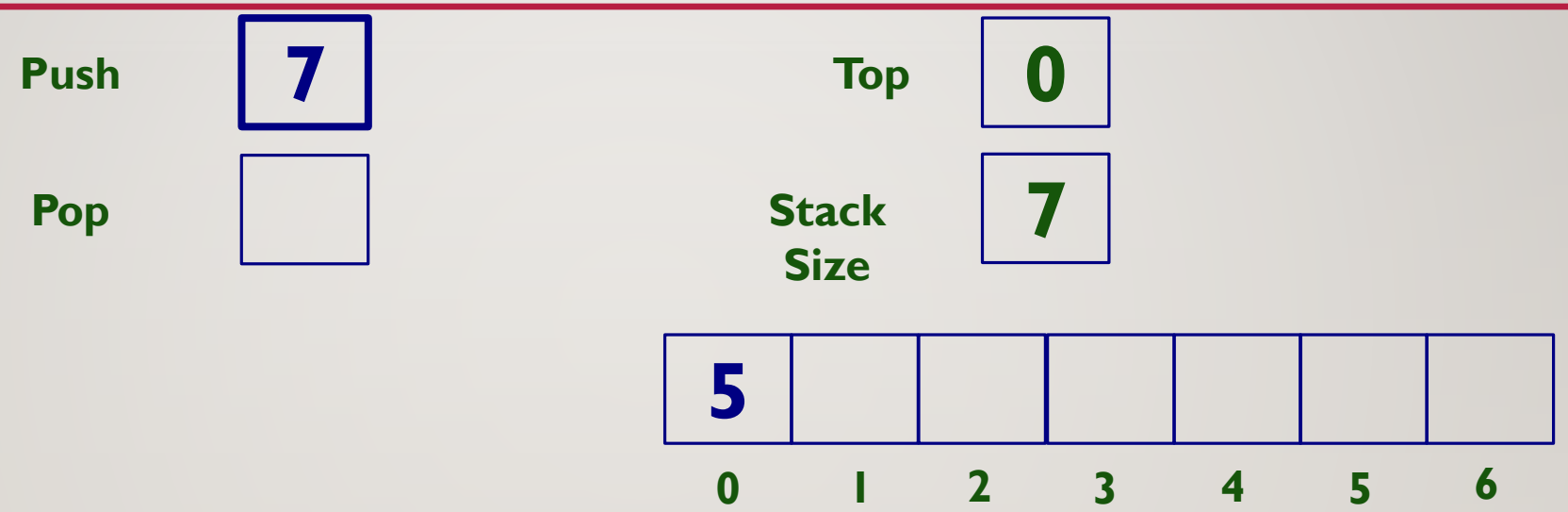
# DEMONSTRATION



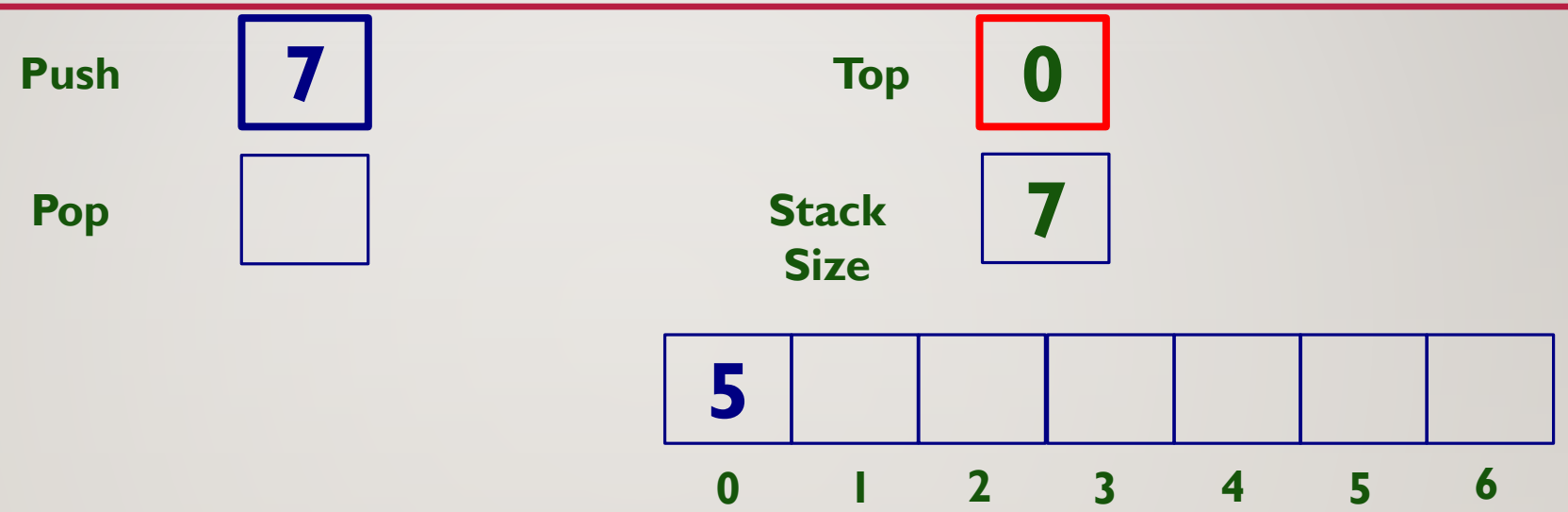
# DEMONSTRATION



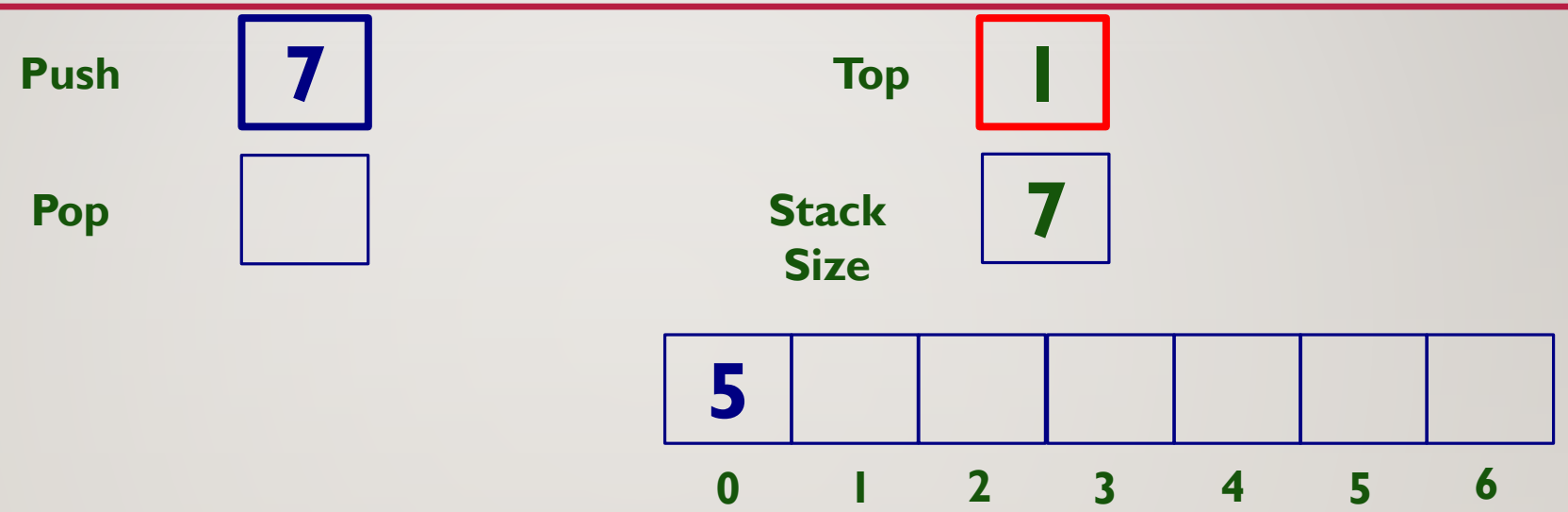
# DEMONSTRATION



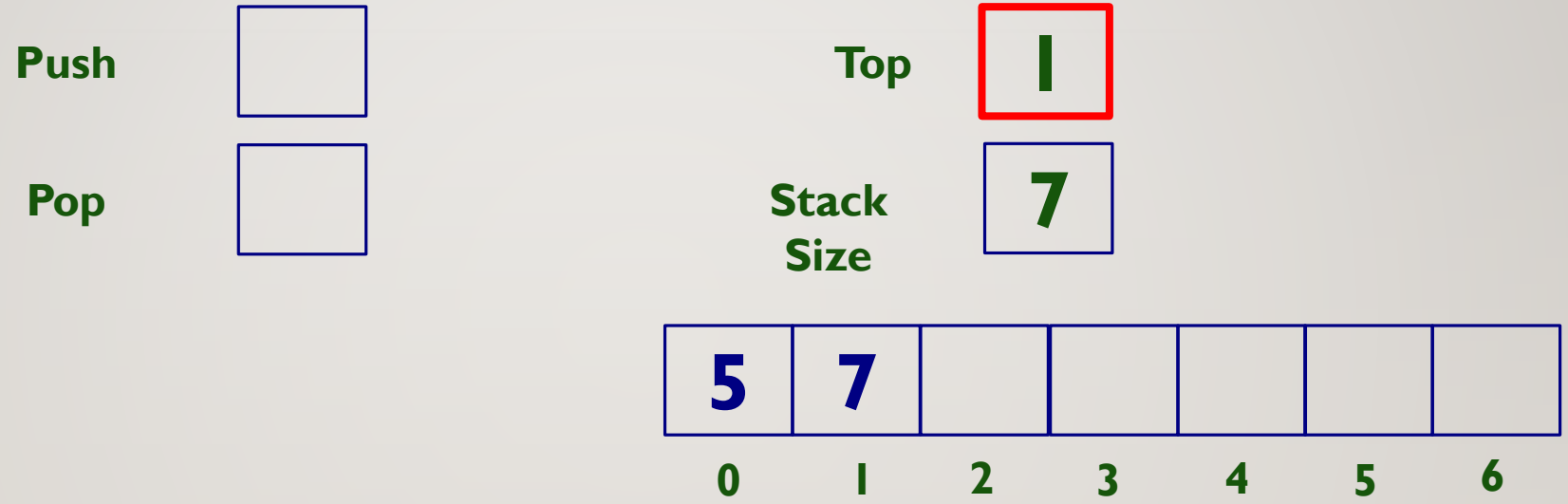
# DEMONSTRATION



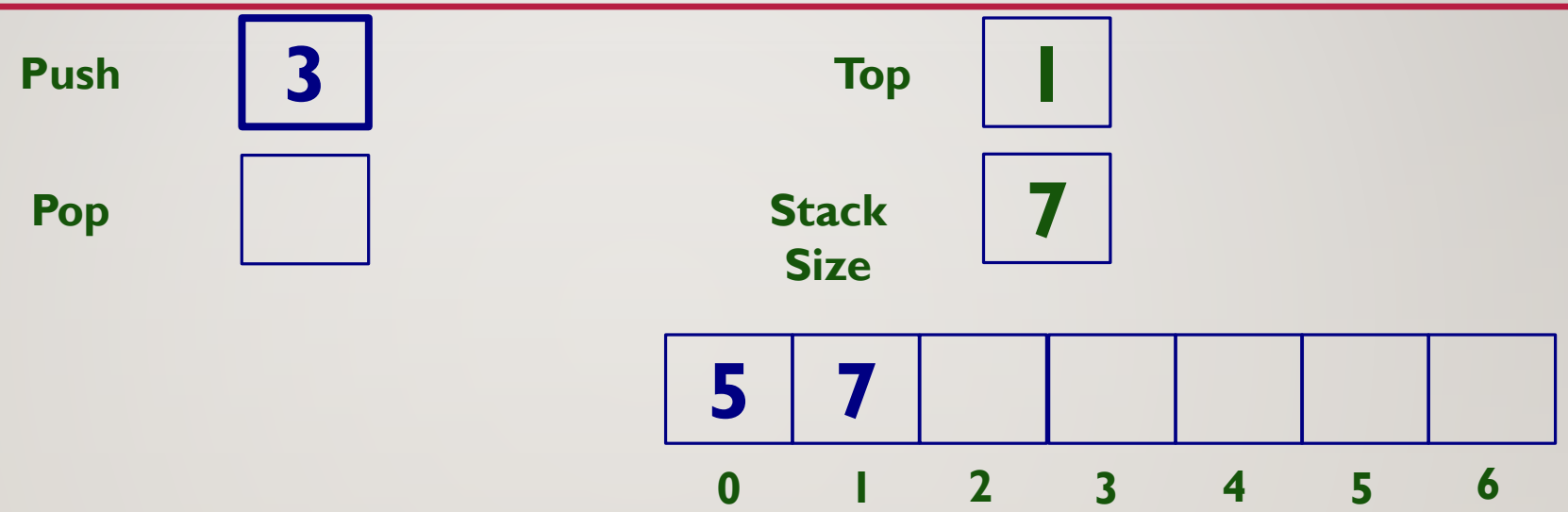
# DEMONSTRATION



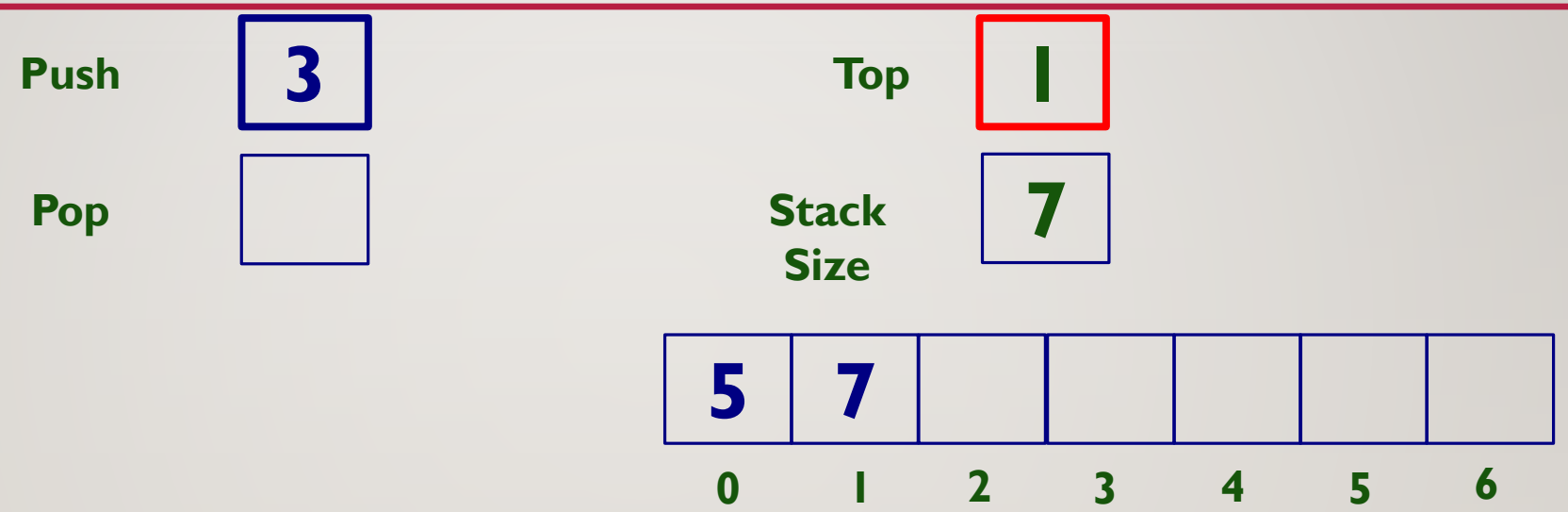
# DEMONSTRATION



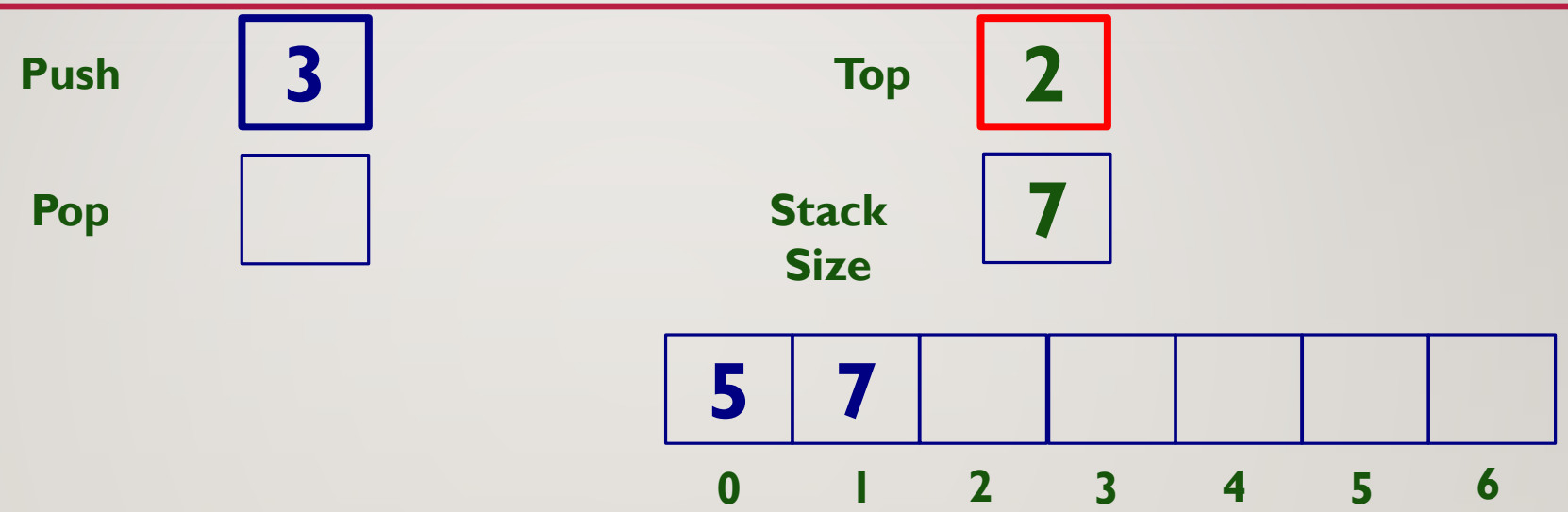
# DEMONSTRATION



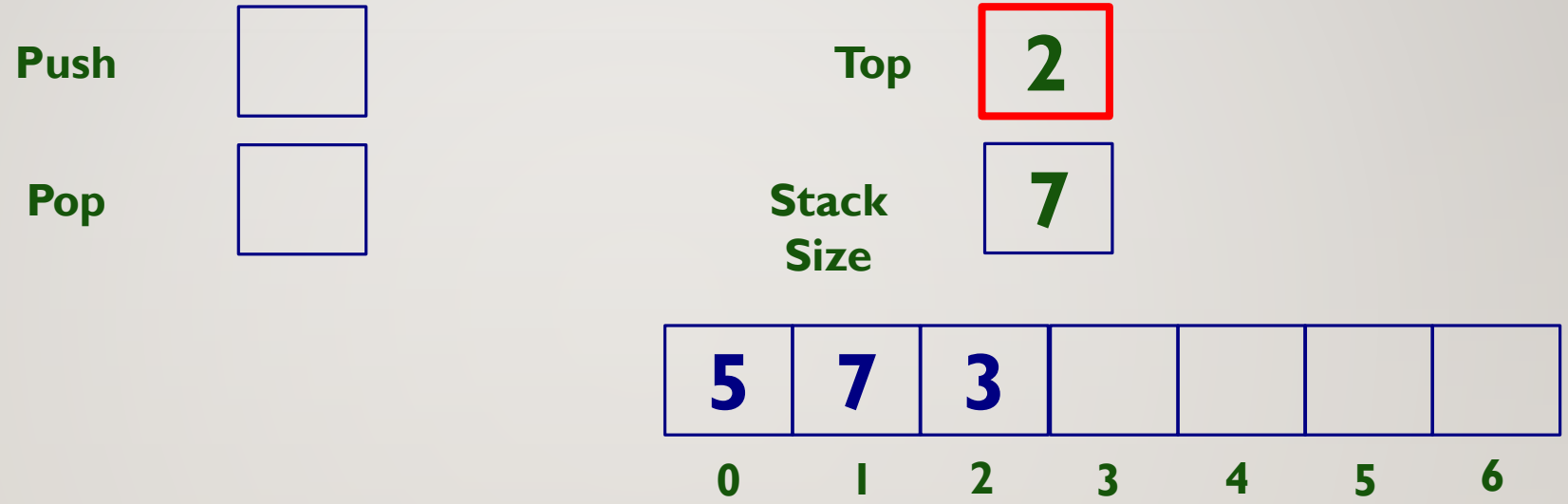
# DEMONSTRATION



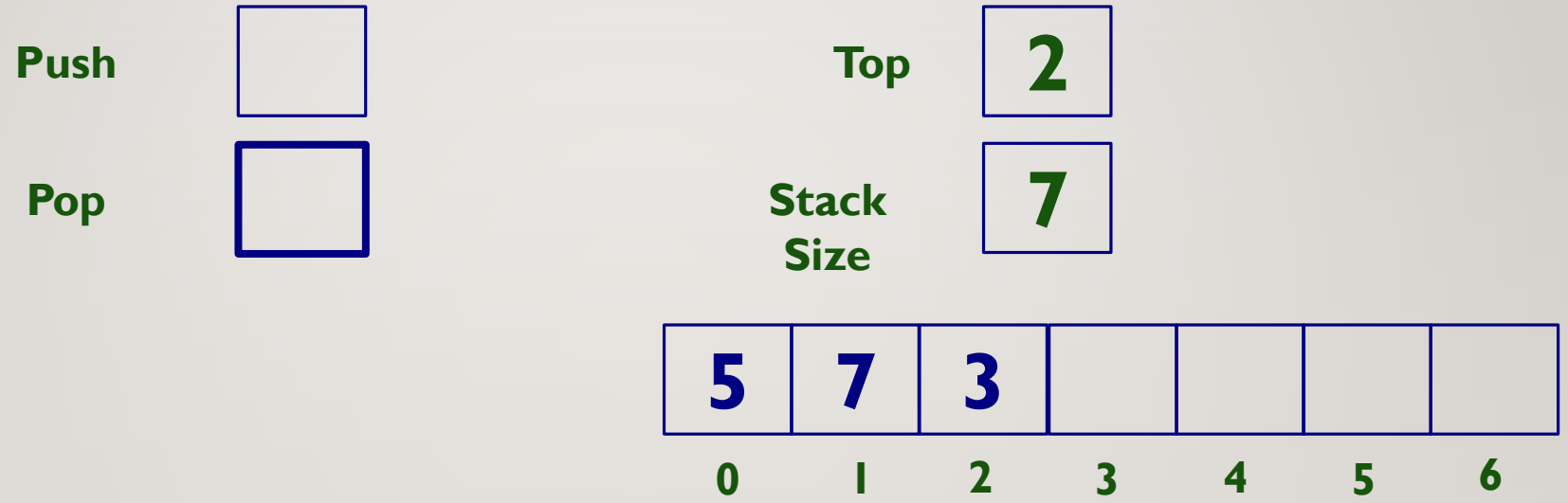
# DEMONSTRATION



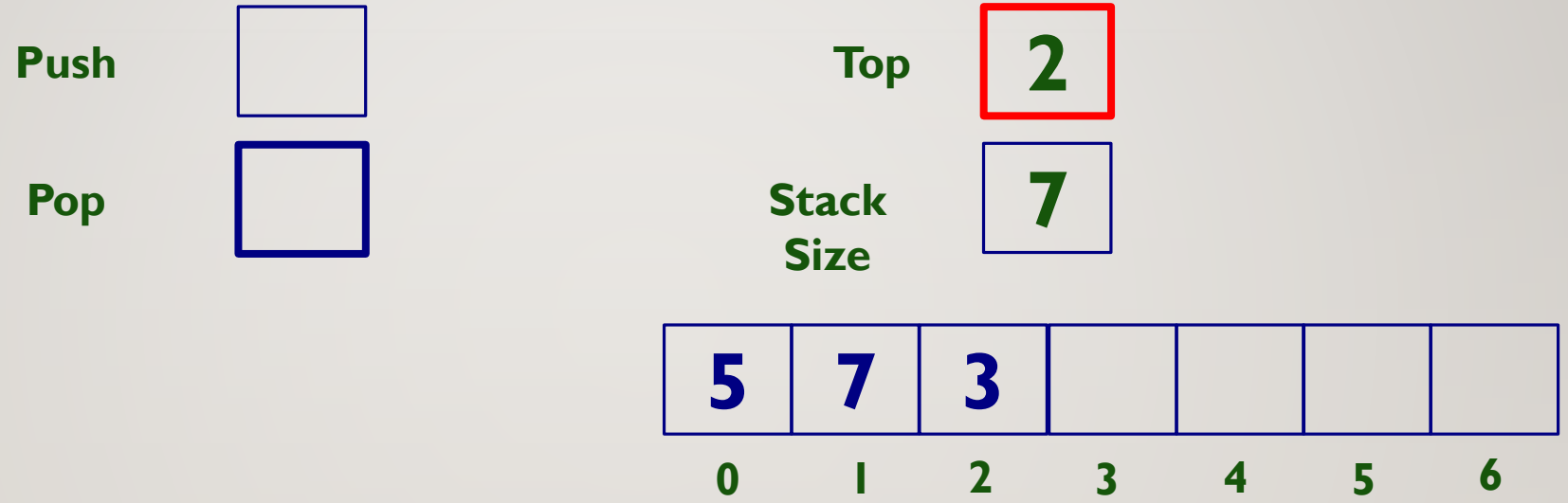
# DEMONSTRATION



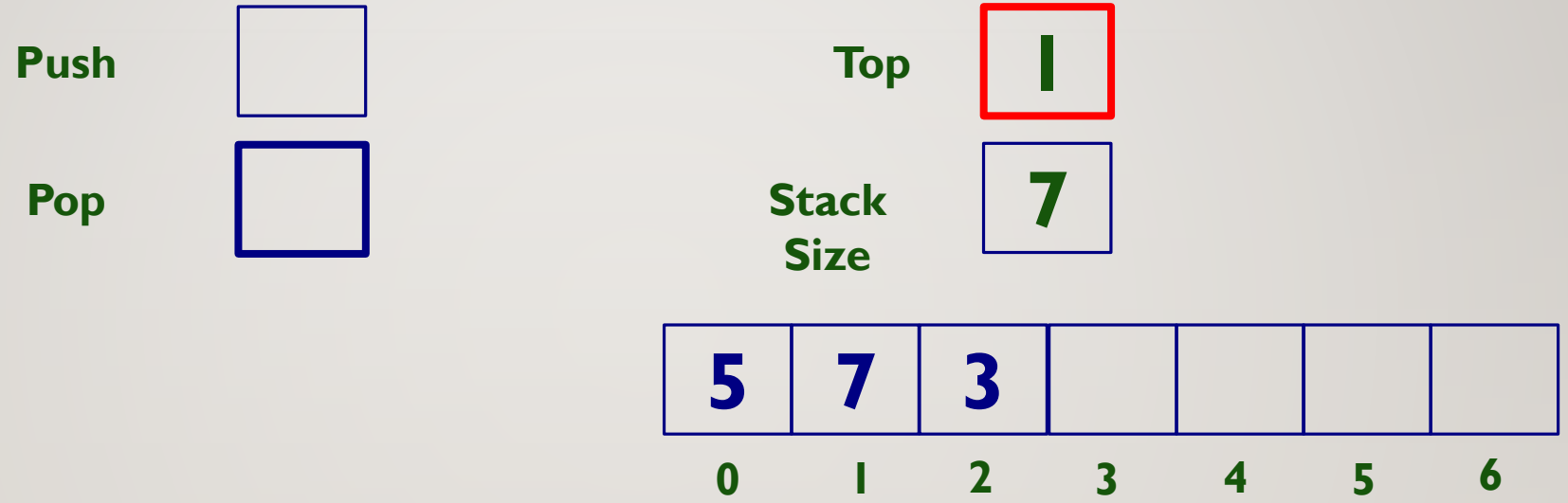
# DEMONSTRATION



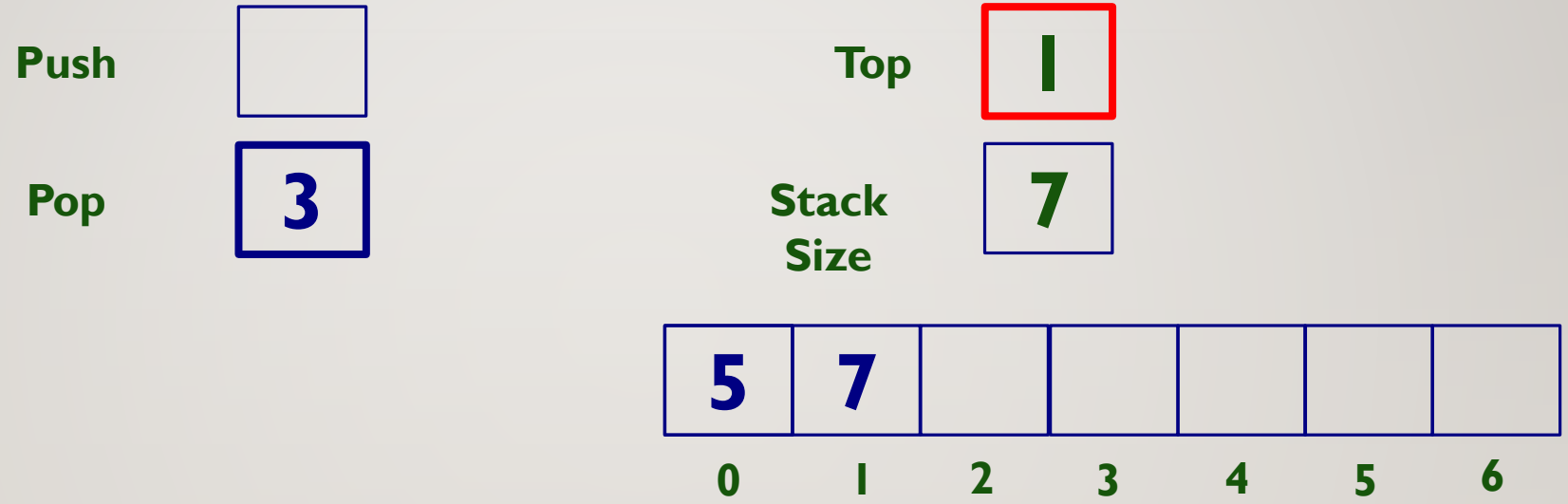
# DEMONSTRATION



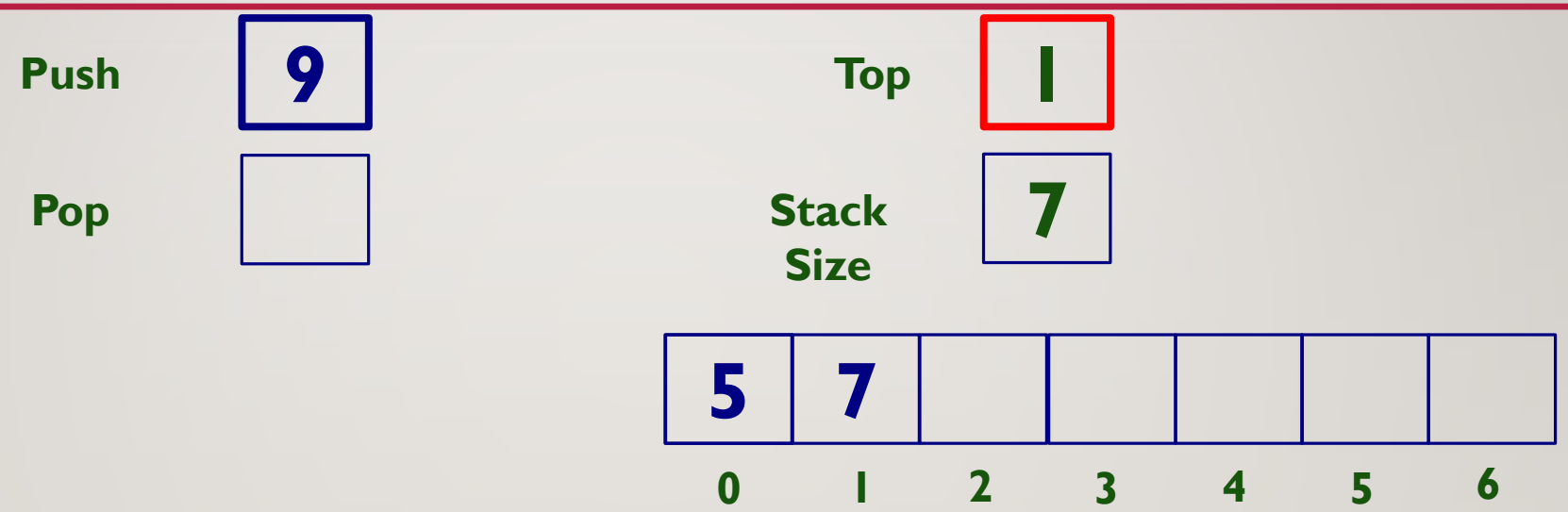
# DEMONSTRATION



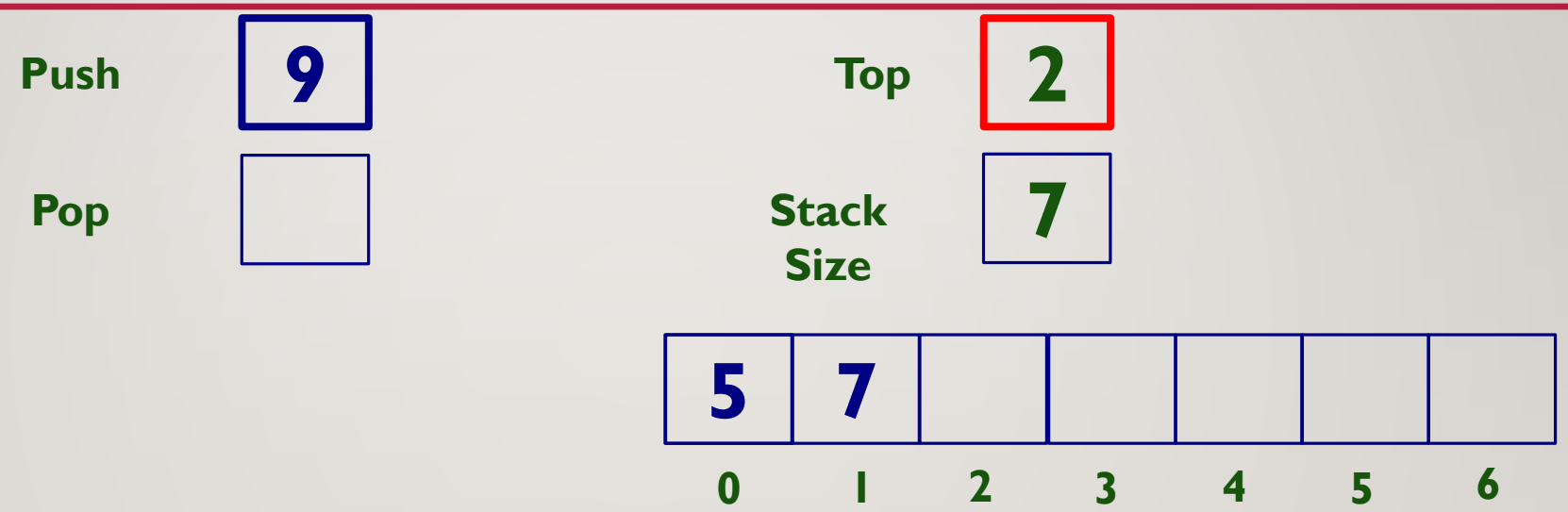
# DEMONSTRATION



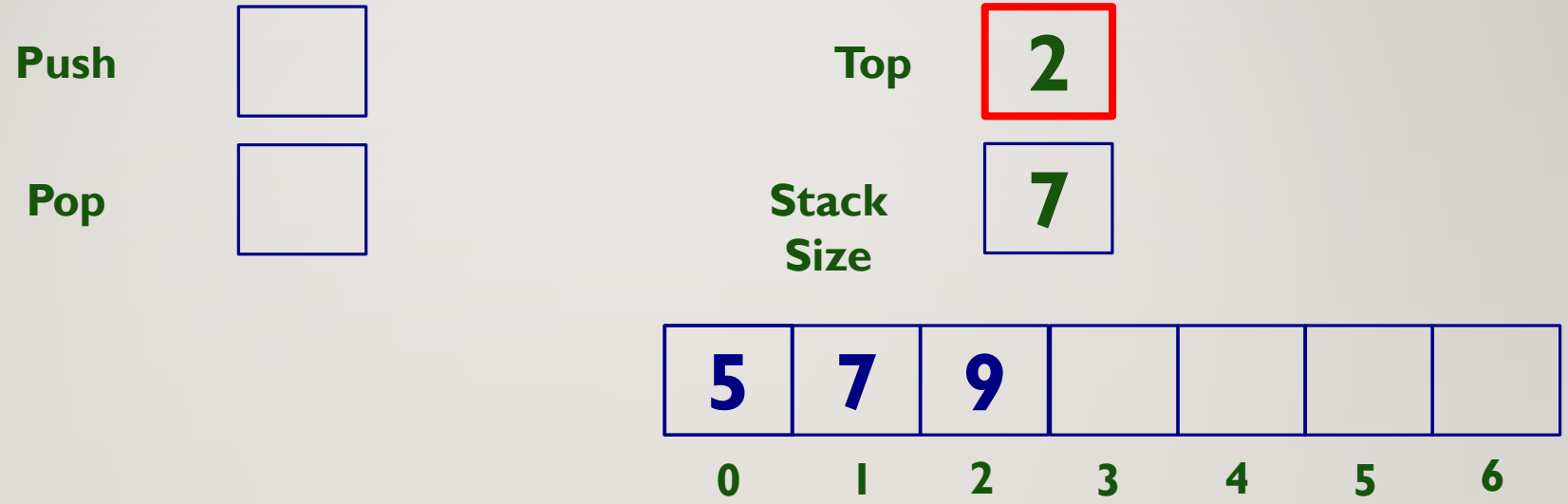
# DEMONSTRATION



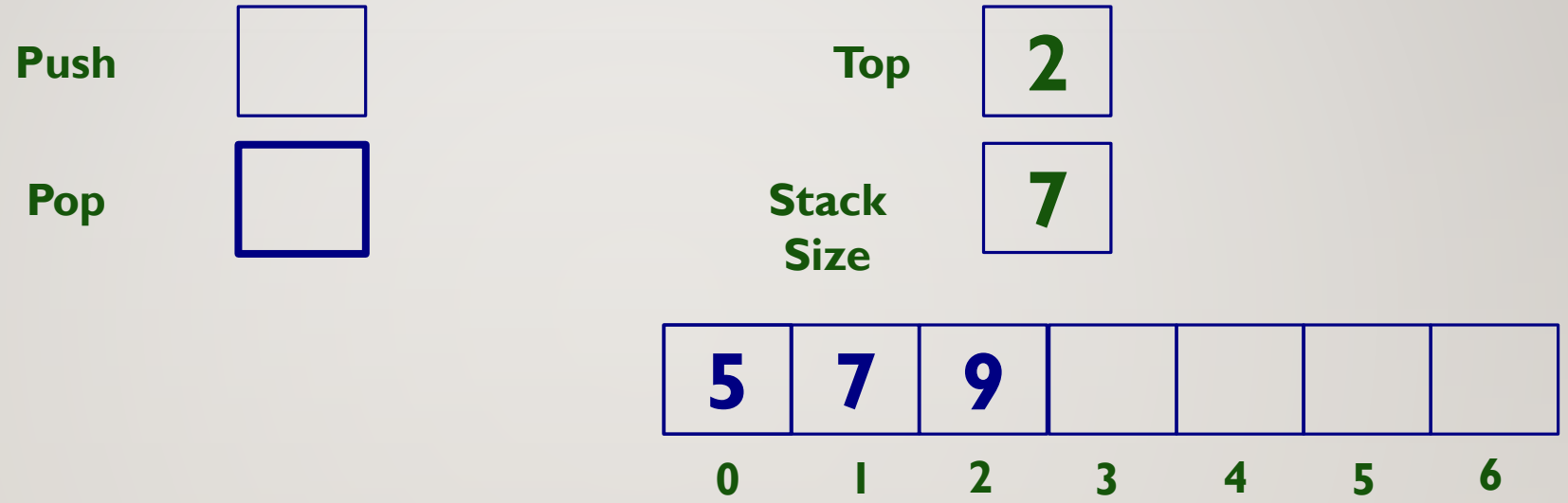
# DEMONSTRATION



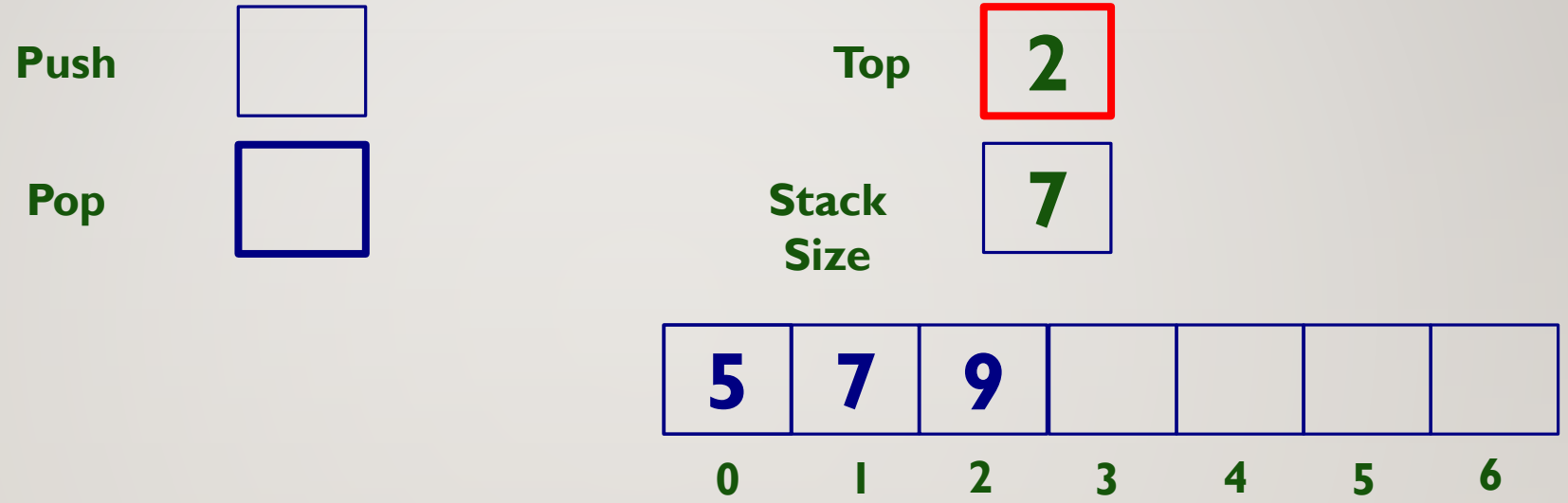
# DEMONSTRATION



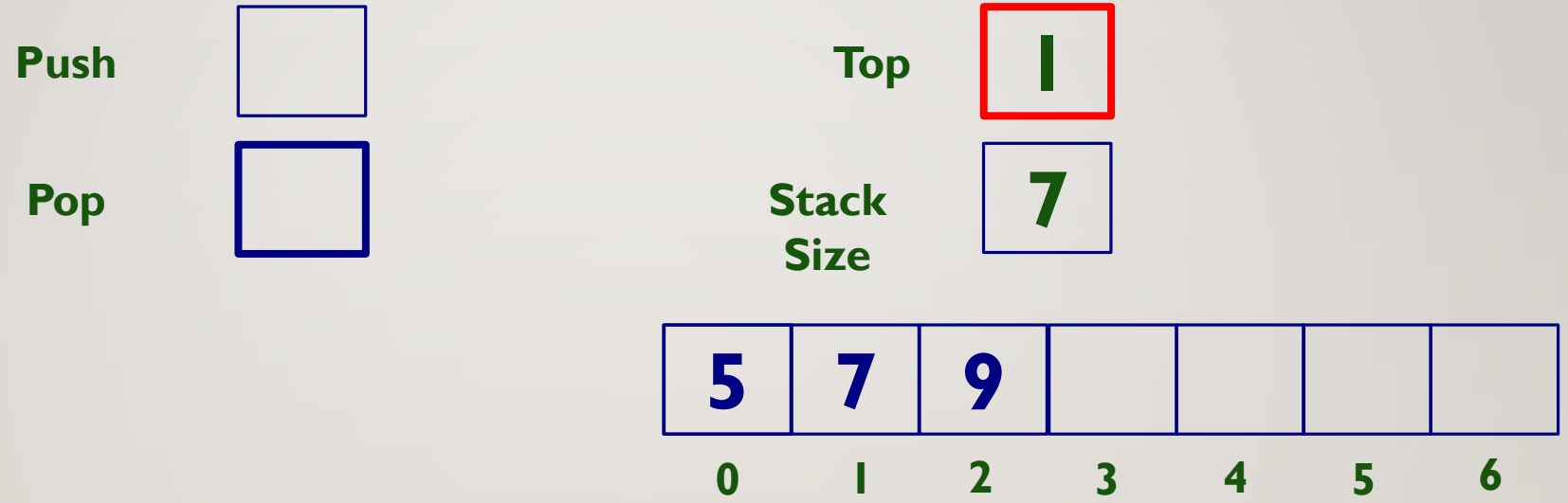
# DEMONSTRATION



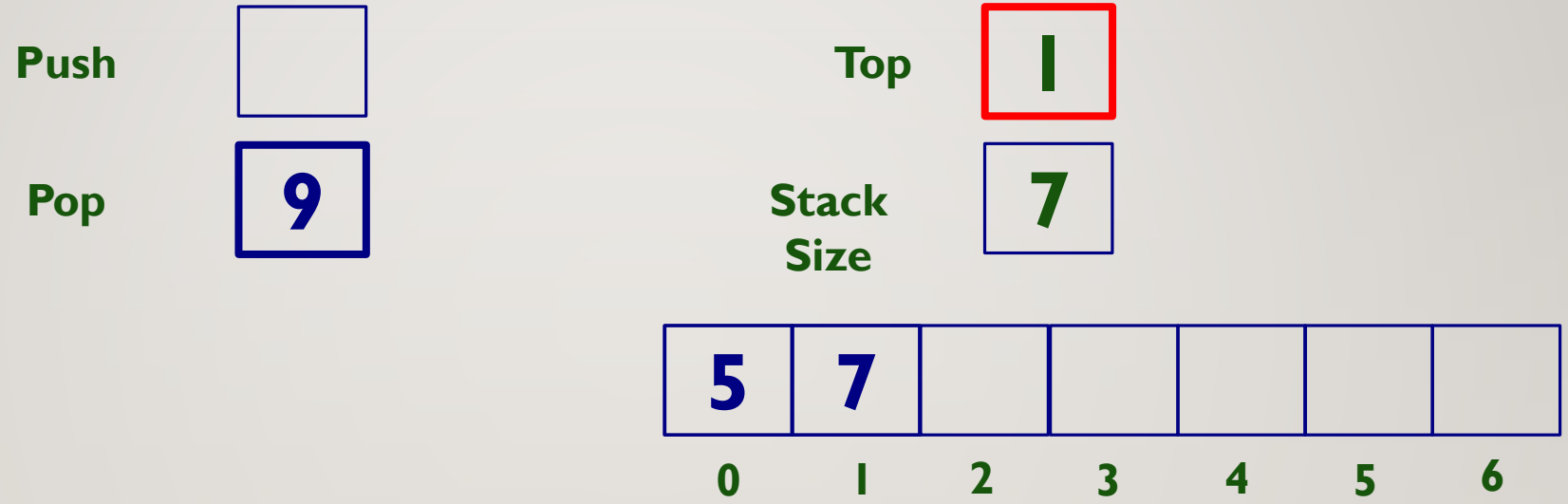
# DEMONSTRATION



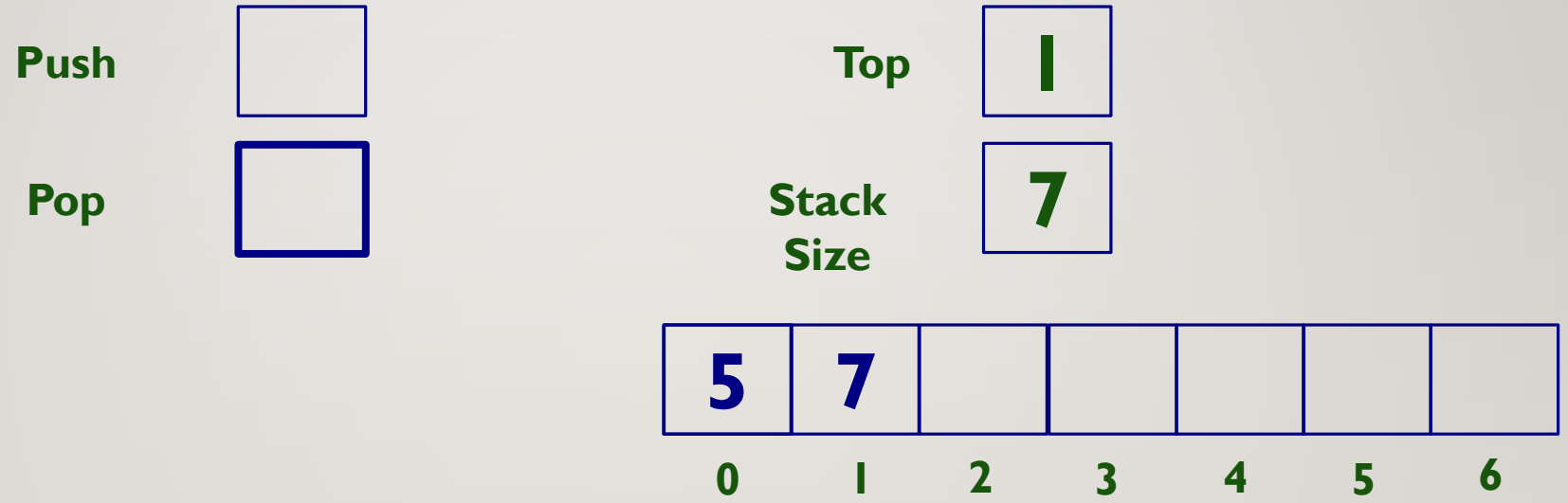
# DEMONSTRATION



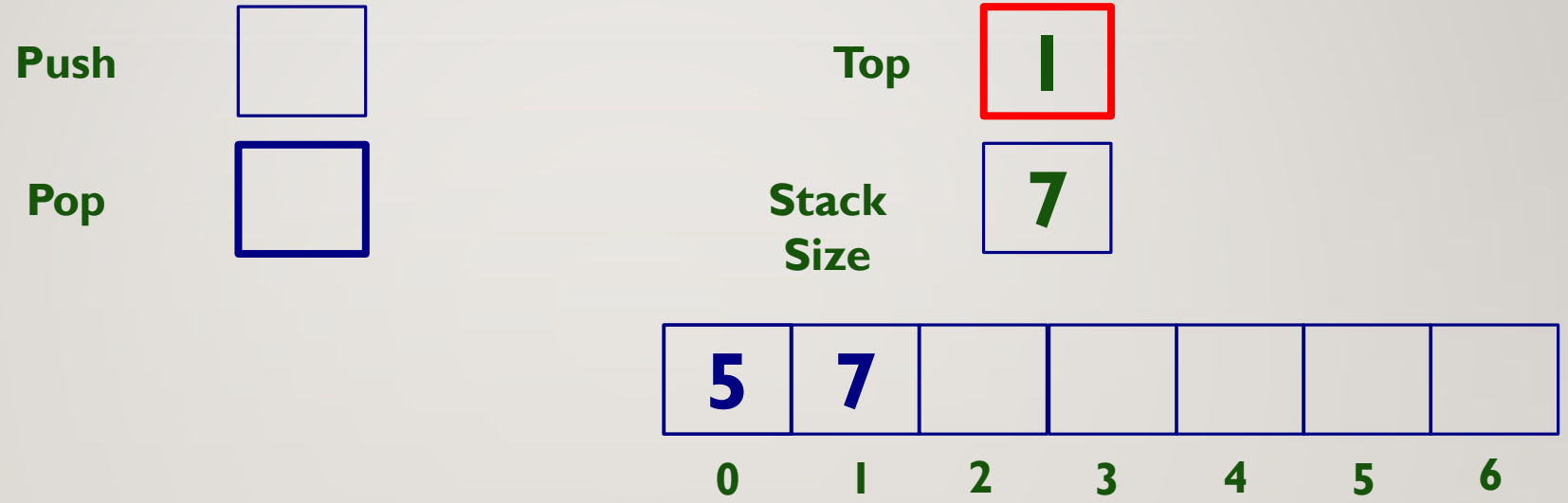
# DEMONSTRATION



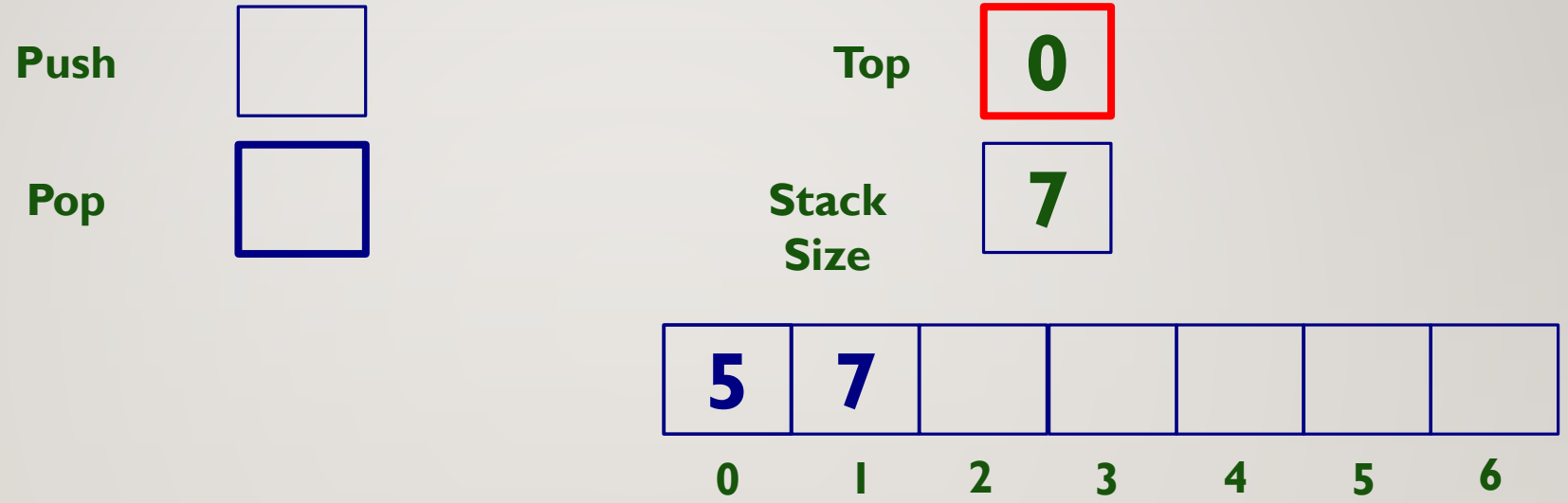
# DEMONSTRATION



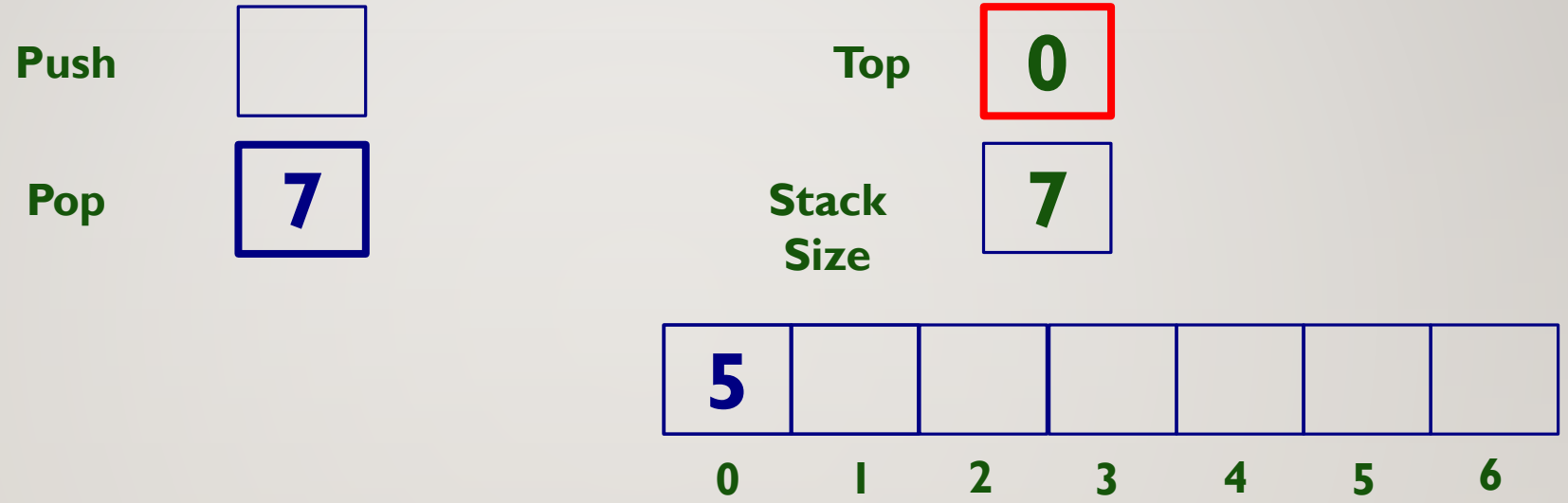
# DEMONSTRATION



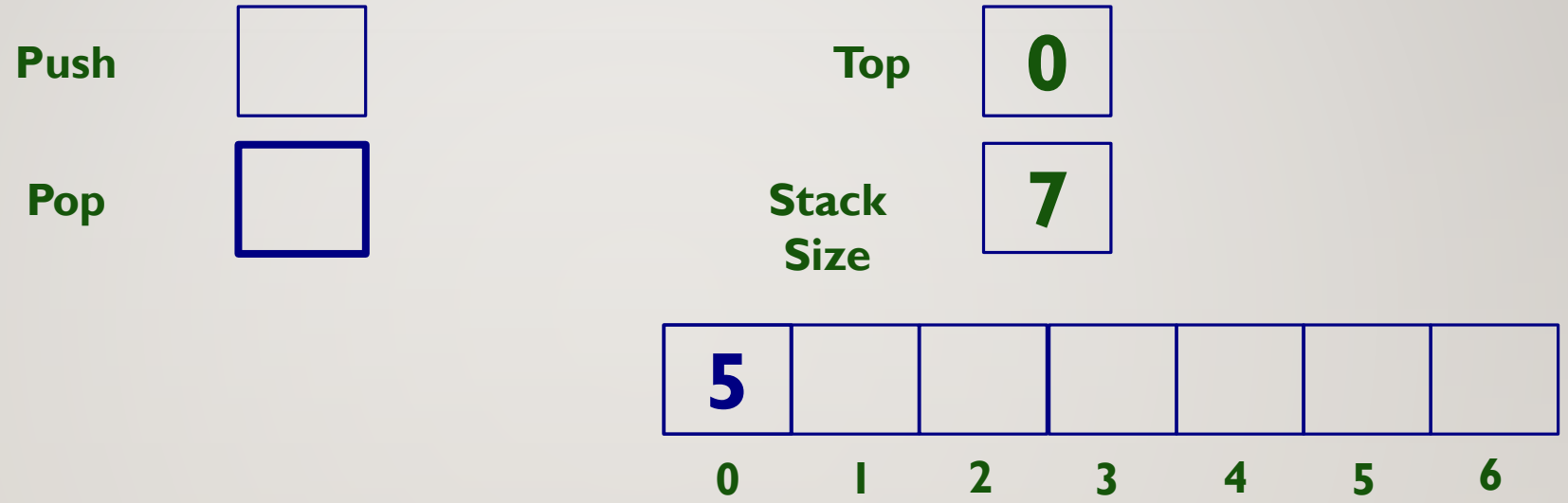
# DEMONSTRATION



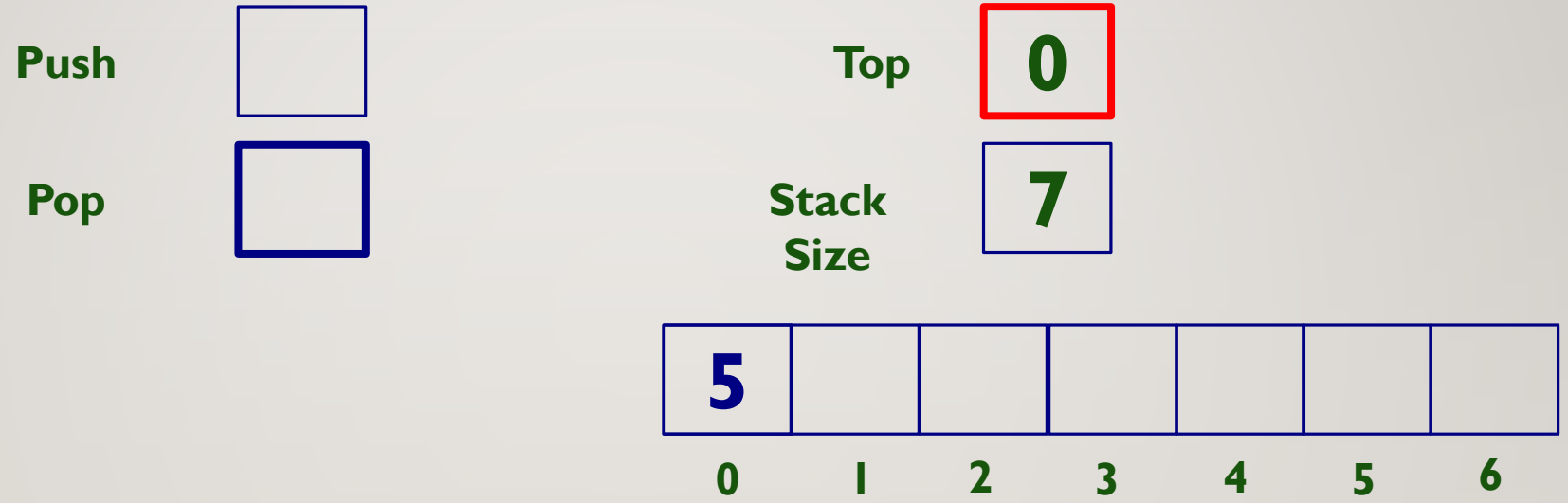
# DEMONSTRATION



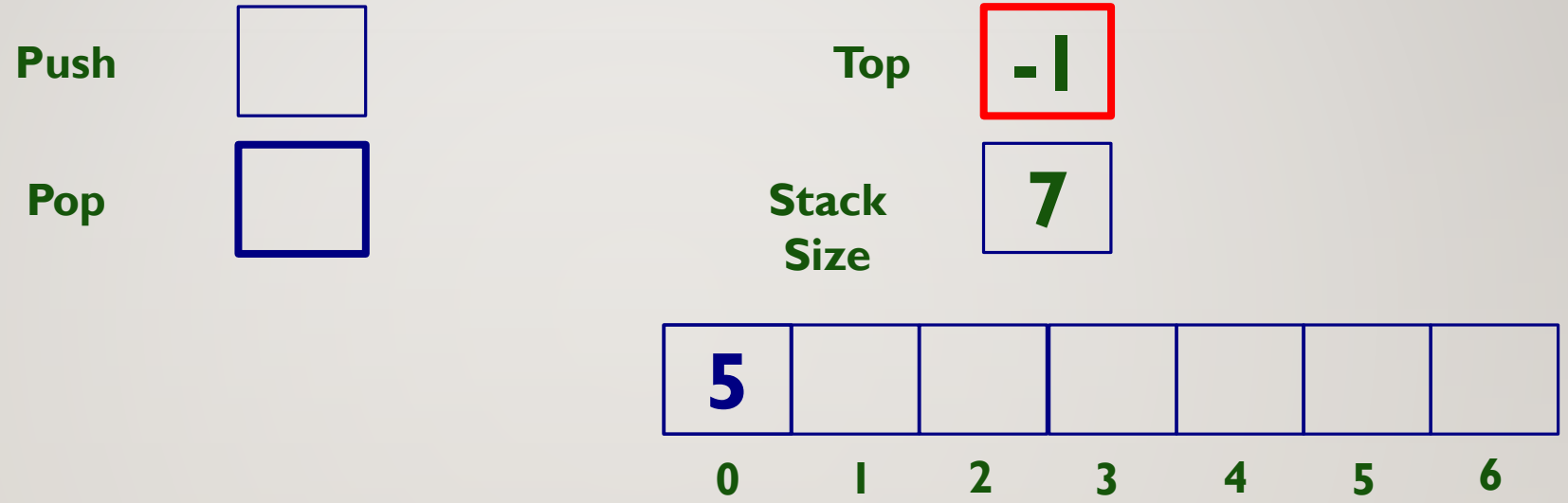
# DEMONSTRATION



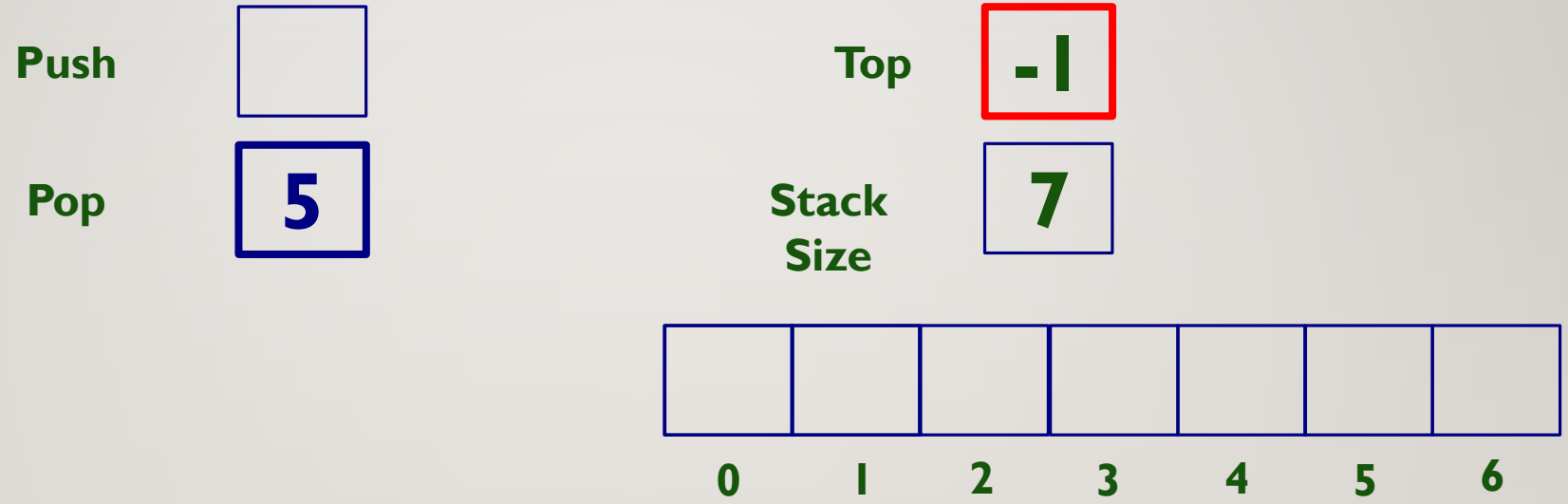
# DEMONSTRATION



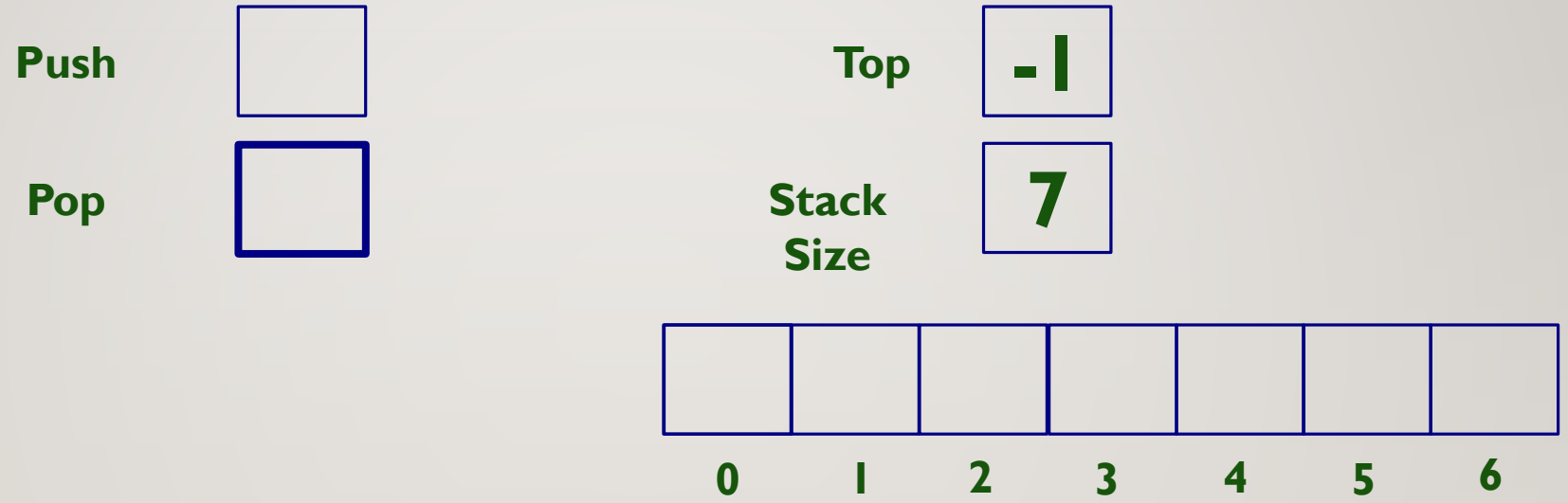
# DEMONSTRATION



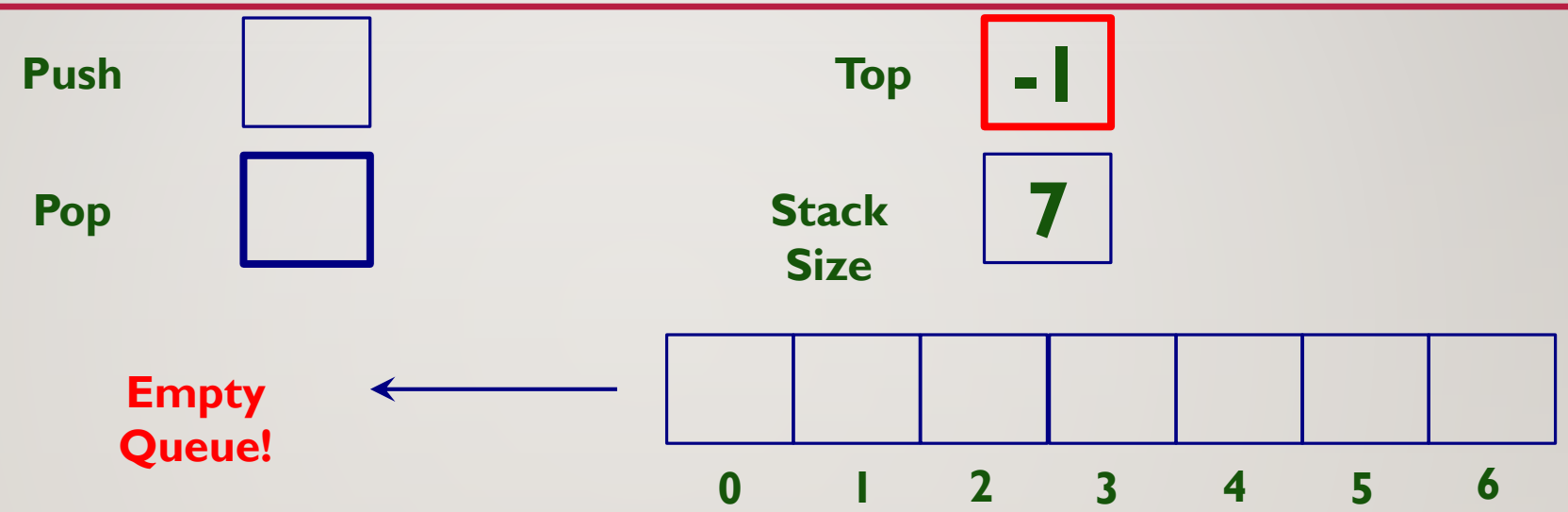
# DEMONSTRATION



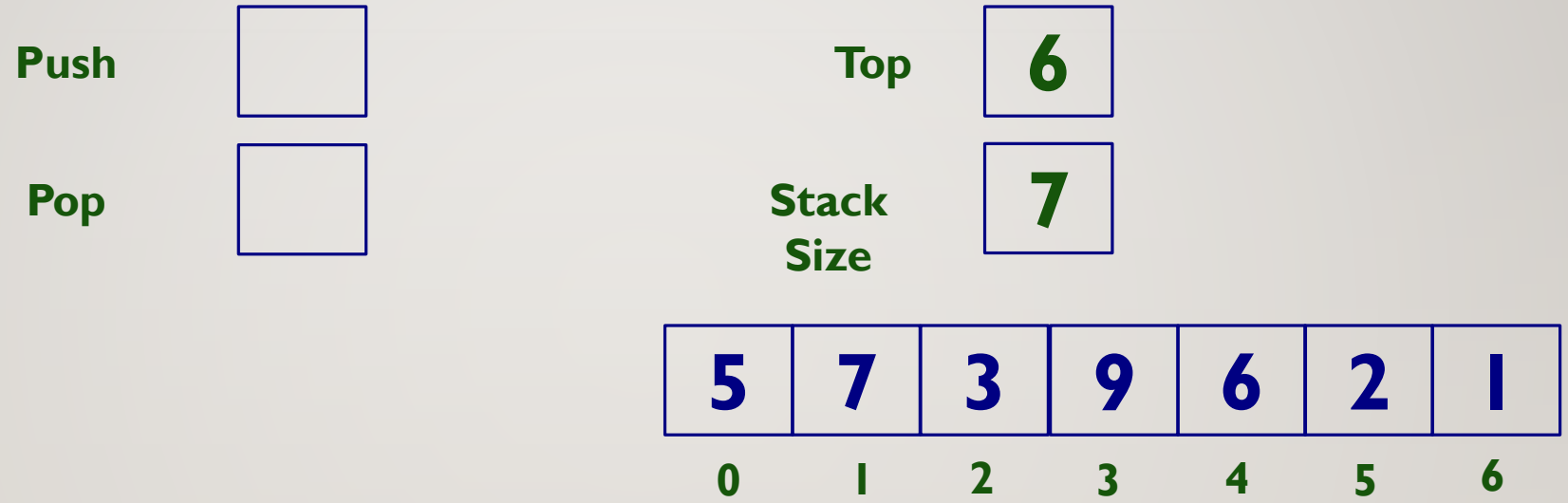
# DEMONSTRATION



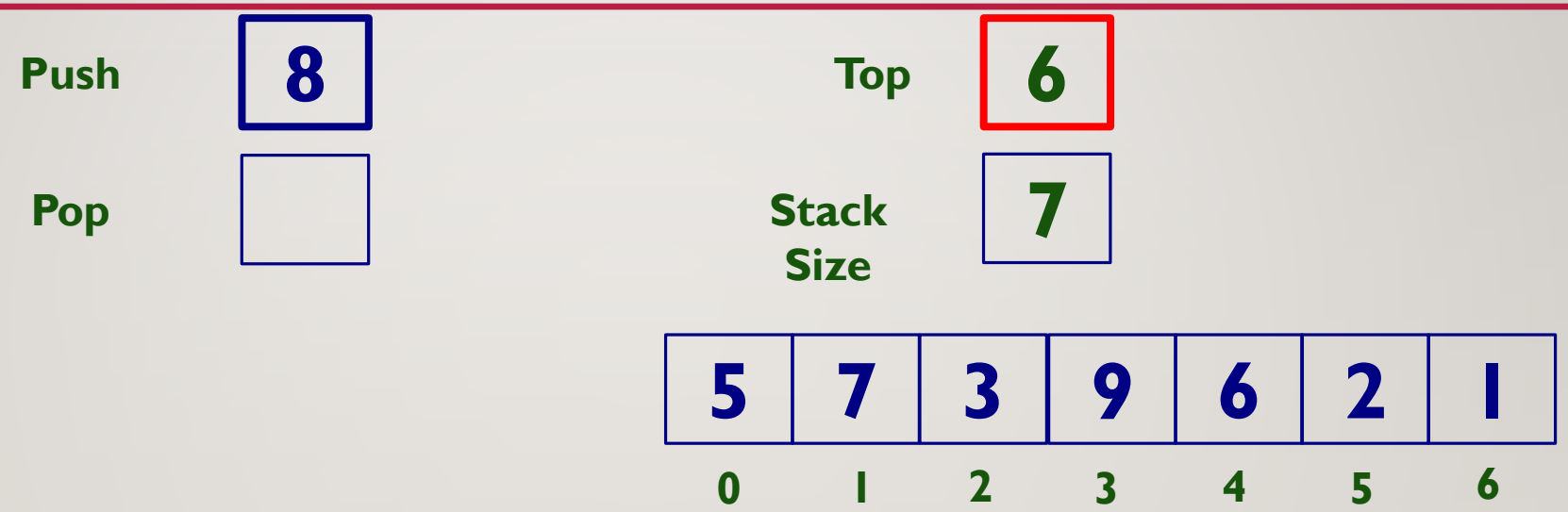
# DEMONSTRATION



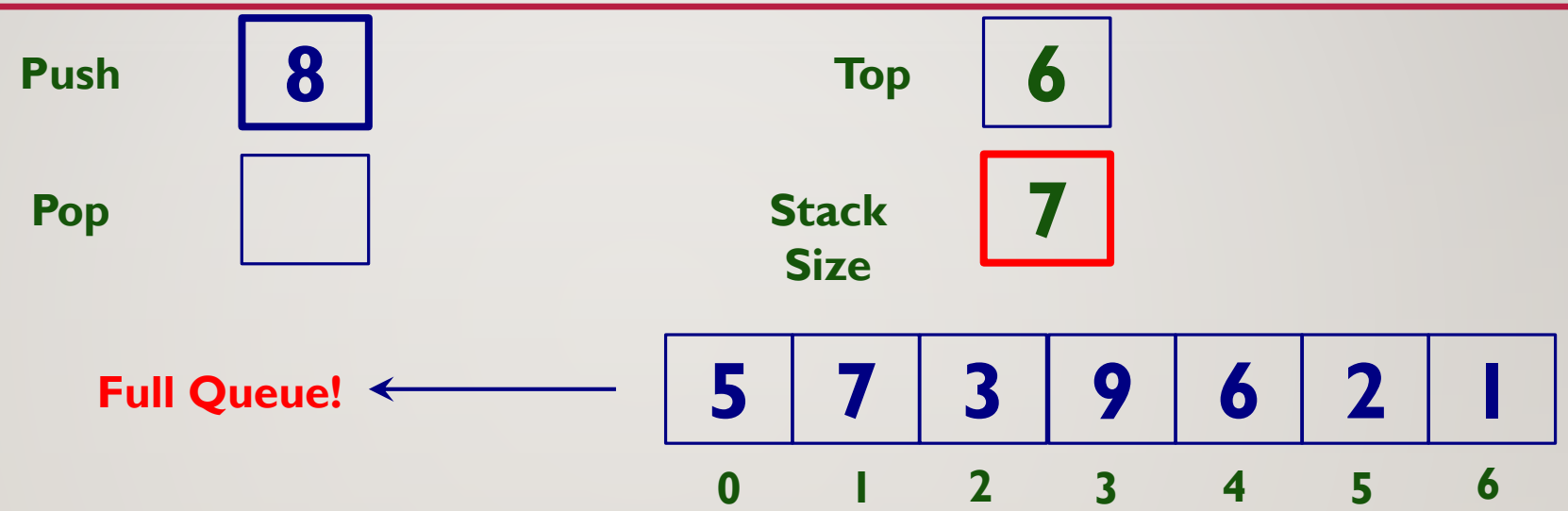
# DEMONSTRATION



# DEMONSTRATION



# DEMONSTRATION

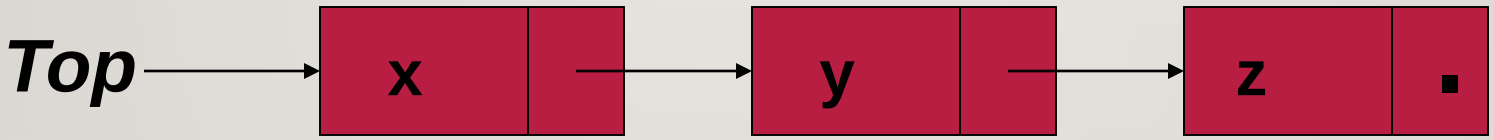


# STACK TEMPLATES

The stack class so far work with integers only. A stack template can be used to work with any data type.

# A LINKED-LIST IMPLEMENTATION OF STACKS

- Stack can *expand* or *shrink* with each PUSH or POP operation.
- PUSH and POP operate only on the header cell and the first cell on the list.



# LINKED LIST IMPLEMENTATION OF STACK

```
class Stack
{
    struct node
    {
            int data;
            node *next;
    }*top;
public:
    void Push(int newelement);
    void Pop(int &);
    bool IsEmpty();
};
```

```
void Stack::Push(int newelement)
{
   node *newptr;
   newptr=new node;
   newptr->data=newelement;
   newptr->next=top;
   top=newptr;
}

void Stack:Pop(int& returnvalue)
{
   if (IsEmpty()) { cout<<“underflow error”;  return;}
   tempptr=top;
   returnvalue=top->data;
   top=top->next;
   delete tempptr;
}
```

```
bool Stack::IsEmpty()
{
   if (top==NULL)
            return true;
   else
            return false;
}
```

# Program 3

```
// This program demonstrates the dynamic stack
// class DynIntClass.

#include <iostream.h>
#include "dynintstack.h“

void main(void)
{
   DynIntStack stack;
   int catchVar;


   cout << "Pushing 5\n";
   stack.push(5);
   cout << "Pushing 10\n";
   stack.push(10);
   cout << "Pushing 15\n";
   stack.push(15);
```

```
    cout << "Popping...\n";
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;
    stack.pop(catchVar);
    cout << catchVar << endl;

    cout << "\nAttempting to pop again... ";
    stack.pop(catchVar);
}
```

**Program Output**

```
Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5

Attempting to pop again... The stack is empty.
```

# APPLICATIONS OF STACKS

# ALGEBRAIC EXPRESSION

- An algebraic expression is a legal combination of operands and the operators.
  - Operand is the quantity on which a mathematical operation is performed.
  - Operator is a symbol which signifies a mathematical or logical operation.

# INFIX, POSTFIX AND PREFIX EXPRESSIONS

- **INFIX:** expressions in which operands surround the operator.
- **POSTFIX**: operator comes after the operands, also Known as Reverse Polish Notation (RPN).
- **PREFIX:** operator comes before the operands, also Known as Polish notation.
- Example
  - Infix: A+B-C  Postfix: AB+C-  Prefix: -+ABC

# EXAMPLES OF INFIX TO PREFIX AND POSTFIX

| Infix | PostFix | Prefix |
|---|---|---|
| A+B | AB+ | +AB |
| (A+B) * (C + D) | AB+CD+* | *+AB+CD |
| A-B/(C*D^E) | ? | ? |

# A+B*C IN POSTFIX

- Applying the rules of precedence, we obtained

A+B*C

A+(B*C)  Parentheses for emphasis

A+(BC*)  Convert the multiplication,

ABC*+  Postfix Form

# ( (A+B)*C-(D-E) ) $ (F+G)

Conversion to Postfix Expression

( (AB+)*C-(DE-) ) $ (FG+)

( (AB+C*)-(DE-) ) $ (FG+)

(AB+C*DE--) $ (FG+)

AB+C*DE- -FG+$

Exercise: Convert the following to Postfix

( A + B ) * ( C – D)

A $ B * C – D + E / F / (G + H)

# WHY DO WE NEED PREFIX/POSTFIX?

- Appearance may be misleading, INFIX notations are not as simple as they seem

- To evaluate an infix expression we need to consider
  - Operators' Priority
  - Associative property
  - Delimiters

# WHY DO WE NEED PREFIX/POSTFIX?

- Infix Expression Is Hard To Parse and difficult to evaluate.
- Postfix and prefix  do not rely on operator priority and are easier to parse.

# WHY DO WE NEED PREFIX/POSTFIX?

- An expression in infix form is thus converted into prefix or postfix form and then evaluated without considering the operators priority and delimiters.

# CONVERSION OF INFIX EXPRESSION TO POSTFIX

A+B*C =ABC*+

There must be a precedence function. `prcd(op1, op2), where op1 and op2 are chars representing operators.`

This function returns TRUE if op1 has precedence over op2 when op1 appears to the left of op2 in an infix expression without parenthesis. prcd(op1,op2) returns FALSE otherwise.

`prcd('*','+') and prcd('+','+') are TRUE whereas prcd('+','*') is FALSE.`

# ALGORITHM TO CONVERT INFIX TO POSTFIX

```
opstk = the empty stack;
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
            add symb to the postfix string
    else {
            while (!empty(opstk) &&
    prcd(stacktop(opstk),symb) ) {
                topsymb = pop(opstk);
                add topsymb to the postfix string;
            } /* end while */
            push(opstk, symb);
    } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
    topsymb = pop(opstk);
    add topsymb to the postfix string;
} /* end while */
```

**Example-1: A+B*C**

| symb | Postfix string | opstk |
|---|---|---|
| **A** | **A** | |
| **+** | **A** | **+** |
| **B** | **AB** | **+** |
| ***** | **AB** | **+ *** |
| **C** | **ABC** | **+ *** |
| | **ABC*** | **+** |
| | **ABC*+** | |

# ALGORITHM TO CONVERT INFIX TO POSTFIX

```
opstk = the empty stack;
while (not end of input) {
    symb = next input character;
    if (symb is an operand)
            add symb to the postfix string
    else {
            while (!empty(opstk) &&
    prcd(stacktop(opstk),symb) ) {
                topsymb = pop(opstk);
                add topsymb to the postfix string;
            } /* end while */
            push(opstk, symb);
    } /* end else */
} /* end while */
/* output any remaining operators */
while (!empty(opstk) ) {
    topsymb = pop(opstk);
    add topsymb to the postfix string;
} /* end while */
```

**Example-1: A*B+C**

| symb | Postfix string | opstk |
|---|---|---|
| **A** | **A** | |
| ***** | **A** | ***** |
| **B** | **AB** | ***** |
| **+** | **AB*** | **+** |
| **C** | **AB*C** | **+** |
| | **AB*C+** | |