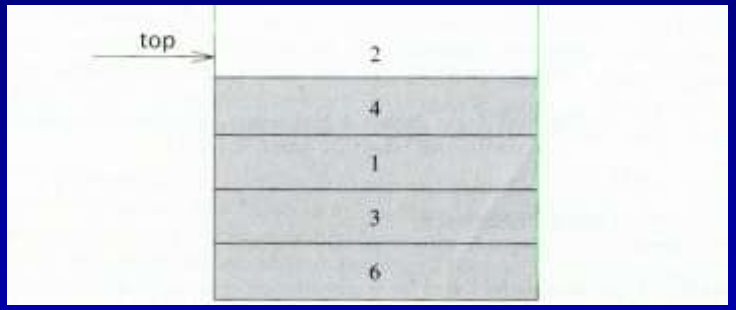
Stacks

Data Structures

What is a stack?

- A stack is a list with the restriction
  - that insertions and deletions can only be performed at the *top* of the list



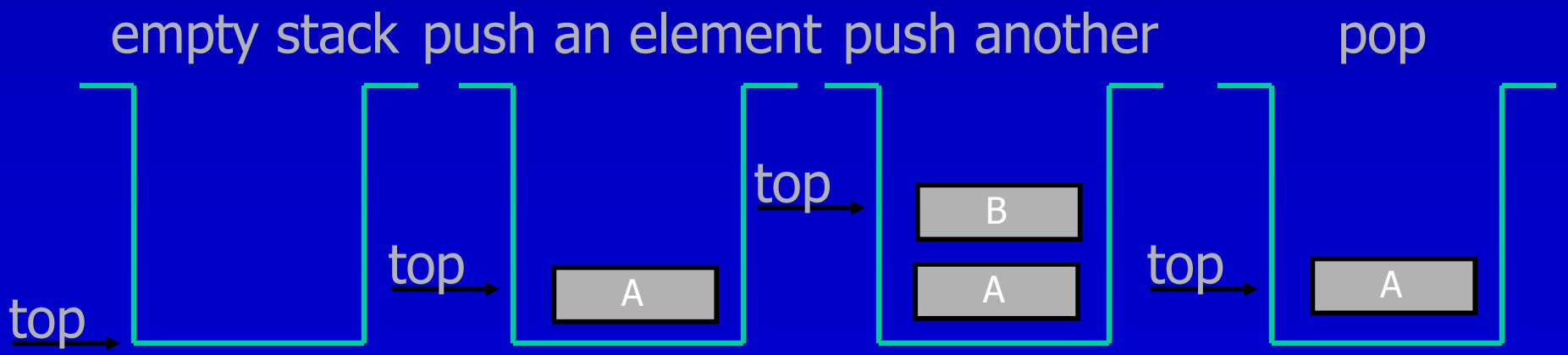
  - The other end is called bottom
- Fundamental operations:
  - Push: Equivalent to an insert
  - Pop: Deletes the most recently inserted element
  - Top: Examines the most recently inserted element

Stack

- Stacks are less flexible
  - ✓but are more efficient and easy to implement
- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved

Push and Pop

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack
- Pop
  - Remove the element at the top of the stack



Implementation of Stacks

- Any list implementation could be used to implement a stack
  - Arrays (static: the size of stack is given initially)
  - Linked lists (dynamic: never become full)
- We will explore implementations based on array and linked list
- Let's see how to use an array to implement a stack first

`Stack` class

- Attributes of `Stack`
  - `maxTop`: the max size of stack
  - `top`: the index of the top element of stack
  - `values`: point to an array which stores elements of stack
- Operations of `Stack`
  - `IsEmpty`: return true if stack is empty, return false otherwise
  - `IsFull`: return true if stack is full, return false otherwise
  - `Top`: return the element at the top of stack
  - `Push`: add an element to the top of stack
  - `Pop`: delete the element at the top of stack
  - `DisplayStack`: print all the data in the stack

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

```
void Stack::Push(const double x) {
      if (IsFull())
            cout << "Error: the stack is full." << endl;
      else
            values[++top]     =     x;
}
```

Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don't forgot to decrement `top`

```
double Stack::Pop() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else {
            return values[top--];
      }
}
```

Stack Top

- `double Top()`
  - Return the top element of the stack
  - Unlike `Pop`, this function does not remove the top element

```
double Stack::Top() {
      if (IsEmpty()) {
            cout << "Error: the stack is empty." << endl;
            return -1;
      }
      else
            return values[top];
}
```

Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart
  - e.g. [( )] is legal, but [( ]) is illegal
- Algorithm

(1) Make an empty stack.

(2) Read characters until end of file

i. If the character is an opening symbol, push it onto the stack

ii. If it is a closing symbol, then if the stack is empty, report an error

iii. Otherwise, pop the stack. If the symbol popped is not the corresponding opening symbol, then report an error

(3) At end of file, if the stack is not empty, report an error

Postfix Expressions

- Calculate 4.99 * 1.06 + 5.99 + 6.99 * 1.06
  - Need to know the precedence rules
- Postfix (reverse Polish) expression
  - 4.99 1.06 * 5.99 + 6.99 1.06 * +
- Use stack to evaluate postfix expressions
  - When a number is seen, it is pushed onto the stack
  - When an operator is seen, the operator is applied to the 2 numbers that are popped from the stack. The result is pushed onto the stack
- Example
  - evaluate 6 5 2 3 + 8 * + 3 + *
- The time to evaluate a postfix expression is O(N)
  - processing each element in the input consists of stack operations and thus takes constant time