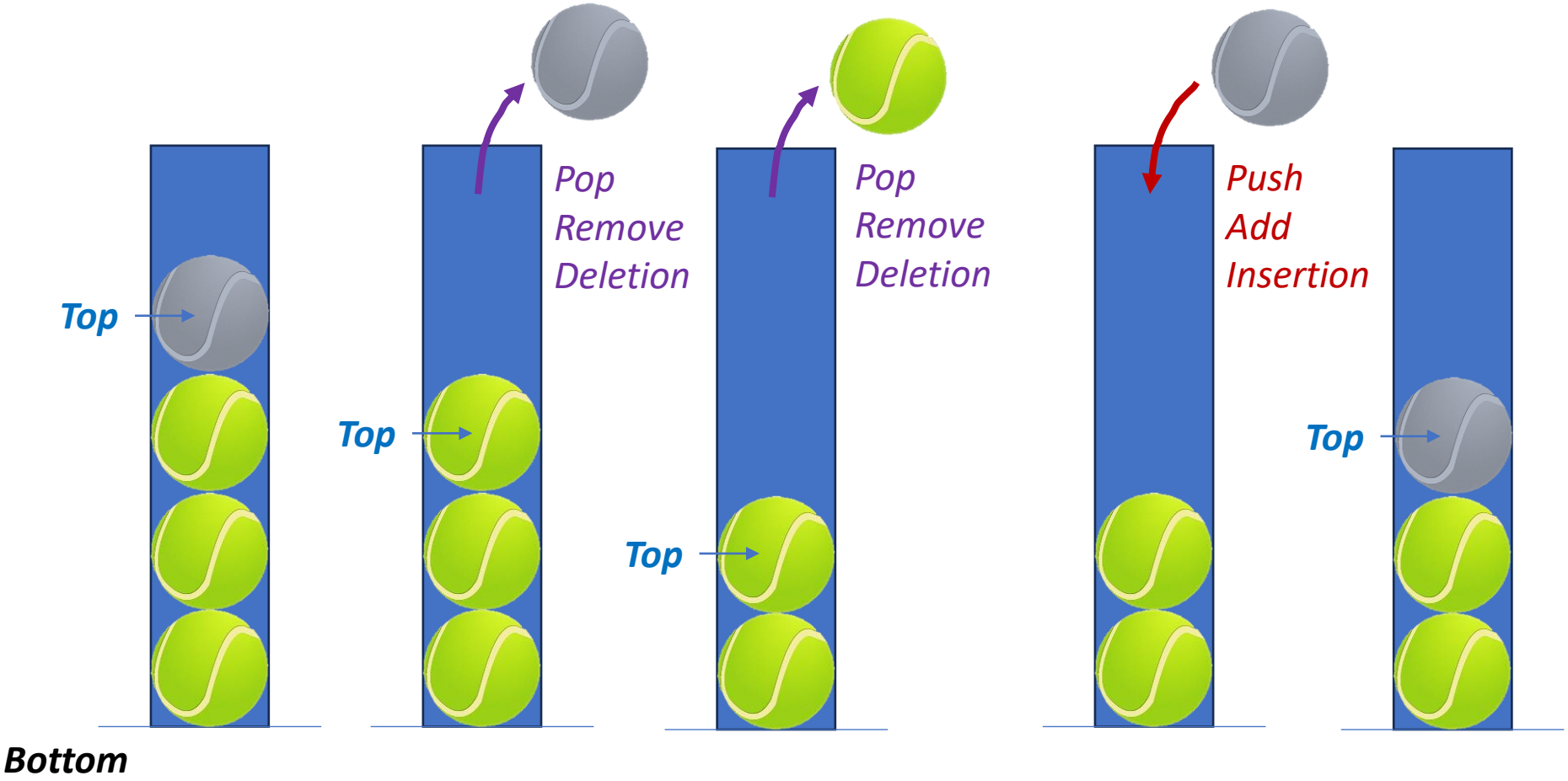
Stacks

Ch 3.1 – 3.2

Stack is a special case of ordered list

- Insertions and deletions are made at one end, called the *top*.
- Example: Take tennis balls from a can to play and then return them.



Last-In-First-Out (LIFO)

Application of stack: Balanced Brackets

- Check whether the brackets are balanced.
- Examples:
  - “]” → False
  - “(a+b)*c-d” → True
  - “{[(a+b)*c+d-e]/(f+g)-(h+j)*(k-l)}/(m-n)” → True
  - “{[(a+b)*c+d-e]” → False

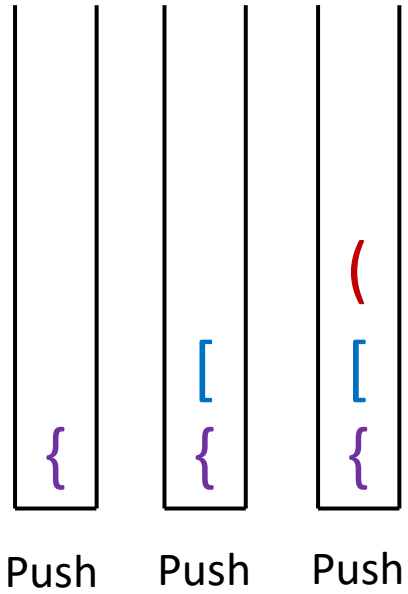
- Steps:
  - Scan the expression from left to right
  - When a left bracket, “[”, “{”, or “(”, is encountered, push it into stack.
  - When a right bracket, “]”, “}”, or “)”, is encountered, pop the top element from stack and check whether they are matched.
    - If the right bracket and the pop out element are not matched, return False.
  - After scanning the whole expression, if the stack is not empty, return False.

Example 1

- {[(a+b)*c+d-e]

→

Scan

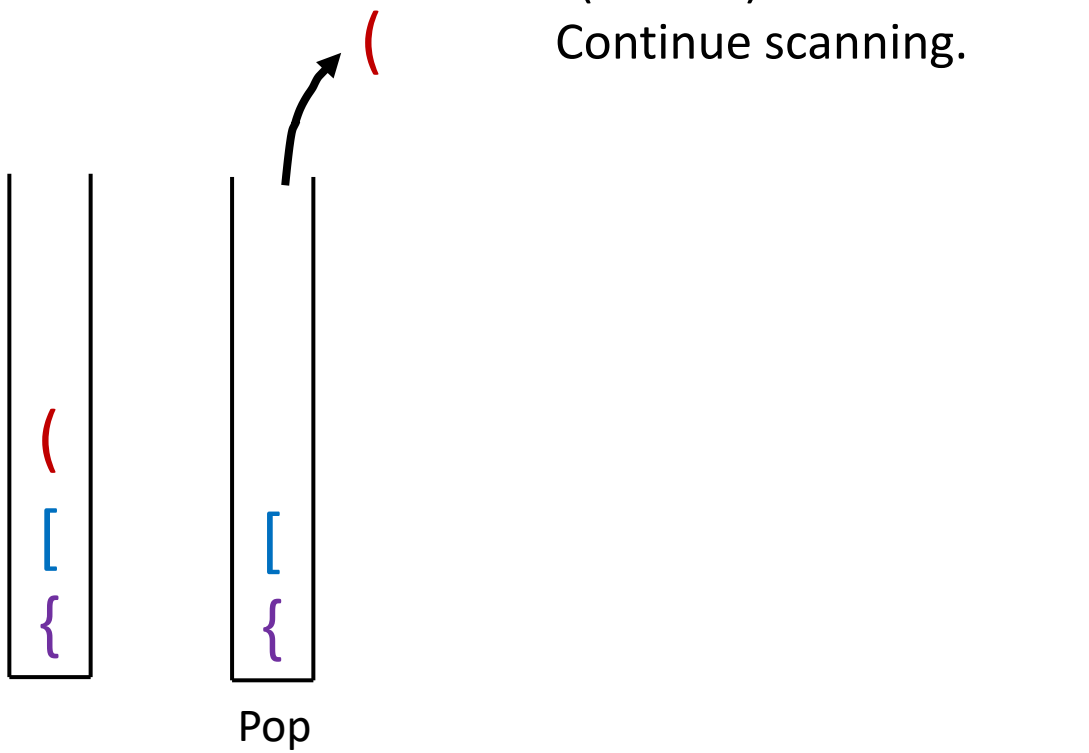


Example 1

- {[(a+b)*c+d-e]

Scan → Encounter a right bracket “)”

“(” and “)” are Matched.
Continue scanning.

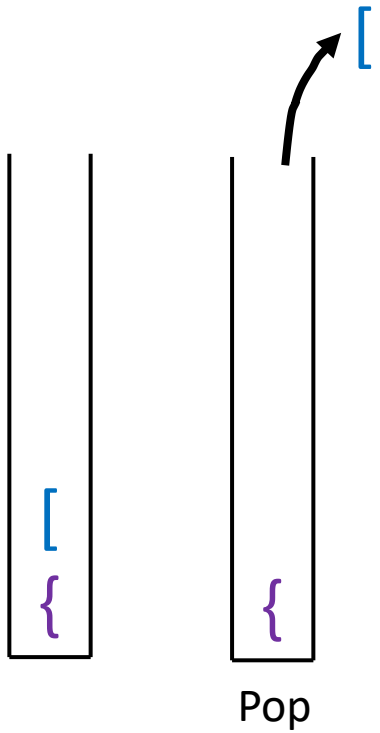


Example 1

- {[(a+b)*c+d-e]

Scan → Encounter a right bracket “]”

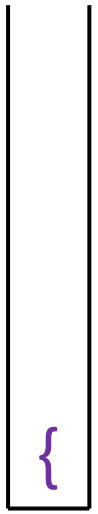
“[” and “]” are Matched.
Continue scanning.



Example 1

- {[(a+b)*c+d-e]

Scan → Finish scanning



The stack is not empty.
A left bracket “{” remains, so the brackets in the expression are unbalanced.

Group Discussion

- Examine whether the brackets in the following expression are balanced.

{[(a+b)*c+d-e]/(f+g)-(h+j)*k-l)}/(m-n)

- Q3: Are the brackets in the expression balanced?
- Q4: How many times of push and pop are required?
- Q5: Please list the order of push and pop.

Example:

{[(a+b)*c+d-e]:

- **Not** balanced
- 3 times of push and 2 times of pop.
- Push, push, push, pop, pop

Please reply your answers of Q3, Q4, and Q5 via the following link:

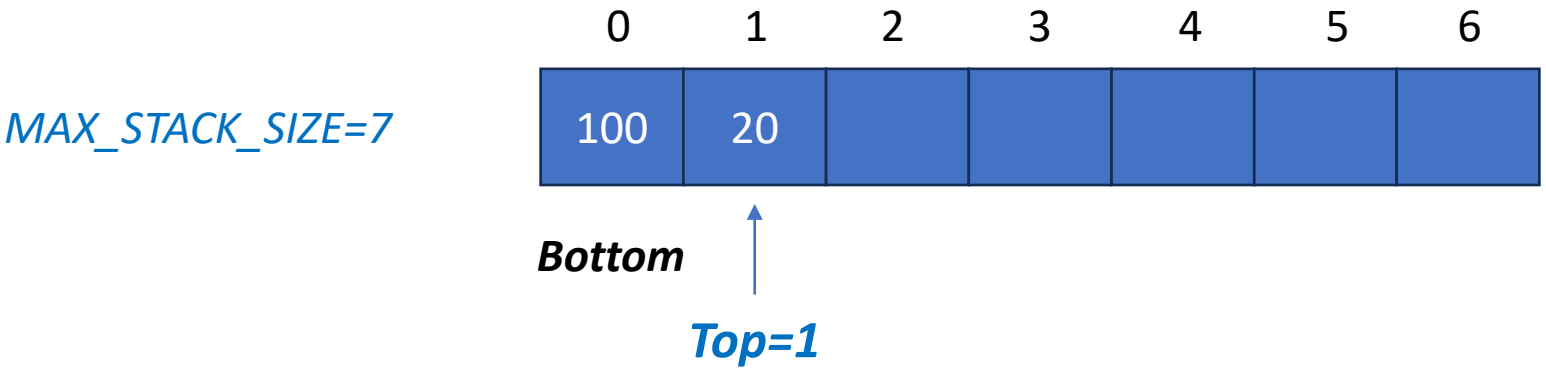
Group members: 1~3 people

Operations of stacks

- **CreateS**: create an empty stack
- **IsFull**: return True if the stack is full
- **IsEmpty**: return True if the stack is empty
- **Top**: return top element of stack
- **Push**: insert an element into top of stack
- **Pop**: remove and return the element at the top of the stack

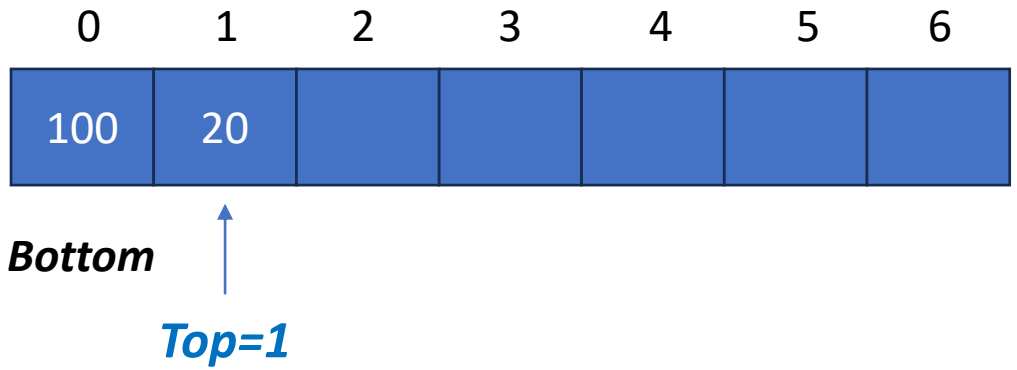
Implementation of stacks

- Using a 1D array to represent a stack. *stack[MAX_STACK_SIZE]*



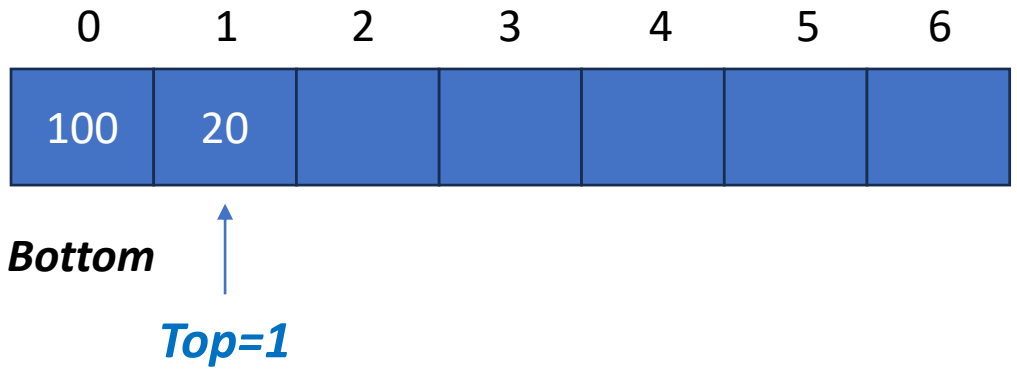
- The first, bottom, element of the stack is stored in *stack[0]*.
- An integer variable, *top*, points to the top element in the stack.
  - Empty stack: *top* = -1
- Stack elements are stored in *stack[0]* through *stack[top]*.

Using variable *top* to implement operations



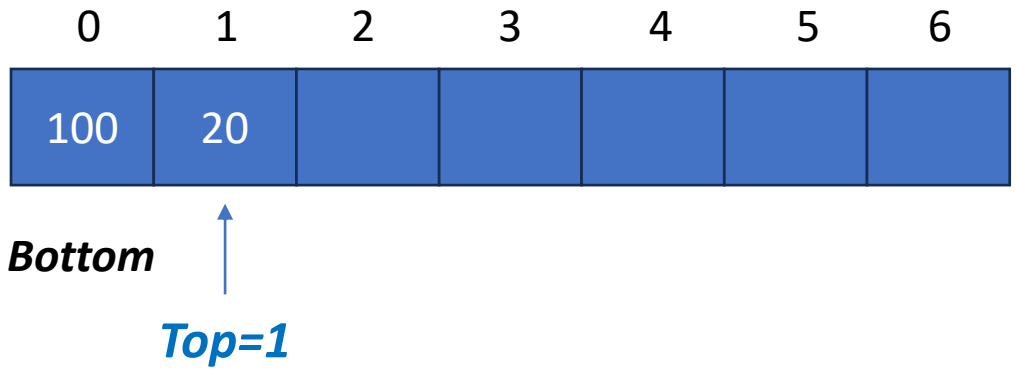
- IsEmpty(): check whether *top* >= 0
- IsFull(): check whether *top* == MAX_STACK_SIZE-1
- Top(): if not empty, return stack[*top*]

Push: Add an item to a stack




```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE - 1)
       StackFull();
   /* add at stack top */
   stack[++top] = item;
}
```

Pop: Add an item to a stack




```
element pop()
{/* delete and return the top element from the stack*/
   if (top == -1)
       return StackEmpty(); /* returns an error code */

   return stack[top--];
}
```

Determine MAX_STACK_SIZE at compile time

- How?
- To create an empty stack stack[MAX_STACK_SIZE], determining MAX_STACK_SIZE is necessary.
- Using dynamic arrays can overcome this problem.

Stacks using dynamic arrays

- Using *malloc* to create an empty stack.

```
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity = 1;
int top = -1;
```

- When stack is full, we can double array capacity. → array doubling

```
void stackFull()
{
   REALLOC(stack, 2*capacity*sizeof(*stack));
   capacity *= 2;
{
```