Spring 2019 (Course Number 009952)

Data Structures and Lab
(Lecture 07: Stacks)

Prof. A. P. Shrestha, Ph.D.
Dept. of Computer Science and Engineering, Sejong University

Last Class

- XOR and Circular linked list
- Josephus Problem

Today

- Stacks and its implementation
- Applications of Stack

Next class

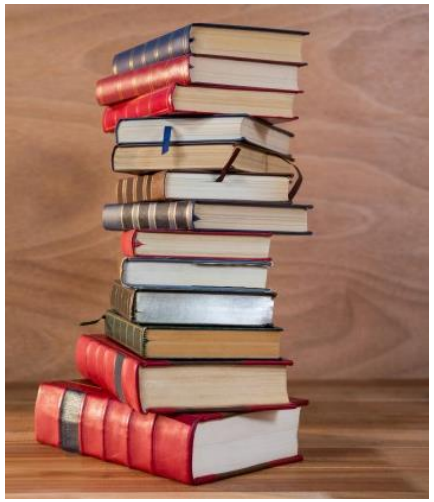
- Infix to postfix using stack
- Postfix Evaluation

7.1.1 Stack

- Linear data structure which stores a sequence of values
- Accessible at only one end of the sequence
- Follows Last-in-first-out (LIFO) or First-in-last-out (FILO) order
- The last thing we added is the first thing that gets pulled off.
- Add new items at the top
- Remove an item at the top

7.1.2 Stacks: Analogy

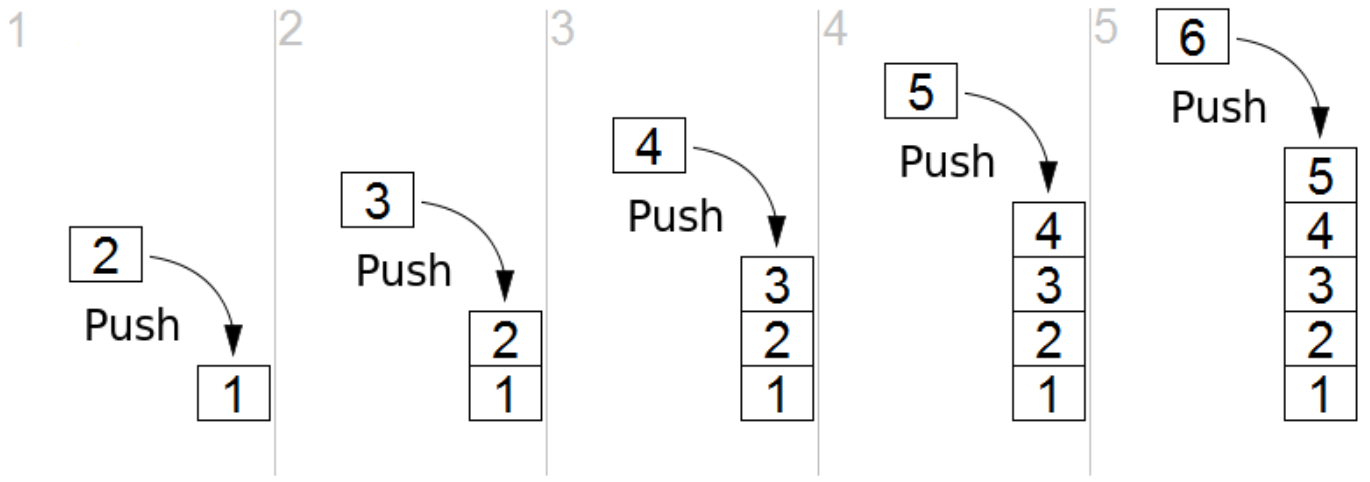
- Can only add plate/book/ball at the top
- Can remove plate/book/ball at the top



7.1.3 Stack Operations-Push

Push:

- insert an item into the stack.
- If the stack is full, then it is said to be an *Overflow* condition.

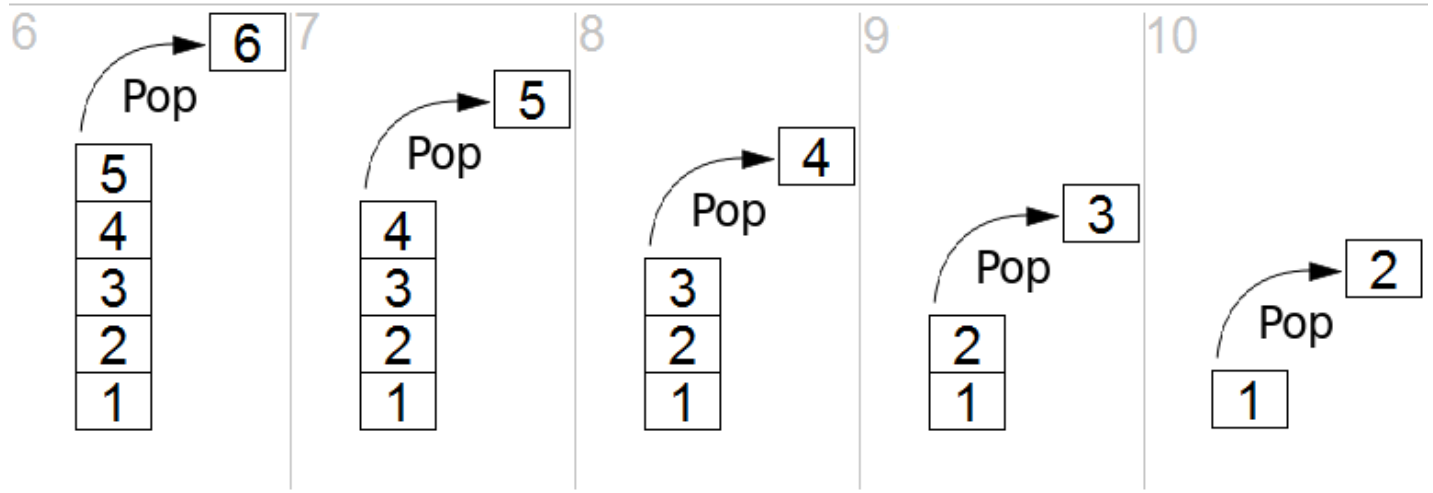


Representation of a stack runtime with *push* operations (Wikipedia)

7.1.4 Stack Operations: Pop

Pop:

- Removes an item from the stack.
- The items are popped in the reversed order in which they were inserted
- If the stack is empty, then it is said to be an Underflow condition.



Simple representation of a stack runtime with *pop* operations (Wikipedia)

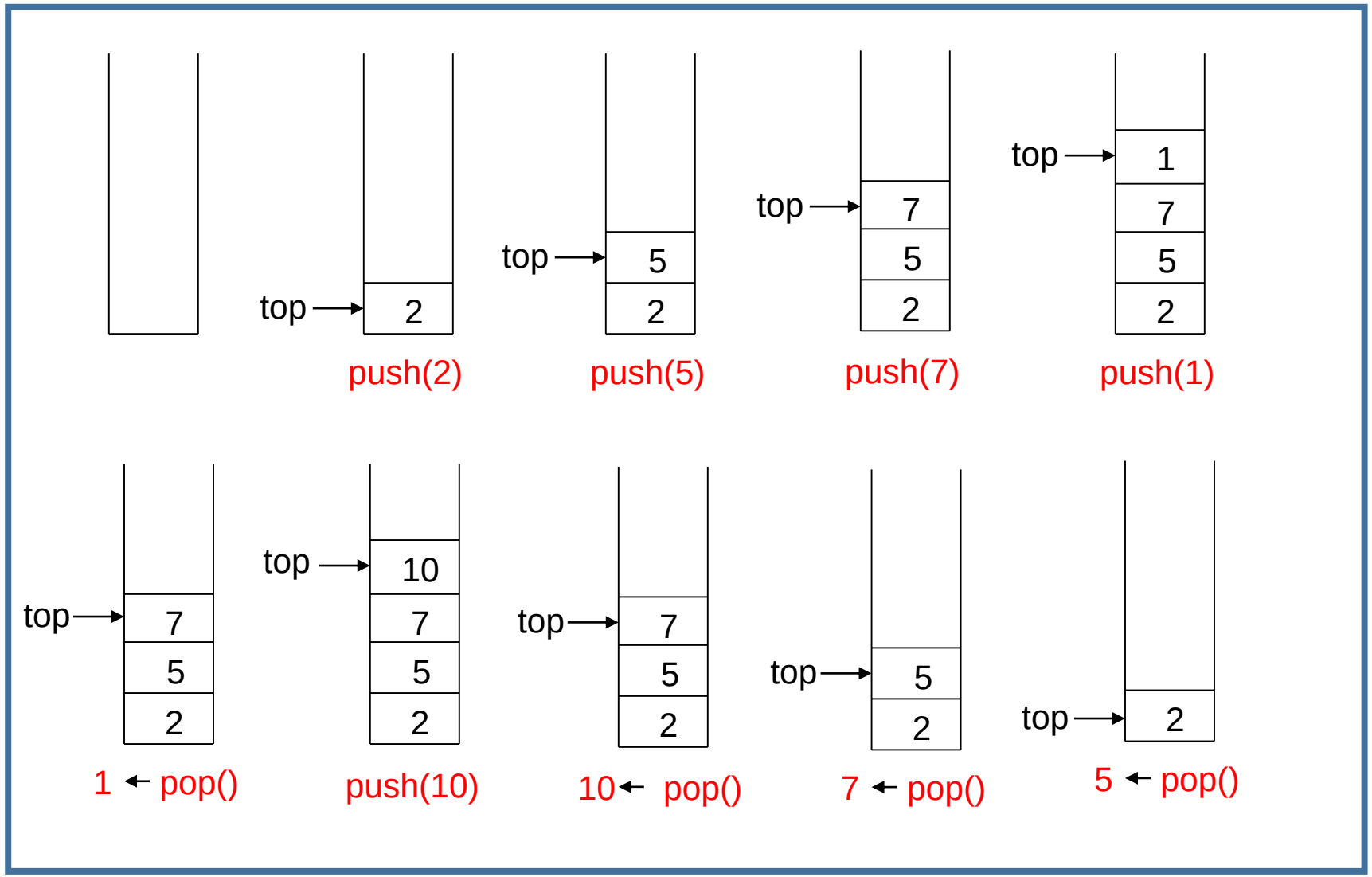
7.1.5 Stack Operations (Other)

- Peek() or Top() :
  - Returns the top-most element of stack without removing it.

- isFull():
  - A Boolean operation needed for static stacks.
  - Returns true if the stack is full. Otherwise, returns false

- isEmpty():
  - A Boolean operation needed for all stacks.
  - Returns true if the stack is empty. Otherwise, returns false.

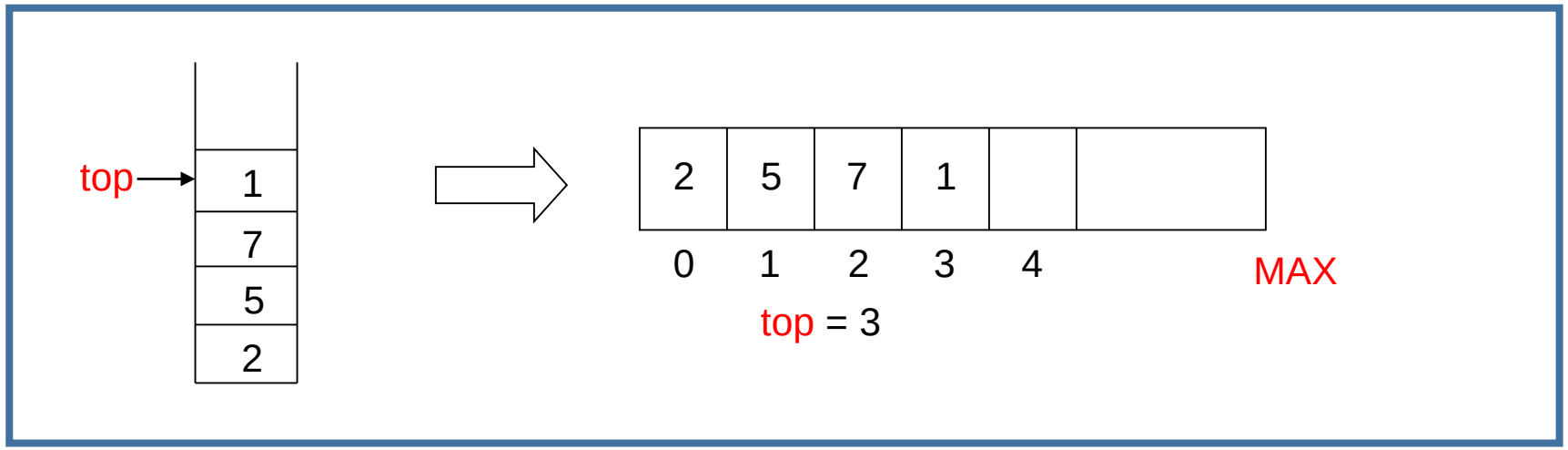
7.1.6 Stack Operations (Example)



7.1.7 Static and Dynamic Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list
- Both implementations must implement all the stack operations

7.2.1 Stack Implementation: Array



- In case of an array, it is possible that the array may “fill-up” if we push enough elements.
- Have a boolean function **`isFull()`** which returns true is stack (array) is full, false otherwise.
- Make sure to call this function before pushing a value in stack

7.2.2 Stack Operations-Push (Algorithm)

```
void push(ITEM)
{
        if (isFull())
                Print “Stack Overflow”
                Exit
        else
                TOP = TOP + 1
        End if

        A[TOP] = ITEM
        Exit
}
```

7.2.3 Stack Operations: Pop (Algorithm)

```
int pop()
{
        if (isEmpty())
                Print “Stack Underflow”
                Exit
        else
                ITEM = A[TOP]
        End if

        TOP = TOP - 1

        return ITEM
}
```

7.2.4 Stack Operations : Miscellaneous

```
int peek()
{
        return A[top];
}
```

```
int isEmpty()
{
        if (top < 0)
                return TRUE;
        else
                return FALSE;
}
```

```
int isFull()
{
        if (top == MAX)
                return TRUE;
        else
                return FALSE;
}
```

7.3.1 Stack Using Linked List

- We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements.

- As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

7.3.2 Stack Using Linked List

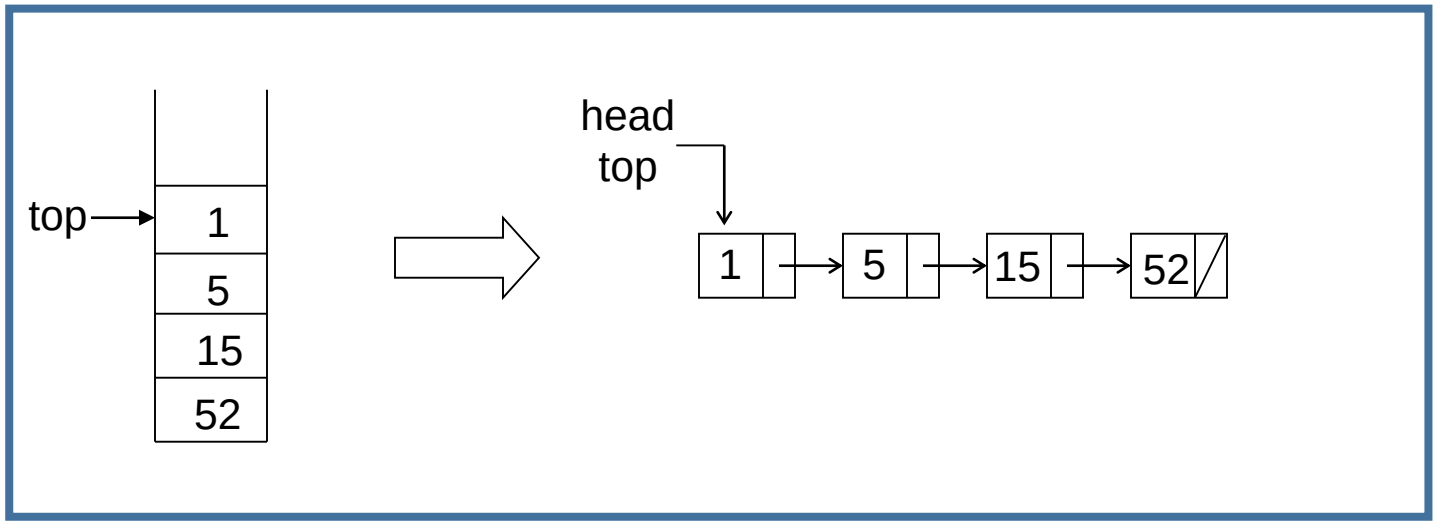
- For a singly-linked list, insert at start or end takes constant time using the head and current pointers respectively.

- Removing an element at the start is constant time but removal at the end required traversing the list to the node one before the last.

- Make sense to place stack elements at the start of the list because insert and removal are constant time at the start.

7.3.3 Stack Using Linked List

- No need for the current pointer; head is enough.



7.4.1 Stack Operation: Singly Linked List

push(item)

```
Step 1: create a newNode with provided value (item).
Step 2: check whether stack is empty (top==NULL)
Step 3: If it is empty, then set top=newNode and newNode →next =NULL
Step 4: If it is not empty, then set newNode →next=top
Step 5: Finally, set top=newNode
```

7.4.2 Stack Operation: Singly Linked List

pop()

```
Step 1: check whether stack is empty (top==NULL)
Step 2: If it is empty, then print “stack is EMPTY” and
        terminate the function
Step 3: If it is not empty, then define a nodePtr temp = top
Step 4: set top=top->next
Step 5: delete temp i.e. free(temp)
```

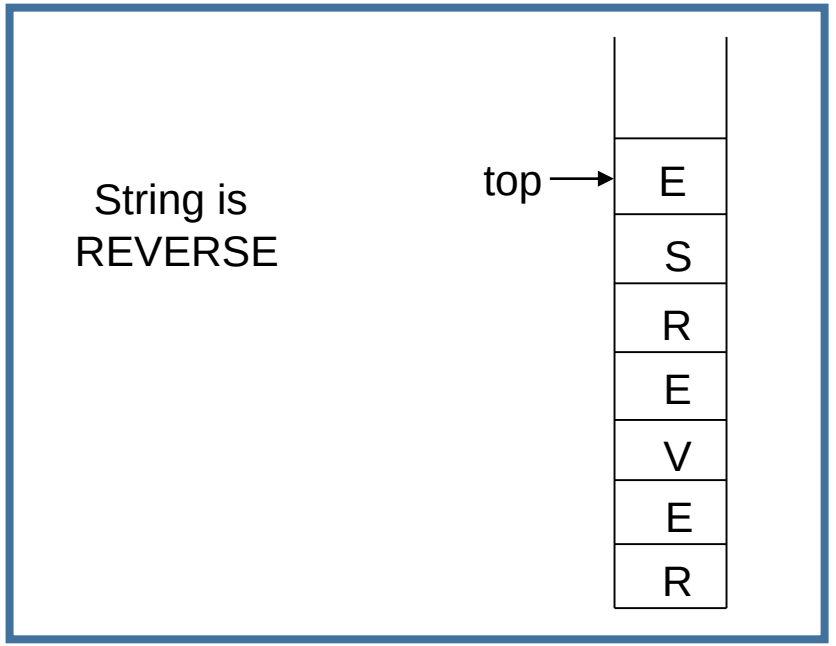
7.4.3 Stack Operation: Singly Linked List

display()

```
Step 1: check whether stack is empty (top==NULL)
Step 2: If it is empty, print “stack is EMPTY” and terminate
        function
Step 3: If it is not empty, then define a node pointer temp and
        set it to top
Step 4: Display “temp→data” and move temp to next node. Repeat
        until temp →next!=NULL.
```

7.5.1 Stack Application

- Reversing string
  - Read string from left to right
  - Push each character onto the stack one by one
  - After all the characters in string are pushed, pop each character one by one.

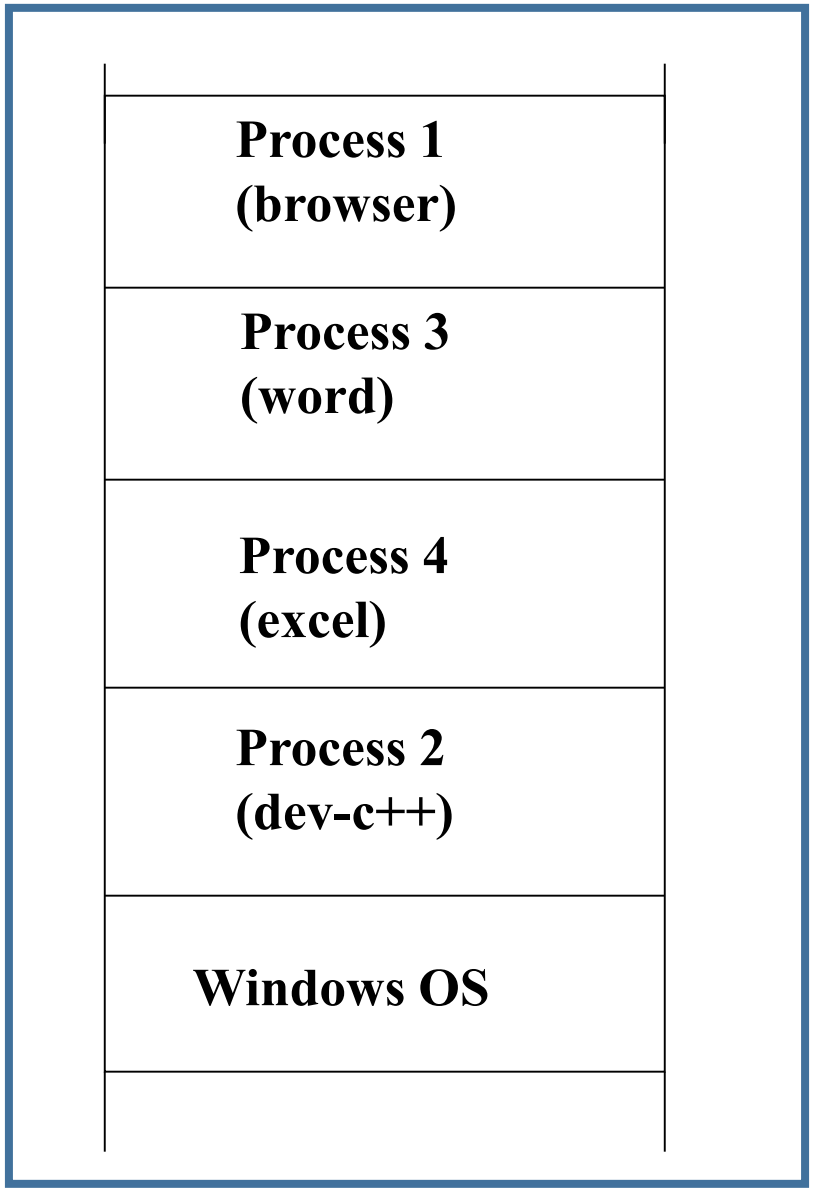


7.5.2 Stack Applications

- Memory Management

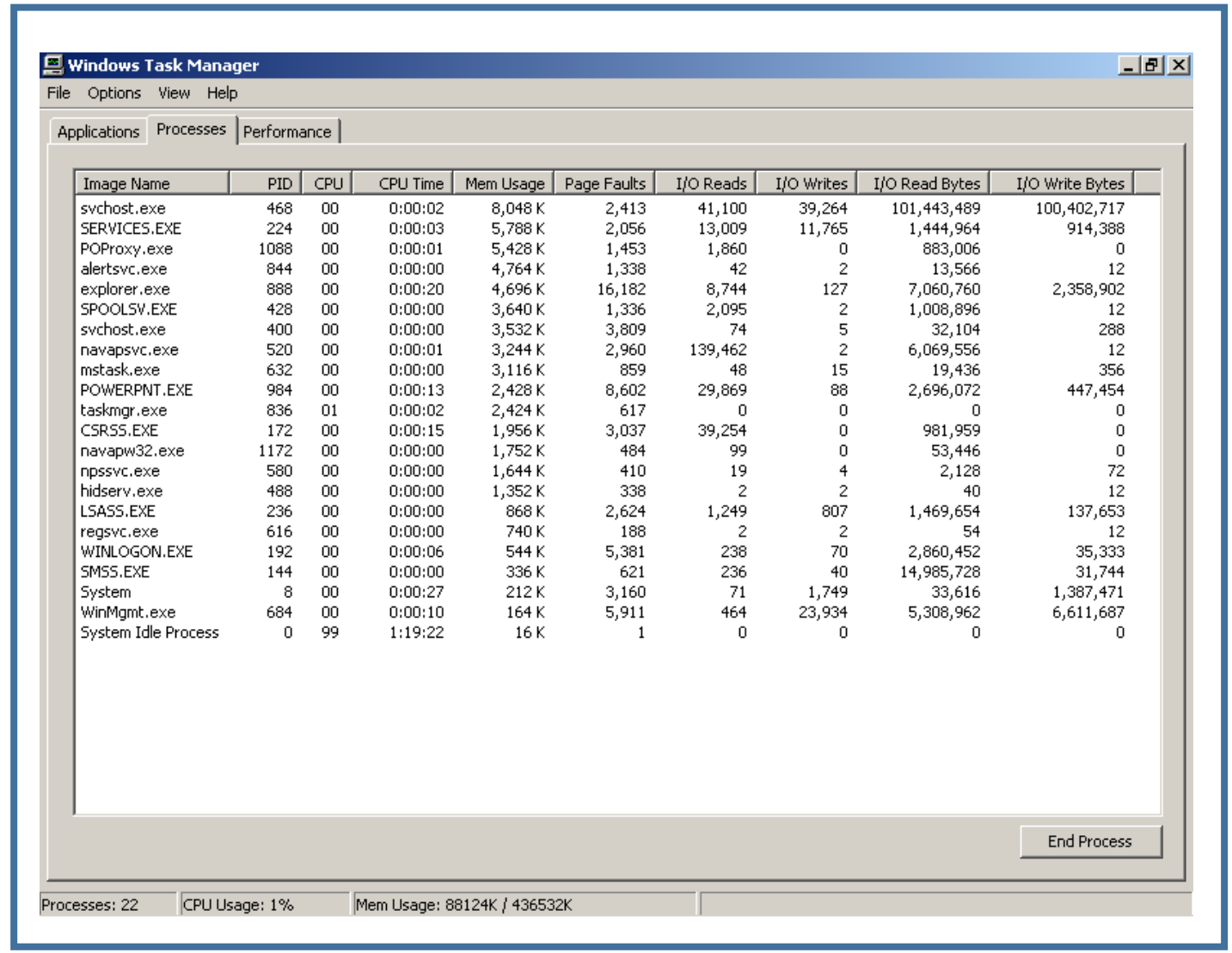
  - When a program (.exe) is run, it is loaded in memory. It becomes a process.

- The process is given a block of memory.

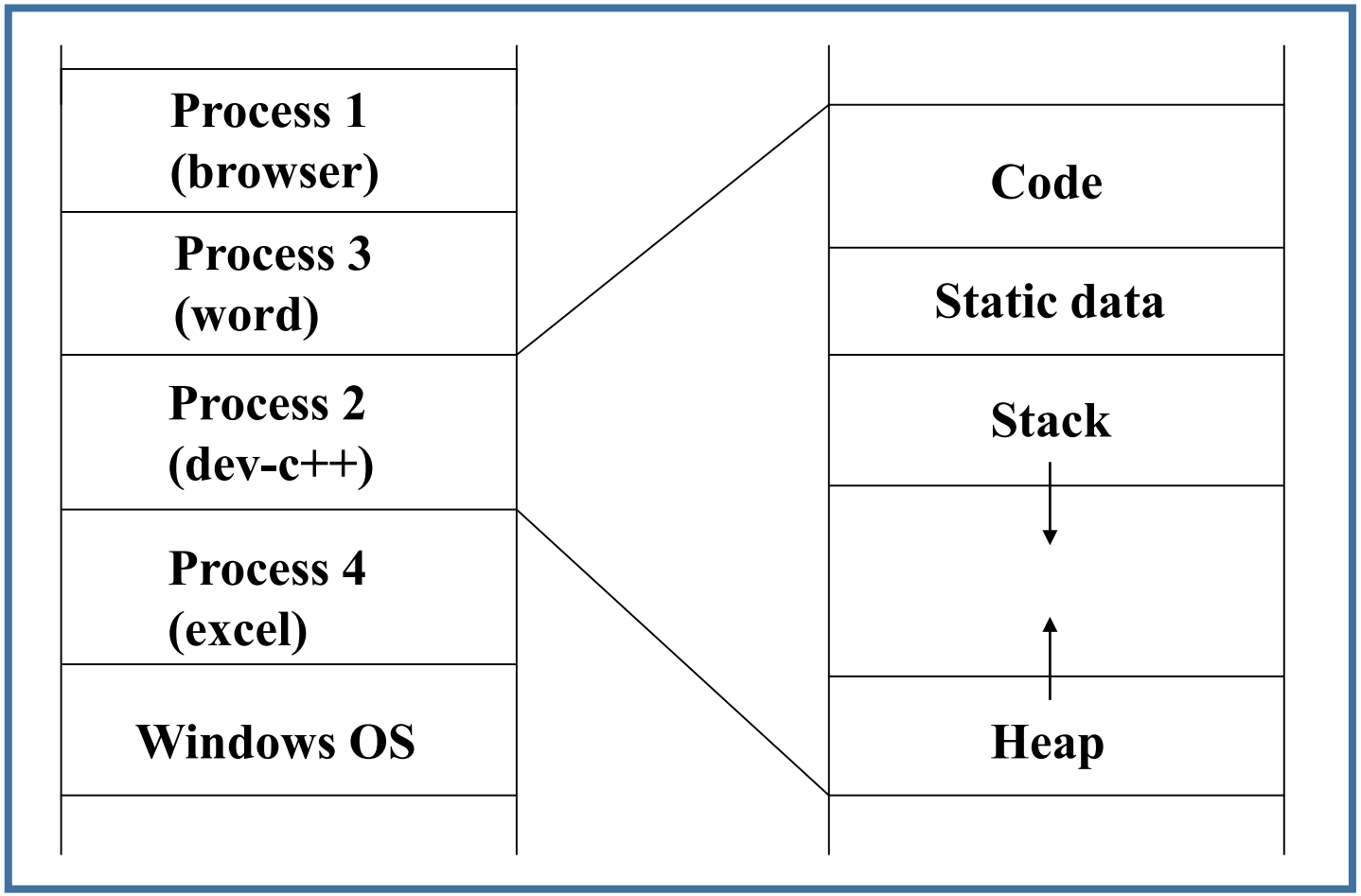


7.5.3 Stack Applications

[Control+Alt+DEL]



7.5.4 Stack Applications



7.5.5 Stack Application

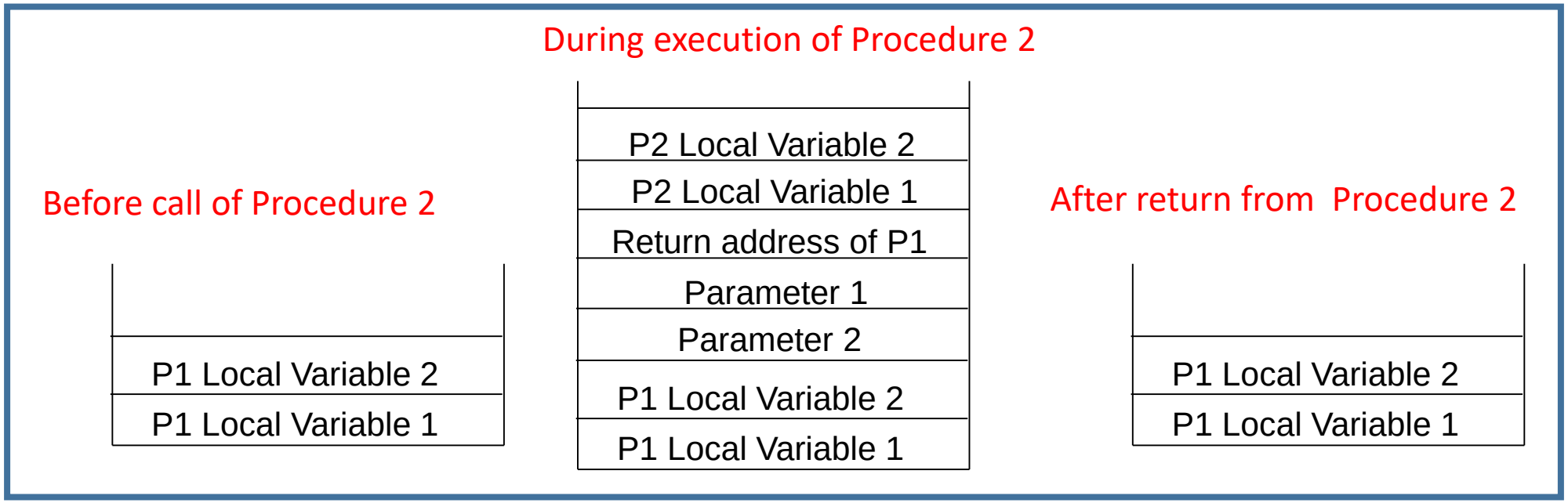
- Call stack
  - whenever a function is called, the computer has to remember where to continue after the function returns
  - It is done by pushing where it had got to onto the call stack

7.5.6 Stack Applications

- Call stack
  - Example Procedure 1 calls Procedure 2

```
Procedure 1()
{
   Local Variable 1, 2
   Procedure(1, 2);
   …….
   ……
}
```

```
Procedure 2(Parameter 1, Parameter 2)
{
   Local Variable 1, 2
   …….
   ……
   return
}
```



7.5.7 Stack Applications

- Validity of an expression containing nested parenthesis
  - Important task for a compiler
  - Used to check whether the pairs and orders in a nested parenthesis are correct
  - For each left parenthesis braces or brackets, there should be a corresponding closing symbol and symbols that are appropriately nested.

7.5.8 Stack Applications

```c
#include<stdio.h>
int main()
{
        int i;
        for (i = 1; i < 5; i++)
        {
                printf("%d\n", i);
        }
        return 0;

}
```

Valid Inputs

{}
({[]})
{[] ()}
[{ ({} [])}]

Invalid Inputs

{(}
([(()])
{}[] ()}
[{) }(] }]

7.5.9 Stack Applications

Algorithm

1. Start traversing the expression
2. If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack
3. If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack
4. If the popped character is the matching starting bracket then it is correct else parenthesis are not balanced.
5. After complete traversal,

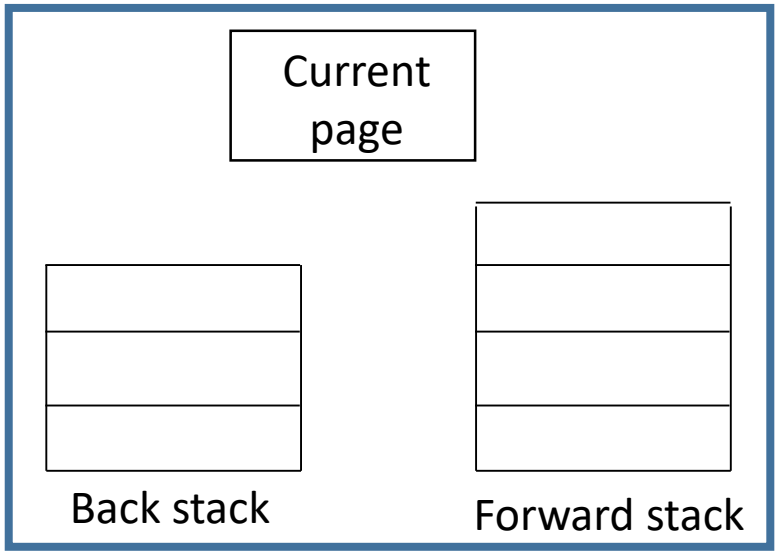
   - if there is some starting bracket left in stack
   - Or stack is empty and still some bracket left in the expression

   then nested is not appropriately balanced

7.5.10 Stack Applications

- Back and Forward in Web Browser

- To allow the user to move both forward and backward two stacks are employed.
- When user presses the back button
  - -Item from the backward stack is popped, and becomes the current web page
  - -The previous web page is pushed on the forward stack
- When the user pushes the forward button
  - -item from the forward stack is popped, and becomes the current web page.
  - -The previous web page is pushed on the back stack.



7.5.11 Stack Applications

- Evaluation of infix expression is not always easy
- So an alternative notion, termed postfix expression is employed
- Need for parenthesis in the postfix form is avoided, as are any rules for precedence and associativity
- Computation of arithmetic expressions can be efficiently carried out in postfix notation with the help of a stack.
- Infix can be converted to postfix using stack (will be covered in next class)

Infix	2 + 3	2 + 3 * 4	(2 + 3) * 4	2 + 3 + 4	2 - (3 – 4)
Postfix	2 3 +	2 3 4 * +	2 3 + 4 *	2 3 + 4 +	2 3 4 - -

7.6.1 Implementing Stacks: Array

- Advantages
  - Constant time to push or pop an element
- Disadvantage
  - fixed size- array requires allocation ahead of time
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

7.6.2 Implementing a Stack: Linked List

- Advantages:
  - Always constant time to push or pop an element
  - List uses only as much memory as required by the nodes

- Disadvantages
  - Allocating and deallocating memory for list nodes does take more time than preallocated array.
  - List pointers (head, next) require extra memory

- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

Q & A ?