Data Structure & Algorithm
CSE-225

Lecture-11: Stack

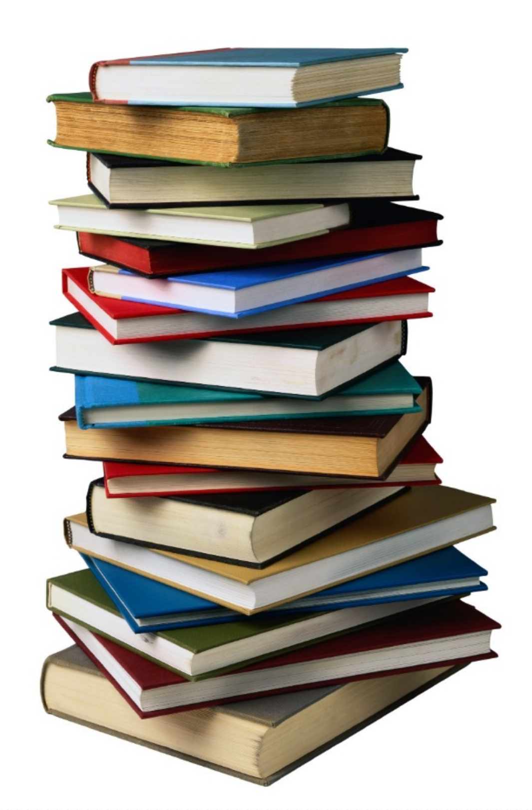
- There are certain frequent situations in computer science when one wants to restrict insertion and deletion so that they can take place only at the beginning or at the end not in the middle.

  - **Stack**
  - **Queue**

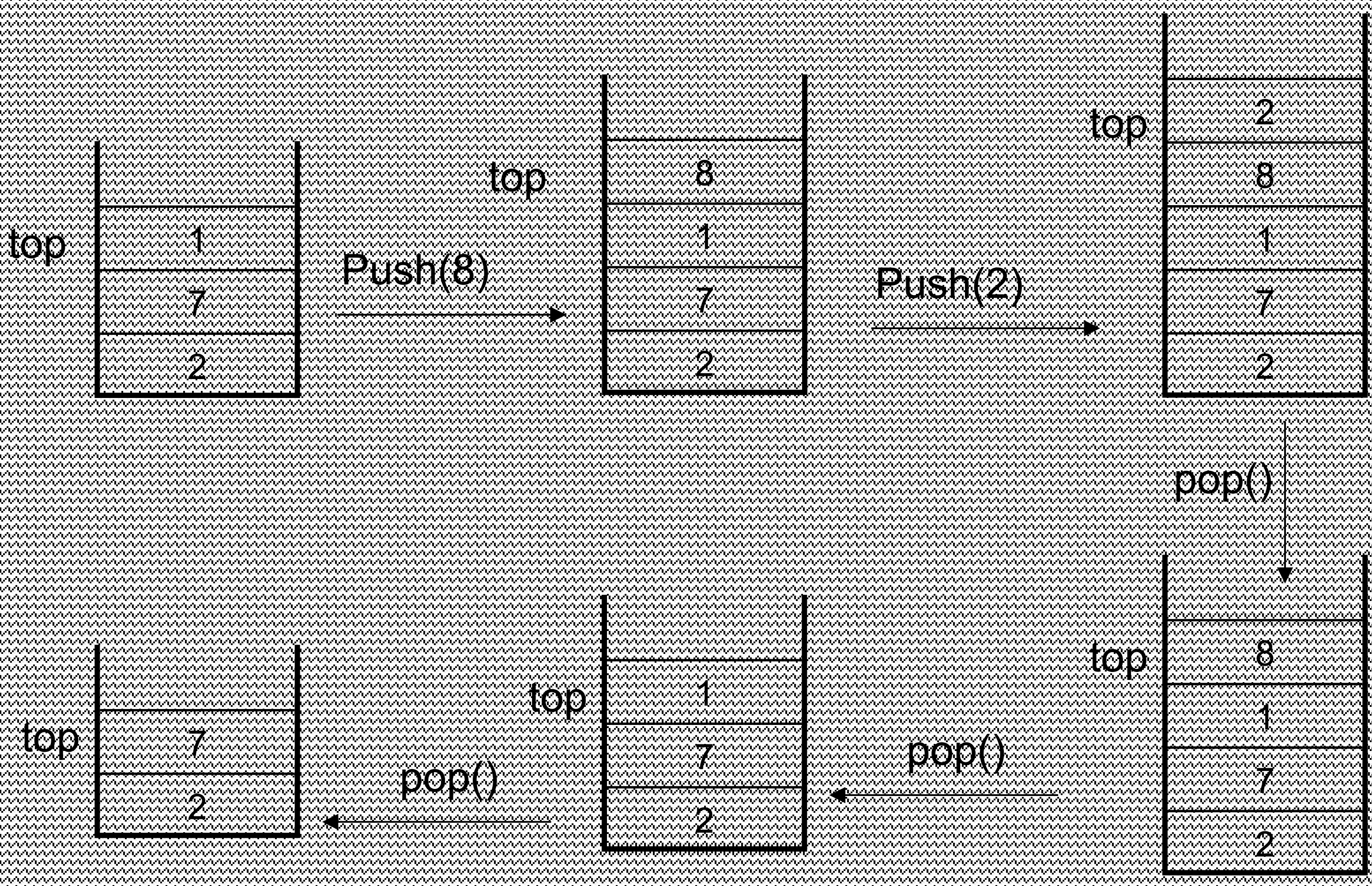
Stack

Stack

- A list
- Data items can be added and deleted
- Maintains **Last In First Out (LIFO)** order



An Example of a Stack



Stack

- A Stack is a list of elements in which an element may be inserted or deleted only at one end, call top of the Stack

- Two basic operations are associated with Stack
  - **Push :** Insert an element into a stack
  - **Pop :** Delete an element from a stack

Stack

- Stores a set of elements in a particular order

- Stack principle:

  LAST IN FIRST OUT= LIFO

- It means: the last element inserted is the first one to be removed

- Which is the first element to pick up?

Representation of Stack

Stack can be represented in two different ways :

[1] Linear ARRAY

[2] One-way Linked list

Array Representation of Stack

STACK

A	**B**	**C**					
0	1	2	3	4	5	6	7

TOP 2 MAXSTK 7

TOP = -1 or TOP = NULL
will indicates that the stack is empty

Specification of StackType

Structure:	**Elements are added to and removed from the top of the stack.**
Definitions (provided by user):	
MAX_ITEMS	Maximum number of items that might be on the stack.
ItemType	Data type of the items on the stack.
Operations (provided by the ADT):	
MakeEmpty	
Function	Sets stack to an empty state.
Postcondition	Stack is empty.
Boolean IsEmpty	
Function	Determines whether the stack is empty.
Precondition	Stack has been initialized.
Postcondition	Returns true if stack is empty and false otherwise.
Boolean IsFull	
Function	Determines whether the stack is full.
Precondition	Stack has been initialized.
Postcondition	Returns true if stack is full and false otherwise.

Specification of StackType

Push(ItemType newItem)	
Function	Adds newItem to the top of the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is full), exception FullStack is thrown, else newItem is at the top of the stack.
Pop()	
Function	Removes top item from the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is empty), exception EmptyStack is thrown, else top element has been removed from stack.
ItemType Top()	
Function	Returns a copy of the top item on the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is empty), exception EmptyStack is thrown, else a copy of the top element is returned.

stacktype.h

```
#ifndef STACKTYPE_H_INCLUDED
#define STACKTYPE_H_INCLUDED

const int MAX_ITEMS = 5;

class FullStack
{};  // Exception class thrown by Push when stack is full.
class EmptyStack
{};  // Exception class thrown by Pop and Top when stack is empty.

template <class ItemType>
class StackType
{
    public:
        StackType();
        bool IsFull();
        bool IsEmpty();
        void MakeEmpty();
        void Push(ItemType);
        void Pop();
        ItemType Top();
    private:
        int top;
        ItemType  items[MAX_ITEMS];
};
#endif // STACKTYPE_H_INCLUDED
```

stacktype.cpp

```
#include "StackType.h"
template <class ItemType>

StackType<ItemType>::StackType()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
    return (top == -1);
}
void StackType<ItemType>::MakeEmpty()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsFull()
{
    return (top ==  MAX_ITEMS-1);
}
```

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType
newItem)
{
    if( IsFull() )
        throw FullStack();
    top++;
    items[top] = newItem;
}

template <class ItemType>
void StackType<ItemType>::Pop()
{
    if( IsEmpty() )
        throw EmptyStack();
    top--;
}
template <class ItemType>
ItemType StackType<ItemType>::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    return items[top];
}
```

stacktype.cpp

```
#include "StackType.h"
template <class ItemType>

StackType<ItemType>::StackType()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
    return (top == -1);
}
void StackType<ItemType>::MakeEmpty()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsFull()
{
    return (top ==  MAX_ITEMS-1);
}
```

O(1) O(1) O(1) O(1)

```
template <class ItemType>
void StackType<ItemType>::Push(ItemType
newItem)
{
    if( IsFull() )
        throw FullStack();
    top++;
    items[top] = newItem;
}
template <class ItemType>
void StackType<ItemType>::Pop()
{
    if( IsEmpty() )
        throw EmptyStack();
    top--;
}
template <class ItemType>
ItemType StackType<ItemType>::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    return items[top];
}
```

O(1) O(1) O(1)

PUSH Operation

- Perform the following steps to PUSH an ITEM onto a Stack

  [1] If TOP = MAXSTK, Then print: Overflow,

  Exit [Stack already filled]

[2] Set TOP++

[3] Set STACK[TOP] = ITEM

[Insert Item into new TOP Position]

[4] Exit

POP Operation

Delete top element of STACK and assign it to the variable ITEM

[1] If TOP = -1, Then print Underflow and Exit

[2] Set ITEM= STACK[TOP]

[3] TOP--

[4] Exit

Push Operation

```
void push(int num){
 if(isFull()){
     cout<<"STACK is FULL."<<endl;
     return;
   }
   ++TOP;
   STACK[TOP]=num;
   cout<<num<<" has been inserted."<<endl;
}
```

POP Operation

- POP operation is accomplished by deleting the node pointed to by the TOP pointer [**Delete the first node in the list**]

Pop operation

```
//pop - to remove item
void pop(){
    int temp;
    if(isEmpty()){
        cout<<"STACK is EMPTY."<<endl;
        return;
    }

    temp=STACK[TOP];
    TOP--;
    cout<<temp<<" has been deleted."<<endl;
}
```

Data Structures

ARITHMETIC EXPRESSIONS; POLISH NOTATION

- **Infix notation:** The operator symbol is placed between its two operands. Example: A+B, A-B, A*B, A/B

- **Prefix notation:** The operator symbol is placed before its two operands. Example: +AB, -AB, *AB, /AB
  - This notation is also known as **Polish notation**, named after the Polish mathematician Jan Lukasiewicz.
  - The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

ARITHMETIC EXPRESSIONS; POLISH NOTATION

- **Postfix notation:** The operator symbol is placed after its two operands. Example: AB+, AB-, AB*, AB/
  - This notation is also known as **Reverse Polish notation**.
  - One never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation.
- The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to **postfix notation**, and **then it evaluates the postfix expression.** In each step, the stack is the main tool that is used to accomplish the given task.

Data Structures (Operator Precedence)

OPERATOR	PRECEDENCE	VALUE
Exponentiation ($ or ↑ or ^)	Highest	3
*, /	Next highest	2
+, -	Lowest	1

- **Two operators of same priority can't stay together.**
- **Higher priority operator will not stay in the stack when lower priority operator will be inserted.**
- **(…..) => pop all the operators from stack and place them in the postfix.**

- **Example 6.7: Transform the following arithmetic infix expression Q into its equivalent postfix expression P:**

  Q: A + (B*C - (D /E ↑ F)*G)*H

  Solution:

  - Push "(" onto stack and then add ")" to the end of Q. Thus, Q becomes

    Q: A + (B * C - (D / E ↑ F) * G) * H)

  - Start scanning. The following table shows the status of STACK and of the string P as each element of Q is scanned.

- **Example 6.7: Transform the following arithmetic infix expression Q into its equivalent postfix expression P:**

Q: A + (B*C - (D /E ↑ F)*G)*H)

Solution:

Symbol Scanned	STACK	Expression, P
A	(	A

Data Structures

- **Example 6.7: Transform . . . . . postfix expression P:**

  Q: A + (B*C - (D /E ↑ F)*G)*H)

Symbol Scanned	STACK	Expression, P

- **Solved Problem 6.10: Transform the following arithmetic infix expression Q into its equivalent postfix expression P:**

  Q: ((A + B)*D) ↑ (E-F)

  Solution:
  - Push "(" onto stack and then add ")" to the end of Q. Thus, Q becomes

    Q: ((A + B) * D) ↑ (E-F))
  - Start scanning. The following table shows the status of STACK and of the string P as each element of Q is scanned.

Data Structures

- **Solved Problem 6.10: Transform . . . postfix expression P:**

Q: ((A + B)*D) ↑ (E-F))

Symbol Scanned	STACK	Expression, P
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	A B
)	((	A B +
*	((*	A B +
D	((*	A B + D
)	(	A B + D *
↑	(↑	A B + D *
(	(↑(	A B + D *
E	(↑(	A B + D * E
-	(↑(-	A B + D * E
F	(↑(-	A B + D * E F
)	(↑	A B + D * E F -
)		A B + D * E F - ↑

- **Algorithm 6.6: Write an algorithm that transforms the infix expression into its equivalent postfix expression.**

POLISH (Q, P)
Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK and add ")" to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.

Data Structures

- **Algorithm 6.6: Write an algorithm that transforms the infix expression into its equivalent postfix expression.**

5. If an operator Ä is encountered, then

    (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than Ä.

    (b) Add Ä to STACK.

    [End of if structure]

6. If a right parenthesis is encountered, then

    (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.

    (b) Remove the left parenthesis.

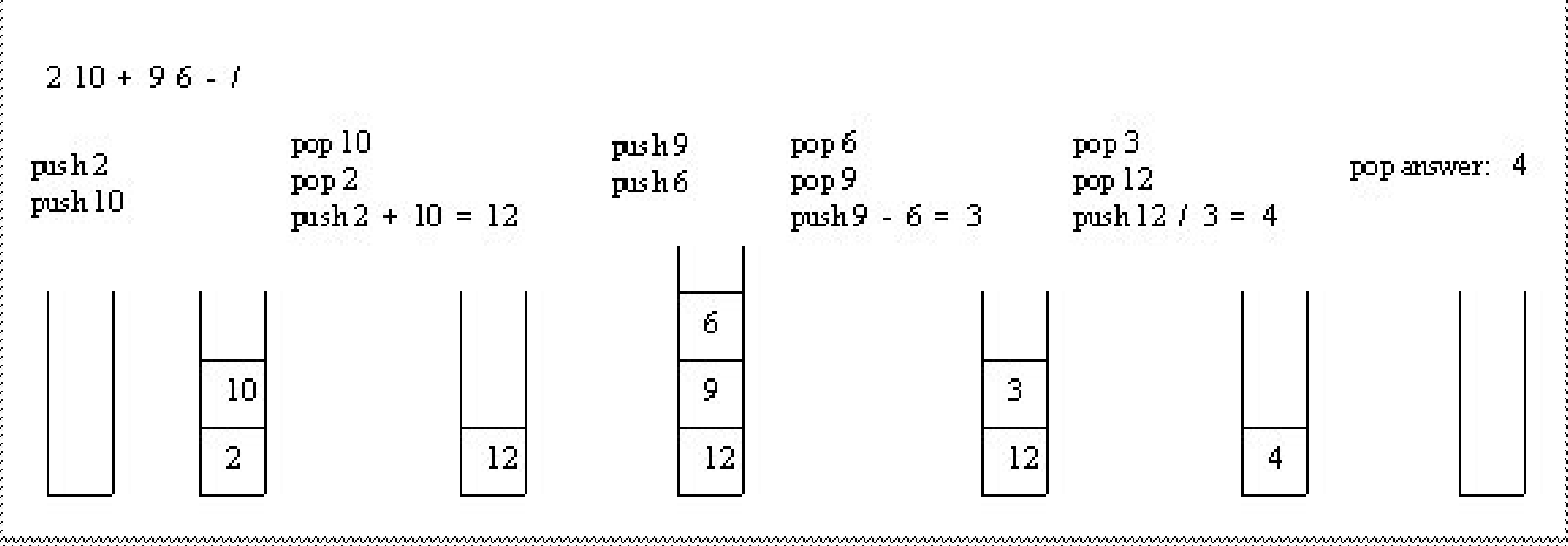
    [End of if structure]

  [End of Step 2 loop]

7. Exit.

Application of Stacks - Evaluating Postfix Expression

- Example: Consider the postfix expression, **2 10 + 9 6 - /**, which is **(2 + 10) / (9 - 6)** in infix, the result of which is 12 / 3 = 4.
- The following is a trace of the postfix evaluation algorithm for the postfix expression:



Data Structures

- **Example 6.6:** Find the value of the following arithmetic expression P written in postfix notation:

  P: 5, 6, 2, +, *, 12, 4, /, -

  Solution:
  - First we add a sentinel right parenthesis at the end of P:

P:	5,	6,	2,	+,	*,	12,	4,	/,	-,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

  - Start scanning from left to right. The following table shows the contents of STACK as each element of P is scanned. The final number in STACK, 37, which is assigned to VALUE when the sentinel ")" is scanned, is the value of P.

- **Example 6.6:** Find the value of the following arithmetic expression P written in postfix notation:

P:	5,	6,	2,	+,	*,	12,	4,	/,	-,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

Solution:

Symbol Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	

Data Structures

- **Algorithm 6.5: Write an algorithm that finds the value of an arithmetic expression written in postfix notation.**

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P.
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator is encountered, then:
   a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element. Ä
   b) Evaluate B A.
   c) Place the result of (b) back on STACK.
   [End of If structure] Ä
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

Data Structures

Recursion

- A procedure is called a **recursive** procedure if it contains a Call statement to itself.
- A recursive procedure must have the following two properties:
  - There must be certain criteria, called **base criteria**, for which the procedure does not call itself.
  - Each time the procedure does call itself, it must be closer to the base criteria.

- **Recursion**

  Example 6.9: Calculate 4! using the recursive definition.

  Solution: This calculation requires the following nine steps:

1) 4! = 4. 3!
2) 3! = 3. 2!
3) 2! = 2 . 1!
4) 1! = 1 . 0!
5) 0! = 1
6) 1! = 1 . 1 = 1
7) 2! = 2 . 1 =2
8) 3! = 3. 2 = 6
9) 4!=4 . 6 = 24

Data Structures

- **Procedure 6.9A:** Write a procedure that calculates N!

FACTORIAL(FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Set FACT := 1.
3. Repeat for K = 1 to N.

   Set FACT := K*FACT.

   [End of loop.]
4. Return.

- **Procedure 6.9B:** Write a recursive procedure that calculates N!

FACTORIAL(FACT, N)

This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Call FACTORIAL(FACT, N — 1).
3. Set FACT := N*FACT.
4. Return.

Data Structures

- **Fibonacci Sequence**

The Fibonacci sequence (usually denoted by F_0, F_1, F_2, ...) is as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

That is, $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci Sequence)

a) If $n = 0$ or $n = 1$, then $F_n = n$.

b) If $n > 1$, then $F_n = F_{n-2} + F_{n-1}$

- **Fibonacci Sequence**

Procedure 6.10: Write a recursive procedure that finds the nth Fibonacci number.

FIBONACCI(F1B, N)

This procedure calculates F_N and returns the value in the first parameter FIB.

1. If N = 0 or N = 1, then: Set FIB := N, and Return.
2. Call FIBONACCI(F1BA, N - 2).
3. Call FIBONACCI(FIBB, N - 1).
4. Set FIB := F1BA + FIBB.
5. Return.

Data Structures

- **Recursion vs. Iteration**
  - Roughly speaking, recursion and iteration perform the same kinds of tasks:
    Solve a complicated task one piece at a time, and combine the results.
  - Emphasis of iteration:
    keep repeating until a task is "done"
  - Emphasis of recursion:
    Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results. Example: recursive factorial function.

Data Structures

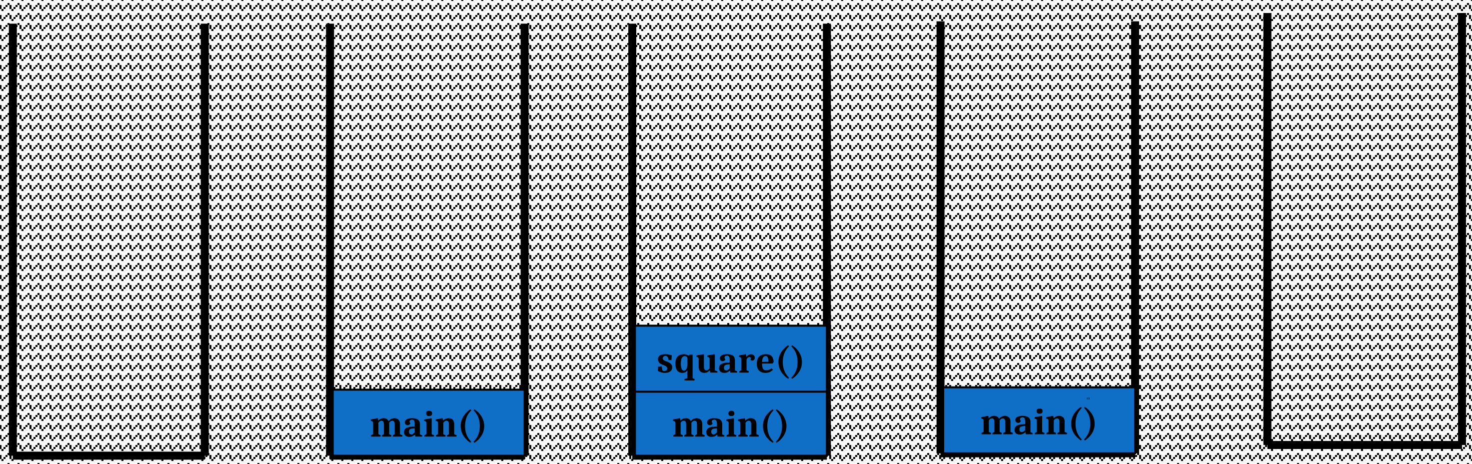
- **Recursion vs. Iteration**
  - The function calls itself recursively on a smaller version of the input (n - 1). The solution to the problem is then devised by combining the solutions obtained from the simpler versions of the problem.

  - Use of recursion in an algorithm has both advantages and disadvantages.
    - The main advantage is usually simplicity.
    - The main disadvantage is often that the algorithm may require large amounts of memory if the depth of the recursion is very large.
    - Recursive isn't always better. This takes $O(2^n)$ steps. Unusable for large n.
    - Iterative approach is "linear"; it takes O(n) steps.

Stacks and Methods

- When you run a program, the computer creates a stack for you.
- Each time you invoke a method, the method is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple C++ program.

Stacks and Methods



Time: 0
Empty Stack

Time 1:
Push: main()

Time 2:
Push: square()

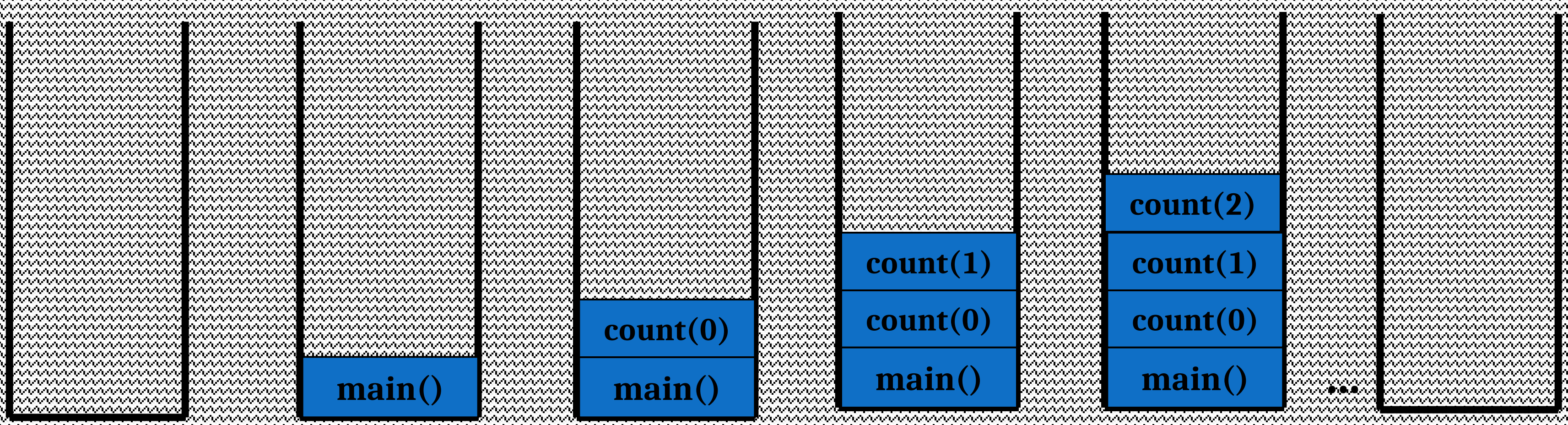
Time 3:
Pop: square()
returns a value.
method exits.

Time 4:
Pop: main()
returns a value.
method exits.

Stacks and Recursion

- Each time a method is called, you *push* the method on the stack.
- Each time the method returns or exits, you *pop* the method off the stack.
- If a method calls itself recursively, you just push another copy of the method onto the stack.
- We therefore have a simple way to visualize how recursion really works.

Stacks and Recursion in Action



Time: 0 Empty Stack	Time 1: Push: main()	Time 2: Push: count(0)	Time 3: Push: count(1)	Time 4: Push: count(2)	Times 5-8: Pop everything

Inside count(0):
print (index); ← 0
if (index < 2)
    count(index+1);

Inside count(1):
print (index); ← 1
if (index < 2)
    count(index+1);

Inside count(2):
print (index); ← 2
if (index < 2)
    count(index+1);

This condition now fails! Hence, recursion stops, and we proceed to pop all functions off the stack.

Stack Short-Hand

- Rather than draw each stack like we did last time, you can try using a short-hand notation.

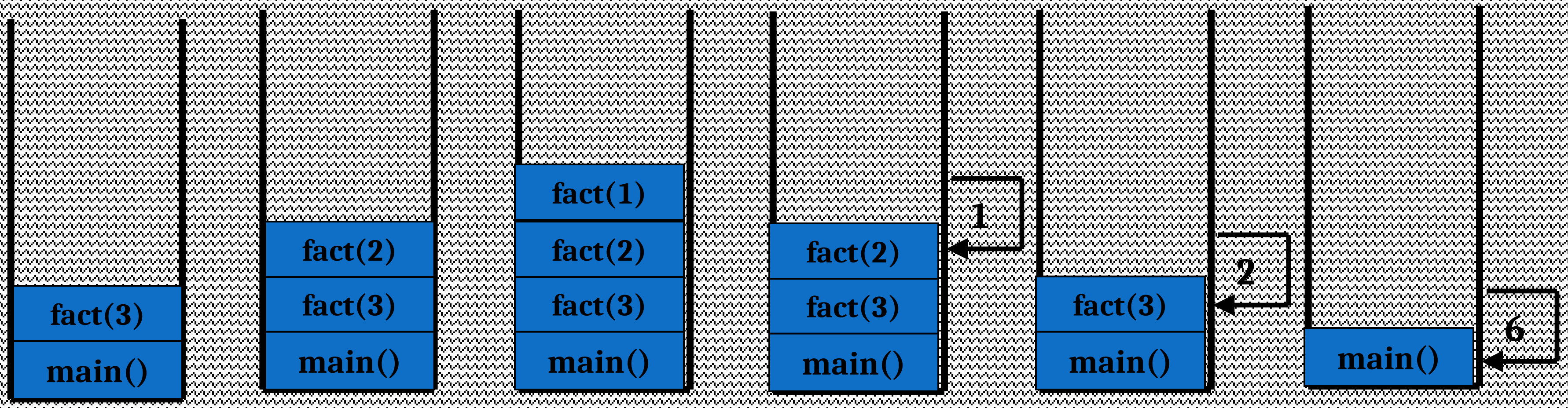
time stack	output
time 0:	empty stack
time 1:	f(4)
time 2:	f(4), f(3)
time 3:	f(4), f(3), f(2)
time 4:	f(4), f(3), f(2), f(1)
time 5:	f(4), f(3), f(2)
time 6:	f(4), f(3)
time 7:	f(4)
time 8:	empty

Level: 1
Level: 2
Level: 3
Level: 4
LEVEL: 4
LEVEL: 3
LEVEL: 2
LEVEL: 1

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:
  n! = n * (n-1) * (n-2) …. * 1;
- For example:
  1! = 1 (Base Case)
  2! = 2 * 1 = 2
  3! = 3 * 2 * 1 = 6
  4! = 4 * 3 * 2 * 1 = 24
  5! = 5 * 4 * 3 * 2 * 1 = 120

Finding the factorial of 3



Time 2:
Push: fact(3)

Time 3:
Push: fact(2)

Time 4:
Push: fact(1)

Time 5:
Pop: fact(1)
returns 1.

Time 6:
Pop: fact(2)
returns 2.

Time 7:
Pop: fact(3)
returns 6.

Inside findFactorial(3):
if (number <= 1) return 1;
else return (3 * factorial (2));

Inside findFactorial(2):
if (number <= 1) return 1;
else return (2 * factorial (1));

Inside findFactorial(1):
if (number <= 1) return 1;
else return (1 * factorial (0));

Example Using Recursion: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)

    - fibonacci(0) and fibonacci(1) are base cases
  - Golden ratio (golden mean)

Recursion vs. Iteration

- Iteration
  - Uses repetition structures (for, while or do…while)
  - Repetition through explicitly use of repetition structure
  - Terminates when loop-continuation condition fails
  - Controls repetition by using a counter
- Recursion
  - Uses selection structures (if, if…else or switch)
  - Repetition through repeated method calls
  - Terminates when base case is satisfied
  - Controls repetition by dividing problem into simpler one

Recursion vs. Iteration (cont.)

- Recursion
  - More overhead than iteration
  - More memory intensive than iteration
  - Can also be solved iteratively
  - Often can be implemented with only a few lines of code