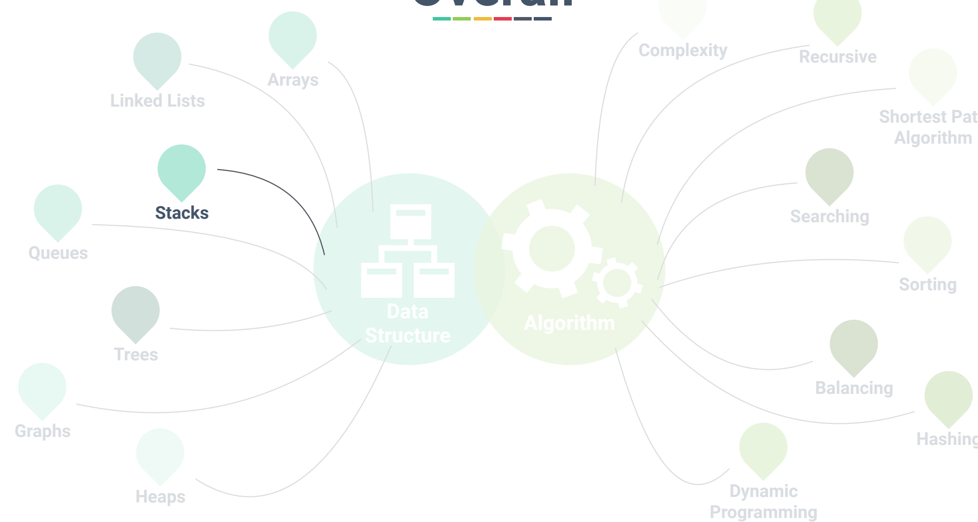
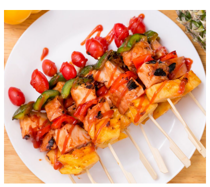
Chapter 3: Stacks

Dr. Sirasit Lochanachit

Overall



Stacks

A stack is a collection of objects that are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle [1].

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Stacks

Object can be pushed into a stack at any time, but only the most recently inserted object can be removed or accessed (Pop).

The name “stack” is derived from the metaphor of a stack of plates in a plate dispenser.

[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

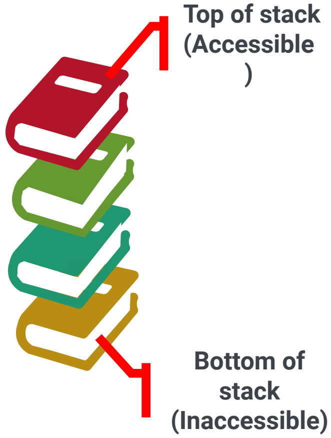
Stacks

Formally, there are 2 main operations of stacks:

1) Push = Add an item to the top of stack
2) Pop = Remove and return the top item of the stack
   - Only the item at the top of the stack can be removed and read.
   - Unable to remove the bottom items without taking out all previous items

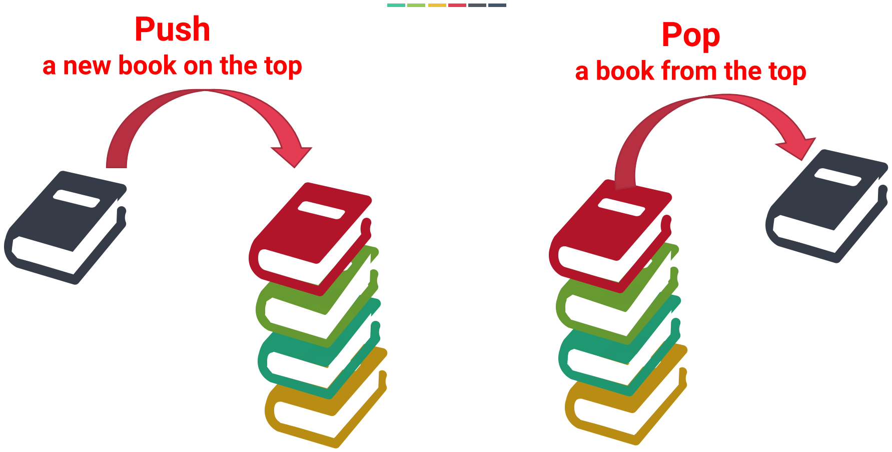
Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

Stacks



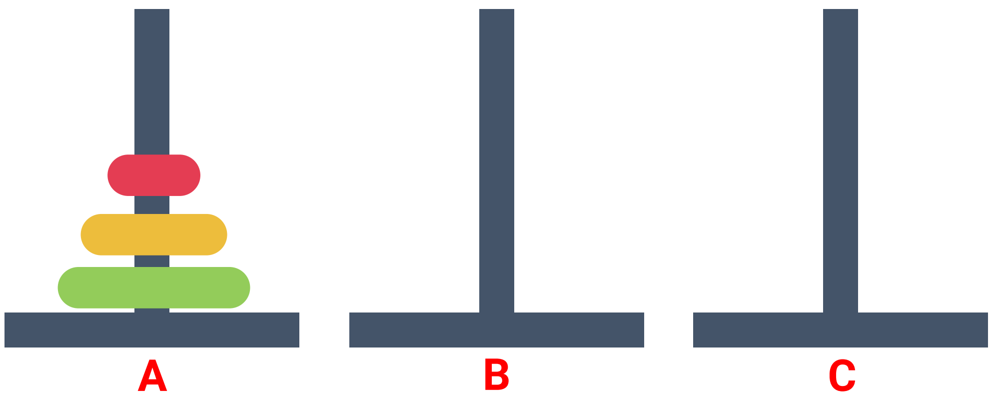
Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

Stacks

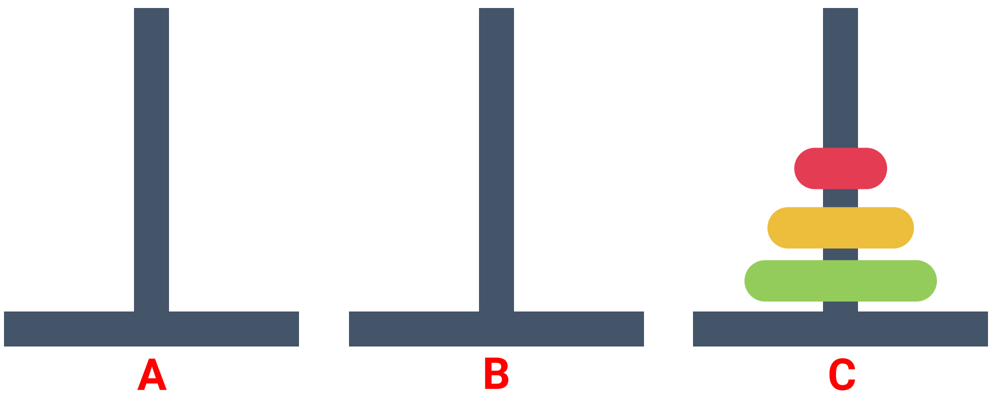


Faculty of Information Technology King Mongkut's Institute of Technology Ladkrabang

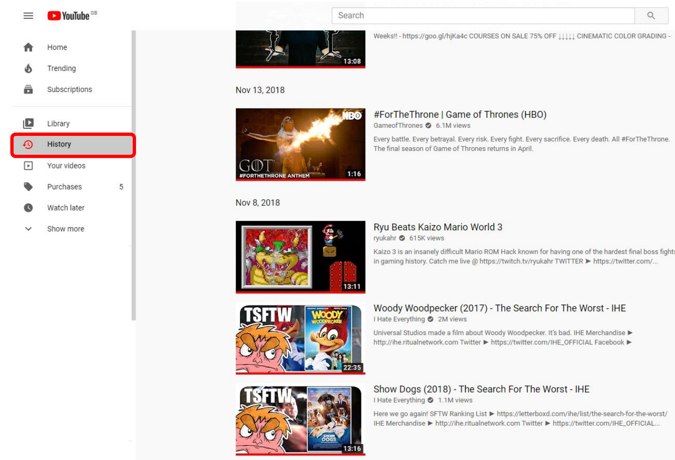
Towers of Hanoi



Towers of Hanoi



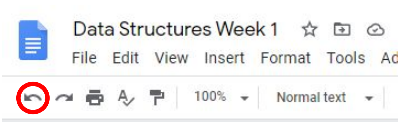
Stack Applications



Recently watched vdo clips

Web Browser's history of recently visited sites

Stack Applications



Probably the most useful button in the world - **Undo button!!**

Stacks

Additional operations of stacks:

1) Top = Return the top item of the stack without removing it
2) is_empty = Check whether a stack is empty
3) len(stack) = Return the number of elements in a stack

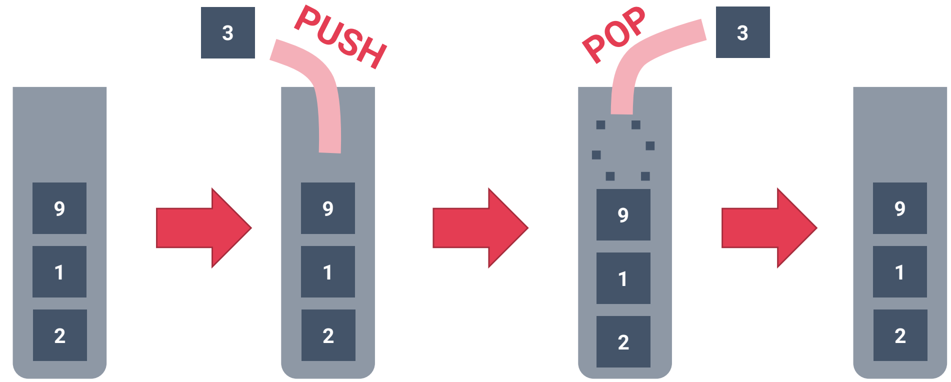
[1] Michael T. Goodrich et al., Data Structures and Algorithms in Python, 2013

Stacks

How to Implement a Stack?

Array!!

Stacks



Operation Example

Operation	Return Value	Stack
S.push(9)	-	9
S.push(5)	-	9 5
S.top()	5	9 5
S.push(2)	-	9 5 2
S.pop()	2	9 5
S.is_empty()	False	9 5
len(S)	2	9 5

Asymptotic Performance

Operation	Running Time
S.push(element)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Stack Applications

Balanced Grouping Symbols

Each opening symbol must match its corresponding closing symbol.

Parentheses: (and)
Brackets: [and]
Braces: { and }

Examples

Balanced:
([ ]) [ ] ( )
((([{ [ ] }])) ( ))

Unbalanced:
(# missing)
] [# Incorrect order
({ [ ]) } # Incorrect order

Stack Applications

How to check balanced pairs?

Perform a left-to-right scan of the original sequence, using a stack S to facilitate the matching of grouping symbols.

Each time we encounter an opening symbol, we push that symbol onto stack S.

Each time we encounter a closing symbol, we pop a symbol from the stack S (assuming S is not empty), and check that these two symbols form a valid pair.

If we reach the end of the expression and the stack is empty, then the original expression was properly matched.

Otherwise, there must be an opening delimiter on the stack without a matching symbol.

Stack Applications

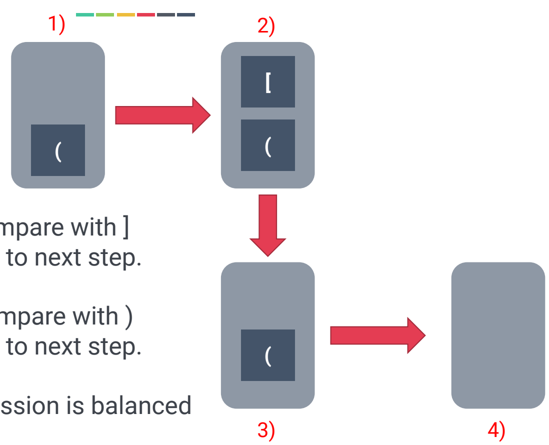
Example: (1 + x - [y + z])

1) Push (into stack S.
2) Push [into stack S.

3) Pop [from stack S and compare with]
   - Correct, then proceed to next step.

4) Pop (from stack S and compare with)
   - Correct, then proceed to next step.

5) Stack is empty = the expression is balanced



Pseudocode

IsBalanced(str):

```
Stack stack
lefty = '([{'
righty = ')]}'
for char in str:
    if char in lefty:
        stack.push(char)
    else if char in righty:
        If stack.is_empty():
            return False
        top = stack.pop()
        if (top = '(' and char != ')') or (top = '[' and char != ']') or
        (top = '{' and char != '}')
            return False
return stack.is_empty()
```

Pseudocode

IsBalanced(str):

Stack *stack*
lefty = ‘([{‘
righty = ‘)]}’
for *char* in *str*:
    if *char* in *lefty*:
        stack.push(char)
    else if *char* in *righty*:
        If *stack*.is_empty():
            return False
        top = *stack*.pop()
        if (*top* = ‘(‘ and *char* != ‘)’) or (*top* = ‘[‘ and *char* != ‘]’) or
        (*top* = ‘{‘ and *char* != ‘}’)
            return False
return *stack*.is_empty()

Incorrect order, such as] [

Incorrect case, such as [ )

Missing case, such as (

Pseudocode

IsBalanced(str):

Complexity: *O*(?)

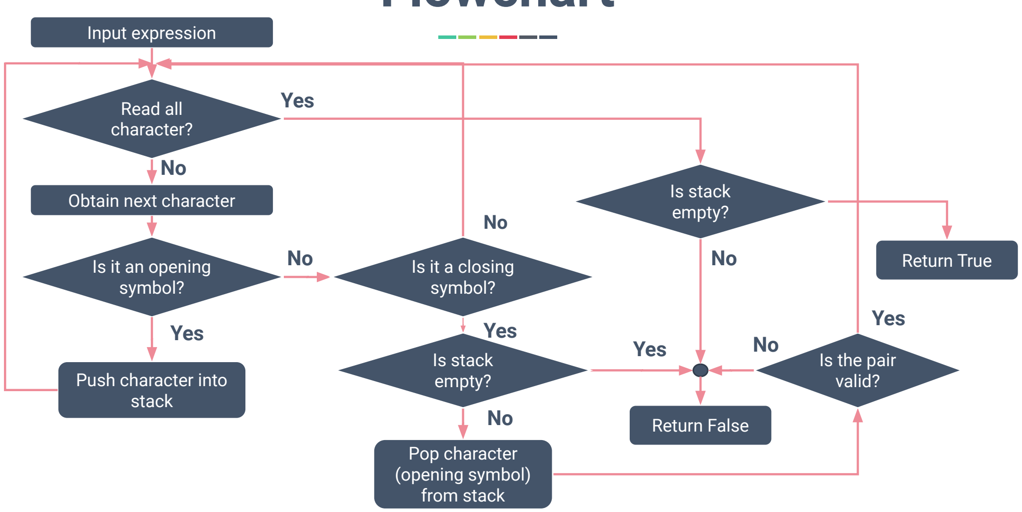
Stack *stack*
lefty = ‘([{‘
righty = ‘)]}’
for *char* in *str*:
    if *char* in *lefty*:
        stack.push(char)
    else if *char* in *righty*:
        If *stack*.is_empty():
            return False
        top = *stack*.pop()
        if (*top* = ‘(‘ and *char* != ‘)’) or (*top* = ‘[‘ and *char* != ‘]’) or
        (*top* = ‘{‘ and *char* != ‘}’)
            return False
return *stack*.is_empty()

Incorrect order, such as] [

Incorrect case, such as [ )

Missing case, such as (

Flowchart



Recursion

- To repeat a computer program, Python’s **while-loop** and **for-loop** can be used.
- Alternatively, recursion repeats by calling a function itself one or more times.
- Real-life example: Matryoshka doll
  - Also known as Russian doll or nested doll
  - Set of wooden dolls where smaller dolls are placed inside another.
  - A symbol of motherhood and fertility.

Recursion

- Examples
  - Summation
  - Power
  - Factorial
  - Binary search
- One rule for recursion: there must be a stop condition.

Recursion

- Summing list elements recursively.

```
def getSum(data):
  """ Return the sum of the first n numbers of sequence data."""
  if len(data)==0:
      return 0
  else:
      return data[0] + getSum(data[1:])
```

Time Complexity: *O*(?)

Recursion

- Power function:

$$\text{power(base, exp)} = \begin{cases} 1 & \text{if exp} = 0 \\ \text{base} * \text{power (base, exp-1)} & \text{otherwise} \end{cases}$$

```
def getPower(base,exp):
  """ Return the multiplciation of base with exp times.
"""
  if exp==0:
    return 1
  else:
    return base*getPower(base, exp-1)
```

Time Complexity: *O*(?)

Recursion

- Factorial
  - n! = Product of integers from 1 to n
  - If n = 0, then n! = 1 by convention
- Formally, for any integers n>=0,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) * \ldots 3 * 2 * 1 & \text{if } n >= 1. \end{cases}$$

Recursion

- Factorial
  - Used to calculate the number of **permutations** of n items.
  - That is, given *n* items, how many ways to arrange them into a sequence.
  - For instance, 3 characters: 1, 2, 3 can be arranged in 3! = 3*2*1 = 6 ways.
    - 123, 132, 213, 231, 312, 321

Recursion

- Recursive definition of factorial function

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{(base case)} \\ n * (n-1)! & \text{if } n >= 1. \text{ (recursive case)} \end{cases}$$

- For example, 4! = 4 * 3!

Recursion

Loop

```
# Factorial loop version
def factorial1(n):
  if n == 0:
    return 1
  else:
    total = 1
    for num in range(2,n+1):
      total *= num
    return total
```

Time Complexity: *O*(?)

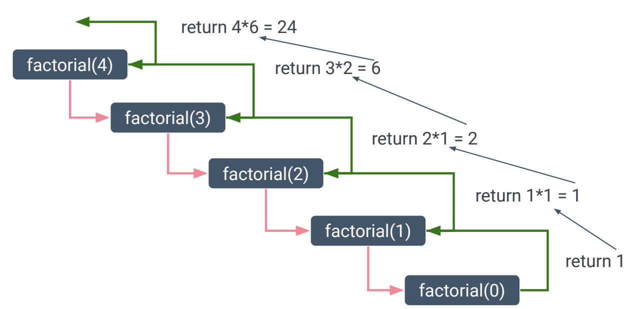
Recursive

```
# Factorial recursive version
def factorial2(n):
  if n == 0:
    return 1
  else:
    return n*factorial2(n-1)
```

Time Complexity: *O*(?)

Recursion

- For recursive, repetition is conducted by repeatedly invoke the function call.



Recursion

- Binary Search
  - Locate a target value in a sequence of *n* elements that are sorted.
  - mid = (low + high) / 2
  - Initially, low = 0, high = n-1
- For instance, find number 5.

Data	1	5	7	9	10	11	20
Index	0	1	2	3	4	5	6

Recursion

- Binary Search
  - If target value < data[mid], next interval is from low to mid-1.
  - If target value > data[mid], next interval is from mid + 1 to high.

	low			mid			high
Data	1	5	7	9	10	11	20
Index	0	1	2	3	4	5	6

mid = (0 + 6) / 2 = 3

	low	mid	high				
Data	1	5	7	9	10	11	20
Index	0	1	2	3	4	5	6

mid = (0 + 2) / 2 = 1

Recursion


```python
# Iterative version
def binary_search_iterative(data, target):
  """ Return True if target is found in the given Python list."""
  low = 0
  high = len(data)-1
  while low <= high:
    mid = (low + high) // 2
    if target == data[mid]:
      return True
    elif target < data[mid]:
      # only consider the left portion left of the middle
      high = mid - 1
    else:
      # only consider the right portion of the middle
      low = mid + 1
 return False
```

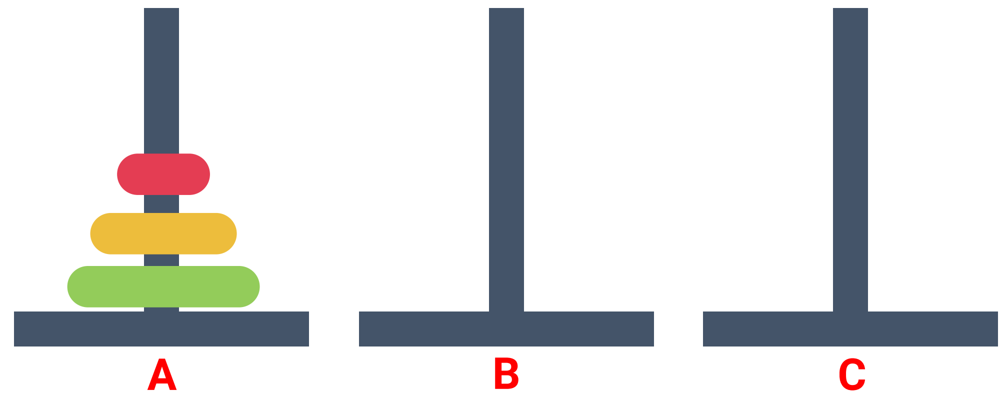
Recursion


```python
def binary_search(data, target, low, high):
  """ Return True if target is found in indicated portion of a Python list.
  The search only considers the portion from data[low] to data[high] inclusive.
"""
  if low > high:
    return False       # interval is empty; no match
  else:
    mid = (low + high) // 2   # divide without remaining decimals
    if target == data[mid]:
      return True
    elif target < data[mid]:
      # recur on the left portion of the middle
      return binary_search(data, target, low, mid-1)
    else:
      # recur on the right portion of the middle
      return binary_search(data, target, mid+1, high)
```

Recursion

- Binary Search for a sorted sequence
  - Running time is proportional to the number of recursive calls executed, which is O(log n).

Towers of Hanoi



Tower of Hanoi

tower(disk, source, intermediate, destination)

IF disk is equal 1, THEN

    move disk from source to destination

ELSE

    tower(disk - 1, source, destination, intermediate)   // Step 1

    move disk from source to destination   // Step 2

    tower(disk - 1, intermediate, source, destination)   // Step 3

END IF

END

Tower of Hanoi

Time Complexity: $O(2^n)$