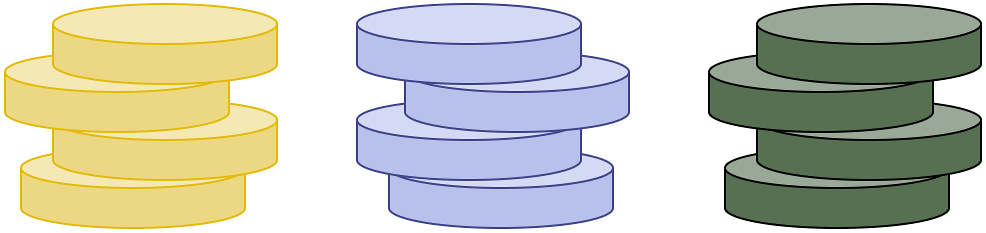
Stacks

Algorithms & Data Structures
ITCS 6114/8114

Dr. Dewan Tanvir Ahmed
Department of Computer Science
University of North Carolina at Charlotte

Stacks



Outline and Reading

- The Stack ADT (§2.1.1)
- Applications of Stacks (§2.1.1)
- Array-based implementation (§2.1.1)
- Growable array-based stack (§1.5)

Abstract Data Types (ADTs)

- It is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with the operations

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element

- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception

    - In the Stack ADT, operations pop and top cannot be performed if the stack is empty

Applications of Stacks

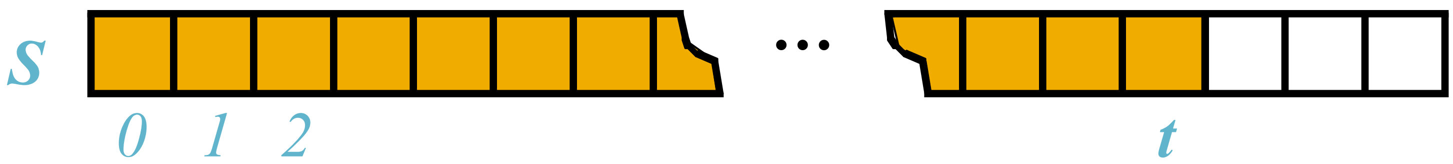
- Direct applications
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Page-visited history in a Web browser

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

- A simple way – use array
- add elements from left to right
- A variable keeps track of the index of the top element

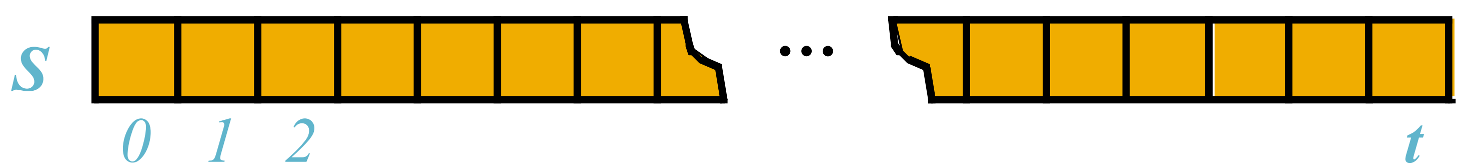
```
Algorithm size()
  return t + 1

Algorithm pop()
  if isEmpty() then
    throw EmptyStackException
   else
    t ← t − 1
  return S[t + 1]
```



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
  if t = S.length − 1 then
    throw FullStackException
   else
    t ← t + 1
    S[t] ← o
```



Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori
  - Trying to push a new element into a full stack causes an implementation-specific exception

Application #1: Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Application #1: Parentheses Matching Algorithm

Algorithm ParenMatch(X,n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
        S.push(X[i])
    else if X[i] is a closing grouping symbol then
        if S.isEmpty() then
            return false {nothing to match with}
        if S.pop() does not match the type of X[i] then
            return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else
    return false {some symbols were never matched}
```

Growable Array-based Stack

- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant ***c***
  - doubling strategy: double the size

```
Algorithm push(item)
  if t = S.length − 1 then
    A ← new array of
          size …
    for i ← 0 to t do
      A[i] ← S[i]
    S ← A
  t ← t + 1
  S[t] ← item
```

Comparison of the Strategies

- Compare incremental strategy and doubling strategy
  - by analyzing the total time $\boldsymbol{T}(\boldsymbol{n})$ needed to perform a series of $\boldsymbol{n}$ push operations
- Assume that we start with an empty stack represented by an array of size 1
- Amortized time of a push operation the average time taken by a push over the series of operations, i.e., $\boldsymbol{T}(\boldsymbol{n})/\boldsymbol{n}$

Incremental Strategy Analysis

- We replace the array $\boldsymbol{k} = \boldsymbol{n}/\boldsymbol{c}$ times

- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ push operations is proportional to
$$\boldsymbol{n} + \boldsymbol{c} + 2\boldsymbol{c} + 3\boldsymbol{c} + 4\boldsymbol{c} + \ldots + \boldsymbol{kc}$$
$$= \boldsymbol{n} + \boldsymbol{c}(1 + 2 + 3 + \ldots + \boldsymbol{k}) = \boldsymbol{n} + \boldsymbol{ck}(\boldsymbol{k} + 1)/2$$

- Since $\boldsymbol{c}$ is a constant, $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n} + \boldsymbol{k}^2)$, i.e., $\boldsymbol{O}(\boldsymbol{n}^2)$
- The amortized time of a push operation is $\boldsymbol{O}(\boldsymbol{n})$

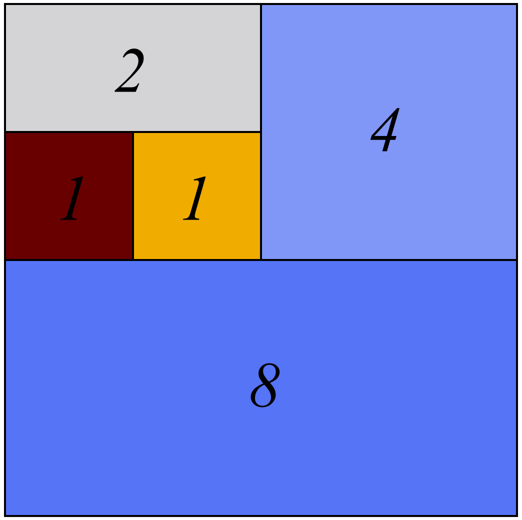
Doubling Strategy Analysis

- We replace the array $\boldsymbol{k} = \log_2 \boldsymbol{n}$ times
- The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ push operations is proportional to

$$\boldsymbol{n} + 1 + 2 + 4 + 8 + \ldots + 2^{\boldsymbol{k}} =$$
$$\boldsymbol{n} + 2^{\boldsymbol{k}+1} - 1$$

- $\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n})$
- The amortized time of a push operation is $\boldsymbol{O}(1)$

geometric series



Reference

- Algorithm Design: Foundations, Analysis, and Internet Examples. Michael T. Goodrich and Roberto Tamassia. John Wiley & Sons.
- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

Thank you!