CS213/CS213M DS&A

Sharat Chandran

www.cse.iitb.ac.in/~sharat/current/213m

Agenda

- Basic Data Structures
- Abstract Data Type
- Dynamic Allocation and Amortized Complexity

Abstract Data Types

- An abstract data type (ADT) is an abstraction (functional)
- Key point: Algorithm or implementation is not specified
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

- Example: A simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

The Stack ADT

- The Stack ADT stores arbitrary objects and an arbitrary number of these elements
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Insertions and deletions follow the last-in first-out scheme
- Why Stack? Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in a language that supports recursion
- Indirect applications: Auxiliary data structure

Implementation

- How can we practically use a stack?
- C++
  - Using STL (or roll your own)
- Python
  - Using list(), or the collections framework, (or roll your own)
- Performance of data structure is dependent on the implementation
  - We will approximate what has been done by the gurus to understand
  - Exposes us to memory management

Demo Before Implementation

- How to implement *push()* and *pop()* using *list?*
- How much memory does an empty *list* consume?
  - Maybe the number of actual elements, reference to actual data
- How much time does it take to push?

Agenda

- Abstract Data Type
- Basic Data Structures
  - Stack with an unlimited amount of entries
- Dynamic Allocation
  - Wanted to avoid hardcoded numbers like `int a[100000]`
- Key experimental observations: In Python's list implementation,
  - The list grew unpredictably large, so surely some operations are expensive?
  - Yet, it takes the same time per insert, i.e., inserting 100 million items was as cheap as inserting 1000 items
- Can we mathematically quantify this?

The Stack ADT

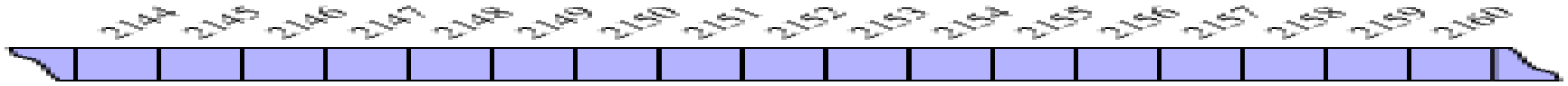
- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer len(): returns the number of elements stored
  - boolean is_empty(): indicates whether no elements are stored

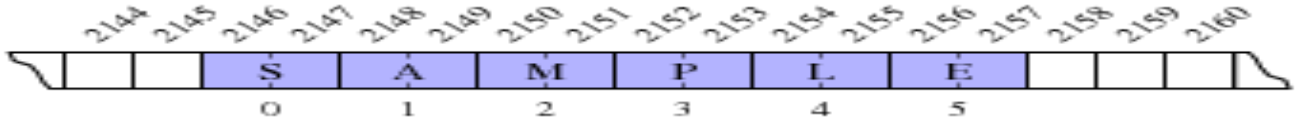
Example

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

Raw Material for Implementation

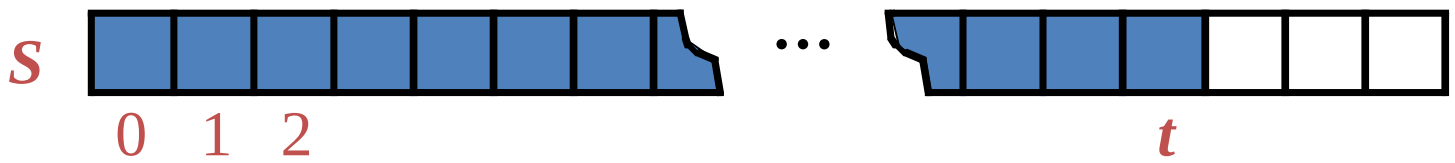
- A chunk of memory (often main memory or RAM) is available
- The address space is supposed to be with O(1) access
  - As easy to retrieve 0x9867231 as 0x29
  - Internal address arithmetic start + cellsize * index





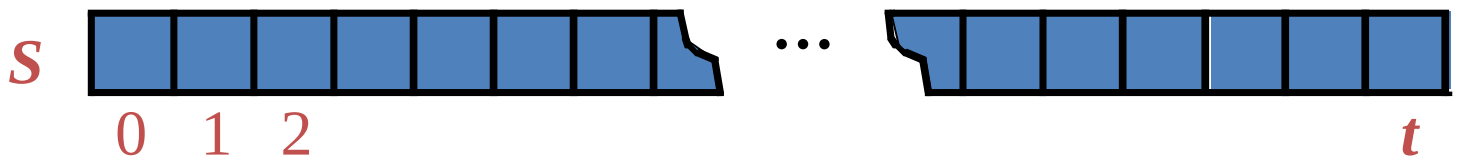
Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element



Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then need to grow the array and copy all the elements over.



Growth

- In an append operation, we always add at the end
- When the array is full, we call the **memory manager**!
- How large should the new array be?
  - Incremental strategy: increase the size by a LARGE constant c
  - Doubling strategy: double the size

```
Algorithm append(o)
  if t = S.length – 1 then
    A ← new array of
          size …
    for i ← 0 to n–1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n–1] ← o
```

Comparison of the Strategies

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

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n append operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

Doubling Strategy Analysis

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

  $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
  $3n - 1$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series