Algorithms & Data Structures
Dr Michele Sevegnani

Joao Almeida-Domingues[*]

University of Glasgow

January 14th, 2020 – March 25th, 2020

Contents

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

[*]2334590D@student.gla.ac.uk

1 Abstract Data Types

1.1 definition. ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations

ADTs are abstractions of the structure of data from the data itself, they define behaviour and state, a contract whose concrete instances must follow. When implemented in code we do so via *data structures*. For example, a `List` can be implemented as an `array`

1.2 remark. In `Java` you can think in terms of `Interfaces` and `Classes`

1.1 *Stack*

1.3 definition. Stack is a collection of objects that are inserted and removed following the *LIFO* principle

1.4 remark. Web browsers' histories use stacks, and undo sequences in most other software work in a similar manner

Operations

- Essential Update Methods
  1. push
  2. pop
- Accessor Methods
  1. top : "pop" without removing
  2. size
  3. isEmpty

Implementation - Array

The array implementation of a stack is simple and efficient *if* one has a good idea , in advance , of the number of elements it will contain. Then, given n, t where n represents the size of the stack and t is a cursor var which keeps track of the index of top element, one adds elements from [0...n-1=t]

1.5 remark. by convention , $S = \emptyset \iff t = -1$

1.6 definition. Stack Overflow push() into full stack

1.7 definition. Stack Underflow pop() empty stack

Operations

Algorithm 1: isEmpty

Data: Stack S
Result: true, false

```
STACK-EMPTY(S)
    return S.top = -1
```

Algorithm 2: Push

Data: Stack S , x

```
PUSH(S,x)
    S.top := S.top + 1
S[S.top] := x
```

Algorithm 3: Pop

Data: Stack S
Result: "underflow", S[S.top + 1]

```
POP(S)
if STACK-EMPTY(S) then
    return "underflow"
else
    S.top := S.top - 1
    return S[S.top + 1]
```

Analysis

Each method executes a constant number of statements involving arithmetic operations, comparisons, and assignments, or calls to size and isEmpty, which both run in constant time, i.e they're all $O(1)$, and memory is $O(n)$

Implementation - Dynamic Array

One can overcome the fixed size constraint by adding a `RESIZE` operation, allowing the array to both shrink and grow.

Simple implementations are doubling its size when full, and half it when it is one quarter full. So, when an overflow is detected the following happens:

1. Allocate a new array S' with larger capacity
2. Set $S'[k] = S[k], k = 0, \ldots, n-1$
3. Set $S = S'$

Operations

Algorithm 4: Resize

```
Data: Stack S , new capacity n′
RESIZE(S, n′)
    new S′[0..n′-1]
    for i = 0 to S.top do
    |   S′[i] := S[i]
    end
    S := S′
```

Algorithm 5: Push

```
Data: Stack S , x
PUSH(S,x)
if S.top = n -1 then
|   RESIZE(S, 2*n)                     /* double array if full */
    S.top := S.top + 1
S[S.top] := x
```

Algorithm 6: Pop

```
Data: Stack S
Result: ”underflow”, S[S.top + 1]
POP(S)
if STACK-EMPTY(S) then
|   return ”underflow”
else
|   x := S[S.top]
|   S.top := S.top - 1
|   if S.top ¿ 0 and S.top = n/4 then
|   |   RESIZE(S,n/2)                  /* half array if 1/4 full */
|   end
|   return S[S.top + 1]
```

Analysis

Memory : $O(kn)$

see L.8.35 for amortised analysis

Running Time : O(n) , since when expanding by a constant proportion each insertion takes *amortised* constant time

Implementation - Linked List

Let `l.head := S.top` , then `PUSH = INSERT@Head` , `POP = DEL@Head` and by keeping track of size with `S.size` we can perform `SIZE` at constant time

Operations

Algorithm 7: Push

```
Data: Stack S , x
PUSH(S,x)
    x.next := S.top                         /* S.top == L.head */

S[S.top] := x
```

Algorithm 8: Pop

```
Data: Stack S
Result: "underflow", x
POP(S)
if S.top != NILL then
    x := S.top
    S.top := S.top.next
    return x
else
    return "underflow"
```

Analysis

Memory : $O(n)$

Running Time : O(1)

References

CS2 Software Design & Data Structures **calculating˙program˙running˙time**

CS2 Software Design & Data Structures. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/AnalProgram.html.

Divide-and-conquer algorithm **divideConquerWiki**

Divide-and-conquer algorithm. Jan. 2020. URL: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm.

Goodrich et al.: Data Structures and Algorithms in Java, 6th Edition **goodrich˙tamassia˙2014**

Michael Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 6th Edition*. John Wiley & Sons, 2014.

Hochstein: What is tail recursion? **hochsteinSO**

Lorin HochsteinLorin Hochstein. *What is tail recursion?* Aug. 1958. URL: https://stackoverflow.com/questions/33923/what-is-tail-recursion.

Merging Algorithms **merge**

Merging Algorithms. URL: http://www.cs.rpi.edu/~musser/gp/algorithm-concepts/merge-algorithms-screen.pdf.