# 2 Fundamental Data Structures

5 October 2022

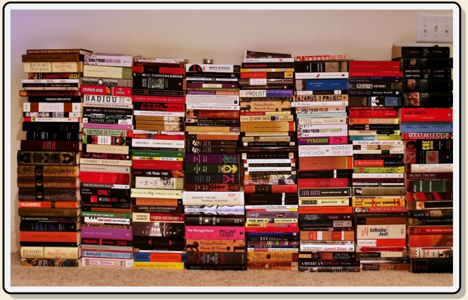
Sebastian Wild

COMP526 (Fall 2022)
University of Liverpool
*version* `2022-10-04 18:14`

# Learning Outcomes

1. Understand and demonstrate the difference between *abstract data type (ADT)* and its *implementation*
2. Be able to define the ADTs *stack*, *queue*, *priority queue* and *dictionary / symbol table*
3. Understand *array*-based implementations of stack and queue
4. Understand *linked lists* and the corresponding implementations of stack and queue
5. Know *binary heaps* and their performance characteristics
6. Understand *binary search trees* and their performance characteristics

## Unit 2: *Fundamental Data Structures*



# Outline

## 2 Fundamental Data Structures

# Recap: The Random Access Machine

- Data structures make heavy use of pointers and dynamically allocated memory.

- Recall: Our RAM model supports
  - basic pseudocode (≈ simple Python code)
  - creating arrays of a fixed/known size.
  - creating instances (objects) of a known class.

Python abstracts this away!

no predefined capacity!

**There are *no arrays* in Python, only its built-in *lists*.**

But: Python *implementations create* lists based on fixed-size arrays (stay tuned!)

Python ≠ RAM: $\boxed{\text{Not every built-in Python instruction runs in } O(1) \text{ time!}}$

## 2.1 Stacks & Queues

# Abstract Data Types

**abstract data type (ADT)**

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

abstract base classes

≈ Java `interface`, Python `ABCs`
(with comments)

VS.

**data structures**

- specify exactly
  **how** data is represented
- **algorithms** for operations
- has concrete costs
  (space and running time)

≈ Java/Python `class`
(non abstract)

# Abstract Data Types

**abstract data type (ADT)**

- list of supported operations
- **what** should happen
- **not:** how to do it
- **not:** how to store data

abstract base classes

≈ Java `interface`, Python `ABC`s
(with comments)

VS.

**data structures**

- specify exactly
  **how** data is represented
- **algorithms** for operations
- has concrete costs
  (space and running time)

≈ Java/Python `class`
(non abstract)

*Why separate?*

- Can swap out implementations ⇝ "drop-in replacements"

⇝ **reusable code**!

- (Often) better abstractions
- Prove generic lower bounds ( ⇝ Unit 3)

# Abstract Data Types

**abstract data type (ADT)**

- list of supported operatio
- **what** should happen
- **not:** how to do it
- **not:** how to store data

abst

≈ Java `interface`, Python A
(with comments)

*Why separate?*

- Can swap out implement

⇝ **reusable code**!

- (Often) better abstractions
- Prove generic lower bounds ( ⇝ Unit 3)

## Clicker Question

Which of the following are examples of abstract data types?

- **A** ADT
- **B** Stack
- **C** Deque
- **D** Linked list
- **E** binary search tree
- **F** Queue
- **G** resizable array
- **H** heap
- **I** priority queue
- **J** dictionary/symbol table
- **K** hash table

→ sli.do/comp526

# Clicker Question

Which of the following are examples of abstract data types?

- **A** ~~ADT~~
- **B** Stack
- **C** Deque
- **D** ~~Linked list~~
- **E** ~~binary search tree~~
- **F** Queue
- **G** ~~resizable array~~
- **H** ~~heap~~
- **I** priority queue
- **J** dictionary/symbol table
- **K** ~~hash table~~

→ sli.do/comp526

# Stacks

**Stack ADT**

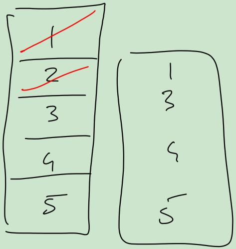
- `top()`
  Return the topmost item on the stack
  Does not modify the stack.
- `push(`$x$`)`
  Add $x$ onto the top of the stack.
- `pop()`
  Remove the topmost item from the stack
  (and return it).
- `isEmpty()`
  Returns `true` iff stack is empty.
- `create()`
  Create and return an new empty stack.

# Clicker Question

Suppose a stack initially contains the numbers 1, 2, 3, 4, 5 with 1 at the top.
What is the content of the stack after the following operations:

```
pop(); pop(); push(1);
```

- **A** 1,2,3,1
- **B** 3,4,5,1
- **C** 1,3,4,5
- **D** empty
- **E** 1,2,3,4,5




→ sli.do/comp526

# Clicker Question

Suppose a stack initially contains the numbers $1, 2, 3, 4, 5$ with $1$ at the top.
What is the content of the stack after the following operations:

```
pop(); pop(); push(1);
```

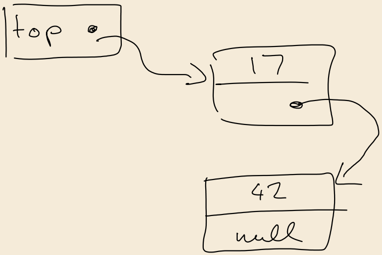
- **A** ~~1,2,3,1~~
- **B** ~~3,4,5,1~~
- **C** 1,3,4,5
- **D** ~~empty~~
- **E** ~~1,2,3,4,5~~

→ sli.do/comp526

# Linked-list implementation for Stack

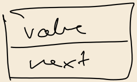
**Invariants:**

- maintain pointer *top* to topmost element
- each element points to the element below it (or `null` if bottommost)




```
class Node
    value
    next

class Stack
    top := null
    procedure top()
        return top.value
    procedure push(x)
        top := new Node(x, top)
    procedure pop()
        t := top()
        top := top.next
        return t
```




isEmpty

# Linked-list implementation for Stack – Discussion

**Linked stacks:**

- require $\Theta(n)$ space when $n$ elements on stack
- All operations take $O(1)$ time

- require $\Theta(n)$ space when $n$ elements on stack

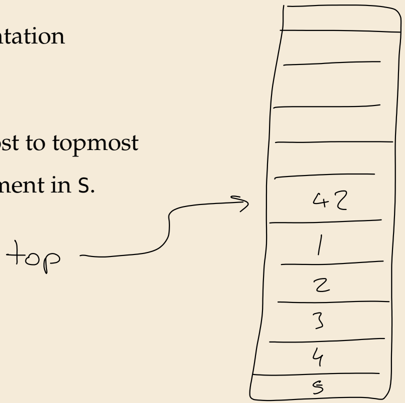
  Can we avoid extra space for pointers?

# Array-based implementation for Stack

If we want no pointers $\rightsquigarrow$ array-based implementation

**Invariants:**

- maintain array $S$ of elements, from bottommost to topmost
- maintain index $top$ of position of topmost element in S.



# Array-based implementation for Stack

If we want no pointers $\rightsquigarrow$ array-based implementation

**Invariants:**

- maintain array $S$ of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in `S`.

What to do if stack is full upon `push`?

**Array stacks:**

- require *fixed capacity* $C$ (decided at creation time)!
- require $\Theta(C)$ space for a capacity of $C$ elements
- all operations take $O(1)$ time

# 2.2 Resizable Arrays

# *Digression* – Arrays as ADT

Arrays can also be seen as an ADT!

**Array operations:**

- `create(`$n$`)` *Java*: `A = new int[`$n$`];` *Python:* `A = [0] * ` $n$
  Create a new array with $n$ cells, with positions $0, 1, \ldots, n-1$;
  we write $A[0..n) = A[0..n-1]$
- `get(`$i$`)` *Java/Python*: `A[`$i$`]`
  Return the content of cell $i$
- `set(`$i$`,`$x$`)` *Java/Python*: `A[`$i$`] = `$x$`;`
  Set the content of cell $i$ to $x$.

$\rightsquigarrow$ Arrays have *fixed* size (supplied at creation). ($\neq$ `lists` in Python)

# *Digression* – Arrays as ADT

Arrays can also be seen as an ADT! . . . but are commonly seen as specific data structure

**Array operations:**

- `create(n)` *Java*: `A = new int[n];` *Python:* `A = [0] * n`
  Create a new array with $n$ cells, with positions $0, 1, \ldots, n-1$;
  we write $A[0..n) = A[0..n-1]$

- `get(i)` *Java/Python*: `A[i]`
  Return the content of cell $i$

- `set(i,x)` *Java/Python*: `A[i] = x;`
  Set the content of cell $i$ to $x$.

$\rightsquigarrow$ Arrays have *fixed* size (supplied at creation). ($\neq$ `lists` in Python)

Usually directly implemented by compiler + operating system / virtual machine.

**Difference to "real" ADTs:** ***Implementation usually fixed***
to "a contiguous chunk of memory".

# Doubling trick

*Can we have unbounded stacks based on arrays?* Yes!

# Doubling trick

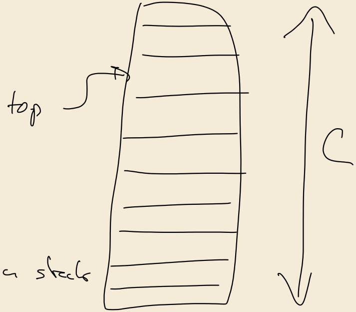
*Can we have unbounded stacks based on arrays?* Yes!

**Invariants:**

- maintain array $S$ of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in S
- maintain capacity $C = S.length$ so that $\frac{1}{4}C \le n \le C$

$n$ = # elements on stack

$\rightsquigarrow$ can always push more elements!



# Doubling trick

*Can we have unbounded stacks based on arrays?* Yes!

**Invariants:**

- maintain array $S$ of elements, from bottommost to topmost
- maintain index *top* of position of topmost element in S
- maintain capacity $C = S.length$ so that $\frac{1}{4}C \leq n \leq C$

$\rightsquigarrow$ can always push more elements!

*How to maintain the last invariant?*

- before `push`
  If $n = C$, allocate new array of size $2n$, copy all elements.
- after `pop`
  If $n < \frac{1}{4}C$, allocate new array of size $2n$, copy all elements.

$\rightsquigarrow$ ***"Resizing Arrays"***

an implementation technique, not an ADT!

# Clicker Question

Which of the following statements about resizable array that currently stores $n$ elements is correct?

**A** The elements are stored in an array of size $2n$.

**B** Adding or deleting an element at the end takes constant time.

**C** A sequence of $m$ insertions or deletions at the end of the array takes time $O(n + m)$.

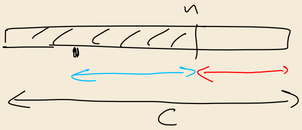
**D** Inserting and deleting any element takes $O(1)$ amortized time.

→ sli.do/comp526

# Amortized Analysis

- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost $T$ means $\Omega(T)$ next operations are cheap!

# Amortized Analysis

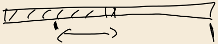



- Any individual operation `push` / `pop` can be expensive!
  $\Theta(n)$ time to copy all elements to new array.
- **But:** An one expensive operation of cost $T$ means $\Omega(T)$ next operations are cheap! $\frac{1}{4}C \le n \le C$

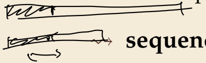
distance to boundary

since $n \le C \le 4n$

**Formally:** consider "credits/potential" $\Phi = \min\{\underline{n - \frac{1}{4}C}, \underline{C - n}\} \in [0, 0.6n]$



- amortized cost of an operation $=$ actual cost (array accesses) $- 4 \cdot$ change in $\Phi$
  - cheap `push`/`pop`: actual cost $1$ array access, consumes $\le 1$ credits $\rightsquigarrow$ amortized cost $\le 5$
  - copying `push`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n + 1$ credits $\rightsquigarrow$ amortized cost $\le 5$
  - copying `pop`: actual cost $2n + 1$ array accesses, creates $\frac{1}{2}n - 1$ credits $\rightsquigarrow$ amortized cost $5$



$\rightsquigarrow$ **sequence** of $m$ operations: total actual cost $\le$ total amortized cost $+$ final credits

here: $\le \quad 5m \quad + \quad 4 \cdot 0.6n \quad = \Theta(m + n)$

$$a_i = c_i - 4(\phi_i - \phi_{i-1}) \leq 5$$

$$\sum_{i=1}^{m} a_i \leq 5m \geq \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i - 4 \underbrace{\sum_{i=1}^{m} (\phi_i - \phi_{i-1})}_{\phi_m - \phi_0}$$

$$\sum_{i=1}^{m} c_i \leq 5m + 4\phi_m - 4\phi_0$$

$$\leq 5m + 4\phi_m$$

# Clicker Question

Which of the following statements about resizable array that currently stores $n$ elements is correct?

**A** The elements are stored in an array of size $2n$.

**B** Adding or deleting an element at the end takes constant time.

**C** A sequence of $m$ insertions or deletions at the end of the array takes time $O(n + m)$.

**D** Inserting and deleting any element takes $O(1)$ amortized time.

→ sli.do/comp526

# Clicker Question

Which of the following statements about resizable array that currently stores $n$ elements is correct?

- **A** ~~The elements are stored in an array of size~~ $\cancel{2n}$~~.~~
- **B** ~~Adding or deleting an element at the end takes constant time.~~
- **C** A sequence of $m$ insertions or deletions at the end of the array takes time $O(n + m)$. ✓
- **D** ~~Inserting and deleting any element takes~~ $\cancel{O(1)}$ ~~amortized time.~~

→ sli.do/comp526

# Queues

**Operations:**

- `enqueue(`$x$`)`
  Add $x$ at the end of the queue.

- `dequeue()`
  Remove item at the front of the queue and return it.

Implementations similar to stacks.

# Bags

*What do Stack and Queue have in common?*

# Bags

*What do Stack and Queue have in common?*

They are special cases of a ***Bag***!

**Operations:**

- `insert(`$x$`)`
  Add $x$ to the items in the bag.
- `delAny()`
  Remove any one item from the bag and return it.
  (Not specified which; any choice is fine.)
- roughly similar to Java's `java.util.Collection`
  Python's `collections.abc.Collection`

Sometimes it is useful to state that order is irrelevant $\rightsquigarrow$ Bag
Implementation of Bag usually just a Stack or a Queue