3 Fundamental Data Structures

21 October 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg

version `2025-10-20 15:35`

Learning Outcomes

Unit 3: ***Fundamental Data Structures***

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
7. Know high-level idea of basic *hashing strategies* and their performance characteristics

Outline

3 Fundamental Data Structures

Clicker Question

What's the running time (on our word-RAM model with word size w) of this Java instruction?

`Object[] A = new Object[`n`];`

- **A** 1
- **B** $\Theta(1)$
- **C** $\Theta(\log n)$
- **D** $\Theta(w)$
- **E** $\Theta(n/w)$
- **F** $\Theta(n)$
- **G** $\Theta(n \log n)$
- **H** $\Theta(nw)$
- **I** $\Theta(n^2)$

→ sli.do/cs566

Clicker Question

What's the running time (on our word-RAM model with word size w) of this Java instruction?

```
Object[] A = new Object[n];    // n·w bit
```

- **A** $\cancel{1}$
- **B** $\cancel{\Theta(1)}$
- **C** $\cancel{\Theta(\log n)}$
- **D** $\cancel{\Theta(w)}$
- **E** $\cancel{\Theta(n/w)}$ ✓
- **F** $\Theta(n)$ ✓
- **G** $\cancel{\Theta(n \log n)}$
- **H** $\cancel{\Theta(nw)}$
- **I** $\cancel{\Theta(n^2)}$

→ sli.do/cs566

Recap: The Random Access Machine

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

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

Recap: The Random Access Machine

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

- Recall: Our RAM model supports
  - basic pseudocode ($\approx$ simple Python/Java 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!)

Java

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

3.1 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 `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)

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 implemen
- ⇝ **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/cs566`

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/cs566

Stacks

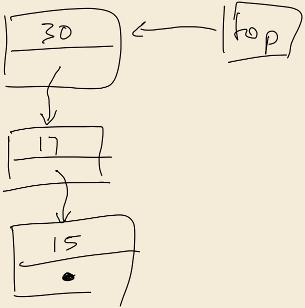
Stack ADT

- `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.

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
```

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

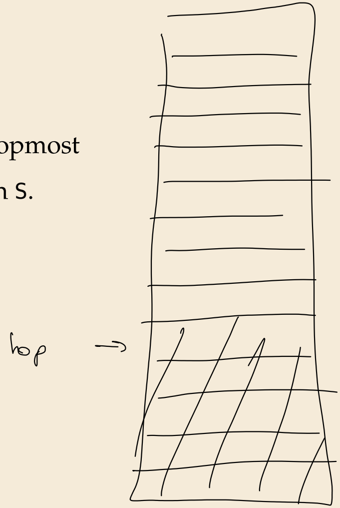
Can we avoid extra space for pointers?

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:

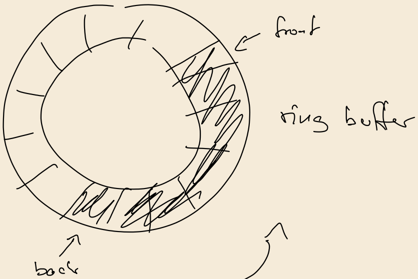
- require *fixed capacity* C (decided at creation time)!
- require $\Theta(C)$ space for a capacity of C elements
- all operations take $O(1)$ time

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***!

Update 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`
- always support iterating over content (read only)

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

3.2 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 $\frac{1}{4}C \le n \le C$

$\rightsquigarrow$ 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 $\frac{1}{4}C \le n \le C$

$\rightsquigarrow$ 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?

always

- **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/cs566

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!

Amortisierte Analyse

c_i = echte Kosten von Operation i

Φ_i = "Potential" nach Operation i

a_i = amortisierte Kosten

$$:= c_i + \alpha \cdot (\Phi_i - \Phi_{i-1})$$

Ziel: Zeigen, dass $a_i \leq A$

$$m \cdot A \geq \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \Big(c_i + \alpha \underbrace{(\Phi_i - \Phi_{i-1})}_{\text{Teleskopsumme!}}\Big) = \sum_{i=1}^{m} c_i + \alpha(\Phi_m - \Phi_0)$$

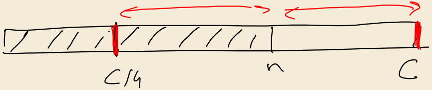
$$\Rightarrow \quad \sum_{i=1}^{m} c_i \leq m \cdot \underset{\substack{\parallel \\ 5}}{A} - \underset{\substack{\parallel \\ -4}}{\alpha} \left(\Phi_m - \Phi_0 \right)$$

für resizing arrays

$$\sum_{i=1}^{m} c_i \leq 5 \cdot m + 4 \cdot \Phi_m \leq \underline{5m + 2 \cdot 4 \cdot n}$$

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!



distance to boundary

since $n \leq C \leq 4n$

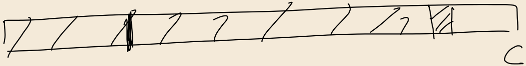
Formally: consider "credits/potential" $\Phi = \min\{n - \frac{1}{4}C, C - n\} \in [0, 0.6n]$

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

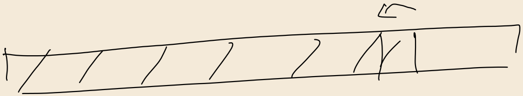
here: $\leq \quad 5m \quad + \quad 4 \cdot 0.6n \quad = \Theta(m + n)$

günstiges push / pop



C $\quad$ c_i

$\Delta\Phi = -1 \qquad a_i = \overset{c_i}{\overset{\shortparallel}{1}} - 4 \cdot \Delta\Phi = 5$

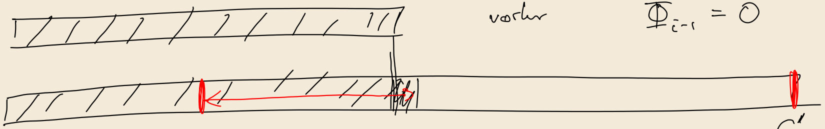


$\Delta\Phi = 1 \qquad a_i = 1 - 4 \cdot 1 = -3 \leq 5$

teure push

C

vorher $\qquad \Phi_{i-1} = 0$



nachher

$\approx \frac{n}{4} \qquad n \qquad n' = n+1 \qquad \Phi_i = \qquad C' = 2n$

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/cs566

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/cs566