

# Vectors, Lists, Stacks, Queues

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

# ADTVector

- A Vector can be seen as an abstract **resizable** “array”
  - It is **not** an array (remember, it's abstract)
  - but it behaves like one
- We can access existing elements based on their **position** (random access)
- We can insert and remove elements **at the end** of the vector (dynamic size)
- We can **search** for elements (but this is usually inefficient)
- We can **iterate** over elements (also possible using random access)

a.k.a. Dynamic/Growable/Resizable/Mutable Array, Array List, List, ...

# Create, destroy

```
// Ένα vector αναπαριστάται από τον τύπο Vector. Ο χρήστης δε χρειάζε  
// γνωρίζει το περιεχόμενο του τύπου αυτού, απλά χρησιμοποιεί τις συν  
// vector_<foo> που δέχονται και επιστρέφουν Vector.  
//  
// Ο τύπος Vector ορίζεται ως pointer στο "struct vector" του οποίου  
// περιεχόμενο είναι άγνωστο (incomplete struct), και εξαρτάται από τ  
// υλοποίηση του ADT Vector.  
  
typedef struct vector* Vector;  
  
// Δημιουργεί και επιστρέφει ένα νέο vector μεγέθους size, με στοιχεί  
// αρχικοποιημένα σε NULL. Αν δεν υπάρχει διαθέσιμη μνήμη επιστρέφει  
// VECTOR_FAIL.  
  
Vector vector_create(int size, DestroyFunc destroy_value);  
  
// Ελευθερώνει όλη τη μνήμη που δεσμεύει το vector vec.  
  
void vector_destroy(Vector vec);
```

An initial size is given at creation (ignore `destroy_value` for now).

# Random access

```
// Επιστρέφει την τιμή στη θέση pos του vector vec (μη ορισμένο αποτέ  
// pos < 0 ή pos >= size)
```

```
Pointer vector_get_at(Vector vec, int pos);
```

```
// Αλλάζει την τιμή στη θέση pos του Vector vec σε value. ΔΕΝ μεταβάλ  
// μέγεθος του vector, αν pos >= size το αποτέλεσμα δεν είναι ορισμέν
```

```
void vector_set_at(Vector vec, int pos, Pointer value);
```

- Example [4, 6, 2, 1]
  - Get at 1: 6
  - Set 8 at 0: [8, 6, 2, 1]

# Insert and delete at the end

```
// Προσθέτει την τιμή value στο _τέλος_ του vector vec. Το μέγεθος το  
// μεγαλώνει κατά 1. Αν δεν υπάρχει διαθέσιμη μνήμη το vector παραμέν  
// ήταν (αυτό μπορεί να ελεγχθεί με τη vector_size)
```

```
void vector_insert_last(Vector vec, Pointer value);
```

```
// Επιστρέφει την τιμή της τελευταίας θέσης του vector.  
// Το μέγεθος του vector μικραίνει κατά 1.
```

```
void vector_remove_last(Vector vec);
```

- The size of the vector is modified (in contrast to C arrays)!
- Example [4, 6, 2, 1]
  - Insert 3: [4, 6, 2, 1, 3]
  - Remove: [4, 6, 2, 1]

# Search

```
// Βρίσκει και επιστρέφει το πρώτο στοιχείο στο vector που να είναι ί  
// (με βάση τη συνάρτηση compare), ή NULL αν δεν βρεθεί κανένα στοιχε
```

```
Pointer vector_find(Vector vec, Pointer value, CompareFunc compare);
```

- Usually sequential search (remember, the implementation is not fixed!)
- Redundant, could be implemented by iterating

# Iteration

```
// Μέσω random access
```

```
int size = vector_size(vec);  
for (int i = 0; i < size; i++) {  
    int* value = vector_get_at(vec, i);  
    printf("%d\n", *value);  
}
```

```
// Μέσω κόμβων
```

```
for(VectorNode node = vector_first(vec);           // ξενικάμε από τον π  
    node != VECTOR_EOF;                             // μέχρι να φτάσουμε  
    node = vector_next(vec, node)) {               // μετάβαση στον επόμε  
  
    int* value = vector_node_value(vec, node);     // η τιμή του συγκεκρι  
    printf("value: %d\n", *value);  
}
```

# Memory management

- The memory reserved for the **vector itself** is managed by the module
- We are responsible for the **contents** (Pointers)
- Simple memory management:
  - `destroy_value` function to be called when a value is **removed**

```
Vector vec = vector_create(0, free);  
  
vector_insert_last(vec, strdup("foo"));  
vector_insert_last(vec, strdup("bar"));  
  
vector_remove_last(vec);    // free bar  
  
vector_destroy(vec);       // free foo (και destroy το ίδιο το vecto
```



# When to use Vectors

- General purpose containers
- When we need random access
- When we don't need to insert at random positions
- When we don't need efficient search

# ADTList

- We sacrifice random access for **insert/delete flexibility**
- Only **sequential** access
- We can insert and remove elements **anywhere**
- We can **search** for elements (but this is usually inefficient)
- We can **iterate** over elements in the order of insertion

a.k.a. Forward list (also, “List” sometimes means something else)

# Insert and delete anywhere

```
// Προσθέτει έναν νέο κόμβο __μετά__ τον node, ή στην αρχή αν node ==  
// με περιεχόμενο value.
```

```
void list_insert_next(List list, ListNode node, Pointer value);
```

```
// Αφαιρεί τον __επόμενο__ κόμβο από τον node, ή τον πρώτο κόμβο αν n
```

```
void list_remove_next(List list, ListNode node);
```

- Positions represented by **nodes**
- Insert/remove happens **after** the given node
- Example (4, 6, 2, 1)
  - Insert 3 after 6: (4, 6, 3, 2, 1)
  - Remove after 4: (4, 3, 2, 1)

# Iteration

```
// Μόνο μέσω κόμβων
```

```
for(ListNode node = list_first(list);           // ξεκινάμε από τον πρ  
    node != LIST_EOF;                          // μέχρι να φτάσουμε σ  
    node = list_next(list, node)) {           // μετάβαση στον επόμε  
  
    int* value = list_node_value(list, node); // η τιμή του συγκεκρι  
    printf("value: %d\n", *value);  
}
```

# Other functions

Same as for Vectors.

```
List list_create(DestroyFunc destroy_value);  
  
// Επιστρέφει τον αριθμό στοιχείων που περιέχει η λίστα.  
  
int list_size(List list);  
  
// Επιστρέφει την πρώτη τιμή που είναι ισοδύναμη με value  
// (με βάση τη συνάρτηση compare), ή NULL αν δεν υπάρχει  
  
Pointer list_find(List list, Pointer value, CompareFunc compare);  
  
// Ελευθερώνει όλη τη μνήμη που δεσμεύει η λίστα list.  
// Οποιαδήποτε λειτουργία πάνω στη λίστα μετά το destroy είναι μη ορι  
  
void list_destroy(List list);
```

# When to use Lists

- General purpose containers
- When sequential access is enough
- When we need to insert/delete at random positions
- When we don't need efficient search

# Stacks

- Very limited functionality
  - but useful in practice
  - allows for efficient implementations
- Insert and delete at the **top**
  - Last-in, first-out (LIFO)
- Access only the **top** element
  - No random access
  - No iteration

# Examples of Stacks in Real Life





# LIFO access

```
// Επιστρέφει το στοιχείο στην κορυφή της στοίβας (μη ορισμένο αποτέλ  
// στοίβα είναι κενή)  
  
Pointer stack_top(Stack stack);  
  
// Προσθέτει την τιμή value στην κορυφή της στοίβας stack.  
  
void stack_insert_top(Stack stack, Pointer value);  
  
// Αφαιρεί την τιμή στην κορυφή της στοίβας (μη ορισμένο  
// αποτέλεσμα αν η στοίβα είναι κενή)  
  
void stack_remove_top(Stack stack);
```

- Example [4, 6, 2, 1]
  - Insert 3: [4, 6, 2, 1, 3]
  - Remove: [4, 6, 2, 1]
- Commonly called **push** and **pop**

# When to use Stacks

- When LIFO access is enough
- Many applications
  - Storing information of active function calls
  - Parsing algorithms
  - Expression evaluation algorithms
  - Backtracking algorithms
  - ...

# Using a Stack to check for balanced parentheses

- Determine whether parentheses/brackets balance properly in algebraic expressions.

- Example:

$$\{a^2 - [(b + c)^2 - (d + e)^2] * [\sin(x - y)]\} - \cos(x + y)$$

- This expression contains parentheses, square brackets, and braces in balanced pairs according to the pattern

$$\{[(())][()]\}()$$

# The Algorithm

- Start with an **empty stack**
- Scan the algebraic expression from left to right
  - On `(, [, {` we **insert** it to the stack.
  - On `), ], }` we **remove** the top item and check that its type matches
- The expression is balanced **iff**
  - all pairs match, and
  - at the end the stack is **empty**

# Postfix Expressions

- Expressions are usually written in **infix** notation  $L \text{ op } R$ 
  - The operator appears **between** the operands
  - eg.  $(a + b) * 2 - c$
  - Parentheses are used to denote the order
- **Postfix**: write the operator **after** the operands  $L R \text{ op}$ 
  - eg.  $ab + 2 * c$
  - Advantage: **no need for parentheses!**

# Examples

<b>Infix</b>	<b>Postfix</b>
$(a + b)$	$a b +$
$(x - y - z)$	$x y - z -$
$(x - y - z) / (u + v)$	$x y - z - u v + /$
$(a^2 + b^2) * (m - n)$	$a 2 ^ b 2 ^ + m n - *$

# Using a Stack to evaluate postfix expressions

- Scan from **left to right**
- When we find an **operand**  $X$ , **insert** it into the stack
- When we find an **operator**  $op$ 
  - **remove** the top operand into a variable  $R$  (right operand)
  - **remove** another topmost operand into a variable  $L$  (left operand)
  - Perform the operation  $L op R$
  - **Insert** the value back into the stack
- End of expression: its value is the (only) item remaining in the stack

# Translating Infix expressions to Postfix

- We can also use a Stack to translate **fully parenthesized** infix arithmetic expressions to postfix.
- **Algorithm** to convert  $(L \text{ op } R)$  to the postfix form  $L R \text{ op}$ 
  - ignore the left parenthesis
  - convert  $L$  to postfix
  - save  $\text{op}$  on the stack
  - convert  $R$  to postfix
  - then, on  $)$ , pop the stack and output the  $\text{op}$



# Example

- We want to translate the infix expression  $((5*(9+8))+7)$  into postfix.
- The result will be  $5\ 9\ 8\ +\ *\ 7\ +$

Input	Output	Stack
(		
(		
5	5	
*		*
(		*
9	9	*
+		* +
8	8	* +
)	+	*
)	*	
+		+
7	7	+
)	+	

# Queues

- Very limited functionality (similarly to stacks)
  - but useful in practice
  - allows for efficient implementations
- Insert in the **back**, remove from the **front**
  - First-in, first-out (FIFO)
- Access only the **front** and **back** elements
  - No random access
  - No iteration

# FIFO access

```
// Επιστρέφει το στοιχείο στο μπροστινό μέρος της ουράς  
Pointer queue_front(Queue queue);  
  
// Επιστρέφει το στοιχείο στο πίσω μέρος της ουράς  
Pointer queue_back(Queue queue);  
  
// Προσθέτει την τιμή value στο πίσω μέρος της ουράς queue.  
void queue_insert_back(Queue queue, Pointer value);  
  
// Αφαιρεί την τιμή στο μπροστά μέρος της ουράς  
void queue_remove_front(Queue queue);
```

- Example [4, 6, 2, 1]
  - Insert 3: [4, 6, 2, 1, 3]
  - Remove: [6, 2, 1, 3]
- Commonly called **push/pop** (or enqueue/dequeue)

# When to use Queues

- When FIFO access is enough
- Many applications
  - Scheduling of processes
  - Access to resources (CPU, printers, etc)
  - Breadth First Search in trees and graphs
  - ...

# Example, experimental simulation

- A new job arrives to the CPU each second with pb  $p$
- Each job takes between 1 and 4 seconds to execute (random)
- What is the **average** waiting time?
- We can write a program that **simulates** the system using a queue
  - time represented by an integer `t`
  - at each second, insert a job randomly (using `rand`)
  - select `duration` also randomly
  - remove job after `duration` seconds, compute its waiting time

# Readings

- T. A. Standish. Data Structures, Algorithms and Software Principles in C. Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C., Κεφ. 4.