# Δομές Δεδομένων

## Μέρος 4ο: Διανύσματα, Στοίβες, Ουρές

Εξάμηνο Σπουδών: 6ο

Κωδικός Μαθήματος: 681

Τμήμα Μαθηματικών
Πανεπιστήμιο Ιωαννίνων

Μιχάλης Α. Μπέκος
bekos@uoi.gr

# ΑΤΔ: Διάνυσμα

ADT: vector or arraylist

- Ο ΑΤΔ διάνυσμα επεκτείνει την έννοια του πίνακα αποθηκεύοντας μια ακολουθία αντικειμένων.
- Κάθε στοιχείο μπορεί να προσπελαστεί, εισαχθεί ή διαγραφεί καθορίζοντας ένα δείκτη (index) (σ.σ. αριθμός στοιχείων που προηγούνται).

| | |
|---|---|
| `empty()` | ελέγχει αν η δομή είναι κενή |
| `size()` | επιστρέφει το πλήθος των στοιχείων της δομής |
| `at(integer i)` | επιστρέφει το στοιχείο στη θέση i (χωρίς να το διαγράφει) |
| `set(integer i, element e)` | αντικαθιστά το στοιχείο στη θέση i με το δοθέν στοιχείο e |
| `insert(integer i, element e)` | ενθέτει το στοιχείο e στη θέση i |
| `erase(integer i)` | διαγράφει το στοιχείο στη θέση i |
| `push_back(element e)` | ενθέτει το στοιχείο e στο τέλος της δομής |

- Μια εξαίρεση δημιουργείται εάν δοθεί λανθασμένος δείκτης (π.χ. αρνητικός)

# Εφαρμογές

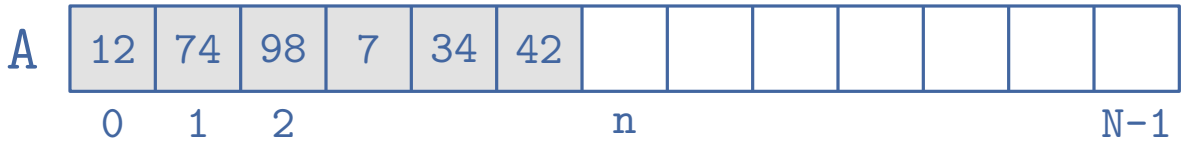
- Άμεσες:
  - Ταξινομημένη συλλογή αντικειμένων (απλή βάση δεδομένων)

- Έμμεσες:
  - Βοηθητική δομή δεδομένων για αλγόριθμους
  - Συστατικό στοιχείο άλλων δομών δεδομένων

# Υλοποίηση με Χρήση Πίνακα

Array-based implementation

- Χρησιμοποιήστε έναν πίνακα `A` χωρητικότητας `N`
- Μια μεταβλητή `n` καταγράφει το μέγεθος του διανύσματος (σ.σ. πλήθος αποθηκευμένων στοιχείων).

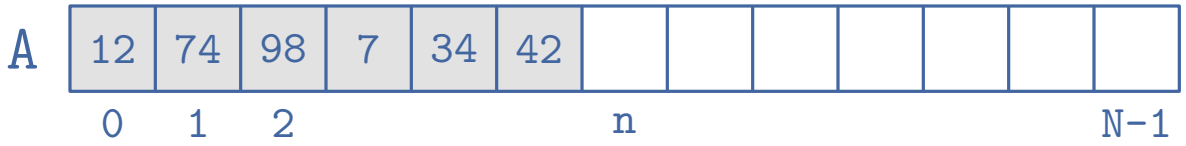



  - Η μέθοδος `at(i)` υλοποιείται σε $\mathcal{O}(1)$ χρόνο επιστρέφοντας το στοιχείο `A[i]`
  - Η μέθοδος `set(i,e)` υλοποιείται σε $\mathcal{O}(1)$ χρόνο θέτοντας `A[i]=e`

# Υλοποίηση με Χρήση Πίνακα

Array-based implementation

- Χρησιμοποιήστε έναν πίνακα `A` χωρητικότητας `N`
- Μια μεταβλητή `n` καταγράφει το μέγεθος του διανύσματος (σ.σ. πλήθος αποθηκευμένων στοιχείων).

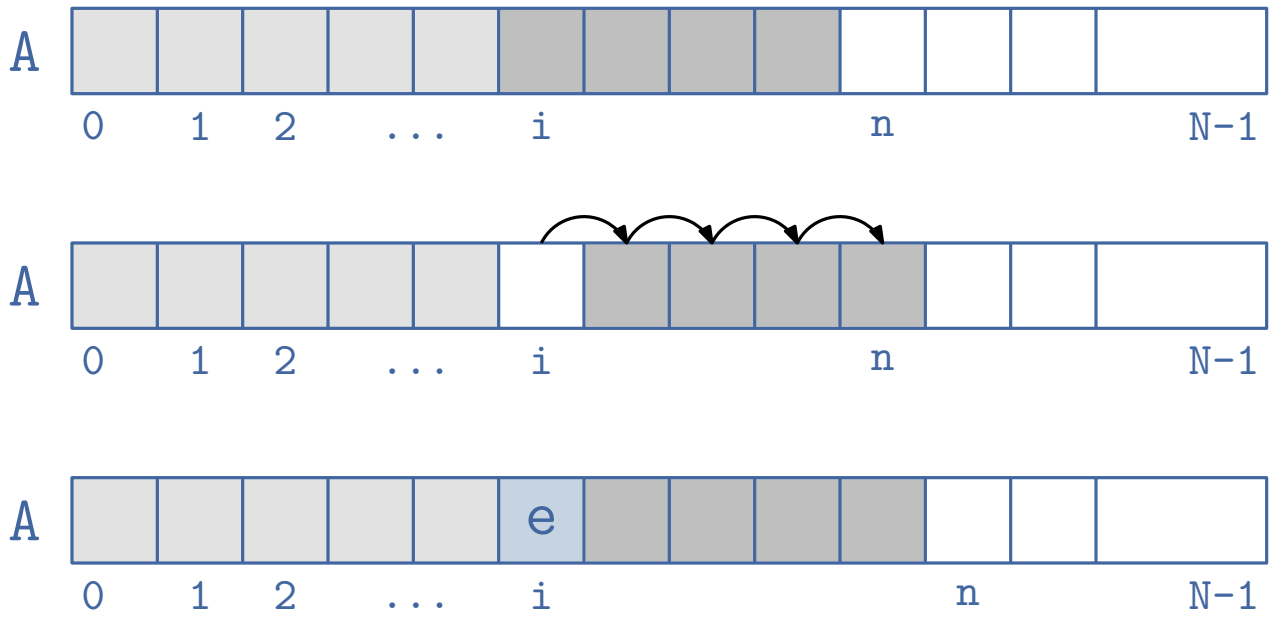



- Η μέθοδος `at(i)` υλοποιείται σε $\mathcal{O}(1)$ χρόνο επιστρέφοντας το στοιχείο `A[i]`
- Η μέθοδος `set(i,e)` υλοποιείται σε $\mathcal{O}(1)$ χρόνο θέτοντας `A[i]=e`

```
template <typename E>
class ArrayVector {
private:
    E  *A;
    int N, n;
public:
    E& at(int i) {
        return A[i];
    }
    void set(int i, const E &e) {
        A[i] = e;
    }
    ....
};
```

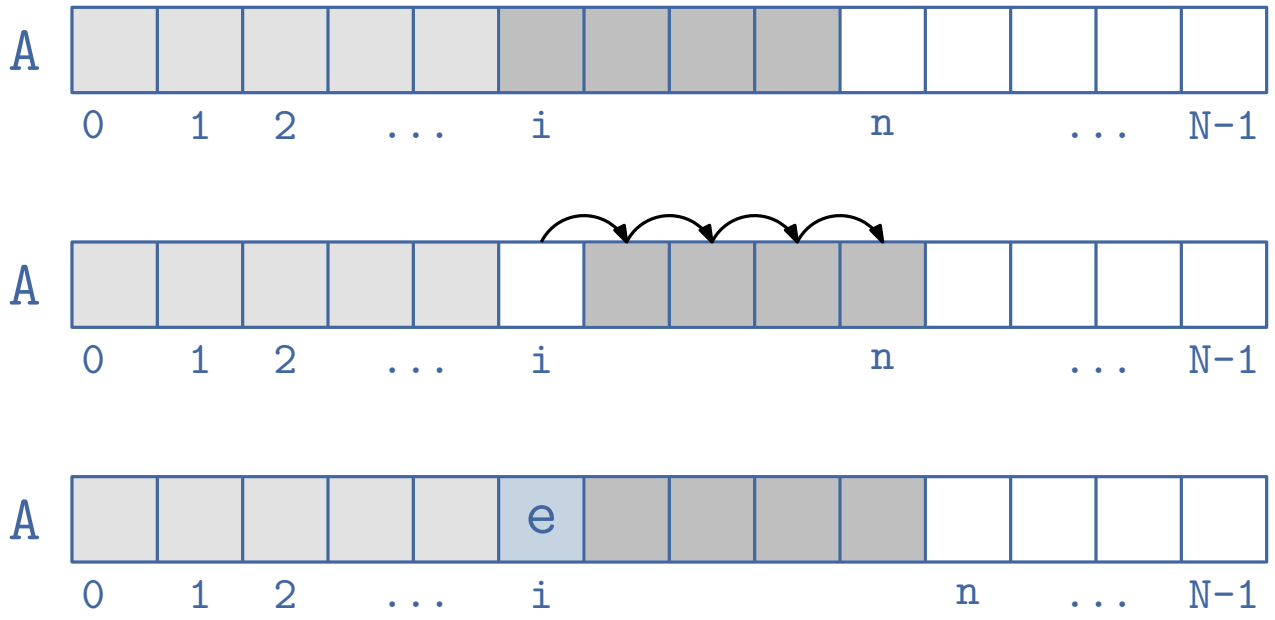
# Εισαγωγή: `insert(integer i, element e)`

- Για την υλοποίηση της μεθόδου `insert(i, e)`, πρέπει να “κάνουμε χώρο” για το νέο στοιχείο μετατοπίζοντας προς τα εμπρός τα `n - i` στοιχεία `A[i],..., A[n-1]`
- Στη χειρότερη περίπτωση `(i = 0)`, απαιτείται χρόνος $\mathcal{O}(n)$



# Εισαγωγή: `insert(integer i, element e)`

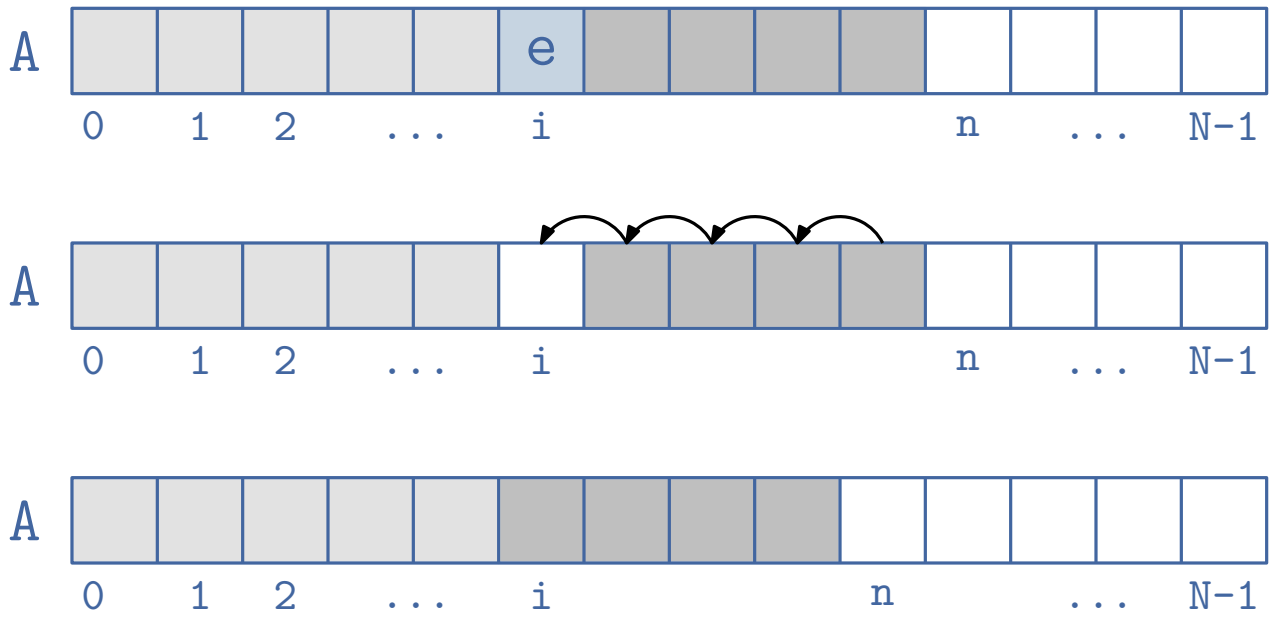
- Για την υλοποίηση της μεθόδου `insert(i, e)`, πρέπει να “κάνουμε χώρο” για το νέο στοιχείο μετατοπίζοντας προς τα εμπρός τα `n - i` στοιχεία `A[i],..., A[n-1]`
- Στη χειρότερη περίπτωση `(i = 0)`, απαιτείται χρόνος $\mathcal{O}(n)$




```
void insert(int i, const E &e) {
  // shift
  for (int j = n-1; j >= i; j--) {
    A[j+1] = A[j];
  }
  A[i] = e;
  n++;
}
```

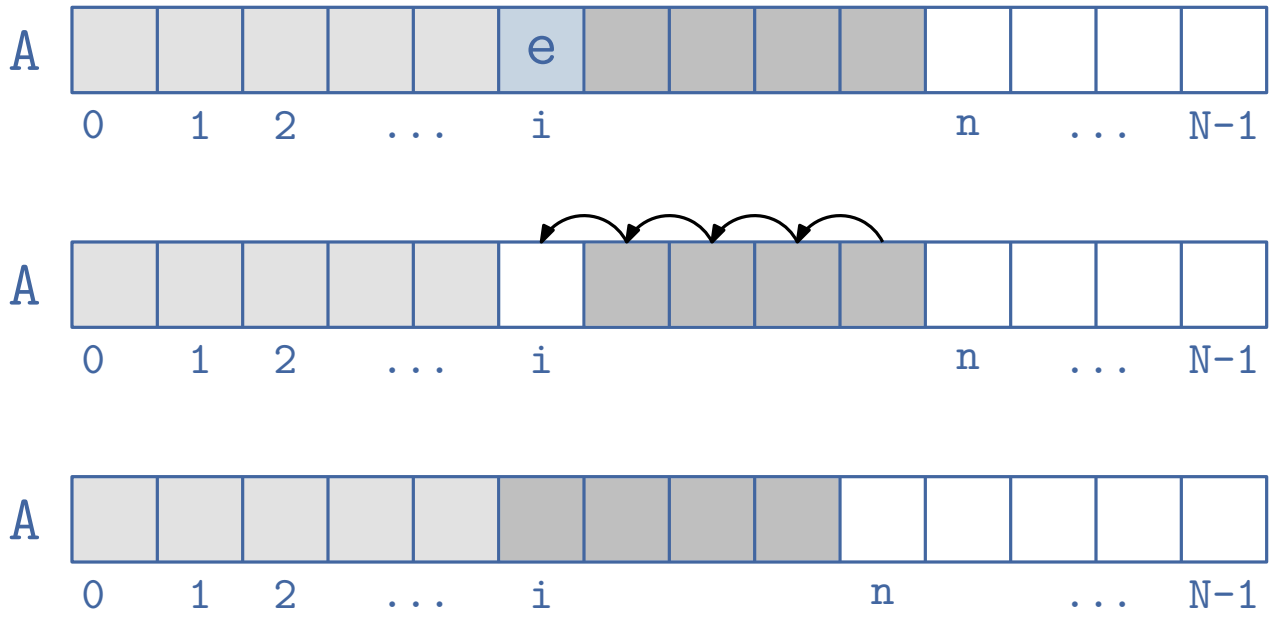
# Εισαγωγή: `erase(integer i)`

- Για την υλοποίηση της μεθόδου `erase(i)`, πρέπει να "γεμίσουμε το χώρο" που αφήνεται από το αφαιρεθέν στοιχείο μετατοπίζοντας προς τα πίσω τα `n-i-1` στοιχεία `A[i+1],...,A[n-1]`
- Στη χειρότερη περίπτωση `(i = 0)`, απαιτείται χρόνος $\mathcal{O}(n)$



# Εισαγωγή: `erase(integer i)`

- Για την υλοποίηση της μεθόδου `erase(i)`, πρέπει να "γεμίσουμε το χώρο" που αφήνεται από το αφαιρεθέν στοιχείο μετατοπίζοντας προς τα πίσω τα `n-i-1` στοιχεία `A[i+1], ..., A[n-1]`
- Στη χειρότερη περίπτωση `(i = 0)`, απαιτείται χρόνος $\mathcal{O}(n)$




```
void erase(int i) {
  // shift
  for (int j = i+1; j < n; j++) {
    data[j-1] = data[j];
  }
  n--;
}
```

# Σημειώσεις

- Στην υλοποίηση του ΑΤΔ διάνυσμα με χρήση πίνακα:
  - Ο χώρος που χρησιμοποιείται για την αποθήκευση $n$ στοιχείων είναι $\mathcal{O}(n)$
  - Οι μέθοδοι `size`, `empty`, `at` και `set` υλοποιούνται σε $\mathcal{O}(1)$ χρόνο
  - Οι μέθοδοι `insert`, και `erase` υλοποιούνται σε $\mathcal{O}(n)$ χρόνο
  - Αποφύγαμε τη συζήτηση εκφυλισμένων περιπτώσεων

    π.χ. τη διαγραφή ενός στοιχείου από μια κενή δομή

- Το μέγιστο μέγεθος του πίνακα πρέπει να οριστεί εκ των προτέρων και δεν μπορεί να αλλάξει
  - Ιδέα: Κατά την εισαγωγή ενός στοιχείου, όταν ο πίνακας είναι γεμάτος, αντί να δημιουργήσουμε μια εξαίρεση, μπορούμε να αντικαταστήσουμε τον πίνακα με έναν μεγαλύτερο $\rightarrow$ Υλοποίηση με πίνακα αυξανόμενου μεγέθους (ArrayList)

# Υλοποίηση με Πίνακα Αυξανόμενου Μεγέθους

Growable array-based implementation

- Ερώτημα: Στην υλοποίηση του ΑΤΔ διάνυσμα με πίνακα αυξανόμενου μεγέθους, ποιό θα πρέπει να είναι το μέγεθος του νέου πίνακα;
  - Στρατηγική σταδιακής αύξησης: Incremental strategy — Το μέγεθος του πίνακα αυξάνεται κατά μια σταθερά $c$
  - Στρατηγική διπλασιασμού: Doubling strategy — Το μέγεθος του πίνακα διπλασιάζεται

- Ποιά στρατηγική είναι πιο αποδοτική;
  - Συγκρίνουμε τις δύο στρατηγικές αναλύοντας τον συνολικό χρόνο $T(n)$ που απαιτείται για την εκτέλεση μιας ακολουθίας $n$ ενθέσεων `push_back()`.

    $\Rightarrow$ Επιμερισμένος χρόνος: $\frac{T[n]}{n}$
  - Υπόθεση: Αρχικά το (κενό) διάνυσμα υλοποιείται με τη χρήση ενός πίνακα μεγέθους 1.

# Υλοποίηση με Πίνακα Αυξανόμενου Μεγέθους

- Στρατηγική σταδιακής αύξησης: Ο πίνακας αντικαθίσταται $k = \frac{n}{c}$ φορές

$$\begin{aligned} T[n] &\leq n + c + 2c + 3c + \ldots + kc \\ &= n + c \cdot (1 + 2 + 3 + \ldots + k) \\ &= n + c \cdot \tfrac{k(k+1)}{2} \\ &= \mathcal{O}(n^2) \quad \Rightarrow \text{Επιμερισμένος χρόνος: } \mathcal{O}(n) \end{aligned}$$

- Στρατηγική διπλασιασμού: Ο πίνακας αντικαθίσταται $k \approx \log n$ φορές

$$\begin{aligned} T[n] &\leq n + 1 + 2 + 4 + \ldots + 2^k \\ &= n + 2^{k+1} - 1 \\ &= 3n - 1 \\ &= \mathcal{O}(n) \quad \Rightarrow \text{Επιμερισμένος χρόνος: } \mathcal{O}(1) \end{aligned}$$

# ΑΤΔ: Διάνυσμα

| | Array-based | Growable array-based |
|---|---|---|
| `empty()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `size()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `at(integer i)` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `set(integer i, element e)` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `insert(integer i, element e)` | $\mathcal{O}(n)$ | $\mathcal{O}(n)^*$ |
| `erase(integer i)` | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| `push_back(element e)` | $\mathcal{O}(1)$ | $\mathcal{O}(1)^*$ |
| | | |

*Επιμερισμένος χρόνος

# STD Vectors

```
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> v(10);   // initial capacity: 10
  for (unsigned i=0; i<v.size(); i++) {
    v.push_back(i);
  }

  // reverse vector using operator[]:
  for (unsigned i=0; i<v.size()/2; i++) {
    std::swap(v[i], v[v.size()-i-1]);
  }

  std::cout << "Vector contains:";
  for (unsigned i=0; i<v.size(); i++) {
    std::cout << v[i] << ' ';
  }
}
```

https://www.cplusplus.com/reference/vector/vector/

# ΑΤΔ: Στοίβα

ADT: stack

- Οι εισαγωγές και οι διαγραφές ακολουθούν το σχήμα “last-in first-out”

| | |
|---|---|
| `empty()` | ελέγχει αν η δομή είναι κενή |
| `size()` | επιστρέφει το πλήθος των στοιχείων της δομής |
| `push(element e)` | ενθέτει το στοιχείο e στη δομή |
| `pop()` | διαγράφει το τελευταίο εισαχθέν στοιχείο |
| `top()` | επιστρέφει το τελευταίο εισαχθέν στοιχείο (χωρίς να το διαγράφει) |

- Εφαρμογές:
  - Ιστορικό επισκέψεων ιστοσελίδων ενός web browser
  - Αναιρέσεις (undo) ενός προγράμματος επεξεργασίας κειμένου
  - Βοηθητική δομή δεδομένων για αλγόριθμους
  - Συστατικό στοιχείο άλλων δομών δεδομένων

# Υλοποίηση με Χρήση Διανύσματος

Vector-based implementation

- Ένας απλός τρόπος υλοποίησης του ΑΤΔ στοίβα είναι με τη χρήση ενός διανύσματος
- Η ένθεση/διαγραφή στοιχείων γίνεται στο τέλος της λίστας
  γιατί όχι στην αρχή του διανύσματος;
  - `push(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `insert(size())` του διάνυσματος
  - `pop()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `erase(size()-1)` του διάνυσματος
  - `top()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `at(size()-1)` του διάνυσματος

# Υλοποίηση με Χρήση Διανύσματος

Vector-based implementation

- Ένας απλός τρόπος υλοποίησης του ΑΤΔ στοίβα είναι με τη χρήση ενός διανύσματος
- Η ένθεση/διαγραφή στοιχείων γίνεται στο τέλος της λίστας

  γιατί όχι στην αρχή του διανύσματος;

  - `push(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `insert(size())` του διάνυσματος
  - `pop()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `erase(size()-1)` του διάνυσματος
  - `top()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `at(size()-1)` του διάνυσματος

```
template <typename E>
class ArrayStack : private ArrayVector<E> {
public:
  ....
  void push(const E &e) {
    ArrayVector<E>::insert(size(), e);
  }
  void pop(){
    if (empty()) throw StackEmpty();
    ArrayVector<E>::erase(size()-1);
  }
  const E& top() {
    if (empty()) throw StackEmpty();
    return ArrayVector<E>::at(size()-1);
  }
  ...
};
```

# Υλοποίηση με Χρήση Λίστας

List-based implementation

- Ένας ακόμη τρόπος υλοποίησης του ΑΤΔ στοίβα είναι με τη χρήση μίας λίστας
- Η ένθεση/διαγραφή στοιχείων γίνεται στην αρχή της λίστας

  - `push(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push_front(e)` της λίστας
  - `pop()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `pop_front()` της λίστας
  - `top()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `front()` της λίστας

# Υλοποίηση με Χρήση Λίστας

List-based implementation

- Ένας ακόμη τρόπος υλοποίησης του ΑΤΔ στοίβα είναι με τη χρήση μίας λίστας
- Η ένθεση/διαγραφή στοιχείων γίνεται στην αρχή της λίστας
  - `push(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push_front(e)` της λίστας
  - `pop()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `pop_front()` της λίστας
  - `top()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `front()` της λίστας

```
template <typename E>
class LinkedStack : private DLinkedList<E>{
public:
  ....
  void push(const E &e) {
    DLinkedList<E>::push_front(e);
  }
  void pop(){
    if (empty()) throw StackEmpty();
    DLinkedList<E>::pop_front();
  }
  const E& top() {
    if (empty()) throw StackEmpty();
    return DLinkedList<E>::front();
  }
  ...
};
```

# ΑΤΔ: Στοίβα

- Στις υλοποιήσεις του ΑΤΔ στοίβα που εξετάσαμε:
  - Ο χώρος που χρησιμοποιείται για την αποθήκευση $n$ στοιχείων είναι $\mathcal{O}(n)$
  - Κάθε μέθοδος υλοποιείται σε $\mathcal{O}(1)$ (επιμερισμένο) χρόνο

| | Vector-based | List-based |
|---|---|---|
| `empty()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `size()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `push(element e)` | $\mathcal{O}(1)^*$ | $\mathcal{O}(1)$ |
| `pop()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `top()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| | | |

*Επιμερισμένος χρόνος

# STD Stacks

```
#include <iostream>
#include <stack>

int main ()
{
  std::stack<int> s;

  for (int i=0; i<5; ++i) s.push(i);

  std::cout << "Popping out elements...";
  while (!s.empty())
  {
     std::cout << ' ' << s.top();
     s.pop();
  }
  std::cout << '\n';

  return 0;
}
```

---

https://cplusplus.com/reference/stack/stack/

# Εφαρμογές: Ταίριασμα Παρανθέσεων

- Είσοδος: Ένα πίνακας $A$ μεγέθους $n$, κάθε στοιχείο του οποίου είναι ένας από τους ακόλουθους χαρακτήρες: “(´´, “{´´, “[´´, “]´´, “}´´, “)´´
- Έξοδος: `true`, αν η παρενθετικοποίηση είναι ορθή. Διαφορετικά, `false`.

- Παραδείγματα:
  - Ορθές παρενθετικοποιήσεις:

    ( )(( )){[ ]}(({{ }}))

    ((( )))[[ ]]{(([ ]))}[{{ }}]
  - Λανθασμένες παρενθετικοποιήσεις:

    )(( )){[ ]}(({{ }}))

    ({[ ])}

- Αλγόριθμος:

```
1  S ← empty stack;
2  for i = 0 to n − 1 do
3  |  if A[i] is an opening symbol then
4  |  |  S.push(A[i]);
5  |  else
6  |  |  if S.empty() or S.top() does not match A[i] then
7  |  |  |  return false;
8  |  |  else
9  |  |  |  S.pop();
10 return S.empty();
```

# Εφαρμογές: HTML Validation

- Είσοδος: Ένα έγγραφο σε HTML format
- Έξοδος: `true`, αν οι ετικέτες HTML ταιριάζουν. Διαφορετικά, `false`.

- Παράδειγμα:

```
<html>
<head>
<title>This is the title of the page</title>
</head>
<body>
<center>
  <h1>This is a header</h1>
</center>
<p>This is a <b>bold text</b> in a paragraph.</p>
<ol>
  <li>Case 1</li>
  <li>Case 2</li>
</ol>
</body>
</html>
```

- Ερώτημα: Πώς θα ελέγχατε αν ένα HTML έγγραφο είναι δομημένο σωστά;

# ΑΤΔ: Ουρά

ADT: queue

- Οι εισαγωγές και οι διαγραφές ακολουθούν το σχήμα “first-in first-out”

| | |
|---|---|
| `empty()` | ελέγχει αν η δομή είναι κενή |
| `size()` | επιστρέφει το πλήθος των στοιχείων της δομής |
| `enqueue(element e)` | ενθέτει το στοιχείο e στο τέλος της δομής |
| `dequeue()` | διαγράφει το πρώτο στοιχείο στη δομή |
| `front()` | επιστρέφει το πρώτο στοιχείο στη δομή (χωρίς να το διαγράφει) |

- Εφαρμογές:
  - Πρόσβαση σε κοινόχρηστους πόρους (π.χ. εκτυπωτής)
  - Πολυπρογραμματισμός
  - Βοηθητική δομή δεδομένων για αλγόριθμους
  - Συστατικό στοιχείο άλλων δομών δεδομένων

# Υλοποίηση με Χρήση Λίστας

List-based implementation

- Ένας τρόπος υλοποίησης του ΑΤΔ ουρά είναι με τη χρήση μίας λίστας
- Η ένθεση (διαγραφή) στοιχείων γίνεται στο τέλος (στην αρχή) της λίστας

  - `enqueue(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push_back(e)` της λίστας
  - `dequeue()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `pop_front` της λίστας
  - `front()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `front()` της λίστας

# Υλοποίηση με Χρήση Λίστας

List-based implementation

- Ένας τρόπος υλοποίησης του ΑΤΔ ουρά είναι με τη χρήση μίας λίστας
- Η ένθεση (διαγραφή) στοιχείων γίνεται στο τέλος (στην αρχή) της λίστας
  - `enqueue(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push_back(e)` της λίστας
  - `dequeue()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `pop_front` της λίστας
  - `front()`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `front()` της λίστας

```
template <typename E>
class LinkedQueue : private DLinkedList<E>{
public:
  ...
  void enqueue(const E &e) {
    DLinkedList<E>::push_back(e);
  }
  void dequeue(){
    if (empty()) throw QueueEmpty();
    DLinkedList<E>::pop_front();
  }
  const E& front() {
    if (empty()) throw QueueEmpty();
    return DLinkedList<E>::front();
  }
  ...
};
```

# Υλοποίηση με Χρήση Δύο Στοιβών

- Ένας ακόμη τρόπος υλοποίησης του ΑΤΔ ουρά είναι με τη χρήση δύο στοιβών
- Η ένθεση (διαγραφή) στοιχείων γίνεται στην πρώτη (στην δεύτερη) στοίβα

  - `enqueue(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push(e)` της πρώτης στοίβας
  - `dequeue()`: αν η δεύτερη στοίβα είναι κενή, τα στοιχεία της πρώτης αντιγράφονται στη δεύτερη. Ακολούθως, καλείται η μέθοδος `pop` της δεύτερης στοίβας
  - `front()`: αν η δεύτερη στοίβα είναι κενή, τα στοιχεία της πρώτης αντιγράφονται στη δεύτερη. Ακολούθως, καλείται η μέθοδος `top` της δεύτερης στοίβας

```
template <typename E>
class StackedQueue {
private:
  Stack<E> *s1, *s2;
public:
  ...
  StackedQueue() {
    s1 = new LinkedStack<E>;
    s2 = new LinkedStack<E>;
  }
  int size() const {
    return s1->size() + s2->size();
  }
  bool empty() const {
    return size()==0;
  }
  ...
};
```

# Υλοποίηση με Χρήση Δύο Στοιβών

- Ένας ακόμη τρόπος υλοποίησης του ΑΤΔ ουρά είναι με τη χρήση δύο στοιβών
- Η ένθεση (διαγραφή) στοιχείων γίνεται στην πρώτη (στην δεύτερη) στοίβα

  - `enqueue(e)`: υλοποιείται σε $\mathcal{O}(1)$ χρόνο καλώντας τη μέθοδο `push(e)` της πρώτης στοίβας
  - `dequeue()`: αν η δεύτερη στοίβα είναι κενή, τα στοιχεία της πρώτης αντιγράφονται στη δεύτερη. Ακολούθως, καλείται η μέθοδος `pop` της δεύτερης στοίβας
  - `front()`: αν η δεύτερη στοίβα είναι κενή, τα στοιχεία της πρώτης αντιγράφονται στη δεύτερη. Ακολούθως, καλείται η μέθοδος `top` της δεύτερης στοίβας

```
template <typename E>
class StackedQueue {
public:
  ...
  void enqueue(const E &e) {
    s1->push(e);
  }
  void dequeue() {
    if (empty()) throw QueueEmpty();
    if (s2->empty()) {
      while (!s1->empty()) {
        s2->push(s1->top()); s1->pop();
      }
    }
    s2->pop();
  }
  ...
};
```

# Ανάλυση με τη Μέθοδο του Τραπεζίτη

- Οι μέθοδοι `enqueue(e)`, `dequeue()` και `front()` υλοποιούνται σε $\mathcal{O}(1)$ επιμερισμένο χρόνο.
  υποθέτοντας ότι κάθε πράξη του ΑΤΔ στοίβα υλοποιείται σε $\mathcal{O}(1)$ επιμερισμένο χρόνο.

  Απόδειξη:

  "Χρεώνουμε" τρείς μονάδες για κάθε `enqueue(e)` πράξη.

  - Μία μονάδα καλύπτει το κόστος της `enqueue(e)` πράξης $\Rightarrow$
  - Κάθε στοιχείο το οποίο βρίσκεται στην πρώτη στοίβα συσχετίζεται με δύο διαθέσιμες μονάδες.
  - Το κόστος μετακίνησης ενός στοιχείου από την πρώτη στοίβα στην δεύτερη καλύπτεται από τη μία από τις δύο διαθέσιμες μονάδες του $\Rightarrow$
  - Κάθε στοιχείο το οποίο βρίσκεται στη δεύτερη στοίβα συσχετίζεται με μία διαθέσιμη μονάδα, η οποία καλύπτει το κόστος της διαγραφής του από τη δομή.

# ΑΤΔ: Ουρά

- Στις υλοποιήσεις του ΑΤΔ ουρά που εξετάσαμε:
  - Ο χώρος που χρησιμοποιείται για την αποθήκευση $n$ στοιχείων είναι $\mathcal{O}(n)$
  - Κάθε μέθοδος υλοποιείται σε $\mathcal{O}(1)$ (επιμερισμένο) χρόνο

| | List-based | Stack-based |
|---|---|---|
| `empty()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `size()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `enqueue(element e)` | $\mathcal{O}(1)$ | $\mathcal{O}(1)^*$ |
| `dequeue()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)^*$ |
| `front()` | $\mathcal{O}(1)$ | $\mathcal{O}(1)^*$ |
| | | |

*Επιμερισμένος χρόνος

# STD Queues

```
#include <iostream>
#include <queue>

int main ()
{
  std::queue<int> q;

  for (int i=0; i<5; ++i) q.push(i);

  std::cout << "Queue contains: ";
  while (!q.empty())
  {
    std::cout << ' ' << q.front();
    q.pop();
  }
  std::cout << '\n';

  return 0;
}
```

---

https://cplusplus.com/reference/queue/queue/

# Επιπλέον υλικό

- Ενότητα 2.1:
  Michael T. Goodrich, Roberto Tamassia, Αλγόριθμοι σχεδίαση και εφαρμογές
  ISBN: 9789605126971, εκδόσεις Γκιούρδα, 2016.