How can we implement ADTVector?

- A Vector can be seen as an abstract resizable “array”
- So it makes sense to implement it using a real array
  - store Vector’s elements in the array
  - `vector_get_at, vector_set_at` are trivial
- But what about `vector_insert_last`?
  - Arrays in C have fixed size

Dynamic arrays

- Main idea: resize the array
  - such arrays are called “dynamic” or “growable”
- Problem: we need to copy the previous values
- A possible algorithm for `vector_insert_last`
  - Allocate memory for `size+1` elements
  - Copy the `size` previous elements
  - Set the new element as last
  - Increase `size`
- What is the complexity of this?
  - $O(n)$, because of the copy!
  - Can we do better?
Improving the complexity of insert

- **Idea**: allocate *more memory* than we need!
  - e.g., allocate memory for 100 "empty" elements
    - **capacity**: total allocated memory
    - **size**: number of inserted elements
  - Insert is $O(1)$ if we have free space (just copy the new value)

- Does this change the complexity?
  - in the worst-case?
  - in the average-case?

- **No**, for some values of $n$ the operation is still slow!
  - For any values, "average-case" makes no difference

---

Amortized-time complexity

- We see here the value of *amortized-time* complexity
  - A single execution can be slow
  - But "most" are fast
  - In many application we only care about the average wrt all executions

- Assume we reserve 100 more elements each time
  - How many steps each insert takes on average?

- Intuitively, $\frac{n}{100}$. So still $O(n)$, same complexity!
  - Same for any constant number of empty elements $k$
  - Remember, complexity cares about large $n$! Think $n \gg k$
  - Can we do better?

---

Improving the complexity of insert

- **Idea**: allocate *more memory* than we need!
  - e.g., allocate memory for 100 "empty" elements
    - **capacity**: total allocated memory
    - **size**: number of inserted elements
  - Insert is $O(1)$ if we have free space (just copy the new value)

- Does this change the complexity?
  - in the worst-case?
  - in the average-case?

- **No**, for some values of $n$ the operation is still slow!
  - For any values, "average-case" makes no difference

---

Amortized-time complexity

- We see here the value of *amortized-time* complexity
  - A single execution can be slow
  - But "most" are fast
  - In many application we only care about the average wrt all executions

- Assume we reserve 100 more elements each time
  - How many steps each insert takes on average?

- Intuitively, $\frac{n}{100}$. So still $O(n)$, same complexity!
  - Same for any constant number of empty elements $k$
  - Remember, complexity cares about large $n$! Think $n \gg k$
  - Can we do better?
How to improve the complexity

- **Idea**: the number of empty elements must depend on \( n \)
  - Use more empty elements as the Vector grows!
- Standard approach: reserve \( a \cdot n \) extra elements
  - for some constant \( a > 1 \), called the **growth factor**
- Common values
  - \( a = 2 \)
  - \( a = 1.5 \)
- In this class we will use \( a = 2 \)
  - we always **double** the capacity

From linear to constant time

- We always **double** the capacity
  - What is the amortized-time complexity of insert?
- We do \( n \) insertions starting from an empty Vector
  - Assume the last one was "slow" (the most "unlucky" case)
- How many **steps** did we perform in total?
  - \( n \) steps just for placing each element
  - \( n \) steps for the **last resize**
  - How many for **all the previous resizes together**?
    \[
    \frac{n}{2} + \frac{n}{4} + \ldots + 1 = n - 1
    \]
- So less than \( 3n \) in total!
  - On average: \( \frac{3n}{n} = O(1) \)

A property to remember

- Consider the **geometric progression** with ratio 2
  \[1, 2^1, 2^2, \ldots, 2^n\]
- Summing \( n \) terms, we get the **next one minus 1**
  \[1 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1\]
- So each term is **larger** than **all the previous** together!
  - This is important since several quantities **double** in data structures
- **Key point**: previous inserts are insignificant compared to the last one
Removing elements
- What about `vector_remove_last`?
- Simplest strategy: just consider the removed space as “empty”
  - `vector_remove_last` is clearly worst-case $O(1)$
  - Insert is not affected (we never reduce the amount of free space)
- Commonly used in practice
  - eg. `std::vector` in C++
- **Problem**: wasted space

Recovering wasted space
- **Idea**: if **half** of the array becomes empty, resize
  - the opposite of the doubling growing strategy
  - Is this ok?
- Careful
  - this is ok if we only remove
  - but a combination of remove+insert might become slow!
- Think of the following scenario
  - Insert $n$ elements with $n = 2^k$
  - The vector is now full
  - Perform a series of: insert, remove, insert, remove, ...

Better strategy
- when only $\frac{1}{4}$ of the array is full
- resize to $\frac{1}{4}$ of the capacity!
- So we still have “room” to both insert and remove
- We can show that even a combination of insert+remove is $O(1)$ amortized-time
Implementation

**Types**

```c
// Ένα VectorNode είναι pointer σε αυτό το struct.
struct vector_node {
    Pointer value; // Η τιμή του κόμβου.
};
// Ένα Vector είναι pointer σε αυτό το struct
struct vector {
    VectorNode array; // Τα δεδομένα, πίνακας από struct ve
    int size; // Πόσα στοιχεία έχουμε προσθέσει
    int capacity; // Πόσο χώρο έχουμε δεσμεύσει
    DestroyFunc destroy_value; // Συνάρτηση που καταστρέφει ένα στοι
};
```

**Implementation**

```c
Vector vector_create(int size, DestroyFunc destroy_value) {
    // Αρχικά το vector περιέχει size μη-αρχικοποιημένα στοιχεία, αλλ
    // δεσμεύουμε χώρο για τουλάχιστον VECTOR_MIN_CAPACITY για να απ
    // πολλαπλά resizes
    int capacity = size < VECTOR_MIN_CAPACITY ? VECTOR_MIN_CAPACITY :
    // Δέσμευση μνήμης, για το struct και το array.
    Vector vec = malloc(sizeof(*vec));
    VectorNode array = calloc(capacity, sizeof(*array)); // αρχικοπο
    vec->size = size;
    vec->capacity = capacity;
    vec->array = array;
    vec->destroy_value = destroy_value;
    return vec;
}
```

**Implementation**

```c
Random access is simple, since we have a real array.
Pointer vector_get_at(Vector vec, int pos) {
    return vec->array[pos].value;
}
```

**Implementation**

```c
void vector_set_at(Vector vec, int pos, Pointer value) {
    // Αν υπάρχει συνάρτηση destroy_value, την καλούμε για
    // το στοιχείο που αντικαθίσταται
    if (value != vec->array[pos].value && vec->destroy_value != NULL)
        vec->destroy_value(vec->array[pos].value);
    vec->array[pos].value = value;
}
```

**Implementation**

```c
Insert, we just need to deal with resizes.
void vector_insert_last(Vector vec, Pointer value) {
    // Μεγαλώνουμε τον πίνακα και προσθέτουμε το στοιχείο
    vec->array[vec->size].value = value;
    vec->size++;
}
```
Takeaways

- **Dynamic arrays** are the standard way to implement ADTVector
- Insert is $O(1)$
  - but amortized-time!
  - would you use a dynamic array in the software controlling an Airbus?
- Remove is also $O(1)$
  - also amortized, if we care about recovering wasted space
- Random access (get/set) is always worst-case $O(1)$