Data Structures

Dynamic Array in C

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Dynamic Array

as ADT

An array with the following operations:

- Add a new element at the end of the array, $ADD(V, x)$.
- Insert a new element before position i of the array, $ADD\_AT(V, i, x)$.
- Remove the element at position i of the array, $REMOVE\_AT(V, i, x)$.
- Return the i-th element of the array, $GET(V, i)$.
- Change the value of the i-th element of the array, $PUT(V, i, x)$.
- Check if the array is empty, $ISEMPTY(V)$.
- Return the number of elements that are stored in the array, $SIZE(V)$.
- ..

Dynamic Array

A dynamic array allocates memory dynamically in order to be able to add or remove elements and at the same time have random access to our stored elements.

Many programming languages contain such a data structure:

- in C++ it is called `std::vector`
- Java has two implementations, class `ArrayList` and class `Vector` which is also thread-safe

Dynamic Array

Some requirements:

- Separate interface from implementation
- Users are not allowed to change the internal representation (information-hiding)
- Users can create multiple dynamic arrays at the same time

Dynamic Array

interface in C (vector.h)

```
#ifndef _VECTOR_H
#define _VECTOR_H

typedef struct _vector* vector;
typedef int value_type;

vector vector_create();
void vector_destroy(vector);

value_type vector_get(vector, int);
void vector_put(vector, int, value_type);

void vector_add(vector, value_type);

void vector_add_at(vector, int, value_type);
value_type vector_remove_at(vector, int);

int vector_is_empty(vector);
int vector_size(vector);
void vector_clear(vector);

#endif
```

Dynamic Array Implementation

Representation (vector.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "vector.h"

#define INITIAL_CAPACITY 64
#define min(x,y) (((x)<(y))?(x):(y))

struct _vector {
    value_type* array;
    int size;
    int capacity;
};
```

Dynamic Array Implementation

Creation (vector.c)

```
vector vector_create() {
    vector v = (vector) malloc(sizeof(struct _vector));
    if (v == NULL) {
        fprintf(stderr, "Not enough memory!");
        abort();
    }
    v->size = 0;
    v->capacity = INITIAL_CAPACITY;
    v->array = (value_type*) malloc( \
        sizeof(value_type)* v->capacity);
    if (v->array == NULL) {
        fprintf(stderr, "Not enough memory!");
        abort();
    }
    return v;
}
```

Dynamic Array Implementation

Destruction (vector.c)

```
void vector_destroy(vector v) {
    assert(v);

    free(v->array);
    free(v);
}
```

Dynamic Array Implementation

Double Capacity (vector.c)

```
static
void vector_double_capacity(vector v) {
    assert(v);
    int new_capacity = 2 * v->capacity;
    value_type* new_array = (value_type*) malloc( \
        sizeof(value_type)*new_capacity);
    if (new_array == NULL) {
        fprintf(stderr, "Not enough memory!");
        abort();
    }

    for(int i = 0; i < v->size; i++) {
        new_array[i] = v->array[i];
    }

    free(v->array);
    v->array = new_array;
    v->capacity = new_capacity;
}
```

Dynamic Array Implementation

Half Capacity (vector.c)

```
static
void vector_half_capacity(vector v) {
    assert(v);
    if (v->capacity <= INITIAL_CAPACITY) {
        return;
    }

    int new_capacity = v->capacity / 2;
    value_type* new_array = (value_type*) malloc( \
        sizeof(value_type)*new_capacity);
    if (new_array == NULL) {
        fprintf(stderr, "Not enough memory!");
        abort();
    }

    for(int i = 0; i < min(v->size, new_capacity); i++) {
        new_array[i] = v->array[i];
    }

    free(v->array);
    v->array = new_array;
    v->capacity = new_capacity;
    v->size = min(v->size, new_capacity);
}
```

Dynamic Array Implementation

Add element at the end (vector.c)

```
void vector_add(vector v, value_type value) {
    assert(v);

    if (v->size >= v->capacity) {
        vector_double_capacity(v);
    }
    v->array[v->size++] = value;
}
```

Dynamic Array Implementation

Read and Write (vector.c)

```
value_type vector_get(vector v, int i) {
    assert(v);
    if (i < 0 || i >= v->size) {
        fprintf(stderr, "Out of index!");
        abort();
    }
    return v->array[i];
}

void vector_put(vector v, int i, value_type value) {
    assert(v);
    if (i < 0 || i >= v->size) {
        fprintf(stderr, "Out of index!");
        abort();
    }
    v->array[i] = value;
}
```

Dynamic Array Implementation

Add at the middle (vector.c)

```
void vector_add_at(vector v, int i, value_type value) {
    assert(v);

    if (i < 0 || i >= v->size) {
        fprintf(stderr, "Out of index!");
        abort();
    }

    if (v->size >= v->capacity) {
        vector_double_capacity(v);
    }

    for(int j = v->size; j>i; j--) {
        v->array[j] = v->array[j-1];
    }
    v->array[i] = value;
    v->size++;
}
```

Dynamic Array Implementation

Remove (vector.c)

```
value_type vector_remove_at(vector v, int i) {
    assert(v);

    if (i < 0 || i >= v->size) {
        fprintf(stderr, "Out of index!");
        abort();
    }

    value_type ret = v->array[i];
    for(int j = i+1; j < v->size; j++) {
        v->array[j-1] = v->array[j];
    }
    v->size--;

    if (4 * v->size < v->capacity) {
        vector_half_capacity(v);
    }

    return ret;
}
```

Dynamic Array Implementation

Is-Empty and Size (vector.c)

```
int vector_is_empty(vector v) {
    assert(v);
    return v->size == 0;
}

int vector_size(vector v) {
    assert(v);
    return v->size;
}
```

Dynamic Array Implementation

Clear (vector.c)

```
void vector_clear(vector v) {
    assert(v);

    v->size = 0;
    while (v->capacity > INITIAL_CAPACITY) {
        vector_half_capacity(v);
    }
}
```

Testing the Dynamic Array

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "vector.h"

int main() {
    vector v;
    v = vector_create();

    for(int i = 0; i < 1000000; i++) {
        vector_add(v, i);
        assert(vector_size(v) == i+1);
        assert(vector_get(v, i) == i);
    }

    assert(vector_size(v) == 1000000);

    while(!vector_is_empty(v)) {
        vector_remove_at(v, vector_size(v)-1);
    }

    assert(vector_is_empty(v));
    assert(vector_size(v) == 0);

    vector_destroy(v);

    return 0;
}
```