Arrays (Dinâmicos) em C

Rafael Guimarães 31 de março de 2020 Estruturas de Dados
#ArrayList, #DataStructures, #EstruturasdeDados, #vector

Arrays (Dinâmicos) em C

Um **array ou vetor**, é a estrutura mais primitiva para agrupar diversos elementos, possui **tamanho fixo** e ocupa um **espaço contíguo** na memória que nos permite acessar qualquer de seus elementos a partir de um ponteiro para o seu primeiro elemento.

Uma das principais vantagens do array é a possibilidade de acessar valores randomicamente em **O(1)** através do operador de indexação.

O **Array Dinâmico** é uma estrutura de dados que internamente possui um **espaço contíguo** na memória, no entanto, quando a capacidade chega no limite máximo, o vetor é realocado para um tamanho maior que o anterior em tempo de execução.

Nesse artigo vamos implementar essa estrutura do absoluto zero, destacando os pontos mais importantes, usando a linguagem de programação C.

1. Vamos definindo o tipo a estrutura ***Array*** que consiste do seguinte modo.

```
typedef struct array {
    int size;
    int max;
    int *v;
} Array;
```

Em seguida, de acordo com checklist de funcionalidades abaixo, vamos implementar os seguintes métodos

- **inicializar()** -> retorna um ponteiro do tipo ***Array***.
- **inserir()** -> deve inserir um novo elemento no final do Array
- **erase()** -> apaga um valor contido numa posição **i** passada como parâmetro
- **size()** -> retorna a quantidade de elementos do vetor
- **get()** -> acessa e retorna um elemento do vetor
- **set()** -> altera o valor de uma posição do array
- **realocar()** -> altera a capacidade máxima do array
- **Iterar()** -> percorre todo o array, pode ser usado por exemplo para imprimir.

Criando um Array

O método **init** é responsável por alocar o espaço na memória que guardará o endereço do **Array** e inicializar suas variáveis (atributos) internos do array. Como em tempo de execução nem tudo funciona perfeitamente, é importante o método retorne um valor **Boolean**. Esse tipo de retorno será usado em todos os métodos do Array Dinâmico.

Você vai perceber também que sempre no início dos métodos e após o uso do **malloc** ou **realloc** existe uma verificação:

```
if ( !arr ) return 0; //  arr == NULL ?
```

Essa verificação é importante para evitar erros do tipo **segmentation fault** tornando seu algoritmo mais robusto.

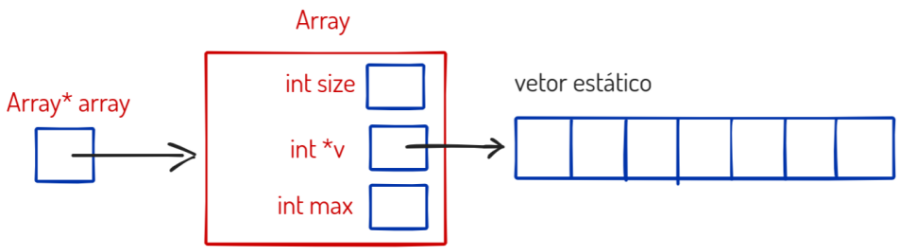
Agora vamos para o método **init**:

```c
#include <stdio.h>
#include <stdlib.h>
#include "array.h"

#define MAX 100

int init(Array** array) {
    Array* arr = (Array*) malloc(sizeof(Array));
    if ( !arr ) return 0;
    arr->size = 0;
    arr->max = MAX;
    arr->v = (int*) malloc(arr->max*sizeof(int));
    if (! (arr->v) ) return 0;
    *array = arr; // update the reference
    return 1;
}
```

A melhor forma de descrever esse trecho de código é usando um esquemático. Cada retângulo representa um espaço na memória, apesar da struct possuir um tamanho representativo na imagem, não ocupa tanto espaço na memória, porque ela armazena apenas duas variáveis do tipo **int** e o endereço da região da memória em que o vetor inicia.



Inserindo elementos.

Para inserir elementos, devemos verificar primeiro, se há espaço no array, comparando o tamanho atual em **size** com o limite máximo do vetor em **max.**

Caso a condição seja satisfeita, o vetor será realocado e posteriormente a operação de inserção no final.

Obs. Sem realocação de memória com a função **resize**() essa operação é feita em O(1).

```
int pushback(Array* arr, int value) {
    if (!arr) return 0;
    if (arr->size == arr->max) {
        if ( resize(arr) ) {
            arr->v[arr->size] = value;
            arr->size++;
            return 1;
        }
        return 0; // não foi possível realocar.
    }
    arr->v[arr->size++] = value;
    return 1;
}
```

Redimensionando o Array

Ocasionalmente será necessário redimensionar o array conforme a quantidade de elementos cresce. Essa operação é implementada na função **resize** mostrada abaixo.

```
int resize(Array* arr) {
    if (!arr) return 0;
    arr->max *= 2;
    arr->v = (int*) realloc(arr->v, arr->max*sizeof(int));
    if ( !arr->v ) return 0;
    return 1;
}
```

Para mudar o tamanho interno do nosso array, basta atualizar a variável **max** e depois utilizar a função **realloc** como demostrado.

É importante enfatizar que a função **realloc** na verdade, aloca um novo espaço de memória e em seguida copia todos os valores do antigo vetor para o novo, portanto em notação de complexidade essa operação é realizada em O(n).

Removendo Elementos.

Existem dois cenários aqui, se você precisa remover no final do array essa operação possui tempo de execução **constante** . Se você precisa remover no meio ou no inicio do array, seu tempo de execução será **linear**, tendo em vista a necessidade de reorganizar todo array a partir da posição removida, até o final.

```c
int erase(Array* arr, int index) {
    if ( !arr ) return 0;
    if (arr->size == 0) return 0;
    if ( index < 0 || index >= arr->size) return 0;
    arr->size--;
    for (int i = index; i < arr->size; i++) {
        arr->v[i] = arr->v[i+1];
    }
    return 1;
}
```

Atualizando as posições do Array Dinâmico após remoção de um valor

Tamanho do array.

Para consultar o tamanho do array, basta retornar o valor contido em size dentro da struct array;

```c
int size(Array* arr) {
    if (!arr) return 0;
    return arr->size;
}
```

Método get.

Utilizamos o método get para retornar um valor do array tendo como parâmetro a posição do array que o usuário deseja acessar, (desde que essa posição exista no array), essa implementação retorna **False** caso o valor não seja encontrado.

```c
int get(Array* arr, int *value, int index) {
    if (!arr) return 0;
    if (index >= 0 && index < arr->size) {
        *value = arr->v[index];
        return 1;
    }
    value = NULL;
    return 0;
}
```

Método set.

O objetivo do método **set** é setar uma posição do array com um determinado valor.

```
int set(Array* arr, int value, int index ) {
    if (!arr) return 0;
    if ( index < 0 || index >= arr->size ) return 0;
    arr->v[index] = value;
    return 1;
}
```

Percorrendo o array.

Por ultimo, não menos importante, para percorrer o array e visualizar se as operações anteriores estão certinhas você pode se basear por esse método.

```
int show(Array* arr) {
    if (!arr) return 0;
    printf("[ ");
    for (int i = 0; i < arr->size; i++) {
        int v;
        get(arr, &v, i);
        printf("%d ", v);
    }
    printf("]\n");
    return 1;
}
```

O código completo está disponível neste **repositório** no meu **github**, mais especificamente no diretório **DynamicArray**

Resumo do tempo de execução.

Os métodos **size**, **redimensionar e percorrer,** independem de posições no array e por isso possuem o mesmo tempo de execução.

Obs: ***Inserção no final*** do array é **Linear*** apenas no pior caso e **constante** caso contrário.

Pior caso: Quando realocamos o array para um novo tamanho.

Operação	Inicio	Meio	Final
Inserir	Linear	Linear	Linear*
Remover	Linear	Linear	Constante
Acessar (get)	Constante	Constante	Constante
Alterar(set)	Constante	Constante	Constante
Size	Constante	Constante	Constante
Redimensionar	Linear	Linear	Linear
Percorrer	Linear	Linear	Linear

Se isso te ajudou, por favor dê seu **feedback** nos comentários! Bons estudos e obrigado! 😉

Rafael Guimarães 31 de março de 2020 Estruturas de Dados
#ArrayList, #DataStructures, #EstruturasdeDados, #vector

Deixe um comentário

Digite seu comentário aqui...