Maratona de programação 2024

Escrito por Diego Francisco

Vetores (Arrays)

Vetores, também conhecidos como arrays, são estruturas que armazenam múltiplos valores do mesmo tipo. Em C++, um vetor é definido com um tamanho fixo.

```
#include <iostream>

int main() {
    // Declaração e inicialização de um vetor com 5 elementos
    int numeros[5] = {1, 2, 3, 4, 5};

    // Acesso e impressão dos elementos do vetor
    for (int i = 0; i < 5; ++i) {
        std::cout << "Número na posição " << i << ": " << numeros[i] << std::endl;
    }

    return 0;
}
```

Modificação de Elementos Você pode acessar e modificar elementos do vetor usando o índice:

```
#include <iostream>

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};

    // Modificação do valor no índice 2
    numeros[2] = 35;

    // Impressão dos elementos modificados
    for (int i = 0; i < 5; ++i) {
        std::cout << "Número na posição " << i << ": " << numeros[i] << std::endl;
    }

    return 0;
}
```

Matrizes (Arrays Bidimensionais)

Matrizes são arrays de arrays. Em outras palavras, são vetores de vetores. Elas são úteis para representar tabelas e grids.

Declaração e Inicialização

```
#include <iostream>

int main() {
    // Declaração e inicialização de uma matriz 2x3
    int matriz[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Impressão dos elementos da matriz
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << "Elemento na posição [" << i << "][" << j << "]: " <<
matriz[i][j] << std::endl;
        }
    }

    return 0;
}
```

Modificação de Elementos Você pode acessar e modificar elementos da matriz usando dois índices:

```
#include <iostream>

int main() {
    int matriz[2][3] = {
        {10, 20, 30},
        {40, 50, 60}
    };

    // Modificação do valor na posição [1][1]
    matriz[1][1] = 55;

    // Impressão dos elementos modificados
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << "Elemento na posição [" << i << "][" << j << "]: " <<
matriz[i][j] << std::endl;
        }
    }

    return 0;
}
```

1. Busca Linear

A busca linear é um método simples que verifica cada elemento de um array, um por um, até encontrar o elemento desejado ou até o final do array.

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

```
int buscaLinear(const std::vector<int>& arr, int x) {
    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == x) {
            return i; // Retorna o índice do elemento encontrado
        }
    }
    return -1; // Retorna -1 se o elemento não for encontrado
}

int main() {
    std::vector<int> arr = {5, 3, 7, 1, 9};
    int x = 7;
    int resultado = buscaLinear(arr, x);
    if (resultado != -1)
        std::cout << "Elemento encontrado na posição " << resultado << std::endl;
    else
        std::cout << "Elemento não encontrado" << std::endl;
    return 0;
}
```

2. Busca Binária

A busca binária é eficiente para arrays ordenados, reduzindo o tempo de busca. Ela funciona dividindo o intervalo de busca pela metade a cada iteração.

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

int buscaBinaria(const std::vector<int>& arr, int x) {
    int low = 0, high = arr.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid; // Retorna o índice do elemento encontrado
        else if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Retorna -1 se o elemento não for encontrado
}

int main() {
    std::vector<int> arr = {1, 3, 5, 7, 9};
    int x = 7;
    int resultado = buscaBinaria(arr, x);
    if (resultado != -1)
        std::cout << "Elemento encontrado na posição " << resultado << std::endl;
    else
        std::cout << "Elemento não encontrado" << std::endl;
```

```
    return 0;
}
```

alguns dos principais algoritmos de ordenação que você pode usar em C++:

1. Bubble Sort

Um dos algoritmos mais simples, mas ineficiente para grandes conjuntos de dados. Ele compara pares de elementos adjacentes e os troca se estiverem na ordem errada, repetindo isso até que a lista esteja ordenada.

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                std::swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

2. Selection Sort

Esse algoritmo divide a lista em duas partes: uma ordenada e uma não ordenada. A cada iteração, ele seleciona o menor (ou maior) elemento da parte não ordenada e o move para a parte ordenada.

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIdx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        std::swap(arr[minIdx], arr[i]);
    }
}
```

3. Insertion Sort

Constrói a lista ordenada um item de cada vez, pegando elementos da parte não ordenada e inserindo-os na posição correta na parte ordenada.

```
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
```

```
            j--;
        }
        arr[j + 1] = key;
    }
}
```

4. Merge Sort

Um algoritmo de ordenação baseado em divisão e conquista. Divide o array em sub-arrays menores, ordena esses sub-arrays e depois os une (merge) para formar o array ordenado.

```
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = new int[n1];
    int *R = new int[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0;
    int j = 0;
    int k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    delete[] L;
    delete[] R;
```

```
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

5. Quick Sort

Outro algoritmo baseado em divisão e conquista. Seleciona um elemento pivô e particiona o array em duas partes, com elementos menores que o pivô à esquerda e maiores à direita, e então ordena recursivamente as duas partes.

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```