C/C++

Estruturas de Dados STL

Prof. Edson Alves - UnB/FGA

2019

Sumário

std::vector

Vector

Declaração do Vector

```
std::vector<tipo_dado> nome;
```

Um vector traz a ideia de um array dinamico, um armazenamento contínuo de elementos na memória que se realoca dinamicamente quando necessário.

Vector

- Inserção: possui complexidade constante ou linear, vai depender da necessidade de realocar o vetor ou não.
- Acesso: o acesso é constante pois é um simples acesso direto à uma localização da memória.
- Deleção: caso tente apagar um número ao final do vetor o tempo será constante, porém para apagar qualquer elemento que não seja o último o vetor terá de ser realocado.
- Pesquisa: achar um valor no vetor tem complexidade de tempo linear com o tamanho, pois precisa percorrer todo ele no pior caso, porém se estiver ordenado a pesquisa será logarítmica utilizando busca binária.

Vector

- Inserção:

```
vector<int> a { 1, 2 };
a.push_back(1);
// a = { 1, 2, 1 }
```

- Acesso:

```
vector<int> a { 0, 1 };
a[0]; // retorna 0
a.at(1); // retorna 1
a.front() // retorna 0
a.back() // retorna 1
```

Vector

- Deleção:

```
vector<int> a { 0, 1, 2 };
a.erase(a.begin() + 1); // a se torna { 0, 2 }
a.pop_back(); // a se torna { 0 }
a.clear(); // limpa a, a se torna {  }
```

- Pesquisa:

```
vector<int> a { 1, 2, 3, 4 };
auto it = find(a.begin(), a.end(), 2);
// it será um iterador para o elemento
auto bit = binary_search(a.begin(), a.end(), 2);
// bit é um iterador para o elemento encontrado com complexidade
// O(log n) pois é resultado de uma busca binária
```

Vector

- Informações:

```
vector<int> a { 1, 2, 3 };
a.size(); // retorna 3
a.empty(); // retorna false
```

Vector

```cpp
#include <vector>

using namespace std;

int main () {
    vector<int> v0;          // empty vector
    vector<int> v1(5);       // v1 = 0 0 0 0 0
    vector<int> v2(5, 1);    // v2 = 1 1 1 1 1
    vector<int> v3(v2);      // v3 = v2
    vector<int> v4 {1, 3};   // v4 = 1 3

    return 0;
}
```

Vector

```cpp
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

int main () {
    vector<int> v {0, 1, 2};

    v.push_back(3);
    // v : 0 1 2 3

    cout << v[0] << endl;
    // prints: 0

    cout << v.back() << endl;
    // prints: 3

    v.pop_back();
    cout << v.back() << endl;
    // prints: 2
```

Vector

```
23      // v : 0 1 2
24      auto it = v.begin() + 1;
25      v.erase(it);
26      // v : 0 2
27
28      auto found = find(v.begin(), v.end(), 2);
29      if(found != v.end())
30          cout << *found << endl;
31          // prints: 2
32
33      for(auto e : v)
34          cout << e << " ";
35      cout << endl;
36      // prints: 0 2
37
38      return 0;
39  }
```

std::queue

Queue

Declaração da Queue

```
std::queue<tipo_dado> nome;
```

No STL, a queue é um container associativo. Ela traz uma estrutura FIFO (first in, first out). É possível inserir elementos no fim da fila e retirar apenas da frente.

Queue

- Inserção: a inserção será sempre constante em relação à sua complexidade de tempo.
- Acesso: o acesso ao primeiro e ao último elemento é constante.
- Deleção: a interface permite apenas deletar o elemento da frente da fila, complexidade de tempo constante.

Queue

- Inserção:

```
queue<int> q;
q.push(1); // q = { 1 }
```

- Acesso:

```
queue<int> q { 1, 2, 3 };
q.front(); // 1
q.back(); // 3
```

Queue

- Deleção:

```
queue<int> q { 1, 2, 3 };
q.pop(); // q = { 2, 3 }
```

- Informações:

```
queue<int> q { 1, 2, 3 };
q.size(); // retorna 3
q.empty(); // retorna false
```

Queue

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

using namespace std;

int main () {
    queue<int> q;
    int f;

    q.push(1);
    q.push(10);

    while(!q.empty()) {
        f = q.front();
        foo(f);
        q.pop();
    }

    return 0;
}
```

std::stack

Stack

Declaração da Stack

```
std::stack<tipo_dado> nome;
```

No STL, o stack é um container associativo. Ele traz uma estrutura LIFO (last in, first out). É possível inserir elementos no topo da pilha e retirar apenas do topo.

Stack

- Inserção: a inserção será sempre constante em relação à sua complexidade de tempo.
- Acesso: o acesso ao primeiro e ao último elemento é constante.
- Deleção: a interface permite apenas deletar o elemento do topo da fila, complexidade de tempo constante.

Stack

- Inserção:

```
stack<int> s;
s.push(1); // s = { 1 }
```

- Acesso:

```
stack<int> s { 0, 1, 2 };
s.top(); // 2
```

Stack

- Deleção:

```
stack<int> s { 1, 2 };
s.pop(); // s = { 1 }
```

- Informações:

```
stack<int> s { 1, 2, 3 };
s.size(); // retorna 3
s.empty(); // retorna false
```

Stack

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

using namespace std;

int main () {
    stack<int> s;
    int t;

    s.push(1);
    s.push(10);

    while(!s.empty()) {
        t = s.front();
        foo(t);
        t.pop();
    }

    return 0;
}
```

std::set

Set

Declaração do set

```
std::set<tipo_dado> nome;
```

O set é uma estrutura de dado não linear, normalmente implementada com uma árvore binária vermelha-preta. Como ela é mantida balanceada o set possui boa performance para pesquisas.

Set

- Inserção: a inserção vai ter complexidade logarítmica em relação ao tamanho, é necessário achar o melhor lugar para inserir e manter a árvore balanceada.
- Deleção: também possui complexidade logarítmica em relação ao tamanho por causa da necessidade de manter a árvore balanceada.
- Pesquisa: achar um valor no set possui uma performance logarítmica, o que é um dos melhores motivos para se utilizar um set.

Set

- Inserção:

```
set<int> s;
s.insert(1); // s = { 1 }
```

- Deleção:

```
set<int> s;
s.insert(1); // s = { 1 }
s.erase(1); // retorna 1 (num de elementos apagados) e apaga
s.erase(2); // retorna 0
```

Set

- Pesquisa:

```
set<int> s { 1, 2, 3 };
s.find(2); // retorna um iterador para o elemento
```

- Informações:

```
set<int> s { 1, 2, 3 };
s.size(); // retorna 3
s.empty(); // retorna false
s.count(1); // retorna se um item está no set
```

Set

- O set não possui elementos repetidos

```
set<int> s;
s.insert(1); // s = { 1 }
s.insert(1); // s = { 1 }
```

Set

```cpp
#include <set>
#include <iostream>

using namespace std;

int main () {
    set<int> s { 1, 2, 3, 10, 15 };

    s.insert(1); // s = { 1, 2, 3, 10, 15 }

    auto it = s.find(2); // it é o iterador para o elemento 2

    for(auto i = it; i != s.end(); i++)
        cout << *i << " ";
    cout << endl;
    // 3 10 15

    return 0;
}
```

std::map

Map

Declaração do map

```
std::map<tipo_chave, tipo_valor> nome;
```

O map é uma estrutura de dado não linear, ela possui pares de chaves e valores. Para cada chave existe um valor.

Map

- Inserção:
- Deleção:
- Pesquisa:

Map

- Inserção:

```
map<string, int> m;
m["hello"] = 1;
```

- Deleção:

```
map<int, double> m;
m[1] = 2.01;
m.erase(1);
```

Map

- Pesquisa:

```
map<int, string> m;
m[100] = "bla";
m.find(100); // retorna iterador para o elemento
```

Map