

Árvores Binárias de Busca

STL

Prof. Edson Alves – UnB/FGA

1. Introdução
2. Set
3. map

Introdução

Árvores Binárias de Busca na STL

- A STL (*Standard Template Library*) da linguagem C++ não oferece uma implementação básica de árvores binárias de busca que permita o acesso direto aos nós e seus ponteiros
- Entretanto, ela oferece tipos de dados abstratos cujas implementações utilizam árvores binárias de busca auto-balanceáveis
- O padrão da linguagem não especifica qual árvore deve ser utilizada na implementação, e sim as complexidades assintóticas esperadas para cada operação
- Segundo o site CppReference¹, em geral são utilizadas árvores *red-black*
- Os principais tipos de dados abstratos implementados são os conjuntos (*sets*) e os dicionários (*maps*)

¹<https://en.cppreference.com/w/>

Set

- O conjunto (set) é um tipo de dado abstrato que representa um conjunto de elementos únicos
- Estes elementos são mantidos em ordem crescente, de acordo com a implementação do operador $<$ do tipo de elemento a ser armazenado
- O tipo de dado a ser armazenado é paramétrico, e deve ser definido na instanciação do conjunto
- A principal característica dos conjuntos é a eficiência nas operações de inserção, remoção e busca
- Todas as três tem complexidade $O(\log N)$, onde N é o número de elementos no conjunto

Construção de um set

- O padrão C++11 oferece cinco construtores distintos para um set
- O primeiro deles, denominado *default constructor*, não tem parâmetros, e constrói um conjunto vazio
- O segundo, *range constructor*, permite a construção de um conjunto a partir de dois iteradores, *first* e *last*, que determinam um intervalo de valores a serem inseridos, do primeiro ao penúltimo
- Este construtor também permite a definição de um alocador de memória customizado
- O terceiro, *copy constructor*, cria uma cópia exata do set passado como parâmetro
- O quarto, *move constructor*, move o conteúdo do set passado como parâmetro para o novo conjunto
- O quinto, *initializer-list constructor*, cria um novo set com os elementos passados na lista de inicialização

Exemplo de uso dos construtores do set

```
1 #include <set>
2 #include <string>
3 #include <iostream>
4
5 int main()
6 {
7     std::set<int> s1;                                // Conjunto de inteiros vazio
8
9     std::string s { "Teste" };
10    std::set<char> s2(s.begin() + 1, s.end()); // s2 = { 'e', 's', 't', 'e' }
11
12    std::set<char> s3(s2);                            // s3 == s2
13
14    std::set<char> s4(std::move(s3));                 // s4 == s2, s3 vazio
15
16    std::set<double> s5 { 2.0, 1.5, 3.7 };           // s5 = { 1.5, 2.0, 3.7 }
17
18    return 0;
19 }
```


Principais operações

- As principais operações em um conjunto são a inserção, remoção e busca, todas com complexidade $O(\log N)$, onde N é o número de elementos armazenados no conjunto
- A inserção é feita através do método `insert()`, que pode receber ou o valor a ser inserido ou uma lista de inicialização com os elementos a serem inseridos
- A inserção de um valor que já existe no conjunto não tem efeito
- O método `erase()` remove o nó que contém o valor passado como parâmetro, se existir tal valor no conjunto

- O retorno do método pode ser utilizado para se determinar quantos elementos foram removidos
- Para se determinar se um elemento está ou não no conjunto há duas alternativas
- A primeira é utilizar o método `count()`, cujo retorno significa o número de ocorrências do valor passado como parâmetro
- A segunda é utilizar o método `find()`: ele retorna o iterador para o elemento que contém o valor, ou o iterador `end()`, caso contrário

Exemplo de uso das principais operações do set

```
1 #include <set>
2
3 int main()
4 {
5     std::set<int> s;
6
7     s.insert(3);           // s = { 3 }
8     s.insert(3);           // s = { 3 }
9     s.insert( { 1, 2 } );  // s = { 1, 2, 3 }
10
11     auto n = s.erase(3);    // n = 1
12     n = s.erase(4);         // n = 0
13
14     n = s.count(1);          // n = 1
15
16     auto it = s.find(3);     /// n = s.end()
17
18     return 0;
19 }
```

Operações relevantes

- O método `empty()` verifica se o conjunto está ou não vazio
- O método `size()` determina o número de elementos armazenado no conjunto
- Tanto `empty()` quando `size()` tem complexidade constante
- O método `lower_bound()` retorna um iterator para o primeiro elemento do conjunto cuja informação é maior ou igual ao valor passado como parâmetro
- O método `upper_bound()` tem comportamento semelhante, retornando um iterator para o elemento cuja informação é estritamente maior do que valor passado como parâmetro
- Ambos métodos tem complexidade $O(\log N)$

Exemplo de uso de operações relevantes no set

```
1 #include <set>
2
3 int main()
4 {
5     std::set<int> s;
6
7     auto ok = s.empty();           // ok = true
8     auto N = s.size();             // N = 0
9
10    s.insert( {10, 20, 30, 40, 50} );
11
12    auto it = s.lower_bound(17);    // *it = 20
13    it = s.lower_bound(30);         // *it = 30
14
15    it = s.upper_bound(30);          // *it = 40
16    it = s.upper_bound(50);         // *it = s.end()
17
18    return 0;
19 }
```

- A STL também oferece a implementação de um conjunto que permite a inserção de elementos repetidos, denominado `multiset`
- O retorno do método `count()` corresponde ao número de ocorrências de um mesmo valor
- O método `erase()` deve ser usado com cuidado: ele apaga todas as ocorrências do valor passado como parâmetro
- Para remover somente uma ocorrência, esta ocorrência deve ser localizada com o método `find()` e o iterador de retorno deve ser passado como parâmetro para o método `erase()`
- Uma travessia usando *range for* passa uma vez em cada ocorrência de cada elemento
- O método `equal_range()` retorna um par de iteradores que delimitam o intervalo de valores idênticos ao valor passado como parâmetro

Exemplo de uso de multiset

```
1 #include <bits/stdc++.h>
2
3 int main() {
4     std::multiset<int> ms { 1, 2, 2, 2, 3 };
5     auto n = ms.count(2);           // n = 3
6     auto it = ms.find(2);
7
8     ms.erase(it);                   // ms = { 1, 2, 2, 3 }
9     n = ms.count(2);                 // n = 2
10    ms.erase(2);
11    ms.count(2);                      // n = 0
12
13    ms.insert( { 2, 2, 2, 2 } );
14    auto [a, b] = ms.equal_range(2);
15
16    for (auto i = a; i != b; ++i)
17        std::cout << *i << '\n';    // 2 2 2 2
18
19    return 0;
20 }
```

map

- map é um tipo abstrato de dados da STL do C++ que abstrai o conceito de dicionário
- Cada elemento do map é composto de uma chave (*key*) e um valor associado (*value*)
- Tanto o tipo da chave quanto do valor são paramétricos, e podem ser distintos
- Os elementos são ordenados por meio de suas chaves
- As chaves são únicas
- A inserção de um par (*key*, *value*) para uma chave já inserida modifica o valor da chave existente
- As operações de inserção, remoção e busca são eficientes, com complexidade $O(\log N)$, onde N é o número de elementos inseridos no map

- Embora sejam ADTs distintos, as interfaces do map e do set contém inúmeras interseções
- De fato, todos os métodos apresentados anteriormente para o set estão também disponíveis para o map
- A principal diferença reside no fato de que os iteradores do map são pares
- O primeiro elemento de um iterador é a chave e o segundo elemento é o valor
- Além do map, a STL também oferece o multimap, o qual suporta chaves repetidas

Exemplo de uso de map e multimap

```
1 #include <map>
2 #include <vector>
3 #include <iostream>
4
5 using namespace std;
6
7 int main()
8 {
9     map<int, int> m1;                                // Mapa de pares de inteiros vazio
10
11     map<int, char> m2 { { 1, 'a' }, { 2, 'b' }, { 3, 'c' } };
12
13     map<int, char> m3(m2);                            // m3 == m2
14
15     map<int, char> m4(m2.begin(), m2.end());          // m4 == m2
16
17     map<int, char> m5(move(m2));                      // m5 == m4, m2 vazio
18
19     map<string, int> m;
```

Exemplo de uso de map e multimap

```
21  m["abc"] = 1;                                // m = { "abc": 1 }
22  m.insert(make_pair("xyz", 2));                // m = { "abc": 1, "xyz": 2 }
23
24  m.erase("xyz");                              // m = { "abc": 1 }
25
26  auto it = m.find("xyz");                      // it == m.end()
27
28  auto n = m.count("abc");                      // n == 1
29
30  m.insert({ { "xyz", 2 }, { "rst", 3 } });
31
32  for (auto [x, y] : m)
33      cout << x << ": " << y << endl;
34
35  auto ok = m.empty();                         // ok == false
36  n = m.size();                                // n = 3
37
38  it = m.lower_bound("mno");                   // *it == { "rst", 3 };
39  it = m.lower_bound("abc");                   // *it == { "abc", 1 };
40  it = m.upper_bound("zzz");                   // *it == m.end();
```

Exemplo de uso de map e multimap

```
42  multimap<int,int> ms {{1, 1}, {1, 2}, {1, 2}, {1, 3}, {2, 1}, {2, 2}};
43
44  n = ms.count(1);           // n = 4
45  n = ms.count(2);           // n = 2
46
47  // 1: 1, 1: 2, 1: 2, 1: 3, 2: 1, 2: 2
48  for (auto it : ms)
49      cout << it.first << ": " << it.second << ", ";
50  cout << endl;
51
52  //ms[1] = 4;                // Erro de compilação!
53
54  ms.erase(1);
55
56  // 2: 1, 2: 2
57  for (auto it : ms)
58      cout << "--- " << it.first << ": " << it.second << endl;
59
60  return 0;
61 }
```

1. [CppReference – Map](#), acesso em 03/04/2019.
2. [CppReference – Multimap](#), acesso em 04/04/2019.
3. [CppReference – Multiset](#), acesso em 04/04/2019.
4. [CppReference – Set](#), acesso em 03/04/2019.