Árvores Binárias de Busca

STL

Prof. Edson Alves – UnB/FGA

Sumário

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 CppReference[1], 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

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 operator $<$ 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

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

int main()
{
    std::set<int> s1;                           // Conjunto de inteiros vazio

    std::string s { "Teste" };
    std::set<char> s2(s.begin() + 1, s.end());  // s2 = { 'e', 's', 't', 'e' }

    std::set<char> s3(s2);                      // s3 == s2

    std::set<char> s4(std::move(s3));           // s4 == s2, s3 vazio

    std::set<double> s5 { 2.0, 1.5, 3.7 };      // s5 = { 1.5, 2.0, 3.7 }

    return 0;
}
```

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
- Outro método de inserção é o `emplace()`, que recebe como parâmetros os mesmos parâmetros do construtor do elemento a ser inserido e constrói o elemento durante a inserção
- 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

Principais operações

- 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

```cpp
#include <set>

int main()
{
    std::set<int> s;

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

    auto n = s.erase(3);    // n = 1
    n = s.erase(4);         // n = 0

    n = s.count(1);         // n = 1

    auto it = s.find(3);    /// n = s.end()

    return 0;
}
```

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 iterador 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

```cpp
#include <set>

int main()
{
    std::set<int> s;

    auto ok = s.empty();            // ok = true
    auto N = s.size();              // N = 0

    s.insert( {10, 20, 30, 40, 50} );

    auto it = s.lower_bound(17);    // *it = 20
    it = s.lower_bound(30);         // *it = 30

    it = s.upper_bound(30);         // *it = 40
    it = s.upper_bound(50);         // *it = s.end()

    return 0;
}
```

multiset

- 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

```cpp
#include <bits/stdc++.h>

int main() {
    std::multiset<int> ms { 1, 2, 2, 2, 3 };
    auto n = ms.count(2);                           // n = 3
    auto it = ms.find(2);

    ms.erase(it);                                   // ms = { 1, 2, 2, 3 }
    n = ms.count(2);                                // n = 2
    ms.erase(2);
    ms.count(2);                                    // n = 0

    ms.insert( { 2, 2, 2, 2 } );
    auto [a, b] = ms.equal_range(2);

    for (auto i = a; i != b; ++i)
        std::cout << *i << '\n';                   // 2 2 2 2

    return 0;
}
```

map

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`

Operações um `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

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

using namespace std;

int main()
{
    map<int, int> m1;                           // Mapa de pares de inteiros vazio

    map<int, char> m2 { { 1, 'a' }, { 2, 'b' }, { 3, 'c' } };

    map<int, char> m3(m2);                      // m3 == m2

    map<int, char> m4(m2.begin(), m2.end());    // m4 == m2

    map<int, char> m5(move(m2));                // m5 == m4, m2 vazio

    map<string, int> m;
```

Exemplo de uso de map e multimap

```
21    m["abc"] = 1;                          // m = { "abc": 1 }
22    m.emplace("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 [k, v] : ms)
49          cout << k << ": " << v << ", ";
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 [k, v] : ms)
58          cout << "--- " << k << ": " << v << endl;
59
60      return 0;
61  }
```

Árvores com estatísticas

- As interfaces das estruturas baseadas em árvores binárias de busca da STL não fornecem, em sua API, métodos que permitam extrair estatísticas sobre a ordenação dos nós
- Contudo, o GCC implementa uma árvore binária oferece tais métodos
- Para utilizar estas árvores em códigos, é preciso incluir os seguintes arquivos *header*:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
```

- Atente também ao uso do **namespace __gnu_pbds**

Declaração da árvore com estatísticas

```
template<
        typename Key,
        typename Mapped,
        typename Cmp_Fn = std::less<Key>,
        typename Tag = rb_tree_tag,
        template<
        typename Const_Node_Iterator,
        typename Node_Iterator,
        typename Cmp_Fn_,
        typename Allocator_>
        class Node_Update = null_node_update,
        typename Allocator = std::allocator<char> >
class tree;
```

Parâmetros da árvore com estatísticas

- `typename Key` indica o tipo do elemento que será armazenado na árvore
- `typename Mapped` indica que a árvore seguirá a relação de chave-valor, como em `map`. Para usar a árvore como um conjunto, substitua este parâmetro pelo valor `null_type`
- `typename Tag` indica qual será o tipo de árvore balaceada que será implementada. Valores possíveis são `rb_tree_tag` para uma árvore *red-black*, `splay_tree_tag` para uma *splay tree* e `ov_tree_tag` para uma *ordered-vector tree*
- Em competições deve se usar a árvore *red-black*, pois as outras duas tem operações de *split* lineares
- `class Node_Update` indica se as estatísticas devem ser ou não adicionadas aos nós. O padrão é `null_node_update`: para obter as estatísticas, use neste parâmetro o valor `tree_order_statistics_node_update`

Métodos para obtenção de estatísticas

- Uma vez inicializada a árvore com os parâmetros apropriados, dois novos métodos se tornam disponíveis na interface
- O primeiro deles é o método `find_by_order()`, que recebe como parâmetro um número natural n e que retorna o iterador para o n-ésimo elemento da árvore, de acordo com a ordenação dos elementos, se existir
- O segundo método é o `order_by_key()`, que recebe um elemento x, do mesmo tipo dos elementos armazenados na árvore, e que retorna o número de elementos que são estritamente menores do que x
- Ambos métodos tem complexidade $O(\log N)$, onde N é o número de elememento armazenados na árvore

Exemplo de uso da árvore com estatísticas do GCC

```cpp
#include <bits/stdc++.h>

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<
    int,
    null_type,
    less<int>,      // Para um multiset use less_equal<int>
    rb_tree_tag,
    tree_order_statistics_node_update>
ordered_set;

int main()
{
    ordered_set s;
```

Exemplo de uso da árvore com estatísticas do GCC

```
21      s.insert(2);
22      s.insert(3);
23      s.insert(5);
24      s.insert(7);
25      s.insert(11);
26      s.insert(13);
27
28      cout << "Segundo primo: " << *s.find_by_order(1) << '\n';
29      cout << 11 << " é o " << s.order_of_key(11) + 1 << "º primo\n";
30
31      return 0;
32  }
```

Referências

1. **adamant**. C++ STL: Policy based data structures, acesso em 03/02/2025.
2. CppReference – Map, acesso eme 03/04/2019.
3. CppReference – Multimap, acesso em 04/04/2019.
4. CppReference – Multiset, acesso em 04/04/2019.
5. CppReference – Set, acesso em 03/04/2019.
6. **gcc-mirror/gcc**. tree_order_statistics.cc, acesso em 03/09/2025.