Estrutura de Dados para Conjuntos Ordenados

Bruno Monteiro, UFMG
Rafael Grandsire, UFMG

Resumo

Este artigo apresenta uma estrutura de dados para representar conjuntos e multiconjuntos ordenados de inteiros não-negativos utilizando uma *segment tree* dinâmica. A estrutura suporta operações clássicas, como inserção, remoção, consulta do k-ésimo menor elemento e cálculo de *order statistics*, além de operações avançadas de *split* e *merge*, todas com complexidade amortizada $\mathcal{O}(\log N)$. O trabalho descreve os fundamentos da estrutura, demonstra sua implementação e discute otimizações de memória baseadas em *tries* comprimidas. Este artigo é fortemente inspirado pelo blog por Ziqian Zhong no Codeforces [3].

1 O que ela pode fazer

Queremos uma estrutura de dados que possa ser pensada como um conjunto/multiconjunto de **inteiros não-negativos**, ou mesmo como um *array* ordenado. Queremos suportar todas as seguintes operações:

1. Criar uma estrutura vazia;
2. Inserir um elemento na estrutura;
3. Remover um elemento da estrutura;
4. Imprimir o k-ésimo menor elemento (se pensarmos na estrutura como um *array* ordenado `a`, estamos pedindo por `a[k]`);
5. Imprimir quantos números menores que x existem no conjunto (similar a `lower_bound` em um `std::vector`);
6. Dividir a estrutura em duas: uma contendo os k menores elementos, e a outra contendo o restante;
7. Fundir duas estruturas em uma (sem condições extras necessárias).

Acontece que, assumindo que todos os elementos são menores que N, podemos realizar qualquer sequência de t operações em tempo $\mathcal{O}(t \log N)$, ou seja, cada operação custa $\mathcal{O}(\log N)$ **amortizado** (como veremos, todas as operações exceto (7) requerem tempo $\mathcal{O}(\log N)$ no pior caso).

2 Como funciona

Vamos usar uma *segment tree* dinâmica para representar os elementos. Pense em cada um deles como um índice, uma folha em uma *segment tree*.

Então, inicialmente, não há nós na estrutura. Se inserirmos o valor 2, precisamos criar o caminho da raiz até a folha que representa 2 (ver Figura 1). Também é útil armazenar, em cada nó, o número de folhas criadas na subárvore daquele nó.

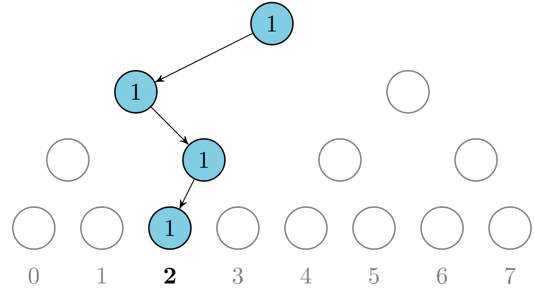

Figura 1: Representação do conjunto $S = \{2\}$.

Vamos também adicionar 1, 4 e 5. No final, a árvore ficará como na Figura 2.

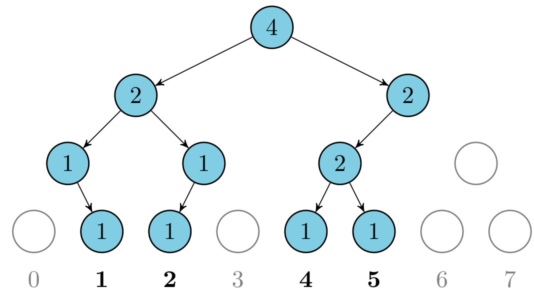

Figura 2: Representação de $S = \{1, 2, 4, 5\}$.

Usando esta representação, é muito direto implementar as operações (1), (2) e (3). As operações (4) e (5) também são fáceis, elas são operações clássicas de *segment tree*. Para fazer a operação (6), podemos descer pela árvore e chamar recursivamente para o filho esquerdo ou direito (Código 1).

```
node* split(node*& i, int k) {
  if (!k or !i) return NULL;
  node* ret = new node();
  if (i is a leaf) {
    // se for multiset
    i->cnt -= k;
    ret->cnt += k;
  } else {
    if (k <= i->left->cnt){
      // split esquerda
      ret->left = split(i->left, k);

    }
    else {
      // pega tudo da esquerda
      // split na direita
      ret->left = i->left;
      ret->right = split(i->right,
            k - i->left->cnt);
      i->left = NULL;
    }
  }
  return ret;
}
```

Código 1: Função `split`

Fica claro que todas estas operações custam $\mathcal{O}(\log N)$. Na operação (6), note que apenas descemos recursivamente para o filho esquerdo ou direito (Linha 12 ou Linha 19).

Mas e a operação (7)? Acontece que podemos unir duas estruturas da seguinte forma: para unir as subárvores definidas pelos nós `l` e `r`, se um deles for vazio, apenas retorne o outro. Caso contrário, una recursivamente `l->left`

e `r->left`, e `l->right` e `r->right`. Depois disso, podemos deletar `l` ou `r`, porque eles agora são redundantes: precisamos apenas de um deles.

Agora vamos ver porque qualquer sequência de t operações levará tempo $\mathcal{O}(t \log N)$. O número de nós que criamos é limitado por $\mathcal{O}(t \log N)$, pois cada operação cria no máximo $\mathcal{O}(\log N)$ nós. Agora note que o algoritmo de união ou retorna em tempo $\mathcal{O}(1)$, ou deleta um nó. Então o número total de vezes que o algoritmo não retorna em tempo $\mathcal{O}(1)$ é limitado pelo número total de nós criados, então é $\mathcal{O}(t \log N)$.

3 Implementação

Existem alguns truques de implementação que tornam o código mais fácil de escrever e usar. Primeiramente, não precisamos definir N como uma constante. Em vez disso, podemos ter $N = 2^k - 1$ para algum k, e se quisermos inserir um elemento maior que N, podemos apenas aumentar k, e atualizar a *segment tree* (apenas precisaremos criar uma nova raiz, e definir seu filho esquerdo como a raiz antiga).

Além disso, é fácil mudar entre representação de conjunto e multiconjunto. Também é fácil inserir x ocorrências de um elemento de uma vez, apenas aumente a variável `cnt` da folha respectiva de acordo. Esta implementação usa $\mathcal{O}(t \log N)$ de memória, para t operações.

A implementação completa encontra-se em `https://shorturl.at/cvWQG`.

4 Como usar

O código pode representar tanto conjunto quanto multiconjunto. Para criar uma estrutura vazia:

```
// set
sms<int> s;
// multiset
sms<int, true> s;
// Para inserir mais de 1e9 elementos
sms<int, true, long long> s;
```

A maioria das operações é similar a `std::set`:

```
// multiset
sms<int, true> s;
// insere valor '3'
s.insert(3);
// insere valor '4' tres vezes
s.insert(4, 3);
// apaga uma ocorrencia do valor '4'
s.erase(4);
// apaga 5 ocorrencias do valor '3'
s.erase(3, 5);
// apaga todas as ocorrencias de '4'
s.erase_all(4);
```

Operações de *order statistic*:

```
sms<int, true> s = {4, 7, 3, 4, 1, 6, 3};
// agora s = {1, 3, 3, 4, 4, 6, 7}
cout << s.order_of_key(4) << endl;
// printa "3"
cout << s[0] << " " << s[3] << " " << s[5] << endl;
// printa "1 4 6"
```

Finalmente, operações de *split* e *merge*:

```
sms<int, true> s = {1, 3, 3, 4, 4, 6, 7};
sms<int, true> s2;
// pega os 4 menores valores de s
s.split(4, s2);
// agora s = {4, 6, 7}, s2 = {1, 3, 3, 4}

sms<int, true> s3 = {2, 4, 6};
s3.merge(s2);
// agora s3 = {1, 2, 3, 3, 4, 4, 6}
```

5 Otimização de memória

Outra forma de pensar sobre esta estrutura de dados é como uma **trie**. Se quisermos inserir os números 1, 2, 4 e 5, podemos pensar neles como `001`, `010`, `100` e `101`.

Usando esta ideia, é possível representar esta *trie* de forma compactada, como uma *Patricia Trie* ou *Radix Trie*: apenas precisamos armazenar os nós não-folha que possuem 2 filhos (ver Figura 3). Então, apenas precisamos armazenar nós que são LCA de algum par de folhas. Se tivermos t folhas, isso são $2t - 1$ nós.

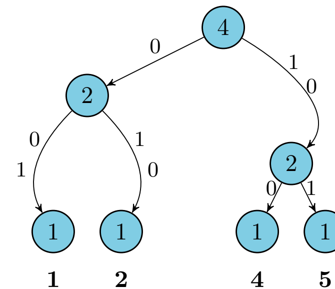

Figura 3: A *trie* comprimida da Figura 2.

Isso é bem complicado de ser implementado, mas reduz o uso de memória de $\mathcal{O}(t \log N)$ para $\mathcal{O}(t)$: `https://shorturl.at/O5n1z`.

6 Considerações finais

O principal da ideia já foi explicado no blog de Ziqian Zhong [3], e também encontrei um artigo [2] que descreve seu uso como um dicionário (map). Além disso, enquanto eu escrevia isto, Lucian Bicsi postou um blog [1] em que ele mostra uma forma mais fácil de ter a otimização de memória, mas parece tornar a operação de *split* mais complicada.

Referências

[1] Lucian Bicsi. Compressed segment trees and merging sets in o(n logu). `https://codeforces.com/blog/entry/83170`. Data de acesso: 04/04/2025.

[2] Adam Karczmarz. A simple mergeable dictionary. In *15th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2016)*, pages 7–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[3] Ziqian Zhong. Using merging segment tree to solve problems about sorted list. `https://codeforces.com/blog/entry/49446`. Data de acesso: 04/04/2025.