

Árvore de Fenwick

Definição, inserção e remoção

Prof. Edson Alves - UnB/FGA

2019

1. Motivação
2. Definição
3. Range sum queries

Motivação

Range sum query

- O problema conhecido como *range sum query* ($RSQ(i, j)$) consiste em determinar a soma de todos os elementos de uma sequência a_k cujos índices pertencem ao intervalo $[i, j]$
- Por exemplo, se $a_k = \{1, 2, 3, 4, 5\}$, então $RSQ(2, 4) = 9$, $RSQ(3, 3) = 3$ e $RSQ(1, 5) = 15$
- Cada *query* pode ser respondida em $O(N)$, onde N é o número de elementos da sequência a_k , iterando em todos os elementos a_i, a_{i+1}, \dots, a_j
- Se, para um mesmo vetor a_k , forem realizadas Q *queries*, a complexidade do algoritmo seria $O(NQ)$
- Contudo, esta complexidade pode ser reduzida para $O(N + Q)$, se a sequência for preprocessada, de modo que cada *query* pode ser respondida em $O(1)$

Implementação do *RSQ* com complexidade $O(N)$ por query

```
1 template<typename T>
2 T RSQ(const vector<T>& as, int i, int j)
3 {
4     T sum = 0;
5
6     for (int k = i; k <= j; ++k)
7         sum += a[k];
8
9     return sum;
10 }
```

Soma dos prefixos de uma sequência

- Seja p_i a soma dos elementos do prefixo $a[1..i]$ da sequência a_k , isto é,

$$p_i = \sum_{k=1}^i a_k$$

- Por exemplo, para $a_k = \{1, 2, 3, 4, 5\}$, segue que $p_k = \{1, 3, 6, 10, 15\}$
- A sequência p_k , com $p_0 = 0$, pode ser utilizada para determinar $RSQ(i, j)$ em $O(1)$
- De fato,

$$RSQ(i, j) = p_j - p_{i-1}$$

- Como a sequência p_k é gerada com complexidade $O(N)$, a resposta para Q tem complexidade $O(N + Q)$

Implementação da soma dos prefixos

```
1 // Os elementos as tem índices de 1 a N
2 template<typename T>
3 vector<T> prefix_sum(const vector<T>& as, int N)
4 {
5     vector<T> ps(N + 1, 0);
6
7     for (size_t i = 1; i <= N; ++i)
8         ps[i] = ps[i - 1] + as[i];
9
10    return ps;
11 }
12
13 template<typename T>
14 T RSQ(const vector<T>& ps, int i, int j)
15 {
16     return ps[j] - ps[i - 1];
17 }
```

Range sum queries com sequência dinâmica

- Embora a soma dos prefixos resolva o problema quando a sequência a_k é estática, ela não se aplica em sequências dinâmicas
- Se a sequência modificar um ou mais de seus elementos entre duas *queries*, a sequência p_k também fica modificada, e esta atualização tem complexidade $O(N)$, voltando à complexidade da solução original
- Para responder *range sum queries* em sequências dinâmicas com melhor complexidade é preciso o uso de estruturas mais sofisticadas, que permitam a atualização das somas pré-computadas de forma eficiente
- Uma destas estruturas é a *Binary Indexed Tree* (*BITree* ou *Fenwick Tree*), proposta pelo professor Peter M. Fenwick em 1994

Definição

Árvore de Fenwick

- A árvore de Fenwick (*Binary Indexed Tree*) é uma estrutura de dados que permite responder *range sum queries* de forma eficiente
- Ela suporta as operações de *range sum query* e a atualização de valores da sequência com complexidade $O(\log N)$
- A ideia principal da árvore de Fenwick é armazenar as somas de certos intervalos de índices da sequência de modo que qualquer intervalo $[1, j]$ possa ser representado unicamente pela união de, no máximo, $O(\log N)$ intervalos disjuntos
- Embora ela seja uma árvore, ela é implementada implicitamente por meio de um *array*, de forma semelhante à utilizada pelas *heaps* binárias

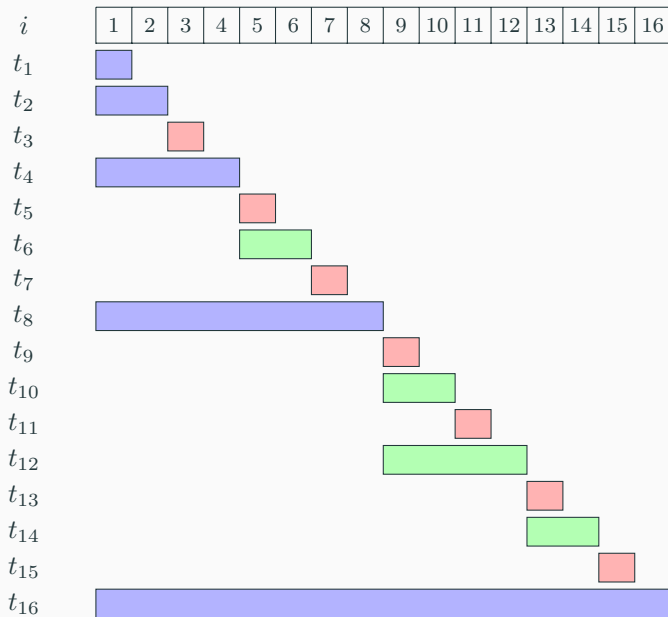
Implementação de uma árvore de Fenwick

- Assim como as *heaps* binárias, o índice zero do *array* t é descartado
- Seja $p(n)$ a maior potência de 2 que divide o inteiro positivo n
- Assim, t_i será a soma dos elementos de a_k cujos intervalos pertencem ao intervalo $I_i = [i - p(i) + 1, i]$, isto é,

$$t_i = \sum_{k=i-p(i)+1}^i a_k$$

- Por exemplo, para $a_k = \{1, 2, 3, 4, 5, 6, 7, 8\}$, segue que $t_k = \{1, 3, 3, 10, 5, 11, 7, 36\}$
- Esta escolha de intervalos permite a representação de qualquer intervalo $[1, j]$ por meio da união de $O(\log N)$ intervalos disjuntos cujas somas estão armazenadas no *array* t_k

Visualização dos intervalos correspondentes aos elementos t_i



Exemplo de representação de uma BITree em C++

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 template<typename T>
6 class BITree {
7 private:
8     vector<T> ts;
9     size_t N;
10
11 public:
12     BITree(size_t n) : ts(n + 1, 0), N(n) {}
13 }
```

Range sum queries

Range sum query em uma árvore de Fenwick

- Considere que

$$[1, j] = I_{k_1} \cup I_{k_2} \cup \dots I_{k_r},$$

com $r = O(\log N)$, $I_i \cap I_j = \emptyset$ se $i \neq j$ e I_k é o intervalo associado a t_k

- Por exemplo,

$$[1, 15] = [1, 8] \cup [9, 12] \cup [13, 14] \cup [15, 15]$$

- Defina

$$S(j) = \sum_{i=k_1}^{k_r} t_i$$

- Assim, fazendo $t_0 = 0$, segue que

$$RSQ(i, j) = S(j) - S(i - 1)$$

Identificação da decomposição de intervalos

- Para encontrar a decomposição de um intervalo $[1, j]$ em intervalos disjuntos associados aos elementos t_k , é preciso computar os valores de $p(n)$
- De fato, $p(n)$ corresponde ao *bit* menos significativo de n em base binária
- Este *bit* pode ser determinado de forma eficiente através de uma operação binária

$$p(n) = n \wedge (-n),$$

onde \wedge corresponde ao “e” lógico

- Os índices r_i dos intervalos I que decompõem $[1, j]$ foram a sequência de inteiros positivos

$$j, j - p(j), [j - p(j)] - p(j - p(j)), \dots,$$

Implementação da RSQ em uma BITree

```
14  T RSQ(int i, int j)
15  {
16      return RSQ(j) - RSQ(i - 1);
17  }
18
19 private:
20     int LSB(int n) { return n & (-n); }
21
22     T RSQ(int i)
23     {
24         T sum = 0;
25
26         while (i >= 1)
27         {
28             sum += ts[i];
29             i -= LSB(i);
30         }
31
32         return sum;
33     }
34
```

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k								
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2							
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2						
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1					
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6				
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1			
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4		
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4	-2	
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Construção da árvore de Fenwick

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = RSQ(1, 7) - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = \mathbf{RSQ(1, 7)} - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = 8 - RSQ(1, 2)$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = 8 - \mathbf{RSQ(1, 2)}$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = 8 - 2$$

Visualização de uma RSQ em uma BITree

Cálculo de $RSQ(3, 7)$

t_k	2	2	1	6	-1	4	-2	12
	1	2	3	4	5	6	7	8
a_k	2	0	1	3	-1	5	-2	4

$$RSQ(3, 7) = 6$$

Construção e atualização de uma BITree

Construção de uma árvore de Fenwick

- Uma maneira de se construir uma árvore de Fenwick é começar com um vetor cujos elementos são todos iguais a zero
- Depois, cada valor a ser atribuído à i -ésima posição deve ser somado a esta posição
- Como a operação de atualização/soma tem complexidade $O(\log N)$, onde N é o número máximo de elementos a serem inseridos na árvore, esta rotina tem complexidade $O(N \log N)$

1. **FENWICK**, Peter M. [A New Data Structure for Cumulative Frequency Tables](#), Journal of Software: Practice and Experience, volume 24, issue 3, 1994.
2. HackerEarth. [Fenwick \(Binary Indexed\) Trees](#), acesso em 06/05/2019.
3. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2010.
4. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, acesso aberto¹.
5. Wikipedia. [Fenwick Tree](#), acesso em 06/05/2019.

¹<https://cses.fi/book/index.html>