# Segment Tree

**Rafael Soratto**[1], **Michel Gomes**[1], **João Marcos P. Lopes**[1]

[1]Programa de Pós Graduação em Ciência da Computação
PPGCC – Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

{soratto,michels,joaolopes}@alunos.utfpr.edu.br

***Resumo.*** *Este artigo tem a finalidade de apresentar a estrutura de dados Segment Tree, seu contexto, suas implementações e suas aplicações práticas.*

## Sumário



## 1. Introdução

Este trabalho foi realizado para a disciplina de Estrutura de Dados (PPGCC02) do programa de mestrado PPGCC de Campo Mourão - Paraná - Brasil. O professor André

Kawamoto solicitou que cada equipe fizesse uma descrição completa sobre diferentes estruturas de dados relevantes para programação competitiva.

As estruturas de dados são essenciais para desenvolver algoritmos eficientes [Gupta 2020, Canuto 2021]. Uma dessas estruturas é a **árvore de segmentos**, que é muito útil para **consultas e atualizações eficientes em intervalos de elementos**. Neste artigo, vamos explorar os conceitos básicos das segment trees, seus algoritmos e aplicações.

A árvore de segmentos é uma estrutura de dados que armazena informações sobre intervalos de um vetor em uma árvore. Suas principais vantagens são:

1. **Busca eficiente** de intervalos;
2. **Flexibilidade** nas modificações do vetor.

### 1.1. Range minimum interval (RMQ)

Um exemplo de pesquisa de intervalo é o "**Range Minimum Query**" (RMQ), um problema comum em algoritmos e estruturas de dados. Ele envolve encontrar o valor mínimo em um intervalo contínuo de uma sequência de números. Essa consulta é frequentemente realizada em conjuntos de dados grandes, onde é necessário identificar o valor mínimo em um intervalo específico [Fischer 2011].

A solução para o problema RMQ geralmente envolve a construção de uma estrutura de dados como uma árvore de segmentos ou uma tabela esparsa. Essas estruturas permitem consultas rápidas e eficientes de mínimo em intervalos, armazenando informações pré-processadas sobre a sequência original [Fischer 2011].

O problema RMQ é amplamente utilizado em várias áreas da computação, como processamento de sequências de DNA, processamento de imagens e pesquisa em texto. Ele desempenha um papel fundamental em muitos algoritmos e aplicativos, onde a eficiência no processamento de intervalos e a rápida identificação do valor mínimo são requisitos essenciais [Fischer 2011].

## 2. Segmentation Tree (Árvore de segmentos)

Árvore de segmentos é uma **árvore binária completa**. Isso significa que todos os níveis da árvore estão completamente preenchidos, exceto possivelmente o último nível, que é preenchido da esquerda para a direita. Cada nó interno da árvore representa um **intervalo contíguo de elementos do vetor**, e cada nó folha representa um único elemento do vetor. Essa estrutura garante que a árvore de segmentos seja **balanceada** e eficiente para realizar **consultas** e **atualizações** em intervalos.

Nesta estrutura, divide-se o conjunto em intervalos menores **recursivamente** até que cada nó represente um único elemento. Cada nó contém **informações agregadas**, como a soma, o mínimo ou o máximo dos elementos em seu intervalo correspondente [de Berg et al. 2000, Cormen et al. 2009].

Essa estrutura **acelera** os cálculos das **consultas de intervalo** mínimo (descritas na seção anterior). Cada nó da árvore divide seus filhos pela metade, formando uma árvore binária [de Berg et al. 2000, Cormen et al. 2009].

Utilizando a árvore de segmentos as consultas em intervalos pode ser realizadas em **tempo** $O(\log n)$. Além disso, permite modificações no vetor, substituindo um elemento ou alterando todos os elementos de um sub-segmento inteiro.

É possível adicionar operações complexas em suas verificações para obter queries mais complexas. Também é possível escalar ela para dimensões maiores. Por ser uma **estrutura flexível** pode resolver vários tipos de problemas [CP-Algorithms 2023].

**Propriedade importante das Segment Trees**

Elas requerem apenas uma **quantidade linear de memória**. A árvore de segmentos padrão requer $4n$ nós para trabalhar em um vetor de tamanho $n$

### 2.1. Definição Formal da Árvore de Segmentos

A definição formal da árvore de segmentos é a seguinte:

Dado um vetor $a[0 \dots n-1]$, a árvore de segmentos permite realizar diversas operações simples em tempo $O(\log n)$, tais como:

- Calcular a soma dos elementos entre os índices $l$ e $r$ ($\sum_{i=l}^{r} a[i]$);
- Atualizar os valores dos elementos no vetor ($a[i] = x$).

### 2.2. Estrutura da Árvore de segmentos

A abordagem de **dividir e conquistar** pode ser utilizada nos segmentos de um vetor. Durante cada interação para construir a árvore calcula-se e armazena-se uma operação (por exemplo: soma, mínimo e máximo) para cada nível da árvore.

**Recursão**

Por exemplo: um vetor de tamanho $a[0 \dots n-1]$ é separado em duas metades $a[0 \dots n/2-1]$ e $a[n/2 \dots n-1]$ e calculamos e armazenamos a soma de cada metade. Cada uma dessas duas metades, por sua vez, é dividida ao meio recursivamente até que todos os segmentos atinjam o tamanho 1.

Portanto, os **segmentos formam uma árvore binária** onde a raíz é o segmento $a[0 \dots n-1]$, e cada **vértice intermediário** (não folha) tem exatamente **dois filhos**. A representação da estrutura requer um **número linear de vértices**. A maior parte das implementações não são construídas explicitamente. No pior caso, o **número de vértices** é estimado pela soma:

$$1 + 2 + 4 + \ldots + 2^{\lceil \log_2 n \rceil} \leq 2^{\lceil \log_2 n \rceil + 1} \leq 4n$$

.

Sempre que $n$ não for uma potência de dois, nem todos os níveis da Árvore de Segmentos serão totalmente preenchidos. Isto interfere diretamente na implementação.

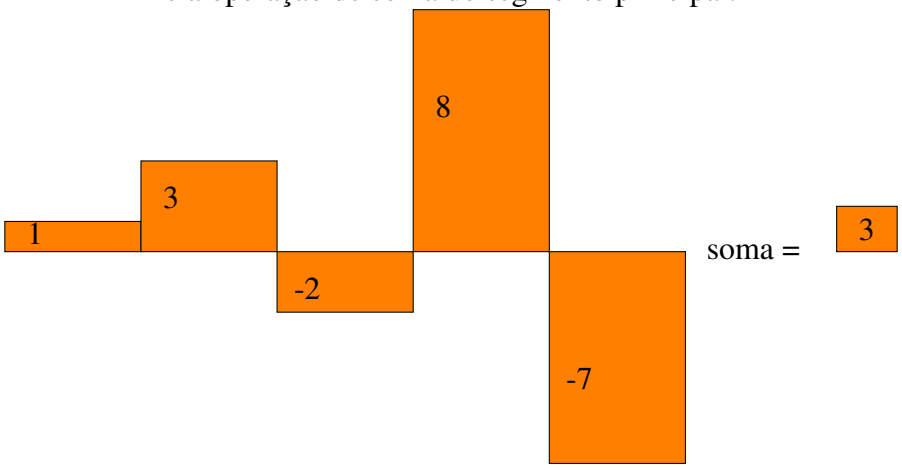
### 2.3. Exemplos

Por exemplo, dado um vetor

$$a = [1, 3, -2, 8, -7]$$

que pode ser representado pela Figura 1:

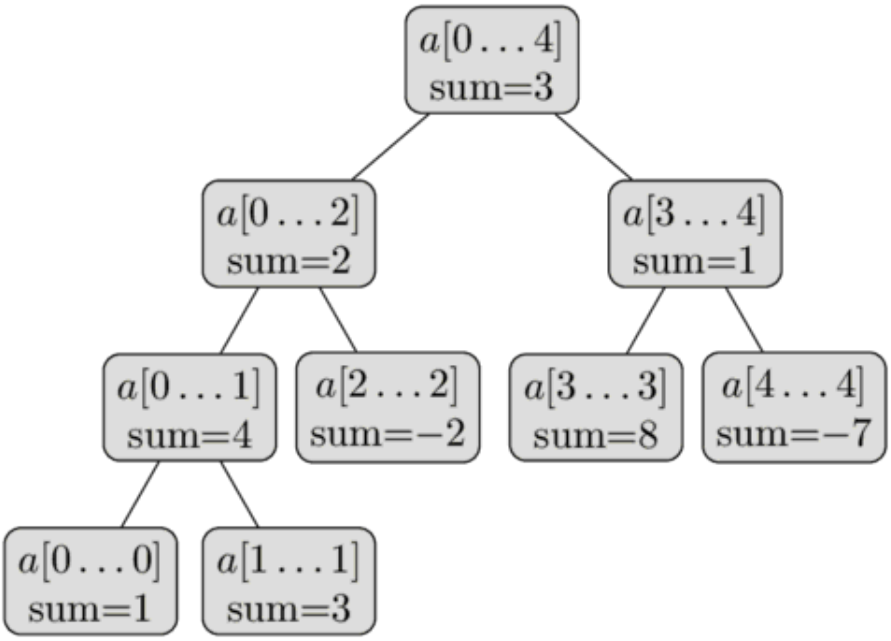
**Figura 1. Visualização do vetor** $a = [1, 3, -2, 8, -7]$
e a operação de soma do segmento principal.




**Fonte**: Autoria Própria 2023 (feito no tikz).

Podemos saber de forma rápida e eficiente qual a soma de todo vetor e também de cada nível de segmento da árvore conforme apresentado na Figura 2.

**Figura 2. Árvore de Segmentos do vetor** $a$ **utilizando a operação de soma.**




**Fonte :** [CP-Algorithms 2023].

De acordo com a Figura 2 podemos confirmar algumas propriedades da árvore de segmentos:

- Utilizando a operação de soma, sempre a raíz possuirá a soma de todos elementos no vetor:

  - Ou seja, a soma total do intervalo $0 \ldots N-1$ que neste caso é $a[0 \ldots 5-1] = a[0 \ldots 4]$.
- Durante a divisão recursiva pela metade, eventualmente serão gerados intervalos unitários que são as folhas da árvore.
  - Ou seja, os dois primeiros segmentos da recursão são $a[0 \ldots 2]$ e $a[2 \ldots 4]$.
  - A primeira folha é o elemento mais a esquerda (posição $a[0 \ldots 0]$);
  - E a última folha o elemento mais a direita (posição $a[4 \ldots 4]$).

## 3. Construção da Árvore e das Consultas

Para construir cada segmento da árvore é utilizada uma função $F(x) = y$ onde:

1. $F(x)$ é a operação a ser realizada e consequentemente o valor que será armazenado em cada segmento: a soma dos elementos do segmento, o menor/maior número do segmento, entre outros tipos. No exemplo, armazena-se a soma dos valores do intervalo $f(x)$ realizada em um intervalo $[0, n]$.
2. E $y$ é o resultado dessa função, por exemplo, a soma dos elementos do segmento ou o menor número.

Para construção da árvore é necessário começar pelas folhas e aplicando a função de merge $F(x)$ para armazenar o valor de todos segmentos até a raíz.

O procedimento de construção, se chamado em um vértice não folha, faz o seguinte:

1. construir recursivamente os valores dos dois vértices filhos;
2. mesclar os valores calculados desses filhos.

Inicia-se a construção no vértice raiz, portanto, ele pode armazenar o resultado das operações em toda a árvore de segmentos. O **tempo de construção** é de $O(n)$ neste caso assumindo que o tempo da operação de merge é constante (chamada N vezes).

### 3.1. Construção das Consultas

A consulta recebe dois inteiros $l$ e $r$ para calcular a soma dos elementos $a[l \ldots r]$ em tempo $O(\log n)$. O cálculo da **consulta é uma travessia da árvore**, que se espalha por todos os ramos necessários da árvore e usa os valores de soma pré-computados dos segmentos na árvore.

**A complexidade da consulta** é $O(\log n)$ por causa dos **níveis da árvore**.

Para cada nível, visitamos não mais do que quatro nós.

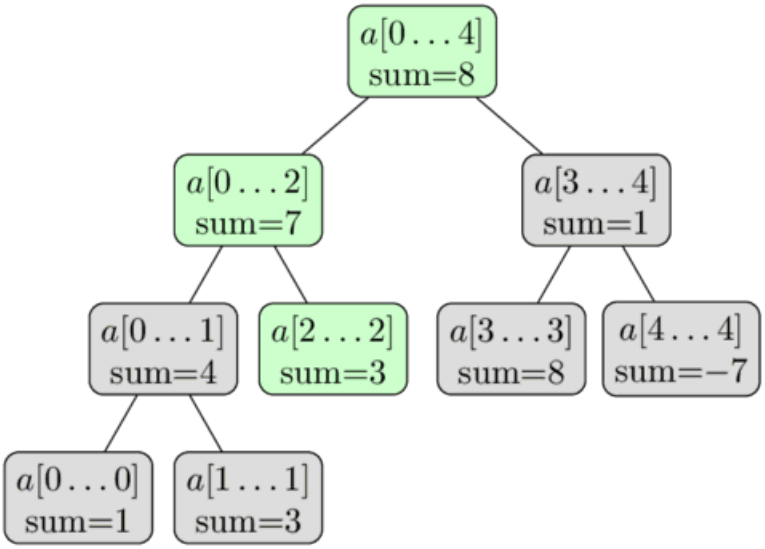
E como a altura da árvore é $O(\log n)$, obtém-se o tempo de execução desejado.

Ou seja, visitamos no máximo $4 \log n$ vértices no total, e isso é igual a um tempo de execução de $O(\log n)$.

### 3.2. Atualização de Consultas

Para alterar um elemento no vetor e reconstruir a árvore basta considerar que cada nível de uma Árvore de Segmentos forma uma partição do vetor. Portanto, um elemento $a[i]$ contribui apenas para um segmento de cada nível.

**Figura 3. Atualização do elemento $a[2] = 3$**




**Fonte:** [CP-Algorithms 2023].

Portanto, o **número de vértices** que precisão ser **atualizados** é $O(\log n)$.

A função recebe o vértice da árvore atual e chama a si mesma recursivamente com um dos dois vértices filhos (aquele que contém $a[i]$ em seu segmento), e depois disso recalcula seu valor de soma, similar ao que é feito no método de construção. Esse processo está presente na Figura 3:

## 4. Implementação

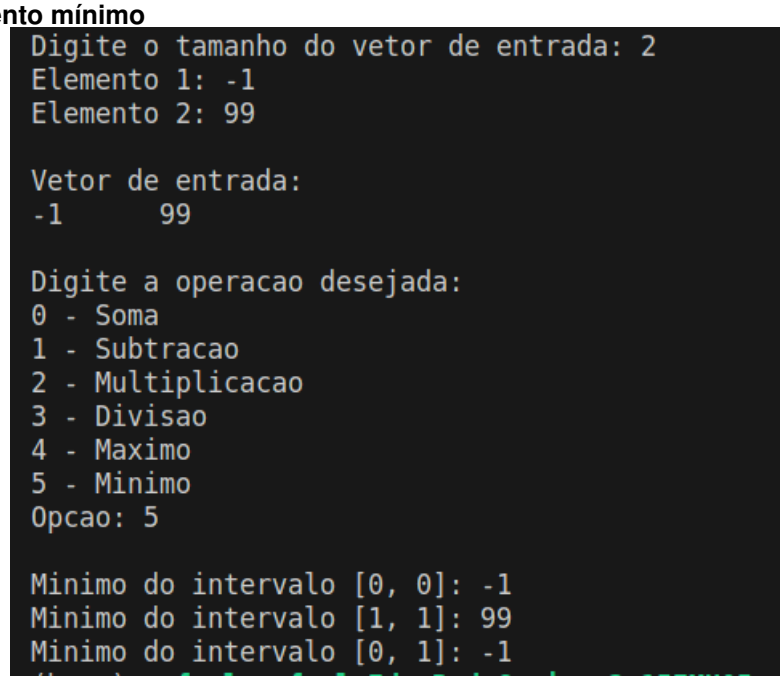
### 4.1. Código

O código em C apresentado implementa as operações básicas da Árvore de Segmentos, como a construção da árvore e 5 tipos diferentes de consultas:

1. Soma;
2. Subtração;
3. Multiplicação;
4. Divisão;
5. Máximo;
6. Mínimo.

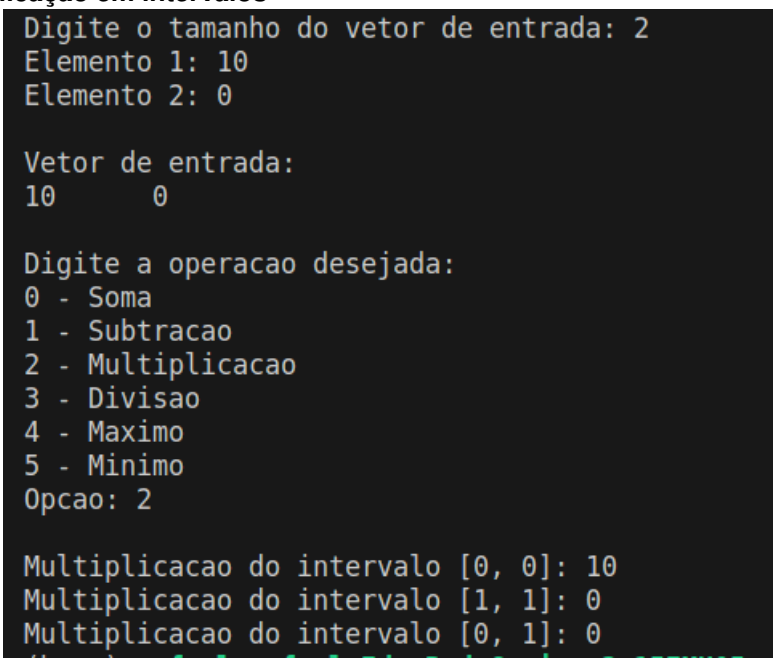
Criamos uma interface para o usuário colocar o input e selecionar a opção desejada:

**Figura 4. Exemplo de entrada e saída de dados do segment-tree para consulta de elemento mínimo**



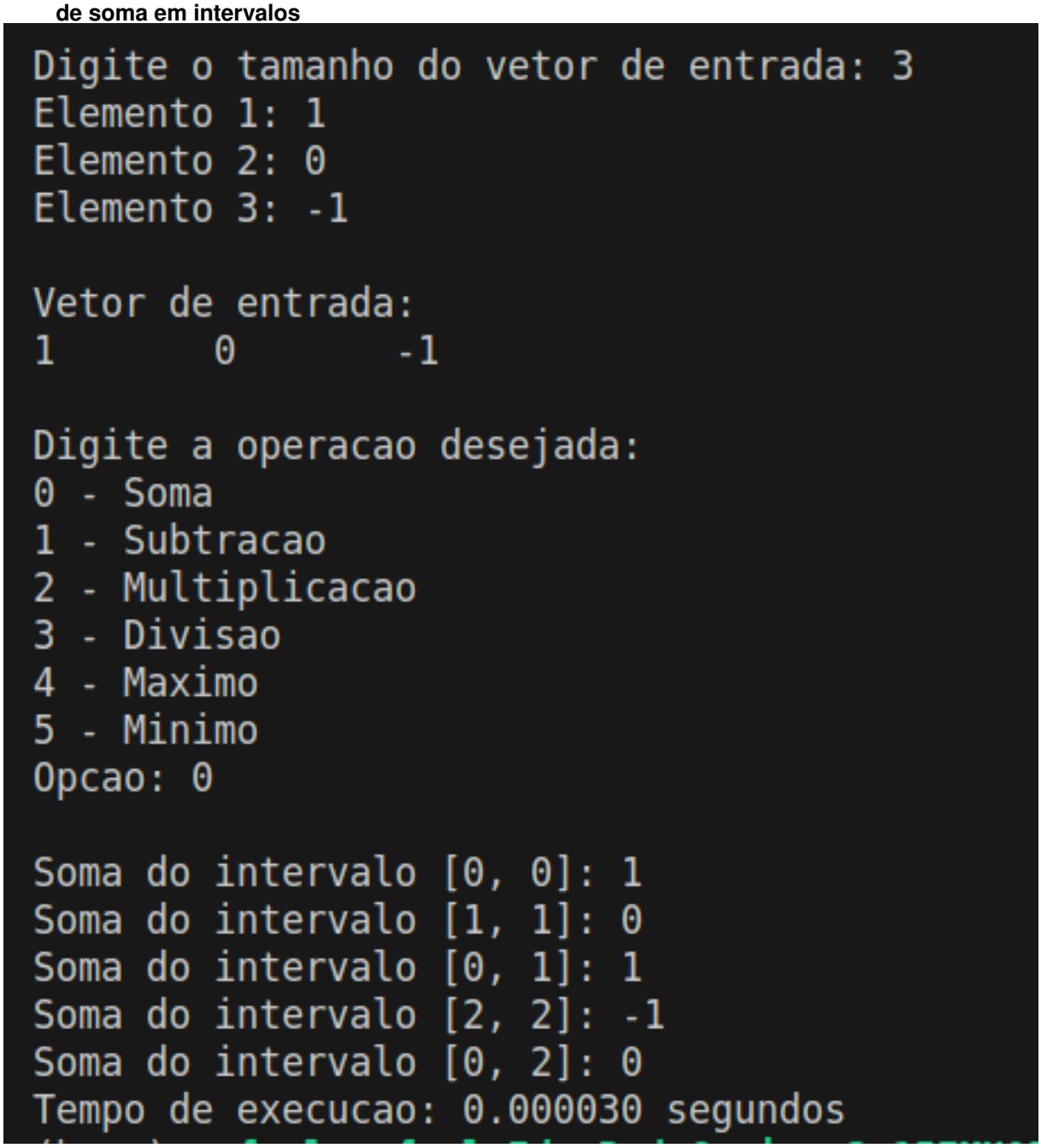

**Fonte:** Autoria Própria 2023.

**Figura 5. Exemplo de entrada e saída de dados do segment-tree para consulta de multiplicação em intervalos**




**Fonte:** Autoria Própria 2023.

**Figura 6. Exemplo de entrada e saída de dados do segment-tree para consulta de soma em intervalos**




**Fonte:** Autoria Própria 2023.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define OP_SOMA 0
#define OP_SUBTRACAO 1
#define OP_MULTIPLICACAO 2
#define OP_DIVISAO 3
#define OP_MAX 4
#define OP_MIN 5

typedef struct SegmentTree {
    int *tree;
    int *arr;
    int size;
} SegmentTree;

int getMid(int start, int end) {
    return start + (end - start) / 2;
}

int queryOp(SegmentTree st, int node, int start, int end, int left, int right) {
    if (left <= start && right >= end) {
        return st.tree[node];
    }
    if (end < left || start > right) {
        return 0;
    }
    int mid = getMid(start, end);
    int leftSum = queryOp(st, 2 * node + 1, start, mid, left, right);
    int rightSum = queryOp(st, 2 * node + 2, mid + 1, end, left, right);
    return leftSum + rightSum;
}

void buildSegmentTreeUtil(SegmentTree st, int node, int start, int end, int op) {
    if (start == end) {
        st.tree[node] = st.arr[start];
    } else {
        int mid = getMid(start, end);
        buildSegmentTreeUtil(st, 2 * node + 1, start, mid, op);
        buildSegmentTreeUtil(st, 2 * node + 2, mid + 1, end, op);

        if (op == OP_SOMA) {
            st.tree[node] = st.tree[2 * node + 1] + st.tree[2 * node + 2];
        }
        else if (op == OP_SUBTRACAO) {
            st.tree[node] = st.tree[2 * node + 1] - st.tree[2 * node + 2];
        }
        else if (op == OP_MULTIPLICACAO) {
            st.tree[node] = st.tree[2 * node + 1] * st.tree[2 * node + 2];
```

```
51          }
52          else if (op == OP_DIVISAO) {
53              st.tree[node] = st.tree[2 * node + 1] / st.tree[2 * node +
    2];
54          }
55          else if (op == OP_MAX) {
56              if (st.tree[2 * node + 1] > st.tree[2 * node + 2]) {
57                  st.tree[node] = st.tree[2 * node + 1];
58              } else {
59                  st.tree[node] = st.tree[2 * node + 2];
60              }
61              printf("Maximo do intervalo [%d, %d]: %d\n", start, end, st
    .tree[node]);
62          }
63          else if (op == OP_MIN) {
64              if (st.tree[2 * node + 1] < st.tree[2 * node + 2]) {
65                  st.tree[node] = st.tree[2 * node + 1];
66              } else {
67                  st.tree[node] = st.tree[2 * node + 2];
68              }
69          }
70      }
71
72      if (op == OP_SOMA) {
73          printf("Soma do intervalo [%d, %d]: %d\n", start, end, st.tree[
    node]);
74      }
75      else if (op == OP_SUBTRACAO) {
76          printf("Subtracao do intervalo [%d, %d]: %d\n", start, end, st.
    tree[node]);
77      }
78      else if (op == OP_MULTIPLICACAO) {
79          printf("Multiplicacao do intervalo [%d, %d]: %d\n", start, end,
     st.tree[node]);
80      }
81      else if (op == OP_DIVISAO) {
82          printf("Divisao do intervalo [%d, %d]: %d\n", start, end, st.
    tree[node]);
83      }
84      else if (op == OP_MAX) {
85          printf("Maximo do intervalo [%d, %d]: %d\n", start, end, st.
    tree[node]);
86      }
87      else if (op == OP_MIN) {
88          printf("Minimo do intervalo [%d, %d]: %d\n", start, end, st.
    tree[node]);
89      }
90  }
91
92  SegmentTree buildSegmentTree(int arr[], int n, int op) {
93      SegmentTree st;
94      st.arr = arr;
95      st.size = n;
96
97      int treeSize = 4 * n;
98      st.tree = (int*)malloc(treeSize * sizeof(int));
```

```
99      buildSegmentTreeUtil(st, 0, 0, n - 1, op);
100
101     return st;
102 }
103
104 void defaultInput(){ //Mesmo vetor e operacao do artigo
105     printf("* Default input *\n\n");
106
107     int arr[] = {1, 3, -2, 8, -7};
108     int n = sizeof(arr) / sizeof(arr[0]);
109     int i;
110
111     printf("Vetor de entrada:\n");
112     for (i = 0; i < n; i++) {
113         printf("%d\t", arr[i]);
114     }
115
116     int op = OP_SOMA;
117     printf("\n\nOperacao: Soma");
118     printf("\n\n");
119
120     clock_t start = clock();
121     SegmentTree st = buildSegmentTree(arr, n, op);
122     clock_t end = clock();
123     double elapsed_time = ((double) (end - start)) / CLOCKS_PER_SEC;
124
125     free(st.tree);
126
127     printf("Tempo de execucao: %.6f segundos\n", elapsed_time);
128 }
129
130 void customInput(){ //Vetor e operacao a definir
131     int n;
132     printf("* Custom input *\n\n");
133     printf("Digite o tamanho do vetor de entrada: ");
134     scanf("%d", &n);
135
136     int arr[n];
137     int i;
138
139     for (i = 0; i < n; i++) {
140         printf("Elemento %d: ", i + 1);
141         scanf("%d", &arr[i]);
142     }
143
144     printf("\nVetor de entrada:\n");
145     for (i = 0; i < n; i++) {
146         printf("%d\t", arr[i]);
147     }
148
149     int op;
150     while (1) {
151         printf("\n\nDigite a operacao desejada: \n");
152         printf("0 - Soma\n");
153         printf("1 - Subtracao\n");
154         printf("2 - Multiplicacao\n");
```

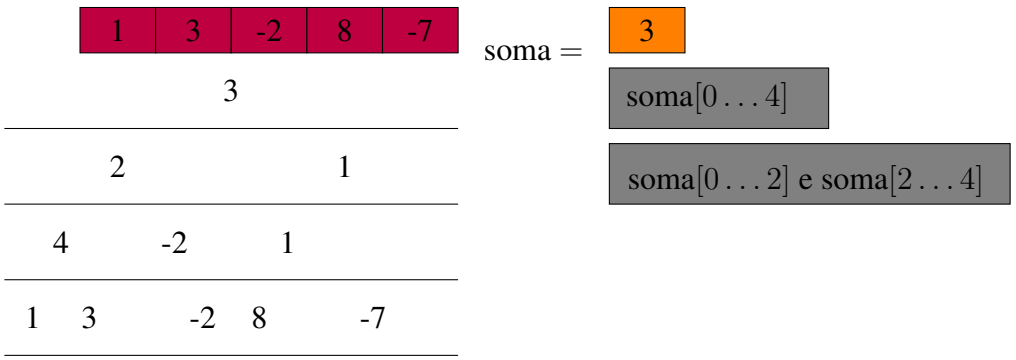
```
155         printf("3 - Divisao\n");
156         printf("4 - Maximo\n");
157         printf("5 - Minimo\n");
158         printf("Opcao: ");
159         scanf("%d", &op);
160         if (op != 0 && op != 1 && op != 2 && op != 3 && op != 4 && op
    != 5) {
161             printf("\nOpcao invalida! Tente novamente.");
162             continue;
163         }
164         else {
165             break;
166         }
167     }
168 
169     printf("\n");
170 
171     clock_t start = clock();
172     SegmentTree st = buildSegmentTree(arr, n, op);
173     clock_t end = clock();
174     double elapsed_time = ((double) (end - start)) / CLOCKS_PER_SEC;
175 
176     free(st.tree);
177 
178     printf("Tempo de execucao: %.6f segundos\n", elapsed_time);
179 }
180 
181 int main() {
182     //defaultInput();
183     customInput();
184     return 0;
185 }
```

### 4.2. Documentação

A documentação de cada função do código:

1. **int getMid(int start, int end)**: Esta função calcula o índice médio entre dois índices start e end da árvore de segmentos.
2. **int queryOp(SegmentTree st, int node, int start, int end, int left, int right)**: Esta função realiza uma operação de consulta na árvore de segmentos. Ela retorna o resultado da operação especificada (OP_SOMA, OP_SUBTRACAO, OP_MULTIPLICACAO, OP_DIVISAO, OP_MAX ou OP_MIN) para um determinado intervalo definido pelos índices left e right .
3. **void buildSegmentTreeUtil(SegmentTree st, int node, int start, int end, int op)**: Esta função auxiliar é responsável por construir a árvore de segmentos. Ela é chamada recursivamente para dividir o intervalo de elementos e calcular os resultados das operações para os nós da árvore.
4. **SegmentTree buildSegmentTree(int arr[], int n, int op)**: Esta função constrói a árvore de segmentos com base em um vetor de entrada arr de tamanho n. Ela chama a função buildSegmentTreeUtil para preencher os nós da árvore com os resultados das operações.
5. **void defaultInput()**: Esta função define um vetor de entrada padrão e uma operação padrão (OP_SOMA). Ela chama a função buildSegmentTree para construir a árvore de segmentos e imprime os resultados das operações no intervalo definido.
6. **void customInput()**: Esta função permite que o usuário defina um vetor de entrada e a operação desejada. Ela lê o tamanho do vetor e os elementos do usuário e chama a função **buildSegmentTree** para construir a árvore de segmentos com base nesses valores. Em seguida, ela imprime os resultados das operações no intervalo definido.
7. **int main()**: Esta função principal chama a função **customInput** para permitir ao usuário fornecer um vetor de entrada personalizado e a operação desejada. Alternativamente, a função **defaultInput** pode ser chamada para usar um vetor de entrada padrão e a operação padrão.

### 4.3. Teste de Mesa




## 5. Contexto e Aplicação

A estrutura de dados Segment Tree é usada para resolver problemas que exigem a identificação de uma resposta para uma consulta em um intervalo de um arranjo. Existem várias versões avançadas da Árvore de Segmentos, incluindo aquelas capazes de lidar

com consultas mais complexas, como aquelas que envolvem a identificação do mínimo ou do máximo em um intervalo [HackerEarth 2023].

Embora a estrutura de dados Segment Tree possa ser usada para resolver uma ampla variedade de problemas, ela não é a solução ideal para todos os problemas. Por exemplo, a Árvore de Segmentos não é a estrutura de dados ideal para problemas que envolvem consultas em intervalos que se sobrepõem [HackerEarth 2023].

## 5.1. Aplicações

Um dos primeiros métodos amplamente utilizados na área da Geometria Computacional é a árvore de intervalos/segmentos [Six and Wood 1982]. Essa estrutura tem se mostrado eficaz na resolução de diversos problemas geométricos, tais como a remoção eficiente de ruídos em áreas com baixa textura, bem como o realce das bordas dos objetos [Mei et al. 2013] [Azali et al. 2022].

No contexto da Geometria Computacional, a Segment Tree também esta associada ao mapeamento denso de disparidades, sendo útil na determinação de profundidades de cena [Scharstein et al. 2001], como ilustrado na Figura 7 e na Figura 8.

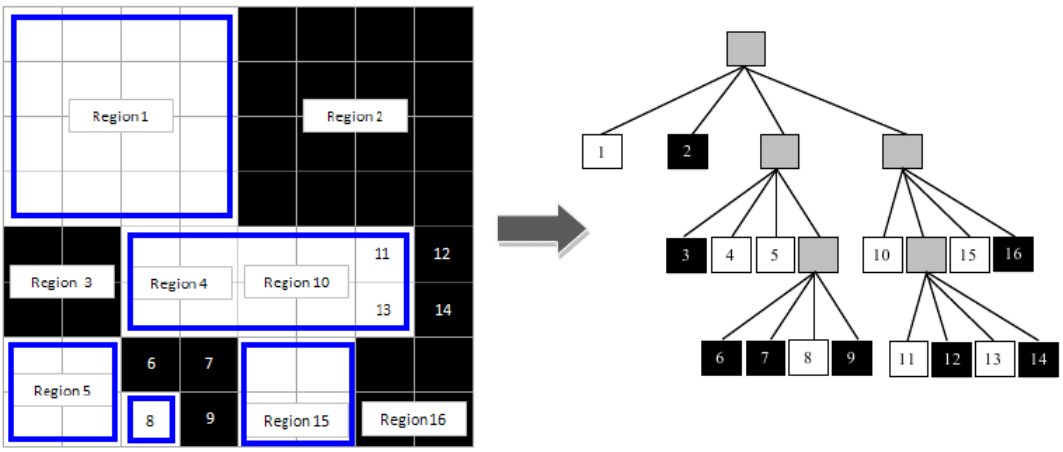



**Figura 7. Uma quadtree, um tipo de Segment Tree, usada para mapeamento de imagens.**

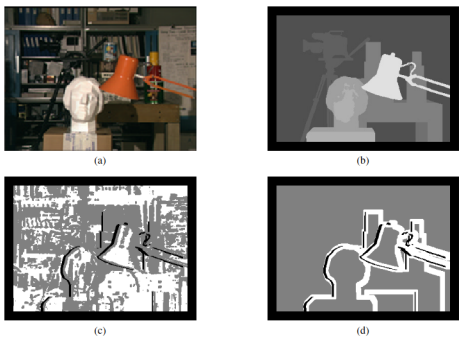


**Figura 8. Exemplo de uma imagem segmentada: (a) imagem original, (b) disparidades verdadeiras, (c) regiões sem textura (branco) e regiões ocultas (preto), (d) regiões com descontinuidade de profundidade (branco) e regiões ocultas (preto).**

Além disso, a Segment Tree também encontra aplicações em problemas de redes, como roteamento de pacotes e endereçamento IP. No artigo mencionado [Su 2000], essa estrutura é utilizada para armazenar os campos de endereço de destino dos pacotes de rede. A Segment Tree é construída com base nas informações dos prefixos de endereços IP, onde cada prefixo é representado por um segmento com dois pontos finais. Os nós internos da árvore correspondem a intervalos que são uniões dos intervalos elementares associados às folhas. Essa abordagem garante eficiência tanto na classificação quanto no armazenamento dos filtros aplicados a cada intervalo de endereço IP.

A aplicação dessa estrutura pode ser observada na Figura 9 do artigo, onde é possível visualizar como os segmentos são organizados e como as consultas de roteamento são realizadas de forma eficiente.

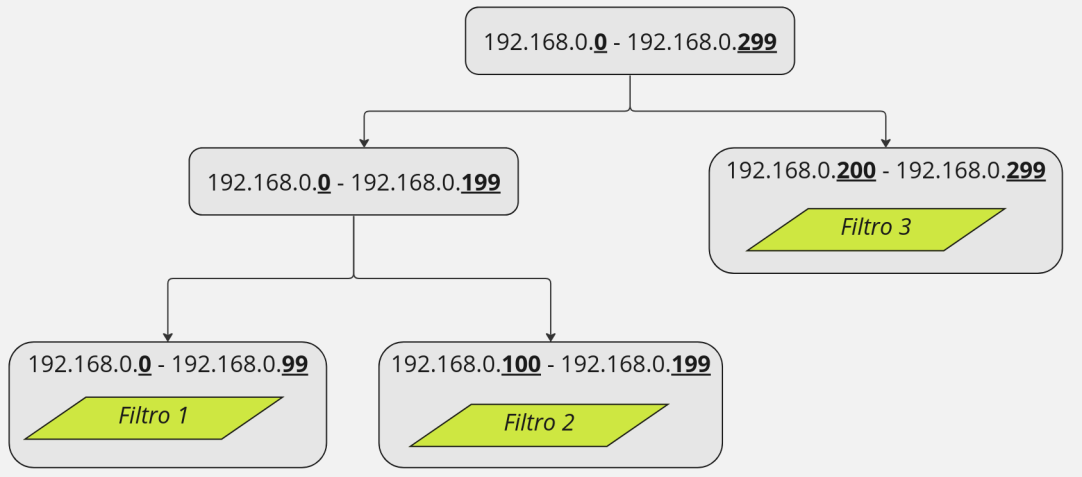



**Figura 9. Exemplo de aplicação de filtro por segmentos de Ips utilizando Segment Tree, nesse caso por sufixo.**

Essa capacidade da Árvore de Segmentos de lidar com problemas de redes demonstra sua versatilidade e adaptabilidade a diferentes domínios e necessidades. Sua aplicação nesse contexto específico contribui para melhorar o desempenho e a escalabilidade dos sistemas de roteamento, resultando em uma comunicação de rede mais eficiente e confiável.

## 6. Conclusão

A árvore de segmentos é uma estrutura de dados poderosa que permite realizar consultas eficientes em intervalos e atualizações em um vetor. Ela é especialmente útil para problemas que envolvem encontrar o mínimo, máximo, soma ou qualquer outra operação agregada em um intervalo específico de elementos.

Além disso, a árvore de segmentos é uma estrutura flexível que pode ser adaptada para resolver problemas mais complexos, adicionando operações personalizadas e escalando para dimensões maiores. Sendo ela uma ferramenta valiosa para lidar com consultas e atualizações eficientes em intervalos de elementos. Sua aplicação pode trazer melhorias significativas no desempenho e na eficiência dos algoritmos, permitindo soluções mais rápidas e precisas para uma ampla gama de problemas computacionais.

Criamos um algoritmo dinâmico que recebe um vetor do usuário, seleciona qual consulta deve ser executada nos segmentos gerados pela estrutura de dados. Como saída, apresentamos a consulta para todos segmentos do vetor. O algoritmo além de executar todas as consultas implementadas, mostra o tempo de execução de cada consulta. Medir o tempo de execução usando a função clock() em C/C++ pode ajudar a validar a eficiência teórica da implementação.

## Referências


[Azali et al. 2022] Azali, M., Hamzah, R., and Noh, Z. (2022). Disparity map algorithm using census transform and hierarchical segment-tree from stereo image. In *Engineering Technology International Conference (ETIC 2022)*, volume 2022, pages 244–249.

[Canuto 2021] Canuto, F. (2021). Estruturas de dados: Árvore de segmentos. *Blog Código Fluente*.

[Cormen et al. 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

[CP-Algorithms 2023] CP-Algorithms (Accessed 2023). Segment tree.

[de Berg et al. 2000] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. C., de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. C. (2000). More geometric data structures: Windowing. *Computational Geometry: algorithms and applications*, pages 211–233.

[Fischer 2011] Fischer, J. (2011). Range minimum queries and applications. *ACM SIGACT News*, 42(1):37–49.

[Gupta 2020] Gupta, A. (2020). Segment trees: A data structure for range queries. *Medium*.

[HackerEarth 2023] HackerEarth (Accessed 2023). Segment trees - hackerearth.

[Mei et al. 2013] Mei, X., Sun, X., Dong, W., Wang, H., and Zhang, X. (2013). Segment-tree based cost aggregation for stereo matching. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 313–320.

[Scharstein et al. 2001] Scharstein, D., Szeliski, R., and Zabih, R. (2001). A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, pages 131–140.

[Six and Wood 1982] Six and Wood (1982). Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers*, C-31(3):181–187.

[Su 2000] Su, C.-F. (2000). High-speed packet classification using segment tree. In *Globecom '00 - IEEE. Global Telecommunications Conference. Conference Record (Cat. No.00CH37137)*, volume 1, pages 582–586 vol.1.