

FGV EMap
João Pedro Jerônimo

Projeto e Análise de Algoritmos
Revisão para A1

Rio de Janeiro
2025

Conteúdo

1	Notação Assintótica	3
2	Recorrência	6
2.1	Método da substituição	7
2.2	Método da árvore de recursão	7
2.3	Método da Recorrência	8
2.4	Método mestre	9
3	Algoritmos de busca	11
3.1	Busca em um vetor ordenado	12
3.2	Árvores	12
3.3	Árvores Binárias de Busca	16
4	Tabela Hash	18
4.1	Um problema	20
4.1.1	Primeira abordagem: Endereçamento Direto	20
4.1.2	Segunda abordagem: Lista Encadeada	20
4.2	Definição	21
4.3	Soluções para colisão	21
4.3.1	Tabela hash com encadeamento	21
4.3.2	Hash uniforme simples	23

Notação Assintótica

Já vemos desde o começo do curso que um algoritmo é um conjunto de instruções feitas com o objetivo de resolver um determinado problema. Porém, certos problemas apresentam diversos tipos de solução!



Figura 1: Caminhos problema-solução

Então como podemos comparar eles? Como eu sei qual que é o melhor caminho até minha solução? De primeira a gente pode pensar: “Vê quanto tempo executou!”, mas isso gera um problema... Se eu executo um algoritmo em um computador de hoje em dia e o mesmo algoritmo em um computador de 1980, com certeza eles vão levar tempos diferentes para executar, correto? Isso pode afetar na medição que eu estou fazendo do meu algoritmo!

Então o que fazer? O mais comum é analisarmos o quão bem meu algoritmo consegue funcionar de acordo com o quão grande meu problema fica!

Definição 1.1 (Função de Complexidade): A complexidade de um algoritmo é a função $T : U^+ \rightarrow \mathbb{R}$ que leva do espaço do tamanho das entradas do problema até a quantidade de instruções feitas para realizá-lo

Exemplo:

```
1 def sum(numbers: list):
2     result = 0
3     for number in numbers:
4         result += 1
5     return result
```

Eu tenho que, para esse algoritmo, $T(n) = n$, pois, quanto maior é a quantidade de números na minha lista, maior é o tempo que a função vai ficar executando

Só que achar qual é essa função exatamente pode ser muito trabalhoso, além de que muitas funções são parecidas e podem gerar uma dificuldade na hora da análise. Então o que fazemos?

Faz sentido dizermos que, se a partir de algum ponto uma função T_1 cresce mais do que T_2 , então o algoritmo T_1 acaba sendo pior, então criamos a definição:

Definição 1.2 (Big O): Dizemos que $T(n) = O(f(n))$ se $\exists c, n_0 > 0$ tais que

$$T(n) \leq cf(n), \quad \forall n \geq n_0 \quad (1)$$

Ou seja, dado algum c e n_0 qualquer, depois de n_0 , $f(n)$ SEMPRE cresce mais que $T(n)$

Definição 1.3 (Big Ω): Dizemos que $T(n) = \Omega(f(n))$ se $\exists c, n_0 > 0$ tais que

$$T(n) \geq cf(n), \quad \forall n \geq n_0 \quad (2)$$

Definição 1.4 (Big Θ): Dizemos que $T(n) = \Theta(f(n))$ se $T(n) = \Omega(f(n))$ e $T(n) = O(f(n))$

Recorrência

Alguns algoritmos são fáceis de terem suas complexidades calculadas, porém, na programação, existem casos onde uma função utiliza ela mesma dentro de sua chamada, as temidas **recursões**

```
1 def fatorial(n):  
2     if n == 1:  
3         return 1  
4     return n*fatorial(n-1)
```

py

Então nós temos um $T(n)$ que chama $T(n-1)$, o que fazemos? Temos 4 métodos de resolver esse problema

- **Método da substituição**
- **Método da árvore de recursão**
- **Método da iteração**
- **Método mestre**

2.1 Método da substituição

Vamos provar por **indução** que $T(n)$ é O de uma função **pressuposta**. Só posso usar quando eu tenho uma hipótese da solução. Precisamos provar exatamente a hipótese. Pode ser usado para limites superiores e inferiores

Exemplo:

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{se } n > 1 \end{cases} \quad (3)$$

Vamos pressupor que $T(n) = O(n^2)$. Queremos então provar $T(n) \leq cn^2$.

Caso base: $n = 1 \Rightarrow T(1) = 1 \leq cn^2$

Passo Indutivo: Vamos supor que vale para $\frac{n}{2}$, e ver se vale para n . Então temos:

$$T\left(\frac{n}{2}\right) \leq c \frac{n^2}{4} \quad (4)$$

Vamos testar para $T(n)$ então

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \Rightarrow T(n) \leq 2c \frac{n^2}{4} + n \\ &\Leftrightarrow T(n) \leq \frac{cn^2}{2} + n \\ &\Leftrightarrow \frac{cn^2}{2} + n \leq cn^2 \\ &\Leftrightarrow 2n \leq 2cn^2 - cn^2 \\ &\Leftrightarrow \frac{n}{2} \leq c \end{aligned} \quad (5)$$

Ou seja, conseguimos escolher um c e um n_0 de forma que $\forall n \geq n_0, T(n) \leq cn^2$, logo, $T(n) = O(n^2)$

2.2 Método da árvore de recursão

O método da árvore de recursão consiste em construir uma árvore definindo em cada nível os sub-problemas gerados pela iteração do nível anterior. A forma geral é encontrada ao somar o custo de todos os nós

- Cada nó representa um subproblema.
- Os filhos de cada nó representam as suas chamadas recursivas.
- O valor do nó representa o custo computacional do respectivo problema.

Esse método é útil para analisar algoritmos de divisão e conquista.

Exemplo:

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + n & \text{se } n > 1 \end{cases} \quad (6)$$

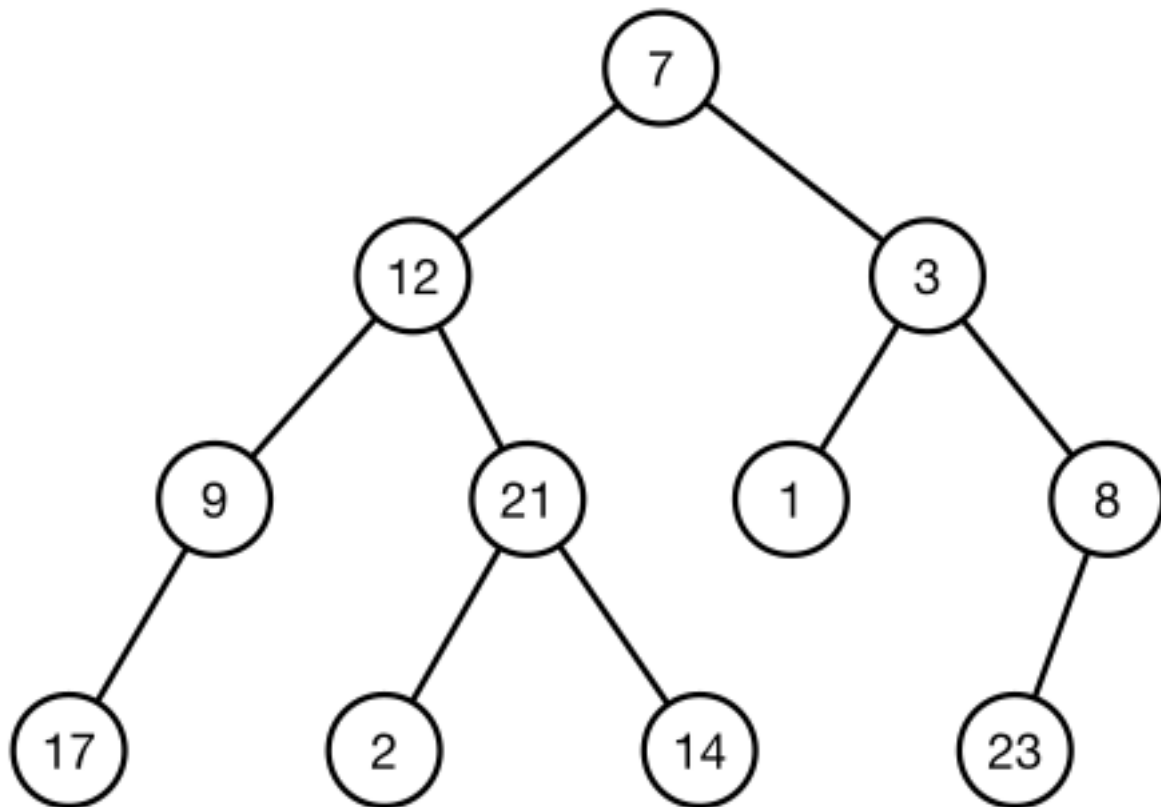


Figura 2: Árvore de $T(n)$

Temos então que:

$$T(n) = \sum_{k=0}^{\log(n)} 2^k \frac{n}{2^k} = n \log(n) + n \quad (7)$$

Então temos que $T(n) = O(n \log(n))$

2.3 Método da Recorrência

O método da iteração consiste em expandir a relação de recorrência até o n -ésimo termo, de forma que seja possível compreender a sua forma geral

Exemplo:

$$T(n) = \begin{cases} \theta(1) & \text{se } n = 1 \\ 2T(n-1) + n & \text{se } n > 1 \end{cases} \quad (8)$$

Expandindo, temos:

$$\begin{aligned}
T(n) &= 2T(n-1) + n \\
T(n) &= 2(2T(n-2) + n) + n \\
&\vdots \\
T(n) &= 2^k T(n-k) + (2^k - 1)n - \sum_{j=1}^{k-1} 2^j j
\end{aligned} \tag{9}$$

Para chegar na última iteração, temos que $k = n - 1$

$$T(n) = 2^{n-1} + (2^{n-1} - 1)n - \sum_{j=1}^{n-2} 2^j j \tag{10}$$

Temos que: $\sum_{j=1}^{n-2} 2^j j = \frac{1}{2}(2^n n - 3 \cdot 2^n + 4)$, então podemos fazer:

$$\begin{aligned}
T(n) &= 2^{n-1} + 2^{n-1}n - n - 2^{n-1}n + 3 \cdot 2^{n-1} - 2 \\
&\Leftrightarrow T(n) = 2^{n-1} - n + 3 \cdot 2^{n-1} - 2 \\
&\Leftrightarrow T(n) = 4 \cdot 2^{n-1} - n - 2 = 2^{n+1} - n - 2 \\
&\Leftrightarrow T(n) = \Theta(2^n)
\end{aligned} \tag{11}$$

2.4 Método mestre

Esse teorema é uma decoreba. Ele te dá um caso geral e vários casos de resultado dependendo dos valores na estrutura de $T(n)$

Teorema 2.4.1 (Teorema Mestre): Dada uma recorrência da forma

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{12}$$

Considerando $a \geq 1$, $b > 1$ e $f(n)$ assintoticamente positiva

- Se $f(n) = O(n^{\log_b(a) - \varepsilon})$ para alguma constante $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b(a)})$
- Se $f(n) = \Theta(n^{\log_b(a)})$, então $T(n) = \Theta(f(n) \log(n))$
- Se $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$ para alguma constante $\varepsilon > 0$ e atender a uma condição de regularidade $af\left(\frac{n}{b}\right) \leq cf(n)$ para alguma constante positiva $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$

Exemplo (Primeiro caso):

$$T(n) = 9T\left(\frac{n}{3}\right) + n \tag{13}$$

Então $a = 9$, $b = 3$ e $f(n) = n$, calculamos então:

$$n^{\log_b(a)} = n^{\log_3(9)} = n^2 \tag{14}$$

Ou seja, conseguimos escolher $\varepsilon = 1$ de forma que

$$f(n) = O(n^{2-1}) = O(n) \tag{15}$$

Ou seja, $T(n) = \Theta(n^2)$

Exemplo (Segundo caso):

$$T(n) = T\left(\frac{2n}{3}\right) + 1 \quad (16)$$

Então $a = 1$, $b = \frac{3}{2}$ e $f(n) = 1$, calculamos então:

$$n^{\log_b(a)} = n^{\log_{\frac{3}{2}}(1)} = 1 \quad (17)$$

Ou seja, $f(n) = \Theta(n^{\log_b(a)})$, e isso quer dizer que $T(n) = \Theta\left(n^{\log_{\frac{3}{2}}(1) \log(n)}\right) = \Theta(\log(n))$

Exemplo (Terceiro caso):

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log(n) \quad (18)$$

Então $a = 3$, $b = 4$ e $f(n) = n \log(n)$, calculamos então:

$$n^{\log_b(a)} = n^{\log_4 3} \approx n^{0.79} \quad (19)$$

Temos então que $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ para um $\varepsilon \approx 0.2$. Então agora vamos analisar a condição de regularidade:

$$\begin{aligned} af\left(\frac{n}{b}\right) &\leq cf(n) \\ 3\left(\frac{n}{4} \log\left(\frac{n}{4}\right)\right) &\leq cn \log(n) \Rightarrow c \geq \frac{3}{4} \end{aligned} \quad (20)$$

Ou seja, $T(n) = \Theta(n \log(n))$

Exemplo (Exemplo que não funciona):

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log(n) \quad (21)$$

Para agilizar, isso se encaixa no caso em que $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$. Vamos então checar a regularidade:

$$\begin{aligned} af\left(\frac{n}{b}\right) &\leq cf(n) \\ 2 \frac{n}{2} \log\left(\frac{n}{2}\right) &\leq cn \log(n) \\ \Leftrightarrow c &\geq 1 - \frac{1}{\log(n)} \end{aligned} \quad (22)$$

Impossível! Já que $c < 1$

Esse método pode ser simplificado para uma categoria específica de funções

Teorema 2.4.2 (Teorema mestre simplificado): Dada uma recorrência do tipo:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k) \quad (23)$$

Considerando $a \geq 1$, $b > 1$ e $k \geq 0$:

- Se $a > b^k$, então $T(n) = \Theta(n^{\log_b a})$
- Se $a = b^k$, então $T(n) = \Theta(n^k \log n)$
- Se $a < b^k$, então $T(n) = \Theta(n^k)$

Algoritmos de busca

Vamos apresentar algoritmos de busca e suas complexidades

3.1 Busca em um vetor ordenado

Dado um vetor ordenado de inteiros:

```
1 int* v = { 2, 5, 9, 18, 23, 27, 32, 33, 37, 41, 43, 45 };
```

C++

Queremos escrever um algoritmo que recebe o vetor v , um número x e retorna o índice de x no vetor v se $x \in v$. Temos dois algoritmos principais para esse problema

BUSCA LINEAR

C++

```
1 int linear_search(const int v[], int size, int x) {  
2     for (int i = 0; i < size; i++) {  
3         if (v[i] == x) {  
4             return i;  
5         }  
6     }  
7     return -1;  
8 }
```

No pior caso, esse algoritmo tem complexidade $\Theta(n^2)$

Porém, se considerarmos uma lista ordenada, podemos fazer algo mais inteligente. Comparamos do meio do vetor e dependendo se o valor atual é maior ou menor comparado ao avaliado, então eu ignoro uma parte do vetor. O algoritmo consiste em avaliar se o elemento buscado (x) é o elemento no meio do vetor (m), e caso não seja executar a mesma operação sucessivamente para a metade superior (caso $x > m$) ou inferior (caso $x < m$).

BUSCA BINÁRIA

C++

```
1 int search(int v[], int leftInx, int rightInx, int x) {  
2     int midInx = (leftInx + rightInx) / 2;  
3     int midValue = v[midInx];  
4     if (midValue == x) {  
5         return midInx;  
6     }  
7     if (leftInx >= rightInx) {  
8         return -1;  
9     }  
10    if (x > midValue) {  
11        return search(v, midInx + 1, rightInx, x);  
12    } else {  
13        return search(v, leftInx, midInx - 1, x);  
14    }  
15 }
```

Podemos escrever a complexidade da função como:

$$T(n) = T\left(\frac{n}{2}\right) + c \quad (24)$$

Fazendo os cálculos, obtemos que $T(n) = \Theta(\log(n))$

3.2 Árvores

Uma árvore binária consiste em uma estrutura de dados capaz de armazenar um conjunto de nós.

- Todo nó possui uma chave

- Opcionalmente um valor (dependendo da aplicação).
- Cada nó possui referências para dois filhos
- Sub-árvores da direita e da esquerda.
- Toda sub-árvore também é uma árvore.

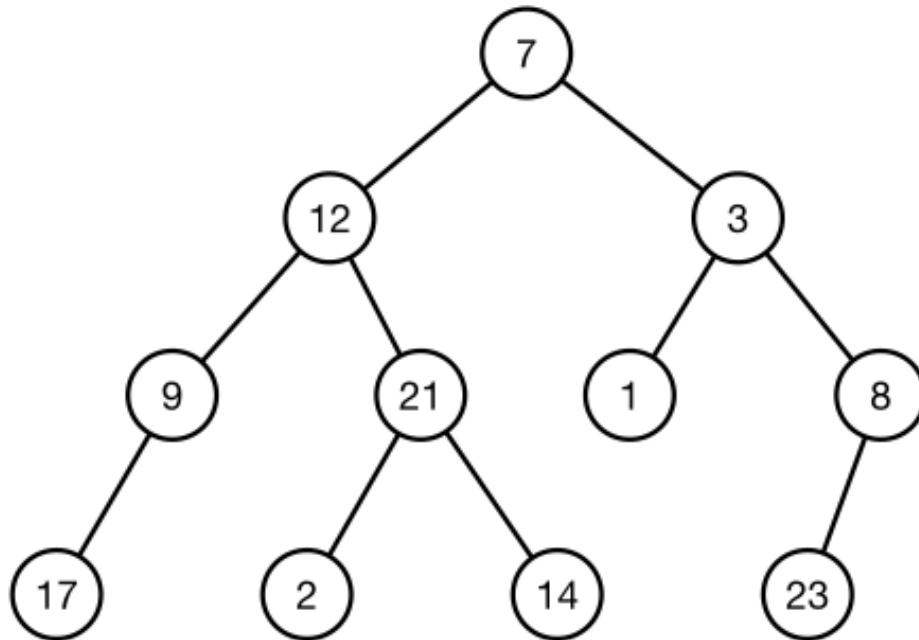


Figura 3: Exemplo de árvore

Um nó sem pai é uma **raiz**, enquanto um nó sem filhos é um nó **folha**

Definição 3.2.1 (Altura do nó): Distância entre um nó e a folha mais afastada. A altura de uma árvore é a altura do nó raiz

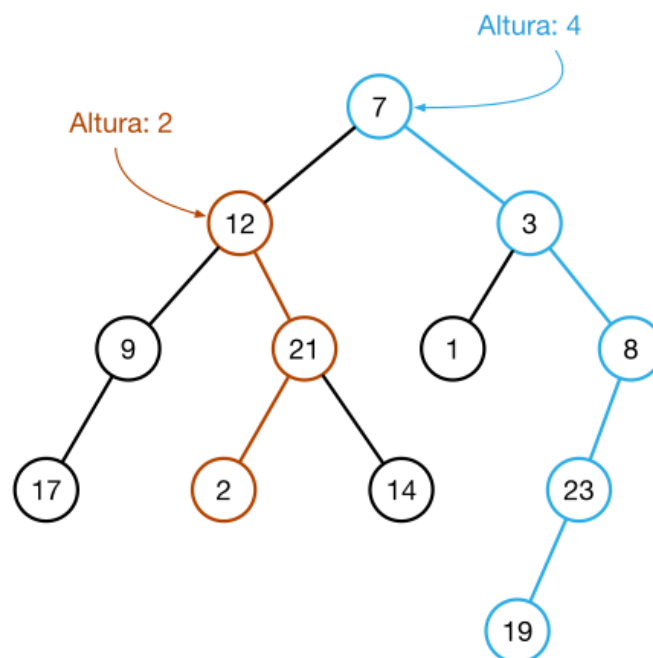


Figura 4: Exemplificação de altura

Teorema 3.2.1: Dada uma árvore de altura h , a quantidade máxima de nós n_{\max} e mínima n_{\min} são:

$$\begin{aligned} n_{\min} &= h + 1 \\ n_{\max} &= 2^{h+1} - 1 \end{aligned} \quad (25)$$

Definição 3.2.2: Uma árvore está **balanceada** quando a altura das subárvores de um nó apresentem uma diferença de, no máximo, 1

Teorema 3.2.2: Dada uma árvore com n nós e balanceada, a sua altura h será, no máximo:

$$h = \log(n) \quad (26)$$

Para códigos posteriores, considere a seguinte estrutura:

```
1  class Node {
2      public:
3          Node(int key, char data)
4              : m_key(key)
5              , m_data(data)
6              , m_leftNode(nullptr)
7              , m_rightNode(nullptr)
8              , m_parentNode(nullptr) {}
9          Node & leftNode() const { return * m_leftNode; }
10         void setLeftNode(Node * node) { m_leftNode = node; }
11
12         Node & rightNode() const { return * m_rightNode; }
13         void setRightNode(Node * node) { m_rightNode = node; }
14
15         Node & parentNode() const { return * m_parentNode; }
16         void setParentNode(Node * node) { m_parentNode = node; }
17
18     private:
19         int m_key;
20         char m_data;
21         Node * m_leftNode;
22         Node * m_rightNode;
23         Node * m_parentNode;
24 };
```

Temos alguns tipos de problemas para trabalhar em cima das árvores e suas soluções:

Problema: Dada uma árvore binária A com n nós encontre a sua altura

```
1  int nodeHeight(Node * node) {
2      if (node == nullptr) {
3          return -1;
```

```


4    }
5
6    int leftHeight = nodeHeight(node->leftNode());
7    int rightHeight = nodeHeight(node->rightNode());
8
9    if (leftHeight < rightHeight) {
10       return rightHeight + 1;
11    } else {
12       return leftHeight + 1;
13    }
14 }

```

A complexidade dessa solução é $\Theta(n)$

Problema: Dada uma árvore binária A imprima a chave de todos os nós através da busca em profundidade. Desenvolva o algoritmo para os 3 casos: Em ordem, pré-ordem, pós-ordem

EM ORDEM


 C++

```

1 void printTreeDFSInorder(class Node * node) {
2     if (node == nullptr) {
3         return;
4     }
5     printTreeDFSInorder(node->leftNode());
6     cout << node->key() << " ";
7     printTreeDFSInorder(node->rightNode());
8 }

```

PRÉ-ORDEM


 C++

```

1 void printTreeDFSPreorder(class Node * node) {
2     if (node == nullptr) {
3         return;
4     }
5     cout << node->key() << " ";
6     printTreeDFSPreorder(node->leftNode());
7     printTreeDFSPreorder(node->rightNode());
8 }

```

PÓS-ORDEM

 C++

```

1 void printTreeDFSPostorder(class Node * node) {
2     if (node == nullptr) {
3         return;
4     }
5     printTreeDFSPostorder(node->leftNode());
6     printTreeDFSPostorder(node->rightNode());
7     cout << node->key() << " ";
8 }


```

Problema: dada uma árvore binária A imprima a chave de todos os nós através da busca em largura.

```

1 void printTreeBFSWithQueue(Node * root) {
2     if (root == nullptr) {
3         return;
4     }

```

 C++

```

5  queue<Node*> queue;
6  queue.push(root);
7  while (!queue.empty()) {
8      Node * node = queue.front();
9      cout << node->key() << " ";
10     queue.pop();
11     Node * childNode = node->leftNode();
12     if (childNode) {
13         queue.push(childNode);
14     }
15     childNode = node->rightNode();
16     if (childNode) {
17         queue.push(childNode);
18     }
19 }
20 }

```

3.3 Árvores Binárias de Busca

Definição 3.3.1 (Árvores de busca): São uma classe específica de árvores que seguem algumas características:

- A chave de cada nó é maior ou igual a chave da raiz da sub-árvore esquerda.
- A chave de cada nó é menor ou igual a chave da raiz da sub-árvore direita

$\text{left.key} \leq \text{key} \leq \text{right.key}$

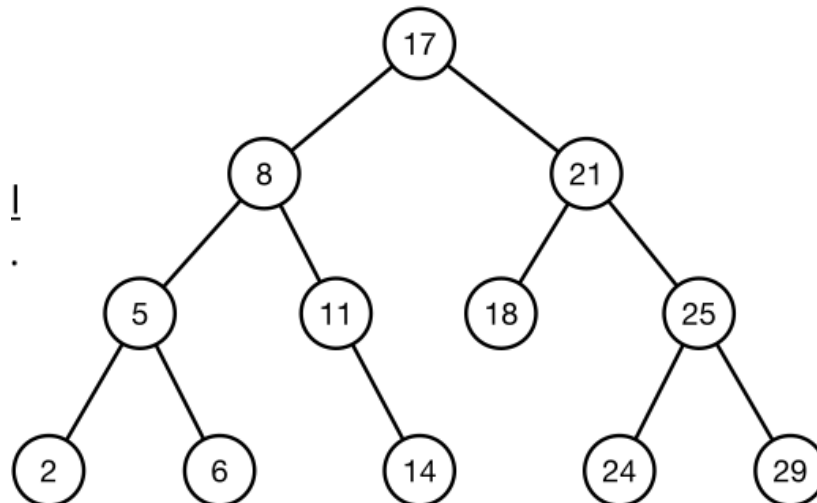


Figura 5: Exemplo de árvore binária

Então queremos utilizar essa árvore para poder procurar valores. Na verdade ela é bem parecida com o caso de aplicar uma busca binária em um vetor ordenado.

Problema: dada uma árvore binária de busca A com altura h encontre o nó cuja chave seja k .

BUSCA EM ÁRVORE BINÁRIA (RECURSÃO)

C++

```

1  Node * binaryTreeSearchRecursive(Node * node, int key) {
2      if (node == nullptr || node->key() == key) {
3          return node;
4      }
5      if (node->key() > key) {

```




```

6     return binaryTreeSearchRecursive(node->leftNode(), key);
7 } else {
8     return binaryTreeSearchRecursive(node->rightNode(), key);
9 }
10 }

```

Esse algoritmo tem complexidade $\Theta(h)$

BUSCA EM ÁRVORE BINÁRIA (ITERATIVO)

 C++

```

1 Node * binaryTreeSearchIterative(Node * node, int key) {
2     while (node != nullptr && node->key() != key) {
3         if (node->key() > key) {
4             node = node->leftNode();
5         } else {
6             node = node->rightNode();
7         }
8     }
9     return node;
10 }

```

Tabela Hash

Famosos dicionários do python, nós a utilizamos para armazenar e pesquisar tuplas $\langle \text{chave}, \text{valor} \rangle$.

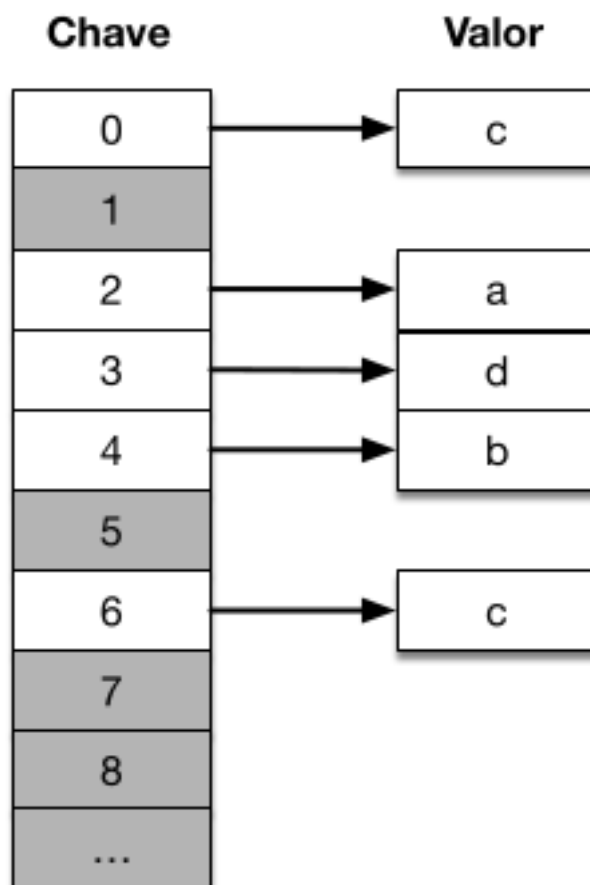


Figura 6: Desenho de tabela hash

Nós queremos criar funções $\Theta(1)$ para executar funções de **inserção**, **busca** e **remoção**. Todas as chaves contidas na tabela são **únicas**, já que elas identificam os valores unicamente.

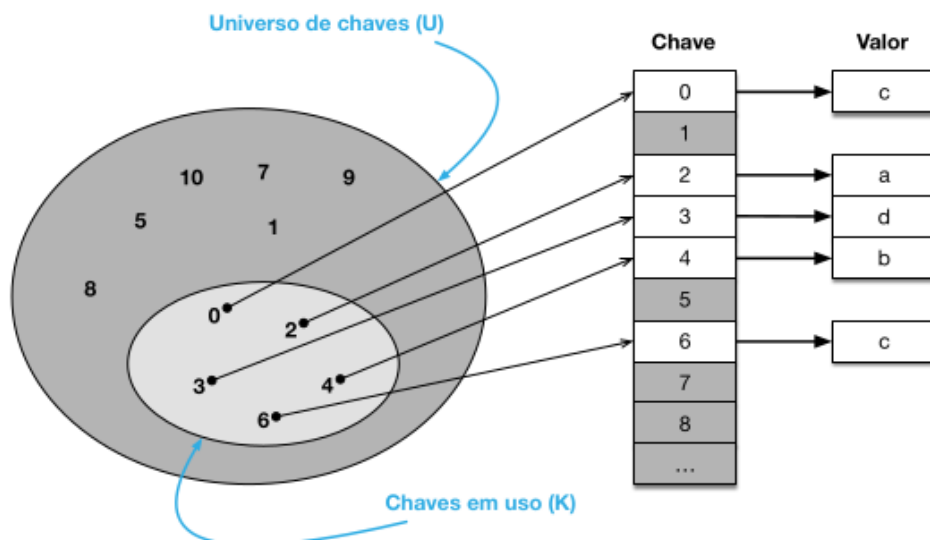


Figura 7: Estruturação da Hash Table

- **Universo de Chaves (U):** Conjunto de chaves possíveis
- **Chaves em Uso (K):** Conjunto de chaves utilizadas

Vamos idealizar um problema para motivar os nossos objetivos.

4.1 Um problema

Problema: Considere um programa que recebe eventos emitidos por veículos ao entrar em uma determinada região. Cada evento é composto por um inteiro representando o ID do veículo. O programa deve contar o número de vezes que cada veículo entrou na região. Ocasionalmente o programa recebe uma requisição para exibir o número de ocorrências de um dado veículo.

Mandatário: a contagem deve ser incremental, sem qualquer estratégia de cache. Uma requisição para exibir o resultado parcial da contagem deverá contemplar todos os eventos recebidos até o momento.

4.1.1 Primeira abordagem: Endereçamento Direto

```
1  int table[U];
2  for (int i = 0; i < U; i++) {
3      table[i] = 0;
4  }
5
6  // Função de incrementação em cada elemento
7  void add(int key) {
8      table[key]++;
9  }
10
11 // Função de busca
12 int search(int key) {
13     return table[key]
14 }
```

- $\text{add} = \Theta(1)$
- $\text{search} = \Theta(1)$

4.1.2 Segunda abordagem: Lista Encadeada

```
1  typedef struct LLNode CountNode;
2  struct LLNode {
3      int id;
4      int count;
5      CountNode * next;
6  };
7
8  void add(int key) {
9      CountNode * node = m_firstNode;
10     while (node != nullptr && node->id != key) {
11         node = node->next;
12     }
13     if (node != nullptr) {
14         node->count += 1;
15     } else {
16         CountNode * newNode = new CountNode;
17         newNode->id = key;
18         newNode->count = 1;
19         newNode->next = m_firstNode;
20         m_firstNode = newNode;
21     }
22 }
23 int search(int key) {
```

```

24 CountNode * node = m_firstNode;
25 while (node != nullptr && node->id != key) {
26     node = node->next;
27 }
28 return node != nullptr ? node->count : 0;
29 }

```

Infelizmente nessa abordagem nós não atingimos o objetivo principal de realizar as operações em $\Theta(1)$, já que a função de busca é $\Theta(n)$

4.2 Definição

Agora que entendemos toda a ideia da hash table, podemos fazer uma definição melhor para ela

Definição 4.2.1 (Hash Table): A **tabela hash** é uma estrutura de dados baseada em um vetor de M posições acessado através de endereçamento direto

Definição 4.2.2 (Função de Espalhamento/Hashing): É uma função que mapeia uma chave em um índice $[0, M - 1]$ do vetor. O resultado dessa função é comumente chamado de **hash**. O objetivo da função de espalhamento é reduzir o intervalo de índices de forma que M seja muito menor que o tamanho do universo U .

Exemplo:

```

1 hash(key) = key % M

```

Definição 4.2.3 (Colisão): É quando a função de espalhamento gera os mesmos hashes para chaves diferentes. Existem várias abordagens para resolver esse problema

Uma função de hash é considerada **boa** quando minimiza as colisões (Mas elas são inevitáveis)

4.3 Soluções para colisão

Vamos ver algumas abordagens para resolver o problema de colisão

4.3.1 Tabela hash com encadeamento

O problema de colisão é solucionado armazenando os elementos com o mesmo hash em uma lista encadeada

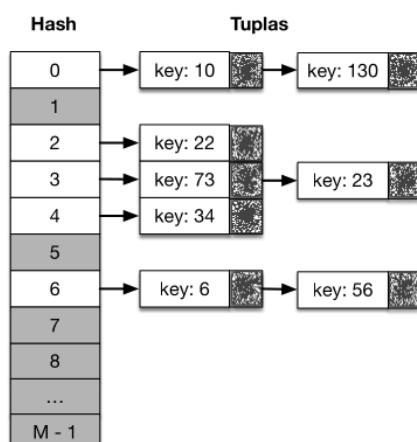
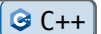


Figura 8: Tabela hash com encadeamento

EXEMPLO DE IMPLEMENTAÇÃO



```
1  typedef struct HashTableNode HTNode;
2  struct HashTableNode {
3      unsigned key;
4      int value;
5      HTNode * next;
6      HTNode * previous;
7  };
8
9  class HashTable {
10 public:
11     HashTable(int size)
12         : m_table(nullptr)
13         , m_size(size) {
14         m_table = new HTNode*[size];
15         for (int i=0; i < m_size; i++) { m_table[i] = nullptr; }
16     }
17     ~HashTable() {
18         for (int i=0; i < m_size; i++) {
19             HTNode * node = m_table[i];
20             while (node != nullptr) {
21                 HTNode * nextNode = node->next;
22                 delete node;
23                 node = nextNode;
24             }
25         }
26         delete[] m_table;
27     }
28     ...
29 private:
30     unsigned hash(unsigned key) const { return key % m_size; }
31     HTNode ** m_table;
32     int m_size;
33 };
34
35
36 void insert_or_update(unsigned key, int value) {
37     unsigned h = hash(key);
38     HTNode * node = m_table[h];
39     while (node != nullptr && node->key != key) {
40         node = node->next;
41     }
42     if (node == nullptr) {
43         node = new HTNode;
44         node->key = key;
45         node->next = m_table[h];
46         node->previous = nullptr;
47         HTNode * firstNode = m_table[h];
48         if (firstNode != nullptr) {
49             firstNode->previous = node;
50         }
51         m_table[h] = node;
```

```

52  }
53  node->value = value;
54  }
55
56
57  HTNode * search(unsigned key) {
58      unsigned h = hash(key);
59      HTNode * node = m_table[h];
60      while (node != nullptr && node->key != key) {
61          node = node->next;
62      }
63      return node;
64  }
65
66
67  bool remove(unsigned key) {
68      unsigned h = hash(key);
69      HTNode * node = m_table[h];
70      while (node != nullptr && node->key != key) {
71          node = node->next;
72      }
73      if (node == nullptr) {
74          return false;
75      }
76      HTNode * nextNode = node->next;
77      if (nextNode != nullptr) {
78          nextNode->previous = node->previous;
79      }
80      HTNode * previousNode = node->previous;
81      if (previousNode != nullptr) {
82          node->previous->next = node->next;
83      } else {
84          m_table[h] = node->next;
85      }
86      delete node;
87      return true;
88  }

```

O pior caso dessa implementação é quando todas as chaves são mapeadas em uma única posição

- **Inserção/Atualização:** $\Theta(n)$
- **Busca:** $\Theta(n)$
- **Remoção:** $\Theta(n)$

4.3.2 Hash uniforme simples

Cada chave possui a mesma probabilidade de ser mapeada em qualquer índice $[0, M)$. Essa é uma propriedade desejada para uma função de espalhamento a ser utilizada em uma tabela hash. Infelizmente esse resultado depende dos elementos a serem inseridos. Não sabemos a priori a distribuição das chaves ou mesmo a ordem em que serão inseridas. Heurísticas podem ser utilizadas para determinar uma função de espalhamento com bom desempenho

Alguns métodos mais comuns:

- **Simple**

- Se a chave for um número real entre $[0, 1)$
- $\text{hash}(\text{key}) = \lfloor \text{key} \cdot M \rfloor$
- **Método da divisão**
 - Se a chave for um número inteiro
 - $\text{hash}(\text{key}) = \text{key} \% M$
 - Costuma-se definir M como um número primo.
- **Método da multiplicação**
 - $\text{hash}(\text{key}) = \lfloor \text{key} \cdot A \% 1M \rfloor$
 - A é uma constante no intervalo $0 < A < 1$.

Observe que a chave pode assumir qualquer tipo suportado pela linguagem

Exemplo: `countries["BR"]`

A função de espalhamento é responsável por gerar um índice numérico com base no tipo de entrada

EXEMPLO DE HASH PARA STRINGS

C++

```
1 int hashStr(const char * value, int size) {
2     unsigned hash = 0;
3     for (int i=0; value[i] != '\0'; i++) {
4         hash = (hash * 256 + value[i]) % size;
5     }
6     return hash;
7 }
```