FGV EMAp

João Pedro Jerônimo

Projeto e Análise de Algoritmos

Revisão para A1

Rio de Janeiro

2025

Conteúdo

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 $T : U^+ \to \mathbb{R}$ que leva do espaço do tamanho das entradas do problema até a quantidade de instruções feitas para realizá-lo

Exemplo:

```py
def sum(numbers: list):
  result = 0
  for number in numbers:
    result += 1
  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

$$T(n) \leq cf(n), \ \forall n \geq n_0 \tag{1}$$

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

$$T(n) \geq cf(n), \ \forall n \geq n_0 \tag{2}$$

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**

```py
def fatorial(n):
  if n == 1:
    return 1
  return n*fatorial(n-1)
```

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:

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

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:

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

Vamos testar para $T(n)$ então

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

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:

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

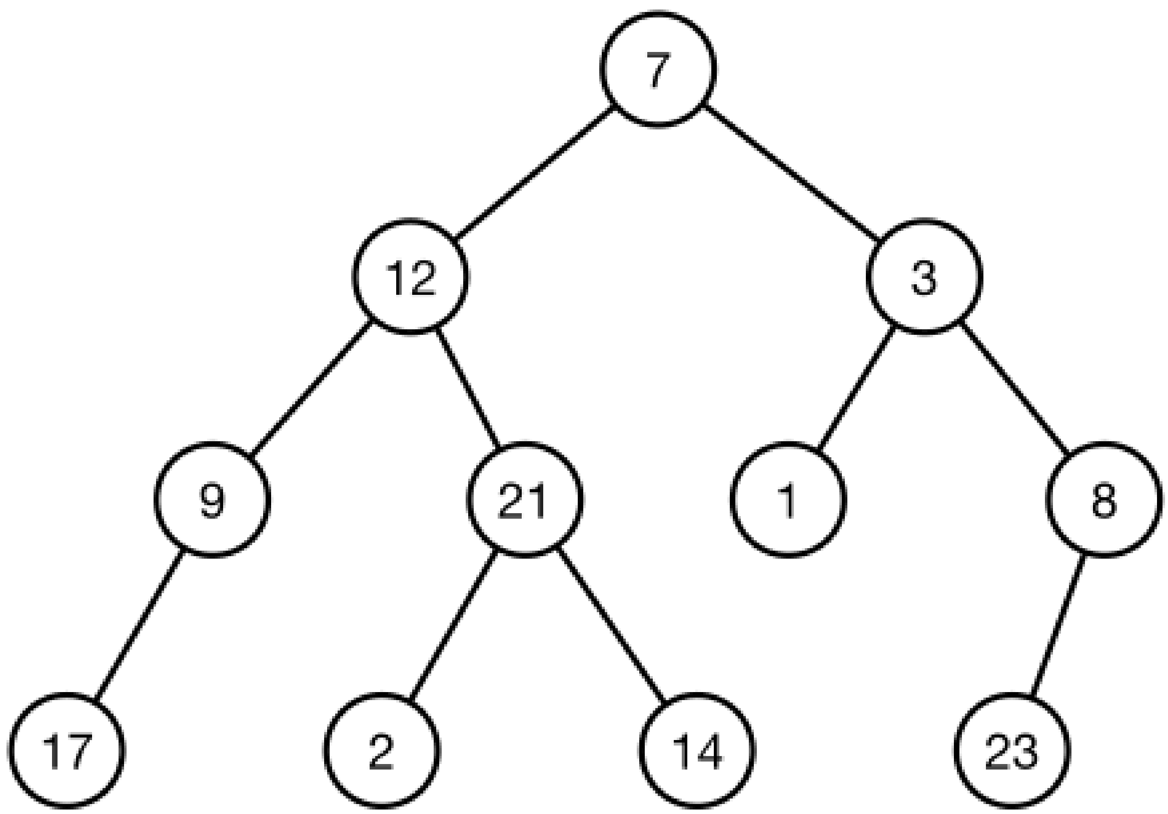


Figura 2: Árvore de $T(n)$

Temos então que:

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

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:

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

Expandindo, temos:

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

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 $f(n) = O\left(n^{\log_b(a) - \varepsilon}\right)$ para alguma constante $\varepsilon > 0$, então $\boldsymbol{T(n) = \Theta\left(n^{\log_b(a)}\right)}$
- Se $f(n) = \Theta\left(n^{\log_b(a)}\right)$, então $\boldsymbol{T(n) = \Theta(f(n)\log(n))}$
- Se $f(n) = \Omega\left(n^{\log_b(a) + \varepsilon}\right)$ 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 $\boldsymbol{T(n) = \Theta(f(n))}$

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):

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

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

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

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

Exemplo (Terceiro caso):

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

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

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

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

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

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

Exemplo (Exemplo que não funciona):

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

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

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

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:

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

Considerando $a \geq 1$, $b > 1$ e $k \geq 0$:
- Se $a > b^k$, então $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$
- Se $a = b^k$, então $\boldsymbol{T(n) = \Theta\left(n^k \log n\right)}$
- Se $a < b^k$, então $\boldsymbol{T(n) = \Theta\left(n^k\right)}$

Algoritmos de busca

Vamos apresentar algoritmos de busca e suas complexidades

3.1 Busca em um vetor ordenado

Dado um vetor ordenado de inteiros:

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

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 algorimtos principais para esse problema

BUSCA LINEAR

```cpp
int linear_search(const int v[], int size, int x) {
  for (int i = 0; i < size; i++) {
    if (v[i] == x) {
      return i;
    }
  }
  return -1;
}
```

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

```cpp
int search(int v[], int leftInx, int rightInx, int x) {
  int midInx = (leftInx + rightInx) / 2;
  int midValue = v[midInx];
  if (midValue == x) {
    return midInx;
  }
  if (leftInx >= rightInx) {
    return -1;
  }
  if (x > midValue) {
    return search(v, midInx + 1, rightInx, x);
  } else {
    return search(v, leftInx, midInx - 1, x);
  }
}
```

Podemos escrever a complexidade da função como:

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

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.

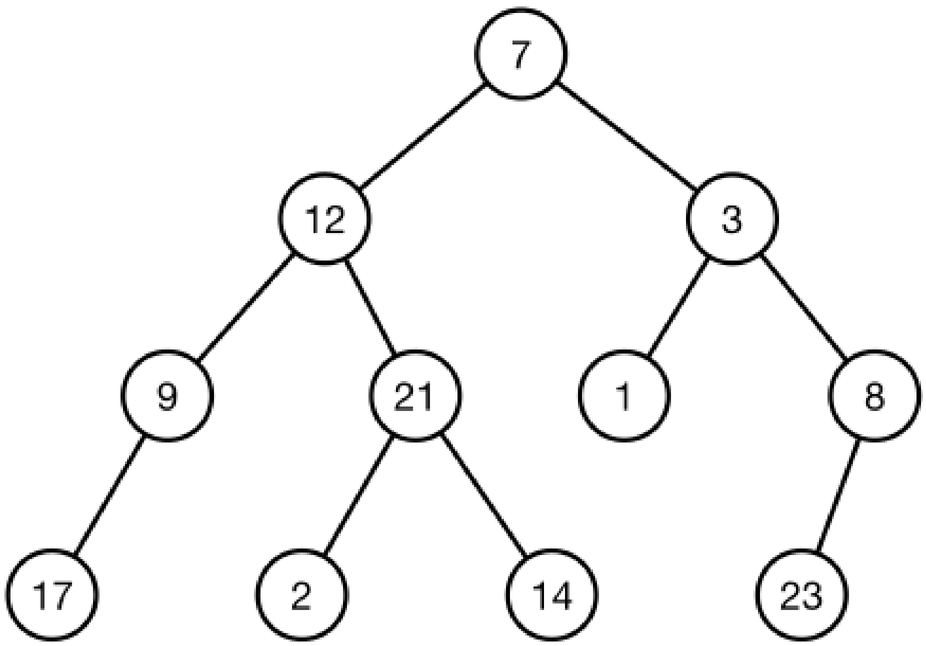


Figura 3: Exemplo de árvore

Um nó sem pai é uma **raíz**, 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 algura do nó raíz

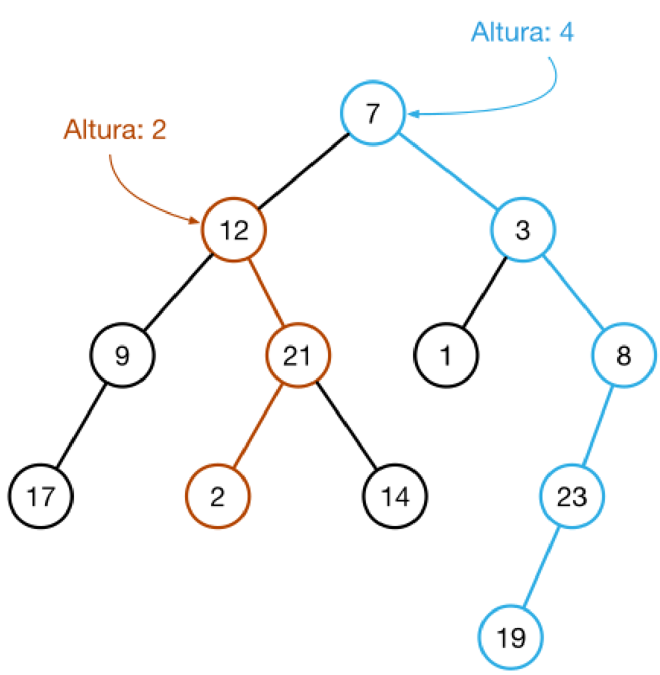


Figura 4: Exemplificação de altura

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

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

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:

$$h = \log(n) \tag{26}$$

Para códigos posteriores, considere a seguinte estrutura:

```cpp
class Node {
  public:
    Node(int key, char data)
      : m_key(key)
      , m_data(data)
      , m_leftNode(nullptr)
      , m_rightNode(nullptr)
      , m_parentNode(nullptr) {}
    Node & leftNode() const { return * m_leftNode; }
    void setLeftNode(Node * node) { m_leftNode = node; }

    Node & rightNode() const { return * m_rightNode; }
    void setRightNode(Node * node) { m_rightNode = node; }

    Node & parentNode() const { return * m_parentNode; }
    void setParentNode(Node * node) { m_parentNode = node; }

  private:
    int m_key;
    char m_data;
    Node * m_leftNode;
    Node * m_rightNode;
    Node * m_parentNode;
};
```

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

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

```cpp
  }

  int leftHeight = nodeHeight(node->leftNode());
  int rightHeight = nodeHeight(node->rightNode());

  if (leftHeight < rightHeight) {
    return rightHeight + 1;
  } else {
    return leftHeight + 1;
  }
}
```

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++

```cpp
void printTreeDFSInorder(class Node * node) {
  if (node == nullptr) {
    return;
  }
  printTreeDFSInorder(node->leftNode());
  cout << node->key() << " ";
  printTreeDFSInorder(node->rightNode());
}
```

PRÉ-ORDEM

C++

```cpp
void printTreeDFSPreorder(class Node * node) {
  if (node == nullptr) {
    return;
  }
  cout << node->key() << " ";
  printTreeDFSPreorder(node->leftNode());
  printTreeDFSPreorder(node->rightNode());
}
```

PÓS-ORDEM

C++

```cpp
void printTreeDFSPostorder(class Node * node) {
  if (node == nullptr) {
    return;
  }
  printTreeDFSPostorder(node->leftNode());
  printTreeDFSPostorder(node->rightNode());
  cout << node->key() << " ";
}
```

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

C++

```cpp
void printTreeBFSWithQueue(Node * root) {
  if (root == nullptr) {
    return;
  }
```

```cpp
  queue<Node*> queue;
  queue.push(root);
  while (!queue.empty()) {
    Node * node = queue.front();
    cout << node->key() << " ";
    queue.pop();
    Node * childNode = node->leftNode();
    if (childNode) {
      queue.push(childNode);
    }
    childNode = node->rightNode();
    if (childNode) {
      queue.push(childNode);
    }
  }
}
```

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

`left.key <= key <= right.key`

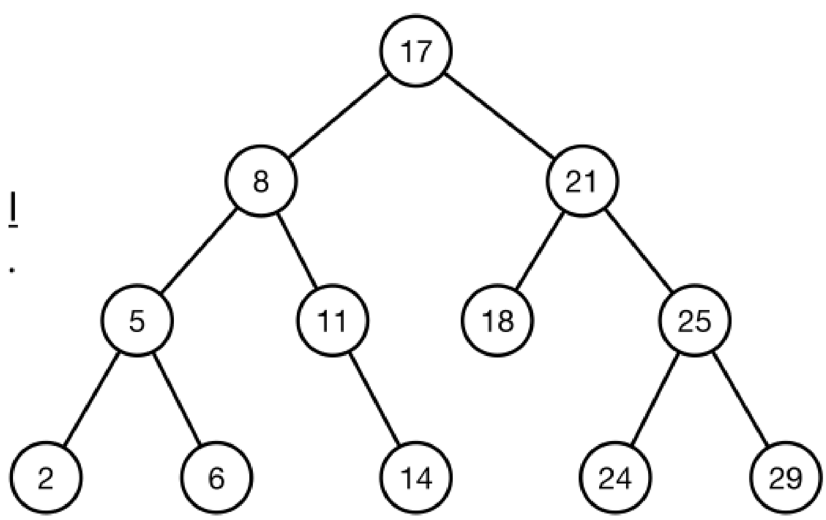


Figura 5: Exemplo de árvore binária

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++

```cpp
Node * binaryTreeSearchRecursive(Node * node, int key) {
  if (node == nullptr || node->key() == key) {
    return node;
  }
  if (node->key() > key) {
```

```
    return binaryTreeSearchRecursive(node->leftNode(), key);
  } else {
    return binaryTreeSearchRecursive(node->rightNode(), key);
  }
}
```

Esse algoritmo tem complexidade $\Theta(h)$

BUSCA EM ÁRVORE BINÁRIA (ITERATIVO) C++

```
Node * binaryTreeSearchIterative(Node * node, int key) {
  while (node != nullptr && node->key() != key) {
    if (node->key() > key) {
      node = node->leftNode();
    } else {
      node = node->rightNode();
    }
  }
  return node;
}
```

Tabela Hash

Nós a utilizamos para armazenar e pesquisar tuplas <*chave, valor*>. São comumente chamadas de **dicionários**, porém, podemos classificar assim:

- **Dicionários**: Maneira genérica de mapear *chaves* e *valores*
- **Hash Tables**: Implementação de um dicionário por meio de uma função de **hash**

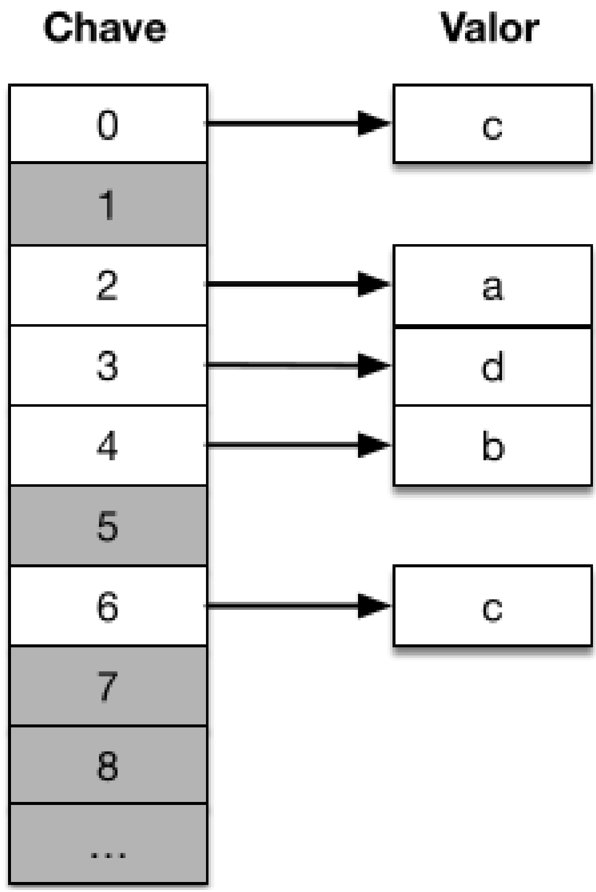


Figura 6: Desenho de tabela hash

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.

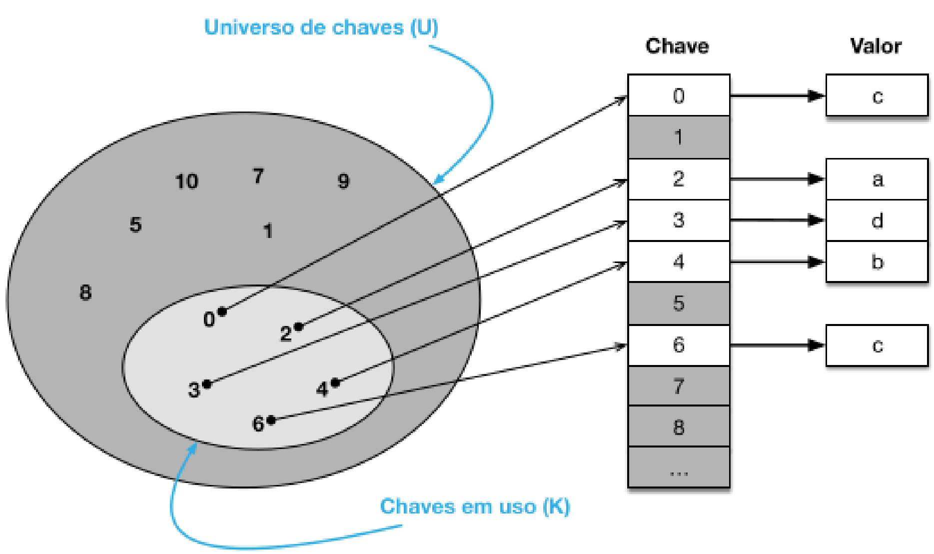


Figura 7: Estruturação da Hash Table

- **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. **Mandatório**: 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

```cpp
// Aloca-se um vetor com o tamanho do universo U:
int table[U];
for (int i = 0; i < U; i++) {
  table[i] = 0;
}

// Ao processar cada evento incrementa-se a posição no vetor
void add(int key) {
  table[key]++;
}

// Lê-se a contagem acessando a posição do vetor diretamente
int search(int key) {
  return table[key]
}
```

- `add` $= \Theta(1)$
- `search` $= \Theta(1)$

4.1.2 Segunda abordagem: Lista Encadeada

```cpp
typedef struct LLNode CountNode;
struct LLNode {
  int id;
  int count;
  CountNode * next;
};

void add(int key) {
  CountNode * node = m_firstNode;
  while (node != nullptr && node->id != key) {
    node = node->next;
  }
  if (node != nullptr) {
    node->count += 1;
  } else {
    CountNode * newNode = new CountNode;
    newNode->id = key;
    newNode->count = 1;
    newNode->next = m_firstNode;
    m_firstNode = newNode;
```

```
  }
}
int search(int key) {
  CountNode * node = m_firstNode;
  while (node != nullptr && node->id != key) {
    node = node->next;
  }
  return node != nullptr ? node->count : 0;
}
```

Infelizmente nessa abordagem nós não atingimos o objetivo principal de realizar as operações em $\Theta(1)$, já que a função de busca é $\Theta(n)$ no pior caso

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:

```
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, pelo princípio da casa dos pombos, 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.

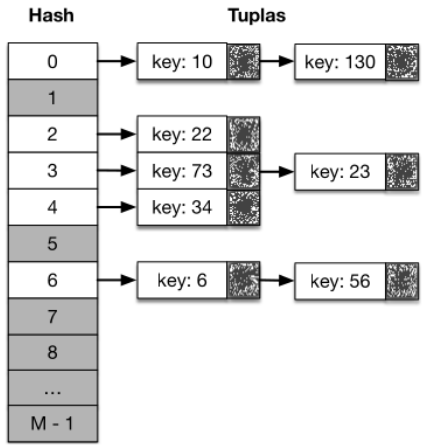


Figura 8: Tabela hash com encadeamento

EXEMPLO DE IMPLEMENTAÇÃO C++

```cpp
typedef struct HashTableNode HTNode;
struct HashTableNode {
  unsigned key;
  int value;
  HTNode * next;
  HTNode * previous;
};

class HashTable {
  public:
  HashTable(int size)
    : m_table(nullptr)
    , m_size(size) {
    m_table = new HTNode*[size];
      for (int i=0; i < m_size; i++) { m_table[i] = nullptr; }
    }
  ~HashTable() {
    for (int i=0; i < m_size; i++) {
      HTNode * node = m_table[i];
      while (node != nullptr) {
        HTNode * nextNode = node->next;
        delete node;
        node = nextNode;
      }
    }
    delete[] m_table;
  }
  ...
  private:
    unsigned hash(unsigned key) const { return key % m_size; }
    HTNode ** m_table;
    int m_size;
};


void insert_or_update(unsigned key, int value) {
  unsigned h = hash(key);
```

```
  HTNode * node = m_table[h];
  while (node != nullptr && node->key != key) {
    node = node->next;
  }
  if (node == nullptr) {
    node = new HTNode;
    node->key = key;
    node->next = m_table[h];
    node->previous = nullptr;
    HTNode * firstNode = m_table[h];
    if (firstNode != nullptr) {
      firstNode->previous = node;
    }
    m_table[h] = node;
  }
  node->value = value;
}


HTNode * search(unsigned key) {
  unsigned h = hash(key);
  HTNode * node = m_table[h];
  while (node != nullptr && node->key != key) {
    node = node->next;
  }
  return node;
}


bool remove(unsigned key) {
  unsigned h = hash(key);
  HTNode * node = m_table[h];
  while (node != nullptr && node->key != key) {
    node = node->next;
  }
  if (node == nullptr) {
    return false;
  }
  HTNode * nextNode = node->next;
  if (nextNode != nullptr) {
    nextNode->previous = node->previous;
  }
  HTNode * previousNode = node->previous;
  if (previousNode != nullptr) {
    node->previous->next = node->next;
  } else {
    m_table[h] = node->next;
  }
  delete node;
  return true;
}
```

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

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

Nas operações estamos considerando o pior caso.

4.3.2 Hash uniforme simples (A solução ideal)

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 à 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:

- **Simples**
  - Se a chave for um número real entre [0, 1)
  - `hash(key)` $= \lfloor \text{key} \cdot M \rfloor$
- **Método da divisão**
  - Se a chave for um número inteiro
  - `hash(key)` $= \text{key} \% M$
  - Costuma-se definir M como um número primo.
- **Método da multiplicação**
  - **hash(key)** $= \lfloor \text{key} \cdot A \% 1 M \rfloor$
  - 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++

```cpp
int hashStr(const char * value, int size) {
  unsigned hash = 0;
  for (int i=0; value[i] != '\0'; i++) {
    hash = (hash * 256 + value[i]) % size;
  }
  return hash;
}
```

A complexidade da função de espalhamento é constante: $\Theta(1)$. Em uma busca mal sucedida, temos que a complexidade é $T(n, m) = \frac{n}{m}$, isso se dá pois temos m entradas no array da tabela hash e temos n entradas utilizadas no todo, então a **complexidade média** do tempo de busca fica $\frac{n}{m}$. Então nosso objetivo é sempre que n seja bem menor que m, de forma que isso seja muito próximo de $\Theta(1)$.

Então podemos calcular a complexidade das operações de **remoção**, **inserção** e **busca** como:

$$T(n) = \frac{1}{n} \sum_{i}^{n} \left(1 + \sum_{j=i+1}^{n} \frac{1}{m} \right) = \Theta\left(1 + \frac{n}{m}\right) \tag{27}$$

Esse $\frac{1}{n} \sum_{i}^{n}$ representa uma média aritmética em todos os nós do valor que vem dentro da soma. Esse 1 dentro representa a operação de *hash* para descobrir o “slot” chave que você irá procurar.

Depois que você procurar o slot e achá-lo (Slot em que a chave que você está buscando estará), você vai percorrer um **número esperado** de $\sum_{j=i+1}^{n} \frac{1}{m}$ chaves ($\frac{1}{m}$ = Probabilidade (Considerando o hash uniforme simples) de uma chave i colidir com uma chave j)

Considerando a hipótese de hash uniforme simples podemos assumir que cada lista terá aproximadamente o mesmo tamanho.

Conforme você insere elementos na tabela o desempenho vai se degradando, calculando $\alpha = n/m$ a cada inserção conseguimos calcular se a tabela está em um estado ineficiente.

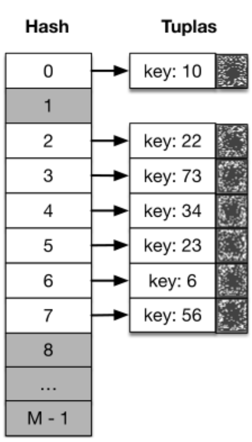
A operação de redimensionamento aumenta o tamanho do vetor de m para M', porém, isso invalida o mapeamento das chaves anteriores, já que a minha métrica era feita especificamente para o tamanho que eu tinha. Para contornar isso, podemos reinserir todos os elementos. Porém, isso é $\Theta(n)$. Se a operação de `resize` & `rehash` tem complexidade $\Theta(n)$, como manter $\Theta(1)$ para as demais operações?

Então temos a **análise amortizada**, que avalia a complexidade com base em uma sequência de operações.

A sequência de operações na tabela de dispersão consiste em:
- n operações de inserção com custo individual $\Theta(1)$
- k operações para redimensionamento com custo total $\sum_{i=1}^{\log(n)} 2^i = \Theta(n)$
  - Considerando que $M' = 2M$

$$\frac{n \cdot \Theta(1) + \Theta(n)}{n} = \Theta(1) \tag{28}$$

4.3.3 Tabela hash com endereçamento aberto



O problema de colisão é solucionado armazenando os elementos na primeira posição vazia a partir do índice definido pelo hash. Ou seja, quando eu vou inserir um elemento y na tabela, mas ele tem o mesmo hash do meu elemento x (Que já está inserido na tabela), eu simplesmente armazeno no próximo slot vazio

Vídeo muito bom com desenhos sobre endereçamento aberto (Clique aqui)

Estrutura de um nó da lista:

```cpp
typedef struct DirectAddressHashTableNode DANode;
struct DirectAddressHashTableNode {
  int key;
  int value;
};
```

Ao buscar (ou sondar) um elemento com a chave `key`, nós checamos: Se a posição `table[hash(key)]` estiver **vazia**, nós garantimos que a chave não está presente na tabela, mas se estiver **ocupada**, precisamos verificar se `table[hash(key)].key = key`, já que eu posso ter inserido uma outra chave lá.

Exemplo de implementação:

```cpp
class DirectAddressHashTable {
  public:
    DirectAddressHashTable(int size)
        : m_table(nullptr)
        , m_size(size) {
      m_table = new DANode[size];
      for (int i=0; i < m_size; i++) {
        m_table[i].key = -1;
        m_table[i].value = 0;
      }
    }
    ~DirectAddressHashTable() { delete[] m_table; }

  private:
    unsigned hash(int key) const { return key % m_size; }

    DANode * m_table;
    int m_size;
};


bool insert_or_update(int key, int value) {
  unsigned h = hash(key);
  DANode * node = nullptr;
  int count = 0;
  for (; count < m_size; count++) {
    node = &m_table[h];
    if (node->key == -1 || node->key == key) {
      break;
    }
    h = (h + 1) % m_size;
  }
  if (count >= m_size) {
    return false; // Table is full
  }
  if (node->key == -1) {
    node->key = key;
  }
  node->value = value;
  return true;
}


DANode * search(int key) {
  unsigned h = hash(key);
  DANode * node = nullptr;
  int count = 0;
  for (; count < m_size; count++) {
    node = &m_table[h];
    if (node->key == -1 || node->key == key) {
      break;
    }
```

```cpp
    h = (h + 1) % m_size;
  }
  return count >= m_size || node->key == -1 ? nullptr : node;
}


bool remove(int key) {
  DANode * node = search(key);
  if (node == nullptr) {
    return false;
  }
  node->key = -1;
  node->value = 0;
  return true;
}
```

A remoção em uma tabela hash com endereçamento aberto apresenta um problema:

- Ao remover uma chave key de uma posição h, partindo de uma posição h_0, tornamos impossível encontrar qualquer chave presente em uma posição $h' > h$, pois, quando eu procuro partindo de h_0, eu vou interpretar que, como h está vazio, eu não preciso mais ir para frente, porém a chave que estou procurando pode estar depois.

Antes M=10

hash	0	1	2	3	4	5	6	7	8	9
chave	60	21	51	131	33	91	76	61	-1	99

Depois

hash	0	1	2	3	4	5	6	7	8	9
chave	60	21	51	-1	33	91	76	61	-1	99

Figura 10: Exemplo de tabela com problema na remoção

Uma possível solução consiste em marcar o nó removido de forma que a busca não o considere vazio.

- Podemos criar uma flag para representar que o nó será reciclado.

```cpp
typedef struct DirectAddressHashTableNode DANode;
struct DirectAddressHashTableNode {
  int key;
  int value;
  bool recycled;
};
```

- E inicializá-la com o valor false no construtor:

```cpp
m_table[i].recycled = false;
```

Então vamos adaptar as funções de busca e remoção

```cpp
DANode * search(int key) {
  unsigned h = hash(key);
  DANode * node = nullptr;
  int count = 0;
  for (; count < m_size; count++) {
    node = &m_table[h];
    if ((node->key == -1 && !node->recycled) || node->key == key) {
      break;
    }
    h = (h + 1) % m_size;
  }
  return count >= m_size || node->key == -1 ? nullptr : node;
}


bool remove(int key) {
  DANode * node = search(key);
  if (node == nullptr) {
    return false;
  }
  node->key = -1;
  node->value = 0;
  node->recycled = true;
  return true;
}
```

O fator de carga da abordagem de endereçamento aberto é definido da mesma forma: $\alpha = n/M$

- No entanto observe que nesse caso teremos sempre $\alpha \leq 1$ visto que M é o número máximo de elementos no vetor (Antes eu podia ter mais chaves do que espaços no meu vetor).
- A busca por uma determinada chave depende da sequência de sondagem `hash(key, i)` fornecida pela função de espalhamento
- Observe que existem M! permutações possíveis para a sequência de sondagem.
- A sondagem linear é o método mais simples de gerar a sequência de espalhamento `hash(key, i) = (hash'(key) + i) % M`

Porém, a abordagem linear rapidamente se torna ineficaz, já que em determinado momento o problema se transforma basciamente em inserir elementos em uma lista, então surge a alternativa da abordagem quadrática

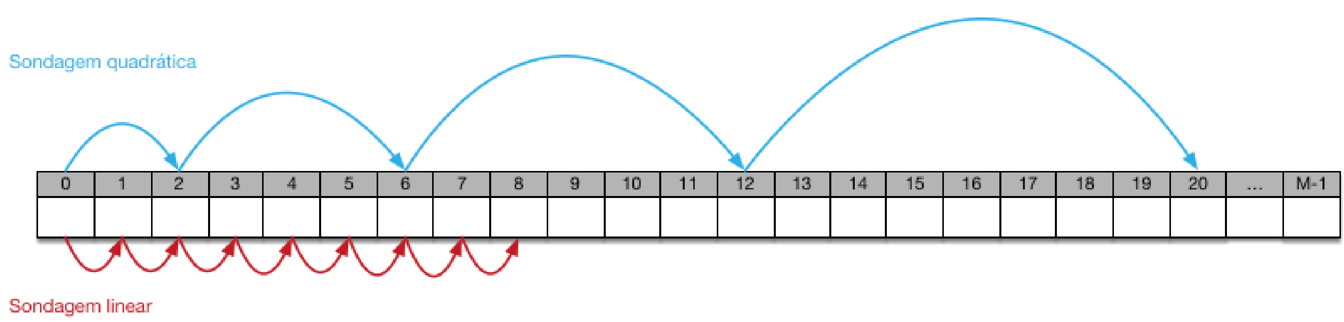


Figura 11: Exemplificação de sondagem quadrática

Agora temos que a função de hash segue o seguinte padrão: `hash(key, i) = ( hash'(key) + b*i + a*i**2 ) % m`

Porém isso gera agrupamentos secundários, ou seja, se duas chaves caem no mesmo local inicial `hash'(key)`, então elas seguirão a mesma sequência e tentarão ocupar os mesmos slots (Aí podemos inserir outras abordagens)

Para melhorar ainda mais nossas abordagens, podemos introduzir o **hash duplo**, que eu vou ter **duas** funções de hash diferentes hash_1 e hash_2 de forma que o novo hash de uma chave vai ser dado por: `hash(key, i) = (hash1(key) + i * hash2(key)) % m` de forma que, mesmo que uma mesma chave

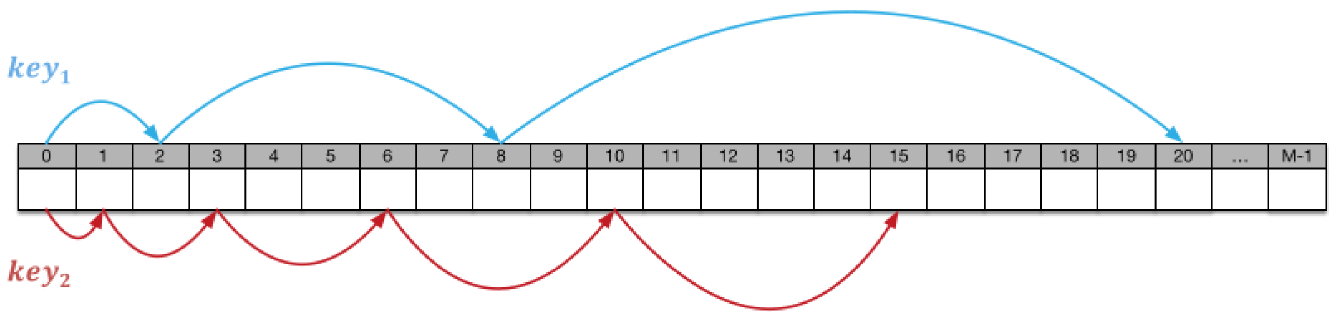


Figura 12: Exemplificação de hash duplo

Porém, vale ressaltar que minha segunda função de hash deve satisfazer:

- Ser completamente diferente da primeira
- Não retornar 0

O número de sondagens para inserir uma chave em uma tabela hash de endereçamento aberto (No caso médio) é:

$$T(n) = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha} = O(1) \tag{29}$$

ALGORITMOS DE ORDENAÇÃO

Queremos que, dada uma sequência de valores, escreva um algoritmo capaz de retornar a sequência ordenada de valores a partir de uma entrada de vários números não-ordenados.

```cpp
int v[] = {8, 11, 2, 5, 10, 16, 7, 15, 1, 4};
```

- Exceto quando especificado de outra forma, assuma que o tipo dos valores são números inteiros
- Utilizaremos o vetor como estrutura de dados, no entanto os algoritmos apresentados podem ser implementados utilizando outras estruturas, como listas encadeadas

Um algoritmo de ordenação é considerado **estável** quando, ao final do programa, elementos de mesmo valor aparecem na mesma ordem que antes. Por exemplo:

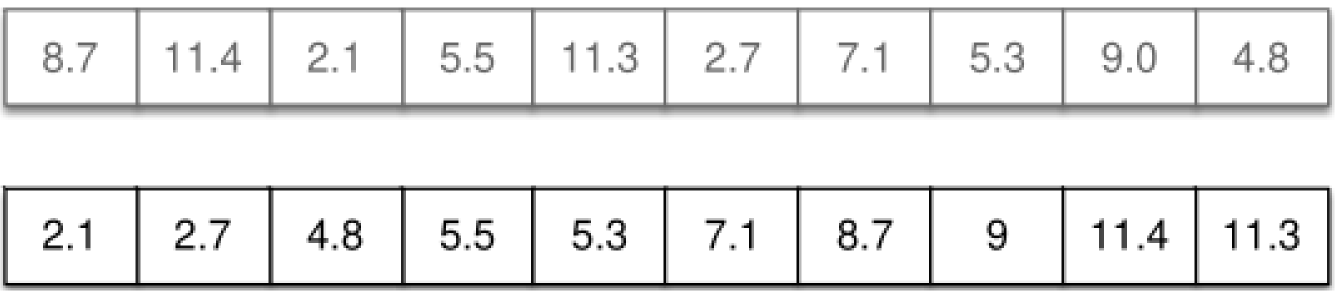


Figura 13: Exemplo com números fracionários

Considere um algoritmo que ordena o vetor mostrado acima considerando **apenas a parte inteira**. Nesse algoritmo, 5.5 e 5.3 tem o mesmo valor (Já que estamos considerando apenas a parte inteira), e no array antes da ordenação, 5.5 aparece **antes** do 5.3. Se o algoritmo for estável, então essa ordem deverá ser mantida e, como podemos ver no array ordenado, ela de fato foi

5.1 Bubble Sort

O algoritmo bubble sort (ordenação por flutuação) é uma das soluções mais simples para o problema de ordenação. A solução consiste em inverter (trocar) valores de posições adjacentes sempre que `v[i + 1] < v[i]`. Essa operação é executada para cada posição 0 ≤ i < n − 1 ao percorrer a sequência. Observe que ao percorrer a sequência j = n − 1 vezes executando esse procedimento atingimos a sequência ordenada.

Exemplo: Considere o seguinte array:

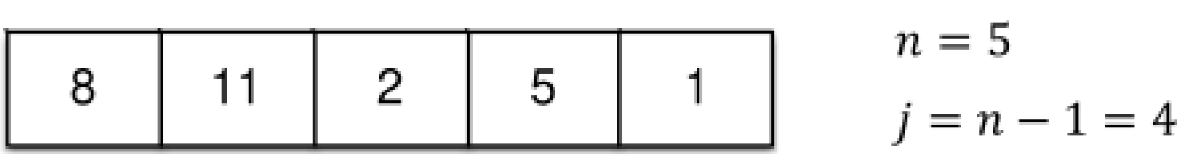


Figura 14: Bubble Sort array de exemplo

E a execução do código decorrerá da forma:

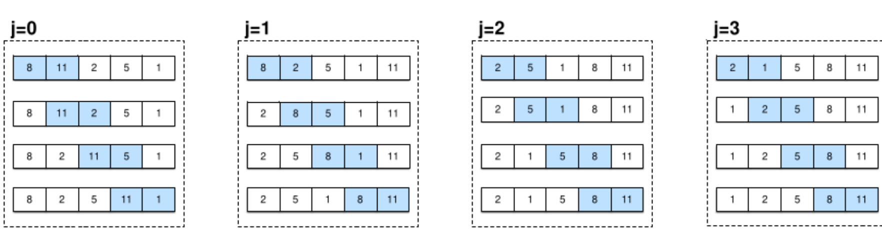


Figura 15: Bubble Sort exemplo de fluxo de código

Então o meu código vai percorrer cada item da minha lista e, sempre que um elemento a esquerda de outro é maior que ele, os dois trocam de posição

IMPLEMENTAÇÃO C++

```cpp
#define swap(v, i, j) { int temp = v[i]; v[i] = v[j]; v[j] = temp; }
void bubbleSort(int v[], int n) {
  for (int j = 0; j < n - 1; j++) {
    for (int i = 0; i < n - 1; i++) {
      if (v[i] > v[i + 1]) {
        swap(v, i, i + 1);
      }
    }
  }
}
```

O procedimento é executado $n - 1$ vezes e, a cada iteração maior, ele executa $n - 1$ subprocessos, logo, no final vamos ter um total de $T(n) = \Theta(n^2)$ de complexidade (Tanto melhor quanto pior caso)

Porém, podemos fazer uma otimização no algoritmo:

IMPLEMENTAÇÃO ORDENADA C++

```cpp
void bubbleSortOptimized(int v[], int n) {
  for (int j = 0; j < n - 1; j++) {
    bool swapped = false;
    for (int i = 0; i < n - 1; i++) {
      if (v[i] > v[i + 1]) {
        swap(v[i], v[i + 1]);
        swapped = true;
      }
    }
    if (!swapped) { break; }
  }
}
```

Essa otimização checa se, dentro de um loop maior houve alguma troca, se não houve nenhuma, então o algoritmo é encerrado. Ao fazer isso, a complexidade do melhor caso desce para $\Theta(n)$

5.2 Selection Sort

No selection sort, fazemos uma busca em **cada posição** pelo i-ésimo valor que **deveria** estar naquela posição

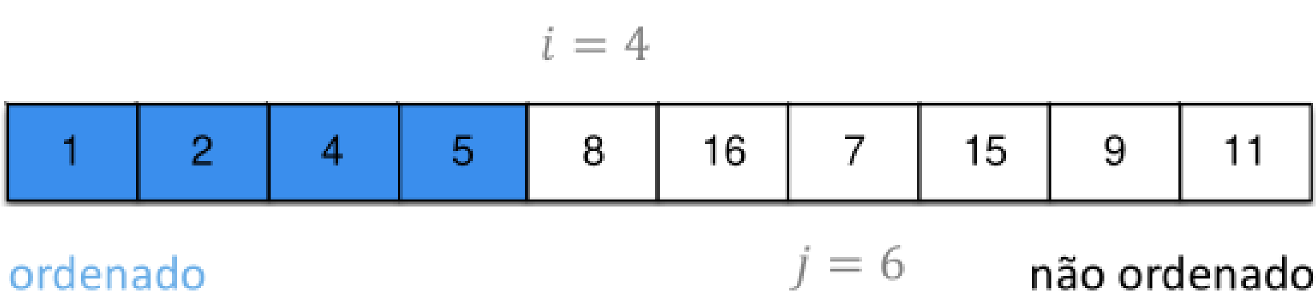


Figura 16: Selection Sort exemplificação

Dado uma posição i, e assumindo que todas as posições anteriores já estão ordenadas, ele vai procurar dentre as próximas $n - i$ posições um valor menor que o da posição i. Se isso acontece, significa que ele deveria estar na posição que i está ocupando, então eu vou trocá-los de posição

Exemplo: Considere o caso:

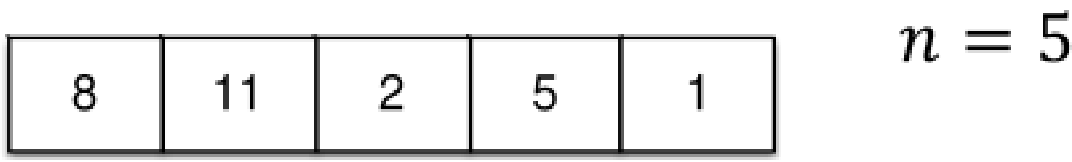


Figura 17: Selection Sort Caso de Exemplo

E assim, o fluxo durante a execução do programa será:

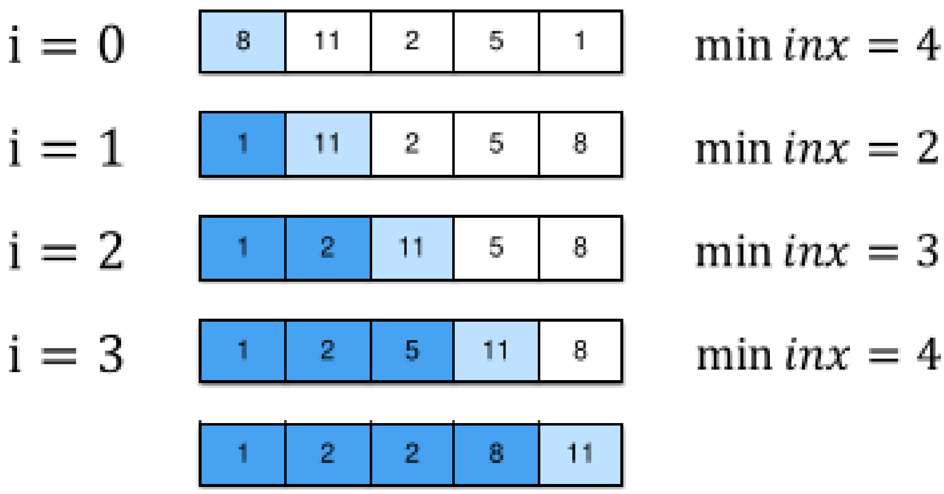


Figura 18: Selection Sort Fluxo do Programa

IMPLEMENTAÇÃO C++

```cpp
void selectionSort(int v[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int minInx = i;
    for (int j = i + 1; j < n; j++) {
      if (v[j] < v[minInx]) {
        minInx = j;
      }
    }
    swap(v, i, minInx);
  }
}
```

Para avaliar o seu desempenho, podemos montar seu custo total utilizando vendo que, a cada iteração, eu vou avaliar um elemento a menos, de forma que podemos expressar a **função de complexidade** como:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + \ldots + 1 + 0 \\ &= \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \end{aligned} \tag{30}$$

Ou seja, obtemos que $T(n) = \Theta(n^2)$, que também é a complexidade no melhor caso

5.3 Insertion Sort

Muito parecido com o algoritmo de **Selection Sort**, porém, eu vou fixar uma posição i e avaliar o valor naquela posição, e procurar dentre as posições $[0, i-1]$ qual deveria ser a posição que o valor da posição i deveria estar

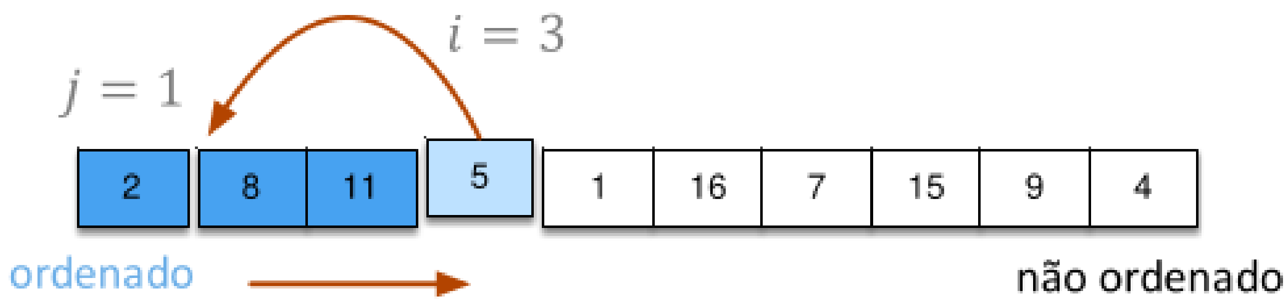


Figura 19: Insertion Sort Exemplificação

Exemplo:

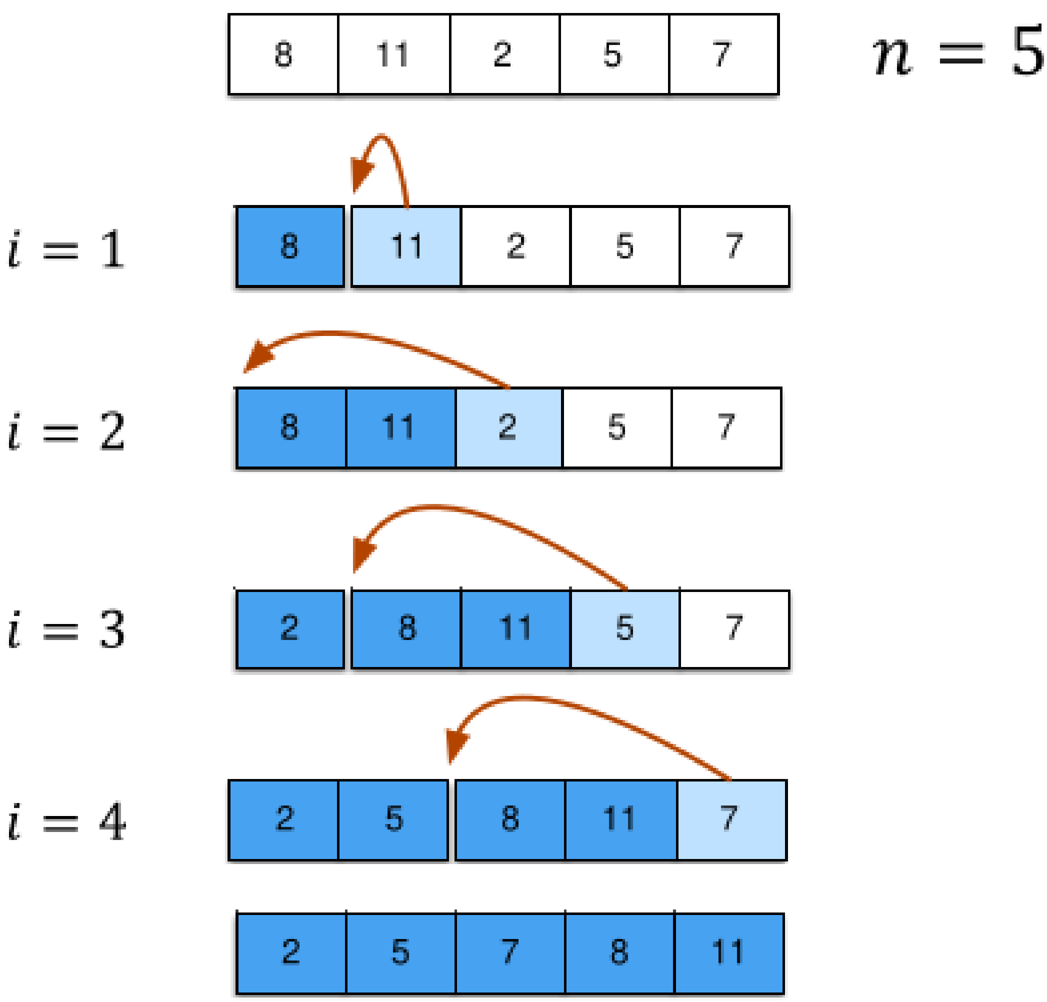


Figura 20: Insertion Sort Exemplo

IMPLEMENTAÇÃO C++

```cpp
void insertionSort(int v[], int n) {
  for (int i = 1; i < n; i++) {
    int currentValue = v[i];
    int j;
```

```
    for (j = i - 1; j >= 0 && v[j] > currentValue; j--) {
      v[j + 1] = v[j];
    }
    v[j + 1] = currentValue;
  }
}
```

A complexidade desse algoritmo também é expressa na forma:

$$T(n) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2) \tag{31}$$

Porém, no melhor caso, temos que $T(n) = \Theta(n)$

5.4 Mergesort

O algoritmo Mergesort consiste em dividir a sequência em duas partes, executar chamadas recursivas para cada sub-sequência, e juntá-las (merge) de forma ordenada. Esse algoritmo depende de um algoritmo auxiliar de intercalação (merge)

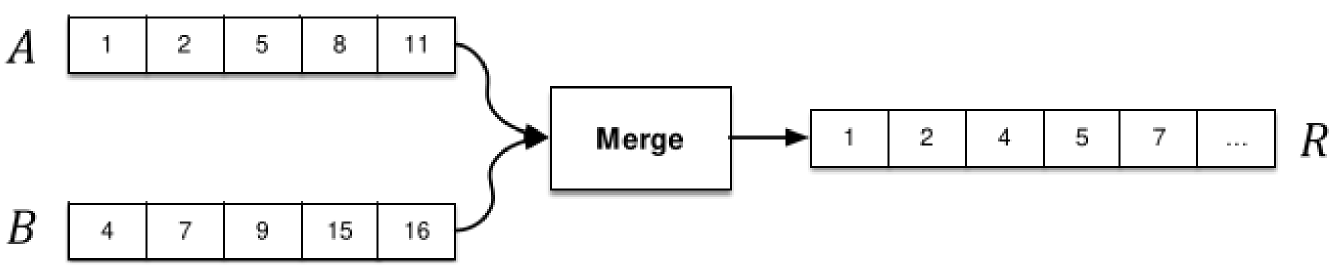


Figura 21: Mergesort Exemplificação

EXEMPLIFICAÇÃO DA FUNÇÃO DE INTERCALAÇÃO C++

```cpp
void merge(int v[], int startA, int startB, int endB)
{
  int r[endB - startA];
  int aInx = startA;
  int bInx = startB;
  int rInx = 0;
  while (aInx < startB && bInx < endB) {
    r[rInx++] = v[aInx] <= v[bInx] ? v[aInx++] : v[bInx++];
  }
  while (aInx < startB) { r[rInx++] = v[aInx++]; }
  while (bInx < endB) { r[rInx++] = v[bInx++]; }
  for (aInx = startA; aInx < endB; ++aInx) {
    v[aInx] = r[aInx - startA];
  }
}
```

Dado que cada sequência tem n entradas, são executadas $2n$ operações, logo, a complexidade é $\Theta(n)$

O algoritmo consiste em dividir a sequência R em duas subsequências A e B. Eu recursivamente ordeno essas duas sequências e repito o processo até que R esteja ordenado

IMPLEMENTAÇÃO C++

```c
void mergeSort(int v[], int startInx, int endInx) {
  if (startInx < endInx - 1) {
    int midInx = (startInx + endInx) / 2;
    mergeSort(v, startInx, midInx);
    mergeSort(v, midInx, endInx);
    merge(v, startInx, midInx, endInx);
  }
}
```

Podemos então avaliar a função de complexidade:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{32}$$

E já vimos em capítulos anteriores que isso é $T(n) = \Theta(n \log(n))$. Perceba que ele não compara todos os pares mesmo no pior caso, porém, ele exige um espaço de memória $O(n)$ **adicional** para a ordenação

5.5 Quicksort

Um tipo de mergesort, mas contém um algoritmo auxiliar específico, com exceção também que buscamos um algoritmo que não necessite dos $O(n)$ de espaço adicional. O algoritmo escolhe um elemento, o qual chamamos de **pivô**, e separamos em duas partições. Os elementos maiores e menores que o **pivô**

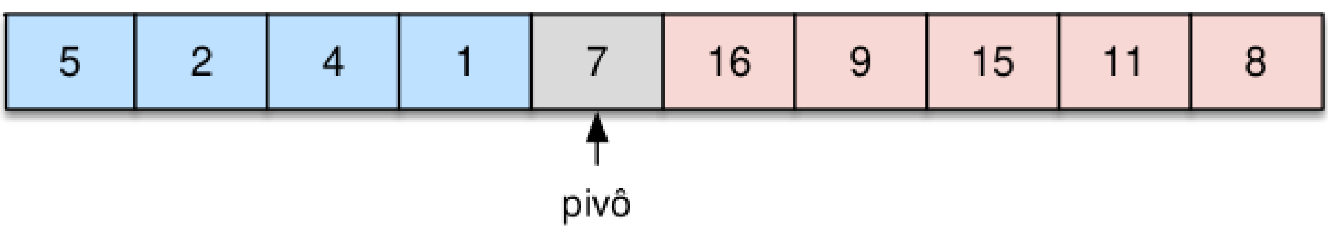


Figura 22: Exemplificação Quicksort

Então resumimos o problema do particionamento como:

Dada uma sequência v e um intervalo $[p, ..., r]$ transponha elementos desse intervalo de forma que ao retornar um índice j (pivô) tenhamos:

$$v[p, ..., j-1] \leq v[j] \leq v[j+1, ..., r] \tag{33}$$

Uma solução muito comum segue o seguinte:

```
função particionamento(v ∈ ℝ^r) {
  var pivô = v[r]
  var j = r
  percorrer sequência avaliando cada posição i ∈ [p, r − 1] {
    se v[i] ≤ pivô {
      trocar v[i] e v[j]
      j = j + 1
    }
  }
}
```

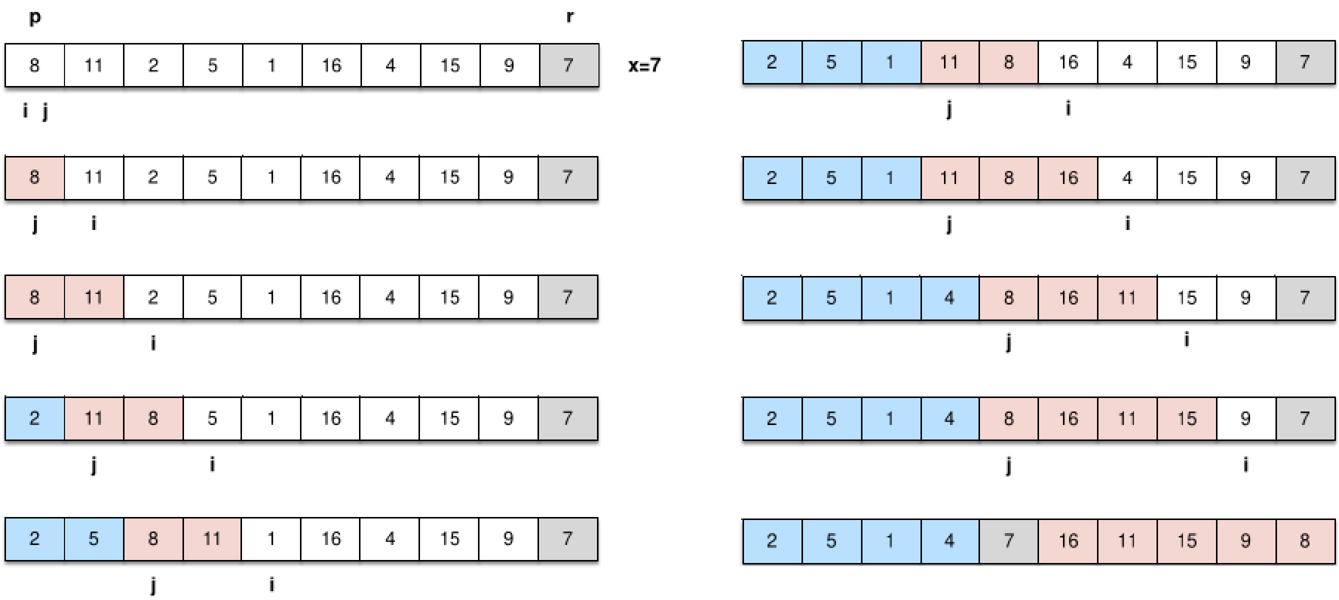


Figura 23: Exemplo visual do algoritmo quicksort

IMPLEMENTAÇÃO C++

```cpp
#define swap(v, i, j) { int temp = v[i]; v[i] = v[j]; v[j] = temp; }
int partition(int v[], int p, int r) {
  int pivot = v[r];
  int j = p;
  for (int i=p; i < r; i++) {
    if (v[i] <= pivot) {
      swap(v, i, j);
      j++;
    }
  }
  swap(v, j, r);
  return j;
}
```

Como a sequência de n elementos é percorrida uma única vez executando operações constantes, temos que $T(n) = \Theta(n)$

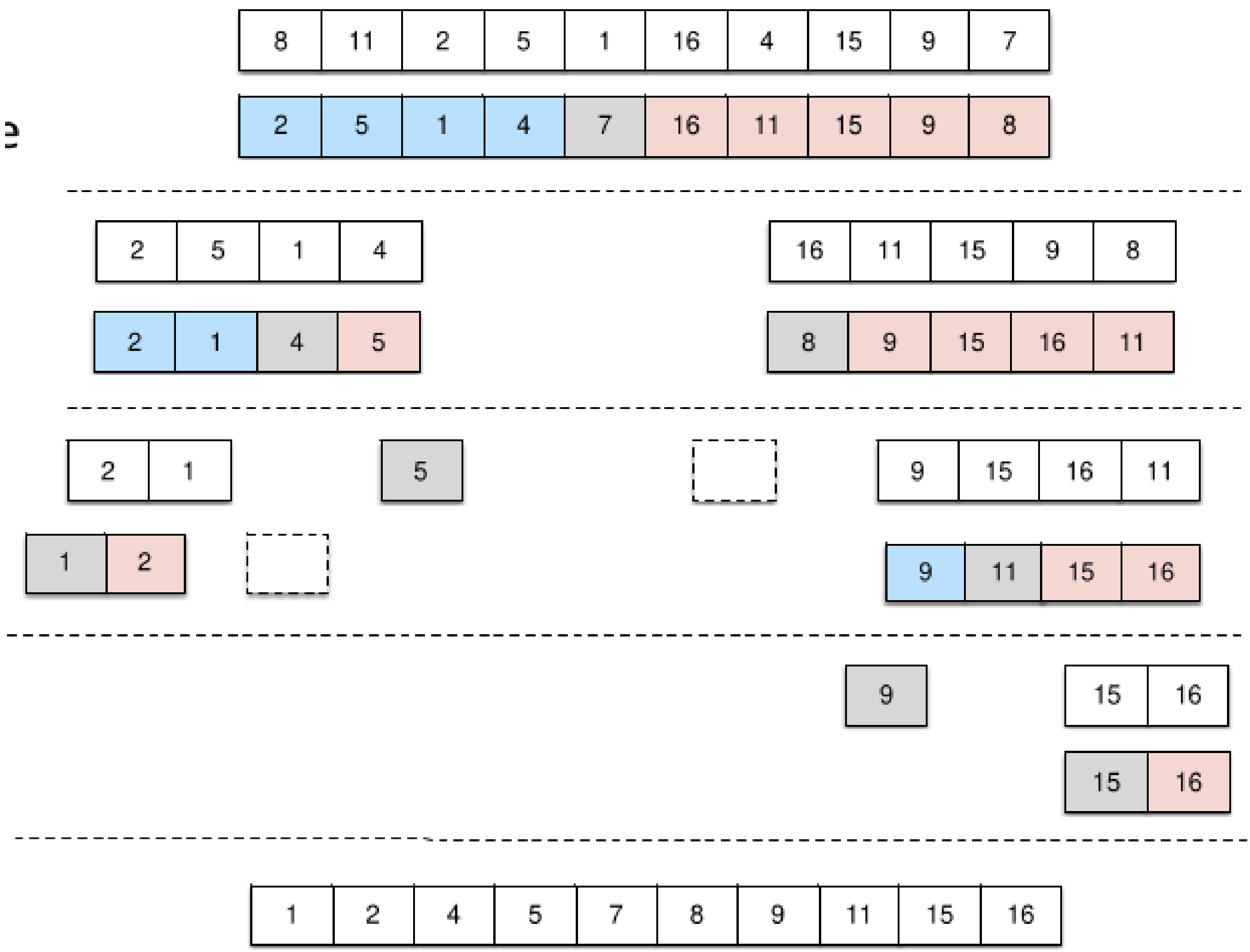


Figura 24: Passo-a-passo do algoritmo

IMPLEMENTAÇÃO C++

```cpp
void quicksort(int v[], int p, int r) {
  if (p < r) {
    int j = part j + 1, r);
  }
}
quicksort(v, 0, n - 1);
```

Temos que sua função de complexidade é tal que:

$$
\begin{aligned}
T(n) &= T(j) + T(n - j - 1) + n \\
&= T(0) + 1 + 2 + 3 + \ldots + (n - 1) + n \\
&= \frac{n(n + 1)}{2} \\
&= \Theta(n^2)
\end{aligned}
\tag{34}
$$

O pior caso acontece quando o pivô é o último/primeiro elemento e ele é o maior/menor elemento, já que uma partição fica vazia

Já o melhor caso ocorre quando o algoritmo sempre divide todas as partições ao meio

$$T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log(n)) \tag{35}$$

Já o caso médio ocorre quando o algoritmo divide em partições de tamanho diferente. Imagine que o algoritmo divide em partições do tipo $0.1n$ e $0.9n$

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n \tag{36}$$

Podemos avaliar como $\Theta(n \log(n))$, ou seja, é o mesmo caso do melhor caso possível, mas tem uma constante maior. Ou seja, conseguimos perceber que o desempenho do algoritmo depende **da escolha do pivô**. O pior caso é $\Theta(n^2)$, porém só ocorre em casos muito extremos

5.6 Heapsort

O algoritmo Heapsort consiste em organizar os elementos em um heap binário e reinseri-los utilizando uma estratégia semelhante à do algoritmo de ordenação por seleção.

O heap (monte) é uma estrutura de dados capaz de representar um vetor sob a forma de uma árvore binária, que apresenta as seguintes propriedades:
- É uma árvore quase completa
- Todos os níveis devem estar preenchidos exceto pelo último.
- É mínimo ou máximo
  - Heap mínimo – cada filho será maior ou igual ao seu pai.
  - Heap máximo – cada filho será menor ou igual ao seu pai.

Por enquanto, vamos considerar os **heaps máximos**. A altura de um **heap** com n nós é dada por $\lfloor \log_2(n) \rfloor$. Podemos representar um heap utilizando um **array**, de forma que ele segue as seguintes regras:
- O índice 1 é a raíz da árvore
- O pai de qualquer índice p é $\frac{p}{2}$, com exceção do nó raíz
- O filho esquerdo de um nó p é $2p$
- O filho esquerdo de um nó p é $2p + 1$

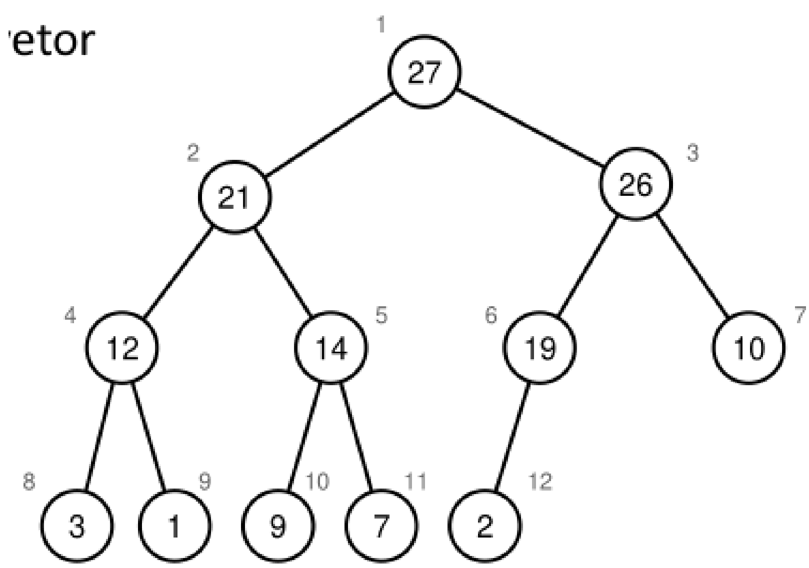


Figura 25: Exemplo de HEAP

27	21	26	12	14	19	10	3	1	9	7	2
1	2	3	4	5	6	7	8	9	10	11	12

Figura 26: Forma da Figura 25 como vetor

Podeos ordenar uma árvore em um heap caso as propriedades do vetor não sejam satisfeitas. Podemos utilizar do algoritmo **max-heapify**

- Assume-se que as sub-árvores do nó são heaps-máximos.
- Caso $v[p]$ seja menor que $v[2p]$ ou $v[2p+1]$ escolhe o maior e executa a troca.
- Em seguida executa max-heapify recursivamente no nó filho alterado.

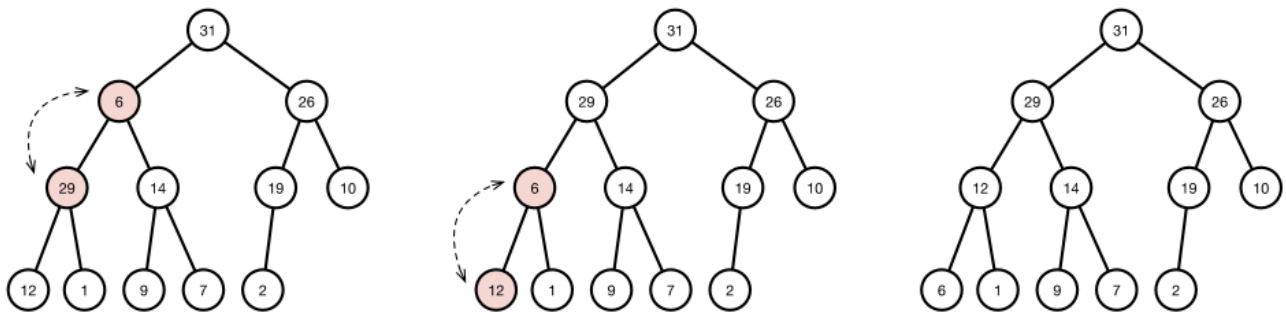


Figura 27: Visualização do algoritmo **max-heapify**

IMPLEMENTAÇÃO C++

```cpp
void heapify(int v[], int n, int i) {
  int inx = i;
  int leftInx = 2 * i + 1;
  int rightInx = 2 * i + 2;
  if ((leftInx < n) && (v[leftInx] > v[inx])) {
    inx = leftInx;
  }
  if ((rightInx < n) && (v[rightInx] > v[inx])) {
    inx = rightInx;
  }
  if (inx != i) {
    swap(v, i, inx);
    heapify(v, n, inx);
  }
}
```

A complexidade desse algoritmo é $T(n) = O(\log n)$.

Dado o devido contexto sobre os **heaps**, vamos voltar para o algoritmo de **heapsort**. Esse algoritmo tem dois passos principais:

1. Organizar o vetor de entradas em um **heap**
2. Ordenar os elementos executando os seguintes passos para $v[n, ..., 1]$:
   - Trocar o elemento atual $v[i]$ pela raíz $v[1]$ ($v[1]$ é o maior elemento do heap)
   - Corrigir o **heap** usando o **heapify** para a raíz

IMPLEMENTAÇÃO C++

```cpp
void buildHeap(int v[], int n) {
  for (int i=(n/2-1); i >= 0; i--) {
    heapify(v, n, i);
  }
}
void heapSort(int v[], int n) {
  buildHeap(v, n);
  for (int i=n-1; i > 0; i--) {
```

```
        swap(v, 0, i);
        heapify(v, i, 0);
    }
}
```

Para analisar o desempenho, podemos fazer o seguinte:

- **Construção do heap**: Executa um **heapify** em um vetor de $\approx n/2$ posições, logo $O(n \log(n))$
- **Ordenação**: Executa o **heapify** para cada elemento em um vetor de $n-1$ posições, logo, temos um $O(n \log(n))$

No final, somando tudo, temos que $T(n) = \Theta(n \log(n))$. Melhorando um pouco a argumentação sobre a etapa de construção do heap, vamos analisar **aproximadamente**

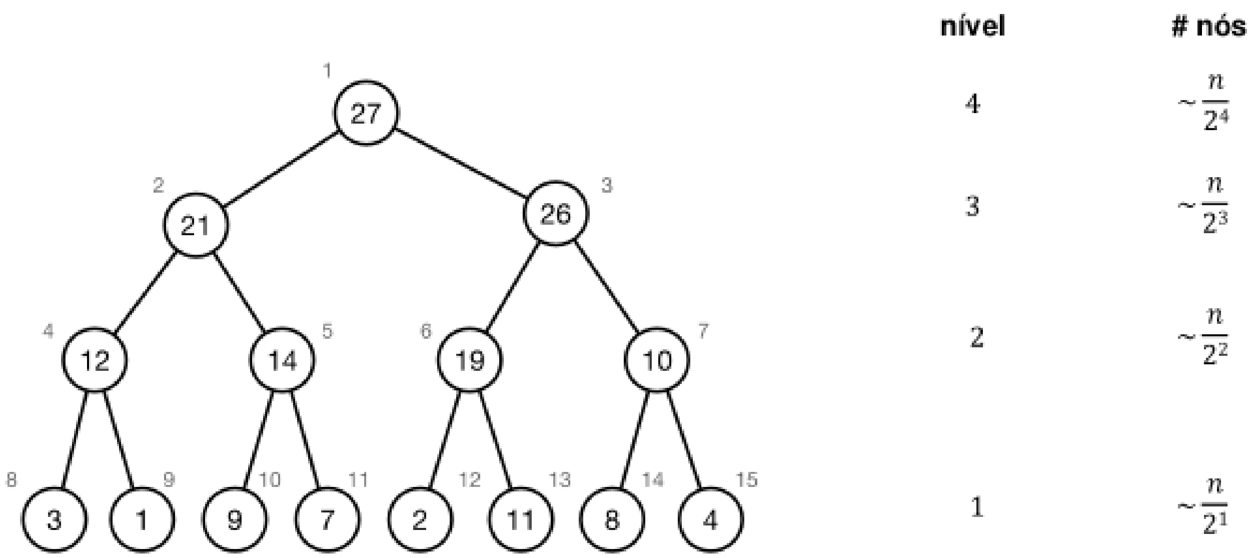


Figura 28: Aproximação de um HEAP

Cada nível i, de baixo para cima, tem aproximadamente $n/2^i$ nós, ou seja, o custo total vai ser:

$$T(n) = \sum_{i=1}^{\log(n)} i \cdot \frac{n}{2^i} = O(n) \qquad (37)$$