FGV EMAp

Escrita: João Pedro Jerônimo

Revisão: Thalis Ambrosim Falqueto

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 saber qual é o melhor caminho até a solução? De primeira podemos pensar: "É só ver quanto tempo demora para executar!", mas isso gera um problema... Se executarmos um algoritmo em um computador atual e o mesmo algoritmo em um computador de 1980, com certeza eles vão levar tempos diferentes para executar, correto? Isso pode afetar na medição do algoritmo!

Então, o que fazer? O mais comum é analisarmos o quão bem o algoritmo consegue funcionar de acordo com o quão grande o 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*:
>
> ```cpp
> int sum(const int numbers[], int size) {
>   int result = 0;
>   for (int i = 0; i < size; i++) {
>       result += numbers[i];
>   }
>   return result;
> }
> ```
>
> Portanto, temos que, para esse algoritmo, $T(n) = n$, pois o algoritmo depende diretamente do tamanho da entrada, já que passa uma vez por cada elemento Como isso acontece somente uma vez(além de declarações unitárias de variáveis que não dependendem de n), $T(n) = n$.

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

Se a partir de algum ponto certa função T_1 cresce mais do que T_2, então o algoritmo T_1 é pior que T_2, por isso, 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

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

Note que a definição acima apenas limita inferiormente, enquanto a primeira limita superiormente a função $T(n)$.

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

Ou seja, o algoritmo é completamente limitado e definido por $f(n)$(perceba que nem sempre é possível limitar o algoritmo superiormente e inferiormente pela mesma função).

Por fim, perceba que, se que $T(n) = O(f(n))$, e $h(n_0) > f(n_0)\ \forall n > n_0$, então $T(n) = O(h(n))$.

Exemplo: Digamos que $T(n) = O(n)$. Logo, a partir de certo ponto, $T(n) = O(n^2)$, já que também consegue ser limitado pela função n^2.

Essa ideia serve principalmente para dizer que qualquer função maior que $f(n)$ pode limitar $T(n)$, e é claro que você, caro leitor, pode achar isso óbvio, mas parece um pouco duvidoso achar que $T(n) = n$ é $O(n^3)$, porém isso é verdade.

O mesmo vale para funções menores que $\Omega(f(n))$, por isso, fique atento a esse tipo de pegadinha!

Recorrência

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

Exemplo:

```cpp
int fatorial(int n) {
  if (n == 1) {
      return 1;
  }
  return n * fatorial(n - 1);
}
```

Aqui, temos um $T(n)$ que chama $T(n-1)$, até que se chegue no caso base de $n = 1$, como calcular a complexidade disso?

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

A ideia é provar por **indução** que $T(n)$ é O de uma função **pressuposta**. Por isso, é claro, só é passível de uso quando se tem uma hipótese da solução, e provamos exatamente a hipótese na indução. 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

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

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

A ideia 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:

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

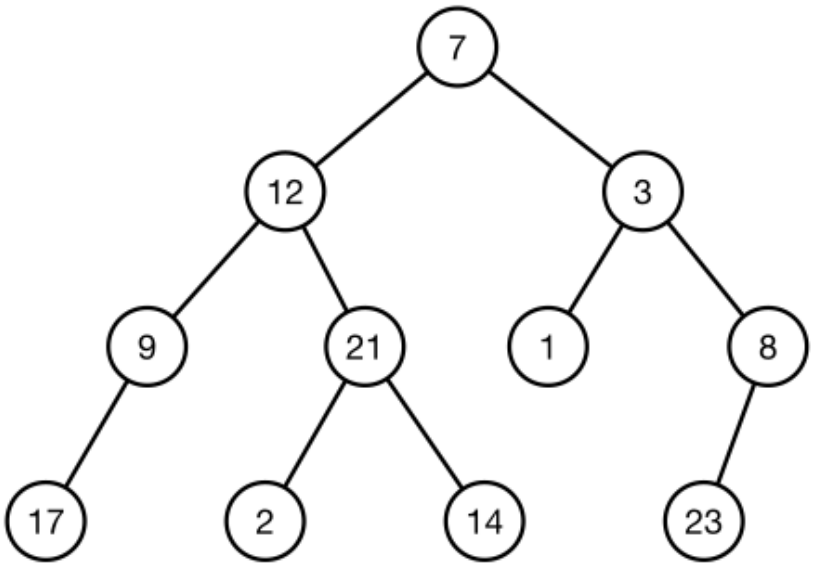


Figura 2: Árvore de $T(n)$

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:

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

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 $f(n) = O\big(n^{\log_b(a) - \varepsilon}\big)$ para alguma constante $\varepsilon > 0$, então $\boldsymbol{T(n) = \Theta\big(n^{\log_b(a)}\big)}$
- Se $f(n) = \Theta\big(n^{\log_b(a)}\big)$, então $\boldsymbol{T(n) = \Theta(f(n)\log(n))}$
- Se $f(n) = \Omega\big(n^{\log_b(a) + \varepsilon}\big)$ para alguma constante $\varepsilon > 0$ e atender a uma condição de regularidade $af\big(\frac{n}{b}\big) \leq cf(n)$ 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, $f(n) = \Theta\big(n^{\log_b(a)}\big)$, e isso quer dizer que $T(n) = \Theta\Big(n^{\log_{\frac{3}{2}}(1)\log(n)}\Big) = \Theta(\log(n))$

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

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

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 $f(n) = \Omega\big(n^{\log_b(a)+\varepsilon}\big)$. Vamos então checar a regularidade:

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

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:

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

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

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

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 $\Theta(n)$

- Nota: Um erro comum de interpretação é se perguntar por quê foi usado um $\Theta(n)$ se, claramente, o algoritmo é $\Omega(1)$. Acontece que estamos analisando o pior caso, ou seja, quando o elemento é o último da lista. Por isso, não existem análises de pior e melhor caso se sabemos que teremos que percorrer n elementos até chegar no inteiro que estamos procurando, por isso, no caso do pior caso, o algoritmo tem complexidade $\Theta(n)$.

Porém, se considerarmos uma lista ordenada, podemos fazer algo mais inteligente. Começamos comparando o elemento do meio do vetor e dependendo se o valor que queremos buscar é maior ou menor comparado ao avaliado, podemos ignorar a parte oposta do vetor.Ou seja, 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:

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

Pense no método de Árvore de Recursão, que aprendemos há pouco. Note que, nesse caso, $T(n)$ só se separa em um termo, e, a cada iteração, temos o tamanho do vetor dividido por 2 e um termo $+c$. Portanto, nosso custo é $\frac{n}{2^k}$ e queremos achar o k que satisfaz esse termo chegar ao caso base, $T(1)$. Logo,

$$\frac{n}{2^k} = 1 \Rightarrow \log(n) - \log\left(2^k\right) = 0 \Rightarrow \log(n) = k\log(2) \Rightarrow k = \frac{\log(n)}{\log(2)} \tag{25}$$

Portanto, temos o custo por nó de c, e assim o custo total fica:

$$\sum_{k=1}^{\log(n)} c = c \sum_{k=1}^{\log(n)} = c\log(n) \tag{26}$$

Então, obtemos que $T(n) = O(\log(n))$ Num contexto de melhor caso, acharíamos o valor na primeira iteração e, portanto, $T(n) = \Omega(1)$.

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.

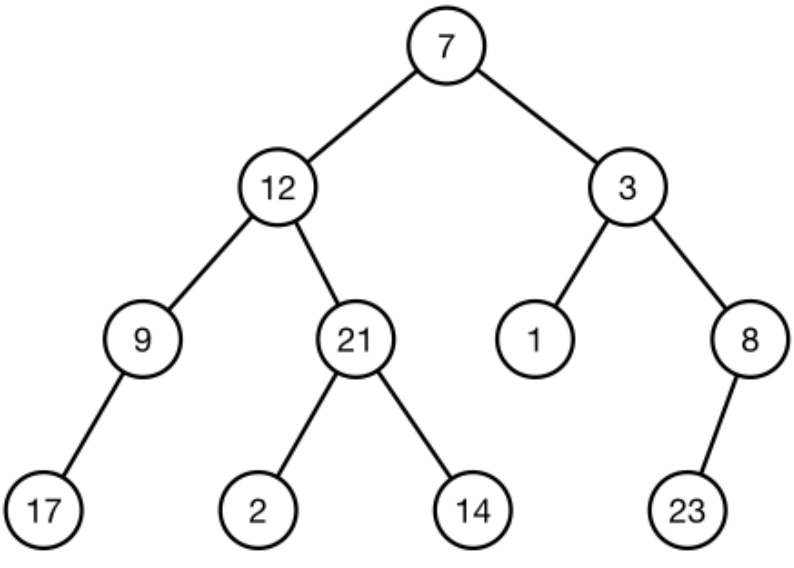


Figura 3: Exemplo de árvore binária

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

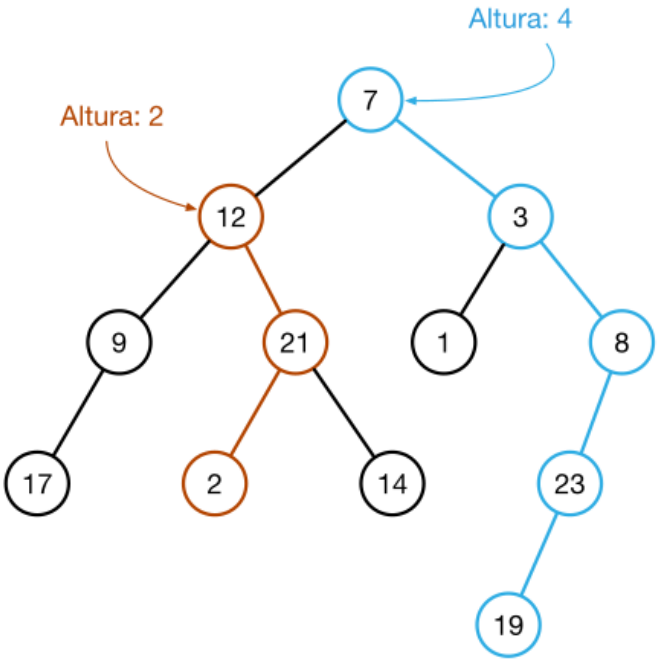


Figura 4: Exemplificação de altura em árvore binária

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

$$\begin{aligned} n_{\min} &= h + 1 \\ n_{\max} &= 2^{h+1} - 1 \end{aligned} \tag{27}$$

Para $n_{\min}$, pense apenas numa lista encadeada. Ela é uma árvore, certo? Ela é o menor caso possível intuitivamente e tem $h + 1$ nós.

Para $n_{\max}$, pense numa árvore completa, claro. Se isso ocorrer, temos 2^0 nós na altura 0, 2^1 nós na altura 1, e assim sucessivamente. Temos então um somatório de nós até a altura h:

$$S_h = \sum_{k=0}^{k=h} 2^k = 1.\frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1 \tag{28}$$

pela forma da soma da PG.

Definição 3.2.2: Uma árvore está **balanceada** quando a altura das subárvores de um nó apresentam 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:

$$h = \log(n) \tag{29}$$

Demonstração:

Seja $N(h)$ o número mínimo de nós de uma árvore balanceada de altura h. Temos a recorrência (pior caso):

$$N(h) = 1 + N(h-1) + N(h-2) \tag{30}$$

O 1 vem da raiz, $N(h-1)$ de alguma das sub-árvores, e $N(h-2)$ vem da outra sub-árvore, com a distância entre as duas de 1.

Ainda, temos que $N(0) = 1$ e $N(1) = 2$,

Hipótese: $N(h) \geq 2^{\frac{h}{2}}$ para todo $h \geq 0$.

Base: Para $h = 0$: $N(0) = 1 \geq 2^0$. Para $h = 1$: $N(1) = 2 \geq 2^{\frac{1}{2}}$.

Passo indutivo: suponha válido para $h-1$ e $h-2$. Então

$$N(h) \geq N(h-1) + N(h-2) \geq 2^{\frac{h-1}{2}} + 2^{\frac{h-2}{2}} = 2^{\frac{h-2}{2}}\left(2^{\frac{1}{2}} + 1\right). \tag{31}$$

Como $2^{\frac{1}{2}} + 1 > 2$, segue $N(h) \geq 2^{\frac{h}{2}}$.

Se a árvore tem n nós então $n \geq N(h) \geq 2^{\frac{h}{2}}$, logo

$$n \geq 2^{\frac{h}{2}} \Rightarrow \log_2(n) \geq \frac{h}{2}\log_2(2) \Rightarrow h \leq 2\log_2(n) \tag{32}$$

Portanto, $h = O(\log n)$. □

Essa hipótese é um pouco não trivial, mas se quiser, também é possível provar reconhecendo que os temos $N(h) = N(h-1) + N(h-2)$ lembram bastante Fibonacci.

Agora, vamos ver algumas formas de andar por essa árvore binária, ou seja, andar corretamente de nó em nó usando a estrutura que criamos, além de formas para descobrir sua altura. 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;
  }

  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 $\Theta(n)$, pois independente da ideia, precisamos passar por todos os nós para termos certeza da altura.

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

Relembrando os 3 casos que você provavelmente já viu em Estrutura de Dados:

- Em ordem: Esquerda -> Raizz -> Direita

EM ORDEM

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

- Pré-ordem : Raiz -> Esquerda -> Direita

PRÉ-ORDEM

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

- Pós - ordem: Esquerda -> Direita -> Raiz

PÓS-ORDEM

```cpp
void printTreeDFSPostorder(class Node * node) {
  if (node == nullptr) {
    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. Imagino que você saiba, mas relembrando, busca em largura nada mais é que a busca de altura em altura, começando pela raiz.

C++

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

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

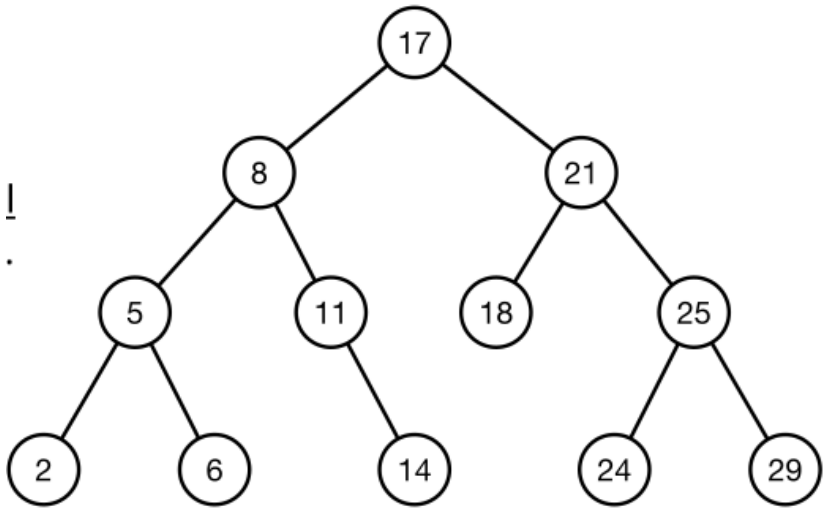


Figura 5: Exemplo de árvore binária de busca(ordenada)

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 $\Theta(h)$ no pior caso.

BUSCA EM ÁRVORE BINÁRIA (ITERATIVO) C++

```cpp
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**

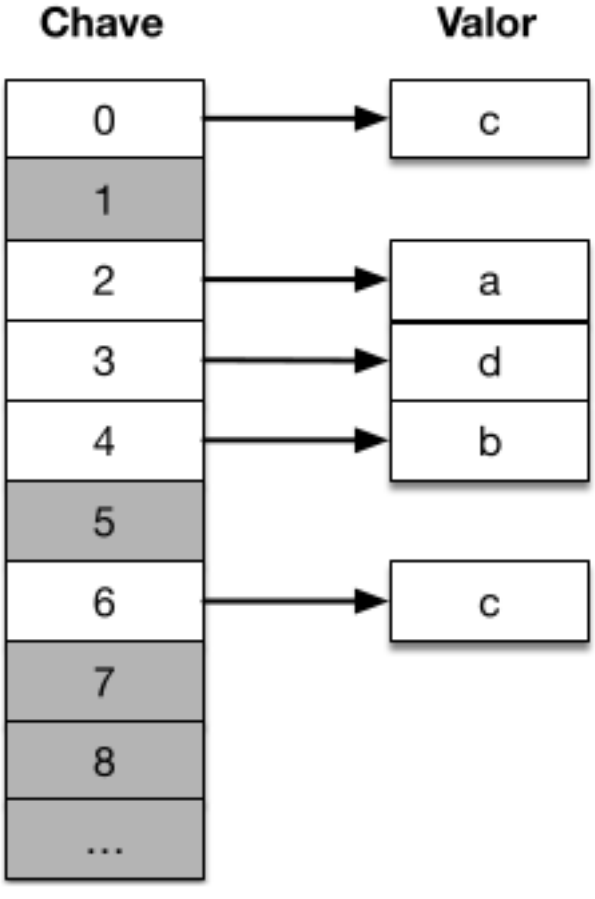


Figura 6: Exemplificação do algoritmo de tabela hash

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.

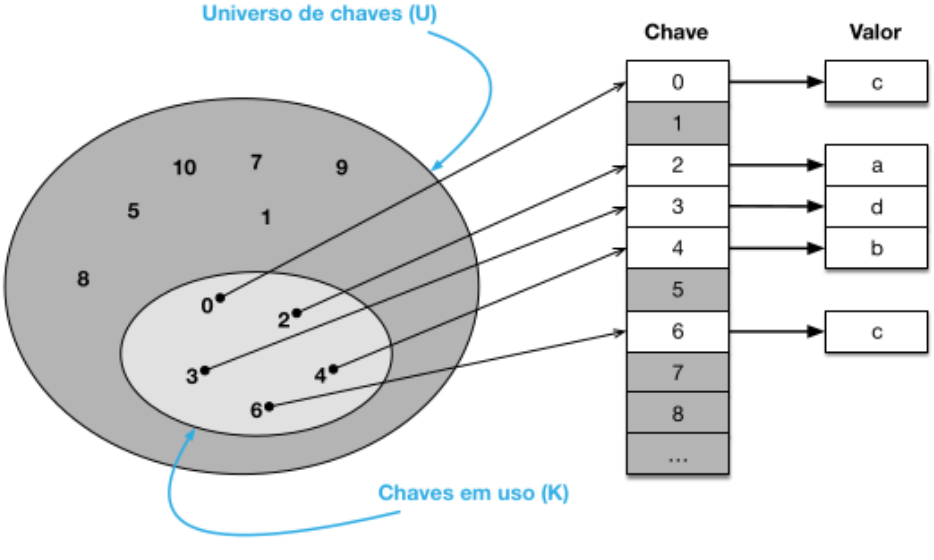


Figura 7: Estruturação da Hash Table

- **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 Desafio

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

C++

```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

C++

```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;
```

```
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 $\Theta(n)$ no pior caso. Como melhorar isso?

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, pelo princípio da casa dos pombos, elas são inevitáveis, pois quase sempre existem mais elementos do que chaves).

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.

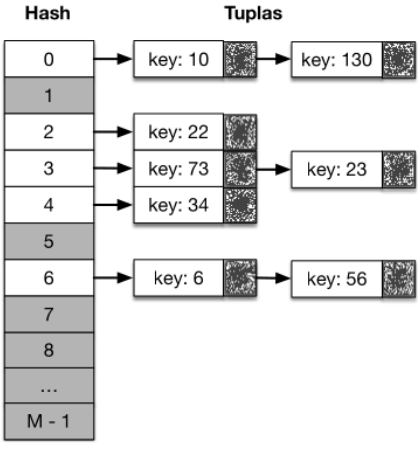


Figura 8: Tabela hash com encadeamento

EXEMPLO DE IMPLEMENTAÇÃO C++

```
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**: $\Theta(n)$

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)` $= \lfloor \text{key} \cdot M \rfloor$
  - Exemplo: Suponha M = 10, então teremos $0, ..., 9$ hashes:
    - chave $= 0,27 \Rightarrow 0,27.10 = 2,7 \Rightarrow \lfloor 2.7 \rfloor = 2$
    - chave $= 0,92 \Rightarrow 0,92.10 = 9,2 \Rightarrow \lfloor 9.2 \rfloor = 9$
- **Método da divisão**
  - Se a chave for um número inteiro
  - `hash(key)` $= \text{key}\%M$
  - Costuma-se definir M como um número primo.
  - Exemplo: Suponha M = 23, logo, temos 23 hashes.
    - chave $= 14 \Rightarrow 14\%23 = 14$
    - chave $= 35 \Rightarrow 35\%23 = 12$
- **Método da multiplicação**
  - `hash(key)` $= \lfloor M.((\text{key} \cdot A)\%1) \rfloor$
  - A é uma constante no intervalo $0 < A < 1$.
  - Exemplo: Suponha M $= 10$, A $= 0,618$, 100 hashes
    - chave $= 123 \Rightarrow 123.0,618 = 76,014 \Rightarrow 76,014\%1 = 0,014 \Rightarrow \lfloor 0.014 * 100 \rfloor = \lfloor 1.4 \rfloor = 1$

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;
}
```

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, e esperamos que, escolhendo uma função de espalhamento que espalhe os valores uniformemente, a **complexidade média** do tempo de busca fica $\frac{n}{m}$. 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:

$$T(n) = \frac{1}{n}\sum_{i}^{n}\left(1 + \sum_{j=i+1}^{n}\frac{1}{m}\right) = \Theta\left(1 + \frac{n}{m}\right) \tag{33}$$

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 inserimos elementos na tabela o desempenho vai se degradando, e calculando $\alpha = n/m$ a cada inserção conseguimos calcular se a tabela está em um estado ineficiente, e quando a considerarmos ineficiente, teremos então que fazê-la ficar eficiente novamente, mas como? Redimensionando-a.

A operação de redimensionamento aumenta o tamanho do vetor de m para M', porém, isso invalida o mapeamento das chaves anteriores, já que a métrica era feita especificamente para o tamanho anterior . 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 $M' = 2M$

$$\frac{n \cdot \Theta(1) + \Theta(n)}{n} = \Theta(1) \tag{34}$$

Esse n no denominador vem exatamente da amortização da análise, n é o número de elementos inseridos.

Exemplo de Análise Amortizada

Vamos considerar a inserção de $n = 8$ elementos em uma tabela hash que dobra de tamanho sempre que enche.

Inicialmente $m = 1$, e os redimensionamentos ocorrem da seguinte forma: $1 \to 2 \to 4 \to 8$.

1. Inserir o 1° elemento $\to$ custo 1.
2. Inserir o 2° elemento $\to$ tabela cheia, redimensiona para 2 e re-hash de 1 elemento. Custo: 1 (re-hash) + 1 (inserção) = 2.

3. Inserir o 3° elemento $\rightarrow$ tabela cheia, redimensiona para 4 e re-hash de 2 elementos. Custo: 2 (re-hash) + 1 (inserção) = 3.
4. Inserir o 4° elemento $\rightarrow$ custo 1.
5. Inserir o 5° elemento $\rightarrow$ redimensiona para 8 e re-hash de 4 elementos. Custo: 4 (re-hash) + 1 (inserção) = 5.
6. Inserir o 6° elemento $\rightarrow$ custo 1.
7. Inserir Inserir o 7° elemento $\rightarrow$ custo 1.
8. Inserir o 8° elemento $\rightarrow$ custo 1.

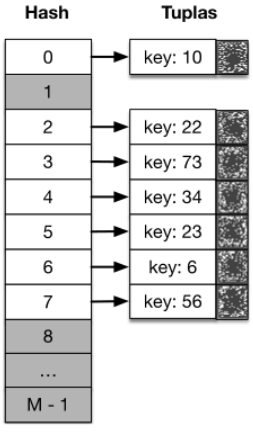
- Inserções sem redimensionamentos: 5 operações (1, 4, 6, 7, 8)
- Redimensionamentos: $2 + 3 + 5 = 10$.
- Custo total: 15.

Portanto, foram $n = 8$ inserções no total. O custo amortizado é dado por:

$$\frac{\text{Custo total}}{\text{inserções}} = \frac{15}{8} = 1.875 \approx \Theta(1) \tag{35}$$

Assim, mesmo com redimensionamentos custosos, o custo médio por operação permanece constante.

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,ao inserir um elemento y na tabela, se ele tem o mesmo hash do elemento x (que já está inserido na tabela), basta inserir num 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) {
```

```
6         m_table = new DANode[size];
7         for (int i=0; i < m_size; i++) {
8           m_table[i].key = -1;
9           m_table[i].value = 0;
10        }
11      }
12      ~DirectAddressHashTable() { delete[] m_table; }
13
14    private:
15      unsigned hash(int key) const { return key % m_size; }
16
17      DANode * m_table;
18      int m_size;
19  };
20
21
22  bool insert_or_update(int key, int value) {
23    unsigned h = hash(key);
24    DANode * node = nullptr;
25    int count = 0;
26    for (; count < m_size; count++) {
27      node = &m_table[h];
28      if (node->key == -1 || node->key == key) {
29        break;
30      }
31      h = (h + 1) % m_size;
32    }
33    if (count >= m_size) {
34      return false; // Table is full
35    }
36    if (node->key == -1) {
37      node->key = key;
38    }
39    node->value = value;
40    return true;
41  }
42
43
44  DANode * search(int key) {
45    unsigned h = hash(key);
46    DANode * node = nullptr;
47    int count = 0;
48    for (; count < m_size; count++) {
49      node = &m_table[h];
50      if (node->key == -1 || node->key == key) {
51        break;
52      }
53      h = (h + 1) % m_size;
54    }
55    return count >= m_size || node->key == -1 ? nullptr : node;
56  }
57
```

```
58
59 bool remove(int key) {
60   DANode * node = search(key);
61   if (node == nullptr) {
62     return false;
63   }
64   node->key = -1;
65   node->value = 0;
66   return true;
67 }
```

Porém, a remoção em uma tabela hash com endereçamento aberto também 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 o algoritmo procura partindo de h_0, como h está vazio, interpretará que não precisa continuar a busca, porque ele não sabe que a key da posição h foi removida.

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 erro possível no uso de endereçamento aberto

No exemplo acima, perceba que tínhamos um hash uniforme simples, com o hash = $\text{key}\ \%M$, e provavelmente a sequência de ordenação como

$$...131 \rightarrow 33 \rightarrow 91 \rightarrow 76 \rightarrow 61 \rightarrow ... \tag{36}$$

e que, logo após, removemos o número 131. Depois, buscamos os valores 91 e 61, mas não os encontramos, pois o primeiro slot onde eles se encaixariam(o do 131) está vazio. Por isso, o algoritmo para e retorna que eles não estão na lista(por isso estão acinzentados).

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.

```
1 typedef struct DirectAddressHashTableNode DANode;                C++
2 struct DirectAddressHashTableNode {
3   int key;
4   int value;
5   bool recycled;
6 };
```

- 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,podíamos ter mais chaves do que espaços no vetor).
- A busca por uma determinada chave depende da sequência de sondagem `hash(key, i)` fornecida pela função de espalhamento. (i é o número da iteração da sondagem).
  - Exemplo linear: `hash(key, i) = (hash'(key) + i) mod M`
    - `hash(key, 0) = hash'(key)` → `hash(key, 1) = (hash'(key) + 1) mod M`

Note que `hash(key,i)` é a função de sondagem completa, que depende tanto da chave quanto da tentativa, enquanto `hash'(key)` é a função de espalhamento base, ou seja, a posição inicial da chave antes da colisão

- Observe que existem M! permutações possíveis para a sequência de sondagem(em geral isso não importa muito).

Porém, a abordagem linear rapidamente se torna ineficaz, já que em determinado momento o problema se transforma basicamente em inserir elementos em uma lista. Temos, por isso, outras alternativas:

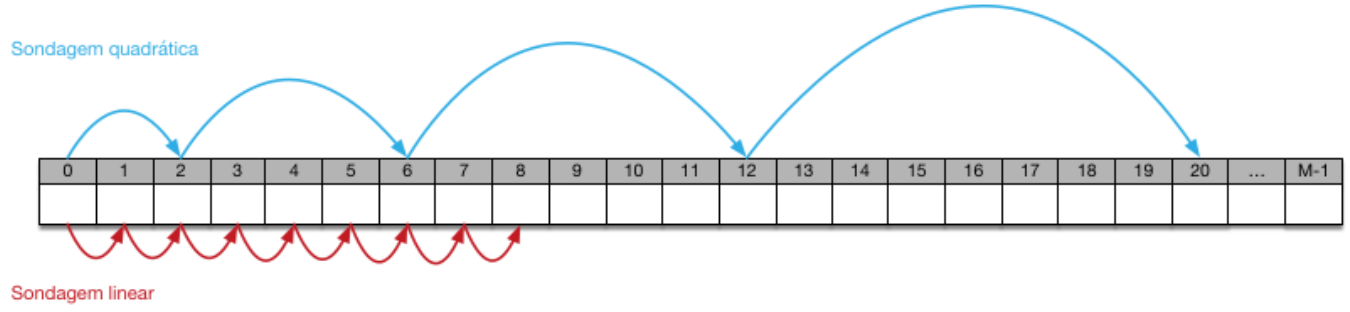


Figura 11: Exemplificação do endereçamento aberto usando de sondagem quadrática

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

Exemplo:

- Tamanho da tabela: $M = 11$
- Função de hash base: `hash'(key) = key mod M`
- Parâmetros: `a = 1`, `b = 0`

Sequência de sondagem quadrática:

```
hash(27, 0) = (5 + 0 * 0 + 1 * 0) mod 11 = 5
hash(27, 1) = (5 + 0 * 1 + 1 * 1) mod 11 = 6
hash(27, 2) = (5 + 0 * 2 + 1 * 4) mod 11 = 9
hash(27, 3) = (5 + 0 * 3 + 1 * 9) mod 11 = 3
hash(27, 4) = (5 + 0 * 4 + 1 * 16) mod 11 = 10
```

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 (podemos inserir outras abordagens).

Podemos introduzir o **hash duplo**, tal que temos **duas** funções de hash diferentes hash_1 e hash_2 de forma que o novo hash de uma chave será dado por: `hash(key, i) = (hash1(key) + i * hash2(key)) % M'`. Dessa forma, mesmo que uma mesma chave colida com outra na primeira função de hash, a segunda função garante que cada tentativa subsequente irá gerar um novo índice diferente, distribuindo melhor as chaves na tabela.

Exemplo:

- Tamanho da tabela: $M = 11$
- Funções de hash:
  - `hash1(key) = key mod 11`
  - `hash2(key) = 1 + (key mod (M - 1))`

Sequência de sondagem com hash duplo:

```
hash(27, 0) = (5 + 0 * 8) mod 11 = 5
hash(27, 1) = (5 + 1 * 8) mod 11 = 2
hash(27, 2) = (5 + 2 * 8) mod 11 = 10
hash(27, 3) = (5 + 3 * 8) mod 11 = 7
hash(27, 4) = (5 + 4 * 8) mod 11 = 4
```

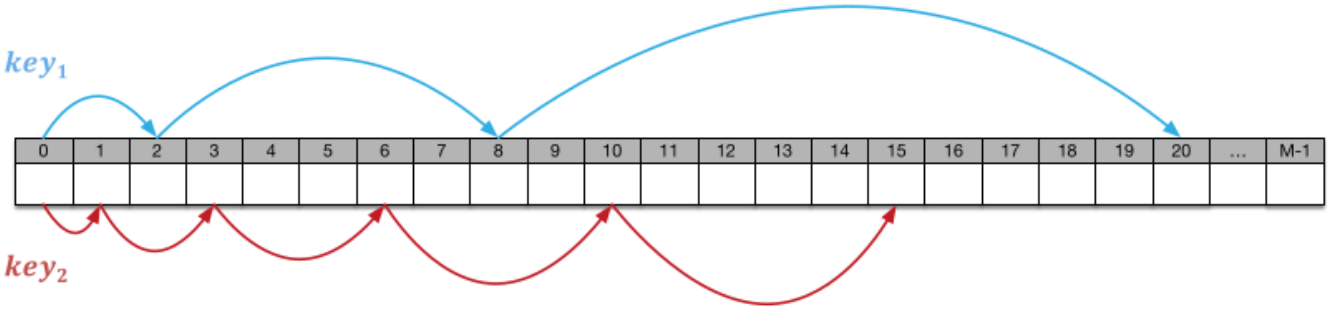


Figura 12: Exemplificação do endereçamento aberto usando hash duplo

Porém, vale ressaltar que a segunda função de hash deve:

- Ser completamente diferente da primeira
- Não retornar 0

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

$$T(n) = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} = O(1) \tag{37}$$

também pela forma da PG.

Algoritmos de Ordenação

Agora, 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:

8.7	11.4	2.1	5.5	11.3	2.7	7.1	5.3	9.0	4.8

2.1	2.7	4.8	5.5	5.3	7.1	8.7	9	11.4	11.3

Figura 13: Exemplo de algoritmo estável 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, como podemos ver no array ordenado, a ordem deverá ser mantida.

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 $0 \leq i < n - 1$ 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:

8	11	2	5	1

$$n = 5$$
$$j = n - 1 = 4$$

Figura 14: Lista para exemplo do algoritmo Bubble Sort

E a execução do código decorrerá da forma:

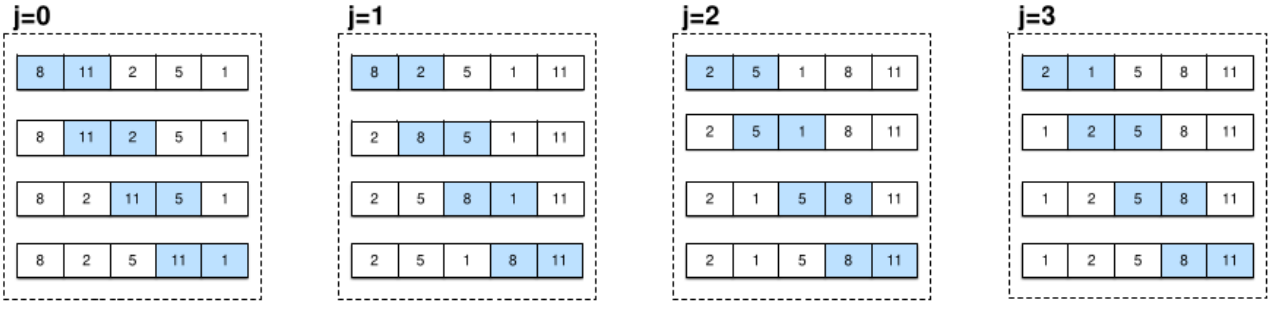


Figura 15: Fluxo de código do algoritmo Bubble Sort

Então, a ideia é percorrer cada item da lista e, sempre que um elemento a esquerda é maior que o elemento a direita, 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 algoritmo é executado $n-1$ vezes e, a cada iteração do for de fora, ele executa $n-1$ subprocessos, logo, no final teremos um total de $T(n) = \Theta(n^2)$ de complexidade (tanto melhor quanto pior caso, já que independentemente da lista os dois fors vão até $n-1$).

Porém, fazendo uma otimização no algoritmo:

IMPLEMENTAÇÃO OTIMIZADA 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 do loop maior houve alguma troca, se não houve nenhuma, então o algoritmo é encerrado, pois significa que está ordenado. 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

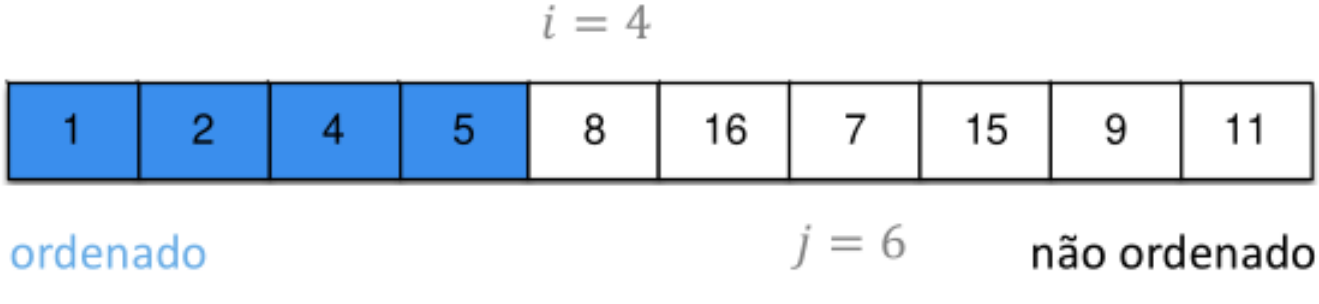


Figura 16: Exemplificação do algoritmo Selection Sort

Dada uma posição i, e assumindo que todas as posições anteriores já estão ordenadas, o algoritmo irá procurar dentre as próximas $n - i$ posições um valor menor que o da posição i. Se isso acontece, significa que esse valor deveria estar na posição que i, e então trocamos de posição.

Exemplo: Considere o caso:

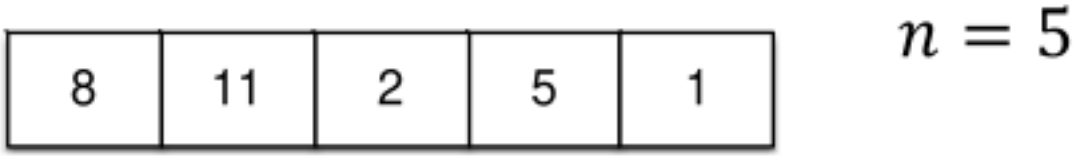


Figura 17: Lista para exemplo do algoritmo Selection Sort

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

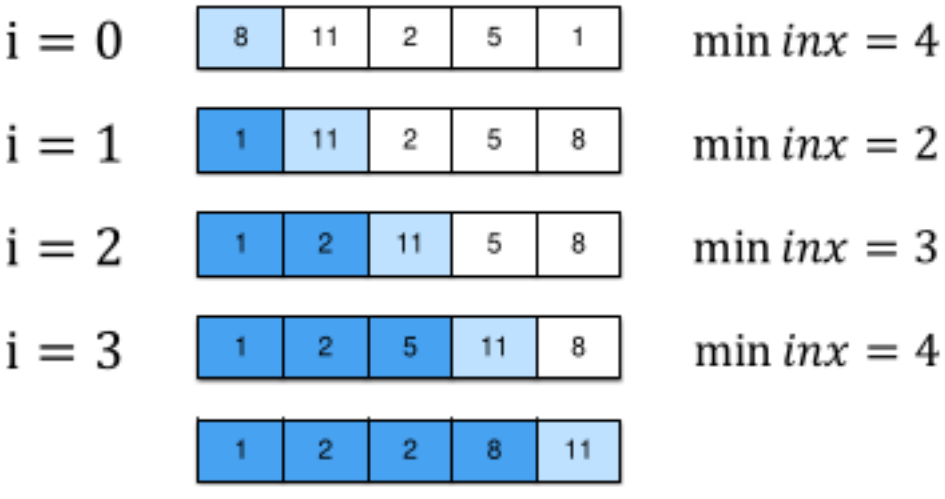


Figura 18: Fluxo do código do algoritmo Selection Sort

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);
  }
}
```

No primeiro for, pegamos o índice `i`, e no segundo loop passamos em todos os índices a frente de `i`, e se for menor que o `v[i]`(ou algum que foi substituído), realiza a troca depois de todas as verificações. Dessa forma, sempre pegamos o menor valor de da lista do índice `i` para frente.

Para avaliar o desempenho, podemos montar seu custo total percebendo que, a cada iteração, o algoritmo avalia um elemento a menos, de forma que podemos expressar a **função de complexidade** como:

$$T(n) = (n-1) + (n-2) + ... + 1 + 0$$
$$= \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{38}$$

Ou seja, obtemos que $T(n) = \Theta(n^2)$, que também é a complexidade no melhor caso, já que, novamente, os fors dependem totalmente de n.

5.3 Insertion Sort

Parecido com o algoritmo de **Selection Sort**, que acabamos de ver. Porém, a ideia é fixar uma posição i e avaliar o valor naquela posição, procurando dentre as posições $[0, i-1]$ qual deveria ser a posição que o valor da posição i deveria estar:

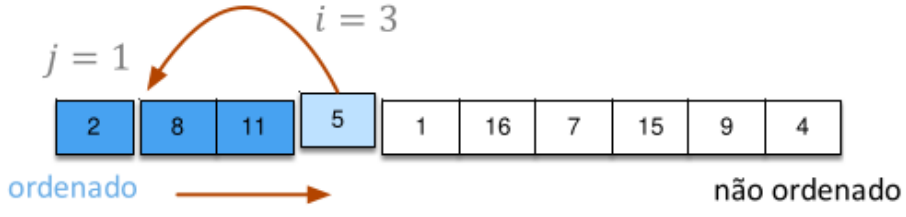


Figura 19: Exemplificação do algoritmo Insertion Sort

Exemplo:

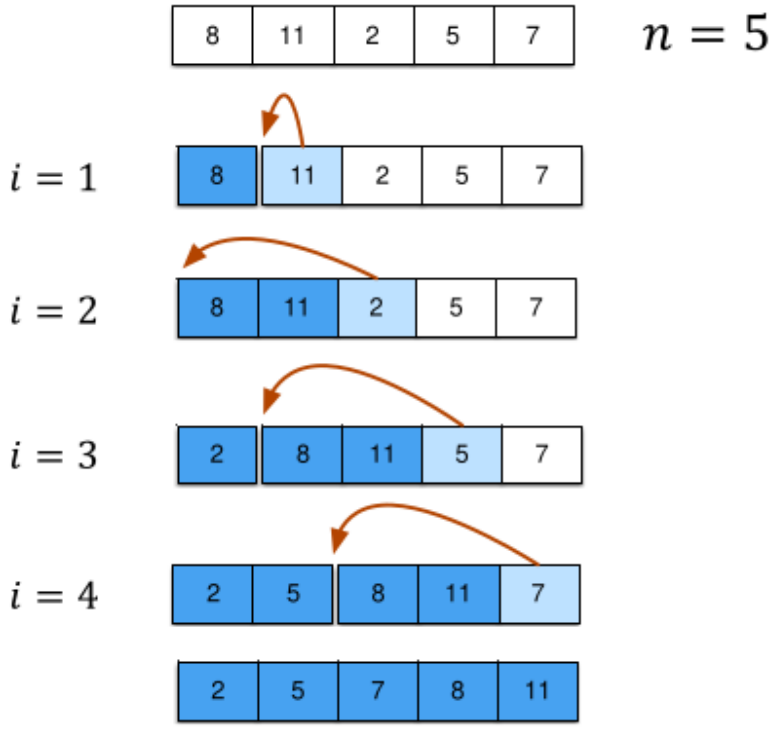


Figura 20: Exemplo do algoritmo Insertion Sort

IMPLEMENTAÇÃO C++

```
1  void insertionSort(int v[], int n) {
2    for (int i = 1; i < n; i++) {
3      int currentValue = v[i];
4      int j;
5      for (j = i - 1; j >= 0 && v[j] > currentValue; j--) {
6        v[j + 1] = v[j];
7      }
```

```
8       v[j + 1] = currentValue;
9     }
10  }
```

O loop externo começa no segundo elemento porque o primeiro já forma uma sublista ordenada. A cada iteração, o valor `v[i]` é guardado em `currentValue`. O loop interno percorre da direita para a esquerda os elementos da sublista ordenada, deslocando todos os valores maiores que `currentValue` uma posição à direita. O algoritmo para quando encontramos um elemento menor ou igual a `currentValue` ou quando chegamos ao início do vetor. Assim, a posição `j+1` é o local correto para inserir o `currentValue`, garantindo que, ao final da iteração, os elementos de `v[0..i]` estejam ordenados. A complexidade desse algoritmo também é expressa na forma:

$$T(n) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2) \tag{39}$$

já que o loop de dentro apresenta um range parecido com o do algoritmo Selection Sort. Perceba que, no melhor caso, $T(n) = \Theta(n)$, pois o loop de dentro sempre será quebrado em $O(1)$.

5.4 Mergesort

A ideia do algoritmo consiste em dividir a sequência em duas partes, executar chamadas recursivas para cada sub-sequência, até que o tamanho das sequências sejam tão pequenas que caiam no caso trivial de se ordenar, e juntá-las (merge) de forma ordenada. Esse algoritmo depende de um algoritmo auxiliar de intercalação (merge)

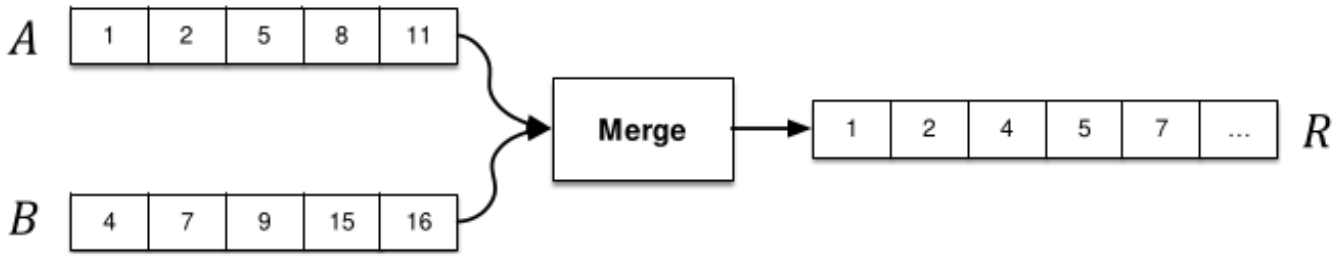


Figura 21: Exemplificação do algoritmo Mergesort

FUNÇÃO DE INTERCALAÇÃO C++

```
1   void merge(int v[], int startA, int startB, int endB) {
2     int r[endB - startA];
3     int aInx = startA;
4     int bInx = startB;
5     int rInx = 0;
6     while (aInx < startB && bInx < endB) {
7       if (v[aInx] <= v[bInx]) {
8         r[rInx++] = v[aInx++];
9       } else {
10        r[rInx++] = v[bInx++];
11      }
12    }
13    while (aInx < startB) {
14      r[rInx++] = v[aInx++];
15    }
16    while (bInx < endB) {
17      r[rInx++] = v[bInx++];
```

```
18    }
19    for (aInx = startA; aInx < endB; ++aInx) {
20      v[aInx] = r[aInx - startA];
21    }
22  }
```

Cria-se um vetor r do tamanho da lista passada antes da separação, e definimos alguns inteiros para não alterarmos os tamanhos originais e conseguirmos saber o que estamos fazendo com a lista. Sabemos que no vetor `v`, a parte `startA` até `startB - 1` está ordenada corretamente, e o mesmo vale para `startB` até `endB`.

O primeiro while serve para usar a ordem criada nas duas subsequências a nosso favor, ou seja, verificamos até alguma das duas chegar em seu tamanho final e, enquanto isso não acontece, comparamos cada elemento inicial de cada sub-sequência, e sabendo que estão ordenadas, não precisamos verificar outros elementos. O vetor `r` fica completamete ordenado, mas em caso de alguma contagem de índice(`aIdx` ou `bIdx`) acabar antes de outra, significa que alguns elementos do outro índice não foram passados para `r` ainda, e é para isso que servem os outros dois whiles no final. Por fim, o último for serve apenas para passar os elementos na ordem correta de `r` para o vetor original e ainda não ordenado `v`.

Os 3 whiles somam n operações, e o for final outras n. Logo, a complexidade é

$$T(n) = \Theta(n) \tag{40}$$

Agora que temos uma função que junta duas listas ordenadamente, conseguimos fazer o mergeSort. O algoritmo consiste em dividir a sequência do vetor V em duas subsequências A e B, fazendo isso recursivamente até que V esteja ordenado.

IMPLEMENTAÇÃO C++

```
1 void mergeSort(int v[], int startInx, int endInx) {
2   if (startInx < endInx - 1) {
3     int midInx = (startInx + endInx) / 2;
4     mergeSort(v, startInx, midInx);
5     mergeSort(v, midInx, endInx);
6     merge(v, startInx, midInx, endInx);
7   }
8 }
```

Olhando o algoritmo, note que ele chama a recursão do mergeSort até que a diferença entre os dois index sejam um, portanto essa recursão acontece até que tenhamos n listas de tamanho 1, e o merge fará todo o trabalho de “ordenar”. Olhe o exemplo:

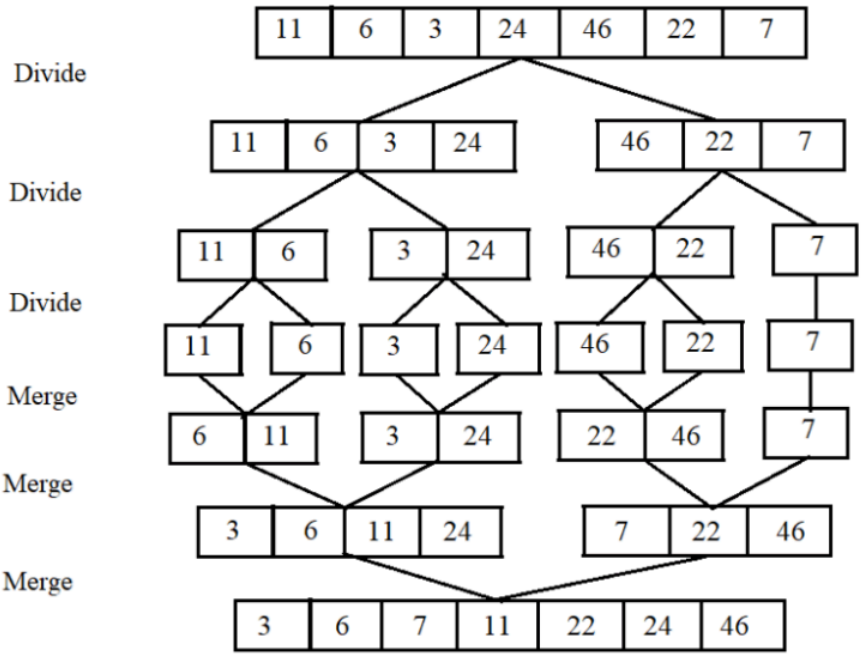


Figura 22: Exemplo do algoritmo MergeSort

Podemos então avaliar a função de complexidade:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \tag{41}$$

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

É uma ideia parecida com o 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 separa em duas partições: Os elementos maiores e menores que o **pivô**.

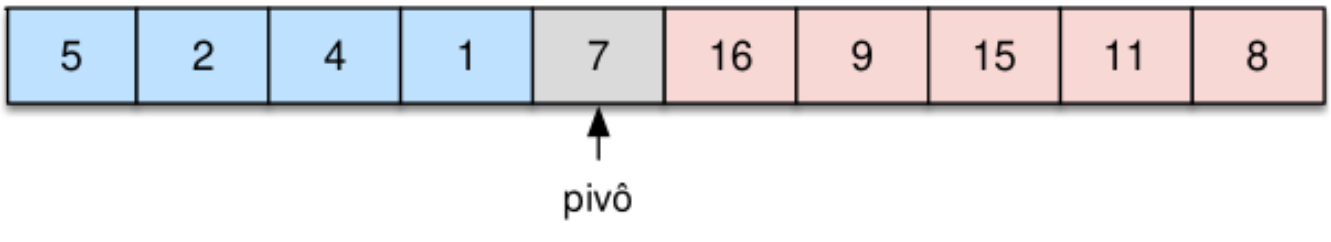


Figura 23: Exemplificação do algoritmo Quicksort

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:

$$v[p, ..., j-1] \leq v[j] \leq v[j+1, ..., r] \tag{42}$$

Temos a seguinte implementação para o partition:

IMPLEMENTAÇÃO C++

```c
#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;
}
```

Olhando para o algoritmo temos `r`, que é o índice da lista que vamos ordenar em função, temos também `p`, que é o índice de onde vamos começar a ordenar, e `j`, que será a quantidade a partir de `p` de elementos menores que `v[r]`. No caso, ordenaremos para a sublista `v[p, ..., r]` em função de `v[r]`.

Fazemos um for de `p` até `r`, e se `v[i]` for menor que o pivô, trocamos o elemento indexado em `i` com o em `j`. Como `j` só é incrementado quando acha um valor menor, então o que estamos fazendo é separando uma área para os elementos menores que `v[r]` enquanto deixamos que os maiores continuem em suas posições(a menos de troca com menores). No final, trocamos o `v[r]` com a última incrementação de `j`, deixando menores a esquerda e maiores a direita. Retorna a posição correta de `v[r]`.

Como a sequência de n(a real é que a sequência é definida por `p` e `r`, mas normalmente esses valores serão o início e o fim da lista, respectivamente) elementos é percorrida uma única vez executando operações constantes, temos que $T(n) = \Theta(n)$

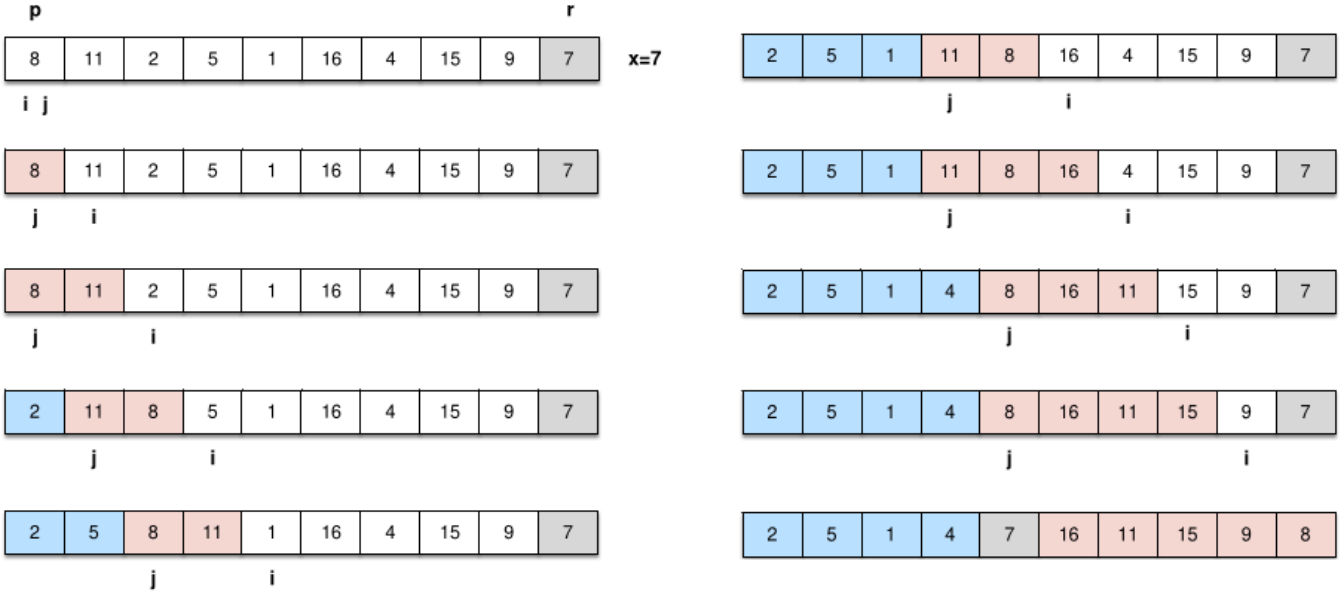


Figura 24: Exemplo do algoritmo Partition

Agora que temos a base, vamos para o algoritmo principal!

IMPLEMENTAÇÃO C++

```c
void quicksort(int v[], int p, int r) {
  if (p < r) {
```

```
3      int j = partition (v, p, r);
4      quicksort(v, p , j - 1);
5      quicksort(v, j + 1, r);
6    }
7  }
8  quicksort(v, 0, n - 1);
```

Funciona da seguinte forma: enquanto tivemos mais de um elemento na lista(isso que o if verifica), particionamos em cima de algum elemento da lista, e fazemos a mesma coisa recursivamente para a parte a esquerda e a direita da lista, com o elemento de j já ordenado.

Note que, se sempre pegarmos um elemento do muito ruim, a complexidade do algoritmo irá aumentar. Por isso, temos algumas formas de escolher o elemento para ordenar em função:

1. Podemos permutar a entrada do vetor;
2. Sortear algum índice aleatório.

Dessa forma aumentamos a probabilidade da partição ser razoavelmente equilibrada na média.

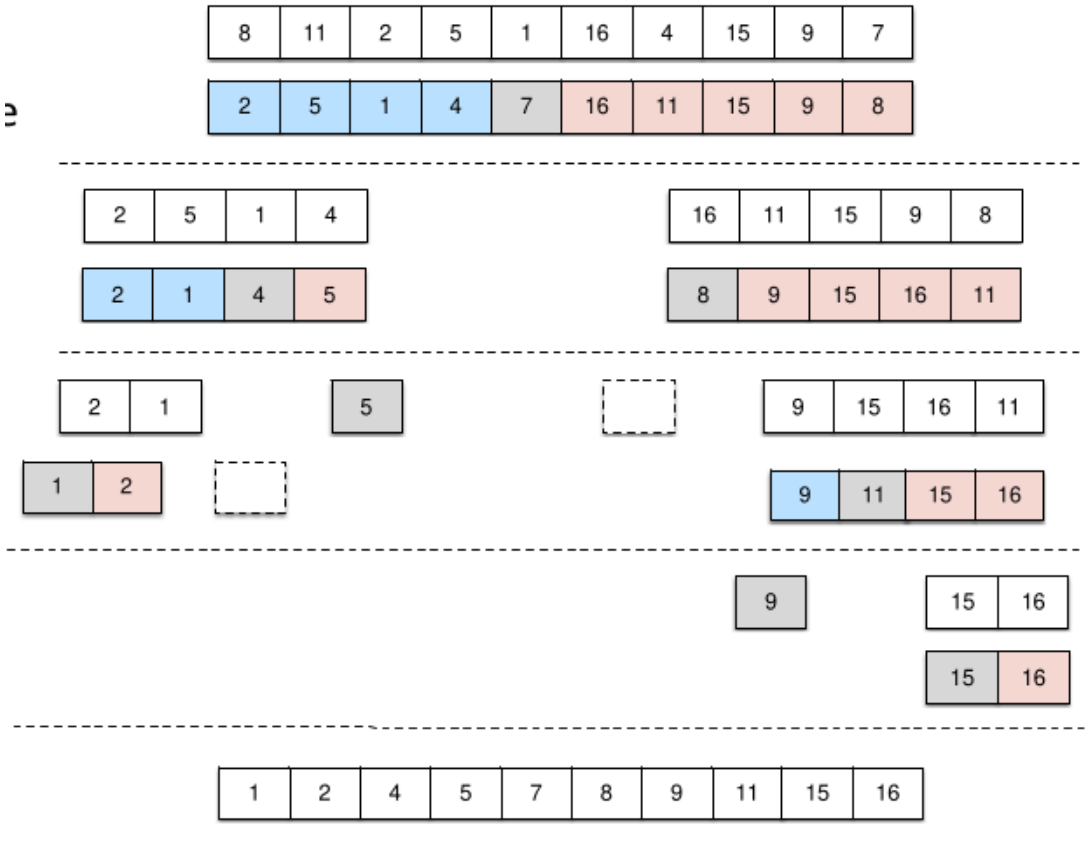


Figura 25: Exemplo do algoritmo quicksort

Temos que sua função de complexidade é tal que:

$$\begin{aligned} T(n) &= T(j) + T(n-j-1) + n \\ &= T(0) + 1 + 2 + 3 + ... + (n-1) + n \\ &= \frac{n(n+1)}{2} \\ &= \Theta(n^2) \end{aligned} \tag{43}$$

onde j é o índice do primeiro elemento particionado.

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

$$T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n\log(n)) \tag{44}$$

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 $0.9n$

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n \tag{45}$$

Podemos avaliar como $\Theta(n\log(n))$, ou seja, é o mesmo caso do melhor caso possível, mas tem uma constante maior. Então, 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 $2p + 1$

Essa abordagem de implementação elimina a necessidade de ponteiros para o pai e para os filhos.

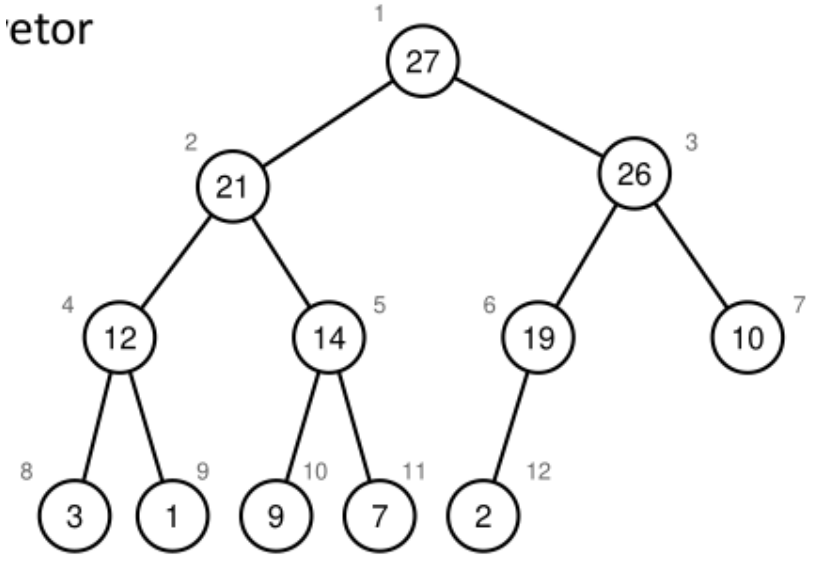


Figura 26: Exemplo de HEAP

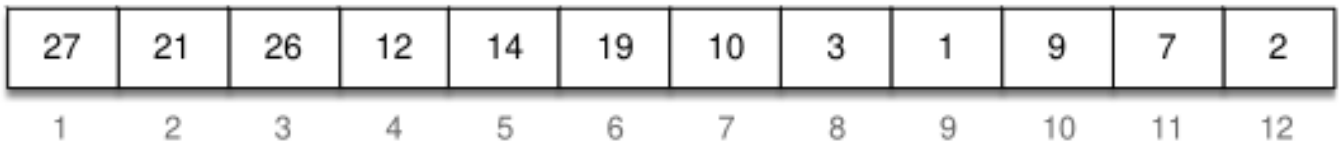


Figura 27: Forma da Figura 26 como vetor

Podemos ordenar uma árvore em um heap caso as propriedades do vetor não sejam satisfeitas. Para isso, utilizamos o algoritmo **max-heapify**

- Assume-se que as sub-árvores do nó são heaps-máximos(ou mínimos no outro caso).
- 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.

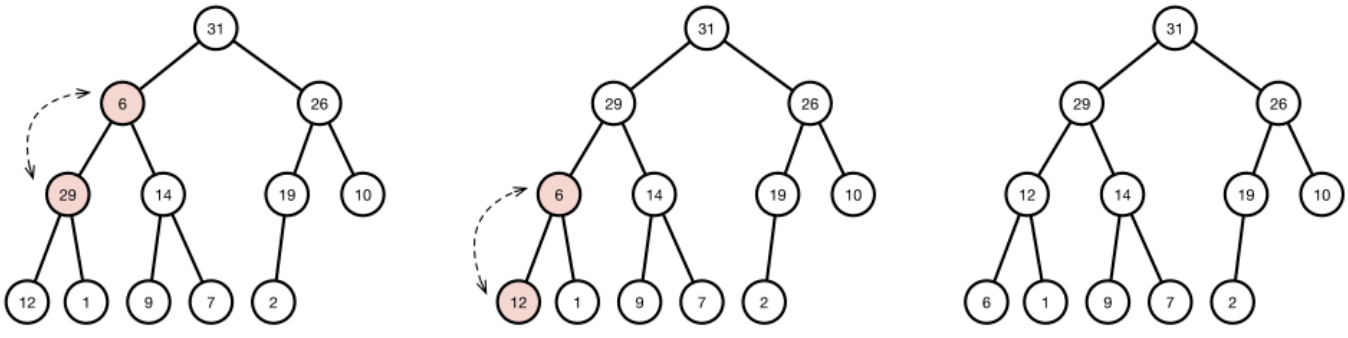


Figura 28: Visualização do algoritmo max-heapify

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);
  }
}
```

Os índices da esquerda e da direita são criados como $2 * i + 1$ e $2 * i + 2$, em vez de $2 * i$ e $2 * i + 1$ por causa da indexação inicial de vetores nas linguagens. o inteiro `i` é o índice do nó que queremos corrigir.

Na primeira verificação vemos se o índice a esquerda que calculamos existe(sendo menor que n) e se o filho a esquerda do elemento `i` é maior. Fazemos a mesma coisa só que com o filho a direita de `v[i]`(note que a verificação do da direita compara não só com o pai, mas também com o filho a esquerda).

Se o índice mudou, ou seja, se algum filho é maior que o pai, então trocamos o pai pelo maior filho e fazemos heapify novamente.

A complexidade desse algoritmo é $T(n) = O(\log n)$, pela propriedade do heap que é criado como uma árvore quase completa, com a altura crescendo proporcionalmente com o número de elementos.

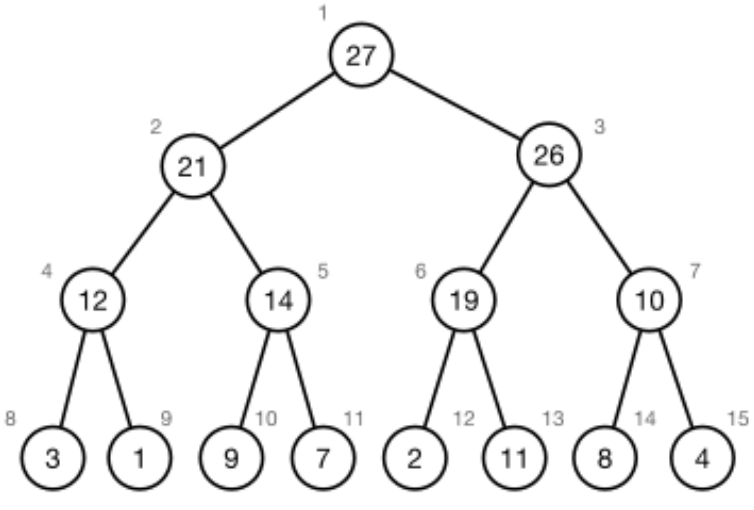
Agora, vamos aprender a construir esse heap:

IMPLEMENTAÇÃO C++

```
1 void buildHeap(int v[], int n) {
2   for (int i=(n/2-1); i >= 0; i--) {
3     heapify(v, n, i);
4   }
5 }
```

Bom, não é nada muito difícil. Todos os nós depois de $\frac{n}{2}$ são folhas, logo, como o heapify garante a propriedade de heap para o nó `i` e sua subárvore, "afundando" o valor de `v[i]` se necessário, fazemos um for que ordenará desde a raiz até o último pai, garantindo a propriedade do heap por construção.

Vamos ver a complexidade do buildHeap:



nível	# nós
4	$\sim\frac{n}{2^4}$
3	$\sim\frac{n}{2^3}$
2	$\sim\frac{n}{2^2}$
1	$\sim\frac{n}{2^1}$

Figura 29: Aproximação de um algoritmo heapsort

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

$$T(n) = \sum_{i=1}^{\log(n)} i \cdot \frac{n}{2^i} = O(n) \tag{46}$$

pois o somatório converge. Agora, 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

Dessa forma ignoramos a parte já ordenada que colocamos de `v[i]` para frente e fazemos heapify com o restante da lista.

IMPLEMENTAÇÃO C++

```
1 void heapSort(int v[], int n) {
```

```
2   buildHeap(v, n);
3   for (int i=n-1; i > 0; i--) {
4     swap(v, 0, i);
5     heapify(v, i, 0);
6   }
7 }
```

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 $T(n) = \Theta(n\log(n))$.

5.7 Counting Sort

O algoritmo de ordenação por contagem consiste em computar para cada elemento quantos elementos menores existem na lista, pois, sabendo que o elemento v_i possui j elementos menores do que ele, podemos definir sua posição final como $j+1$

Vamos enunciar novamente nosso problema:

Desejamos ordenar os elementos do vetor $v[0,...,n-1]$ considerando as seguintes restrições:

- Os elementos são números inteiros.
- Os números estão presentes no intervalo $[0,...,k-1]$
- O universo possui tamanho k.
- k é pequeno

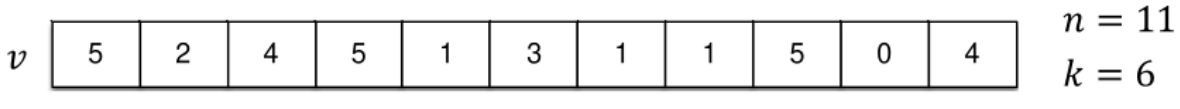


Figura 30: Array de Exemplo

Nesse exemplo, temos um total de 11 elementos e 6 opções entre eles (os elementos vão de 0 à 5)

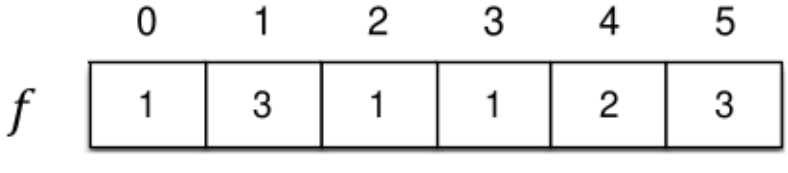


Figura 31: Array de frequência de cada número f

Criamos então um a **sequência auxiliar** de tamanho k. Nessa sequência, cada índice representa um elemento específico do array e os elemento representam **quantas vezes esses elementos aparecem na lista original**. Com essa sequência f, vamos gerar **outra** sequência auxiliar sf, de tal forma que:

$$sf_i = \sum_{j=0}^{i-1} f[j] = f[i-1] - sf[i-1] \qquad (i > 0) \tag{47}$$

Ou seja, o elemento i de sf é a **quantidade de elementos menores que** i

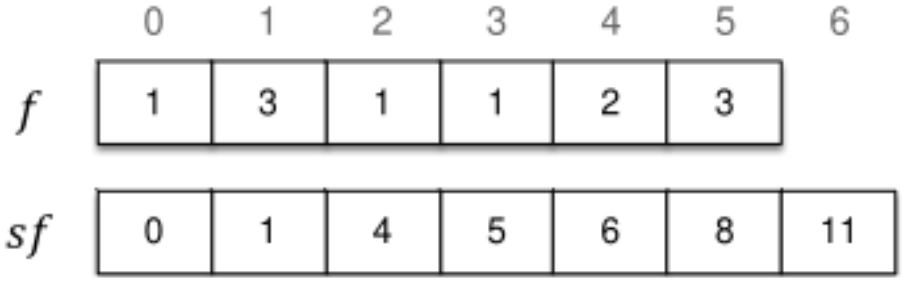


Figura 32: Segunda lista auxiliar sf

Então, utilizando sf, podemos criar uma nova lista ordenada, de forma que o elemento $i \in [0, k]$ estará localizado no índice sf_i

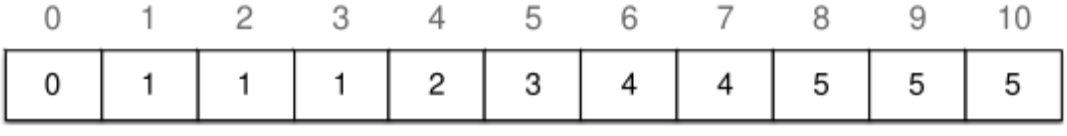


Figura 33: Lista ordenada

IMPLEMENTAÇÃO C++

```cpp
void countingSort(int v[], int n, int k) {
  int fs[k + 2];                //+1 indexação de arrays +1  de fs[0] = 0
  int temp[n];
  for (int j = 0; j < k + 2; j++) {
    fs[j] = 0;
  }
  for (int i = 0; i < n; i++) {
    fs[v[i] + 1] += 1;            //note que fs[0] = 0
  }
  for (int j = 1; j <= k; j++) {
    fs[j] += fs[j - 1];
  }
  for (int i = 0; i < n; i++) {
    int j = v[i];
    temp[fs[j]] = v[i];
    fs[j]++;
  }
  for (int i = 0; i < n; i++) {
    v[i] = temp[i];
  }
}
```

Criamos o vetor da soma de frequências com $k + 1$ entradas, e o vetor temp, que servirá para ordenar depois das frequências contadas. O primeiro for simplesmente preenche cada elemento de `fs` como 0. O segundo for conta quantas vezes cada valor aparece (armazenando em `fs[v[i] + 1]`).

O terceiro loop somará todas as frequências anteriores para definir a posição correta dos elementos de índice j, transformando `fs` em um array de prefixos acumulados. No quarto loop pegamos j que é o valor de `v[i]` e vemos qual o começo desse valor na lista de frequências, e adicionamos no lugar certo de temp graças a isso. Após, incrementamos o valor de `fs[j]`, pois, se o mesmo número aparecer, ele deve ir no próximo elemento depois do adicionado na iteração.

Por fim, o último loop apenas passa a lista ordenada em `temp` para `v`.

Podemos avaliar o desempenho do algoritmo Counting Sort através da seguinte função:

$$\begin{aligned} f(n,k) &= c_1k + c_2n + c_3k + c_4n + c_5n \\ &= (c_1 + c_3)k + (c_2 + c_4 + c_5)n \\ &= \Theta(k+n) \end{aligned} \tag{48}$$

Exige $O(n+k)$ de espaço adicional. Portanto, se k for muito pequeno a complexidade será $\Theta(n)$. É considerado um algoritmo eficiente para ordenar sequências com elementos repetidos.

5.8 Radix Sort

O algoritmo Radix sort consiste em ordenar os elementos de uma sequência digito à digito, do menos significativo para o mais significativo. Considera que cada elemento é uma sequência com w dígitos (Todos os elementos precisam ter w dígitos). Embora o termo dígito remeta à números do conjunto $\{0, 1, ..., 9\}$, podemos utilizar qualquer valor que possa ser mapeado em um inteiro

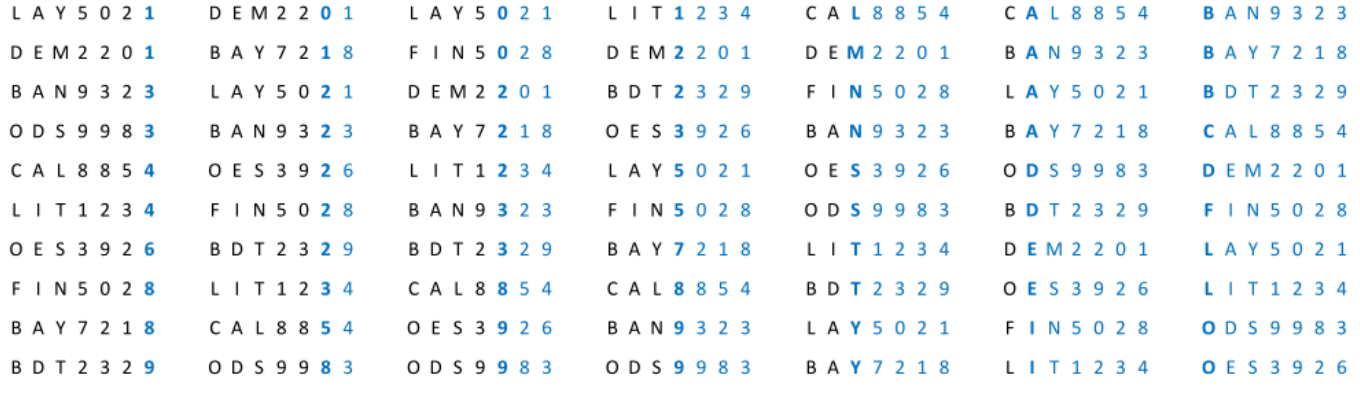


Figura 34: Radix Sort exemplo

IMPLEMENTAÇÃO C++

```
1  void radixSort(unsigned char * v[], int n, int W, int K) {
2    int fp [K + 1];
3    unsigned char* aux[n];
4    for (int w = W - 1; w >= 0; w--) {
5      for (int j = 0; j <= K; j++) { fp[j] = 0; }
6      for (int i = 0; i < n; i++) {
7        fp[v[i][w] + 1] += 1;
8      }
9      for (int j = 1; j <= K; j++) { fp[j] += fp[j - 1]; }
10     for (int i = 0; i < n; i++) {
11       int j = v[i][w];
12       aux[fp[j]] = v[i];
13       fp[j]++;
14     }
15     for (int i = 0; i < n; i++) { v[i] = aux[i]; }
16   }
17 }
```

Note que esse código é literalmente o Counting Sort só que para cada “dígito” do elemento. Portanto, a explicação do código está uma seção acima.

Podemos avaliar o desempenho do algoritmo Radix Sort através da seguinte função:

$$
\begin{aligned}
f(n,k,w) &= w(c_1k + c_2n + c_3k + c_4n + c_5n) \\
&= w((c_1 + c_3)k + (c_2 + c_4 + c_5)n) \\
&= \Theta(w(k+n))
\end{aligned} \tag{49}
$$

Exige $O(n+k)$ de espaço adicional. Se k e w forem pequenos a complexidade pode ser avaliada como $\Theta(n)$.

5.9 Bucket Sort

Esse algoritmo vai utilizar de hash tables para fazer ordenação de números potencialmente uniformes. Vamos pegar uma sequência de números **fracionários** v com n elementos e dividí-la em n grupos (baldes).

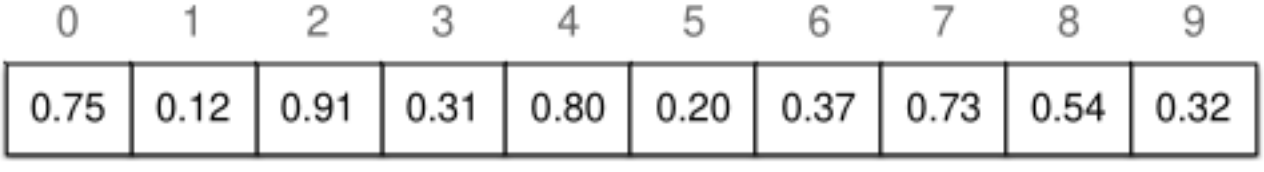


Figura 35: Lista de valores em $[0, 1)$ para exemplificação do algoritmo Bucketsort

Vamos pressupor que os valores estão **todos** entre $[0, 1]$

- Criar um vetor b com tamanho n
  - Cada elemento de b é uma lista encadeada
- Para cada elemento $v[i]$
  - Inserir em b na posição $\lfloor n \cdot v[i] \rfloor$
- Para cada lista $b[i]$
  - Ordenar os seus elementos utilizando algum algoritmo conhecido
    - Ex: Insertion Sort
- Para cada lista $b[i]$
  - Para cada elemento j de $b[i]$
    - Inserir na lista original v

```cpp
void bucketSort(float v[], int n) {
  vector<float> b[n];
  for (int i = 0; i < n; i++) {
    int inx = n * v[i];
    b[inx].push_back(v[i]);
  }
  for (int i = 0; i < n; i++) {
    insertionSort(b[i]);
  }
  int index = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < b[i].size(); j++) {
      v[index++] = b[i][j];
    }
  }
}
```

Podemos avaliar o desempenho do algoritmo através da seguinte função:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} n_i^2 \tag{50}$$

Portanto, temos que:

- O melhor caso é $\Theta(n)$ - cada balde recebe exatamente um elemento.
- O pior caso é $\Theta(n^2)$ - um único balde recebe n elementos.
- O caso médio é $\Theta(n)$ - considerando a distribuição uniforme esperada, poucos elementos caem no mesmo balde.
- Exige $O(n)$ de espaço adicional.

Algoritmos de Seleção

Dada uma sequência não ordenada,
1. como retornar a mediana do conjunto sem ordenar a sequência?
2. como retornar o i-ésimo elemento sem ordenar a sequência?

Se pudessemos ordenar... Bastaria:
1. acessar `v[n/2]` se n ímpar ou (`v[n/2]` + `v[n/2 + 1]`)/2 se n par;
2. apenas acessar o elemento `v[i]`.

Infelizmente, esse não é o caso. Então vamos aprender alguns algoritmos que descobrem isso sem ordenar a lista!

6.1 Quickselect

O algoritmo consiste em particionar a sequência conforme o algoritmo Quicksort, e buscar recursivamente escolhendo uma das partições.

Dada a sequência `v[0 ... n-1]` e a posição `x` buscada:
- Executamos o partition, obtendo a posição do pivô `j`. (note que estamos falando de índices)
  - se `j = x - 1`, encontramos o elemento (o - 1 vem da indexação)
  - se `j > x - 1`, executamos a recursão para `[0, ..., j - 1]`.
  - se `j < x - 1`, executamos a recursão para `[j + 1, ..., n - 1]`.

Exemplo: vamos buscar o terceiro menor elemento da sequência abaixo:

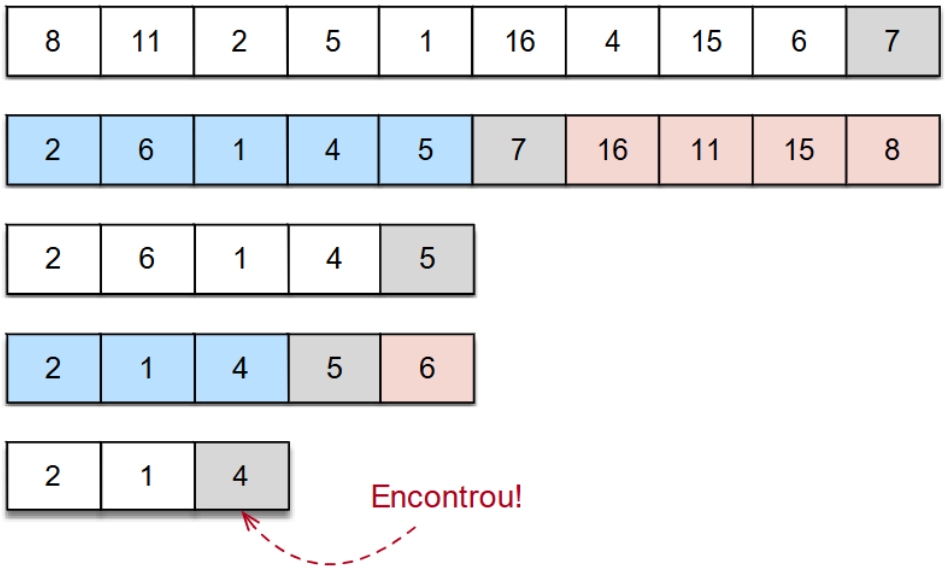


Figura 36: Exemplo do algoritmo Quickselect

Vamos para sua implementação!

```cpp
int quickselect(int v[], int l, int r, int x) {
  if (x > 0 && x <= r - l + 1) {
    int j = partition(v, l, r);
    if (j - l == x - 1) {
      return v[j];
    }
    if (j - l > x - 1) {
      return quickselect(v, l, j - 1, x);
    }
    return quickselect(v, j + 1, r, x - j + l - 1);
  }
  return -1;
```

```
13 }
```

O inteiro `l` é o índice do primeiro elemento da lista que iremos ordenar, e o inteiro `r` é o índice do último elemento. O primeiro if verifica se o x está dentro da lista, e a partir daí pega o seu índice `j`(a posição correta do último elemento da lista) pelo partition, e verificamos se ela é exatamente o índice `x` que procuramos. Se não, vai ao próximo if(seria como se fosse um else if nesse caso) e verifica a esquerda de `j`, `[l, ..., j-1]` procurando por `x`.

Por fim, o último return do if inicial faz o quickselect para a esquerda de `j`, `[j + 1, ... , r]`, porém note que ele muda a indexação do `x`, por quê?

Pois ao procurar `x`, note que o `x` não é o mesmo do ínicio, pois nos movemos `l` casas a direita. Por exemplo:

Exemplo:

Queremos encontrar o **4° menor elemento** no vetor `v = [7, 2, 9, 4, 6]`:

`quickselect(v, l=0, r=4, x=4)`

- Subarray considerado: `[7, 2, 9, 4, 6]`
- Pivô escolhido: 6

Após a partição: `[2, 4, 6, 9, 7]`

- Índice global do pivô: j = 2
- Procuramos x - 1 = 3
- Como j - l = 2 < 3, o elemento desejado está à direita do pivô.

Note que aqui deveríamos então fazer a recursão a direita, e teríamos então o vetor `[9,7]` para olhar. Porém note que olhando para o mesmo `x` do começo, saíriamos da lista. Por isso precisamos "normalizar" o `x`, somando a quantidade de elementos que já passamos(`j - l`), e o `+1` do índice. Por isso `x = x -(j - l + 1)`.

Agora que entendemos o algoritmo, vamos analisar a complexidade:

No melhor caso a sequência será particionada usando a mediana como pivô, levando à seguinte função:

$$T(n) = cn + T\left(\frac{n}{2}\right) \tag{51}$$

e, continuando:

$$\begin{aligned} T(n) &= cn + T\left(\frac{n}{2}\right) \\ &= cn + \frac{cn}{2} + T\left(\frac{n}{4}\right) \\ &= cn + \frac{cn}{2} + \frac{cn}{4} + T\left(\frac{n}{8}\right) \\ &= cn\left(1 + \frac{1}{2} + \frac{1}{4} + ...\right) \leq O(n) \end{aligned} \tag{52}$$

No entanto, no pior caso teremos sempre um conjunto vazio e um conjunto com $n-1$ elementos, levando a seguinte função de recorrência:

$$T(n) = cn + T(n-1) \tag{53}$$

e, continuando:

$$
\begin{aligned}
T(n) &= cn + T(n-1) \\
&= cn + c(n-1) + T(n-2) \\
&= cn + c(n-1) + c(n-2) + T(n-3) \\
&= c(n + (n-1) + (n-2) + ... + 2 + 1) \\
&= \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned} \tag{54}
$$

Portanto, o pior caso $O(n^2)$ e o melhor caso $O(n)$.

6.2 Mediana das Medianas