Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET-TJ)

Programa de Pós Graduação em Ciência da Computação (PPCIC)

Disciplina: Análise e Projetos de Algoritmos

Professor: Diego Nunes Brandão

Aluna: Cristiane Gea

Resolução da 2ª Lista de Exercícios

Questão 1

Quantos antecedentes tem um nó no nível n em uma árvore binária? Prove!

Contextualização

Árvore binária:

- Cada nó pode ter zero, 1 ou 2 filhos. Portanto, cada nó tem, no máximo, 2 filhos.
- A árvore é representada pelo seu nó raiz, de maneira recursiva.
- A árvore pode ser definida como (i) uma árvore vazia ou (ii) um nó raiz contendo 2 subárvores (identificadas como subárvore da direita e subárvore da esquerda).

Regra para determinação do nível de um nó numa árvore binária:

- A raiz da árvore tem nível 0;
- O nível de qualquer outro nó na árvore é um nível a mais que o nível de seu pai.

Profundidade (altura) de uma árvore binária: nível máximo de qualquer folha na árvore, correspondendo ao tamanho do percurso mais distante da raiz até qualquer folha.

Resposta

Para o desenvolvimento da questão utilizaremos como exemplo representativo a árvore binária cheia (ou árvore binária completa), visto que é uma árvore cujos nós folhas estão no mesmo nível e não pode haver acréscimo de nenhuma nova folha sem implicar no aumento da altura da árvore. Diante do fato de a árvore binária cheia ser uma árvore binária com o número máximo de nós para determinada profundidade, qualquer outra estrutura de árvore binária (que não seja cheia) possuirá um número menor de nós.

Obs.: uma árvore binária cheia de profundidade d é a árvore estritamente binária em que todas as folhas estão no nível d.

Segundo Celes et al. (2016), uma árvore cheia, no nível n, possuirá n^2 nós. Além disso, o número de nós de determinado nível de uma árvore cheia é uma unidade a mais do que a soma de todos os nós dos níveis anteriores. Em outras palavras, o número de nós antecessores a um nó no nível n em uma árvore binária cheia corresponderá a:

$$N = \sum_{i}^{n} 2^i = 2^{n+1} - 1$$

Se uma árvore binária contiver k nós no nível n, então ela conterá no máximo $2k$ nós no nível $n + 1$. Como uma árvore binária pode conter no máximo um nó no nível zero (raiz), ela poderá conter no máximo 2^l nós no nível l. Uma árvore binária completa de profundidade n é a árvore binária de profundidade n que contém exatamente 2^l nós em cada nível l entre 0 e n. O número total de nós em uma árvore binária cheia que possua profundidade igual a n é igual à soma dos números de nós em cada nível entre 0 e n. Portanto,

$$N = 2^0 + 2^1 + \cdots + 2^n = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Assim, como todas as folhas nesta árvore estão no nível n, a árvore contém 2^n folhas e, portanto, $2^n - 1$ nós sem folhas.

Por conseguinte, diante do exposto é possível inferir que para o caso geral, uma árvore binária terá no máximo $2^n - 1$ nós sem folhas.

Questão 2

Escreva um programa que implementa funções para:

(a) Contar o número de nós em uma árvore binária;

(b) Contar o número de folhas;

(c) Contar o número de filhos à direita;

(d) Contar a altura da árvore.

Contextualização

Segundo Tenenbaum et al. (2004), uma árvore binária é um conjunto finito de elementos que está vazio ou é particionado em 3 subconjuntos disjuntos:

- raiz (contém um único elemento);
- subárvore à esquerda da árovre original;
- subárvore à direita da árvore original.

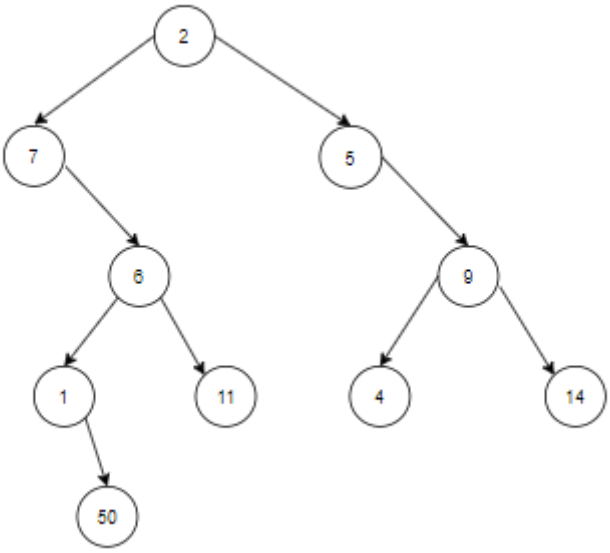
Além disso, cada elemento de uma árvore binária é chamado de nó da árvore.

Complementando a definição, em uma árvore binária (Celes et al., 2016; Necaise, 2011)):

- cada nó tem, no máximo, 2 filhos;
- cada nó tem zero, 1 ou 2 filhos;
- a árvore é representada pelo seu nó raiz, de maneira recursiva;
- a árvore pode ser definica como (i) uma árvore vazia ou (ii) um nó raiz tendo 2 subárvores identificadas como subárvore esquerda e subárvore direita.
- a altura de uma árvore corresponde ao comprimento do camionho da raiz até a folha mais distante.
- as arestas representam a relação entre os nós que são encadeados com setas ou arestas direcionais para formar uma estrutura hierárquica.

Resposta

A árvore utilizada para a resolução dos tópicos abaqixo possui a seguinte representação:



In [1]:

```
# Criação de uma classe para instanciar a árvore binária
class BinaryTree(object):
    # Construtor
    def __init__(self, value):
        self.value = value      # início da árvore // nó raiz
        self.left = None        # subárvore esquerda
        self.right = None       # subárvore direita
        self.count = 1

    # Função para contar o número de nós
    def __str__(self):
        return 'Valor: {0}, quantidade: {1}'.format(self.value, self.count)
```

In [2]:

```
# Definição dos valores dos nós da árvore binária
root = BinaryTree(2)
root.left = BinaryTree(7)
root.right = BinaryTree(5)
root.left.right = BinaryTree(6)
root.left.right.left = BinaryTree(1)
root.left.right.right = BinaryTree(11)
root.left.right.left.right = BinaryTree(50)
root.right.right = BinaryTree(9)
root.right.right.left = BinaryTree(4)
root.right.right.right = BinaryTree(14)
```

Contagem do número de nós de uma árvore

In [3]:

```
def countAllNodes(root):
    if root is None:
        return 0
    return 1 + countAllNodes(root.left) + countAllNodes(root.right)

print("Número total de nós:" , countAllNodes(root))
```

```
Número total de nós: 10
```

Contagem do número de folhas de uma árvore

In [4]:

```
def countLeaf(node):
    if node is None:
        return 0
    if(node.left is None and node.right is None):
        return 1
    else:
        return countLeaf(node.left) + countLeaf(node.right)

print("Número de folhas:", countLeaf(root))
```

```
Número de folhas: 4
```

Contagem do número de filhos à direita

In [5]:

```
# Considerando somente os filhos que estão à direita
def countRightChildren(root):
    if root == None:
        return 0
    res = 0
    if(root.left == None and root.right != None):
        res += 1
    res += (countRightChildren(root.left) + \
            countRightChildren(root.right))
    return res

print("Números de filhos à direita:", countRightChildren(root))
```

```
Números de filhos à direita: 3
```

In [6]:

```
# Considerando os filhos contidos na subárvore à direita da árvore original
def countSubTreeRightNodes(root):
    if root is None:
        return 0
    res = 0
    if(root.left == None and root.right != None):
        res += 1
    res += countAllNodes(root.right)
    return res

print("Número total de filhos da subárvore à direita da árvore original:" , countSubTreeRig
```

```
Número total de filhos da subárvore à direita da árvore original: 4
```

Contagem da altura da árvore

In [7]:

```
def countHeight(node):
    if node is None:
        return 0
    else:
        leftDepth = countHeight(node.left)   # cálculo da altura da subárvore esquerda
        rightDepth = countHeight(node.right)  # cálculo da altura da subárvore direita
        if (leftDepth > rightDepth):
            return leftDepth + 1
        else:
            return rightDepth + 1

print("Altura da árvore:", countHeight(root))
```

```
Altura da árvore: 5
```

Questão 3

Escreva um programa de referência cruzada que construa uma árvore binária de pesquisa, com todas as palavras incluídas a partir de um arquivo de texto, e registre os números das linhas nas quais essas palavras foram usadas. Esses números de linhas devem ser armazenados em listas associadas aos nós da árvore. Depois de o arquivo de entrada ter sido processado, imprima em ordem alfabética todas as palavras do arquivo texto, junto com a lista de números de linhas correspondente nas quais as palavras ocorrem.

Contextualização:

Segundo Celes et al. (2016), as árvores binárias de busca possuem a seguinte propriedade fundamental: o valor associado à raiz é sempre maior que o valor associado a qualquer nó da subárvore esquerda e é sempre menor que o valor associado a qualquer nó da subárvore direita. Em outras palavras, se x é um nó em uma árvore binária de busca e se y é um nó da subávore esquerda de x, então $y.chave \leq x.chave$. Caso contrário, se y é um nó na subárvore direita de x, então $y.chave \geq x.chave$.

Neste sentido, a propriedade de árvore binária de busca permite imprimir todas as chaves em uma única árvore binária de busca em sequência ordenada por meio de um algoritmo recursivo, chamado percurso de árvore in-ordem (Cormen et al, 2012). O percurso de árvore in-ordem (`inorder` // left-root-right) imprime a chave raiz de uma subárvore entre a impressão dos valores em sua subárvore esquerda e a impressão dos

valores em sua subárvore direita. Por outro lado, um percurso de árvore pré-ordem (`preorder` // root-left-right) imprime a raiz antes dos valores das subárvores e um percurso de árvore pós-ordem (`postorder` // left-right-root) imprime a raiz depois dos valores em suas subárvores.

Resolução:

Obs.: Apesar de ter imprimido em ordem alfabética todas as palavras do arquivo texto, não consegui imprimi-lo junto com a lista de números de linhas correspondente nas quais as palavras ocorrem.

In [8]:

```
# Função para leitura do arquivo de texto e criação da lista contendo todas as letras conti
def textWords(filename):
    openfile = open(filename, "r")
    templist = []
    letterslist = []
    for lines in openfile:
        for i in lines:
            ii = i.lower()
            letterslist.append(ii)
    for p in letterslist:
        if p not in ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r
                     's','t','u','v','w','x','y','z',"'","-",' '] and p.isdigit() == False:
            letterslist.remove(p)
    wordslist = list("".join(letterslist).split())
    return wordslist
```

In [9]:

```
my_list = textWords("text_user.txt")
print(my_list)
```

```
['time-variation', 'in', 'disagreement', 'about', 'future', 'inflation', 'i
s', 'a', 'stylized', 'fact', 'in', 'survey', 'data', 'but', 'little', 'is',
'known', 'on', 'how', 'disagreement', 'interacts', 'with', 'the', 'efficac
y', 'of', 'monetary', 'policy']
```

Para esta questão, será utilizada a mesma classe de criação de árvore binária desenvolvida na questão anterior.

In [10]:

```
# Função para inserir nós na árvore
def insertNode(root, value):
    if not root:
        return BinaryTree(value)
    elif root.value == value:
        root.count += 1
    elif value < root.value:
        root.left = insertNode(root.left, value)
    else:
        root.right = insertNode(root.right, value)
    return root

# Função para a árvore binária a partir das palavras extraídas do arquivo de texto
def createBinarySearchTree(sequency):
    root = None
    for word in sequency:
        root = insertNode(root, word)
    return root

# Função para procurar elementos na árvore binária de busca
def searchNode(root, word, depth=1):
    if not root:
        return 0, 0
    elif root.value == word:
        return depth, root.count
    elif word < root.value:
        return searchNode(root.left, word, depth + 1)
    else:
        return searchNode(root.right, word, depth + 1)

# Função para imprimir a árvore binária de busca
def displayBinarySearchTree(root):
    if root:
        displayBinarySearchTree(root.left)
        print(root)
        displayBinarySearchTree(root.right)

# Função para o percurso pós-ordem (left-rigt-root)
#def postOrder(root):
#    if(root.left!=None):
#        postorder(root.left)
#    if(root.right!=None):
#        postorder(root.right)
#    print (root.value)

# Função para o percurso in-ordem (left-root-rigt)
def inOrder(root):
    if(root.left!=None):
        inOrder(root.left)
    print (root.value)
    if(root.right!=None):
        inOrder(root.right)
```

In [11]:

```
# Leitura do arquivo de texto e criação de uma lista com as letras contidas no arquivo
src = my_list

# Criação da árvore de busca binária com as letras contidas no arquivo de texto
bst = createBinarySearchTree(src)

# Impressão da árvore binária de busca
displayBinarySearchTree(bst)
```

```
Valor: a, quantidade: 1
Valor: about, quantidade: 1
Valor: but, quantidade: 1
Valor: data, quantidade: 1
Valor: disagreement, quantidade: 2
Valor: efficacy, quantidade: 1
Valor: fact, quantidade: 1
Valor: future, quantidade: 1
Valor: how, quantidade: 1
Valor: in, quantidade: 2
Valor: inflation, quantidade: 1
Valor: interacts, quantidade: 1
Valor: is, quantidade: 2
Valor: known, quantidade: 1
Valor: little, quantidade: 1
Valor: monetary, quantidade: 1
Valor: of, quantidade: 1
Valor: on, quantidade: 1
Valor: policy, quantidade: 1
Valor: stylized, quantidade: 1
Valor: survey, quantidade: 1
Valor: the, quantidade: 1
Valor: time-variation, quantidade: 1
Valor: with, quantidade: 1
```

In [12]:

```
# Impressão da subárvore esquerda da árvore original
displayBinarySearchTree(bst.left)
```

```
Valor: a, quantidade: 1
Valor: about, quantidade: 1
Valor: but, quantidade: 1
Valor: data, quantidade: 1
Valor: disagreement, quantidade: 2
Valor: efficacy, quantidade: 1
Valor: fact, quantidade: 1
Valor: future, quantidade: 1
Valor: how, quantidade: 1
Valor: in, quantidade: 2
Valor: inflation, quantidade: 1
Valor: interacts, quantidade: 1
Valor: is, quantidade: 2
Valor: known, quantidade: 1
Valor: little, quantidade: 1
Valor: monetary, quantidade: 1
Valor: of, quantidade: 1
Valor: on, quantidade: 1
Valor: policy, quantidade: 1
Valor: stylized, quantidade: 1
Valor: survey, quantidade: 1
Valor: the, quantidade: 1
```

In [13]:

```
# Impressão da subárvore direita da árvore original
displayBinarySearchTree(bst.right)
```

```
Valor: with, quantidade: 1
```

In [14]:

```
# Percurso de árvore in-orderm
inOrder(bst)
```

```
a
about
but
data
disagreement
efficacy
fact
future
how
in
inflation
interacts
is
known
little
monetary
of
on
policy
stylized
survey
the
time-variation
with
```

Questão 4

Existe um resultado matemático conhecido como “paradoxo do aniversário”. Ele afirma que, se existe mais de 23 pessoas em uma determinada sala, existe mais de 50% de chance de que duas pessoas façam aniversário no mesmo dia. Explique por que este paradoxo é um exemplo do maior problema existente na técnica de hash.

Contextualização

De acordo com Drozdeck (2013), hashing pode ser compreendido como uma abordagem diferente de pesquisa, com base no cálculo da posição da chave na tabela de acordo com o valor da chave. Neste sentido, o valor da chave funciona como indicação da posição do elemento dentro da tabela. Diante disso, a função de hash (h) é utilizada para transformar uma chave em particular (K) em um índice na tabela utilizado para armazenar os itens do mesmo tipo que K.

Uma função de hash é dita ser perfeita se consegue transformar diferentes chaves em diferentes números, situação na qual não há colisões. Contudo, apesar de a criação de uma função de hash perfeita requerer que a tabela contenha pelo menos o mesmo número de posições que o número de elementos, o número de elementos nem sempre é conhecido antes do tempo. Além disso, mesmo se a tabela acomodar todas as variáveis no programa, não há garantia de que a função de hash retorne chaves únicas aos valores das variáveis, incorrendo, portanto, em colisões.

O número de funções de hash pode ser utilizado para atribuir posições aos n itens em uma tabela de m posições (para $n \leq m$). Assim, o número perfeito de funções de hash é o mesmo que o número de diferentes posicionamentos desses itens na tabela. Contudo, na maioria dos casos, uma ou mais chaves podem ser associadas à mesma posição.

Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.

Um método de pesquisa com o uso da transformação de chave com o uso da transformação de chave é constituído de duas etapas principais:

- Computar o valor da função de transformação, a qual transforma a chave de pesquisa em um endereço da tabela.
- Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é preciso utilizar um método que lide com as colisões (que surgem quando duas ou mais chaves de busca são mapeadas para um mesmo índice da tabela de dispersão).

Contudo, qualquer que seja a função de transformação, algumas colisões irão ocorrer (e tais colisões têm de ser resolvidas). Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entrdas da tabela, há uma alta probabilidade de haver colisões.

Paradoxo do aniversário

Hipóteses:

- As pessoas são indexadas com os inteiros $1, 2, \ldots, k$, onde k é o número de pessoas na sala.
- São ignoradas as questões dos anos bissextos.
- Todos os anos possuem $n = 365$ dias.
- Para $i = 1, 2, \ldots, k$, b_i corresponde ao dia do ano no qual cai o aniversávario da pessoa i, onde $1 \leq b_i \leq$.
- Os aniversários são uniformemente distribuídos pelos n dias do ano, de modo que $\Pr b_i = r = 1/n$ para $1, 2, \ldots, k$ e $r = 1, 2, \ldots, n$.

A probabilidade de que 2 pessoas (i, j) tenham datas de aniversário coincidentes depende do fato de a seleção aleatória de aniversários ser independente. Assim, supondo que os aniversários são independentes, a probabilidade de o aniversário de i e o aniversário de j caírem, ambos, no dia r é $\Pr\left\{b_i = r \text{ e } b_j = r\right\} = \Pr\left\{b_i = r\right\} \Pr\left\{b_j = r\right\} = 1/n^2$. Por outro lado, a probabilidade de ambos caírem no mesmo dia é $\Pr\left\{b_i = b_j\right\} = \sum_{r=1}^{n} \Pr\left\{b_i = r \text{ e } b_j = r\right\} = 1/n$.

Obs.: a coincidência das datas de aniversário depende da suposição de que os dias de aniversário são independentes.

Resposta

Se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%. Diante disso, para hashing, a tabela possui 365 posições, com adição de 23 chaves e chance de mais de 50% de haver colisão.

A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \left(\frac{M-1}{M}\right) \times \left(\frac{M-2}{M}\right) \times \cdots \times \left(\frac{M-N+1}{M}\right) = \prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

Como existe um número fixo de dias e a probabilidade de se nascer em um determinado dia é aleatória e igualmente provável para cada dia, a chance da primeira pessoa nascer no mesmo dia que a segunda pessoa é $1/365$. Contudo, se houver uma terceira pessoa, há probabilidade de $1/365$ dela nascer no mesmo dia que a primeira pessoa, que tem a probabilidade de $1/365$ de nascer no mesmo dia que a segunda pessoa. Contudo, à medida que o número de pessoas vai aumentando, a probabilidade também vai aumentando.

As funções de hash recebem como entrada um dado de tamanho arbitrário e produzem como saída um dado de tamanho fixo. Assim, se um número muito grande de entradas distintas forem utilizadas, a probabilidade de colisão de hashes se torna cada vez maior. Em outras palavras, é possível produzir colisões simplesmente

gerando um número suficientemente alto de entradas distintas. Contudo, pelo paradoxo do aniversário, esse número é muito menor do que o total de saídas distintas.

Questão 5

Observando cada um dos caracteres que formam a palavra C O M P U T A Ç Ã O. Crie uma tabela de dispersão (*hash table*) que armazene cada um destes caracteres, utilizando a seguinte função: `h(char) = (ordem de char no alfabeto)%5` .

Desenhe o esquema de como ficaram a *hash table* e as listas encadeadas.

Contextualização

Segundo Cormen et al. (2012), a tabela de dispersão (hashing) generaliza a noção mais simples de um arranjo comum. O endereçamento direto de um arranjo comum faz uso eficiente da habilidade de examinar uma função arbitrária em um arranjo no tempo $O(1)$.

Quando o número de chaves realmente armazenados é pequeno em relação ao número total de chaves possíveis, as tabelas de dispersão se tornam uma alternativa eficaz para endereçar diretamente um arranjo, já que normalmente uma tabela de dispersão utiliza um arranjo de tamanho proporcional ao número de chaves realmente armazenadas. Em vez de usar uma chave doretamente como um índice de arranjo, o índice de arranjo é calculado a partir da chave (a posição da chave k é calculada por meio da função hash).

Método de divisão: para a criação da função hash, é realizado um mapeamento de uma chave k para uma de m posições, tomando o resto da divisão de k por $m \Rightarrow h(k) = k \bmod m = k\%m$

Resposta

Função de dispersão: h(key) = key % M => `h(char) = (ordem de char no alfabeto)%5`

Número de elementos da tabela de dispersão: M = 5

Obs.: apesar de ter feito o desenho esquemático de como ficaria a tabela de dispersão e as listas encadeadas, no código não consegui fazer a implementação da tabela de dispersão com as listas encadeadas (mas, somente a implementação da tabela de dispersão com listas aninhadas).

In [15]:

```
my_list = ['C', 'O', 'M', 'P', 'U', 'T', 'A', 'C', 'A', 'O']
len(my_list)
```

Out[15]:

```
10
```

In [16]:

```
alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
            'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Z', 'Y']
```

In [17]:

```
alphabet.index('C')%5
```

Out[17]:

```
2
```

In [18]:

```
alphabet.index('O')%5
```

Out[18]:

```
4
```

In [19]:

```
alphabet.index('M')%5

# Colisão com C
```

Out[19]:

```
2
```

In [20]:

```
alphabet.index('P')%5
```

Out[20]:

```
0
```

In [21]:

```
alphabet.index('U')%5

# Colisão com P
```

Out[21]:

```
0
```

In [22]:

```
alphabet.index('T')%5

# Colisão com O
```

Out[22]:

```
4
```

In [23]:

```
alphabet.index('A')%5

# Colisão com P e com U
```

Out[23]:

```
0
```

In [24]:

```
alphabet.index('C')%5

# Colisão com o 1º C
```

Out[24]:

```
2
```

In [25]:

```
alphabet.index('A')%5

# Colisão com P, com U e com o 1º A
```

Out[25]:

```
0
```

In [26]:

```
alphabet.index('O')%5

# Colisão com 1º O e com T
```

Out[26]:

```
4
```

In [27]:

```
# Função para imprimir a tabela de dispersão
def printHash(hashTable):
    for i in range(len(hashTable)):
        print(i, end = " ")
        for j in hashTable[i]:
            print("-->", end = " ")
            print(j, end = " ")
        print()

# Função hashing para retornar a chave para cada valor
def Hashing(keyvalue):
    return keyvalue % len(HashTable)

# Insert Function to add values to the hash table
def insertHash(Hashtable, keyvalue, value):
    hash_key = Hashing(keyvalue)
    Hashtable[hash_key].append(value)
```

In [28]:

```
m = 13

# Criação de uma tabela de dispersão como uma lista aninhada
HashTable = [[] for _ in range(m)]
```

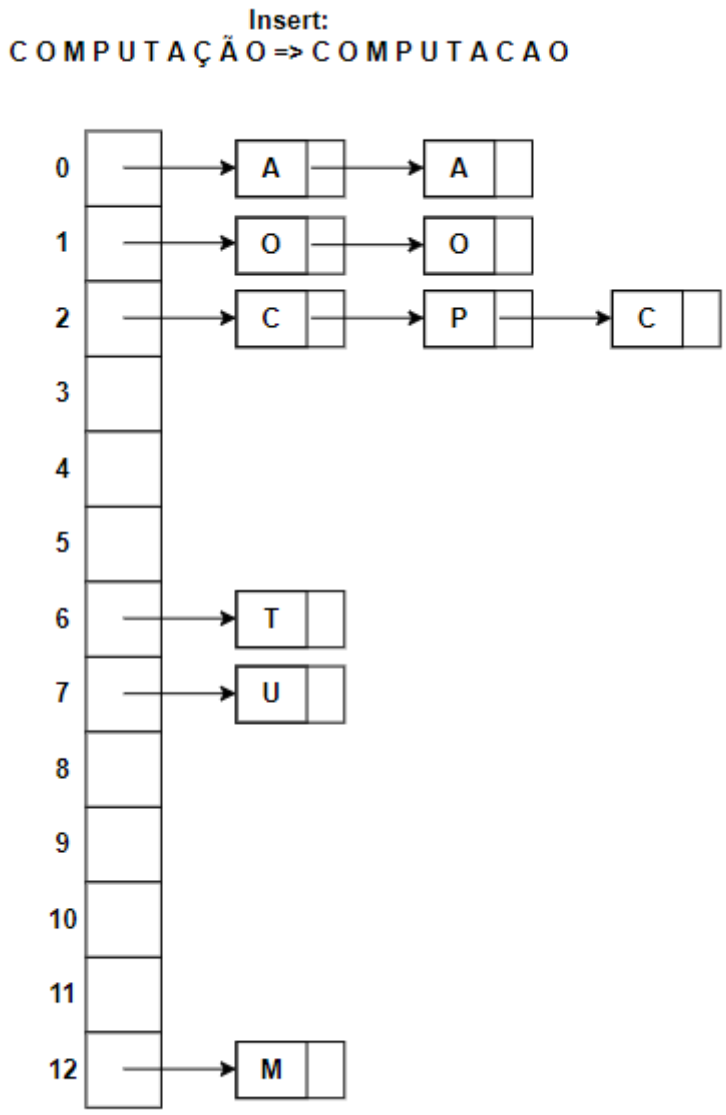
In [29]:

```
for i in list(my_list):
    insertHash(HashTable, alphabet.index(i), i)

printHash(HashTable)
```

```
0 --> A --> A
1 --> O --> O
2 --> C --> P --> C
3
4
5
6 --> T
7 --> U
8
9
10
11
12 --> M
```

Cada posição da tabela está associada com uma lista encadeada, ou cadeia de estruturas. Neste método, a tabela pode nunca estourar, pois as listas encadeadas são prolongadas somente com a chegada de novas chaves, conforme ilustrado na imagem abaixo.



Segundo Drozdek (2013), este método requer espaço adicional para a manutenção dos ponteiros. A tabela armazena somente ponteiros e cada nó requer um campo de ponteiro. Neste sentido, para n chaves, $n + (\text{tamanho\_da\_tabela})$ ponteiros são necessários.

Questão 6

Crie um arquivo `hash` com registros para 25 nomes de cidades do estado de Minas Gerais cujos nomes começem com as seguintes letras do alfabeto: `a`, `b`, `c` e `s`. As chaves de cada registro é o nome a cidade, não sendo necessários outros campos. Coloque inicialmente os nomes das cidades em ordem alfabética.

(a) Examine a lista ordenada. Que padrões você observa que podem afetar a sua escolha de uma função hash?

(b) Implemente uma função `hash()` que utiliza alguma combinação dos códigos ASCII das letras do nome, mas de forma que você possa alterar o número de caracteres que são utilizados na combinação. Execute o `hash()` várias vezes, cada vez utilizando um número diferente de caracteres e produzindo as seguintes estatísticas para cada execução:

- O número de colisões
- O número de endereços com $0, 1, 2, 3, \ldots, 10,$ ou mais de 10 cidades associadas.

Discuta os resultados de seu experimento em termos de efeitos da escolha de diferentes quantidades de caracteres e como eles se relacionam com o resultado que você poderia esperar de uma distribuição aleatória. (Implemente e teste um ou mais dos métodos de hash descritos no texto, ou use um método inventado por você).

Resposta

25 nomes de municípios de Minas Gerais

- Começando com a letra `a` : Aimorés, Alfenas, Andradas, Araguari, Araxá, Arcos
- Começando com a letra `b` : Barbacena, Belo Horizonte, Betim, Boa Esperança, Bom Despacho, Brumadinho
- Começando com a letra `c` : Caeté, Caratinga, Conselheiro Lafaiete, Contagem, Coronel Fabriciano, Curvelo
- Começando com a letra `s` : Sabará, Santos Dumont, São Francisco, São João del Rei, São Lourenço, São Sebastião do Paraíso, Sete Lagoas

Olhando para os nomes das cidades, é possível identificar alguns padrões que podem induzir à ocorrência de colisões:

- Repetição da primeira letra de cada nome de cidade.
- Repetição das duas primeiras letras dos nomes `Ar` aguari, `Ar` axá, `Ar` cos, `Be` lo Horizonte, `Be` tim, `Bo` a Esperança, `Ca` eté, `Car` atinga, `Co` nselheiro Lafeiete, `Co` ntagem, `Sa` bará, `Sa` ntos Dumont, `São` Francisco, `São` João del Rei, `São` Lourenço, `São` Sebastião do Paraíso.
- Repetição das três primeiras letras dos nomes `Ara` guari, `Ara` xá.

Para fazer a segunda parte da questão foram utilizados dois métodos para combinar os valores dos códigos ASCII:

(i) Formada a partir da soma dos códigos ASCII de todas as letras da palavra;

(ii) Formada a partir da soma dos códigos ASCII das letras que estejam na posição ímpar.

Posteriormente, foram realizadas simulações de alteração e remoção de caracteres a fim de examinar a possível ocorrência de colisões e o número de cidades associadas a cada endereço.

Além disso, vale a pena destacar que as funções utilizadas para a criação, adição e impressão da hash são as mesmas utilizadas para o desenvolvimento da questão 6.

A tabela abaixo mostra o comparativo entre os 2 métodos utilizados para realizar combinações de código ASCII. Conforme exposto na tabela, é possível observar que ambas as combinações analisada apresentam desempenho semelhante. Além disso, é possível evidenciar que os casos onde houve a remoção de caracteres especiais apresentam maior número de colisões, em detrimento dos demais casos analisado. Esse resultado é esperado uma vez que a concatenação dos caracteres especiais com espaçamentos funcionam como características específicas de cada nome e a retirada dos mesmos dificulta a transformação de diferentes chaves em diferentes números. Além disso, em uma análise comparativa a combinação 1 gerou menos colisões durante as simulaçõs, em comparação com a combinação 2 (com exceção das simulações 7 e 12).

Combinação 1 (soma dos códigos ASCII de todas as letras da palavra)

Simulações	Número de colisões	Endereço com nenhuma cidade associada	Endereço com 1 cidade associada	Endereço com 2 cidades associadas	Endereço com 3 cidades associadas	Endereço com 4 cidades associadas	Endereço com 5 cidades associadas

Simulações	Número de colisões	Endereço com nenhuma cidade associada	Endereço com 1 cidade associada	Endereço com 2 cidades associadas	Endereço com 3 cidades associadas	Endereço com 4 cidades associadas	Endereço com 5 cidades associadas
(1) nomes das cidades em seu formato original	4	11	14	1	3	0	0
(2) todos os nomes das cidades em minúsculo	5	10	14	4	1	0	0
(3) todos os nomes das cidades em maiúsculo	4	10	15	3	1	0	0
(4) nomes das cidades sem acento e sem caracteres especiais (ex.: "ç")	6	13	10	5	0	0	1
(5) nomes das cidades em minúsculo, sem acento e sem caracteres especiais (ex.: "ç")	6	11	12	5	1	0	0
(6) nomes das cidades em maiúsculo, sem acento e sem caracteres especiais (ex.: "ç")	5	10	14	4	1	0	0
(7) nomes das cidades sem espaçamento	8	13	8	7	1	0	0
(8) nomes das cidades em minúsculo e sem espaçamento	6	13	10	3	3	0	0
(9) nomes das cidades em maiúsculo e sem espaçamento	5	11	13	4	1	0	0
(10) nomes das cidades sem espaçamento e sem caracteres especiais (acentos e "ç")	6	11	12	5	1	0	0
(11) nomes das cidades em minúsculo sem espaçamento e sem caracteres especiais (acentos e "ç")	6	14	9	4	1	0	1
(12) nomes das cidades em maiúsculo sem espaçamento e sem caracteres especiais (acentos e "ç")	6	11	12	5	1	0	0

Combinação 2 (soma dos códigos ASCII das letras que estão nas posições ímpares da palavra)

Simulações	Número de colisões	Endereço com nenhuma cidade associada	Endereço com 1 cidade associada	Endereço com 2 cidades associadas	Endereço com 3 cidades associadas	Endereço com 4 cidades associadas	Endereço com 5 cidades associadas
(1) nomes das cidades em seu formato original	6	11	12	5	1	0	0
(2) todos os nomes das cidades em minúsculo	8	13	8	7	1	0	0
(3) todos os nomes das cidades em maiúsculo	5	12	12	3	1	1	0

Simulações	Número de colisões	Endereço com nenhuma cidade associada	Endereço com 1 cidade associada	Endereço com 2 cidades associadas	Endereço com 3 cidades associadas	Endereço com 4 cidades associadas	Endereço com 5 cidades associadas
(4) nomes das cidades sem acento e sem caracteres especiais (ex.: "ç")	7	13	9	5	2	0	0
(5) nomes das cidades em minúsculo, sem acento e sem caracteres especiais (ex.: "ç")	7	11	11	7	0	0	0
(6) nomes das cidades em maiúsculo, sem acento e sem caracteres especiais (ex.: "ç")	6	11	12	5	1	0	0
(7) nomes das cidades sem espaçamento	8	13	8	7	1	0	0
(8) nomes das cidades em minúsculo e sem espaçamento	7	13	9	5	2	0	0
(9) nomes das cidades em maiúsculo e sem espaçamento	8	13	8	7	1	0	0
(10) nomes das cidades sem espaçamento e sem caracteres especiais (acentos e "ç")	8	14	7	6	2	0	0
(11) nomes das cidades em minúsculo sem espaçamento e sem caracteres especiais (acentos e "ç")	7	13	9	5	2	0	0
(12) nomes das cidades em maiúsculo sem espaçamento e sem caracteres especiais (acentos e "ç")	6	13	10	4	1	1	0

In [30]:

```
# Combinação 1: Considerando a posição dos elementos somente com base na soma dos valores A
def ascii_number(key):
    x = 0
    for i in list(key):
        ascii_code = ord(i)
        x += ascii_code
    return x
```

In [31]:

```
# Combinação 2: Considerando a posição dos elementos somente com base na soma dos valores A
#em posição ímpar
from math import sqrt

# Função para verificar se o valor é primo
def isPrime(n):
    if(n == 0 or n == 1):
        return False
    for i in range(2, int(sqrt(n)) + 1) :
        if (n % i == 0):
            return False
    return True

# Função para somar os valores do código ASCII dos elementos que ocupam posições ímpares
def ascii_prime(key, n):
    x = 0
    for i in range(n):
        if (isPrime(i + 1)):
            x += ord(key[i])
    return x
```

In [32]:

```
m = 29
```

Simulação 1: nomes das cidades em seu formato original

In [33]:

```
city1 = ['Aimorés', 'Alfenas', 'Andradas', 'Araguari', 'Araxá', 'Arcos',
         'Barbacena', 'Belo Horizonte', 'Betim', 'Boa Esperança', 'Bom Despacho', 'Brumadin
         'Caeté', 'Caratinga', 'Conselheiro Lafaiete', 'Contagem', 'Coronel Fabriciano',
         'Curvelo', 'Sabará', 'Santos Dumont', 'São Francisco', 'São João del Rei',
         'São Lourenço', 'São Sebastião do Paraíso', 'Sete Lagoas']
```

Combinação 1

In [34]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [35]:

```
for i in list(city1):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> Araguari
1
2 --> Alfenas --> Conselheiro Lafaiete --> Contagem
3 --> São Lourenço
4 --> Betim
5 --> Caeté
6
7 --> Caratinga
8
9 --> Barbacena
10 --> São Francisco
11 --> Aimorés --> Arcos --> Curvelo
12 --> Araxá
13 --> São João del Rei
14
15 --> Andradas
16
17 --> Belo Horizonte --> Sete Lagoas
18 --> Boa Esperança --> Brumadinho --> Sabará
19 --> Santos Dumont
20 --> Coronel Fabriciano
21
22 --> São Sebastião do Paraíso
23 --> Bom Despacho
24
25
26
27
28
```

Combinação 2

In [36]:

```
# Criação da tabela de dispersão como lista aninhada
HashTable = [[] for _ in range(m)]
```

In [37]:

```
for i in list(city1):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> Alfenas
1 --> Araxá
2 --> Caratinga --> Coronel Fabriciano
3 --> Barbacena
4
5
6 --> São Francisco
7 --> Araguari --> Betim
8 --> Aimorés --> Curvelo --> São Lourenço
9 --> Arcos
10
11
12
13 --> Contagem
14 --> Bom Despacho
15
16 --> Boa Esperança
17
18 --> São João del Rei
19 --> Sabará
20
21
22 --> Conselheiro Lafaiete --> São Sebastião do Paraíso
23
24
25 --> Caeté
26 --> Santos Dumont --> Sete Lagoas
27 --> Andradas --> Brumadinho
28 --> Belo Horizonte
```

Simulação 2: todos os nomes das cidades em minúsculo

In [38]:

```
city2 = ['aimorés', 'alfenas', 'andradas', 'araguari', 'araxá', 'arcos',
         'barbacena', 'belo horizonte', 'betim', 'boa esperança', 'bom despacho',
         'brumadinho', 'caeté', 'caratinga', 'conselheiro lafaiete', 'contagem',
         'coronel fabriciano', 'curvelo', 'sabará', 'santos dumont',
         'são francisco', 'são joão del rei', 'são lourenço', 'são sebastião do paraíso',
         'sete lagoas']
```

Combinação 1

In [39]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [40]:

```
for i in list(city2):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> bom despacho
1
2 --> são sebastião do paraíso
3 --> araguari
4
5 --> alfenas --> contagem
6
7 --> betim
8 --> caeté --> conselheiro lafaiete
9 --> são lourenço
10 --> caratinga
11
12 --> barbacena
13
14 --> aimorés --> arcos --> curvelo
15 --> araxá
16 --> são francisco
17
18 --> andradas
19
20
21 --> brumadinho --> sabará
22 --> são joão del rei
23 --> belo horizonte --> sete lagoas
24 --> boa esperança
25 --> santos dumont
26 --> coronel fabriciano
27
28
```

Combinação 2

In [41]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [42]:

```
for i in list(city2):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> alfenas
1 --> araxá
2 --> caratinga --> coronel fabriciano
3 --> barbacena
4
5
6
7 --> araguari --> betim
8 --> aimorés --> curvelo
9 --> arcos --> são francisco
10
11 --> são lourenço
12
13 --> contagem
14
15
16
17 --> bom despacho
18
19 --> boa esperança --> sabará
20
21 --> são joão del rei
22
23
24
25 --> caeté --> conselheiro lafaiete --> são sebastião do paraíso
26 --> santos dumont --> sete lagoas
27 --> andradas --> brumadinho
28 --> belo horizonte
```

Simulação 3: todos os nomes das cidades em maiúsculo

In [43]:

```
city3 = ['AIMORÉS', 'ALFENAS', 'ANDRADAS', 'ARAGUARI', 'ARAXÁ', 'ARCOS',
        'BARBACENA', 'BELO HORIZONTE', 'BETIM', 'BOA ESPERANÇA', 'BOM DESPACHO', 'BRUMADINH
         'CAETÉ', 'CARATINGA', 'CONSELHEIRO LAFAIETE', 'CONTAGEM', 'CORONEL FABRICIANO',
         'CURVELO',  'SABARÁ', 'SANTOS DUMONT', 'SÃO FRANCISCO', 'SÃO JOÃO DEL REI',
         'SÃO LOURENÇO', 'SÃO SEBASTIÃO DO PARAÍSO', 'SETE LAGOAS']
```

Combinação 1

In [44]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [45]:

```
for i in list(city3):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> ARAXÁ
1
2
3 --> SABARÁ
4 --> CORONEL FABRICIANO
5 --> SÃO LOURENÇO
6
7
8 --> ARAGUARI
9 --> CONSELHEIRO LAFAIETE --> SÃO FRANCISCO
10 --> CONTAGEM
11
12 --> CARATINGA --> SÃO JOÃO DEL REI
13 --> ALFENAS --> BELO HORIZONTE
14 --> BARBACENA
15
16
17 --> BOA ESPERANÇA
18 --> SANTOS DUMONT
19
20 --> BRUMADINHO
21 --> BETIM
22 --> AIMORÉS --> CAETÉ --> CURVELO --> SETE LAGOAS
23 --> ANDRADAS
24
25 --> BOM DESPACHO
26 --> SÃO SEBASTIÃO DO PARAÍSO
27
28 --> ARCOS
```

Combinação 2

In [46]:

```
# Criaçaõ da tabela de hash com uma lista aninhada.
HashTable = [[] for _ in range(m)]
```

In [47]:

```
for i in list(city3):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> ARCOS
1 --> BOA ESPERANÇA --> CONSELHEIRO LAFAIETE --> CONTAGEM --> SÃO SEBASTIÃO
DO PARAÍSO
2 --> BOM DESPACHO
3
4
5
6 --> SÃO JOÃO DEL REI
7
8
9
10 --> CORONEL FABRICIANO --> SABARÁ
11 --> SANTOS DUMONT
12
13 --> BELO HORIZONTE
14 --> SETE LAGOAS
15 --> ANDRADAS --> BRUMADINHO
16 --> CAETÉ
17 --> ALFENAS
18
19 --> CARATINGA
20 --> BARBACENA --> SÃO FRANCISCO
21 --> ARAXÁ
22
23
24 --> ARAGUARI
25 --> AIMORÉS --> CURVELO --> SÃO LOURENÇO
26
27 --> BETIM
28
```

Simulação 4: nomes das cidades sem acento e sem caracteres especiais (ex.: "ç")

In [48]:

```
city4 = ['Aimores', 'Alfenas', 'Andradas', 'Araguari', 'Araxa', 'Arcos',
         'Barbacena', 'Belo Horizonte', 'Betim', 'Boa Esperanca', 'Bom Despacho', 'Brumadin
         'Caete', 'Caratinga', 'Conselheiro Lafaiete', 'Contagem', 'Coronel Fabriciano', 'C
         'Sabara', 'Santos Dumont', 'Sao Francisco', 'Sao Joao del Rei', 'Sao Lourenco',
         'Sao Sebastiao do Paraiso', 'Sete Lagoas']
```

Combinação 1

In [49]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [50]:

```
for i in list(city4):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> Araguari --> Araxa
1
2 --> Alfenas --> Boa Esperanca --> Conselheiro Lafaiete --> Contagem --> Sa
o Lourenco
3
4 --> Betim
5
6 --> Sabara
7 --> Caratinga --> Sao Sebastiao do Paraiso
8
9 --> Barbacena
10
11 --> Arcos --> Curvelo
12
13
14 --> Sao Joao del Rei
15 --> Andradas
16
17 --> Belo Horizonte --> Sete Lagoas
18 --> Brumadinho --> Caete
19 --> Santos Dumont
20 --> Coronel Fabriciano
21
22
23 --> Bom Despacho
24 --> Aimores
25 --> Sao Francisco
26
27
28
```

Combinação 2

In [51]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [52]:

```
for i in list(city4):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> Alfenas
1
2 --> Caratinga --> Coronel Fabriciano
3 --> Barbacena
4
5
6
7 --> Araguari --> Betim --> Sao Lourenco
8 --> Aimores --> Curvelo --> Sao Sebastiao do Paraiso
9 --> Arcos --> Caete
10
11
12
13 --> Contagem
14 --> Bom Despacho
15
16 --> Boa Esperanca
17
18 --> Araxa
19 --> Sabara --> Sao Joao del Rei
20
21 --> Sao Francisco
22 --> Conselheiro Lafaiete
23
24
25
26 --> Santos Dumont --> Sete Lagoas
27 --> Andradas --> Brumadinho
28 --> Belo Horizonte
```

Simulação 5: nomes das cidades em minúsculo, sem acento e sem caracteres especiais (ex.: "ç")

In [53]:

```
city5 = ['aimores', 'alfenas', 'andradas', 'araguari', 'araxa', 'arcos',
         'barbacena', 'belo Horizonte', 'betim', 'boa esperanca', 'bom despacho', 'brumadin
         'caete', 'caratinga', 'conselheiro lafaiete', 'contagem', 'coronel fabriciano', 'c
         'sabara', 'santos dumont', 'sao francisco', 'sao joao del rei', 'sao lourenco',
         'sao sebastiao do paraiso', 'sete lagoas']
```

Combinação 1

In [54]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [55]:

```
for i in list(city5):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> bom despacho
1
2 --> sao francisco
3 --> araguari --> araxa
4
5 --> alfenas --> contagem
6
7 --> betim
8 --> boa esperanca --> conselheiro lafaiete --> sao lourenco
9 --> sabara
10 --> caratinga
11
12 --> barbacena
13
14 --> arcos --> curvelo
15
16 --> sao sebastiao do paraiso
17
18 --> andradas
19
20 --> belo Horizonte
21 --> brumadinho --> caete
22
23 --> sao joao del rei --> sete lagoas
24
25 --> santos dumont
26 --> coronel fabriciano
27 --> aimores
28
```

Combinação 2

In [56]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [57]:

```
for i in list(city5):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> alfenas
1
2 --> caratinga --> coronel fabriciano
3 --> barbacena
4
5
6
7 --> araguari --> betim
8 --> aimores --> curvelo
9 --> arcos --> caete
10 --> sao lourenco
11 --> sao sebastiao do paraiso
12
13 --> contagem
14
15
16
17 --> bom despacho
18 --> araxa
19 --> boa esperanca --> sabara
20
21
22 --> sao joao del rei
23
24 --> sao francisco
25 --> conselheiro lafaiete
26 --> santos dumont --> sete lagoas
27 --> andradas --> brumadinho
28 --> belo Horizonte
```

Simulação 6: nomes das cidades em maiúsculo, sem acento e sem caracteres especiais (ex.: "ç")

In [58]:

```
city6 = ['AIMORES', 'ALFENAS', 'ANDRADAS', 'ARAGUARI', 'ARAXA', 'ARCOS',
        'BARBACENA', 'BELO HORIZONTE', 'BETIM', 'BOA ESPERANCA', 'BOM DESPACHO', 'BRUMADINH
        'CAETE', 'CARATINGA', 'CONSELHEIRO LAFAIETE', 'CONTAGEM', 'CORONEL FABRICIANO', 'CU
        'SABARA', 'SANTOS DUMONT', 'SAO FRANCISCO', 'SAO JOAO DEL REI', 'SAO LOURENCO',
         'SAO SEBASTIAO DO PARAISO', 'SETE LAGOAS']
```

Combinação 1

In [59]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [60]:

```
for i in list(city6):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0
1 --> BOA ESPERANCA
2
3
4 --> CORONEL FABRICIANO --> SAO LOURENCO
5
6 --> AIMORES --> CAETE
7
8 --> ARAGUARI
9 --> CONSELHEIRO LAFAIETE
10 --> CONTAGEM
11 --> SAO SEBASTIAO DO PARAISO
12 --> CARATINGA
13 --> ALFENAS --> BELO HORIZONTE --> SAO JOAO DEL REI
14 --> BARBACENA
15
16
17 --> ARAXA
18 --> SANTOS DUMONT
19
20 --> BRUMADINHO --> SABARA
21 --> BETIM
22 --> CURVELO --> SETE LAGOAS
23 --> ANDRADAS
24 --> SAO FRANCISCO
25 --> BOM DESPACHO
26
27
28 --> ARCOS
```

Combinação 2

In [61]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [62]:

```
for i in list(city6):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> ARCOS --> CAETE
1 --> BOA ESPERANCA --> CONSELHEIRO LAFAIETE --> CONTAGEM
2 --> BOM DESPACHO
3
4
5
6 --> SAO FRANCISCO
7 --> SAO JOAO DEL REI
8
9 --> ARAXA
10 --> CORONEL FABRICIANO --> SABARA
11 --> SANTOS DUMONT
12
13 --> BELO HORIZONTE
14 --> SETE LAGOAS
15 --> ANDRADAS --> BRUMADINHO
16 --> SAO SEBASTIAO DO PARAISO
17 --> ALFENAS
18
19 --> CARATINGA
20 --> BARBACENA
21
22
23
24 --> ARAGUARI --> SAO LOURENCO
25 --> AIMORES --> CURVELO
26
27 --> BETIM
28
```

Simulação 7: nomes das cidades sem espaçamento

In [63]:

```
city7 = ['Aimorés', 'Alfenas', 'Andradas', 'Araguari', 'Araxá', 'Arcos',
         'Barbacena', 'BeloHorizonte', 'Betim', 'BoaEsperança', 'BomDespacho', 'Brumadinho'
         'Caeté', 'Caratinga', 'ConselheiroLafaiete', 'Contagem', 'CoronelFabriciano', 'Cur
         'Sabará', 'SantosDumont', 'SãoFrancisco', 'SãoJoãodelRei', 'SãoLourenço',
         'SãoSebastiãodoParaíso', 'SeteLagoas']
```

Combinação 1

In [64]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [65]:

```
for i in list(city7):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> Araguari --> SãoLourenço
1
2 --> Alfenas --> Contagem
3
4 --> Betim --> SãoJoãodelRei
5 --> Caeté
6
7 --> Caratinga --> SãoFrancisco
8
9 --> Barbacena
10
11 --> Aimorés --> Arcos --> Curvelo
12 --> Araxá
13 --> SãoSebastiãodoParaíso
14 --> BeloHorizonte --> SeteLagoas
15 --> Andradas --> BoaEsperança
16 --> SantosDumont
17 --> CoronelFabriciano
18 --> Brumadinho --> Sabará
19
20 --> BomDespacho
21
22
23
24
25
26
27
28 --> ConselheiroLafaiete
```

Combinação 2

In [66]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [67]:

```
for i in list(city7):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> Alfenas
1 --> Araxá
2 --> Caratinga
3 --> Barbacena --> SantosDumont
4
5
6
7 --> Araguari --> Betim --> SãoLourenço
8 --> Aimorés --> Curvelo
9 --> Arcos
10
11
12
13 --> Contagem --> CoronelFabriciano
14
15
16
17 --> BoaEsperança
18
19 --> Sabará --> SeteLagoas
20
21
22 --> BomDespacho --> SãoJoãodelRei
23 --> SãoFrancisco
24 --> ConselheiroLafaiete
25 --> Caeté --> SãoSebastiãodoParaíso
26 --> BeloHorizonte
27 --> Andradas --> Brumadinho
28
```

Simulação 8: nomes das cidades em minúsculo e sem espaçamento

In [68]:

```
city8 = ['aimorés', 'alfenas', 'andradas', 'araguari', 'araxá', 'arcos',
         'barbacena', 'belohorizonte', 'betim', 'boaesperança', 'bomdespacho', 'brumadinho'
         'caeté', 'caratinga', 'conselheirolafaiete', 'contagem', 'coronelfabriciano', 'cur
         'sabará', 'santosdumont', 'sãofrancisco', 'sãojoãodelrei', 'sãolourenço',
         'sãosebastiãodoparaíso', 'setelagoas']
```

Combinação 1

In [69]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [70]:

```
for i in list(city8):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0
1
2
3 --> araguari
4
5 --> alfenas --> conselheirolafaiete --> contagem
6 --> sãolourenço
7 --> betim
8 --> caeté
9
10 --> caratinga
11
12 --> barbacena
13 --> sãofrancisco --> sãojoãodelrei
14 --> aimorés --> arcos --> curvelo
15 --> araxá
16
17
18 --> andradas
19
20 --> belohorizonte --> setelagoas
21 --> boaesperança --> brumadinho --> sabará
22 --> santosdumont --> sãosebastiãodoparaíso
23 --> coronelfabriciano
24
25
26 --> bomdespacho
27
28
```

Combinação 2

In [71]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [72]:

```
for i in list(city8):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> alfenas --> belohorizonte
1 --> araxá
2 --> caratinga
3 --> barbacena
4
5
6 --> santosdumont
7 --> araguari --> betim --> sãolourenço
8 --> aimorés --> curvelo
9 --> arcos
10
11
12
13 --> contagem --> coronelfabriciano
14
15
16
17 --> boaesperança
18
19 --> sabará
20
21
22 --> bomdespacho --> setelagoas
23 --> sãofrancisco
24 --> conselheirolafaiete
25 --> caeté --> sãojoãodelrei --> sãosebastiãodoparaíso
26
27 --> andradas --> brumadinho
28
```

Simulação 9: nomes das cidades em maiúsculo e sem espaçamento

In [73]:

```
city9 = ['AIMORÉS', 'ALFENAS', 'ANDRADAS', 'ARAGUARI', 'ARAXÁ', 'ARCOS',
         'BARBACENA', 'BELOHORIZONTE', 'BETIM', 'BOAESPERANÇA', 'BOMDESPACHO', 'BRUMADINHO'
         'CAETÉ', 'CARATINGA', 'CONSELHEIROLAFAIETE', 'CONTAGEM', 'CORONELFABRICIANO', 'CUR
         'SABARÁ', 'SANTOSDUMONT', 'SÃOFRANCISCO', 'SÃOJOÃODELREI', 'SÃOLOURENÇO',
         'SÃOSEBASTIÃODOPARAÍSO', 'SETELAGOAS']
```

Combinação 1

In [74]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [75]:

```
for i in list(city9):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> ARAXÁ
1 --> CORONELFABRICIANO
2 --> SÃOLOURENÇO
3 --> SABARÁ --> SÃOJOÃODELREI
4
5
6 --> CONSELHEIROLAFAIETE --> SÃOFRANCISCO
7
8 --> ARAGUARI
9
10 --> BELOHORIZONTE --> CONTAGEM
11
12 --> CARATINGA
13 --> ALFENAS
14 --> BARBACENA --> BOAESPERANÇA
15 --> SANTOSDUMONT
16
17 --> SÃOSEBASTIÃODOPARAÍSO
18
19 --> SETELAGOAS
20 --> BRUMADINHO
21 --> BETIM
22 --> AIMORÉS --> BOMDESPACHO --> CAETÉ --> CURVELO
23 --> ANDRADAS
24
25
26
27
28 --> ARCOS
```

Combinação 2

In [76]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [77]:

```
for i in list(city9):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> ARCOS --> CONSELHEIROLAFAIETE
1 --> CONTAGEM --> SÃOSEBASTIÃODOPARAÍSO
2 --> BOAESPERANÇA
3
4
5
6
7 --> BOMDESPACHO --> SÃOJOÃODELREI
8 --> SÃOFRANCISCO
9
10 --> SABARÁ --> SETELAGOAS
11 --> BELOHORIZONTE
12
13
14
15 --> ANDRADAS --> BRUMADINHO
16 --> CAETÉ
17 --> ALFENAS
18
19 --> CARATINGA
20 --> BARBACENA --> SANTOSDUMONT
21 --> ARAXÁ --> CORONELFABRICIANO --> SÃOLOURENÇO
22
23
24 --> ARAGUARI
25 --> AIMORÉS --> CURVELO
26
27 --> BETIM
28
```

Simulação 10: nomes das cidades sem espaçamento e sem caracteres especiais (acentos e "ç")

In [78]:

```
city10 = ['Aimores', 'Alfenas', 'Andradas', 'Araguari', 'Araxa', 'Arcos',
          'Barbacena', 'BeloHorizonte', 'Betim', 'BoaEsperanca', 'BomDespacho', 'Brumadinho
          'Caete', 'Caratinga', 'ConselheiroLafaiete', 'Contagem', 'CoronelFabriciano', 'Cu
          'Sabara', 'SantosDumont', 'SaoFrancisco', 'SaoJoaodelRei', 'SaoLourenco',
          'SaoSebastiaodoParaiso', 'SeteLagoas']
```

Combinação 1

In [79]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [80]:

```
for i in list(city10):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0 --> Araguari --> Araxa
1
2 --> Alfenas --> Contagem
3
4 --> Betim
5 --> SaoJoaodelRei
6 --> Sabara
7 --> Caratinga
8
9 --> Barbacena
10
11 --> Arcos --> Curvelo
12
13
14 --> BeloHorizonte --> SeteLagoas
15 --> Andradas
16 --> SantosDumont
17 --> CoronelFabriciano
18 --> Brumadinho --> Caete
19
20 --> BomDespacho
21
22 --> SaoFrancisco
23
24 --> Aimores
25
26
27 --> SaoSebastiaodoParaiso
28 --> BoaEsperanca --> ConselheiroLafaiete --> SaoLourenco
```

Combinação 2

In [81]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [82]:

```
for i in list(city10):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> Alfenas
1 --> BoaEsperanca
2 --> Caratinga
3 --> Barbacena --> SantosDumont
4
5
6
7 --> Araguari --> Betim
8 --> Aimores --> Curvelo --> SaoJoaodelRei
9 --> Arcos --> Caete --> SaoFrancisco
10 --> SaoSebastiaodoParaiso
11
12
13 --> Contagem --> CoronelFabriciano
14
15
16
17
18 --> Araxa
19 --> Sabara --> SeteLagoas
20
21
22 --> BomDespacho --> SaoLourenco
23
24 --> ConselheiroLafaiete
25
26 --> BeloHorizonte
27 --> Andradas --> Brumadinho
28
```

Simulação 11: nomes das cidades em minúsculo sem espaçamento e sem caracteres especiais (acentos e "ç")

In [83]:

```
city11 = ['aimores', 'alfenas', 'andradas', 'araguari', 'araxa', 'arcos',
          'barbacena', 'belohorizonte', 'betim', 'boaesperanca', 'bomdespacho', 'brumadinho
          'caete', 'caratinga', 'conselheirolafaiete', 'contagem', 'coronelfabriciano', 'cu
          'sabara', 'santosdumont', 'saofrancisco', 'saojoaodelrei', 'saolourenco',
          'saosebastiaodoparaiso', 'setelagoas']
```

Combinação 1

In [84]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [85]:

```
for i in list(city11):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0
1
2
3 --> araguari --> araxa
4
5 --> alfenas --> boaesperanca --> conselheirolafaiete --> contagem --> saol
ourenco
6
7 --> betim --> saosebastiaodoparaiso
8
9 --> sabara
10 --> caratinga
11
12 --> barbacena
13
14 --> arcos --> curvelo --> saojoaodelrei
15
16
17
18 --> andradas
19
20 --> belohorizonte --> setelagoas
21 --> brumadinho --> caete
22 --> santosdumont
23 --> coronelfabriciano
24
25
26 --> bomdespacho
27 --> aimores
28 --> saofrancisco
```

Combinação 2

In [86]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [87]:

```
for i in list(city11):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> alfenas --> belohorizonte
1 --> boaesperanca
2 --> caratinga
3 --> barbacena
4
5
6 --> santosdumont
7 --> araguari --> betim
8 --> aimores --> curvelo
9 --> arcos --> caete --> saofrancisco
10 --> saosebastiaodoparaiso
11 --> saojoaodelrei
12
13 --> contagem --> coronelfabriciano
14
15
16
17
18 --> araxa
19 --> sabara
20
21
22 --> bomdespacho --> saolourenco --> setelagoas
23
24 --> conselheirolafaiete
25
26
27 --> andradas --> brumadinho
28
```

Simulação 12: nomes das cidades em maiúsculo sem espaçamento e sem caracteres especiais (acentos e "Ç")

In [88]:

```
city12 = ['AIMORES', 'ALFENAS', 'ANDRADAS', 'ARAGUARI', 'ARAXA', 'ARCOS',
          'BARBACENA', 'BELOHORIZONTE', 'BETIM', 'BOAESPERANCA', 'BOMDESPACHO', 'BRUMADINHO
          'CAETE', 'CARATINGA', 'CONSELHEIROLAFAIETE', 'CONTAGEM', 'CORONELFABRICIANO', 'CU
          'SABARA', 'SANTOSDUMONT', 'SAOFRANCISCO', 'SAOJOAODELREI', 'SAOLOURENCO',
          'SAOSEBASTIAODOPARAISO', 'SETELAGOAS']
```

Combinação 1

In [89]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [90]:

```
for i in list(city12):
    insertHash(HashTable, ascii_number(i), i)

printHash(HashTable)
```

```
0
1 --> CORONELFABRICIANO --> SAOLOURENCO
2 --> SAOSEBASTIAODOPARAISO
3
4 --> SAOJOAODELREI
5
6 --> AIMORES --> CAETE --> CONSELHEIROLAFAIETE
7
8 --> ARAGUARI
9
10 --> BELOHORIZONTE --> CONTAGEM
11
12 --> CARATINGA
13 --> ALFENAS
14 --> BARBACENA
15 --> SANTOSDUMONT
16
17 --> ARAXA
18
19 --> SETELAGOAS
20 --> BRUMADINHO --> SABARA
21 --> BETIM --> SAOFRANCISCO
22 --> BOMDESPACHO --> CURVELO
23 --> ANDRADAS
24
25
26
27 --> BOAESPERANCA
28 --> ARCOS
```

Combinação 2

In [91]:

```
# Criaçaõ da tabela de hash com uma lista aninhada
HashTable = [[] for _ in range(m)]
```

In [92]:

```
for i in list(city12):
    insertHash(HashTable, ascii_prime(i, len(i)), i)

printHash(HashTable)
```

```
0 --> ARCOS --> CAETE --> CONSELHEIROLAFAIETE
1 --> CONTAGEM
2
3
4
5
6
7 --> BOMDESPACHO --> SAOLOURENCO
8
9 --> ARAXA
10 --> SABARA --> SETELAGOAS
11 --> BELOHORIZONTE
12
13
14
15 --> ANDRADAS --> BOAESPERANCA --> BRUMADINHO --> SAOSEBASTIAODOPARAISO
16
17 --> ALFENAS
18
19 --> CARATINGA
20 --> BARBACENA --> SANTOSDUMONT
21 --> CORONELFABRICIANO
22 --> SAOJOAODELREI
23 --> SAOFRANCISCO
24 --> ARAGUARI
25 --> AIMORES --> CURVELO
26
27 --> BETIM
28
```

Questão 7 (Questão de Pesquisa)

A estrutura `heap` é um tipo particular de árvore binária, onde (i) o valor em um nó não é menor do que os valores armazenados em seus filhos, (ii) a árvore é perfeitamente balanceada e as folhas do último nível estão todas nas posições mais à esquerda. Qual é o número de comparações e de trocas, no melhor caso, para se criar um heap usando (a) o método de Williams (Algorithm 232: Heapsort) e (b) o método de Floyd (Algorithm 245: Treesort 3)?

Contextualização:

Segundo Drozdeck (2013), o heap é um tipo particular de árvore binária que apresenta as seguintes propriedades (que definem um head de máximo):

- O valor de cada nó é maior ou igual aos valores armazenados em cada um de seus filhos;
- A árvore é perfeitamente balanceada e as folhas no último nível estão, todas, localizadas nas posições mais a esquerda.

Obs.: no caso do heap de mínimo, o valor de cada nó será menor ou igual aos valores armazenados em cada um de seus filhos.

A partir das informações supracitadas, é possível fazer uma comparação entre heap de máximo e heap de mínimo. Neste sentido, enquanto que no heap de máximo a raiz contém o maior elemento, no heap de mínimo a raíz contém o menor elemento. Por fim, o número de níveis de uma árvore que possui propriedade de heap é $O(\log_2 n)$. Diante disso, as operações básicas são executadas em tempo que é, no máximo, proporcional à altura da árvore, demorando um tempo $O(\log_2 n)$ para serem executadas (Cormen et al., 2012).

Contudo, os elementos contidos em um heap não estão perfeitamente ordenados. Neste sentido, é conhecido somente que o maior elemento está na raiz e que, para cada nó, todos os seus descendentes são menores ou iguais ao referido nó (pai).

Resposta:

Heapsort (método de Williams)

Segundo Tenembaum et al. (2004), o heapsprt é uma classificação in loco que exige somente $O(n \log_2 n)$ operações independentemente da ordem de entrada. Visto que o próprio heap pode ser implementado dentro do vetor de entrada (utilizando uma implementação sequencial de árvore binária quase completa), o único espaço extra necessátio é para as variáveis do programa.

Para a análise do heapsort, a árvore binária completa com n nós (onde n é um a menos que uma potência de 2) tem $\log_2(n+1)$ níveis. Por conseguinte, se cada elemento no vetor fosse uma folha, exigindo que fosse filtrado pela árvore interia dirante a criação e o ajuste do heap, a classificação ainda seria $O(n \log_2 n)$.

De acordo com Drozdeck (2013), heaps podem ser implementados como arrays e, nesse sentido, cada heap é um array (contudo, nem todo array é um heao). Diante disso, no heapsort é necessário converter um array em um heap, organizando o dado no array de modo que a organização resultante represente um heap. Uma forma de fazer é utilizando o método proposto por Williams (método de cima para baixo), onde inicia-se com um heap vazio e, posteriormente, os elementos são incluidos sequencialmente elementos em um heap em expansão. Assim, o heap é expandido empilhando novos elementos ao heap.

Antes de analisar o melhor caso, é importante discutir acerca do pior caso, que ocorre quando um novo elemento adicionado tem de ser movido para a raiz da árvore, $\lfloor \log_2 k \rfloor$ mudanças são realizadas em um heap de k nós. Assim, se n elementos são empilhados (considerando o pior caso),

$$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor \leq \sum_{k=1}^{n} \log_2 k = \log_2 1 + \ldots \log_2 n = \log_2(n!) = O(n \log_2 n)$$

mudanças são realizads durante a execução do algoritmo e o mesmo número de comparações. Diante disso, é possível inferir que no melhor caso há zero mudanças, visto que espera-se que a adição do novo elemento não implique em movê-lo para a raiz da árvore (permancenedo no nível mais baixo do heap e, portanto, como folha). Por outro lado, o número de comparações é o mesmo do pior caso, ou seja, $O(n \log_2 n)$, conforme mencionado anteriormente e em linha com o trabalho de Bollobás et al. (1996).

Treesort (método de Floyd)

Posteriormente, Floyd desenvolveu um algoritmo no qual o heap é construído de de baixo para cima. Assim, pequenos heaps são formados e repetidamente são concatenados aos heaps maiores (Drozdeck, 2013). Diante disso, o processo inicia-se a partir do último nó da folha (`data[(n/2) - 1]`), com n sendo o tamanho do array. Se `data[(n/2) - 1]` é menor que seu nó filho, então é realizada a troca entre estes dois nós (nó pai e nó filho). Depois de realizar a troca, uma nova árvore (subárvore) é criada. Esse procedimento se repete até que nenhuma troca entre nós pai e filho seja necessária. Além disso, antes do elemento ser considerado, suas subárvores já foram convertidas em heaps.

Assume-se que uma árvore binária completa é criada ($n = 2^k - 1$, para algum k). Para criar o heap, a função para fazer a troca entre os nós pai e filho é chamada $(n + 1)/2$, um para cada nó não folha. Contudo, antes de analisar o melhor caso, é importante discutir acerca do pior caso. No pior caso, esta função move o dado do próximo para o último nível, consistindo de $(n + 1)/4$ nós, para baixo em um nível para o nível da folha, realizando $(n + 1)/4$ trocas. Portanto, todos os nós deste nível fazem $(n + 1)/4$ movimentos. Dados do segundo ao último nível, que tem $(n + 1)/8$ nós, são movidos dois níveis abaixo para alcançar o nível das folhas. Portanto, os nós deste nível realizam $2.(n + 1)/8$ movimentos e, assim por diante, até a raiz.

A raiz da árvore, à medida que a árvore se torna um heap, é movida, no pior caso, $\log_2(n + 1) - 1 = \log_2(n + 1)/2$ níveis abaixo da árvore para terminar em uma das folhas. Como há apenas uma raiz, isso contribui com $\log_2(n + 1)/2$ movimentos. O número total de movimentos pode ser dado pela soma:

$$\sum_{i=2}^{\log_2(n+1)} \frac{n+1}{2^i}(i-1) = (n+1) \sum_{i=2}^{\log_2(n+1)} \frac{i-1}{2^i}$$

que é $O(n)$.

Diante disso, é possível inferir que no melhor caso haja zero mudanças, visto que espera-se que a raiz da árvore não precisa ser movidaníveis abaixo da árvore para terminar em uma das folhas, ou seja, a árvore já se encontra balanceada. Por outro lado, o número de comparações é o mesmo observado para o melhor caso do algoritmo Heapsort, ou seja, $O(n \log_2 n)$.

Para um array que não é uma árvore binária completa, a complexidade é ainda mais limitada por $O(n)$. O pior caso para comparações é 2 vezes este valor, que também é $O(n)$, visto que para cada nó envolvido na troca entre nós pai e filho, os filhos são comparados aos outros para escolher o maior. Por outro lado, olhando para o melhor caso é possível inferir que o número de trocas continua zero, visto que é esperado que a árvore já esteja balanceada, e o número de comparações passa a ser $O(n)$.

Referências

- Bollobás, B., Fenner, T.I., & Frieze, A.M. (1996). *On the Best Case of Heapsort.* Journal od Algorithms 20, pp. 205-217.
- Celes, W., Cerqueira, R., & Rangel, J.L. (2016). *Introdução a Estrutura de Dados com técnicas de programação em C.* LTC - 2ª edição.
- Cormen, T.H, Leiserson, C.E, Rivest, R.L, & Stein, C. (2012). *Algoritmos: Teoria e Prática*. Elsevier - 11ª tiragem.
- Drozdek, A. (2013). *Data Structures and Algorithms in C++.* Cengage Learning - 4ª edição.
- Tenenbaum, A.M, Langsam, Y., & Augenstein, M.J. (2004). *Estruturas de dados usando C.* Pearson Makron Books.