ALESSANDRO SOARES DA SILVA

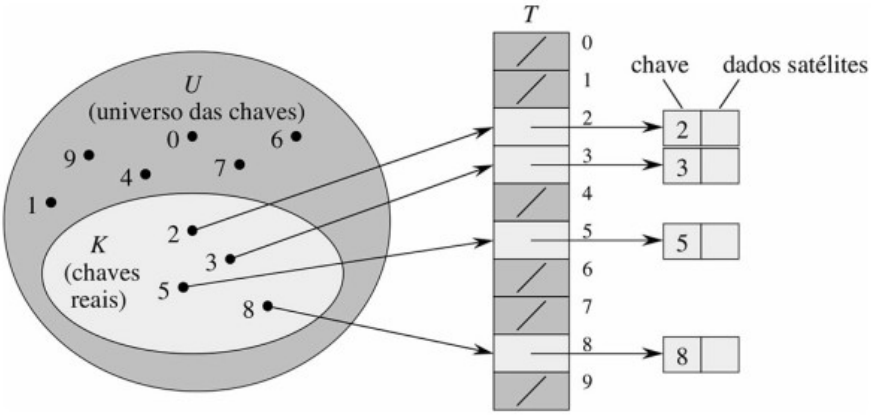
MATRICULA: 20231023705

5° LISTA DE ALGORITMO

1. Proponha um problema que precisa ser resolvido com um dicionário. Use implementações existentes de diversos tipos de dicionários para resolver o problema proposto e avalie os resultados de desempenho para as operações básicas de inserção, remoção, e busca.

Definição de dicionário em estrutura de dados:

Algoritmos podem exigir a execução de vários tipos diferentes de operações em conjuntos. Por exemplo, muitos algoritmos precisam apenas da capacidade de inserir e eliminar elementos em um conjunto e testar a pertinência de elementos a um conjunto. Damos o nome de ***dicionário*** ao conjunto dinâmico que suporta essas operações.



O endereçamento direto é uma técnica simples que funciona bem quando o universo U de chaves é razoavelmente pequeno. Suponha que uma aplicação necessite de um conjunto dinâmico no qual cada elemento tem uma chave extraída do universo $U = \{0, 1, ..., m - 1\}$, onde m não é muito grande. Consideramos que não há dois elementos com a mesma chave.

A implementação das operações de dicionário é trivial.

```
DIRECT-ADDRESS-SEARCH(T, k)
1   return T[k] = x

DIRECT-ADDRESS-INSERT(T, x)
1   T[x.chave] = x

DIRECT-ADDRESS-DELETE(T, x)
1   T[x.chave] = NIL
```

Cada uma dessas operações leva somente o tempo $O(1)$.

1 – INSERIR

```python
def adicionar(lista, email, nome, telefone):
    while True:
        email = email
        if not existe_contato(lista, email):
            break
        else:
            print("\nEsse e-mail já foi utilizado.")
            print("Por favor, tente um novo e-mail")

        # Nesse passo, o e-mail recebido será único
    contato = {
        "email": email,
        "nome": nome,
        "telefone": telefone
    }
    lista.append(contato)
    print("\nO contato {} foi cadastrado com sucesso!\n".format(contato["nome"]))
```

2- EXLUIR

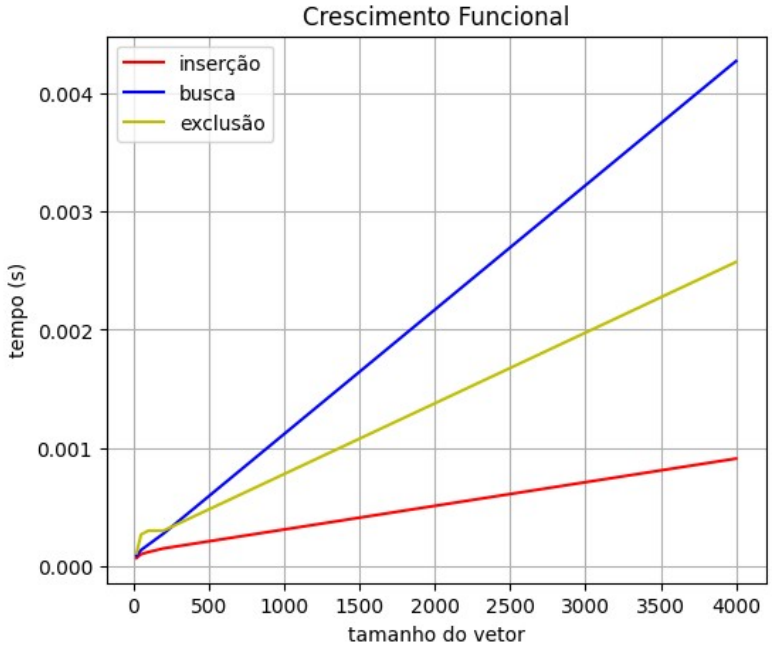
```python
def excluir(lista, email):
    print(" == Excluir Contato ==")
    if len(lista) > 0:
        email = email#input("Digite o e-mail do contato a ser excluído: ")
        if existe_contato(lista, email):
            print("\nO contato foi encontrado. Segue as informações: ")
            for i, contato in enumerate(lista):
                if contato['email'] == email:
                    print("================================================")
                    print("Nome: {}".format(contato['nome']))
                    print("Email: {}".format(contato['email']))
                    print("Telefone: {}".format(contato['telefone']))
                    print("================================================\n")
                    del lista[i]
            print("O contato foi excluído com sucesso!\n")
        else:
            print("\nNão existe contato cadastrado no sistema com e-mail de {}.\n".format(email))
    else:
        print("Não existe nenhum contato cadastrado no sistema.\n")
1 usage
```

3 – BUSCAR

```python
def buscar(lista, email):
    print(" == Buscar Contato ==")
    if len(lista) > 0:
        email = email #input("Digite o e-mail do contato a ser encontrado: ")
        if existe_contato(lista, email):
            print("\nO contato foi encontrado. Segue as informações: ")
            for contato in lista:
                if contato['email'] == email:
                    print("================================================")
                    print("Nome: {}".format(contato['nome']))
                    print("Email: {}".format(contato['email']))
                    print("Telefone: {}".format(contato['telefone']))
                    print("================================================\n")
        else:
            print("\nNão existe contato cadastrado no sistema com e-mail de {}.\n".format(email))
    else:
        print("Não existe nenhum contato cadastrado no sistema.\n")
```

Definição do problema: Na atividade numero 1 implementei uma lista de contatos, usando estruturas de listas e dicionários em python. Acabei deixando o algoritmo muito extenso com a inserção de um menu, o que fato não era o objetivo da questão, mas me ajudou a melhorar meu entendimento e prática de algoritmos.

Avaliação dos resultados:



Inserção: No algoritmo a chave era o e-mail de cada contato, coloquei validação no momento da inserção de modo a garantir que não ouvesse duplicação de contatos, mas não limitei o tamanho do dicionário e também não implementei nenhuma função ou tabela hash, e como esperado o gráfico aumentou a medida que o vetor cresce.

Busca: Tive a ideia de implementar uma lista de contato, mas como informei não limitei o tamanho do vetor e também não gerei nenhuma tabela ou função hash, e segundo o gráfico fiquei supreso que a busca tenha ficado tenha seja mais lenta que a exlusão, pois na exclução existe uma operação a mais que é a de apagar o elemento.

Remoção: A exclusão usa a busca seguido de uma operação de delete do valor, o na teorio, o gráfico deveria estar invertido com a busca, mas não é o que podemos observar. No entanto, está coerente, cresce a medida que o tamanho do dicionário cresce.

2. Implemente uma tabela *hash* usando funções *hash* e usando endereçamento aberto. Realize experimentos para mostrar numericamente as vantagens e desvantagens de cada caso.

SEGUE UM PEQUENO RESUMO

Busca: Muitos métodos de busca funcionam desta forma: procurar a informação desejada com base na comparação de suas chaves, ou seja, com base em algum valor que a compoe

Problema: algoritmos eficientes exigem que os elementos sejam armazenados de forma ordenada.
- custo de ordenação no melhor caso: nlogn

-custo da busca: logn

Busca ideal: Acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves: custo O(1)

Arrays: estruturas que utilizam “indices” para armazenar informação, permite acesar um determinada posição do array com custo O(1)

Problema: arrays não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada, de modo que a operação de busca em uma array não é O(1)

Solução: usar tabela hash: para espalhar os elementos que queremos armazenar na tabela

função: função que espalha os elementos na tabela

vantagens:

-alta eficiência na operação de busca

-tempo de busca é praticamente independente do numero de chaves armazenadas na tabela, e tem implementação simples

desvantagens:

-alto custo para recuperar os elementos da tabela ordenada pela chave.Nesse caso é preciso ordenar a tabela.

-Pior caso : é o O(n): alto numero de colisões

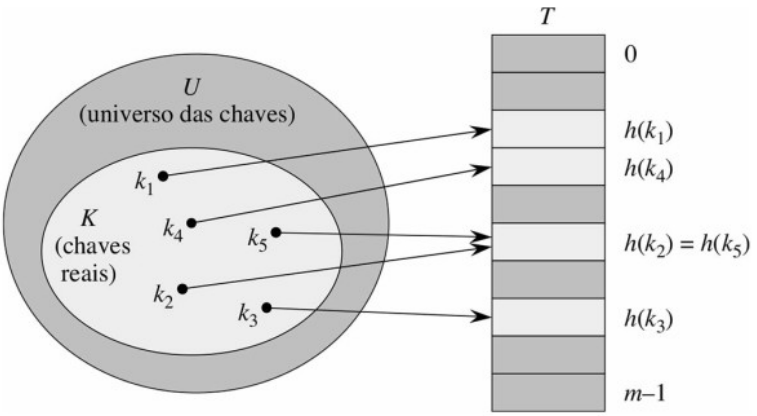


Figura 11.2 Utilização de uma função hash h para mapear chaves para posições de uma tabela de espalhamento. Como são mapeadas para a mesma posição, as chaves k_2 e k_5 colidem.

FUNÇÃO HASH:

Uma função hash é uma função matemática que transforma um conjunto de dados (normalmente uma sequência de caracteres ou números) em um valor fixo de tamanho fixo. Essa saída de tamanho fixo é chamada de "hash code" ou "digest". A função hash é projetada de tal forma que, para um conjunto de dados de entrada específico, o hash resultante seja sempre o mesmo. Além disso, uma boa função hash tende a distribuir uniformemente as saídas de hash para diferentes entradas, minimizando colisões (quando duas entradas diferentes produzem o mesmo hash).

ENDEREÇAMENTO ABERTO:

O endereçamento aberto (em inglês, "open addressing") é uma técnica usada em estruturas de dados, como tabelas de dispersão (hash tables), para resolver colisões. Colisões ocorrem quando dois ou mais elementos têm a mesma função hash, ou seja, são mapeados para a mesma posição da tabela de dispersão. O endereçamento aberto lida com colisões inserindo o item em outra posição da tabela quando ocorre uma colisão, em vez de usar estruturas de dados adicionais, como listas vinculadas, para resolver as colisões.

Vantagens do Endereçamento Aberto:

1. **Uso eficiente de memória:** O endereçamento aberto pode ser mais eficiente em termos de memória do que outras técnicas, como listas vinculadas, porque não requer estruturas de dados adicionais para armazenar itens colididos.
2. **Acesso mais rápido:** A busca em tabelas de dispersão com endereçamento aberto pode ser mais rápida do que a busca em estruturas de dados mais complexas, como listas vinculadas, quando não há muitas colisões.
3. **Simplicidade:** A implementação do endereçamento aberto geralmente é mais simples em comparação com outras técnicas, como encadeamento separado (separating chaining).

Desvantagens do Endereçamento Aberto:

1. **Limitações de carga (load factor):** O desempenho do endereçamento aberto pode degradar rapidamente à medida que a tabela de dispersão fica mais cheia (carga aumenta). Isso pode levar a um aumento nas colisões e, consequentemente, a um pior desempenho.
2. **Complexidade no tratamento de remoções:** O tratamento de remoções em tabelas de dispersão com endereçamento aberto pode ser complexo, pois você deve considerar como marcar as posições de elementos removidos sem quebrar a busca subsequente.
3. **Necessidade de estratégias de sondagem adequadas:** A escolha da estratégia de sondagem (probing strategy) adequada é crítica para o desempenho do endereçamento aberto. Uma estratégia de sondagem inadequada pode resultar em colisões frequentes e desempenho ruim.
4. **Colisões agrupadas:** Em situações onde a função de hash não é bem distribuída ou quando os dados têm padrões de distribuição específicos, o endereçamento aberto pode resultar em colisões agrupadas, onde muitos elementos são mapeados para posições próximas, levando a uma degradação significativa do desempenho.
5. **Rehashing caro:** Quando a tabela de dispersão fica muito cheia, é necessário redimensioná-la (rehashing), o que pode ser uma operação custosa em termos de tempo e memória.

Implementação em Python:

Tabela Hash

```python
class HashNode:
    def __init__(self, key, value):
        self.key = key
        self.values = [value]  # Usamos uma lista para armazenar múltiplos valores
        self.next = None
def create_hash_table(size):
    # Inicializa uma tabela hash com 'size' slots, todos inicializados como None
    hash_table = [None] * size
    return hash_table
def hash_function(key, size):
    # Função hash simples que retorna o índice com base na chave
    return key % size
```

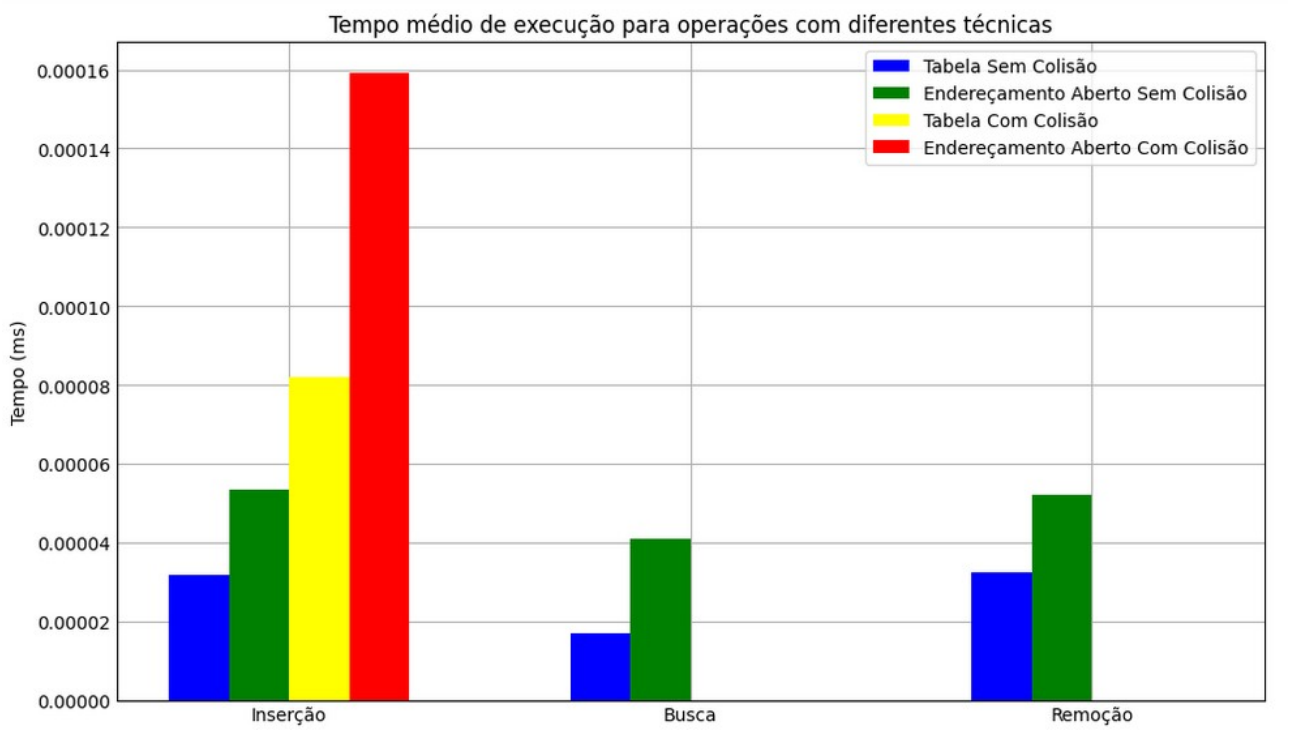
```python
def insert(hash_table, key, value):
    size = len(hash_table)
    index = hash_function(key, size)
    if hash_table[index] is None:
        # Se o slot estiver vazio, cria um novo nó com uma lista de valores
        hash_table[index] = HashNode(key, value)
    else:
        # Se houver colisão, insere o valor na lista de valores
        print("houve colisão")
        current = hash_table[index]
        while current.next is not None:
            if current.key == key:
                current.values.append(value)
                return
            current = current.next
        if current.key == key:
            current.values.append(value)
        else:
            current.next = HashNode(key, value)
def search(hash_table, key):
    size = len(hash_table)
    index = hash_function(key, size)
    current = hash_table[index]
    while current is not None:
        if current.key == key:
            return current.values
        current = current.next
    # Chave não encontrada
    return None
def delete(hash_table, key, value=None):
    size = len(hash_table)
    index = hash_function(key, size)
    current = hash_table[index]
    prev = None
    while current is not None:
        if current.key == key:
            if value is None or (value in current.values):
                if len(current.values) > 1 and value is not None:
                    # Se houver múltiplos valores para a chave e um valor específico for fornecido,
                    # remova apenas esse valor da lista de valores
                    current.values.remove(value)
                else:
                    # Se não houver valor específico fornecido ou se houver apenas um valor na lista,
                    # remova o nó inteiro
                    if prev is None:
                        # Remove o primeiro nó da lista
                        hash_table[index] = current.next
                    else:
                        prev.next = current.next
                return

        prev = current
        current = current.next
```

Endereçamento Aberto (Probing Linear)

```python
def cria_contato_lista(tamanho):
    # Inicializa uma lista de tamanho 'size' com None
    contato_lista = [None] * tamanho
    return contato_lista
def se_cheio(contato_lista):
    # Verifica se a lista de contatos está cheia
    return all((contato is not None) and (contato != "delete") for contato in contato_lista)
def hash_funcao(email, tamanho):
    # Função hash simples que retorna o índice com base no e-mail
    return hash(email) % tamanho
def insere_contato(contato_lista, email, nome, telefone):
    tamanho = len(contato_lista)
    if se_cheio(contato_lista):
        print("A lista de contatos está cheia. Não é possível inserir mais contatos.")
        return
        breakpoint()
    index = hash_funcao(email, tamanho)
    while contato_lista[index] is not None and contato_lista[index] != "delete":
        # Sondagem linear
        index = (index + 1) % tamanho
        print("ocorreu colisão")
    contato_lista[index] = {"email": email, "nome": nome, "telefone": telefone}
def procura_contato(contato_lista, email):
    tamanho = len(contato_lista)
    index = hash_funcao(email, tamanho)
    while contato_lista[index] is not None:
        contato = contato_lista[index]
        if contato == "delete":
            # Ignora posições marcadas como "delete"
            index = (index + 1) % tamanho
            continue
        if contato["email"] == email:
            return contato
        index = (index + 1) % tamanho
    # Valor não encontrado
    print(f"Contato com e-mail '{email}' não encontrado.")
    return None
def delete_contato(contato_lista, email):
    tamanho = len(contato_lista)
    index = hash_funcao(email, tamanho)
    while contato_lista[index] is not None:
        contato = contato_lista[index]
        if contato == "delete":
            # Ignora posições marcadas como "delete"
            index = (index + 1) % tamanho
            continue
        if contato["email"] == email:
            # Marcando o contato como excluído
            contato_lista[index] = "delete"
            return
        index = (index + 1) % tamanho
def mostra_menu():
    print('\n=== Agenda Telefônica ===')
    print("1. Inserir Contato")
    print("2. Buscar Contato")
    print("3. Excluir Contato")
    print("4. Sair")
# Exemplo de uso com menu
contato_lista = cria_contato_lista(500)
```

Resultados:



(Analisando numericamente os resultados):

→ **Vantagens:**

Tabela Hash: Para esta implementação de tabela com função hash, o tempo de processamento necessário foi menor nas operações de inserção, busca e remoção se comparado com o endereçamento aberto (Probing Linear).

Endereçamento aberto: Pelo que pude perceber, a vantagem está no fato de acupar um espaço menor de memória e conseguir contornar as colisões, quando elas acontecem, e quanto menor o tamanho do vetor mais colisões ocorrem, isso parece óbvil, uma vez que precisamos sempre ter o controle da carga de armazenamento, afim de evitar a sondagem linear

→ **Desvantagens:**

Tabela Hash: Teoricamente com o aumento do número de colisões, o tempo gasto para inserir um elemento aumenta de forma menos equilibrada se comparado a mesma solução com endereçamento aberto, no entanto não foi o que eu verifiquei, mesmo com vetores maiores, ainda sim o tempo de inserção sempre foi menor que a sondagem linear, mesmo com colisões.

Endereçamento aberto: Precisa de mais operações/funções de forma a elevar o tempo computacional gasto na operação de insersão se comparado com a mesma solução utilizando tabela hash, uma vez que o número de colisões aumenta a medida que o vetor se torna carregado.

3. Compare implementações (existentes ou sua própria) de tabelas *hash* com *hashing* duplo e *probing* linear. Observe e justifique os resultados dos seus experimentos em máquinas modernas.

Teoria

Hash Probing:

- Vantagens:

  1. Simplicidade: É relativamente simples de impementar

  2. Baixa sobrecarga de mémoria, geralmente requer menos espaço adicional na tabela de hash.

  3. Pode ser eficaz em cargas leves de tabela de hash.

- Desvantegens:

  1. Possibilidade de agrupamento: Colisões frequentes podem resultar em agrupamentos de elementos na tabela, o que diminui a eficiência.

  2. Desempenho variável: O desempenho depende da qualidade da função de hash e da distribuição dos valores inseridos.

Duplo Hash:

- Vantagens:

  1. Reduz agrupamento: Tende a reduzir o agrupamento de elementos, pois usa uma segunda função de hash para encontrar um novo local em caso de colisão;

  2. Melhor desempenho: Em muitos casos, pode ter um desempenho melhor do que o probing linear.

- Desvantagens:

  1. Complexidade: A implementação é um pouco mais complexa do que o probing linear.

  2. Possível sobrecarga de memória: Pode exigir mais espeço adicional para armazenar a segunda função de hash.

IMPLEMENTAÇÃO:

O código de probing Linear já foi mostrado no exercício 2, portanto na questão 3 está sendo mostrado somente a implementação do duplo hash.

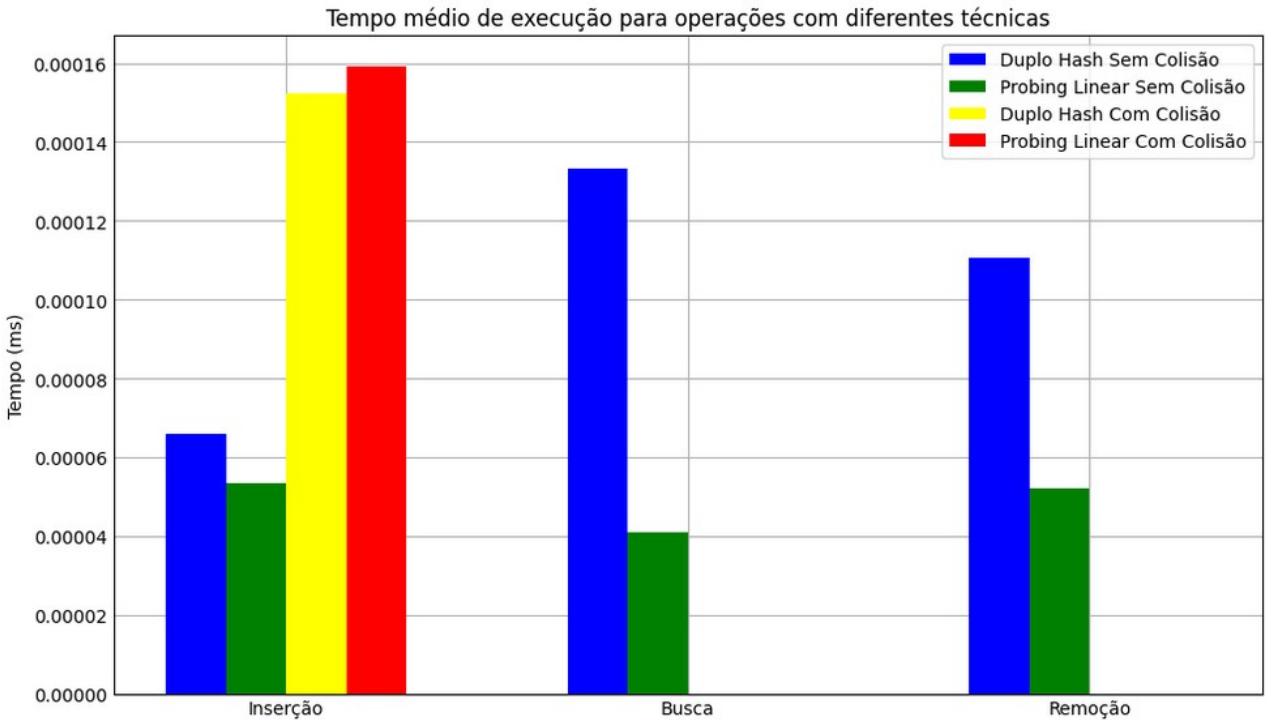
```python
def create_contact_list(size):
    # Inicializa uma lista de tamanho 'size' com None
    contact_list = [None] * size
    return contact_list
def is_full(contact_list):
    # Verifica se a lista de contatos está cheia
    return all(contact is not None and contact != "delete" for contact in contact_list)
def hash_function(email, size):
    # Função hash simples que retorna o índice com base no e-mail
    return hash(email) % size
def hash_function2(email, size):
    # Segunda função hash para o hashing duplo
    return (hash(email) % (size - 2)) + 1  # Garante que o valor não seja zero
def insert_contact(contact_list, email, name, phone):
    size = len(contact_list)
    if is_full(contact_list):
        print("A lista de contatos está cheia. Não é possível inserir mais contatos.")
        return
    index = hash_function(email, size)
    step = hash_function2(email, size)
    while contact_list[index] is not None and contact_list[index] != "delete":
        # Hashing duplo
        index = (index + step) % size
        print("colisão")
    contact_list[index] = {"email": email, "name": name, "phone": phone}
def search_contact(contact_list, email):
    size = len(contact_list)
    index = hash_function(email, size)
    step = hash_function2(email, size)

    while contact_list[index] is not None:
        contact = contact_list[index]
        if contact == "delete":
            # Ignora posições marcadas como "delete"
            index = (index + step) % size
            continue
        if contact["email"] == email:
            return contact
        index = (index + step) % size

    # Valor não encontrado
    print(f"Contato com e-mail '{email}' não encontrado.")
    return None
def delete_contact(contact_list, email):
    size = len(contact_list)
    index = hash_function(email, size)
    step = hash_function2(email, size)

    while contact_list[index] is not None:
        contact = contact_list[index]
        if contact == "delete":
            # Ignora posições marcadas como "delete"
            index = (index + step) % size
            continue
        if contact["email"] == email:
            # Marcando o contato como excluído
            contact_list[index] = "delete"
            return
        index = (index + step) % size
```

Resultados:



(Analisando numericamente os resultados):

Ao testar os algoritmos, percebi que a quantidade de colisões no **duplo hash** foi menor do que no **Probing**, no entanto notei, e o gráfico mostra isso, que as funções de remoção e busca no duplo hash ficaram mais lentas, eu acredito que o fato de ter que calular duas vezes o hash possa ter alguma influência. Outro ponto é que existe um agrupamento de elementos **no probing linear** que pode, em alguns casos, ser benéfico especialmente em cenários onde a tabela de hash é percorrida em sequência em um pipeline de processamento. No entanto, essa vantagem geralmente é específica para certos tipos de cargas de trabalho e não se aplica a todos os casos.
A vantagem do agrupameto no **probing linear** ocorre por que, quando os elementos estão agrupados na tabela, é mais provável que eles estejam próximos na memória, o que pode levar a um melhor aprovietamento do cache e a tempos de acesso mais rápidos em máquinas modernas. No entanto, isso depende muito da arquitetura da máquina, do tamano da tabela de hash e do padrão de acesso dos dados.
Pórem, o duplo hash busca espalhar os elementos colididos pela tabela de hash, o que pode reduzir o agrupamento, mas também pode resultar em uma distribuição menos previsível dos elementos de acesso aos dados.