Estruturas de Dados II

Tabela Hash

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Tabela de Acesso Direto

- Em ED1 e ED2 vimos diversas estruturas para armazenar e gerenciar dados.
- Mas existe uma estrutura simples, de implementação descomplicada, e muito eficiente que ainda não discutimos em detalhe: **Tabela de Acesso Direto**.
- Tabela de Acesso Direto são eficientes na inserção, deleção e recuperação de dados.
- Especificamente, efetuamos essas três operações em tempo O(1) (tempo constante).
- Até o momento, vocês não viram nenhuma estrutura de dados tão eficiente assim.

Tabela de Acesso Direto

- Considere que tenhamos um universo de chaves $\mathcal{K}$, de tamanho $|\mathcal{K}|$.
- Ou seja, considerando uma indexação que inicia em zero, $\mathcal{K} = 0, 1, \ldots, |\mathcal{K}| - 1$.
- Para representar esse conjunto, podemos utilizar uma Tabela de Acesso Direto, tipicamente implementada usando um vetor (array) de elementos.

Tabela de Acesso Direto

- Cada chave no universo $\mathcal{K}$ corresponde a um índice na tabela.
- O conjunto de chaves utilizadas $\mathcal{S}$ determina as posições que contêm ponteiros para os elementos.

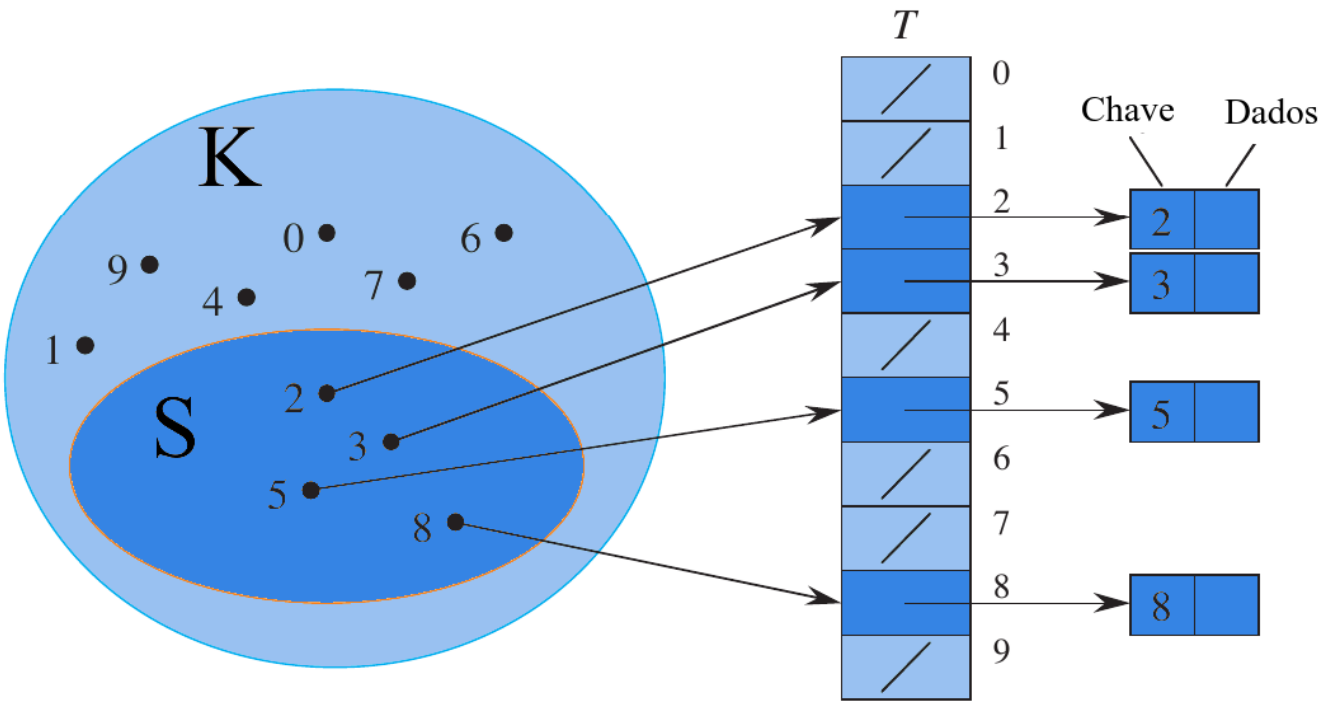


Tabela de Acesso Direto

- A inserção, exclusão e recuperação de dados ocorre em tempo O(1), pois essas operações dependem apenas do acesso direto ao índice no vetor, determinado pelo valor da chave associada ao elemento.
- Essa abordagem utiliza endereçamento direto, onde a chave de um elemento determina diretamente sua posição na tabela. Por exemplo, se a chave do elemento for 256, o elemento será armazenado na posição de índice 256 do vetor.
- Assim, o endereço do elemento na estrutura de dados é igual ao valor da própria chave, eliminando a necessidade de cálculos ou buscas adicionais.

Tabela de Acesso Direto

- Entretanto, caso o universo de chaves $\mathcal{K}$ seja suficientemente grande, essa abordagem se torna inviável.
- Para utilizar essa estrutura, seria necessário alocar memória para o tamanho total do universo de chaves, o que implica reservar espaço para todas as chaves possíveis, independentemente de sua utilização.
- Além disso, seu uso pode resultar em desperdício significativo de memória, caso o conjunto de chaves utilizadas $\mathcal{S}$ for pequeno comparado ao tamanho de $\mathcal{K}$.
- Portanto, seu uso é eficaz quando $\mathcal{K}$ é relativamente pequeno.

Vetor de Bits

- Uma alternativa mais eficiente em termos de espaço, mas sem capacidade de armazenamento de dados, permitindo apenas a identificação da existência do elemento, é o Vetor de Bits.
- O Vetor de Bits é uma estrutura de dados usada para representar um conjunto de valores binários de forma compacta.
- Cada bit em um vetor pode representar a presença ou ausência de um item em um conjunto, ou pode ser utilizado para armazenar qualquer informação que se beneficie da representação binária.

Vetor de Bits

- O Vetor de Bits é um vetor onde cada posição armazena um único bit (0 ou 1).
- O Vetor de Bits pode ser usado para representar um conjunto de números, onde o índice de cada bit corresponde a um elemento do universo. Se um elemento está no conjunto, o bit correspondente é 1; caso contrário, é 0.
- O vetor pode ser interpretado de diversas formas[1], por exemplo, zero (0) pode significar "ausente" e um (1) pode significar "presente".
- Exemplificando, para representar o conjunto $\mathcal{S} = \{2, 5, 7\}$ no universo $\mathcal{K} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, o Vetor de Bits seria:

$$\text{Vetor de Bits} = 0010010100$$

- O valor 1 nas posições 2, 5 e 7 indica que esses elementos estão presentes no conjunto.

[1] Ativo/Inativo, Prioridade Alta/Baixa, Ocupado/Livre, Sucesso/Erro, Ligado/Desligado, etc.

Vetor de Bits

- Vantagens:

⇒ Eficiência em termos de espaço: cada bit ocupa apenas 1 bit de memória.

⇒ Operações rápidas: As operações bit a bit são realizadas de forma rápida em hardware.

- Desvantagens:

⇒ Tamanho definido por K: assim como no caso das Tabela de Acesso Direto, seu uso pode resultar em desperdício significativo de memória.

⇒ Não é possível armazenar dados associados a chave no Vetor de Bits.

Tabela de Acesso Direto e Vetor de Bits

- Portanto, a Tabela de Acesso Direto se torna inviável quando o universo $\mathcal{K}$ é muito grande.
- Também pode resultar em um desperdício significativo de memória em cenários onde o universo $\mathcal{K}$ é grande e o conjunto de chaves ativas $\mathcal{S}$ é relativamente pequeno.
- Uma alternativa mais compacta é o Vetor de Bits, que, embora eficiente para identificar a presença ou ausência de elementos, não permite o armazenamento de dados associados.
- Surge, então, a necessidade de encontrarmos uma solução que combine eficiência espacial com a capacidade de associar dados aos elementos.

Tabela Hash

- Uma **Tabela Hash** é uma estrutura de dados que usa uma **função hash** para mapear uma chave a uma posição específica (índice) dentro de um vetor.
- Com o endereçamento direto, um elemento com chave c é armazenado na posição c, mas com a Tabela Hash, usamos uma função hash h para calcular a posição a partir da chave c, de nodo que o elemento vai para a posição $h(c)$.
- Ou seja, a função hash h mapeia o universo $\mathcal{K}$ de chaves para as posições de uma tabela hash $\mathcal{T}$, ou seja:

$$h : K \rightarrow \{0, 1, \ldots, m-1\}$$

- Portanto, o tamanho da Tabela Hash é tipicamente muito menor que $|\mathcal{K}|$

Tabela Hash

- Dizemos que um elemento com chave k é mapeado para a posição $h(k)$, e dizemos que $h(k)$ é o valor hash da chave k.
- A função hash reduz o intervalo dos índices do vetor e, consequentemente, o tamanho do vetor.
- Em vez de um tamanho de $|\mathcal{K}|$, o vetor pode ter tamanho m, onde m é o tamanho definido para a Tabela Hash.
- Ou seja, o requerimento de espaço da Tabela Hash é $O(|\mathcal{S}|)$.

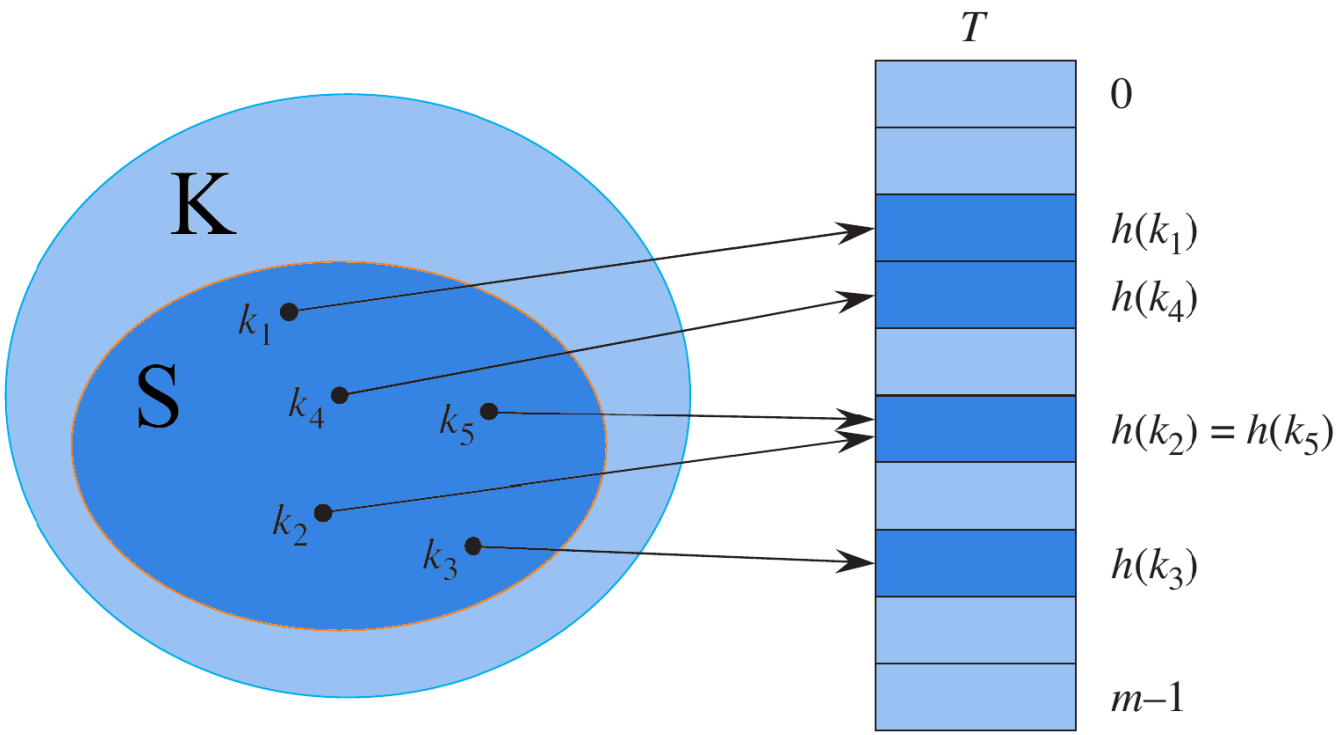


Tabela Hash

- Há um problema que talvez já tenham percebido. Como $|\mathcal{K}| > m$, podem haver pelo menos duas chaves com o mesmo valor hash, ou seja, que serão colocadas na mesma posição da Tabela Hash.
- Chamamos essa situação de **colisão**, onde duas ou mais chaves podem ser mapeadas para a mesma posição.
- A boa notícia é que existem técnicas eficazes para resolver o conflito criado por colisões.

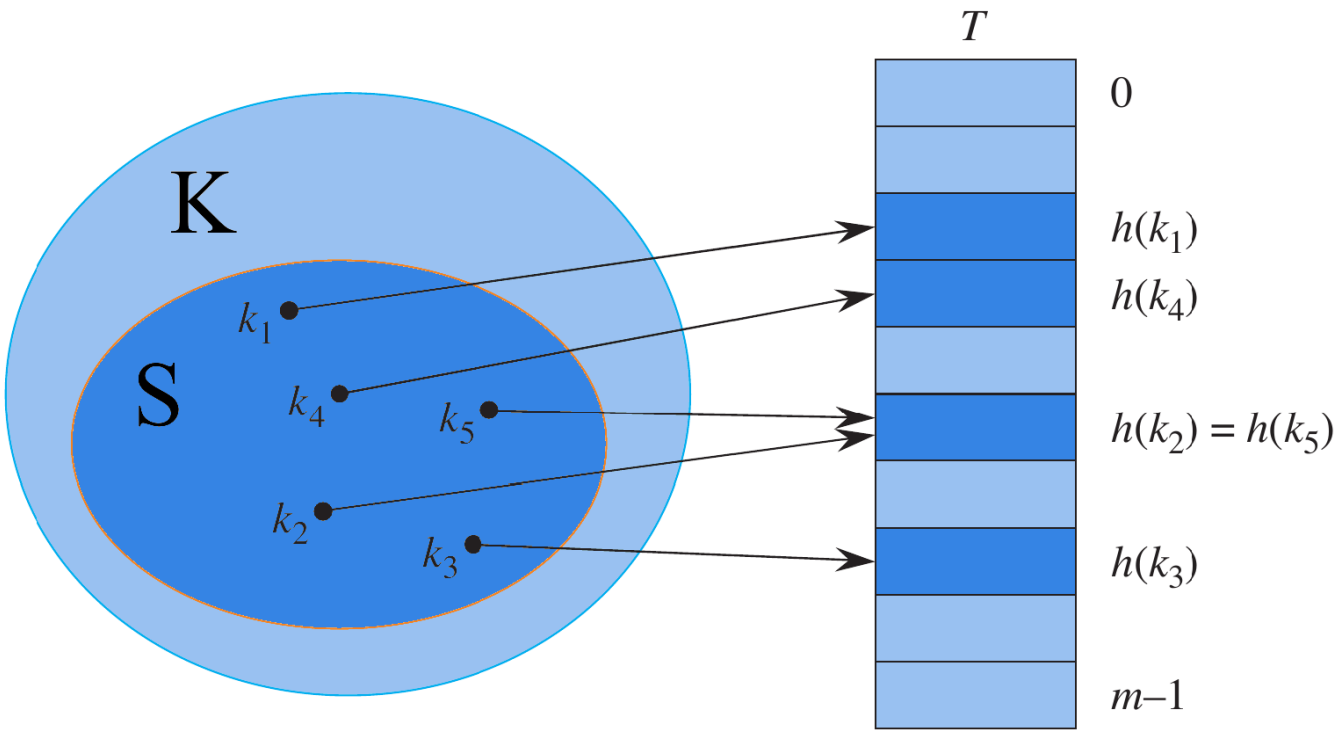
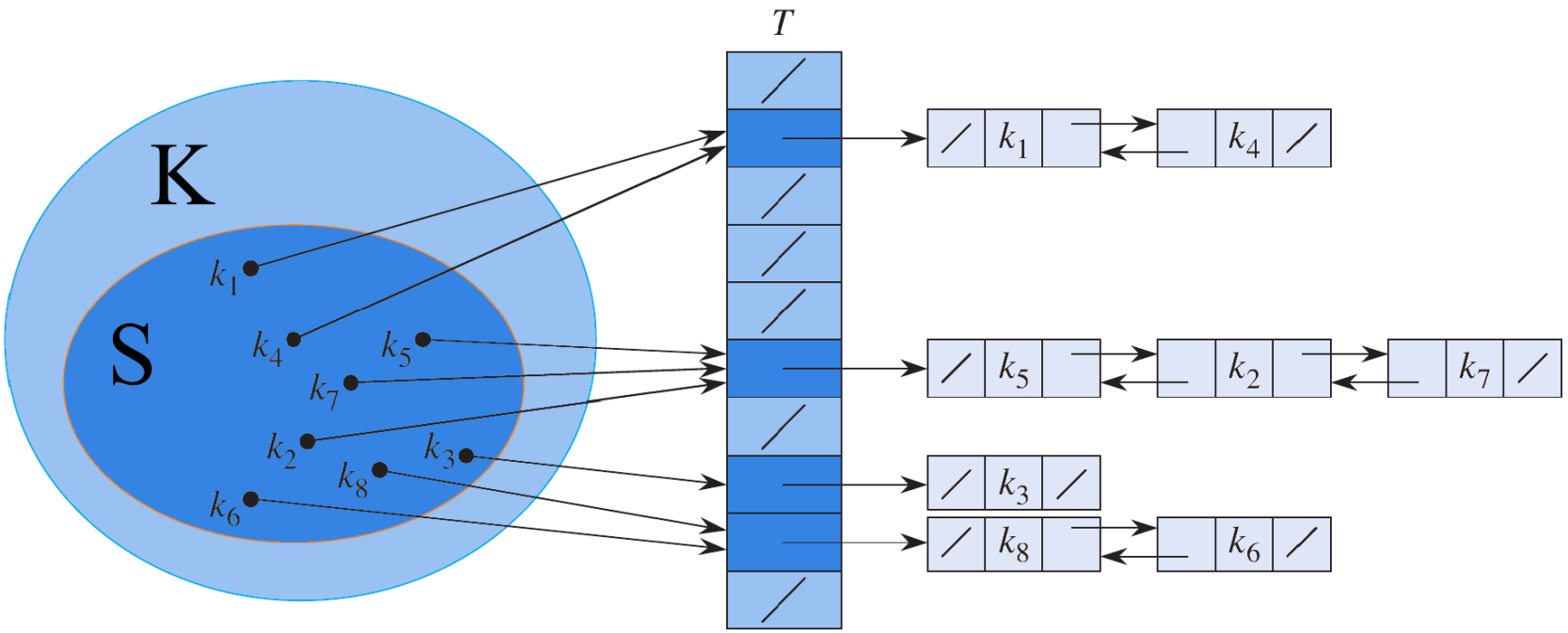


Tabela Hash

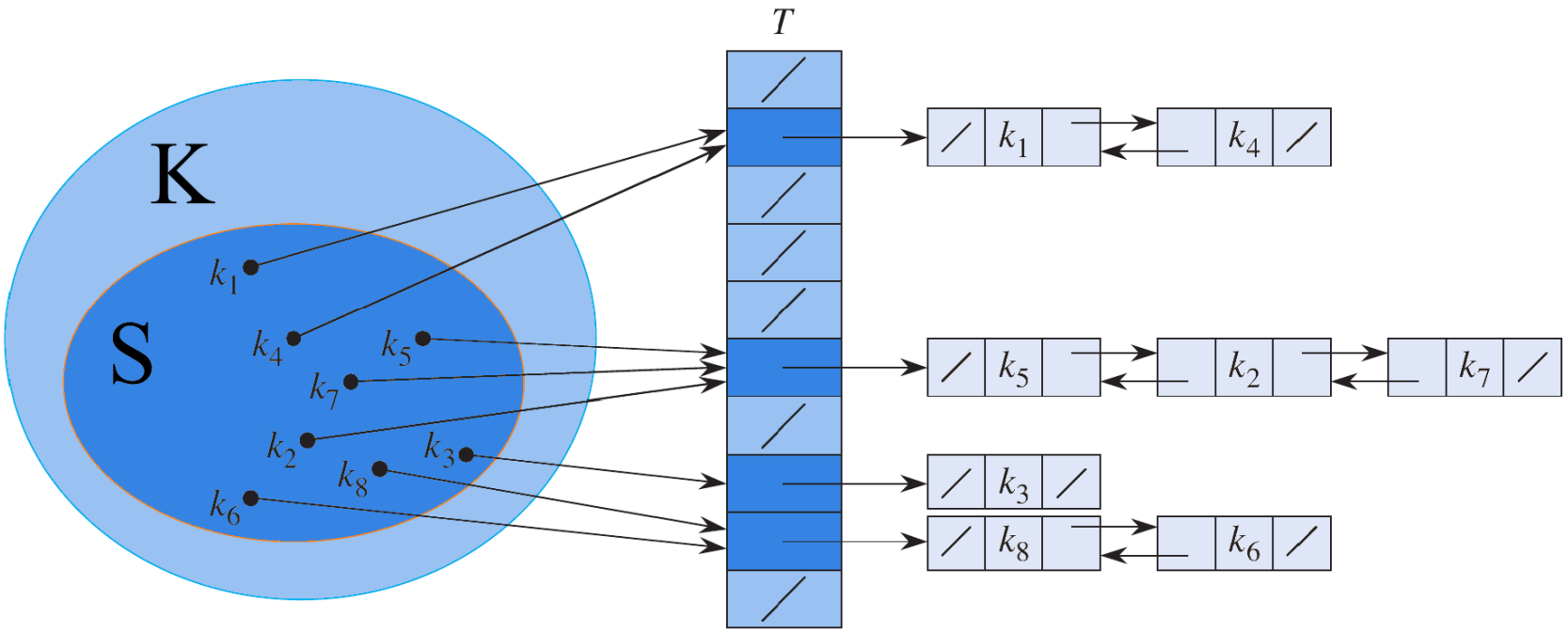
- Idealmente, buscamos evitar colisões, ou pelo menos, minimizá-las.
- Para este fim, devemos escolher uma função hash h adequada.
- É evidente que uma função hash h deve ser determinística, ou seja, uma entrada k dada deve sempre produzir a mesma saída $h(k)$.
- Mas colisões poderão ocorrer, portanto, precisaremos de alguma abordagem para solucionar este problema.

Encadeamento

- A técnica mais simples para resolução de colisões é chamada de **encadeamento**.
- Cada posição não vazia é um ponteiro para uma lista ligada. Todos os elementos que de mesmo valor hash são adicionados na lista encadeada associada à posição.
- Ou seja, em uma posição j da Tabela Hash, calculada pela função hash para uma chave c, temos um ponteiro para o início de uma lista ligada contendo todos os elementos os quais o valor hash resultou em j.

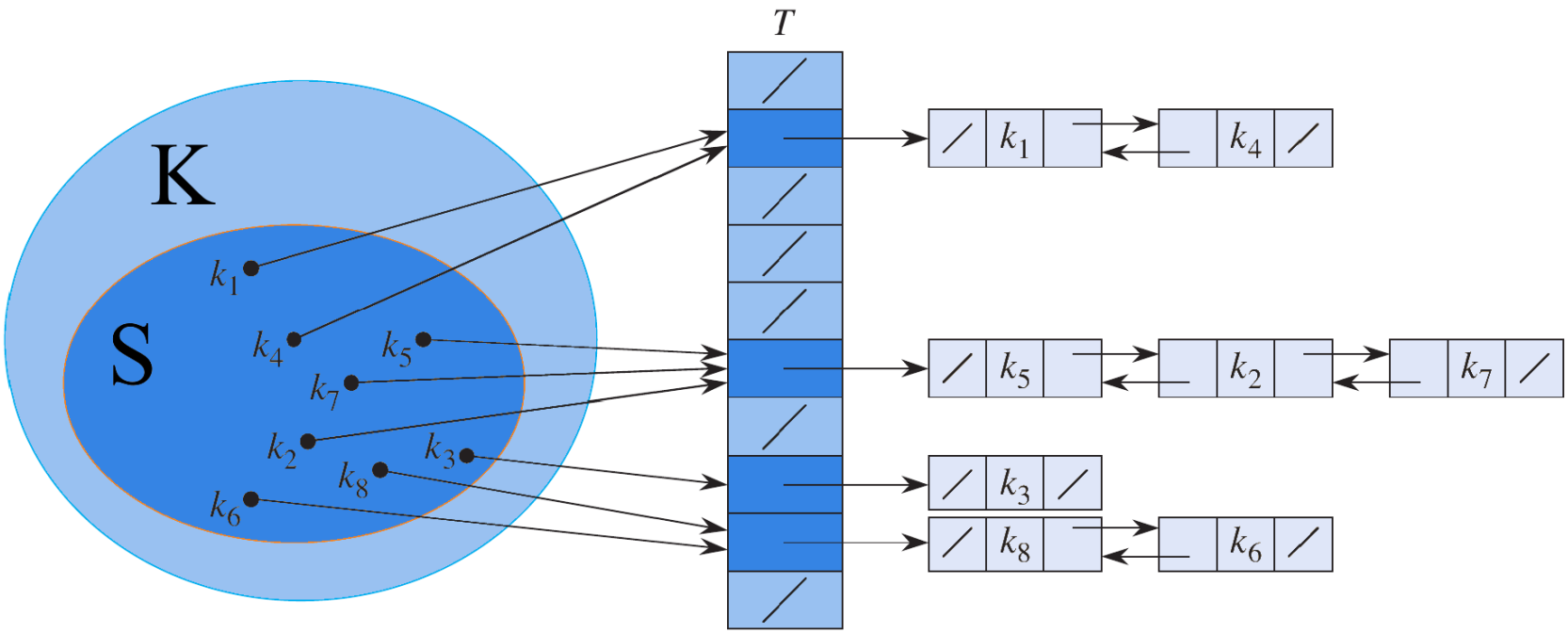


Encadeamento – Operações Básicas



- Quais os custos para as operações básicas de inserção, deleção e busca em uma Tabela Hash com encadeamento?

Encadeamento – Pior Caso



- A inserção pode ser feita de modo rápido, utilizando a inserção em lista ligada não-ordenada, portanto, o tempo será O(1) no pior caso.
- A deleção, utilizando uma lista duplamente ligada, executará em O(1). Entretanto, ela envolve encontrar o item, portanto vamos analisar a operação de busca.
- No pior caso, todos os elementos são mapeados para a mesma posição e criamos uma lista ligada de tamanho $|\mathcal{S}|$, ou seja, se $|\mathcal{S}| = n$, o tempo da operação de busca será O(n).

Encadeamento – Caso Médio

- Entretanto, na prática a Tabela Hash tende a ser eficiente. Para fazermos este argumento, precisamos considerar o caso médio.
- O desempenho no caso médio depende de quão bem a função de hash distribui o conjunto de chaves a ser armazenado entre as m posições alocadas, na média.
- A análise que fundamenta esse argumento está além do escopo do curso e disciplina, mas será apresentada de forma introdutória para que vocês possam ter uma noção geral do tema.

Encadeamento – Caso Médio

- O desempenho no caso médio depende de quão bem a função de hash h distribui o conjunto de chaves a serem armazenadas entre as m posições, na média.
- Assumimos que qualquer elemento tem a mesma probabilidade de ser mapeado para qualquer uma das m posições. Ou seja, que a função de hash é **uniforme**.
- Além disso, supomos que o mapeamento de um elemento para uma posição é **independente** do mapeamento de outros elementos.

Encadeamento – Caso Médio

- No **caso médio**, a busca de um elemento é realizada em $O(1 + \alpha)$, onde onde $\alpha = \frac{n}{m}$, $n = |S|$ e m é o número de posições disponíveis.
- Isso significa que o fator de carga α tem uma influência direta no tempo de execução, pois ele reflete a densidade de elementos na tabela.
- Quanto maior o α, maior o impacto das colisões, já que mais elementos compartilham as mesmas posições, aumentando o custo da busca no caso médio.

Encadeamento – Caso Médio

- O caso médio assume um cenário típico de uso, onde o sistema não é deliberadamente sobrecarregado e n é controlado em relação a m.
- Embora possam existir casos extremos (como n $\gg$ m), eles são exceções e não o comportamento padrão.
- Deste modo, podemos assumir que o número de elementos na tabela é no máximo proporcional ao número de posições da Tabela Hash. Ou seja, $n = O(m)$.
- Isso significa que existe uma constante $c > 0$ tal que $n \leq c \cdot m$ para todo n suficientemente grande.
- Ou seja, no máximo, n cresce linearmente em relação a m.

Encadeamento – Caso Médio

- Deste modo, se $n = O(m)$, como $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$.
- O tempo médio de busca em uma tabela hash é $O(1 + \alpha)$, portanto, com $\alpha = O(1)$, temos que $O(1 + O(1)) = O(1)$.
- Ou seja, a busca é realizada em tempo constante, no caso médio (!!).
- Como a inserção leva tempo $O(1)$ no pior caso e a exclusão também leva tempo $O(1)$ no pior caso, resulta que todas as operações basicas executam em tempo $O(1)$, no caso médio.

Endereçamento Aberto

- O **Endereçamento Aberto** é um método de resolução de colisões que não utiliza armazenamento externo a Tabela Hash. Todos os elementos ocupam a própria Tabela Hash.
- Cada entrada na tabela contém um elemento do conjunto ou estará vazio (nulo).
- As colisões são tratadas do seguinte modo: quando um novo elemento precisa ser inserido na tabela, ele é colocado em sua primeira opção de localização, se possível.
- Se essa posição já estiver ocupada, o novo elemento será colocado em sua segunda opção.
- Esse processo continua até que uma posição vazio seja encontrada para armazenar o novo elemento.
- Diferentes elementos têm ordens de preferência distintas para as posições.

Endereçamento Aberto – Inserção

- Para realizar a inserção usando endereçamento aberto, "sonda-se" a Tabela Hash até encontrar uma posição vazia onde o elemento possa ser colocado.
- Em vez de seguir uma ordem fixa como $0, 1, \ldots, m-1$, a sequência de posições sondadas depende da chave a ser inserida.
- Para determinar quais posições sondar, a função de hash inclui o número da sondagem como uma segunda entrada. Assim, a função hash torna-se:

$$h : K \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

.

- O Endereçamento Aberto exige que, para cada chave k, a sequência de sondagem $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ seja uma permutação de $\{0, 1, \ldots, m-1\}$, garantindo que toda posição da tabela hash seja eventualmente considerada como uma posição para um novo elemento à medida que a tabela é preenchida.

Endereçamento Aberto – Busca

- O algoritmo de busca pelo elemento de chave k sonda a mesma sequência de posições que o algoritmo de inserção examinou quando a chave k foi inserida.
- Portanto, a busca pode terminar ao encontrar uma posição vazia (não encontrou o elemento), já que k teria sido inserida naquele ponto e não em uma posição posterior na sua sequência de sondagem.

Endereçamento Aberto – Deleção

- A deleção em uma Tabela Hash com Endereçamento Aberto possui um complicador.
- Quando você deleta um elemento da posição p, não deverá marcar essa posição como vazia armazenando nulo nela.
- Se isso fosse feito, poderia se tornar impossível recuperar qualquer chave k para a qual a posição p foi sondada e encontrada ocupada no momento em que k foi inserida.
- Uma forma de resolver esse problema é marcando a posição com um valor especial, por exemplo, armazenando nele o valor **deletado** em vez de nulo.

Endereçamento Aberto – Desvantagem

- Uma desvantagem evidente do endereçamento aberto é que a Tabela Hash pode ficar completamente preenchida, impossibilitando novas inserções.
- Por outro lado, na solução com encadeamento, isso não ocorre, pois as listas ligadas em cada posição podem crescer indefinidamente, dependendo apenas da memória disponível.

Endereçamento Aberto – Complexidade

- Inserção: O(n), caso a tabela esteja quase cheia e seja necessário percorrer todos os espaços até encontrar o único vazio.
- Deleção: O(n), mesma situação anterior, onde a sondagem verificará quase todas as posições.
- Busca: O(n), novamente, todas posições podem podem precisar ser examinadas devido a colisões antes do final da sequência de sondagem.

Endereçamento Aberto – Complexidade

- Assim como no caso do encadeamento, a análise do caso médio é a mais interessante, embora seja mais complexa.
- No caso médio, a inserção tem uma complexidade de $O\left(\frac{1}{1-\alpha}\right)$, onde $\alpha = \frac{n}{m}$.
- Para as operações de deleção e busca, o caso médio tem complexidade de $O\left(\frac{1}{\alpha} \ln \frac{1}{1-\alpha}\right)$.

Endereçamento Aberto – Complexidade

- Ou seja, para valores razoáveis de α, o tempo pode ser praticamente constante ($O(1)$), mas, à medida que α se aproxima de 1, o desempenho pode cair para o pior caso de $O(n)$, especialmente em cenários com colisões frequentes.
- Por exemplo, na busca, se $\alpha = 0,5$ (tabela 50% cheia), o número esperado de sondagens em uma busca bem-sucedida é $< 1,387$. Se $\alpha = 0,9$, o número esperado de sondagens é $< 2,559$.
- Mas se $\alpha = 1$, ou seja, uma busca malsucedida, **todas** as m posições serão sondadas.

Exercícios

1. Considere uma Tabela Hash com 9 posições e a função hash $h(k) = k \mod 9$. Demonstre o que acontece ao inserir as chaves $5, 28, 19, 15, 20, 33, 12, 17, 10$, com as colisões resolvidas por encadeamento (Cormen 11.2-2).
2. O Professor Marley hipotetiza que pode obter ganhos substanciais de desempenho ao modificar o esquema de encadeamento para manter cada lista em ordem crescente. Como a modificação do professor afeta o tempo de execução para buscas bem-sucedidas, buscas malsucedidas, inserções e deleções? (Cormen 11.2-3).
3. Insira as chaves 10, 22, 31, 4, 15, 28, 17, 88, 59 em uma Tabela Hash de tamanho m $= 11$ utilizando endereçamento aberto. Ilustre o resultado da inserção dessas chaves usando sondagem linear com $h(k, i) = (k + i) \mod m$ e usando dupla sondagem com $h_1(k) = k$ e $h_2(k) = 1 + (k \mod (m - 1))$ (Cormen 11.4-1).