Estruturas de Dados

Tabelas Hash

1. Tabelas hash (Cormen, capítulo 11. Seções 11.1, 11.2, 11.3, 11.4)

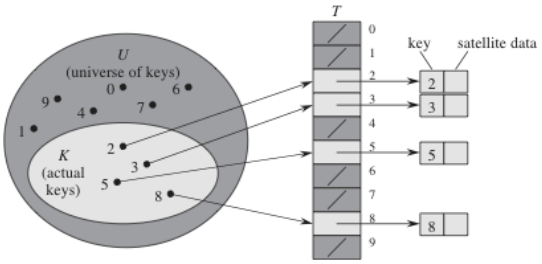
Tabelas hash: Estruturas úteis para aplicações onde são necessárias somente operações de inserção, remoção e pesquisa, e onde a pesquisa precisa ser rápida. Na prática, tabelas hash conseguem realizar pesquisas em tempo médio O(1). Exemplo de aplicação: dicionários, tabelas de símbolos.

Em uma tabela hash, elementos são inseridos na tabela com base em sua chave. A forma mais simples de se implementar uma tabela hash, portanto, é através de **tabelas de endereçamento direto**.

Tabelas de endereçamento direto: Essa técnica é bastante simples e funciona bem quando o universo de chaves é razoavelmente pequeno. Assumindo que cada elemento a ser armazenado na tabela tenha uma chave diferente, cada chave corresponderá a uma posição distinta da tabela. É feita uma relação direta entre chave do elemento e posição da tabela. As operações de inserção, remoção e busca têm complexidade O(1).

tabela = T; chave = k; elemento = x

search: return T[k]
insert: T[chave[x]] = x
delete: T[chave[x]] = NIL

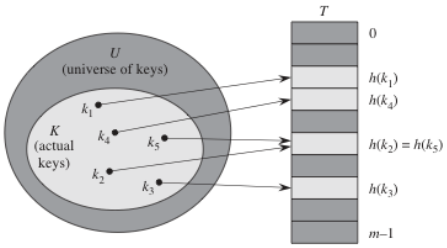


O que fazer se o universo *U* de chaves for muito grande? Armazenar uma tabela de tamanho |*U*| pode ser impraticável. O número de chaves de fato armazenadas em um dado momento pode ser muito menor que |*U*|, de forma que a maioria do espaço alocado para a tabela estaria desperdiçado.

Tabelas hash: tabelas hash usam uma tabela de tamanho menor que o universo de chaves para realizar o armazenamento. O tempo de busca ainda será, em média, O(1) (no pior caso, no entanto, a busca não é mais O(1)). Tabelas hash usam o conceito de funções hash (*h*) para para calcular a posição de um elemento na tabela. Ao invés de usar *k*, usa-se *h(k)*. *h* mapeia o universo *U* de chaves nas posições da tabela.

h(k) -> valor hash da chave k

Problema fundamental: vão existir **colisões**. Uma boa função hash (bem projetada e de aparência aleatória) ajuda a evitar colisões. No entanto, é impossível que elas não existam. Como tratá-las?
Colisão: acontece quando a o valor hash de duas chaves distintas é o mesmo. Exemplo: função mod10 como hash. 10, 20, 30, etc. possuem o mesmo valor hash.



Resolução de colisões por encadeamento: todos os elementos que entrarem em colisão para a mesma posição da tabela são colocados em uma lista encadeada.

search: procura elemento na lista T[h(chave[x])] => O(1 + α)
insert: T[chave[x]] = insere elemento no início da lista T[h(chave[x])] => O(1)
delete: T[chave[x]] = remove elemento da lista T[h(chave[x])] => O(1)

Fator de carga: número médio de elementos armazenados em uma cadeia. O fator de carga da tabela hash é definido por α = n/m, onde m é o tamanho da tabela hash e n o número de elementos que ela armazena.

Suposição do hash uniforme simples: qualquer elemento dado tem igual probabilidade de efetuar o hash para qualquer uma das *m* posições da tabela, não importando para onde foi feito o hash de qualquer outra chave. Isso dependerá, obviamente, da função hash usada.

Se o número de posições da tabela hash for proporcional ao número de elemento da tabela, temos que: n = O(m);
α = n/m = O(m)/m = O(1).

Resolução de colisões por endereçamento aberto: nesse modo de resolução de colisões não usamos estruturas auxiliares. Todos os elementos são armazenados na própris tabela hash. A tabela pode ficar cheia, e o fator de carga nunca é maior que 1. No endereçamento aberto **sondamos** a tabela até encontrar uma posição vazia para inserir certa chave. A sequencia de posições sondadas depende da chave inserida.

Sondagem linear: forma mais simples de endereçamento aberto. A função hash usada é $h(k, i) = (h'(k) + i) \bmod m$.
Problema: **agrupamento primário**. Longas sequencias são construídas, piorando a busca.

Sondagem quadrática: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Vantagem do endereçamento aberto: não usamos ponteiros. As posições são calculadas e, portanto, temos mais memória livre. A tabela hash pode ser maior para a mesma memória ocupada.
Desvantagem do encadeamento aberto: mais difícil de implementar. Deletar chaves é difícil.

O que faz uma boa função hash?
Uma boa função hash satisfaz (aproximadamente) a suposição do hash uniforme simples. Em geral, não é possível verificar essa condição mas, na prática, é possível usar heurísticas para criar uma função hash que provavelmente terá um bom desempenho. A função hash deve ser independente de qualquer padrão que possa existir nas chaves. Além disso, uma boa função hash deve executar rapidamente, uma vez que ela será usada com frequencia.