Boa tarde, vou apresentar agora o trabalho de estrutura de dados sobre tabela de espalhamento, ou, como vou me referir a partir de agora, tabela Hash. Resolvi separar a apresentação em “três partes”, a primeira será uma introdução, veremos o que é a tabela hash e alguns exemplos, já a segunda parte será sobre o tratamento de colisões, entenderemos na primeira parte o que é uma colisão, o tratamento de colisões não será simplesmente pra evitar colisões, mas, quando acontecer uma colisão, o que se pode fazer. A terceira parte será sobre a implementação da tabela hash com tratamento de colisões, através de lista encadeada.  
  
Então, vamos ver aqui um problema que justificaria o uso de uma tabela hash. Ler. Digamos que eu queira fazer uma busca nesse banco de dados, então o RA estaria vinculado a um aluno. Imagine, esse banco de dados pode ter milhares de alunos, precisaríamos de uma busca eficiente, senão seria algo inviável. Outro exemplo seria fazer login em algum site, ele teria que ir num banco de dados e checar se a senha que eu coloquei para aquele respectivo e-mail está certa.  
  
Eu posso fazer essa busca de algumas formas, como a busca sequencial e a busca binária.  
Na busca sequencial como a gente viu ontem, eu começo do primeiro elemento e vou fazendo a checagem. É o primeiro? Não. Vou pro próximo. É ele? Não. A busca só iria parar caso encontrasse o valor procurado ou chegasse no final e não encontrasse. É claro que essa seria uma das piores buscas, já que teria um custo computacional grande e demorado. Se o valor estivesse na última posição, por exemplo, eu precisaria percorrer todos os elementos até achar o valor, é algo que não é muito eficiente.  
Na busca binária a gente pressupõe que o vetor está ordenado de alguma forma. Ou a gente criaria um vetor ordenado ou teria que ordenar o vetor de alguma maneira. Nela a gente vai no meio do vetor e faz a comparação: É menor, maior ou igual ao valor procurado? Se for maior a gente pega essa segunda metade e continua fazendo comparações, de meio em meio, até achar o valor ou sobrar apenas um e não ser ele. A mesma coisa pra caso seja menor, a gente faria essas comparações até terminar.  
Então, qual seria uma outra forma de resolver o meu problema, de modo eficiente, sem utilizar uma busca sequencial ou binária? Sim, a gente vai utilizar a tabela hash.  
  
Primeiramente, vamos entender: O que é uma tabela hash? Ler.  
A gente vai implementar a tabela hash num vetor e teremos o par: chave; valor.  
O valor vai ser um número inteiro que é o índice do vetor. De zero até n-1.  
A chave é o que a gente vai colocar dentro do vetor e também vamos usar ela pra fazer a busca. A chave pode ser int, float, string, não importa muito, pode ser de qualquer tipo. A gente vai transformar essa chave num valor inteiro, que vai ser o índice do vetor.  
  
Pra transformar a chave num valor inteiro a gente precisa uma função, chamada função hash.  
Um exemplo de função hash seria: Digamos que eu vá pegar o nome do aluno como chave, minha função é a seguinte: Quantidade de letras do nome do aluno – 1. O mínimo de letras pra um nome, digamos que seria 1, então, pra ocupar a posição 0 eu preciso fazer –1.  
  
Como exemplo, não pergunte de onde eu tirei esse nome (risos) vamos adicionar o nome “Nyasm”. Quantas letras tem? 5. Passando pela função hash faria: 5 – 1 = 4. Então, eu vou adicionar o nome “Nyasm” no índice 4. A gente ligou a string “Nyasm” com o valor 4 através da função hash.  
Vamos usar outro exemplo: Melk. Melk tem 4 letras, utilizando a função hash a gente teria o valor 3, então iria ligar a string Melk com o índice 3 do nosso vetor.  
  
Agora vamos fazer uma operação de busca, eu quero buscar o nome “Melk” no vetor. Através da função hash eu já teria o índice que “Melk” estaria inserido, então vou buscar diretamente lá. Encontrei Melk? Sim. Então ele está no vetor. Buscando Nyasm seria o mesmo processo, a função hash já me daria a posição que a minha chave Nyasm poderia estar no vetor, procurei lá e encontrei, ótimo. Agora vou buscar Beca, Beca tem 3 letras, através da função hash eu teria o índice 3, procurei no índice três mas não achei Beca, achei Melk, então ele pressupõe que a Beca não tá lá. Então vamos adicionar a Beca, só que, como a gente já viu, Melk tá na posição que eu adicionaria a Beca, então, houve uma colisão, que é o que atrapalha uma tabela hash. Na parte 2 eu vou explicar como tratar colisões.  
  
Cada estrutura de dados tem um objetivo, o da tabela hash é fazer buscas de forma eficiente. A fila, por exemplo... Digamos que eu vou implementar uma impressora, pra isso eu uso uma fila, a pessoa submete um arquivo e ele entra na fila, a primeira impressão que chegar vai ser a primeira feita, a última a chegar vai ser a última feita.  
  
Só que eu posso ter uma função hash ruim, que gera muitas colisões, então eu não vou ter uma busca eficiente. Pra uma tabela hash ser boa é preciso uma função hash boa, ela precisa produzir um número ínfimo de colisões; ser facilmente computável, não posso ficar uma eternidade pra fazer uma operação; e ser uniforme, fazer as operações num tempo constante.  
  
Vamos ver outro exemplo de tabela hash, com aquele mesmo vetor de antes, com tamanho 5. Eu também tenho o fator de carga, que é o cálculo da quantidade de elementos que eu vou colocar no vetor dividido pelo tamanho do vetor. Nos exemplos que eu vou mostrar não vou levar em conta o fator de carga, já que o vetor tem poucas posições, é só pra exemplificação mesmo. Pro fator de carga a gente também precisar ter o tamanho do vetor maior que o número máximo de elementos. Eu não posso, por exemplo, ter esse vetor com tamanho 5 e colocar seis elementos nele. Quanto mais próximo de 1, significa que mais perto a quantidade de elementos está do tamanho do vetor, ou seja, existem poucas posições vazias, o que é ruim pro tratamento de colisões, que vou falar depois. Quanto mais perto de 0, mais posições vazias eu vou ter após ter colocado todos os elementos, o que também é ruim, já que vou ter muita memória sem uso. Então, é preciso ter um equilíbrio aí.  
  
A função hash que vou usar pra esse exemplo é: chave, que vai ser um número inteiro positivo, módulo tamanho do vetor. O primeiro elemento que vou adicionar vai ser o 9, então faço 9 módulo 5, que é o tamanho de vetor, o que vai resultar 4, então vou inserir a chave 9 no índice 4. Quando eu quiser buscar o 9 é só eu fazer a mesma coisa, passa pela função hash, vou pegar a posição 4 e verificar se ele está lá, pensando que não houve nenhuma colisão ao inserir os elementos.  
Continua dando os exemplos até a colisão.  
  
Onde eu colocaria o 20? Existem duas opções. Ou eu coloco do lado do 15, usando uma lista encadeada, ou eu coloco no próximo espaço livre. Pensando em colisões a forma de inserir e buscar muda, dependendo de como eu faça.  
  
  
  
Bom, agora vamos ver o que fazer pra tratar as colisões numa tabela hash. Ler. Que são as duas formas principais.  
Temos o encadeamento exterior, que a gente usa listas encadeadas. Não vamos colocar as chaves dentro do vetor e sim de forma externa, em listas encadeadas, por isso o nome “encadeamento exterior”. Cada posição do vetor vai ter um ponteiro pra uma lista encadeada, que vai armazenar as colisões daquele espaço.  
E também temos o encadeamento interior, que significa que as chaves vão estar dentro do vetor.  
Existem dois tipos de encadeamento interior, o heterogêneo e o homogêneo. Jajá eu vou explicar melhor os dois. No homogêneo existem vários testes, como o teste quadrático e tals, mas eu vou utilizar o mais comum, que é o teste linear.  
  
Começando com o encadeamento exterior, vou pegar aquele mesmo vetor que a gente tinha anteriormente, e vou adicionar novamente o valor 15, que vai parar na posição 0 do vetor. Só que o 15 não vai ficar aqui dentro do vetor, ele vai ficar numa lista encadeada. A setinha significa que a posição do vetor tá armazenando o endereço dessa lista encadeada. Enquanto não tiverem elementos na posição do vetor, aquela posição tá vazia, mas a partir do momento que eu insiro um elemento ela vai passar a ter o endereço da lista encadeada.  
Vou agora colocar aqueles mesmos valores de antes, o 9 e o 17. Agora, vou adicionar o 20, que, através da nossa função hash, dá resto 0, só que eu vi que na posição 0 já tem uma lista encadeada, então o 20 vai ser o próximo elemento da lista encadeada. Quando tem colisão eu só tô aumentando a quantidade de elementos da lista encadeada. Vou adicionar agora o 25, que também dá resto 0, fui lá na posição 0 e vi que tinha o 15, então vou ver se existe algum elemento depois do 15. Existe o 20, vou fazer a checagem de novo, e vi que não tem nenhum, então eu vou inserir o 25 logo após o 20.  
  
Existem 3 principais operações quando a gente trabalha com a função hash: a inserção, que a gente viu agora; a busca e a remoção.  
Vamos ver agora um exemplo de busca, quero ver se tem o valor 14. 14 por 5 dá resto 4, então vou lá no índice 4, existe um ponteiro pra uma lista encadeada lá, então vou checar na lista. Encontrei o 9, não é o 14, então vou ver se existe um próximo valor. Não tem, então o 14 não foi inserido. Agora quero buscar o 13. Dá resto 3, só que não existe nenhum ponteiro no índice 3, então eu já sei que não tá, nem preciso ver nos outros. Se fosse uma busca sequencial eu teria que checar todos esse de cima pra ver se o 13 foi inserido. O pior caso possível seria se eu tivesse uma função hash que mandasse todos os elementos pro mesmo índice, seria a mesma coisa de uma busca sequencial.  
Explica o remover.  
  
Ok, agora vamos ver o encadeamento interior heterogêneo. Exemplo da água e óleo. Vai ter as posições normais do vetor, mas eu vou separar uma parte do vetor pra armazenar apenas as colisões. Explica até homogêneo.  
  
Agora vamos pro encadeamento interior homogêneo, com o teste linear. Eu vou manter aquele mesmo vetor e a mesma função hash, só removi a parte do vetor para as colisões. Explica até o exemplo.