De acordo com a especificação do TP, a documentação deve conter apenas:
• Explicação de qual a função hash utilizada.
• Análise de complexidade dos algoritmos.
• Tabela contendo o tempo de execução dos 4 casos para cada arquivo texto de exemplo.

Documentação Genérica - Parte 1 (a mesma para todo mundo)

Antes de mais nada, vamos entender o que é uma hashtable (não precisam colocar isso na documentação, mas é bom entenderem).

Imaginem que vocês têm o seguinte conjunto de pessoas:

Ana	Luca	Lia	Bruno	Maria	João	Vitor

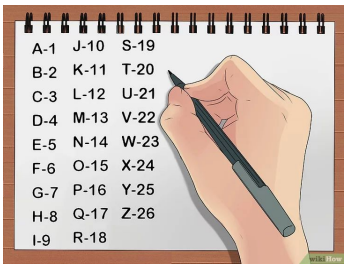
Agora, imaginem que querem consultar essa tabela e pegar os dados do João. Como a tabela não tem nenhuma ordenação e não sabemos onde ele está (finjam que não sabemos), vamos olhar um por um até encontrá-lo. Posição 0... Ana. Posição 1... Luca. Posição 2... Lia. Posição 3… Bruno. Posição 4… Maria. Posição 5… João!
Agora, imaginem que essa tabela tivesse bilhões de nomes e vocês fossem procurar desta maneira. Completamente inviável!

Para isso, podemos utilizar a função hash. Ela faz um mapeamento dos itens e os insere na tabela de acordo com um código, que vou explicar já.

Suponham que o código de cada pessoa seja feito assim:

1. Somamos o valor numérico das letras dos nomes de cada pessoa (A=1, B=2, ...);
2. Somamos esse valor à idade da pessoa (finjam que sabemos as idades);
3. Pegamos o resto da divisão do resultado encontrado acima pelo número de colunas da tabela.

Colocando em prática:



Ana: 1+14+1+15 = 31 % 7 = **3**
Luca: 12+21+3+1+34 = 71 % 7 = **1**
Lia: 12+9+1+19 = 41 % 7 = **6**
Bruno: 2+18+21+14+15+21 = 91 % 7 = **0**
Maria: 12+1+18+9+1+53 = 95 % 7 = **4**
João: 10+15+1+1+34 = 61 % 7 = **5**
Vitor: 22+9+20+15+18+16 = 100 % 7 = **2**

Como podem perceber, cada um ficou com um código diferente. Naturalmente, os códigos estão entre 0 e 6, pois calculamos o módulo de um número com 7 (ou seja, o resto da

divisão de um número por 7). Agora, vamos reorganizar aquela nossa tabela de nomes seguindo a nossa nova ordem.

Bruno	Luca	Vitor	Ana	Maria	João	Lia

Ótimo. Agora, vamos fingir que nada aconteceu e procurar pelo registro do João (coincidentemente, ele está na mesma posição de antes). Antes de sairmos procurando, vamos somar as letras do nome dele com a idade dele e achar o resto da divisão por 7: 10+15+1+1+34 = 61 % 7 = **5**. Ok, vamos direto à posição 5 da tabela e *voilà*! Achamos o João sem ter que percorrer a tabela inteira!

Então vamos entender o que aconteceu aqui. Nós criamos uma função que podemos chamar de “Soma letras do nome com idade”, escolhemos um número para fazer o módulo (no caso, o tamanho da tabela) e demos a cada item da tabela um valor que representa a posição em que ele estará alocado. Isso é hash!

Como nem tudo são flores, percebam que pode ocorrer de itens diferentes terem o mesmo código. Para mostrar o exemplo acima, eu dei uma manipulada nas idades para que a soma desse resultados diferentes em módulo 7, mas na vida real não é assim. Por isso, é necessário encontrar maneiras de resolver as chamadas “colisões”.

Vou mostrar aqui as duas formas mais comuns de resolver: endereçamento aberto e endereçamento fechado (ou encadeamento).

Endereçamento Aberto

Agora, imaginem que temos outras pessoas, fizemos o mesmo cálculo e chegamos ao seguinte endereçamento:

Luiza - **4,** Marcelo - **3,** Patrícia - **6,** Mirella - **4,** Caio - **0,** Fernanda - **5,** José - **6**

Inserimos a Luiza na posição 4, o Marcelo na 3, a Patrícia na 6… Ops! Mirella está na posição 4 também!

			Marcelo	Luiza		Patrícia

Como a posição 4 já está ocupada pela Luiza, vamos inserir a Mirella na posição seguinte, ou seja, na 5. Depois, inserimos o Caio na 0… Ops! A Fernanda tem que ir pra posição 5, mas acabamos de colocar a Mirella lá!

Caio			Marcelo	Luiza	Mirella	Patrícia

Vamos seguir a mesma ideia que usamos antes. Se a posição 5 está ocupada, olhamos para a 6. Ela também está ocupada, então olhamos para a próxima, que é a 0 (imaginem

que o final e o começo da tabela são ligados). Na 0, já temos o Caio, então olhamos para a 1. Está vazia, então colocamos a Fernanda lá. O José é o próximo a ser inserido, e ele vai para a posição 2, já que a 6, a 0 e a 1 estão ocupadas e a seguinte está vazia.

Caio	Fernanda	José	Marcelo	Luiza	Mirella	Patrícia

Feita a tabela, podemos procurar pela pessoa que quisermos. Suponha que procuramos por Fernanda. Fazemos o cálculo do código dela e achamos 5. Vamos à posição 5 e achamos a Mirella. A partir daí, vamos ter que percorrer um a um até achar a Fernanda.

Uma desvantagem deste método é exatamente essa. Colisões são muito prováveis de acontecer e, quanto maior o número delas, maior a chance de um item da tabela ficar distante da posição original. No geral, já é uma economia enorme em relação à **busca sequencial** (olhar um por um até achar a pessoa). Outra desvantagem é no momento de inserção do item na tabela. Se houverem muitas colisões, será necessário checar posição por posição em busca de uma que esteja disponível.

Endereçamento Fechado ou Encadeamento

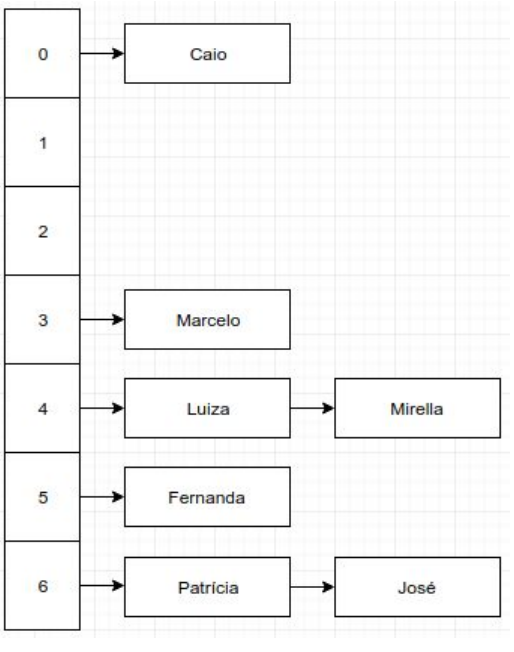
Vamos entender este outro método para resolver o problema das colisões. Suponha nosso mesmo conjunto de pessoas com seus respectivos códigos já calculados:

Luiza - **4,** Marcelo - **3,** Patrícia - **6,** Mirella - **4,** Caio - **0,** Fernanda - **5,** José - **6**

Agora, imaginem nossa mesma tabela, na qual inserimos Luiza na posição 4, Marcelo na 3, Patrícia na 6.

			Marcelo	Luiza		Patrícia

Para inserir a Mirella, vamos fazer um encadeamento na posição 4 (se lembram do TP2? Usamos uma lista encadeada para fazer a lista de adjacências!). Fazemos com que a posição 4 contenha as duas moças. Faremos o mesmo para todos os casos de colisões, como mostrado na imagem a seguir.



Percebam que nem todos os espaços foram utilizados. Uma grande vantagem deste método é que não precisa ficar buscando um lugar para inserir um item, basta encadeá-lo. O mesmo ocorre para a busca. Deve-se procurar o item da mesma forma, mas faz-se isso dentro da posição específica da tabela.

Agora que entendemos como funciona a tabela hash, o que são colisões e como resolvê-las, vamos ver as vantagens e desvantagens do uso deste tipo de método (hash):

Vantagens: Algoritmo simples, busca, retirada e inserção simples e eficientes. Busca com complexidade O(1) no caso médio (já vai direto no item que estamos procurando). Tempo de busca é praticamente independente do número de chaves armazenadas na tabela.

Desvantagens: Nenhuma garantia de balanceamento (distribuição uniforme dos registros), possibilidade de desperdício de espaço alocado para a tabela (vejam no caso do encadeamento) e o grau de balanceamento depende da função hashing escolhida (dificilmente será perfeita).

Ok, galera. Agora que entendemos como funciona uma tabela hash, vamos ao problema do TP!

Temos um texto e queremos contar a frequência de cada palavra, ou seja, quantas vezes cada uma aparece. Usando o exemplo da especificação, seja o texto:

“Dois mais dois são quatro, já dez vezes dez são cem.”

A frequência de cada palavra seria:

cem (1), dez (2), dois (2), e (1), mais (1), quatro (1), são (2), vezes (1)

Para fazer isso, vamos pegar palavra por palavra do texto e endereçá-la utilizando o método que expliquei. A função pode ser escolhida como acharmos mais conveniente (somar as letras e fazer módulo com o número de registros; somar a primeira letra com a segunda, multiplicar por 577 e fazer módulo com 41; multiplicar por laranjas e fazer módulo com maçãs… etc). O importante é que a função evite ao máximo o número de colisões, e utilizar módulo com números primos pode ser uma boa opção. Vou detalhar isso na parte 2 da documentação, que será individual para cada pessoa.

Feito o endereçamento, inserimos as palavra na tabela hash que criamos e, a cada vez que encontrarmos a mesma palavra no texto, incrementamos seu contador.

Ao final do processo, temos nossa tabela hash com as palavras e suas respectivas “aparições” no texto. Com isso, podemos colocá-las em um vetor e ordenar alfabeticamente.

A especificação do trabalho pede que o resolvamos de 4 formas diferentes:

1. Pesquisa sequencial com ordenação em $O(n^2)$

2. Pesquisa sequencial com ordenação em O(n log n)
3. Pesquisa em hash com ordenação em O(n²)
4. Pesquisa em hash com ordenação em O(n log n)

A pesquisa sequencial é aquela em que procuramos um registro olhando um por um na tabela. Imaginem fazer isso para cada palavra em um texto de mais de 100000 linhas! Terrível!

A pesquisa com hash nós já vimos como funciona e com certeza será mais rápida que a sequencial.

Para ordenação em O(n²), podemos usar Bubble Sort, Insertion Sort ou Selection Sort. Se a complexidade é O(n²), quer dizer que será na ordem do quadrado do número de palavras. Isso não é bom, quando se trata de uma quantidade grande de palavras! Imaginem no texto de 100000 linhas!

A ordenação em O(n log n) é bem melhor que a em O(n²), no caso de grande volume de dados. Nessa ordem de complexidade, temos Quick Sort, Heap Sort e Merge Sort, por exemplo.

Resultados esperados

Percebam que pesquisa sequencial, no caso de grande volume de dados, é uma bosta. Ordenação O(n²), nesse caso, também. Então, imaginem unir os dois e trabalhar com arquivos de texto de 100000 linhas! Muito ruim, não é mesmo? Os resultados de tempo de execução e os prints dos testes serão individuais, mas já podemos inferir que a ordem de pior para melhor será a seguinte:

1. Sequencial + ordenação em O(n²)
2. Sequencial + ordenação em O(n log n)
3. Hash + ordenação em O(n²)
4. Hash + ordenação em O(n log n)