

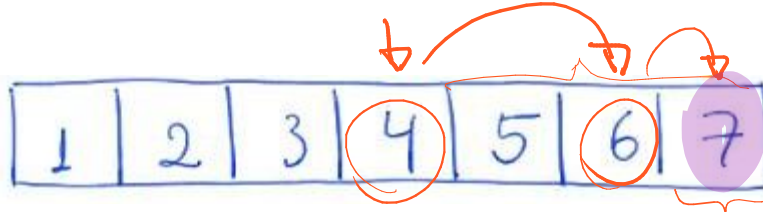
Hash tables, espalhamento e colisões

Para apreciar quão revolucionária é a abordagem das hash tables,

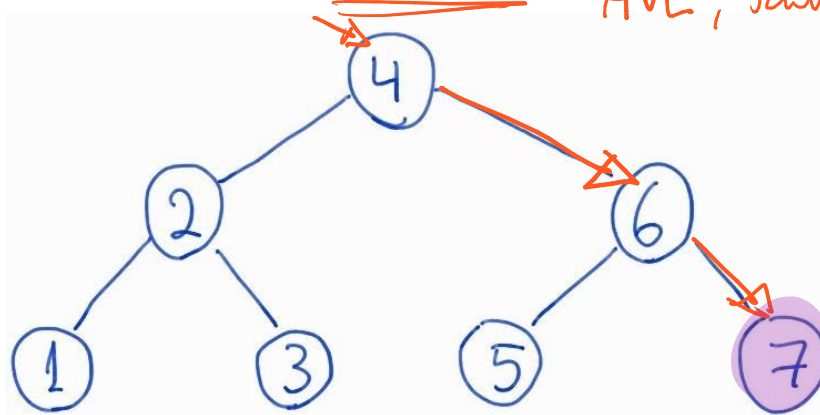
- vamos verificar o que têm em comum
 - outras implementações de tabelas de símbolos.

Para tanto, considere a busca pelo elemento 7 em

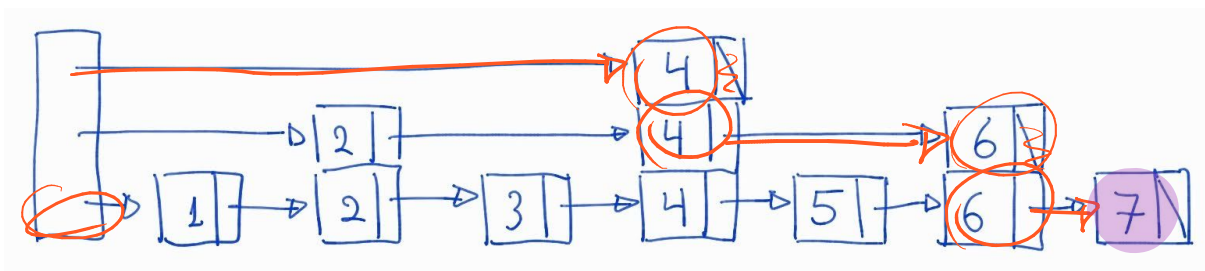
- um vetor ordenado *usando busca binária*



- uma árvore binária de busca balanceada



- ou em skip lists



Note que, todas são baseadas em

- manter os itens ordenados por chave
 - e em dividir o espaço de busca a cada comparação.
- Com isso, levam tempo de busca proporcional a $\log n$,
 - sendo n o número de itens no espaço de busca.
- Isso porque, $\log$ de um número é o número de vezes
 - que este número pode ser dividido pela base.

Será que conseguimos fazer melhor utilizando uma abordagem diferente?

Será que conseguimos fazer melhor utilizando uma abordagem diferente?

- Uma ideia é utilizar um vetor diretamente indexado pelas chaves.
- Uma vantagem dessa abordagem é que o tempo de acesso
 - é constante em relação ao número de elementos, i.e., $O(1)$.
- Se as chaves estiverem num intervalo pequeno,
 - por exemplo, inteiros entre 1 e 1000, isso é viável,
 - pois só precisamos alocar um vetor com mil posições.
- Infelizmente (ou não), esse raramente é o caso.

Como exemplo, considere que queremos criar uma lista de telefones.

- Neste caso, as chaves são os nomes das pessoas,
 - os quais, em geral, não correspondem a inteiros entre 1 e mil.
 - De fato, em geral, nomes não correspondem a inteiros :).
- Insistindo na ideia, podemos converter nomes em números inteiros.
 - Por exemplo, considerando a representação binária de cada nome.
 - Então, usamos esse número para indexar o vetor.
 - Qual o problema dessa abordagem?
- Vamos calcular o tamanho que terá o vetor resultante,
 - lembrando que todo índice deve ter uma posição correspondente.
 - No caso dos nomes, teríamos uma posição
 - para cada nome possível.
 - Considerando que cada caractere tem 26 possibilidades
 - e que um nome pode ter até 30 caracteres, o vetor teria
 - $26^{30} \approx 2,81 \cdot 10^{42}$
 $\approx 2,81 \cdot (10^3)^{14}$
 $\approx 2,81 \cdot (2^{10})^{14} \approx 2,81 \cdot 2^{140}$ posições
 - Como comparação, o armazenamento total disponível na Terra
 - é da ordem de 10^{25} bits.
- Em geral, o tamanho do vetor seria da ordem de 2^n ,
 - sendo n o número de bits da chave.

O exemplo anterior deixa claro que essa abordagem é inviável.

- No entanto, existe uma estrutura de dados que sofisticava essa ideia,
 - que suporta busca, inserção e remoção em tempo constante,
 - i.e., $O(1)$,
 - e ocupa espaço proporcional ao número de elementos armazenados.
- Essa estrutura é a tabela de espalhamento ou,
 - do inglês, Hash table.

Tabelas de espalhamento (hash tables)

Trata-se de uma implementação bastante popular e eficiente

- para tabelas de símbolos.

Hash tables **propriamente implementadas** suportam operações de

- consulta, inserção e remoção muito eficientes,
 - na prática levando tempo constante por operação,
 - em relação ao número de elementos armazenados.

A eficiência das operações depende da hash table ter

- tamanho adequado,
- bom tratamento de colisões e
- uma boa função de espalhamento (hash function).

Vamos detalhar cada um desses tópicos, de baixo pra cima :).

Sobre funções de espalhamento, vale destacar que

- elas também são úteis em outros contextos, como
 - verificação de integridade de dados transmitidos e
 - validação de identificadores, como RGs e CPFs.

Queremos implementar uma tabela de símbolos para

- armazenar itens que possuem chave e valor.
- As chaves estão distribuídas num universo U bastante grande,
 - mas o conjunto de itens S é bem menor.
- Caso contrário, basta usar a ideia anterior do vetor indexado por chave.

Vamos usar um vetor de tamanho M ,

- com M sendo proporcional a $|S|$.

Além de uma função de espalhamento (hash function)

- $h: U \rightarrow \{0, \dots, M-1\}$.

É importante destacar limitações das hash tables, como

- ser fácil implementar uma **função de espalhamento problemática**,
 - ou seja, que não espalha bem os dados.
- Além disso, por melhor que seja a função de espalhamento,
 - a hash table não tem boa garantia de eficiência no pior caso,
 - pois sempre existem conjuntos de dados patológicos.
- Por exemplo, sempre existe chance de todas as chaves em S
 - serem mapeadas para a mesma posição.
- Isso porque, estamos mapeando um conjunto universo grande U
 - para apenas M valores.
 - No mínimo, $|U|/M$ chaves serão indexadas a cada posição.
- Por fim, ao contrário de outras implementações para tabelas de símbolos,
 - nas hash tables os dados não ficam ordenados.
- Por isso, operações como
 - mínimo, máximo, sucessor, antecessor, rank e seleção
 - não são eficientes.

Implementação das funções de **espalhamento** (hash functions)

vetor M
 $h: U \rightarrow 0 \dots M-1$

Mínimo necessário: h deve converter cada chave para um índice do vetor, i.e.,

- $h(\text{chave}) = \text{chave} \% M$

```
int hash(Chave chave, int M) {  
    return chave % M;   
}
```

- Nesta função, chaves próximas
 - tendem a cair próximas ou na mesma posição.
- $h(\text{chave}) = (a * \text{chave} + b) \% M$ 

```
int hash(Chave chave, int M) {  
    return (17 * chave + 43) % M;  
}
```

- Nesta função chaves próximas são mais espalhadas,
 - mas por um fator constante.
- Além disso, em ambas os dígitos menos significativos
 - podem ser os únicos relevantes,
 - dependendo do valor de M .
 - Por exemplo, considere $M = 10$ ou 100 .

Objetivo desejado: h deve ser

- determinística, 
- rápida de calcular, 
- ocupar pouco espaço e 
- espalhar as chaves **uniformemente** pela extensão do vetor.

$U \rightarrow M$

Isso porque, idealmente, cada chave deveria receber uma posição exclusiva,

- como no vetor indexado por chaves que nos inspirou.

Note que, uma função uniforme aleatória tem várias características que desejamos.

- Apesar disso, ela não pode ser usada. Por que?

Supondo que a **chave é uma string**, a seguinte função

- faz com que todo caractere tenha influência no resultado.

```
int hash(Chave chave, int M) {  
    int i, h = 0;  
    for (i = 0; chave[i] != '\0'; i++)  
        h += chave[i];  
    h = h % M;   
    return h;  
}
```

- No entanto, chaves próximas tendem a cair próximas
 - e caracteres com valores múltiplos de M são irrelevantes.

A seguinte função tenta melhorar esses aspectos

- multiplicando cada caractere por um número primo.

```
int hash(Chave chave, int M) {
    int i, h = 0;
    int primo = 127;
    for (i = 0; chave[i] != '\0'; i++) {
        h += primo * chave[i];
    }
    h = h % M;
    return h;
}
```

$$\text{"primo"} = 8 \quad \text{mdc}\{8, 12\} = 4$$

$$M = 12$$

$$8.5 \% 12 = 40 \% 12 = 40 - 3.12 = 4$$

$$8.9 \% 12 = 72 \% 12 = 72 - 6.12 = 0$$

$$8.4 \% 12 = 32 \% 12 = 32 - 2.12 = 8$$

$$\text{caractere} \cdot \text{"primo"} \% M = \text{mdc}\{\text{"primo"}, M\} \cdot k, k \in \mathbb{Z}^+$$

$$\text{primo} = 13$$

$$M = 50$$

- Vale destacar a importância do fator multiplicado ser primo com relação a M,
 - i.e., não ter divisores comuns com M.

$$\text{mdc}\{a, b\} = D$$

$$a = c_a D \quad b = c_b D$$

$$a = q \cdot b + a \% b$$

$$a \% b = a - q \cdot b$$

$$= c_a D - q c_b D$$

$$= D(a - q b)$$

- Caso contrário, as posições múltiplas de $\text{mdc}(M, \text{primo})$
 - serão privilegiadas ou até exclusivas.

- Também é importante que o primo tenha valor próximo de M.

- Caso contrário, em pequenos intervalos de chaves,
 - as menores serão mapeadas consistentemente
- para posições menores.

- Extra: lembrem que em AED1 conhecemos um algoritmo eficiente
 - para determinar se dois números têm divisores comuns.

Apesar dessas melhorias, observe que, chaves que são anagramas, ou seja,

- compostas pelos mesmos caracteres em diferentes ordens,
 - são mapeadas para a mesma posição.
- Mais ainda, quaisquer chaves cujos caracteres somam o mesmo valor
 - continuam caindo na mesma posição.

A seguinte função evita esses problemas,

- usando uma ideia inspirada na notação posicional.

```
int hash(Chave chave, int M) {
    int i, h = 0;
    int primo = 127;
    for (i = 0; chave[i] != '\0'; i++)
        h += pow(primo, i) * chave[i];
    h = h % M;
    return h;
}
```

$$\hookrightarrow \text{primo}^i$$

Chegamos a uma função mais robusta quanto ao espalhamento,

- mas sua eficiência e corretude ainda podem ser melhoradas.

Vale notar que, podemos implementar uma função semelhante à anterior,

- sem usar a **operação de potência**. Isso porque,
 - $c_1 \cdot p + c_2 \cdot p^2 + c_3 \cdot p^3 + \dots = p (c_1 + p (c_2 + p (c_3 + \dots)))$ 

```
int hash(Chave chave, int M) {  
    int i, h = 0;  
    int primo = 127;  
    for (i = 0; chave[i] != '\0'; i++)  
        h = (h * primo + chave[i]);   
    h = h % M;   
    return h;  
}
```

- Note que, esta função tem a mesma ideia da notação posicional,
 - mas os índices menores são multiplicados pelas maiores potências.
- Uma preocupação é que, em todas as nossas funções,
 - o valor de h pode crescer tanto
 - a ponto de ocorrer **erro numérico** de estouro de variável.

Para evitar esse tipo de erro, podemos usar

- a seguinte propriedade do resto
 - $(a + b) \% M = (a \% M + b \% M) \% M$. 
- Assim, chegamos à seguinte função de espalhamento

```
int hash(Chave chave, int M) {  
    int i, h = 0;  
    int primo = 127;  
    for (i = 0; chave[i] != '\0'; i++)  
        h = (h * primo + chave[i]) % M;   
    return h;  
}
```

Eficiência de tempo:

- Nossas funções tem eficiência **proporcional ao número de dígitos da chave**.
 - Por isso, podem não ser eficientes com chaves muito grandes.
- Mas, com relação ao número de itens na nossa tabela,
 - o tempo é constante, i.e., $O(1)$.

Funções de espalhamento de referência:

- FarmHash, MurmurHash3, SpookyHash, MD5

Testar o desempenho de diferentes funções de espalhamento (suas ou da literatura)

- com os dados do seu problema é essencial para realizar uma boa escolha.

Colisões são inevitáveis

Uma colisão ocorre quando a função h mapeia duas chaves diferentes

- para a mesma posição do vetor.

Nosso esforço para projetar uma boa função de espalhamento

- objetivou justamente minimizar o número de colisões.
 - Apesar desse esforços, colisões não são apenas inevitáveis,
 - mas são comuns.

Para obter intuição sobre isso, considere o “paradoxo” do aniversário.

- Nesse, temos um grupo de pessoas numa sala e queremos saber
 - a chance de ao menos duas fazerem aniversário no mesmo dia.
- Num grupo com n pessoas e um ano com 365 dias, a chance
 - de um par específico de pessoas aniversariar no mesmo dia é $1/365$.
- Mas, temos $\binom{n}{2} = C_2^n = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$ pares
- Assim, a probabilidade de ocorrer alguma colisão é $\approx (1/365) \cdot (n^2/2)$.

$$Pr(\text{colisão}) = \frac{n^2}{2} \cdot \frac{1}{365} \quad Pr(\text{1 par colidiu}) = \frac{n^2}{2 \cdot 365}$$

$$Pr(\text{colisão}) = 1/2 = \frac{n^2}{2 \cdot 365} \Rightarrow n^2 = 365$$

$$n = \sqrt{365} \approx 19$$

- Onde está o erro/abuso da expressão acima?
 - Probabilidade da união não é igual à soma das probabilidades.

Generalizando a fórmula anterior, trocamos o número de dias no ano por M .

- Assim, a probabilidade de uma colisão é $1/2$ quando $n \approx \text{raiz}(M) = \sqrt{M}$
 - e é muito provável encontrar colisões quando $n \approx 2 \text{ raiz}(M) = 2\sqrt{M}$
- Note que, para M grande, digamos 10^6 , (1 milhão)
 - devemos encontrar as primeiras colisões quando

$$2 \cdot \sqrt{M} = 2 \sqrt{10^6} = 2 \cdot 10^3 \approx 2 \text{ mil elementos}$$

- elementos forem inseridos na tabela.
 - Ou seja, quando apenas 0,2% da tabela estiver ocupada.