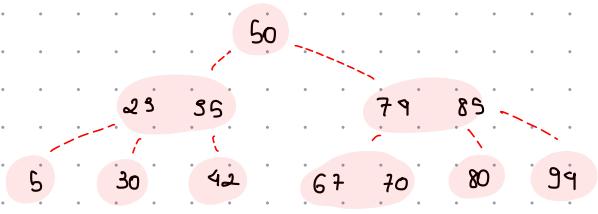


19 de abril

Árvores平衡adas

Árvores 2-3

- cada nó tem 1 ou 2 elementos (2 ou 3) apontadores.
- toda folha está na mesma altura.

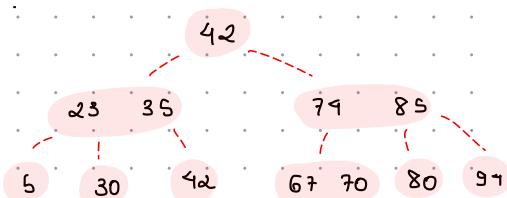


remoção?

- (1) se o elemento a ser removido não está em folha, troca pelo maior menor que ele (ou menor maior) e remove esse elemento.

ex: remover 50:

Troca por 42 (ou 67) e remove 42 da subárvore.



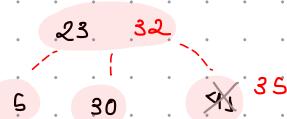
podemos nos concentrar no caso em que estamos removendo de folhas

caso 1: folha é 3-nó

não remove

caso 2: folha é 2-nó

exemplo: 2.1 irmão rico



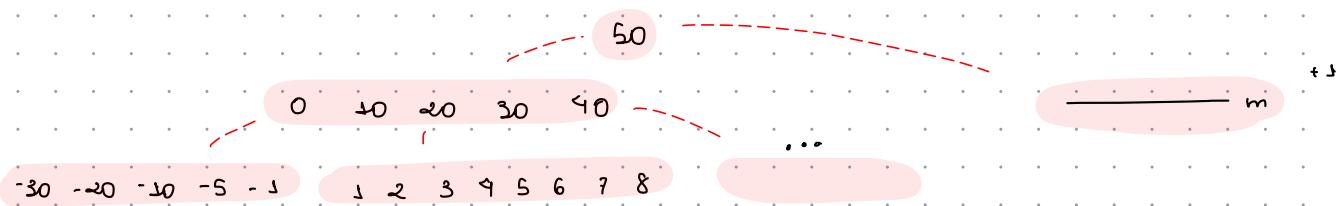
Pega um elemento do irmão, coloca no pai e derruba o pai para o nó de onde removi.

2.2 Irmão pobre



Podemos generalizar árvores 2-3 para armazenar até m chaves em um nó (e $m+1$ ponteiros)

para $m=10$, cada nó deve ter pelo menos $\left\lfloor \frac{m}{2} \right\rfloor = 5$ e no máximo $m=10$ chaves (a menos da raiz).



Para inserir um elemento em uma B-árvore temos 2 casos:

(1) A folha onde o elemento vai ser inserido tem menos de m chaves: Só insere.



(2) O nó tem m chaves: divide em dois nós com $\left\lfloor \frac{m}{2} \right\rfloor$ e $\left\lceil \frac{m}{2} \right\rceil$

chave é o elemento do meio sobe para o pai.



Para remover... podemos considerar que estamos removendo de uma folha.

caso 1: a folha tem mais de $\left\lfloor \frac{m}{2} \right\rfloor$ elementos

fácil! só remove

caso 2: folha tem exatamente $\left\lfloor \frac{m}{2} \right\rfloor$ elementos

caso 2.1: irmão vazio: irmão tem mais de $\left\lfloor \frac{m}{2} \right\rfloor$ elementos. Um elemento do irmão

sobe para o pai e o pai desce para o nó de onde removeu

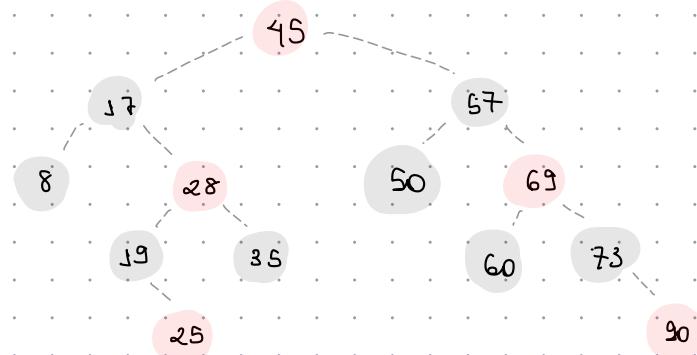
caso 2.2: irmão tem exatamente $\left\lfloor \frac{m}{2} \right\rfloor$ elementos: posso juntar os dois nós trazendo

o elemento do pai.

Árvore vermelho-magenta red-black tree

Uma árvore de busca binária com as seguintes propriedades:

- (i) cada nó tem uma cor (vermelho ou preto)
- (ii) nós vermelhos têm filhos pretos
- (iii) todo caminho da raiz até um NULL tem o mesmo número de nós pretos.



Chamamos de altura preta da árvore o número de nós pretos da raiz até um ponteiro NULL.

Como um nó vermelho tem filhos pretos, a altura da árvore é limitada por $2 \cdot hp$. Além disso, a árvore vermelho-magenta é completa até a altura hp .

Se a árvore tem n nós

$$2^{hp+1} - 1 \leq n \leq 2^{2hp+1} - 1$$

A altura máxima da árvore é $2 \cdot hp$

$$h \leq 2 \cdot hp \leq 2 (\log_2 n + 1) = O(\log_2 n)^{2 \cdot hp}$$

26 de abril

Ideia:

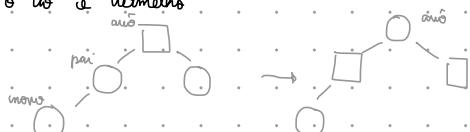
Insera o elemento e pinta de vermelho.

se a árvore está vazia ou o filho de preto OK



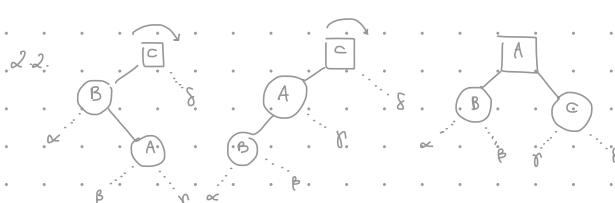
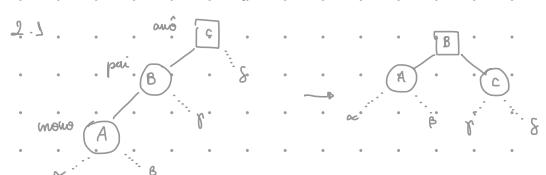
se não tem avô, pinta a raiz de preto

caso contrário, use o tio é vermelho



pode dar problema para cima

caso contrário



```

No * putRB(No * raiz, Key key, Item val){
    if (raiz == nullptr){
        raiz = new No(key, val, vermelho, nullptr);
        return
    }
    No * p = raiz;
    bool achou = false;
    while (!achou){
        if((p -> chave > key) && (p -> esq != nullptr))
            p = p -> esq;
        else if((p -> chave > key) && (p -> esq == nullptr))
            achou = true;
        else if((p -> chave < key) && (p -> dir != nullptr))
            p = p -> dir;
        else if((p -> chave < key) && (p -> dir == nullptr))
            achou = true;
        else achou = true;
    }
    if (p -> chave == key){
        p -> val = val;
        return raiz;
    }

    No * novo = new No(key, val, vermelho, p);
    No * filho = novo;
    while(true){ /* se p eh pai de no vermelho filho */
        if(p -> cor == preto) break;
        No * avo = p -> pai;
        if(avo == nullptr){
            p -> cor = preto;
            break;
        } //avo eh preto
        No * tio = outro filho do avo;
        if(tio != nullptr && tio -> cor == vermelho){
            avo -> cor = vermelho;
            p -> cor = tio -> cor = preto;
            filho = avo;
            p = avo -> pai;
            if(p == nullptr) break;
        }
        else { //caso 2.1
            if(p == avo -> esq && filho == p -> esq){
                No * q = rodDir(avo);
                q -> cor = preto;
                avo -> cor = vermelho;
                if(raiz == avo) raiz = q;
            } break;
            else if(p == avo -> esq && filho == p -> dir){ /* 2.2*/
                No * q = rodaEsq(p);
                No * r = rodaDir(avo);
                r -> cor = preto;
                avo -> cor = vermelho;
                if(raiz == avo) raiz = r;
                break;
            }
            else if // 2.3
            ...
        }
        else {
        }
    }
    return raiz
}

```

```
No * rodaEsq(No * r){  
    No * aux = r -> dir;  
    if(aux == nullptr) return r;  
    r -> dir = aux -> esq;  
    aux -> esq = r;  
    aux -> pai = r -> pai;  
    r -> pai = aux;  
    if(r -> dir != nullptr)  
        r -> dir -> pai = r;  
    return aux;  
}
```

Remoção de árvore Rubro-Negra

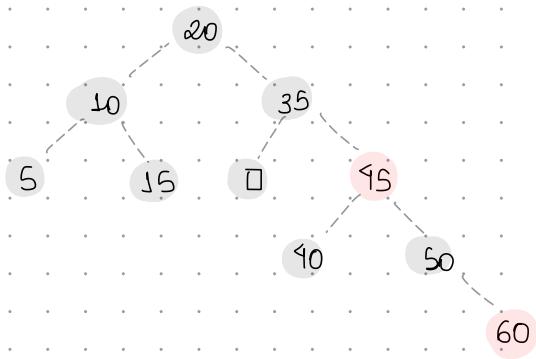
tirei fotos

* Lembrar o que faz cada caso

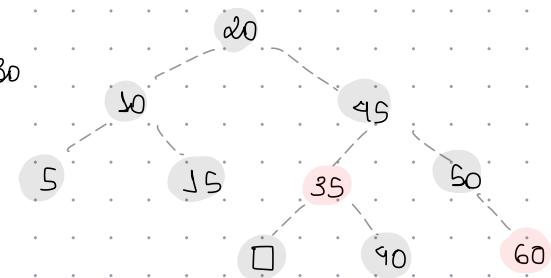
Exemplo: remoção árvore Rubro-Negra

duplo preto

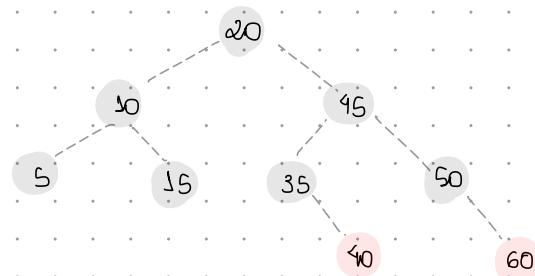
28 de abril



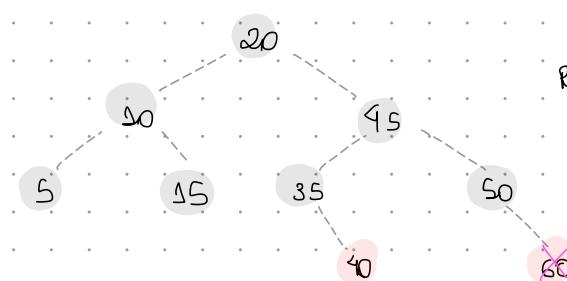
Remove 30



caso 2.2



caso 2.1



Remove 60

tempo de inserção: altura da árvore

no caso da Rubro-Negra: $\log n$

tempo de remoção: altura da árvore: $\log n$

Tabelas de símbolos

	busca	inserção	remoção	rank
vetor desordenado	n	constante	$n + \text{constante}$	n
vetor ordenado	$\log n$	$\log n + n$	$\log n + n$	$\log n$
lista ligada desord.	n	$n + \text{constante}$	$n + \text{constante}$	n
lista ligada ordon.	n	$n + \text{constante}$	$n + \text{constante}$	n
ABB	altura	altura	altura [$\log n \leq \text{altura} \leq n - 1$]	altura
treaps	altura	altura	altura [$\text{altura esperada: } \log n$]	altura
2-3	$\log_2 n$	$\log_2 n$	$\log_2 n$	altura
R.N	$\log_2 n$	$\log_2 n$	$\log_2 n$	altura
AVL	$\log n$	$\log n$	$\log n$	altura

Tabelas de Hashing

Uma tabela de hashing é, de certa forma, uma generalização de um vetor. Temos um conjunto U de onde vêm as possíveis chaves, do qual n serão selecionados e armazenados em uma estrutura.

Exemplo: Em um país com 10000 habitantes a chave de identificação poderia ter 7 dígitos e seria o índice de um vetor representando o código.

Nem sempre temos memória suficiente para fazer isso, pois $|U| > n$.

Devemos ter uma função que leva os elementos do conjunto U para índices desta tabela $[0 \dots m-1]$.

Essa é a ideia de hashing.

Idealmente esperamos que não haja muitas colisões, ou seja, dois ou mais elementos de U que foram selecionados compartilham o mesmo valor da função de hash.

Para projetar uma boa estratégia de hashing precisamos atacar dois problemas:

1. construção da função de hashing

2. tratamento das colisões

1. Funções de hashing

Considere uma tabela de hashing de tamanho m que armazena n valores.

A razão $\frac{n}{m}$ é chamada de fator de carga.

A função de hashing deve devolver um índice $[0, m-1]$.

Hashing modular

Transforme a chave em um número e aplique $\% m$. Se m é primo, o espalhamento é melhor.

Hipótese de hashing uniforme

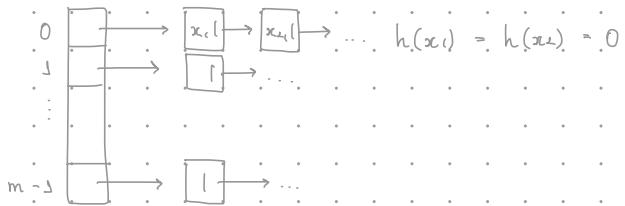
A função de hashing distribui as chaves de forma uniforme e independente entre $[0, m-1]$.

Com isso, se $k_1, k_2 \in U$

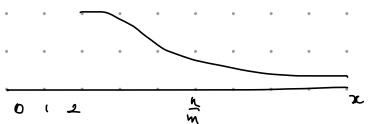
$$\text{prob}[h(k_1) = h(k_2)] = \frac{1}{m}$$

Tratamento de colisões

• Encadeamento



O tamanho esperado de cada lista é $\frac{n}{m}$. Mas, será que não é possível que tenhamos uma lista muito longa?



A probabilidade de que uma lista tenha tamanho K

$$\binom{n}{k} \cdot \left(\frac{1}{m}\right)^k \cdot \left(1 - \frac{1}{m}\right)^{n-k}$$

verifica K caem no mesmo lugar, outras caem
diferentes lugares

Essa probabilidade satisfaz uma distribuição binomial. Tomando $\alpha = \frac{n}{m}$

$$\binom{n}{k} \cdot \left(\frac{\alpha}{n}\right)^k \cdot \left(1 - \frac{\alpha}{n}\right)^{n-k}$$

Por exemplo, $\alpha = 10$, a prob de termos uma lista com ≥ 20 elementos é $< 8\%$

Normalmente usamos $\alpha = 2, 3 \dots$

03 de maio

Hashing

- encadeamento

- teste linear (The art of Comp. Prog., V2, Knuth)

Para inserir um elemento você procura uma posição vazia a partir de $h(x)$ e insere o elemento ali. Para buscar um elemento inicie a busca a partir de $h(x)$ até encontrar-lo ou encontrar uma posição vazia. Para a remoção precisamos diferenciar entre a posição estar vazia ou disponível para inserção.

É possível pronosticar que o número médio de buscas para encontrar um elemento na tabela é dado por $\frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$ onde $\alpha = \frac{n}{m}$.

Exemplo: $\alpha = \frac{1}{2}$ ($m = 2n$) o nº esperado de testes é $\frac{1}{2} (1 + 2) = \frac{3}{2}$.

Para uma busca bem sucedida, o nº esperado de comparações é $\frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$

Ex: $\alpha = \frac{1}{2}$ ($m = 2n$)

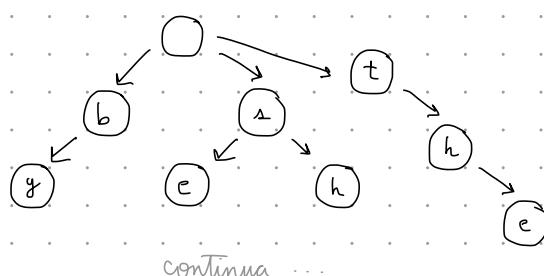
No livro do Sedgewick & Wayne tem uma explicação desse método.

Trie

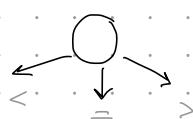
Trie é uma estrutura para implementar uma tabela de símbolos sobre strings.

É uma árvore de busca composta por nós e ponteiros.

Ex: {she, sells, sea, shells, by, the, where}



Para economizar tempo podemos fazer tries ternárias



É uma árvore em que cada nó corresponde a um caractere. As strings são obtidas concatenando os caracteres dos nós do caminho da raiz até um vértice.

O espaço ocupado por uma trie pode ser muito grande (no máximo $R^{\text{maior palavra}}$) onde R é o tamanho do alfabeto.

05 de maio

O tempo para encontrar uma palavra prima true é \log_R^n

Observe que isso independe do número de strings armazenadas na true.

Para uma busca com sucesso, examinaremos em média \log_R^n (n número de palavras, R tamanho do alfabeto)

Ex: $n = 10000$

$$\approx 3$$

$$R = 26$$

A probabilidade de uma string aleatória ser extremamente igual a palavra do conjunto das t primeiras letras é $\frac{1}{R^t}$. Então, para diferir de todas as palavras do conjunto é $\left(1 - \frac{1}{R^t}\right)^n$. Com isso, a probabilidade de que essa string coincida nas t primeiras letras com alguma palavra do conjunto é $1 - \left(1 - \frac{1}{R^t}\right)^n$

$$t = 0 \quad \downarrow \rightarrow \checkmark$$

procurando busca com sucesso

$$t = 1 \quad \downarrow \rightarrow \left(1 - \frac{1}{R}\right)^n$$

$$t = 2 \quad \downarrow \rightarrow \left(1 - \frac{1}{R^2}\right)^n$$

Lembrando que $\left(1 - \frac{1}{x}\right)^x \approx e^{-1}$

$$\left(1 - \frac{1}{R^t}\right)^n = \left(\left(1 - \frac{1}{R^t}\right)^{R^t}\right)^{\frac{n}{R^t}}$$

$$t = 1 \rightarrow 1 - (e^{-1})^{\frac{n}{R}} \rightarrow 1$$

$$t = 2 \rightarrow 1 - (e^{-1})^{\frac{n}{R^2}} \rightarrow 1$$

Exemplo: $n = 10000$

$$t = \log_R^n \rightarrow < 1$$

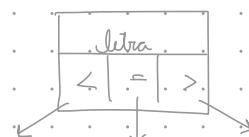
$$R = 26$$

isomando para $t = 0, 1, 2, \dots$ ate $t = \log_R^n$ os valores são menores que 1 e a partir daí, os valores são próximos de 0.

Ex: $\underbrace{t=0}_1 + \underbrace{t=1}_{1-2 \cdot 10^{-7}} + \underbrace{t=2}_{1-0,22} + \underbrace{t=3}_{1-0,94} + \underbrace{t=4}_{1-0,999} < \log_R^n$

O problema das tries é a memória muito grande. Podemos resolver isso usando

as tries ternárias:



Observe que o tempo para achar uma string aumenta.

Uma trie permite que verifiquemos eficientemente se uma string t é prefixo de alguma string u do conjunto dado para construir a trie.

Aplicação: Suponha que seu tenho uma sequência u de DNA muito longa e queremor saber se uma outra sequência t é subsequência de u .

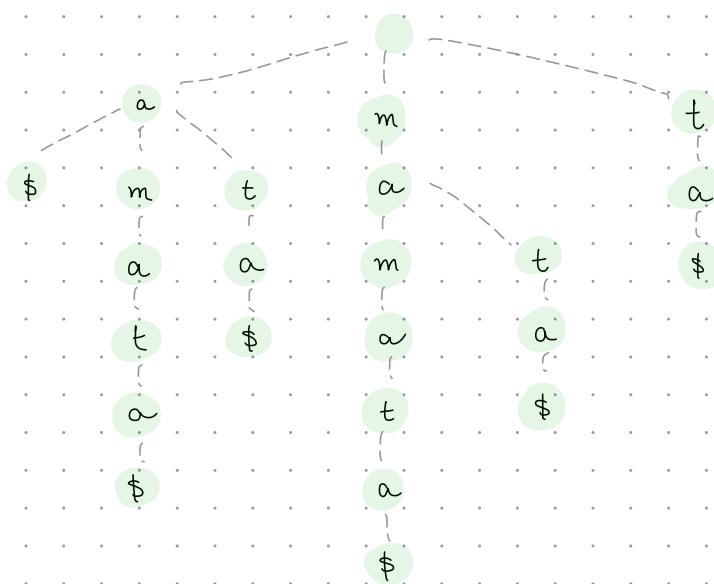
$u: \text{ATACCGA} \dots \text{TAACGGATC}$ $\xrightarrow{10^9}$

$t: \underline{\text{ATGGACTAAT}}$ $\xrightarrow{10^9}$
segmentos

Podemos construir uma trie com todos os sufícos da sequência.

Ex: $u = \text{umamata\$}$

$$\mathcal{S} = \{ \text{umamata\$}, \text{amata\$}, \text{mata\$}, \text{ata\$}, \text{ta\$}, \text{a\$} \}$$



Para economizar espaço podemos guardar, em cada nó, uma string que compõe os caminhos de vértices de grau 1.

