ESTUDOS PARA P3

- TIPOS ABSTRATOS DE DADOS
  - BAG
  - PILHA
  - FILAS
- ANÁLISE DE ALGORITMOS
- TABELAS DE SÍMBOLOS
  - LISTA LIGADA
  - VETOR ORDENADO
  - ÁRVORES DE BUSCA BINÁRIA
  - TREAPS
  - ★ HEAPS
  - ★ ÁRVORE AVL
  - ÁRVORE 2-3
  - ÁRVORE RUBRO-NEGRA
  - TABELAS DE HASHING
  - TRIE

- altura
- busca
- remoção

exercícios

* Questão de simulação:
Ex: Simule a inserção/deleção em uma RN ou 23 ok

* 1 de Pilhas e Filas
Ex: Passe os elementos de uma Fila de inteiros de 1 até n até outra fila de modo ordenado usando duas pilhas.

* 1 de Tabela de Símbolos:
Ex: Dada uma sequência de números, exiba sem repetição e analise a complexidade para cada tipo (VO, ABB, 23 e RN)

* 1 de Backtracking, lista ligada, tries, heap, hashing

Exercício de pilhas e filas

1 1 2 3

1 2 3

{ busca
{ adiciona

VO	ABB	2-3	RN
$\log_2 n$	n	$\log_2 n$	$\log_2 n$
n	n	n	n
=	=	=	=
$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(n \log_2 n)$

TABELAS DE SÍMBOLOS

	busca	inserção	remoção	rank	altura	comparações
v desord.	n	1	$n+1$	n		
VO	$\log n$	$\log n + n$	$\log n + n$	$\log n$		$2(\log_2 n + 1)$
LL desord.	n	$n+1$	$n+1$	n		n^2
LLO	n	$n+1$	$n+1$	n		$2(\log_2 n + 1)$
ABB	altura	altura	altura $[\log_2 \leq \text{altura} \leq n-1]$	altura	maior profundid	$2(\log_2 n)$
Treaps	altura	altura	altura $[\log n]$	altura	$\log_2 n$	—
Heaps	—	—	—	—	$\log_2 n$	—
AVL	$\log_2 n$	$\log_2 n$	$\log_2 n$	altura	$\log_2 n$	
2-3	$\log_2 n$	$\log_2 n$	$\log_2 n$	altura	$\log_3 n$ $\log_2 n$	
R.N.	$\log_2 n$	$\log_2 n$	$\log_2 n$	altura	$2(\log_2 n + 1)$	
Hashing	n (?)	—	—	—		buscas: $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
Trie	—	—	—	—	maior chave	

TIPOS ABSTRATOS DE DADOS

BAG

é uma estrutura de dados capaz de armazenar uma coleção de objetos

operações:
```
void add(Item item)
bool isEmpty()
int size()
Item at(int i)
```

a implementação pode usar vetores estáticos, lista ligada, vetores dinâmicos, etc.

PILHA

é uma estrutura em que inserções, remoções e acessos são feitos em uma das extremidades, chamada topo

operações:
```
push()
pop()
top()
isEmpty()
size()
```

a implementação pode usar lista ligada e vetor dinâmico

- problemas com pilha:
  dada uma sequência (, [, { (ou o problema do estacionamento), verificar se é bem formada.
  resolução: colocar na pilha enquanto 'abre' e tirar da pilha se encontrar algum par.

FILAS

estrutura de dados em que inserções são feitas em uma das extremidades (fim) e as remoções e acessos na outra (começo)

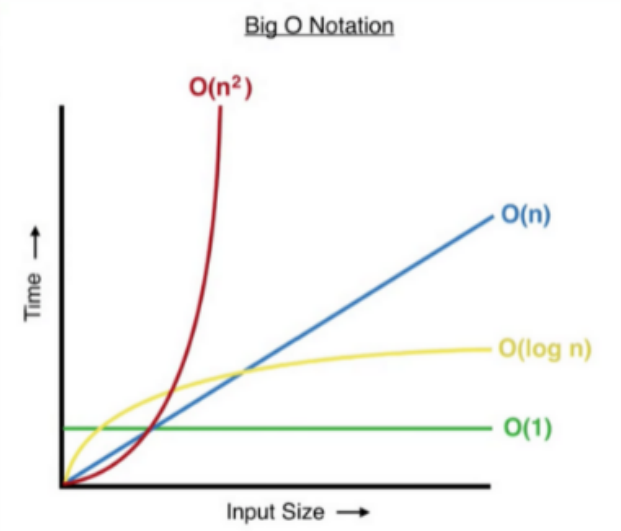
operações:
- add()
- remove()
- first()
- isEmpty()
- size()

a implementação pode usar lista ligada e vetor dinâmico

análise da complexidade de algoritmos

- o tempo que leva para um algoritmo ser executado é baseado no número de passos.
- a **Big O Notation** indica a complexidade de um algoritmo
- a função de tempo T(n) representa a complexidade do algoritmo, tal que Tn = O(n)

$O(1) > O(\log n) >$
$O(n) > O(n \log n) >$
$O(n^2) > O(2^n)$



CONSTANTE: O(1)

- o número de operações não muda, mesmo quando o input varia.

toda complexidade diferente da constante se dá em relação ao número de itens que a sua função recebe.

LOGARÍTMICA: O(log n)

- logaritmos são o inverso de exponenciais. log 2 (8) = 3
- para uma lista de 8 números, só é necessário verificar 3 no máx.
- busca binária em uma lista já ordenada.
- em Big O Notation, O(log n) é o mesmo para todas as bases.

LINEAR: O(n)

- proporcional ao tamanho da entrada
- se tiver dois 'for', por exemplo, a complexidade será O(n) e não O(2n)

quanto maior a entrada, menos importa se é n ou 2n

QUADRÁTICA: $O(n^2)$

- 2 loops aninhados
- é diferente de $O(2n)$, pois um loop está dentro do outro
- o mesmo vale para complexidade cúbica $O(n^3)$

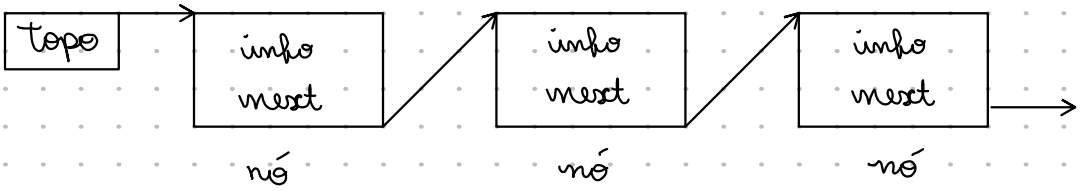
ANÁLISE AMORTIZADA

Tem várias formas de fazer análise amortizada. Uma delas é o método de contagem e o outro é o método de créditos.

LISTA LIGADA

uma lista ligada é uma estrutura que corresponde a uma sequência lógica de entradas ou **nós**. Tipicamente, em uma lista ligada há um ou dois pontos conhecidos de acesso - normalmente o **topo** da lista (seu primeiro elemento) e eventualmente o fim da lista (seu último elemento).

Cada nó armazena também a localização do próximo elemento na sequência, ou seja, de seu **nó sucessor**. Desse modo, o armazenamento de uma lista não requer uma área contígua de memória.



VETOR x LISTA LIGADA

Operação	Vetor	Encadeamento
Inserção início	O(n)	**O(1)**
Inserção final	O(1)	O(1)
Inserção posição "x"	O(n)	O(n)
Pesquisa elemento "x"	O(n)	O(n)
Pesquisa posição "x"	**O(1)**	O(n)
Exclusão início	O(n)	**O(1)**
Exclusão final	**O(1)**	O(n)
Exclusão posição ou elemento "x"	O(n)	O(n)

VETOR ORDENADO

com vetores ordenados implementamos a busca em tempo $O(\log n)$, mas a inserção pode consumir tempo $O(n)$, porque temos de 'empurrar' os elementos para inserir o elemento novo.

- a busca binária faz no máximo $2(\log_2 n + 1)$ comparações

ÁRVORES DE BUSCA BINÁRIA

Uma árvore de busca binaria é uma árvore binária em que para todo nó, todos os elementos de sua subárvore esquerda são menores que ele, e todos os elementos da subárvore direita são maiores.

Buscas bem sucedidas (inserções) numa ABB com n chaves em ordem aleatória fazem em média $2(\log_2 n)$ comparações

- o número de comparações para encontrar um nó é uma a mais que a profundidade de um nó
- profundidade média = $\frac{\text{soma das profundidades}}{n}$

seja C(n) = soma dos comprimentos

∴ nº esperado de comp. em uma busca com sucesso: $1 + \frac{C(n)}{n}$

> o consumo de tempo das funções busca, insere e remove é, no pior caso, proporcional à **altura** da árvore

TREAPS (tree + heap)

Key e priority

Estudos mostram que com inserções e remoções uma árvore, mesmo criada aleatoriamente, pode ficar desbalanceada.

Uma ideia simples para ter uma árvore de altura esperada ~ $\log_2 n$ são as treaps.

Uma treap é uma árvore de busca binária aleatorizada, onde cada nó tem um campo extra chamado prior (= priority)

As prioridades satisfazem a propriedade de um heap: todo nó deve ter prioridade maior que a prioridade dos seus filhos.

> o consumo de tempo de busca e remoção numa treap é, no pior caso, proporcional à sua altura (~ $\log_2 n$)

HEAPS

- os filhos são menores ou iguais
- AB completa ou quase-completa da esquerda para a direita
- prioridade
- altura ~ $\log_2 n$

ÁRVORE AVL

A AVL (Adelson - Velskii e Landis) é uma árvore altamente balanceada, isto é, nas inserções e exclusões, procura-se executar uma rotina de balanceamento tal que as alturas das sub-árvores esquerda e sub-árvores direita tenham alturas **bem próximas** (diferem no máximo por uma unidade).

- complexidade $O(\log_2 n)$ (busca, inserção, remoção)

ÁRVORE 2-3

uma árvore 2-3 é uma árvore de busca em que os nós podem ser do tipo:

(1) **2-nós**: contém uma <chave, valor> e dois apontadores (um para os menores e outro para os maiores.

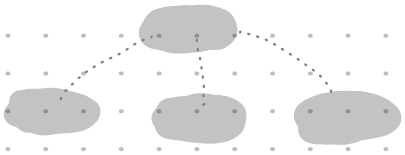
(2) **3-nós**: contém dois pares <chave, valor> e três apontadores (um para os menores que o primeiro, um para os entre as duas chaves e um para os maiores).

- uma árvore 2-3 é balanceada se todas as folhas estão na mesma altura.

<ch1, v1>	<ch2, v2>	dois-no (bool)
ap1	ap2	ap3

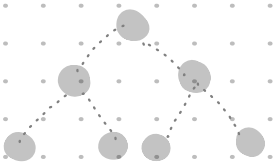
estrutura do nó de uma árvore 2-3

- a menor árvore 2-3 é aquela que todo nó é um 3-nós



$\log_3 n$

- a maior árvore 2-3 só tem 2-nós



$\log_2 n$

- a altura da árvore varia entre $\log_3 n$ e $\log_2 n$

ÁRVORE RUBRO - NEGRA

uma árvore de busca binária com as seguintes propriedades

(1) cada nó tem uma cor (vermelho ou preto).

(2) nós vermelhos têm filhos pretos

(3) todo caminho da raiz até um NULL tem o mesmo número de nós pretos.

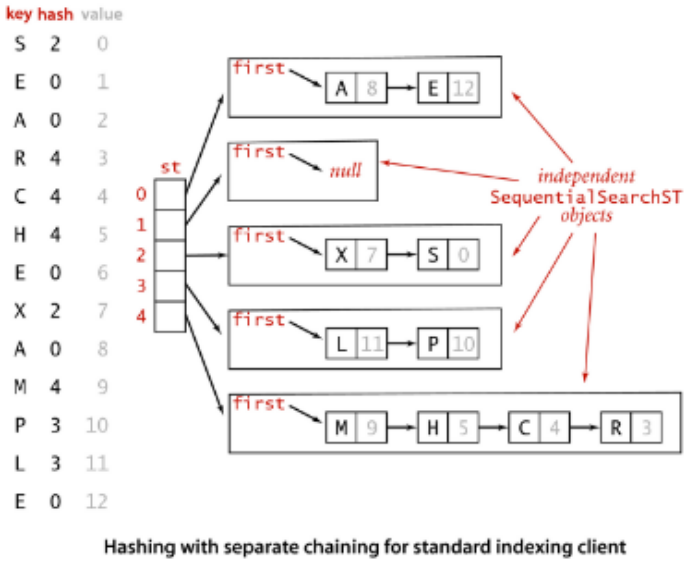
- altura de no máximo $2 \log(n+1)$

HASHING

- função de espalhamento
- mecanismo de resolução de colisões

as colisões na tabela de dispersão podem ser resolvidas por meio de **listas ligadas**: todas as chaves que tem um código hash são armazenadas numa lista ligada.

Podemos supor que os elementos da tabela de dispersão são ponteiros para listas ligadas.



Hashing with separate chaining for standard indexing client

- número médio de buscas: $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ onde $\alpha = \frac{n}{m}$

n = número de chaves

m = número de posições

- complexidade $O(1)$ se n é proporcional a m

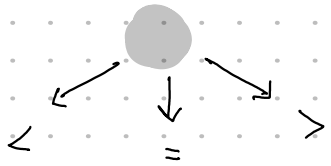
  pior caso: $O(n)$ (tem que percorrer a lista ligada)

- número médio de comparações: $\alpha = \frac{n}{m}$

TRIE

é uma estrutura de dados para implementar uma tabela de símbolo sobre strings.

- é uma árvore de busca formada por nós e ponteiros



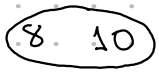
inserção 2-3

8 - 10 - 14 - 7 - 3 - 16 - 21 - 28

34 - 1 - 5 - 11

1. 

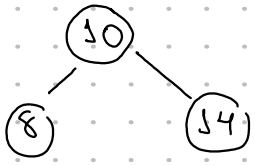
insere 8

2. 

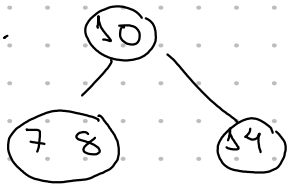
insere 10

3. 

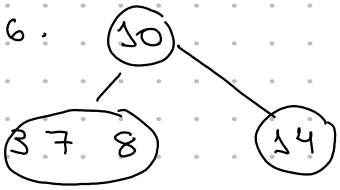
insere 14, porém o nó está muito grande → dividir

4. 

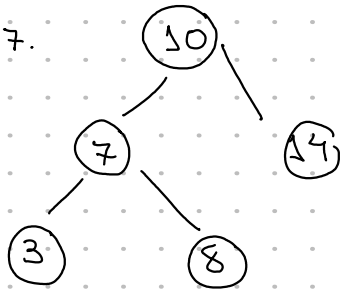
insere 7 →

5. 

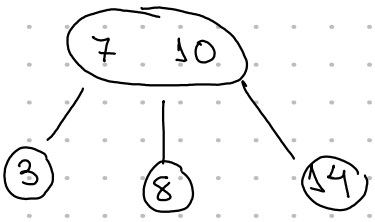
insere 3

6. 

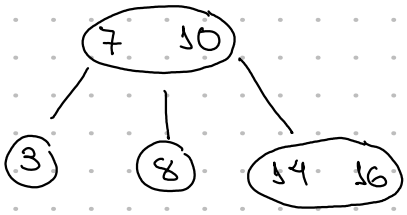
nó muito grande → dividir

7. 

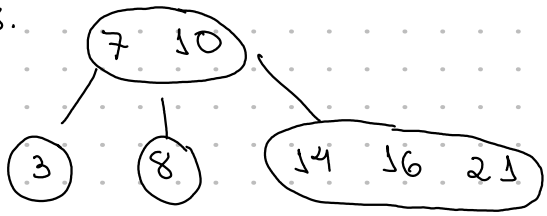
o 7 sobe para balancear

8. 

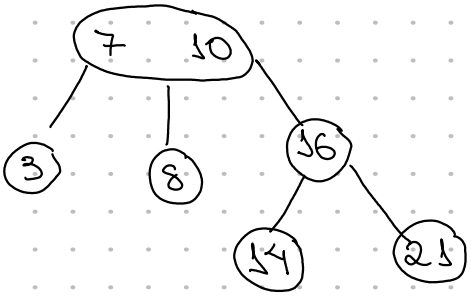
insere o 16 →

9. 

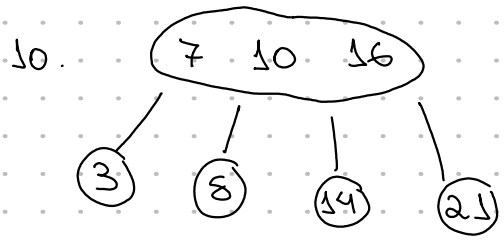
insere o 21 →

8. 

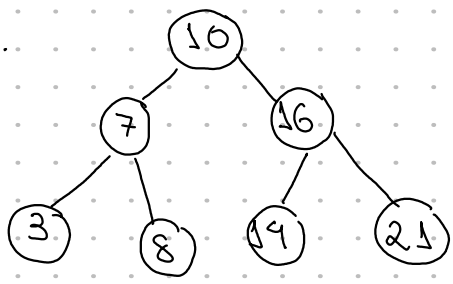
nó muito grande → divide

9. 

sobe o 16 para balancear

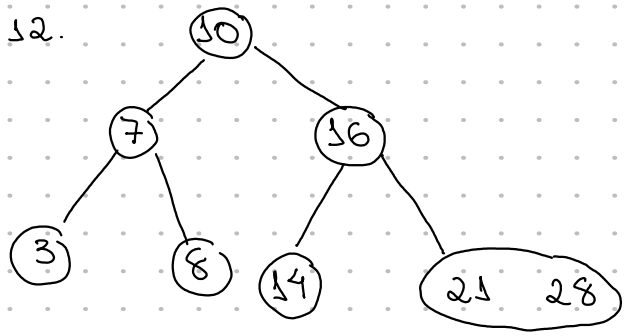
10. 

nó muito grande → divide

11. 

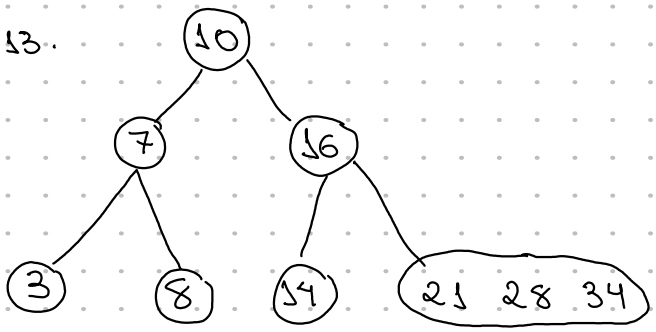
insere 28 →

12.



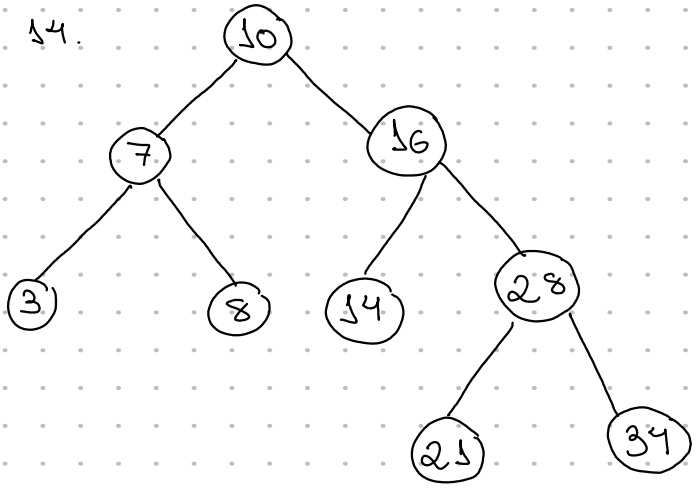
insere 34 →

13.



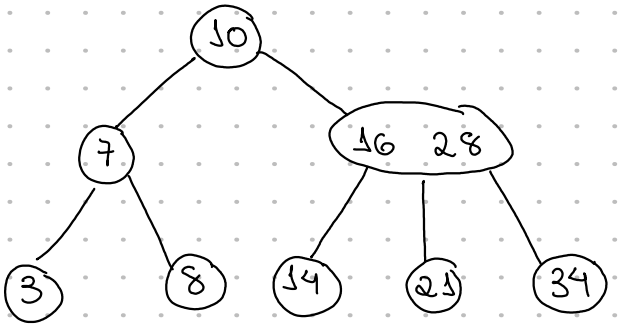
nó muito grande → divide

14.



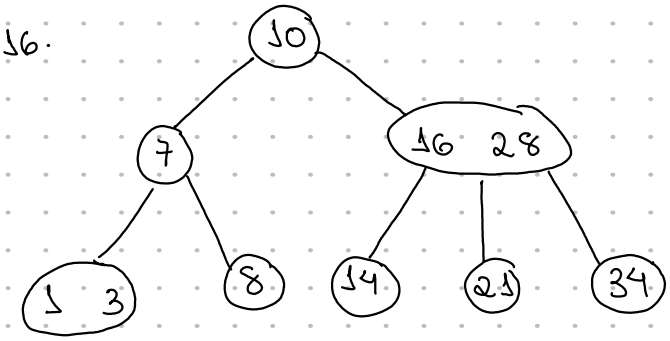
sobe o 28 →

15.



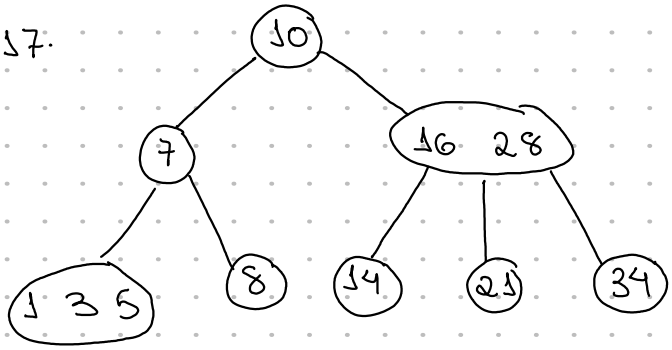
insere o 1 →

16.



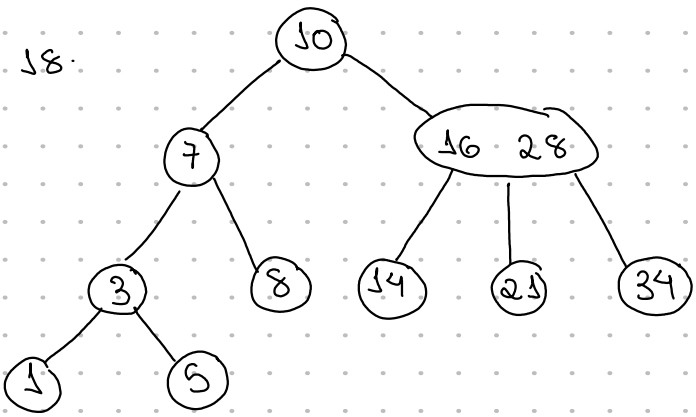
insere o 5 →

17.



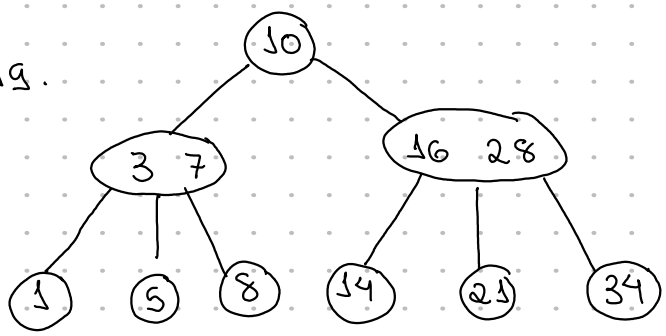
divide

18.



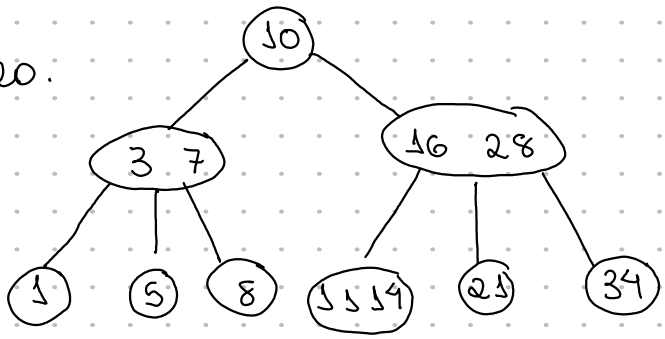
sobe o 3

19.



insere o 11

20.

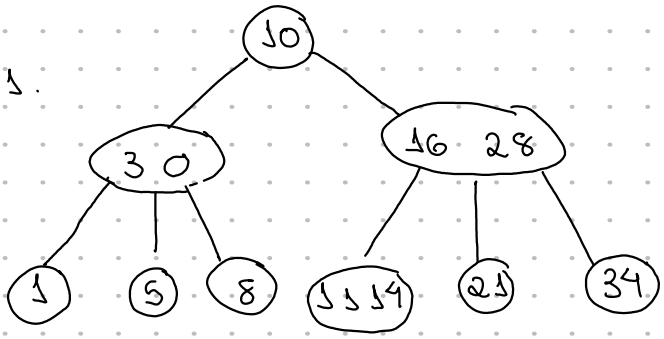


remoção 2 - 3

lembrar do sucessor

7 - 3 - 5

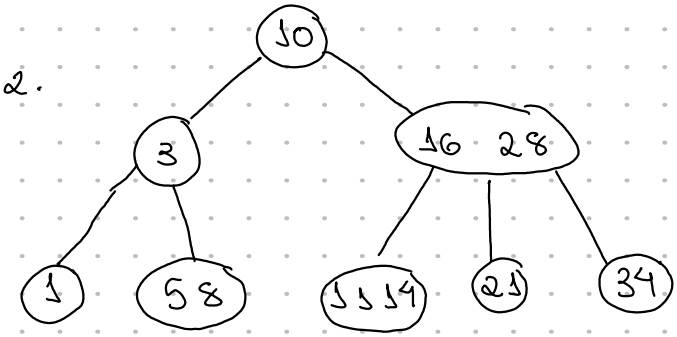
1.



remove 7

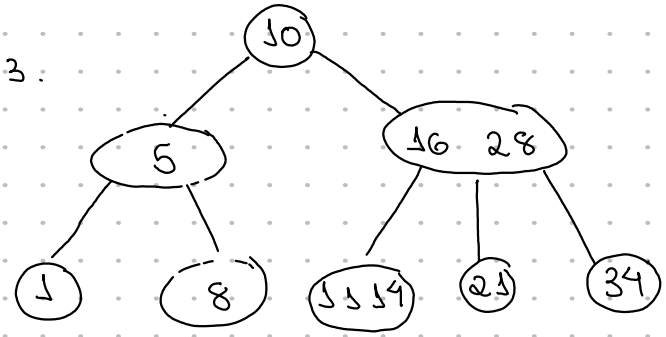
juntamos 5 e 8

2.



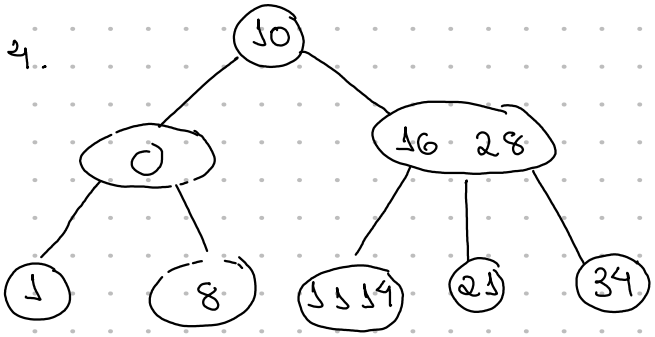
remove 3

3.



sobe o sucessor = 5

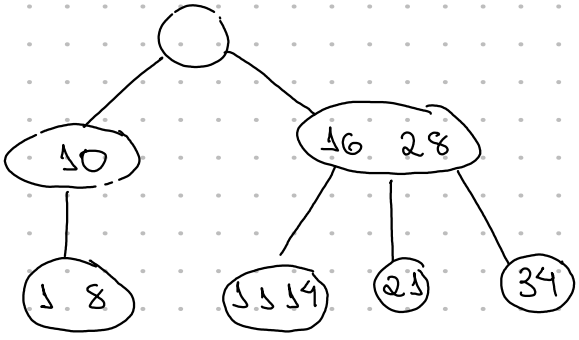
4.



remove 5

junta 1 e 8

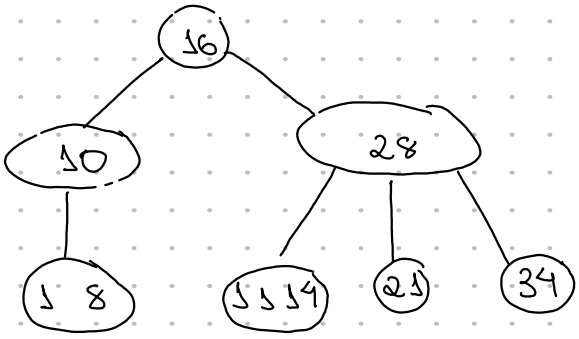
5.



desce o 10

e o 16

6.

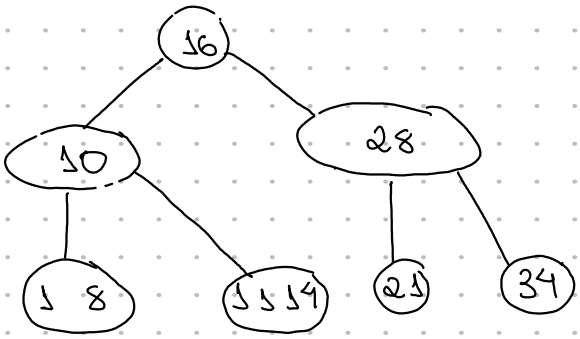


o nó é muito pequeno

então passamos o

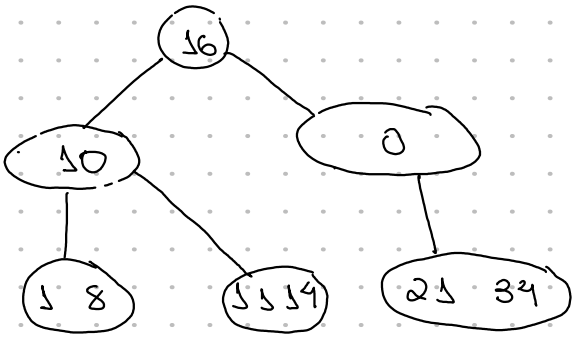
11 e 14

7.



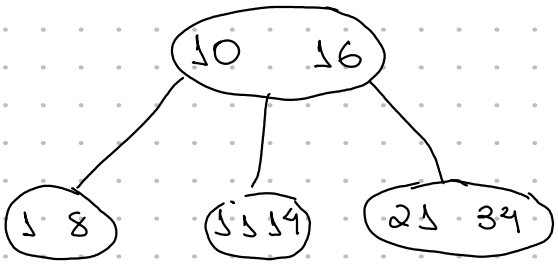
remove o 28

8.



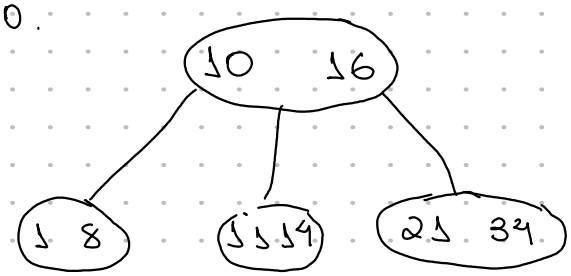
juntamos o 21 e 34
desce o 16

9.



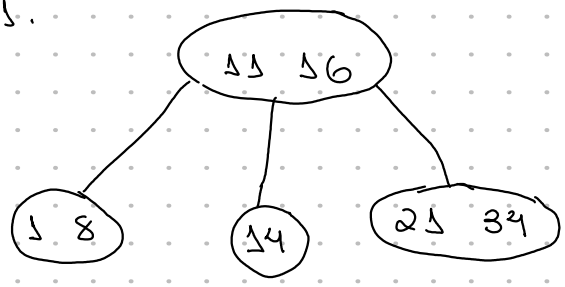
junta 10 e 16

10.



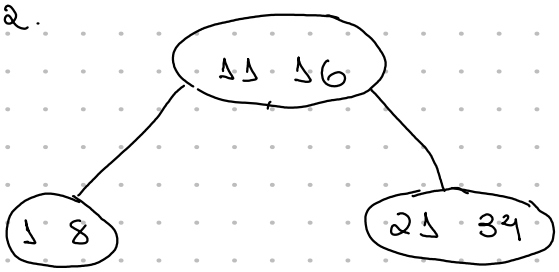
remove 10

11.



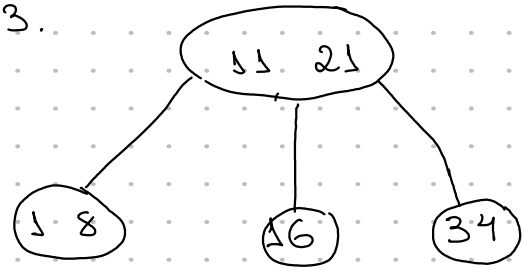
sobe o sucessor
remove 14

12.



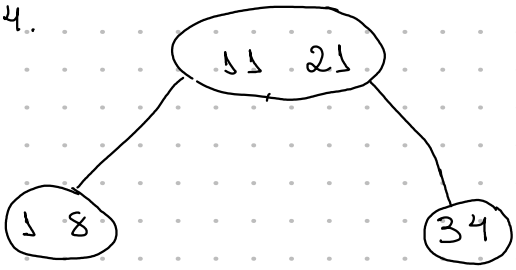
desce o 16 e
sobe 21

13.



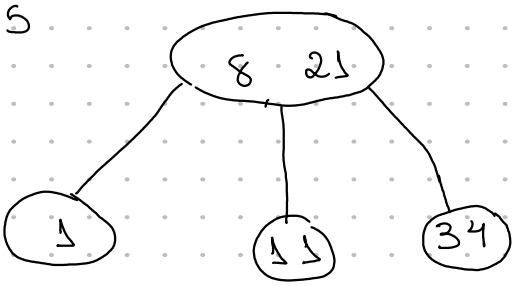
remove 16

14.



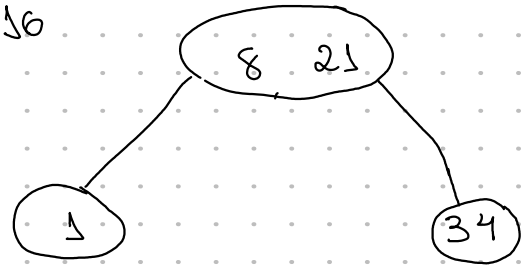
desce o 11 e
sobe o 8

15



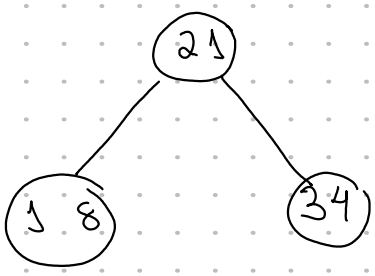
remove 11

16



junta 8 e 1

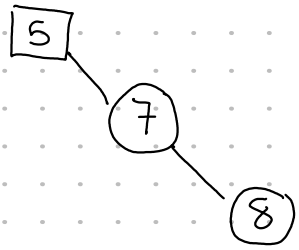
17



inserção RB

5 - 7 - 8 - 3 - 4

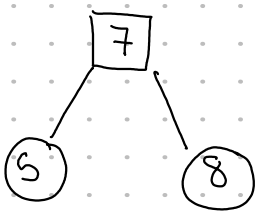
1.



dois vermelhos

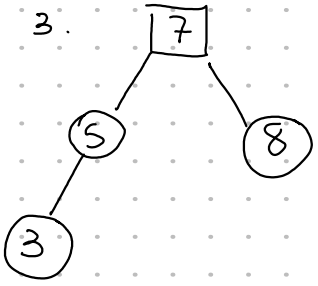
rotaciona

2.



insere 3

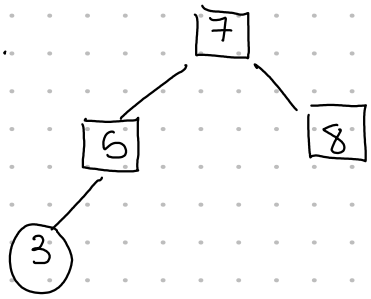
3.



troca cor

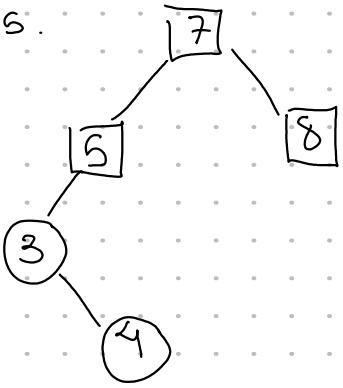
pai tio avô

4.



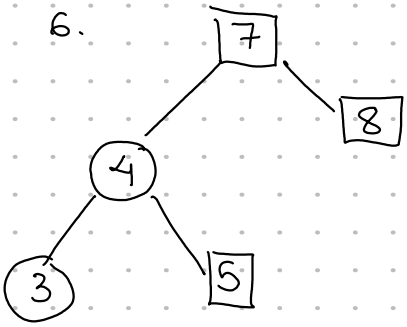
insere 4

5.



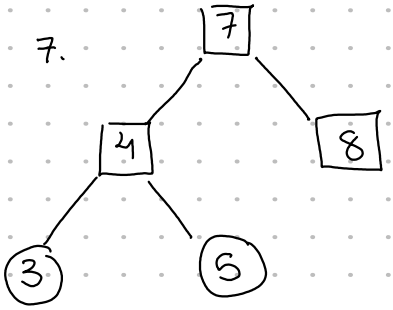
rotaciona

6.



troca cor

7.

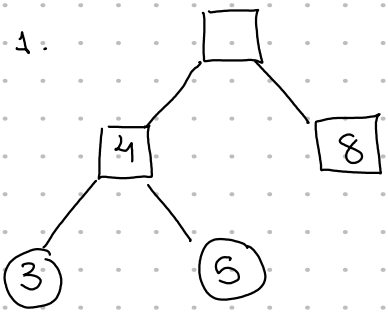


remoção

antecessor

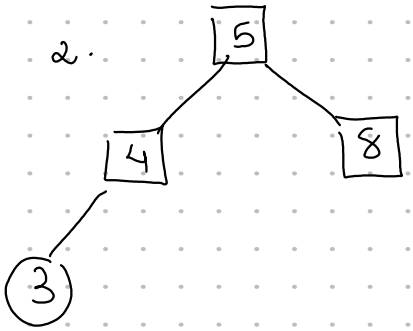
7 - 5

1.



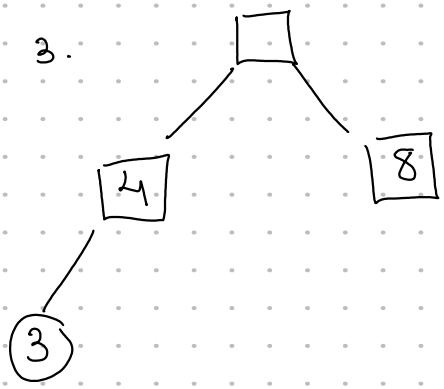
sobe o antecessor

2.

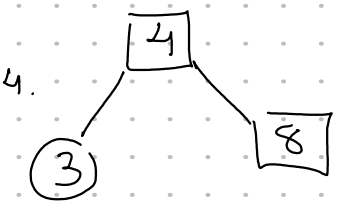


remove 5

3.



4.



troca cor do 8

5.