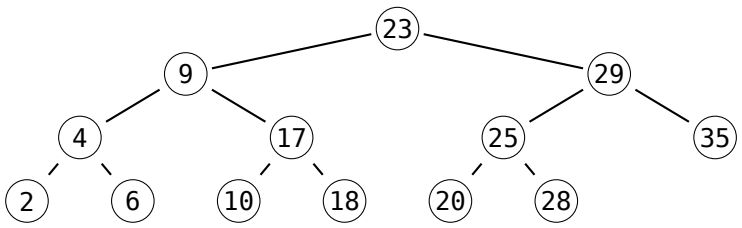
Estruturas de Informação

aula 17: A árvore 2-3

1 Introdução

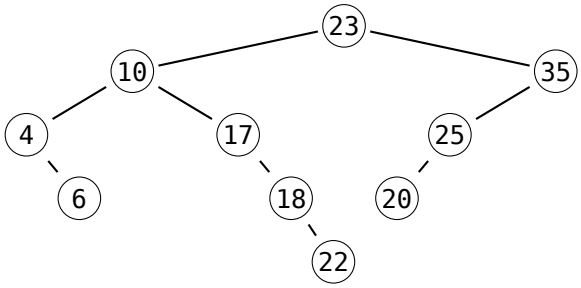
Hoje nós vamos tratar a questão da eficiência das árvores binárias de busca.

Quer dizer, nós sempre podemos construir a árvore binária de busca como uma árvore completa — (*onde todos os níveis estão completos, com a possível exceção do último*)



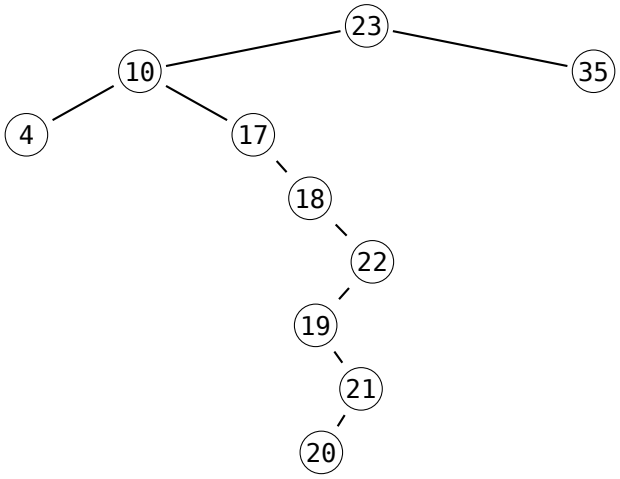
E daí, as operações de busca, inserção e remoção vão executar em tempo $O(\log n)$ — (*porque?*)

Mas o problema é que depois de inserir e remover alguns poucos elementos, a árvore pode ficar bem incompleta



— (*essa árvore é o resultado da remoção do* `9`*,* `29`*,* `28` *e* `2`*, e a inserção do* `22`)

E mais outras poucas remoções e inserções podem deixar a árvore assim



Aqui, a coisa já parece mais uma lista encadeada do que uma árvore binária de busca.

E nesse tipo de situação, as operações de busca, inserção e remoção vão executar em tempo $O(n)$.

2 A árvore 2-3

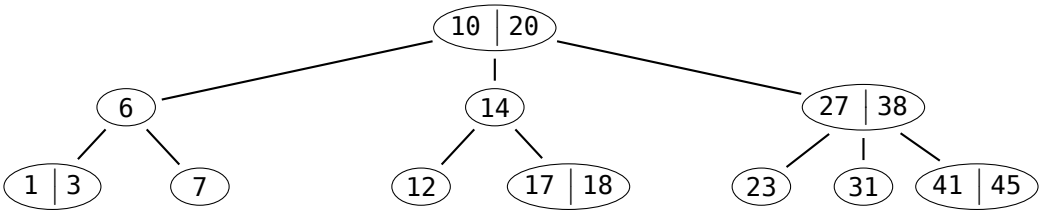
Mas esse problema pode ser resolvido com uma pequena trapaça — (*ou uma pequena esperteza, depende de como se olha ...*)

Quer dizer, a ideia é permitir que os elementos da árvore armazenem um ou dois valores.

E isso significa que agora os elementos vão ter 2 ou 3 filhos

— (*é daí que vem o nome árvore 2-3*)

Abaixo nós temos um primeiro exemplo



Como no caso das árvores binárias de busca, nós sempre podemos construir a árvore 2-3 como uma árvore completa — (*porque?*)

Só que dessa vez a árvore é realmente completa — (*porque o último nível também está completo*)

E o mais legal é que a árvore 2-3 permanece completa mesmo após as operações de inserção e remoção.

Isso signifca que as operações de busca, inserção e remoção sempre vão executar em tempo $O(\log n)$.

Legal, não é?

Mas, vamos ver como a coisa funciona na prática.

2.1 Busca na árvore 2-3

O primeiro passo é definir o registro `RegInt23`

```
struct RegInt23
    int               tipo
    int               num1, num2
    struct RegInt23   esq, meio, dir
```

Daí, nós definimos a estratégia recursiva de busca assim

→ *Para procurar o número* `k` *na árvore 2-3*

- *primeiro nós verificamos se ele está na raiz*

- *se não estiver, nós procuramos na subárvore esquerda, do meio ou direita, dependendo do tipo do registro e dos valores* `num1` *e* `num2`

E daí, a coisa fica assim

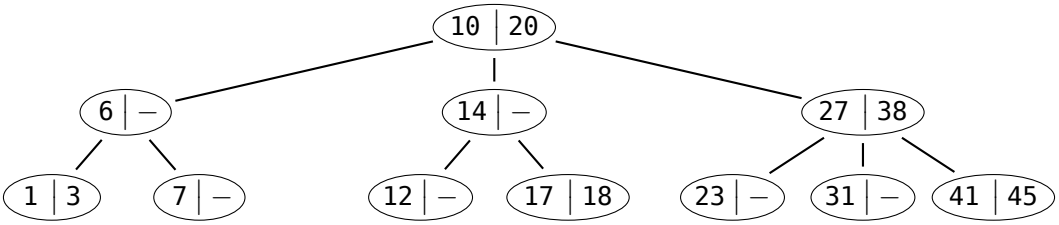
```
func buscaArv23 (q,k)
     Se (q == Nulo)                          Retorna ('Não está lá ...')
     Sse (q->num1 == k)                      Retorna ('Achei')
     Sse (q->tipo == 3  E  q->num2 == k)     Retorna ('Achei')
     Sse (q->num1 > k)                       resp = buscaArv23 (q->esq,k)
     Sse (q->tipo == 2  OU  q->num2 > k)     resp = buscaArv23 (q->meio,k)
     Senão                                   resp = buscaArv23 (q->dir,k)
     Retorna (resp)
```

3 Inserção na árvore 2-3

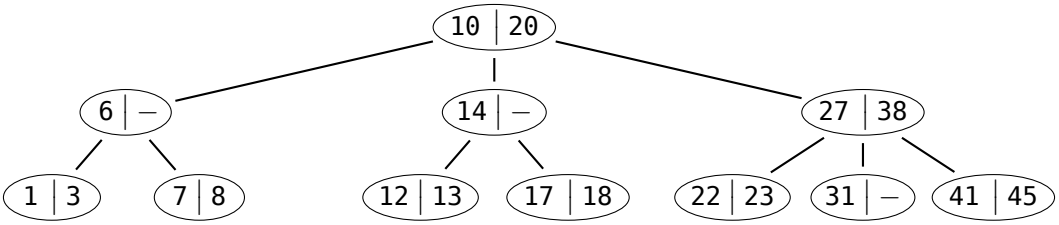
Legal.

Mas é agora que nós vamos ver a esperteza da árvore 2-3.

Para isso, nós visualizamos os elementos que possuem apenas um valor como elementos com uma posição vazia



Daí, é fácil ver que nós podemos inserir novos elementos nas posições vazias das folhas da árvore sem qualquer dificuldade



— (*aqui foram inseridos o* `8`*,o* `13` *e o* `22`)

Mas isso ainda não é a história inteira.

Porque alguma hora essas posições vazias vão acabar.

Por exemplo, imagine que agora nós queremos inserir o número `9` na árvore.

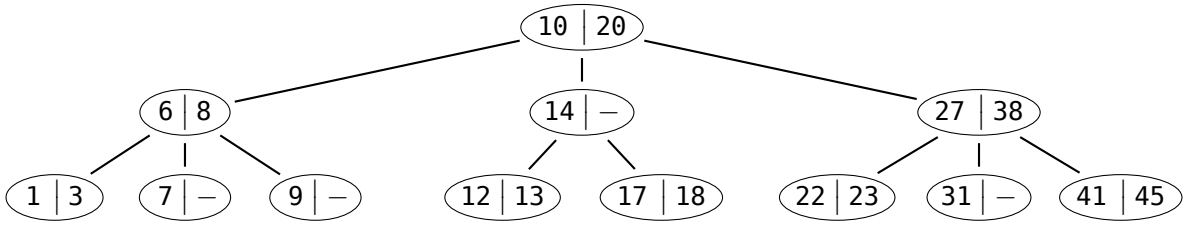
Daí, a primeira observação é que ele não vai ser colocado embaixo da folha que contém o 7 e 8.

Porque isso ia deixar a árvore incompleta.

Ao invés disso, nós colocamos o 9 dentro da folha que contém o 7 e 8.

Só que isso faz folha explodir

E a ideia é que o 8 vai ser mandado para cima, para ser inserido no pai.



Nesse caso, o 8 encontrou uma posição vazia em cima.

Mas quando o número é mandado para cima e o pai já está cheio, ocorre uma nova explosão e a coisa continua subindo.

Dessa maneira, a gente vai preenchendo as posições vazias que estão nos niveis de cima da árvore.

Legal.

Mas em algum momento essas posições vazias também vão acabar.

E daí, o que acontece é uma cascata de explosões, que chega até a raiz da árvore.

Por exemplo, imagine que agora nós queremos inserir o 2.

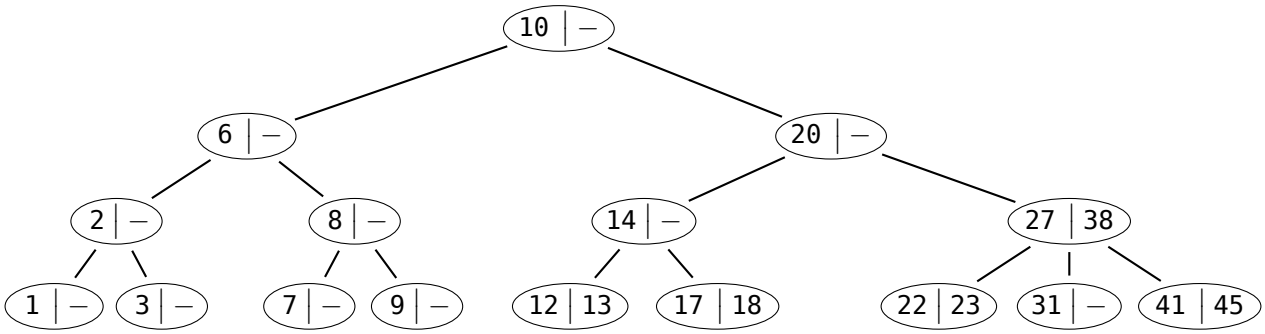
Então, a primeira explosão acontece assim

Daí, a inserção do 2 no nível de cima produz mais uma explosão

E a inserção do 6 no nível de cima produz a explosão da raiz

Só que agora, o 10 não tem mais para onde ir.

Então, ele se torna a nova raiz da árvore



Aqui a gente vê que a árvore 2-3 é uma árvore que cresce para cima.

E é por isso que ela sempre permanece completa.

Não é legal?

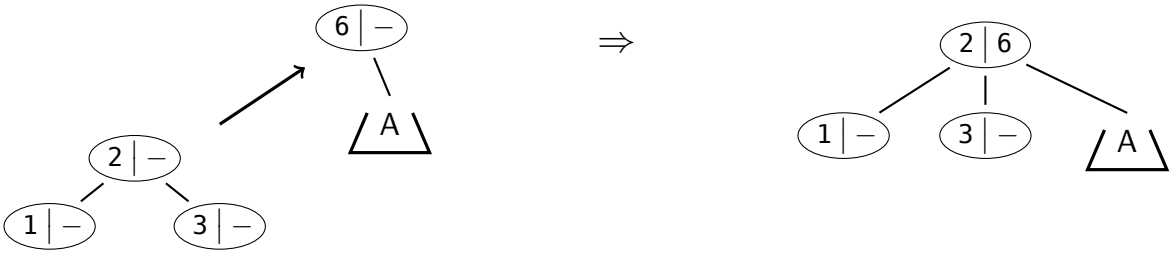
3.1 Implementação

Sim, mas agora nós precisamos escrever o algoritmo de inserção.

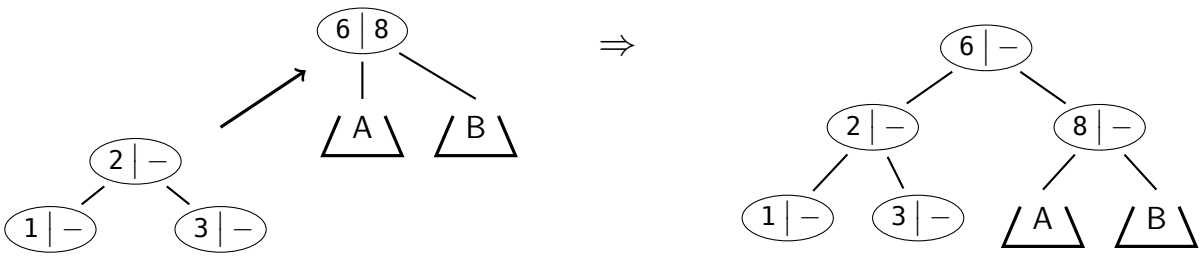
Para isso, nós vamos escrever uma função que realiza a explosão.

A ideia é que essa função vai retornar uma subárvore

Daí, quando existe uma posição vazia em cima a raiz da subárvore é colocada lá, e os seus filhos são movidos pra lá



E quando não existe posição vazia ocorre uma nova explosão, que retorna outra subárvore



Quer dizer, a função de explosão recebe

- uma árvore `a1` com raiz do tipo 2 — (*que está provocando a explosão*)
- uma árvore `a2` com raiz do tipo 3 — (*que vai ser explodida*)

E ela sempre retorna uma árvore.

```
func explosãoArv23 (a1,a2)

    r = Novo (RegInt23);   r->tipo = 2

    Se (a1->num1  <  a2->num1)                    // Caso 1: explosão pela esquerda
        r->num1 = a2->num1
        r->esq,meio,dir = a1,a2,Nulo
        a2->tipo = 2;   a2->num1 = a2->num2
        a2->esq = a2->meio;   a2->meio = a2->dir

    Sse (a1->num1  <  a2->num2)                   // Caso 2: explosão pelo meio
        r->num1 = a1->num1
        r->esq,meio,dir = a1,a2,Nulo
        a1->num1 = a2->num1;   a2->num1 = a2->num2
        aux = a1->meio;   a1->meio = a2->esq;   a2->esq = aux
        aux = a1->esq;   a1->esq = a1->meio;   a1->meio = aux

    Senão                                         // Caso 3: explosão pela direita
        r->num1 = a2->num2
        r->esq,meio,dir = a1,a2,Nulo
        a2->tipo = 2
```

Agora que nós temos a função de explosão, é fácil implementar a operação de inserção.

Quer dizer,

- primeiro a gente desce até a base da árvore — (*a chamada que vê o* `Nulo`)
- daí, a gente cria e retorna o novo registro
- e daí, cada elemento do tipo 3 que recebe um retorno diferente de `Nulo` realiza a explosão e retorna o resultado para cima

```
func inserçãoArv23 (q,k)

    Se (q == Nulo)
        r = Novo (RegInt23);   r->tipo = 2
        r->tipo = 2;   r->num1 = k
        r->esq,meio,dir = Nulo,Nulo,Nulo
        Retorna (r)
```

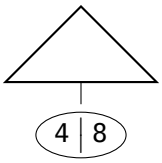
```
Sse (q->num1  >  k)                    f = inserçãoArv23 (q->esq,k)
Sse (q->tipo == 2  OU  q->num2 > k)    f = inserçãoArv23 (q->meio,k)
Senão                                  f = inserçãoArv23 (q->dir,k)

Se (f != Nulo)
      Se (q->tipo == 3)
            q = explosãoArv23 (f,q);   Retorna (q)
      Sse (q->num1 > k)
            q->num2 = q->num1;   q->num1 = f->num1
            q->dir = q->meio
            q->esq,meio = f->esq,meio
            Retorna (Nulo)
      Senão
            q->num2 = f->num1
            q->meio,dir = f->esq,meio
            Retorna (Nulo)
Senão
      Retorna (Nulo)
```

4 Remoção na árvore 2-3

A remoção na árvore 2-3 é um pouco mais complicada.

E nós vamos apenas examinar como ela é feita — (*a sua implementação vai ficar como exercício*)

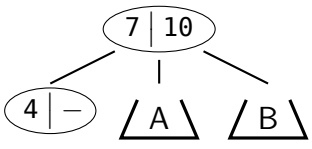
A remoção de um elemento em uma folha do tipo 3 é bem fácil

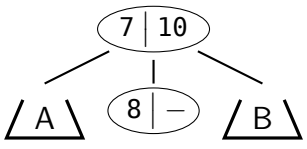


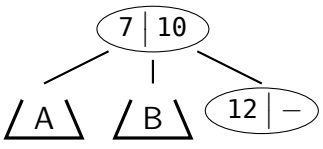
- a remoção do 8 apenas marca a folha como tipo 2
- e a remoção do 4 passa o 8 para o lugar dele

Agora imagine que nós vamos fazer a remoção em uma folha do tipo 2, cujo pai é do tipo 3.

Abaixo nós temos os diversos casos

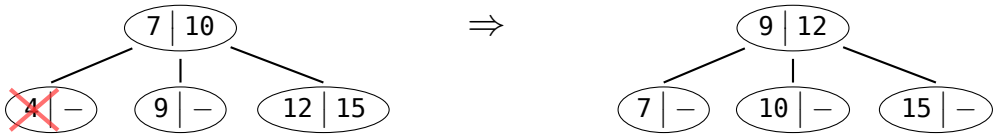






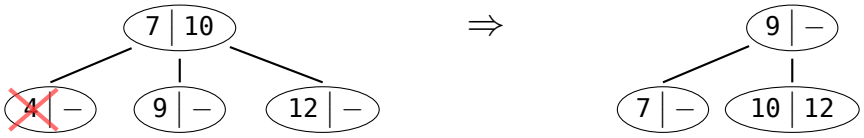
Em todos esses caso, se A ou B é do tipo 3, então a coisa pode ser resolvida trocando elementos de lugar.

Por exemplo,



E no caso em que A e B são do tipo 2, a coisa é resolvida transformando o pai em um elemento 2.

Por exemplo,



Nas duas situações, a operação não se propaga para cima na árvore.

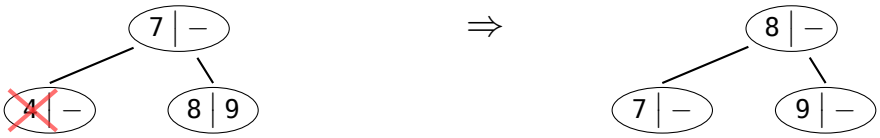
E a remoção é feita em tempo $O(1)$.

Agora imagine que nós fazemos a remoção em uma folha do tipo 2, cujo pai tem o tipo 2.

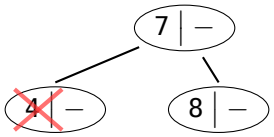
Abaixo nós temos os casos

Nos dois casos, se A é do tipo 2, então a coisa pode ser resolvido trocando os elementos de lugar.

Por exemplo,

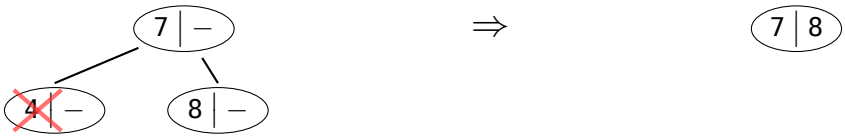


Mas, esse tipo de coisa não pode ser feito quando A também tem o tipo 2



Porque agora, só vão restar dois elementos nessa subárvore.

E a solução é fazer uma desexplosão — (*uma implosão?*)



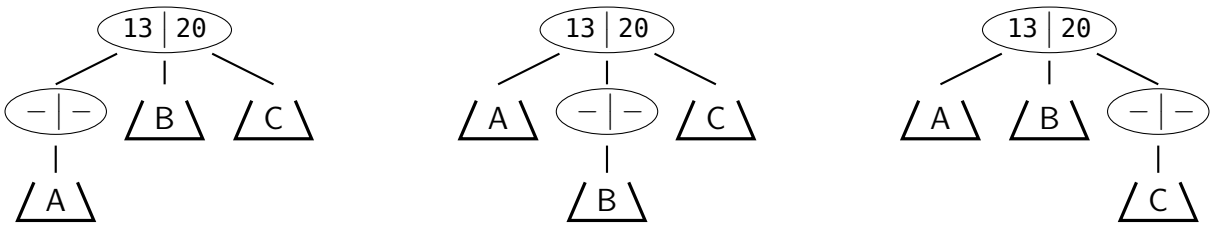
Só que, o elemento que é criado dessa maneira não pode ficar no nível do pai.

Porque senão o nível das folhas ia ficar incompleto.

Essa é a situação em que a remoção é propagada para cima.

Então agora, a gente considera o caso em que um elemento no interior da árvore fica vazio — (*como resultado de uma remoção que aconteceu mais embaixo*)

Abaixo nós temos os diversos casos, onde o pai do elemento tem tipo 3



E aqui a coisa se repete.

Porque a situação pode ser resolvida trocando elementos de lugar.

Ou então fazendo a desexplosão do pai.