AED2 - Aula 02
Árvores binárias de busca, altura e balanceamento

Queremos uma tabela de símbolos que suporta as seguintes operações:

- busca - dada uma chave k, devolva um apontador para um objeto com esta chave. Se não existir devolva “none”.
- min (max) - devolva um apontador para um objeto com a menor (maior) chave.
- predecessor (sucessor) - dada uma chave k, devolva um apontador para o objeto com a maior (menor) chave menor (maior) que k. Se não existir devolva “none”.
- percurso ordenado - devolva todos os objetos seguindo a ordem de suas chaves.
- seleção - dado um inteiro i, entre 1 e n, devolva um apontador para o objeto com a i-ésima menor chave.
- rank - dada uma chave k, devolva o número de objetos com chave menor ou igual a k.
- inserção - dada uma chave e um valor, insera um novo item com esses dados na tabela de símbolos.
- remoção - dada uma chave k, remova um item com essa chave da tabela de símbolos.

Árvores binárias de busca balanceadas são um tipo de estrutura de dados interessante para implementar uma tabela de símbolos.

- busca - O(log n).
- min (max) - O(log n).
- predecessor (sucessor) - O(log n).
- percurso ordenado - O(n).
- seleção - O(log n).
- rank - O(log n).
- inserção - O(log n).
- remoção - O(log n).

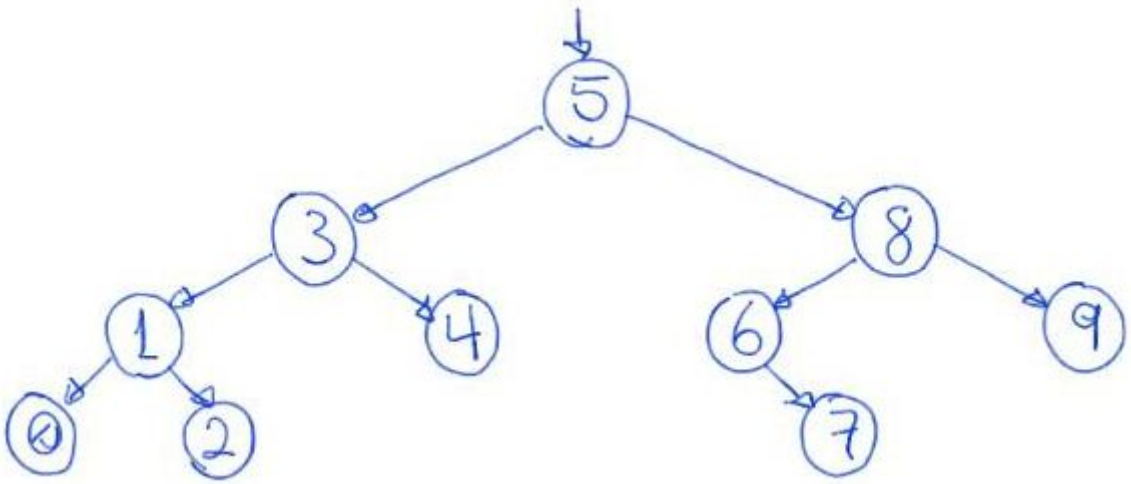
Para entendermos como essas estruturas de dados realizam tantas operações com eficiência, vamos relembrar conceitos importantes como:

- árvores binárias,
- altura,
- propriedade de busca.

Além de estudar um conceito central para o balanceamento de árvores,

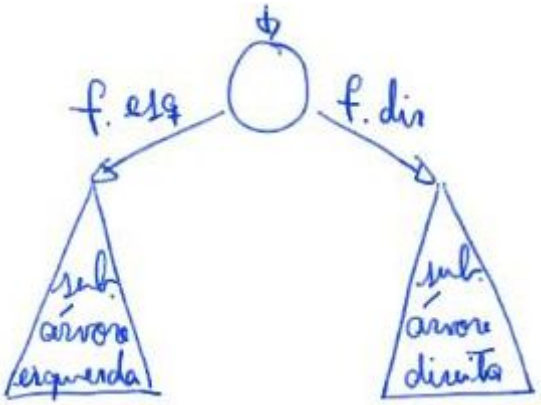
- as rotações.

Árvores binárias

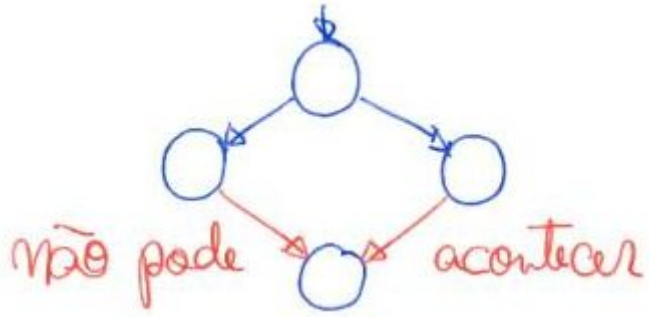


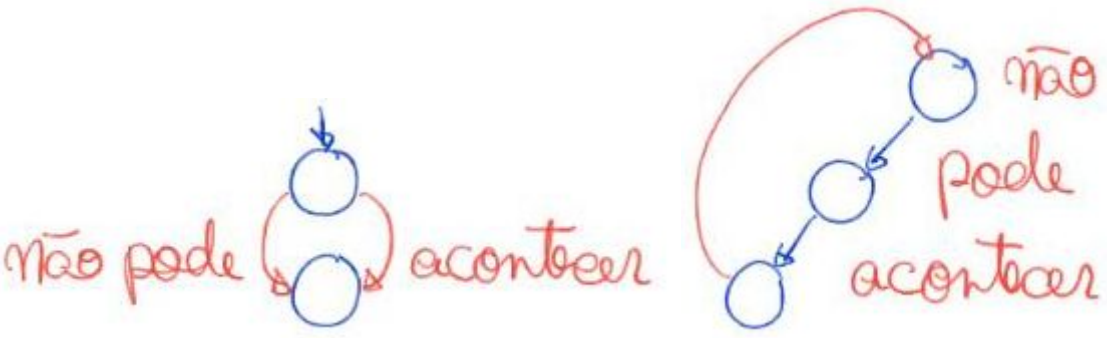
Definição de uma árvore binária:

- Temos a propriedade recursiva, segundo a qual toda árvore binária
    - é um elemento com uma subárvore esquerda e uma subárvore direita
    - ou é uma árvore vazia.



- Adicionamos à propriedade recursiva que
    - cada elemento de uma árvore tem no máximo um pai,
        - sendo que o único elemento sem pai é a raíz.
    - os filhos esquerdo e direito de cada elemento são distintos.
- É interessante verificar quais anomalias essas propriedades evitam
    - como união de caminhos e ciclos.

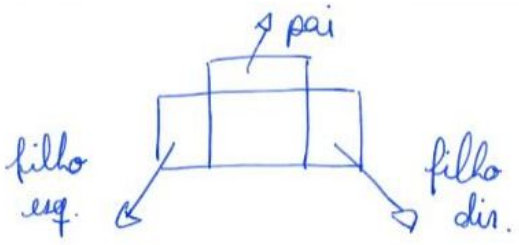




- Destacamos que,
    - a propriedade recursiva vai nos ajudar a pensar nas operações.

Cada elemento de uma árvore binária é armazenado em um nó,

- implementado como um registro que possui os campos:
    - conteúdo,
    - apontador para o filho esquerdo,
    - apontador para o filho direito,
    - apontador para o pai
        - campo opcional, corresponde ao campo anterior
            - usado em listas duplamente encadeadas,
        - e só precisamos dele para algumas operações.




```c
typedef int Cont;

typedef struct noh
{
   Cont conteudo;
   struct noh *pai; // opcional
   struct noh *esq;
   struct noh *dir;
} Noh;
```

Notação e convenções:

- Usamos o termo árvore para nos referir
    - tanto ao conjunto de elementos que compõe um árvore

    - quanto ao endereço da raiz de uma árvore.
        - Por isso, o uso da definição de tipo Arvore

```
typedef Noh *Arvore;
```

- Definimos a subárvore de um nó x, como sendo
    - x e seu conjunto de nós descendentes,
        - i.e., todos os nós para os quais existe caminho a partir de x.
    - Também podemos dizer que trata-se da árvore enraizada em x.
- Chamamos de folhas os nós da árvore que não tem filhos,
    - i.e., cujos filhos são subárvores vazias.

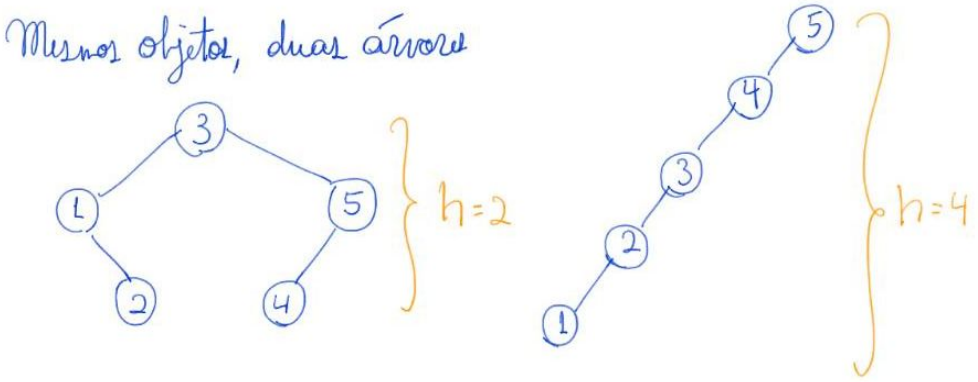
Altura de uma árvore

Definimos a altura de um nó x como sendo

- o comprimento do maior caminho de x até uma folha de sua subárvore,
    - i.e., o número de saltos entre nós em tal caminho.
- A altura (h) de uma árvore é a altura do nó raiz da mesma.

Vale notar que, árvores binárias diferentes,

- podem armazenar o mesmo conjunto de objetos. Por exemplo:

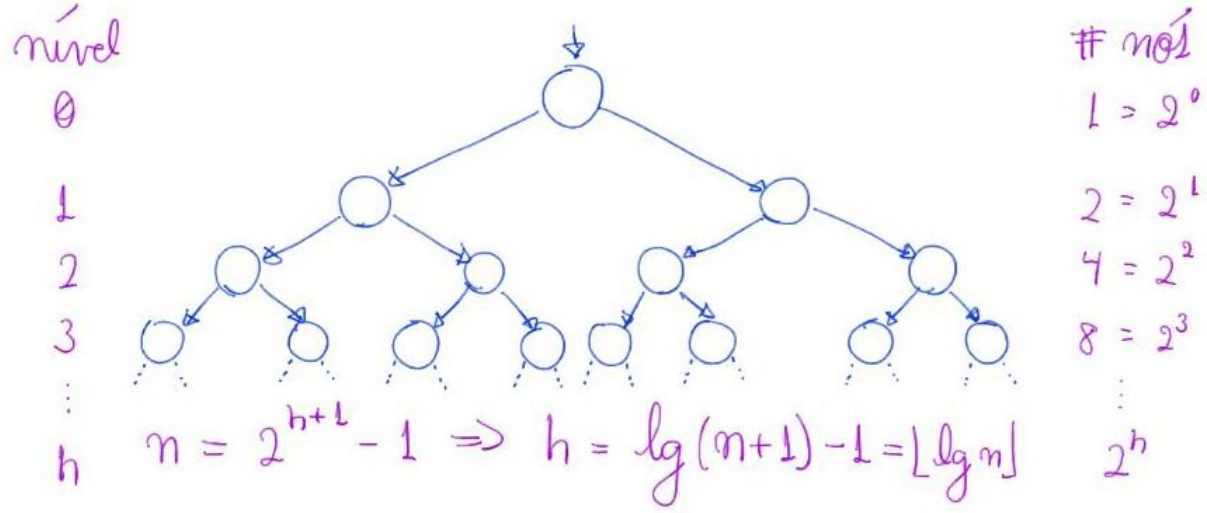


Observando as árvores anteriores, da esquerda para a direita, temos que:

- O nó com elemento 3 é raiz da primeira
    - e o nó 5 é raiz da segunda.
- Os nós 2 e 4 são folhas da primeira,
    - e o nó 1 é folha da segunda.
- A altura da primeira é 2
    - e da segunda é 4.

Vale destacar que, a altura de uma árvore binária com n nós pode variar muito:

- desde n - 1, caso seja uma lista encadeada,
- até ~= lg n, caso seja completa ou quase completa, caso em que
    - todos os níveis estão cheios, exceto talvez o último.



Para calcular a altura de uma árvore, podemos explorar sua estrutura recursiva.

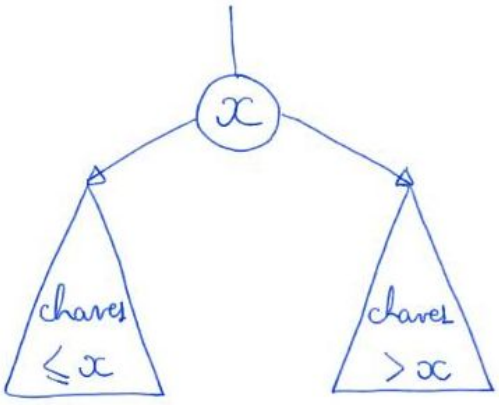
- altura - com eficiência O(n)
    - obtenha recursivamente a altura da subárvore esquerda,
    - obtenha recursivamente a altura da subárvore direita,
    - devolva 1 mais a altura da maior subárvore.

```
int altura(Arvore r)
{
   int hesq, hdir;
   if (r == NULL)
       return -1;
   hesq = altura(r->esq);
   hdir = altura(r->dir);
   if (hesq > hdir)
       return hesq + 1;
   return hdir + 1;
}
```

Árvores binárias de busca

O que diferencia uma árvore binária qualquer de uma árvore binária de busca

- é a propriedade de busca, i.e., dado um nó com chave x:
    - os elementos na subárvore esquerda tem chave <= x
    - e os objetos na subárvore direita tem chave > x.



- Observe que esta propriedade mantém os elementos ordenados na árvore.

Uma importante aplicação de árvores binárias de busca

- é na implementação de tabelas de símbolos.

```c
typedef int Chave;
typedef int Cont;

typedef struct noh
{
   Chave chave;
   Cont conteudo;
   int tam;          // opcional para operações rank e seleção
   struct noh *pai;
   struct noh *esq;
   struct noh *dir;
} Noh;

typedef Noh TS;
```

Vamos relembrar como implementar

- as operações mais importantes de uma árvore binária de busca, i.e.,
    - busca, inserção e remoção,
- além de analisar a eficiência das mesmas em função da altura (h) da árvore:

Busca(k) - com eficiência O(altura)

- comece na raiz
- repita o seguinte processo até chegar num apontador vazio
    - se a chave do nó atual = k devolva apontador para ele
    - se k < chave do nó atual desça para o filho esquerdo
    - se k > chave do nó atual desça para o filho direito

- devolva “none”

```
Noh *TSbuscaR(Arvore r, Chave chave)
{
   if (r == NULL)
       return r;
   if (r->chave == chave)
       return r;
   if (chave < r->chave)
       return TSbuscaR(r->esq, chave);
   // r->chave < chave
   return TSbuscaR(r->dir, chave);
}
```

Inserção(k) - com eficiência O(altura)

- comece na raiz
- repita o seguinte processo até chegar num apontador vazio
    - se k <= chave do nó atual desça para o filho esquerdo
    - se k > chave do nó atual desça para o filho direito
- substitua o apontador vazio pelo novo objeto.

```
Noh *novoNoh(Chave chave, Cont conteudo)
{
   Noh *novo;
   novo = (Noh *)malloc(sizeof(Noh));
   novo->chave = chave;
   novo->conteudo = conteudo;
   novo->esq = NULL;
   novo->dir = NULL;
   //    novo->pai = ??
   return novo;
}

Arvore insereR(Arvore r, Noh *novo)
{
   if (r == NULL)
   {
       novo->pai = NULL;
       return novo;
```

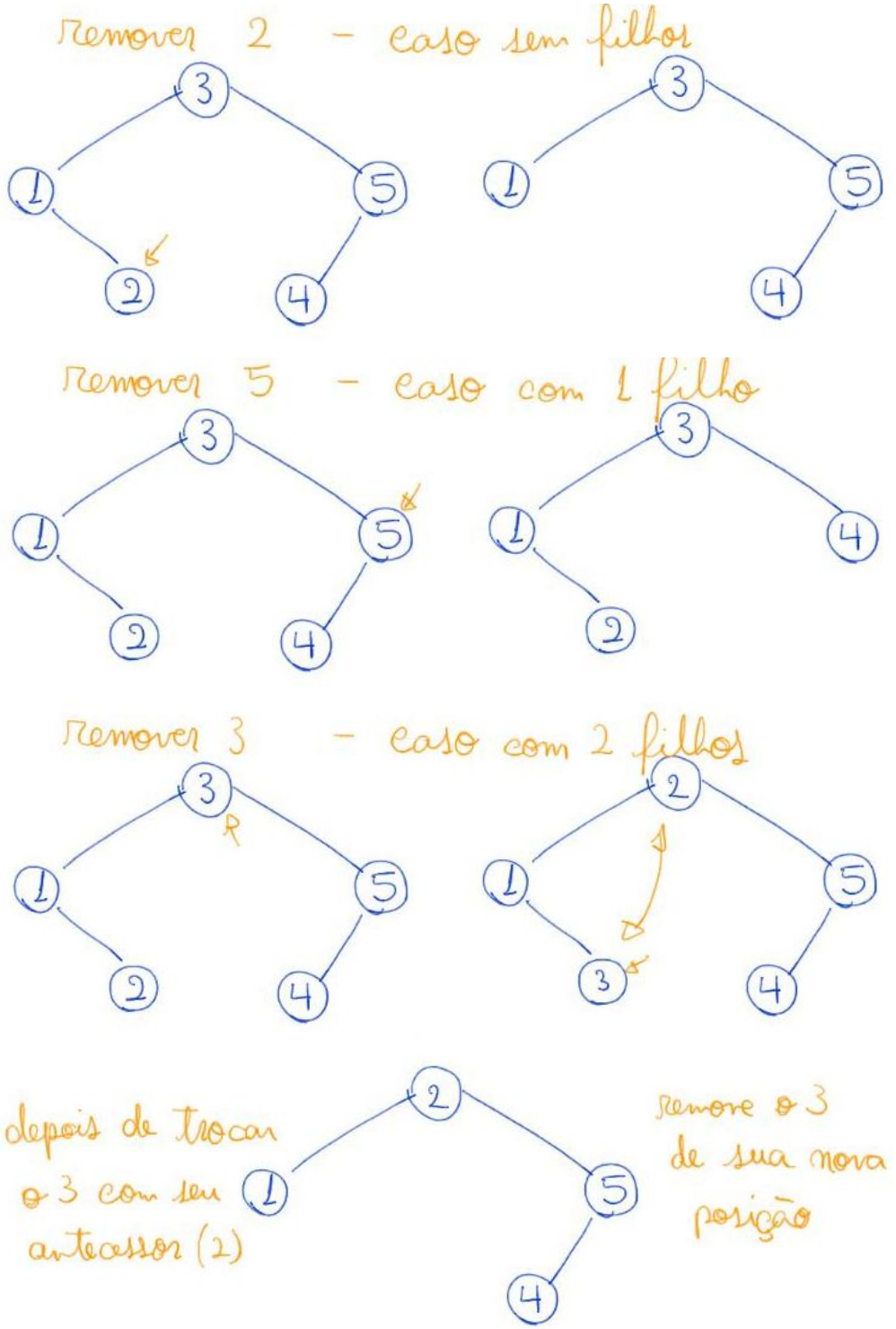
```
    }
    if (novo->chave <= r->chave)
    {
        r->esq = insereR(r->esq, novo);
        r->esq->pai = r;
    }
    else
    {
        r->dir = insereR(r->dir, novo);
        r->dir->pai = r;
    }
    return r;
}

TS *TSinserir(TS *tab, Chave chave, Cont conteudo)
{
    Noh *novo = novoNoh(chave, conteudo);
    return insereR(tab, novo);
}
```

Remoção(k) - com eficiência O(altura)

- use a busca para localizar um objeto x a ser removido.
    - se tal objeto não existe não há o que fazer.
- se x não possui filhos basta removê-lo e fazer o apontador de seu pai para ele igual a NULL.
    - se x fosse a raiz, a nova árvore é vazia.
- se x possui um filho conecte diretamente o pai de x com o filho de x, atualizando seus apontadores.
    - se x fosse a raiz, seu filho se torna a nova raiz.
- se x possui dois filhos troque x pelo objeto y que antecede x, ou seja,
    - pelo maior elemento da subárvore esquerda de x.
    - note que temporariamente a propriedade de busca é violada por x em sua nova posição.
    - então remova x de sua nova posição
        - note que essa remoção cairá num dos casos mais simples, já que na nova posição x não tem filho direito
            - caso contrário y não seria o maior elemento da subárvore esquerda.

- a seguir exemplos dos vários casos da remoção:




```
Arvore removeRaiz(Arvore alvo)
{
   Noh *aux, *p;
   if (alvo->esq == NULL && alvo->dir == NULL)
   {
       // printf("Caso 1\n");
```

```
        free(alvo);
        return NULL;
    }
    if (alvo->esq == NULL || alvo->dir == NULL)
    {
        // printf("Caso 2\n");
        if (alvo->esq == NULL)
            aux = alvo->dir;
        else // alvo->dir == NULL
            aux = alvo->esq;
        aux->pai = alvo->pai;
        free(alvo);
        return aux;
    }
    // printf("Caso 3\n");
    aux = TSmax(alvo->esq);
    printf("chave do predecessor do alvo = %d\n", aux->chave);
    alvo->chave = aux->chave;
    alvo->conteudo = aux->conteudo;
    p = aux->pai;
    if (p == alvo)
        p->esq = removeRaiz(aux);
    else // aux->pai != alvo
        p->dir = removeRaiz(aux);
    return alvo;
}

TS *TSremover(TS *tab, Chave chave)
{
    Noh *alvo, *p, *aux;
    alvo = TSbusca(tab, chave);
    if (alvo == NULL)
        return tab;
    p = alvo->pai;
    aux = removeRaiz(alvo);
```

```
    if (p == NULL)
        return aux;
    if (p->esq == alvo)
        p->esq = aux;
    if (p->dir == alvo)
        p->dir = aux;
    return tab;
}
```

Árvores binárias de busca têm eficiência proporcional à sua altura (h) para a maioria destas operações

- busca - O(h).
- min (max) - O(h).
- predecessor (sucessor) - O(h).
- percurso ordenado - O(n).
- seleção - O(h).
- rank - O(h).
- inserção - O(h).
- remoção - O(h).

Lembrando que a altura de uma árvore binária varia entre lg n e n,

- i.e., lg n <= h <= n

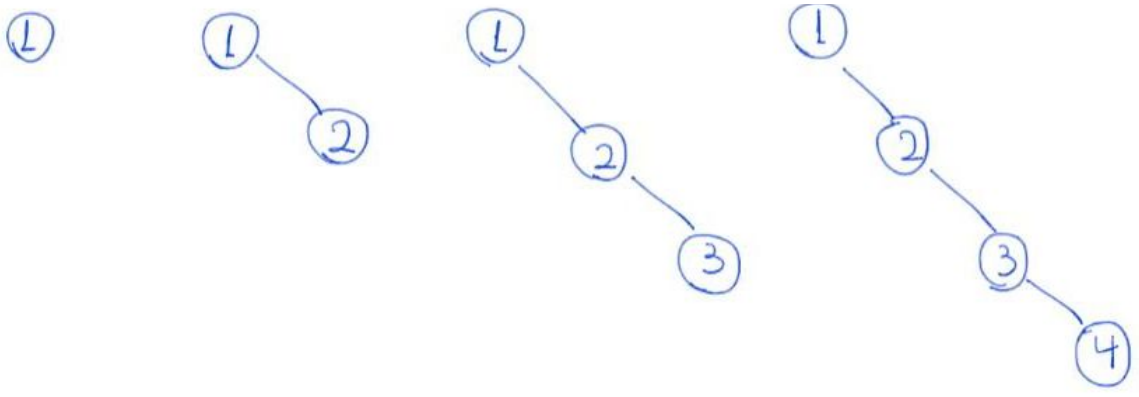
Árvores binárias de busca balanceadas

Uma árvore binária é balanceada se

- sua altura é da ordem de lg n, i.e., O(lg n).

Note que é fácil uma árvore ficar muito desbalanceada.

- Por exemplo, basta inserir os elementos em ordem.



- De fato, apenas as operações de inserção e remoção

    - podem acabar mudando o balanceamento de uma árvore.
- Por isso, serão estas operações que modificaremos nas próximas aulas.

Existem diversas estratégias para resolver o problema do balanceamento

- e estas dão origem a diferentes árvores.
- Ex: árvores AVL, árvores rubro-negras, splay-trees, árvores 2-3, árvores B.

Várias bibliotecas possuem implementações de árvores balanceadas. Como exemplos temos:

- a classe map em C++,
- a classe TreeMap e Java.

Vamos estudar a estratégia de rotações

- e veremos árvores balanceadas baseadas nessa estratégia:
    - árvores AVL,
    - árvores rubro-negras.