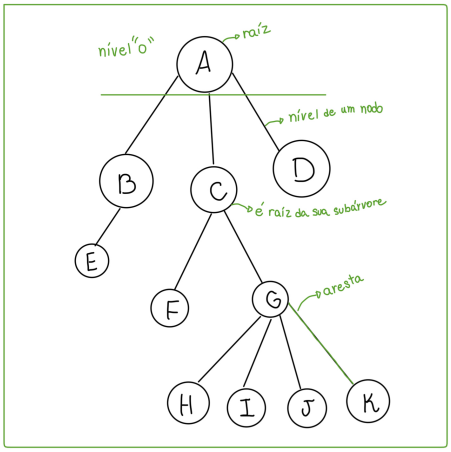
P2 árvores

▼ Conceitos básicos

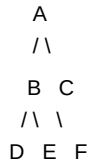
- é hierárquica e não linear
- Raiz (root)→ nodo principal (sempre nível 0)
- Nodos internos → nodos que possuem filhos (tirando a raiz)
- Folhas → nodos que não possuem filhos (tirando a raiz)
- Grau de um nodo → números de filhos (0 não tem filhos)
- Nível de um nodo → quantidade de arestas até a raiz
- Altura da árvore → maior nível que ela tem
- Irmãos → filhos do mesmo pai
- Floresta → conjunto de árvores separadas



- podem ser representadas por:
- contiguidade → arranjo e tem referencia para os filhos
- encadeamento → nodo com referencia para seus filhos e para o seu pai
- cada nodo tem uma lista de sub-árvores (filhos)

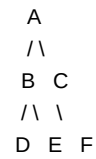
▼ Caminhamentos

- Pós fixado → primeiro o nó da esquerda, depois o nó da direita e, por último, o nó raiz



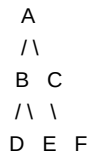
ordem - D, E, B, F, C, A

- Pré fixado → primeiro o nó raiz, depois o nó da esquerda e, por último, o nó da direita



ordem - A, B, D, E, C, F

- Em largura → por níveis crescentes de profundidade. Visita todos os nós do nível atual antes de avançar para o próximo nív



ordem - A, B, C, D, E, F

▼ Métodos gerais

- `root()` → retorna a raiz da árvore
- `parent(v)` → retorna o nodo pai de v, ocorrendo um erro se for a raiz
- `children(v)` → retorna os filhos do nodo v

- `isInternal(v)` → se o nodo for interno retorna true, se não false
- `isExternal(v)` → se o nodo for externo retorna true, se não false
- `isRoot(v)` → se o nodo for a raiz retorna true, se não false

- `size()` → retorna o número de nodos na árvore
- `isEmpty()` → testa se a árvore tem algum nodo
- `iterator()` → percorre e acessa os nós da árvore sequencialmente
- `positions()` → retorna uma coleção com todos os nodos da árvore
- `replaceElement(v,e)` → retorna o elemento armazenado em v e substitui por e

▼ Árvore genérica

- 1 nodo → 0 a N filhos
- Alocação dinâmica → estruturas encadeadas com a informação do nodo/ referencia para o pai/ referência para os filhos
- Para inserir → criar o nodo e linkar com o seu pai e coloca-lo como filho
- Para remover → remover na lista de sub-árvores do pai

▼ class Node

//atributos

1. Node father
2. Integer element
3. LinkedListOfNodes subtrees

//Métodos

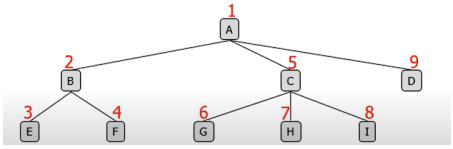
1. Node (Integer element)
2. void addSubtree(Node n)
3. boolean removeSubtree(Node n)
4. Node getSubtree(int i)
5. int getSubtreesSize()

▼ métodos

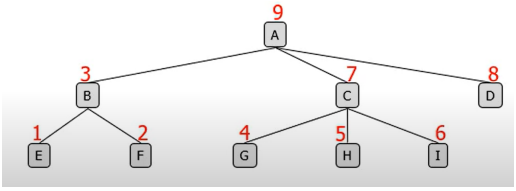
- `boolean add(Integer e, Integer father)` → insere elemento
- `Integer getRoot()` → retorna o elemento armazenado na raiz
- `Integer getParent(Integer e)` → retorna o pai do elemento e
- `boolean removeBranch(Integer e)` → remove o elemento e e seus filhos
- `boolean contains(Integer e)` → retorna true se a árvore contém o elemento e
- `boolean isInternal(Integer e)` → retorna true se o elemento está armazenado em um nodo interno
- `boolean isExternal(Integer e)` → retorna true se o elemento está armazenado em um nodo externo
- `boolean isRoot(Integer e)` → retorna true se o elemento e está armazenado na raiz
- `boolean isEmpty()` → retorna true se a árvore está vazia
- `int size()` → retorna o número de elementos armazenados na árvore
- `void clear()` → remove todos os elementos da árvore
- `LinkedListOfInteger positionsPre()` → retorna uma lista com todos os elementos da árvore na ordem pré-fixada
- `LinkedListOfInteger positionsPos()` → retorna uma lista com todos os elementos da árvore na ordem pos-fixada
- `LinkedListOfInteger positionsWidth()` → retorna uma lista com todos os elementos da árvore com um caminhamento em lar

▼ caminhamentos

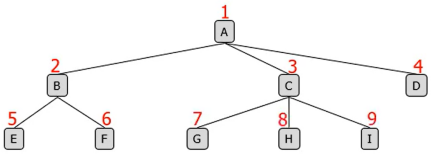
- Pré-ordem → visita a raiz e percorre cada um dos filhos desse nó na ordem em que aparecem



Pós-ordem → primeiro visita os filhos e depois a raiz (recursivo)



Largura → visita por níveis começando pelo 0 (não recursivo - usa fila)



▼ Árvore binária

- cada nodo tem no máximo 2 filhos (grau de cada nodo é ≤2)
- se tem só um filho tem que especificar se é da esquerda ou da direita
- própria → só tem nodo com 2 filhos ou nodo folha
- terão → informação do nodo/ referencia para a sub-árvore da esquerda e da direita e o pai

  implementação → estrutura encadeada (alocação dinâmica)
- transformar para binária →

1. não desliga apenas o primeiro filho
2. mesmo nível p/ direita
3. nível maior p/ esquerda

▼ class Node

//atributos

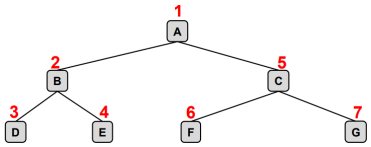
1. int element
2. Node father
3. Node left
4. Node right

//Métodos

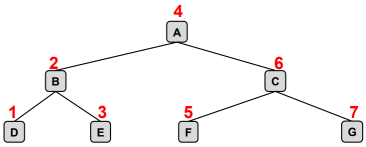
1. Node (int e)
2. father = null
3. left = null
4. right = null
5. element = e

▼ caminhamento

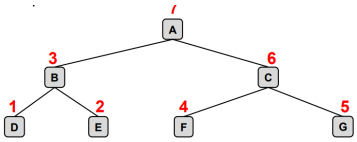
- Pré-fixado → bolinha na esquerda/ nodo antes dos descendentes



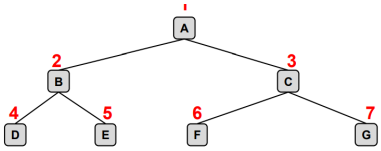
- Em-ordem (central)→ bolinha em baixo/ sub-árvore da esquerda/ raiz / sub-árvore da direita



- Pós-fixado → bolinha na direita / nodo depois dos descendes

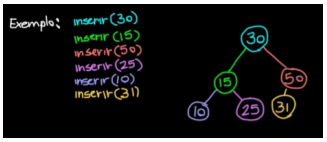


- Largura → por níveis / começa pelo nível 0 (raiz)/ da esquerda para direita



▼ ABP

- árvore binária que possui regras quanto a organização de seus elementos e é recursivo
- O(log n)= balanceada / O(n)= não balanceada
- deve ser possível comparar os elementos
- mesma regra dos nodos da árvore binária
- possui os mesmos caminhamentos da binária (o central sempre vai dar os elementos na ordem correta)



- Regras inserção →

1. se já houver a raiz, insere o elemento de acordo com seu valor:
2. se for menor que a raiz insere na sub-árvore da esquerda
3. se for maior que a raiz insere na sub-árvore da direita

- Regras remoção →

→ caso 1°) o nó não tem filhos = remove o nó

→ caso 2°) o nó possui apenas um único filho = só remover o nó e coloca seu filho no lugar

→ caso 3°) o nó possui dois filhos = substituir pelo maior nó da sub-árvore da esquerda ou pelo menor nó da sub-árvore da direit

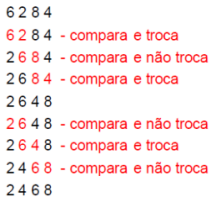
▼ AVL

- Árvore balanceada
- faz rotações
- as operações básicas (busca, inserção, remoção) levam um tempo proporcional ao número de níveis
- quanto menor melhor, pois leva menos tempo
- detectar equilíbrio → subtrai o numero de níveis na sub-árvore da esquerda do número de níveis da direita

▼ Algoritmos de ordenação

▼ Bubble-sort

- menos eficiente- $O(n)^2$
- compara pares de elementos adjacentes
- Os pares são trocados quando estiverem fora de ordem
- Repete o algoritmo até que todos estejam ordenados



▼ Insertion-sort

- mais simples de ordenação
- Considera um elemento de cada vez, colocando-o na ordem correta em relação aos demais
- Primeiro elemento: é considerado classificado
- Segundo elemento: se for menor que o primeiro, troca
- Terceiro elemento em diante: troca para a posição anterior até que fique em ordem

▼ pesquisa sequencial

- O(n)
- não muda se estiver em ordem ou não pois ele vai percorrer igual

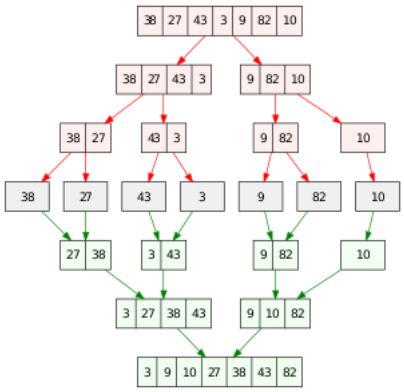
▼ pesquisa binária

- O(log n)
- Primeira iteração: low-high vale aproximadamente n

- Segunda iteração: low-high vale aproximadamente n/2
- Terceira iteração: low-high vale aproximadamente n/4

▼ merge sort

- divisão e conquista
- divide
- esquerda



▼ quick sort

- pega o do meio mais a direita
- números menores= lado esquerdo
- números maiores = lado direito

▼ Custo

- Bubble sort → O(n)
- Insertion sort → melhor- O(n) / pior- $O(n^2)$
- Merge sort → O(n log (n))
- Quick sort → pior caso $O(n^2)$
- pesquisa binaria → O(log n)
- pesquisa sequencial → O(n)