Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Sistemas e Computação
Graduação em Ciência da Computação

Árvore Binária de Busca

1. Escreva sua definição dos seguintes conceitos abaixo:
    a. Árvore binária (BT)
    b. Árvore binária de busca (BST)
    c. Árvore estritamente binária
    d. Árvore binária completa

    Resposta:

    Letra a => Uma árvore binária é uma estrutura de dados com características específicas que a definem. Ela pode ou não possuir uma raiz; se não houver uma raiz, a árvore é chamada de árvore nula. Em uma árvore binária, cada nó pode ter até dois filhos, com seu grau variando entre 0 e 2, sendo 0 o mínimo e 2 o máximo. Nós que possuem grau 0 são chamados de folhas. Além disso, há sempre apenas um caminho que conecta a raiz a qualquer outro nó da árvore.

    Letra b => Uma árvore binária de busca (BST) segue a definição básica de uma árvore binária, mas com restrições adicionais. Para cada nó, exceto as folhas, todos os nós em sua subárvore esquerda possuem valores menores que o do nó, enquanto todos os nós em sua subárvore direita possuem valores maiores. Se essa condição for violada, a árvore deixa de ser uma BST.

    Letra c => Uma árvore estritamente binária segue a definição básica de uma árvore binária, mas com restrições adicionais: cada nó possui grau 0 ou grau 2, o que significa que não há nós internos com grau 1. Em outras palavras, em uma árvore estritamente binária, cada nó ou é uma folha, sem filhos, ou possui exatamente dois filhos, um à esquerda e outro à direita. Vale ressaltar que essa definição se baseia na estrutura de uma árvore binária (BT), portanto, as restrições de uma árvore binária de busca (BST) não se aplicam aqui.

    Letra d => Uma árvore binária completa segue a definição básica de uma árvore binária, mas com a restrição de que todos nós, exceto as folhas devem possuir grau dois, o que diferencia de uma árvore estritamente binária é que todos os seus nós devem estar no mesmo nível.

2. Qual o número máximo de folhas em uma árvore estritamente binária com altura h? Justifique sua resposta.

    Em uma árvore binária estritamente binária (ou completa) com altura h, o número máximo de folhas é 2^h. Para entender isso, é importante primeiro compreender a estrutura de uma árvore binária completa. Nessa árvore, todos os níveis, exceto possivelmente o último, estão completamente preenchidos, e todos os nós no último nível estão alinhados à esquerda.

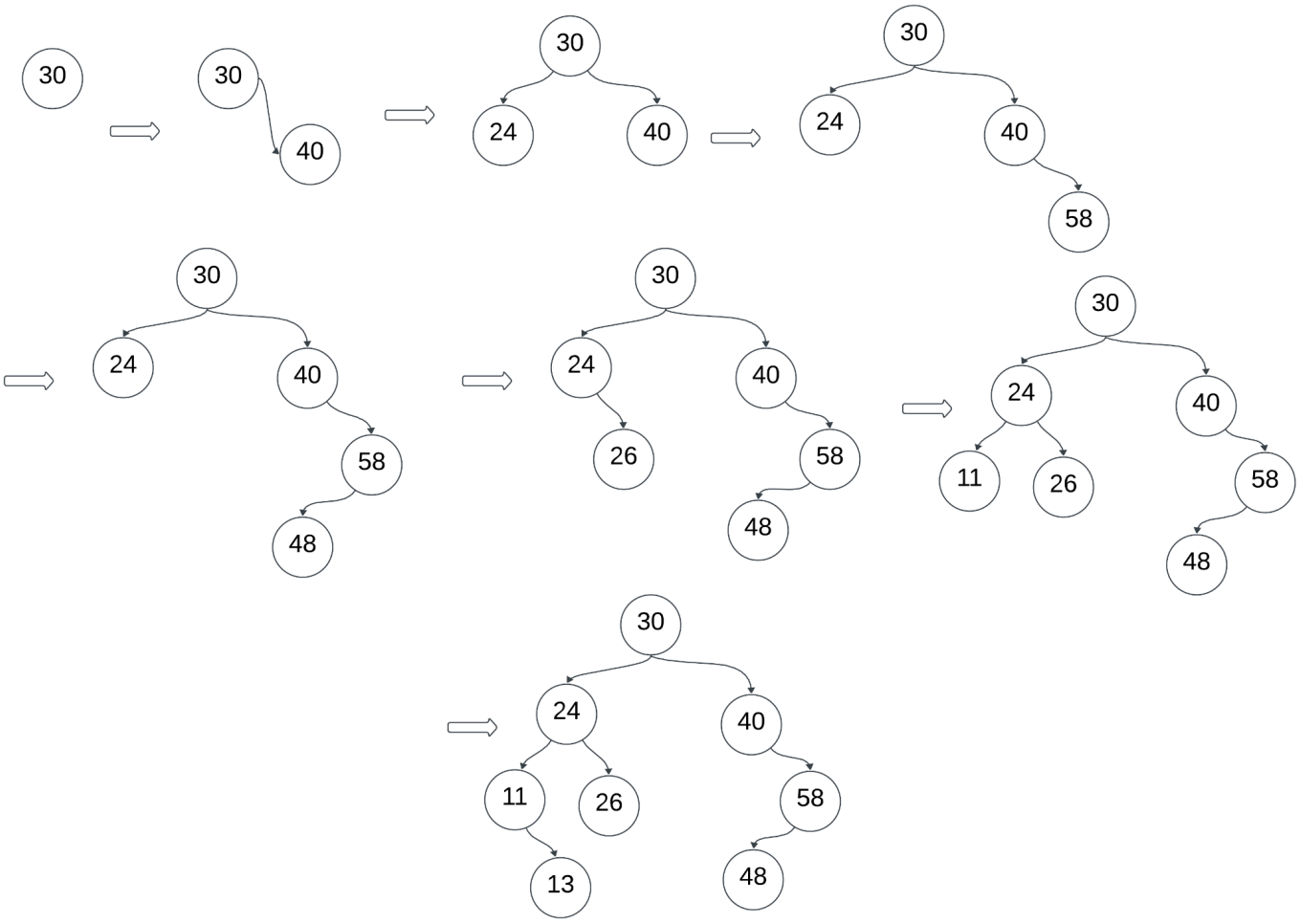
    No nível 0 (a raiz), há 1 nó. No nível 1, há 2 nós, e assim por diante. Em geral, o número de nós em um nível k é 2^k. Portanto, o total de nós em uma árvore binária completa até o nível h−1 (ou seja,

todos os níveis completos antes do nível h) é a soma de uma progressão geométrica que resulta em 2^h - 1 nós.

O nível hhh é o nível mais baixo da árvore e é o nível onde as folhas estão localizadas. Em uma árvore binária completa, o número de nós no nível h pode atingir até 2^h. Como todos os nós nesse nível são folhas, o número máximo de folhas na árvore é o número de nós no nível h, que é 2^h.

Portanto, em uma árvore binária estritamente binária com altura h, o número máximo de folhas é 2^h, o que ocorre quando a árvore está completamente preenchida até o último nível.

3. Insira os números 30,40,24,58,48,26,11,13 nesta ordem em uma árvore binária de busca inicialmente vazia e mostre a árvore resultante (após a inserção de todos os elementos).
   Resposta



4. Considere o código de uma BST abaixo. Utilizando recursão (no próprio método ou em algum método auxiliar), complete a implementação do método descendingOrder(), que retorna um array contendo as chaves da árvore ordenadas em valor decrescente. Faça a análise de seu algoritmo. Obs: sua implementação NÃO pode originar um array ordenado e depois inverter!

```
public class BSTNode<T extends
                Comparable<T> > {
      protected T data;
      protected BSTNode<T> left;
      protected BSTNode<T> right;
      protected BSTNode<T> parent;
}
```

```
public class BSTImpl<T extends
       Comparable<T>> implements
         BST<T> {

      protected BSTNode<T> root;
  public T[] descendingOrder(){...}
}
```

```
public T[] descendingOrder() {
        List<T> resultList = new ArrayList<>();
        descendingOrderHelper(root, resultList);
        return resultList.toArray((T[]) new Comparable[resultList.size()]);
}

private void descendingOrderHelper(BSTNode<T> node, List<T> resultList) {
        if (node == null) { return; }
        descendingOrderHelper(node.right, resultList);
        resultList.add(node.data);
         descendingOrderHelper(node.left, resultList);
 }
```

5. Considerando a representação acima e usando recursão (no próprio método ou em um método auxiliar) escreva um método que diz se uma BST é igual a outra recebida como parâmetro. Faça a análise de seu algoritmo.
   Respondido pelo professor em sala

6. A sequência de nós visitados no caminhamento pré-fixado à esquerda (Raiz, Esq, Dir) para uma Árvore Binária de Busca é 44, 30, 12, 26, 36, 33, 92, 64, 46, 98. Qual seria a sequência de nós no caminhamento pré-fixado à direita (Raiz, Dir, Esq) para a mesma árvore?

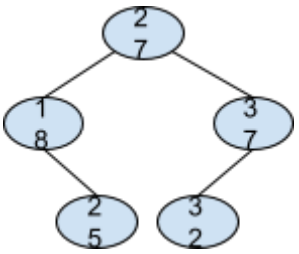
   A sequência de nós no caminhamento pré-fixado à esquerda (Raiz, Esq, Dir) dada é **44, 30, 12, 26, 36, 33, 92, 64, 46, 98**. A partir dessa sequência, podemos reconstruir a Árvore Binária de Busca (BST), em que o nó **44** é a raiz. A subárvore esquerda de **44** contém os nós menores que ele: **30, 12, 26, 36, 33**, enquanto a subárvore direita contém os nós maiores: **92, 64, 46, 98**.

   Para encontrar a sequência no caminhamento pré-fixado à direita (Raiz, Dir, Esq), seguimos a ordem: **Raiz**, depois a **subárvore direita**, e por fim a **subárvore esquerda**. Aplicando essa regra à árvore reconstruída, começamos visitando a raiz **44**, depois seguimos para a subárvore direita (**92, 98, 64, 46**), e por fim para a subárvore esquerda (**30, 36, 33, 12, 26**).

   Portanto, a sequência de nós no caminhamento pré-fixado à direita para a mesma árvore é: **44, 92, 98, 64, 46, 30, 36, 33, 12, 26**.

7. O menor ancestral comum entre dois nós de uma BST n_1 e n_2 é o ascendente de n_1 e n_2 que está mais próximo deles. Faça um algoritmo que encontre o menor ancestral comum de dois nós de uma BST e analise o tempo de seu algoritmo. Considere que os valores n_1 e n_2 pertencem à BST.
   Respondido pelo professor em sala

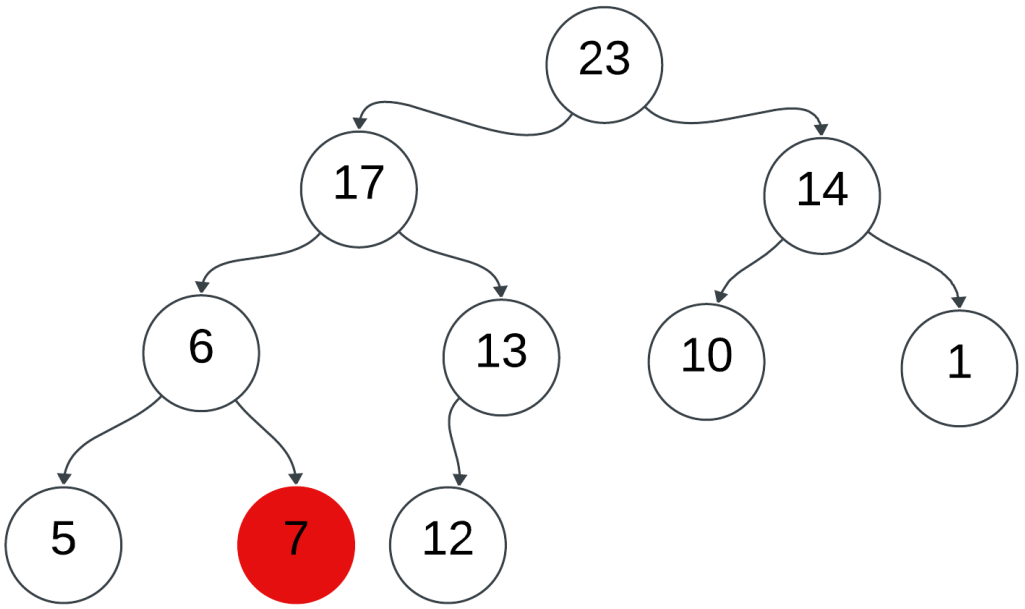
8. De quantas formas os números 18,25,27,32 e 37 poderiam ser inseridos em uma BST inicialmente vazia de forma que a árvore resultante seja a da figura abaixo?



Existem **2 formas** de inserir os números 18, 25, 27, 32 e 37 em uma BST vazia para resultar na árvore dada. Primeiro, o número 27 deve ser inserido como a raiz. Em seguida, 25 e 32 podem ser inseridos em qualquer ordem, desde que 25 fique à esquerda de 27 e 32 à direita. Após isso, 18 deve ser inserido à esquerda de 25, e 37 à direita de 32. Isso gera duas possíveis sequências de inserção.

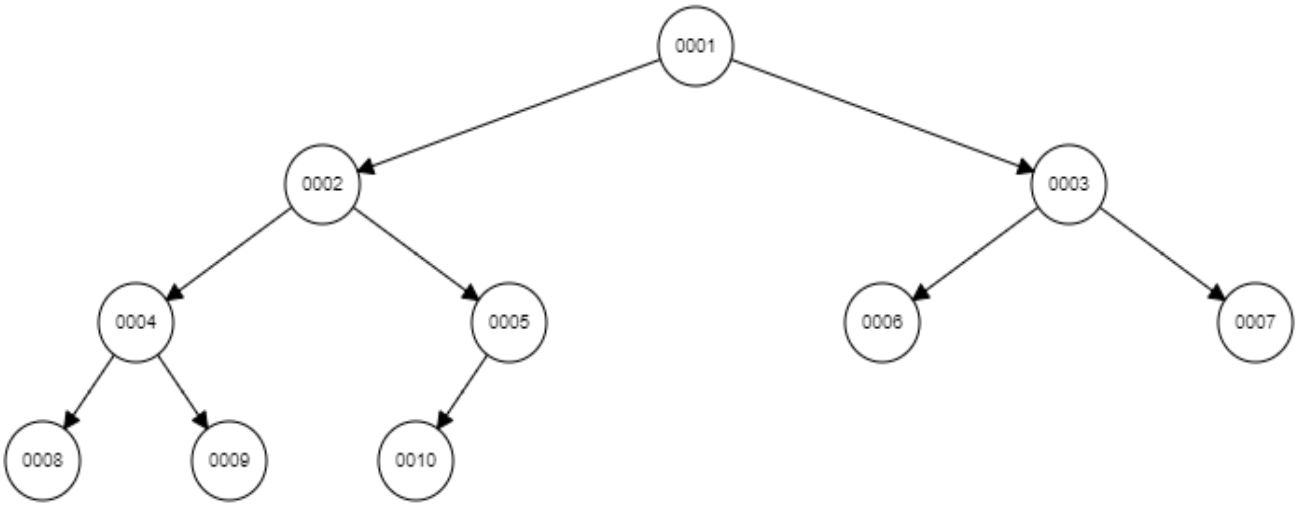
Heap Binária

1. A sequência [23,17,14,6,13,10,1,5,7,12] é uma heap? Justifique sua resposta.
   Resposta



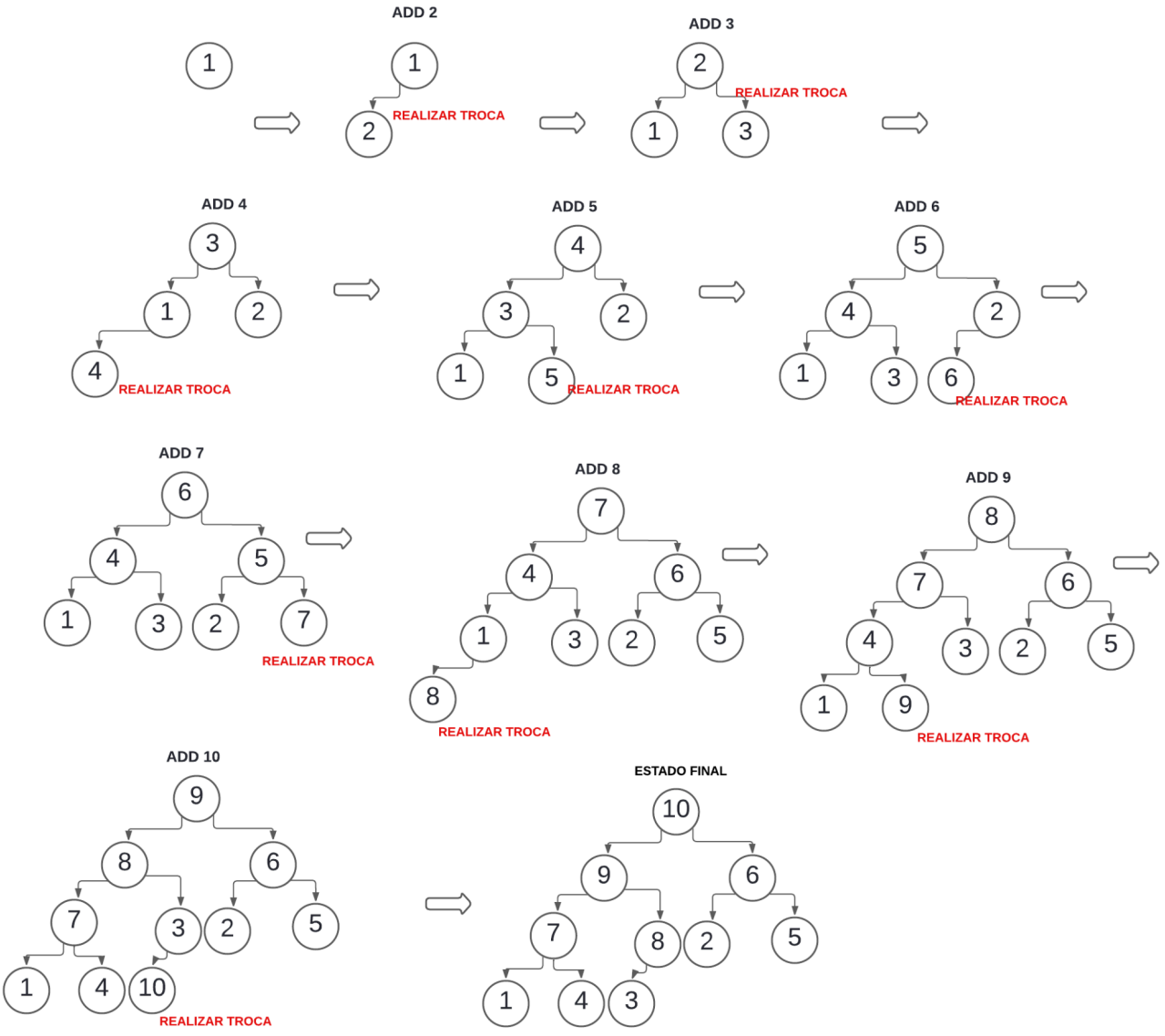
Verificando se é min heap => ela não é um min heap só pelo fato do maior nó está na raiz, logo de cara podemos descartar a possibilidade de ser um min heap.
Verificando se é max heap => para verificar se é um max heap, precisa se atentar a propriedade de que dado um nó todos os seus filhos são menores do que ele, para o nó 6 percebemos que o seu filho a direita é maior do que ele, assim ferindo a propriedade de ser um max heap, logo a sequência não é um max heap.
Conclusão => a sequência não é um heap.

2. Insira os elementos [1,2,3,4,5,6,7,8,9,10] nessa ordem em uma min heap vazia e mostre seu estado final.



3. Insira os elementos [1,2,3,4,5,6,7,8,9,10] nessa ordem em uma max heap vazia e mostre seu estado final.

Resposta



4. Critique a seguinte ideia: para transformar um vetor arbitrário em max-heap, basta colocá-lo em ordem decrescente.

   Ordenar um vetor em ordem decrescente **não é suficiente** para transformá-lo em um max-heap. A ordenação decrescente ignora a estrutura de árvore binária e não assegura que a relação pai-filho em um heap seja mantida corretamente. O algoritmo adequado para essa tarefa é o processo de heapify, que ajusta o vetor de acordo com a estrutura da heap de maneira eficiente.

5. Suponha que v[1..m] é um max-heap. Suponha que i < j e v[i] < v[j]. Se os valores de v[i] e v[j] forem trocados, v[1..m] continuará sendo um max-heap? Repita o exercício sob a hipótese v[i] > v[j].
   Resposta
   1. para v[i] < v[j]:
      Após a troca, v[i] será maior do que antes e v[j] será menor do que antes. Isso pode causar uma violação na propriedade de max-heap, porque a troca pode criar uma situação onde um pai se torna menor que um filho, violando a propriedade do max-heap.
   2. para v[i] > v[j]:
      Após a troca, v[j] será maior que v[i], e isso pode criar uma violação se v[j] estiver em uma posição que exige que seja menor do que os seus filhos, ou se v[i] for movido para uma posição onde deveria ser maior que seus filhos.

6. Ilustre a operacao heapify(3) no array [27,17,3,16,13,10,1,5,7,12,4,8,9,0].
   [27,17,3,16,13,10,1,5,7,12,4,8,9,0] - troca 3 com 10
   [27,17,10,16,13,3,1,5,7,12,4,8,9,0] - troca 3 com 9
   [27,17,10,16,13,9,1,5,7,12,4,8,3,0] - estado final

7. Implemente uma fila usando uma min-heap como estrutura sobrejacente.
   Respondido pelo professor em sala

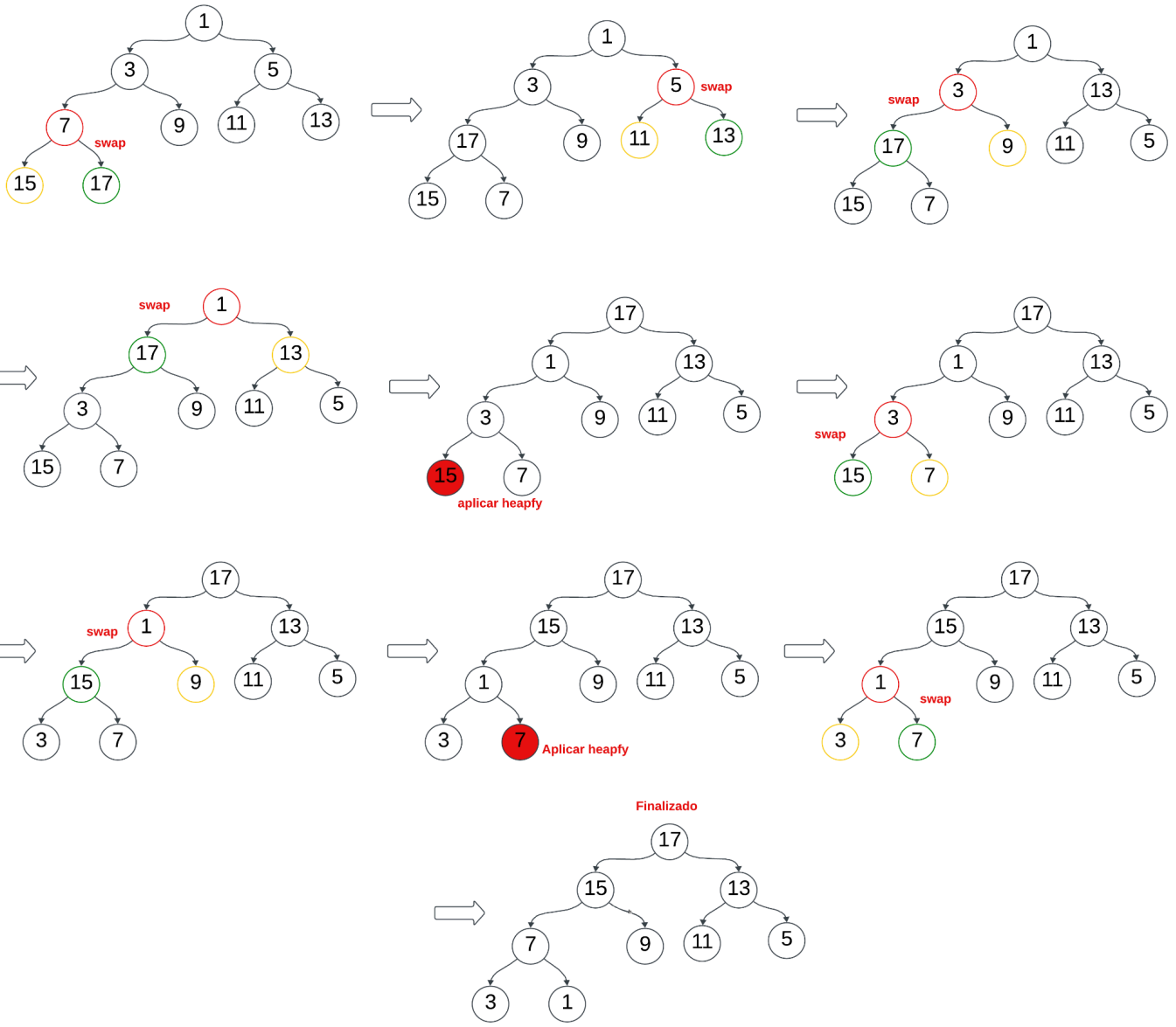
8. Considerando que a heap tem a estrutura conforme a figura abaixo e que os demais métodos dela encontram-se implementado, implemente um novo método que recebe o nível da heap e retorna um array contendo todos os elementos daquele nível.

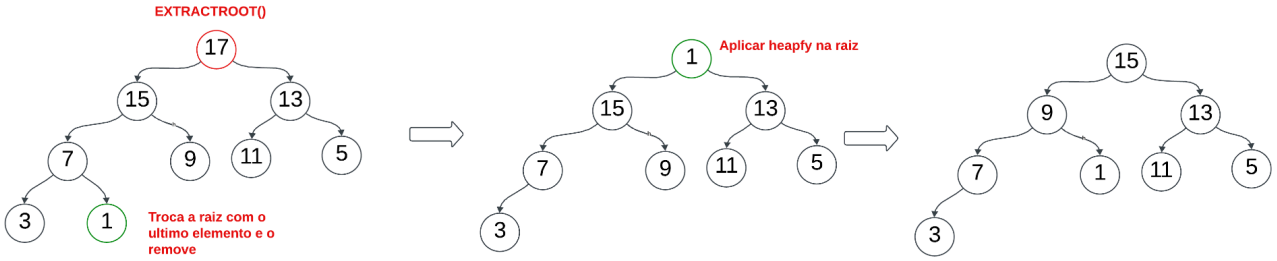
```
public class HeapImpl<T extends Comparable<T>> implements Heap<T> {
    protected T[] heap;
    protected int index = -1;
    ...
    public T[] elementsByLevel(int level){
        //implemente o método aqui
    }
}
```

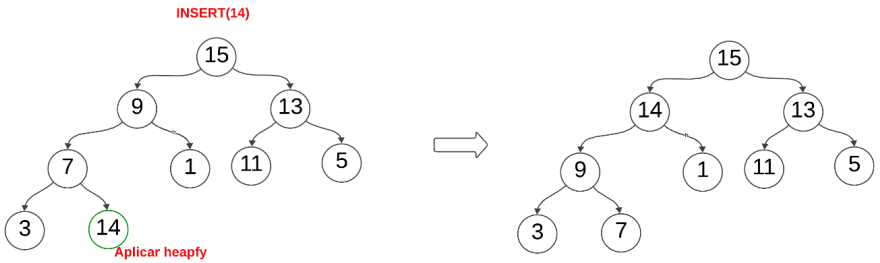
   Respondido pelo professor em sala

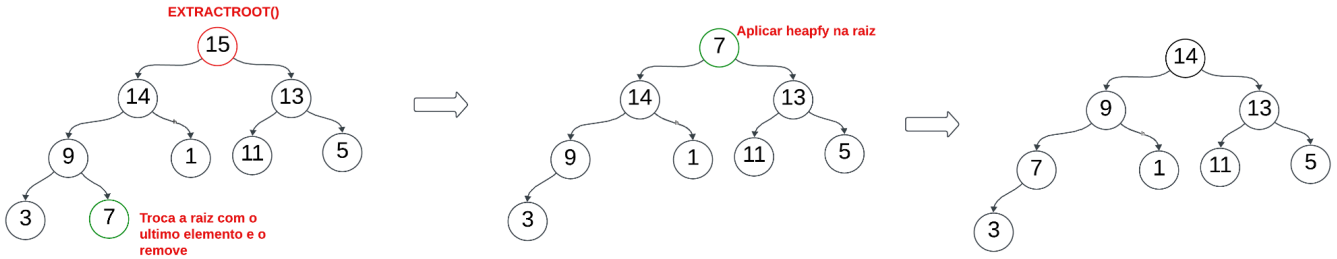
9. Aplicando o buildheap no array [1,3,5,7,9,11,13,15,17] consegue-se transformá-lo em uma max-heap. Depois disso realiza-se as operações: extractRoot(), insert(14), extractRoot(), extractRoot(). Qual seria o estado final da heap após essas operações?
   Resposta

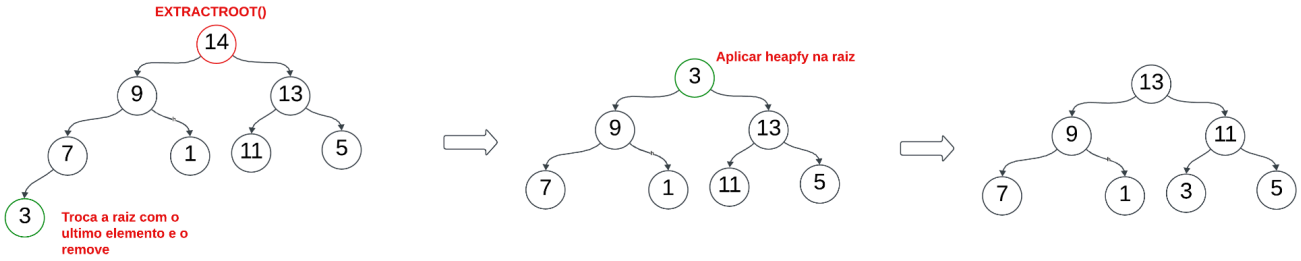
Aplicando buildheap











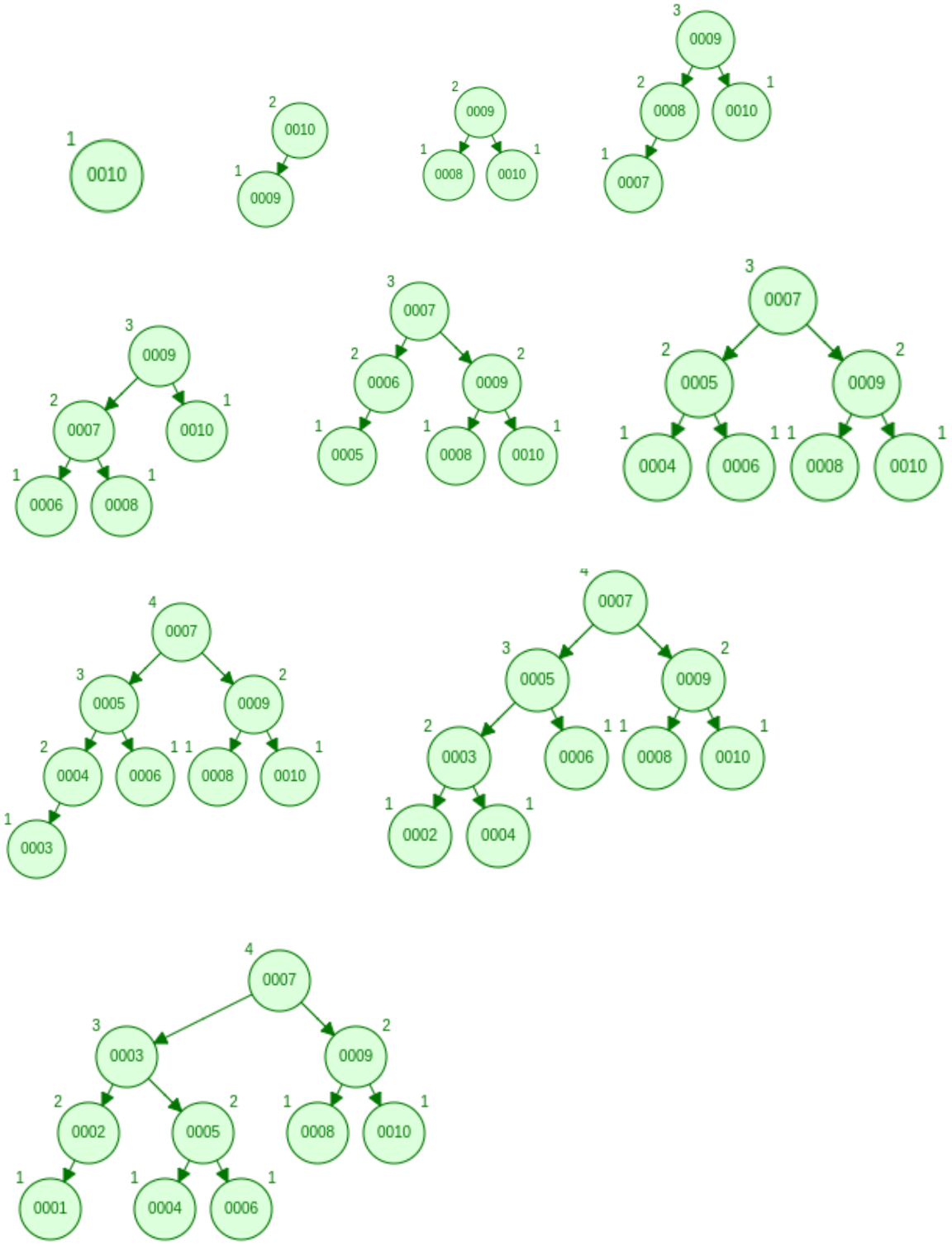
10. Considere uma min-heap inicialmente vazia onde os elementos são inseridos nela de forma a deixá-la com o último nível completo. O preenchimento se dá em um laço, cuja quantidade de iterações é determinada de forma automática, baseado no nível/altura da heap. Por exemplo, uma heap de altura 0, seria preenchida com os elementos de [1]. Uma heap de altura 1 seria preenchida com os elementos [3,2,1], uma heap com altura 2 seria preenchida com os elementos [7, … ,1] e assim por diante.
    a. Encontre o array necessário para preencher uma heap com altura h (em função de h)
    b. Encontre o número de trocas que acontecem com os elementos que são inseridos em um determinado nível h da heap (número de trocas em função de h, onde h é um determinado nível da heap)

    Respondido pelo professor em sala

AVL

1. Insira os números 10,9,8,7,6,5,4,3,2,1 nessa ordem em uma AVL inicialmente vazia e mostre o estado final da árvore.

Resposta



2. Imagine que você tem um problema de calcular a k-ésima estatística de ordem de um conjunto de dados. Faria alguma diferença se você usasse uma BST ou uma AVL para resolver esse problema? Justifique sua resposta.

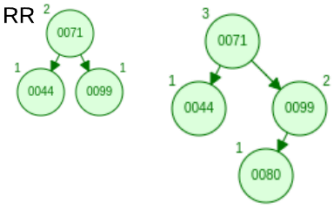
Usar uma **BST** ou uma **AVL** para calcular a k-ésima estatística de ordem faz diferença em termos de eficiência. A **BST** comum não garante que a árvore seja balanceada, o que pode resultar em uma estrutura semelhante a uma lista encadeada no pior caso (por exemplo, se os dados forem inseridos de forma ordenada). Nesse cenário, a busca pela k-ésima estatística de ordem pode ter complexidade **O(n)**, tornando o processo lento em grandes conjuntos de dados.

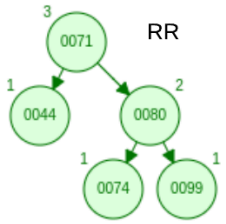
Já a **AVL** é uma árvore binária de busca balanceada, garantindo que a altura da árvore seja sempre **O(log n)**. Como resultado, a busca pela k-ésima estatística de ordem em uma AVL é mais eficiente, com complexidade **O(log n)**, independentemente da ordem de inserção dos dados. A AVL mantém esse balanceamento automaticamente durante as inserções e remoções, o que garante um desempenho consistente para a busca.

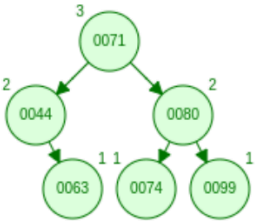
3. Partindo de uma árvore AVL vazia, realize a inserção da seguinte sequência de chaves: 99, 44, 71, 80, 74, 63, 59, 120, 98, 150. Redesenhe a árvore a cada inserção. Indique para cada rotação feita, o nó desregulado e a rotação aplicada (LL,RR,LR,RL). Em seguida remova os nós 59 e 63 mostrando as árvores resultantes a cada exclusão e a rotação aplicada (quando necessário).
   Resposta

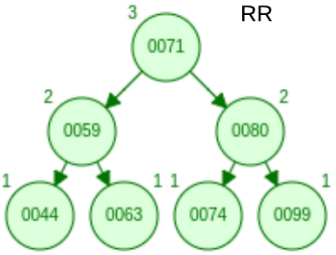
inserção

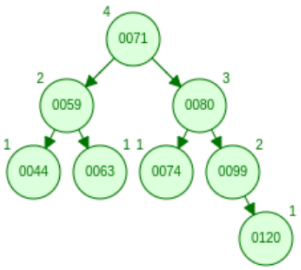

1
0099
2
0099
1
0044

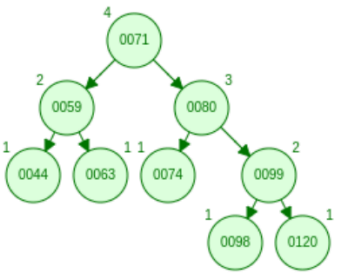

RR
2
0071
1
0044
1
0099
3
0071
1
0044
2
0099
1
0080

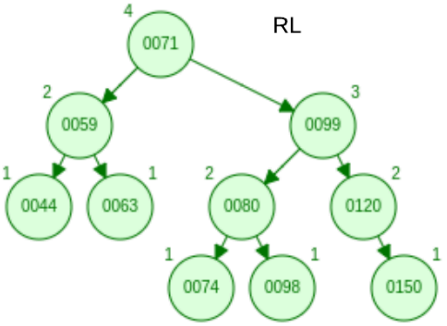

3
0071
RR
1
0044
2
0080
1
0074
1
0099


3
0071
2
0044
2
0080
1 1
0063
0074
1
0099

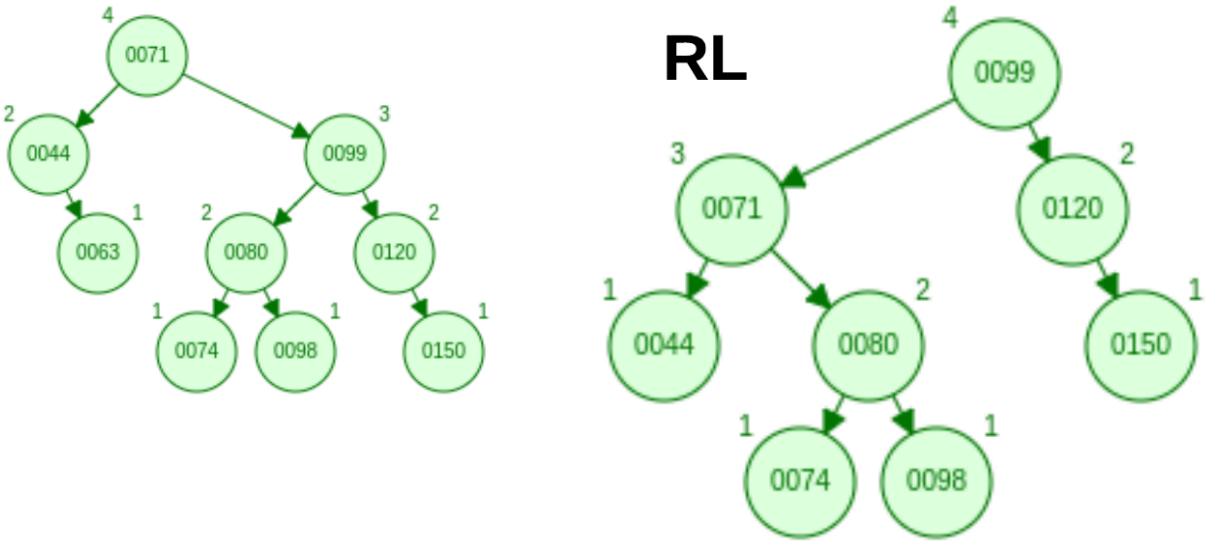

3
0071
RR
2
0059
2
0080
1
0044
1 1
0063
0074
1
0099


4
0071
2
0059
3
0080
1
0044
1 1
0063
0074
2
0099
1
0120

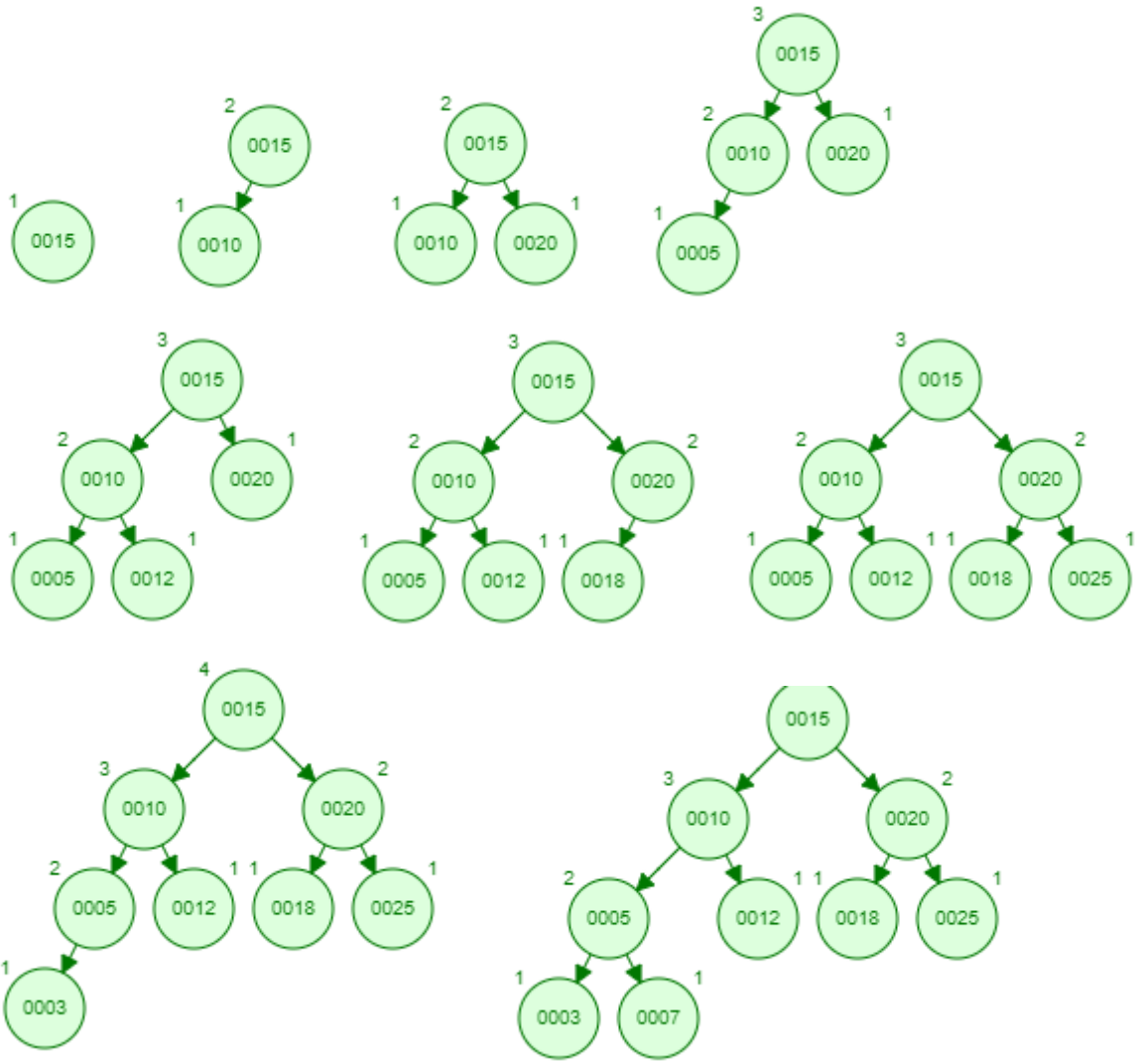

4
0071
2
0059
3
0080
1
0044
1 1
0063
0074
2
0099
1
0098
1
0120


4
0071
RL
2
0059
3
0099
1
0044
1
0063
2
0080
2
0120
1
0074
1
0098
1
0150

Remoção



4. Crie uma sequência de números inteiros, sem repetição, para preencher uma Árvore AVL inicialmente vazia, de tal forma que a árvore resultante tenha uma altura total 4, a sub-árvore à esquerda da raiz tenha uma altura total 3, a sub-árvore à direita da raiz tenha altura total 2 e nenhuma rotação seja realizada durante a inserção dos valores. Redesenhe a árvore a cada inserção.



5. Partindo de uma árvore AVL inicialmente vazia, realize a inserção da seguinte sequência de chaves: [20, 10, 15, 12, 14, 13, 19, 21, 9, 16, 17, 18, 7, 11],. Redesenhe a árvore a cada inserção. Indique para cada rotação feita, o nó desregulado e a rotação aplicada (LL, RR, LR, RL). Em seguida, remova os nós [10, 11, 16, 19, 20] mostrando as árvores resultantes a cada exclusão e a rotação aplicada (quando necessário).

   Resposta