Árvores Binárias de Busca

Busca e Travessia: Problemas Resolvidos

Prof. Edson Alves - UnB/FGA

2019

Sumário

URI 1195 – Árvore Binária de Busca

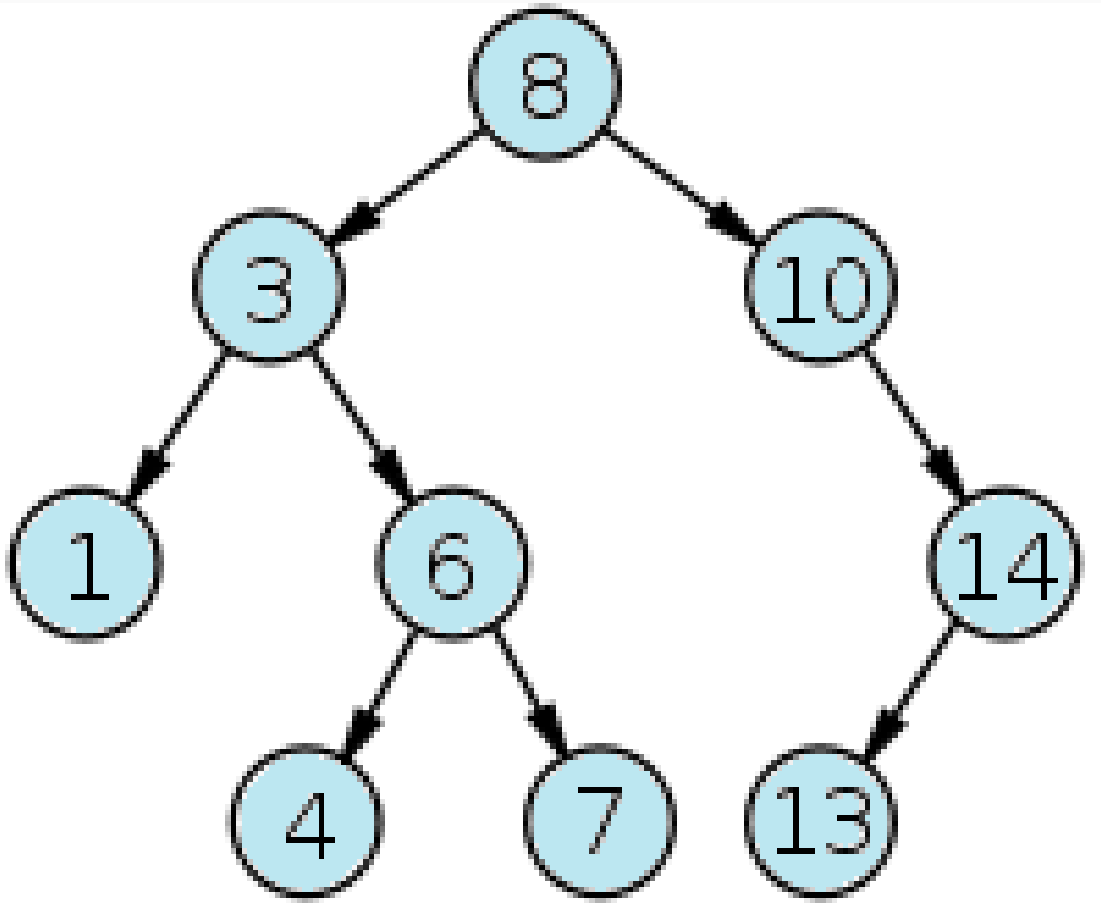
Problema

Em computação, a árvores binária de busca ou árvore binária de pesquisa é uma estrutura baseada em nós (nodos), onde todos os nós da subárvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nós da subárvore direita possuem um valor superior ao nó raiz (e assim sucessivamente). O objetivo desta árvore é estruturar os dados de forma flexível, permitindo a busca binária de um elemento qualquer da árvore.

A grande vantagem das árvores de busca binária sobre estruturas de dados convencionais é que os algoritmos de ordenação (percurso infixo) e pesquisa que as utilizam são muito eficientes.

Para este problema, você receberá vários conjuntos de números e a partir de cada um dos conjuntos, deverá construir uma árvore binária de busca. Por exemplo, a sequência de valores: 8 3 10 14 6 4 13 7 1 resulta na seguinte árvore binária de busca:

Problema



Entrada e saída

Entrada

A entrada contém vários casos de teste. A primeira linha da entrada contém um inteiro C $(C \leq 1000)$, indicando o número de casos de teste que virão a seguir. Cada caso de teste é composto por 2 linhas. A primeira linha contém um inteiro N $(1 \leq N \leq 500)$ que indica a quantidade de números que deve compor cada árvore e a segunda linha contém N inteiros distintos e não negativos, separados por um espaço em branco.

Saída

Cada linha de entrada produz 3 linhas de saída. Após construir a árvore binária de busca com os elementos de entrada, você deverá imprimir a mensagem "Case n:", onde n indica o número do caso de teste e fazer os três percursos da árvore: prefixo, infixo e posfixo, apresentando cada um deles em uma linha com uma mensagem correspondente conforme o exemplo abaixo, separando cada um dos elementos por um espaço em branco.

Exemplo de entradas e saídas

Exemplo de Entrada

```
2
3
5 2 7
9
8 3 10 14 6 4 13 7 1
```

Exemplo de Saída

```
Case 1:
Pre.: 5 2 7
In..: 2 5 7
Post: 2 7 5

Case 2:
Pre.: 8 3 1 6 4 7 10 14 13
In..: 1 3 4 6 7 8 10 13 14
Post: 1 4 7 6 3 13 14 10 8
```

Solução com complexidade $O(N^2)$

- A solução do problema tem início com a codificação de uma árvore binária de busca
- Além do construtor, é preciso implementar a rotina de inserção (a qual tem complexidade $O(S)$ no pior caso, onde S é o tamanho da árvore)
- Além disso, é preciso implementar as três travessias por profundidade notáveis (cuja complexidade de cada travessia também é $O(S)$)
- Por fim, para cada caso de teste, basta instanciar uma árvore, inserir os elementos indicados e produzir a saída usando as travessias indicadas

Solução AC com complexidade $O(N^2)$

```cpp
#include <iostream>

struct BST {
    struct Node
    {
        int info;
        Node *left, *right;
    };

    Node *root;

    BST() : root(nullptr) {}

    void inorder(const Node* node) const
    {
        if (node) {
            inorder(node->left);
            std::cout << ' ' << node->info;
            inorder(node->right);
        }
    }
```

Solução AC com complexidade $O(N^2)$

```
22
23      void preorder(const Node* node) const
24      {
25          if (node)
26          {
27              std::cout << ' ' << node->info;
28              preorder(node->left);
29              preorder(node->right);
30          }
31      }
32
33      void postorder(const Node* node) const
34      {
35          if (node)
36          {
37              postorder(node->left);
38              postorder(node->right);
39              std::cout << ' ' << node->info;
40          }
41      }
42
```

Solução AC com complexidade $O(N^2)$

```
43      void insert(int info)
44      {
45          Node **node = &root;
46
47          while (*node)
48          {
49              if ((*node)->info == info)
50                  return;
51              else if (info < (*node)->info)
52                  node = &(*node)->left;
53              else
54                  node = &(*node)->right;
55          }
56
57          *node = new Node { info, nullptr, nullptr };
58      }
59  };
60
```

Solução AC com complexidade $O(N^2)$

```
61 int main()
62 {
63     std::ios::sync_with_stdio(false);
64 
65     int C;
66     std::cin >> C;
67 
68     for (int test = 1; test <= C; ++test)
69     {
70         int N;
71         std::cin >> N;
72 
73         BST tree;
74 
75         while (N--)
76         {
77             int info;
78             std::cin >> info;
79 
80             tree.insert(info);
81         }
```

Solução AC com complexidade $O(N^2)$

```
82
83          std::cout << "Case " << test << ":\n";
84
85          std::cout << "Pre.:";
86          tree.preorder(tree.root);
87          std::cout << "\n";
88
89          std::cout << "In..:";
90          tree.inorder(tree.root);
91          std::cout << "\n";
92
93          std::cout << "Post:";
94          tree.postorder(tree.root);
95          std::cout << "\n\n";
96      }
97
98      return 0;
99  }
```

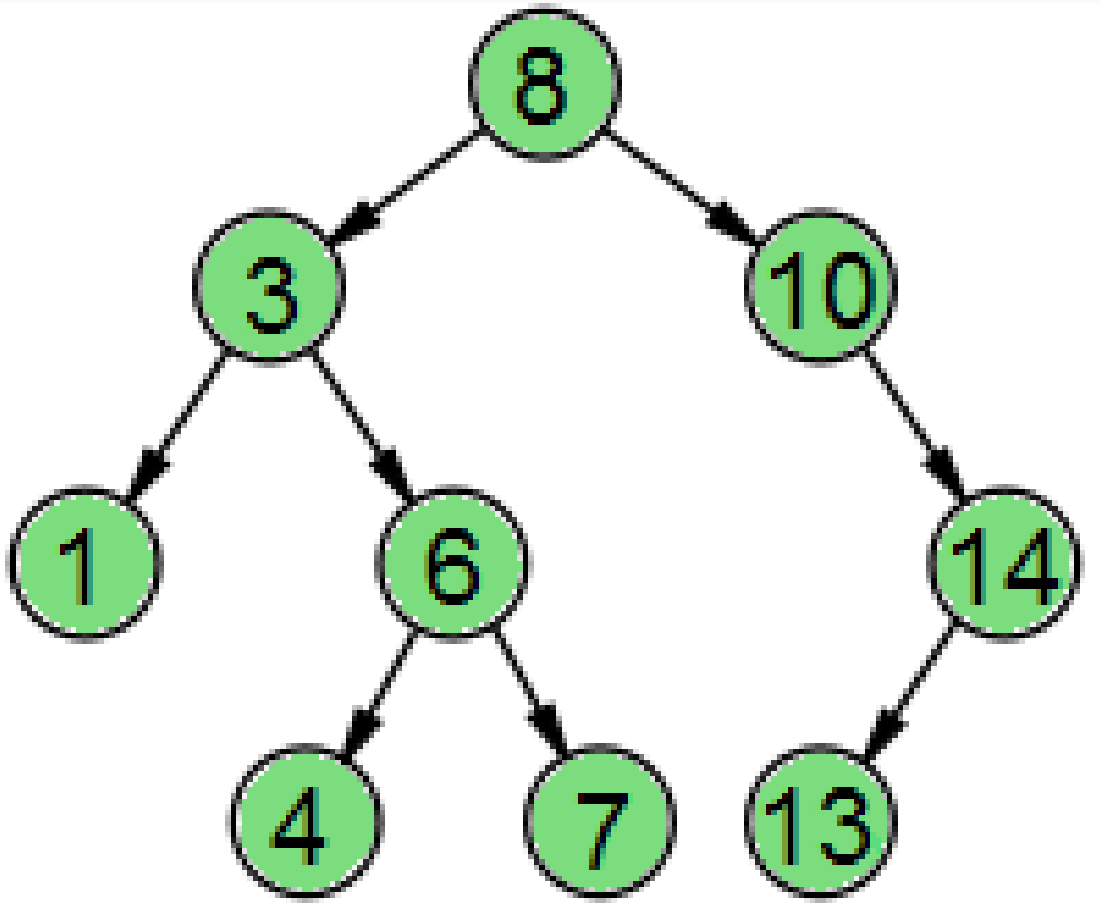
URI 1466 – Percurso em Árvore por Nível

Problema

Em uma árvore binária, o percurso por nível é um percurso denominado *breadth first search* (BFS) ou em português, busca em largura, a qual seria não-recursiva por natureza. Este percurso utiliza uma fila ao invés de pilha para armazenar os próximos 2 nodos que devem ser pesquisados (filho à esquerda e à direita). Esta é a razão pela qual você deve percorrer os nodos na ordem FIFO ao invés da ordem LIFO, obtendo desta forma a recursão.

Portanto nossa tarefa aqui, após algumas operações de inserção sobre uma árvore binária de busca (pesquisa), é imprimir o percurso por nível sobre estes nodos. Por exemplo, uma entrada com a sequência de valores inteiros: 8 3 10 14 6 4 13 7 1 resultará na seguinte árvore:

Problema



Entrada e saída

Entrada

A entrada contém vários casos de teste. A primeira linha da entrada contém um inteiro C $(C \leq 1000)$, indicando o número de casos de teste que virão a seguir. Cada caso de teste é composto por 2 linhas. A primeira linha contém um inteiro N $(1 \leq N \leq 500)$ que indica a quantidade de números que deve compor cada árvore e a segunda linha contém N inteiros distintos e não negativos, separados por um espaço em branco.

Entrada e saída

Saída

Para cada caso de teste de entrada você deverá imprimir a mensagem "Case n:", onde n indica o número do caso de teste seguido por uma linha contendo a listagem por nível dos nodos da árvore, conforme o exemplo abaixo.

Obs: Não deve haver espaço em branco após o último item de cada linha e há uma linha em branco após cada caso de teste, inclusive após o último. A árvore resultante não terá nodos repetidos e também não terá mais do que 500 níveis.

Exemplo de entradas e saídas

Exemplo de Entrada

```
2
3
5 2 7
9
8 3 10 14 6 4 13 7 1
```

Exemplo de Saída

```
Case 1:
5 2 7

Case 2:
8 3 10 1 6 14 4 7 13
```

Solução com complexidade $O(N^2)$

- A solução deste problema é semelhante à do problema anterior busca
- Novamente é necessário implementar o método construtor e a rotina de inserção (a qual tem complexidade $O(S)$ no pior caso, onde S é o tamanho da árvore)
- Além disso, é preciso implementar a travessia por largura (BFS) (cuja complexidade de cada travessia também é $O(S)$)
- Esta implementação é iterativa e requer o uso de uma fila para a organização do processamento dos nós
- Por fim, para cada caso de teste, basta instanciar uma árvore, inserir os elementos indicados e produzir a saída usando a travessia por largura

Solução com complexidade $O(N^2)$

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

struct BST {
    struct Node
    {
        int info;
        Node *left, *right;
    };

    Node *root;

    BST() : root(nullptr) {}

```

Solução com complexidade $O(N^2)$

```
18      void BFS() const
19      {
20          vector<int> xs;
21          queue<Node *> q;
22          q.push(root);
23
24          while (not q.empty()) {
25              auto node = q.front();
26              q.pop();
27
28              if (node) {
29                  xs.push_back(node->info);
30                  q.push(node->left);
31                  q.push(node->right);
32              }
33          }
34
35          for (size_t i = 0; i < xs.size(); ++i)
36              cout << xs[i] << (i + 1 == xs.size() ? '\n' : ' ');
37          cout << '\n';
38      }
```

Solução com complexidade $O(N^2)$

```
39
40      void insert(int info)
41      {
42          Node **node = &root;
43
44          while (*node)
45          {
46              if ((*node)->info == info)
47                  return;
48              else if (info < (*node)->info)
49                  node = &(*node)->left;
50              else
51                  node = &(*node)->right;
52          }
53
54          *node = new Node { info, nullptr, nullptr };
55      }
56  };
57
```

Solução com complexidade $O(N^2)$

```
58 int main()
59 {
60     ios::sync_with_stdio(false);
61 
62     int C;
63     cin >> C;
64 
65     for (int test = 1; test <= C; ++test)
66     {
67         int N;
68         cin >> N;
69 
70         BST tree;
71 
72         while (N--)
73         {
74             int info;
75             cin >> info;
76 
77             tree.insert(info);
78         }
```

Solução com complexidade $O(N^2)$

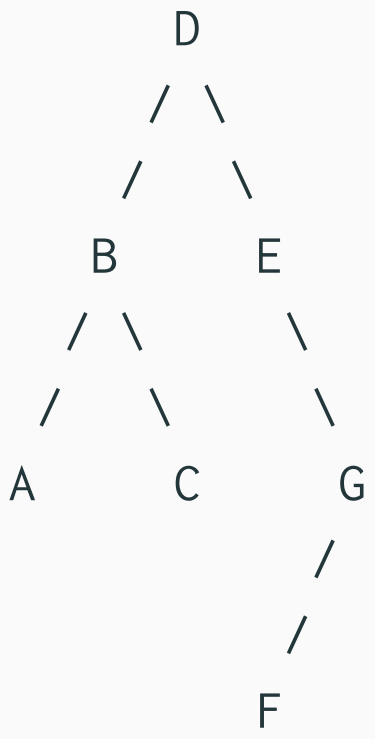
```
79
80          cout << "Case " << test << ":\n";
81          tree.BFS();
82      }
83
84      return 0;
85  }
```

URI 1191 – Recuperação da Árvore

Problema

A pequena Valentina gostava muito de brincar com árvores binárias. Seu jogo favorito era construir árvores binárias aleatórias com letras em maiúsculo nos nodos.

Este é um exemplo de uma de suas criações:



Problema

Para salvar suas árvores para uso futuro, ela escreveu duas strings para cada árvore: o percurso prefixo (raíz, sub-árvore esquerda, sub-árvore direita) e o percurso infixo (sub-árvore esquerda, raíz, sub-árvore direita).

Para o desenho acima o percurso prefixo é DBACEGF e o infixo é ABCDEFG.

Agora, anos depois, olhando para as strings, ela notou que reconstruir as árvores era realmente possível, mas só porque ela não havia usado a mesma letra duas vezes na mesma árvore.

Reconstruir a árvore a mão tornou-se chato.

Então agora ela pede que você escreva um programa que faça o trabalho por ela!

Entrada e saída

Entrada

A entrada irá conter um ou mais casos de teste. Cada caso de teste consiste em uma linha contendo duas strings representando o percurso prefixo e infixo de uma árvore binária. Ambas as strings consistem de letras maiúsculas, sem repetir. (Então elas não são maiores de 26 caracteres.)

Entrada termina com EOF (fim de arquivo).

Saída

Para cada caso de teste, imprima uma linha com o percurso posfixo (sub-árvore esquerda, sub-árvore direita, raíz).

Exemplo de entradas e saídas

Exemplo de Entrada

```
DBACEGF ABCDEFG
BCAD CBAD
```

Exemplo de Saída

```
ACBFGED
CDAB
```

Solução com complexidade $O(N^2)$

- Mais uma vez é necessário implementar o método construtor e a rotina de inserção
- A reconstrução da árvore parte de dois fatos importantes
- O primeiro deles é que a travessia em-ordem estabelece uma ordenação que permite a comparação entre os valores das informações a serem inseridas na árvore
- Esta ordenação deve ser utilizada na inserção, substituíndo a ordenação lexicográfica padrão imposta pelo operador $<$
- A ordem de inserção dos elementos na árvore é determinada pela travessia pré-ordem
- Finalizada a inserção, basta imprimir na saída a árvore usando a travessia pós-ordem

Solução com complexidade $O(N^2)$

```
1  #include <iostream>
2
3  struct BST {
4      struct Node
5      {
6          char info;
7          Node *left, *right;
8      };
9
10     Node *root;
11
12     BST() : root(nullptr) {}
13
14     void postorder(const Node* node) const
15     {
16         if (node) {
17             postorder(node->left);
18             postorder(node->right);
19             std::cout << node->info;
20         }
21     }
```

Solução com complexidade $O(N^2)$

```
22
23      void insert(char info, const int rank[])
24      {
25          Node **node = &root;
26
27          while (*node)
28          {
29              if ((*node)->info == info)
30                  return;
31              else if (rank[info - 'A'] < rank[(*node)->info - 'A'])
32                  node = &(*node)->left;
33              else
34                  node = &(*node)->right;
35          }
36
37          *node = new Node { info, nullptr, nullptr };
38      }
39  };
40
```

Solução com complexidade $O(N^2)$

```
41 int main()
42 {
43     std::string preorder, inorder;
44 
45     while (std::cin >> preorder >> inorder) {
46         int rank[30], nxt = 1;
47 
48         for (const auto& c : inorder)
49             rank[c - 'A'] = nxt++;
50 
51         BST tree;
52 
53         for (const auto& c : preorder)
54             tree.insert(c, rank);
55 
56         tree.postorder(tree.root);
57         std::cout << "\n";
58     }
59 
60     return 0;
61 }
```

Referências

1. URI 1195 – Árvore Binária de Busca
2. URI 1466 – Percurso em Árvore por Nível
3. URI 1191 – Recuperação da Árvore