22.09.2024

Tarefa 3 - Árvores Binárias

Júlia Tadeu - 2312392

Theo Canuto - 2311293

Professor Luiz Fernando Seibel

INF1010 - 3WA

1. **Objetivo**

O objetivo deste trabalho é implementar e estender a funcionalidade de uma árvore binária através da inserção por nível, além de adicionar informações complementares para melhorar a estrutura e a compreensão da árvore. As extensões e funcionalidades implementadas incluem:

Inserção por nível: Inserir as chaves fornecidas sequencialmente na árvore, de modo que os nós sejam adicionados em ordem de níveis, preenchendo primeiro os nós mais próximos da raiz.

Nível de cada nó: Atribuir o nível de profundidade de cada nó, com a raiz no nível 0, e os níveis subsequentes aumentando conforme a profundidade da árvore.

Altura de cada nó: Calcular a altura de cada nó, sendo a altura definida como a maior distância entre o nó em questão e uma folha.

Ponteiro para o nó pai: Adicionar um ponteiro que permita navegar de um nó até seu pai, facilitando a movimentação reversa na árvore binária.

Essas funcionalidades adicionais ajudarão a compreender a estrutura hierárquica da árvore de maneira mais detalhada e permitirão uma análise mais precisa da eficiência da árvore, em termos de altura e profundidade dos nós. O trabalho também exige que a árvore binária seja listada após a adição dessas informações, proporcionando uma visualização clara e informativa da estrutura da árvore e de suas relações internas.

2. **Estrutura do programa**

O programa é composto por uma estrutura de dados modificada e seis funções principais, além da função *main*. A estrutura de dados foi expandida em relação à versão anterior (Lab 3A), para incluir o nível, altura e o ponteiro para o pai de cada nó, proporcionando uma compreensão mais detalhada da hierarquia da árvore binária. Cada função desempenha um papel essencial na criação, inserção e exibição da árvore, garantindo que essas novas informações sejam calculadas e exibidas corretamente. A seguir, estão detalhadas a estrutura de dados e as funções responsáveis pela implementação dessas funcionalidades.

Estrutura de Dados:

```
typedef struct Node {
    int key, level, height;
    struct Node* left, * right, * parent;
} Node;
```

A estrutura Node define os nós da árvore binária, contendo:

- *key*: Um número inteiro que representa a chave armazenada no nó.
- *left*: Ponteiro para o nó filho à esquerda.
- *right*: Ponteiro para o nó filho à direita.
- *parent*: Um ponteiro para o nó pai, permitindo navegação reversa na árvore.
- *level*: O nível de profundidade do nó na árvore.
- *height*: A altura do nó, que representa a maior distância entre o nó e uma folha.

Função *NewNode(int key):*

- **Descrição:** Cria um novo nó da árvore com a chave fornecida, inicializando os ponteiros *left*, *right* e parent como *NULL*, e definindo os valores de *level* e *height*.
- **Parâmetros:** *int key*: A chave que será armazenada no nó.
- **Retorno:** Retorna um ponteiro para o novo nó criado.

Função *insert(Node** nodes, int key, int* index):*

- **Descrição:** Insere um nó na árvore de forma nivelada, atualizando o ponteiro do nó inserido e calculando os valores de *level* e *height* durante a inserção.
- **Parâmetros:** *Node** nodes*: Um array de ponteiros para nós, representando os níveis da árvore; *int key*: A chave a ser inserida no novo nó; *int* index*: Um ponteiro para o índice que aponta para o nó atual no qual a inserção está ocorrendo.
- **Retorno:** A função não retorna valor (void).

obs: A nova versão da função *insert* é mais completa, pois além de inserir os nós, também atualiza o ponteiro para o nó pai e armazena o nível e a altura, fornecendo informações importantes para a visualização e análise da árvore.

Função *height(Node* node):*

- **Descrição:** Calcula e armazena a altura de cada nó individualmente, além de calcular a altura geral da árvore.
- **Parâmetros:** *Node* node*: Ponteiro para o nó cujo subárvore será analisado.
- **Retorno:** Retorna um número inteiro que representa a altura da árvore.

obs: Na nova versão, a função *height* não apenas calcula a altura geral da árvore, mas também atualiza a altura de cada nó individual, facilitando o acesso direto a essa informação.

Função *printLevel(Node* root, int level)*:

- **Descrição:** Imprime os nós de um nível específico da árvore binária, incluindo informações sobre a chave, altura, nível, endereço de memória de cada nó, além de informações sobre os nós pai e filhos (com seus respectivos endereços de memória).
- **Parâmetros:** *Node* root:* Ponteiro para o nó raiz da árvore; *int level:* O nível atual da árvore a ser impresso.
- **Retorno:** A função não retorna valor (void).

Função *printTree(Node* root)*:

- **Descrição:** Imprime a árvore binária por nível, incluindo as informações de nível e altura de cada nó, além de exibir a estrutura hierárquica da árvore com mais detalhes.
- **Parâmetros:** *Node* root:* Ponteiro para o nó raiz da árvore.
- **Retorno:** A função não retorna valor (void).

Função *main()*:

- **Descrição:** Ponto de entrada do programa. Inicializa a árvore binária, insere as chaves fornecidas, calcula o nível e altura de cada nó, e imprime a árvore com as informações adicionais.
- **Parâmetros:** Não possui parâmetros.
- **Retorno:** Retorna um inteiro *(int)* que indica o status da execução do programa, onde 0 indica que o programa foi executado com sucesso.

  obs: A main da nova versão segue a mesma lógica de inicialização e inserção, mas agora exibe a árvore com mais informações, como altura e nível de cada nó, além de calcular essas métricas conforme os nós são inseridos.

3. **Soluções**

A solução para a construção e impressão da árvore binária foi desenvolvida usando listas encadeadas de nós, onde cada nó possui um ponteiro para seu filho esquerdo e direito. A árvore é montada com base na inserção de nós de forma nivelada (breadth-first), e os nós são impressos por nível. Abaixo estão os passos detalhados:

- Criação de Nós:

  A função *newNode* é responsável por criar e inicializar um novo nó na árvore. Cada nó recebe uma chave e tem seus ponteiros para os filhos esquerdo e direito inicializados como *NULL*. A função também inicializa o ponteiro para o nó pai como *NULL*, além de definir o nível e a altura como 0.

- Inserção de Nós:

  A inserção dos nós na árvore binária é feita de forma nivelada, garantindo que cada nó receba, no máximo, dois filhos (um à esquerda e um à direita). A função *insert* insere o nó na primeira posição disponível, percorrendo os nós da árvore em ordem de níveis. A inserção mantém a integridade da árvore binária, com as inserções

seguindo a ordem de entrada dos nós. Além disso, a função atualiza os campos level, height e parent conforme necessário.

- Cálculo da Altura da Árvore:

  A função *height* calcula a altura da árvore, que é a distância máxima da raiz até o nó mais profundo. Essa altura é necessária para determinar quantos níveis a árvore possui e, assim, imprimir os nós por nível. A altura de cada nó individual também é calculada e armazenada diretamente no campo height de cada nó.

- Impressão da Árvore por Nível

  A árvore é impressa por nível utilizando a função *printTree*, que chama a função *printLevel* para exibir os nós de cada nível. O algoritmo percorre a árvore por níveis e imprime os valores da esquerda para a direita, incluindo as informações de nível, altura e o ponteiro para o nó pai.

Testes e soluções:

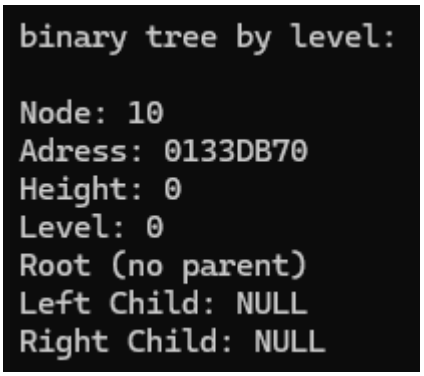
Foram realizados testes utilizando diferentes conjuntos de chaves para garantir que o programa se comportasse corretamente. Abaixo estão alguns exemplos de cenários de teste:

- **Cenário 1:** Inserção em Árvore Vazia

Entrada: {10}

Esperado: A árvore deve ser inicializada com o nó 10 como a raiz, com nível 0 e altura 0.

Saída:

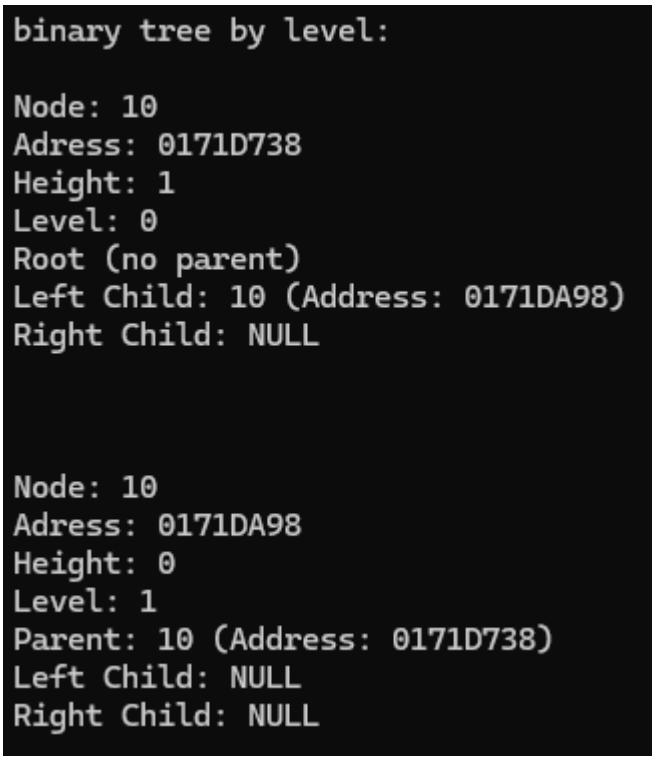


- **Cenário 2:** Inserção de Chaves Repetidas

Entrada: {10, 10}

Esperado: O código deve ignorar que a inserção está repetida, pois atualmente o código não trata chaves duplicadas.

Saída:



- **Cenário 3:** Inserção de um Número Elevado de Nós

Entrada: {10, 5, 15, 3, 7, 13, 20, 1, 4, 6, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, 205, 210, 215, 220, 225, 230, 235, 240, 245, 250, 255, 260, 265, 270, 275, 280, 285, 290, 295, 300, 305, 310, 315, 320, 325, 330, 335, 340, 345, 350, 355, 360, 365, 370, 375, 380, 385, 390, 395, 400, 405, 410, 415, 420, 425, 430, 435, 440, 445, 450}

Esperado: A árvore deve ser preenchida com até 100 elementos, com inserção correta em ordem de nível. O código deve continuar funcionando corretamente sem falhas de memória ou desempenho notável.

Saída:

```
binary tree by level:

Node: 10
Adress: 00BADA98
Height: 6
Level: 0
Root (no parent)
Left Child: 5 (Address: 00BAD9C0)
Right Child: 15 (Address: 00BADC48)



Node: 5
Adress: 00BAD9C0
Height: 5
Level: 1
Parent: 10 (Address: 00BADA98)
Left Child: 3 (Address: 00BADD68)
Right Child: 7 (Address: 00BADCD8)


Node: 15
Adress: 00BADC48
Height: 5
Level: 1
Parent: 10 (Address: 00BADA98)
Left Child: 13 (Address: 00BADAE0)
Right Child: 20 (Address: 00BADDB0)



Node: 3
Adress: 00BADD68
Height: 4
Level: 2
Parent: 5 (Address: 00BAD9C0)
Left Child: 1 (Address: 00BADA08)
Right Child: 4 (Address: 00BAD858)


Node: 7
Adress: 00BADCD8
Height: 4
Level: 2
Parent: 5 (Address: 00BAD9C0)
Left Child: 6 (Address: 00BAD978)
Right Child: 25 (Address: 00BAD780)
```

```
Node: 13
Adress: 00BADAE0
Height: 4
Level: 2
Parent: 15 (Address: 00BADC48)
Left Child: 30 (Address: 00BADB70)
Right Child: 35 (Address: 00BADDF8)


Node: 20
Adress: 00BADDB0
Height: 3
Level: 2
Parent: 15 (Address: 00BADC48)
Left Child: 40 (Address: 00BADBB8)
Right Child: 45 (Address: 00BADB28)



Node: 1
Adress: 00BADA08
Height: 3
Level: 3
Parent: 3 (Address: 00BADD68)
Left Child: 50 (Address: 00BADA50)
Right Child: 55 (Address: 00BAD930)


Node: 4
Adress: 00BAD858
Height: 3
Level: 3
Parent: 3 (Address: 00BADD68)
Left Child: 60 (Address: 00BADC00)
Right Child: 65 (Address: 00BAD6A8)


Node: 6
Adress: 00BAD978
Height: 3
Level: 3
Parent: 7 (Address: 00BADCD8)
Left Child: 70 (Address: 00BADE40)
Right Child: 75 (Address: 00BADC90)
```

[Indo para os últimos nós...]

```
Node: 405
Adress: 00BB10E0
Height: 0
Level: 6
Parent: 185 (Address: 00BAF7D8)
Left Child: NULL
Right Child: NULL


Node: 410
Adress: 00BB12D8
Height: 0
Level: 6
Parent: 190 (Address: 00BAF820)
Left Child: NULL
Right Child: NULL


Node: 415
Adress: 00BB0FC0
Height: 0
Level: 6
Parent: 190 (Address: 00BAF820)
Left Child: NULL
Right Child: NULL


Node: 420
Adress: 00BB1098
Height: 0
Level: 6
Parent: 195 (Address: 00BAF868)
Left Child: NULL
Right Child: NULL


Node: 425
Adress: 00BB1320
Height: 0
Level: 6
Parent: 195 (Address: 00BAF868)
Left Child: NULL
Right Child: NULL
```

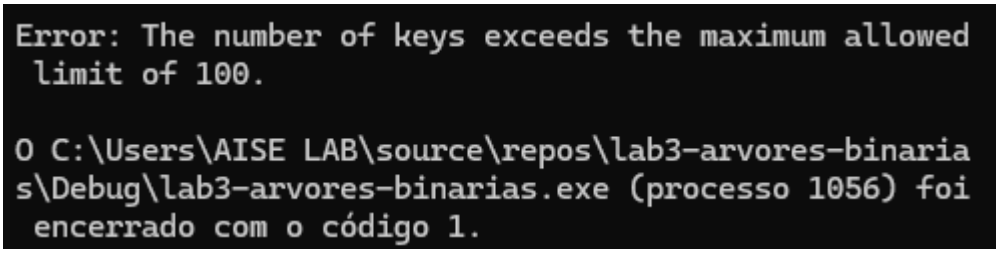
```
Node: 430
Adress: 00BB1368
Height: 0
Level: 6
Parent: 200 (Address: 00BAFCE8)
Left Child: NULL
Right Child: NULL


Node: 435
Adress: 00BB1008
Height: 0
Level: 6
Parent: 200 (Address: 00BAFCE8)
Left Child: NULL
Right Child: NULL


Node: 440
Adress: 00BB1248
Height: 0
Level: 6
Parent: 205 (Address: 00BAF9D0)
Left Child: NULL
Right Child: NULL


Node: 445
Adress: 00BB0E10
Height: 0
Level: 6
Parent: 205 (Address: 00BAF9D0)
Left Child: NULL
Right Child: NULL


Node: 450
Adress: 00BB1128
Height: 0
Level: 6
Parent: 210 (Address: 00BAFA18)
Left Child: NULL
Right Child: NULL
```

- **Cenário 4:** Inserção em uma Árvore maior do que o suportado

Entrada: {10, 5, 15, 3, 7, 13, 20, 1, 4, 6, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, 205, 210, 215, 220, 225, 230, 235, 240, 245, 250, 255, 260, 265, 270, 275, 280, 285, 290, 295, 300, 305, 310, 315, 320, 325, 330, 335, 340, 345, 350, 355, 360, 365, 370, 375, 380, 385, 390, 395, 400, 405, 410, 415, 420, 425, 430, 435, 440, 445, 450, 455, 460, 465, 470, 475, 480, 485, 490, 495, 500, 505, 510, 515, 520, 525, 530, 535, 540, 545, 550, 555, 560, 565, 570, 575, 580, 585, 590, 595, 600, 605, 610, 615, 620, 625, 630, 635, 640, 645, 650, 655, 660, 665, 670, 675, 680, 685, 690, 695, 700, 705, 710, 715, 720, 725, 730, 735, 740, 745, 750, 755, 760, 765, 770, 775, 780, 785, 790, 795, 800, 805, 810, 815, 820, 825, 830, 835, 840, 845, 850, 855, 860, 865, 870, 875, 880, 885, 890, 895, 900, 905, 910, 915, 920, 925, 930, 935, 940, 945, 950, 955, 960, 965, 970, 975, 980, 985, 990, 995, 1000, 1001}

Esperado: O código deve emitir uma mensagem indicando que o limite foi atingido.

Saída:



- **Cenário 5:** Testar Alturas de Nós

Entrada: {10, 15, 20}

Esperado: Em uma árvore desbalanceada, os nós devem ser inseridos e a altura calculada corretamente. Por exemplo, o nó 10 (raiz) deve ter altura 2, o nó 15 deve ter altura 1, e o nó 20, altura 0.

Saída:

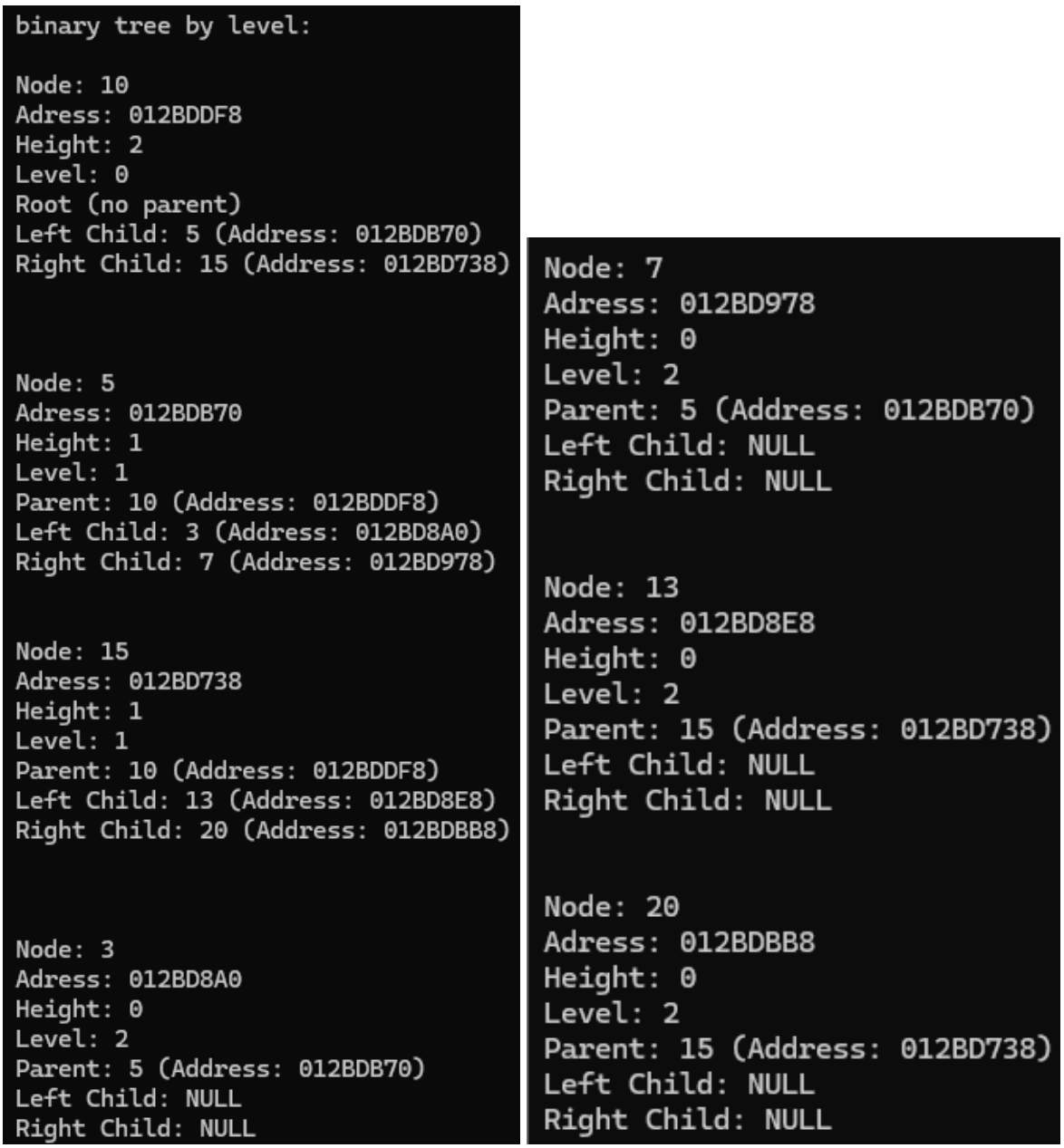
```
binary tree by level:

Node: 10
Adress: 00D6D6F0
Height: 1
Level: 0
Root (no parent)
Left Child: 15 (Address: 00D6DB28)
Right Child: 20 (Address: 00D6D978)



Node: 15
Adress: 00D6DB28
Height: 0
Level: 1
Parent: 10 (Address: 00D6D6F0)
Left Child: NULL
Right Child: NULL


Node: 20
Adress: 00D6D978
Height: 0
Level: 1
Parent: 10 (Address: 00D6D6F0)
Left Child: NULL
Right Child: NULL
```

- **Cenário 6:** Testar a Ligação entre Nó Pai e Nó Filho

Entrada: {10, 5, 15, 3, 7, 13, 20}

Esperado: Após cada inserção, o campo *parent* de cada nó deve estar corretamente atualizado para apontar para o nó pai adequado.

Saída:

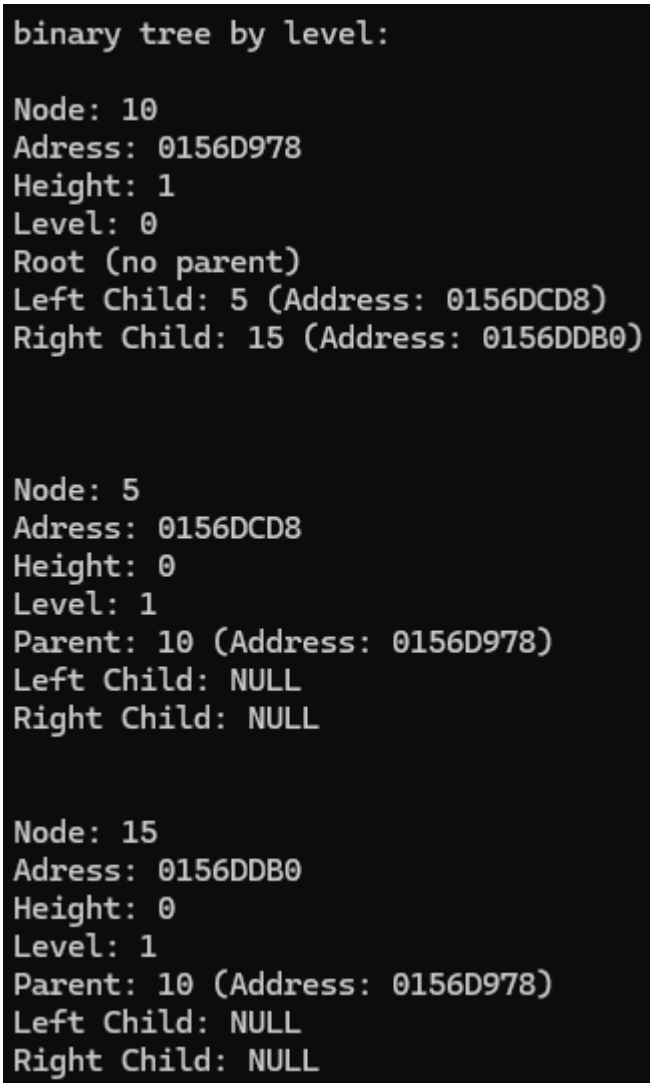


- **Cenário 7:** Testar Impressão de Ponteiro para o Pai

Entrada: {10, 5, 15}

Esperado: O nó raiz (10) deve ter *parent* nulo, enquanto os outros nós (5 e 15) devem exibir o ponteiro correto para o nó pai. A impressão não deve causar erros.

Saída:



4. **Observações e conclusões**

Durante a implementação da inserção de chaves por nível em uma árvore binária, alguns pontos importantes se destacaram:

Facilidades:

- Inserção por nível: A implementação da inserção por nível facilitou a construção da árvore binária, garantindo que os nós fossem alocados na primeira posição disponível. Isso resultou em uma árvore bem estruturada e alinhada por nível, o que também simplificou o processo de impressão por nível.

- Estrutura de nós: A adição dos campos de nível, altura e ponteiro para o pai em cada nó foi uma melhoria significativa. Esses dados adicionais permitiram maior controle sobre a estrutura da árvore, facilitando a visualização e a navegação, além de permitir a execução de testes mais detalhados.

Dificuldades:

- Controle de chaves duplicadas: Um dos principais desafios foi o fato de que o código não tratava chaves duplicadas. Durante os testes, a inserção de chaves repetidas resultou em comportamentos inesperados. Isso indicou a necessidade de melhorar a função de inserção para lidar com duplicidade de chaves, emitindo um erro ou ignorando a segunda inserção.

- Inserção de grandes quantidades de nós: Durante o teste com inserção de 100 ou mais nós, o código funcionou conforme esperado, mas não havia um controle explícito para limitar a quantidade de nós inseridos. Embora o array nodes[] tenha capacidade limitada, o código pode ser aprimorado para fornecer uma mensagem clara ao atingir o limite, evitando um possível estouro de memória.

- Falha de alocação de memória: Simular cenários de falta de memória foi um ponto que exigiu mais atenção. O código atual não trata casos em que malloc() falha, o que pode resultar em comportamentos indesejados. Uma melhoria seria incluir verificações de erro ao alocar memória para novos nós.

Conclusão:

A implementação da árvore binária com inserção por nível usando um array de ponteiros alcançou os objetivos propostos, permitindo a construção e impressão de nós sem a necessidade de ordenação. A árvore foi preenchida e exibida corretamente, com o nível, altura e ponteiro para o pai sendo associados a cada nó de forma satisfatória.

Para cenários futuros, podem ser considerados aprimoramentos, como o controle de chaves duplicadas para evitar inserções repetidas. Esse ajuste traria maior robustez ao código. No geral, o programa apresentou resultados satisfatórios e funcionou conforme o esperado para os cenários testados.

Segue o resultado do funcionamento geral do programa com as chaves propostas no enunciado. A aplicação para a conclusão se limitou a isso, apesar do código ter sido escrito para lidar com entradas mais variadas e complexas, garantindo um gerenciamento mais apurado dos nós da árvore, além de realizar o tratamento de erros citados.

```
binary tree by level:

Node: 10
Adress: 0117D8C0
Height: 3
Level: 0
Root (no parent)
Left Child: 5 (Address: 0117DCF8)
Right Child: 15 (Address: 0117DBD8)



Node: 5
Adress: 0117DCF8
Height: 2
Level: 1
Parent: 10 (Address: 0117D8C0)
Left Child: 3 (Address: 0117DE18)
Right Child: 7 (Address: 0117D950)


Node: 15
Adress: 0117DBD8
Height: 1
Level: 1
Parent: 10 (Address: 0117D8C0)
Left Child: 13 (Address: 0117DE60)
Right Child: 20 (Address: 0117DD88)



Node: 3
Adress: 0117DE18
Height: 1
Level: 2
Parent: 5 (Address: 0117DCF8)
Left Child: 1 (Address: 0117DA70)
Right Child: 4 (Address: 0117D710)


Node: 7
Adress: 0117D950
Height: 1
Level: 2
Parent: 5 (Address: 0117DCF8)
Left Child: 6 (Address: 0117DC20)
Right Child: NULL
```

```
Node: 13
Adress: 0117DE60
Height: 0
Level: 2
Parent: 15 (Address: 0117DBD8)
Left Child: NULL
Right Child: NULL


Node: 20
Adress: 0117DD88
Height: 0
Level: 2
Parent: 15 (Address: 0117DBD8)
Left Child: NULL
Right Child: NULL



Node: 1
Adress: 0117DA70
Height: 0
Level: 3
Parent: 3 (Address: 0117DE18)
Left Child: NULL
Right Child: NULL


Node: 4
Adress: 0117D710
Height: 0
Level: 3
Parent: 3 (Address: 0117DE18)
Left Child: NULL
Right Child: NULL


Node: 6
Adress: 0117DC20
Height: 0
Level: 3
Parent: 7 (Address: 0117D950)
Left Child: NULL
Right Child: NULL
```