Estrutura de Dados II

Universidade de Vila Velha

Comprometida com a excelência no ensino, pesquisa e inovação.

✉ wanderson.santana@uvv.br

Av. Comissário José Dantas de Melo, 21 – Boa Vista – Vila Velha/ES

www.uvv.br | (27) 3421-2001

2 Árvores

Resumo

Como listas ligadas, árvores são constituídas de células. Uma espécie comum de árvores é a árvore binária, em que cada célula contém referências a duas outras células (possivelmente nulas). Tais referências são chamadas de subárvore esquerda e direita. Como as células de listas ligadas, as células de árvores também contém uma carga.

"Nunca se preocupe com números: ajude uma pessoa de cada vez e comece sempre pela mais próxima de você."
Madre Tereza de Calcutá

Árvores

Árvores são estruturas de dados hierárquicas, amplamente utilizadas em ciência da computação para representar relações do tipo pai-filho entre elementos. Diferentemente de estruturas lineares como listas, filas e pilhas — onde os dados são organizados de forma sequencial — as árvores organizam os dados de maneira não linear, permitindo representações mais flexíveis e eficientes em diversos contextos. Segundo Cormen et.al. (2009) [1], árvores são fundamentais para a implementação de algoritmos eficientes de busca, ordenação e manipulação de dados estruturados.

O elemento superior de uma árvore é conhecido como **nó raiz**, o qual representa o ponto de partida da estrutura. Cada nó pode estar conectado a outros nós, chamados **filhos**, formando uma organização em níveis. Esse modelo permite representar dados complexos e estabelecer relações de hierarquia de forma natural e eficiente.

Estruturas de árvore são fundamentais em diversas aplicações, como a análise de expressões, algoritmos de busca, ordenação por prioridade e representação de dados estruturados (como em documentos `XML` e `HTML`). Através do estudo das árvores, é possível entender como organizar e acessar informações de forma otimizada.

Nesta unidade, abordaremos os seguintes tópicos:

- Termos e definições de árvores
- Árvores binárias e árvores de busca binária
- Travessia de árvore
- Árvores de busca binária

Terminologia

Para compreender o funcionamento das estruturas de dados em árvore, é essencial conhecer os principais termos associados a esse tipo de representação hierárquica. Embora semelhantes em alguns aspectos a listas ligadas — pois ambas utilizam nós para armazenar dados e referências — as árvores introduzem novos conceitos estruturais.

A Figura 2.1 apresenta uma representação conceitual de uma árvore, composta por nós rotulados com letras de A a M.

Os principais termos utilizados para descrever árvores são:

- **Nó**: elemento fundamental de uma árvore, responsável por armazenar dados. Cada letra na Figura 2.1 representa um nó distinto.
- **Nó raiz**: o nó no topo da hierarquia, sem nenhum pai. É o ponto de partida da árvore. No exemplo, o nó A é a raiz da árvore.
- **Subárvore**: qualquer estrutura em árvore que se origina a partir de um nó específico. Por exemplo, os nós F, K e L formam uma subárvore da árvore principal.
- **Grau**: número de filhos diretos que um nó possui. Por exemplo, o grau do nó A é 2; o de B é 3; e o de G é 1.

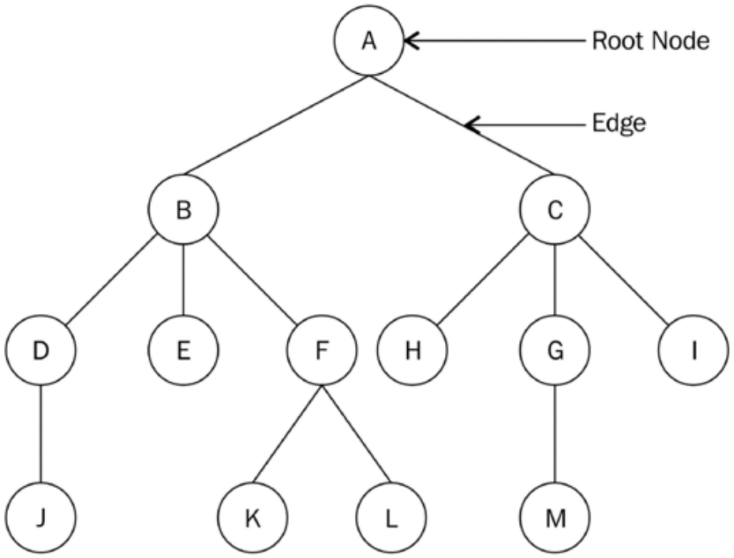


Figura 2.1: Exemplo de estrutura de dados em árvore, com nós rotulados de A a M.

- **Nó folha**: nó terminal, que não possui filhos. Seu grau é sempre zero. No diagrama, os nós J, E, K, L, H, M e I são folhas.
- **Borda (ou aresta)**: conexão entre dois nós. Em uma árvore com n nós, há exatamente $n-1$ bordas. Exemplos de bordas estão representados na Figura 2.1 pelas ligações entre os nós.
- **Pai**: nó que possui pelo menos um filho. Por exemplo, B é pai de D, E e F; e F é pai de K e L.
- **Filho**: nó descendente de um pai. Por exemplo, B e C são filhos de A; e G, H e I são filhos de C.
- **Irmãos**: nós que compartilham o mesmo pai. Por exemplo, B e C são irmãos; assim como D, E e F.
- **Nível**: indica a profundidade de um nó em relação à raiz. O nó raiz está no nível 0; seus filhos estão no nível 1; os filhos destes, no nível 2; e assim por diante. No exemplo, A está no nível 0; B e C no nível 1; D, E, F, G, H e I no nível 2.
- **Altura da árvore**: corresponde ao número de nós no caminho mais longo entre a raiz e uma folha. No exemplo, a altura da árvore é 4, considerando os caminhos A–B–D–J, A–C–G–M e A–B–F–K.

- **Profundidade**: número de bordas no caminho entre a raiz e um nó específico. Por exemplo, a profundidade do nó H é 2.

Enquanto estruturas lineares, como listas, pilhas e filas, organizam dados em sequência e permitem percorrer todos os elementos em uma única varredura, as árvores estruturam os dados de forma não linear. Nelas, os elementos são organizados hierarquicamente em relações de dependência (pai-filho), e a travessia completa pode exigir estratégias específicas.

Importante notar que, em uma árvore válida, não podem existir ciclos, e toda árvore deve ser composta por uma única raiz. Quando não há nenhum nó presente, temos uma **árvore vazia**.

Dentre os diversos tipos de árvores, destaca-se um modelo fundamental e amplamente utilizado: a **árvore binária**, que será abordada a seguir.

Árvore Binária

Uma **árvore binária** é uma estrutura de dados hierárquica na qual cada nó pode ter, no máximo, dois filhos — tradicionalmente chamados de **subárvore esquerda** e **subárvore direita**. Essa estrutura é amplamente empregada em algoritmos de busca, ordenação e manipulação de dados hierárquicos [2].

A Figura 2.2 apresenta um exemplo básico de árvore binária, onde o nó raiz possui dois filhos.

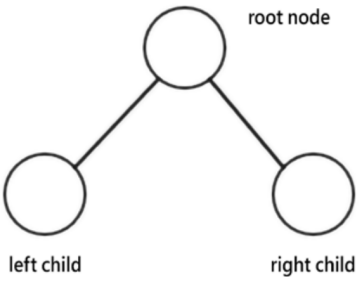


Figura 2.2: Exemplo de árvore binária.

Na árvore binária, cada subárvore também é, por definição, uma árvore binária. A Figura 2.3 ilustra uma estrutura com cinco nós, na qual o nó raiz R tem duas subárvores ($T1$ e $T2$).

Existem diversas classificações para árvores binárias, conforme o padrão de preenchimento de seus nós:

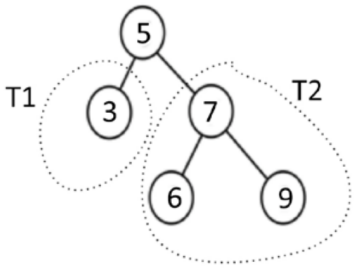


Figura 2.3: Exemplo de árvore binária com cinco nós.

- **Árvore binária completa:** é aquela em que todos os níveis, com exceção do último, estão completamente preenchidos; no último nível, os nós devem estar dispostos da esquerda para a direita, sem lacunas. Essa definição é amplamente adotada na literatura especializada [3]. A Figura 2.4 ilustra um exemplo.

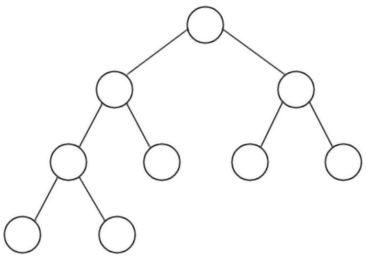


Figura 2.4: Exemplo de árvore binária completa.

- **Árvore binária regular (ou cheia):** também conhecida como *estritamente binária*, é aquela em que todos os nós internos possuem exatamente dois filhos. Nenhum nó interno pode ter apenas um descendente [4]. A Figura 2.5 apresenta essa estrutura.

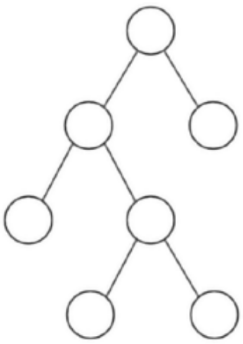


Figura 2.5: Exemplo de árvore binária regular.

- **Árvore binária perfeita:** todos os nós internos possuem dois filhos, e todas as folhas estão no mesmo nível, tornando a árvore completamente preenchida [2]. Veja o exemplo na Figura 2.6.

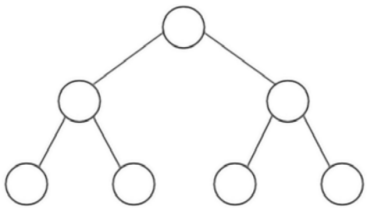


Figura 2.6: Exemplo de árvore binária perfeita.

- **Árvore binária balanceada:** ocorre quando, para todo nó da árvore, a diferença de altura entre as subárvores esquerda e direita é, no máximo, 1. Esse balanceamento é fundamental para garantir a eficiência das operações de busca, inserção e remoção [2]. A Figura 2.7 ilustra esse conceito.

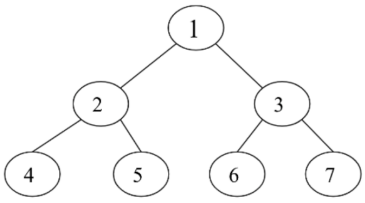


Figura 2.7: Exemplo de árvore binária balanceada.

- **Árvore binária desbalanceada:** caracteriza-se por conter ao menos um nó cuja diferença de altura entre as subárvores esquerda e direita excede 1. Esse desbalanceamento pode impactar negativamente no desempenho de operações fundamentais, tornando a árvore comparável a uma lista encadeada em casos extremos [5]. Veja a Figura 2.8.

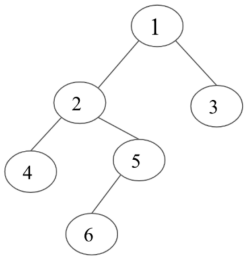


Figura 2.8: Exemplo de árvore binária desbalanceada.

Compreender as variações estruturais das árvores binárias é essencial para selecionar a abordagem mais adequada a cada problema computacional. A seguir, discutiremos como implementar uma estrutura de árvore binária simples.

Implementação dos nós de uma árvore

Como discutido anteriormente, uma árvore binária é composta por **nós**, e cada nó contém um valor (ou carga) e, no máximo, duas referências — uma para o filho à esquerda e outra para o filho à direita.

A seguir, implementamos uma classe `Node`, em `Python`, que representa um nó de uma árvore binária. Essa classe conterá três atributos: o dado armazenado e os ponteiros para os filhos esquerdo e direito:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.left_child = None
5         self.right_child = None
```

Para entender como essa estrutura funciona na prática, construiremos uma árvore binária simples com quatro nós, conforme ilustrado na Figura 2.9. Essa árvore possui um nó raiz ($n1$), dois filhos ($n2$ e $n3$) e um neto ($n4$), posicionado como filho esquerdo de $n2$.

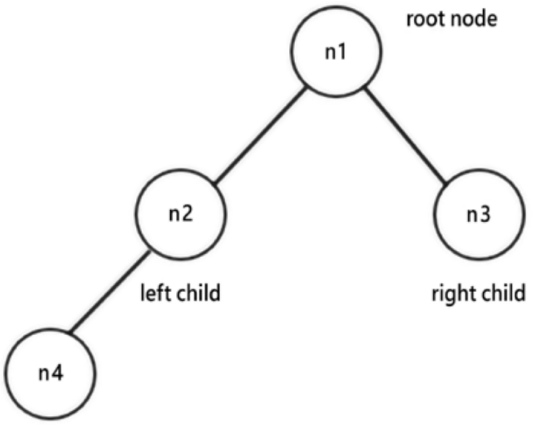


Figura 2.9: Exemplo de árvore binária com quatro nós.

A criação dos nós em código é realizada da seguinte forma:

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

Após a criação, conectamos os nós entre si, seguindo as regras de uma árvore binária:

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

Com isso, temos uma árvore binária corretamente estruturada com hierarquia entre os nós. Essa representação permite a execução de diversas operações fundamentais em árvores, como inserção, busca e, principalmente, **travessia** (ou percurso), que será o foco da próxima seção.

A travessia de uma árvore é o processo de visitar todos os seus nós em uma determinada ordem. Dependendo do objetivo da aplicação, diferentes estratégias de travessia podem ser utilizadas, como as abordagens em pré-ordem, em ordem e em pós-ordem [2].

Travessia em ordem

A **travessia em ordem** (in-order traversal) consiste em percorrer recursivamente a subárvore esquerda, visitar o nó atual (raiz local) e, por fim, percorrer recursivamente a subárvore direita. Essa abordagem é especialmente útil para árvores binárias de busca, pois resulta na visita dos nós em ordem crescente.

A travessia em ordem segue os seguintes passos:

- Percorrer recursivamente a subárvore esquerda;
- Visitar o nó atual;
- Percorrer recursivamente a subárvore direita.

Considere a árvore binária ilustrada na Figura 2.10. A travessia em ordem para essa estrutura ocorre da seguinte forma:

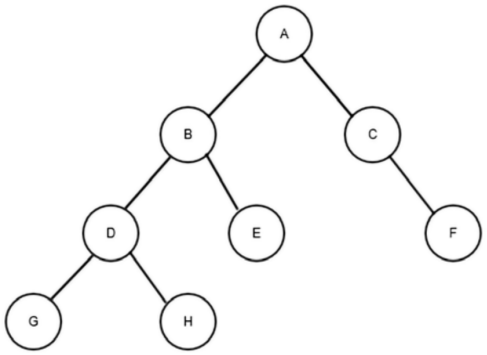


Figura 2.10: Um exemplo de árvore binária para travessia.

1. Iniciamos pela subárvore esquerda do nó A (nó raiz), cujo nó principal é B.
2. Em B, percorremos sua subárvore esquerda, que tem como raiz o nó D.
3. O nó D possui dois filhos: primeiro visitamos seu filho esquerdo (G), depois o próprio D, e em seguida o filho direito (H).
4. Completando a travessia da subárvore de D, retornamos a B e visitamos seu filho direito (E).
5. Após concluir a subárvore esquerda de A, visitamos o próprio nó A.
6. Em seguida, avançamos para a subárvore direita de A, que possui como raiz o nó C.
7. O nó C não possui filho à esquerda, então visitamos C e depois seu filho direito, F.

A ordem final da travessia em ordem é: `G - D - H - B - E - A - C - F`.

A implementação em Python dessa abordagem é mostrada a seguir:

```
def inorder(root_node):
    current = root_node
    if current is None:
        return

    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)

inorder(n1)
```

Aplicando essa função à árvore de quatro nós definida anteriormente (com $n1$ como nó raiz), obtemos a seguinte saída:

```
left grandchild node
left child node
root node
right child node
```

Travessia pré-ordenada

A **travessia pré-ordenada** (pre-order traversal) percorre os nós na seguinte ordem: nó atual (raiz), subárvore esquerda e subárvore direita. Essa abordagem é útil, por exemplo, para copiar uma árvore ou avaliar expressões prefixadas.

A sequência da travessia pré-ordenada é:

1. Visitar o nó atual (raiz);
2. Percorrer recursivamente a subárvore esquerda;
3. Percorrer recursivamente a subárvore direita.

Vamos aplicar essa lógica à árvore representada na Figura 2.10:

1. Começamos pelo nó raiz A.
2. A subárvore esquerda de A tem como raiz o nó B, que também é visitado.
3. Em B, vamos para a subárvore esquerda (D), que por sua vez visita G (esquerda), depois H (direita), e retorna a D.
4. Voltamos a B e visitamos seu filho direito, E.
5. Após completar a subárvore esquerda de A, seguimos para a subárvore direita, onde visitamos C, depois seu filho direito F.

A sequência final da travessia pré-ordenada é: `A - B - D - G - H - E - C - F`.

A seguir, o código Python correspondente:

```python
def preorder(root_node):
    current = root_node
    if current is None:
        return

    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)

preorder(n1)
```

Aplicando essa função à árvore de quatro nós definida anteriormente, com $n1$ como nó raiz, obtemos a seguinte saída:

```
root node
left child node
left grandchild node
right child node
```

Travessia pós-ordenada

A **travessia pós-ordenada** (post-order traversal) visita os nós na seguinte ordem: subárvore esquerda, subárvore direita e nó atual (raiz). Essa abordagem é útil para liberar memória ou avaliar expressões pós-fixadas.

A sequência da travessia pós-ordenada é:

1. Percorrer recursivamente a subárvore esquerda;
2. Percorrer recursivamente a subárvore direita;
3. Visitar o nó atual (raiz).

Aplicando essa lógica à árvore da Figura 2.10, temos:

1. Começamos pela subárvore esquerda de A (nó raiz), cujo nó principal é B.
2. Em B, vamos para sua subárvore esquerda (D), visitamos G, depois H, e por fim D.
3. Retornamos a B e visitamos o filho direito E, e por fim o próprio B.
4. A subárvore direita de A contém o nó C, com um filho à direita: F. Visitamos F, depois C.
5. Por fim, visitamos o nó raiz A.

A sequência final da travessia pós-ordenada é: `G - H - D - E - B - F - C - A`.

Veja a implementação correspondente em Python:

```
def postorder(root_node):
    current = root_node
    if current is None:
        return

    postorder(current.left_child)
    postorder(current.right_child)
    print(current.data)

postorder(n1)
```

Ao aplicar essa função à árvore de quatro nós previamente criada, obtemos a seguinte saída:

```
left grandchild node
left child node
right child node
root node
```

Travessia por ordem de nível

A **travessia por ordem de nível**, também chamada de *level-order traversal*, percorre a árvore visitando todos os nós de um nível antes de avançar para o próximo. Ela é implementada com o uso de uma estrutura de dados do tipo *fila* (FIFO – *First-In, First-Out*).

Funcionamento:

1. Começamos pela raiz da árvore (nível 0).
2. Visitamos os nós do nível seguinte (nível 1), da esquerda para a direita.
3. Repetimos esse processo até todos os níveis da árvore serem percorridos.

Considere a árvore binária ilustrada na Figura 2.11:

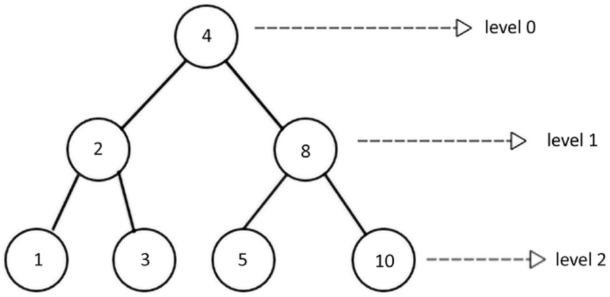


Figura 2.11: Uma árvore de exemplo para entender a travessia por ordem de nível.

A sequência da travessia por ordem de nível para essa árvore é:

`4, 2, 8, 1, 3, 5, 10`

Essa travessia pode ser implementada em `Python` utilizando a classe `deque`, da biblioteca `collections`, como mostrado a seguir:

```python
from collections import deque

class Node:
    def __init__(self, data):
        self.data = data
        self.left_child = None
        self.right_child = None
```

```
8
9  n1 = Node("root node")
10 n2 = Node("left child node")
11 n3 = Node("right child node")
12 n4 = Node("left grandchild node")
13
14 n1.left_child = n2
15 n1.right_child = n3
16 n2.left_child = n4
17
18 def level_order_traversal(root_node):
19     list_of_nodes = []
20     traversal_queue = deque([root_node])
21
22     while len(traversal_queue) > 0:
23         node = traversal_queue.popleft()
24         list_of_nodes.append(node.data)
25
26         if node.left_child:
27             traversal_queue.append(node.left_child)
28         if node.right_child:
29             traversal_queue.append(node.right_child)
30
31     return list_of_nodes
32
33 print(level_order_traversal(n1))
```

Explicação do código:

- O nó raiz é adicionado à fila.
- Enquanto houver elementos na fila:
  - Retiramos o primeiro elemento;
  - Registramos seu valor;
  - Adicionamos à fila os filhos esquerdo e direito, caso existam.

Saída esperada:

```
['root node', 'left child node', 'right child node', 'left grandchild node']
```

Considerações finais sobre as travessias:

Cada tipo de travessia atende a propósitos específicos:

- **Em ordem:** útil para retornar dados em ordem crescente (ou decrescente, se invertida).
- **Pré-ordem:** apropriada para exportar ou copiar uma árvore.
- **Pós-ordem:** útil para deletar uma árvore, avaliar expressões ou montar código em compiladores.
- **Por nível:** ideal para verificar a estrutura da árvore ou determinar largura.

Aplicações comuns de árvores binárias:

1. **Avaliação de expressões aritméticas:** Árvores binárias são amplamente utilizadas na representação e avaliação de expressões matemáticas, especialmente em compiladores, onde são conhecidas como *árvores de expressão*. Cada nó interno representa um operador (como +, −, ∗, /) e cada folha representa um operando. Essa estrutura facilita a análise sintática e semântica de linguagens formais [6, 7].
2. **Compressão de dados – Codificação de Huffman:** Um dos algoritmos de compressão sem perda mais eficientes é a codificação de Huffman, que utiliza árvores binárias para construir códigos prefixados com base na frequência de ocorrência dos símbolos. Essa técnica é aplicada em formatos como JPEG e MP3, sendo fundamental para a redução de espaço de armazenamento [8, 9].
3. **Estruturas de busca eficientes – Árvores de busca binária (BST):** BSTs permitem operações de busca, inserção e remoção em tempo médio de $O(\log n)$, desde que a árvore esteja balanceada. São amplamente utilizadas em bancos de dados, sistemas de arquivos e armazenamento em memória principal [2, 5, 4].
4. **Filas de prioridade – Árvores heap:** Estruturas como *heaps binários*, que também são árvores binárias, são usadas para implementar filas de prioridade. Elas permitem encontrar e remover o maior ou menor elemento em tempo logarítmico, sendo aplicadas

em algoritmos de escalonamento, compressão de dados e algoritmos de caminho mínimo como Dijkstra [4, 2].

Analisando uma expressão polonesa reversa

A construção de uma árvore de expressão a partir da notação pós-fixa (também conhecida como notação polonesa reversa) é comumente realizada utilizando uma pilha. O algoritmo processa um símbolo por vez: se o símbolo for um operando, criamos um nó para ele e o empilhamos; se o símbolo for um operador, dois nós são desempilhados para formar uma nova subárvore, onde o segundo nó desempilhado se torna o filho esquerdo e o primeiro nó desempilhado o filho direito do operador. A referência dessa subárvore recém-criada é então empilhada novamente. Ao final do processo, a pilha conterá uma única referência para a raiz da árvore de expressão completa [6, 7].

Consideremos a expressão pós-fixa:

$$4 \quad 5 \quad + \quad 5 \quad 3 \quad - \quad *$$

Inicialmente, empilhamos os operandos 4 e 5, como ilustrado na Figura 2.12:

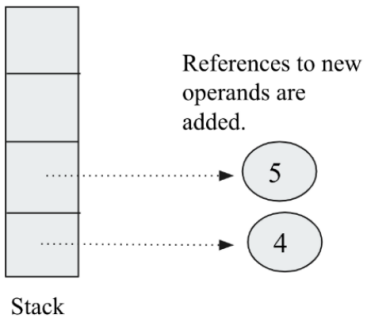


Figura 2.12: Empilhamento dos operandos 4 e 5.

Ao encontrar o operador $+$, criamos um nó raiz para este operador. Em seguida, desempilhamos os dois nós superiores da pilha: o primeiro desempilhado se torna o filho direito e o segundo, o filho esquerdo da subárvore, conforme mostra a Figura 2.13:

Os operandos seguintes, 5 e 3, são empilhados. Ao processar o operador $-$, aplicamos o mesmo procedimento, gerando a subárvore ilustrada na Figura 2.14:

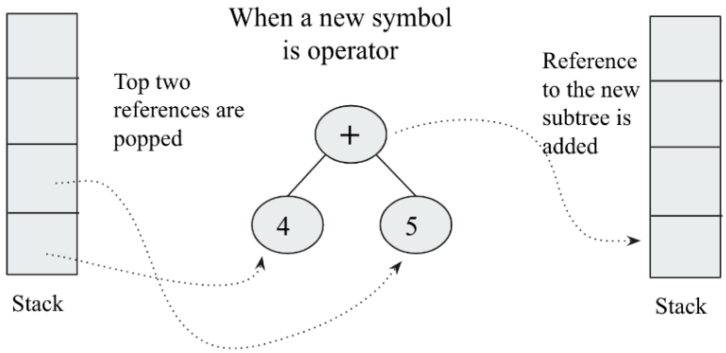


Figura 2.13: Criação da subárvore com o operador $+$.

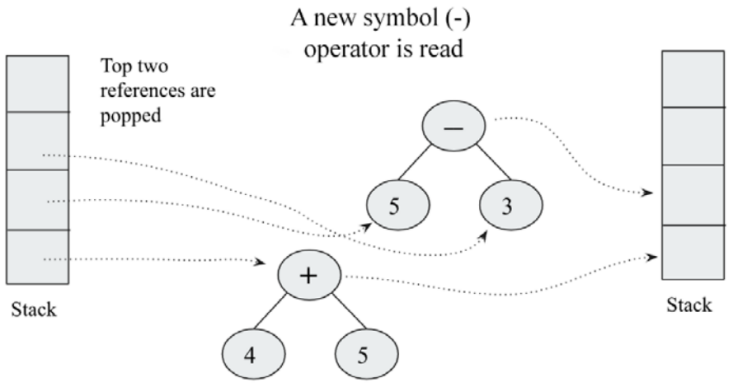


Figura 2.14: Criação da subárvore com o operador $-$.

Finalmente, o operador $*$ processa as duas subárvores empilhadas, resultando na árvore completa mostrada na Figura 2.15:

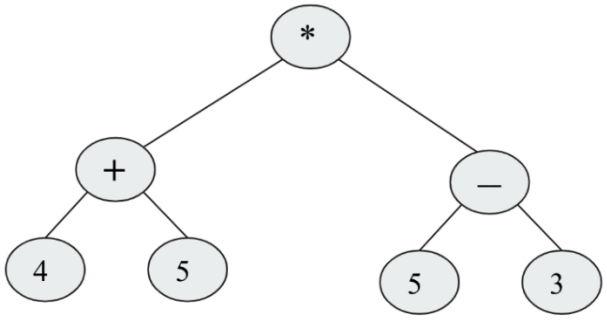


Figura 2.15: Árvore de expressão final com o operador $*$.

Implementação em Python

A classe que representa um nó da árvore pode ser implementada da seguinte forma:

```
class TreeNode:
    def __init__(self, data=None):
        self.data = data
        self.right = None
        self.left = None
```

Uma implementação simples de pilha pode ser feita assim:

```
class Stack:
    def __init__(self):
        self.elements = []

    def push(self, item):
        self.elements.append(item)

    def pop(self):
        return self.elements.pop()
```

Para construir a árvore a partir de uma expressão pós-fixa, podemos processar cada termo da expressão da seguinte maneira:

```
expr = "4 5 + 5 3 - *".split()
stack = Stack()

for term in expr:
    if term in "+-*/":
        node = TreeNode(term)
        node.right = stack.pop()
        node.left = stack.pop()
    else:
        node = TreeNode(int(term))
    stack.push(node)
```

Observe que os operandos são convertidos para inteiros para permitir cálculos posteriores. A ordem em que os filhos são atribuídos é importante para preservar a correta associatividade e avaliação da expressão.

Para avaliar a árvore resultante, definimos uma função recursiva que processa os operadores conforme suas operações aritméticas:

```
def calc(node):
    if node.data == "+":
        return calc(node.left) + calc(node.right)
    elif node.data == "-":
        return calc(node.left) - calc(node.right)
    elif node.data == "*":
        return calc(node.left) * calc(node.right)
    elif node.data == "/":
        return calc(node.left) / calc(node.right)
    else:
        return node.data
```

Finalmente, extraímos a raiz da árvore da pilha e calculamos o resultado:

```
root = stack.pop()
result = calc(root)
print(result)
```

A execução do código acima produz o valor 18, que corresponde à avaliação da expressão $(4 + 5) \times (5 - 3)$.

Considerações finais

Árvores de expressão são ferramentas fundamentais para a representação e avaliação de expressões aritméticas, especialmente em compiladores e interpretadores [6]. Além disso, são úteis para a avaliação de expressões em notações pós-fixa, prefixa e infixa, e para a análise da precedência e associatividade dos operadores [7].

Utilizando o módulo `queue` para travessias em árvores

Ao estudar estruturas de dados como árvores binárias, é importante compreender não apenas seus conceitos teóricos, mas também como implementá-los na prática. Em particular, **algoritmos de travessia** em árvores fazem uso de estruturas auxiliares, como **pilhas** ou **filas**.

O módulo nativo queue da linguagem Python fornece uma forma simples e eficaz de representar essas estruturas auxiliares — especialmente úteis em travessias como **BFS** (Busca em Largura) e **DFS** (Busca em Profundidade).

O módulo `queue` em Python

O módulo `queue` fornece três classes principais:

- `Queue`: Representa uma **fila** tradicional (FIFO – First In, First Out).
- `LifoQueue`: Representa uma **pilha** (LIFO – Last In, First Out).
- `PriorityQueue`: Representa uma fila com **prioridades**, útil quando os elementos devem ser processados com base em algum critério de ordenação.

Essas classes oferecem métodos como:

- ✓ `put(item)`: Insere um elemento na estrutura.
- ✓ `get()`: Remove e retorna o próximo elemento da estrutura.
- ✓ `empty()`: Retorna `True` se a estrutura estiver vazia.
- ✓ `full()`: Retorna `True` se a estrutura estiver cheia (caso tenha um limite).
- ✓ `qsize()`: Retorna o número de elementos na estrutura.

```python
import queue

fila_fifo = queue.Queue()
fila_lifo = queue.LifoQueue()
fila_prioridade = queue.PriorityQueue()
```

Travessia em Largura com `queue.Queue`

Um exemplo clássico do uso de `queue.Queue` é a **travessia em largura (BFS)** em uma árvore binária. Nesse tipo de travessia, os nós são visitados por **nível**, começando pela raiz, depois os filhos diretos, e assim por diante.

```python
from queue import Queue

class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None

def bfs(raiz):
    if not raiz:
        return
    fila = Queue()
    fila.put(raiz)

    while not fila.empty():
        no_atual = fila.get()
        print(no_atual.valor, end=' ')
        if no_atual.esquerda:
            fila.put(no_atual.esquerda)
        if no_atual.direita:
            fila.put(no_atual.direita)

# Exemplo de uso
raiz = No(1)
raiz.esquerda = No(2)
raiz.direita = No(3)
raiz.esquerda.esquerda = No(4)
raiz.esquerda.direita = No(5)

bfs(raiz)
```

Saída:

```
1 2 3 4 5
```

Esse algoritmo utiliza uma fila FIFO (`queue.Queue`) para garantir que os nós sejam processados por nível — um conceito essencial na representação de estruturas hierárquicas como

árvores.

Comparando FIFO, LIFO e Priority Queue

Além da fila FIFO, o módulo queue permite simular **outras estratégias** de processamento, que também podem ser úteis em estruturas de dados como árvores:

```
import queue

# FIFO - First In, First Out
fifo = queue.Queue()
fifo.put("A")
fifo.put("B")
print(fifo.get())  # Saida: A

# LIFO - Last In, First Out (Pilha)
lifo = queue.LifoQueue()
lifo.put("A")
lifo.put("B")
print(lifo.get())  # Saida: B

# Fila com prioridade
prioridade = queue.PriorityQueue()
prioridade.put((2, "A"))
prioridade.put((1, "B"))
print(prioridade.get())  # Saida: (1, 'B')
```

Cada abordagem pode ser adequada a diferentes estratégias de processamento em algoritmos relacionados a árvores:

- **FIFO**: ideal para travessia em largura.
- **LIFO**: útil para travessia em profundidade (como uma pilha).
- **PriorityQueue**: pode ser usada em algoritmos com prioridade, como busca A* ou Dijkstra em árvores ou grafos.

Exercícios: Travessias em Árvores (BFS e DFS)

1. **Travessia em Largura (BFS) — interpretação de código**

   Considere o código abaixo:

```python
from queue import Queue

class No:
    def __init__(self, valor):
        self.valor = valor
        self.esquerda = None
        self.direita = None

raiz = No('A')
raiz.esquerda = No('B')
raiz.direita = No('C')
raiz.esquerda.esquerda = No('D')
raiz.esquerda.direita = No('E')
raiz.direita.direita = No('F')

def bfs(raiz):
    fila = Queue()
    fila.put(raiz)
    while not fila.empty():
        no = fila.get()
        print(no.valor, end=' ')
        if no.esquerda:
            fila.put(no.esquerda)
        if no.direita:
            fila.put(no.direita)

bfs(raiz)
```

   Pergunta: Qual será a saída do programa?

   (a) A B C D E F

   (b) A B D E C F

   (c) D B E A F C

(d) A C F B D E

Resposta esperada: (a) A B C D E F

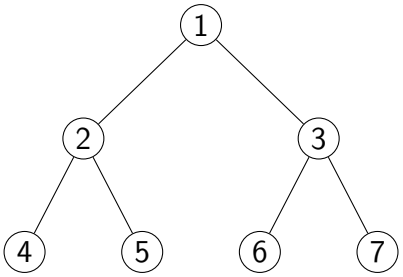
2. **Travessia em Profundidade (DFS) — implementação com pilha**

   Reescreva o algoritmo de travessia em profundidade (DFS) para a mesma árvore do exercício anterior, utilizando uma estrutura de pilha com `queue.LifoQueue()`.

   Dica: A DFS pode ser feita de forma iterativa usando uma pilha onde os filhos são empilhados na ordem inversa da desejada.

3. **Diferença prática entre BFS e DFS**

   Crie duas funções, `bfs(raiz)` e `dfs(raiz)`, que imprimem os nós de uma árvore binária. Em seguida, use a árvore abaixo para testar ambas as funções:

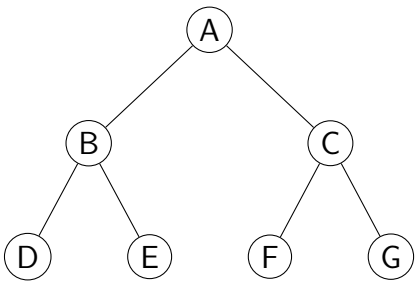


   Saída esperada:

   - `bfs(raiz)`: 1 2 3 4 5 6 7
   - `dfs(raiz)` (pré-ordem): 1 2 4 5 3 6 7

4. **Contando nós em cada nível com BFS**

   Modifique o algoritmo de travessia em largura para imprimir a **quantidade de nós em cada nível da árvore**. Use a seguinte árvore como exemplo:



Saída esperada:

```
Nivel 0: 1 no(s)
Nivel 1: 2 no(s)
Nivel 2: 4 no(s)
```

5. **Aplicação: Localizando um valor na árvore (BFS vs DFS)**

   Implemente duas funções para buscar um valor em uma árvore binária:

   - Uma utilizando BFS.
   - Outra utilizando DFS.

   Utilize a árvore do Exercício 3 para testar ambas. Meça a quantidade de nós visitados até encontrar o valor 7. Compare o desempenho das duas abordagens.

   Pergunta: Em qual das abordagens o valor 7 foi encontrado mais rapidamente? Por quê?

Referências Bibliográficas

[1] CORMEN, T. H. et al. *Algoritmos: teoria e prática*. 3. ed.. ed. Rio de Janeiro: Elsevier, 2009.

[2] CORMEN, T. H. et al. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

[3] GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados e Algoritmos em Java*. 6. ed. Rio de Janeiro: LTC, 2015.

[4] WEISS, M. A. *Data Structures and Algorithm Analysis in C++*. 4. ed. Boston: Pearson, 2013.

[5] KNUTH, D. E. *The Art of Computer Programming*. vol. 1–3, terceira edição. Reading, MA, USA: Addison-Wesley, 1997.

[6] AHO, A. V.; ULLMAN, J. D. *Fundamentals of Computer Science: Principles, Processes, and Structures*. [S.l.]: Computer Science Press, 1979.

[7] GRIES, D. *The Science of Programming*. [S.l.]: Springer, 1981.

[8] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, v. 40, n. 9, p. 1098–1101, 1952.

[9] SALOMON, D. *Data Compression: The Complete Reference*. 4. ed. [S.l.]: Springer, 2007.