APLICAÇÃO DE PERCURSOS EM ÁRVORES

Principais aplicações em árvores binárias de busca

Uma árvore binária é uma estrutura de dados útil quando precisamos tomar decisões bidirecionais em cada ponto de determinado processo. Muitas aplicações podem ser modeladas por essa estrutura, vejamos a seguir:

Aplicação 1

Neste cenário, suponha que precisamos encontrar todas as repetições em uma lista de números. Uma maneira de fazer isso é comparar cada número com todos que o precedem. Entretanto, isso envolve muitas comparações, que podem ser reduzidas no uso de árvore binária.

Aplicação 2

Neste cenário, em uma aplicação de árvores binárias, temos uma expressão aritmética contendo operandos e operadores binários por uma árvore estritamente binária. Na raiz dessa árvore, conterá um operador que deve ser aplicado aos resultados das expressões representadas pelas subárvores esquerda e direita.

Árvores de expressões aritméticas

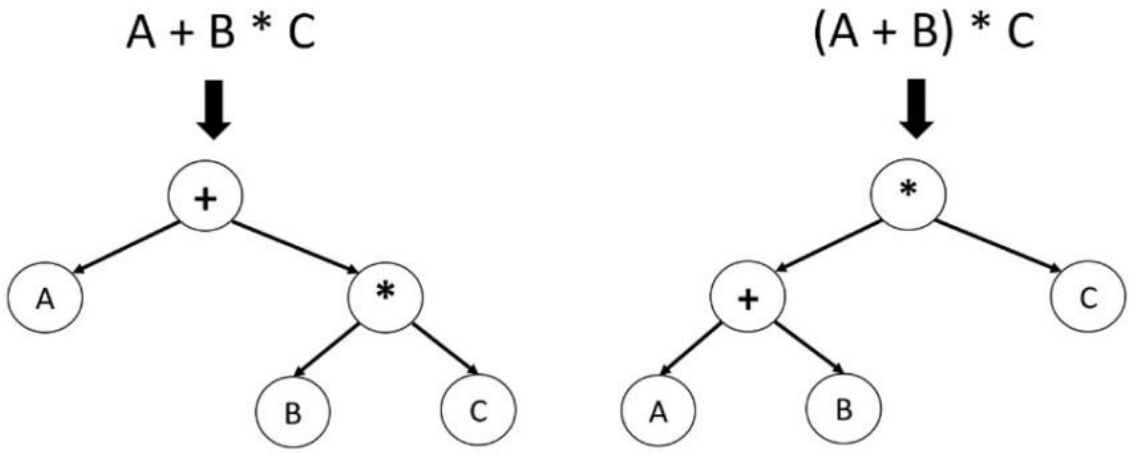
Uma árvore é uma forma natural para representar a estrutura de uma expressão aritmética. Ao contrário de outras notações, a árvore pode representar a computação de forma não ambígua.

Exemplo

A expressão infixa 1 + 2 * 3 é ambígua, a menos que saibamos, de forma antecipada, que a multiplicação é feita antes da adição.

Modelando as expressões aritméticas em árvores, um nó de uma árvore representa um operador, um nó que não é folha, enquanto um nó representando um operando é uma folha.

Vamos considerar a expressão A + B * C. Os operadores + e * ainda aparecem entre os operandos, mas há um problema. Sobre quais operando eles estão atuando? Primeiro, aplicamos o + sobre A e B ou o * sobre B e C? A expressão parece ambígua. Veja:



Vamos interpretar a expressão A + B * C usando a suas precedências. B e C são multiplicados primeiro, em seguida, A é adicionado ao resultado. (A + B) * C forçaria que a adição de A e B fosse realizada primeiro, antes da multiplicação. Na expressão A + B + C, pela associatividade da precedência, o + à esquerda seria feito primeiro; adicionamos A a B e, em seguida, o resultado a C.

Embora tudo isso possa ser óbvio, lembre-se de que os sistemas computacionais precisam saber exatamente quais operadores executar, e em que ordem. Uma maneira de escrever uma expressão que garanta que não haverá confusão alguma com respeito à ordem em que as operações são executadas é a criação de uma expressão totalmente parentizada.

Esse tipo de expressão usa um par de parênteses para cada operador. Os parênteses ditam a ordem em que as operações são executadas e não há ambiguidade. Também não há necessidade de lembrar de regras de precedência.

Expressões prefixas, infixas e pós-fixas

Considere a expressão aritmética infixa A + B. O que acontece se movemos o operador para antes dos dois operandos?

A expressão ficaria + A B. Da mesma forma, poderíamos mover o operador para o fim. Nós obteríamos A B +. Essas expressões parecem estranhas.

A expressão A + B * C + D pode ser reescrita como ((A + (B * C)) + D), para mostrar que a multiplicação acontece primeiro, seguida da adição, mais à esquerda. A + B + C + D pode ser escrito como (((A + B) + C) + D), já que operações de adição são associadas da esquerda para a direita.

Essas mudanças na posição do operador em relação aos operandos criam dois formatos de expressão:

Prefixa

Requer que todos os operadores precedam os dois operandos sobre os quais atuam.

close

Pós-fixa

Requer que seus operadores venham depois dos operandos correspondentes.

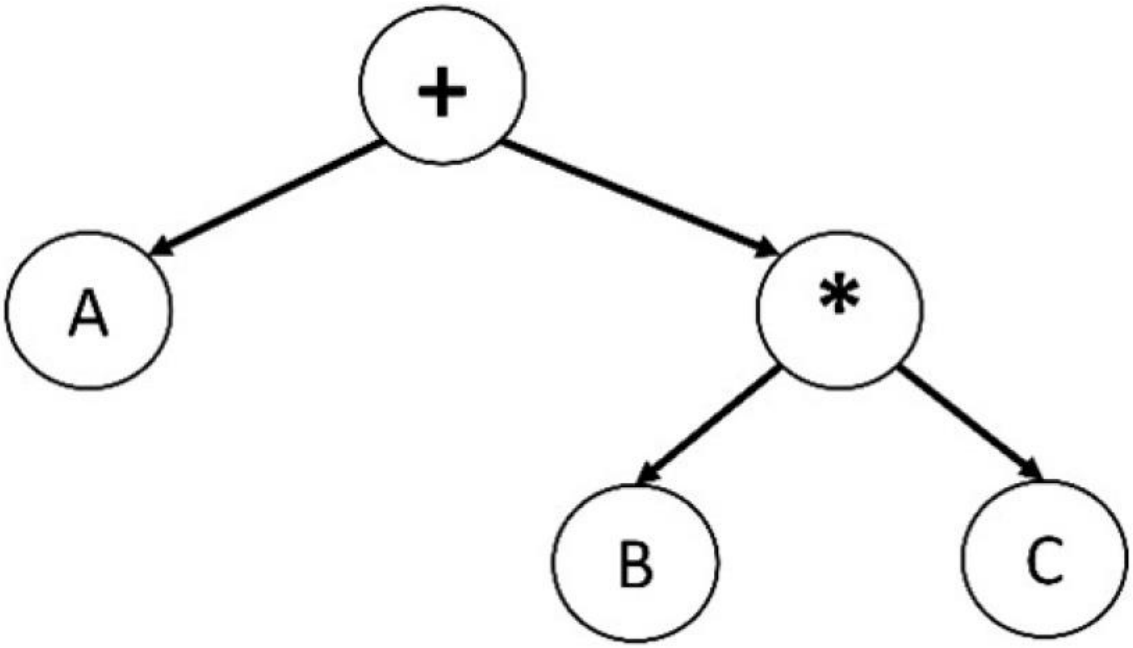
Expressão infixa	Expressão prefixa	Expressão posfixa
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Algoritmos de percursos prefixos, infixos e pós-fixos

Seguem exatamente a mesma filosofia aplicada aos algoritmos de percursos em árvores de busca, ou seja:

1. Se a árvore for lida em ordem raiz-esquerda-direita-raiz, teremos a expressão aritmética em notação prefixa.
2. Se a árvore for lida em ordem esquerda-raiz-direita, teremos a expressão aritmética em notação infixa.
3. Se a árvore for lida em ordem esquerda-direita-raiz, teremos a expressão aritmética em notação pós-fixa.

Para os exemplos da execução, considere a árvore de expressão aritmética:



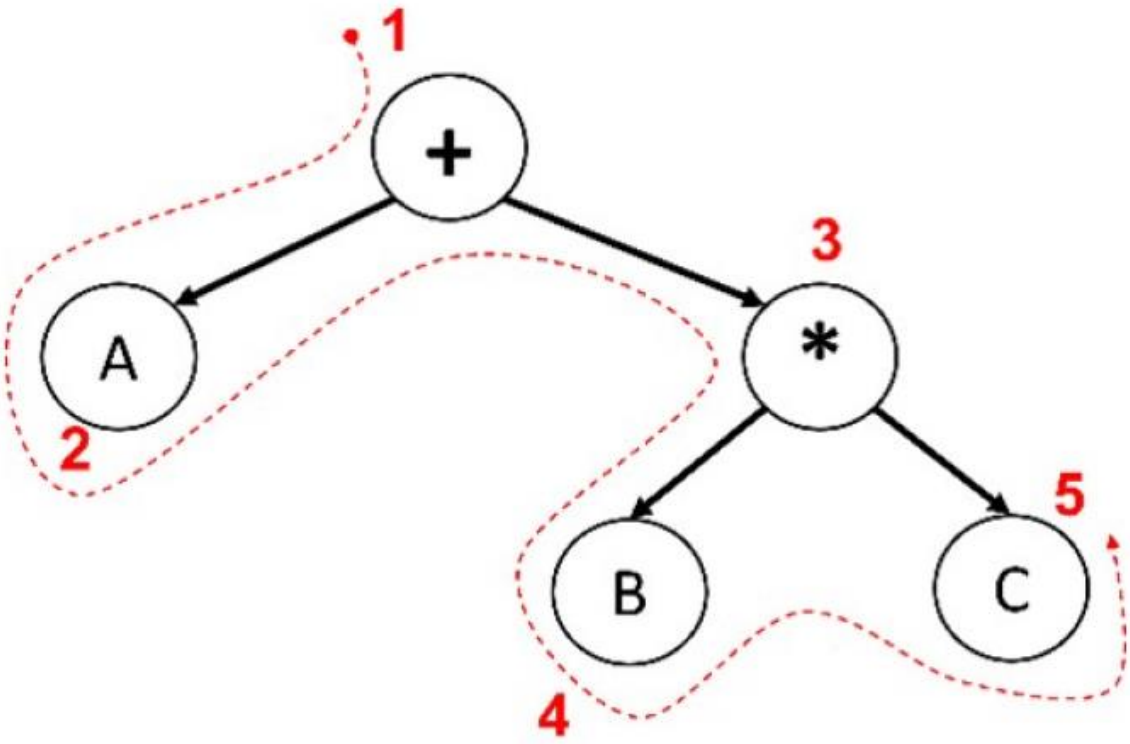
Algoritmo prefixo em Python

Pode ser implementado sob diversas técnicas de programação. A técnica que usaremos aqui é a recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz *r* da árvore *T*, percorre-se a árvore de expressões da seguinte forma:

1. Visita-se a raiz.
2. Percorre-se a subárvore esquerda de *T* em prefixo.
3. Percorre-se a subárvore direita de *T* em prefixo.

Considerando a árvore da expressão da imagem 12, o resultado da visita prefixa é apresentado pela numeração:



Pré-Ordem:

```
def Prefixa(raiz):
  if (raiz):
    print(raiz.chave)
    Prefixa(raiz.esquerda)
    Prefixa(raiz.direita)
```

Assim, a complexidade computacional do percurso prefixo é **O**(**n**), em que **n** é a quantidade de nós na árvore de expressões.

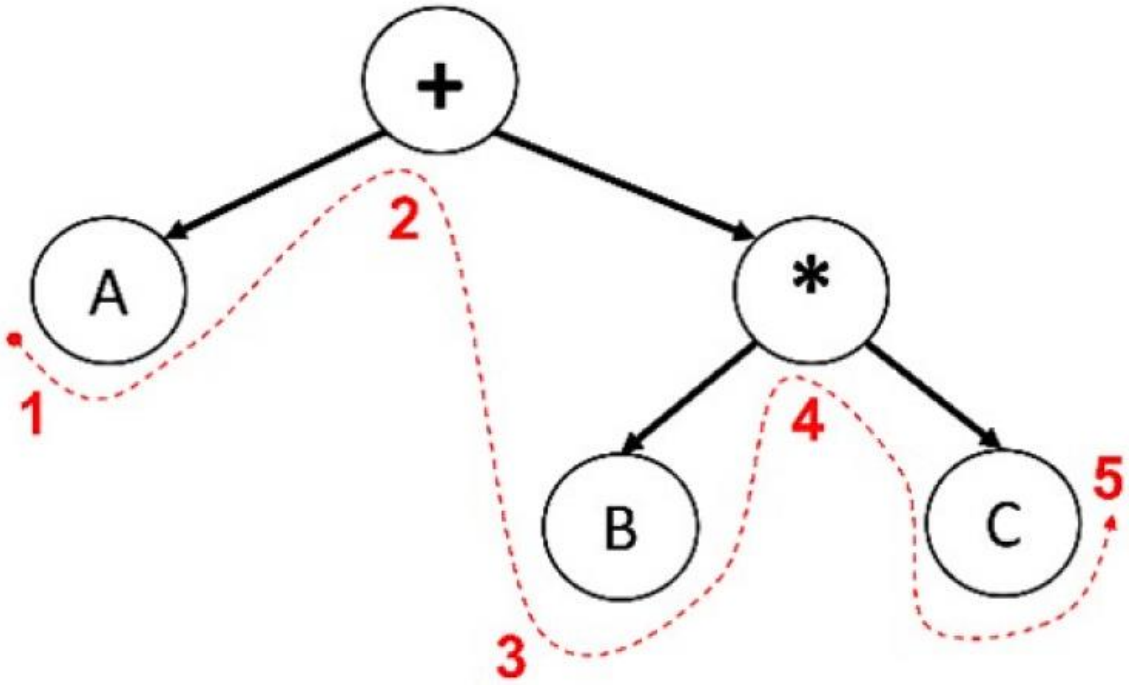
Algoritmo infixo em Python

Pode ser implementado sob diversas técnicas de programação. A técnica que usaremos aqui também é a recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz *r* da árvore *T*, percorre-se a árvore de expressões da seguinte forma:

1. Percorre-se a subárvore esquerda de *T* em ordem simétrica (infixo).
2. Visita-se a raiz.
3. Percorre-se a subárvore direita de *T* em ordem simétrica (infixo).

Considerando a árvore da expressão da imagem 12, o resultado da visita infixa é apresentado pela numeração da:



No algoritmo 12, considerando que as expressões aritméticas estão modeladas em uma árvore binária, a raiz contém o topo da árvore, esquerda e direita são os filhos de um nó. Segue-se a implementação, que é semelhante à aplicada na visita em ordem (simétrica).

```
def Infixo(raiz):
  if (raiz):
    Infixo(raiz.esquerda)
    print(raiz.chave)
    Infixo(raiz.direita)
```

A análise de complexidade é análoga à realizada no algoritmo do percurso prefixo. Observe que a única diferença é a ordem das visitas. Sendo assim, a complexidade computacional do algoritmo para percurso infixo é **O**(**n**), em que **n** é a quantidade de nós na árvore de expressões.

Algoritmo pós-fixo em Python

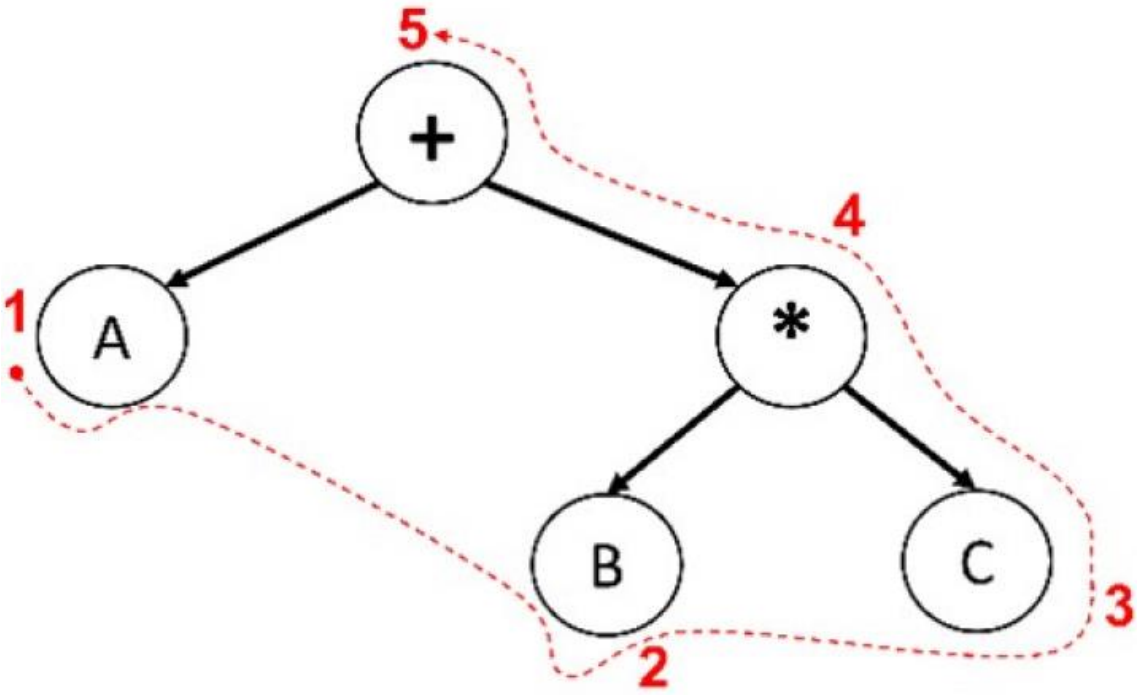
Também usaremos a técnica da recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz r da árvore T, percorre-se a árvore de expressões da seguinte forma:

1. Percorre-se a subárvore esquerda de T em pós-fixo.

2. Percorre-se a subárvore direita de T em pós-fixo.
3. Visita-se a raiz.

Considerando a árvore da expressão da imagem 12, o resultado da visita pós-fixa é apresentado pela seguinte numeração:



No algoritmo 13, considerando que as expressões aritméticas estão modeladas em uma árvore binária, a raiz contém o topo da árvore, esquerda e direita são os filhos de um nó. A implementação é semelhante à aplicada na visita em pós-ordem.

```
def Posfixo(raiz):
    if (raiz):
        Posfixo(raiz.esquerda)
        Posfixo(raiz.direita)
        print(raiz.chave)
```

A análise da complexidade é totalmente análoga à análise feita para prefixo e infixo, o que faz com que o algoritmo tenha complexidade **O**(**n**), em que **n** é a quantidade de nós na árvore de expressões.