

Introducción a programación > 4. Estruturas de datos

4. ESTRUTURAS DE DATOS



Unha **estrutura de datos** é unha maneira organizada de **almacenar e xestionar datos** para que se poidan utilizar de maneira eficiente.

A elección da **estrutura de datos axeitada** pode **influir** moito na eficiencia e no **rendemento** dun algoritmo ou aplicación.

As estruturas de datos permiten realizar operacións como inserción, eliminación, busca e actualización de datos de maneira eficiente.

4.1. PILAS

Unha **pila** (ou **stack** en inglés) é unha estrutura de datos que segue o principio **LIFO** (*Last In, First Out*), que significa que o último elemento que se inserta na pila é o primeiro en saír.

Pódese pensar na pila como unha caixa onde só se pode acceder ao elemento que está no “tope” (ou na parte superior), de maneira semellante a unha pila de libros onde só se pode sacar o libro que está na parte superior.

4.1.1. Características Principais

- **Principio LIFO:** O último elemento que se engade é o primeiro en ser retirado.
- **Acceso restrinxido:** Só se pode acceder ao elemento que está no tope da pila. Non se pode acceder directamente aos elementos no medio ou na parte inferior da pila.

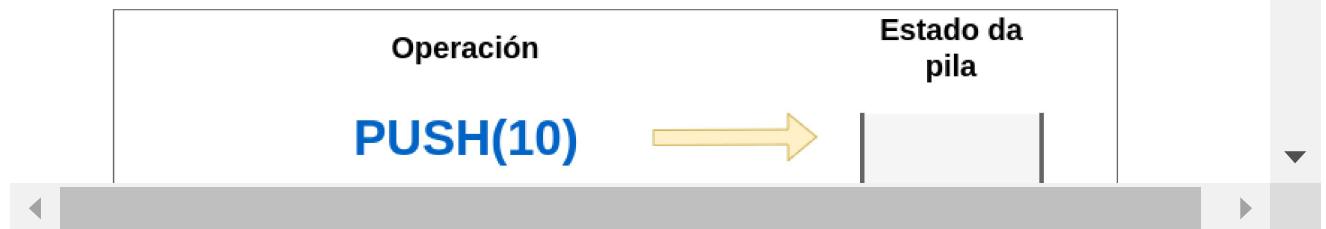
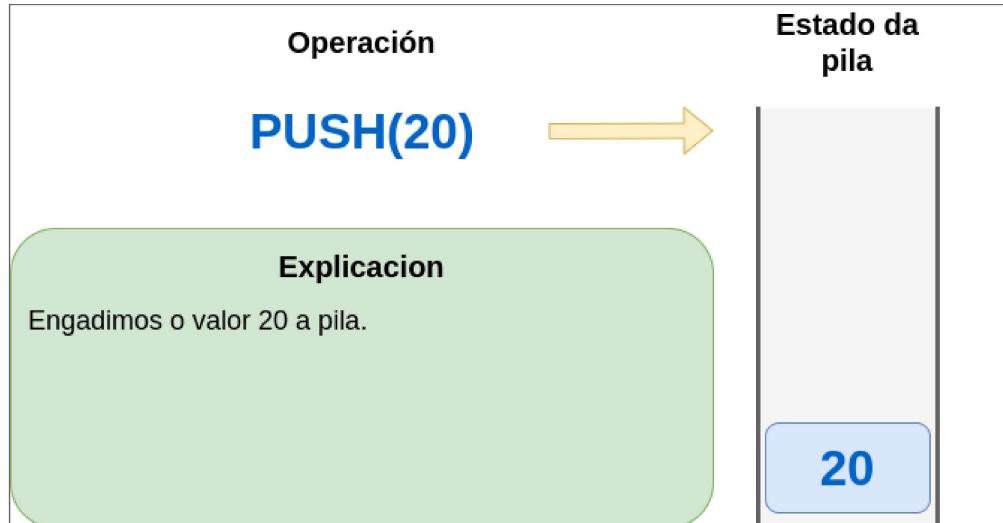
4.1.2. Operacións básicas

- **Push:** Engade un elemento no tope da pila.
- **Pop:** Elimina o elemento no tope da pila e retorna ese elemento.
- **Peek** ou **Top:** Visualiza o elemento no tope da pila sen eliminalo.
- **IsEmpty:** Verifica se a pila está baleira.
- **Size:** Determina o número de elementos na pila.

4.1.3. Usos Comúns das Pilas

- **Desfacer operacións:** En edición de texto ou outros contextos onde se queren desfacer cambios, as pilas permiten reverter as accións na orde correcta.
- **Recursión:** As chamadas a funcións recursivas utilizan a pila de chamadas para almacenar o estado das funcións activas.
- **Algoritmos de Busca:** Algoritmos como a busca en profundidade (DFS) usan pilas para seguir o camiño a través dos nodos.

4.1.4. Exemplo



4.2. COLAS

A **cola** (ou **queue** en inglés) é unha estrutura de datos que segue o principio **FIFO** (First In, First Out), o que significa que o primeiro elemento que se engade á cola é o primeiro en saír.

Pódese comparar cunha fila de persoas onde a primeira persoa en entrar é a primeira en ser atendida.

4.2.1. Características Principais

- **Principio FIFO:** O primeiro elemento que se engade é o primeiro en ser retirado.
- **Acceso Restringido:** Só se pode acceder ao elemento que está no “inicio” da cola para retiralo, e só se pode engadir novos elementos ao “final” da cola.

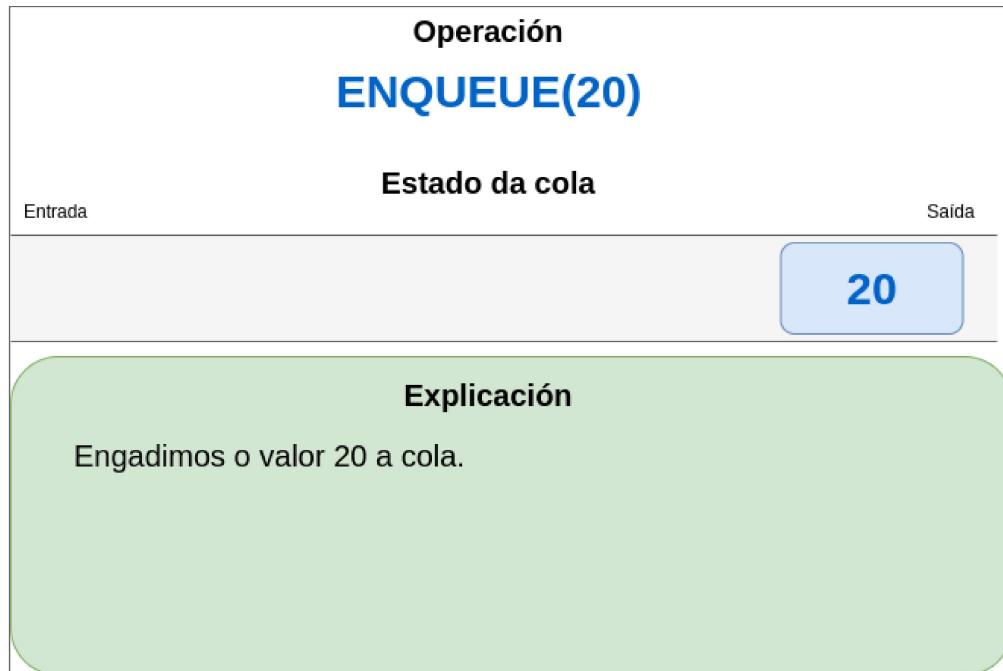
4.2.2. Operacións básicas

- **Enqueue:** Engade un elemento no final da cola.
- **Dequeue:** Elimina o elemento no inicio da cola e retorna ese elemento.
- **Peek ou Front:** Visualiza o elemento no inicio da cola sen eliminarlo.
- **IsEmpty:** Verifica se a cola está baleira.
- **Size:** Determina o número de elementos na cola.

4.2.3. Usos Comúns das Colas

- **Xestión de tarefas:** En sistemas operativos e servidores, as colas son utilizadas para xestionar tarefas e procesos en espera de ser executados.
- **Procesamento de datos:** Colas son útiles para xestionar datos en sistemas de mensaxería ou transmisión de datos, onde os datos se procesan en orde de chegada.
- **Impresión:** As colas se utilizan para xestionar a impresión de documentos en impresoras, onde os documentos se imprimen na orde en que se envían á cola de impresión.

4.2.4. Exemplo



Operación



4.3. LISTAS

Unha lista é unha estrutura de datos que almacena unha colección de elementos. A principal característica das listas é que **os elementos están organizados en orde e se pode acceder a eles mediante índices ou referencia**.

Índices en informática

No ámbito da informática, comézase a contar dende o número 0 e non dende o número 1 como se fai na mayoría de ámbitos.

Polo tanto, o primeiro índice dunha lista non é o índice 0, senón que o **primeiro índice dunha lista é o 0**.

4.3.1. Características Principais

- **Acceso indexado:** Os elementos dunha lista poden ser accedidos por posición ou índice.
- **Tamano dinámico:** A mayoría das implantacións permiten que as listas cambien de tamaño dinamicamente.
- **Orde:** Os elementos están almacenados en orde, e a posición dos elementos é importante.
- **Tipos de elementos:** As listas poden conter elementos do mesmo tipo ou tipos diferentes, dependendo da implantación do linguaxe.

4.3.2. Operacións básicas

As operacións básicas nunha estrutura de datos dunha lista inclúen aquelas que permiten a manipulación e o acceso aos elementos da lista.

- **Append:** Engadir un elemento á lista, xa sexa ao inicio, ao final ou nunha posición específica.
- **Get:** Acceder a un elemento específico na lista sen eliminarlo a partir do seu índice.
- **Remove:** Quitar un elemento da lista, xa sexa ao inicio, ao final ou nunha posición específica.
- **Size:** Determina o número de elementos na lista.
- **IsEmpty:** Verificar se a lista non ten elementos.

Máis operacións sobre listas

- **Modify:** Cambiar o valor dun elemento nunha posición específica.
- **Delete:** Eliminar todos os elementos da lista.
- **Concat:** Unir dúas listas para formar unha soa.
- **Splice:** Obter unha nova lista que é unha subsección da lista orixinal.
- **Search:** Buscar a primeira posición dun elemento dado na lista.

4.3.3. Usos comúns das listas

- **Almacenamiento de datos:** Listas son utilizadas para almacenar e xestionar datos en orde.
- **Manipulación de datos:** Listas permiten realizar operacións como clasificación, filtrado e modificación de datos.

4.3.4. Exemplo

Estado da lista		Operación
Índices	Valores	
0	20	APPEND(20)

Explicación

Engadimos o valor 20 a lista. O seu índice é o 0.

Estado da lista	Operación
-----------------	-----------



4.4. ÁRBORES BINARIAS DE BUSCA

Unha **árbore** é unha estrutura de datos que consiste en nodos conectados entre si de maneira que cada nodo ten un valor e referencias a nodos fillos. A estrutura ten unha raíz e os nodos están organizados de tal maneira que se forman niveis ou capas.

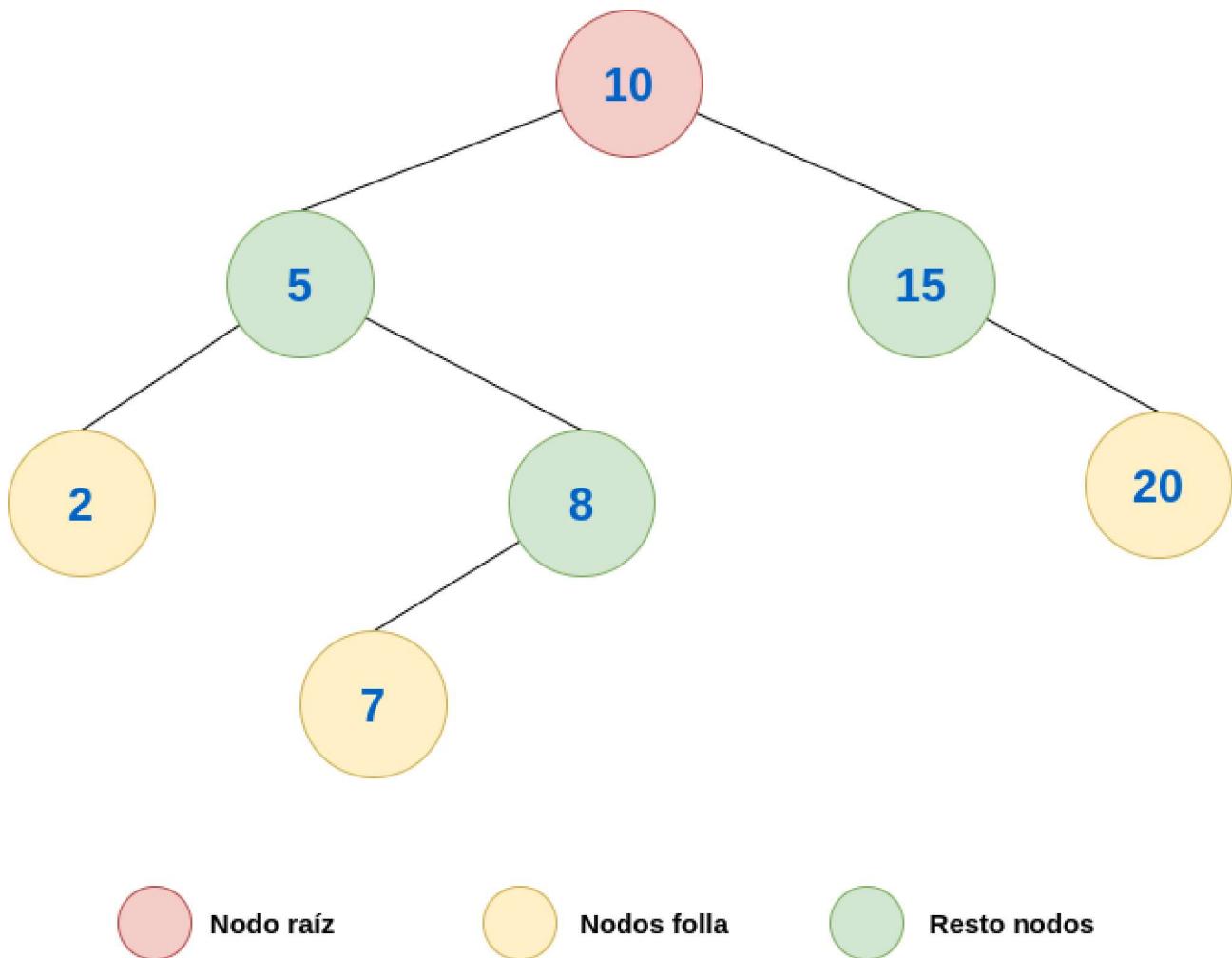
As características principais das árbores son:

- **Nodos:** Elementos da árbore que contén un valor e referencias aos seus nodos fillos.
- **Nodo raíz:** O nodo superior da árbore, que non ten pai.
- **Fillos:** Nodos que están directamente conectados a un nodo pai.
- **Nodos folla:** Nodos que non teñen nodos fillos.
- **Profundidade/Nivel:** O nivel de un nodo é a distancia desde a raíz ata ese nodo.
- **Altura:** A altura da árbore é a profundidade do nodo máis profundo.
- **Subárbore:** unha subárbore nunha árbore binaria de busca é calquera nodo da árbore xunto con todos os seus descendentes. Noutras palabras, unha subárbore é unha árbore que está contida dentro doutra árbore más grande.

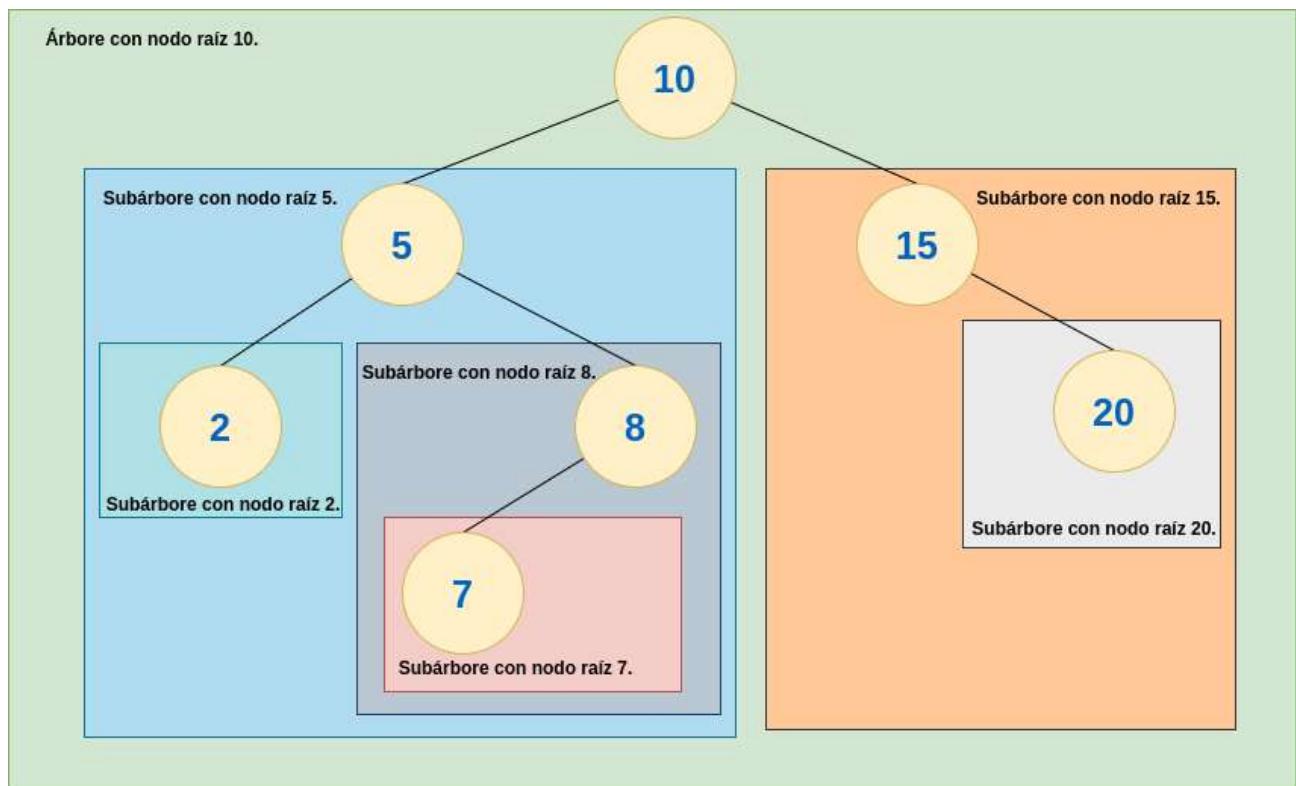
Os tipos de árbores son as seguintes:

- **Árbores Binarias:** As árbores binarias son un tipo específico de árbore onde cada nodo ten como máximo dous fillos: un fillo esquierdo e un fillo derecho.
- **Árbores Binarias de Busca (BST):** Un tipo de árbore binaria onde para cada nodo, todos os valores no subárbore esquierdo son menores e todos os valores no subárbore derecho son maiores.
- **Árbores Binarias Completas:** Cada nivel da árbore está completamente cheo, e todos os nodos son tan completos como sexa posible.
- **Árbores Binarias Balanceadas:** Árbores onde as alturas dos subárbores esquierdo e derecho de cada nodo difiren por como máximo unha unidade.

Nesta imaxe podemos observar un **exemplo de árbore binaria de busca**:



Deste exemplo podemos observar que hai varias **subárbores**. De feito hai unha por cada nodo, na que cada un deles funciona como nodo raíz de cada unha das subárbores.



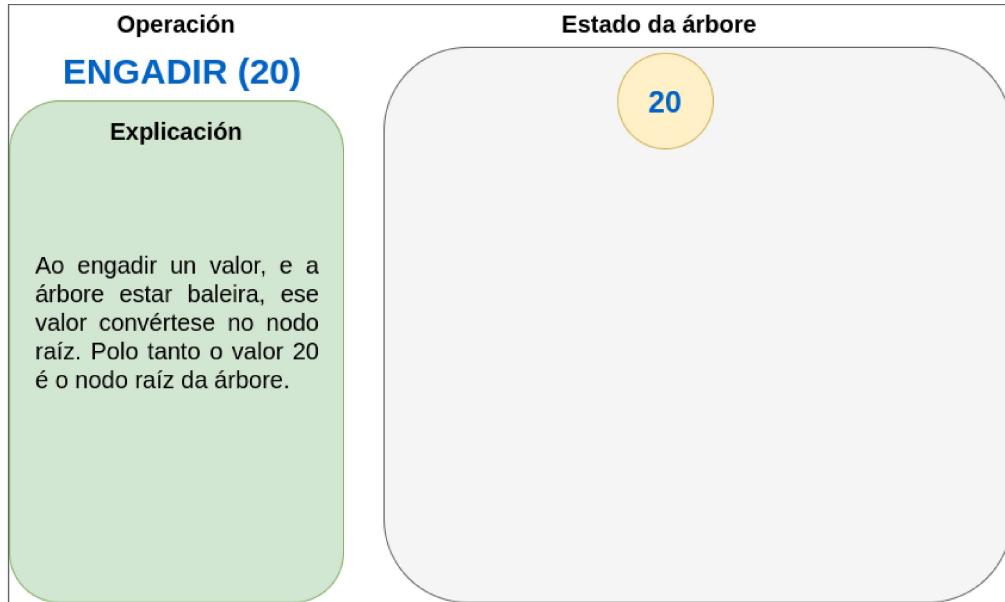
4.4.1. INSERCIÓN DE DATOS

A inserción de datos nunha árbore binaria de busca é un proceso fundamental que se basea en **comparar os valores dos nodos** para manter a estrutura da árbore ordenada.

O proceso de inserción é o seguinte:

1. **Comezo co nodo raíz:** Se a árbore está baleira, o novo nodo convértese na raíz da árbore.
2. **Comparación:** Se a árbore xa ten nodos, comeza pola raíz e compara o valor do novo dato co valor do nodo actual.
3. **Desprazamento á esquerda ou á dereita:**
 - Se o valor do novo dato é menor que o valor do nodo actual, desprazas a comparación cara ao subárbore esquierdo.
 - Se o valor do novo dato é maior que o valor do nodo actual, desprazas a comparación cara ao subárbore derecho.
4. **Nodo folla:** Continúas comparando e desprazando cara á esquerda ou dereita ata chegar a un nodo folla.
5. **Inserción:** Engádese o novo nodo como fillo do nodo folla na posición axeitada (esquerda ou dereita, segundo a última comparación).

4.4.1.1. Exemplo



4.4.2. PERCORRIDOS

Os **percorridos** das árbores binarias de busca (BST) son **técnicas para visitar todos os nodos da árbore de diferentes maneiras**.

Estes percorridos son útiles para realizar operacións como listar elementos, buscar datos e modificar a estrutura da árbore.

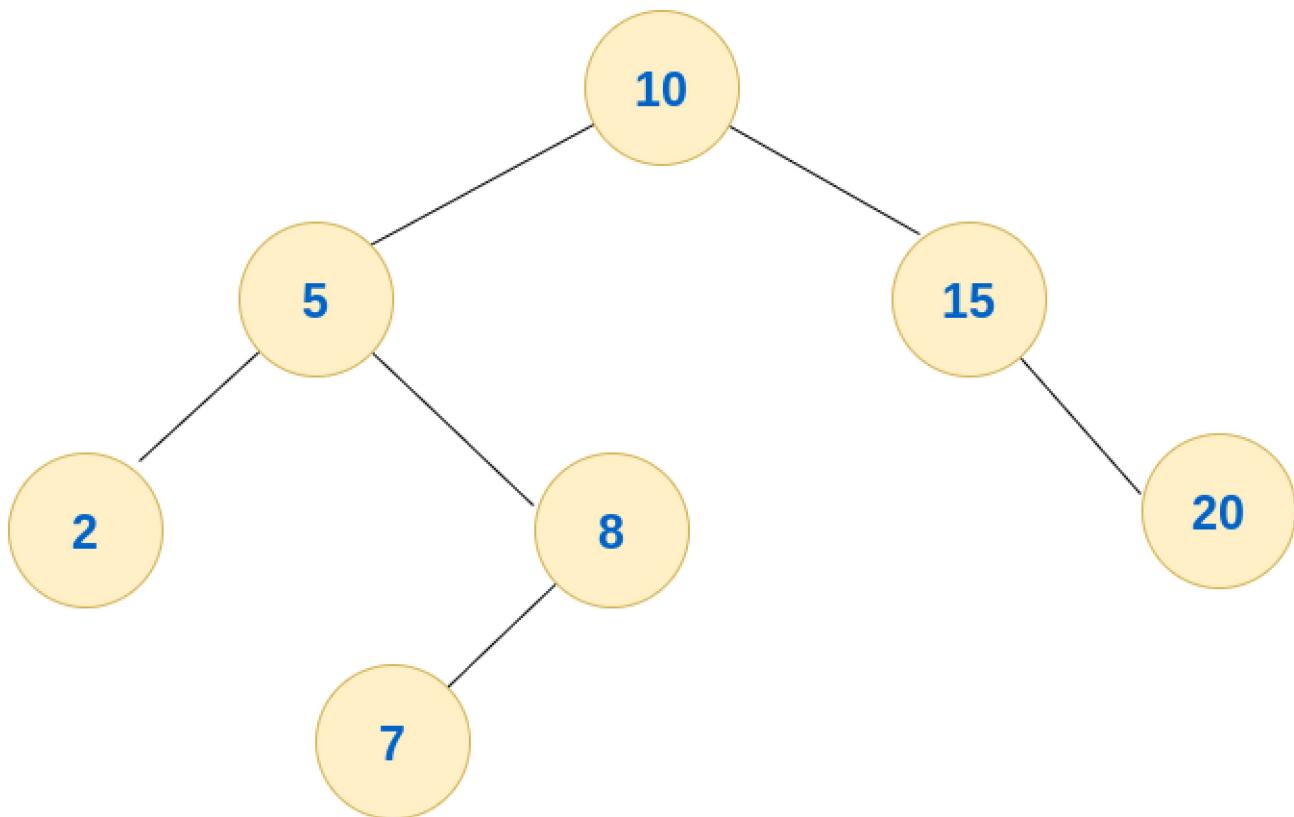
Os tres principais tipos de percorridos para árbores binarias son:

4.4.2.1. Percorrido preorde

É útil cando se necesita procesar a raíz antes dos seus nodos fillos, como na copia dunha árbore ou na serialización da estrutura da árbore. O percorrido preorde visita os nodos da árbore na seguinte orde:

1. **Imprimir nodo raíz**
2. **Consultar subárbore esquerda**
3. **Consultar subárbore dereita**

Exemplo



1. Imprimimos o nodo raíz: **10**.
2. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 5)
3. Imprimimos o nodo raíz desta subárbore: **5**.
4. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 2)
5. Imprimimos o nodo raíz desta subárbore: **2**.
6. Non ten subárbore esquerda.
7. Non ten subárbore dereita.
8. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
9. Imos a subárbore dereita. (A que ten como nodo raíz o valor 8)
10. Imprimimos o nodo da raíz desta subárbore: **8**.

11. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 7)
12. Imprimimos o nodo da raíz desta subárbore: **7**.
13. Non ten subárbore esquerda.
14. Non ten subárbore dereita.
15. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 8)
16. Non ten subárbore dereita.
17. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
18. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
19. Imos a subárbore da dereita (A que ten como nodo raíz o valor 15)
20. Imprimimos o nodo da raíz desta subárbore: **15**.
21. Non ten subárbore esquerda.
22. Imos a subárbore dereita. (A que ten como nodo raíz 20)
23. Imprimimos o nodo raíz desta subárbore: **20**.
24. Non ten subárbore a esquerda.
25. Non ten subárbore a dereita.
26. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 15)
27. Non ten subárbore dereita.
28. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
29. Éste é o nodo raíz, polo que damos por concluído o recorrido.

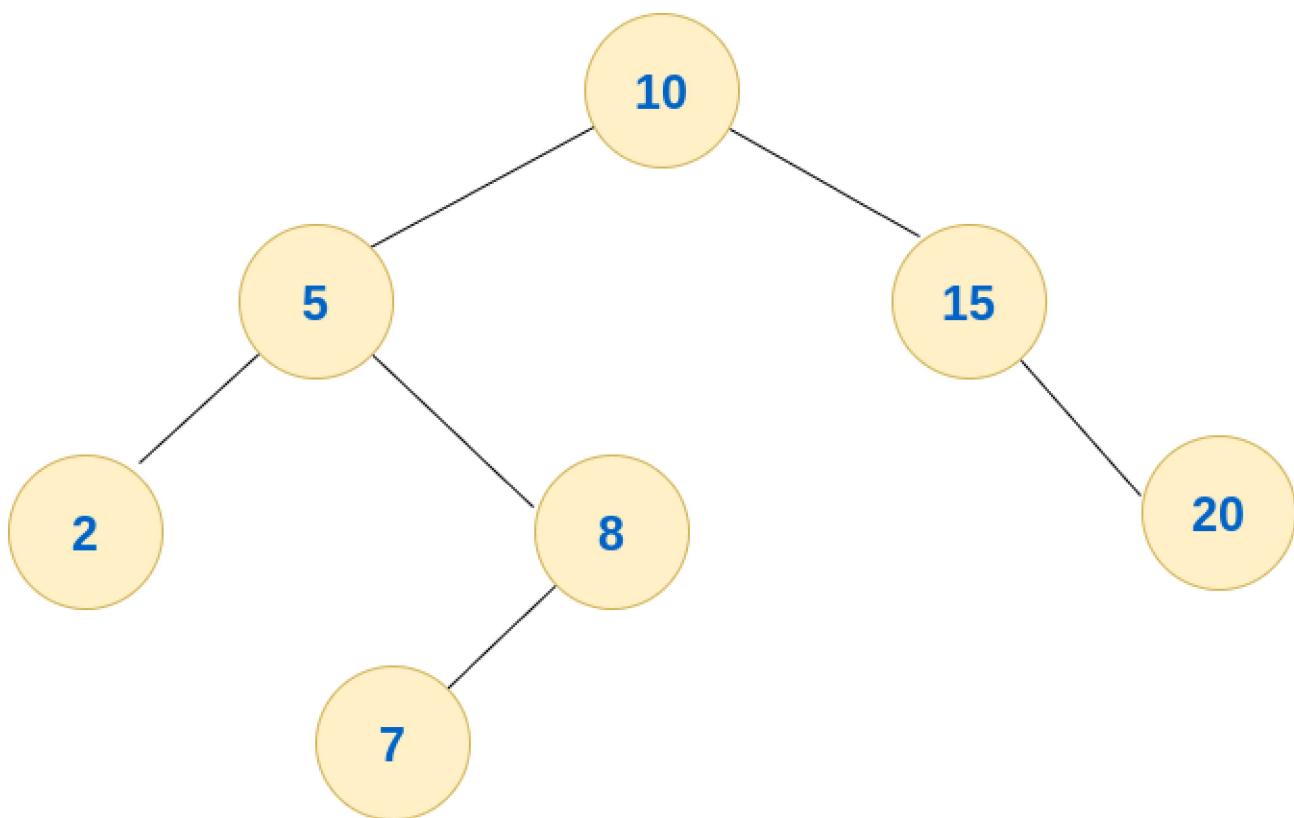
Polo tanto o recorrido preorde queda do seguinte xeito: **10 - 5 - 2 - 8 - 7 - 15 - 20**.

4.4.2.2. Percorrido Inorde

Especialmente útil en árbores binarias de busca (BST) porque **visita os nodos en orde ascendente**. Permite obter os elementos da árbore en **orde ordenada**. O percorrido inorde visita os nodos da árbore na seguinte orde:

1. **Consultar subárbore esquerda**
2. **Imprimir nodo raíz**
3. **Consultar subárbore dereita**

Exemplo



1. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 5)

2. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 2)
3. Non ten subárbore esquerda.
4. Imprimimos o nodo da raíz desta subárbore: **2**.
5. Non ten subárbore a dereita.
6. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
7. Imprimimos o nodo da raíz desta subárbore: **5**.
8. Imos a subárbore da dereita (A que ten como nodo raíz o valor 8)
9. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 7)
10. Non ten subárbore esquerda.
11. Imprimimos o nodo da raíz desta subárbore: **7**.
12. Non ten subárbore a dereita.
13. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 8)
14. Imprimimos o nodo da raíz desta subárbore: **8**.
15. Non ten subárbore a dereita.
16. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
17. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
18. Imprimimos o nodo da raíz desta subárbore: **10**.
19. Imos a subárbore da dereita (A que ten como nodo raíz o valor 15)
20. Non ten subárbore esquerda.
21. Imprimimos o nodo da raíz desta subárbore: **15**.
22. Imos a subárbore da dereita (A que ten como nodo raíz o valor 20)
23. Non ten subárbore esquerda.
24. Imprimimos o nodo da raíz desta subárbore: **20**.
25. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 15)
26. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
27. Éste é o nodo raíz, polo que damos por concluído o recorrido.

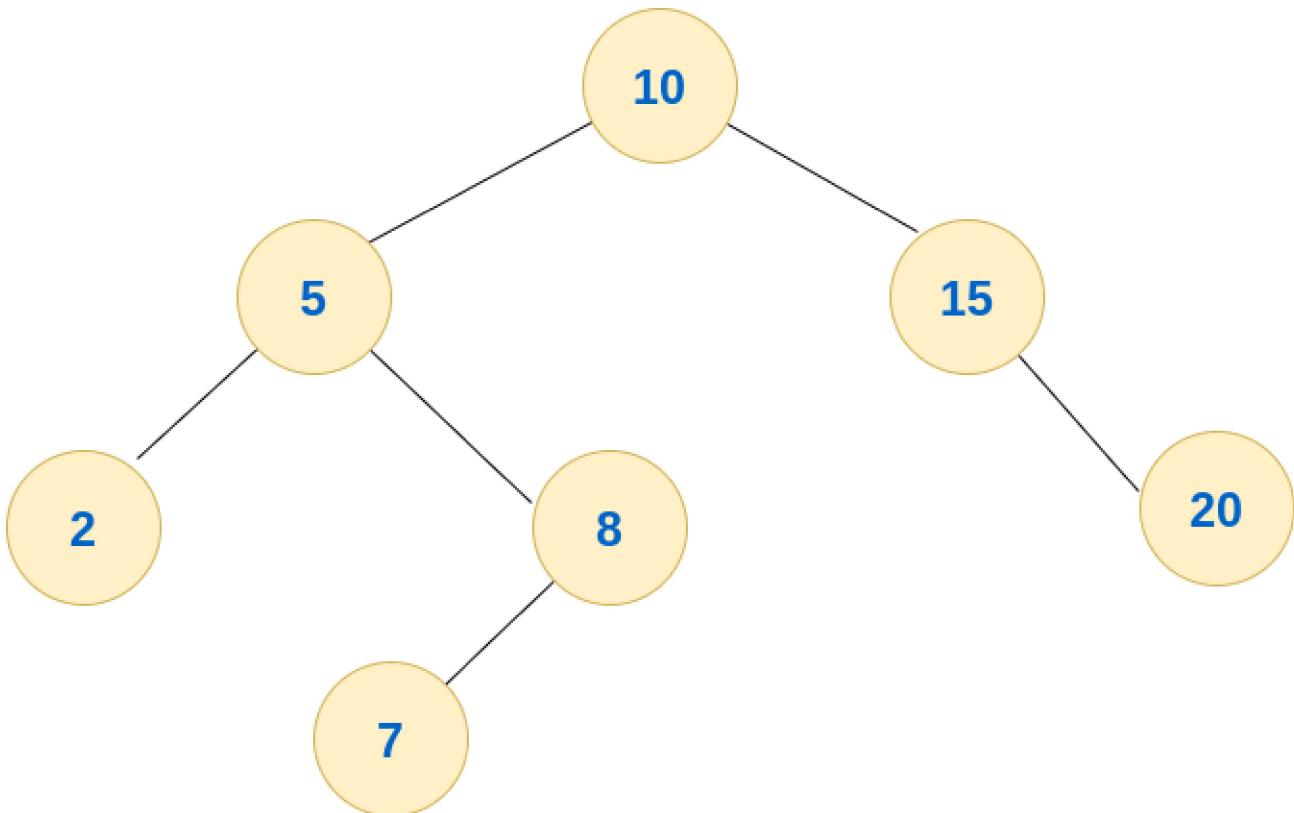
Polo tanto o recorrido inorde queda do seguinte xeito: **2 - 5 - 7 - 8 - 10 - 15 - 20**.

4.4.2.3. Percorrido Postorde

Útil para procesos que necesitan asegurar que se procesen todos os nodos fillos antes de procesar a raíz, como na eliminación de nodos ou na liberación de memoria. O percorrido postorde visita os nodos da árbore na seguinte orde:

1. ***Consultar subárbore esquerda**
2. **Consultar subárbore dereita**
3. **Imprimir nodo raíz**

Exemplo



1. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 5)
2. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 2)
3. Non ten subárbore esquerda.
4. Non ten subárbore a dereita.
5. Imprimimos o nodo da raíz desta subárbore: **2**.
6. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
7. Imos a subárbore da dereita (A que ten como nodo raíz o valor 8)
8. Imos a subárbore esquerda. (A que ten como nodo raíz o valor 7)
9. Non ten subárbore esquerda.
10. Non ten subárbore a dereita.
11. Imprimimos o nodo da raíz desta subárbore: **7**.
12. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 8)
13. Non ten subárbore a dereita.
14. Imprimimos o nodo da raíz desta subárbore: **8**.
15. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 5)
16. Imprimimos o nodo da raíz desta subárbore: **5**.
17. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
18. Imos a subárbore da dereita (A que ten como nodo raíz o valor 15)
19. Non ten subárbore esquerda.
20. Imos a subárbore da dereita (A que ten como nodo raíz o valor 20)
21. Non ten subárbore esquerda.
22. Non ten subárbore a dereita.
23. Imprimimos o nodo da raíz desta subárbore: **20**.
24. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 15)
25. Imprimimos o nodo da raíz desta subárbore: **15**.
26. Subimos un nivel. (Á subárbore que ten como nodo raíz o valor 10)
27. Imprimimos o nodo da raíz desta subárbore: **10**.
28. Éste é o nodo raíz, polo que damos por concluído o recorrido.

Polo tanto o recorrido postorde queda do seguinte xeito: **2 - 7 - 8 - 5 - 20 - 15 - 10**.

4.4.3. UTILIDADES

- **Busca Rápida:** Árbores binarias de búsqueda (BST) permiten buscar elementos de forma eficiente.
 - **Xestión de Datos:** Usadas para manter datos ordenados e facilitar operacións como inserción e eliminación.
-

4.5. TÁBOAS HASH

Unha **táboa hash**; tamén coñecidas como **conjunto de chaves-valor**, **dicionario** ou **array asociativo**, é unha estrutura de datos que **permite almacenar e recuperar valores de forma eficiente utilizando chaves**.

Utiliza unha **función de hash** para mapear as chaves a índices nunha lista, facilitando o acceso directo aos valores asociados. A función hash toma a chave como entrada e produce un número enteiro que normalmente se modula polo tamaño da lista asociada a *hash table* para obter un índice válido.

Función hash

Unha función de *hash* é un **algoritmo que toma una entrada** e xérase un valor de lonxitude fixa, normalmente representado por unha cadea de caracteres (como un número hexadecimal). Este valor, chamado *hash*, representa un resumo dos datos de entrada.

As principais características dunha función de hash son:

- **Determinística:** Para unha entrada dada, a función de *hash* sempre producirá o mesmo valor de *hash*. Se a entrada cambia áinda que sexa mínimamente, o valor de *hash* tamén cambiará de maneira significativa.
- **Preimaxé resistente:** Non debería ser posible (ou debería ser extremadamente difícil) descubrir a entrada orixinal a partir do valor *hash*.
- **Resistencia a colisións:** Dúas entradas diferentes deberían producir valores de *hash* diferentes. As boas funcións de hash fan que sexa moi pouco probable atopar dúas entradas diferentes co mesmo valor de *hash*.

As funcións de hash utilizanse en moitas aplicacións, como a verificación da integridade de datos, as contrasinais, a indexación de datos en bases de datos, a criptografía, entre outros.

As **colisións** acontecen cando dúas chaves diferentes producen o mesmo índice. Hai varios métodos para manexar colisións.

4.5.1. Características Principais

- **Función de Hash:** Transforma a chave nun índice que se usa para almacenar ou recuperar o valor asociado.
- **Acceso Rápido:** Permite un acceso medio en $O(1)$ para operacións de inserción, eliminación e busca, debido á función de *hash*.
- **Colisións:** Pode ocorrer cando diferentes chaves se mapean ao mesmo índice. As colisións son xestionadas mediante técnicas específicas.

4.5.2. Operacións básicas

As operacións básicas nunha estrutura de datos dunha táboa hash inclúen aquellas que permiten a manipulación e o acceso aos elementos da táboa hash.

- **Insert(Chave, valor):** Engadir un par chave-valor á táboa *hash*. Se a chave xa existe, actualizar o valor asociado a esa chave.
- **Get:** Obter o valor asociado a unha chave específica.

- **ContainsKey**: Verificar si una clave existe en el diccionario hash.
- **Delete**: Quitar un par clave-valor del diccionario hash utilizando la clave.
- **Size**: Determinar el número de pares clave-valor en el diccionario hash.
- **IsEmpty**: Verificar si el diccionario hash no tiene pares clave-valor.

Más operaciones sobre tablas hash

- **KeySet**: Obtener todas las claves en el diccionario hash.
- **Values**: Obtener todos los valores en el diccionario hash.

4.5.3. Usos comunes de las tablas hash

- **Indexación rápida**: Para realizar búsquedas rápidas en bases de datos o estructuras de datos.
- **Xestión de datos**: Para almacenar e recuperar datos asociados a claves, como en cachés e diccionarios.

4.5.4. Técnica de Hashing por Folding (Folding Method)

A técnica de **folding** es un método para crear una función hash a partir de una clave que es un número largo o una cadena de caracteres. Esta técnica divide la clave en partes más pequeñas, realiza algunas operaciones sobre esas partes, y finalmente combina esos resultados para generar el hash final.

Hay varias variantes de la técnica de folding, pero la más común implica los siguientes pasos:

1. **Dividir la clave en partes iguales**: La clave se divide en partes más pequeñas, que tienen el mismo número de dígitos o caracteres.
2. **Suma de las partes**: Despues de dividir la clave, se suman todas las partes. Ejemplo:

$$123 + 456 + 789 + 000 = 1368$$
3. **Aplicar modulación (opcional)**: Si el tamaño de la tabla hash es m , aplicamos la operación de módulo m al resultado de la suma para asegurarnos de que el índice resultante está dentro del rango de la tabla hash.

A continuación, vamos a utilizar la cadena de texto "ABCDEFGHI" como clave y aplicar el método de **folding** para calcular su hash.

- 1. **Convertir la Cadea en Números**: Primero, convertiremos cada carácter de la cadena en un número utilizando su valor ASCII (o otro esquema de conversión similar).
 - A = 65
 - B = 66
 - C = 67
 - D = 68
 - E = 69
 - F = 70
 - G = 71
 - H = 72

Por lo tanto, la cadena "ABCDEFGHI" puede ser representada como una secuencia numérica:

$65 \ 66 \ 67 \ 68 \ 69 \ 70 \ 71 \ 72$.

- 2. **Dividir la Secuencia en Partes**: Ahora, dividiremos esta secuencia en partes iguales. Supongamos que dividimos en partes de dos dígitos.
- 3. **Suma de las Partes**: Despues de dividir, sumaremos todas las partes.

$65 + 66 + 67 + 68 + 69 + 70 + 71 + 72 = 548$

- 4. **Aplicar Modulación (opcional)** Finalmente, se estamos a traballar cunha táboa hash que ten un tamaño fixo, aplicamos a operación de módulo m para asegurar que o resultado se atopa dentro do rango da táboa. Se o tamaño da táboa hash é 100, entón calculamos $548 \% 100 = 48$. O índice final na táboa hash sería 48.

4.5.5. Exemplo

Consideracións do exemplo

- Neste exemplo vamos supoñer que non acontecen colisións.
- Utilizaremos a técnica de Hashing por Folding dividindo en 3 números.
- O tamaño da *hash table* será de 5.

Estado da lista		Operación
Índices	Valores	
0		
1		
2	20	INSERT(DAW, 20)
3		
4		

Explicación

Calculamos o hash de "DAW":

1. Collemos os códigos ASCII: 68 65 87.
2. Dividimos en partes: 685 e 587.
3. Sumamos as partes: 1272
4. Calculamos o resto de dividir entre 5: 2.

Polo tanto almacenaremos o valor 20 no índice 2.

Estado da lista

Operación

INSERT(DAW, 20)