Estructuras de datos no lineales implementadas en librerías

Miguel Ortiz

Programación competitiva para ICPC

Octubre 2023

Introducción

- Una estructura de datos es una forma de almacenar datos en una computadora
- Es importante elegir la estructura de datos apropiada para cada problema
- Estructuras de datos sofisticadas y poderosas ya están implementadas en los lenguajes más populares
- Veremos algunas de las más importantes presentes en C++

Arreglos dinámicos - lineal

- Un arreglo dinámico puede cambiar su tamaño durante la ejecución del programa
- `vector` en C++
- Puede usarse casi como un arreglo normal

Arreglos dinámicos - lineal

Crear un vector vacío y añadir elementos:

```
#include <vector>
...
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3, 2]
v.push_back(5); // [3, 2, 5]
```

Arreglos dinámicos - lineal

Crear un vector vacío y añadir elementos:

```
#include <vector>
...
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3, 2]
v.push_back(5); // [3, 2, 5]
```

Se puede acceder a los elementos como en un arreglo normal:

```
cout << v[0] << '\n'; // 3
cout << v[1] << '\n'; // 2
cout << v[2] << '\n'; // 5
```

Arreglos dinámicos - lineal

La función `size` devuelve el tamaño del vector:

```
for (int i = 0; i < v.size(); i++) {
  cout << v[i] << " ";
}
// 3 2 5
```

Una forma más corta de iterar sobre los elementos:

```
for (auto x : v) {
  cout << x << " ";
}
// 3 2 5
```

Arreglos dinámicos - lineal

Inicializar un vector con 5 elementos:

```
vector<int> v = {3,2,5,1,4};
```

También se puede inicializar con un tamaño y un valor en cada posición:

```
// tamaño 10, lleno de 0s
vector<int> v(10);

// tamaño 10, lleno de -1s
vector<int> v(10, -1);

// tamaño 5, lleno de vectores vacios
vector<vector<int>> v(5, vector<int>());
```

Arreglos dinámicos - lineal

Otras funciones:

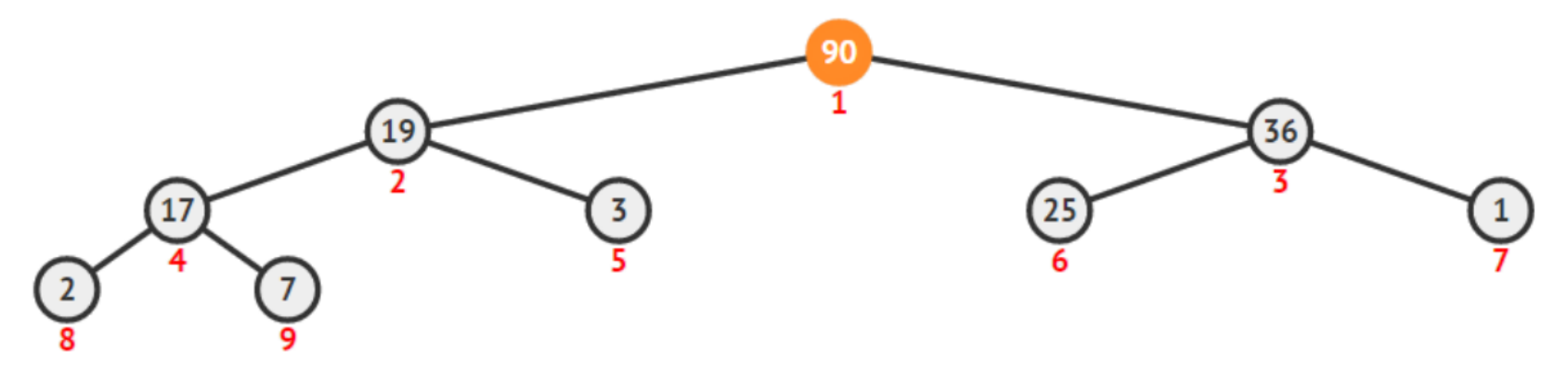
- `v.pop_back()`: elimina el último elemento
- `v.resize(n)`: cambia el tamaño del vector a *n*
- `v.assign(n, val)`: cambia el tamaño del vector a *n* y llena todas las posiciones con *val*
- `v.clear()`: elimina todos los elementos del vector

Ordenar un vector:

```
#include <algorithm>
...
sort(v.begin(), v.end());
```

Binary Heap - Introducción

- Organiza los datos en un árbol binario *completo*
- Por cada nodo intermedio, el valor del nodo es mayor o igual que el valor de sus hijos
- El nodo raíz tiene el mayor valor de todos
- Los valores $\{1, 2, 3, 7, 17, 19, 25, 36, 90\}$ pueden generar el siguiente binary heap:



Binary Heap - Cola de prioridad

- Operaciones
  - Insertar un elemento (`push`) en $O(\log n)$
  - Obtener el elemento con mayor valor (`top`) en $O(1)$
  - Eliminar el elemento con mayor valor (`pop`) en $O(\log n)$

- Nosotros definimos el tipo de dato que almacena el binary heap
- `priority_queue<int> pq;`
- `priority_queue<pair<int, int>> pq;`

Binary Heap - Cola de prioridad

- push, top y pop:

```
#include <queue>
...
priority_queue<int> pq;
pq.push(3);
pq.push(2);
pq.push(5);
pq.push(3);
cout << pq.top() << '\n'; // 5
pq.pop();
cout << pq.top() << '\n'; // 3
```

Binary Heap - Cola de prioridad

- Manatener un binary heap con el menor elemento en el tope:

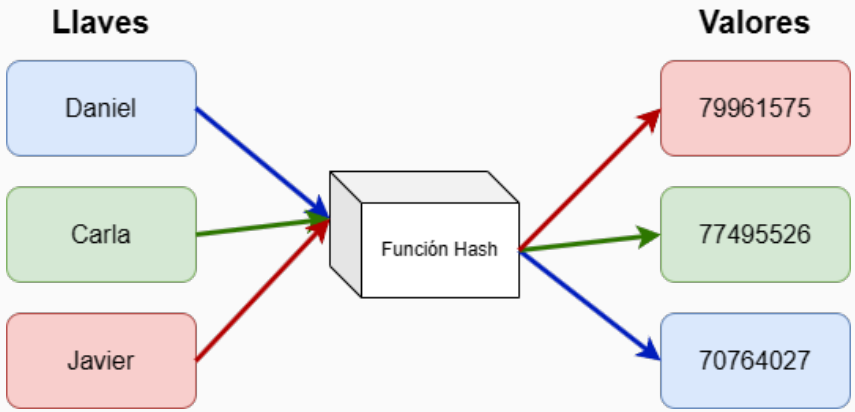
```
...
priority_queue<int, vector<int>, greater<int>> pq;
pq.push(3);
pq.push(2);
pq.push(5);
pq.push(3);
cout << pq.top() << '\n'; // 2
pq.pop();
cout << pq.top() << '\n'; // 3
```

Hash Table

- Estructura de datos que almacena pares de elementos (llave, valor)
- Usamos la llave para acceder al valor
- Se usa `unordered_map<tipo_llave, tipo_valor>` cuando no importa el orden de los elementos
- Se usa `unordered_set<tipo_llave>` para verificar la existencia de llaves cuando no importa el orden de los elementos

Hash Table

- Estructura de datos que almacena pares de elementos (llave, valor)
- Usamos la llave para acceder al valor
- Se usa `unordered_map<tipo_llave, tipo_valor>` cuando no importa el orden de los elementos
- Se usa `unordered_set<tipo_llave>` para verificar la existencia de llaves cuando no importa el orden de los elementos
- Insertar, buscar/acceder y eliminar un elemento es $O(1)$ usando una función hash



Hash Table - unordered_map

- insert, count, erase y clear en unordered_map:

```
#include <unordered_map>
...
unordered_map<string, int> m;
m["mono"] = 3;
m["gato"] = 5;
cout << m["mono"] << '\n'; // 3
if (!m.count("tecla")) {
  // llave no existe
}
cout << m["tecla"] << '\n'; // 0
if (m.count("tecla")) {
  // llave existe
}
m.erase("mono");
m.clear();
```

Hash Table - unordered_set

- insert, count, erase y clear en unordered_set:

```
#include <unordered_set>
...
unordered_set<string> s;
s.insert("mono");
s.insert("gato");
cout << s.count("mono") << '\n'; // 1
cout << s.count("tecla") << '\n'; // 0
s.erase("mono");
cout << s.size() << '\n'; // 1
s.clear();
```

Hash Table

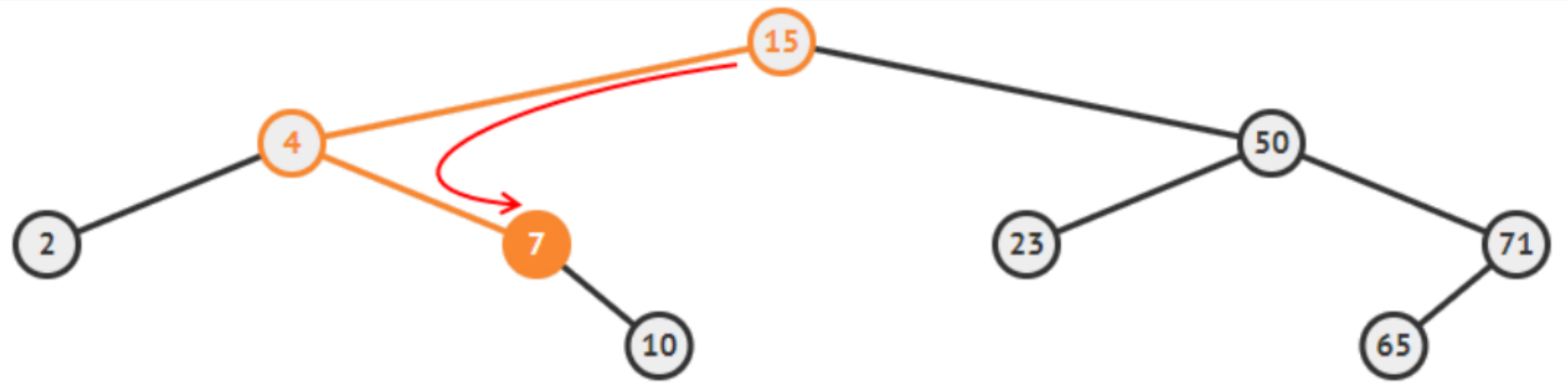
- Recorrer un unordered_map y un unordered_set:

```
unordered_map<string, int> m;
m["mono"] = 3; m["gato"] = 5;
for (auto p : m) {
  cout << p.first << " " << p.second << '\n';
}
// mono 3
// gato 5

unordered_set<string> s;
s.insert("arbol"); s.insert("planta");
for (auto x : s) { cout << x << '\n'; }
// planta
// arbol
```

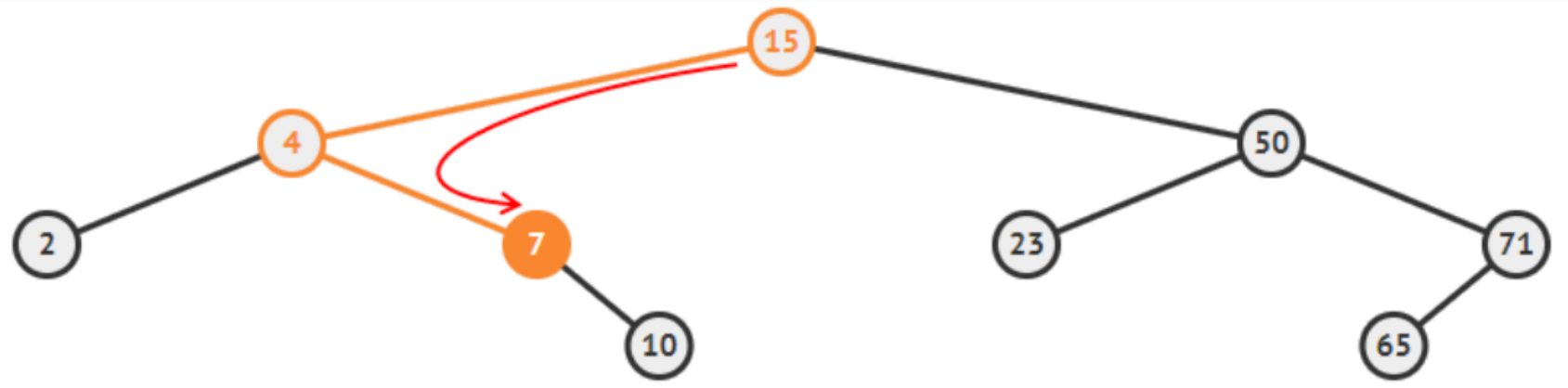
Binary Search Tree

- Similarmente al binary heap, organiza los datos en un árbol binario
- Por cada nodo intermedio, el valor del nodo es mayor que el valor de su hijo izquierdo y menor que el valor de su hijo derecho



Binary Search Tree

- Similarmente al binary heap, organiza los datos en un árbol binario
- Por cada nodo intermedio, el valor del nodo es mayor que el valor de su hijo izquierdo y menor que el valor de su hijo derecho



- Permite insertar, buscar y eliminar un elemento en $O(\log n)$ si el árbol está *balanceado*
- C++ tiene dos implementaciones: `set` y `map`

Binary Search Tree - `set`

- Un `set` almacena un conjunto de elementos (sin repeticiones)
- Se puede insertar, eliminar y verificar si un elemento está en el conjunto
- Al ser un árbol binario de búsqueda, mantiene los elementos ordenados

Binary Search Tree - set

- insert, count y erase:

```cpp
#include <set>
...
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
s.insert(3); // no se inserta
cout << s.count(3) << '\n'; // 1
cout << s.count(4) << '\n'; // 0
cout << s.size() << '\n'; // 3
s.erase(3);
cout << s.count(3) << '\n'; // 0
cout << s.size() << '\n'; // 2
```

Binary Search Tree - `set`

- Recorrer un `set`:

```
set<int> s = {2, 4, 1, 6, 1};
for (auto x : s) {
  cout << x << " ";
}
// 1 2 4 6
```

Binary Search Tree - `set`

- `set` tambien permite hallar el elemento más cercano a un valor dado
- `lower_bound` devuelve un iterador (puntero) al primer elemento mayor o igual que el valor dado
- `upper_bound` devuelve un iterador (puntero) al primer elemento mayor que el valor dado

```
set<int> s = {2, 4, 1, 6, 1};

auto it_1 = s.lower_bound(3);
cout << *it_1 << '\n'; // 4

auto it_2 = s.upper_bound(4);
cout << *it_2 << '\n'; // 6
```

Binary Search Tree - `set`

- Se puede mover un iterador a la izquierda o derecha usando los operadores `--` y `++`

```
set<int> s = {2, 4, 1, 6, 1};
auto it = s.upper_bound(3); // apunta a 4
it++; // apunta a 6
cout << *it << '\n'; // 6
it--; // apunta a 4
it--; // apunta a 2
cout << *it << '\n'; // 2
```

Binary Search Tree - `set`

- Se puede mover un iterador a la izquierda o derecha usando los operadores `--` y `++`

```
set<int> s = {2, 4, 1, 6, 1};
auto it = s.upper_bound(3); // apunta a 4
it++; // apunta a 6
cout << *it << '\n'; // 6
it--; // apunta a 4
it--; // apunta a 2
cout << *it << '\n'; // 2
```

- Podemos usar esto para hallar el elemento más cercano que sea menor o menor-igual que un valor dado

Binary Search Tree - `set`

```cpp
set<int> s = {2, 4, 1, 6, 1};

// Elemento menor o igual a 4
auto it_1 = s.upper_bound(4);
it--;
cout << *it_1 << '\n'; // 4

// Elemento menor a 4
auto it_2 = s.lower_bound(4);
it--;
cout << *it_2 << '\n'; // 2
```

Conjuntos con elementos repetidos

- C++ también contiene `multiset` y `unordered_multiset`
- La única diferencia es que permiten elementos repetidos

```
multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << '\n'; // 3
```

- Se debe tener cuidado si solo se quiere eliminar un elemento

```
s.erase(s.find(5));
cout << s.count(5) << '\n'; // 2
s.erase(5);
cout << s.count(5) << '\n'; // 0
```

Binary Search Tree - `map`

- Un `map` almacena pares de elementos (llave, valor)
- La llave puede ser de cualquier tipo que tenga definida la función $<$ (que se pueda ordenar)
- `map` mantiene los elementos ordenados por su llave

Binary Search Tree - `map`

- Insertar y acceder a elementos:

```
#include <map>
...
map<string, int> m;
m["mono"] = 3;
m["gato"] = 5;
m["cuaderno"] = 2;
cout << m["mono"] << '\n'; // 3
cout << m["tecla"] << '\n'; // 0
```

Binary Search Tree - map

- count, erase y recorrer un map:

```
if (m.count("tecla")) {
  // llave existe
}
m.erase("mono");
for (auto p : m) {
  // p.first  -> llave
  // p.second -> valor
  cout << p.first << " " << p.second << '\n';
}
// cuaderno 2
// gato 5
// tecla 0
```

Consideraciones

- Las operaciones de `unordered_map` y `unordered_set` son, generalmente, $O(1)$, a diferencia de `map` y `set` donde son $O(\log n)$. Siempre usar las versiones que usen un hash table a menos que sea necesario ordenar las llaves.
- Si un tipo de dato no tiene implementada una función hash (como `pair<int, int>`), no se puede usar como llave en un `unordered_map`.