Universitat de Barcelona

Facultat de Matemàtiques i Informàtica

Pràctica de Programació Científica

Arbres Binàries de Cerca (BST) en C

Assignatura: Programació Científica
Curs: 2024–2025
Data: 16 de maig de 2025

Objectius

L'objectiu d'aquesta pràctica és implementar i entendre el funcionament d'un **arbre binari de cerca** (BST), fent ús d'una estructura `typedef struct` en C. Treballareu amb dades d'alumnes i aprendreu a inserir, estructurar i recórrer aquestes dades de manera eficient.

1 Introducció als Arbres Binàries de Cerca (BST)

Els arbres binaris de cerca (*Binary Search Trees*, BST) són estructures de dades jeràrquiques que permeten emmagatzemar informació de manera ordenada i eficient. Un arbre binari de cerca es construeix segons la següent propietat:

> *Per a cada node, tots els valors del subarbre esquerre són menors que el valor del node, i tots els valors del subarbre dret són majors.*

Aquest ordre facilita operacions de cerca, inserció i recorregut amb una eficiència que pot arribar a ser logarítmica (`O(log n)`) si l'arbre està equilibrat.

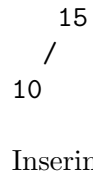
Aplicació al nostre cas: Utilitzarem l'ID dels alumnes com a clau de comparació dins del BST. Així, podrem emmagatzemar els alumnes i obtenir-los ordenats per ID mitjançant un recorregut *inordre*.

Exemple d'inserció pas a pas

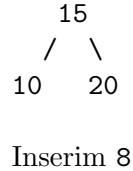
Suposem que volem inserir els següents alumnes (només indiquem l'ID per simplicitat): `15, 10, 20, 8, 12`. L'estructura de l'arbre evolucionaria així:

1. Inserim `15`:

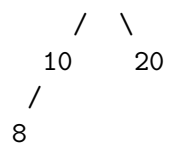
2. Inserim `10` (menor que 15, va a l'esquerra):



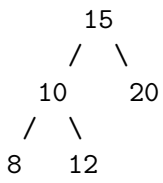
3. Inserim `20` (major que 15, va a la dreta):



4. Inserim `8` (menor que 15 i 10):



5. Inserim `12` (menor que 15, major que 10):



Amb un recorregut `inordre`, visitaríem els nodes en l'ordre: `8, 10, 12, 15, 20`.

Recorregut `inordre` (in-order traversal)

Un dels recorreguts més útils en un arbre BST és el recorregut `inordre`. Aquest recorregut permet obtenir els elements de l'arbre en **ordre creixent segons la clau** (en el nostre cas, l'`id` dels alumnes). El recorregut `inordre` segueix l'ordre següent per a cada node:

1. Visitar el subarbre esquerre.
2. Processar el node actual (per exemple, imprimir-lo).
3. Visitar el subarbre dret.

Aquest esquema és naturalment recursiu, ja que s'aplica la mateixa lògica a cada subarbre.

2 Enunciat

Es demana escriure un programa en C que emmagatzemi dades d'alumnes dins d'un arbre binari de cerca. Cada alumne està identificat per un `id` i una `nota`.

1. Definiu una estructura anomenada `Alumne` amb els camps següents:

- `int id;`
- `float nota;`

2. Definiu una estructura `Node` per representar un node d'un arbre BST:

- `Alumne data;`
- `Node* left;`
- `Node* right;`

3. Implementeu la funció:

```
// Afegir un alumne en l'arbre BST segons el seu id
Node* inserir(Node* arrel, Alumne a);
```

Aquesta funció ha de seguir les regles del BST:

- Si l'arbre és buit, crea un nou node amb l'alumne.
- Si l'ID és menor, insereix a l'esquerra.
- Si l'ID és major, insereix a la dreta.
- Si l'ID ja existeix, ignora la inserció.

4. Implementeu una funció recursiva per recórrer l'arbre en ordre (inordre) i imprimir els alumnes ordenats per ID:

```
void imprimirInordre(Node* arrel);
```

5. Programeu una funció que permeti llegir dades d'alumnes des de l'entrada o des d'un fitxer.

Al `main`, llegiu una llista d'alumnes, inseriu-los a l'arbre i mostreu-los ordenats. Heu de treballar sempre amb memòria dinàmica.

Exemple d'entrada

```
5
15 7.3
10 5.5
20 8.9
8 6.1
12 7.7
```

Exemple de sortida esperada

```
ID: 8    Nota: 6.10
ID: 10   Nota: 5.50
ID: 12   Nota: 7.70
ID: 15   Nota: 7.30
ID: 20   Nota: 8.90
```

3 Extensió (opcional, però on s'aprèn de veritat!)

Si voleu aprofundir, podeu:

- Comparar aquest resultat amb una ordenació del vector original amb QuickSort. Per fer-ho, cal definir una funció de comparació i utilitzar la funció `qsort` de la llibreria `<stdlib.h>`.
- Comptar el nombre de crides recursives i mesurar el temps d'execució.
- Implementar una funció de cerca dins del BST.

Apèndix: Anàlisi de la complexitat computacional

En aquest apèndix es descriu l'eficiència computacional de les operacions principals implementades durant la pràctica. L'anàlisi es fa en funció del nombre d'elements n a tractar.

Inserció en un BST

La complexitat temporal de la funció `inserir` depèn de l'altura de l'arbre:

- **Millor cas** (arbre equilibrat): $\mathcal{O}(\log n)$
- **Pitjor cas** (arbre degenerat en una llista): $\mathcal{O}(n)$

Recorregut inordre

Cada node es visita exactament una vegada, de manera que la complexitat és:

- $\mathcal{O}(n)$

Ordenació amb QuickSort

En cas de voler ordenar el vector original amb la funció `qsort`:

- **Complexitat mitjana**: $\mathcal{O}(n \log n)$
- **Pitjor cas**: $\mathcal{O}(n^2)$ (si l'elecció del pivot és molt desfavorable)

Complexitat espacial

- **BST:** $\mathcal{O}(n)$ nodes dinàmics distribuïts a memòria.
- **Vector:** $\mathcal{O}(n)$ posicions contigües en memòria.

Aquestes complexitats reflecteixen que, tot i tenir costos similars en memòria, la tria de l'estructura pot afectar molt el rendiment en temps segons el cas concret.