Structuri de date

Rotaru Alexandru Andrei
rotarualexandruandrei94@gmail.com
2019-2020

University Politehnica of Bucharest

1 Arbori

1.1 Arbori balansați

O problema destul de subtila care apare la arbori binari de căutare ar fi ca exista șansa ca anumite cai frunza-rădăcina sa fie foarte lungi ridicând artificial înălțimea arborelui și în același timp făcând operațiile de baza mai costisitoare. Structura efectiva este identica ca cea de la un arbore binar de căutare simplu.

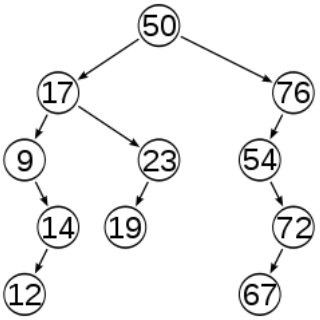

Figure 1: Arbore binar de căutare nebalansat

De aceea ne propunem sa construim anumite operații pe arbori astfel încât sa păstrăm înălțimea unui arbore cat mai mica posibil iar arbori sa rămână constant balansați.

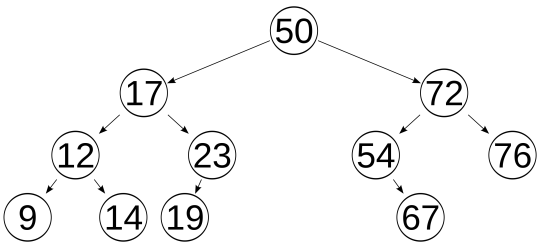

Figure 2: Arbore binar de căutare balansat

Acest lucru se obține prin modificarea funcțiilor insert/delete astfel încât sa realizeze niște rotații ce mențin arborii balansați.

Tipuri de arbori balansați:

- AVL-tree
- Splay-tree
- Red-Black tree

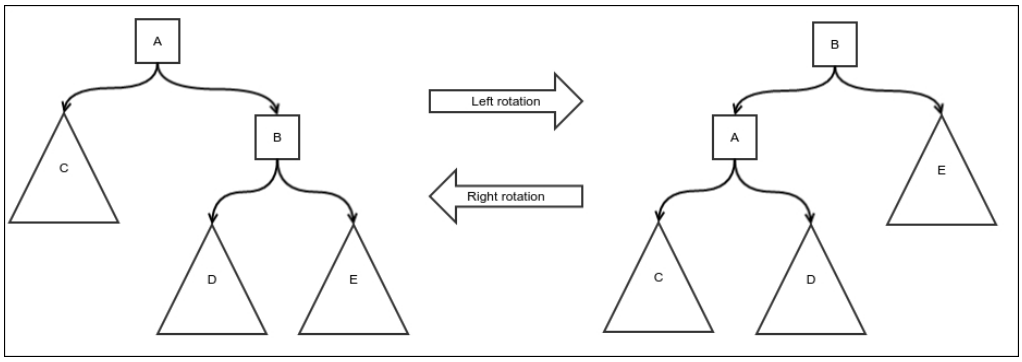

Figure 3: Rotirea la dreapta/stânga

1.2 Trie

Un trie este o structură de tip arbore care retine șiruri de caractere și prefixe de șiruri de caractere. În general este folosit ca o structura rapida de loop-up pentru șiruri de caractere și pentru a retine șirurile mai eficient.

Fiecare nod din arbore poate avea pana la 26 de fii, fiecare fiu fiind asociat cu o litera din alfabet, astfel prin căile de la rădăcina la frunze putem codifica cuvinte.

”

```
1 #define ALPHABET_SIZE 26
2 typedef struct trie_node {
3     uint8_t is_word_end;
4     struct trie_node *children[ALPHABET_SIZE];
5 } trie_node_t;
```

Trie-ul are următoarele proprietăți importante:

- + oferă inserări și căutări de elemente foarte rapid (aceste operații nu depind de numărul de elemente)
- + elementele sunt păstrate sortate alfabetic
- - consuma foarte multa memorie

Ca exemple de funcționalități ce se bazează pe o structura de tip trie: auto-complete, căutare de prefixe comune, stocarea unui dicționar de cuvinte (DEX).

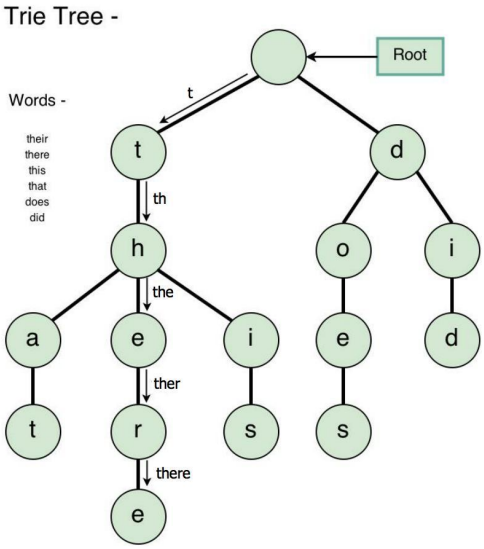

Figure 4: Trie structure

1.3 Exemplu inserare

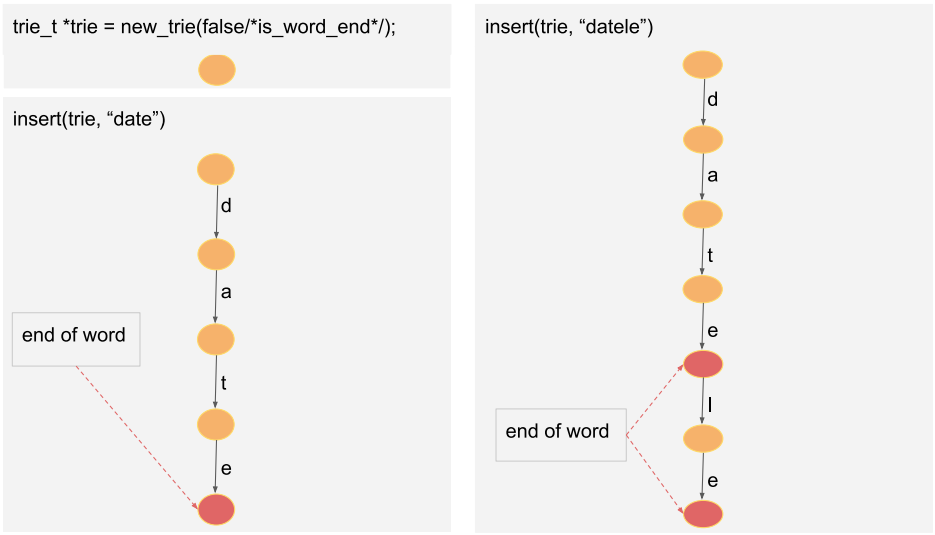

Figure 5: Trie insertion examples

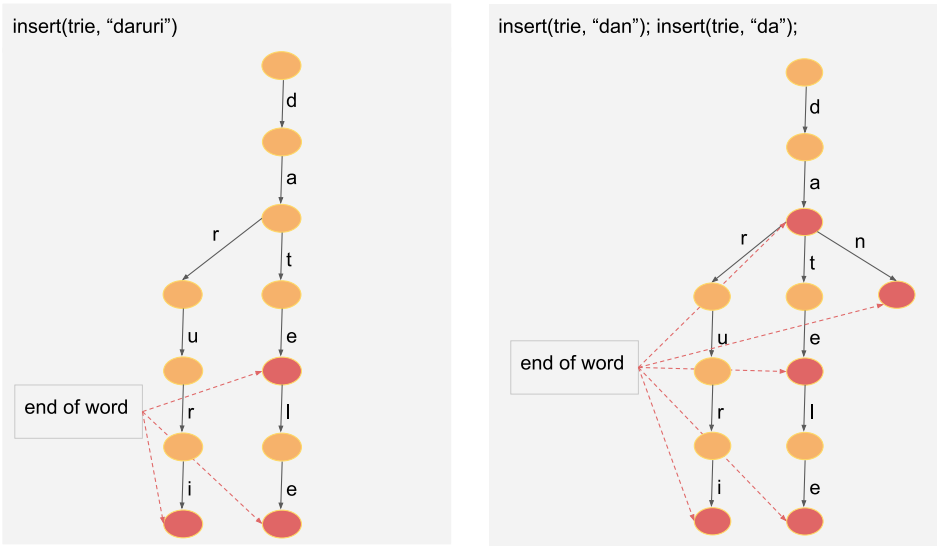

Figure 6: Trie insertion examples

2 Precizari

Laborantul va preciza ce exercitii veti avea de rezolvat. Pentru rezolvarea fiecarui exercitiu veti folosi un fisier separat. **Optional**, va puteti verifica folosind checker-ul (daca este pus la dispozitie).

2.1 Vizualizare

Pentru a putea visualiza o parte din algoritmi si intelege mai bine cum functioneaza structurile de date accesati:
`https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`

2.2 Link-uri catre platforme online cu probleme

[leftmargin=*]Hackerrank: `https://www.hackerrank.com/domains/data-structures`
Leetcode: `https://leetcode.com/problemset/all/`

2.3 Tool-uri de debug

[leftmargin=*]GDB: `https://cs.baylor.edu/~donahoo/tools/gdb/tutorial.html` Valgrind: `http://valgrind.org/docs/manual/quick-start.html`

2.4 Feedback

Pentru a semnala probleme sau a oferi sugestii pentru laboratoarele urmatoare:
`https://forms.gle/8bRw7mPgqaJ9tRzd6`

3 Exerciții

3.1 Exercițiul 1 [6 puncte]

Uneori se întâmpla ca într-o aplicație sa avem nevoie de a salva structurile de date pe disc. Pentru a ne referi la operațiile legate de scrierea/citirea unei structuri de date de pe disc folosim termenii serializare / deserializare. Deoarece nu dorim sa facem efectiv operațiile cu discul o sa ne limitam prin a reprezenta structurile de date ca șiruri de caractere din care le putem recrea. Se da structura unui nod de arbore binar de căutare (bst = binary_search_tree), sa se implementeze operațiile:

- serializare: scrierea unui arbore într-un fișier pe disc
- deserializare: citirea și încărcarea unui ABC dintr-un fișier de pe disc

Modul in care puteti sa realizati acest lucru este sa parcurgeti in inordine arborele si pentru serializarea fiecarui ndd sa incadrati reprezentaera subarborelui intre paranteze rotunde:

Listing 1: Semnăturile funcțiilor

```
sprintf(buffer, "(%s %d %s)", serializare(root->left),
    root->value, serializare(root->right));
```

Listing 2: Semnăturile funcțiilor

```
typedef struct bst_node {
    double value;
    struct bst_node *parent;
    struct bst_node *left;
    struct bst_node *right;
} bst_node_t;

char *serialize(const bst_node_t *root);
bst_node_t *deserialize(const char *serialized_tree);
```

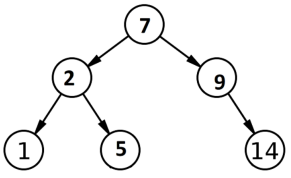

Figure 7: BST

Listing 3: Serializarea arborelui din figura 7

```
"((( 1 ) 2 ( 5 )) 7 ( 9 ( 14 )))"
```

3.2 Exercițiul 2 [10 puncte]

Dându-se o structura de trie sa se implementeze operațiile:

- insert: inserează un sir în trie
- remove: șterge un sir din trie
- contains: verifica dacă un sir a fost deja introdus în trie
- matchings: returnează toate șirurile din trie care au ca prefix un sir dat ca parametru

Listing 4: Semnăturile funcțiilor

```
1 void insert(trie_node_t *root, char *str);
2 bool contains(trie_node_t *root, char *str);
3 void remove(trie_node_t *root, char *str);
4 char **matchings(trie_node_t *root, char *str);
```