Structuri de Date
Tema 2: Căutări de Cuvinte (în fișiere sau vectori de indecși)

Dan Novischi

14 aprilie 2018

1. Obiectivele temei

În urma parcurgerii acestei teme studentul va fi capabil să:

- implementeze un arbore AVL
- agumenteze structra de date a unui arbore AVL (sau echilibrat)
- implementeze eficient o structură de dicționar
- să folosească dicționarul pentru rezvolvarea diverselor tipuri de probleme

2. Descrierea problemei

Dorel isi petrece timplul liber citind literatura si are obiceiul de a-si nota cuvinte noi pe care nu le intelege intr-un fisiere text. Deoarce aceste fisiere contin deobicei foarte multe cuvinte, sortarea acestora este din ce in ce mai anevoioasa. Astfel, Dorel își doreste o aplicație cu ajutorul căreia să poată căuta rapid anumite cuvinte sau secvențe (range-uri) de cuvinte. Avand la dispozitie unul din fisierele lui Dorel (`text.txt`) va trebui sa creati un demo al aplicatiei care sa ofere urmatorele facilitati:

- căutarea după toate cuvintele care incep cu o anumita secventa de (siruri de) carctere.
- căutarea după toate cuvintele intre doua secvente de (siruri de) caracatere.

Pentru realizarea acesteia se va folosi o structură de date ADT (abstract data type) multi-dicționar, care stochează perechi de elemente date sub forma de `<cheie, valoare>` – unde cheia poate fi sau nu duplicată (sau altfel spus, o cheie nu este unică). Una din implementările eficiente a unei astfel de structuri are la bază un arbore binar de căutare echilibrat suprapus peste o listă dublu înlănțuită.

3. Cerințe

Cerința 1 (5p) Implementați în fișierul `AVLTree.h` un multi-dicționar, bazat pe un arbore de căutare (echilibrat) AVL care va stoca un element sub forma unui șir de trei caractere (string) în campul `void* elem` și indexul de început al șirului din fișier în campul `void* info`, respectand următoarele cerințe:

a) Folosind definițiile prezentate mai jos implementați funcțiile de interfața a unui arbore AVL în ordinea dată în fișier (vezi indicații).

b) Modificați relațiile de inserare și ștergere astfel încat nodurile arborelui să formeze (în același timp) o listă dublu înlănțuită ordonată. Cheile duplicat vor face parte **numai din listă**, în timp ce din arbore vor face parte numai cheile unice (vezi indicații).

Indicații:
Definițiile ADT pentru un arbore AVL și un nod al acestuia date în fișierul `AVLTree.h` arată astfel:

```
typedef struct node{
  void* elem;
  void* info;
  struct node *pt;
  struct node *lt;
  struct node *rt;
  struct node* next;
  struct node* prev;
  struct node* end;
  long height;
}TreeNode;
```

```
typedef struct TTree{
  TreeNode *root;
  void* (*createElement)(void*);
  void (*destroyElement)(void*);
  void* (*createInfo)(void*);
  void (*destroyInfo)(void*);
  int (*compare)(void*, void*);
  long size;
}TTree;
```

unde:

a) Un nod conține legăturile afrente arborelui – `lt` copil stanga, `rt` copil dreapta și `pt` parinte, legăturile aferente listei – `prev` pentru nodul anterior, `next` pentru nodul următor și `end` pentru nodul care reprezinta sfarșitul listei de duplicate (vezi Figura 1 si Figura 2), înăltimea nodului în arbore `height`, elementul nodului `elem` și informatia asociata unui element `info`.

b) Definiția arborelelui conține legătura rădacină `root`, pointeri catre funcții – `createElement` / `destroyElement` pentru creearea/distrugerea campului `elem` al unui nod, `createInfo` / `destroyInfo` pentru creerea/distrugerea campului `info` al unui nod și `compare` pentru compararea elementelor `elem` – și `size` care indică numarul de noduri din arbore.

c) Spre deosebire de implementările ADT din cadrul laboratoarelor definițiile campurilor `elem` și `info` sunt de tipul `void*` (pointer către necunoscuți). Acest lucru inseamnă că alocarea, de-alocarea si compararea acestora se va face strict prin intermediul funcțiilor a căror pointeri sunt stocați in definiția arborelui (vezi punctul b). Totodată, trebuie avut in vedere faptul că: la inserție, căutare sau ștergere din multi-dicționar `elem`/`info` vor fi `convertite explicit de la un tip de date la void*`, iar la utilizarea elementului unui nod inafara funcțiilor de lucru cu arborele acestea vor fi `convertite explicit de la void* la tipul de date aferent`.

d) Definiția arborelui NU folosește santinele, cea ce inseamna ca implementarea foloseste volarea `NULL`.

e) `NULL` va avea întotdeauna înălțimea `height` egală cu zero.

f) Un nod nou va fi creeat și introdus în arbore la inserție numai dacă `elem` nu există deja. Altfel, va fi inserat la capatul listei asociate (adica la `end`). De asemenea, campul `end` al unui nod va fi actualizat numai dacă acesta face parte din arbore, altfel acest camp va arăta intotdeauna către nodul in cauza (vezi Figura 1 si Figura 2).

g) Legăturile unui nod nou vor arăta întotdeauna către `NULL` și vor fi actualizate succesiv în baza operațiilor de inserție și stergere. Părintele nodului radăcină (`root`) va arăta întotdeauna către `NULL`.

h) Campul `height` al unui nod nou din arbore va avea întotdeauna valoarea egală cu unu, fiind succesiv actualizat în baza operațiilor de echilibrare.

i) Ștergerea unui nod din dicționar presupune stergerea nodului din arbore dacă acesta nu are duplicate. În caz contrar, se va șterge ultimul duplicat din (porțiunea de) listă aferentă nodului.

Astfel avand un arbore cu chei de tip intreg, în urma inserțiilor succesive ale elementelor:

```
2 3 4 5 6 7 8 5 2 5
```

vom avea arborele din Figura 1 și lista asociata din Figura 2. Săgețile care au culoarea **mov** în diagrama arborelui specifică legături ale listei către nodurile precedente și succesoare astfel încat acesta arată ca în Figura 2.

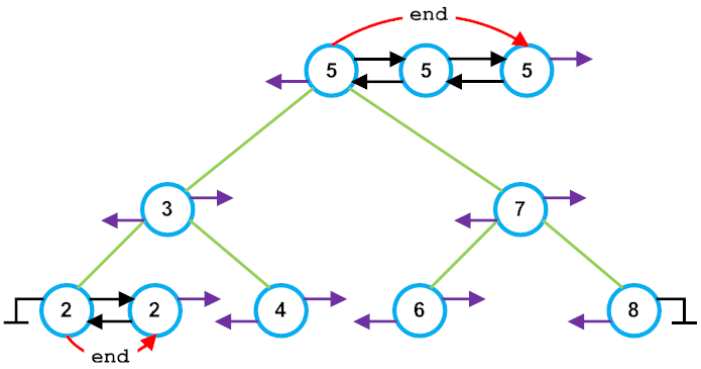

Figura 1: Abore AVL agumentat de o listă

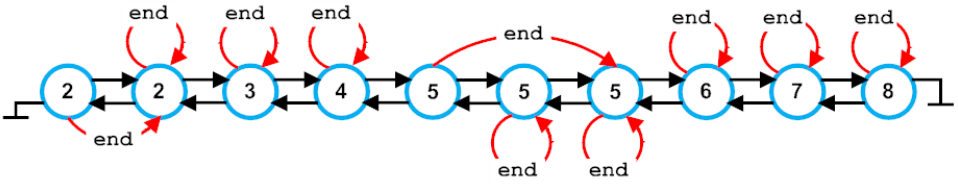

Figura 2: Lista formată prin inserții în arbore

Observatie: Testați implementarea la fiecare pas folosind `make test`. O implementare complet corectă va avea urmatorul output:

```
....$ make test
gcc -std=c9x -g -O0  TestDictionary.c -o TestDictionary -lm
valgrind --leak-check=full ./TestDictionary
==1279== Memcheck, a memory error detector
==1279== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==1279== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==1279== Command: ./TestDictionary
==1279==
. Testul Create&IsEmpty a fost trecut cu succes!         Puncte: 0.050
. Testul Insert-Tree a fost trecut cu succes!            Puncte: 0.050
. Testul Search a fost trecut cu succes!                 Puncte: 0.025
. Testul Minimum&Maximium a fost trecut cu succes!       Puncte: 0.025
. Testul Successor&Predecessor a fost trecut cu succes!  Puncte: 0.025
. Testul Delete-Tree a fost trecut cu succes!            Puncte: 0.075
. Testul Tree-List-Insert a fost trecut cu succes!       Puncte: 0.100
. Testul Tree-List-Delete a fost trecut cu succes!       Puncte: 0.100
. Testul Destroy: *Se va verifica cu valgrind*           Puncte: 0.050

Scor total: 0.50 / 0.50

==1279==
==1279== HEAP SUMMARY:
==1279==     in use at exit: 0 bytes in 0 blocks
==1279==   total heap usage: 68 allocs, 68 frees, 3,016 bytes allocated
==1279==
==1279== All heap blocks were freed -- no leaks are possible
==1279==
==1279== For counts of detected and suppressed errors, rerun with: -v
==1279== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cerința 2 (1p) În fișierul `Tema2.c` implementați următoarele funcții ale caror antete sunt deja date:

- `createStrElement` creeaza un element cu primele trei litere din stringul primit ca parametru.
- `destroyStrElement` distruge un element creat anterior.
- `createIndexInfo` creeaza un index din cel primit ca parametru (pentru utilizarea în cardrul unui arbore).
- `destroyIndexInfo` distruge un index creat anterior.
- `compareStrElem` compara două elemente string ale unor cuvinte date sub forma de memorare în arbore. Funcția întoarce `-1` pentru condiția echivalentă ($a < b$), `1` pentru condiția echivalentă ($a > b$) și `0` pentru condiția echivalentă ($a == b$).
- `buildTreeFromFile` populează un multi-dictionar folosind cuvintele si indecsi acestora din fișierul `text.txt`.

Indicații:

a) Funcții similare (NU indentice), pentru creerea, compararea și distrugerea unor campuri ale unui nod din arbore se gasesc în fișierul `TestDictionary.c`.

b) Funcția `buildTreeFromFile` va parcurge linie cu linie fișierul `text.txt` și va insera perechile `<cuvant, index>` în dicționar nou creat.

c) Indexul fiecarui cuvant este dat numarul de carctere de la inceputul fisierului pana la prima litera a cuvantului (linefeed '\n' reprezinta un singur caracter).

d) O traversare in-order după o populare corectă a dicționarului nou creat va genera urmatorea afișare:

```
Tree In Order:
47:ale  5:atu  31:ca  104:cru  17:dat  124:dau  69:de  128:el:  72:foi  26:gan  59:in  55:jos
85:las  132:mel  145:mel  12:mi-  21:pri  80:pri  139:pro  152:pro  115:put  110:s-a  121:sa
44:si  93:si  38:sta  34:tot  76:ude  0:vez  62:vra  96:vre
```

Cerința 3 (1p) În fișierul `Tema2.c` este definită următoarea structură:

```c
typedef struct Range{
  int *index; // vector de indecsi
  int size; // numarul de elemente
  int capacity; // capacitatea vectorului
}Range;
```

Creați o funcție `singleKeyRangeQuery` care primește un arbore și o cheie de căutare și întoarce un range de indecsi (`Range*`). Funcția va implementa o strategie de căutare prin intermediul careia se vor obține toți indecșii a căror cuvinte incep cu cheia de căutare. Atenție cheia de căutare poate sau nu să fie o cheie exactă, astfel căutand succesiv după `"p"`, `"pr"` și `"pri"` se vor genera urmatoarele rezultate:

```
Single search:
1. prin:21
2. prin:80
3. prost:139
4. prost:152
5. putea:115
```

```
Single search:
1. prin:21
2. prin:80
3. prost:139
4. prost:152
```

```
Single search:
1. prin:21
2. prin:80
```

Observație: Afișarea a fost generată folosind funcția `printWordsInRangeFromFile` dată deja în fișierul `Tema2.c`

Cerința 4 (2p) În fișierul `Tema2.c` creați o funcție `multiKeyRangeQuery` care primește un arbore, un interval de cautare delimitat de cheile `q` si `p` și întoarce un range de indecsi (`Range*`). Functia va implementa o strategie de căutare prin intermediul careia se vor obține toți indecșii a căror cuvinte se alfla in intervalul dat de cheile `q` si `p` (inclusiv). Astfel o cautare in intervalul `j--pr` va genera urmatorul rezultat:

```
Multi search:
1. jos:55
2. lastari:85
3. melcul:132
4. melcul:145
5. mi-a:12
6. prin:21
7. prin:80
8. prost:139
9. prost:152
```

Observație: La fel ca si in cadrul Cerintei 3, cheile de cautare `q` si `p` pot fi sau nu exacte.

5. Punctaj

- Nota acordata unei teme se va socoti prin cumularea punctajelor aferente cerintelor rezolvate corect (maxim 9 puncte) si claritatea codului + README (1 punct) numai daca deadline-ul temei a fost respectat.
- Copiatul se sanctioneaza... pe scurt: **Nu sunt sigur daca vei trece la anul!** :)