Structuri de Date
Tema 2: Căutări de Produse

Dan Novischi, Oana Bălan

5 aprilie 2017

1. Obiectivele temei

În urma parcurgerii acestei teme studentul va fi capabil să:

- implementeze un arbore AVL
- agumenteze structură 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

Un magazin online specializat pe comercializarea de copiatoare si imprimante dorește să îmbunătățească exprianța pe care clienții săi o au în căutarea de produse. Astfel, se dorește introducerea următoarelor posibilități:

- căutarea după toate produsele apartinand unui grup de modele
- căutarea după toate produsele într-un anumit interval de prețuri
- căutarea după toate produsele dintr-un range de modele
- căutarea după toate produsele dintr-un range de modele într-un anumit interval de preturi

În vederea dezvoltarii unei aplicații care să incorporeze aceste functionalități, magazinul pune la dispoziție o bază de date cu produsele din stocul curent dată sub forma unui fișier `*.csv` de următoarea formă:

```
MG6950,580
MG3650,300
...
L2540DN,800
L2520DW,700
...
X3320,1400
XP6022,1000
...
WCX6505,2400
```

unde pe fiecare linie se găsește modelul produsului și prețul acestuia separate prin virgulă.

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.

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;
  TreeNode *nil;
  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 elemente duplicate, înăltimea nodului în arbore `height`, elementul nodului `elem` și informația asociata unui element `info`.

b) Definiția arborelelui conține legătura rădacină `root`, santinela `nil` care specifică nodul nul, pointeri către funcții – `createElement/destroyElement` pentru creearea/distrugerea campului `elem`, `createInfo/destroyInfo` pentru creerea/distrugerea campului `info`, `compare` pentru compararea cheilor `elem` – și `size` care indică numărul 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 memorați în definiția arborelui (vezi punctul b). Totodată, trebuie avut in vedere faptul că: la inserție, căutare sau ștergere din multi-dicționar campurile `elem/info` vor fi `convertite explicit de la un tip de date la tipul void*`, iar la utilizarea elementului unui nod înafara funcțiilor de lucru cu arborele acestea vor fi `convertite explicit de la tipul void* la tipul de date aferent`.

d) Definiția arborelui folosește santinela (dummy object) `nil` în locul valorii NULL. Santinela va avea întotdeauna înălțimea `height` egală cu zero, campurile `elem/info` nule și toate legaturile arătand către (însăși) aceasta.

e) La inserție, un nod nou va fi creeat și introdus în arbore numai dacă `elem` nu există deja (nu este duplicat) în acesta. Altfel, va fi inserat la capatul listei asociate (la `end`) pentru nodul existent în arbore. De asemenea, campul `end` a unui nod va fi actualizat numai dacă acesta face parte din arbore, altfel acest camp va arăta către santinela `nil` (vezi Figura 2).

f) Legăturile unui nod nou vor arăta întotdeauna către santinela `nil` și vor fi actualizate succesiv în baza operațiilor de inserție și stergere. Părintele nodului radăcină va arăta întotdeauna către santinela `nil` (vezi Figura 1).

g) Campul `height` al unui nod nou din arbore va avea întotdeauna valoarea egală cu 1 și este succesiv actualizat în baza operațiilor de echilibrare.

h) Ștergerea unui nod din dicționar presupune stergerea nodului din arbore dacă acesta nu are duplicate, iar 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 din Figura 2. Săgețile care au culoarea mov macarte în diagrama arborelui specifică legături ale listei către nodurile precedente și succesoare astfel încat aceasta arată ca în Figura 2.

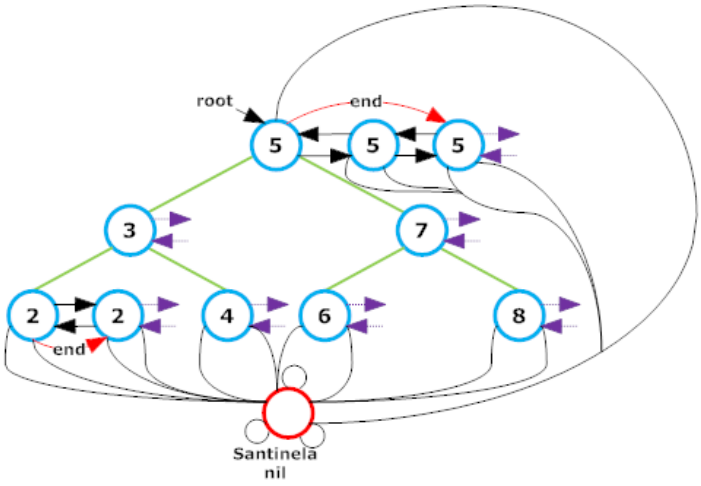

Figura 1: Abore AVL agumentat de o listă

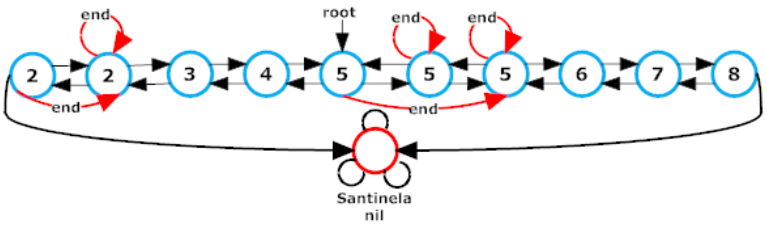

Figura 2: Lista formată prin inserții în arbore

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
==21250== Memcheck, a memory error detector
==21250== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==21250== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==21250== Command: ./TestDictionary
==21250==
. Testul Create&IsEmpty a fost trecut cu succes!         Puncte: 0.05
. Testul Insert-Tree a fost trecut cu succes!            Puncte: 0.075
. 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: 0.05
. Testul Tree-List-Insert a fost trecut cu succes!       Puncte: 0.10
. Testul Tree-List-Delete a fost trecut cu succes!       Puncte: 0.10
. Testul Destroy: *Se va verifica cu valgrind*           Puncte: 0.05

Scor total: 0.50 / 0.50

==21250==
==21250== HEAP SUMMARY:
==21250==     in use at exit: 0 bytes in 0 blocks
==21250==   total heap usage: 69 allocs, 69 frees, 3,096 bytes allocated
==21250==
==21250== All heap blocks were freed -- no leaks are possible
==21250==
==21250== For counts of detected and suppressed errors, rerun with: -v
==21250== 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 caractere din stringul primit ca parametru (primele trei caractere din modelul produsului).
- `destroyStrElement` distruge un element de model creeat anterior.
- `createPriceElement` creeaza un element din cel primit ca parametru (pretul unui produs).
- `destroyPriceElement` distruge un element de preț creeat anterior.
- `createIndexInfo` creează un index din cel primit ca parametru care poate fi memorat într-un arbore.
- `destroyIndexInfo` distruge un index creeat anterior.
- `compareStr` compara două elemente string reprezentand modelele unor produse date sub forma de memorare în arbore. Funcția întoarce `-1` pentru codiția echivalentă ($a < b$), `1` pentru condiția echivalentă ($a > b$) și `0` pentru condiția echivalentă ($a == b$).
- `comparePrice` compara două elemente reprezentand preturile produse date sub forma de memorare în arbore. Funcția întoarce `-1` pentru codiția echivalentă ($a < b$), `1` pentru condiția echivalentă ($a > b$) și `0` pentru condiția echivalentă ($a == b$).
- `buildTreesFromFile` populează două multi-dictionare `modelTree` și `priceTree` aferente modelelor și prețurilor din fișierul `input.csv`.

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 `buildTreesFromFile` va parcurge linie cu linie fișierul `input.csv` și va insera:

- perechile `<model, index>` în dicționarul `modelTree`
- perechile `<price, index>` în `priceTree`

unde indexul este dat de poziția de inceput **in număr de caractere** (nu de linii) a unei intrări din fișierul `input.cvs`

c) O traversare in-order după o populare corectă a celor două dicționare va genera urmatorele afișari:

```
Model Tree In Order:
458:DCP 472:DCP 486:HL- 500:HL- 109:L25 122:L25 301:L25 313:L27 326:L84 135:M20 146:M20
158:M20 446:MB5 433:MB7 407:MC8 420:MC8 0:MG2 12:MG2 49:MG3 73:MG3 85:MG3 97:MG3 37:MG5
25:MG6 61:MG6 341:SP2 368:SP2 354:SP3 381:SP3 394:SPC 245:WCX 259:WCX 273:WCX 287:WCX
170:X30 181:X30 194:X30 207:X33 219:XP6 232:XP6


Price Tree In Order:
0:180 12:200 97:240 37:250 85:250 170:270 25:300 49:300 73:300 135:300 194:470 486:470
458:500 500:500 61:580 301:580 368:580 146:600 158:600 341:600 122:700 381:710 181:750
313:780 109:800 354:870 219:1000 472:1300 207:1400 394:1900 326:2000 245:2300 273:2400
232:2500 446:3000 287:4000 259:4200 433:10000 420:12000 407:17000
```

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 `modelGroupQuery` care primește dicționarul `modelTree` 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 ai căror chei incep cu cheia de căutare (căutare după un grup de modele). Atenție cheia de căutare poate sau nu să fie o cheie exactă, astfel căutand succesiv după `"M"`, `"MG"` și `"MG3"` se vor genera urmatoarele rezultate:

```
Group Model Search:
M2026:300  M2070W:600  M2070F:600  MB562:3000  MB770:10000  MC873:17000  MC853:12000  MG2250:180
MG2250s:200  MG3636:300  MG3650:300  MG3051:250  MG3052:240  MG5750:250  MG6850:300  MG6950:580

Group Model Search:
MG2250:180  MG2250s:200  MG3636:300  MG3650:300  MG3051:250  MG3052:240  MG5750:250  MG6850:300  MG6950:580


Group Model Search:
MG3636:300  MG3650:300  MG3051:250  MG3052:240
```

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

Cerința 4 (2p) În fișierul `Tema2.c` implementați următoarele funcții:

a) `priceRangeQuery` care primește dicționarul `priceTree` și un interval de cautare și întoarce un range de indecși (`Range*`). Funcția va implementa o strategie de căutare prin intermediul careia se pot obține toți indecșii produselor ale caror prețuri se se află în intervalul închis specificat (căutarea după toate produsele într-un anumit interval de prețuri). Spre exemplu, căutarea după intervalul `100 - 400` va genera urmatorul rezultat:

```
Price Range Search:
MG2250:180  MG2250s:200  MG3052:240  MG5750:250  MG3051:250  X3020:270  MG6850:300  MG3636:300
MG3650:300  M2026:300
```

b) `modelRangeQuery` care primește un dicționar și două chei (model string) de căutare și întoarce un range de indecși (`Range*`). Funcția va implementa o strategie de căutare prin intermediul careia se pot obține toți indecșii ai caror chei se află în range-ul dat de cele două chei de căutare. Atenție și în acest caz cheile de căutare pot sau nu să fie chei exacte, astfel încat să permită orice combinație. Spre exemplu, astfel căutand succesiv după `"L--MB5"`, `"L27-MB5"`, `"L27--M"` și `"L2--M"` se vor genera urmatoarele rezultate:

```
Model Range Search:
L2540DN:800  L2520DW:700  L2500D:580  L2700DN:780  L8400CDN:2000  M2026:300  M2070W:600  M2070F:600
MB562:3000


Model Range Search:
L2700DN:780  L8400CDN:2000  M2026:300  M2070W:600  M2070F:600  MB562:3000


Model Range Search:
L2700DN:780  L8400CDN:2000  M2026:300  M2070W:600  M2070F:600  MB562:3000  MB770:10000  MC873:17000
MC853:12000  MG2250:180  MG2250s:200  MG3636:300  MG3650:300  MG3051:250  MG3052:240  MG5750:250
MG6850:300  MG6950:580

Model Range Search:
L2540DN:800  L2520DW:700  L2500D:580  L2700DN:780  L8400CDN:2000  M2026:300  M2070W:600  M2070F:600
MB562:3000  MB770:10000  MC873:17000  MC853:12000  MG2250:180  MG2250s:200  MG3636:300  MG3650:300
MG3051:250  MG3052:240  MG5750:250  MG6850:300  MG6950:580
```

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

Cerința 5 (1p) Flosindu-vă de experiența acaparată în cadrul cerințelor anterioare realizați funția `modelPriceRangeQuery` care implementează o strategie pentru căutarea după toate produsele dintr-un range de modele într-un anumit interval de preturi. Antetul acestei funcții este următorul:

```
1  Range* modelPriceRangeQuery(char* fileName, TTree* tree,
2                              char* m1, char* m2,
3                              long p1, long p2);
```