Structuri de Date (CD 2021-2022)

Tema 2 - Vigenère Chiper

Această temă presupune implementarea unei structuri de dicționar bazată pe arbori AVL și folosirea acesteia pentru decriptarea unor serii de mesaje.

Obiectivele temei

Rezolvarea acestei teme va consta în:

- Implementarea unui arbore AVL
- Augmentarea structurii de arbore echilibrat pentru a suporta noduri duplicate sub forma unei liste dublu înlănțuite
- Implementarea eficientă a unei structuri de dicționar (pentru reținerea perechilor de tipul <cheie, valoare>)
- Folosirea acestei structuri de date pentru rezolvarea a diverse tipuri de probleme

Descrierea problemei

Alice și Bob au descoperit recent o pasiune pentru criptografie și doresc să folosească unul dintre algoritmii învățați pentru a securiza mesajele pe care aceștia și le trimit. Întrucât nici unul dintre cei doi nu dorește să fie nevoie de a descifra manual mesajele sau de a reține cheia cu care acestea au fost criptate, au convenit să folosească un program pentru a automatiza procesul de criptare și decriptare, iar cheia va fi ascunsă într-un fișier text la care atât Alice cât și Bob au acces.

Algoritmul pe care Alice și Bob îl folosesc poartă numele de Vigenère Chiper și funcționează în felul următor: având un text ce se dorește a fi criptat și o cheie de criptare, acestea se parcurg simultan, iar pentru un caracter de la poziția *i* din textul inițial, acesta va fi înlocuit cu valoarea sa shiftată la dreapta cu un număr de poziții indicat de elementul de la indicele *i* din cheie. În cazul în care cheia este mai scurtă decât textul ce se dorește a fi codificat, pur și simplu se va continua procesul de la începutul cheii odată ce s-a ajuns la capătul acesteia. Procesul descris anterior se poate observa și în figura următoare.

Plaintext: SDA IS FUN Key: ABC (repetată până se ajunge la lungimea textului)

Ciphertext: SECITHUO

Plaintext	S	D	A	I	S	F	U	N
Key	**A**	**B**	**C**	A	B	C	A	B
+(Mod26)	+0 ↓	+1 ↓	+2 ↓	+0 ↓	+1 ↓	+2 ↓	+0 ↓	+1 ↓
Ciphertext	S	E	C	I	T	H	U	O

Figura 1 – Exemplu de codificare folosind Vigenère Chiper

În ceea ce privește construcția cheii, aceasta va necesita căutarea anumitor noduri din cadrul arborelui AVL – detalii exacte ale construcției cheii vor fi descrise în cadrul cerințelor aferente. Pentru realizarea arborelui se va implementa o structura de date ADT (abstract data type) multi-dicționar, responsabilă pentru stocarea unor perechi de tipul <cheie, valoare>, unde cheia poate sau nu să fie duplicată – cheile nu trebuie neapărat să fie unice. O implementare eficientă a unei astfel de structuri are la bază un arbore binar de căutare echilibrat suprapus peste o listă dublu înlănțuită.

Cerința 1 (60p)

Să se implementeze în fișierul **`TreeMap.c`** un multi-dicționar bazat pe un arbore binar de căutare echilibrat AVL plecând de la următoarele definiții de funcții:

a) **`createTree`** – Funcție care inițializează un arbore: alocă memorie, setează toate câmpurile acestuia, inițializează mărimea (numărul de noduri din arbore) și setează rădăcina să pointeze către **`NULL`**.
b) **`isEmpty`** – Verifică dacă un arbore este sau nu gol
c) **`search`** – Verifică dacă un anumit element se află în nodurile arborelui și întoarce nodul respectiv sau **`NULL`** dacă acesta nu există
d) **`minimum`** – Funcție care returnează nodul cu elementul de valoare **minimă** din arborele curent (considerând nodul primit ca argument drept rădăcină)
e) **`maximum`** - Funcție care returnează nodul cu elementul de valoare **maxim**ă din arborele curent (considerând nodul primit ca argument drept rădăcină)
f) **`successor`** – Funcție care returnează nodul cu valoarea minimă dar mai mare decât cea a nodului curent (nodul imediat următor din punct de vedere al valorii). În cazul în care funcția este apelată pentru elementul maxim din arbore atunci se va întoarce **`NULL`** (elementul maxim nu are un succesor).
g) **`predecessor`** – Funcție care returnează nodul cu valoarea maximă mai mică decât cea a nodului curent (nodul imediat anterior din punct de vedere al valorii). În cazul în care funcția este apelată pentru elementul minim din arbore atunci se va întoarce **`NULL`** (elementul minim nu are predecesor).
h) **`avlRotateLeft`** – Funcție ce primește ca parametru un arbore (structura ce conține un pointer către rădăcina arborelui, mărimea acestuia și metodele asociate pentru crearea, distrugerea și compararea nodurilor) și un nod x și realizează o rotație la stânga a subarborelui care are vârful în x
i) **`avlRotateRight`** – Funcție ce primește ca parametru un arbore (structura ce conține un pointer către rădăcina arborelui, mărimea acestuia și metodele asociate pentru crearea, distrugerea și compararea nodurilor) și un nod y și realizează o rotație la dreapta a subarborelui care are vârful în y
j) **`insert`** – Inserează un nou nod într-un arbore AVL asigurând reechilibrarea arborelui
k) **`delete`** – Elimină un nod dintr-un arbore AVL asigurând reechilibrarea arborelui și eliberarea memoriei asociate nodului respectiv. În cazul în care există duplicate pentru nodul ce se dorește a fi eliminat atunci se va șterge ultimul nod din lista de duplicate.
l) **`destroyTree`** – Eliberează memoria asociată arborelui

Operațiile de inserare și ștergere trebuie implementate astfel încât nodurile arborelui să formeze simultan și o listă dublu înlănțuită ordonată. Cheile duplicat vor face parte **doar din listă** în timp ce arborele va fi compus din cheile unice.

Definițiile ADT pentru un arbore AVL și un nod al acestuia conform fișierului **`TreeMap.h`** sunt următoarele:

```
typedef struct node{
        void* elem;
        void* info;
        struct node *parent;
        struct node *left;
        struct node *right;
        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 aferente arborelui – **`left`** copilul stâng, **`right`** copilul drept și **`parent`** părintele;
   - legăturile aferente listei – **`prev`** pentru nodul anterior, **`next`** pentru nodul următor și **`end`** pentru nodul care reprezintă sfârșitul listei asociate nodului curent;
   - înălțimea nodului în arbore în câmpul **`height`** (distanța până la cea mai îndepărtată frunză);
   - elementul nodului în câmpul **`elem`** (cheia);
   - informația asociată unui element în câmpul **`info`** (valoarea);

b) Definiția arborelui conține legătura (link-ul) către rădăcina **`root`**, pointeri către funcții – **`createElement/destroyElement`** pentru creearea/distrugerea câmpului **`elem`** al unui nod, **`createInfo/destroyInfo`** pentru creearea/distrugerea câmpului **`info`** al unui nod, **`compare`** pentru compararea elementelor **`elem`** și **`size`** care indică numărul de noduri din arbore.

c) Față de implementările tipurilor de date abstracte folosite în cadrul laboratoarelor, definițiile câmpurilor **`elem`** și **`info`** sunt de tipul **`void*`**. Pentru lucrul cu aceste câmpuri – alocare, de-alocare și compararea acestora se va realiza strict prin intermediul funcțiilor aferente arborelui (vezi punctul b). De asemenea, pentru utilizatrea câmpurilor **`elem/info`** în afara funcțiilor de lucru cu arborele (precum insert, delete etc.), acestea trebuie convertite explicit de la **`void*`** la tipul de date aferent.

d) Definiția arborelui **NU** folosește santinele. Valoarea **`NULL`** va avea întotdeauna înălțimea egală cu zero.

e) Pentru inserare, în urma creării noului nod, trebuie verificat dacă acesta există deja în cadrul arborelui. În cazul în care acest nod reprezintă o cheie nouă (nu se regăsește câmpul **`elem`** în nici unul din nodurile arborelui) atunci adăugarea se va realiza în mod normal având totuși grijă de realizarea legăturilor pentru lista dublu înlănțuită. În cazul în care cheia noului nod

face parte din arbore, acesta va fi adăugat la capătul listei asociate (va fi noul **`end`** în lista aferentă nodului cu câmpul **`elem`** deja existent în arbore)

f) Câmpul **`height`** al unui nod nou în arbore va avea întotdeauna valoarea egală cu unu și va fi succesiv actualizat pe baza operațiilor de echilibrare.

g) Ștergerea unui nod din dicționar presupune ștergerea nodului din arbore în cazul în care acesta nu are duplicate. În caz contrar se va șterge ultimul nod duplicat din (porțiunea de) listă asociată nodului.

h) O implementare corectă va construi și menține o listă dublu înlănțuită ordonată – pornind de la nodul minim din arbore și parcurgând lista dublu înlănțuită ar trebui să fie parcurse toate nodurile (inclusiv cele duplicate) în ordine crescătoare a cheilor (câmpurilor **`elem`**).

Având un arbore cu chei de tip întreg, în urma inserțiilor succesive ale următoarelor elemente:

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

se vor obține arborele din Figura 2 și lista din Figura 3. Săgețile marcate prin culoarea mov în diagramele următoare indică legături către nodurile precedente și succesoare, astfel încât lista construită (în mod automat prin inserție în arbore) arată conform listei descrise în Figura 3.

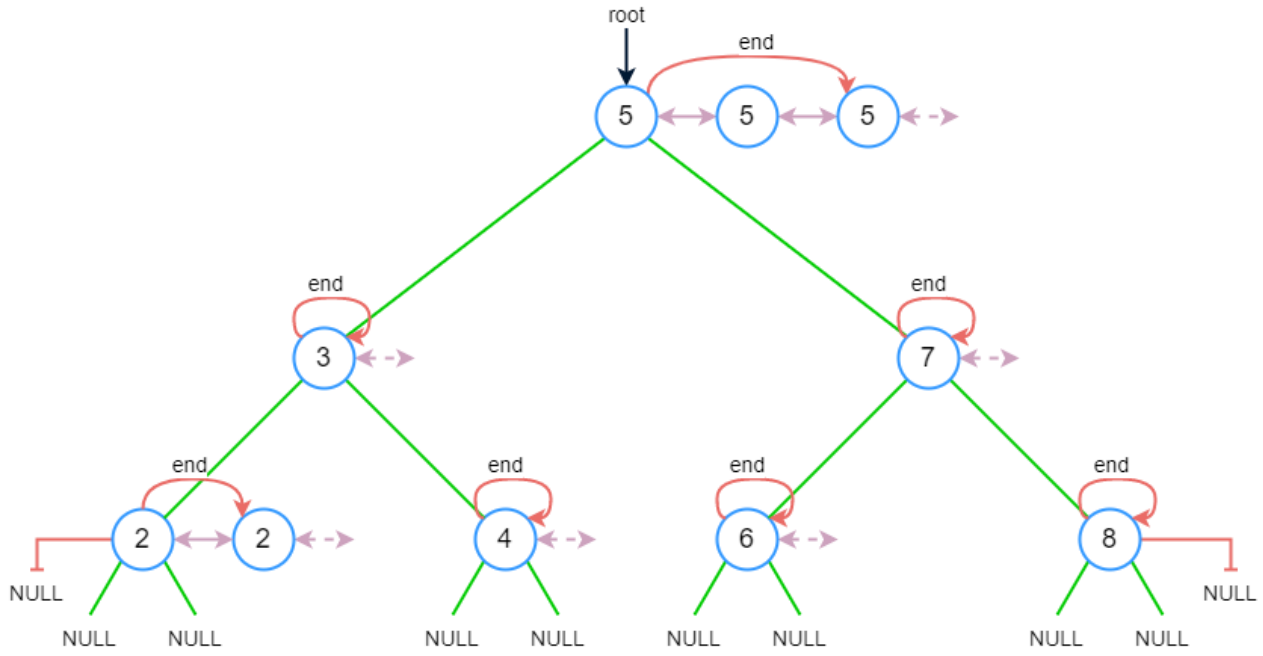


Figura 2 - Arbore AVL augmentat cu listă dublu înlănțuită

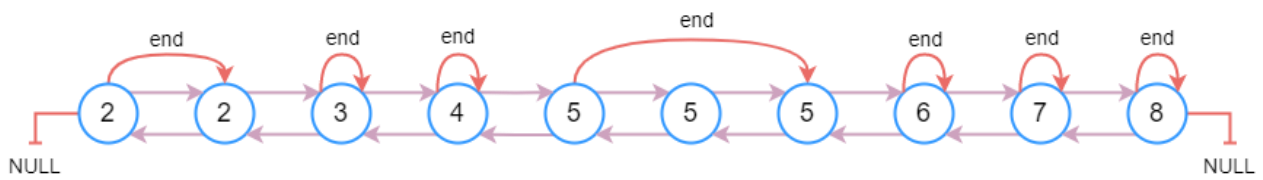


Figura 3 – Lista formată prin adăugări successive în arbore

Cerința 2 (10p)

În fișierul **`tema2.c`** aveți la dispoziție următoarele funcții (deja implementate):

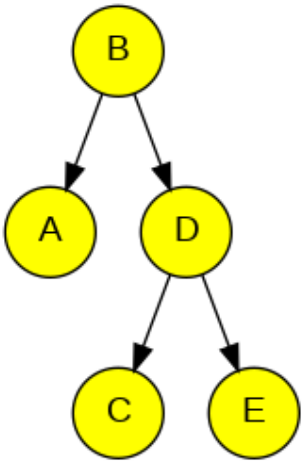
- **`createStrElement`** – creează un element cu primele 5 litere din șirul de caractere primit ca parametru (cuvintele din fișierele necesare pentru decriptare)
- **`destroyStrElement`** – distruge un element creat anterior
- **`createteIndexInfo`** – creează informația asociată unui nod pe baza unui index primit ca parametru
- **`destroyIndexInfo`** – distruge un index creat anterior
- **`compareStr`** – compară două șiruri de caractere reprezentând cuvintele (trunchiate la primele 5 litere) memorate în arbore. Funcția întoarce -1 pentru condiția echivalentă (a < b), 1 pentru condiția echivalentă (a > b) și – pentru condiția echivalentă (a == b)

Porndind de la definițiile anterioare să se implementeze funcția **`buildTreeFromFile`** care construiește un dicționar având drept chei cuvintele (ce vor fi trunchiate la primele 5 litere) și ca valoare indexul unde cuvântul respectiv începe ignorând caracterele separatoare. Cu alte cuvinte, în vederea obținerii valorii asociate unui cuvânt sunt contorizate doar caracterele cuvintelor anterioare.

Ex. Pentru inputul: THIS, IS AN EXAMPLE – se vor obține următoarele valori:

THIS – 0 IS – 4 AN – 6 EXAMPLE - 8

Observație: În urma populării corecte a arborelui pe baza fișierului `inputs/simple_key.txt`, ar trebui să obțineți următorul arbore:



Având următoarele valori:

A – 0

B – 1, 2

C – 3

D – 4

E – 5

În cadrul fișierului **`tema2.c`** puteți decomenta liniile de cod comentate din cadrul funcției **`test_build_tree`** (liniile 659 și 683) pentru a obține și reprezentări grafice ale arborilor.

Cerința 3 (20p)

În fișierul **`Cipher.h`** aveți la dispoziție următoarea definiție a unui **`Range`** de indecși (mulțime de indecși):

```
typedef struct Range{
	int *index;
	int size;
	int capacity;
}Range;
```

În vederea obținerii cheii de decriptare și descifrarea mesajelor dintre Alice și Bob, va fi nevoie de aflarea anumitor indecși (valori din dicționar) conform unor criterii descrise în rândurile următoare și salvarea acestora într-un **`range`** ce va reprezenta practic cheia. Un index reprezintă cantitatea cu care un caracter din textul criptat trebuie shiftat la stânga pentru a fi decriptat – în cazul în care lungimea **`range`**-ului este mai mică decât cea a textului se va proceda conform exemplului din Figura 1 și odată ajuns la finalul cheii se va continua procesul plecând încă o dată de la începutul cheii.

a) (5p) Prima idee pe care Alice și Bob au avut-o pentru construirea cheii a fost următoarea: pornind de la un fișier text, cheia este reprezentată de indecșii de început ai tuturor cuvintelor luați în ordinea indicată de sortarea cuvintelor (sortate crescător, incluzând duplicatele). Implementați funcția **`inorderKeyQuery`** care va întoarce mulțimea de indecși obținută conform descrierii precedente pornind de la dicționarul creat anterior.

Observație: O implementare corectă a funcției **`inorderKeyQuery`** va conduce la următorul output în cadrul fișierului `outputs/output_inorder_key.out` (valorile din dicționar au fost salvate în urma aplicării modulo 26 – 'A' + 27 este identic cu 'A' + 1, iar din considerente de lizibilitate și conversie a valorilor în caractere ale alfabetului englez acestea au fost salvate modulo 26).

```
Decryption key of length 96 is:
6 1 17 16 23 13 25 25 6 18
4 8 10 5 16 6 19 16 1 15
16 11 8 8 5 13 19 11 7 20
24 11 4 4 22 23 14 6 13 8
8 17 17 16 1 21 0 12 6 3
7 7 1 20 20 18 24 1 14 16
7 2 16 5 11 10 7 21 2 20
23 9 24 10 21 1 0 20 24 24
3 25 22 0 5 13 14 16 19 2
14 15 22 13 5 13
```

b) (10p) După o perioadă de gândire Bob a ajuns la concluzia că strategia de compunere a cheii este destul de simplă și ar putea fi ușor replicată. Astfel, el a decis ca noua strategie de construcție a cheii să implice doar indecșii de la nodurile aflate pe un anumit nivel al arborelui (dicționarului), mai exact de pe nivelul corespunzător nodului cu cel mai mare număr de apariții (cel mai frecvent cuvânt din fișierul de construcție a cheii/nodul cu cea mai lungă listă de duplicate asociată). Să se implementeze funcția **`levelKeyQuery`** care întoarce valorile nodurilor (câmpurile **`info`**)

aflate la nivelul celui mai frecvent cuvânt (nodurile fiind parcurse în ordine crescătoare) incluzând listele de duplicate.

Observație: O implementare corectă a funcției **`levelKeyQuery`** va conduce la următorul output în cadrul fișierului `outputs/output_level_key.out`:

```
Decryption key of length 36 is:
23 25 4 16 16 15 11 8 20 24
22 23 13 8 8 17 1 6 1 20
1 16 5 11 2 25 0 5 16 2
14 15 22 13 5 13
```

c) (5p) Văzând inițiativa lui Bob de a spori securitatea algoritmului lor de a genera cheia de criptare/decriptare, Alice a decis că este și de datoria ei de a propune o nouă strategie. Ea a decis ca în formarea cheii să fie folosite doar acele noduri aflte într-un anumit interval de valori posibile (cheile nodurilor să fie cuprinse între două chei de căutare date). Cheile de căutare pot sau nu să fie exacte – pot avea o lungime mai mică decât cea a cheii aflate în nod. Să se implementeze funcția **`rangeKeyQuery`** care primește un dicționar și două chei de căutare (de maxim 5 caractere fiecare) și întoarce un range de indecși (**`Range`** *) conform descrierii anterioare.

Observație: O implementare corectă a funcției **`rangeKeyQuery`** va conduce la următorul output în cadrul fișierului `outputs/output_range_key.out`:

```
Decryption key of length 11 is:
8 10 5 16 6 19 16 1 15 16
11
```

Indicații suplimentare

Pentru lucrul cu funcțiile arborelui și în general cu **`(void *)`** puteți inspecta definițiile funcțiilor din cadrul fișierului `tema2.c`: **`createLong`**, **`createStrElement`**, **`destroyLong`**, **`destroyStrElement`**, **`compareLong`**, **`compareStr`** precum și implementările oferite în cadrul fișierului `TreeMap.c` pentru funcțiile **`createTreeNode`** și **`destroyTreeNode`**. Pe parcursul implementării temei puteți de asemenea folosi funcția **`draw_tree`** (din cadrul fișierului `tema2.c`) pentru generarea unei reprezentări grafice a structurii de arbore obținută. Atenție ca inainte de a încărca arhiva să ștergeți/comentați apelurile care generează imaginile arborilor – folosiți funcția **`draw_tree`** doar pentru debugging. În mod implicit, checker-ul șterge fișierele de output în urma rulării testelor. În cazul în care doriți să inspectați rezultatele generate în directorul `outputs/` puteți comenta ultima parte a regulii de clean din Makefile.

```
# Comportament implicit
rm -f $(EXEC) $(OFILES) $(OUTPUT_DIR)/*.out $(OUTPUT_DIR)/*.dot

# Pentru a nu sterge fisierele de output
rm -f $(EXEC) $(OFILES) # $(OUTPUT_DIR)/*.out $(OUTPUT_DIR)/*.dot
```

Implementare și notare

Pentru rezolvarea cerinței 1 se va completa scheletul de cod aferent în cadrul fișierului **`TreeMap.c`**, iar pentru cerințele 2 și 3 rezolvarea se va face în cadrul fișierului **`Cipher.c`.** Pentru obținerea punctajului de 100 de puncte, pe langă cerințele menționate anterior, este necesară eliberarea completă a memoriei. Pentru o implementare corectă, rularea checker-ului ar trebui să producă următoarele mesaje:

```
All heap blocks were freed -- no leaks are possible

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

- Este obligatorie realizarea unui fișier **`README`** ce va conține detalii de implementare ale funcțiilor. În caz contrar se vor aplica depunctări de maxim 5 puncte.
- Implementarea temei ar trebui sa urmeze regulile și convențiile de scriere a codului – codul ar treui sa fie frumos și lizibil (**coding style**). În caz contrar se vor aplica depunctări de maxim 5 puncte.
- **Nu** este permisă modificarea structurilor de date sau a antetelor funcțiilor menționate (dar puteți creea alte funcții ajutătoare). **Nu** se vor folosi variabile globale.
- Implementări care trec fradulos teste vor primi 0 pentru temă. Orice implementări copiate vor primi 0 pe temă. Temele care **nu** compilează pe vmchecker **nu** vor fi punctate.

Tema va fi încărcată pe vmchecker sub forma unei arhive zip ce va conține fișierele **`TreeMap.c`**, **`Cipher.c`** și fișierul **`README`**. Pentru rularea checker-ului puteți folosi script-ul inclus în cadrul arhivei temei numit `run_checker.sh`. O implementare corectă și completă a temei va conduce la următorul output în urma rulării checker-ului:

```
All heap blocks were freed -- no leaks are possible

For lists of detected and suppressed errors, rerun with: -s
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Test init ................................... passed
Test search ................................. passed
Test minmax ................................. passed
Test succ_pred .............................. passed
Test rotations .............................. passed
Test insert ................................. passed
Test delete ................................. passed
Test list_insert ............................ passed
Test list_delete ............................ passed

Test build_tree ............................. passed
Test inorder_key ............................ passed
Test level_key .............................. passed
Test range_key .............................. passed

Valgrind errors ............................. passed

                                        Total: 100
```