Subiecte lucrarea 2 SDA (28.05.2021)

1) Sa se scrie in limbaj C următoarele funcții:

   a) Funcție de sortare prin selecție cu prototipul:

   void selectionSort(int nodes[], int first, int last, int (*cmp)(int * a, int* b));

- **nodes[]** este tabloul ce conține elementele de sortat; dupa execuția funcției acest tablou conține elementele sortate
- sortarea se va face intre elementele **first** si **last**
- **cmp** este o funcție de comparare (care va fi scrisa separat)

   b) Funcție de traversare in lățime a unui arbore binar cu prototipul:

   void tl_tree(struct node_btree* r)

- **r** este pointer la rădăcina arborelui care se traversează
- funcția afișează informația din fiecare nod

Indicație de rezolvare:

a) Se modifica procedura de la curs prin introducerea a 2 funcții noi: de comparare si de interschimbare cmp si xchange:

```
void selectionSort(int nodes[], int first, int last, int (*cmp (int * a,int * b))
{
   int i, j;
   int min;

   for (i = first; i <last+1; i++)
   {
      min = i;
      for (j = i + 1; j < last+1; j++)
      {
         if (cmp(nodes[j],nodes[min])==-1)
            min = j;
      }
      xchange(nodes[i], nodes[min]);

   }
}
```

```
int cmp(int* a, int* b)
{
   if (*a > *b) return 1;
   if (*a == *b) return 0;
   if (*a < *b return -1;
}

// interschimba a cu b;

void xchange (int * a, int* b)
{
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

b) Se aplica algoritmul de traversare in latime de la grafuri, adaptat pentru arbori (nu se mai creaza liste de adiacenta – descendentii arborelui sint cunoscuti in mod direct)

Algoritmul traversare in latime (vezi curs):

- se defineste o coada cu procedurile add_q si del_q

functia de parcurgere in adincime:
void tl(int x);
x - nodul de pornire

```
tl(x):
initializare coada
visit[x]=1
adauga in coada nodul x
cit timp coada nu e vida
        extrage element din coada (notat cu x)
        afiseaza elementul x
        pentru orice nod y adiacent lui x
                daca (visit[y] diferit de 1)
                        adauga y in coada
                        []
                visit[y]=1
        []
[]
[]
```

Cod:

```
// structura unui nod in arbore
struct node_btree {
   int node_id;
   int visit;
   struct node_btree* left;
   struct node_btree* right;
};

void tl_tree(struct node_btree* r)
{
   struct node_btree* p;

   // adauga in coada
   add_queue(r);
   r->visit=1;

   while (front->next != rear) // cit coada e nevida
   {
      p = del_queue();
      printf("%d ", p->frequency);

      // adauga in coada toate nodurile adiacente nodului p, nevizitate
      if (p->left != NULL && p->left->visit == 0 )
      {
         add_queue(p->left);
         p->left->visit = 1;
      }
      if (p->right != NULL && p->right->visit == 0 )
      {
         add_queue(p->right);
         p->right->visit = 1;
      }

   }
}
```

Se defineste o coada de noduri (pointeri) ale arborelui

```
// structura unui nod coada
struct node {
   struct node_btree* t;
```

```
   struct node* next;
};


// adauga in coada
void add_queue(struct node_btree *p)
{
   struct node* h;

   rear->t = p;

   h=new_node(NULL);
   rear->next=h;
   rear=h;

}
// extrage din coada
struct node_btree* del_queue(void)
{
   struct node* p;
   struct node_btree* x;

   if (front->next != rear )
   {
      p=front->next;
      x=p->t;
      front->next=p->next;
      free(p);
      return x;
   }
}

// creaza nod nou coada
struct node* new_node(struct node_btree* a)
{
   struct node* p;

   p = (struct node* )malloc(sizeof(struct node)) ;
   p->t=a;
   p->next = NULL;

   return p;
}
```

2) Folosind obligatoriu funcțiile de la punctul 1) – eventual adaptate, sa se scrie funcțiile pentru generarea unui arbore de codare Huffman cu maxim 26 de simboluri si pentru afișarea codului binar pentru fiecare simbol. Se va exemplifica modul de apelare a acestor funcții in programul principal. Implementarea acestor funcții se va face dinamic.

Indicație de rezolvare:

Se va crea conform algoritmului de generare a unui arbore Huffman , arborele de codare.

creaza un tablou cu N noduri corespunzatoare simbolurilor
pos=0
cit timp pos < N-1 executa
sorteaza dupa frecventa nodurile dintre pozitiile pos si N-1
alege primele 2 noduri, x si y
calculeaza suma frecventelor, val, din nodurile x (in pozitia pos) si y (in pozitia pos+1)
creaza nod nou z, informatia val
leaga x in stinga lui z
leaga y in dreapta lui z
incrementeaza pos
actualizeaza tabloul de noduri
[]

Se vor modifica in mod corespunzator:

- *Functia de sortare prin selectie – se modifica prototipul pentru compararae noduuri arbore*
- *Functia de comparare – se compara cimpul de frecventa*
- *Functia de interschimbare – se interschimba toata informatia nodului (toate cimpurile)*

Pentru afisarea codurilor se va folosi functia de traversare in latime a arborelui construit modificata astfel incit sa determine numarul nodului (de la 1 2., ...) corespunzator nivelului. Se creaza o tabela de coduri associate fiecarui nod terminal.

Indexul in tabela este numarul nodului -2.

La traversare se afiseaza codul citit din tabela in pozitia (numar nod -2) doar pentru nodurile terminale.

Structura nodului:

```
struct node_btree {
  int node_id;
  int node_level;
  int visit;
  char symbol;
  int frequency;
  struct node_btree* left;
```

```
   struct node_btree* right;
};
```

Codul posibil:

```
#include <stdio.h>
#include <stdlib.h>

// generarea arborelui de codare Huffman

#define N 8

// structura unui nod in arbore
struct node_btree {
   int node_id;
   int node_level;
   int visit;
   char symbol;
   int frequency;
   struct node_btree* left;
   struct node_btree* right;
};

// structura unui nod coada
struct node {
   struct node_btree* t;
   struct node* next;
};

struct node* front;
struct node* rear;

struct node_btree* new_node_btree(int id, int lev, int sym, int frq);
struct node* new_node(struct node_btree* a);

void add_queue(struct node_btree* p);
struct node_btree* del_queue(void);

void tl_tree(struct node_btree* r);
int cmp(struct node_btree* a, struct node_btree* b);
void selectionSort(struct node_btree* nodes[], int first, int last, int (*cmp)(struct node_btree* a,
struct node_btree* b));
void xchange (struct node_btree* a, struct node_btree* b);
```

```
void det_code(struct node_btree* r);

struct node_btree* alfabet[N]; // tabloul de noduri initiale

int main() {

   int pos;                          // pozitia in tablou
   int val;                          // suma frecventelor
   struct node_btree* root_H;        // radacina arborelui Huffman

   // coada vida;
   rear=new_node(NULL);
   front=new_node(NULL);
   front->next=rear;

   // creaza nodurile ce vor constitui arborele Huffman
   alfabet[0]= new_node_btree(-1, -1, 'A',40);
   alfabet[1]= new_node_btree(-1, -1, 'B',10);
   alfabet[2]= new_node_btree(-1, -1, 'C',8);
   alfabet[3]= new_node_btree(-1, -1, 'D',15);
   alfabet[4]= new_node_btree(-1, -1, 'E',2);
   alfabet[5]= new_node_btree(-1, -1, 'F',5);
   alfabet[6]= new_node_btree(-1, -1, 'G',14);
   alfabet[7]= new_node_btree(-1, -1, 'H',6);

   // construire arbore Huffman
   pos=0;
   while (pos < N-1) {
      // sorteaza nodurile
      selectionSort(alfabet, pos, N-1, cmp);
      // creaza nod nou cu primele 2 noduri in pozitia pos si pos+1, dup asortare
      val = alfabet[pos]->frequency + alfabet[pos+1]->frequency;
      root_H = new_node_btree(-1, -1, ' ', val);
      root_H->left = alfabet[pos];
      root_H->right = alfabet[pos+1];
      pos++;
      alfabet[pos] = root_H;
   }

   // root_H este radacina arborelui Huffman
   // traversare in latime
   //tl_tree(root_H);
   printf("\n");
```

```
   // determinare cod
   // root_H este radacina arborelui Huffman
   // traversare in latime - modificata
   det_code(root_H);
   //printf("\n");

   return 0;
}

// creaza nod nou coada
struct node* new_node(struct node_btree* a)
{
   struct node* p;

   p = (struct node* )malloc(sizeof(struct node)) ;
   p->t=a;
   p->next = NULL;
   return p;
}

// creaza nod nou arbore
struct node_btree* new_node_btree(int id, int lev, int sym, int frq)
{
   struct node_btree* p;

   p = (struct node_btree* )malloc(sizeof(struct node_btree)) ;
   p->node_id = id;
   p->node_level=lev;
   p->frequency=frq;
   p->symbol=sym;
   p->visit=0;
   p->left=NULL;
   p->right=NULL;

   return p;
}

// adauga in coada
void add_queue(struct node_btree *p)
{
   struct node* h;
   rear->t = p;

   h=new_node(NULL);
```

```
   rear->next=h;
   rear=h;

}
// extrage din coada
struct node_btree* del_queue(void)
{
   struct node* p;
   struct node_btree* x;

   if (front->next != rear )
   {
      p=front->next;
      x=p->t;
      front->next=p->next;
      free(p);
      return x;
   }
}
//
// traversarea in latime a arborelui
// r nodul initial (radacina)
void tl_tree(struct node_btree* r)
{
   struct node_btree* p;

   // adauga in coada
   add_queue(r);
   r->visit=1;

   while (front->next != rear) // cit coada e nevida
   {
      p = del_queue();
      printf("%d ", p->frequency);

      // adauga in coada toate nodurile adiacente nodului p, nevizitate
      if (p->left != NULL && p->left->visit == 0 )
      {
         add_queue(p->left);
         p->left->visit = 1;
      }
      if (p->right != NULL && p->right->visit == 0 )
      {
         add_queue(p->right);
```

```
            p->right->visit = 1;
        }

    }
}

int cmp(struct node_btree* a, struct node_btree* b)
{
    if (a->frequency > b->frequency) return 1;
    if (a->frequency == b->frequency) return 0;
    if (a->frequency < b->frequency) return -1;
}

void selectionSort(struct node_btree* nodes[], int first, int last, int (*cmp)(struct node_btree* a,
struct node_btree* b))
{
    int i, j;
    int min;

    for (i = first; i <last+1; i++)
    {
        min = i;
        for (j = i + 1; j < last+1; j++)
        {
            if (cmp(nodes[j],nodes[min])==-1)
                min = j;
        }

        xchange(nodes[i], nodes[min]);

    }
}

// interschimba a cu b;

void xchange (struct node_btree* a, struct node_btree* b)
{
    struct node_btree* tmp;

    tmp->node_id = a->node_id;
    tmp->node_level=a->node_level;
    tmp->visit=a->visit;
    tmp->symbol=a->symbol;
    tmp->frequency=a->frequency;
```

```
   tmp->left=a->left;
   tmp->right=a->right;

   a->node_id = b->node_id;
   a->node_level=b->node_level;
   a->visit=b->visit;
   a->symbol=b->symbol;
   a->frequency=b->frequency;
   a->left=b->left;
   a->right=b->right;

   b->node_id = tmp->node_id;
   b->node_level=tmp->node_level;
   b->visit=tmp->visit;
   b->symbol=tmp->symbol;
   b->frequency=tmp->frequency;
   b->left=tmp->left;
   b->right=tmp->right;

}

void det_code(struct node_btree* r)
{
   struct node_btree* p;
   int index; // maxim 26 caractere (maxim 32 noduri terminale)
   // index - calculat ca (nr nod -2)
   char* CODES[] ={ "0", "1", // coduri pe nivel 1 (noduri 2, 3)
   "00", "01", "10", "11", // coduri pe nivel 2 (noduri 4-7)
   "000", "001", "010", "011", "100", "101", "110", "111", // coduri pe nivel 3 (noduri 8-15)
   "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
  // coduri pe nivel 4 (noduri 16-31)
    "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111",
    "00000", "00001", "00010", "00011", "00100", "00101", "00110", "00111",
  // coduri pe nivel 5
    "01000", "01001", "01010", "01011", "01100", "01101", "01110", "01111", // noduri (32-63)
      "10000", "10001", "10010", "10011", "10100", "10101", "10110", "10111",
      "11000", "11001", "11010", "11011", "11100", "11101", "11110", "11111"
   };
   //char procent='%';

   // adauga in coada
   add_queue(r);
   r->visit=1;
   r->node_id=1;
```

```
   r->node_level=0;

   while (front->next != rear) // cit coada e nevida
   {
      p = del_queue();

      // actualizeaza id si nivelul nodurilor descendente
      if (p->left != NULL )
      {
         p->left->node_id = 2 * p->node_id;
         p->left->node_level = p->node_level + 1;
      }

      if (p->right != NULL )
      {
         p->right->node_id = 2 * p->node_id + 1;
         p->right->node_level = p->node_level + 1;
      }

      if (p->left == NULL && p->right == NULL)
      {
         index = p->node_id - 2;
 //printf("simbol =  %c, frecventa = %d%c, (nivel %d), ", p->symbol, p->frequency, procent, p-
>node_level);
 printf("simbol =  %c, frecventa = %.2f, (nivel %d), ", p->symbol, (float)p->frequency/100, p-
>node_level);
         printf("cod = %s", CODES[index]);
         printf("\n");
      }

      // adauga in coada toate nodurile adiacente nodului p, nevizitate
      if (p->left != NULL && p->left->visit == 0 )
      {
         add_queue(p->left);
         p->left->visit = 1;
      }
      if (p->right != NULL && p->right->visit == 0 )
      {
         add_queue(p->right);
         p->right->visit = 1;
      }

   }
}
```