Laborator 1– Algoritmi

Modul de implementare corectă şi structurată a unei liste dublu înlănţuite:

```
typedef struct nod_lista {
    int cheie;
    struct nod_lista *next, *prev;
} t_nod_lista;

typedef struct {
    t_nod_lista *head;
} t_lista;
```

Obs:

- Structura se defineşte în fisierul header
- Accesul la un câmp al unei liste transmise ca parametru IN-OUT în funcție se face cu "->": de ex. `lista->head` [lista e pointer (*)]
- Accesul la un câmp pentru o listă transmisă ca parametru IN se face prin "." : ex. lista.head
- Alocarea unui nod se face cu *malloc()*:

```
(t_nod_lista*)malloc(sizeof(t_nod_lista))
```

- Dezalocarea se face cu *free():* free(x)
- *malloc()* si *free()* se afla in *stdlib.h*

1. Implementați operațiile de SEARCH, INSERT, DELETE, PRINT, FREE pentru o listă dublu înlănțuită. Lucrați cu fișiere header.

```
MAIN
        LISTA L, Nod_lista x
        MAKENULL( L )
        s := -1
        WHILE s <> 0 DO
            READ s //selecție
            CASE s=1: READ key
                      Allocate memory for x
                      x.cheie := key
                      LIST_INSERT( L , x)
            CASE s=2: READ key
                      x:= LIST_SEARCH(L, key)
                      IF x <> NULL THEN
                          PRINT key, x
                      ELSE
                          PRINT "Cheie negasita"
             CASE s=3: READ key
                      x := LIST_SEARCH( L, key)
                      IF x <> NULL THEN
                          LIST_DELETE( L, x)
                      ELSE
                          PRINT "Cheie negasita"
            CASE 4:
                      LIST_PRINT( L )
            END CASE
        END WHILE
        LIST_FREE( L )
END MAIN
```

```
MAKENULL(LISTA L) // L – parametru IN-OUT
        Allocate memory for L.head
        L.head.next := NULL
        L.head.prev := NULL
END MAKENULL
```

```
LIST_SEARCH(LISTA L, KEY) //L param. IN
//fct returnează un pointer către nodul găsit
        Nod_lista x //Nod_lista este un pointer
        x := L.head
        WHILE x <> NULL AND x.cheie <> KEY DO
            x := x.next //deplasare în listă
        END WHILE
        RETURN x
END LIST_SEARCH
```

```
LIST_FREE(LISTA L) //L param. IN-OUT
        Nod_lista x // pointer către un nod
        x := L.head.next
        WHILE x <> NULL DO
            LIST_DELETE( L, x)
            Free x //dezalocă nodul
            x := L.head.next
        END WHILE
        Free L.head
END LIST_FREE
```

```
LIST_DELETE(LISTA L, Nod_lista X)
  // L si X – parametrii IN-OUT deoarece se modifică în functie
        IF X.prev <> NULL THEN
                X.prev.next := X.next
        ELSE
                L.head.next := X.next
        END IF
        IF X.next <> NULL THEN
                X.next.prev := X.prev
        END IF
END LIST_DELETE
```

```
LIST_INSERT (LISTA L, Nod_lista X)
// L si X – parametrii IN-OUT deoarece se modifică în funcție
        IF X == NULL THEN
                RETURN
        X.next := L.head.next
        IF L.head.next <> NULL THEN
                L.head.next.prev := X
        END IF
        L.head.next := X
        X.prev := L.head
END LIST_INSERT
```

```
LIST_PRINT (LISTA L)
        Nod_lista  x // pointer către un nod
        x := L.head.next
        IF x == NULL THEN
                PRINT "Lista e vida"
        END IF      //continuare  →
```

```
        WHILE x <> NULL DO
                PRINT   x.cheie
                x := x.next  //deplasare la următorul
        END WHILE
END LIST_PRINT
```

2. Implementați operațiile de PUSH și POP pentru o stivă, utilizând șiruri. Lucrați cu fișiere header.

```
MAIN
        STIVA s
        e := -1
        INIT_STACK(s, 20)
        WHILE e<>0 DO
                READ e
                PUSH(s, e)
                PRINT_STACK(s)
        END WHILE
        WHILE !STACK_EMPTY(s) DO
                POP(s)
                PRINT_STACK(s)
        END WHILE
        FREE_STACK(s)
END MAIN
```

```
INIT_STACK(STIVA S, INIT_SIZE) //S param.  IN-OUT
        Allocate mem. S.data[INIT_SIZE]
        S.top := 0
END INIT_STACK
```

```
FREE_STACK(STIVA S)
        Free  S.data
        S.data := NULL
        S.top := -1
END FREE_STACK
```

```
STACK_EMPTY( STIVA S)
        IF S.top == 0
                RETURN TRUE
        ELSE
                RETURN FALSE
        END IF
END STACK_EMPTY
```

```
POP(STIVA S)
        IF STACK_EMPTY(S)
                PRINT  "Stiva e goala"
                RETURN -1
        ELSE
                e := S.data[S.top]
                S.data[S.top] := 0
                S.top := S.top -1
                RETURN e
        END IF
END POP
```

```
PUSH(STIVA S, E)
        S.top := S.top + 1
        S.data[S.top]=E
END PUSH
```

```
PRINT_STACK(STIVA S)
        FOR i := 1 TO S.top
                PRINT S.data[ i ]
        END FOR
END PRINT_STACK
```

Modul de implementare corectă şi structurată a stivei este:

`typedef struct stiva {` `  int top;` `  int *data;` `}t_stiva;`	Obs: • Se defineşte în fisierul header • Accesul la elementele stivei se face ca și mai sus prin ” -> ”(pentru param. IN-OUT) si “ . ” (param IN) • Alocarea șirului de date din stivă se face cu *malloc()*- De ex. dacă avem numere întregi în stivă: *stiva->data =(int *)malloc(init_size * sizeof(int))*

3. Implementati operațiile de ENQUEUE si DEQUEUE pentru o coadă, utilizand siruri. Faţă de structura de tip stivă, coada are ca și componente *șirul de date*, *head* (capul) , *tail* (coada) și *length* (numărul de elemente din coada); pornind de la exemplul de mai sus, definiți structura de coadă utilizand *typedef* și *struct*. De asemenea definiţi funcţia QUEUE_EMPTY care returnează TRUE dacă nu mai sunt elemente în coada și FALSE dacă mai sunt elemente în coadă. Lucrați cu fișiere header.

MAIN COADA q e := -1 INIT_QUEUE(q, 20) WHILE e<>0 DO READ e ENQUEUE(s, e) PRINT_QUEUE (s) END WHILE WHILE !QUEUE_EMPTY(s) DO DEQUEUE(s) PRINT_QUEUE(s) END WHILE FREE_QUEUE(s) END MAIN	INIT_QUEUE(*COADA* Q, INIT_SIZE) // Q param IN-OUT *Allocate mem. Q.data[INIT_SIZE]* Q.head := 0 Q.tail := 0 Q.length := 0 END INIT_QUEUE FREE_QUEUE(*COADA* Q) //Q param IN-OUT *Free* Q.data Q.data := NULL Q.head := -1 Q.tail := -1 END FREE_QUEUE
DEQUEUE (*COADA* Q) //Q param. IN-OUT e := Q.data[Q.head] Q.length := Q.length -1 IF Q.head == Q.length THEN Q.head := 1 ELSE Q.head := Q.head + 1 END IF RETURN e END DEQUEUE	ENQUEUE (*COADA* Q, E) //Q param IN-OUT Q.data[Q.tail] := E Q.length := Q.length +1 IF Q.tail == Q.length THEN Q.tail := 1 ELSE Q.tail := Q.tail + 1 END IF END ENQUEUE
!Definiti funcția care determină dacă coada mai are elemente sau nu QUEUE_EMPTY(…) … END QUEUE_EMPTY	PRINT_QUEUE(*COADA* Q) FOR i := Q.head TO Q.head+Q.length-1 PRINT Q.data[i] END FOR END PRINT_STACK

Temă. Probleme opționale

1. Se dă ADT-ul listă, si 2 implementări ale acestuia, cu listă simplu înlănțuită și cu șir. Se cere implementarea operațiilor INSERT, LOCATE, RETRIEVE, DELETE, NEXT, PREVIOUS, MAKENULL, FIRST și PRINTLIST în cele 2 cazuri. Semnificațiile acestor operații sunt cele prezentate la curs.