5. Liste

Intermezzo : Mulţimi

“Definiţie”:(Cantor) Prin mulţime înţelegem o colecţie de obiecte bine determinate şi distincte. Obiectele din care este constituită mulţimea se numesc *elementele mulţimii*.

Două mulţimi sunt egale dacă ele sunt formate din exact aceleaşi elemente.

Remember:

Un program este compus în principal dintr-o parte algoritmică (partea de instrucţiuni) în care se descrie cum vor fi prelucrate datele stocate în calculator şi o parte de declaraţii în care se stabileşte structurarea datelor

Niklaus Wirth a exprimat acest lucru prin formula pregnantă dată în cursul 1:

ALGORITMI + STRUCTURI DE DATE = PROGRAME

În general dorim să analizăm mulţimile care stau la baza domeniilor de valori, mulţimi care reprezintă un concept fundamental atât în informatică cât şi în matematică.

DAR

în timp ce “mulţimile matematice” sunt statice, mulţimile prelucrate de algoritmi pot creşte, se pot micşora sau modifica într-un anumit fel în timp.

Aceste mulţimi sunt numite şi mulţimi dinamice.

Într-o implementare tipică pentru o mulţime dinamică, elementele mulţimii sunt reprezentate prin obiecte (de exemplu structuri) ale căror câmpuri distincte pot fi verificate şi prelucrate.

Accesul la câmpuri este asigurat de cele mai multe ori printr-o referinţă (pointer) pe acel obiect.

La multe mulţimi dinamice unul din câmpuri este câmpul cheie care identifică obiectul. În continuare, îl vom numi pe scurt CHEIE.

Adeseori se presupune că cheile obiectelor mulţimii dinamice aparţin unei mulţimi total ordonate, ca de exemplu mulţimea numerelor reale, sau cuvinte peste un alfabet de litere.

O ordine totală ne permite să punem problema *elementului minim* al mulţimii dinamice (obiectul cu cheia cea mai mică) sau al *elementului următor* mai mare decât elementul curent/dat.

Operaţiile pe mulţimi dinamice pot fi împărţite în 2 categorii:

Interogări: trebuie să furnizeze informaţii despre mulţimea memorată

Operaţii de modificare: modifică mulţimea

O anumită aplicaţie va avea în general nevoie numai de unele din următoarele operaţii pe mulţimi:

Notaţii: S ($\neq \emptyset$) mulţimea dinamică
x un element al mulţimii
k cheia unui element

- **Search(S,k):**
  Interogare care returnează un pointer pe un element $x \in S$ în cazul în care key[x] = k, altfel NIL în cazul în care un asemenea obiect nu aparţine lui S.
- **Insert(S,x):**
  Operaţie de modificare: insereaza elementul x în mulţimea S
- **Delete(S,x):**
  Operaţie de modificare: elimină (şterge) elementul x din mulţimea S

- **Minimum(S):**

  Interogare a mulţimii total ordonate S care returnează elementul cu cheia cea mai mică
- **Maximum(S):**

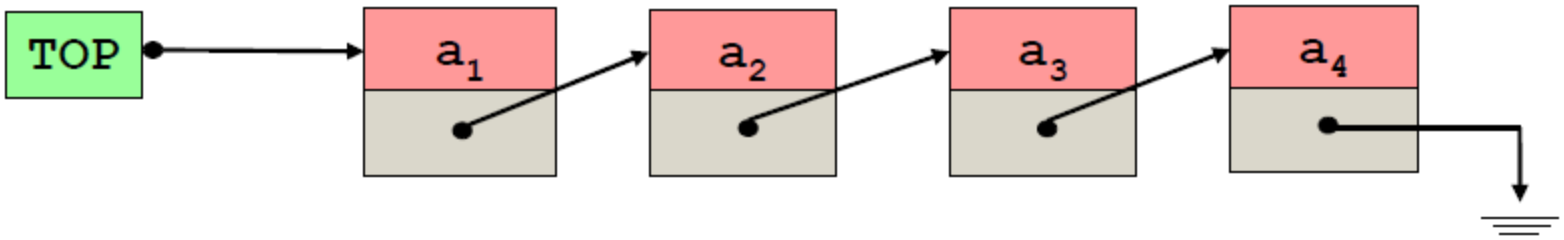
  Analog cu Minimum pentru căutarea elementului cu cea mai mare cheie
- **Succesor(S,x):**

  O interogare care returnează pentru elementul dat x, a cărui cheie este conţinută în mulţimea total ordonată S, următorul element ca mărime. Dacă x este maximul, atunci se returnează NIL.
- **Predecesor(S,x):**

  Analog cu Succesor(S,x) pentru determinarea primului element mai mic decât x. Dacă x este minimul, atunci se returnează NIL

Timpul pentru efectuarea unei operaţii este de obicei dat ca o funcţie de cardinalul mulţimii. Interesante sunt SD care pot efectua operaţiile amintite pe o mulţime de dimensiune *n* într-un timp O(log(n)).

Liste înlănţuite



• *Reprezentare flexibilă* a mulţimilor dinamice care sprijină toate operaţiile amintite mai sus.
• *Ordonare liniară* a obiectelor, ordonare stabilită prin legături/pointeri în fiecare obiect (spre deosebire de array unde ordinea liniară este stabilită de indicii array-ului).

Interesul principal = a parcurge secvenţial o colecţie de articole
=> organizare ca o listă înlănţuită: o structură de bază în care fiecare articol conţine informaţia necesară pentru a ajunge la următorul articol.

Avantajul principal faţă de *array* este că înlănţuirea ne dă posibilitatea de a rearanja eficient articolele. Acest avantaj se plăteşte prin **costul** accesului rapid la un articol oarecare din listă, deoarece singurul mod de a ajunge la el este de a urma înlănţuirile de la un nod la următorul.

Există mai multe moduri de a organiza listele înlănţuite pornind de la definiţia de bază:

Definitie 1: (Sedgewick) O listă înlănţuită este o mulţime de articole în care fiecare articol face parte dintr-un nod care conţine de asemenea o legătură la un nod.

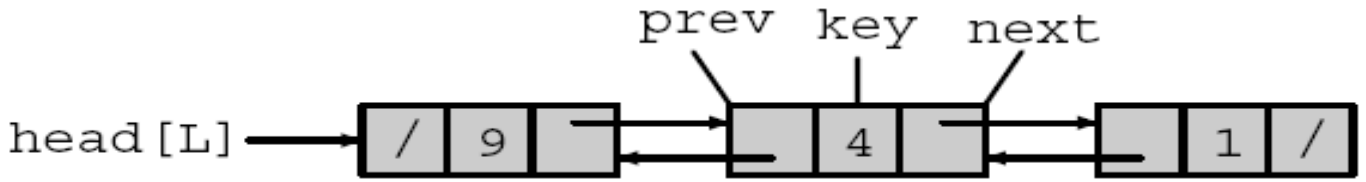
La implementare într-un limbaj, aceste noduri nu sunt reprezentate neapărat într-o zonă contiguă de memorie. Legătura dintre elemente va fi adresa, iar lucrul cu adrese este facilitat de variabilele de tip **pointer**.

Noi definim nodurile ca referinţe la noduri, astfel încât listele înlănţuite sunt uneori numite şi *structuri autoreferite*.

În plus, cu toate că legătura (link-ul) unui nod de obicei indică un nod diferit, ea ar putea să indice nodul însuşi. Deci listele înlănţuite pot fi şi structuri *ciclice*.

- Se face distincţie între liste simplu şi dublu înlănţuite, liste ordonate şi neordonate (la listele ordonate, cheile trebuie să posede o ordine totală, de exemplu Real, string, etc.), liste circulare, etc.

- Elementele x ale unei liste dublu înlănţuite constau dintr-o cheie (şi alte date utile) ca şi din 2 pointeri PREV şi NEXT.



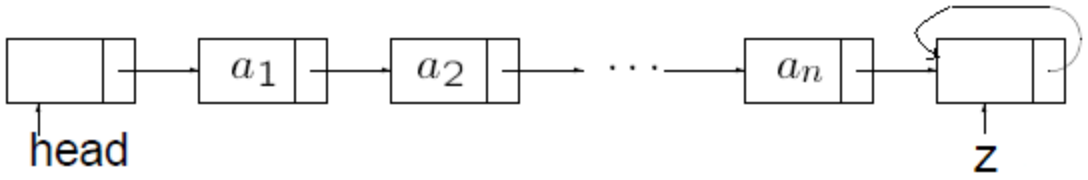
- Dacă prev[x] = NIL, atunci x este primul element (head), dacă next[x] = NIL, atunci x este ultimul element (tail).

- Un obiect head[L] indică spre primul element al listei L, dacă head[L] = NIL, atunci L este lista vidă.

Definiţia 2 (recursivă):

L=
- Φ (**vidă)**
- (**e**; **L_1**); unde L_1 este o listă, iar e este un element; (cap, coadă) sau (head, tail)

Implementarea operaţiilor uzuale pe liste utilizând diferite convenţii



Dummy head and tail nodes

Initializare lista vida: *head = malloc(sizeof *head);*
*z = malloc(sizeof *z);*
head->next = z; z->next = z;
insert t after x: *t->next = x->next; x->next = t;*
delete after x: *x->next = x->next->next;*
traversal loop: *for (t = head->next; t != z; t = t->next)*
test ifempty: *if (head->next == z)*

Circular, never empty

first insert: *head->next = head;*

insert t after x: *t->next = x->next; x->next t;*

delete after x: *x->next = x->next->next;*

traversal loop: *t head;*

do { ... t = t->next; } while (t != head);

test if one item: *if (head->next == head)*

Head pointer, null tail

initialize: *head = NULL;*

insert t after x: *if (x == NULL) { head = t; head->next = NULL; }*

else { t->next = x->next; x->next = t; }

delete after x: *t = x->next; x->next = t->next;*

traversal loop: *for (t = head; t!= NULL; t = t->next)*

test ifempty: *if (head == NULL)*

Dummy head node, null tail

initialize: *head = malloc(sizeof *head);*
head->next NULL;
insert t after x: *t->next = x->next; x->next = t;*
delete after x: *t = x->next; x->next = t->next;*
traversal loop: *for (t = head->next; t != NULL; t = t->next)*
test if empty: *if (head->next == NULL)*

Interfaţa pentru TAD listă

Codul poate fi memorat într-un fisier list.h
Specifică tipurile de noduri şi de legături
declară câteva operaţii care pot fi efectuate cu ele

```
typedef struct node* link;
struct node { itemType item; link next; };
typedef link Node;

void initNodes(int);

link newNode(int);

void freeNode(link);

void insertNext(link, link);

link deleteNext(link);

link Next(link);

int Item(link) ;
```

Implementarea operaţiilor

```
#include <stdlib.h>
#include "list.h"

link freelist;

void initNodes(int N)
{ int i;
       freelist = malloc((N+1)*(sizeof *freelist));
       for (i = 0; i < N+1; i++)
            freelist[i].next = &freelist[i+1];
        freelist[N].next = NULL;
 }

link newNode(int i)
{ link x = deleteNext(freelist);
    x->item = i;
    x->next = x;
    return x;
 }
```

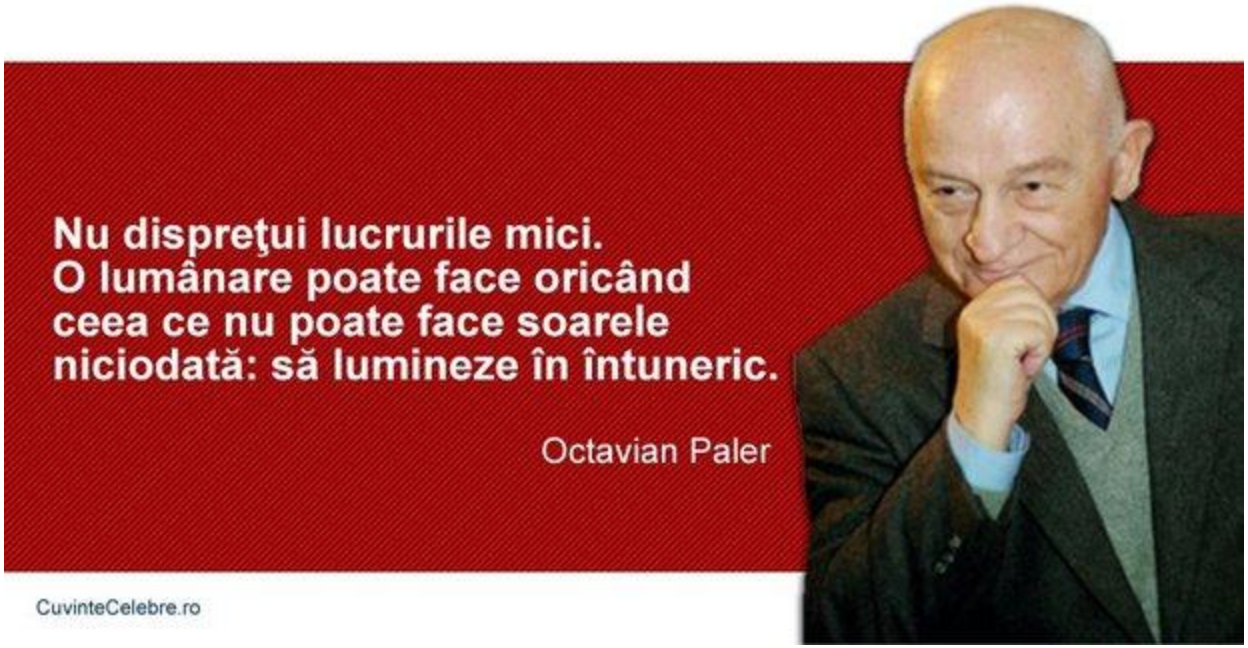
```
void freeNode(link x)
{ insertNext(freelist, x); }

 void insertNext(link x, link t)
{ t->next = x->next; x->next = t; }

 link deleteNext(link x)
{ link t = x->next; x->next = t->next;
return t; }

 link Next(link x) { return x->next; }

 int Item(link x) { return x->item; }
```


Nu dispreţui lucrurile mici.
O lumânare poate face oricând
ceea ce nu poate face soarele
niciodată: să lumineze în întuneric.
Octavian Paler
CuvinteCelebre.ro