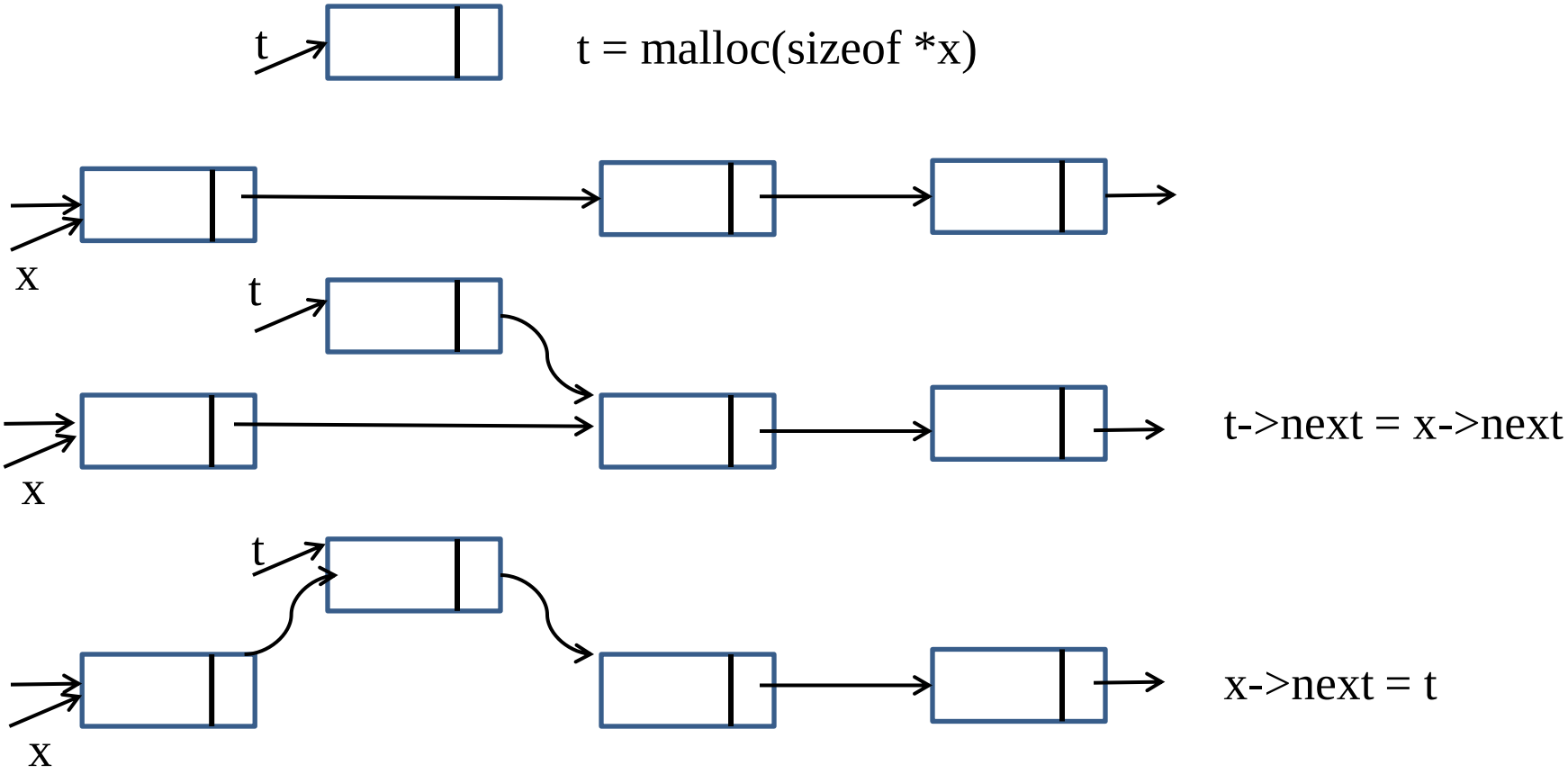
Continuare Liste



Inserarea unui nod într-o listă simplu înlănţuită

Merge Sort implementat cu liste simplu inlănţuite

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


typedef struct _node {
   char element;
   struct _node *next;
} node;

typedef node *olist;


olist init ()
{
   olist L;

   /* Create the dummy node */
   L = (node *)malloc(sizeof(node));
   L -> element = '\0';
   L -> next = NULL;
   return L;
}
```

```
olist insert ( olist L , char ch , int pos )
/* inserare in lista */
{
   int i;
   node *p, *n;

   if (pos < 0) {
      fprintf(stderr, "insert: Invalid index %d\n", pos);
      return L;
   }
   p = L;
   i = 0;
   while (i < pos) {
      p = p -> next;
      if (p == NULL) {
         fprintf(stderr, "insert: Invalid index %d\n", pos);
         return L;
      }
      ++i;
   }
   n = (node *)malloc(sizeof(node));
   n -> element = ch;
   n -> next = p -> next;
   p -> next = n;
   return L;
}
```

```
olist delete ( olist L , int pos ) /* stergerea unui nod din lista */
{
   int i;
   node *p;

   if (pos < 0) {
      fprintf(stderr, "delete: Invalid index %d\n", pos);
      return L;
   }
   p = L;
   i = 0;
   while ((i < pos) && (p -> next != NULL)) {
      p = p -> next;
      ++i;
   }
   if (p -> next == NULL) {
      fprintf(stderr, "delete: Invalid index %d\n", pos);
      return L;
   }
   p -> next = p -> next -> next;
   return L;
}
int isPresent ( olist L , char ch ) /*cautarea unui element */
{
   int i;
   node *p;

   i = 0;
   p = L -> next;
   while (p != NULL) {
      if (p -> element == ch) return i; /* returnare pozitie*/
      p = p -> next;
      ++i;  }
   return -1;
}
```

```
char getElement ( olist L , int pos )/*cautare element, returnare pointer pe element */
{
   int i;
   node *p;

   i = 0;
   p = L -> next;
   while ((i < pos) && (p != NULL)) {
      p = p -> next;
      ++i;
   }
   if (p == NULL) {
      fprintf(stderr, "getElement: Invalid index %d\n", pos);
      return '\0';
   }
   return p -> element;
}
void print ( olist L ) /* listare elemente lista */
{
   node *p;

   p = L -> next;
   while (p != NULL) {
      printf("%c", p -> element);
      p = p -> next;
   }
}
```

```
olist merge (olist a, olist b) /* procedura interclasare */
{ struct _node head; olist c = &head;
     while ((a != NULL) && (b != NULL))
           if ((a->element < b->element))
              { c->next = a; c = a; a = a->next; }
           else
              { c->next = b; c = b; b = b->next; }
     c->next = (a == NULL) ? b : a;
return head.next;
}

olist merge (olist a, olist b); /* procedura sortare recursiva*/
olist mergesort(olist c)
{ olist a, b;
  if (c == NULL || c->next == NULL) return c;
  a = c; b = c->next;
  while ((b != NULL) && (b->next != NULL))
    { c = c->next; b = b->next->next; }
  b = c->next; c->next = NULL;
return merge (mergesort (a) , mergesort(b));

}
```

```
int main () /*exemplificare operatii lista*/
{
   olist L;

   L = init(); /* creare lista vida*/
   L = insert(L,'E',0);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'L',0); /*inserare */
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = delete(L,5); /*stergere*/
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'I',1);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,' ',3);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'4',4);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'4',4);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'0',4);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = delete(L,5);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'0',5);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'2',5);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'I',3);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'A',0);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'P',1);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'R',2);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'I',3);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = delete(L,19);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = delete(L,7);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'1',0);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,' ',1);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = delete(L,10);
    printf("Lista actuala este : "); print(L);
printf("\n");
   L = insert(L,'1',12);
   printf("Lista actuala este : "); print(L);
printf("\n");
   printf("Elementul pe pozitia 2 este %c\n",
getElement(L,2));

mergesort(L); /* sortare lista creata*/
  printf("\n"); printf("Lista sortata este : ");
print(L); printf("\n");

}
```

Stiva

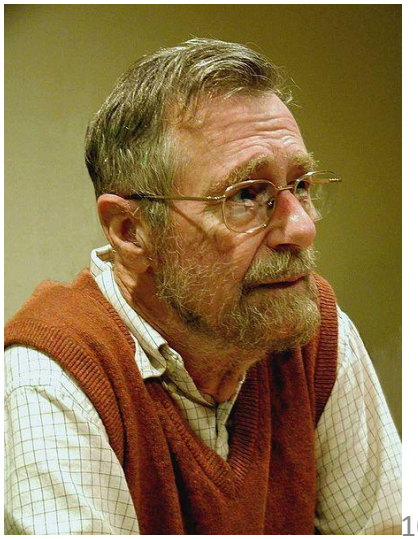

Pizza
37 99 56

Edsger Dijkstra ar fi zis: „**A**bstract **D**ata **T**ypes are a remarkable theory, whose purpose is to describe STACKS”.

Caracteristic pentru o abstractizare de date este „ce se poate face cu ea“ adică operaţiile care pot fi utilizate cu ea

Dintre toate tipurile de date care *suportă insert* şi *delete* pentru colecţii de objecte, cel mai important este numit *pushdown stack.(Sedgewick)*

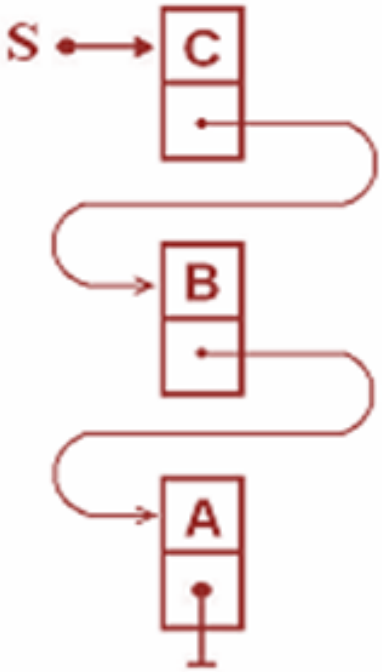
Edsger Wybe Dijkstra (n. 11. Mai 1930 in Rotterdam; † 6. August 2002 in Nuenen, Olanda) a fost un informatician olandez, deschizător de drum în Programarea structurată. Dijkstra a rămas celebru pentru algoritmul drumului minim într-un graf, algoritm care-i poartă numele. În 1972 a obţinut Turing Award.

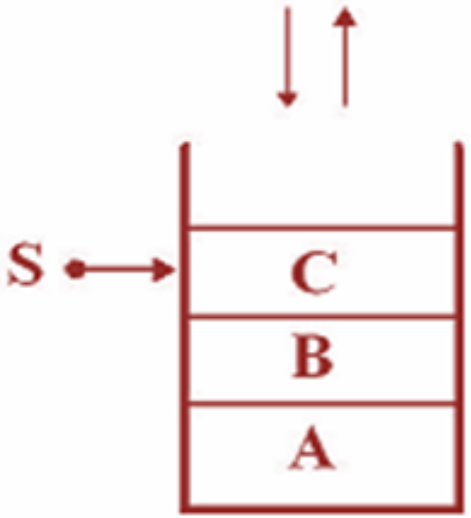


Programele pe calculator sunt în mod natural organizate în acest mod. Ele amână frecvent unele task-uri în timp ce execută altele, mai mult decât atât, ele trebuie să se întoarcă la task-urile cele mai recent amânate. Astfel *stivele pushdown* apar ca structură de date fundamentală pentru mulţi algoritmi.

Definitie: O **stivă pushdown** este un **TAD** *care cuprinde două operaţii de bază: inserează* (push) *un nou element, şi şterge* (pop) *elementul care a fost inserat cel mai recent.*

Reprezentarea stivei

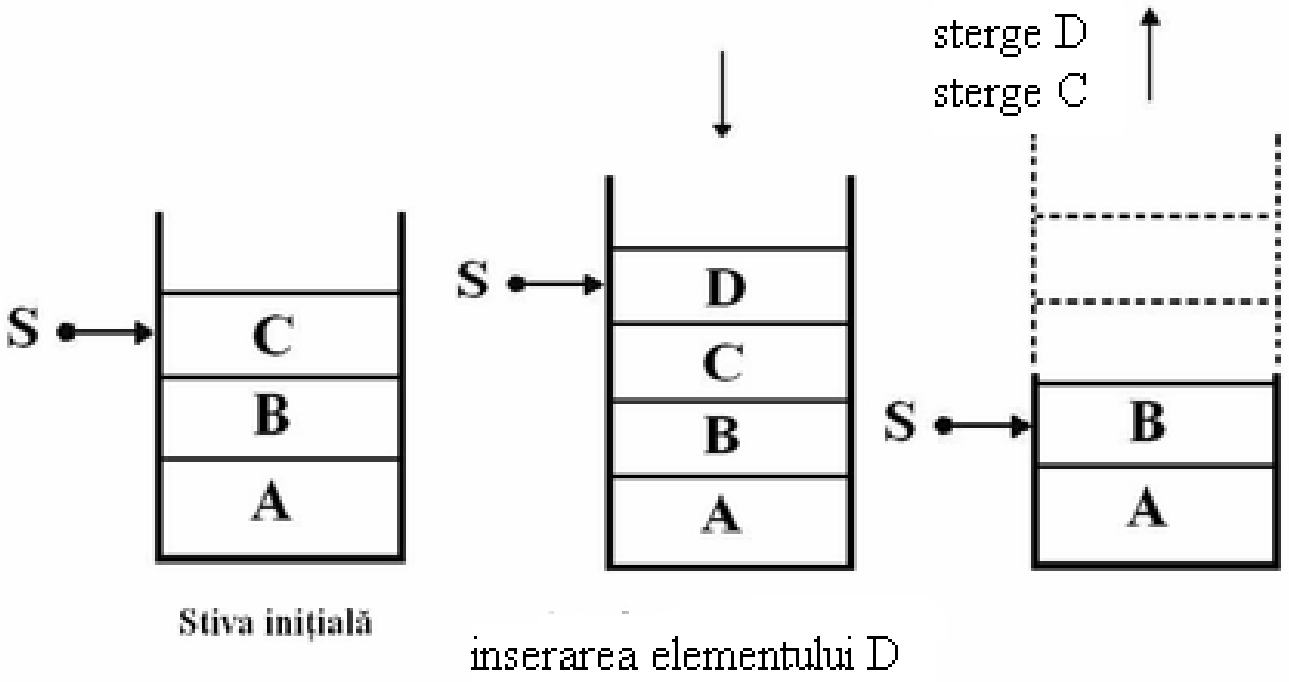




Principalele operaţii cu stiva

- **push(e,S):** Inserează un element *e* la capătul superior al stivei *S*; rezultatul este o stivă modificată.
- **pop(S):** Sterge ultimul element introdus pe stivă; rezultatul este o stivă modificată.
- **top(S):** furnizează o copie a elementului din vârful stivei.
- **create():** crează o stivă vidă.
- **isEmpty(S):** returnează *true* dacă stiva *S* este vidă.

Inserarea si stergerea unui element din stiva



Când vorbim despre un **TAD** *stivă pushdown* ne referim la o descriere a operaţiilor *push* şi *pop* care este suficient de bine specificată încât un program client poate să le utilizeze şi la unele implementări ale operaţiilor care carecterizează stiva pushdown: elementele sunt îndepărtate conform disciplinei/principiului *last-in, first-out (**LIFO**)*

Pentru a scrie programe care utilizează abstractizarea *stivă pushdown* trebuie definită interfaţa.

În C se declară cele patru operaţii care pot fi utilizate de către programele client.

Exemplu:

```
void STACKinit(int);
int STACKempty();
void STACKpush(Item);
item STACKpop();
```

Această interfaţă defineşte operaţiile de bază ale tipului stivă pushdown. Presupunem că cele patru declaraţii sunt într-un fişier Stack.h care este referit ca un *include file* de programele client care utilizează aceste funcţii şi implementări care asigură codul lor; şi că atât clienţii cât şi implementările definesc Item, poate prin includerea unui Item.h (care poate avea un typedef sau care poate defini o interfaţă mult mai generală).
În plus, ne aşteptăm ca să nu mai existe altă legătură între programele client şi implementări.

Argumentul de la STACKinit specifică numărul maxim de elemente care se aşteaptă să le aibă stiva.

Vom exemplifica utilizarea stivei la evaluarea expresiilor aritmetice.

Să presupunem că dorim să evaluăm o expresie aritmetică simplă care conţine înmulţiri şi adunări de numere întregi.
5 * (((9 + 8) * (4 * 6)) + 7)
Calculul expresiei implică salvarea rezultatelor intermediare, de exemplu a lui 9+8 =17 până se va calcula 4*6.

O stivă este mecanismul ideal pentru a salva asemenea rezultate intermediare.

Începem prin a considera problema mai simplă în care operatorul apare după operanzii săi. Aceasta formă, la care poate fi adusă expresia se numeste *postfixată* spre deosebire de forma *infixată*.
5 9 8 + 4 6 * * 7 + *
Se mai numeşte şi *forma poloneză postfixată* de la logicianul polonez Lukasiewicz.

Transformarea inversă:
5 9 8 + 46* * 7 + *
5 (9 + 8) (4 * 6) * 7 + *
5 ((9 + 8) * (4 * 6)) 7 + *
5 (((9 + 8 * (4 * 6)) + 7) *
(5*(((9+ 8) * (4 * 6)) + 7))

Utilizând stiva putem efectua operaţiile şi evalua orice expresie postfixată. Pornind de la stânga la dreapta :

- un operand însemnează “pune pe stivă operandul”
- un operator însemnează “scoate ultimii 2 operanzi de pe stivă, efectuează operaţia şi pune pe stivă rezultatul”

Generare forma poloneză postfixată

```
#include <stdio.h>
#include <string.h>
//#include <alloc.h>
#include<stdlib.h>

typedef char Item;
void STACKinit(int);
int STACKemptyO;
void STACKpush(Item);
Item STACKpop();



//Implementare stivă ca Array
static Item *s;
static int N;
void STACKinit(int maxN)
{ s = (Item*)malloc(maxN*sizeof(Item));
 N = 0;}
int STACKempty()
{ return N == 0; }
void STACKpush(Item item)
{ s[N++] = item; }
Item STACKpop()
{ return s[--N]; }
```

```
//Conversie  Infix->Postfix

char a[100];

int main(){
//{ char *a = argv[1]; int i, N = strlen(a);
int i;
gets(a);
int N=strlen(a);
STACKinit(N);
char *b=(char*)malloc(200*sizeof(char));
char *p=&b[0];

for (i = 0; i < N; i++){
  if (a [i] == ')')
    *p++=STACKpop();
  if ((a[i]== '+') || (a[i] == '*'))
    STACKpush(a[i]);
  if ((a[i] >= '0') && (a[i] <= '9')){
    *p++=a[i];
  *p++=' ';}
}
*p++='\0';
printf("%s\n",b);

return 0;
}
```

Evaluare forma poloneză postfixată

```
#include <stdio.h>
#include <string.h>
//#include <alloc.h>
#include<stdlib.h>
//#define N 37

typedef int Item;
void STACKinit(int);
int STACKemptyO;
void STACKpush(Item);
Item STACKpop();



//Implementare stivă ca Array
static Item *s;
static int N;
void STACKinit(int maxN)
{ s = (Item*)malloc(maxN*sizeof(Item));
N = 0;}
int STACKempty()
{ return N == 0; }
void STACKpush(Item item)
{ s[N++] = item; }
Item STACKpop()
{ return s[--N]; }
```

```
//evaluare forma poloneza;

char a[100];

int main(){
//{ char *a = argv[1]; int i, N = strlen(a);
int i;
gets(a);
int N=strlen(a);
STACKinit(N);
puts(a);
for (i = 0;  i < N; i++){
  if (a[i] == '+')
    STACKpush(STACKpop()+STACKpop());
  if (a[i] == '*')
    STACKpush(STACKpop()*STACKpop());
  if  ((a[i] >= '0') && (a[i] <= '9'))
    STACKpush(0);
  while ((a[i] >= '0') && (a[i] <= '9'))
    STACKpush(10*STACKpop() + (a[i++]-'0'));
}
printf("%d \n", STACKpop());
return 0;
}
```

generare forma poloneza, evaluare forma poloneza

Lansare în execuţie cu redirectare

>genfpol >a.txt

>evalfpol <a.txt

La începutul examenului profesorul anunţă: „aveţi exact 2 ore la dispoziţie. După aceea nu mai primesc nici o lucrare.
După 2 ore profesorul strigă: “ Încheiaţi doamnelor şi domnilor!”
Totuşi un student scrie în continuare ca nebun... O jumătate de oră mai târziu, profesorul are în faţă teancul de lucrări strânse, iar ultimul student vrea să-şi predea şi el lucrarea. Profesorul refuză să o primească.
Studentul se “dă mare“ : Domnule profesor, ştiţi pe cine aveţi în faţă?”
“ Nu” răspunde profesorul. “Minunat” zice studentul şi îşi vâră lucrarea în mijlocul STIVEI...