Tipuri abstracte de date.

Limbajele de programare furnizează tipuri de date standard (sau *tipuri primitive*). De exemplu în C acestea sunt: `char, int, float, double`.

Un tip de date precizează o mulţime **D**, finită şi ordonată de valori (*constantele tipului*) şi o mulţime de operaţii aplicate valorilor, de forma **F : D x D → D** pentru operaţii binare sau **D → D** pentru operaţii unare

Astfel pentru tipul standard `int` avem: **D** - submulţimea întregilor cuprinsă între -32768 şi 32767, **F = {+, −, *, /, %}**

Un ***Tip Abstract de Date*** (TAD) este o specificare a unui set de date de un anumit tip, împreună cu un set de operaţii care pot fi executate cu aceste date. Acesta este o *entitate matematică abstractă*, cu existenţă independentă.

Abstractizarea reprezintă *concentrarea asupra esenţialului*, ignorând detaliile (contează „ce" nu „cum").

De ce sunt necesare TAD? Limbajele de programare, deşi nu pot oferi toată varietatea de tipuri necesară utilizatorului pentru a-şi dezvolta aplicaţiile, posedă un mecanism de creere a unor noi tipuri de date. Implementările acestor tipuri de nivel înalt, dispersate în program, cresc complexitatea şi influienţează negativ asupra clarităţii. Dacă operaţiile asupra acestor tipuri de nivel înalt nu se folosesc în mod consistent, atunci pot apare erori în program.

Specificarea Tipurilor Abstracte de Date.

Un TAD (Tip Abstract de Date) este specificat printr-un triplet **(D, F, A)**, cu::

- **D** - o mulţime de domenii (sau tipuri).
- **F** - o mulţime de funcţii
- **A** – o mulţime de axiome, care precizează proprietăţile funcţiilor, şi care trebuiesc respectate la implementare

Exemple:

1. Mulţimea numerelor naturale

- domenii: {0,1,2,…} {true,false}
- operaţii: zero, iszero, succ, add
- axiome: iszero(0) = true iszero(succ(x)) = false
  add(0,x) = x add(succ(x),y)=succ(add(x,y))

2. O *coadă de întregi*.:.

- domenii: int {true,false} Q = mulţimea cozilor de întregi
- funcţii:

new	crează o coadă de întregi vidă
enq	inserează un întreg ca ultimă poziţie în coadă
deq	şterge primul element din coadă
front	furnizează întregul din vârful cozii
size	furnizează numărul de întregi din coadă
isEmpty	testează dacă lungimea cozii este 0 sau nu

- axiome: new() întoarce o coadă de întregi

```
front(enq(x, new()) = x
deq(enq(x, new()) = new()
front(enq(x, enq(y, Q))) = front(enq(y, Q))
deq(enq(x,enq(y,Q))) = enq(x, deq(enq(y, Q)))
```

Alte exemple de structuri matematice care pot reprezenta TAD sunt mulțimile, grafurile, arborii, matricele, polinoamele, etc.

Pentru starea "`coadă vidă`" operațiile `front` şi `deq` nu sunt definite. Trebuiesc stabilite *precondiții* pentru fiecare operație, indicând exact când se pot aplica aceste operații.

O *precondiție* a unei operații este o aserțiune logică care specifică restricțiile impuse asupra argumentelor operației. O precondiție falsă a unei operații, conduce la o operație nesigură, la un program incorect.
Astfel o precondiție atât pentru `front` cât şi pentru `deq` este: "coadă nevidă"

`front(Q)` şi `deq(Q)` precondiție: `!isEmpty(Q)`

Pentru ca un TAD să fie util, trebuie să fie îndeplinite precondițiile pentru fiecare operație. Un TAD bine definit indică clar precondițiile pentru fiecare operație.

TAD documentează şi *postcondițiile* -condiții care devin adevărate după executarea unei operații. De exemplu, după executarea operației `enq` apare postcondiția "*coadă nevidă*":

`enq(x, Q)` postcondiție: `!isEmpty(Q)`

Operațiile TAD pot fi gândite ca funcții în sens matematic. Precondițiile şi postcondițiile definesc domeniul şi codomeniul funcției. TAD este complet specificat numai în momentul în care toate operațiile au fost definite, cu toate precondițiile şi postcondițiile implicate.

Un TAD este specificat sintactic prin:
-nume
-tipurile (domeniile) cu care este construit
-semnăturile operațiilor (nume, intrări - tipuri, număr şi ordine parametri, ieşiri -tipul valorii întoarse)

Semantic, TAD este precizat axiomatic printr-o serie de reguli logice (*axiome*) care leagă operațiile între ele, sau descriind explicit semnificația operațiilor în funcție de operațiile asupra altor TAD.

De exemplu, o stivă poate fi specificată astfel:

*- Nume: Stiva(**de** TipElem).*
- Domenii:
-`Stiva` – mulțimea instanțelor tuturor stivelor
-`Elem` – mulțimea tuturor elementelor ce pot apare într-o instanță `Stiva`
-`int` – tipul primitiv întreg
- Funcții

constructor:	`new:`	`-> Stiva`
modificatori:	`push:`	`Stiva x Elem -> Stiva`
	`pop:`	`Stiva -/-> Stiva`
accesori:	`top:`	`Stiva -/-> Elem`
	`isEmpty:`	`Stiva -> boolean`
destructor:	`delete:`	`Stiva ->`

- Semantici (axiome):

```
top(push(S,x))=x            pop(push(S,x))=S
isEmpty(new())=true         isEmpty(push(S,x))=false
```

- Precondiții: `pop(S): not Empty(S)`
`top(S): not Empty(S)`

Operaţiile `top(new())` şi `pop(new())` sunt incorecte. Dacă argumentul este o stivă vidă, operaţiile `top` şi `pop` sunt nedefinite.

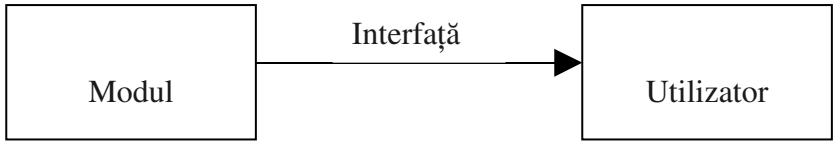
Un invariant al unei instanţe a unui TAD este o proprietate care se păstrează între operaţiile instanţei

Implementarea Tipurilor Abstracte de Date.

Există o distincţie între TAD matematic şi implementarea sa într-un limbaj de programare. Un singur TAD poate avea mai multe implementări diferite.

Stăpânirea aplicaţiilor complexe se obţine prin descompunerea în module. Un modul trebuie să fie simplu, cu complexitatea ascunsă în interiorul lui. Modulele au o interfaţă simplă care permite folosirea, fără a cunoaşte implementarea

Un TAD folosit într-un program, este implementat printr-un ***modul***. Un *modul* este o parte a programului izolată de restul programului printr-o interfaţă bine definită, care lămureşte modul în care este folosit modulul.. Modulele asigură servicii (funcţii, tipuri de date) *Clienţilor*.
Un *client* poate fi orice (program, persoană, alt modul) care foloseşte servicile modulului. Spunem că servicile se *exportă* clientului, sau sunt *importate din* modul.

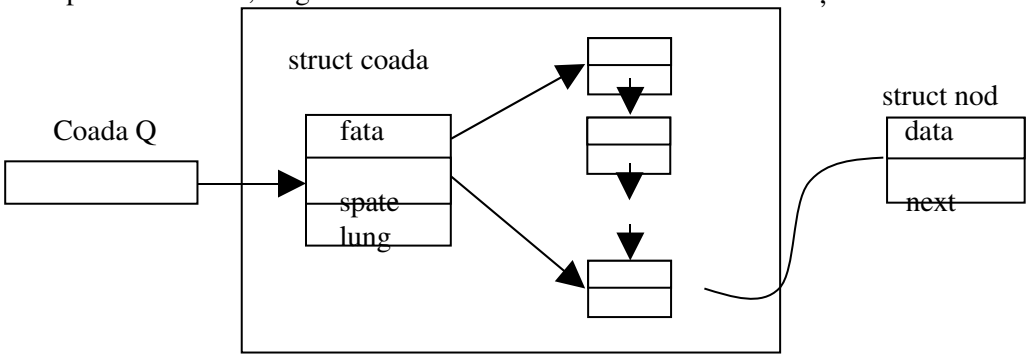


Utilizarea tipurilor abstracte de date asigură ***ascunderea*** şi ***încapsularea informaţiei.***

Conceptul de modul implementează ideea *ascunderii informaţiei*, clienţii nu au acces la detaliile implementării modulului. Clientul are acces la serviciile exportate prin interfaţă. În general, există un modul separat pentru fiecare TAD. Implementarea este ascunsă în spatele interfeţei, care rămâne neschimbată, chiar dacă implementarea se schimbă.

Datele şi operaţiile care le manipulează sunt toate ***combinate într-un singur loc***, adică sunt ***încapsulate*** în interiorul modulului.

În C fiecare TAD constă dintr-o structură, ascunsă utilizatorului. Utilizatorul TAD (clientul) primeşte numai o *referinţă* (un pointer "opac") la această structură. Pentru fiecare operaţie a TAD se defineşte o funcţie, care conţine ca argument pointerul la TAD. Clientul nu poate folosi acest pointer pentru a accesa direct câmpurile structurii, asigurându-se în acest mod ascunderea informaţiei.



Implementarea trebuie să conţină:

- o funcție care crează un nou obiect (constructor)
- o funcție care eliberează memoria asociată cu obiectul TAD când acesta nu se mai foloseşte (destructor).

In C, se separă:

- *implementarea modulului* TAD într-un fişier `.c` care conține structura concretă şi definițiile funcților,
- *interfața modulului*, într-un fişier `.h` care conține definiri de tipuri şi prototipurile funcțiilor exportate.

Exemplu

```
/* Fisier: Coada.h */
struct coada;                 //anunta structura
typedef struct coada* Coada; //furnizeaza un pointer “opac” la ea

Coada Q_New ();               // constructor
void  Q_Delete(Coada* pQ);    // destructor

// Functii de acces
void* Front(Coada Q);
int   Q_Size(Coada Q);
int   Q_Empty(Coada Q);

/* Functii de manipulare */
void Enq (Coada Q, void *el);
void Deq (Coada Q);
```

În interfața `Coada.h` se defineşte un *pointer opac* `Coada` la o structură `struct coada`, care este numai declarată, fiind definită în altă parte (în fişierul de implementare `Coada.c`).

Clientul va include fişierul de interfață prin `#include "Coada.h"` , deci apelurile funcțiilor exportate vor fi recunoscute de către compilator.

Clientul îsi poate declara obiecte de tip `Coada` şi-şi poate defini funcții cu argumente sau valori întoarse de tip `Coada`. Clientul nu se poate referi direct la câmpurile structurii prin intermediul pointerului `Coada` deoarece nu dispune de definiția structurii `struct Coada`.

Fişierul de implementare va conține:

```
/* Fisier: Coada.c */
#include<stdio.h>
#include<stdlib.h>
#include "Coada.h"
typedef struct nod{ /* structura, neexportata */
   void*  data;
   struct nod* next;
} Nod;

typedef Nod* RefNod;

typedef struct coada{
   RefNod fata;
```

```
   RefNod spate;
   int lung;
} Coada;

/* Constructor-Destructor */
Coada Q_New(){
   Coada Q;
   Q = malloc(sizeof((*Coada));
   Q->fata = Q->spate = NULL;
   Q->lung = 0;
   return(Q);
}
void Q_Delete(Coada* pQ) {...}

/* Functii de access */
void* Front(Coada Q) {...}
int Q_Size(Coada Q) {...}
int Q_Empty(Coada Q) {...}

/* Functii de manipulare */
void Enq (Coada Q, void* data) {...}
void Deq (Coada Q) {...}
```

Constructorul şi destructorul pentru structura internă `Nod` sunt folosite de funcțiile `Enq` şi `Deq`. Aceste funcții nu trebuie exportate, deoarece ele lucrează direct cu structura internă `Nod`.

Clientul trebuie să-şi scrie numai aplicația (de exemplu fişierul `TestCoada.c`, care testează modulul `Coada`.

Coada de întregi nu este o coadă generală motiv pentru care vom scrie o coadă de "orice".
Există două soluții posibile.

- Se defineşte tipul `TipElement` în fişierul `.h` prin:

```
typedef int TipElement;
```

Putem schimba astfel tipul elementului editând o singură linie de cod.
Dacă dorim să lucrăm în acelaşi program cu două cozi, una de `int` şi cealaltă de `double` atunci am avea nevoie de două module coadă diferite.

- Se defineşte `TipElement` ca un *pointer generic*:

```
typedef void* TipElement
```

Soluția este mai periculoasă, fiind mai greu de depanat.

Un TAD este caracterizat prin următoarele *proprietăți*:

- exportă un tip
- exportă o mulțime de operații ce alcătuiesc interfața
- operațiile interfeței sunt singurul mijloc de acces la structura de date a TAD
- axiomele şi precondițiile definesc domeniul de aplicație al TAD

O instanță a TAD (o variabilă) este introdusă printr-o referință la structura de date a TAD) numită *pointer opac*.

Interfața este definită într-un fişier antet separat şi nu conține reprezentarea structurii de date, care nu poate fi modificată direct.

Contractul utilizator-implementator.

Între proiectantul TAD şi utilizatorul acestora (clientul) se stabileşte un *contract* prin care:

- proiectantul asigură:
  - -structuri de date şi algoritmi eficienți şi siguri
  - -implementare convenabilă
  - -întreținere simplă
- clientul pretinde:
  - -îndeplinirea obiectivelor urmărite
  - -folosirea TAD fără efortul înțelegerii detaliilor interne
  - -să dispună de un set suficient de operații

Descrierea interfeței formează un contract client-furnizor care:

-dă responsabilitățile clientului. –care trebuie să respecte precondițiile

-dă responsabilitățile furnizorului – care trebuie să asigure respectarea postcondițiilor, asigurând funcționarea corectă a operațiilor

Criteriile de proiectare ale interfețelor Tipurilor Abstracte de Date

- *coeziune*: toate operațiile trebuie să servească unei singure destinații (să descrie o singură abstractizare)

- *simplitate*: se evită facilitățile care nu sunt necesare (o interfață mai mică este mai uşor de folosit)

- *lipsa redundanței*: se oferă un serviciu o singură dată

- *atomicitate*: nu se combină operațiile necesare în mod individual. Operațiile trebuie să fie primitive, neputând fi descompuse în alte operații ale interfeței.

- *completitudine*: toate operațiile primitive trebuie să asigure total abstractizarea

- *consistență*: operațiile trebuie să fie consistente privind convențiile de nume, folosirea argumentelor şi valorilor întoarse

- *refolosire*: TAD suficient de generale pentru a fi refolosite în contexte diferite

- *robustețe la modificare*: interfața să rămână stabilă chiar dacă implementarea TAD se modifică

- *comoditate*: se prevăd operații suplimentare față de setul complet care să asigure o funcționare comodă

Colecţii de date.

Exemple de colecţii.

Colecţiile (sau *containerele*) sunt *structuri de date care păstrează obiecte asemănătoare*, în număr finit. Exemple de colecţii sunt:

- ***vectorul*** – este grup de elemente de acelaşi tip accesate direct, printr-un indice întreg.(indexate cu chei întregi.).

Un *tablou static* conţine un număr fixat de elemente, alocate la compilare.
Un *tablou dinamic* se alocă la execuţie şi poate fi redimensionat.
Faţă de tipul predefinit tablou, colecţia *vector* permite verificarea încadrării indicilor în limite, alocarea dinamică de memorie, etc. Adăugarea unui element are complexitate `O(1)`, pe când ştergerea şi căutarea `O(n)`.

- ***şirul de caractere*** (***string***) - este un vector de caractere, cu operaţii specifice: determinarea lungimii şirului, compararea a două şiruri, copierea unui şir, concatenarea unui şir la altul, căutarea unui subşir, etc.
- ***lista*** - este o colecţie omogenă, cu număr de elemente variabil în limite foarte largi. Elementele listei sunt accesibile secvenţial cu complexitate `O(n)`. Adăugarea şi ştergerea de elemente se fac eficient cu complexitate `O(1)`.
- ***stiva*** (***stack***) – este o listă cu acces restrâns la unul din capete (vârful stivei). Operaţiile specifice: punerea (`push`) şi scoaterea unui element din stivă (`pop`) se fac eficient cu complexitate `O(1)`.
- ***coada*** (***queue***) – este o listă la care inserările se fac pe la un capăt (spatele cozii), iar ştergerile se fac pe la celălalt capăt (faţa cozii). Cozile păstrează elementele în ordinea sosirii.
- ***mulţimea*** (***set***) – este o colecţie de valori unice. Permite operaţii eficiente (cu complexitate `O(log n)`) de inserare, ştergere, test incluziune şi operaţii specifice mulţimilor ca intersecţia, reuniunea, diferenţa, etc.
- ***coada prioritară*** (***priority queue***) – este o colecţie coadă în care elementele au asociate priorităţi şi la care ştergerea elementului cu prioritate maximă se face în `O(1)`, iar inserarea unui element se face corespunzător priorităţii în `O(log n)`. Inserarea elementelor în *coada prioritară* se face în ordinea priorităţii, astfel încât să fie extras întotdeauna elementul cel mai prioritar. Un spital de urgenţă foloseşte ca model coada prioritară. Cozile prioritare sunt folosite la planificarea joburilor într-un sistem de operare (jobul cel mai prioritar va fi primul executat).
- ***dicţionarul*** (***dictionary, map***) – reprezintă o colecţie indexată de elemente, la care indexul poate fi orice valoare ordonabilă

Operaţii specifice colecţiilor.

Colecţiile avute în vedere au următoarele particularităţi:

- sunt dinamice, deci o colecţie nou creată va fi vidă
- sunt colecţii referinţă, deci conţin pointeri la elementele colecţiei şi nu valorile elementelor, pentru a putea avea orice tipuri de elemente în colecţie.

Alocarea dinamică de memorie pentru o colecţie se face:

- *secvențial*, printr-o zonă de memorie contiguă (tablou), alocată la crearea colecției. Această zonă poate fi realocată, dar operația este costisitoare şi se foloseşte rar.
- *înlănțuit*, cu alocare dinamică pentru fiecare element nou adăugat colecției.

Posibilitatea de a nu specifica tipul elementelor, adică de a avea ***colecții generice*** (colecții de elemente de orice tip) se realizează în C utilizând pointeri la void (**`void *`**). Aceasta impune cunoaşterea dimensiunii fiecărui element, pentru alocarea corespunzătoare de memorie.

Proprietăți ale colecțiilor:

- pot fi copiate.
- *Capacitatea* colecției este numărul maxim de elemente pe care îl poate conține colecția. *Cardinalul* colecției este numărul actual de elemente conținute în colecție.
- colecțiile *nemodificabile* nu suportă operații de modificare (ca `add`, `remove` sau `clear`).
- colecțiile *imutabile* nu permit modificarea elementelor colecției.
- colecție cu *acces aleatoriu* asigură acelaşi timp de acces pentru toate elementele.

O *colecție* (sau un *container*) implementează următoarele ***operații***:

- crearea unei colecții noi vide (constructorul): **`Col new();`**
- ştergerea tuturor obiectlor colecției (destructorul): **`void delete(Col *pC);`**
- raportarea numărului de obiecte al colecției: **`int size(Col C);`**
- inserarea unui nou obiect în colecție: **`void add(Col C, Iter p, void *el);`**
- scoaterea unui obiect din colecție: **`void *remove(Col C, Iter p)`**
- accesul la un obiect din colecție: **`void *get(Col C, Iter p );`**

Mai pot fi definite operații precum:

- test colecție vidă: **`int isEmpty(Col C);`**
- modificarea unui element din colecție: **`void modif(Col C, Iter p, void *el);`**
- copierea unei colecții: **`Col copy(Col C);`**

Parcurgerea colecțiilor.

Traversarea sau parcurgerea colecției presupune enumerarea sistematică a tuturor elementelor colecției, folosind în acest scop un ***iterator*** sau *enumerator*. El poate fi văzut ca un pointer la oricare element din colecție

Un *iterator* se implementează prin trei funcții care asigură:

- *poziționarea iteratorului pe primul element*
- *poziționarea iteratorului pe următorul (precedentul) element din colecție*
- *detectarea sfârşitului colecției (după ultimul element)*

Pentru a descrie un domeniu de valori vom folosi doi iteratori care vor indica limitele domeniului.

- poziționare pe primul element al colecției

`Iter begin(Col C);`..

- poziționare la sfârşitul colecției, după ultimul element

`Iter end(Col C)`;

- poziționare pe următorul element din colecție

`Iter next(Col C, Iter p);`

Traversarea unei colecții poate fi abstractizată prin:

`Col C;`

```
Iter p;
for(p= begin(C); p!=end(C); p=next(C, p))
      vizitare_element(get(C, p));
```

Criterii de clasificarea a colecțiilor.

a) ***Numărul de succesori*** al unui element:

- *Colecții liniare* - având ca reprezentanți: secvența, mulțimea, dicționarul, grupul, etc
- *Colecții neliniare* (arborescente, recursive) - de tipul arborilor.

1. Colecții liniare.

O *colecție liniară* conține elemente ordonate prin poziție.Astfel există primul element al colecției, al doilea element,..., ultimul element.
Într-o *colecție neliniară* elementele sunt identificate fără o relație pozițională.

Metoda de acces la elemente separă colecțiile liniare în:

- ***Colecții cu acces direct*** – orice element poate fi selectat fără a accesa în prealabil elementele care îl preced.

- ***Colecții cu acces secvențial*** – accesul la un element se face pornind de la primul element al colecției prin deplasare către elementul căutat.

Listele liniare sunt exemple de colecții cu acces secvențial. Într-o listă liniară numărul de elemente variază în limite foarte largi şi operațiile de inserare şi ştergere sunt foarte frecvente.

Un *fişier* este o colecție plasată pe un suport de memorie externă căreia i se asociază o structură de date numită *flux (stream)*. Fişierele disc permit acces direct, celelalte numai acces secvențial. Operația de citire şterge (extrage) un element din fluxul de intrare, iar cea de scriere adaugă (inserează) un element în fluxul de ieşire.

- ***Colecții cu indexare generalizată*** – Tabloul este o colecție care permite accesul direct la orice element folosind un indice întreg. În general, fiecărui element i se poate asocia o *cheie* (care nu mai este neapărat un întreg, ca indexul), care să fie folosită pentru a accesa elementul.

Un *tabel de dispersie* păstrează *date* şi *chei* asociate acestora. *Cheia* este transformată într-un *index* întreg, folosit pentru a localiza data.

Dicționarele (tablourile asociative) constau din *asocieri* – perechi *cheie – valoare*. Valoarea din asociere este accesată direct, folosind cheia, ca un *index generalizat*.

2. Colecțiile neliniare se clasifică în:

- ***Colecții ierarhice*** – în care elementele sunt partiționate pe niveluri. Fiecare element de pe un nivel are mai mulți succesori pe nivelul următor.

Arborele este o colecție ierarhică în care toate elementele emană dintr-o singură sursă numită *rădăcina* arborelui. Elementele arborelui se numesc noduri şi fiecare nod îşi indică descendenții (copiii). Fiecare nod are un predecesor unic (exceptând rădăcina).

Arborele este structura ideală care descrie sistemul de fişiere cu directoare şi subdirectoare, ca şi diagramele de organizare ale firmelor.

O formă specială de arbore – *arborele binar* impune fiecărui nod să aibă cel mult doi descendenți.
Impunerea unei relații de ordine între cheile unui arbore binar defineşte *arborele binar de căutare*, o structură de date eficientă pentru păstrarea volumelor mari de date.

Heapul este un arbore special (*arbore parțial ordonat*) în care cel mai mic (sau cel mai mare) element se află întotdeauna în rădăcină. Folosirea unui heap permite sortarea unei liste printr-o metodă foarte eficientă (*heapsort*).

- ***Colecţii grupuri*** – sunt colecţii neliniare în care elementele nu sunt ordonate în nici un fel.

De exemplu *mulțimea* reprezintă un grup. Operațiile specifice sunt: reuniunea, intersecția, diferența, testul de apartenență, relația de incluziune.

Un *graf* este o structură de date care modelează relaţiile între obiecte prin două mulțimi: o mulțime de vârfuri şi o mulțime de muchii care conectează aceste vârfuri.

Grafurile au aplicaţii în planificarea lucrărilor, probleme de transport, etc. Operaţii specifice sunt: adaugă / şterge vârf, găsirea vârfurilor accesibile, pornind dintr-un anumit vârf şi efectuând o parcurgere specifică: în adâncime, în lățime, etc.

O *rețea* este o formă specială de graf, care asociază fiecărei muchii un cost.

Colecţiile uzuale sunt : tabelele de dispersie, cozile, stivele, dicționarele, şi listele.

b) ***Unicitatea elemetelor*** din colecție:

- *Colecții cu elemente distincte* : mulțimea
- *Colecții cu elemente multiple* : lista

c) ***Prezența*** sau absența **unei "*chei*"**:

- *Colecții cu cheie* – dacă o parte din element (cheia) este relevantă pentru accesul la un element din colecție. Cheile se compară folosind operatori relaționali.
- *Colecții fără cheie*

d) Posibilitatea **definirii unei *relații de egalitate*** *între valorile elementelor sau între valorile cheilor, deci a unei operații de căutare în colecție*: Pentru colecţiile care au definită cheie, pentru tipul cheie trebuie să fie definită relaţia de egalitate, colecţia fiind cunoscută ca o *colecţie cu egalitate de chei*.

- *Colecții cu operație de egalitate* - două elemente din colecţie sunt egale dacă toate componentele lor sunt egale
- *Colecții fără operații de egalitate* (secvențe de elemente)

Colecţiile pentru care nu este definită nici egalitatea de chei, nici de elemente (ca de exemplu secvenţa sau heap-ul) nu permit localizarea elementelor prin valoare sau prin testarea conţinutului.

e) ***Existența unei relații de ordine*** *între elemente:*

- *Colecții ordonate (sortate)* - cu elementele sortate printr-o relaţie de ordine. De exemplu elemente şiruri de caractere sortate lexicografic (alfabetic). Un element dintr-o colecţie sortată poate fi accesat rapid, folosind relaţia de ordine pentru a-i determina poziţia. Colecţiile neordonate pot fi şi ele implementate astfel încât să permită acces rapid la elemente (de exemplu tabela de dispersie).

- *Colecții neordonate*– între elementele cărora nu există nici o relaţie de ordine

De observat că nu orice tip de elemente este ordonabil; de aceea una din proprietăţile unui container poate fi *sortabilitatea* lui. (De exemplu, *o listă de figuri geometrice nu este sortabilă*).
O colecţie sortată trebuie să aibă definită fie egalitatea cheilor, fie a elementelor.

In unele colecţii liniare ca: mulţimea, mulţimea cu chei, dicţionarul, etc nu pot exista două elemente egale sau două elemente cu chei egale. Asemenea colecţii se numesc *colecţii unice*.

Alte colecţii precum: grupul, grupul cu chei, relaţia, heapul pot avea două elemente egale sau două elemente cu chei egale. Acestea reprezintă *colecţii multiple*.

Nu există o colecţie unică, fără egalitate de chei şi egalitate de elemente, deoarece pentru o asemenea colecţie nu ar putea fi definită o funcţie de apartenenţă.

Dintre colecţiile liniare cu acces restrictiv:

- stiva, coada şi coada cu două capete se bazează pe secvenţă
- coada prioritară se bazează pe grupul cu chei sortate

f) ***Posibilităţi de acces*** *la elementele colecţiei:*

- *cu acces la orice element din colecţie, pe baza poziţiei în colecţie*
- *cu acces la orice element din colecţie, pe baza valorii elementului*
- *cu acces limitat la primul şi/sau la ultimul element din colecţie*

g) ***Limitarea numărului de elemente*** *din colecţie:*

- *Colecţii cu dimensiune limitată*
- *Colecţii cu dimensiune nelimitată*

h) ***Utilizarea colecţiei***:

- Colecţii pentru *memorarea temporară* a unor date (de tip buffer), care au un *conţinut foarte volatil: stive, cozi, multimi*, s.a.
- *Colecţii de căutare*, cu un conţinut mai stabil şi cu *operaţii frecvente de căutare*: *liste, dicţionare, arbori*, s.a.Uneori se consideră că o SD generală este o colecţie de înregistrări (structuri). Pentru SD abstracte folosite în căutare se evidenţiază un câmp discriminant, folosit la identificarea unică a fiecărei înregistrări şi numit *cheie* (Key) sau *cheie de căutare* (Search Key). Cheia poate fi şi o combinaţie a două câmpuri din înregistrare (de ex. concatenarea a două şiruri, cum ar fi numele şi prenumele).

i) *Natura elementelor componente:*

- *O colecţie directă conţine chiar datele aplicaţiei*, toate componentele sunt de un acelaşi tip, deci este o *colecţie omogenă* .
- *Un colecţie indirectă conţine pointeri (adrese) la date alocate dinamic*, date care pot fi de tipuri diferite, deci o *colecţie eterogenă* de date.

Utilizarea colecţilori indirecte se justifică prin:

a) *economia de memorie* în cazul că elementele sunt şiruri de caractere de lungime foarte variabilă;

b) *economia de timp*- dacă obiectele memorate ocupă multă memorie (structuri mari), se evită operaţiile de copiere a datelor care se înlocuiesc prin copierea pointerilor

c) *reunirea de date diferite*, cum ar fi o colecţie de figuri geometrice ce formează împreună un desen.

d) Pentru a avea un *container general*, cu pointeri la un tip neprecizat, înlocuiţi ulterior cu pointeri la tipuri precise, necesare aplicaţiei. Aceasta este soluţie de a avea o bibliotecă de subprograme generale pentru operaţii cu structuri de date uzuale, înainte de apariţia claselor derivate şi de introducerea *tipurilor parametrizate* (sau *generice*) în C++ (*template*).

Criterii de alegere a colecţiei.

La *alegerea tipului colecţiei celei mai adecvate* pentru rezolvarea unei probleme se consideră următoarele criterii:

- *Cum sunt accesate valorile ?*

Dacă este important accesul direct, atunci se folosesc colecţiile vector sau coadă cu două capete.
Dacă valorile sunt accesate conform unei relaţii de ordonare, atunci se foloseşte mulţimea ordonată.

In caz că este suficient accesul secvențial atunci sunt potrivite:lista şi variantele cu acces restrâns: stiva şi coada.

- *Este importantă ordinea în care se păstrează valorile în colecție?*

Dacă ordonarea este importantă, atunci se foloseşte mulțimea ordonată.
De asemenea pot fi folosite lista sau vectorul, la care sortarea să se faca după un număr de inserari.
In caz că prezintă importanță ordinea în care se fac inserările se va folosi o stivă sau o coadă.

- *Dimensiunea colecției se modifică în limite largi în timpul execuției?*

In caz afirmativ, cea mai bună selecție o reprezintă lista.
In caz că dimensiunea colecției este relativ stabila se foloseşte vectorul sau coada cu două capete.

- *Este posibilă estimarea dimensiunii colecției?*

Colecția vector ne permite, în acest caz să alocăm un bloc de memorie de dimensiune data.

- *Reprezintă testul de apartenență o operație frecventă?*

In caz afirmativ se folosesc mulțimea sau dicționarul.

- *Este colecția indexată (poate fi văzută ca o serie de perechi cheie – valoare)?*

Dacă cheile sunt întregi se folosesc vectorul sau coada cu două capete.
Dacă cheile reprezintă numai valori ordonate, atunci dicționarul reprezintă selecția potrivită.

- *Pot fi comparate valorile din colecție între ele?*

Dacă valorile nu pot fi comparate folosind operatorul relațional <= atunci nu se poate folosi mulțimea, nici dicționarul.

- *Reprezintă găsirea şi ştergerea elementului maxim o operație frecventă?*

In caz afirmativ se foloseşte coada prioritară.

- *In ce poziție se inserează sau se şterg elementele din colecție?*

Dacă inserarea şi ştergerea se face în orice poziție, atunci lista reprezintă cea mai bună alegere.
Dacă aceste operații se fac numai la început, lista sau coada cu două capete reprezintă structuri potrivute.
Dacă valorile se insereaza şi se şterg la un singur capăt, atunci se foloseşte stiva.

- *Este interclasarea o operație frecventă?*

In caz afirmativ se utilizează mulțimea sau lista.
In situațiile în care pot fi folosite mai multe colecții diferite, se iau în considerare timpii de execuție ai algoritmilor corespunzatori structurilor alese.

În alegerea celor mai potrivitestructuri de date, în funcție de specificul aplicației se fac următoarele recomandări:

- Dacă *numărul de componente poate fi estimat* destul de *exact şi nu se mai modifică* atunci se vor alege *tablouri cu alocare constantă* (vectori, matrice), care permit accesul direct la orice componentă şi scrierea unor programe simple şi usor de înțeles; de exemplu pentru anumite probleme cu stive, cu cozi, cu grafuri;
- Dacă *numărul de componente este foarte variabil dar* poate fi *cunoscut* înaintea valorilor componentelor şi *nu se mai modifică pe parcursul programului*, atunci se pot folosi *vectori alocati dinamic*;
- Dacă numărul de componente este foarte variabil sau dacă se modifică mult în cursul programului, atunci se vor folosi structuri de date dinamice, cu pointeri de legătură între componente: liste înlănțuite sau arbori cu legături; în funcție de aplicație poate fi necesară menținerea unei ordonări a componentelor structurii de date;
- Dacă sunt necesare căutări frecvente după conținut (după chei) şi sunt multe modificări (adăugări şi ştergeri), atunci soluția optimă poate fi un arbore binar (dinamic) sau un tabel de dispersie (hash); un arbore binar ordonat poate însă menține şi o relație de ordine între componente.

Eficiența utilizării structurilor de date

In mod ideal proiectul unei aplicaţii este gândit în structuri de date abstracte şi în module program (funcţii) care traduc un anumit algoritm.

O *structură de date abstractă* poate fi materializată prin diferite *structuri de date concrete*. Alegerea unei implementări dintre cele posibile se face, teoretic, după performanţele relative ale fiecărei SD concrete (funcţie de dimensiunea colecţiei de date).

Multe lucrări prezintă asemenea formule de calcul pentru *timpul mediu* (sau *timpul maxim*) de căutare într-o structură sau alta. De exemplu, timpul mediu de căutare într-un vector neordonat sau într-o listă (ordonată sau neordonată) este de ordinul `N/2`, iar timpul maxim este de ordinul `N`, unde `N` este dimensiunea vectorului (listei).

Timpul mediu de căutare într-un arbore binar ordonat echilibrat sau într-un vector ordonat după cheile de căutare este de ordinul $\log_2(N)$.

Pentru alte structuri de date (de ex. tabel de dispersie) timpul de căutare este mai greu de estimat, pentru că depinde şi de alţi factori.

Trebuie spus totuşi că dimensiunea relativ mică a colecţiilor de date din memoria internă combinată cu viteza mare de calcul a procesoarelor actuale şi cu durata mică a operaţiilor elementare (de obicei comparaţii şi incrementări) fac ca diferenţa absolută de timp între utilizarea unor SD diferite să fie de multe ori nesemnificativă, relativ şi la alţi timpi (calcule cu numere reale, operaţii cu fişiere disc, s.a.).

De aceea, în practică poate fi mai importantă simplitatea şi lungimea subprogramelor care realizează operaţiile cu SD respectivă, sau posibilitatea de reutilizare a unor subprograme existente.

Pe de altă parte, în cazul colecţiilor de date memorate pe disc (fişiere, baze de date), timpul mare de acces la disc şi dimensiunea mai mare a colecţiilor de date fac ca aceste estimări de performanţe să fie importante, iar diferenţa dintre diferite soluţii de organizare (structurare) a colecţiei de date să fie mai importantă pentru performanţele de ansamblu ale aplicaţiei.
Eficienţa alegerii unei SD nu se reduce la *durata operaţiilor de prelucrare* ci şi la *memoria ocupată* (cu adrese de legătură, informaţii asociate blocurilor alocate dinamic sau alte informaţii auxiliare).

Alegerea între o structură cu pointeri şi o structură vector depinde de:

- Lungimea variabilelor pointer şi dimensiunea datelor aplicaţiei.
- Numărul de colecţii de dimensiune variabilă şi posibilitatea de estimarea dimensiunii maxime pentru aceste colecţii.

De exemplu, *o stivă vector este preferabilă atunci când numărul de elemente puse în stivă* poate fi estimat destul de corect sau *nu poate fi foarte mare*, ca în cazul generării sau interpretării expresiilor postfixate.

Complexitatea algoritmilor.

Calitatea unui program depinde de corectitudine, comoditatea interfeței cu utilizatorul, uşurința întreținerii, robustețea şi nu în ultimul rând eficiența.

Eficiența unui algoritm caracterizează resursele consumate de algoritm la execuție (timp de execuție – *eficiența timpului* şi memorie consumată – *eficiența spațiului*).
Exprimarea timpului de execuție în unități de timp nu este semnificativă, deoarece s-ar referi mai mult la performanțele calculatorului decât la cele ale algoritmului. Chiar dacă se are în vedere un singur calculator, se pot obține performanțe diferite folosind compilatoare diferite pentru acelaşi limbaj de programare.

Notații asimptotice.

Analiza algoritmică (numită şi *analiză asimptotică*) caracterizează comportarea la execuție a algoritmului independent de platformă, compilator sau limbaj de programare.

Din acest punct de vedere, exprimarea *numărului de operații elementare* funcție de *dimensiunea problemei* (volumul datelor de intrare) caracterizează cel mai bine complexitatea algoritmului.

$n_{op} = T(N)$

Funcția T poate fi:

- constantă: $T(N) = c_0$
- liniară $T(N) = c_0N + c_1$
- pătratică: $T(N) = c_0N^2 + c_1N + c_2$
- polinomială: $T(N) = c_0N^p + \ldots + c_p, \quad p > 2$
- exponențială: $T(N) = c_0a^n, \quad a > 1$

Vom spune că funcția de cost $T(N)$ este *dominată* de funcția $f(N)$, dacă există o constantă pozitivă c astfel încât:

$T(N) \leq c.f(N)$

Dominanța asimptotică presupune existența constantelor pozitive c şi N_0 a.î.:

$T(N) \leq c.f(N)$ pentru $\forall\ N \geq N_0$

Pentru două funcții nenegative T şi f, spunem că T *este de ordinul lui* f, dacă şi numai dacă f domină asimptotic pe T şi vom nota aceasta prin:

$T = O(f)$

Pentru dimensiuni mari ale problemei (N mare), în forma polinomială termenul c_0N^p este predominant în raport cu ceilalți, care pot fi neglijați.

Pentru exprimarea complexității unui algoritm se utilizează *notația* O.

Prin definiție:

$T(N) = O(f(N))$ dacă $\exists\ N_0 \in N^*, \exists\ c > 0,$
a.î. $\forall\ N \geq N_0,: T(N) < c*f(N)$

$T(N) = \Omega(f(N))$ dacă $\exists\ N_0 \in N^*, \exists\ c > 0,$
a.î. $\forall\ N \geq N_0, : T(N) > c*f(N)$

$T(N) = \Theta(f(N))$ dacă $\exists\ N_0 \in N^*, \exists\ c_1, c_2 > 0$
a.î. $\forall\ N \geq N_0, : c_1f(N) < T(N) < c_2f(N)$

Constanta c_0 influienţează mai puţin creşterea timpului de execuţie decât gradul polinomului p, motiv pentru care nu mai apare în expresia complexităţii algoritmului.

Notaţia O dă o estimare a limitei superioare a complexităţii algoritmului T(N), iar Ω o estimare a limitei inferioare.

Pentru estimarea complexităţii folosind notaţia O vom folosi proprietăţile:

O(c * f) = O(f)

O(f * g) = O(f) * O(g)

O(f / g) = O(f) / O(g)

O(f+g) = Max[O(g), O(g)]

$O(f) \geq O(g)$ dacă şi numai dacă f domină pe g

Pentru a aprecia influienţa complexităţii asupra timpului de execuţie vom considera că o operaţie elementară durează 10^{-6}sec; în acest caz complexitatea în timp va fi pentru N=100:

$\log_2 N$ $6.5 / 10^6 = 6.5*10^{-6}$ sec

N $10^2 / 10^6 = 10^{-4}$ sec

$N \log_2 N$ $6.5*10^2 / 10^6 = 6.5*10^{-4}$ sec

N^2 $10^4 / 10^6 = 10^{-2}$ sec

N^3 $10^6 / 10^6 = 1$ sec

2^N $2^{100}/10^6$ sec $\cong 10^{24}$ sec $\cong 3.10^{14}$ secole

Creşterera performanţelor calculatorului nu se reflectă în aceeaşi măsură asupra algoritmului rezolvat, ci este dependentă de complexitatea algoritmului:

Astfel dacă creşterea vitezei calculatorului este de 10^3

Complexitatea algoritmului	Creşterea performanţelor algoritmului
N	1000 ori
$N \log_2 N$	$\cong$ 140 ori
N^2	$\cong$ 31 ori
N^3	10 ori
2^N	$\cong$ +10 intrări
3^N	$\cong$ +6 intrări

Complexitatea algoritmului este dependentă de configuraţia datelor de intrare. Se introduc noţiunile de *cea mai bună comportare a algoritmului*, *comportare în medie a algoritmului* şi *comportare în situaţia cea mai nefavorabilă*. Astfel cea mai bună comportare a algoritmului de căutare secvenţială corespunde găsirii valorii căutate după o singură comparaţie; comportarea în situaţia cea mai nefavorabilă corespunde găsirii valorii în ultima poziţie, având complexitatea O(N).

Evaluarea comportării în medie a unui algoritm se face mai greu deoarece trebuiesc considerate distribuţiile statistice ale datelor de intrare. In cazul căutării secvenţiale aceasta este:

(1 + 2 +…+N) / N = (N + 1) / 2 = O(N)

Evaluarea comportării medii a unui grup de operaţii (*analiza amortizată*) poate conduce la costuri rezonabile, chiar dacă una dintre operaţiile grupului este costisitoare.

Analiza algoritmică are anumite limitări şi anume:

- pentru algorimi complicaţi analiza O poate fi imposibil de realizat

- este dificilă determinarea cazului tipic
- analiza O este o măsură grosieră, care nu poate surprinde micile diferențe dintre algoritmi
- analiza O nu este concludentă pentru volume mici ale datelor de intrare

Astfel funcțiile de cost $T_1(N) = 10^{-3}N$ şi $T_2(N) = 10^3N$ au ambele complexitatea $O(N)$ deşi prima este de un milion de ori mai rapidă decât cea de a doua.

Funcția de cost $T(N) = 10^{-5}N^5 + 10^3N^4$ are complexitate $O(N^5)$ numai când primul termen este predominant, ceeace corespunde lui $N > 10^8$; pentru $N < 10^8$ complexitatea este $O(N^4)$.

Sunt situații în care analiza algoritmică a timpului de execuție nu este un criteriu suficient pentru alegerea unui algoritm. Astfel:

- dacă programul se execută rar, costul scrierii şi depanării poate domina asupra costului execuției
- dacă datele de intrare sunt puține, atunci comportarea asimptotică nu reprezintă o măsură exactă a timpului de execuție (în acest caz constantele de proporționalitate, neglijate în analiza asimptotică pot avea un rol determinant)
- algoritmii complicați, care nu sunt documentați se evită, chiar dacă prezintă performanțe mai bune, deoarece operarea de modificări reprezintă un factor de risc

pentru algoritmii numerici, *precizia* şi *stabilitatea la perturbații* sunt elemente la fel de importante ca şi analiza algoritmică.