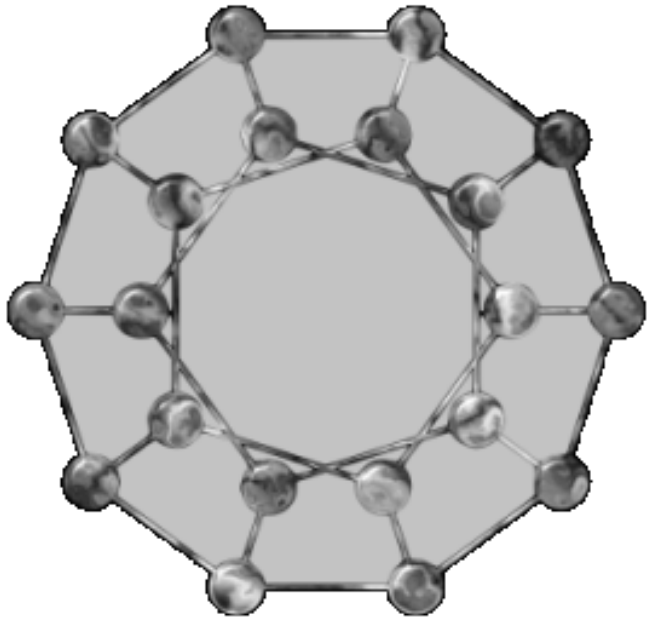
Programarea algoritmilor

Îndrumar de laborator



Gabriel M. Danciu, Alexandra Dobrinaș

2022
Editura Universitatii Transilvania

Prefață

Acest îndrumar de laborator este conceput ca suport pentru cursul de "Proiectarea Algoritmilor" predat în cadrul facultății IESC. Modul de desfășurare este următorul: laborantul va prezenta problemele la începutul laboratorului.

Desigur dacă există lămuriri necesare, se va insista asupra acestora însă nu mai mult de o cincime din timpul alocat laboratorului (care este în mod uzual 100 de minute). Apoi studenții vor fi rugați să lucreze la temele de laborator, urmărind observațiile menționate în laboratorul introductiv.

Timpul necesar îndeplinirii sarcinilor reprezintă cam 70% din timpul total alocat laboratorului. În această perioadă, studenții vor pune întrebări ajutătoare și vor ruga laborantul să le explice sau să îi ajute punctual în momentul în care aceștia se blochează. Dacă un student nu reușește să îndeplinească sarcina în timpul laboratorului, este binevenit să rezolve problema acasă și să o prezinte data viitoare, insistând pe ce anume a rezolvat și a modului în care a a făcut-o. Aceasta se admite excepțional și nu se admit alte întârzieri.

De regulă, laborantul va nota la sfârșitul fiecărui laborator fiecare student cu o notă ce surprinde evoluția acestuia în acea sesiune. La finalul semestrului, se va face o medie aritmetică a tuturor notelor și acea valoare rotunjită va fi nota finală a laboratorului. Această notă va avea o pondere de 40% în media finală, restul de 60% fiind nota dobândită în examenul de la finalul semestrului.

Pentru a putea avea succes în cadrul acestui laborator, cunoștințe anterioare de Java sunt necesare pentru a putea implementa algoritmii studiați.

Pentru fiecare laborator există cod scris în Java, menit să ofere un exemplu pentru a înlesni indeplinirea sarcinilor de lucru. Aceste resurse pot fi consultate la următorul link:
https://github.com/danciugaby/Laboratoare_Algoritmi
De asemenea informații adiționale, cărți recomandate, se pot consulta aici:
https://danciugabriel.ro

Cuprinsul detaliat

Cuprinsul detaliat

1 Introducere în algoritmi

În cadrul acestui prim laborator ne vom familiariza cu cei mai simplii algoritmi pentru ca mai apoi să întelegem algoritmi din ce în ce mai complecși și aplicabilitatea acestora [6].

1.1 Partea Teoretică

1.1.1 Sortarea șirurilor

Un algoritm de sortare este un algoritm care pune elementele unui șir de elemente într-o anumită ordine. Cel mai frecvent, se realizează ordonarea numerică și cea lexicografică.

Utilizarea unei sortări eficiente este importantă pentru optimizarea altor algoritmi pentru care datele de intrare trebuie să se afle în liste sortate. În orice situație în care se caută obiecte (elemente), sortarea este prezentă.

Sortarea este utilă și pentru *canonicalizarea* datelor, dar și pentru producerea de rezultate citibile de către om. Formal, datele de ieșire ale oricărui algoritm de sortare îndeplinesc condițiile:

1. Ieșirea este în ordine crescătoare (fiecare element nu este mai mic decât elementul anterior conform ordinii totale dorite);

2. Ieșirea este o permutare (o reordonare, dar care păstrează toate elementele originale) a intrării.

Intrarea se consideră *o instanță* a problemei. O instanţă a unei probleme este reprezentată de toate intrările necesare pentru a determina o soluţie. Dacă un algoritm rezolvă o problemă dată atunci se consideră ca este un algoritm corect. Un algoritm incorect nu va rezolva toate instanţele unei probleme, determinînd alte rezultate decât cele aștepate.

Fie un șir de numere, ce nu sunt neapărat ordonate în nici un fel. Se cere sortarea acestui șir.

Sortarea prin inserție

Figura 1.1 prezintă modul în care lucrează algoritmul de sortare prin inserție pentru un șir dat. Algoritmul parcurge tot șirul de la stânga la dreapta, inserând la fiecare pas elementul curent în șirul nou format, deja sortat. Elementele din stânga elementului curent reprezintă elementele deja sortate, iar elementele din dreapta sunt elementele ce urmează a fi soertate. La fiecare iteraţie a algoritmului este selectat un element și pornind de la poziția de dinaintea lui, se mută toate elementele cu o poziţie la dreapta, până când se ajunge la poziţia potrivită pentru inserarea elementului.

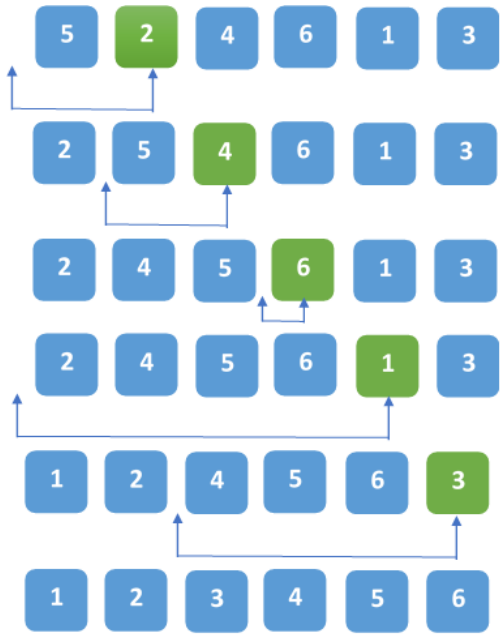


Figura 1.1: Exemplu pentru algoritmul de sortare prin inserție

Sortarea prin selecția minimului

Spre deosebire de sortarea prin inserţie, sortarea prin selecţia minimului, plasează la fiecare pas elementul selectat pe poziţia lui finală.

Algoritmul 2 funcţionează astfel: pornind de la primul element în șir, folosind variabile `minj` și `minx` pentru a reţine poziţia, respectiv valoarea celui mai mic element. Se parcurge restul șirului căutând cel mai mic element din subșirul format din elementele ce urmează elementului actual. Se găsește cea mai mică valoare, se reţine poziţia, și se interschimbă cu

elementul actual (dacă este cazul). Se continuă parcurgerea șirul până la penultimul element, aici șirul va fi deja sortat.

Acest tip de sortare este destul de rapid pentru șiruri relativ mici (sub 100 elemente). Pașii acestui algoritm sunt descriși în figura 1.2.

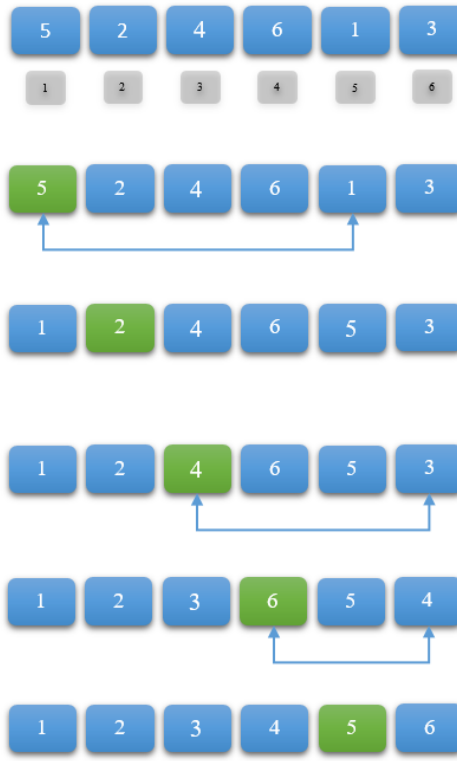


Figura 1.2: Exemplu pentru algoritmul de sortare prin selecție

1.1.2 Algoritmi elementari

Algoritmii fiind noțiunea fundamentală a informaticii, totul se construiește în jurul lor. Algoritmii elementari sunt o categorie de algoritmi care stau la baza algoritmilor complecși, care pot îngloba 2, 3 sau chiar mai mulți algoritmi elementari.

Algoritmul lui Euclid

Algoritmul ce poartă numele lui Euclid este o metodă eficientă de a calcula CMMDC (cel mai mare divizor comun) a două numere întregi.

Algoritmul se bazează pe principiul că divizorul a două numere nu se schimbă dacă scădem numărul cel mai mic din cel mare. De exemplu, CMMDC-ul lui 102 și 18 este 6. Dacă scădem 18 din 102 obţinem 84, iar CMMDC-ul lui 18 și 84 este tot 6.

Conform algoritmul lui Euclid, CMMDC-ul a două numere se poate afla astfel: se află restul împărţirii celui mai mare număr la cel mai mic și se reţine acest rest, câtul și vechiul deîmpărţit devin noile numere cărora le aplicăm aceeași operaţie, până când restul devine zero. Penultimul rest este cmmdc-ul numerelor iniţiale.

Aplicațiile practice ale acestui algoritm variază de la simplificarea reprezentării fracțiilor, la construcția sistemului de criptare RSA sau la elaborarea strategiilor de tip *look-ahead*.

Pentru exemplul ales mai sus și anume 102 și 18 algoritmul va funcționa conform schemei de mai jos, unde `temp` reprezintă restul împărțirii primului număr la cel de-al doilea:

$temp \leftarrow 18$
$n \leftarrow 12$
$m \leftarrow 18$

    $temp \leftarrow 12$
    $n \leftarrow 6$
    $m \leftarrow 12$

        $temp \leftarrow 6$
        $n \leftarrow 0$
        $\mathbf{m \leftarrow 6}$

Algoritmul lui Fibonacci

Șirul lui Fibonacci este definit prin următoarea recurenţă:

$$\begin{cases} f_0 = 0, f_1 = 1 \\ n < 2 f_n = f_{n-1} + f_{n-2} \quad n \geq 2 \end{cases} \tag{1.1}$$

Acest șir a fost descoperit de Leonardo Pisano, cunoscut sub numele Leonardo Fibonacci. Cel de-al n-lea termen din șir se poate scrie folosind definiţia și utilizarea unui algoritm recursiv (de cele mai multe ori), însă există și o variantă iterativa, cu o complexitate mai mică. Secvența Fibonacci are o semnificație mai mare decât cea educațională. În lumea naturală există diferite specii de plante care se dezvoltă în conformitate cu această serie sau au în componență părți ce pot fi aproximate cu ajutorul acestui șir. De asemenea, anumite metodologii de dezvoltare și estimare de applicații software, folosesc seria Fibonacci pentru a determina timpul necesar rezolvării unui proiect.

Înmulțirea matricilor

Una dintre operațiile des folosite pentru matrici este înmulțirea lor. Deoarece pe parcursul acestui laborator se va lucra destul de mult cu matrici, pentru introducere în utilizarea matricilor este prezentat algoritmul de înmulțire.

Această operație necesită ca matricile să respecte condiția ca lățimea primei matrici (numărul de linii) sa fie egală cu înălțimea celei de-a doua (numărul de coloane). Astfel multiplicând o matrice de dimensiune mxn cu una nxp, unde m, n, p nu sunt neapărat egale, se va obține o matrice nxp.

Fiecare element al produsului matricilor se calculează pe baza relației:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Algoritmul care are la bază această formulă de calcul, nu este cel mai optim mod de calcul al produsului. Un algoritm eficient pentru calculul produsului este algoritmul inventat de Volker Strassen.

1.1.3 Înmulțirea a la Russe

Este o metodă ce nu implică folosirea tablei înmulţirii, ci doar împărțirea la 2 și adunarea. Este o metodă care poate fi implementată ușor în tehnica de calcul modernă.

Principiul acestei metode este de a împărții în mod repetat la 2 primul număr și dublarea (adunarea cu el însuși) a celui de-al doilea număr.

De exemplu pentru numerele 52 și 15 algoritmul funcționează astfel:

52	15	–
26	30	–
13	60	60
6	120	–
3	240	240
1	480	480
		780

Tabela 1.1: Înmulțirea a la russe pentru numerele 52 și 15

În cazul în care împărțirea cu 2 a primului număr a avut ca rezultat un număr impar (prima coloană), se reține, în a treia coloană, rezultatul dublării celui de-al doilea număr (doua coloană). Pentru a obține rezultatul (produsul dintre numerele inițiale) se însumează numerele din a treia coloană.

Această metodă are aplicații în domeniul educațional, fiind o metodă ușor de înțeles și de implementat, dar și în domeniul ingineresc deoarece aduce o îmbunătățire a timpului de execuție acolo unde numerele sunt foarte mari.

1.2 Partea Practică

Prezentarea algoritmilor în pseudocod este următoarea:

1.2.1 Sortarea prin inserție

Algoritm 1 Sortarea prin inserție

```
1:  procedure INSERTION − SORT(A)
2:      for j ← 2 to LENGTH[A] do
3:          key ← A[j]
4:        ▷ inserează A[j] în secvența sortată A[i..j-1]
5:          i ← j − 1
6:          while i > 0 AND A[i] > key do
7:              A[i + 1] ← A[i]
8:              i ← i − 1
9:          end while
10:         A[i + 1] ← key
11:     end for
12: end procedure
```

1.2.2 Sortarea prin selecția minimului

Algoritm 2 Sortare prin selecție

```
1: procedure SELECTION-SORT(A[1..n])
2:     for i ← 1 to n-1 do
3:         minj ← i; minx ← A[i]
4:       ▷ caut poziția finală a lui A[i] în șir
5:         for j ← i+1 to n do
6:             if A[j] < minx then
7:                 minj ← j
8:                 minx ← A[j]
9:             end if
10:        end for
11:     ▷ la final schimb elementul actual cu cel mai mic găsit
12:        A[minj] ← A[i]
13:        A[i] ← minx
14:    end for
15: end procedure
```

1.2.3 Algortitmul lui Euclid

Algoritm 3 Algoritmul lui Euclid

```
1: procedure EUCLID(m, n)
2:     while n ≠ 0 do
3:         temp ← n
4:         n ← m%n
5:         m ← temp
6:     end while
7: return m
8: end procedure
```

1.2.4 Șirul lui Fibonacci

Algoritm 4 Calculul termenului n al șirului Fibonacci.
Varianta recursivă

```
1: procedure FIBO_R(n)
2:     if n < 2 then   return n
3:     else  return FIBO_R(n-1) + FIBO_R(n-2)
4:     end if
5: end procedure
```

Algoritm 5 Calculul termenului n al șirului Fibonacci.
Varianta iterativă

```
1: procedure FIBO_I(n)
2:     i ← 0; j ← 1
3:     s ← 1
4:     for k ← 1 to n do
5:         i ← j
6:         j ← s
7:         s ← i + j
8:     end for
9: return i
10: end procedure
```

1.2.5 Produsul a două matrici

Algoritm 6 Produsul a două matrici

```
1: procedure PRODUS(A, B)
2:     for i ← 1 to m do
3:         for j ← 1 to p do
4:             for k ← 1 to n do
5:                 C_{i,j} ← C_{i,j} + A_{i,k} B_{k,j}
6:             end for
7:         end for
8:     end for
9: end procedure
```

1.2.6 Înmulțirea a la russe

Algoritm 7 Înmulțirea a la russe

```
1: procedure RUSSE(a, b)
2:     arrays X, Y
3:     X[1] ← a; Y[1] ← b
4:     i ← 1
5:     ▷ se construiesc cele două coloane
6:     while X[i] > 1 do
7:         X[i + 1] ← X[i]/2
8:         Y[i + 1] ← Y[i] + Y[i]
9:         i ← i + 1
10:    end while
11:    prod ← 0
12:    while i > 0 do
13:        if X[i]%2 = 1 then
14:            prod ← prod + Y[i]
15:        end if
16:        i ← i − 1
17:    end while
18: return prod
19: end procedure
```

1.3 Tema de laborator

Implementați algoritmii prezentați în Java.

Observații:

- se va folosi o singură clasă;
- fiecare algoritm va reprezenta o funcție (metodă) a clasei;
- datele vor fi citite dintr-un fișier sau li se vor da valori încă de la declarare (**nu se va folosi citirea de la tastatura, în consolă**);
- funcția *main* va fi folosită pentru citirea datelor, apelul funcțiilor și afișarea rezultatelor(de așa natură încât să se poată înțelege ce reprezintă);
- dacă este nevoie alături de rezultate se vor afișa și datele inițiale;

2 Structuri de date

2.1 Partea Teoretică

Agner Krarup Erlang, matematician și inginer danez care a lucrat la Copenhagen Telephone Company, a publicat în 1909 prima lucrare privind teoria așteptării în care sunt enunțate noțiunile FIFO, LIFO care, mai târziu, au stat la baza structurilor de date *coadă* respectiv, *stivă*. Totodată, *Erlang* este unitatea de măsură pentru volumul trafcului din telecomunicații [4]. Stiva, utilizată în arhitectura calculatorului a fost patentată în 1957 de Friedrich L. Bauer, un pioner în știința calculatoarelor, de origine germană.

Tipurile de operaţii care pot fi efectuate pe mulţimi dinamice (cu număr extensibil de elemente) sunt prezentate sub forma unor proceduri, pentru a abstractiza mediul de programare folosit. Acestea se împart în două tipuri:

1. interogare - vor returna informaţii despre elementele din mulţime:
   a) `SEARCH(S,k)`- returnează poziția pe care se găsește elementul `k` în structură sau `NULL`, în caz contrar;
   b) `MINIMUM(S)` - returnează cea mai mică valoare din structură;
   c) `MAXIMUM(S)` - returnează cea mai mare valoare din structură;
   d) `SUCCESOR(S,x)` - returnează următorul element aflat după `x` din S, sau `NULL`, dacă după `x` nu mai urmează niciun alt element.
   e) `PREDECESOR(S,x)`returnează element aflat înainte de `x` din S, sau `NULL`, dacă inainte de `x` nu maieste niciun element.
2. modificare - vor modifica elemente din mulţime:
   a) `INSERT(S,x)` - inserează în structură elementul `x`;
   b) `DELETE(S,x)` - șterge din structură elementul `x`;

Așa cum se va observa în următoarea parte a laboratorului, fiecare structură de date, are operațiile specifice.

2.1.1 Stiva și coada

Stiva și coada sunt mulţimi dinamice în care un element va fi adăugat sau șters prin intermediul unor operaţii specifice.

Stiva este o structură de dată de tip last-in first out, sau **LIFO**.

Operaţia INSERT de mai sus se numește în cazul stivei și `PUSH`, iar DELETE (operaţie care nu are ca argument pe x) se numește `POP`. Aceste nume sunt consacrate și provin de la ordinea in care farfuriile sunt aranjate în stive, în restaurante de exemplu.

Elementul numit `top` reprezintă elementul cel mai recent adăugat în stivă sau altfel spus, *vârful stivei*. Atunci când `top=0` se zice că stiva este goală. Dacă se încearcă scoaterea unui element dintr-o stivă goală, spunem că stiva va avea un *underflow* adică o operaţie eronată. Atunci când `top=n` (`n` – numărul de elemente din stivă) și se4 încearcă introducerea unui element nou, se cheamă că stiva are un *overflow*, de asemenea o operaţie eronată.

Coada este structura ce se conformează principiului **FIFO** și anume primul sosit primul ieșit. Coada are `cap` (`head`), primul element și o `coadă` (`tail`), adică ultimul element. Atunci când un nou element este introdus, el devine elementul *coadă* din structură.

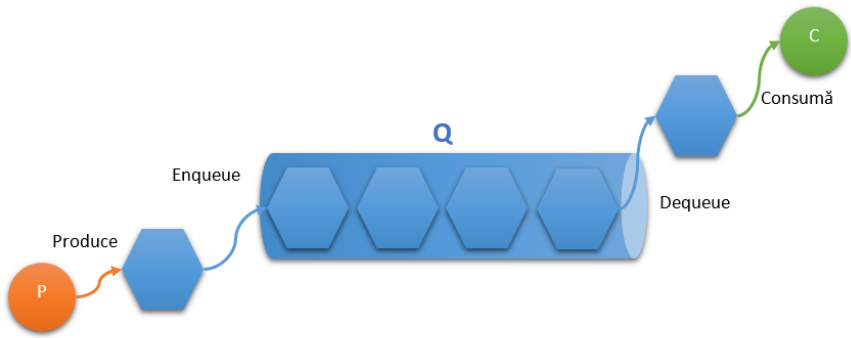


Figura 2.1: Exemplu de utilizare a cozii.

Asemenea stivei, avem operaţii specifice de adăugare și ștergere al unui element și anume `ENQUEUE` pentru introducerea unui nou element și `DEQUEUE` pentru ștergerea (eliminarea, afișarea) primului element din coadă.

Atunci când `head=tail` înseamnă că avem o coadă goală. Iniţial orice coadă reprezentată astfel, are `head=tail=1`. Încercarea de a scoate dintr-o

coadă goală duce la *underflow*, iar o încercare de a introduce dintr-o coadă plină, duce la *overflow*.

2.1.2 Listele înlănțuite

Dacă elementele care formează o stivă sau coadă erau elemente de tipuri simple de date, elementele care formează listele înlănțuite se numesc **noduri** și sunt formate din două câmpuri: **valoarea reținută** și **adresa (referință)** către următorul element.

Dacă legătura unui nod duce la el însuși, sau un nod conţine o legătură la un nod ce va direcţiona prin N legături către nodul de start, structura se numește **ciclică**.

Ultimul nod din listă poate îndeplini doar una din următoarele condiţii:

- este o legătură `NULL` ce nu indică nici un nod;
- este o legătură *dummy* ce nu conţine informaţie validă;
- conţine o legătură către primul nod, lista devenind circulară;

O **listă simplu înlănțuită** `L` va avea valoarea fiecărui element notată cu `key`, elementul `head` reprezintă primul element(nod) din listă, dacă `head=NULL`, atunci lista este goală. Referinţa către următorul element din listă este dată prin atributul `next` al fiecărui nod. Având în vedere ca pentru o astfel de listă poate fi accesat numai primul element al listei, aceasta va trebui parcursă doar de la primul spre ultimul element.

O **listă dublu înlănțuită** reprezintă o mulţime dinamică în care fiecare nod din listă este un obiect ce conţine o cheie și două referinţe la obiectele anterior, respectiv următor. Pentru acest tip de listă este reținut atât primul element al listei, cât și ultimul, astfel încât lista să poată fi parcusă și de la ultimul spre primul element.

Reprezentări grafice ale operațiilor specifice listelor simplu înlănțuite

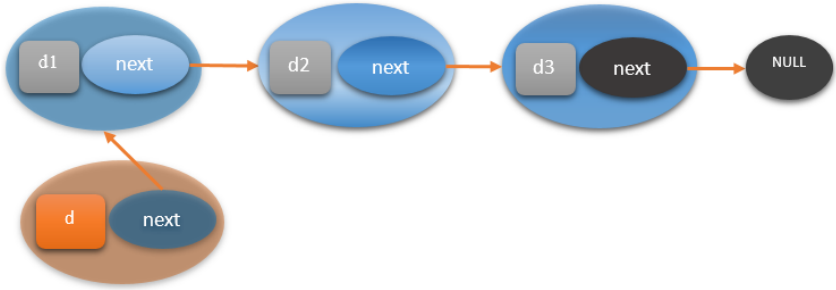


Figura 2.2: Inserarea unui nou element la începutul listei.

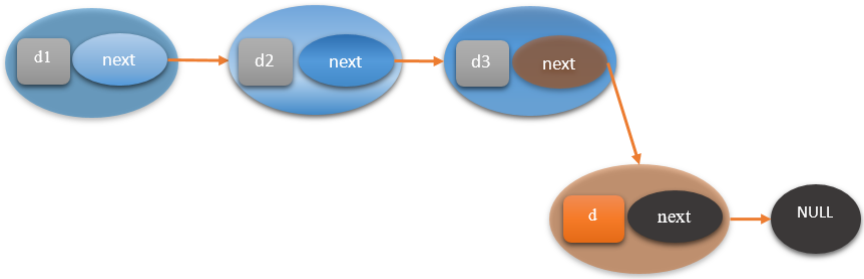


Figura 2.3: Inserarea unui nou element la finalul listei.

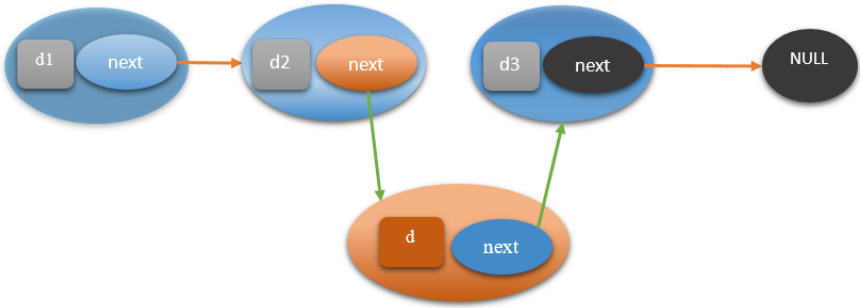


Figura 2.4: Inserarea unui nou element în interiorul listei.

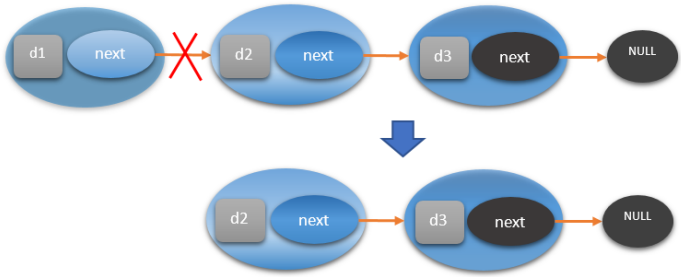


Figura 2.5: Ștergerea unui element de la începutul listei.

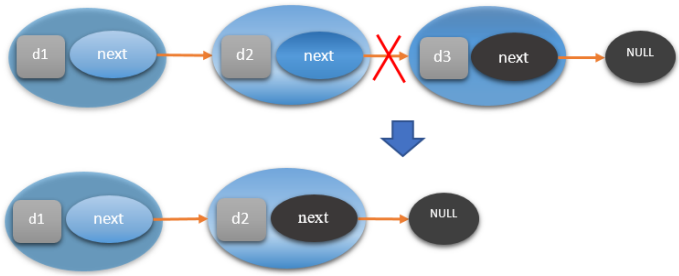


Figura 2.6: Ștergerea unui element de la finalul listei.

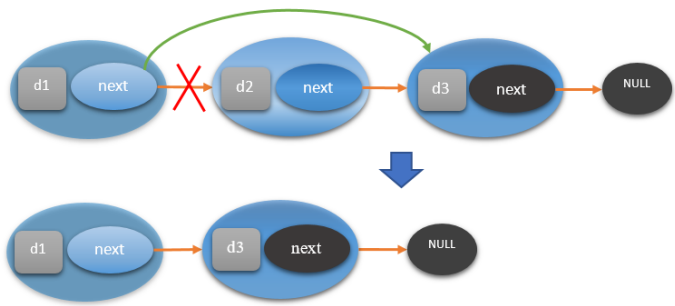


Figura 2.7: Ștergerea unui element din interiorul listei.

2.2 Partea practică

2.2.1 Coada

Algoritm 8 Adăugarea unui element într-o coadă

```
1: procedure ENQUEUE(Q, x)
2:     Q[tail] ← x
3:     if tail=length[Q] then
4:         tail ← 1
5:     else tail[Q] ← tail[Q] + 1
6:     end if
7: end procedure
```

Algoritm 9 Eliminarea unui element dinntr-o coadă

```
1: procedure ENQUEUE(Q)
2:     x ← Q[head]
3:     if head = length[Q] then
4:         head ← 1
5:     else head ← head + 1
6:     end if
7: return x
8: end procedure
```

Acești algoritmi nu tratează cazurile de *underflow* și *overflow*, deoarece implementează o *coadă circulară*, preferată în programare.

2.2.2 Stiva

Algoritm 10 Adăugarea unui element într-o stivă

```
1: procedure PUSH(S, x)
2:     top ← top + 1
3:     S[top] ← x
4: end procedure
```

Algoritm 11 Eliminarea unui element dintr-o stivă

```
1: procedure POP(Q)
2:     if STAK_EMPTY(S) then
3:         error "underflow"
4:     else top ← top − 1
5:     end if
6: return S[top + 1]
7: end procedure
```

Algoritm 12 Verificare stivă goală

```
1: procedure POP(Q)
2:     if top=0 then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
```

Algoritmul de evaluare a unei expresii postfixate

Algoritm 13

```
1: while mai există cuvinte în expresie do
2:     citește cuvântul următor
3:     if cuvântul este o valoare then
4:         push (cuvânt)
5:     else cuvântul este un operator
6:         pop() două valori de pe stivă
7:         rezultat = operația aplicată pe cele două valori
8:         push(rezultat)
9:     end if
10: end while
11: rezultat final = pop()
```

2.2.3 Liste înlănțuite

Algoritm 14 Inserarea pe prima poziție a unui element nou, într-o listă simplu înlănțuită

```
1: procedure SList_Insert(L, x)
2:     next[x] ← head
3:     head ← x
4: end procedure
```

Algoritm 15 Ștergerea unui element dat, dintr-o listă dublu înlănțuită

```
1: procedure DList_Delete(L, x)
2:     if prev[x] ≠ NULL then
3:         next[prev[x]] ← next[x]
4:     else
5:         head[L] ← next[x]
6:     end if
7:     if next[x] ≠ NULL then
8:         prev[next[x]] ← prev[x]
9:     end if
10: end procedure
```

2.3 Tema de laborator

1. Creați o clasă proprie care simulează comportamentul unei stive. Folosind această clasă, determinați valoarea expresiei $512 + 4 * +3-$, dată în forma postfixată, urmărind pașii descriși de algoritmul 13.
2. Utilizând structura de stivă, realizați un algoritm care să ofere rezolvarea determinării unui drum printr-un labirint. Folosiți ca referință https://en.wikipedia.org/wiki/Maze_solving_algorithm?fbclid=IwAR1rMrnNQjK_Na2sI5z7jJzuB8MqIWTYPdfuBOeuXxisoMtHhu1p1ZdhvJ8
3. Plecând de la algoritmi de adăugare și ștergere prezentați, dar și de la reprezentările grafice ale operațiilor specifice pentru listele simplu, respectiv dublu înlănțuite, creați câte un proiect în care să implementați o clasă pentru lista simplu înlănțuită și o clasă pentru lista dublu înlănțuită. Pentru fiecare din aceste tipuri de liste, creați câte o listă cu cel puțin 5 elemente, ștergeți al doilea element din listă și adăugați un element nou pe poziția de mijloc a listei obținute după ștergere, ștergeți din lista nou obținute primul și ultimul element. Afișați la fiecare pas noua listă.
4. Știind că ultimul element al unei liste dublu înlănțuită poate avea o referință către primul element al listei, simulați comportamentul unei **liste circulare**, implementând aceleași operații ca pentru problema 3.

3 Structuri de date 2. Reprezentarea grafurilor si a arborilor

3.1 Partea teoretică

3.1.1 Noțiuni introductive. Terminologie

Grafuri

Un graf este o pereche $G =< V, E >$, unde V este o mulțime de **vârfuri**, iar $E \subseteq V \times V$ este o mulțime de **muchii**. În graful din figura 3.1 $G = (V, E)$ cu $V = 1, 2, 3, 4, 5, 6$ și $E = (1, 2), (2, 1), (1, 4), (4, 1), (2, 5), (5, 2), (4, 5), (5, 4), (2,$

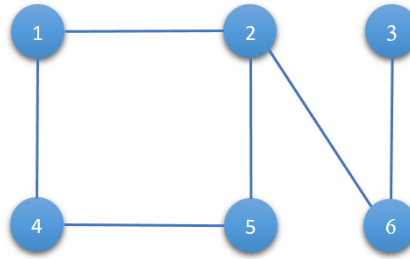
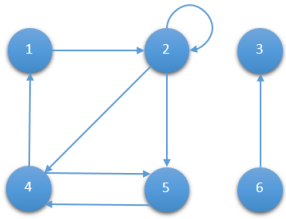


Figura 3.1: Exemplu de graf.

Un graf orientat sau **digraf** este o pereche notată $G =< V, E >$ în care fiecare pereche notată $\{u, v\}$ se numește **arc dintre nodurile u,v**, și reprezintă legătura care pleacă din u și ajunge în v. În acest tip de graf, este permis ca un arc să aibă același nod ca destinație și sursă. În figura 3.2a este prezentat un graf orientat [1].



(a) Graf orientat

(b) Subgraf obținut din graful orientat

Figura 3.2: Exemplu de graf orientat.

Un subgraf este un graf $G =< V', E' >$, unde $V' \subseteq V$, iar E' este o submulțime a lui E, formată din muchile/arcele ce unesc nodurile din V'.

Un graf parțial este un graf $G =< V_1, E" >$, în care $E" \subseteq E$, iar $V_1 = V$. Un exemplu de astfel de graf se găsește în figura 3.3. Acest graf a fost obținut plecând de la graful din figura 3.2a.

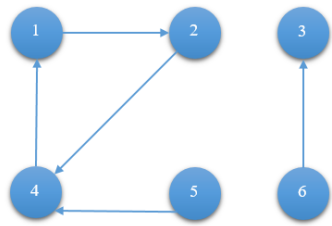


Figura 3.3: Exemplu de graf parțial.

Două vârfuri unite printr-o/un muchie/arc se numesc **vârfuri adiacente** și **incidente** cu muchia/arcul pe care îl/o formează.

Un lanț/drum este o succesiune de muchii/arce. Lungimea unui lanț/drum este egală cu numărul muchiilor care îl constituie. Un lanț/drum se numește **elementar** dacă este un lanț/drum în care nici un vârf nu se repetă. Un lanț/drum se numește **simplu** dacă este un lanț/drum în care nici o/un muchie/arc nu se repetă. Un exemplu de drum se găsește în figura 3.4 trasat cu roșu. Drumul poate fi exprimat astfel: (1,2), (2,5), (5,4).

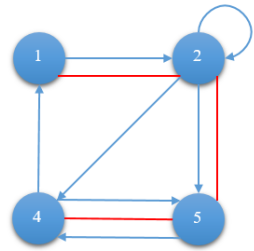


Figura 3.4: Exemplu de drum.

Un ciclu/circuit este un lanț/drum, în care primul și ultimul vârf coincid. **Un graf aciclic** este un graf neorientat fără cicluri.

Un graf conex este un graf în care între oricare două vârfuri există un lanț/drum. Un graf este **tare conex** dacă el este conex maximal (dacă s-ar mai adăuga un nod acesta nu ar mai fi conex). Pentru un graf neconex se pot determina componentele sale conexe/tare conexe. În figura 3.5 avem un exemplu de graf conex, iar în figura 3.6 avem o reprezentare a unor componente conexe.

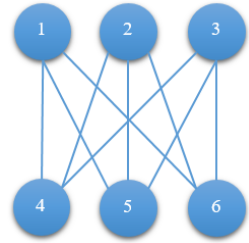


Figura 3.5: Exemplu de graf neorientat conex.

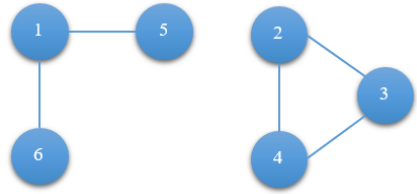


Figura 3.6: Exemplu de graf cu 2 componente conexe: 1,5,6 și 2,3,4.

Vârfurilor unui graf li se pot atașa informații/valori, iar muchiilor li se pot atașa informații numite costuri.

Arbori

Un arbore liber este un graf aciclic și conex. Altfel spus, un arbore este un graf în care există exact un lanț între oricare două vârfuri. Un arbore conține exact $|V|$ noduri și exact $|V-1|$ muchii. În figura 3.7 este reprezentat un arbore liber.

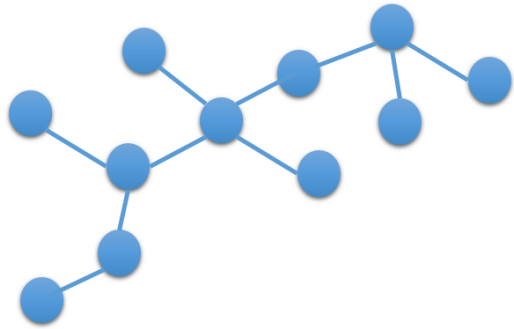


Figura 3.7: Exemplu de arbore liber.

Un graf format din doi sau mai mulți arbori se numește **pădure**. În figura 3.8 avem un exemplu de o pădure alcătuită din trei componente conexe, fiecare componentă fiind un arbore.

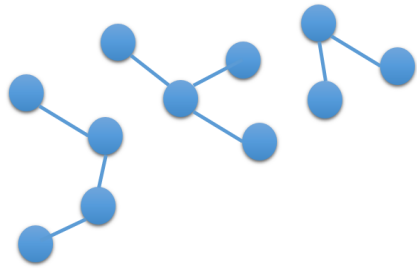


Figura 3.8: Exemplu de pădure.

Un arbore cu radăcină este un arbore în care un singur nod este desemnat ca rădăcină, iar celelalte noduri „cad". Un arbore cu rădăcină are următoarele proprietăți:

- Are un nod aparte numită *rădăcină*
- Fiecare nod, în afară de rădăcină, are un părinte unic
- Fiecare nod are 0 sau mai mulți fii.
- Un nod fără succesori se numește *frunză* sau *terminal*

Un exemplu de arbore cu rădăcină este prezentat în figura 3.9.

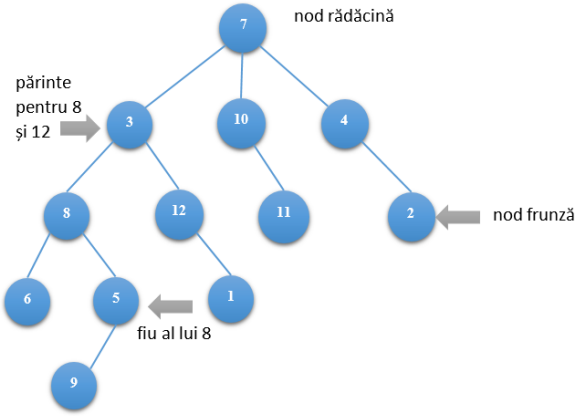


Figura 3.9: Exemplu de arbore cu rădăcină.

3.1.2 Reprezentarea grafurilor

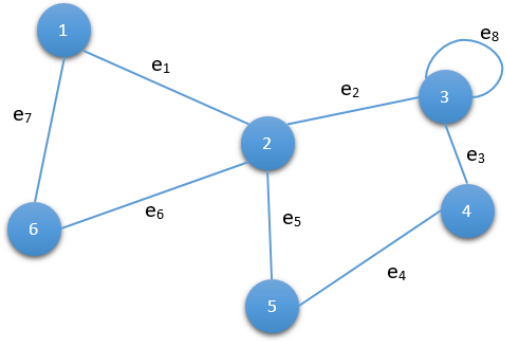


Figura 3.10: Graf neorientat.

Având ca exemplu graful din figura 3.10 vom analiza fiecare tip de reprezentare a grafurilor.

Matricea de adiacență/costurilor

Matrice de adiacență A, este o matrice de $|V| \times |V|$ în care $A[i, j] = true$ dacă nodurile i și j sunt adiacente și $A[i, j] = false$ în caz contrar.

În cazul în care fiecare muchie are atribuit un cost, $A[i, j]$ va avea ca valoare costul muchiei dintre i și j, considerând $A[i, j] = +\infty$ atunci când cele două vârfuri nu sunt adiacente. Pentru graful din figura 3.10 matricea de adiacență este cea din tabelul 3.1.

0	1	0	0	0	1
1	0	1	0	1	1
0	1	1	1	0	0
0	0	1	0	1	0
0	1	0	1	0	0
1	1	0	0	0	0

Tabela 3.1: Reprezentarea unei matrice de adiacență

Liste de muchii

O listă de muchii este un șir de perechi ordonate, fiecare pereche sau tuplu reprezentând o muchie. Dacă unei muchii ii se asociază un cost, atunci perechea devine un triplu format din nodul i, nodul j si costul c.

Pentru graful din figura 3.10, lista de muchii este următoarea:

(1,2,e_1),(2,3,e_2),(3,3,e_8),(3,4,e_3),(4,5,e_4),(5,2,e_5),(6,2,e_6),(1,6,e_7)

unde $e_{1..8}$ reprezintă costul unei muchii adică un număr întreg.

3.1.3 Liste de adiacență

Într-o listă de adiacență fiecare nod i conține un șir de noduri adiacent lui. Astfel avem o listă sau un șir de $|V|$ liste de adiacență precum cea din figura 3.11, corespunzătoare grafului din figura 3.10.

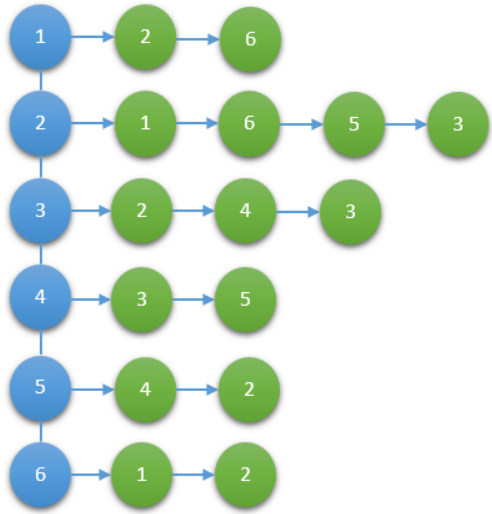


Figura 3.11: Liste de adiacență

3.1.4 Reprezentarea arborilor

Reprezentarea cu ajutorul tablourilor

Pentru a reprezenta un arbore T cu ajutorul tablourilor va trebui să:

1. Numerotăm nodurile arborelui T de la 1 la n.
2. Asociem nodurilor, elementele șirului astfel: nodului i corespunde locația i din șir.
3. Pe fiecare poziție i se memorează locația părintelui. Astfel $T[i] = j$ unde j este părintele lui i.
4. Dacă $A[i] = 0$ atunci i este rădăcina arborelui.
5. Valorile nodurilor (denumite și chei) se vor salva într-un alt șir V pe pozițiile i corespunzătoare nodurilor din arborele T. Astfel în $C[i]$ se reține valoarea din nodul i.

În figura 3.12 se găsește reprezentarea arborilor folosind șirul T pentru păstrarea pozițiilor părinților nodurilor și V pentru păstrarea cheilor sau valorilor din interiorul nodurilor.

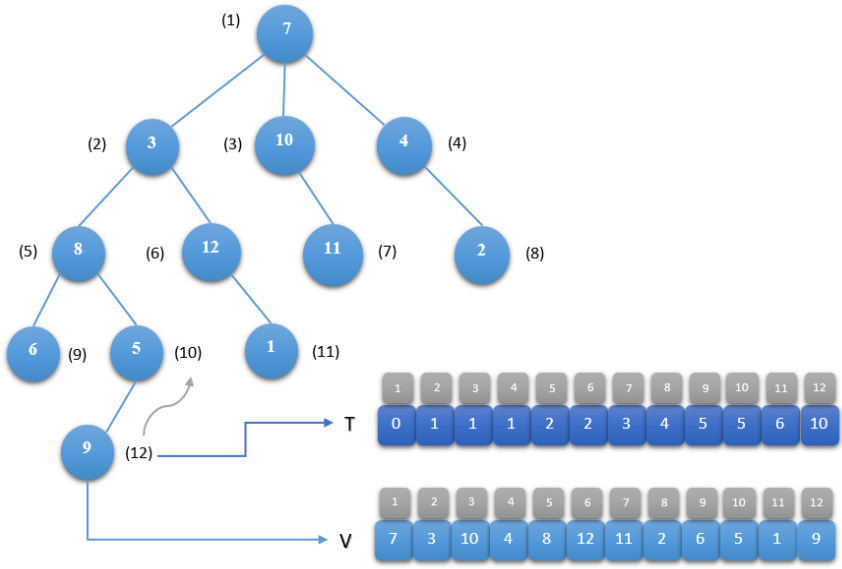


Figura 3.12: Reprezentarea arborelui cu rădăcină.

Reprezentarea cu ajutorul listelor simplu înlănțuite

Pornind de la reprezentarea grafurilor cu ajutorul listelor vom genera pentru fiecare nod o listă a fiilor săi.

În lista principală vom păstra nodurile din arbore în ordinea următoare: de la rădăcină către fii și de la stânga la dreapta.

Vârfurile ce au cel puțin un fiu vor reține o listă cu fii lor. Nodurile frunză vor indica un element NULL. Totodată în fiecare nod din listă va conține și informația despre cheia acelui nod.

În figura 3.13 este reprezentat sub forma unor liste de vecini, arborele din figura 3.9.

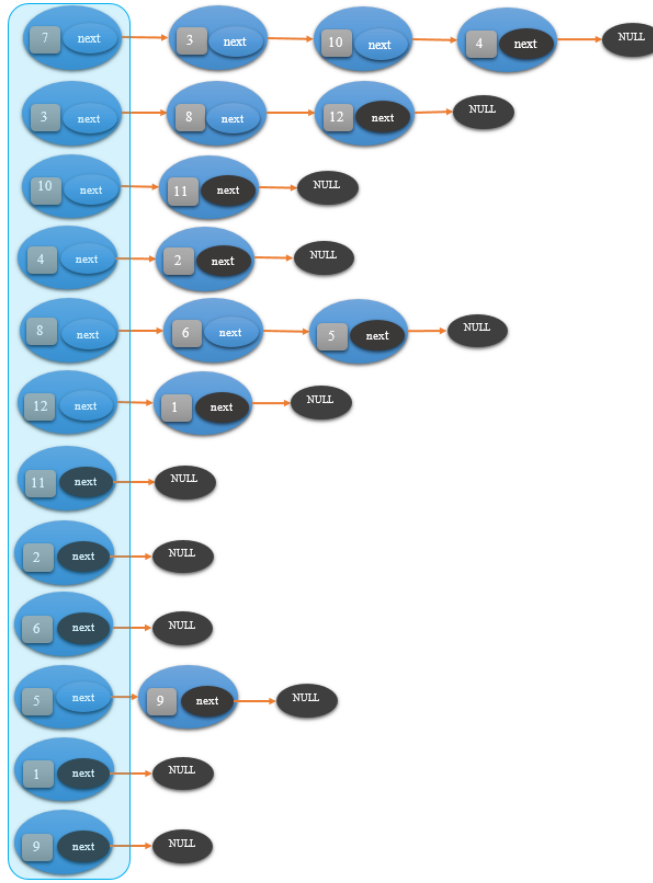


Figura 3.13: Reprezentarea ca lista de vecini a arborelui.

Reprezentarea cu ajutorul listelor multiplu înlănțuite

În cazul în care se dorește parcurgerea arborilor cât mai eficient în orice sens, și de la rădăcină spre frunze și invers, putem folosi o structură asemănătoare cu listele dublu înlănțuite care conține pe lângă valoarea (cheia) nodului două referințe:

1. Referința către părinte. În cazul în care nodul este chiar rădăcina, referința va fi NULL

2. Listă de referințe către copii. Dacă nodul este frunză atunci lista este formată dintr-un obiect NULL

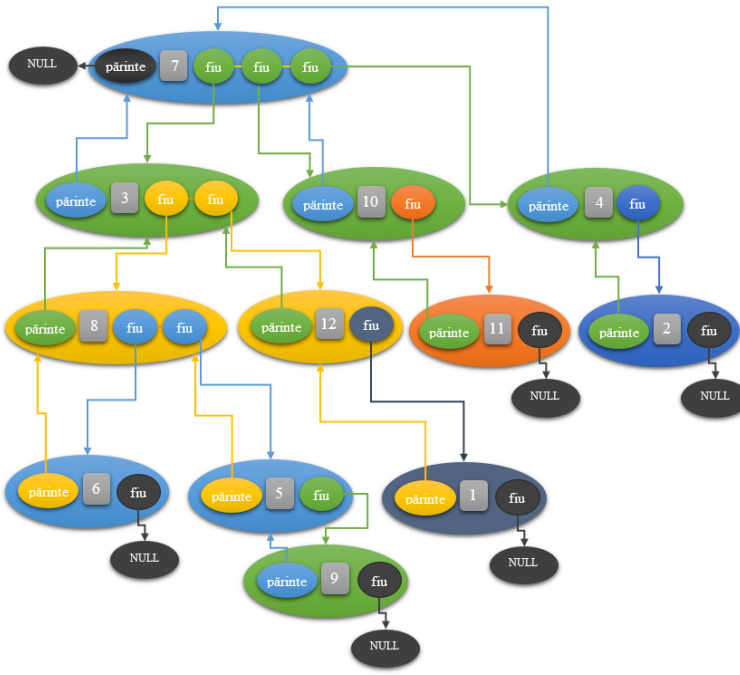


Figura 3.14: Reprezentarea ca listă multiplu înlănțuită, a arborelui.

3.2 Partea practică. Tema de laborator

Algoritm 16 Inserarea unui nou nod în arbore

```
1: procedure INSERT_NEW(T, cheie, nouaval)
2:   ▷ Alocăm memorie pentru noul nod
3:   newnode ← new Node
4:     ▷ Atribuim valoarea specifică acestuia
5:   newnode.v ← nouaval
6:     ▷ Nodul nou nu are fii
7:   newnode.fii ← NULL
8:     ▷ Parcurgem arborele până la nodul părinte al nodului de inserat
9:   node ← SEARCHTREE_BYKEY(T.RĂDĂCINĂ, CHEIE)()
10:    ▷ Refacem cele două legături conectând astfel nodul nou la arbore
11:      ▷ De la noul nod la părinte
12:  newnode.parinte = node
13:    ▷ Și de la părinte la nou nod
14:  list ← node.fii
15:  while list.next ≠ NULL do
16:      list ← list.next
17:  end while
18:  lis.next ← newnode
19: end procedure
```

Algoritm 17 Căutarea unui nod în arbore după cheie

```
1: procedure SEARCHTREE_BYKEY(node, cheie)
2:   ▷ Se preia lista de copii ai nodului curent
3:   list ← node.fii
4:   while list.next ≠ NULL do
5:  ▷ Dacă se găsește cheia în lista de fii, se returnează acel fiu
6:       if list.v = cheie then return list
7:       end if
8:     ▷ Dacă nu, se parcurge în adâncime arborele
9:       newnode ← SEARCHTREE_BYKEY(LIST, CHEIE)()
10:      if newnode.v = cheie then return newnode
11:      end if
12:    ▷ Se trece la următorul fiu din listă
13:      list ← list.next
14:  end while
15: end procedure
```

1. Pentru graful din figura 3.2a, realizați un program care în urma citirii listelor de muchii dintr-un fișier, crează și afișează matricea de aciacență și listele de adiacență. Pentru crearea listelor de adiacență se va folosii o structură de date de tipul listelor înlănțuite.
2. Fiind dat arborele cu rădăcină din figura 3.12, realizați o optimizare a reprezentării cu șiruri, astfel încât să se utilizeze un singur șir în care va fi memorată o structură de date care va conține atât informații despre părintele nodului curent, cât și cheia nodului.
3. Realizați o implementare a arborelui din figura 3.12 folosind listele multiplu înlănțuite, plecând de la exemplul de reprezentare a unui nod de mai jos. Folosind algoritmii 16 și 17 realizați inserarea și căutarea unui nod.

```
struct {
Data v;
Node parinte;
Lista<Node> fii;
} Node;
```

4 Backtracking

4.1 Partea Teoretică

Backtracking este un principiu fundamental de elaborarea a algoritmilor pentru probleme de optimizare sau de găsire a unor soluții la o problemă dată [11]. Acești algoritmi caută una sau mai multe soluții pentru problemă, realizând o căutare exhaustiva, însă mai eficientă față de o abordare „generează și testează", deoarece o soluție parțială care nu duce la o soluție finală, este abandonată.

Acești algoritmi pot fi folosiți pentru probleme care presupun o căutare în spațiul stărilor. Chiar dacă în timp ce cautăm o soluție se ajunge la un „deadend", din cauza unei alegeri greșite sau se găsește o soluție, algoritmul nu se încheie, deoarece se dorește căutarea tuturor soluților. În acest moment se realizează o întoarcere pe pașii făcuți (backtrack) și la un moment dat se va lua altă decizie. Deși, din punct de vedere conceptual, este relativ simplu de implementat, complexitatea algoritmului este exponențială.

Algoritmii backtracking se folosesc în rezolvarea problemelor care respectă următoarele condiții:

- soluția trebuie reprezentată printr-un tablou `s[]=(s[1], s[2], ..., s[n])`, unde fiecare element `s[i]` aparține unei mulțimi cunoscute `A_i`. De obicei acest tablou este o stivă;
- fiecare mulțime `A_i` este finită, iar elementele ei se află într-o relație de ordine precizată, de multe ori cele `n` mulțimi sunt identice;
- trebuie determinate toate soluțiile problemei sau o anumită soluție care nu poate fi determinată într-un alt mod (de regulă mai rapid).

Pentru construirea soluției problemei, în proiectarea algoritmului Backtracking se respectă următoarele:

- Determinarea elementelor soluției se face în ordine succesivă, incrementală. La fiecare pas se completează un element posibil al soluției. Soluția în care doar o parte din elemente sunt completate se numește soluție parțială.
- Pentru fiecare soluție parțială se stabilește dacă este validă.

- Dacă se întâlnește o soluție parțială invalidă, atunci se revine la elementul anterioar și se încearcă determinarea unei alte valori care poate conduce la o soluție parțială validă.

Implementarea algoritmilor de tip backtracking urmăresc, de obicei, aceași linie. Astfel, orice algoritm de acest tip urmărește implementarea funcțiilor și peocedurilor:

- subprogramul `Succesor(k)` pentru verificarea condițiilor externe;
- subprogramul `Valid(k)` pentru verificarea condițiilor interne, dacă s-a determinat o soluție parțială validă;
- subprogramul `Solution(k)` verifică dacă soluția parțială este și o soluție finală;
- subprogramul `Print(s)` afișează soluția finală determinată;
- subprogramul `Back(n)` (numită **rutina Backtracking**- procedura unde sunt „asamblate" toate funcțiile și procedurile de mai sus;

4.2 Partea Practică

Deși se pune problema eficienței algoritmilor de tip Backtracking, există, totuși, foarte multe probleme (de exemplu, problemele NP-complete sau NP-dificile) care rezolvate prin algoritmi de tip backtracking ajung la soluții mai eficient decât prin rezolvarea de tip „forta bruta" (generarea tuturor alternativelor și selectarea soluțiilor).

O eficientizare a algoritmilor se poate realiza prin combinarea lor cu tehnici de propagare a restricțiilor.

Orice problemă care are nevoie de determinarea tuturor soluțiilor poate fi rezolvată cu backtracking.

4.2.1 Backtracking iterativ

Subprogramele folosite de această rutină sunt descrie în secținea precedentă și au puncte diferite în funcție de fiecare problemă în parte.

4.2.2 Backtracking recursiv

Varianta recursiva a algoritmului face ca acest algoritm sa fie mai lizibil, observându-se mai clar determinarea unei noi soluții.

Algoritm 18 Backtracking iterativ

```
1:  procedure BackIt(n)
2:      Init(k)
3:      while k > 0 do
4:          isS ← false
5:          isV ← false
6:          if k <= n then
7:              repeat
8:                  if Succesor(k) then
9:                      Valid(k)
10:                 end if
11:             until isS = true AND isV = true
12:         end if
13:         if isV = true then
14:             if Solutution(k) then
15:                 Print(s)
16:             else
17:                 Init(k)
18:             end if
19:         else
20:             k ← k − 1
21:         end if
22:     end while
23: end procedure
```

Algoritm 19 Backtracking recursiv

```
1:  procedure BackRec(k)
2:      if Solutution(k) then
3:          Print(s)
4:      else
5:          for j ← 1 to n do
6:              s[k] ← a[j]
7:              if Valid(k) then
8:                  BackRec(k + 1)
9:              end if
10:         end for
11:     end if
12: end procedure
```

Permutările unei mulțimi

Problema determinării permutărilor unei mulțimi de numere este problema specifică a tehnicii backtracking. Modul în care este modificată stiva soluției pentru o mulțime de 3 elemente este exemplificat în figura **??** de mai jos.

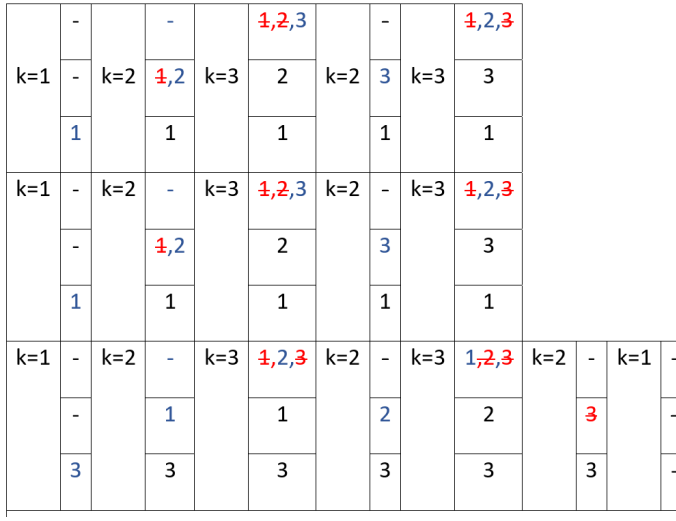


Figura 4.1: Permutările unei mulțimi de 3 elemente.

Algoritm 20 Permutările unei mulțimi

```
1: procedure PermRec(k)
2:     if k = n + 1 then
3:         WRITE s[1..n]
4:     else
5:         for j ← 1 to n do
6:             s[k] ← i
7:             if Valid(k) then
8:                 PermRec(k + 1)
9:             end if
10:        end for
11:    end if
12: end procedure
```

4.3 Tema de laborator

Folosind algoritmi de tip backtracking, rezolvați problemele de mai jos:

1. Permutări. Aranjamente. Combinări
2. *Generarea tuturor secvenţelor de n (par) paranteze care se închid corect.* Să se genereze toate șirurile de n parateze rotunde închise corect.
   Exemplu:
   Pt n=4: (()) ; ()()
   Pt n=6: ((())) ; ()(()) ; ()()() ; (()()) ; (())()

   Indicații

   - Se va nota cu 1 paranteza stânga și cu 2 paranteza din dreapta.
   - Un vectorul soluție va fi de forma: $s = (1, 1, 2, 2)$, adică (()).
   - Se va reține în variabila `ps` numărul de paranteze stângi folosite și în variabila `pd` numărul de parateze drepte folosite.
   - Se pot identifica următoarele particularităţi şi condiţii:
     - $A = \{1, 2\}$
     - vectorul soluție: $S = (s1, s2, ...sn) \in A^n$
     - $s[1] = 1$; $s[n] = 2$
     - Fiecare element $s_k \in \{1, 2\}$
     - `Valid(k)` va conține condițiile $ps <= n/2$ și $ps >= pd$
     - Se obţine soluţia dacă $k = n + 1$ și $ps = pd$
3. *Problema aranjării damelor pe o tablă de șah.* Dându-se o tablă de şah de dimensiune nxn (n>3) să se aranjeze pe ea n dame fără ca ele să se atace. O damă este atacată dacă pe linia, coloana şi cele 2 diagonale pe care se află mai există încă o damă. În figura 4.2, de mai jos, celulele colorate sunt atacate de dama poziţionată unde indică litera “D”.

   Indicații

   - Pe fiecare linie este plasată o damă.
   - Condiţia de a putea plasa o damă pe poziţia k presupune verificarea ca să nu se atace cu nici una dintre celelalte $k - 1$ dame deja plasate pe tabla.
   - Valoarea de pe poziţia k din vectorul s reprezintă coloana pe care se plasează pe tablă dama k.

- Se pot identifica următoarele particularităţi şi condiţii:
  - vectorul soluție: $s[k] \in \{1, 2, ...n\}$
  - `Valid(k)` va conține condițiile $s[i] \neq s[k]$ și $|k-1| \neq |s[k] - s[i]|, i = 2, ..., k-1$
  - Se obţine soluţia dacă $k = n + 1$

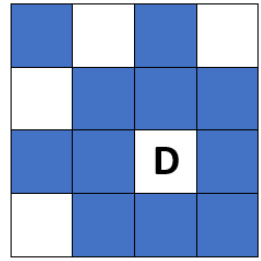


Figura 4.2: Dama atacată.

O soluție a acestei probleme este reprezentată în figura 4.3.

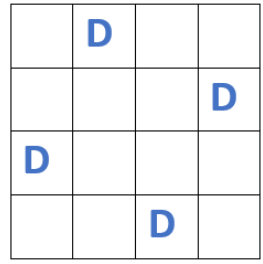


Figura 4.3: Soluție pentru problema damelor.

5 Tehnica Greedy

5.1 Partea Teoretică

5.1.1 Greedy în algoritmica grafurilor

Deoarece majoritatea algoritmilor care lucrează cu grafuri sunt descriși folosind principiile de programare Greedy, în continuare sunt prezentați cei mai reprezentativi astfel de algoritmi [7].

Drumuri minime în grafuri

Problema determinării drumurilor minime în grafuri poate fi privită în două sensuri:

- Fiind date două noduri u, v se cere să se determine cel mai scurt drum dintre ele.
- Se dă un *nod de start* s și se cere să se determine cele mai scurte drumuri dintre s și toate celelalte noduri.

Se propune spre rezolvare cea de-a doua problemă, astfel că:

Fie un graf orientat $G = (V, E)$ și un nod de start s. Se presupune că există drum între s și orice alt nod al grafului. Se cere să se determine cel mai scurt drum dintre s si toate celelalte noduri ale grafului. Se știe ca fiecare muchie e are o pondere $l_e \geq 0$ (indicând costul, timpul, distanța, etc.) dintre cele două noduri. Pentru un drum P, ponderea drumului, $L(P)$, reprezintă suma tuturor ponderilor muchilor care îl formează.

Chiar dacă problema este dată pentru un graf orientat, rezolvarea se poate adapta și grafurilor neorintate, prin înlocuirea unei muchii $e = (u, v)$ de pondere l_e cu două arce $(u, v$ și (v, u), fiecare cu aceeași pondere l_e.

Pentru graful din figura 5.1, drumul $D_1 = (1, 2, 3, 4)$ are lungimea 1+4+3=8, iar drumul $D_2 = (1, 2, 5)$ are lungimea 1+1=2.

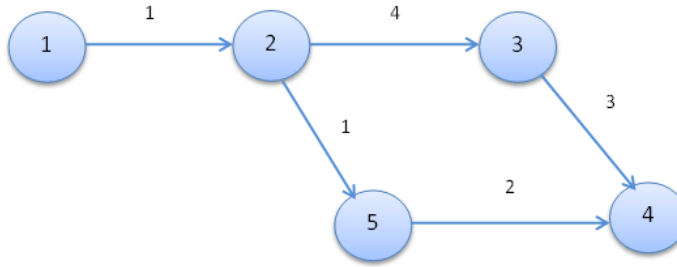


Figura 5.1: Exemplificarea drumurilor minime

Algoritmul Dijkstra

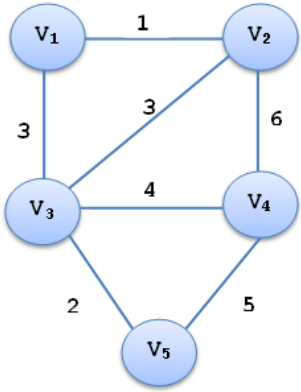
Pentru determinarea drumurilor minime dintr-un graf ponderat se va folosi un algoritm Greedy, inventat de Dijkstra (1959). Se notează cu C, mulțimea vârfurilor disponibile și cu S mulțimea nodurilor care au fost selectate. Astfel că S conţine acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, iar mulțimea C conţine toate celelalte vârfuri. La început S conține doar vârful sursă, iar la final mulţimea va conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă la el, este minimă. Spunem că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparţin lui S.

Algoritmul Dijsktra lucrează astfel:

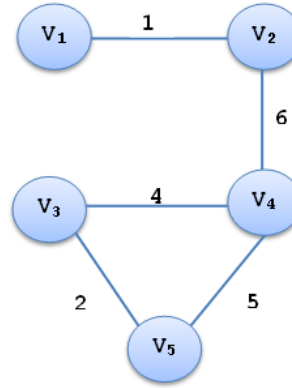
- La fiecare pas al algoritmului, un D conţine lungimea celui mai scurt drum special către fiecare vârf al grafului.
- După ce adăugăm un nou vârf v la S, cel mai scurt drum special către v ,va fi și cel mai scurt drum din toate drumurile de la sursă la v .
- Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate vârfurile sunt speciale și valorile din D reprezintă soluția problemei.

Arbori parțiali de cost minim

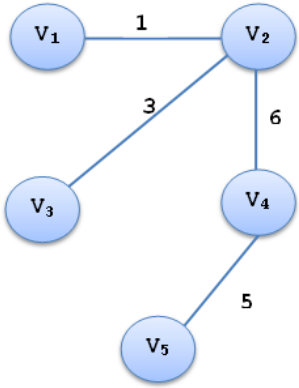
Se presupune că se dorește modernizarea unei rețele de drumuri între mai multe locații. Din cauza faptului că bugetul este destul de scăzut, se dorește ca rețeaua de drumuri să fie minimă, dar să existe o cale de acces între oricare două locații.



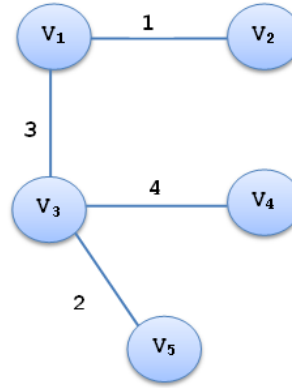
(a) Graf conex, neorientat, ponderat



(b) Dacă se elimină muchia $[v3, v4]$, graful rămâne conex



(c) Arbore parțial pentru graful dat



(d) Arbore parțial de cost minim pentru graful dat

Figura 5.2: Graf ponderat și trei subgrafuri ale sale

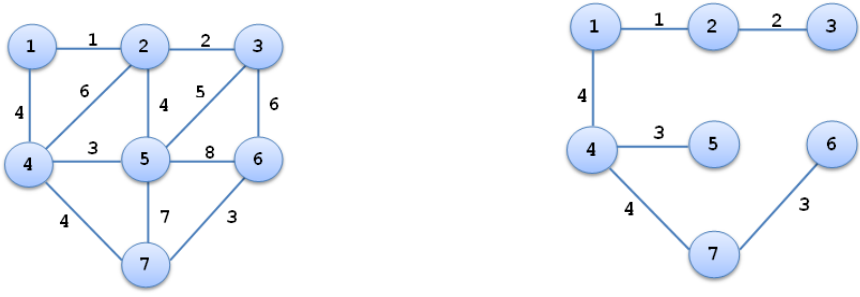
Realizând o paralela între problema dată și noțiunea de graf se ajunge la concluzia că eliminând anumite muchii dintr-un graf ponderat, conex, neorientat se formează un subgraf în care toate nodurile sunt conectate, iar suma muchiilor este minimă.

Algoritmul Kruskal

Modalitatea prin care acest algoritm determină arbolere parțial de cost minim constă în selectarea muchie cu muchie alegând mai întâi, muchia de cost minim, iar apoi se repeta procedeul având în vedere ca noua muchie aleasă să nu formeze cu precedentele un ciclu. Ideea de construire a arborelui parțial de cost minim, (V, A), este următoarea:

- se ordonează crescător muchiile grafului dat
- se completează mulțimea A, inițial vidă, cu muchii de cost minim care nu formează ciclu în arborele construit până la momentul respectiv

La fiecare pas se observă formarea unei păduri de componente conexe, determinată din pădurea anterioară și uniunea altor două componente. Fiecare componentă conexă este un arbore parțial de cost minim pentru nodurile pe care le conectează. La final se obține o singură componentă conexă ce reprezintă arborele parțial de cost minim. Astfel, pentru graful din figura 5.3a se obține arborele parțial de cost minim din figura 5.3b. Secvența de adăugare a muchiilor este prezentată în tabelul **??**.



(a) Graf neorientat conex

(b) Arbore parțial de cost minim

Figura 5.3: Graf neorientat conex și arborele parțial de cost minim corespunzător

Algoritmul Prim

Al doilea algoritm folosit pentru determinarea arborelui parțial de cost minim este algoritmul lui Prim (1957). În acest algoritm , la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U de vârfuri aferente muchiilor din A, formează arborele parțial de cost minim pentru subgraful $< u, A >$ al lui G. Inițial mulțimea U conține un singur vârf, oarecare, din V (rădăcina), iar mulțimea A este vidă. LA fiecare pas se alege muchia de

cost minim care are o extremitate în mulțimea U. Astfel, arborele parțial se formează ramură cu ramură, până când $U = V$.

5.2 Partea Practică

5.2.1 Algoritmul Dijkstra

În industrie, acest algoritm (sau variații ale acestui algoritm) este utilizat în aplicațiile de transport, unde este necesar găsirea unui drum minim între două locații, în rețelele sociale, unde poate fi folosit în sugerarea unor noi conexiuni (e.g., prieteni) și în rețelistică, unde este folosit în protocoale pentru rutare.

Presupunem că vârfurile sunt numerotate $V = 1, 2, \ldots, n$, vârful 1 fiind sursa, matricea L dă lungimea fiecărei muchii, unde $L[i, j] = +\infty$, dacă muchia $[i, j]$ nu există. Soluţia se va construi în tabloul $D[2..n]$ conform algoritmului 21:

Algoritm 21 Algoritmul Dijkstra pentru determinarea drumurilor minime

```
1:  function DIJKSTRA                                          ▷ inițializare
2:      C ← 2, 3, 4, ..., n
3:      S ← 1
4:      for i ← 2 to n do
5:          D[i] ← L[1, i]
6:      end for
7:      for i ← 3 to n do      ▷ se repeta de n-2 ori instrcțiunile următoare
8:          v ← vârful din care C minimizează D[v]
9:          C ← C \{v}
10:         S ← S ∪ {v}
11:         for fiecare w ∈ C do
12:             D[w] ← min(D[w], D[v] + L[v, w])
13:         end for
14:     end for
15:     return D
16: end function
```

5.2.2 Algoritmul Kruskal

Implementarea algoritmului Kruskal presupune folosirea subprogramelor:

Algoritm 22 Funcția *FIND*, necesară in algoritmul Kruskal, 24

```
1: function FIND (x) ▷ returnează eticheta mulțimii care îl conține pe x
2:     return set[x]
3: end function
```

Algoritm 23 Procedura *MERGE*, necesară in algoritmul Kruskal,24

```
1: procedure MERGE (a, b)        ▷ unește mulțimile etichetate cu a și b
2:     i ← a
3:     j ← b
4:     if i > j then
5:         interschimbă i cu j
6:     end if
7:     for k ← 1 to N do
8:         if set[k] = j then
9:             set[k] = i
10:        end if
11:    end for
12: end procedure
```

În descrierea algoritmului Kruskal, graful va fi reprezentat printr-o listă de muchii, fiecare având asociat un cost, astfel încât acestea vor fi ordonate dupa cost.

5.2.3 Algoritmul Prim

Pentru o altă implementare a algoritmul lui Prim (vezi 26), se consideră următoarele:

- vârfurile din V sunt numerotate de la 1 la n, $V = \{1, 2, .., n\}$
- se definește matricea costurilor C, unde $C[i, j] = +\infty$, dacă muchia $\{i, j\}$ nu există.
- se folosesc două tablouri: pentru fiecare $i \in V \setminus U$, $vecin[i]$ conține vîrful din U care este conectat la i printr-o muchie de cost minim, și $mincost[i]$ va conține acel cost. Pentru $i \in U$, $mincost[i] = -1$.
- multimea U este inițializată cu 1.
- elementele $vecin[1]$ și $mincost[1]$ nu se folosesc.

Algoritm 24 Algoritmul Kruskal pentru determinarea arborelui parțial de cost minim

```
1: function KRUSKAL (G =< V, M >)
2:                                                          ▷ inițializare
3:     sortează M crescător în funcție de cost
4:     n ← #V
5:     A ← ∅          ▷ va conține muchiile arborelui parțial de cost minim
6:     inițializează n mulțimi disjuncte, conținând fiecare cate un element
   din V
7:                                                          ▷ bucla Greedy
8:     repeat
9:         u, v ← muchia de cost minim care nu a fost analizată
10:        ucomp ← find(u)
11:        vcomp ← find(v)
12:        if ucomp ≠ vcomp then
13:            merge(ucomp, vcomp)
14:            A ← A ∪ (u, v)
15:        end if
16:    until #A = n − 1
17:    return A
18: end function
```

Algoritm 25 Descrierea generală a algoritmului lui Prim

```
1: function PRIM (G =< V, M >)
2:                                                          ▷ inițializare
3:     A ← ∅          ▷ va conține muchiile arborelui parțial de cost minim
4:     U ← un vârf oarecare din V
5:     while U ≠ V do
6:         găsește {u, v} de cost minim, astfel încât u ∈ V \ U și v ∈ U
7:         A ← A ⋃ {{u, v}}
8:         U ← U ⋃ {u}
9:     end while
10:    return A
11: end function
```

Algoritm 26 Algoritmului lui Prim

```
1: function PRIM (C[1..n, 1..n])
2:                                  ▷ inițializare, numai vârful 1 se află în U
3:     A ← ∅
4:     for i ← 2 to n do
5:         vecin[i] ← 1
6:         mincost[i] ← C[i, 1]
7:     end for
8:                                                          ▷ bucla Greedy
9:     for n − 1 times do
10:        min ← +∞
11:        for j ← 2 to n do
12:            if 0 < mincost[i] < min then
13:                min ← mincost[j]
14:                k ← j
15:            end if
16:        end for
17:        A ← A ⋃{{k, vecin[k]}}]
18:        mincost[k] ← −1
19:                                                  ▷ adaugă vârful k la U
20:        for j ← 2 to n do
21:            if C[k, j] < mincost[j] then
22:                mincost[j] ← C[k, j]
23:                vecin[j] ← k
24:            end if
25:        end for
26:    end for
27:    return A
28: end function
```

5.3 Tema de laborator

Implementați algoritmii prezentați în Java.

6 Tehnica Greedy 2

6.1 Partea Teoretică

6.1.1 Arbori binari

Arborii binari sunt o structură compusă dintr-un set te noduri în care un nod fie:

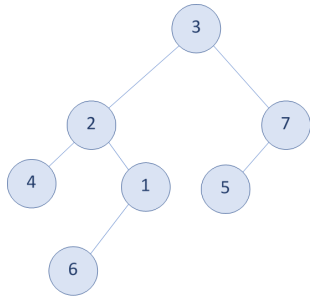
- nu conține nici un nod copil;
- este compus din trei tipuri de noduri: rădăcină, ce conține un subarbore binar stâng, și un subarbore binar drept

Arborele binar de primul tip, se numește *null* sau *gol*, și se mai notează cu NIL. Dacă subarborele stâng nu este gol, rădăcina se numește copilul stâng al rădăcinii întregului arbore. Același lucru este valabil și pentru subarborele drept. Dacă un subarbore este NIL, se spune că nu există copil pentru subarborele respectiv.

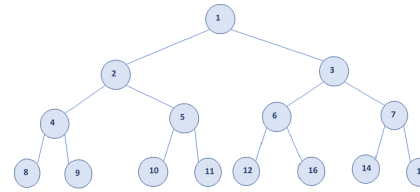
Ceea ce este foarte important într-un arbore binar este că poziția unui copil, fie stânga fie dreapta contează, și de obicei se va decide pe baza unei comparații între valorile deținute în noduri.

Se spune că un arbore binar este de adâncime k, dacă vârfurile acestea sunt dispuse pe $k+1$ niveluri, rădăcina având nivelul 0. Astfel, într-un arbore binar, numărul maxim de vârfuri de adâncime k este 2^k.

Un arbore se poate reprezenta secvențial, folosind un tablou T, punând vârfurile de adâncime k, de la stânga la dreapta în pozițiile $[\ T^{2k}] [\ T^{2k}+1] , \ldots , [\ T^{2k+1}-1]$ (eventual cu excepția nivelului 0 care poate fi incomplet), vezi figura 6.1b.



(a) Un arbore binar trasat în mod normal



(b) Numerotarea corectă a vârfurilor unui arbore

Figura 6.1: Arbori binari

6.1.2 Heap-uri

Un **heap** (în traducere aproximativă, "grămadă ordonată"), este un șir de obiecte care poate fi interpretat ca un arbore binar complet. Fiecărui nod din arbore îi corespunde un element din șir, element ce conține valoarea nodului. Arborele este complet, excepție făcând eventual, ultimul nivel, unde pot lipsi noduri terminale [2].

Proprietatea principală a elementelor este: valoare fiecărui vârf este mai mare sau egală cu cea a fiecărui fiu al său. În tabelul 6.1 este un heap care poate fi prezentat folosind un tablou unidimensional:

10	7	9	4	7	5	2	2	1	6
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

Tabela 6.1: Un Heap implementat folosind un vector

Caracteristica de bază a acestei structuri de dată este că modificarea valorii unui vârf se face foarte eficient, și se păstrează totodată proprietatea de heap. Dacă valoarea unui vârf crește, și depășește valoarea tatălui este suficient să schimbăm între ele aceste valori, până când proprietatea de heap este îndeplinită. Se spune că valoarea modificată a fost filtrată (percolated) către noua sa poziție.

Dacă valoarea vârfului scade, ai devine mai mică decât a unui fiu, este suficient să schimbăm valoarea cu cea mai mare valoare a fiilor, si apoi să continuăm procesul, până când proprietatea de heap este restabilită. Vom spune că valoarea modificată a fost cernută (sifted down) către noua sa poziție.

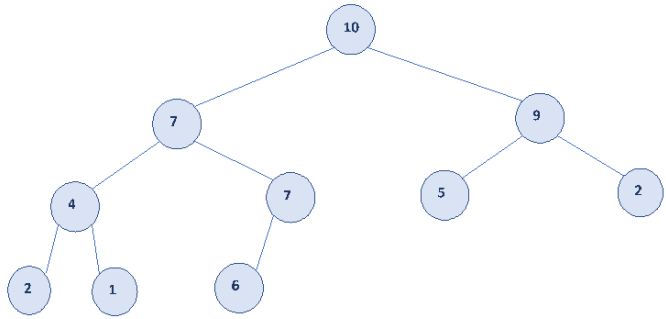


Figura 6.2: Arborele binar corespunzator heapului din 6.1

6.1.3 Interclasarea optimă a șirurilor ordonate

Să presupunem că avem două șiruri S_1, S_2 ordonate crescător și că dorim să obţinem prin interclasarea lor, șirul ordonat crescător ce conţine elementele din ambele șiruri. Dacă interclasarea are loc prin deplasarea elementelor din cele două șiruri, atunci evident numărul deplasărilor este $\#S_1 + \#S_2$.

Dacă generalizăm, considerăm n șiruri $S_1, S_2, \ldots, S_n$ unde fiecare șir $S_i \leq i \leq n$ fiind format din q_i elemente ordonate crescător (vom denumi q_i *lungimea* lui S_i). Ne propunem să obţinem șirul S,ordonat crescător, unde S conţine toate elementele din cele n șiruri. Vom realiza aceasta prin interclasări succesive de câte două șiruri. Problema constă în determinarea ordinii optime în care trebuie efectuare aceste interclasări, astfel ca numărul total de deplasări să fie minim.

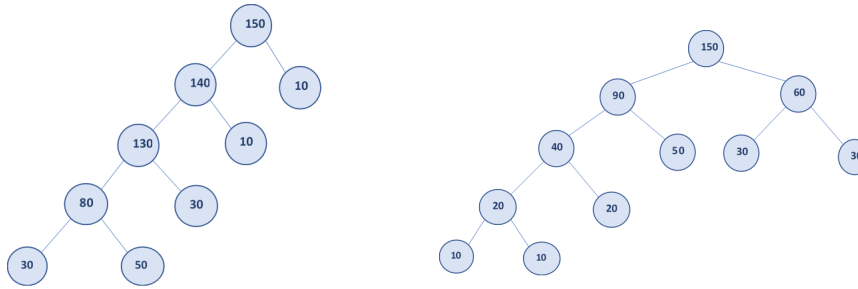
Strategia greedy constă în a interclasa mereu cele mai scurte șiruri disponibile la momentul curent. Interclasând șirurile $S_1, S_2, \ldots, S_n$ de lungimile $q_1, q_2, \ldots, q_n$ obţinem pentru fiecare strategie câte un arbore binar cu n vârfuri terminale, numerotate de la 1 la n și $n-1$ vârfuri neterminale, numerotate de la $n+1$ la $2n-1$. Definim pentru un arbore oarecare A de acest tip, *lungimea externă ponderată* ca fiind:

$$L(A) = \sum_{i=1}^{n} a_i q_i$$

unde a_i este adâncimea vârfului i. Se observă că numărul total de deplasări de elemente pentru strategia corespunzătoare lui A este chiar $L(A)$. Soluţia optimă este deci arborele pentru care lungimea ponderată este minimă.

Exemplu

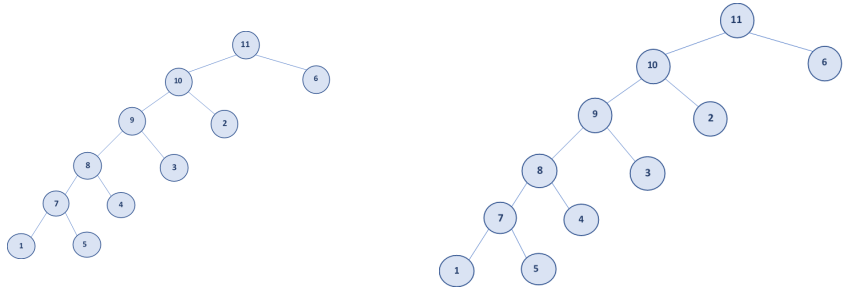
Fie șirurile $S_1, S_2, \ldots S_3$ cu lungimile $q_1 = 30, q2 = 10, q_3 = 20, q4 = 30, q_5 = 50, q_6 = 10$ două dintre strategiile de interclasare sunt reprezentaţi prin arborii din figura 6.3.



(a) (b)

Figura 6.3: Reprezentarea strategiilor de interclasare

Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 șiruri iniţiale și 5 vârfuri neterminale, adică cele 5 interclasări care definesc strategia. Vârfurile sunt numerotate astfel: vârful terminal $i, 1 \leq i \leq 6$ va corespunde șiruluiS_i, iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obţinerii lor, ca în figura 6.4.



(a) (b)

Figura 6.4: Numerotarea vârfurilor arborilor din figura 6.3

Strategia greedy apare în figura 6.3b, și constă în a interclasa mereu cele mai scurte șiruri disponibile la momentul curent.

6.2 Partea practică

6.2.1 Heapuri

Crearea heapurilor

Există două moduri de a forma un heap, pornind de la un tablou neordonat $T[\,1 \ldots n]$.

O soluție este de a porni cu un heap nou și să adăugăm rând pe rând, elementele în heap:

Algoritm 27 Algoritmul SLOW-MAKE-HEAP

```
1: function SlowMakeHeap(T[1..n])      ▷ formează în mod ineficient, din T un heap
2:     for i ← 2 to n do
3:         Percolate(T,i)
4:     end for
5: end function
```

Soluția prezentată prin algoritmul 27 nu este una eficientă. O soluție eficientă, în timp liniar este cea prezentată de algoritmul 28:

Algoritm 28 Algoritmul MAKE-HEAP

```
1: function MakeHeap(T[1..n])
2:     for i ← (n div2) downto 1 do
3:         SiftDown(T,i)
4:     end for
5: end function
```

Algoritm 29 Funcția SiftDown

```
1: function SiftDown(T[1..n], i)
2:                                          ▷ se cerne valoarea din T[i]
3:     k ← i
4:     repeat
5:         j ← k
6:                                  ▷ Găsește fiul cu valoarea cea mai mare
7:         if 2j ≤ n AND T[2j]>T[k] then
8:             k ← 2j
9:         end if
10:        if 2j < n AND T[2j+1]>T[k] then
11:            k ← 2j + 1
12:        end if
13:    until j = k
14: end function
```

Algoritm 30 Funcția Percolate

```
1: function Percolate(T[1..n], i)
2:                                          ▷ se filtrează valoarea din T[i]
3:     k ← i
4:     repeat
5:         j ← k
6:
7:         if j > 1 AND T[j DIV 2] < T[k] then
8:             k ← j DIV 2
9:         end if
10:        interschimbă T[j] și T[k]
11:    until j = k
12: end function
```

Modificarea heapurilor

Modificarea unui heap se poate face fie prin modificarea valorii unui nod, caz în care se va restabilii proprietatea de heap, fie prin inserarea unui nou nod.

Algoritm 31 Funcția AlterHeap

```
1: function AlterHeap(T[1..n], i, v)
2:                                  ▷ T[1..n] este heap, iar 1 ≤ i ≤ n este poziția
3:                          ▷ v reprezintă valoarea atribuită nodului de pe poziția i
4:                    ▷ La sfârșitul procedurii propritatea de heap este restabilită
5:
6:     x ← T[i]
7:     T[i] ← v
8:     if v < x then
9:         SiftDown(T,i)
10:    else
11:        Percolate(T,i)
12:    end if
13: end function
```

Algoritm 32 Funcția Insert

```
1: function Insert(T[1..n],v)
2:                                        ▷ inserează elementul v în Heapul T
3:
4:     T[n + 1] ← v                       ▷ Se restabilește proprietatea de Heap
5:
6:     SiftDown(T[1 . . . n-1],1)
7: end function
```

6.2.2 Interclasarea optimă a șirurilor ordonate

La scrierea algoritmului care generează arborele strategiei greedy de interclasare, vom folosi un *min-heap*. Fiecare element al acestuia este o pereche (q, i), unde i este numărul unui vârf din arborele strategiei de interclasare, iar q este lungimea șirului. Proprietatea de min-heap se referă la q. Algoritmul 33 va construi arborele strategiei greedy. Un vârf i al arborelui va fi memorat în trei locaţii diferite conţinând:

$LU[i]$=lungimea șirului reprezentat de vârf
$ST[i]$=]=lungimea șirului corespondent fiului stâng
$DR[i]$=lungimea șirului corespondent fiului drept

Algoritm 33 Algoritmul Interclasării optime

```
1: function INTEROPT(Q[1..n])
2:   ▷ construiește arborele strategiei greedy de interclasare a șirurilor de
   lungimi Q[i] = q_i, 1 ≤ i ≤ n
3:     H ← min-heap vid
4:     for i ← 1 to n do
5:         (Q[i], i) ⇒ H                          ▷ inserează în min-heap
6:         LU[i] ← Q[i]
7:         ST[i] ← 0
8:         DR[i] ← 0
9:     end for
10:    for i ← n + 1 to 2n − 1 do
11:        (s, j) ⇐ H                             ▷ extrage rădăcina lui H
12:        (r, k) ⇐ H                             ▷ extrage rădăcina lui H
13:        ST[i] ← j
14:        DR[i] ← k
15:        LU[i] ← s + r
16:    end for
17: end function
```

6.3 Tema de laborator

Implementați în limbaj Java algoritmii prezentați.

7 Divide et Impera

7.1 Partea Teoretică

Divide et Impera este o tehnică de programare care presupune ca problema să fie împărțită în mai multe subprobleme. Fiecare subproblemă va avea o rezolvare individuala, de obicei recursivă, urmând ca pentru rezultatul final să se combine soluțiile parțiale obținute[9].

7.1.1 Căutarea Binară

Algoritmul unde se poate observa repede, clar şi unde metoda *Divide et Impera* este foarte simplu implementată este algoritmul de căutare a unui element într-un şir ordonat, *Căutarea binară (BinSearch)*. Astfel că, fiind dată o valoare x, se cere să se detrmine poziţia pe care aceasta se află într-un şir S, sau să se afişeze valoarea -1, dacă nu există în şir. Se comparară x cu elementul aflat la mijlocul şirului. În cazul în care poziţia nu este determinată, în funcţie de răspunsul primit se reia căutarea fie în prima jumătate a şirului dat, fie în cea de-a doua. Astfel, se reduce numărul comparaţilor efectuate împărţind şirul în mai multe subşiruri. Algoritmul descris mai sus este 34.

7.1.2 Quick Sort

Quicksort este un algoritm de sortare al cărui ordin de timp, pentru cel mai nefavorabil caz este de $O(n^2)$, dacă se cere sortarea unui șir de n valori. În ciuda acestui fapt, acest algoritm este folosi ca *best practice* pentru sortarea unei structuri de date, deoarece este eficient în cazul mediu. Timpul său mediu este de $O(nlogn)$.De asemenea constantele din ordinul de timp sunt relativ mici.

Alt avantaj este faptul că folosește sortarea *in place*. Acest lucru înseamnă că sortarea nu folosește un alt șir auxiliar. Astfel, memoria virtuală este economisită.

Descrierea algoritmului

Să presupunem că un subșir de $r - p + 1$ elemente, $A[p..r]$ trebuie sortat. Rezolvarea propusă de acest algoritm, folosind metoda *Divide et impera* este:

- *Divide:* șirul $A[p..r]$ este partiţionat (rearanjat) în două subșirului nevide, $A[p..q]$ și $A[q + 1..r]$, astfel ca fiecare element din $A[p..q]$ să fie mai mic sau egal ca oricare element din $A[q + 1..r]$. Indexul q este calculat conform procedurii de partiţionare de mai jos.

- *Cucerește:* cele două subșiruri $A[p..q]$ și $A[q + 1..r]$ sunt sortate prin apeluri recursive ale aceleiași proceduri de quicksort.

- *Combină:* din moment ce subșirurile sunt sortate în același șir, nu mai trebuie să le combinăm, șirul este sortat $A[p..r]$ și reprezintă soluţia.

Algoritmul va forţa în repetiţii succesive obţinerea a două subșiruri cu elementele celui din stânga mai mici decât cele aparţinând celui din dreapta. Procedura PARTITION determină acest lucru astfel:

- Se selectează un element $x = A[p]$ din șirul $A[p..r]$ ca element pivot pentru a sorta acest șir.

- Se vor dezvolta două regiuni $A[p..i]$ și $A[j..r]$ de la începutul șirului, respectiv de la sfârșit, astfel că fiecare element din $A[p..i]$ va fi mai mic sau egal cu orice element din $A[j..r]$ sau cu x, reprezentantul lui $A[j..r]$ deoarece fiecare element din acest subșir este mai mare sau egal cu x.

- Iniţial $i = p - 1$ și $j = r + 1$, astfel că cele două subșiruri sunt iniţial vide.

- În cadrul corpului lui while, indexul j este decrementat și indexul lui i este incrementat, până când $A[i] \geq x \geq A[j]$.

- Presupunând că aceste inegalităţi sunt stricte, $A[i]$ este prea mare ca el să se afle în partea de jos a șirului, iar $A[j]$ este prea mic ca el să se afle în partea superioară șirului. De aceea, se efectuează interschimbarea între $A[i]$ și $A[j]$.

- Instrucţiunile lui *while* se repetă până când $i \geq j$, moment în care sigur șirul $A[p..r]$a fost partiţionat în două subșiruri $A[p..q]$ și $A[q + 1..r]$ cu $p \leq q < r$, astfel ca nici un element din $A[p..q]$ să nu fie mai mare decât oricare element din $A[q + 1..r]$

Modalitatea de funcționare a procedurii *Partion* pe un șir simplu este prezentată în figura 7.1, de mai jos.

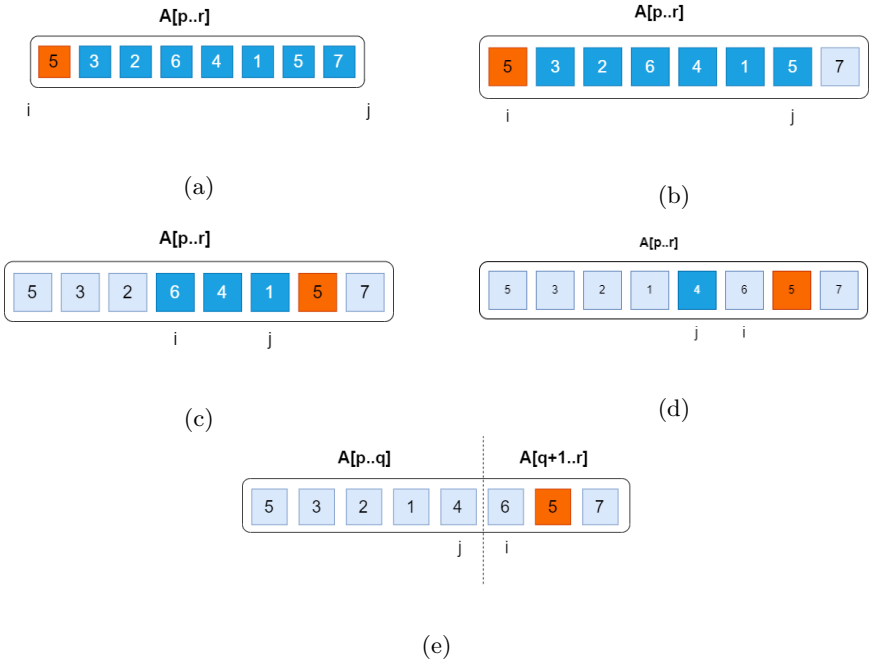


Figura 7.1: Exemplificarea procedurii *Partion*

Procedura *Partion* separă șirul în două părți: partea din stânga care va conține elementele mai mici decât pivotul și cea din dreapta care va conține elementele mai mari decât pivotul, la încheierea execuției acestei proceduri.

Descrierea modificarilor aduse de procedura *Partion* pentru șirul din figura 7.1:

- Elementele deschise la culoare au fost plasate în locaţiile corecte, iar elementele închise la culoare nu sunt încă la locul final.
- Figura 7.1a reprezintă șirul inițial, valorile lui i și j fiind chiar capetele exterioare ale șirului. Pivotul va fi $x = A[p] = 5$, semnalat prin culoarea portocalie.
- După prima iterație a buclei *While* se obțin noile poziţii ale lui i și j (figura 7.1b).
- Figura 7.1c reprezintă rezultatul interschimbării între elementele de pe poziţiile i și j și noile valori ale acestor indecși.
- După interschimbarea corespunzătoare, se reia o nouă iterație a buclei *while* se obțin valorile lui i și j din figura 7.1d.

- Următoarea iterație a buclei *while* determină rezultatele din figura 7.1e, astfel procedura se încheie deoarece $i \leq j$ și se returnează valoarea $q = j$.
- În acest moment elementele de la stânga lui j, inclusiv, sunt mai mici sau egale cu $x = 5$, iar cele de la dreapta sunt mai mari sau egale cu $x = 5$.

7.2 Partea Practică

7.2.1 Algoritmul de Căutare Binară

Algoritm 34 Algoritmul de Căutare Binară

```
1:  function BinSearch(S,x)
2:      low ← 1
3:      heigh ← length[S]
4:      location ← −1
5:      while low<heigh AND location = −1 do
6:          mid ← (low + heigh)\2
7:          if x = S[mid] then location ← mid
8:          else
9:              if x < S[mid] then
10:                 heigh ← mid − 1
11:             else
12:                 low ← mid + 1
13:             end if
14:         end if
15:     end while
16:     return location
17: end function
```

Se observă că algoritmul prezentat nu este un algoritm recursiv, așa cum majoritatea algoritmilor Divide et Impera sunt. S-a ales această variantă pentru a a reduce costul de memorie. Varianta recursivă este descrisă de algoritmul 35.

Algoritm 35 Algoritmul de Căutare Binară Recursiv

```
1: function BinSearchRec(S,x,low,heigh)
2:     if low < height then
3:         mid ← (low + heigh)\2
4:         if x = S[mid] then
5:             return mid
6:         else
7:             if x < S[mid] then
8:                 BinSearchRec(S, x, low, mid − 1)
9:             else
10:                BinSearchRec(S, x, mid + 1, heigh)
11:            end if
12:        end if
13:    else
14:        return −1
15:    end if
16: end function
```

7.2.2 Algoritmul Quick Sort

Algoritm 36 Algoritmul QUICKSORT

```
1: function QUICKSORT(A, p, r)
2:     if p < r then
3:         q ← PARTITION(A, p, r)
4:         QUICKSORT(A, p, q)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end function
```

Algoritm 37 Funcția PARTITION

```
1: function PARTITION(A, p, r)
2:     x ← A[p]
3:     i ← p − 1
4:     j ← r + 1
5:     while TRUE do
6:         repeat
7:             j ← j − 1
8:         until A[j] ≤ x
9:         repeat
10:            i ← i + 1
11:        until A[i] ≥ x
12:        if i < j then
13:            interschimbă A[i] cu A[j]
14:        else
15:            return j
16:        end if
17:    end while
18: end function
```

7.3 Tema de laborator

Implementați în limbaj Java algoritmii prezentați.

8 Acoperirea convexă

8.1 Partea Teoretică

O mulțime de puncte aflate în spațiul Eucledian sunt definite ca fiind convexă, dacă segmentul format din perechile de puncte ce aparțin mulțimii aparține la rândul său aceleiașu mulțimi.

Mai exact, fie dată mulțimea $P = p_1, p_2, p_3, \ldots, p_n$. Dacă punctele $p_1, p_2 \in P$ atunci și segmentul format din cele două puncte $[p_1, p_2] \in P$ reprezentând astfel acoperirea convexă (Convex Hull).

Această acoperire convexă este dată de mai multe aspecte:

- Cel mai mic poligon convex care conține toate punctele din P;
- Intersecția tuturor mulțimilor convexe ce conțin P;
- Intersecția tuturor semispațiilor ce conțin P;
- Reuniunea tuturor triunghiurilor determinate de puncte din P;
- Mulțimea tuturor combinațiilor convexe de puncte din P;

Algoritmii care determină acoperirea convexă a unui poligon, sunt algoritmi de geometrie computațională folosiți pentru determinarea formelor [10].

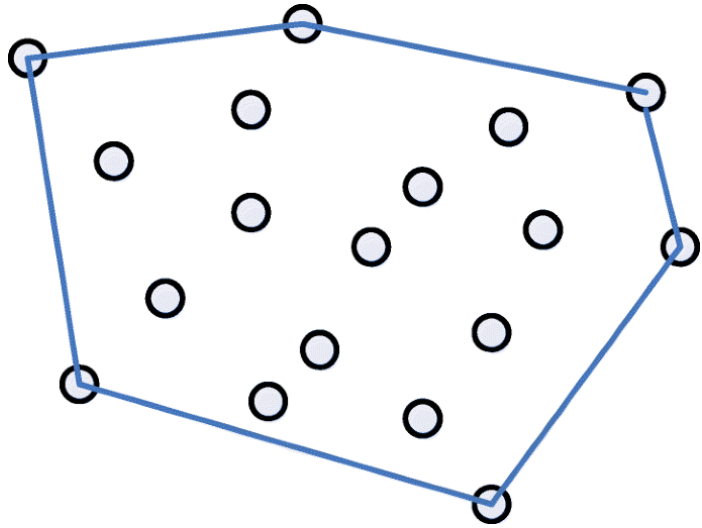


Figura 8.1: Reprezentare pentru acoperirea convexă

Acești algoritmi sunt utilizați în aplicațiile de editare de imagini, precum Photoshop, în implementarea Magic-wand tool, dar și în robotică în motion planning.

8.1.1 Algoritmul Graham Scan

Algoritmul Graham Scan este o metodă ce permite calcularea acoperirii convexe a unui set finit de puncte într-un plan.

A fost numit după Roland Graham care a publicat algoritmul pentru prima dată în anul 1972. Algoritmul determină toate laturile poligonul convex, parcurgând punctele aflate "la graniță". Se folosește o stivă pentru a putea elimina, eficient, concavitatea dintre puncte.

Complexitatea acestui algoritm este de $O(n\ log(n))$.

8.1.2 Algoritmul Quickhull

Algoritm de calculare a acoperirii convexe a unui set de puncte într-un plan. Folosește o abordare de tip Divide Et Impera similar cu QuickSort de unde îi derivă și numele. A fost publicat de către Bradford Barber și David Dopkin în anul 1995.

Complexitatea acestui algoritm este de $O(n\ log(n))$.

Ideea de bază a acestui algoritm este:

Se presupune că există o acoperire convexă (H) ce conține un set de puncte cunoscut (S) atunci există o altă acoperire convexă (H1) ce conține un set de puncte (S1) unde S + un nou punct se calculează astfel:

- P1 și P2 sunt cele mai apropiate puncte de P în stânga respectiv în dreapta acestuia
- Se exclude segmentul creat de punctele P1 și P2 din acoperirea convexă H
- Se adaugă segmentele create de punctele P1, P și P2, P la acoperirea convexă H pentru a obține H1

8.1.3 Algoritumul Gift wrapping

Creat în anul 1973 de către Ray Jarvis, algoritm de tip *incremental*, unde nu necesită o sortare în prealabil fiind foarte bun pentru mulțimile de puncte cu două dimensiuni. Având aplicabilitate la autovehicule (în detectarea obstacolelor) cât și în analiza formelor.

Complexitatea acestui algoritm este de $O(n\ m)$, unde n reprezintă numărul total de puncte, iar m numărul de puncte de frontieră. Acesta are o bună aplicabilitate atunci când avem un număr mic de puncte totale și un număr și mai mic de puncte frontieră ($m < n$).

8.2 Partea Practică

8.2.1 Algoritmul Graham Scan

Pentru a implementa acest algoritm trebuie să se urmărească următorii pași:

1. Din mulțimea de numere se alege punctul cu cea mai mică valoare pe coordonata y.
2. Se calculează unghiul format dintre cel mai de jos punct cu toate celelalte puncte.
3. Mărimea unghiul va determina în ce ordine se va itera prin puncte. Se aleg unghiurile în ordine crescătoare.

4. Iterând peste puncte, se adaugă un punct la vectorul de ieșire doar dacă există o rotație inversă acelor de ceasornic către punctul anterior. Pentru a identifica dacă rotirea este în sensul acelor sau invers, ne putem folosi de produsul a doi vectori.

5. Dacă un punct se află în sensul acelor de ceasornic relativ față de punctul anterior, atunci acel punct nu este adăugat vectorului de ieșire.

Algoritm 38 *GRAHAM_SCAN*

```
1: function GRAHAM_SCAN(P[1..n])
2:     stivas ← stiva_goala
3:     -> gaseste punctul cel mai din stanga si cu coordonata y cea mai mica, P0
4:     -> sorteaza punctele in după coordonatele polare in functie de P0, în cazul în care 2 puncte au aceleasi coordonate polare, se va retine cel mai depărtat
5:     for point in points do
6:     ▷ scoatem din stiva ultimul punct din stiva daca ne rotim in sensul acelor de ceasornic pentru a ajunge la el
7:         while lungime stiva > 1 and ccw(next_to_top(stiva), top(stiva), point) ≤ 0 do
8:             pop stiva
9:         end while
10:        push point în stiva
11:    end for
12: end function
```

Funcția ccw are urmatoarea functionalitate:

1. Va returna o valoare mai mare ca 0 daca cele 3 puncte creaza o parcurgere in sens invers al acelor de ceasornic

2. Va returna o valoare mai mica decat 0 daca cele 3 puncte creaza o parcurgere in sensul acelor de ceasornic

3. Va returna 0 daca cele 3 puncte sunt colineare (pe aceeasi dreapta).

Funcția next_to_top() este o funcție ce returneaza primul obiect de dupa primul din stiva, fara a modifica structura stivei.

Similar, functia top() returneaza prima valoare din stivă.

8.2.2 Algoritmul Quickhull

Pentru a implementa algoritmul QuickHull trebuie să se urmărească următorii pași:

1. Se găsește punctul cu coordonata x cea mai mică (A), respectiv punctul cu coordonata x cea mai mare (B)
2. Se crează o linie unind cele două puncte, unde această linie va împărți mulțimea de puncte în două părți
3. Pentru una din părți se găsește cel mai îndepărtat punct de linie (P) unde împreună cu acesta va forma un triunghi. Punctele aflate în interiorul triunghiului nu vor face parte niciodată din acoperirea convexă.
4. Pașii descriși mai sus împarte problema în două subprobleme ce se rezolvă în mmod recursiv. Acum liniile create cu P și A respectiv P și B reprezintă noile linii unde punctele aflate în afara triunghiului reprezintă noua mulțime de puncte
5. Se repetă punctul 3 până când nu mai rămân puncte exterioare.

Algoritm 39 *QuickHull*

```
1: function QuickHull
2:     convex_hull ← empty_array
3:     -> gaseste punctele cele mai din stanga si din dreapta, A și B
4:     -> adauga A și B la convex_hull
5:     -> dreapta formata din punctele A și B despart restul de puncte in
   2 grupuri, S1 și S2
           ▷ S1 reprezinta punctele care sunt in partea stanga dintre A și B
         ▷ S2 reprezinta punctele care sunt in partea dreapta dintre A și B
6:     FindHull(S1, A, B)
7:     FindHull(S, B, A)
8: end function
```

8.2.3 Algoritmul Gift wrapping

Primul pas constă în a se alege un punct unde se știe sigur că acel punct reprezintă un vârf al acoperirii convexe (cum ar fi punctul cel mai din stânga sau cel mai de jos sau cel mai de sus sau cel mai din dreapta). Pentru a continua explicația vom alege cel mai din stânga punct. Acest punct reprezintă primul element din lista de puncte de frontieră.

Apoi, lista este actualizată prin determinarea următorului punct care se află cel mai la stânga de ultimul punct calculat.

Algoritm 40 *QuickHull*

```
1: function FindHull(Sk, P, Q)
        ▷ Găsește puncte în acoperirea convexă din setul de puncte Sk
          ▷ Ce se află pe partea dreaptă a liniei formată de la P la Q
2:     if Sk nu are puncte then return
3:     end if
          ▷ Din setul de puncte Sk se alege punctul cel mai îndepărtat (numit C) de segmentul PQ
       ▷ Se adaugă punctul găsit la acoperirea convexă în locația dintre P și Q
       ▷ Cele trei puncte P, Q și C împarte mulțimea de puncte Sk în 3 diviziuni: S0, S1 și S2
          ▷ S0 reprezintă punctele care se află în interiorul triunghiului PCQ
       ▷ S1 reprezintă punctele aflate în partea dreaptă a segmentului format din punctele P și C
       ▷ S2 reprezintă punctele aflate în partea dreaptă a segmentului format din punctele C și Q
4:     FindHull(S1, P, C)
5:     FindHull(S, C, Q)
6: end function
```

Selecția se oprește atunci când se revine prin calcule la primul punct din lista de puncte de frontieră.

8.3 Tema de laborator

Să se implementeze algoritmii prezentați în limbaj Java.

Algoritm 41 *Gift wrapping*

```
1: function gftwrp(S)
                                          ▷ S reprezintă mulțimea de numere
2:     pointOnHull ← cel mai din stânga punct   ▷ punct care face sigur parte din frontieră
3:     i ← 0
4:     repeat
5:         P[i] ← pointOnHull
6:         endpoint ← S[0]      ▷ inițializează endpoint cu un punct care e posibil să facă parte din frontieră
7:         for j from 1 to |S| do
8:             if (endpoint == pointOnHull) or (S[j] se află în partea stângă de la P[i] la endpoint) then
9:                 endpoint ← S[j]
10:            end if
11:        end for
12:        i ← i + 1
13:        pointOnHull ← endpoint
14:    until endpoint ← P[0]
15: end function
```

9 Programare dinamică

9.1 Partea Teoretică

Programarea dinamică, ca și metoda divide et impera, rezolvă problemele combinând soluţiile subproblemelor. După cum am văzut, algoritmii de divide et impera, partiţionează problemele în subprobleme independente, rezolvă subproblemele în mod recursiv, iar apoi combină soluţiile lor pentru a rezolva problema iniţială [3]. Dacă subproblemele conţin subprobleme comune, în acest caz, este mai bine să folosim tehnica programării dinamice.

Dezvoltarea unui algoritm de programare dinamică poate fi descrisă de următoarea succesiune de pași:

- Se caracterizează structura unei soluţii optime
- Se definește recursiv valoarea unei soluţii optime
- Se calculează de jos în sus valoarea unei soluţii optime
- Dacă pe lângă valoarea unei soluţii optime se dorește și soluţia propriu-zisă atunci se mai efectuează și acest pas:

  Din informaţiile calculate se construiește de sus în jos o soluţie optimă.

Să analizăm pentru început ce se întâmplă cu un algoritm de divide et impera în această situaţie. Descompunerea recursivă a cazurilor în subcazuri ale aceleiași probleme, care sunt apoi rezolvate în mod independent, poate duce la calcularea de mai multe ori a aceluiași subcaz, și deci la o eficienţă scăzută a algoritmului. Să ne amintim de exemplu algoritmul lui Fibonacci expus în primul curs.

Sau să calculăm coeficientul binomial conform relației de recurență:

$$\binom{n}{n} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & pentru\ 0 < k < n \\ 1 & altfel \end{cases}$$

În mod direct aceasta se realizează astfel:

```
1: function C(n, k)
2:     if k=0 or k=n then
3:         return 1
4:     else
5:         return C(n − 1, k − 1) + C(n − 1, k)
6:     end if
7: end function
```

Se poate observa că se ajunge sa se apeleze de mai multe ori aceleasi instrucțiuni, ceea ce duce la o crestere exponentială a timpului de rulare. Solutia pentru aceasta problemă este memoizarea - reținerea rezultatului aflat la un anumit pas, pentru ca acea valoare să fie folosită la un pas ulterior. Multe din valorile C(i,j), i ≤ n, j ≤ k sunt calculate în mod repetat. Deoarece rezultatul final este obţinut prin adunarea a $\binom{n}{k}$ de 1, rezultă că timpul de execuţie pentru un apel C(n,k) este în $\Omega(\binom{n}{k})$. Dacă memorăm aceste valori într-un tablou de forma celui din figura de mai jos, obţinem un algoritm mai eficient.

	0	1	2	…	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

Figura 9.1: Coeficienţii binomiali

Acesta este triunghiul lui Pascal. Este suficient să memorăm un vector de lungime k, reprezentând linia curentă din triunghiul lui Pascal din figură. Dacă am completa o matrice de (n, k) putem spune că este vorba de elementele de sub diagonala principală. Noul algoritm necesită un timp de $O(n, k)$.

Primul principiu de bază al programării dinamice este evitarea calculării de mai multe ori a aceluiași subcaz, prin memorarea rezultatelor intermediare.

Putem spune că metoda divide et impera operează de sus în jos, descompunând un caz în subcazuri din ce în ce mai mici pe care le rezolvă separat. **Al doilea** principiu fundamental al programării dinamice este faptul că operează de jos în sus. Se pornește de obicei de la cele mai mici cazuri. Combinând soluţiile lor se obţin soluţii pentru subcazuri din ce în ce mai mari, până se ajunge în final la soluţia cazului iniţial.

Ca și în cazul algoritmilor greedy, soluţia optimă nu este în mod necesar unică. Dezvoltarea unui algoritm de programare dinamică poate fi descrisă de următoarea succesiune de pași:

- Se caracterizează structura unei soluţii optime
- Se caracterizează structura unei soluţii optime
- Se calculează de jos în sus valoarea unei soluţii optime

  Dacă pe lângă valoarea unei soluţii optime se dorește și soluţia propriuzisă atunci se mai efectuează și acest pas:

- • Din informaţiile calculate se construiește de sus în jos o soluţie optimă.

9.2 Partea Practică

9.2.1 Secventa crescatoare de lungime maxima

Prima problemă pe care o vom studia este cea mai lungă subsecvență de elemente ordonate crescător. Formularea acesteia este: dat fiind o secvență de elemente aranjate aleatoriu, găsiți acea subsecvență de lungime maximă în care elementele sunt ordonate crescător.

Aplicații ale acestei probleme sunt în diferite discipline precum în informatică (software ce afișează diferențele dintre două fișiere), teoria matricelor aleatorii, biologie (alinierea diferitelor genomuri).

Descrierea algoritmului

Avem nevoie de câteva variabile ce vor avea două tipuri de funcționalități: de a reține începutul și sfîrșitul secvenței maxime (*current_start*, *max_start* și *max_len*) și de a itera prin șirul inițial (*i current_len*). Pseudocodul este prezentat în cele ce urmează.

De exemplu pentru șirul $[2, 5, 4, 8, 10, 7, 12]$, secvența crescătoare de lungimă maximă este $[4, 8, 10]$ care începe pe poziția 3 și se termină pe poziția 5. Cum funcționează? Se va memora într-un contor *current_len* secvența

Algoritm 42

```
1: function LONGEST INCREASING SUBSEQUENCE
2:     Read A                                                    ▷ inițializare
3:     current_start, max_start, max_len, current_len ← 0
4:     for i ← 1 to n do
5:         if A[i] ≥ A[i − 1] then
6:             current_len ← current_len + 1
7:             if current_len ≥ max_len then
8:                 max_start ← current_start
9:                 max_len ← current_len − 1
10:            end if
11:        else
12:            current_start ← i
13:            current_len = 1
14:        end if
15:
16:        return max_start, max_len
```

maximă până la un punct actual și într-un contor *current_len* secvența maximă pentru tot șirul. În caz că la poziția actuală se respectă condiția de ordornare ($A[i] \geq A[i-1]$) atunci se mărește lungimea secvenței curente. Dacă nu, înseamnă ca o nouă subsecvență a început și e cazul să resetăm contoarele *current_start*.

9.2.2 Triangularea cu ponderea minimă

Problema aceasta este cea a găsirii unui mod de a împărți suprafața unui poligon convex în triunghiuri a căror perimetre însumate dau cea mai mică valoare. Adică, dat fiind un poligon convex prin punctele care il formează, să se împartă suprafața lui în triunghiuri ce conectează punctele poligonului astfe încât suma perimetrelor triunghiurilor să fie minimă. Problema este asemănătoare cu triangularea Delaunay și este folositoare în o serie de algoritmi ce trebuie să definească o suprafață 2D sau 3D (mesh) care să caracterize un nor de puncte.

Figura 9.2 prezintă un poligon cu șase laturi și câteva soluții de a triangula acest poligon.

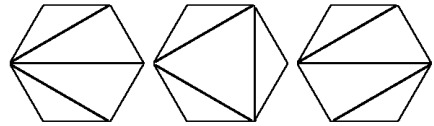


Figura 9.2: Trei moduri de a triangula un poligon cu șase laturi

Problema este de complexitate NonPolinomială însă are o soluție ce implică programarea dinamică. Să exprimăm poligonul prin ca un set de noduri și anume punctele de la intersecția a două laturi consecutive. Nodurile sunt numerotate în mod consecutiv de la primul punct al poligonului la ultimul. Pentru fiecare diagonală ce unește nodul i cu nodul j, triangularea optimă ia în considerare toate triunghiurile ikj din poligon. Aici k este orice punct din semi-poligonul ij. Dacă notăm cu $MWT(i, j)$ ponderea optimă a triangulării poligonului taiat de muchia ij, algoritmul este următorul:

Algoritm 43

```
1: function MAXIMUM TRAINGULATION
2:     Read P, n = len(P)                                   ▷ inițializare
3:     for i ← n-1 downto 1 do
4:         for j ← i+1 to n do
5:             if ij este muchie a poligonului then
6:                 MWT(i, j) ← len(ij)
7:             else
8:                 if ij nu este in jumătatea superioară a poligoului then
9:                     MWT(i, j) ← ∞
10:                else
11:                    MWT(i, j) ← len(ij) +
    min_{i<k<j} (MWT(i, k) + MWT(k, j))
12:                end if
13:            end if
14:
```

La final $MWT(1, n)$ va conține suma totală a perimetrelor triunghiurilor alese în mod optim pentru a împărți suprafața poligonului.

9.3 Tema de laborator

Implementați în Java algoritmii ce urmăresc filozofia programării dinamice: triunghiul lui Pascal, găsirea celei mai lungi subsecvențe de elemente crescătoare, problema triangulării cu pondere minimă.

10 String Matching

10.1 Partea Teoretică

Detectarea similitudinilor între secvențele de date, procedeu denumit în literatura în limba engleză *string matching*, reprezintă o tehnică cu o aplicabilitate extinsă într-o gamă largă de domenii. *String matching* este un caz particular al algoritmilor de detectare a modelelor/tiparelor. Algoritmii de detectare a similitudinilor își găsesc aplicabilitatea în procesarea limbajului natural, procesarea imaginilor și vedere artificială, transliterarea dialogurilor și alte domenii care presupun procesarea unor mari cantități de date [8]. Acești algoritmi furnizează metode care sunt utile și în cadrul altor ramuri ale informaticii (ex. proiectare software), dar și în științe conexe. De exemplu, acești algoritmi pot fi utilizați în domeniul calculelor din biologie, de exemplu pentru căutarea unor secvențe specifice în lanțurile ADN.

Algoritmii de detectare a similitudinilor sunt utilizați în viața de zi cu zi de mai mult de jumătate din populația globului (se face referire la persoanele care au acces la internet în momentul de față): folosirea lor de către motoarele de căutare pentru a găsi pagini de internet relevante pentru cuvintele cheie introduse de către utilizator.

Alte aplicații ale algoritmilor de detectarea a similitudinilor folosite pe scară largă sunt:

- Funcțiile de căutare și înlocuire (eng. *search and replace*) din editoarele de text
- Generarea automată de text pe baza cuvintelor rostite

Algoritmii de detectare a similitudinilor dintre șiruri se împart în două categorii principale:

- **algoritmii exacți** - folosiți pentru a căuta un șir conținând un număr redus de caractere (denumit în continuare “cuvânt” sau “tipar”) în cadrul unei secvențe mai lungi de caractere (denumit în continuare “text”);

Numele algoritmului	**Timp de preprocesare**	**Timp teoretic de execuție**	**Timp uzual de execuție**
Algoritmul naiv	0	O((n-m+1)×m)	O((n-m+1)×m)
Rabin-Karp	O(m)	O((n-m+1)×m)	O(n+m)
Boyer-Moore	$O(m + \sigma)$	O(m × n)	O(n/m)
Knuth-Morris-Pratt	O(m)	O(m + n)	O(m+n)
Automat de stări	$O(m \times \sigma)$	O(n)	O(n)

Tabela 10.1: Timpii de preprocesare și de execuție pentru algoritmii analizați

- **algoritmii aproximativi** - se folosesc pentru a detecta într-un text cuvinte similare cu tiparul de căutat (se caută paronime ale cuvântului de căutat). Similitudinile pot fi construite pe baza mai multor operații cu șiruri sau combinații ale acestora:
  - inserare: dorm → dor**i**m
  - ștergere: st**r**âng → stâng
  - înlocuire: far**s**ă → fa**l**să
  - inserare și înlocuire: ca**r**ul → ca**lc**ul
  - etc.

În cadrul acestui laborator se vor analiza doar algoritmii exacți de detectare a similitudinilor dintre șiruri [5].

Scopul algoritmilor exacți de detectare a similitudinilor dintre șiruri este să se găsească toate aparițiile cuvântului x de lungime m într-un text y de lungime n.

Algoritmi de detectare a similitudinilor între șiruri (DSS) pot fi comparați în funcție de timpi de execuție, enumerați în Tabelul 10.1. Timpul total de rulare a unui algoritm reprezintă suma timpilor de preprocesare și de execuție.

10.1.1 Algoritmul naiv de detectare a similitudinilor între șiruri

Caracteristici principale:

- Algoritmul nu are nevoie de o etapă de preprocesare.
- Timpul de căutare este de O((n-m+1)×m)
- Fereastra de comparare se mută cu o poziție la fiecare iterație.
- deplasarea ferestrei de comparare se poate face atât de la dreapta la stânga, cât și de la stânga la dreapta.

Acest algoritm presupune deplasări succesive ale ferestrei de comparație, cu câte o poziție, de-a lungul textului. La fiecare deplasare, se compară caracterele tiparului cu caracterele corespunzătoare din șirul de căutat.

Un exemplu de folosire a unui algoritm naiv de detectare a similitudinilor (care folosește forța brută) este vizibil în Figura 10.1. Acest algoritm presupune deplasarea unei ferestre de comparare, cu lungimea egală cu cea a tiparului, de-a lungul textului. La fiecare deplasare, fiecare caracter al tiparului este comparat cu caracterul corespunzător din cadrul ferestrei de deplasare.

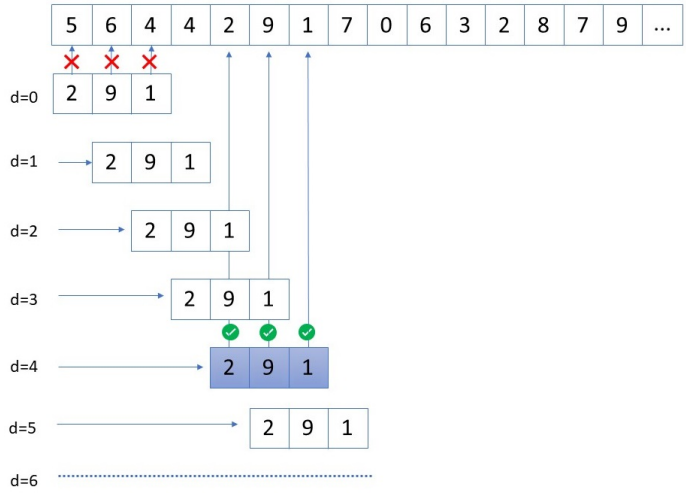


Figura 10.1: Exemplu de aplicare al algoritmului naiv de detectare a similitudinilor dintre șiruri.

10.1.2 Algoritmul Rabin-Karp

Caracteristici principale:

- Pentru căutarea aparițiilor tiparului în text, acest algoritm folosește o funcție de dispersie (numită și funcție de rezumat - eng. *hash function*).
- Timpul de preprocesare este de O(m)
- Timpul maxim de căutare este de O(n×m)
- Timpul obișnuit de căutare este de O(n + m)

Folosirea funcției de dispersie face trecerea de la compararea fiecărui element al tiparului cu elementul corespondent al textului (a se vedea Figura 10.1) la compararea rezultatului funcției de rezumat aplicată asupra tiparului cu rezultatul funcției de rezumat aplicată asupra ferestrei de text existentă în fiecare iterație. Funcțiile de dispersie trebuie să aibă un grad ridicat de discriminare, ceea ce înseamnă că funcția trebuie să returneze același rezultat pentru un număr cât mai restrâns de intrări diferite.

Algoritmul Rabin-Karp presupune compararea rezultatului aplicării funcției de dispersie (denumit în continuare codul *hash*) asupra tiparului cu codul *hash* al fiecărei secvențe de caractere demarcată prin deplasarea ferestrei de comparație. În momentul în care se întâlnește o egalitate, se compară fiecare caracter al tiparului cu caracterele corespunzătoare ale textului pentru a se valida sau a se invalida găsirea unei noi potriviri a tiparului în text.

Valoarea funcției de dispersie, atunci când fereastra de comparație se deplasează spre dreapta cu o poziție se poate calcula eficient din punct de vedere al timpului, pe baza valorii calculate la pasul anterior, folosind formula 10.1. Cu a este notat caracterul care nu mai este cuprins de fereastra de deplasare în iterația curentă a algoritmului, cu b este notat noul caracter cuprins de fereastra de deplasare, *baza* reprezintă baza în care se lucrează, iar h reprezintă rezultatul aplicării funcției de dispersie din iterația anterioară a algoritmului.

$$rehash(a, b, h, baza) = (\underbrace{\underbrace{(h - a \times baza^{m-1})}_{\text{se elimină caracterul din stânga}} \times baza + b}_{\text{se adaugă caracterul din dreapta}}) \; modulo \; q \tag{10.1}$$

În Figura 10.2 este prezentat un exemplu de aplicare a formulei 10.1. Fereastra de deplasare este compusă din 6 caractere care formează un număr. Din acest motiv s-a putut folosi funcția *modulo* ca funcție de dispersie, unde împărțitorul are valoarea 23.

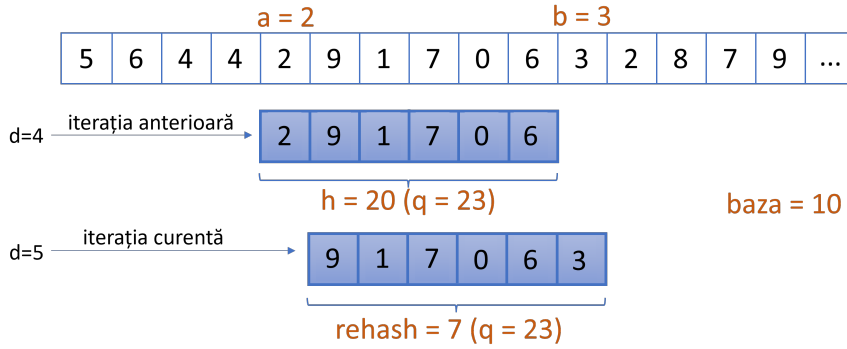


Figura 10.2: Exemplu de calcul al rezultatului funcției de dispersie pentru o nouă poziție a ferestrei de deplasare (d=5).

În Figura 10.3, este reprezentată funcționarea algoritmului Rabin-Karp atunci când funcția de dispersie este operația "modulo" iar împărțitorul are valoarea 11. Se observă că tiparul are codul *hash* 8. Doar o singură combinație de cifre a avut un cod *hash* identic. Timpul de căutare în acest caz este O(n + 2m), ceea ce înseamnă că timpul de calcul pentru căutarea tuturor aparițiilor tiparului în text este redus.

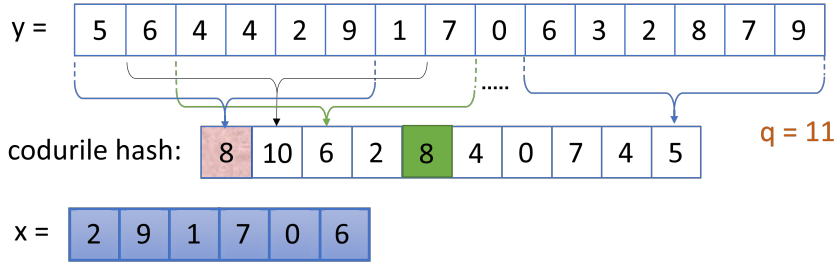


Figura 10.3: Aplicarea algoritmului Rabin-Karp pe un șir de caractere numerice, având funcția de dispersie *modulo 11*

10.2 Partea Practică

10.2.1 Algoritmul naiv de căutare a tiparului într-un text

Algoritmul 44 se desfășoară într-un timp de *O((n-m+1)× m)*. Acest timp se poate diminua în aplicațiile reale, dacă se introduce o directivă de ieșire din buclă (*break*) în blocul *dacă....atunci* de la liniile 5..7. Totuși, pentru cazul în care se caută un șir x=99...99, $|x| = m$, într-un șir y = 99...99, $|y| = n$, se atinge timpul maxim de rulare al algoritmului. Dacă $m = n/2$, timpul de rulare devine O(n^2).

Algoritm 44 Algoritmul naiv de căutare a aparițiilor tiparului într-un text

1: **function** $Naive - Searching(y, x, n, m)$
2: **for** i ← 0,n-m **do** ▷ se parcurg caracterele textului
3: *inegalitate* ← 0 ▷ se inițializează cu 0 variabila care va arăta dacă două caractere care ar fi trebuit să fie egale la o apariție a tiparului în text sunt diferite
4: **for** $j \leftarrow 0, m-1$ **do** ▷ se parcurg caracterele tiparului
5: **if** y[i]≠ x[j] **then**
6: *inegalitate* ← 1 ▷ dacă două caractere corespunzătoare nu au fost egale, se marchează faptul că tiparul nu apare pe pozițiile la care a ajuns fereastra de comparație
7: **end if**
8: **end for**
9: **if** inegalitate = 0 **then**
10: afișează S-a găsit o apariție a tiparului care începe la poziția i din text
11: **end if**
12: **end for**
13: **return** $NULL$
14: **end function**=0

10.2.2 Algoritmul Rabin-Karp

Preprocesarea care se face la începutul rulării algoritmului constă în calculul funcției de dispersie pentru tiparul a cărui apariții trebuie căutate în text (liniile 7-10 în Algoritmul 45. Această etapă are complexitatea O(m). Pentru un maxim de eficiență, se calculează în același timp și funcția de dispersie a cuvântului format din primele m caractere din text (linia 9). Referitor la liniile 8 și 9, se observă că se aplică operația modulo după fiecare adunare; acest lucru este benefic deoarece operațiile cu numere mai mici sunt efectuate mai rapid; atât aplicarea operației modulo după fiecare adunare, cât și aplicarea operației modulo doar după calculul complet al unui număr returnează același rezultat. De asemenea, tot din cadrul preprocesării face parte calculul termenului $baza^{m-1}$ (linia 4 din 45).

Bucla *for* de la liniile 11-27 este folosită parcurgerea textului în vederea căutării aparițiilor tiparului. Fereastra de deplasare este mutată cu câte o poziție la dreapta. Codul *hash* (rezultatul funcției de dispersie) al tiparului este comparat cu codul *hash* al cuvântului delimitat de fereastra de deplasare (linia 12). Dacă cele două coduri sunt egale se trece la compararea, caracter cu caracter, a tiparului cu respectivul cuvânt (liniile 13-19). Dacă cele două cuvinte sunt identice, se constată găsirea unei apariții a tiparului în text (liniile 20-21). După ce s-a terminat procesarea cuvântului de la poziția curentă a ferestrei de deplasare, se calculează codul *hash* al cuvântului obținut prin deplasarea ferestrei la dreapta cu o poziție (linia 25). Condiția de la linia 24 evită accesarea caracterului y[n] care este caracterul null.

Acest proces se repetă până când fereastra de deplasare s-a suprapus cu ultimele m caractere din text.

Algoritm 45 Algoritmul Rabin-Karp

```
1: procedure ALGORITMUL-RABIN-KARP(y, x, baza, q)
2:     m ← lungime(x)
3:     m ← lungime(y)
4:     h ← baza^(m−1) modulo q ▷ se calculează valoarea care va fi folosită la fiecare recalculare a rezultatului funcției de dispersie asupra cuvântului indicat de fereastra de deplasare
5:     nr_tipar ← 0 ▷ se inițializează numărul care va reprezenta valoarea numerică a tiparului
6:     nr_fd ← 0 ▷ se inițializează numărul care va reprezinta valoarea textelor încadrate de fereastra de deplasare la fiecare iterație a algoritmului
7:     for i ← 0, m − 1 do ▷ se parcurg caracterele tiparului și primele m caractere ale textului
8:         nr_tipar ← (10 × nr_tipar + x[i]) modulo q
9:         nr_fd ← (10 × nr_fd + y[i]) modulo q
10:    end for
11:    for d ← 0, n − m do ▷ se parcurge întregul text pentru a găsi aparițiile tiparului
12:        if nr_tipar = nr_fd then ▷ dacă rezultatele funcției de dispersie aplicate asupra tiparului și asupra ferestrei de deplasare coincid, se trece la compararea celor două cuvinte, caracter cu caracter
13:            inegalitate ← 0 ▷ se inițializează cu 0 variabila care va arăta dacă două caractere care ar fi trebuit să fie egale la o apariție a tiparului în text sunt diferite
14:            for i ← 0, m − 1 do ▷ se parcurg caracterele tiparului
15:                if y[d+i]≠ x[i] then
16:                    inegalitate ← 1 ▷ dacă două caractere corespunzătoare nu au fost egale, se marchează faptul că tiparul nu apare pe pozițiile la care a ajuns fereastra de comparație
17:                    break ▷ la prima nepotrivire găsită se iese din buclă
18:                end if
19:            end for
20:            if inegalitate = 0 then
21:                afișează "S-a găsit o apariție a tiparului care începe la poziția" d "din text"
22:            end if
23:        end if
24:        if d < n-m then
25:            nr_fd=(nr_fd - y[d]×h) × baza + y[d+m-1]) modulo q
26:        end if
27:    end for
28:    return NULL
29: end procedure
```

10.3 Tema de laborator

Implementați în Java algoritmii prezentați și folosiți exemplele date pentru testarea lor. Pentru Algoritmul Rabin-Karp, alegeți diferite valori pentru împărțitorul folosit în funcția de dispersie și observați rezultatele.

Bibliografie

[1] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. Technical Report UCB/ERL M79/54, EECS Department, University of California, Berkeley, Aug 1979.

[2] A. Arvind and C. Pandu Rangan. Symmetric min-max heap: A simpler data structure for double-ended priority queue. *Inf. Process. Lett.*, 69(4):197–199, 1999.

[3] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.

[4] E. Brockmeyer, H.L. Halstrøm, A. Jensen, and A.K. Erlang. *The Life and Works of A.K. Erlang*. Academy of Technical sciences. Copenhagen Telephone Company, 1948.

[5] Christian Charras and Thierry Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[7] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[8] Saqib Iqbal Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan Amin Gilkar, Wazir Zada Khan, and Muhammad Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE access*, 7:69614–69637, 2019.

[9] Eric A. Poser, Kathryn E. Spier, and Adrian Vermeule. Divide and conquer. *Journal of Legal Analysis*, 2:417–471, 2010.

[10] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, feb 1977.

[11] Tiberiu Socaciu, Bogdan Patrut, and Eugenia Iancu. *The Backtracking Method: Examples in Pascal and C++*. LAP Lambert Academic Publishing, Koln, DEU, 2012.