1. Evaluare expresii

Se cere să se evalueze o expresie aritmetică compusă din operatori (+,-,/,*), operanzi (numere întregi) şi paranteze folosind forma poloneză.

Se vor implementa următoarele:

- clasa template *Stiva*
- clasa template *Coada*
- clasa *Lexema* şi clasele derivate *Operator* şi *Operand*
- funcţiile:
  - *Coada Lexer(string)* – transformă şirul primit într-o coadă de lexeme
  - *Coada FormaPoloneză(Coada&)* – rescrie şirul în forma poloneză
  - *int Evaluare(Coada&)* – evaluează expresia scrisă în forma poloneză

2. Planificarea proceselor

Se consideră o listă de procese pentru care proces se cunosc: prioritatea (0 – minima, 5-maximă), durata (în număr de cuante de timp) şi denumirea. Procesele sunt memorate într-un fişier text, câte unul pe fiecare linie în formatul: `prioritate durata denumire`.

Procesele sunt planificate pentru execuţia pe procesor după următoarele reguli:

- procesele cu prioritate mai mare se execută înaintea proceselor cu o prioritate mai mică
- procesele cu aceeaşi prioritate se execută după modelul Round Robin (fiecare proces este executat pe durata unei cuante de timp după care este reintrodus la sfârşitul cozii de aşteptare)

Se cere să se afişeze pe ecran planificarea proceselor.

Se vor implementa:

- clasa template *CoadaPrioritate* care implementează o coadă în care disciplina de extragere a elementelor este dată de prioritatea acestora (compararea priorităţilor se face folosind operatorul „<”)
- clasa *Proces* care va conţine informaţiile referitoare la un proces
- Funcţia pentru citirea datelor din fişier
- Funcţia pentru planificarea proceselor
- Programul principal pentru testarea aplicaţiei

3. Implementare stive

Stivele sunt structuri de date care permit stocarea datelor şi prelucrarea acestora după ordinea ultimul venit – primul servit. Implementarea acestora se poate realiza în mai multe moduri. Pentru exemplificare se vor construi următoarele:

- clasa de bază abstractă Stiva (clasă template) care conţine declaraţiile operaţiilor de bază
- clasa derivată *StivaVector* care implementează operaţiile folosind un vector redimensionabil alocat dinamic
- clasa derivată *StivaLista* care implementează operaţiile folosind o structură dinamică de tip listă simplu înlănţuită
- funcţia *void TestareStiva(Stiva& stiva)* care simulează un număr mare de operaţii pe stivă

Se cere să se măsoare timpii necesari prelucrărilor simulate de funcţia *TestareStiva* pentru cele două implementări.

4. Vectori specializaţi

Să se implementeze clasa template *Vector* care să respecte următoarele cerinţe:

- vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
- implementează operaţiile: adăugare, ştergere elemente şi sortare vector
- operatori pentru referire elemente, comparare şi concatenare de vectori
- subclasa *Iterator* care conţine:
    - operatorul * pentru obţinerea valorii elementului curent
    - operatorul ++ pentru avansarea în vector
    - operatorul ! pentru testarea sfârşitului de vector
- metodă pentru obţinerea unui iterator la începutul vectorului

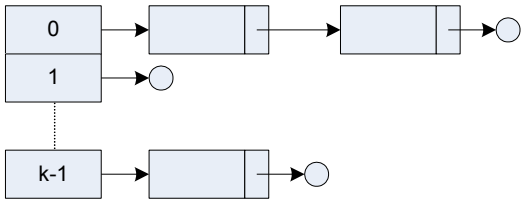
Pentru clasa *Vector* se va implementata o specializare pentru tipul *bool*. Aceasta va folosi pentru stocare un vector de întregi fără semn, iar fiecare element va fi stocat pe un bit.

Clasa se va testa pentru obiecte de tip *Persoana* şi pentru elemente de tip *bool*.

5. Tabele de dispersie

Tabelele de dispersie sunt structuri de date care permit regăsirea foarte rapidă a informaţiilor. Acestea sunt construite pe baza unei funcţii de dispersie de forma $f : M -> \{0,1,..,k-1\}$, unde M este mulţimea cheilor elementelor de stocat, iar k este numărul de intrări ale tabelei. Funcţia f asociază fiecărui element un indice în tabela de dispersie; această funcţie nu este neapărat bijectivă, deci pot exista mai multe elemente care trebuie stocate în aceeaşi intrare a tabelei.

Intern, tabelele de dispersie se memorează sub forma unui vector de liste, după modelul prezentat în figura alăturată. Fiecare element al listei conţine cheia împreună cu datele asociate.



La adăugarea unui element se calculează indicele pe baza funcţiei f şi se inserează elementul în lista corespunzătoare. Căutarea se face calculând indicele cu ajutorul funcţiei de dispersie şi parcurgând lista corespunzătoare.

Se cere să se implementeze clasa template *TabelaDispersie* cu următoarele caracteristici:

- are trei parametrii template: tipul cheii (TK), tipul stocat (TS) şi funcţia de dispersie (F)
- implementează operaţiile de adăugare, ştergere, regăsire
- supraîncarcă operatorul [] în forma *TS& operator[](const TK& cheie)* pentru operaţii de adăugare / regăsire / modificare

Clasa se va testa pe obiecte de tip *Persoana* cu chei de tip string şi funcţie de dispersie de forma: suma codurilor ASCII ale caracterelor modulo *k*.

6. Dicţionar

Se consideră un fişier text care conţine pe fiecare linie perechi de cuvinte. Se cere să se scrie un program care să traducă un fişier text folosind fişierul dicţionar.

Se vor implementa:

- clasa template *ArboreCautare* care implementează un arbore binar de căutare
- clasa *Pereche* care conţine perechile de cuvinte din dicţionar
- funcţia *ArboreCautare<Pereche> ConstruireDictionar(char* numeFisier)* care construieşte dicţionarul pe baza fişierului
- funcţia *void Traducere(const ArboreCautare<Pereche>& dictionar, char* numeFisier)* care traduce fişierul pe baza dicţionarului

7. Gestiune (Arbore)

Pentru o societate comercială se cunoaşte situaţia stocurilor la începutul lunii şi operaţiile care au avut loc în luna curentă. Datele se află în două fişiere text:

- *produse.txt* cu structura *[CodProdus Denumire Cantitate Preţ]* care conţine informaţiile de la începutul lunii
- *operatii.txt* cu structura *[CodProdus Operaţie Cantitate]*, unde *Operaţie* este 0 pentru intrare şi 1 pentru ieşire, care conţine toate operaţiile din cadrul lunii

Se cere:

- să se construiască clasa template *ArboreCautare* care implementează un arbore binar de căutare
- să se construiască clasa *Produs* care implementează:
  - operatorii relaţionali (folosind codul produsului)
  - operatorii pentru operaţiile de intrare/ieşire
  - operatorii + şi – pentru scădere/adăugare cantitate
- să se construiască arborele corespunzător situaţiei iniţiale şi să se aplice operaţiile din cadrul lunii pentru a obţine stocurile curente
- să se afişeze stocurile curente în ordinea codurilor prin parcurgerea arborelui

8. Clase de echivalenţă

O relaţie ~ pe mulţimea A se numeşte relaţie de echivalenţă dacă respectă următoarele condiţii:

- tranzitivitate: $a \sim b \ \wedge \ b \sim c \Rightarrow a \sim c$ pentru $\forall a,b,c \in A$
- simetrie: $a \sim b \Rightarrow b \sim a$ pentru $\forall a,b \in A$
- reflexivitate: $a \sim a$ pentru $\forall a \in A$

Pentru oricare element $a \in A$, clasa de echivalenţă a lui a este mulţimea elementelor din A aflate în relaţia ~ cu a.

Se cere:

- să se construiască clasa template *Mulţime* cu elementele stocate sub formă de listă care să implementeze operaţiile de bază pentru mulţimi şi un mecanism de accesare a elementelor
- să se implementeze o funcţie *ClaseEchivalenţă* care să primească ca parametri o mulţime şi un pointer la funcţie (o relaţie de echivalenţă între elementele mulţimii) şi să întoarcă clasele de echivalenţă sub forma unei mulţimi de mulţimi
- să se testeze aplicaţia pe o mulţime de persoane (două persoane sunt considerate echivalente dacă au aceeaşi vârstă)

•

9. Reţea de calculatoare

Se consideră o reţea de *n* pentru care se cunosc toate legăturile. Se cere să se determine drumurile cu număr minim de segmente între oricare două calculatoare.

Se vor implementa:

- clasa template *MatriceRara* care conţine operaţiile de bază pe matrice stocate ca liste de tripleţi de forma (linie, coloana, informaţii)
- funcţia *MatriceRara<int> DrumuriMinime(const Matrice<bool>& matAdiacenta)* care calculează drumurile minime folosind înmulţirea de matrice
- programul principal pentru citirea datelor dintr-un fişier text şi afisarea rezultatelor

10. Apa minerală

Un distribuitor de apă minerală de pe litoral doreşte să-şi determine necesarul pentru sezonul viitor pe baza datelor din trecut. Datele pentru ultimele *n* sezoane sunt disponibile într-un fişier text unde pe fiecare linie se găseşte câte o pereche de forma (Număr turişti, Cantitate consumată). Consumul pentru sezonul viitor se determină în doi paşi:

- se determină trendul şi pe baza acestuia se extrapolează numărul estimat de turişti pentru sezonul viitor
- se determină prin regresie liniară relaţia dintre numărul de turişti şi cantitatea consumată care se aplică pe numărul estimat de turişti

Se vor implementa:

- clasa template *Vector* care respectă următoarele cerinţe:
  - vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
  - implementează operaţiile: adăugare, ştergere elemente şi sortare vector
  - operatori pentru referire elemente, comparare şi concatenare de vectori
- clasa *Pereche* care conţine informaţiile despre o observaţie (variabila dependentă şi variabila independentă) şi operatori de intrare/ieşire pentru consolă şi fişiere text
- funcţia de aplicare a metodei celor mai mici pătrate cu prototipul *Pereche MCMMP(const Vector<Pereche>& observatii)*
- programul principal pentru citirea datelor din fişier text, aplicarea MCMMP pentru determinarea coeficientilor pentru trend şi regresie şi afişarea rezultatelor

11. Conferinţe (Vector)

Pentru o sală de conferinţe, intr-o anumită zi, există mai multe cereri de programare. Pentru fiecare cerere se cunosc următoarele: denumirea, ora de început, ora de sfârşit. Se cere să se construiască o planificare astfel încât numărul de conferinţe programate să fie maxim şi să nu existe suprapuneri.

Se vor implementa:

- clasa template *Vector* care respectă următoarele cerinţe:
  - vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
  - implementează operaţiile: adăugare, ştergere elemente şi sortare vector
  - operatori pentru referire elemente, comparare şi concatenare de vectori
- clasa *Conferinta* care conţine informaţiile despre conferinţe, operatori de intrare/ieşire pentru consolă şi fişiere text şi operatorul relaţional < (comparaţia se face în funcţie de ora de sfârşit)
- funcţia de planificare *Vector<Conferinta> Planificare (Vector <Conferinta>& conferinte)* care face planificarea conferinţelor
- programul principal pentru citirea datelor dintr-un fişier text şi afişarea rezultatelor

12. Centre de distribuţie

O societate comercială deţine 3 centre de aprovizionare cu produse lactate şi n de centre de desfacere (pentru centre se memorează denumirea şi oraşul). Se cunosc drumurile posibile între centre şi distanţele asociate acestora.

Se cere sa se determine pentru fiecare centru de desfacere depozitul de la care să se facă aprovizionarea şi ruta corespunzătoare astfel încât distanţa parcursă sa fie minimă.

Să se implementeze:

- clasa *Centru* care să conţină datele referitoare la un centru de desfacere sau aprovizionare
- clasa template *Graf* care să conţină datele asociate nodurilor şi distanţele dintre acestea memorate sub forma unei matrice de adiacenţă
- o funcţie *DrumMinim* care să determine ruta minimă de la un nod la celelalte noduri ale grafului

un program principal care să citească datele dintr-un fişier text şi să afişeze pe ecran rutele minime

13. Gestiune (vector)

Pentru o societate comercială se cunoaşte situaţia stocurilor la începutul lunii şi operaţiile care au avut loc în luna curentă. Datele se află în două fişiere text:

- *produse.txt* cu structura *[CodProdus Denumire Cantitate Preţ]* care conţine informaţiile de la începutul lunii
- *operatii.txt* cu structura *[CodProdus Operaţie Cantitate]*, unde *Operaţie* este 0 pentru intrare şi 1 pentru ieşire, care conţine toate operaţiile din cadrul lunii

Se cere:

- să se construiască clasa template *VectorSortat* cu următoarele caracteristici:
  - vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
  - implementează operaţiile: adăugare, ştergere element cu păstrarea ordinii elementelor
  - operatorul () pentru căutarea unui element în vector folosind căutarea binară
  - operatorul [] pentru referirea elementelor vectorului
- să se construiască clasa *Produs* care implementează:
  - operatorii relaţionali (folosind codul produsului)
  - operatorii pentru operaţiile de intrare/ieşire
  - operatorii + şi – pentru scădere/adăugare cantitate
- să se construiască vectorului corespunzător situaţiei iniţiale şi să se aplice operaţiile din cadrul lunii pentru a obţine stocurile curente
- să se afişeze stocurile curente în ordinea codurilor

14. Mulţime ca arbore

Să se construiască clasa template *Mulţime* cu următoarele caracteristici:

- elementele sunt stocate în memorie sub forma unui arbore binar de căutare
- implementează operaţiile cu elemente: adăugare, ştergere, apartenenţă
- implementează operatori operaţiile cu mulţimi: reuniune, intersecţie, diferenţă, produs cartezian, test includere, test egalitate
- conţine o clasă *Iterator* pentru parcurgerea în ordine a elementelor mulţimii

Programul principal va testa funcţionalităţile clasei pentru mulţimi de numere naturale şi de studenţi (identificaţi prin număr matricol).

15. Sisteme de ecuaţii (Matrice ca vector)

Să se scrie programul pentru rezolvarea unui sistem liniar de ecuaţii.

Se vor implementa:

- clasa template *Vector* care cu următoarele caracteristici:
  - vectorul este alocat dinamic (fără redimensionare)
  - implementează operatorul [] pentru adresarea elementelor
  - funcţie pentru obţinerea dimensiunii
- funcţia pentru rezolvarea sistemului de ecuaţii memorat ca vector de vectori: *Vector<double> RezolvareSistem(Vector< Vector<double> >& sistem)*
- programul principal pentru citirea sistemului dintr-un fişier text şi afişarea rezultatelor

16. Organigrama (Arbore binar)

Datele despre angajaţii unei companii se găsesc într-un fişier text cu linii de forma *Marca MarcaSef Nume*; directorul general are marca 0.

Se cere să se afişeze sub formă de arbore structura organizaţiei. Pentru fiecare post de conducere se va afişa suplimentar numărul de subordonaţi.

Se vor implementa:

- clasa template *Arbore* pentru lucrul cu arbori oarecare cu următoarele caracteristici:
  - memoraţi sub forma fiu-frate (fiecare nod va avea două legături: una pentru primul fiu şi una pentru următorul nod aflat la acelaşi nivel)
  - fiecare nod va avea o legătura la nodul părinte
  - implementează operaţiile de adăugare/ştergere element
  - conţine o clasă *Iterator* pentru navigarea prin arbore
- o clasă *Angajat* pentru memorarea datelor referitoare la un angajat
- o funcţie pentru construirea arborelui de angajaţi pornind de la datele din fişier
- o funcţie pentru calcularea numărului de subordonaţi
- o funcţie pentru afişarea arborelui

17. Registrul acţionarilor

O societate comercială listată la Bursa de Valori Bucureşti doreşte să distribuie dividendele aferente exerciţiului financiar încheiat. Pentru aceasta, societatea trebuie să reconstituie registrul acţionarilor la data de referinţă pe baza registrului actual şi a informaţiilor referitoare la tranzacţii furnizate de către bursă. Aceste informaţii sunt stocate în două fişiere text:

- *registru.txt*: registrul curent cu linii de forma *(NumărCont Nume NumărAcţiuni)*
- *trazactii.txt*: tranzacţiile efectuate între data curentă şi data de referinţă în forma *(ContSursă ContDestinatie NumărAcţiuni)*

Se cere să se afişeze registrul la data de referinţă sub forma unui tabel cu coloanele *NumărCont Nume NumărAcţiuni Procent*.

Se vor implementa:

- clasa template *VectorSortat* cu următoarele caracteristici:
  - vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
  - implementează operaţiile: adăugare, ştergere element cu păstrarea ordinii elementelor
  - operatorul () pentru căutarea unui element în vector folosind căutarea binară
  - operatorul [] pentru referirea elementelor vectorului
- clasa *Cont* care conţine:
  - operatorii relaţionali (folosind numărul contului)
  - operatorii pentru operaţiile de intrare/ieşire
  - operatorii += şi –= pentru scăderea/adăugarea unui număr de acţiuni
- programul principal pentru obţinerea şi afişarea registrului la data de referinţă aplicând tranzacţiile în sens invers

18. Sisteme de ecuaţii (Matrice rară)

Să se scrie programul pentru rezolvarea unui sistem liniar de ecuaţii.
Se vor implementa:

- clasa template *Matrice* care cu următoarele caracteristici:
  - elementele matricei sunt stocate sub forma unei liste de tripleţi de forma (linie, coloană, valoare)
  - implementează operatorul () pentru adresarea elementelor
  - funcţie pentru obţinerea dimensiunilor
- funcţia pentru rezolvarea sistemului de ecuaţii memorat ca vector de vectori: *Matrice<double> RezolvareSistem(Matrice <double>& sistem)*
- programul principal pentru citirea sistemului dintr-un fişier text şi afişarea rezultatelor

19. Conferinţe (Coada prioritate)

Pentru o sală de conferinţe, intr-o anumită zi, există mai multe cereri de programare. Pentru fiecare cerere se cunosc următoarele: denumirea, ora de început, ora de sfârşit. Se cere să se construiască o planificare astfel încât numărul de conferinţe programate să fie maxim şi să nu existe suprapuneri.

Se vor implementa:

- clasa template *CoadaPrioritate* care implementează o coadă în care disciplina de extragere a elementelor este dată de prioritatea acestora (compararea priorităţilor se face folosind operatorul „<”)
- clasa *Conferinta* care conţine informaţiile despre conferinţe, operatori de intrare/ieşire pentru consolă şi fişiere text şi operatorul relaţional < (comparaţia se face în funcţie de ora de sfârşit)
- funcţia de planificare *CoadaPrioritate<Conferinta> Planificare (CoadaPrioritate<Conferinta>& conferinte)* care face planificarea conferinţelor
- programul principal pentru citirea datelor dintr-un fişier text şi afişarea rezultatelor

20. Problema rucsacului (Coada prioritate)

Un distribuitor are la dispoziţie un camion care poate transporta o greutate maximă *G* cu care poate transporta bunuri aflate în depozitul său către magazine. Pentru bunurile aflate în depozit se cunosc greutatea şi profitul obţinut de pe urma transportului acestuia. Presupunând că există posibilitatea de a transporta doar o parte dintr-un bun pentru a se obţine o încărcare optimă a camionului, se cere să se determine bunurile care vor fi încărcate în camion şi profitul total.

Se vor implementa:

- clasa template *CoadaPrioritate* care implementează o coadă în care disciplina de extragere a elementelor este dată de prioritatea acestora (compararea priorităţilor se face folosind operatorul „<”)
- clasa *Produs* care conţine informaţiile despre un bun, operatori de intrare/ieşire pentru consolă şi fişiere text şi operatorul relaţional < pentru determinarea priorităţii
- funcţia de planificare *CoadaPrioritate<Produs> ListaBunuri(CoadaPrioritate<Produs>& bunuri)* care determină lista bunurilor de transportat
- programul principal pentru citirea datelor dintr-un fişier text şi afişarea rezultatelor

21. Problema rucsacului (Vector)

Un distribuitor are la dispoziţie un camion care poate transporta o greutate maximă *G* cu care poate transporta bunuri aflate în depozitul său către magazine. Pentru bunurile aflate în depozit se cunosc greutatea şi profitul obţinut de pe urma transportului acestuia. Presupunând că există posibilitatea de a transporta doar o parte dintr-un bun pentru a se obţine o încărcare optimă a camionului, se cere să se determine bunurile care vor fi încărcate în camion şi profitul total.

Se vor implementa:

- clasa template *Vector* care respectă următoarele cerinţe:
  - vectorul este stocat în memorie alocată dinamic şi are lungime variabilă
  - implementează operaţiile: adăugare, ştergere elemente şi sortare vector
  - operatori pentru referire elemente, comparare şi concatenare de vectori
- clasa *Produs* care conţine informaţiile despre un bun, operatori de intrare/ieşire pentru consolă şi fişiere text şi operatorul relaţional < pentru determinarea priorităţii
- funcţia de planificare *Vector<Produs> ListaBunuri(Vector<Produs>& bunuri)* care determină lista bunurilor de transportat
- programul principal pentru citirea datelor dintr-un fişier text şi afişarea rezultatelor

22. Organigrama (Arbore oarecare)

Datele despre angajaţii unei companii se găsesc într-un fişier text cu linii de forma *Marca MarcaSef Nume*; directorul general are marca 0.

Se cere să se afişeze sub formă de arbore structura organizaţiei. Pentru fiecare post de conducere se va afişa suplimentar numărul de subordonaţi.

Se vor implementa:

- clasa template *Arbore* pentru lucrul cu arbori oarecare cu următoarele caracteristici:
  - fiecare nod va avea un vector alocat dinamic de legături către fii şi o legătura la nodul părinte
  - implementează operaţiile de adăugare/ştergere element
  - conţine o clasă *Iterator* pentru navigarea prin arbore
- o clasă *Angajat* pentru memorarea datelor referitoare la un angajat
- o funcţie pentru construirea arborelui de angajaţi pornind de la datele din fişier
- o funcţie pentru calcularea numărului de subordonaţi

o funcţie pentru afişarea arborelui