Compte rendu — TP4 : Listes et Tableaux Triés

PAROIS Angélique, ARIMON Vincent

7 décembre 2025

Table des matières

Chapitre 1

Introduction

L'objectif de ce TP est de comparer deux structures de données — une liste chaînée triée et un tableau trié — dans le cadre de la gestion d'un répertoire de personnes. Les opérations étudiées sont :

- l'ajout ;
- l'affichage ;
- la recherche ;
- la suppression.

Chapitre 2

Structures de données

Question 1 — Structure personne

```
// Code de la structure personne
typedef struct {
    std::string nom;
    std::string prenom;
    std::string tel;
} personne;
```

Question 2 — Structure elementListe

```
// Code de la structure elementListe
typedef struct elementListe {
    personne pers;
    elementListe* suivant;
} elementListe;
```

Chapitre 3

Utilitaires

Question 3 — genererPersonne

```
// Code de genererPersonne
personne genererPersonne() {
    personne p;
    p.nom = genererNomPrenom();
    p.prenom = genererNomPrenom();
    p.tel = genererTel();
    return p;
}
```

Question 4 — creerElementListe

```
// Code de creerElementListe
elementListe* creerElementListe(personne p) {
    elementListe* e = new elementListe;
    e->pers = p;
    e->suivant = nullptr;
    return e;
}
```

Question 5 — affichagePersonne

```
// Code de affichagePersonne
void affichagePersonne(const personne& p) {
    std::cout << p.nom << "␣" << p.prenom
              << "␣(" << p.tel << ")" << std::endl;
}
```

Question 6 — egalitePersonne

```
// Code de egalitePersonne
bool egalitePersonne(const personne& a, const personne& b) {
    return (a.nom == b.nom
         && a.prenom == b.prenom
         && a.tel == b.tel);
}
```

Question 7 — comparerPersonne

```
// Code de comparerPersonne
bool comparerPersonne(const personne& a, const personne& b) {
    if (a.nom != b.nom) return a.nom < b.nom;
    if (a.prenom != b.prenom) return a.prenom < b.prenom;
    return a.tel < b.tel;
}
```

Chapitre 4

Création et utilisation d'un répertoire

Question 8 — ajouter (liste chaînée)

```
// Code de ajouterListe
elementListe* ajouterListe(elementListe* tete, personne p) {
    elementListe* e = creerElementListe(p);

    if (!tete || comparerPersonne(p, tete->pers)) {
        e->suivant = tete;
        return e;
    }

    elementListe* cur = tete;
    while (cur->suivant && comparerPersonne(cur->suivant->pers, p))
        cur = cur->suivant;

    if (egalitePersonne(cur->pers, p))
        return tete;

    e->suivant = cur->suivant;
    cur->suivant = e;
    return tete;
}
```

Question 8 — ajouter (tableau trié)

```
// Code de ajouterTableau
int ajouterTableau(personne tab[], int& n, const personne& p) {
    int i = 0;
    while (i < n && comparerPersonne(tab[i], p)) i++;

    if (i < n && egalitePersonne(tab[i], p))
        return n;

    for (int j = n; j > i; j--)
        tab[j] = tab[j-1];

    tab[i] = p;
    n++;
    return n;
}
```

Question 9 — afficher

```
// Code afficherListe
void afficherListe(elementListe* tete) {
    while (tete) {
        affichagePersonne(tete->pers);
        tete = tete->suivant;
    }
}

// Code afficherTableau
void afficherTableau(personne tab[], int n) {
    for (int i = 0; i < n; i++)
        affichagePersonne(tab[i]);
}
```

Question 10 — rechercher

```
// Code rechercherTableau
int rechercherTableau(personne tab[], int n, const personne& p) {
    int g = 0, d = n - 1;
    while (g <= d) {
        int m = (g + d)/2;
        if (egalitePersonne(tab[m], p)) return m;
        if (comparerPersonne(tab[m], p)) g = m + 1;
        else d = m - 1;
    }
    return -1;
}
```

Question 11 — supprimer

```
// Code supprimerListe
elementListe* supprimerListe(elementListe* tete, const personne& p) {
    if (!tete) return nullptr;
    if (egalitePersonne(tete->pers, p)) {
        elementListe* tmp = tete->suivant;
        delete tete;
        return tmp;
    }
    elementListe* cur = tete;
    while (cur->suivant && !egalitePersonne(cur->suivant->pers, p))
        cur = cur->suivant;

    if (cur->suivant) {
        elementListe* tmp = cur->suivant;
        cur->suivant = tmp->suivant;
        delete tmp;
    }
    return tete;
}
```

Chapitre 5

Tests et résultats

Question — Benchmark des structures

```
/* Benchmark des structures: liste cha n e tri e vs tableau tri
 * Mesure: cr ation (insertion tri e), affichage, recherche 100,
    suppression 100
 */

#include <iostream>
#include <vector>
#include <algorithm>
#include <random>
#include <ctime>
#include "utilitaires.h"

using namespace std;

int main() {
    const int N = 1000;
    const int M = 100; // nombre pour recherches & suppressions

    // G n rer un pool de N personnes uniques
    vector<personne> pool;
    pool.reserve(N);
    while ((int)pool.size() < N) {
        personne p = genererPersonne();
        bool found = false;
        for (const auto &q : pool) if (egalitePersonne(p, q)) { found =
            true; break; }
        if (!found) pool.push_back(p);
    }

    // m langer pool
    std::mt19937 rng((unsigned)time(nullptr));
    shuffle(pool.begin(), pool.end(), rng);

    // Construire liste cha n e tri e
    clock_t t1 = clock();
    elementListe* head = nullptr;
    for (const auto &p : pool) {
        head = ajouter(head, p);
    }
    clock_t t2 = clock();
    double time_list_create = double(t2 - t1) / CLOCKS_PER_SEC;
```

```
// Construire tableau tri
clock_t t3 = clock();
vector<personne> arr;
arr.reserve(N);
for (const auto &p : pool) {
    auto it = lower_bound(arr.begin(), arr.end(), p,
       comparerPersonne);
    arr.insert(it, p);
}
clock_t t4 = clock();
double time_array_create = double(t4 - t3) / CLOCKS_PER_SEC;

// Affichage (mesur )
clock_t t5 = clock();
elementListe* cur = head;
while (cur) { cur = cur->suivant; }
clock_t t6 = clock();
double time_list_display = double(t6 - t5) / CLOCKS_PER_SEC;

clock_t t7 = clock();
for (const auto &p : arr) { (void)p; }
clock_t t8 = clock();
double time_array_display = double(t8 - t7) / CLOCKS_PER_SEC;

// Pr parer M personnes    rechercher et supprimer
vector<personne> targets;
targets.reserve(M);
for (int i = 0; i < M; ++i) targets.push_back(pool[i]);

// Recherche M personnes (liste)
clock_t t9 = clock();
for (const auto &p : targets) {
    rechercher(head, p);
}
clock_t t10 = clock();
double time_list_search = double(t10 - t9) / CLOCKS_PER_SEC;

// Recherche M personnes (tableau)
clock_t t11 = clock();
for (const auto &p : targets) {
    auto it = lower_bound(arr.begin(), arr.end(), p,
       comparerPersonne);
}
clock_t t12 = clock();
double time_array_search = double(t12 - t11) / CLOCKS_PER_SEC;

// Suppression M personnes (liste)
clock_t t13 = clock();
for (const auto &p : targets) {
    elementListe* cur_del = head;
    while (cur_del) {
        if (egalitePersonne(cur_del->info, p)) {
            elementListe* next = cur_del->suivant;
            elementListe* prev = cur_del->precedent;
            if (prev) prev->suivant = next;
            if (next) next->precedent = prev;
            if (cur_del == head) head = next;
```

```
                delete cur_del;
                break;
            }
            cur_del = cur_del->suivant;
        }
    }
    clock_t t14 = clock();
    double time_list_delete = double(t14 - t13) / CLOCKS_PER_SEC;

    // Suppression M personnes (tableau)
    clock_t t15 = clock();
    for (const auto &p : targets) {
        auto it = std::find_if(arr.begin(), arr.end(),
           [&](const personne &x){ return egalitePersonne(x, p); });
        if (it != arr.end()) arr.erase(it);
    }
    clock_t t16 = clock();
    double time_array_delete = double(t16 - t15) / CLOCKS_PER_SEC;

    // Resultats
    cout << "Creation␣(liste)␣:␣" << time_list_create << "␣s\n";
    cout << "Creation␣(tableau)␣:␣" << time_array_create << "␣s\n";
    cout << "Affichage␣(liste)␣:␣" << time_list_display << "␣s\n";
    cout << "Affichage␣(tableau)␣:␣" << time_array_display << "␣s\n";
    cout << "Recherche␣100␣(liste)␣:␣" << time_list_search << "␣s\n";
    cout << "Recherche␣100␣(tableau)␣:␣" << time_array_search << "␣s\n";
    cout << "Suppression␣100␣(liste)␣:␣" << time_list_delete << "␣s\n";
    cout << "Suppression␣100␣(tableau)␣:␣" << time_array_delete << "␣s\n
       ";

    return 0;
}
```

Résultats et analyse

```
-scheisse_parois_arimon$ ./tp4_exec
Création (liste) : 0.002529 s
Création (tableau) : 0.003696 s
Affichage (liste) : 1e-06 s
Affichage (tableau) : 0 s
Recherche 100 (liste) : 0.000276 s
Recherche 100 (tableau) : 2.7e-05 s
Suppression 100 (liste) : 0.000307 s
Suppression 100 (tableau) : 0.000877 s
yshoti@yshoti-HP-Pavilion-Gaming-Laptop-15
```

FIGURE 5.1 – Résultats

Analyse des performances

Les tests réalisés portent sur la création, l'affichage, la recherche de 100 personnes et la suppression de 100 personnes dans deux structures différentes : une liste chaînée triée et un tableau trié. Les résultats obtenus sont les suivants :

Opération	**Liste chaînée**	**Tableau trié**
Création	0.002529 s	0.003696 s
Affichage	1e-06 s	0 s
Recherche 100	0.000276 s	2.7e-05 s
Suppression 100	0.000307 s	0.000877 s

1. Création du répertoire

La création du répertoire est plus rapide avec la liste chaînée. En effet, l'insertion dans une liste chaînée triée nécessite uniquement le parcours jusqu'à la position d'insertion, puis une modification de pointeurs. La complexité est de $O(n)$ par insertion.

Dans un tableau trié, chaque insertion provoque un décalage de tous les éléments situés à droite, ce qui augmente le coût mémoire. Ainsi, même si la complexité reste $O(n)$, le temps réel est plus élevé.

Conclusion : la liste chaînée est plus efficace pour la création.

2. Affichage

Les temps mesurés sont extrêmement faibles pour les deux structures. Cela s'explique par le fait que l'affichage réel n'est pas effectué : seul le parcours des structures est mesuré.

Conclusion : aucune différence pertinente ne peut être établie ici.

3. Recherche

La recherche dans la liste chaînée est linéaire ($O(n)$), ce qui conduit à un temps plus élevé : 0.000276 s pour 100 recherches.

Dans le tableau trié, la recherche utilise `lower_bound`, qui s'appuie sur une dichotomie. La complexité devient alors $O(\log n)$, ce qui explique le temps très faible : 2.7e-05 s.

Conclusion : le tableau trié est de loin le plus performant pour la recherche.

4. Suppression

La suppression dans la liste chaînée nécessite une recherche linéaire, mais la suppression elle-même ne coûte que $O(1)$ (uniquement des ajustements de pointeurs). On obtient donc un temps de 0.000307 s.

Dans le tableau, après la recherche, il faut décaler tous les éléments situés à droite, ce qui entraîne un coût $O(n)$ par suppression. Le temps total est donc plus élevé : 0.000877 s.

Conclusion : la liste chaînée est plus rapide pour la suppression.

Synthèse générale

- La **liste chaînée** est plus efficace pour la création et la suppression, car elle évite les copies et décalages en mémoire.
- Le **tableau trié** est nettement supérieur pour la recherche grâce à la recherche dichotomique.
- Aucun des deux n'est supérieur en tout point : chaque structure est adaptée à un type d'opération différent.

Les résultats expérimentaux correspondent parfaitement aux complexités théoriques.