Université Claude Bernard Lyon 1

Année universitaire : 2021 / 2022

LIFAP3 : Algorithmique et programmation avancée

Epreuve Anonyme Commune
7 janvier 2022
Durée : 1h30

Note : / 20

Nom :
Prénom :
N° d'étudiant :
Signature :

coller ici

Documents et téléphones portables interdits. Répondez dans les emplacements prévus à cet effet. Travaillez au brouillon d'abord de sorte à rendre une copie propre – nous ne pouvons pas vous garantir une copie supplémentaire. **Il se tenu compte de la présentation et de la clarté de vos réponses**.

File de priorité

Une file de priorité est un type de donnée abstrait sur laquelle on peut effectuer trois opérations :

- Insérer un élément
- Extraire et retourner l'élément ayant la plus grande priorité
- Tester si la file est vide

Elle permet par exemple d'implémenter un planificateur de tâches où un accès rapide aux tâches importantes est souhaité. On la retrouve dans les ordonnanceurs des systèmes d'exploitation, notamment le noyau Linux.

Nous allons étudier plusieurs implémentations possibles pour ce type de donnée abstrait. Une première implémentation de la file de priorité consiste à utiliser une liste simplement chaînée (comme vu en CM et TD).

Pour rappel, une liste simplement chaînée (classe **`Liste`**) a une seule donnée membre **`adPremiere`** qui est un pointeur sur une structure **`Cellule`** ayant pour champs **`info`** (la donnée) et **`suivant`** (un pointeur sur la cellule suivante). Voir l'annexe pour la liste des fonctions membres de la classe **`Liste`**.

Question 1 : Donner et justifier la complexité des fonctions **`ajouterEnTete`** et **`ajouterEnQueue`** de cette classe **`Liste`** en fonction de n, le nombre d'éléments dans la liste.

La complexité de **`ajouterEnTete`** est O(1) car la classe fournit un accès direct (via **`adPremiere`**) à la première cellule de la liste.

La complexité de **`ajouterEnQueue`** est O(n) car on doit parcourir toute la liste pour insérer en fin.

Afin de pouvoir stocker l'information de la priorité d'un élément, on décide que le champ **info** de la structure **Cellule** sera lui-même une structure. Cette structure, que l'on nommera **ElementFP**, aura deux champs : le champ **valeur** qui contiendra la valeur de l'élément (dans l'exemple de cet exercice on prendra des réels en simple précision) et un champ **priorite** qui contiendra la priorité de l'élément. La priorité est un entier positif ou nul. Une priorité de zéro représente la plus basse priorité et plus sa valeur augmente plus la priorité est grande.

Question 2 : Donner le code C++ de la structure **Cellule** mise à jour ainsi que de la nouvelle structure **ElementFP**.

```
struct Cellule {
   ElementFP info;
   Cellule * suivant;
};


struct ElementFP {
   float valeur;
   unsigned int priorite;
};
```

Question 3 : Soit la classe **FilePriorite** suivante. Donner le code C++ des fonctions membres **estVide**, **inserer** et **elementPrioritaire**.

```
class FilePriorite {

public :

   // constructeur
   FilePriorite();

   // destructeur
   ~FilePriorite();

   // Teste si la file est vide
   bool estVide() const;

   // Insère l'élément e en paramètre en début de file (tête de liste)
   void inserer(ElementFP e);
```

```
   // Retourne l'élément de plus grande priorité
   // Précondition : la file n'est pas vide
   ElementFP elementPrioritaire() const;

private :

   Liste l; // Implémentation grâce à une liste chaînée
};
```

```
bool FilePriorite::estVide() const {
   return l.estVide();
}

void FilePriorite::inserer(ElementFP e) {
   l.ajouterEnTete(e);
}

ElementFP FilePriorite::elementPrioritaire() const {
   Cellule * cmax = l.adPremiere;
   Cellule * c = cmax->suivant;
   while (c != NULL) {
      if (c->info.priorite > cmax->info.priorite) {
         cmax = c;
      }
   }
   return cmax->info;
}
```

Question 4 : Donner et justifier les complexités de ces 3 fonctions.

La fonction **estVide** appelle la fonction **estVide** de **Liste** qui vérifie simplement si **adPremiere** vaut **NULL**. Cette fonction est donc en O(1).

La fonction **inserer** appelle la fonction **ajouterEnTete** de **Liste** qui insère en tête de liste simplement chaînée (cf. question 1.1). Elle est donc aussi en O(1).

La fonction **elementPrioritaire** doit parcourir toutes les cellules de la liste de la première à la dernière. Pour chaque cellule, elle vérifie simplement la valeur de la priorité et met à jour **cmax** si besoin. Elle est en O(n).

Question 5 : Si finalement on choisit d'utiliser une liste doublement chainée (comme vu en TP), quelles seraient les complexités de ces 3 fonctions ?

Avec une liste doublement chaînée, les complexités ne changent pas. On peut toujours tester si la file est vide directement et insérer en tête de liste. Et il faut toujours parcourir toutes les cellules pour trouver celle de priorité maximale.

Question 6 : Donner un programme principal qui créé une instance de la classe **`FilePriorite`**, y insère 50 éléments dont la valeur et la priorité sont tirées aléatoirement et qui affiche à l'écran l'élément de plus grande priorité (sa valeur et sa priorité seront affichées).

```
#include "FilePriorite.h"
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main() {
   srand(time(NULL));

   FilePriorite fp;

   for (unsigned int i=0; i<50; i++) {
      float v = (rand()%101)/10.0 – 5.0;
      unsigned int p = rand()%100;
      ElementFP e; e.valeur = v; e.priorite = p;
      fp.inserer(e);
   }

   ElementFP max = fp.elementPrioritaire();
   cout << "L'élement prioritaire est : " << max.valeur << " de priorité " << max.priorite << endl;

   return 0;
}
```

Nous allons maintenant implémenter notre file de priorité avec un tableau dynamique (cf. annexe pour la liste des fonctions membres de la classe **`TableauDynamique`**). Pour cela la classe **`FilePriorite`** va avoir une donnée membre privée **`TableauDynamique t;`** à la place de la liste chaînée et les éléments du tableau seront des **`ElementFP`**.

Afin d'obtenir de meilleures performances qu'avec les listes chaînées, nous allons nous assurer que la file est triée par ordre croissant de priorité (c'est-à-dire que les éléments de grande priorité sont en fin de tableau et les éléments de faible priorité au début).

Nous avons deux stratégies possibles : soit trier les éléments uniquement quand on en a besoin, soit s'assurer qu'à chaque manipulation (insertion et extraction) les éléments restent triés.

Pour implémenter la première stratégie, la classe doit fournir une fonction de tri des éléments par ordre croissant de priorité.

Question 7 : Donner le code C++ de la fonction membre **`triSelection`** de la classe **`FilePriorite`** qui trie les éléments par ordre croissant de priorité. Cette fonction utilisera le principe du tri par sélection (sélection, à chaque itération, de l'élément de priorité minimale dans le reste du tableau non trié).

```
void FilePriorite::triSelection () {
    for (unsigned int i=0; i<t.taille_utilisee-1; i++) {
        unsigned int indmin = i;
        for (unsigned int j=i+1; j<t.taille_utilisee; j++){
            if (t.valeurIemeElement(j).priorite < t.valeurIemeElement(indmin).priorite)
               indmin = j;
        }
        ElementPF minElement = t.valeurIemeElement(indmin);
        t.modifierIemeElement(indmin, t.valeurIemeElement(i));
        t.modifierIemeElement(i, minElement);
    }
}
```

Question 8 : Donner et justifier brièvement la complexité de cette fonction **`triSelection`** en fonction de n, le nombre d'éléments dans le tableau.

La fonction **`triSelection`** a une complexité en $O(n^2)$. En effet pour chaque indice du tableau, on cherche l'élément de priorité minimale dans le reste du tableau (de taille proportionnelle à n) et on l'échange avec l'élément courant.

Avec la deuxième stratégie, les fonctions d'insertion et d'extraction doivent assurer que l'ordre des éléments est conservé à chaque appel à ces fonctions.

Question 9 : Donner le code C++ d'une fonction membre **`extraire`** respectant cette contrainte. Cette fonction retournera et supprimera de la file l'élément de plus grande priorité et devra être de complexité amortie constante O(1).

```
ElementFP FilePriorite::extraire() {
   // L'élément de plus grande priorité est à la fin du tableau (indice taille_utilise-1)
   ElementFP e = t.valeurIemeElement(t.taille_utilisee-1);
   t.supprimerElement(t.taille_utilisee-1);
   return e;
}
```

Question 10 : Expliquer en quelques phrases comment vous coderiez la fonction d'insertion de sorte qu'elle soit de complexité minimale.

La fonction d'insertion peut s'écrire de deux façons (une seule est demandée).

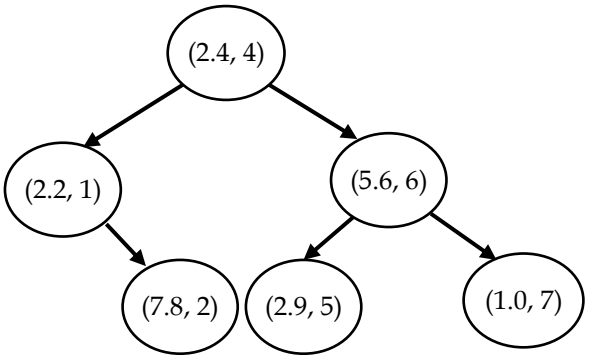
1) On recherche par dichotomie l'indice où l'élément passé en paramètre doit s'insérer (complexité O(log n)). Puis on insère, dans le pire des cas en début de tableau donc nécessite la recopie de tous les éléments (complexité O(n)). Donc une complexité totale de O(n+log n)=O(n).
2) On parcourt le tableau depuis le début à la recherche de l'indice (complexité en O(n) dans le pire des cas). Puis on insère l'élément et on décale le reste du tableau. Le nombre d'éléments manipulés au total (soit utilisés dans la recherche soit dans l'insertion) est le nombre total d'éléments, d'où une complexité en O(n).

Un autre moyen d'organiser les éléments afin de les conserver triés par priorité est de les stocker dans un arbre binaire de recherche (ABR, comme vu en TP). Pour rappel, dans un ABR tous les éléments dans le sous-arbre gauche d'un nœud sont plus petits que ce nœud et tous les éléments dans le sous-arbre droit sont plus grands (cf. annexe pour les fonctions membres de la classe **`Arbre`**).

Nous allons donc maintenant implémenter notre file de priorité avec un ABR et pour cela la classe **`FilePriorite`** va avoir une donnée membre privée **`Arbre a;`** à la place du tableau dynamique. Et nous allons nous servir de la valeur de priorité pour ordonner les nœuds.

Un **`Noeud`** sera défini comme une structure contenant trois champs : **`info`** de type **`ElementFP`**, **`fg`** et **`fd`** tous les deux de type pointeur sur **`Noeud`**. La seule donnée membre de la classe **`Arbre`** est **`adRacine`**, un pointeur sur le **`Noeud`** racine de l'arbre.

Question 11 : Dessiner l'arbre obtenu après l'insertion des éléments (c'est-à-dire les paires (valeur, priorité)) suivants : (2.4, 4), (5.6, 6), (2.2, 1), (1.0, 7), (7.8, 2), (2.9, 5). Vous écrirez les paires (valeur, priorité) dans les nœuds.



Question 12 : Où se trouve toujours l'élément de plus grande priorité ?

L'élément de plus grande priorité se situe dans le nœud le plus à droite de l'arbre. Depuis la racine on y accède donc en suivant les liens **`fd`** jusqu'à ce que son fils droit soit égal à **`NULL`**.

Question 13 : Expliquer en quelques phrases comment vous coderiez la fonction d'insertion. Quelle est la complexité de cette fonction dans le pire et le meilleur des cas ?

La fonction d'insertion parcourt la branche de l'arbre où l'insertion doit se faire. On va à gauche ou à droite en fonction de la différence entre la priorité de l'élément à insérer et la priorité de l'élément courant. Dès que l'on trouve NULL on insère. La complexité est donc en O(n) dans le pire des cas (arbre dégénéré) et O(log n) dans le meilleur des cas (arbre équilibré).

Question 14 : Quel parcours permet de parcourir les éléments dans l'ordre croissant des priorités ?

Le parcours infixe (ou en ordre) donne les éléments dans l'ordre croissant des priorités.

Annexe : liste des fonctions membres des classes TableauDynamique, Liste, File, Pile et Arbre (constructeurs et destructeurs omis)

Classe TableauDynamique

```
void vider ();
void ajouterElement (ElementTD e);
ElementTD valeurIemeElement (unsigned int indice) const;
void modifierValeurIemeElement (ElementTD e, unsigned int indice);
void afficher () const;
void supprimerElement (unsigned int indice);
void insererElement (ElementTD e, unsigned int indice);
int rechercherElement (ElementTD e) const;
```

Classe Liste

```
void vider ();
bool estVide () const;
ElementL iemeElement (unsigned int indice) const;
void modifierIemeElement (unsigned int indice, ElementL e);
void afficherGaucheDroite () const;
void afficherDroiteGauche () const;
void ajouterEnTete (ElementL e);
void ajouterEnQueue (ElementL e);
void supprimerTete ();
int rechercherElement (ElementL e) const;
void insererElement (ElementL e, unsigned int indice);
void supprimerElement (ElementL e);
```

Classe File

```
void enfiler (ElementF e);
ElementF premierDeLaFile () const;
void defiler ();
bool estVide () const;
unsigned int nbElements () const;
```

Classe Pile

```
void empiler (ElementP e);
ElementP consulterSommet () const;
void depiler ();
bool estVide () const;
```

Classe Arbre

```
bool estVide () const;
void vider ();
void insererElement (ElementA e);
void supprimerElement (ElementA e);
Noeud * rechercherElement (ElementA e) const;
```