UNIVERSITÉ DE NANTES

Rapport de projet

Algorithmique et structure de donnés 2

Eslam HUMAID et Abrahim Fekri BAMATRAF

Groupe 485

2019-2020

Introduction

Ce travail a été fait dans la cadre de projet de l'UE Algorithmique et structures de données 2, ce projet a pour but la gestion d'un system de dépôt de bagage en créant deux classes (consigne et ticket) et en bien identifiant l'SDA et l'SDC de chaque classe.

Partie 1 :

La classe Ticket :

SDA :

Méthode	Rôle	Précondition
creerTicket(T) Ticket → ∅	Créer un ticket et générer son code aléatoire.	-
detruireTicket(T) Ticket → ∅	Détruire l'objet Ticket	-
getCodeTicket(T) Ticket → chaine	Accesseur pour le code du ticket.	-
Operator==(T,T) Ticket, Ticket → Boolean	Tester l'égalité entre deux tickets.	-
Operator!=(T,T) Ticket, Ticket → Boolean	Tester l'inégalité entre deux tickets.	-

SDC :

Pour effectuer ces opérations efficacement il suffit de stocker le code du ticket afin de l'accéder simplement pour la méthode getCodeTicket(T).

La création et destruction du ticket se feront simplement en temps constante sans la nécessité d'autre attributs.

Pour tester l'égalité et l'inégalité entre deux tickets, on compare leurs codes car chaque code est unique.

Alors l'SDC se résume comme suit :

Type Ticket = enregistrement

- chaine *_codeTicket* (le code unique du ticket)

Opération	Complexité
CreerTicket(T)	Θ(1)
detruireTicket(T)	Θ(1)
getCodeTicket(T)	Θ(1)
Operator==(T,T)	Θ(1)
Operator!=(T,T)	Θ(1)

La classe Storage :

SDA :

Méthode	Rôle	Précondition
creerStorage(S,n) Storage x entier → ∅	Créer une consigne.	n > 0
detruireStorage(S) Storage → ∅	Détruire l'objet Storage	-
isFull(S) Storage → boolean	Tester si la consigne est pleine.	-
deposit(S,B) Storage x Bagage → Ticket	Déposer un bagage dans un casier libre dont la dernière utilisation est la plus ancienne et retourner un ticket.	La consigne n'est pas pleine. Le ticket corresponde bien à une case dans la consigne.
Collect(S,T) Storage x Ticket → Bagage	Récupérer le bagage dans S avec le ticket T.	La consigne n'est pas vide.

SDC :

Pour effectuer les opérations de récupération et dépôt en temps constant, on aura besoin de deux propriétés importantes :

1- Accès rapide à la case dont la dernière utilisation est la plus ancienne.
2- Accès rapide à une case contenant un bagage déposé par son ticket.

Pour la première, une stratégie sera d'utiliser une file et y mettre les cases vidées lors des dépôts. Ainsi, la tête de la file sera toujours la case dont la dernière utilisation est la plus ancienne.

Pour la deuxième, une table associative sera la plus appropriée pour ce cas dû au fait que l'accès par une clé dans la table associative se fera en temps constante.

On peut aussi ajouter un vecteur pour avoir une représentation plus concrète de la consigne en mémoire. Les cases de ce vecteur contiendront les tickets des bagages déposées dedans s'ils ne sont pas vides.

Pour lier les cases dans le vecteur qui représente la consigne et la table associative il faut indiquer pour chaque bagage stocké dans la table associative, la case dans le vecteur où ce bagage est déposé. Pour cela on crée une structure de données appelée t_case, et on stocke le bagage et l'indice de la case dans le vecteur où ce bagage est déposé.

Type t_case = enregistrement (une structure qui représente une case dans la consigne)

- Entier IndexInVect (l'indice de la case dans le Vector)
- Bagage bag (le bagage stocké dans la case)

Du coupe la table associative associe un ticket avec un t_case comme suit :

Table<Ticket,t_case> _storage

Pour implémenter efficacement la méthode isFull(), il va falloir stocker le nombre totale des cases et le nombre des cases remplis dans la consigne afin de rapidement déterminer si la consigne est plein par comparer les deux.

En fin, pour créer la consigne en temps constantes sans avoir à initialiser toutes les cases dans la queue (qui se fera en temps linéaire) on met les indices des cases vides dans la file uniquement après le premier usage en utilisant l'attribut _usingCase qui s'incrément à chaque dépôt dans la consigne (si _usingCase est inférieur au nombre des cases, on met le bagage dans la case suivant non-utilisée dans la consigne et sinon c'est soit que la consigne est pleine soit qu'on a déjà vidé une case donc on prend la première case dans la file).

Du coup, l'SDC se résume comme suit :

Type Storage = enregistrement

- Vector<Ticket> _cases (un vecteur représentant les cases de la consigne)
- Table<Ticket,t_case> _storage (table associative Ticket → t_case)
- File<entier> _emptyCases (File des cases vide après premier usage)
- Entier _nbCases (nombre des cases dans la consigne)
- Entier _filledCases (le nombre des cases actuellement pleine)
- Entier _usingCase (le nombre total des dépôts dans la consigne)

Opération	Complexité
creerStorage(S,n)	Θ(1)
detruireStorage(S)	Θ(1)
isFull(S)	Θ(1)
deposit(S,B)	Θ(1)
Collect(S,T)	Θ(1)

Partie 2 :

La classe VStorage :

SDA :

Méthode	Rôle	Précondition
CreerVStorage (S, L) Storage × liste → Ø	Créer une consigne avec des volumes donnés dans la liste de couples (ni,vi).	|liste| > 0.
CreerVStorage (S, L, L) Storage × liste1 × liste2 → Ø.	Créer une consigne avec des volumes donné dans la liste1 et la nombre de case correspondant au volume dans la liste2.	|liste1| = |liste2|.
DetruireVStorage(S) Storage → Ø.	Détruire l'objet Storage	-
isFull(S) Storage → booléen.	Tester si la consigne est pleine.	-
deposit(S,B) Storage × Bagage → Ticket.	Déposer un bagage dans un casier libre avec un volume plus grand ou égal au volume du bagage et retourner un ticket.	Il existe un casier avec un volume plus grand ou égal au volume du bagage.
collect(S,T) Storage × Ticket → Bagage.	Récupérer le bagage dans S avec le ticket T.	La consigne n'est pas vide et le ticket corresponde bien à une case dans la consigne.
haveSpace(S, X) Storage × réel → booléen.	Tester s'il y a un casier avec un volume plus grand ou égal à X.	-
getVolumes(S) Storage → liste.	Retourner la liste des volumes de la Storage.	-
getEmptyCases(S) Storage → liste.	Retourner la liste des cases libres.	-

SDC :

Dans cette classe on s'intéresse à une stratégie de dépôt optimale et pour cela il faut prendre en compte trois propriétés importantes :

1- Facilement récupérer un bagage déposé par son ticket.
2- Les cases libres doivent être ordonnées de manière à efficacement trouver la case dont la dernière utilisation est la plus ancienne.
3- Efficacement trouver la case de volume minimal et supérieur à celle de bagage déposé.

Comme la partie précédente, on peut utiliser une table associative pour la récupération rapide des bagages.

Un vecteur sera aussi utilisé ici pour la représentation plus concrète de la consigne mais au lieu de stocker les tickets, il vaut mieux stocker la volume de chaque casier pour y accéder plus simplement.

Identiquement à la partie précédente, on associe les cases du vecteur et les cases dans la table associative par une structure de données (t_case) qui stocke le bagage et l'indice de la case dans le vecteur.

Type t_casev = enregistrement (une structure qui représente une case dans la consigne)

- Entier IndexInCasesVolumes (l'indice de la case dans le vector)
- Bagage bag (le bagage stocké dans la case)

Du coupe la table associative associe un ticket avec un t_casev comme suit :

Table<Ticket,t_casev> _storage

Pour la deuxième propriété, c'est plus possible d'utiliser une file car il va falloir parcourir les cases libres lors de dépôt pour trouver la case la plus adaptée au bagage. Une stratégie sera d'utiliser un vecteur et y mettre la case libérée à la fin de vecteur après chaque récupération de bagage, comme ça les cases du vecteur seront toujours ordonnées par temps de libération et du coup le premier élément du vecteur sera la case dont la dernière utilisation est la plus ancienne.

Pour trouver la case la plus convenable (de volume minimal et supérieur à celle de bagage) il va falloir parcourir les cases libres et trouver la case avec le volume le plus bas possible pour qu'il sera supérieur à celle du bagage. Cette opération peut se faire avec une boucle for sans la nécessité d'autre attributs.

La méthode isFull(S) est identique à celle de la partie précédente donc on compare les attributs nbCases et filledCases pour tester si la consigne est pleine ou pas.

La création d'un vconsigne peut se faire de deux manières :

1- Soit par passer une liste de couple en argument (ni,vi) où vi est un volume spécifique et ni est le nombre de case à créer avec ce volume.
2- Soit par passer deux listes, une liste pour les ni et une liste pour les vi.

En conclusion, l'SDC se résume comme suit :

Type VStorage = enregistrement

- Table<Ticket,t_casev> _storage (table associative Ticket → t_casev)
- Entier _nbCases (le nombre des cases dans la consigne)
- Entier _filledCases (le nombre des cases actuellement pleine)
- Vector<entier> _emptyCases (le vecteur des cases vide)
- Vector<réel> _casesVolumes (le vecteur représentant les cases de la consigne)

Opération	Complexité
CreerVStorage (S, L)	Θ(n²)
CreerVStorage (S, L, L)	Θ(n²)
DetruireVStorage(S)	Θ(1)
isFull(S)	Θ(1)
deposit(S,B)	Θ(n)
collect(S,T)	Θ(1)
haveSpace(S, X)	Ω(1), O(n)
getVolumes(S)	Θ(1)
getEmptyCases(S)	Θ(1)

La classe Bagage :

Cette classe abstraite est ajoutée afin de faciliter l'extensibilité du programme. On peut facilement ajouter des différents types de bagage sans avoir à modifier le code déjà existant.

SDA :

Chaque type de bagage possède un volume calculé de manière diffèrent de l'autre. L'identifiant de ces bagages peut lui aussi être diffèrent dans certains cas (par exemple : l'Id d'une valise peut contenir quelque chose qui la distingue des sacs à dos). Pour cette raison les méthodes ci-dessous sont abstraite pur et il faut les implémenter dans les sous classes.

Méthode	Rôle	Précondition
getID(B) Bagage → string.	Retourner l'ID du bagage.	-
grtVolume(B) Bagage → réel.	Retourner le volume du bagage.	-

SDC :

Tous les bagages sont identifiés par leurs id et leur volume.

Type Bagage = enregistrement

- Chain _ID (Le ID de la bagage)
- Réel _volume (Le volume de la bagage)

La classe Backbag :

SDA :

Une sous-classe de la classe Bagage donc il faut implémenter les méthodes (getID(B) , getVolume(B))

Méthode	Rôle	Précondition
créerBackbag(B,S,X) Backbag ×string × volume → Ø.	Cree un bagage du type Backbag avec un string et un volume.	-
DetruireBackbag(B) Backbag → Ø.	Détruire l'objet Backbag.	-
getID(B) Backbag → string.	Retourner la ID du bagage.	-
getVolume(B) Backbag → réel.	Retourner la volume du bagage.	-

SDC :

Cette classe hérite les attributs (Chain _ID , Réel _volume) de la classe Bagage.

Toutes les opérations peuvent se faire en temps constantes donc il n'y a pas besoin d'autre attributs.

Opération	Complexité
créerBackbag(B,S,X)	Θ(1)
DetruireBackbag(B)	Θ(1)
getID(B)	Θ(1)
getVolume(B)	Θ(1)

Conclusion :

Le travail sur ce projet nous a permet d'améliorer nos capacités de créer un system complet et à chercher les meilleurs moyens pour rendre ce system plus efficace et plus performant en pratiquant la programmation orientée objet en C++.