PARIS DESCARTES IUT

DUT Informatique - 1ère année
SDA - Structures de Données et Algorithmes

TRAVAUX DIRIGES – Semaine n°5

Thèmes

- Types abstraits de données Liste et File
- Algorithmique de liste et de file
- Résolution de problème

L'objectif du TD est de maîtriser l'utilisation des TDA Liste et File présentés en cours. Vous pourrez écrire les algorithmes utilisés en pseudo-langage avant de passer à leur implémentation en C++.

En annexe, nous rappelons les primitives des types `Liste` et `File` (cf. Cours n°6) ainsi que leurs primitives en langage pseudo-naturel.

Exercice 1. Crible d'Erathostène

L'algorithme du crible d'Eratosthène permet de trouver la liste des nombres premiers inférieurs ou égaux à `n`.

L'algorithme est le suivant :

(1) Construire le crible : une liste des entiers naturels de 2 à n
(2) Supprimer du crible tous les entiers qui ne sont pas premiers :
Balayer le crible depuis son début jusqu'à sa fin*
si un nombre est trouvé dans le crible (il est premier) *alors*
supprimer du crible tous les multiples de ce nombre
finsi
(3) En fin de traitement, le crible contient les nombres premiers

Remarque sur la limite du balayage (fin*) : *on peut démontrer que l'on peut limiter le traitement aux nombres inférieurs ou égaux à racine de n.*

1.1. Spécifiez et codez la fonction qui implémente l'algorithme du crible d'Eratosthène pour donner la liste des nombres premiers inférieurs ou égaux à n. Vous utiliserez le composant de Liste vu en cours (cf. Annexe1 page 3).

1.2. Codez le programme principal qui affiche la liste des nombres premiers inférieurs ou égaux à 200.

Exercice 2. Gestion d'exécution de processus

Avec un système d'exploitation qui permet l'exécution concurrente de plusieurs processus, ces derniers sont exécutés suivant leur ordre de lancement, chacun leur tour, pendant un quantum de temps CPU. Dans cet exercice, nous supposerons que le temps total d'exécution d'un processus est connu.

Un processus est caractérisé par son numéro et par le temps d'exécution nécessaire à sa terminaison. L'exécution d'un processus pendant un quantum de temps diminue d'autant le temps d'exécution qui lui reste (dans la limite du temps restant à exécuter).

L'objectif est de modéliser la gestion des processus (concurrents) en attente de ressource CPU dans les conditions décrites ci-dessus.

2.1. Définissez le type `Processus` permettant de représenter un processus.

Spécifiez et codez la fonction `creerProcessus` qui permet de créer un processus à partir de son numéro et de son temps total d'exécution.

2.2. Définissez le type Système permettant de représenter le système précédemment décrit. Puis, prototypez et codez les opérations suivantes :
`executerProcessus` : exécuter un processus pendant le quantum CPU,
`lancerProcessus` : lancer un processus,
`gererProcessus` : gérer l'exécution des processus concurrents.

2.3. Simulez le scénario suivant : lancement dans l'ordre de trois processus (numérotés 1, 2 et 3) de durée d'exécution respective 250 ms, 100 ms et 210 ms aux temps t_0, $t_0+\varepsilon$, $t_0+2\,\varepsilon$ ($\varepsilon < 50$ms). En supposant que le quantum CPU est de 100 ms et qu'aucun autre processus n'est lancé avant 1 seconde, gestion de l'exécution des processus. Visualisez l'état du système au cours du temps.

ANNEXE 1.

Primitives de Liste
Conteneur de base en mémoire dynamique et extensible (ConteneurTDE)

```
/** @brief Initialiser une liste l extensible de capacité c et de pas d'extension p (c>0 et p>0) */
void initialiser(Liste& l, unsigned int c, unsigned int p);
/** @brief Désallouer une liste l*/
 void detruire(Liste& l);
/** @brief Longueur d'une liste l */
 unsigned int longueur(const Liste& l);
/** @brief Lire un élément de la liste l à la position pos (0<=pos<longueur(l))*/
 Item lire(const Liste& l, unsigned int pos);
/** @brief Ecrire un item it dans une liste l à la position pos (0<=pos<longueur(l)) */
 void ecrire(Liste& l, unsigned int pos, const Item& it);
/** @brief Insérer un item it dans une liste l à la position pos (0<=pos<=longueur(l)) */
 void inserer(Liste& l, unsigned int pos, const Item& it);
/** @brief Supprimer un élément dans une liste à la position pos (longueur(l)>0 et 0<=pos<longueur(l)) */
 void supprimer(Liste& l, unsigned int pos);
```

Primitives de Liste en langage pseudo-naturel

Primitives	*Rôle*	*Précondition*
`N ← longueur(Liste)`	Nombre d'éléments de la liste	
`Item ← lire(Liste, N)`	Lecture de l'élément d'indice donné	Indice variant de 0 à longueur-1
`Liste ← écrire(Liste, Item, N)`	Ecriture d'un élément à un indice donné	Indice variant de 0 à longueur-1
`Liste ← insérer(Liste, Item, N)`	Insertion d'un élément à un indice donné	Indice variant de 0 à longueur
`Liste ← supprimer(Liste, N)`	Suppression de l'élément d'indice donné	Indice variant de 0 à longueur-1

De plus, la constante `ListeVide` permet d'obtenir une liste vide.

ANNEXE 2.

Primitives de File
Conteneur de base en mémoire dynamique et extensible (ConteneurTDE)

```
/** @brief initialiser une file f extensible de capacité c et de pas d'extension p (c>0 et p>0) */
void initialiser(File& f, unsigned int c, unsigned int p);
/** @brief Désallouer une file f */
void detruire(File& f);
/** @brief Test de file f pleine */
bool estPleine(const File& f);
/** @brief Test de file f vide */
bool estVide(const File& f);
/** @brief Entrer un item it dans la file f (f n'est pas pleine) */
void entrer(File& f, const Item& it);
/** @brief Sortir l'item en tête de a file f (f n'est pas vide) */
void sortir(File& f);
/** @brief Lire l'item en tête de file f (f n'est pas vide) */
Item tete(const File& f);
```

Primitives de File en langage pseudo-naturel

Primitives	*Rôle*	*Précondition*
`File ← entrer(File, Item)`	Ajout d'un élément en queue de file	
`File ← sortir(File)`	Suppression de l'élément en tête de file	La file n'est pas vide
`Item ← tête(File)`	lecture de l'élément en tête de file	La file n'est pas vide
`Booléen ← estVide(File)`	Indicateur de file vide	

De plus, la constante `FileVide` permet d'obtenir une file vide.