Structures de données abstraites : Piles et files

Les structures de données

- structures de base : entiers, flottants, caractères...
- structures linéaires :
  - tableaux
  - listes chaînées
- structures hybrides : tables de hachage
- structures arborescentes :
  - arbres binaires de recherche
  - tas
- graphes

Liste (chaînée) : définition et propriétés

- contient des éléments tous du même type
- définition récursive ; une liste est :
  - soit vide
  - soit un nœud de tête et une suite qui est une liste
- ```
  struct noeud{
    val: type des éléments contenus dans la liste
    suiv: référence vers le noeud suivant
  }
  ```
- une liste est une référence vers le premier nœud de la liste
- elle est vide si cette référence vaut `Nil`
- structure de données dynamique, pas de débordement
- pour accéder à un élément, il faut parcourir tous ceux qui le précède dans la liste
  ```

Structures de données abstraites

- permettent de manipuler un ensemble **dynamique** de données
- sont définies par les **opérations** qui manipulent les données
  - insertion d'un élément
  - recherche
  - element maximum ou minimum...
- on ne se préoccupe pas de leur implémentation
- l'implémentation peut être réalisée par différentes **structures de données**

Exemples :

- pile, file, file de priorité
- dictionnaire
- ensemble...

Ce qu'on va faire

On doit se servir d'une structure abstraite de données uniquement par les fonctions qui la manipule et pas en utilisant sa représentation concrète. Utile pour programmer simplement.

On va aussi expliquer l'implémentation concrète pour :

- calculer la complexité d'une opération et faire le choix de la meilleure structure pour un problème
- être capable de définir ses propres structures de données

La pile

Propriété principale : on supprime le dernier élément ajouté (politique LIFO : Last In First Out)

- `creer_pile(): Pile` : créé et retourne une pile vide ;
- `pile_vide(p: Pile): booleen` : teste si la pile est vide ;
- `insere_pile(p: Pile, x: entier)` : insère l'entier x dans la pile ; la pile est modifiée dans la fonction ;
- `extrait_pile(p: Pile): entier` : retire l'entier sur le dessus de la pile et retourne sa valeur ; la pile est modifiée dans la fonction ; **précondition :** la pile en entrée ne doit pas être vide ;

La pile : propriétés

- `pile_vide(creer_pile())`
  vaut `Vrai`
- `insere_pile(p, n); pile_vide(p)`
  vaut `Faux`
- `insere_pile(p, n); extrait_pile(p)`
  la pile retourne à son état initial

quid de
`n <- extrait_pile(p); insere_pile(p, n)`

La pile : exemples

Que contient la pile à la fin de cette séquence ?

```
p <- creer_pile()
insere_pile(p, 3)
insere_pile(p, 9)
extrait_pile(p)
insere_pile(p, 1)
insere_pile(p, 7)
n <- extrait_pile(p)
extrait_pile(p)
insere_pile(p, n)
```

Pile d'appels de fonctions

Donner l'évolution de la pile des appels du programme suivant

```
void f1(){
  f2()
}
void f2(){}
void f3(){
  f1()
}
void main(){
  f1()
  f3()
}
```

Algorithmes sur les piles

Écrire les algorithmes suivants

- taille d'une pile : retourne le nombre d'éléments dans une pile
- inverser une pile : inverse l'ordre des éléments dans la pile
- suppression du dernier élément d'une pile non vide : le dernier élément de la pile est retiré en conservant l'ordre des autres

Implémentation d'une pile

Les opérations se font en temps constant :

- avec un tableau si la pile est de taille bornée
- avec une liste chaînée

La file

Propriété principale : on supprime les éléments dans leur ordre d'arrivée (politique FIFO : First In First Out)

Exemples : files d'attente, buffer des événements clavier/souris...

- `creer_file(): File`
  créé et retourne une file vide ;
- `file_vide(f: File): booleen`
  teste si une file est vide ;
- `insere_file(f: File, x: entier)`
  insère l'entier x dans la file ; la file est modifiée dans la fonction ;
- `extrait_file(f: File): entier`
  retire un entier de la file et retourne sa valeur ; la file est modifiée dans la fonction ; **précondition :** la file en entrée ne doit pas être vide ;

Manipulation d'une file : exemple

```
f <- creer_file()
insere_file(f, 3)
insere_file(f, 9)
extrait_file(f)
insere_file(f, 1)
insere_file(f, 7)
n <- extrait_file(f)
extrait_file(f)
insere_file(f, n)
```

Implémentation d'une file

Les opérations se font en temps constant :

- avec un tableau si la file est de taille bornée
- avec une liste doublement chaînée

La file de priorité

Propriété principale : aux éléments sont associées des priorités. La file de priorité extrait l'élément de priorité maximale

Exemples : gestion des événements par le système d'exploitation, utilisé dans divers algorithmes (Dijkstra, Huffman, A*)...

- `creer_fdp(): fdp`
  retourne une file de priorité vide
- `fdp_vide(f: fdp): boolén`
  teste si la file de priorité est vide
- `insere_fdp(f: fdp, (n, p): entiers)`
  insère la valeur n de priorité p dans la fdp qui est modifiée par effet de bord
- `fdp extrait_fdp(f: fdp)`
  extrait l'entier de priorité maximale de la fdp et retourne sa valeur. La fdp est modifiée par effet de bord. **précondition :** la file en entrée ne doit pas être vide ;

La file de priorité : implémentation

Quelle structure de données utiliser pour implémenter efficacement les opérations de base d'une file de priorité ?