Cours d'Algorithmique et Complexité

Structures de données

Catalin Dima

Contenu du cours

- Rappels : piles, files, listes chaînées (de divers types).
- Complexité des opérations sur les implémentations des piles/files/listes chaînees.
- Tables de hachage.
- Arbres de divers types.

Piles

- **Toute** structure de donnée implémentant une politique de manipulation des données de type dernier arrivé, premier sorti (angl. LIFO).
- Classe fournissant trois méthodes :
  1. `pile.empiler(élément)` qui modifie donc la pile.
  2. `pile.dépiler()`, qui renvoie le sommet de la pile.
     - Cas de la pile vide : exception ! *(merci Java !)*
  3. `pile.vide()`, qui renvoie vrai/faux si la pile est vide ou pas.
- Comportement :

```
pile.empiler(2);
pile.empiler(10);
pile.empiler(-123);
pile.empiler(54);
pile.empiler(100);
int x = pile.dépiler(); // résultat x=100
```

Utilisation des piles

1. Évaluation des expressions : **pile** d'opérateurs.
2. Éditeurs de texte : fonction undo.
3. Navigateurs web : historique des pages visitées.
4. Interpréteurs de commandes (genre `bash`) : pile d'appels de fonctions.
5. Parcours en profondeur de graphes/arbres.

Implémentation de piles à l'aide des tableaux

Implémentation Java :

```
class Pile{
    private int tab[];
    private int taille;
    private int taillemax;
    }
    Pile(int nb){
        tab = new int[nb];
        taillemax=nb;
        taille=0;
    }
}
```

Implémentation de piles à l'aide des tableaux (suite)

```
public void empiler(int elem){
    if (taille >= taillemax){
        int nvtab[] = new int[taillemax+100];
        tabcopy(tab,nvtab);
           // fonction statique de la classe Pile
           // à implémenter !
        tab=nvtab;
        taillemax=taille+100;
    }
    tab[taille++]=elem;
    return;
}
public int depiler() throws MonExceptionPileVide {
    if (taille==0) throw new MonExceptionPileVide();
    return tab[--taille];
}
```

Implémentation de piles à l'aide des tableaux (suite 2)

- Souvent, pour éviter de gaspiller la mémoire, on rajoute du code dans `depiler` pour diminuer la taille du tableau !
- Par exemple, lorsque `taille < taillemax-100`.
  - Implémentez-le !
- Mais chaque modification de la taille du tableau nécessite des opérations supplémentaires, qui modifient de manière significative la durée d'exécution des opérations `empiler/dépiler`.
- On peut faire une analyse du temps moyen nécessaire :
  - Sans modification de la taille du tableau : temps constant !
  - Avec modification : au pire de cas, temps linéaire dans la taille de la pile !
- Morale de l'implémentation à l'aide des tableaux :
  - Facile, mais...
  - Parfois coûteuse !

Files

- **Toute** structure de donnée implémentant une politique de manipulation des données de type premier arrivé, premier sorti (angl. FIFO).
- Classe fournissant trois méthodes :
  1. `file.enfiler(élément)` (ou `file.inserer(element)` qui modifie donc la pile.
  2. `file.défiler()` (ou `file.supprimer()` qui renvoie le sommet de la pile.
     - Cas de la file vide : exception !
  3. `file.vide()`, qui renvoie vrai/faux si la pile est vide ou pas.
- Comportement :

```
file.enfiler(2);
file.enfiler(10);
file.enfiler(-123);
file.enfiler(54);
file.enfiler(100);
int x = file.defiler(); // résultat x=2
```

Utilisation des files

1. Ordonnancement des processus : l'algorithme du tourniquet (ou Round-Robin), implémenté dans la plupart des systèmes d'exploitation.
2. Traîtement des requêtes par les serveurs de n'importe quel type.
3. Programmation événementielle.
4. Lecture/écriture des données à l'aide des tampons.
5. Algorithmes de parcours en largeur de graphes/arbres.

Implémentation des files à l'aide des tableaux

```
class File{
    private int tab[];
    private int fin;
    private int taillemax;
    }
    Pile(int nb){
        tab = new int[nb];
        taillemax=nb;
        fin=0;
    }
}
```

Implémentation de files à l'aide des tableaux (suite)

```
public void enfiler(int elem){
    if (fin >= taillemax){
        int nvtab[] = new int[taillemax+100];
        tabcopy(tab,nvtab);  // à implémenter !
        tab=nvtab;
        taillemax=fin+100;
    }
    tab[fin++]=elem;
    return;
}
public int defiler() throws MonExceptionFileVide {
    if(fin==0) throw new MonExceptionFileVide();
    int val=tab[0];
    for(i=0;i<fin-1;i++)
        tab[i]=tab[i+1];
    return val;
}
```

- Sauf que l'action de défiler est trop coûteuse !
- Combien ?...

Implémentation de files à l'aide des tableaux (suite 2)

- Recherche implémentation de `defiler` qui s'exécute en temps constant !
- Idée : utiliser deux indices : `debut` et `fin`.
- Tant que `fin < taillemax` on enfile en fin de tableau.
- Mais lorsque `fin = taillemax` et il faut enfiler, **on enfile à l'indice 0** !
- Notre tableau est vu de manière circulaire !
  - Dessin au tableau...
- Petit problème : comment différencier une file vide d'une file pleine ?...
  - Car si on remplit complètement le tableau, on aura `debut == fin`, pareil que si le tableau est vide !
- Solution : une case du tableau reste toujours vide !
- Test de file pleine : `debut == (fin+1) % taillemax`.
  - Alors soit on lève une exception,
  - ... soit on crée un nouveau tableau et on recopie le contenu de la file.

Implémentation de files à l'aide des tableaux (suite 3)

```
class File{
    private int tab[];
    private int debut, fin;
    private int taillemax;
    }
    public void enfiler(int elem){
        if ((fin == (debut+1) % taillemax )){
            // file pleine, on augmente la taille
        }
        tab[fin]=elem;
        fin=(fin+1) % taillemax;
        return;
    }
    public int defiler() throws MonExceptionFileVide {
        if(debut == fin) throw new MonExceptionFileVide();
        int val = tab[debut];
        debut = (debut+1) % taillemax;
        return val;
    }
```

- Mêmes remarques que pour les piles quant à la complexité des opérations enfiler/défiler !
- Recherche implémentations plus rapides !

Listes chaînées

- Structure de données permettant de stocker des objets dans un ordre linéaire.
- Chaque objet possède un lien vers le suivant dans l'ordre des objets dans la liste.
- Le lien du dernier objet est d'un type particulier ne pouvant pas représenter un objet de la liste (presque tjrs un pointeur vide !).
- Liste doublement chaînée : l'ordre linéaire croissante est doublée de l'ordre inverse.
  - Donc chaque objet possède un lien vers son successeur et un autre vers son prédécesseur.
- Implémentation en C connue !
- Liste circulaire : l'ordre n'est pas linéaire mais circulaire.
  - Donc le successeur du "dernier" objet est le "premier".

Listes chaînées

- Implémentation en Java des listes simplement chaînées :

```
class Elem{
    int val;
    public Elem nxt;
    Elem(int nb){
        val=nb;
        nxt=null;
    }
    int contenu(){
        return val;
    }
}
class Liste{
    Elem tete;
}
class MonProg{
    public static void main(String args[]){
        Liste l;
        l.tete = new Elem(100);
        l.tete.nxt = new Elem(200);
.....
}
```

Implémentation de piles à l'aide des listes chaînées

- Pour éviter d'avoir des empilements et dépilements qui nécessitent du temps linéaire.
- Implémentation en C : déjà vue en cours d'algorithmique !
  - Faut-il la rappeler ?...
- Implémentation en Java possible aussi !

```
class Pile{
    Elem tete;
    Pile(){
      tete=null;
    }
    void empiler(int v){
      Elem e = new Elem(v);
      e.nxt=tete;
      tete=e;
    }
    int depiler() throws MonExceptionPileVide {
      if(tete==null) throw new MonExceptionPileVide();
      Elem temp=tete;
      tete=tete.nxt;
temp.nxt=null;
      return temp.val();
    }
}
```

Complexité des opérations avec les piles basées sur les listes chaînées

- Très dépendante de la complexité des créations d'objets !
- Le dépilement crée un objet qui n'est plus référencé par qui que ce soit !
  - C'est l'objet `temp` de la classe `Elem`.
- C'est le garbage collector qui se chargera d'éliminer cet objet !
  - Il se met en route périodiquement et supprime les objets non-référencés.
  - Ce qui sera le cas lorsqu'on sort de la fonction `depiler`.
  - Remarquer l'instanciation `temp.nxt=null`, bien utile pour éviter de préserver les références des objets dans la pile qui ne servent plus à rien, et donc duper le garbage collector !

Implémentation de files à l'aide des listes chaînées

Avec des listes doublement chaînées !

```
class File{                          |     class Elem{
    Elem debut, fin;                 |        int cont;
    Pile(){                          |        Elem nxt,prev;
      tete=null;                     |        Elem(int nb){
    }                                |            cont=nb;
    void enfiler(int v){             |            nxt=prev=null;
      Elem e = new Elem(v);          |        }
      if(debut){                     |     }
         fin.nxt=e;
         e.prev=fin;
         fin=e;
      }
      else{
         debut=fin= new elem(v);
      }
    }
    int defiler() throws MonExceptionPileVide {
      if(debut==null) throw new MonExceptionPileVide();
      Elem temp=debut;
      debut=debut.nxt;
      debut.prev=temp.nxt=null;
      return temp.val();
    }
}
```

Implémentation des pointeurs en tableaux

- Tableaux de structs/objets contenant les liens (next/prev).
- C'est le type d'implémentation du tas qui existe dans chaque langage de programmation !
- Les cases mémoire libres sont aussi représentées à l'aide des listes chaînées.

Classe `Stack` en Java

- Implémentation utilisant `Vector`.
  - Donc c'est une implémentation de type tableau !
- Classe générique : on peut stocker n'importe quel objet !
- Même plus, on peut instancier une pile d'objets de type particulier :

```
Stack<Integer> mapile = new Stack();
        // on pourra y mettre que des entiers !
```

- Constructeur `Stack()` créant une pile vide.
- Méthodes :
  - `boolean empty()`.
  - `E peek()` : renvoie l'objet au sommet de la pile, sans le dépiler !
  - `E push(E item)` : empile un objet (de type générique ou de type compatible avec la déclaration de l'objet pile).
  - `E pop()` : renvoie l'objet au sommet de la pile et le dépile.

Exemple d'utilisation...

Et les files ?...

- Interface `Queue` générique.
- Diverses implémentations :
  - `AbstractQueue, ArrayDeque, DelayQueue, LinkedList...`
- Qqs méthodes à implémenter :
  - `E poll()` : dé-file l'élément le plus ancien.
  - `E peek()` : renvoie l'élément le plus ancien de la file sans le dé-filer.
  - `boolean offer(E elem)` : enfile l'argument.
- Classe `LinkedList` fournissant des méthodes (beaucoup) de manipulation des listes doublement-chaînées.

Arbres binaires

- Structure de données contenant des objets qui possèdent chacun au moins trois champs : valeur, fils-gauche et fils-droit.
- Souvent il est utile d'avoir aussi un 4e champ : parent.
- Utilisation :
  - Stockage structuré de l'information de façon à optimiser les recherches – recherche dichotomique.
  - Compilation : arbre syntaxique.
  - Tas = arbre "linéarisé".
- Opérations :
  - Parcours d'arbre **en préfixe** : "visiter/manipuler" (récursivement) un noeud, puis ses fils.
  - Parcours **en infixe** : d'abord le fils gauche, puis le noeud, puis son fils droite.
  - Parcours **en suffixe** : d'abord les deux fils, puis le noeud.
- Parcours servant à calculer une fonction (valeur d'une expression pour un arbre syntaxique), à modifier les informations dans l'arbre (tri par tas), ou simplement à chercher une information.

Arbres binaires en Java

```
class Noeud{
   int cont;
   Noeud pere, filsg, filsd;
   Noeud(int val){
      cont=val;
      pere=filsg=filsd=null;
   Noeud(int val, Noeud p, Noeud n1, Noeud n2){
      cont=val;
      pere=n1;
filsg=n1;
filsd=n2;
}
class Tree{
   Noeud racine;
   Tree(int valracine, Tree a1, Tree a2){
      racine = new Noeud(valracine, null, a1.racine, a2.racine);
      a1.racine.pere = racine;
      a2.racine.pere = racine;
   }
   static int sommeinfix(Noeud n){
      if(!n) return 0;
      return sommeinfix(n.filsg) + n.cont + sommeinfix(n.filsd);
   }
}
```

Arbres à ramification non-bornée

```
class Noeud{
   int cont;
   Noeud pere, fils, frere;
   Noeud(int val){
      cont=val;
      pere=fils1=frere=null;
   Noeud(int val, Noeud p, Noeud n1, Noeud n2){
      cont=val;
      pere=n1;
fils1=n1;
frere=n2;
}
class Treenb{
   Noeud racine;
   Treenb(int valracine, ArrayList<Noeud> fils){
      ListIterator<Noeud> li = fils.listIterator();
      racine = new Noeud(valracine, null, li.next(), null);
      Noeud n=racine.fils;
      while(li.hasNext()){
          n.frere=li.next();
          n.frere.pere=racine;
          n=n.frere;
      }
   }
}
```