Module TC2
-
Master 1 EEA-AEETS

Nicolas Rivière

Université Toulouse III

Septembre 2011

Plan du cours

- Les structures de données
- Algorithmes de tri
- Compilation séparée

- Piles.
- Files.
- Listes.

Piles : définition

- Une pile est un ensemble de données auxquelles on accède dans l'ordre inverse où elles sont insérées : Last Input First Output, LIFO.
- Seul le sommet de la pile est accessible.
- Trois opérations sur une pile :
  - empiler élément : rajoute un élément sur la pile ;
  - dépiler élément : renvoie la valeur de l'élément au sommet de la pile et le supprime ;
  - vide : renvoie VRAI si et seulement si la pile est vide.
- Utilisation d'une pile (1) : gestion des appels et retours de fonctions.
- Utilisation d'une pile (2) : passage des paramètres à une fonction ou stockage des variables locales.
- Implémentation simple par tableau ; liste chaînée possible.

Piles : Exemple d'utilisation

- Calcul en notation polonaise inversée (chaque opérateur vient après son dernier argument)
- Forme postfixée donc pas de parenthèse
- Soit l'opération : $\frac{4+2\sqrt{16}}{6}$
- la liste suivante représente cette opération :
  `{(, 4, +, 2, *, sqrt, (, 16, ), ), /, 6}`
- sous forme polonaise inversée : `{4, 2, 16, sqrt, *, +, 6, /}`
- Un algorithme peut être...

Piles : Exemple d'utilisation

```
pour chaque élément=valeur
  si élément=valeur
     empiler(valeur)
  si élément=opérateur unaire
     opération=élément
     résultat=opération(dépiler)
     empiler(résultat)
  si élément=opérateur binaire
     opération=élément
     opérande1=dépiler
     opérande2=dépiler
     résultat=opération(opérande1,opérande2)
     empiler(résultat)
```

Files

- Une file est un ensemble de données qui sont ajoutées en queue de file et supprimées en tête : First Input First Output, FIFO.
- Trois opérations sur une file :
  - empiler élément : rajoute un élément en queue de la file ;
  - dépiler élément : renvoie la valeur de l'élément en tête de la file et le supprime ;
  - vide : renvoie VRAI si et seulement si la file est vide.
- Utilisation d'une file : applications pilotées par des événements.
- Utiles pour simuler des files d'attente.

Listes

- Une liste est un ensemble fini d'éléments ordonnés noté $L = e_1, e_2, ..., e_n$
- Utilisées pour gérer des ensembles de données.
- Efficaces pour des opérations d'insertion et de suppression d'éléments.
- Manipulation de l'allocation dynamique car taille variable en cours d'exécution.
- Opérations effectuées élément par élément.
- Trois types de listes :
  - simplement chaînées
  - doublement chaînées
  - circulaires

Opérations sur une liste

- Créer une liste
- Supprimer une liste
- Rechercher un élément particulier
- Insérer un élément
- Supprimer un élément particulier
- Permuter deux éléments
- Concaténer deux listes
- etc.

Listes simplement chaînées(1)

- Liaison entre les différents éléments par un simple pointeur
- Chaque élément est formé de 2 parties :
  1. un champ contenant une donnée ou un pointeur vers une donnée
  2. un pointeur vers l'élément suivant de la liste
- Le premier élément est la *tête*.
- Le dernier élément est la *queue*.
- Le dernier élément est initialisé à une valeur fixe : `NULL` en C.

Listes simplement chaînées(2)

- Sens unique : Accès à un élément en parcourant la liste de la *tête* vers la *queue*.
- Eléments créés par allocation dynamique donc pas contigus en mémoire.
- La suppression d'un élément sans précaution ne permet plus d'accéder aux éléments suivants

```
struct s_element
{
  int donnee;
  struct s_element* suivant;
};
typedef struct s_element t_element;
```

Listes doublement chaînées

- Liaison entre les différents éléments par deux pointeurs
- Chaque élément est formé de 3 parties :
  1. un champ contenant une donnée ou un pointeur vers une donnée,
  2. un pointeur vers l'élément suivant de la liste,
  3. un pointeur vers l'élément précédent de la liste.

```
struct s_element
{
  int donnee;
  struct s_element* suivant;
  struct s_element* precedent;
};
typedef struct s_element t_element;
```

Listes circulaires

- Liaison entre les différents éléments par un seul pointeur.
- Pas de *queue* !
- Chaque élément est formé de 2 ou 3 parties :
  1. un champ contenant une donnée ou un pointeur vers une donnée,
  2. un pointeur vers l'élément suivant de la liste,
  3. un pointeur vers l'élément précédent de la liste, si doublement chaînée.

Opérations sur une liste (1) : Insertion

- Insertion peut se faire :
  - en tête
  - en queue
  - n'importe où
- Opérations à effectuer dans l'ordre :
  - allouer de la mémoire pour le nouvel élément
  - copier les données
  - faire pointer le nouvel élément ves le suivant de celui pointé par *courant*

Insertion : Exemple d'implémentation

```
void insertion(t_element** tete, t_element* courant, int data)
{
t_element* nouveau;
nouveau=(t_element*) malloc(sizeof(t_element));
nouveau->donnee=data;
if (courant!=NULL)
{
nouveau->suivant=courant->suivant;
courant->suivant=nouveau;
}
else /* insertion en tete ou liste vide*/
{
nouveau->suivant=*tete;
*tete=nouveau;
}
}
```

Opérations sur une liste (1) : Suppression

- Opérations à effectuer dans l'ordre :
  - récupérer le pointeur de tête par adresse pour pouvoir le modifier si la suppression rend la liste vide ou si l'élément à supprimer était le premier de la liste.
  - distinguer s'il s'agit du premier élément
  - dans ce cas, chercher le précédent de courant
  - et récupérer le lien contenu dans le champ "suivant" de l'élément à supprimer
  - sinon modifier le pointeur de tête.
  - libérer la mémoire !

Suppression : Exemple d'implémentation

```
void suppression(t_element** tete, t_element* courant)
{
t_element* precedent;
precedent=(t_element*) malloc(sizeof(t_element));
if (courant!=*tete) /* pas le premier element */
{
  precedent=*tete;
  while (precedent->suivant!=courant)
  precedent=precedent->suivant;
  precedent->suivant=courant->suivant;
}
else /* suppression du premier element*/
*tete=courant->suivant;
free(courant);
}
```

Suppression : Exemple d'utilisation(1)

```
int main()
{
t_element* tete=NULL;
t_element* courant=NULL;
/* insertion du premier element : liste=1 */
insertion(&tete,courant,1);

/* deuxieme insertion : liste=1 2 */
courant=tete;
insertion(&tete,courant,2);

/* troisieme insertion : liste=1 2 3*/
courant=courant->suivant;
insertion(&tete,courant,3);
...
```

Suppression : Exemple d'utilisation (2)

```
 /* courant pointe sur 2 : liste = 1 2 4 3 */
insertion(&tete,courant,4);

/* insertion en tete : liste = 5 1 2 4 3 */
insertion(&tete,NULL,5);

/* suppression de l'élément 1 : liste = 5 2 4 3 */
courant=tete->suivant;
suppression(&tete,courant);

/* suppression de la tete : liste = 2 4 3 */
courant=tete;
suppression(&tete,courant);
return 0;
}
```

- Tri par sélection.
- Tri à bulles.
- Tri par insertion.

Tri par sélection

- Principe
  - Soit un ensemble de n éléments indicés de 0 à n-1
  - On suppose les m premiers éléments triés : 0 à m-1
  - On cherche la position k du plus petit élément parmi les éléments m à n-1
  - On le permutte avec l'élément m : ensemble trié de 0 à m
  - On parcourt donc l'ensemble de m=0 à n-2
- Exemple

18	10	3	25	9	2
2	10	3	25	9	18
2	3	10	25	9	18
2	3	9	10	25	18
2	3	9	10	18	25

Tri par sélection : Algorithme

- Soit à trier un tableau de N éléments
- POUR m=0 à N-2 FAIRE
- k $\leftarrow$ p (indice du plus petit élément de m à N-1)
- SI k différent de m ALORS permutter élément k avec élément m
- FIN SI
- FIN POUR
- L'implémentation peut se faire à partir de 3 fonctions : calcul du plus petit indice entre m et N-1, permutation et tri.

Tri à bulles : Principe

- Soit à trier un tableau de N éléments
- Il faut parcourir les éléments de l'ensemble de i=0 à n-1 en permuttant les éléments consécutifs non ordonnés.
- On recommence de i=0 à N-2 si pas de permutation à l'étape précédente
- Si pas de permutation alors ensemble trié

Tri à bulles : Exemple

18	10	3	25	9	2
10	18	3	25	9	2
10	3	18	25	9	2
10	3	18	25	9	2
10	3	18	9	25	2
10	3	18	9	2	25
3	10	18	9	2	25
3	10	18	9	2	25
3	10	9	18	2	25
3	10	9	2	18	25
3	10	9	2	18	25
3	9	10	2	18	25
3	9	2	10	18	25
3	9	2	10	18	25
3	2	9	10	18	25
2	3	9	10	18	25

Tri à bulles : Algorithme

- k $\leftarrow$ N-1
- FAIRE
- POUR i=0 à J-1 FAIRE
- SI élément i $>$ élément i+1 ALORS
- permutter élément i avec élément i+1
- permutation=VRAI
- FIN SI
- FIN POUR
- TANT QUE permutation=VRAI
- L'implémentation peut se faire à partir de 3 fonctions : calcul du plus petit indice entre m et N-1, permutation et tri.

Tri par insertion : Principe

- Soit un ensemble de n éléments indicés de 0 à n-1
- On prend 2 éléments et on les met dans l'ordre
- On prend un 3è élément qu'on insère entre les 2 éléments déjà triés et ainsi de suite
- Un élément d'indice m va être inséré dans l'ensemble des éléments triés de 0 à m-1
- Il faut donc chercher l'élément de valeur immédiatement supérieure ou égale à celle de l'élément à insérer
- Si k est l'indice de cet élément alors on décale les éléments k à m-1 vers k+1 à m
- L'élément à insérer est placé en position k

Tri par insertion : Exemple

18	10	3	25	9	2
10	18	3	25	9	2
3	10	18	25	9	2
3	10	18	25	9	2
3	9	10	18	25	2
2	3	9	10	18	25

Tri par insertion : Algorithme

- ???

Récursivité

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Calcul de factorielle : $n! = n * (n-1) * (n-2)...3 * 2 * 1$

```
long int factorielle(int n)
{
  long int Facto=1;
  int i;

  for (i=1;i<=N;i++)
    Facto = Facto*i;

  return Facto;
}
```

Récursivité

- De manière récursive : Facto(n)=n*Facto(n-1) avec F(0)=1

```
long int factorielle(int n)
{
  if (N==0)
    return 1;
  else
    return N*factorielle(N-1);
}
```

Préprocesseur

- Lors de chaque compilation, le compilateur C fait appel lors du premier passage au préprocesseur
- Le préprocesseur est un traitement spécialisé qui permet
  - la macro-substitution
  - l'inclusion de fichiers
  - la compilation conditionnelle
- Le préprocesseur n'obéit qu'aux lignes de commande commençnt par le caractère #, qui doit être placé en première colonne

La macro-substitution(1)

- Permet de remplacer dans le texte du fichier, un identificateur (mot, opérateur, symbole, ...) par un texte de substitution
  ```
  #define identificateur texte_de_substitution
  #define TAILLE 10
  ```
- Utilisation des majuscules pour les constantes
- Allègement du programme
- Si le texte de substitution s'écrit sur plus d'une ligne alors utiliser le caractère \
  ```
  #define TROG_LONG "attention! cette phrase est trop \
  longue pour tenir sur une seule ligne"
  ```

La macro-substitution(2)

- Possibilité de définir des macro-instructions

```
#define CUBE(X) X*X*X
int puiss3, val=4;
puiss3 = CUBE(val);
```

- dans le cadre d'expressions numériques, utiliser les parenthèses autour de l'expression et de chaque arguments :
  `#define CUBE(X) ((X)*(X)*(X))`
- dans le cas d'instructions, utiliser des accolades autour de l'expression
  `#define ERREUR(num) {printf("\nErreur %d\n", (num));}`

Le préprocesseur

Inclusion de fichiers :

- Le contenu d'un fichier texte peut être inséré dans un autre fichier grâce à la directive `include`
- les fichiers inclus contiennent en général des déclarations de fonctions, de types, de constantes : leur suffixe est `.h`
- Un grand nombre des fichiers standards se trouvent dans `/usr/include`
- Leur inclusion s'effectue avec des chevrons "<,>" ainsi : `#include <stdio.h>`
- L'inclusion de fichiers créés par l'utilisateur se fait avec des guillemets : `#include "ma-biblio.h"`

Compilation conditionnelle

- Il est possible de choisir des parties de programme source qui seront compilées ou pas dans le programme objet selon les valeurs prises par des expressions constantes.
- Directives pour la compilation conditionnelle :
  `if else endif ifdef ifndef elif`
- Exemple :

```
#ifndef TAILLE
#define TAILLE 100
#endif
```

- Problème des inclusions multiples : lorsqu'un fichier est inclus, faire en sorte qu'il le soit bien une seule fois.

Construction d'un fichier ".h"

Une bonne manière de construire les fichiers d'en-tête .h :

```
#ifndef MA-BIBLIO_H
#define MA-BIBLIO_H
struct S{
  int i;
  int j;
};
void mafonction(void);
#endif
```

Paramètres de ligne de commande

- Possibilité de passer des arguments (paramètres) à l'exécution d'un programme : `le_prog Jean Dupont`
- `int main(int argc, char * argv[])`
- `argc` est un entier contenant le nombre d'arguments
- `argv` est un tableau de chaînes de caractères dont la 1è chaîne `argv[0]` est le nom du programme
- Les chaînes peuvent être converties au besoin
  $\Rightarrow$ fonctions de la librairie `stdlib.h`

Pointeurs génériques

- Le langage C ne permet pas d'affecter des pointeurs de types différents
- Possibilité de créer des fonctions indépendamment du types des paramètres
  $\Rightarrow$ pointeurs génériques de type `void *`
- Problème car on ne peut accéder au contenu avec l'opérateur *
- Utilisation de la fonction `memcpy()` pour copier le contenu d'un variable pointée vers une autre :
  `void *memcpy(void * pa, void * pb, unsigned int taille)`
- On copie "taille" octets de adresse "pb" vers adresse "pa"
- Pour obtenir la taille : fonction `sizeof()`

Pointeurs génériques : exemple

Une fonction qui permute n'importe quel type de valeurs

```
void permuter(void* pa, void* pb, unsigned int taille)
{
void * pc;

pc = malloc(taille);
memcpy(pc,pa, taille);
memcpy(pa,pb,taille);
memcpy(pb,pc,taille);
free(pc);
}
```

Pointeurs génériques : exemple

Elle pourra être utiliser ainsi :

```
int ia=4,ib=5;
float fa=2.1,fb=1.2;

permuter(&ia,&ib,sizeof(ia));
permuter(&fa,&fb,sizeof(fa));
```