

Structures de données (LU2IN006)

Matrices, Listes, Files, Piles

Nawal Benabbou

Licence Informatique - Sorbonne Université

2022-2023



Un premier type de données abstrait : les matrices

Définition

Une matrice est un ensemble d'éléments **accessibles par un index** (entier) à une ou deux dimensions (parfois plus). Opérations classiques : accéder à un élément, modifier un élément, etc.

Implémentation en langage C

- **Matrice à une dimension** : ces matrices correspondent exactement aux tableaux C à une dimension (voir les rappels de C sur moodle).
- **Matrice à deux dimensions** : ces matrices peuvent être codées par des tableaux C à deux dimensions. Comme un tableau à deux dimensions peut être vu comme un tableau de tableaux, il est représenté par un pointeur sur pointeur en langage C.

Remarque : il est possible de coder une matrice à deux dimensions par un tableau C à une dimension en “aplatissant la matrice” ligne par ligne. Plus précisément, il suffit de faire correspondre la case (i, j) de la matrice avec la case en position $i*n+j$ du tableau, où n est le nombre de colonnes de la matrice.

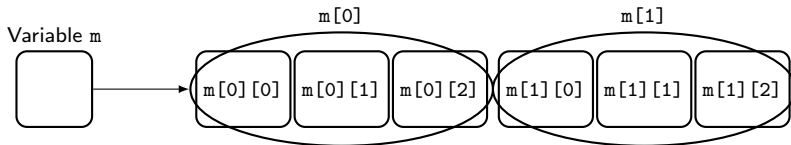
Les tableaux à deux dimensions en langage C

Allocation statique

En mémoire, les données sont stockées les unes après les autres, ligne par ligne. Un tableau C à deux dimensions peut être alloué de différentes manières. Par exemple :

- `int m[2][3];` permet d'allouer l'espace mémoire pour un tableau 2D de taille 2×3 contenant des entiers, sans initialiser ses valeurs.
- `int m[2][3] = {{1,5,4}, {3,2,1}};` permet de faire les deux.
- `int m[2][3] = {1,5,4,3,2,1};` permet de faire la même chose.

Remarque : `m` est ici de type `int**` (pointeur sur pointeur sur entier).



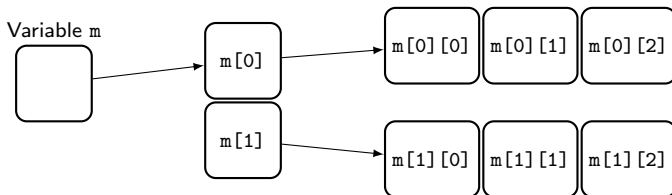
Les tableaux à deux dimensions en langage C

Quand la mémoire ne possède pas de zone espace contiguë assez grande, on préfère réaliser une allocation dynamique plutôt que statique.

Allocation dynamique

Allouer dynamiquement un tableau à une dimension, dont chaque case pointe sur un tableau à une dimension alloué dynamiquement. Exemple avec le même tableau à deux dimensions :

```
1 int i;  
2 int** m = (int**) malloc(2*sizeof(int*));  
3 for (i=0;i<2;i++){  
4     m[i] = (int*) malloc(3*sizeof(int));  
5 }
```



Les tableaux à deux dimensions en langage C

Désallocation : libérer les sous-tableaux, avant de libérer le tableau principal.

```
1 int i;  
2 for (i=0;i<2;i++){  
3     free(m[i]);  
4 }  
5 free(m);
```

Gestion des problèmes d'allocation : libérer la mémoire allouée avant l'erreur.

```
1 int i,j;  
2 int** m = (int**) malloc(2*sizeof(int*));  
3 if (m == NULL){  
4     return NULL;  
5 }  
6 for (i=0;i<2;i++){  
7     m[i] = (int*) malloc(3*sizeof(int));  
8     if (m[i]==NULL){  
9         for (j=0;j<i;j++){  
10             free(m[j]);  
11         }  
12         free(m);  
13         return NULL;  
14     }  
15 }
```

Complexité des opérations sur les données

Question : Pour un tableau de taille n , donner la complexité des opérations suivantes (en utilisant la notation de Landau O) :

- **Accès :** cette opération consiste à retourner un élément dont on connaît l'index.
→ Opération en $O(1)$ (temps constant).
- **Recherche :** cette opération consiste à rechercher si un élément est dans le tableau, et à retourner son indice si l'élément y est.
→ Opération en $O(n)$ dans le cas général car on parcourt le tableau entièrement dans le pire des cas.

Remarque : Cette complexité peut être améliorée dans le cas de tableaux triés (recherche dichotomique en $O(\log(n))$).

- **Suppression :** cette opération supprime un élément dont on connaît l'index.
→ Cette opération est en $O(n)$, car on doit décaler les éléments restants pour garder les cases du tableau contiguës.

Remarque : à la place, on pourrait mettre la case correspondante à une valeur particulière (par exemple "-1") pour indiquer qu'elle est "vide", ce qui donnerait une complexité en $O(1)$, mais l'opération de recherche serait plus compliquée...

- **Insertion :** cette opération consiste à insérer un élément dans une case libre.
→ Cette opération est en $O(1)$ (si les cases sont maintenues contiguës, sinon $O(n)$ pour trouver la première case "vide").

Remarque : si on souhaite dépasser la taille du tableau, cette structure de données n'est peut-être pas celle qui est la plus adaptée à votre application.

Un autre type de donnée abstrait : les listes

Définition

Une liste est un type de données abstrait défini par une suite finie d'éléments (x_1, \dots, x_n) . La liste est vide si $n = 0$. On passe de la case contenant l'élément x_i à la case contenant l'élément x_{i+1} par une fonction successeur. Une liste (non vide) a deux extrémités, que l'on appelle début (ou tête) et fin (ou queue).

Opérations classiques : recherche, suppression, insertion, etc.

Implémentation en langage C

Une liste peut être codée par un tableau C à une dimension, mais en l'absence d'indices ou de tri sur les éléments, on n'a aucun intérêt à utiliser de la mémoire contiguë, qui consomme beaucoup de place en mémoire et qui pose des problèmes lors de l'insertion et de la suppression d'un élément.

⇒ on privilégie une implémentation par *liste chaînée*.

Liste (simplement/doublement) chaînée

Définition

Une liste chaînée est une structure de données définie par un ensemble de cellules, possédant chacune un élément qui pointe sur un autre élément de la liste, de sorte à former une chaîne linéaire (ou une "ligne").

La liste est dite *simplement chaînée* si elle ne possède qu'une fonction successeur, et *doublement chaînée* si elle possède en plus une fonction prédécesseur permettant d'accéder à l'élément précédent.

Implémentation avec struct

En langage C, on va coder une cellule à l'aide d'un struct. Exemple pour une liste composée de paires (lettre, nombre), on peut utiliser le struct suivant :

```
1 typedef struct cellule {  
2     char lettre;  
3     int nombre;  
4     struct cellule *suiv;  
5 } Cellule;
```

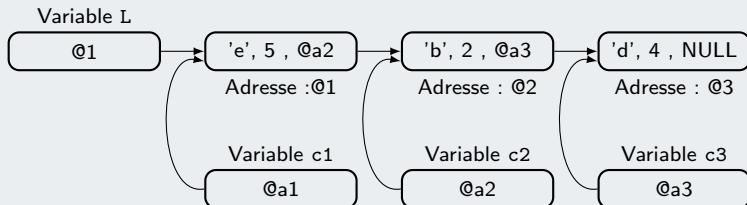
Pour une liste doublement chaînée, il faudrait ajouter un champ prec.

Représentation avec une liste simplement chaînée

Exemple

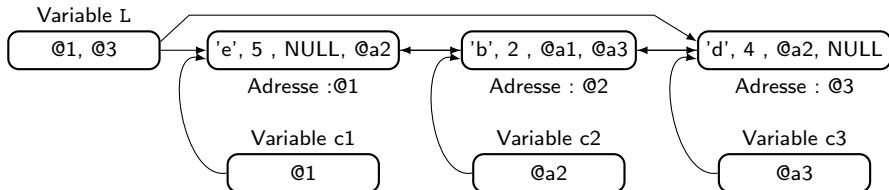
Construire une liste L contenant ('e',5), ('b',2), puis ('d',4).

```
1 Cellule* c1 = (Cellule*) malloc (sizeof(Cellule));
2 Cellule* c2 = (Cellule*) malloc (sizeof(Cellule));
3 Cellule* c3 = (Cellule*) malloc (sizeof(Cellule));
4 Cellule* L = c1;
5 c1->lettre='e';
6 c1->nombre=5;
7 c1->suiv=c2;
8 c2->lettre='b';
9 c2->nombre=2;
10 c2->suiv= c3;
11 c3->lettre='d';
12 c3->nombre=4;
13 c3->suiv=NULL;
```



Représentation avec une liste doublement chaînée

```
1 typedef struct tete { //On utilise une tete fictive
2     Cellule* debut;
3     Cellule* fin;
4 } TeteListe;
5 TeteListe* L = (TeteListe*) malloc (sizeof(TeteListe));
6 Cellule* c1 = (Cellule*) malloc (sizeof(Cellule));
7 Cellule* c2 = (Cellule*) malloc (sizeof(Cellule));
8 Cellule* c3 = (Cellule*) malloc (sizeof(Cellule));
9 L->debut=c1;
10 L->fin=c3;
11 c1->lettre='e';
12 c1->nombre=5;
13 c1->prec=NULL;
14 c1->suiv=c2;
15 c2->lettre='b';
16 c2->nombre=2;
17 c2->prec=c1;
18 c2->suiv= c3; //etc...
```



Quelques opérations sur les listes simplement chaînées

```
1  /* Affiche le premier element de la liste L */
2  void affiche_debut(Cellule *L){
3      if (L==NULL){
4          printf("Liste_vide\n");
5      } else {
6          printf("(%c,%d)\n",L->lettre,L->nombre);
7      }
8  }
```

```
1  /* Affiche le dernier element de la liste L */
2  void affiche_fin(Cellule *L){
3      Cellule *cour;
4      if (L==NULL){
5          printf("Liste_vide\n");
6      } else {
7          cour=L;
8          while (cour->suiv!=NULL){
9              cour=cour->suiv;
10             }
11         printf("(%c,%d)\n",cour->lettre,cour->nombre);
12     }
13 }
```

Quelques opérations sur les listes simplement chaînées

```
1 void affichage_des_elements(Cellule *L){
2     Cellule* cour=L;
3     while (cour!=NULL){
4         printf("(%c,%d)_",cour->lettre, cour->nombre);
5         cour=cour->suiv;
6     }
7     printf("\n");
8 }
```

```
1 int possede_element(Cellule *L, char c, int i){
2     Cellule* cour=L;
3     while ((cour!=NULL)&&
4         ((cour->lettre!=c)|| (cour->nombre!=i))){
5         cour=cour->suiv;
6     }
7     return (cour!=NULL);
8 }
```

```
1 void suppression_au_debut(Cellule **L){
2     Cellule *temp;
3     if (*L!=NULL){
4         temp=(*L)->suiv;
5         free(*L); /* Desallocation de la case supprimee */
6         *L=temp;
7     }
8 }
```

Quelques opérations sur les listes simplement chaînées

```
1  /* Supprime l'element (c,i) s'il existe */
2  void suppression_element(Cellule **L, char c, int i){
3      Cellule *cour;
4      Cellule *prec;
5      if (*L!=NULL){
6          cour=*L;
7          if ((cour->lettre==c) && (cour->nombre==i)){
8              suppression_au_debut(L);
9          }else{
10             prec=*L;
11             cour=cour->suiv;
12             while ((cour!=NULL) &&
13                 ((cour->lettre!=c) || (cour->nombre!=i))){
14                 prec=cour;
15                 cour=cour->suiv;
16             }
17             if (cour!=NULL){
18                 prec->suiv=cour->suiv;
19                 free(cour);
20             }
21         }
22     }
23 }
```

Quelques opérations sur les listes simplement chaînées

```
1 void insertion_au_debut(Cellule **L, char c, int i){
2     Cellule * nouv;
3     nouv=(Cellule *) malloc(sizeof(Cellule));
4     nouv->lettre=c;
5     nouv->nombre=i;
6     nouv->suiv=*L;
7     *L=nouv; /* mise a jour de la tete de liste */
8 }
```

```
1 int main(){
2     Cellule *L=NULL; /* Initialise la liste*/
3     affichage_debut(L); // Affiche "Liste vide"
4     insertion_au_debut(&L, 'A', 3);
5     insertion_au_debut(&L, 'B', 56);
6     insertion_au_debut(&L, 'C', 43);
7     affichage_debut(L); //Affiche (C,43)
8     suppression_element(&L, 'B', 56);
9     affichage_des_elements(L); //Affiche (C,43) (A,3)
10    if (!possede_element(L, 'D', 10)){
11        insertion_au_debut(&L, 'D', 10);
12    }
13    return 0;
14 }
```

Complexité des opérations de la liste simplement chaînée

Pour une liste chaînée contenant n éléments, donner la complexité des opérations suivantes :

- **Accès/Recherche** : opération consistant à accéder/rechercher un élément.
→ La recherche est en $O(n)$ car, dans le pire des cas, on parcourt toute la liste. L'accès à un élément quelconque est en $O(n)$, tout comme l'accès à l'élément fin. L'accès à l'élément début est quant à lui en $O(1)$.
- **Suppression** : opération consistant à supprimer un élément.
→ La suppression d'un élément quelconque est en $O(n)$, car il faut d'abord rechercher l'élément (en $O(n)$), puis le supprimer en "recollant" la liste (se fait en $O(1)$ car il suffit de mettre à jour un pointeur suiv). La suppression de l'élément début est en $O(1)$.
- **Insertion** : opération consistant à insérer un élément.
→ L'insertion au début de la liste se fait en $O(1)$. Si par contre l'insertion doit être réalisée en fin de liste, l'opération est en $O(n)$.

Remarque 1 : Le défaut de la liste simplement chaînée est son coût de l'accès à l'élément fin. Si cette opération est importante dans votre application, il faut opter pour une liste doublement chaînée (c'est l'implémentation choisie par C++ et java).

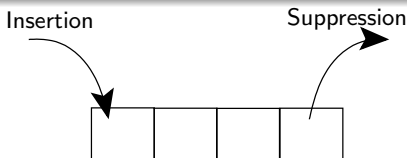
Remarque 2 : Si vous souhaitez garder un pointeur sur le dernier élément visité, il faut utiliser une liste circulaire (mais cela n'améliore pas les performances pire-cas).

Type de données abstrait : les files

Définition

Une file est un type de données abstrait représentant un ensemble de données homogènes, fondé sur le principe du “premier entré, premier sorti” (aussi appelé FIFO pour “First In, First Out”). Dans une file, les premiers éléments ajoutés à la file seront les premiers à être retirés de la file. Ce type de données permet l’insertion aisée de nouvelles données, et l’extraction aisée de la plus ancienne donnée insérée.

Opérations classiques : enfiler, défiler, etc.



Implémentation

En langage C, on peut implémenter une file par une liste (doublement) chaînée.

Nombreuses applications : mémoires tampons, “pipe” sous linux, files d’attente, compilation, algorithmes divers (exemple : parcours en largeur).

Complexité des opérations sur les files

Opérations usuelles

Les opérations usuelles sur les files sont les suivantes :

- `creer_file` : crée une liste vide.
- `est_vide` : teste si une file est vide.
- `enfiler` : insérer un élément au début de file.
- `defiler` : supprimer l'élément à la fin de la file.

Complexité de ces opérations

Avec une implémentation par liste doublement chaînée, toutes les opérations sont en $O(1)$. En effet, `enfiler` et `défiler` sont tous les deux en $O(1)$ car :

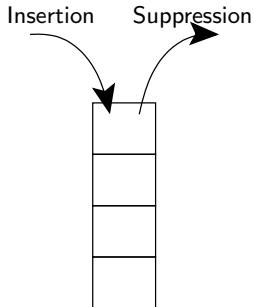
- on a un accès au premier et au dernier élément en $O(1)$.
- les fonctions *successeur* et *prédécesseur* permettent de "regrouper" deux morceaux en $O(1)$.

Type de données abstrait : les piles

Définition

Une pile est un type de données abstrait représentant un ensemble de données homogènes, fondé sur le principe du “dernier entré, premier sorti” (aussi appelé LIFO pour “Last In, First Out”). Dans une pile, les derniers éléments ajoutés à la pile seront les premiers à être retirés. Ce type de données permet l'insertion aisée de nouvelles données, et l'extraction aisée des dernières données insérées.

Opération classiques : empiler, dépiler, etc.



Implémentation

En langage C, on peut implémenter une file avec un tableau C à une dimension, ou avec une liste (simplement) chaînée.

Nombreuses applications : Pile d'exécution, “undo/redo” dans un logiciel, algorithmes divers (exemple : parcours en profondeur).

Complexité des opérations sur les piles

Opérations usuelles

Les opérations usuelles sur les piles sont les suivantes :

- `creer_pile` : crée une pile vide.
- `est_vide` : teste si une file est vide.
- `empiler` : insérer un élément au début de la file.
- `depiler` : supprimer l'élément au début de la file.

Complexité de ces opérations

Avec une implémentation par liste chaînée ou avec un tableau `C` à une dimension, toutes les opérations sont en $O(1)$. En effet, avec une liste chaînée, on peut insérer un élément en tête en $O(1)$, puis récupérer le dernier élément ajouté en accédant au début de la liste, ce qui se fait aussi en $O(1)$. Avec un tableau `C` à une dimension, il suffit d'insérer les nouveaux éléments en fin de tableau, puis récupérer le dernier élément ajouté en accédant directement à la dernière case du tableau. La différence entre les deux implémentations est uniquement sur la réservation mémoire (contiguë ou non contiguë).

Retour sur la pile d'exécution

Appel de fonction et zone mémoire

Lors d'un appel à une fonction, une zone mémoire est réservée pour effectuer cet appel. Cette zone contient les variables déclarées dans la fonction, ainsi que les variables passées en paramètres. La zone mémoire n'est libérée que lorsque l'accolade fermante ou la commande `return` sont atteintes. Si une fonction `f` appelle une fonction `g`, alors la zone mémoire de `f` reste en place pendant l'exécution de `g`, et l'exécution de `f` reprend une fois `g` terminée.

Pile d'exécution

Le stockage et l'ordre des exécutions des fonctions sont gérées par une pile que l'on appelle la pile d'exécution. Les appels s'empilent dans l'ordre de leur appel et se dépilent dans l'ordre de leur terminaison. Entre temps, la mémoire n'est donc pas libérée ! Cela peut poser problème s'il y a beaucoup d'appels imbriqués, comme c'est le cas pour des appels récursifs.

Récurtivité et zone mémoire

On appelle récurtivité le fait qu'une fonction s'appelle elle-même. La récurtivité utilise de la mémoire en grande quantité car les appels récurtifs conservent une place en mémoire jusqu'à la fin de leur exécution. Par exemple, avec la fonction suivante, les appels ne seront terminés que lorsque le cas de base est atteint :

```
1 int somme(int n){  
2   if (n==0){  
3     return 0;  
4   }else{  
5     return n+somme(n-1);  
6   }  
7 }
```

⇒ Quand cela est possible, on préfère utiliser une récurtivité terminale.

Récurtivité terminale

Une fonction est récurtivité terminale si l'appel récurtif est directement retourné par la fonction appelante. Le retour de la fonction ne doit donc pas être le résultat d'une opération utilisant le retour de l'appel récurtif. La récurtivité terminale permet de limiter les appels récurtifs qui sont stockés dans la pile d'exécution du programme.

Retour sur la fonction somme

Dans la première version, toutes les opérations “n+” sont stockées dans la pile. On dépile ces opérations une fois le cas de base atteint.

```
1 int somme(int n){  
2     if (n==0){  
3         return 0;  
4     }else{  
5         return n+somme(n-1);  
6     }  
7 }
```

Dans la deuxième version, la fonction est récursive terminale, ce qui permet de ne pas saturer la pile d'exécution du programme avec toutes ces informations.

```
1 int sommeTerminale(int n, int a){  
2     if (n==0){  
3         return a;  
4     }else{  
5         return sommeTerminale(n-1,n+a);  
6     }  
7 }
```

En plus de cette écriture terminale, il faut compiler avec l'option -O2 de gcc pour que le compilateur optimise les récursivités.