

Chapitre 6 : Structures de données

Construction et maintenance de logiciels

Guy Francoeur

basé sur du matériel d'Alexandre Blondin Massé, professeur

5 septembre 2019

UQÀM | **Département d'informatique**

Table des matières

1. Allocation dynamique

2. valgrind pour linux

3. Structures de données

Piles

Files

Tableaux dynamiques

Arbres binaires de recherche

Graphes

1. Allocation dynamique
2. valgrind pour linux
3. Structures de données

Allocation dynamique

- ▶ Lors de la déclaration d'un **tableau**, on doit fixer la **taille**;
- ▶ Ce n'est pas toujours **acceptable**, surtout lorsque la taille du tableau est très **variable**;
- ▶ La **solution** consiste à allouer l'espace mémoire à un pointeur selon les besoins.
- ▶ La fonction

```
1 void *malloc (size_t taille);
```

réserve de l'espace mémoire de **taille donnée** et retourne un **pointeur** vers le bloc de mémoire en question;

- ▶ Disponible dans la bibliothèque **stdlib.h**;
- ▶ Souvent (pour ne pas dire toujours), **malloc** est utilisée en conjonction avec l'opérateur **sizeof**.

Exemple

```
1 //ex20.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main(void) {
7     char s[] = "bonjour";
8     printf("sizeof(s) = %lu\n", sizeof(s));
9     char *t = (char*) malloc(sizeof(s));
10    t[0] = '\0';
11    printf("avant : %s\n", t);
12    strcpy(t, s);
13    printf("après : %s\n", t);
14    return 0;
15 }
```

Affiche :

avant :

après : bonjour

Autres fonctions de gestion de mémoire (1/2)

- ▶ La fonction

```
1 void *calloc(size_t nb, size_t taille);
```

alloue un **bloc de mémoire** de taille $nb \times taille$ en initialisant tous les bits à **0**;

- ▶ Si une **erreur** survient pendant une allocation de mémoire, la valeur **NULL** est retournée.

- ▶ La fonction

```
1 void free(void *p);
```

permet de **libérer** l'espace mémoire alloué pointé par **p**;

Autres fonctions de gestion de mémoire (2/2)

- ▶ La fonction

```
1 void *realloc(void *p, size_t taille);
```

permet de **redimensionner** l'espace mémoire pointé par **p** selon la nouvelle **taille donnée**.

- ▶ Parfois, les données doivent être **transférées** vers une nouvelle zone de mémoire s'il n'y a pas d'**espace suffisant**;
- ▶ Si la nouvelle taille est plus **grande** que l'ancienne, les octets supplémentaires sont **indéterminés**;
- ▶ Si la nouvelle taille est plus **petite**, les octets inclus sont **préservés**;
- ▶ La valeur **NULL** est retournée en cas d'erreur.

Fuite de mémoire

- ▶ **Memory leak** en anglais;
- ▶ Toute instruction **malloc** devrait être couplée avec une instruction **free**;
- ▶ Scénario (malloc / free)
 - ▶ dans une fonction `creerPile()` que soit alloué/réservé de la via malloc pour y mettre des infos;
 - ▶ il n'est pas possible de faire le free dans cette même fonction pour des raisons évidentes;
 - ▶ Alors il faudra prévoir une autre fonction, disons `detruirePile()`, pour faire le relâchement de la mémoire réservé;
- ▶ nous savons tous ici que `fopen()` et `fclose()` est un cas similaire.

Table des matières

1. Allocation dynamique
2. valgrind pour linux
3. Structures de données

- ▶ Est une application **Linux** qui permet de détecter des fuites de mémoire.
- ▶ Il est possible de savoir ce qui n'est pas libéré à la fin de l'exécution.
- ▶ Un exemple de son usage :

```
1 #!/bin/bash
2 #valgrind.sh
3 #pour avoir de l'aide sur valgrind :
4 valgrind --help
5 #pour lancer valgrind :
6 valgrind ./tp1
7 #ou
8 valgrind --tool=memcheck --leak-check=yes --show-reachable=
  yes --num-callers=20 --track-fds=yes ./tp1
```

Table des matières

1. Allocation dynamique

2. valgrind pour linux

3. Structures de données

Piles

Files

Tableaux dynamiques

Arbres binaires de recherche

Graphes

Définition et utilité

- ▶ Une **structure de données** est une représentation **logique** d'un ensemble de **données** qui permet de **simplifier le traitement** de celles-ci;
- ▶ La représentation à pour d'être efficace, ie., améliorer l'**efficacité** des traitements (recherche, consultation, modification, effacement, insertion, ...)
- ▶ Dans plusieurs cas, il s'agit de **collections de données**, mais représentées différemment selon les opérations qui sont les **plus fréquentes**.

Type abstrait/implémentation (1/4)

- ▶ Definition : Les données n'étant pas accessibles directement, mais par l'intermédiaire d'un pointeur passé à des fonctions, la seule information accessible à l'utilisateur est l'adresse du bloc de mémoire qui contient les données.
- ▶ Exemples de **types abstraits** :
 - ▶ Pile;
 - ▶ File;
 - ▶ Liste;
 - ▶ Ensemble;
 - ▶ Table associative;
 - ▶ Arbre;
 - ▶ Graphe.

Type abstrait/implémentation (2/4)

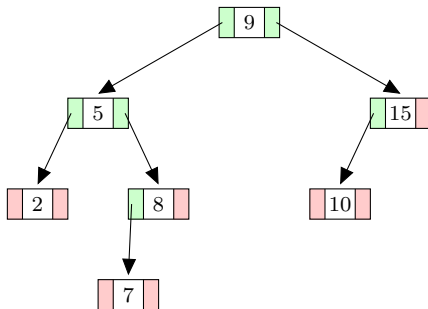
- ▶ Un type **abstrait** peut être implémenté de **différentes façons**;
- ▶ Exemples d'**implémentation** :
 - ▶ Tableau statique;
 - ▶ Tableau dynamique (redimensionnable);
 - ▶ Liste simplement chaînée;
 - ▶ Liste doublement chaînée;
 - ▶ Structure arborescente;
 - ▶ Tableau multidimensionnel;
 - ▶ Liste d'adjacence (pour les graphes), etc.

Type abstrait/implémentation (3/4)

- ▶ Il est nécessaire de définir des fonctions pour interagir avec le **type abstrait de donnée**;
- ▶ Par exemple, un **ensemble** :
 - ▶ **creerEnsemble()** (créer un ensemble vide);
 - ▶ **ajouterElement(E,e)** (ajouter l'élément e dans l'ensemble E , s'il existe déjà, l'ensemble ne change pas);
 - ▶ **supprimerElement(E,e)** (retirer l'élément e de l'ensemble E , s'il n'y est pas, l'ensemble ne change pas);
 - ▶ **union(E,F)** (faire la réunion de E et F , sans doublon);
 - ▶ etc.

Type abstrait/implémentation (4/4)

- ▶ Une **implémentation** est généralement décrite selon comment elle sera représentée **en mémoire**;
- ▶ Par exemple, un **ensemble** peut être implémenté à l'aide d'un **arbre binaire de recherche**;
- ▶ Souvent, on représente l'implémentation à l'aide d'un **dessin**.



Piles (stacks)

- ▶ Une **pile** fournit généralement les opérateurs suivants :
 - ▶ **creerPile** (créer une pile vide);
 - ▶ **estPileVide** (vérifier si une pile est vide ou pas);
 - ▶ **empiler** (ajouter un élément sur la pile);
 - ▶ **depiler** (supprimer l'élément au sommet de la pile);
 - ▶ **sommet** (consulter l'élément au sommet de la pile);
 - ▶ **detruire** (détruire une pile).
- ▶ Une **implémentation** courante est d'utiliser une **liste simplement chaînée** avec un pointeur en **tête de liste** seulement;

Structures en C

```
1 //stack.h
2 #ifndef STACK_H
3 #define STACK_H
4 // Types
5
6 struct StackNode {
7     char content; // Contenu du noeud
8     struct StackNode *next; // Noeud suivant
9 };
10
11 typedef struct {
12     struct StackNode *first; // Pointeur vers le premier noeud
13     int size; // Nombre d'elements dans la pile
14 } Stack;
15
16 // Prototypes
17
18 Stack stackCreate();
19 bool stackIsEmpty(const Stack *s);
20 void stackPush(Stack *s, char content);
21 char stackPop(Stack *s);
22 void stackDelete(Stack *s);
23
24 #endif
```

Voir fichier [stack.c](#)

- ▶ Une file fournit généralement les **opérateurs** suivants :
 - ▶ **creerFile** (créer une file vide);
 - ▶ **estFileVide** (vérifier si une file est vide ou pas);
 - ▶ **enfiler** (ajouter un élément en fin de file);
 - ▶ **defiler** (supprimer l'élément en début de file);
 - ▶ **premier** (consulter le premier élément de la file);
 - ▶ **detruireFile** (détruire la file);
- ▶ Une **file** peut être implémentée à l'aide d'une **liste simplement chaînée**, d'un pointeur vers la **tête** de la liste, ainsi que d'un pointeur en **fin** de liste.
- ▶ L'implémentation d'une **file** est laissée en exercice.

Structures en C

```
1 //queue.h
2 #ifndef QUEUE_H
3 #define QUEUE_H
4 // Types
5 struct QueueNode {
6     char content; // Contenu du noeud
7     struct QueueNode *prev; // Noeud precedent
8     struct QueueNode *next; // Noeud suivant
9 };
10
11 typedef struct {
12     struct QueueNode *first; // Pointeur vers le premier noeud
13     struct QueueNode *last; // Pointeur vers le dernier noeud
14 } Queue;
15
16 // Prototypes
17 Queue queueCreate();
18 bool queueIsEmpty(const Queue *s);
19 void queuePush(Queue *s, char content);
20 char queuePop(Queue *s);
21 void queueDelete(Queue *s);
22 #endif
```

Tableaux dynamiques

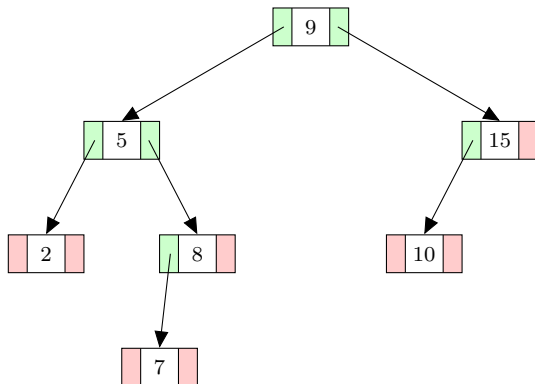
- ▶ Appelés **vecteurs** en C++;
- ▶ Essentiellement, ils offrent les **mêmes opérations** que les tableaux, mais il est possible de les **redimensionner** s'il n'y a plus assez d'espace;
- ▶ Les **opérations** suivantes sont généralement offertes :
 - ▶ **Insertion** en fin de tableau;
 - ▶ **Suppression** d'un élément en position i ;
 - ▶ **Recherche** d'un élément;
 - ▶ **Accès** à l'élément en position i .
- ▶ Avantages/inconvénients si on choisit une **liste dynamique** ou un **tableau dynamique** ?

Structures en C

```
1 //array.h
2 #ifndef ARRAY_H
3 #define ARRAY_H
4     // Types
5
6     typedef struct {
7         int *values;      // Les valeurs dans le tableau
8         int currentSize;  // Le nombre d'lements dans le tableau
9         int capacity;     // Capacite du tableau
10    } Array;
11
12    // Prototypes
13
14    Array arrayCreate();
15    void arrayInsert(Array *a, int element);
16    void arrayRemove(Array *a, int i);
17    bool arrayHasElement(const Array *a, int element);
18    int arrayGet(const Array *a, int i);
19    void arrayDelete(Array *a);
20 #endif
```

Voir fichier `array.c`

Définition



Invariant : Pour toute paire de noeuds x et y ,

- ▶ si y est à **gauche** de x , alors $x.\text{element} \geq y.\text{element}$;
- ▶ si y est à **droite** de x , alors $x.\text{element} \leq y.\text{element}$.

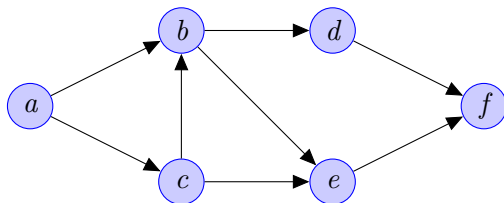
Implémentation

- ▶ Permettent d'**implémenter** différents **types abstraits** : ensembles, table associative (*map*), liste triée, etc.
- ▶ Le cas le **moins favorable** est obtenu lorsque l'arbre est en fait une **liste chaînée** (arbre dégénéré);
- ▶ Il existe des techniques permettant d'avoir des arbres **équilibrés** (AVL, rouge-noir, arbre 2-3-4, etc.);
- ▶ Combien coûtent les opérations suivantes au **pire cas** ?

	Non équilibré	Équilibré
Insertion		
Suppression		
Recherche		

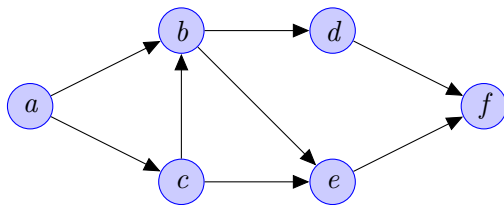
Exemple : table associative (*TreeMap*)

```
1 //treemap.h
2 #ifndef TREEMAP_H
3 #define TREEMAP_H
4     struct TreeNode {
5         char *key;           // Cle du noeud
6         char *value;         // Valeur associee
7         struct TreeNode *left; // Fils gauche
8         struct TreeNode *right; // Fils droit
9     };
10
11     typedef struct { struct TreeNode *root; } TreeMap;
12
13     TreeMap tmCreate();
14     char *tmGet(const TreeMap *t, char *key);
15     void tmSet(TreeMap *t, char *key, char *value);
16     void tmInsert(struct TreeNode **node, char *key, char *value);
17     bool tmHasKey(const TreeMap *t, char *key);
18     struct TreeNode *tmGetNode(const TreeMap *t, char *key);
19     void tmPrint(const TreeMap *t);
20     void tmPrintRecursive(const struct TreeNode *node);
21     void tmDelete(TreeMap *t);
22     void tmDeleteRecursive(struct TreeNode *node);
23 #endif
```



- ▶ Un **graphe** est un couple $G = (V, E)$ où V est un ensemble de **sommets** et E est un ensemble d'**arêtes** (ou d'**arcs**);
- ▶ Nombreuses variantes : **graphe orientés**, **graphes non orientés**, **multigraphes**, etc.
- ▶ Permettent de modéliser de **nombreux problèmes** : **réseaux**, **automates**, **relations**, etc.
- ▶ On distingue **deux types** d'implémentation des graphes.

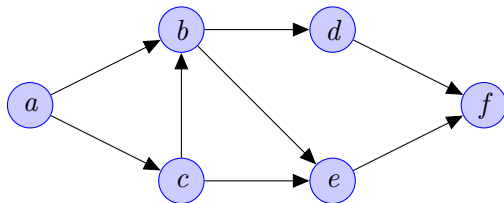
Graphes denses



- Si le graphe est **dense**, c'est-à-dire que chaque sommet est connecté à presque tous les autres sommets, alors on utilise en général une **matrice d'adjacence** :

$$\begin{array}{c} \begin{array}{cccccc} & a & b & c & d & e & f \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \left(\begin{array}{cccccc} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right)$$

Graphes épars



- Autrement, on utilise une **liste d'adjacence** ou **matrice creuse** :

Sommet	Successeur(s)
<i>a</i>	<i>b, c</i>
<i>b</i>	<i>d, e</i>
<i>c</i>	<i>b, e</i>
<i>d</i>	<i>f</i>
<i>e</i>	<i>f</i>
<i>f</i>	—

- ▶ Vous en saurez plus sur les **structures de données** dans le cours **INF3105 Structures de données et algorithmes** :
 - ▶ Arbres binaires de recherche,
 - ▶ Arbres équilibrés,
 - ▶ Monceaux (*heaps*),
 - ▶ Graphes, etc.
- ▶ D'autres structures **plus avancées** :
 - ▶ Ensembles disjoints,
 - ▶ Listes à saut (*skip lists*),
 - ▶ Tas binomiaux, de Fibonacci, etc.