



Programmation Impérative 2

Entrées/sorties en C

Licence 2 Informatique

Antoine Spicher

`antoine.spicher@u-pec.fr`

Plan du cours : E/S en C



- Manipulation des fichiers
- Chaînes de caractères
- Spécification de fonctions d'E/S

Manipulation des fichiers

■ Motivations

- Sauvegarder des données/l'état d'un programme
- Charger des données/l'état d'un programme
- Communiquer avec d'autres programmes
- Communiquer avec différents périphériques

■ Fichiers

- Définition
 - « *flot d'informations* accessible à partir d'un nom, ou *chemin d'accès* »
- Sous les systèmes UNIX, *tout est fichier* (cf. cours de système)
 - Réguliers (stockés dans une mémoire de masse : disque dur, clé USB, CD-ROM)
 - Sockets (accès réseau)
 - Périphériques (e.g., imprimante)
- Opérations (suivant les droits) : `#include <stdio.h>`
 - Ouvrir, fermer, lire, écrire et se déplacer

Manipulation des fichiers

■ Descripteur de fichier

- Représentation d'un **fichier ouvert** : `FILE*`
- Spécification du type `FILE` : *il n'est pas nécessaire de la connaître !*
 - Dépend des systèmes d'exploitation
 - Contient les informations sur l'ouverture actuelle (position, droits, etc.)
 - Équivalent à un *curseur*

■ Descripteurs de fichier prédéfinis

- **Entrée standard** : `stdin` (***standard input***)
D'où viennent les données reçues par `scanf` (généralement le clavier)
- **Sortie standard** : `stdout` (***standard output***)
Où vont les données envoyées par `printf` (généralement l'écran)
- **Sortie erreur** : `stderr` (***standard error***)
Où sont envoyées les erreurs (généralement la même chose que `stdout`)

Manipulation des fichiers

■ Utilisation type d'un fichier

1. Ouverture

`fopen`

2. Actions

■ Écrire :

`fwrite, fputc, putchar, fputs, puts, fprintf, printf`

■ Lire :

`fread, fgetc, getchar, fgets, gets, fscanf, scanf`

■ S'informer sur/positionner le curseur :

`feof, fseek, ftell, fgetpos, fsetpos, rewind, fflush`

3. Fermeture

`fclose`

Manipulation des fichiers

■ Ouverture d'un fichier

□ Prototype

```
#include <stdio.h>

FILE* fopen(const char* fname, const char* mode);
```

□ Spécification (> **man fopen**)

- Ouvre un descripteur vers le fichier de chemin d'accès fname
- Retourne NULL si l'ouverture échoue
- Le paramètre mode précise le type d'accès demandé
 - mode = "r" : le fichier est ouvert en lecture seule
 - mode = "w" : le fichier est ouvert en écriture seule depuis le début (l'ancien contenu est écrasé)
 - mode = "a" : le fichier est ouvert en écriture seule à la fin
 - mode = "...+" : le fichier est ouvert en lecture et écriture en mode "..."
 - mode = "...b" : le fichier est ouvert en mode binaire

Manipulation des fichiers

■ Fermeture d'un fichier

□ Prototype

```
#include <stdio.h>

int fclose(FILE* fdesc);
```

□ Spécification (> **man fclose**)

- Ferme le descripteur de fichier fdesc
- Retourne 0 en cas de succès et EOF en cas d'échec
- Appelle `fflush(fdesc)` (termine les écritures en cours) avant la fermeture

```
FILE* fdesc = fopen(fname, "rb+");
if (fdesc == NULL) {
    fprintf(stderr, "cannot open file %s in mode rb+\n", fname);
    exit(1);
}
... /* actions */
fclose();
```

*A chaque **fopen** doit correspondre un **fclose***

Manipulation des fichiers

■ Écriture dans un fichier

□ Prototype

```
#include <stdio.h>

size_t fwrite(void* ptr, size_t size, size_t nmemb, FILE* fdesc);
```

□ Spécification (> **man fwrite**)

- Écrit nmemb éléments (chacun de taille size) à partir de l'adresse ptr dans le fichier décrit par fdesc
- Décale d'autant le curseur dans le fichier
- Retourne le nombre d'éléments *effectivement* écrits
- Permet la spécification de toutes les fonctions d'écriture
- Voir son utilisation dans la 3^{ème} partie de ce cours

Manipulation des fichiers

■ Écriture d'un caractère dans un fichier

□ Prototype

```
#include <stdio.h>

int fputc(int ch, FILE* fdesc);
int putchar(int ch);
```

□ Spécification (> **man fputc**)

- Convertit ch d'int à unsigned char et l'écrit dans le fichier décrit par fdesc
- Décale d'un caractère le curseur dans le fichier
- Retourne ch en cas de succès et EOF en cas d'échec

```
fputc('x', fdesc);
```



```
char c = 'x';
fwrite(&c, sizeof(char), 1, fdesc);
```

```
putchar('x');
```



```
fputc('x', stdout);
```

Manipulation des fichiers

■ Écriture d'une chaîne de caractères dans un fichier

□ Prototype

```
#include <stdio.h>

int fputs(char* s, FILE* fdesc);
int puts(char* s);
```

□ Spécification (> **man fputs**)

- Écrit la chaîne s (sans le caractère `'\0'`) dans le fichier décrit par fdesc
- Décale d'autant de caractères le curseur dans le fichier
- Retourne un nombre non négatif en cas de succès et EOF en cas d'échec

```
fputs("toto", fdesc);
```



```
char* s = "toto";
fwrite(s, sizeof(char), 4, fdesc);
```

```
puts("toto");
```



```
fputs("toto", stdout);
```

Manipulation des fichiers

■ Écriture d'une chaîne de caractères formatée

□ Prototype

```
#include <stdio.h>

int fprintf(FILE* fdesc, char* fmt, ...);
int printf(char* fmt, ...);
```

□ Spécification (> **man fprintf**)

- Génère une chaîne de caractères formatée à partir du format fmt et des paramètres supplémentaires, puis l'écrit dans le fichier décrit par fdesc
- Décale d'autant de caractères le curseur dans le fichier
- Retourne le nombre de caractères écrits en cas de succès, un nombre négatif en cas d'échec
- Voir la spécification du format et des arguments dans la 2^{ème} partie du cours

```
printf("%d\n", 3);
```



```
fprintf(stdout, "%d\n", 3);
```

Manipulation des fichiers

■ Lecture depuis un fichier

□ Prototype

```
#include <stdio.h>

size_t fread(void* ptr, size_t size, size_t nmemb, FILE* fdesc);
```

□ Spécification (> **man fread**)

- Lit nmemb éléments (chacun de taille size) depuis le fichier décrit par fdesc et les stocke à partir de l'adresse ptr
- Décale d'autant le curseur dans le fichier
- Retourne le nombre d'éléments effectivement lus
- Permet la spécification de toutes les fonctions de lecture
- Voir son utilisation dans la 3^{ème} partie de ce cours

Manipulation des fichiers

■ Lecture d'un caractère depuis un fichier

□ Prototype

```
#include <stdio.h>

int fgetc(FILE* fdesc);
int getchar(void);
```

□ Spécification (> **man fgetc**)

- Lit puis convertit en `int` un caractère depuis le fichier décrit par `fdesc`
- Décale d'un caractère le curseur dans le fichier
- Retourne `ch` en cas de succès et `EOF` en cas d'échec

```
char c = fgetc(fdesc);
```



```
char c;
fread(&c, sizeof(char), 1, fdesc);
```

```
x = getchar();
```



```
x = fgetc(stdin);
```

Manipulation des fichiers

■ Lecture d'une chaîne de caractères dans un fichier

□ Prototype

```
#include <stdio.h>

char* fgets(char* s, int size, FILE* fdesc);
char* gets(char* s);
```

□ Spécification (> **man fgets**)

- Lit au plus size-1 caractères depuis le fichier décrit par fdesc et les stocke dans la chaîne s et place un `'\0'` en fin de chaîne
- S'arrête de lire en fin de fichier ou à un passage à la ligne
- Décale d'autant de caractères le curseur dans le fichier
- Retourne s en cas de succès et `NULL` en cas d'échec
- `gets` n'a pas de limite de taille

Manipulation des fichiers

■ Lecture d'une chaîne de caractères formatée

□ Prototype

```
#include <stdio.h>

int fscanf(FILE* fdesc, char* fmt, ...);
int scanf(char* fmt, ...);
```

□ Spécification (> **man fscanf**)

- Analyse le fichier décrit par fdesc à partir du format fmt et récupère les valeurs à placer dans les paramètres supplémentaires
- Décale d'autant de caractères le curseur dans le fichier
- Retourne le nombre d'entrées correctement lues ou EOF en cas d'échec
- Voir la spécification du format et des arguments dans la 2^{ème} partie du cours

```
scanf("%d", &i);
```



```
fscanf(stdin, "%d", &i);
```

Manipulation des fichiers

■ Fin de fichier (*End Of File*)

□ Prototype

```
#include <stdio.h>

int feof(FILE* fdesc);
```

□ Spécification (> **man feof**)

- Retourne si la fin de fichier a été atteinte

Manipulation des fichiers

■ Position du curseur

□ Prototype

```
#include <stdio.h>

long ftell(FILE* fdesc);
int fgetpos(FILE* fdesc, fpos_t* pos);
```

□ Spécification (> **man ftell**)

- Retourne la position actuelle du curseur dans le fichier
- `fgetpos` fait la même chose mais est plus portable (utilisation du type `fpos_t`) ; il retourne 0 en cas de succès, -1 sinon

Manipulation des fichiers

■ Modification de la position du curseur

□ Prototype

```
#include <stdio.h>

int fseek(FILE* fdesc, long offset, int whence);
int fsetpos(FILE* fdesc, fpos_t* pos);
```

□ Spécification (> **man fseek**)

- Déplace le curseurs de offset octets à partir de la position indiquée par
 - whence = SEEK_SET : début de fichier
 - whence = SEEK_CUR : position courante du curseur
 - whence = SEEK_END : fin de fichier
- Retourne 0 en cas de succès, -1 sinon
- fsetpos fait la même chose mais est plus portable (utilisation du type `fpos_t`) ; il retourne 0 en cas de succès, -1 sinon

Manipulation des fichiers

■ *Be Kind Rewind*

□ Prototype

```
#include <stdio.h>

void rewind(FILE* fdesc);
```

□ Spécification (> **man rewind**)

- Place le curseur en début de fichier

Manipulation des fichiers

■ Forcer l'écriture

□ Prototype

```
#include <stdio.h>

int fflush(FILE* fdesc);
```

□ Spécification (> **man fflush**)

- Force l'écriture de toutes les données mises en tampon
- Renvoie 0 en cas de succès et `'EOF'` en cas d'échec
- Permet notamment de s'assurer qu'un affichage à l'écran a été fait au bon moment (pratique pour debugger)

Plan du cours : E/S en C



- Manipulation des fichiers
- Chaînes de caractères
- Spécification de fonctions d'E/S

Chaînes de caractères

■ Définition

- Tableau de caractères se terminant par le caractère spécial `'\0'`

```
char string[11] = "Helloworld";
```

- Pointeur vers le premier caractère

```
char* string = "Helloworld";
```



■ Manipulation des chaînes

- Opérations classiques : `#include <string.h>`

`strlen, str(n)cpy, str(n)cat, str(n)cmp, strchr, strrchr, strstr`

- Formater les chaînes : `#include <stdio.h>`

`printf, fprintf, sprintf, asprintf (GNU uniquement),
scanf, fscanf, sscanf`

Chaînes de caractères

■ Opérations classiques

□ `size_t strlen(const char* s)`

Calcule la longueur de la chaîne s

□ `char* strcpy(char* dst, const char* src)`

Copie la chaîne src à l'emplacement pointé par dst (une place suffisante aura due être allouée)

□ `size_t strncpy(char* dst, const char* src, size_t n)`

Similaire à `strcpy` mais copie au plus n caractères

□ `char* strcat(char* dst, const char* src)`

Concatène (copie) la chaîne src à la suite de la chaîne dst (une place suffisante aura due être allouée)

□ `size_t strncat(char* dst, const char* src, size_t n)`

Similaire à `strcat` mais concatène au plus n caractères

Chaînes de caractères

■ Opérations classiques

□ `int strcmp(const char* s1, const char* s2)`

Compare les chaînes de caractères en utilisant l'ordre des entiers (qui coïncide avec l'ordre lexicographique dans le code ASCII) ; retourne un entier négatif, nul ou positif si s1 est respectivement inférieur, égal ou supérieur à s2

□ `int strncmp(const char* s1, const char* s2, size_t n)`

Similaire à `strcmp` mais compare au plus n caractères

□ `char* strchr(const char* s, int c)`

Recherche la première occurrence du caractère c dans la chaîne s ; renvoie le pointeur vers ce caractère dans s s'il existe, `NULL` sinon

□ `char* strrchr(const char* s, int c)`

Similaire à `strchr` mais retourne un pointeur vers la dernière occurrence de c

□ `char* strstr(const char* s1, const char* s2)`

Recherche la première occurrence de la sous-chaîne s2 dans la chaîne s1 ; renvoie le pointeur vers cette occurrence si elle existe, `NULL` sinon

Chaînes de caractères

■ Formater les chaînes

□ `int printf(const char* fmt, ...)`

Écrit la chaîne de caractères spécifiée par fmt et les arguments supplémentaires dans la sortie standard ; retourne le nombre de caractères écrits

□ `int fprintf(FILE* fdesc, const char* fmt, ...)`

Similaire à `printf` mais écrit dans le fichier décrit par fdesc

□ `int sprintf(char* str, const char* fmt, ...)`

Similaire à `printf` mais écrit dans à l'adresse pointée par str (la chaîne doit avoir été allouée au préalable)

□ `int asprintf(char** strp, const char* fmt, ...)`

Similaire à `sprintf` mais alloue une chaîne de caractères de taille suffisante pour contenir la sortie (ne pas oublier de faire libérer la mémoire avec `free`) et stocke le pointeur de cette chaîne à l'adresse pointée par strp ; cette fonction est une extension GNU

Chaînes de caractères

■ Formater les chaînes

- `int scanf(const char* fmt, ...)`

Analyse l'entrée standard en utilisant la spécification fmt et stocke le résultat dans aux adresses données en arguments supplémentaires

- `int fscanf(FILE* fdesc, const char* fmt, ...)`

Similaire à `scanf` mais analyse le fichier décrit par fdesc


- `int sscanf(char* str, const char* fmt, ...)`

Similaire à `printf` mais analyse la chaîne de caractères pointée par str


Chaînes de caractères

■ Formater les chaînes

- Utilisation du format et des arguments supplémentaires



```
printf("%d %f %s\n", n, x, str);
```



```
scanf("%d %f %s", &n, &x, str);
```

- Spécification des types

Specifiers	printf arg. type	scanf arg. type
"%c"	char	char*
"%d"	int	int*
"%f"	double	double*
"%p"	void*	void**
"%s"	char*	char*

cf. [printf specifiers](#) et [scanf specifiers](#)

Plan du cours : E/S en C



- Manipulation des fichiers
- Cas particulier des chaînes de caractères
- Spécification de fonctions d'E/S

Spécification de fonctions d'E/S

■ Motivation

- *Sauvegarde* des données dans un fichier
- *Chargement* des données depuis un fichier

■ Choix du *format* de sauvegarde

- Formats *textes* (e.g. XML)
Simples à vérifier ou à modifier à la main, portables, facilement adaptables
- Formats *binaires*
Plus compacts, facilement chargeables, moins exigeant ressources processeurs

■ Sérialisation/désérialisation

- « *Processus de conversion d'une structure de données ou de l'état d'un objet dans un format *linéaire* et qui peut être *reconstruit* plus tard dans le même environnement ou dans un autre environnement.* »
- En anglais : *(un)marshaling, (de)serialization, deflating/inflating*

Spécification de fonctions d'E/S

■ Exemples de spécifications

- Utilisation pour les données simples des fonctions **fwrite** et **fread**
- Spécification d'une **sauvegarde** : `bool save_t(t elt, FILE* fdesc)`
 - elt : l'élément de type `t` à *sauvegarder*
 - fdesc : descripteur du fichier *ouvert en écriture* où sauvegarder
 - Retourne *si la sauvegarde s'est effectuée correctement*
- Spécification d'un **chargement** : `bool load_t(t *eltp, FILE* fdesc)`
 - eltp : pointeur vers un élément *déjà alloué* de type `t` où *charger*
 - fdesc : descripteur du fichier *ouvert en lecture* où charger
 - Retourne *si le chargement s'est effectué correctement*
- Algorithmique
 - Effet *miroir* entre `save_t` et `load_t`
 - Dépendance à la nature du type `t`
Type simple, tableau, structure, « graphe »

Spécification de fonctions d'E/S

■ E/S des types *simples*

- Type *simple* : `char`, `short`, `int`, `long`, `float`, `double`, `long double`, ...
On peut *utiliser directement* les fonctions **fwrite** et **fread**

- Fonction de sauvegarde

```
bool save_t(t elt, FILE* fdesc) {  
    int n = fwrite(&elt, sizeof(t), 1, fdesc);  
    return (n==1);  
}
```

- Fonction de chargement

```
bool load_t(t *eltp, FILE* fdesc) {  
    int n = fread(eltp, sizeof(t), 1, fdesc);  
    return (n==1);  
}
```

miroir

On ne prend pas compte dans ce cours certaines subtilités (e.g., *big endian* vs. *little endian*)

Spécification de fonctions d'E/S

■ E/S des types *simples*

□ Exemples

```
bool save_int(int i, FILE* fdesc) {  
    return (fwrite(&i, sizeof(int), 1, fdesc)==1);  
}
```

```
bool load_int(int *ip, FILE* fdesc) {  
    return (fread(ip, sizeof(int), 1, fdesc)==1);  
}
```

```
bool save_char(char c, FILE* fdesc) {  
    return (fwrite(&c, sizeof(char), 1, fdesc)==1);  
}
```

```
bool load_char(char *cp, FILE* fdesc) {  
    return (fread(cp, sizeof(char), 1, fdesc)==1);  
}
```


Spécification de fonctions d'E/S

■ E/S de tableaux *simples*

- Tableau simple de type t^* : t est un *type simple*

On peut *utiliser directement* les fonctions **fwrite** et **fread**

- Fonction de sauvegarde

```
bool save_t_tab(t* tab, int size, FILE* fdesc) {  
    if (!save_int(size, fdesc)) return false;  
    return (size == fwrite(tab, sizeof(t), size, fdesc));  
}
```

- Fonction de chargement

```
bool load_t_tab(t* *tabp, int *sizep, FILE* fdesc) {  
    if (!load_int(sizep, fdesc)) return false;  
    *tabp = (t*)malloc(*sizep * sizeof(t));  
    if (*tabp == NULL) return false;  
    return (*sizep == fread(*tabp, sizeof(t), *sizep, fdesc));  
}
```

miroir

Spécification de fonctions d'E/S

■ E/S de tableaux *simples*

□ Exemple : E/S d'un tableau de flottants

```
bool save_float_tab(float* tab, int size, FILE* fdesc) {  
    if (!save_int(size, fdesc)) return false;  
    return (size == fwrite(tab, sizeof(float), size, fdesc));  
}
```

```
bool load_float_tab(float* *tabp, int *sizep, FILE* fdesc){  
    if (!load_int(sizep, fdesc)) return false;  
    if ((*tabp = (float*)malloc(*sizep * sizeof(float))) == NULL)  
        return false;  
    return (*sizep == fread(*tabp, sizeof(float), *sizep, fdesc));  
}
```

Spécification de fonctions d'E/S

■ E/S de tableaux *simples*

□ Exemple : cas particulier des chaînes de caractères

```
bool save_string(char* s, FILE* fdesc) {  
    int size = strlen(s);  
    if (!save_int(size, fdesc)) return false;  
    return (size == fwrite(s, sizeof(char), size, fdesc));  
}
```

```
bool load_string(char* *sp, FILE* fdesc){  
    int size;  
    if (!load_int(&size, fdesc)) return false;  
    if ((*sp = (char*)malloc((size+1) * sizeof(char))) == NULL)  
        return false;  
    *sp[size] = '\\0';  
    return (size == fread(*sp, sizeof(char), size, fdesc));  
}
```

Spécification de fonctions d'E/S

■ E/S de structures

- Attention alignement possible

On *n'utilise pas* les fonctions **fwrite** et **fread**

- Fonction de sauvegarde

```
struct t {  
    t1 field1;  
    ...  
    tn fieldn;  
}
```

```
bool save_struct_t(struct t st, FILE* fdesc) {  
    if (!save_t1(st.field1, fdesc)) return false;  
    ...  
    if (!save_tn(st.fieldn, fdesc)) return false;  
    return true;  
}
```

- Fonction de chargement

```
bool load_struct_t(struct t *stp, FILE* fdesc) {  
    if (!load_t1(&((*stp).field1), fdesc)) return false;  
    ...  
    if (!load_tn(&((*stp).fieldn), fdesc)) return false;  
    return true;  
}
```

miroir

Spécification de fonctions d'E/S

■ E/S de structures

- Exemple : E/S d'une structure représentant une date

```
struct date {  
    char    day;  
    char*   month;  
    int     year;  
}
```

```
bool save_date(date d, FILE* fdesc) {  
    if (!save_char(d.day, fdesc)) return false;  
    if (!save_string(d.month, fdesc)) return false;  
    if (!save_int(d.year, fdesc)) return false;  
    return true;  
}
```

```
bool load_date(date *dp, FILE* fdesc) {  
    if (!load_char(&(dp->day), fdesc)) return false;  
    if (!load_string(&(dp->month), fdesc)) return false;  
    if (!load_int(&(dp->year), fdesc)) return false;  
    return true;  
}
```

Spécification de fonctions d'E/S

■ E/S de tableaux *quelconques*

- Tableau de type t^* : t est *un type quelconque*
- Fonctions de sauvegarde et de chargement

```
bool save_t_tab(t* tab, int size, FILE* fdesc) {  
    int i;  
    if (!save_int(size, fdesc)) return false;  
    for(i=0; i<size; i++)  
        if (!save_t(tab[i], fdesc)) return false;  
    return true;  
}
```

```
bool load_t_tab(t* *tabp, int *sizep, FILE* fdesc) {  
    int i;  
    if (!load_int(sizep, fdesc)) return false;  
    *tabp = (t*)malloc(*sizep * sizeof(t));  
    if (*tabp == NULL) return false;  
    for(i=0; i<*sizep; i++)  
        if (!load_t(&((*tabp)[i]), fdesc)) return false;  
    return true;  
}
```

miroir

Spécification de fonctions d'E/S

■ E/S de tableaux *quelconques*

□ Exemple : E/S d'un tableau de dates

```
bool save_date_tab(date* tab, int size, FILE* fdesc) {
    int i;
    if (!save_int(size, fdesc)) return false;
    for(i=0; i<size; i++)
        if (!save_date(tab[i], fdesc)) return false;
    return true;
}
```

```
bool load_date_tab(date* *tabp, int *sizep, FILE* fdesc) {
    int i;
    if (!load_int(sizep, fdesc)) return false;
    *tabp = (date*)malloc(*sizep * sizeof(date));
    if (*tabp == NULL) return false;
    for(i=0; i<*sizep; i++)
        if (!load_date(&((*tabp)[i]), fdesc)) return false;
    return true;
}
```

Spécification de fonctions d'E/S

■ E/S de pointeurs

- Information **propre à une exécution**

Il ne faut pas conserver les pointeurs dans une sauvegarde

- **Association** des pointeurs à des entiers

- Utilisation d'un **dictionnaire** pour l'association (entier \Leftrightarrow pointeur)

- Type représentant un dictionnaire : `map`

- Création d'un dictionnaire contenant le couple (0 \Leftrightarrow NULL)

```
map new_map()
```

- Destruction d'un dictionnaire

```
void delete_map(map m)
```

- Récupération d'un identifiant connaissant le pointeur associé

```
int get_id(map m, void* ptr)
```

en cas d'absence, une association est automatiquement créée

- Récupération d'un pointeur connaissant l'identifiant associé

```
void* get_ptr(map m, int id, size_t sz)
```

en cas d'absence, la mémoire est automatiquement allouée et le dico est m à j

Spécification de fonctions d'E/S

■ E/S de pointeurs

□ Fonction de sauvegarde d'un pointeur

```
bool save_t_ptr(t* ptr, map m, FILE* fdesc) {  
    int id = get_id(m, (void*)ptr);  
    return save_int(id, fdesc);  
}
```

□ Fonction de chargement d'un pointeur

```
bool load_t_ptr(t* *ptrp, map m, FILE* fdesc) {  
    int id;  
    if (!load_int(&id, fdesc)) return false;  
    *ptrp = (t*)get_ptr(m, id, sizeof(t));  
    return true;  
}
```

miroir

Spécification de fonctions d'E/S

■ E/S de pointeurs

□ Fonction de sauvegarde d'une valeur pointée

```
bool save_pointed_t(t* ptr, map m, FILE* fdesc) {  
    if (!save_t_ptr(ptr, m, fdesc)) return false;  
    if (!save_t(*ptr, fdesc)) return false;  
    return true;  
}
```

□ Fonction de chargement d'une valeur pointée

```
bool load_pointed_t(t* *ptrp, map m, FILE* fdesc) {  
    if (!load_t_ptr(ptrp, m, fdesc)) return false;  
    if (!load_t(*ptrp, fdesc)) return false;  
    return true;  
}
```

miroir

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Définition des types

```
typedef struct node_ {  
    char*          label;    // label of a node  
    int            nb_succ;   // number of successors  
    struct node_** succ;     // array of pointers to the successors  
    int            nb_pred;   // number of predecessors  
    struct node_** pred;     // array of pointers to the predecessors  
} node;
```

```
typedef struct graph_ {  
    int            nb_node;   // number of nodes  
    node**         nodes;     // array of pointers to the nodes  
} graph;
```

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Application de *E/S d'une structure*

```
bool save_graph(graph g, FILE* fdesc) {  
    map m = new_map();  
    bool ret = save_pointed_node_tab(g.nodes, g.nb_node, m, fdesc);  
    delete_map(m);  
    return ret;  
}
```

```
bool load_graph(graph *gp, FILE* fdesc) {  
    map m = new_map();  
    bool ret = load_pointed_node_tab(&(gp->nodes),  
                                     &(gp->nb_node), m, fdesc);  
    delete_map(m);  
    return ret;  
}
```

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Application de *E/S d'un tableau quelconque*

```
bool save_pointed_node_tab(node** t, int sz, map m, FILE* fdesc){
    int i;
    if (!save_int(sz, fdesc)) return false;
    for (i=0; i<sz; i++)
        if (!save_pointed_node(t[i], m, fdesc)) return false;
    return true;
}
```

```
bool load_pointed_node_tab(node** *tp, int *szp, map m, FILE* fdesc){
    int i;
    if (!load_int(szp, fdesc)) return false;
    *tabp = (node**)malloc(*szp * sizeof(node*));
    if (*tabp == NULL) return false;
    for (i=0; i<*szp; i++)
        if (!load_pointed_node(&((*tabp)[i]), m, fdesc)) return false;
    return true;
}
```

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Application de *E/S d'une valeur pointée*

```
bool save_pointed_node(node* p, map m, FILE* fdesc) {  
    if (!save_node_ptr(p, m, fdesc)) return false;  
    if (!save_node(*p, m, fdesc)) return false;  
    return true;  
}
```

```
bool load_pointed_node(node* *pp, map m, FILE* fdesc) {  
    if (!load_node_ptr(pp, m, fdesc)) return false;  
    if (!load_node(*p, m, fdesc)) return false;  
    return true;  
}
```

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Application de *E/S d'un pointeur*

```
bool save_node_ptr(node* p, map m, FILE* fdesc) {  
    int id = get_id(m, (void*)p);  
    return save_int(id, fdesc);  
}
```

```
bool load_node_ptr(node* *pp, map m, FILE* fdesc) {  
    int id;  
    if (!load_int(&id, fdesc)) return false;  
    *pp = (node*)get_ptr(m, id, sizeof(node));  
    return true;  
}
```

Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

□ Application de *E/S d'une structure*

```
bool save_node(node n, map m, FILE* fdesc) {  
    if (!save_string(n.label, fdesc)) return false;  
    if (!save_node_ptr_tab(n.succ, n.nb_succ, m, fdesc)) return false;  
    if (!save_node_ptr_tab(n.pred, n.nb_pred, m, fdesc)) return false;  
    return true;  
}
```

```
bool load_node(node *np, map m, FILE* fdesc) {  
    if (!load_string(&(np->label), fdesc)) return false;  
    if (!load_node_ptr_tab(&(np->succ), &(np->nb_succ), m, fdesc))  
        return false;  
    if (!load_node_ptr_tab(&(np->pred), &(np->nb_pred), m, fdesc))  
        return false;  
    return true;  
}
```


Spécification de fonctions d'E/S

■ Exemple : E/S d'un graphe

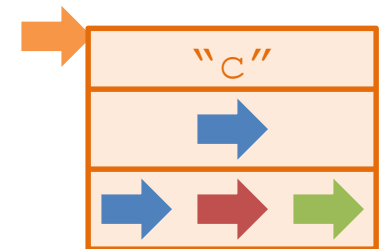
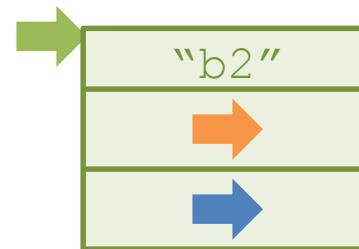
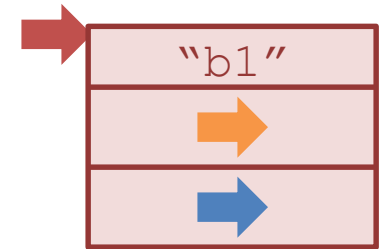
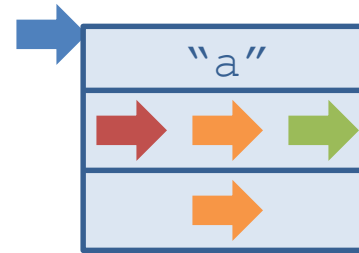
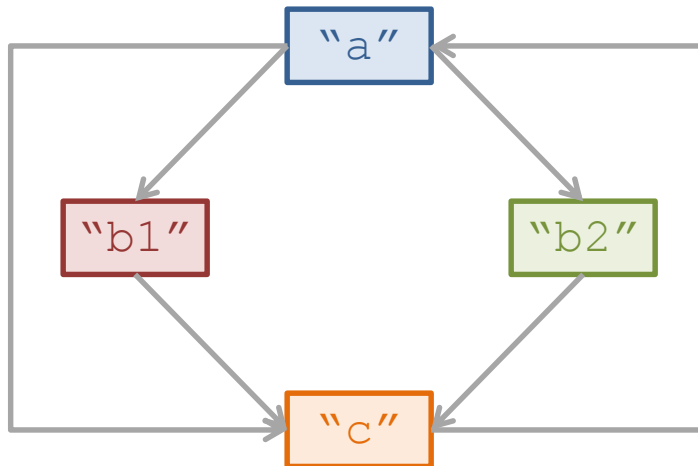
□ Application de *E/S d'une tableau quelconque*

```
bool save_node_ptr_tab(node** t, int sz, map m, FILE* fdesc) {
    int i;
    if (!save_int(sz, fdesc)) return false;
    for (i=0; i<sz; i++)
        if (!save_node_ptr(t[i], m, fdesc)) return false;
    return true;
}
```

```
bool load_node_ptr_tab(node** *tp, int *szp, map m, FILE* fdesc) {
    int i;
    if (!load_int(szp, fdesc)) return false;
    *tabp = (node**)malloc(*szp * sizeof(node*));
    if (*tabp == NULL) return false;
    for (i=0; i<*szp; i++)
        if (!load_node_ptr(&((*tabp)[i]), m, fdesc)) return false;
    return true;
}
```

Simulation de la sauvegarde d'un graphe

■ Représentation mémoire d'un graphe

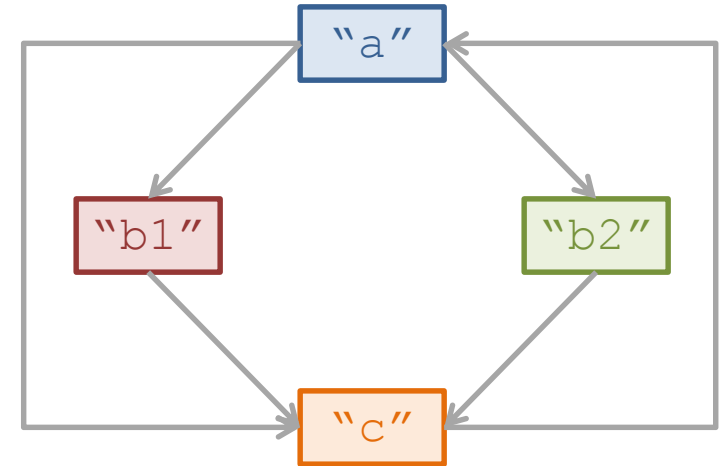


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph(     )
```

Input



Dictionnaire

Output

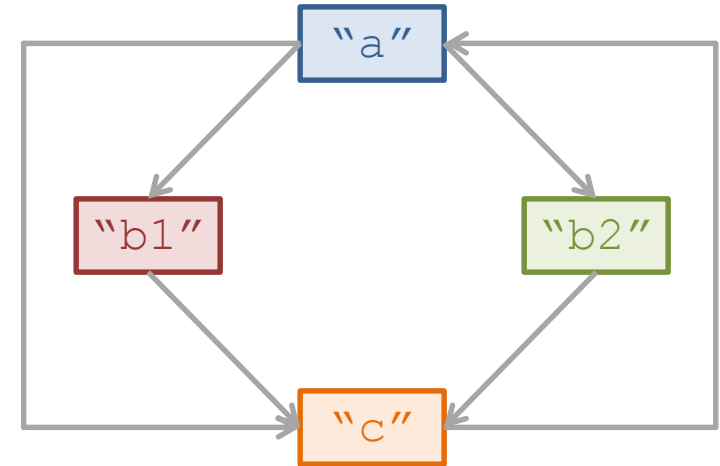
Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph(  )
```

```
new_map()
```

Input



Dictionnaire

0 ⇔ NULL

Output

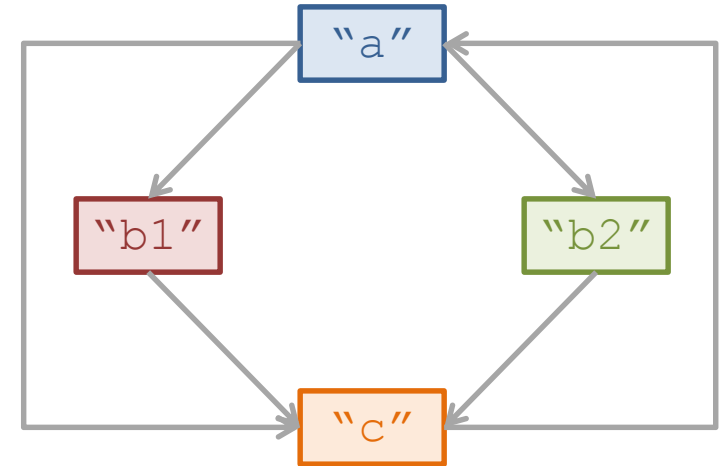
Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph(     )
```

```
save_pointed_node_tab(     )
```

Input



Dictionnaire

0 ⇔ NULL

Output

Simulation de la sauvegarde d'un graphe

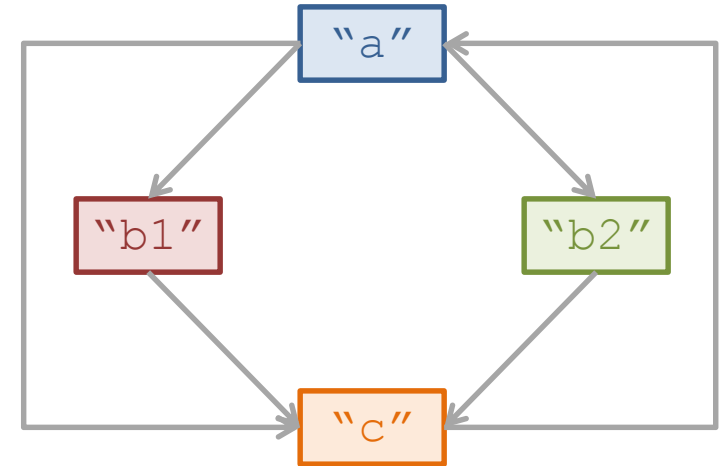
Pile d'appel des fonctions

```
save_graph( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node_tab( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_int(4)
```

Input



Dictionnaire

0 ⇔ NULL

4

Output

Simulation de la sauvegarde d'un graphe

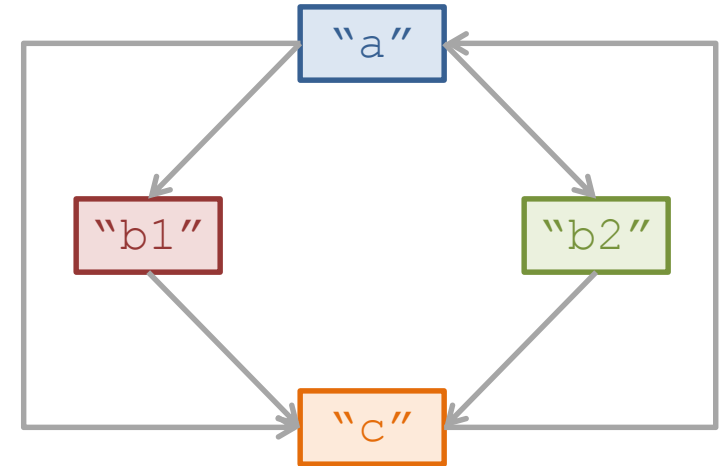
Pile d'appel des fonctions

```
save_graph( [→] [→] [→] [→] )
```

```
save_pointed_node_tab( [→] [→] [→] [→] )
```

```
save_pointed_node( → )
```

Input



Dictionnaire

0 ⇔ NULL

4

Output

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph( [→] [→] [→] [→] )
```

```
save_pointed_node_tab( [→] [→] [→] [→] )
```

```
save_pointed_node( → )
```

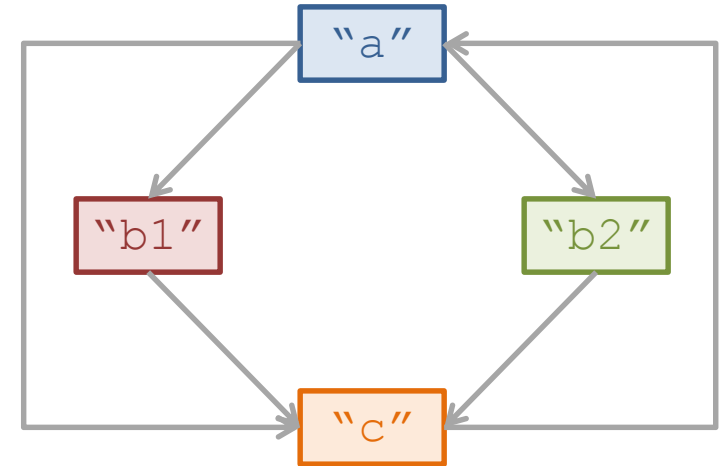
```
save_node_ptr( → )
```

Dictionnaire

0 ⇔ NULL

4

Input



Output

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph(  )
```

```
save_pointed_node_tab(  )
```

```
save_pointed_node(  )
```

```
save_node_ptr(  )
```

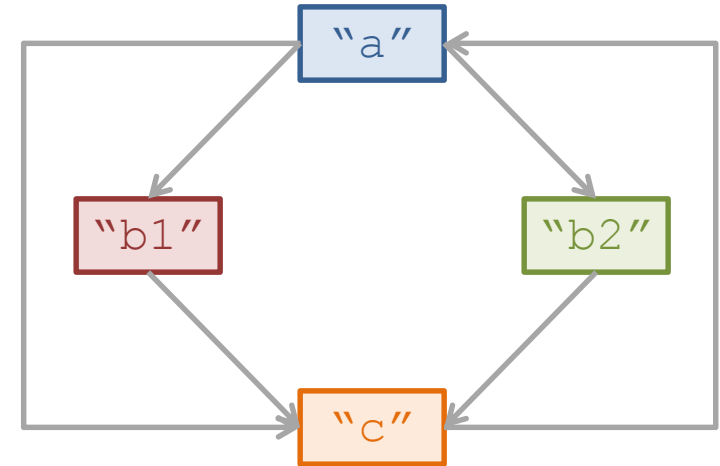
```
save_int( get_id(  ) )
```

Dictionnaire

0 ⇔ NULL

4

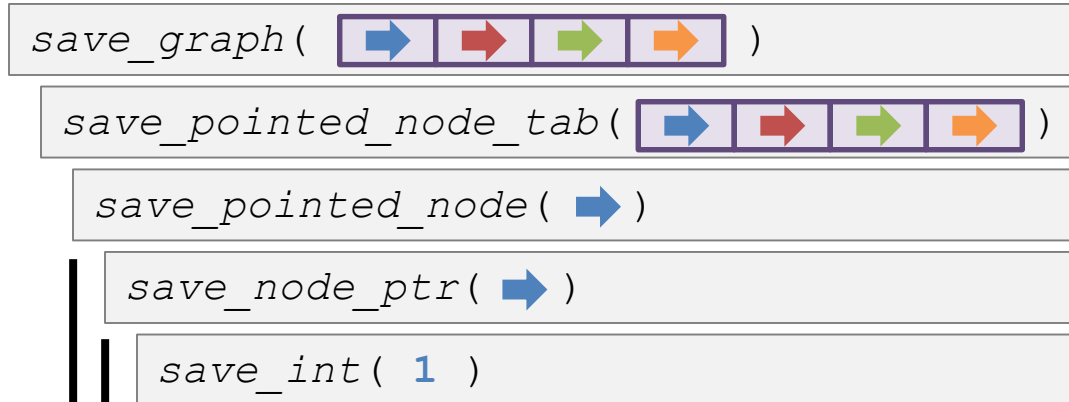
Input



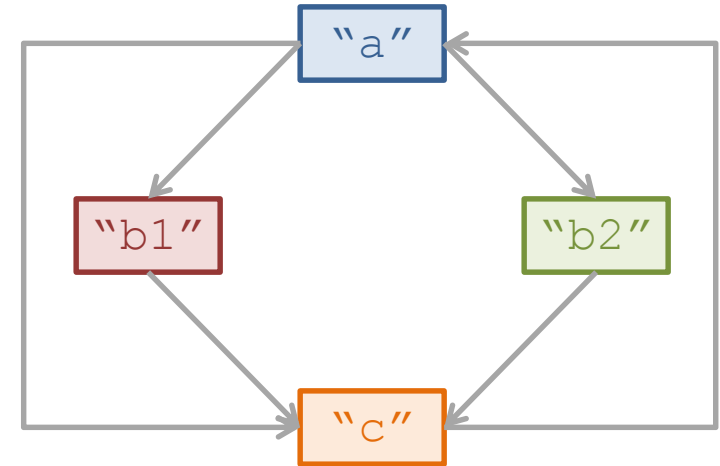
Output

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

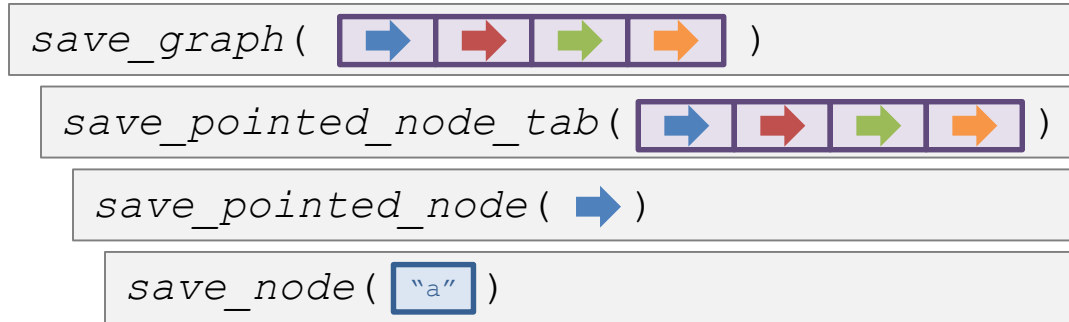
1 ⇔ blue arrow



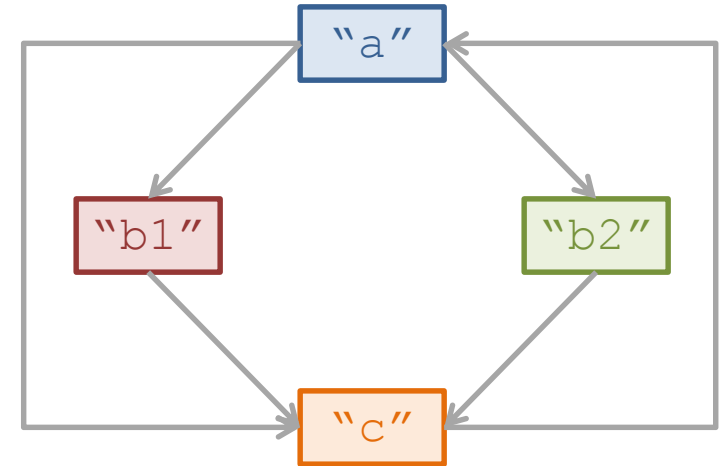
Output

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

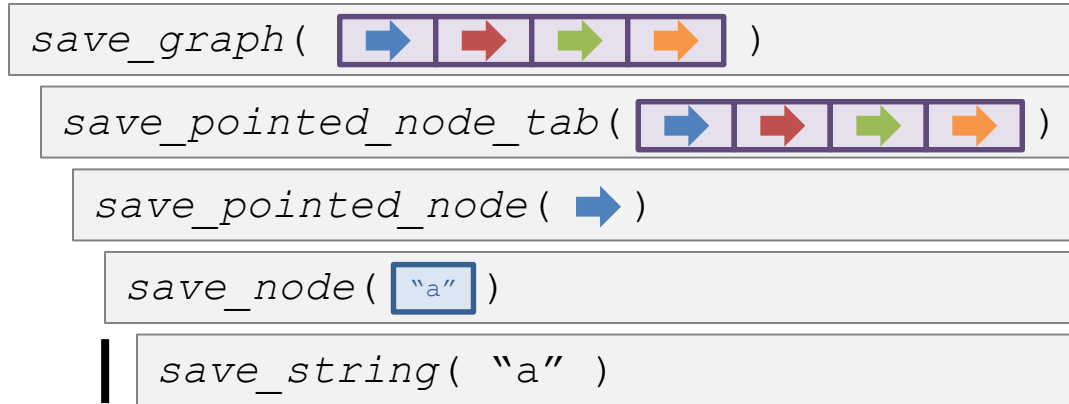
1 ⇔ blue arrow



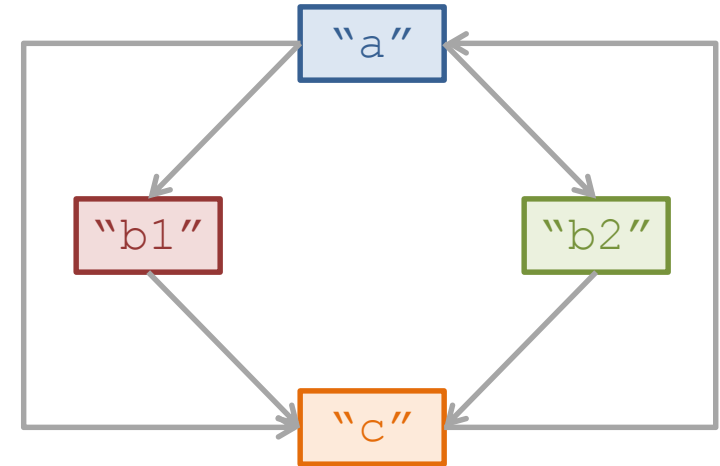
Output

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

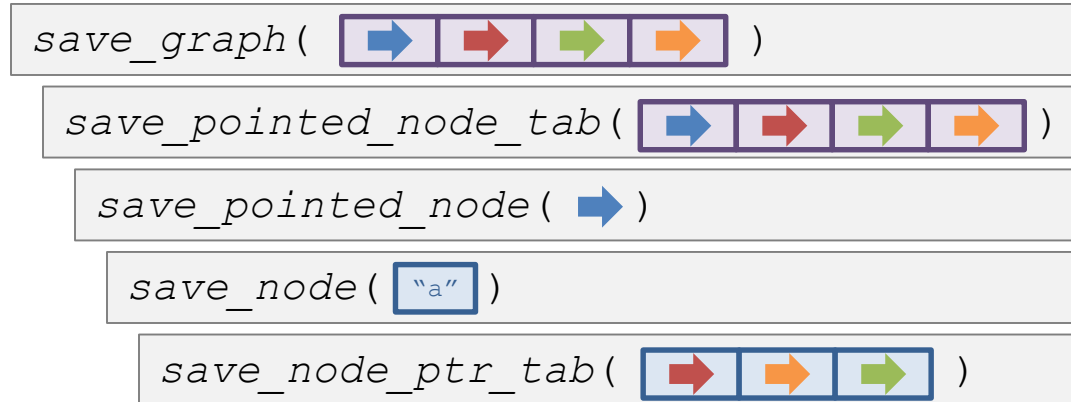
1 ⇔ blue arrow

Output

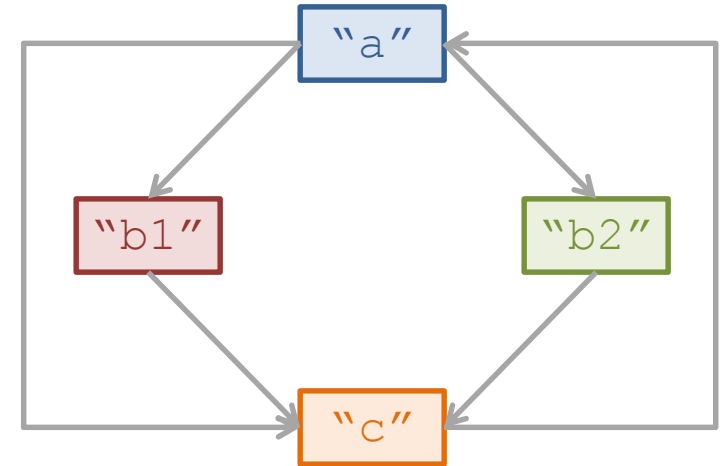


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

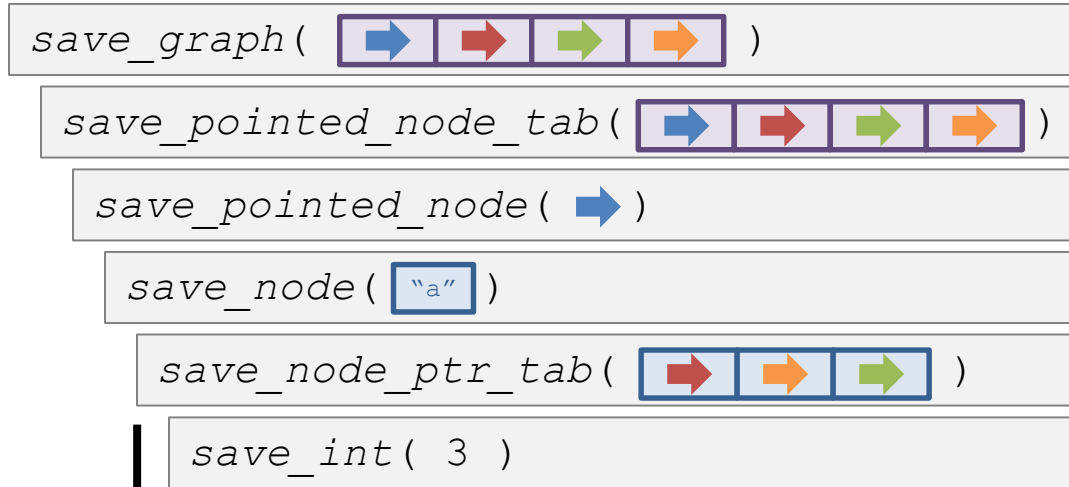
1 ⇔ [blue arrow]

Output

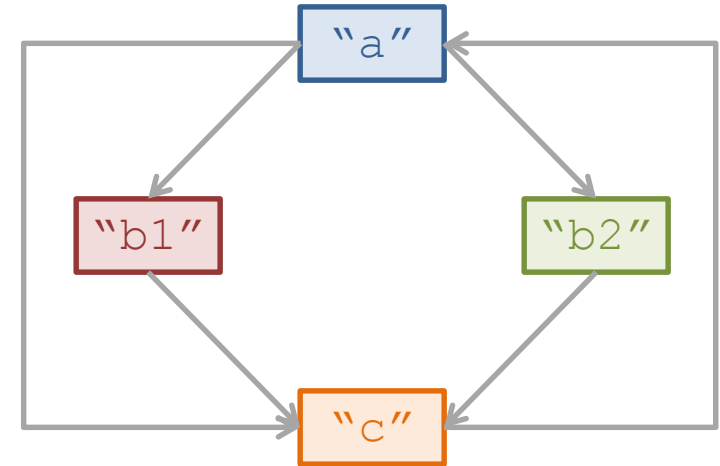


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

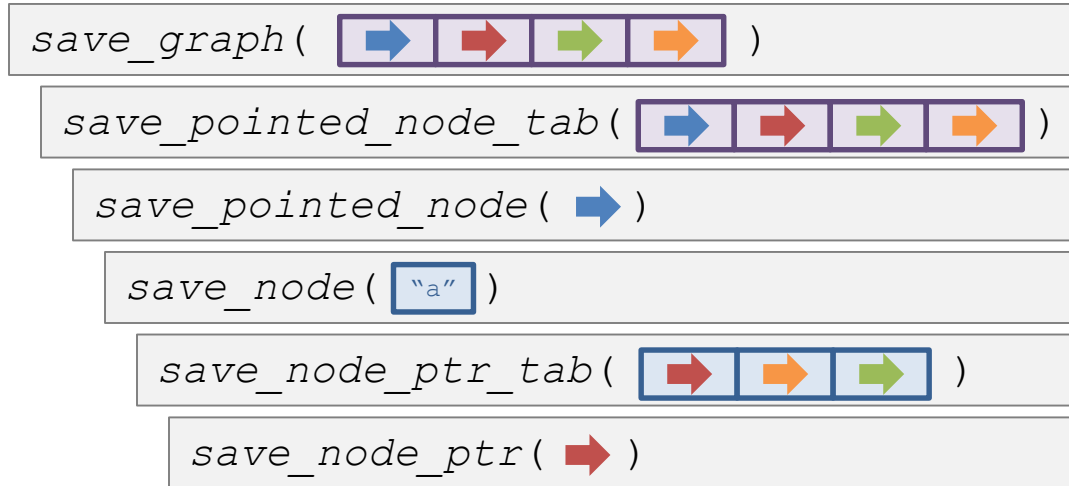
1 ⇔ →

Output

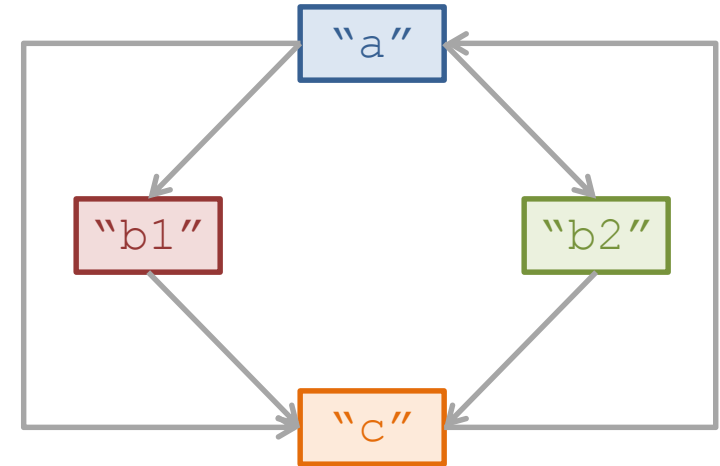


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

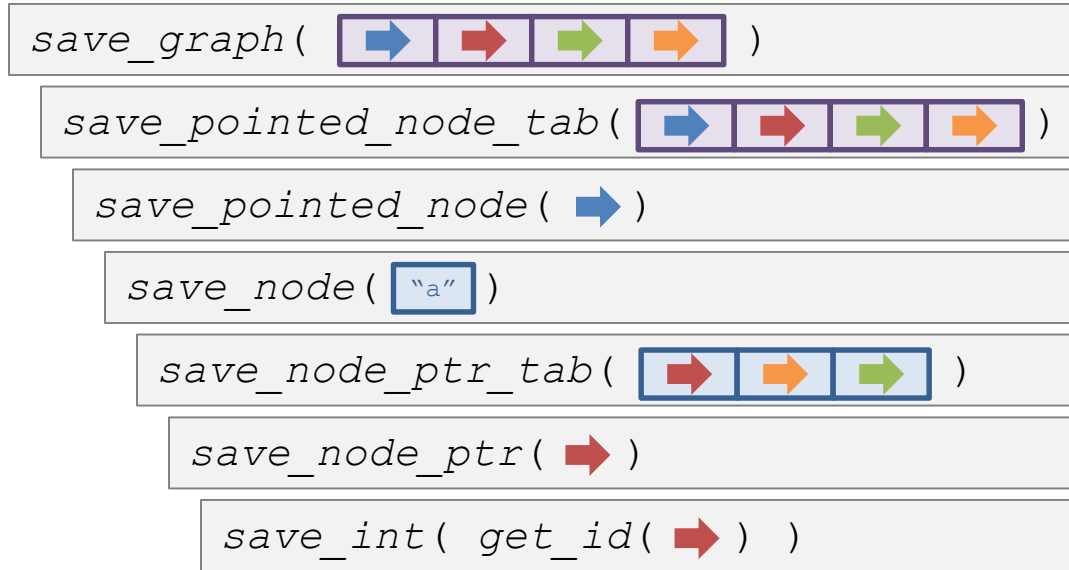
1 ⇔ blue arrow

Output

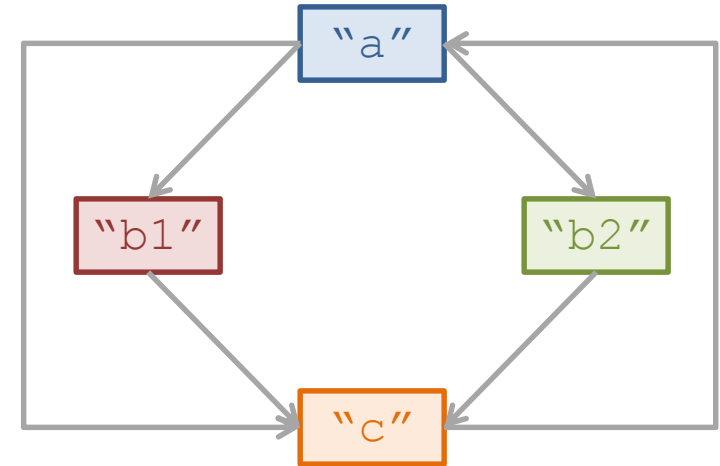


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

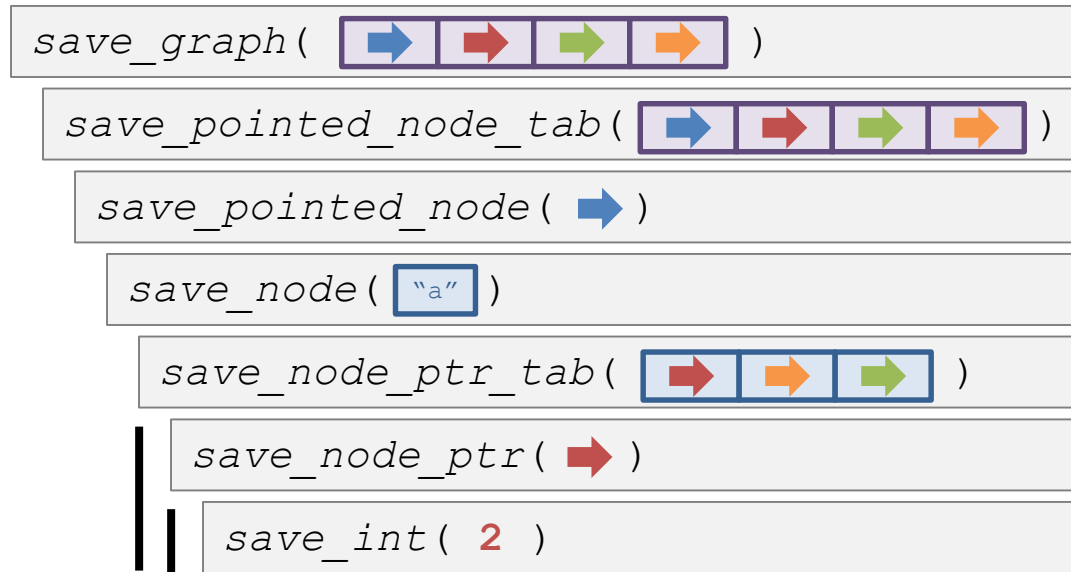
1 ⇔ blue arrow

Output

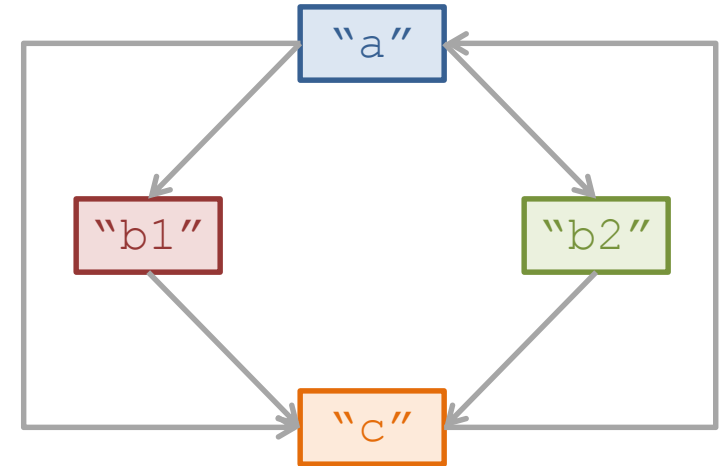


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL

1 ⇔ blue arrow

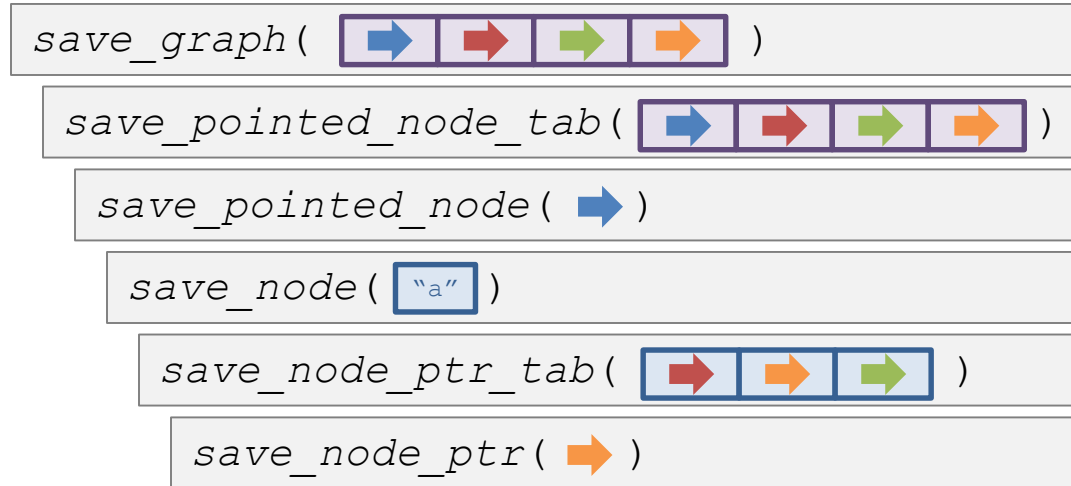
2 ⇔ red arrow

Output

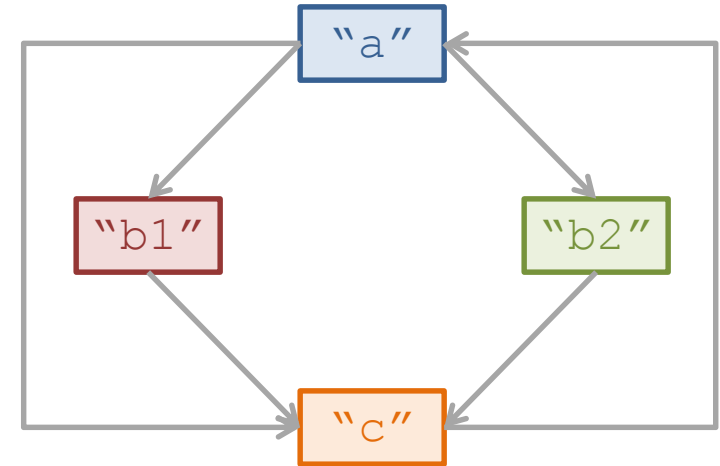


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

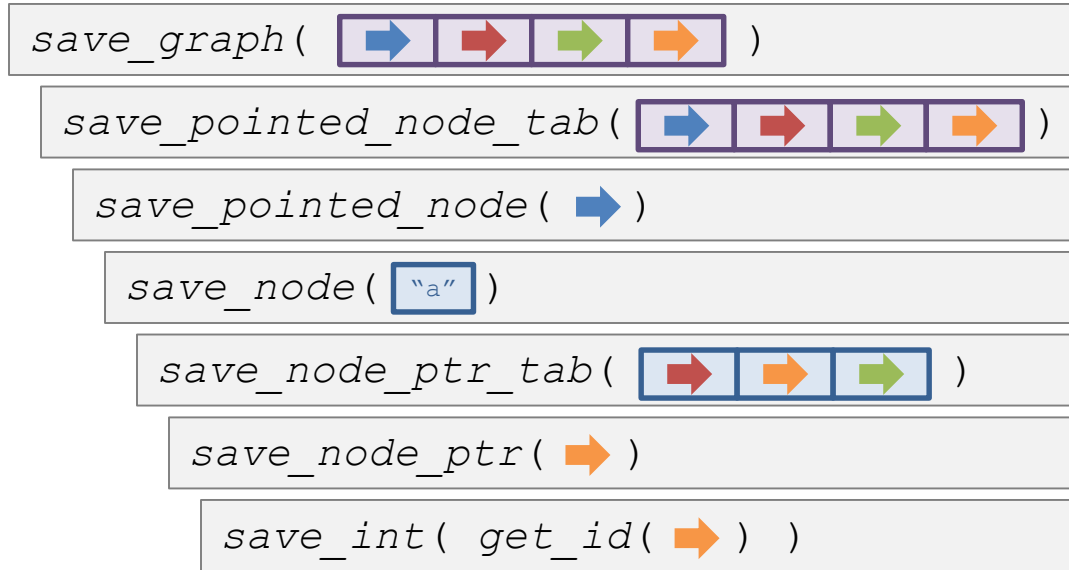
0 ⇔ NULL
1 ⇔ blue arrow
2 ⇔ red arrow

Output

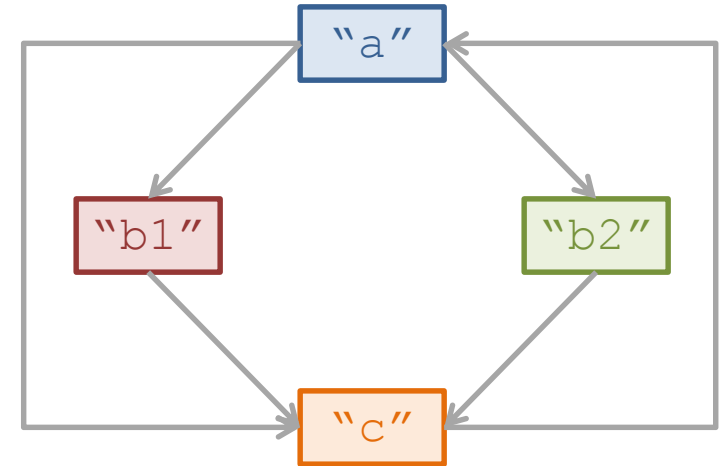


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

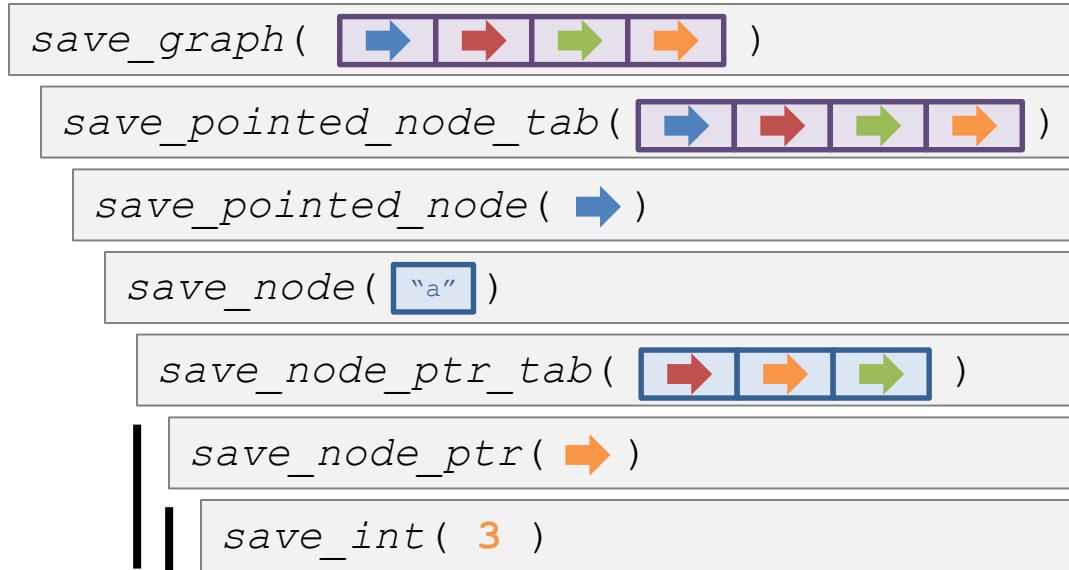
0 ⇔ NULL
1 ⇔ blue arrow
2 ⇔ red arrow

Output

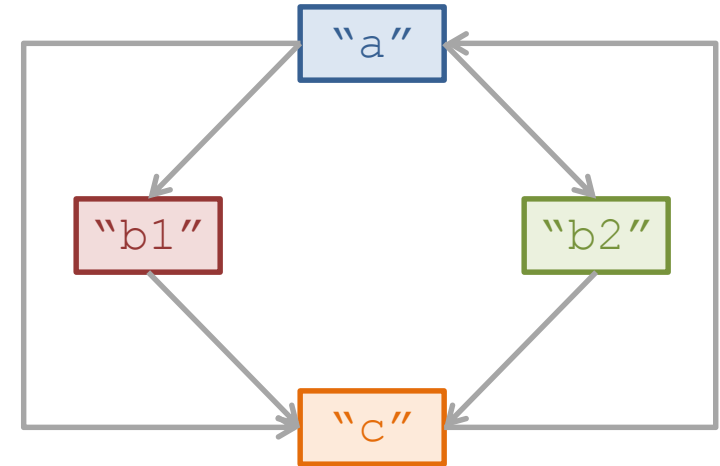


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

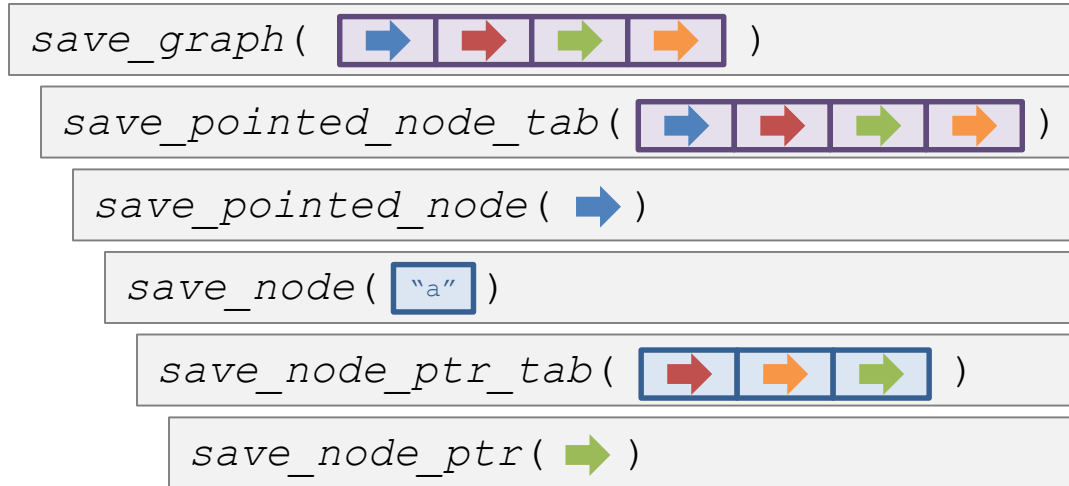
0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow

Output

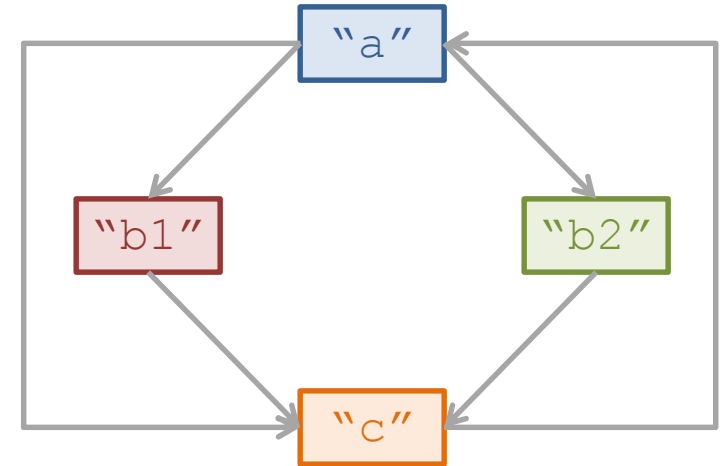
4	1	1	'a'	3	2	3
---	---	---	-----	---	---	---

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0 ⇔ NULL
1 ⇔ blue
2 ⇔ red
3 ⇔ orange

Output



Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node_tab( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node( blue arrow )
```

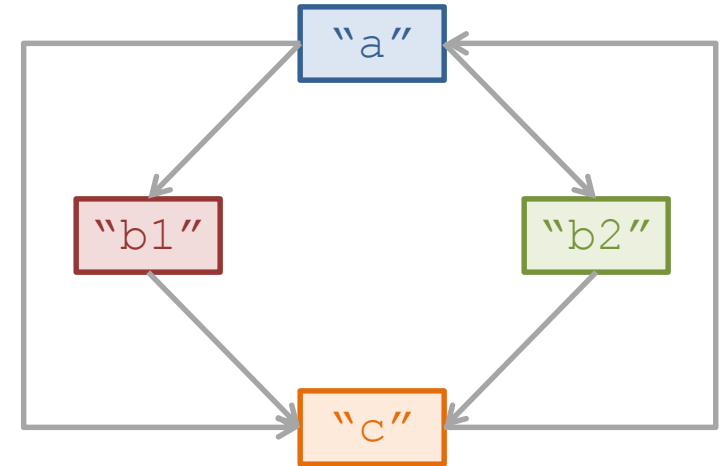
```
save_node( "a" )
```

```
save_node_ptr_tab( [red arrow, orange arrow, green arrow] )
```

```
save_node_ptr( green arrow )
```

```
save_int( get_id( green arrow ) )
```

Input



Dictionnaire

0 ⇔ NULL

1 ⇔ blue arrow

2 ⇔ red arrow

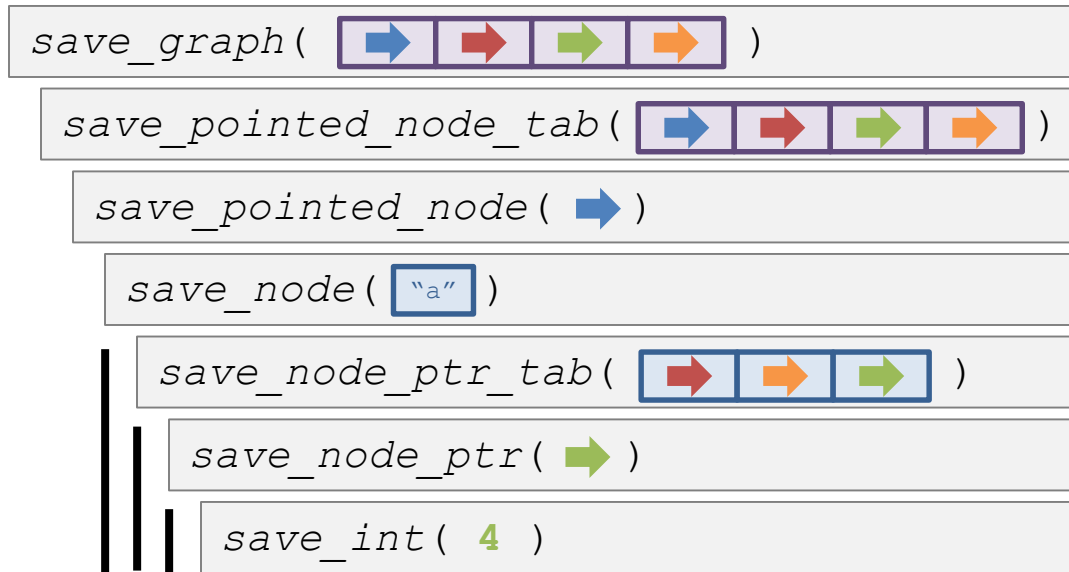
3 ⇔ orange arrow

Output

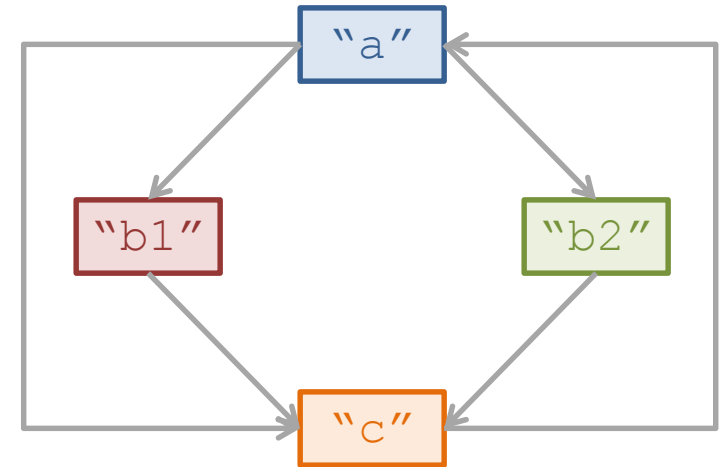
4	1	1	'a'	3	2	3
---	---	---	-----	---	---	---

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

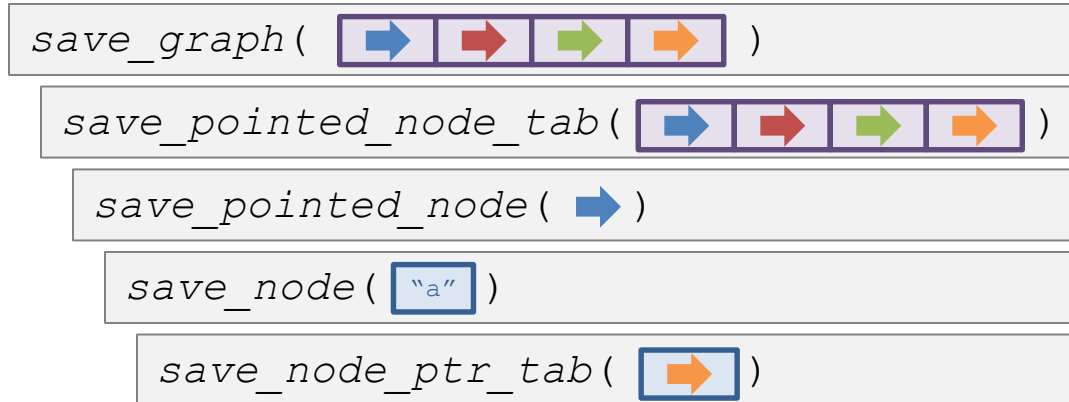
0	↔	NULL
1	↔	blue
2	↔	red
3	↔	orange
4	↔	green

Output

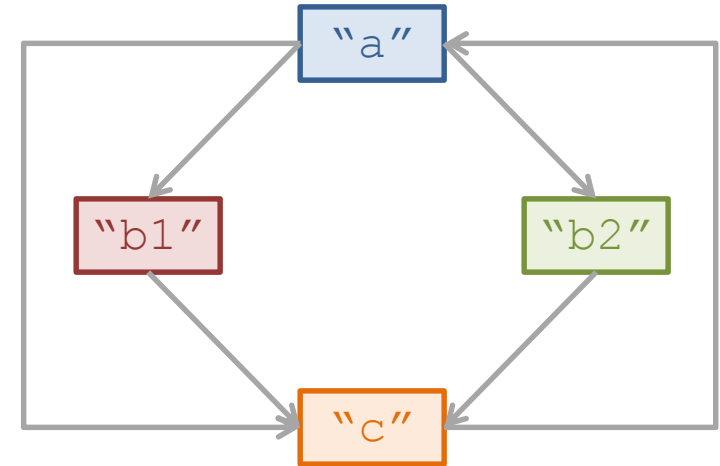


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

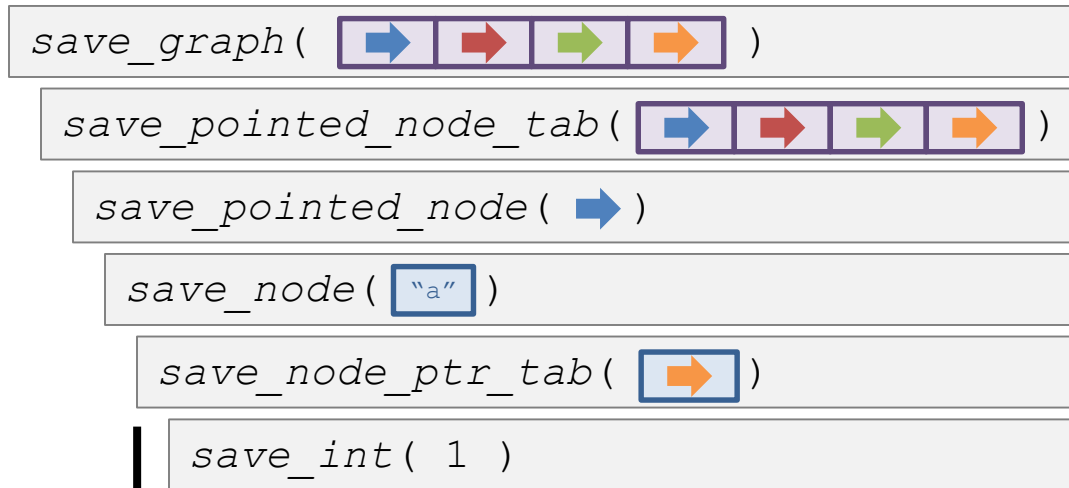
0	↔	NULL
1	↔	blue
2	↔	red
3	↔	orange
4	↔	green

Output

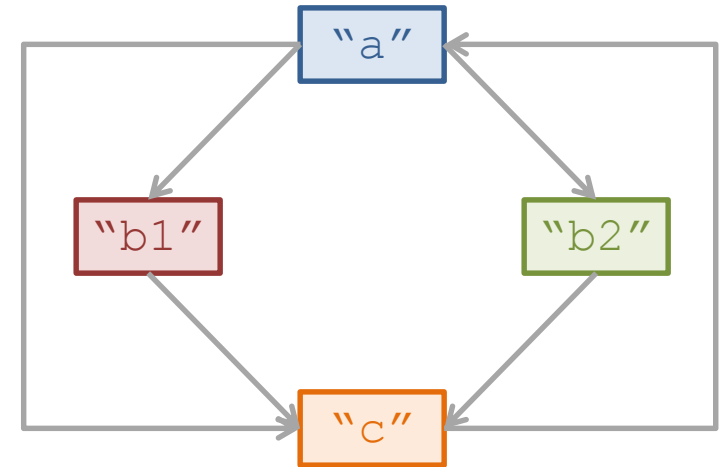


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

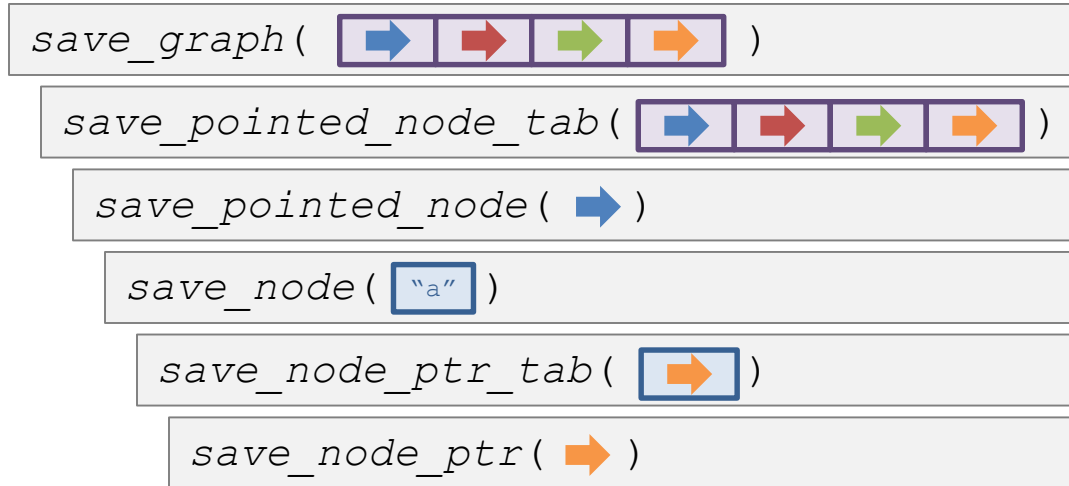
0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output

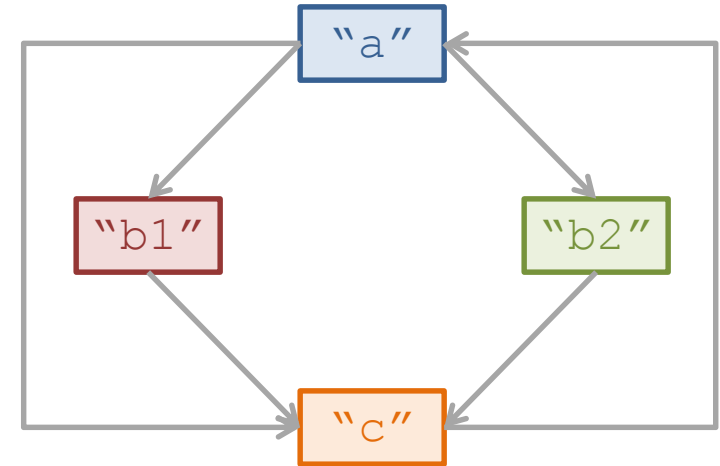


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0	↔	NULL
1	↔	blue
2	↔	red
3	↔	orange
4	↔	green

Output



Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node_tab( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node( blue arrow )
```

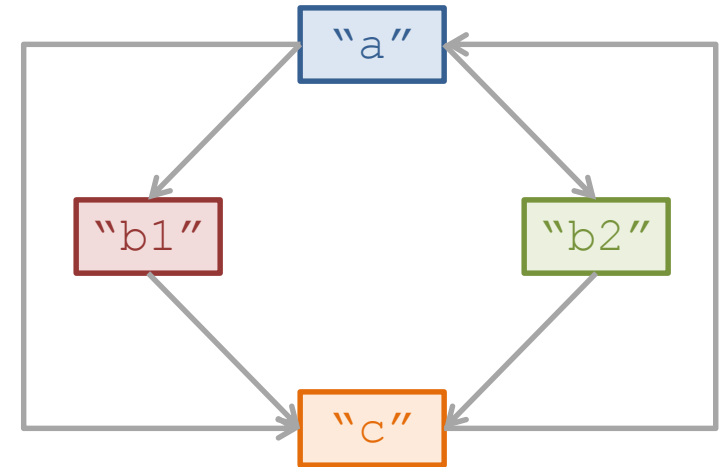
```
save_node( "a" )
```

```
save_node_ptr_tab( [orange arrow] )
```

```
save_node_ptr( orange arrow )
```

```
save_int( get_id( orange arrow ) )
```

Input



Dictionnaire

0 ⇔ NULL

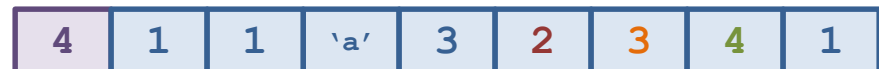
1 ⇔ blue arrow

2 ⇔ red arrow

3 ⇔ orange arrow

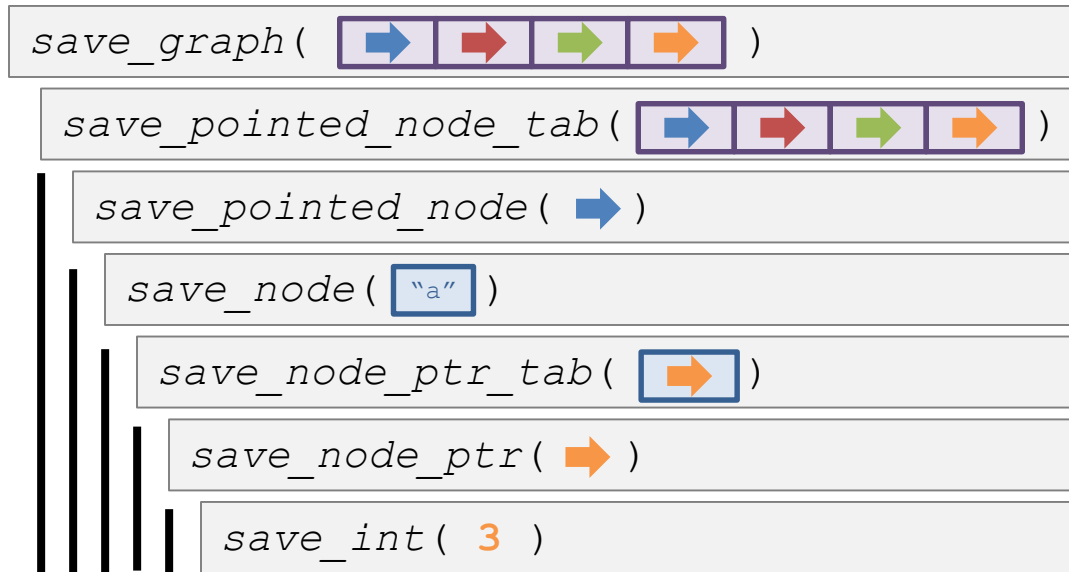
4 ⇔ green arrow

Output

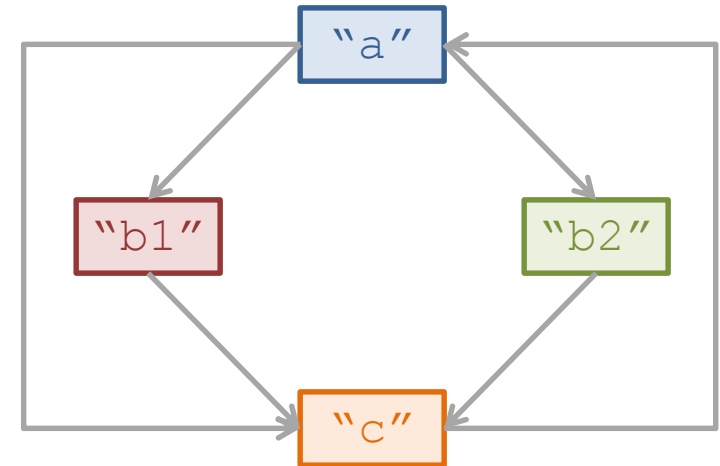


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output



Simulation de la sauvegarde d'un graphe

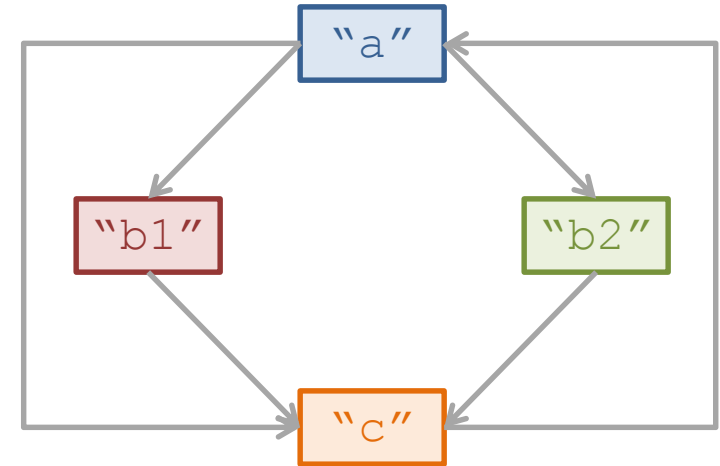
Pile d'appel des fonctions

```
save_graph( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node_tab( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node( red arrow )
```

Input



Dictionnaire

0 ⇔ NULL

1 ⇔ blue arrow

2 ⇔ red arrow

3 ⇔ orange arrow

4 ⇔ green arrow

Output

[4 | 1 | 1 | 'a' | 3 | 2 | 3 | 4 | 1 | 3 | ...]

...

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph( [blue arrow, red arrow, green arrow, orange arrow] )
```

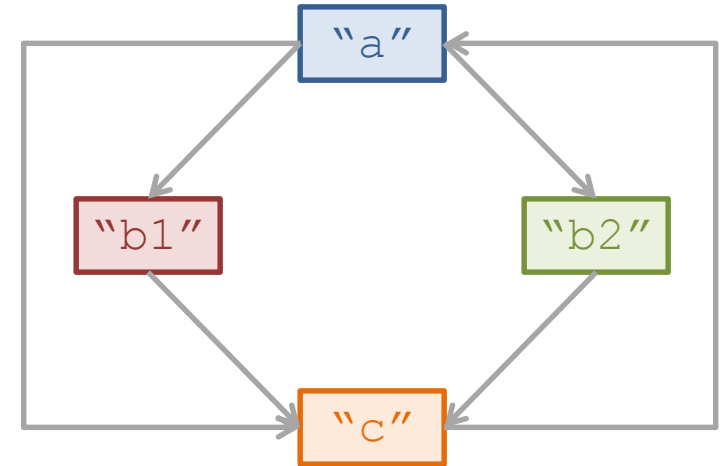
```
save_pointed_node_tab( [blue arrow, red arrow, green arrow, orange arrow] )
```

```
save_pointed_node( red arrow )
```

```
save_node_ptr( red arrow )
```

```
save_int( get_id( red arrow ) )
```

Input



Dictionnaire

0 ⇔ NULL

1 ⇔ blue arrow

2 ⇔ red arrow

3 ⇔ orange arrow

4 ⇔ green arrow

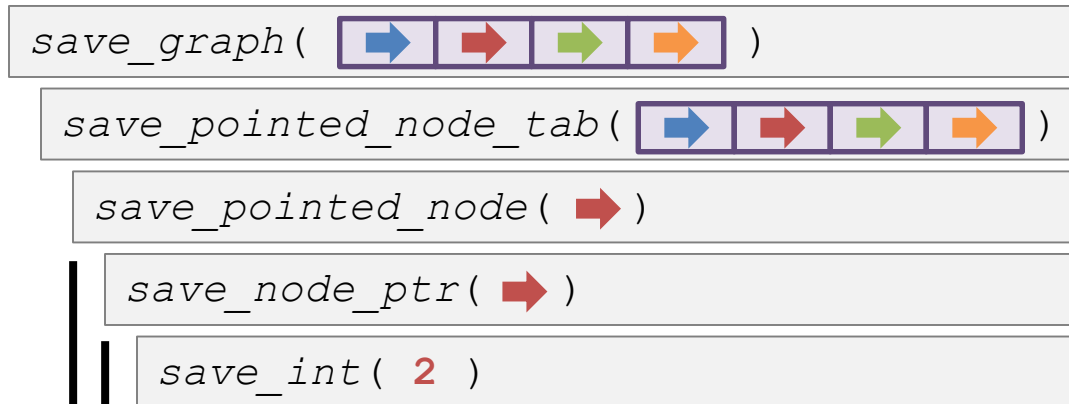
Output

[4, 1, 1, 'a', 3, 2, 3, 4, 1, 3] ...

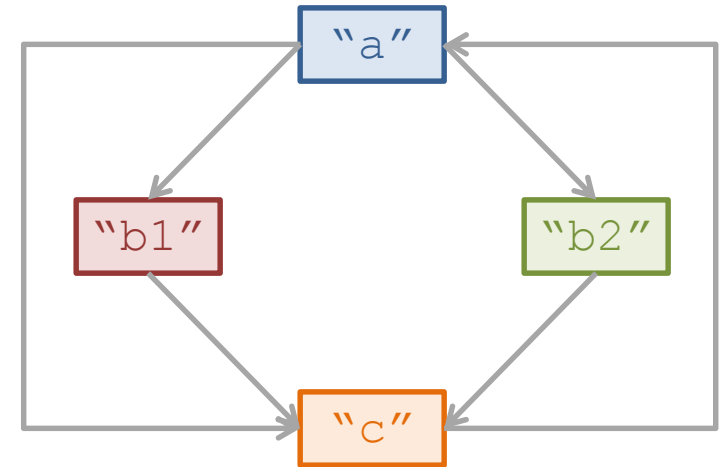
...

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



Input



Dictionnaire

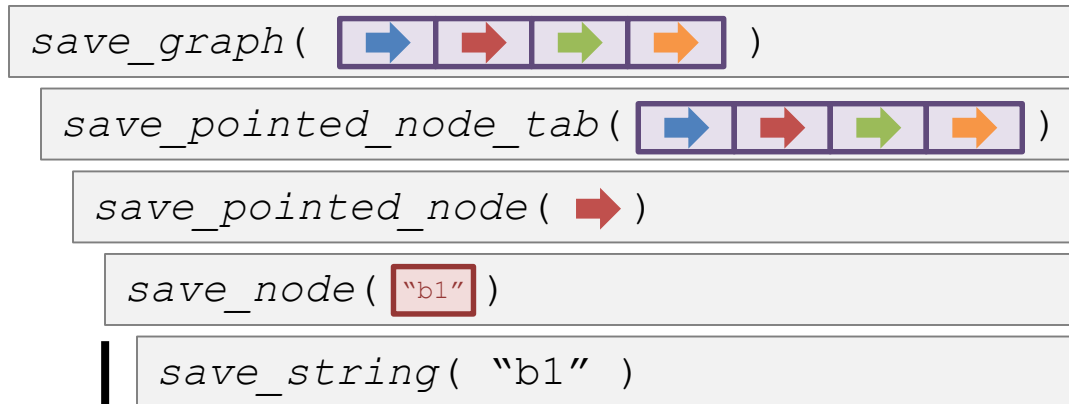
0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output

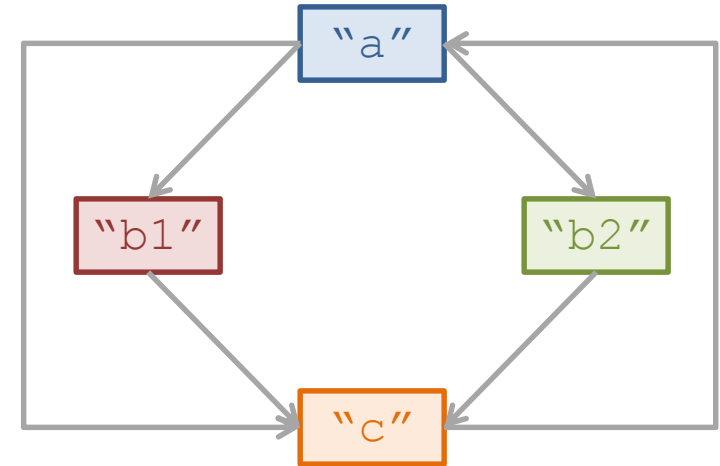


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



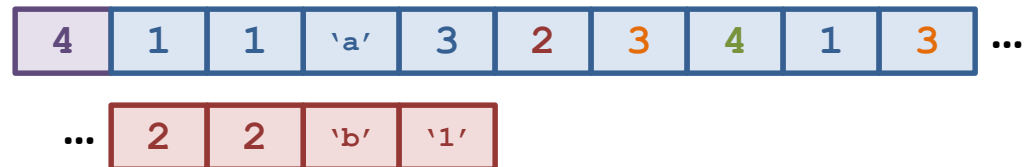
Input



Dictionnaire

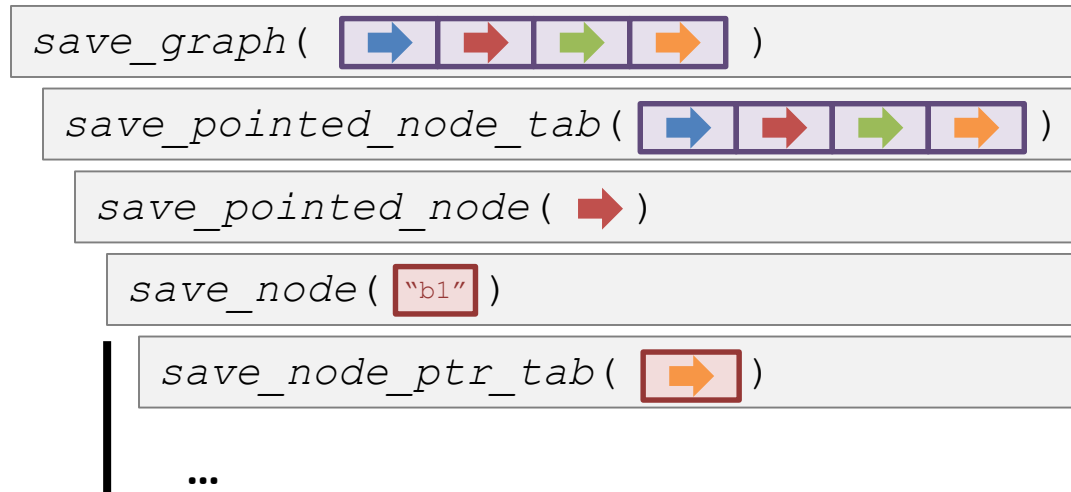
0	⇔	NULL
1	⇔	blue arrow
2	⇔	red arrow
3	⇔	orange arrow
4	⇔	green arrow

Output

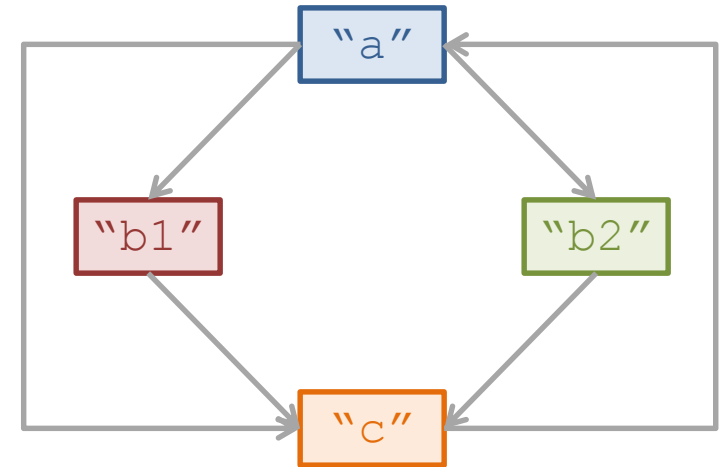


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



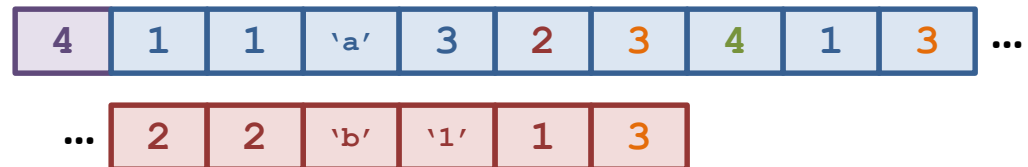
Input



Dictionnaire

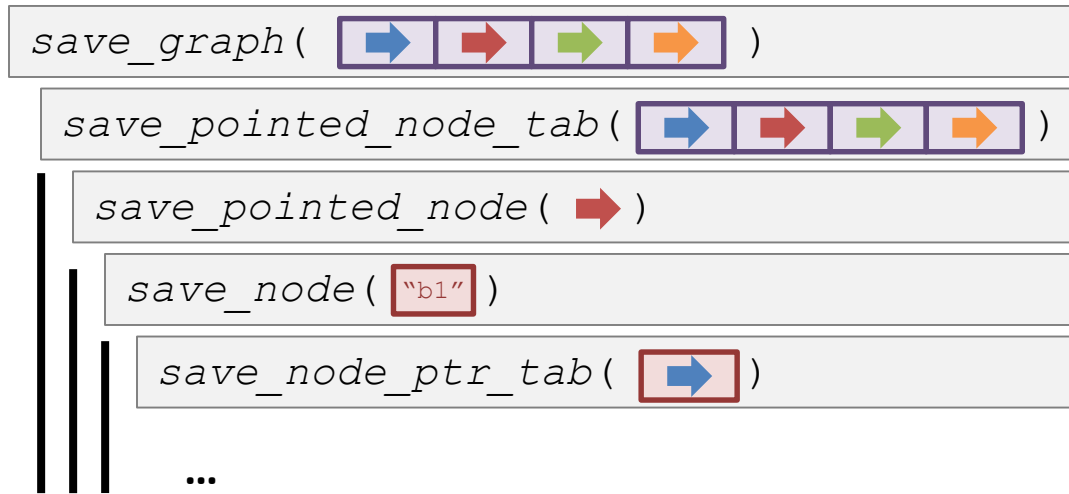
0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output

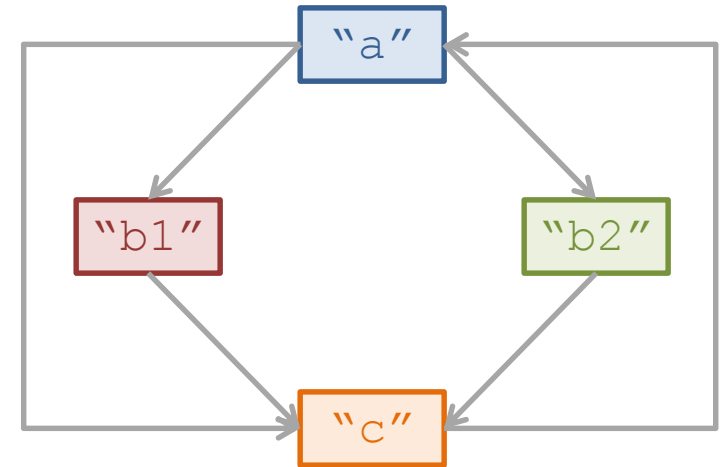


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



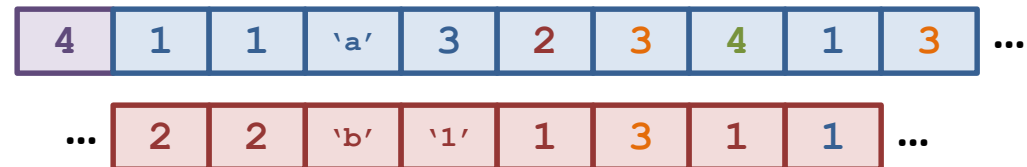
Input



Dictionnaire

0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output



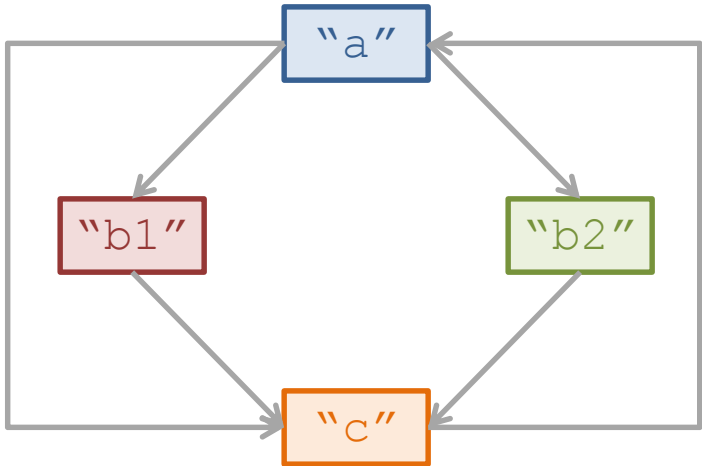
Pile d'appel des fonctions

```
save_graph(  )
```

```
save_pointed_node_tab(  )
```

```
save_pointed_node( ➡ )
```

Input



Dictionnaire

0 ⇔ NULL

1 ⇔ →

2 ⇔ →

3 ⇌ →

4 ⇌ →

Output

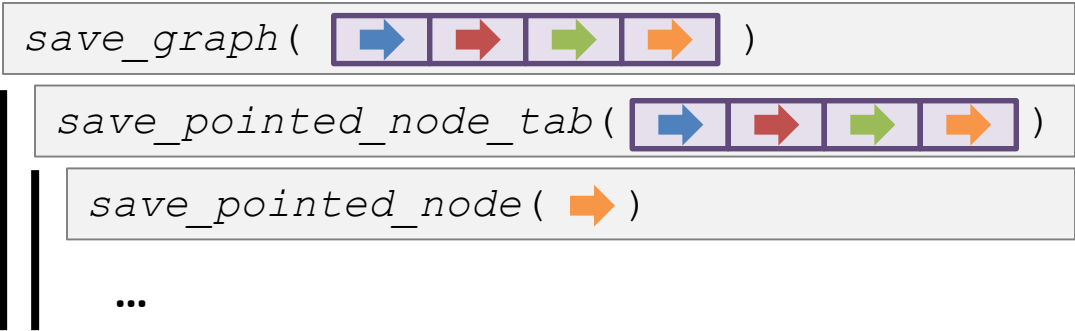
4	1	1	'a'	3	2	3	4	1	3	...
---	---	---	-----	---	---	---	---	---	---	-----

... 2 2 'b' '1' 1 3 1 1 ...

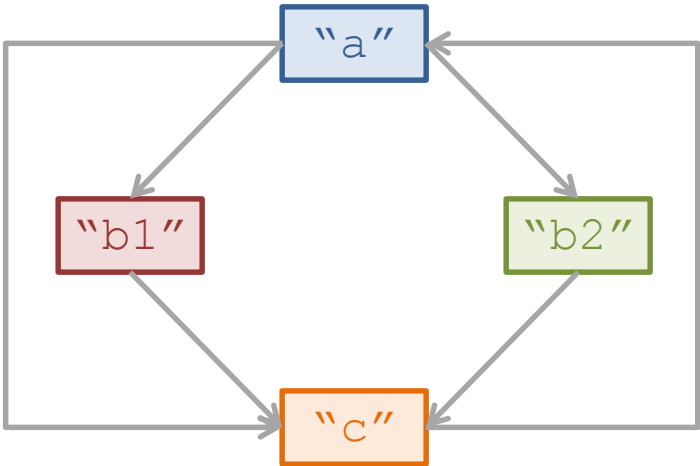
... 4 2 'b' '2' 1 3 1 1 ...

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions



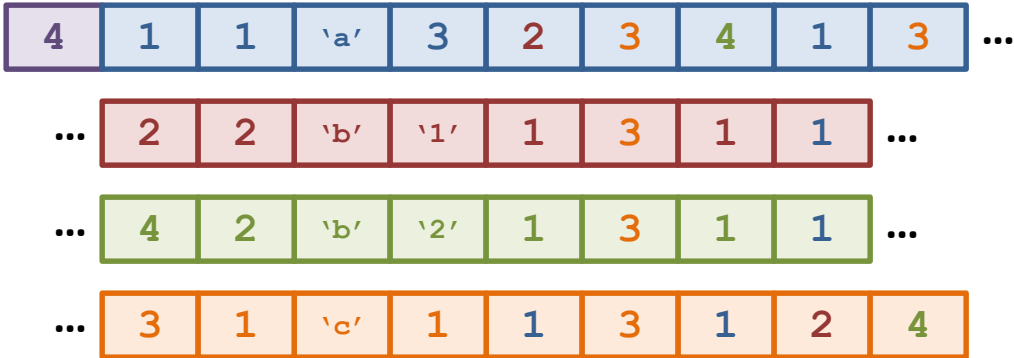
Input



Dictionnaire





0	↔	NULL
1	↔	blue arrow
2	↔	red arrow
3	↔	orange arrow
4	↔	green arrow

Output

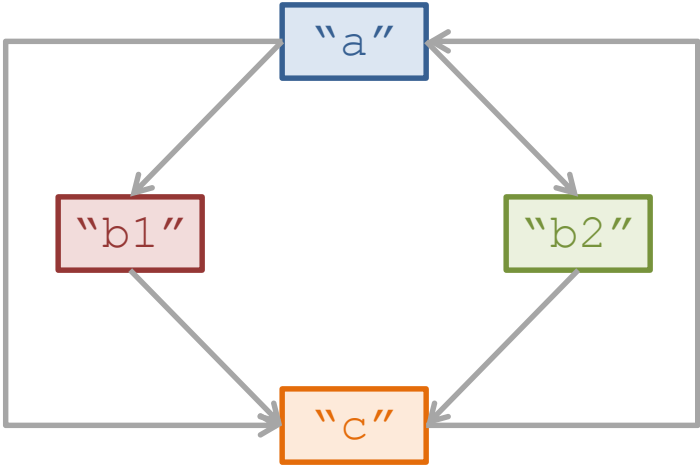


Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

```
save_graph(     )  
delete_map()
```

Input



Dictionnaire

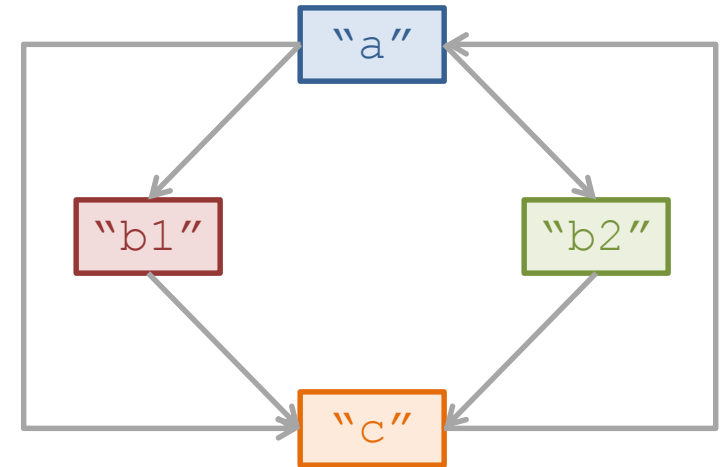
Output

4	1	1	'a'	3	2	3	4	1	3	...
...	2	2	'b'	'1'	1	3	1	1	...	
...	4	2	'b'	'2'	1	3	1	1	...	
...	3	1	'c'	1	1	3	1	2	4	

Simulation de la sauvegarde d'un graphe

Pile d'appel des fonctions

Input



Dictionnaire

Output

