

# INF3135

## Construction et maintenance de logiciels

### **Chapitre 5: Structures de données**

Alexandre Blondin Massé

Université du Québec à Montréal  
Département d'informatique

Été 2020

# Généralités

## Structure de données

Organisation **logique** d'un ensemble de données

## Plusieurs objectifs

- **Simplifier** le traitement
- Offrir des opérations **efficaces**
- Économiser de l'espace **mémoire**

## Interface et implémentation

- **Interface**: opérations supportées (type abstrait)
- **Implémentation**: organisation des données en mémoire, actions effectuées pour réaliser les opérations

## Type abstrait: exemples

- **Pile** (*stack*): principe *last in first out* (*LIFO*)
- **File** (*queue*): principe *first in first out* (*FIFO*)
- **File à priorité** (*priority queue*): la priorité des éléments peut être augmentée ou diminuée
- **Liste**: les éléments sont ordonnés, on peut avoir des opérations d'accès, d'insertion, de suppression, ...
- **Ensemble** (*set*): aucune donnée répétée (doublon), vérification d'appartenance d'éléments, données ordonnées ou non, etc.
- **Tableau associatif** (*map*): un ensemble de paires clé-valeur, les clés doivent être uniques, les valeurs peuvent être répétées
- **Partition**: division d'un ensemble en parties, fusion entre parties, vérifier si deux éléments sont dans la même partie, ...
- **Graphe**: relations symétriques (graphes non orientés) ou non symétrique (graphes orientés) entre entités
- etc.

# Implémentation: exemples

- **Tableau statique:** mémoire allouée et fixe, capacité maximale permise
- **Tableau dynamique:** tableau compressé ou allongé selon les besoins
- **Liste simplement chaînée:** chaque élément a une référence au suivant
- **Liste doublement chaînée:** chaque élément a une référence à l'élément précédent et à l'élément suivant
- **Structure arborescente:** arbres binaires, arbres préfixes, arbres suffixes, arbres d'arité quelconque, arbres coloriés, arbres-kd, etc.
- **Tableau multidimensionnel:** statiques ou dynamiques
- **Liste d'adjacence:** matrice creuse, graphes
- etc.

# Invariants et opérations

## Invariant

- **Propriété** qui doit être satisfaite en tout temps
- Généralement vérifiable à l'aide d'une **fonction** booléenne

## Opération

- Toute fonction qui **modifie** la structure de données
- Doit toujours préserver les **invariants**

## Exemples

- **Chaîne de caractères**: termine par '`\0`'
- **Liste simplement chaînée**: le dernier noeud pointe vers NULL
- **Arbre binaire de recherche**: les clés respectent l'ordre, ...

# Table des matières

- 1 Allocation dynamique
- 2 Gestion de la mémoire
- 3 Piles
- 4 Tableaux dynamiques
- 5 Tableaux multidimensionnels
- 6 Arbres binaires de recherche

## Allocation dynamique

# Allocation dynamique

- Jusqu'à maintenant: mémoire réservée de façon **statique**
- Or, cette information n'est **pas toujours connu** à l'avance
- **Solution**: allouer l'espace mémoire de façon **dynamique**
- Dans la bibliothèque `stdlib.h`:

```
// Réserve un bloc de taille `size`  
void *malloc(size_t size);  
// Libère l'espace mémoire pointé par `ptr`  
void free(void *ptr);  
// Réserve un bloc de taille `nmemb * size` initialisé à 0  
void *calloc(size_t nmemb, size_t size);  
// Redimensionne un bloc de taille `size` déjà alloué dynamiquement  
void *realloc(void *ptr, size_t size);  
// Redimensionne un bloc de taille `nmemb * size`  
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```



# Les fonctions malloc et calloc

```
void *malloc(size_t size);
```

- Réserve sur le **tas** (*heap*) un bloc de mémoire
- De **taille** size
- **Retourne** un pointeur vers ce bloc
- **Retourne** NULL s'il n'y a plus d'espace mémoire

```
void *calloc(size_t nmemb, size_t size);
```

- Réserve nmemb blocs de mémoire **consécutifs**
- De taille **individuelle** size
- **Initialise** toutes les valeurs à 0
- **Retourne** un pointeur vers ce bloc
- **Retourne** NULL s'il n'y a plus d'espace mémoire

# La fonction free

```
void free(void *ptr);
```

- **Libère** l'espace mémoire pointé par ptr
- Réserve lors d'un appel **précédent** à malloc ou calloc
- La taille libérée est **égale** à celle réservée
- Si ptr == NULL, alors rien ne se passe

## Attention

- Si free a déjà été appelé sur ptr
- Ou si la mémoire pointée par ptr n'a pas été allouée précédemment

Alors le comportement est **indéfini**.

# Les fonctions realloc et reallocarray

```
void *realloc(void *ptr, size_t size);
```

- **Redimensionne** un bloc de mémoire à la taille `size`
- **Préalablement** réservé avec `malloc` ou `calloc`
- **Retourne** un pointeur vers le bloc redimensionné
- **Retourne** `NULL` s'il n'y a plus d'espace mémoire

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

- **Équivalent** à `realloc(ptr, nmemb * size)`

## Attention

- La valeur des **octets** présents avant et après est **préservée**
- Si **agrandissement**, les « nouveaux » octets sont **indéterminés**
- Pointeur retourné peut être **différent** du pointeur en 1er argument

## Gestion de la mémoire

# Fuite de mémoire (*memory leak*)

- Mémoire **réservée** mais non **référéncée**
- Provoquée lorsqu'on appelle `malloc` ou `calloc`
- Et qu'on oublie de libérer avec `free`
- **Attention**: souvent « caché » derrière une autre fonction

## Exemples

- Initialisation d'une **structure de données** dynamique
- Utilisation de la fonction `strdup` (duplication de chaîne)
- Bibliothèque **SDL**: `SDL_Init`

## Comment les éviter?

- S'assurer que tout **appel** à `malloc` ou `calloc`
- Est **couplé** à un appel de la fonction `free`
- Souvent à l'aide de **fonctions**

## Responsabilité de mémoire

- Si une fonction utilise `malloc` sans `free` associé
- Le comportement doit être **documenté** (*docstring*)
- **Exemple**: la fonction `strdup`

*The `strdup()` function returns a pointer to a new string which is a duplicate of the string `s`. Memory for the new string is obtained with `malloc(3)`, and can be freed with `free(3)`.*

- Fournir une fonction **complémentaire** qui libère l'espace alloué
- **Exemple**: `SDL_Quit` est l'« inverse » de `SDL_Init`

### Attention

- Habitude des langages avec **ramasse-miettes** (*garbage collector*)
- Dans lequel on utilise `new` sans ménagement
- Préférer un passage par **adresse**
- Et utiliser `malloc/calloc/free` seulement lorsqu'inévitable

# L'outil Valgrind

- Cadriciel permettant de concevoir des outils d'**analyse dynamique**
- Permet de détecter des erreurs de **gestion de mémoire**
- Et de **profiler** un programme en détail
- **Lien officiel:** <http://valgrind.org/>
- **Invocation:**

```
$ valgrind [options valgrind] [programme] [options programme]
```

Plusieurs dizaines d'**options**:

- `--tool=<toolname>`: outil (par défaut, memcheck)
- `--leak-check=<no|summary|yes|full>`: vérifier fuites mémoires
- `--time-stamp=<yes|no>`: afficher chronologie
- `--track-origins=<yes|no>`: origine des valeurs non initialisées
- etc.

## Exemple (1/5)

```
#include <stdio.h>
#include <stdlib.h>

double *sum(const double *v1, const double *v2, unsigned int n) {
    double *v = malloc(n * sizeof(double));
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
    return v;
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5, 3.4};
    double v2[] = {-1.0, 2.1, -0.8};
    print_vector(sum(v1, v2, 3), 3);
    return 0;
}
```

## Résultat:

```
[ 1.000000 0.600000 2.600000 ]
```



## Exemple (2/5)

```
$ gcc sum_malloc.c -o sum_malloc
$ valgrind ./sum_malloc
$ valgrind --leak-check=yes ./sum_malloc
==6724== Memcheck, a memory error detector
==6724== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6724== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6724== Command: ./sum_malloc
==6724==
[ 1.000000 0.600000 2.600000 ]==6724==
==6724== HEAP SUMMARY:
==6724==     in use at exit: 24 bytes in 1 blocks
==6724==   total heap usage: 2 allocs, 1 frees, 1,048 bytes allocated
==6724==
==6724== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6724==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/[...])
==6724==    by 0x10876B: sum (in [...]/code/sum_malloc)
==6724==    by 0x1088BE: main (in [...]/code/sum_malloc)
==6724==
==6724== LEAK SUMMARY:
==6724==     definitely lost: 24 bytes in 1 blocks
==6724==     indirectly lost: 0 bytes in 0 blocks
==6724==     possibly lost: 0 bytes in 0 blocks
==6724==     still reachable: 0 bytes in 0 blocks
==6724==     suppressed: 0 bytes in 0 blocks
==6724==
==6724== For counts of detected and suppressed errors, rerun with: -v
==6724== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

## Example (3/5)

```
#include <stdio.h>
#include <stdlib.h>

double *sum(const double *v1, const double *v2, unsigned int n) {
    double *v = malloc(n * sizeof(double));
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
    return v;
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5, 3.4};
    double v2[] = {-1.0, 2.1, -0.8};
    double *v = sum(v1, v2, 3);
    print_vector(v, 3);
    free(v);
    return 0;
}
```

## Résultat:

```
[ 1.000000 0.600000 2.600000 ]
```

## Exemple (4/5)

```
$ gcc sum_malloc_free.c -o sum_malloc_free
$ valgrind --leak-check=yes ./sum_malloc_free
==10376== Memcheck, a memory error detector
==10376== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10376== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10376== Command: ./sum_malloc_free
==10376==
[ 1.000000 0.600000 2.600000 ]==10376==
==10376== HEAP SUMMARY:
==10376==      in use at exit: 0 bytes in 0 blocks
==10376==    total heap usage: 2 allocs, 2 frees, 1,048 bytes allocated
==10376==
==10376== All heap blocks were freed -- no leaks are possible
==10376==
==10376== For counts of detected and suppressed errors, rerun with: -v
==10376== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

### Allocation dynamique

- Est-ce que malloc est vraiment nécessaire ici?
- **Réponse:** non!

## Exemple (5/5)

```
#include <stdio.h>
#include <stdlib.h>

void compute_sum(const double *v1, const double *v2,
                 double *v, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5, 3.4};
    double v2[] = {-1.0, 2.1, -0.8};
    double v[3];
    compute_sum(v1, v2, v, 3);
    print_vector(v, 3);
    return 0;
}
```

**Résultat:**

[ 1.000000 0.600000 2.600000 ]

Piles

# Pile

- Structure de donnée **fondamentale**
- Stratégie *LIFO* = *last in first out*

## Interface minimale (pile de caractères)

```
// Initialize the stack
void stack_initialize(stack *s);
// Is stack empty?
bool stack_is_empty(const stack *s);
// Push a value on top
void stack_push(stack *s, char value);
// Pop the value from the top
char stack_pop(stack *s);
// Delete a stack
void stack_delete(stack *s);
```

## Implémentation

- **Tableau** statique ou dynamique
- Liste **simplement chaînée**

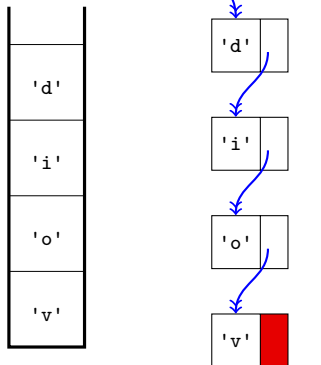
# Représentation

## Représentation schématique:

### Déclaration de types en C:

```
// Node
struct stack_node {
    char value;
    struct stack_node *next;
};

// Stack
typedef struct {
    struct stack_node *first;
    unsigned int size;
} stack;
```



# Invariants

## Trois invariants

Pour toute **pile** `s` et pour tout **noeud** `node` de `s`,

- `node.next == NULL` **ssi** `node` est le dernier noeud de `s`
- `s.first == NULL` **ssi** `stack_is_empty(s)` est *vrai*
- Le nombre de noeuds dans `s` est donné par `s.size`

## Utilité?

- Quand on implémente `stack_push` et `stack_pop`
- **Supposer** les invariants satisfaits en **début de fonction**
- Et s'assurer qu'ils sont encore satisfaits à la **fin**



# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

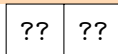
```
// Initialize the stack
```

```
void stack_initialize(stack *s) {
```

```
    s->first = NULL;
```

```
    s->size = 0;
```

```
}
```



```
// Is stack empty?
```

```
bool stack_is_empty(const stack *s) {
```

```
    return s->size == 0;
```

```
}
```

```
// Push a value on top
```

```
void stack_push(stack *s, char value) {
```

```
    struct stack_node *node = malloc(sizeof(struct stack_node));
```

```
    node->value = value;
```

```
    node->next = s->first;
```

```
    s->first = node;
```

```
    ++s->size;
```

```
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```



```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```

```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```



```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```

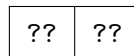
```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```



```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```



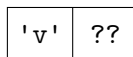
```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```



```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```



```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```



```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```



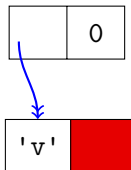
```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```

```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```

```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



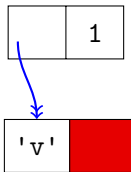


# Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}
```

```
// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}
```

```
// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



## Implémentation (2/2)

```
// Pop the value from the top
char stack_pop(stack *s) {
    if (!stack_is_empty(s)) {
        char value = s->first->value;
        struct stack_node *node = s->first;
        s->first = node->next;
        free(node);
        --s->size;
        return value;
    } else {
        fprintf(stderr, "Cannot pop from empty stack\n");
        exit(1);
        return '?';
    }
}

// Delete a stack
void stack_delete(stack *s) {
    while (!stack_is_empty(s)) stack_pop(s);
}
```

## Exemple d'utilisation: parenthésage équilibré

```
/**
 * Returns true if and only if an expression is balanced
 *
 * @param expr The expression to check
 * @returns True if and only if the expression is balanced
 */
bool is_balanced(char *expr) {
    bool balanced = true;
    stack s;
    stack_initialize(&s);
    for (unsigned int i = 0; balanced && expr[i] != '\0'; ++i) {
        if (expr[i] == '(') {
            stack_push(&s, '(');
        } else if (expr[i] == '[') {
            stack_push(&s, '[');
        } else if (expr[i] == '{') {
            stack_push(&s, '{');
        } else if (expr[i] == ')' || expr[i] == ']' || expr[i] == '}') {
            if (stack_is_empty(&s))
                balanced = false;
            else
                balanced = expr[i] == stack_pop(&s);
        }
    }
    balanced = balanced && stack_is_empty(&s);
    stack_delete(&s);
    return balanced;
}
```

## Tableaux dynamiques

# Tableau dynamique

- Tableau dont la taille **varie** selon l'usage
- Supporté **par défaut** dans plusieurs langages
- **Exemples**: C++ (vector) Java (ArrayList), Python (listes)

## Implémentation

- **Capacité** (*capacity*): taille « réelle » du tableau en mémoire
- **Taille** (*size*): nombre d'éléments « pertinents » dans le tableau

## Redimensionnement

- **Automatiquement**, en doublant la taille
- Ou **manuellement**, par un appel de fonction
- Parfois, **contracté** automatiquement quand trop d'espace inoccupé

# Interface

```
// Initialize an empty array
void array_initialize(array *a);

// Append an element to the end of an array
void array_append(array *a, int e);

// Insert an element in an array
void array_insert(array *a, unsigned int i, int element);

// Remove an element from an array at a given index
void array_remove(array *a, unsigned int i);

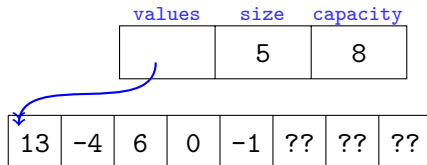
// Check if an array contains a given element
bool array_has_element(const array *a, int e);

// Return the element at a given index in an array
int array_get(const array *a, unsigned int i);

// Delete an array
void array_delete(array *a);
```

# Représentation

```
// A resizable array
typedef struct {
    int *values;           // The values of the array
    unsigned int size;      // The current size
    unsigned int capacity; // The capacity
} array;
```



## Invariants:

- La taille réservée par `values` est `capacity * sizeof(int)`
- `size <= capacity`
- Seules les `size` premières valeurs sont « pertinentes »

# Implémentation (1/2)

```
// Initialize an empty array
void array_initialize(array *a) {
    a->values = malloc(sizeof(int));
    a->size = 0;
    a->capacity = 1;
}

// Append an element to the end of an array
void array_append(array *a, int e) {
    array_resize_if_needed(a);
    a->values[a->size] = e;
    ++a->size;
}

// Insert an element in an array
void array_insert(array *a, unsigned int i, int e) {
    array_check_index(a, i);
    array_resize_if_needed(a);
    for (unsigned int j = a->size - 1; j > i; --j)
        a->values[j] = a->values[j - 1];
    a->values[i] = e;
    ++a->size;
}
```



## Implémentation (2/2)

```
// Remove an element from an array at a given index
void array_remove(array *a, unsigned int i) {
    array_check_index(a, i);
    ++i;
    while (i < a->size) {
        a->values[i - 1] = a->values[i];
        ++i;
    }
    --a->size;
}

// Check if an array contains a given element
bool array_has_element(const array *a, int e) {
    unsigned int i = 0;
    while (i < a->size && array_unsafe_get(a, i) != e)
        ++i;
    return i < a->size;
}

// Return the element at a given index in an array
int array_get(const array *a, unsigned int i) {
    array_check_index(a, i);
    return array_unsafe_get(a, i);
}

// Delete an array
void array_delete(array *a) {
    free(a->values);
}
```

# Fonctions supplémentaires

```
// Resize an array if the capacity is reached
void array_resize_if_needed(array *a) {
    if (a->size == a->capacity) {
        a->capacity *= 2;
        a->values = realloc(a->values, a->capacity * sizeof(int));
    }
}

// Return the element at index i in an array (no check)
int array_unsafe_get(const array *a, unsigned int i) {
    return a->values[i];
}

// Print an array to stdout
void array_print(const array *a) {
    unsigned int i;
    printf("[");
    for (i = 0; i < a->size; ++i) {
        printf(" %d", a->values[i]);
    }
    printf(" ]");
}

// Check if an index is out of bound
void array_check_index(const array *a, unsigned int i) {
    if (i >= a->size) {
        fprintf(stderr, "Invalid index %d (size = %d)\n", i, a->size);
        exit(1);
    }
}
```

# Utilisation

```
int main() {
    array a; array_initialize(&a);
    printf("Appending 3, 2, 5, 7, 8, 7: ");
    array_append(&a, 3); array_append(&a, 2); array_append(&a, 5);
    array_append(&a, 7); array_append(&a, 8); array_append(&a, 7);
    array_print(&a);
    printf("\nRemoving at position 2: "); array_remove(&a, 2); array_print(&a);
    printf("\nRemoving at position 4: "); array_remove(&a, 4); array_print(&a);
    printf("\nRemoving at position 2: "); array_remove(&a, 2); array_print(&a);
    printf("\nInserting 7 at position 1: ");
    array_insert(&a, 1, 7); array_print(&a); printf("\n");
    for (int e = 0; e <= 9; e += 2)
        printf("Has element %d ? %s\n", e,
            array_has_element(&a, e) ? "yes" : "no");
    array_delete(&a);
}
```

## Résultat:

```
Inserting 3, 2, 5, 7, 8, 7: [ 3 2 5 7 8 7 ]
Removing at position 2: [ 3 2 7 8 7 ]
Removing at position 4: [ 3 2 7 8 ]
Removing at position 2: [ 3 2 8 ]
Inserting 7 at position 1: [ 3 7 2 8 ]
Has element 0 ? no
Has element 2 ? yes
Has element 4 ? no
Has element 6 ? no
Has element 8 ? yes
```

## Tableaux multidimensionnels

# Tableau multidimensionnel

- Généralisation d'un tableau à **plusieurs dimensions**
- Aussi appelé **matrice**
- Simplifie la manipulation des **données** homogènes
- En les organisant selon leurs **dimensions**

## Version aplatie

- Tableau **unidimensionnel**
- **Plus** compact
- Mais on doit gérer l'**accès** (indexation)

## Par indirection

- Tableau de **pointeurs**
- **Moins** compact
- Mais plus facile de gérer l'indexation

# Interface (partielle)

```
// Initialize a matrix
void matrix_initialize(struct matrix *m,
                      unsigned int r,
                      unsigned int c,
                      bool random);

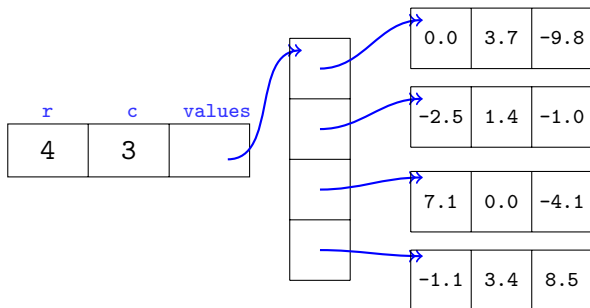
// Print a matrix to stdout
void matrix_print(const struct matrix *m);

// Add the matrix `m2` to the matrix `m1`
void matrix_add(struct matrix *m1, const struct matrix *m2);

// Delete the given matrix
void matrix_delete(struct matrix *m);
```

# Représentation

```
struct matrix {  
    unsigned int r; // Number of rows  
    unsigned int c; // Number of columns  
    double **values; // Values in matrix  
};
```



# Implémentation (1/2)

```
// Initialize a matrix with 0 or random values
void matrix_initialize(struct matrix *m,
                      unsigned int r,
                      unsigned int c,
                      bool random) {
    m->r = r;
    m->c = c;
    m->values = malloc(r * sizeof(double*));
    for (unsigned int i = 0; i < r; ++i) {
        m->values[i] = malloc(c * sizeof(double));
        for (unsigned int j = 0; j < c; ++j) {
            if (random) {
                m->values[i][j] = (float)rand() /
                                   (float)(RAND_MAX / 20.0) - 10.0;
            } else {
                m->values[i][j] = 0.0;
            }
        }
    }
}
```



## Implémentation (2/2)

```
// Print a matrix to stdout
void matrix_print(const struct matrix *m) {
    for (unsigned int i = 0; i < m->r; ++i) {
        printf("[ ");
        for (unsigned int j = 0; j < m->c; ++j) {
            printf("%.2lf ", m->values[i][j]);
        }
        printf("]\n");
    }
}

// Add the second matrix to the first one
void matrix_add(struct matrix *m1, const struct matrix *m2) {
    if (m1->r != m2->r || m1->c != m2->c) {
        fprintf(stderr, "Error: matrices have different dimensions\n");
        exit(1);
    }
    for (unsigned int i = 0; i < m1->r; ++i)
        for (unsigned int j = 0; j < m1->c; ++j)
            m1->values[i][j] += m2->values[i][j];
}

// Delete a matrix
void matrix_delete(struct matrix *m) {
    for (unsigned int i = 0; i < m->r; ++i)
        free(m->values[i]);
    free(m->values);
}
```

# Utilisation

```
int main(void) {
    srand(time(NULL));
    struct matrix m1, m2;
    printf("Initializing m1:\n");
    matrix_initialize(&m1, 3, 5, true); matrix_print(&m1);
    printf("Initializing m2:\n");
    matrix_initialize(&m2, 3, 5, true); matrix_print(&m2);
    printf("Adding m2 to m1:\n");
    matrix_add(&m1, &m2); matrix_print(&m1);
    matrix_delete(&m1); matrix_delete(&m2);
    return 0;
}
```

## Résultat (les valeurs peuvent varier):

Initializing m1:

[	0.01	8.16	-7.81	-5.44	-3.65	]
[	-2.18	-1.61	-5.82	-7.56	-1.02	]
[	4.41	-3.73	-7.61	0.10	8.28	]

Initializing m2:

[	-3.21	-1.22	-4.39	8.25	2.22	]
[	-1.98	-3.68	4.90	-6.46	6.50	]
[	6.21	-3.26	-8.34	-1.79	-3.21	]

Adding m2 to m1:

[	-3.20	6.94	-12.20	2.81	-1.43	]
[	-4.16	-5.29	-0.93	-14.02	5.48	]
[	10.62	-6.99	-15.95	-1.69	5.07	]

## Arbres binaires de recherche

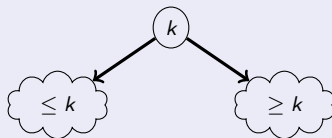
# Arbres binaires de recherche

## Arbre binaire

- Ensemble de noeuds organisés de façon **hiérarchique**
- Chaque noeud référence **deux** enfants
- Possiblement **vides**

## Arbre binaire de recherche (ABR)

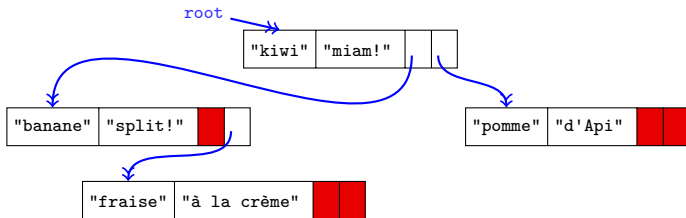
- Chaque noeud est identifié par une **clé**
- Choisie dans un ensemble de clés **totalelement ordonné**
- Et contient une **valeur**
- **Invariant:**



# Représentation

```
// A tree node
struct tree_node {
    char *key;           // Key
    char *value;         // Value
    struct tree_node *left; // Left subtree
    struct tree_node *right; // Right subtree
};
```

```
// A tree map
typedef struct {
    struct tree_node *root; // Root of tree
} treemap;
```



# Interface

```
// Initialize an empty tree map
void treemap_initialize(treemap *t);

// Return the value associated with the given key in a tree map
char *treemap_get(const treemap *t, const char *key);

// Set the value for the given key in a tree map
void treemap_set(treemap *t, const char *key, const char *value);

// Indicate if a key exists in a tree map
bool treemap_has_key(const treemap *t, const char *key);

// Print a tree map to stdout
void treemap_print(const treemap *t);

// Delete a tree map
void treemap_delete(treemap *t);
```

# Récupérer un noeud

```
struct tree_node *treemap_get_node(const struct tree_node *node,
                                   const char *key) {
    if (node == NULL) {
        return NULL;
    } else {
        int cmp = strcmp(key, node->key);
        if (cmp == 0)
            return (struct tree_node*)node;
        else if (cmp < 0)
            return treemap_get_node(node->left, key);
        else
            return treemap_get_node(node->right, key);
    }
}
```

# Insérer un noeud

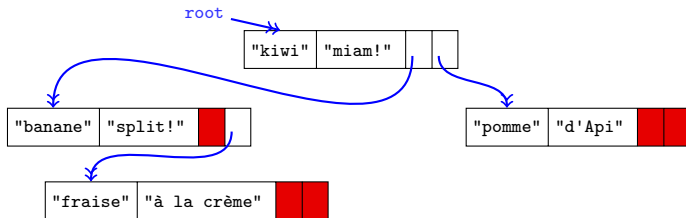
```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```

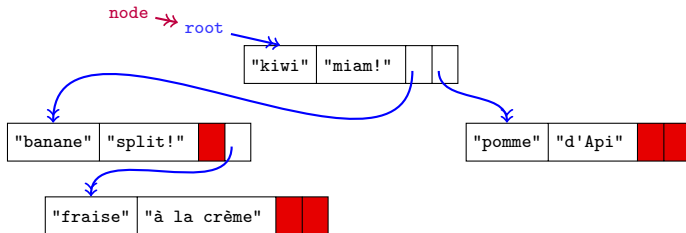
On souhaite insérer  
la clé melon



# Insérer un noeud

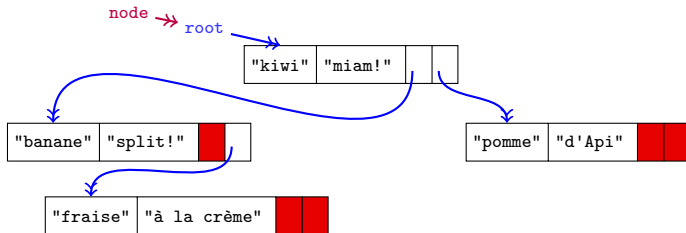
```
void treemap_insert_node(struct tree_node **node,  
                        const char *key,  
                        const char *value) {  
    if (*node == NULL) {  
        *node = malloc(sizeof(struct tree_node));  
        (*node)->key = strdup(key);  
        (*node)->value = strdup(value);  
        (*node)->left = NULL;  
        (*node)->right = NULL;  
    } else if (strcmp(key, (*node)->key) < 0) {  
        treemap_insert_node(&(*node)->left, key, value);  
    } else {  
        treemap_insert_node(&(*node)->right, key, value);  
    }  
}
```

On souhaite insérer  
la clé melon



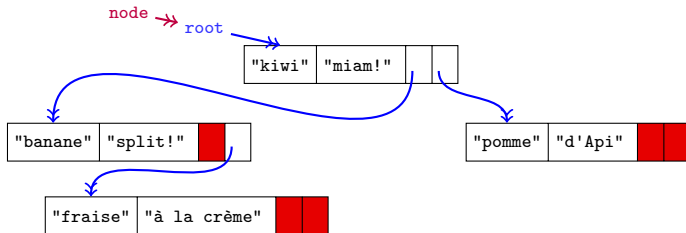
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,  
                        const char *key,  
                        const char *value) {  
    if (*node == NULL) {  
        *node = malloc(sizeof(struct tree_node));  
        (*node)->key = strdup(key);  
        (*node)->value = strdup(value);  
        (*node)->left = NULL;  
        (*node)->right = NULL;  
    } else if (strcmp(key, (*node)->key) < 0) {  
        treemap_insert_node(&(*node)->left, key, value);  
    } else {  
        treemap_insert_node(&(*node)->right, key, value);  
    }  
}
```



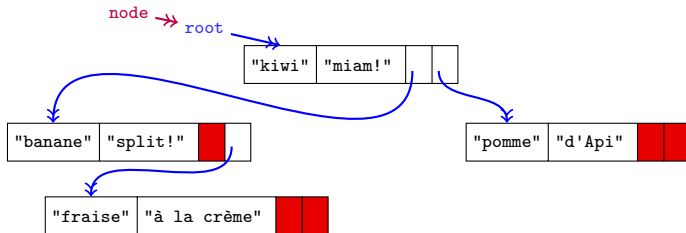
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



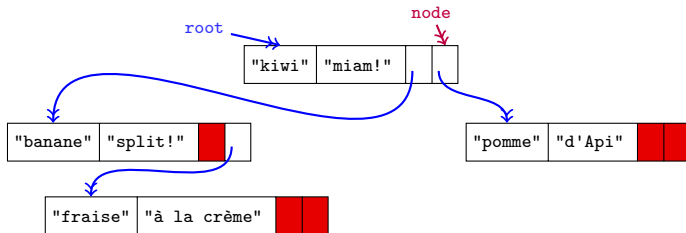
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



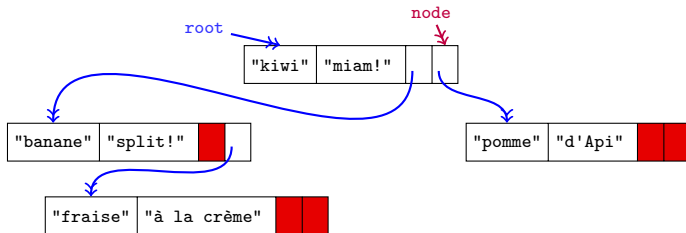
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,  
                        const char *key,  
                        const char *value) {  
    if (*node == NULL) {  
        *node = malloc(sizeof(struct tree_node));  
        (*node)->key = strdup(key);  
        (*node)->value = strdup(value);  
        (*node)->left = NULL;  
        (*node)->right = NULL;  
    } else if (strcmp(key, (*node)->key) < 0) {  
        treemap_insert_node(&(*node)->left, key, value);  
    } else {  
        treemap_insert_node(&(*node)->right, key, value);  
    }  
}
```



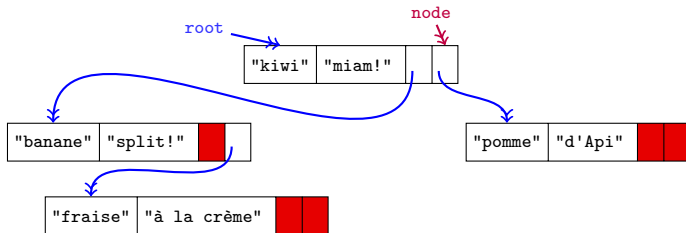
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,  
                        const char *key,  
                        const char *value) {  
    if (*node == NULL) {  
        *node = malloc(sizeof(struct tree_node));  
        (*node)->key = strdup(key);  
        (*node)->value = strdup(value);  
        (*node)->left = NULL;  
        (*node)->right = NULL;  
    } else if (strcmp(key, (*node)->key) < 0) {  
        treemap_insert_node(&(*node)->left, key, value);  
    } else {  
        treemap_insert_node(&(*node)->right, key, value);  
    }  
}
```



# Insérer un noeud

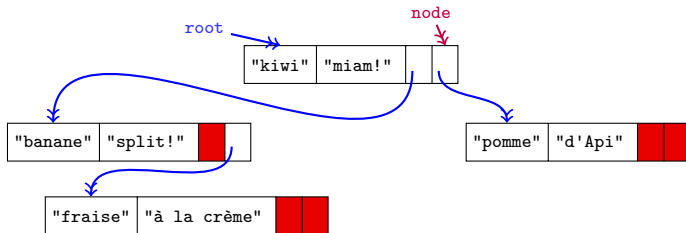
```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```





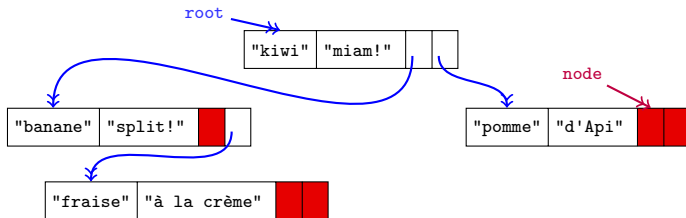
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



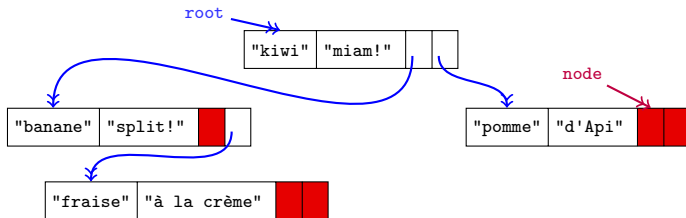
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,  
                        const char *key,  
                        const char *value) {  
    if (*node == NULL) {  
        *node = malloc(sizeof(struct tree_node));  
        (*node)->key = strdup(key);  
        (*node)->value = strdup(value);  
        (*node)->left = NULL;  
        (*node)->right = NULL;  
    } else if (strcmp(key, (*node)->key) < 0) {  
        treemap_insert_node(&(*node)->left, key, value);  
    } else {  
        treemap_insert_node(&(*node)->right, key, value);  
    }  
}
```



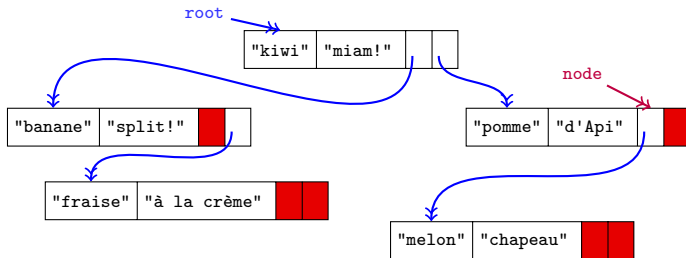
# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



# Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                        const char *key,
                        const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



# Implémentation (1/2)

```
void treemap_initialize(treemap *t) {
    t->root = NULL;
}

char *treemap_get(const treemap *t, const char *key) {
    struct tree_node *node = treemap_get_node(t->root, key);
    return node == NULL ? NULL : node->value;
}

bool treemap_has_key(const treemap *t, const char *key) {
    return treemap_get_node(t->root, key) != NULL;
}

void treemap_set(treemap *t, const char *key, const char *value) {
    struct tree_node *node = treemap_get_node(t->root, key);
    if (node != NULL) {
        free(node->value);
        node->value = strdup(value);
    } else {
        treemap_insert_node(&(t->root), key, value);
    }
}
```

## Implémentation (2/2)

```
void treemap_print(const treemap *t) {
    printf("TreeMap {\n");
    treemap_print_node(t->root);
    printf("}\n");
}

void treemap_print_node(const struct tree_node *node) {
    if (node != NULL) {
        treemap_print_node(node->left);
        printf("  %s: %s\n", node->key, node->value);
        treemap_print_node(node->right);
    }
}

void treemap_delete(treemap *t) {
    treemap_delete_node(t->root);
}

void treemap_delete_node(struct tree_node *node) {
    if (node != NULL) {
        treemap_delete_node(node->left);
        treemap_delete_node(node->right);
        free(node->key);
        free(node->value);
        free(node);
    }
}
```

# Utilisation

```
int main() {
    treemap t;
    treemap_initialize(&t);
    treemap_set(&t, "firstname", "Doina"); treemap_set(&t, "lastname", "Precup");
    treemap_set(&t, "city", "Montreal"); treemap_set(&t, "province", "Quebec");
    treemap_set(&t, "country", "Canada"); treemap_set(&t, "position", "DeepMind");
    printf("Printing the tree map\n"); treemap_print(&t);
    printf("Get \"firstname\": %s\n", treemap_get(&t, "firstname"));
    printf("Get \"province\": %s\n", treemap_get(&t, "province"));
    printf("Get \"position\": %s\n", treemap_get(&t, "position"));
    printf("Changing country to Romania\n"); treemap_set(&t, "country", "Romania");
    printf("Get \"country\": %s\n", treemap_get(&t, "country"));
    printf("Printing the tree map\n"); treemap_print(&t);
    treemap_delete(&t);
}
```

## Résultat:

```
Printing the tree map
TreeMap {
  city: Montreal
  country: Canada
  firstname: Doina
  lastname: Precup
  position: DeepMind
  province: Quebec
}
Get "firstname": Doina
Get "province": Quebec
Get "position": DeepMind
```

```
Changing country to Romania
Get "country": Romania
Printing the tree map
TreeMap {
  city: Montreal
  country: Romania
  firstname: Doina
  lastname: Precup
  position: DeepMind
  province: Quebec
}
```