Partie 6

Structures de données

19 novembre 2019

Plan

1. Introduction

2. Pile et file

3. Arbre

4. Dictionnaire

Plan

1. Introduction

2. Pile et file

3. Arbre

4. Dictionnaire

Types de données abstraits

Un type de données abstrait définit :

- Un ensemble de données
- Un ensemble d'opérations sur ces données

Types d'opérations standards :

- Création, destruction d'un objet du type donné
- Accès aux données
- Modification des données
  - Insertion et suppression de nouvelle données si l'ensemble est dynamique

Exemples jusqu'ici : nombres complexes, matrices, grilles (cf. projet 1).

Types de données abstraits : implémentation

On implémente concrètement un type de données abstrait en utilisant une structure de données.

Une structure de données consiste en :

- une représentation des données
- une représentation des relations entre ces données

Exemples : tableaux 1D, 2D, liste liée, arbres, graphes...

Pour un même TDA, plusieurs implémentations (structures de données) sont généralement possibles.

On analyse les performances d'une structure particulière selon deux critères :

- Complexité en temps des opérations
- Complexité en espace nécessaire pour la structure

Exprimées dans le pire cas en fonction de la quantité de données présentes dans la structure.

Types de données abstraits : en C (rappel)

Fichier d'entête (`.h`) contient les prototypes des opérations et la définition du type (opaque). Le fichier source (`.c`) contient la définition concrète de la structure et l'implémentation des opérations

```c
// complex.h

#ifndef _COMPLEXE_H
#define _COMPLEXE_H

// définition du nouveau type

typedef struct complex_t complex;

// prototypes des fonctions

complex *complex_new(double, double);
void complex_destroy(complex *);
void complex_sum(complex *, complex *);
void complex_product(complex *, complex *);
...

#endif
```

```c
// complex.c
#include <math.h>
#include "complex.h"

struct complex_t {
  double re, im;
};

complex *complex_new(double re, double im) {
  ...
}
void complex_sum(complex *a, complex *b) {
  ...
}
void complex_product(complex *a, complex *b) {
  ...
}
...
```

On utilise des pointeurs sur void (et éventuellement des pointeurs de fonction) si on veut pouvoir stocker des valeurs arbitraires dans la structure.

TDA standard

Depuis plus de 50 ans, plusieurs TDA standards, utiles dans de nombreuses applications, ont été définis et étudiés dans la littérature.

Principalement des ensemble de données dynamiques.

Quelques exemples :

- Pile et file : collection d'objets accessibles selon un politique LIFO/FIFO.
- File à priorité : collection d'objets accessibles selon un ordre de priorité.
- Liste : séquence d'objets accessibles à partir de leur position relative.
- Arbre : collection de valeurs associées au nœuds d'un arbre
- Dictionnaire : collection d'objets accessibles de manière arbitraire via une clé.
- Graphe : collection de valeurs associées aux nœuds d'un graphe et accessible selon ce graphe.

Plan

1. Introduction

2. Pile et file
   - Principe et applications
   - Implémentation par tableau
   - Listes liées
   - Implémentation par liste liée

3. Arbre

4. Dictionnaire

Plan

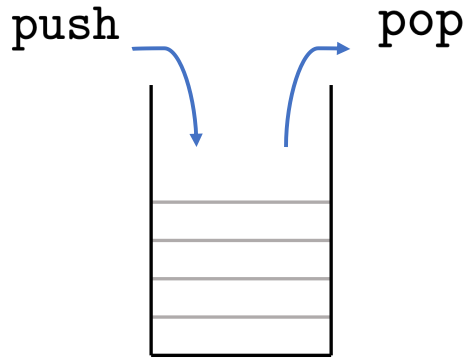
1. Introduction

2. Pile et file
   - Principe et applications
   - Implémentation par tableau
   - Listes liées
   - Implémentation par liste liée

3. Arbre

4. Dictionnaire

Pile



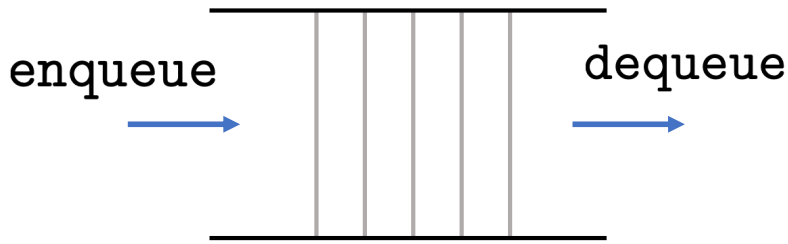
Une pile est une collection de valeurs, accessibles selon une discipline LIFO (Last In First Out)

Interface :

- `push(s,v)` : ajoute la valeur `v` au sommet de la pile `s`
- `pop(s)` : retire la valeur au sommet de la pile `s`, et retourne cette valeur. Signale une erreur si la pile est vide.
- `top(s)` : retourne la valeur présente au sommet de la pile `s`, sans la dépiler. Signale une erreur si la pile est vide.
- `size(s)` : retourne le nombre de valeurs présentes dans la pile `s`.
- `isEmpty(s)` : détermine si la pile `s` est vide.
- + création de la structure et libération de la mémoire

File

Une file est une collection de valeurs, accessibles selon une discipline FIFO (First In First Out)



Interface :

- `enqueue(q,v)` : ajoute la valeur `v` à la fin de la file `q`
- `dequeue(q)` : retire la valeur au début de la file `q` et la retourne. Signale une erreur si la file est vide.
- `front(q)` : retourne la valeur présente au début de la file `q`, sans la retirer. Signale une erreur si la file est vide.
- `size(q)` : retourne le nombre de valeurs présentes dans la file `q`.
- `isEmpty(q)` : détermine si la file `q` est vide.
- + création de la structure et libération de la mémoire

Exemple d'interface en C

stack.h

```c
#ifndef _STACK_H
#define _STACK_H

typedef struct Stack_t Stack;

Stack *stackCreate();
void   stackFree(Stack *);
int    stackPush(Stack *, void *);
void  *stackPop(Stack *);
void  *stackTop(Stack *);
int    stackSize(Stack *);
int    stackIsEmpty(Stack *);

#endif
```

queue.h

```c
#ifndef _QUEUE_H
#define _QUEUE_H

typedef struct Queue_t Queue;

Queue *queueCreate();
void   queueFree(Queue *);
void   queueEnqueue(Queue *, void *);
void  *queueDequeue(Queue *);
void  *queueHead(Queue *);
int    queueSize(Queue *);
int    queueIsEmpty(Queue *);

#endif
```

Les données sont stockées dans la pile/file sous la forme de pointeurs sur `void`

- type de retour des fonctions `stackPop`, `stackTop`, `queueDequeue` et `queueHead` et type du deuxième argument des fonctions `stackPush` et `queueEnqueue`).

Applications

Ces deux structures fondamentales ont de nombreuses applications.

Piles

- Allocation de resources selon un politique “dernier arrivé premier servi”
- Mécanisme d’appels de fonctions dans les langages de programmation
- Implémentation des compilateurs
- Opération undo/back dans certains applications.
- Parcours en profondeur d’abord d’un graphe

Files

- Gestion de resources selon une politique “premier arrivé, premier servi”
- Transfert de données asynchrone (gestion des opérations printf)
- Gestion de requêtes sur un serveur
- Simulation de files d’attente dans la vie réelle
- Parcours en largeur d’abord d’un graphe.

Une application de la file

On aimerait trier les adresses emails d'une liste en vue d'implémenter le filtre de notre serveur d'emails.

Les adresses se trouvent dans un fichier `addresses.txt` (une adresse par ligne) et on aimerait les placer dans un tableau pour le tri.

Solution sur base d'une file :

- Lire les adresses une par une en les stockant dans une file
- Créer un tableau de la même taille que la file
- Retirer les adresses une par une de la file en les stockant dans le tableau

Peut-on utiliser une pile ?

Une application de la file : en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

const int BUFFER_SIZE = 1000;
int main() {

  char buffer[BUFFER_SIZE];
  // lecture du fichier (sur l'entrée) et stockage dans une file
  Queue *q = queueCreate();
  while (fgets(buffer, BUFFER_SIZE, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';   // supprime la fin de ligne

    queueEnqueue(q, strdup(buffer));
  }
  // Creation et remplissage du tableau
  int sizeQ = queueSize(q);
  char **array = malloc(sizeQ * sizeof(char *));
  for (int i = 0; i<sizeQ; i++)
    array[i] = (char *) queueDequeue(q);

  queueFree(q);
  ...
}
```

Une application de la file : en C

Remarques :

- `char *fgets(char *str, int size, FILE *fp)` : lit une ligne dans le fichier `fp` (qui peut être l'entrée standard) et place le résultat dans la chaîne `str`. La chaîne est terminée par le caractère de fin de ligne (`'\n'`) et le caractère NUL (`'\0'`). Si la ligne fait plus de `size` caractères, seulement les `size-1` premiers sont lus pour éviter une dépassement de `str`. Renvoie `str` si tout s'est bien passé.
- `size_t strlen(char *str)` : renvoie la longueur de la chaîne (caractère NUL non compris).
- `char *strdup(const chart *src)` : copie la chaîne `src` dans une nouvelle chaîne (en allouant la mémoire nécessaire) et retourne un pointeur vers cette chaîne.

Une application de la pile

La manière standard d'écrire une expression mathématique est la notation infixe, basée sur les parenthèses pour la précédence :

- Exemple : $((3 + 4) * 5) - 8$ $(= 27)$

Notation postfixe (ou polonaise inversée) : les opérateurs suivent les opérandes.

- Exemple : 3 4 $+$ 5 $*$ 8 $-$

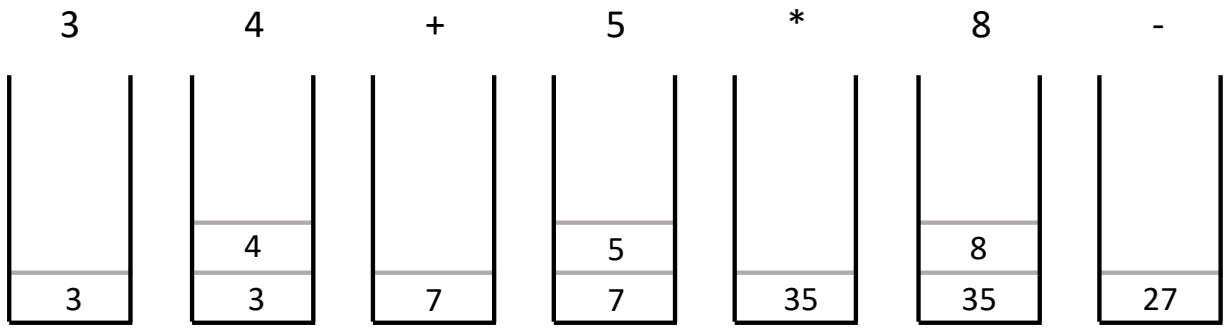
Aucune parenthèse n'est nécessaire dans cette notation : il n'y a qu'une seule manière de mettre des parenthèses.

Les expressions en notation postfixe sont faciles à évaluer en se basant sur une pile.

Evaluation sur base d'une pile

Principe de l'évaluation :

- Si on lit un nombre, on le met (`push`) sur la pile.
- Si on lit un opérateur (binaire) : on retire le second puis le premier opérandes sur la pile (`pop`), on leur applique l'opérateur et on met le résultat sur la pile.



Evaluation d'expression en notation postfixe

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack-double.h"

const int BUFFER_SIZE = 1000;

int main() {

  char buffer[BUFFER_SIZE];
  Stack *s = stackCreate();

  while (fgets(buffer, BUFFER_SIZE, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';

    if (strcmp(buffer, "+") == 0)
      stackPush(s, stackPop(s)+stackPop(s));
    else if (strcmp(buffer, "-") == 0)
      stackPush(s, -stackPop(s)+stackPop(s));
    else if (strcmp(buffer, "/") == 0)
      stackPush(s, stackPop(s)/stackPop(s));
    else if (strcmp(buffer, "*") == 0)
      stackPush(s, stackPop(s)*stackPop(s));
    else {
      stackPush(s, strtod(buffer, NULL));
    }
  }

  printf("Result = %f\n",stackPop(s));
}
```

```
> gcc -o rpn rpn-evaluation.c stack-double.c
> ./rpn
3
4
+
5
*
8
-
Result = 27.000000
```

Plan

1. Introduction

2. Pile et file
   - Principe et applications
   - Implémentation par tableau
   - Listes liées
   - Implémentation par liste liée

3. Arbre

4. Dictionnaire

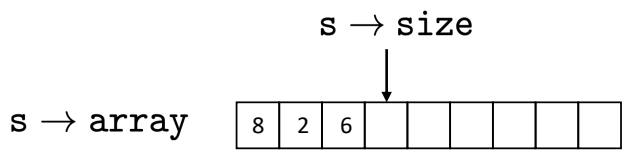
Implémentation par tableau d'une pile : principe

Principes :

- Les valeurs contenues dans la pile sont placées dans les composantes successives d'un tableau
- Un indice `size` contient la taille de la pile, qui est aussi la première position libre dans le tableau

La taille du tableau doit être fixée a priori et une erreur signalée lorsque la taille de la pile excède la taille du tableau.

```
const int  MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};
```



Implémentation par tableau d'une pile : code C

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

const int MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
Stack *stackCreate() {
  Stack *s = malloc(sizeof(Stack));
  if (!s)
    terminate("Stack can not"\
              "be created");
  s -> size = 0;
  return s;
}

void stackFree(Stack *s) {
  free(s);
}
```

Implémentation par tableau d'une pile : code C

Accès et insertion.

```
void stackPush(Stack *s, void *data) {
  if (s -> size >= MAX_STACK_SIZE)
    terminate("Maximum stack size"\
              "reached");

  s -> array[s -> size++] = data;
}

void *stackTop(Stack *s) {

  if (s -> size == 0)
    terminate("Stack is empty");

  return s -> array[s -> size - 1];
}
```

```
void *stackPop(Stack *s) {
  if (s -> size == 0)
    terminate("Stack is empty");

  return s -> array[--(s -> size)];
}


int stackSize(Stack *s) {
  return s -> size;
}

int stackIsEmpty(Stack *s) {
  return (s -> size == 0);
}
```

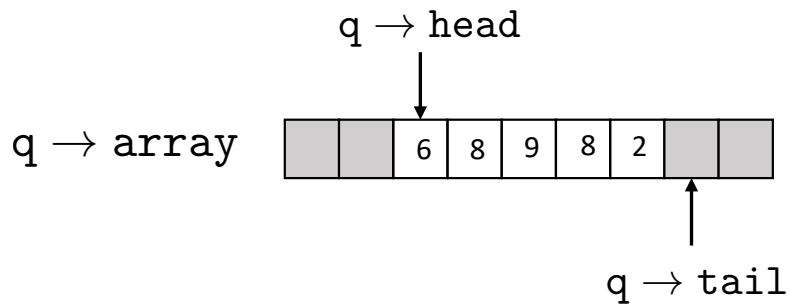
Implémentation par tableau d'une file : principe

Principes :

- Les valeurs contenues dans la file sont placées dans les composantes successives d'un tableau.
- Un indice `head` (resp. `tail`) indique la valeur en tête (resp. en queue) de file.
- La tableau est géré de manière circulaire.

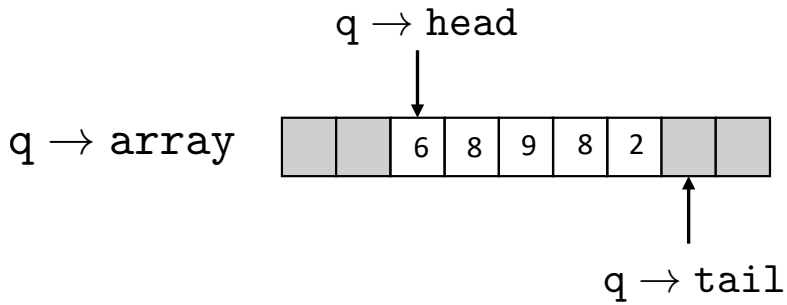
La taille du tableau doit être fixée a priori et une erreur signalée lorsque la taille de la file excède la taille du tableau.

```
const int  MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};
```

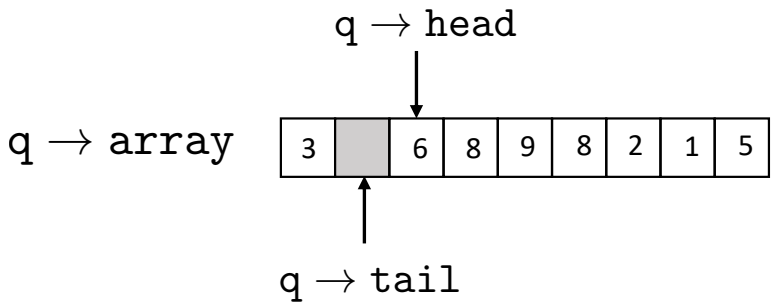


Illustration

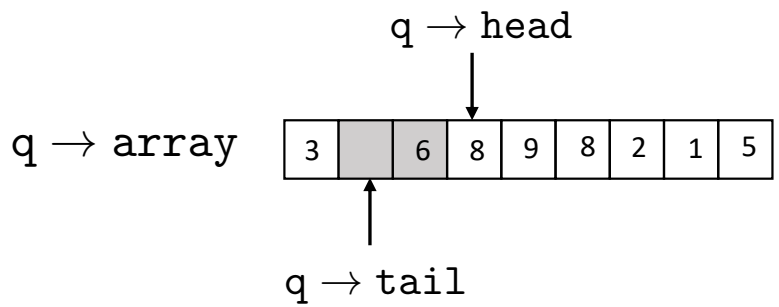
File initiale :



`queueEnqueue(q,1)`, `queueEnqueue(q,5)`, `queueEnqueue(q,3)`



`queueDequeue(q)` $\Rightarrow 6$



Implémentation par tableau d'une file : code C

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "queue.h"

const int MAX_QUEUE_SIZE = 1000;

struct Queue_t {
  void *array[MAX_QUEUE_SIZE];
  int head, tail;
};

static void terminate(const char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
Queue *queueCreate() {
  Queue *q = malloc(sizeof(Queue));
  if (!q)
    terminate("Queue can not"\
              "be created");
  q -> head = 0;
  q -> tail = 0;
  return q;
}

void queueFree(Queue *q) {
  free(q);
}
```

Implémentation par tableau d'une file : code C

Accès et insertion.

```
void queueEnqueue(Queue *q, void *data) {
  if (queueSize(q) >=
                    MAX_QUEUE_SIZE - 1)
    terminate("Queue is full");

  q -> array[q -> tail] = data;
  q -> tail = (q -> tail + 1)
              % MAX_QUEUE_SIZE;
}

void *queueHead(Queue *q) {
  if (q -> tail == q -> head)
    terminate("Queue is empty");
  return q -> array[q -> head];
}
```

```
void *queueDequeue(Queue *q) {
  if (q -> tail == q -> head)
    terminate("Queue is empty");

  void *data = q -> array[q -> head];

  q -> head = (q -> head + 1)
              % MAX_QUEUE_SIZE;

  return data;
}

int queueSize(Queue *q) {
  return (MAX_QUEUE_SIZE + q -> tail
          - q -> head) % MAX_QUEUE_SIZE
}

int queueIsEmpty(Queue *q) {
  return (q -> head == q -> tail);
}
```

Complexité en temps et en espace

La complexité en temps de toutes les opérations est $O(1)$

La complexité en espace est en fait $O(1)$ si la pile/file contient n valeurs, qui exprime que la taille de la structure ne dépend pas de la quantité de données qui y est stockée.

Deux inconvénients :

- Il y a une limite sur le nombre de valeurs qu'on peut stocker dans la structure
- Utiliser un tableau de taille fixe entraîne un gaspillage en terme de mémoire

Pour résoudre ce problème, il faut utiliser une structure de données dynamique $\Rightarrow$ la liste liée

Plan

1. Introduction

2. Pile et file
   - Principe et applications
   - Implémentation par tableau
   - Listes liées
   - Implémentation par liste liée

3. Arbre

4. Dictionnaire

Liste (simplement) liée

Structure de données composée d'une séquence d'éléments de liste.

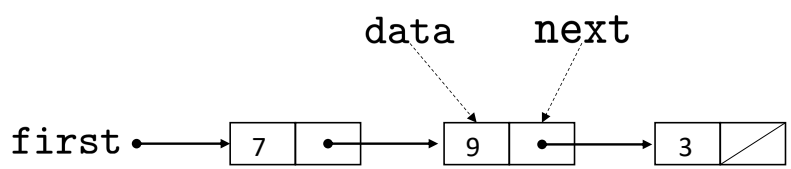
Chaque élément de liste (aussi appelé un nœud) est composé :

- d'un contenu utile de type arbitraire (les valeurs qu'on souhaite stocker dans la structure)
- d'un pointeur vers l'élément suivant dans la séquence (NULL si l'élément est le dernier de la liste)

Une liste liée est un pointeur vers le premier élément de la liste.

Exemple d'une liste liée d'entiers :

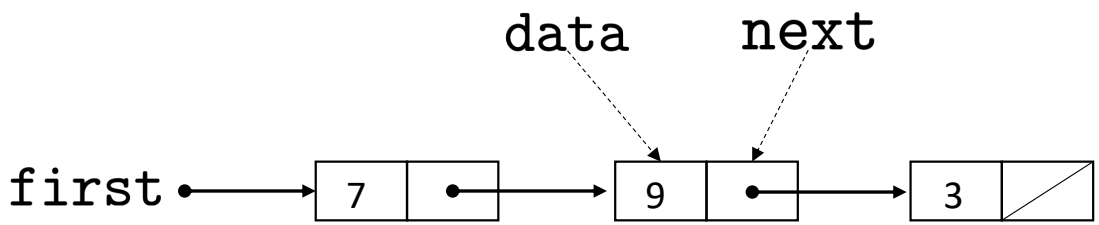
```
typedef struct Node_t {
  int data;
  struct Node_t *next;
} Node;
```



Manipulation d'une liste liée

Construction d'une liste :

```
Node *n1 = malloc(sizeof(Node));
Node *n2 = malloc(sizeof(Node));
Node *n3 = malloc(sizeof(Node));

n1 -> data = 7;
n1 -> next = n2;
n2 -> data = 9;
n2 -> next = n3;
n3 -> data = 3;
n3 -> next = NULL;

Node *first = n1;
```

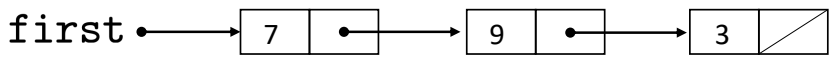


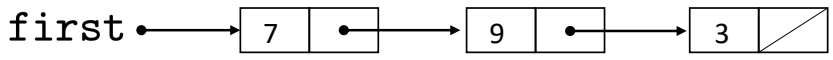
Manipulation d'une liste liée

Extraction du premier élément

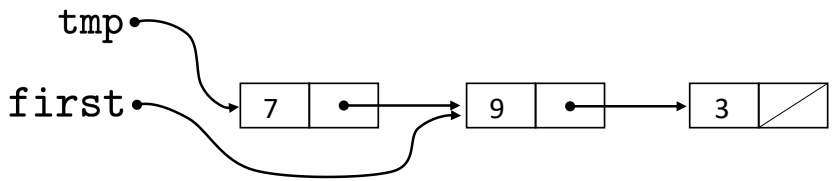
```
int value = first -> data;

Node *tmp = first;
first = first -> next;

free(tmp);
return value;
```

value
7

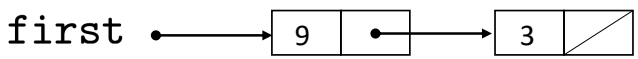




value
7



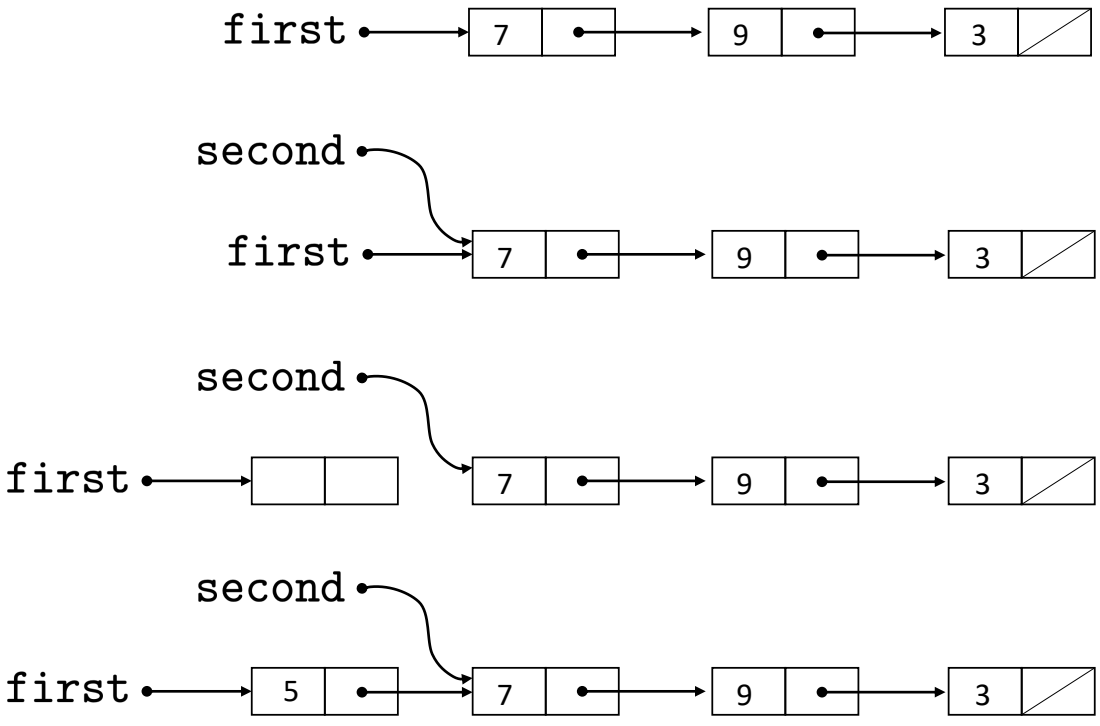
value
7



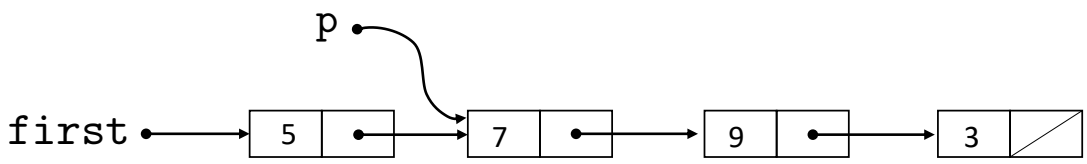
Manipulation d'une liste liée

Ajout d'un élément en début de liste

```
Node *second = first;

first = malloc(sizeof(Node));

first -> data = value;
first -> next = second;
```



Manipulation d'une liste liée : traverser la liste




```
Node *p = first;
while (p != NULL) {
  printf("%d\n", p -> data);
  p = p -> next;
}
```

Sortie :

```
5
7
9
3
```

Liste liée versus tableau

Liste liée et tableau peuvent tous deux représenter une séquence de valeurs.

Liste liée :

- Accès relatif uniquement aux éléments de la séquence (via pointeur `next`)
- Taille dépend directement (linéairement) du nombre d'élements
- Insertion aisée ($O(1)$) de valeurs au milieu de la séquence

Tableau :

- Accès direct aux éléments en fonction de leur rang
- Taille fixée à priori
- Insertion compliquée ($O(n)$) de valeurs au milieu de la séquence.

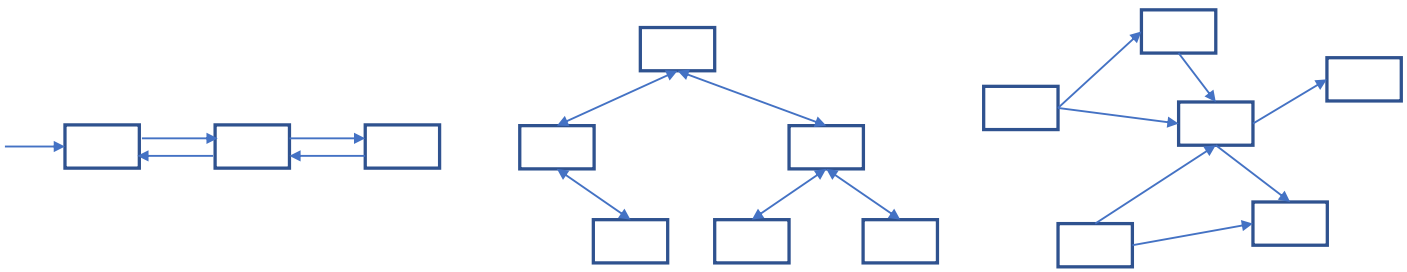
Généralisation : structures liées

Un seul pointeur (`next`) par nœud permet de représenter déjà pas mal de structures, au delà d'une simple séquence.

Le concept peut néanmoins se généraliser facilement en rajoutant d'autres pointeurs.

Exemples :

- Liste doublement liée : pointeur `previous` vers l'élément précédent. Facilite certaines opérations.
- Arbre binaire : pointeurs vers le parent, le fils gauche et le fils droit (voir plus loin).
- Graphe : pointeur vers chaque nœud adjacent.



Plan

1. Introduction

2. Pile et file
   - Principe et applications
   - Implémentation par tableau
   - Listes liées
   - Implémentation par liste liée

3. Arbre

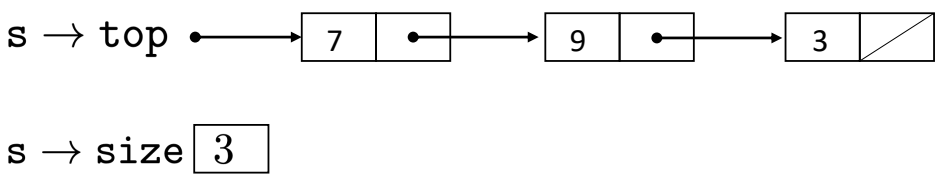
4. Dictionnaire

Implémentation d'une pile par liste liée

Principe :

- Les valeurs contenues dans la pile sont retenues dans une liste liée.
- L'opération `push` place la valeur en tête de liste. Les opérations `pop` et `top` travaillent en tête de liste également.
- Un indice `size` contient la taille de la pile.

```
struct Node_t {
  void           *data;
  struct Node_t  *next;
};

struct Stack_t {
  Node *top;
  int  size;
};
```



Implémentation d'une pile par liste liée

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

typedef struct Node_t {
  void           *data;
  struct Node_t *next;
} Node;

struct Stack_t {
  Node *top;
  int   size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
Stack *stackCreate() {
  Stack *s = malloc(sizeof(Stack));
  if (!s)
    terminate("Stack can not"\
              "be created");
  s -> top = NULL;
  s -> size = 0;
  return s;
}

void stackFree(Stack *s) {
  Node *n = s -> top;
  while (n) {
    Node *nNext = n -> next;
    free(n);
    n = nNext;
  }
  free(s);
}
```

Implémentation d'une pile par liste liée

Accès et insertion.

```
void stackPush(Stack *s,
               void *data) {
  Node *n = malloc(sizeof(Node));

  if (!n)
    terminate("Stack node can "\
              "not be created");

  n -> data = data;
  n -> next = s -> top;
  s -> top = n;
  s -> size++;
}

void *stackTop(Stack *s) {

  if (!(s -> top))
    terminate("Stack is empty");

  return s -> top -> data;
}
```

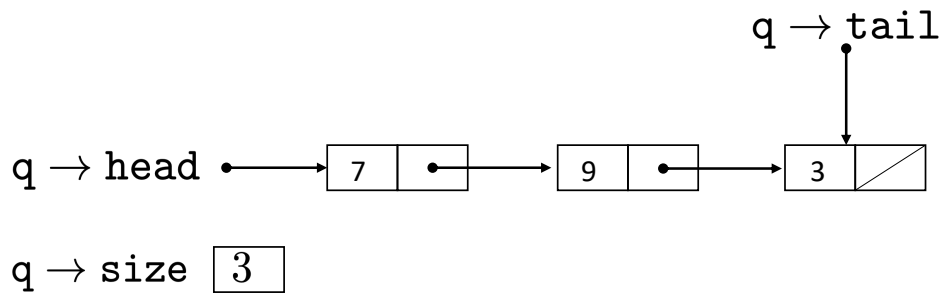
```
void *stackPop(Stack *s) {
  if (!(s -> top))
    terminate("Stack is empty");

  Node *n = s -> top;
  void *data = n -> data;

  s -> top = n -> next;
  s -> size--;

  free(n);

  return data;
}

int stackSize(Stack *s) {
  return s -> size;
}

int stackIsEmpty(Stack *s) {
  return (s -> size == 0);
}
```

Implémentation d'une file par liste liée

Principe :

- Les valeurs contenues dans la file sont retenues dans une liste liée.
- L'opération `enqueue` place les nouvelles valeurs en fin de liste, l'opération `dequeue` retire les valeurs en début de liste.
- Des pointeurs `head` et `tail` indiquent resp. le début et la fin de la liste.
- Un indice `size` contient la taille de la pile.

```
typedef struct Node_t {
  void          *data;
  struct Node_t *next;
} Node;

struct Queue_t {
  Node *head;
  Node *tail;
  int   size;
};
```



Implémentation d'une file par liste liée

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "queue.h"

typedef struct Node_t {
  void          *data;
  struct Node_t *next;
} Node;

struct Queue_t {
  Node *head;
  Node *tail;
  int   size;
};

static void terminate(char *m) {
  printf("%s\n", m);
  exit(EXIT_FAILURE);
}
```

```c
Queue *queueCreate() {
  Queue *q = malloc(sizeof(Queue));
  if (!q)
    terminate("Queue can not"\
              "be created");
  q -> head = NULL;
  q -> tail = NULL;
  q -> size = 0;
  return q;
}

void queueFree(Queue *q) {
  Node *n = q -> head;
  while (n) {
    Node *nNext = n -> next;
    free(n);
    n = nNext;
  }
  free(q);
}
```

Implémentation d'une file par liste liée

Accès et insertion.

```
void queueEnqueue(Queue *q,
                  void *data) {
  Node *n = malloc(sizeof(Node));
  if (!n)
    terminate("Queue node can"\
              "not be created");
  n -> data = data;
  n -> next = NULL;

  if (q -> tail)
    q -> tail -> next = n;
  else
    q -> head = n;
  q -> tail = n;

  q -> size++;
}

void *queueHead(Queue *q) {
  if (!(q -> head))
    terminate("Queue is empty");
  return q -> head -> data;
}
```

```
void *queueDequeue(Queue *q) {
  if (!(q -> head))
    terminate("Queue is empty");

  Node *n = q -> head;
  void *data = n -> data;

  q -> head = n -> next;
  q -> size--;
  if (q -> size == 0)
    q -> tail = NULL;
  free(n);
  return data;
}

int queueSize(Queue *q) {
  return q -> size;
}

int queueIsEmpty(Queue *q) {
  return (q -> size == 0);
}
```

Implémentation par liste liée : complexité

Complexité en temps est $O(1)$ pour toutes les opérations, comme pour la représentation par tableau.

Complexité en espace est $O(n)$ si la pile/file contient n éléments.

Il n'y a plus de limite a priori sur la taille de la pile/file.

Plan

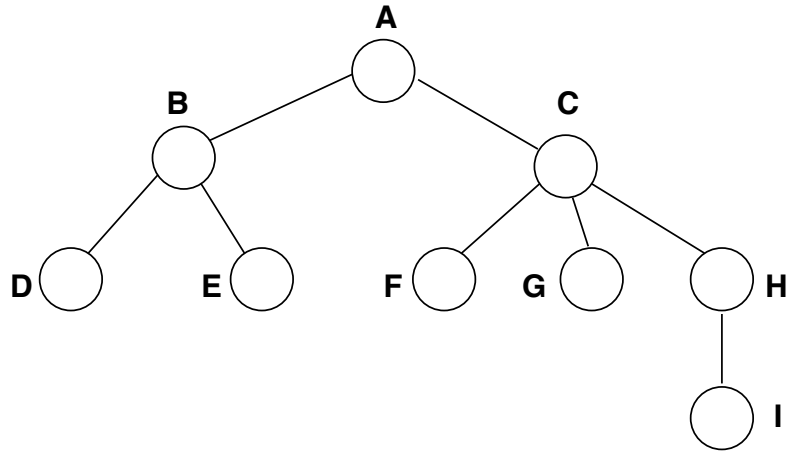
1. Introduction

2. Pile et file

3. Arbre
   - Définitions et opérations
   - Implémentation
   - Parcours

4. Dictionnaire

Arbre : définitions



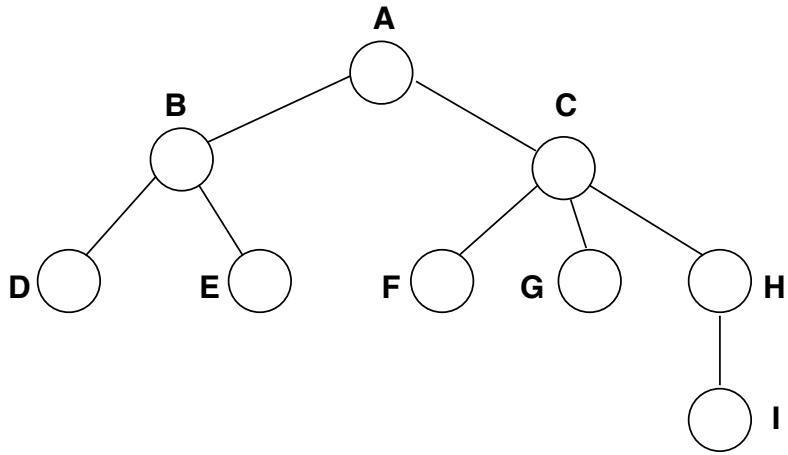
Un arbre (enraciné) T est un graphe dirigé (N, E), où :

- N est un ensemble de nœuds, et
- $E \subset N \times N$ est un ensemble d'arêtes,

possédant les propriétés suivantes :

- T est connexe et acyclique
- Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
- Pour toute arête $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
  - La racine de T ne possède pas de parent.
  - Les autres nœuds de T possèdent un et un seul parent.

Arbres : définitions



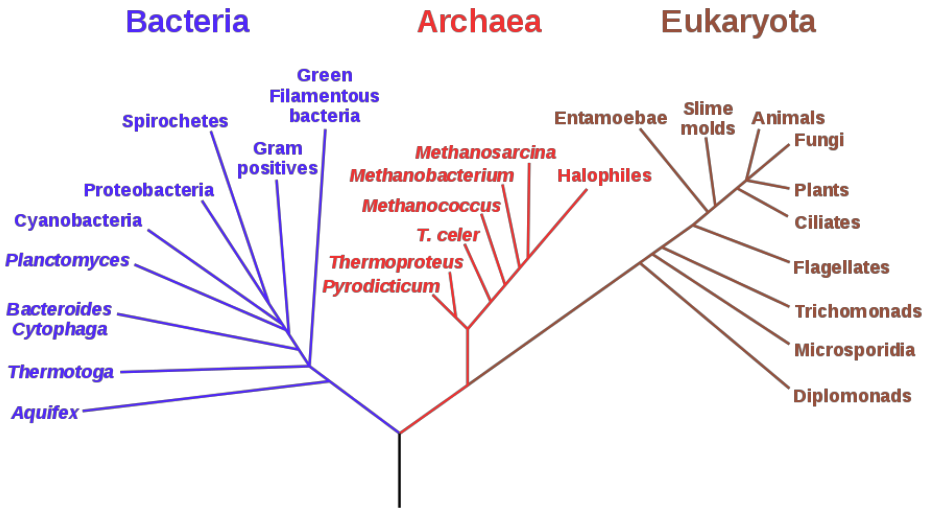
- Si n_2 est le parent de n_1, alors n_1 est le fils (*child*) de n_2.
- Deux nœuds n_1 et n_2 qui possèdent le même parent sont des frères (*siblings*).
- Un nœud qui possède au moins un fils est un nœud interne.
- Un nœud externe (c'est-à-dire, non interne) est une feuille (*leaf*) de l'arbre.

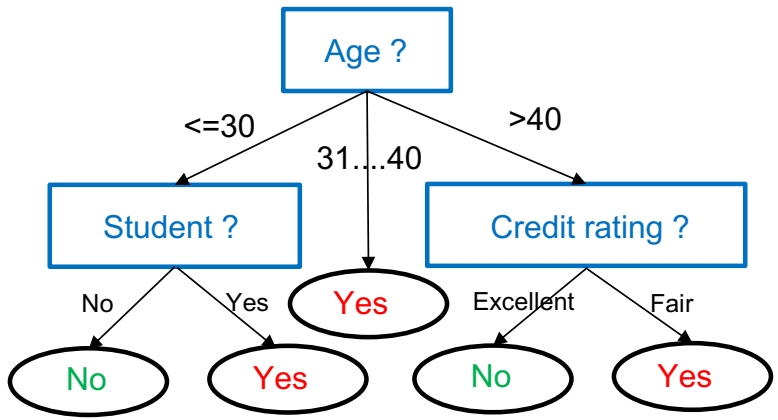
Arbre : structure de données

- Principe :
  - Des données sont associées aux nœuds d'un arbre
  - Les nœuds sont accessibles les uns par rapport aux autres selon leur position dans l'arbre
- Interface : Pour un arbre T et un nœud n
  - $\textsc{Parent}(T, n)$: renvoie le parent d'un nœud n (signale une erreur si n est la racine)
  - $\textsc{isEmpty}(T)$: renvoie vrai si l'arbre est vide
  - $\textsc{Children}(T, n)$: renvoie une structure de données (ordonnée ou non) contenant les fils du nœud n (liste, tableau, etc.)
  - $\textsc{isRoot}(T, n)$: renvoie vrai si n est la racine de l'arbre
  - $\textsc{isInternal}(T, n)$: renvoie vrai si n est un nœud interne
  - $\textsc{isExternal}(T, n)$: renvoie vrai si n est un nœud externe
  - $\textsc{GetData}(T, n)$: renvoie les données associées au nœud n
  - $\textsc{Root}(T)$: renvoie le nœud racine de l'arbre
  - Pour un arbre binaire (ordonné) :
    - $\textsc{Left}(T, n)$, $\textsc{Right}(T, n)$: renvoie les fils gauche et droit de n
    - $\textsc{hasLeft}(T, n)$, $\textsc{hasRight}(T, n)$: détermine si le nœud n possède un fils respectivement gauche et droit.

Arbre : applications

Beaucoup de données sont représentées de manière hiérarchique : arbres généalogique, arbres de décision, arbre phylogénétique, etc.





Les arbres sont utilisé comme structures de données pour de nombreux algorithmes : tri par tas, files à propriété, arbres binaires de recherche, arbre de couverture minimale, etc (voir INFO0902).

Exemple d'interface en C (arbre binaire)

BTree.h

```
#ifndef _BTREE_H
#define _BTREE_H

typedef struct BTNode_t BTNode;
typedef struct BTree_t BTree;

BTree  *btCreate():
void    btFree(BTree *tree);
BTNode *btCreateRoot(BTree *tree, int data);
BTNode *btInsertLeft(BTree *tree, BTNode *n1, int data);
BTNode *btInsertRight(BTree *tree, BTNode *n2, int data);

BTNode *btRoot(BTree *tree);
BTNode *btLeft(BTree *tree, BTNode *n);
BTNode *btRight(BTree *tree, BTNode *n);
BTNode *btParent(BTree *tree, BTNode *n);
int     btGetData(BTree *tree, BTNode *n);
int     btSize(BTree *tree);

int     btIsRoot(BTree *tree, BTNode *n);
int     btIsInternal(BTree *tree, BTNode *n);
int     btIsExternal(BTree *tree, BTNode *n);
int     btHasLeft(BTree *tree, BTNode *n);
int     btHasRight(BTree *tree, BTNode *n);
#endif
```

Exemples d'opération sur un arbre

Calcul de la profondeur d'un nœud (sa distance, en nombre d'arêtes, à la racine)

Récursivement :

```
int btNodeDepth(BTree *tree, BTNode *n) {
  if (btIsRoot(tree,n))
     return 0;
  else
     return 1+btNodeDepth(tree, btParent(tree, n));
}
```

Itérativement :

```
int btNodeDepth(BTree *tree, BTNode *n) {
  int depth = 0;
  while (!btIsRoot(tree,n)) {
     n = btParent(tree, n);
     depth++;
  }
  return depth;
}
```

Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Exemples d'opération sur un arbre

Calcul de la hauteur de l'arbre (la plus longue distance, en nombre d'arêtes, d'un nœud à la racine)

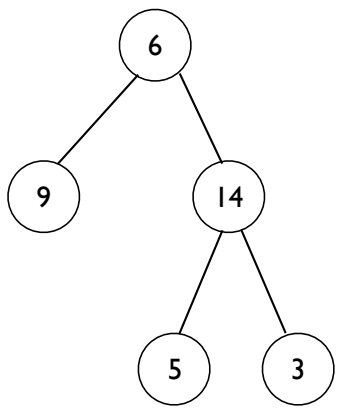
```
int btHeight(BTree *tree) {
   return btHeightAux(tree, btRoot(tree));
}

int btHeightAux(BTree *tree, BTNode *n) {
  if (btIsExternal(tree, n))
     return 0;
  else {
    int hl = 0;
    int hr = 0;
    if (btHasLeft(tree, n))
      hl = btHeightAux(tree, btLeft(tree, n));
    if (btHasRight(tree, n))
      hr = btHeightAux(tree, btRight(tree, n));
    return 1 + (hl>hr? hl: hr);
  }
}
```

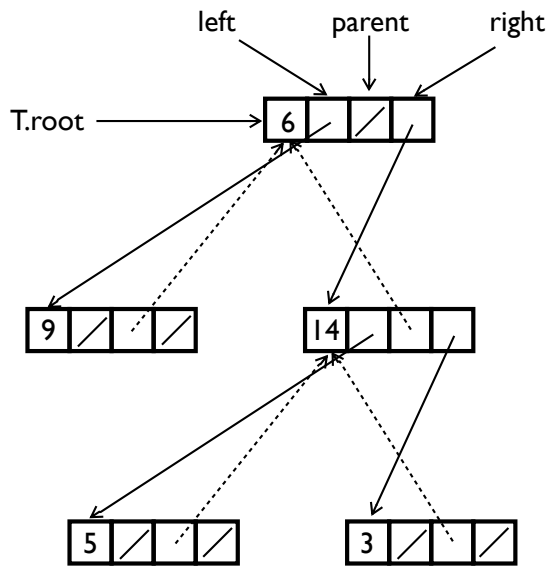
Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Implémentation en C (arbre binaire)

En utilisant une structure liée :

- BTNode : une structure contenant pour chaque nœud `n` :
  - Un champ de données (`data`)
  - Un pointeur vers son nœud parent (`parent`)
  - Un pointeur vers ses fils gauche et droit (`left` et `right`)
- BTree : une structure contenant un pointeur vers la racine (`root`)
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $O(n)$ pour n nœuds





(généralise la notion de liste liée)

Implémentation par structure liée : création et libération

```c
#include <stdio.h>
#include <stdlib.h>
#include "BTree.h"

struct BTNode_t {
  BTNode *parent;
  BTNode *left;
  BTNode *right;
  int data;
};

struct BTree_t {
  BTNode *root;
  int size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
BTree  *btCreate() {
  BTree *tree = malloc(sizeof(BTree));
  if (!tree)
    terminate("Tree can not be created");
  tree->root = NULL;
  tree->size = 0;
  return tree;
}

static void FreeNodesRec(BTNode *n) {
  if (!n)
    return;
  FreeNodesRec(n->left);
  FreeNodesRec(n->right);
  free(n);
}

void btFree(BTree *tree) {
  FreeNodesRec(tree->root);
  free(tree);
}
```

Implémentation par structure liée : création de nœuds

```
static BTNode *createNode(int data) {
  BTNode *n = malloc(sizeof(BTNode));
  if (!n)
    terminate("Node can not be created");
  n->data   = data;
  n->left   = NULL;
  n->right  = NULL;
  n->parent = NULL;
  return n;
}

BTNode *btCreateRoot(BTree *tree,
                     int data) {
  BTNode *root = createNode(data);
  tree->root = root;
  tree->size = 1;
  return root;
}
```

```
BTNode *btInsertLeft(BTree *tree,
                     BTNode *n,
                     int data) {
  BTNode *nleft = createNode(data);
  n->left = nleft;
  nleft->parent = n;
  tree->size++;
  return nleft;
}

BTNode *btInsertRight(BTree *tree,
                      BTNode *n,
                      int data) {
  BTNode *nright = createNode(data);
  n->right = nright;
  nright->parent = n;
  tree->size++;
  return nright;
}
```

Implémentation par structure liée : accesseurs

```
BTNode *btRoot(BTree *tree) {
  return tree->root;
}
BTNode *btLeft(BTree *tree, BTNode *n) {
  return n->left;
}
BTNode *btRight(BTree *tree, BTNode *n) {
  return n->right;
}
BTNode *btParent(BTree *tree, BTNode *n) {
  return n->parent;
}
int btGetData(BTree *tree, BTNode *n) {
  return n->data;
}
int btSize(BTree *tree) {
  return tree->size;
}
```

```
int btIsRoot(BTree *tree, BTNode *n) {
  return (n->parent==NULL);
}
int btIsInternal(BTree *tree, BTNode *n) {
  return (n->left!=NULL || n->right!=NULL);
}
int btIsExternal(BTree *tree, BTNode *n) {
  return (n->left==NULL && n->right==NULL);
}
int btHasLeft(BTree *tree, BTNode *n) {
  return (n->left!=NULL);
}
int btHasRight(BTree *tree, BTNode *n) {
  return (n->right!=NULL);
}
```

Remarques :

- Pour plusieurs fonctions, l'argument `tree` ne sert à rien. Ca pourrait être le cas cependant pour une autre implémentation.
- On pourrait ajouter une gestion d'erreur (pour empêcher l'accès à un nœud inexistant par exemple)

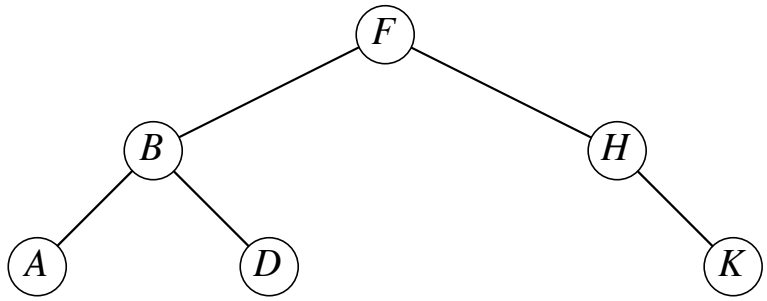
Parcours d'arbres (binaire)

Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir

Différents types de parcours :

- Parcours en profondeur :
  - Infixe (en ordre)
  - Préfixe (en préordre)
  - Suffixe (en postordre)
- Parcours en largeur

Parcours infixe



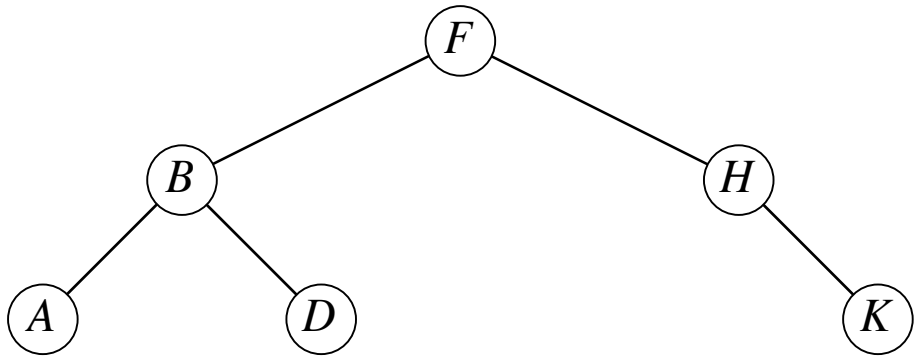
$$\Rightarrow \langle A, B, D, F, H, K \rangle$$

Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit

```
static void btInOrderTreeWalk(BTree *tree, BTNode *n) {
  if (btHasLeft(tree, n))
    btInOrderTreeWalk(tree, btLeft(tree, n));
  printf(" %d", btGetData(tree, n));
  if (btHasRight(tree, n))
    btInOrderTreeWalk(tree, btRight(tree, n));
}
```

Appel : `btInOrderTreeWalk(tree, btRoot(tree))`

Parcours préfixe



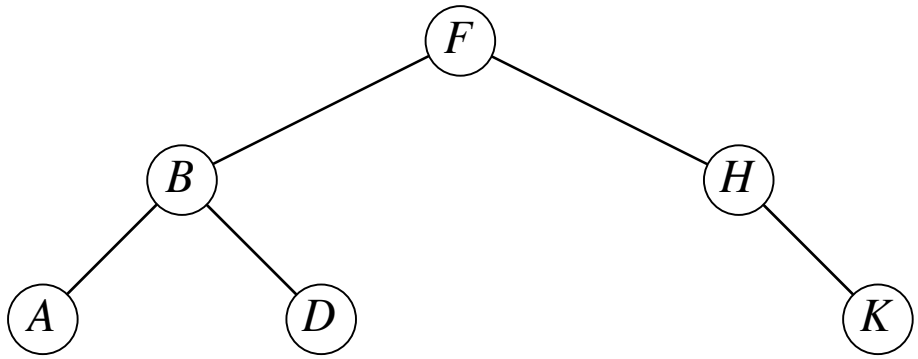
$$\Rightarrow \langle F, B, A, D, H, K \rangle$$

Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils

```
static void btPreOrderTreeWalk(BTree *tree, BTNode *n) {
  printf(" %d", btGetData(tree, n));
  if (btHasLeft(tree, n))
    btPreOrderTreeWalk(tree, btLeft(tree, n));
  if (btHasRight(tree, n))
    btPreOrderTreeWalk(tree, btRight(tree, n));
}
```

Appel : `btPreOrderTreeWalk(tree, btRoot(tree))`

Parcours postfixe



$$\Rightarrow \langle A, D, B, K, H, F \rangle$$

Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

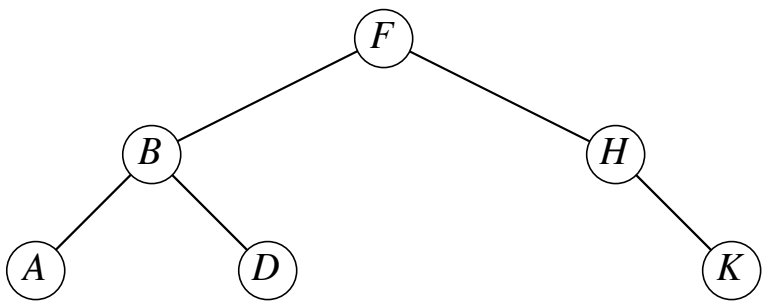
```
static void btPostOrderTreeWalk(BTree *tree, BTNode *n) {
  if (btHasLeft(tree, n))
    btPostOrderTreeWalk(tree, btLeft(tree, n));
  if (btHasRight(tree, n))
    btPostOrderTreeWalk(tree, btRight(tree, n));
  printf(" %d", btGetData(tree, n));
}
```

Appel : `btPostOrderTreeWalk(tree, btRoot(tree))`

Parcours en largeur

Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...

Implémentation à l'aide d'une file :



$\Rightarrow \langle F, B, H, A, D, K \rangle$

```
void btBreadthFirstTreeWalk(BTree *tree) {
  Queue *q = queueCreate();
  if (btSize(tree)>0)
    queueEnqueue(q, btRoot(tree));
  while (!queueIsEmpty(q)) {
    BTNode *n = queueDequeue(q);
    printf(" %d ", btGetData(tree, n));
    if (btHasLeft(tree, n))
      queueEnqueue(q, btLeft(tree, n));
    if (btHasRight(tree, n))
      queueEnqueue(q, btRight(tree, n));
  }
  queueFree(q);
}
```

(Exercice : Implémenter les parcours en profondeur de manière non récursive)

Complexité des parcours

Tous les parcours sont :

- $O(n)$ en temps : dans tous les cas, on ne passe qu'une et une seule fois sur chaque nœud.
- $O(n)$ en mémoire dans le pire cas *(à quoi correspondent-ils ?)*

Plan

1. Introduction

2. Pile et file

3. Arbre

4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Plan

1. Introduction

2. Pile et file

3. Arbre

4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Dictionnaire : principe

Un dictionnaire est une collection de paires (clé, valeur) où

- clé est une valeur permettant d'identifier de manière unique un élément du dictionnaire. Par exemple : un entier, une chaîne de caractères, etc.
- valeur est une valeur qu'on souhaite associer à cet élément.

Interface :

- `Insert(d, key, value)` : insère la paire (`key`,`value`) dans le dictionnaire `d`. Si la clé s'y trouve déjà, sa valeur est mise à jour.
- `Search(d, key)` : cherche la clé `key` dans le dictionnaire `d`. Si la valeur est trouvée, la valeur associée est renvoyée, sinon `NULL`.
- `Remove(d, key)` : supprime la clé `key` (et sa valeur) du dictionnaire.
- $+$ création d'un dictionnaire vide et libération de la mémoire.
- $+$ parcours de toutes les clés

Dictionnaire : principe

Un dictionnaire s'appelle aussi un tableau associatif ou une table de symboles.

Un dictionnaire peut être vu comme un généralisation d'un tableau dont les indices sont remplacés par n'importe quel type de clé.

`Insert(d, "Pierre",4)` $\Leftrightarrow$ `d["coucou"] = 4`

Les clés sont souvent supposées pouvoir être ordonnées mais ce n'est pas toujours nécessaire.

Contraintes de performance implicites :

- Les opérations d'insertion et de recherche doivent être les plus efficaces possibles.
- L'espace mémoire consommé doit être minimal et idéalement s'adapter au nombre de clés.

Applications

Les applications sont très nombreuses :

- Liste de contacts : clé = nom, valeur = numéro de téléphone, adresse
- Dictionnaire : clé = mot, valeur = définition
- Recherche internet : clé = mot clé, valeur = liste de pages web
- domain name service (DNS) : clé = nom de domaine (`www.uliege.be`), valeur = adresse IP (`139.165.X.Y`)
- Compilateur : clé = nom de variable, valeur = valeur et type de la variable.
- Système de fichier : clé = nom de fichier, valeur = localisation du fichier sur le disque
- ...

Ensemble : un dictionnaire simplifié

Un ensemble (set) est une collection de clés uniques.

Équivalent à un dictionnaire, sans valeurs associées aux clés.

Interface :

- `Insert(s, key)` : ajoute la clé `key` dans l'ensemble `s` si elle ne s'y trouve pas déjà.
- `Contains(s, key)` : renvoie 1 si la clé `key` est contenue dans l'ensemble `s`, 0 sinon.
- `Remove(s, key)` : supprime la clé `key` de l'ensemble `s`.
- $+$ création et libération de la structure.
- $+$ parcours des clés de l'ensemble.

Les techniques d'implémentation d'un dictionnaire peuvent s'adapter trivialement à l'ensemble.

Exemple d'interface en c

En supposant des clés de type `int` et des valeurs de type `void *`.

dict.h

```
#ifndef _DICT_H
#define _DICT_H

typedef struct Dict_t Dict;

Dict *dictCreate();
void   dictFree(Dict *);
void   dictInsert(Dict *, int, void *)
void  *dictSearch(Dict *, int);
int    dictContains(Dict *, int);
void  *dictRemove(Dict *, int);

#endif
```

set.h

```
#ifndef _SET_H
#define _SET_H

typedef struct Set_t Set;

Set *setCreate();
void   setFree(Stack *);
void   setInsert(Set *, int);
int    setContains(Set *, int);
void  *setRemove(Set *, int);

#endif
```

Remarque : on ne peut pas remplacer le type de la clé par un type `void *`, sauf à rajouter des pointeurs de fonctions en argument aux fonctions `dicoCreate` et `setCreate`. Par exemple pour passer une fonction de comparaison des clés.

Application 1 : filtrage d'adresses emails

(voir Partie 5, slide 239)

```
...
#include "set.h"

int main() {
  char buffer[10000];

  // construction de l'ensemble
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer) - 1;
    buffer[lenstr]='\0';
    setInsert(s, strdup(newstring));
  }

  // filtrage
  if (setContains(s, "p.geurts@uliege.be")) {
    ...
  }
  setFree(s);
}
```

Utilisation : `./filter < adresses.txt`

Application 2 : suppression des doublons dans un fichier

```
...
#include "set.h"

int main() {
  char buffer[1000];
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';

    if (!setContains(s, buffer)) {
      printf("%s\n", buffer);
      setInsert(s, strdup(newstring));
    }
  }
  setFree(s);
}
```

Utilisation : `./removeduplicates < list.txt > listunique.txt`

Application 3 : compter la fréquence des mots

```
...
#include "dict.h"

int main() {
  char buffer[1000];
  Dict *d = dictCreate(25000);

  while (fgets(buffer, 1000, stdin)) {
    char *string = buffer;
    char *token;
    while ((token = strsep(&string," \t\n,;.:?\"\r*-_()[]'")) != NULL) {
      if (strlen(token)>0) {
        int c = dictSearch(d, token);
        if (c == -1) {
          dictInsert(d, strdup(token), 1);
        } else {
          dictInsert(d, token, c+1);
        }
      }
    }
  }
  dictPrint(d); // affiche le contenu du dictionnaire
  dictFree(d);
}
```

Utilisation : `./countwords < le_rouge_et_le_noir.txt > countings.csv`

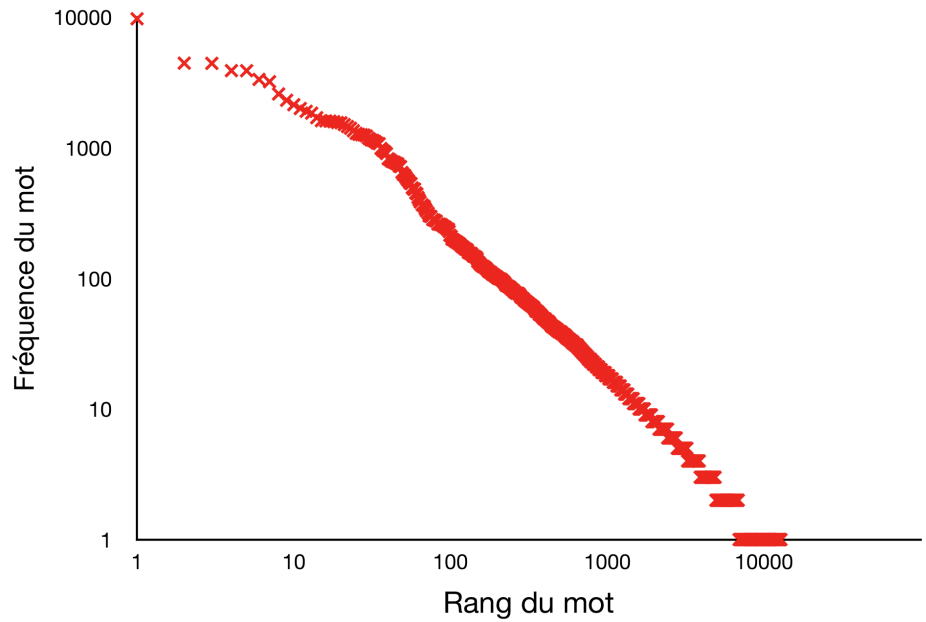
Application 3 : compter la fréquence des mots

Application en linguistique, compression de données, etc.

La loi de Zipf[4] dit que la fréquence du $n$ème mot le plus fréquent dans un texte est $f(n) = \frac{K}{n}$ où K est une constante.

Exemple : "le rouge et le noir" de Stendhal[5] (180000 mots dont 13000 uniques)

Mot	Rang	Freq.
de	1	9738
il	2	4454
la	3	4439
le	4	3902
à	5	3898
l	6	3344
et	7	3212
un	8	2587
que	9	2315
d	10	2140



4. https://fr.wikipedia.org/wiki/Loi_de_Zipf
5. Téléchargé de https://www.gutenberg.org

Plan

1. Introduction

2. Pile et file

3. Arbre

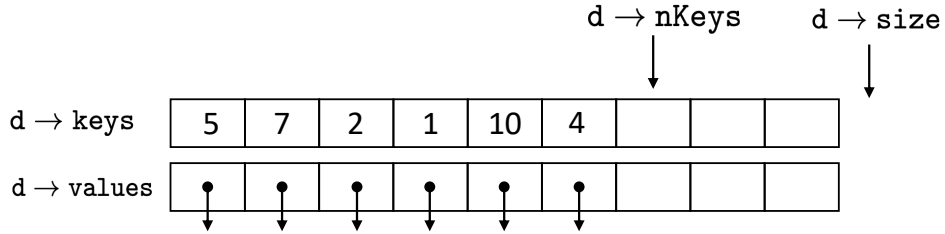
4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Implémentation par tableau

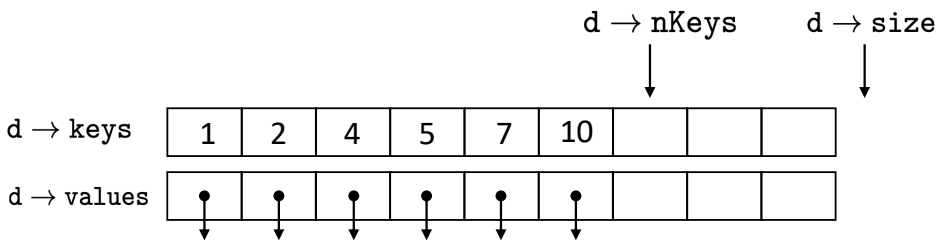
```
struct Dict_t {
  int *keys;           // clés à valeurs entières
  void **values;       // valeurs de type (void *)
  unsigned int size;   // taille du tableau
  unsigned int nKeys; // nombre de clés
};
```

Deux options :

- Tableau de clés non trié : recherche linéaire et insertion à la fin



- Tableau de clés trié : recherche dichotomique et insertion en décalant vers la droite.



(Implémentation laissée comme exercice)

Complexité

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Tableau non trié :
  - Recherche : $O(n)$
    - On parcourt le tableau jusqu'à trouver la clé recherchée ($O(n)$).
  - Insertion : $O(n)$
    - On cherche la clé dans le tableau ($O(n)$) : si présente, on écrase sa valeur ($O(1)$), sinon on la place à la fin du tableau ($O(1)$).
- Tableau trié :
  - Recherche : $O(\log n)$
    - On utilise la recherche dichotomique.
  - Insertion : $O(n)$
    - On insére la clé en fin de tableau et on la fait remonter vers la gauche jusqu'à sa position dans l'ordre (cf. tri par insertion)

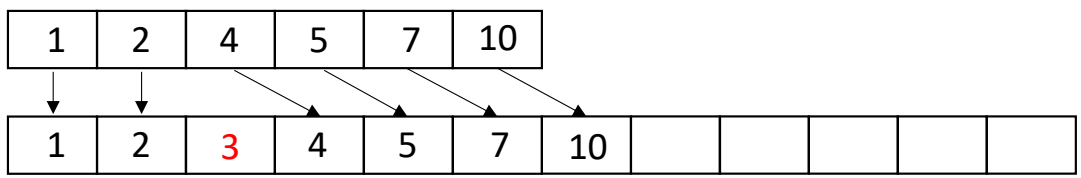
En espace : $O(1)$, car la taille du tableau ne dépend pas du nombre de paires (clé, valeur) stockées.

Tableau extensible

Problème de l'approche précédente : la taille du tableau est fixée et n'évolue pas en fonction des données (complexité en espace $O(1)$).

Solution :

- Initialisez les tableaux `keys` et `values` à une certaine taille.
- Quand les tableaux deviennent trop petits :
  - On alloue des nouveaux tableaux, deux fois plus grands.
  - On recopie les anciennes clés et valeurs dans ces nouveaux tableaux.
  - On libére les anciens tableaux.



Complexités :

- en temps : $O(n)$ (inchangée)
- en espace : $O(n)$ (au pire, on gaspille $O(n)$).

Plan

1. Introduction

2. Pile et file

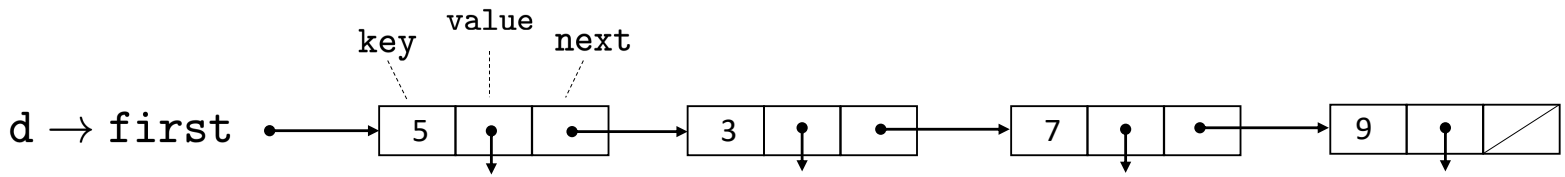
3. Arbre

4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Implémentation par liste liée

```c
typedef struct Node_t {
  int key;                // clés à valeurs entières
  void *value;            // valeurs de type (void *)
  struct Node_t *next;
} Node;

struct Dict_t {
  Node *first;
  unsigned int nKeys;  // nombre de clés (optionnel)
};
```

Recherche en parcourant la liste, insertion en début de liste.



(Implémentation laissée comme exercice)

Complexités

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Recherche : $O(n)$
  - On parcourt la liste depuis le début et on s'arrête quand on trouve la clé recherchée (ou quand la fin de liste est atteinte).
- Insertion : $O(n)$
  - On recherche la clé dans la liste ($O(n)$). Si présente, on écrase sa valeur ($O(1)$), sinon on l'ajoute en début de liste ($O(1)$).

En espace : $O(n)$

- L'espace mémoire nécessaire croît linéairement avec le nombre de paires stockées

Plan

1. Introduction

2. Pile et file

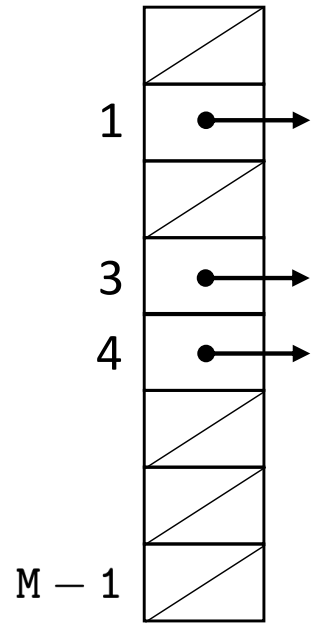
3. Arbre

4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Implémentation par tableau : deuxième version

Si on suppose que les clés prennent des valeurs entre 0 et $M-1$ avec M petit, on peut obtenir une implémentation beaucoup plus efficace.

```
struct Dict_t {
  void *values[M];
}

void dictInsert(Dict d, int key, void *value) {
  d -> values[key] = value;
}

void *dictSearch(Dict d, int key) {
  return values[key];
}
```



Complexités :

- en temps : $O(1)$ pour la recherche et l'insertion.
- en espace : $O(1)$, car l'espace mémoire ne dépend pas du nombre de clés.

Table de hachage : fonction de hachage

Problème de l'approche précédente : les clés doivent être entières et prendre des valeurs pas trop grandes.

Idée 1 : Soit U l'ensemble des valeurs possibles de clés, même non entières. On définit une fonction de hachage h :

$$h : U \to \{0, \ldots, M-1\}$$

envoyant chaque clé $k \in U$ vers une position $h(k)$ dans la table.

Insertion et recherche modifiées (toujours $O(1)$ si h est $O(1)$) :

```
void dictInsert(Dict d, int key, void *value) {
  d -> values[h(key)] = value;
}

void *dictSearch(Dict d, int key) {
  return values[h(key)];
}
```

Table de hachage : gestion des collisions

Problème : Sans connaître les clés, il est difficile de garantir que $h(k_1) \neq h(k_2)$ lorsque $k_1 \neq k_2$. Et c'est impossible dès que $|U| > M \Rightarrow$ le code précédent est incorrect.

Idée 2 : Chaque case i de la table contient une liste liée retenant toutes les clés k insérées telles que $h(k) = i$.

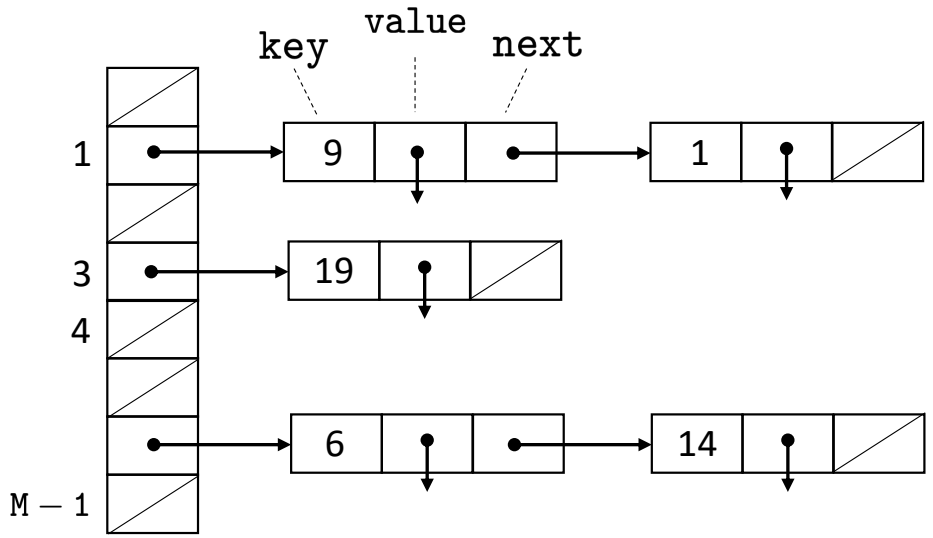


Table de hachage : implémentation (structure et création)

```c
typedef struct Node_t {
  int key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize; // taille du tableau
  unsigned int nKeys;     // nombre de clés (optionnel)
};

Dict *dictCreate(int m) {
  Dict *d = malloc(sizeof(Dict));
  if (d == NULL) exit(-1);
  d -> array = calloc(m, sizeof(Node));
  if (d -> array == NULL) exit(-1);
  d -> arraySize = m;
  d -> nKeys = 0;
  return d;
}
```

Remarque : Contrairement à `malloc`, `calloc` initialise la mémoire à 0 (ou NULL) en plus de faire l'allocation. Nécessaire ici !

Table de hachage : implémentation (recherche)

```
void *dictSearch(Dict d, int key, void *value) {
  Node *p = d -> array[h(d, key)];
  while (p != NULL && p -> key != key)
    p = p -> next;

  if (p != NULL)
    return p -> value;
  else
    return NULL;
}
```

Remarque : La fonction de hachage `h` prend la table en argument (voir plus loin).

Table de hachage : implémentation (insertion)

```
void *dictInsert(Dict d, int key, void *value) {
  Node *p = d -> array[h(d, key)];
  while (p != NULL && p -> key != key)
    p = p -> next;
  if (p != NULL)
    p -> value = value;
  else {
    Node *newNode = malloc(sizeof(Node));
    newNode -> key = key;
    newNode -> value = value;
    newNode -> next = d -> array[h(d, key)];
    d -> array[h(d, key)] = newNode;
    d -> nKeys++;
  }
}
```

Complexité : éléments d'analyse

La complexité dépend de la taille M de la table et de la fonction de hachage (supposée $O(1)$).

Dans le pire cas :

- Toutes les clés sont envoyées dans la même case.
- $\Rightarrow$ Insertion et recherche en $O(n)$.

Dans le meilleur cas :

- Les clés sont réparties uniformément dans toutes les cases de la table.
- $\Rightarrow$ Insertion et recherche en $O(n/M)$.
- $\Rightarrow$ Proche de $O(1)$ si M est $O(n)$.

Complexité en espace : $O(M + n)$ (pour la table et pour les éléments de liste liée).

Fonction de hachage

Le choix de la fonction de hachage détermine l'efficacité des opérations :

- Elle doit être facile à calculer (c'est-à-dire $O(1)$).
- Elle doit répartir les clés aussi uniformément que possible dans la table (très difficile à assurer).

Lorsque les clés sont à valeurs entières, une approche simple est la méthode de division :

$$h(k) = k \mod M.$$

En C :

```
static unsigned int h(Dict *d, int key) {
  return key % d->arraySize;
}
```

Fonction de hachage : chaînes de caractères

Lorsque la clé n'est pas à valeur entière, il faut d'abord passer par une fonction d'encodage.

Exemples pour les chaînes de caractères :

- On additionne les caractères de la chaîne (naïf) :

```
static unsigned int h(Dict *d, char *key) {
  unsigned int hash = 0;
  while (*key != '\0') {
    hash += *key;
    key++;
  }
  return hash % d->arraySize;
}
```

- Une meilleure approche (`djb2`) :

```
static unsigned int h(Dict *d, char *key) {
  unsigned long hash = 5381;
  while (*key != '\0') {
    hash = hash * 33 + *key;
    key++;
  }
  return hash % d->arraySize;
}
```

Implémentation générique

On peut définir une table de hachage plus générique en stockant dans la structure une fonction de comparaison et une fonction d'encodage de clés (passées en arguments à `dictCreate`) :

```
typedef struct Node_t {
  void *key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize;
  unsigned int nKeys;

  int (*compareFunction)(const void *key1, const void *key2);
  unsigned long (*encodeKey)(const void *key);
};

static unsigned int h(Dict *d, void *key) {
  return d -> encodeKey(key) % d->arraySize;
}
```

Plan

1. Introduction

2. Pile et file

3. Arbre

4. Dictionnaire
    - Principe
    - Implémentation par tableau
    - Implémentation par liste liée
    - Implémentation par table de hachage
    - Illustration

Application au filtrage d'adresses email

Quelle est la complexité du code du slide 350 en fonction de l'implémentation de l'ensemble ?

Création et remplissage de l'ensemble :

- $O(n^2)$ dans le pire cas pour toutes les implémentations
- $O(n \log n)$ avec l'implémentation tableau si on trie le tableau seulement quand toutes les adresses ont été lues (possible seulement si toutes les adresses sont connues à l'avance).
- $O(n)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

Recherche d'une adresse :

- $O(n)$ dans le pire cas pour la liste liée et la table de hachage.
- $O(\log n)$ pour le tableau trié.
- $O(1)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

(Complexité de la suppression de doublons et du comptage de mots ?)

Application au filtrage d'adresses email

Tests empiriques :

- Génération de données : n chaînes de caractères (`a-z`) aléatoires de longueur 10.
- Requêtes : $10n$ chaînes prises au hasard dans la liste (recherches positives) ou en dehors (recherches négatives).

Tableau trié (par fusion) et recherche dicho. versus table de hachage ($M = 2n$, `djb2`) :

Création de la structure :

n	Tri (s)	Table (s)
12500	0,003	0,002
25000	0,005	0,002
50000	0,012	0,007
100000	0,026	0,013
200000	0,057	0,036
400000	0,128	0,085

Recherches positives

n	Rech. dico. (s)	Table (s)
500000	3,09	1,31
1000000	7,95	2,89

Recherches négatives

n	Rech. dico. (s)	Table (s)
500000	3,76	1,23
1000000	9,21	2,72

La table de hachage permet de traiter trois fois plus d'adresses à la seconde.

Conclusion

L'implémentation par table de hachage est en pratique plus efficace que les implémentations par liste liée et par tableau, malgré une complexité dans le pire cas identique, voire moins bonne.

Les performances de la recherche dans un tableau trié sont plus stables mais l'insertion est beaucoup plus couteuse, à cause du décalage.

D'autres structures existent qui permettent de combiner la facilité d'insertion de la liste liée avec la rapidité et la stabilité de la recherche dichotomique. Par exemple les arbres binaires de recherche (INFO0902).

Il existe aussi des structures spécifiques pour les chaînes de caractères. Par exemple les tries.