Partie 6

Structures de données

18 novembre 2018

Plan

Plan

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.

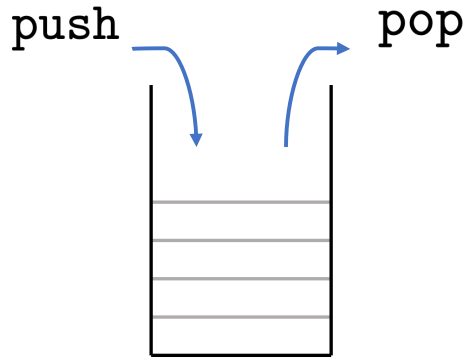
Principalement des ensemble de données dynamiques.

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.
- 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

Pile



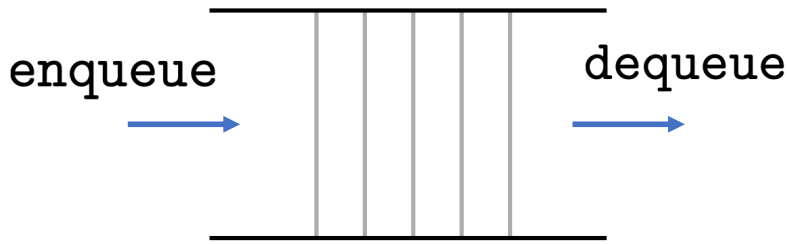
Une pile est une collection de valeurs, accessibles selon une discipline LIFO (Last In First Out)

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'email.

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

    char *newstring = malloc((lenstr+1)*sizeof(char));
    if (!newstring) exit(-1);
    strcpy(newstring, buffer);

    queueEnqueue(q, newstring);
  }
  // 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 *strcpy(char *dest, const chart *src)` : copie la chaîne `src` dans la chaîne `dst` (caractère NUL compris). Renvoie `dest`.

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 $-$

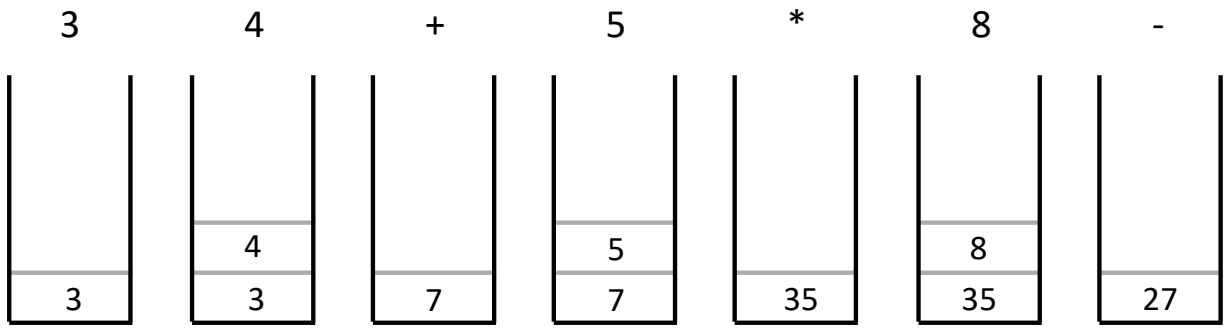
Aucune parenthèse n'est nécessaire dans cette notation : il n'y a qu'une seule manière de mettre des parenthèses.

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

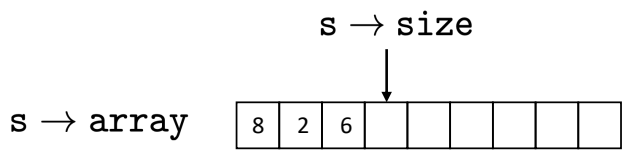
Implémentation par tableau d'une pile : principe

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)
    return NULL;

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

```
void *stackPop(Stack *s) {
  if (s -> size == 0)
    return NULL;

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


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

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

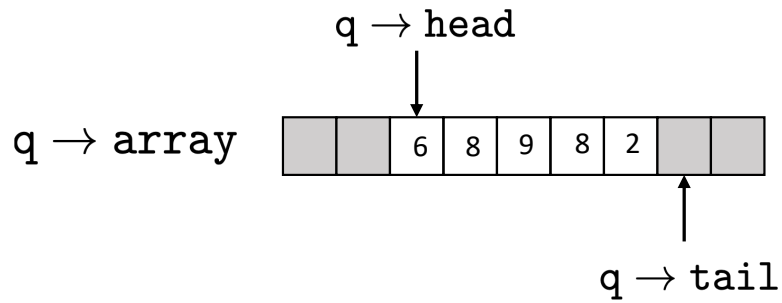
Implémentation par tableau d'une file : principe

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.

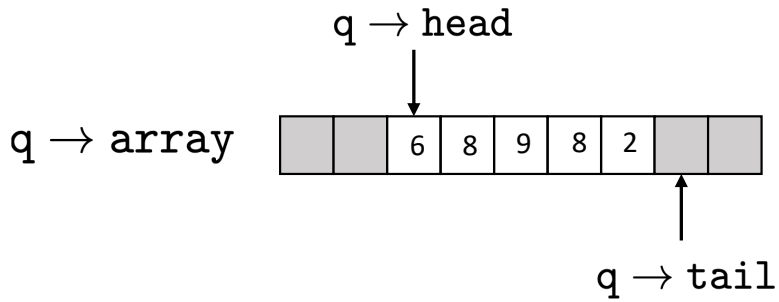
La taille du tableau doit être fixée a priori et une erreur signalée lorsque la taille de la file excède la taille du tableau.

```
const int  MAX_STACK_SIZE = 1000;

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

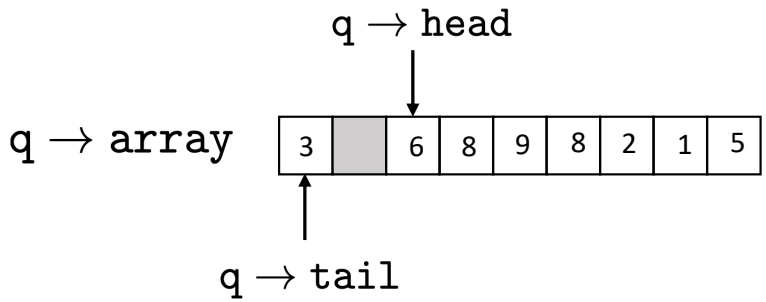


Illustration

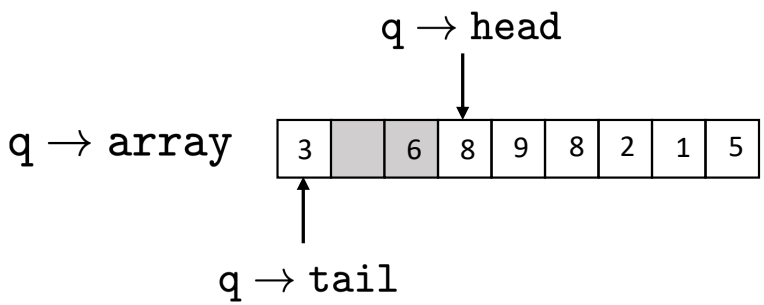
File initiale :



`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 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

Liste (simplement) liée

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

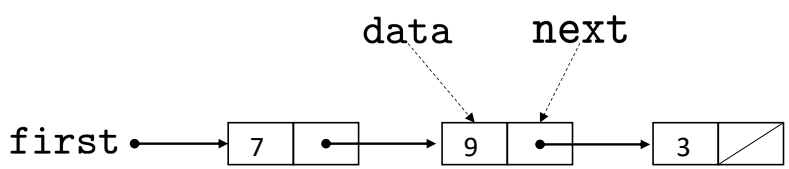
Chaque élément de liste (aussi appelé un nœud) est composé :

- 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 :

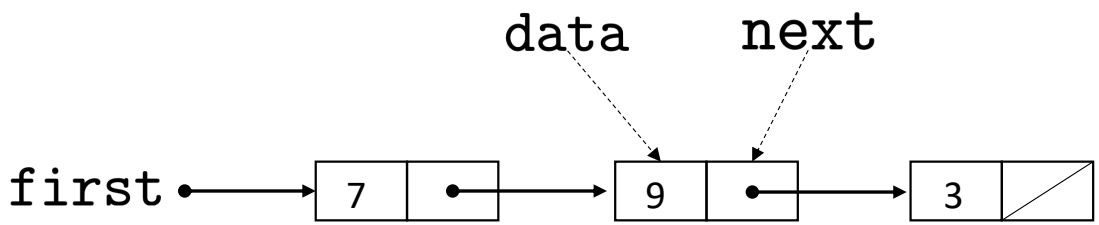
```
typedef struct Node_t {
  int data;
  struct Node_t next;
} Node;
```



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;
```

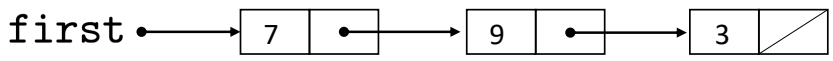


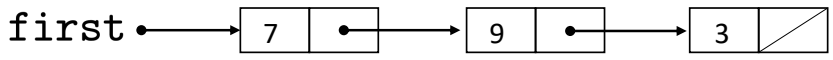
Manipulation d'une liste liée

Extraction du premier élément

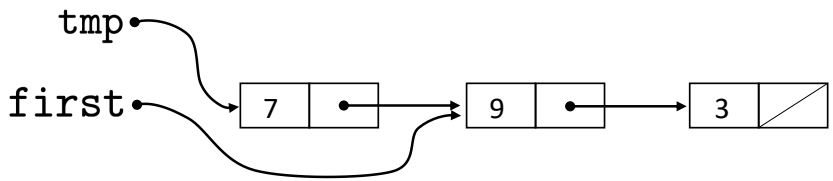
```
int value = first -> data;

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

free(tmp);
return value;
```

value
7

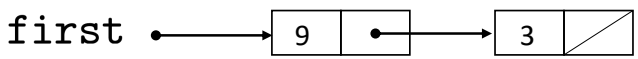




value
7



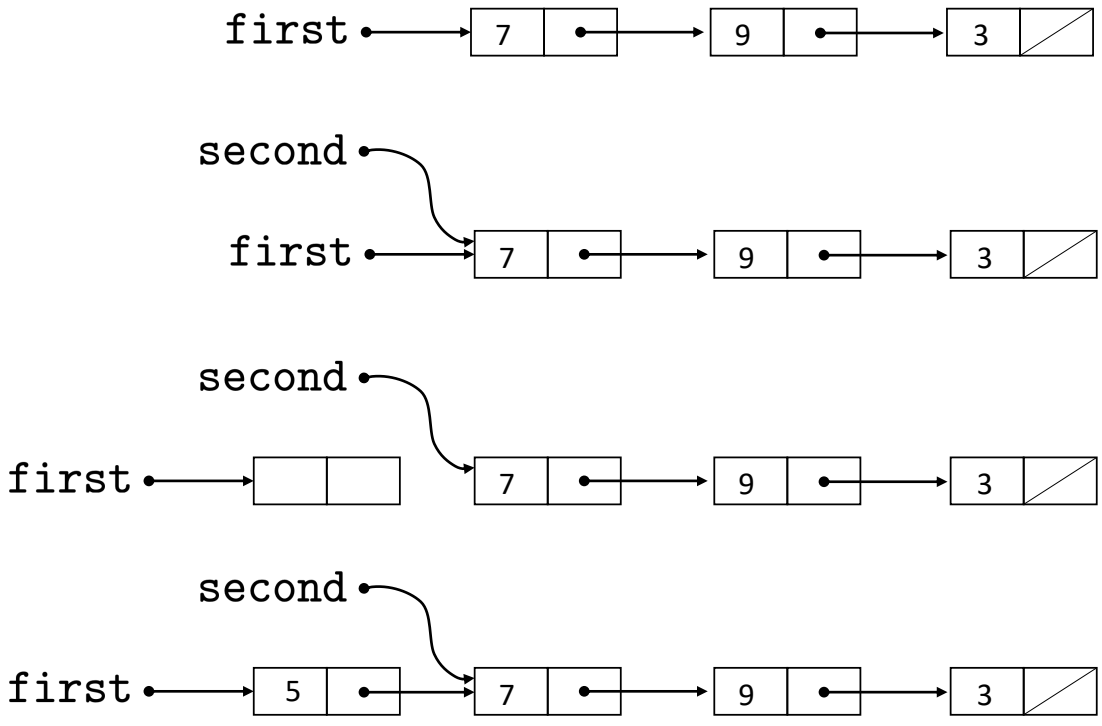
value
7



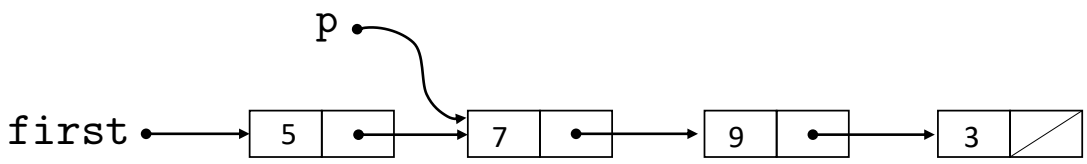
Manipulation d'une liste liée

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.

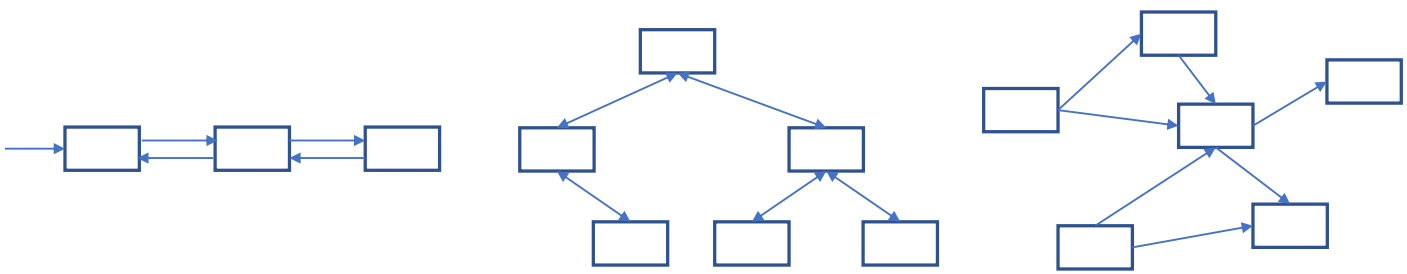
Généralisation : structures liées

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

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.
- Graphe : pointeur vers chaque nœud adjacent.



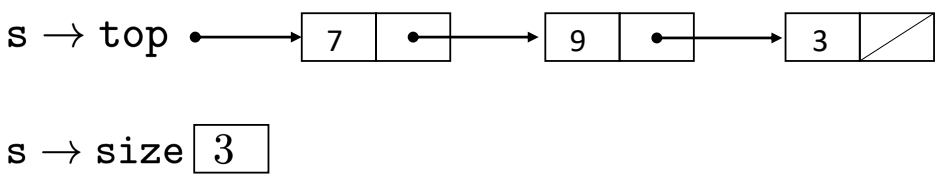
plan

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))
    return NULL;

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

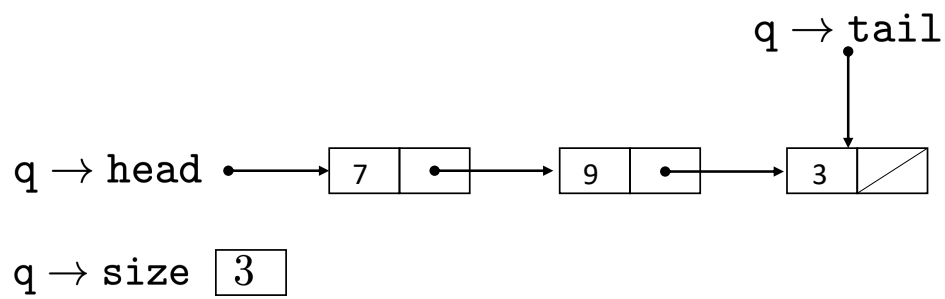
```
void *stackPop(Stack *s) {
  if (!(s -> top))
    return NULL;

  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 -> tail))
    return NULL;
  return q -> head -> data;
}
```

```
void *queueDequeue(Queue *q) {
  if (!(q -> head))
    return NULL;

  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.