

# Compléments de Programmation

Benoit Donnet  
Année Académique 2023 - 2024



## Agenda

- Chapitre 1: Raisonnement Mathématique
- Chapitre 2: Construction de Programme
- Chapitre 3: Introduction à la Complexité
- Chapitre 4: Récursivité
- Chapitre 5: Types Abstraits de Données
- Chapitre 6: Listes
- Chapitre 7: Piles
- **Chapitre 8: Files**
- Chapitre 9: Elimination de la Récursivité

# Agenda

- Chapitre 8: Files
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

# Agenda

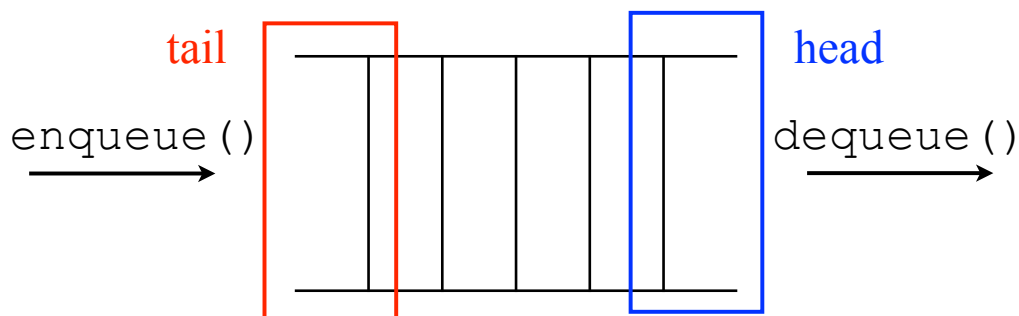
- Chapitre 8: Files
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

# Principe

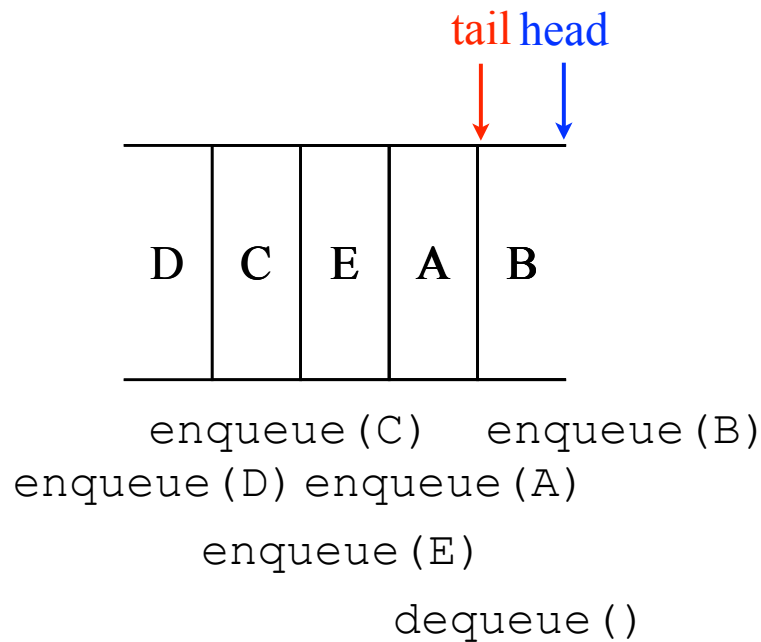
- TAD très utilisé en programmation
  - queue
- Notion intuitive
  - file d'attente à un guichet
  - file de documents à imprimer
  - ...
- FIFO
  - *First-In, First-Out*

## Principe (2)

- Deux opérations de base
  - insertion d'un élément
    - ✓ `enqueue()`
  - suppression d'un élément
    - ✓ `dequeue()`
- Les opérations sont limitées aux 2 extrémités
  - insertion en fin de file (tail)
  - suppression en début de file (head)



# Principe (3)



# Agenda

- Chapitre 8: Files
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

# Spécification Abstraite

- Syntaxe

**Type:**

Queue

**Utilise:**

Boolean, Element, Natural

**Opérations:**

empty\_queue:  $\rightarrow$  Queue

is\_empty: Queue  $\rightarrow$  Boolean

size : Queue  $\rightarrow$  Natural

enqueue: Queue  $\times$  Element  $\rightarrow$  Queue

dequeue: Queue  $\rightarrow$  Queue

head: Queue  $\rightarrow$  Element

# Spécification Abstraite (2)

- Sémantique
  - préconditions

**Préconditions:**

$\forall q \in \text{Queue}$

$\forall q, \neg \text{is\_empty}(q), \text{dequeue}(q)$

$\forall q, \neg \text{is\_empty}(q), \text{head}(q)$

# Spécification Abstraite (3)

		Opérations Internes		
		empty_queue	dequeue (·)	enqueue (·)
Observateurs	is_empty(·)			
	size(·)			
	head(·)	∅		

- Sémantique
  - axiomes
- Combien d'axiomes?
  - 3 Observateurs × 3 Opérations Internes = 9 Axiomes
- Peut-on réduire?
  - préconditions
  - FIFO

# Spécification Abstraite (4)

## Axiomes:

$\forall q \in \text{Queue}, \forall e \in \text{Element}$   
 $\text{is\_empty}(\text{empty\_queue}) = \text{True}$   
 $\text{is\_empty}(\text{enqueue}(q, e)) = \text{False}$   
 $\text{is\_empty}(\text{dequeue}(q)) = \begin{cases} \text{True} & \text{if } \text{size}(q) = 1 \\ \text{False} & \text{otherwise} \end{cases}$

		Opérations Internes		
		empty_queue	dequeue (·)	enqueue (·)
Observateurs	is_empty(·)	✓	✓	✓
	size(·)			
	head(·)	∅		

# Spécification Abstraite (5)

## Axiomes:

$\forall q \in \text{Queue}, \forall e \in \text{Element}$

$\text{size}(\text{empty\_queue}) = 0$

$\text{size}(\text{enqueue}(q, e)) = \text{size}(q) + 1$

$\text{size}(\text{dequeue}(q)) = \text{size}(q) - 1$

		Opérations Internes		
		<code>empty_queue</code>	<code>dequeue(·)</code>	<code>enqueue(·)</code>
Observateurs	<code>is_empty(·)</code>	✓	✓	✓
	<code>size(·)</code>	✓	✓	✓
	<code>head(·)</code>	∅		

# Spécification Abstraite (6)

## Axiomes:

$\forall q \in \text{Queue}, \forall e \in \text{Element}$

$\text{head}(\text{enqueue}(q, e)) = \begin{cases} e & \text{if } \text{is\_empty}(q) \\ \text{head}(q) & \text{otherwise} \end{cases}$

$\text{head}(\text{dequeue}(q)) = \text{head}(q)$

		Opérations Internes		
		<code>empty_queue</code>	<code>dequeue(·)</code>	<code>enqueue(·)</code>
Observateurs	<code>is_empty(·)</code>	✓	✓	✓
	<code>size(·)</code>	✓	✓	✓
	<code>head(·)</code>	∅	✓	✓

# Spécification Abstraite (7)

## Axiomes:

$\forall q \in \text{Queue}, \forall e \in \text{Element}$   
 $\text{dequeue}(\text{enqueue}(q, e)) = \begin{cases} \text{empty\_queue} & \text{if } \text{is\_empty}(q) \\ \text{enqueue}(\text{dequeue}(q), e) & \text{otherwise} \end{cases}$

		Opérations Internes		
		empty_queue	dequeue (·)	enqueue (·)
Observateurs	is_empty (·)	✓	✓	✓
	size (·)	✓	✓	✓
	head (·)	∅	✓	✓

# Agenda

- Chapitre 8: Files
  - Principe
  - Spécification Abstraite
  - Implémentation
    - ✓ Interface
    - ✓ Implémentation Statique par Tableau Contigu
    - ✓ Implémentation Statique par Tableau Circulaire
    - ✓ Implémentation Dynamique
    - ✓ Complexité
  - Utilisation



# Interface

- Fichier `queue.h`

```
#ifndef __QUEUE__
#define __QUEUE__

#include "boolean.h"

typedef struct queue_t Queue;

Queue *empty_queue();

Boolean is_empty(Queue *q);

unsigned int size(Queue *q);

Queue *enqueue(Queue *q, void *e);

Queue *dequeue(Queue *q);

void *head(Queue *q);

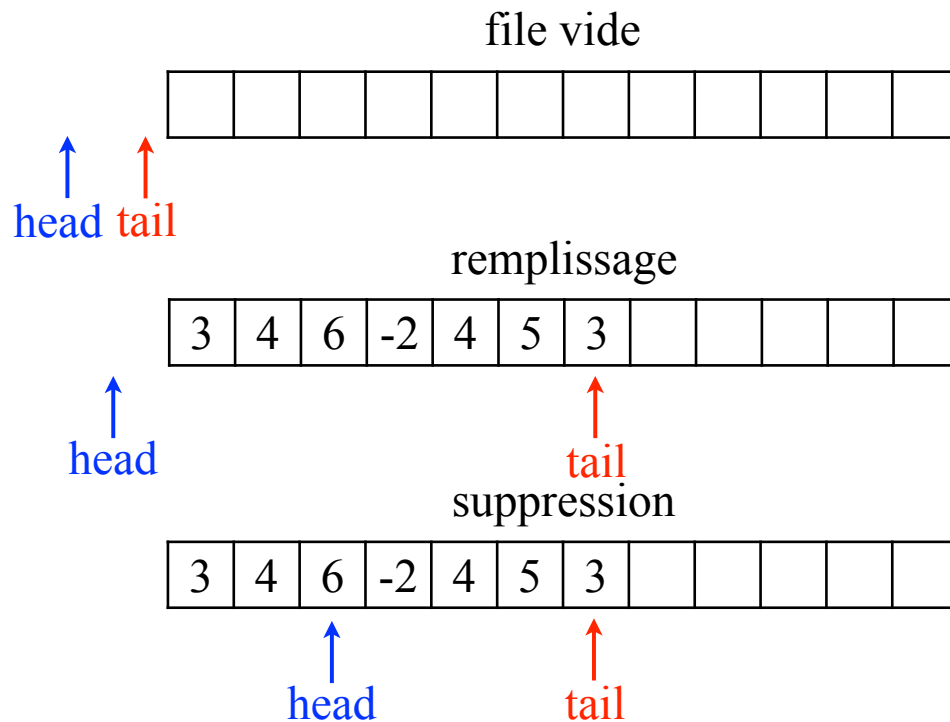
#endif
```

# Tableau Contigu

- Représentation simple par des éléments contigus dans un tableau
  - *head*
    - ✓ position précédant le 1<sup>er</sup> élément
  - *tail*
    - ✓ position du dernier élément

# Tableau Contigu

- Illustration



## Tableau Contigu (2)

- Fichier `static_contiguous_queue.c`

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

#include "queue.h"

#define MAX_SIZE 10

struct queue_t{
    void *queue[MAX_SIZE];
    int head;
    int tail;
};
```

# Tableau Contigu (3)

```
Queue *empty_queue(){
    Queue *q = malloc(sizeof(Queue));
    if(q==NULL)
        return NULL;

    q->head = -1;
    q->tail = -1;

    return q;
} //end empty_queue()

Boolean is_empty(Queue *q){
    return q->head == q->tail;
} //end is_empty()

unsigned int size(Queue *q){
    return q->tail - q->head;
} //end size()
```

# Tableau Contigu (4)

```
void *head(Queue *q){
    assert(!is_empty(q));

    return q->queue[q->head+1];
} //end head()

Queue *enqueue(Queue *q, void *e){
    if(q->tail >= MAX_SIZE-1){
        printf("Error: queue full\n");
        exit(-1);
    }

    q->tail++;
    q->queue[q->tail] = e;

    return q;
} //end enqueue()
```

# Tableau Contigu (5)

```
Queue *dequeue(Queue *q){
    assert(!is_empty(q));

    free(q->queue[q->head+1]);
    q->head++;

    return q;
} //end dequeue()
```

# Tableau Contigu (6)

- Utilisation

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

#include "queue.h"
#include "element.h"

int main(){
    Queue *q = empty_queue();

    if(is_empty(q))
        printf("Queue empty!\n");
    else
        printf("Queue not empty!\n");

    q = enqueue(q, create_element(5));
    q = enqueue(q, create_element(2));

    //à suivre
} //end main()
```

# Tableau Contigu (7)

```
int main(){
    //cfr. slide précédent
    q = enqueue(q, create_element(10));
    q = enqueue(q, create_element(20));
    q = enqueue(q, create_element(30));
    q = enqueue(q, create_element(40));
    q = enqueue(q, create_element(50));

    printf("Queue size: %u\n", size(q));

    Element *elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);

    q = dequeue(q);
    q = dequeue(q);

    elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);
    printf("Queue size: %u\n", size(q));
    return 1;
} //end main()
```

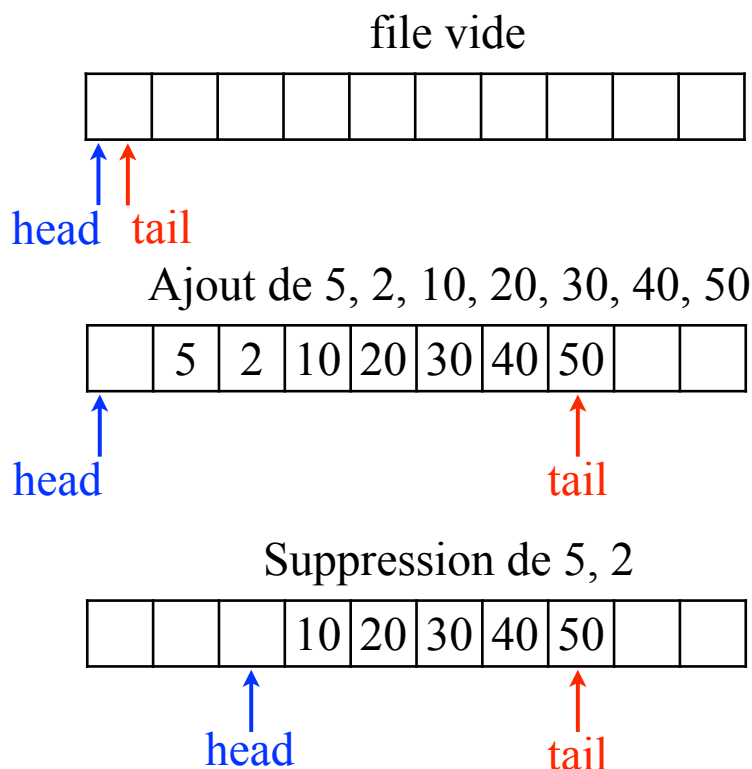
# Tableau Contigu (8)

- Problèmes avec cette implémentation
  - les indices de tête et fin ne font qu'augmenter
    - ✓ qu'on ajoute ou retire des éléments
  - l'utilisation de la file est limitée dans le temps
    - ✓ quand la taille maximale est atteinte, la file n'est plus utilisable

# Tableau Circulaire

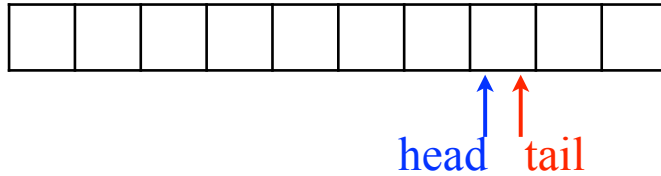
- Gestion circulaire par tableau
  - gérer le tableau de manière circulaire
  - l'élément suivant la position  $i$  est l'élément à la position  $(i+1) \% \text{MAX\_SIZE}$
- La file est autorisée à contenir  $\text{MAX\_SIZE}$  éléments
- Initialisation
  - `head = 0;`
  - `tail = 0;`
- Toutes les places du tableau sont exploitées

## Tableau Circulaire (2)

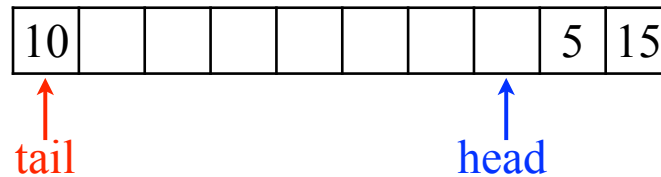


# Tableau Circulaire (3)

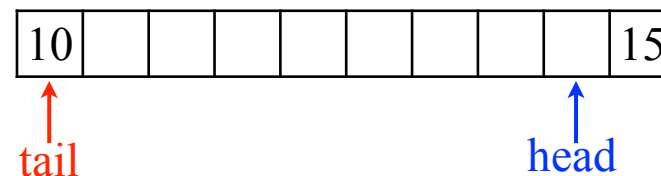
Suppression de 10, 20, 30, 40, 50



Ajout de 5, 15, 10

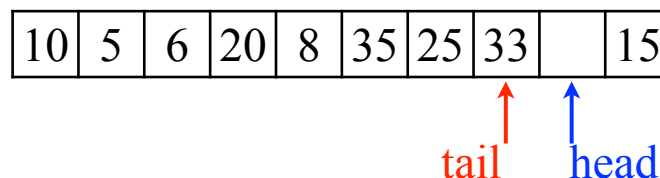


Suppression de 5



# Tableau Circulaire (4)

Ajout de 5, 6, 20, 8, 35, 25, 33



# Tableau Circulaire (5)

- Fichier `static_circular_queue.c`

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

#include "queue.h"

#define MAX_SIZE 10

struct queue_t{
    void *queue[MAX_SIZE];
    int head;
    int tail;
};
```

# Tableau Circulaire (6)

```
Queue *empty_queue(){
    Queue *q = malloc(sizeof(Queue));
    if(q==NULL)
        return NULL;

    q->head = 0;
    q->tail = 0;

    return q;
} //end empty_queue()

Boolean is_empty(Queue *q){
    return q->head == q->tail;
} //end is_empty()

unsigned int size(Queue *q){
    return q->tail - q->head;
} //end size()
```



# Tableau Circulaire (7)

```
void *head(Queue *q){
    assert(!is_empty(q));

    return q->queue[(q->head+1) % MAX_SIZE];
} //end head()

Queue *enqueue(Queue *q, void *e){
    if(q->tail >= MAX_SIZE-1){
        printf("Error: queue full\n");

        exit(-1);
    }

    q->tail = (q->tail+1) % MAX_SIZE;
    q->queue[q->tail] = e;

    return q;
} //end enqueue()
```

# Tableau Circulaire (8)

```
Queue *dequeue(Queue *q){
    assert(!is_empty(q));

    free(q->queue[(q->head+1) % MAX_SIZE]);
    q->head = (q->head+1) % MAX_SIZE;

    return q;
} //end dequeue()
```

# Tableau Circulaire (9)

- Utilisation

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

#include "queue.h"
#include "element.h"

int main(){
    Queue *q = empty_queue();

    if(is_empty(q))
        printf("Queue empty!\n");
    else
        printf("Queue not empty!\n");

    q = enqueue(q, create_element(5));
    q = enqueue(q, create_element(2));

    //à suivre
} //end main()
```

# Tableau Circulaire (10)

```
int main(){
    //cfr. slide précédent
    q = enqueue(q, create_element(10));
    q = enqueue(q, create_element(20));
    q = enqueue(q, create_element(30));
    q = enqueue(q, create_element(40));
    q = enqueue(q, create_element(50));

    printf("Queue size: %u\n", size(q));

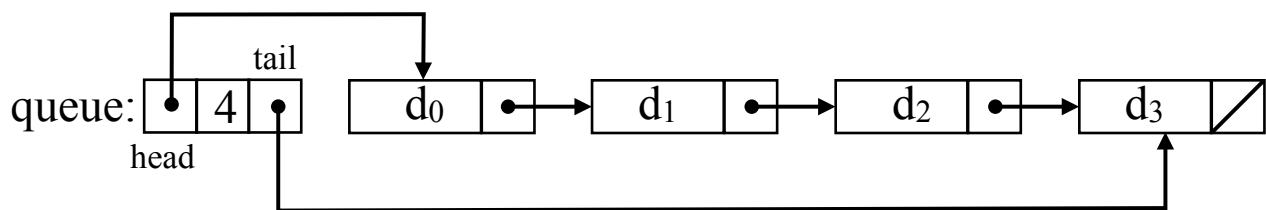
    Element *elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);

    q = dequeue(q);
    q = dequeue(q);

    elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);
    printf("Queue size: %u\n", size(q));
    return 1;
} //end main()
```

# Implem. Dynamique

- Inconvénient de l'implémentation statique
  - taille limitée de la file
- Quid si on veut une file illimitée?
  - implémentation via une liste chaînée avec pointeur début/fin
    - ✓ permet un ajout facile
    - ✓ permet un retrait facile



## Implem. Dynamique (2)

- Fichier `dynamic_queue.c`

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

#include "queue.h"
```

```
typedef struct cell{
    void *data;
    struct cell *next;
}cell;
```

```
struct queue_t{
    cell *head;
    cell *tail;
    unsigned int size;
};
```

Liste chaînée "générique"

Liste chaînée avec ptr début/fin

# Implem. Dynamique (3)

```
static cell *create_cell(void *data){
    cell *n_cell = malloc(sizeof(cell));
    if(n_cell == NULL)
        return n_cell;

    n_cell->next = NULL;
    n_cell->data = data;
    return n_cell;
} //end create_cell()
```

```
Queue *empty_queue(){
    Queue *q = malloc(sizeof(Queue));
    if(q==NULL)
        return NULL;

    q->head = NULL;
    q->tail = NULL;
    q->size = 0;

    return q;
} //end empty_queue()
```

# Implem. Dynamique (4)

```
Boolean is_empty(Queue *q){
    return q->head == NULL;
} //end is_empty()

unsigned int size(Queue *q){
    return q->size;
} //end size()

void *head(Queue *q){
    assert(!is_empty(q));

    return q->head->data;
} //end head()
```

# Implem. Dynamique (5)

```
Queue *enqueue(Queue *q, void *e){
    cell *n_cell = create_cell(e);
    if(n_cell == NULL)
        return q;

    if(is_empty(q)){
        q->tail = n_cell;
        q->head = n_cell;
    }else{
        q->tail->next = n_cell;
        q->tail = n_cell;
    }
    q->size++;

    return q;
} //end enqueue()
```

cas particulier: ajout quand file vide

cas général

# Implem. Dynamique (6)

```
Queue *dequeue(Queue *q){
    assert(!is_empty(q));

    cell *tmp = q->head;
    q->head = q->head->next;

    tmp->next = NULL;
    free(tmp);

    q->size--;

    return q;
} //end dequeue()
```

# Implem. Dynamique (7)

- Utilisation

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

#include "queue.h"
#include "element.h"

int main(){
    Queue *q = empty_queue();

    if(is_empty(q))
        printf("Queue empty!\n");
    else
        printf("Queue not empty!\n");

    q = enqueue(q, create_element(5));
    q = enqueue(q, create_element(2));

    //à suivre
} //end main()
```

# Implem. Dynamique (8)

```
int main(){
    //cfr. slide précédent
    q = enqueue(q, create_element(10));
    q = enqueue(q, create_element(20));
    q = enqueue(q, create_element(30));
    q = enqueue(q, create_element(40));
    q = enqueue(q, create_element(50));

    printf("Queue size: %u\n", size(q));

    Element *elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);

    q = dequeue(q);
    q = dequeue(q);

    elem = (Element *)head(q);
    printf("The head is: %d\n", elem->e);
    printf("Queue size: %u\n", size(q));
    return 1;
} //end main()
```

# Complexité

- Les opérations sur les files sont toutes en  $O(1)$
- Valable pour les différentes implémentations

# Agenda

- Chapitre 8: Files
  - Principe
  - Spécification Abstraite
  - Implémentation
  - Utilisation

# Utilisation

- Gestion des travaux d'impression d'une imprimante
  - imprimante en réseau
  - les tâches d'impression arrivent aléatoirement depuis n'importe quelle machine
  - les tâches sont placées dans la file
    - ✓ traitement dans l'ordre d'arrivée
- Ordonnanceur dans les OS
  - maintenir une file de processus en attente d'un temps machine
- Parcours en largeur d'un arbre
  - ✓ cfr. INFO0902