

File d'attente, liste triée, liste doublement chaînée

Algorithmes et structures de données, 2025-2026

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA
2026-01-06

En partie inspiré des supports de cours de P. Albuquerque

La file d'attente

La file d'attente (1/2)

- Structure de données abstraite permettant le stockage d'éléments.
- *FIFO* : First In First Out, ou première entrée première sortie.
- Analogie de la vie "réelle" " " :
 - File à un guichet,
 - Serveur d'impressions,
 - Mémoire tampon, ...

Fonctionnalités

La file d'attente (1/2)

- Structure de données abstraite permettant le stockage d'éléments.
- *FIFO* : First In First Out, ou première entrée première sortie.
- Analogie de la vie "réelle" " " :
 - File à un guichet,
 - Serveur d'impressions,
 - Mémoire tampon, ...

Fonctionnalités

- Enfiler : ajouter un élément à la fin de la file.
- Défiler : extraire un élément au devant de la file.
- Tester si la file est vide.

La file d'attente (1/2)

- Structure de données abstraite permettant le stockage d'éléments.
- *FIFO* : First In First Out, ou première entrée première sortie.
- Analogue de la vie “réelle” “ :
 - File à un guichet,
 - Serveur d'impressions,
 - Mémoire tampon, ...

Fonctionnalités

- Enfiler : ajouter un élément à la fin de la file.
- Défiler : extraire un élément au devant de la file.
- Tester si la file est vide.
- Lire l'élément de la fin de la file.
- Lire l'élément du devant de la file.
- Créer une file vide.
- Détruire une file.

La file d'attente (2/2)

Implémentation possible

- La structure de file, contient un pointeur vers la tête et un autre vers le début de la file.
- Entre les deux, les éléments sont stockés dans une liste chaînée.

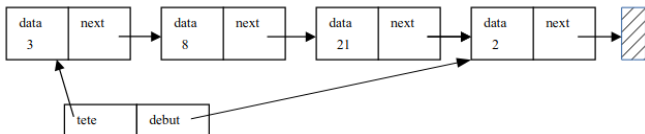


Figure 1 : Illustration d'une file d'attente.

Structure de données en C ?

La file d'attente (2/2)

Implémentation possible

- La structure de file, contient un pointeur vers la tête et un autre vers le début de la file.
- Entre les deux, les éléments sont stockés dans une liste chaînée.

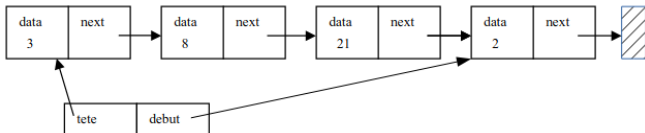


Figure 1 : Illustration d'une file d'attente.

Structure de données en C ?

```
typedef struct _element { // élément de liste
    int data;
    struct _element* next;
} element;

typedef struct _queue { // file d'attente
    element* head; // tête de file d'attente
    element* tail; // queue de file d'attente
```

Fonctionnalités d'une file d'attente

Création et consultations

Fonctionnalités d'une file d'attente

Création et consultations

```
void queue_init(queue *fa); // fa->head = fa->tail = NULL
bool queue_is_empty(queue fa); // fa.head == fa.tail == NULL
int queue_tail(queue fa); // return fa.tail->data
int queue_head(queue fa); // return fa.head->data
```

Manipulations et destruction

Fonctionnalités d'une file d'attente

Création et consultations

```
void queue_init(queue *fa); // fa->head = fa->tail = NULL
bool queue_is_empty(queue fa); // fa.head == fa.tail == NULL
int queue_tail(queue fa); // return fa.tail->data
int queue_head(queue fa); // return fa.head->data
```

Manipulations et destruction

```
void queue_enqueue(queue *fa, int val);
// adds an element before the tail
int queue_dequeue(queue *fa);
// removes the head and returns stored value
void queue_destroy(queue *fa);
// dequeues everything into oblivion
```

Deux cas différents

1. La file est vide (faire un dessin) :

Deux cas différents

1. La file est vide (faire un dessin) :

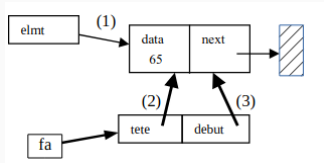


Figure 2 : Insertion dans une file d'attente vide.

2. La file n'est pas vide (faire un dessin) :

Enfilage

Deux cas différents

1. La file est vide (faire un dessin) :

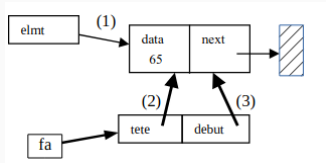


Figure 2 : Insertion dans une file d'attente vide.

2. La file n'est pas vide (faire un dessin) :

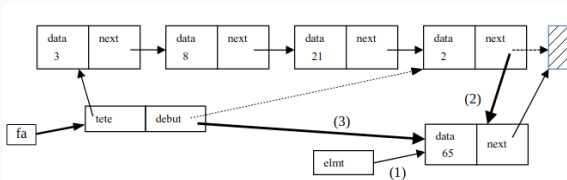


Figure 3 : Insertion dans une file d'attente non vide.

Live (implémentation)

Live (implémentation)

```
void queue_enqueue(queue *fa, int val) {
    element* elmt = malloc(sizeof(*elmt));
    elmt->data = val;
    elmt->next = NULL;
    if (queue_is_empty(*fa)) {
        fa->head = elmt;
    } else {
        fa->tail->next = elmt;
    }
    fa->tail = elmt;
}
```

Trois cas différents

1. La file a plus d'un élément (faire un dessin) :

Trois cas différents

1. La file a plus d'un élément (faire un dessin) :

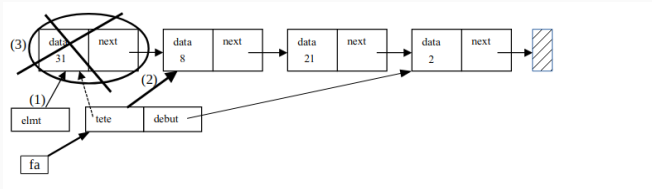


Figure 4 : Extraction d'une file d'attente

2. La file un seul élément (faire un dessin) :

Trois cas différents

1. La file a plus d'un élément (faire un dessin) :

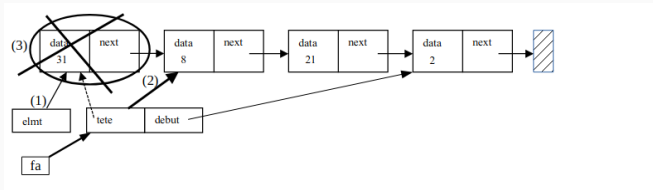
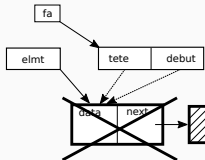


Figure 4 : Extraction d'une file d'attente

2. La file un seul élément (faire un dessin) :



Live (implémentation)

Live (implémentation)

```
int queue_dequeue(queue *fa) {  
    element* elmt = fa->head;  
    int val = elmt->data;  
    fa->head = fa->head->next;  
    free(elmt);  
    if (NULL == fa->head) {  
        fa->tail = NULL;  
    }  
    return val;  
}
```

Live (implémentation)

```
int queue_dequeue(queue *fa) {  
    element* elmt = fa->head;  
    int val = elmt->data;  
    fa->head = fa->head->next;  
    free(elmt);  
    if (NULL == fa->head) {  
        fa->tail = NULL;  
    }  
    return val;  
}
```

Problème avec cette implémentation ?

Comment on fait la désallocation ?

Comment on fait la désallocation ?

On défile jusqu'à ce que la file soit vide !

Quelles sont les complexités de

- Enfiler ?

Quelles sont les complexités de

- Enfiler ?
- Défiler ?

Quelles sont les complexités de

- Enfiler ?
- Défiler ?
- Détruire ?

Quelles sont les complexités de

- Enfiler ?
- Défiler ?
- Détruire ?
- Est vide ?

Comment implémenter la file autrement ?

Comment implémenter la file autrement ?

- Données stockées dans un tableau ;
- Tableau de taille connue à la compilation ou pas (réallouable) ;
- `tail` serait un indice du tableau ;
- `capacity` serait la capacité maximale ;
- On *enfile* “au bout” du tableau, au défile au début (indice 0).

Implémentation alternative

Comment implémenter la file autrement ?

- Données stockées dans un tableau ;
- Tableau de taille connue à la compilation ou pas (réallouable) ;
- `tail` serait un indice du tableau ;
- `capacity` serait la capacité maximale ;
- On *enfile* “au bout” du tableau, au *défile* au début (indice 0).

Structure de données

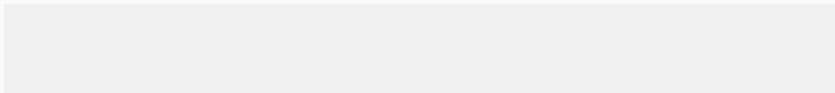
```
typedef struct _queue {  
    int *data;  
    int tail, capacity;  
} queue;
```

File basée sur un tableau

- Initialisation ?

File basée sur un tableau

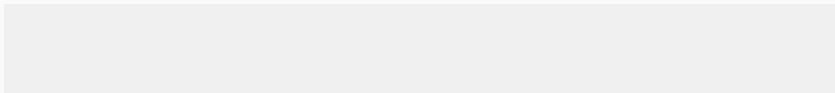
- Initialisation ?



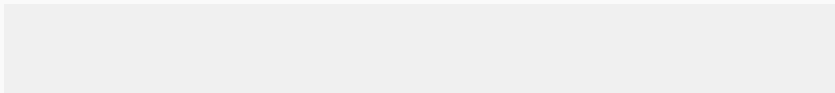
- Est vide ?

File basée sur un tableau

- Initialisation ?



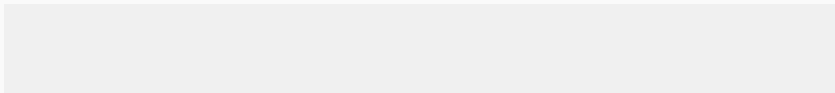
- Est vide ?



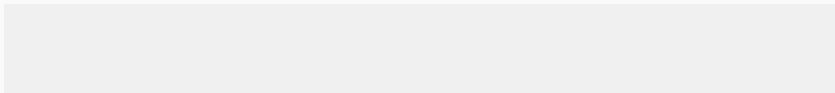
- Enfiler ?

File basée sur un tableau

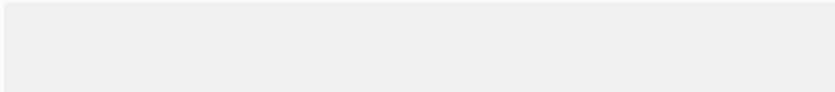
- Initialisation ?



- Est vide ?



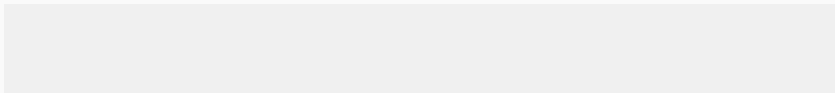
- Enfiler ?



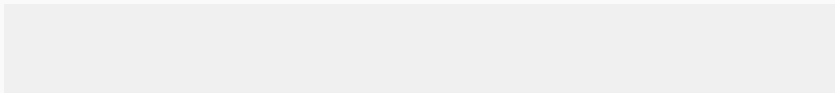
- Défiler ?

File basée sur un tableau

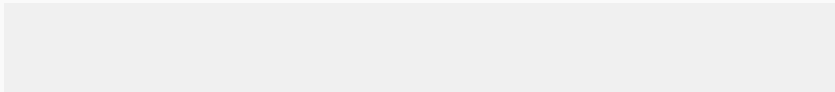
- Initialisation ?



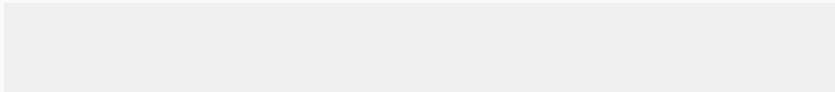
- Est vide ?



- Enfiler ?



- Défiler ?



Complexité

Quelles sont les complexités de

- Initialisation ?

Complexité

Quelles sont les complexités de

- Initialisation ?

- Est vide ?

Complexité

Quelles sont les complexités de

- Initialisation ?

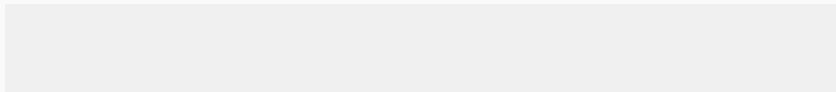
- Est vide ?

- Enfiler ?

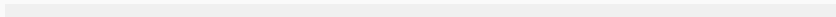
Complexité

Quelles sont les complexités de

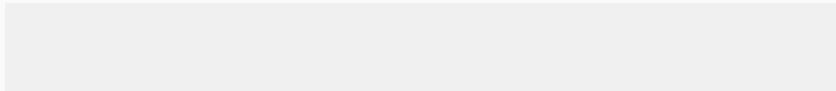
- Initialisation ?



- Est vide ?



- Enfiler ?



- Défiler ?

Complexité

Quelles sont les complexités de

- Initialisation ?

- Est vide ?

- Enfiler ?

- Défiler ?

Une file plus efficace

Comment faire une file plus efficace ?

- Où est-ce que ça coince ?

Une file plus efficace

Comment faire une file plus efficace ?

- Où est-ce que ça coince ?
- Défiler est particulièrement lent $\mathcal{O}(N)$.

Solution ?

Une file plus efficace

Comment faire une file plus efficace ?

- Où est-ce que ça coince ?
- Défiler est particulièrement lent $\mathcal{O}(N)$.

Solution ?

- Utiliser un indice séparé pour head.

```
typedef struct _queue {  
    int *data;  
    int head, tail, capacity;  
} queue;
```

Une file plus efficace (implémentation)

Enfilage

```
void queue_enqueue(queue *fa, int val) {
    if ((fa->head == 0 && fa->tail == fa->capacity-1) ||
        (fa->tail == (fa->head-1))) {
        return; // queue is full
    }
    if (fa->head == -1) { // queue was empty
        fa->head = fa->tail = 0;
    } else if (fa->tail == fa->capacity-1 && fa->head != 0) {
        // the tail reached the end of the array
        fa->tail = 0;
    } else {
        // nothing particular
        fa->tail += 1;
    }
    fa->data[fa->tail] = val;
}
```

Une file plus efficace (implémentation)

Défilage

```
void queue_dequeue(queue *fa, int *val) {
    if (queue_is_empty(*fa)) {
        return; // queue is empty
    }
    *val = fa->data[fa->head];
    if (fa->head == fa->tail) { // that was the last element
        fa->head = fa->tail = -1;
    } else if (fa->head == fa->capacity-1) {
        fa->head = 0;
    } else {
        fa->head += 1;
    }
}
```

Quel but ?

- Permet de retrouver rapidement un élément.
- Utile pour la recherche de plus court chemin dans des graphes.
- Ordonnancement de processus par degré de priorité.

Comment ?

- Les implémentations les plus efficaces se basent sur les tableaux.
- Possibles aussi avec des listes chaînées.

Les listes triées

Quelle structure de données dans notre cas ?

Une liste chaînée bien sûr (oui, c'est pour vous entraîner) !

```
typedef struct _element { // chaque élément
    int data;
    struct _element *next;
} element;
typedef element* sorted_list; // la liste
```

Fonctionnalités

```
// insertion de val
sorted_list sorted_list_push(sorted_list list, int val);
// la liste est-elle vide?
bool sorted_list_is_empty(sorted_list list); // list == NULL
// extraction de val (il disparaît)
sorted_list sorted_list_extract(sorted_list list, int val);
// rechercher un élément et le retourner
element* sorted_list_search(sorted_list list, int val);
```

L'insertion (1/3)

Trois cas

1. La liste est vide.

L'insertion (1/3)

Trois cas

1. La liste est vide.

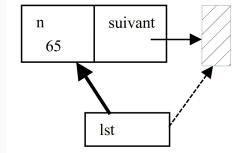


Figure 6 : Insertion dans une liste vide, `list == NULL`.

L'insertion (1/3)

Trois cas

1. La liste est vide.

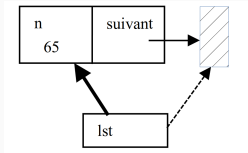


Figure 6 : Insertion dans une liste vide, `list == NULL`.

```
sorted_list sorted_list_push(sorted_list list, int val) {  
    if (sorted_list_is_empty(list)) {  
        list = malloc(sizeof(*list));  
        list->data = val;  
        list->next = NULL;  
        return list;  
    }  
}
```

L'insertion (2/3)

2. L'insertion se fait en première position.

L'insertion (2/3)

2. L'insertion se fait en première position.

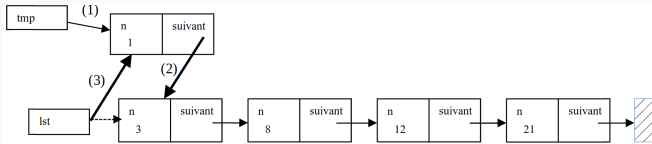


Figure 7 : Insertion en tête de liste, `list->data >= val`.

L'insertion (2/3)

2. L'insertion se fait en première position.

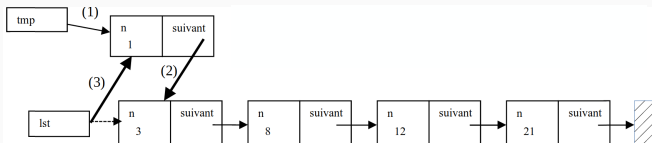


Figure 7 : Insertion en tête de liste, `list->data >= val`.

```
sorted_list sorted_list_push(sorted_list list, int val) {  
    if (list->data >= val) {  
        element *tmp = malloc(sizeof(*tmp));  
        tmp->data = val;  
        tmp->next = list;  
        list = tmp;  
        return list;  
    }  
}
```

L'insertion (3/3)

3. L'insertion se fait sur une autre position que la première.

L'insertion (3/3)

3. L'insertion se fait sur une autre position que la première.

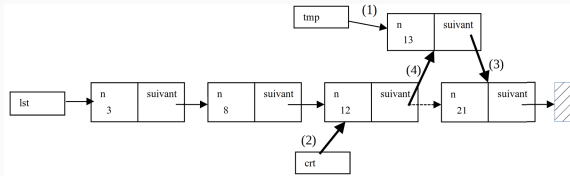


Figure 8 : Insertion sur une autre position, $list \rightarrow data < val$.

L'insertion (3/3)

3. L'insertion se fait sur une autre position que la première.

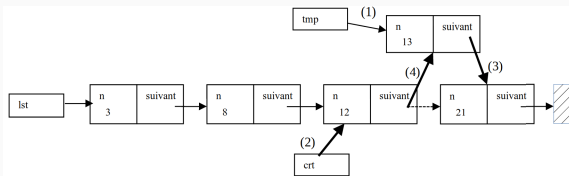


Figure 8 : Insertion sur une autre position, `list->data < val`.

```
sorted_list sorted_list_push(sorted_list list, int val) {  
    element *tmp = malloc(sizeof(*tmp));  
    tmp->data = val;  
    element *crt = list;  
    while (NULL != crt->next && val > crt->next->data) {  
        crt = crt->next;  
    }  
    tmp->next = crt->next;  
    crt->next = tmp;  
    return list;  
}
```


L'extraction (1/3)

Trois cas

1. L'élément à extraire n'est **pas** le premier élément de la liste.

L'extraction (1/3)

Trois cas

1. L'élément à extraire n'est **pas** le premier élément de la liste.

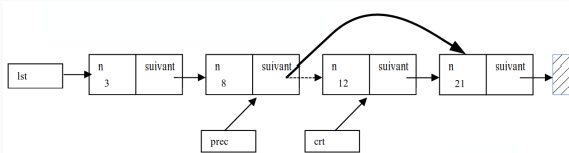


Figure 9 : Extraction d'un élément qui n'est pas le premier.

L'extraction (1/3)

Trois cas

1. L'élément à extraire n'est **pas** le premier élément de la liste.

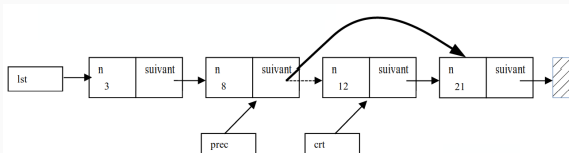


Figure 9 : Extraction d'un élément qui n'est pas le premier.

```
sorted_list sorted_list_extract(sorted_list list, int val) {
    element *prec = *crt = list; // needed to glue elements together
    while (NULL != crt && val > crt->data) {
        prec = crt;
        crt = crt->next;
    }
    if (NULL != crt && prec != crt && crt->data == val) { // glue things together
        prec->next = crt->next;
        free(crt);
    }
    return list;
}
```

L'extraction (2/3)

-
2. L'élément à extraire est le premier élément de la liste.

L'extraction (2/3)

2. L'élément à extraire est le premier élément de la liste.

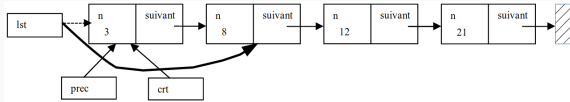


Figure 10 : Extraction d'un élément qui est le premier.

L'extraction (2/3)

2. L'élément à extraire est le premier élément de la liste.

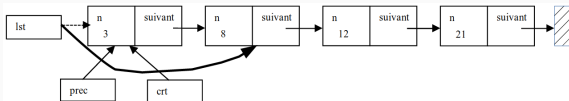


Figure 10 : Extraction d'un élément qui est le premier.

```
sorted_list sorted_list_extract(sorted_list list, int val) {  
    element *prec = *crt = list; // needed to glue elements together  
    while (NULL != crt && val > crt->data) {  
        prec = crt;  
        crt = crt->next;  
    }  
    // glue things together  
    if (NULL != crt && crt->data == val && prec == crt) {  
        list = list->next;  
        free(crt);  
    }  
    return list;  
}
```

L'extraction (3/3)

3. L'élément à extraire n'est **pas** dans la liste.
- La liste est vide.
 - La valeur est plus grande que le dernier élément de la liste.
 - La valeur est plus petite que la valeur de crt.

L'extraction (3/3)

3. L'élément à extraire n'est **pas** dans la liste.

- La liste est vide.
- La valeur est plus grande que le dernier élément de la liste.
- La valeur est plus petite que la valeur de crt.

On retourne la liste inchangée.

L'extraction (3/3)

3. L'élément à extraire n'est **pas** dans la liste.

- La liste est vide.
- La valeur est plus grande que le dernier élément de la liste.
- La valeur est plus petite que la valeur de crt.

On retourne la liste inchangée.

```
sorted_list sorted_list_extract(sorted_list list, int val) {  
    element *prec = *crt = list; // needed to glue elements together  
    while (NULL != crt && val > crt->data) {  
        prec = crt;  
        crt = crt->next;  
    }  
    if (NULL == crt || crt->data != val) { // val not present  
        return list;  
    }  
}
```

La recherche

```
element* sorted_list_search(sorted_list list, int val);
```

- Retourne NULL si la valeur n'est pas présente (ou la liste vide).
- Retourne un pointeur vers l'élément si la valeur est présente.

La recherche

```
element* sorted_list_search(sorted_list list, int val);
```

- Retourne NULL si la valeur n'est pas présente (ou la liste vide).
- Retourne un pointeur vers l'élément si la valeur est présente.

```
element* sorted_list_search(sorted_list list, int val) {  
    // search for element smaller than val  
    element* pos = sorted_list_position(list, val);  
    if (NULL == pos && val == list->data) {  
        return list; // first element contains val  
    } else if (NULL != pos && NULL != pos->next  
        && val == pos->next->data)  
    {  
        return pos->next; // non-first element contains val  
    } else {  
        return NULL; // well... val's not here  
    }  
}
```

La recherche

La fonction `sorted_list_position`

```
element* sorted_list_position(sorted_list list, int val);
```

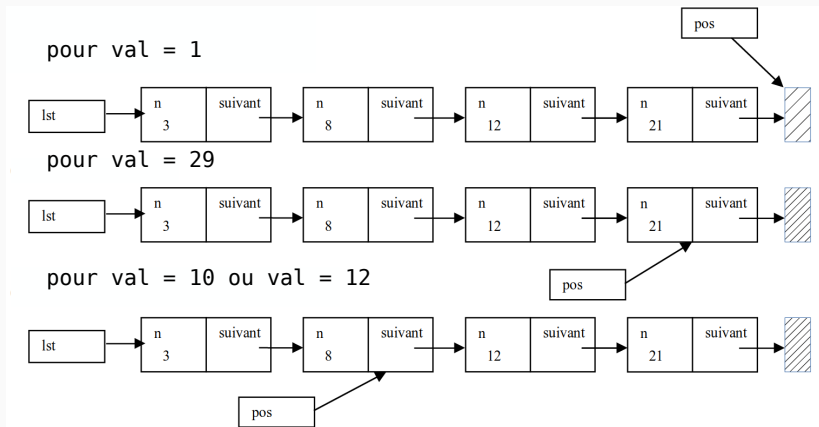


Figure 11 : Trois exemples de retour de la fonction `sorted_list_position()`.

Exercice : implémenter

```
element* sorted_list_position(sorted_list list, int val);
```

Exercice : implémenter

```
element* sorted_list_position(sorted_list list, int val);

element* sorted_list_position(sorted_list list, int val) {
    element* pos = list;
    if (sorted_list_is_empty(list) || val <= list->data) {
        pos = NULL;
    } else {
        while (NULL != pos->next && val > pos->next->data) {
            pos = pos->next;
        }
    }
    return pos;
}
```

Complexité de la liste chaînée triée

L'insertion ?

Complexité de la liste chaînée triée

L'insertion ?

$$\mathcal{O}(N).$$

L'extraction ?

Complexité de la liste chaînée triée

L'insertion ?

$$\mathcal{O}(N).$$

L'extraction ?

$$\mathcal{O}(N).$$

La recherche ?

Complexité de la liste chaînée triée

L'insertion ?

$$\mathcal{O}(N).$$

L'extraction ?

$$\mathcal{O}(N).$$

La recherche ?

$$\mathcal{O}(N).$$