Files d'attente et listes triées

Algorithmique et structures de données, 2022-2023

P. Albuquerque (B410), P. Künzli et O. Malaspinas (A401), ISC, HEPIA

2022-12-21

En partie inspirés des supports de cours de P. Albuquerque

La file d'attente (1/N)

- 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

La file d'attente (1/N)

- 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.

La file d'attente (1/N)

- 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 liste vide.
- Détruire une liste vide.

La file d'attente (2/N)

Implémentation possible

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

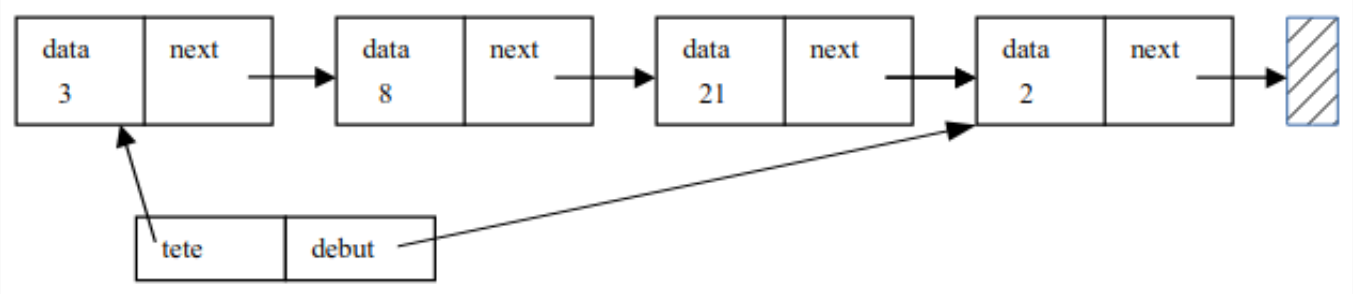

Figure 1: Illustration d'une file d'attente.

Structure de données en C?

La file d'attente (2/N)

Implémentation possible

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

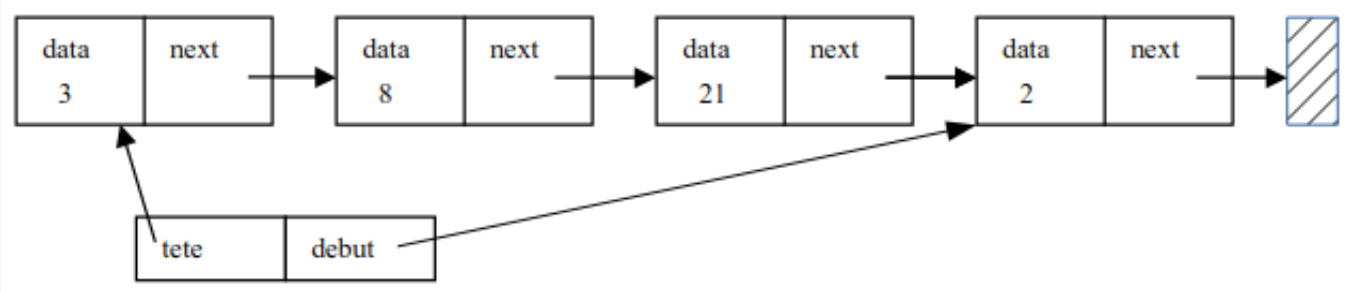

Figure 1: Illustration d'une file d'attente.

Structure de données en C?

```
typedef struct _element {  // Elément 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
} queue;
```

Fonctionnalités d'une file d'attente

Creation et consultations

Fonctionnalités d'une file d'attente

Creation et consultations

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

Manipulations et destruction

Fonctionnalités d'une file d'attente

Creation et consultations

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

Enfilage

Deux cas différents:

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

Enfilage

Deux cas différents:

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

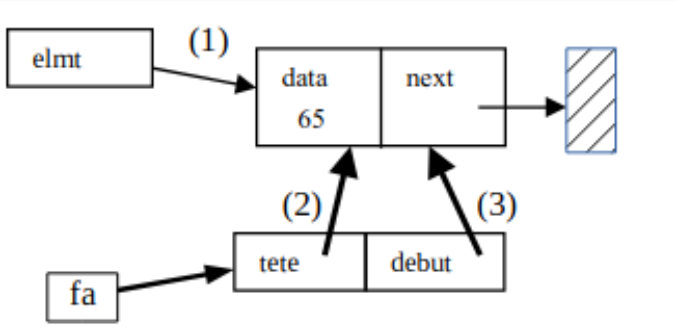

Figure 2: Insertion dans une file d'attente vide.

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

Enfilage

Deux cas différents:

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

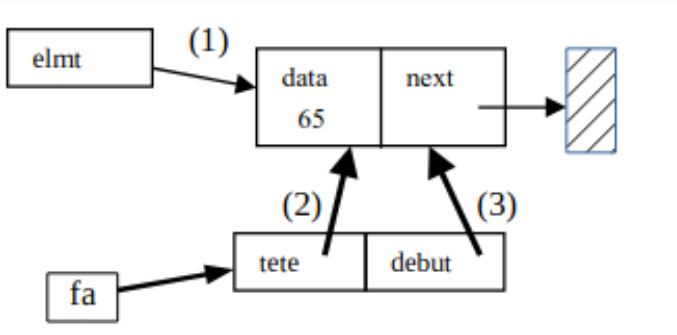

Figure 2: Insertion dans une file d'attente vide.

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

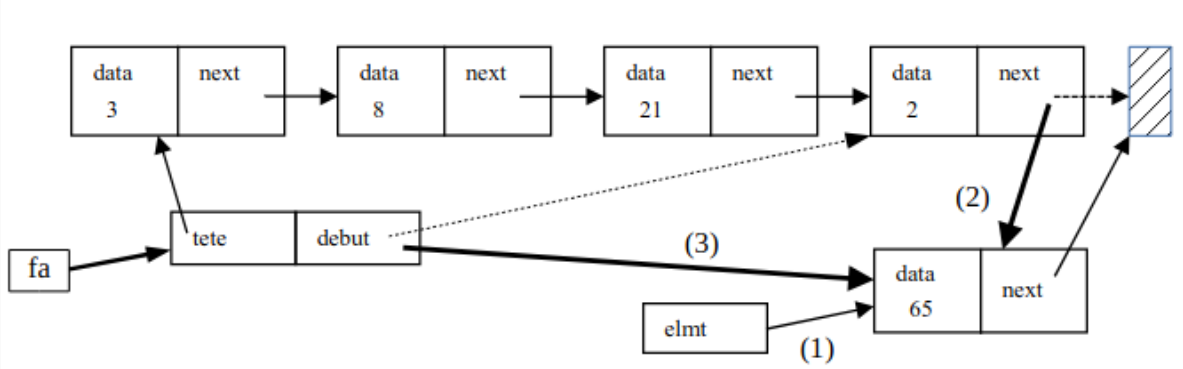

Figure 3: Insertion dans une file d'attente non-vide.

Enfilage

Live (implémentation)

Enfilage

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;
        fa->tail = elmt;
    } else {
        fa->tail->next = elmt;
        fa->tail = elmt;
    }
}
```

Défilage

Trois cas différents

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

Défilage

Trois cas différents

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

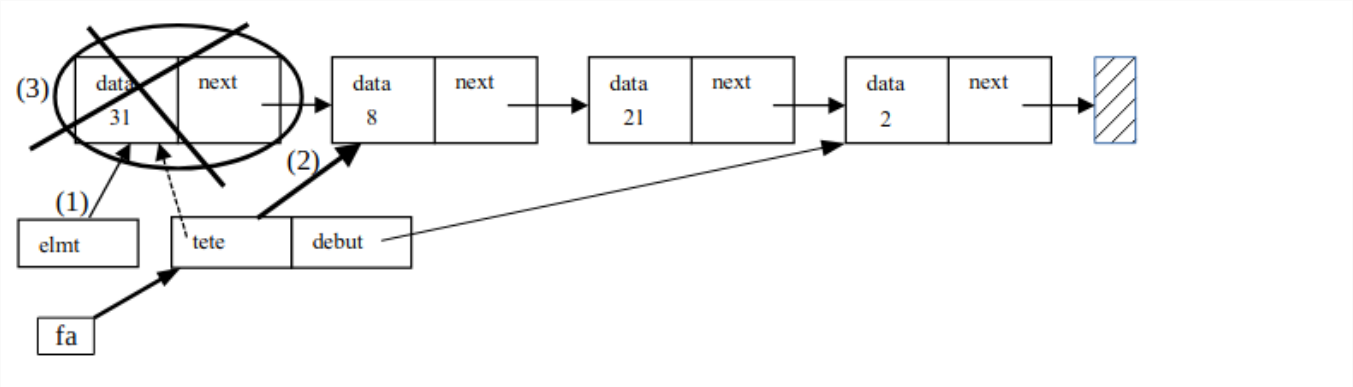

Figure 4: Extraction d'une file d'attente

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

Défilage

Trois cas différents

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

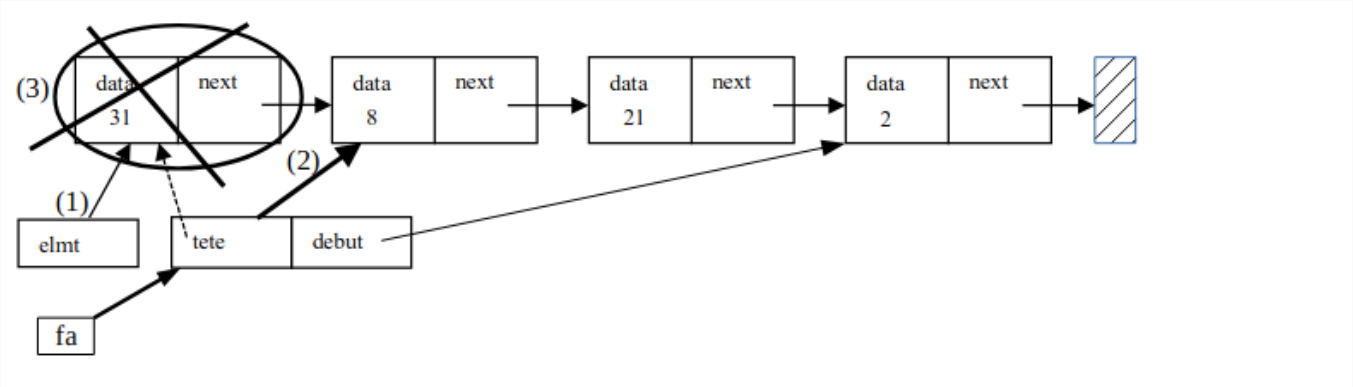

Figure 4: Extraction d'une file d'attente

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

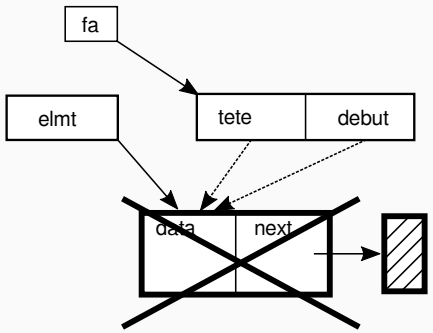

Figure 5: Extraction d'une file d'attente de longueur 1.

Défilage

Live (implémentation)

Défilage

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;
}
```

Défilage

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?

Destruction

Comment on faire la désallocation?

Destruction

Comment on faire la désallocation?

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

Complexité

Quelle sont les complexité de:

- Enfiler?

Complexité

Quelle sont les complexité de:

- Enfiler?
- Défiler?

Complexité

Quelle sont les complexité de:

- Enfiler?
- Défiler?
- Détruire?

Complexité

Quelle sont les complexité de:

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

Implémentation alternative

Comment implémenter la file autrement?

Implémentation alternative

Comment implémenter la file autrement?

- Données stockées dans un tableau;
- Tableau de taille connue à la compilation ou pas (réallouable);
- `tail` seraient les indices du tableau;
- `capacity` seraient 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` seraient les indices du tableau;
- `capacity` seraient 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

Quelle sont les complexités de:

- Initialisation?

Complexité

Quelle sont les complexités de:

- Initialisation?

- Est vide?

Complexité

Quelle sont les complexités de:

- Initialisation?

- Est vide?

- Enfiler?

Complexité

Quelle sont les complexités de:

- Initialisation?
- Est vide?
- Enfiler?
- Défiler?

Complexité

Quelle 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) % (fa->capacity-1))) {
        return; // queue is full
    }
    if (fa->head == -1) { // queue was empty
        fa->head = fa->tail = 0;
        fa->data[fa->tail] = val;
    } else if (fa->tail == fa->capacity-1 && fa->head != 0) {
        // the tail reached the end of the array
        fa->tail = 0;
        fa->data[fa->tail] = val;
    } 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;
    }
}
```

Les listes triées

Une liste chaînée triée est:

- une liste chaînée
- dont les éléments sont insérés dans l'ordre.

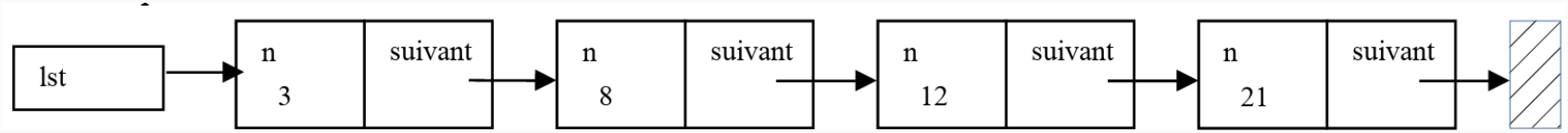

Figure 6: Exemple de liste triée.

Les listes triées

Une liste chaînée triée est:

- une liste chaînée
- dont les éléments sont insérés dans l'ordre.

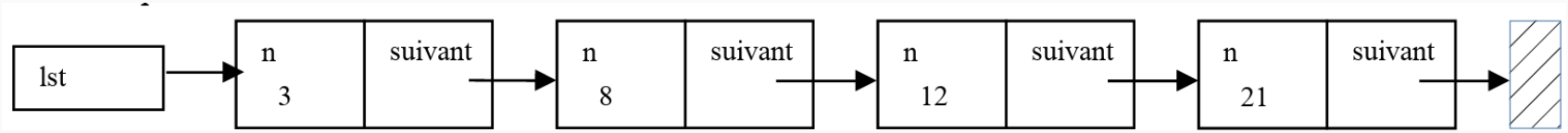

Figure 6: Exemple de liste triée.

- L'insertion est faite telle que l'ordre est maintenu.

Quelle structure de données?

Les listes triées

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 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

Trois cas

1. La liste est vide.

L'insertion

Trois cas

1. La liste est vide.

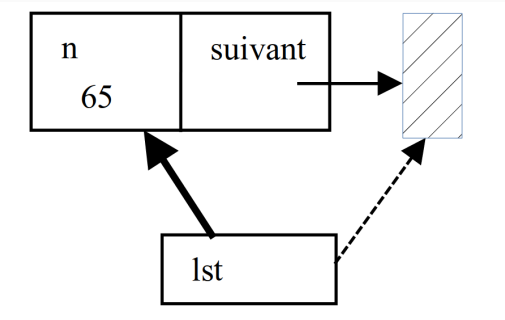

Figure 7: Insertion dans une liste vide, `list == NULL`.

L'insertion

Trois cas

1. La liste est vide.

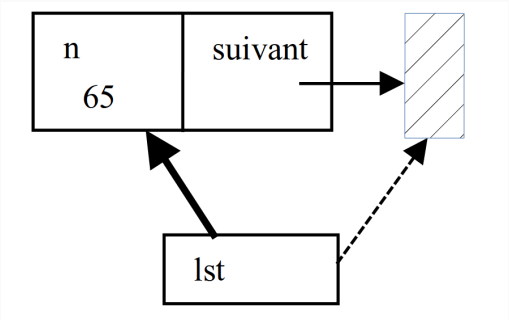

Figure 7: Insertion dans une liste vide, `list == NULL`.

```
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. L'insertion se fait en première position.

L'insertion

2. L'insertion se fait en première position.

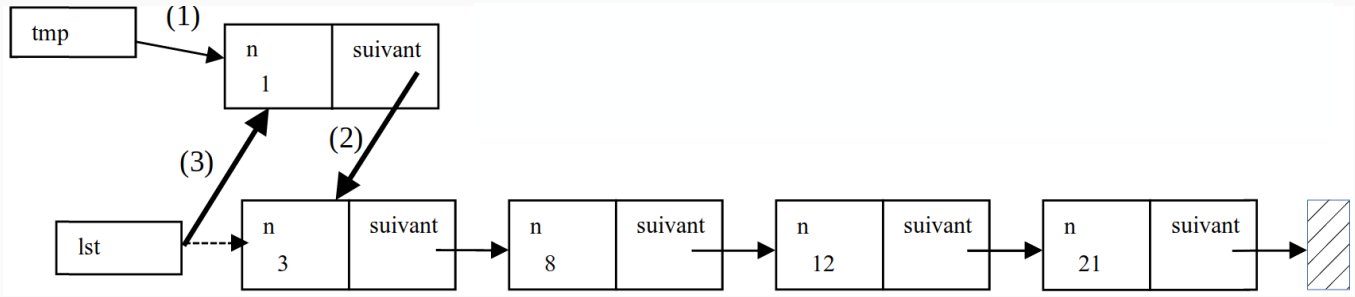

Figure 8: Insertion en tête de liste, `list->data >= val`.

L'insertion

2. L'insertion se fait en première position.

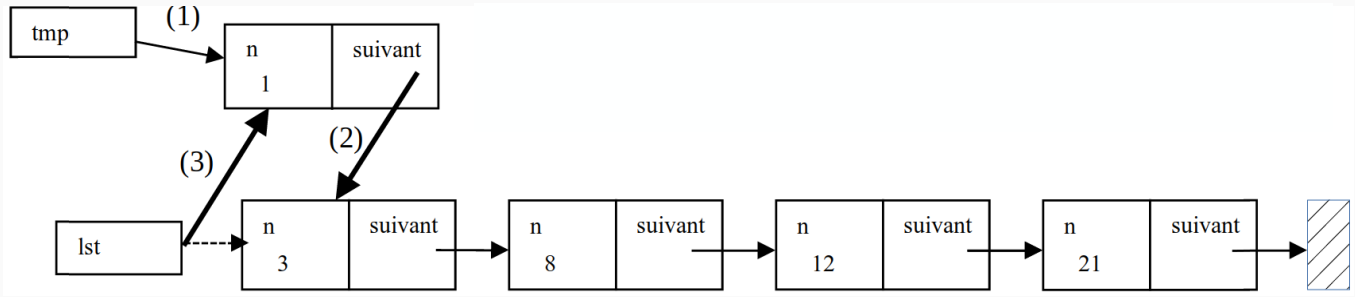

Figure 8: Insertion en tête de liste, `list->data >= val`.

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

L'insertion

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

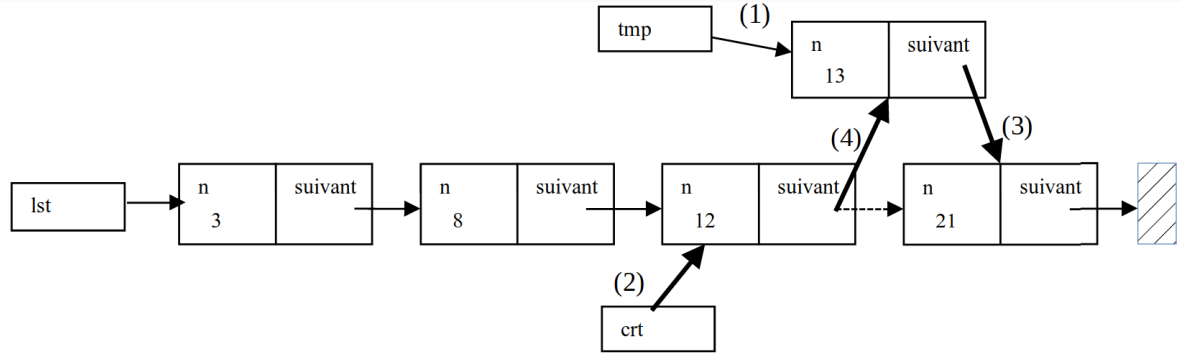

Figure 9: Insertion sur une autre position, list->data $<$ val.

L'insertion

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

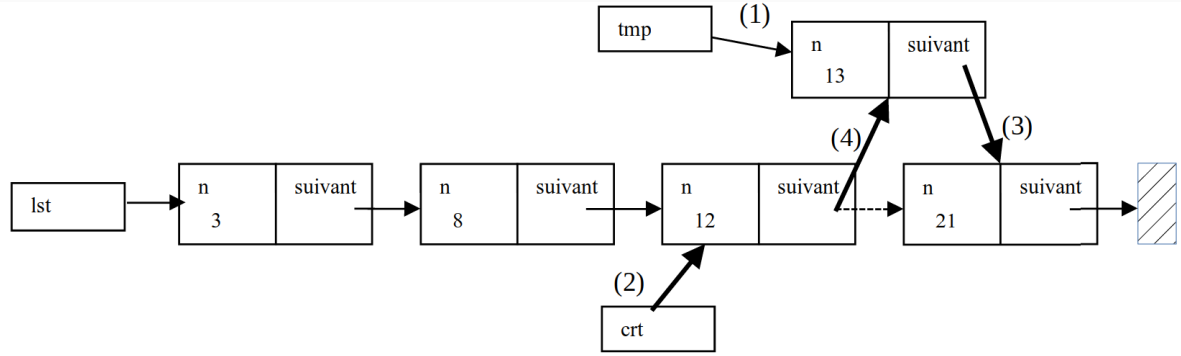

Figure 9: Insertion sur une autre position, list->data < val.

```
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;
}
```