3

STRUCTURES DE DONNÉES COMPLEXES

3.1 INTRODUCTION

Dans ce chapitre nous introduisons les structures de données complexes. Nous présentons les types usuels, leurs définitions en C++ et les différentes primitives correspondants.

Chacune de ces structures est accompagnée de figures illustratives permettant une meilleure compréhension des différents concepts et fonctions. Comme nous essayons d'étudier et de mettre en avant la présentation de la structure en mémoire et sa mise en œuvre et son emplacement dans un programme C++.

Ce chapitre est organisé de la manière suivante :

- La première section introduit les notions des structures linéaires (ou séquentielles). Elle regroupe les listes contiguës, les listes chaînées, les piles et les files.
- La deuxième section traite les structures arborescences, à savoir les arbres et les graphes.

3.2 STRUCTURES LINÉAIRES

Les structures linéaires permettent de représenter en mémoire des données de même type. L'une des caractéristiques fondamentales de ces structures est que chaque élément (ou donnée) possède un successeur sauf le dernier.

Dans ce cours nous introduirons quatre structures linéaires, à savoir les listes chaînées (simples et double), les piles et les files.

3.2.1 Listes chaînées simples

Une liste chaînée représente un ensemble d'éléments organisés séquentiellement, tout comme un tableau. Sauf que dans un tableau, l'organisation séquentielle est donnée implicitement par la position dans le tableau. Dans une liste chaînée, on utilise un arrangement explicite dans lequel chaque élément appartient à un nœud qui contient un lien au nœud suivant. La figure 1 [Sedgewick,1991]

montre une liste chaînée dans laquelle les éléments sont représentés par des lettres, les nœuds par des cercles et les liens par des segments reliant les nœuds.

Il est habituel de trouver un nœud sentinelle à premier bout de la liste, ce nœud appelé "début" ou *entête*, pointera sur le premier élément de la liste.

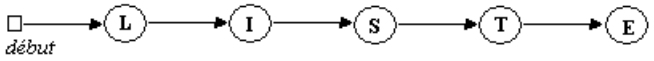


FIGURE 1 – Liste chaînée

Définition de la liste en C++

Une liste chaînée est une structure composée d'une suite de maillons, dont chacun porte une donnée et une valeur adresse qui pointe sur le maillon suivant.

Afin d'avoir accès aux éléments (ou maillons) de la liste chaînée, nous avons besoin d'un élément sentinelle, appelé *entête* qui doit pointer sur le premier.

La déclaration en C++ du maillon et de l'entête de la liste se donne comme suit :

```
struct liste
{
        TypeElt elt;
        liste * suivant;
}
liste * tete = NULL;
```

L'entête de la liste est initialisé à $NULL$, puisque la liste est initialement vide.

La figure ci-dessous 2 représente la structure d'un maillon i, qui est composé de deux champs *elt* de valeur val_i et l'adresse en mémoire $@_i$. La valeur val_i est la valeur de la donnée de type *TypeElt* que nous devons charger dans la liste.

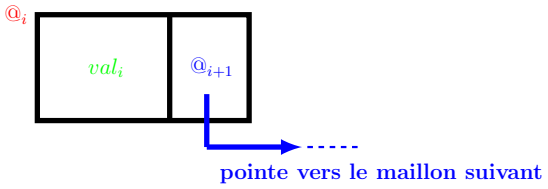


FIGURE 2 – Structure d'un maillon

Le lien entre les différents éléments est réalisé par un chaînage simple, tout en vérifiant la relation de *successeur* entre ces éléments.

La figure 3 représente la structure d'une liste simplement chaînée, qui est composée de $n + 1$ éléments.

Nous constatons que chaque élément i possède une adresse $@_i$ et contient une donnée et pointe sur l'élément suivant (ou l'élément successeur).

Contrairement à la structure tableau, où les éléments sont superposés d'une manière contiguë en mémoire, la disposition des éléments d'une liste chaînée

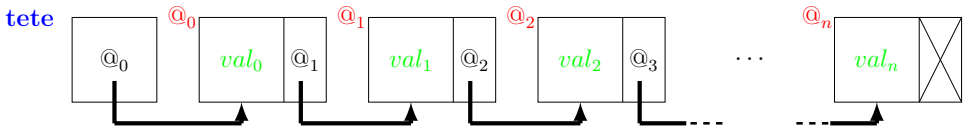


FIGURE 3 – Structure d'une liste simplement chaînée

dépend des emplacements libres en mémoire. Ce qui donne l'impression que ces éléments sont éparpillés en mémoire, tel illustré dans la figure 4.

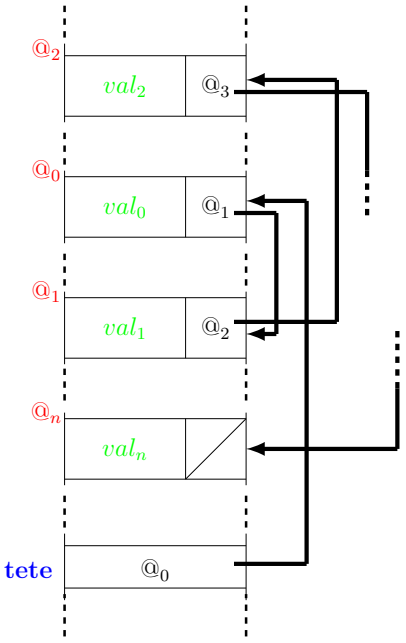


FIGURE 4 – Illustration d'une liste simplement chaînée en mémoire

Création d'un maillon

Sachant qu'une liste chaînée est une structure dynamique, on fait correspondre à chaque apparition d'un nouvel élément à insérer, une allocation dynamique d'un espace mémoire suffisant.

Nous avons ci-dessous la fonction *liste * creer_maillon(TypeElt)* qui permet de réserver l'espace mémoire nécessaire pour le maillon et charger la valeur de la donnée de type *TypeElt*. Elle retourne l'adresse du maillon (de type *liste* *).

```
liste * creer_maillon(TypeElt val)
{
  liste * maillon = new liste;
  if (maillon) {
    maillon->elt = val;
    maillon->suivant = NULL
  }
  return maillon;
}
```

Dans le cas où l'espace mémoire est insuffisant, la fonction retourne la valeur *NULL*, ce qui exprime l'impossibilité de créer le maillon.

Ajout d'un élément

La taille de la liste n'est pas connue au préalable, elle démarre de 0 et s'incrémente à chaque insertion d'un nouveau maillon.

L'insertion d'un maillon dans la liste se donne en C++ comme suit :

1. Il faut tout d'abord créer le nouvel élément et saisir ou stocker les valeurs.
2. Insérer cet élément dans la liste. La cohérence de la liste doit finalement être rétablie en indiquant que le nouvel élément est désormais le successeur de celui après lequel il va prendre place.

L'insertion dans une liste chaînée simple peut se faire au début, au milieu et à la fin.

A. Insertion en tête de liste : La fonction d'insertion à l'entête de la liste se donne via la fonction *void ajout_tete(liste*&, TypeElt)*. L'élément à insérer est créé en faisant appel à la fonction *creer_maillon*.

```
void ajout_tete(liste*& tete, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = tete;
    tete = maillon;
  }
}
```

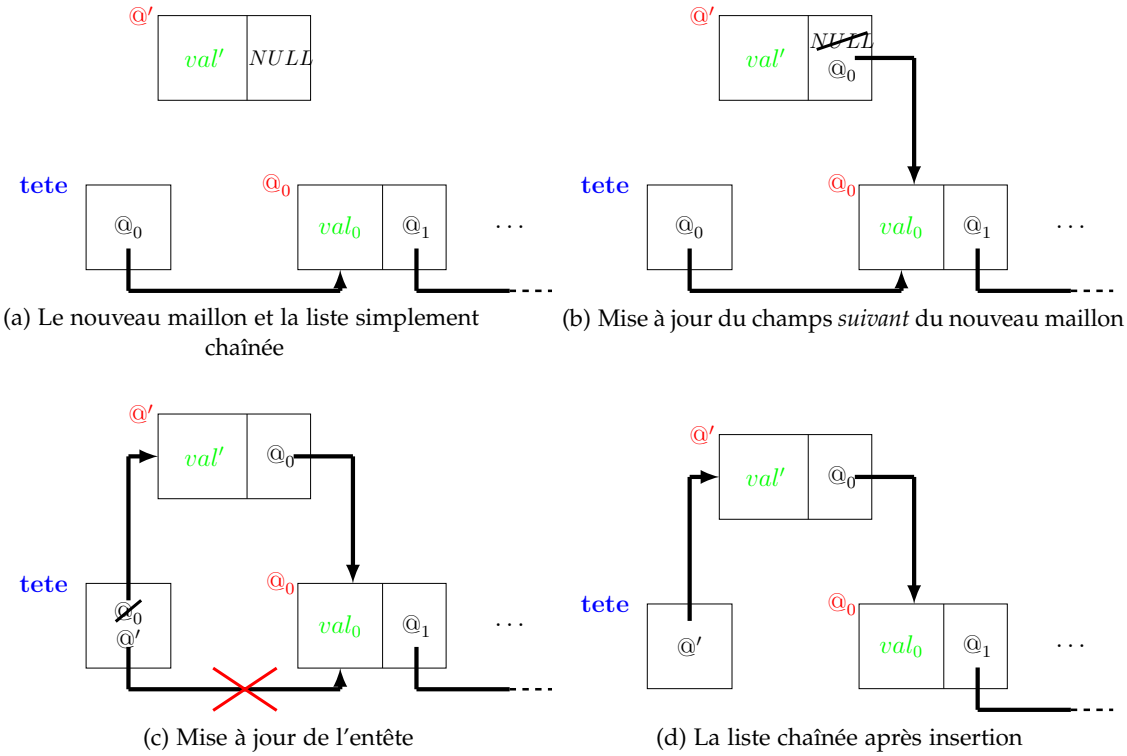
Dans le cas où le maillon est déjà présent en mémoire, on se contente de mettre son adresse en entrée de la fonction. Ce qui donne en C++ la version suivante :

```
void ajout_tete(liste *& tete, liste * maillon)
{
    maillon->suivant=tete;
    tete = maillon;
}
```

La figure ci-dessous 5 illustre les différentes étapes d'insertion d'un maillon à l'entête d'une liste chaînée.

Initialement la variable *tete* pointe sur le premier élément ayant l'adresse mémoire $@_0$, le maillon à insérer est présent en mémoire dont l'adresse est $@'$ (voir la sous-figure 5a.

L'instruction *maillon->suivant = tete* est illustrée dans la sous-figure 5b. Le champ suivant du nouveau maillon dont la valeur initiale est *NULL*, est mis à jour par l'adresse du premier élément de la liste. Par conséquent, le premier élément de la liste devient le successeur du nouveau.



(a) Le nouveau maillon et la liste simplement chaînée

(b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour de l'entête

(d) La liste chaînée après insertion

FIGURE 32 – Insertion d'un maillon en tête de liste

La sous-figure suivante 32c montre l'impact de l'exécution de *tete* = *maillon* sur la liste. L'entête est mis à jour, il pointe sur le nouveau maillon d'adresse @′.

Enfin, nous avons la sous-figure 32d qui illustre l'état de la liste après l'exécution de la fonction *ajout_tete*. Nous avons une nouvelle liste dont le nouveau maillon devient le premier et pointe sur un deuxième élément d'adresse $@_0$.

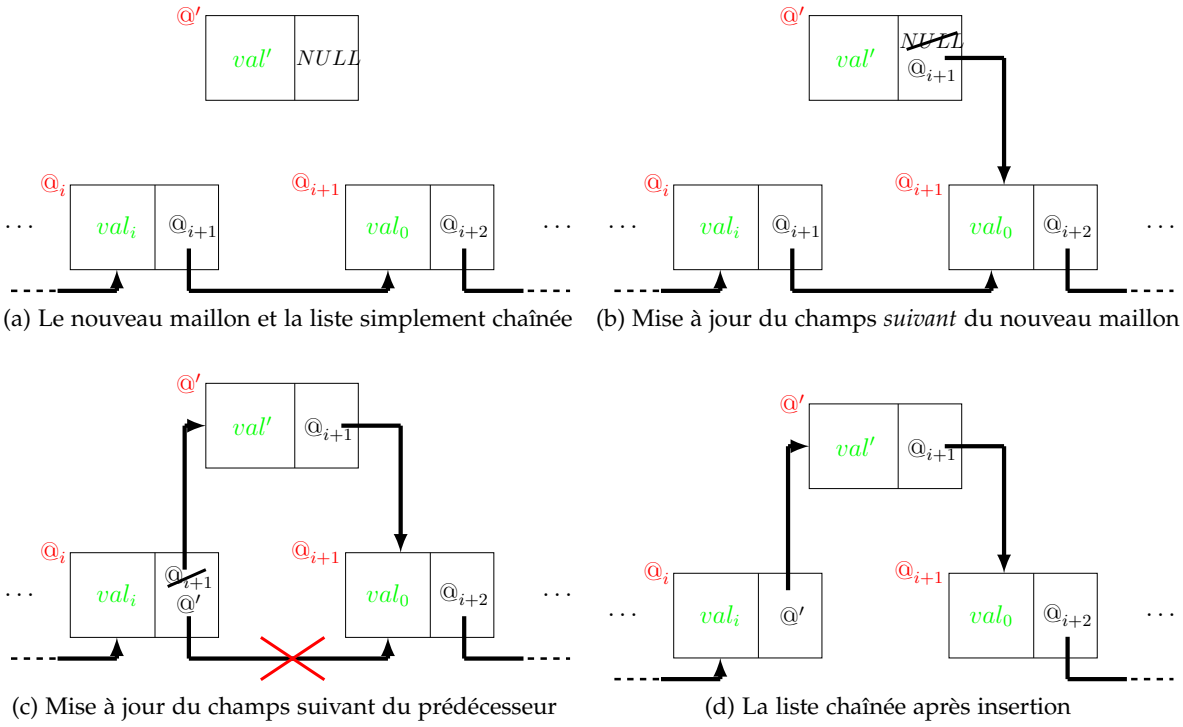
B. Insertion au milieu de la liste : L'insertion au milieu de la liste impose la connaissance de l'adresse de l'élément qui doit précéder le nouveau maillon.

La fonction d'insertion au milieu de la liste se donne comme suit :

```
void ajout_milieu(liste * pred, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = pred->suivant;
    pred->suivant = maillon;
  }
}
```

Sachant que *pred* est l'adresse de l'élément qui va précéder l'élément à insérer.

Nous montrons via la figure ci-dessous 33 les différentes étapes d'insertion d'un maillon au milieu d'une liste chaînée, entre les deux maillons dont les adresses respectives $@_i$ et $@_{i+1}$.



(a) Le nouveau maillon et la liste simplement chaînée

(b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour du champs suivant du prédécesseur

(d) La liste chaînée après insertion

FIGURE 33 – Insertion d'un maillon au milieu de la liste

Initialement le maillon i est le prédécesseur du maillon $i+1$, qui joue à son tour le rôle de successeur, le maillon à insérer est présent en mémoire dont l'adresse est @′ (voir la sous-figure 33a.

L'application de l'instruction *maillon->suivant = pred->suivant* sur la liste est illustrée via la sous-figure 33b. Le champ suivant du nouveau maillon dont la valeur initiale est $NULL$, est remplacé par l'adresse du successeur du $i^{ème}$ élément : $@_{i+1}$. Par conséquent, le successeur de l'élément i devient le successeur du nouveau maillon.

La sous-figure suivante 33c montre l'impact de l'exécution de *pred->suivant = maillon* sur la liste. La variable *pred* contient l'adresse @i du $i^{ème}$ élément, dont le champ suivant est mis à jour et pointe sur le nouveau maillon d'adresse @′.

Le résultat final de l'exécution de la fonction *ajout_milieu* est illustré dans la sous-figure 33d. Nous avons une nouvelle liste dont le nouveau maillon est bien inséré entre les éléments i et $i+$.

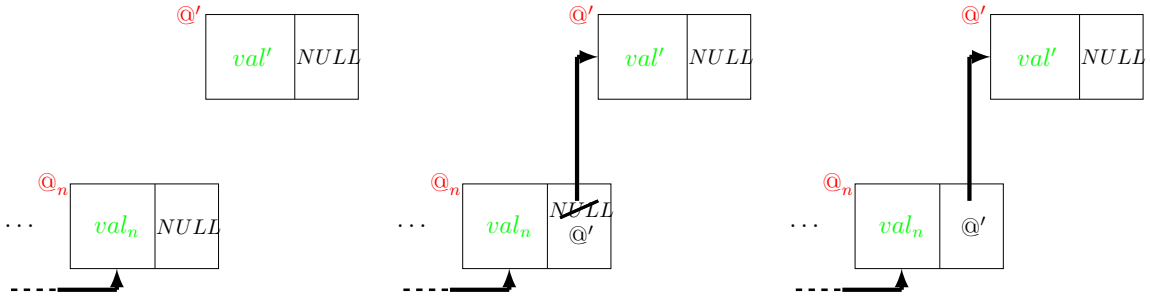
C. Insertion en queue de la liste : Afin d'insérer un maillon en queue de la liste, nous devions avoir en entrée l'adresse du maillon en queue (ou en fin) de la liste.

La fonction d'insertion en queue de la liste se donne comme suit :

```
void ajout_queue(liste * q, TypeElt val)
{
  liste *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = NULL;
    q->suivant = maillon;
  }
}
```

Avec, q l'adresse de l'élément en queue de liste avant l'insertion.

Nous montrons via la figure ci-dessous 34 les différentes étapes d'insertion d'un maillon en queue de la liste chaînée.



(a) Le nouveau maillon et la liste simplement chaînée

(b) Mise à jour du champs *suivant* du dernier maillon

(c) La liste après insertion

FIGURE 34 – Insertion d'un maillon en queue de liste

Initialement, tel illustré dans la sous-figure 33a, le maillon en queue de la liste possède l'adresse $@_n$ et il ne pointe sur aucun autre élément, le champ suivant est *NULL* et le maillon à insérer est déjà créé à l'adresse est $@'$.

L'application de l'instruction *q->suivant = maillon* sur la liste est donnée via la sous-figure 33b. Le champ suivant du $n^{ème}$ maillon est mis à jour en lui affectant l'adresse $@'$.

La sous-figure suivante 33c montre l'impact de la fonction *ajout_queue* sur la liste. Nous avons une nouvelle liste dont le nouveau maillon est bien inséré en queue de la liste.

Suppression d'un élément d'une liste chaînée

La création d'un élément d'une liste chaînée fait appel à une allocation dynamique de la mémoire. Lorsque certains (ou presque tous les éléments) de la liste ne sont plus utiles, il est donc nécessaire de restituer cette mémoire au système.

La logique des opérations est assez simple :

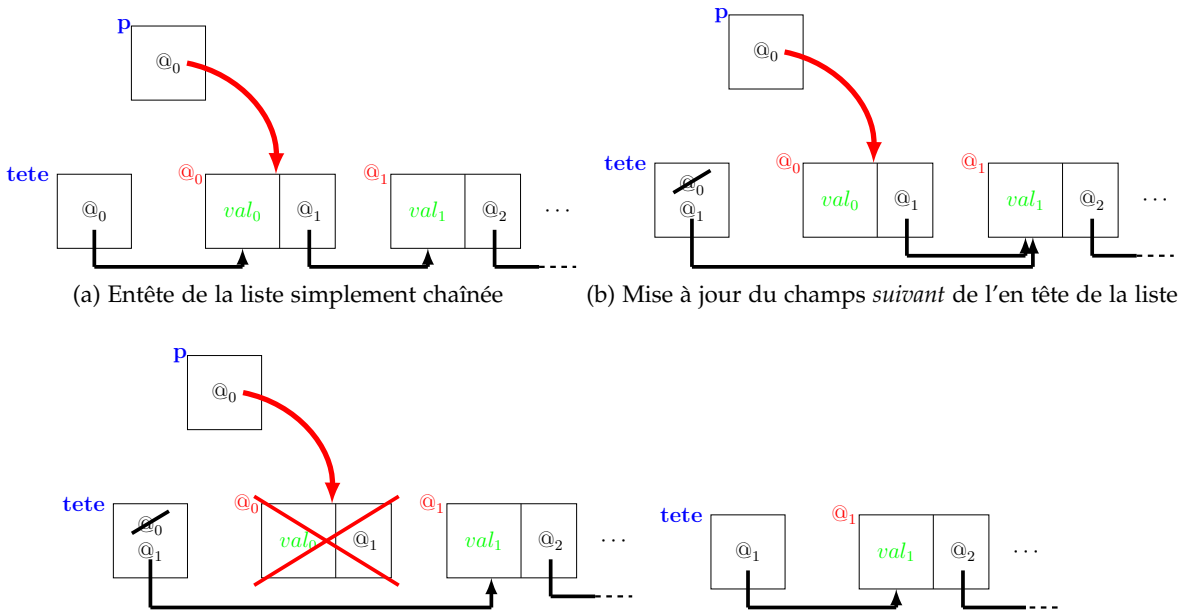
1. L'élément précédant celui qu'on doit supprimer doit adopter comme successeur le successeur de ce dernier.
2. L'espace mémoire réservé à l'élément supprimé de la liste doit être libéré.

Pareillement à l'insertion, la suppression dans une liste chaînée est autorisée au début, au milieu et en queue de la liste.

A. Suppression à l'entête de la liste : Nous donnons ci-dessous la fonction *void supprimer_tete(liste *& tete)* qui permet la suppression du premier élément de la liste.

```
void supprimer_tete(liste * & tete)
{
  if (tete != NULL)
  {
    liste * p = tete;
    tete = p->suivant;
    delete p;
  }
}
```

La figure ci-dessous 35 illustre les différentes étapes d'exécution de la fonction *supprimer_tete*.



(a) Entête de la liste simplement chaînée

(b) Mise à jour du champs *suivant* de l'en tête de la liste

(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 35 – Suppression d'un maillon à l'entête de la liste

Tel illustré dans la sous-figure 35a, initialement la variable *tete* pointe sur le premier élément ayant l'adresse mémoire $@_0$, qui pointe à son tour sur l'élément d'adresse $@_1$.

L'effet de l'instruction *tete = p->suivant* sur la liste se donne via la sous-figure 35b. L'entête est mis à jour par l'adresse du deuxième élément de la liste $@_1$. Par conséquent, le premier élément de la liste devient l'élément qui prenait auparavant la deuxième position.

La sous-figure suivante 35c montre l'impact de l'instruction *delete p* sur le maillon supprimé de la liste et l'espace mémoire qui lui a été réservé. La case mémoire d'adresse $@_0$ et qui contenait la valeur val_0 et l'adresse du successeur $@_1$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 35d qui montre l'état de la liste après exécution de *supprimer_tete*. Nous avons une nouvelle liste dont l'entête pointe sur l'élément d'adresse $@_1$.

B. Suppression au milieu de la liste : La fonction ci-dessous dont l'entête *void supprimer_milieu(liste * pred)* permet la suppression d'un maillon au milieu de la liste, avec *pred* paramètre qui contient l'adresse de son prédécesseur.

```
void supprimer_milieu(liste * pred)
{
  if (pred->suivant != NULL)
  {
    liste * p = pred->suivant;
    pred->suivant = p->suivant;
    delete p;
  }
}
```

La figure ci-dessous 36 illustre les différentes étapes de suppression du maillon d'adresse $@_i$ et dont le prédécesseur et le successeur sont d'adresses respectives $@_{i-1}$ et $@_{i+1}$.

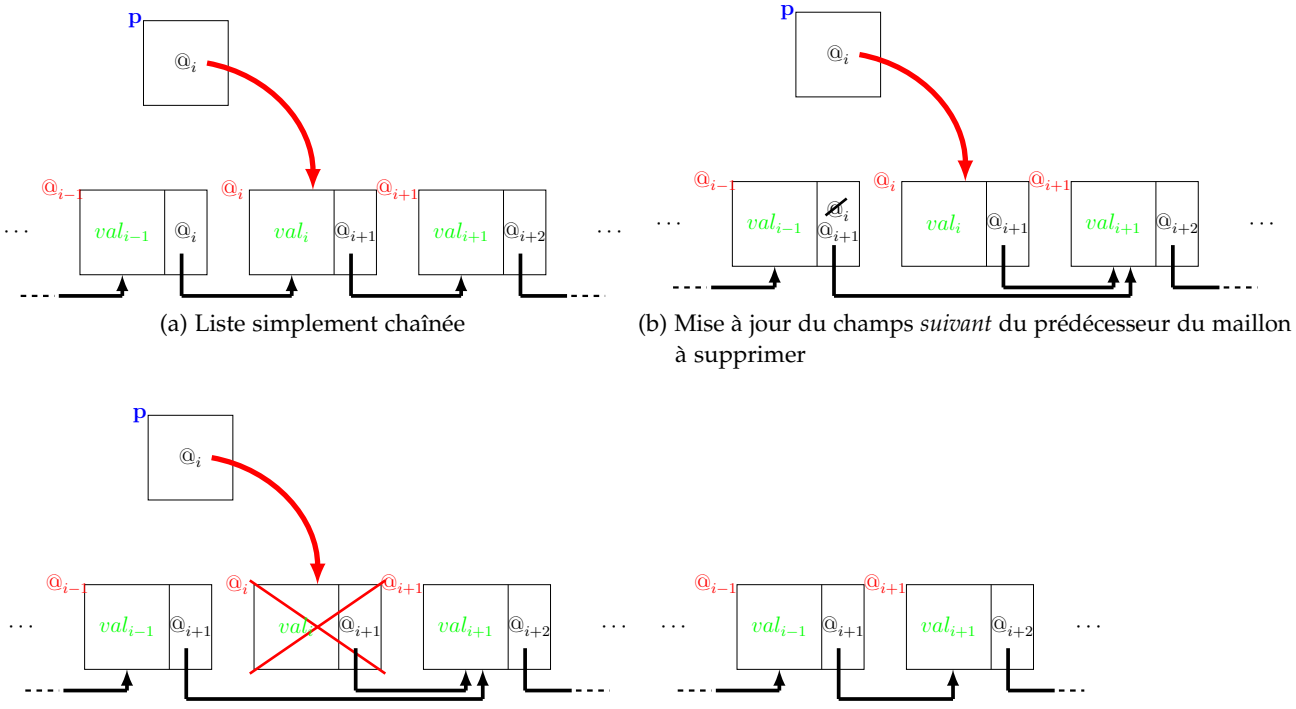
La sous-figure 36a illustre l'état initial de la liste. Le champ *suivant* du prédécesseur pointe sur l'élément ayant l'adresse mémoire $@_i$, qui pointe à son tour sur l'élément d'adresse $@_{i+1}$. La case mémoire qui correspond à la variable *p* contient l'adresse de l'élément à supprimer $@_i$, résultat de l'exécution de l'instruction *liste * p = pred->suivant*.

L'instruction *pred->suivant = p->suivant* est illustrée dans la sous-figure 36b. Le champ suivant du maillon $i-1$ est mis à jour par l'adresse de l'élément $i+1$. Par conséquent, l'élément $i+1$ de la liste devient l'élément successeur de l'élément $i-1$.

La sous-figure suivante 36c montre l'impact de l'exécution de *delete p* sur le maillon supprimé. La case mémoire d'adresse $@_i$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 36d qui illustre l'état final de la liste après application de la fonction *void supprimer_milieu(liste *&, TypeElt)*.

C. Suppression en queue de la liste : Ci-dessous, nous décrivons la primitive *void supprimer_queue(liste * pred)* de suppression du dernier élément de la liste. Le paramètre *pred* correspond à l'adresse de l'avant dernier maillon de notre structure.



(a) Liste simplement chaînée

(b) Mise à jour du champs *suivant* du prédécesseur du maillon à supprimer

(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 36 – Suppression d'un maillon au milieu de liste

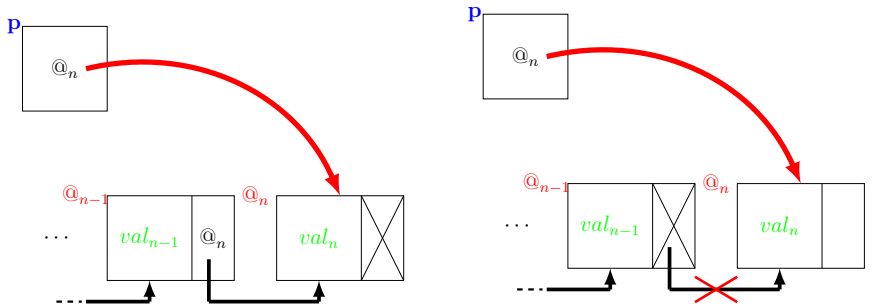
```
void supprimer_queue(liste * pred)
{
  if (pred->suivant != NULL)
  {
    liste * p = pred->suivant;
    pred->suivant = NULL;
    delete p;
    }
}
```

La figure ci-dessous 37 illustre les différentes étapes de suppression d'un maillon au milieu d'une liste chaînée.

On suppose qu'on désire supprimer le maillon d'adresse $@_i$, situé entre ceux d'adresses respectives $@_{i-1}$ et $@_{i+1}$.

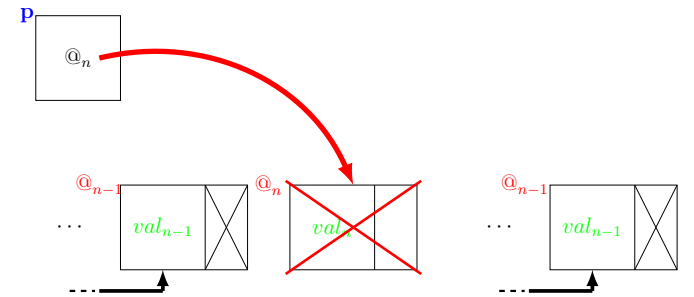
La sous-figure 37a illustre l'état initial de la liste. Le champ *suivant* de l'avant dernier élément *pred* pointe sur le dernier élément d'adresse $@_n$. La variable p contient l'adresse de l'élément à supprimer $@_n$, ce qui est le résultat de l'exécution de l'instruction *liste * p = pred->suivant*.

L'instruction *pred->suivant = NULL* est illustrée dans la sous-figure 37b. Le champ suivant du maillon $n-1$ est mis à *NULL*, puisqu'il ne doit pointer sur aucun autre élément.



(a) Liste simplement chaînée

(b) Mise à jour du champs *suivant* du prédécesseur du maillon à supprimer



(c) Libération de l'espace mémoire occupé par le maillon supprimé de la liste

(d) La liste chaînée après suppression

FIGURE 37 – Suppression d'un maillon au milieu de liste

La sous-figure suivante 37c montre l'impact de l'exécution de *delete p* sur le maillon supprimé de la liste. La case mémoire d'adresse $@_n$ est libérée afin qu'elle puisse être exploitée par d'autres processus lancés par le système.

Enfin, nous avons la sous-figure 37d qui illustre l'état de la liste après l'exécution de la fonction *void supprimer_queue(liste * &, TypeElt)*.

3.2.2 *Exemples d'applications sur les listes chaînées*

Soit une liste simplement chaînée d'entiers et dont la déclarons en C++ est comme suit :

```
struct liste
{
        int elt;
        liste * suivant;
}
liste * tete = NULL;
```

Calcul de la longueur de la liste chaînée

Calculer le nombre d'entiers chargés dans la liste revient à compter le nombre de maillons qui composent la liste. Ce qui donne lieu à la fonction *int calcul_longueur(liste * tete)* :

```
int calcul_longueur(liste * tete)
{
  int n=0; liste * p = tete;
  while(p!=NULL)
  {
    n++;
    p = p->suivant;
   }
 return n;
}
```

La boucle *while* permet de boucler sur les maillons à travers le champ *suivant*, jusqu'à atteindre la valeur *NULL* ; condition d'arrêt qui marque la fin de la liste.

Saisie des n éléments de la liste

La fonction suivante *void saisie_element(liste *&)* permet l'insertion des éléments à l'entête de la liste.

```
void saisie_element(liste *& tete)
{
  liste * p;
  TypeElt val;
  for(int i=0; i<n; i++)
  {
        lire(val); //saisir les valeurs du maillon
        ajout_tete(tete, val);
  }
}
```

3.2.3 *Listes doublement chaînées*

Une liste doublement chaînée est une liste qu'on peut parcourir dans les deux sens ; du premier élément au dernier et inversement. Ce qui impose l'ajout d'un autre champ à la structure du maillon qu'on appellera *precedent* et qui doit contenir l'adresse de son prédécesseur. La structure d'un maillon d'une liste doublement chaînée est illustrée dans la figure 38.

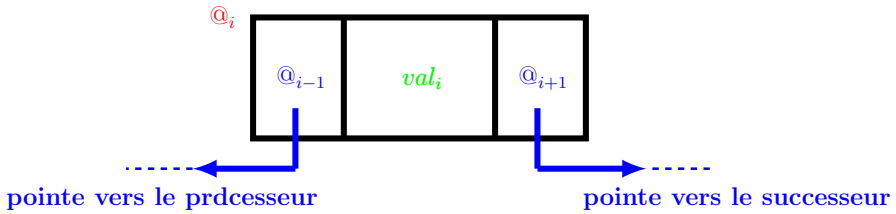


FIGURE 38 – Structure d'un maillon d'une liste doublement chaînée

Définition de la liste en C++

En suivant la structure du maillon illustrée dans la figure 38, la déclaration d'une liste doublement chaînée se donne en C++ comme suit :

```
struct listd
{
        TypeElt elt;
        listd * suivant;
        listd * precedent;
}
listd * tete = NULL;
```

Dans le cas où la liste contient n éléments, nous obtenons une liste qui a la structure que nous exposons via la figure ci-dessous 39.

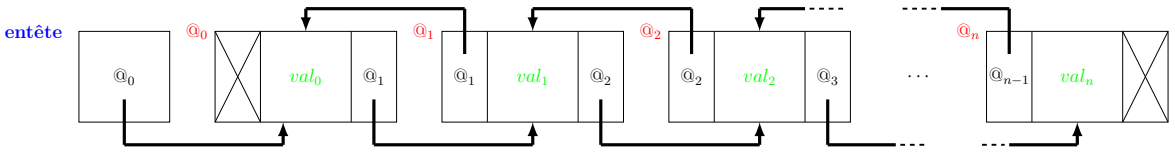


FIGURE 39 – Structure d'une liste doublement chaînée

Création d'un maillon

La fonction *liste * creer_maillon(TypeElt val)* alloue dynamiquement la mémoire nécessaire pour le nouveau maillon à travers l'instruction *listd * maillon = new listd*. Un test est réalisé afin de vérifier si l'allocation a été effectuée, afin de charger la valeur *val* qui doit être de type *TypeElt*. Les champs *precedent* et *suivant* sont initialisés à *NULL*.

La dernière instruction *return maillon* retourne l'adresse du maillon créé. Cette adresse va nous servir ensuite pour réaliser différents traitements sur ce maillon.

```
liste * creer_maillon(TypeElt val)
{
  listd * maillon = new listd;
  if (maillon!=NULL)
  {
    maillon->elt = val;
    maillon->suivant = NULL;
    maillon->precedent = NULL;
  }
  return maillon;
}
```

Ajout d'un élément

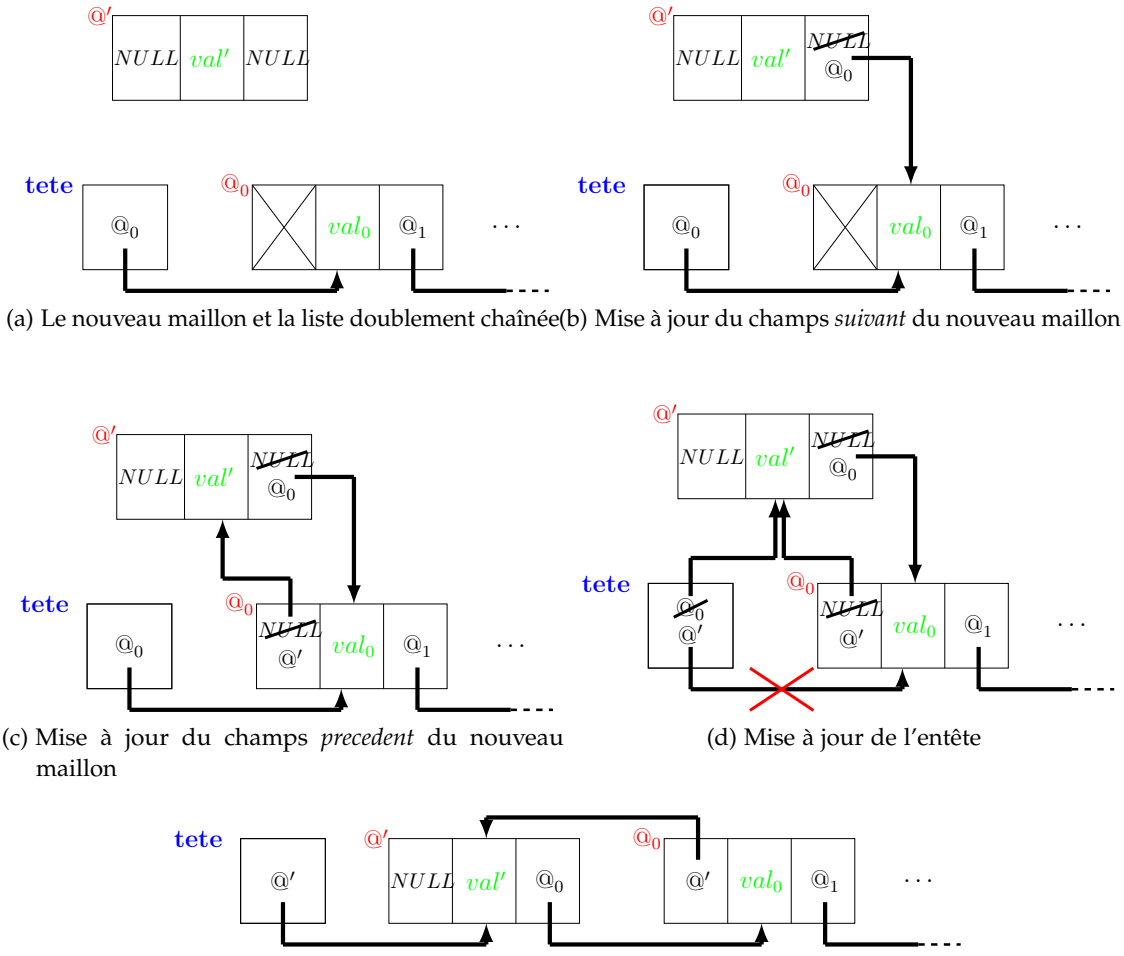
L'insertion d'un maillon dans une liste doublement chaînée nécessite plus d'étapes qu'une liste simple, puisque nous avons un nouveau champ à prendre en considération *precedent*. Ces étapes se donnent comme suit :

1. Il faut tout d'abord créer le nouvel élément et saisir ou stocker les valeurs.
2. Insérer cet élément dans la liste. La cohérence de la liste doit finalement être rétablie en indiquant que le nouvel élément est désormais le successeur de celui après lequel il va prendre place et le prédécesseur de celui avant lequel il doit être inséré.

A. Insertion en tête de liste : L'insertion nécessite la mise à jour du champ suivant du nouveau maillon, du champ précédent du premier élément de la liste et de *tete*. La fonction ci-dessous *void ajout_tete(listd * & tete, TypeElt val)* permet la création du nouveau maillon et son insertion à l'entête de la liste :

```
void ajout_tete(listd *& tete, TypeElt val)
{
  liste *maillon=ceer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = tete;
    maillon->precedent = NULL;
    if (tete) tete->precedent =  maillon;
    tete = maillon;
  }
}
```

Nous résumons les instructions d'insertion via la figure 40.



(a) Le nouveau maillon et la liste doublement chaînée(b) Mise à jour du champs *suivant* du nouveau maillon

(c) Mise à jour du champs *precedent* du nouveau maillon

(d) Mise à jour de l'entête

(e) La liste chaînée après insertion

FIGURE 40 – Insertion d'un maillon à l'entête de la liste

La sous-figure 40a illustre l'état de la liste doublement chaînée avec le nouveau maillon créé. L'adresse du premier maillon est $@_0$ et celle du nouveau maillon est $@'$.

La mise à jour du champ *suivant* du nouveau maillon se donne via la sous-figure 40b, résultat de l'instruction *maillon->suivant = tete*.

Après avoir mis le nouveau maillon en première position de la liste, il doit jouer le rôle de prédécesseur de l'élément d'adresse $@_0$. Cette étape est illustrée dans la figure 42c qui montre l'impact de l'instruction *tete->precedent=maillon*. Le champ *precedent* de l'élément d'adresse $@_0$ contient la valeur $@'$.

La dernière instruction exécutée dans la fonction est *tete=maillon* ; telle illustré dans la figure 40d. La variable *tete* est mise à jour et elle contient l'adresse $@'$.

L'état final de la liste après insertion se donne via la figure 40e.

B. Insertion au milieu de la liste : Idem que l'insertion au milieu d'une liste simple, en entrée de la fonction nous avons l'adresse du prédécesseur. Les différentes étapes d'insertion au milieu d'une liste doublement chaînée se donne via la fonction d'en tête *void ajout_milieu(listd * pred, TypeElt val)*.

```
void ajout_milieu(listd * pred, TypeElt val)
{
  listd *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->suivant = pred->suivant;
    maillon->precedent = pred;
    pred->suivant->precedent = maillon;
    pred->suivant = maillon;
  }
}
```

La figure 41 décrit en détails l'application de la fonction *ajout_milieu* qui insert un nouveau maillon d'adresse @′ au milieu de la liste.

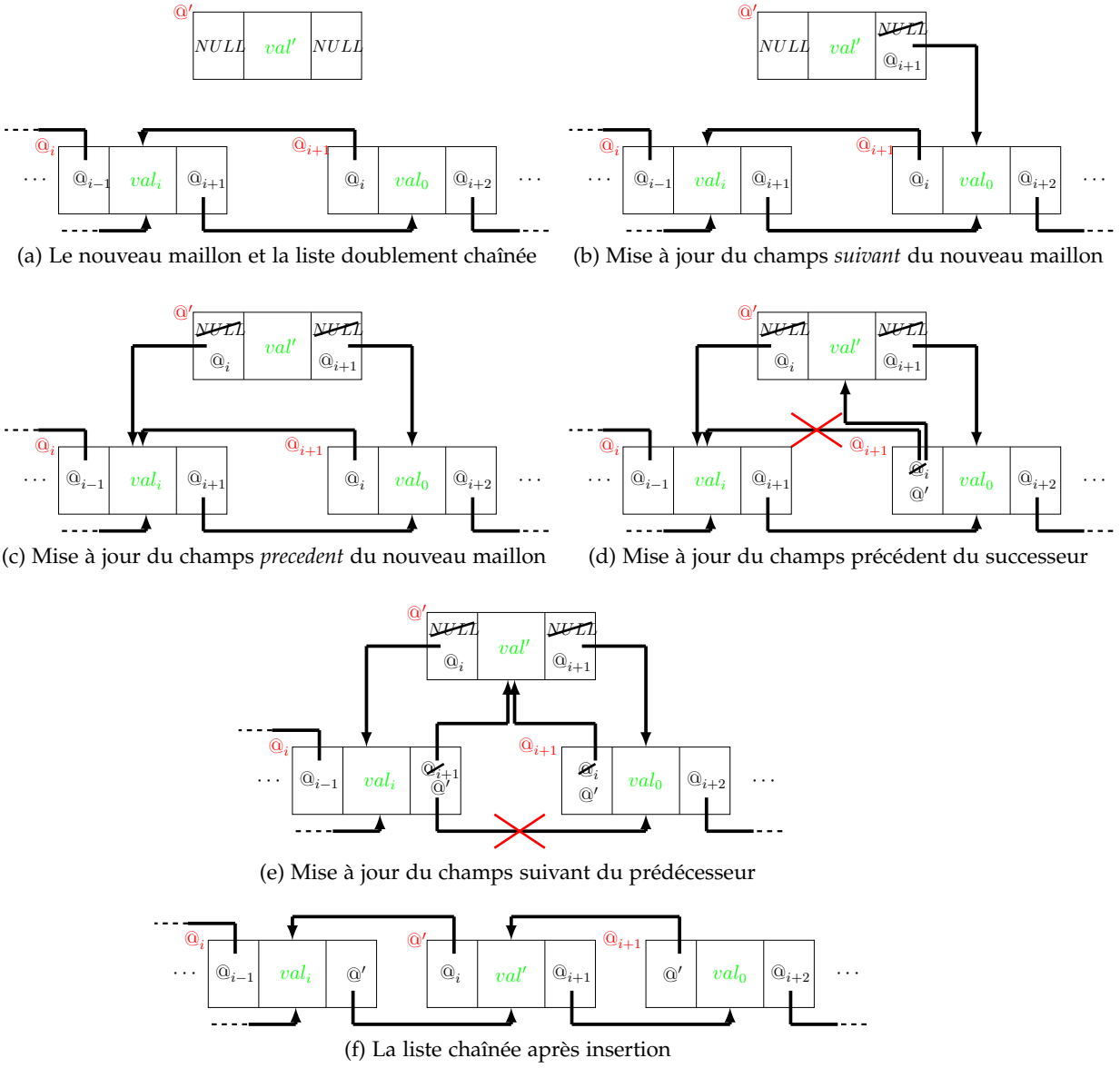


FIGURE 41 – Insertion d'un maillon au milieu de la liste

La sous-figure 41a illustre l'état de la liste doublement chaînée et le nouveau maillon créé d'adresse @′. Le maillon doit être inséré entre les éléments d'adresses respectifs $@_i$ et $@_{i+1}$. Le paramètre *pred* est donc de valeur $@_i$.

La première étape consiste à mettre à jour les champs *precedent* et *suivant* du nouveau maillon. Cette mise à jour est réalisée à travers les deux instructions *maillon->suivant = pred->suivant* et *maillon->precedent = pred*. L'impact de ces deux instructions est illustrée dans les deux sous-figures respectives 41b et 41c. Nous obtenons les champs pointeurs du nouveau maillon *precedent* $= @_i$ et *suivant* $= @_{i+}$.

La sous-figure 41d montre le résultat de l'instruction *pred->suivant->precedent=maillon* ; le nouveau maillon doit être le prédécesseur de l'élément d'adresse $@_{i+1}$. Par conséquent, le champ *precedent* de l'élément d'adresse $@_{i+1}$ contient la valeur @′.

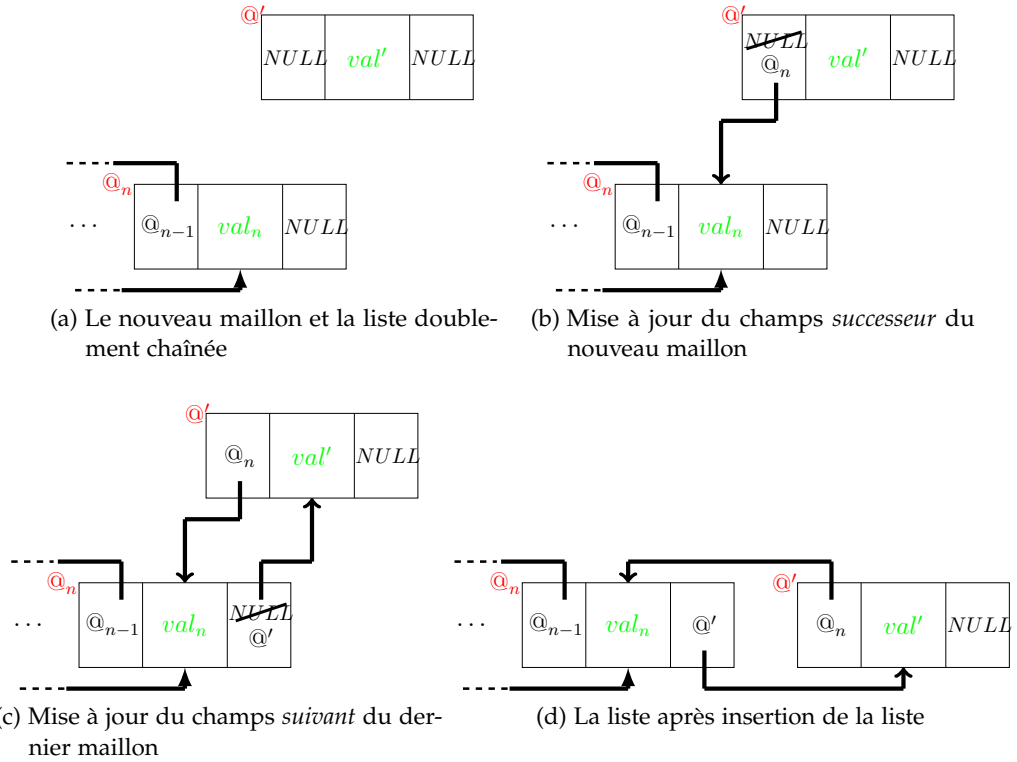
La dernière instruction exécutée dans la fonction est *pred->suivant = maillon ;* nous affectons au champs *suivant* du $i^{ème}$ élément la valeur @′ (l'adresse du nouveau maillon) (voir la figure 41e).

L'état final de la liste se donne via la sous-figure 41f.

C. Insertion en queue de la liste : La fonction *void ajout_queue(listd * pred, TypeElt val)* donnée ci-dessous permet l'insertion du nouveau maillon en queue de la liste, avec *p* pointeur sur l'avant dernier élément.

```
void ajout_queue(listd * pred, TypeElt val)
{
  listd *maillon=creer_maillon(val);
  if (maillon != NULL)
  {
    maillon->precedent = pred;
    maillon->suivant = NULL;
    pred->suivant = maillon;
  }
}
```

La figure 42 représente les différentes étapes d'insertion d'un maillon en queue d'une liste doublement chaînée et qui contient $n + 1$ éléments.



(a) Le nouveau maillon et la liste doublement chaînée

(b) Mise à jour du champs *successeur* du nouveau maillon

(c) Mise à jour du champs *suivant* du dernier maillon

(d) La liste après insertion de la liste

FIGURE 42 – Insertion d'un maillon en tête de liste

La figure 42a représente la fin d'une liste doublement chaînée dont le dernier élément est d'adresse $@_n$ et le nouveau maillon d'adresse @′.

La première étape consiste à mettre à jour le champs *precedent* du nouveau maillon avec l'adresse $@_n$, et ce à travers l'instruction *maillon->precedent = pred* où $pred = @_n$ (voir la figure 42b). Le champ suivant est *NULL* puisque le nouveau maillon sera le dernier et ne doit pointer sur aucun successeur.

La figure 42c illustre l'impact de l'exécution de l'instruction *pred->suivant=maillon* sur la liste ; le $n^{'eme}$ élément a comme successeur le nouveau maillon.

Le résultat final de la fonction *ajout_queue* est illustré dans la figure 42d.

Suppression d'un élément d'une liste chaînée

La logique des opérations qui permettent la suppression d'un élément d'une liste doublement chaînée est assez simple :

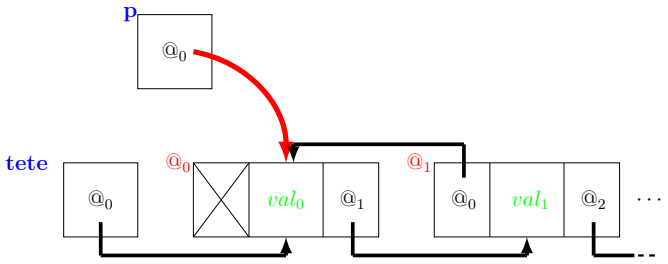
1. L'élément précédant celui qu'on doit supprimer doit adopter comme successeur le successeur de ce dernier.
2. L'élément succédant celui qu'on doit supprimer doit adopter comme prédécesseur le prédécesseur de ce dernier.
3. L'espace mémoire déjà réservé à l'élément supprimé doit être libéré.

Effectivement, après la mise à jour des pointeurs *precedent* et *suivant*, nous devons libérer l'espace mémoire occupé par le maillon supprimé de la liste. Ce qui nécessite le chargement de l'adresse du maillon dans une variable temporaire noté *p*.

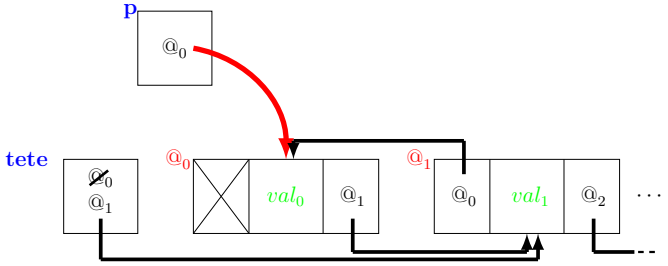
A. Suppression à l'entête de la liste : La suppression à l'entête de la liste désigne la suppression du premier élément de la liste. Elle est décrite via la fonction *void supprimer_tete(listd * & tete)*. L'adresse de ce dernier est chargé dans la variable pointeur *p*. Par la suite, l'entête et le champ *precedent* du deuxième élément sont mis à jour.

```
void supprimer_tete(listd * & tete)
{
  if (tete != NULL)
  {
    liste * p = tete;
    tete = p->suivant;
    p->precedent = NULL;
    delete p;
  }
}
```

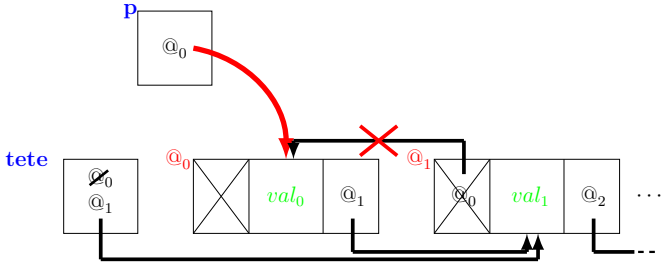
La figure 43 décrit en détail les différentes étapes de suppression du maillon en première position d'une liste doublement chaînée.



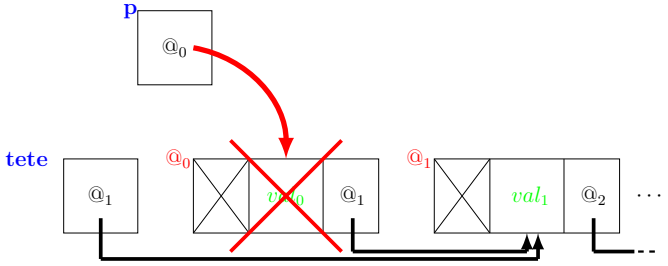
(a) L'entête de la liste doublement chaînée



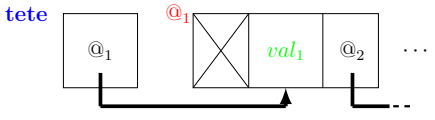
(b) Mise à jour de la variable *entete*



(c) Mise à jour du champ *precedent* du maillon succédant celui à supprimer



(d) Libération de l'espace mémoire occupé par le maillon supprimé



(e) La liste chaînée après suppression

FIGURE 43 – Suppression d'un maillon à l'entête de la liste

La première sous-figure 43a illustre l'entête d'une liste doublement chaînée. Nous avons la variable entête $tete = @_0$ et la variable temporaire $p = @_0$ qui pointe sur l'élément à supprimer.

La sous-figure 43b illustre le changement de la valeur de *tete* qui pointe sur le deuxième élément d'adresse $@_1$. Elle met en avant l'impact de l'instruction *tete = p->suivant*.

L'instruction qui suit est *p->precedent=NULL*, son résultat est illustré dans la figure 43c. L'élément d'adresse $@_1$ n'a plus de prédécesseur ; le champ *precedent* est mis à *NULL*.

Après la mise à jour des chaînages, il faut libérer l'espace mémoire occupé par le maillon supprimé et dont l'adresse est dans p. Cette action est réalisée par l'instruction *delete p* et illustrée dans la figure 43d.

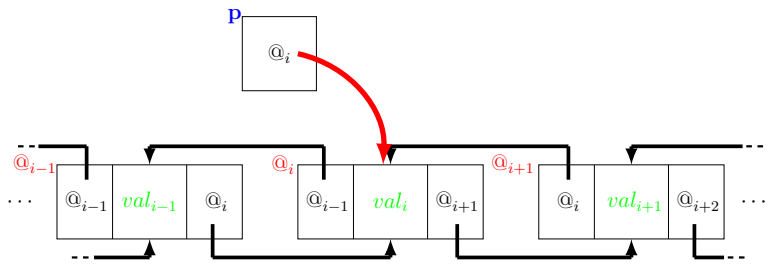
L'état final de la liste après suppression est donné dans la figure 43e.

B. Suppression au milieu de la liste : La fonction ci-dessous *void supprimer_milieu(listd * p)* réalise la suppression d'un maillon au milieu d'une liste doublement chaînée et dont l'adresse est représentée par le paramètre d'entrée p.

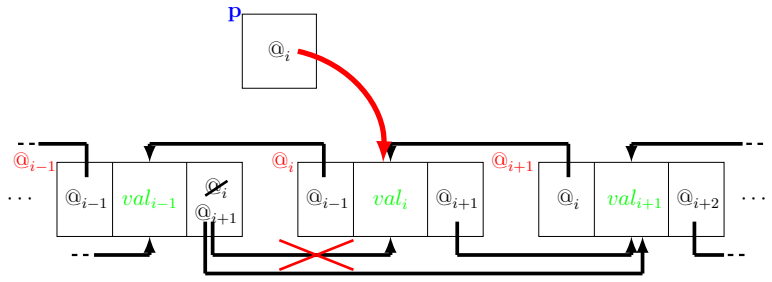
```
void supprimer_milieu(listd * p)
{
    p->precedent->suivant = p->suivant;
    p->suivant->precedent = p->precedent;
    delete p;
}
```

L'exécution de cette fonction est illustrée dans la figure 44.

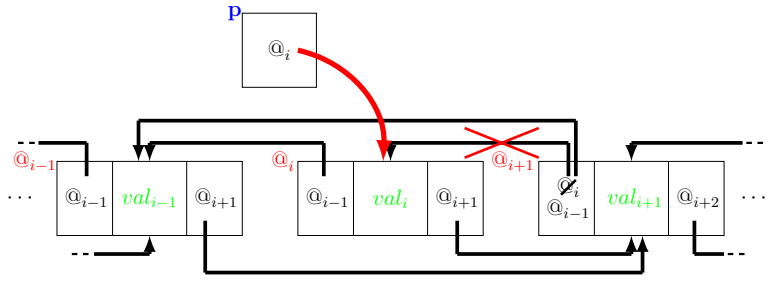
L'élément à supprimer est d'adresse $@_i$ entre un élément prédécesseur et un élément successeur d'adresses respectives $@_{i-1}$ et $@_{i+1}$. La variable p contient la valeur $@_i$; l'adresse du maillon à supprimer.



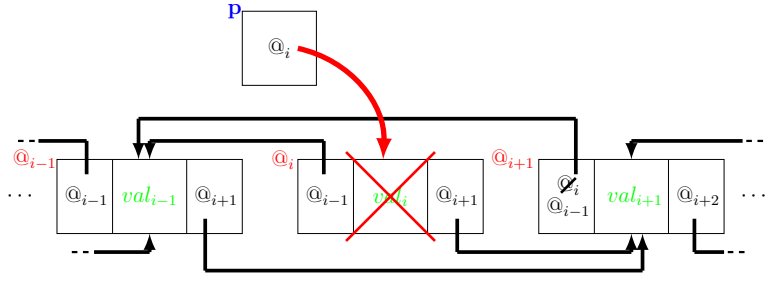
(a) La liste doublement chaînée



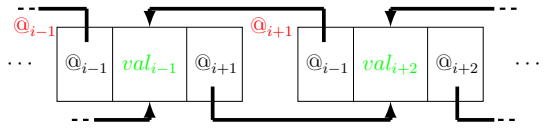
(b) Mise à jour du champ *suivant* du maillon précédant celui à supprimer



(c) Mise à jour du champ *precedent* du maillon succédant celui à supprimer



(d) Libération de la mémoire du maillon supprimé



(e) La liste chaînée après insertion

FIGURE 44 – Suppression d'un maillon au milieu d'une liste doublement chaînée

La sous-figure 44a montre la liste doublement chaînée et la variable p qui pointe sur le maillon à supprimer d'adresse.

La sous-figure 44b représente l'exécution de l'instruction *p->precedent->suivant = p->suivant* et qui permet d'affecter la valeur $@_{i+1}$ au champ *suivant* du maillon d'adresse $@_{i-1}$.

L'instruction suivante *p->suivant->precedent = p->precedent* assigne le champ *precedent* de l'élément d'adresse $@_{i-1}$ avec la valeur $@_{i+1}$. Elle est illustrée dans la figure 44c.

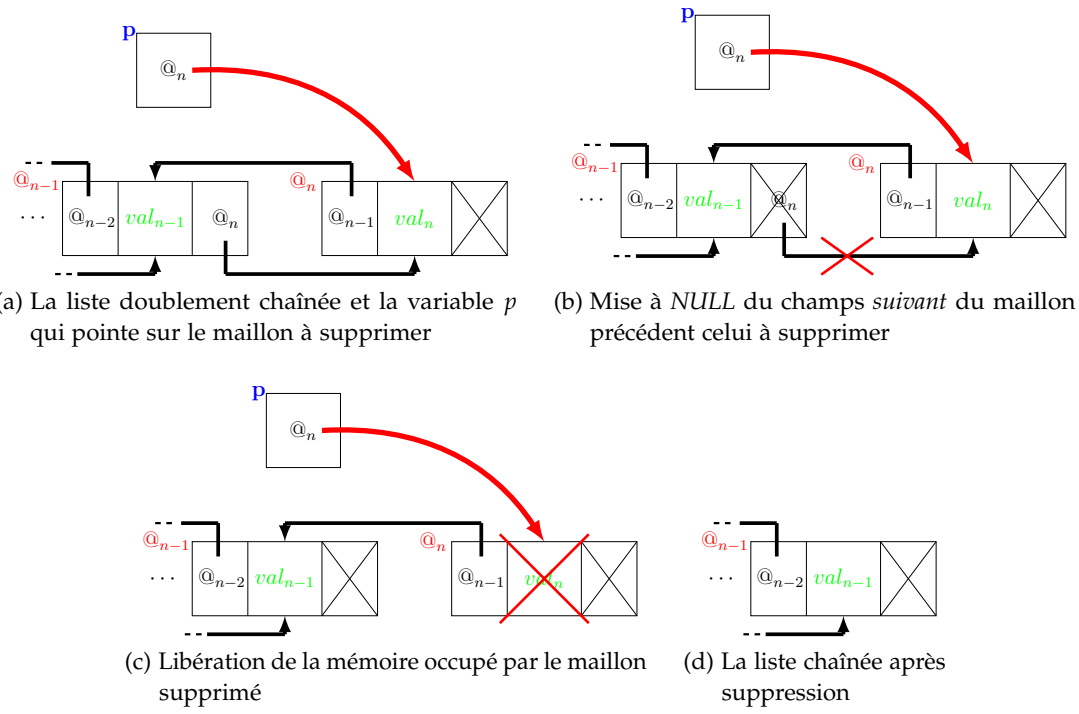
La dernière étape consiste à libérer l'espace mémoire occupé par le maillon supprimé de la liste, et ce en exécutant l'instruction *delete p*. Elle est illustrée dans la figure 44d.

La dernière sous-figure 44e représente l'état final de la liste après exécution de la fonction.

C. Suppression en queue de la liste : La fonction ci-dessous *void supprimer_en_queue(listd * p)* réalise la suppression du dernier maillon de la liste doublement chaînée et dont l'adresse est l'entrée *p* de la fonction.

```
void supprimer_en_queue(listd * p)
{
    p->precedent->suivant = NULL;
    delete p;
}
```

L'exécution de cette fonction est illustrée dans la figure 45. L'élément à supprimer est d'adresse $@_n$. Elle est contenue dans la variable pointeur *p*.



(a) La liste doublement chaînée et la variable *p* qui pointe sur le maillon à supprimer

(b) Mise à *NULL* du champs *suivant* du maillon précédent celui à supprimer

(c) Libération de la mémoire occupé par le maillon supprimé

(d) La liste chaînée après suppression

FIGURE 45 – Suppression du maillon en queue d'une liste doublement chaînée

La sous-figure 45a illustre la fin de la liste doublement chaînée et la variable p qui pointe sur le maillon à supprimer.

La sous-figure 45b représente l'exécution de l'instruction *p->precedent->suivant = NULL*. Elle permet d'affecter la valeur *NULL* au champ *suivant* du maillon d'adresse $@_{n-1}$.

La dernière étape consiste à libérer l'espace mémoire occupé par le maillon supprimé de la liste, en exécutant l'instruction *delete p*. Elle est illustrée dans la figure 45c.

La dernière sous-figure 45d représente l'état final de la liste.

3.2.4 *Piles*

Une pile est une structure de données où les insertions et les suppressions se font toutes du même coté. Une telle structure est aussi appelée LIFO (Last In First Out).

Représentation d'une pile en C++

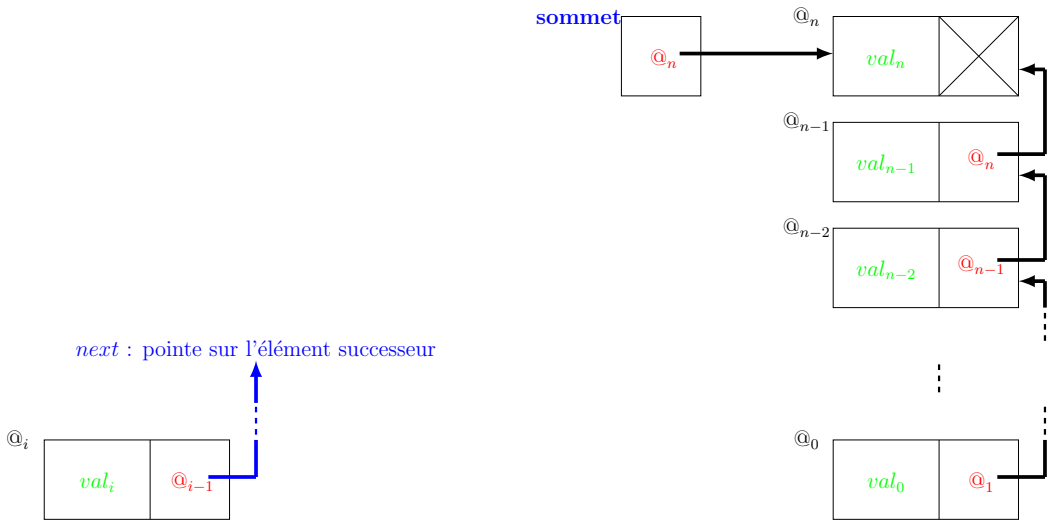
Nous exposons dans cette section la structure d'une pile basée sur les listes simplement chaînées. Effectivement, une pile est déclarée sous forme d'une liste simplement chaînée avec l'élément sentinelle *sommet*.

Les éléments de la pile sont chaînés entre eux, et le sommet d'une pile non vide est le premier de la liste ; dans le cas où la pile est vide *sommet* est *NULL*.

La définition de la pile en C++ se donne comme suit :

```
struct pile
{
        TypeElt elt;
        pile * next;
};
pile * sommet=NULL;
```

La figure ci-dessous 46 illustre la structure d'une pile.



(a) Elément d'une pile (b) La structure d'une pile

FIGURE 46 – Structure d'une pile en utilisant une liste simplement chaînée

La figure 46a décrit la structure d'un élément d'une pile qui a la structure d'une liste simplement chaînée, donnée à son tour dans la figure 46b. L'élément sentinelle est *sommet* qui pointe sur le sommet de la pile. Les éléments ont des adresses numérotées de 0 à n, sachant que l'adresse $@_0$ correspond à la première insertion, l'adresse $@_1$ à la deuxième insertion, etc. La variable *sommet* pointe sur l'élément d'adresse $@_n$; adresse du dernier élément inséré. Sachant que, l'insertion n'est autorisée qu'au sommet de la pile.

Nous précisons qu'aucun accès n'est autorisé aux autres éléments de la pile hors le sommet.

Les piles admettent deux primitives, à savoir *empiler* et *dpiler*, qui permettent respectivement l'empilement et le dépilement d'un élément. Bien sûr, les deux primitives ne sont appliquées qu'au sommet de la pile [1].

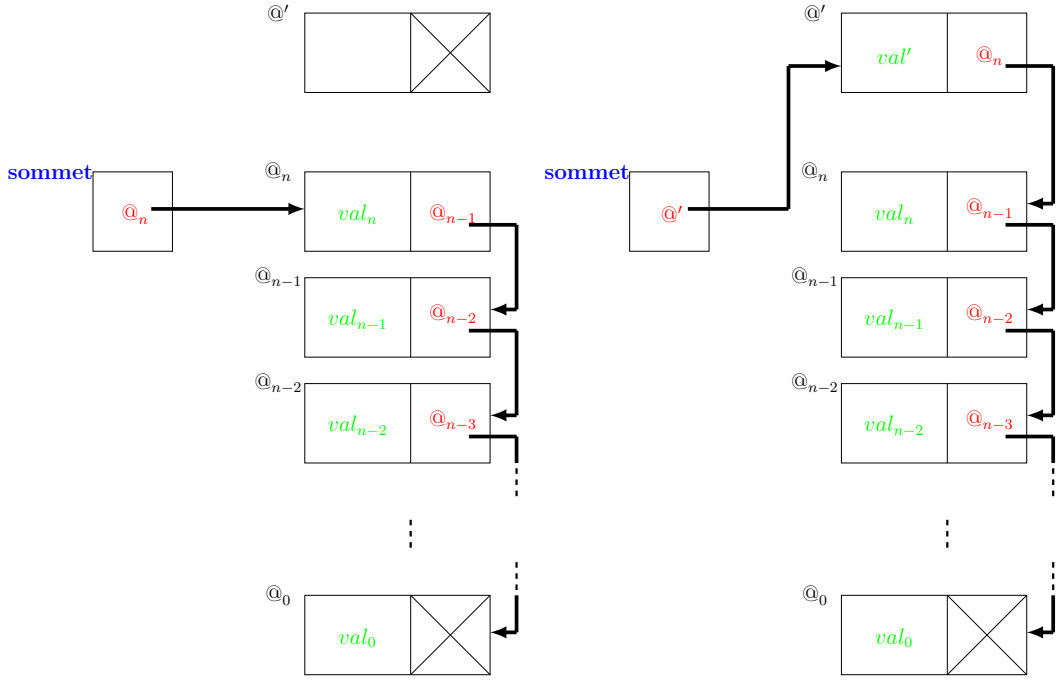
Empiler un élément

Empiler un nouvel élément dans la liste revient à insérer un nouveau maillon à l'entête de la liste. La fonction d'empilement *void empiler(pile * & sommet, TypeElt)* est similaire à celle de l'insertion à l'entête d'une liste simplement chaînée décrite dans section 3.2.2.

1. Le sommet de la pile correspond à l'entête de la liste simplement chaînée

```
void empiler(pile *& sommet, TypeElt val)
{
  pile *E = new pile;
  if (E)
  {
    E->elt = val;
    E->next = sommet;
    sommet = E;
  }
}
```

La figure ci-dessous 47 illustre l'exécution de la fonction *empiler*



(a) La pile avant l'epilement de l'élément d'adresse @'

(b) La pile après empilement

FIGURE 47 – Insertion d'un élément dans une pile

Nous avons dans la figure 47a le maillon à empiler d'adresse @' et la pile dont le sommet est d'adresse $@_n$.

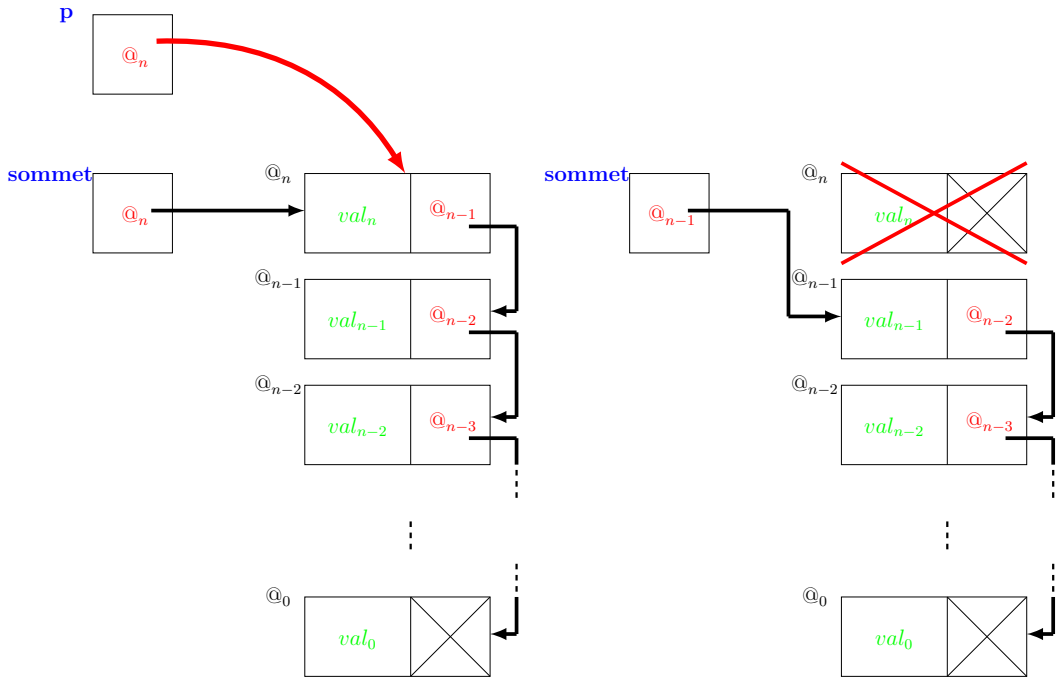
Après exécution de la fonction, situation illustrée dans la sous-figure 47b, l'élément sentinelle *sommet* est mis à jour. L'élément au sommet de la pile est le nouveau qui pointe sur l'élément d'adresse $@_n$

Dépiler un élément

Dépiler un élément de la pile revient à supprimer l'élément au sommet de la pile. La fonction correspondant *void depiler(pile *& sommet)* est similaire à la fonction de suppression d'un maillon au sommet d'une liste simplement chaînée (voir la section 3.2.2).

```
void depiler(pile *& sommet)
{
  if (sommet == NULL) return;
  else
  {
    pile *p = sommet;
    sommet = sommet->lien;
    delete p;
  }
}
```

La figure 48 illustre l'exécution de la fonction *void depiler(pile *& sommet)*.



(a) La pile avant l'empilement (b) La pile après le dépilement

FIGURE 48 – Suppression d'un élément dans une pile

La sous-figure 51a montre la mise à jour de l'élément sentinelle *sommet* à $@_{n-1}$. Nous avons par la suite, la libération de la mémoire occupée par l'élément supprimé.

L'état final de la pile se donne dans la figure 48bavec $sommet = @_{n-1}$.

Valeur du sommet de la pile

Nous avons ci-dessous la fonction *TypeElt depiler(pile * sommet)* qui retourne la valeur chargée au sommet de la pile.

Si on considère la figure 46b, la fonction retourne la valeur val_n. Dans le cas où la pile est vide elle retourne la valeur *val_pile_vide*, une valeur fixé par le programmeur et qui exprime l'état « vide »de la pile.

```
TypeElt depiler(pile * sommet)
{
  if (sommet == NULL) return val_pile_vide;
  else return sommet->elt;
}
```

5.2.5 *Files*

Dans le cas d'une file, on fait les adjonctions (insertions) à une extrémité, les accès et les suppressions de l'autre extrémité. Les files sont aussi appelées FIFO (First In First Out). En d'autres termes, l'accès à un élément de la file suit le raisonnement d'une file d'attente « premier venu premier servi »

Représentation d'une file en C++

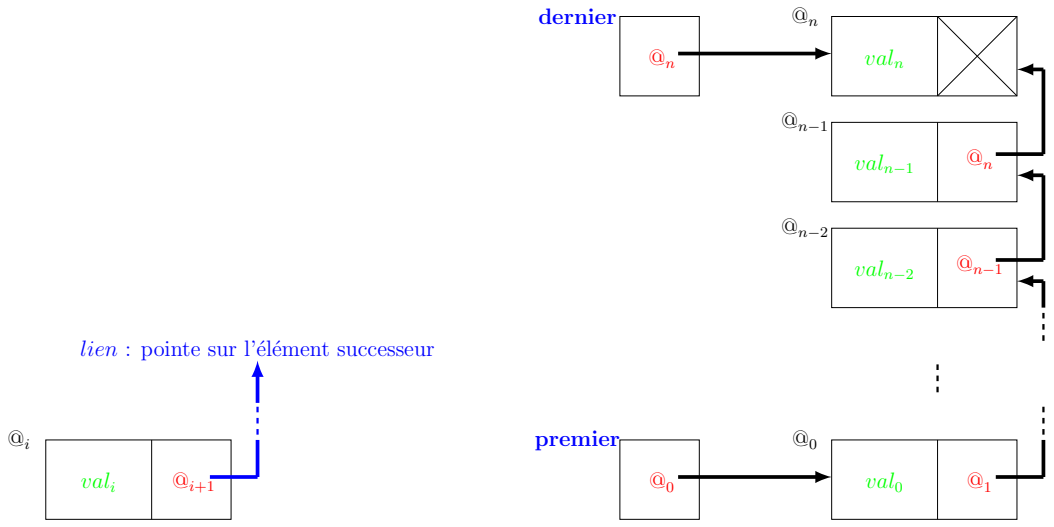
Dans notre cours, la structure d'une file repose sur les listes chaînées simples. Une file possède deux éléments sentinelle *premier* et *dernier* :

premier : pointe sur le premier élément de la file.

dernier : pointe sur le dernier élément de la file.

```
struct file
{
        TypeElt elt;
        file * lien;
};
file * premier=NULL;
file * dernier=NULL;
```

La figure 49 montre la structure d'un composant de la file et la structure d'une file avec un chaînage simple.



(a) Elément d'une file (b) La structure d'une file

FIGURE 49 – Structure d'une file en utilisant une liste simplement chaînée

La sous-figure 49a est une illustration d'un élément de la file ; il contient un champ *valeur* et un autre adresse du successeur *lien*.

La structure de la file est donnée dans la sous-figure 49b. Les éléments sont enfilés de $@_0$ à $@_n$, raison pour laquelle *premier* pointe sur l'élément d'adresse $@_0$ et *dernier* sur l'élément d'adresse $@_n$.

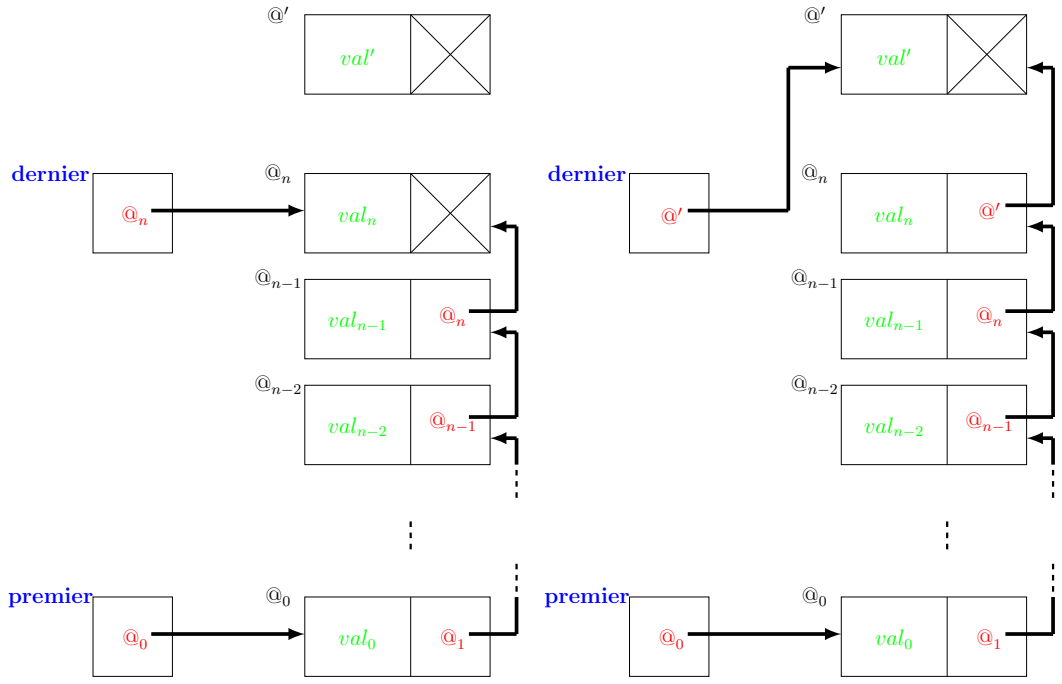
Enfiler un élément dans la file

L'insertion d'un élément dans une file (ou l'enfilement) se fait en queue de la file. Par conséquent la variable *dernier* doit être mise à jour avec l'adresse du nouvel élément. La primitive *enfiler* se donne comme suit :

```
void enfiler(file *& dernier, file *& premier, TypeElt val)
{
  file * E = new file;
  if (E)
  {
    E->elt = val;
    E->lien = NULL;
    if (dernier==NULL)
    {
                dernier = E;
                premier=dernier;
        }
    else dernier->lien = E;
  }
}
```

Nous constatons que la fonction admet le paramètre *premier* par référence en plus du paramètre *dernier*, et ce afin de traiter le cas où la file est vide $premier = dernier = NULL$.

La figure 50 illustre l'exécution de la fonction sur une file non-vide $premier \neq NULL$ et $dernier \neq NULL$.



(a) La file avant l'insertion de l'élément d'adresse @'

(b) La file après insertion

FIGURE 50 – Insertion d'un élément dans une file

Nous avons dans la sous-figure 50a l'élément à enfiler d'adresse @' et la file qui contient $n+1$ élément dont les adresses varient de $@_0$ à $@_n$.

La sous-figure 50b illustre l'application de *enfiler*. La variable *dernier* est mise à jour, ainsi que le champ *lien* du maillon d'adresse $@_n$. Par conséquent, *dernier* pointe sur @', successeur de l'élément d'adresse $@_n$.

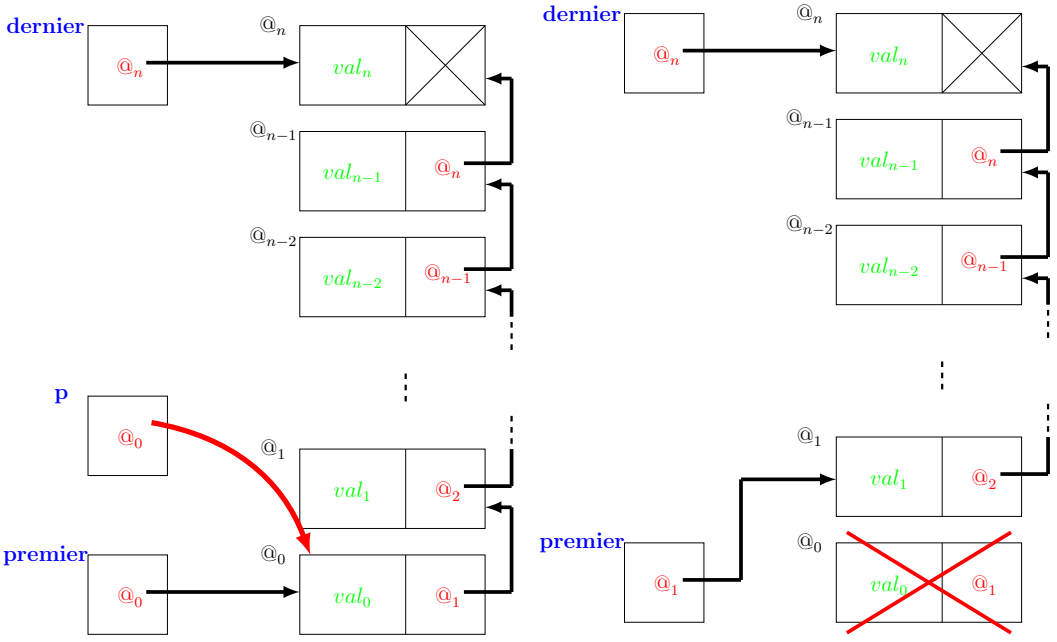
Défiler un élément de la file

La suppression d'un élément d'une file (ou le défilement) se fait en tête de la liste comme suit :

```
void defiler(file *& premier,file *& dernier)
{
        file *p = premier;
    if (premier)
    {
        premier = premier->lien;
        delete p;
                if (!dernier) dernier=NULL;
        }
}
```

La fonction admet le paramètre *dernier* par référence en plus du paramètre *premier*, et ce afin de traiter le cas où la file ne contient qu'un seul élément $premier = dernier \neq NULL$.

La figure 51 illustre l'action *défiler* appliquée sur une file qui contient plus d'un élément.



(a) La file avant suppression (b) La file après suppression

FIGURE 51 – Suppression d'un élément dans une file

Nous considérons dans la figure 51a une file qui contient $n+1$ éléments.

Puisque nous ne pouvons supprimer que le premier élément de la file, celui d'adresse $@_0$ est l'élément qui sera éliminé de la file (l'élément d'adresse $@_0$ est le premier enfilé dans la file). La sous-figure 50b montre le résultat de l'application de la primitive sur la file, avec $premier = @_1$ et l'espace mémoire occupé par l'élément supprimé est libéré.

3.3 STRUCTURES ARBORESCENTES

3.3.1 *Arbres*

Un arbre est un ensemble de nœuds organisés de façon hiérarchique, à partir d'un nœud distingué appelé racine.

Une propriété intrinsèque de la structure d'arbre est la récursivité et les définitions caractéristiques des arbres s'écrivent très naturellement de manière récursive.

Arbres binaires

Soit la définition 3.3.1 ci-dessous :

Définition 3.3.1 *[Froidevaux et al., 1993] Un arbre binaire est soit vide (noté ∅), soit de la forme $B = \langle \circ, B_1, B_2 \rangle$, où B_1 et B_2 sont des arbres disjoints et $\circ$ est un nœud appelé racine.*

Cette définition est récursive ; elle peut s'écrire sous la forme de l'équation : $B = \varnothing + \langle \circ, B, B \rangle$, tel que B représente l'ensemble des arbres binaires, $\varnothing$ représente l'arbre vide et $\circ$ désigne un nœud.

1. Représentation d'un arbre binaire en C++

Un arbre est une structure arborescente avec un élément sentinelle appelé *racine*. Dans cette section nous introduisons les arbres binaires où chaque nœud ne peut avoir plus de deux fils : fils gauche et fils droit.

Par exemple, la figure 52 prise de[Froidevaux et al., 1993] représente un arbre binaire qui modélise l'expression arithmétique $(x - (2 * y)) + (x + (y/z) * 3)$.

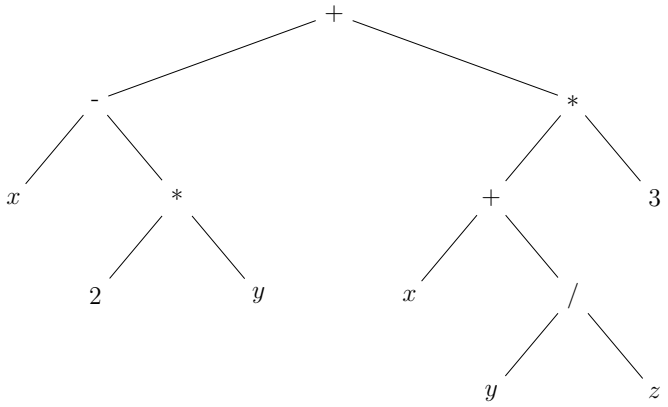


FIGURE 52 – Expression arithmétique $(x - (2 * y)) + (x + (y/z) * 3)$

Nous remarquons que la racine correspond à l'opérateur **+**, dont le fils gauche et le fils droit donnent lieu respectivement à :

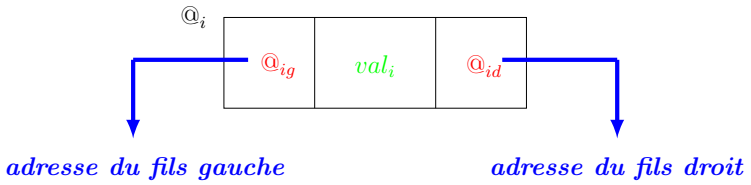
— un sous-arbre gauche qui modélise l'expression $x - (2 * y)$,

— et un sous-arbre droit qui modélise l'expression $(x + (y/z) * 3$.

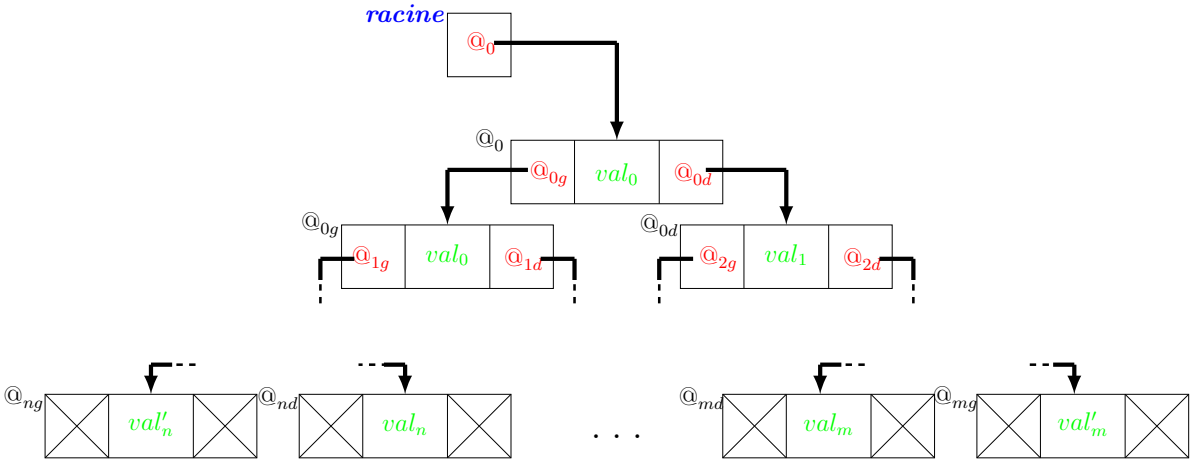
La déclaration en C++ d'un arbre binaire se donne comme suit :

```
struct noeud
{
        TypeElt elt;
        noeud * gauche;
        noeud * droit;
}
noeud * racine=NULL;
```

La figure 53 donne la structure d'un arbre binaire et de ses nœuds.



(a) Nœud d'un arbre binaire



(b) Structure d'un arbre binaire avec chaînage

FIGURE 53 – Structure d'un arbre binaire

La sous-figure 53a illustre la composition d'un nœud ; il comporte un champ pointeur qui pointe sur le fils gauche, la valeur à charger et un autre champ pointeur sur le fils droit.

Nous montrons via la figure 53b la représentation en mémoire d'un arbre binaire. L'élément *racine* est d'adresse $@_0$; il a un fils gauche d'adresse $@_0g$ et un fils droit $@_0d$. Ces derniers, peuvent à leur tour avoir des fils gauche et droit. Les nœuds ayant les adresses $@_{ng}$, $@_{nd}$, $@_{mg}$ et $@_{md}$ n'ont pas d'enfants (ni fils gauche, ni fils droit) sont appelés *feuilles*.

Hors le nœud *racine* et les nœuds *feuille*, le nœud est dit *interne*.

Pour une meilleur compréhension de la structure nous introduisons ci-dessous un exemple 5.3.3 qui introduit un arbre binaire d'entiers et sa représentation en mémoire.

Exemple 3.3.2 (arbre binaire d'entiers) *Soit l'arbre binaire d'entier illustré dans la figure 54 ci-dessous*

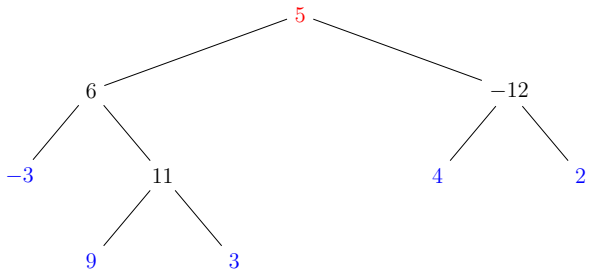


FIGURE 54 – Arbre binaire d'entiers

La représentation en mémoire de l'arbre est représenté par la figure 55 ci-dessous :

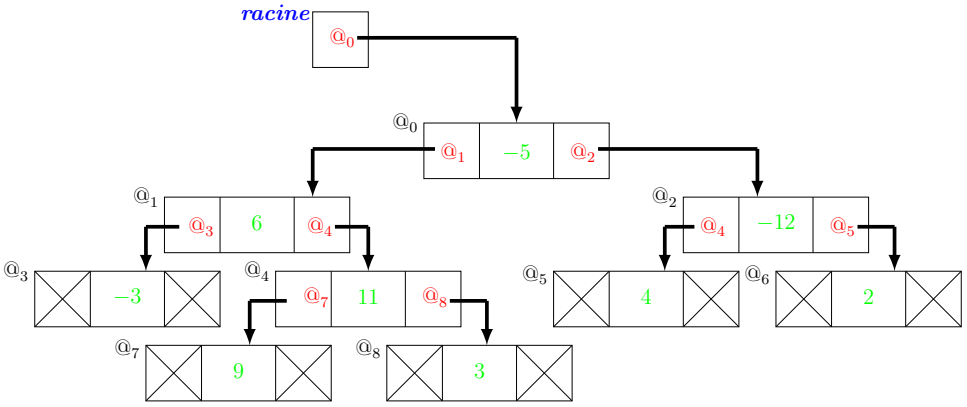


FIGURE 55 – Représentation en mémoire de l'arbre de la figure 54

2. *Vérifier si l'arbre est vide*

Un arbre binaire vide est marqué par la variable *racine* qui doit être de valeur $NULL$.

La primitive *est_vide* retourne *true* si l'arbre est vide et $false$ sinon.

```
bool est_vide(noeud * racine)
{
  return (racine==NULL);
}
```

Récupérer l'un des deux fils

Afin de récupérer l'adresse du fils gauche ou celle du fils droit d'un nœud donné nous avons les deux primitives *gauche* et *droit*.

Le fils gauche : La fonction *noeud * gauche(noeud * R)* retourne l'adresse du fils gauche du nœud d'adresse *R*.

```
noeud * gauche(noeud * R)
{
  if (est_vide(R)) return NULL;
  else return R->gauche;
}
```

Évidemment, il faut vérifier si $R \neq NULL$ [2].

Le fils droit : La fonction *noeud * droit(noeud * R)* retourne l'adresse du fils droit du nœud d'adresse R.

```
noeud * droit(noeud * R)
{
  if (est_vide(R)) return NULL;
  else return R->droit;
}
```

3. Vérifier si un nœud est une feuille

La fonction *bool est_feuille(noeud * R)* vérifie si le nœud d'adresse R est une feuille. Autrement dit, elle retourne la valeur *true* si le nœud n'a pas de fils et $false$ sinon.

```
bool est_feuille(noeud * R)
{
  if (est_vide(R)) return false;
  else
  {
    if (est_vide(R->gauche) && est_vide(R->droit)) return true;
    else return false;
  }
}
```

4. Vérifier si un nœud est interne (pas une feuille)

La fonction *bool est_interne(noeud * R)* vérifie si le nœud d'adresse R est un nœud interne. Autrement dit, elle retourne la valeur *vrai* si le nœud a des enfants et $faux$ sinon.

```
bool est_interne(noeud * R)
{
  return (! est_feuille(R));
}
```

2. Dans le cas $R = NULL$, l'accès à un des fils gauche ou droit peut être l'origine d'un bug

5. Calcul de la hauteur d'un arbre

Le calcul de la hauteur d'un arbre revient à calculer le nombre de niveaux contenus dans l'arbre.

Pour mieux expliquer la notion de « hauteur d'un arbre », nous introduisons l'exemple 3.3.3 ci-dessous :

Exemple 3.3.3 (hauteur d'un arbre) *Soit la figure 56 ci-dessous qui donne la structuration en niveaux de l'arbre binaire d'entier introduit dans l'exemple 5.3.3.*

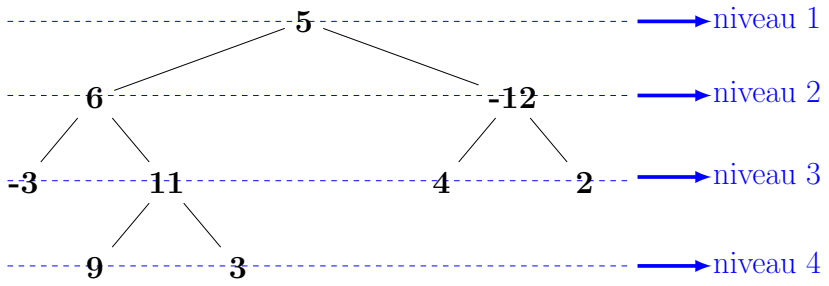


FIGURE 56 – Niveaux de l'arbre binaire de la figure 54

Les niveaux sont numérotés de 1 *à* 4 *en commençant par celui de la racine. Donc, la hauteur de l'arbre est* 4.

La fonction récursive *unsigned int hauteur(noeud * racine)* retourne la hauteur d'un arbre binaire en démarrant de la racine[3].

```
unsigned int hauteur(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  return(1+max(hauteur(gauche(racine)), hauteur(droit(racine))));
}
```

6. Calcul du nombre de nœuds d'un arbre

La fonction *unsigned int nbre_noeud(noeud * racine)* calcule le nombre de nœuds contenus dans un arbre binaire.

```
unsigned int nbre_noeud(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  return(1+nbre_noeud(gauche(racine)),nbre_noeud(droit(racine))));
}
```

3. La fonction max est une fonction prédéfinie dans la bibliothèque *math.h*, elle calcule le maximum de deux paramètres

7. Calcul du nombre de feuilles d'un arbre

Calculer le nombre de feuilles revient à calculer le nombre de nœuds sans enfants. Nous avons ci-dessous la fonction *unsigned int nbre_feuille(noeud * racine)* qui retourne le nombre de feuilles contenues dans d'un arbre binaire dont l'adresse de la racine est mis en paramètre d'entrée *racine*.

```
unsigned int nbre_feuille(noeud * racine)
{
  if (est_vide(racine)) return 0;
  else
  {
    if (est_feuille(racine)) return 1;
    else
    return
    nbre_feuille(gauche(racine))+nbre_feuille(droit(racine));
  }
}
```

8. Différents parcours d'un arbre

Parcourir un arbre, c'est accéder aux nœuds de l'arbre en suivant un des types de parcours ci-dessous :

1. parcours en profondeur préfixe,
2. parcours en profondeur infixe,
3. parcours en profondeur suffixe,
4. parcours en largeur.

Parcours en profondeur préfixe : Dans ce cas, les nœuds sont parcourus dans l'ordre : $racine \rightarrow fils\_gauche \rightarrow fils\_droit$. Ce qui donne en C++ la fonction ci-dessous *parcours_profondeur_prefixe*.

```
void parcours_profondeur_prefixe(noeud * racine)
{
  if (! est_vide(racine))
  {
    traiter(racine);//traiter le noeud racine
    parcours_profondeur_prefixe(gauche(racine));
    parcours_profondeur_prefixe(droit(racine));
  }
}
```

En prenant comme exemple l'arbre binaire illustré dans la figure 54 et en utilisant le parcours en profondeur préfixe, l'accès au nœuds suit l'ordre suivant : $5, 6, -3, 11, 9, 3, -12, 4, 2$.

Parcours en profondeur infixe : Dans ce cas, les nœuds sont parcourus dans l'ordre : $fils\_gauche \rightarrow racine \rightarrow fils\_droit$.

Nous introduisons ci-dessous la fonction *parcours_profondeur_infixe* qui exprime le parcours *infixe* en C++ :

```
void parcours_profondeur_infixe(noeud*racine)
{
  if (!est_vide(racine))
  {
    parcours_profondeur_infixe(gauche(racine));
    traiter(racine);//traiter le noeud racine
    parcours_profondeur_infixe(droit(racine));
  }
}
```

L'application de ce type de parcours sur l'arbre de la figure 54, les nœuds seront parcourus comme suit : $-3, 6, 9, 11, 3, 5, 4, -12, 2$.

Parcours en profondeur suffixe : Dans ce cas, la racine est parcourue en dernier, l'ordre des nœuds se donne comme suit : $fils\_gauche \rightarrow fils\_droit \rightarrow racine$.

```
void parcours_profondeur_sufffixe(noeud*racine)
{
  if (! est_vide(racine))
  {
    parcours_profondeur_sufffixe(gauche(racine));
    parcours_profondeur_sufffixe(droit(racine));
    traiter(racine);//traiter le noeud racine
  }
}
```

En suivant ce parcours, l'ordre d'accès aux nœuds de l'arbre de la figure 54 se donne comme suit : $-3, 9, 3, 11, 6, 4, 2, -12, 5$.

Parcours en largeur : Il est différent du parcours en profondeur, dans ce cas le parcours de l'arbre se fait par niveau, tel exprimé par la fonction ci-dessous *parcours_largeur*.

```
void parcours_largeur(noeud * racine)
{
  noeud * temp;
  file F;//F est une file pour stocker les noeuds en attente
  if (!est_vide(racine))
  {
    enfiler(F,racine);
    while(! est_vide(F))
    {
      temp = defiler(F);
      // traiter le noeud temp
     if (!est_vide(gauche(Temp))) enfiler(F,Temp);
     if (!est_vide(droit(Temp))) enfiler(F,Temp);
    }
  }
}
```

Dans la fonction nous utilisons une structure intermédiaire *file*. Le principe est d'enfiler les valeurs des nœuds par niveau. Par conséquent le fait de défiler les éléments de la file implique la récupération des valeurs de gauche à droite par niveau.

L'application de la fonction sur l'arbre de la figure 56 donne en résultat l'ordre des valeurs comme suit : $5, 6, -12, -3, 11, 4, 2, 9, 3$.

9. Créer un nœud d'un arbre binaire

Puisque nous utilisons une structure dynamique, la création d'un nœud nécessite l'allocation de l'espace mémoire suffisant pour charger ce nœud.

La fonction *noeud * creer_noeud(TypeElt val)* permet la création d'un nœud d'un arbre binaire, dont la valeur est donnée en entrée *val* et l'adresse est le résultat retourné.

```
noeud * creer_noeud(TypeElt val)
{
  noeud * r = new noeud;
  if (r == NULL) cout << "mémoire insuffisante!" << endl;
  else
  {
    r->elt = val;
    r->gauche = NULL;
    r->droit = NULL;
  }
    return r;
}
```

10. Créer la racine d'un arbre binaire

La fonction *noeud * creer_racine(TypeElt val, noeud *g, noeud *d)* permet la création de la racine d'un arbre binaire, avec l'existence au préalable des deux fils d'adresses respectives g et d.

```
noeud * creer_racine(TypeElt val, noeud * g, noeud * d)
{
  noeud * r = new noeud;
  if (r == NULL) cout << "mémoire insuffisante!" << endl;
  else
  {
    r->elt =val;
    r->gauche = g;
    r->droit = d;
  }
  return r;
}
```

11. Exemple d'insertion d'un nœud

Soient les valeurs à insérer dans l'ordre, dans un arbre binaire vide *racine* = *NULL* : 5, 6, -12, -3, 11, 4 et 2.

Nous introduisons ci-dessous la déclaration de l'arbre.

```
struct noeud
{
        int elt;
        noeud * gauche;
        noeud * droit;
}
noeud * racine=NULL;
```

La fonction ci-dessous *ajout_elt* permet l'insertion de ces valeurs de telle sorte à ce qu'elle nous donne en résultat l'arbre binaire illustré dans la figure 57.

```
void ajout_elt(noeud * r, int val)
{
  if (r != NULL)
  {
    if (est_vide(gauche(r))) r->gauche = creer_noeud(val);
    else
      if (est_vide(droit(r))) r->droit = creer_noeud(val);
      else ajout_elt(gauche(r), val);
   }
   else r= creer_racine(val, NULL,NULL);
}
```

L'insertion de 7 éléments d'un tableau d'entiers se fait via l'appel de *ajout_elt* se donne comme suit :

```
...
int tab[7]={5,6,-12,-3,11,4,2};
for(int i=0;i<=6;i++)
        ajouter_elt(racine,tab[i]);
...
```

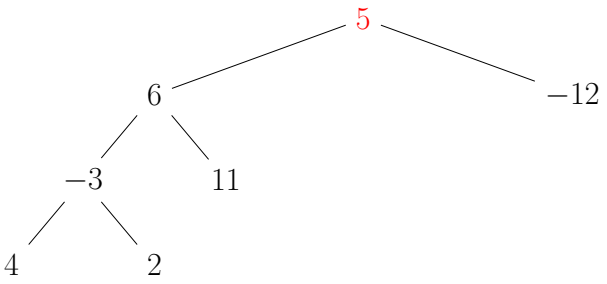


FIGURE 57 – Arbre binaire correspondant à l'application de la fonction *ajout_elt* après insertion des valeurs $5, 6, -12, -3, 11, 4, 2$

Arbre planaires généraux

Soit la définition 3.3.4 suivante :

Définition 3.3.4 (arbre planaire général) *[Froidevaux et al., 1993] Un arbre $A = \langle \circ, A_1, \cdots A_p \rangle$ est la donnée d'un racine et d'une liste finie, éventuellement vide (si $p = 0$), d'abord disjoints.*

Dans une structure arborescente appelée arbre planaire général ou tout brièvement arbre général ou arbre, le nombre de fils de chaque nœud n'est plus limité à 2. On peut représenter les arbres en trois façons[Froidevaux et al., 1993].

1. Représentation dynamique

On peut donner à chaque nœud la liste de ses fils. Cette représentation des arbres peut être utile, mais elle se prête mal à une gestion dynamique.

```
struct list_noeud
{
        int noeud; //valeur du noeud
        list_noeud * suivant; //pointeur vers le noeud frère
}
liste_noeud * vect_noeud[nbre_noued]; //nbre_noeud est le
                                      //nombre de noeuds
```

2. Représentation analogue à un arbre binaire

On peut aussi décrire une représentation analogue à celle des arbres binaires. Chaque nœud contient un pointeur vers chacun des sous-arbres et éventuellement un champ pour stocker l'élément contenu dans le nœud.

```
struct arbre_noeud
{
        int val_noeud; //valeur du noeud
        list_noeud * frere1; //pointeur vers premier frère
        list_noeud * frere2; //pointeur vers le 2ème frère
           ...
        list_noeud * freren; //pointeur vers le n-ème frère
}
arbre_noeud * racine;
```

Pour utiliser cette structure il faut connaître le nombre maximum de fils que peut avoir un nœud de l'arbre.

3. Représentation par un arbre binaire

On peut transformer l'arbre général en un arbre binaire et utiliser la structure en C++ décrite précédemment dans la section 3.3.1. La transformation se fait au niveau de chaque nœud de la manière suivante :

Considérons un nœud x qui a n fils notés f_1, f_2, ...f_n, dans un arbre général. Pour constituer un arbre binaire :

- f_1 devient le fils gauche de x,
- f_2 devient le fils droit de f_1,
- f_3 devient le fils droit de f_2,
- ...
- f_n devient le fils droit de f_{n-1}.

Exemple 3.3.5 (passage d'un arbre général à un arbre binaire) *Considérons l'arbre général d'entiers donné dans la figure ci-dessous 58.*

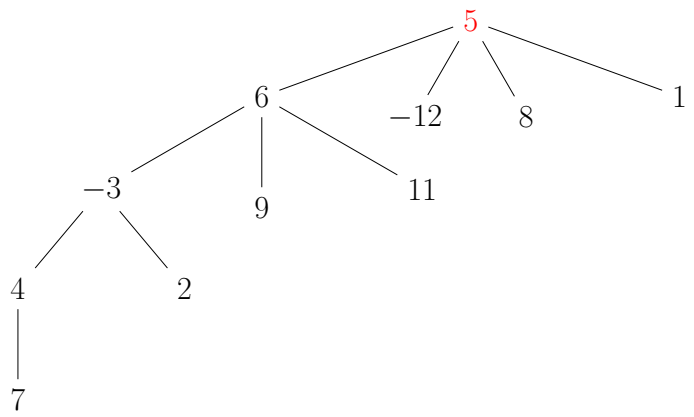


FIGURE 58 – Arbre général d'entiers

En suivant le principe de passage vers un arbre binaire, cité précédemment, nous obtenons l'arbre binaire illustré dans la figure ci-dessous 59.

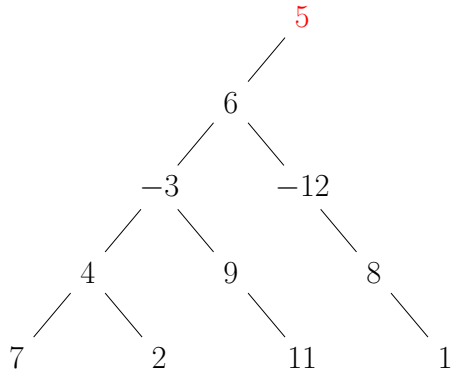


FIGURE 59 – Arbre binaire correspondant à l'arbre général 58

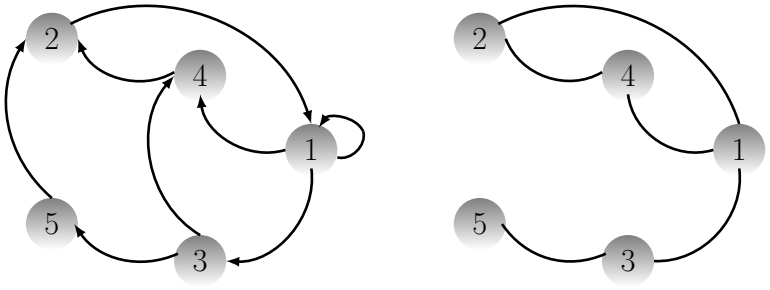


FIGURE 60 – Exemple de graphes orienté et non-orienté