Chapitre 17

Structures de données élémentaires

1. Généralités

- Les structures de données sont une part essentielle de la programmation.
- Une structure de données correspond à un ensemble de deux ou plusieurs données, formant ainsi un groupe ou un ensemble que l'on peut traiter à loisir.
- Cet ensemble de données peut-être constitué de différentes sortes d'éléments : des entiers, réels, caractères, comptes bancaires etc.
- On ne s'intéresse pas aux éléments de l'ensemble i.e. à leur implémentation mais aux opérations que nous pouvons réaliser sur cet ensemble.

- Les opérations de base définies sur cet ensemble sont comme suit:
  - Tester si l'ensemble est vide.
  - Ajouter un élément à l'ensemble.
  - Vérifier si un élément appartient à l'ensemble.
  - Supprimer un élément de l'ensemble.

- La gestion de cet ensemble doit tenir compte de deux critères essentiels :
  - Un minimum de place mémoire utilisée.
  - Un minimum d'opérations i.e. réduire le temps de traitement des opérations.

- Parmi les avantages :
  - Ces structures de données peuvent être utilisées par d'autres programmes sans se soucier de la manière avec laquelle elles ont été implémentées.
  - Cette approche permet aussi de rationaliser une implémentation et donc, elle permet d'aider à faire la spécification.

2. Types de collections

2.1 Séquentiels

- Dans cette catégorie de collections, l'accès est linéaire. C'est l'ordre qui compte.
- Les éléments peuvent être stockés sous la forme :
  - Un tableau
    - Dans ce cas, les éléments sont accessibles à partir d'un index.
    - Ajouter ou enlever des éléments à partir de la fin du tableau est une opération très rapide à exécuter.
    - Par contre, insérer un élément au début ou bien au milieu du tableau, il prend un certain temps car les éléments doivent être déplacés un par un pour créer une place au nouvel arrivant.
  - Une file
    - C'est un tableau dynamique qui peut grossir dans les deux directions : une file bilatérale.
    - L'insertion des éléments au début ou bien à la fin de la file est une opération rapide à exécuter.
    - Par contre insérer un élément au milieu va prendre un certain temps car il faudra déplacer les éléments.
  - Une liste
    - Les listes ne permettent pas un accès aléatoire c.-à-d. à un index donné comme le font les tableaux et les files. Pour accéder au Nième élément de la liste, il faut passer par les N-1 prédécesseurs.
    - L'accès à un élément donné prend un temps linéaire nettement supérieur que, si cet élément était dans un tableau ou une file.
    - L'avantage des listes est que l'insertion ou le retrait d'un élément se fait rapidement.
    - Il suffit de modifier les adresses des pointeurs, des prédécesseurs et des successeurs.

2.2 Associatifs

- Dans cette catégorie de collections, l'accès se fait par clés.

- Les éléments sont rangés suivant un certain critère. Ce critère est une sorte de fonction qui compare suivant la valeur ou bien la clé associée à la valeur.

- Parmi ces collections nous citerons les dictionnaires (« map »).

2.3 Adaptateurs

- Dans cette catégorie de collections, l'accès est réglementé.

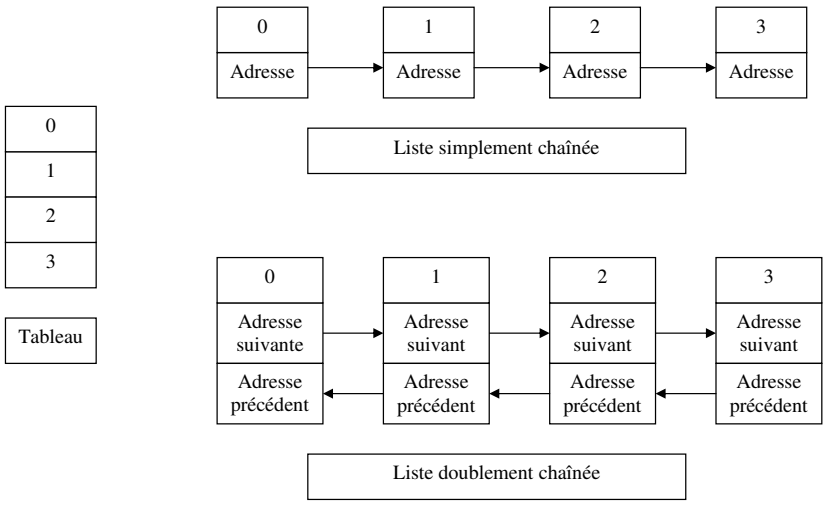
- Ces conteneurs adaptent les conteneurs séquentiels afin de remplir des missions précises.

- Parmi ces adaptateurs, nous citerons: pile (« stack »), file (« queues ») et les files prioritaires (« priority queues »).

3. Listes

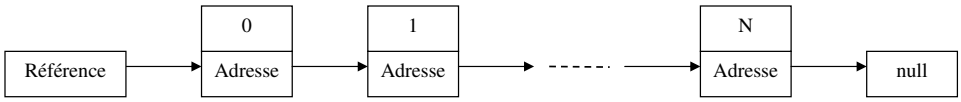
3.1. Généralités

- Un tableau volumineux reste inefficace quand on veut insérer un élément dans une position donnée (une opération très coûteuse).
- Dans ce cas là, nous avons recours aux liens dynamiques comme par exemple les listes.
- Chaque élément de la liste a son propre segment de mémoire et peut pointer son prédécesseur et/ou son successeur.
- Dans une liste, le nombre d'éléments et l'ordre de ses éléments peuvent être facilement changés.
- Par contre, il faut parcourir la liste (à partir de la tête parfois à partir de la queue) pour retrouver un élément donné.
- Cette opération est moins rapide que celle permettant d'accéder grâce à l'index, à un élément d'un tableau donné.



- Liste simplement chaînée : l'accès se fait à partir du premier élément vers l'avant uniquement.
- Liste doublement chaînée : l'accès peut se faire à partir du premier ou dernier élément, donc indifféremment vers l'avant ou vers l'arrière.

3.2. Référence à une liste



Une référence à une liste est référence au 1er élément qui constitue la tête. Le dernier élément (queue) pointe vers un objet vide (nul).

3.3. Composantes d'une liste

Un élément de la liste est composé de 2 parties:

- Les attributs pour spécifier la "valeur" de l'élément.
- Un pointeur vers son successeur.

```
class element {

      déclaration des attributs de l'élément;
      element prochain;
      éventuellement des méthodes;

};
```


Par exemple, pour une liste dont les éléments sont des entiers, les composants d'une telle liste sont comme suit :

```
class Cellule {
      int valeur;
      Cellule* prochain;
};
```

Ou bien:

```
class Cellule {
      int valeur;
      Cellule* prochain;
public:
      Cellule(int v, Cellule* p) {
            valeur = v;
            prochain = p;
      }
      // avec une panoplie de méthodes… voir plus bas …
};
```

3.4. Opérations possibles sur une liste

3.4.1. Ajouter un élément : pour une liste simplement chaînée, on insère l'élément à la tête de la liste. Si l'on doit insérer « 1 » à la tête de la liste représentée par {2,3,4}, le résultat obtenu est : {1,2,3,4}

```
Cellule* ajout_elt(int v) {

     Cellule* tete = new Cellule(v,this);
     return tete;

     // return (new Cellule(v,this);
}
```

3.4.2. Connaître la longueur de la liste : nous devons juste balayer la liste pour connaître le nombre d'éléments qu'elle contient. Par exemple, la liste {1,2,3,4,5} contient 5 éléments.

```
int longueur() {
    int l=0;
    Cellule* ref = this;
    while(ref) {
        l++;
        ref = ref->prochain;
    }
    return l;
}
```


3.4.3. Trouver le énième élément et le retourner : la liste est parcourue élément par élément jusqu'à l'index recherché. Si ce dernier existe, l'élément correspondant est retourné. Par exemple, retourner le 3^{e} élément de la liste {1,2,3,4,5}. Les erreurs de déplacement ne sont pas traitées.

```
Cellule* nieme_cellule(int n) {
    Cellule* t;
    if (n==1) t=this;
    else t = prochain->nieme(n-1);
    return t;
}
```

3.4.4. Trouver le énième élément et retourner sa valeur : ce qui est différent avec 3.4.3 c'est qu'ici, on ne retourne pas toute la cellule mais uniquement la valeur associée à cette cellule. Dans cet exemple, on retourne un « int » car les valeurs des éléments de la liste sont des « int ».

```
int nieme_valeur(int n) {
    int valeur ;
    if (n==1) valeur=this->valeur;
    else valeur = prochain->nieme(n-1)->valeur;
    return valeur;
}
```

3.4.5. Concaténer deux listes : l'opération consiste à ajouter à une liste tous les éléments d'une seconde liste. Par exemple, concaténer les listes {1,2,3} et {4,5} donne le résultat suivant : {1,2,3,4,5}.

```
void append(Cellule* l) {

     if (prochain == NULL) prochain =l;
     else prochain->append(l);

}
```

3.4.6. Ajouter une cellule de valeur v à la queue de la liste : dans une liste doublement chaînée, cette cellule peut-être ajoutée aussi à la tête de la liste.

```
void ajout_queue(int v) {
     if (prochain == NULL) prochain = new Cellule(v,NULL);
     else prochain->ajout_queue(v);
}
```


3.4.7. Enlever le premier élément de valeur v : on parcourt la liste à la recherche de l'élément en question. On le retire si on le trouve. On arrête par la suite la recherche même s'il reste dans la liste des éléments ayant la valeur recherchée.

```
Cellule* enlever_premier(int v) {
     if (valeur == v) return prochain;
     else {
          prochain = prochain->enlever_premier(v);
          return this;
     }
}
```

3.4.8. Enlever tous les éléments de valeur v : on parcourt la liste à la recherche de l'élément en question. On le retire si on le trouve. On poursuit la recherche sur les éléments restants.

```
Cellule* enlever_tous(int v) {
     if (valeur == v) return prochain->enlever_tous(v);
     else {
          prochain = prochain->enlever_tous(v);
          return this;
     }
}
```

3.5 Une autre implantation de la liste

Afin de réaliser une meilleure abstraction dans l'implantation, on peut distinguer entre l'élément et la suite d'éléments. L'élément est représenté par une cellule (nœud) et la suite d'éléments chaînés par la liste. La liste peut garder par exemple des informations comme le nombre d'éléments qu'elle contient, un pointeur vers le premier élément ainsi que vers le dernier élément etc.

```
class Cellule {
      int valeur;
      Cellule* prochain;
   public:
      Cellule(int v, Cellule* suivant);
      Cellule* get_prochain();
      void set_prochain(Cellule* suivant);
      int get_valeur();
}

class Liste {

      Cellule *tete, *queue;
      int longueur;

      etc.
}
```


L'ajout d'un élément se fait comme suit :

À partir de la tête

```
void Liste::ajout_tete(int v) {
    Cellule* t = new Cellule(v,tete);
    tete = t;
    longueur++;
    if (queue == NULL) queue=t; // si t est le 1er élément ajouté.
}
```

À partir de la queue

```
void Liste::ajout_queue(int v) {
    if (queue == NULL)
        tete=queue= new Cellule(v,NULL);
    else {
        Cellule* t = new Cellule(v,NULL);
        queue->set_prochain(t);
        queue = t;
    }
    longueur++;
}
```

D'une autre liste

```
void Liste::ajout_liste(Liste* l) {
    if (queue==NULL){
        queue=l->queue;
        tete=l->tete;
    }else{
        queue->set_prochain(l->tete);
        queue = l->queue;
    }
    longueur += l->longueur;
}
```

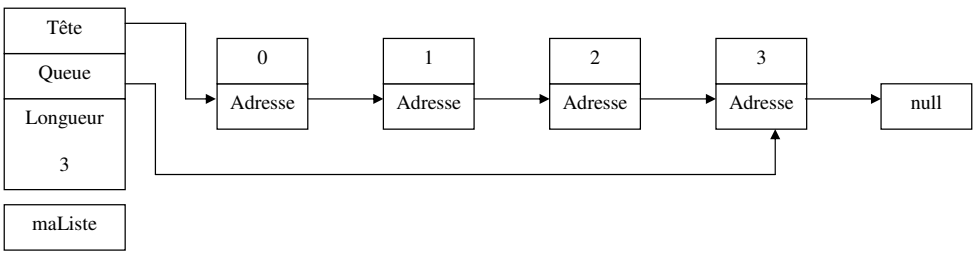
Afficher les éléments de la liste

```
void Liste::affiche(){
    Cellule* t = tete;
    while(t){
            cout << t->get_valeur();
            t=t->get_prochain();
            if (t) cout << " ";
    }
    cout << endl;
}
```


Exemple sur une liste d'entiers

```
Liste maListe; // tete=queue=null;longueur=0;

maListe.ajout_queue(1);
maListe.ajout_queue(2);
maListe.ajout_queue(3);
maListe.ajout_tete(0);
```



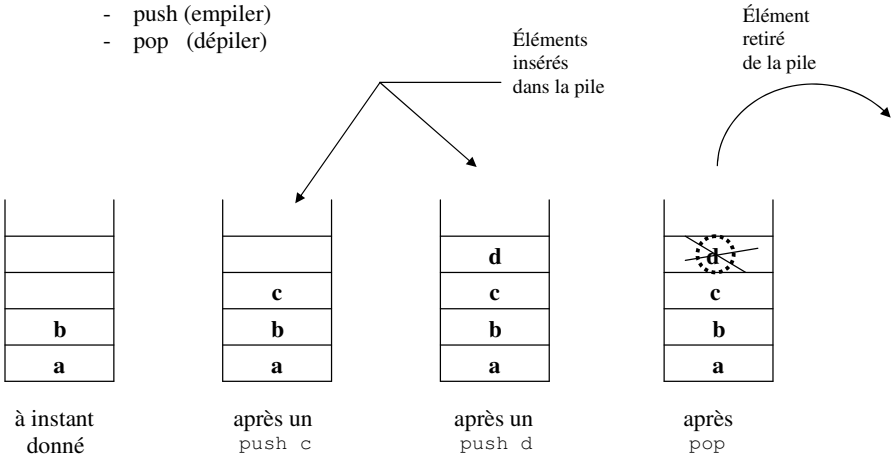
Voir les fichiers « liste.h » et « liste.cpp ».

4. Piles

4.1. Description de la pile

La pile décrite dans cet exemple a les caractéristiques suivantes:

- C'est une structure de données contenant des éléments du type `int`.
- Elle gère les 2 opérations :
  - push (empiler)
  - pop (dépiler)



- Elle est du type LIFO (Last In First Out)

4.2. Programme

Voir les fichiers « pile.h » et « pile.cpp »

4.3. Exercice

Modifier la classe « Pile » pour traiter le type « char * ».