Algorithmique 3

L2 informatique

Listes, piles, files

Adrien GOËFFON
adrien.goeffon@univ-angers.fr

Types et structures de données

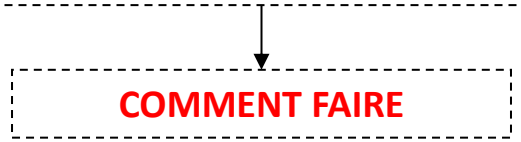
Une **donnée** représente une information que l'on peut stocker ou manipuler à l'aide d'un algorithme ou d'un système informatique.
Un **type** caractérise l'ensemble des valeurs admissibles pour une donnée ainsi que les opérations applicables.
Un **type abstrait** de données est une définition formelle, indépendante de toute implémentation, du comportement attendu d'un type : spécification des primitives (*opérations*), propriétés, préconditions, éventuellement complexité. Il constitue l'**interface** logique que l'implémentation devra respecter.

Une **structure de données** décrit le stockage, l'organisation et l'accès aux données de sorte qu'elles puissent être utilisées efficacement. L'**implémentation** d'une structure de données précise comment représenter les données et réaliser les opérations définies dans le type abstrait.

Types et structures de données

Une **donnée** représente une information que l'on peut stocker ou manipuler à l'aide d'un algorithme ou d'un système informatique.
Un **type** caractérise l'ensemble des valeurs admissibles pour une donnée ainsi que les opérations applicables.
Un **type abstrait** de données est une définition formelle, indépendante de toute implémentation, du comportement attendu d'un type : spécification des primitives (*opérations*), propriétés, préconditions, éventuellement complexité. Il constitue l'**interface** logique que l'implémentation devra respecter.

Une **structure de données** décrit le stockage, l'organisation et l'accès aux données de sorte qu'elles puissent être utilisées efficacement. L'**implémentation** d'une structure de données précise comment représenter les données et réaliser les opérations définies dans le type abstrait.



Types et structures de données

Types primitifs : éléments fondamentaux servant de base à la construction des autres types
entier, flottant, booléen, caractère, pointeur

Types composés
tableau, enregistrement

Types abstraits
chaîne de caractères, liste, pile, file, arbre, tas, table de hachage, graphe, …

Les bibliothèques standard des langages fournissent des implémentations de types abstraits.
L'utilisation de ces structures repose sur leur interface ; leur implémentation peut rester cachée pour l'utilisateur.

Types et structures de données

Types primitifs : éléments fondamentaux servant de base à la construction des autres types
entier, flottant, booléen, caractère, pointeur

Types composés
tableau, enregistrement

Types abstraits
chaîne de caractères, liste, pile, file, arbre, tas, table de hachage, graphe, …

Les bibliothèques standard des langages fournissent des implémentations de types abstraits.
L'utilisation de ces structures repose sur leur interface ; leur implémentation peut rester cachée pour l'utilisateur.

UE Algorithmique 3 :
comprendre l'intérêt et l'usage des types abstraits, les implémenter et employer des structures de données avancées

Listes

Une **liste** est une suite ordonnée, éventuellement vide, d'éléments de même type.

$$L = (e_1, e_2, \dots, e_n)$$

Listes

Une **liste** est une suite ordonnée, éventuellement vide, d'éléments de même type.

$$L = (e_1, e_2, \dots, e_n)$$

Exemples de listes :

- (2, 3, 5, 7, 11, 13, 17, 19) : liste des nombres premiers inférieurs à 20.
- (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) : liste des jours de la semaine
- (G, G, N, G, P, N, G, P, G, N, G) : liste de résultats d'un jeu à 2 joueurs sur 11 manches
- () : liste vide

Listes

Une **liste** est une suite ordonnée, éventuellement vide, d'éléments de même type.

$$L = (e_1, e_2, \dots, e_n)$$

Exemples de listes :

- (2, 3, 5, 7, 11, 13, 17, 19) : liste des nombres premiers inférieurs à 20.
- (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) : liste des jours de la semaine
- (G, G, N, G, P, N, G, P, G, N, G) : liste de résultats d'un jeu à 2 joueurs sur 11 manches
- () : liste vide

La liste est un type très général, pouvant être représentée par une structure de données dite *linéaire* (ou *séquentielle*). On utilise une liste pour gérer un ensemble de données homogène, sur lequel on peut effectuer des ajouts et suppressions, ainsi que des accès à tout élément.

Opérations usuelles :

→ créer (initialiser) une liste
→ accéder au premier élément / au dernier élément / au *k*-ième élément
→ insérer un élément en tête de liste / en fin de liste / en position *k*
→ supprimer le premier élément / le dernier élément / le *k*-ième élément
→ rechercher un élément dans une liste
→ afficher une liste
→ concaténer deux listes
→ supprimer (détruire) une liste
→ …

Listes : interface

La spécification abstraite décrit les propriétés générales des opérations propres à la structure de données : syntaxe du type et signification des opérations.

Exemple de spécification d'une liste d'éléments de type T :

Nom : liste_T

Utilise T, entier, booleen

Opérations :

- Creer : $\varnothing \rightarrow$ liste_T — *crée une liste vide*
- Est_vide : liste_T $\rightarrow$ booleen — *détermine si une liste donnée est vide*
- Acces_element : liste_T $\times$ entier $\rightarrow$ T — *retourne l'élément figurant à une position donnée*
- Longueur : liste_T $\rightarrow$ entier — *retourne le nombre d'éléments de la liste donnée*
- Inserer_position : liste_T $\times$ T $\times$ entier $\rightarrow$ liste_T — *insère un élément dans une liste à une position donnée*
- Supprimer_position : liste_T $\times$ entier $\rightarrow$ liste_T — *supprime l'élément d'une liste figurant à une position donnée*
- Rechercher : liste_T $\times$ T $\rightarrow$ entier — *retourne la position de la première occurrence d'un élément donné, ou 0 par défaut*

Préconditions :

- Acces_element(L,k) défini si $1 \leq k \leq$ Longueur(L)
- Inserer_position(L,e,k) défini si $1 \leq k \leq$ Longueur(L) + 1
- Supprimer_position(L,k) défini si $1 \leq k \leq$ Longueur(L)

Listes : interface

En pratique, on peut inclure dans l'interface des éléments généraux de l'implémentation relatifs au paradigme de programmation utilisé (ici la **programmation impérative**), ce qui permet de préciser la nature des paramètres.
La manière de passer les différents paramètres sera établie en fonction du langage de programmation et des choix d'implémentation de la structure de données.

fonction Creer () : liste_T *ou* fonction Creer (**res** L : liste_T)
fonction Est_vide (L : liste_T) : booleen
fonction Acces_element (L : liste_T, pos : entier) : T
fonction Longueur (L : liste_T) : entier
procedure Inserer_position (**dm** L : liste_T, elt : T, pos : entier)
procedure Supprimer_position (**dm** L : liste_T, pos : entier)
fonction Rechercher (L : liste_T, elt : T) : entier

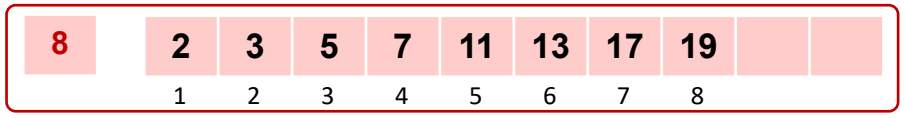
Une liste peut être spécifiée de manière plus ou moins détaillée selon l'utilisation prévue.
Un ajout de fonctionnalités peut rendre son utilisation moins efficace si des compromis doivent être effectués lors du choix de l'implémentation.

Listes : implémentation

Les deux implémentations classiques pour une liste sont :
- l'implémentation contigüe, à l'aide de tableaux
- l'implémentation chaînée, avec allocation dynamique de la mémoire

(2, 3, 5, 7, 11, 13, 17, 19)

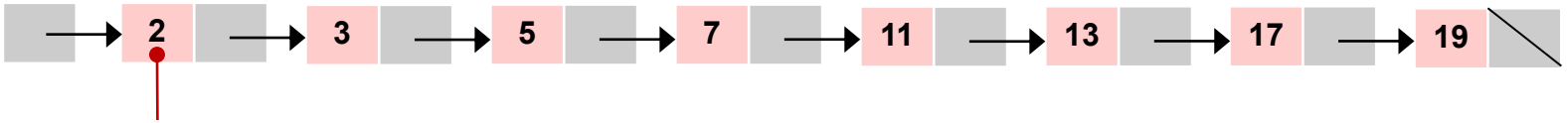
Implémentation contigüe




```
type Liste_entier = enregistrement
        longueur : entier
        T: tableau [1..Lmax] de entier
fin
```

Implémentation par liste chaînée

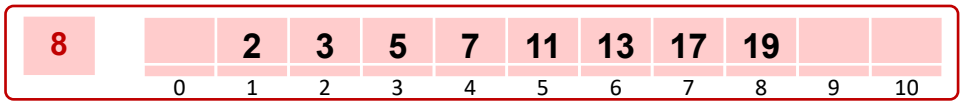
```
type Liste_entier = ^maillon
type maillon = enregistrement
        valeur : entier
        suivant : ^maillon
fin
```



élément de type donné (généralement type simple ou pointeur vers type structuré)

Listes : implémentation (C++)

Exemple 1




```
const int Lmax = 10 ;
struct Liste_entier
{
	int longueur ;
	int T[Lmax+1] ;
} ;

Liste_entier creer ()
{
	Liste_entier L;
	L.longueur = 0 ;
	return L;
}

int longueur (Liste_entier L)
{
	return L.longueur ;
}
```

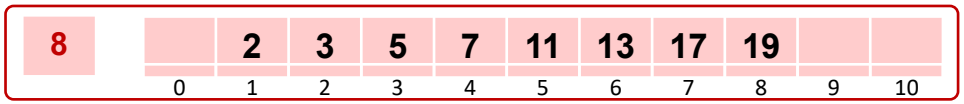
```
int acces_element (Liste_entier L, int pos)
{
	return L.T[pos] ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
	for (int i = L.longueur; i >= pos; --i)
		L.T[i+1] = L.T[i] ;
	L.T[pos] = elt ;
	++L.longueur ;
}

// …
```

Listes : implémentation (C++)

Exemple 1'




```
const int Lmax = 10 ;
struct Liste_entier
{
        int longueur ;
        int T[Lmax+1] ;
} ;

void creer (Liste_entier & L)
{
        L.longueur = 0 ;
}

int longueur (const Liste_entier & L)
{
        return L.longueur ;
}
```

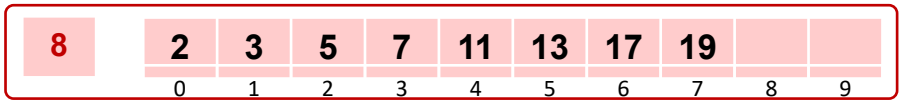
```
int acces_element (const Liste_entier & L, int pos)
{
        return L.T[pos] ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
        for (int i = L.longueur; i >= pos; --i)
                L.T[i+1] = L.T[i] ;
        L.T[pos] = elt ;
        ++L.longueur ;
}

// …
```

Listes : implémentation (C++)

Exemple 2




```
const int Lmax = 10 ;
struct Liste_entier
{
        int longueur ;
        int T[Lmax] ;
} ;

void creer (Liste_entier & L)
{
        L.longueur = 0 ;
}

int longueur (const liste_entier & L)
{
        return L.longueur ;
}
```

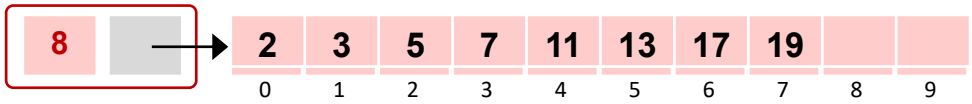
```
int acces_element (const Liste_entier & L, int pos)
{
        return L.T[pos-1] ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
        for (int i = L.longueur; i >= pos; --i)
                L.T[i] = L.T[i-1] ;
        L.T[pos-1] = elt ;
        ++L.longueur ;
}

// …
```

Listes : implémentation (C++)

Exemple 3




```
const int Lmax = 10 ;
struct Liste_entier
{
        int longueur ;
        int * T ;
} ;

void creer (Liste_entier & L)
{
        L.longueur = 0 ;
        L.T = new int [Lmax] ;
}

int longueur (const Liste_entier & L)
{
        return L.longueur ;
}
```

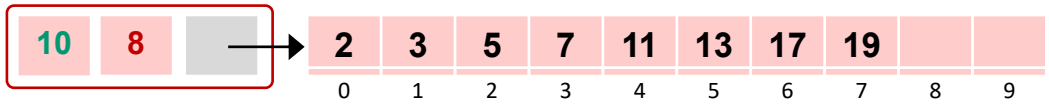
```
int acces_element (const Liste_entier & L, int pos)
{
        return L.T[pos-1] ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
        for (int i = L.longueur; i >= pos; --i)
                L.T[i] = L.T[i-1] ;
        L.T[pos-1] = elt ;
        ++L.longueur ;
}

// …
```

Listes : implémentation (C++)

Exemple 4




```
const int Lmax = 10 ;
struct Liste_entier
{
        int longueur_max ;
        int longueur ;
        int * T ;
} ;
```

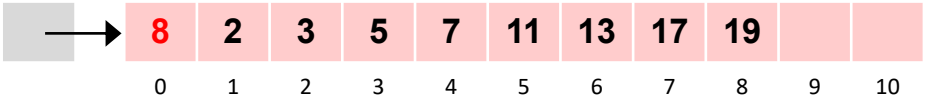
```
void creer (Liste_entier & L)
{
        L.longueur_max = Lmax ;
        L.longueur = 0 ;
        L.T = new int [Lmax] ;
}

void creer (Liste_entier & L, int longueur_max)
{
        L.longueur_max = longueur_max ;
        L.longueur = 0 ;
        L.T = new int [longueur_max] ;
}

// …
```

Listes : implémentation (C++)

Exemple 5




```
const int Lmax = 10 ;
using Liste_entier = int * ;

void creer (Liste_entier & L)
{
	L = new int [Lmax+1] ;
	L[0] = 0 ;
}

int longueur (const Liste_entier & L)
{
	return L[0] ;
}
```

```
int acces_element (const Liste_entier & L, int pos)
{
	return L[pos] ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
	for (int i = L[0]; i >= pos; --i)
		L[i+1] = L[i] ;
	L[pos] = elt ;
	++L[0] ;
}

// …
```

Listes : implémentation (C++)

Exemple 6


```
struct maillon
{
	int valeur ;
	maillon * suivant ;
} ;
using Liste_entier = maillon * ;

void creer (Liste_entier & L)
{
	L = nullptr ;
}
```

```
int longueur (Liste_entier L)
{
	int longueur = 0 ;
	while (L != nullptr)
	{
		L = L->suivant ;
		++longueur ;
	}
	return longueur ;
}
```

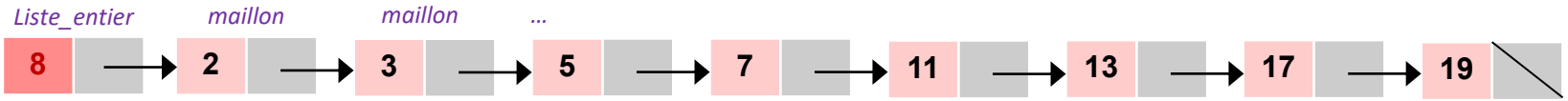
```
int acces_element (Liste_entier L, int pos)
{
	if (pos == 1) return L->valeur ;
	else return acces_element(L->suivant,pos-1) ;
}

void inserer_position (Liste_entier & L, int elt, int pos)
{
	if (pos == 1)
	{
		maillon * p = new maillon ;
		p->valeur = elt ;
		p->suivant = L ;
		L = p ;
	}
	else inserer_position(L->suivant,elt,pos-1) ;
}

// …
```

Listes : implémentation (C++)

Exemple 7




```
struct maillon
{
	int valeur ;
	maillon * suivant ;
} ;
struct Liste_entier
{
	int longueur ;
	maillon * liste ;
} ;
```

```
void creer (Liste_entier & L)
{
	L.longueur = 0 ;
	L.liste = nullptr ;
}

int longueur (Liste_entier & L)
{
	return L.longueur ;
}

int acces_element (Liste_entier & L, int pos)
{
	maillon * p = L.liste ;
	for (int i = 1; i < pos; ++i)
		p = p->suivant ;
	return p->valeur ;
}
```

```
void inserer_position (Liste_entier & L, int elt, int pos)
{
	maillon * p = L.liste ;
	for (int i = 2 ; i < pos ; ++i)
		p = p->suivant ;
	maillon * q = new maillon ;
	q->valeur = elt ;
	if (pos == 1)
	{
		q->suivant = p ;
		L.liste = q ;
	}
	else {
		q->suivant = p->suivant ;
		p->suivant = q ;
	}
	++L.longueur ;
}

// …
```

Listes : complexité

Implémentation contigüe

Accès à une position en **O(1)**

Insertion en fin de liste en **O(1)**

Insertion en tête de liste en **O(n)**

Insertion à une position quelconque en **O(n)**
Accès en O(1) + Insertion en O(n)

Concaténation de deux listes en **$O(n_2)$**
n_2 : longueur de la seconde liste

Recherche d'un élément donné en **O(n)**
ou **O(log n)** pour les listes triées
(recherche dichotomique)

Listes : complexité

Implémentation contigüe

Accès à une position en **O(1)**

Insertion en fin de liste en **O(1)**

Insertion en tête de liste en **O(n)**

Insertion à une position quelconque en **O(n)**
Accès en O(1) + Insertion en O(n)

Concaténation de deux listes en **$O(n_2)$**
n_2 : longueur de la seconde liste

Recherche d'un élément donné en **O(n)**
ou **O(log n)** pour les listes triées
(recherche dichotomique)

Implémentation par liste chaînée

Accès à une position en **O(n)**

Insertion en fin de liste en **O(n)**

Insertion en tête de liste en **O(1)**

Insertion à une position quelconque en **O(n)**
Accès en O(n) + Insertion en O(1)

Concaténation de deux listes en **$O(n_1)$**
n_1 : longueur de la première liste

Recherche d'un élément donné en **O(n)**
O(n) également pour les listes triées

Listes : résumé

Implémentation contigüe

Limitation de la taille

Accès **direct** aux éléments

Insertion ou suppression **en fin**
en temps constant

Insertion ou suppression ailleurs
de complexité linéaire

Recherche d'un élément
de complexité logarithmique
s'il s'agit d'une liste triée

Implémentation par liste chaînée

Pas de limitation de la taille

Accès **séquentiel** aux éléments

Insertion ou suppression **en tête**
en temps constant

Insertion ou suppression ailleurs
de complexité linéaire

Recherche d'un élément de complexité linéaire