# V. Structures de données séquentielles

1. Listes
   1.1 Représentation des listes
      1.1.a Représentation contiguë
      1.1.b Représentation chaînée
2. Piles
   1.2 Représentation des piles
      1.2.a Représentation contiguë
      1.2.b Représentation chaînée
3. Files
   1.3 Représentation des files
      1.3.a Représentation contiguë
      1.3.b Représentation chaînée
4. Ensembles
   1.4 Représentation des ensembles
      1.4.a Représentation par des tableaux de booléens

## 1. Listes

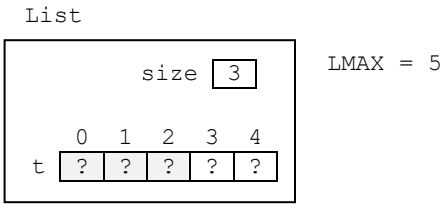
Une liste linéaire est une suite finie, éventuellement vide, d'éléments repérés selon leur rang dans la liste. L'ordre des éléments dans une liste est fondamental. Cependant, ce n'est pas un ordre sur les éléments, mais un ordre sur les places des éléments.

### 1.1. Représentation des listes

#### Représentation contiguë

```
const int LMAX = _;

struct List{
       int size;
       element t[LMAX];
};
```




```
void deleteItem(List& l, int k){
       int size = l.size;

       if (k < size){
              for(int i = k; i < size - 1; i++){ l.t[i] = l.t[i + 1]; }
              l.size--;
       }
}
```

ou

```
void deleteItem(List& l, int k){
       int size = l.size;

       if (k < size){
              for(int i = k + 1; i < size; i++){ l.t[i - 1] = l.t[i]; }
              l.size--;
       }
}
```

```
void addItem(List& l, int k, element x){
       int size = l.size;
       if (size < LMAX && k <= size){
              for (int i = size - 1; i >= k; i++){ l.t[i + 1] = l.t[i]; }

              l.t[k] = x;
              l.size++;
       }
}
```

ou

```
void addItem(List& l, int k, element x){
	int size = l.size;
	if (size < LMAX && k <= size){
		for (int i = size; i > k; i++){ l.t[i] = l.t[i - 1]; }

		l.t[k] = x;
		l.size++;
	}
}
```

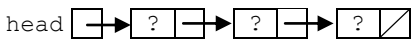
**Remarques**

1. La taille (longueur) de la liste ne peut pas dépasser la taille du tableau qui est surdimensionné.
2. L'accès et le parcours sont très efficaces, mais les adjonctions ailleurs qu'en fin de liste sont coûteuses, et il faut savoir majorer la taille des listes.

**Représentation chaînée**

On utilise des pointeurs pour chaîner entre eux les éléments successifs. La liste est alors déterminée par l'adresse de son premier élément.

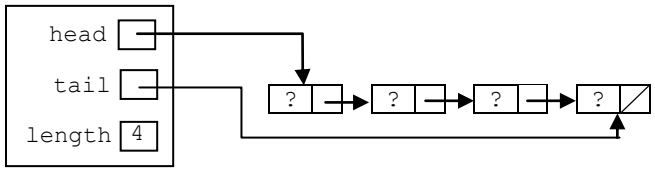
```
typedef struct cell* List;
struct cell{
	element value;
	List link;
};
```




ou

```
typedef struct cell* ptr;
struct cell{
	element value;
	ptr link;
};

struct List{
	ptr head;
	ptr tail;
	int length;
};
```




**Remarque :** La liste vide est représentée par le pointeur `NULL`.

**Avantages**

- La longueur `LMAX` n'est pas imposée.
- Les opérations de traitement sont faciles : insertion, parcours séquentiel, suppression, concaténation.

**Inconvénients**

- Calcul de la longueur nécessite un parcours de toute la liste.
- Il faut prévoir une place supplémentaire pour les pointeurs.
- L'accès au kième élément n'est plus direct.

```
int length(List l){
	int count = 0;
	while (l != NULL){
		count++;
		l = l->link;
	}
	return count;
}
```

**Recherche du kième élément dans une liste**

```
void ieme(List l, int k, element& x, bool& found){
	while (l != NULL && k != 0){
		k--;
		l = l->link;
	}

	if (l == NULL){ found = false; }
	else {
		found = true;
		x = l->value;
	}
}
```

Si `k` est négatif , la fonction ieme() retourne `found = false`. Mais l'inconvénient est que la liste sera parcouru totalement.

**Autres solutions à discuter**

```
void ieme(List l, int k, element& x, bool& found){
	while (l != NULL && k > 0){ // or k > 1
		k--;
		l = l->link;
	}

	if (l == NULL){ found = false; }
	else if (k == 0){ // do we need this condition
		found = true;
		x = l->value;
	}
}
```

```
void ieme(List l, int k, element& x, bool& found){
	while (l != NULL && k > 0){ // or k > 1
		k--;
		l = l->link;
	}

	if (l != NULL and k == 0){
		found = true;
		x = l->value;
	}
	else if (k == 0){ found = false; }
}
```

Pour résumer, si on a une collection d'objet de même nature avec la notion d'ordre, on utilise une liste.

## 2. Piles

Dans les piles, les insertions et les suppressions se font à une seule extrémité appelée sommet de la pile. Les piles sont appelées LIFO (Last-In First-Out).

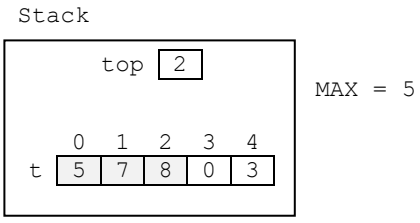
Les opérations sur les piles sont :
- Tester si une pile est vide.
- Accéder au sommet de la pile.
- Empiler un élément.
- Retirer l'élément se trouvant au sommet de la pile (dépiler).
- etc.

### 2.1 Représentation des piles

#### Représentation contiguë

```
const int MAX = _;

struct Stack{
       int top;
       element t[MAX];
};
```




```
void emptyStack(Stack&); // or void initializeStack(Stack&);
bool isEmpty(Stack);
bool isFull(Stack);
void push(Stack&, element);
void pop(Stack&); // or element pop(Stack&);
element top(Stack);
```

#### Implémentation des opérations

```
// Initialize stack to an empty state
void emptyStack(Stack& s){
       s.top = -1;
}

// Determine whether the stack is empty
bool isEmpty(Stack s){
       return s.top == -1;
}

// Determine whether the stack is full
bool isFull(Stack s){
       return s.top == MAX - 1;
}
```

```
// Add a new item to the stack
void push(Stack& s, element x){
	if (!isFull(s)){
		s.top++;
		s.t[s.top] = x;
	}
	else { cout << "Full stack!" << endl; }
}

// Remove the top element of the stack
void pop(Stack& s){
	if (!isEmpty(s)){ s.top--; }
	else { cout << "Empty stack!" << endl; }
}

//element pop(Stack& s){
//	if (!isEmpty(s)){
//		element value = s.t[s.top];
//		s.top--;
//		return value;
//	}
//	else { cout << "Empty stack!" << endl; }
//}
// Function must return a value even if the stack is empty.
// How to fix this?

// Return the top element of the stack
element top(Stack s){
	if (!isEmpty(s)){ return s.t[s.top]; }
	else { cout << "Empty Stack!" << endl; }
}
// Function must return a value even if the stack is empty.
// How to fix this?
```

**Représentation chaînée**

Les éléments de la pile sont chaînés entre eux.

```
struct node{
	element value;
	node* next;
};

typedef node* Stack;

void emptyStack(Stack&);
bool isEmpty(Stack);
void push(Stack&, element);
void pop(Stack&);
element top(Stack);
```

```
// Initialize stack to an empty state
void emptyStack(Stack& s){
	s = NULL;
}

// Determine whether the stack is empty
bool isEmpty(Stack s){
	return s == NULL;
}
```

```
// Add a new item to the stack
void push(Stack& s, element x){
       Stack p = new node;
       p->value = x;
       p->next = s;
       s = p;
}

void pop(Stack& s){
       if (!isEmpty(s)){
              Stack p = s;
              s = s->next;
              delete p;
       }
       else { cout << "Empty stack!" << endl; }
}

// Return the top element of the stack
element top(Stack s){
       if (!isEmpty(s)){ return s->value; }
       else { cout << "Empty Stack!" << endl; }
}
// Function must return a value even if the stack is empty.
// How to fix this?
```

## 3. Files

Dans une file, on fait les adjonctions à une extrémité, les accès et les suppressions à l'autre extrémité. L'élément présent depuis longtemps est le premier. Les files sont appelées FIFO (First-In First-Out).

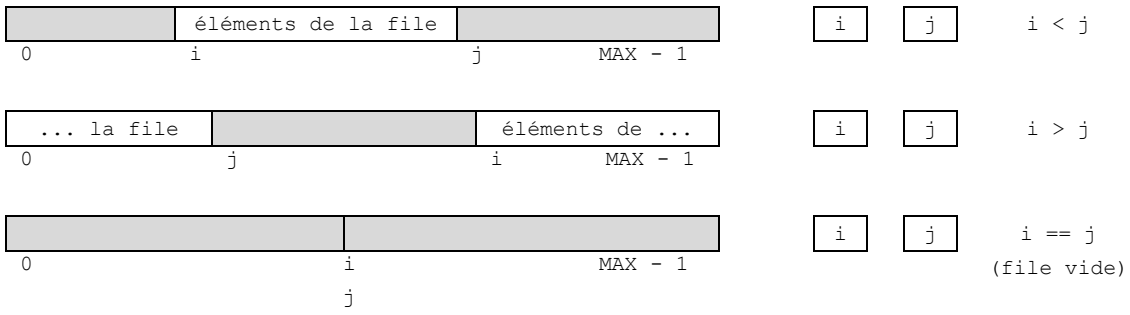
Les opérations sur les files sont :

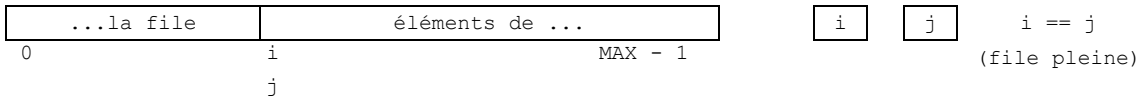
- Tester si une file est vide.
- Accéder au premier élément d'une file.
- Ajouter un élément dans la file.
- Retirer le premier élément de la file.
- etc.

### 3.1 Représentation des files

#### Représentation contiguë

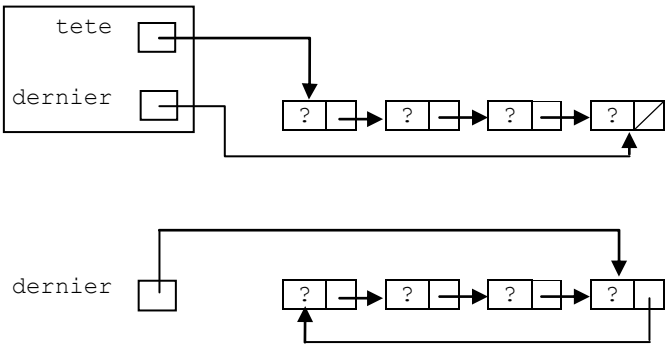
On conserve l'indice `i` du premier élément et l'indice `j` de la première case libre après la file. Ces indices progressent modulo la taille du tableau.






**Représentation chaînée**

Soit on utilise deux pointeurs tête et dernier, soit on utilise le pointeur du dernier élément pour repérer le premier élément (représentation circulaire).







L'implémentation des files (représentation contiguë et chaînée) sera faite en TDs.

## 4. Ensembles

Dans les ensembles, l'ordre des éléments n'a aucune importance. La propriété importante est la présence ou l'absence des éléments.

Les opérations sur les ensembles sont :

- Tester l'appartenance d'un élément à un ensemble.
- Ajouter un élément.
- Supprimer un élément.
- Tester si un ensemble est vide.
- etc.

Lorsqu'on considère des répétitions d'un élément, on parle alors d'ensemble avec répétitions ou multi-ensembles.

### 4.1 Représentation des ensembles

**Représentation par des tableaux de booléens**

Lorsqu'on manipule les ensembles dont les éléments possibles sont en nombre fini `n`, il est possible de représenter tout ensemble par un tableau de `n` booléens.

Nous abordons les ensembles avec plus de détails en TDs.