Liste

C’est quoi une liste lineaire ???

Une liste linéaire (LL) est la représentation informatique d’un ensemble fini, de taille variable et éventuellement nul, d’éléments de type T.

Pour la SD pile : les adjonctions (empiler), les suppressions (depiler) et les recherches (dernier) sont faites par rapport au sommet

On dit que la SD pile est une structure à un seul point d’accès

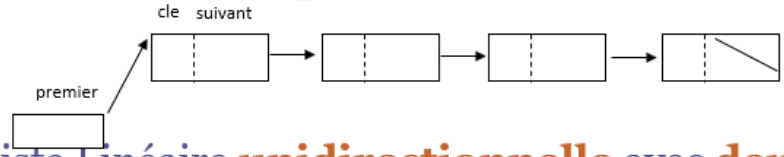
Pour la SD file : les adjonctions (enfiler) sont faites par rapport au queue, les suppressions (défiler) et les recherches (premier) sont faites par rapport à la tête

On dit que la SD file est une structure à deux points d’accès

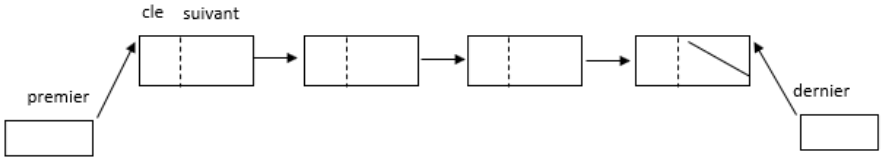
Pour la SD LL : les adjonctions, les suppressions et les recherches ne sont pas faites systématiquement ni par rapport à tête, ni par rapport à queue.

Variantes de la SD LL

❶Liste Linéaire **unidirectionnelle** avec **un seul point d'entrée** (premier)

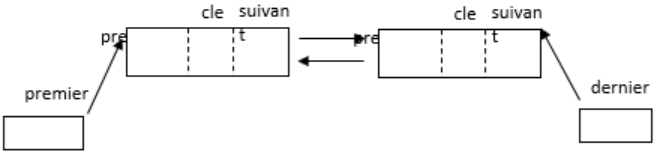


❷Liste Linéaire **unidirectionnelle** avec **deux points d'entrée** (premier et dernier)



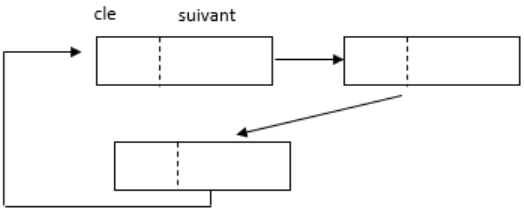
Pour les deux variantes (1) et (2), la liste linéaire est unidirectionnelle
À partir d'un élément donné on peut passer à son successeur.
Ceci est possible grâce au champ de chaînage suivant
dans la variante (1), le premier élément est privilégié (accès direct)
dans la variante (2) le premier et le dernier élément sont privilégiés (accès directe)

❸Liste Linéaire **bidirectionnelle** avec **deux points d'entrée** (premier et dernier)



À partir d'un élément donné, on peut passer soit à son successeur soit à son prédécesseur

❹Liste Linéaire **circulaire ou anneau**



Les notions de premier et dernier disparaissent
ces notions n'ont pas de sens dans un anneau.
Un anneau est doté uniquement d'un point d'entrée quelconque

➔ Imlpementation:

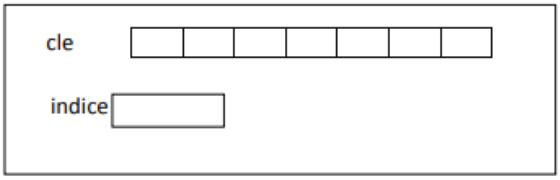
Liste Lineaire :

SD Liste Linéaire

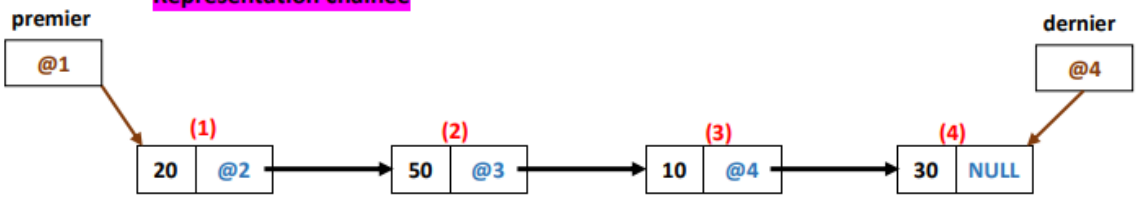
Représentation physique

Représentation contiguë

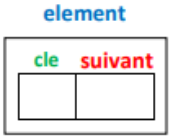
```
#define n 100
struct liste
{
   int cle[n];
   unsigned indice;
};
```



Représentation chainéé




```
typedef struct element
{
        int cle;
        struct element*suivant;
}element;
```




```
typedef struct liste
{
        element* premier;
        element* dernier;
}liste;
```

ListeLinéaire.h

```
/*repr?sentation physique chain?e*/

struct element
{
    int cle;
    struct element*suivant;
};


struct liste
{
    struct element* premier;
    struct element* dernier;
};

/*services ? exporter*/

void creer_liste(struct liste*);  /*service de cr?ation*/
unsigned liste_vide(struct liste); /*service de consultation*/
void ajout_entete(int,struct liste*); /*service de modification*/
void ajout_enqueue(int,struct liste*);/*service de modification*/
void ajouter_apres_ele_ref(int info,struct element* ref,struct liste* l);/*servic
e de modification*/
void supprimer_ele_ref(struct element* ref,struct liste* l);/*service de suppress
ion*/
struct element * chercher(int, struct element*);/*service de recherche*/
void parcours(struct element*,void(*oper)(struct element*));/*service de parcours
*/
void afficher_premier_vers_dernier(struct liste1 l);/*service d'affichage*/
```

ListeLinéaire.c

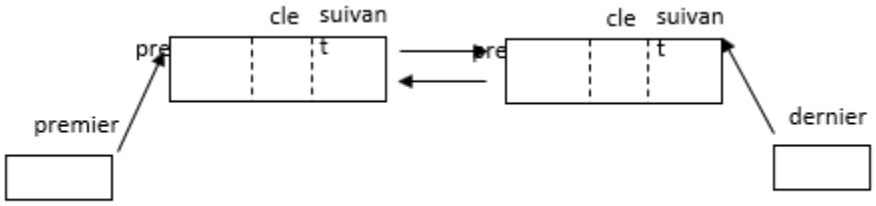
```c
#include<stdlib.h>
#include<assert.h>
#include "liste.h"

void creer_liste(struct liste* ll)
{
    assert(ll!=NULL);
    ll->premier=NULL;
    ll->dernier=NULL;
}

unsigned liste_vide(struct liste ll)
{
    return ll.premier==NULL && ll.dernier==NULL;
}

void ajout_entete(int info,struct liste* ll)
{
    struct element *q;
    assert(ll!=NULL);
    q=(struct element*)malloc(sizeof(struct element));
    assert(q!=NULL);
    q->cle=info;
    q->suivant=ll->premier;
    ll->premier=q;
    if(ll->dernier==NULL)/*if (liste_vide(*ll))*/
        ll->dernier=q;
}

void ajout_enqueue(int info, struct liste *ll)
{
    struct element*q;
    assert(ll!=NULL);
    q=(struct element*)malloc(sizeof(struct element));
    assert(q!=NULL);
    q->cle=info;
    q->suivant=NULL;
    if(liste_vide(*ll))
    {
        ll->premier=q;
    }
    else
    {
```

```
        ll->dernier->suivant=q;
    }
    ll->dernier=q;
}

void ajouter_apres_ele_ref(int info,struct element* ref,struct liste* l)
{
    struct element*e,*f;

    assert(l!=NULL);
    assert(ref!=NULL);
    assert(!liste_vide(*l));
    e=(struct element*)malloc(sizeof(struct element));
    f=(struct element*)malloc(sizeof(struct element));
    assert(f!=NULL);
    assert(e!=NULL);
    e->cle=info;
    e->suivant=f;
    if(ref!=l->dernier)
        ref->suivant=e;
        e->suivant=f;
    else
        l->dernier=e;
        e->suivant=NULL;
}

void supprimer_ele_ref(struct element* ref,struct liste* l)
{
    struct element*e;

    e=(struct element*)malloc(sizeof(struct element));
    struct element*f;
    f=(struct element*)malloc(sizeof(struct element));
    assert(ref!=NULL);
    assert(l!=NULL);
    e->suivant=ref;
    ref->suivant=f;
    if(ref->suivant!=NULL)
        e->suivant=f;
    else
    {
        l->dernier=e->suivant;
    }
    free(ref);
}
```

```c
struct element * chercher(int info, struct element*p)
{
    while(p&&(p->cle!=info))
        /*l'ordre des deux sous expressions est significatif*/
    p=p->suivant ;
        /*passer à l'élément suivant*/
        /*à la sortie de la boucle (!p) || (p->cle)==info
          échec : !p => p== NULL
          succès :p->cle==info*/
    return(p) ;
}

void parcours(struct element*p,void(*oper)(struct element*))
{
    while(p)
    {
    /*appliquer à l'élément porté par p le traitement est fourni par oper*/
        (*oper)(p) ;
    /*passer à l'élément suivant */
        p=p->suivant ;
       }
}

// struct element * point_de_depart ;
// void afficher(struct element * q)
// {
//      printf (''%u d\n '', q->cle) ;
// }
// /* activation de parcours */
// parcours(point_de_depart, afficher) ;

// struct element *point_de_depart ;
// void incrementer (struct element *q)
// {
//     q->cle++ ;
// }
// /*activation */
// parcours(point_de_depart, incrementer) ;

void afficher_premier_vers_dernier(struct liste1 l)
{
    struct element*e;
    e=l.premier;
    while(e)
    {
```

```
        printf("%d\n",e->cle);
        e=e->suivant;
    }
}
```

Liste Bidirectionnelle :



ListeBidirectionnelle.h

```
/*repr?sentation physique*/
typedef struct element
{
int cle;
    struct element *precedent;
    struct element *suivant;
}element;

typedef struct listeBi
{   element* premier;
    element*dernier;
}listeBi;

/*services ? exporter*/

void creer_liste(listeBi*);  /*service de c?reation*/
unsigned liste_vide(listeBi); /*service de consultation*/

void ajouter_entete(int,listeBi*); /*service de modification*/
void ajouter_enqueue(int,listeBi*); /*service de modification*/
void ajouter_apres_ele_ref(int,element*,listeBi*);/*service de modification*/
void supprimer_ele_ref(element*,listeBi*);/*service de modification*/
void tri_insertion(listeBi*);/*service de modification*/
```

```
//////////////////////////////////
void afficher_premier_vers_dernier(listeBi); /*service de consultation*/
void afficher_dernier_vers_premier(listeBi); /*service de consultation*/
element* rechercher(int, listeBi); /*service de consultation*/
```

ListeBidirectionnelle.c

```
#include"ListeBidirectionnelle.h"
#include<stdlib.h>
#include<assert.h>
#include<stdio.h>


void creer_liste(listeBi* l)
{
    assert(l!=NULL);
    l->premier=NULL;
    l->dernier=NULL;
}

unsigned liste_vide(listeBi l)
{
    return l.premier==NULL && l.dernier==NULL;
}

void ajouter_entete(int info,listeBi* l)
{
    element* e;
    assert(l!=NULL);
    e=(element*)malloc(sizeof(element));
    assert(e!=NULL);
    e->cle=info;
    e->suivant=l->premier;
    e->precedent=NULL;
    if(l->premier!=NULL)/*la liste n'est pas vide*/
        l->premier->precedent=e;
    else /* il s'agit du premier l ment  ajouter dans une liste vide*/
        l->dernier=e;

    l->premier=e;
}
```

```c
void ajouter_enqueue(int info,listeBi* l)
{
    element* e;
    assert(l!=NULL);
    e=(element*)malloc(sizeof(element));
    assert(e!=NULL);
    e->cle=info;
    e->suivant=NULL;
    e->precedent=l->dernier;
    if(l->dernier!=NULL)/*la liste n'est pas vide*/
        l->dernier->suivant=e;
    else /*on va ajouter le premier l m ent dans une liste vide*/
        l->premier=e;
    l->dernier=e;
}

void ajouter_apres_ele_ref(int info,element* ref,listeBi* l)
{
    element*e;
    assert(l!=NULL);
    assert(ref!=NULL);
    assert(!liste_vide(*l));
    e=(element*)malloc(sizeof(element));
    assert(e!=NULL);
    e->cle=info;
    e->suivant=ref->suivant;
    e->precedent=ref;
    if(ref!=l->dernier)
        ref->suivant->precedent=e;
    else
        l->dernier=e;
    ref->suivant=e;
}

void supprimer_ele_ref(element* ref,listeBi* l)
{
    assert(ref!=NULL);
    assert(l!=NULL);
    if(ref->precedent!=NULL)
        ref->precedent->suivant=ref->suivant;
    else
    {
        l->premier=ref->suivant;
    }
    if(ref->suivant!=NULL)
```

```
        ref->suivant->precedent=ref->precedent;
    else
    {
        l->dernier=ref->precedent;
    }
    free(ref);
}

void tri_insertion(listeBi* l)
{
    int v;
    element *p,*q;
    assert(l!=NULL);
    assert(!liste_vide(*l));
    p=l->premier->suivant;
    while(p) /* p!=NULL*/
    {
        v=p->cle;
        q=p;
        while(q->precedent!=NULL && q->precedent->cle>v)
        {
            q->cle=q->precedent->cle;
            q=q->precedent;
        }
        q->cle=v;
        p=p->suivant;
    }
}

////////////////////////////////////////
void afficher_premier_vers_dernier(listeBi l)
{
    element*e;
    e=l.premier;
    while(e)
    {
        printf("%d\n",e->cle);
        e=e->suivant;
    }
}
void afficher_dernier_vers_premier(listeBi l)
{
    element*e;
    e=l.dernier;
    while(e)
```

```
    {
        printf("%d\n",e->cle);
        e=e->precedent;
    }
}

element* rechercher(int info, listeBi l)
{
    element *e;
    e=l.premier;
    while(e && e->cle!=info)
        e=e->suivant;
    return e;
}
```