Listes linéaires

Définition

Une **liste linéaire** est une suite finie éventuellement vide d'éléments repérés par leur place dans la liste.

Exemples: l1=Vide; l2=<3, 5,1,7,1,3,14,-1,0,3>

On peut noter qu'un même nombre peut apparaître plusieurs fois à des positions différentes dans la liste.

Type abstrait de données Liste

Définition itérative

Sortes Liste, Place

Utilise: Booléen, Element, Entier

Opérations:

- liste_vide: -> Liste
- supprimer: Listex Entier-> Liste
- ajouter: Element x Liste x Entier -> Liste
- Rechercher : Element x Liste -> Place
- succ : Place -> Place
- PlaceImpossible: -> Place // la place du successeur du dernier élément
- acces: Liste x Entier -> Place
- contenu: Place -> Element
- ième: Liste x Entier -> Element
- longueur: Liste -> Entier

Propriétés:

- supprimer(L,n) a un sens si $1 \leqslant n \leqslant longueur(L)$
- ajouter(e,L,n) a un sens si $1 \leqslant n \leqslant longueur(L)+1$ (pour permettre l'ajout en fin de liste)
- succ(p) a un sens si $p \neq PlaceImpossible$
- ième(L,n) a un sens si $1 \leqslant n \leqslant longueur(L)$
- contenu(p) a un sens si $p \neq PlaceImpossible$
- acces(L,n)=PlaceImpossible si n > longueur(L)
- succ(acces(L,n))=acces(L, n+1)
- ieme(L,n)=contenu(acces(L,n))
- longueur(listeVide)=0
- longueur(supprimer(e,L,k))=longueur(L)-1 si on est dans le domaine de def des opérations
- longueur(ajouter(L,k))=1+longueur(L) si on est dans le domaine de def des opérations
- ieme(supprimer(L,k),i)=ieme(L,k) si i<k // rien n'a bougé avant l'endroit de la suppression

- ieme(supprimer(L,k),i)=ieme(L,i+1) si k ≤ i // après, tout a été décalé d'un cran à gauche
- ieme(ajouter(e,L,k),i)=ieme(L,k) si i<k // rien n'a bougé avant l'endroit de l'ajout
- ieme(ajouter(e,L,k),i)=ieme(L,i-1) si k < i // après, tout a été décalé d'un cran à droite
- ieme(ajouter(e,L,k),k)=e
- si ieme(L,i)=e et ieme(L,j) différent de e pour j<i alors recherche(L,i)=acces(L,i)
- si ieme(L,j) différent de e pour tout j alors recherche(L,i)=PlaceInvalide

Toute liste non vide peut être construit en ajoutant ses éléments à la liste vide. Un ensemble est donc soit vide, soit de la forme ajouter(e, E).

Définition récursive

Sortes Liste, Place

Utilise: Booléen, Element, Entier

Opérations:

- liste_vide: -> Liste
- cons: Element x Liste -> Liste
- tete: Liste -> Place
- queue: Liste -> Liste
- Rechercher : Element x Liste -> Place
- succ : Place -> Place
- PlaceImpossible: -> Place // la place du successeur du dernier élément
- contenu: Place -> Element

Propriétés:

Toute liste non vide peut être construit en ajoutant ses éléments à la liste vide. Une liste L' est donc soit vide, soit de la forme cons(e, L).

- tete(L) a un sens si L est non vide
- queue a un sens si L est non vide
- succ(p) a un sens si $p \neq PlaceImpossible$
- contenu(p) a un sens si $p \neq PlaceImpossible$
- succ(tete(L))=tete(queue(L))
- contenu(tete(cons(e,L)))=e
- queue(cons(e,L))=L
- Rechercher(e,listeVide)=PlaceImpossible
- Rechercher(e, cons(e,L))=tete(L)
- rechercher(e, cons(f,L))=rechercher(e, L) si e et f sont différents

On peut montrer que ces deux définitions sont équivalentes et définir les opérations de l'une avec celle de l'autre et vice-versa.

Implantation à l'aide de tableaux (contiguë)

```
typedef struct ttableau
{
  int tailleMax;
  int taille;
  int * tab;
} tableau;
////////////////////////////////////////////////////////////////////////////
//  Nom :       creeTableau
//
//  Arguments : le nombre max d'élément du tableau (>= 0)
//
//  retourne :  un tableau vide pouvant contenir au plus n éléments
//
tableau creeTableau(int n)
{
  tableau res;
  assert(n>=0);
  res.tab=malloc(sizeof(int)*n);
  assert(res.tab != NULL);
  res.tailleMax = n;
  res.taille=0;
  return res;
}
////////////////////////////////////////////////////////////////////////////
//  Nom :       libereTableau
//
//  Arguments : un tableau
//
//  libere la mémoire réservée pour t.tab
//  on met taillemax à 0 par prudence
void libereTableau(tableau * t)
{
  free(t->tab);
  t->tailleMax=0;
}
////////////////////////////////////////////////////////////////////////////
//  Nom :       ajouter
//
//  Arguments : un élément e, l'adresse d'un tableau pt, un indice i
//
//  ajoute l'élément e au tableau désigné par pt à la position i
//
void ajouter(int e, tableau * pt, int i)
```



```
{
  int cb;
  assert(pt-> taille < pt->tailleMax); // mauvais gestion des débordements
  // on décale tout d'un cran vers la droite
  // à noter: le sens de parcours de cb
  for (cb=pt->taille; cb > i; cb --)
    pt->tab[cb]=pt->tab[cb-1];
  pt->tab[i]=e;
  pt->taille ++;
}
////////////////////////////////////////////////////////////////////////////
//  Nom :       supprimer
//
//  Arguments : l'adresse d'un tableau pt, un indice i
//
//  supprime  l'élément d'indice i du tableau désigné par pt
//
void supprimer(tableau * pt, int i)
{
  int cb;
  assert(i>=0);
  assert(i<pt->taille);
  pt->taille --;
  // on décale tout d'un cran vers la gauche
  // à noter: le sens de parcours de cb (normal)
  for (cb=i; cb <pt->taille; cb ++)
    pt->tab[cb]=pt->tab[cb+1];
}
int longueur(tableau t)
{
  return t.taille;
}

int ieme(tableau t, int i)
{
  assert(i>=0);
  assert(i<t.taille);
  return(t.tab[i]);
}
////////////////////////////////////////////////////////////////////////////
//  Nom :       rechercher
//
//  Arguments :
//
//  retourne :  -1 si e n'est pas dans t, la place de e sinon
```

```
//
int rechercher(int e, tableau t)
{
    int cb=0;
    while ((cb < t.taille) && (t.tab[cb] != e))
      cb ++;
    if (cb == t.taille)
      return -1;
    return cb;
}
```

Implantation à l'aide de listes chaînées (chaînée)

```
typedef struct telement {
  int info;
  struct telement * suivant;
} element;

typedef  element * liste;


liste cree_element(int e, liste suite){
  liste ltmp;
  ltmp=malloc(sizeof(element));
  ltmp->suivant=suite;
  ltmp->info=e;
  return ltmp;
}

liste ajoute_n(int e, liste l,int place){
  liste ltmp=l;
  int cb;
  if (place == 1){
    ltmp= cree_element(e, l);
  }
  else {
   assert(l != NULL);
    for (cb=1; (cb < place -1) && (l != NULL); cb ++){
      l=l->suivant;
    }
    assert(l != NULL); // teste en fait place <= longueur(l)
    l->suivant=cree_element(e,l->suivant);
  }
  return ltmp;
}
```



```
liste supprime_n(liste l,int place){
  liste ltmp=l;
  int cb;
  assert(l != NULL);
  assert(place >=1);
  if (place == 1){
    l=l->suivant;
    free(ltmp);
  }
  else {
    for (cb=1; (cb < place -1) && (l != NULL); cb ++){
      l=l->suivant;
   }
    assert(l != NULL); // teste en fait place <= longueur(l)
    ltmp=l->suivant;
    assert(ltmp != NULL);
    l->suivant=ltmp->suivant;
    free(ltmp);
  }
  return l;
}
void affiche_liste(liste l ){
  if (l != NULL){
    printf("%i\n",l->info);
    affiche_liste(l->suivant);
  }
}

int  main (void){
  liste l1=NULL;
  l1=ajoute_n(1,l1,1);
  l1=ajoute_n(2,l1,1);
  l1=ajoute_n(4,l1,1);
  l1=ajoute_n(3,l1,2);
  printf("affichage de la liste:\n");
  affiche_liste(l1);
  l1=supprime_n(l1, 2);
  printf("affichage de la liste:\n");
  affiche_liste(l1);

  return EXIT_SUCCESS;
 }
```