Problème 3, skip liste

(Skip liste : une structure pour les ensembles ou les tableaux associatifs)

Il existe plusieurs structures de données qui permettent de stocker des éléments ou des relations et fournissent des fonctions rapides pour les recherches/insertions/suppressions. Ces structures sont utiles lorsqu'on veut représenter des ensembles $X = \{x_i\}_{i=1...n}$ ou des tableaux associatifs $T = \{(x_i, y_i)\}_{i=1...n}$, autrement dit des dictionnaires à la python. Les plus connus sont les tables de hachage et les arbres rouge/noir.

Une *skip liste* (inventée en 1990 par W. Pugh) est une structure de données probabiliste, basée sur des couches superposées de listes chaînées, dont les éléments sont croissants.

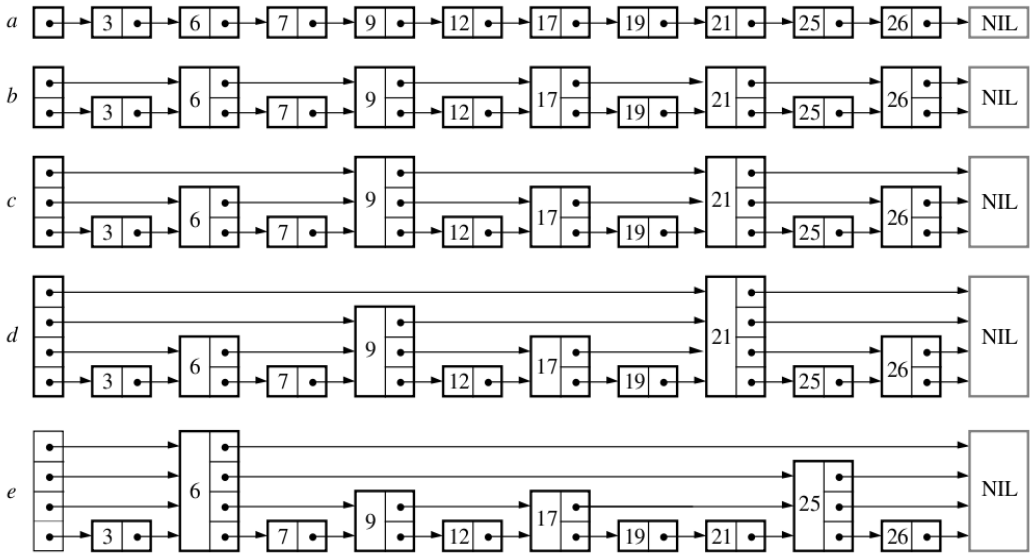

Figure 1: Principe de la skip liste.

La Figure 1 montre l'idée sous-jacente à la skip-liste. Observons sur la Figure 1a une liste chaînée standard, dont les éléments sont triés. Rechercher un élément nécessite de traverser les n éléments de la liste. Si maintenant un sommet sur deux contient un pointeur vers son deuxième successeur (Figure 1b), alors une recherche ne nécessite plus que de traverser au plus $n/2 + 1$ éléments. En poursuivant le raisonnement, un sommet sur 4 peut avoir un pointeur sur son quatrième successeur et la recherche ne nécessite plus que de traverser $n/4+2$ sommets (Figure 1c). Si chaque 2^i-ème sommet a un pointeur vers le sommet 2^i plus loin, alors la recherche ne nécessite plus la visite que de $\log_2 n$ sommets (Figure 1d).

C'est efficace pour la recherche, mais catastrophique pour l'insertion ou la suppression. En effet, un élément ajouté décale les indices de tous les suivants, et il faudrait recalculer de nombreux pointeurs. L'idée majeur de la skip liste, est de ne pas spécifier **où** sont les sommets qui ont beaucoup de pointeurs, mais de seulement préciser une **proportion** de sommets qui ont beaucoup de pointeurs.

Ainsi, environ un sommet sur deux auront deux pointeurs, un sommet sur quatre auront trois pointeurs, un sommet sur huit auront quatre pointeurs, etc (Figure 1e). De plus, un sommet a trois pointeurs ne pointera pas sur son quatrième successeur, mais sur le prochain sommet qui a (au moins) trois pointeurs. Le nombre de pointeurs d'un sommet (on parle de *niveau*) sera tiré aléatoirement selon une loi qui respecte la proportion indiquée.

On se propose dans la suite de créer une structure de données en C pour les skip-listes, ainsi que quelques algorithmes.

Question 1. Nombre de pointeurs dans une skip-liste

On suppose que tout sommet a au moins un pointeur, un sur deux a au moins deux pointeurs, un sur quatre a au moins trois pointeurs, etc (cas de la Figure 1d). Combien y aura-t-il de pointeurs en tout au maximum ? (le nombre d'éléments n est une puissance de 2.)

```
On a d'abord n pointeurs (niveau 1), puis n/2 pointeurs (niveau
2), puis n/4 ... soit en tout n*(1 + 1/2 + 1/4 + 1/8 + ...) <= 2*n.
```

Question 2. Pourquoi une skip-liste ?

Que permet de modéliser efficacement une telle structure de données ?

```
Un ensemble dont les éléments sont munis d'une relation d'ordre,
ou alors un tableau associatif dont les clés sont munies d'une
relation d'ordre.
```

Question 3. Modélisation de chaque cellule d'une skip-liste

Il s'agit d'abord de modéliser chaque élément dans la skip-liste. Un élément est créé avec un niveau donné (nb de pointeurs) et une valeur donnée. Ecrire la `struct` C correspondante (appelée `Element`) ainsi que la fonction en créant une :

```
/* retourne un élément alloué dynamiquement. */
Element* CreeElement( int valeur, int niveau );
```

N'oubliez pas d'initialiser à `null` les pointeurs.

```
typedef struct SElement {
  int val;
  int niv;
  struct SELement** ptr;
} Element;
Element* CreeElement( int valeur, int niveau )
{
  Element* e = (Element*) malloc( sizeof(Element) );
  e->val = valeur;
  e->niv = niveau;
  e->ptr = (Element**) malloc( sizeof(Element*) * niveau );
  for (int i = 0; i < niveau; i++ ) e->ptr[ i ] = null;
  return e;
}
```

Question 4. Suivant d'une cellule

Ecrire la fonction qui retourne le suivant de degré i donné d'un élément, ou `null` s'il n'existe pas. Ecrivez de plus la fonction qui modifie le suivant de degré i d'un élément. (NB : i commence à 0.)

```
Element* Suivant( Element* e, int i);
void ModifieSuivant( Element* e, int i, Element* nouv_succ );

Element* Suivant( Element* e, int i)
{
  if ( i >= e->niv ) return null;
  return e->ptr[ i ];
}
void ModifieSuivant( Element* e, int i, Element* nouv_succ )
{
  if ( i < e->niv )
    e->ptr[ i ] = nouv_succ;
}
```

Question 5. Type skip-liste

Ecrivez maintenant la structure C `SkipListe` qui contient un élément initial (non valide, cf Figure) ainsi que le niveau maximal donné. Donnez aussi la fonction qui alloue dynamiquement une `SkipListe` initialisée à la liste vide.

```
SkipListe* CreeSkipListe( int niveau_max );
```

On note que `niveau_max` doit valoir à peu près $\log_2 n$, où n sera le nombre d'éléments maximal de la liste.

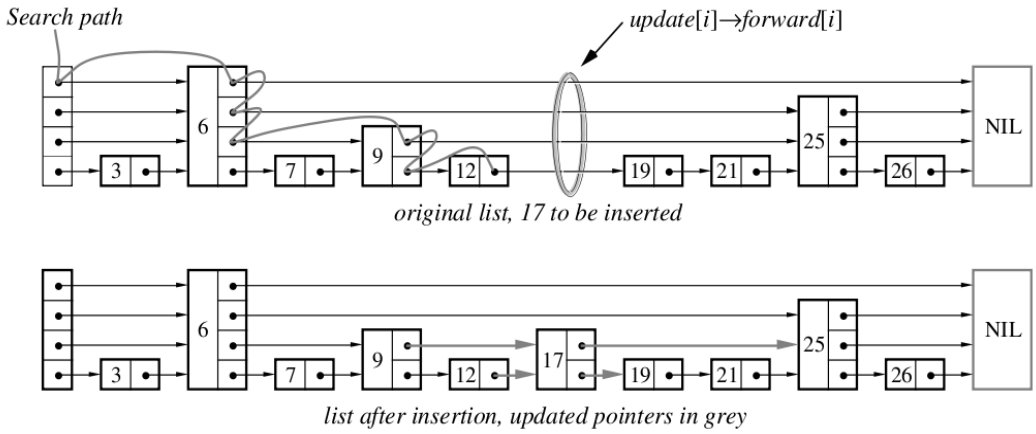

Figure 2: Principe de l'insertion dans une skip liste.

```
typedef struct {
  Element* start;
} SkipListe;
SkipListe* CreeSkipListe( int niveau_max )
{
  SkipListe* sl = (SkipListe*) malloc( sizeof(SkipListe) );
  /* l'element fictif contiendra le niveau max. */
  sl->start = CreeElement( 0, niveau_max );
}
```

Question 6. Recherche dans une skip-liste

En supposant que la skip-liste contient un nombre arbitraire d'éléments, écrire la fonction de recherche d'un élément dans la skip liste :

```
/* Retourne un pointeur vers l'élément si trouvé ou null sinon. */
Element* Cherche( SkipListe* sl, int valeur );

Element* Cherche( SkipListe* sl, int valeur )
{
  Element* cur = sl->start;
  for ( int i = sl->start->niv - 1; i >= 0; i-- )
    while ( ( Suivant( cur, i ) != null )
            && ( Suivant( cur, i )->val < valeur ) )
      cur = Suivant( cur, i );
  cur = Suivant( cur, 0 );
  if ( ( cur != 0 ) && ( cur->val == valeur ) )
    return cur;
  return null;
}
```

Question 7. Insertion dans une skip-liste

On se propose maintenant de réaliser l'insertion d'un nouvel élément dans une skip liste. Il y a donc une phase de recherche, puis un tirage aléatoire du niveau du nouvel élément et enfin une phase de modification de certains pointeurs (Figure 2).

On vous **donne** une fonction qui vous retourne aléatoirement un nouveau niveau pour un élément d'une skip-liste, en respectant les probabilités spécifiées.

```
/* Retourne un pointeur vers l'élément si trouvé ou null sinon. */
int NiveauAleat( SkipListe* sl );
```

Ecrivez maintenant la fonction d'insertion dans une skip-liste.

```
void Inserer( SkipListe* sl, int valeur );
```

On note qu'il faut maintenir dans un tableau les derniers pointeurs successeurs de chaque niveau, afin de mettre facilement à jour les pointeurs du nouvel élément.

```
void Inserer( SkipListe* sl, int valeur )
{
  Element* cur = sl->start;
  Element** mem = (Element**) malloc( sizeof(Element*) * cur->niv );
  for ( int i = sl->start->niv - 1; i >= 0; i-- )
    {
      while ( ( Suivant( cur, i ) != null )
              && ( Suivant( cur, i )->val < valeur ) )
        cur = Suivant( cur, i );
      mem[ i ] = cur;
    }
  cur = Suivant( cur, 0 );
  if ( ( cur != 0 ) && ( cur->val == valeur ) )
    return; /* l'element existe déjà. */
  Element* e = CreeElement( valeur, NiveauAleat( sl ) );
  for ( int i = e->niv - 1; i >= 0; i-- )
  {
    ModifieSuivant( e, i, Suivant( mem[ i ], e ) );
    ModifieSuivant( mem[ i ], i, e );
  }
}
```

Question 8. Pourquoi la recherche est-elle efficace ?

Dans les deux questions précédentes, pourquoi la recherche comme l'insertion sont-elles plus efficaces que dans une simple liste triée ? Donnez une indication informelle de la complexité moyenne de ces deux opérations.

Les pointeurs de niveau important permettent de faire des grands pas dans la liste, au contraire d'une liste chaînée où l'on ne peut avancer que de un par un. La complexité sera environ logarithmique en moyenne dans les deux cas (l'insertion ne coûte pas plus cher car elle est de complexité $niveau_{max}$*, donc égale à* $log_2 n$*).*

Question 9. Suppression

Ecrivez enfin la fonction de suppression d'un élément.