Structures de données

L'objectif de ce cours est de définir les types abstraits et les structures de données séquentielles les plus courantes.

1. Introduction

L'utilisation des tableaux se heurtent à plusieurs problèmes en langage C : les déclarations de tableaux statiques impliquent la connaissance a priori de la taille afin que le système d'exploitation puisse allouer la place mémoire requise pour manipuler les données. Les éléments du tableau sont stockés de manière contigüe. Lorsque la taille n'est pas connue a priori, l'utilisation tes tableaux statiques conduit souvent à surdimensionner leur taille. L'ajout d'éléments se fait alors dans la limite des places initialement allouées. La suppression d'éléments se fait en décalant le contenu des différentes cellules du tableau, ce qui peut s'avérer coûteux en termes de temps d'exécution.
La déclaration de tableaux dynamiques permet de contourner certains problèmes tels que la connaissance a priori de la taille du tableau mais est elle aussi limitée quant à la ré-allocation de places mémoire en cas de modification du tableau.
Même si l'avantage que constitue l'accès direct à l'information n'est pas à négliger dans le cas des tableaux, le stockage en mémoire reste un inconvénient qui peut freiner son utilisation notamment dans le cas de données de grandes dimensions.

Les types abstraits sont des structures de données particulières, définies par l'utilisateur. Elles résultent d'un cahier des charges permettant de décrire non seulement les données elles-mêmes (comme pour les structures en C) mais aussi leur comportement (opérations pouvant être effectuées sur le type). Comme les tableaux, il s'agit de collections d'enregistrements uniformes.

2. Les types abstraits

2.1. Introduction aux types abstraits

Lorsqu'un projet se créée en informatique, la question des données manipulées et de leur stockage se pose rapidement. Le cahier des charges permet de spécifier les structures de données adaptées au projet. Il constitue un contrat entre l'utilisateur et l'implanteur de la structure. De ce fait, il convient de définir de manière précise ce qui est attendu de cette structure.

Parfois, les types natifs, tels que les tableaux en C, les listes en Python, permettent de répondre aux attentes de l'utilisateur mais suivant les langages utilisés et suivant les exigences du cahier des charges, l'implanteur est parfois amené à définir sa propre structure de données, adaptée au projet. Il crée alors un type abstrait.

La spécification d'un type abstrait est indépendante de son implémentation dans un langage particulier. Elle consiste à décrire très précisément :

- Le nom de la structure ;
- Les données manipulées ;
- Les opérations pouvant être réalisées sur la structure. Pour ce faire, on décrit la signature et la sémantique de ces opérations. La manipulation des données se fera ensuite exclusivement grâce à ces opérations.

L'implanteur du type abstrait doit vérifier que sa structure répond aux spécifications du cahier des charges ; il doit donc définir les tests et les critères de certification, en collaboration avec l'utilisateur. Il doit aussi fournir une implémentation modulaire : les opérations des types abstraits restent des boîtes noires pour l'utilisateur, qui doit connaître les comportements possibles de la structure sans maîtriser les détails de la programmation. L'utilisateur n'accède qu'à une interface.

On appelle type concret le résultat de l'implémentation d'un type abstrait dans un langage.

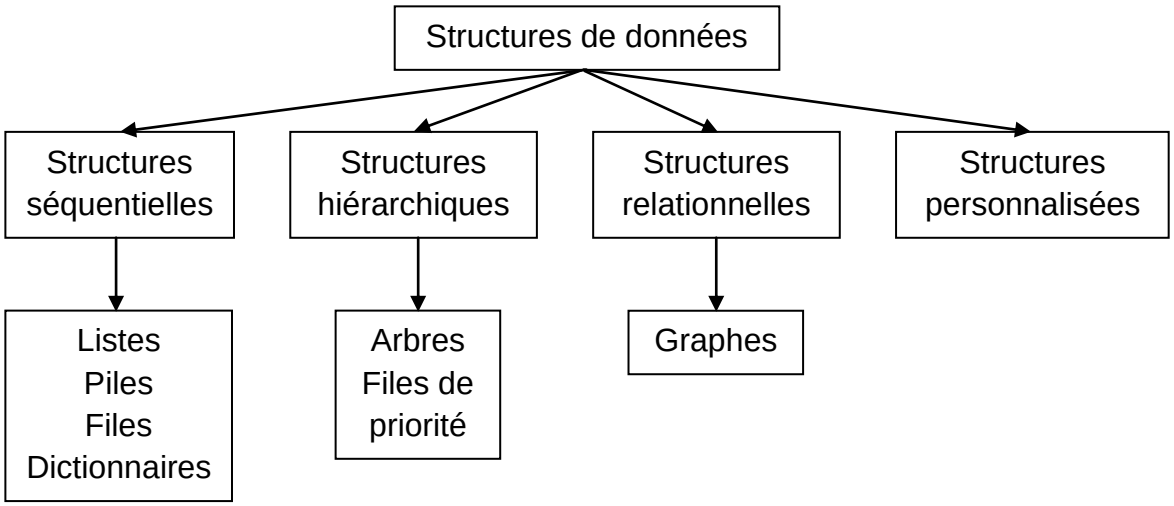
2.2. Opérations sur les types abstraits

On regroupe habituellement les opérations sur les types abstraits en trois familles.

Les opérations possibles sur un type abstrait T sont les opérations :

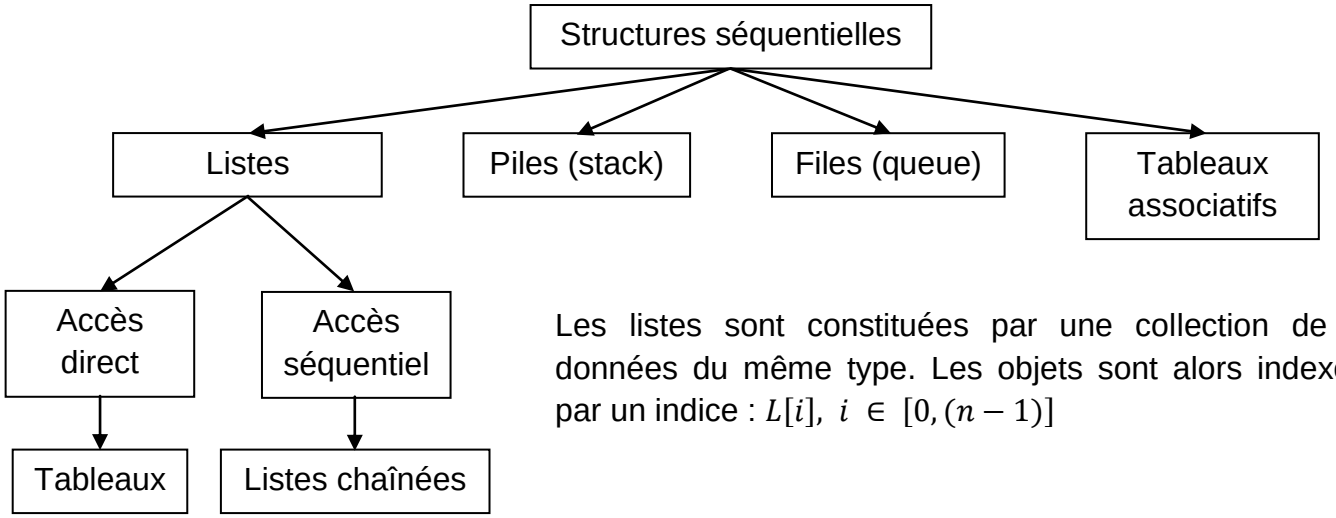
- d'initialisation : constructeur du type. T est le résultat de l'opération sans en être une entrée ;
- d'accès : accesseur. T est une entrée pour cette opération mais n'en constitue pas une sortie ;
- de transformation : transformateur. T est à la fois une entrée et une sortie pour cette opération.

2.3. Classification des structures de données



3. Les structures de données séquentielles

Les structures de données séquentielles sont constituées de séquences de données. L'accès aux données peut être direct, et c'est le cas des tableaux et des tables de hachage, ou séquentiel comme pour les listes chaînées, piles et les files.



Les listes sont constituées par une collection de n données du même type. Les objets sont alors indexés par un indice : $L[i],\ i \in [0,(n-1)]$

On distingue parmi les listes :

- Les tableaux : collection de données dans des espaces mémoires contigus, accès direct grâce à l'indice ;
- Les listes chaînées : collection de données dans des espaces mémoires non contigus, accès séquentiel.

Suivant les langages, ces types sont ou non déjà programmés, présents ou non dans des modules. Ainsi, en C, les tableaux sont des types de base, au même titre que les booléens, entiers et flottants. Les opérations sont déjà programmées (constructeur : `int tab[2] = {0, 1}`), accesseur (`tab[i]`) et transformateur (`tab[i] = 2`). Les autres types indiqués ci-dessus doivent être implémentés.

En Caml, ces différents types sont déjà programmés dans des modules, tout comme en python.

4. Les listes

4.1. Les tableaux

Les tableaux sont des collections d'éléments de même type, dont les valeurs sont mutables (i.e. dont les valeurs peuvent être modifiées). L'accès direct à un élément est garanti par la présence d'un indice, souvent numéroté à partir de 0, précisé entre crochets en algorithmique et dans la plupart des langages informatiques.

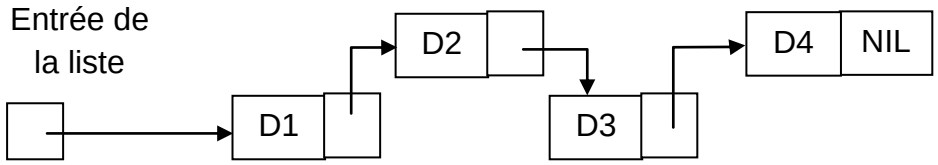
Indices	0	1	2	3	4	5	6	7
Tableau T			T [2]					

La taille d'un tableau correspond aux nombres d'éléments de ce tableau. Elle est souvent notée n.

4.2. Les listes chaînées

Les listes chaînées sont composées d'éléments de même type, nommé maillon ou nœud. Chaque maillon contient :

- Les informations (structure de données, entiers, …)
- Un lien vers l'élément suivant



Afin de faciliter la manipulation, une entrée de liste (interface avec la liste chaînée) est souvent créée : ce nœud particulier ne contient aucune information mais uniquement le lien vers le premier élément de la liste chaînée.

Lorsque la liste chaînée est vide, l'adresse indiquée dans l'interface est NIL (NULL en C).

Si la présence d'un lien vers le maillon suivant permet de gagner en souplesse d'un point de vue allocation mémoire, les listes chaînées présentent néanmoins deux inconvénients. Le premier vient du fait que le lien vers l'élément suivant prend de la place en mémoire. Le second est la perte de l'accès direct à une information. En effet, contrairement au tableau, la taille d'une liste chaînée n'est pas connue a priori. Il faut alors la parcourir à la recherche de la donnée demandée ou jusqu'à l'élément situé à la position demandée, voire dans son intégralité si la donnée en est absente ou si la position est supérieure au nombre de maillons constituant la liste chaînée. L'accès à un élément est donc, dans le pire des cas en $O(n)$, si n est le nombre d'éléments stockée dans la liste chaînée.

Spécification d'une liste chaînée

Nom : liste chaînée

Donnée manipulées :

- Information
- Adresse de l'élément suivant : pointeur ou référence suivant les langages (remarque : un pointeur est noté ^ en algorithmique).

Opérations :

- Constructeur : initialisation de l'interface.
- Accesseur : accède à un élément à une position donnée
- Transformateurs :
    - Insertion d'un élément à une position donnée
    - Suppression d'un élément à une position donnée

Description des opérations

Constructeur : initialisation

Argument : aucun argument
Retourne : pointeur sur NIL
Sémantique :

- Crée le premier nœud de la liste (interface)
- Initialise l'entrée comme un pointeur sur NIL

Accesseur : accès à un élément

Argument :

Retourne :

Sémantique :

Transformateur : insertion d'un élément

Argument :

Retourne :

Sémantique :

Transformateur : suppression d'un élément

Argument :

Retourne :

Sémantique :

Autres opérations envisageables :

- afficher la totalité de la liste chaînée,
- retourner le nombre d'éléments contenus dans la liste chaînée,
- retourner la position d'une valeur dans la liste chaînée,
- retourner les données statistiques sur une liste chaînées si des champs contiennent des valeurs sur lesquelles il est possible de réaliser une étude statistique,
- …

5. Les tableaux associatifs

Un dictionnaire, ou tableau associatif, est donc une structure de données particulière en informatique.

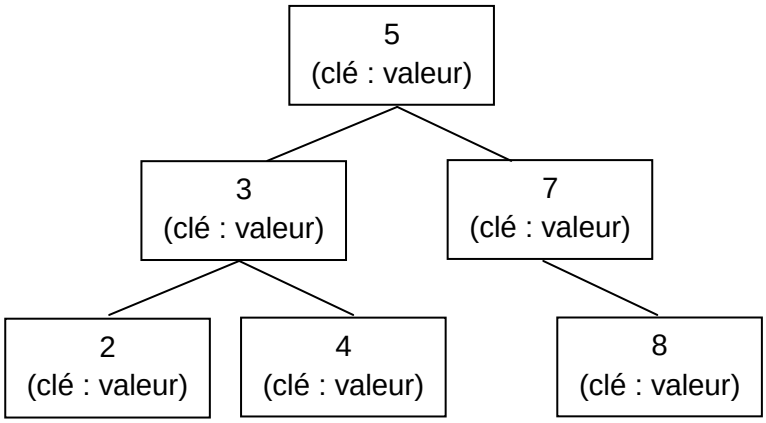
Les variables stockées dans un dictionnaire, nommées valeurs, ne vont pas être associées à un indice comme dans les tableaux, mais vont être associées à une clé. Chaque élément du dictionnaire est donc un couple constitué d'une clé et d'une valeur.
Un exemple est celui du carnet d'adresse, dont un élément peut être {clé : « Grand Schtroumpf » ; valeur : « Champignon jaune et rouge, Forêt enchantée »}. Dans ce cas, la clé et la valeur sont des chaînes de caractères. D'autres choix peuvent être faits.
Une attention particulière doit être apportée au choix de la clé : il ne peut s'agir d'un objet mutable (c'est-à-dire pouvant être modifié) car chaque modification de la clé devra entraîner une modification du stockage du dictionnaire.

Un problème peut survenir lorsque deux variables possèdent la même clé : on parle alors de collision.
Dans l'exemple du carnet d'adresse, on peut avoir plusieurs personnes ayant le même patronyme (par exemple, deux frères) mais habitant à des adresses différentes. Si l'on souhaite ajouter une valeur dans le dictionnaire avec une clé déjà existante, la valeur présente initialement sera effacée et remplacée par la nouvelle.
Il convient donc de choisir avec soin l'objet qui servira de clé pour le dictionnaire et de s'assurer qu'un couple (clé : valeur) ayant la même clé n'est pas déjà présent dans le dictionnaire avant de valider l'ajout du nouvel élément.
Ces collisions sont gérées de manières différentes suivant le choix qui a été fait pour implanter le dictionnaire au niveau informatique.

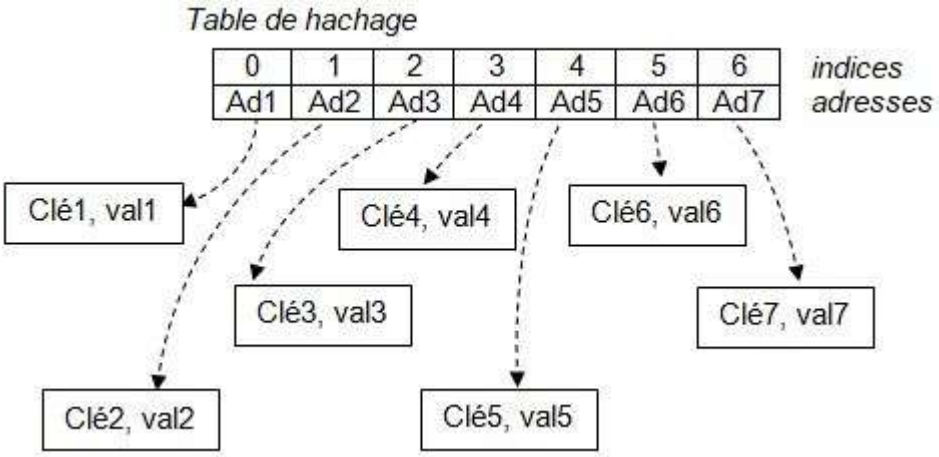
Un autre problème est celui de l'accès à l'information : la recherche de la clé dans la séquence non ordonnée des clés conduit à une complexité identique à celle observée pour les listes chaînée en ce qui concerne l'accès à l'information, à savoir une complexité linéaire ($O(n)$, où n est le nombre d'éléments dans le dictionnaire).
Afin de réduire cette complexité, des choix particuliers peuvent être faits en termes d'implémentation des dictionnaires.

Dans les différents langages informatiques, les dictionnaires vont être stockés dans des tableaux appelés tables de hachage ou dans des arbres binaires de recherche (programme du second semestre).

Pour information, un arbre est un graphe ne présentant aucun cycle. Un arbre binaire est un arbre dont chaque sommet a au plus deux successeurs. Enfin, un arbre binaire de recherche possède la particularité suivante : les sommets sont associés à un entier. En partant de la racine de l'arbre, puis pour chaque sommet, les entiers associés aux fils gauches sont plus petits que l'entier associé au sommet parent, alors que les entiers associés aux fils droits sont plus grands.



Dans le cas d'un stockage dans une table de hachage, on conserve l'avantage d'un accès direct au contenu de la variable, mais sans la contrainte de l'occupation contiguë des données en mémoire : la table de hachage regroupe les adresses mémoire auxquelles sont stockées les différentes valeurs.

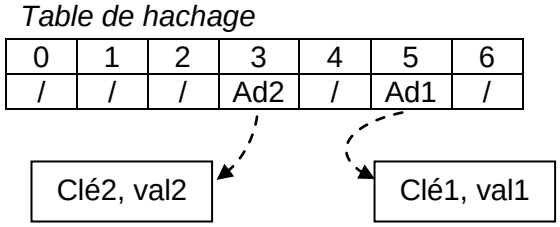


La clé doit alors être associée à un entier et c'est la connaissance de cet entier qui va permettre l'accès au contenu de la valeur via la table de hachage (l'entier sert alors d'indice dans la table ou dans l'ABR). Pour un ABR, la complexité pour l'accès à l'information est alors en O(ln (n)), alors que pour un tableau associatif, l'accès est direct (O(1)).

L'obtention de cet entier, nommé valeur de hachage, se fait grâce à une fonction de hachage. Cette fonction, $h(cle) = p$, doit être bijective afin que deux éléments distincts, donc ayant des clés différentes, ne correspondent pas à la même valeur de hachage, p. Dans un tel cas, on retrouverait le problème de collision déjà évoqué.

On peut noter que, dans le cas d'une table de hachage pouvant contenir n adresses, si le dictionnaire contient $m < n$ couples (clé : valeur), les valeurs de hachage utilisées effectivement ne correspondent pas forcément à des cases contiguës de la table de hachage.

Les fonctions de hachage comprennent la plupart du temps une fonction $modulo(n)$ (opérateur %) qui permet de ramener le résultat du calcul dans l'intervalle $[0\,;\,n-1]$, n étant le nombre d'éléments que peut contenir la table de hachage.



L'exemple suivant illustre le problème posé par une fonction de hachage peu efficace :

Dictionnaire : { « Rey » : « MPSI1 » ;
« Barbaroux » : « PCSI3 » ;
« Allain » : « PCSI1 »}

Fonction de hachage : somme des valeurs ASCII des caractères constituant la clé, modulo 2.
Les valeurs de hachage sont alors : $h("Rey") = 0$, $h("Barbaroux") = 0$ et $h("Allain") = 1$.

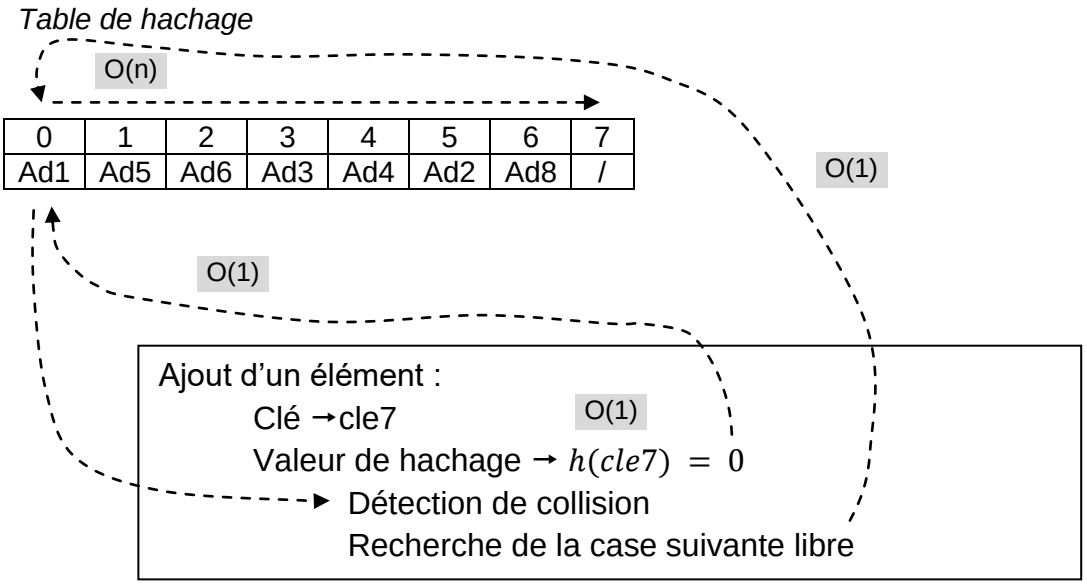
La fonction de hachage choisie fournit les mêmes valeurs de hachage pour les clés « Rey » et « Barbaroux», provoquant une collision : les deux objets ne pourront être stockés dans le dictionnaire.

La gestion des collisions peut se faire de différentes manières.

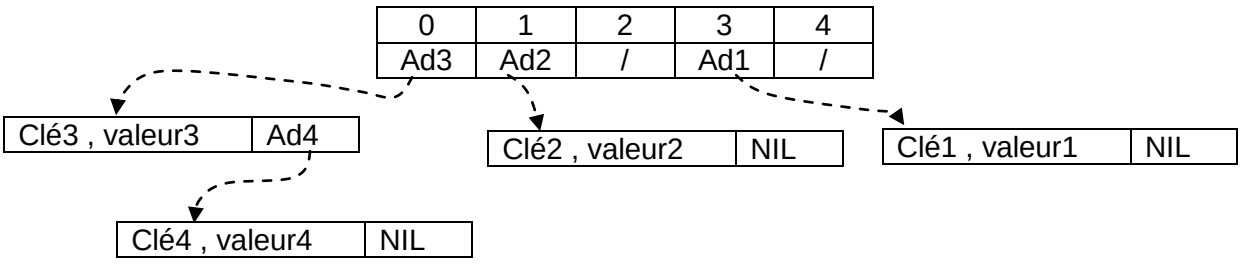
Une stratégie peut être de rechercher la case suivante libre la plus proche de celle ayant provoqué la collision.

Dans le cas d'une fonction de hachage peu efficace, le recours à cette stratégie fait perdre l'avantage de l'accès à une donnée de dictionnaire en temps constant ($O(1)$) : en effet, si l'on considère que le calcul de la valeur de hachage et de l'accès à la table ainsi qu'au couple (clé : valeur) à une adresse donnée sont tous en temps constant, le fait de devoir parcourir les adresse suivante pour rechercher la clé recherchée se fera en temps linéaire ($O(n)$).

Cette stratégie de gestion de collision par adressage fermé (puisque l'on reste dans la table de hachage) présente alors le risque d'avoir une complexité temporelle d'accès à l'information identique à celui d'une liste chaînée.

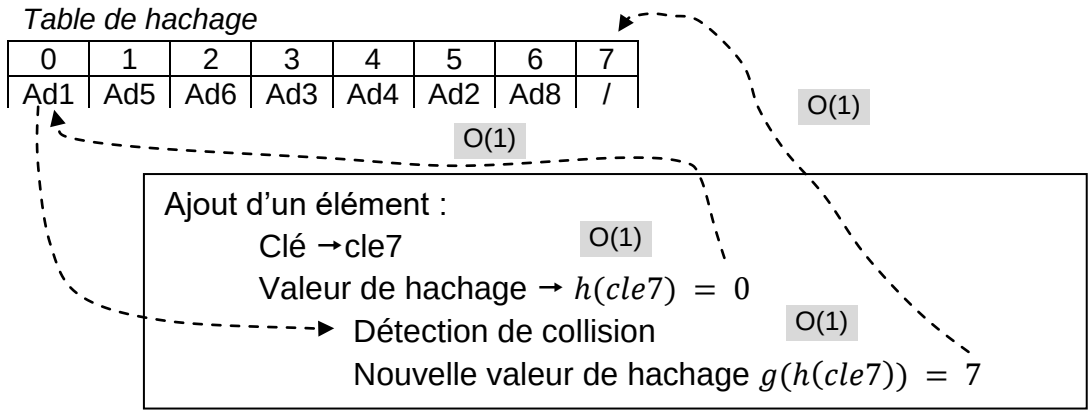


La stratégie de gestion de collisions par adressage ouvert consiste à modifier le contenu de chaque élément du dictionnaire pour y ajouter l'éventuelle adresse d'un autre couple (clé : valeur). On construit alors une liste chaînée dont tous les maillons ont la même valeur de hachage et dont le premier élément est à l'adresse contenu dans la table de hachage.



Cette stratégie est plus coûteuse en mémoire puisque, pour chaque couple (clé : valeur), il convient de prévoir le stockage d'une adresse.

D'autres stratégies existent, notamment celle consistant à introduire une seconde fonction de hachage, utilisant comme variable la valeur de hachage obtenu avec la première fonction ; on a alors $p' = g\,(h(clé))$, qui conduit à un accès en temps constant.



Si en Caml et en Python les dictionnaires sont déjà implémentés, ils n'existent pas en C. Il faut alors choisir et programmer la fonction de hachage, implémenter les différentes opérations en utilisant un table comme table de hachage et prévoir la gestion des collisions.

La définition des différentes opérations associée à un dictionnaire en C fera l'objet d'un TP (exercice 7.2.).

6. Les piles et les files

Les piles et les files sont deux structures de données très présentes en informatique. Elles se caractérisent par le fait que seul un élément, situé à une extrémité de la structure, est accessible.

Si dans certains langages tels que python ou Caml, les piles et les files sont déjà implémentées dans des modules, elles n'existent pas en C.

Elles peuvent alors être stockées dans des tableaux ou dans des listes chaînées. Dans le cas d'une implantation par liste chaînée, deux types de fonctions existeront : celles gérant les listes chaînées, telles que définies précédemment, et celles gérant la pile ou la file, qui font appel aux fonctions dédiées aux listes chaînées.

6.1. Les piles (stack)

Les piles sont une collection d'éléments de même type ; leur particularité est que seul un élément, celui situé au sommet de la pile, est accessible. L'ajout d'un élément se fait aussi au sommet de la pile.

Ces structures de données sont donc linéaires et dynamiques. Elles ont un fonctionnement de type LIFO (Last In, First Out) et ont de nombreuses utilisation : pile des blocs d'activation en mémoire, touche undo sur les logiciel, retour à la page précédente sur un navigateur internet, parcours d'arbres ou de graphes en profondeur, calculatrice à notation polonaise inversée, …

Spécification d'une pile

Nom : pile

Donnée manipulées : informations

Opérations :

- Constructeur : initialisation de la pile.
- Accesseur :
  - Est_vide : indique s'il y a un élément au sommet de la pile
  - Peek : consulte la valeur de l'élément au sommet de la pile (opération normalement interdite)
- Transformateurs :
  - Empiler (push) : Insertion de l'élément au sommet de la pile
  - Dépiler (pop) : suppression de l'élément au sommet de la pile

Description des opérations

Constructeur : initialisation

Argument : aucun argument ou taille maximum de la pile
Retourne : pile
Sémantique :

- Crée le premier élément de la pile
- L'insère dans la pile

Accesseur : consultation de l'élément au sommet de la pile (peek)

Argument : pile
Retourne : information contenue dans l'élément au sommet de la pile
Sémantique :

Retourne l'information contenue au sommet de la pile (normalement, cette opération est réalisée en faisant un pop, une duplication puis un push de l'élément au sommet de la pile)
Retourne un indicateur particulier si la pile est vide

Accesseur : teste si la pile est vide
- Argument : pile
- Retourne : un booléen
- Sémantique :
  - Consulte le sommet de la pile
  - Retourne `true` si l'élément au sommet de la pile est inexistant, `false` sinon

Transformateur : insertion d'un élément
- Argument : pile, élément
- Retourne : pile
- Sémantique :
  - Ajoute l'élément passé en argument au sommet de la pile
  - Retourne la pile modifiée si l'opération s'est correctement déroulée
  - Retourne la pile non modifiée si l'opération ne s'est pas correctement déroulée

Transformateur : suppression d'un élément
- Argument : pile
- Retourne : pile
- Sémantique :
  - Vérifie que la pile n'est pas vide
  - Supprime l'élément au sommet de la pile si la pile n'est pas vide
  - Retourne la pile vide si la pile était initialement vide
  - Retourne la pile modifiée sinon

Les piles ne seront manipulées qu'au travers de ces opérations.

Exercice

On considère une pile nommée pile. Cette structure ne peut être manipulée qu'à travers les opérations définies précédemment : créer_pile, est_vide, peek, push et pop.

Ecrire les algorithmes suivant :

- `Depile_K` : fonction recevant une pile et un entier naturel k et dépilant k éléments de la pile si celle-ci contient au moins k éléments et vide la pile sinon.

- `Depile_jusqua` : fonction recevant une pile et un élément e, qui dépile la pile jusqu'à l'élément e exclu si e est présent dans la pile ou toute la pile sinon. Cette fonction retourne la pile modifiée.

- `Inverse_pile` : fonction recevant une pile et retournant une nouvelle pile, dont l'ordre des éléments est inversé par rapport à la pile passée en argument.

6.2. Les files (queue)

Comme les piles, les files sont une collection d'éléments de même type ; leur particularité est que seul un élément, celui situé au sommet de la pile, est accessible. L'ajout d'un élément se fait à l'autre extrémité de la file.

Ces structures de données sont elles aussi linéaires et dynamiques.

Elles ont un fonctionnement de type FIFO (First In, First Out) et sont utilisées pour la gestion des files d'attente (accès à une ressource, par exemple), les parcours d'arbres et de graphes en largeur ou pour les phénomènes pour lesquels la chronologie est importante.

Spécification d'une file

Nom : file

Donnée manipulées : informations

Opérations :

- Constructeur : initialisation de la file.
- Accesseur :
  - Est_vide : indique s'il y a un élément en tête de la file
  - Peek : consulte la valeur de l'élément au sommet de la pile (opération normalement interdite)
- Transformateurs :
  - Enfiler (push) : insertion de l'élément en queue de file
  - Défiler (pop) : suppression de l'élément en tête de file

Description des opérations

Constructeur : initialisation

Argument : aucun argument ou taille maximum de la pile
Retourne : file
Sémantique :

- Crée le premier élément de la file
- L'insère dans la file

Accesseur : consultation de l'élément en tête de file (peek)

Argument : file
Retourne : information contenue dans l'élément en tête de file
Sémantique :

Retourne l'information contenue en tête de
Retourne un indicateur particulier si la pile est vide

Accesseur : teste si la file est vide

Argument : file

Retourne : un booléen
Sémantique :
Consulte la tête de la file
Retourne `true` si l'élément en tête de file est inexistant, `false` sinon

Transformateur : insertion d'un élément
Argument : file, élément
Retourne : file
Sémantique :
Ajoute l'élément passé en argument en queue de file
Retourne la file modifiée si l'opération s'est correctement déroulée
Retourne la file non modifiée si l'opération ne s'est pas correctement déroulée

Transformateur : suppression d'un élément
Argument : file
Retourne : file
Sémantique :
Vérifie que la file n'est pas vide
Supprime l'élément en tête de file si la file n'est pas vide
Retourne la file vide si la file était initialement vide
Retourne la file modifiée sinon

Les files ne seront manipulées qu'au travers de ces opérations.

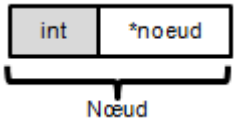
7. Exercices

7.1. Listes chaînées en C

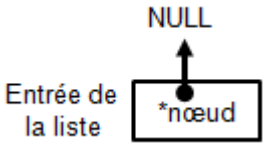
L'objectif de l'exercice est de programmer les différentes opérations permettant de gérer les listes chaînées en C.
Dans cet exercice, l'information contenue dans un nœud sera un entier.

Fonctions préliminaires

1) Créer le type `nœud`, associé à une structure nommée `structure_noeud`. Cette structure contient deux champs :
   - `info` de type `integer`
   - `suivant` de type pointeur sur `noeud`.



2) Créer le type associé à l'interface de la liste chaînée. L'entrée de la liste est un nœud particulier, nommé `listeCHainee` associée à une structure nommée `liste` ne contenant qu'un seul champ nommé `entree` : un pointeur sur un `nœud`, initialisé à `NULL` lors de la création de l'interface.



3) Ecrire une fonction de signature `noeud* creeElement ( int valeur )` permettant la création d'un nœud. Le champ `suivant` du nœud sera affecté à `NULL`. Afin de permettre la libération du nœud, la création sera dynamique (utilisation du `malloc`). En cas d'échec de la création du nœud, la fonction devra retourner un pointeur sur `NULL`.

4) Ecrire une fonction de signature `listeChainee* creeInterface ()` permettant la création de l'interface avec le premier élément de la liste chaînée. Le champ `entree` sera affecté à `NULL`. En cas d'échec de la création du 'interface, la fonction devra renvoyer un pointeur sur `NULL`.

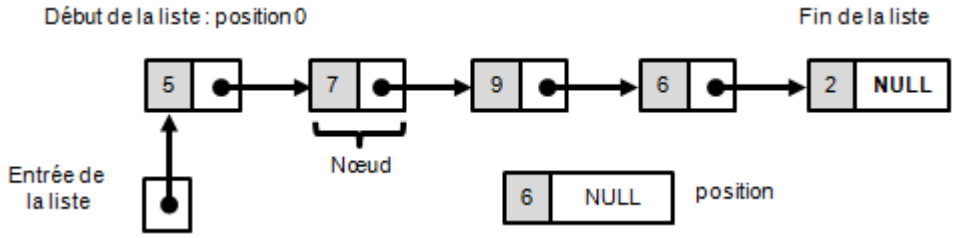
5) Ecrire une fonction de signature `void afficheListe(listeChainee lst)` permettant d'afficher les valeurs des informations contenues dans chaque nœud de la liste chaînée sur laquelle pointe l'interface passée en argument.

Opérations propres aux listes chaînées

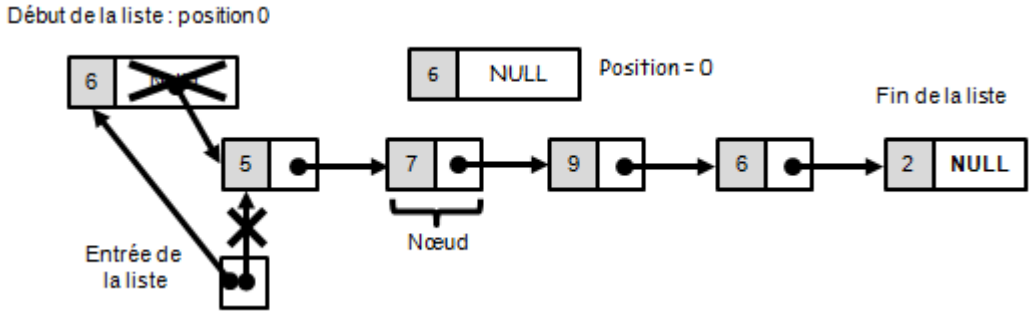
Les fonctions ci-dessous sont celles utilisées pour la gestion d'une liste chaînée. Pour ce type abstrait, on rappelle que le nombre d'éléments présents dans la liste n'est pas connu. Les fonctions devront tenir compte de cet impératif.

6) Accesseur : écrire une fonction de signature `noeud* accedeNoeud ( listeChainee lst, int position)` renvoyant un pointeur sur le nœud situé à la position `position` dans la liste chaînée passée en argument. Cette fonction renvoie `NULL` si `position` est supérieur au nombre d'éléments présents dans la liste chaînée.

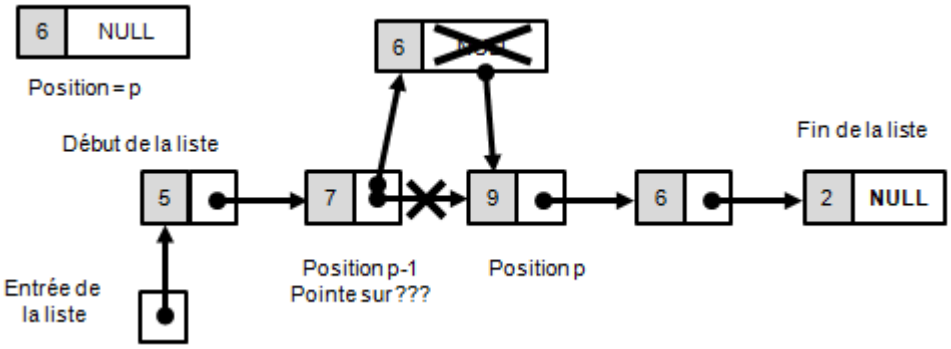
7) Transformateur : insertion d'un nœud à la position `position` : écrire une fonction de signature `int insereNoeud (listeChainee *lst, int valeur, int position)` permettant d'insérer un nœud contenant la valeur passée en argument à la position `position` dans la liste. Pour ce faire, cette fonction devra :
   - Créer le nœud associé à la valeur `valeur`,
   - Vérifier que le nœud a bien été créé,



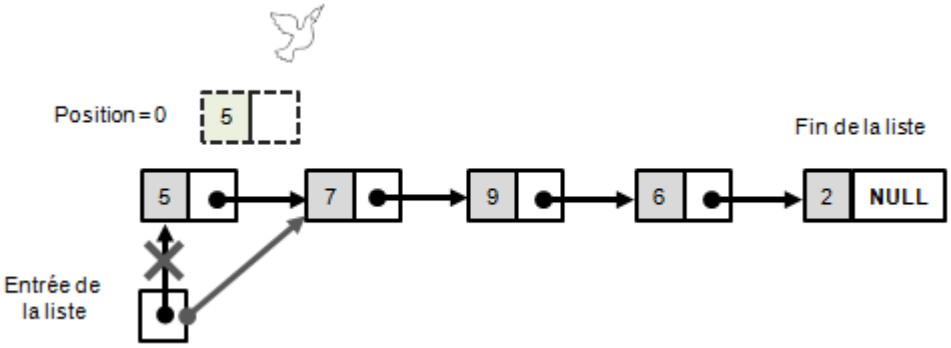
   - Tester si le nœud doit être inséré en tête de liste. Dans ce cas, l'interface doit être modifiée et le champ suivant du nœud inséré doit pointer sur l'adresse initialement contenue dans l'interface. Une fois l'insertion réalisée, la fonction retourne l'indicateur 0.



- Si le nœud doit être inséré à une autre position que la tête de la liste chaînée, la fonction doit :
    - Vérifier que la position demandée est valide,
    - Si la position est valide, il faut affecter au champ suivant du nœud à insérer le champ suivant du nœud précédent dans la liste, et affecter au champ suivant du nœud précédent dans la liste l'adresse du nœud à insérer. La fonction retourne alors l'indicateur 0.
    - Si la position n'est pas valide, la fonction retournera l'indicateur -1.

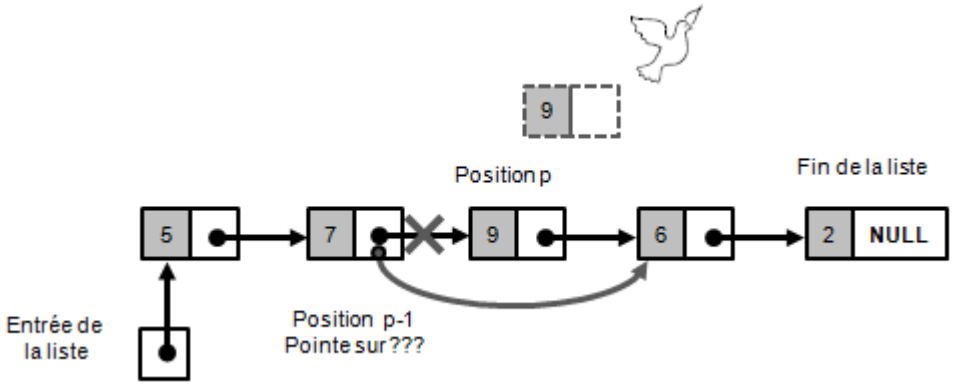


8) Transformateur : suppression d'un nœud à la position `position` : écrire une fonction de signature `int supprimeElement(listeChainee *lst, int position)` permettant de supprimer le nœud indiqué à la position passée en argument. Cette fonction devra :
    - Tester si la position est celle du premier nœud de la liste chaînée (position 0). Dans ce cas, si la liste chaînée n'est pas vide, l'interface doit être modifiée afin qu'elle pointe sur l'adresse contenue dans le champ suivant du nœud à supprimer. Le nœud supprimé doit être libéré (`free`). Si la liste chaînée est vide, l'indicateur -1 est retourné.



- Si la position est autre que la tête de liste chaînée (position non nulle), la fonction doit vérifiée que la position est valide. Si elle ne l'est pas, l'indicateur -1

est retourné. Si la position est valide, le champ suivant du nœud précédent doit recevoir l'adresse contenue dans le champ suivant du nœud à supprimer. Le nœud est ensuite libérer (`free`) et la fonction retourne l'indicateur 0.



Fonctions complémentaires

Les fonctions suivantes peuvent être envisagées :

- `int longueur_liste (listeChainee lst)`
  Retourne le nombre d'élément dans la liste chaînée `lst`

- `int recherche_valeur (listeChainee lst, int valeur)`
  Recherche de la position d'une valeur donnée dans la liste chaînée `lst`
  Retourne -1 si la valeur est absente, la position sinon

- `void concatene_liste (listeChainee *lst1, listeChainee, lst2)`
  Ajoute la liste chaînée `lst2` à la fin de la liste chaînée `lst1`

- `void echange_dans_liste (listeChainne *lst)`
  Echange les deux moitiés d'une liste chaînée lst

- `int est_prefixe(listeChainee lst1, listeChainee lst2)`
  Retourne 0 si `lst2` est prefixe de la liste chaînée `lst1` , retourne -1 si non
  Fonction récursive

- `int copie_liste(listeChainee lst1, listeChainee *copie)`
  Recopie une liste chainée `lst1` dans une liste chainée copie, déjà initialisée
  Retourne 0 si succès / -1 si échec

7.2. Dictionnaire : gestion d'une base de données de chats chez un vétérinaire (DS 2021)

Un vétérinaire souhaite mettre en place une base de données regroupant les données des chats qu'il reçoit régulièrement en consultation. Les données à stocker sont : le nom du chat, son âge, le nom de son propriétaire et son numéro d'identifiant (tatouage ou puce électronique). On suppose que tous les chats de la base ont un nom différent.

A. Stockage des données

Dans un premier temps, un enregistrement regroupant les différentes informations doit être défini. Les attributs retenus sont :

- `nom`, chaîne de caractères de taille indéfinie ;
- `proprietaire`, chaîne de caractères de taille indéfinie;
- `age`, entier ;
- `tatouage`, chaîne de 7 caractères.

1) Ecrire la définition du type `animal`, associé à une structure `s_animal`, répondant au cahier des charges.
2) Une première implémentation envisagée consiste à stocker ces enregistrements dans un tableau. Citez les avantages et inconvénients d'une telle implantation dans ce contexte.
3) Une seconde implémentation par liste chaînée est envisagée. Citez les avantages et inconvénients d'une telle implantation dans ce contexte.

B. Gestion d'une table de hachage simple

Le choix de l'implantation s'est finalement porté sur un dictionnaire (ou tableau associatif).
Les enregistrements figurant dans le dictionnaire sont définis par le type animal :

```
typedef struct s_animal
{
 char *id;                  // Nom correspondant à la clé
 description Animal;        // Caractéristiques du chat
} animal;
```

Avec :

```
typedef struct s_description
// Caractéristiques du chat
{
 char *proprietaire;
 char *tatouage;
 int age;
} description;
```

La fonction suivante permet de créer un enregistrement de type `animal` à partir des caractéristiques du chat fournies en argument et de retourner un pointeur sur cet enregistrement :

```
animal *CreeAnimal(char *nom, char *maitre, int ans, char *tatoo)
// Précondition : caractéristiques du chat
//               nom->str (clé), nom du chat
//               maitre->str, proprietaire
//               age->int
//               tatoo->str[7], n° identification du chat
// Post-condition : retourne un pointeur sur animal ou abort si échec
```

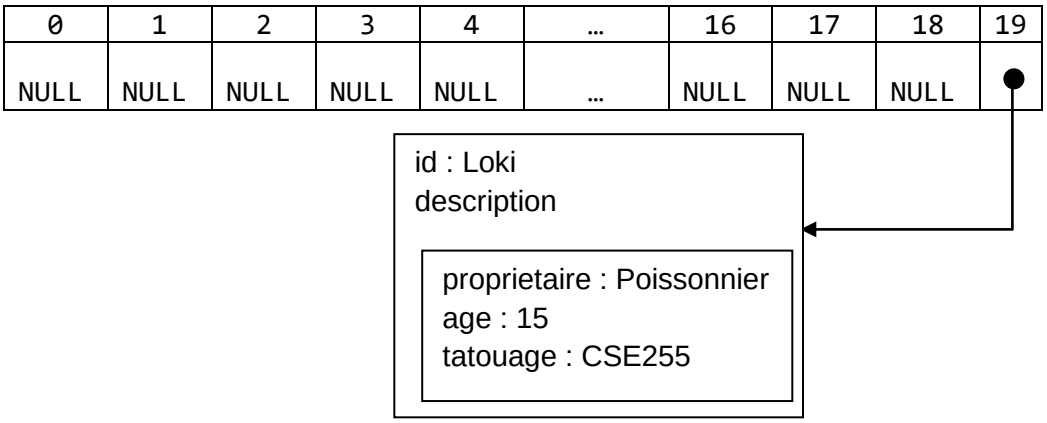
Les pointeurs issus de la création de ces enregistrements seront stockés dans un tableau.

Afin d'améliorer l'accès aux enregistrements, le tableau sera une table de hachage : la position d'un enregistrement dans le tableau dépend de l'application d'une fonction de hachage à la clé. Dans le cas du cabinet vétérinaire, la table de hachage contiendra 20 cases (`int nb_animaux = 20`). La fonction de hachage prendra en entrée la clé (nom du chat) et retournera un entier, la valeur de hachage, résultant de la somme de valeurs ASCII associées aux caractères du nom modulo `nb_animaux`.

Ainsi, pour un chat nommé Loki, on a : p = (76 + 111 + 107 + 105) % 20 = 19.

Le pointeur sur l'enregistrement correspondant au chat nommé Loki sera donc positionné dans la case 19 de la table de hachage.

Table de hachage : `table`



4) Ecrire la fonction de signature : `animal **CreeTable (int taille)`

```
// Précondition : taille->int, nb d'éléments dans la table de hachage
// Postcondition : table de hachage (tableau dynamique de pointeurs sur animal initialisés à NULL)
```

5) Ecrire la fonction permettant de calculer le haché d'un nom. Cette fonction a pour signature :

```
int FonctionHachage(char *nom, int taille)
// Précondition : nom->string, nom du chat,
//                taille->int, modulo pour le haché
// Poscondition : hache->int, hache (somme ascii % taille)
```

Remarque : en C, le caractère 'a' peut indifféremment être interprété comme le caractère `'a'` ou comme l'entier 97. On peut par exemple écrire `'a'+2`. On obtient alors `'c'` ou 99.

6) La fonction permettant l'ajout d'un nouveau chat est la suivante :

```
int AjouteChat(animal *table[], char *nom, char *maitre, int ans, char
*tatoo, int taille)
{
 animal *nouveau = CreeAnimal(nom, maitre, ans, tatoo) ;
 int hache = FonctionHachage(nom, taille) ;
 int controle = RechercheChat(table, nom, taille) ;
      // Fonction RechercheChat :
      // Fonction retournant -1 si le chat est absent de la table
      // Et retournant le haché du chat s'il est présent

 if (controle == -1) {
      table[hache] = nouveau ;
      return 0 ;
 }
 else {
      return -1 ;
 }
}
```

   a. Proposer les spécifications de l'opération réalisée par cette fonction.
   b. Ecrire la fonction recherchant un chat dans la table :

```
int RechercheChat(animal *table[], char nom[], int taille)
// Precondition : table->tableau de pointeurs sur animaux
//     nom->str, clé d'un chat
//     taille->int nombre d'éléments dans table de hachage
//Postcondition : controle->int,
//     hache si chat présent dans la table
//     La présence d'un chat n'est validée que si son nom correspond
//     au nom (attribut id) contenu dans l'enregistrement pointé.
//     -1 si le chat est absent de la table
```

M. Mellouët a trouvé un chat qu'il souhaite adopter. Il l'amène au cabinet vétérinaire afin de le faire identifier et vacciner. Les attributs de ce nouveau venu sont :

Nom : Tiberius
Tatouage : KIRK77
Age : 7 ans

7) Calculer le haché de Tiberius. Quel problème pose l'insertion du chat de M. Mellouët dans la table ? A quoi cela est-il dû ?

Un tel problème est qualifié de collision.

8) Modifier la fonction `AjouteChat` afin que les collisions soient détectées :

```
// Post-condition : controle ->int entier de controle
//    0 : réussite de l'ajout du chat
//    -1 : détection d'une collision – échec de l'ajout du chat
//    -2 : echec car chat déjà présent
```

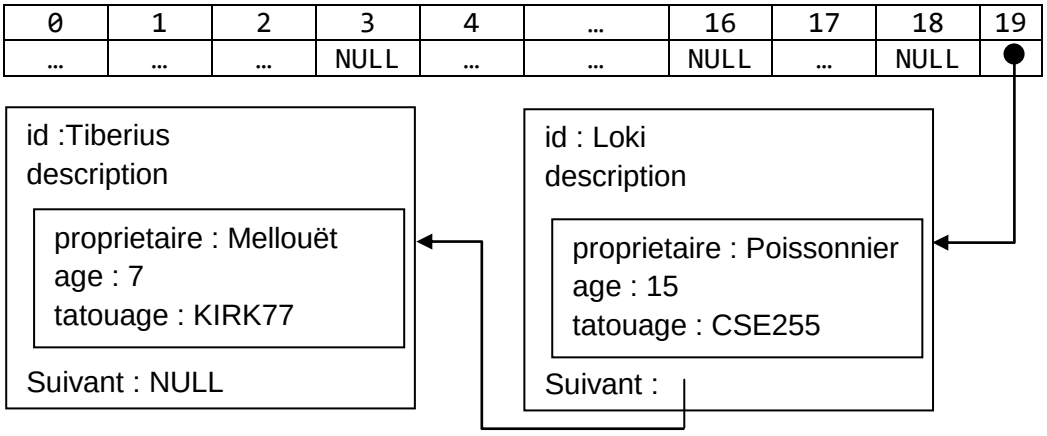
Seules les conditionnelles (partie sur les tests) seront ré-écrites ; la signature et les déclarations étant inchangées, il est inutile de les rappeler.

C. Gestion des collisions par chaînage externe

Afin de gérer les problèmes de collisions, le type abstrait est modifié afin de pouvoir obtenir la table suivante :

Valeur de hachage	nom	proprietaire	age	tatouage
2	Bulbizarre	Ouisse	4	GEEK01
2	Tangente	Lelaidier	2	COS314
4	Moebius	Gouret	5	HOP421
4	Schrodinger	Geindreau	8	BOI109
4	Chapka	Vault	10	HOP421
5	Finwe	Lemerle	7	SAM107
5	Ulysse	Tronelle	9	ILL097
6	Courgette	Gouret	10	HIP111
6	Babiole	Rialland	3	NOE221
12	Cotangente	Lelaidier	1	SIN628
12	Moustique	Adnot	3	BZE777
12	Pomare	Bescond	7	POL987
13	Silawe	Clorennec	4	TAN002
13	Wolfi	Clorennec	11	MUS440
14	Hubble	Gate	3	EXT123
18	Douala	Garnaud	6	TOG007
19	Loki	Poissonnier	14	CSE255
19	Tiberius	Mellouët	7	KIRK77

La gestion des collisions se fait par chaînage externe : à chaque élément de la table de hachage correspond une liste chaînée.



Le type animal devient :

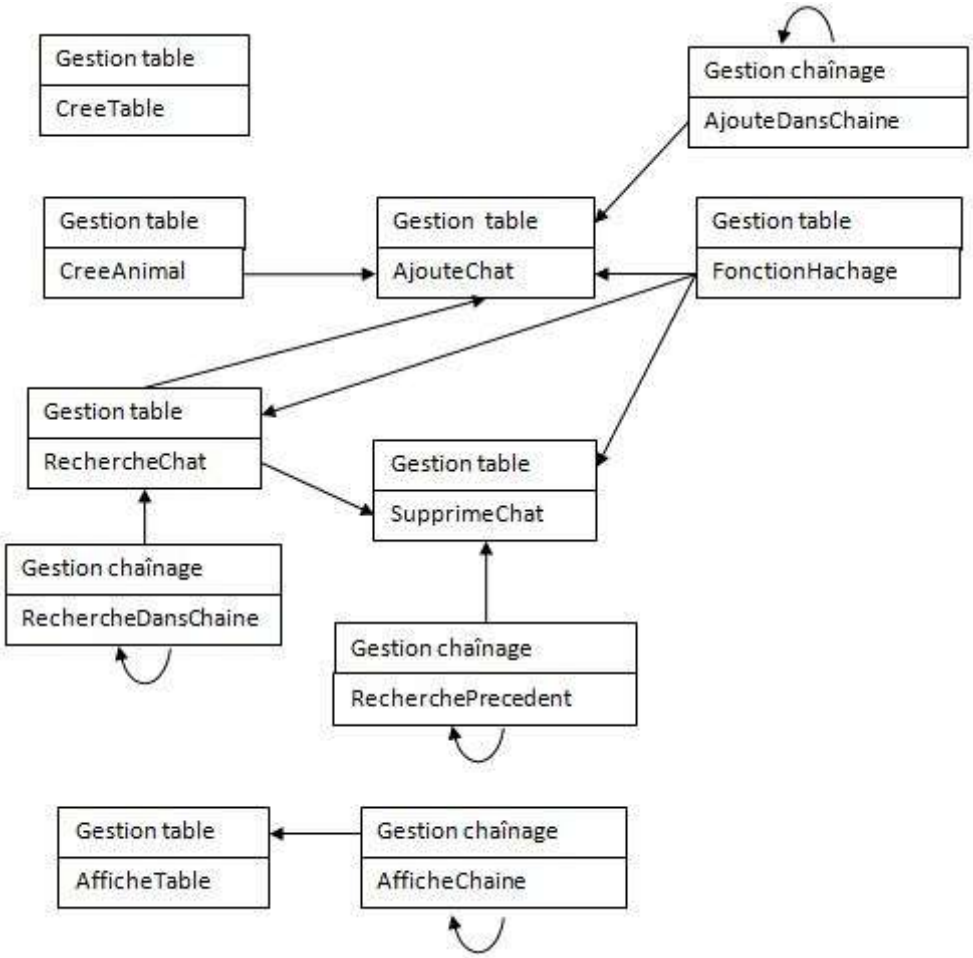
```
typedef struct s_animal
{
      char *id ;                        // Nom correspondant à la clé
      description Animal ;       // Caractéristiques du chat
      struct s_animal *suivant ;      // pour chaînage
}animal ;
```

Deux types de fonctions sont alors écrites : celles gérant la table de hachage et celles gérant les listes chaînées pointées par la table.

```
// Gestion table
animal *CreeAnimal(char *nom, char *maitre, int ans, char *tatoo) ;
animal **CreeTable(int taille) ;
int FonctionHachage(char nom[], int taille) ;
animal* RechercheChat(animal *table[], char nom[], int taille) ;
int AjouteChat(animal *table[], char *nom, char *maitre, int ans, char
*tatoo, int taille) ;
void AfficheTable(animal* table[], int taille) ;
int SupprimeChat(animal* table[], char *nom, int taille) ;

// Gestion chaînage externe
animal* RechercheDansChaine(animal* courant, char *nom) ;
int AjouteDansChaine(animal* nouveau, animal* entree) ;
void AfficheChaine(int nb, animal* courant) ;
animal *RecherchePrecedent(animal* courant, char *nom) ;
```

On choisit de représenter les interactions entre les différentes fonctions sous la forme d'un diagramme.
Dans ce diagramme, une flèche indique qu'une fonction appelle une autre fonction. Ainsi, la fonction `AfficheTable` appelle la fonction `AfficheChaine`. La fonction `AfficheChaine` s'appelle elle-même : elle est donc récursive.
Les fonctions permettant la libération des pointeurs ne figurent pas sur ce diagramme.



10) Décrire la sémantique de l'opération de suppression d'un chat de la table. Justifier les interactions avec les autres fonctions.

11) Ecrire la fonction `RecherchePrecedent`, de signature :

```
animal *RecherchePrecedent(animal* courant, char *nom)
// Précondtion : courant->pointeur sur animal pour recherche individu
//     nom->str, cle du chat recherché, présent dans la liste
// Post-condition : retourne un pointeur sur le chat précédent
```

12) Ecrire la fonction `SupprimeChat`, de signature :

```
int SupprimeChat(animal* table[], char *nom, int taille)
// Précondition : table de hachage,
//     taille->int nb éléments dans la table
//     nom->nom du chat à supprimer de la table
// Post-condition : controle -> int entier de contrôle
//     0 : réussite de la suppression du chat
//     -1 : echec car chat absent
```

7.3. Calculatrice NPI

En cours d'écriture…