Les listes

Une **liste** est une structure de données permettant de regrouper des données. C'est une collection finie et ordonnée d'éléments, cela signifie que chaque élément d'une liste est repéré par son index (sa position). A la différence d'un tableau qui est de taille fixe, une liste est extensible : on peut lui ajouter, retirer des éléments ; sa taille n'est donc pas fixe.

Un peu d'histoire pour commencer

Crédit image : — https://www.flickr.com/photos/null0/272015955/, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=1297606

Le langage LISP, inventé par John Mac Carthy (photo) en 1958, est un des premiers à utiliser la notion de *liste* (LISP vient de l'anglais *list processing* signifiant « traitement de listes »).

Les listes du langage LISP sont composées de deux parties :

- une **tête** correspondant au premier élément de la liste
- une **queue** correspondant au reste de la liste

On pouvait alors construire une liste à partir d'un premier élément (sa tête) et d'une liste (sa queue).

L' interface minimale du type abstrait `Liste`

Le type abstrait `Liste` peut alors être défini par l'*interface* suivante contenant 5 opérations primitives :

- Des constructeurs :
  - `listevide()` pour construire une liste vide
  - `construit(e, L)` pour construire une nouvelle liste contenant un premier élément `e` (sa tête) et une suite `L` (sa queue, qui est une liste). Cet opérateur est aussi souvent noté `cons`.
- Des sélecteurs :
  - `premier(L)` pour accéder au premier élément de la liste `L`, sa tête. Cet opérateur est aussi souvent noté `car`.
  - `reste(L)` pour accéder au reste de la liste `L` c'est-à-dire sa queue. Cet opérateur est aussi souvent noté `cdr`.
- Un prédicat :
  - `estvide(L)` pour tester si une liste est vide.

Ainsi, pour construire une liste formée par les nombres 5, 3, 8 (dans cet ordre) on fait :

```
maliste1 = construit(5, construit(3, construit(8, listevide())))
```

Dans ce cas,

- `premier(maliste1)` correspond à sa tête, c'est-à-dire `5`
- `reste(maliste1)` correspondant à sa queue, c'est-à-dire la liste correspondant à `construit(3, construit(8, listevide()))` formée des nombres 3 et 8.

> On sait depuis les travaux de Mac Carthy sur le langage LISP, qu'avec ces 5 opérations on peut reconstruire toutes les opérations sur les listes (accéder à un élément, modifier un élément, ajouter/supprimer un élément, calculer la longueur, tester l'appartenance, etc.)

Implémentations possibles

Une implémentation avec des couples en Python

Cette première implémentation est basée sur des paires (couples) qui comportent chacune un élément et la suite de la liste, qui elle-même peut être une paire... Elle réutilise le type `tuple` de Python.

> On définit ainsi notre structure de données de manière *récursive* pour respecter la philosophie du langage LISP. Nous étudierons cette notion de récursivité plus tard dans l'année.

In [1]:
```
def listevide():
    return None  # on utilise None pour une liste vide

def construit(e, L):
    return (e, L) # renvoie un tuple de deux éléments

def premier(L):
    return L[0] # accès au premier élément du couple (la tête de L)

def reste(L):
    return L[1] # accès au deuxième élément du couple (la queue de L)

def estvide(L):
    return L is None  # L est égal à None ?
```

On peut alors tester les instructions précédentes et en affichant le contenu de `maliste1`, on se rend compte de l'implémentation choisie avec des paires imbriquées.

In [2]:
```
maliste1 = construit(5, construit(3, construit(8, listevide())))
maliste1
```

Out[2]:
```
(5, (3, (8, None)))
```

On a donc logiquement :

In [3]:
```
premier(maliste1)
```

Out[3]:
```
5
```

In [4]:
```
reste(maliste1)
```

Out[4]:
```
(3, (8, None))
```

Avec cette implémentation, il est intéressant de noter que la contruction de `maliste1` avec l'instruction

```
construit(5, construit(3, construit(8, listevide())))
```

nécessite la construction 3 paires intermédiaires qui sont construites de la plus imbriquée (liste vide) à la moins imbriquée.

On peut visualiser facilement cela avec Python tutor ici.

Comme dit précédemment, on peut construire les autres opérations sur les listes à partir des cinq opérations définies plus haut.

Par exemple, pour obtenir le dernier élément d'une liste, on peut implémenter l'opération `dernier(L)` à partir des autres :

In [5]:
```
def dernier(L):
    """
    Liste --> Element
    Précondition : L n'est pas vide.
    """
    while reste(L) != listevide(): # tant que le reste de la liste n'est pas vide
        L = reste(L) # on passe au reste
    return premier(L) # on renvoie le premier élément de la dernière paire
```

En partant de la liste `L`, il suffit de passer au reste jusqu'à ce que le reste soit la liste vide. On sait alors que l'on est arrivé à la dernière paire et il suffit de renvoyer le premier élément de cette dernière paire.

En appelant la fonction sur `maliste1` on peut vérifier qu'elle renvoie bien 8.

In [6]:
```
maliste1 = construit(5, construit(3, construit(8, listevide())))
dernier(maliste1)
```

Out[6]:
```
8
```

A faire : Activités 1 et 2

Une implémentation avec le type `list` de Python

Il est possible d'implémenter les 5 opérations définissant le type abstrait `Liste` en utilisant le type prédéfini `list` de Python. Les fonctions sont très ressemblantes à celles utilisant les couples. Cela fait l'objet des activités 3 et 4 qui proposent deux implémentations différentes : la première avec copie des listes intermédiaires (comme l'implémentation avec les couples) et la seconde avec modification de la liste en place.

A faire : Activités 3 et 4

Les listes en pratique

Créer des nouvelles listes ou les modifier en place ?

Dans certaines implémentations, les opérations créent des nouvelles listes par copie des listes passées en paramètre (la liste passée en paramètre n'est donc pas modifiée) et dans d'autres, les listes sont modifiées *en place* par les opérations (c'est la liste passée en paramètre qui est modifiée).

Sans entrer dans les détails des avantages et inconvénients de chaque méthode, il est important d'en tenir compte pour que l'interface et l'implémentation soient cohérentes.

Le fait que les modifications s'effectuent en place sur une liste ou non, doit être spécifié dans l'interface du type abstrait `Liste` car cela a un impact sur la façon de l'implémenter. De plus, il faut essayer d'être cohérent dans le nommage des opérations : si l'opération `construit(e, L)` définie au départ sous-entend qu'une nouvelle liste est *construite* par l'appel à cette fonction, le verbe *csontruire* n'est pas adapté à des modification en place de la liste passée en paramètre, on lui préférera par exemple le nom `ajouter_en_tete(e, L)` pour davantage de clarté.

> Ceci a été abordé dans les activités 3 et 4.

Des interfaces plus complètes

Il n'est pas rare de définir le type abstrait `Liste` par un jeu d'opérations différent et surtout plus complet. Par exemple, avec l'interface proposée au début, si l'utilisateur souhaite insérer un élément dans la liste, il doit programmer lui-même l'opération `inserer` à partir des 5 opérations de base. C'est pourquoi certaines interfaces contiennent des opérations plus avancées, afin de faciliter le travail de l'utilisateur.

Exemple

L'interface suivante, qui spécifie aussi 5 opérations, permet également de définir le type abstrait `Liste` :

- `listevide()` : pour construire une liste vide.
- `taille(L)` : renvoie le nombre d'éléments de la liste `L`.
- `lire(L, i)` : renvoie l'élément en position `i` dans `L`. *Précondition :* `i` *est dans 0..taille(L)-1.*
- `inserer(L, e, i)` : insère l'élément `e` dans `L` en position `i` en incrémentant la position de tous les éléments à partir de `i`. *Précondition : i est dans 0 taille(L).*
- `supprimer(L, i)` : supprime l'élément en position `i` dans `L`, en décrémentant la position des tous les éléments à partir de `i` + 1.

Avec ces 5 opérations on peut également construire toutes les autres sur les listes. Cette interface ne change pas la nature du type abstrait `Liste` mais modifie la façon d'écrire des algorithmes l'utilisant : par exemple,

- pour accéder au premier élément, on ne va plus écrire `premier(L)` mais `lire(L, 0)`,
- pour accéder au dernier élément, on n'écrira plus `dernier(L)` mais `lire(L, taille(L)-1)`,
- etc.

> **Remarque** : les termes utilisés pour spécifier les opérations sont importants : ici, il faut bien comprendre que les modifications (d'insertion, suppression) se font *en place* : c'est la liste `L` elle-même qui est modifiée.

Les deux implémentations classiques

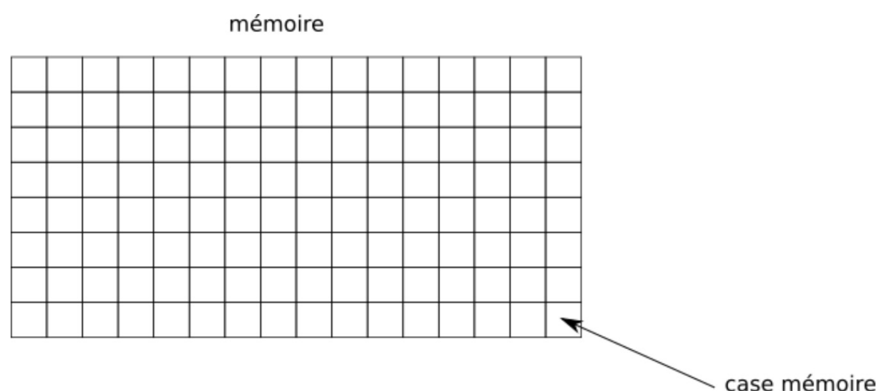
En pratique, il y a classiquement deux manières pour implémenter efficacement des listes :

- à l'aide de tableaux (dynamiques) ou
- à l'aide de listes chaînées

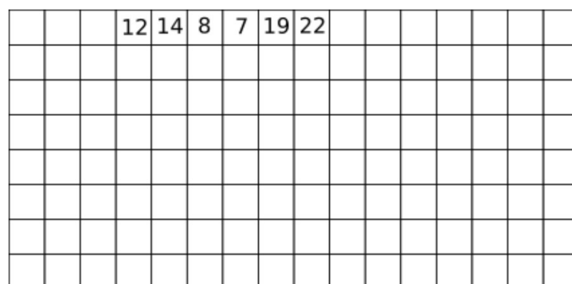
Chacune des deux implémentations rend plus ou moins efficaces (au sens de *coûteuses*) certaines des opérations.

Tableaux (dynamiques)

Un tableau est une suite contiguë de cases mémoires (les adresses mémoires se suivent).



Dans le cas d'une implémentation par un tableau dynamique, les éléments de la liste sont stockées dans ces cases mémoires contuguës. Comme un tableau possède une taille fixe, leur utilisation ne permet d'implémenter que des listes dont la taille maximale est définie à l'avance.

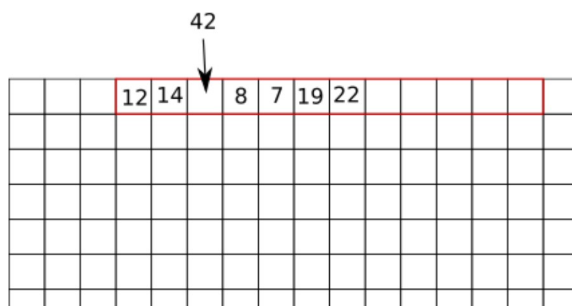


Pour pallier à cette limitation, il existe ce qu'on appelle des **tableaux dynamiques** qui sont des tableaux dont la taille peut varier en fonction des besoins. Une taille est définie au départ, et si le nombre d'éléments de la liste vient à dépasser celle-ci, il faut alors créer un tableau plus grand (le double en général) et y recopier tous les éléments du premier ainsi que la donnée supplémentaire au bon endroit.

> Le type `list` de Python correspond en fait à un tableau dynamique dans la majorité des implémentations Python. Malgré ce nom ambiguë, il s'agit donc d'un type abstrait beaucoup plus complet que le type `Liste`.

Coût de quelques opérations

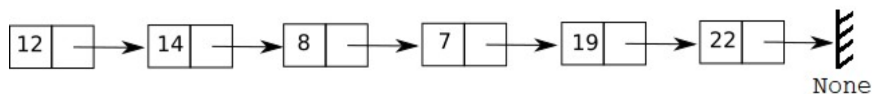
Si une liste `L` (de n éléments) est implémentée par un tableau, pour insérer un nouvel élément `e` en position `i` (opération `inserer(L, e, i)`) il faut commencer par déplacer tous les éléments à partir de `i` d'une case vers la droite. Cette opération est coûteuse car dans le *pire cas* (insertion en tête) il faut déplacer les n éléments de `L` d'un "cran" vers la droite : le coût est proportionnel à n, on parle de complexité $O(n)$.



En revanche, l'accès à un élément en position `i` (opération `lire(L, i)`) se fait de manière directe donc est peu coûteuse : elle se fait en temps constant ($O(1)$) car on accède à la case mémoire du tableau contenant l'élément en position `i` en ajoutant la valeur i à l'adresse mémoire de la première case (cela revient à faire une simple addition).

Listes chaînées

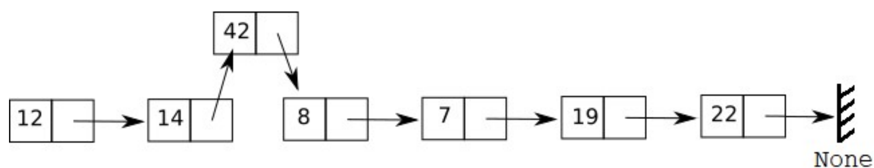
L'autre manière classique d'implémenter des listes est d'utiliser ce qu'on appelle des **listes chaînées**. A la différence d'un tableau, il s'agit d'une représentation *non contiguë*. En effet, on utilise alors des *maillons* comportant chacun un élément et une référence au suivant (l'adresse mémoire du suivant), ces maillons pouvant être éloignés les uns des autres en mémoire. Le dernier maillon contient le dernier élément et une référence `None` vers le suivant puisqu'il n'y a pas.



> Il s'agit d'une implémentation récursive d'une liste, qui se rapproche de celle avec des couples.

Coût de quelques opérations

Si une liste `L` est implémentée par une liste chaînée, l'opération `inserer(L, e, i)` (et l'opération `supprimer(L, i)`) est beaucoup moins coûteuse quand l'insertion (la suppression) est proche du début de la liste : il faut atteindre la position `i` puis insérer un nouveau maillon en jouant sur les références au suivant.



En revanche, comme la structure est récursive, pour accéder à un élément en position `i`, il faut parcourir la chaîne de maillon en maillon en commençant par le premier. L'opération `lire(L, i)` est donc coûteuse puisque dans le pire cas (accès au dernier élément), il faut parcourir les n maillons de la liste de proche en proche : la complexité est donc $O(n)$.

> **Moralité** : l'implémentation choisie entraîne des coûts différents pour les opérations sur la structure. Le choix est donc à faire selon le contexte et les besoins algorithmiques.

Bilan

- Nous avons que la structure de données *liste* permet de regrouper des éléments de manière ordonnée puisque chacun d'eux est repéré par sa position dans la liste.
- Il est possible de définir le type abstrait `Liste` en spécifiant son interface avec 5 *opérations primitives* qui permettent de reconstruire toutes les autres. Pour faciliter le travail des utilisateurs, on rencontre souvent des interfaces avec davantage d'opérations pour ne pas avoir besoin de les écrire soi-même.
- Les implémentations sont nombreuses et permettent selon les cas, de créer des listes *non mutables* ou *mutables*. Pour des raisons de coût mémoire, les implémentations les plus classiques permettent de créer des listes *mutables* permettant des transformations *en place* de celles-ci.
- Selon l'implémentation choisie, les opérations peuvent avoir des différences importantes en terme de complexité algorithmique des opérations. Il faut donc tenir compte du contexte pour privilégier ou limiter l'usage d'une implémentation.
- Nous aurons l'occasion de reparler des listes à d'autres moments au cours de l'année.

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe JERMANN et Christophe DECLERCQ.
- Document de Frédéric Mouton sur l'implémentation du type `list` de Python et les complexités des opérations : ici.
- Cours de David Roche pour les illustrations : ici.

Germain BECKER, Lycée Mounier, ANGERS

Ressource éducative libre distribuée sous Licence Creative Commons Attribution - Pas d’Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International