Chapitre III

Structures de données

1 Piles et files

1.1 Définitions

Les piles (anglais : stack) et les files (anglais : queue) sont des suites finies d'éléments telles qu'on ne peut ajouter ou retirer un élément qu'en tête ou en queue. De plus, elles suivent le principe suivant :

- Les **piles** sont sur le modèle FILO (pour First-In-Last-Out) ou LIFO (pour Last-In-First-Out), qui sont équivalents. Le prochain élément à sortir est le dernier arrivé, cet élément est appelé **sommet** (top).
  Il s'agit du fonctionnement d'une pile d'assiettes : la dernière assiette posée est la première que l'on enlève.
- Les **files** sont sur le modèle FIFO (pour First-In-First-Out) ou LILO (pour Last-In-Last-Out), qui sont équivalents. Le prochain élément à sortir est le premier arrivé, il est appelé **tête** (head), le dernier arrivé s'appelle **queue** (tail).
  Il s'agit du fonctionnement d'une file d'attente : les premières personnes arrivées sont les premières à sortir de la file.

1.2 Implémentation des piles

Pour une pile, on implémente communément les primitives suivantes :

- `initialize` : initialise une pile vide.
- `is_empty` : teste si la pile est vide.
- `push` (traduction anglaise de empiler) : ajout d'un élément à la pile.
- `pop` (traduction anglaise de dépiler) : retrait d'un élément à la pile.
  La fonction `pop` retourne l'élément qui a été retiré de la pile.
- `size` : renvoie le nombre d'éléments présents dans la pile.

L'objectif est d'implémenter les primitives `pop` et `push` à temps constant.

On suppose que les objets que l'on veut empiler sont d'une classe nommée `Item`.

On désigne par `nil` la valeur d'une variable qui ne référence aucun objet. En Python, `nil` est la valeur `None`.

1.2.1 Implémentation des piles comme tableau

C'est réalisable à peu près à l'aide d'un tableau. Pour mémoire, dans un tableau, tous les éléments sont stockés à des emplacements mémoire successifs. On se souviendra aussi que les listes Python sont implémentées sous forme de tableau.

Nous définissons une classe `Stack` contenant 3 attributs :

Algorithme 1.

```
class Stack
    maxsize :  entier  # taille du tableau
    top :  entier # indice du sommet de la pile
    items :  array of Item    #tableau contenant la pile
```

On considère que les indices des tableaux commencent à 0.

Pour la liste vide, on définit `top=-1`. La fonction d'initialisation d'une pile est :

Algorithme 2.

```
fonction initialize(Stack s, Integer size)
    s.items = new Array of Item that length is size
    s.maxsize = size
    s.top = -1
```

L'opération dépiler ne présente alors pas de difficulté puisque le nombre d'éléments stockés diminue.

Algorithme 3. Dépiler

```
fonction pop(Stack s)
    si top == -1 alors
        retourner nil
    sinon
        r = s.items[s.top]
        s.top = s.top - 1
        retourner r
```

L'opération empiler pose problème si on atteint la taille du tableau.

Algorithme 4. Empiler

```
fonction push(Stack s, Item x):
    si s.top+1 >= s.maxsize alors
         lever l'exception stack overflow
    sinon
        s.top = s.top + 1
        s.items[s.top] = x
```

Dans cette implémentation, les deux primitives `pop` et `push` sont à temps constant. L'inconvénient est qu'on a imposé une taille maximale à la pile.

Si on ne veut pas limiter la taille de la pile, il faut augmenter l'espace de stockage, ce qui peut imposer de déplacer la totalité des éléments du tableau. `push` n'est plus alors à temps constant.

Algorithme 5. Empiler

```
fonction push(Stack s, Item x):
    si s.top+1 >= s.maxsize alors
        choisir new_maxsize
        t = new Array of new_maxsize Item
        t = copy(s.items, s.top)   # complexité O(length(s))
        s.items=t
        s.maxsize=new_maxsize

   s.top = s.top + 1
   s.items[s.top] = x
```

Comment choisir la taille `new_maxsize` du nouveau tableau ? Une idée est de la doubler. Il s'agit d'optimiser d'une part le coût d'un déplacement, d'autre part l'espace mémoire occupé.

Vis-à-vis de la complexité temporelle, un bon choix est suivre une progression géométrique pour la longueur du tableau.

Faisons un calcul avec une raison 2, dans le pire des cas où on empile successivement n éléments, sans les dépiler, en partant d'un tableau de taille 1, il y aurait $p = \lceil \log_2 n \rceil$ déplacements et recopies du tableau, chacun d'un coût linéaire.

La complexité totale des déplacements et recopies serait $O(\sum_{k=1}^{p} 2^{k-1}) = O(2^p - 1) = O(n)$. En effet, pour le premier déplacement du tableau, il faut recopier $1 = 2^0$ valeur, pour le deuxième $2 = 2^1$, le troisième 2^2 et ainsi de suite.

La complexité moyenne des déplacements et recopies de tableau serait en $O(n)/n = O(1)$. On obtiendrait une complexité moyenne à temps constant.

Python permet d'implémenter directement les files comme des listes avec les méthodes `pop` et `append`.

1.2.2 Implémentation des piles comme liste chaînées

On crée une classe `Frame` définie par deux attributs :

Algorithme 6.

```
class Frame
    data : Item
    next : Frame ou nil
```

Nous définissons une classe `Stack` contenant deux attributs :

Algorithme 7.

```
class Stack
    top :  Frame ou nil
    size :  entier
```

Algorithme 8. Initialisation d'une pile vide

```
fonction initialize(Stack s)
    s.top = nil
    s.size = 0
```

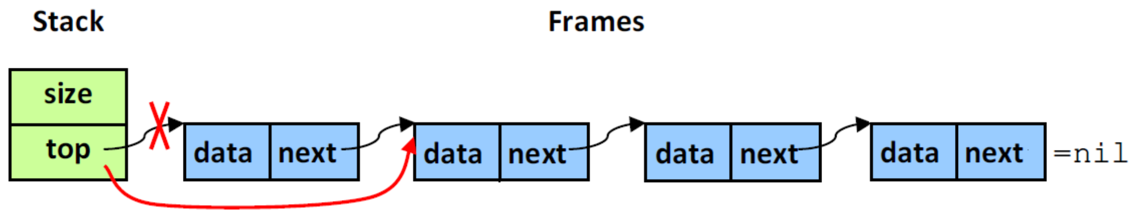


schéma de l'opération dépiler si la pile n'est pas vide

Algorithme 9. Dépiler

```
fonction pop(Stack s)
    si s.top == nil alors
        return nil
    r = s.top.data
    s.top = s.top.next
    s.size = s.size - 1
    return r
```

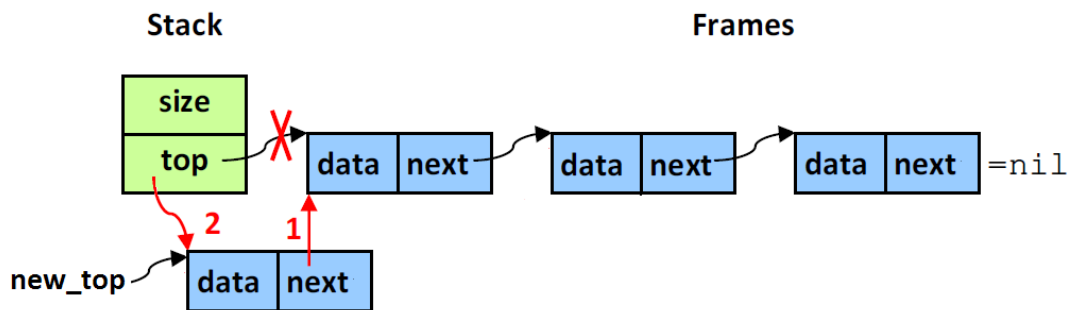


schéma de l'opération empiler

Algorithme 10. Empiler

```
fonction push(Stack s, Item x)
    new_top = new Frame
    new_top.data = x
    new_top.next = s.top  # opération 1 du schéma
    s.top = new_top       # opération 2 du schéma
    s.size = s.size + 1
```

Dans cette implémentation, les deux primitives `pop` et `push` sont à temps constant.

L'avantage des listes est qu'il n'y a plus de taille maximale. Par contre, on perd en espace mémoire (complexité spatiale).

1.3 File

Dans une file, on implémente communément les primitives suivantes :

- `initialize` : initialise une pile vide.
- `is_empty` : teste si la file est vide.
- `enqueue` (traduction anglaise de enfiler) : ajout d'un élément à la liste.
- `dequeue` (traduction anglaise de défiler) : retrait d'un élément à la liste.
  La fonction dequeue retourne l'élément qui a été retiré de la file.
- `length` : renvoie le nombre d'éléments présents dans la file.

L'objectif est ici aussi d'implémenter les primitives `enqueue` et `dequeue` à temps constant.

1.3.1 File comme tableau

Nous verrons en tp comment réaliser une file avec un tableau.

Python permet d'implémenter directement les files comme des listes avec les méthodes `popleft` et `append`.

1.3.2 Implémentation des piles comme liste chaînées

Pour implémenter les primitives `enqueue` et `dequeue` à temps constant sur des listes, il faut utiliser des listes doublement chaînées.

On crée une classe `Frame` avec deux attributs :

Algorithme 11.

```
class Frame
    data : Item
    next : Frame ou nil
    prev : Frame ou nil
```

Nous définissons une classe `Queue` contenant trois attributs :

Algorithme 12.

```
class Queue
    head : Frame or nil
    tail : Frame or nil
    size : entier
```

Initialisation d'une file vide

Algorithme 13. Initialisation d'une file vide

```
fonction initialize(Queue q)
    q.head=nil
    q.tail=nil
    q.size = 0
```

Algorithme 14. Défiler

```
fonction dequeue(Queue q)
    #si la liste est vide, on ne fait rien
    si q.head == nil alors
        retourner nil

    # sinon sauvegarde de l'élément défiler
    r = q.head.data
    # écriture du chainage aller : 1 cas
    q.head = q.head.next

    # écriture du chainage retour : 2 cas
    si q.head == nil alors
        # si la liste devient vide
        q.tail=nil
    sinon
        # sinon q.head.prev est valide
        q.head.prev = nil
    q.size = q.size - 1

    return r
```

Algorithme 15. Enfiler

```
fonction enqueue(Queue q, Item x)
    new_tail = new Frame
    new_tail.data = x

    # écriture du chainage aller : 2 cas
    new_tail.next = nil
    si q.tail == nil alors
       # si la file est vide, q.head doit pointer sur new_tail
       q.head= new_tail
    sinon
        # Sinon, q.tail.next est valide et doit pointer sur new_tail
        q.tail.next = new_tail

    # écriture du chainage retour : 1 cas
    new_tail.prev = q.tail
    q.tail=new_tail
    q.size = q.size + 1
```

Dans cette implémentation, les deux primitives `enqueue` et `dequeue` sont à temps constant.

2 Arbres binaires

2.1 Généralités

Définition : arbres binaires

Soit D un ensemble de données, les arbres binaires sur D peuvent être définis récursivement ainsi :

- `nil` est un arbre binaire appelé arbre binaire vide.
- Si A_g et A_d sont des arbres binaires et $data \in D$, alors $(data, A_g, A_d)$ est un arbre binaire.

Représentation :

On représente les arbres binaires vers le bas où chaque nœud correspond à un triplet dans la définition récursive ci-dessus. La racine est le premier noeud.
On ne représente pas les arbres et sous-arbres vides.

Définition : feuille

Un nœud qui n'a pas de sous-arbres (non-vides) est une feuille.

Définition : hauteur

La hauteur h d'un arbre est le nombre d'arêtes qu'il faut traverser dans le chemin le plus long de la racine à une feuille.

Définition : arbre plein

Un arbre est plein si tous les nœuds qui ne sont pas des feuilles ont deux enfants.

Définition : arbre complet

Un arbre est complet s'il est plein et que toutes ses feuilles sont à la même hauteur (la longueur du chemin de la racine à une feuille est constant).

Propriété : nombre de nœuds d'un arbre binaire

Un arbre de hauteur h possède entre $h+1$ (tous les nœuds sont sur une même branche) et $2^{h+1}-1$ nœuds (l'arbre est complet).

2.2 Parcours d'un arbre binaire

Remarque : tous les algorithmes de ce chapitre sont écrits pour des arbres non vides, il est facile de les adapter pour qu'ils acceptent l'arbre vide.

On nomme `Item` la classe des objets que l'on stocke dans l'arbre.

Un arbre binaire peut-être implémenté comme une classe `Tree` possédant un attribut :

Algorithme 16.

```
class Tree
    root : Node ou nil
    size : Integer   # nombre de nœuds, parfois utile
```

Pour les noeuds, on crée la classe `Node` possédant trois attributs :

Algorithme 17.

```
class Node
   data : Item
   left : Node ou nil
   right : Node ou nil
```

On suppose que `visit(Node n)` est une fonction qui effectue une action sur le champ `data`.

On peut parcourir facilement un arbre binaire par des algorithmes récursifs.

2.2.1 Parcours préfixe

Dans un ordre préfixe, chaque nœud est visité avant chacun de ses enfants.

Algorithme 18. Parcours récursif préfixe

```
fonction visiter_prefixe(Node n)
    visit(n)
    si n.left != nil alors
          visiter_prefixe(n.left)
    si n.right != nil alors
          visiter_prefixe(n.right)
```

Remarque : Ici, il faut vérifier que l'arbre n'est pas vide avant de lancer le premier appel récursif. Sinon on récupère un nœud `nil` et `n.left` n'est pas valide.

On peut aussi écrire la fonction comme suit :

Algorithme 19. Parcours récursif préfixe

```
fonction visiter_prefixe(Node n)
    si n!= nil alors
        visit(n)
        visiter_prefixe(n.left)
        visiter_prefixe(n.right)
```

Remarque : Ici, on fait des appels récursifs même pour des nœuds nil, donc l'efficacité est moins bonne

2.2.2 Parcours postfixe ou suffixe

Dans un parcours postfixe ou suffixe, on visite chaque nœud après avoir visité chacun de ses enfants.

Algorithme 20. Parcours récursif postfixe

```
fonction visiter_postfixe(Node n)
    si n!= nil alors
        visiter_postfixe(n.left)
        visiter_postfixe(n.right)
        visit(n)
```

2.2.3 Parcours infixe

Dans un parcours infixe, on visite chaque nœud après son enfant gauche et avant son enfant droit.

Algorithme 21. Parcours récursif infixe

```
visiter_infixe(Node n)
    si n!= nil alors
        visiter_infixe(n.left)
        visit(n)
        visiter_infixe(n.right)
```

2.2.4 Parcours en profondeur itératif

Lors d'un parcours en profondeur, il s'agit d'aller au bout de chaque branche avant de passer à la suivante. Les parcours préfixe, postfixe et infixe sont des parcours en profondeur.

Pour effectuer ce parcours de manière itérative, il est nécessaire de stocker les éléments en attente de traitement dans une pile.

Parcours en profondeur préfixe itératif pour un arbre non vide

Algorithme 22. Parcours en profondeur itératif

```
fonction parcours_profondeur(Tree t)
   p = new Stack
   initialize(p)
   si t.root != nil alors
       push(p,t.root)
   tant que is_empty(p)==false
       n = pop(p)
       visit(n)
       si n.right != nil alors
           push(p, n.right)
       si n.left != nil alors
           push(p, n.left)
```

ici on a choisi de ne pas empiler les nœuds qui valent nil.

On peut écrire un code plus court mais **moins efficace** en empilant les nœuds nil.

Algorithme 23. Parcours en profondeur itératif

```
fonction parcours_profondeur(Tree t)
   p = new Stack
   initialize(p)
   tant que is_empty(p)==false
       n = pop(p)
       si n!=nil alors
           visit(n)
           push(p, n.right)
           push(p, n.left)
```

Ce code est moins efficace puisqu'on fait **autant de comparaisons avec nil mais significativement plus d'opérations empiler et dépiler** que dans le code précédent.

2.2.5 Parcours en largeur itératif

Maintenant, on veut parcours chaque niveau de l'arbre de gauche à droite avant de passer au niveau suivant. D'où le nom parcours en largeur.

L'implémentation est quasiment identique mais il faut cette fois utiliser une file de `Node`.

Parcours en largeur pour un arbre non vide

Algorithme 24. Parcours en largeur itératif

```
fonction parcours_largeur(Tree t)
   f = new Queue
   initialize(f)
   si t.root != nil alors
       enqueue(f,t.root)
   tant que not is_empty(f)
       n= dequeue(f)
       visit(n)
       si n.left != nil alors
           enqueue(f, n.left)
       si n.right != nil alors
           enqueue(f, n.right)
```

2.3 Arbres binaires de recherche

On suppose maintenant que le domaine D des données dispose d'une relation d'ordre total.

On cherche une structure de données permettant de stocker un ensemble de n éléments de D et effectuant en particulier les opérations suivantes de manière efficace :

- insertion d'un élément
- test d'appartenance
- élimination d'un élément de A

Si l'on utilise une liste non-ordonnée, la recherche coûte $O(n)$.

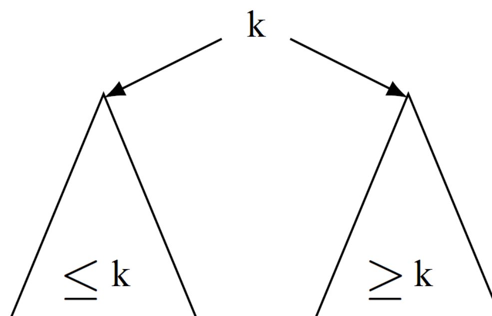
Si on utilise une liste ordonnée, on peut faire une recherche par dichotomie à condition d'avoir un accès à temps constant au i-ème élément de la liste, ce qui élimine les listes chaînées. Mais pour les tableaux triés, les coûts de l'insertion et de la suppression d'une valeur sont en $O(n)$.

Avec les arbres binaires de recherche, ces opérations sont en $O(h)$ où h est la hauteur de l'arbre qui représente l'ensemble.

Définition : arbres binaires de recherche (binary search trees en anglais, ABR en abrégé)

Un ABR est un arbre binaire qui vérifie en tout nœud n :

- pour tout $n' \in n.left$, $n'.data \leqslant n.data$. Autrement dit, la donnée de n est supérieure à la donnée de tous les nœuds du sous-arbre gauche.
- pour tout $n' \in n.right$, $n'.data \geqslant n.data$. Autrement dit, la donnée de n est inférieure à la donnée de tous les nœuds du sous-arbre droit.



Définition d'un ABR

On utilise les ABR pour stocker des données telles que les **clés de tri sont uniques** (autrement dit, deux données distincts ont des clés distinctes). **On ajoute cette hypothèse par la suite de ce chapitre**.

Un arbre binaire peut-être implémenté comme une classe `BST` dérivée de `Tree`

Avec l'exception de l'opération d'impression qui est linéaire dans la taille de l'ABR, toutes les autres opérations sont linéaires dans la hauteur de l'ABR.

Impression : principe

L'impression par ordre croissant d'un ABR correspond à un parcours infixe.

Exercice : écrire l'algorithme.

Insertion : principe

Pour insérer un élément x, on navigue dans l'ABR jusqu'à trouver :

- soit x : on ne fait rien.
- soit la feuille `nil` où il faut placer x.

Exercice : écrire l'algorithme.

Appartenance : principe

Pour déterminer si un élément x est dans l'ABR, on navigue dans l'arbre jusqu'à trouver :

- soit x : on retourne le nœud.
- soit nil : on retourne `Nil`.

Exercice : écrire l'algorithme.

Minimum : principe

Pour trouver le minimum de l'ABR, on suit toujours le branchement à gauche jusqu'à trouver un nœud dont l'enfant gauche est nil.

Exercice : écrire l'algorithme.

Elimination du minimum : principe

Pour éliminer l'élément minimum, on suit le branchement gauche jusqu'à trouver un nœud dont l'enfant gauche est nil. Ensuite on remplace ce nœud par son enfant droit (il peut être nil).

Exercice : écrire l'algorithme.

Elimination : principe

Pour éliminer un élément x dans l'ABR, on cherche x dans l'arbre jusqu'à trouver :

- soit nil et on ne fait rien.
- soit x et on distingue 3 cas :
  1) x a 0 enfant. Le parent de x va pointer vers nil.
  2) x a 1 enfant. Le parent de x va pointer vers l'enfant de x.
  3) x a 2 enfants. On retire le minimum du sous-arbre droit de x et le met à la place de x. Le sous-arbre gauche de x n'est pas modifié.

Exercice : écrire l'algorithme.

Une solution symétrique qui modifie le sous-arbre gauche est possible. Il est aussi possible de combiner en une seule opération la recherche du minimum et son élimination.

D'autres opérations comme la recherche de l'élément maximum et la recherche du successeur (ou du prédécesseur) d'un élément donné peuvent être réalisées en suivant les mêmes idées.

2.4 Tas

La structure de tas fera l'objet d'un TP. Ce chapitre n'est qu'une introduction pour ce futur TP.

Définition : sous-arbre enraciné

Soit T un arbre et n un nœud de T, on appelle (sous-)arbre enraciné en n, le sous-arbre dont n est la racine.

Numérotation des nœuds d'un arbre binaire

On peut numéroter les positions possibles des nœuds dans un arbre binaire de la racine vers les feuilles et de gauche à droite.
On choisit ici de numéroter à partir de 0 pour simplifier le codage en Python. On a alors les numéros suivants :

- 0 pour la racine,
- 1, 2 pour le niveau 1,
- 3, 4, 5, 6 pour le niveau 2,
- 7, 8, …, 14, pour le niveau 3,
- …
- $2^n - 1$ à $2^{n+1} - 2$ pour le niveau n.

Justification

Comme chaque niveau k dispose exactement de 2^k positions, il y a $\sum_{k=0}^{n-1} 2^k = 2^n - 1$ positions avant le niveau n. Comme on numérote à partir de 0, le premier numéro du niveau n est $2^n - 1$.

Définition : arbre binaire quasi-complet

Un arbre binaire avec n nœuds est quasi-complet si ses nœuds occupent exactement les positions numérotées 0 à $n - 1$.

Remarque : Un arbre quasi-complet est soit complet, soit ses feuilles sont sur les deux derniers niveaux, celles du dernier niveau étant plus à gauche que celles de l'avant-dernier niveau.

Propriété : stockage d'arbre binaire quasi-complet dans un tableau

Un arbre binaire quasi-complet avec n nœuds peut être stocké de manière efficace dans un tableau de longueur n, en identifiant les indices dans le tableau avec les positions dans l'arbre.
On a alors les propriétés suivantes :

- Les enfants du nœud d'indice i sont aux indices $2i + 1$ (si $2i + 1 < n$) et $2i + 2$ (si $2i + 2 < n$).
- Le parent du nœud d'indice $i \geqslant 1$ est à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$.

Définition : tas-min (tas-max)

Un tas-min (tas-max) est un arbre quasi-complet tel que chaque nœud a une valeur inférieure ou égale (respectivement supérieure ou égale) à celles de ses enfants.

Dans le TP, on écrira les algorithmes de manipulation des tas ainsi qu'un algorithme de tri par tas parmi les plus performants parmi les tris par comparaison.