Graphes : Parcours en profondeur (DFS)

Quentin Fortier

May 16, 2022

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.
2. **Parcours en largeur** (ou BFS pour Breadth-First Search) : on visite les sommets par distance croissante depuis un sommet de départ.

Parcours de graphe

On a souvent besoin de parcourir les sommets d'un graphe un par un. Voici les deux principaux parcours, visitant les sommets de proche en proche :

1. **Parcours en profondeur** (ou DFS pour Depth-First Search) : on visite les sommets le plus profondément possible avant de revenir en arrière.
2. **Parcours en largeur** (ou BFS pour Breadth-First Search) : on visite les sommets par distance croissante depuis un sommet de départ.

Si le graphe est connexe, tous les sommets sont visités.
Sinon, on applique un parcours sur chacune des composantes connexes.

Parcours en profondeur (DFS)

Un **parcours en profondeur** sur un graphe $G = (V, E)$ depuis un sommet u consiste à visiter u puis visiter récursivement chaque voisin de u qui n'a pas déjà été vu.

Parcours en profondeur (DFS)

Un **parcours en profondeur** sur un graphe $G = (V, E)$ depuis un sommet u consiste à visiter u puis visiter récursivement chaque voisin de u qui n'a pas déjà été vu.

Il est nécessaire de se souvenir des sommets déjà visités pour éviter de faire une infinité d'appels récursifs (en revisitant toujours les mêmes sommets).
D'où l'utilisation d'une liste `visited` et d'une fonction auxiliaire récursive (de façon à avoir accès à `visited` dans tous les appels récursifs).

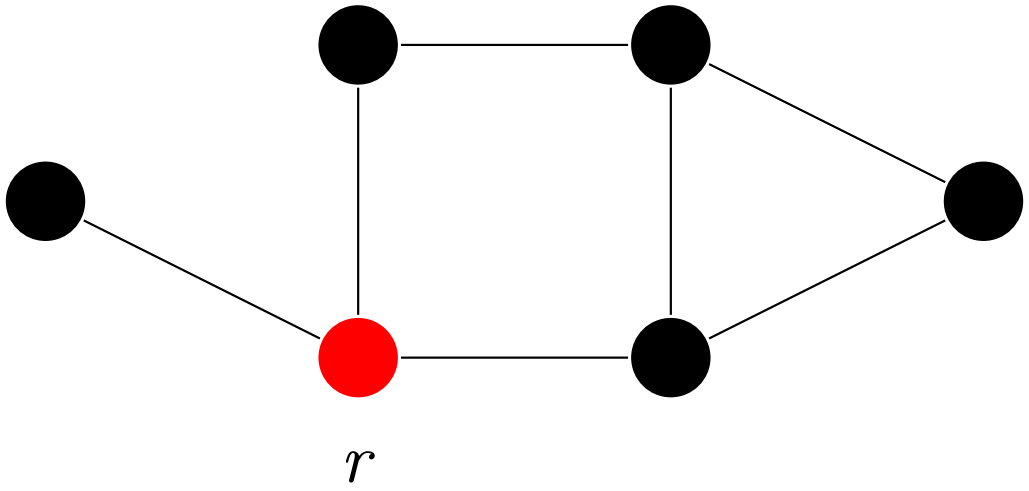
Parcours en profondeur (DFS)

Parcours en profondeur sur un graphe G représenté par liste d'adjacence, depuis un sommet s :

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

`visited[u]` vaut `True` si `u` a déjà été visité

Exemple



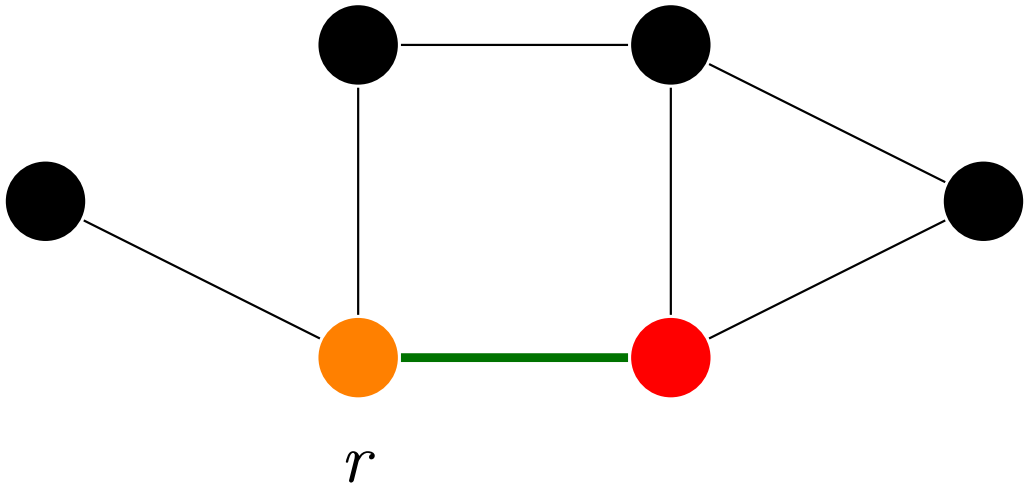
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



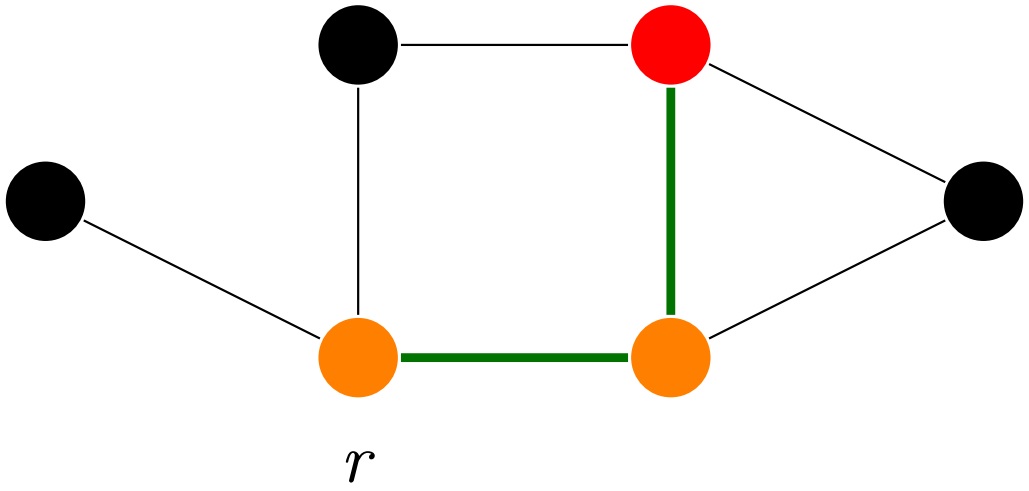
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



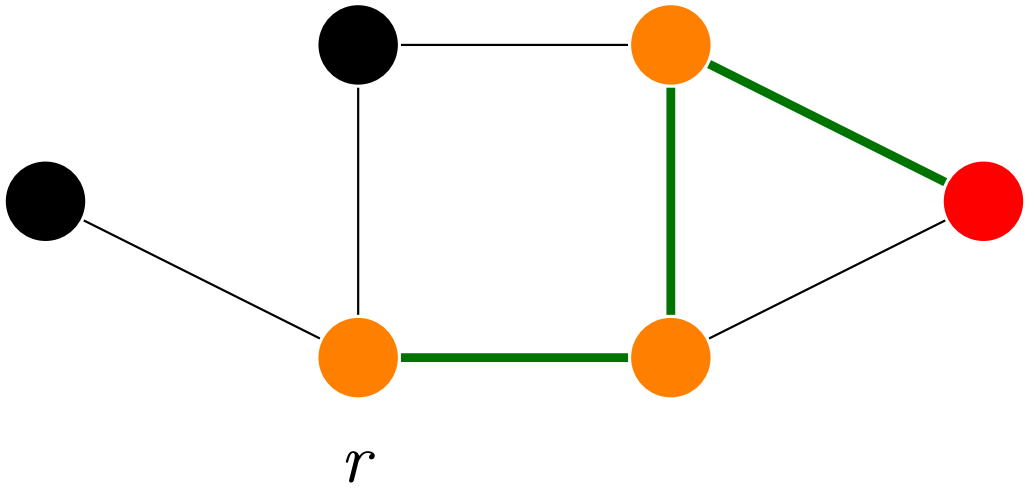
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



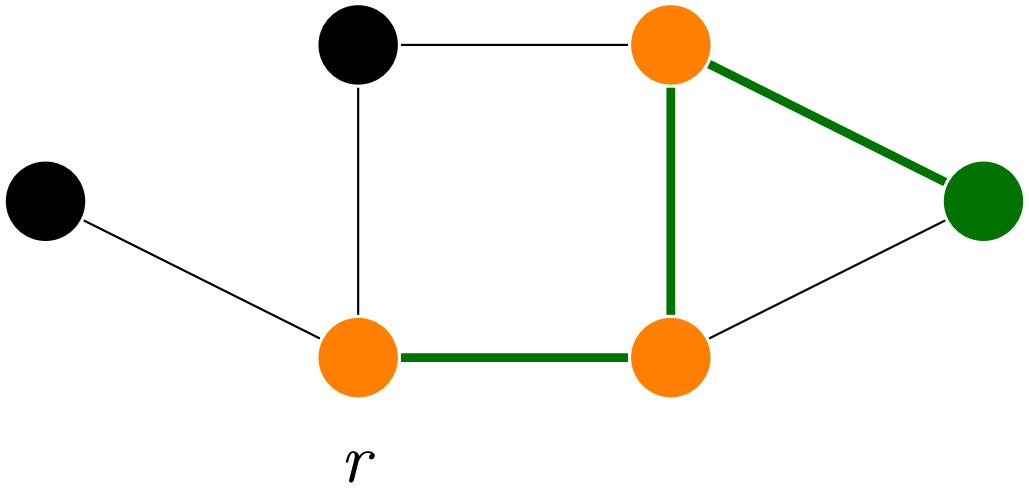
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



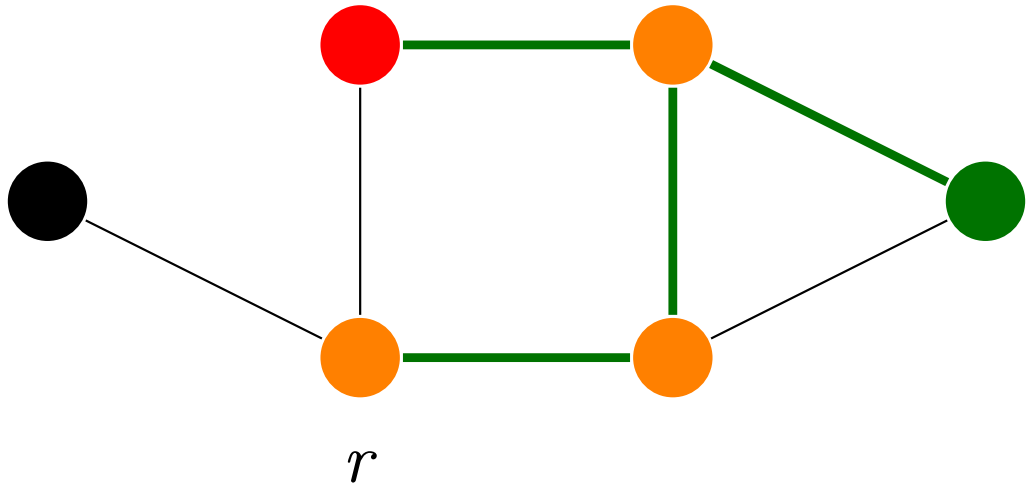
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



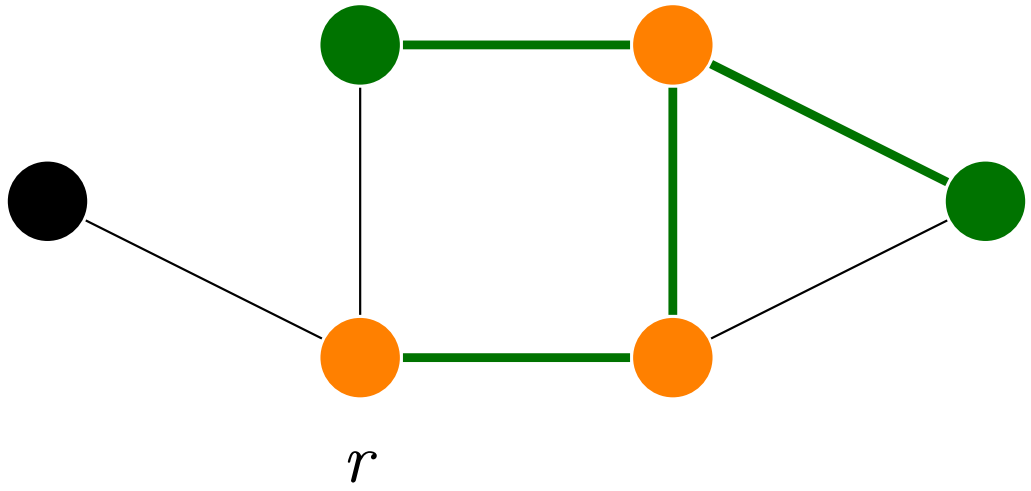
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



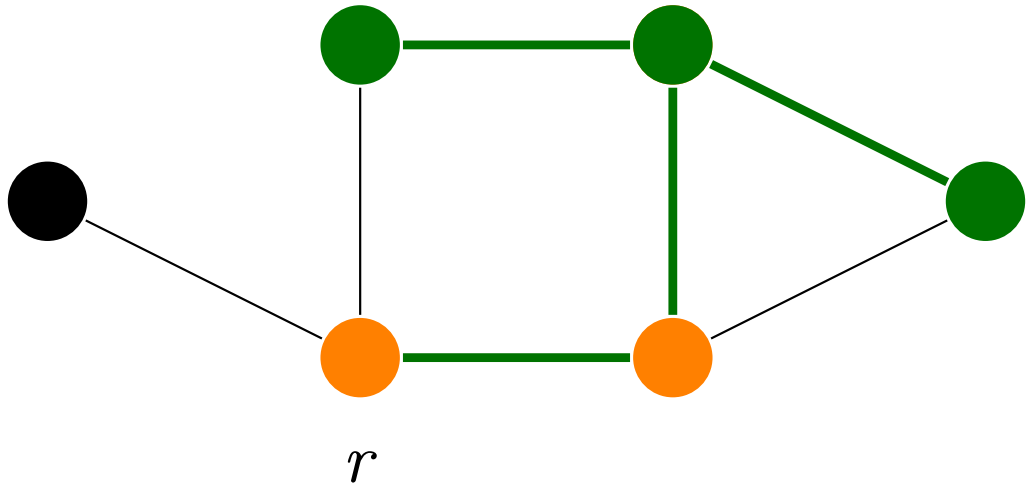
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



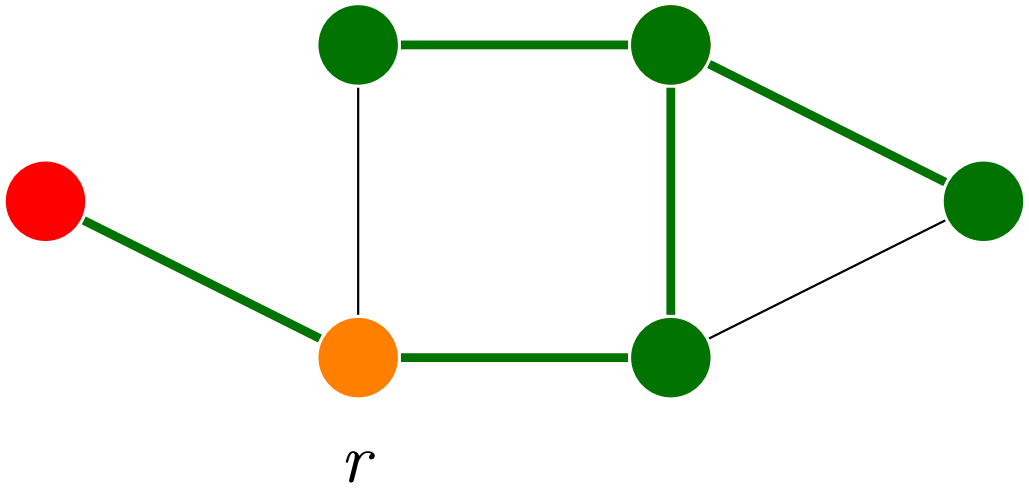
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



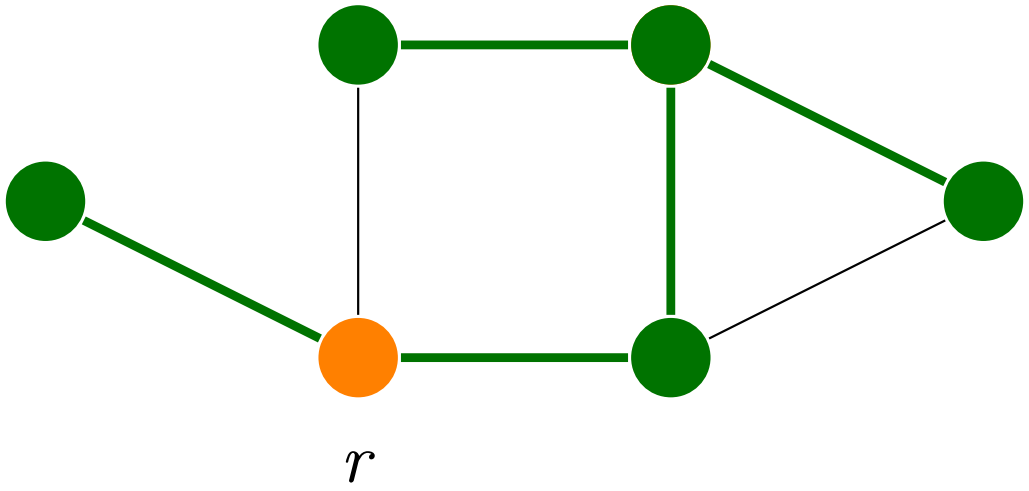
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



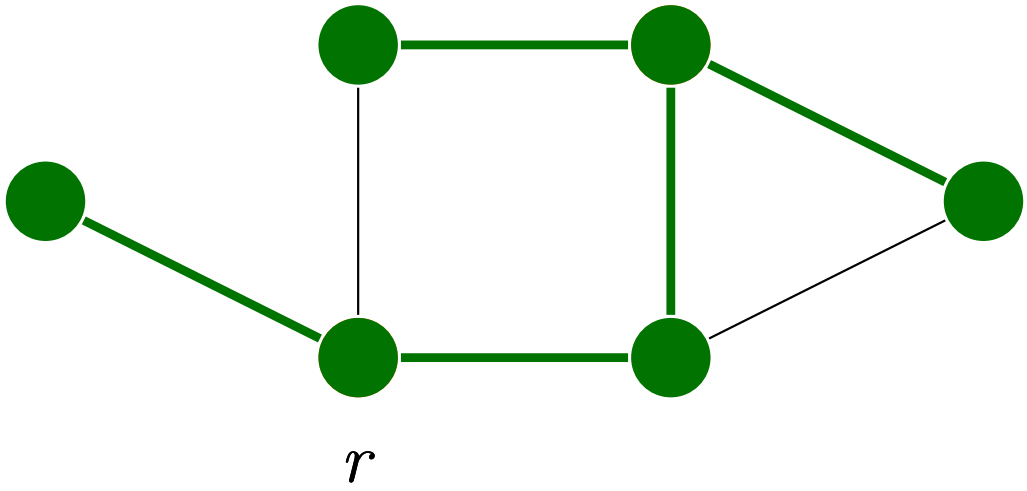
sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple



sommet pas encore visité

sommet en cours de traitement

appel récursif en pause

visite du sommet terminé

Exemple

Exercice

Adapter le code de la fonction ci-dessous pour afficher (avec `print`) les sommets dans l'ordre de leur visite dans le parcours en profondeur.

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    aux(s)
```

Exemple

Exercice

Écrire un parcours en profondeur pour un graphe représenté par matrice d'adjacence.

Application : connexité

Question

Comment déterminer si un graphe **non orienté** est connexe?

Application : connexité

Question

Comment déterminer si un graphe **non orienté** est connexe?

Il suffit de vérifier que le tableau `visited` ne contient que des `True`.

Application : connexité

Si le graphe n'est pas connexe, on peut effectuer un parcours sur chacune des composantes connexes :

```
def dfs(G, s):
    visited = [False]*len(G)
    def aux(u):
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                aux(v)
    for u in range(n):
        aux(u)
```

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père !

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père !

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

On regarde si on revient sur un sommet déjà visité...et que ce n'est pas un fils qui revient sur son père !

déjà visité

Application : Détection de cycle

Question

Comment déterminer si un graphe **non orienté** contient un cycle?

Il faut donc éviter de s'appeler récursivement sur son père :

```
def has_cycle(G, s):
    # renvoie True si G a un cycle atteignable depuis s
    visited = [False]*len(G)
    def aux(u, p): # p : sommet ayant permis de découvrir u
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                if v != p and aux(v, u):
                    return True
        return False
    return aux(s, -1)
```

DFS avec pile

Au lieu d'utiliser la récursivité, on peut aussi stocker les prochains sommets à visiter dans une liste, utilisée comme une **pile** :

```
def dfs(G, s): # G représenté par liste d'adjacence
    visited = [False]*len(G)
    pile = [s]
    while len(pile) > 0:
        u = pile.pop()
        if not visited[u]:
            visited[u] = True
            for v in G[u]:
                pile.append(v)
```

Pile

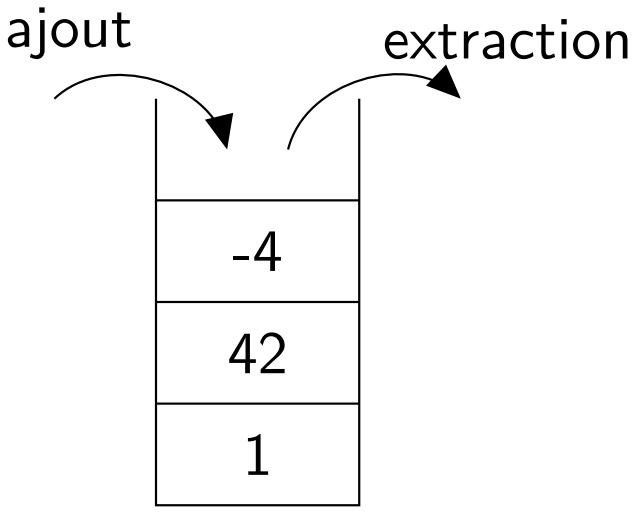
Une **pile** est une structure de donnée possédant les opérations :

- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.

Pile

Une **pile** est une structure de donnée possédant les opérations :

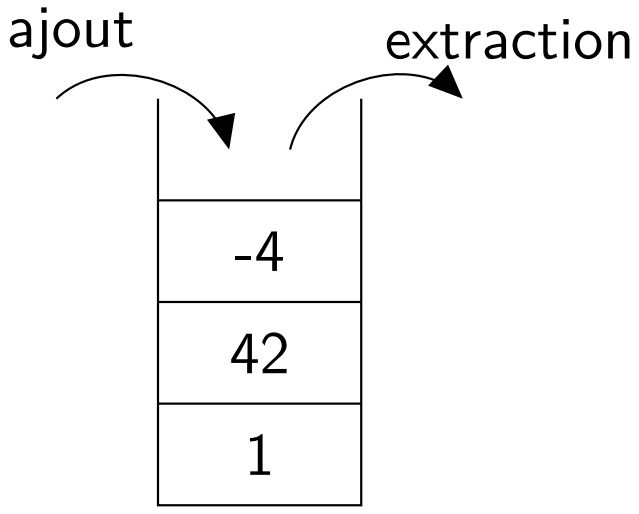
- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.



Pile

Une **pile** est une structure de donnée possédant les opérations :

- Ajout d'un élément au dessus de la pile.
- Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.



On peut implémenter une pile avec une liste `L` en Python, en utilisant `L.append(e)` pour ajouter un élément `e` et `L.pop()` pour l'extraction.

Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.

Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.
- Passer de récursif à itératif en simulant des appels récursifs. Par exemple, pour utiliser une pile au lieu d'une fonction récursive pour le parcours en profondeur.

Pile

- La *call stack* (pile d'appel) est une pile utilisée par le processeur pour stocker les appels de fonctions en cours d'exécution ainsi que leur arguments. L'erreur *stack overflow* signifie qu'il n'y a plus assez de mémoire dans la *call stack*.
- Passer de récursif à itératif en simulant des appels récursifs. Par exemple, pour utiliser une pile au lieu d'une fonction récursive pour le parcours en profondeur.
- Dans un éditeur de texte, chaque action est ajoutée à une pile. Lorsque vous revenez en arrière (Ctrl + Z), l'éditeur extrait le dessus de la pile pour revenir à l'état précédent.