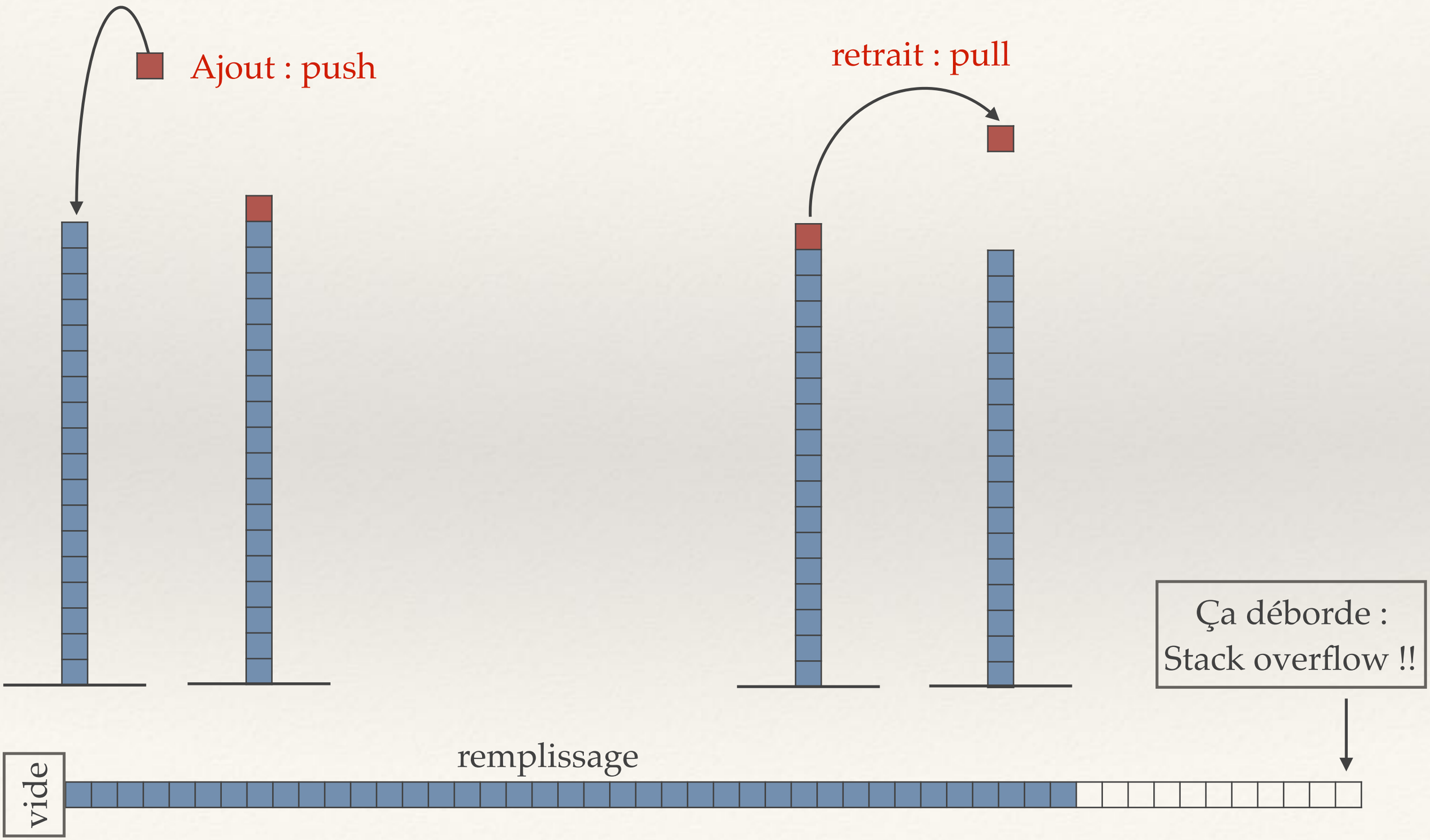
INFORMATIQUE

Structures de Données
Piles et Queues

1 - Structure de Pile

La structure de Pile, très courante, est caractérisée par deux opérations simples : push/pull
Selon la logique FILO « First In Last Out ».



Pour empêcher le débordement de Pile on peut mettre en place la logique d'allocation dynamique de mémoire.

Toutefois les opérations « bas-niveau », celles réalisées au niveau de l'électronique notamment, n'ont pas d'autre choix que d'avoir une taille de pile fixée, et on ne peut que constater un dépassement de capacité ou Stack OverFlow le cas échéant.

Structure de donnée minimale :

Pile

Attributs :

- hauteur
- hauteurMax

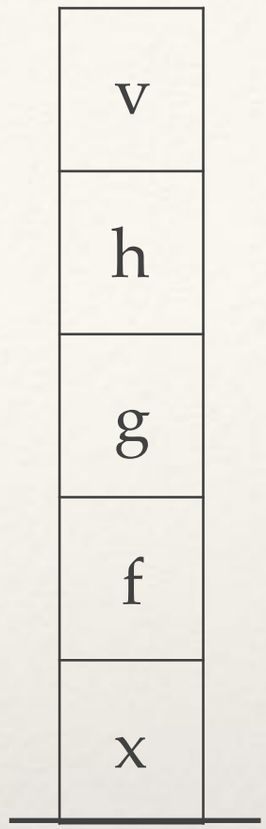
Méthodes :

- push(elem)
- pull()
- isEmpty()
- isFull()

Exemple de base :

-> La pile d'appels fonctionnels et les appels récursifs

x = **f**(**g**(**h**(variable **v**)))



1 - On empile les appels :

On ne peut pas évaluer tant qu'on a pas la variable finale

2 - On dépile en évaluant :

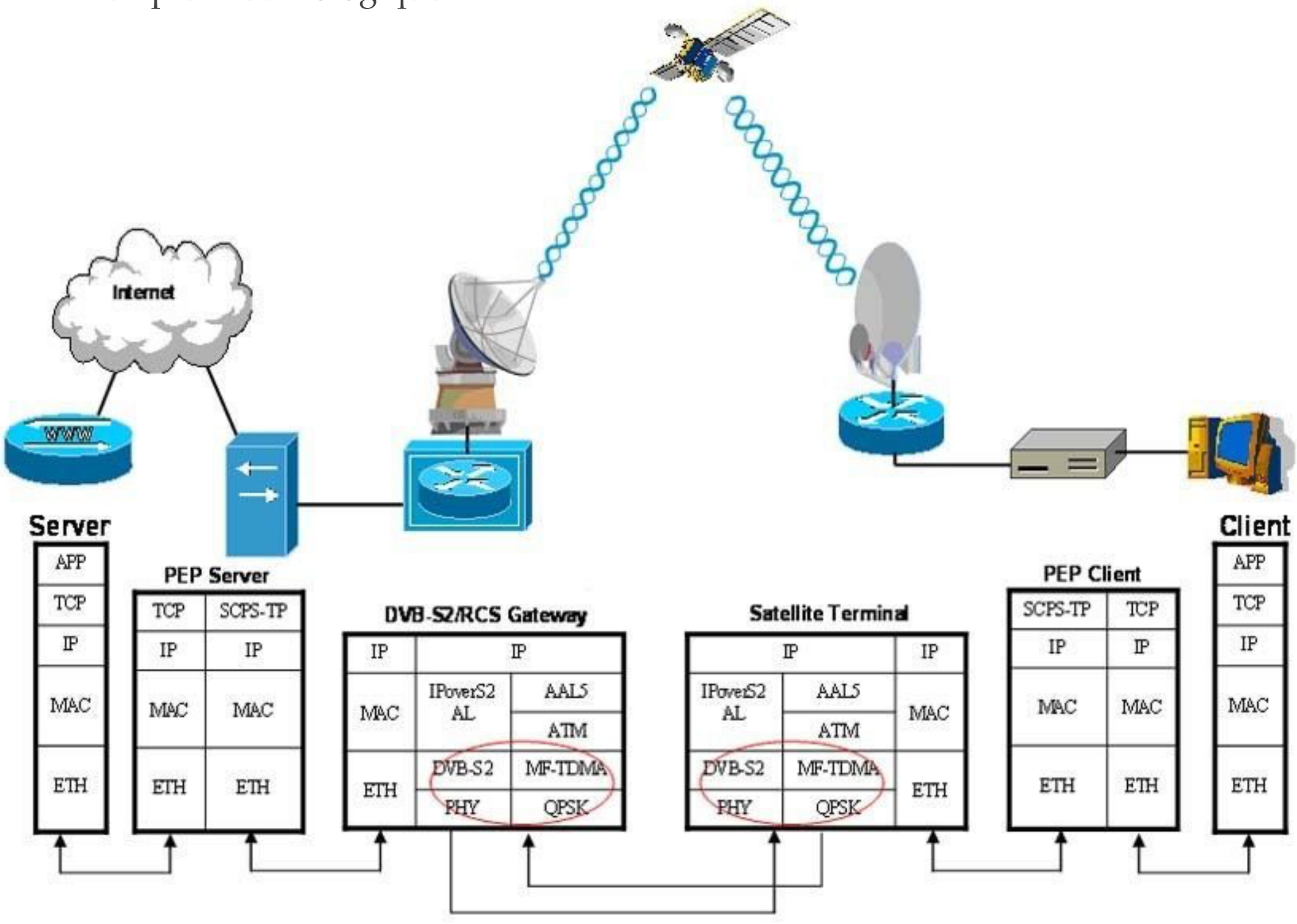
On peut libérer la mémoire des appels précédents

Récursivité : Dans le cas de la récursivité, on empile des appels à la même fonction jusqu'à atteindre la condition de terminaison. [On est limité à ~4000 appels]

Tout programme récursif peut être remplacé par un programme non récursif, qui réalise le même algorithme, mais il faudra introduire une pile annexe pour garder les résultats intermédiaires en mémoire [Technique du Petit Poucet…]

Le modèle en couche est une structure de Pile Protocolaire

Exemple « technologique »



Dijkstra : algorithme à deux piles

On souhaite créer un algorithme pour automatiser l'évaluation d'expressions formelles simples utilisant les 4 opérations telles que l'exemple ci-dessous :

(5*2+3)*4+1

Cette expression [chaîne de caractères] doit être re-travaillée pour faire apparaître toutes les parenthèses nécessaires. Soit :

((((5*2)+3)*4)+1)

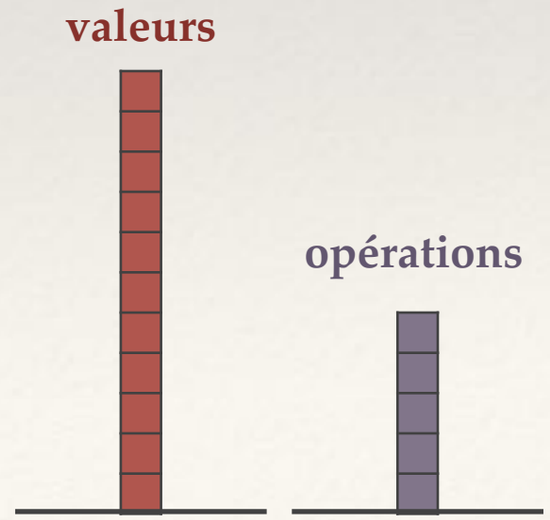
<u>Soit le séquentiel :</u>

(	(	(	(	5	*	2	)	+	3	)	*	4	)	+	1	)

L'algorithme consiste alors à balayer l'expression de gauche à droite en distinguant les valeurs et les opérations.

<u>On crée pour cela deux piles :</u>

- une pile de Valeurs qui stocke les valeurs rencontrées.
- une pile d'opérations stockant les opérations effectuées.



Au cours du balayage on applique l'algorithme de Dijkstra suivant sur les 2 piles :

Celui-ci procède en fonction du « caractère » rencontré :

Dijkstra

- **Valeur** : mettre sur la pile de valeur
- **Opération** : mettre sur la pile d'opération
- **Parenthèse gauche** : on passe
- **Parenthèse droite** : on fait l'opération entre les 2 dernières valeurs
  => on dépile 2 valeurs & 1 opération
  => on rempile le résultat sur la pile de valeur.

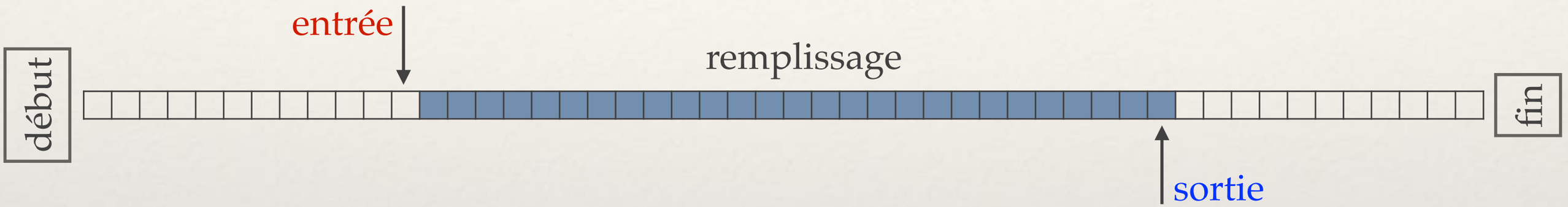
A vous de jouer :

(	(	(	(	5	*	2	)	+	3	)	*	4	)	+	1	)

2 - Structure de Queue

La structure de Queue, tout aussi courante, est caractérisée par deux opérations simples : push/pull mais cette fois selon la logique FIFO « First In First Out ».

Cette logique est privilégiée dès que l'ordre d'entrée des données représente un enjeu.



Tout ce qu'il faut ici c'est maintenir la connaissance de la case d'entrée et de la case de sortie :

- Si la taille de la mémoire est fixée on peut raisonner en modulo N, comme si le tableau était un anneau, jusqu'à ce que la queue soit pleine.

- Si l'allocation dynamique est possible, on peut recopier les données en partant de la fin lorsque la queue est pleine.

Structure de donnée minimale :

Queue

Attributs :

- Début
- Fin
- longueur
- longueurMax

Méthodes :

- enQueue(elem)
- deQueue()
- isEmpty()
- isFull()

Exemple simple :

La **saisie des caractères du clavier** dans un terminal se fait en remplissant une queue, car il faut absolument maintenir l'ordre de la saisie des caractères !
On parle de **buffer** ou **mémoire tampon** : en règle générale la taille du buffer (longueur max de la queue) est fixée car c'est une mémoire dédiée.

C'est la même chose pour récupérer les données de toutes les interfaces ou périphériques : Click - Click - Click de souris / acquisition audio / données reçues par la carte réseaux

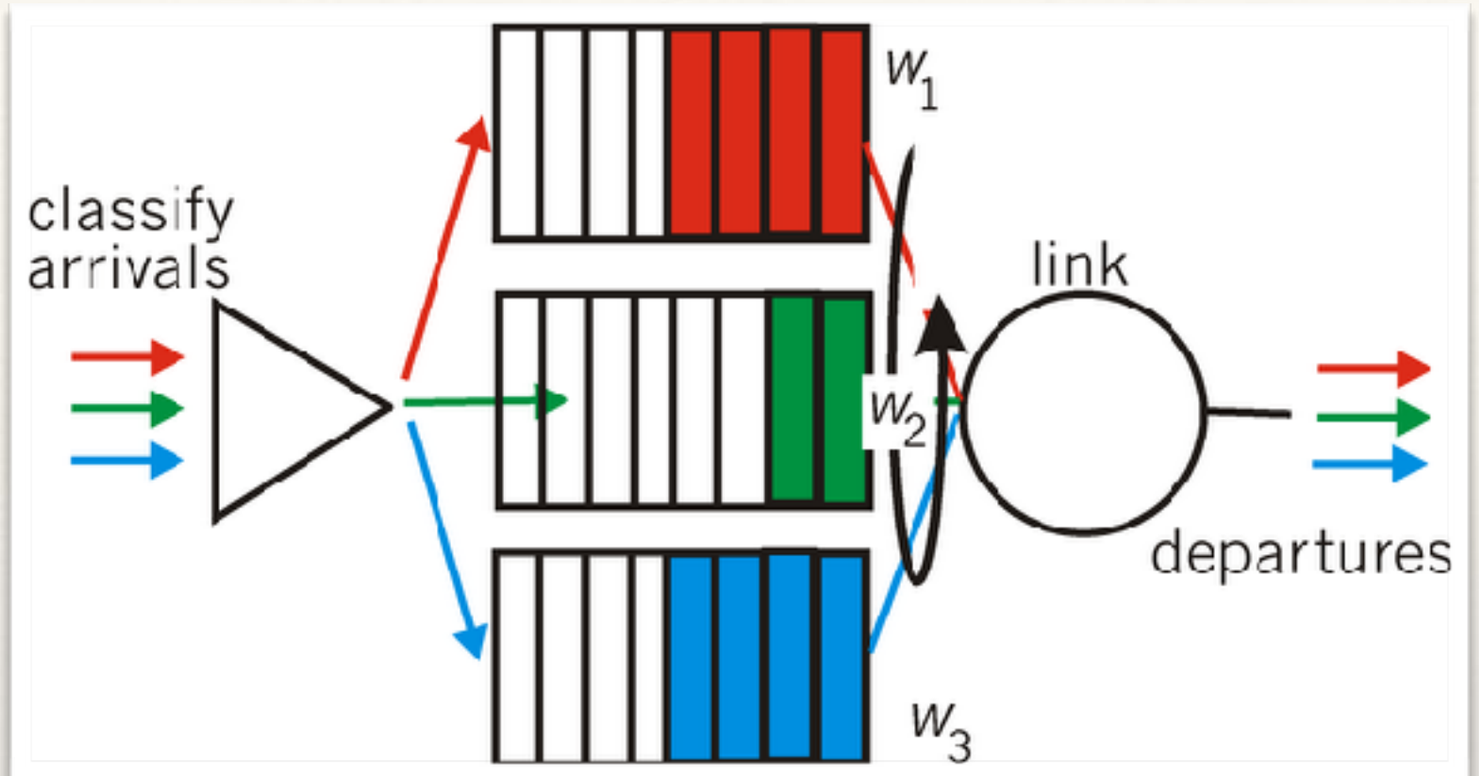
D'une part :
Les données arrivent beaucoup trop vite pour que le traitement se fasse immédiatement.
Il faut donc stocker ces données dans une mémoire à accès rapide.

D'autre part :
Il ne se passe généralement pas grand chose entre deux phases de réception, on parle d'activité « bursty » typique des réseaux.

Dans tous les cas la question qui se pose est celle de la **gestion du flux** :
Il faut pouvoir vider la queue avant qu'elle ne soit pleine !

La gestion des paquets dans un routeur implémente une structure de Queue

Schématiquement :



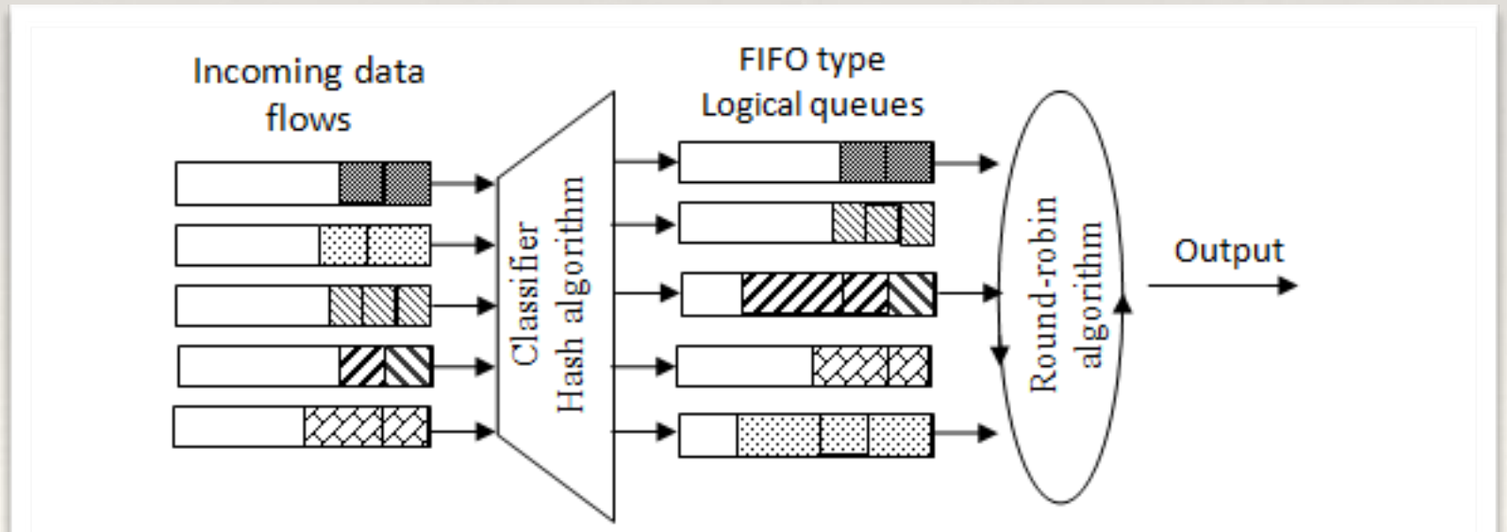
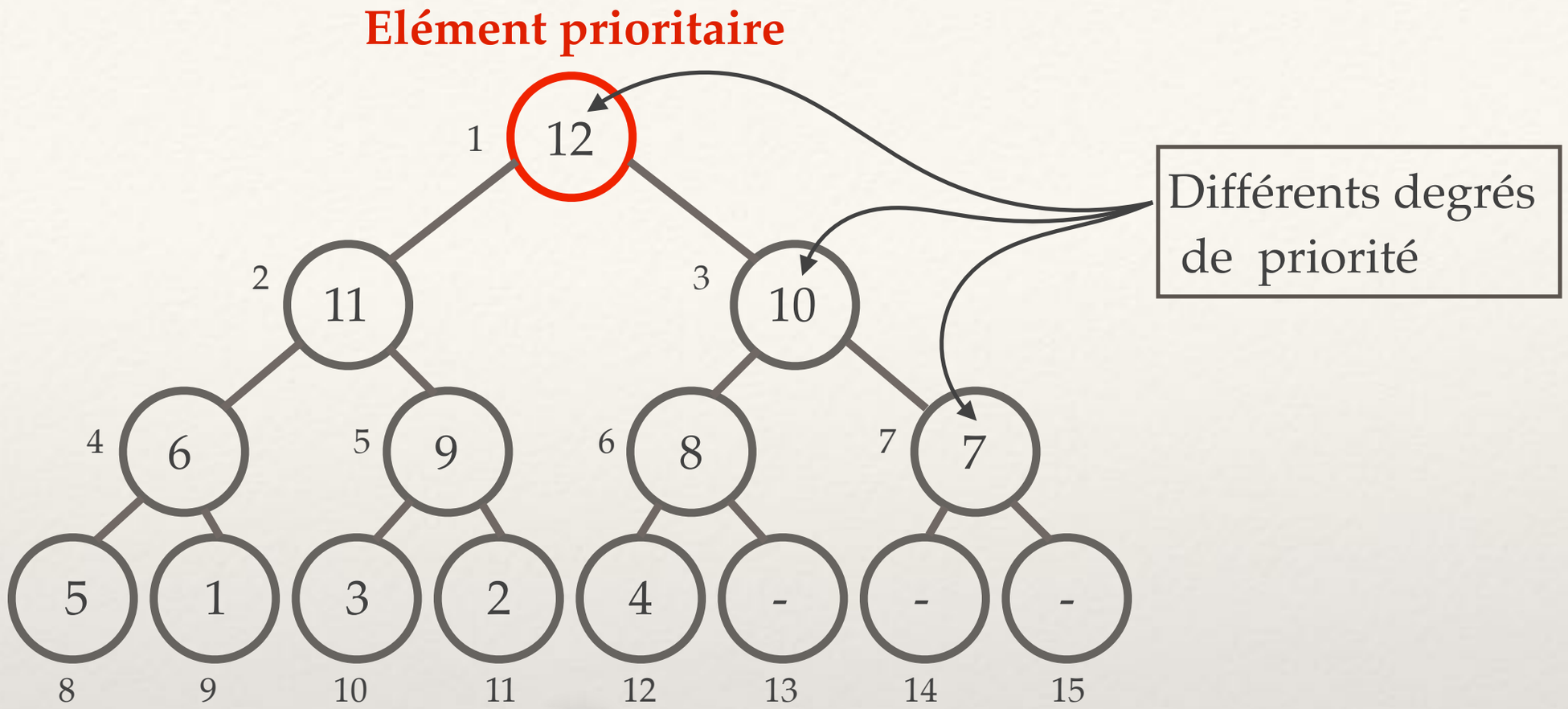


Figure 8.3. SFQ operation

3 - Queue Prioritaire : on s'appuie sur un arbre binaire

Abstraction :
Structure de données



Système de conversion

Réalité mémoire : tableau

Rq : n = i + 1

12	11	10	6	9	8	7	5	1	3	2	4	-	-	-
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

n ->

Structure de donnée minimale :

Queue Prioritaire

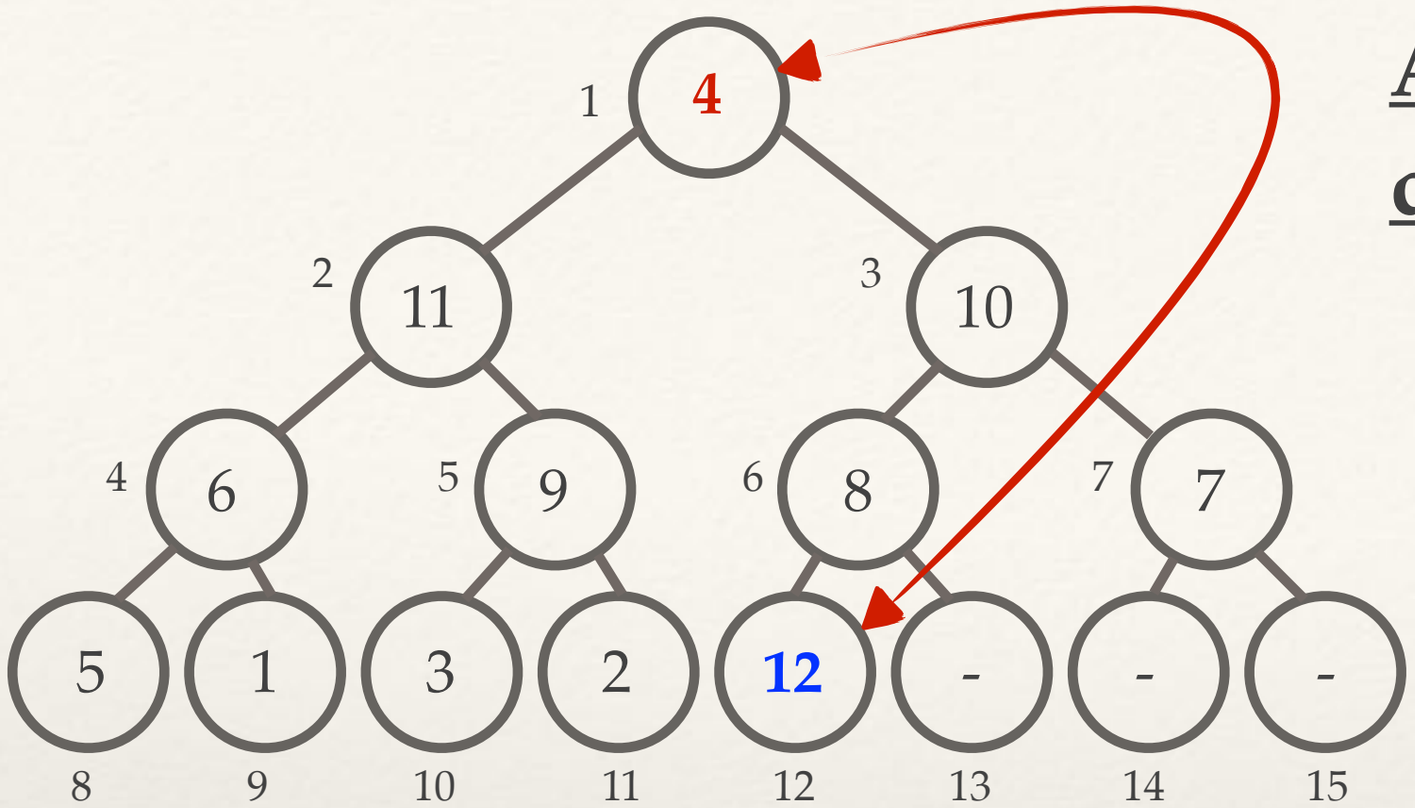
Attributs :

- valeurMax
- longueur
- longueurMax

Méthodes :

- insert(elem)
- popMax()
- isEmpty()
- isFull()

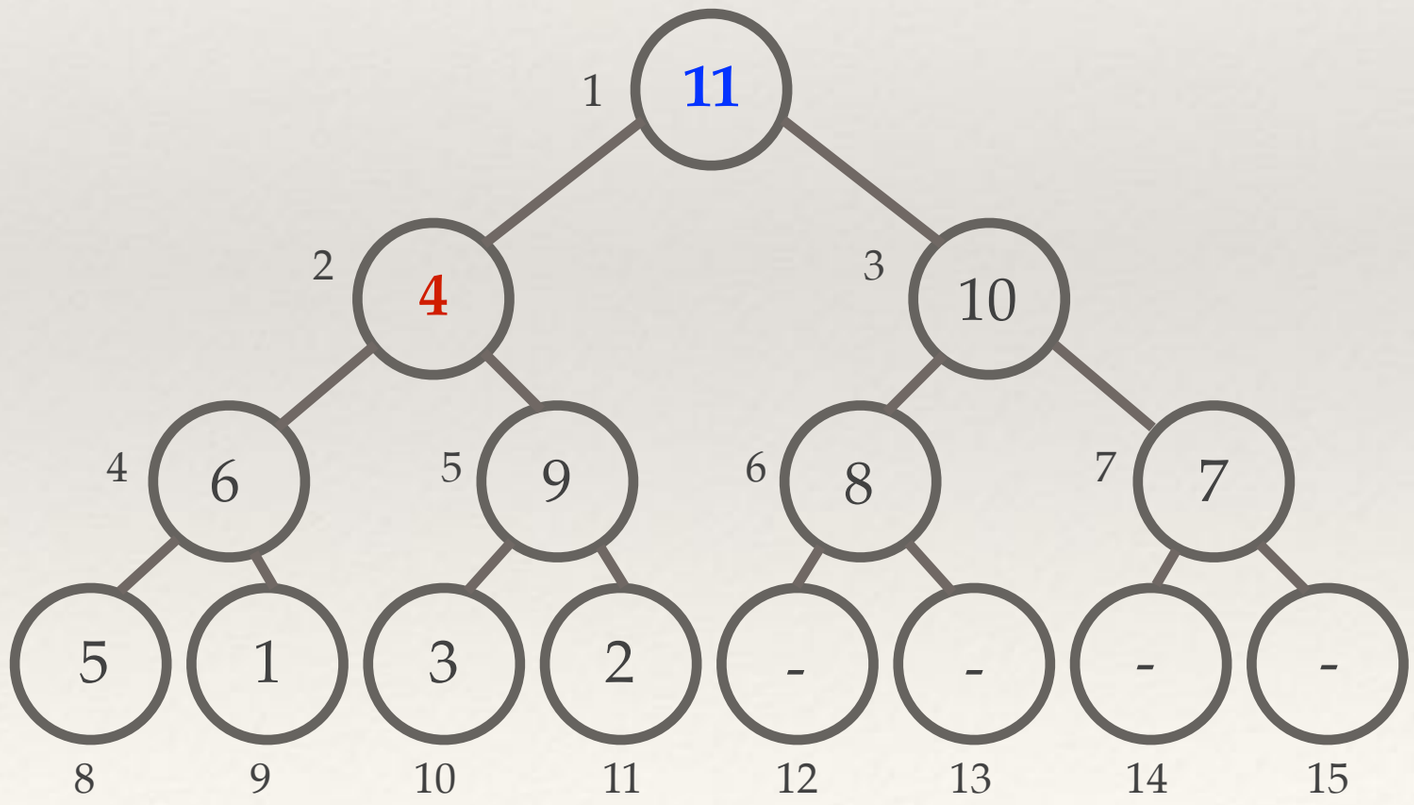
Algorithme POP d'extraction du max:

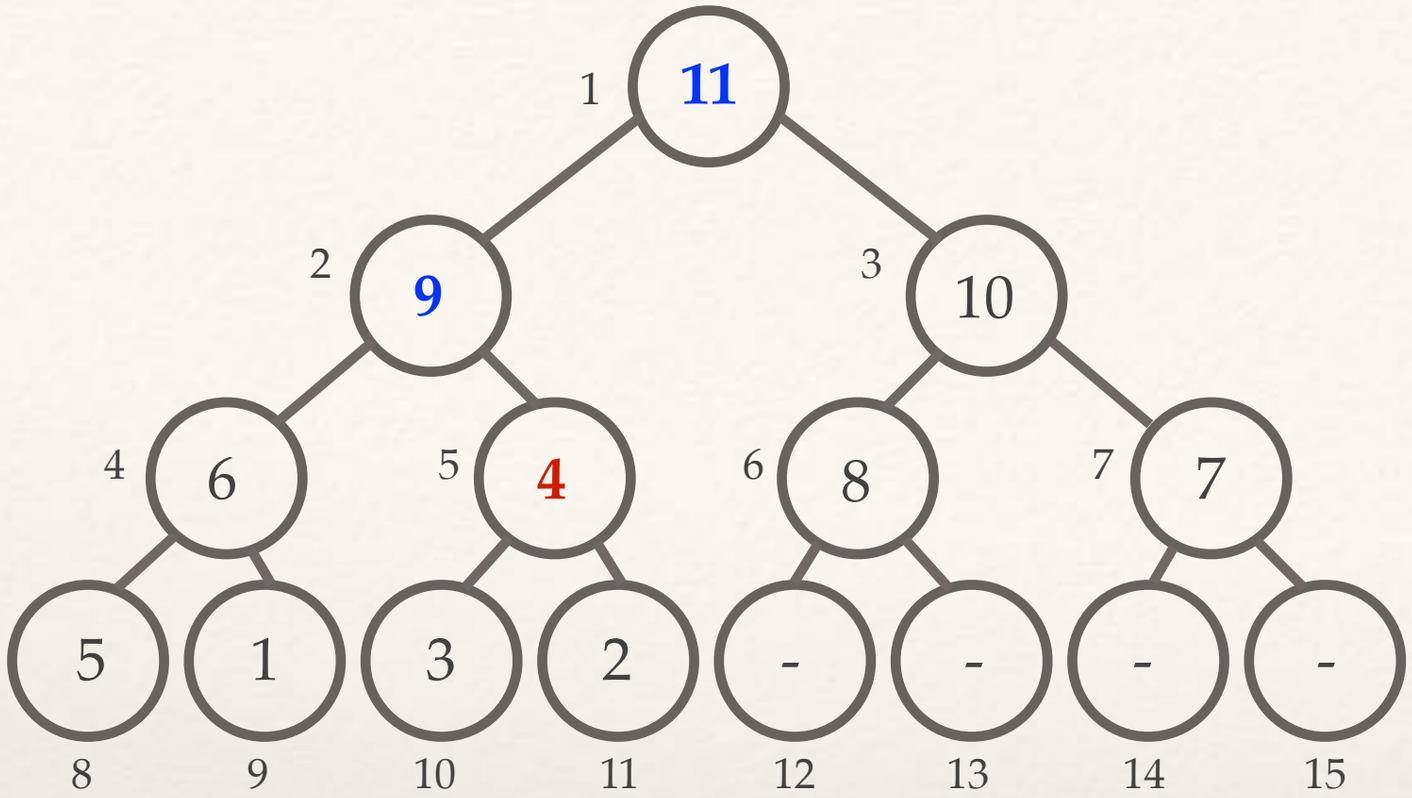


POP : 12

Re-structuration

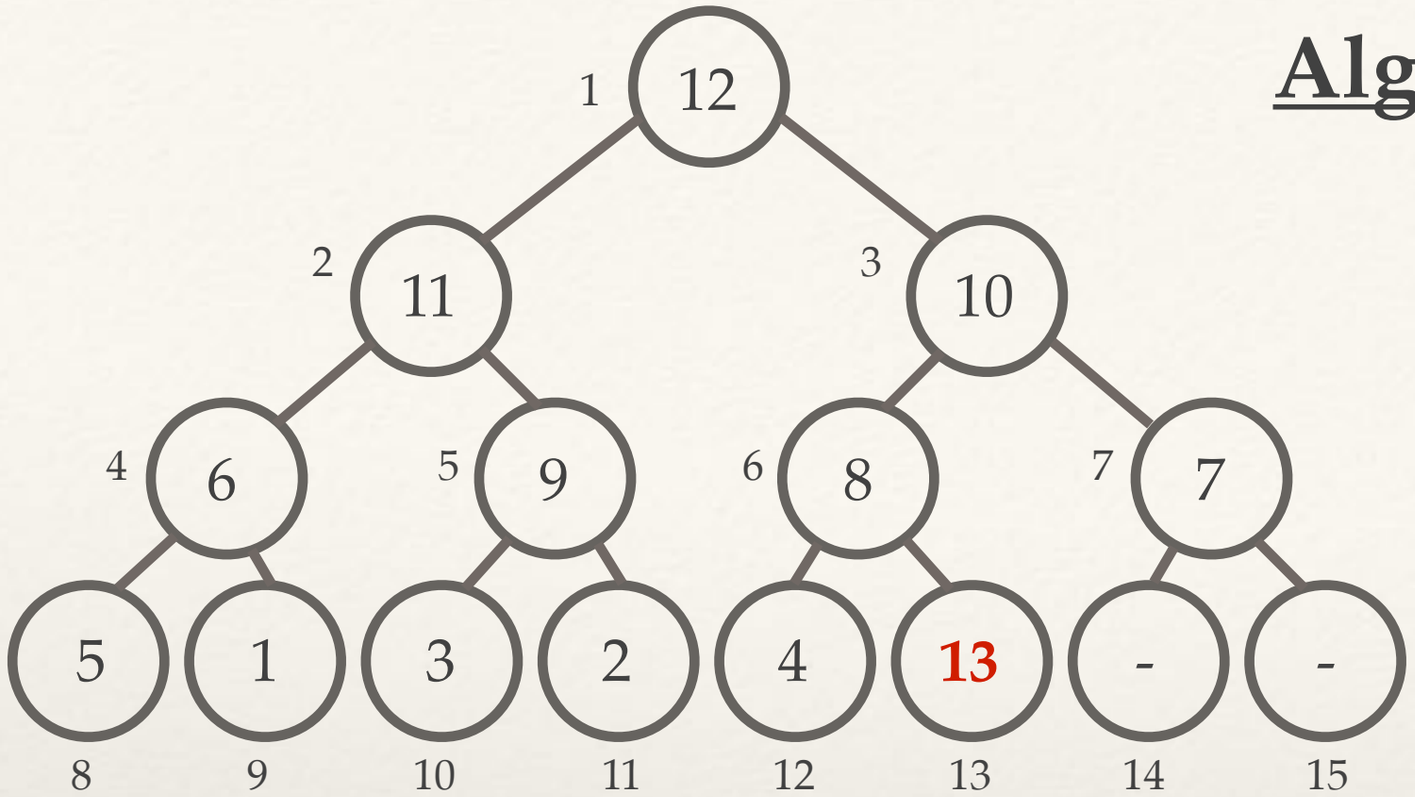
Maintenir l'invariant structurel

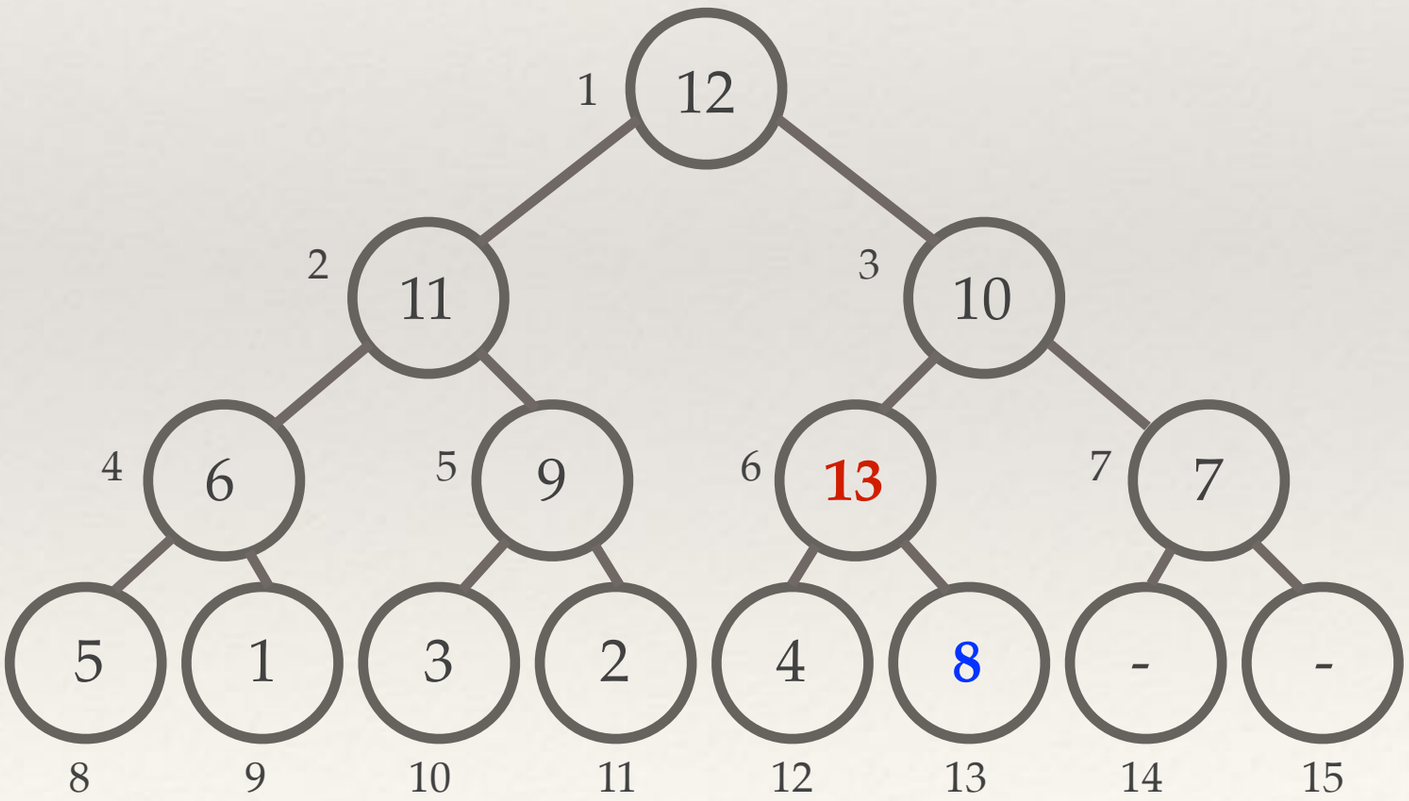


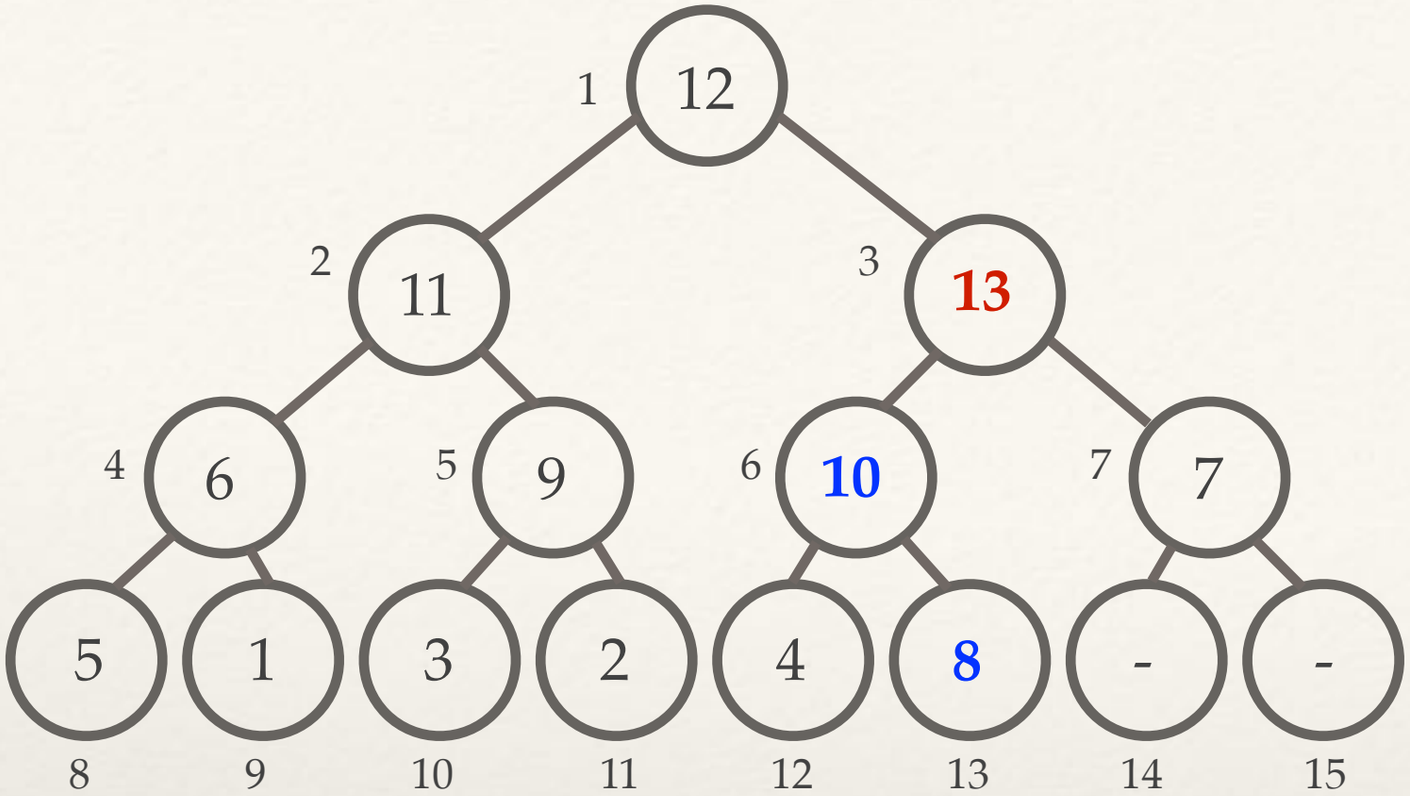


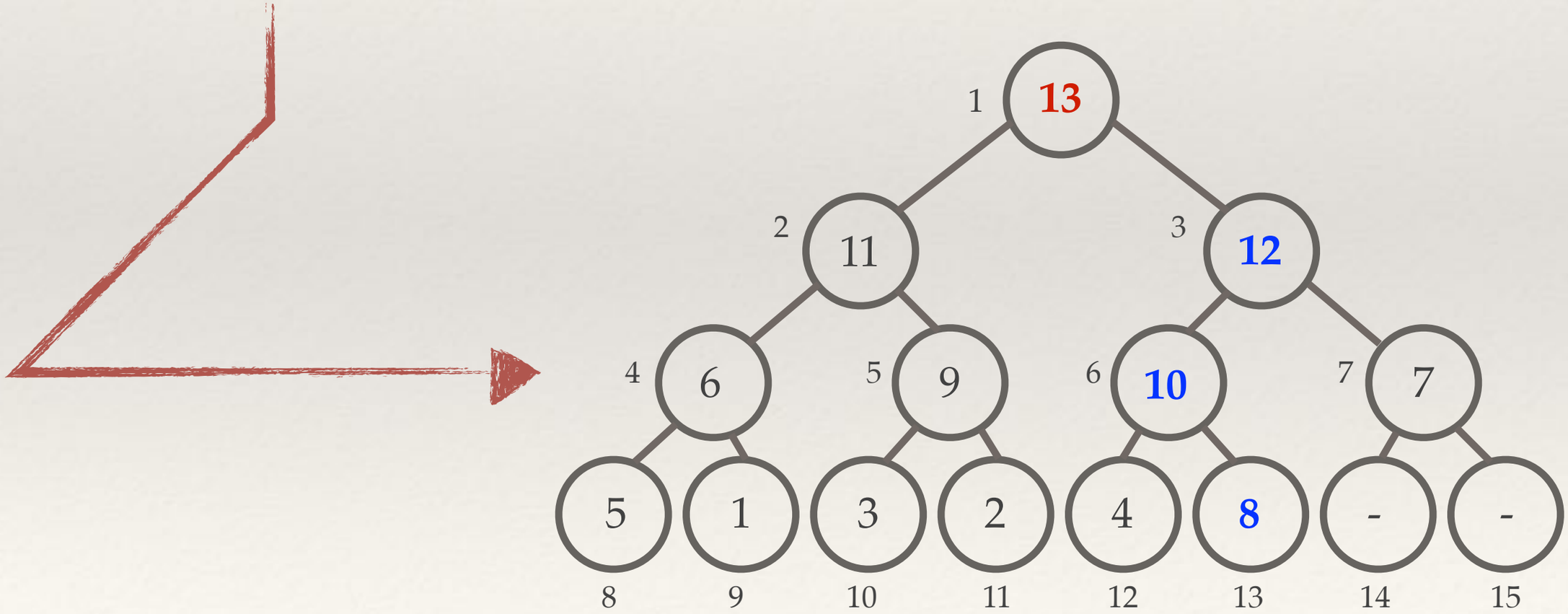
Re-structuration

Algorithme d'insertion :






1
12
2
11
3
13
4
6
5
9
6
10
7
7
5
1
3
2
4
8
-
-
8
9
10
11
12
13
14
15


1
13
2
11
3
12
4
6
5
9
6
10
7
7
5
1
3
2
4
8
-
-
8
9
10
11
12
13
14
15

Résolution de Problème :

- Proposer un algorithme de Tri en s'appuyant sur la queue prioritaire.
- On part d'un tableau de données aléatoires et de taille N.
- Evaluer sa complexité