

Structures Linéaires





16. STL heaps

Heaps (tas)



- Fonctions de la librairie `<algorithm>`
- Typiquement appliquées au contenu d'un `std::vector<T>`

Fonction	Description
<code><u>std::push_heap</u></code>	insère le dernier élément dans le tas
<code><u>std::pop_heap</u></code>	supprime l'élément au sommet du tas
<code><u>std::make_heap</u></code>	crée un tas à partir d'un tableau quelconque
<code><u>std::sort_heap</u></code>	trie les éléments à partir d'un tas
<code><u>std::is_heap</u></code>	vérifie si les éléments respectent la condition de tas
<code><u>std::is_heap_until</u></code>	trouve le premier élément ne respectant pas la condition de tas

Interface des fonctions xxx_heap()



```
template <class RandomAccessIterator>  
void xxx_heap (RandomAccessIterator first, RandomAccessIterator last);
```

- S'applique à la séquence d'éléments de l'intervalle [first, last[
- Utilise T::operator< (...) pour définir les priorités.

Interface des fonctions xxx_heap()



```
template <class RandomAccessIterator, class Compare>  
  
void xxx_heap (RandomAccessIterator first, RandomAccessIterator last,  
               Compare comp);
```

- S'applique à la séquence d'éléments de l'intervalle [first, last[
- Peut recevoir la fonction de comparaison en paramètre. Par exemple

```
template <typename T>  
bool comp(const T& a, const T& b) {  
    return a < b;  
}
```



Fonction	Précondition	Postcondition	Description
<u><code>std::push_heap</code></u>	<code>[first, last-1[</code> est un tas	<code>[first, last[</code> est un tas	insère l'élément en <code>last-1</code> dans le tas
<u><code>std::pop_heap</code></u>	<code>[first, last[</code> est un tas	<code>[first, last-1[</code> est un tas	supprime l'élément au sommet du tas, et le met en <code>last-1</code>
<u><code>std::make_heap</code></u>		<code>[first, last[</code> est un tas	crée un tas à partir d'un tableau quelconque
<u><code>std::sort_heap</code></u>	<code>[first, last[</code> est un tas	<code>[first, last[</code> est trié	trie les éléments à partir d'un tas
<u><code>std::is_heap</code></u>			vérifie si les éléments respectent la condition de tas
<u><code>std::is_heap_until</code></u>			trouve le premier élément ne respectant pas la condition de tas

Pré-conditions



Fonctions	Pré-condition
<code>pop_heap(first, last)</code> <code>sort_heap(first, last)</code>	<code>[first, last[</code> respecte la condition de tas
<code>push_heap(first, last)</code>	<code>[first, last-1[</code> respecte la condition de tas

- Aucune vérification de ces pré-conditions n'est faite.

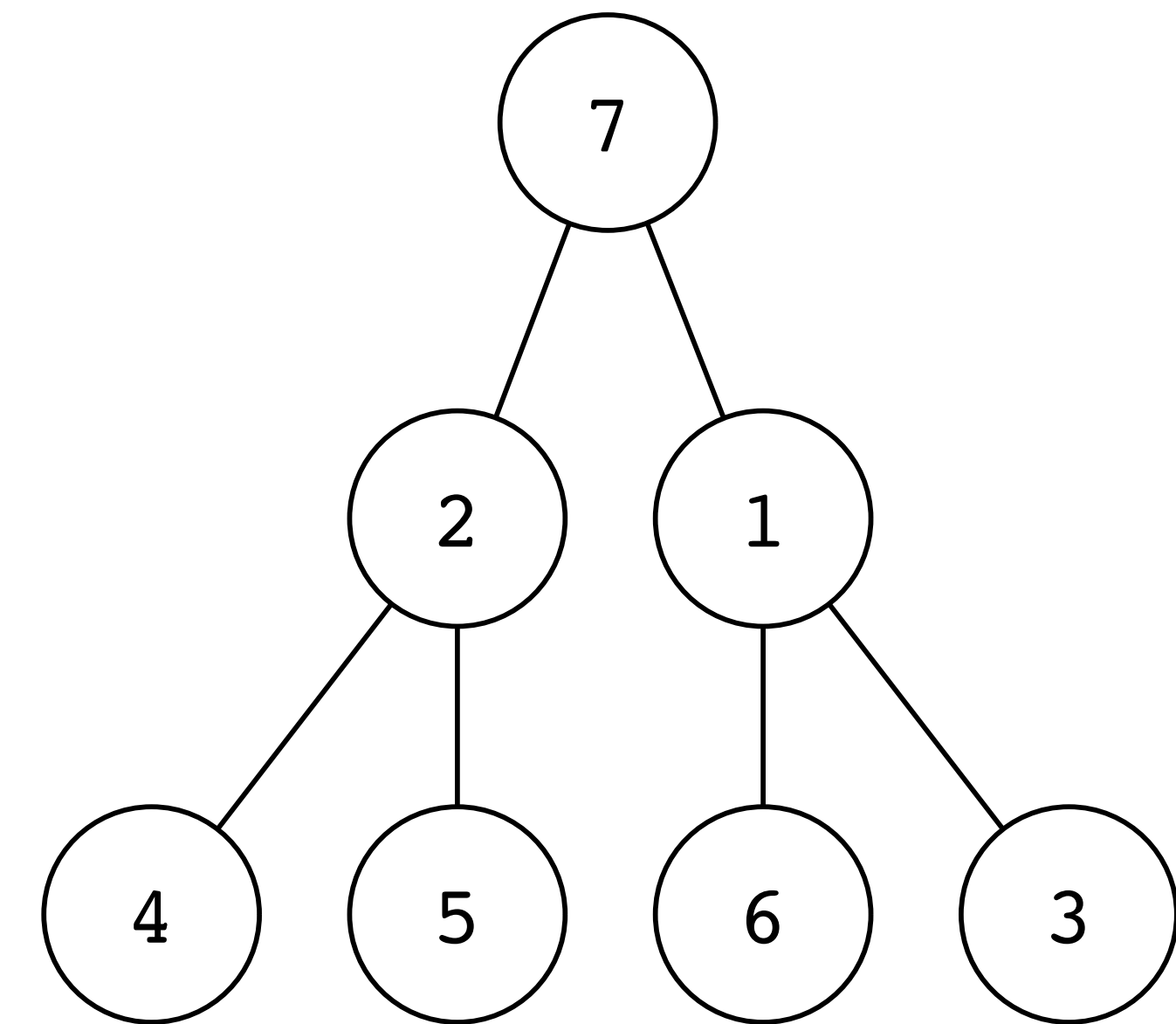
⚠ Si elles ne sont pas valides, l'algorithme s'applique, mais le résultat n'est pas celui attendu

Pré-condition sur push_heap



```
// ajout dans un tas incorrect  
v.assign({ 1, 2, 3, 4, 5, 6, 7 });  
push_heap(v.begin(), v.end());  
cout << is_heap(v.begin(), v.end());
```

false



{ 7, 2, 1, 4, 5, 6, 3 }

Pré-condition sur pop_heap



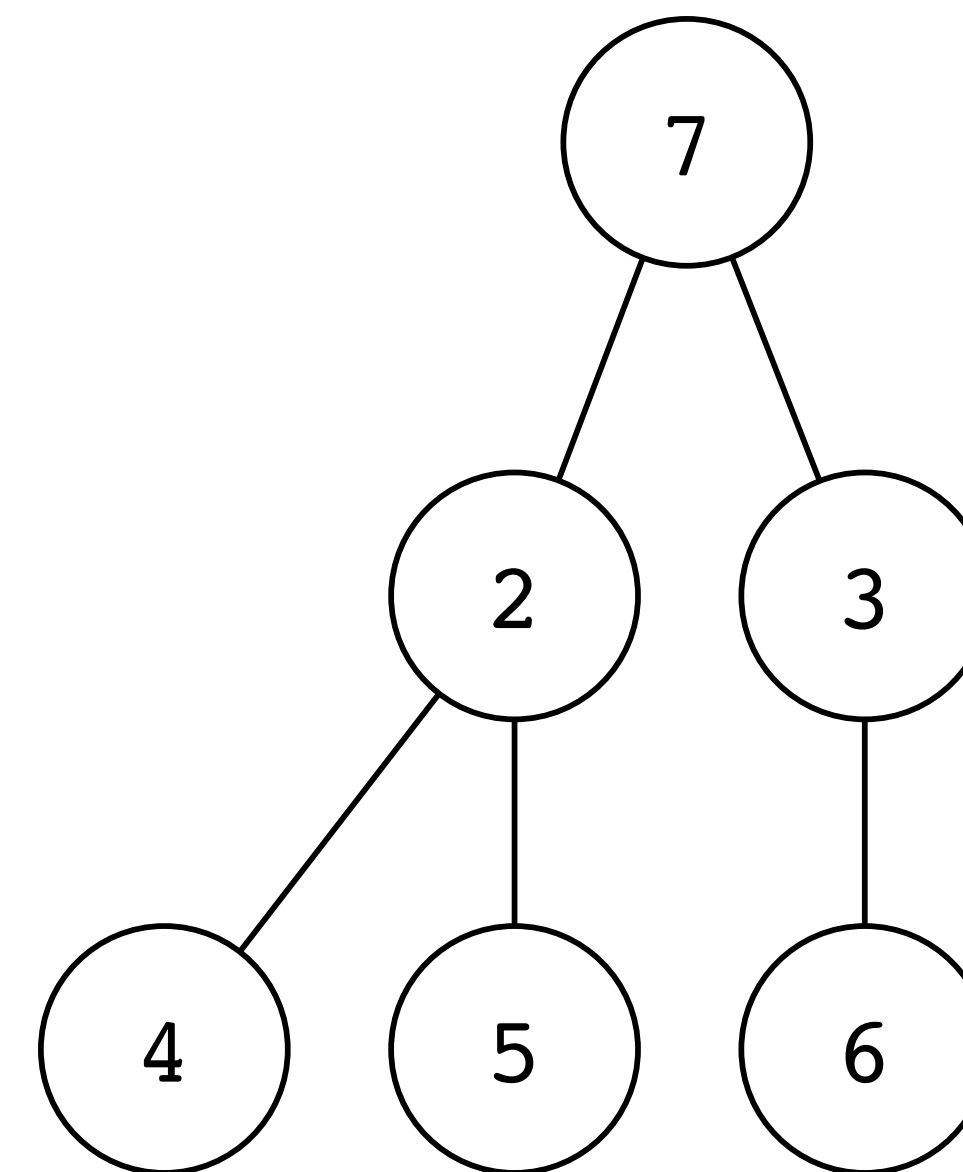
```
// suppression d'un tas incorrect
```

```
v.assign({ 1, 2, 3, 4, 5, 6, 7 });
```

```
pop_heap(v.begin(), v.end());
```

```
cout << is_heap(v.begin(), v.end()-1);
```

false



{ 7, 2, 3, 4, 5, 6, 1 }

Pré-condition sur sort_heap



```
// tri à partir d'un tas incorrect  
v.assign({ 1, 2, 3, 4, 5, 6, 7 });  
sort_heap(v.begin(), v.end())  
cout << is_sorted(v.begin(), v.end());
```

Pré-condition sur sort_heap



```
// tri à partir d'un tas incorrect  
v.assign({ 1, 2, 3, 4, 5, 6, 7 });  
sort_heap(v.begin(), v.end())  
cout << is_sorted(v.begin(), v.end());
```

false

{ 2, 3, 4, 5, 6, 7, 1 }

Limitations



- Cette interface ne fait que réordonner les éléments du conteneur.
- **Il n'y a pas d'ajout/suppression d'éléments du conteneur.**
- Il faut donc le faire soi-même avant ou après.

Créer : make_heap



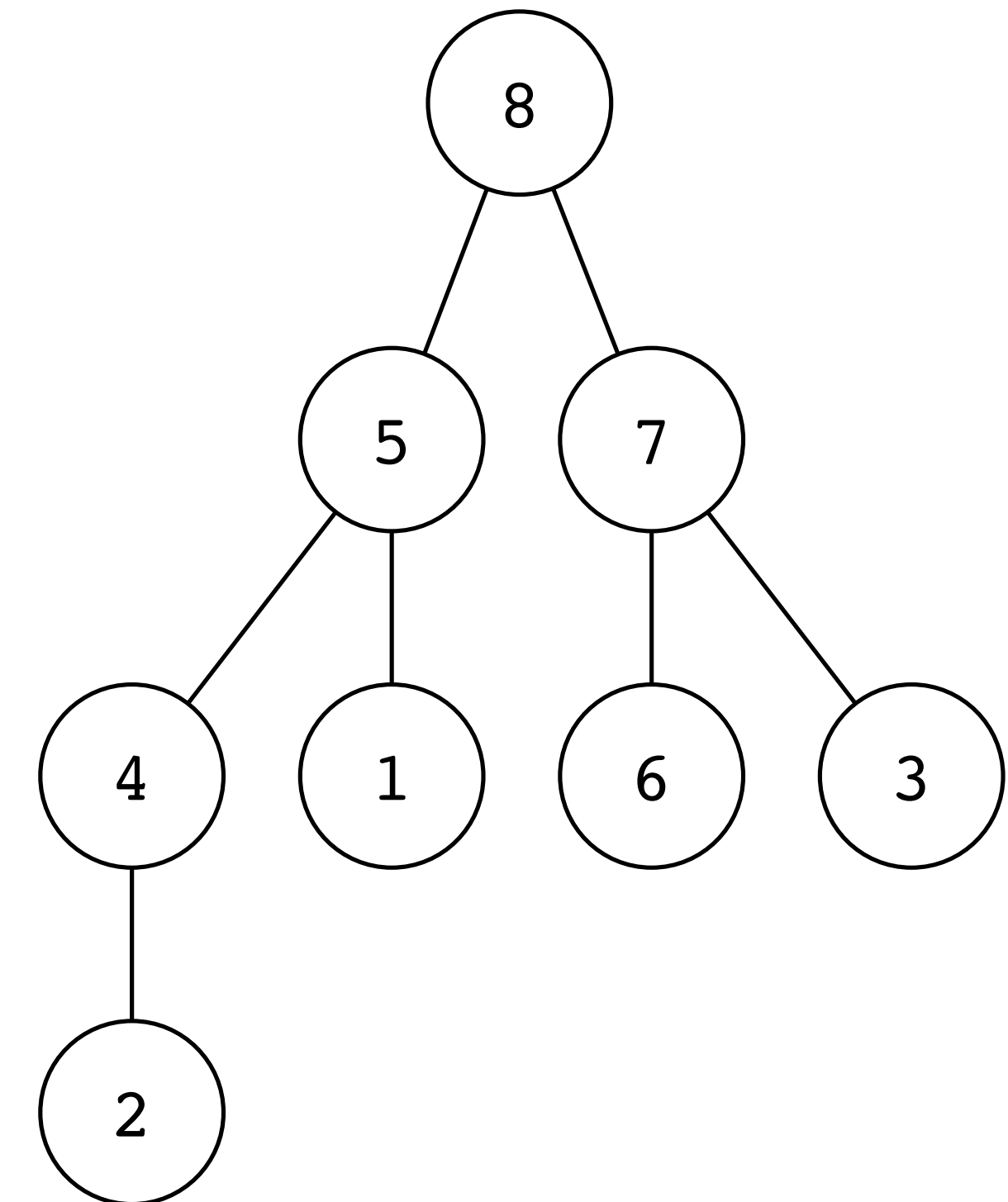
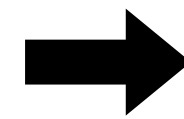
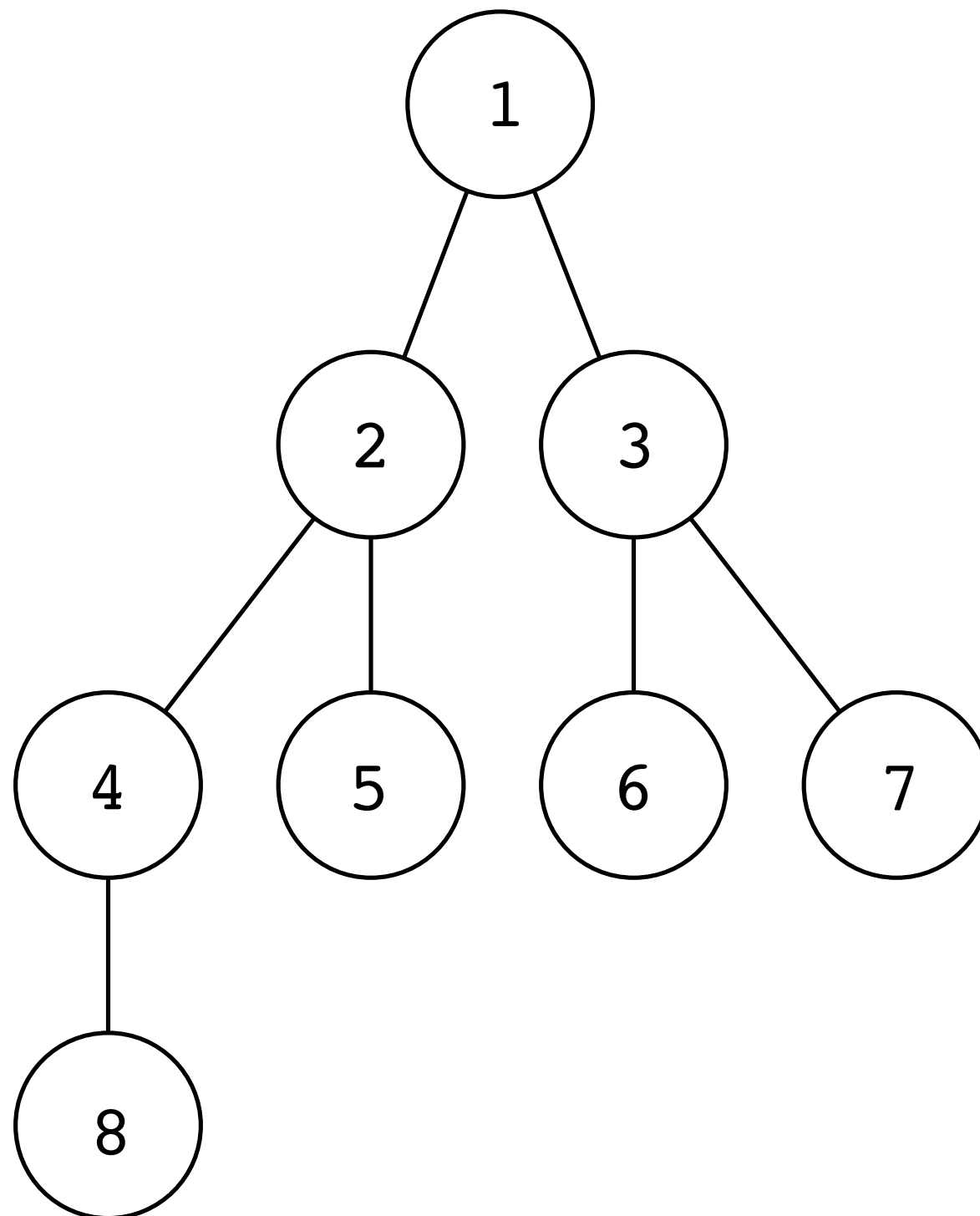
// Créer le tas

```
std::make_heap( v.begin(), v.end() );
```

```
std::vector<int> v =
```

```
{ 1, 2, 3, 4, 5, 6, 7, 8 };
```

```
{ 8, 5, 7, 4, 1, 6, 3, 2 };
```



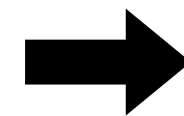
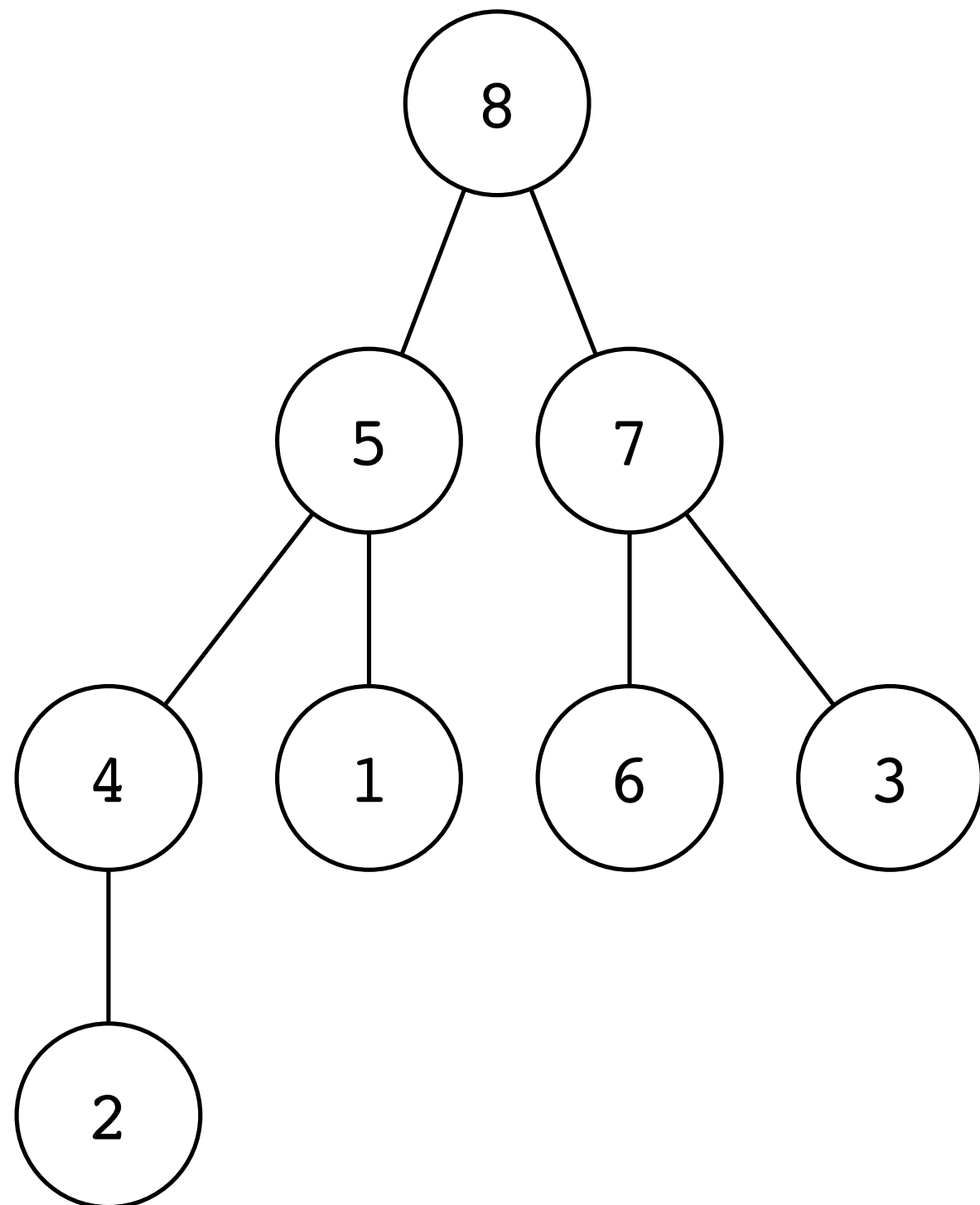
Insérer (1) : push_back



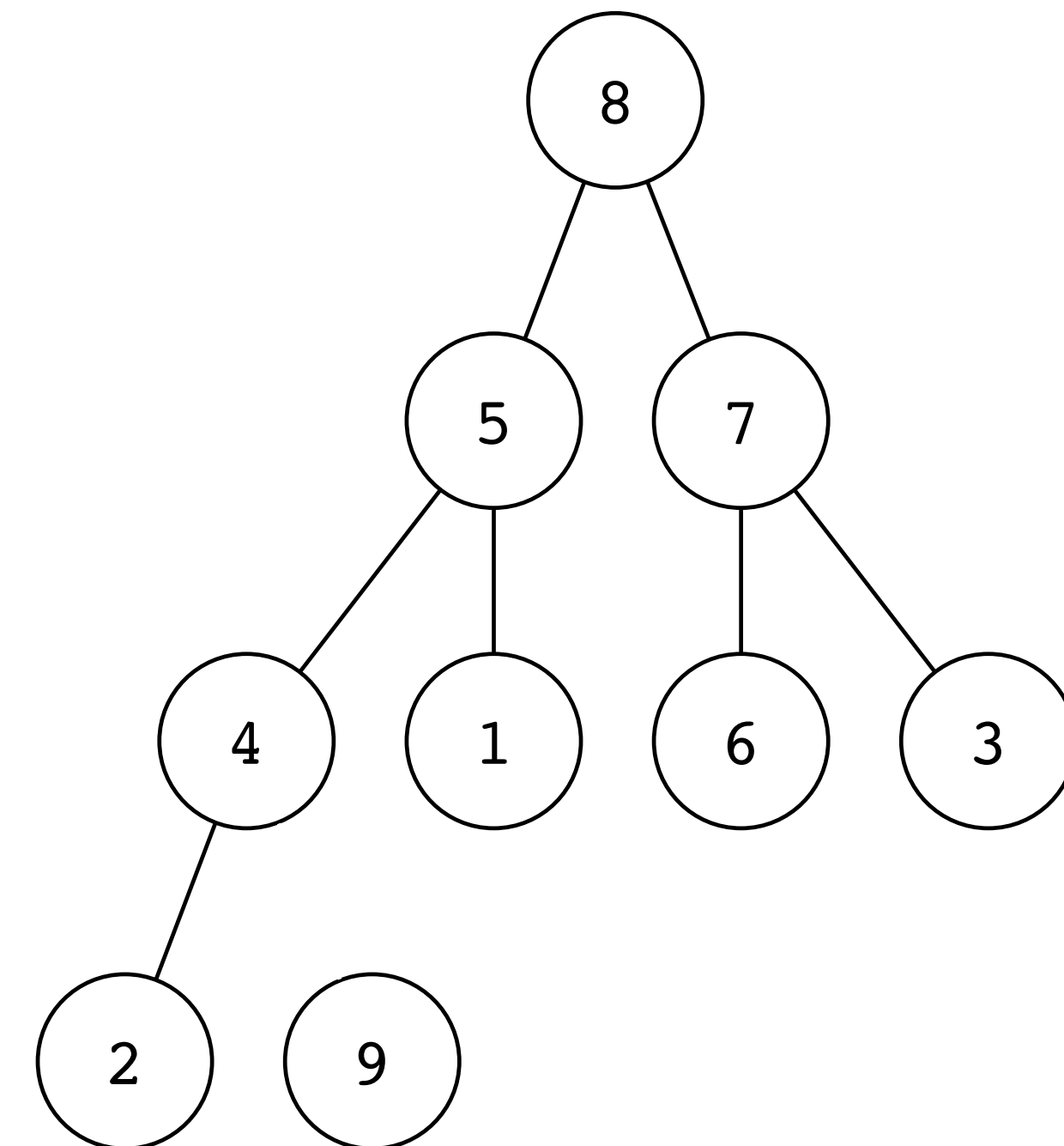
// Ajouter l'élément 9 en fin de vecteur

`v.push_back( 9 );`

{ 8, 5, 7, 4, 1, 6, 3, 2 }



{ 8, 5, 7, 4, 1, 6, 3, 2, 9 }



Insérer (2) : push_heap

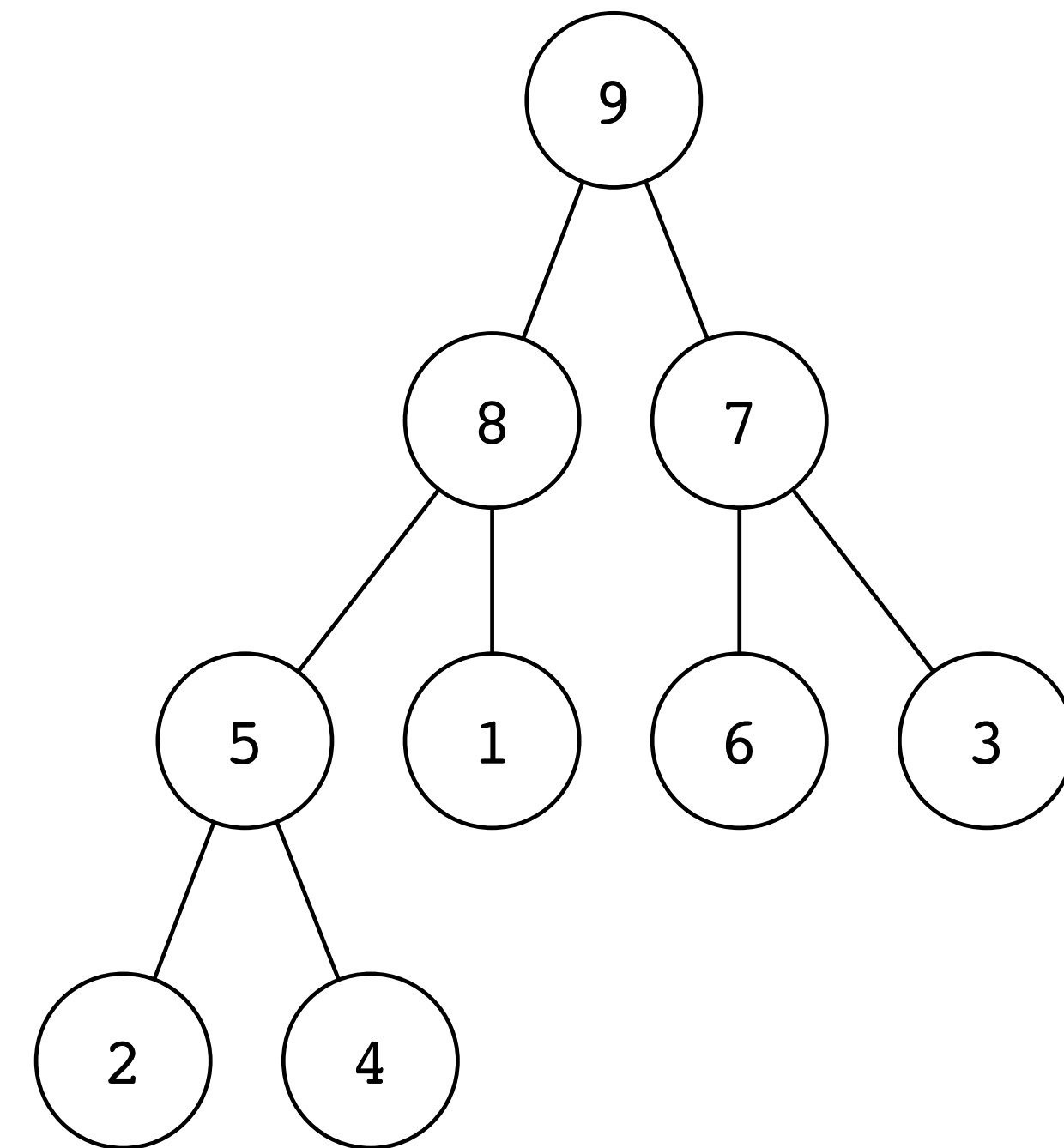
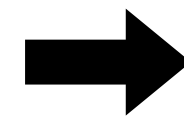
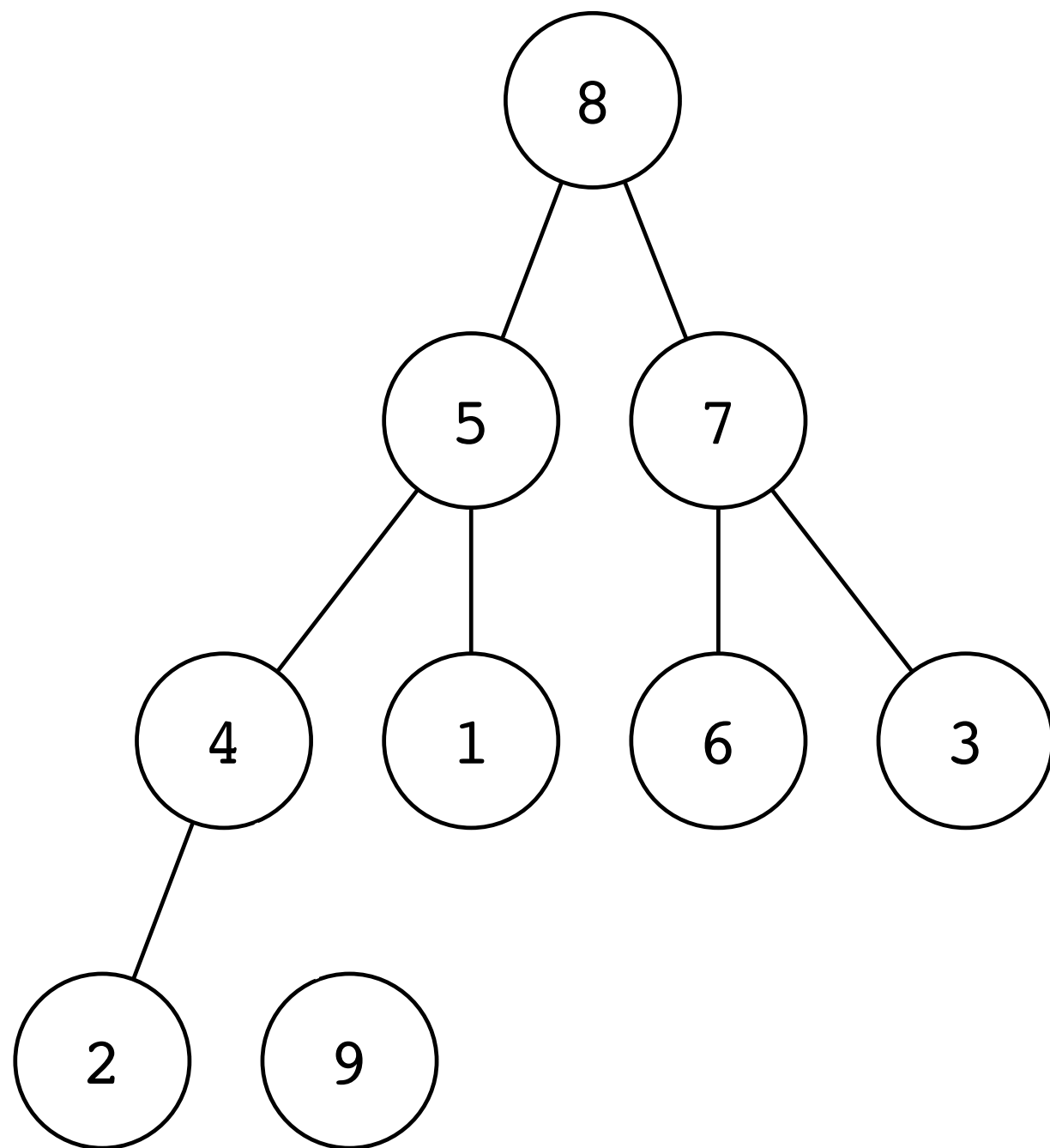


// Placer le dernier élément dans le tas

```
std::push_heap( v.begin(), v.end() );
```

{ 8, 5, 7, 4, 1, 6, 3, 2, 9 }

{ 9, 8, 7, 5, 1, 6, 3, 2, 4 }



Supprimer (1) : pop_heap

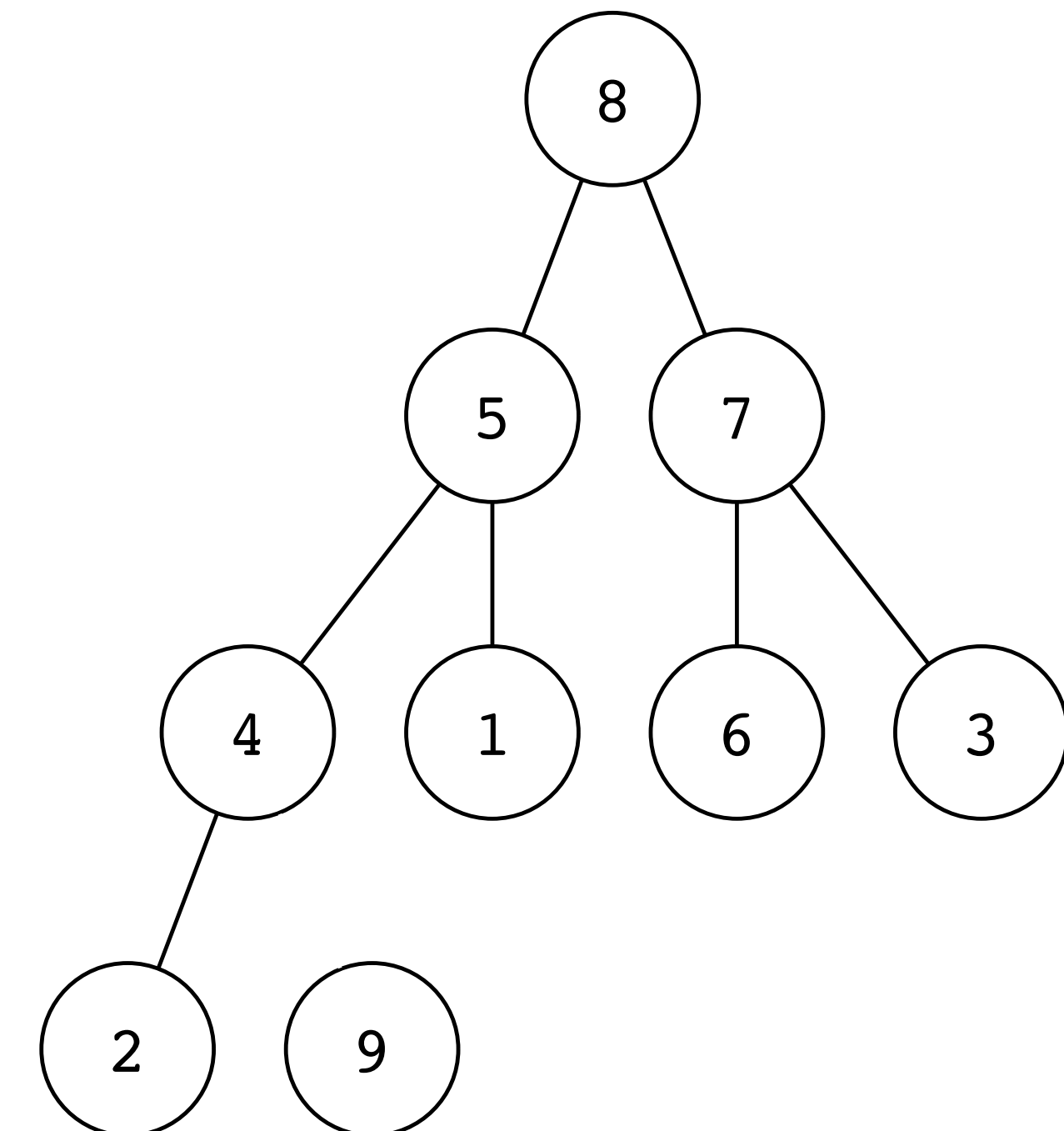
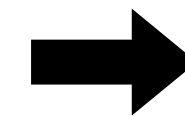
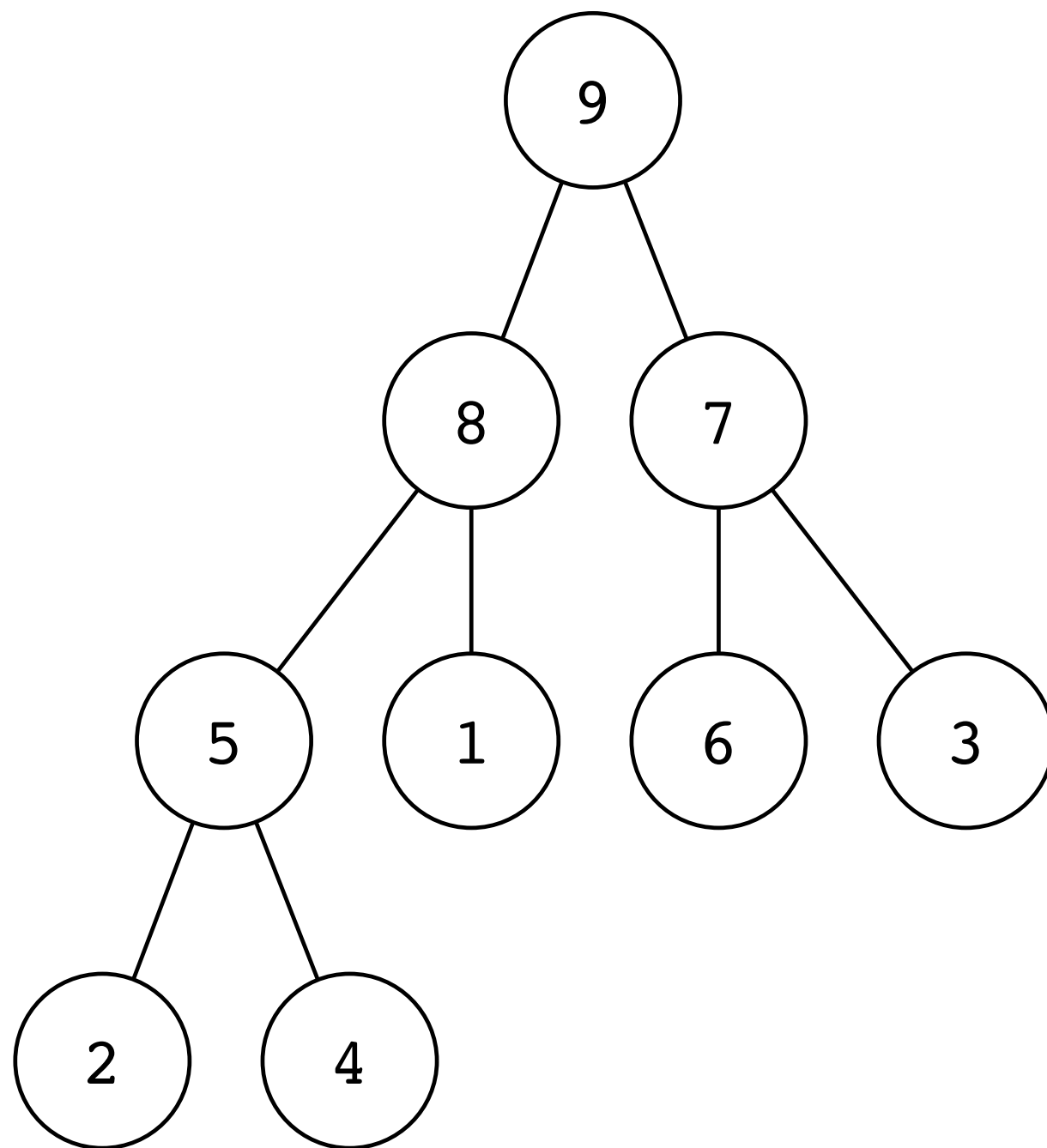


// Placer le sommet en fin, rétablir le tas

```
std::pop_heap( v.begin(), v.end() );
```

{ 9, 8, 7, 5, 1, 6, 3, 2, 4 }

{ 8, 5, 7, 4, 1, 6, 3, 2, 9 }



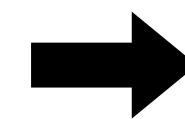
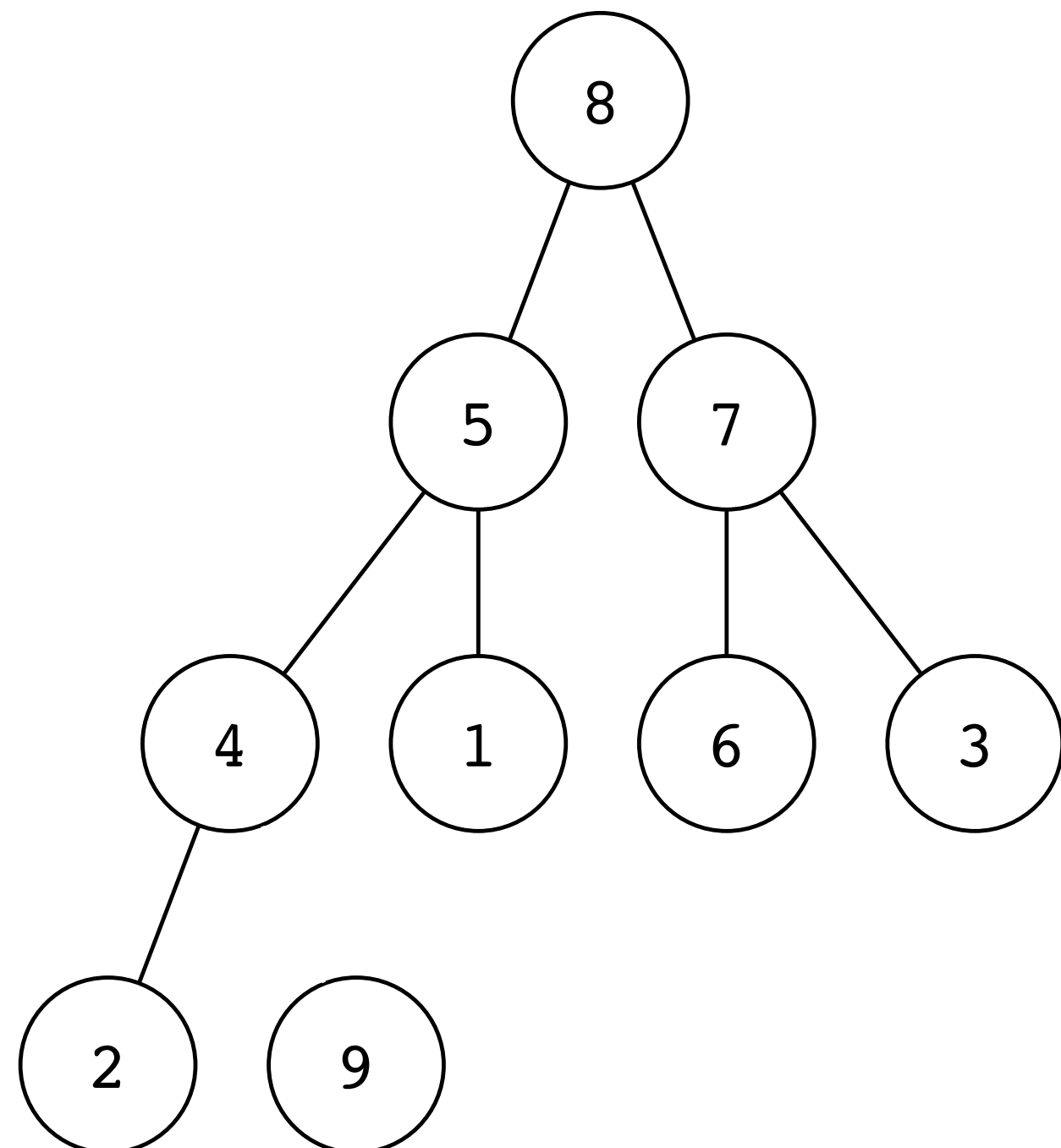
Supprimer (2) : pop_back



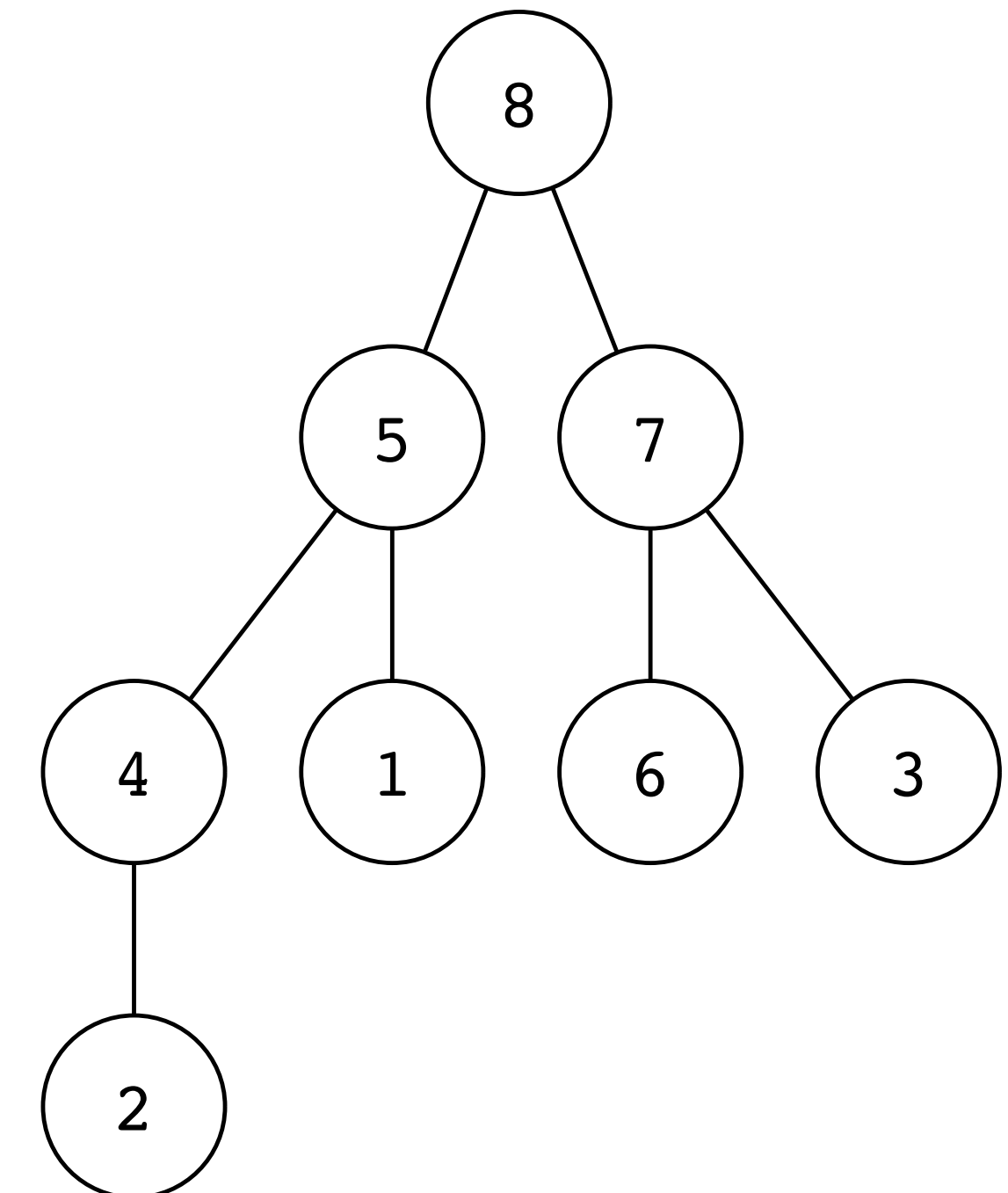
// supprimer le dernier élément

`v.pop_back();`

{ 8, 5, 7, 4, 1, 6, 3, 2, 9 }



{ 8, 5, 7, 4, 1, 6, 3, 2 }



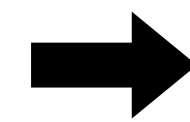
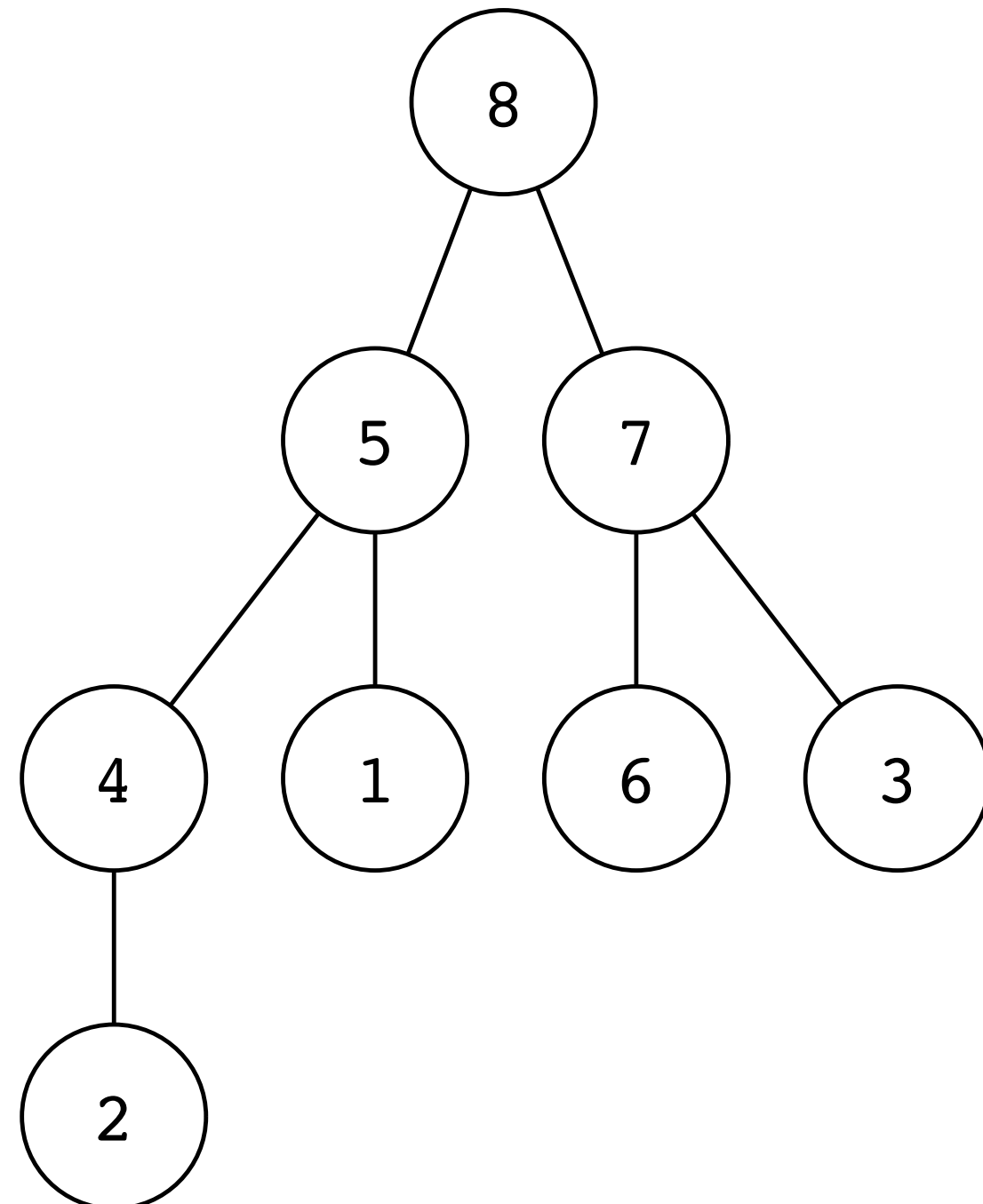
Trier : sort_heap



```
// Trier à partir du tas v
```

```
std::sort_heap( v.begin(), v.end() );
```

{ 8, 5, 7, 4, 1, 6, 3, 2 }



{ 1, 2, 3, 4, 5, 6, 7, 8 }

Condition de tas ? `is_heap`

```
{ 1, 2, 3, 4, 5, 6, 7, 8 }
```

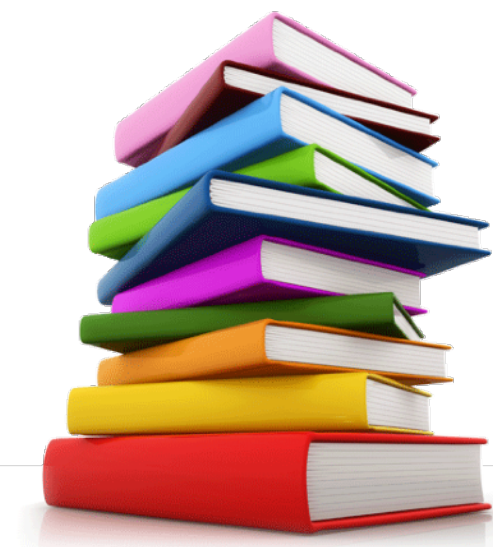
```
// vérifier si c'est un tas
```

```
std::cout << std::is_heap( v.begin(), v.end() );
```

```
false
```

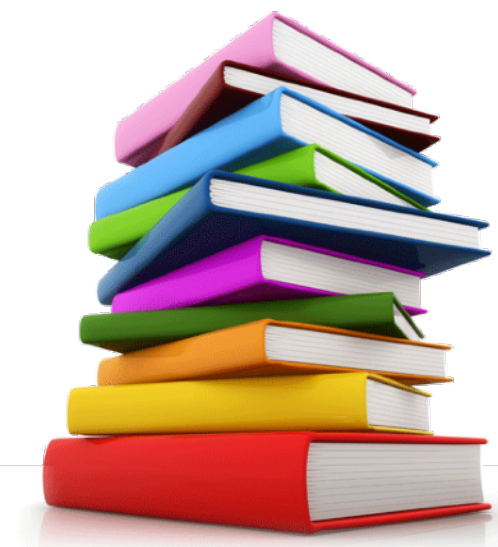
```
// Trouver le premier élément plus grand que son parent
```

```
std::cout << *(std::is_heap_until(v.begin(), v.end()));
```



17. Pile (stack)

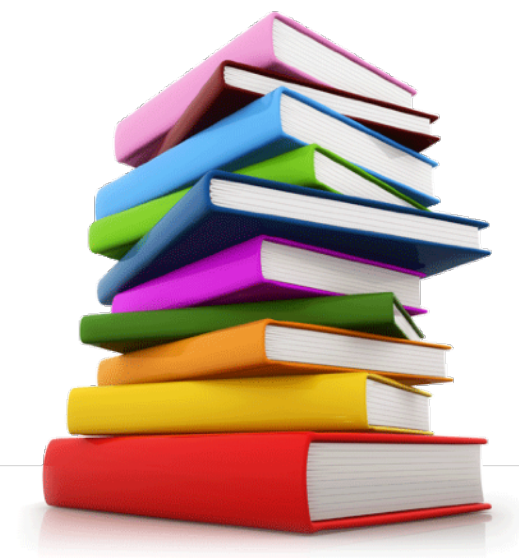
Pile



- Insérer et supprimer du même côté:
le sommet
- “LIFO” - Last In, First Out
- 3 opérations
 - empiler (push)
 - dépiler (pop)
 - sommet (top)



Pile

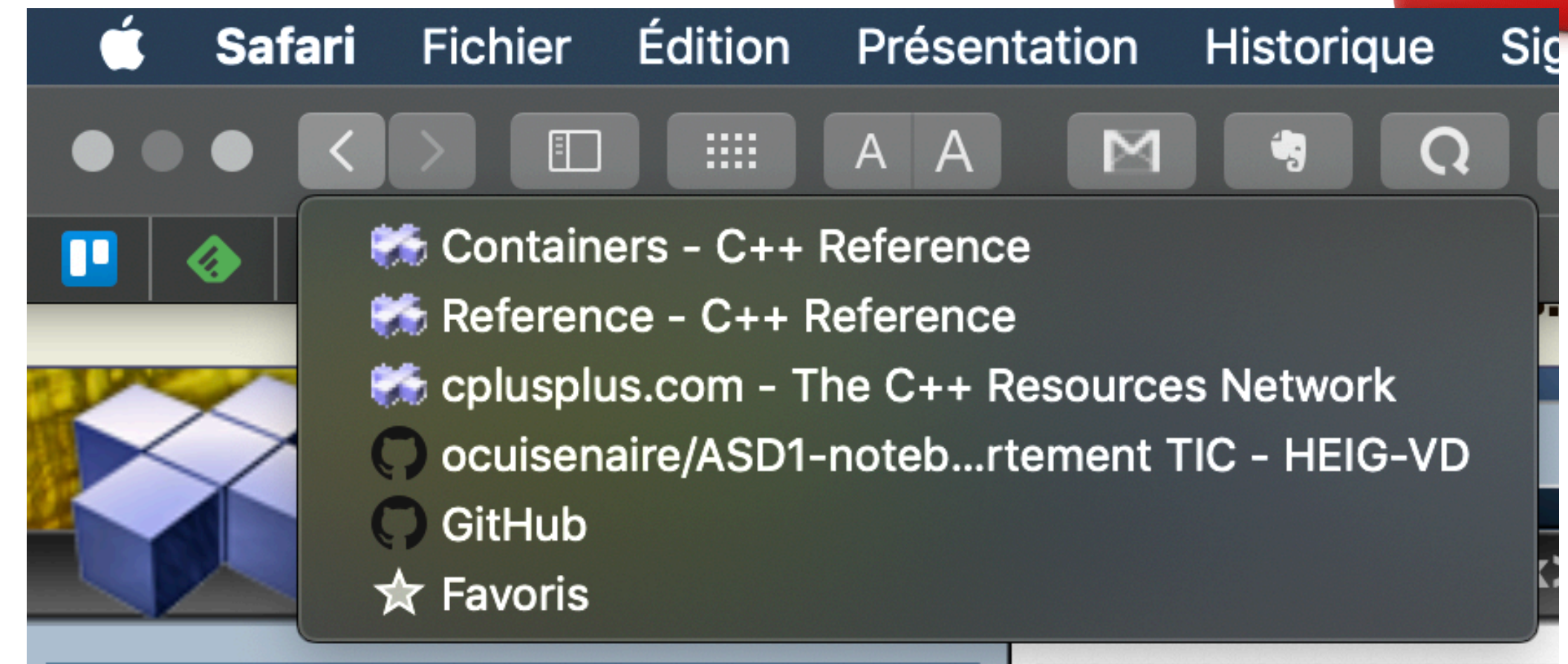
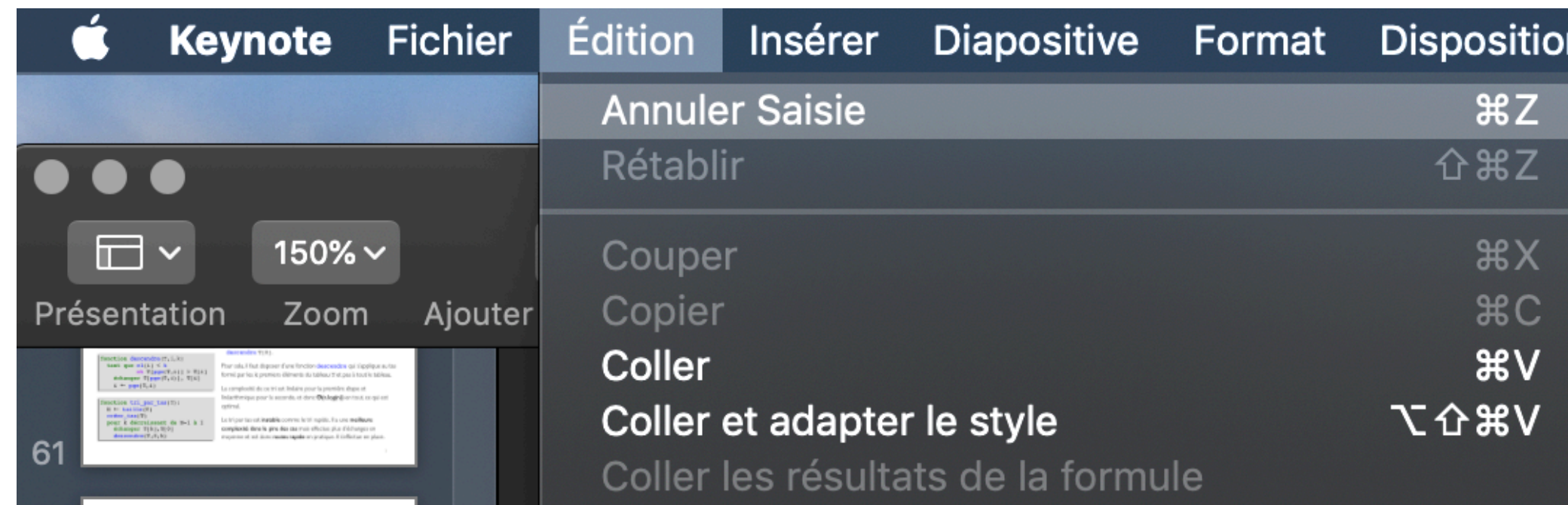
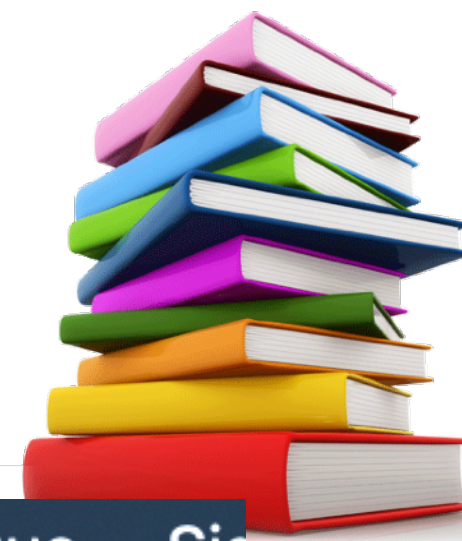


Une pile est un **type de données abstrait**.

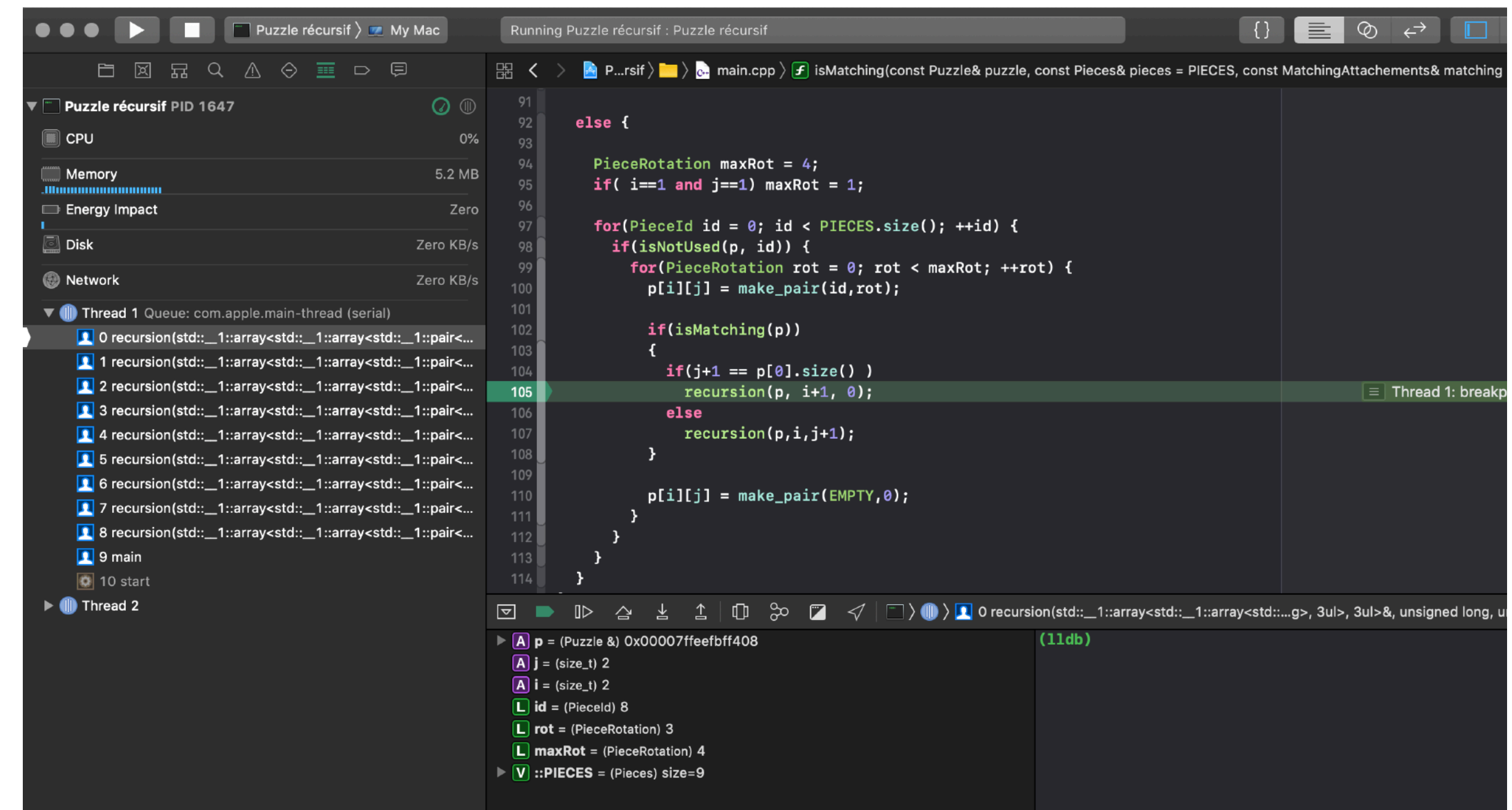
Rien n'impose la structure concrète sous-jacente.

Structure concrète	Sommet
tableau de taille variable	fin
tableau de capacité variable	fin
liste simplement chaînée	début
liste doublement chaînée	début ou fin
deque	début ou fin

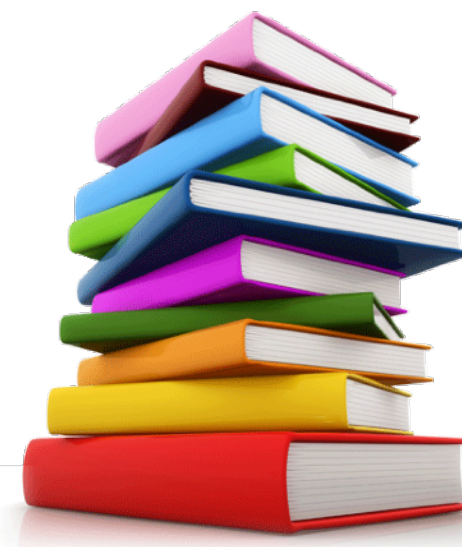
Applications des piles



- « Undo » dans un traitement de texte
- « Back » dans un navigateur web
- Stockage des adresses de retour et variables locales lors d'appels de fonctions
- Dé-récursification explicite d'un algorithme récursif



Evaluation d'expressions



- Evaluer l'expression

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

- Ne s'effectue pas de gauche à droite mais dans l'ordre suivant

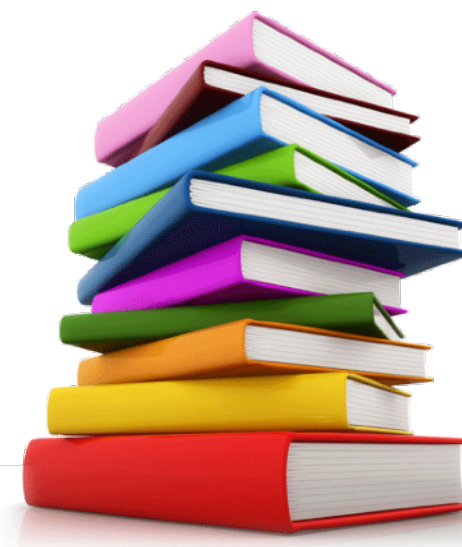
$$2 + 3 = 5$$

$$4 * 5 = 20$$

$$5 * 20 = 100$$

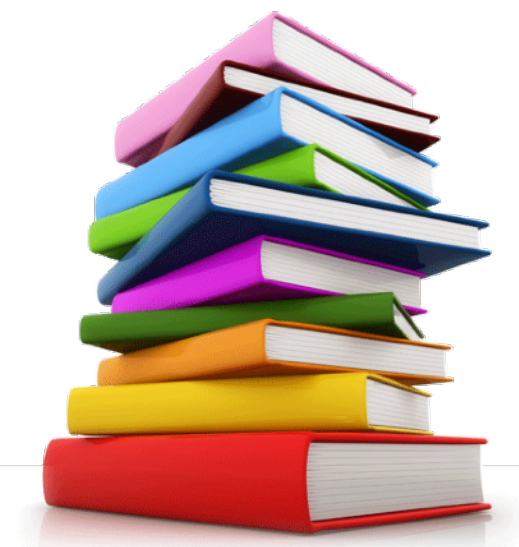
$$1 + 100 = 101$$

Evaluation d'expressions



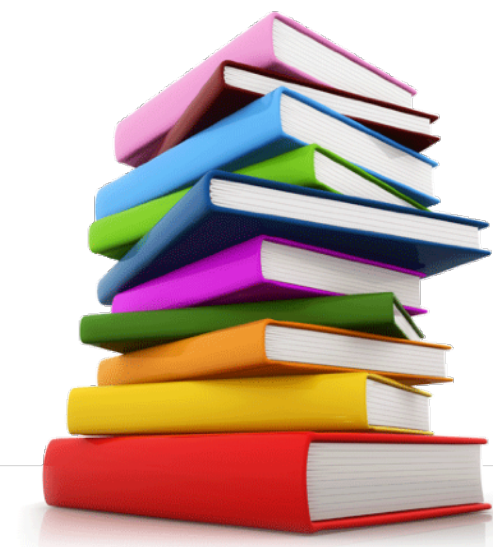
- En général, une expression s'écrit $(v1 \ op \ v2)$
- op vaut $+$, $-$, $*$, ou $/$
- $v1$ et $v2$ sont soit une valeur numérique, soit une expression
- Une expression s'évalue quand on rencontre la parenthèse $)$.
- Avant d'évaluer, il faut stocker $v1$, $v2$ et op

Première intuition



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

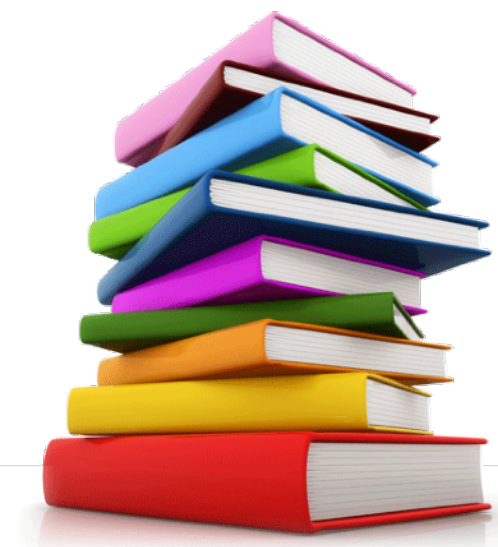
Première intuition



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

- Avancer jusqu'au prochain ‘)’.

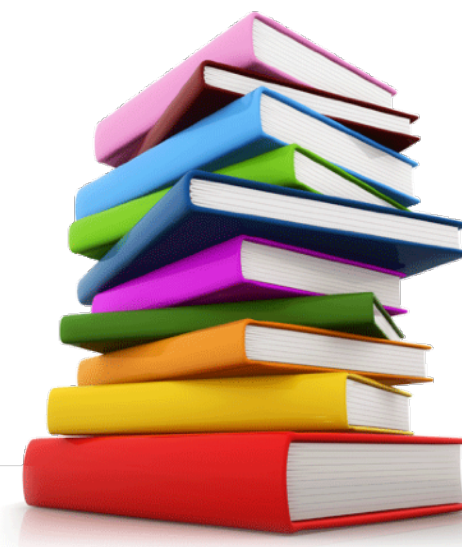
Première intuition



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

- Avancer jusqu'au prochain ')

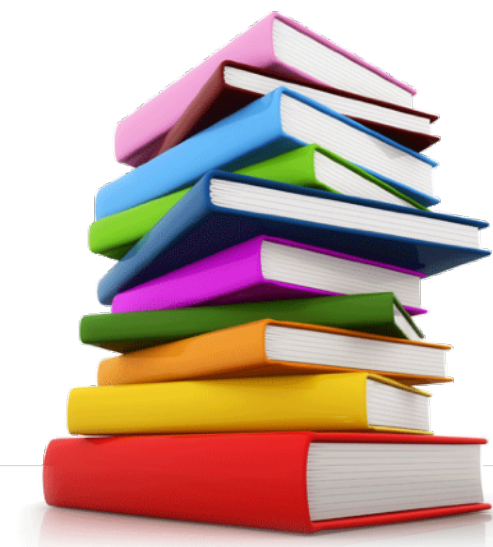
Première intuition



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

- Avancer jusqu'au prochain ‘)’.
- Calculer l'opération sur les 2 derniers éléments.

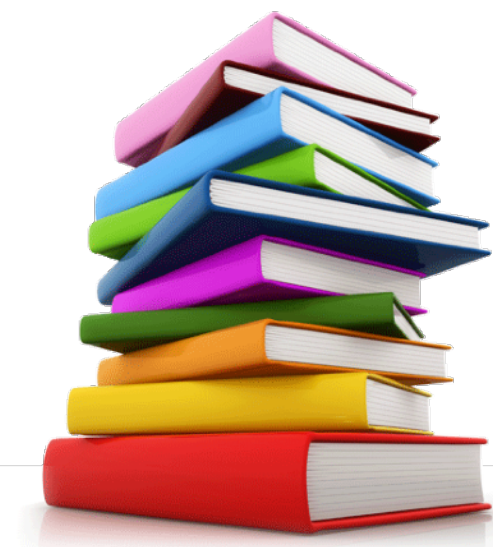
Première intuition



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

- Avancer jusqu'au prochain ') '.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

Première intuition

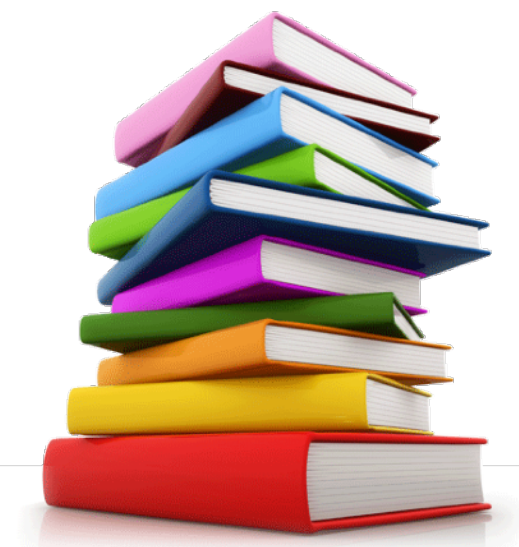


(1 + ((2 + 3) * (4 * 5)))
(1 + (5 * (4 * 5)))

A diagram showing the reduction of the expression (4 * 5) to 20. A yellow box highlights the closing parenthesis of the innermost expression. Two curved arrows, one orange and one grey, point from the box to the number 20, indicating the result of the multiplication.

- Avancer jusqu'au prochain ‘)’.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

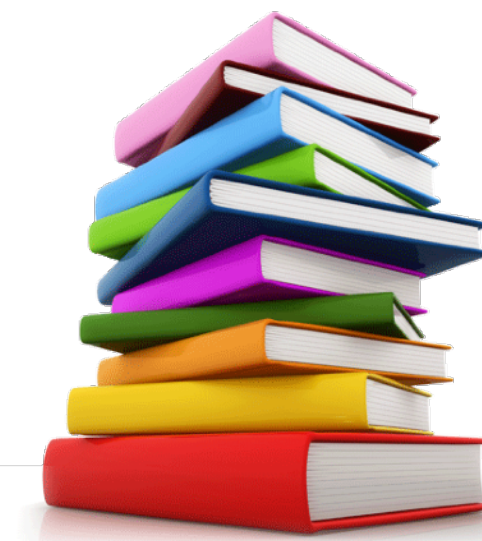
Première intuition



(1 + ((2 + 3) * (4 * 5)))
(1 + (5 * (4 * 5)))
(1 + (5 * 20))

- Avancer jusqu'au prochain ') '.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

Première intuition

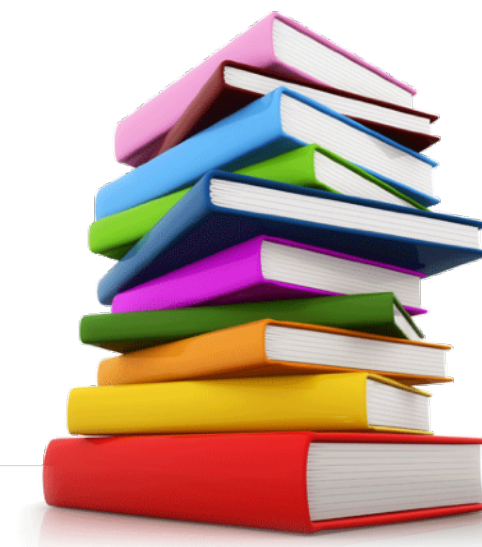


(1 + ((2 + 3) * (4 * 5)))
(1 + (5 * (4 * 5)))
(1 + (5 * 20))

A diagram illustrating the reduction of the expression (5 * 20) to 100. It shows three lines of text. The first line is the full expression: (1 + ((2 + 3) * (4 * 5))). The second line shows the result of the innermost operation: (1 + (5 * (4 * 5))). The third line shows the result of the next operation: (1 + (5 * 20)). An orange arrow points from the '20' in the third line to the '5' in the second line. A grey arrow points from the '*' in the third line to the '*' in the second line. A yellow box highlights the closing parenthesis ')' in the third line.

- Avancer jusqu'au prochain ') '.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

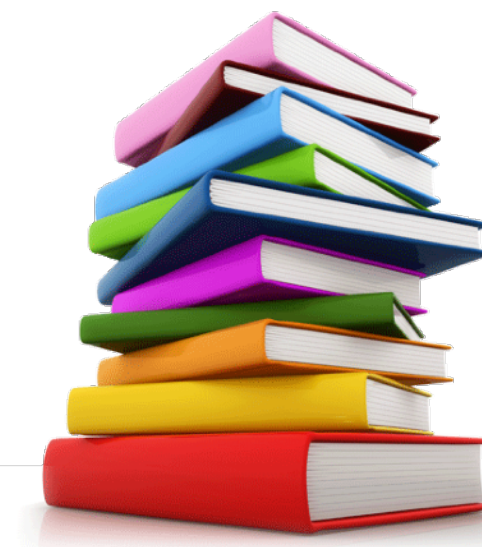
Première intuition



$$\begin{aligned}
 & (1 + ((2 + 3) * (4 * 5))) \\
 & (1 + (5 * (4 * 5))) \\
 & (1 + (5 * 20)) \\
 & (1 + 100 \quad \text{■})
 \end{aligned}$$

- Avancer jusqu'au prochain ') '.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

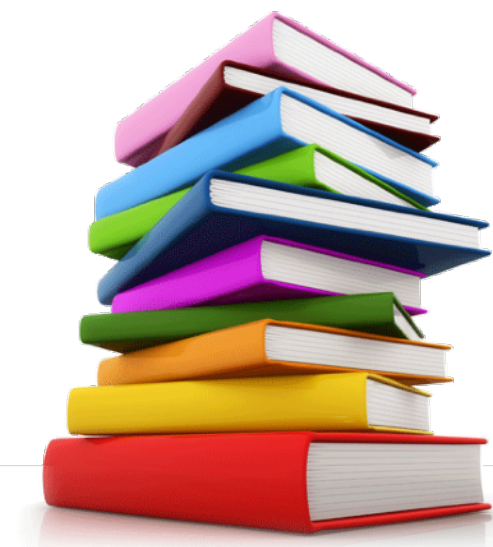
Première intuition



$$\begin{array}{l}
 (1 + ((2 + 3) * (4 * 5))) \\
 (1 + (5 * (4 * 5))) \\
 (1 + (5 * 20)) \\
 (1 + 100)
 \end{array}$$

- Avancer jusqu'au prochain ') '.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

Première intuition

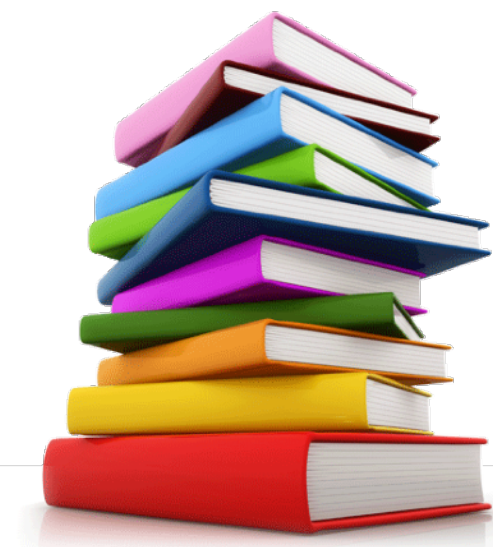


(1 + ((2 + 3) * (4 * 5)))
(1 + (5 * (4 * 5)))
(1 + (5 * 20))
(1 + 100)
101



- Avancer jusqu'au prochain ‘)’.
- Calculer l'opération sur les 2 derniers éléments.
- Remplacer ces éléments et leur opérateur par le résultat.

L'algorithme

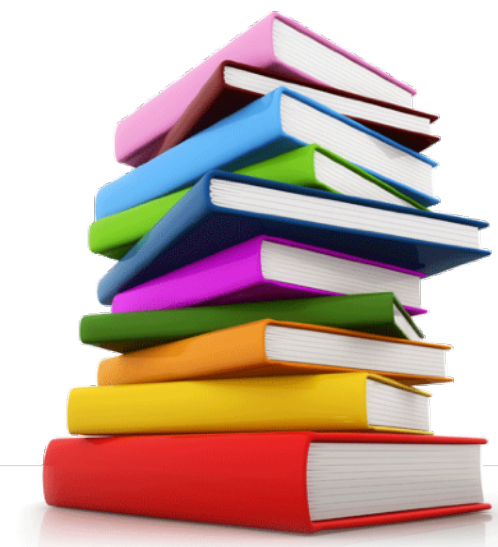


(1 + ((2 + 3) * (4 * 5)))

Valeurs

Opérateurs

L'algorithme



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

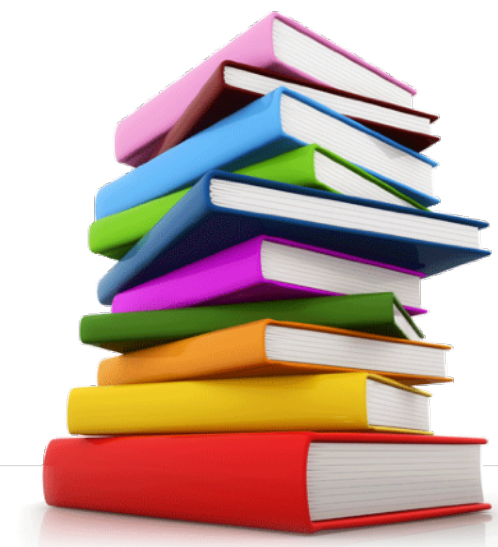
- Ajouter chaque valeur à une pile.

1

Valeurs

Opérateurs

L'algorithme



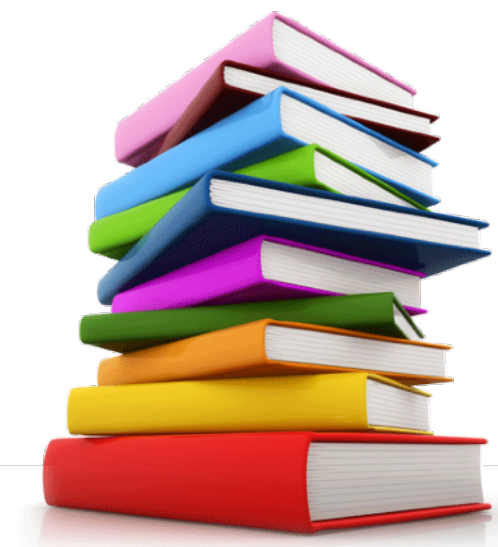
(1 + ((2 + 3) * (4 * 5)))

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.

1
Valeurs

+
Opérateurs

L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `),`

3

2

1

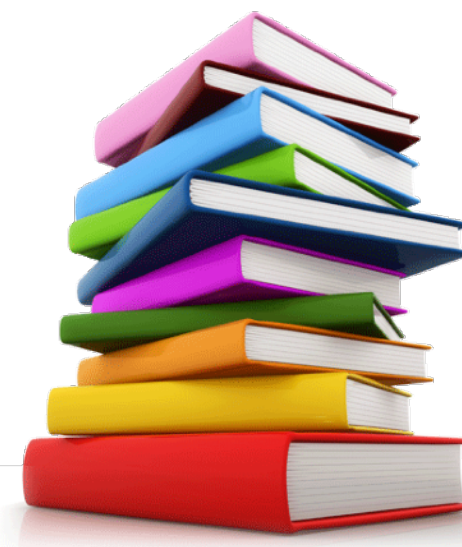
Valeurs

+

+

Opérateurs

L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)` ,
 - Retirer le dernier opérateur et les deux dernières valeurs.

3

2

1

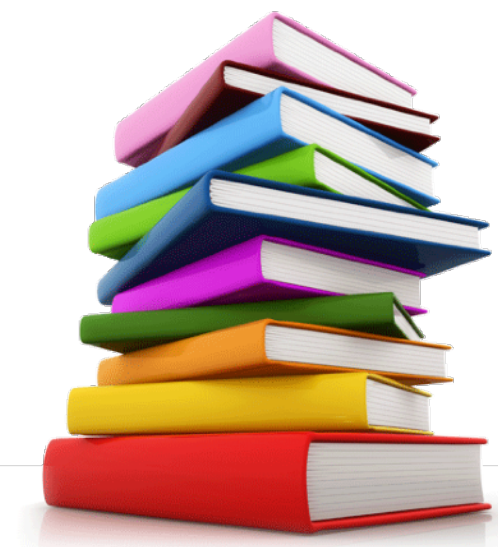
Valeurs

+

+

Opérateurs

L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)`,
 - Retirer le dernier opérateur et les deux dernières valeurs.
 - Ajouter le résultat du calcul sur la pile des valeurs.

5

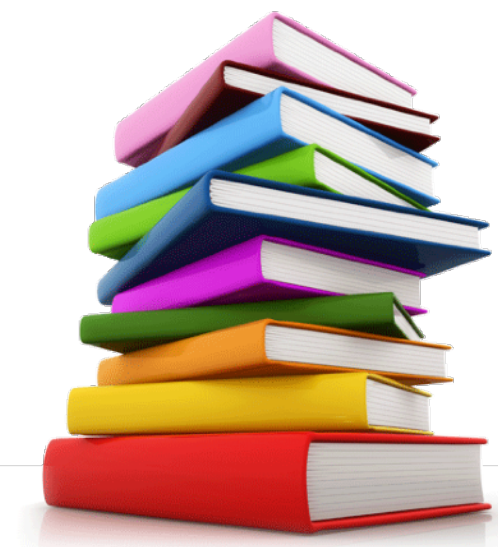
1

Valeurs

+

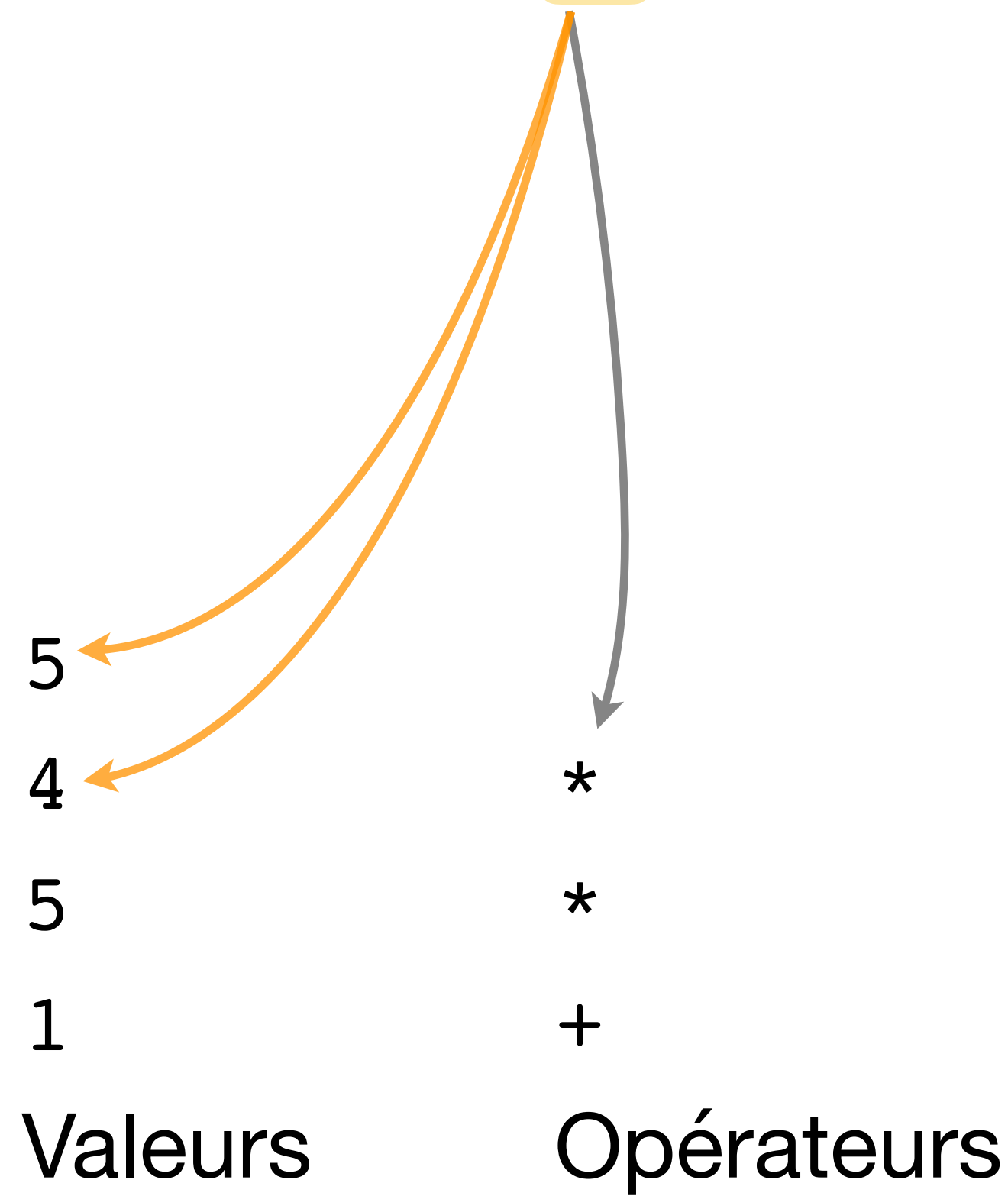
Opérateurs

L'algorithme

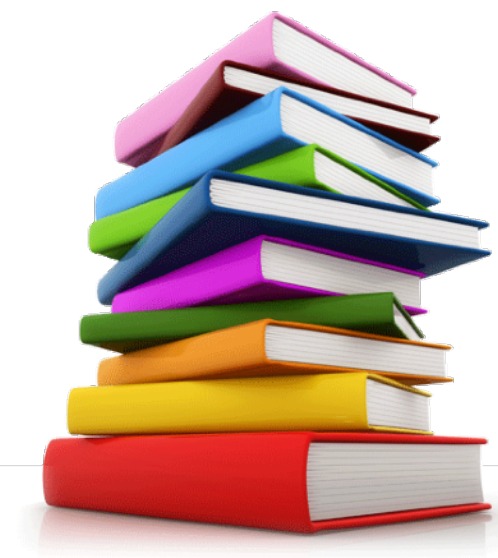


(1 + ((2 + 3) * (4 * 5)))

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)`,
 - Retirer le dernier opérateur et les deux dernières valeurs.
- Ajouter le résultat du calcul sur la pile des valeurs.



L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)` ,
 - Retirer le dernier opérateur et les deux dernières valeurs.
 - Ajouter le résultat du calcul sur la pile des valeurs.

20

5

*

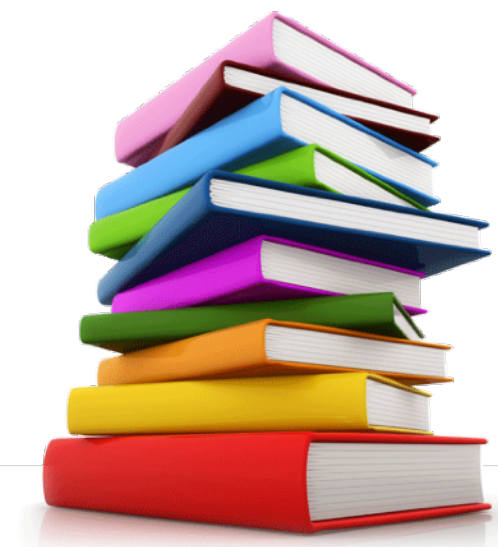
1

+

Valeurs

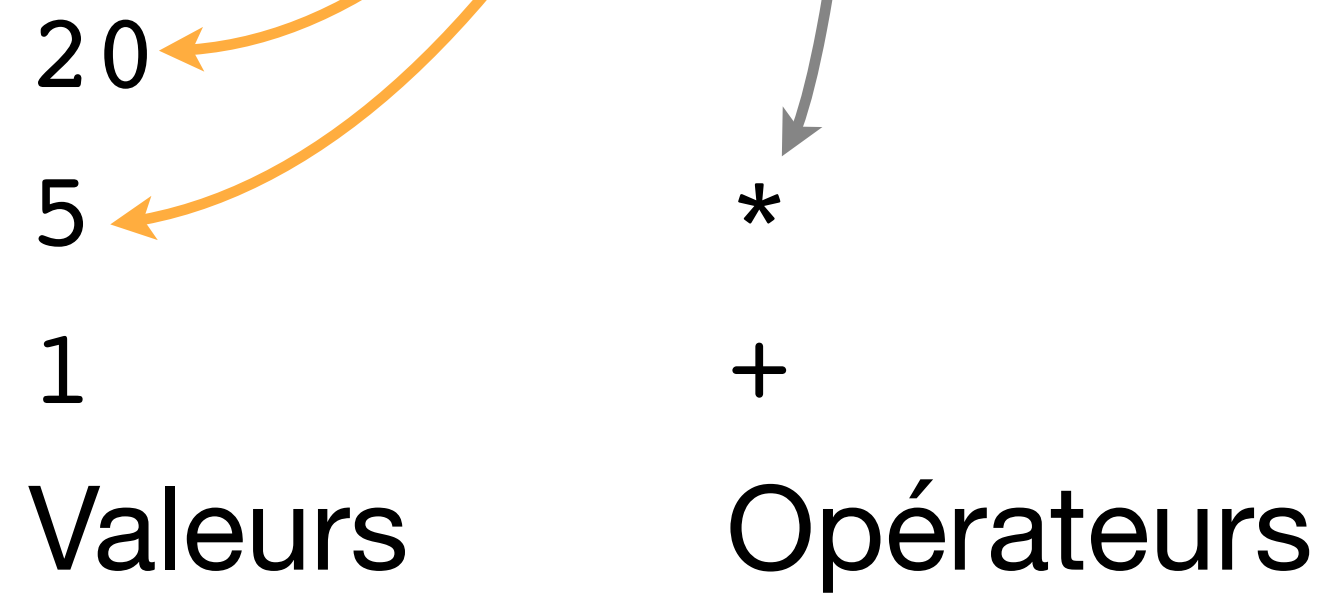
Opérateurs

L'algorithme

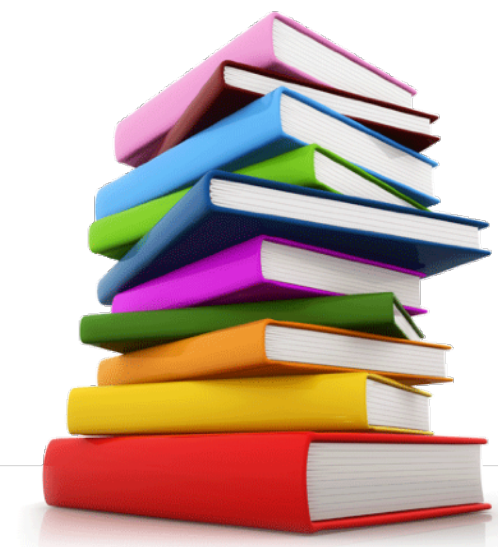


(1 + ((2 + 3) * (4 * 5)))

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)` ,
 - Retirer le dernier opérateur et les deux dernières valeurs.
- Ajouter le résultat du calcul sur la pile des valeurs.



L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)`,
 - Retirer le dernier opérateur et les deux dernières valeurs.
 - Ajouter le résultat du calcul sur la pile des valeurs.

100

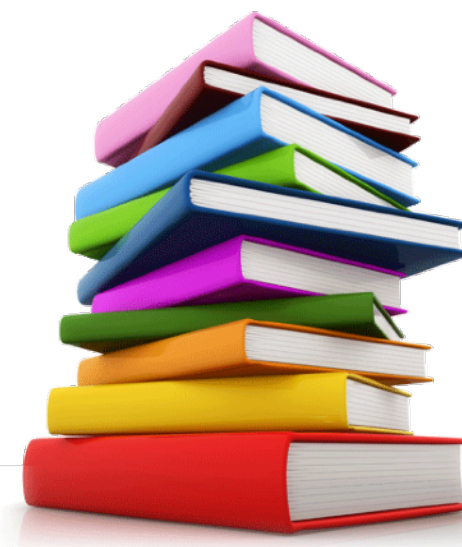
1

Valeurs

+

Opérateurs

L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)`,
 - Retirer le dernier opérateur et les deux dernières valeurs.
- Ajouter le résultat du calcul sur la pile des valeurs.

100

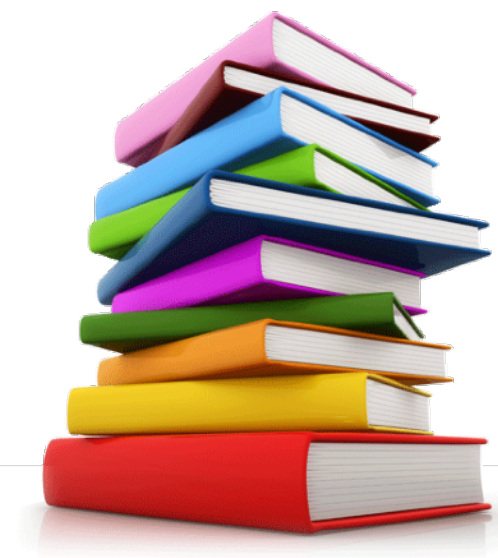
1

Valeurs

+

Opérateurs

L'algorithme



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

- Ajouter chaque valeur à une pile.
- Ajouter chaque opérateur à une pile.
- À chaque `)` ,
 - Retirer le dernier opérateur et les deux dernières valeurs.
- Ajouter le résultat du calcul sur la pile des valeurs.

101

Valeurs

Opérateurs

Avec 2 piles



Edsger Dijkstra (1930-2002)

```
fonction évaluer(expression):
```

```
    pileOp ← Pile vide
```

```
    pileVal ← Pile vide
```

```
    tant qu'il reste des éléments à lire
```

```
        e ← lire un élément(expression)
```

```
        si e est '('
```

```
            ignorer
```

```
        sinon, si e est un opérateur
```

```
            pileOp.push(e)
```

```
        sinon, si e est une valeur numérique
```

```
            pileVal.push(e)
```

```
        sinon, si e est ')'
            v2 ← pileVal.pop()
```

```
            v1 ← pileVal.pop()
```

```
            op ← pileOp.pop()
```

```
            r ← v1 op v2
```

```
            pileVal.push(r)
```

```
            pileVal.push(r)
```

```
retourner pileVal.pop()
```



HEIG-VD Exemple

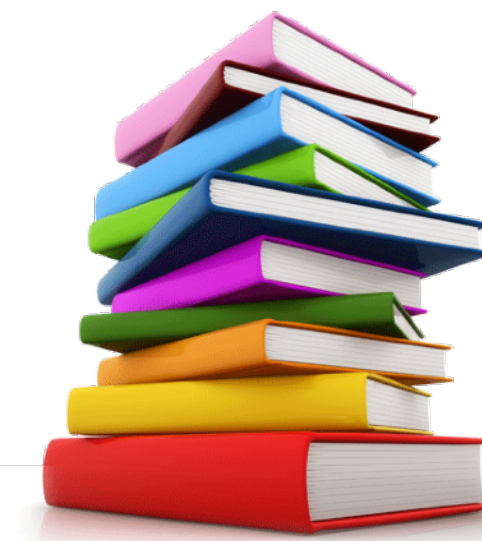
si e est un opérateur
pileOp.push(e)

sinon, si e est numérique
pileVal.push(e)

sinon, si e est ')'
v2 ← pileVal.pop()
v1 ← pileVal.pop()
op ← pileOp.pop()
r ← évaluer v1 op v2
pileVal.push(r)

pileVal	pileOp	Expression restante à évaluer
		(1 + ((2 + 3) * (4 * 5)))
1		+ ((2 + 3) * (4 * 5))
1	+	((2 + 3) * (4 * 5))
1 2	+	+ 3) * (4 * 5))
1 2	+ +	3) * (4 * 5))
1 2 3	+ +	) * (4 * 5))
1 5	+	* (4 * 5))
1 5	+ *	(4 * 5))
1 5 4	+ *	* 5))
1 5 4	+ * *	5))
1 5 4 5	+ * *	))
1 5 20	+ *	))
1 100	+	)
101		

Notation polonaise inverse



Jan Łukasiewicz (1878-1956)

- Pour évaluer, l'ordre opérateur / opérande n'importe pas

(v1 op v2) pourrait s'écrire (v1 v2 op)

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

≡

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

- En plaçant l'opérateur après ses opérandes, plus besoin de parenthèses !

1 2 3 + 4 5 * * +

- Une seule pile suffit alors.



18. File (queue) et file de priorité

File



- Insérer d'un côté, supprimer de l'autre
- “FIFO” - First In, First Out
- 4 opérations
 - insérer (push/enqueue)
 - supprimer (pop/dequeue)
 - premier élément (front)
 - dernier élément (back)



File



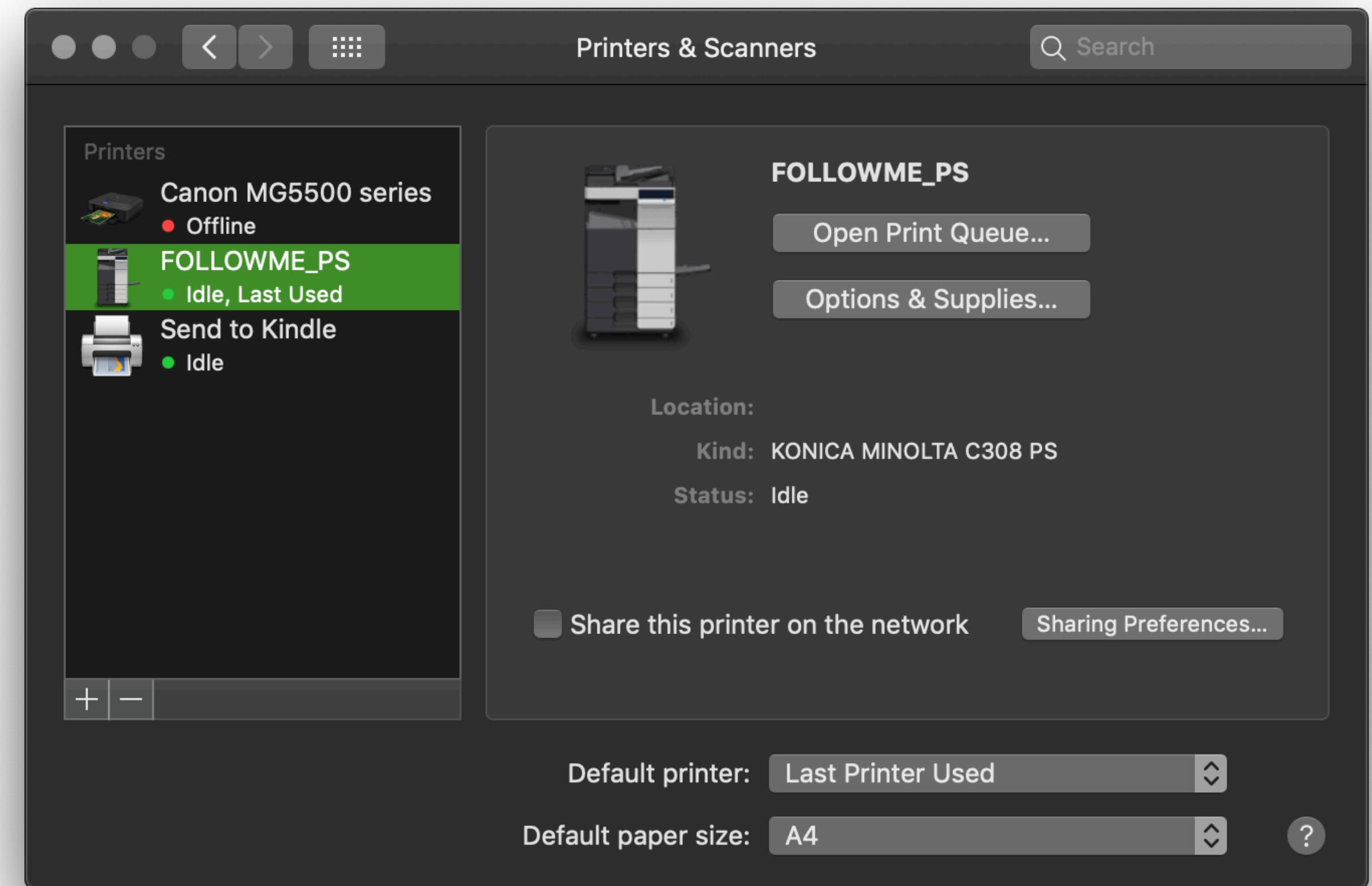
Comme le pile, la file est un **type de données abstrait**.

Rien n'impose la structure concrète sous-jacente.

Structure concrète	Insérer	Supprimer
Buffer circulaire	début ou fin	fin ou début
Liste simple avec pointeur sur le dernier élément	fin	début
Liste doublement chaînée	début ou fin	fin ou début
deque	début ou fin	fin ou début



Entrées au clavier, au cas où le système est plus lent que l'input.



File d'attente d'impression.

File de priorité



- File organisée selon un critère de priorité
- Mise en oeuvre typiquement avec un tas
- 3 opérations
 - insérer (push)
 - accéder à l'élément le plus prioritaire (top)
 - supprimer l'élément le plus prioritaire (pop)



- Compression de données - code de Huffman (PKZIP, JPEG, MP3)
- Algorithmes sur les graphes - Dijkstra , Prim (ASD2)
- Simulation à événements discrets - clients en attente, collision de particules, ...
- Statistiques - recherche de M plus grandes valeurs dans un ensemble
- Système d'exploitation - balance de charge, gestion des interruptions

Chemin le plus court



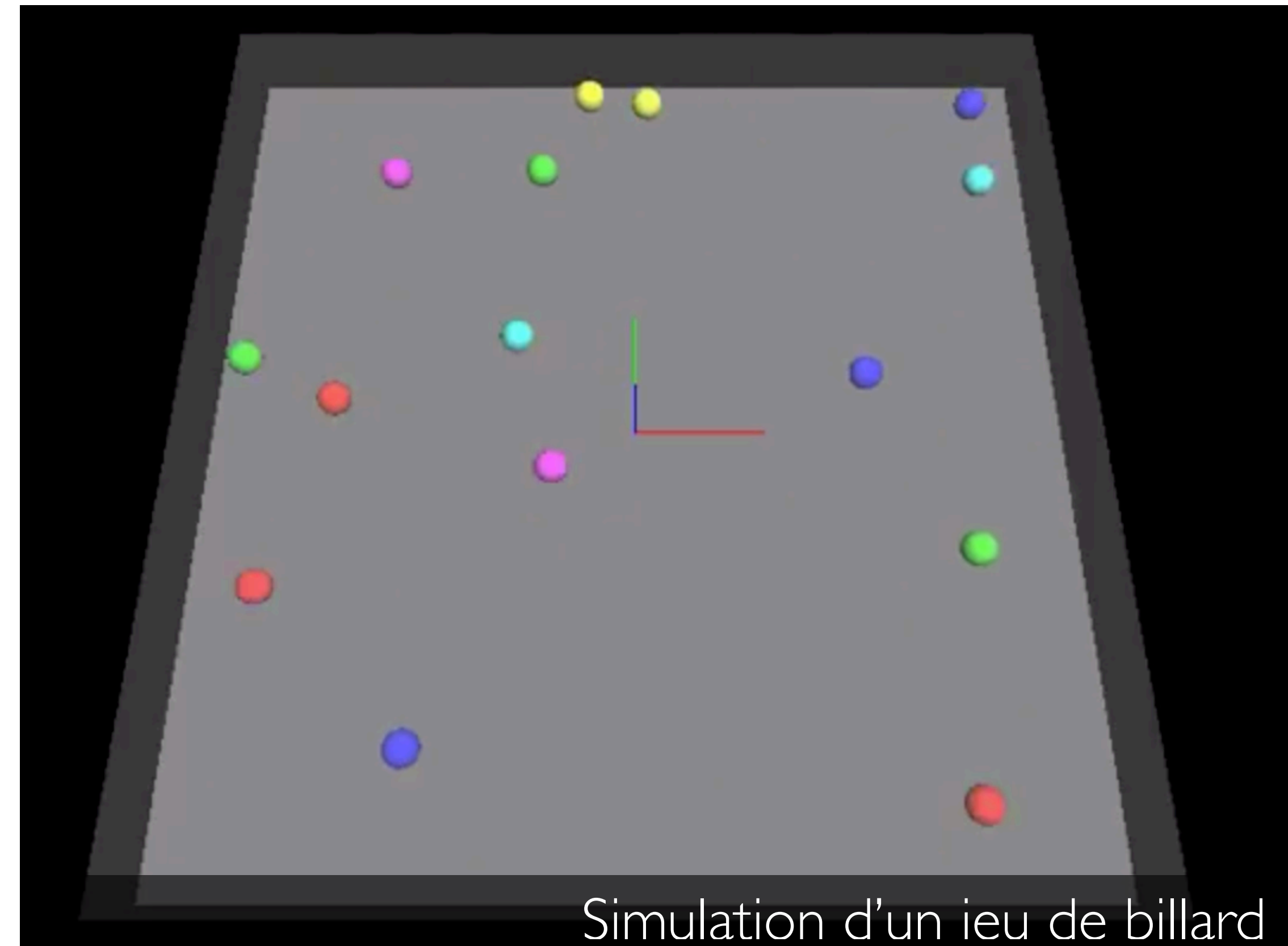
- Algorithme de parcours d'un graphe par distance croissante du point d'origine
- A chaque sommet du graphe, on calcule la distance à l'origine des sommets voisins en passant par le sommet courant
- On traite les sommets par distance à l'origine croissante.



Simulation à évènements discrets : jeu de billard



- Quand on lance une boule, on calcule toutes les collisions possibles avec les bords / les autres boules
- On place ces évènements dans une liste de priorité priorisée par le temps
- On traite le premier évènement, qui modifie la trajectoire d'une ou plusieurs boules. Pour ces boules on calcule de nouveaux évènements
- Quand on sort un évènement de la liste, on commence par vérifier que les boules associées ont toujours les positions et vitesses stockées.



Simulation d'un jeu de billard



19. STL container adaptors

Container adaptors



- Utilisent les sequence containers précédents
- Fournissent une interface spécifique

Classe	Description
<u><code>std::stack<T></code></u>	pile <i>Last In First Out</i>
<u><code>std::queue<T></code></u>	file <i>First In First Out</i>
<u><code>std::priority_queue<T></code></u>	file de priorité

`std::stack <T, Container>`



- TDA Pile
- A besoin d'un `Container` fournissant les opérations en fin : `back`, `push_back`, `emplace_back`, `pop_back`.
- Peut utiliser `std::list<T>`, `std::vector<T>`, ou `std::deque<T>` (par défaut)

Méthode	Description
<code>empty()</code>	le conteneur est-il vide?
<code>size()</code>	nombre d'éléments
<code>push(t)</code> <code>emplace(t)</code>	ajoute un élément au sommet
<code>top()</code>	référence à l'élément au sommet
<code>pop()</code>	supprime l'élément au sommet

`std::queue <T, Container>`



- TDA File
- A besoin d'un `Container` fournissant les opérations en début et en fin : `front`, `back`, `push_back`, `pop_front`
- Peut utiliser `std::list<T>` ou `std::deque<T>` (par défaut)

Méthode	Description
<code>empty ()</code>	le conteneur est-il vide?
<code>size ()</code>	nombre d'éléments
<code>push (t)</code> <code>emplace (t)</code>	insère un élément
<code>front ()</code>	référence au prochain élément
<code>back ()</code>	référence au dernier élément
<code>pop ()</code>	supprime l'élément suivant

std::priority_queue <T, Container, Compare>



- TDA File de priorité
- Besoin d'un `Container` pouvant stocker un tas : `RandomAccessIterator` et opérations en fin
- `priority_queue::push(v)` effectue
`c->push_back(v);`
`push_heap(c.begin(), c.end());`
- Peut utiliser `std::vector<T>` (par défaut) ou `std::deque<T>`
- La fonction de comparaison est `T::operator<(...)` par défaut

Méthode	Description
<code>empty()</code>	le conteneur est-il vide?
<code>size()</code>	nombre d'éléments
<code>push(t)</code> <code>emplace(t)</code>	insère un élément
<code>front()</code>	référence à l'élément le plus prioritaire
<code>pop()</code>	supprime l'élément le plus prioritaire



Création d'une file de priorité d'entiers, utilisant un `std::vector<int>` comme container et `std::greater<int>` comme comparateur.

Le comparateur `std::greater<int>` retourne le résultat de `T::operator>()`, le sommet est donc le minimum.

```
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;  
minHeap.push(30);  
minHeap.push(10);  
minHeap.push(20);  
  
while (!minHeap.empty()) {  
    std::cout << minHeap.top() << '\n';  
    minHeap.pop();  
}
```