IN103
Résolution de problèmes algorithmiques
une approche fondée sur les structures de données

Alexandre Chapoutot

Année académique 2023-2024
https://perso.ensta-paris.fr/~chapoutot/teaching/in103/

Plan du cours

1. Tas

2. Ensembles disjoints

Une nouvelle organisation des données

Jusqu'à présent, des structures de données linéaires ont été étudiées :

- tableau : taille fixe, accès direct
- liste chainée : taille variable, accès indirect
- pile : comportement LIFO
- file : comportement FIFO
- ensemble : unicité des éléments
- arbres binaires de recherche (équilibrés)

Les arbres sont une structure de données associées à une relation d'ordre mais pas que

Tas

Motivation pour une nouvelle structure de données

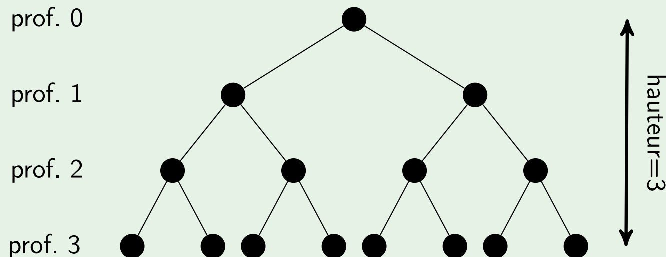
- Les **files** qui se comportent comme des files d'attente. On ne modifie pas l'ordre interne des éléments.
- Une **file de priorité** est une structure dynamique
  Fonctionnement comme une salle d'attente aux urgences : à chaque personne est attribuée une priorité de passage en fonction de la gravité de ses blessures et le traitement des patients est réalisé dans l'ordre des priorités avec potentiellement des préemptions suivant les nouvelles arrivées.
- Mise en œuvre par une structure de données **tas**
- **Applications**
  - Dans les systèmes d'exploitation, choix de l'ordre d'exécution des tâches
  - Dans les systèmes de correction orthographique, choix des propositions de correction (en fonction de distance comme celle de Levenshtein)
  - Tri de données
  - Dans les algorithmes de graphes : cf prochains cours

Arbres : encore du vocabulaire

Arbre k-aire **complet**

Un arbre dont toutes les feuilles sont à la même profondeur et tous les nœuds internes sont de degré k.

Arbre binaire complet



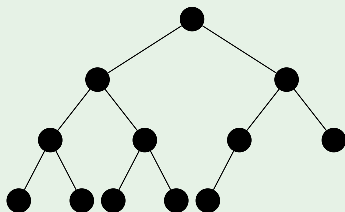
Remarque un arbre k-aire complet de hauteur h a k^h feuilles ; un arbre k-aire complet avec n feuilles a une hauteur $\log_k(n)$ (d'où l'équilibrage dans les AVL).

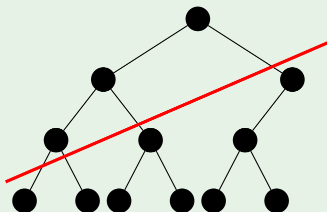
Arbres : encore du vocabulaire

Arbre *k*-aire **quasi-complet** ou **tassé**

Un arbre dont tous les niveaux sont remplis sauf éventuellement le dernier niveau qui est rempli sur la gauche.

Arbre binaire quasi-complet



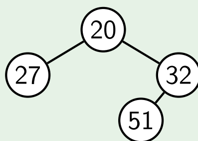


Arbres : encore du vocabulaire

Arbre binaire **partiellement ordonné** ou **arbre tournoi**

Suposons un ensemble de **clefs** muni d'un ordre total, un arbre binaire partiellement ordonné est tel que chaque nœud n a une valeur inférieure (ou supérieure) à ses enfants.

Exemple



Rappel : une relation binaire $\mathcal{R}$ sur un ensemble S est un **ordre** si elle est

- **réflexive** : $\forall x \in S,\ x\,\mathcal{R}\,x$
- **antisymétrique** $\forall x, y \in S,\ x\,\mathcal{R}\,y \wedge y\,\mathcal{R}\,x \Rightarrow x = y$
- **transitive** $\forall x, y, z \in S,\ x\,\mathcal{R}\,y \wedge y\,\mathcal{R}\,z \Rightarrow x\,\mathcal{R}\,z$

Un ordre est **total** si en plus $\forall x, y \in S,\ x\,\mathcal{R}\,y \vee y\,\mathcal{R}\,x$

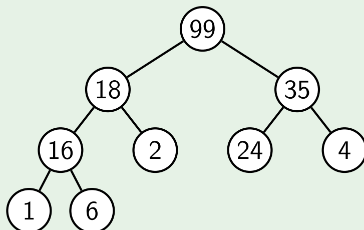
Structure de données **Tas**

Tas

Un tas est un arbre binaire quasi-complet partiellement ordonné. On parle de

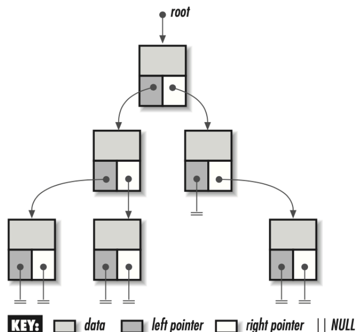
- **Tas min** quand la racine est la plus petite valeur de l'arbre
- **Tas max** quand la racine est la plus grande valeur de l'arbre

Exemple : tas max



Rappel implémentation par pointeurs des arbres

En langage C, les pointeurs sont utilisés pour faire un chaînage [1]



Définition d'un nœud

```
typedef struct integer_bitreenode_ {
  int data;
  struct integer_bitreenode_ *left;
  struct integer_bitreenode_ *right;
} integer_bitreenode_t;
```

Définition d'un arbre

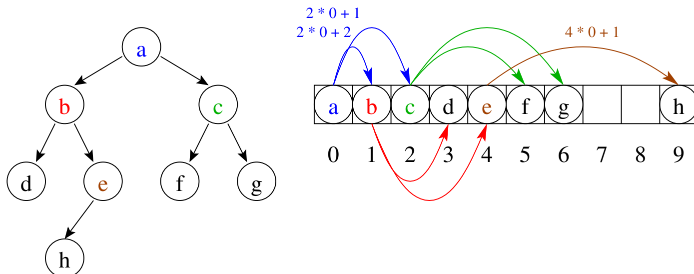
```
typedef struct integer_bitree_ {
  int size;
  integer_bitreenode_t *root;
} integer_bitree_t;
```

1. Crédit image : O'Reilly 'Mastering Algorithms with C'

Implémentation des arbres binaires par tableau

Avec un tableau :

- Un nœud a un indice i.
- Ses enfants sont aux indices $2i+1$ et $2i+2$.
- Son parent est à l'indice $\lfloor 0.5 \times (i-1) \rfloor$ (avec $\lfloor \cdot \rfloor$ la partie entière inférieure).
- $\Rightarrow$ pas besoin de pointeurs.



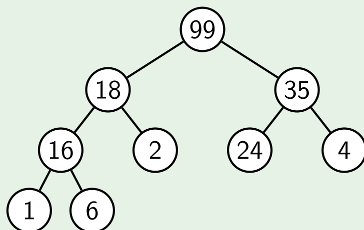
Remarque

Cette représentation est particulièrement adaptée pour la représentation des arbres binaires (quasi-) complets.

Représentation d'un tas avec un tableau

Exemple : tas max

0	1	2	3	4	5	6	7	8	9	10
99	18	35	16	2	24	4	1	6		



Remarque

La structure d'arbre quasi-complet permet d'éviter les espaces entre les éléments du tableau

Mise en œuvre dans libin103

Définition d'un tas

```
typedef enum integer_heap_type_ {
  integer_MAX_HEAP, integer_MIN_HEAP
} integer_heap_type_t;

typedef struct integer_heap_ {
  int size;
  integer_heap_type_t heap_type;
  int *tree;
} integer_heap_t;
```

Définitions de deux types

- Un type énuméré pour choisir un tas min ou tas max
- Une structure représentant un tas
  - taille
  - type de tas
  - tableau de valeurs respectant les proriétés d'arbre quasi-complet et partiellement ordonné

API des Tas

API Création/Destruction

```
void integer_heap_init(integer_heap_t*, integer_heap_type_t);

void integer_heap_destroy(integer_heap_t*);
```

API Accesseurs

```
int integer_heap_size(integer_heap_t*);
```

API Insertion/Suppression

```
int integer_heap_insert(integer_heap_t*, int);

int integer_heap_extract(integer_heap_t*, int*);
```

Complexité des opérations sur les tas

Fonction	Complexité	Commentaire
Initialisation	$\mathcal{O}(1)$	Mise à zéro des champs de la structure
Destruction	$\mathcal{O}(1)$ / $\mathcal{O}(n)$	Pour types primitifs / Parcourir tout le tableau pour supprimer les éléments
Taille	$\mathcal{O}(1)$	
Insertion	$\mathcal{O}\left(\log(n)\right)$	Quelques permutations pour maintenir la propriété d'arbre partiellement ordonné
Extraction	$\mathcal{O}\left(\log(n)\right)$	Quelques permutations pour maintenir la propriété d'arbre partiellement ordonné

Remarque

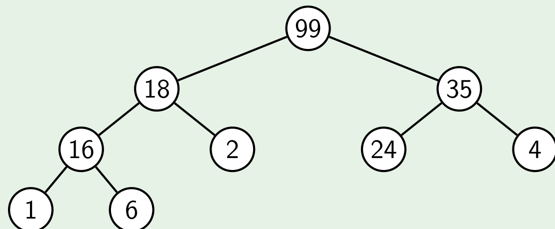
Les fonctions d'insertion et d'extraction modifient dynamiquement la taille du tableau de valeurs.

Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22

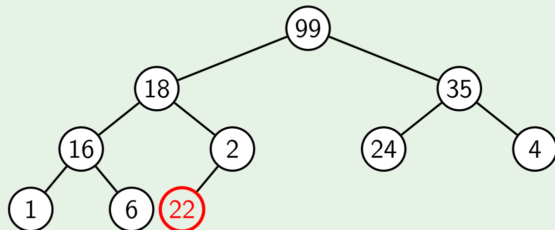


Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22



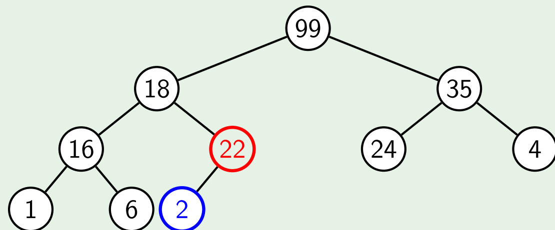
1. Insertion en fin de tableau

Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22



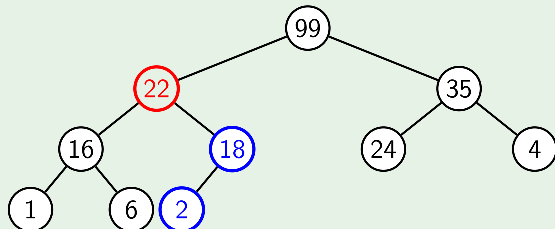
1. Insertion en fin de tableau
2. Permutation avec le parent car $2 < 22$

Insertion dans un tas max

Algorithme

- Placez la valeur dans la dernière case du tableau
- Tant que parent plus petit, permuter la valeur avec parent

Exemple : insertion de la valeur 22



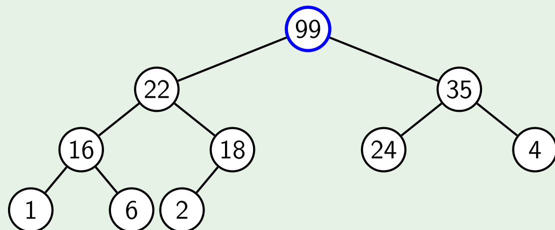
1. Insertion en fin de tableau
2. Permutation avec le parent car $2 < 22$
3. Permutation avec le parent car $18 < 22$
4. Fin

Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire **redescendre** la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction



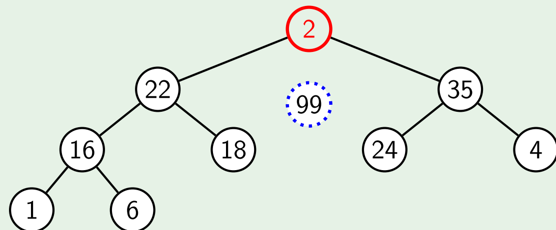
1. Extraction de la plus grande valeur

Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire **redescendre** la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction



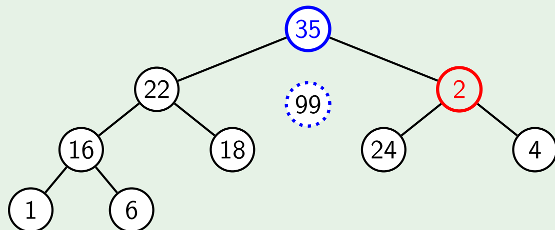
1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud

Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire **redescendre** la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction



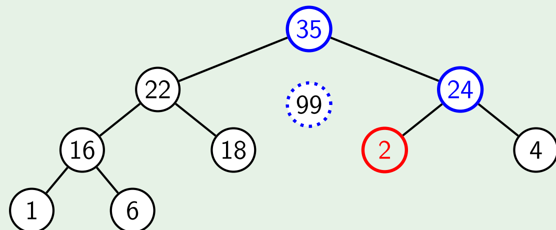
1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud
3. Permutation car $35 > 2$

Extraction dans un tas max

Algorithme

- Copier la valeur du dernier nœud du tableau à la racine
- Faire **redescendre** la valeur de la racine en permutant avec le plus grand de ses enfants.
- Tant qu'un enfant a une plus grande valeur.

Exemple : extraction



1. Extraction de la plus grande valeur
2. Permutation avec le dernier nœud
3. Permutation car $35 > 2$
4. Permutation car $24 > 2$
5. Fin

Exemple d'utilisation de la bibliothèque libin103

```c
#include <stdio.h>
#include <stdlib.h>
#include "heap.h"

int main (int argc, char* argv[]) {

  int size = 7;
  double tab[] = { 0.1, -2.0, 12.3, 3.14159, -1.34, 202.9, -2.67 };

  real_heap_t heap;
  real_heap_init (&heap, real_MIN_HEAP);

  for (int i = 0; i < size; i++) {
    real_heap_insert (&heap, tab[i]);
  }

  double min;
  real_heap_extract(&heap, &min);
  printf ("min␣=␣%f\n", min);

  real_heap_destroy (&heap);

  return EXIT_SUCCESS;
}
```

« Tassification » d'un tabeau

Une autre façon de construire un tas est de « tasser » un tableau existant.
Conséquence pas d'utilisation d'un second tableau pour le tas.

```
/* A appliquer sur tous les elements du tableau a partir de la fin */
Function heapify_max_from_index (array, index) :
    left = 2 * index + 1;
    right = 2 * index + 2;
    largest = index;
    if left < size(array) and  tab[left] > tab[largest] then
        largest = left;
    end
    if right < size(array) and  tab[rigth] > tab[largest] then
        largest = right;
    end
    if i != largest then
        swap (array[i], array[largest]);
        heapify_max_from_index (array, largest);
    end
```

Ensembles disjoints

Motivation pour une nouvelle structure de données

- La structure de données `Union-Find` (ou ensemble disjoint) est utilisée pour représenter des partitions d'un ensemble (d'entiers).
- La structure de données a 2 opérations :
  - `find(u)` : donne le sous-ensemble auquel appartient l'élément u. Un sous-ensemble est représenté par un témoin (un de ses éléments)
  - `union(u,v)` fusionne les deux sous-ensembles représentés par u et v.

De nombreuses applications, cf plus loin

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

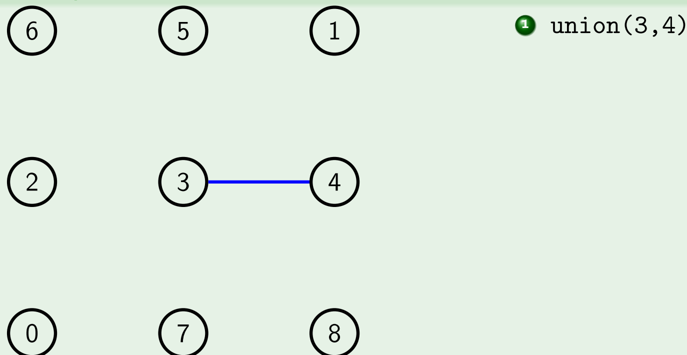
Exemple

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



1. `union(3,4)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple

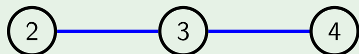
1. `union(3,4)`
2. `union(8,0)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



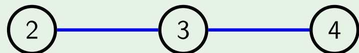
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



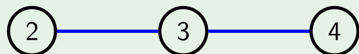
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



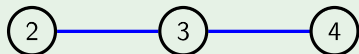
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



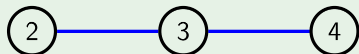
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



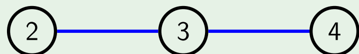
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



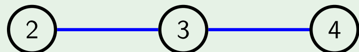
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`

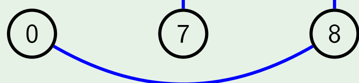
Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple





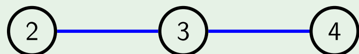
1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`
9. `union(4,8)`

Fonctionnement d'un union-find

Point de départ :

- l'ensemble $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
- représenté par 9 sous-ensembles singletons

Exemple



1. `union(3,4)`
2. `union(8,0)`
3. `union(2,3)`
4. `union(5,6)`
5. `are_connected(0,2)` $\Rightarrow$ non
6. `are_connected(2,4)` $\Rightarrow$ oui
7. `union(5,1)`
8. `union(7,3)`
9. `union(4,8)`
10. `are_connected(0,2)` $\Rightarrow$ oui
11. `are_connected(2,4)` $\Rightarrow$ oui

Relation d'équivalence et classe d'équivalence

Relation d'équivalence

Une relation $\mathcal{R}$ définie sur un ensemble S respectant les propriétés :

- **Reflexivité** : $\forall a \in S,\ a \mathcal{R} a$
- **Symmétrie** : $\forall a, b \in S,\ a \mathcal{R} b \Leftrightarrow b \mathcal{R} a$
- **Transitivité** : $\forall a, b, c \in S,\ a \mathcal{R} b \wedge b \mathcal{R} c \Rightarrow a \mathcal{R} c$

Classe d'équivalence

La classe d'équivalence de l'élément $a \in S$ est le sous-ensemble S_a de S tel que $S_a = \{x \in S : x \mathcal{R} a\}$.
Remarque l'ensemble des classes d'équivalence forme une partition.

La structure de données union-find permet de calculer dynamiquement les classes d'équivalence d'un ensemble avec

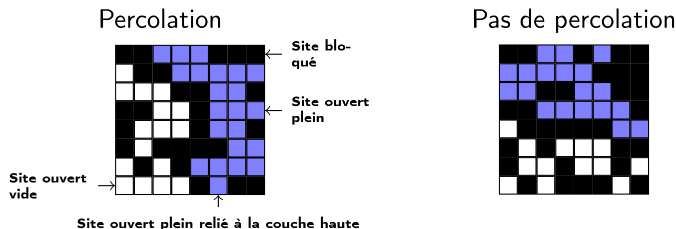
- l'opération `find(a)` qui donne la classe d'équivalence de a
- l'opération `union(x,y)` qui joint les classes d'équivalence de x et y en une seule classe.

Applications des union-find

Percolation

Désigne le passage d'un fluide à travers d'un milieu plus ou moins perméable.

Problèmes similaires : la conductivité d'un matériaux ou la facilité de communication dans les réseaux sociaux.



Algorithme de Hoshen–Kopelman

- On considère une grille avec des cases notées occupées et libres
- Balayer des cases libres et numéroter les ensembles connexes.
- Percolation si les lignes de dessus et du dessous sont connectées.

Applications des union-find

- **Segmentation d'images** (similaire au problème de percolation)
- **Utile dans les algorithmes de graphes** (cf plus loin dans le cours)
- **Procédure de décision en logique** avec formules avec égalité et fonctions non interprétées
  - Permet de prouver l'insatisfiabilité par l'algorithme de fermeture congurente (*Congruence Closure Algorithm*)
  - P. ex., $f(a,b) = a \land f(f(a,b),b) \neq a$ est insatisfiable
    car a, $f(a,b)$ et $f(f(a,b),b)$ sont dans la même classe d'équivalence
- ...

Implémentation des union-find avec des forêts

Idées générales

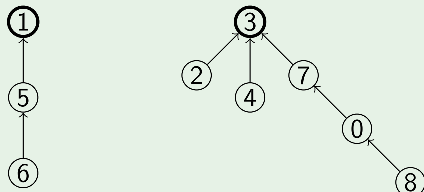
- Les éléments d'un ensemble sont stockés dans des nœuds d'arbres *n*-aires ;
- Il y a un arbre par classe d'équivalence d'où la notion de forêt ;
- La racine de l'arbre représente le **témoin**/**représentant** de la classe d'équivalence (valeur retournée par la fonction `find`) ;
- Les nœuds de l'arbre pointent vers les parents
  Attention c'est l'inverse des structures de données aborescentes vues jusqu'à présent !

Exemple

Une représentation des sous-ensembles

- $\{6, \underline{1}, 5\}$
- $\{0, 2, \underline{3}, 4, 7, 8\}$

pourrait être



Mise en œuvre dans libin103

Définition d'un union-find

```
typedef struct integer_uf_elm {
    struct integer_uf_elm *parent;
    int value;
    int depth;
} integer_uf_elm_t;

typedef struct integer_uf {
  integer_uf_elm_t** forest;
  int size;
  int components;
} integer_uf_t;
```

Remarque

Cette structure de données n'existe qu'en version avec des valeurs entières !

Un type élément d'un arbre (`integer_uf_elm_t`)

- avec un pointeur vers le parent
- une donnée
- une profondeur dans l'arbre

Un type pour la structure de donnée (`integer_uf_t`)

- un tableau de nœuds
- une taille du tableau, c'est-à-dire, la cardinalité de l'ensemble.
- un nombre de classes d'équivalence

API des union-find

API Création/Destruction

```
/* On donne le nombre max d'elements de l'ensemble */
void integer_uf_init (integer_uf_t*, int size);
void integer_uf_destroy(integer_uf_t*);
```

API Accesseurs et prédicats

```
int integer_uf_size (integer_uf_t*);
int integer_uf_components (integer_uf_t*);
bool integer_uf_are_connected (integer_uf_t*, int , int);
```

API Insertion/Suppression

```
int integer_uf_add_element(integer_uf_t*, int);
int integer_uf_find(integer_uf_t*, int , integer_uf_elm_t** result);
int integer_uf_union(integer_uf_t*, int , int);
```

Complexité des opérations sur les union-find

Fonction	**Complexité**	**Commentaire**
Initialisation	$\mathcal{O}(1)$	Mise à zéro des champs de la structure (et allocation tableau)
Destruction	$\mathcal{O}(n)$	Parcourir tout le tableau pour supprimer les éléments
Taille / Composantes	$\mathcal{O}(1)$	
Union	$\mathcal{O}(\alpha(n))$	Repose sur la fonction `find`
Find	$\mathcal{O}(\alpha(n))$	Si utilisation d'heuristiques "path compression" et "union by rank"

Remarque

Les fonctions d'insertion et d'extraction modifient dynamiquement la taille du tableau de valeurs, gérée par la fonction `realloc`.

Illustration implémentation des union-find

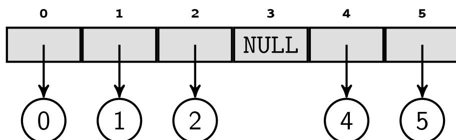
0	1	2	3	4	5
NULL	NULL	NULL	NULL	NULL	NULL

1. `init (&dset, 6);`

Remarque

On utilise des entiers pour avoir un accès direct aux éléments de l'ensemble avec les indices de tableau.

Illustration implémentation des union-find

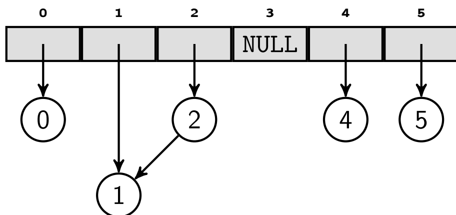


1. `init (&dset, 6);`
2. `add_element (&dset, 0);`
   (idem pour 1, 2, 4 et 5)

Remarque

On utilise des entiers pour avoir un accès direct aux éléments de l'ensemble avec les indices de tableau.

Illustration implémentation des union-find



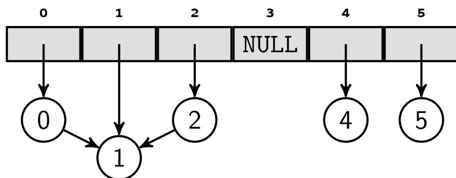
1. `init (&dset, 6);`
2. `add_element (&dset, 0);`
   (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



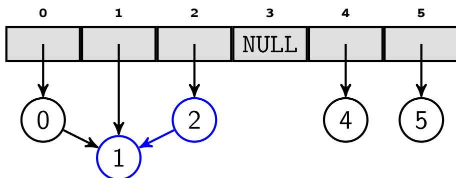
1. `init (&dset, 6);`
2. `add_element (&dset, 0);`
   (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



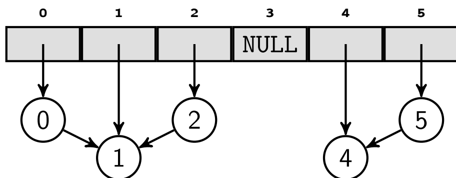
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



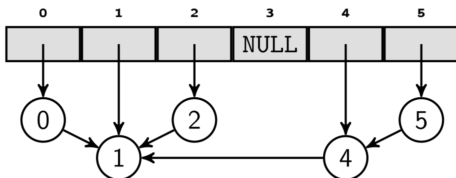
1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`
6. `union (&dset, 4, 5);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Illustration implémentation des union-find



1. `init (&dset, 6);`
2. `add_element (&dset, 0);` (idem pour 1, 2, 4 et 5)
3. `union (&dset, 1, 2);`
4. `union (&dset, 0, 1);`
5. `find(&dset, 2);`
6. `union (&dset, 4, 5);`
7. `union (&dset, 2, 4);`

Remarque

Opération d'union naïve a une complexité O(1) (mise à jour de pointeurs).

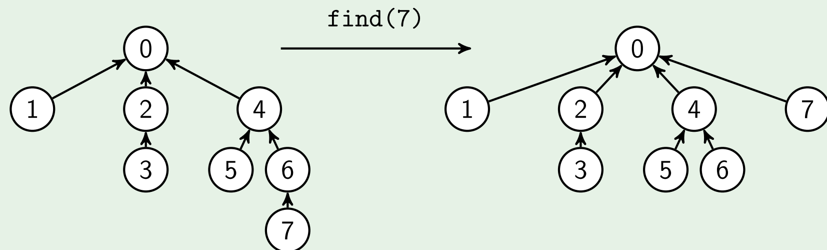
- Sans stratégie la hauteur de l'arbre peut grandir avec un impact sur la complexité de la fonction `find` $\in \mathcal{O}(n)$
- Stratégie *union by rank* on fait pointer le plus petit arbre vers le plus grand : `find` $\in \mathcal{O}(\log(n))$

Opérations d'union-find améliorées

Path compression

Lors de l'appel à `find` le nœud en argument est modifié pour pointer directement vers la racine.

Exemple du fonctionnement dans la bibliothèque `libin103`



Remarque

On peut encore améliorer la complexité en reliant tous les nœuds sur le chemin vers la racine dans l'appel à `find` à la racine.

Un mot sur la complexité (pour le fun)

La complexité amortie[2] de la fonction `find` est $\mathcal{O}(m\alpha(n))$ avec m le nombre d'opérations et n le nombre d'éléments dans l'ensemble.

Analyse de complexité amortie

Calcul de complexité sur **l'exécution d'une séquence d'opérations** sur une structure des données. (En général, analyse difficile)

Fonction d'Ackermann

$$A(0,n) = n+1$$
$$A(m+1,0) = A(m,1)$$
$$A(m+1,n+1) = A(m,A(m+1,n))$$

- A a une croissance très très rapide,
- Par exemple, $A(4,2)$ est un nombre avec 19729 chiffres équivalent à $2^{65536}-3$!

α représente la fonction inverse de la fonction d'Ackermann ($\alpha(n) = \min\{k : A(k,1) \geqslant n\}$) donc une fonction à croissance très très faible !

2. voir "Introduction to algorithms" de Thomas H. Cormen *et al.* Chap. 17 et 21

Exemple d'utilisation de la bibliothèque libin103

```
#include <stdio.h>
#include <stdlib.h>
#include "integer_uf.h"

int main (){
  int size = 5; integer_uf_t uf;
  integer_uf_init (&uf, size);

  for (int i = 0; i < size; i++) {
    integer_uf_add_element (&uf, i);
  }

  integer_uf_union (&uf, 1, 3);
  integer_uf_union (&uf, 2, 4);
  integer_uf_union (&uf, 4, 1);

  if (integer_uf_are_connected (&uf, 3, 2)) {
    printf ("3 et 2 sont dans la meme classe\n");
  }

  integer_uf_destroy (&uf);

  return EXIT_SUCCESS;
}
```

Application : génération de labyrinthe (voir TP)

- Un petit labyrinthe carré de 5 cases de côté ;
- Considérer toutes les cases de 0 à 24 comme singletons ;
- Liste des murs internes $S = \{(0,1),(0,5),(1,2),(1,6),(2,3),\dots\}$.
- M est la liste des murs à conserver durant la construction du labyrinthe.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Algorithme

- Tant que toutes les cases ne sont pas dans le même ensemble :
  - prendre au hasard $(c_1, c_2) \in S$
  - si c_1 et c_2 sont dans le même ensemble, le mur (c_1, c_2) est conservé, mettre dans M
  - si c_1 et c_2 ne sont pas dans le même ensemble, `union(`c_1`, `c_2`)` et ne plus considérer (c_1, c_2)
- $M \cup S$ sont les murs du labyrinthe.
- Choisir une entrée et une sortie

Application : génération de labyrinthe (voir TP)

- Un petit labyrinthe carré de 5 cases de côté ;
- Considérer toutes les cases de 0 à 24 comme singletons ;
- Liste des murs internes $S = \{(0,1),(0,5),(1,2),(1,6),(2,3),\dots\}$.
- M est la liste des murs à conserver durant la construction du labyrinthe.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Algorithme

- Tant que toutes les cases ne sont pas dans le même ensemble :
  - prendre au hasard $(c_1, c_2) \in S$
  - si c_1 et c_2 sont dans le même ensemble, le mur (c_1, c_2) est conservé, mettre dans M
  - si c_1 et c_2 ne sont pas dans le même ensemble, `union(`c_1`,` c_2`)` et ne plus considérer (c_1, c_2)
- $M \cup S$ sont les murs du labyrinthe.
- Choisir une entrée et une sortie

Application : génération de labyrinthe (voir TP)

Propriétés du labyrinthe

- Toutes les cases sont accessibles à partir des autres
  Structure arborescente des chemins, enracinée au niveau de l'entrée.
- Pas de cycles, c'est-à-dire, pas de création d'ilôts de murs n'étant pas reliés aux bords