Partie 4

Structures de données

Plan

1. Introduction
2. Pile et file
3. Structures linéaires : Liste, vecteur, séquence
4. Arbres
5. File à priorité
6. Ensembles disjoints

Concept

- Une structure de données est une manière d'organiser et de stocker l'information
  - Pour en faciliter l'accès ou dans d'autres buts
- Une structure de données a une interface qui consiste en un ensemble de procédures pour ajouter, effacer, accéder, réorganiser, etc. les données.
- Une structure de données conserve des données et éventuellement des méta-données
  - Par exemple : un tas utilise un tableau pour stocker les clés et une variable $A.\mathit{heap\text{-}size}$ pour retenir le nombre d'éléments qui sont dans le tas.
- Un type de données abstrait (TDA) = définition des propriétés de la structure et de son interface ("cahier des charges")

Structures de données

Dans ce cours :

- Principalement des ensembles dynamiques (dynamic sets), amenés à croître, se rétrécir et à changer au cours du temps.
- Les objets de ces ensembles comportent des attributs.
- Un de ces attributs est une clé qui permet d'identifier l'objet, les autres attributs sont la plupart du temps non pertinents pour l'implémentation de la structure.
- Certains ensembles supposent qu'il existe un ordre total entre les clés.

Opérations standards sur les structures

- Deux types : opérations de recherche/accès aux données et opérations de modifications
- Recherche : exemples :
  - $\textsc{Search}(S, k)$: retourne un pointeur x vers un élément dans S tel que $x.key = k$, ou NIL si un tel élément n'appartient pas à S.
  - $\textsc{Minimum}(S)$, $\textsc{Maximum}(S)$: retourne un pointeur vers l'élément avec la plus petite (resp. grande) clé.
  - $\textsc{Successor}(S, x)$,$\textsc{Predecessor}(S, x)$ retourne un pointeur vers l'élément tout juste plus grand (resp. petit) que x dans S, NIL si x est le maximum (resp. minimum).
- Modification : exemples :
  - $\textsc{Insert}(S, x)$: insère l'élément x dans S.
  - $\textsc{Delete}(S, x)$: retire l'élément x de S.

Implémentation d'une structure de données

- Etant donné un TDA (interface), plusieurs implémentations sont généralement possibles
- La complexité des opérations dépend de l'implémentation, pas du TDA.

- Les briques de base pour implémenter une structure de données dépendent du langage d'implémentation
  - Dans ce cours, les principaux outils du C : tableaux, structures à la C (objets avec attributs), liste liées (simples, doubles, circulaires), etc.
- Une structure de données peut être implémentée à l'aide d'une autre structure de données (de base ou non)

Quelques structures de données standards

- Pile : collection d'objets accessibles selon une politique LIFO
- File : collection d'objets accessibles selon une politique FIFO
- File double : combine accès LIFO et FIFO
- Liste : collection d'objets ordonnés accessibles à partir de leur position
- Vecteur : collection d'objets ordonnés accessibles à partir de leur rang
- Arbre : collection d'objets organisés en une structure d'arbre
- File à priorité : accès uniquement à l'élément de clé (priorité) maximale

- Dictionnaire : structure qui implémente les 3 opérations recherche, insertion, suppression (cf. partie 5)

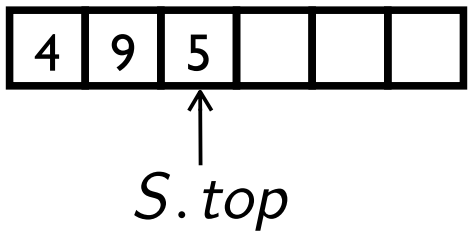
Plan

Pile

- Ensemble dynamique d'objets accessibles selon une discipline LIFO ("Last-in first-out").
- Interface
  - $\textsc{Stack-Empty}(S)$ renvoie vrai si et seulement si la pile est vide
  - $\textsc{Push}(S, x)$ pousse la valeur x sur la pile S
  - $\textsc{Pop}(S)$ extrait et renvoie la valeur sur le sommet de la pile S
- Applications :
  - Option 'undo' dans un traitement de texte
  - Langage postscript
  - Appel de fonctions dans un compilateur
  - ...
- Implémentations :
  - avec un tableau (taille fixée a priori)
  - au moyen d'une liste liée (allouée de manière dynamique)
  - ...

Implémentation par un tableau

- S est un tableau qui contient les éléments de la pile
- $S.top$ est la position courante de l'élément au sommet de S



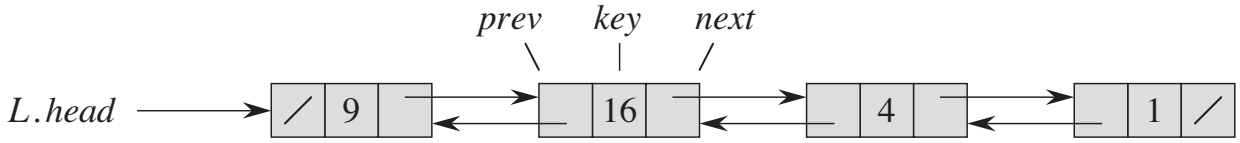

```
STACK-EMPTY(S)
1  return S.top == 0
```

```
PUSH(S, x)
1  if S.top == S.length
2        error "overflow"
3  S.top = S.top + 1
4  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

- Complexité en temps et en espace : $O(1)$
  (Inconvénient : L'espace occupé ne dépend pas du nombre d'objets)

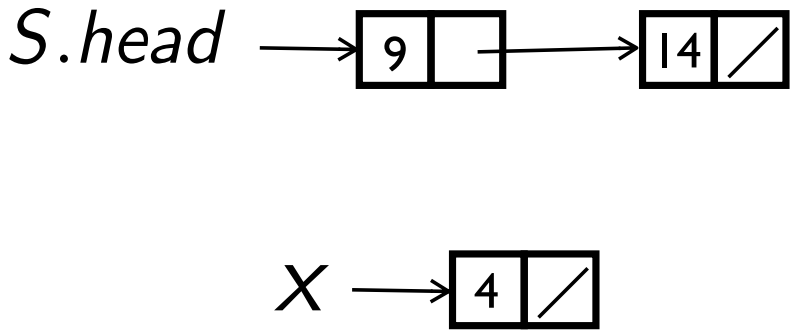
Rappel : liste simplement et doublement liée



- Structure de données composée d'une séquence d'éléments de liste.
- Chaque élément x de la liste est composé :
  - d'un contenu utile $x.data$ de type arbitraire (par exemple une clé),
  - d'un pointeur $x.next$ vers l'élément suivant dans la séquence
  - *Doublement liée :* d'un pointeur $x.prev$ vers l'élément précédent dans la séquence
- Soit L une liste liée
  - $L.head$ pointe vers le premier élément de la liste
  - *Doublement liée :* $L.tail$ pointe vers le dernier élément de la liste
- Le dernier élément possède un pointeur $x.next$ vide (noté NIL)
- *Doublement liée :* Le premier élément possède un pointeur $x.prev$ vide

Implémentation d'une pile à l'aide d'une liste liée

- S est une liste simplement liée ($S.head$ pointe vers le premier élément de la liste)




```
STACK-EMPTY(S)
1  if S.head == NIL
2        return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  x.next = S.head
2  S.head = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else x = S.head
4        S.head = S.head.next
5        return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

Application

- Vérifier l'appariement de parenthèses ([],() ou {}) dans une chaîne de caractères
  - Exemples : $((x)+(y)]/2 \rightarrow$ non, $[-(b)+\text{sqrt}(4*(a)*c)]/(2*a) \rightarrow$ oui
- Solution basée sur une pile :

```
ParenthesesMatch(A)
 1  S = pile vide
 2  for i = 1 to A.length
 3       if A[i] est une parenthèse gauche
 4            Push(S, A[i])
 5       elseif A[i] est une parenthèse droite
 6            if Stack-Empty(S)
 7                 return False
 8            elseif Pop(S) n'est pas du même type que A[i]
 9                 return False
10  return Stack-Empty(S)
```

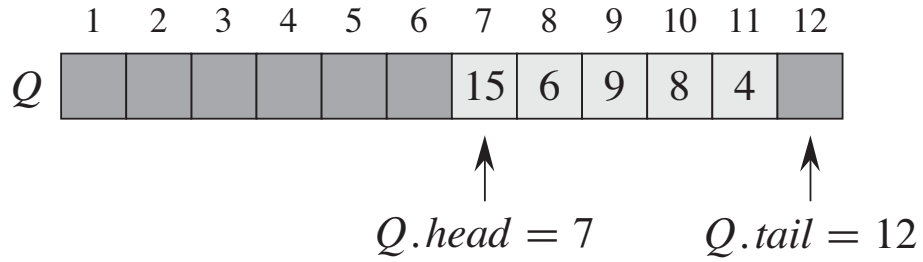
File

- Ensemble dynamique d'objets accessibles selon une discipline FIFO ("First-in first-out").
- Interface
  - $\textsc{Enqueue}(Q, s)$ ajoute l'élément x à la fin de la file Q
  - $\textsc{Dequeue}(Q)$ retire l'élément à la tête de la file Q

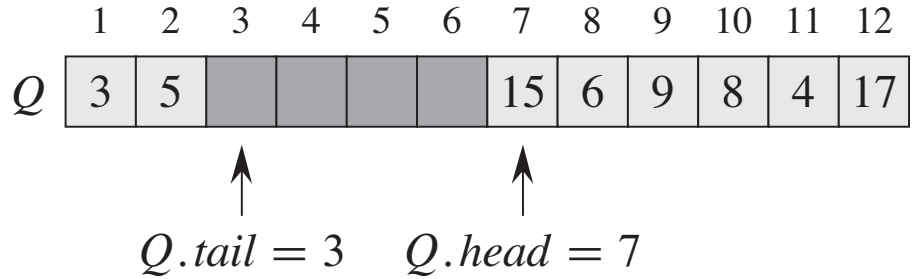
- Implémentation à l'aide d'un tableau circulaire
  - Q est un tableau de taille fixe $Q.\mathit{length}$
    - Mettre plus de $Q.\mathit{length}$ éléments dans la file provoque une erreur de dépassement
  - $Q.\mathit{head}$ est la position à la tête de la file
  - $Q.\mathit{tail}$ est la première position vide à la fin de la file
  - Initialement : $Q.\mathit{head} = Q.\mathit{tail} = 1$

Enqueue et Dequeue

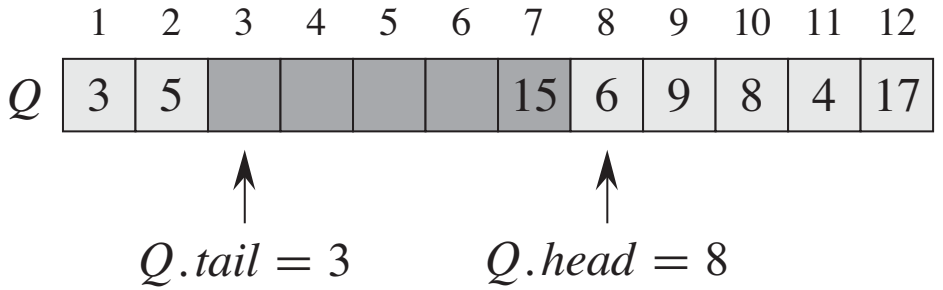
Etat initial :



Enqueue$(Q, 17)$, Enqueue$(Q, 3)$, Enqueue$(Q, 5)$



Dequeue$(Q) \rightarrow 15$



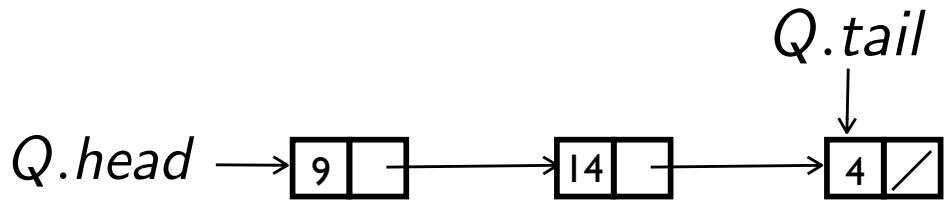
Enqueue et Dequeue

```
ENQUEUE(Q,x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(1)$.
- *Exercice : ajouter la gestion d'erreur*

Implémentation à l'aide d'une liste liée



- Q est une liste simplement liée
- $Q.head$ (resp. $Q.tail$) pointe vers la tête (resp. la queue) de la liste

```
ENQUEUE(Q,x)
1  x.next = NIL
2  if Q.head == NIL
3        Q.head = x
4  else Q.tail.next = x
5  Q.tail = x
```

```
DEQUEUE(Q)
1  if Q.head == NIL
2        error "underflow"
3  x = Q.head
4  Q.head = Q.head.next
5  if Q.head == NIL
6        Q.tail = NIL
7  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

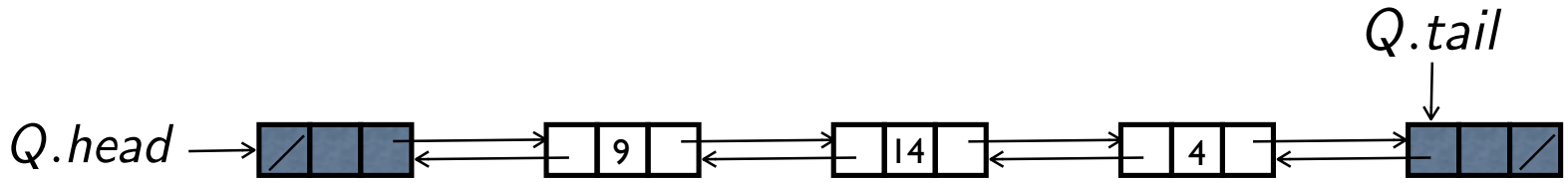
File double

Double ended-queue (deque)

- Généralisation de la pile et de la file
- Collection ordonnée d'objets offrant la possibilité
  - d'insérer un nouvel objet avant le premier ou après le dernier
  - d'extraire le premier ou le dernier objet
- Interface :
  - INSERT-FIRST(Q, x) : ajoute x au début de la file double
  - INSERT-LAST(Q, x) : ajoute x à la fin de la file double
  - REMOVE-FIRST(Q) : extrait l'objet situé en première position
  - REMOVE-LAST(Q) : extrait l'objet situé en dernière position
  - ...
- Application : équilibrage de la charge d'un serveur

Implémentation à l'aide d'une liste doublement liée

- A l'aide d'une liste doublement liée
- Soit la file double Q :
  - $Q.head$ pointe vers un élément sentinelle en début de liste
  - $Q.tail$ pointe vers un élément sentinelle en fin de liste
  - $Q.size$ est la taille courante de la liste



- Les sentinelles ne contiennent pas de données. Elles permettent de simplifier le code (pour un coût en espace constant).

Exercice : implémentation de la file double sans sentinelles, implémentation de la file simple avec sentinelle

Implémentation à l'aide d'une liste doublement liée

```
INSERT-FIRST(Q, x)
1  x.prev = Q.head
2  x.next = Q.head.next
3  Q.head.next.prev = x
4  Q.head.next = x
5  Q.size = Q.size + 1
```

```
INSERT-LAST(Q, x)
1  x.prev = Q.tail.prev
2  x.next = Q.tail
3  Q.tail.prev.next = x
4  Q.head.prev = x
5  Q.size = Q.size + 1
```

```
REMOVE-FIRST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.head.next
4  Q.head.next = Q.head.next.next
5  Q.head.next.prev = Q.head
6  Q.size = Q.size − 1
7  return x
```

```
REMOVE-LAST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.tail.prev
4  Q.tail.prev = Q.tail.prev.prev
5  Q.tail.prev.next = Q.head
6  Q.size = Q.size − 1
7  return x
```

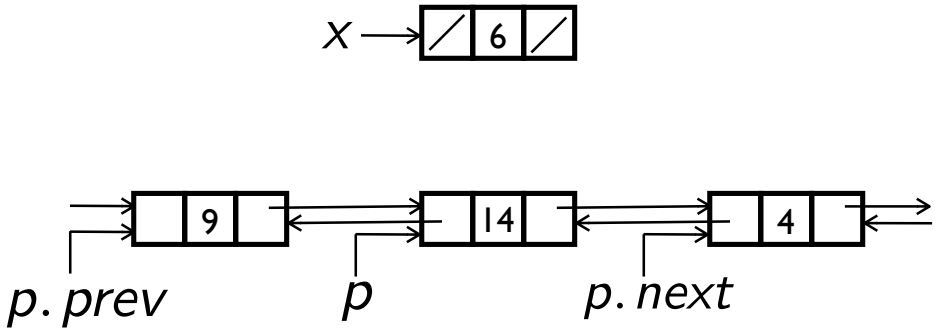
Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

Plan

Liste

- Ensemble dynamique d'objets ordonnés accessibles relativement les uns aux autres, sur base de leur position
- Généralise toutes les structures vues précédemment
- Interface :
  - Les fonctions d'une liste double (insertion et retrait en début et fin de liste)
  - INSERT-BEFORE(L, p, x) : insére x avant p dans la liste
  - INSERT-AFTER(L, p, x) : insère x après p dans la liste
  - REMOVE(L, p) : retire l'élement à la position p
  - REPLACE(L, p, x) : remplace par l'objet x l'objet situé à la position p
  - FIRST(L), LAST(L) : renvoie la première, resp. dernière position dans la liste
  - PREV(L, p), NEXT(L, p) : renvoie la position précédant (resp. suivant) p dans la liste
- Implémentation similaire à la file double, à l'aide d'une liste doublement liée (avec sentinelles)

Implémentation à l'aide d'une liste doublement liée




```
REMOVE(L, p)
1  p.prev.next = p.next
2  p.next.prev = p.prev
3  L.size = L.size − 1
4  return p
```

```
INSERT-BEFORE(L, p, x)
1  x.prev = p.prev
2  x.next = p
3  p.prev.next = x
4  p.prev = x
5  L.size = L.size + 1
```

```
INSERT-AFTER(L, p, x)
1  x.prev = p
2  x.next = p.next
3  p.next.prev = x
4  p.next = x
5  L.size = L.size + 1
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

Vecteur

- Ensemble dynamique d'objets occupant des rangs entiers successifs, permettant la consultation, le remplacement, l'insertion et la suppression d'éléments à des rangs arbitraires
- Interface
  - ELEM-AT-RANK(V, r) retourne l'élément au rang r dans V.
  - REPLACE-AT-RANK(V, r, x) remplace l'élément situé au rang r par x et retourne cet objet.
  - INSERT-AT-RANK(V, r, x) insère l'élément x au rang r, en augmentant le rang des objets suivants.
  - REMOVE-AT-RANK(V, r) extrait l'élément situé au rang r et le retire de r, en diminuant le rang des objets suivants.
  - VECTOR-SIZE(V) renvoie la taille du vecteur.
- Applications : tableau dynamique, gestion des éléments d'un menu,...
- Implémentation : liste liée, tableau extensible...

Implémentation par un tableau extensible

- Les éléments sont stockés dans un tableau extensible $V.A$ de taille initiale $V.c$.
- En cas de dépassement, la capacité du tableau est doublée.
- $V.n$ retient le nombre de composantes.
- Insertion et suppression :

```
INSERT-AT-RANK(V, r, x)
 1  if V.n == V.c
 2        V.c = 2 · V.c
 3        W = "Tableau de taille V.c"
 4        for i = 1 to n
 5              W[i] = V.A[i]
 6        V.A = W
 7  for i = V.n downto r
 8        V.A[i + 1] = V.A[i]
 9  V.A[r] = x
10  V.n = V.n + 1
```

```
REMOVE-AT-RANK(V, r)
1  tmp = V.A[r]
2  for i = r to V.n − 1
3        V.A[i] = V.A[i + 1]
4  V.n = V.n − 1
5  return tmp
```

Complexité en temps

- Insert-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations d'insertion en début de vecteur
  - $\Theta(n)$ pour n opérations d'insertion en fin de vecteur
- Justification :
  - Si la capacité du tableau passe de c_0 à $2^k c_0$ au cours des n opérations, alors le coût des transferts entre tableaux s'élève à

    $$c_0 + 2c_0 + \ldots + 2^{k-1}c_0 = (2^k - 1)c_0.$$

    Puisque $2^{k-1}c_0 < n \leq 2^k c_0$, ce coût est $\Theta(n)$.
  - On dit que le coût amorti par opération est $O(1)$
  - Si on avait élargi le tableau avec un incrément constant m, le coût aurait été

    $$c_0 + (c_0 + m) + (c_0 + 2m) + \ldots + (c_0 + (k-1)m) = kc_0 + \frac{k(k-1)}{2}m.$$

    Puisque $c_0 + (k-1)m < n \leq c_0 + km$, ce coût aurait donc été $\Theta(n^2)$.

Complexité en temps

- Remove-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations de retrait en début de vecteur
  - $\Theta(n)$ pour n opérations de retrait en fin de vecteur

- Remarque : Un tableau circulaire permettrait d'améliorer l'efficacité des opérations d'ajout et de retrait en début de vecteur.

Séquence

- Ensemble dynamique d'objets ordonnées combinant les propriétés d'une liste et d'un vecteur, c'est-à-dire dont les objets sont accessibles tant sur base de leur position absolue que relative
- Interface :
  - Toutes les opérations d'un vecteur
  - Toutes les opérations d'une liste
  - $\textsc{atRank}(S, r)$: retourne la position de l'élément possédant le rang r.
  - $\textsc{rankOf}(S, p)$: retourne le rang de l'élément situé à la position p.

- Implémentation à l'aide d'une liste doublement liée (laissée comme exercice)
  - $\textsc{atRank}$,$\textsc{rankOf}$, $\textsc{Elem-At-Rank}$, $\textsc{Remove-At-Rank}$, $\textsc{Replace-At-Rank}$: $O(n)$, où n est le nombre d'éléments appartenant à la séquence.
  - Autres opérations : $O(1)$.

Plan

Type de données abstrait pour un arbre

- Principe :
  - Des données sont associées aux nœuds d'un arbre
  - Les nœuds sont accessibles les uns par rapport aux autres selon leur position dans l'arbre
- Interface : Pour un arbre T et un nœud n
  - $\textsc{Parent}(T, n)$: renvoie le parent d'un nœud n (signale une erreur si n est la racine)
  - $\textsc{isEmpty}(T)$: renvoie vrai si l'arbre est vide
  - $\textsc{Children}(T, n)$: renvoie une structure de données (ordonnée ou non) contenant les fils du nœud n (exemple : une liste)
  - $\textsc{isRoot}(T, n)$: renvoie vrai si n est la racine de l'arbre
  - $\textsc{isInternal}(T, n)$: renvoie vrai si n est un nœud interne
  - $\textsc{isExternal}(T, n)$: renvoie vrai si n est un nœud externe
  - $\textsc{GetData}(T, n)$: renvoie les données associées au nœud n
  - $\textsc{Root}(T)$: renvoie le nœud racine de l'arbre
  - $\textsc{Size}(T)$: renvoie le nombre de nœuds de l'arbre
  - Pour un arbre binaire (ordonné) :
    - $\textsc{Left}(T, n)$, $\textsc{Right}(T, n)$: renvoie les fils gauche et droit de n
    - $\textsc{hasLeft}(T, n)$, $\textsc{hasRight}(T, n)$: détermine si le nœud n possède un fils respectivement gauche et droit.

Exemples d'opération sur un arbre

- Calcul de la profondeur d'un nœud

```
DEPTH-REC(T, n)
1  if ISROOT(T, n)
2        return 0
3  return 1 + DEPTH-REC(T, PARENT(T, n))
```

- Version itérative

```
DEPTH-ITER(T, n)
1  d = 0
2  while not ISROOT(T, n)
3        d = d + 1
4        n = PARENT(T, n)
5  return d
```

- Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Exemples d'opération sur un arbre

- Calcul de la hauteur de l'arbre

```
HEIGHT(T, n)
1  if ISEXTERNAL(T, n)
2        return 0
3  h = 0
4  for each n2 in CHILDREN(T, n)
5        h = max(h, HEIGHT(T, n2))
6  return h + 1
```

- Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Implémentation d'un arbre binaire

Première solution : numérotation de niveaux

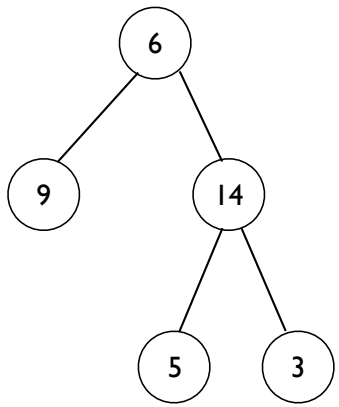
- L'arbre est représenté par un vecteur (ou un tableau)
- Chaque position dans l'arbre est associée à un rang particulier :
  - La racine est en position 1
  - Si un nœud est au rang r, son successeur gauche est au rang $2r$, son successeur droit au rang $2r + 1$
- Si l'arbre binaire n'est pas un arbre binaire complet, le vecteur contiendra des trous (qu'il faudra pouvoir identifier)
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $O(2^n)$ pour un arbre de n nœuds ($\Theta(n)$ pour un arbre binaire complet)

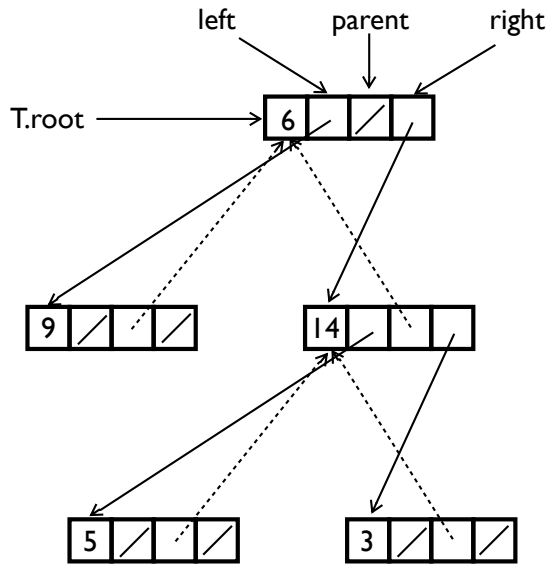
(Exercice : peut-on étendre à des arbres quelconques ?)

Implémentation d'un arbre binaire

Deuxième solution : structure liée

- Principe : on retient pour chaque nœud n de l'arbre :
  - Un champ de données ($n.data$)
  - Un pointeur vers son nœud parent ($n.parent$)
  - Un pointeur vers ses fils gauche et droit ($n.left$ et $n.right$)
  - $T.root$ pointe vers la racine de l'arbre
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $\Theta(n)$ pour n nœuds



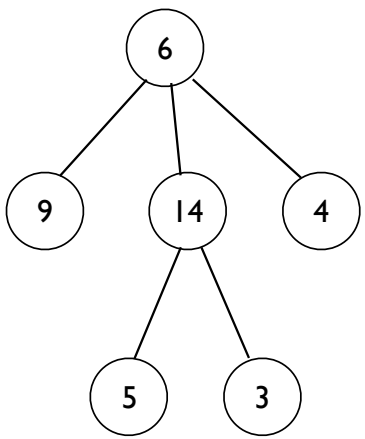


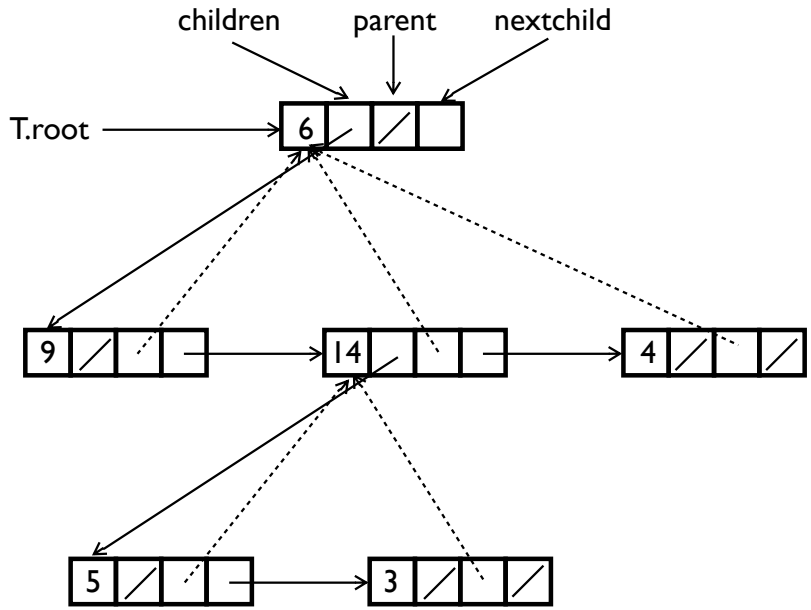
(généralise la notion de liste liée)

Implémentation d'un arbre quelconque

Première solution : structure liée

- Comme pour un arbre binaire
- Mais on remplace $n.left$ et $n.right$ par un pointeur $n.children$ vers un ensemble dynamique.
- Le type d'ensemble dynamique (vecteur, liste, ...) dépendra des opérations devant être effectuées
- Exemple avec une liste liée :

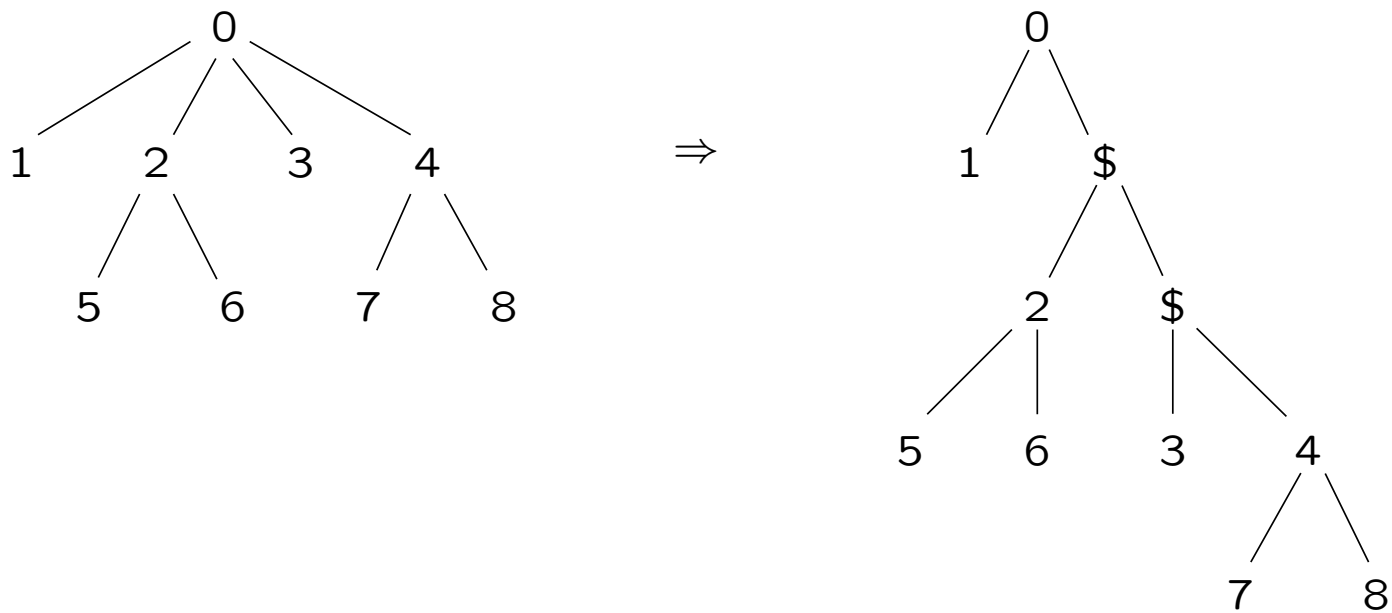




Implémentation d'un arbre quelconque

Deuxième approche : représenter l'arbre quelconque par un arbre binaire

- Si le nœud n possède les fils $n_1, n_2, \ldots, n_p$, avec $p > 2$, alors le sous-arbre issu de n est représenté par une arbre binaire dont :
  - n est la racine
  - Le fils gauche est la racine du sous-arbre issu de n_1
  - Le fils droit est associé à une valeur distinguée "$", et représente un arbre dont la racine possède les fils $n_2, n_3, \ldots, n_p$.
- Illustration :

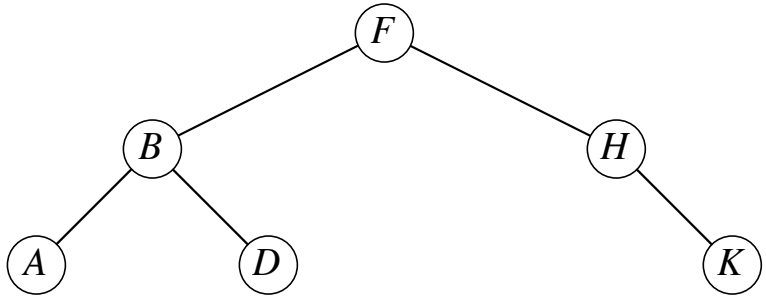


- Complexité des opérations ?

Parcours d'arbres (binaire)

- Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir
- Différents types de parcours :
  - Parcours en profondeur :
    - Infixe (en ordre)
    - Préfixe (en préordre)
    - Suffixe (en postordre)
  - Parcours en largeur

Parcours infixe

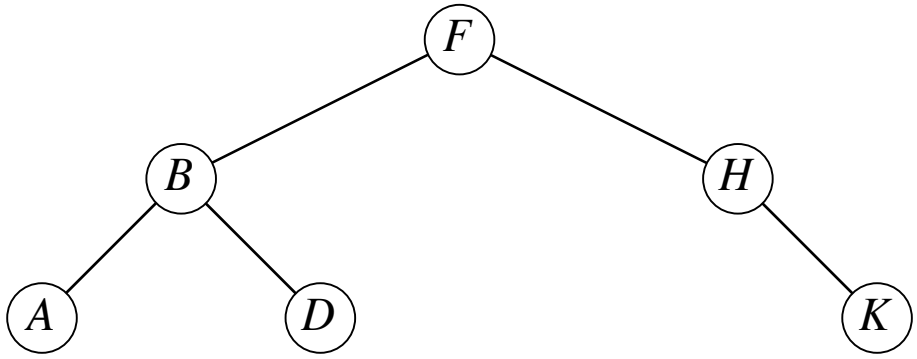


$$\Rightarrow \langle A, B, D, F, H, K \rangle$$

- Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit

```
INORDER-TREE-WALK(T, x)
1  if HASLEFT(T, x)
2        INORDER-TREE-WALK(T, LEFT(x))
3  print GETDATA(T, x)
4  if HASRIGHT(T, x)
5        INORDER-TREE-WALK(T, RIGHT(x))
```

Parcours préfixe

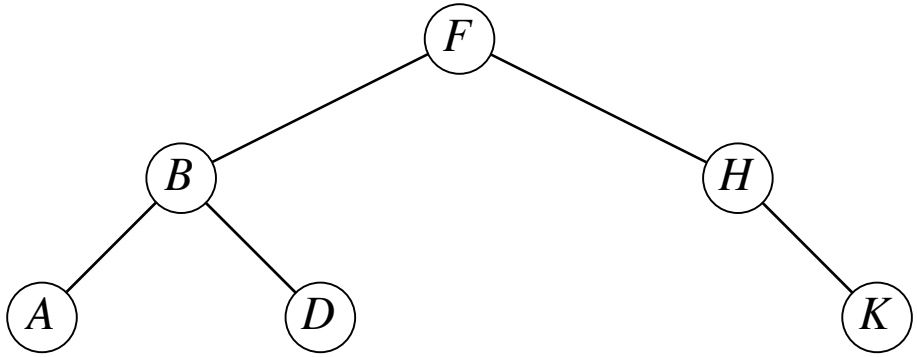


$$\Rightarrow \langle F, B, A, D, H, K \rangle$$

- Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils

```
PREORDER-TREE-WALK(T, x)
1  print GETDATA(T, x)
2  if HASLEFT(T, x)
3       PREORDER-TREE-WALK(T, LEFT(x))
4  if HASRIGHT(T, x)
5       PREORDER-TREE-WALK(T, RIGHT(x))
```

Parcours postfixe



$$\Rightarrow \langle A, D, B, K, H, F \rangle$$

- Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

```
POSTORDER-TREE-WALK(T, x)
1  if HASLEFT(T, x)
2       POSTORDER-TREE-WALK(T, LEFT(x))
3  if HASRIGHT(T, x)
4       POSTORDER-TREE-WALK(T, RIGHT(x))
5  print GETDATA(T, x)
```

Complexité des parcours

Tous les parcours en profondeur sont $\Theta(n)$ en temps

- Soit $T(n)$ le nombre d'opérations pour un arbre avec n nœuds
- On a $T(n) = \Omega(n)$ (on doit au moins parcourir chaque nœud).
- Etant donné la récurrence, on a :

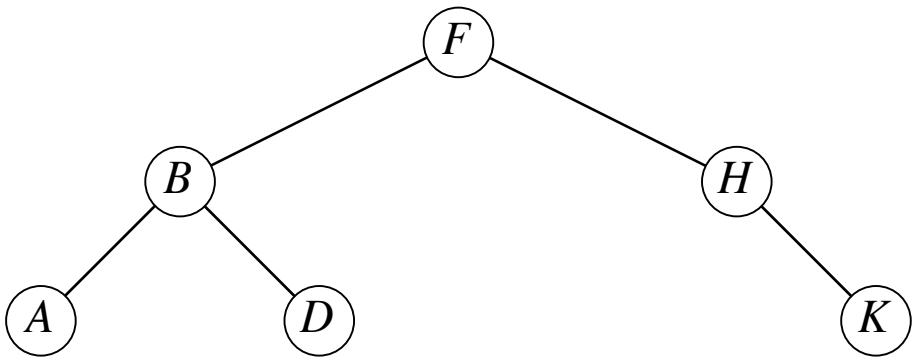
$$T(n) \leq T(n_L) + T(n - n_L - 1) + d$$

où n_L est le nombre de nœuds du sous-arbre à gauche et d une constante

- On peut prouver par induction que $T(n) \leq (c + d)n + c$ où $c = T(0)$.
- $T(n) = \Omega(n)$ et $T(n) = O(n) \Rightarrow T(n) = \Theta(n)$

Parcours en largeur

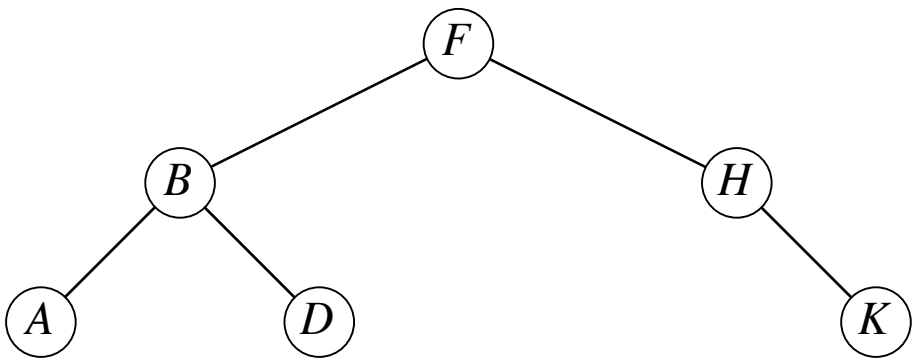
- Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...



$\Rightarrow \langle F, B, H, A, D, K \rangle$

Parcours en largeur

- Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...
- Implémentation à l'aide d'une file en $\Theta(n)$ *(complexité en espace ?)*



$\Rightarrow \langle F, B, H, A, D, K \rangle$

```
BREADTH-TREE-WALK(T)
 1  Q ="Empty queue"
 2  if not ISEMPTY(T)
 3       ENQUEUE(Q, ROOT(T))
 4  while not QUEUE-EMPTY(Q)
 5       y = DEQUEUE(Q)
 6       print GETDATA(T, y)
 7       if HASLEFT(T, y)
 8            ENQUEUE(Q, LEFT(y))
 9       if HASRIGHT(T, y)
10            ENQUEUE(Q, RIGHT(y))
```

(Exercice : Implémenter les parcours en profondeur de manière non récursive)

Plan

File à priorité

- Ensemble dynamique d'objets classés par ordre de priorité
  - Permet d'extraire un objet possédant la plus grande priorité
  - En pratique, on représente les priorités par les clés
  - Suppose un ordre total défini sur les clés
- Interface :
  - $\textsc{Insert}(S, x)$: insère l'élément x dans S.
  - $\textsc{Maximum}(S)$: renvoie l'élément de S avec la plus grande clé.
  - $\textsc{Extract-Max}(S)$: supprime et renvoie l'élément de S avec la plus grande clé.
- Remarques :
  - Extraire l'élément de clé maximale ou minimale sont des problèmes équivalents
  - La file FIFO est une file à priorité où la clé correspond à l'ordre d'arrivée des élements.
- Application : gestion des jobs sur un ordinateur partagé

Implémentations

- Implémentation à l'aide d'un tableau statique
  - Q est un tableau statique de taille fixée $Q.length$.
  - Les éléments de Q sont triés par ordre croissant de clés. $Q.top$ pointe vers le dernier élément.
  - Complexité en temps : extraction en $O(1)$ et insertion en $O(n)$ où n est la taille de la file
  - Complexité en espace : $O(1)$
- Implémentation à l'aide d'une liste liée
  - Q est une liste liée où les éléments sont triés par ordre décroissant de clés
  - Complexité en temps : extraction en $O(1)$ et insertion en $O(n)$ où n est la taille de la file
  - Complexité en espace : $O(n)$
- Peut-on faire mieux ?

Exercice : comment obtenir $O(1)$ pour l'insertion et $O(n)$ pour l'extraction ?

Implémentation à l'aide d'un tas

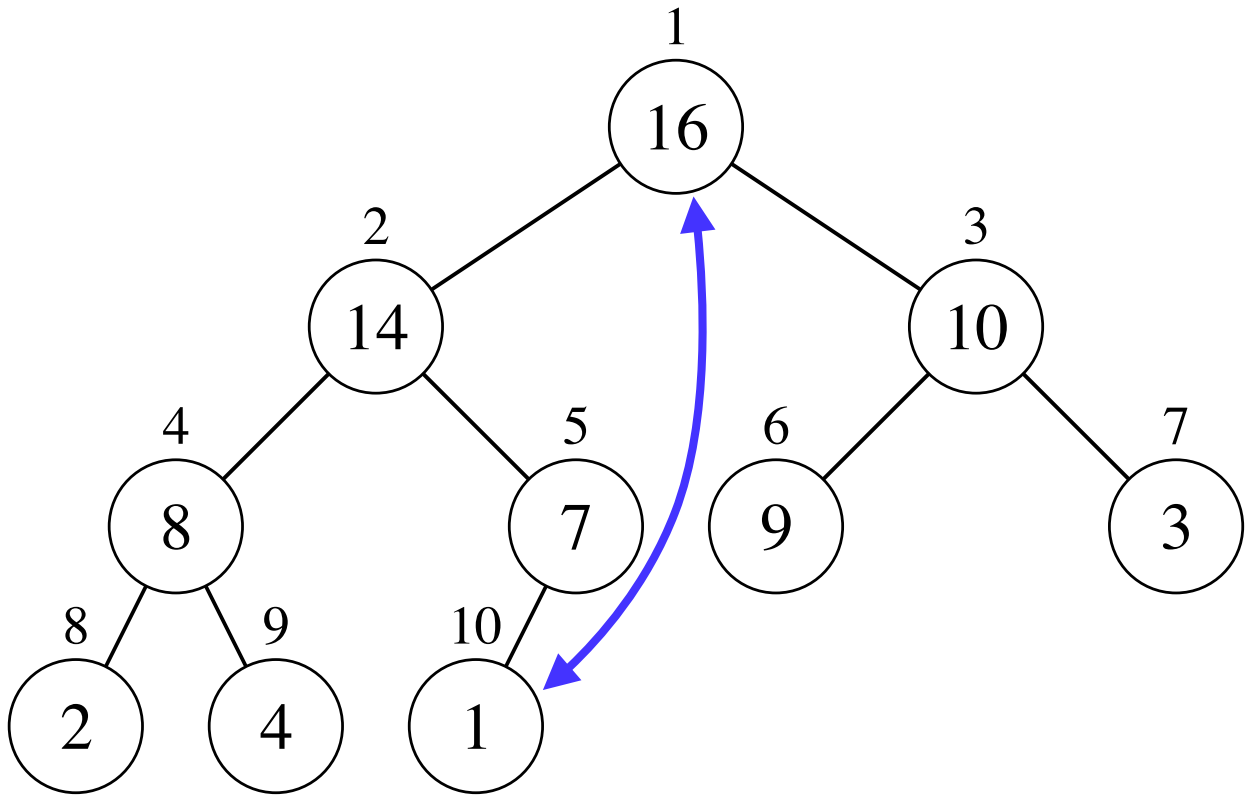
- La file est implémentée à l'aide d'un tas(-max) (voir slide 192)
- Accès et extraction du maximum :

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1) // reconstruit le tas
7  return max
```

- Complexité : $O(1)$ et $O(\log n)$ respectivement (voir chapitre 3)

Implémentation à l'aide d'un tas



Implémentation à l'aide d'un tas : insertion

- INCREASE-KEY(S, x, k) augmente la valeur de la clé de x à k (on suppose que $k \geq$ à la valeur courante de la clé de x).

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        swap(A[i], A[PARENT(i)])
6        i = PARENT(i)
```

- Complexité : $O(\log n)$ (la longueur de la branche de la racine à i étant $O(\log n)$ pour un tas de taille n).

Implémentation à l'aide d'un tas : insertion

- Pour insérer un élément avec une clé *key* :
  - l'insérer à la dernière position sur le tas avec une clé $-\infty$,
  - augmenter sa clé de $-\infty$ à *key* en utilisant la procédure précédente

```
HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

- Complexité : $O(\log n)$.

$\Rightarrow$ Implémentation d'une file à priorité par un tas : $O(\log n)$ pour l'extraction et l'insertion.

Plan

Ensembles disjoints ("Union-Find")

- Structure de données qui maintient à jour une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ d'ensembles dynamiques disjoints.
- Chaque ensemble est identifié par un représentant, qui est un élément quelconque de l'ensemble.

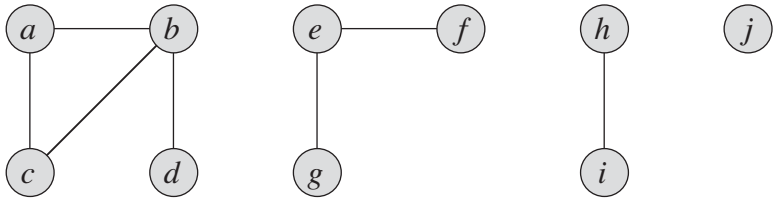
- Interface :
  - $\textsc{Make-Set}(D, x)$: crée un nouvel ensemble dont le seul membre, et donc représentant, est x. x ne doit pas déjà appartenir à un autre ensemble.
  - $\textsc{Union}(D, x, y)$: réunit les ensembles dynamiques qui contiennent x et y. x et y doivent appartenir à des ensembles différents avant l'union.
  - $\textsc{Find}(D, x)$: Renvoie un pointeur sur le représentant de l'ensemble (unique) contenant x. Chaque appel doit renvoyer le même représentant tant que la structure n'est pas modifiée.

Application

Calcul des composantes connexes d'un graphe

```
CONNECTED-COMPONENTS(G)
1  D = "Empty disjoint set"
2  for each vertex v ∈ G.V
3       MAKE-SET(D, v)
4  for each edge (u, v) ∈ G.E
5       if FIND-SET(D, u) ≠ FIND-SET(D, v)
6            UNION(D, u, v)
```

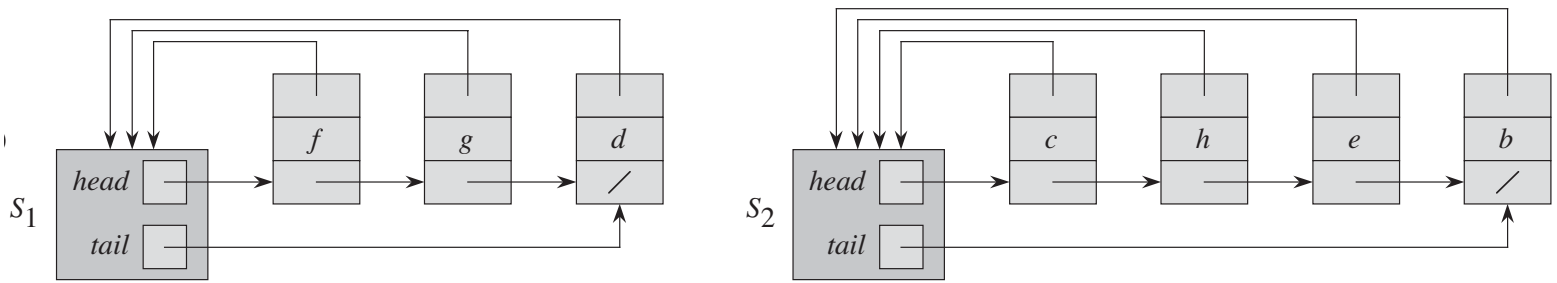
```
SAME-COMPONENT(u, v)
1  if FIND-SET(D, u) == FIND-SET(D, v)
2       return TRUE
3  else return FALSE
```



$$\mathcal{S} = \{\{a, b, c, d\}, \{e, f, g\}, \{h, i\}, \{j\}\}$$

Implémentation par une liste liée

- Chaque ensemble est représenté par une liste liée dont le représentant est le premier objet
- Une sentinelle pour chaque ensemble contient un pointeur vers le début de la liste (*head*) et un pointeur vers la fin (*tail*)
- Chaque objet de la liste contient un élément de l'ensemble, un pointeur sur l'objet contenant l'élément suivant, et un pointeur sur le représentant de l'ensemble.



- MAKE-SET : crée une nouvelle liste contenant seulement x.
- FIND : Suit le pointeur vers la sentinelle et puis renvoie le premier élément.
- $O(1)$ pour les deux fonctions

Union : implémentation naïve

- On fusionne les listes contenant x et y :
  - On concatène la liste de y à la suite de la liste x
  - Nécessite de mettre à jour les pointeurs vers la sentinelle pour tous les éléments de la liste de y
  - Complexité linéaire en fonction de la taille de la liste contenant y
  - $\Theta(n)$ dans le pire cas si n appels à Make-Set ont précédé

- Analyse de complexité pour m opérations :
  - Supposons
    - m opérations (Make-Set, Find, ou Union)
    - n appels à Make-Set (et donc $n-1$ appels à Union au plus)
    - Les n premières opérations sont des appels à Make-Set (pour simplifier l'analyse mais pas nécessaire)
  - Complexité au pire cas en fonction de m et n ?

Pire cas

En ignorant les appels à FIND :

Opération	Nombre d'objets mis à jour
MAKE-SET(x_1)	1
MAKE-SET(x_1)	1
$\vdots$	$\vdots$
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$n-1$

Nombre de mises à jour d'objets : $n + \sum_{i=1}^{n-1} i \Rightarrow \Theta(n^2)$ ($\Rightarrow$ coût amorti par opération : $\Theta(n)$)

En intercalant $m - 2n - 1$ appels à FIND ($O(1)$), on obtient $\Theta(m + n^2)$ pour le pire cas.

Union pondérée

Version plus efficace :

- On maintient la taille de chaque liste dans la sentinelle
- A chaque appel à UNION, on attache la liste la plus courte à la fin de la plus longue
- Complexité linéaire en fonction de la taille de la liste la plus courte
- Toujours $\Theta(n)$ dans le pire cas si n appels à MAKE-SET ont précédé

Analyse pour m opérations (dont n MAKE-SET) :

- Meilleur cas correspond au pire cas de l'approche naïve : $\Theta(m+n)$
- Pire cas ?

Borne supérieure sur le pire cas de l'union pondérée

Montrons que le pire cas est $O(m + n \log n)$

- Soit un objet x, combien de fois son pointeur vers la sentinelle sera-t-il mis à jour au plus ?
  - Quand il est mis à jour, la liste à laquelle il appartient est la plus courte des deux qui sont fusionnées
  - La taille de sa liste est donc au moins doublée
  - Si k mises à jour ont lieu, on doit donc avoir $2^k \leq n \Rightarrow k \leq \log(n)$
- Pour n éléments, les $n - 1$ opérations d'union demandent donc un temps $O(n \log(n))$.
- Pour m opérations au total : $O(m + n \log(n))$

(Exercice : Montrez que le pire correspond à fusionner à chaque étape les deux ensembles les plus petits)

Note : Il existe une implémentation à base d'arbres qui est $O(m \cdot \alpha(n))$ avec $\alpha(n)$ un fonction de croissante très lente en fonction de n ($\alpha(10^{80}) = 4$).

Ce qu'on a vu

- Quelques structures de données classiques et différentes implémentations pour chacune d'elles
  - Liste, files simples, doubles et à priorité
  - Listes, vecteurs, séquences
  - Arbres
  - Ensembles disjoints
- Analyse amortie pour un vecteur

Ce qu'on n'a pas vu

- Notion d'itérateur
- Tas binomial : alternative au tas binaire qui permet la fusion rapide de deux tas
- Evolution dynamique de la taille d'un tas (analyse amortie)
- Implémentation à base d'arbres d'une structure d'ensembles disjoints
- ...