

DIU EIL 2019 / Bloc 5

Algorithmique avancée

Séance 1 – Structures de données abstraites

Définitions liminaires

- Structure de données (**SD**)
= Organisation d'une collection de données en vue de leur exploitation efficace (spatial et temporel)
- Structure de données abstraite (**SDA**)
(*aka Type abstrait de données*)
= Vue « logique » d'une SD ; définition des opérations permises sur la structures (accès, modifications, ...)
- Structure de données concrète (**SDC**)
(*aka Implémentation*)
= Vue « physique » d'une SD ; implémentation du stockage exact des données et des opérations correspondantes
→ Bloc 4

Un exemple : la structure **Sac** (1)

- Considérons une structure de données permettant de collectionner des éléments, et appelons la **un Sac**.
- Un Sac = une collection de données (sans ordre particulier)
 - Exemple : répertorier les dernières destinations de vacances des élèves de NSI : France, Allemagne, Espagne, France, ...
- Que veut-on pouvoir faire avec un Sac de pays ?
 - Y ajouter des éléments
 - En retirer ? (e.g., s'il y a eu une réponse erronée)
 - Compter les pays / les visites

⇒ C'est une SDA : les données, leurs liens et les opérations permises sont caractérisées, mais on ne sait pas (et on ne veut pas savoir) comment c'est fait concrètement.

Structures de données abstraites

Les structures de données abstraites sont classées selon la nature de l'organisation de la collection de données :

- **Séquence** : il y a un *premier* élément et un *dernier* ; chaque élément a un *prédécesseur*¹ et un *successeur*²
- **Association** : les éléments sont repérés par une *clé* ; ils n'ont pas de lien entre eux
- **Hiérarchie** : il y a un (parfois plusieurs) élément racine ; chaque élément dépend d'un *antécédent*³ et a des *descendants*⁴
- **Relation** : chaque élément est en relation directe avec des *voisins*, ou bien a des *prédécesseurs* et des *successeurs*

[1] sauf le premier

[2] sauf le dernier

[3] sauf la/les racine/s

[4] sauf les feuilles

Opérations des SDA

Les opérations usuelles d'une SDA :

- Accéder à un élément
 - Soit ceux qui sont directement repérables par la structure (e.g., premier d'une séquence, ou associé à une clé donnée)
 - Soit à partir d'un élément préalablement repéré (e.g., successeur d'un élément donné)
- Ajouter un élément, en précisant comment il s'intègre dans l'organisation globale de la collection
- Retirer un élément, en précisant comment ceux qui lui étaient liés se réorganisent
- Éventuellement, des opérations plus avancées (e.g., rechercher un élément, trier la collection, fusionner deux collections)

Chaque opération doit être bien spécifiée (entrées, sorties, précondition)

La structure **Sac** (2)

- Quelques opérations sur un Sac :

- Ajouter/Enlever une donnée

ajouter : Sac \times Élément \rightarrow Sac // pré: \emptyset

enlever : Sac \times Élément \rightarrow Sac // pré: élément présent

- Vérifier si une donnée est présente

estPrésent : Sac \times Élément \rightarrow Booléen // pré: \emptyset

- Compter le nombre de données identiques

nbOccurrences : Sac \times Élément \rightarrow Entier // pré: \emptyset

- Compter le nombre de données distinctes

nbDistincts : Sac \rightarrow Entier // pré: \emptyset

- Identifier l'élément le plus présent

dominant : Sac \rightarrow Élément // pré : non vide

Il faut préciser aussi les cas d'ex-æquo !

Note : La notation algébrique utilisée ici peut être remplacée par une notation fonctionnelle plus « algorithmique », e.g., ajouter(s,e)

Efficacité des SDA

- Dans le Bloc 5, on ne s'intéresse pas à l'implémentation des SDA, on considère par défaut que la SD est la plus efficace possible
 - ⇒ le stockage de N données occupe $O(N)$ en espace
 - ⇒ chaque opération est en $O(1)$ en temps et espace
- Parfois, il faudra affiner cette analyse : on sait (*cf. Bloc 4*) que certaines SDA ne peuvent pas implémenter toutes leurs opérations en temps constant ⇒ il faudra choisir les opérations à privilégier

SDA et algorithmes

Les SD à utiliser pour réaliser un traitement algorithmique doivent être choisies

- sur le plan « logique »
 - Comment les données doivent-elles être organisées ? Quels sont leurs liens ?
 - Quelles opérations sont nécessaires pour le traitement ?
⇒ Choix d'une SDA
- et sur le plan « physique »
 - Quelle est l'implémentation la plus efficace, en temps et en espace, pour le traitement considéré ?
⇒ choix d'une SDC implémentant la SDA retenue

La structure **Sac** (3)

- On veut écrire un algorithme qui détermine la destination de vacances préférée des élèves de NSI : le Sac est approprié pour cela !

```
fonction destinationPréférée() → Pays  
  variables Sac destinations, Pays vacances  
  pour chaque élève dans la classe :  
    vacances ← saisie par l'élève  
    ajouter(destinations, vacances)  
  finp  
  retourner dominant(destinations)
```

- Que peut-on dire de cet algorithme ?
 - Il est correct et termine ssi les opérations sur Sac le sont
 - Il prend un temps en $O(\text{nombre d'élèves})$ ssi les opérations sur Sac prennent un temps $O(1)$ (*toutes ?*)

Séance 1 – Structures de données abstraites

1.1 – Structures séquentielles

Liste

- Représentation : ['a', 'b', 'c']
- Opérations :
 - `taille(L)` : nombre d'éléments contenus dans L
 - `L[i]` : accès (en lecture/écriture) au i-ème élément
précondition : i dans $0..taille(L)-1$
 - `insérer(L,e,i)` : insère l'élément e dans L en position i, en incrémentant la position tous les éléments à partir de i
précondition : i dans $0..taille(L)$
 - `supprimer(L,i)` : supprime l'élément en position i, en décrémentant la position de tous les éléments à partir de i+1
précondition : i dans $0..taille(L)-1$

Déjà bien utilisées au blocs 1 et 2 !

Pile

- Représentation : $\succ 'a', 'b', 'c'$
 'a' est le **sommet**, 'c' le **fond** de pile
 - Opérations :
 - $\text{taille}(P)$: nombre d'éléments dans P
 - $\text{sommet}(P)$: accès à l'élément au sommet de P
 précondition : P n'est pas vide
 - $\text{empiler}(P,e)$: ajoute l'élément e au sommet de P
 - $\text{dépiler}(P)$: retire l'élément au sommet de P
 précondition : P n'est pas vide
- ⚡ Remarque : Les signatures algorithmiques *précises* peuvent varier (e.g., **estVide** au lieu de **taille** ; ou encore **dépiler** renvoyant le sommet) ; c'est un choix libre, qui ne change pas la nature de la SDA mais la façon d'écrire les algorithmes l'utilisant

Pile – Exercices (20')

- 1) On veut réaliser un validateur HTML. Écrire un algorithme qui prend en entrée toutes les balises extraites d'une page web et indique si elles sont toutes correctement fermées.
- 2) Écrire l'algorithme qui évalue une expression arithmétique donnée en notation polonaise inverse, e.g., $3\ 5\ +\ 2\ *$ qui se lit $(3+5)*2$ en notation usuelle.

Comme toujours : décontextualisation, spécification puis, seulement, algorithme ... et pourquoi pas complexité et preuves ! ;-)

Éléments de solution : le validateur HTML (1)

- Décontextualisation : les balises constituent des symboles ouvrants ou fermants qui s'associent par paire ; vérifier la bonne imbrication des balises, c'est vérifier un « parenthésage » correct des symboles ouvrants/fermants
- Spécification :
 - Entrée : liste de symboles LS
 - Sortie : un booléen OK
 - Rôle : OK=Vrai ssi le parenthésage dans LS est correct
 - Précondition : les symboles sont connus d'avance
- Principe de l'algorithme :
 - On consulte les symboles un à un dans l'ordre de la liste
 - Si on voit un ouvrant, il faut attendre pour trouver le fermant correspondant
 - Si on voit un fermant, le **dernier** ouvrant *qui n'a pas encore été associé* doit lui correspondre

⇒ Une pile est appropriée pour conserver les ouvrants non encore associés !

Éléments de solution : le validateur HTML (2)

- Hypothèse simplificatrice : on dispose des fonctions
 - ouvrant(S) retourne Vrai ssi le symbole S est un ouvrant
 - associés(S1, S2) retourne Vrai ssi S1 est l'ouvrant associé au fermant S2
- Algorithme :

```
fonction validateur(Liste de symboles LS) → Booléen
  variables Pile nonAssociés ; Booléen OK ; Entier indice
  OK ← Vrai ; indice ← 0
  tant que OK et indice < taille(LS) faire
    si ouvrant(LS[indice]) alors
      empiler(nonAssociés, LS[indice])
    sinon
      si taille(nonAssociés)=0 alors
        OK ← Faux
      sinon
        OK ← associés(sommet(nonAssociés), LS[indice])
        dépiler(nonAssociés)
      finsi
    finsi
    indice ← indice+1
  fintq
  retourner OK et taille(nonAssociés)=0
```

Le simuler pour LS =

- []()
- [({} ())]

Solution en Python : évaluation NPI

```
def évaluationNPI(expression): # tester évaluationNPI([3,2,'+',5,'*'])
    pile = [] # pile vide pour le stockage des opérandes en attente
    for terme in expression:
        if terme in ['+', '*']: # autres opérations négligées
            operande2 = pile.pop()
            operande1 = pile.pop()
            if terme=='+' :
                resultat = operande1 + operande2
            else: # terme=='*'
                resultat = operande1 * operande2
            pile.append(resultat)
        else: # c'est un nombre
            pile.append(terme)
    assert len(pile)==1 # tous les termes ont été "consommés"
    return pile[-1] # le résultat final est au sommet
```

File

- Représentation : $\langle 'a', 'b', 'c' \rangle$
 'a' est le **premier**, 'c' le **dernier**
- Opérations :
 - $\text{taille}(F)$: nombre d'éléments dans F
 - $\text{premier}(F)$: accès à l'élément en premier dans F
 précondition : F n'est pas vide
 - $\text{enfiler}(F, e)$: ajoute l'élément e en dernier dans F
 - $\text{défiler}(F)$: retire l'élément en premier dans F
 précondition : F n'est pas vide

File – Exercices (20')

On veut gérer la file d'attente d'une imprimante.

- 1) Écrire l'algorithme qui annule l'impression d'un document de numéro donné.
- 2) Écrire l'algorithme qui insère un document prioritaire en tête de la file d'attente.

Éléments de solution : annulation d'impression (1)

- Décontextualisation : les travaux d'impression sont repérés par un identifiant qui peut être vu comme un numéro unique \Rightarrow la file d'attente des impressions est une file d'entiers tous distincts.
- Spécification :
 - Entrée : File d'entiers enAttente ; Entier àAnnuler
 - Sortie : la file enAttente modifiée
 - Rôle : retirer l'entier àAnnuler de la file enAttente
 - Précondition : l'entier àAnnuler est présent dans la file enAttente
- Principe de l'algorithme :
 - On extrait tous les entiers un à un de enAttente
 - Si l'entier extrait correspond à àAnnuler, il est jeté
 - Sinon, l'entier est réinjecté en fin de file enAttente

Éléments de solution : annulation d'impression (2)

- Algorithme :

```
fonction annulerImpression(File enAttente, Entier àAnnuler)
  variables Entiers impression, numéro
  pour numéro dans 1..taille(enAttente) faire
    impression ← premier(enAttente)
    défiler(enAttente)
    si impression = àAnnuler alors
      // rien à faire
    sinon
      enfiler(enAttente, impression)
    finsi
  finp
```

- Que peut-on dire de cet algorithme ?
 - Il termine car les opérations sur file terminent et utilise un pour
 - Il est correct car tous les travaux sont ré-enfilés, dans l'ordre initial, sauf celui à retirer
 - Il prend un temps $O(\text{nombre d'impressions en attente})$ car les opérations sur file sont en temps $O(1)$

Solution en Python : ajout impression prioritaire

```
def impressionPrioritaire(enAttente,impression):  
    enAttente.append(impression) # ajout en queue  
    for i in range(len(enAttente)-1): # défilement ...  
        imp = enAttente.pop(0) # ... par retrait en tête ...  
        enAttente.append(imp) # ... et ajout en queue
```

Tester avec :

```
ListeDAttente = [2,3,4]  
  
impressionPrioritaire(ListeDAttente,1)  
  
print(ListeDAttente)
```

Séance 1 – Structures de données abstraites

1.2 – Structures associatives

Dictionnaire

- Représentation : $\{ ('a',8), ('b',3), ('c',8) \}$
Ensemble de couples (**clé**,**valeur**) ; clés **uniques**
- Opérations :
 - $\text{taille}(D)$: nombre de couples dans D
 - $\text{trousseau}(D)$: liste des clés dans D
Utile pour parcourir une structure de dictionnaire !
 - $\text{estDans}(D,c)$: vérifie s'il existe une clé c associée dans D
 - $D[c]$: accès (en lecture/écriture) à la valeur associée à c
Précondition : c doit exister dans D
 - $\text{associer}(D,c,v)$: associe la clé c à la valeur v dans D
Remplace la valeur associée à c si elle était déjà dans D ; ajoute (c,v) sinon
 - $\text{dissocier}(D,c)$: retire le couple de clé c dans D
Précondition : c doit exister dans D

Déjà bien utilisé au bloc 1, et on en reparle à la prochaine séance

Séance 1 – Structures de données abstraites

1.3 – Structures hiérarchiques

Arbres – Terminologie

Donnez une définition et un exemple pour chacun des termes :

Arbre

Sommet

Arête

Arbre enraciné

Nœud

Arc

Père

Fils

Étiquette

Arbre binaire

Arbre ordonné

Arbre complet

Racine

Feuille

Nœud interne

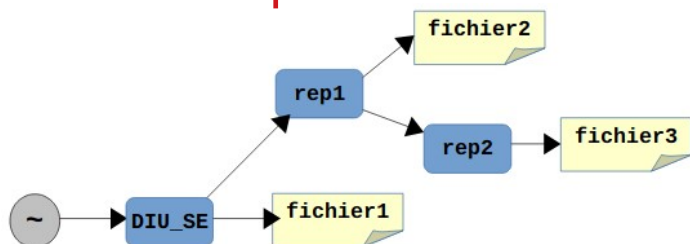
Hauteur

Profondeur

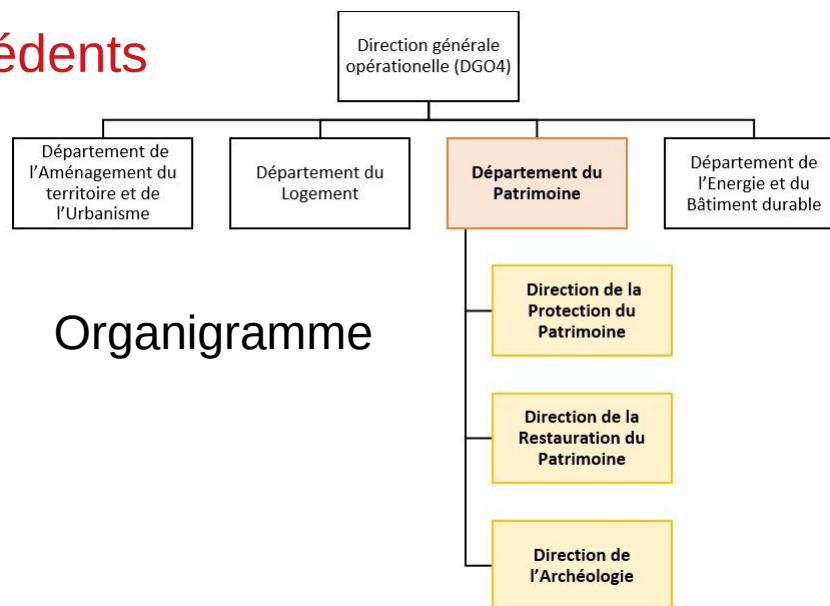
Sous-arbre

Exemples d'arbres

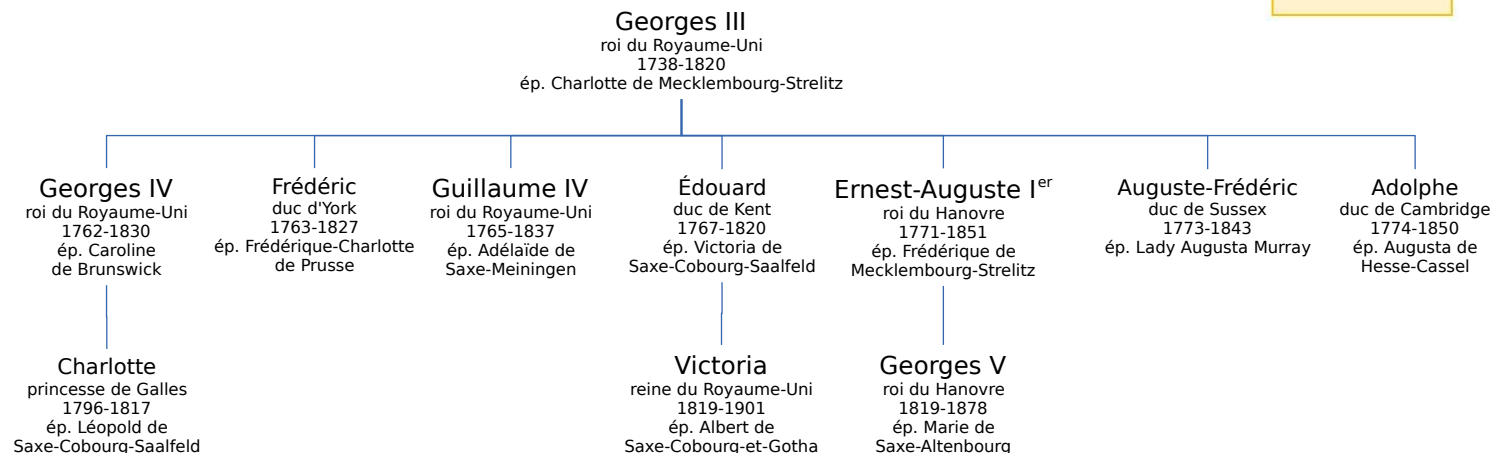
Illustrer quelques-uns des termes précédents sur ces exemples



Arborescence de dossiers/fichiers



Organigramme



Arbre généalogique (descendants) *Quid des ascendants ?
Et des descendantes ?*

Situations où les arbres sont utiles

- Représenter des relations de composition
 - Recette de cuisine (tarte = pâte + appareil + fruits, pâte = farine + beurre + sucre, ...)
 - Image bitmap (image = 4 quarts d'image, un quart d'image = 4 seizièmes d'images, ... jusqu'au pixel)
 - Une expression arithmétique, opérateurs dans les nœuds internes, opérandes dans les feuilles
- Représenter une classification
 - Classification décimale de Dewey (1876) pour les fonds bibliographiques et documentaires
 - Arbre phylogénétique pour la classification des espèces
 - Arbre des rencontres pour un tournoi à élimination directe

• ...

Arbres binaires

- Seuls les arbres binaires (enracinés, ordonnés) sont au programme de NSI *(et c'est bien suffisant ! ;-)*
- Représentation : graphiquement, racine en haut, feuilles en bas, fils gauche à gauche, fils droit à droite
- Opérations : il existe de nombreux jeux d'opérations selon la finalité, mais en général on retrouve les suivantes
 - EstVide(A) : retourne Vrai ssi A ne contient aucun nœud
 - racine(A) : accès au nœud racine de A
pré : A n'est pas vide
 - gauche(A) : accès au sous-arbre gauche de A
pré : A n'est pas vide
 - droit(A) : accès au sous-arbre droit de A
pré : A n'est pas vide

Arbres binaires – Propriétés

Quelques propriétés *(à illustrer ou démontrer)*:

- Un arbre binaire à N nœuds a
 - une hauteur comprise entre $\log_2(N+1)-1$ et $N-1$;
 - un nombre de feuilles compris entre 1 et $(N+1)/2$
- Un arbre binaire complet de hauteur H a
 - un nombre de nœuds compris entre 2^H et $2^{H+1}-1$;
 - un nombre de feuilles compris entre 2^{H-1} et 2^H .

Algorithmes sur arbres binaires

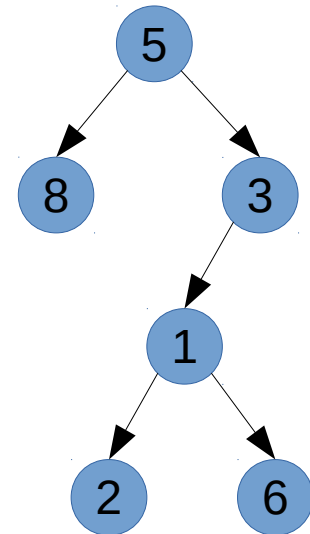
- La plupart des traitements sur des arbres enracinés sont naturellement **récur­sifs** : on traite un nœud courant et on demande à traiter les nœuds fils.
- On distingue trois ordres particuliers pour le traitement exhaustif des nœuds d'un arbre binaire :
 - **Préfixe** = le nœud courant est traité, puis son sous-arbre gauche et son sous-arbre droit.
 - **Infixe** (ou symétrique)= le nœud courant est traité entre son sous-arbre gauche et son sous-arbre droit.
 - **Suffixe** (ou postfixe) = le nœud courant est traité après son sous-arbre gauche et son sous-arbre droit.

Un premier algorithme : afficher les étiquettes

- On veut afficher toutes les étiquettes des nœuds d'une arbre binaire A :

```
fonction afficher_rec(ArbreBinaire A)  
  si non estVide(A) alors  
    afficher à l'écran racine(A)  
    afficher_rec(gauche(A))  
    afficher_rec(droit(A))  
  finsi
```

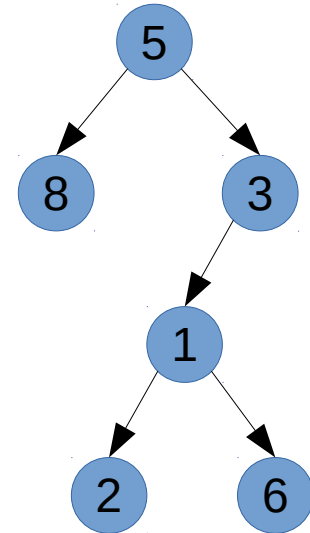
- Appliquer cet algorithme sur l'arbre ci-contre.
- De quel ordre de traitement s'agit-il ?
- Que modifier dans ces algorithmes pour obtenir les autres ordres de traitement ?



Un second algorithme : afficher les étiquettes (?!)

- Cassons tout de suite la « règle » : voici un algorithme itératif pour l'affichage

```
fonction afficher(ArbreBinaire A)
  variable Pile P ; ArbreBinaire a
  empiler(P,A)
  tant que taille(P)>0 faire
    a ← sommet(P) ; dépiler(P)
    si non estVide(a) alors
      afficher à l'écran racine(a)
      empiler(P,droit(a))
      empiler(P,gauche(a))
    finsi
  fintq
```



- Appliquer cet algorithme sur l'arbre ci-contre.
- De quel ordre de traitement s'agit-il ?
- Que modifier dans cet algorithme pour obtenir les autres ordres?

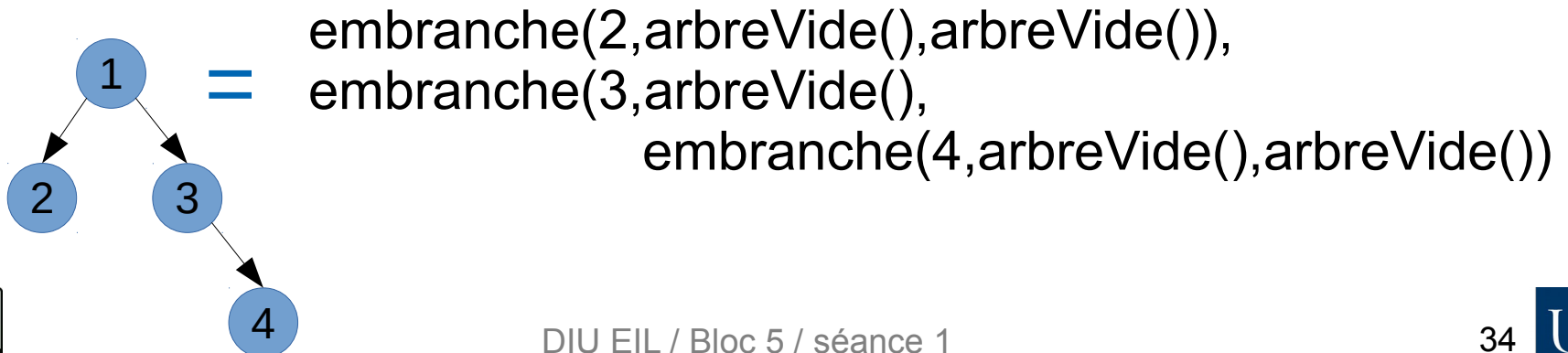
Arbres binaires – Exercice (20')

Écrire un algorithme qui reconstruit un arbre binaire à partir de deux listes : L_p la liste des ses étiquettes par ordre préfixe, et L_i celle de ses étiquettes par ordre infixe

Pour construire un nouvel arbre, on considérera les deux opérations suivantes :

- `arbreVide()` : retourne un nouvel arbre vide
- `embranche(R,G,D)` : retourne un nouvel arbre dont la racine a l'étiquette R et le sous-arbre gauche (resp. droit) est G (resp. D)

Exemple : `embranche(1,`



Éléments de solution : reconstruire un AB (1)

- Idées :
 - Dans la liste préfixe L_p , le premier élément est toujours la racine R
 - Dans la liste infixe L_i , tous les éléments avant (resp. après) R sont dans le sous-arbre gauche G (resp. droit D)

⇒ en décomposant récursivement les listes, on identifie la structure de l'arbre
- Exemple :

$$L_p = [\textcircled{1}, \boxed{2}, \boxed{3, 4}]$$

G R D

$$L_i = [\boxed{2}, \textcircled{1}, \boxed{3, 4}]$$

Éléments de solution : reconstruire un AB (1)

- Algorithme :

```
fonction reconstruireAB(Li,Lp) → ArbreBinaire
  variables ArbreBinaires G,D ; Entiers r,n
  n ← taille(Li) # = taille(Lp) = nb nœuds
  si n=0 alors
    retourner arbreVide() # aucun nœud => arbre vide
  sinon
    r ← recherche(Li,Lp[0]) # indice de la racine dans Li
    G ← reconstruireAB(Li[0..r-1],Lp[1..r]) # gauche
    D ← reconstruireAB(Li[r+1..n-1],Lp[r+1..n-1]) # droit
    retourner embranche(Lp[0],G,D)
  finsi
```

Notes :

- $L[i..j]$ représente la sous-liste de L entre i et j, vide si $i > j$
- Il s'agit d'un algorithme de la famille « diviser-pour-régner »

Conclusion

- Les structures de données en algorithmique s'envisagent à différents niveaux (abstrait, concret)
- Ce sont leurs opérations qui guident le choix de la structure pour un traitement algorithmique donné
- Elles simplifient la réalisation de traitements algorithmiques complexes ...
- ... mais sont aussi une source de traitements algorithmiques complexes !
- Dans les prochaines séances, nous allons développer l'algorithmique des arbres binaires et des graphes