centralelille

AUDACE • EXIGENCE • RESPECT

Algorithmique Avancée & Programmation

Un grand merci à Christian Vercauter, rédacteur des documents dont ce poly s'inspire

Séance 2 : types abstraits de données

Test 2

- QCM, sur moodle
- 15 minutes

Cadrage Séance 2 : Structures, récursivité

- Test papier : pointeurs
  - Surtout syntaxe
- Structures
  - struct
  - typedef
  - Notations pointées, fléchées
- Structures auto-référentes
  - Listes chaînées
- Types abstraits de données
  - TAD pile et file avec les listes
- Programmation modulaire
  - Compilation séparée
  - Librairies

Après cette séance, vous devez savoir :

- Concevoir des structures adaptées à un problème donné
- Comprendre et réutiliser des structures existantes
- Concevoir, dessiner, implémenter et mettre au point des algorithmes évolués utilisant des types abstraits de données
- Produire du code réutilisable

centralelille

AUDACE • EXIGENCE • RESPECT

Rappels
Cf. Capsule 5
Cf. TEA S1

Structures

Allocation dynamique de mémoire

Pointeurs sur structure

Structures

- Utilisation du mot-clé struct
- Permet de regrouper des types hétérogènes
  - Chaque champ porte un nom et un type
- Possibilité de les initialiser comme les tableaux
- Possibilité de les affecter comme des variables !
- [Bonne pratique] Définir d'abord un type pour les nouvelles structures
- Utiliser des constantes symboliques comme valeurs pour les champs

```
typedef struct {
    int x; // abscisse
    int y; // ordonnée
    char forme; // CAN, BOT
    char etat ; // EMPTY, FULL
} T_objet;
T_objet o1 = {-10,25,CAN, EMPTY};
T_objet o2;
o2 = o1; // copie autorisée !
o1.x = 3;
```

Passage de paramètres de type structures ou tableaux

- Les tableaux sont passés aux fonctions par référence !
  - Ce que l'on passe, c'est l'adresse du tableau
  - On pourra donc modifier le tableau depuis la fonction !
- Les structures sont passées par valeur
  - On manipule une copie de la structure au sein de la fonction
  - On ne pourra pas modifier la structure ayant servi lors de l'appel depuis la fonction

Allocation dynamique de mémoire
Cf. man malloc

- malloc(), calloc(), realloc() : Pour initialiser les pointeurs
  - Renvoie l'adresse de début de la zone mémoire (void *)
- void *malloc(size_t size);
  - zone non initialisée
- void *calloc(size_t nmemb, size_t size);
  - zone initialisée à 0
- void *realloc(void *ptr, size_t size);
  - agrandissement d'une zone préalablement allouée
- void free(void *ptr);
  - libère la zone de mémoire pointée

Allocation dynamique de mémoire : bonnes pratiques

- Il est d'usage de "*caster*" l'adresse de retour de la fonction malloc
  - int * p = (int *) malloc(MAXTAB * sizeof(int))
  - int * p = (int *) calloc(MAXTAB, sizeof(int))
- Si ces fonctions échouent, elles renvoient NULL
  - Elles peuvent donc échouer !
  - Cf. Macro CHECK_IF, capsule 6 (Cf. include/check.h)
- Penser à libérer explicitement la mémoire avec lorsqu'elle n'est plus utilisée :
  - free(p);
  - p = NULL;

Pointeur sur structure

- Si l'on dispose d'un pointeur sur une structure, l'accès à ses champs se fait en utilisant la notation fléchée

```
typedef struct {
    int champ1;
} T_Structure;
T_Structure * pS1;
pS1->champ1 = 3;
(*pS1).champ1 = 4; // équivalent mais lourd...
```

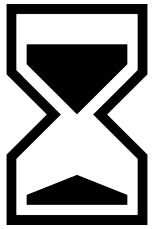
Programmation défensive

- void assert (int test)
  - Cf. fichier d'entête assert.h
- si test == 0, le programme s'arrête et le message suivant est envoyé sur stderr :
  - *Assertion failed: test, file NomdeFichier, line NoLigne*

centralelille

AUDACE • EXIGENCE • RESPECT

3h30



TAD : Type Abstrait de Données

TAD : Définition

- Spécification d'une structure de données à partir de l'ensemble des opérations qu'elle peut effectuer, et de ses propriétés
  - Indépendante des choix d'implémentation
- Définit le comportement mais pas les détails internes d'organisation des données ni les détails de réalisation des traitements

TAD : Exemple

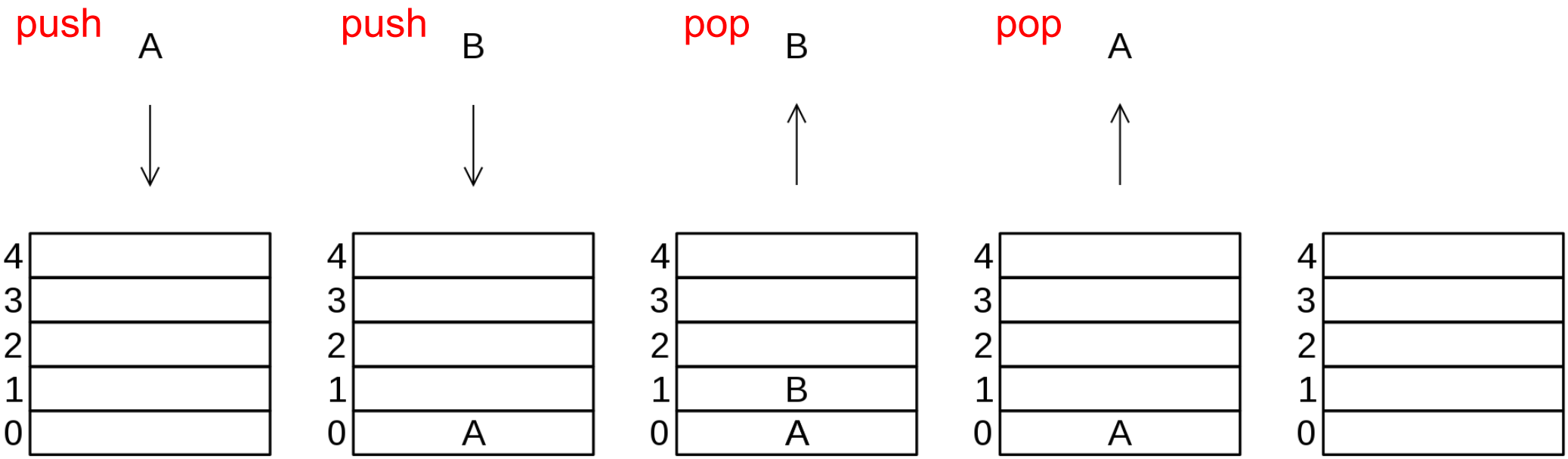
- TAD “fichier”
  - Doit permettre les opérations Ouvrir, Fermer, Lire la donnée courante et avancer, …
- Implémentation en C : type « opaque » FILE :
  - L’utilisateur n’a pas à connaître la nature précise de FILE
  - FILE *fopen(const char *fname, const char *fmode);
  - int fclose (FILE * flux);
  - int getc(FILE *flux);

TAD Pile

- Structure de données linéaire à accès séquentiel, basée sur le principe « Dernier arrivé, premier sorti » ou LIFO (Last In, First Out).
  - Une autre structure classique : File : FIFO
- Un seul élément est directement accessible : l'élément en sommet de pile (s'il existe)

Concrètement (Implémentation contiguë)

- https://fr.wikipedia.org/wiki/Pile_(informatique)



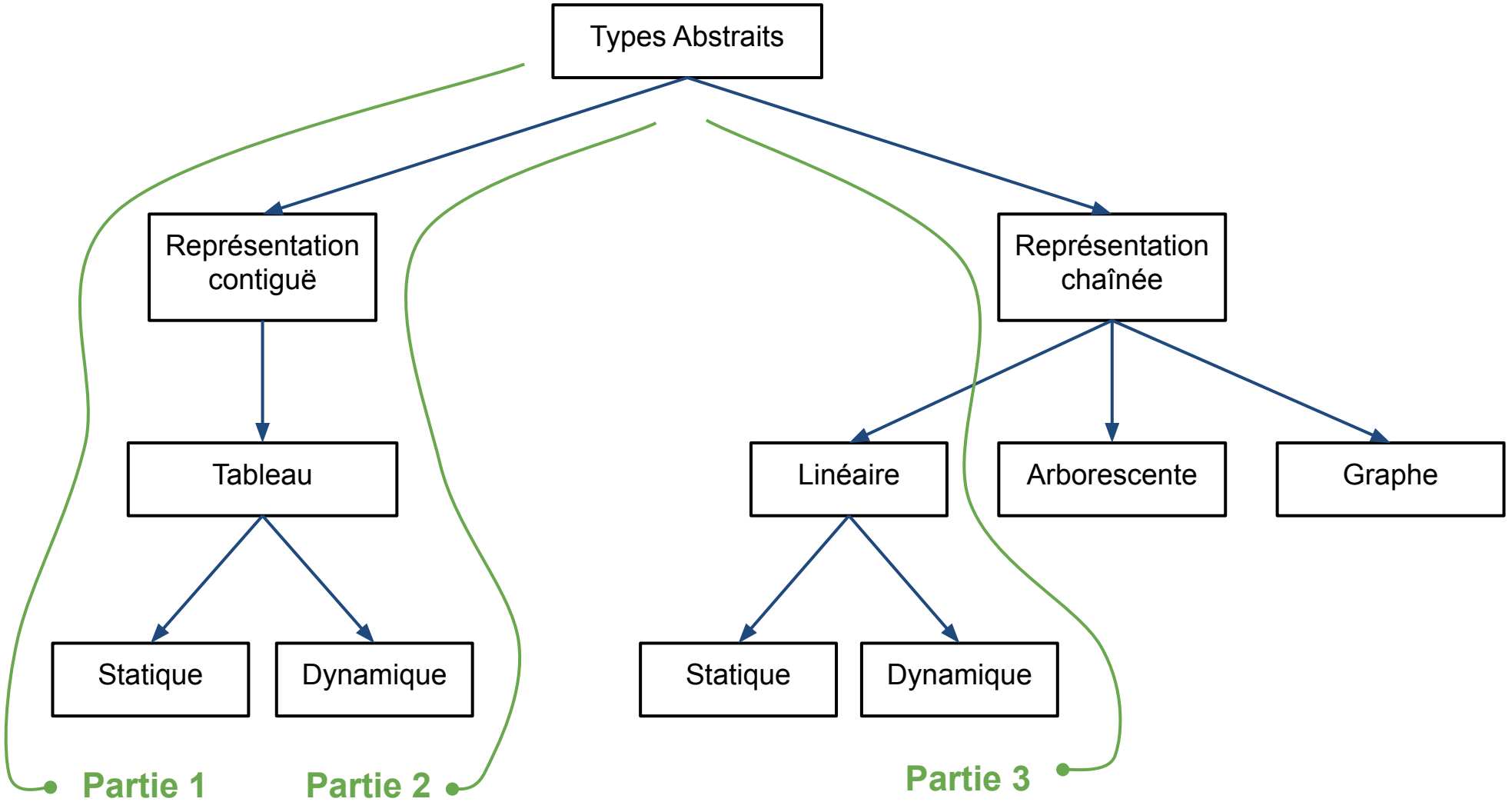
TAD Pile (stack)

Type	*Nom*	*Opération*	*Signification*
création	newStack	(rien) -> T_stack	Opération sans argument permettant de créer une pile vide
accès	isEmpty	T_stack -> booléen	Cette opération permet de tester si la pile est vide.
modification	push	T_stack x T -> T_stack	Opération consistant à empiler un élément T sur la pile modifiant ainsi son état
modification	pop	T_stack -> T x T_stack	Cette opération extrait de la pile, l'élément au sommet, modifiant ainsi son état
accès	top	T_stack -> T	Cette opération permet d'accéder à l' élément en sommet de pile

Utilisation de piles

- Langages de programmation : pour **transmettre les paramètres et stocker les variables locales des fonctions**
- Processeurs : pile pour la sauvegarde de l'adresse de reprise lors d'un appel de sous-programme
- Applications logicielles : enregistrement des actions de l'utilisateur pour pouvoir les annuler
- Algorithmique :
  - Dérécursivation d'algorithmes
  - Analyse d'expressions arithmétiques dans les compilateurs et interpréteurs

Implémentations du TAD pile

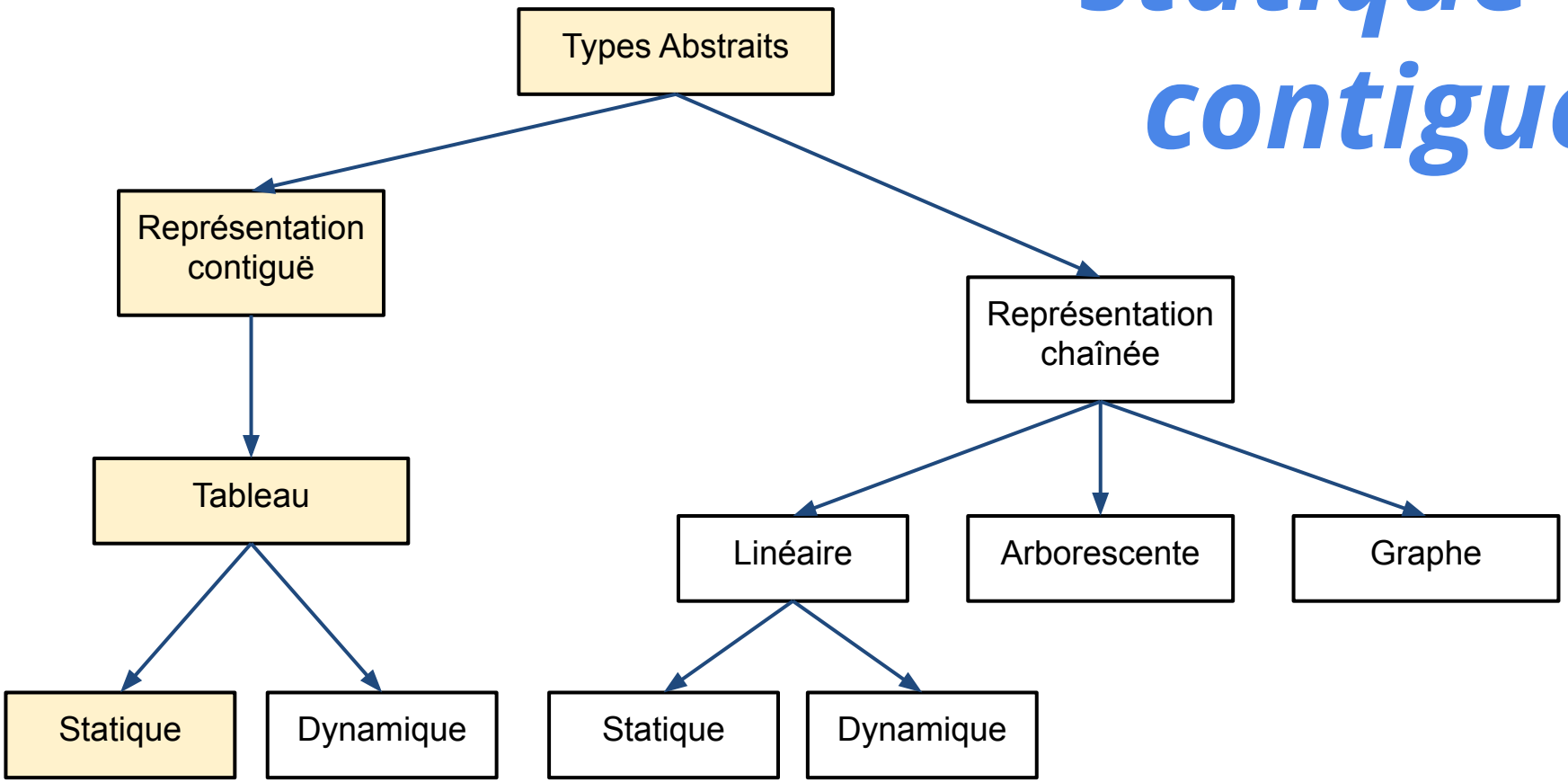


centralelille

AUDACE • EXIGENCE • RESPECT

3h20

TAD pile : Implémentation statique et contiguë



TAD pile : Implémentation statique et contiguë

- Type Abstrait : on y stocke des T_elt génériques
- data : tableau où sont empilés les éléments
- sp : indice de la case où sera rangé le prochain élément
  - “Stack Pointer”

```
#define STACK_NBMAX_ELT 10

typedef struct {
    int sp;
    T_elt data[STACK_NBMAX_ELT];
} T_stack;
```

Rappel

- L'affectation de structures entre elles est permise
  - Contrairement aux tableaux
- Du coup, une fonction peut renvoyer une structure allouée sur la pile
- Elle sera recopiée dans l'emplacement mémoire recevant le résultat de la fonction

```
T_maStructure foo_returning_struct() {
    T_maStructure aux;
    // var locale, allouée sur la pile !
    …
    return aux;
}

main() {
    T_maStructure s;
    // var locale de main()
    s = foo_returning_struct();
    // On recopie dans s tous les champs de la structure renvoyée !
}
```

Rappel

- Les structures sont passées aux fonctions par valeur
  - On manipule une copie de la structure au sein de la fonction
  - On ne pourra pas modifier la structure ayant servi lors de l'appel depuis la fonction
- Solution : utiliser des pointeurs sur structures comme arguments des fonctions !
  - Passage par référence
  - Utilisation de la notation fléchée :

    (*pS).champ == pS->champ

Exemple : ex1 (1)

3h15

15 min

- Définir un T_elt comme un int
- Créer et initialiser une première pile de 3 T_elt
- Produire une fonction d'affichage de la pile et la tester :
  void showStack (const T_stack * p)
  - Afficher en commençant par le sommet de la pile
  - Tester aussi en affichant une pile vide
- Préparer votre code à changer d'implémentation à l'aide de constantes symboliques
  - IMPLEMENTATION_STATIC_CONTIGUOUS
  - IMPLEMENTATION_DYNAMIC_CONTIGUOUS
  - IMPLEMENTATION_DYNAMIC_LINKED

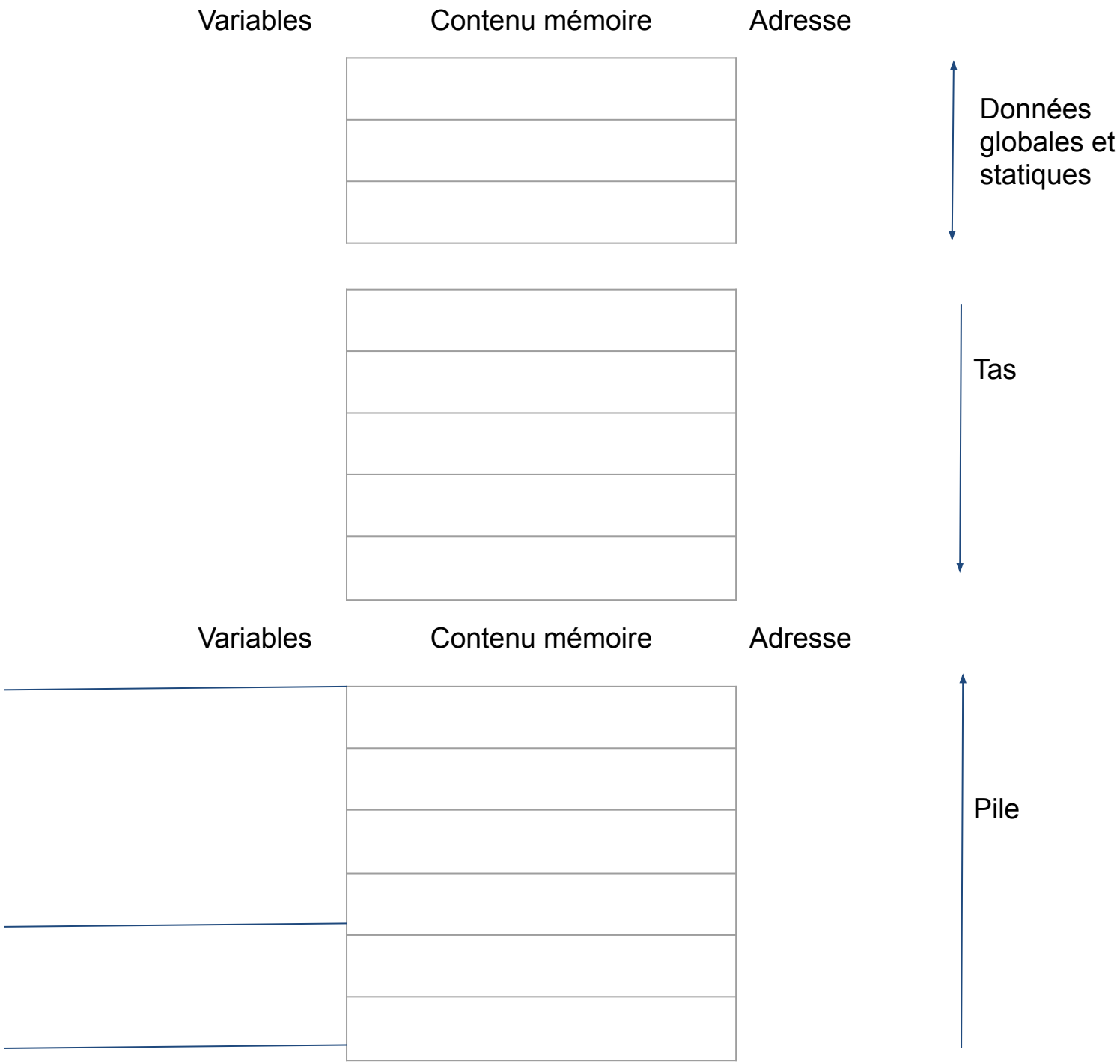
Exercice corrigé : ex1 (2)

3h00

15 min

- T_stack newStack(void)
  - Renvoie une nouvelle pile vide
  - Que deviendra-t-elle ?
- La forme T_stack * newStack(void) est-elle nécessaire ?
  - Est-elle meilleure ?
- NB : Dans cette implémentation, nous n'utilisons pas d'allocation dynamique de mémoire

- void emptyStack (T_stack *p)
  - Vide une pile
- T_stack exampleStack (int n)
  - Renvoie une pile initialisée avec n éléments
  - Quelle hypothèse sur le paramètre n doit être vérifiée ?


Variables
Contenu mémoire
Adresse
Données globales et statiques
Tas
Variables
Contenu mémoire
Adresse
Pile

2h45

15 min

Module de gestion de pile

- Doit être prêt à traiter tous types d'objets T_elt
- Passages par référence
  - meilleures performances
- Programmation défensive
  - Cf. assert()

Focus : Bonnes pratiques de nommage

- Termes en anglais
- ‘new’ pour une fonction d’allocation de mémoire
- ‘gen’ pour une fonction générant une valeur automatiquement
  - ‘init” ou ‘example’ pour une fonction initialisant la structure ou créant un exemple manuellement
- ‘show’ pour une fonction qui affiche
- ‘toString’ pour une fonction qui renvoie une chaîne
  - Mais n’affiche rien !
- ‘get’ pour une fonction récupérant un champ d’une structure
- ‘set’ pour une fonction modifiant un champ d’une structure
- ‘p_’ pour une variable de type pointeur

Focus : Bonnes pratiques d'architecture

- Répartir le travail dans plusieurs fichiers
  - réutilisables dans d'autres projets
- Prévenir les inclusions multiples d'un même fichier à l'aide de directives de compilation conditionnelle
- Activer ou désactiver des morceaux de code par des constantes symboliques
- TP2 : on répartit les implémentations de T_stack en trois fichiers
  - Un seul de ces fichiers contiendra du code actif à la fois
  - Pas forcément classique, mais permet de s'y retrouver (choix pédagogique)
  - Le fichier makefile_sources indique quels sont les fichiers sources à compiler

Exemple : ex2
elt.h & elt.c

- Développer les fichiers elt.h et elt.c
- Type T_elt
- char * toString(T_elt)
  - Renvoie une chaîne de caractère représentant un T_elt pour l'afficher
  - Penser à utiliser sprintf(buffer, format, ...)
- T_elt genElt(void)
  - Génère un nouveau T_elt différent du précédent (utiliser une variable statique)
- Utiliser des constantes symboliques pour sélectionner le type de T_elt parmi ELT_CHAR, ELT_INT ou ELT_STRING
  - ELT_INT par défaut
- Tester les 3 types possibles

Exercice corrigé : ex3 (1)
stack_cs.h & stack_cs.c

_cs.* : contiguë statique

2h30

20 min

- Développer les fichiers stack_cs.h et stack_cs.c
- void emptyStack (T_stack *p)
- T_stack newStack(void)

} déjà écrit !

- void showStack (const T_stack * p)
- T_stack exampleStack(int n)
- T_elt pop(T_stack *p)
- T_elt top(const T_stack *p)
- void push(T_elt e, T_stack *p)
- int isEmpty (const T_stack * p)

⇒ à adapter : utilisation de toString()

⇒ à adapter : utilisation de push() et genElt()

Utiliser de la programmation défensive

Exercice corrigé : ex3 (2)

main.c

2h10

10 min

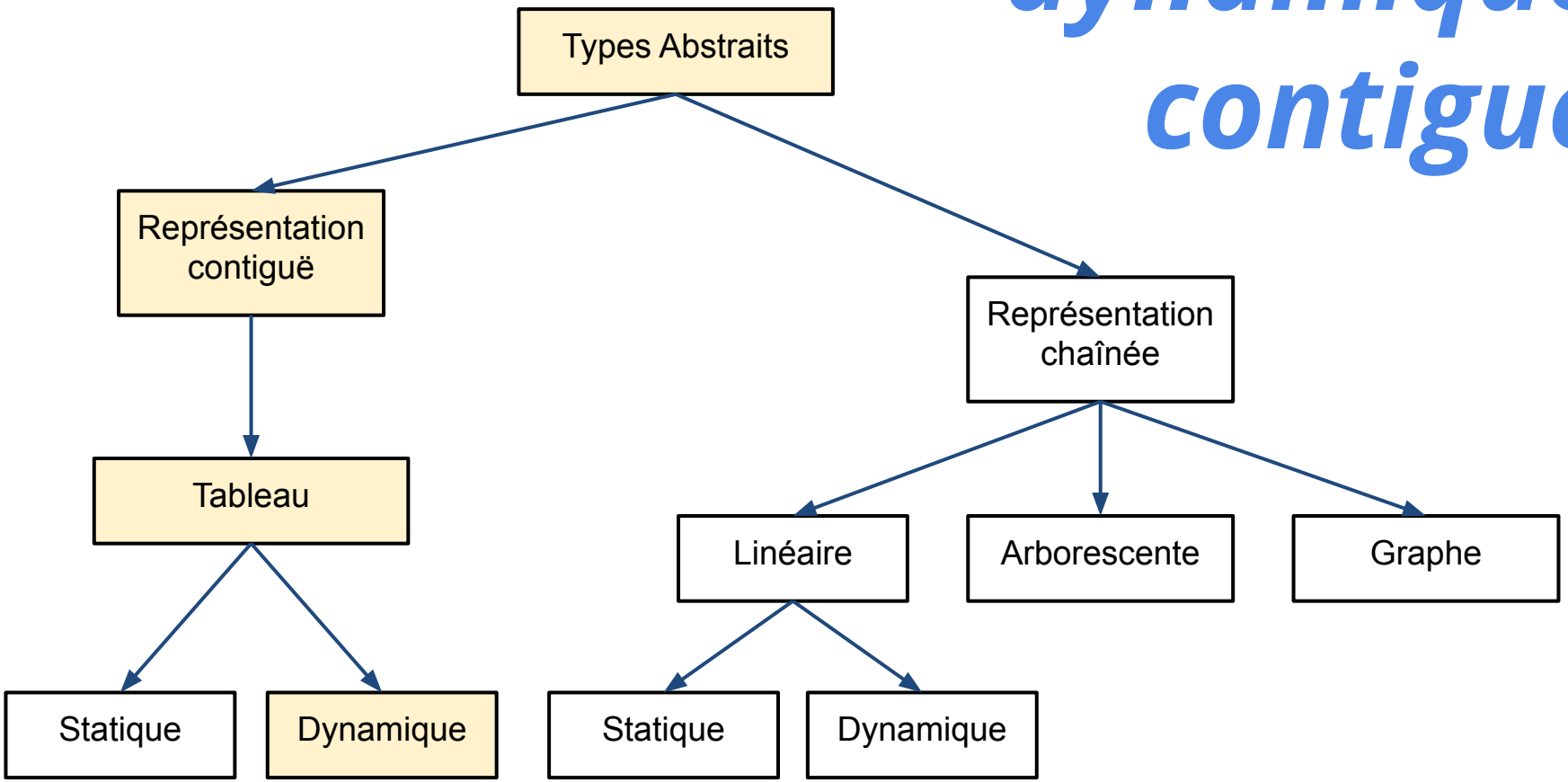
- Tester votre module de gestion de pile
  - Tester les 3 types de T_elt
- Comment empêcher la pile de déborder ?
  - ⇒ Responsabilité du programmeur du module ?
    - Quelles contre-mesures sont mises en place ?
  - ⇒ Responsabilité du programmeur utilisateur ?
    - Nécessite de connaître l'implémentation interne
    - Quelles fonctions lui proposer ?

centralelille

AUDACE • EXIGENCE • RESPECT

2h00

TAD pile : Implémentation dynamique et contiguë



TAD pile : Implémentation **dynamique** et contiguë

- La pile peut être aussi grande que l'on veut
- Peu de modifications nécessaires :
- ~~T_stack newStack(void)~~
- T_stack newStack(**int size**)

~~#define NBMAX_ELT 10~~

```
typedef struct {
    int sp;
    int nbMaxElt;
    ~~T_elt data[NBMAX_ELT];~~
    T_elt * data;
} T_stack;
```

void *realloc(void *ptr, size_t size);
Cf. man realloc

- Modifie la taille du bloc de mémoire pointé par ptr à la taille de size octets
- Le contenu de la mémoire de départ n'est pas modifié
- Le contenu de la zone de mémoire nouvellement allouée n'est pas initialisé
  - La zone mémoire peut être déplacée si l'espace disponible ne permet pas de l'étendre sur place. Si la zone pointée est déplacée, un free(ptr) est effectué auparavant (sur l'ancienne zone)
- Si ptr est NULL, l'appel est équivalent à malloc(size), si size vaut zéro et ptr est non NULL, l'appel est équivalent à free(ptr)

Exercice corrigé : ex4 (1)
stack_cd.h & stack_cd.c

_cd.* : contiguë dynamique

- Implémenter le module de gestion de pile dynamique et contiguë : stack_cd.h et stack_cd.c
- **T_stack newStack(int size)**
- **void freeStack (T_stack *p)**
- **void push(T_elt e, T_stack *p)**
- **T_stack exampleStack(int n)**
- void showStack (const T_stack * p)
- T_elt pop(T_stack *p)
- T_elt top(const T_stack *p)
- void emptyStack (T_stack *p)
- int isEmpty (const T_stack * p)

⇒ à adapter : utilisation de newStack(n)

rien ne change !

Exercice corrigé : ex4 (2)
stack_cd.h & stack_cd.c

_cd.* : contiguë dynamique

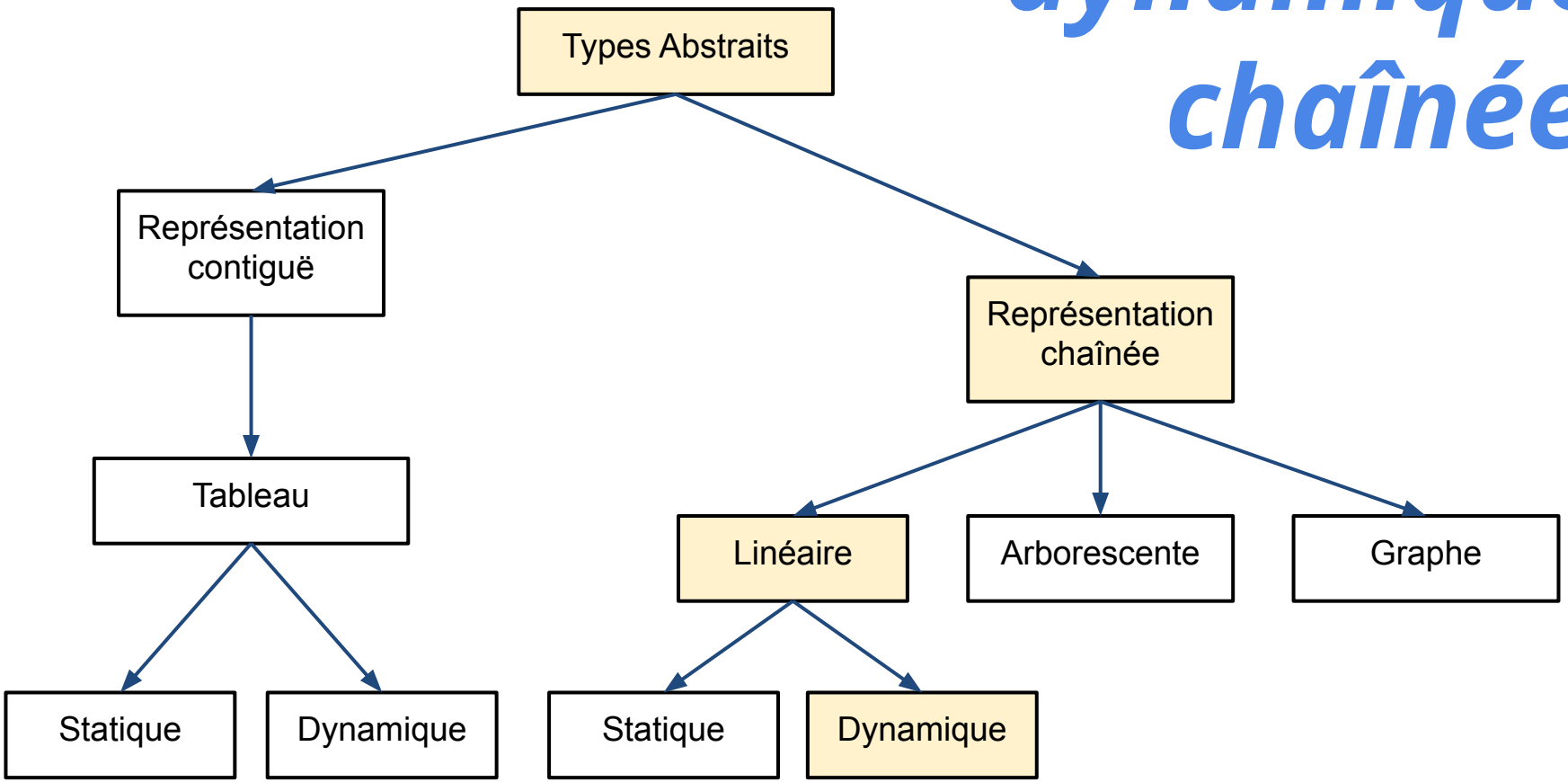
- Adapter votre programme de test
  - Tester l'ajout dans une pile trop petite
- Attention aux ajustements de la taille de la pile dans la fonction void push(T_elt e, T_stack *p)
  - Prévoir des seuils d'agrandissement (constante symbolique STACK_THRESHOLD) pour éviter des realloc systématiques

centralelille

AUDACE • EXIGENCE • RESPECT

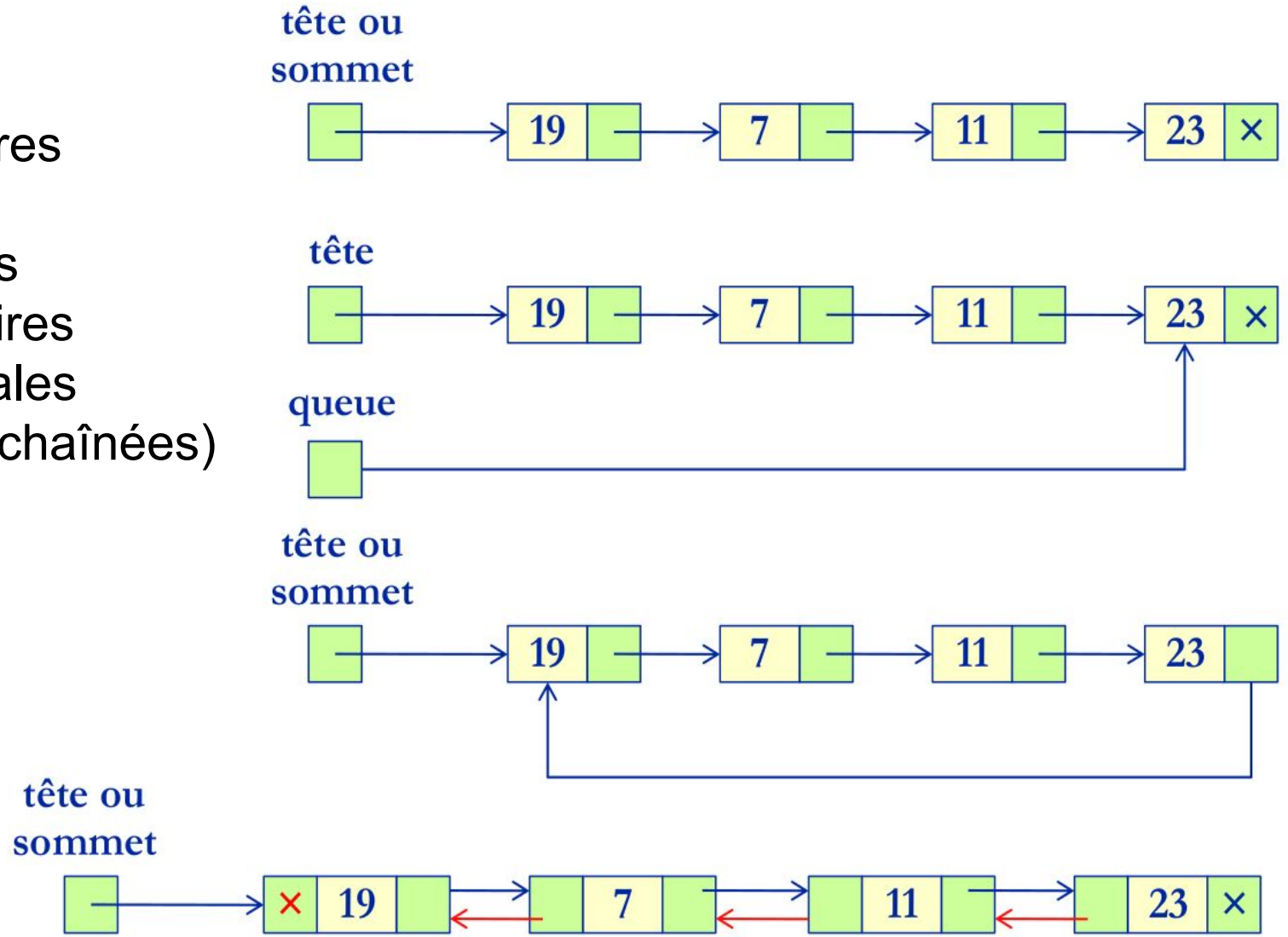
1h30

TAD pile : Implémentation dynamique et chaînée



Structures linéaires chaînées

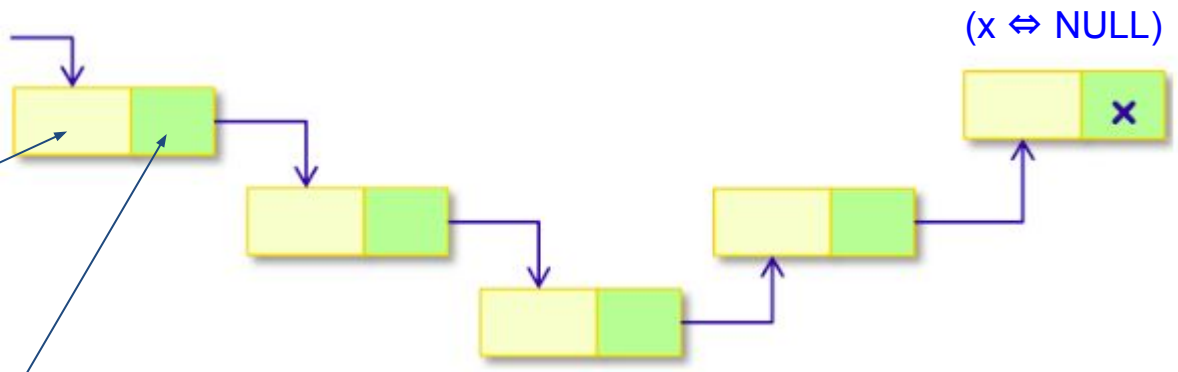
- Plusieurs structures possibles :
  - Listes simples
  - Listes circulaires
  - Listes bilatérales (doublement chaînées)



Liste chaînée

- Liste chaînée : un ensemble de cellules (ou mailles, noeuds) dont seul le 1er élément est accessible directement
- Caractérisée par :
  - Le type d'information portée par chaque cellule
  - Un pointeur sur la tête de liste
  - Un ensemble d'opérations élémentaires

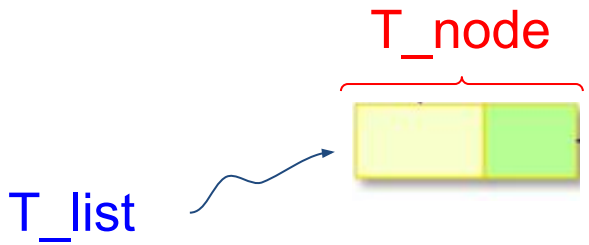
Liste chaînée : Structure *auto-référente*




```
typedef struct node {
    T_elt data;
    struct node *pNext;
} T_node, * T_list;
// typedef T_node * T_list;
```

T_node ⇔ struct node

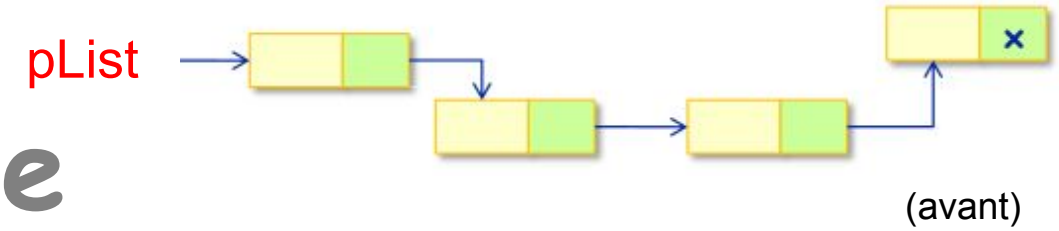
T_list ⇔ T_node *



Opération de base

- Créer une maille (node), la remplir et l'accrocher en tête d'une liste existante (ou vide), renvoyer la nouvelle tête
  - T_node * addNode (T_elt e, T_node * p);
- De manière équivalente :
  - T_list addNode (T_elt e, T_list l);

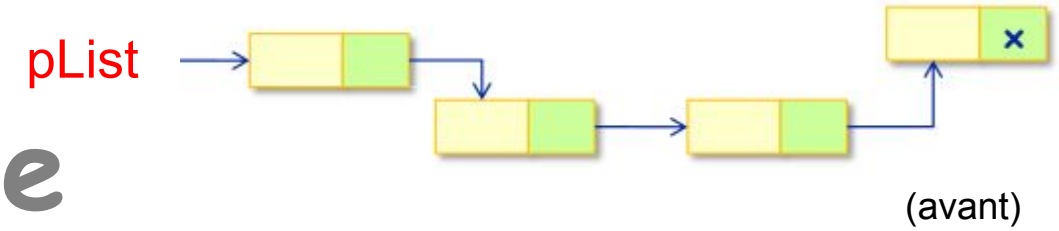
Opération de base



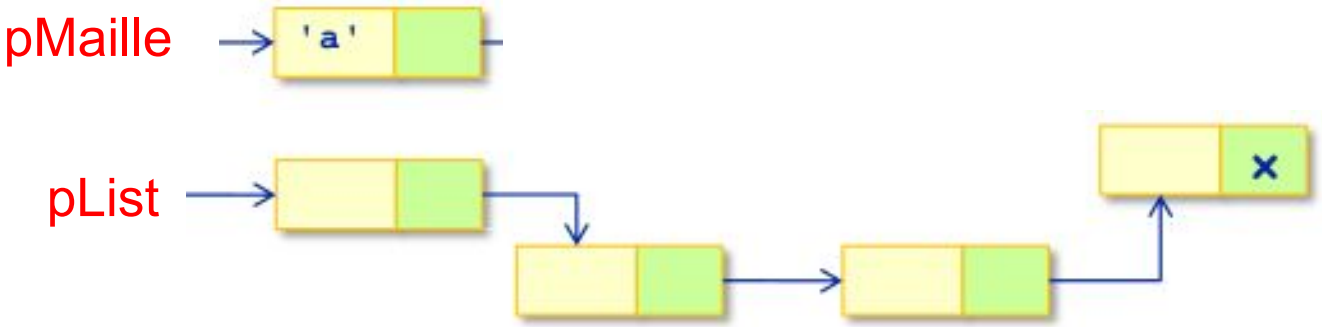
(avant)

- Opération : pList = addNode(‘a’,pList);

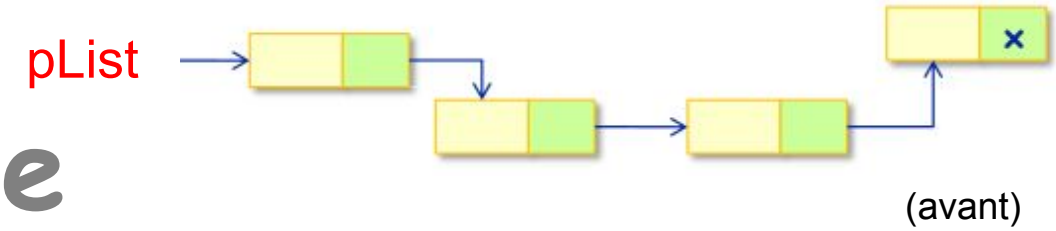
Opération de base



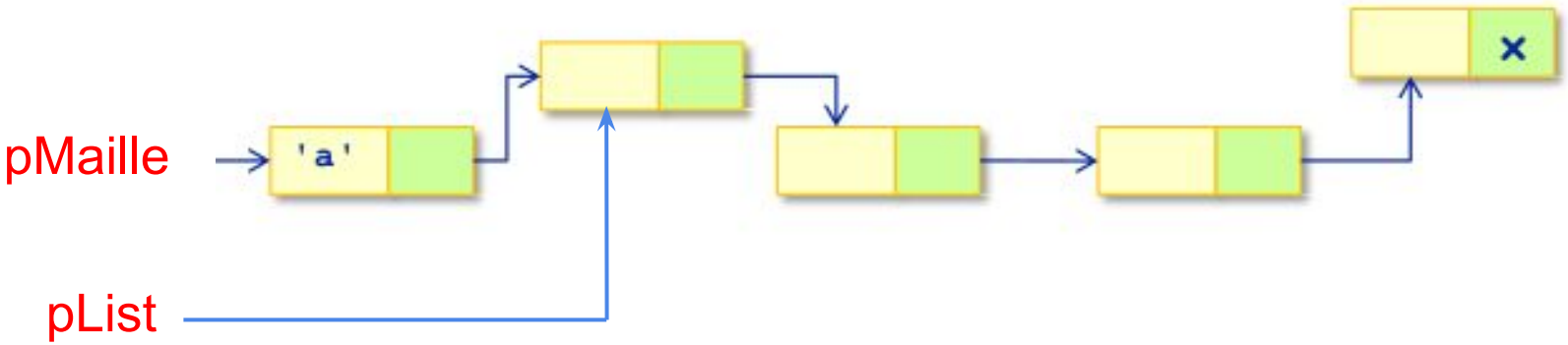
- Opération : pList = addNode(‘a’,pList);
- Créer une maille, la remplir



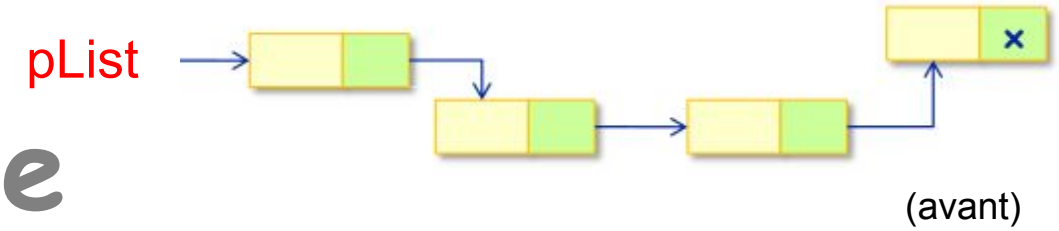
Opération de base



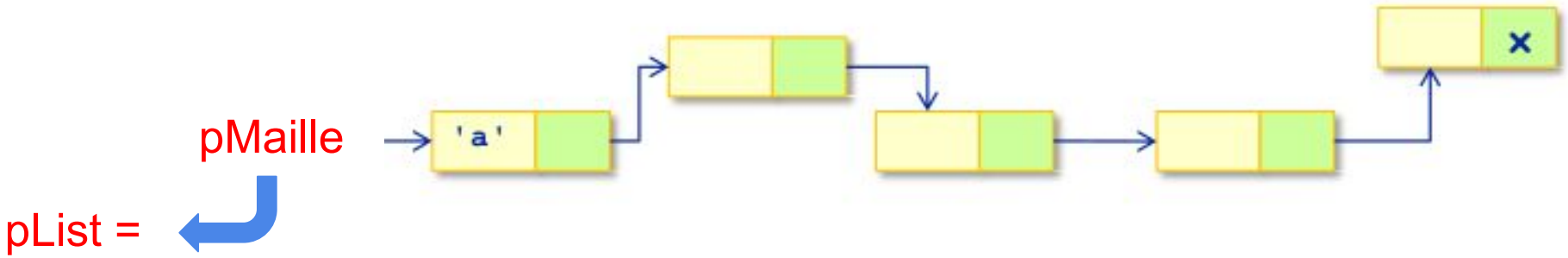
- Opération : pList = addNode(‘a’,pList);
- Créer une maille, la remplir
- L’accrocher en tête de liste



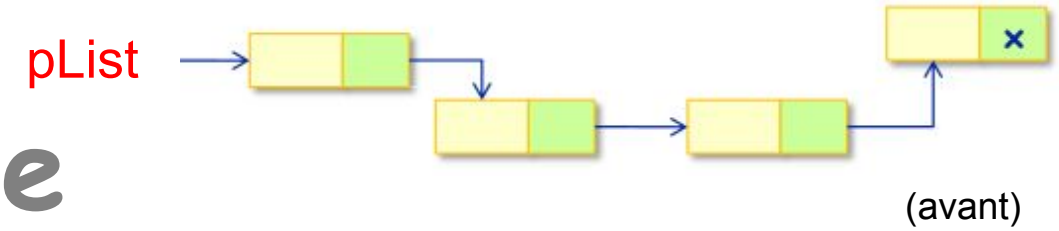
Opération de base



- Opération : pList = addNode(‘a’,pList);
- Créer une maille, la remplir
- L’accrocher en tête de liste
- Renvoyer la nouvelle tête

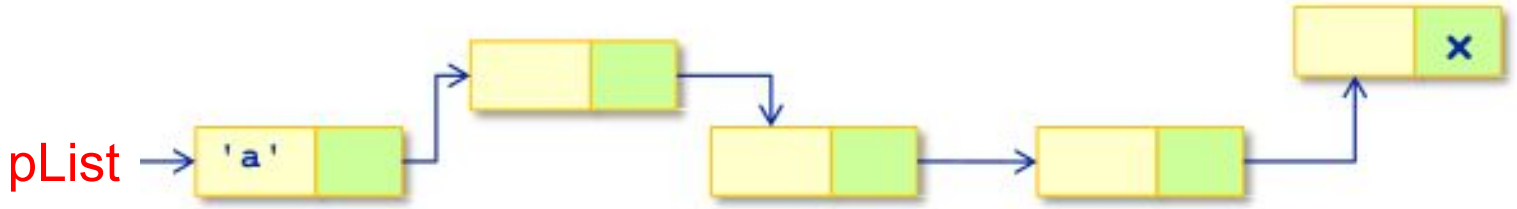


Opération de base



(avant)

- Opération : pList = addNode(‘a’,pList);
- Créer une maille, la remplir
- L’accrocher en tête de liste
- Renvoyer la nouvelle tête



Exercice : ex5 (1)

1h15

30 min

- Produire un module de gestion de liste complet :
  - Fichier d'entête list.h
  - Fichier source list.c
  - Fichier source elt.h
  - Fichier source elt.c
- Penser à y associer des traces d'exécution et vérifier les valeurs de retour des appels système
- Tester en créant une liste de T_elt de type "chaîne"
  - Tester sa libération

Exercice : ex5 (2)

list.c

Développer en itératif, pas en récursif

1h15

30 min

- T_node * addNode (T_elt e, T_node * p)
- T_node * newNode(T_elt e)
  - Déclarer cette fonction static : elles ne sera pas utilisable ailleurs que dans le fichier list.c
- T_elt getFirstElt(T_list l)
- T_list removeFirstNode(T_list l)
- void showList(T_list l)
- void freeList(T_list l)
- T_list newListe(void) est-elle nécessaire ?

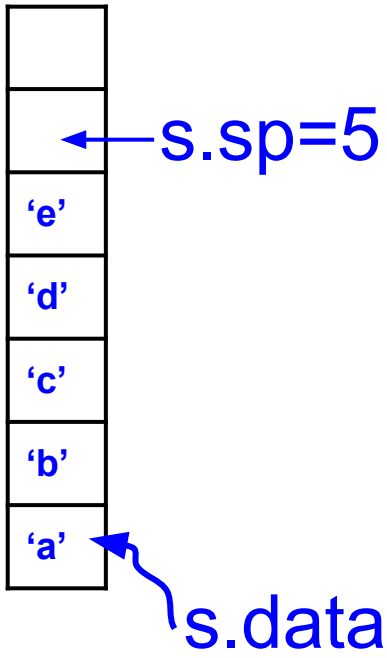
Pile et liste chaînée

- Liste chaînée = implémentation adaptée pour une pile lorsque :
  - Le nombre maximal d'éléments à empiler n'est pas connu
  - Il est nécessaire d'éviter des realloc fréquents et coûteux en temps de traitement

Pile et liste chaînée : structure de pile

// implémentation contiguë

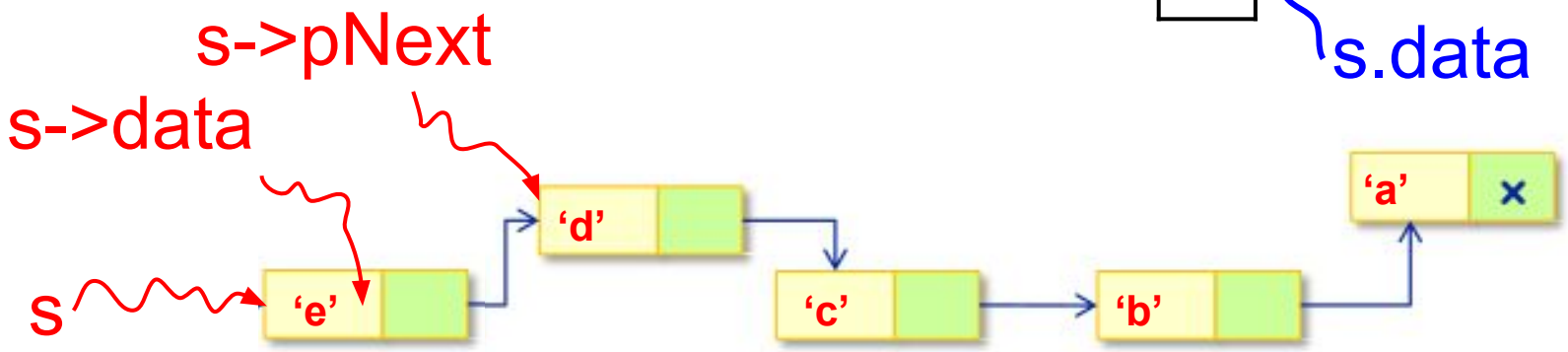
T_stack s;



- typedef T_node * T_stack;
- Équivalent à typedef T_list T_stack;
- Une pile se représente tout naturellement par une liste chaînée !!

// Implémentation chaînée

T_stack s;



Exercice : ex6 (1)
stack_cld.h & stack_cld.c

_cld.* : chaînée linéaire dynamique

- Développer les fichiers stack_cld.h et stack_cld.c
- ~~T_stack newStack(int size)~~
- T_stack newStack(void)
- void freeStack (T_stack *p) ⇒ freeList
- void push(T_elt e, T_stack *p) ⇔ addNode
- void showStack (const T_stack * p) ⇔ showList
- T_elt pop(T_stack *p) ⇒ getFirstElt, removeFirstNode
- T_elt top(const T_stack *p) ⇔ getFirstElt
- int isEmpty (const T_stack * p)

Exercice : ex6 (2)
stack_cld.h & stack_cld.c

_cld.* : chaînée linéaire dynamique

45min

20 min

- T_stack décrivant maintenant un pointeur sur T_node, comment redéfinir les prototypes pour éviter une lourdeur d'implémentation ?
  - Ne pas le programmer

Récursivité

- En C, il est possible de développer une fonction qui s'appelle elle-même
  - Fonction récursive
- Processus de développement similaire à la démonstration par récurrence en mathématiques
- Pas forcément performant, mais rapide à développer !
  - Sollicite beaucoup la… pile d'exécution !
- Attention aux boucles infinies !

Récursivité

- En Maths
- Exprimer la propriété P(n)
  - Ordre de la récurrence : n
- Vérifier P(0)
- Faire l'hypothèse P(n)
  - Vérifier P(n+1)

- En Info
  - Identifier l'ordre de la récurrence
    - un paramètre entier, la taille d'une liste, d'un tableau…
- Coder foo() :
  - Coder le cas de base (si ordre d'appel = ordre initial)
  - Coder le cas général en utilisant des appels récursifs à foo()
    - Vérifier que l'ordre de la récurrence a strictement diminué entre l'appel initial et l'appel récursif !
      - Sinon… Boucle infinie !

Exercice corrigé : ex5 (3)

25min

15 min

- Implémenter les versions récursives des fonctions suivante s:

  void showList_rec(T_list l)

  void showList_inv_rec(T_list l)

  void freeList_rec(T_list l)

- [BONUS ***Facile***] Une constante symbolique permet de sélectionner la version (récursive ou itérative) à employer

Dérécursivation d'algorithmes

- Transformation d'un algorithme récursif en algorithme itératif
- Toujours possible
  - Facile si récursion terminale
  - Si récursion non terminale, utiliser une pile !
- Automatiquement réalisé par le compilateur si récursion terminale
  - Option de gcc : -O2
  - https://fr.wikipedia.org/wiki/R%C3%A9cursion_terminale
  - https://dev.to/rohit/demystifying-tail-call-optimization-5bf3
- [BONUS ***Facile***] Implémenter la fonction showList_rec de manière itérative

Dérécursivation d'algorithmes
Récursivité terminale : facile !

```
Foo(cas)
    Si  (IS_BASE_CASE(cas))
        renvoyer BASE_RESULT(cas)
    Sinon
        next_cas= OPERATIONS(cas)
        renvoyer Foo(next_cas)
```

// appel à Foo() terminal : on renvoie directement le résultat de l'appel récursif

```
Foo(cas)
    aux = cas
    TANT QUE ( ! IS_BASE_CASE(aux))
    FAIRE
        aux= OPERATIONS(aux)
        // aux récupère à chaque itération
        // ce qu'aurait dû traiter l'appel récursif
    FIN FAIRE

renvoyer BASE_RESULT(aux)
```

BONUS

10min

5 min

- Tours de Hanoï
- https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF
- Quelle est la complexité de cette fonction ?
  - Sans doute pas très grande, si elle prend 5min à écrire...
  - Réponse au prochain épisode...

TEA S2

- TEA S2 - 2022
- Résolution du jeu “Le compte est bon”
  - Approche exhaustive (par “force brute”)
- Parcours d’un arbre à la volée en profondeur d’abord
- Application des piles :
  - Evaluation d’expressions en notation polonaise inverse

Dépôt du TEA

- Modalités :
  - Dépôt sur moodle
  - Livrer au minimum 24h avant la prochaine séance
  - Travail en équipe de 4 étudiants maximum, du même groupe TD
  - Pas plus de 8 équipes par groupe TD
  - Rendre une seule archive par groupe
- Cf. capsule 6 :
  - Porter attention au respect du cahier des charges mais aussi :
    - de la forme du code
    - de la qualité du code
  - Travailler avec goût :
    - CR avec introduction, conclusion, jeux d'essais
  - Respecter les critères de livraison (pdf, 1 seule archive, attention au nommage, pas de binaires...)

centralelille

AUDACE • EXIGENCE • RESPECT

Critères d' évaluation des travaux à rendre

Critères d'évaluation : Livraison

- Délais / contacts / protocole
- Ne rendre qu'un seul travail pour tout le groupe
- Pas de contenu binaire !
  - Seulement les fichiers sources, et le fichier makefile associé
  - Ne passe pas les filtres antivirus
- Tous les documents dans une même archive :
  - Code source, makefile, README, CR

Remise du travail : Attention aux irritants !

- Soigner le nom du fichier
  - Il doit porter un nom représentatif du travail ET de son auteur
    - Pour qu'on puisse le distinguer des devoir rendus par les autres étudiants…
  - Jamais d'espaces dans les noms des fichiers/répertoires
  - Une archive compactée si plusieurs fichiers
- Fichier "README" reprenant les auteurs et une présentation rapide du travail
  - Format texte
- Fichier CR.pdf
  - Le compte-rendu lui-même
  - Utiliser un format portable, pas de docx, odt..

Critères d'évaluation : Fond

- Respect du cahier des charges
- Ergonomie de la solution
- Le CR doit comporter jeux d'essais et analyse
  - Rédigé avec goût : introduction, conclusion & perspectives, bibliographie
  - Section Gestion de Projet si travail collectif : organisation, qui a fait quoi...
  - Ne pas cacher les problèmes rencontrés : les analyser / en tirer des leçons

Critères d'évaluation : Forme

- Tout ce qui permet de s'y retrouver quand on reprend le programme
- Lisibilité globale
  - Indentation : tabulations, passage à la ligne
  - Quantité, mise en forme et pertinence des commentaires
  - Conventions de nommage
  - Choix des noms de paramètres, variables, fonctions, librairies pertinents
- Orthographe et expressions dans l'interface
- Organisation des fichiers

Critères d'évaluation : Qualité

- Tout ce qui est indépendant de la fonction du programme
- Emploi de structures de contrôle adaptées et évoluées quand c'est nécessaire (foreach, switch)
- Fermeture des fichiers ouverts
- Tests aux limites : appels systèmes …
- Utilisation raisonnée des variables globales et locales
- Emploi de chemins relatifs uniquement pour faciliter la vérification du travail
- Modularité adaptée (traitements réutilisables, paramètres utiles et bien documentés, pas d'effets de bord néfastes), utilisation de librairies
- Robustesse, messages d'erreur pour l'utilisateur

centralelille

AUDACE • EXIGENCE • RESPECT

Code Couleur

Légende des textes

- mot-clé important, variable, contenu d'un fichier, code source d'un programme
- chemin ou url, nom d'un paquet logiciel
- commande, raccourci
- commentaire, exercice, citation
- culturel, optionnel

Culturel / Approfondissement

- A ne pas connaître intégralement par coeur
  - Donc, le reste… est à maîtriser parfaitement !
- Pour anticiper les problématiques que vous rencontrerez en stage ou dans d'autres cours
- Pour avoir de la conversation à table ou en soirée...

Exemples ou Exercices

- Brancher le cerveau
- Participer
- Expérimenter en prenant le temps...

Bonnes pratiques, prérequis

- Des éléments d'organisation indispensables pour un travail de qualité
- Des rappels de concepts déjà connus