Université Claude Bernard Lyon 1

Année universitaire : 2025 / 2026

LIFAPSD : Algorithmique, Programmation et Structures de Données

Epreuve de Seconde Chance
5 janvier 2026
Durée : 1h30

Note : / 20

Nom :
Prénom :
N° d'étudiant :
Signature :

coller ici

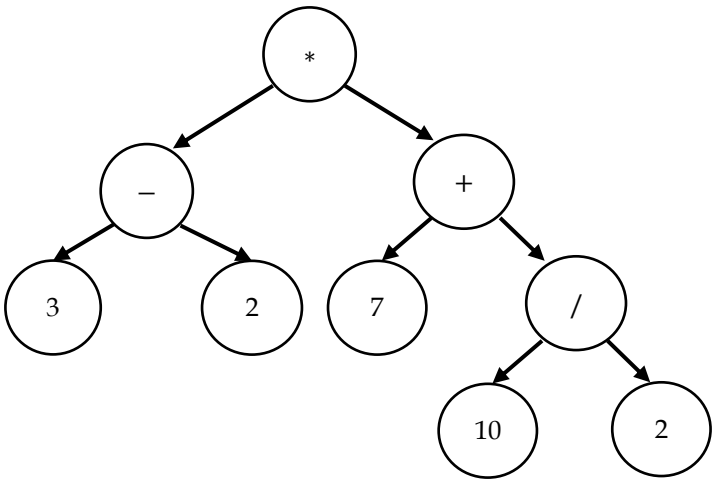
Documents et téléphones portables interdits. Répondez dans les emplacements prévus à cet effet. Travaillez au brouillon d'abord de sorte à rendre une copie propre – nous ne pouvons pas vous garantir une copie supplémentaire. **Il se tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Arbre binaire et expression arithmétique

On considère les expressions arithmétiques sur les entiers n'utilisant que les opérateurs +, −, * et /. Ces expressions peuvent être représentées par des arbres binaires dont les nœuds « internes » (nœuds qui ne sont pas des feuilles) sont étiquetés par l'un des quatre opérateurs tandis que les feuilles sont étiquetées par des entiers.

Pour rappel, le **`Noeud`** d'un arbre binaire est défini par une structure contenant trois champs : **`info`** de type **`ElementA`**, **`fg`** et **`fd`** tous les deux de type pointeur sur **`Noeud`**. Et la seule donnée membre de la classe **`Arbre`** est **`adRacine`**, un pointeur sur le **`Noeud`** racine de l'arbre.
Les fonctions membres de la classe **`Arbre`** sont rappelées en annexe.

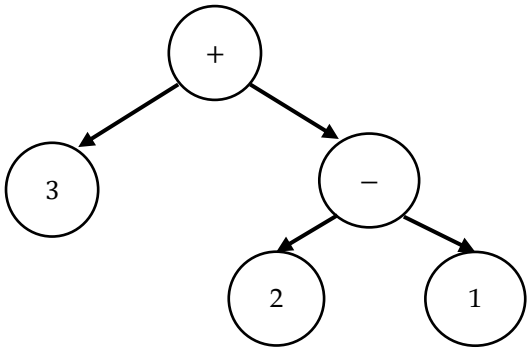
Ainsi, l'arbre ci-dessous représente l'expression arithmétique (3 − 2) * (7 + 10 / 2).



Les opérateurs seront représentés par le caractère (**`char`**) qui leur correspond (**`'+'`**, **`'-'`**, **`'*'`** ou **`'/'`**) dans les nœuds internes. Les entiers seront représentés par des entiers (**`int`**) dans les feuilles.

Question 1 : Dessiner l'arbre correspondant à l'expression arithmétique $3 + (2 - 1)$.

L'arbre est :



Question 2 : Afin de stocker l'information des nœuds, nous allons remplacer le champ **`info`** de la structure **`Noeud`** par deux nouveaux champs, l'un pour le caractère d'un opérateur, nommé **`op`**, et l'autre pour un entier, nommé **`val`**. Donner la définition de la nouvelle structure **`Noeud`**.

```
struct Noeud {
   char op;
   int val;
   Noeud * fg;
   Noeud * fd;
};
```

Question 3 : Afin de faciliter le traitement de l'expression arithmétique et donc de l'arbre, donner le code C++ d'une nouvelle fonction membre de la classe **`Arbre`** :

Fonction estFeuille (n : lien sur Noeud) : booléen
Précondition : le lien n est une adresse valide d'un nœud de l'arbre
Résultat : vrai si le nœud en paramètre est une feuille, faux sinon
Paramètre en mode donnée : n

```
bool Arbre::estFeuille (const Noeud * n) const {
   return n->fg == nullptr && n->fd == nullptr;
}
```

Question 4 : Ecrire en C++ la fonction membre **valide** de la classe **Arbre** qui teste si l'arbre binaire, ne contenant que des entiers et des opérateurs, représente bien une expression arithmétique valide, c'est-à-dire ici simplement que chaque nœud interne a deux fils non vides.
Indication : on peut vérifier que la condition est vraie récursivement sur les sous-arbres gauche et droit.

```
bool Arbre::valide () const {
   if (estVide()) return true;
   return valideAPartirDeNoeud(adRacine);
}

bool Arbre::valideAPartirDeNoeud (const Noeud * n) const {
   if (estFeuille(n)) return true;
   if (n->fg == nullptr) return false;
   bool gauche = valideAPartirDeNoeud(n->fg);
   if (n->fd == nullptr) return false;
   bool droit = valideAPartirDeNoeud(n->fd);
   return gauche && droit;
}
```

Question 5 : Écrire la fonction membre **evaluation** de la classe **Arbre** qui renvoie la valeur entière correspondant à l'évaluation de l'expression arithmétique représentée par l'arbre. Par exemple sur l'arbre illustré en première page, la fonction retournera la valeur entière 12 et sur l'arbre de la question 1 elle retournera la valeur 4. On suppose ici que l'arbre représente une expression valide (y compris qu'il n'y a pas de division par 0).
Indication : si le nœud est une feuille, l'évaluation en ce nœud doit renvoyer la valeur de l'entier, si par contre le nœud correspond à un opérateur, l'évaluation doit récupérer le résultat de l'évaluation du fils gauche et du fils droit et réaliser l'opération entre les deux.

```
int Arbre::evaluation () const {
   if (estVide()) return 0;
   return evaluationAPartirDeNoeud(adRacine);
}

int Arbre::evaluationAPartirDeNoeud (const Noeud * n) const {
   if (estFeuille(n)) return n->val;
   int val_gauche = evaluationAPartirDeNoeud(n->fg);
   int val_droit = evaluationAPartirDeNoeud(n->fd);
   switch (n->op) {
      case '+': return val_gauche + val_droit; break;
      case '-': return val_gauche - val_droit; break;
      case '*': return val_gauche * val_droit; break;
      case '/': return val_gauche / val_droit; break;
   }
}
```

Question 6 : Quel parcours récursif en profondeur de l'arbre permet de récupérer la notation habituellement utilisée pour une expression arithmétique (opérande de gauche puis opérateur puis opérande de droite, i.e. comme dans l'exemple de la première page et la question 1, à l'exception des parenthèses) ?

Il s'agit du parcours infixe.

Question 7 : Donner le code C++ de la fonction membre **`afficher`** de la classe **`Arbre`** qui affiche à l'écran cette notation habituelle. Vous ajouterez l'affichage des parenthèses ouvrantes et fermantes (qu'elles soient indispensables ou non).

```
void Arbre::afficher () const {
   if (!estVide()) afficherAPartirDeNoeud(adRacine);
}

void Arbre::afficherAPartirDeNoeud (const Noeud * n) const {
   if (estFeuille(n)) cout << n->val;
   else {
      cout << "(" ;
      afficherAPartirDeNoeud(n->fg);
      cout << ") " << n->op << " (" ;
      afficherAPartirDeNoeud(n->fd);
      cout << ")" ;
   }
}
```

Question 8 : Donner et justifier brièvement la complexité algorithmique de la fonction **`afficher`**, en notation *O*, en fonction de n le nombre de nœuds de l'arbre.

La fonction parcourt l'entièreté de l'arbre en ne traitant, récursivement, qu'une seule fois chaque nœud. La fonction est donc de complexité *O*(n).

Question 9 : Quelle relation pouvez-vous établir entre le nombre d'opérateurs (nœuds internes) et le nombre d'entiers (feuilles) de l'arbre ?

Les opérateurs sont tous binaires (opérandes gauche et droite). Un opérande est soit 1 entier soit une expression arithmétique (sous-arbre avec 1 opérateur et 2 opérandes). Il y aura donc toujours 1 feuille de plus que de nœuds internes.

Exercice 2 : Insertion avec extension

Question 10 : Donner le code C++ de la fonction membre **`insererElement`** de la classe **`TableauDynamique`** qui insère un élément à un indice donné. Cette fonction autorisera un indice plus grand que le nombre d'éléments du tableau.

Procédure insererElement (e : ElementTD, indice : entier naturel)
Précondition : indice ≥ 0
Postcondition : l'élément e est inséré à l'indice passé en paramètre. Si l'indice correspond à un élément n'existant pas encore, l'élément e est ajouté autant de fois que nécessaire pour l'insérer à l'indice passé en paramètre.
Paramètre en mode donnée : e, indice
Exemple : Si le tableau contient les éléments [4,1,2,8] et que l'on insère l'élément 3 à l'indice 2, le tableau contiendra les éléments [4,1,3,2,8]. Si le tableau contient les éléments [2,5,0] et que l'on insère l'élément 4 à l'indice 5, le tableau contiendra les éléments [2,5,0,4,4,4].

```
void TableauDynamique::insererElement (ElementTD e, unsigned int indice) {
   if (indice >= taille_utilisee) { // nécessite que des ajouts en fin
      for (int i = 1; i <= indice - taille_utilisee + 1; i++) { // autant que nécessaire
         ajouterElement(e);
      }
   }
   else {
      ajouterElement(e); // ajout temporaire d'un élément à la fin
      for (int i = taille_utilisee – 1; i > indice; i--) ad[i] = ad[i-1]; // décalage à droite
      ad[indice] = e; // insertion de l'élément
   }
}
```

Question 11 : Donner le code C++ de la fonction membre **`insererElement`** de la classe **`Liste`** doublement chaînée qui insère un élément à un indice donné. Cette fonction autorisera un indice plus grand que le nombre d'éléments de la liste.

Procédure insererElement (e : ElementL, indice : entier naturel)
Précondition : indice ≥ 0
Postcondition : l'élément e est inséré à l'indice passé en paramètre. Si l'indice correspond à un élément n'existant pas encore, l'élément e est ajouté autant de fois que nécessaire pour l'insérer à l'indice passé en paramètre.
Paramètre en mode donnée : e, indice
Exemple : Si la liste contient les éléments (4,1,2,8) et que l'on insère l'élément 3 à l'indice 2, la liste contiendra les éléments (4,1,3,2,8). Si la liste contient les éléments (2,5,0) et que l'on insère l'élément 4 à l'indice 5, la liste contiendra les éléments (2,5,0,4,4,4).

```
void Liste::insererElement (ElementL e, unsigned int indice) {
   if (indice == 0) ajouterEnTete(e); // ajout en tête
   else {
      Cellule * c = prem;
      int i = 0; // on se place sur la cellule de l'indice ou bien au bout
      while (c != nullptr && i < indice) {c = c->suivant; i++;}
      if (c == nullptr) { // au bout
         for (int i = 1; i <= indice - taille_utilisee + 1; i++) { // ajouts autant que nécessaire
            ajouterEnQueue(e);
         }
      }
      else { // insertion "normale"
         Cellule * nouv = new Cellule; // création de la nouvelle cellule
         nouv->info = e;
         nouv->suivant = c;
         nouv->precedent = c->precedent;
         c->precedent->suivant = nouv; // mise à jour liens
         c->precedent = nouv;
      }
   }
}
```

Annexe : liste des fonctions membres des classes TableauDynamique, Liste, File, Pile et Arbre (constructeurs et destructeurs omis)

Classe TableauDynamique

```
void vider ();
void ajouterElement (ElementTD e);
ElementTD valeurIemeElement (unsigned int indice) const;
void modifierValeurIemeElement (ElementTD e, unsigned int indice);
void afficher () const;
void supprimerElement (unsigned int indice);
void insererElement (ElementTD e, unsigned int indice);
int rechercherElement (ElementTD e) const;
```

Classe Liste

```
void vider ();
bool estVide () const;
ElementL iemeElement (unsigned int indice) const;
void modifierIemeElement (unsigned int indice, ElementL e);
void afficherGaucheDroite () const;
void afficherDroiteGauche () const;
void ajouterEnTete (ElementL e);
void ajouterEnQueue (ElementL e);
void supprimerTete ();
int rechercherElement (ElementL e) const;
void insererElement (ElementL e, unsigned int indice);
void supprimerElement (ElementL e);
```

Classe File

```
void enfiler (ElementF e);
ElementF premierDeLaFile () const;
void defiler ();
bool estVide () const;
unsigned int nbElements () const;
```

Classe Pile

```
void empiler (ElementP e);
ElementP consulterSommet () const;
void depiler ();
bool estVide () const;
```

Classe Arbre

```
bool estVide () const;
void vider ();
void insererElement (ElementA e);
void supprimerElement (ElementA e);
Noeud * rechercherElement (ElementA e) const;
```