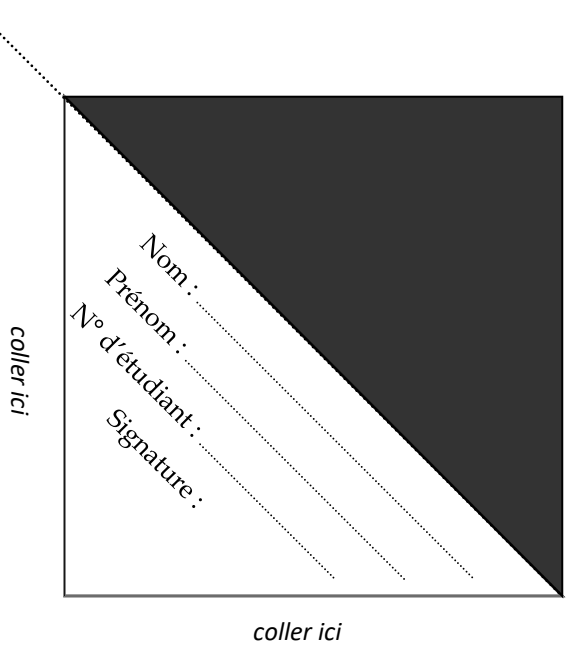
Université Claude Bernard Lyon 1

Année universitaire : 2017 / 2018

LIFAP3 : Algorithmique et programmation avancée

Contrôle final
12 janvier 2018
Durée : 1h30

Note : / 20



Documents et téléphones portables interdits. Le barème est donné à titre indicatif. 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 sera tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Tableau dynamique et liste chaînée (8 points)

Soit la procédure globale **`mystere`** suivante, donnée en notation algorithmique.

Procédure mystere (tab : TableauDynamique d'entiers, n : entier, l1 : Liste d'entiers, l2 : Liste d'entiers)
Précondition : l1 et l2 sont des listes vides
Paramètres en mode donnée : tab et n
Paramètre en mode donnée-résultat : l1 et l2
Variables locales : i : entier positif ou nul
Début
  Pour i allant de 0 à tab.taille_utilisee-1 par pas de 1 faire
    Si tab[i] modulo n = 0 alors l1.ajouterEnTete(tab[i])
    Sinon l2.ajouterEnTete(tab[i]) FinSi
  FinPour
Fin mystere

Question 1.1 : Traduire cette procédure en C++.

```
void mystere (const TableauDynamique & tab, int n, Liste & l1, Liste & l2) {
   for (unsigned int i = 0; i < tab.taille_utilisee; i++) {
      if (tab[i] % n == 0) l1.ajouterEnTete(tab[i]);
      else l2.ajouterEnTete(tab[i]);
   }
}
```

Soit le programme principal suivant, donné en notation algorithmique.

```
Variables locales : tab : TableauDynamique d'entiers, l1 et l2 : Liste d'entiers
Début
  tab.ajouterElement(0)
  tab.ajouterElement(-1)
  tab.ajouterElement(3)
  tab.ajouterElement(2)
  tab.ajouterElement(-3)
  mystere(tab,2,l1,l2)
  l1.afficherGaucheDroite()
  l2.afficherGaucheDroite()
Fin
```

Question 1.2 : Qu'affiche ce programme principal à l'écran ?

```
2 0
-3 3 -1
```

Question 1.3 : Expliquer ce que fait la procédure **`mystere`**.

La procédure **`mystere`** sépare les multiples de n des non multiples. Elle met les multiples dans l1 et les non multiples dans l2. L'ordre des éléments dans les deux listes est inversé par rapport à l'ordre où ils apparaissent dans le tableau.

Question 1.4 : Donner le code permettant de créer et remplir un tableau dynamique avec les nombres entiers de 2 à 99.

```
TableauDynamique tab;
for (unsigned int i = 2; i <= 99; i++) tab.ajouterElement(i);
```

Question 1.5 : Compléter les trous dans la fonction **nombresPremiers** suivante qui calcule et retourne les nombres premiers plus petits que 99. La fonction prend en paramètre le tableau créé à la question précédente en mode donnée-résultat et retourne un nouveau tableau créé sur le tas contenant les nombres premiers. Le tableau en paramètre est vidé.

```
TableauDynamique * nombresPremiers (TableauDynamique & tab) {
   TableauDynamique * premiers = new TableauDynamique;
   while (tab.taille_utilisee > 0) {
      premiers->ajouterElement(tab.ad[0]);             // ajout du premier élément de tab
      Liste multiples;                                 // création de deux listes
      Liste autres;
      mystere(tab,tab.ad[0],multiples,autres);        // appel à mystere
      Cellule * c = multiples.adPremiere;
      while (c != NULL) {       // parcours d'une liste pour supprimer ses éléments de tab
         tab.supprimerElement(tab.rechercherElement(c->info));
         c = c->suivant;
      }
   }
   return premiers;
}
```

Exercice 2 : Arbre binaire compact (12 points)

Dans cet exercice, on s'intéresse aux types de donnée abstrait représentant un arbre binaire compact. Dans un arbre binaire compact tous les niveaux sauf le dernier doivent être totalement remplis et si le dernier ne l'est pas totalement, alors il doit être rempli de gauche à droite (cf. Fig. 1).

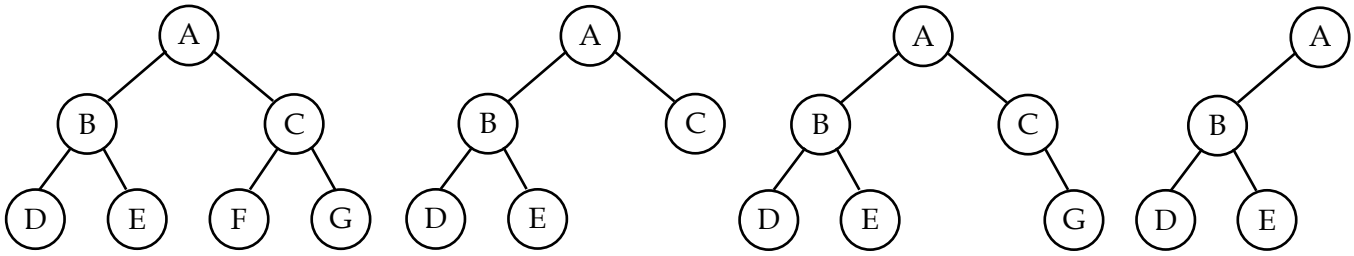


Fig. 1 : Les deux arbres binaires (a) et (b) sont compacts.
(c) n'est pas compact car le dernier niveau incomplet n'est pas rempli de gauche à droite.
(d) n'est pas compact car le troisième niveau est commencé alors que le deuxième n'est pas complet.

Lorsqu'un arbre est compact, on peut l'implémenter avec un tableau statique dont la taille est le nombre de nœuds (cf. Fig. 2). Dans ce tableau, l'information à la racine se situe à l'indice 0. Pour un élément à l'indice i du tableau, l'information du fils gauche est à l'indice $2i + 1$ et celle du fils droit à l'indice $2i + 2$.

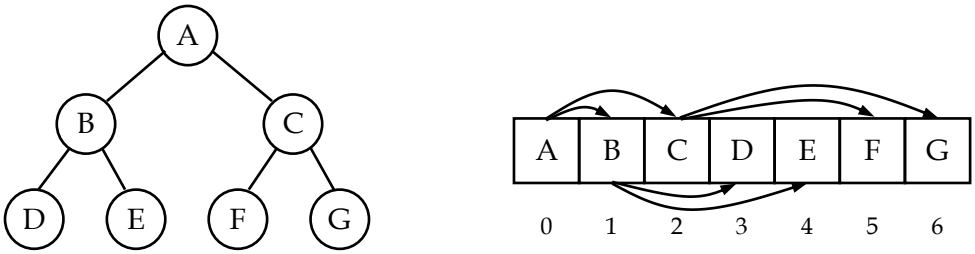


Fig. 2 : L'arbre binaire compact à gauche peut être représenté par le tableau statique à droite

Pour rappel, la classe **Arbre** a une seule donnée membre : **adRacine** de type pointeur sur **Noeud**. La structure **Noeud** a trois champs : **info** de type **ElementA**, **fg** et **fd** tous deux de type pointeur sur **Noeud**. Voir l'annexe pour les fonctions membres.

Question 2.1 : Quel est le principal avantage à représenter un arbre binaire compact avec un tel tableau statique plutôt qu'avec la représentation utilisée en cours ?

La représentation en tableau statique prend moins d'espace mémoire. On a en effet que à stocker les éléments, l'organisation (ex. le parcours de l'arbre) est effectuée en manipulant les indices dans les calculs. Dans la représentation vue en cours, on stockait en plus les adresses des fils gauche et droit de chaque nœud et l'adresse de la racine.

Question 2.2 : Ecrire la fonction globale **convertirArbreEnTableau** permettant de créer un tableau statique d'**ElementA** représentant un arbre binaire compact. L'arbre sera passé en paramètre en mode donnée et le tableau sera créé sur le tas et retourné par la fonction. Vous pourrez supposer que vous avez une fonction membre **nbElements()** qui calcule et retourne le nombre d'éléments présents dans l'arbre.
Indication : vous pourrez créer une procédure récursive intermédiaire qui remplit le tableau à partir d'un nœud de l'arbre et son indice dans le tableau.

```
ElementA * convertirArbreEnTableau (const Arbre & a) {
   if (a.estVide()) return NULL;
   ElementA * tab = new ElementA[a.nbElements()];
   tab[0] = a.adRacine->info;
   remplirTabAPartirDeNoeud(a.adRacine,0,tab);
   return tab;
}

void remplirTabAPartirDeNoeud(Noeud * n, unsigned int i, ElementA * tab) {
   if (n->fg != NULL) {
      tab[2*i+1] = n->fg->info;
      remplirTabAPartirDeNoeud(n->fg,2*i+1,tab);
   }
   if (n->fd != NULL) {
      tab[2*i+2] = n->fd->info;
      remplirTabAPartirDeNoeud(n->fd,2*i+2,tab);
   }
}
```

Question 2.3 : Rappeler le code de la procédure membre **`insererElement`** de la classe **`Arbre`** faite en TP.

```
void Arbre::insererElement (ElementA e) {
    insererElementDepuisNoeud(e,adRacine);
}

void Arbre::insererElementDepuisNoeud(ElementA e, Noeud * & n) {
    if (n == NULL) {
        n = new Noeud;
        n->info = e;
        n->fg = NULL;
        n->fd = NULL;
    }
    else  {
        if (estInferieurElementA(e,n->info)) insererElementDepuisNoeud(e,n->fg);
        else if (estSuperieurElementA(e,n->info)) insererElementDepuisNoeud(e,n->fd);
    }
}
```

Question 2.4 : Ecrire la nouvelle procédure membre **`insererElement`** de la classe **`Arbre`** qui insère le nouvel élément dans un arbre compact utilisant la représentation en tableau statique. Vous supposerez que la classe **`Arbre`** a maintenant uniquement deux données membres : **`tab`** de type **`ElementA *`** et **`taille`** de type **`unsigned int`**. Quel est le principal inconvénient de cette méthode d'insertion par rapport à la précédente ?

```
void Arbre::insererElement (ElementA e) {
   // un nouvel élément s'insère forcément en fin de tableau
   // insérer e consiste donc à agrandir le tableau d'une case et y ajouter e
   ElementA * t = new ElementA[taille+1];
   for (unsigned int i=0; i < taille; i++)
      t[i] = tab[i];
   delete [] tab;
   t[taille] = e;
   taille++;
   tab = t;
}
```

Principal inconvénient de la nouvelle procédure d'insertion :
A chaque insertion d'un élément, il faut recopier tout l'ancien tableau, ce qui a un coût en $O(n)$.

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;
unsigned int nbElements () 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);
void supprimerCellule (Cellule * c);
```

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;
```