TP d’Algorithmique

n° 6

#### Table des matières

Exercice 1 2

Rappel de l’énoncé 2

Analyse du travail à faire 2

Algorithmes 4

Programmes 13

Jeux d’essai 22

# Exercice 1

## Rappel de l’énoncé

Il nous a été demandé de créer dans un premier temps un module d’arbre binaire avec ses fonctions usuelles, ainsi qu’une fonction de calcul de sa hauteur. Ensuite, avons eu un programme permettant la création d’un arbre donné à écrire. Et enfin, nous avons du écrire puis tester des procédures affichant l’arbre précédemment créé selon les 3 parcours en profondeur et le parcours en largeur.

## Analyse du travail à faire

Le module arbre utilise le module de liste chainée pour les fonctions de parcours. Il contient deux types : un type nœud représentant les nœuds des arbres binaires et un type abin, représentant un arbre.

Dans ce module, nous avons créé les fonctions d’usage.

*Créer un nouvel arbre*

On initialise à NULL un arbre.

*Désinitialiser un arbre*

Si l’arbre n’est pas vide, nous rappelons la fonctions de désinitialisation sur les sous arbres droit et gauche. Puis, nous initialisons ces sous arbres à NULL et enfin, détruisons l’arbre.

*Enraciner deux arbres à un nœud donné*

Nous créons un nouvel arbre dont l’étiquette et les sous arbres droit et gauche prennent les valeurs passée en paramètre. Ensuite, nous retournons cet arbre.

*Obtenir le sous arbre gauche*

Si l’arbre n’est pas vide, nous retournons son sous arbre gauche. S’il l’est, nous retournons NULL.

*Obtenir le sous arbre droit*

Si l’arbre n’est pas vide, nous retournons son sous arbre droit. S’il l’est, nous retournons NULL.

*Savoir si l’arbre est vide*

Nous retournons vrai si le pointeur de l’arbre pointe vers NULL. Sinon, nous retournons faux.

*Obtenir la racine de l’arbre*

Nous retournons le contenu du pointeur de l’arbre.

*Calculer la hauteur de l’arbre*

Nous créons et initialisons à 0 deux compteurs. Si l’arbre n’est pas vide, nous rappelons dans le premier compteur la hauteur du sous arbre gauche et dans le deuxième, celle du sous arbre droit. Enfin, nous retournons le compteur le plus grand + 1. Si l’arbre est vide, nous retournons 0.

*Parcours en profondeur préfixe*

Si l’arbre n’est pas vide, nous initialisons une liste chainée. Dans un nouveau maillon, nous stockons l’étiquette de la racine de l’arbre. Puis nous ajoutons ce nouveau maillon en queue de la liste. Ensuite, nous concaténons cette liste avec le parcours du sous arbre gauche. Nous répétons cette même opération avec le parcours du sous arbre droit. Enfin, nous retournons la liste chainée. Si l’arbre est vide, nous retournons NULL.

*Parcours en profondeur infixe*

Si l’arbre n’est pas vide, nous initialisons une liste chainée. Dans un nouveau maillon, nous stockons l’étiquette de la racine de l’arbre. Puis nous concaténons cette liste avec le parcours du sous arbre gauche. Ensuite, nous ajoutons le maillon de stockage en queue de la liste. Nous concaténons la liste avec le parcours du sous arbre droit. Enfin, nous retournons la liste chainée. Si l’arbre est vide, nous retournons NULL.

*Parcours en profondeur postfixe*

Si l’arbre n’est pas vide, nous initialisons une liste chainée. Dans un nouveau maillon, nous stockons l’étiquette de la racine de l’arbre. Puis nous concaténons cette liste avec le parcours du sous arbre gauche. Nous concaténons la liste avec le parcours du sous arbre droit. Ensuite, nous ajoutons le maillon de stockage en queue de la liste. Enfin, nous retournons la liste chainée. Si l’arbre est vide, nous retournons NULL.

*Parcours en largeur*

Si l’arbre n’est pas vide, nous initialisons une liste chainée. Nous stockons la hauteur de l’arbre dans une variable. Puis, nous parcourons l’arbre de toute sa hauteur en concaténant notre liste et la valeur des étiquettes du niveau. Ensuite, nous retournons la liste chainée. Si l’arbre est vide, nous retournons NULL.

*Obtenir la valeur des étiquettes à un niveau donné*

Nous initialisons une liste chainée. Si l’arbre n’est pas vide et que le niveau demandé est 0, Nous créons un nouveau maillon dans lequel nous stockons l’étiquette de la racine de l’arbre. Nous ajoutons ensuite ce maillon à la fin de notre liste et la retournons. Si le niveau demandé n’est pas 0, nous concaténons la liste avec les valeurs des étiquettes contenues au niveau du dessous sur les sous arbres droit et gauche. Nous retournons la liste. Si l’arbre est vide, nous retournons NULL.

Pour réaliser les dernières fonctions, nous avons dû créer un module de liste chainée avec ses fonctions usuelles pour manipuler la liste. Un liste est composée de maillons, eux-mêmes contiennent un caractère et un pointeur vers un autre maillon.

*Initialiser la liste*

Nous faisons pointer le pointeur de la liste vers NULL.

*Désinitialiser la liste*

Si la liste n’est pas vide, nous désinitialisons une sous liste qui commence au maillon suivant. Puis, nous détruisons le pointeur vers la liste initiale.

*Ajouter un maillon en queue de liste*

Nous plaçons un pointeur vers le premier maillon de la liste. Si la liste n’est pas vide, nous parcourons la liste avec ce pointeur jusqu’au dernier maillon. Nous le faisons pointer vers le nouveau maillon que nous faisons pointer vers NULL. Si la liste est vide, nous faisons pointer le pointeur de la liste chainée vers le nouveau maillon. Ce dernier pointera lui vers NULL.

*Concaténer deux listes*

Nous créons un pointeur pointant vers le début de la première liste. Si celle-ci n’est pas vide, alors nous le faisons la parcourir jusqu’à la fin. Ensuite, nous faisons pointer son dernier maillon vers la deuxième liste. Si la première liste est vide, nous la faisons directement pointer vers la deuxième.

*Supprimer un élément en queue de liste*

Nous créons deux pointeurs l’un pointant vers le premier maillon de la liste, l’autre vers NULL. Si la liste n’est pas vide, Nous la parcourons avec l’un des pointeurs jusqu’à la fin. L’autre s’arrête sur l’avant-dernier maillon. Nous détruisons le dernier maillon. Si le second pointeur pointe vers NULL alors nous initialisons le pointeur de la liste à NULL. S’il ne l’est pas, nous faisons pointer ce maillon vers NULL.

*Calculer la longueur d’une liste*

Nous créons un pointeur de maillon et un compteur que nous initialisons à 0. Nous parcourons la liste avec le maillon en incrémentant le compteur. Enfin, nous retournons le compteur.

*Obtenir la queue de la liste*

Nous retournons le contenu du pointeur vers la liste.

*Afficher la liste*

Nous créons un pointeur de maillon. Nous parcourons la liste avec ce pointeur en affichant son contenu pour chaque maillon par lequel il passe.

Nous avons ensuite écrit un programme permettant de construire l’arbre demandé. Ce même programme fera afficher la hauteur de l’arbre et les résultats des différents parcours.

Pour construire l’arbre, nous avons créé et initialisé trois arbres binaires. Dans chacun d’eux, nous avons enraciné les nœuds et les sous arbres voulus (s’il y en avait) en partant des nœuds les plus bas, vers la racine de l’arbre.

Nous affichons ensuite la hauteur de l’arbre.

Pour afficher les valeurs des nœuds de l’arbre selon les parcours, nous avons créé une liste chainée. Dans cette dernière, nous avons stocké les valeurs ordonnées après le parcours en profondeur préfixe. Puis nous avons affiché cette liste.

Nous avons ensuite désinitialisé la liste afin de lui affecter les nouvelles valeurs ordonnées après le parcours en profondeur infixe. Puis, nous l’avons de nouveau affichée.

Nous avons répété cette même opération pour les parcours en profondeur postfixe et en largeur.

## Algorithmes

**Module** arbre {

**Utilise** listeChainee

**Type** noeud = {

etiquette : car

ag, ad : pointeur noeud

}

**Type** abin = **Pointeur** noeud

**//Fonction permettant de créer un arbre**

**//Valeur retournée : un nouvel arbre**

**Procédure** arbNouv( ; a : abin)

**//Procédure permettant de désinitialiser un arbre**

**//Paramètre d'entrée : un arbre**

**//Paramètre de sortie : un arbre désinitialisé**

**//Pré-condition : l'arbre est initialisé**

**//Post-condition : l'arbre est vide et désinitialisé**

**Procédure** desinit( a : abin ; a : abin)

**//Fonction permettant d'enraciner deux arbres à un noeud donné**

**//Paramètres d'entrée : Deux arbres gauche et droit**

**// Un noeud**

**//Valeur retournée : un arbre composé du noeud donné comme racine et des arbres**

**// droit et gauche**

**Fonction** enracine( etiq : car, g : abin, d : abin) **Retourne** abin

**//Fonction retournant le sous-arbre gauche**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : le sous-arbre gauche**

**//Pré-condition : l'arbre est initialisé**

**Fonction** gauche( a : abin) **Retourne** abin

**//Fonction retournant le sous-arbre droit**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : le sous-arbre droit**

**//Pré-condition : l'arbre est initialisé**

**Fonction** droit( a : abin) **Retourne** abin

**//Fonction permettant de savoir si un arbre est vide**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : un booléen indiquant si l'arbre est vide**

**//Pré-condition : L'arbre est initialisé**

**Fonction** est\_vide( a : abin) **Retourne** **Booléen**

**//Fonction retournant la racine**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : la racine**

**//Pré-condition : l'arbre est initialisé et il a une racine**

**Fonction** racine( a : abin) **Retourne** noeud

**//Fonction permettant de calculer la hauteur**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : la hauteur de l'arbre**

**//Pré-condition : l'arbre est initiali**sé

**Fonction** hauteur( a : abin) **Retourne** **Entier**

**//Fonction parcourant l'arbre en profondeur (préfixe)**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : une liste chainée**

**//Pré-condition : l'arbre est initialisé**

**Fonction** profPrefixe( a : abin) **Retourne** **Pointeur** maillon

**//Fonction parcourant l'arbre en profondeur (infixe)**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : une liste chainée**

**//Pré-condition : l'arbre est initialisé**

**Fonction** profInfixe( a : abin) **Retourne** **Pointeur** maillon

**//Fonction parcourant l'arbre en profondeur (postfixe)**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : une liste chainée**

**//Pré-condition : l'arbre est initialisé**

**Fonction** profPostfixe( a : abin) **Retourne** **Pointeur** maillon

**//Fonction parcourant l'arbre en largeur**

**//Paramètre d'entrée : un arbre**

**//Valeur retournée : une liste chainée**

**//Pré-condition : l'arbre est initialisé**

**Fonction** largeur( a : abin) **Retourne** **Pointeur** maillon

**// Fonction retournant une liste chainée contenant les etiquettes au niveau donnée**

**// Paramètres d'entrée : un arbre et un niveau**

**// Valeur retournée : une liste chainée contenant les étiquettes**

**// Pré-condition : l'arbre est initialisé, le niveau est >= 0**

**Fonction** valeursAuNiveau(a : abin, niveau : **Entier**) **Retourne** **Pointeur** maillon

} **Fin** module arbre

**Procédure** arbNouv( ; a : abin)

**Début**

a <- **NULL**

**Fin** arbNouv

**Procédure** desinit( a : abin ; a : abin)

**Début**

**Si** a ≠ **NULL** **Alors**

desinit((**Contenu** a).ag)

desinit((**Contenu** a).ad)

(**Contenu** a).ag <- **NULL**

(**Contenu** a).ad <- **NULL**

**Détruire** a

**Fin** **Si**

**Fin** desinit

**Fonction** enracine( etiq : car, g : abin, d : abin) **Retourne** abin

**Début**

**Avec** nd : abin

nd <- **Nouveau** noeud

(**Contenu** nd).etiquette <- etiq

(**Contenu** nd).ag <- g

(**Contenu** nd).ad <- d

**Retourne** nd

**Fin** enracine

**Fonction** gauche( a : abin) **Retourne** abin

**Début**

**Si** a ≠ **NULL** **Alors**

**Retourne** (**Contenu** a).ag

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** gauche

**Fonction** droit( a : abin) **Retourne** abin

**Début**

**Si** a ≠ **NULL** **Alors**

**Retourne** (**Contenu** a).ad

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** droit

**Fonction** est\_vide( a : abin) **Retourne** **Booléen**

**Début**

**Retourne** (a = **NULL**)

**Fin** est\_vide

**Fonction** racine( a : abin) **Retourne** noeud

**Début**

**Retourne** (**Contenu** a)

**Fin** racine

**Fonction** hauteur( a : abin) **Retourne** **Entier**

**Début**

**Avec** gauche : **Entier**

droit : **Entier**

gauche <- 0

droit <- 0

**Si** a ≠ **NULL** **Alors**

gauche <- hauteur((**Contenu** a).ag)

droit <- hauteur((**Contenu** a).ad)

**Retourne** (1 + (gauche > droite) ? gauche:droite)

**Sinon**

**Retourne** 0

**Fin** **Si**

**Fin** hauteur

**Fonction** profPrefixe( a : abin) **Retourne** **Pointeur** maillon

**Début**

**Avec** np : **Pointeur** maillon

m\_etiq : **Pointeur** maillon

**Si** a ≠ **NULL** **Alors**

initListe(np)

m\_etiq <- **Nouveau** maillon

(**Contenu** m\_etiq).contenu <- (**Contenu** a).etiquette

ajoutEnQueue(np, m\_etiq)

concatListe(np, profPrefixe((**Contenu** a).ag))

concatListe(np, profPrefixe((**Contenu** a).ad))

**Retourne** np

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** profPrefixe

**Fonction** profInfixe( a : abin) **Retourne** **Pointeur** maillon

**Début**

**Avec** np : **Pointeur** maillon

m\_etiq : **Pointeur** maillon

**Si** a ≠ **NULL** **Alors**

initListe(np)

m\_etiq <- **Nouveau** maillon

(**Contenu** m\_etiq).contenu <- (**Contenu** a).etiquette

concatListe(np, profInfixe((**Contenu** a).ag))

ajoutEnQueue(np, m\_etiq)

concatListe(np, profInfixe((**Contenu** a).ad))

**Retourne** np

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** profInfixe

**Fonction** profPostfixe( a : abin) **Retourne** **Pointeur** maillon

**Début**

**Avec** np : **Pointeur** maillon

m\_etiq : **Pointeur** maillon

**Si** a ≠ **NULL** **Alors**

initListe(np)

m\_etiq <- **Nouveau** maillon

(**Contenu** m\_etiq).contenu <- (**Contenu** a).etiquette

concatListe(np, profPostfixe((**Contenu** a).ag))

concatListe(np, profPostfixe((**Contenu** a).ad))

ajoutEnQueue(np, m\_etiq)

**Retourne** np

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** profPostfixe

**Fonction** largeur( a : abin) **Retourne** **Pointeur** maillon

**Début**

**Avec** np : **Pointeur** maillon

hauteurArbre : **Entier**

i : **Entier**

**Si** a != **NULL** **Alors**

initListe(np)

hauteurArbre <- hauteur(a)

**Pour** i de 0 à hauteurArbre - 1 **Faire**

concatListe(np, valeursAuNiveau(a, i))

**Fin** **Faire**

**Retourne** np

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** largeur

**Fonction** valeursAuNiveau(a : abin, niveau : **Entier**) **Retourne** **Pointeur** maillon

**Début**

**Avec** m\_etiq : **Pointeur** maillon

np : **Pointeur** maillon

initListe(np)

**Si** a != **NULL** **Alors**

**Si** niveau = 0 **Alors**

m\_etiq <- **Nouveau** maillon

(**Contenu** m\_etiq).contenu <- (**Contenu** a).etiquette

ajoutEnQueue(np, m\_etiq)

**Retourne** np

**Sinon**

concatListe(np, valeursAuNiveau((**Contenu** a).ag, niveau - 1))

concatListe(np, valeursAuNiveau((**Contenu** a).ad, niveau - 1))

**Retourne** np

**Fin** **Si**

**Sinon**

**Retourne** **NULL**

**Fin** **Si**

**Fin** valeursAuNiveau

**Module** listeChainee

**Type** maillon = {

contenu : car

suivant : **Pointeur** maillon

}

**// Procédure permettant d'initialiser une liste chaînée**

**// Paramètre de sortie : une liste**

**// Post-condition : la liste est initialisée**

**Procédure** initListe(; p : **Pointeur** maillon)

**// Procédure permettant de désinitialiser une liste chaînée**

**// Paramètre d'entrée : la liste à désinitialiser**

**// Paramètre de sortie : la liste désinitialisée**

**// Pré-condition : la liste est initialisée**

**// Post-condition : la liste est désinitialisée**

**Procédure** desinitListe(p : **Pointeur** maillon ; p : **Pointeur** maillon)

**// Procédure permettant d'ajouter un élément en queue de liste**

**// Paramètres d'entrée : une liste et un maillon**

**// Paramètres de sortie : la liste ayant le maillon en queue**

**// Pré-condition : la liste est initialisée et l'élément n'est pas nul**

**// Post-condition : la liste possède l'élément passé en paramètre en queue**

**Procédure** ajoutEnQueue(p : **Pointeur** maillon, nouv : **Pointeur** maillon ; p : **Pointeur** maillon)

**//Procédure permettant de concaténer deux listes**

**//Paramètre d'entrée : Deux listes chainées**

**//Paramètre de sortie : une liste chainée composée des deux précédentes**

**//Pré-condition : les listes sont initialisées**

**//Post-condition : la deuxième liste est ajoutée à la fin de la première**

**Procédure** concatListe( l1 : Pointeur maillon, l2 : **Pointeur** maillon ; l1 : **Pointeur** maillon)

**// Procédure permettant de supprimer l'élément en queue de liste**

**// Paramètre d'entrée : la liste**

**// Paramètre de sortie : la liste avec un élément en moins**

**// Pré-condition : la liste est initialisée**

**// Post-condition : la liste a perdu son dernier élément**

**Procédure** supprQueue(p : **Pointeur** maillon ; p : **Pointeur** maillon)

**// Fonction permettant de connaître la longueur de la liste**

**// Paramètre d'entrée : la liste**

**// Valeur retournée : la longueur de la liste**

**// Pré-condition : la liste est initialisée**

**Fonction** longueurListe(p : **Pointeur** maillon) **Retourne** **Entier**

**// Fonction permettant d'observer la queue de la liste**

**// Paramètre d'entrée : la liste**

**// Valeur retournée : la queue de la liste**

**// Pré-condition : la liste est initialisée et possède au moins 1 élément**

**Fonction** queueListe(p : **Pointeur** maillon) **Retourne** maillon

**// Procédure permettant d'afficher la liste**

**// Paramètre d'entrée : une liste**

**// Pré-condition : la liste est initialisée**

**Procédure** afficherListe(p : **Pointeur** maillon)

} **Fin** module listeChainee

**Procédure** initListe(; p : **Pointeur** maillon)

**Début**

p <- **NULL**

**Fin** initListe

**Procédure** desinitListe(p : **Pointeur** maillon ; p : **Pointeur** maillon)

**Début**

**Si** p ≠ **NULL** **Alors**

desinitListe((**Contenu** p).suivant)

**Détruire** p

**Fin** **Si**

**Fin** desinitListe

**Procédure** ajoutEnQueue(p : **Pointeur** maillon, nouv : **Pointeur** maillon ; p : **Pointeur** maillon)

**Début**

**Avec** tmp : **Pointeur** maillon

tmp <- p

**Si** tmp ≠ **NULL** **Alors**

**Tant** **Que** (**Contenu** tmp).suivant ≠ **NULL** **Faire**

tmp <- (Contenu tmp).suivant

**Fin** **Faire**

(**Contenu** tmp).suivant <- nouv

(**Contenu** nouv).suivant <- **NULL**

**Sinon**

p <- nouv

(**Contenu** p).suivant <- **NULL**

**Fin** **Si**

**Fin** ajoutEnQueue

**Procédure** concatListe( l1 : Pointeur maillon, l2 : **Pointeur** maillon ; l1 : **Pointeur** maillon)

**Début**

**Avec** tmp : **Pointeur** maillon

tmp <- l1

**Si** tmp ≠ **NULL** **Alors**

**Tant** **Que** (**Contenu** tmp).suivant ≠ **NULL** **Faire**

tmp <- (**Contenu** tmp).suivant

**Fin** **Faire**

(**Contenu** tmp).suivant <- l2

**Sinon**

l1 <- l2

**Fin** **Si**

**Fin** concatListe

**Procédure** supprQueue(p : **Pointeur** maillon ; p : **Pointeur** maillon)

**Début**

**Avec** tmp : **Pointeur** maillon

avantdernier : **Pointeur** maillon

tmp <- p

avantdernier <- **NULL**

**Si** tmp ≠ **NULL** **Alors**

**Tant** **Que** (**Contenu** tmp).suivant ≠ **NULL** **Faire**

avantdernier <- tmp

tmp <- (**Contenu** tmp).suivant

**Fin** **Faire**

**Détruire** tmp

**Si** avantdernier ≠ **NULL** **Alors**

(**Contenu** avantdernier).suivant <- **NULL**

**Sinon**

p <- **NULL**

**Fin** **Si**

**Fin** **Si**

**Fin** supprQueue

**Fonction** longueurListe(p : **Pointeur** maillon) **Retourne** **Entier**

**Début**

**Avec** n : **Entier**

tmp : **Pointeur** maillon

n <- 0

tmp <- p

**Tant** **Que** tmp ≠ **NULL** **Faire**

tmp <- (**Contenu** tmp).suivant

n <- n+1

**Fin** **Faire**

**Retourne** n

**Fin** longueurListe

**Fonction** queueListe(p : **Pointeur** maillon) **Retourne** maillon

**Début**

**Retourne** (**Contenu** p)

**Fin** queueListe

**Procédure** afficherListe(p : **Pointeur** maillon)

**Début**

**Avec** tmp : **Pointeur** maillon

tmp <- p

**Tant** **Que** tmp != **NULL** **Faire**

**Afficher** (**Contenu** tmp).contenu

tmp <- (**Contenu** tmp).suivant

**Fin** **Faire**

**Fin** afficherListe

**Programme** principal

**Avec** a1, a2, a3 : abin

listeChainee : **Pointeur** maillon

a1 <- arbNouv()

a2 <- arbNouv()

a3 <- arbNouv()

a1 <- enracine('q', **NULL**, **NULL**)

a2 <- enracine('b', **NULL**, **NULL**)

a3 <- enracine('p', a1, a2)

a1 <- enracine('u', **NULL**, **NULL**)

a2 <- enracine('t', a1, a3)

a1 <- enracine('t', **NULL**, **NULL**)

a1 <- enracine('s', a1, **NULL**)

a3 <- enracine('v', **NULL**, **NULL**)

a3 <- enracine('r', a3, a1)

a1 <- enracine('a', a3, a2)

**Afficher** "Hauteur de l'arbre : "

**Afficher** hauteur(a1)

**Afficher** "Parcours en profondeur préfixe : "

listeChainee <- profPrefixe(a1)

afficherListe(listeChainee)

**Afficher** "Parcours en profondeur infixe : "

desinitListe(listeChainee)

listeChainee <- profInfixe(a1)

afficherListe(listeChainee)

**Afficher** "Parcours en profondeur postfixe : "

desinitListe(listeChainee)

listeChainee <- profPostfixe(a1)

afficherListe(listeChainee)

**Afficher** "Parcours en largeur : "

desinitListe(listeChainee)

listeChainee <- largeur(a1)

afficherListe(a1)

**Fin** principal

## Programmes

* Module arbre

#ifndef \_\_TP06\_\_arbre\_\_

#define \_\_TP06\_\_arbre\_\_

#include <iostream>

#include "listeChainee.h"

struct noeud {

char etiquette;

struct noeud \*ag, \*ad;

};

typedef struct noeud\* abin;

//Fonction permettant de créer un arbre

//Valeur retournée : un nouvel arbre

abin arbNouv();

//Procédure permettant de désinitialiser un arbre

//Paramètre d'entrée : un arbre

//Paramètre de sortie : un arbre désinitialisé

//Pré-condition : l'arbre est initialisé

//Post-condition : l'arbre est vide et désinitialisé

void desinit(abin \*a);

//Fonction permettant d'enraciner deux arbres à un noeud donné

//Paramètres d'entrée : Deux arbres gauche et droit

// Un noeud

//Valeur retournée : un arbre composé du noeud donné comme racine et des arbres

// droit et gauche

abin enracine(char etiq, abin g, abin d);

//Fonction retournant le sous-arbre gauche

//Paramètre d'entrée : un arbre

//Valeur retournée : le sous-arbre gauche

//Pré-condition : l'arbre est initialisé

abin gauche(abin a);

//Fonction retournant le sous-arbre droit

//Paramètre d'entrée : un arbre

//Valeur retournée : le sous-arbre droit

//Pré-condition : l'arbre est initialisé

abin droite(abin a);

//Fonction permettant de savoir si un arbre est vide

//Paramètre d'entrée : un arbre

//Valeur retournée : un booléen indiquant si l'arbre est vide

//Pré-condition : L'arbre est initialisé

bool est\_vide(abin a);

//Fonction retournant la racine

//Paramètre d'entrée : un arbre

//Valeur retournée : la racine

//Pré-condition : l'arbre est initialisé et il a une racine

struct noeud racine(abin a);

//Fonction permettant de calculer la hauteur

//Paramètre d'entrée : un arbre

//Valeur retournée : la hauteur de l'arbre

//Pré-condition : l'arbre est initialisé

int hauteur(abin a);

//Fonction parcourant l'arbre en profondeur (préfixe)

//Paramètre d'entrée : un arbre

//Valeur retournée : une liste chainée

//Pré-condition : l'arbre est initialisé

struct maillon\* profPrefixe(abin a);

//Fonction parcourant l'arbre en profondeur (infixe)

//Paramètre d'entrée : un arbre

//Valeur retournée : une liste chainée

//Pré-condition : l'arbre est initialisé

struct maillon\* profInfixe(abin a);

//Fonction parcourant l'arbre en profondeur (postfixe)

//Paramètre d'entrée : un arbre

//Valeur retournée : une liste chainée

//Pré-condition : l'arbre est initialisé

struct maillon\* profPostfixe(abin a);

//Fonction parcourant l'arbre en largeur

//Paramètre d'entrée : un arbre

//Valeur retournée : une liste chainée

//Pré-condition : l'arbre est initialisé

struct maillon\* largeur(abin a);

// Fonction retournant une liste chainée contenant les etiquettes au niveau donnée

// Paramètres d'entrée : un arbre et un niveau

// Valeur retournée : une liste chainée contenant les étiquettes

// Pré-condition : l'arbre est initialisé, le niveau est >= 0

struct maillon\* valeursAuNiveau(abin a, int niveau);

#endif /\* defined(\_\_TP06\_\_arbre\_\_) \*/

* Implémentation des fonctions du module arbre

#include "arbre.h"

#include "listeChainee.h"

abin arbNouv()

{

return NULL;

}

void desinit(abin \*a)

{

if ( (\*a) != NULL) {

desinit(&(\*(\*a)).ag);

desinit(&(\*(\*a)).ad);

(\*(\*a)).ag = NULL;

(\*(\*a)).ad = NULL;

delete (\*a);

}

}

abin enracine(char etiq, abin g, abin d)

{

abin nd;

nd = new struct noeud;

(\*nd).etiquette = etiq;

(\*nd).ag = g;

(\*nd).ad = d;

return nd;

}

abin gauche(abin a)

{

if (a != NULL)

return (\*a).ag;

else

return NULL;

}

abin droite(abin a)

{

if (a != NULL)

return (\*a).ad;

else

return NULL;

}

bool est\_vide(abin a)

{

return (a == NULL);

}

struct noeud racine(abin a)

{

return (\*a);

}

int hauteur(abin a)

{

int gauche, droite;

gauche = 0;

droite = 0;

if (a != NULL) {

gauche = hauteur((\*a).ag);

droite = hauteur((\*a).ad);

return (1 + ((gauche > droite) ? gauche : droite));

}

else

return 0;

}

struct maillon\* profPrefixe(abin a)

{

struct maillon \*np, \*m\_etiq;

if (a != NULL) {

initListe(&np);

m\_etiq = new struct maillon;

(\*m\_etiq).contenu = (\*a).etiquette;

ajoutEnQueue(&np, m\_etiq);

concatListe(&np, profPrefixe((\*a).ag));

concatListe(&np, profPrefixe((\*a).ad));

return np;

}

else

return NULL;

}

struct maillon\* profInfixe(abin a)

{

struct maillon \*np, \*m\_etiq;

if (a != NULL) {

initListe(&np);

m\_etiq = new struct maillon;

(\*m\_etiq).contenu = (\*a).etiquette;

concatListe(&np, profInfixe((\*a).ag));

ajoutEnQueue(&np, m\_etiq);

concatListe(&np, profInfixe((\*a).ad));

return np;

}

else

return NULL;

}

struct maillon\* profPostfixe(abin a)

{

struct maillon \*np, \*m\_etiq;

if (a != NULL) {

initListe(&np);

m\_etiq = new struct maillon;

(\*m\_etiq).contenu = (\*a).etiquette;

concatListe(&np, profPostfixe((\*a).ag));

concatListe(&np, profPostfixe((\*a).ad));

ajoutEnQueue(&np, m\_etiq);

return np;

}

else

return NULL;

}

struct maillon \*largeur(abin a)

{

struct maillon \*np;

int hauteurArbre, i;

if (a != NULL) {

initListe(&np);

hauteurArbre = hauteur(a);

for (i = 0; i < hauteurArbre; i++) {

concatListe(&np, valeursAuNiveau(a, i));

}

return np;

}

else

return NULL;

}

struct maillon\* valeursAuNiveau(abin a, int niveau)

{

struct maillon \*np, \*m\_etiq;

initListe(&np);

if (a != NULL) {

if (niveau == 0) {

m\_etiq = new struct maillon;

(\*m\_etiq).contenu = (\*a).etiquette;

ajoutEnQueue(&np, m\_etiq);

}

else {

concatListe(&np, valeursAuNiveau((\*a).ag, niveau - 1));

concatListe(&np, valeursAuNiveau((\*a).ad, niveau - 1));

}

}

return np;

}

* Module listeChainee

#ifndef LISTECHAINEE\_H

#define LISTECHAINEE\_H

#include <iostream>

struct maillon {

char contenu;

struct maillon\* suivant;

};

// Procédure permettant d'initialiser une liste chaînée

// Paramètre de sortie : une liste

// Post-condition : la liste est initialisée

void initListe(struct maillon \*\*p);

// Procédure permettant de désinitialiser une liste chaînée

// Paramètre d'entrée : la liste à désinitialiser

// Paramètre de sortie : la liste désinitialisée

// Pré-condition : la liste est initialisée

// Post-condition : la liste est désinitialisée

void desinitListe(struct maillon \*\*p);

// Procédure permettant d'ajouter un élément en queue de liste

// Paramètres d'entrée : une liste et un maillon

// Paramètres de sortie : la liste ayant le maillon en queue

// Pré-condition : la liste est initialisée et l'élément n'est pas nul

// Post-condition : la liste possède l'élément passé en paramètre en queue

void ajoutEnQueue(struct maillon \*\*p, struct maillon \*nouv);

//Procédure permettant de concaténer deux listes

//Paramètre d'entrée : Deux listes chainées

//Paramètre de sortie : une liste chainée composée des deux précédentes

//Pré-condition : les listes sont initialisées

//Post-condition : la deuxième liste est ajoutée à la fin de la première

void concatListe(struct maillon \*\* l1, struct maillon \*l2);

// Procédure permettant de supprimer l'élément en queue de liste

// Paramètre d'entrée : la liste

// Paramètre de sortie : la liste avec un élément en moins

// Pré-condition : la liste est initialisée

// Post-condition : la liste a perdu son dernier élément

void supprQueue(struct maillon \*\*p);

// Fonction permettant de connaître la longueur de la liste

// Paramètre d'entrée : la liste

// Valeur retournée : la longueur de la liste

// Pré-condition : la liste est initialisée

int longueurListe(struct maillon \*p);

// Fonction permettant d'observer la queue de la liste

// Paramètre d'entrée : la liste

// Valeur retournée : la queue de la liste

// Pré-condition : la liste est initialisée et possède au moins 1 élément

struct maillon queueListe(struct maillon \*p);

// Procédure permettant d'afficher la liste

// Paramètre d'entrée : une liste

// Pré-condition : la liste est initialisée

void afficherListe(struct maillon \*p);

#endif /\* LISTECHAINEE\_H \*/

* Implémentation des fonctions du module listeChainee

#include "listeChainee.h"

void initListe(struct maillon \*\*p)

{

(\*p) = NULL;

}

void desinitListe(struct maillon \*\*pp)

{

if ((\*pp) != NULL) {

desinitListe(&(\*(\*pp)).suivant);

delete (\*pp);

}

}

void ajoutEnQueue(struct maillon \*\*pp, struct maillon \*nouv)

{

struct maillon \*tmp;

tmp = (\*pp);

if (tmp != NULL) {

while ((\*tmp).suivant != NULL) {

tmp = (\*tmp).suivant;

}

(\*tmp).suivant = nouv;

(\*nouv).suivant = NULL;

}

else {

(\*pp) = nouv;

(\*(\*pp)).suivant = NULL;

}

}

void concatListe(struct maillon \*\*l1, struct maillon \*l2)

{

struct maillon \*tmp;

tmp = (\*l1);

if (tmp != NULL) {

while ( (\*tmp).suivant != NULL) {

tmp = (\*tmp).suivant;

}

(\*tmp).suivant = l2;

}

else {

(\*l1) = l2;

}

}

void supprQueue(struct maillon \*\*pp)

{

struct maillon \*tmp, \*avantdernier;

tmp = (\*pp);

avantdernier = NULL;

if (tmp != NULL) {

while ((\*tmp).suivant != NULL) {

avantdernier = tmp;

tmp = (\*tmp).suivant;

}

delete tmp;

if (avantdernier != NULL) {

(\*avantdernier).suivant = NULL;

}

else {

(\*pp) = NULL;

}

}

}

int longueurListe(struct maillon \*p)

{

int n;

struct maillon \*tmp;

n = 0;

tmp = p;

while (tmp != NULL) {

tmp = (\*tmp).suivant;

n++;

}

return n;

}

struct maillon queueListe(struct maillon \*p)

{

return (\*p);

}

void afficherListe(struct maillon \*p)

{

struct maillon \*tmp;

tmp = p;

while (tmp != NULL) {

std::cout << (\*tmp).contenu << " ";

tmp = (\*tmp).suivant;

}

}

* Programme principal

#include <iostream>

#include "arbre.h"

#include "listeChainee.h"

using namespace std;

int main(int argc, const char \* argv[]) {

abin a1, a2, a3;

struct maillon\* listeChainee;

a1 = arbNouv();

a2 = arbNouv();

a3 = arbNouv();

a1 = enracine('q', NULL, NULL);

a2 = enracine('b', NULL, NULL);

a3 = enracine('p', a1, a2);

a1 = enracine('u', NULL, NULL);

a2 = enracine('t', a1, a3);

a1 = enracine('t', NULL, NULL);

a1 = enracine('s', a1, NULL);

a3 = enracine('v', NULL, NULL);

a3 = enracine('r', a3, a1);

a1 = enracine('a', a3, a2);

cout << "Hauteur de l'arbre : " << hauteur(a1) << endl;

cout << "Parcours en profondeur préfixe : " << endl;

listeChainee = profPrefixe(a1);

afficherListe(listeChainee);

cout << endl << "Parcours en profondeur infixe : " << endl;

listeChainee = profInfixe(a1);

afficherListe(listeChainee);

cout << endl << "Parcours en profondeur postfixe : " << endl;

listeChainee = profPostfixe(a1);

afficherListe(listeChainee);

cout << endl << "Parcours en largeur : " << endl;

listeChainee = largeur(a1);

afficherListe(listeChainee);

cout << endl;

return 0;

}

## Jeux d’essai

|  |  |  |
| --- | --- | --- |
| **Action testée** | **Résultat attendu** | **Résultat du programme** |
| Parcours en longueur préfixe | a r v s t t u p q b | a r v s t t u p q b |
| Parcours en longueur infixe | v r t s a u t q p b | v r t s a u t q p b |
| Parcours en longueur postfixe | v t s r u q b p t a | v t s r u q b p t a |
| Parcours en largeur | a r t v s u p t q b | a r t v s u p t q b |
| Hauteur de l’arbre | 4 | 4 |

