TP d’Algorithmique

n°4

Gestion d’une file d’attente (suite) 2

Rappel de l’énoncé 2

Module Pile 2

Analyse du travail à faire 2

Algorithme 3

Programme 5

Contenu de pile.h 7

Modification du module FileAttente 8

Analyse du travail 8

Algorithme 8

Programme 9

Algorithme principal 10

Analyse du travail à faire 10

Algorithme 10

Programme 11

# Gestion d’une file d’attente (suite)

## Rappel de l’énoncé

Ajouter une option « Undo » à la liste de choix pour gérer une fille d’attente. Cette option permet d’annuler les dernières modifications effectuées sur la file d’attente.

# Module Pile

## Analyse du travail à faire

On réalise un module Pile à partir des structures de données Pile et CompoPile et des fonctions et procédures pour manipuler les piles.

*Initialiser une Pile :*

Initialiser le pointeur tête de la pile à null.

*Désinitialiser une Pile :*

Retirer tous les éléments de la pile, puis initialiser sa tête à null.

*Ajouter un élément à une pile :*

Nous créons un pointeur temporaire dans lequel on stocke la tête de la pile. Nous ajoutons un nouvel élément sur le pointeur tête et lions cet élément avec l’ancienne tête de pile.

*Retirer un élément à une pile :*

On stocke le deuxième élément de la pile dans un pointeur temporaire puis supprimons la tête. Enfin, relions le nouveau premier élément au pointeur de tête.

*Obtenir le contenu de la tête d’une pile :*

Nous retournons le contenu du pointeur vers la tête de la pile.

*Calculer la hauteur de la pile :*

Nous parcourons la pile en incrémentant un compteur et affichons ainsi la hauteur de la pile.

## Algorithme

**Module** Pile

**Type** Pile = {

tete : **Pointeur** CompoPile

}

**Type** CompoPile = {

contenu : FileAttente

suivant : **Pointeur** CompoPile

}

//Procédure permettant d'initialiser une pile.

//Paramètres de sortie : un pile initialisée

//Post-condition : la pile est initialisée

**Procédure** initPile(; p : Pile)

**Début**

p.tete <- **NULL**

**Fin** initPile

//Procédure permettant de désinitialiser une pile.

//Paramètres d'entrée : une pile

//Paramètres de sortie : une pile

//Pré-condition : la pile est initialisée

//Post-condition : la pile est vide et désinitialisée

**Procédure** desinitPile(p : Pile ; p :Pile)

**Début**

**Tant** **Que** p.tete ≠ **NULL** **Faire**

depiler(p)

**Fin** **Faire**

p.tete <- **NULL**

**Fin** desinitPile

//Procédure permettant d'empiler un élément sur la pile

//Paramètres Entrée : une pile et un élément de pile à ajouter

//Paramètre de sortie : la pile ayant l'élément en tête

//Pré-condition : la pile est initialisée

//Post-condition : la pile possède l'élément ajouté en tête

**Procédure** empiler(p : pile, cp : **Pointeur** CompoPile ; p : pile)

**Début**

**Avec** tmp : **Pointeur** CompoPile

tmp <- p.tete

p.tete <- cp

**Si** p.tete != **NULL** **Alors**

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

**Fin** **Si**

**Fin** empiler

//Procédure permettant de dépiler un élément sur la pile

//Paramètres Entrée : une pile

//Paramètre de sortie : la pile ayant l'élément en tête supprimé

//Pré-condition : la pile est initialisée et non vide

//Post-condition : la pile avec l'élément de tête en moins

**Procédure** depiler(p : Pile ; p : Pile)

**Début**

**Avec** tmp : **Pointeur** CompoPile

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

**Détruire** p.tete

p.tete <- tmp

**Fin** depiler

//Fonction permettant d'obtenir la tête de la pile

//Paramètres Entrée : une pile

//Valeur retournée : Le composant de pile (CompoPile) en tête

//Pré-condition : la pile est initialisée et non-vide

**Fonction** obtenirTete( p:Pile) **Retourne** CompoPile

**Début**

**Retourne** **Contenu** p.tete

**Fin** obtenirTete

//Fonction permettant d'obtenir la hauteur de la pile

//Paramètres d'entrée : un pile

//Valeur retournée : la hauteur de la pile

//Pré-condition : La pile est initialisée

**Fonction** hauteur( p : Pile) **Retourne** **Entier**

**Début**

**Avec** tmp : **Pointeur** CompoPile

cpt : **Entier**

cpt <- 0

tmp <- p.tete

**Tant** **Que** (tmp ≠ **NULL**) **Faire**

n <- n + 1

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

**Fin** **Faire**

**Retourne** cpt

**Fin** hauteur

**Fin** Pile

## Programme

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 1 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37  38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69  70 71 72 73 74 75 76 77 78 79 80 81 82 83 | //  // pile.cpp  // TP03  //  // Created by Oncle D. on 10/10/2014.  // Copyright (c) 2014 Oncle D. All rights reserved.  //  #include "pile.h"  //Procédure permettant d'initialiser une pile.  //Paramètres de sortie : un pile initialisée  //Post-condition : la pile est initialisée  void initPile(struct Pile &p)  {  p.tete = NULL;  }  //Procédure permettant de désinitialiser une pile.  //Paramètres d'entrée : une pile  //Paramètres de sortie : une pile  //Pré-condition : la pile est initialisée  //Post-condition : la pile est vide et désinitialisée  void desinitPile(struct Pile &p)  {  while (p.tete != NULL)  {  depiler(p);  }  p.tete = NULL;  }  //Procédure permettant d'empiler un élément sur la pile  //Paramètres Entrée : une pile et un élément de pile à ajouter  //Paramètre de sortie : la pile ayant l'élément en tête  //Pré-condition : la pile est initialisée  //Post-condition : la pile possède l'élément ajouté en tête  void empiler(struct Pile &p, struct CompoPile \*cp)  {  struct CompoPile \*tmp;  tmp = p.tete;  p.tete = cp;  if (p.tete == NULL)  (\*p.tete).suivant = tmp;  }  //Procédure permettant de dépiler un élément sur la pile  //Paramètres Entrée : une pile  //Paramètre de sortie : la pile ayant l'élément en tête supprimé  //Pré-condition : la pile est initialisée et non vide  //Post-condition : la pile avec l'élément de tête en moins  void depiler(struct Pile &p)  {  struct CompoPile \*tmp;  tmp = (\*p.tete).suivant;  delete p.tete;  p.tete = tmp;  }  //Fonction permettant d'obtenir la tête de la pile  //Paramètres Entrée : une pile  //Valeur retournée : Le composant de pile (CompoPile) en tête  //Pré-condition : la pile est initialisée et non-vide  struct CompoPile obtenirTete(const struct Pile p)  {  return \*p.tete;  }  //Fonction permettant d'obtenir la hauteur de la pile  //Paramètres d'entrée : un pile  //Valeur retournée : la hauteur de la pile  //Pré-condition : La pile est initialisée  int hauteur(const struct Pile p)  {  struct CompoPile \*tmp;  int cpt = 0;  tmp = p.tete;  while (tmp != NULL) {  cpt = cpt + 1;  tmp = (\*tmp).suivant;  }  return cpt;  } |

## Contenu de pile.h

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | //  // pile.h  // TP03  //  // Created by Oncle D. on 10/10/2014.  // Copyright (c) 2014 Oncle D. All rights reserved.  //  #ifndef \_\_TP03\_\_pile\_\_  #define \_\_TP03\_\_pile\_\_  #include "file.h"  struct Pile  {  struct CompoPile \*tete;  };  struct CompoPile  {  FileAttente contenu;  struct CompoPile \*suivant;  };  void initPile( struct Pile &p);  void desinitPile(struct Pile &p);  void empiler(struct Pile &p, struct CompoPile \*cp);  void depiler(struct Pile &p);  struct CompoPile obtenirTete(const struct Pile p);  int hauteur(const struct Pile p);  #endif /\* defined(\_\_TP03\_\_pile\_\_) \*/ |

# Modification du module FileAttente

## Analyse du travail

Nous avions besoin, afin de gérer l’annulation des actions sur une file d’attente, de pouvoir la copier. Nous avons donc créé une fonction de duplication d’une file d’attente.

Si la file à copier n’est pas nulle, on la parcourt avec un pointeur, à chaque maillon, un autre pointeur recrée ce même maillon.

## Algorithme

//Fonction permettant de dupliquer une file

//Paramètres d'entrée : une file

//Valeur retournée : la même file

//Pré-condition : la file est initialisée

**Fonction** dupFile( f : FileAttente) **Retourne** FileAttente

**Début**

**Avec** nf : FileAttente

p\_original : **Pointeur** Maillon

p\_nouveau : **Pointeur** Maillon

**Si** f.tete ≠ **NULL** **Faire**

p\_original <- f.tete

p\_nouveau <- **Nouveau** Maillon

nf.tete <- p\_nouveau

**Tant** **Que** p\_original ≠ **NULL** **Faire**

(**Contenu** p\_nouveau).nom <- (**Contenu** p\_original).nom

(**Contenu** p\_nouveau).numero <- (**Contenu** p\_original).numero

p\_original <- (**Contenu** p\_original).suivant

**Si** p\_original ≠ **NULL** **Alors**

(**Contenu** p\_nouveau).suivant <- **Nouveau** Maillon

(**Contenu** p\_nouveau) <- (**Contenu** p\_nouveau).suivant

**Sinon**

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

**Fin** **Si**

**Fin** **Faire**

nf.fin <- p\_nouveau

**Sinon**

nf.tete <- **NULL**

nf.queue <- **NULL**

**Fin** **Si**

**Retourne** nf

**Fin** dupFile

## Programme

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7  8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35  36 37  38 | //Fonction permettant de dupliquer une file  //Paramètres d'entrée : une file  //Valeur retournée : la même file  //Pré-condition : la file est initialisée  struct FileAttente dupFile (const struct FileAttente f)  {  struct FileAttente nf;  struct maillon \*p\_original;  struct maillon \*p\_nouveau;    if (f.debut != NULL)  {  p\_original = f.debut;  p\_nouveau = new maillon;  nf.debut = p\_nouveau;  while (p\_original != NULL)  {  //(\*p\_nouveau).nom = (\*p\_original).nom;  strcpy((\*p\_nouveau).nom, (\*p\_original).nom);  (\*p\_nouveau).numero = (\*p\_original).numero;  p\_original = (\*p\_original).suivant;  if (p\_original != NULL)  {  (\*p\_nouveau).suivant = new maillon;  p\_nouveau = (\*p\_nouveau).suivant;  }  else  (\*p\_nouveau).suivant = NULL;  }  nf.fin = p\_nouveau;  }  else  {  nf.fin = NULL;  nf.debut = NULL;  }  return nf;  }; |

# Algorithme principal

## Analyse du travail à faire

Nous demandons à l’utilisateur le choix de l’action qu’il veut effectuer sur sa file d’attente. Il a les mêmes choix que précédemment et en plus une option « undo » pour annuler sa dernière action.  
Si son choix modifie la file d’attente, on en fait une sauvegarde dans la pile avant de faire l’action.

La ou les action(s) choisie(s) sont ensuite effectuées comme dans le TP précédent.

Si la pile contient déjà un élément nous faisons pointer la tête de la pile sur le deuxième élément et dépilons le premier, sinon nous affichons que rien ne peut être annulé.

## Algorithme

**Utilise** FileDAttente, Pile

**Programme** pile

**Début**

**Avec** choix : **Entier**

p\_maillon : **Pointeur** maillon

maillon\_temp : maillon

numero : **Entier**

nom : **Chaîne**

f : FileAttente

continuer : **Booléen**

historique : Pile

p\_tmpcompo : **Pointeur** CompoPile

tmpcompo : CompoPile

initPile(historique)

init(f)

continuer <- **Vrai**

**Tant** **Que** continuer **Faire**

**Afficher** "File d'attente"

**Afficher** "1. Ajouter une personne en queue"

**Afficher** "2. Retirer la personne en tête"

**Afficher** "3. Consulter la personne en tête de file"

**Afficher** "4. Calculer la longueur de la file d'attente"

**Afficher** "5. Désinitialiser la liste"

**Afficher** "6. Undo"

**Afficher** "7. QUITTER"

**Afficher** "Saisir choix : "

**Saisir** choix

**Si** choix = 1 ou choix = 2 ou choix = 5 **Alors**

// Choix destructif, on sauvegarde

filetemp <- dupFile(f)

p\_tmpcompo <- **Nouveau** CompoPile

(\*p\_tmpcompo).contenu <- filetemp

empiler(historique, p\_tmpcompo)

**Fin** **si**

**Selon** choix **Faire**

**Cas** 1:

**Afficher** "Entrer le numéro de la personne : "

**Saisir** numero

**Afficher** "Entrer le nom de la personne : "

**Saisir** nom

p\_maillon <- **Nouveau** maillon

(\*p\_maillon).numero <- numero

(\*p\_maillon).nom <- nom

ajoutEnQueue(f, p\_maillon)

**Cas** 2:

supprEnTete(f)

**Cas** 3:

**Si** calculerLongueur(f) > 0 **Alors**

maillon\_temp <- consulterTete(f)

**Afficher** "Le numéro de la personne est : "

**Afficher** maillon\_temp.numero

**Afficher** "Son nom est : "

**Afficher** maillon\_temp.nom

**Sinon**

**Afficher** "La file est vide, il n'y a pas de tête"

**Fin** **Si**

**Cas** 4:

**Afficher** "Longueur de la file d'attente : "

**Afficher** calculerLongueur(f)

**Cas** 5:

desInit(f)

**Afficher** "Liste désinitialisée"

**Cas** 6: // Undo

**Si** hauteur(historique) > 0 **Alors**

tmpcompo <- obtenirTete(historique)

f <- tmpcompo.contenu

depiler(historique)

**Sinon**

**Afficher** "Il n'y a rien à annuler."

**Fin** **Si**

**Cas** 7:

continuer <- **Faux**

**Défaut**:

**Afficher** "Votre choix n'est pas valide"

**Fin** **faire**

**Fin** **faire**

**Fin** **fileAttente**

## Programme

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19  20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 | //  // main.cpp  // TP03  //  // Created by Oncle D. on 03/10/2014.  // Copyright (c) 2014 Oncle D. All rights reserved.  //  #include <iostream>  #include <string.h>  #include "file.h"  #include "pile.h"  using namespace std;  int main(int argc, const char \* argv[]) {    int choix;  struct maillon \*p\_maillon;  struct maillon maillon\_temp;  int numero;  char nom[80];  struct FileAttente f;  struct FileAttente filetemp;  struct Pile historique;  struct CompoPile\* p\_tmpcompo;  struct CompoPile tmpcompo;  bool continuer=true;  initPile(historique);  init(f);    while (continuer) {    cout << "FILE D'ATTENTE" << endl << endl;  cout << "1. Ajouter une personne en queue" << endl;  cout << "2. Retirer la personne en tête" << endl;  cout << "3. Consulter la personne en tête de file" << endl;  cout << "4. Calculer la longueur de la file d'attente" << endl;  cout << "5. Désinitialiser la liste" << endl;  cout << "6. Undo"<< endl << endl;    cout << "7. QUITTER" << endl;    cout << "Saisir choix : ";  cin >> choix;  if (choix == 1 || choix == 2 || choix == 5) { // Choix destructif  // On sauvegarde la file actuelle  filetemp = dupFile(f);  p\_tmpcompo = new CompoPile;  (\*p\_tmpcompo).contenu = filetemp;  empiler(historique, p\_tmpcompo);  }    switch (choix) {  case 1:  cout << "Entrez le numéro de la personne : ";  cin >> numero;  cout << "Entrez son nom : ";  cin >> nom;  p\_maillon = new maillon;  (\*p\_maillon).numero = numero;  strcpy((\*p\_maillon).nom, nom);  (\*p\_maillon).suivant = NULL;  ajoutEnQueue(f, p\_maillon);  break;    case 2:  supprEnTete(f);  break;    case 3:  if (calculerLongueur(f) > 0) {  maillon\_temp = consulterTete(f);  cout << "Le numéro de la personne est : " << maillon\_temp.numero << endl;  cout << "Son nom est : " << maillon\_temp.nom << endl;  }  else  cout << "La liste est vide, il n'y a pas de tête." << endl;  break;    case 4:  cout << "Longueur de la file d'attente : " << calculerLongueur(f) << endl;  break;    case 5:  desInit(f);  cout << "Liste désinitialisée." << endl;  break;    case 6: // Undo  if (hauteur(historique) > 0) {  tmpcompo = obtenirTete(historique);  f = tmpcompo.contenu;  depiler(historique);  }  else  cout << "Il n'y a rien à annuler." << endl;  break;    case 7:  continuer=false;  break;    default:  cout << "Votre choix n'est pas valide" << endl;  break;  }  cout << endl;  }  return 0;  } |

## Jeux d’essai

Note : Nous ne retestons pas la structure file, celle-ci ayant déjà été testée dans le TP précédent. Les actions sont effectuées les unes à la suite des autres.

|  |  |  |
| --- | --- | --- |
| **Cas testé** | **Entrée** | **Résultat** |
| Annuler alors qu’il n’y a rien à annuler | 6 | Il n’y a rien à annuler |
| Annuler un ajout | 1, puis 3, Marc. 6 | Annulé |
| Vérification | 3 | La liste est vide |
| Ajouter deux personnes | 1, puis 2, Jean | Personne ajoutée |
|  | 1, puis 7, Alex | Personne ajoutée |
| Supprimer une personne | 2 | Personne supprimée |
| Consulter la tête de file | 3 | Alex 7 |
| Annuler | 6 | Annulé |
| Consulter la personne en tête de file | 3 | Jean 2 |
| Désinitialiser la liste | 5 | Désinitialisée |
| Vérification | 4 | Longueur : 0 |
| Annuler | 6 | Annulé |
| Calculer la longueur | 4 | Longueur : 2 |
| Consulter la tête de file | 3 | Jean 2 |

























