Kajy University: Informatique

Algorithmes et structures de données

Author: Dimby Rabearivony

Date: 23 avril 2024

Version: 1.0

"Ny fahalalana no valamparihiko."

Table des matières

Introduction

“En fait, je dirai que la différence entre un mauvais programmeur et un bon réside dans le fait qu’il considère son code ou ses structures de données comme plus importants. Les mauvais programmeurs se soucient du code. Les bons programmeurs se soucient des structures de données et de leurs relations.”

- Linus Torvalds

Chapitre Structures de données fondamentales

1.1 Variables et langage de programmation

1.1.1 Déclaration de variables

Les variables constituent l'un des concepts les plus fondamentaux en programmation. En langage C, une variable est un espace de stockage nommé qui peut contenir une valeur modifiable. Les variables sont utilisées pour stocker des données telles que des nombres, des caractères et des adresses mémoire.

```
1 int age; // Declaration d'une variable de type entier appelee "age"
2 float prix; // Declaration d'une variable de type flottant appelee
      "prix"
3 char lettre; // Declaration d'une variable de type caractere
      appelee "lettre"
```

1.1.2 Initialisation des variables

Les variables peuvent être initialisées lors de leur déclaration en leur attribuant une valeur initiale. Par exemple :

```
1 int nombre = 10; // Declaration et initialisation d'une variable de
       type entier avec la valeur 10
2 float pi = 3.14; // Declaration et initialisation d'une variable de
       type flottant avec la valeur 3.14
3 char grade = 'A'; // Declaration et initialisation d'une variable
      de type caractere avec la valeur 'A'
```

1.1.3 Utilisation des variables

Une fois déclarées et éventuellement initialisées, les variables peuvent être utilisées dans le programme pour stocker et manipuler des données. Par exemple :

```
1 #include <stdio.h>;
2 int x = 5;
3 int y = 10;
4 int somme = x + y; // Addition des valeurs des variables x et y
5 printf("La␣somme␣de␣%d␣et␣%d␣est␣%d\n", x, y, somme); // Affichage
      du resultat
```

1.1.4 Portée des variables

La portée d'une variable en C détermine où elle peut être utilisée dans le programme. Les variables peuvent être locales à une fonction, auquel cas elles ne sont accessibles que dans cette fonction, ou elles peuvent être globales, auquel cas elles sont accessibles dans tout le programme.

```
1  #include <stdio.h>;
2
3  int globalVar = 100; // Variable globale
4
5  void exampleFunction() {
6    int localVar = 50; // Variable locale a la fonction
         exampleFunction
7    printf("La␣variable␣globale␣est␣%d\n", globalVar); // Acces a la
         variable globale
8    printf("La␣variable␣locale␣est␣%d\n", localVar); // Acces a la
         variable locale
9  }
10
11 int main() {
12   printf("La␣variable␣globale␣est␣%d\n", globalVar); // Acces a la
         variable globale
13   // printf("La variable locale est %d\n", localVar); // Cela
         generera une erreur car localVar est locale a exampleFunction
14   exampleFunction();
15   return 0;
16 }
```

Les variables sont un element essentiel en langage C et constituent la base de la manipulation des donnees dans les programmes. Il est crucial de comprendre leur declaration, leur initialisation, leur utilisation et leur portee pour ecrire des programmes efficaces et fonctionnels.

1.2 Pointeurs

Les pointeurs sont un concept essentiel en langage C. Un pointeur est une variable qui contient l'adresse mémoire d'une autre variable. En d'autres termes, un pointeur pointe vers l'emplacement en mémoire où une valeur est stockée.

1.2.1 Déclaration de pointeurs

En langage C, un pointeur est déclaré en précédant le nom de la variable avec l'opérateur d'indirection *, qui indique que la variable est un pointeur. Voici un exemple de déclaration de pointeur :

```
1 int *ptr; // Declaration d'un pointeur vers un entier
2 float *ptr_float; // Declaration d'un pointeur vers un flottant
3 char *ptr_char; // Declaration d'un pointeur vers un caractere
```

1.2.2 Initialisation de pointeurs

Les pointeurs peuvent être initialisés avec l'adresse mémoire d'une variable existante à l'aide de l'opérateur d'adresse &. Voici un exemple d'initialisation de pointeur :

```
1 int var = 10; // Declaration et initialisation d'une variable
2 int *ptr; // Declaration d'un pointeur
3 ptr = &var; // Initialisation du pointeur avec l'adresse de la
      variable var
```

1.2.3 Utilisation de pointeurs

Une fois qu'un pointeur est initialisé, il peut être utilisé pour accéder à la valeur à laquelle il pointe ou pour modifier cette valeur. Voici quelques exemples :

```
1 #include <stdio.h>;
2 int var = 10; // Declaration et initialisation d'une variable
3 int *ptr; // Declaration d'un pointeur
4 ptr = &var; // Initialisation du pointeur avec l'adresse de la
      variable var
5 printf("La␣valeur␣de␣var␣est␣%d\n", var); // Affichage de la valeur
       de var
6 printf("L'adresse␣de␣var␣est␣%p\n", &var); // Affichage de l'
      adresse de var
7 printf("La␣valeur␣pointee␣par␣le␣pointeur␣est␣%d\n", *ptr); //
      Affichage de la valeur pointee par le pointeur
8 *ptr = 20; // Modification de la valeur pointee par le pointeur
9 printf("La␣nouvelle␣valeur␣de␣var␣est␣%d\n", var); // Affichage de
      la nouvelle valeur de var
```

Les pointeurs sont un concept puissant en langage C, mais ils nécessitent une manipulation prudente pour éviter les erreurs de segmentation et les fuites de mémoire.

1.3 Types de données

Les types de données en langage C déterminent la nature des valeurs qu'une variable peut contenir. Le langage C prend en charge plusieurs types de données de base, notamment les entiers, les flottants et les caractères, ainsi que des types de données dérivés tels que les tableaux et les structures.

1.3.1 Types de données de base

Les types de données de base définissent les valeurs simples que peuvent contenir les variables en langage C. Voici quelques-uns des types de données de base les plus couramment utilisés :

a) **int** : Pour les entiers signés.

b) **float** : Pour les nombres à virgule flottante.

c) **double** : Pour les nombres à virgule flottante doubles précision.

d) **char** : Pour les caractères ASCII.

Voici comment ces types de données peuvent être utilisés dans des déclarations de variables :

```
1 int age = 30; // Declaration d'une variable de type entier
2 float poids = 75.5; // Declaration d'une variable de type flottant
3 double prix = 99.99; // Declaration d'une variable de type double
4 char grade = 'A'; // Declaration d'une variable de type caractere
```

1.3.2 Types de données dérivés [1]

Outre les types de données de base, le langage C offre la possibilité de créer des types de données dérivés, tels que les tableaux, les structures et les pointeurs. Ces types de données permettent de regrouper des valeurs connexes ou d'adresser des données de manière plus complexe.

a) **Tableaux** :

Un tableau est une collection ordonnée d'éléments du même type. Les éléments d'un tableau sont accessibles via un index numérique. Voici un exemple de déclaration et d'utilisation d'un tableau en langage C :

```
1 int tableau[5]; // Declaration d'un tableau d'entiers de
      taille 5
2 tableau[0] = 10; // Attribution de la valeur 10 au premier
      element du tableau
3 tableau[1] = 20; // Attribution de la valeur 20 au deuxieme
      element du tableau
```

1. Refer back to [1] for this section. Check well that all is correct.

b) **Structures** :

Une structure est une collection de variables de types différents regroupées sous un seul nom. Elle permet de définir des types de données personnalisés. Voici un exemple de déclaration et d'utilisation d'une structure en langage C :

```c
#include <stdio.h>;
#include <string.h>;

struct Personne {
  char nom[50];
  int age;
  float taille;
};

struct Personne p1; // Declaration d'une structure de type Personne
strcpy(p1.nom, "John Doe"); // Attribution d'une valeur au champ nom
p1.age = 30; // Attribution d'une valeur au champ age
p1.taille = 1.75; // Attribution d'une valeur au champ taille
```

c) **Chaînes de caractères** :

Les chaînes de caractères en C sont des tableaux de caractères terminés par un caractère nul ('\0'). Elles sont utilisées pour représenter et manipuler du texte. Voici un exemple de déclaration et d'utilisation de chaînes de caractères en langage C :

```c
#include <stdio.h>;
#include <string.h>;

char chaine[50]; // Declaration d'une chaine de caracteres
strcpy(chaine, "Bonjour"); // Attribution d'une valeur a la chaine
printf("%s\n", chaine); // Affichage de la chaine
```

Les types de données en langage C offrent une flexibilité et une puissance considérables pour la manipulation des données. Il est essentiel de comprendre ces types de données et leurs utilisations pour écrire des programmes efficaces et fonctionnels.

1.4 Exercices

Types de données :

1. Écrivez un programme C pour déclarer une variable de chaque type de données de base et initialisez-les avec des valeurs.
2. Déclarez un tableau d'entiers de taille 10 et initialisez-le avec des valeurs de votre choix. Affichez ensuite ces valeurs à l'écran.
3. Créez une structure `Personne` avec des champs pour le nom, l'âge et la taille. Déclarez une variable de type `Personne` et initialisez-la avec des valeurs fictives. Affichez ensuite ces valeurs à l'écran.
4. Écrivez une fonction en C pour inverser une chaîne de caractères donnée.
5. Écrivez une fonction en C pour trier un tableau d'entiers en utilisant l'algorithme de tri à bulles.

Types de données de base :

1. Écrivez un programme C pour convertir un nombre entier en binaire.
2. Écrivez une fonction en C pour calculer la somme des chiffres d'un nombre entier.
3. Déclarez une variable de type `char` et utilisez-la pour stocker une lettre majuscule. Ensuite, utilisez une opération pour convertir cette lettre en minuscule.
4. Écrivez un programme C pour vérifier si un nombre donné est premier ou non.
5. Écrivez une fonction récursive en C pour calculer le factoriel d'un nombre entier.

Types de données dérivés :

1. Écrivez une fonction en C pour concaténer deux chaînes de caractères données.
2. Déclarez un tableau de structures `Etudiant` avec des champs pour le nom, l'âge et la moyenne. Initialisez-le avec des valeurs fictives et affichez ensuite ces valeurs à l'écran.
3. Écrivez une fonction en C pour ajouter un élément à une liste chaînée.
4. Implémentez une file (queue) en utilisant des listes chaînées en C.
5. Écrivez une fonction en C pour supprimer un élément d'un arbre binaire de recherche.

Chapitre Algorithmes de recherche et de tri

2.1 Qu'est-ce qu'un algorithme ?

Un algorithme est une série d'étapes visant à résoudre un problème ou à accomplir une tâche spécifique. En informatique, les algorithmes permettent de manipuler, trier, rechercher, et transformer des données. Pour qu'un algorithme soit efficace, il doit être :

1. **Correct** : Donner le bon résultat pour tous les cas.
2. **Efficace** : Rapide et utilisant des ressources raisonnables.
3. **Simple** : Compréhensible et facile à mettre en œuvre.
4. **Flexible** : Adaptable à différentes situations.

Un exemple simple d'algorithme qui trouve le maximum de deux nombres :

Algorithm 1 Trouver le maximum de deux nombres

```
if a > b then
    retour a
else
    retour b
end if
```

2.2 Tri

Le tri consiste à organiser les données dans un ordre particulier. Les algorithmes de tri couramment utilisés comprennent :

1. **Tri à bulles** : Compare et échange des éléments adjacents pour les mettre dans le bon ordre.
2. **Tri par insertion** : Insère chaque élément dans sa position correcte.
3. **Tri rapide (QuickSort)** : Utilise la technique de division et conquête.
4. **Tri fusion (MergeSort)** : Divise, trie séparément, puis fusionne.
5. **Tri par tas (HeapSort)** : Utilise une structure de tas.

Voici un exemple de tri à bulles :

Algorithm 2 Tri à bulles

```
1: for i de 0 à n-2 do
2:     for j de 0 à n-i-2 do
3:         if arr[j] > arr[j+1] then
4:             échanger arr[j] et arr[j+1]
5:         end if
6:     end for
7: end for
```

2.3 Algorithmes de recherche

Les algorithmes de recherche permettent de trouver des éléments spécifiques dans un ensemble de données. Les types courants d'algorithmes de recherche incluent :

1. **Recherche linéaire** : Parcourt une liste séquentiellement.
2. **Recherche binaire** : Utilise une technique de division pour accélérer la recherche.
3. **Recherche par hachage** : Utilise une table de hachage pour recherche rapide.
4. **Recherche dans un arbre binaire** : Utilise une structure d'arbre binaire de recherche.

Voici un exemple de recherche linéaire :

Algorithm 3 Recherche linéaire

```
1: for i de 0 à n-1 do
2:     if arr[i] == target then
3:         retour i
4:     end if
5: end for
6: retour -1
```

Et un exemple de recherche binaire :

Algorithm 4 Recherche binaire

```
1: left := 0
2: right := n-1
3: while left <= right do
4:     mid := left + (right - left) / 2
5:     if arr[mid] == target then
6:         retour mid
7:     else if arr[mid] < target then
8:         left := mid + 1
9:     else
10:        right := mid - 1
11:    end if
12: end while
13: retour -1
```

Chapitre Complexité algorithmique

3.1 Récursion

La récursion est un concept fondamental en informatique où une fonction s'appelle elle-même. C'est souvent utilisé pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires. Les fonctions récursives nécessitent une condition de terminaison pour éviter des appels infinis, ce qui peut entraîner des débordements de pile.

Un exemple classique de récursion est le calcul de la factorielle d'un nombre :

```
1  int fact(int n) {
2    if (n == 0) {
3      return 1;
4    } else {
5      return n * fact(n - 1);
6    }
7  }
```

Les récursions peuvent aussi entraîner des complexités spatiales élevées à cause de l'utilisation de la pile pour stocker les appels récursifs.

3.2 La notation O

La notation "O" (grand O) est utilisée pour décrire la complexité temporelle d'un algorithme. Elle donne une estimation du temps d'exécution en fonction de la taille de l'entrée, ce qui aide à évaluer l'efficacité d'un algorithme.

L'estimation du temps peut être déterminée par le nombre de fois qu'une opération clé est effectuée. Par exemple, un algorithme de recherche binaire divise le tableau en deux à chaque itération, ce qui donne une complexité de ‘O(log n)‘ :

```
1  fonction recherche_binaire(tableau, valeur)
2  gauche = 0
3  droite = taille(tableau) - 1
4  \While gauche <= droite
5  milieu = (gauche + droite) / 2
6  \If tableau[milieu] == valeur \Then
7  retour milieu
8  \ElsIf tableau[milieu] < valeur \Then
9  gauche = milieu + 1
```

```
10  \Else
11  droite = milieu - 1
12  \EndIf
13  \EndWhile
14  retour -1
15  fin fonction
```

Ce "log n" provient du fait que chaque étape réduit le problème de moitié, ce qui correspond à la définition de logarithme binaire.

3.3 Les notations Ω et θ

Les notations Ω et θ complètent la notation "O" pour donner une vision complète de la complexité algorithmique.

- Ω représente une borne inférieure. Cela signifie que l'algorithme ne peut pas être plus rapide que cette complexité. - θ représente une borne moyenne. Cela signifie que c'est la complexité attendue en pratique.

Pour un algorithme de somme de tableau, la complexité est $\theta(n)$ car chaque élément doit être parcouru :

```
1  fonction somme(tableau)
2  somme = 0
3  \For i de 0 a taille(tableau) - 1 \Do
4  somme = somme + tableau[i]
5  \EndFor
6  retour somme
7  fin fonction
```

3.4 Complexité de certains algorithmes

La complexité des algorithmes peut varier en fonction de leur nature et de leur structure. Les algorithmes de tri et de recherche ont souvent des complexités bien connues.

- Tri par insertion : Complexité $O(n^2)$. Les opérations de déplacement des éléments génèrent une complexité quadratique. - Tri rapide (QuickSort) : Complexité 'O(n log n)'. Les divisions successives du tableau et le tri récursif permettent cette complexité. - Recherche linéaire : Complexité 'O(n)'. Il faut parcourir tous les éléments pour trouver une correspondance. - Recherche binaire : Complexité 'O(log n)'. Cette complexité provient de la division successive du tableau.

3.5 Exercices

Pour appliquer les concepts étudiés, voici quelques exercices :

1. Implémentez un algorithme qui trouve le maximum dans un tableau d'entiers. 2. Créez un algorithme récursif pour calculer le terme 'n' d'une suite de Fibonacci. 3. Implémentez un algorithme pour trier un tableau d'entiers avec le tri par sélection. 4. Écrivez un algorithme qui compte le nombre d'occurrences d'un élément donné dans un tableau. 5. Implémentez un algorithme qui recherche un élément dans un tableau avec la recherche binaire. 1

Chapitre Structures de données avancées

4.1 Listes chaînées

Les listes chaînées sont des structures de données composées de nœuds, où chaque nœud contient des données et un pointeur vers le nœud suivant. Elles sont utiles pour des opérations d'insertion et de suppression rapides sans nécessiter de réallocation de mémoire.

Exemple d'implémentation d'une liste chaînée en C :

```c
typedef struct Node {
  int data;
  struct Node* next;
} Node;

void insert(Node** head, int new_data) {
  Node* new_node = (Node*)malloc(sizeof(Node));
  new_node->data = new_data;
  new_node->next = *head;
  *head = new_node;
}

void delete(Node** head, int key) {
  Node* temp = *head, *prev;
  if (temp != NULL && temp->data == key) {
    *head = temp->next;
    free(temp);
    return;
  }
  while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
  }
  if (temp == NULL) return;
  prev->next = temp->next;
  free(temp);
}
```

4.2 Piles

Les piles (ou LIFO, Last In First Out) permettent des opérations de push et pop. Elles suivent le principe du dernier entré, premier sorti. Les piles sont couramment utilisées pour les opérations de retour en arrière, comme dans les navigateurs Web ou les algorithmes de récursion.

Voici un exemple simple d'utilisation de pile pour vérifier les parenthèses équilibrées :

```
int areParenthesesBalanced(char expr[]) {
  Stack* stack = createStack(strlen(expr));
  for (int i = 0; i < strlen(expr); i++) {
    if (expr[i] == '(') {
      push(stack, expr[i]);
    } else if (expr[i] == ')') {
      if (isEmpty(stack)) {
        return 0;
      }
      pop(stack);
    }
  }
  return isEmpty(stack);
}
```

4.3 Files

Les files (ou FIFO, First In First Out) permettent des opérations d'enfilage et de défilage. Elles sont utiles pour la gestion des tâches, les files d'attente, et les algorithmes de parcours.

Exemple d'une file utilisant une structure de données basée sur des listes chaînées :

```
typedef struct QueueNode {
  int data;
  struct QueueNode* next;
} QueueNode;

typedef struct Queue {
  QueueNode* front;
  QueueNode* rear;
} Queue;

Queue* createQueue() {
```

```
12    Queue* queue = (Queue*)malloc(sizeof(Queue));
13    queue->front = queue->rear = NULL;
14    return queue;
15  }
16
17  void enqueue(Queue* queue, int data) {
18    QueueNode* new_node = (QueueNode*)malloc(sizeof(QueueNode));
19    new_node->data = data;
20    new_node->next = NULL;
21    if (queue->rear == NULL) {
22      queue->rear = queue->front = new_node;
23    } else {
24      queue->rear->next = new_node;
25      queue->rear = new_node;
26    }
27  }
28
29  int dequeue(Queue* queue) {
30    if (queue->front == NULL) {
31      return -1;
32    }
33    QueueNode* temp = queue->front;
34    queue->front = queue->front->next;
35    if (queue->front == NULL) {
36      queue->rear = NULL;
37    }
38    int data = temp->data;
39    free(temp);
40    return data;
41  }
```

4.4 Arbres binaires

Les arbres binaires sont des structures de données où chaque nœud a au maximum deux enfants, généralement appelés gauche et droite. Les arbres binaires sont utilisés pour des opérations de recherche et des algorithmes de parcours comme pré-ordre, en-ordre et post-ordre.

Exemple d'implémentation d'un arbre binaire :

```c
typedef struct TreeNode {
  int data;
  struct TreeNode* left;
  struct TreeNode* right;
} TreeNode;

TreeNode* createNode(int data) {
  TreeNode* new_node = (TreeNode*)malloc(sizeof(TreeNode));
  new_node->data = data;
  new_node->left = NULL;
  new_node->right = NULL;
  return new_node;
}

void insert(TreeNode** root, int data) {
  if (*root == NULL) {
    *root = createNode(data);
  } else {
    if (data < (*root)->data) {
      insert(&(*root)->left, data);
    } else {
      insert(&(*root)->right, data);
    }
  }
}
```

4.5 Arbres

Les arbres sont des structures de données génériques qui peuvent avoir plusieurs enfants. Ils sont utilisés dans des contextes comme les arbres de syntaxe abstraite, les arbres de préfixes, et les arbres de décision.

Exemple d'utilisation d'un arbre pour un arbre de décision :

```c
typedef struct DecisionNode {
  char* question;
  struct DecisionNode* yes;
  struct DecisionNode* no;
} DecisionNode;

```

```c
DecisionNode* createDecisionNode(char* question) {
  DecisionNode* node = (DecisionNode*)malloc(sizeof(DecisionNode));
  node->question = question;
  node->yes = NULL;
  node->no = NULL;
  return node;
}

void addDecision(DecisionNode* root, char* question, int answer) {
  if (answer == 1) {
    root->yes = createDecisionNode(question);
  } else {
    root->no = createDecisionNode(question);
  }
}
```

4.6 Graphes

Les graphes sont des structures de données qui contiennent des nœuds (ou sommets) et des arêtes (qui relient les nœuds). Les graphes sont utilisés pour des problèmes complexes comme la recherche de plus courts chemins, la détection de cycles, et le partitionnement.

Exemple d'implémentation d'un graphe avec des listes d'adjacence :

```c
typedef struct GraphNode {
  int vertex;
  struct GraphNode* next;
} GraphNode;

typedef struct Graph {
  int num_vertices;
  GraphNode** adj_lists;
} Graph;

Graph* createGraph(int num_vertices) {
  Graph* graph = (Graph*)malloc(sizeof(Graph));
  graph->num_vertices = num_vertices;
  graph->adj_lists = (GraphNode**)malloc(num_vertices * sizeof(
      GraphNode*));
  for (int i = 0; i < num_vertices; i++) {
```

```
16      graph->adj_lists[i] = NULL;
17    }
18    return graph;
19  }
20
21  void addEdge(Graph* graph, int src, int dest) {
22    GraphNode* new_node = (GraphNode*)malloc(sizeof(GraphNode));
23    new_node->vertex = src;
24    new_node->next = graph->adj_lists[dest];
25    graph->adj_lists[dest] = new_node;
26
27    new_node = (GraphNode*)malloc(sizeof(GraphNode));
28    new_node->vertex = dest;
29    new_node->next = graph's␣adj_lists[src];
30 ␣␣␣␣graph's adj_lists[src] = new_node;
31  }
```

4.7 Exercices

Voici quelques exercices pour vous aider à pratiquer et consolider vos connaissances des structures de données avancées :

1. Implémentez une liste chaînée avec des opérations d'insertion et de suppression. Testez votre implémentation avec plusieurs cas d'utilisation.
2. Créez un algorithme qui utilise une pile pour évaluer une expression mathématique simple, comme "3 + 5 * 2".
3. Implémentez une file qui suit le principe FIFO (First In, First Out) avec des opérations d'enfilage et de défilage. Testez-la avec différents types de données.
4. Créez un arbre binaire qui peut insérer des valeurs en maintenant une structure de recherche binaire. Ajoutez des fonctions de recherche, de suppression, et de parcours.
5. Implémentez un graphe qui utilise des listes d'adjacence pour représenter des connexions entre des sommets. Ajoutez des fonctions pour ajouter et supprimer des arêtes, et pour trouver des chemins entre des sommets.

Chapitre Applications d'algorithmes en IA

5.1 Greedy algorithm

Bibliographie

[1] Karumanchi, N. (2017). Data structures and algorithms made easy.

[2] Knuth, D. E. (1986). *The TEX Book*. Addison-Wesley Professional.