データ構造とアルゴリズム演習

1TE23035E 平野 健汰

2025 年 1 月 10 日

はじめに今回作成したコードは github にアップロードしています．以下のリンクからアクセスできます．
`https://github.com/kenta-kenta/clang-algorithm`

第 I 部
HashTable

1　チェイン法での実装

1.1　概略

ハッシュテーブルをチェイン法で実装する．

1.2　仕様

- ハッシュ関数は，$hash(key) = key \mod M$ とする．
- ハッシュテーブルのサイズは $M = 13$ とする．
- ハッシュテーブルの各要素は，単方向リスト構造で実装する．
- ハッシュテーブルにデータを挿入する関数は，`hashInsert` とする．
  - 入力は，ハッシュテーブルの配列のポインタとデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素の先頭にデータを挿入する．
- ハッシュテーブルからデータを検索する関数は，`hashSearch` とする．
  - 入力は，ハッシュテーブルの配列のポインタとデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素を先頭から順に探索し，データが見つかったらそのインデックスを返す．
- ハッシュテーブルからデータを削除する関数は，`hashDelete` とする．
  - 入力は，ハッシュテーブルの配列のポインタとデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素を先頭から順に探索し，データが見つかったらその要素を削除する．

1.3 コード

```
#include <stdio.h>
#include <stdlib.h>

#define M 13

int hash(int key)
{
    return key % M;
}

// チェイン法のハッシュ法
typedef struct node
{
    int data;
    struct node *next;
} Node;

// Nodeを出力する
void printList(Node *head)
{
    while (head != NULL)
    {
        printf("%d \n", head->data);
        head = head->next;
    }
}

// Nodeを先頭に追加する
void prepend(Node **head, int data)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = *head;
    if (*head != NULL)
    {
        newNode->next = *head;
    }
    *head = newNode;
}

// Nodeを削除する
void deleteNode(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);
}

// Nodeを検索する
void search(Node *head, int key)
{
    Node *current = head;
    while (current != NULL)
    {
        if (current->data == key)
        {
            printf("Found %d\n", current->data);
            return;
        }
        current = current->next;
    }
    printf("Not Found\n");
}

int hashInsert(Node *arr[], int key)
{
    int j = hash(key);
    prepend(&arr[j], key);
    return j;
}

int hashSearch(Node *arr[], int key)
{
    int j = hash(key);
    search(arr[j], key);
    return j;
}

int hashDelete(Node *arr[], int key)
{
    int j = hash(key);
    deleteNode(&arr[j], key);
    return j;
}

int main()
{
    Node *arr[M] = {NULL};

    hashInsert(arr, 25);
    hashInsert(arr, 37);
    hashInsert(arr, 18);
    hashInsert(arr, 55);
    hashInsert(arr, 22);
```

```
    hashInsert(arr, 35);
    hashInsert(arr, 50);
    hashInsert(arr, 63);
    hashInsert(arr, 69);
    hashInsert(arr, 95);
    hashInsert(arr, 100);
    hashInsert(arr, 105);
    hashInsert(arr, 110);

    for (int i = 0; i < M; i++)
    {
        printf("arr[%d]: ", i);
        printList(arr[i]);
    }

    printf("Search 55: %d\n", hashSearch(arr, 55));
    printf("Search 100: %d\n", hashSearch(arr, 100));
    printf("Search 110: %d\n", hashSearch(arr, 110));
    printf("Search 111: %d\n", hashSearch(arr, 111));

    hashDelete(arr, 55);
    hashDelete(arr, 100);
    hashDelete(arr, 110);
    hashDelete(arr, 111);

    for (int i = 0; i < M; i++)
    {
        printf("arr[%d]: ", i);
        printList(arr[i]);
    }

    return 0;
}
```

1.4 プログラムの特徴

このプログラムは，ハッシュテーブルをチェイン法で実装している．ある数値をハッシュ関数によってハッシュ値に変換し，そのハッシュ値をインデックスとして配列に格納する．ハッシュ値が重複した場合は，リスト構造を用いてデータを追加する．

2 オープンアドレス方式での実装

2.1 概略

ハッシュテーブルをオープンアドレス方式で実装する．

2.2 仕様

- ハッシュ関数は，$hash(key, index) = (key + i) \mod M$ とする．
- ハッシュテーブルのサイズは $M = 13$ とする．
- ハッシュテーブルの各要素は，単一のデータを格納する．

- ハッシュテーブルにデータを挿入する関数は，hashInsert とする．
  - 入力は，数値の配列とデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素が空でない場合は，次のインデックスにデータを挿入する．
- ハッシュテーブルからデータを検索する関数は，hashSearch とする．
  - 入力は，数値の配列とデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素を順に探索し，データが見つかったらそのインデックスを返す．
- ハッシュテーブルからデータを削除する関数は，hashDelete とする．
  - 入力は，数値の配列とデータ．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素を順に探索し，データが見つかったらその要素を削除する．

2.3 コード

```
#include <stdio.h>
#include <stdlib.h>

#define M 13

int hash(int key, int index)
{
    return (key + index) % M;
}

// オープンアドレス方式のハッシュ法
int hashInsert(int arr[], int key)
{
    int index = 0;
    while (index < M)
    {
        int j = hash(key, index);
        if (arr[j] == NULL)
        {
            arr[j] = key;
            return j;
        }
        else
        {
            index++;
        }
    }
}

int hashSearch(int arr[], int key)
{
    int index = 0;
    while (index < M)
    {
        int j = hash(key, index);
```

```
        if (arr[j] == key)
        {
            return j;
        }
        else if (arr[j] == NULL)
        {
            return -1;
        }
        else
        {
            index++;
        }
    }
}

int hashDelete(int arr[], int key)
{
    int index = 0;
    while (index < M)
    {
        int j = hash(key, index);
        if (arr[j] == key)
        {
            arr[j] = NULL;
            return j;
        }
        else if (arr[j] == NULL)
        {
            return -1;
        }
        else
        {
            index++;
        }
    }
}

int main()
{
    int arr[M] = {NULL};
    hashInsert(arr, 25);
    hashInsert(arr, 37);
    hashInsert(arr, 18);
    hashInsert(arr, 55);
    hashInsert(arr, 22);
    hashInsert(arr, 35);
    hashInsert(arr, 50);
    hashInsert(arr, 63);
    hashInsert(arr, 69);
    hashInsert(arr, 95);
    hashInsert(arr, 100);
    hashInsert(arr, 105);
    hashInsert(arr, 110);

    for (int i = 0; i < M; i++)
    {
        printf("%d ", arr[i]);
```

```
    }
    printf("\n");

    printf("Search 55: %d\n", hashSearch(arr, 55));
    printf("Search 100: %d\n", hashSearch(arr, 100));
    printf("Search 110: %d\n", hashSearch(arr, 110));
    printf("Search 111: %d\n", hashSearch(arr, 111));

    hashDelete(arr, 55);
    hashDelete(arr, 100);
    hashDelete(arr, 110);
    hashDelete(arr, 111);

    for (int i = 0; i < M; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

2.4 プログラムの特徴

このプログラムは，ハッシュテーブルをオープンアドレス方式で実装している．ある数値をハッシュ関数によってハッシュ値に変換し，そのハッシュ値をインデックスとして配列に格納する．ハッシュ値が重複した場合は，次のインデックスにデータを挿入する．

3 まとめ

今回，ハッシュテーブルをチェイン法，オープンアドレス方式で実装した．チェイン法は，ハッシュ値が重複した場合にリスト構造を用いてデータを追加する．オープンアドレス方式は，ハッシュ値が重複した場合に次のインデックスにデータを挿入する．

チェイン法のメリットは，データの追加が容易であること，ハッシュ値が重複してもデータを追加できることである．チェイン法のデメリットは，リスト構造を用いるため，メモリの使用量が増えること，ポインタ管理の手間がかかることである．

オープンアドレス方式のメリットは，配列を利用するのでメモリ効率がいいこと，実装がシンプルなことである．オープンアドレス方式のデメリットは，削除が複雑になりうること，ハッシュ値が重複した場合にデータを挿入するための処理が必要であることである．

以上のことを踏まえて実装する必要があり，データ量が予測可能な場合はオープンアドレス方式を，可変長データや動的な状況に対応する場合はチェイン法を選択するといいだろう．

第 II 部
AccountSystem

4　アカウントシステムの実装

4.1　概略

ハッシュテーブルを利用してアカウントシステムを実装する．

4.2　仕様

- ハッシュ関数は，$hash(password) = password \mod M$ とする．
- ハッシュテーブルのサイズは $M = 100$ とする．
- ハッシュテーブルの各要素は，単一のデータを格納する．
- アカウントを登録する関数は，`registerAccount` とする．
  - 入力は，ハッシュテーブルの配列とアカウント名，パスワード．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素が空でない場合は，パスワードを変更する．
- アカウントにログインする関数は，`login` とする．
  - 入力は，ハッシュテーブルの配列とアカウント名，パスワード．
  - 出力は，ハッシュテーブルのインデックス．
  - ハッシュテーブルの要素を順に探索し，アカウントが見つかったらそのインデックスを返す．

4.3　コード

```
#include <stdio.h>

#define MAX 100

int hash(int password)
{
    return password % MAX;
}

int registerAccount(char arr[], char name, int password)
{
    int j = hash(password);
    if (arr[j] == '\0') // NULLの代わりに'\0'を使用
    {
        arr[j] = name;
        return j;
    }
    else
    {
        printf("パスワードを変更してください\n");
        return -1;
    }
```

```
}

int login(char arr[], char name, int password)
{
    int j = hash(password);
    if (arr[j] == name)
    {
        return j;
    }
    else if (arr[j] == '\0') // NULLの代わりに'\0'を使用
    {
        printf("アカウントが存在しません\n");
        return -1;
    }
    else
    {
        printf("パスワードが違います\n");
        return -1;
    }
}

int main()
{
    char arr[MAX];
    // 配列を初期化
    for (int i = 0; i < MAX; i++)
    {
        arr[i] = '\0'; // NULL文字で初期化
    }

    registerAccount(arr, 'C', 9012);
    registerAccount(arr, 'A', 1234);
    registerAccount(arr, 'B', 5678);
    printf("%d\n", hash(9012));
    printf("%d\n", login(arr, 'C', 9012));
    printf("%d\n", login(arr, 'A', 1234));
    printf("%d\n", login(arr, 'B', 5678));
    printf("%d\n", login(arr, 'D', 3456));
    return 0;
}
```

4.4 プログラムの特徴

このプログラムは，ハッシュテーブルを利用してアカウントシステムを実装している．アカウント名とパスワードを登録し，ログインすることができる．パスワードをハッシュ化してハッシュテーブルにアカウント名を格納し，ログイン時にパスワードをハッシュ化して検索する．つまり，パスワードのハッシュの衝突が起こる可能性があり，その場合は不正ログインすることができる．

ハッシュの衝突が起こる確率を求めてみる．ハッシュ関数は，$hash(password) = password \mod M$ である．ハッシュテーブルのサイズは $M = 100$ であるため，ハッシュの衝突確率は $1/100 = 0.01$ である．したがって，ハッシュの衝突が起こる確率は 1% である．

4.5　まとめ

今回，ハッシュテーブルを利用してアカウントシステムを実装した．このハッシュテーブルは実際のシステムとしてはセキュリティが低いため，実際のシステムではハッシュ関数を改良する必要がある．ハッシュ関数の改良方法としては，ハッシュ値の衝突を減らすためにハッシュ関数を複雑化する方法がある．よく利用されるハッシュ関数としては，MD5，SHA-1，SHA-256 などがある．これらのハッシュ関数は，ハッシュ値の衝突が起こりにくいため，セキュリティが高い．

参考文献

[1] `https://example.com/reference`