Tablica z haszowaniem (hash table)

a) adresowanie otwarte (open addressing)
   - szukanie (sondowanie) liniowe (linear probing),
   - szukanie kwadratowe (qudratic probing),
   - mieszanie podwójne (double hashing).

b) metoda łańcuchowa (separate chaining).

Tablica z haszowaniem jest strukturą danych zapewniająca bardzo szybkie wstawianie jak i wyszukiwanie elementów (O(1)), przez co operacje na nich są znacząco szybsze niż na drzewach (O(log N)). Ponadto są stosunkowo łatwe do zaprogramowania. Wadą tablic z haszowaniem jest to, że bazują na tablicach i tym samym ich rozmiar jest z góry określony (zwiększenie rozmiaru nie rozwiązuje problemu gdyż jednym z parametrów funkcji haszującej jest rozmiar tablicy z haszowaniem). Dla pewnych typów tablic z haszowaniem ich wydajność spada gdy wypełnienie elementami tablicy zbliża się do rozmiaru tablicy. Tablice z haszowaniem również nie zapewniają wygodnego sposobu na odwiedzanie elementów według wybranego porządku (np. od najmniejszej przechowywanej wartości do największej, etc.).

Przykładem wykorzystania tablic z haszowaniem może być program zapewniający dostęp do danych pracowników (obecnie zatrudnionych i tych na emeryturze). Każdy pracownik firmy ma unikalny numer (wartość klucza np. od 1 do 1000). Klucz ten może być użyty w celu uzyskania natychmiastowego dostępu do danych pracownika. W takim wypadku można użyć prostej struktury jaką jest zwykła tablica. Wówczas każdy element tablicy przechowywać będzie dane jednego pracownika.

W wielu jednak przypadkach klucze nie są tak dobrze zorganizowane. Klasycznym przykładem jest słownik. Powiedzmy, że chcemy utworzyć słownik przechowujący 50 000 słów. Aby odczytać zawartość elementu tablicy w sposób bardzo szybki potrzebujemy metody która zamieni nam szukane słowo na numer indeksu tablicy (z haszowaniem) pod którym znajduje się owo słowo (lub też dodatkowe informacje związane ze znaczeniem słowa).

Metodami zamiany liter na wartości liczbowe są różnego typu kodowania. Najbardziej znanym przykładem kodowana jest kodowanie ASCII (np. w kodowaniu tym literze „a” odpowiada wartość 97, literze „b” 98 i tak dalej aż do 122 dla litery „z” (tablica ASCII zawiera więcej w ten sposób zakodowanych znaków, łącznie od wartości 0 do 255 (7 bitów). 8 i więcej bitów jest często używanych w kodowaniach nazywanych *stronami kodowymi*).

Upraszczając zagadnienie można ograniczyć się 27 znaków (np. 0 dla braku znaku, 1 dla „a”, 2 dla „b” aż do 26 dla „z”). Ograniczamy też długość słów do 10 liter.
Przykładowo w notacji dziesiętnej 7546 można zapiać jako:

$$\mathbf{7}*10^3 + \mathbf{5}*10^2 + \mathbf{4}*10^1 + \mathbf{6}*10^0$$

Słowo cats w naszym 27-ym układzie można by zapisać:

$$\mathbf{3}*27^3 + \mathbf{1}*27^2 + \mathbf{20}*27^1 + \mathbf{19}*27^0$$

(odpowiada to w układzie dziesiętnym wartości 60337).
Tym samym na zapis dowolnego dziesięcioliterowego słowa potrzebować będziemy tablicy o

rozmiarze:

26*27^9 + **26***27^8 + **26***27^7 + **26***27^6 + **26***27^5 + **26***27^4 + **26***27^3 + **26***27^2 + **26***27^1 + **26***27^0 (zakodowane słowo „zzzzzzzzzz")

co daje wynik w postaci ponad 198*10^12 komórek tablicy (zakładając, że każda komórka przechowywać będzie jeden bajt informacji wielkość potrzebnej pamięci wyniesie ponad 198 TB).

Oczywiście wiele z tych „słów" nie będzie realnymi słowami (np. aaaaaaaaaa, aaaaaaaaab, fira, firb, firc, fird, etc.)

Haszowanie (hashing)
To czego potrzebujemy to sposob na kompresję olbrzymiego zakresu wartości do tablicy o rozsądnej liczbie elementów. Okazuje się, że dla słownika o liczbie 50 000 słów lepiej jest utworzyć tablicę o dwa razy większej liczbie elementów. Prostym sposobem na ściśnięcie rozmiaru tablicy jest użycie operatora modulo (%) (zwraca resztę z dzielenia jednej liczby przez drugą).

Powiedzmy, że chcemy ścisnąć zakres 0 - 99 do zakresu 0 - 9. Liczby z dużego zakresu niech będą reprezentowane przez zmienną largeNumber a z małego przez zmienną smallNumber. Liczba cyfr z zakresu małego niech będzie dana zmienną smallRange (tutaj smallRange = 10). Wówczas konwersję w języku C++ można zapisać jako:

```
smallNumber = largeNumber % smallRange;
```

(Reszta z dzielenia gdy smallRange = 10 zawsze pozostaje w zakresie cyfr 0 – 9, np. 13%10 daje 3, 157%10 daje 7).
Tego typu konwersja jest przykładem ***funkcji haszującej*** (***hash function***). Haszuje (konwertuje) liczbę z dużego zakresu na liczbę z małego zakresu. Liczba z małego zakresu odpowiada indeksowi tablicy. Tablica elementów w których umieszczane są dane przy użyciu funkcji haszującej nosi nazwę ***tablicy z haszowaniem*** (***hash table***).

Kolizje (Collisions)
Możliwe jest, że kilka kluczy z dużego zakresu w wyniku haszowania da ten sam indeks.
Wówczas w celu uniknięcia nadpisania wstawionego wcześniej elementu (przechowującego klucz, dane lub też dane i klucz) stosuje się adresowanie otwarte lub metodę łańcuchową.

I. Adresowanie otwarte można podzielić na:
   a) szukanie liniowe (szukamy wolnych komórek o indeksach X+1, X+2, X+3, X+4, …, gdzie X indeksem otrzymanym z haszowania; element umieszczamy w pierwszej wolnej komórce),
   b) szukanie kwadratowe (szukamy wolnych komórek o indeksach X+1^2, X+2^2, X+3^2, X+4^2,… , gdzie X indeksem otrzymanym z haszowania; element umieszczamy w pierwszej wolnej komórce),
   c) mieszanie podwójne (szukamy wolnych komórek za pomocą sondowania zależnego od klucza zamiast stosowania tego samego szeregu dla wszystkich kluczy; element umieszczamy w pierwszej wolnej komórce). Sondowanie to w rezultacie jest drugą funkcją haszującą od której wymaga się aby spełniała następujące warunki:
      - nie może być taka sama jak pierwsza funkcja haszująca
      - nie może dawać wartości 0

np.
stepSize = constant - (key % constant);
stepSize = 5 - (key % 5);
W przypadku podwójnego haszowania (mieszania) wymagane jest aby rozmiar tablicy z haszowaniem był liczbą pierwszą (otrzymujemy wówczas nie powtarzające się indeksy używane podczas sondowania i tym samym można w poszukiwaniu wolnego miejsca przejrzeć wszystkie elementy tablicy zamiast tylko kilku).

II. Metoda łańcuchowa. W metodzie tej każdy element tablicy jest powiązany z listą np. listą jednostronnie wiązaną lub też jak w przykładzie nr 3 z posortowaną listą jednostronnie wiązaną (lista posortowana zapewnia szybsze odnajdowanie i kasowanie elementu, lecz kosztem szybkości wstawiania (co w wypadku krótkich list nie musi być ważne)).

Klastrowanie (grupowanie, clustering)
W wyniku wstawiania coraz większej liczby elementów tworzyć się będą grupy elementów o niezerowych wartościach kluczy. Klastrowanie wpływa negatywnie na szybkość odnajdowania i wstawiania elementów.

Kasowanie
Skasowanie elementu (w przypadku adresowania otwartego) z wnętrza klastra odbywa się przez ustawienie wartości klucza na np. -1.

Współczynnik wypełnienia (load factor)
Jest to stosunek liczby istniejących elementów tablicy do jej rozmiaru. W adresowaniu otwartym współczynnik ten nie powinien znacznie przekraczać 0.5 (poniżej 0.5 dla liniowego i poniżej 2/3 dla kwadratowego i podwójnego haszowania). W przypadku metody łańcuchowej współczynnik ten może być większy niż 1 lecz nie zaleca się przekraczania wartości 2).

Przykład 1 (C++). Implementacja tablicy z haszowaniem przy zastosowaniu wyszukiwania liniowego. Zmienna keysPerCell definiuje stosunek kompresji (w programie ma wartość równą 10. Oznacza to, że określając rozmiar tablicy (z haszowaniem) na 20 wpisywane mogą być klucze z zakresu od 0 do 200.

[hash.cpp]

```
//hash.cpp
//demonstrates hash table with linear probing
#include <iostream>
#include <vector>
#include <cstdlib>                    //for random numbers
#include <ctime>                      //for random numbers
using namespace std;

class DataItem
{                                       //(could have more data)
    public:
        int iData;                     //data item (key)

        DataItem(int ii) : iData(ii)       //constructor
        {  }

};  //end class DataItem
```

```
//--------------------------------------------------------------
class HashTable
{
    private:
        vector<DataItem*> hashArray;    //vector holds hash table
        int arraySize;
        DataItem* pNonItem;             //for deleted items

    public:
        HashTable(int size) : arraySize(size)  //constructor
        {
            arraySize = size;
            hashArray.resize(arraySize);    //size the vector
            for(int j=0; j<arraySize; j++)  //initialize elements
                hashArray[j] = NULL;
            pNonItem = new DataItem(-1);    //deleted item key is -1
        }

        void displayTable()
        {
            cout << "Table: ";
            for(int j=0; j<arraySize; j++)
            {
                if(hashArray[j] != NULL)
                    cout << hashArray[j]->iData << " ";
                else
                    cout << "** ";
            }
            cout << endl;
        }

        int hashFunc(int key)
        {
            return key % arraySize;         //hash function
        }

        void insert(DataItem* pItem)        //insert a DataItem
        {                                   //(assumes table not full)
            int key = pItem->iData;         //extract key
            int hashVal = hashFunc(key);    //hash the key
                                            //until empty cell or -1,
            while(hashArray[hashVal] != NULL && hashArray[hashVal]->iData
            != -1)
            {
                ++hashVal;                   //go to next cell
                hashVal %= arraySize;        //wraparound if necessary
            }
            hashArray[hashVal] = pItem;     //insert item
        }  //end insert()

        DataItem* remove(int key)          //remove a DataItem
        {
            int hashVal = hashFunc(key);    //hash the key

            while(hashArray[hashVal] != NULL)  //until empty cell,
            {                                  //found the key?
                if(hashArray[hashVal]->iData == key)
                {
                    DataItem* pTemp = hashArray[hashVal]; //save item
                    hashArray[hashVal] = pNonItem;        //delete item
                    return pTemp;                         //return item
```

```
                }
                ++hashVal;                      //go to next cell
                hashVal %= arraySize;           //wraparound if necessary
            }
            return NULL;                        //can't find item
        }  //end remove()

        DataItem* find(int key)                 //find item with key
        {
            int hashVal = hashFunc(key);        //hash the key

            while(hashArray[hashVal] != NULL)   //until empty cell,
            {                                   //found the key?
                if(hashArray[hashVal]->iData == key)
                    return hashArray[hashVal];  //yes, return item
                ++hashVal;                      //go to next cell
                hashVal %= arraySize;           //wraparound if necessary
            }
            return NULL;                        //can't find item
        }

};  //end class HashTable
//-----------------------------------------------------------------
int main()
{
    DataItem* pDataItem;
    int aKey, size, n, keysPerCell;
    time_t aTime;
    char choice = 'b';
                                        //get sizes
    cout << "Enter size of hash table: ";
    cin >> size;
    cout << "Enter initial number of items: ";
    cin >> n;
    keysPerCell = 10;
                                        //make table
    HashTable theHashTable(size);
    srand( static_cast<unsigned>(time(&aTime)) );
    for(int j=0; j<n; j++)              //insert data
    {
        aKey = rand() % (keysPerCell*size);
        pDataItem = new DataItem(aKey);
        theHashTable.insert(pDataItem);
    }

    while(choice != 'x')                //interact with user
    {
        cout << "Enter first letter of "
             << "show, insert, delete, or find: ";
        char choice;
        cin >> choice;
        switch(choice)
        {
            case 's':
                theHashTable.displayTable();
                break;
            case 'i':
                cout << "Enter key value to insert: ";
                cin >> aKey;
                pDataItem = new DataItem(aKey);
                theHashTable.insert(pDataItem);
```

```
                break;
            case 'd':
                cout << "Enter key value to delete: ";
                cin >> aKey;
                theHashTable.remove(aKey);
                break;
            case 'f':
                cout << "Enter key value to find: ";
                cin >> aKey;
                pDataItem = theHashTable.find(aKey);
                if(pDataItem != NULL)
                    cout << "Found " << aKey << endl;
                else
                    cout << "Could not find " << aKey << endl;
                break;
            default:
                cout << "Invalid entry\n";
        }  //end switch
    }  //end while
    return 0;
}  //end main()
```

Zadanie 1 (C++). W oparciu o kod z przykładu 1 napisz program realizujący strukturę danych typu tablica z haszowaniem z metodą adresowania typu szukanie kwadratowe.

Przykład 2 (C++). Implementacja tablicy z haszowaniem przy zastosowaniu podwójnego haszowania.

[hashDouble.cpp]

```
//hashDouble.cpp
//demonstrates hash table with double hashing
#include <iostream>
#include <vector>
#include <cstdlib>                  //for random numbers
#include <ctime>                    //for random numbers
using namespace std;

class DataItem
{                                   //(could have more items)
    public:
        int iData;                      //data item (key)

        DataItem(int ii) : iData(ii)      //constructor
        {  }

};  //end class DataItem
//-----------------------------------------------------------
class HashTable
{
    private:
        vector<DataItem*> hashArray;    //vctor holds hash table
        int arraySize;
        DataItem* pNonItem;             //for deleted items

    public:
        HashTable(int size) : arraySize(size)  //constructor
        {
```

```
    hashArray.resize(arraySize);   //size the vector
    for(int j=0; j<arraySize; j++) //initialize elements
        hashArray[j] = NULL;
    pNonItem = new DataItem(-1);
}

void displayTable()
{
    cout << "Table: ";
    for(int j=0; j<arraySize; j++)
    {
        if(hashArray[j] != NULL)
            cout << hashArray[j]->iData << " ";
        else
            cout << "** ";
    }
    cout << endl;
}

int hashFunc1(int key)
{
    return key % arraySize;
}

int hashFunc2(int key)
{
    //non-zero, less than array size, different from hF1
    //array size must be relatively prime to 5, 4, 3, and 2
    return 5 - key % 5;
}

//insert a DataItem
void insert(int key, DataItem* pItem)
{                                  //(assumes table not full)
    int hashVal = hashFunc1(key);  //hash the key
    int stepSize = hashFunc2(key); //get step size
                                   //until empty cell or -1
    while(hashArray[hashVal] != NULL &&
    hashArray[hashVal]->iData != -1)
    {
        hashVal += stepSize;        //add the step
        hashVal %= arraySize;       //for wraparound
    }
    hashArray[hashVal] = pItem;    //insert item
}  //end insert()

DataItem* remove(int key)   //delete a DataItem
{
    int hashVal = hashFunc1(key);  //hash the key
    int stepSize = hashFunc2(key); //get step size

    while(hashArray[hashVal] != NULL)  //until empty cell,
    {                                  //is correct hashVal?
        if(hashArray[hashVal]->iData == key)
        {
            DataItem* pTemp = hashArray[hashVal]; //save item
            hashArray[hashVal] = pNonItem;        //delete item
            return pTemp;                         //return item
        }
        hashVal += stepSize;        //add the step
        hashVal %= arraySize;       //for wraparound
```

```
            }
            return NULL;                    //can't find item
        }  //end remove()

        DataItem* find(int key)           //find item with key
        {                           //(assumes table not full)
            int hashVal = hashFunc1(key);  //hash the key
            int stepSize = hashFunc2(key); //get step size

            while(hashArray[hashVal] != NULL)  //until empty cell,
            {                                  //is correct hashVal?
                if(hashArray[hashVal]->iData == key)
                    return hashArray[hashVal];   //yes, return item
                hashVal += stepSize;             //add the step
                hashVal %= arraySize;            //for wraparound
            }
            return NULL;                    //can't find item
        };

};  //end class HashTable
//-----------------------------------------------------------------
int main()
{
    int aKey;
    DataItem* pDataItem;
    int size, n;
    char choice = 'b';
    time_t aTime;
                                        //get sizes
    cout << "Enter size of hash table (use prime number): ";
    cin >> size;
    cout << "Enter initial number of items: ";
    cin >> n;
                                        //make table
    HashTable theHashTable(size);       //seed random numbers
    srand( static_cast<unsigned>(time(&aTime)) );

    for(int j=0; j<n; j++)              //insert data
    {
        aKey = rand()  % (2 * size);
        pDataItem = new DataItem(aKey);
        theHashTable.insert(aKey, pDataItem);
    }

    while(true)                         //interact with user
    {
        cout << "Enter first letter of ";
        cout << "show, insert, delete, find or quit: ";
        cin >> choice;
        switch(choice)
        {
            case 's':
                theHashTable.displayTable();
                break;
            case 'i':
                cout << "Enter key value to insert: ";
                cin >> aKey;
                pDataItem = new DataItem(aKey);
                theHashTable.insert(aKey, pDataItem);
                break;
            case 'd':
```

```
                cout << "Enter key value to delete: ";
                cin >> aKey;
                theHashTable.remove(aKey);
                break;
            case 'f':
                cout << "Enter key value to find: ";
                cin >> aKey;
                pDataItem = theHashTable.find(aKey);
                if(pDataItem != NULL)
                    cout << "Found " << aKey << endl;
                else
                    cout << "Could not find " << aKey << endl;
                break;
            case 'q':
                return 0;
            default:
                cout << "Invalid entry\n";
        }  //end switch
    }  //end while
    return 0;
}  //end main()
```

Zadanie 2 (C++). Uzupełnij w podanym poniżej kodzie ciała funkcji: insert(), remove(), find() oraz displayList(). Implementacja tablicy z haszowaniem ma na celu zastosowanie metody łańcuchowej (tzn. listy jednostronnie wiązanej).

[hashChain.cpp]

```
//hashChain.cpp
//demonstrates hash table with separate chaining
#include <iostream>
#include <vector>
#include <cstdlib>                      //for random numbers
#include <ctime>                        //for random numbers
using namespace std;

class Link
{                                   //(could be other items)
    public:
        int iData;                      //data item
        Link* pNext;                    //next link in list

        Link(int it) : iData(it)           //constructor
        {  }

        void displayLink()                 //display this link
        { cout << iData << " "; }

};  //end class Link
//--------------------------------------------------------------
class SortedList
{
    private:
        Link* pFirst;                   //ref to first list item

    public:
        SortedList()                       //constructor
        { pFirst = NULL; }
```

```
        void insert(Link* pLink)            //insert link, in order
        {
            int key = pLink->iData;
            Link* pPrevious = NULL;          //start at first
            Link* pCurrent = pFirst;

            /* do uzupełnienia */

        }  //end insert()

        void remove(int key)               //delete link
        {                                  //(assumes non-empty list)
            Link* pPrevious = NULL;       //start at first
            Link* pCurrent = pFirst;

            /* do uzupełnienia */

        }  //end remove()

        Link* find(int key)                //find link
        {
            Link* pCurrent = pFirst;        //start at first

            /* do uzupełnienia */

        }  //end find()

        void displayList()
        {
            cout << "List (first->last): ";
            Link* pCurrent = pFirst;     //start at beginning of list

            /* do uzupełnienia */
        }

};  //end class SortedList
//-----------------------------------------------------------------


class HashTable
{
    private:
        vector<SortedList*> hashArray;  //vector of lists
        int arraySize;

    public:
        HashTable(int size)                //constructor
        {
            arraySize = size;
            hashArray.resize(arraySize);    //set vector size
            for(int j=0; j<arraySize; j++)  //fill vector
                hashArray[j] = new SortedList; //with lists
        }

        void displayTable()
        {
            for(int j=0; j<arraySize; j++)  //for each cell,
            {
                cout << j << ". ";            //display cell number
                hashArray[j]->displayList(); //display list
            }
```

```
        }

        int hashFunc(int key)               //hash function
        {
            return key % arraySize;
        }

        void insert(Link* pLink)            //insert a link
        {
            int key = pLink->iData;
            int hashVal = hashFunc(key);    //hash the key
            hashArray[hashVal]->insert(pLink); //insert at hashVal
        }  //end insert()

        void remove(int key)                //delete a link
        {
            int hashVal = hashFunc(key);    //hash the key
            hashArray[hashVal]->remove(key); //delete link
        }  //end remove()

        Link* find(int key)                 //find link
        {
            int hashVal = hashFunc(key);    //hash the key
            Link* pLink = hashArray[hashVal]->find(key);  //get link
            return pLink;                   //return link
        }

};  //end class HashTable
//-----------------------------------------------------------------
int main()
{
    int aKey;
    Link* pDataItem;
    int size, n, keysPerCell = 100;
    time_t aTime;
    char choice = 'b';
                                    //get sizes
    cout << "Enter size of hash table: ";
    cin >> size;
    cout << "Enter initial number of items: ";
    cin >> n;
    HashTable theHashTable(size);   //make table
                                    //initialize random numbers
    srand( static_cast<unsigned>(time(&aTime)) );

    for(int j=0; j<n; j++)          //insert data
    {
        aKey = rand() % (keysPerCell * size);
        pDataItem = new Link(aKey);
        theHashTable.insert(pDataItem);
    }

    while(choice != 'x')            //interact with user
    {
        cout << "Enter first letter of ";
        cout << "show, insert, delete, find or quit: ";
        cin >> choice;
        switch(choice)
        {
            case 's':
                theHashTable.displayTable();
```

```
                break;
            case 'i':
                cout << "Enter key value to insert: ";
                cin >> aKey;
                pDataItem = new Link(aKey);
                theHashTable.insert(pDataItem);
                break;
            case 'd':
                cout << "Enter key value to delete: ";
                cin >> aKey;
                theHashTable.remove(aKey);
                break;
            case 'f':
                cout << "Enter key value to find: ";
                cin >> aKey;
                pDataItem = theHashTable.find(aKey);
                if(pDataItem != NULL)
                    cout << "Found " << aKey << endl;
                else
                    cout << "Could not find " << aKey << endl;
                break;
            case 'q':
                return 0;
            default:
                cout << "Invalid entry\n";
        }  //end switch
    }  //end while
    return 0;
}  //end main()
```

Opracowano na podstawie:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"