Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Мегафакультет компьютерных технологий и управления

Кафедра информатики и прикладной математики


Алгоритмы и структуры данных

# Лабораторная работа №1

# «Хеширование»

Группа: P3218

Студент: Петкевич Константин

Преподаватель: Зинчик А. А.

2017г

# Задание

Требуется написать программу, которая получает на входе набор идентификаторов, организует таблицу по заданному методу и позволяет осуществить многократный поиск идентификатора в этой таблице. Список идентификаторов считать заданным в виде текстового файла. Длина идентификаторов ограничена 32 символами.
Программа должна реализовывать комбинированный способ организации таблицы идентификаторов. Для организации таблицы используется простейшая хэш-функция, указанная в варианте задания, а при возникновении коллизий используется дополнительный метод размещения идентификаторов в памяти. Если в качестве этого метода используется дерево или список, то они должны быть связаны с элементом главной хэш-таблицы.
В каждом варианте требуется, чтобы программа сообщала среднее число коллизий и среднее количество сравнений, выполненных для поиска идентификатора.

Тип хеш-функции (таблицы) : сумма кодов первой, второй и последней букв.
Способ разрешения коллизий : список с простым перебором.

## 1. Схема организации хеш-таблицы

Для навигации по хеш-таблице используется хеш-функция, получающая на вход идентификатор и возвращающая индекс в хеш-таблице:

```
int getHashCode(const char* identifier){
  return (int(identifier[0]) + int(identifier[1]) + int(identifier[strlen(identifier)-1]));
}
```

Изначально же подразумевается, что идентификаторы находятся в отдельном файле, и функция, загружающая их в программу и создающая на их основе хеш-таблицу, имеет следующий вид:

```
hashTableEntry* getHashTable(const char* fileName, int size, int maxlength){
   char identifier[maxlength];
   hashTableEntry* hashTable = new hashTableEntry[size];
   int* numOfCollisions = new int[size];
   int counter = 0;

   for (int i = 0; i < size; i++){
     numOfCollisions[i] = 0;
     hashTable[i].next = 0;
   }
   ifstream identifiers(fileName);

   if (identifiers == NULL){
     return NULL;
   }
   int numOfElements = 0;
   while (identifiers){
     identifiers.getline(identifier, maxlength);
     numOfElements++;
     hashTableEntry* tableEntry = &hashTable[getHashCode(identifier)];
     while ((*tableEntry).next != 0){
       tableEntry = (*tableEntry).next;
     }

     if ((*tableEntry).next == 0){
       if (numOfCollisions[getHashCode(identifier)] == 0){
         counter++;
       }
       numOfCollisions[getHashCode(identifier)]++;
       hashTableEntry *entry = new hashTableEntry;
       (*entry).next = 0;
```

```
      (*tableEntry).next = entry;
      strncpy((*tableEntry).identifier,identifier,sizeof((*tableEntry).identifier));
     }
    }

    int sum = 0;
    int bigsum = 0;
    for (int i = 0; i < size; i++){
      sum += numOfCollisions[i];
      bigsum += getSumEx(numOfCollisions[i]);
    }

    cout << "Average number of collisions : " << sum/counter << endl;
    cout << "Average number of compares : " << bigsum/numOfElements << endl;
    return hashTable;
}
```

Где структура hashTableEntry определена как

```
 struct hashTableEntry{
  char identifier[32];
  hashTableEntry* next;
};
```

## 2. Описание алгоритма поиска в хеш-таблице

Поиск по таблице представляет собой простой перебор элементов связного списка, идентификаторы всех элементов которого имеют соответствующее значение, выдаваемое хеш-функцией. Перебор происходит до тех пор, пока не будет найден требуемый идентификатор, либо же достигнут конец списка (то есть, пока элемент списка еще связан с каким-либо другим).

```
hashTableEntry* searchEntry(const char* identifier, hashTableEntry* hashTable){
  int counter = 1;
  hashTableEntry* tableEntry = &hashTable[getHashCode(identifier)];
  while ((strcmp((*tableEntry).identifier,identifier) != 0) && ((*tableEntry).next != 0)){
    counter++;
    tableEntry = (*tableEntry).next;
  }
  if ((*tableEntry).next == 0){
    cout << "Not found" << endl;
    return 0;
  }
  cout << "Done " << counter << " compares " << endl;
  return tableEntry;
}
```

## 4. Вывод

Таким образом, хеш-таблицы являются довольно эффективным решением проблемы поиска среди элементов какого-либо множества до тех пор, пока количество элементов еще сопоставимо с размером таблицы (а именно, не более чем в несколько раз больше этого размера), и при условии, что хеш-функция подобрана так, что осуществляет равномерное (или в приемлемой степени близкое к равномерному) распределение элементов множества по ячейкам таблицы.