EX5

1. Introduction. This is a literate program which solves the problem set in lab five—Hashing with chaining.

We will start with the, by now familiar, outline of the program.

```
⟨ Headers 4 ⟩
using namespace std;
⟨ Global variables and types 6 ⟩
⟨ Prototypes for the functions 9 ⟩
⟨ The main program 2 ⟩
⟨ Implementation of the functions 10 ⟩
```

2. Main. Ok, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 2 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 3 ⟩
    ⟨ Open and validate the input file 5 ⟩
    ⟨ Store values into the hash table 7 ⟩
    ⟨ Report the results 13 ⟩
    ⟨ Finish and clean up 14 ⟩
  }
```

This code is used in section 1.

3. The first thing we need to do is declare the variables we need to input data. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 3 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also section 8.

This code is used in section 2.

4.

```
⟨ Headers 4 ⟩ ≡
#include <iostream>
#include <fstream>
```

This code is used in section 1.

5. Right—now we can get the file opened, ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open and validate the input file 5 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 2.

6. Before we start reading the file we need to declare the data structure we are going to use to store our hash table nodes. The **bool** variable, *empty* is used to track whether there is data stored in the current node. If *empty* is *false* when we attempt to store a value in our hash table we wil need to use overflow chains to enable us to store the extra data.

While we are here we will also define some global variables. We start with an array of *hashNode* which is the starting *hashArray* and has size `HASH_SIZE`. In addition to this we need a couple of counters; *emptyCount*, which will track the number of unused entries in *hashArray*, and *longest* which will record the length of the longest chain.

```
⟨ Global variables and types 6 ⟩ ≡
  struct hashNode {
    int value;
    bool empty
    {true};
    hashNode *next
    {nullptr};
  };
  const int HASH_SIZE = 100;
  hashNode hashTable[HASH_SIZE];
  int emptyCount = HASH_SIZE;
  int longest = 0;
```

This code is used in section 1.

7. Store values in the table. We are now ready to do the main input loop. we can read the integer values from the input file and store them in the hash table using the function, *tableInsert*.

```
⟨ Store values into the hash table 7 ⟩ ≡
  while (fin ≫ input) {
    tableInsert(input);
  }
```

This code is used in section 2.

8. We need to add *input* to our variables.

```
⟨ Variables of main 3 ⟩ +≡
  int input;
```

9. Ok—let's actually implement the functions as we need them this time. First the prototype.

```
⟨ Prototypes for the functions 9 ⟩ ≡
  void tableInsert(int);
```

See also section 11.

This code is used in section 1.

10. And then the implementation. The function should determine the appropriate storage location and, if this is empty store the *incoming* value.

In the event that this entry is already used we need to follow the chain to store the *incoming* value. We will do this using the recursive function *chainInsert*. The 1 as the last argument of *chainInsert* is used to track the length of the chain at this hash value.

```
⟨ Implementation of the functions 10 ⟩ ≡
  void tableInsert(int incoming)
  {
    int hash = incoming % HASH_SIZE;
    if (hashTable[hash].empty) {
      hashTable[hash].value = incoming;
      hashTable[hash].empty = false;
      emptyCount −−;
      if (longest ≡ 0) longest = 1;
    }
    else hashTable[hash].next = chainInsert(incoming, hashTable[hash].next, 1);
    return;
  }
```

See also section 12.

This code is used in section 1.

11. Follow the chain. This is where things get interesting (and inefficient) as we are goint to use a linked list of **hashNode** to store the overflow values. We implement this process with the recursive *chainInsert* function.

⟨ Prototypes for the functions 9 ⟩ +≡

```
  hashNode *chainInsert(int, hashNode *, int);
```

12. The implementation of the function involves following the chain down until the *current* node is empty. At this point we create a new node, and store the *incoming* value into it. Note: we do not need to mess with *empty* as any non-null **hashNode** will contain valid data automatically.

⟨ Implementation of the functions 10 ⟩ +≡

```
  hashNode *chainInsert(int incoming, hashNode *current, int chainLength)
  {
    if (current ≡ nullptr) {
      current = new hashNode;
      current→value = incoming;
      if (chainLength ≡ longest) longest++;
    }
    else current→next = chainInsert(incoming, current→next, ++chainLength);
    return current;
  }
```

13. Back to main. All that remains is to report the statistics fot the hash table:

```
⟨ Report the results 13 ⟩ ≡
  cout ≪ "The␣number␣of␣empty␣slots␣in␣the␣hash␣array␣=␣" ≪ emptyCount ≪ endl;
  cout ≪ "The␣longest␣chain␣has␣a␣length␣of␣" ≪ longest ≪ endl;
```

This code is used in section 2.

14. We should also close our input file.

```
⟨ Finish and clean up 14 ⟩ ≡
  fin.close( );
  return 0;
```

This code is used in section 2.

15. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.