Algorithms & Data Structures

School of Life Sciences, Engineering & Design

Topic:	**Dictionaries**	Lesson:	**7**
Lecturers:	***Robert de Groote*** ***Felix Hartlieb*** ***Dawid Zalewski***	Revision:	**May 12, 2023**

Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.
Fred Brooks, The Mythical Man Month, 1975

Learning Objectives

After successfully completing this activity a student will be able to:

- Name examples of dictionaries/maps appearing in the real life.
- Name operations supported by a dictionary.
- Use generic algorithms with function pointers in C.
- Explain the concept of a 'pair' in C.
- Implement a simple dictionary in C.

Activities

Counts and frequencies

Calculating the frequency distribution of items is a method commonly applied to get an overview of some situation or to make quick comparisons. What are the three most popular car models in a country? To answer this, calculate the models' frequencies, sort from the greatest to the smallest and take the top three. Which class in a school performs the best? Put the grade frequency distributions of all the classes next to each other and the answer becomes clear.

But there is more! In computational linguistics, the knowledge of frequency of words in a document can be used to determine its type (legal vs. technical), its difficulty level ("thieves were *apprehended*" vs. "thieves were *arrested*") or even its author. Surprisingly, counting letters' frequency has also its uses – every language can be uniquely characterized by the most common letters appearing in texts written in this language, see Wikipedia and letterfrequency.org.

Based on the most commonly occurring letter in each language, it's possible to come up with a *language signature*. For the purpose of this activity, let a *language signature* be defined as a string consisting of the six most common letters in a given language. For instance, in English the most common letters sorted by their frequencies are: *e, t, a, o, i, n, s, h, r, d, l, c, ...*. The language signature for English is then: *etaoin*. Below you can see the *language signatures* of some European languages:

Language	Signature
English	etaoin
French	esaitn
German	ensria
Spanish	eaosrn
Dutch	enatir
Finnish	aintes

Counting letters

Counting letters in a string doesn't require any specialized data structures. Let's assume that a string uses ASCII encoding and the English alphabet (letter from 'a' to 'z'). There are 26 letters in the English alphabet, thus a simple array of 26 integers is enough to implement a letter counter. An element at index `c - 'a'` of this array corresponds to the count of a letter stored in the variable `char c`.

Activity 1: The ctype header file

Search the internet for information on the C header file `ctype.h`. Which functions are useful when processing alphabetic characters? In your logbook, list the prototypes of these functions, and provide a short description of what they do.

Activity 2: Count letters in a string

Open the provided `dictionaries` CMake project. This project contains two subprojects: `letter_counter` and `word_counter`.
In the `letter_counter` directory, locate the function `count_letters`, and read its description (in `letter_counter.h`). The function takes an array `freqs` of (at least 26) `unsigned long`s and a null-terminated string (i.e., `const char*`) as its arguments.

The function must update the `freqs` array by adding the counts of the letters in the string to it, and returns the total number of letters it counted. Counting is not case-sensitive. Remember to ignore all non-letter characters in `str`.

Implement the function, and test it by running the code given below. To make your life easy, use the functions of the `ctype.h` header file.

```c
unsigned long count_letters(unsigned long counts[static 26], const char* str)↩
    ;

int main(void){
  unsigned long counts[26] = {0};
  const char* text = "The Quick Brown Fox Jumps Over The Lazy Dog.";
  unsigned long total = count_letters(&counts[0], text);
  assert(35 == total);
  printCounts(&counts[0], false, true);
}
```

Make sure to test your code - after executing the above piece of code the printed `counts` (note that they are printed in order of decreasing frequency) must be the following:

↪ *Activity continues on the next page.*

- Character ’o’ appears four times,
- Character ’e’ appears three times,
- Characters ’t’, ’r’, ’u’, and ’h’ each appear two times,
- All other characters appear exactly once.

With the `count_letters` function implemented, it’s time to use it to recognize the language of a piece of text! There are four files named `aliceX.txt` where `X` is a number from 0 to 3. Each of them is the text of the Lewis Carroll’s book *Alices Adventures in Wonderland*. Process each of those files, counting the letters using the `process_file` function. This function returns a dynamically allocated array `counts` by pointer – it is the task of the caller of this function to perform clean-up by calling `free` on the returned pointer.

```c
unsigned long* process_file(const char* fileName) {

  // initialize with 0's
  unsigned long * counts = calloc(sizeof(unsigned long[ALPHABET_SIZE]), sizeof(↩
      unsigned long));

  FILE* fp = fopen(fileName, "r");
  unsigned long sum = 0;
  if (fp) {
    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), fp)) {
      sum += count_letters(&counts[0], buffer);
    }
    printf("In total counted %ld letters.\n", sum);
  }
  return counts;
}
```

The only remaining puzzle piece is a function that will create a *language signature* from an array with letter counts.

Activity 3: Recognizing languages

Part 1:
Implement the function `make_signature` (see the prototype given in `letter_counter.h`) that creates a six-letter *language signature* from a letter counts array. A *language signature* consists of the six most frequent letters ordered from the most to the least frequent.

Use the code listed below to test your function.

```c
const char* make_signature(unsigned long counts[static 26]);

int main(void) {
    unsigned long counts1[26] = ↩
        {15,3,4,5,16,6,7,8,9,7,6,3,2,11,14,1,2,12,13};
  unsigned long counts2[26] = ↩
      {16,4,7,5,20,7,4,3,14,5,9,1,2,18,6,12,9,13,9,15};
  assert(strcmp("eaosrn", make_signature(counts1)) == 0);
  assert(strcmp("enatir", make_signature(counts2)) == 0);
}
```

Notice that this function returns a pointer to an array defined locally within a function.

↪ Activity continues on the next page.

Normally, this is illegal and dangerous. However, `signature` is a `static` variable, which means that's it allocated in the global program memory and guaranteed to exist throughout the lifetime of the program. The danger of using such a construct is that the signature returned by `make_signature` is valid only until the next time this function is called. Moreover, returning a pointer from a function usually has the meaning of transferring ownership of a dynamically allocated object (like for the function `process_file`). Here, the `make_signature` function remains the owner of `signature` (and it's illegal to call `free` on the pointer returned by this function). Always document such uncommon behaviors (they also happen in the standard C library, see for example the function local_time.)

Part 2:
Use the functions `process_file`, `make_signature`, and `match_language` (*language_recognizer.h* header) to automatically recognize the languages the `aliceX.txt` files are written in, by:

1. Retrieve the letter counts by calling the function `process_file`,
2. Create a language signature using the function `make_signature`,
3. Match the language using the function `match_language`.

Put the results into the table below:

File	**Signature**	**Language**
`alice0.txt`		
`alice1.txt`		
`alice2.txt`		
`alice3.txt`		

Dictionaries around us

The letter counter that you implemented in the previous part demonstrates a common task in programming - linking one value (a letter) with another (its count). Such a pair of values is called a *key-value pair*. Here, letters were the keys, and their counts were the values that were linked to them. For counting letters it was sufficient to use an array of `unsigned long` integers. The reason for this is that the set of possible *keys* was limited to the 26 letters of the English alphabet. Moreover, there exists a straightforward way to map letters to indices in an array: the letter `a` maps to index 0, `b` to index 1, and so on. In other words, the *key* can be used as an *index* into the array, so that the value associated with the key can be found in constant time.

This is not always the case - more often than not, keys are *not* known upfront, and the set of possible keys is not limited to a small number of values, but can be infinite. To deal with those cases a data structure called a *dictionary* is used.

A *dictionary*, also known as an *associative array* or a *map*, is an abstract data structure that maps keys to values. Just like a *set*, every *key* in a dictionary is unique (i.e., can appear only once). The various names of this data structure reflects the way its functioning can be interpreted. It is:

A *dictionary*, because a key and associated value can be seen as a word and its corresponding definition in a language dictionary.

An *associative array*, because each *key* is associated with a *value*.

A *map*, because each *key* is mapped to a *value* (More formally, there exists a unidirectional, one-to-one correspondence between keys and values).

Real-life examples of ditionaries are for instance: language dictionaries, which link a word in one language to its translation in another language; telephone books, which link a person's name to their phone number; and address books, which link a person's name to their address.

Dictionary data structure

Dictionaries support at least the following four operations:

- `insert(key, value)`: inserts a new key-value pair into the dictionary.
- `remove(key)`: removes the key-value pair with the given key from the dictionary.
- `contains(key)`: checks whether the dictionary contains a key-value pair with the given key.
- `lookup(key)`: retrieves the value associated with the given key.

Other operations like getting the size of a dictionary, or iterating over its entries can also be supported. As said before, the keys stored in a dictionary are unique - each key occurs only once.

Dictionaries in programming languages. Most high-level programming languages provide built-in dictionary data structures either as a part of the language or as a type included in the language's standard library. For instance, in the *C++* standard library a dictionary is included as `std::map` and `std::unordered_map`.

Activity 4: Find out: dictionaries in other languages

Find out how a *dictionary type* is called in at least five other programming languages (including *Python* and *C#*).
For two of them, list the functions that:
- insert a *key-value* pair,
- delete a *key*,
- check if a *key* exists,
- retrieve the *value* associated with a *key*.

Word counter

Let's say that instead of counting letters in a text document, we would like to count words. To do this, we need a dictionary data structure that maps *strings* to *integers*. The C programming language doesn't have a specialized dictionary data structure. This is partly due to its lack of support for *generic typing*. Because C doesn't offer one, we will need to build a dictionary ourselves.

Key-value pairs

Fortunately, we can reuse the *set* data structure for this, by letting it store *key-value pairs* instead of simple types like `double`s or `int`s. Our set data structure should thus work with pairs of `const char*` and `unsigned long`, defined as follows:

```c
typedef struct pair {
  const char* key;
  unsigned long value;
} pair_t;
```

Now that we have a definition for a key-value pair, we next need to store them in a set. Fortunately, a set is already provided in the source code, and can be found in the `set.h` and `set.c` files. The set is a bit different from the set that you've created in the previous week. In particular, it's *generic*, which means that it can store items of any type.

The following set structure is provided in `set.h`:

```c
typedef struct set {
  compare_fun_t cmp_fun; // function for comparing keys
  void *data; // dynamic generic array
  size_t item_size; // size of each item in data
  size_t capacity; // current capacity of data
  size_t count; // current nr. of items in data
} set_t;
```

Here, the interesting member is `data`, which is a so-called `void` or *generic* pointer. The primary use of these `void` pointers is to pass around addresses to data whose type is unknown. For example, if a function needs to manipulate a pointer to some unknown data type, you can declare the function parameter as a void pointer. This allows the function to accept a pointer of any data type.

The `set_t` structure allows data of any type to be stored, as long as the values that need to be stored have a fixed size - specified in the `item_size` member of the `set_t` structure.

The provided set data structure stores its items in a (dynamic) array, which is kept sorted at all times by using binary search to find the right insertion position for new items. This means that a comparison function must be passed to the set on initialization. The declared type of this comparison function is also generic - it accepts generic pointers to items it needs to compare. The type of the comparison function is defined (see `set.h`) as follows:

```c
typedef int (*compare_fun_t)(const void*, const void*);
```

Note that although the function receives its arguments as generic pointers, its implementation can use *type casting* to obtain a pointer to a specific data type. As an example, here's how the set can be used to work with strings:

```c
typedef const char* string_t;

int compare_strings(const void* a, const void* b) {
  return strcmp((string_t) a, (string_t)b);
}

int main() {
  set_t set;
  set_init(&set, 10, sizeof(string_t), compare_strings);
  string_t s1 = "Hello", s2 = "World";
  set_add(&set, &s1);
  set_add(&set, &s2);
  assert(set_contains(&set, &s1));
  assert(set_contains(&set, &s2));
  set_destroy(&set);
}
```

The code uses the `set_init` function to initialize a set, with a default capacity of 10 items, each item being a `string_t` value, using the `compare_strings` function to compare two items. It then uses the `set_add` function to add the two given strings to the set, and asserts that they're present in the set using the `set_contains` function.

Using the set as a dictionary

The generic set described previously can be used to implement a dictionary, which maps strings to integers. To achieve this, we simply need to store our key-value pairs in the set, and define a comparison function that compares the keys of two pairs.

Since we're using a dictionary for a specialized task – the counting of strings, we can make some simplifications to the dictionary operations. Our "counting" dictionary must support the following operations:

- Check whether a given key is present in the dictionary. Note that the set already provides a function (`set_contains`) for this, but it expects us to pass a pointer to the key-value pair we're looking for. Instead, we'd like to pass a pointer to the key only.
- Update the value associated with a given key. The set does not provide a function for this. Since we're using the dictionary to count words, this means that we merely need to increment the value associated with a given key.
- Obtain the value (i.e., count) associated with a given key. This function is also not offered by the set, so we'll need to implement it ourselves.

The file `counter.h` contains the declarations of the functions that must implement these operations. It's your task to implement these functions in `counter.c`. Before you start, we'll dive a bit more in the use of generic algorithms and function pointers in C. This will help you implement the dictionary's operations later.

Working with generic algorithms

In the previous weeks, you've worked with different kinds of data structures, storing values of different types, such as `double`s and `int`s. Data structures that are limited to storing data of a specific type only are quite restrictive. Therefore, when creating a data structure, it should be made as generic as possible, so that values of any type can be stored in it. For the same reason, *algorithms* should be as generic as possible - a sorting algorithm that is only capable of sorting linked lists that contain `double`s is not very useful.

Quick sort The C standard library offers a *sorting function* that implements the *quick sort* algorithm. The function is named `qsort` and can sort any *array* of values. In order to do this, it needs to know the size of the elements in the array, as well as the number of elements in the array. Furthermore, it needs to know how to compare two elements.

The function signature of `qsort` is as follows:

```c
void qsort(void* base, size_t num_items, size_t item_size, int (*compare)(const ↩
    void*, const void*));
```

Here, `base` is a pointer to the first element in the array, `num_items` is the number of elements in the array, `item_size` is the size of each element in the array, and `compare` is a pointer to a function that compares two elements. The function pointed to by `compare` should return a negative number if the first element is smaller than the second element, zero if the elements are equal, and a positive number if the first element is larger than the second element.

The next activity serves to let you work with generic functions, such as the comparison function that is passed to `qsort`.

Activity 5: Generic sorting in C

In the `main` function of `generic_functions.c`, which is part of the CMake target "generic_functions", you'll find a variable declaration for an array of *strings*. Write a function that compares two strings, and use it to sort the array of strings using `qsort`. Print the sorted array to the console to verify that it is sorted. In your log book, include the comparison function you wrote, your `main` function, and the output of the program.

Binary search Another generic function that is provided by the standard library is `bsearch`, which implements the *binary search* algorithm you've implemented in the previous week. However, this function does not return the insertion index in case the searched-for item is not found. Also, direct use of the `bsearch` function is deprecated since C23, so you should not use it in your own code.

Instead, the provided code contains a generic implementation of binary search, which you can use in your own code. This implementation can be found in `binsearch.h`. Its signature is as follows:

```c
bsearch_result_t binsearch(
  const void* needle,
  const void* haystack,
  size_t item_count,
  size_t item_size,
  int (*compare)(const void*, const void*));
```

The function takes a pointer to the item that is being searched for (`needle`), a pointer to the array in which the item is being searched (`haystack`), the number of items in the array (`item_count`), the size of each item (`item_size`), and a pointer to a comparison function (`compare`). The function returns a `bsearch_result_t` structure, which has two members: `found` and `index`. The former is a *boolean* value that indicates whether the item was found in the array, and the latter is the index at which the item was found, or at which it must be inserted if it is not found. The set data structure that you'll be working with in this week uses this function to find the right insertion position for new items.

Activity 6: Generic searching in C

In the `main` function of `generic_functions.c`, use the `binsearch` function to search for the following strings: "Holy Grail", "Parrot", and "Rabbit" (after the array was sorted). For each string, print its index in case it was found, otherwise print a message indicating its absence. In your log book, include the lines you added to your `main` function, and the output of the program.

Implementing the dictionary

The file `counter.h` contains the declarations of the functions that must implement the dictionary's operations:

```c
// Returns the count associated with the given string,
// or 0 if the string is not present
size_t counter_get_count(const counter_t *counter, const char *string);

// Looks up the string in the counter, and increments its associated count (by one)
// if it was found. Otherwise, adds string to the counter with an associated
// count of one.
void counter_increment(counter_t *counter, const char *string);

// Returns the address of the pair_t value stored in the counter's data array,
// at the given index.
const pair_t * counter_get_pair_at(const counter_t *counter, size_t index);

// Initializes a counter so that it works with pair_t values
counter_t *counter_init(counter_t *counter, size_t capacity);

// Destroys the counter, freeing all memory that was allocated for it
void counter_destroy(counter_t *counter);
```

Some of these functions can be implemented by delegating most of the work to the `set_t` data structure. For others, you'll need to reuse code implemented by the set functions, and/or implement new code.

Make sure to take the *lifetime* of the strings that are added as keys to the counter - since most likely these strings are allocated on the stack, you'll need to copy them to the heap before adding them to the counter (a convenient helper function that might be of good use here is the `make_pair` function). This also means that when destroying a `counter_t`, you'll have to release the memory that was allocated for the strings as well.

Activity 7: Counter - function implementations

Some of the functions of the dictionary are already implemented in the *counter.c* file. However, some functions still have an empty body are or only partially given. Complete all functions in `counter.c`, and test your implementation by running the `test_counter` target of the CMake project. Note that most of the functionality you need to implement is already provided by the functions listed in `set.h`, and by the `binary_search` function that you used earlier. For example, the `set_add` function allows you to add a new pair to the set. Internally, the `set_add` function calls the `binary_search` function to determine the correct location for the new value.

Make sure to use the `make_pair` function to copy the strings into the counter. This function allocates memory for storing the pair's key (i.e. the string), which means you need to free this memory when the counter is destroyed (this happens in the `counter_destroy` function). Use an address sanitizer to check for memory leaks. Once your functions pass all tests as well as the address sanitizer's checks, include the code you wrote in your log book.

Counting words in a file

It looks like, with the `counter_t` ready, we can finally use it to see how many times each word occurs in a text file! There is only one piece of the puzzle missing – a function that breaks a string into separate words (and increments the counter for each of them). Instead of doing it manually by scanning a string for the separators (characters that indicate a word boundary), we will use the facilities provided by the C standard library. This can be written as:

```c
void process_line(counter_t* counter, char* line) {
  // separator characters
  const char* sep = "\t (),.;:?!\"\r\n'_-*";

  char* token = strtok(line, sep);
  while (token) {
    if (strlen(token) > 0) process_word(counter, token);
    token = strtok(NULL, sep);
  }
}
```

Curiously, `process_line` doesn't take the `line` argument as a pointer to `const char`. This is because it modifies the string while tokenizing it! Specifically, the strtok function does it by...

Activity 8: Find out: function strtok

It's time for you to find out how the `strtok` function works! Describe what you discovered. Moreover, write down explanations of what each line of code in the snippet above does.

It's time to finish writing the program. Its primary function opens a file and reads it line by line, processing each line with the `process_line` function:

```c
bool process_file(const char* filename, counter_t * counter) {
  FILE* fp = fopen(filename, "r");
  if (fp != NULL) {
    char buffer[1024];
    while (fgets(buffer, sizeof(buffer), fp)) {
      process_line(counter, buffer);
    }
    return true;
  }
  return false;
}
```

Activity 9: How many words?

Write some code in the `main` function, using the `counter_t` type and its functions, as well as the `process_file` function, to answer the following two questions:

1. How many times do the following words occur in the "alice0.txt" document: "Alice", "the", "rabbit"?
2. How many *unique* words are there in the "alice0.txt" document?

In your log book, include the code you wrote to answer the questions, and the answers you got.

Now that you have a working program to count word frequencies, you can use it to determine the most frequent words in a text file. To do this, however, you'll need to do some more sorting work. Note that the `counter_t` stores its key-value pairs in a sorted array, but the sorting is done on the keys (i.e., the words), not on their counts. This means that you'll need to sort the array of pairs by their counts, in descending order, and then print the first few pairs.

Activity 10: Most frequent words

Make a copy of the counter's data array, by copying its contents into a dynamically allocated array of key-value pairs. Use the `memcpy` function (see the C standard library documentation for details, or inspect the code in the `set.c` file) to copy the contents of the counter's data array into the new array.
Next, use the `qsort` function to sort the copied key-value pairs, in such a way that the pairs with the highest counts are at the beginning of the array.
Finally, let your program print the 10 most frequent words, along with their counts. In your log book, include the code you wrote to obtain the top 10 of most frequent words, plus the output of the program.