Piscine — Tutorial D9

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

1 `void` genericity

1.1 Generic linked lists

Normally, you have just implemented a linked list containing integers. But what if we wanted to store not only integers, but other types of data? How can we create a linked list that can be used for any data type?

In *C*, you can use `void` pointers to point to any data type. We also know the size of a pointer, so we can always know the size of the structure to allocate it. Thus, we can represent a generic linked list like:

```
struct list
{
    // Any data type can be stored in this node
    void *data;
    struct list *next;
};
```

1.2 Exercise: Generic Void List

1.2.1 Goal

In this exercise, you will have to implement a generic linked list, along with its manipulation operations.

Be careful!

An empty list is represented by a `NULL` pointer.

1.2.2 `list_prepend`

- **Authorized functions:** `malloc(3)`, `calloc(3)`, `memcpy(3)`

```
struct list *list_prepend(struct list *list, const void *value,
                          size_t data_size);
```

This function must insert a node containing `value` at the beginning of the list. It must return the new list, or `NULL` if an error occurred. You can use `memcpy(3)` to copy `value` into the `data` field of the list structure.

1.2.3 `list_length`

- **Authorized functions:** none.

```
size_t list_length(struct list *list);
```

This function returns the length of the list.

1.2.4 `list_destroy`

- **Authorized functions:** `free(3)`

```
void list_destroy(struct list *list);
```

This function releases all the memory used by `list`.

2 Hash table

2.1 Definition

A hash table (or hash map) is a data structure used to implement key/value associations (called dictionaries or maps). These types of data structures are called *unordered collections* as there is no particular order in which the data is stored (in contrast to arrays or lists).

A hash table uses an array to hold its elements. The size of the array is fixed and cannot be changed. The position of an element in the table is calculated according to the *hash* of its *key*. The hash is computed by a *hash function* that takes the *key* as argument and returns a *unique identifier* that is used to compute the value's *position* in the underlying array.

A hash set is the implementation of a mathematical set using a hash table. As in a set, there is no key/value association, the key and value within a hash set is the same thing: the data type is used as key **and** value.

This change impacts the respective use of each data structure: the hash table is used to store and retrieve elements by their key, whereas the set's use case is to store the presence or absence of an element.

2.1.1 Example

Let `S` = { `"acu"`, `"epita"` } be the data to store in our set.

By associating each letter with its corresponding index in the alphabet (i.e. `a = 1`, `b = 2`, etc.), let us define the hash function `hash(x)` that computes the sum of the letters of the word, adds the number of letters and applies the modulo `n`, `n` being the size of the array (`6`, in this example).

If we apply the function to our data, we have:

- `hash("acu")` = ((1 + 3 + 21) + 3) % 6 = 28 % 6 = 4
- `hash("epita")` = ((5 + 16 + 9 + 20 + 1) + 5) % 6 = 56 % 6 = 2

1	
2	epita
3	
4	acu
5	
6	

In order to check if an element `x` exists in the set, you can compute the result of `hash(x)` and check if the corresponding cell in the table is empty.

If it is, then the element is not in the set.

Another common implementation for sets and maps are self-balancing trees (red-black, AVL, etc.) but it is not today's subject.

2.2 Collisions

In an ideal world, there is a unique hash value for each possible value and our array is big enough to hold all possible values. But in practice, this is almost never the case. There **will** be cases where two different values will have the same hash. This situation is called a **primary collision**.

Let us take the results from the previous example and insert a new element:

- `hash("acu")` = ((1 + 3 + 21) + 3) % 6 = 28 % 6 = 4
- `hash("epita")` = ((5 + 16 + 9 + 20 + 1) + 5) % 6 = 56 % 6 = 2
- `hash("c")` = ((3) + 1) % 6 = 4 % 6 = 4

1		
2	epita	
3		
4	acu	c
5		
6		

There are several ways to fix this problem:

- Resolution by *separate chaining*: linked list chaining, coalesced hashing, ...
- Resolution by *open addressing*: linear probing, double hashing, cuckoo hashing, ...

Furthermore, some hash tables resize themselves when overloaded. We will see all these possibilities here.

2.3 Efficiency

On average, looking up a value in a hash set is done in constant time: `O(1)`, which is most of the time faster than the lookup in a standard array or a linked list. However, if the hash function is inefficient and many collisions occur, the lookup time in the worst case is in linear time:`O(n)`.

Hash sets are obviously very efficient compared to other data structures, especially when the size of the underlying array is optimal and the hash function is efficient.

The efficiency of your hash set is heavily affected by the efficiency of your hash function.

2.4 Some details about the hash function

For any element `x`, the result `hash(x)` is used to find the index of `x` in the table. The index is used for direct access to the element.

The hash function `h` must:

- Provide a *uniform distribution* of hash values.
- Be *deterministic*.
- Take a very *small amount of time*, the complexity of the hash function should not affect the complexity of the hash table (ideally (`O(1)`).
- Use the inherent characteristic of the element as much as possible.

A good hash function is very difficult to write, which is why we usually rely on standard libraries to solve this problem for us.

2.5 Collision resolution techniques

2.5.1 Open addressing

With the open addressing technique, when there is a hash collision, the index is changed until a place is found for the element to insert. There are several ways to find the new index:

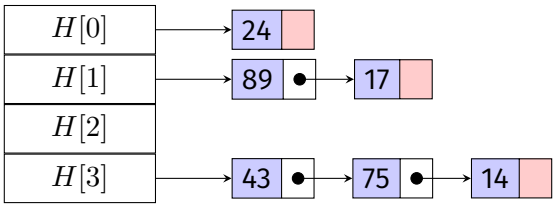
- Linear probing: the index is incremented by a fixed value (usually 1).
- Quadratic probing: the index is incremented by a quadratic function (usually x^2: 0, 1, 2, 4, etc.).
- Double hashing: the increment is given by a second hashing function.

It is important to note that with this method, it is not possible to have more values than the size of its array, and once the array has reached around 70% of its capacity, the performance of open addressing falls down drastically.

The only way to prevent the table being full is to resize the underlying array. The basic implementation is to reallocate the array and reinsert all the elements in the new array by recalculating their hashes taking account the size of the new array. A more complex implementation uses two arrays at the same time. Search is done on both of the arrays, removal is done on the old one and insertion is done on the new one. Once the old array goes empty, it is deallocated.

2.5.2 Separate chaining

In the separate chaining technique, each element of the array is a collection of elements, allowing collisions to be added to the collection. The easiest implementation is by using a linked list:



However a self-balancing tree might also be used to improve performance. The insertion and search is therefore done on the collection at the hash's index.

2.6 Exercise: Hash Map

2.6.1 Goal

In this exercise, you will implement a hash map (a.k.a. a dictionary).

> *"In computing, a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found. During lookup, the key is hashed and the resulting hash indicates where the corresponding value is stored."*
>
> ---Wikipedia https://en.wikipedia.org/wiki/Hash_table.

The file `hash_map.h` provides function prototypes as well as data structure definitions. The hash function is in the provided `hash.c` file. You are highly encouraged to change the hash function to see its effects on the data structure's performance.

```
struct pair_list
{
    const char *key;
    char *value;
    struct pair_list *next;
};

struct hash_map
{
    struct pair_list **data;
    size_t size;
};
```

Be careful!

- You **MUST** use the above structures as is.

hash_map_init

- **Filename:** `hash_map.c`

```
struct hash_map *hash_map_init(size_t size);
```

Returns an empty `hash_map` of size `size`. If an allocation fails, returns `NULL`.

hash_map_insert

- **Filename:** `hash_map.c`

```
bool hash_map_insert(struct hash_map *hash_map, const char *key, char *value,
                     bool *updated);
```

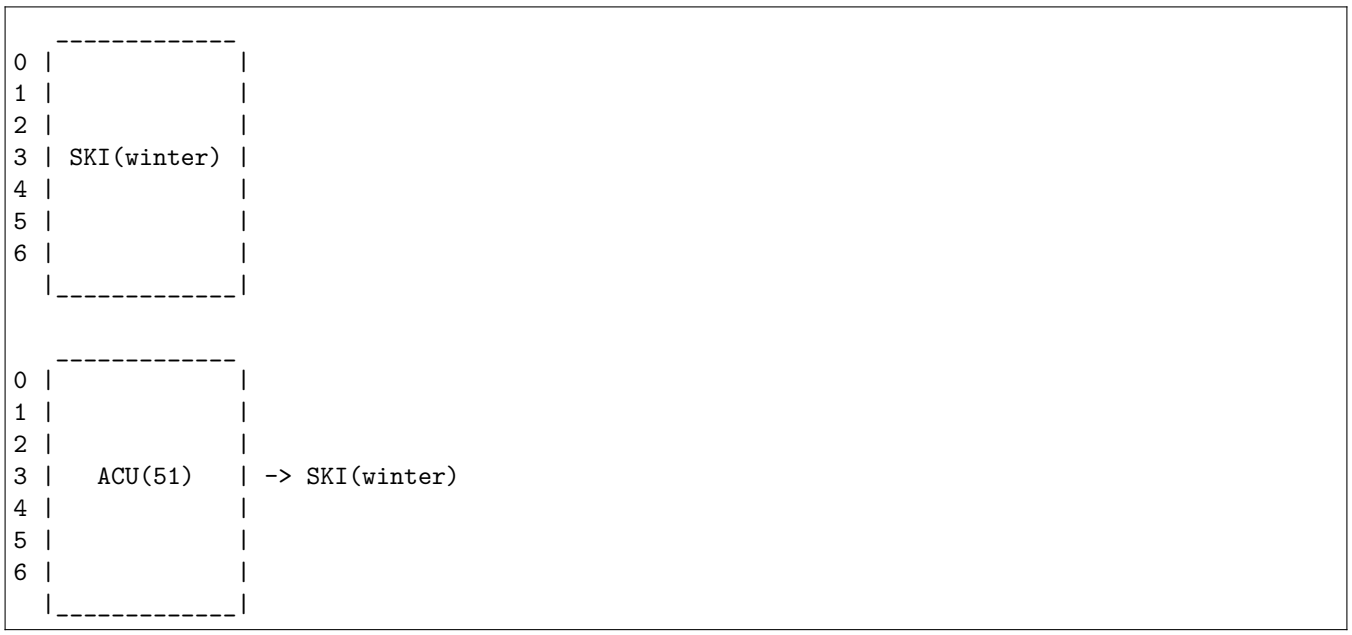
Inserts the `key`/`value` pair in `hash_map`. If two keys have the same hash value, you are expected to resolve the conflict by chaining them. Meaning you must use a linked list for the colliding elements and add the new element at the head of the list.

If this key is already present in the structure, the old value associated with the key is to be updated and the value held by `updated` must be set to `true`. Otherwise, it must be set to `false`. If an allocation fails, the function should return `false` and the value held by `updated` should not be changed. `updated` might be `NULL`.

Finally, if the computed hash value is higher than the map size, you should use the remainder of the computed hash and the map's size.

Example

If we insert `"SKI"` and then `"ACU"`, there is a collision, and the two elements need to be chained:



hash_map_free

- **Filename:** `hash_map.c`

```
void hash_map_free(struct hash_map *hash_map);
```

Free all resources allocated by `hash_map`, including the pointer passed to the function.

hash_map_dump

- **Filename:** `hash_map.c`

```
void hash_map_dump(struct hash_map *hash_map);
```

Prints on the standard output the whole dictionary. The elements are to be printed in the same order as they appear in the array and each list must be separated by a line feed. When one of the lists is empty, you must skip it.

In case of a slot containing multiple key/value pairs, each pair must be separated by a comma followed by a space. Note that the comma should not appear at the end of the line.

In case of an empty list, it must be ignored.

Last but not least, each pair must be printed as follows:

```
key: value
```

Example

Let's suppose our dictionary looks like the following one (values are within parentheses) and that `"ACU"` was added after `"SKI"`:

```
   -------------
0 |             |
1 |   TARAN(W)  |
2 |             |
3 |   ACU(51)   | -> SKI(winter)
4 |             |
5 |    C(42)    |
6 |   42(life)  |
  |-------------|
```

A call to `hash_map_dump` should print the following result on the standard output:

```
42sh$ ./hash_map_dump | cat -e
TARAN: W$
ACU: 51, SKI: winter$
C: 42$
42: life$
```

hash_map_get

- **Filename:** `hash_map.c`

```
const char *hash_map_get(const struct hash_map *hash_map, const char *key);
```

Return the value associated to `key` in `hash_map`.

Return `NULL` if the key was not found.

Example

```
hash_map_insert(my_hash, "test", "42", NULL);
/* do some clever stuff */
hash_map_get(my_hash, "test"); // == "42"
```

hash_map_remove

- **Filename:** `hash_map.c`

```
bool hash_map_remove(struct hash_map *hash_map, const char *key);
```

Remove `key` and its associated value from `hash_map`. You have to free the removed `pair_list` but not its contained `value` and `key`.

Return `false` if the key was not found, `true` otherwise.

3 Regular expressions

Regular expressions, called *regexp* or `regex(7)`, are an essential support for many UNIX utilities such as `grep(1)` and `sed(1)`, that we will study later today.

A *regex* is a pattern that describes a set of strings. Regexes are built like arithmetic: they use different operators to combine smaller expressions.

There are two types of regular expressions: the *basic* ones and the *extended* ones. Most utilities offer one or the other through command line options. The difference between these two forms is primarily whether or not to prefix certain constructions with a *backslash* as we will see below.

A regular expression is mapped to a string character by character.

Tips

In the examples, we use the `=~` operator (it is a common notation used in Bash, Perl, Ruby and other languages), which means a correspondence test (*a match*): the left term being the test string and the right one the regular expression:

```
<my_string> =~ <my_regex>
```

Be careful!

Although it is convenient for our examples, the =~ operator is not defined in the POSIX standard, so you should not use it in POSIX-compliant shell scripts.

If there is a match, even a partial one, `grep` will print the matched characters. If there is no match, nothing will be printed on your shell.

If you want to use the extended notation, use the `-E` flag of `grep`.

Tips

You will see more about `grep` in the following chapters. Do not worry about it for now.

3.1 Basic *regex*

You can find here the syntax for basic regexes, associated with their extended counterpart.

Meaning	Symbol for basic regex	Symbol for extended regex
Alternative	`|`	`|`
1 or more occurrences of the previous element	`\+`	`+`
0 or one occurrence of the previous element	`\?`	`?`
At least n and at most m occurrences of the previous element	`\{n,m\}`	`{n,m}`
At least n occurrences of the previous element	`\{n,\}`	`{n,}`
At most m occurrences of the previous element	`\{0,m\}`	`{0,m}`
Exactly n occurrences the previous element	`\{n\}`	`{n}`
Characters grouping	`\(\)`	`()`

3.2 Extended *regex*

Extended regular expressions have the same power as the basic regular expressions, only the syntax is different: it omits the \ character for certain constructions. This greatly simplifies your regexes.

Note that bash regexes use the extended regex syntax. Thus, we can check whether a string matches a pattern using the following command:

```
bash$ [[ "ABCD" =~ ABCD ]] && echo "match" || echo "mismatch"
match
```

```
ABCD =~ ABCD            => match
abcd =~ ABCD            => mismatch
xyzABCDxyz =~ ABCD      => match (with the substring)
```

Some characters have special meaning: they are called **metacharacters**. Metacharacters are building blocks that can easily match specific patterns.

Tips

If you want to match a character that is also a **metacharacter**, you can escape it with a `'\'`.

3.2.1 The dot '.'

The dot **metacharacter** '.' represents *one* instance of *any* character:

```
AxB =~ A.B              => match
A B =~ A.B              => match
A12B =~ A.B             => mismatch
A12B =~ A..B            => match
A =~ A.                 => mismatch
AxB =~ A\.B             => mismatch
A.B =~ A\.B             => match
```

3.2.2 Start '^' and end '$'

The two special symbols '^' and '$' represent, respectively, the beginning and end of a string. They are not mapped to any actual character but rather with an empty string. For example `^a` means "any string beginning with a" and `b$` means "any string ending with b".

```
abcde =~ ^a             => match
bacde =~ ^a             => mismatch
abcde =~ e$             => match
abcde =~ ^a$            => mismatch
a =~ ^a$                => match
```

Be careful!

The '^' and '$' symbols resume their literal meaning if they are not at the beginning, or at the end of the regex respectively, or if they are correctly escaped:

```
A$B =~ A$B              => match
A^ =~ A\^               => match
(empty string) =~ ^$    => match
^$ =~ ^$                => mismatch
b$ =~ b$                => mismatch
b$ =~ b\$               => match
```

3.2.3 Alternative '|'

The character '|' represents an alternative between two blocks. Thus *"ab|cd"* means: match either *"ab"* or *"cd"*.

```
ab =~ ab|cd             => match
cd =~ ab|cd             => match
axd =~ ab|cd            => mismatch
abd =~ ab|cd            => match
```

3.2.4 List `[...]`

Lists are shortcuts for alternatives: instead of writing `a|b|c`, we can write `[abc]`.

```
abd =~ a[bce]d          => match
acd =~ a[bce]d          => match
axd =~ a[bce]d          => mismatch
```

Be careful!

Inside lists, *metacharacters* resume their literal meaning.

```
\ =~ [.\*\]             => match
[ =~ []abc[]            => match
] =~ []abc[]            => match
a =~ []abc[]            => match
```

If the **first character** of the list is '^', the meaning of the list is reversed and becomes "all except one of the character list":

```
a =~ [^abc]             => mismatch
c =~ [^abc]             => mismatch
e =~ [^abc]             => match
^ =~ [abc^]             => match
```

Be careful!

If '^' is not the first character, it simply holds its literal meaning.

3.2.5 Interval

The intervals are shortcuts to describe successive characters in a list.

```
z3 =~ [a-z][0-9]        => match
zz =~ [a-z][0-9]        => mismatch
a4 =~ a[1-35]           => mismatch
a5 =~ a[1-35]           => match
a6 =~ a[1-35-7]         => match
```

Here, [1-35] does **not** mean "any number from 1 to 35" but "any character between 1 and 3, or 5". Make sure to read character by character.

```
- =~ [-A-C]             => match
B =~ [-A-C]             => match
```

Be careful!

The order of the characters is determined by the ASCII code. Type `man ascii` in your shell for more information.

3.2.6 Repetition

To describe repetitions in regex, we have the following *metacharacters*:

- '`*`' zero or more occurrences of the preceding element.
- '`+`' one or more occurrences of the preceding element.
- '`?`' zero or one occurrence of the preceding element.
- `{n,m}` at least `n` and no more than `m` occurrences of the preceding element.
- `{n,}` at least `n` occurrences of the preceding element.
- `{n}` exactly `n` occurrences of the preceding element.

```
ac =~ ab+c              => mismatch
abc =~ ab+c             => match
abbbc =~ ab+c           => match

ac =~ ab*c              => match
abc =~ ab*c             => match
abbbc =~ ab*c           => match

ac =~ ab?c              => match
abc =~ ab?c             => match
abbbc =~ ab?c           => mismatch

abc =~ ab{2,3}c         => mismatch
abbc =~ ab{2,3}c        => match
abbbc =~ ab{2,3}c       => match
abbbbc =~ ab{2,3}c      => mismatch
```

Note that the repetition of characters combined with *metacharacters* representing alternatives does not expect all occurrences to be the same character:

```
a12345c =~ a.{5}c     => match (matches any character 5 times)
axyzxc =~ a[xyz]+c    => match
```

3.2.7 Grouping

We can group different characters in one element with "`()`". This form is very useful, especially in combination with other **metacharacters**:

```
123 =~ (123){2}          => mismatch
123123 =~ (123){2}       => match

abc =~ (abc)|(def)       => match
def =~ (abc)|(def)       => match
ab =~ (abc)|(def)        => mismatch
abef =~ (abc)|(def)      => mismatch
xyz =~ (abc)|(def)       => mismatch

AxBAyB =~ (A.B){2}       => match
```

3.2.8 Back reference

Be careful!

Bash does not natively support back references, so you must test the next examples using the *basic regex syntax*. You can use grep to test like so:

```
42sh$ echo "<my_string>" | grep "<my_regex>"
```

Where `<my_string>` is the string you want to match, and `<my_regex>` is the regex used to match the string.

The value matched by a group can be reused in the following regular expression through construct '`\N`', with N a number representing the N-th group:

```
ax        =~ \(.\)\1              => mismatch
axa       =~ \(.\)x\1             => match
axbcyabc  =~ \(.\)x\(..\)y\1\2    => match
axbcyabd  =~ \(.\)x\(..\)y\1\2    => mismatch
```

In the previous example, '`\1`' was *'a'*, and '`\2`' was *'bc'*.

Note that the groups can be nested in this case, the N-th group is defined by the N-th opening parenthesis:

```
aceacec   =~ \([ab]\([cd]\)[ef]\)\1\2    => match
acecace   =~ \([ab]\([cd]\)[ef]\)\1\2    => mismatch
```

3.2.9 Shorthand character classes

Some character classes are defined to help you build regexes. Beware there are many variations between the different flavors of regex.

For now, try using what you have learned today, even though the character classes will be very useful later on.

Tips

If you need a tool to help you visualize the behavior of the regex you have written, we strongly recommend you to try regex101.

Be careful!

Regex are not to be confused with globbing. They are often used to match text patterns whereas globbing is more often used to match a file's name and/or content in a terminal for example.

Here is a table showing the differences between the similar syntaxes:

Pattern	Regex	Globbing
*	matches 0 or more of the character it follows	matches 0 or more characters
?	matches 0 or 1 of the character it follows	matches exactly 1 of any character
.	matches any single character	not part of the syntax, it is just a literal point

4 `grep(1)`

4.1 Introduction

`grep(1)` is a command-line tool that searches in a given file (or from standard input if no file is given), for all lines matching a specific regular expression.

By default, `grep(1)` prints lines that contains a pattern matched by a given *regex*.

Go ahead and type `man 1 grep` in your shell to discover the usage of this command.

Tips

Here are some options of `grep(1)`:

- `-E` use extended regular expressions.
- `-v` print lines excluding the chain.
- `-n` number each line containing the string.
- `-A NUM` print `NUM` lines of trailing context after matching lines.

4.2 Examples

First, let us create a test file:

```
42sh$ echo -ne "foo\nbar\nbaz\n" > test.txt
```

Then, try some options:

```
42sh$ grep -n "^ba.$" test.txt
2:bar
3:baz
42sh$ grep -nv "^ba.$" test.txt
1:foo
42sh$ grep -EA 2 "fo{2}" test.txt
foo
bar
baz
```

4.3 Challenges

Here are some questions to practice with your `~/.bash_history` file:

1. How to get all lines beginning with the *'v'* character or by the *"ls"* chain?
2. How to display the lines not containing the pattern *"sh"*?
3. `grep(1)` can also take lines on its standard input, how would you print all files beginning by `.bash` in your home directory?

5 `sed(1)`

`sed(1)` is a non-interactive *stream editor* which main goals are:

- Read lines from a file (or standard input) one by one.
- Apply editing commands.
- Print the result on the standard output.

Tips

As always, `man 1 sed` is your friend!

5.1 Syntax

The `sed(1)` invocation syntax is:

```
sed [options] 'command' file
```

Note that options are obviously optional.

Tips

If you want to use sed-commands stored in a file, use the `-f` option:

```
sed -f sed-script file
```

5.2 Simple commands

5.2.1 The substitute command: `s`

The substitution function changes the first occurrences of a string with another. This is by far the most widely used command in `sed` scripts. Alone, it makes `sed` interesting.

The general syntax is:

```
s/pattern/replacement/flags
```

- *pattern* is a regex.
- *replacement* is a string that replaces the *pattern* in the result. This string can contain two wildcards:
  - '`\N`': `N`, being a number between 1 and 9, is replaced by the `N`-th group in the pattern. You can define a group by putting a part of the regex between `\(\)`.
  - '`&`': is replaced by the complete string that has been matched by the *pattern*.

Some available flags:

- '`g`': replace all encountered occurrences in the current line.
- '`n`': suppress automatic printing of pattern space.
- '`N`'': replace only the `N`-th occurrence of the pattern in the line.
- '`p`': display the line on the standard output if a substitution is made.
- '`w file`': writing the row to *file* if a substitution is made.

There may be several flags simultaneously.

Here are some examples:

```
# Changes the first occurrence of the string 'hello' by 'bye'.
sed 's/hello/bye/' file

# Changes the third occurrence of 'hello'.
```

(continues on next page)

(continued from previous page)

```
sed 's/hello/bye/3' file

# Changes all occurrences of the string 'hello' to 'bye'.
sed 's/hello/bye/g' file
```

The substitution function can obviously be used with a regular expression:

```
# Replaces all 'Strawberry' or 'strawberry' strings by 'STRAWBERRY'.
sed 's/[Ss]trawberry/STRAWBERRY/g' file

# Replaces all 'Strawberry' or 'strawberry' strings
# with '_Strawberry_' or '_strawberry_'.
sed 's/[Ss]trawberry/_&_/g' file

# Removes '<>' surrounding the strings.
# Here, '\1' match the first (and unique) group, made by '(.*)'.
sed 's/<\(.*\)>/\1/g' file
```

Going further...

- Although this is not POSIX-compliant, when using GNU `sed` you can directly modify the given file instead of printing the result on the standard output. To do so use the `-i` (*in-place*: it does not require an intermediary file) option.

  ```
  42sh$ sed -i 's/titi/toto/g' file
  ```

- You can use any separator character (not only `/`). To resume its literal meaning in patterns, it must be escaped by the `\\` character.

  ```
  42sh$ sed 's,titi,toto,g' file
  ```

5.2.2 The transliterate command: `y`

- `y`: `y/src-chars/dest-chars/` transposes all characters from `src-chars` to their equivalent in `dest-chars`, as the UNIX command `tr(1)` does.

```
42sh$ echo "abcd" | sed "y/ac/13/"
1b3d
```

5.2.3 The delete command: `d`

- `d`: deletes the current line and starts the next cycle immediately without executing other commands.

```
42sh$ printf '%s\n%s\n%s\n' tic tac toe | sed '/ta./d'
tic
toe
```

5.2.4 The print command: `p`

- `p`: *print*s the selected line on standard output. It is essential when `sed` is invoked with the `-n` option (see below). This is **not** the `p` flag available with the `s` command.

```
42sh$ printf '%s\n%s\n%s\n' tic tac toe | sed '/ta./p'
tic
tac
tac
toe
42sh$ printf '%s\n%s\n%s\n' tic tac toe | sed -n '/ta./p' # with -n option
tac
```

5.3 Addresses

`sed`'s commands actually follow this syntax:

```
[addr]X[options]
```

Where `X` is a single-letter `sed`-command. `[addr]` is an optional line address. If `[addr]` is specified, the command X will be executed only on the matched lines. `[addr]` can be a single line number, a regular expression, or a range of lines.

Here are some address formats:

- The number '`n`' specifies that only the `n`-th row of data is concerned by the command. The first row is 1.
- '`$`' is the last line of the last data file.
- '`/expression/`' selects all rows containing the regex `expression` (as seen above).
- '`n,m`' selects all lines from the line *n* until the *m* line.
- '`addr1,addr2`' selects all lines between the pattern *addr1* and *addr2* pattern. *addr1* may be replaced by a line number. However, the search for *addr2* only begins after *addr1* line if *addr2* is a regex.
- '`addr!`' removes all lines that have been selected by *addr*, and selects the others.

Here are some examples:

```
42sh$ cat test.txt
l1 tac
l2 toe
l3 tic
l4 tac
l5 toe
l6 tic
l7 tac
l8 toe
l9 tic
l10 tac
42sh$ sed -n /tac/p test.txt
```

(continues on next page)

(continued from previous page)

```
l1 tac
l4 tac
l7 tac
l10 tac
42sh$ sed -n 4,7p test.txt
l4 tac
l5 toe
l6 tic
l7 tac
42sh$ sed 4,7s/tac/foo/ test.txt
l1 tac
l2 toe
l3 tic
l4 foo
l5 toe
l6 tic
l7 foo
l8 toe
l9 tic
l10 tac
42sh$ sed -n /l4/,/l6/p test.txt # Prints between line containing 'l4' to line containing 'l6
↪'.
l4 tac
l5 toe
l6 tic
42sh$ sed -n '$!p' test.txt
l1 tac
l2 toe
l3 tic
l4 tac
l5 toe
l6 tic
l7 tac
l8 toe
l9 tic
```

Be careful!

You must quote '$!p' to avoid variable expansion of '$!'.

5.4 Greediness

Some metacharacters are called *greedy* or *possessive*, this refers to the fact that these characters will consume your stream until the last pattern is found. Here is an example:

You have a list of items separated by commas but you do not know how many items you have, and you only want the second one.

A naive code would be:

```
42sh$ echo "1,2,3,4,5" | sed -n "s/^.*,\(.*\),.*/\1/p"
4
```

Because of greediness, your first expression eats all the first commas. You can check that the first `.*` matched everything until the third comma with the following command:

```
42sh$ echo "1,2,3,4,5" | sed -n "s/\(^.*\),\(.*\),.*/\1/p"
1,2,3
```

In order to avoid that behavior use `[^,]*`, instead of `.*,`. It means *everything that is not a comma and then a comma*.

```
42sh$ echo "1,2,3,4,5" | sed -n "s/^[^,]*,\([^,]*\),.*/\1/p"
2
```

5.5 Advanced commands

`sed` is a language of its own and provides many other commands, which will not be discussed here, but you can find many resources and advanced examples on the GNU documentation or the sourceforge home of `sed`.

Note also that `vim` features a command similar to `sed`.

I must not fear. Fear is the mind-killer.