Searching trees

Find tree structures and use them to access data more elegantly

Enable native scrolling

But first:

possible solutions for the last assignment

Task 1

An array of strings

```
char *month_names[]= {
  "January",
  "February",
  "March",
  ...
}
```

...

Task 2

An array of structs

```
struct month_info_t {
  char *name;
  uint8_t num_days;
};

struct month_info_t month_info[]= {
  {.name= "January",   .num_days= 31},
  …
}
```

...

Task 3

Birthday countdown

```
struct date_t pivot;
struct date_t birthday;

for(;;) {
  uint16_t days_next_bd=
    date_days_between(&pivot, &birthday);
  date_print(&pivot);
  date_increment(&pivot);
}
```

...

Task 4

Pointerized countdown

```
struct date_t {
  uint8_t day_of_month;
  struct month_info_t *month;
};
```

...

Back to the trees

Guess game

```
if(num_cur_guess == num_to_guess)
  Serial.print("Your guess is correct!");

if(num_cur_guess < num_to_guess)
  Serial.print("too small");

if(num_cur_guess > num_to_guess)
  Serial.print("too big");
```

...

Upload the example above and follow the instructions in the serial monitor

Hint: Add a . or another symbol after the numbers you enter

Hint: The number is between 0 and 31

Strategy #1

```
Enter an estimate for the random number:
Your guess (1) is too small try again
Enter an estimate for the random number:
Your guess (2) is too small try again
Enter an estimate for the random number:
Your guess (3) is too small try again
Enter an estimate for the random number:
Your guess (4) is too small try again
…
```


Strategy #1

Can take up to `n` tries for a range of `n` numbers.

Can we do better?

Strategy #2

```
Your guess (16) is too big try again
Enter an estimate for the random number:
Your guess (8) is too big try again
Enter an estimate for the random number:
Your guess (4) is too small try again
Enter an estimate for the random number:
Your guess (6) is too small try again
Enter an estimate for the random number:
Your guess is correct! The number was:7
You needed 5 steps to find the number
```

Split the interval into two subintervals per step

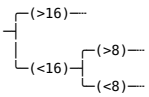
Strategy #2

First Guess: 16

Strategy #2

```
Your guess (16) is too big, try again
```

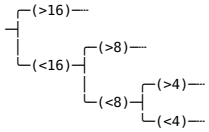
Second Guess: 8



Strategy #2

```
Your guess (8) is too big, try again
```

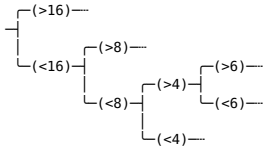
Third Guess: 4



Strategy #2

```
Your guess (4) is too small, try again
```

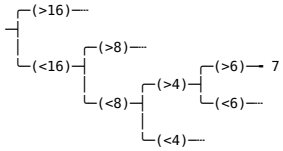
Fourth Guess: 6



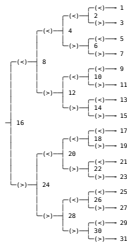
Strategy #2

```
Your guess (6) is too small, try again
```

Fifth Guess: 7



Complete decision tree



... .py

Strategy #2

Observation: By splitting the interval in half for each iteration we are actually following a binary tree structure

For `n` elements we need at most `log_2(n)` steps to find the correct element

Using that observation

Student register

As part of a student database we need a way to search for names by matriculation number

Student register (flat)

```
1 struct student_t {
2   char *name;
3   uint32_t mat_num;
4 };
5
6 struct student_t student_register[MAX_STUDENTS];
```

One way to structure the data is to use a simple array

This is a flat structure

Student lookup (flat)

```
1 struct student_t *
2 sr_flat_lookup(struct student_t *flat,
3                uint32_t mat_num)
```

Download the database sourcecode and find the `sr_flat_lookup` function

Use a `for` loop to iterate through all elements of `flat[]` while checking if the matriculation number matches `mat_num`

Return the current array element if the matriculation number matches

Test your implementation in the serial monitor

Hint: Add a `.` or another symbol after the numbers you enter into the monitor

Student lookup (flat)

```
1 for(size_t i=0; flat[i].name; i++) {
2   if(flat[i].mat_num == mat_num)
3     return(&flat[i]);
4 }
```

In the worst case (student not in database) we would have to look at every single entry

Can we do better?

Student register (tree)

```
1 struct student_t {
2   char *name;
3   uint32_t mat_num;
4   struct student_t *child_lt;
5   struct student_t *child_gt;
6 };
```

...

To make lookups more efficient we can hook the database entries into a tree structure

Student register (tree)

```
1 void sr_print_subtree(struct student_t *root,
2                       uint16_t indent)
3 {
4   sr_print_subtree(root->child_lt, indent+1);
5   Serial.print(root->name);
6   sr_print_subtree(root->child_gt, indent+1);
7 }
```

Download the tree based database sourcecode

Change the `DEBUG_TBUILD` and `DEBUG_LOOKUP` `#define` at the top of the file from `false` to `true`

Watch the output in the serial monitor and try to find out how new elements are inserted into the tree

Finding elements

Now enter some numbers into the serial monitor and try to find out how elements are looked up in the tree

Hint: Add a `.` or another symbol after the numbers you enter

Comparing speed

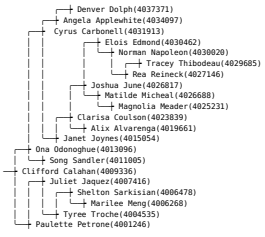
Change the `DEBUG_TBUILD` and `DEBUG_LOOKUP` `#define` at the top of the file back from `true` to `false`

Copy your implementation of `sr_flat_lookup` into the file

Compare the displayed runtimes for the two lookup algorithms

How does the position of an entry in the array influence the lookup times?

Tree balance



In the tree above you can see that there are far more elements behind the `child_lt`-branch of "Clifford Calahan" than behind the `child_gt`-branch

This indicates that the tree is not well balanced

Tree balance influences the lookup speed

Tree balance

```
1 {.name= "Paulette Petrone",  .mat_num= 4001246, …
2 {.name= "Juliet Jaquez",     .mat_num= 4007416, …
3 {.name= "Clifford Calahan",  .mat_num= 4009336, …
4 …
```

Edit the `student_register` array so that the matriculation numbers of the first eight entries are in ascending order

Do not change the other elements

`#define DEBUG_TBUILD` to be `true`

Does sorting the elements before inserting them improve the balance of the tree?