1. Animals. This literate CWEB program is an implementation of Animal Game. The aim of this game is to guess the animal the user is thinking of. The guessing will be done by a series of questions asked to the user in a form of a binary tree. If the animals is not guessed, the user will introduce the new animal to the program, giving it a certain differentiator question from others. With the user's input, the database of animals will expand and the program's reach will be broader.

The overall skeleton of the program is as follows:

```
⟨ Includes 2 ⟩
⟨ Functions 3 ⟩
⟨ The main function 4 ⟩
```

2. First we will include all the other necessary libraries the program will require, which are:

`stdio.h` - to perform input and output operations

`string.h` - functions for handling strings

`stdlib.h` - for dynamic memory management

`unistd.h` - for access to POSIX OS API

```
⟨ Includes 2 ⟩ ≡
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
```

This code is used in section 1.

3. The first thing we'll do is declare a node structure called *node* , which will be the main piece of the program. A *node* will consist in 2 strings of **char** ∗*text*, that will contain a question the user will have to answer and **char** ∗*animal*, with the name of the animal in the case it is a node with no children. The structure will also have 2 integers *no* and *yes* which will point to another node.

```
⟨ Functions 3 ⟩ ≡
  typedef struct {
    char *text;
    char *animal;
    int no;
    int yes;
  } node;
  ⟨ Print nodes 7 ⟩
  ⟨ Free nodes 8 ⟩
  ⟨ Validation functions 11 ⟩
  ⟨ Question to distinguish animals 19 ⟩
```

This code is used in section 1.

4. The structure of the main.

Around section 32 when we ask the user if he wants to play again I realized we didn't had the program in a loop that kept on going if the user was keen on playing again. I decided it was far easier to add the loop here in the structure of the main rather than further down.

```
⟨ The main function 4 ⟩ ≡
  main( )
  {
    ⟨ Global variables 5 ⟩
    printf("Welcome␣to␣the␣Animal␣Game!!\n\n");
    ⟨ Initialise nodes 6 ⟩
    while (playAgain) {
      ⟨ Navigate tree 12 ⟩
      ⟨ Final question 13 ⟩
      ⟨ Play again 33 ⟩
    }
    ⟨ Save animals 39 ⟩
    ⟨ Print this 42 ⟩
    ⟨ Clean up 43 ⟩
    return 0;
  }
```

This code is cited in section 39.

This code is used in section 1.

5. Some global variables should be declared, like the number of nodes *numNodes* found, *root* which shows the id of the root node, *actual* and *past* nodes which both start pointing at the root in the beginning of each program and *hasChildren* which will be used as a boolean to know if the actual node has children or not.

```
⟨ Global variables 5 ⟩ ≡
  int numNodes;
  int root;
  int actual;
  int past;
  int hasChildren = 1;
```

See also sections 10, 14, 16, 25, 34, 37, and 40.

This code is used in section 4.

6. First we will declare and initialize an array called *nodes* that will contain structures of **node**. The array will be the size of a defined *max*, the maximum number of nodes the program can hold.

```
#define max 1999
```

⟨ Initialise nodes 6 ⟩ ≡

```
  node *nodes = malloc(max * sizeof (*nodes));
  ⟨ Initialise default animals 9 ⟩
```

This code is used in section 4.

7. I'll use a function to print the nodes, just for debugging reasons. Maybe printing out data is not the best way to figure out what is going on in your program, but I'm a very visual person and it helps me a lot. So here it goes:

⟨ Print nodes 7 ⟩ ≡

```
  void printNodes(node array[], int size)
  {
    printf("\nNODES:\n");
    int i;
    for (i = 0; i < size; i++) {
      printf("id:␣%d", i);
      printf("␣text:␣%s", array[i].text);
      printf("␣anim:␣%s", array[i].animal);
      printf("␣no:␣%d", array[i].no);
      printf("␣yes:␣%d\n", array[i].yes);
    }
  }
```

This code is used in section 3.

8. In C the memory management is very important, so here is a function to free the allocated memory used to store the **node** array.

⟨ Free nodes 8 ⟩ ≡

```
  void freeNodes(node array[], int size)
  {
    int i;
    for (i = 0; i < size; i++) {
      free(array[i].text);
      free(array[i].animal);
    }
    free(array);
  }
```

This code is used in section 3.

9. In this section the stored animals will be initialised in the array *nodes* of structures type **node**. This will be done by asking the user if they want to open a certain file that contains animals, if not a default file 'stored.dat' will be used (if it does not exist, it will be created).

```
⟨ Initialise default animals 9 ⟩ ≡
  ⟨ Ask for file 35 ⟩
  ⟨ Load file 38 ⟩
```

This code is cited in section 35.

This code is used in section 6.

10. What we need to do next is to navigate the stored nodes according to the users answers. If user says 'yes' go one way, if user answers 'no' go the other way. To do that, first we will need to add 2 more global variables, *ans*[10] and **int** *check* which will hold the answer (yes or no) the user provides as input and an int that will determine if that answer is valid. If the answer is invalid, the value of *check* will be -1 and the user will be prompted to enter an correct answer this time.

```
⟨ Global variables 5 ⟩ +≡
  char answ[10];
  int check;
```

11. To check if the input answer of the user is valid, we'll make function called **int** *checkAnswer*(**char** *array*[ ]) which will receive a string and will return specific numbers in the case of certain answers. The only valid answers are 'Yes' or 'No' in their different formats (all lowercase, all upercase, first letter uppercase, just 'n' or 'y', etc). The function will return: 0 when the answer is 'No', 1 when it is 'Yes' and -1 when the input is not one of the 2 previous options.

```
⟨ Validation functions 11 ⟩ ≡
  int checkAnswer(char array[ ])
  {
    if (strcmp(array, "yes") ≡ 0 ∨ strcmp(array, "Yes") ≡ 0 ∨ strcmp(array, "y") ≡ 0 ∨ strcmp(array,
          "Y") ≡ 0 ∨ strcmp(array, "YES") ≡ 0) {
      return 1;
    }
    if (strcmp(array, "no") ≡ 0 ∨ strcmp(array, "No") ≡ 0 ∨ strcmp(array, "n") ≡ 0 ∨ strcmp(array,
          "N") ≡ 0 ∨ strcmp(array, "NO") ≡ 0) {
      return 0;
    }
    return −1;
  }
```

See also sections 18, 20, 21, 22, and 26.

This code is used in section 3.

12. This part of code will be in charge of navigating through the *nodes* struc array *nodes* until the *actual* node has no children to go to.

```
⟨ Navigate tree 12 ⟩ ≡
  printf("THINK␣OF␣AN␣ANIMAL␣AND␣PRESS␣ENTER:␣\n");
  while (getchar() ≠ '\n') ;
  while (hasChildren) {
    if (nodes[actual].no ≠ −1) {
      printf("%s␣", nodes[actual].text);
      scanf("%[^\n]%*c", answ);
      check = checkAnswer(answ);
      if (check ≡ −1) {
        printf("Invalid␣input,␣try␣again.␣(Yes␣or␣No)\n");
      }
      else {
        past = actual;
        if (check ≡ 0) {
          actual = nodes[actual].no;
        }
        else {
          actual = nodes[actual].yes;
        }
      }
    }
    else {
      hasChildren = 0;
    }
  }
```

This code is used in section 4.

13. When the *actual* node is one without children, it means the user will face a final concrete question of the animal the program thinks the user is thinking about. We need to ask the content inside the *actual* node and make sure the user provides a valid answer. Just like in the code before, we will use the function *checkAnswer*( ) to validate the input.

```
⟨ Final question 13 ⟩ ≡
  while (lastQ) {
    printf("%s␣", nodes[actual].text);
    scanf("%[^\n]%*c", answ);
    check = checkAnswer(answ);
    if (check ≡ −1) {
      printf("Invalid␣input,␣try␣again.␣(Yes␣or␣No)\n");
    }
    else {
      lastQ = 0;
    }
  }
```

See also section 15.

This code is used in section 4.

14. The previous request for the user's input is surrounded by a while, and the program will get out of it until the user provides a correct input. For that to work we'll have to add a new variable *lastQ* which will work as the boolean indicator for the while cycle.

```
⟨ Global variables 5 ⟩ +≡
  int lastQ = 1;
```

15. Supposing the user entered a correct answer, we will now have to check if the program guessed the animal the user was thinking or not. If the animal was guessed the value of *check* will be 1, meaning the user entered 'Yes' as an answer to the final question. *Ex. Is your animal a dog?*

```
⟨ Final question 13 ⟩ +≡
  if (check ≡ 1) {
    printf("\nI␣win!!␣I␣guessed␣your␣animal␣:)\n");
    printf("\n");
  }
  else {
    ⟨ Animal not guessed 17 ⟩
  }
```

16. If the program does not guess the animal the user was thinking off, it should add it to its internal DB. We'll create a char array **char** *inputAnimal*[20], to store the name of the animal the user was thinking off.

```
⟨ Global variables 5 ⟩ +≡
  char inputAnimal2[20];
  char *inputAnimal;
```

17. Since the animal was not guessed, the first step to store this new creature into the local information of animals will be to ask the user for the name of the animal and make the input all lowercase letter in case the user used upper case letters. To make the string lowercase we will use another function called *strlwr*(**char** **string*) we will define further down.

```
⟨ Animal not guessed 17 ⟩ ≡
  printf("What␣was␣your␣animal?␣");
  scanf("%[^\n]%*c", inputAnimal2);
  inputAnimal = strdup(inputAnimal2);
  inputAnimal = strlwr(inputAnimal);
```

See also sections 23, 24, and 28.

This code is used in section 15.

18. This function will receive a string that will make every char of the string a lowercase letter and return a pointer to where the string is located.

```
⟨ Validation functions 11 ⟩ +≡
  char *strlwr(char *str)
  {
    size_t i;
    size_t len = strlen(str);
    for (i = 0; i < len; i++) str[i] = tolower((unsigned char) str[i]);
    return str;
  }
```

19. The next step for storing the new animal is to formulate and ask a differentiator question between the animal of the actual node and the animal the user is thinking about. To generate the differentiator question, we'll be doing an ambicious function **void** *askDistinguishQ*(**char** ∗*stored*, **char** ∗**new**) which will receive the animal of the *actual* node and the new animal entered by the user.

To obtain a question with proper grammatical logic, we will also need to implement 3 functions, *addArticle*( ), *checkSpace*( ) and *substr*( ), which will be implemented further down.

We will also take advantage of the built in function from the `string.h` library *strcat*(**char** ∗*dest*, **char** ∗*source*) to glue together the final differentiator question.

⟨ Question to distinguish animals 19 ⟩ ≡

```
void askDistinguishQ(char *stored, char *new)
{
  char question[100] = "What␣question␣would␣distinguish␣";
  int art1 = addArticle(stored);
  if (art1 ≡ 1) {
    strcat(question, "a␣");
  }
  else {
    strcat(question, "an␣");
  }
  strcat(question, stored);
  strcat(question, "␣from␣");
  int space = checkSpace(new);
  char *newAni;
  if (space ≠ 0) {
    newAni = substr(new, space);
  }
  else {
    newAni = new;
  }
  int art2 = addArticle(newAni);
  if (art2 ≡ 1) {
    strcat(question, "a␣");
  }
  else {
    strcat(question, "an␣");
  }
  strcat(question, newAni);
  strcat(question, "?␣");
  printf("%s\n", question);
}
```

See also section 27.

This code is used in section 3.

20. First we will implement the *addArticle*(**char** *array*[ ]) function, which will receive a string with the name of animal, check if the first char of the string is a vowel and return 2 if is (meaning it will need an 'an') or return 1 (for 'a'). When calling the function, it will indicate the program it if needs to either concatenate an 'an' or 'a' prior to the animal's name.

⟨ Validation functions 11 ⟩ +≡

```
int addArticle(char array[])
{
  if (array[0] ≡ 'a' ∨ array[0] ≡ 'e' ∨ array[0] ≡ 'i' ∨ array[0] ≡ 'o' ∨ array[0] ≡ 'u') {
    return 2;
  }
  return 1;
}
```

21. When the programs prompts the user to enter the animal he was thinking of, we don't know how he is going to enter it. The user might use an article, might not or it may use an incorrect one. Because of that, we will implement the function *checkSpace*(**char**[ ]*array*) that will return the position of where a space is located (if it finds one, if not it will return 0) in the string entered by the user, so in that way we would be able to know which part of the string we can trim off to just keep the animal's name.

⟨ Validation functions 11 ⟩ +≡

```
int checkSpace(char array[])
{
  int i;
  int len = (int) strlen(array);
  for (i = 0; i < len; i++) {
    if (array[i] ≡ '␣') {
      return i + 1;
    }
  }
  return 0;
}
```

22. The last validation function needed for the *askDistinguishQ*( ) function is a function to obtain substring. **char** ∗*substr*(**char** ∗*string*, **int** *start*) will receive a string and a point to start, then it will use the built in funcion *strncpy*( ) to copy the string from the starting point to the end. A pointer to the new substring is returned.

⟨ Validation functions 11 ⟩ +≡

```
char *substr(char *cadena, int comienzo)
{
  int longitud = (int) strlen(cadena);
  if (longitud ≡ 0) longitud = strlen(cadena) − comienzo − 1;
  char *nuevo = (char *) malloc(sizeof(char) * longitud);
  strncpy(nuevo, cadena + comienzo, longitud);
  return nuevo;
}
```

23. Now with the validation functions in place we have the *askDistinguishQ*( ) function complete and we can now call it to continue the storing process of the new animal.

⟨ Animal not guessed 17 ⟩ +≡

```
askDistinguishQ(nodes[actual].animal, inputAnimal);
```

24. The newt step after asking the user for a question to distinguish 2 animals will be to obtain the input the user enters, since it will become the text of one of the new nodes to be created. Sometimes users don't write everything correctly, so we must make sure the first letter is uppercase and that it contains a '?' at the end.

For that will take the input question the user types into a built in function *strlwr*( ) that will make all the chars in the string lowercase. Then we will create a function called *puntctuation*( ) that will take care of the first char being an uppercase letter and to check for the inclusion of the question mark.

⟨ Animal not guessed 17 ⟩ +≡

```
scanf("%[^\n]%*c", userQ);
diffQuest = strdup(userQ);
diffQuest = strlwr(diffQuest);
punctuation(&diffQuest);
```

25. For the previous code to work we will need to add 2 global variables. **char** *userQ*[100] and **char** ∗*diffQuest* which will store the inputs the user enters.

⟨ Global variables 5 ⟩ +≡

```
char userQ[100];
char *diffQuest;
```

26. The function to check that the input the user entered is quite simple, **void** *punctuation*(**char** ∗∗*array*), will receive a string and will make the first character an uppercase letter. It will then check to see if the string contains a '?', if not it will concatanate one at the end.

```
⟨ Validation functions 11 ⟩ +≡
  void punctuation(char **array)
  {
    *array[0] = toupper(*array[0]);
    char *s;
    s = strchr(*array, '?');
    if (s ≡ Λ) {
      strcat(*array, "?");
    }
  }
```

27. Now we will ask the user what would be the answer to the distinguishing question they typed for the animal they where thinking of. For that we will make a function called **void** *questForNewAnimal*(**char** ∗*newAni*) which will formulate a correct question.

```
⟨ Question to distinguish animals 19 ⟩ +≡
  void questForNewAnimal(char *newAni)
  {
    char quest[100] = "What␣would␣be␣the␣answer␣for␣";
    int art1 = addArticle(newAni);
    if (art1 ≡ 1) {
      strcat(quest, "a␣");
    }
    else {
      strcat(quest, "an␣");
    }
    strcat(quest, newAni);
    strcat(quest, "?");
    printf("%s␣", quest);
  }
```

28. We will add this now to the section of code 'Animal not guessed'. We will have to surround the part of code where the user enters an answer with a while in order to make sure they enter a correct answer. Then we will procede to create some new nodes.

```
⟨ Animal not guessed 17 ⟩ +≡
  int space = checkSpace(inputAnimal);
  char *newAni;
  if (space ≠ 0) {
    newAni = substr(inputAnimal, space);
  }
  else {
    newAni = inputAnimal;
  }
  int invalid = 1;
  while (invalid) {
    questForNewAnimal(newAni);
    scanf("%[^\n]%*c", answ);
    check = checkAnswer(answ);
    if (check ≡ −1) {
      printf("Invalid␣input,␣try␣again.␣(Yes␣or␣No)\n");
    }
    else {
      invalid = 0;
    }
  }
  invalid = 1;
  ⟨ Create new nodes 29 ⟩
```

29. We will now procede on creating the new nodes. We will create 2: one that will store the distinguising question the user previously entered and will point to 2 other nodes and the second node will hold a final question according to the animal the user was thinking and was not guessed. *Ex. Is your animal a dog?*

```
⟨ Create new nodes 29 ⟩ ≡
  nodes[numNodes].text = strdup(diffQuest);
  nodes[numNodes].animal = strdup("-");
  if (check ≡ 0) {
    nodes[numNodes].no = numNodes + 1;
    nodes[numNodes].yes = actual;
  }
  else {
    nodes[numNodes].yes = numNodes + 1;
    nodes[numNodes].no = actual;
  }
  numNodes++;
  ⟨ Create final node 30 ⟩
  ⟨ Redirect past node to new 31 ⟩
  ⟨ Check if root needs to change 32 ⟩
```

This code is used in section 28.

30. To create the second node containing the final question, we will need to formulate a grammaticaly correct final quesiton to store in the *nodes*[ ].*text* field. The *yes* and *no* attributes will point to -1 since this node will have no children. The **node**[ ].*animal* field will contain the name of the new animal added to the tree.

⟨ Create final node 30 ⟩ ≡

```
char finalQuest[80] = "Is␣your␣animal␣";
int art = addArticle(newAni);
if (art ≡ 1) {
   strcat(finalQuest, "a␣");
}
else {
   strcat(finalQuest, "an␣");
}
strcat(finalQuest, newAni);
strcat(finalQuest, "?␣");
nodes[numNodes].text = strdup(finalQuest);
nodes[numNodes].animal = strdup(newAni);
nodes[numNodes].yes = −1;
nodes[numNodes].no = −1;
numNodes++;
```

This code is used in section 29.

31. Now that the 2 new nodes are created, we should check the past node to redirect the answer (Yes or No) direction, to point to one of the new nodes. To do this we will go through the *nodes*[ ].*yes* and *nodes*[ ].*no* attributes to see which one of them points to the *actual* **node** and change it so that attribute now points to the first of the nodes created.

⟨ Redirect past node to new 31 ⟩ ≡

```
if ((nodes[past].yes ≡ actual) ∧ nodes[past].yes ≠ −1) {
   nodes[past].yes = numNodes − 2;
}
if ((nodes[past].no ≡ actual) ∧ nodes[past].no ≠ −1) {
   nodes[past].no = numNodes − 2;
}
```

This code is used in section 29.

32. At the end we need to check if one of the nodes we are creating is now going to be the root of the tree. This will only happen in the first run of the game, so we need to make the necessary changes to update this.

```
⟨ Check if root needs to change 32 ⟩ ≡
  if (actual ≡ root) {
    root = numNodes − 2;
  }
```

This code is used in section 29.

33. When the game is over, whether the program guessed the animal or not, it should ask the user if he wants to play again. It should validate that the user inputs a valid answer.

```
⟨ Play again 33 ⟩ ≡
  int answPlay = 1;

  while (answPlay) {
    printf("\nDo␣you␣want␣to␣play␣again?␣");
    scanf("%[^\n]%*c", answ);
    check = checkAnswer(answ);
    if (check ≡ −1) {
      printf("Invalid␣input,␣try␣again.␣(Yes␣or␣No)\n");
    }
    else {
      answPlay = 0;
    }
  }
  answPlay = 1;
  if (check ≡ 0) {
    playAgain = 0;
  }
  else {
    printf("\n");
    actual = root;
    past = root;
    hasChildren = 1;
    lastQ = 1;
  }
```

This code is used in section 4.

34. Now I realise we need to encapsulate some part of the main function in a loop in case the user wants to play again. I think it will be more comprehensible to add a while further up in the specification of the structure of the main. So now we will just add a global variable named *playAgain* that will work as a boolean indicator of the user's answer.

```
⟨ Global variables 5 ⟩ +≡
  int playAgain = 1;
```

35. Up until now the program runs but with a volatile memory, once the program is closed, it forgets about all the previous games and the stored animals. So now we will add the file reading/writing in order to save games and load previous ones. We will start by encouraging the user to type the name of a file he wants to open. We will then proceed to check if the file exists. If not, or if the user does not want to use an external file, the game will have to load with a default file called 'stored.dat'. So up in the ⟨ Initialise default animals 9 ⟩ section we will add this new section.

```
⟨ Ask for file 35 ⟩ ≡
  printf("If␣you␣want␣to␣load␣a␣certain␣file␣type␣it␣(type␣NO␣for␣default␣file):\n");
  scanf("%[^\n]%*c", fname);
  if (strcmp(fname, "NO") ≡ 0 ∨ strcmp(fname, "No") ≡ 0 ∨ strcmp(fname, "no") ≡ 0 ∨ strcmp(fname,
        "N") ≡ 0 ∨ strcmp(fname, "n") ≡ 0) {
    strcpy(fname, "stored.dat");
  }
  else {
    if (access(fname, F_OK) ≠ −1) {
      openF = 1;
    }
    else {
      printf("File␣does␣not␣exist,␣using␣default␣file␣stored.dat...\n");
      strcpy(fname, "stored.dat");
    }
  }
  ⟨ Check default file exists 36 ⟩
```

This code is used in section 9.

36. In the case we need to use the default file, first we will need to check if it exists. If not, we will create one with just one animal, our famous and lovely 'horse'.

```
⟨ Check default file exists 36 ⟩ ≡
  if (strcmp(fname, "stored.dat") ≡ 0) {
    if (access(fname, F_OK) ≠ −1) {
      openF = 1;
    }
    else {
      printf("Default␣file␣does␣not␣exists,␣creating␣stored.dat...\n");
      root = 0;
      numNodes = 1;
      actual = root;
      past = root;
      nodes[0].text = strdup("Is␣your␣animal␣a␣horse?␣");
      nodes[0].animal = strdup("horse");
      nodes[0].no = −1;
      nodes[0].yes = −1;
      printf("[1␣animal␣loaded]\n");
    }
  }
```

This code is used in section 35.

37. We will add a global int variale called *openF* that will be used as a boolean to know if we have to read a file to initialize the game or not. Also we will need *fname* will hold the name of the file the user wants to open, *file* will be a pointer type FILE and *line_buffer*[ ] will be a temporary storage for each line of the entered file.

```
⟨ Global variables 5 ⟩ +≡
  char fname[30];
  FILE *file;
  char line_buffer[BUFSIZ];
  int openF = 0;
```

38. No we will proceed to the actual reading of the file. First we will obtain the id of the root and then the number of nodes in the file and will be stored in *root* and *numNodes* respectively. Then according to *numNodes*, there will be a loop to fill out the information of every node.

```
⟨ Load file 38 ⟩ ≡
  if (openF) {
    file = fopen(fname, "r");
    if (¬file) {
      printf("Couldn't␣open␣file␣%s␣for␣reading!\n", fname);
      return 0;
    }
    printf("Opening␣file␣%s...\n", fname);
    fgets(line_buffer, sizeof (line_buffer), file);
    sscanf(line_buffer, "%d", &root);
    fgets(line_buffer, sizeof (line_buffer), file);
    sscanf(line_buffer, "%d", &numNodes);
    actual = root;
    past = root;
    printf("[%d␣animals␣loaded]\n", (numNodes/2) + 1);
    int i, leni;
    for (i = 0; i < numNodes; i++) {
      fgets(line_buffer, sizeof (line_buffer), file);
      leni = strlen(line_buffer);
      if (leni > 0 ∧ line_buffer[leni − 1] ≡ '\n') {
        line_buffer[leni − 1] = 0;
      }
      nodes[i].text = strdup(line_buffer);
      fgets(line_buffer, sizeof (line_buffer), file);
      leni = strlen(line_buffer);
      if (leni > 0 ∧ line_buffer[leni − 1] ≡ '\n') {
        line_buffer[leni − 1] = 0;
      }
      nodes[i].animal = strdup(line_buffer);
      fgets(line_buffer, sizeof (line_buffer), file);
      sscanf(line_buffer, "%d", &nodes[i].no);
      fgets(line_buffer, sizeof (line_buffer), file);
      sscanf(line_buffer, "%d", &nodes[i].yes);
    }
    fclose(file);
  }
```

This code is used in section 9.

39. Now at the end of the program, when the user does not want to keep playing anymore we will prompt the user if he wants to store these animals in a specific file, otherwise the game will be saved into the default file 'stored.dat'. We will add this section of code just before ⟨ Clean up 43 ⟩ in ⟨ The main function 4 ⟩ section.

⟨ Save animals 39 ⟩ ≡

```
  printf("\nEnter␣a␣file␣to␣store␣your␣game␣(type␣NO␣for␣the␣default␣file):\n");
  scanf("%[^\n]%*c", fsave);
```

See also section 41.

This code is used in section 4.

40. We will need to add a global variable that stores the name of the file where the user wants to save his game.

⟨ Global variables 5 ⟩ +≡

```
  char fsave[30];
```

41. Finally we will write the information form our nodes to the indicated file.

⟨ Save animals 39 ⟩ +≡

```
  if (strcmp(fsave, "NO") ≡ 0 ∨ strcmp(fsave, "No") ≡ 0 ∨ strcmp(fsave, "no") ≡ 0 ∨ strcmp(fsave,
        "n") ≡ 0 ∨ strcmp(fsave, "N") ≡ 0) {
    strcpy(fsave, "stored.dat");
  }
  printf("Storing␣animals␣in␣file␣%s...\n", fsave);
  file = fopen(fsave, "w");
  if (file ≡ Λ) {
    printf("Error␣opening␣file!\n");
    exit(1);
  }
  fprintf(file, "%d\n", root);
  fprintf(file, "%d\n", numNodes);
  int i;
  for (i = 0; i < numNodes; i++) {
    fprintf(file, "%s\n", nodes[i].text);
    fprintf(file, "%s\n", nodes[i].animal);
    fprintf(file, "%d\n", nodes[i].no);
    fprintf(file, "%d\n", nodes[i].yes);
  }
  fclose(file);
  printf("[%d␣animals␣stored]\n", (numNodes/2) + 1);
  printf("\nThanks␣for␣playing!\n");
```

42. This is just to call the function *printNodes*( ). I used it for debugging reasons, so I will mark it as a comment so that it doesn't execute in the program.

⟨ Print this 42 ⟩ ≡ /∗ printNodes(nodes,numNodes); ∗/

This code is used in section 4.

43. At the end of the program we'll call the *freeNodes* funtion to liberate the allocated memory. And after several days of work the program is finished :)

⟨ Clean up 43 ⟩ ≡

```
  freeNodes(nodes, numNodes);
```

This code is cited in section 39.

This code is used in section 4.

44. Index. The index shows where you can find what.

ANIMALS