COMP26120
Academic Session: 2018-19

Lab Exercise 7: The Pangolins Game

Duration: 2 lab sessions

For this lab exercise you should do all your work in your COMP26120/ex7 directory.

Copy the *makefile* from */opt/info/courses/COMP26120/problems/ex7*.

Learning Objectives

By the end of this lab you should be able to:

- Create and manipulate binary trees
- To implement depth-first traversal over trees
- To read and write tree-based data structures to file
- To implement breadth-first traversal over trees

Introduction

In this lab you are implementing a program to play a game. The program is going to require you to store information in a binary tree and serialise/deserialise this to file.

The Game Of Pangolins

There are two players: the computer and you. The game is straightforward.

- You think of an object
- The computer tries to guess the object, by asking you questions a series of 'yes/no' questions.
- If the computer guesses correctly, it wins.
- If the computer cannot guess your object, you win.
- The computer will ask you for three pieces of information so that it can guess the object correctly next time round.

1. The name of the object you were actually thinking of

2. A question (with a yes/no answer) that distinguishes between the computer's recent guess and your object
3. The answer to your question in the case of your object - yes or no.

Initially, the computer only knows about one object (it's a pangolin). It accumulates this information in a very naive way, so it can use it in subsequent rounds of the game. It's just a C programming exercise, not an attempt to create artificial intelligence.

Figure 1 provides an example game. Your input is prefixed by '>', and the computer's output is in bold text.

```
Does it have a tail?
> No
Is it flat, round and edible?
> Yes
Is it a pizza?
> No
Oh. Well you win then :-(
What were you thinking of?
> a biscuit
Please give me a question about a biscuit, so I can tell
the difference between a biscuit and a pizza
> Can you dip it in your tea?
What is the answer for a biscuit?
> Yes
Thanks
```

Figure 1: Example game

A longer version with 5 rounds and examples of the use of a tree to play the game are given in supporting reading material.

Description

Write a C program that plays the game of Pangolins.

Important: We have opted to not be very prescriptive on the input/output format for this exercise. The result is that the online tests mostly exercise your program rather than checking it in detail. You will be expected to demonstrate your game working to the TA during marking. This includes how it handles 'bad' inputs.

This exercise is split into 4 parts of increasing difficulty, with most of the marks awarded for the first 3 parts. You should try to complete the first two parts by the end of the first lab session, and the next two parts by the end of the second lab session. Note: as this lab is over 2 weeks it is worth twice the marks a 1 week lab is worth.

Part 1: Basic structures

You first need to design the data structures to store the current state of the game. They should form a binary tree, with objects and questions at the nodes and 'yes/no' decisions at the edges. The exact way you do this is up to you.

Step 1A

Define your data structure e.g. using a struct. For the Pangolins game, we have two different kinds of nodes: object(name)s and questions. There are several ways we could implement this in C. Figure 2 gives one possibility, using a single struct type for the two kinds of node. It's up to you to decide on the details of the data structure, and how you store strings. For example, if you are feeling adventurous, you might want to use an enum to say for each node whether it is an object or a question, and a union to hold the information about either the object or the question.

```c
struct node {
  // a string declaration to hold an object-name (which may be NULL)
  // a string declaration to hold a question (which may be NULL)
  struct node *yes_ptr; // only NULL for objects
  struct node *no_ptr; // only NULL for objects
}
```

Figure 2: Sample struct for node

Step 1B

Write a diagnostic function called *nodePrint* that will print out the status of a single node of the tree. (This should help you with debugging.)

Look at the decision tree at the end of round 5 of the example game in reading material. If we called the function *nodePrint* on the "does it like to chase mice" question, we might expect output something like this:

```
Object: [NOTHING]
Question: Does it like to chase mice?
Yes: A cat
No: a pangolin
```

Alternatively, calling the same function on the "a pizza" node would give output something like:

```
Object: a pizza
Question: [NOTHING]
```

Notice in this second example we have not printed out the 'Yes' and 'No' alternatives to the question since there isn't a question.

Step 1C

Now, you should 'hard-code' a small tree of questions and answers into your program: you'll need this both to test your *nodePrint* function and later to get the game off to a good start. We suggest you play the game on a piece of paper, and build up a small tree (say 6 or 7 nodes) of questions and objects. Now create a data structure to represent this, and add it to your program. There is no need to use dynamically memory at this stage (although you can) as you can just create the data structures using statically declared variables.

Step 1D

Now, write a new function - *treePrint* - which prints out the whole game tree you have just hard-coded. You should call *nodePrint* for each node. When printing out data-structures, the most important thing is to decide which order to print the nodes in.

There are many choices, the commonest is top-down (depth-first) left-to-right. Doing this recursively should be quite straightforward as you should recurse down one child completely before recursing down the other. Pseudo-code for this is given in Figure 3. Sometimes it is useful to keep track of the depth of a node and use this information to appropriately indent the data of each node.

```
void treePrint(struct node *ptr) {
  if (ptr == NULL)
    do nothing
  else {
    print the data from ptr
    // now print its children, left-to-right:
    treePrint(ptr->left_ptr);
    treePrint(ptr->right_ptr);
  }
}
```

Figure 3: Skeleton code for treePrint

Finish Part 1

Check that there are no memory issues with valgrind and submit your file as *pangolins_part1.c* file to the testing server. This should pass the first test only at this stage i.e. that it runs without memory bugs. Your code is expected to print your hard-coded tree.

Part 2: Making the game interactive

In this part we implement the input/output and the core logic of the game.

Step 2A: Performing input

When the game is being played 'live', it should allow for arbitrarily long questions and object names, and in principle the tree can be extended indefinitely. This means that it is simply not possible to use static arrays to store the tree or its contents; you'll have to dynamically allocate memory using *malloc*.

Use one of the standard 'input' functions from the C library (we'd recommend *fgets*) making sure to tell the function what the maximum number of characters to accept is, and copy the string from your fixed size static buffer into a snug-fitting dynamically allocated buffer within your game tree. **You can assume that questions in the test data fit into 80 characters.**

e.g. use *fgets* to read a whole line from the terminal or a file, and then (if you need to) use *sscanf* to extract the data you want from the line you've just read. As a reminder:

- char *fgets(char *str, int size, FILE *f);

  This reads the next complete line from file *f*. The line (including the end-of-line character, and terminated by \0) is written into the string pointed to by *str*. *str* must be a char array created by the programmer, of size *size*. *fgets* puts as much of the input line as it can into *str*, stopping when *size* is reached. If the read failed for some reason, the result of the function is *NULL*; if it was OK, the result of the function is the same as *str*.

- sscanf(char *str, format, vars...);

  This does exactly the same as *scanf*, but reads input from the string *str*, instead of from `stdin`.

Step 2B: A basic working game of pangolins

Once you have functions to deal with user input, coding the logic of the game is relatively straightforward, as shown in Figure 4. Inserting a new question into the tree, which has to be linked to the new object-name and to the computer's wrong guess, has some similarities to part of exercise 4 - inserting a person into the correct place in a sorted list.

We want to give you some creative freedom here. However, to be able to automatically test your code we have to assume some things. Here are the rules you must follow:

- The strings 'yes' and 'no' must always be acceptable answers to yes/no questions, but your program can support other answers (see below).
- The text responding to a finding out a guess is correct must contain the string "Good".
- Conversely, the text responding to an incorrect guess must contain the string "you win".
- Obviously, we must be able to use these strings to tell the two cases apart.
- Once an incorrect guess has been made the program should:
  - Accept a new question on a single line to differentiate the guessed and actual objects
  - Accept the answer to this question on a single line
  - Ask whether the game should continue

If in doubt, just copy the example game. The tests are quite coarse-grained and are only checking whether the correct guess is made or not. When being marked, the TA will manually run your code and try it out.

This would be a good place to get to by the end of the first lab session.

Step 2C: Dealing sensibly with user input

You should make your game robust to the types of input that a user might give.

1. If the user gives 'pangolin' rather than 'a pangolin' as a reply, does you program produce text that reads correctly the next time round? (i.e. does it assume that the user has given the correct article ('an', 'a', 'the' etc.) before the object name?)
2. What happens if the user forgets to put a question mark at the end of their question?
3. Can they type 'y' as well as 'yes' or 'n' as well as 'no'?
4. Could they even type 'absolutely' or 'correct'?

You won't be able to mitigate against all possible types of nonsense input, but given that the user is not trying to test an artificial intelligence program but might make a few genuine mistakes, can your program cope gracefully with these? The tests contain some example inputs that it is interesting to deal with but there is no expected output as it is up to you to decide how to deal with these (the TA will check).

```
initialise the tree // first version: just one object, a pangolin

// first version: play just one round
current node= root of tree
while (not finished) {
  if (current node is a leaf) { // object node
    make the guess
    if (user says guess is correct) {
      say computer has won
    } else {
      say user has won
      ask object-name and question
      insert the new object-name and question into the tree
    }
    finished
  } else { // question node
    ask the question
    set current node according to Yes/No response
  }
}
```

Figure 4: The algorithm for the game

Finish Part 2

For testing purposes you may have previously hard-coded some start state (beyond the initial object of a pangolin). For the part 2 tests you remove this hard-coded state.

Update: The tests assume that your tree is initialised with a single node consisting of the object named "a pangolin".

When your program exits, all the resources such as file handles and memory chunks that you have used get returned to the operating system. Make sure you have explicitly free-ed all the memory that has been malloc-ed by you. You might want to do this recursively. Check that there are no memory issues with valgrind and submit your file as *pangolins_part2.c* file to the testing server.

Part 3: Loading and saving the game

Of course, at the moment, every time you finish your program, it will forget about everything it has learned. Hardly what you'd want. Your task now is to write functions to load and save the question tree to a file. The format that you use for this is completely up to you - we will ask you to save a tree and then reload it and carry on playing.

Step 3A: Save your game tree to a text file

For this exercise, you just need to print the tree in human-readable form, and then input it again. We know that we can print a tree in many different orders, but does the order affect how easy it is to read it in again?

The simple answer is **no**, as long as the algorithm we use to read the tree does things in **exactly** the same order as we used to write it out

But there is a problem: to read the tree back in again, it has to be able to decide whether it is reading an object, which has no children, or reading a question, which has children. (You may be able to think of different but equivalent ways of describing the problem and then solving it.)

The simplest way to fix this is to change *treePrint* to say what kind of node it is printing, provided in Figure 5 . e.g. for the tree from round 5 of the example game in reading material, *treePrint* would output:

```
question: Does it have a tail?
question: Does it like to chase mice?
object: a cat
object: a pangolin
question: Is it flat, round and edible?
question: Can you dip it in your tea?
object: a biscuit
object: a pizza
object: Pete
```

```
void treePrint(struct node *ptr) {
  if (ptr == NULL)
    do nothing
  else {
    if (ptr is a question) {
      print "question:" and the question
      //now print the yes and no subtrees:
      treePrint(ptr->yes_ptr);
      treePrint(ptr->no_ptr);
    } else { // ptr is an object
      print "object:" and the object-name
    }
  }
}
```

Figure 5: Modified treePrint function

Step 3B: Read a saved game tree and recreate it in memory

Reading a saved game tree should do the reverse of writing the tree to memory. Skeleton code for the reverse of our psuedocode above is given as *treeRead* in Figure 6 . Again, you do not need to follow this method, you just need to make sure that when you read your written file you get back the same game state.

Step 3C: Load at start, Save at end

Finally, update your program so that the first thing it does is ask if the player wants to load a game and the last thing it does (when the player says they do not want to carry on playing) is to ask if the player wants to save the game. The logic of this should be to first accept a yes/no answer and then the name of the file.

```
struct node* treeRead() {
  read the next line of input
  if (nothing there) // i.e. no input
    return NULL;
  else {
    ptr = malloc (size of a struct node)
    if (the line started with "question:") {
      fill ptr with the question from the input line
      //now read the yes and no subtrees:
      ptr->yes_ptr = treeRead()
      ptr->no_ptr= treeRead()
    } else { // the line started with "object:"
      fill ptr with the object-name from the input line
      ptr->yes_ptr= ptr->no_ptr= NULL
    }
  }
}
```

Figure 6: Skeleton code for treeRead

Finish Part 3

Check that there are no memory issues with valgrind and submit your file as *pangolins_part3.c* file to the testing server.

Part 4: Detecting duplicates

This is a challenging last part and we do not expect all students to complete it. This is reflected in the fact that only a small number of marks are associated with this part.

Update: The marking requirements for this part have been updated. The marks are now only associated with *explaining* how breadth-first search could be used to detect duplicates and how it would be implemented.

In this last part you should extend the previous game so that when you add an object you check whether that object has been seen before. We want you to do this via a *breadth-first* search of the tree. You probably implemented a *depth-first* traversal of your tree for printing earlier. By completing these part you will have implemented both search methods. When is it better to use breadth-first search over depth-first search and vice versa? Your TA may ask you this.

In breadth-first search we go left-to-right and then top-to-bottom i.e. we explore all nodes at a depth of 1 and all nodes at a depth of 2 etc. To implement this we will need to *remember* the next nodes as you cannot directly follow the links in a node to its siblings at a similar depth (without significant changes to the data-structure).

Therefore, to implement breadth-first search you will need to use a *queue* data structure where you push on one end and pop from the other. You can achieve this with linked-lists.

Update your original code so that it checks whether an object has been added before and if it has it tells the player that they are cheating.

Check that there are no memory issues with valgrind and submit your file as *pangolins_part4.c* file to the testing server.

A final thought. Stepping through the elements of a data structure in a particular order is a very common task. The well-known iterator pattern suggests that separate the code needed to traverse the items of a collection from the code that operates on those items. What if we wanted to perform a range of different functions for all elements in the tree. How might you do this here?

Marking Scheme

Note that this may be refined to introduce extra cases reflecting special cases if required.

Is part1 complete and correct? Note that we do not have full tests and the TA may manually run your program.	
There is a sensible data structure for representingthe game state as a binary search tree. There is a sensible function for printing out this game state. There are no memory issues. There is a small hard-coded tree that is printed out when the program is run.	(2.5)
As above (top) but there are minor mistakes e.g. the data structure inefficiently creates empty leaves, the printing function produces poorly formatted output, or the hard-coded tree does not make sense. These are just examples.	(2)
As above (top) but there are memory leaks or invalid reads or writes. It is important that students understand how to eliminate these.	(1)
As above (top) but there is no hard-coded tree that can be used to test the printing function.	(1)
An attempt has been made but it is incomplete/does not work.	(0.5)
No attempt has been made.	(0)

Does the student understand the structure of binary trees?	
The student can draw a binary tree and explain its structure, they can explain what the size and height of a tree are, they can describe the number of calls of treePrint required to print the tree in terms of its size, and they can explain how their implementation is used to represent binary trees. All questions are answered with an impressive level of understanding.	(2.5)
As above but answers are good/satisfactory rather than impressive.	(2)
As above but there are some minor points where there is some confusion e.g. related to the height of a tree or complexity of treePrint.	(1.5)
As above but there is a significant gap in understanding i.e. one topic the student cannot comment on.	(1)
The student struggles to explain any of the points.	(0.5)
No attempt has been made	(0)

Is part 2 complete and correct? Note that we do not have full tests and the TA may manually run your program.	
The game can be played as expected. All the tests pass and in addition the TA can manually test the game to ensure that the correct responses are given e.g. when a new object is added and then the same object is guessed again.	(2.5)
As above but there are some minor mistakes e.g. the wrong things are printed at some points.	(2)
What is implemented works but there is some functionality missing e.g. asking the player if they want to play again.	(1)
All functionality has been implemented but it is flawed in a major way e.g. the tree is not updated correctly.	(1)
The are memory errors. It is very important that we get used to removing these.	(1)
An attempt has been made but the game does not work.	(0.5)
No attempt has been made	(0)

Does the solution to part 2 handle unexpected input correctly? There are no tests for this, the TA will test it manually	
All unexpected inputs entered are handled appropriately	(1)
At least one unexpected input entered causes a crash or similar	(0.5)
There has been no effort to guard against unexpected inputs	(0)

Does the student understand how binary search works in a binary tree and how trees are updated?	
The student has impressive understanding. The student can explain how their code searches through the binary tree on each guess and how the tree is updated when a new object needs to be added. They might be asked to draw this process of adding a new node and how this is implemented using pointers. In this game new nodes are only added at leaves, the student can explain what (if anything) would be different if a node needed to be added within the tree itself. The student can explain how a node might be removed from a tree, commenting on similarities to the process of adding nodes.	(3)
The student has good understanding. They can explain the first two points above (how the tree is searched and how nodes are added) well and might make an attempt at the other points.	(2.5)
The student has acceptable understanding. They can generally explain the first two points above (search and addition of nodes) but make small mistakes.	(2)
The student requires a lot of prompting to answer the above questions (search and addition of nodes).	(1)
There are major points of confusion; this topic needs to be revised.	(0.5)
No attempt has been made	(0)

Is part 3 complete and correct? Note that the TA may need to run manual tests.	
The tree can be written to a file succesfully and read back in. The logic for when this happens has been implemented correctly and the test passes. In addition to the test, the TA may run manual tests to check special cases.	(2.5)
As above but some minor mistakes have been made	(2)
The solution partially works but there is a major problem e.g. the tree can be written to a file but not read back in correctly.	(1)
An attempt has been made but it is incomplete and/or incorrect.	(0.5)
No attempt has been made	(0)

Does the student understand their solution to part 3?	
The student can explain how their code writes the tree to a file in such a way that it can be read back in. Given an example tree they can explain in which order the nodes would be written to file.	(2)
The student has some minor points of confusion when explaining their solution but can generally explain what it is doing.	(1.5)
The student cannot satisfactorily explain their solution.	(0.5)
No attempt has been made	(0)

The student can explain how to implement breadth-first search of a binary tree and how this can be used to solve the problem introduced in part4. It is not necessary to have implemented this but clearly implementing the solution will aid understanding.	
The student can explain the general algorithm for breadth-first search and the role of a queue in this implementation. They can explain how a queue could be implemented in C. They can explain the time and space complexity of breadth-first search in terms of properties of the tree and discuss whether these differ from depth-first search. The student can explain how this search can be used to find duplicates in a tree and might be able to comment on whether a different search method might be more efficient.	(2.5)
The student can explain the general algorithm for breadth-first search and the role of a queue in this implementation.	(2)
The student can explain the difference between depth-first search and breadth-first search but not how breadth-first search is implemented.	(1)
No attempt has been made	(0)

The code is of good quality throughout	
The code quality is excellent - the student has factored out common code (e.g. for reading inputs) and documented their solution.	(1.5)
The code quality is good - there are comments and good variable names throughout.	(1)
The code quality is satisfactory.	(0.5)
The code qualtiy is below a satisfactory level.	(0)