Information Processing 2 — Week 12 exercises

Complete the first four exercises in-class on your own laptop. Ask an instructor to check your work during the class, as soon as you finish answering each question. A **late penalty of 50%** will be applied to answers checked after class.

Complete the remaining exercises in-class if possible, or before the start of the next exercise class at latest. Exercises checked **more than one week** after the class will have a **50% late penalty** applied.

1 Binary tree node

Write a structure definition suitable for storing words (as strings) in a binary tree. The structure should contain a pointer to the characters in a `word` and pointer to the `left` and `right` sub-trees (or 0 wherever there is no sub-tree). Use a `typedef` to give your structure the friendly name '`Node`'.

Check your definitions using the following program (download '`12-01-node.c`'):

```
int main()
{
    struct Node l = {  "left",  0,  0 };
    struct Node r = { "right",  0,  0 };
    struct Node n = {  "root", &l, &r };
    struct Node *root = &n;

    printf("%s\n", root->left->word);
    printf("%s\n", root->right->word);
    printf("%s\n", root->word);

    return 0;
}
```

Verify that the output is: `left right root`

▷▷ Ask an instructor to check and record your work.

2 Constructor function

Write a constructor function for your nodes called `Node *newNode(char *word)`. Use `malloc()` to obtain memory for the new node. Store a *copy* of the word (created with `strdup()`) into the `word` member of the new node. Initialise the other two members (`left` and `right`) to 0.

Test your function by creating some nodes and printing their contents (download '`12-02-node.c`'):

```
int main()
{
    struct Node *l    = newNode("left");
    struct Node *r    = newNode("right");
    struct Node *root = newNode("root");
    root->left  = l;
    root->right = r;

    printf("%s\n", root->left->word);
    printf("%s\n", root->right->word);
    printf("%s\n", root->word);

    return 0;
}
```

▷▷ Ask an instructor to check and record your work.

3 Finding words in the tree [2 points]

Write a function `Node *findNode(Node *tree, char *word)` that finds a node with a matching `word` in the given `tree` and returns a pointer to that node. The function should assume that the tree is sorted, i.e., that nodes in the tree are arranged so that the `word` in a given node is larger than all the words in its `left` sub-tree and smaller than all the words in its `right` sub-tree. If there is no node in the tree matching the given `word`, return 0 (the 'null' pointer).

Hint: If the first parameter `tree` is 0 then the search has followed a pointer to a sub-tree that does not exist. In other words, `word` is not in the tree and you can immediately return 0 as the result.

Hint: Use `strcmp(char *s, char *t)` to compare two words. It returns

`strcmp(s, t)` < 0 if `s` $<$ `t` (in dictionary order),
`strcmp(s, t)` > 0 if `s` $>$ `t`, and
`strcmp(s, t)` $= 0$ if `s` and `t` are equal (contain the same characters).

Hint: once you know if the target is smaller or larger than the word in the current node you can call `findNode()` recursively to search the `left` or `right` sub-tree, as appropriate.

Test your function with the following program (download '`12-03-findNode.c`'):

```
int main()
{
    struct Node *l    = newNode("left");
    struct Node *r    = newNode("right");
    struct Node *root = newNode("middle");
    root->left  = l;
    root->right = r;

    printf("%s\n", findNode(root,   "left")->word);
    printf("%s\n", findNode(root, "middle")->word);
    printf("%s\n", findNode(root,  "right")->word);
    printf("%p\n", findNode(root,   "none")      );

    return 0;
}
```

Verify that the output is: `left middle right 0x0`

▷▷ Ask an instructor to check and record your work.

4 Inserting new words into the tree: part 1

If the target `word` is not in the `tree` then `findNode()` eventually meets a '`Node *`' pointer whose value is 0 indicating the node was not found where it was expected to be. Instead of returning 0 we could insert a new node into the tree in the place where the 0 pointer was found. The problem is that `findNode()` knows that a 0 pointer has been encountered but it no longer knows where that 0 pointer was located.

Modify `findNode()` so that its first parameter is a pointer to the variable where the pointer to the tree is stored: `Node *findNode(Node **treep, char *word)`. The body of the function will remain almost the same, with the following differences:

1. At the start of the function you will have to fetch the pointer to the tree from the place where it is stored: `Node *tree = *treep;`
2. When calling `findNode()` recursively you will have to pass the address of the `left` or `right` member holding the sub-tree to be searched. For example:

```
if (cmp < 0) return findNode(&tree->left, word);
```

Test your function as above, except using `findNode(&root, "word")` in the `main()` function. The output should (obviously) be the same.

▷▷ Ask an instructor to check and record your work.

5 Inserting new words into the tree: part 2

Modify `findNode()` to insert new words into the tree. When it discovers that `word` is not already in the tree it should create a new node containing the missing word and then replace the 0 'null' pointer in the tree with a pointer to the new node.

Hint: All you need to modify is the line in `findNode()` that returns 0.

The `findNode()` function can now construct the entire tree. Test your modified function with the following program (download '`12-05-findNode.c`'):

```
int main()
{
    struct Node *root = 0;
    findNode(&root, "middle");
    findNode(&root, "left");
    findNode(&root, "right");

    printf("%s\n", findNode(&root,   "left")->word);
    printf("%s\n", findNode(&root, "middle")->word);
    printf("%s\n", findNode(&root,  "right")->word);

    return 0;
}
```

▷▷ Ask an instructor to check and record your work.

6 Printing the ordered binary tree [2 points]

Write a function `void printTree(Node *tree)` that prints the `word`s stored in the `tree` in ascending alphabetic order. Hint: To print the words in `tree`, print the `word`s in the `tree`'s `left` sub-tree, then the `word` in the `tree` node itself, then the `word`s in the `right` sub-tree.

Test your function with the following program (download '`12-06-printTree.c`'):

```
#define indexableSize(A) (sizeof(A) / sizeof(*(A)))

int main()
{
    Node *root = 0;
```

```
    char *words[] = {
        "Peter", "Piper", "picked", "a", "peck", "of", "picked", "pepper",
        "where", "is", "the", "peck", "that", "Peter", "Piper", "picked",
    };

    for (int i = 0;  i < indexableSize(words);  ++i)
        findNode(&root, words[i]);

    printTree(root);

    return 0;
}
```

Verify that the output is: `Peter Piper a is of peck pepper picked that the where`

▷▷ Ask an instructor to check and record your work.

7 Constructing the tree with words read from the input [2 points]

Write (or reuse a previous implementation of) `int getword(char word[], int size)` to read a single word from the input. Consider any sequence of alphabetic characters to be a word, separated from the next word by a sequence of any non-alphabetic characters.

Modify your `main()` function so that it reads words from the input and inserts them into the tree. When it reaches the end of input, print the tree.

Test your program using the file `IP2-12.txt` (downloaded from Teams) as input:

```
cat IP2-12.txt | ./12-07-readWords
```

or:

```
./12-07-readWords < IP2-12.txt
```

Verify that the output is a list of the 41 unique words in `IP2-12.txt`, in alphabetical order, with each word printed exactly once.

▷▷ Ask an instructor to check and record your work.