

Report - assignment 4

Camilla Kristiansen Birkelund

IDATT2101

September 18th, 2022

Assignment description

There were two parts to the assignment. The first part was to create a program with one method to add numbers, and one to subtract, by using a double linked list. The user should be able to enter long numbers to use for calculation (longer than 20 integers).

The second part of the assignment was to implement a binary search tree that saves a word in each node. The program is supposed to allow the user to write a random amount of words that will be added to the tree in the order they are given in. The program should then print the first four levels of the tree.

Implementation

The source code for the tasks can be viewed in the files “linkedlists.c” and “trees.cpp”. Most of the code concerning structures and operations on the nodes in the lists were retrieved from the syllabus book, “Algoritmer og datastrukturer med eksempler i C og Java” (Hafting & Ljosland, 2014), and will therefore not be discussed in detail.

First task: linked lists

One of the methods implemented to solve this part of the assignment was to perform an addition of the integers in the linked lists. The solution presented is a while-loop containing three if-statements, that iterates through the given numbers starting from the last digit. The first if-statement investigates whether or not there are any digits left in the number on the left hand side (which will be the first number provided), and adds the current digit to the corresponding position in the new number (valuesum). The next if-statement has the same functionality for the number at the right hand side (the second number provided). The last if-statement breaks the loop when there are no digits left in either of the numbers. For each time the loop is run, the potential carry value will be added to the next digit in the iteration.

```

//While-loop to add the numbers.
while (1) {
    valuesum = 0;

    //If the left hand side number is not done, find the current digit and add it to valuesum.
    if (!done(lhs_iter)) {
        lhs_digit = lhs_iter->position;
        valuesum += lhs_digit->value;
    }

    //If the right hand side number is not done, find the current digit and add it to valuesum.
    if (!done(rhs_iter)) {
        rhs_digit = rhs_iter->position;
        valuesum += rhs_digit->value;
    }

    //If both numbers are done, and there is no carry value, break the loop.
    if (done(lhs_iter) && done(rhs_iter) && !carry) {
        break;
    }

    valuesum += carry;
    carry = valuesum / 10;

    putFirst(sum, (valuesum % 10));

    previous(lhs_iter);
    previous(rhs_iter);
}

```

Figure 1: An illustration of the while-loop in the add-method in the source code.

The method to perform a subtraction has the same implementation regarding the iteration through the numbers starting from the last digit. The subtract-method also uses a while-loop that breaks when there are no more digits in the numbers to iterate through. First, the value of the digit at the current position in the number on the left hand side (the first given number) is added to “valuediff” which is the corresponding digit in the resulting number. Secondly, the value of the digit at the current position in the number on the right hand side (the second given number) is subtracted from “valuediff”. A check is then performed to see if “valuediff” is a negative value. If it is, a value of 10 is added to “valuediff”, and a value of 1 is subtracted from the next iteration for “valuediff”.

```

while (1) {
    valuediff = 0;

    //If we are not done iterating through the number on the left hand side, add the current digit to the difference value
    //at the current position.
    if (!done(lhs_iter)) {
        lhs_digit = lhs_iter->position;
        valuediff = lhs_digit->value - borrow;

        //If we are not done iterating through the number on the right hand side, subtract the current digit from the
        //difference value at the current position.
        if (!done(rhs_iter)) {
            rhs_digit = rhs_iter->position;
            valuediff -= rhs_digit->value;
        }

        //If the digit at the current position is a negative value, borrow 10 from the next number and subtract 1 from that
        //same number.
        if (valuediff < 0) {
            valuediff += 10;
            borrow = 1;
        } else {
            borrow = 0;
        }
    }

    //If both numbers are done, break the loop.
    if (done(lhs_iter) && done(rhs_iter)) {
        break;
    }

    putFirst(difference, valuediff);

    previous(lhs_iter);
    previous(rhs_iter);
}

```

Figure 2: An illustration of the while-loop in the subtract-method in the source code.

Second task: trees

The solution to creating a binary search tree was first to implement a node structure, a way to create a new node, and a way to insert nodes into trees. To insert a node into a tree, the chosen method was to use the `string.h` library in `c++`. This provides the possibility of comparing characters according to their ASCII value, which helps to order the words alphabetically. The insert method will also ignore duplicates, so that each node contains a unique word.

```

// Insert a node into a binary tree.
void insert(Node *root, const char *word) {
    // If the current node in the tree is empty, put the word there.
    if (root == NULL) {
        root = newNode(word);

        // If the word is equal to the word in the current node in the tree,
        // do nothing (this avoids duplicates).
    } else if (strcmp(word, root->word) == 0) {
        return;

        // If the new word is before the word in the current node alphabetically,
        // place the word as the left child to the current node.
    } else if (strcmp(word, root->word) < 0) {
        if (root->left == NULL) {
            root->left = newNode(word);
        } else {
            insert(root->left, word);
        }

        // If the new word is after the word in the current node alphabetically,
        // place the word as the right child to the current node.
    } else if (strcmp(word, root->word) > 0) {
        if (root->right == NULL) {
            root->right = newNode(word);
        } else {
            insert(root->right, word);
        }
    }
}

// Method to print the resulting data.
void printResult(Node *root) {
    if (root) {
        printResult(root->left);
        printf("%s ", root->word);
        printResult(root->right);
    }
}

```

Figure 3: Illustration of the insert-method in the source code.

To print the order of the tree (in the order of breadth first search), the printResult-method was implemented. This method will print all words in the tree from left to right.

```

// Method to print the resulting data.
void printResult(Node *root) {
    if (root) {
        printResult(root->left);
        printf("%s ", root->word);
        printResult(root->right);
    }
}

```

Figure 4: Illustration of the printResult-method in the source code.

In addition to the printResult-method, the method printTree was implemented to print an illustration of the tree. The standard library from c++ provides a queue functionality, which was used to traverse through the nodes. The method uses two queues, to keep track of which nodes belong to which level in the tree in the illustration.

```
// Method to print the resulting tree.
void printTree(Node *root, int width) {
    // Using two queues, one for the all the nodes, and one temporary queue to
    // hold nodes for each level in the tree.
    Node *current;
    std::queue<Node *> queue;
    std::queue<Node *> temp;
    temp.push(root);
    int padlen;
    const char *toPrint;

    while (!temp.empty()) {
        queue.swap(temp);

        while (!queue.empty()) {
            current = queue.front();
            queue.pop();

            if (current) {
                //If the current node exists, add it to the tree and create two children (left and right).
                toPrint = current->word;
                temp.push(current->left);
                temp.push(current->right);
                //If the current node does not exist, enter a blank string to keep the tree structure.
            } else {
                toPrint = "";
            }

            //Formatting to keep the nodes centered.
            padlen = (width - strlen(toPrint)) / 2;
            printf("%s%s%s", padlen, "", toPrint, padlen, "");
        }

        printf("\n");
        width = width / 2;
    }
}
```

Figure 5: Illustration of the printTree-method in the source code.

Testing

Testing for the first task

To test the functionality of the add- and subtract-methods, two examples that were given in the assignment were used.

```
java Langetall 10000000001999999999001 + 100007
      10000000001999999999001
+
      100007
= 1000000000200000000099008
```

Figure 6: The provided example for addition in the task.

```
10000000001999999999001
+
      100007
= 1000000000200000000099008
```

Figure 4: The result of the program adding the same values.

```
java Langetall 840000000000000200000 - 100000000007000060004
      840000000000000200000
- 100000000007000060004
= 739999999993000139996
```

Figure 7: The provided example for subtraction in the task.

```
840000000000000200000
- 100000000007000060004
= 739999999993000139996
```

Figure 8: The result of the program subtracting the same values.

Testing for the second task

To test the functionality of the second task, the printResult-method was checked to see if the breadth first search provided the expected result, which would be expected to be an alphabetically sorted version of the input values.

```
./trees hode bein hals arm tann hånd tå
```

Figure 9: The input values to test the printResult-method.

```
arm bein hals hode hånd tann tå
```

Figure 10: The result for the test of the printResult-method.

The printTree-method was tested by comparing the print to the expected tree structure of the given input values.

```
java Søketre hode bein hals arm tann hånd tå
                hode
            bein
        arm
    hals
        hånd
            tann
                tå
```

Figure 11: The expected tree structure of the input-values.

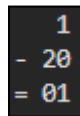
```
                hode
            bein
        arm
    hals
        hånd
            tann
                tå
```

Figure 12: The resulting tree structure from running the printTree-method.

Results

Results for the first task

The implementation of the first task worked as expected when calculating numbers with more than 20 digits, both with addition and subtraction. It will, however, provide an incorrect result if the difference between the values is a negative number when using the subtraction method.



```
  1  
- 20  
= 01
```

Figure 13: The print when the result is a negative number.

As the assignment didn't require the subtraction method to be able to handle negative results, this functionality was not included in the final solution due to time limitations. A possible solution that does not involve implementing this feature would be to provide an error message instead of the wrong result when the result is a negative value.

Results for the second task

The implementation of the second task fulfills all the requirements from the assignment description, but has room for improvements in the printTree-method. The printTree-method works exactly as it should, but the illustration can become messy if there are too many nodes, or the nodes contain long words. This can cause some of the nodes to appear to be in the wrong placement, even though they are not (this can be confirmed in the printResult-method).

Sources

Hafting, H. & Ljosand, M. (2014) *Algoritmer og Datastrukturer med eksempler i C og Java*.
Kopinor Pensum AS, Oslo. (Used in the source code).