Dirk Praetorius,
Michele Ruggeri

Wintersemester 2017/18
14.12./15.12.2017

Übungen zur Vorlesung
Einführung in das Programmieren für TM

Serie 8

Aufgabe 8.1. Write a structure `cMatrix` for the storage of $(m \times n)$-matrices $A \in \mathbb{C}^{m \times n}$ with complex entries. Use the structure `cdouble` from Exercise 7.1 and Exercise 7.2. Furthermore, write the functions `newCMatrix`, `delCMatrix`, `getCMatrixM`, `getCMatrixN`, `getCMatrixCoeff`, and `setCMatrixCoeff`. Test your implementation accurately! Save the source code, split into a header file `cmatrix.h` and `cmatrix.c`, into the directory `serie08`.

Aufgabe 8.2. A complex square matrix $A \in \mathbb{C}^{n \times n}$ is hermitian if $A_{jk} = \overline{A_{kj}}$ for all $j, k = 1, \dots, n$. Write a function `int isHermitian(cMatrix* A)`, which analyzes the structure of a matrix $A \in \mathbb{C}^{n \times n}$. The function returns the value `1` in case of an hermitian matrix and the value `0` otherwise. Use the structures and the features from Exercise 7.1, Exercise 7.2, and Exercise 8.1. Test your implementation in a suitable way! Which property characterizes the diagonal entries of a hermitian matrix? Why? Save your source code as `ishermitian.c` into the directory `serie08`.

Aufgabe 8.3. A *binary tree* is a data structure that can be used for an efficient search for data. Such a tree consists of nodes which contain datas as well as connections to other nodes. In particular, a binary tree has at most two connections to other nodes which are called *left* or *right child*. The node without parents is called *root* of the tree.
Example: In Figure 1, the node with value 15 is the *left child* of the *root* and 20 is the *right child* of 15.

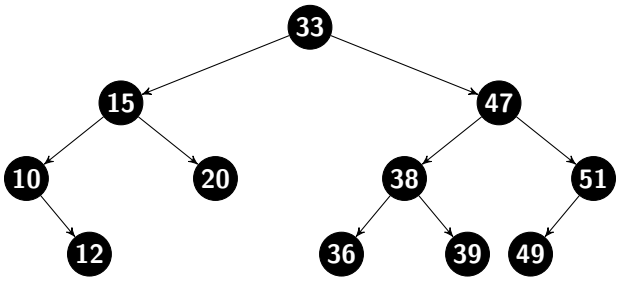

Abbildung 1: A binary tree which satisfies the search tree property.

Write a data structure type `node` which saves an `int`-value and two pointers on the left and the right child nodes. Further, write a data structure type `tree` which saves a pointer on the root of the tree. Write functions `node* newNode(int data, node* leftChild, node* rightChild)` and `tree* newTree()` that generates an empty tree and the get-functions `int getData(node* someNode)`, `node* getLeft(node* someNode)` and `node* getRight(node* someNode)`. How did you test your code for correctness? Save your source code as `treestruct.c` into the directory `serie08`.
Remark: Note here and in the following exercises that we use `NULL` to indicate that a node does not have a child at the considered place or that the tree is empty.

Aufgabe 8.4. Implement a binary search for the tree structure of Aufgabe 8.3. Orientate yourself at slide 85 of the lecture. What is the computational complexity depending on the depth of the tree? How did you test your code for correctness? Save your source code as `binsearch.c` into the directory `serie08`.

Aufgabe 8.5. The data structure of Aufgabe 8.3 represents a binary tree. Binary *search tree* are binary tree whose nodes are arranged in a certain way. Concretely, this means that for each node K

- all left children, grand children, etc. save a value which is smaller than the value of K itself.

- all right children, grand children, etc. save a value which is larger or equal to the value of K itself.

Write a function `void insert(tree* myTree, int content)` which inserts the input value `content` into the tree `myTree` such that the *search tree property* from above is maintained. Therefore, one proceeds as follows: If the tree is empty, we generate a new root with `content` as content. Otherwise, we check if the new node has to be inserted into the left subtree of the root. If this is the case, we (recursively) consider the same question for the left subtree, and otherwise for the right subtree. If we have reached the point where the corresponding subtree is empty, we insert a new node with the content `content` at this place. What is the computational complexity for this operation depending on the depth of the tree? How did you test your code for correctness? Save your source code as `insert.c` into the directory `serie08`.
Hint: Write a recursive function `void insertAtNode(node* rootnode, int content)` and use:

```
void insert(tree* myTree, int content){
   if(myTree->root==NULL)
      myTree->root = newNode(content, NULL, NULL);
   else
      insertAtNode(myTree->root, content);
}
```

Aufgabe 8.6. A binary search tree can also be used to sort a vector. First, one fills the tree stepwise with the entries of the vector. Then, the tree can run through in the correct order. Hereby, the „correct“ order looks as follows:
For a node K:

- Display (recursively!) in the same manner the left subtree on the screen.
- Display the value of the node K on the screen.
- Display (recursively!) in the same manner the right subtree on the screen.

Write a function `fillTree(tree* T, Vector* v)`, which fills the tree with the entries of the vector via `insert`. Use the `Vector` structure from the lecture. Moreover, write a function `void printSorted(tree* T)`, which prints the tree in the above mentioned way on the screen. How did you test your code for correctness? Save your source code as `treeSort.c` into the directory `serie08`.
Hint: It is useful, to write a recursive function `void printSortedAtNode(node* rootnode)`. Afterwards use:

```
void printSorted(tree* myTree){
  printSortedAtNode(myTree->root);
}
```

C++ Exercises

Aufgabe 8.7. Write a class `customer` for a bank customer. The class contains the name of the customer as `string`, the current balance as `double` and a pin code as `int`. Implement `set` and `get` methods for the member variables as well as the following class methods

- `void printBalance()`
  prints the current balance on the screen.
- `bool checkPIN()`
  reads in a PIN code and checks whether it is correct or not.
- `void drawMoney()`
  checks the a given PIN code, reads in the amount the customer wants to draw, and prints the new balance on the screen. The account must not be overdrawn. If necessary, print a warning on the screen.

How did you test your implementation? Save your source code as `customer.{hpp,cpp}` into the directory `serie08`.

Aufgabe 8.8. Write a class `Stopwatch` that simulates a stopwatch. The stopwatch has the following two methods. If the first method is called, then the time measurement starts. If the method is called again, the time measurement stops. The second method is used to reset the time to zero. To realize this situation, implement the methods `pushButtonStartStop` and `pushButtonReset`. Implement another method that prints out the time formatted in the style `hh:mm:ss.xx`, e.g., if the measured time is two minutes, then the output should be `00:02:00.00`. Save your source code as `Stopwatch.{hpp,cpp}` into the directory `serie08`.
Hint: Use the data-type `clock_t` and the function `clock()` from the library `time.h`. It makes sense to use a variable `isRunning` of type `bool`. If the method `pushButtonStartStop`, then this variable is either set to `true` or `false`.