Aufgabe 4: Wir programmieren den ersten Teil des Dacjuk Algorithmus (noch ohne replace_or_register). Beachten sie bitte die Musterlösung im gitlab:

https://gitlab.cip.ifi.lmu.de/cis/basis_modul_ws2018/tree/loesung/aufgabe3

Hier ein Vorschlag zu einem Header File:

```
#ifndef CELLFSA_HPP
#define CELLFSA_HPP CELLFSA_HPP
#include <map>
#include <vector>
#include <iostream>
#include <memory>


       class CellFSA {

       public:
           /**
            * @brief Internal datatype for the representation of the @c State 's transitions
            */
            using CellFSAPtr = std::shared_ptr<CellFSA>;
            using TransMap_t = std::map< wchar_t, CellFSAPtr>;
            typedef std::map< wchar_t, CellFSAPtr> TransMap_tt;

           /** Defaultconstruktur to get a new Cell
            * @param Internal Cell Number, simulates Cell-Label, later used as Storingindex
            */
           CellFSA( size_t number ) {
           }
           /**
            * @return true if it has Children
            */
           bool has_children() {
             }

           /**
            * finds a transition according the Character c
            * @param Transition Character c
            */
           CellFSAPtr delta( wchar_t c ) {
           }

           /**
            * Sets Cell to final
            */
           void setFinal( bool b )  { isFinal_ = b; }

           /**
            * replace Cell
            */
           void replaceState( CellFSAPtr newstate )  {
           }

           /**
            * replace Cell
            */
           void deleteState(  )  {
           }


           /**
            * add new Transition to cell
            * @param Transition Character c and Pointeradress of Cell to point to
            */
           void addTransition( wchar_t c, CellFSAPtr state_adress) {
            }

           /**
            * returns the lastchild_ Pointer
```

```
             *
             */
            const CellFSAPtr last_child() {
            }
            /**
             * @return if Cell is final
             */
            bool isFinal() const {

             }

            /**
             * @return Number of Out Transitions
             */
            int num_out_trans() const {

             }

            // @return Adresses of Transition MAPS
            TransMap_t& get_transitions () {
            }

            // @return Iterator Adresses
            const TransMap_t& get_transitions () const {
            }

           /**
             * @return the @c state's number
             */
            size_t getNumber() const {

             }


        private:
            /** Cellnumber, internal Label
             *
             */
            size_t number_;

            /**
             * Transitions
             * Every Transition has a
             *      Transition Char
             *      Pointer to the Cell, where the transition points to
             */
            TransMap_t transitions_;

            /** Shows it the Cell is a final Cell
             *
             */
            bool isFinal_;

            /**
             * Char of last inserted Value
             */
            wchar_t lastchild_label;

            CellFSAPtr lastchild_;

            /** Number of outgoing transitions
             *
             */
            int num_out_trans_;

        }; // class CellTrie



#endif
```

1. Versuchen Sie nun ihr erstes Programm zu implementierten, das den ersten Teil des Dacjuk Algorithmus nachvollzieht.
2. Bauen Sie in Ihr Programm Kommentare ein, entsprechend den Vorgaben von „doxygen“.
3. Ein wichtiger Vergleich ist die Überprüfung auf Äquivalenz zweier Zustände. Achten Sie bei Ihrer Datenstruktur, dass der Vergleich der Äquivalenz zweier TRIE-Zellen Node1 und Node2 möglichst einfach ist.
    a. Schreiben Sie eine Routine:

```
is_equivalent(Node1, Node2):
```


state. For a given state p (not in the register), we try to find a state q in the register that would have the same right language. To do this, we do not need to compare the languages themselves—comparing sets of strings is computationally expensive. We can use our recursive definition of the right language. State p belongs to the same class as q if and only if:

1. they are either both final or both nonfinal; and
2. they have the same number of outgoing transitions; and
3. corresponding outgoing transitions have the same labels; and
4. corresponding outgoing transitions lead to states that have the same right languages.

Because the postorder method ensures that all states reachable from the states already visited are unique representatives of their classes (i.e., their right languages are unique in the visited part of the automaton), we can rewrite the last condition as:

4′. corresponding transitions lead to the same states.

If all the conditions are satisfied, the state p is replaced by q. Replacing p simply involves deleting it while redirecting all of its incoming transitions to q. Note that all leaf states belong to the same equivalence class. If some of the conditions are not satisfied, p must be a representative of a new class and therefore must be put into the register.