https://github.com/tibumanu/uni-stuff/tree/main/year%203%20sem%201/FLCD/lab2
Țibu Cristian, 937/2

lab2, week4

Documentation for Symbol Table using a Binary Search Tree

The basis for our tree is the following data type:

```
struct Node{
    std::string key;
    int index;
    Node* left;
    Node* right;
};
```

This encodes the information held by each element of the tree: its key (i.e. the name of the identifier or the constant represented by this `Node`), its index (i.e. the natural number value that would be assigned to each entry in the PIF table, or if this were to be reduced to a table-like structure), and its two children.

The class SymbolTree has only public fields and methods.

```
class SymbolTable{
private:
   public:
    Node* rootIdentifiers;
    Node* rootConstants;
    int size, lastIdentifierIndex, lastConstantIndex;
}
```

This class, defining our Symbol Table, contains two roots and hence two trees - one for the identifiers and another for the constants of our Symbol Table. The root of each is stored in the pointer variables `rootIdentifiers` and `rootConstants` respectively. The remaining members are of type `int` and represent the total size of the tree, `size`, the last-used index for identifiers, `lastIdentifierIndex`, and the last-used index for constants, `lastConstantIndex`.

The following methods are available:

```
SymbolTable();
Node* putIdentifier(std::string key);
Node* getIdentifier(Node* x, std::string key);
Node* minIdentifier();
Node* maxIdentifier();
bool containsIdentifier(std::string key);

Node* putConstant(std::string key);
Node* getConstant(Node* x, std::string key);
Node* minConstant();
Node* maxConstant();
bool containsConstant(std::string key);

void printIdentifiersPostOrder(Node* x);
void printIdentifiersLevelOrder(Node* x);
void printConstantsPostOrder(Node* x);
void printConstantsLevelOrder(Node* x);
```

The `SymbolTable()` constructor initializes both roots to `NULL` and the int members to `0`.

We'll begin detailing the rest of the methods.

```
Node* putIdentifier(std::string key);
```

This method is to be used for identifiers. It takes the given key and puts it in the tree, maintaining the structure of the BST. It creates a new `Node` and assigns it the given key and index, afterwards returning said `Node`; its children will both be `NULL`.

```
Node* putConstant(std::string key);
```

More or less the same can be said about this method, however we have some necessary distinctions to be made. First and evidently, this method is to be used for constants. Second, the key of the to-be-added constant is still a string – e.g. the string constant `"testString"` and the integer constant `5` will have the keys `"testString"` and `"5"` respectively, both of the type string in our C++ program. This avoids unnecessary possible hurdles in trying to store them differently, and does not impede the BST structure and its associated algorithms for storing data.

```
Node* getIdentifier(Node* x, std::string key); // x is from where we start the search
Node* getIdentifier(Node* x, std::string key); // x is from where we start the search
```

These two methods search for an Identifier and a Constant respectively, and, if found, return a pointer to the `Node` data type inside the tree. The parameter to be given is the starting `Node` – for searching the entire tree, this should be the root of the tree. The methods work recursively, making use of the lexicographic order present in the tree and calling themselves correspondingly.

```
Node* minIdentifier();
Node* maxIdentifier();
Node* minConstant();
Node* maxConstant();
```

These methods return a pointer to the leftmost (for min) and the rightmost (for max) leaf of the two stored trees, or NULL if such leaves do not exist.

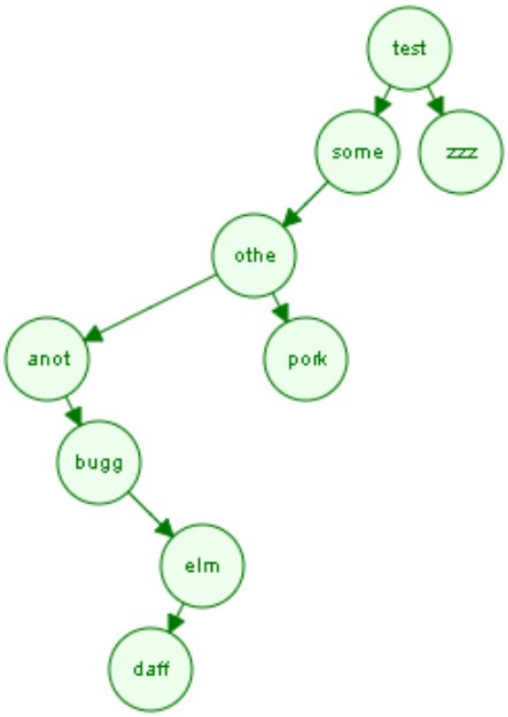
```
bool containsIdentifier(std::string key);
bool containsConstant(std::string key);
```

These methods return a boolean value representing whether or not a given key (or rather, a `Node` with a corresponding key value) can be found in the tree of Identifiers and in the tree of Constants respectively.

```
void printIdentifiersPostOrder(Node* x);
void printIdentifiersLevelOrder(Node* x);
void printConstantsPostOrder(Node* x);
void printConstantsLevelOrder(Node* x);
```

These methods are used for printing. The given argument, for printing the respective trees, should be the root of each tree. As their names suggest, printing is done in either Post-Order or Level-Order. The Post-Order printing methods utilize recursive calls, while the Level-Order ones use a queue. The of a `Nodes` is done on separate lines, with one line for each node; the key will be printed, followed by a whitespace character and the index. See the following screenshots:

Unfortunately the given keys are truncated in this visual representation of the tree, but the keys are: test, someOtherTest, otherTest, anotherTest, zzz, buggsBunny, elmerFudd, daffyDuck and porkyPig.




```
Identifiers post order:

daffyDuck 7
elmerFudd 6
buggsBunny 5
anotherTest 3
porkyPig 8
otherTest 2
someOtherTest 1
zzz 4
test 0
```

```
Identifiers level order:

test 0
someOtherTest 1
zzz 4
otherTest 2
anotherTest 3
porkyPig 8
buggsBunny 5
elmerFudd 6
daffyDuck 7
```