Laboratory 2 + 3 – LFTC

Documentation

The HashTable is an implementation of a simple hash table data structure that supports basic operations such as adding, finding, and deleting elements. It is designed to store and retrieve elements of generic type T. This implementation uses coalesced chaining to handle collisions, meaning that when a collision appears, we search for the first available position from the left.

for the HashTable we have:

```
template<class T>
struct HashEntry {
   T token;
   int next = -1;
};
```

- The HashEntry structure represents an entry in the hash table.
- It holds a generic type T representing the element (in this case, a string token).
- The next field is an index pointing to the next colliding element within the hash table.

```
template<class T>
class HashTable {
private:
   vector<HashEntry<T>> table;
   int size = 103;
   int hashFn(T element);
   int findNextPositionAvailable();
public:
   HashTable();
   HashTable(HashTable &hTable);
   int addValue(T element);
   int findPosition(T element);
   int containsValue(T element);
   int deleteValue(T element);
};
```

- The CoalescedHashTable class is a generic hash table implementation using the coalesced hashing technique.
- It includes functions for adding, finding, checking existence, and deleting elements.
- Collisions are resolved by linking colliding elements within the hash table using the next field.

---

int HashTable<T>::hashFn(T element) {}

- Computes the hash value for the given element using the sum of ASCII values of characters

---

Int HashTable<T>::findNextPositionAvailable () {}

- Finds the next available position in the hashtable

Int HashTable<T>::addValue(T element) {}

- Adds a value to the hash table, handling collisions with chaining

---

Int HashTable<T>::findPosition(T element) {}

- Finds the position of an element in the hash table

Int HashTable<T>::contrainsValue(T element) {}

- Checks if the hashtable contains a specific value

Int HashTable<T>::deleteValue(T element) {}

- Deletes a value from the hashtable

For the SymbolTable:

```
Class SymbolTable {
private:
        HashTable<string> hashTable;
Public:
        SymbolTable();
        Int findPosition(string token);
        Int deleteToken(string token);
        Int addToken(string token);
};
```

- The SymbolTable class is a specialization of the HashTable for storing and managing string tokens

---

```
SymbolTable::SymbolTable() {}
```

- Initializes a new SymbolTable instance

```
Int SymbolTable::findPosition(String token) {}
```

- Finds the position of a token in the symbol table

```
Int SymbolTable::deleteToken(string token) {}
```

- Deletes a token from the symbol table

```
Int SymbolTable::addToken(string token) {}
```

- Adds a token to the symbol table

For the Scanner:

The Scanner class, along with related functionalities, is designed to perform lexical analysis on a source code file it recognizes tokens, identifies their types, and generates a Program Internal Form (PIF). Additionally, It utilizes two different SymbolTable for storing identifiers and constants

---

```
Class Scanner {
Private:
        SymbolTable st_identifiers;
        SymbolTables st_constants;
        Vector<string> keywords;
        Vector <pair<string, int>> PIF;

        string regexIdentifiers = "[a-zA-Z]{1}[a-zA-Z0-9]*";
        string regexInt = "(0|[+-]?[1-9][0-9]*)";
        string regexChar = "\'[a-zA-Z0-9]{1}\'";
         string regexString ="\"[a-zA-Z0-9]*\"";

         bool genPIF(const string& token, int index);
         writeOutput();
        void tokensPopulate();
public:
   Scanner();
   void scanning(const string& filepath);
};
```

Private Methods:

```
void tokensPopulate();
```

- Populates the keywords vector with predefined keywords from an input file (token.in)

---

bool genPIF(const string& token, int index);

- Generates entries in the Program Internal Form(PIF) based on the recognized token and its type
- Identifies identifiers, constants (integers, characters, strings)
- Outputs error messages if a token is a lexical error
- Returns TRUE if the token is lexically correct and FALSE if the token in lexically incorrect

---

void writeOutput();

- Outputs to file ST.out the two SymbolTables for Identifiers and Constants and gives information about the data structure used in the representation
- Outputs to file PIF.out the Program Internal Form list of pairs as they were found

---

string regexIdentifiers = "[a-zA-Z]{1}[a-zA-Z0-9]*";

string regexInt = "(0|[+-]?[1-9][0-9]*)";

string regexChar = "\'[a-zA-Z0-9]{1}\'";

string regexString ="\"[a-zA-Z0-9]*\"";

- Declares regex of Identifiers and Constants

SymbolTable st_identifiers;

SymbolTables st_constants;

- The two separate symbol tables for Identifiers and Constants

---

Vector<string> keywords;

- A vector to store the keywords

Vector <pair<string, int>> PIF;

- A vector of pairs to store the Program Internal Form

Private Methods:

Scanner()

- Is the Constructor for the Scanner

Void scanning(const string & filepath);

- The scanning method initiates the lexical analysis process on a source code file specified by the filepath parameter. It recognizes various tokens, determines their types, and generates a Program Internal Form (PIF). Additionally, this method performs error checking for lexical inconsistencies.
- Paramteres:
    - const string& filepath: The file path to the source code file that needs to be analyzed
- The method traverses the source code file line by line, character by character, identifying tokens based on predefined rules.
- Tokens are classified into categories such as keywords, identifiers, constants, operators, and separators.
- The Program Internal Form (PIF) is generated, capturing the token and its corresponding position or index in the Symbol Table.
- Lexical errors, such as unidentified tokens, are detected and appropriate error messages are displayed.
- Output:
    - The results of the lexical analysis, including Symbol Tables and the Program Internal Form, are written to output files (ST.out and PIF.out).