Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Лабораторна робота №2-3

Виконав: студент групи ІО-91

Щербина О.В.

Київ 2011р.

Тема: Построение и использование объектов узлов древовидных графов. Лексический анализатор.

Листинг программы

package UnitTests;

import LexicalScaner.LexicalAnalizator;

import LexicalScaner.Token;

import junit.framework.TestCase;

import junit.framework.TestSuite;

import junit.textui.TestRunner;

import table.Record;

import table.Table;

import tree.Node;

import tree.TablesOperator;

import Automat.\*;

public class TestClass extends TestCase{

public TestClass(String testName) {

super(testName);

}

public void testTree() {

Node head=null;

String formula="f=a\*b+c-d";

String cFormula=null;

try{

TablesOperator tablesOperator=new TablesOperator();

int a=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"a");

int f=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"f");

int b=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"b");

int c=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"c");

int d=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"d");

int op1=tablesOperator.insertValue(TablesOperator.OPERATIONS,"=");

int op2=tablesOperator.insertValue(TablesOperator.OPERATIONS,"+");

int op3=tablesOperator.insertValue(TablesOperator.OPERATIONS,"\*");

int op4=tablesOperator.insertValue(TablesOperator.OPERATIONS,"-");

Node A=new Node(null,null,TablesOperator.IDENTIFIERS,a,Node.SPLIT\_CENTRE,tablesOperator);

Node B=new Node(null,null,TablesOperator.IDENTIFIERS,b,Node.SPLIT\_CENTRE,tablesOperator);

Node C=new Node(null,null,TablesOperator.IDENTIFIERS,c,Node.SPLIT\_CENTRE,tablesOperator);

Node D=new Node(null,null,TablesOperator.IDENTIFIERS,d,Node.SPLIT\_CENTRE,tablesOperator);

Node F=new Node(null,null,TablesOperator.IDENTIFIERS,f,Node.SPLIT\_CENTRE,tablesOperator);

Node OP4=new Node(C,D,TablesOperator.OPERATIONS,op4,Node.SPLIT\_CENTRE,tablesOperator);

Node OP3=new Node(A,B,TablesOperator.OPERATIONS,op3,Node.SPLIT\_CENTRE,tablesOperator);

Node OP2=new Node(OP3,OP4,TablesOperator.OPERATIONS,op2,Node.SPLIT\_CENTRE,tablesOperator);

head=new Node(F,OP2,TablesOperator.OPERATIONS,op1,Node.SPLIT\_CENTRE,tablesOperator);

cFormula=head.calculateValue();

}catch (Exception e){

e.printStackTrace();

}

assertEquals(formula,cFormula);

}

public void test1Tree() {

Node head=null;

String formula="while(i>0)i--";

String cFormula=null;

try{

TablesOperator tablesOperator=new TablesOperator();

int o1=tablesOperator.insertValue(TablesOperator.OPERATIONS,"(");

int i=tablesOperator.insertValue(TablesOperator.IDENTIFIERS,"i");

int o2=tablesOperator.insertValue(TablesOperator.CONSTANTS,"0");

int op1=tablesOperator.insertValue(TablesOperator.OPERATIONS,">");

int op2=tablesOperator.insertValue(TablesOperator.OPERATIONS,"while");

int op3=tablesOperator.insertValue(TablesOperator.OPERATIONS,"--");

int op4=tablesOperator.insertValue(TablesOperator.OPERATIONS,")");

Node A=new Node(null,null,TablesOperator.IDENTIFIERS,i,Node.SPLIT\_CENTRE,tablesOperator);

Node B=new Node(null,null,TablesOperator.CONSTANTS,o2,Node.SPLIT\_CENTRE,tablesOperator);

Node F=new Node(null,null,TablesOperator.OPERATIONS,op2,Node.SPLIT\_CENTRE,tablesOperator);

Node C=new Node(A,B,TablesOperator.OPERATIONS,op1,Node.SPLIT\_CENTRE,tablesOperator);

Node D=new Node(F,C,TablesOperator.OPERATIONS,o1,Node.SPLIT\_CENTRE,tablesOperator);

Node OP3=new Node(A,null,TablesOperator.OPERATIONS,op3,Node.SPLIT\_CENTRE,tablesOperator);

head=new Node(D,OP3,TablesOperator.OPERATIONS,op4,Node.SPLIT\_CENTRE,tablesOperator);

cFormula=head.calculateValue();

}catch (Exception e){

e.printStackTrace();

}

assertEquals(formula,cFormula);

}

public void testLoadAutomate(){

String[][] t=new String[4][4];

t[0][1]="ab";

t[1][0]="d";

t[1][3]="c";

t[2][2]="e";

t[3][3]="e";

int[] endpos=new int[2];

endpos[0]=1;

endpos[1]=2;

AutomatTable etalon=null;

AutomatTable table=null;

try{

etalon=new AutomatTable(t,0,endpos,1);

table=LoadAutomateTable.load("C:\\automats\\automat2.txt");

}catch(Exception e){

e.printStackTrace();

}

assertEquals(table,etalon);

}

public void testAutomate(){

try{

String[] tableFiles={"C:\\automats\\automat1.txt","C:\\automats\\automat3.txt"};

AutomatesManager automatesManager=new AutomatesManager(tableFiles);

String line1="i0de123ntif";

String line2="3";

int res1=automatesManager.getTokenTypes(line1);

int res2=automatesManager.getTokenTypes(line2);

System.out.println("res1="+res1+" res2="+res2);

assertEquals(res1,Token.IDENTIFIER);

assertEquals(res2,Token.CONSTANT);

}catch (Exception e){

e.printStackTrace();

}

}

public void testAutomateMethodCheck(){

try{

String[] tableFiles={"C:\\automats\\automat4.txt"};

AutomatesManager automatesManager=new AutomatesManager(tableFiles);

String line2="+0.34.5";

int res2=automatesManager.tables.get(0).check(line2);

System.out.println("result="+res2);

assertEquals(res2,Token.CONSTANT);

}catch (Exception e){

e.printStackTrace();

}

}

public LexicalAnalizator createLexicalAnalizator(){

String[]automateFiles=new String[2];

automateFiles[0]="C:\\automats\\automat1.txt";

automateFiles[1]="C:\\automats\\automat3.txt";

LexicalAnalizator lexicalAnalizator=null;

try{

AutomatesManager automatesManager=new AutomatesManager(automateFiles);

Table delimiters=new Table();

delimiters.insert(new Record(" ",null));

delimiters.insert(new Record(",",null));

delimiters.insert(new Record(".",null));

delimiters.insert(new Record("(",null));

delimiters.insert(new Record(")",null));

delimiters.insert(new Record("+",null));

delimiters.insert(new Record("-",null));

delimiters.insert(new Record("\*",null));

delimiters.insert(new Record("/",null));

delimiters.insert(new Record("<",null));

delimiters.insert(new Record(">",null));

delimiters.insert(new Record("=",null));

delimiters.insert(new Record(";",null));

delimiters.insert(new Record("{",null));

delimiters.insert(new Record("}",null));

Table terminalWords=new Table();

terminalWords.insert(new Record("if", null));

terminalWords.insert(new Record("for",null));

terminalWords.insert(new Record("while",null));

lexicalAnalizator=new LexicalAnalizator(delimiters,terminalWords,automatesManager);

}catch (Exception e){

e.printStackTrace();

}

return lexicalAnalizator;

}

public void testMethodScan(){

try{

LexicalAnalizator lexicalAnalizator=this.createLexicalAnalizator();

lexicalAnalizator.scann("if(abc<b){c=b-a;}");

String[] res={"if","(","abc","<","b",")","{","c","=","b","-","a",";","}"};

for(int i=0;i<lexicalAnalizator.parts.length;i++){

assertEquals(res[i],lexicalAnalizator.parts[i]);

}

}catch (Exception e){

e.printStackTrace();

}

}

public void testLexicalAnalizator(){

try{

LexicalAnalizator lexicalAnalizator=this.createLexicalAnalizator();

lexicalAnalizator.analize("if(abc<b){c=b-5;}");

String[] res={"if","(","abc","<","b",")","{","c","=","b","-","5",";","}"};

Token[] resT=new Token[res.length];

resT[0]=new Token(res[0],Token.TERMINALWORD,1);

resT[1]=new Token(res[1],Token.DELIMITER,1);

resT[2]=new Token(res[2],Token.IDENTIFIER,1);

resT[3]=new Token(res[3],Token.DELIMITER,1);

resT[4]=new Token(res[4],Token.IDENTIFIER,1);

resT[5]=new Token(res[5],Token.DELIMITER,1);

resT[6]=new Token(res[6],Token.DELIMITER,1);

resT[7]=new Token(res[7],Token.IDENTIFIER,1);

resT[8]=new Token(res[8],Token.DELIMITER,1);

resT[9]=new Token(res[9],Token.IDENTIFIER,1);

resT[10]=new Token(res[10],Token.DELIMITER,1);

resT[11]=new Token(res[11],Token.CONSTANT,1);

resT[12]=new Token(res[12],Token.DELIMITER,1);

resT[13]=new Token(res[13],Token.DELIMITER,1);

System.out.println("size rest="+resT.length+" lex="+lexicalAnalizator.tokens.length);

for(int i=0;i<lexicalAnalizator.tokens.length;i++){

if(lexicalAnalizator.tokens[i]!=null)

System.out.println("token i="+i+" v="+lexicalAnalizator.tokens[i].line);

}

for(int i=0;i<lexicalAnalizator.tokens.length;i++){

System.out.println("i="+i);

System.out.println("resT="+resT[i].line);

System.out.println("lexical="+lexicalAnalizator.tokens[i].line);

assertEquals(resT[i].line,lexicalAnalizator.tokens[i].line);

assertEquals(resT[i].type,lexicalAnalizator.tokens[i].type);

}

}catch (Exception e){

e.printStackTrace();

}

}

public static void main(String[] args) {

TestRunner runner = new TestRunner();

TestSuite suite = new TestSuite();

suite.addTest(new TestClass("testTree"));

suite.addTest(new TestClass("test1Tree"));

suite.addTest(new TestClass("testLoadAutomate"));

suite.addTest(new TestClass("testAutomate"));

suite.addTest(new TestClass("testMethodScan"));

suite.addTest(new TestClass("testLexicalAnalizator"));

suite.addTest(new TestClass("testAutomateMethodCheck"));

runner.doRun(suite);

}

}

package Automat;

import java.io.IOException;

import java.util.ArrayList;

public class AutomatesManager {

public ArrayList<AutomatTable> tables;

public AutomatesManager(ArrayList<AutomatTable> tables1) throws InvalidTableException{

if(tables1!=null){

tables=tables1;

}else{

throw new InvalidTableException();

}

}

public AutomatesManager(String[] fileNames) throws InvalidTableException,IncorectFileException,IOException{

if(fileNames!=null){

tables=new ArrayList<AutomatTable>();

for(int i=0;i<fileNames.length;i++){

tables.add(LoadAutomateTable.load(fileNames[i]));

}

}else {

throw new InvalidTableException();

}

}

public int getTokenTypes(String token){

int maxN=-1;

int max=-1;

int curr;

for (int i=0;i<tables.size();i++){

curr=tables.get(i).check(token);

System.out.println("curr="+curr);

if(curr>max){

max=curr;

maxN=i;

}

}

if(max>-1){

return tables.get(maxN).typeSolution;

}

return -1;

}

}

package Automat;

public class AutomatTable {

public int typeSolution;

public String[][] table;

public int startPosition;

public int[] endPositions;

public AutomatTable(String[][] table1,int startPosition1,int[] endPositions1,int typeSolution1) throws InvalidTableException{

if(table1==null || endPositions1==null){

System.out.println("table or end position null");

throw new InvalidTableException();

}

if(table1.length==0 || endPositions1.length==0){

System.out.println("table or end position size incorect");

throw new InvalidTableException();

}

if(table1.length!=table1[0].length){

System.out.println("table size incorect");

throw new InvalidTableException();

}

if(startPosition1<0 || startPosition1>table1.length){

System.out.println("start position incorect");

throw new InvalidTableException();

}

for(int i=0;i<endPositions1.length;i++){

if(endPositions1[i]<0 || endPositions1[i]>table1.length){

System.out.println("end position incorect");

throw new InvalidTableException();

}

}

typeSolution=typeSolution1;

table=table1;

startPosition=startPosition1;

endPositions=endPositions1;

}

public int check(String line){

if(line==null){

return -1;

}

if(line.length()==0){

return -1;

}

int stepCount=0;

boolean haveFind;

int currentPos=startPosition;

System.out.println("table size i="+table.length+" j="+table[0].length);

System.out.println("end pos="+endPositions[0]+" length="+endPositions.length);

for(int i=0;i<line.length();i++){

haveFind=false;

for (int j=0;j<table.length;j++){

if(table[currentPos][j]!=null){

System.out.println("Current pos="+currentPos+" symb="+line.charAt(i));

if(table[currentPos][j].indexOf(line.charAt(i))>-1){

System.out.println("have jump to "+j);

currentPos=j;

stepCount++;

haveFind=true;

break;

}

}

}

if(!haveFind){

for (int j=0;j<endPositions.length;j++){

if(endPositions[j]==currentPos){

return stepCount;

}

}

return -1;

}

}

return stepCount;

}

}

package Automat;

public class IncorectFileException extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect data int file");

super.printStackTrace();

}

}

package Automat;

public class InvalidTableException extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect table");

super.printStackTrace();

}

}

package Automat;

import java.io.BufferedReader;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStreamReader;

import java.util.ArrayList;

public class LoadAutomateTable {

public static AutomatTable load(String fileName) throws IOException,IncorectFileException{

BufferedReader in;

AutomatTable automatTable;

ArrayList<String> lines=new ArrayList<String>();

String type="";

try{

in = new BufferedReader(new InputStreamReader(new FileInputStream(fileName)));

boolean fl=false;

while (in.ready()) {

String s = in.readLine();

if(fl){

lines.add(s);

}else{

fl = true;

type=s;

}

}

in.close();

}catch (IOException e) {

throw new IOException("Cann't read file");

}

if(lines.size()<0){

throw new IncorectFileException();

}

String[][] table=new String[lines.size()][lines.size()];

for(int i=0;i<lines.size();i++){

String s=lines.get(i);

int st=0;

while (true){

int stBr=s.indexOf("(");

int stFBr=s.indexOf(" )");

String num=s.substring(st,stBr);

int n=Integer.valueOf(num);

String value=s.substring(stBr+1,stFBr);

table[i][n]=value;

s=s.substring(stFBr+2,s.length());

if(s.length()==0){

break;

}

}

}

for(int i=0;i<table.length;i++){

for(int j=0;j<table[0].length;j++){

System.out.print(table[i][j]+" ");

if(table[i][j]!=null){

System.out.print("length="+table[i][j].length());

}

}

System.out.println();

}

String paternStrart=" start position=";

String paternEnd=" end position=";

int stPos=type.indexOf(paternStrart);

int typeInt=Integer.valueOf(type.substring(0,stPos));

type=type.substring(stPos+paternStrart.length(),type.length());

int endPos=type.indexOf(paternEnd);

int startPosition=Integer.valueOf(type.substring(0, endPos));

type=type.substring(endPos+paternEnd.length(),type.length());

ArrayList<Integer> positions=new ArrayList<Integer>();

while (true){

int colPos=type.indexOf(",");

if(colPos!=-1){

positions.add(Integer.valueOf(type.substring(0,colPos)));

type=type.substring(colPos+1,type.length());

}

System.out.println("type="+type);

if(type.indexOf(",")==-1){

positions.add(Integer.valueOf(type));

break;

}

}

int[] endposVar=new int[positions.size()];

for(int i=0;i<positions.size();i++){

endposVar[i]=positions.get(i);

System.out.println("position="+endposVar[i]);

}

try{

automatTable=new AutomatTable(table,startPosition,endposVar,typeInt);

}catch (InvalidTableException e){

throw new IncorectFileException();

}

return automatTable;

}

}

package LexicalScaner;

public class IncorectAnalizeException extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect data then analize is failed");

super.printStackTrace();

}

}

package LexicalScaner;

public class IncorectTokenException extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect token");

super.printStackTrace();

}

}

package LexicalScaner;

public class InvalidLine extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect line");

super.printStackTrace();

}

}

package LexicalScaner;

import Automat.AutomatesManager;

import table.Record;

import table.Table;

import java.util.ArrayList;

import java.util.StringTokenizer;

public class LexicalAnalizator {

public AutomatesManager automatesManager;

public Table delimiters;

public String[] parts;

public Token[] tokens;

public Table identifiers;

public Table constants;

public Table terminalWords;

public LexicalAnalizator(Table delimiters1,Table terminalWords1,AutomatesManager automatesManager1) throws IncorectLexicalAnalizatorException{

if(automatesManager1==null ||delimiters1==null || terminalWords1==null){

throw new IncorectLexicalAnalizatorException();

}

automatesManager=automatesManager1;

delimiters=delimiters1;

terminalWords=terminalWords1;

identifiers=new Table();

constants=new Table();

}

public void scann(String line) throws InvalidLine{

if(line==null){

throw new InvalidLine();

}

if(line.length()==0){

throw new InvalidLine();

}

String delimSet=new String();

for(int i=0;i<delimiters.getRowCount();i++){

Record record=delimiters.select(i);

if(record!=null){

delimSet+=record.field;

}

}

StringTokenizer tokenizer=new StringTokenizer(line,delimSet,true);

ArrayList<String> tokens=new ArrayList<String>();

while (tokenizer.hasMoreTokens()){

tokens.add(tokenizer.nextToken());

}

parts=new String[tokens.size()];

for (int i=0;i<parts.length;i++){

parts[i]=tokens.get(i);

}

}

public void analize(String line) throws IncorectAnalizeException{

try{

scann(line);

}catch (InvalidLine e){

throw new IncorectAnalizeException();

}

tokens=new Token[parts.length];

try{

for(int i=0;i<parts.length;i++){

Token currentToken;

int r=delimiters.search(parts[i]);

System.out.println("token line="+parts[i]);

if(r!=-1){

System.out.println(" type=delimiter index="+r);

currentToken=new Token(parts[i],Token.DELIMITER,r);

tokens[i]=currentToken;

continue;

}

r = terminalWords.search(parts[i]);

if(r!=-1){

System.out.println(" type=terminalwords index="+r);

currentToken=new Token(parts[i],Token.TERMINALWORD,r);

tokens[i]=currentToken;

continue;

}

System.out.println(" automat start work");

r=automatesManager.getTokenTypes(parts[i]);

if(r!=-1){

if(r==Token.CONSTANT){

System.out.println(" type=constant index="+r);

int index=(int)constants.insert(new Record(parts[i],null));

currentToken=new Token(parts[i],Token.CONSTANT,index);

tokens[i]=currentToken;

continue;

}

if(r==Token.IDENTIFIER){

System.out.println(" type=identifier index="+r);

int index=(int)identifiers.insert(new Record(parts[i],null));

currentToken=new Token(parts[i],Token.IDENTIFIER,index);

tokens[i]=currentToken;

continue;

}

}else{

throw new IncorectAnalizeException();

}

}

}catch (Exception e){

throw new IncorectAnalizeException();

}

}

}

package LexicalScaner;

public class Token {

public static int CONSTANT=3;

public static int DELIMITER=1;

public static int TERMINALWORD=2;

public static int IDENTIFIER=4;

public String line;

public int type; //1-delimiter, 2-terminal word, 3- constant,4-identifier

public int index;

public Token(String line1,int type1,int index1) throws IncorectTokenException{

if(line1==null || type1<1 || type1>4 || index1<0){

throw new IncorectTokenException();

}

line=line1;

type=type1;

index=index1;

}

}

package table;

public class IncorectComparator extends Exception {

public void printStackTrace(){

System.out.println("Exception in calling method with incorect comparatorn");

super.printStackTrace();

}

}

package table;

public class IncorectEnumerationValue extends Exception{

public void printStackTrace(){

System.out.println("Exception in calling method with incorect enumration");

super.printStackTrace();

}

}

package table;

public class IncorectIndex extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect index");

super.printStackTrace();

}

}

package table;

public class IncorectPatern extends Exception{

public void printStackTrace(){

System.out.println("Exception in calling method with incorect patern");

super.printStackTrace();

}

}

package table;

public class IncorectRecordException extends Exception{

public void printStackTrace(){

System.out.println("Exception in calling method with incorect record");

super.printStackTrace();

}

}

package table;

public class MyEnumeration {

private int value;

public MyEnumeration(){

}

public int getValue(){

return value;

}

public void setValue(int value1) throws IncorectEnumerationValue{

if(value1<5 && value1>0){

value=value1;

}else {

throw new IncorectEnumerationValue();

}

}

}

package table;

public class Record {

public long key;

public String field;

public MyEnumeration enumeration;

public Record(String field1, MyEnumeration enumeration1) {

if(field1!=null){

field = field1;

enumeration = enumeration1;

}else{

throw new NullPointerException();

}

}

public void print() {

System.out.print(key + " ");

System.out.print(field);

System.out.print(" "+enumeration.getValue());

System.out.println();

}

}

package table;

public class StringComparator {

public int compare(String patern,String line){

String p=patern.toLowerCase();

String l=line.toLowerCase();

int res=0;

int size=p.length();

if(size>l.length()){

size=l.length();

}

for(int i=0;i<size;i++){

if(p.charAt(i)==l.charAt(i)){

res++;

}else {

if(p.charAt(i)==convertLanguage(l.charAt(i))){

res++;

}

}

}

return res;

}

public char convertLanguage(char in){

switch (in){

//russian->english

case 'і':return 'i';

case 'а':return 'a';

case 'х':return 'x';

case 'р':return 'p';

case 'о':return 'o';

case 'с':return 'c';

case 'у':return 'y';

//english ->russian

case 'y':return 'у';

case 'i':return 'і';

case 'o':return 'о';

case 'a':return 'а';

case 'x':return 'х';

case 'c':return 'с';

case 'p':return 'р';

default:return in;

}

}

}

package table;

import java.util.Vector;

public class Table {

private Vector<Record> fields;

private long lastId;

private int rowCount;

private boolean isSorted;

public Table() {

lastId = 0;

fields = new Vector<Record>();

isSorted = true;

}

public int getRowCount(){

return rowCount;

}

public long insert(Record record) throws IncorectRecordException {

if (record != null) {

lastId++;

record.key = lastId;

fields.add(record);

rowCount++;

isSorted = false;

return lastId;

} else {

throw new IncorectRecordException();

}

}

public void printTable() {

System.out.println("----Start table-----");

for (int i = 0; i < rowCount; i++) {

fields.get(i).print();

}

System.out.println("----End table-----");

}

public Record logicSearch(String patern, StringComparator comparator) throws IncorectComparator, IncorectPatern {

if (patern == null) {

throw new IncorectPatern();

}

if (comparator == null) {

throw new IncorectComparator();

}

int max = 0;

int pos = -1;

for (int i = 0; i < rowCount; i++) {

int c = comparator.compare(patern, fields.get(i).field);

if (c > max) {

max = c;

pos = i;

}

}

if (pos != -1) {

return fields.get(pos);

} else {

return null;

}

}

public boolean delete(long id) {

if (id > 0) {

Record r = select(id);

if (r != null) {

fields.remove(r);

} else {

return false;

}

rowCount--;

return true;

} else {

return false;

}

}

public boolean update(Record record) throws IncorectRecordException {

if (record != null) {

Record savedRecord = select(record.key);

if (savedRecord != null) {

savedRecord.field = record.field;

isSorted = false;

} else {

return false;

}

return true;

} else {

throw new IncorectRecordException();

}

}

public void sort() {

Record record;

for (int i = 0; i < rowCount - 1; i++) {

for (int j = 0; j < i; j++) {

if (fields.get(j).key > fields.get(j + 1).key) {

record = fields.get(j);

fields.remove(j);

fields.add(j, fields.get(j + 1));

fields.remove(j + 1);

fields.add(j + 1, record);

}

}

}

isSorted = true;

}

public Record select(long id) {

if (id >= 0) {

for (int i = 0; i < rowCount; i++) {

if (fields.get(i).key == id) {

return fields.get(i);

}

}

}

return null;

}

public int search(String line){

if(line==null){

return -1;

}

if(line.length()<=0){

return -1;

}

for(int i=0;i<rowCount;i++){

if(fields.get(i).field.equals(line)){

return i;

}

}

return -1;

}

public Record selectBinary(long id) {

if (id > 0) {

if (!isSorted) {

sort();

}

int curentIndex = rowCount / 2;

int delta = curentIndex;

boolean flag = false;

while (!(fields.get(curentIndex).key == id)) {

if (delta % 2 == 1) {

delta = delta / 2 + 1;

} else {

delta = delta / 2;

}

System.out.println("delta=" + delta);

System.out.println("ci=" + curentIndex);

if (fields.get(curentIndex).key < id) {

curentIndex += delta;

} else {

curentIndex -= delta;

}

if (delta < 1) {

if (!flag) {

flag = true;

delta = 1;

} else {

return null;

}

}

}

return fields.get(curentIndex);

} else {

return null;

}

}

}

package tree;

public class IncorectNodeException extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect node");

super.printStackTrace();

}

}

package tree;

public class IncorectParameters extends Exception {

public void printStackTrace(){

System.out.println("Exception Incorect parametrs");

super.printStackTrace();

}

}

package tree;

import table.IncorectIndex;

public class Node {

public Node child1;

public Node child2;

public int type;

public int index;

/\*\*

\* concatType=0 -- operation at left

\* concatType=1 -- operation at centre

\* concatType=2 -- operation at right

\*/

public static int SPLIT\_LEFT=0;

public static int SPLIT\_CENTRE=1;

public static int SPLIT\_RIGHT=2;

public int concatType;

public TablesOperator tablesOperator;

public Node(Node ch1,Node ch2,int type1,int index1,int concatType1, TablesOperator tablesOperator1) throws IncorectNodeException{

child1=ch1;

child2=ch2;

if(concatType1>3 || concatType1<0 || tablesOperator1==null || type1>3 || type1<0){

throw new IncorectNodeException();

}

type=type1;

index=index1;

concatType=concatType1;

tablesOperator=tablesOperator1;

}

public String calculateValue() throws IncorectIndex{

String ch1="";

String ch2="";

if(child1!=null){

ch1=child1.calculateValue();

}

if(child2!=null){

ch2=child2.calculateValue();

}

String res="";

String value=tablesOperator.getValue(type,index);

switch (concatType){

case 0:res=value+ch1+ch2;break;

case 1:res=ch1+value+ch2;break;

case 2:res=ch1+ch2+value;break;

}

return res;

}

}

package tree;

import table.\*;

public class TablesOperator {

/\*

type=0 - identifiers;

type=1 - constants;

type=2 - operations;

\*/

public static int IDENTIFIERS=0;

public static int CONSTANTS=1;

public static int OPERATIONS=2;

private Table identifiers;

private Table constants;

private Table operations;

public TablesOperator(){

identifiers=new Table();

constants=new Table();

operations=new Table();

}

public String getValue(int type,int index) throws IncorectIndex{

String res=null;

switch (type){

case 0: res=identifiers.select(index).field; break;

case 1: res=constants.select(index).field; break;

case 2: res=operations.select(index).field; break;

}

if(res==null){

throw new IncorectIndex();

}

return res;

}

public int insertValue(int type,String value) throws IncorectRecordException{

int index=-1;

switch (type){

case 0: index=(int)identifiers.insert(new Record(value,null)); break;

case 1: index=(int)constants.insert(new Record(value,null)); break;

case 2: index=(int)operations.insert(new Record(value,null)); break;

}

return index;

}

}