Курсова работа

Извличане на Информация от Интернет

Емануил Главчев, курс 4, СИТ, 61162106

**1. Задание**

Курсова работа по ИИИ

1. Да се създаде програмна система на С++, която да реализира опростен web crawler със следната функционалност:
   1. Получава съдържанието на уеб-страницата в изходен вид от уеб сървър.
   2. Извлича текстовото съдържание (мета-данни: заглавие, ключови думи, описание; съдържание на html-документа: игнорирайки текста в таговете за изображения) в балансирано дърво.
   3. Извлича от страницата всички линкове към други страници (.htm или .html) и създава списък с линковете.
   4. Обхожда този списък, обработвайки аналогично всяка една от страниците, като дълбочината на вложеност да не бъде повече от 10.
   5. Извежда в подходящ графичен вид граф на връзките между страниците.
   6. Генерира резултат от търсене по определена дума – извежда линковете на страниците, в които е открита (по реда на откриване, без рейтинг).
2. Да се поставят в сорса достатъчно коментари, поясняващи предназначението на отделните функции и атрибути.
3. Разработката се предава в папка, със заглавна страница, разпечатана **черно-бяла** обяснителната записка, съдържаща кратко описание на отделните класове, техните променливи и функции, разпечатка на хедърните и сорсови файлове, кратко описание на проведеното тестване- подадените входни файлове и резултатите от тестовете. В найлонов джоб на последна страница се предава CD със записаните файлове на курсовата работа.
4. Защитата на курсовата работа е персонална, с тестване с реален начален сайт.

**2. Описание на основните модули**

Уеб кроулерът се състои от следните модули:

HttpClient, HtmlParser, InvertedIndex, SiteMap, Utils.

За всеки header има и cpp (С изключение на MatchException.h). Private членовете на класовете по конвенция завършват с “\_”

**2.1 HttpClient**

Основна функционалност:

Връща текстът на HttpResponse по зададен URL. Това отговаря на функцията *char \* getResponse(char\* url);*

Отделя HTML съдържанието от responseText-а. Това отговаря на функцията *char\* trimToPage(char \*response);*

Инициализира WinSock dll-a за работа. Init();

**2.2 HtmlParser**

Парсерът парсира текст и извлича от него думите и линковете. Затова не е странно да се обобщи неговата основна функционалност с три метода.

Основна функционалност:

Парсира текст, който се подава като първи аргумент. За да обработи правилно линковете на страницата, се нуждае и от втори аргумент – URL-а на страницата. void parse(char \* page, char \* url);

Предоставя аксесори за линковете и думите които е извлякъл. И двете биват съхранявани в двоично дърво във формат: BinNode< StraightIndexValue >

**2.3** **InvertedIndex**

Целта на crawler-а е да изгради този индекс, който държи информация, кои думи в кои документи по колко пъти се срещат. Индексът се изгражда на базата на правите индекси, които формира HtmlParser-а.

**2.4 SiteMap**

SiteMap съдържа информация за връзките между страниците. При първоначална инициализация той трябва да държи поне един адрес на страница.

Отговорности:

- Държи списъкът с линкове които ще бъдат обходени от кроулера, и е отговорен за тяхното поединично предоставяне.

- Поддържа съотвествието между URL-а на документ и неговото id (цяло число).

- Държи информация за връзките между различните url-и и на базата на тази информация може да изобразява граф.

След първоначална инициализация със стартов адрес, SiteMap обработва линковете които HtmlParser-a произвежда.

**2.5 Utils**

Библиотечни класове и функции. Те включват:

**a) BinNode**

BinNode<T> е темплейтизирано двоично дърво, което поддържа търсене по ключ, добавяне и изтриване на стойности, и еднопосочна итерация на съществуващите. Дървото не поддържа автоматично балансиране.

T e стойността на всеки възел. Някои от методите имат изисквания към класът T.

За да се извършва търсене по ключ T трябва да имплементира:

- int hasKey(void\* key);

За да се използва print() функцията T също трябва да имплементира print().

**b) Lexer**

Този клас позволява обхождането на текст и извличането на информация от него чрез най-различни методи.

**c) stringUtil namespace**

Този namespace съдържа имплементации на типични string функции като compare, concat, contains и други. Той е разработен за да се демонстрира на преподавателя че съществува разбиране за това как работят тези операции.

**3. Описание на структурите от данни**

**3.1** **StraightIndexValue**

Съдържа низ и брой срещания. Използва се предимно от HtmlParser-a. Представя или линкове, или думи.

**3.2 BinNode<StraightIndexValue> links\_, index\_**

Двоичното дърво от линкове и това от думи се генерират от HtmlParser-а. Информацията колко пъти се среща линк в дадена страница не се използва в текущата реализация, но се събира. Тези две дървета от думи и линкове добиват пълно значение само в комбинация с url-а на документа от който са извлечени.

**3.3 Posting**

Съдържа брой срещания и id на документ. Добива пълно значение в комбинация с дума.

**3.4 InvertedIndexValue**

Съдържа дума, и двоично дърво от постинги (BinNode<Posting>).

**3.5 InvertedIndex**

Съдържа двоично дърво от InvertedIndexValues. Тоест: Дърво от думи, които сочат към дървета от постинги (<documentId, numberOfOccurences>).

**4. Алгоритъм на изпълнение**

А) Зарежда се WinSock.dll-a чрез httpClient::init();

B) SiteMap и InvertedIndex ще бъдат създадени само веднъж. SiteMap се инициализира със стартовия адрес: „crawlertest.cs.tu-varna.bg“

C) От sitemap се взима следващата необходена страница и се подава на нов HttpClient. Последният праща заявка към сървъра за да получи текста на response-а, след което от този response се извлича текста на Html страницата.

D) Този текст се предава на нов HtmlParser, който извлича двоични дървета с думите и линковете. Думите се предават на InvertedIndex-а който ги записва в своите структури, а линковете се предават на SiteMap-а.

E) SiteMap-ът ги обхожда и обновява своята информация за връзките между страниците, и необходените страници. Той също добавя нова информация в регистъра за съответствието между id и url на документите.

F) Ако sitemap има още линкове за обхождане повтаряме от C).

G) Вече всички страници са обходени.

H) Предлагаме на потребителя 3 възможности. I) J) или K)

I) SiteMap принтира графът, като изобразява списъците на съседство. Връщаме се към H).

J) Търсене на документи по дума. Потребителят въвежда дума, а ние извличаме съответния списък от постинги от InvertedIndex, и го сортираме преди извеждане, за да поставим най-релевантните резултати на първо място. Връщаме се към H).

K) Излизаме от програмата

**5. Листинг**

Можете да го видите в електронен вариант на следния адрес:

https://github.com/perushinkov/web-crawler-example

**5.1.** **Headers (Тук са поставени повечето коментари):**

HttpClient.h

#ifndef \_\_HTTPCLIENT\_H

#define \_\_HTTPCLIENT\_H 1

#include <iostream>

using namespace std;

#include <winsock2.h>

#include "../utils/StringUtil.h"

#define RESPONSE\_MAX\_LENGTH 40000

namespace httpClient {

/\*

These functions are used to extract the different parts of

the url.

URL = <host>/<URI>

USAGE:

dnslookup requires just the host part of the url

HTTP request formation requires just uri

\*/

char\* getHostFromUrl(char\* url);

char\* getUriFromUrl(char\* url);

/\* Called just once for the initialization of the WinSock dll \*/

void init();

/\* Performs an http request and receives a response from server.

Represents the main delay in the execution of the program.

\*/

char \* getResponse(char \* url);

/\* Uses gethostbyname() to extract ip address \*/

char\* getIpByHost(char \*host\_name);

/\* Trims http response and leaves just html content \*/

char\* trimToPage(char \*response);

};

#endif

HtmlParser.h

#ifndef \_\_HTMLPARSER\_H

#define \_\_HTMLPARSER\_H 1

#include "..\utils\BinNode.h"

#include "..\utils\Lexer.h"

#include "StraightIndexValue.h"

/\*

\* XHTML parser.

\* Note: it currently doesn't support some misformed tags that

\* HTML supports. (missing closing tags, etc)

\*/

class HtmlParser {

private:

// Used to identify different NonTerminals in grammar

enum rules\_ {ANYTAG, ATTRIBUTE, CHAR, CHARDATA, NAME,

NAMECHAR, REFERENCE, S\_enum, SYSTEMLITERAL, WORD};

// Two binary trees that hold words and links respectively

BinNode<StraightIndexValue> \* index\_;

BinNode<StraightIndexValue> \* links\_;

// A helper that holds text and walks it via helpful methods

Lexer \* lexer\_;

/\*

This is the url of the current page minus the filename of

the page. It is used to form relative links.

\*/

char\* urlBase\_;

/\* These private variables are used to keep state as the parser

parses the text. More specifically they hold the attribute name

and the attribute value of the currently parsed attribute. Used

for analysis of metadata and links.

\*/

char\* attrName\_;

char\* attrValue\_;

/\*

A function that updates a given index with an entry that is

either a link or a word. If an entry already exists in the index,

its occurences are incremented, else a new entry is created.

\*/

void updateIndex(BinNode<StraightIndexValue> \* anyIndex, char \* word);

/\* Takes a look at the following symbol (or symbols where needed),

and returns true if is unambiguously certain that the given rule

is the one that follows .

\*/

bool follows(rules\_ rule);

/\* It proccesses links:

1. Removes variables in url

2. Makes sure only .html and .htm urls are processed.

3. Handles both absolute and relative urls.

If the result is not null, then it is a new crawlable url.

\*/

char\* processLink(char\* link);

// Used to form urlBase\_. See urlBase\_ description.

char\* getBaseFromUrl(char\* url);

public:

HtmlParser();

//Getters for results

BinNode<StraightIndexValue> \* getWords();

BinNode<StraightIndexValue> \* getLinks();

/\* These parse functions represent a highly simplified parser,

based on the XHTML EBNF grammar.

CONTENT: (anyTag | reference | charData)\*

ANYTAG: "<" ["/"] (attribute)\* ["/"] ">"

ATTRIBUTE: Name "=" "\"" SystemLiteral "\""

NAME: (Letter | '\_' | ':') (NameChar)\*

SYSTEM\_LITERAL: ('"' [^"]\* '"') | ("'" [^']\* "'")

REFERENCE: ('&' Name | '&#' [0-9]+ | '&#x' [0-9a-fA-F]+) ';'

CHARDATA: ([^<a-zA-Z\_]\* | WORD )\*

\*/

void parse(char \* page, char \* url);

void Attribute();

void anyTag();

void CharData();

void content();

char \* Name();

void Reference();

char \* SystemLiteral();

};

#endif

StraightIndexValue.h

#ifndef \_\_STRAIGHTINDEXVALUE\_H

#define \_\_STRAIGHTINDEXVALUE\_H 1

/\*

This is a class that holds a string and the number of its occurences.

It implements all it needs to become binary tree content.

USAGE:

It is the value by which the number of occurences of a link or a word are stored.

\*/

class StraightIndexValue {

private:

char \* word\_;

int occurences\_;

public:

StraightIndexValue(char \* word);

void increment();

/\*

BinNode compatibility

\*/

int hasKey(void \* ptr);

int equals(void \* object);

void print();

//Getters and setters

char \* getWord();

int getOccurences();

};

#endif

InvertedIndex.h

#ifndef \_\_INVERTEDINDEX\_H

#define \_\_INVERTEDINDEX\_H 1

#include "../utils/BinNode.h"

#include "../htmlparser/StraightIndexValue.h"

#include "InvertedIndexValue.h"

/\*

This class holds the words that have been found.

Each InvertedIndexValue has a word key, and a tree of postings.

Each Posting has a docId and number of occurences.

\*/

class InvertedIndex {

private:

BinNode<InvertedIndexValue>\* wordIndex\_;

public:

InvertedIndex();

/\*

This method is used to update the inverted index with straight index information.

\*/

void updateIndex(BinNode<StraightIndexValue> \* links\_, int docId);

void printIndex();

/\*

Returns a pointer to the InvertedIndexValue that represents the search results

for this word.

\*/

InvertedIndexValue\* searchWord(char \* word);

};

#endif

InvertedIndexValue.h

#ifndef \_\_INVERTEDINDEXVALUE\_H

#define \_\_INVERTEDINDEXVALUE\_H 1

#include "Posting.h"

#include "../utils/BinNode.h"

/\*

A word and a tree of Postings.

Each Posting holds a document and a number of occurences.

\*/

class InvertedIndexValue {

private:

char \* word\_;

BinNode<Posting> \* posts\_;

public:

InvertedIndexValue(char \* word);

void addOccurence(int docId, int howMany);

int hasKey(void \* ptr);

int equals(void \* object);

void print();

char \* getWord();

BinNode<Posting> \* getPostings();

};

#endif

Posting.h

#ifndef \_\_POSTING\_H

#define \_\_POSTING\_H 1

#include "../sitemap/SiteMap.h"

/\*

Holds document id, and number of occurences in this document.

\*/

class Posting {

private:

int docId\_;

int occurences\_;

public:

Posting(int docId);

void increment(int howMany);

int hasKey(void \* ptr);

int equals(void \* object);

void print();

//Getters and Setters

int getDocId();

int getOccurences();

};

#endif

SiteMap.h

#ifndef \_\_SITEMAP\_H

#define \_\_SITEMAP\_H 1

#include "../utils/BinNode.h"

#include "../htmlparser/StraightIndexValue.h"

#include <vector>

#include <map>

using namespace std;

/\*

SiteMap has several responsibilities:

- it holds the links that will be crawled by the crawler and is responsible

for delivering them one by one.

- it maintains mapping between docids and docUrls

- It holds information about the connnections between the different urls

and can draw a graph.

\*/

class SiteMap {

private:

// Current linkToBeCrawled

int current\_;

//New linksToBeCrawled are added here

vector<string> linksToBeCrawled\_;

//URL to ID and ID to URL maps

map<string, int> urlIds\_;

map<int, string> idUrls\_;

// Holds graph information. Works just with docIds

map<int, vector<int>> urlGraph\_;

public:

SiteMap(char \* startingAddress);

// Delivers next url to be crawled

char\* getNextUrl();

// Tells if there are still urls to be crawled

bool hasNextUrl();

// Interface to private Id to URL map

string getUrlById(int id);

// Interface to private URL to Id maps

int getDocId(char \* url);

/\*

Takes a straight index tree and the url of the page and

updates sitemap state that includes all of the above

(Links to be crawled, <docId, docUrl> maps, graph).

\*/

void updateMap(BinNode<StraightIndexValue> \* links\_, char\* parentUrl);

/\*

Uses a layered approach to paint a "graph" representation on

the console. The representation shows which page points to

which pages. Neighbour lists.

\*/

void printPrettyPicture();

};

#endif

BinNode.h

#ifndef \_\_BINNODE\_H

#define \_\_BINNODE\_H 1

/\* The binary tree which this project uses.

It supports basic search, add, delete and iteration functionality.

Some of its methods have requirements of T.

For instance T must implement

bool hasKey(void\* key)

to use findByKey(void\* key);

and print() to use the BinNode print() method;

\*/

template <class T>

class BinNode {

private:

BinNode<T> \* parent\_;

BinNode<T> \* leftChild\_;

BinNode<T> \* rightChild\_;

T \* content\_;

public:

BinNode() {

parent\_ = nullptr;

content\_ = nullptr;

leftChild\_ = nullptr;

rightChild\_ = nullptr;

}

BinNode(BinNode \* parent, T \* content) {

parent\_ = parent;

content\_ = content;

leftChild\_ = nullptr;

rightChild\_ = nullptr;

}

/\*\*

\* T must have a method hasKey(void \* key)

\*/

T \* findByKey(void \* key) {

if (content\_ == nullptr) {

return nullptr;

}

int comparison = content\_->hasKey(key);

if (comparison == 0) {

return content\_;

}

if (comparison > 0 && rightChild\_ != nullptr) {

return rightChild\_->findByKey(key);

}

if(comparison < 0 && leftChild\_ != nullptr) {

return leftChild\_->findByKey(key);

}

return nullptr;

}

/\*\*

\* Returns true if no content was overridden

\*/

bool add(T \* entry) {

if (content\_ == nullptr) {

content\_ = entry;

return true;

}

int comparison = content\_->equals((void \*)entry);

if (comparison == 0) {

content\_ = entry;

return false;

}

if (comparison > 0) {

if (rightChild\_ != nullptr) {

return rightChild\_->add(entry);

}

rightChild\_ = new BinNode<T>(this, entry);

}

if(comparison < 0) {

if (leftChild\_ != nullptr) {

return leftChild\_->add(entry);

}

leftChild\_ = new BinNode<T>(this, entry);

}

return true;

}

/\*

Some memory management

\*/

void deleteAll() {

delete content\_;

if (leftChild\_ != nullptr) {

leftChild\_->deleteAll();

}

if (rightChild\_ != nullptr) {

rightChild\_->deleteAll();

}

}

/\*

Printer

\*/

void print() {

if (leftChild\_ != nullptr) {

leftChild\_->print();

}

if (content\_ != nullptr)

content\_->print();

if (rightChild\_ != nullptr) {

rightChild\_->print();

}

}

/\*

Forward Iterator functionality

\*/

BinNode<T>\* next() {

if (leftChild\_ != nullptr) return leftChild\_;

if (rightChild\_ != nullptr) return rightChild\_;

BinNode<T>\* lastOne = this;

while (lastOne->parent\_ != nullptr) {

if (lastOne->parent\_->leftChild\_ == lastOne &&

lastOne->parent\_->rightChild\_ != nullptr) {

return lastOne->parent\_->rightChild\_;

}

lastOne = lastOne->parent\_;

continue;

}

return nullptr;

}

T\* getContent() {

return content\_;

}

};

#endif

Lexer.h

#ifndef \_\_LEXER\_H

#define \_\_LEXER\_H 1

/\* A utility class that is initialized with a text, which it then

walks via its helpful methods.

\*/

class Lexer{

private:

char \* text\_;

int pos\_;

char \* rem\_;

public:

/\* Reads a char. Exceptionn if end of text. \*/

char nextChar();

static enum lexemeType {SP, HTTPVERSION};

Lexer(char \* text);

/\* Reads a line from the text and returns it \*/

char \* getLine();

/\* Matches text. This means that it reads this text from the current position.

If this teext does not follow, an exception is thrown.

\*/

void match(char \* txt);

/\* Matches until it meets a certain text. Note that it does not read this text. \*/

int matchUntil(char \* txt);

/\* Returns true if parameter text follows. \*/

bool isNext(char \* txt);

/\* Reads and returns remaining text. \*/

char \* getRemainingText();

/\* Reads a word. Words are reduced to lowercase.

IF no word follows, returns nullptr.

\*/

char \* matchWord();

/\* Finds next word and returns it or returns nullptr. \*/

char \* findWord();

/\* Returns a character ahead of current position,

without moving the current position.

\*/

char lookahead(int howmuch);

/\* Fetches last n characters from current position. Does not move read pointer. \*/

char \* fetchLastNChars(int n);

};

#endif

MatchException.h

#ifndef \_\_MATCHEXCEPTION\_H

#define \_\_MATCHEXCEPTION\_H 1

#include <exception>

class MatchException: public std::exception {};

#endif

StringUtil.h

#ifndef \_\_STRINGUTIL\_H

#define \_\_STRINGUTIL\_H 1

#include <string.h>

namespace stringUtil {

/\*

compares a to b. Works just as strcmp does.

\*/

int compare(char \* a, char \* b);

/\*

Concatenates a to b and returns the new char\*.

Does not change the parameters.

\*/

char \* concat(const char \* a, const char \* b);

/\*

Tells if a string contains a char

\*/

bool contains(char \* subject, char containee);

/\*

A copy that copies from first param to second a certain

amount of chars.

\*/

void copy(const char \* from, char \* to, int numberOfChars);

/\*

Returns position of string a in string b, or -1.

\*/

int findAinB(char \* a, char \* b);

/\*

Equivalent of strlen

\*/

int length(const char \* a);

/\*

Equivalent of substring

\*/

char \* substring(char \* text, int len);

/\*

Changes param to lowercase

\*/

void toLower(char \* text);

}

#endif

HttpClient.cpp

#include "HttpClient.h"

#include <Winsock2.h>

#include "../utils/StringUtil.h"

char\* httpClient::getHostFromUrl(char\* url) {

int pos = stringUtil::findAinB("/", url);

if (pos == -1) {

return url;

}

return stringUtil::substring(url, pos);

}

char\* httpClient::getUriFromUrl(char\* url) {

int pos = stringUtil::findAinB("/", url);

if (pos == -1) {

return "/";

}

int len = stringUtil::length(url);

return stringUtil::substring(url + pos, len - pos);

}

char \* httpClient::getResponse(char \* url) {

char\* ip = getIpByHost(getHostFromUrl(url));

if (ip == nullptr) return nullptr;

int sock\_ = socket(AF\_INET, SOCK\_STREAM, 0);

int \*p\_int = (int\*)malloc(sizeof(int));

\*p\_int = 1;

int opt1 = setsockopt(sock\_, SOL\_SOCKET, SO\_REUSEADDR,

(char \*)p\_int, sizeof(int));

int opt2 = setsockopt(sock\_, SOL\_SOCKET, SO\_KEEPALIVE,

(char \*)p\_int, sizeof(int));

if (opt1 == -1 || opt2 == -1)

exit(2);

struct sockaddr\_in my\_addr;

my\_addr.sin\_family = AF\_INET;

my\_addr.sin\_port = htons(80);

memset(&(my\_addr.sin\_zero), 0, 8);

my\_addr.sin\_addr.s\_addr = inet\_addr(ip);

if (connect(sock\_, (struct sockaddr\*)&my\_addr, sizeof(my\_addr)) != 0){

exit(4);

}

char\* query = (char \*)malloc(150);

query[0] = '\0';

query = stringUtil::concat(query, "GET ");

query = stringUtil::concat(query, getUriFromUrl(url));

query = stringUtil::concat(query, " HTTP/1.1\nHost:");

query = stringUtil::concat(query, ip);

query = stringUtil::concat(query, "\n\n\0");

send(sock\_, query, stringUtil::length(query), 0);

char \* text = (char \*)malloc(RESPONSE\_MAX\_LENGTH);

// A huge container for the response

int received = 0;

int newAmount = 0;

received += recv(sock\_, text + received, 100, 0);

if (text[9] != '2') //Checking status code

return nullptr;

do {

newAmount = recv(sock\_, text+received, 7000, 0);

received += newAmount;

} while (newAmount > 0);

// A normal container for the exact size response

char \* returnText = (char \*)malloc(received + 1);

// Copy the huge response into it

\*(text + received\*sizeof(char)) = '\0';

stringUtil::copy(text, returnText, stringUtil::length(text));

int a = strlen(text);

int b = strlen(returnText);

free(text);

return returnText;

}

void httpClient::init() {

unsigned short wVersionRequested;

WSADATA wsaData;

int err;

// PART ONE: Initiating use of Winsock DLL

// Winsock may have already been initialized, if

// a call to getIpByHost has already been made.

wVersionRequested = MAKEWORD(2, 2);

err = WSAStartup(wVersionRequested, &wsaData);

if (err != 0 || (LOBYTE(wsaData.wVersion) != 2 ||

HIBYTE(wsaData.wVersion) != 2)) {

exit(1);

}

}

char\* httpClient::trimToPage(char\* response) {

// Moving pointer to start of html body

response += stringUtil::findAinB("<html", response);

return response;

}

char\* httpClient::getIpByHost(char \*host\_name) {

int iResult;

struct hostent \*remoteHost;

remoteHost = gethostbyname(host\_name);

if (remoteHost == nullptr) {

return 0;

}

if (remoteHost->h\_addrtype == AF\_INET &&

remoteHost->h\_addr\_list[0] != 0) {

return inet\_ntoa(\*(struct in\_addr \*)\*remoteHost->h\_addr\_list);

}

return 0;

}

HtmlParser.cpp

#include "HtmlParser.h"

#include "../utils/StringUtil.h"

#include "../utils/MatchException.h"

#include <iostream>

using namespace std;

/\*\*

\* Updates index with new word.

\*

\* If wordIndex exists, updates it,

\* else creates it.

\*/

void HtmlParser::updateIndex(BinNode<StraightIndexValue> \* targetIndex,

char \* word) {

StraightIndexValue \* wordIndex;

wordIndex = targetIndex->findByKey((void \*)word);

if (wordIndex != nullptr) {

wordIndex->increment();

} else targetIndex->add(new StraightIndexValue(word));

}

char\* HtmlParser::processLink(char\* link) {

//REMOVE VARIABLES

int questionMarkLocation = stringUtil::findAinB("?", link);

if (questionMarkLocation != -1)

link = stringUtil::substring(link, questionMarkLocation);

int len = stringUtil::length(link);

// IF ABSOLUTE URL

if (stringUtil::findAinB("http://", link) == 0 &&

(stringUtil::findAinB(".htm", link) == len - 4 ||

stringUtil::findAinB(".html", link) == len - 5)) {

return link + 7;

}

// IF RELATIVE PATH TO A FILE

if ( (stringUtil::findAinB(".htm", link) == len - 4 ||

stringUtil::findAinB(".html", link) == len - 5) &&

stringUtil::findAinB("http://", link) == -1) {

return stringUtil::concat(urlBase\_, link);

}

// No other hrefs supported!

return nullptr;

}

char\* HtmlParser::getBaseFromUrl(char\* url) {

if (stringUtil::findAinB("/", url) == -1) {

return stringUtil::concat(url, "/");

}

else {

for (int i = stringUtil::length(url) - 1; i > 0; i--) {

if (\*(url + i) == '/') {

return stringUtil::substring(url, i + 1);

}

}

}

throw new exception("Base url not formed correctly!");

return nullptr;

}

//PUBLIC

HtmlParser::HtmlParser() {

lexer\_ = nullptr;

index\_ = nullptr;

links\_ = nullptr;

}

BinNode<StraightIndexValue> \* HtmlParser::getWords() { return index\_; }

BinNode<StraightIndexValue> \* HtmlParser::getLinks() { return links\_; }

void HtmlParser::parse(char\* t, char\* url) {

urlBase\_ = getBaseFromUrl(url);

if (index\_ != nullptr) {

delete index\_;

}

index\_ = new BinNode<StraightIndexValue>(nullptr, nullptr);

if (links\_ != nullptr)

delete links\_;

links\_ = new BinNode<StraightIndexValue>(nullptr,

new StraightIndexValue("\_"));

lexer\_ = new Lexer(t);

content();

}

void HtmlParser::Attribute() {

bool isHref = false;

attrName\_ = Name();

while (lexer\_->lookahead(0) != '=') {

lexer\_->nextChar();

}

lexer\_->nextChar();

while (lexer\_->lookahead(0) != '"' && lexer\_->lookahead(0) != '\'') {

lexer\_->nextChar();

}

attrValue\_ = SystemLiteral();

}

void HtmlParser::anyTag() {

lexer\_->match("<");

if (lexer\_->isNext("!--")) {

lexer\_->matchUntil("-->");

lexer\_->match("-->");

return;

}

if (lexer\_->isNext("/")) lexer\_->nextChar();

char\* tagName = Name();

char \* metaName = nullptr; //Used only for metas

char \* metaContent = nullptr;

while (follows(S\_enum)) {

lexer\_->nextChar();

if (follows(ATTRIBUTE)) {

Attribute();

}

if (!stringUtil::compare(attrName\_, "href")) {

char\* mylink = processLink(attrValue\_);

if (mylink != nullptr) {

updateIndex(links\_, mylink);

}

}

if (!stringUtil::compare(tagName, "meta")) {

if (!stringUtil::compare(attrName\_, "name")) {

metaName = attrValue\_;

}

if (!stringUtil::compare(attrName\_, "content")) {

metaContent = attrValue\_;

}

}

}

if (metaName != nullptr &&

(!stringUtil::compare(metaName, "description") ||

!stringUtil::compare(metaName, "keywords"))) {

Lexer lexer(attrValue\_);

char\* word = lexer.findWord();

while (word != nullptr) {

stringUtil::toLower(word);

updateIndex(index\_, word);

word = lexer.findWord();

}

}

if (lexer\_->isNext("/")) lexer\_->nextChar();

if (lexer\_->matchUntil(">") != -1) {

lexer\_->match(">");

}

else {

lexer\_->getRemainingText();

}

}

void HtmlParser::CharData() {

while (lexer\_->lookahead(0) != '<') {

if (follows(WORD)) {

//Here is where word extraction takes place

updateIndex(index\_, lexer\_->matchWord());

} else {

if (lexer\_->lookahead(0) == '\0') return;

lexer\_->nextChar();

}

}

}

//content ::= (anyTag | CharData | Reference)\*

void HtmlParser::content() {

while(true) {

if (lexer\_->lookahead(0) == '\0') break;

if (lexer\_->lookahead(0) == '<'){

anyTag();

} else if (follows(REFERENCE)) {

Reference();

} else {

CharData();

}

}

}

//Name ::= (Letter | '\_' | ':') (NameChar)\*

char \* HtmlParser::Name() {

int i = 0;

lexer\_->nextChar();

i++;

while (follows(NAMECHAR)) {

lexer\_->nextChar();

i++;

}

return lexer\_->fetchLastNChars(i);

}

//Reference ::= ('&' Name | '&#' [0-9]+ | '&#x' [0-9a-fA-F]+) ';'

void HtmlParser::Reference() {

lexer\_->match("&");

if (follows(NAME)) {

Name();

} else {

lexer\_->match("#");

if (lexer\_->lookahead(0) == 'x') {

lexer\_->match("x");

}

lexer\_->matchUntil(";");

}

lexer\_->match(";");

}

// SystemLiteral ::= ('"' [^"]\* '"') | ("'" [^']\* "'")

char \* HtmlParser::SystemLiteral() {

char \* retVal;

char current = lexer\_->lookahead(0);

if (current == '"') {

lexer\_->nextChar();

retVal = lexer\_->fetchLastNChars(lexer\_->matchUntil("\""));

lexer\_->match("\"");

} else if (current == '\'') {

lexer\_->nextChar();

retVal = lexer\_->fetchLastNChars(lexer\_->matchUntil("'"));

lexer\_->match("'");

} else {

throw new MatchException();

}

return retVal;

}

/\*\*

\* Takes a look at the following symbol (or symbols where needed),

\* and returns true if is unambiguously certain that the given rule

\* is the one that follows .

\*/

bool HtmlParser::follows(rules\_ rule) {

char val = lexer\_->lookahead(0);

switch(rule) {

case ATTRIBUTE: //[\_:A-Za-z]

return follows(NAME);

break;

case ANYTAG: // <[\_:A-Za-z/]

{

char val2 = lexer\_->lookahead(1);

if (val == '<'

&& (val2 == '\_'

|| val2 == ':' || val2 == '/'

|| (val2 >= 'A' && val2 <= 'Z')

|| (val2 >= 'a' && val2 <= 'z'))) {

return true;

}

break;

}

case NAME: //[\_:A-Za-z]

if (stringUtil::contains("\_:", val)

|| (val>='A' && val <='Z')

|| (val>='a' && val <='z') ){

return true;

}

break;

case NAMECHAR: // [.-\_:A-Za-z0-9]

if (stringUtil::contains(".-\_:", val)

|| (val>='A' && val <='Z')

|| (val>='a' && val <='z')

|| (val>='0' && val <='9') ) {

return true;

}

break;

case REFERENCE: // [&]

if (val == '&') {

return true;

}

break;

case S\_enum: // [\r\n\t ]

if (stringUtil::contains("\r\n\t ", val)) {

return true;

}

break;

case SYSTEMLITERAL: // ["']

if (stringUtil::contains("\"'", val)) {

return true;

}

break;

case WORD: // [A-Za-z]

if ((val >= 'a' && val <= 'z')

|| (val >= 'A' && val <= 'Z')) {

return true;

}

}

return false;

}

StraightIndexValue.cpp

#include "StraightIndexValue.h"

#include "string.h"

#include <iostream>

using namespace std;

StraightIndexValue::StraightIndexValue(char \* word) {

word\_ = word;

occurences\_ = 1;

}

void StraightIndexValue::increment() {

occurences\_++;

}

int StraightIndexValue::hasKey(void \* ptr) {

int comparison = strcmp(word\_, (char\*)ptr);

if (comparison == 0) {

return 0;

}

if (comparison > 0) {

return 1;

}

return -1;

}

/\*\*

\* Two SIVs are considered equal if their words are equal.

\* That is so that they are sorted by words in the binary tree.

\*/

int StraightIndexValue::equals(void \* object) {

char \* objectWord = ((StraightIndexValue\*)object)->getWord();

return this->hasKey((void \*) objectWord);

return 0;

}

void StraightIndexValue::print() {

printf("Word %s occurs %d times\n", word\_, occurences\_);

this->occurences\_;

this->word\_;

}

char \* StraightIndexValue::getWord() { return word\_; }

int StraightIndexValue::getOccurences() { return occurences\_; }

InvertedIndex.cpp

#include "InvertedIndex.h"

InvertedIndex::InvertedIndex() {

wordIndex\_ = new BinNode<InvertedIndexValue>(nullptr,

new InvertedIndexValue("\_"));

}

void InvertedIndex::updateIndex(BinNode<StraightIndexValue> \* words,

int docId) {

do {

StraightIndexValue\* docWithWords = words->getContent();

if (docWithWords == nullptr) {

words = words->next();

continue;

}

char\* word = docWithWords->getWord();

int occurence = docWithWords->getOccurences();

InvertedIndexValue\* wordWithPostings =

wordIndex\_->findByKey((void\*)word);

if (wordWithPostings != nullptr) {

wordWithPostings->addOccurence(docId, occurence);

}

else {

wordWithPostings = new InvertedIndexValue(word);

wordWithPostings->addOccurence(docId, occurence);

wordIndex\_->add(wordWithPostings);

}

words = words->next();

} while (words != nullptr);

}

InvertedIndexValue\* InvertedIndex::searchWord(char \* word) {

return wordIndex\_->findByKey((void \*)word);

}

void InvertedIndex::printIndex() {

wordIndex\_->print();

}

InvertedIndexValue.cpp

#include "InvertedIndexValue.h"

#include "../utils/StringUtil.h"

#include <iostream>

using namespace std;

int InvertedIndexValue::hasKey(void \* key) {

return strcmp(word\_, (char \*)key);

}

int InvertedIndexValue::equals(void \* object) {

return strcmp(word\_, ((InvertedIndexValue \*)object)->getWord());

}

void InvertedIndexValue::print() {

printf("Word: %s \n", word\_);

posts\_->print();

}

InvertedIndexValue::InvertedIndexValue(char \* word) {

word\_ = word;

posts\_ = new BinNode<Posting>(nullptr, nullptr);

}

void InvertedIndexValue::addOccurence(int docId, int howMany) {

Posting\* posting = posts\_->findByKey((void \*)&docId);

if (posting == nullptr) {

posting = new Posting(docId);

posts\_->add(posting);

}

posting->increment(howMany);

}

char \* InvertedIndexValue::getWord() {

return word\_;

}

BinNode<Posting> \* InvertedIndexValue::getPostings() {

return posts\_;

}

Posting.cpp

#include "Posting.h"

#include <iostream>

using namespace std;

Posting::Posting(int docId) {

docId\_ = docId;

occurences\_ = 0;

}

void Posting::increment(int howMany) {

occurences\_ += howMany;

}

int Posting::hasKey(void \* ptr) {

if (docId\_ == \*((int\*)ptr)) {

return 0;

}

if (docId\_ > \*((int\*)ptr)) {

return 1;

}

return -1;

}

/\*\*

\* Two postings are considered equal if their docIds are equal.

\* That is so that they are sorted by docIds in binary tree.

\*/

int Posting::equals(void \* object) {

int objectDocId = ((Posting\*)object)->getDocId();

void \* p\_int = &objectDocId;

return hasKey(p\_int);

return 0;

}

void Posting::print() {

printf("%d\t - \t%s\n", occurences\_, docId\_);

}

int Posting::getDocId() {

return docId\_;

}

int Posting::getOccurences() {

return occurences\_;

}

SiteMap.cpp

#include "SiteMap.h"

#include <algorithm>

#include "../utils/StringUtil.h"

#include <string>

using namespace std;

SiteMap::SiteMap(char \* startingAddress) {

linksToBeCrawled\_.push\_back(startingAddress);

current\_ = 0;

urlIds\_[startingAddress] = 0;

idUrls\_[0] = startingAddress;

}

string SiteMap::getUrlById(int id) {

if (idUrls\_.find(id) != idUrls\_.end()) {

return idUrls\_[id];

}

return "";

}

char\* SiteMap::getNextUrl() {

if (current\_ >= linksToBeCrawled\_.size())

return nullptr;

string nextUrl = linksToBeCrawled\_.at(current\_);

current\_++;

return stringUtil::concat(nextUrl.c\_str(), "");

}

bool SiteMap::hasNextUrl() {

if (current\_ >= linksToBeCrawled\_.size())

return false;

return true;

}

void SiteMap::updateMap(BinNode<StraightIndexValue> \* links, char\* parentUrl) {

static int id = 1;

vector<int> outcoming;

do {

StraightIndexValue\* link = links->getContent();

char\* url = link->getWord();

if (!stringUtil::compare(url, "\_")) {

links = links->next();

continue;

}

if (urlIds\_.find(url) == urlIds\_.end()) {

urlIds\_[url] = id;

idUrls\_[id] = url;

id++;

linksToBeCrawled\_.push\_back(url);

}

outcoming.push\_back(urlIds\_[url]);

links = links->next();

} while (links != nullptr);

urlGraph\_[urlIds\_[parentUrl]] = outcoming;

}

int SiteMap::getDocId(char \* url) {

return urlIds\_[url];

return 0;

}

void SiteMap::printPrettyPicture() {

system("cls");

for (int x = 0; x < 2 + 2 \* urlIds\_.size(); x++) {

for (int y = 0; y < 2 + 3 \* urlIds\_.size(); y++) {

// UI Filters applied here

if (x == 1 && y % 3 == 0 && y > 1) printf("%c", 186);

else if (x == 1 || y == 1) printf("%c", 178);

else if (x == 0 && y > 1 && y % 3 == 0) printf("%c", (y-3)/3 + 'A');

else if (y == 0 && x > 1 && x % 2 == 0) printf("%c", (x-2)/2 + 'A');

else if (x == 0 && y > 1 && (y+1) % 3 == 0) printf("%c", '/');

else if (x == 0 && y > 1 && (y-1) % 3 == 0) printf("%c", '\\');

else if (y % 3 == 0 && y > 1 &&

urlGraph\_.find((y-3)/3) != urlGraph\_.end() &&

urlGraph\_[(y-3)/3].size() != 0) {

int sourceId = (y-3)/3;

vector<int> targets = urlGraph\_[sourceId];

bool targetHere = false;

int lastDest = \*std::max\_element( std::begin(targets),

std::end(targets);

int lastDestX = 2 + 2 \* lastDest;

x % 2 == 0 && find(targets.begin(), targets.end(),

(x - 2) / 2) != targets.end()) {

targetHere = true;

}

if (x < lastDestX) {

if (targetHere) std::printf("%c", 185);

else std::printf("%c", 186);

} else if (x == lastDestX) std::printf("%c", 188);

else std::printf("%c", ' ');

} else if (y % 3 == 2 && y > 1 &&

urlGraph\_.find((y-2)/3) != urlGraph\_.end() &&

urlGraph\_[(y - 2) / 3].size() != 0) {

vector<int> targets = urlGraph\_[(y - 2) / 3];

if (x % 2 == 0 && find(targets.begin(), targets.end(), (x-2)/2)

!= targets.end()) std::printf("%c", 205);

else std::printf(" ");

} else if (y%3==1 && y>1 &&

urlGraph\_.find((y-1)/3) != urlGraph\_.end() &&

urlGraph\_[(y - 1) / 3].size() != 0) {

vector<int> targets = urlGraph\_[(y - 1) / 3];

if (x % 2 == 0 && find(targets.begin(), targets.end(),

(x-2)/2) != targets.end())

std::printf("%c", (x - 2) / 2 + 'A');

else std::printf(" ");

} else std::printf(" ");

}

}

std::printf("\n");

}

printf("Legend:\n");

typedef std::map<int, std::vector<int>>::iterator it\_type;

for (int i = 0; i < urlIds\_.size(); i++) {

printf("%c -> %s \n", 'A' + i, idUrls\_[i].c\_str());

}

}

BinNode.cpp

#include "BinNode.h"

//Definitions are in header, because (quoting Microsoft):

// "Visual C++ has a limitation in which member

// templates must be fully defined within the

// enclosing class"

Lexer.cpp

#include "Lexer.h"

#include "StringUtil.h"

#include "MatchException.h"

#include <iostream>

using namespace std;

#include <stdlib.h>

#include <string.h>

#include "StringUtil.h"

Lexer::Lexer(char \* text) {

text\_ = text;

int endOfFilePosition = stringUtil::findAinB("</html>", text\_);

if (endOfFilePosition != -1)

\*(text\_ + endOfFilePosition + 7) = '\0';

char\* test = text\_ + endOfFilePosition;

pos\_ = 0;

}

char \* Lexer::getLine() {

int startPosition = pos\_;

char current = text\_[pos\_];

while(current != '\n' && current != '\0') {

current = nextChar();

}

char \* line = (char \*)malloc(pos\_ - startPosition + 1);

line[pos\_ - startPosition] = '\0';

stringUtil::copy(text\_ + startPosition, line, pos\_ - startPosition);

current = nextChar();

return line;

}

/\*\*

\* Returns '\0' upon reaching end of string.

\* Caller is responsible for not calling nextChar() when there is

\* no next char.

\*/

char Lexer::nextChar() {

pos\_++;

rem\_ = text\_ + pos\_;

return text\_[pos\_];

}

char \* Lexer::getRemainingText() {

int charsRemaining = strlen(text\_ + pos\_);

char \* remainder = (char \*)malloc(charsRemaining + 1);

remainder[charsRemaining] = '\0';

stringUtil::copy(text\_ + pos\_, remainder, charsRemaining);

return remainder;

}

/\*\*

\* Reads from the text the text that is sent as parameter.

\* If it is not the same function returns false.

\*/

void Lexer::match(char \* txt) {

int len = strlen(txt);

char \* remaining = text\_ + pos\_;

char current = text\_[pos\_];

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

if (current != txt[i]) {

throw new MatchException;

}

current = nextChar();

}

}

/\*\*

\* Matches text until it meets a certain expression txt.

\* Does not match the txt expression.

\* Returns number of matched characters.

\*

\* Example:

\* TextRemaining: "we are! They are, you are!"

\* matchUntil("are,");

\* TextRemaining after call: "are, you are!"

\*/

int Lexer::matchUntil(char \* txt) {

int displacement = stringUtil::findAinB(txt, text\_ + pos\_);

if (displacement != -1)

pos\_ += displacement;

return displacement;

}

bool Lexer::isNext(char \* txt) {

int relPos = stringUtil::findAinB(txt, text\_ + pos\_);

if (relPos == 0) {

return true;

}

return false;

}

/\*\*

\* Matches word containing English letters.

\* Words are reduced to lowerCase.

\*/

char \* Lexer::matchWord() {

int startPos = pos\_;

char current = text\_[pos\_];

while ((current >= 'A' && current <= 'Z')

|| (current >= 'a' && current <= 'z')) {

current = nextChar();

}

if (/\*current == 0 || \*/startPos == pos\_) {

return nullptr;

}

char \* word = stringUtil::substring(text\_ + startPos, pos\_ - startPos);

stringUtil::toLower(word);

return word;

}

/\*\*

\* Finds next word and returns it or returns nullptr.

\*/

char \* Lexer::findWord() {

char current = text\_[pos\_];

while (!((current >= 'A' && current <= 'Z')

|| (current >= 'a' && current <= 'z')) &&

current != '\0') {

current = nextChar();

}

if (current != '\0') return matchWord();

return nullptr;

}

char Lexer::lookahead(int howmuch) {

return \*(text\_ + pos\_ + howmuch);

}

char \* Lexer::fetchLastNChars(int n) {

char \* retVal = (char \*)malloc(n + 1);

retVal[n] = '\0';

stringUtil::copy(text\_ + pos\_ - n, retVal, n);

return retVal;

}

StringUtil.cpp

#include "StringUtil.h"

#include "MatchException.h"

/\*

standard:

0 for equality

-1 or 1 respectively for other results

\*/

int stringUtil::compare(char \* a, char \* b) {

int pos = 0;

char a\_cur, b\_cur;

while (true) {

a\_cur = \*(a + pos);

b\_cur = \*(b + pos);

if (a\_cur == b\_cur) {

if (a\_cur != '\0') {

pos++;

continue;

}

return 0;

}

if (a\_cur == '\0') {

return -1;

}

if (b\_cur == '\0') {

return 1;

}

if (a\_cur > b\_cur) {

return 1;

}

if (a\_cur < b\_cur) {

return 1;

}

throw new MatchException();

}

}

char \* stringUtil::concat(const char \* a, const char \* b) {

char \* c;

int len\_a, len\_b, len\_c;

len\_a = length(a);

len\_b = length(b);

len\_c = len\_a + len\_b;

c = (char \*) malloc(len\_c + 1);

c[len\_c] = '\0';

copy(a, c, len\_a);

copy(b, c + len\_a, len\_b);

return c;

}

bool stringUtil::contains(char \* subject, char containee) {

int pos = 0;

while (true) {

if (\*(subject + pos) == '\0') {

break;

}

if (\*(subject + pos) == containee) {

return true;

}

pos++;

}

return false;

}

void stringUtil::copy(const char \* from, char \* to, int numberOfChars) {

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

to[i] = from[i];

}

}

/\*\*

\* A naive simple algorithm due to time constraints

\*/

int stringUtil::findAinB(char \* a, char \* b) {

int a\_len = length(a);

int b\_len = length(b);

bool match = false;

for (int i = 0; i <= b\_len -a\_len; i++) {

match = true;

for (int j = 0; j < a\_len; j++) {

if (a[j] != b[j + i]) {

match = false;

break;

}

}

if (match) {

return i;

}

}

return -1;

}

int stringUtil::length(const char \* a) {

int pos = 0;

while (1) {

if (a[pos] == '\0') break;

pos++;

}

return pos;

}

char \* stringUtil::substring(char \* text, int len) {

char \* newString = (char \*)malloc(len+1);

newString[len] = '\0';

copy(text, newString, len);

return newString;

}

void stringUtil::toLower(char \* text) {

int pos = 0;

while (1) {

if (text[pos] == '\0') break;

if (text[pos] >= 'A' && text[pos] <= 'Z') {

text[pos] += 'a' - 'A';

}

pos++;

}

}

Crawler.cpp

#include "client\HttpClient.h"

#include "sitemap\SiteMap.h"

#include "htmlparser\HtmlParser.h"

#include "invertedIndex\InvertedIndex.h"

#include "utils\BinNode.h"

#include "utils\StringUtil.h"

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#include <iostream>

#include <map>

using namespace std;

void main() {

httpClient::init(); // WSASTARTUP

SiteMap siteMap("crawlertest.cs.tu-varna.bg");

InvertedIndex index;

while (siteMap.hasNextUrl()) {

char\* url = siteMap.getNextUrl();

printf("\nFetching page %s. Please wait...\n", url);

char \*response = httpClient::getResponse(url);

if (response == nullptr) {

printf("Page %s was not found!\n", url);

continue;

}

// Fetching actual htmlPage from response

char \*page = httpClient::trimToPage(response);

if (page == nullptr) {

continue;

}

HtmlParser parser;

// Parser parses page and...

parser.parse(page, url);

// ...produces the two indexes that the siteMap

// and the InvertedIndex need

siteMap.updateMap(parser.getLinks(), url);

index.updateIndex(parser.getWords(), siteMap.getDocId(url));

}

system("pause");

do {

system("cls");

printf("%c", 201);

for (int i = 0; i < 30; i++) printf("%c", 205);

printf("%c\n", 187);

printf("%c 1. Print the graph %c\n", 186, 186);

printf("%c 2. Search for a word %c\n", 186, 186);

printf("%c 3. Exit %c\n", 186, 186);

printf("%c", 200);

for (int i = 0; i < 30; i++) printf("%c", 205);

printf("%c\n\n Type your choice: ", 188);

int choice;

cin >> choice;

std::string s;

switch (choice) {

case 1:

system("cls");

siteMap.printPrettyPicture();

printf("\n");

system("pause");

break;

case 2: {

system("cls");

printf("Type in a single word: ");

char word[30];

scanf("%s", word);

// Fetching result from invertedIndex

InvertedIndexValue\* result = index.searchWord(word);

BinNode<Posting>\* postings = result->getPostings();

// Will use map to sort result by number of word occurences

map<int, string> searchResults;

do {

if (postings->getContent() != nullptr) {

searchResults[postings->getContent()->getOccurences()] =

siteMap.getUrlById(postings->getContent()->getDocId());

}

postings = postings->next();

} while (postings != nullptr);

printf("\n");

for (map<int, string>::reverse\_iterator it

= searchResults.rbegin();

it != searchResults.rend(); it++) {

printf("%d -> %s\n", it->first, it->second.c\_str());

}

system("pause");

break;

}

case 3:

return;

};

} while (true);

}