ISSN: 2688-8645

Small Search Engine based on Red and Black Tree

Tingyu Li

College of Computer Science and Engineering, Northeastern University, Shenyang, 110819, China

20203465@stu.neu.edu.cn

Abstract

This procedure uses C + + language to achieve a small search engine each part, including search, analysis, index, query four processes. And establish corresponding data files to speed up the follow-up query. The data structure used is mainly red and black trees. The algorithms used include multi-channel merge sorting (loser tree optimization) and tf-idf algorithm.

Keywords

Red-black Tree; Merge Sort; Term Frequency-Inverse Document Frequency Algorithm; File Operation; Search.

1. Introduction

1.1. Significance of Research

With the rapid development of the Internet, search engine has become an indispensable part of people. Search engine with its powerful crawling ability, the crawling range has covered about 20% of the web pages of the whole network at this stage. Google (Google) has about 6 billion web pages, Baidu has reached 1 billion, and they have formed their own way of life (selling technology, bidding ranking, placing advertisements). Google is known as the greatest company on the Internet, Baidu has become the second largest Internet company in China, which all announce the great development of the search era, and search engines have brought great convenience to people who roam on the Internet. On the one hand, search engines can expand their popularity, for example, Baidu started with search engines, and search engines are a bridge linking users and website promoters. Now Baidu is not only a company, but synonymous with search engines. On the other hand, it can master big data, such as today's artificial intelligence relies on big data statistics to achieve the so-called artificial intelligence, while big data is mostly provided by search engines.

At present, the network resources are very rich, but how to search information effectively is a difficult thing. Research on search engine has become an effective way to solve this problem.

1.2. Solutions

According to the provided crawling pieces (or crawling the web page content by yourself), word segmentation and index are entered, and finally the query function is realized by using the obtained word file and index file.

1.3. Value

The value that searches engine brings to us is to find and dig the information we need. Because there are many web pages in the Internet, but these web pages involve a wide range, it is difficult to find the content we need quickly, so we can crawl, analyze and store the web pages through search engines. When we enter a keyword in the search engine input box, it can rank the related web pages through a series of algorithm commands, and then output, which can greatly improve our search efficiency.

ISSN: 2688-8645

2. Related Works

2.1. Search

Extract the contents of the news.csv file and number the web pages.

2.1.1. Search for Specific Content

The obtained news.csv file is read, and a two-dimensional container vector < vector < string > > is used to store the relevant web page information. The first column of each line is the website address of the website, the second column is the title of the website, and the following will store the web page content; Each web page occupies one line, and the subscript corresponding to one dimension of the container is the number of the web page, and the number starts from 0.

2.2. Analysis

Read the page information and segment the words, and create a word file and a temporary index file.

2.2.1. Analysis of Specific Content

The content of the web page in the web page information container is segmented (space and punctuation marks are used as separators), and the words of each web page are stored with map < string, int >, string is the word itself, int is the corresponding word number, and a word file is generated after reading all the words, and the storage format is. txt. A temporary index file is generated with map < string, int >, and the storage format is. txt. A word frequency array is used to record the occurrence times of the words corresponding to the word number of each web page in the web page, the array is emptied when the next web page is read, and twenty web page words are written into a temporary index file as a group, in which the first column of the word number, the second column of the web page number and the third column of the occurrence times of the web page.

2.3. Index

An interface function that generates an inverted index file with the temporary index file generated by analysis and provides it to the query function.

2.3.1. Specific Contents of Index

The k-path merging and sorting algorithm optimized by loser tree is used to merge all temporary index files into inverted index files, and the word frequency of all words in all web pages where the words appear and the label of the web pages are stored according to the word number order.

2.4. Queries

In response to the user's request, according to the inverted index to obtain the relevant pages and word frequency, calculate the page ranking, the relevant pages in order to return the query results to the user.

2.4.1. Query Specific Content

After segmenting the query text input by the user, the number of the corresponding search word is obtained by using the obtained word file, and then the inverted search file is read by using the interface function. The corresponding web page number and the word frequency in the web page are searched by the word number in the inverted index file, and the relevant web pages are searched, modified and stored by using a map < web page, term frequency-inverse document frequency (hereinafter referred to as "1" tf-idf) >. In this process, the inverse document frequency (hereinafter referred to as idf) value of the search term is calculated according to the total number of web page files, the term frequency (hereinafter referred to as tf) value is calculated according to the total number of web page words, and the tf-idf value of

the related web page is modified by multiplying the tf and idf values of the search term. At the same time, a vector < tf-idf, web page > is used to sort related web pages. Finally, it returns a vector that has been ordered by the size order of tf-idf values, and then outputs the corresponding URL and title according to the webpage number.

3. Discussion on Main Achievements

3.1. Search

The search part is mainly composed of the class readCSV, and the main function is LoadDataFromCSV in the class, which realizes reading the web page information in the news.csv file and putting it into the container. Specific functions are shown in Table 1.

Table 1. List of related functions

Function name	Function function	Parameter	Parameter function
LoadDataFromCSV	Read the news.csv file and place it in a container whose subscript corresponds to the number of each web page. The first column is the website address, the second column is the title, and then the webpage content.	① vector < vector < string > > & content	Used to store the read web page information. The member dataF in the class readCSV is the path to the new.csv file.

3.2. Analysis

The analysis part is mainly composed of isletter, isnumber, Switch, map functions, which are implemented separately, judging whether characters are letters or numbers, transforming numbers into strings, creating word files, creating word map and temporary index files. Specific functions are shown in Table 2.

Table 2. List of related functions

Function name	Function function	Parameter	Parameter function
isletter	Determines whether the character is a letter.	① char c	c is the character to be judged.
isnumber	Determines whether the character is a number.	① char c	c is the character to be judged.
Switch	Converts a number to a string.	① int a	A is the number to be converted to a string.
map	Segmentation of Web page content stored in container, creation of word map and temporary index file	① map < string, int > * m, ② vector < vector < string > > & content	m stores the word and its corresponding word number, and content is the container for storing web page information.

3.3. Index

The index part is mainly composed of several functions such as partition, random, Quicksort, part_sort, adjust, CreateTree, k_merge, merge, save, read and search, and the structure word, which realizes the operation of merging temporary index files into inverted index files, storing and reading stored word label files, and querying all the appeared web pages corresponding to given word label and the word frequency under the web pages.

3.4. Inquiry

The query part is mainly composed of four functions: inquiry, gettf_idf, trans_map and jiang, which respectively realize word segmentation of search terms, obtain relevant web page

numbers and tf-idf values, convert map into vector, and output quick sorting in descending order. Specific functions are shown in Table 4.

Table 3. List of related functions and structures

Function name	Function function	Parameter	Parameter function
partition	One Partition in Quick Sorting	(1) word a [] ② int p ③ int r	A [] is the array to be sorted, p is the starting position to be sorted, and r is the ending position to be sorted
random	Optimizing the Time Complexity of Quick Sorting by Using Random Selection Benchmark	(1) word a [] ② int p ③ int r	A [] is the array to be sorted, p is the starting position to be sorted, and r is the ending position to be sorted
Quicksort	Quick sort call function	(1) word a [] ② int p ③ int r	A [] is the array to be sorted, p is the starting position to be sorted, and r is the ending position to be sorted
part_sort	Each temporary index file is quickly sorted for subsequent k-path merge sorting	None	Directly adjust the corresponding. txt file without parameters
adjust	k-path merge sorting uses the loser tree to adjust the loser tree once, and puts the minimum value at the root of the tree	① word* b ② int* l ③ int s	b is the data stored in the label corresponding to the loser number, l is the loser tree array, and s is the label corresponding to the modified point
CreateTree	Create a loser tree and initialize	① ifstream* ary ② word* b ③ int* l	*For all the file input stream array used to read each file to be merged, b for the number of losers corresponding to the label stored in the data, l for the loser tree array.*
k_merge	Calling function of k-path merge	① ifstream* ary ② int* l ③ word*b	*For all the file input stream array used to read each file to be merged, b for the number of losers corresponding to the label stored in the data, l for the loser tree array.*
merge	Calling function for merging the overall temporary index file into an inverted index file	Parametric-free	Read the corresponding txt file directly
search	Query all the web pages corresponding to the given word label and the word frequency under the web page.	① int id	The id is the label corresponding to the word to be queried
save	Saves the generated word label file so that the next call can be read directly from the file	① map < string, int > * m	m is the word label data structure to be stored
read	Read the word label from the generated word label storage file	① map < string, int > * m	m is the word label data structure to be initialized
Structure name	Action	Member list	Action
word	Stored as a data structure (word label, web page label, word frequency on the web page) triple	① int id ② int idex ③ int sum	id stands for word label, idex stands for page label, and sum stands for word frequency.

4. Implementation of Key Technologies

4.1. Red and Black Trees

Simulate the map in stl to achieve the goal of storing key-value pairs and sorting them according to their key values. Because the normal binary tree will degenerate in some cases, and the AVL

tree with perfect balance will consume more time when adjusting nodes because of its high requirements. Therefore, we choose red and black trees, that is, a transformation form of 2, 3 and 4B trees, to realize the simulation of map.

Table 4. List of related functions

Function name	Function function	Parameter	Parameter function
inquire	Segment the user's search content, and store the relevant word numbers by vector	① map < string, int > * m ② int files	m is the word index file, find the corresponding word number, and files is the total number of web page files
gettf_idf	According to the word number and the inverted index interface function, the tf-idf value of the relevant web page is calculated and stored in the map	① vector < int > intse, ② int files	*intse is the word number content of the search term, and files is the total number of web page files*
trans_map	Convert map < page number, tf-idf > to vector < tf-idf, page number >	① map < int, float > * m1	Pass the stored related web page number and tf-idf value to the trans_map () function
jiang	Quicksort related web pages according to tf-idf value	vector < pair < int, float > > & m2 ② int x ③ int y	m2 is the vector container for storing related web pages, x is the initial position of the container, and y is the last position

Algorithm 1: Red and Black Trees

Input: Type of key and key value, key-value pair to insert.

Output: Container map, in which elements are sorted by key size.

4.1.1. Properties of Red and Black Trees

1. Each node is either red or black.
2. The root node is black.
3. Each leaf node (NIL) is black.
4. If a node is red, its two child nodes are black.
5. For each node, the simple path from this node to all its descendant leaf nodes contains the same number of black nodes.

Under the requirement of property 4, the number of black nodes on any simple path (top-down path) must be greater than the number of red nodes. We call the number of black nodes on the simple path from the root node to any leaf node the "black height" of the tree. Then when all the black heights are the same, it is conceivable that the height of the tree must be greater than and less than twice the black height. Next, we need to understand the relationship between black height and the total number of nodes n. Imagine pulling away all the red nodes in the red-black tree, and the left black node happens to be a complete binary tree. Assuming that the number of black nodes is nb, the height of the complete binary tree is lognb. Even if the red nodes are restored, the black height of the red-black tree is still lognb, regardless of the number of red nodes. Considering nb < n, the black height of red-black trees is also less than logn. Combined with our previous conclusion that "the height of the tree must be greater than the black height and less than twice the black height", it can be seen that the height of the tree is less than 2logn.

That is to say, Property 5 ensures that the red-black tree is an approximately balanced binary tree, which ensures that the search time complexity is O (logn). It can be said that properties 4 and 5 are the core of the red-black tree. In order to ensure that these two properties are not

ISSN: 2688-8645

destroyed when modifying the red-black tree, the algorithm becomes somewhat complicated, and the relevant details are described in detail below

4.1.2. Rotation of the Tree

Rotation can adjust the structure of binary search tree, and the order of binary search tree remains unchanged before and after rotation. Figure 1 is a rotation operation on two nodes.

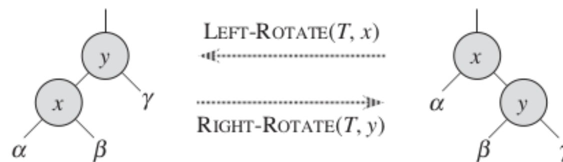

Figure 1. Rotation of the tree

4.1.3. Node Insertion

Insertion is a core operation of red-black trees, and it is also an operation that may cause damage to red-black properties. For the common binary search tree, the insertion operation only needs to do a search, find the appropriate insertion position, and insert the new element as a leaf node. However, for red-black trees, the first question is, should the newly added nodes be red or black?

If it is black, properties 1 ~ 4 are satisfied, but property 5 will be destroyed, because the number of black nodes on the simple path where the new node is located is 1 more than that on other paths.

If it is red, properties 1 ~ 3 and 5 are satisfied, but property 4 may be destroyed, because the parent node of the new node may also be red.

It seems that no matter what color is set, there is no guarantee that the red and black properties will continue to be satisfied. The wiser choice is the second one, because the second one is only likely to be destroyed, and we can only do some remedial measures if the red and black properties are destroyed.

According to different tree structures, different correction strategies need to be adopted. But in general, we can divide it into three situations, as shown in Figure 2.

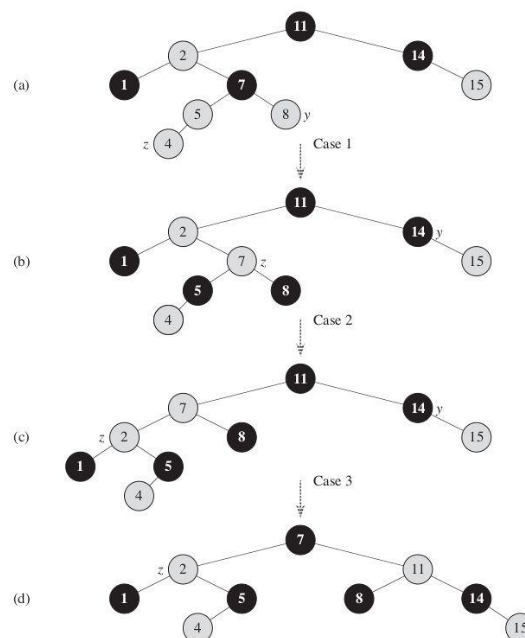

Figure 2. Tree Adjustment Strategies in Different Situations

In the graph, dark nodes are black nodes and light nodes are red nodes. Nodes marked by z are those that break the red-black property. It should be noted that z is not necessarily a newly inserted node, because in the correction algorithm, the tree structure may need to be corrected several times, so z is a new node at first, but then z may rise along the tree.

Let's look at the first case first. As shown in Figure (a), the uncle node y of z is red. At this time, we turn the parent node of z and the uncle node of z into black at the same time, and turn the parent node of the parent node of z into red, and get the result shown in Figure (b). In this operation, the subtree represented by Node 7 satisfies the red-black property. At the same time, however, Node 7 may break the red-black property again, depending on whether the parent Node 2 of Node 7 is red or not. If it is red, we mark Node 7 as the new z Node and enter the next cycle. If it is black, the red-black properties are satisfied, the cycle is over, and the whole tree is corrected.

On the contrary, if the first case is not satisfied, then the tertiary node y of z is black, that is, the case shown in graphs (b) and (c). The difference between Figure (b) and Figure (c) is whether z is a right child or a left child, which we call Case 2 and Case 3 respectively. However, let's not distinguish between situations 2 and 3, and first come up with the most direct remedial measures. Since 2 and 7 are both red, it is obvious that in order to repair the red-black property 4, we need to turn one of them into black. From the treatment of the first case, we can find that we tend to move the problematic nodes upward, and try our best to ensure that the lower nodes meet the red and black properties. Here, we still follow this principle, and turn the upper node into black, and then we will find out the truth of this principle. Of course, the upper nodes in Figure (b) and Figure (c) are 2 and 7, respectively. After dyeing them black, the red-black property 4 was repaired, but unfortunately, the red-black property 5 was destroyed again because there was an extra black node. In order to further implement the idea of moving the problematic node up, we forced the black height on the left side to be reduced by one by dyeing z's grandfather node 11 red. Note that this is feasible because z's uncle node 14 happens to be black, thus avoiding the embarrassing situation of two consecutive red nodes, which is also the key to the second and third situations. However, although the black height on the left side is reduced by one, the black height on the right side is also reduced by one, and the black height on the left side is still one more than that on the right side. Cleverly, at this time, only one right-hand rotation to root node 11 can completely solve this problem. After dextral rotation, the new root node is 2 or 7 (transformed from Figure (b) and Figure (c), respectively), and it is black, resulting in the right black height plus one, which is equal to the left black height, and the red-black nature is repaired.

However, our above explanation does not distinguish between case two and case three. In fact, the second case can't repair the red and black nature with the above operation. Because after right-turning, the red node z will be hung on the red node 11, thus destroying the red-black property 4 again. It is equivalent to shifting the conflict from the left to the right, instead of moving the conflict up, which will lead to an infinite loop and never completely solve the problem. The correct way is to do a left-hand rotation on Node z in Case 2, convert it to Case 3, and then deal with it according to Case 3.

The above is the realization of the basic method of red and black tree. In addition, this map class also provides a search and traversal method, and the common binary search tree is the same, so no longer repeat, code details in the attachment map class implementation.

4.2. k-path Merge Sort

The core of the algorithm is to use the loser tree to increase the k value without increasing the internal sorting time.

Algorithm 2 k-path merging

Input: Temporary index file

ISSN: 2688-8645

Output: Merged inverted index file

```
/**
* arry: All data that is multiplexed
* b: Hold the first address array of multiplexing
* ls: loser tree array
* temp_sum: Total number of temporary index files, i.e. several merges
 */
//Create a loser tree
void CreateTree(ifstream* ary, word* b, int* l) {
   for (int i = 0; i < temp_sum; i++) {
      int x, y, z;
      ary[i] >> x >> y >> z;
      b[i].id = x;
      b[i].idex = y;
      b[i].sum = z;
   }
   b[temp_sum].id = mink;
   for (int i = 0; i < temp_sum; i++)
      l[i] = temp_sum;
   for (int i = temp_sum - 1; i >= 0; i--)
      adjust(b, l, i);
 //Adjust the loser tree so that the minimum value is at the root node
}
//Merge the ordered temporary index files, which is the core code of the algorithm
void k_merge(ifstream* ary, int* l, word* b) {
   int pre = -1;
   string name = "..\\final file.txt";
   ofstream final;
   final.open(name, ios::out);
//Store data in the final inverted index file while (b[l[0]].id != maxk) {
      int s = l[0];
      if (pre == b[s].id) {
         final << " " << b[s].idex << " " << b[s].sum;
      }
      else {
         if (pre != -1)
            final << "$" << endl;
         final << b[s].id << " " << b[s].idex << " " << b[s].sum;
         pre = b[s].id;
      }
      int x, y, z;
      if (ary[s] >> x >> y >> z) {
         b[s].id = x;
         b[s].sum = z;
         b[s].idex = y;
      }
      else {
         b[s].id = maxk;
      }
      adjust(b, l, s);
```

ISSN: 2688-8645

```
  }
  final << "$" << endl;
  final.close();
}
//Overall merge call function
void merge () {
//Quicksort each temporary index file once
part_sort ();
part_sort();
  ifstream* in = new ifstream[temp_sum];
  string file = "..\\temp\\temp_";
  for (int i = 0; i < temp_sum; i++) {
    string temp = file + to_string(i);
    temp += ".txt";
    in[i].open(temp, ios::in);
  }
  int* l = new int[temp_sum];
  word* b = new word[temp_sum + 1];
//Create a loser tree
    CreateTree(in, b, l);
//Merge k paths and generate inverted index files
     k_merge(in, l, b);
}
```

4.3. tf-idf Algorithm

The core of the algorithm is to calculate the tf and idf values of search terms. For each web page where search terms appear, the tf-idf values of a group of search terms are calculated respectively. By adding them, the tf-idf values of the whole web page file can be obtained. Then, according to the tf-idf value of the web page, sort it in descending order, and output the website address and title corresponding to the web page number.

Algorithm 3. tf-idf

Input: Word number file, inverted index file, total number of web page files

Output: Web page numbering container ordered by tf-idf value

```
vector < pair < INT, FLOAT > > gettf_idf (vector < INT > intse, INT files) {//intse word number, total number of files
INT t; //Temporary storage of word numbers
FLOAT idfre; //Inverse document frequency
float tifre; //Web page tf-idf value
float cifre; //Word frequency
 vector<pair<INT, FLOAT>> w_c;
//Web Pages-Word Frequency Container
map<int, float>* m1 = NEW map<INT, FLOAT>();
//Web pages found
vector<pair<INT, FLOAT>> m2;
 //Use the vector to save the web page
 for (i = 0; i < intse.size(); i++) {
//Traverses all page numbers and corresponding word frequencies where search terms appear
```

ISSN: 2688-8645

```
    t = intse[i];
    w_c = search(t);
    //search function looks for web pages-word frequency,
    idfre = log(files / (w_c.size() + 1));
    FOR (j = 0; j < w_c. size (); j + +) {
              t = m1-> Find (w_c [j]. first);
              IF (t! = NULL) {//Find page
              LONG LONG INT a = Fre_all [w_c [j]. first];
              cifre = w_c [j]. second/a;
              t-> Second + = idfre * cifre;
              //Modify key-value pairs
              }
              ELSE {//No page found
              LONG LONG INT a = Fre_all [w_c [j]. first];
              cifre = w_c [j]. second/a;
              tifre = idfre * cifre;
              m1->Insert(pair<int, float>(w_c[j].first, tifre));
                    }
              }
              w_c.clear();
    }
    RETURN trans_map (m1);
}

vector < pair < INT, FLOAT > > inquire (map < STRING, INT > * m, INT files)
//Query mp for inverted index (page-word frequency)
{
    vector<string>se; //Store search terms
    vector<int>intse; //Store search term numbers
    FOR (i = 0; i <= search.size(); i++) {
    //Segment the search content and store it in vector
    IF (search[i] == ' ') {
     se.push_back(sub);
     sub = "";
  }
  ELSE IF(i != search.size()) {
     sub += search[i];
  }
  IF(i == search.size() && sub != "")
     se.push_back(sub);
}
INT a = 0;
FOR(i = 0; i < se.size(); i++) {
  t = m->Find(se[i]);
```

ISSN: 2688-8645

```
IF (t != NULL) {//map to find word numbers
        a = t->Second;
        intse.push_back(a);
  }
    //Save word numbers in intse (vector)
  a = 0;
}
map<float, int>* m_w = NEW map< float, int>();
//Sort the corresponding page numbers
RETURN gettf_idf(intse, files);}
```

5. Performance Analysis

5.1. Search

① Traverse the News.Csv File

Time complexity is O (n): Traversing the web page information in the news.csv file, the time taken is linear, that is, O (n).

Space complexity is O (n): Record the information of each web page with a container, and the occupied space is linear, that is, O (n).

5.2. Analysis

①Traverse the container for storing web page information

Time complexity is O (n): Traversing the part of the container where web page information is stored, the time taken is linear, that is, O (n).

Space complexity is O (n): use map to store words and corresponding word numbers, and create temporary index files, which occupy linear space, that is, O (n).

5.3. Index

①k-path merging sorting algorithm

Let each: The data of the file is n, and there are m files in total. In k-path balanced merge, if the loser tree is not used, each data needs to be compared k-1 times, a total of n data, and each merge needs to be compared (n-1) (k-1) times. If there are m initial merging segments, the number of merging passes is logk(m), and the total number of comparisons is log-k(m) (n-1) (k-1). After introducing the loser tree, the number of comparisons of each data islog-2(k) (binary tree only needs to be compared with the parent node), and the total number of comparisons is log-k(m) (n-1) (log-2k), which is simplified to log-2(m)(n-1).

Time complexity: Operand is mnlogn+logm(n-1) + logm(mn-1)

The number of file merge operations is: logm/logk.

Space complexity O (m): The loser tree and data storage array need to be established as auxiliary space.

5.4. Inquiry

① tf-idf algorithm

Time complexity is O (n): according to the web page-word frequency corresponding to each word in the inverted index file search (search process O (1)), the tf-idf value of the found web page is continuously updated, and the time taken is linear, that is, O (n).

Spatial complexity is O (n): the spatial resource required to record the tf-idf value of each web page is the web page where all search terms have appeared, that is, O (n).

② Quick platoon

Time complexity: The best-case time complexity is O (nlogn), and the worst-case time complexity is O (n ^ 2). O (nlogn) on average.

The space complexity is O (1).

6. Background

Search engine references to a system that collects information from the Internet agreement to a certain strategy and uses specific computer programs to collect information from the Internet, and provides users with retrieval services after organizing and processing information, and displays the relevant information retrieved by users to users. search engines include full-text indexing, catalog indexing, metasearch engines, vertical search engines, collective search engines, portal search engines, and free link lists.

Search engineers exist to help people find the information they need, it can Expand visibility, Master big data.

Development trend of China's search engine industry: 5G, AI and other technologies will enable search engine products to take a new level, Intelligent search and voice search are inevitable trends in the future, In addition to providing basic search functions, intelligent search can also provide automatic identification of user interests, content semantic understanding, information filtering and pushing functions, with knowledge processing ability and understanding ability, can import information retrieval from the current keyword-based level to based on knowledge and concepts. search results are al So more humane, closer to user needs, faster response, higher search efficiency, focus on providing knowledge and services, showing the development trend of intelligence, personalization, scenario, diversification, collaboration and interactive convention.

References

[1] Li Zhaofeng. Construction method of efficient web page classifier in topic search engine [J]. Science and Technology Bulletin, 2013, 29 (08): 109-111. DOI: 10.13774/J.cnki.kjtb.2013. 08.047.

[2] Ma Botao, Sun Peng, Zhu Xiaoyong. Research Review of Red and Black Tree Algorithm [J]. Network New Media Technology, 2018, 7 (04): 56-62.

[3] Chen Guang, Wood Peng. An improved algorithm of red-black tree [J]. Journal of Inner Mongolia Normal University (Educational Science Edition), 2012, 25 (12): 75-79.

[4] Gao Qing, Jiang Fan. Red-black tree algorithm and its application [J]. Software Guide, 2008 (09): 40-42.

[5] Elmasry A, Kahla M, Ahdy F, et al. Red–black trees with constant update time[J]. Acta Informatica, 2019, 56 (5).