Advanced Data Structures and Algorithm Analysis

Project 1: Roll Your Own Mini Search Engine

Date: 2024-09-30

2024-2025 Autumn&Winter Semester

Table of Content

Chapter 1: Introduction

1.1 Problem Description

The project required us to create a **mini search engine** which can handle inquiries over "The Complete Works of William Shakespeare".

Here are some specific requirements:

- Run a word count over the Shakespeare set, extract all words from the documents by word stemming and try to identify the stop words.
- Create a customized inverted index over the Shakespeare set with word stemming. The stop words identified must not be included.
- Write a query program on top of the inverted file index, which will accept a user-specified word (or phrase) and return the IDs of the documents that contain that word.
- Run tests to show how the thresholds on query may affect the results.

1.2 Purpose of Report

- Show the details of the implementation of the mini search engine by showcasing essential data structures and algorithms.
- Demonstrate the correctness and efficiency of the program by analysis based on testing data and diagrams.
- Summarize the whole project, analyze the pros and cons of the mini search engine, and put forward the prospect of further improvement.

1.3 Backgound of Data Structures and Algorithms

1. **B+ Trees**: It's a improved version of search trees, widely used in the relational database and file management in operating systems. We will use this data structure to store and access to the inverted index.
2. **Queue**:
3. **Hashing**:

Chapter 2: Data Structure / Algorithm Specification

2.1 Algorithm Architecture

The overall algorithm architecture in the program is shown below:

In the following sections, I will introduce these algorithms from top to down, but with some slight adjustement, in the hope that you can gain a deeper insight into my whole program.

2.2 The Main Program

2.3 Word Count

2.4 Stop Words

2.5 Word Stemming

2.6 Inverted Index

Maybe this is the most complicated part of the whole program, because in this part we have a relatively complex algorithm architecture, and we use a couple of data structrues and algorithms, such as B+ trees, implicit queue ADT and linked list ADT. Here is the diagram of the functions used in the inverted index:

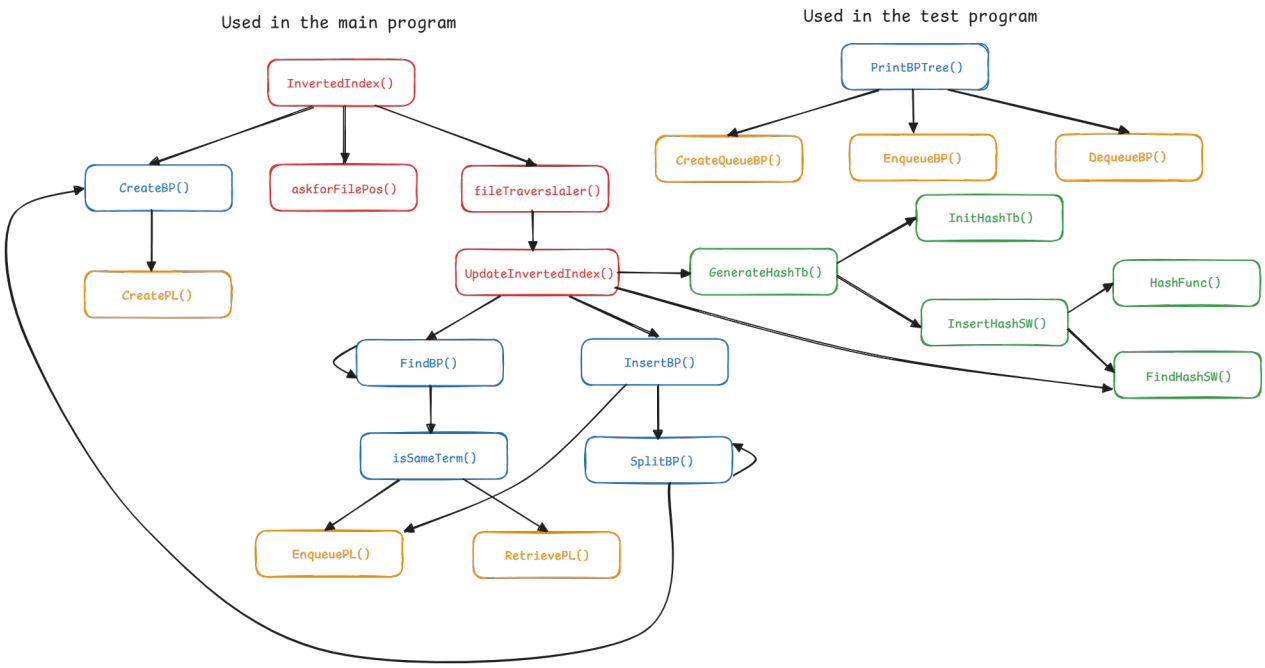

Figure 1: Relation diagram of all functions for inverted index

- Red: Overall Functions
- Blue: B+ Tree Operations
- Green: Hashing Functions
- Yellow: Other Functions

We'll introduce these functions in detail below.

2.6.1 Overall Functions

(1) InvertedIndex

Function: The highest-level function, which users can call it directly.

Inputs:

- *isTest*: `-t` or `--test` mode, just use one particular file
- *containStopWords*: `-s` or `--stopwords` mode, contain stop words when building inverted index

Outputs:

- *InvIndex*: A B+ tree containing the inverted index

Procedure: InvertedIndex(*isTest*: **bool**, *containStopWords*: **bool**)

```
1  Begin
2      InvIndex ← CreateBP()
```

```
3     askforFilePos(dir, fname, isTest)
4     // dir: directory name, fname: filename
5     InvIndex ← fileTraversaler(InvIndex, dir, fname, isTest,
      containStopWords)
6     if InvIndex→size > 0 then
7         print("Build successfully!")
8     else
9         print("Fail to build an inverted index!")
10    endif
11    return InvIndex
12 End
```

(2) askforFilePos

Function: Ask for the position of the directory or file.

Inputs:

- *dir*: directory name
- *fname*: file name
- *isTest*: `-t` or `--test` mode, just use one particular file

Outputs: None, but will update either *dir* or *fname*

Procedure: askforFilePos(dir: **string**, fname: **string**, isTest: **boolean**)

```
1  Begin
2     if isTest is true then
3         print("Now testing the correctness of inverted Index:")
4         print("Please input the name of the input sample file:")
5         input("Name:", fname)
6     else
7         print("Now building an inverted Index:")
8         print("Please input the directory of the documents:")
9         input("Path:", dir)
10    endif
11 End
```

(3) fileTraversaler

Function: Make a traversal of all files(or a single file) and build the inverted index from them(or it).

Inputs:

- *T*: A B+ tree containing the inverted index
- *dir*: directory name
- *fname*: file name
- *isTest*: `-t` or `--test` mode, just use one particular file
- *containStopWords*: `-s` or `--stopwords` mode, contain stop words when building inverted index

Outputs: An updated B+ tree *T*

Procedure: fileTraversaler(T: **BplusTree**, *dir*: **string**, *fname*: **string**, *isTest*: **boolean**, *containStopWords*: **boolean**)

```
1  Begin
2      docCnt = 0 // Count the number of documents and act as the
       index of the documents at the same time
3      if isTest == false then
4          if dir exists then
5              for file in the dir directory do
6                  filename ← the name of file // string
7                  docNames[docCnt] ← filename
8                  // docNames: an array containing names of
                   documents(global variable)
9                  wholePath ← dir + "/" + filename
10             // wholePath: the complete path of the file to be read
11             end
12             closefile(fp)
13         else
14             Error("Could not open directory!")
15         endif
16     else
17         docNames[docCnt] ← fname
18         dir ← DEFAULTFILEPOS // Constant: "tests"(string)
19         wholePath ← dir + "/" + fname
```

```
20      endif
21      fp ← openfile(wholePath, "r") // read mode
22      // fp: the pointer to the file
23      T ← UpdateInvertedIndex(T, docCnt, fp, containStopWords)
24      return T
25  End
```

(4) UpdateInvertedIndex

Function: Update the Inverted Index while reading a new document.

Inputs:

- *T*: A B+ tree containing the inverted index
- *docCnt*: the index of the document
- *fp*: pointer to the file
- *containStopWords*: `-s` or `--stopwords` mode, contain stop words when building inverted index

Outputs: An updated B+ tree *T*

Procedure: UpdateInvertedIndex(T: **BplusTree**, *docCnt*: **integer**, *fp*: **filePointer**, *containStopWords*: **boolean**)

```
 1  Begin
 2      H ← GenerateHashTb()
 3      while reading texts in the file pointed by fp do
 4          if find an English word then
 5              term ← the English word
 6              if containStopWords == false and FindHashSW(term, H,
                true) >= 0 then
 7                  continue
 8              endif
 9              term ← WordStemming(term)
10              isDuplicated ← false
11              nodebp ← FindBP(term, docCnt, T, isDuplicated)
12              if isDuplicated == false then
13                  T = InsertBP(term, docCnt, nodebp, T)
14              endif
```

```
15          else
16              continue searching for next English word.
17          endif
18      end
19      return T
20  End
```

2.6.2 B+ Tree Operations

(1) CreateBP

Function: Create a B+ tree.

Inputs: None

Outputs: A new and initialized B+ Tree

Procedure: CreateBP()

```
1  Begin
2      Allocate a memory block for new B+ tree T
3      for all data and children in T do
4          Allocate memory blocks for term and poslist of the data, and
           children
5          // Use CreatePL() to intialize the poslist
6      end
7      T→size ← 0
8      T→childrenSize ← 0
9      T→parent ← NULL
10     return T
11 End
```

(2) FindBP

Function: Find a term in B+ tree.

Inputs:

- *term*: term
- *docCnt*: the index of the document

- T: inverted index
- *flag*: true if the term is found, false otherwise
- *isSearch*: mark the find mode(-f or –find)

Outputs: the updated B+ tree T or recursively call itself again

Procedure: FindBP(*term*: **string**, *docCnt*: **integer**, T: **BplusTree**, *flag*: **booleanPointer**, *isSearch*: **boolean**)

```
1  Begin
2      if T→childrenSize == 0 then
3          isSameTerm(term, docCnt, T, flag, isSearch)
4          return T
5      endif
6      pos ← −1
7      for i in range(0, T→size) do // not contains T→size
8          if term has less lexicographical order than T→data[i]→term
           then
9              pos ← i
10             break
11         endif
12     end
13     if pos == −1 then
14         pos ← i
15     endif return FindBP(term, docCnt, T→children[pos], isSearch)
16 End
```

(3) isSameTerm

Function: Check if the term exists in the B+ tree.

Inputs:

- *term*: term
- *docCnt*: the index of the document
- *nodebp*: the appropriate node where the term may exists or will exists after insertion
- *flag*: true if the term is found, false otherwise
- *isSearch*: mark the find mode(-f or –find)

Outputs: None, but may update the flag and print some information regarding *term*

Procedure: isSameTerm(*term*: **string**, *docCnt*: **integer**, *nodebp*: **NodeBP**, *flag*: **booleanPointer**, *isSearch*: **boolean**)

```
1  Begin
2      if nodebp→size > 0 then
3          for i in range(0, T→size) do // not contains T→size
4              if term == nodebp→data[i]→term then
5                  if isSearch == false then
6                      EnqueuePL(docCnt, nodebp→data[i]→poslist)
7                  else
8                      poslist ← nodebp→data[i]→poslist
9                      size ← poslist→size
10                     cnt ← 0
11                     print("Successfully find the word!")
12                     print("The word was found in files below:")
13                     posArr ← RetrievePL(poslist)
14                     for j in range(0, size) do // not contains size
15                         if posArr[j][1] <= 1 then
16                             print("{docNames[posArr[j][0]]}: {posArr[j][1]}
                               time")
17                         else
18                             print("{docNames[posArr[j][0]]}: {posArr[j][1]}
                               times")
19                         endif
20                         cnt ← cnt + posArr[j][1]
21                     end
22                     print("Frequency: {cnt}")
23                     print("————————————————————————")
24                 endif
25                 flag ← true
26                 break
27             endif
28         end
```

```
29      endif
30  End
```

(4) InsertBP

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(5) SplitBP

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(6) PrintBPTree

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

2.6.3 Hashing Operations

(1) GenerateHashTb

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(2) InitHashTb

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(3) FindHashSW

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(4) InsertHashSW

Function:

Inputs:

Outputs:

Procedure:

```
1 Begin
2     End
```

(5) HashFunc

Function:

Inputs:

Outputs:

Procedure:

```
1 Begin
2     End
```

2.6.4 Other Functions

(1) CreateQueueBP

Function:

Inputs:

Outputs:

Procedure:

```
1 Begin
2     End
```

(2) EnqueueBP

Function:

Inputs:

Outputs:

Procedure:

```
1 Begin
2     End
```

(3) DequeueBP

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(4) CreatePL

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(5) EnqueuePL

Function:

Inputs:

Outputs:

Procedure:

```
1  Begin
2      End
```

(6) RetrievePL

Function:

Inputs:

Outputs:

Procedure:

```
1 Begin
2     End
```

2.7 Query

Chapter 3: Testing Results

3.1 Test 1: Inverted Index

3.2 Test 2: Thresholds for Queries

3.3 Test 3: Speed Test

3.4 (Maybe)Debug Mode

Chapter 4: Analysis comments

4.1 Time Complexity

4.2 Space Complexity

Appendix: Source code

5.1 File Structure

References

Declaration

We hereby declare that all the work done in this project titled "Roll Your Own Mini Search Engine" is of our independent effort as a group.