كلية علوم الحاسب والمعلومات
قسم تقنية المعلومات

King Saud University
College of Computer and Information Sciences
Department of Information Technology

Csc212 Data structure

CSC212 project

Section no.66510
Group no.37

Student name	ID	Tasks
Ftoon Fawaz binkhattaf	443200499	We worked on all classes together, report
Maha aljandoul	444200646	We worked on all classes together, report
Alanoud Alhayyan	444200067	we worked on all classes together , report

instructor’s name: Abeer Alshaya.

1. Explanation of our search engine:

In our code, we implement a search engine that stores and retrieves information from documents.
Components:

1.1 Term and TermRank Classes:

- The classes represent the words (terms), and information associated with them.
- Term: Stores a word along with a boolean array indicating whether the word appears in any documents.
- TermRank: identifies the frequency with which a word appears in each document.

1.2 Inverted index

- This is a data structure that maps words to the documents they appear in.

  Different implementations are provided:

- Inverted index with BST: Uses a BST to store words and their associated Term objects.
- Inverted index with AVL: Uses an AVL tree for efficient storage and retrieval.
- Inverted index with ranking: Stores word frequencies for each document

1.3 Search Engine Class:

- Loads data from a specified file

  Provides methods for:

- Term Retrieval: Finds documents containing a specific word or term.
- Boolean Retrieval: Retrieves documents based on Boolean operations such as AND, OR, and NOT.
- Ranked Retrieval: Orders documents by relevance using word frequency or ranking algorithms.
- Indexing Information: Provides details about indexed documents and their associated tokens.

2. How the code works :

2.1 Indexing Process:

- Documents are analyzed and processed.
- Words are extracted and organized into suitable data structures, such as a simple index, inverted index, or ranked inverted index.
- For ranked retrieval, word frequencies are calculated and recorded.

2.2 Searching:

- Term Retrieval: The search engine locates the specified word in the data structure and retrieves the corresponding documents.
- Boolean Retrieval: The search engine analyzes the query, applies Boolean operators (AND, OR) to combine results for multiple terms, and returns matching documents.
- Ranked Retrieval: The search engine computes relevance scores for documents using word frequencies or other ranking methods and provides the highest-ranked results.

3. Performance analysis

3.1 Class inverted index

The Inverted_Index class is another way to implement an inverted index. It stores words and the documents that contain those words. Unlike the previous implementation, this one uses a linked list to organize the words and their corresponding document IDs.

Method: adddocument()

Code	Frequency	Total
if (DocumentIndex.empty()) {}	1	O(1)
DocumentIndex.findFirst();	1	O(1)
while (!DocumentIndex.last()) {}	n−1	O(n-1)
if (DocumentIndex.retrieve().word.word.compareTo(word) == 0) {}	n-1	O((n-1)*c)
DocumentIndex.insert(term);	1	O(1)
total		O(n*c)

Method: found()

Code	Frequency	Total
if (DocumentIndex.empty()) {}	1	O(1)
DocumentIndex.findFirst();	1	O(1)
while (!DocumentIndex.last()) {}	n−1	O(n-1)
if (DocumentIndex.retrieve().word.word.compareTo(word) == 0) {}	n-1	O((n-1)*c)
total		O(n*c)

Method: AND_OR_Function

Code	**Frequency**	**Total Complexity**
if (!string.contains(" OR ") && !string.contains(" AND ")) {}	1	**O(k)**
String[] AND_ORs = string.split(" OR ");	1	**O(k)**
result = AND_Function(AND_ORs[0]);	1	**O(t*h)**
for (int i = 1; i < AND_ORs.length; i++) {}	t-1	**O((t−1)·(h+50))**
Total		**O(k+t·(h+50))**

Method: AND_Function

Code	**Frequency**	**Total Complexity**
String[] ANDs = string.split(" AND ");	1	**O(k)**
if (this.found(ANDs[0].toLowerCase().trim())) {}	1	**O(h)**
for (int i = 1; i < ANDs.length; i++) {}	t - 1	**O((t−1)·(h+50))**
Total		**O(k+t*h+50t)**

Method: OR_Function

Code	Frequency	Total Complexity
String[] ORs = string.split(" OR ");	1	**O(k)**
if (this.found(ORs[0].toLowerCase().trim())) {}	1	**O(h)**
for (int i = 1; i < ORs.length; i++) {}	t- 1	**O((t−1)·(h+50))**
Total		**O(k+t·h+50t)**

Method: printDocment()

Code	Frequency	Total Complexity
if (DocumentIndex.empty())	1	**O(1)**
while (!this.DocumentIndex.last()) {}	n−1	**O(n−1)**
System.out.println(DocumentIndex.retrieve())	n	**O(n)**
Total		**O(n)**

Total inverted index:

Code	Time Complexiy
adddocument()	**O(n·c)**
found	**O(n·c)**
AND_OR_Function	**O(k+t·(h+50))**
AND_Function	**O(k+t·(h+50))**
OR_Function	**O(k+t·(h+50))**
printDocment	**O(n)**
Total	**O(k+t·h+50t+n·c)**

3.2 Class Inverted index BST:

The Inverted_Index_BST class is an implementation of an inverted index using a Binary Search Tree (BST). Data structures such as inverted indexes are commonly used in search engines to map terms (words) to documents

Code	Frequency	Total complexity
BST<String,Term> DocumentindexBST;	1	O (1)
public Inverted_Index_BST () { DocumentindexBST = new BST<>(); }	1	O (1)
public int size() { return DocumentindexBST.size(); }	1	O (1)

Method: size ()

Code	Frequency	Total complexity
public int size () {	1	O (1)
return DocumentindexBST.size();	1	O (1)

Method: adddocument(int docID, String word)

Code	Frequency	Total complexity
public boolean adddocument(int docID, String word) {	1	O (1)
if (DocumentindexBST.empty()) {	1	O (1)
term = new Term(); term.setVocabulary(new Vocabulary(word));	1	O(1)
term.add_documentID(docID);	1	O(1)
DocumentindexBST.insert(word, term);	1	O (log n)
if (DocumentindexBST.find(word)) {	1	O (log n)
term = DocumentindexBST.retrieve(); term.add_documentID(docID);	1	O (log n)
DocumentindexBST.update(term);	1	O(log n)

Method: found (String word)

Code	Frequency	Total complexity
public boolean found (String word) { return DocumentindexBST.find(word); }	1	O (log n)

Method: printDocument ()

Code	Frequency	Total complexity
public void printDocument() { DocumentindexBST.Traverse(); }	1	O(n)

Method: AND OR Function

Code	Frequency	Total complexity
if (! string.contains(" OR ") && ! string.contains(" AND "))	1	O(k)
string = string.toLowerCase().trim();	1	O(k)
if (this.found(string)) result = DocumentindexBST.retrieve().getDocs()	1	O(log n)
return result;	1	O(1)
if (string.contains(" OR ") && string.contains(" AND ")) {	1	O(k)
String[] AND_ORs = string.split(" OR ");	1	O(k)
result = AND_Function(AND_ORs[0]);	1	O(m·logn)
for (int j = 0 ; j < 50 ; j++) result [j] = result[j] || tempResult[j];	Loop through subqueries and combine results using "OR".	O(s·(m·logn+50))

Method: AND Function

Code	Frequency	Total complexity
String [] ANDs = string.split(" AND ");	1	O(k)
if (this.found(ANDs[0].toLowerCase().trim())) {	1	O(log n)
for (int j = 0 ; j < 50 ; j++) result [j] = result[j] && tempResult[j];	t	O(t·(logn+50))

Method: OR Function

Code	Frequency	Total complexity
String [] ORs = string.split(" OR ");	1	O(k)
if (this.found(ORs[0].toLowerCase().trim())) {	1	O(log n)
for (int j = 0 ; j < 50 ; j++) result [j] = result[j] || tempResult[j];	Loop through subqueries and combine results using "OR".	O(t·(logn+50))

Method	Total
size()	O (1)
adddocument()	O(logn)
found()	O(logn)
printDocument()	O(n)
AND_OR_Function()	O(k+s·m·logn)
AND_Function()	O(k+t·logn)
OR_Function()	O(k+t·logn)
O(total)=	O(d·w·logn+k+s·m·logn)
O(average)=	O(d·w·logn+n+q·k+s·q·m·logn+2·s·q·k+2·s·q·t·logn)

3.3 class Index:
An inverted index is a data structure that maps words to the documents where those words appear. This allows for efficient searching of text documents. Instead of scanning every document for a specific word, we can quickly look up the word in the index and retrieve the documents associated with it.

code	frequency	total
public class Index {	0	0
class Document {	0	0
int documentID;	1	1
LinkedList<String> wordList;	1	1
public Document() {	0	0
documentID = 0;	1	1
wordList = new LinkedList<String>();	1	1
}		
total		O(1)

Method:addWord

code	frequency	total
wordList.insert(word);	1	O(1)
total		O(1)

Method:hasWord

code	frequency	total
if (wordList.empty())	1	O(1)
wordList.findFirst();	1	O(1)
while (i < wordList.size) {	Loops over k words	O(k)
if (wordList.retrieve().compareTo(word) == 0)	executes up to k	O(k)
wordList.findNext();	executes up to k	O(k)
total		O(k)

Constructor Index()

code	frequency	total
documents = new Document[50];	1	O(1)
while (i < documents.length) {	Loops 50 times	O(n)
documents[i] = new Document();	Executes 50 times	O(n)
documents[i].documentID = i;	Executes 50 times	O(n)
total		O(n)

Method: addToDocument

code	frequency	total
documents[documentID].addWord(word);	1	O(1)
total		O(1)

Method: displayDocument

code	frequency	total
if (documents[documentID].wordList.empty())	1	O(1)
documents[documentID].wordList.findFirst();	1	O(1)
while (i < documents[documentID].wordList.size) {	K(words)	O(K)
System.out.print(documents[documentID].wordList.retrieve() + " ");	k	O(K)
documents[documentID].wordList.findNext();	k	O(K)
total		O(K)

Method: retrieveDocuments

code	frequency	total
while (i < matches.length) {	n	O(n)
matches[i] = false;	n	O(n)
while (i < matches.length) {	n	O(n)
if (documents[i].hasWord(word))	n	O(n*k)
matches[i] = true;	n	O(n)
total		O(n*k)

Methode AND_OR_Functio

code	frequency	total
if (!query.contains(" OR ") && !query.contains(" AND ")) {	1	O(1)
boolean[] results = retrieveDocuments(query.toLowerCase().trim());	1	O(n*k)
if (query.contains(" OR ") && query.contains(" AND ")) {	1	O(q)
boolean[] results = AND_Function(subQueries[0]);	1	O(h*n*k)
while (i < subQueries.length) { boolean[] subResults = AND_Function(subQueries[i]);	s-1 subQueries	O((s-1)*h*n*k)
while (j < 50) { results[j] = results[j] || subResults[j];		SubResults[j]
total		O(h*n*k *s)

Method: AND_Function

code	frequency	total
String[] andParts = query.split(" AND ");	1	O(h)
boolean[] results = retrieveDocuments(andParts[0].toLowerCase().trim());	1	O(n*k)
while (i < andParts.length) { boolean[] subResults = retrieveDocuments(andParts[i].toLowerCase().trim());	h-1	O((h-1)*n*k)
while (j < 50) { results[j] = results[j] && subResults[j];	n*h	O(n*k)
total		O(h*n*k)

Method: OR_Function

code	frequency	total
String[] orParts = query.split(" OR ");	1	O(h)
boolean[] results = retrieveDocuments(orParts[0].toLowerCase().trim());	1	O(n*k)
while (i < orParts.length) {boolean[] subResults = retrieveDocuments(orParts[i].toLowerCase().trim());	h-1	O((h-1)*n*k)
while (j < 50) { results[j] = results[j] || subResults[j];		subResults[j];
total		O(h*n*k)

Total:

method	total
Index()	O(n)
HasWord()	O(k)
addToDocument ()	O(1)
DisplayDocument()	O(n*k)
RetrieveDocuments()	O(n*k)
AND_OR_Function()	O(m*k)
AND_Function()	O(m*(n*k))
OR_Function()	O(m*(n*k))
total	O(m*(n*k))

Comparison

- **Class Index: O(m(n*k)) is the linear complexity of the simplest structure. It works well with tiny datasets but might not work well with big ones.**
- **Class Inverted Index: Performs better than Class Index when handling more frequent document updates, but is a little more complicated because of t*h and n*c.**
- **The best way to achieve scalability because operations for updates and searches grow logarithmically (log n). Large datasets benefit greatly from the BST's logarithmic performance, despite its higher constants.**

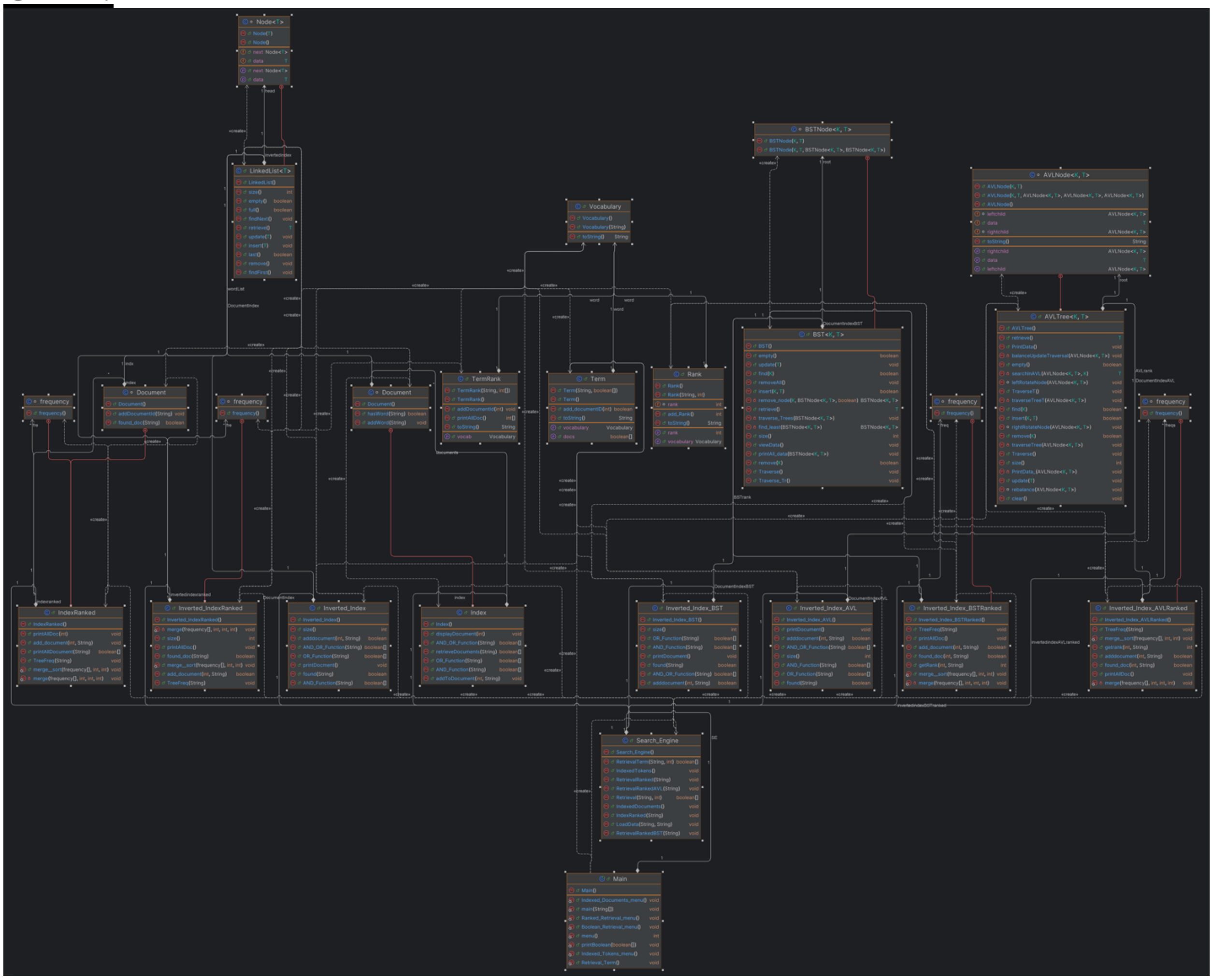
Best Choice

Class Inverted Index BST: is the best solution overall since it is more efficient for large datasets and frequent lookups or updates due to its logarithmic complexity O (log n) for operations.

Our github:

https://github.com/Ft0oo0n6/data-structure-.git

UML:



Node<T>
LinkedList<T>
BSTNode<K, T>
AVLNode<K, T>
Vocabulary
BST<K, T>
AVLTree<K, T>
frequency
Document
frequency
Document
TermRank
Term
Rank
frequency
frequency
IndexRanked
Inverted_IndexRanked
Inverted_Index
Index
Inverted_Index_BST
Inverted_Index_AVL
Inverted_Index_BSTRanked
Inverted_Index_AVLRanked
Search_Engine
Main

Sample run:

```
 ----jGRASP exec: java Main
tokens 1443
vocabs 640
1. Term Retrieval.
2. Boolean Retrieval.
3. Ranked Retrieval.
4. Indexed Documents.
5. Indexed Tokens.
6. Exist.
enter choice
1
------------- Retrieval Term -------------
1. index
2. inverted index
3. inverted index using BST
4. inverted index using AVL
enter your choice
1
Enter Term :
market
Result doc IDs: [ 0, 12, 41]
```

```
1. Term Retrieval.
2. Boolean Retrieval.
3. Ranked Retrieval.
4. Indexed Documents.
5. Indexed Tokens.
6. Exist.
enter choice
2
------------- Boolean Retrieval -------------
1. index
2. inverted index
3. inverted index using BST
4. inverted index using AVL
enter your choice
1
Enter boolean term ( AND / OR) :
market AND sports
Q#: market AND sports
Result doc IDs: Boolean_Retrieval using index list
[ 41]
```

```
1. Term Retrieval.
2. Boolean Retrieval.
3. Ranked Retrieval.
4. Indexed Documents.
5. Indexed Tokens.
6. Exist.
enter choice
3
----- Ranked Retrieval -----
1. index
2. inverted index
3. inverted index using BST
4. inverted index using AVL
enter your choice
1
enter term: market
## Q: market
DocIDt   Score
get ranked from index list

 DocID: t    Score:
0     1
12    1
41    1
```

```
1. Term Retrieval.
2. Boolean Retrieval.
3. Ranked Retrieval.
4. Indexed Documents.
5. Indexed Tokens.
6. Exist.
enter choice
4
-----Indexed Documents -----
Indexed Documents
All Documents with the number of words in them
Document# 0  with size(23)
Document# 1  with size(25)
Document# 2  with size(22)
Document# 3  with size(22)
Document# 4  with size(24)
Document# 5  with size(20)
Document# 6  with size(21)
Document# 7  with size(20)
Document# 8  with size(19)
Document# 9  with size(19)
Document# 10  with size(18)
```

```
1. Term Retrieval.
2. Boolean Retrieval.
3. Ranked Retrieval.
4. Indexed Documents.
5. Indexed Tokens.
6. Exist.
enter choice
5
----- Indexed Tokens -----
tokens
All tokens with the documents appear in it
abilities
the document it's: 33
accelerating
the document it's: 28
accommodate
the document it's: 48
accuracy
the document it's: 49
accurate
the document it's: 30
activities
the document it's: 34
activity
the document it's: 46
adapt
```