Exercise 4.1

N = no of postings for the term t = 48 million
G = granularity = number of postings per synchronization point

We need to minimize disk I/O which be contributed by synchronization points and the block B that is identified by binary search on synchronization point

Number of synchronization points = N/G
Number of postings in block B = G

We need to minimize the bytes that would be read from the disk
(N/G + G)*6 (Each byte is 6 bytes)

Y = (N/G + G) * 6
dY/dG = 0 (for what values of G would Y be minimum)
6 (-N/G^2 + 1) = 0
N = G^2
G = sqrt(N) = sqrt(48000000) = 6928 (approx)

The optimal granularity would be 6928.

Number of synchronization points (should be an integer) = (N-N%G) /G + 1 = 6928 + 1 = 6929
// 1 is for N%G = 2816
Number of postings per block= 6928
Number of postings for last block = 2816

The number of bytes read would be (6929+6928)*6 which is **83142**
In case last block is read it would be (6929 + 2816)*6 which is **58470**

Exercise 4.3

First place

The createIndexPartition code which is used to create on-disk inverted file , first sorts the in-memory dictionary in lexicographical order. General-purpose sorting algorithms are bound by Ω(nlog(n)), n is the number of terms in the file. This is one place where there is a hidden logarithmic factor.

Line 17 in code
Sort in-memory dictionary entries in lexicographical order

Second place

In merging the index partitions, we need to merge all the partitions. If the number of index partitions is large (more than 10), then the algorithm would be improved by arranging the index partitions in a priority queue(e.g. a heap), which would be ordered according to the next term in the respective partition. Using this information, the inverted index can be built in a sorted manner by just accessing the root of heap. After the term has been inserted into the final index file, we need to remove it and rearrange the heap. In case of large number of partitions, linear scan would not be used.

Updating the heap and rearranging would take O(logn) time for each term.

Exercise 4.5
Solution :-

Burst Trie (Heinz et al., 2002) proposed by STEFFEN HEINZ, JUSTIN ZOBEL, and HUGH E. WILLIAMS which achieves a single-term lookup performance close to a hash table but— unlike a hash table— can also be used to resolve prefix queries. This is experimental data structure.

A **burst trie** is an in-memory data structure, designed for sets of records that each have a unique string that identifies the record and acts as a key. Formally, a string s with length n consists of a series of symbols or characters ci for i = 0,...,n, chosen from an alphabet A of size |A|. We assume that |A| is small, typically no greater than 256.
A burst trie consists of three distinct components, a set of records, a set of containers, and an access trie:
Records:
A record contains a string; information as required by the application using the burst trie (that is, for information such as statistics or word locations); and pointers as required to maintain the container holding the record. Each string is unique.
Containers:
A container is a small set of records, maintained as a simple data structure such as a list or a BST. For a container at depth k in a burst trie (depth is discussed below), all strings have length at least k, and the first k characters of all strings are identical. It is not necessary to store these first k characters. Each container also has a header, for storing the statistics used by heuristics for bursting.

Access trie:
An access trie is a trie whose leaves are containers. Each node consists of an array p, of length |A|, of pointers, each of which may point to either a trie node or a container, and a single empty-string pointer to a record. The |A| array locations are indexed by the characters c ∈ A. The remaining pointer is indexed by the empty string.
The depth of the root is defined to be 1. Leaves are at varying depths.

The use of a BST as a container enables the retrieval of records in sort order. An inorder traversal of the burst trie starts at the root. Although the records within the container store only suffixes of the original strings, they can be reconstructed by keeping track of the traversal path.

How Burst Trie works :-

A Burst Trie uses containers to store keys/values before creating branches.
When the containers are full, they "burst" and are turned into branches.
The benefit is that a more efficient data structure for small sets of key ,g s/values can be used in the container, making it faster than a conventional tree.
This structure is a collection of small data structures, which , we call containers ,that are accessed via a conventional trie.
Searching involves using the first few characters of a query string to identify a particular container then using the string to identify a particular container, then using the remainder of the query string to find a record in the container.

A container can be any data structure that is reasonably efficient for small sets of strings, such as a list or a binary search tree.

Search:

Searching a burst trie has two distinct stages, as follows.
Input is a query string c1 ··· cn of n characters.
Output is a pointer to a record, or null in the case of unsuccessful search.
1.The access trie is traversed according to the leading characters in the query string. Initially the current object is the root of the access trie and the current depth i is 1.
While the current object is a trie node t of depth i ≤ n,
(a)Update the current object to be the node or container pointed to by the cith element of t's array p, and
(b) Increment i.
If the current object is a trie node t of depth i = n + 1, the current object is the object pointed to by the empty-string pointer, which for simplicity can be regarded as a container of either zero or one records.
else if (current object==null)
search terminates since string is not present in the burst trie.
2. The current object is a valid container. If i ≤ n, use the remaining characters ci ··· cn to search the container, returning the record if found or otherwise returning null.
Otherwise, the current object is a container of zero or one records that is pointed to by the emptystring pointer (since the entire string was consumed by traversal of the trie), which should be returned.

Note that in many languages the most common words are typically short. Thus, these terms are typically stored at an empty-string pointer, and are found after simply following a small number of access trie pointers with no search in a container at all. This ease of access to common terms is one of the major reasons the burst trie is so fast.

Insertion:

A burst trie can grow in two distinct ways. First is insertion, when a record is added to a (possibly initially empty) container. This occurs when a new string is encountered. Second is bursting, when a container at depth k is replaced by an access trie node pointing to a set of containers, each of which is at depth k + 1. This occurs when the container is deemed, by heuristics as discussed later, to be inefficient. Bursting is considered below. We now consider insertion; input is a new string c1 ··· cn of length n; the outcome is addition of a record to a burst trie.

1. Stage 1 of the search algorithm above is used to identify the container in which the record should be inserted. This container, which can be empty, is at depth k
For the special case of arriving at a trie node at depth k = n + 1, the container is under the empty-string pointer.
2. If k ≤ n, the standard insertion algorithm for the container data structure is used to add the record to the container, using only the suffix characters ck+1 ··· cn. Otherwise, the record is added under the empty-string pointer.

Bursting:

Bursting is the process of replacing a container at depth k by a trie node and a set of new containers at depth k + 1, which between them contain all the records in the original container.

When to do bursting has many options. We are considering here as size of the container as the reference value to decide if the bursting is required. The limit of container is set to 10. If the size of container exceeds this limit, it should burst and convert into Access Trie.
The outcome is replacement of a container with a new leaf node in the access trie and a set of child containers.

1. A new access trie node b is created. Each pointer is set to an empty container.
2. For each record r in the original container, which has string c^r k+1 ··· c^r n,

(a) The leading character cr k+1 is removed from the string, and
(b) The record is added to the container indicated by the cr k+1th pointer of b, using the string cr k+2 ··· cr n. If the record was the empty string, it is added under b's emptystring pointer.

3. The original container is discarded.

Pseudocode for the Burst Trie:-

Search:

Searching a burst trie has two distinct stages, as follows.
Input is a query string c1 ··· cn of n characters.
Output is a pointer to a record, or null in the case of unsuccessful search.

```
public burstTrieComponent Search(String query, AccessTrie trie_root){
Current_object=trie_root;
Current_depth=1;
While(isAccessTrie(current_object)&&depth(current_object)<=n){
        Current_object_position=getTriePosition(current_object)  //t[c[i]]
        i=i+1;
        if(isAccessTrie(current_object)&&depth(current_object)==n+1)
                current_object=getContainer(current_object);
        elseif(current_object==null)
                return;
}
if(isValidContainer(current_object)&&i<=n){
                for (characters i to n)
//This function will search the characters from i to n and return the position of the last character in the BST. If fails it retuns null.
current_position=searchBST(current_object);
return current_position
}
```

```
else{
        //the current_object belongs to the container of zero or one character and should just return the position.It will simply be trie search character by character.
                current_position=(get_Position(current_object));
                return current_position;
}
}
```

Insertion:

We now consider insertion; input is a new string c1 ··· cn of length n; the outcome is addition of a record to a burst trie.

```
Public void burstTrieAdd(burstTrieComponent component, String query){

//If the given query is part of container the container gets updated
if (isValidContainer(component)){
        for(i=1 to n){
leadingchar=query.charAt(i)
                ((Container) component,BSTinserRecord(new Record(i)));
                }
        }
        //otherwise if its part of AccessTrie, Trie gets updated with the new characters by removing the leading character and going for suffix string.
        else if(component isInstanceOf AccessTrie){
                for(i=1 to n){
                ((AccessTrie)component.AccessTrieinsertRecord(new Record(i)))
}
                }
}
```

Bursting:

Bursting is the process of replacing a container at depth k by a trie node and a set of new containers at depth k + 1, which between them contain all the records in the original container.

Input is a burst trie.

The outcome is replacement of a container with a new leaf node in the access trie and a set of child containers.

```
Public void burstBurstTrie(burstTrieComponent component ,String query){
//first check if the burst is applicable to do on container.
//Here, I am considering Burst heuristic as size of container if the size of container exceeds the limit(which is currently set to 10) it bursts and then get converted into Trie.
int size=getSizeOf(component.Container);
        if(size<limit)
```

```
                return;
        else{
                AccessTrie trie=new AccessTrie();
                current_depth=getDepth(component);
                query=query.subString(1);
burstTrieAdd(component,query);
//update the emptyString pointer for the current_depth of component.record.
//clear the existing container since Access trie has been build for the same.
Clear(component.Container);

        }
}
```

Conclusion: -

- Burst Tries are highly efficient for managing large sets of strings in memory.
- Use of Containers allow dramatic space reductions with no impact on efficiency.
- Experiments show that Burst Tries are:
    - Faster than compact tries, using 1/6 of the space
    - More compact than binary trees or splay trees and are over two times faster
    - **Close to hash tables in efficiency yet keep the data in sort order.**.

The Burst Trie is dramatically more efficient than any previously known structure for the task of managing sorted strings.

References: -

1.https://www.researchgate.net/publication/220515693_Burst_Tries_A_Fast_Efficient_Data_Structure_for_String_Key

Burst Tries: A Fast, Efficient Data Structure for String Keys Hienz, Zbloe , Willi ShlfCtSid Williams; School of Computer Science and Information Technology, RMIT University (Australia).

2. Information Retrieval: Implementing and Evaluating Search Engines. Buttcher, Clarke, and Cormack.

3.http://www.cs.uvm.edu/~xwu/wie/CourseSlides/Schips-BurstTries.pdf

Excercise 4.5

Approach 2 :-

To obtain the better performance for both single-term looks up and multi term look up(needed for prefix queries), we can use combined data structure where the single term look up are done using hash based implementation whereas multi-term look up can be done by sort-based dictionary.

Below is the high level approach for doing these functions :-

Index Construction using Hash based Dictionary for single-term look up: -

```
buildIndex (inputTokenizer)
1 position ← 0
2 while inputTokenizer.hasNext() do
3 T ← inputTokenizer.getNext()
4 obtain dictionary entry for T; create new entry if necessary
5 append new posting position to T's postings list
6 position ← position + 1
7. for each term T in the dictionary do
8. write T's postings list to disk
9. write the dictionary to disk
10. return
```

Here dictionary is constructed using hash function.
It will first calculate the hash value for given term and if it finds the collision it will update the existing posting list. Otherwise it inserts the new entry in the hash-map.

Index construction using Sort based Dictionary for multi-term look up: -

```
buildIndex sortBased (inputTokenizer)
1 position ← 0
2 while inputTokenizer.hasNext() do
3 T ← inputTokenizer.getNext()
4 obtain dictionary entry for T; create new entry if necessary
5 termID ← unique term ID of T
6 write record Rposition ≡ (termID, position) to disk
7 position ← position + 1
8 tokenCount ← position
9 sort R0 . . .RtokenCount−1 by first component; break ties by second component
10 perform a sequential scan of R0 . . .RtokenCount−1, creating the final index
11 return
```

Function to call either of these functions as per the need :- (single-term query will call hash Based Dictionary while multi-term query will call sort based dictionary.)

```
buildIndex_BasedOn_MemoryType(inputTokenizer)
1.If the Memory type is InMemory and have single term look up :-
        Call function buildIndex (inputTokenizer)
2.else
        Call function buildIndex sortBased (inputTokenizer)
3.exit.
```