

# Find the k most frequent words from a file

Given a book of words. Assume you have enough main memory to accommodate all words. design a data structure to find top K maximum occurring words. The data structure should be dynamic so that new words can be added.

A simple solution is to **use Hashing**. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts.

We can **use Trie and Min Heap** to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time(Use of Min Heap is same as we used it to find k largest elements in [this](#) post).

Trie and Min Heap are linked with each other by storing an additional field in Trie 'indexMinHeap' and a pointer 'trNode' in Min Heap. The value of 'indexMinHeap' is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, 'indexMinHeap' contains, index of the word in Min Heap. The pointer 'trNode' in Min Heap points to the leaf node corresponding to the word in Trie.

Following is the complete process to print k most frequent words from a file.

Read all words one by one. For every word, insert it into Trie. Increase the counter of the word, if already exists. Now, we need to insert this word in min heap also. For insertion in min heap, 3 cases arise:

1. The word is already present. We just increase the corresponding frequency value in min heap and call minHeapify() for the index obtained by "indexMinHeap" field in Trie. When the min heap nodes are being swapped, we change the corresponding minHeapIndex in the Trie. Remember each node of the min heap is also having pointer to Trie leaf node.
2. The minHeap is not full. we will insert the new word into min heap & update the root node in the min heap node & min heap index in Trie leaf node. Now, call buildMinHeap().
3. The min heap is full. Two sub-cases arise.
  - ...3.1 The frequency of the new word inserted is less than the frequency of the word stored in the head of min heap. Do nothing.
  - ...3.2 The frequency of the new word inserted is greater than the frequency of the word stored in the head of min heap. Replace & update the fields. Make sure to update the corresponding min heap index of the "word to be replaced" in Trie with -1 as the word is no longer in min heap.
4. Finally, Min Heap will have the k most frequent words of all words present in given file. So we just need to print all words present in Min Heap.

```
// A program to find k most frequent words in a file
#include <stdio.h>
#include <string.h>
#include <ctype.h>

# define MAX_CHARS 26
# define MAX_WORD_SIZE 30

// A Trie node
struct TrieNode
{
    bool isEnd; // indicates end of word
    unsigned frequency; // the number of occurrences of a word
    int indexMinHeap; // the index of the word in minHeap
    TrieNode* child[MAX_CHARS]; // represents 26 slots each for 'a' to 'z'.
};

// A Min Heap node
struct MinHeapNode
{
    TrieNode* root; // indicates the leaf node of TRIE
    unsigned frequency; // number of occurrences
    char* word; // the actual word stored
};

// A Min Heap
struct MinHeap
{
    unsigned capacity; // the total size a min heap
    int count; // indicates the number of slots filled.
    MinHeapNode* array; // represents the collection of minHeapNodes
};

// A utility function to create a new Trie node
TrieNode* newTrieNode()
{
    // Allocate memory for Trie Node
    TrieNode* trieNode = new TrieNode;

    // Initialize values for new node
    trieNode->isEnd = 0;
    trieNode->frequency = 0;
    trieNode->indexMinHeap = -1;
    for( int i = 0; i < MAX_CHARS; ++i )
        trieNode->child[i] = NULL;
}
```

```

    return trieNode;
}

// A utility function to create a Min Heap of given capacity
MinHeap* createMinHeap( int capacity )
{
    MinHeap* minHeap = new MinHeap;

    minHeap->capacity = capacity;
    minHeap->count = 0;

    // Allocate memory for array of min heap nodes
    minHeap->array = new MinHeapNode [ minHeap->capacity ];

    return minHeap;
}

// A utility function to swap two min heap nodes. This function
// is needed in minHeapify
void swapMinHeapNodes ( MinHeapNode* a, MinHeapNode* b )
{
    MinHeapNode temp = *a;
    *a = *b;
    *b = temp;
}

// This is the standard minHeapify function. It does one thing extra.
// It updates the minHeapIndex in Trie when two nodes are swapped in
// in min heap
void minHeapify( MinHeap* minHeap, int idx )
{
    int left, right, smallest;

    left = 2 * idx + 1;
    right = 2 * idx + 2;
    smallest = idx;
    if ( left < minHeap->count &&
        minHeap->array[ left ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = left;

    if ( right < minHeap->count &&
        minHeap->array[ right ]. frequency <
        minHeap->array[ smallest ]. frequency
    )
        smallest = right;

    if( smallest != idx )
    {
        // Update the corresponding index in Trie node.
        minHeap->array[ smallest ]. root->indexMinHeap = idx;
        minHeap->array[ idx ]. root->indexMinHeap = smallest;

        // Swap nodes in min heap
        swapMinHeapNodes ( &minHeap->array[ smallest ], &minHeap->array[ idx ] );

        minHeapify( minHeap, smallest );
    }
}

// A standard function to build a heap
void buildMinHeap( MinHeap* minHeap )
{
    int n, i;
    n = minHeap->count - 1;

    for( i = ( n - 1 ) / 2; i >= 0; --i )
        minHeapify( minHeap, i );
}

// Inserts a word to heap, the function handles the 3 cases explained above
void insertInMinHeap( MinHeap* minHeap, TrieNode** root, const char* word )
{
    // Case 1: the word is already present in minHeap
    if( (*root)->indexMinHeap != -1 )
    {
        ++( minHeap->array[ (*root)->indexMinHeap ]. frequency );

        // percolate down
        minHeapify( minHeap, (*root)->indexMinHeap );
    }

    // Case 2: Word is not present and heap is not full
    else if( minHeap->count < minHeap->capacity )
    {
        int count = minHeap->count;
        minHeap->array[ count ]. frequency = (*root)->frequency;
        minHeap->array[ count ]. word = new char [ strlen( word ) + 1 ];
        strcpy( minHeap->array[ count ]. word, word );

        minHeap->array[ count ]. root = *root;
        (*root)->indexMinHeap = minHeap->count;
    }
}

```

```

    ++( minHeap->count );
    buildMinHeap( minHeap );
}

// Case 3: Word is not present and heap is full. And frequency of word
// is more than root. The root is the least frequent word in heap,
// replace root with new word
else if ( (*root)->frequency > minHeap->array[0]. frequency )
{
    minHeap->array[ 0 ]. root->indexMinHeap = -1;
    minHeap->array[ 0 ]. root = *root;
    minHeap->array[ 0 ]. root->indexMinHeap = 0;
    minHeap->array[ 0 ]. frequency = (*root)->frequency;

    // delete previously allocated memory and
    delete [] minHeap->array[ 0 ]. word;
    minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];
    strcpy( minHeap->array[ 0 ]. word, word );

    minHeapify ( minHeap, 0 );
}
}

// Inserts a new word to both Trie and Heap
void insertUtil ( TrieNode** root, MinHeap* minHeap,
                 const char* word, const char* dupWord )
{
    // Base Case
    if ( *root == NULL )
        *root = newTrieNode();

    // There are still more characters in word
    if ( *word != '\0' )
        insertUtil ( &((*root)->child[ tolower( *word ) - 97 ]),
                    minHeap, word + 1, dupWord );
    else // The complete word is processed
    {
        // word is already present, increase the frequency
        if ( (*root)->isEnd )
            ++( (*root)->frequency );
        else
        {
            (*root)->isEnd = 1;
            (*root)->frequency = 1;
        }

        // Insert in min heap also
        insertInMinHeap( minHeap, root, dupWord );
    }
}

// add a word to Trie & min heap. A wrapper over the insertUtil
void insertTrieAndHeap(const char *word, TrieNode** root, MinHeap* minHeap)
{
    insertUtil( root, minHeap, word, word );
}

// A utility function to show results, The min heap
// contains k most frequent words so far, at any time
void displayMinHeap( MinHeap* minHeap )
{
    int i;

    // print top K word with frequency
    for( i = 0; i < minHeap->count; ++i )
    {
        printf( "%s : %d\n", minHeap->array[i].word,
                minHeap->array[i].frequency );
    }
}

// The main function that takes a file as input, add words to heap
// and Trie, finally shows result from heap
void printKMostFreq( FILE* fp, int k )
{
    // Create a Min Heap of Size k
    MinHeap* minHeap = createMinHeap( k );

    // Create an empty Trie
    TrieNode* root = NULL;

    // A buffer to store one word at a time
    char buffer[MAX_WORD_SIZE];

    // Read words one by one from file. Insert the word in Trie and Min Heap
    while( fscanf( fp, "%s", buffer ) != EOF )
        insertTrieAndHeap(buffer, &root, minHeap);

    // The Min Heap will have the k most frequent words, so print Min Heap nodes
    displayMinHeap( minHeap );
}

```

```
// Driver program to test above functions
int main()
{
    int k = 5;
    FILE *fp = fopen ("file.txt", "r");
    if (fp == NULL)
        printf ("File doesn't exist ");
    else
        printKMostFreq (fp, k);
    return 0;
}
```

Output:

```
your : 3
well : 3
and : 4
to : 4
Geeks : 6
```

The above output is for a file with following content.

Welcome to the world of Geeks  
This portal has been created to provide well written well thought and well explained solutions for selected questions If you like Geeks for Geeks and would like to contribute here is your chance You can write article and mail your article to contribute at geeksforgeeks org See your article appearing on the Geeks for Geeks main page and help thousands of other Geeks