Document Ranking Application Updated Report

Participants:

Junaid Ahmed Laskar.

CSM21021

&

Jasraj Choudhury

CSM21028

Under the guidance of:

Dr. Arindam Karmakar.

Specification:

Aim: In this project we are going to create an application to rank document according to search keyword using a priority queue to rank the document with file handling in C++ programming language.

Steps:

(i) Open folder and read the files.
(ii) Ask the user for the keyword according to which the document will be ranked.
(iii) Creating priority queue using heap with keywords to rank the documents.
(iv) Using priority queue data structure we will extract relevant documents with the keyword.
(v) We will discuss the results and advantages of using priority queue in such a problem. Tool: C++ Programming language, File handling.

Feasibility: feasible to create the application within one month with C++ and required file handling tool.

Note: This application has been tested on mingw g++ on windows platform with directory retrieve using windowsHand (in windows.h). For user of unix/linux based OS, the directory retrieve and file handling approach must be changed according OS on which the application is to be run. The Algorithm and approach remain same.

Executive Summary:

In this project, we created a program detecting in which documents a given keyword is most frequently found. We have followed a simple procedure. We take a word from the user. We asked the user how many document they would like to see and listed most relevant documents based on the user's choices. To find in which documents the entered keyword appears most frequently, we have used priority queue structure, we preferred binomial max heap instead of normal max heap. Since there will be multiple document whose keyword will be ranked, normal heap will not be sufficient.

In binomial heap structure we keep how many times a word occurs in a particular document (say doc), and document itself. A document structure will consist of name of document and text inside it. We preferred to use maximum priority queues to operate this process. Since in a maximum priority queue a document with higher number of keywords is served before a document with lower number of keywords.

Creating Document Structure:

First we create a document structure to hold our text document and its name and then we will create Binomial heap which will hold those document according to rank o keyword.

```
#include"getsystemheader.hpp"
#ifndef DOCUMENT_H_INCLUDED
#define DOCUMENT_H_INCLUDED
struct document{
    char * name;
    char *text;
    ~document();
}typedef document;
#endif // DOCUMENT_H_INCLUDED
```

document.cpp

```
#include"document.hpp"
document::~document(){
    delete [] name;
    delete [] text;
}
```

Getting the input from the User:

We first ask user to enter a keywords that will be searched in the documents and among how many document user want wants to find relative ranking.

```
//Getting the keyword from useraccording to which document will be ranked
char * keyword = new char[150];
cout<<"[PLEASE ENTER THE KEYWORD]:"<<endl;
output<<"[KEYWORD ACCORDING TO WHICH DOCUMENT IS RANKED]:"<<endl;
cout<<"=>";
output<<"=>";
cin>>keyword;
output<<keyword<<endl;
toLowerCase(keyword);
cout<<endl<<endl;
output<<endl<<endl;
```

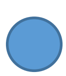
```
int numberOfRelevant;
cout<<"Now,Enter how many document do you need?"<<endl;
cout<<"=>";
cin>>numberOfRelevant;
output<<"[NUMBER OF MOST RELEVANT DOCUMENT]:"<<endl;
output<<"=> "<<numberOfRelevant<<endl<<endl;
```

After taking the keyword from the user we construct a maximum binomial heap according to how many times the keyword is appeared in each document.

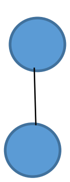
Binomial Heap:

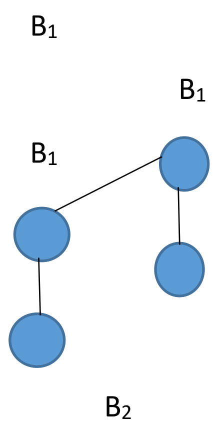
Before discussing binomial heap, let us discuss binomial tree. A binomial tree B_k is an ordered tree defined recursively.

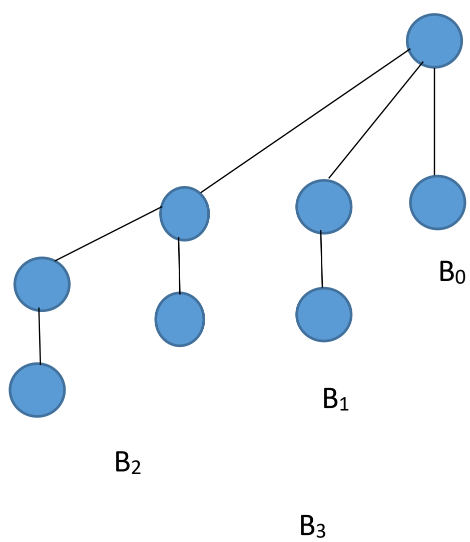
The binomial tree B_0 consist of single node. The binomial tree B_1 is consist two binomial tree of B_0.Similarly, the binomial tree of B_k consists of two binomial trees B_{k-1} that are linked together at k^{th} level. Example:

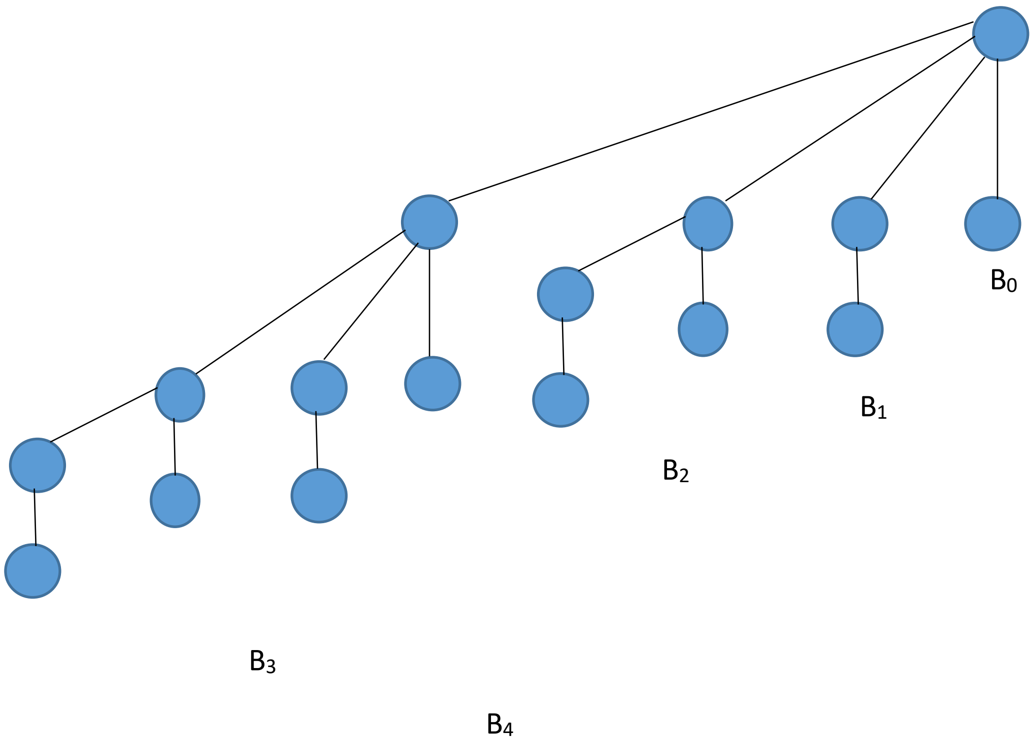


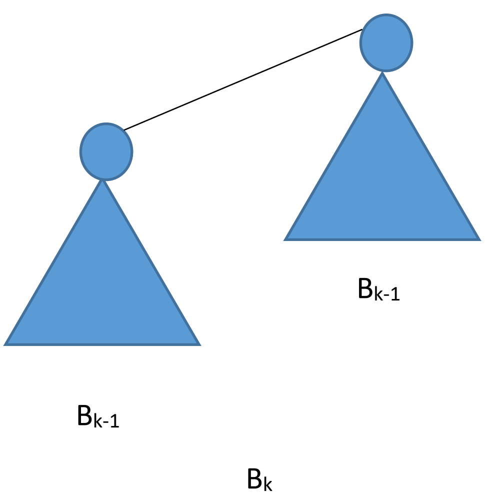
B_0




B1
B1
B1
B2


B0
B1
B2
B3


B0
B1
B2
B3
B4


Bk-1
Bk-1
Bk

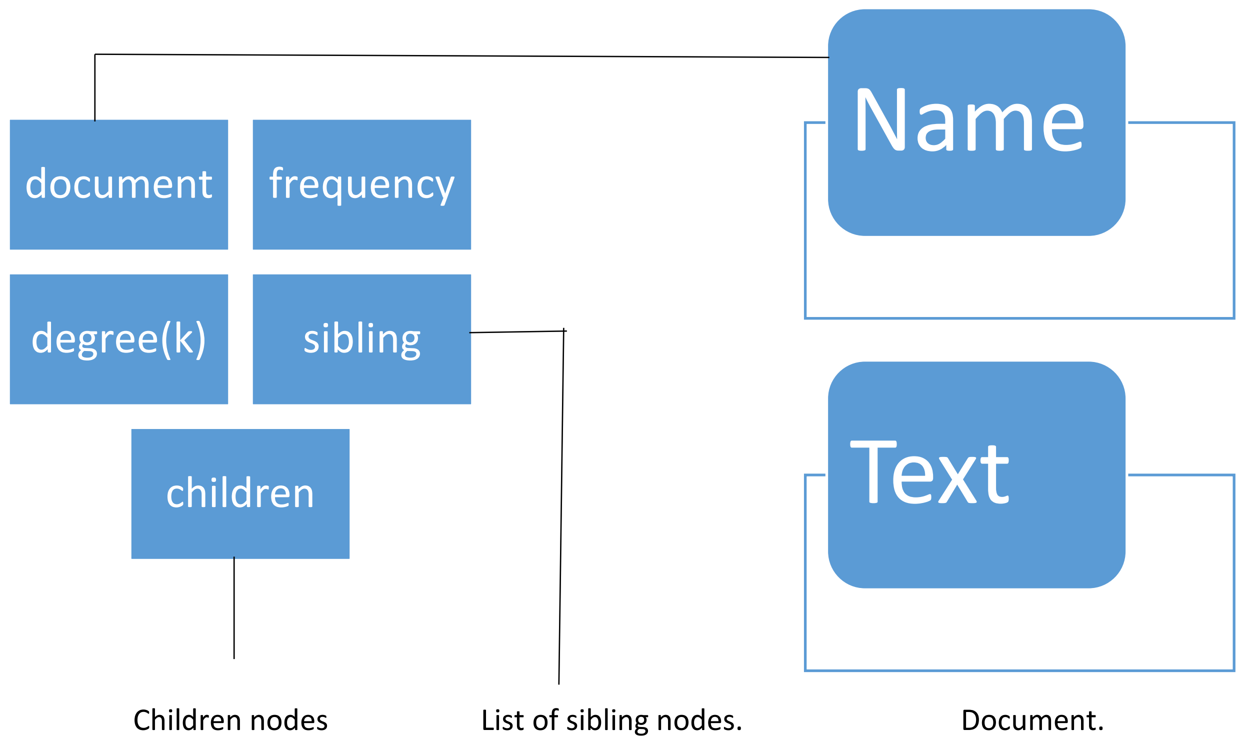
Points:

1. There are 2^k nodes in binomial tree of k-level. Example in B_4, there are $2^4 = 16$ nodes.
2. The height of the tree is K.
3. There are exactly ${}^kC_i$ nodes at depth i for i = 0,1,2,….,k. Example, At root of B_4 there is ${}^4C_0 = 1$ nodes at root. At depth 3 in B_4, there are ${}^4C_3 = 4$ nodes.
4. The root has degree k which is greater than that of any other node.
5. If i is the children of the root are numbered from left to right k-1, k-2,….,0 child I is the root of a subtree B_i.

Binomial heap heap H is a set of Binomial trees that satisfies the properties:

1. Each binomial tree in H obeys minimum/maximum property.
2. For a non-negative integer k, there is at most one binomial one binomial tree B_k in H.
3. An n-node binomial heap H consist of at most ceil($\log_2 n$)+1 binomial trees.
4. Because binary representation of n has ceil($\log_2 n$)+1 bits.

Memory representation of binomial tree nodes in our application is as follows:



Enqueue And Dequeue Operation In Binomial Heap:

To perform enqueue and dequeue operation in binomial heap we need to perform the union operation first. Following is an algorithm for union operation.

Algorithm:

1. Merge the root list of binomial maximum heaps H_1 and H_2 into a single linked list that is stored in non-decreasing order of their degree.
2. Link the roots of equal degree until at most one root remains of each degree.
    i. If degree[x] != degree[next-x] or degree[x] = degree[next-x] = degree[sibling-next-x], then move the pointer one position right.
    ii. If degree[x] = degree[next-x] != degree[sibling-next-x] then:
        a. If frequency[x] > frequency[next-x] then make next-x as child of x.
        b. If frequency[x] < frequency[next-x], then make x as leftmost child of [next-x]

Time Complexity of Union Operation: Let us suppose two heap H_1 : B_0->B_1->B_2-> ….

And H_2 : B_0->B_1->B_2-> …. Both takes $\log_2 n_1$ and $\log_2 n_2$ times. Merging them will take $\log_2 n_1$ + $\log_2 n_2 = \log_2 n$ times, where $n= n_1 n_2$. Again each tree in heap will be arranged once and there will be $\log_2 n$ binomial tree in n-node binomial heap. Therefore arranging their time is $\log_2 n + \log_2 n$ approximately equals to $\log_2 n$ time. i.e., $O(\log_2 n)$.

Enqueue Algorithm:

1. Create a binomial heap H containing new element.
2. Apply union of two binomial maximum heap H and original binomial heap H'.

Time Complexity: constant time to create a new binomial heap and $\log_2 n$ time to union operation. Total time complexity is $O(\log_2 n)$.

Now, before discussing dequeuer operation let us discuss extract maximum according to which dequeue operation will be done.

It is generally a linear search over root list of all binomial tree. And there are $\log_2 n$ binomial trees in a binomial heap. Therefore time complexity of it is $O(\log_2 n)$.

Dequeue Algorithm:

1. Set the element to be deleted as large positive number(say infinity).
2. Compare it with its parent values and replace it when it is greater than its parent. It will eventually be at root of its tree, since it is largest.
3. Delete that maximum from the root list.
4. Arrange children of that node in reverse list order. It will create another heap H'.

5. Apply union operation among original heap H and newly created heap H’.

Time Complexity: Increasing max node to infinitely large take $O(\log_2 n)$ time, constructing new heap take $O(\log_2 n)$ time and union operation takes $O(\log_2 n)$ times.

Total time complexity $\log_2 n+\log_2 n+\log_2 n = \log_2 m$ where $m = n^3$. That is we can say time complexity = $O(\log_2 m)$ or $O(\log_2 n)$.

```
#include"document.hpp"
#ifndef BINOMIALHEAP_H_INCLUDED
#define BINOMIALHEAP_H_INCLUDED
struct BinomialTree{
        document * doc; //document
        int frequency; //frequency will be key according to which we will extract maximum using max heap
        int k; //degree of nodes
        BinomialTree * children; //childs
        BinomialTree * sibling;
        ~BinomialTree();
}typedef BinomialTree;
BinomialTree * createBinomialTree(int,document *);
void attachTrees(BinomialTree *,BinomialTree *);
BinomialTree * mergeSortRoot(BinomialTree *, BinomialTree *);
BinomialTree * reverseChildrenNode(BinomialTree *, BinomialTree *);
void deleteDepthWise(BinomialTree *);
class BinomialHeap{
    private:
        BinomialTree * head;
        int size;
    public:
        BinomialHeap(BinomialTree * = NULL,int = 0);
        BinomialTree * getHead()const{return head;} //accessor
        void setHead(BinomialTree * value = NULL){head = value;} //mutator
        void setSize(int value = 0){size = value;} //mutator
        int getSize()const{return size;} //accessor
        void unionTree(BinomialHeap *);
        void enqueue(int,document *);
        int getMax();
        BinomialTree * dequeue();
        //destructor
        ~BinomialHeap();
};

#endif // BINOMIALHEAP_H_INCLUDED
```

binomialheap.cpp

```cpp
#include"binomialheap.hpp"
BinomialTree::~BinomialTree(){
    delete doc;
    frequency = 0;
    k = 0;
    children = NULL;
    sibling = NULL;
}
BinomialTree * createBinomialTree(int freq,document * doc){
    BinomialTree * temp = new BinomialTree;
    temp->children = NULL;
    temp->sibling = NULL;
    temp->k = 0;
    temp->frequency = freq;
    temp->doc = doc;
    return temp;
}
void attachTrees(BinomialTree * first,BinomialTree * second){//O(1)
    //We attach the nodes having same degree. It is easy
    //second become child of current or first tree
    second->sibling = first->children;
    first->children = second;
    first->k++;
}
```

```cpp
BinomialTree * mergeSortRoot(BinomialTree * first, BinomialTree * second){
    BinomialTree * result = createBinomialTree(0,NULL); //dummy node
    BinomialTree * tail = result;
    //creating root list of all binomial tree in non-decreasing order of k
    while(first != NULL || second != NULL){
        if(first == NULL){
            tail->sibling = second;
            break;
        }
        else if(second == NULL){
            tail->sibling = first;
            break;
        }
        else if(second->k < first->k){
            tail->sibling = second;
            second = second->sibling;
        }
        else{
            tail->sibling = first;
            first = first->sibling;
        }
        //We insert the node having smaller degree
        //tail helps us to insert node woth O(1) time
        tail = tail->sibling;
    }
    return result->sibling; //returning nodes with original document avoiding first dummy node we created
    //result was temporary nodes;
}
BinomialTree * reverseChildrenNode(BinomialTree * curr, BinomialTree * prev){
    if(curr != NULL){
        BinomialTree * temp = curr->sibling;
        curr->sibling = prev;
        return reverseChildrenNode(temp,curr);
    }
    return prev;
    //Reversing the children is important because each binomial tree has decreasing order
    //children. If we delete max element we need reverse list of its children so that we
    //union the two heap
}
```

```cpp
void deleteDepthWise(BinomialTree * node){
    if(node == NULL) return;
    while(node->children != NULL){
        deleteDepthWise(node->children);
    }
    while(node->sibling != NULL){
        deleteDepthWise(node->sibling);
    }
    delete node;
}
BinomialHeap::BinomialHeap(BinomialTree * bt,int sz):head(bt){
    size = sz;
}
BinomialHeap::~BinomialHeap(){
    deleteDepthWise(head);
    size = 0;
}
```

```cpp
void BinomialHeap::unionTree(BinomialHeap * h){
    if(h->getHead() == NULL){return;} //If heap2 is empty then return heap1
    this->size += h->getSize(); //In our implementation we want to keep the size of the heap
    BinomialTree * uniqueKs = mergeSortRoot(this->head,h->getHead());
    //Resulting tree node after merging forests. It is non decreasing order
    //but there could be multiple tree nodes having the same degree so
    //in following code will we find these nodes and we will merge them
    h->setHead();
    h->setSize();
    //we delete heap2
    BinomialTree * prev = NULL; //x
    BinomialTree * curr = uniqueKs; //n-x
    BinomialTree * next = curr->sibling; //s-n-x
    //Thses nodes helps us to detect consecutive nodes having the same degree k

    while(next != NULL){
        if((next->k != curr->k) || (next->sibling != NULL && curr->k == next->k && next->k==next->sibling->k)){
            prev = curr;
            curr = next;
            //If x != n-x then move all x, n-x and s-n-x by one step
            //If x == n-x == s-n-x then move by one step to get binomial combination of higher degree at later
        }
        //The node having bigger value should be parent. The other shoulod be child.(MAX HEAP)
        else{
            //max-binomial heap
            if(curr->frequency > next->frequency){
                curr->sibling = next->sibling;
                attachTrees(curr,next);
                //Current becomes the root of the next
            }
```

```cpp
            else{
                if(prev == NULL){uniqueKs = next;}
                else{prev->sibling = next;}
                //when current becomes child, the sibling of prev should be next,
                //otherwise prev would point dangerous node(o_o)
                attachTrees(next,curr);
                curr = next;
            }
        }
        next = curr->sibling;
    }
    this->head = uniqueKs;
    //The binomial tree nodes are in non-decreasing order and unique
}
void BinomialHeap::enqueue(int freq,document * doc){ //O(logn)
    BinomialTree * bt = createBinomialTree(freq,doc);
    BinomialHeap temp(bt,1);
    unionTree(&temp);
}
int BinomialHeap::getMax(){//O(logn)
    //To get maximum frequency in heap
    int max = INT_MIN;
    BinomialTree * maxFinder = this->head;
    while(maxFinder != NULL){
        max = max < maxFinder->frequency ? maxFinder->frequency : max;
        maxFinder = maxFinder->sibling;
    }
    return max;
}
```

```
BinomialTree * BinomialHeap::dequeue(){//O(logn)
    int maxValue = getMax();

    BinomialTree * maxNode = this->head;
    BinomialTree * tempHead = createBinomialTree(0,NULL);

    tempHead->sibling = maxNode;
    BinomialTree * prev = tempHead;

    while(maxNode->frequency != maxValue){
        maxNode = maxNode->sibling;
        prev = prev->sibling;
    }
    //Both max node and previous of max node is found
    prev->sibling = maxNode->sibling;
    //max node has been removed from from root list. But we should put its children as
    //reverse listed heap during union
    this->head = tempHead->sibling; //getting back head
    BinomialTree * childrenOfMin = reverseChildrenNode(maxNode->children,NULL);
    //children are reversed, now they are in non-decreasing order

    BinomialHeap bhTempMerge(childrenOfMin);
    //A temporary heap to merge two heap;
    unionTree(&bhTempMerge);
    //children have been put back in the heap
    this->size--;
    return maxNode;
}
```

Accessing File:

In this application we listed our files inside a directory called ‘file’, read those file and then passed file as document to binomial heap and ranked according to keyword.

accessfile.hpp

```
#include"binomialheap.hpp"
#ifndef ACCESSFILE_H_INCLUDED
#define ACCESSFILE_H_INCLUDED
void toLowerCase(char *);
char * subString(char *,int start,int end);
void accessOperation(void);
std::vector<std::string> getFileList(std::string &);
#endif // ACCESSFILE_H_INCLUDED
```

```
#include"accessfile.hpp"
using namespace std;
void toLowerCase(char * word){
    int size = (int)strlen(word);
    for(int i = 0; i < size; i++){
        word[i] = tolower(word[i]);
    }
}
char * subString(char * buffer, int start, int end){
    int size = end - start - 1;
    char * word = new char[size+1];
    for(int i = start+1; i < end; i++){
        word[i-start-1] = buffer[i];
    }
    word[size] = '\0';
    return word;
}
vector<string> getFileList(string folder)
{
    vector<string> names;
    string search_path = folder + "/*.*";
    WIN32_FIND_DATA fd;
    HANDLE hFind = ::FindFirstFile(search_path.c_str(), &fd);
    if(hFind != INVALID_HANDLE_VALUE) {
        do {
            // read all (real) files in current folder
            // , delete '!' read other 2 default folder . and ..
            if(! (fd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) ) {
                names.push_back(fd.cFileName);
            }
        }while(::FindNextFile(hFind, &fd));
        ::FindClose(hFind);
    }
    return names;
}
```

```cpp
void accessOperation(){
    ofstream output;
    output.open("output.txt",std::ios::out); //opening file in write mode
    char currentDirectory[PATH_MAX]; //maximum size of path depending on system
    _getcwd(currentDirectory,PATH_MAX); //for windows
    //getcwd(currentDirectory,PATH_MAX); for linux -- header uinstd
    strcat(currentDirectory,"\\file"); //current directory with folder file
    int stop = (int)strlen(currentDirectory);

    DIR * dir = opendir(currentDirectory);
    if(dir == NULL){
        cout<<"#######################################################################"<<endl;
        cout<<"# There is no directory called file in your Present Working Directory #"<<endl;
        cout<<"#           Make sure to create a directory namely \'file\'            #"<<endl;
        cout<<"#                 And store all document dataset in it                #"<<endl;
        cout<<"#######################################################################"<<endl;
        cout<<"BYE BYE!"<<endl<<endl<<endl;

        output<<"#######################################################################"<<endl;
        output<<"# There is no directory called file in your Present Working Directory #"<<endl;
        output<<"#           Make sure to create a directory namely \'file\'            #"<<endl;
        output<<"#                 And store all document dataset in it                #"<<endl;
        output<<"#######################################################################"<<endl;
        output<<"BYE, BYE!"<<endl<<endl<<endl;

        exit(0);
    }

    vector<string> fileList = getFileList(currentDirectory);
```

```cpp
    if(fileList.empty()){
        cout<<"#####################################################"<<endl;
        cout<<"#             There is no file to read              #"<<endl;
        cout<<"# Please make sure to include file to rank document #"<<endl;
        cout<<"#####################################################"<<endl;
        cout<<"BYE, BYE!"<<endl<<endl<<endl;

        output<<"#####################################################"<<endl;
        output<<"#             There is no file to read              #"<<endl;
        output<<"# Please make sure to include file to rank document #"<<endl;
        output<<"#####################################################"<<endl;
        output<<"BYE, BYE!"<<endl<<endl<<endl;

        exit(0);
    }
```

```
    //Since we need most frequent word we use MAX-HEAP
    BinomialHeap priorityQueue;
    //priorty queue of STL type queue<document>

    //Reading each document one by one
    for(const auto & it: fileList){
        document * doc = new document;
        doc->name = (char *)it.c_str();
        //cout<<doc.name<<endl;
        if(doc->name[0] == '.'){continue;}
        //If the file open with . then it is not content file and should not open it
        strcat(currentDirectory,"\\");
        strcat(currentDirectory,doc->name);
        ifstream input;
        input.open(currentDirectory,ios::in);
        currentDirectory[stop] = '\0';
        if(!input){continue;}
        //file is opened

        input.seekg(0,ios_base::end); //making get pointer to point last position in file
        //ifstream thre is get pointer --to end of file(eof)
        int size = (int)input.tellg();
        //cout<<size<<endl;
        //getting the size of document -- since we rae going to read entire file, not lin eby lir
        char * buffer = new char[size+1];
        input.seekg(0,ios_base::beg); //rewinding file pointer to initial point
        //cout<<input.tellg()<<endl;
        for(int i = 0; i < size; i++){
            buffer[i] = input.get();
        }
        buffer[size] = '\0';
        doc->text = buffer;
```

```
        int frequency = 0;
        //How many keywords are there in the current document?
        int start = -1, end;
        for(end = 0; end <= size; end++){
            if(!isalnum(buffer[end])&&buffer[end]!='\''){//alnum check alphanumeric defined in ctype
                //On encountering separator we divide the word as substring from sentence
                if(start+1 < end){
                    char * word = subString(buffer,start,end);
                    toLowerCase(word);
                    if(strcmp(word,keyword)==0){
                        frequency++;
                    }
                }
                start = end;
            }
        }
        priorityQueue.enqueue(frequency,doc);
        input.close();
    }

    cout<<"Priority Queue has been created"<<endl;
```

Extracting Output:

We extract or copy most relevant document to an array of Binomial Tree type from the binomial heap and the print the relevant documents to console as well as write it in our output file.

```
    BinomialTree * answers[numberOfRelevant];
    for(int i = 0; i < numberOfRelevant; i++){
        //This array will help us to store most frequent document on the basis of keyword
        //So that we can print them
        answers[i] = NULL;
    }

    //Let's dequeue or pull the document
    for(int i = 0; i < numberOfRelevant; i++){
        answers[i] = priorityQueue.dequeue();
    }

    //Printing the relevance order
    cout<<"#################################################################"<<endl;
    cout<<"#                    PRINTING DOCUMENT                         #"<<endl;
    cout<<"#################################################################"<<endl;
    cout<<endl<<endl<<endl;
    output<<"#################################################################"<<endl;
    output<<"#                    PRINTING DOCUMENT                         #"<<endl;
    output<<"#################################################################"<<endl;
    output<<endl<<endl<<endl;
    for(int i = 0; i < numberOfRelevant; i++){
        cout<<i+1<<". [name of document = "<<answers[i]->doc->name<<" | frequency of keyword = "<<answers[i]->frequency<<"]"<<endl;
        output<<i+1<<". ["<<answers[i]->doc->name<<" | "<<answers[i]->frequency<<"]"<<endl;
        cout<<answers[i]->doc->text<<endl<<endl<<endl<<endl<<endl;
        output<<answers[i]->doc->text<<endl<<endl<<endl<<endl<<endl;
    }
    output.close();
}
```

Output:

In testing of the output of our document, we downloaded the NLP dataset of emotion detection, but this dataset is huge and used for sentiment analysis. But our project is ranking of dcoument work with meta data or some specific keyword. So we reduced volume, reduced each text file data into 500 lines of file and then performed searching on documents.

We have 3 file namely doc1.txt, doc2.txt, doc3.txt in the directory file and then we run the progran and choose the keyword ‘joy’ according to which we are going to rank our document and choose most three relavant documents.

We got the ouput of document ranked and written in output file as follows:

1. [doc2.txt | 183]
2. [doc3.txt | 173]
3. [doc1.txt | 151]

Which means keyword ‘joy’ appears 183 times in in document 2, 173 times in document 3 and 151 times in document 1. The most relevant document is document 2.

```
D:\Programming\OOPDS\Document_Ranking\bin\Debug\Document_Ranking.exe
[PLEASE ENTER THE KEYWORD]:
=>joy


Priority Queue has been created
Now,Enter how many document do you need?
=>3
#################################################################
#                    PRINTING DOCUMENT                          #
#################################################################
```

Output File Content is as follows:

For output content, one must check output.txt file which will be created by application in its current directory, due large volume of data we are not adding it into report (not to increase report size).

Discussion:

What is advantage of using priority queue for such a problem?

When an average computer engineering student who had not heard of priority queues faced such a problem, he/she would probably follow a process like the one below.

- Getting the number of relevant words k and a keyword.
- Finding how many times the given keyword appears in each document.
- Creating an array of pairs where a pair consists of the frequency of the keyword in the document and the document itself.
- And sorting the array according to the frequencies.
- Getting the first k documents and printing them.

When we look at the steps she follows, we see that the time complexity of these operations is O (nlogn) since we are sorting the array of the documents. Actually as we have have discussed earlier, construction a priority queue is also O(nlogn). So why do we use priority queues then?

Of course we use priority queues because inserting an element into a priority queue is really fast. It so fast that, according to some of the resources on the Internet, insertion may take O(1) time if we prefer to use some efficient algorithms such as binomial heaps to construct this priority queue. This means that actually constructing a priority queue may take O(n) time, which is much faster than O(nlogn).

There is another reason. We use priority queues because sometimes we only want to reach a few elements which are the biggest or smallest elements of a given array. Assuming k is the number of relevant words, it would take O (nlogn + k) time to obtain the most suitable documents using the sorting algorithm, while using a priority queue it would take O (n + klogn) time. For small k, using a priority queue is much faster.

Refernces:

Blinkent University, (2008), CS473 Algorithms Lecture Binomial Heaps, Available at:

www.cs.bilkent.edu.tr/~atat/502/BinomialHeaps-new.ppt

Binary & binomial heaps at Foundation of Data Science by Damon Wischik, Available at:

https://www.youtube.com/watch?v=FMAG0aunrmM&list=PLknxdt7zG11PZjBJzNDpO-whv9x3c6-1H&index=20

Data Set Used(and we reduced volume manually): Emotion dataset for NLP by:

https://www.kaggle.com/datasets/praveengovi/emotions-dataset-for-nlp