**INTRODUCTION**

Binomial Heaps are highly advantageous data structures invented by French computer scientist Jean Vuillemin in 1977. The most noteworthy aspect of this structure is that it allows different heaps to merge together, whereas providing O(log n) or O(1) efficiency for all operations.

In this project, we are expected to read 42 files according to the input taken. Then, we should be holding our records in a binomial heap file. **Here, I preferred a max-heap property.** Since we would like to access the files with the most frequency of the given keyword, holding the ones with the highest similarity score at the topmost part is essential.

**SCREENSHOTS AND IMPLEMENTATION**

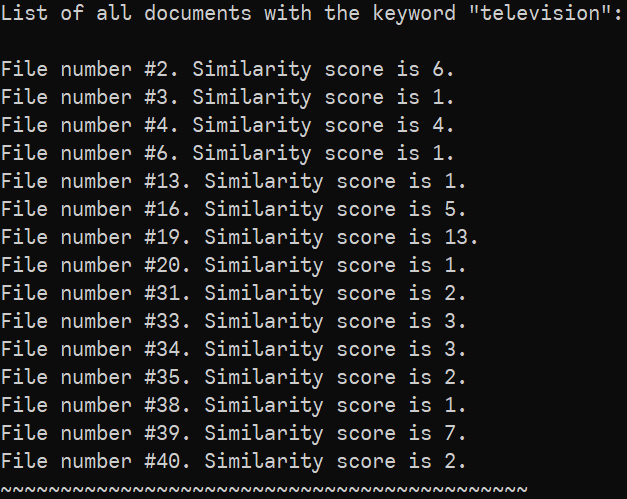
Before anything else, we should be taking the input from the user. I will be using “television” as an example.



Here, we take this keyword and start calculating similarity score for all 42 files. If a match is found in a document, its similarity score is incremented by one.

We have many documents containing the word “television”. Hence, let’s display the number of relevant documents:

*please see next page*



From this point, we should build our priority queue using a binomial heap implementation to hold these records. Then, we will extract the maximums one by one.

We will use “INSERT” operation for all of these files. Then, our binomial heap will be constructed.

Inserting in a binomial heap actually means creating a new heap with the current node and then using UNION operation with the main heap that we already got.

My source code is as follows:

*please see next page*

1. INSERT operation

**document\* insertNodeToHeap(document\* H, document\* insertThis) { /\* Inserting to a binomial heap actually means creating a new, empty heap and unifying it with what we already have. \*/**

**document\* H1 = initializeHeap();**

**/\* Set all fields as empty...\*/**

**insertThis->parent = NULL;**

**insertThis->child = NULL;**

**insertThis->sibling = NULL;**

**insertThis->degree = 0;**

**H1 = insertThis;**

**/\* Now, we have a heap from our node. \*/**

**/\* And unify it with the main heap we have! Also, upgrade it. \*/**

**H = unionHeaps(H, H1);**

**return H; /\* Return our main heap. \*/**

**}**

During the insertion, we use UNION to connect our newly-created null heap to the main heap we have

1. UNION operation

**document\* unionHeaps(document\* H1, document\* H2) { /\* This function here is identical to what we've seen in our lecture slides. We iterate through our nodes, checking all**

**possible conditions regarding the degrees and unifying them eventually.\*/**

**document\* previous;**

**document\* next;**

**document\* current;**

**document\* HEAD = initializeHeap();**

**HEAD = mergeHeaps(H1, H2); /\* merging the root lists, as we seen in our lectures. \*/**

**if (HEAD == NULL) /\* Taking precaution against empty heaps. \*/**

**return HEAD;**

**previous = NULL;**

**current = HEAD;**

**next = current->sibling;**

**while (next != NULL) {**

**if ((current->degree != next->degree) || ((next->sibling != NULL) /\* CASE ONE \*/**

**&& (next->sibling)->degree == current->degree)) {**

**previous = current;**

**current = next; /\* MOVE ONE RIGHT \*/**

**} else { /\* CASE TWO \*/**

**if (current->similarityScore > next->similarityScore) {**

**current->sibling = next->sibling;**

**Get\_a\_BT\_k(next, current);**

**} else { /\* CASE THREE \*/**

**if (previous == NULL)**

**HEAD = next;**

**else /\* CASE FOUR \*/**

**previous->sibling = next;**

**Get\_a\_BT\_k(current, next);**

**current = next;**

**}**

**}**

**next = current->sibling;**

**}**

**return HEAD;**

**}**

This was the enqueue part. For dequeue operation, I search the linked list of roots and find the maximum node as shown in our lectures. Then, I set its flag to -1. After this, I set all of my heap to NULL and construct the binomial heap WITHOUT that extracted node (if the flag value is equal to -1 for a node, then it will not be included in the construction of our binomial heap). I use a for loop to extract top 5 documents. If we have less than this, a break statement finalizes all found results. My implementation is below:

**for(m=0; m<5; m++) { /\* For extracting possible top five documents, do the following: \*/**

**document\* maxNode = findMaximumNode(H); /\* Finding the maximum number towards the linked list of roots, \*/**

**if(maxNode != NULL) {**

**if(m==4){**

**printf("Doc%d(%d).\n\n", maxNode->documentNumber, maxNode->similarityScore); /\* giving the results \*/**

**} else {**

**printf("Doc%d(%d), ", maxNode->documentNumber, maxNode->similarityScore);} /\* giving the results \*/**

**printResults[m] = maxNode; /\* This array will be used for printing later \*/**

**numberInserter++; /\* Iterate through the next \*/**

**H = NULL; /\* Now, this is the important part! After the extracted our maximum number, it's flag value is set to -1. Also now, I completely resetted my heap. It is now null.**

**It will be rebuilt WITHOUT the extracted max.\*/**

**maxNode->flag = -1;**

**for(i=0; i<sizeOfArr; i++) {**

**if(relevantDocuments[i]->flag != -1) /\* Do not take the already-extracted maximum nodes into consideration \*/{**

**H = insertNodeToHeap(H, relevantDocuments[i]); /\* Build the heap \*/**

**} else {**

**continue;}**

**}} else {**

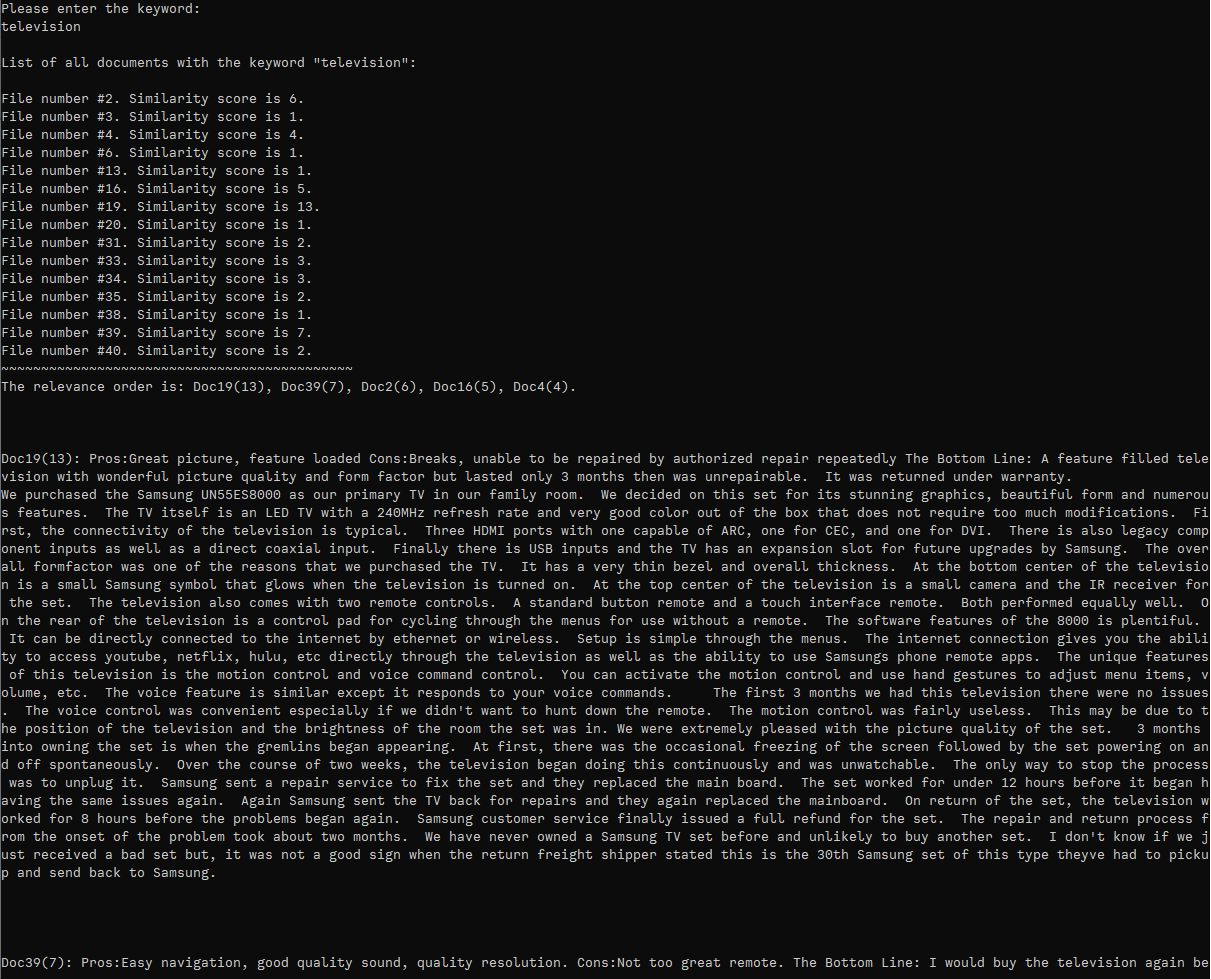
**break;}**

**}**

These two parts conclude both constructing the binomial heap (i.e. enqueue) and obtaining maximum nodes from it.

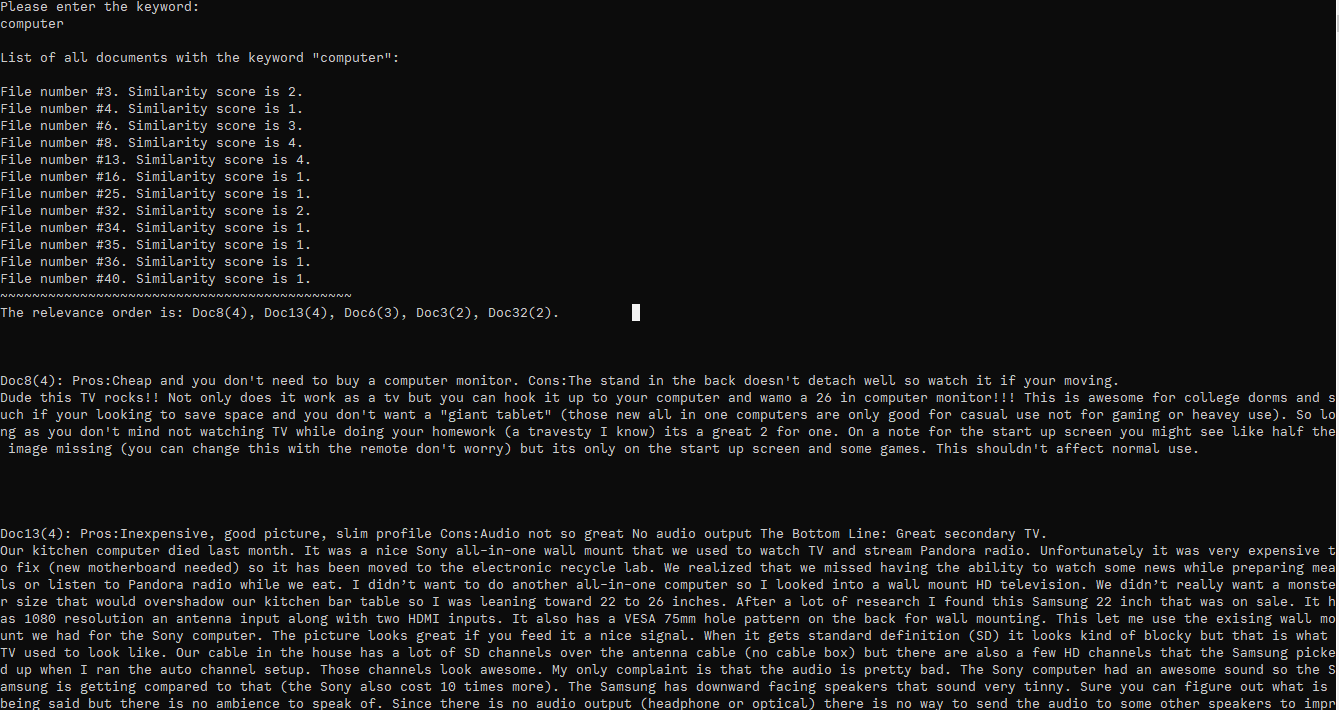
**GETTING THE RESULTS**

*please see next page*

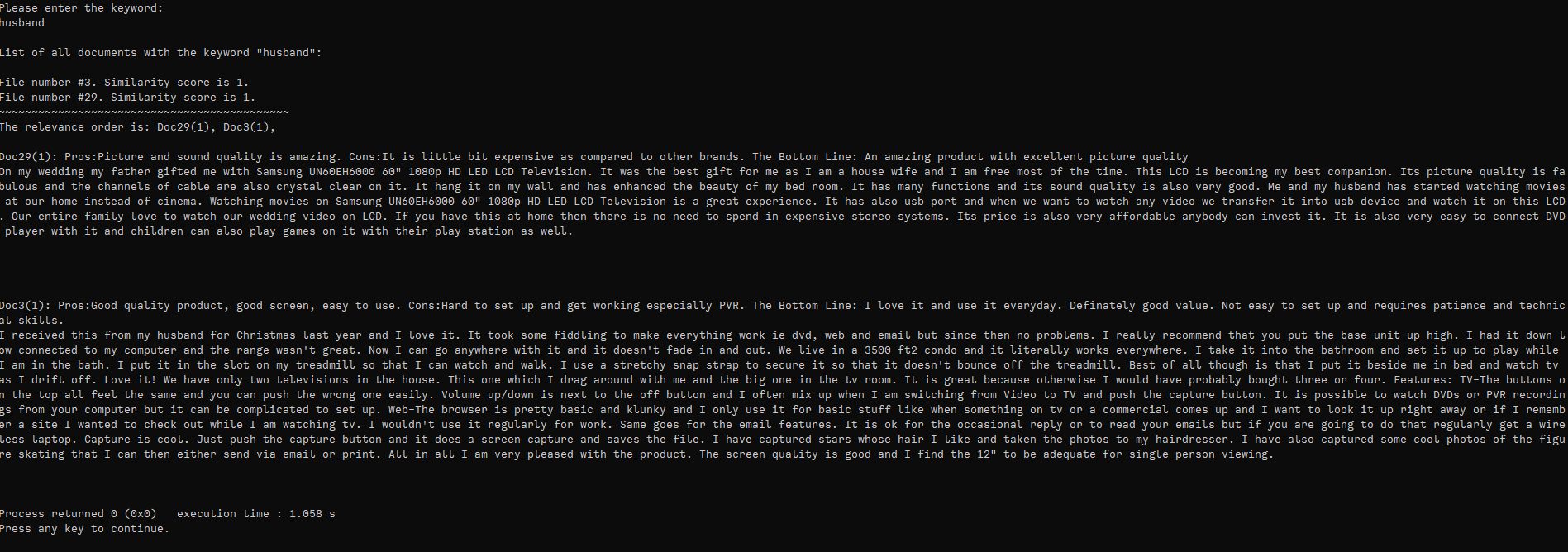


(The results continue and it prints out all documents with the desired output format.)

Let’s show a different example, this time using the word “computer”...

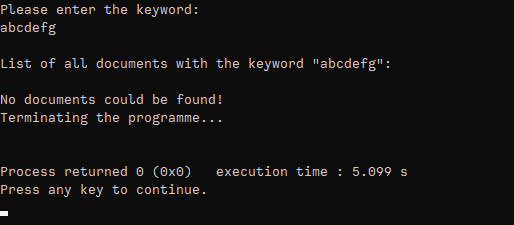


Also, let’s demonstrate a situation where we have less than 5 relevant documents.



The word “husband” had occurences in only 2 files, each with a similarity score of 1. Our programme successfuly terminated the loop before reaching 5.

Finally, let’s show what happens if no relevant documents can be found:



These screenshots conclude our results section.

**DISCUSSION: ADVANTAGES OF PRIORITY QUEUE**

In this project, we are given the task of ranking documents. This means that some of the records we hold are more *important* than others and means more for our purpose. Yes, they can be hold in arrays, linked lists or something else: however, if we hold them in a data structure which we can reach the most prioritized one with the easiest effort, scaling of our project for large inputs will be much more efficient. Here what we give importance to is the similarity scores. Therefore, documents with higher similarity score should be kept in a max-heap structure for obtaining them even faster.

There are fibonacci heaps, classical binary heaps and binomial heaps that can provide this priority queue property. Binomial heaps are a *forest* of trees and merging them is lot faster, O(log n). This is the reason why we’ve chosen priority queues instead of other data structures.

This explanation concludes my project report.

Thank you for the term, Sir/Madam