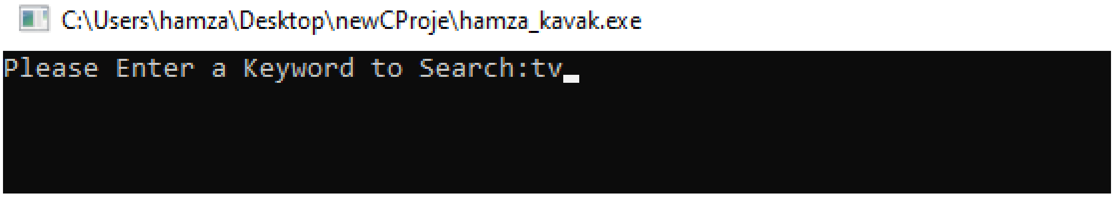
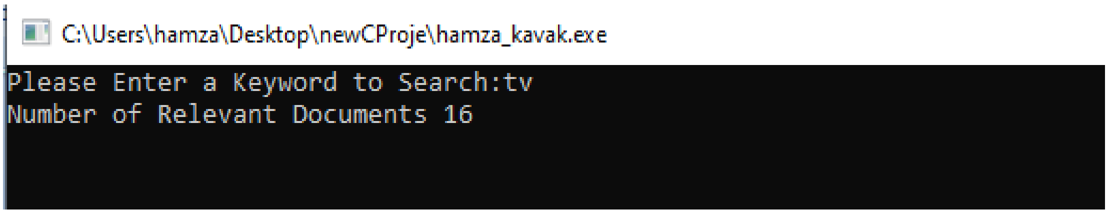
CSE225 Data Structures Project #2 Report

1-) For example: **tv**



2-)Number of relevant documents (Total - not relevant documents)



3-)Enqueue

```
//newnode function

struct node* newNode(int value,char fileName[200]){

        struct node* x;

        x = (struct node*)malloc(sizeof(struct node));

        x->child = x->parent =x->sibling = NULL;

        x->degree=0;

        strcpy(x->fileName,fileName);//fileName

        x->value = value; // value is frequency of the parameters searched key

        return x;

}
```

```
// heapMerge function

struct node* heapMerge ( Node* H1,  Node* H2){

        //H1=H uniondan gelen

        //H2=H1 yine uniondan gelen

        Node* H = (Node* )malloc(sizeof(Node));

        H=NULL;

        struct node* x= (Node*)malloc(sizeof(Node));

        struct node* y= (Node*)malloc(sizeof(Node));

        struct node* t= (Node*)malloc(sizeof(Node));

        struct node* z= (Node*)malloc(sizeof(Node));

        H=NULL;

        x = H1;

        y = H2;

        if(H1==NULL)return H2;

        if(H2==NULL)return H1;

         if (x != NULL && y != NULL) {

          if (x->degree <= y->degree){ H=x; }elseH=y;}

        while(x != NULL && y != NULL) {

                if (x->degree < y->degree){ x=x->sibling; }

                else if (x->degree == y->degree){

                        t=x->sibling;
                        x->sibling = y;
                         x=t;

                }

                else{
                           z=y->sibling;
                           y->sibling=x;
                            y=z;
                        }
                    }

        return H;
    }
```

// heapUnion function

```
Node* heapUnion( Node* H1,  Node* H2){

        if(H1== NULL && H2==NULL){ return NULL; }

        Node* H = (Node* )malloc(sizeof(Node));

        H = heapMerge(H1, H2);
           Node *prev_x = NULL;
           Node *temp = H;
           Node * next_x=temp->sibling;

        while (next_x != NULL) {

                if ((temp->degree != next_x->degree) || ((next_x->sibling != NULL) && next_x->sibling->degree
                == temp->degree)){                          prev_x = temp;
                temp = next_x;
                 } else if (temp->value >= next_x->value){ //Here
                //Here we are doing max heap here. To keep the biggest ones as root
                  temp->sibling = next_x->sibling;
                binomialLink(next_x,temp);
                } else {
                if (prev_x == NULL){
                      H = next_x;
                 }else{
                     prev_x->sibling = next_x;
                }
                binomialLink(temp,next_x);
                temp= next_x;
                }
                next_x = temp->sibling;

        } return H; }
```

```
//this function checks the siblings and finds the largest of them

int exractMethod(struct node* H) {

 struct node* p;
 struct node* temp;
 struct node* pre;
 struct node* next;
int tempMax=0;
char tempFileName[100];
    if (H == NULL) {
            printf("\n Empty Heap");
            return 0;
 }

    p = H;
 while (p != NULL) {
// printf("%d", p->value);
 if(p->value > tempMax)
 {
        tempMax = p->value;
        strcpy(tempFileName,p->fileName);
         temp=p;
 }

//if (p->sibling != NULL)
 //printf("-->");
p = p->sibling;
 }
//Here too I print the biggest one on the screen
printf( "Filename:%s (%d)\n",tempFileName,tempMax);
deleteNode(&HH,tempMax,tempFileName);
}



//It puts the (largest root) to be deleted here

 deleteNode(struct node** heap, int key, char fileName[200]) {
 struct node *temp = *heap;
 struct node *prev;
  if (temp != NULL && temp->value == key && (strcmp(fileName,temp->fileName) == 0)) {
      *heap = temp->sibling;
       return;
 }

  // Find the key to be deleted
while (temp != NULL && temp->value != key) {
  prev = temp;
  temp = temp->sibling;
 }

if (temp == NULL) return;

 prev->sibling = temp->sibling;
printNode(temp->child);

//I throw it to the printf function, to add the ones below it back to the heap
}
```

//I find the one to be deleted from the root, detach it, take its child and add all the underlying elements (childs and siblings) back to the existing list.

```
printNode(struct node *n){
if(n==NULL)
 return;

  struct node* np;
// printf("%s değeri %d \n ",n->fileName,n->value);

 HH = heapUnion(HH, newNode(n->value,n->fileName));

//printf("%s değeri%d - sibling %d- cdilf %d\n ",n->fileName,n->value,n->sibling->value,n->child->value);

    printNode(n->sibling);
    printNode(n->child);
}
```

4-)Relevance Order

```
Relevance Order is
1 -Filename:content_609140248196 (8)
2 -Filename:content_627493604996 (7)
3 -Filename:content_646599577220 (6)
4 -Filename:content_646179491460 (5)
5 -Filename:content_275965120132 (4)
```

<u>Full output</u>

*Please Enter a Keyword to Search:**tv***

*Number of Relevant Documents **16***

Relevance Order is

*1 -Filename:content_609140248196 (**8**)*

*2 -Filename:content_627493604996 (**7**)*

*3 -Filename:content_646599577220 (**6**)*

*4 -Filename:content_646179491460 (**5**)*

*5 -Filename:content_275965120132 (**4**)*

5-)

The advantage of using priority queue is that it is easy to implement and processes with different priorities can be managed efficiently. In addition, according to the structure we have established, we can very easily reach the value we want min-max in terms of time complexity. Because priority queue, nodes can be weighted; This ensures that the high priority ones are always moved towards the start of the tail in front of the lower priority ones instead of being added to the tail of the tail as in a normal queue.

Hamza KAVAK

150118886