```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

// all the basic data structures and functions are included in this template
// you can add your own auxiliary functions as you like

// data type for heap nodes
typedef struct HeapNode {
        // each node stores the priority (key), name, execution time,
        //  release time and deadline of one task
        int key; //key of this item
        int TaskName;  // task name
        int Etime; //execution time
        int Rtime; // release time
        int Dline; // deadline
        struct HeapNode *parent; //pointer to parent
        struct HeapNode *left; //pointer to left child
        struct HeapNode *right; //pointer to right child
} HeapNode;

//data type for a priority queue (heap)
typedef struct Heap{ //this is heap header
        int  size;      // count of items in the heap
        HeapNode *LastNode; // last node pointer
        HeapNode *root; // pointer to the root of heap
} Heap;

// create a new heap node to store an item (task)
HeapNode *newHeapNode(int k, int n, int c, int r, int d, HeapNode *L, HeapNode *R, HeapNode *P)
{ // k:key, n:task name, c: execution time, r:release time, d:deadline
  // L, R, L: pointers to left child, right child and parent, respectively
        HeapNode *new;
        new = malloc(sizeof(HeapNode));
        assert(new != NULL);
        new->key = k;
        new->TaskName = n;
        new->Etime = c;
        new->Rtime = r;
        new->Dline = d;
        new->left = L; // left child
        new->right = R; // righ child
        new->parent = P; // parent
        return new;
}

// create a new empty heap-based priority queue
 Heap *newHeap()
{ // this function creates an empty heap-based priority queue
        Heap *T;
        T = malloc(sizeof(Heap));
        assert (T != NULL);
        T->size = 0;
        T->LastNode=NULL;
        T->root = NULL;
        return T;
}
//Find the position of next node after insert
 HeapNode *LastNodeForInsert(HeapNode *node)
{
        HeapNode *parents,*new;
        parents = node->parent ;
        if(parents!= NULL){     //not the root node
                while(parents!=NULL && parents->left!=node)          //go up until a left child or
the root is reached
                {
                        node = node->parent;
                        parents = parents->parent;
```

```
                }
                if(parents==NULL){                          //if root is reached,node is point to
 the root node
                        new=node;
                }
                else{                                                                   //if a
 left child is reached
                        new = parents->right;                                   //go to the right
 child
                        if (new==NULL)  return parents;         //if right child does not
 exist,it's the position of next node
                }
                while (new->left!=NULL)  new = new->left;   //go down left until a leave is
 reached
                return new;
        }
        new = node;  //if node is the root
        return new;
 }

 // Update the LastNode after removal
 HeapNode *LastNodeForRemove(HeapNode *node)
 {
        HeapNode *parents,*new;
        parents = node->parent;
        if(parents!= NULL){                                         //not the root node
                while(parents!=NULL &&parents->right!=node)     //go up until a right child or the
 root is reached
                {
                        parents = parents->parent;
                        node = node->parent;
                }
                if(parents== NULL) new=node;                    //if root is reached,node is point
 to the root node
                else new = parents->left;                       //if a right child is reached, go
 to the left child

                while (new->right!=NULL){
                        new = new->right;                           //go down right until a leave
 is reached
                }
                return new;
        }
        new = node;  //if node is the root
        return new;


 }

 //time complexity: Since the heap has height logN ,it's O(logN).
 void Insert(Heap *T, int k, int n, int c, int r, int d)
 { // k: key, n: task name, c: execution time, r: release time, d:deadline
   // You don't need to check if this item already exists in T
        HeapNode *parents,*new,*grand,*lefts,*rights,*last;
        if (T->size ==0){                           //for the base case
                new = newHeapNode(k,n,c,r,d,NULL,NULL,NULL);
                T->root = new;
                T->LastNode = T->root;
                T->size =1;
                return;
        }
        last = LastNodeForInsert(T->LastNode);       //find the new position of new node
        new = newHeapNode(k,n,c,r,d,NULL,NULL,last);
        if (last->left == NULL)last->left =new;      //insert new item into heap
        else last->right = new;
        T->size +=1;                                 //increase the heap size

        if (new->key >= last->key){                  //if it's ordered,no need of upheap
                T->LastNode = new;
```

```
                return;
        }
        parents = last;

        // upheap until reached root or parent has a key smaller than or equal to node
        while (parents!=NULL && new->key < parents->key ){
                if(parents->parent==NULL){              //if the root node is reached
                        new->parent = NULL;
                        T->root = new;
                }
                else{
                        grand = parents->parent;
                        if(grand->right == parents)  grand->right = new; //decide whether it's a
left child or right child of the grand node
                        else grand->left =new;
                        new->parent = grand;
                }

                rights= new->right;           //record the left and right children of current node
                lefts= new->left;
                if(parents->right == new) {   //swap current node with its parent node
                        new->right =parents;
                        if (parents->left !=NULL){
                                new->left = parents->left;
                                parents->left->parent = new;
                        }
                }
                else{
                        new->left = parents;
                        if (parents->right !=NULL){
                                new->right = parents->right;
                                parents->right->parent = new;
                        }
                }

                parents->left = lefts;                          //link with new children
                parents->right = rights;
                if (rights!=NULL) rights->parent = parents;
                if (lefts!=NULL) lefts->parent = parents;
                parents->parent = new;
                parents = new->parent;                          //compare with new parent node

        }
        T->LastNode = last;
}

void Print(HeapNode *node){
        if (node!=NULL){
                printf(" %d%d ",node->TaskName,node->key);
                Print(node->left);
                Print(node->right);
        }
}

//time complexity:Since the node with the smallest key in at the root of the Heap,it's O(1).
//While after removal,it takes O(logN) for downheap.(The height of heap is logN)
//Hence,it's O(logN)
HeapNode *RemoveMin(Heap *T)
{
        HeapNode *node,*lefts,*rights,*last,*temp,*record;
        node=NULL;
        if (T->size ==1){                                       // if only one node left.
                node = T->LastNode;
                T->root=NULL;
                T->LastNode =NULL;
                T->size = 0;
                return node;
        }
        last = T->LastNode;
```

```
        node = T->root;
        record = newHeapNode(node->key, node->TaskName, node->Etime, node->Rtime,node->Dline,NULL,NULL,NULL);    //record the value of removed node
        node->key = last->key;                                  //replace the root key and attributes with last node.
        node->Dline = last->Dline;
        node->Etime = last->Etime;
        node->Rtime = last->Rtime;
        node->TaskName = last->TaskName;
        T->LastNode = LastNodeForRemove(last);                //update the last node
        if (last->parent->left == last) last->parent->left =NULL;       //remove last node;
        else last->parent->right =NULL;
        last->parent = NULL ;
        T->size -= 1;                                          //decrease the heap size

        //downheap until reached a leaf or children have keys greater than or equal to node.
        while((node->left!=NULL && node->key > node->left->key) || (node->right!=NULL && node->key > node->right->key )){
                if (node->right==NULL || (node->right!=NULL && node->left->key <= node->right->key)){    //swap the root with child which has smaller key
                        temp = node->left;
                        lefts =temp->left;
                        rights = temp->right;
                        temp->right = node->right;
                        if (node->right !=NULL)node->right->parent = temp;
                        temp->left = node;
                }
                else{
                        temp = node->right;
                        lefts = temp->left;
                        rights = temp->right;
                        temp->left = node->left;
                        if(node->left !=NULL) node->left->parent = temp;
                        temp->right = node;
                }
                if (node->parent!=NULL){
                        if (node->parent->left==node) node->parent->left = temp;
                        else node->parent->right =temp;
                }
                temp->parent = node->parent;
                node->parent = temp;
                node->left = lefts;
                node->right = rights;
                if(lefts!=NULL) lefts->parent = node;
                if(rights!=NULL) rights ->parent = node;
                if (node==T->root) {
                        T->root = node->parent;
                        T->root->parent =NULL;
                }
                if (temp==T->LastNode){
                        T->LastNode = node;
                }
        }
        return record;
}

//returns the latest start time of the root item.
int MinHelp(Heap *T){
        if (T->size !=0)
                return T->root->Dline-T->root->Etime;
        else return -1;
}

//returns the smallest key in heap.
//time complexity: O(1)
int Min(Heap *T)
{
        if (T->size !=0)
                return T->root->key;
```

```
        else return -1;
}


//time complexity:for a file with n tasks,it need n times Insert operation.
//Since a Insert operation is O(logn),it's O(nlogn).
int TaskScheduler(char *f1, char *f2, int m )
{
        int a,b,c,d;
        HeapNode *releasenode,*readynode;
        FILE *fp = fopen(f1,"r");
        if (fp==NULL){
                printf("%s does not exist",f1);
                exit(1);
        }
        Heap *F;                        //read data from file1
        F = newHeap();
        while(!feof(fp)){
                fscanf(fp,"%d %d %d %d\n",&a,&b,&c,&d);
                Insert(F,c,a,b,c,d);      //insert item in heap F,with release time as the keys.
        }
        fclose(fp);

        FILE *fp2 = fopen(f2,"w");    //write data to file2
        int target=0,k=0,j;
        int core[m];                  //use array to record the usage of core

        for (j=0;j<m;j++){            //initialize the array
                core[j]=0;
        }
        Heap *R;
        R = newHeap();
        while(F->size!=0 || R->size!=0 ){
                while(F->size!=0 && Min(F)<=k){     //check whether the tasks is ready
                        releasenode = RemoveMin(F);         //insert the task into heap R,with
deadline as keys
                        Insert(R,releasenode->Dline,releasenode->TaskName,releasenode-
>Etime,releasenode->Rtime,releasenode->Dline);
                }
                for(int j=0;j<m;j++){               //find suitable core for each task
                        int lateststart = MinHelp(R);   //get the latest start time for earliest
task in
                        if (lateststart==-1) break;
                        if(R->size!=0 && core[j]<=lateststart  ){       // check whether the core
is suitable for the task
                                readynode = RemoveMin(R);
                                if( core[j]<readynode->Rtime) core[j]=readynode->Rtime;
                                fprintf(fp2,"%d core%d %d\n ",readynode->TaskName,j,core[j]);
                                core[j]+=readynode->Etime;
                        }
                        if (j==0) target =core[j];
                        if (core[j]>target) target=core[j];
                }
                if(target>k) k=target;
                else k++;
                if(F->size==0 && k>releasenode->Dline) return 0;
        }
        return 1;
}

int main() //sample main for testing
{   int i;
        i=TaskScheduler("samplefile1.txt", "feasibleschedule1.txt", 4);
        if (i==0) printf("No feasible schedule!\n");
        /* There is a feasible schedule on 4 cores */
        i=TaskScheduler("samplefile1.txt", "feasibleschedule2.txt", 3);
        if (i==0) printf("No feasible schedule!\n");
        /* There is no feasible schedule on 3 cores */
        i=TaskScheduler("samplefile2.txt", "feasibleschedule3.txt", 5);
```

```
        if (i==0) printf("No feasible schedule!\n");
        /* There is a feasible scheduler on 5 cores */
        i=TaskScheduler("samplefile2.txt", "feasibleschedule4.txt", 4);
        if (i==0) printf("No feasible schedule!\n");
        /* There is no feasible schedule on 4 cores */
        i=TaskScheduler("samplefile3.txt", "feasibleschedule5.txt", 2);
        if (i==0) printf("No feasible schedule!\n");
        /* There is no feasible scheduler on 2 cores */
        i=TaskScheduler("samplefile4.txt", "feasibleschedule6.txt", 2);
        if (i==0) printf("No feasible schedule!\n");
        /* There is a feasible scheduler on 2 cores */
        return 0;
}
```