Project: Elevators

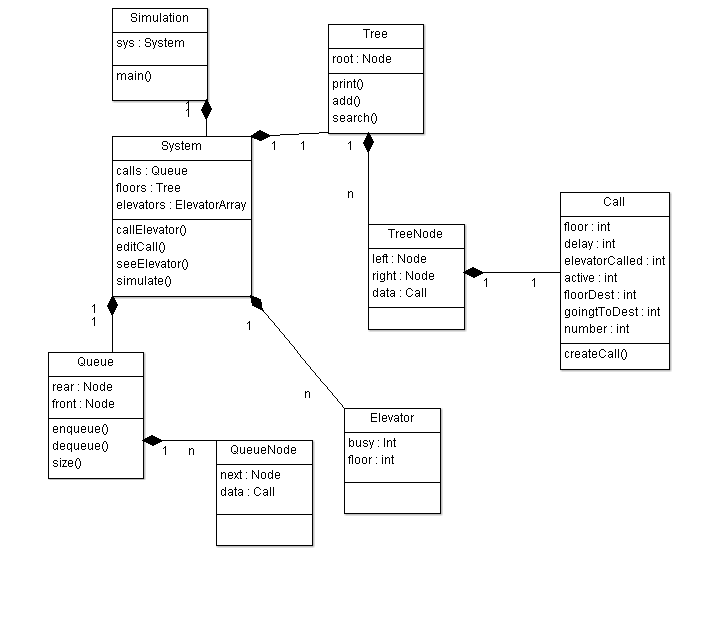
Author: Tomás Lucas Vilaboa

Hours: 7

Code Lines: 295

Files: Main.c, call.h,elevator.h,queueCalls.h,treeElevators.h

UML



Main.c

#include <stdio.h>

#include <stdlib.h>

#include "Elevator.h"

#include "call.h"

#include "treeElevators.h"

#include "queueCalls.h"

int cantElevators;

//Elevators calls are stored and processed when 5 secs are simulated

int main()

{

node \* root;

root =NULL;

printf("Welcome to Elevators 2.0\n");

printf("Please enter the numbers of elevators to simulate.\n");

scanf("%d",&cantElevators);

struct elevator elevators[cantElevators];

int i=0;

for(i;i<cantElevators;i++){

struct elevator elevator;

elevator.floor=0;

elevator.busy=0;

elevators[i]=elevator;

}

i=0;

for(i;i<25;i++){

struct call call;

call.floor=i;

call.active=0;

call.delay=0;

insert(&root,call);

}

printf("Simulation begins.\n");

int menu;

int cantCalls=0;

do{printf("\n\n1- See elevator floor \n2- Call a elevator \n3- Simulate 5 seconds \n4- End simulation \n\n");

scanf("%d",&menu);

switch(menu){

case 1: seeElevator(elevators); break;

case 2: {storeCall(cantCalls); cantCalls=cantCalls+1; break;}

case 3: simulate(root,elevators);break;

}}

while(menu<4);

print\_inorder(root);

return 0;

}

void storeCall(int number){

struct call call;

call.number=number;

push(call);

}

void simulate(node \* tree,struct elevator elevators[]){

int i=0;

for(i;i<cantElevators;i++){

if(elevators[i].floor ==0 && elevators[i].busy==0 ){

struct call call;

struct Node \*nextCall=delQueue();

call=nextCall->Data;

if(call.active != 2){

callElevator(elevators[i],tree,call,i);

}

}

}

traverseTree(tree,elevators);

}

void traverseTree(node \* tree,struct elevator elevators[])

{

if (tree)

{

traverseTree(tree->left,elevators);

if(tree->data.active ==1){

if(tree->data.floor != elevators[tree->data.elevatorCalled].floor && tree->data.goingToDest ==0){

tree->data.delay=tree->data.delay + 1;

elevators[tree->data.elevatorCalled].floor=elevators[tree->data.elevatorCalled].floor + 1;

} else {

tree->data.goingToDest=1;

tree->data.delay=tree->data.delay + 1;

}

if(tree->data.floorDest < elevators[tree->data.elevatorCalled].floor){

elevators[tree->data.elevatorCalled].floor=elevators[tree->data.elevatorCalled].floor - 1;

} else if(tree->data.floorDest > elevators[tree->data.elevatorCalled].floor){

elevators[tree->data.elevatorCalled].floor=elevators[tree->data.elevatorCalled].floor + 1;

} else {

tree->data.active=0;

elevators[tree->data.elevatorCalled].floor=0;

elevators[tree->data.elevatorCalled].busy=0;

}

}

traverseTree(tree->right,elevators);

}

}

void seeElevator(struct elevator elevators[]){

printf("Please enter the number of elevator to see.\n");

int elev;

scanf("%d",&elev);

printf("Elevator %d is in floor %d",elev,elevators[elev].floor);

}

void callElevator(struct elevator elevator,node \* root,struct call call,int i){

printf("\nCall number %d \n",call.number);

printf("Please enter the floor the elevator is being called from.\n");

int floor;

scanf("%d",&floor);

printf("Elevator %d is assigned.",i);

printf("Please enter the floor the elevator has to go.\n");

int floorDest;

scanf("%d",&floorDest);

elevator.busy=1;

createCall(&call,floor,i,floorDest,call.number);

editCall(root,call);

}

void editCall(node \* root,struct call call){

node \*tmp;

tmp = search(&root, call);

if (tmp)

{

tmp->data.elevatorCalled=call.elevatorCalled;

tmp->data.active=1;

tmp->data.floorDest=call.floorDest;

}

else

{

printf("Data Not found in tree.\n");

}

}

Call.h

struct call{

int floor;

int delay;//=0

int elevatorCalled;

int active;//=0

int floorDest;

int goingToDest;

int number;

};

void createCall(struct call \*call,int floor,int elevatorCalled,int floorDest,int number){

call->active=1;

call->floor=floor;

call->delay=0;

call->elevatorCalled=elevatorCalled;

call->floorDest=floorDest;

call->goingToDest=0;

call->number=number;

}

Elevator.h

struct elevator {

int busy; //=0

int floor;

};

QueueCalls.h

int siz=0;

//equals to pop, delete first value from queue and returns it

//active=2 means its not an actual Call, but in order to avoid a bug and don’t let the queue empty I

//need to simulate a call

struct Node\* delQueue()

{

struct Node \*temp, \*var=rear;

if(siz >1)

{

rear = rear->next;

siz-=1;

return var;

}

else if(siz== 1){

struct call call;

call.active=2;

push(call);

siz-=1;

rear = rear->next;

return var;

} else {

printf("\nQueue Empty");

}

}

void push(struct call value)

{

struct Node \*temp;

temp=(struct Node \*)malloc(sizeof(struct Node));

temp->Data=value;

if (front == NULL)

{

front=temp;

front->next=NULL;

rear=front;

}

else

{

front->next=temp;

front=temp;

front->next=NULL;

}

siz+=1;

}

void display()

{

struct Node \*var=rear;

if(var!=NULL)

{

printf("\nElements are as: ");

while(var!=NULL)

{

printf("un call");

var=var->next;

}

printf("\n");

}

else

printf("\nQueue is Empty");

}

int sizeofqueue(){

return siz;

}

TreeElevators.h

struct bin\_tree {

struct call data;

struct bin\_tree \* right, \* left;

};

typedef struct bin\_tree node;

void insert(node \*\* tree, struct call val)

{

node \*temp = NULL;

if(!(\*tree))

{

temp = (node \*)malloc(sizeof(node));

temp->left = temp->right = NULL;

temp->data = val;

\*tree = temp;

return;

}

if(val.floor < (\*tree)->data.floor)

{

insert(&(\*tree)->left, val);

}

else if(val.floor > (\*tree)->data.floor)

{

insert(&(\*tree)->right, val);

}

}

void print\_preorder(node \* tree)

{

if (tree)

{

printcall(tree);

print\_preorder(tree->left);

print\_preorder(tree->right);

}

}

void print\_inorder(node \* tree)

{

if (tree)

{

print\_inorder(tree->left);

printcall(tree);

print\_inorder(tree->right);

}

}

void print\_postorder(node \* tree)

{

if (tree)

{

print\_postorder(tree->left);

print\_postorder(tree->right);

printcall(tree);

}

}

void printcall(node \* tree){

printf("Floor:%d\n",tree->data.floor);

printf("TotalDelay:%d\n\n",tree->data.delay);

}

void deltree(node \* tree)

{

if (tree)

{

deltree(tree->left);

deltree(tree->right);

free(tree);

}

}

//search and return a call

node\* search(node \*\* tree, struct call val)

{

if(!(\*tree))

{

return NULL;

}

if(val.floor < (\*tree)->data.floor)

{

search(&((\*tree)->left), val);

}

else if(val.floor > (\*tree)->data.floor)

{

search(&((\*tree)->right), val);

}

else if(val.floor == (\*tree)->data.floor)

{

return \*tree;

}

}