CSPC43 - Operating Systems

Programming Assignment

106119100 - Rajneesh Pandey

Topic : Staircase Scheduler and Completely Fair Scheduler

The data structures I used here include **Min Heap** and **Red Black Trees**.
The implementation language is **C++.**

The source code has 3 classes which provide the required abstraction.

1) **Class Scheduler** -
This is the top level class which provides the **scheduler functionality**. The methods that belong to this class are.
- void printjob(int JobID); prints the matching job id
- void printjob(int JobIDlow,int JobIDhigh); prints the range of jobIDs
- void nextjob(int JobID); prints the next greatest job wrt inorder traversal
- void prevjob(int JobID); prints last highest job id wrt inorder traversal
- void insert(int JobID, int time); inserts new job
- scheduler(); constructor that initializes the counter and creates objects of class heap & rbt
- void syncTime(int time); calls dispatch method until counter matches the timestamp of current comand
- int dispatch(int); schedules jobs
- int ifjob(); checks if there are any jobs in the queue.

2) **Class heap**
This class facilitates the **Min Heap data structure**. The methods are
- heapnode* insert(int, int, int,rbtnode*); insert a new node into the heap.
- struct heapnode* removeMin(); remove the item from top of the heap ie minimum executed time.
- void swapJob(struct heapnode* a,struct heapnode* b); swap the positions of two nodes
- void heapify(); fix heap properties after a remove min
- void updateMin(int exec_time) // updates the root and re arranges the heap
- heap(int) // constructor
- void execute(int) // function to execute a job

3) **Class rbt**
This class is responsible for all features of a **red black tree data structure**. The methods are
- void rotateleft(rbtnode *&, rbtnode *&); do a left rotation
- void rotateright(rbtnode *&, rbtnode *&); do a right rotation
- void fixtree(rbtnode *&, rbtnode *&); fix rbt properties after insert operation
- rbt(); constructor to initalize the tree
- rbtnode* insert(const int &n); insert new node into the tree
- rbtnode* findnode(int jobid); find node based on job id
- void nextnode(int jobid); find next node in inorder traversal
- void prevnode(int jobid); frind prev node in inorder traversal
- void inorder(int, int); print all values in inorder travel in the range low-high
- void deletenode(rbtnode*); - delete a node from the tree
- void fixviolation(rbtnode*); fix violation caused by deletion

Main.cpp file and Scheduler Class

```
//106119100 Rajneesh Pandey

#include <iostream>
#include <fstream>
#include <sys/time.h>
#include <cstddef>
#include <cstring>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

#define BUFFER_SIZE 1000

ofstream fout;
#include "heap.h"
#include "rbt.h"
#include "heap.cpp"
#include "rbt.cpp"

class scheduler
{
private:
    int counter;
    rbt *myrbt;
    heap *myheap;

protected:
public:
    //Print(jobID,executed_time,total_time) for given jobID
    void printjob(int JobID);
    //Print(jobID,executed_time,total_time) for given range of jo
bIDs
    void printjob(int JobIDlow, int JobIDhigh);
    //Print the NextJob(jobID,executed_time,total_time) for next
greatest job id
    void nextjob(int JobID);
    void prevjob(int JobID);
    void insert(int JobID, int time);
    scheduler();
    void syncTime(int time);
    int dispatch(int);
    int ifjob();
};

int scheduler::dispatch(int x)
{
    int retval = 0;
```

```cpp
    heapnode *thisnode = &(myheap->root[1]);
    thisnode->exec_time += x;
    if (thisnode->total_time - thisnode->exec_time <= 0)
    {
        retval = thisnode->exec_time - thisnode->total_time;

        myrbt->deletenode(thisnode->twin);
        myheap->removeMin();
    }
    myheap->heapify();

    return retval;
}
void scheduler::syncTime(int time)
{
    int param, remain;
    while (time > counter)
    {
        if (time - counter < 5)
            param = time - counter;
        else
            param = 5;
        if (ifjob())
        {
            remain = dispatch(param);
            counter += param - remain;
        }
        else
            counter = time;
    }
}
scheduler::scheduler()
{
    counter = 0;
    myheap = new heap(BUFFER_SIZE);
    myrbt = new rbt;
    //create rbt
}
void scheduler::printjob(int JobID)
{
    myrbt->findnode(JobID);
}

void scheduler::printjob(int JobIDlow, int JobIDhigh)
{
    myrbt->inorder(JobIDlow, JobIDhigh);
}
```

```
void scheduler::nextjob(int JobID)
{
    myrbt->nextnode(JobID);
}

int scheduler::ifjob()
{
    return myheap->last;
}

void scheduler::prevjob(int JobID)
{
    myrbt->prevnode(JobID);
}

void scheduler::insert(int JobID, int totaltime)
{

    rbtnode *temp2 = myrbt->insert(JobID);

    //insert into heap
    myheap->insert(JobID, 0, totaltime, temp2);
}

int main(int argc, char **argv)
{

    // create a scheduler object
    scheduler cfs_scheduler;

    char *ptr, *ptr1, buff[100];
    int num;
    ifstream file;

    //some book keeping
    if (argc < 2)
    {
        cout << "Error: Not enough arguments" << endl;
        exit(1);
    }

    file.open(argv[1]);            // open input file
    fout.open("output_file.txt"); //open output file

    for (std::string line; getline(file, line, '\n');)
    {

        //variables for string parsing
```

```
        int temp1, temp2;
        ptr = strcpy(buff, line.c_str());
        ptr = strtok(ptr, ":");
        num = atoi(ptr);

        //schedule jobs until counter = command time
        cfs_scheduler.syncTime(num);

        ptr = strtok(NULL, "(");

        // Insert new job
        if (strcmp(ptr + 1, "Insert") == 0)
        {
            ptr = strtok(NULL, ",");
            temp1 = atoi(ptr);
            ptr = strtok(NULL, ")");
            temp2 = atoi(ptr);
            cfs_scheduler.insert(temp1, temp2);
        }

        // print jobs
        else if (strcmp(ptr + 1, "PrintJob") == 0)
        {
            ptr = strtok(NULL, ")");
            int i, flag = 0;
            for (i = 0; i < strlen(ptr); i++)
            {
                if (ptr[i] == ',')
                {
                    flag = 1;
                    break;
                }
            }

            if (flag == 1)
            {
                // print range of jobs
                ptr1 = strtok(ptr, ",");
                ptr += strlen(ptr1) + 1;
                cfs_scheduler.printjob(atoi(ptr1), atoi(ptr));
            }
            else
            {
                // print single job
                cfs_scheduler.printjob(atoi(ptr));
            }
        }
        // print next job
```

```
        else if (strcmp(ptr + 1, "NextJob") == 0)
            cfs_scheduler.nextjob(atoi(strtok(NULL, ")")));

        // print previous job
        else if (strcmp(ptr + 1, "PreviousJob") == 0)
            cfs_scheduler.prevjob(atoi(strtok(NULL, ")")));
        //error case
        else
            fout << "unknown operation" << endl;
    }

    // End of input file. schedule all the pending jobs
    while (cfs_scheduler.ifjob())
    {
        cfs_scheduler.dispatch(100);
    }

    //close file handlers
    file.close();
    fout.close();

    return 0;
}
```

Heap Class

```
//106119100 Rajneesh Pandey

// constructor for heap class. allocates heap of size BUFFER_SIZE
heap::heap(int size){
    root = new heapnode[size];
    last = 0;
    size = BUFFER_SIZE;
}


void heap::execute(int x){
    // function to execute the jobs
    root[1].exec_time+=x;

    heapify();
}

// method to swap two jobs
void heap::swapJob(struct heapnode* a,struct heapnode* b){

    int jobID,exec_time,total_time;
    rbtnode *tmp1,*tmp2;
    // heapnode* tmp;
    jobID = a->jobID;
    a->jobID = b->jobID;
    b->jobID = jobID;

    exec_time = a->exec_time;
    a->exec_time = b->exec_time;
    b->exec_time = exec_time;

    total_time = a->total_time;
    a->total_time = b->total_time;
    b->total_time = total_time;

    tmp1 = a->twin;
    tmp2 = b->twin;

    tmp1->twin = b;
    tmp2->twin = a;

    tmp1 = a->twin;
    a->twin = b->twin;
    b->twin = tmp1;

    //yet to swap twin pointer
}
```

```
// insert new node into the heap
heapnode* heap::insert(int jobID,int exec_time,int total_time,rbt
node* p){

    int ptr = ++last;

    if(last==size){ // if heap is full, double the array
        size*=2;
        root = (struct heapnode*)realloc(root,sizeof(struct heapn
ode)*size);
    }

    root[last].jobID = jobID;
    root[last].exec_time = exec_time;
    root[last].total_time = total_time;
    root[last].twin = p;
    p->twin = &root[ptr];

    // balance after insert
    while(ptr>1){
        if(root[ptr].exec_time < root[ptr/2].exec_time){
            swapJob(&root[ptr],&root[ptr/2]);
            ptr = ptr/2;
        }
        else
            break;
    }
    return &root[ptr];
}

// used while scheduling
void heap::updateMin(int exec_time){
    root[1].exec_time = exec_time;
    heapify();
}


//regain heap properties after remove min
void heap::heapify(){
    int i=1,j;
    while(1){
        if(i*2>last )
            break;
        if(i*2+1>last)
            j=i*2;
        else
            j = root[i*2].exec_time < root[i*2+1].exec_time ? i*2
:i*2+1;
```

```
        if(root[j].exec_time < root[i].exec_time){
            swapJob(&root[i],&root[j]);
            i = j;
        }
        else
            break;

    }
}


//remove from top of the heap.
struct heapnode* heap::removeMin(){
    int i=1;

    struct heapnode *result = new heapnode;

    result->jobID = root[i].jobID;
    result->exec_time = root[i].exec_time;
    result->total_time = root[i].total_time;

    root[i].jobID = root[last].jobID;
    root[i].exec_time = root[last].exec_time;
    root[i].total_time = root[last].total_time;

    last--;
    heapify();
    return result;
}
```

```
//structure declaration for heap nodes.
struct heapnode{

    int jobID,exec_time,total_time;
    struct rbtnode* twin;
};


//Class of heap data structure

class heap{
    public:
    struct heapnode* root;
    int last; // keeps track of last element in the heap array
    int size; // keeps track of total size allocated for heap arr
ay

    heapnode* insert(int, int, int,rbtnode*); // insert a new nod
e into the heap.

    struct heapnode* removeMin(); // remove the item from top of
the heap ie minimum executed time.

    void swapJob(struct heapnode* a,struct heapnode* b); // swap
two nodes of the heap

    void heapify(); // regain heap property

    void updateMin(int exec_time); // update exec_time of root no
de and heapify the structure

    heap(int); // constructor declaration

    void execute(int); // execute a job

    ~heap(){ // destructor declaration
        delete root;
    }
};
```

RBT Class

```
//106119100 Rajneesh Pandey

//healper function for find prev
void findlower(rbtnode *root,int jobid,rbtnode* last, int* flag)
{
    if(*flag==1)
        return;
    if (root==NULL)
        return;

    findlower(root->left,jobid,last,flag);
    if(root->jobid ==jobid){
        if(last==NULL)
            return;
        heapnode* temp2 = last->twin;
        fout<<"("<<last->jobid<<","<<temp2-
>exec_time<<","<<temp2->total_time<<")"<<endl;
        *flag = 1;
    }
    last = root;
    findlower(root->right,jobid,last,flag);
}

//healper function for nextnode
void findhigher(rbtnode *root,int jobid, int* flag){
    if(*flag==1)
        return;
    if (root==NULL)
        return;

    findhigher(root->left,jobid,flag);
    if(root->jobid >jobid){
        heapnode* temp2 = root->twin;
        fout<<"("<<temp2->jobID<<","<<temp2-
>exec_time<<","<<temp2->total_time<<")"<<endl;
        *flag=1;
        return;
    }
    findhigher(root->right,jobid,flag);
}



// find smallest node larger than given job id
void rbt::nextnode(int jobid){
    int flag=0;
    findhigher(root,jobid,&flag);
```

```
    if(flag!=1)
        fout<<"(0,0,0)"<<endl;

}

// find largest node less than given jobid
void rbt::prevnode(int jobid){
    int flag=0;

    findlower(root,jobid,NULL,&flag);

    if(flag==0)
        fout<<"(0,0,0)"<<endl;

}



// helper function to find a node in the tree
rbtnode* findnodeHelper(rbtnode* root,int jobid){
    if(root==NULL)
        return NULL;
    else if(root->jobid == jobid)
        return root;
    else if(jobid < root->jobid)
        return findnodeHelper(root->left,jobid);
    else
        return findnodeHelper(root->right,jobid);
}

//method to find a node in the tree
rbtnode* rbt::findnode(int jobid){

    rbtnode* temp = findnodeHelper(root,jobid);
    if(temp==NULL)
        fout<<"(0,0,0)\n";
    else{
        heapnode* temp2 = temp->twin;
        fout<<"("<<temp2->jobID<<","<<temp2-
>exec_time<<","<<temp2->total_time<<")"<<endl;
    }
    return temp;
}


// recursive function to do in order traversal in range low->high
void inorderHelper(rbtnode *root,int low,int high,int *flag)
{
```

```
    if (root==NULL)
        return;
    if(root->jobid > low)
        inorderHelper(root->left,low,high,flag);
    if(root->jobid >= low && root->jobid <=high){
        if(*flag==0)
          *flag=1;
        else
          fout<<",";
        fout <<"("<< root->jobid <<",";

        heapnode* temp = root->twin;
        fout<<temp->exec_time<<","<<temp->total_time<<")";
    }
    if(root->jobid <high)
        inorderHelper(root->right,low,high,flag);
}


// function to insert a new node into the structure
rbtnode* rbtinsert(rbtnode* root, rbtnode *ptr){
    if (root == NULL)
       return ptr;

    if (ptr->jobid < root->jobid){
        root->left  = rbtinsert(root->left, ptr);
        root->left->parent = root;
    }
    else if (ptr->jobid > root->jobid){
        root->right = rbtinsert(root->right, ptr);
        root->right->parent = root;
    }
    return root;
    // fixing of properties is done from the rbt method
}

 //do a rotate left operation
void rbt::rotateleft(rbtnode *&root, rbtnode *&ptr){
    rbtnode *ptr_right = ptr->right;

    ptr->right = ptr_right->left;

    if (ptr->right != NULL)
        ptr->right->parent = ptr;

    ptr_right->parent = ptr->parent;

    if (ptr->parent == NULL)
```

```
        root = ptr_right;

    else if (ptr == ptr->parent->left)
        ptr->parent->left = ptr_right;

    else
        ptr->parent->right = ptr_right;
    ptr->parent = ptr_right;
    ptr_right->left = ptr;
}

// do a rotate right operation
void rbt::rotateright(rbtnode *&root, rbtnode *&ptr){
    rbtnode *ptr_left = ptr->left;
    ptr->left = ptr_left->right;

    if (ptr->left != NULL)
        ptr->left->parent = ptr;

    ptr_left->parent = ptr->parent;

    if (ptr->parent == NULL)
        root = ptr_left;

    else if (ptr == ptr->parent->left)
        ptr->parent->left = ptr_left;

    else
        ptr->parent->right = ptr_left;

    ptr_left->right = ptr;
    ptr->parent = ptr_left;
}


// function fixes rbt violations caused by bst insertion
void rbt::fixtree(rbtnode *&root, rbtnode *&ptr){
    rbtnode *parent_ptr = NULL;
    rbtnode *grand_parent_ptr = NULL;

    while ((ptr != root) && (ptr->color != BLACK) && (ptr-
>parent->color == RED)){

        grand_parent_ptr = ptr->parent->parent;
        parent_ptr = ptr->parent;

        // when X=L
        if (parent_ptr == grand_parent_ptr->left){
```

```
            rbtnode *uncle_ptr = grand_parent_ptr->right;

            // when r = red ie uncle red
            if (uncle_ptr != NULL && uncle_ptr->color == RED){
                parent_ptr->color = BLACK;
                grand_parent_ptr->color = RED;
                uncle_ptr->color = BLACK;
                ptr = grand_parent_ptr;
            }

            else{
                // when Y = R
                if (ptr == parent_ptr->right){
                    rotateleft(root, parent_ptr);
                    ptr = parent_ptr;
                    parent_ptr = ptr->parent;
                }

                // when Y = L
                rotateright(root, grand_parent_ptr);
                {
                    bool clr;
                    clr = parent_ptr->color;
                    parent_ptr->color = grand_parent_ptr->color;
                    grand_parent_ptr->color = clr;
                }
                ptr = parent_ptr;
            }
        }

        // when X = R
        else{
            rbtnode *uncle_ptr = grand_parent_ptr->left;

            //when r=red
            if ((uncle_ptr != NULL) && (uncle_ptr-
>color == RED)){
                parent_ptr->color = BLACK;
                uncle_ptr->color = BLACK;
                grand_parent_ptr->color = RED;
                ptr = grand_parent_ptr;
            }
            else{
                //when Y = L
                if (ptr == parent_ptr->left){
                    rotateright(root, parent_ptr);
                    ptr = parent_ptr;
```

```
                    parent_ptr = ptr->parent;
                }
                //when Y = R
                rotateleft(root, grand_parent_ptr);
                {
                    bool clr;
                    clr = parent_ptr->color;
                    parent_ptr->color = grand_parent_ptr->color;
                    grand_parent_ptr->color = clr;
                }

                ptr = parent_ptr;
            }
        }
    }
    root->color = BLACK;
}


// rbt method to insert a new node
rbtnode* rbt::insert(const int &jobid){
    rbtnode *ptr = new rbtnode(jobid);
    rbtnode* bk = ptr;
   // bst insert
    root = rbtinsert(root, ptr);

    return bk;
    //violations are fixed from the main program
}

// Function in order traversal in range low to high

void rbt::inorder(int low,int high){
    int flag=0;
    inorderHelper(root,low,high,&flag);
    if(flag==0)
      fout<<"(0,0,0)";
    fout<<endl;
}




rbtnode* successor(rbtnode *node)
{
      rbtnode *k=NULL;
     if(node->left!=NULL)
     {
```

```
            k=node->left;
            while(k->right!=NULL)
                 k=k->right;
        }
        else
        {
            k=node->right;
            while(k->left!=NULL)
                 k=k->left;
        }
        return k;

}


void rbt::deletenode(rbtnode* p){

        p=root;
        rbtnode *k=NULL,*y=NULL, *q=NULL;


        if(p->left==NULL||p->right==NULL)
             y=p;
        else
             y=successor(p);

        if(y->left!=NULL)
             q=y->left;
        else
        {
             if(y->right!=NULL)
                  q=y->right;
             else
                  q=NULL;
        }
        if(q!=NULL)
             q->parent=y->parent;
        if(y->parent==NULL)
             root=q;
        else
        {
            if(y==y->parent->left)
               y->parent->left=q;
            else
               y->parent->right=q;
        }
        if(y!=p)
        {
```

```
            p->color=y->color;
            p->jobid=y->jobid;
            p->twin = y->twin;

            struct heapnode* temp;
            temp = p->twin;
            temp->twin = p;

       }
       if(k!=NULL)
            fixviolation(q);

}

//left rotate after delete
void rbt::leftrotate(rbtnode *p)
{
       if(p->right==NULL)
              return ;
       else
       {
              rbtnode *y=p->right;
              if(y->left!=NULL)
              {
                       p->right=y->left;
                       y->left->parent=p;
              }
              else
                       p->right=NULL;
              if(p->parent!=NULL)
                     y->parent=p->parent;
              if(p->parent==NULL)
                     root=y;
              else
              {
                   if(p==p->parent->left)
                            p->parent->left=y;
                   else
                            p->parent->right=y;
              }
              y->left=p;
              p->parent=y;
       }
}
//right rotation after delete
void rbt::rightrotate(rbtnode *p)
{
       if(p->left==NULL)
```

```
          return ;
     else
     {
         rbtnode *y=p->left;
         if(y->right!=NULL)
         {
                  p->left=y->right;
                  y->right->parent=p;
         }
         else
                 p->left=NULL;
         if(p->parent!=NULL)
                 y->parent=p->parent;
         if(p->parent==NULL)
               root=y;
         else
         {
             if(p==p->parent->left)
                   p->parent->left=y;
             else
                   p->parent->right=y;
         }
         y->right=p;
         p->parent=y;
     }
}


//function to fix violation after delete
void rbt::fixviolation(rbtnode *p)
{
    rbtnode *s;
    while(p!=root&&p->color==BLACK){
          if(p->parent->left==p){
                  s=p->parent->right;
                  if(s->color==RED){
                         s->color=BLACK;
                         p->parent->color=RED;
                         leftrotate(p->parent);
                         s=p->parent->right;
                  }
                  if(s->right->color==BLACK&&s->left-
>color==BLACK){
                         s->color=RED;
                         p=p->parent;
                  }
                  else{
                      if(s->right->color==BLACK){
```

```
                             s->left->color=BLACK;
                             s->color=RED;
                             rightrotate(s);
                             s=p->parent->right;
                      }
                      s->color=p->parent->color;
                      p->parent->color=BLACK;
                      s->right->color=BLACK;
                      leftrotate(p->parent);
                      p=root;
                  }
          }
          else{
                  s=p->parent->left;
                  if(s->color==RED){
                        s->color=BLACK;
                        p->parent->color=RED;
                        rightrotate(p->parent);
                        s=p->parent->left;
                  }
                  if(s->left->color==BLACK&&s->right-
>color==BLACK){
                        s->color=RED;
                        p=p->parent;
                  }
                  else{
                        if(s->left->color==BLACK){
                              s->right->color=BLACK;
                              s->color=RED;
                              leftrotate(s);
                              s=p->parent->left;
                        }
                        s->color=p->parent->color;
                        p->parent->color=BLACK;
                        s->left->color=BLACK;
                        rightrotate(p->parent);
                        p=root;
                  }
          }
       p->color=BLACK;
       root->color=BLACK;
    }
}
```

```
enum color {RED, BLACK};

struct rbtnode{

    int jobid;
    bool color;
    rbtnode *left, *right, *parent;
    heapnode* twin;

    rbtnode(int jobid){
       this->jobid = jobid;
       parent=left=right=NULL;
    }

};


// class of redblack tree
class rbt{

private:
    rbtnode *root;
protected:

    void rotateleft(rbtnode *&, rbtnode *&); // rotate left opera
tion of rbt
    void rotateright(rbtnode *&, rbtnode *&); // rotate right ope
ration of rbt
    void fixtree(rbtnode *&, rbtnode *&); // fix the tree to rega
in rbt properties
public:
    rbt(); // constructor
    rbtnode* insert(const int &n); // insert new node into the tr
ee
    rbtnode* findnode(int jobid); // find a node in the tree
    void nextnode(int jobid); // find the next lowest node wrt in
order traversal
    void prevnode(int jobid); // find the previous largest node w
rt inorder traversal

    void inorder(int, int); // find a range of job id's in range
low,high

    void deletenode(rbtnode*); // delete a node from tree

    void fixviolation(rbtnode*); // fix violation caused due to d
elete
```

```
    void rightrotate(rbtnode *p); // rotate right after delete
    void leftrotate(rbtnode *p); // rotate left after elete

};

//constructor
rbt::rbt(){
    root = NULL;
}
```

Input / Output

Input format

Insert(jobID,total_time)
PrintJob(jobID)
PrintJob(jobID1,jobID2)
NextJob(jobID)
PreviousJob(jobID)

1. Input / Output

sample_input1.txt

```
0: Insert(50,200)
30: Insert(19,472)
90: Insert(30,300)
200: PrintJob(19)
240: Insert(1250,142)
260: PrintJob(10,500)
263: Insert(3455,450)
270: NextJob(1250)
349: PreviousJob(1250)
400: Insert(60,140)
412: Insert(1,230)
467: Insert(96,12)
512: Insert(55,534)
520: Insert(455,987)
560: NextJob(3455)
600: PreviousJob(55)
630: PrintJob(55)
680: PrintJob(30)
720: Insert(33,300)
750: PrintJob(120,1200)
```

output_file.txt

```
(19,70,472)
(19,80,472),(30,80,300),(50,80,200)
(3455,7,450)
(50,80,200)
(0,0,0)
(30,80,300)
(55,53,534)
(30,80,300)
(455,75,987)
```

2. Input / Output

sample_input2.txt

```
0: Insert(50,60000)
49950: Insert(19,55000)
99950: Insert(30,58000)
145900: PrintJob(19)
199500: Insert(1250,47000)
250000: PrintJob(10,520)
299600: Insert(3455,61000)
349000: NextJob(1250)
490000: PreviousJob(1250)
536000: Insert(60,49143)
599000: Insert(1,56748)
645578: Insert(96,63210)
645590: PrintJob(55)
699599: PrintJob(30)
745798: Insert(33,30030)
899700: PrintJob(120,1200)
```

output_file.txt

```
(19,49975,55000)
(0,0,0)
(3455,49400,61000)
(0,0,0)
(0,0,0)
(0,0,0)
(0,0,0)
```

3. Input / Output

sample_input3.txt

```
0: Insert(50,60000)
49950: Insert(19,55000)
99950: Insert(30,58000)
125900: PrintJob(19)
199500: Insert(1250,47000)
229500: Insert(1350,37000)
230000: PrintJob(30,5200)
235000: NextJob(1350)
236000: PreviousJob(1350)
```

output_file.txt

```
(19,49975,55000)
(1250,30000,47000),(1350,500,37000)
(0,0,0)
(1250,30000,47000)
```

4. Input / Output

sample_input4.txt

```
0: Insert(50,550)
10: Insert(19,550)
15: Insert(20,550)
20: Insert(21,550)
25: Insert(23,550)
30: Insert(25,550)
10000: Insert(26,550)
```

output_file.txt

```

```

No Output in this input