```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>


typedef struct _process
{
    int pid;
    int size; // how big is process
    char name; // keep track of name
    int tickets;
    struct _process *next;

} Process; //name of struct

typedef struct _queue
{
    Process *first;
    Process *last;
} Queue; // will hold  the processes and keep track of pointers

// when a new process shows up, the new process needs to be last

Queue* createQueue()
{
    Queue* newQue = malloc(sizeof(Queue)); //malloc will look to find enough space
to fit for queue and return an address
    newQue->first = NULL;
    newQue->last = NULL;
    return newQue;
}


Process* createProcess(int process_id, int process_size, char process_name, int
process_tickets)
{
    Process* newProcess = malloc(sizeof(Process));
    newProcess->pid = process_id;
    newProcess->size = process_size;
    newProcess->name = process_name;
    newProcess->tickets = process_tickets;
    newProcess->next = NULL;
    return newProcess;
}

int addToQueueLast(Queue* q, Process* p)
{
    if(q == NULL)
    {
        return -1;
    }
    if(p== NULL)
    {
        return -1;
    }
    p->next = NULL;
    if(q->last == NULL)
    {
```

```
        q->last = p;
        q->first = p;
        return 0; // sends what's in here and what's outside the conditional
statement
    }
    q->last->next = p;
    q->last=p;
    return 0;
}

Queue* cloneQueue(Queue* original) // makes duplicate of original queue to be able
to reuse
{
    Queue* newQueue = createQueue();
    Process* temp = original->first;
    while (temp) {
        addToQueueLast(newQueue, createProcess(temp->pid, temp->size, temp->name,
temp->tickets));
        temp = temp->next;
    }
    return newQueue;
}

void freeQueue(Queue* q) // frees allocated memory for processes and the queue
entirely
{
    Process* temp;
    while (q->first) {
        temp = q->first;
        q->first = q->first->next;
        free(temp);
    }
    free(q);
}

Process* findProcessByName(Queue* q, char name) {
    Process* temp = q->first;
    while (temp) {
        if (temp->name == name) {
            return temp;
        }
        temp = temp->next;
    }
    printf("%c could not be found", name);
    return NULL;
}

int getTotalTickets(Queue* q)
{
    int total = 0;
    Process* temp = q->first;
    while (temp)
    {
        total += temp->tickets;
        temp = temp->next;
    }
    return total;
}
```

```
Process* runLottery(Queue* q)
{
    if (q == NULL || q->first == NULL)
        return NULL;

    int totalTickets = getTotalTickets(q);
    if (totalTickets == 0)
        return NULL; // Avoid division by zero

    int winningTicket = rand() % totalTickets;
    int ticketCount = 0;

    Process* temp = q->first;
    while (temp) {
        ticketCount += temp->tickets;
        if (winningTicket < ticketCount)
        {
            return temp; // Winner found
        }
        temp = temp->next;
    }
    return NULL; // Shouldn't reach here
}

void transferTickets(Queue* q, char donorname, char receivername,int ticketnum)
{
    Process* donor = findProcessByName(q, donorname);
    Process* receiver = findProcessByName(q, receivername);

    if (donor == NULL || receiver == NULL)
    {
        printf("\n Error: One or both processes not found for ticket transfer\n");
        return;
    }

    if (donor->tickets >= ticketnum)
    {
        donor->tickets -= ticketnum;
        receiver->tickets += ticketnum;
        printf("\n Transferred %d tickets from %c to %c\n", ticketnum, donor->name,
receiver->name);
    }
    else
    {
        printf("\n Error: Not enough tickets to transfer\n");
    }
}

void inflateTickets(Queue* q, char targetName, int extratickets) {
    Process* p = findProcessByName(q, targetName);
    if (p == NULL)
    {
        printf("\n Error: Process %c not found for ticket inflation.\n",
targetName);
        return;
    }
    p->tickets += extratickets;
    printf("\n Inflated tickets for %c by %d\n", p->name, extratickets);
}
```

```
void manageQueue(Queue* q, Process* p, bool removeOnly)
{
    if (q == NULL || p == NULL)
        return;

    if (q->first == p) // handles the case that the process is first in queue
    {
        q->first = p->next;
        if (q->last == p)
        {
            q->last = NULL; // If it's also the last process, set last to NULL
        }
    }
    else
    {
        Process* prev = q->first;
        while (prev->next != p)
            prev = prev->next; // Finds the previous node of the one that needs to
be removed (p)
        prev->next = p->next; // sets p's previous next to it's own next

        if (q->last == p)
        {
            q->last = prev; // sets the last element in q to p's previous node
        }
    }

    if (!removeOnly) // process is not done, add back into queue
    {
        addToQueueLast(q, p);
    }
}

int lotteryScheduler(Queue* q, int decrement) {
    if (q == NULL) return -1;

    while (getTotalTickets(q) > 0)
    {
        Process* p = runLottery(q);
        if (p == NULL)
        {
            continue;
        }

        printf(" Process %c is running: size = %d, tickets = %d\n", p->name, p-
>size, p->tickets);
        p->size -= decrement;

        if (p->size <= 0)
        {
            printf(" Process %c complete!\n", p->name);
            manageQueue(q, p, true); // Remove the process entirely
            free(p); // Free the completed process
        } else
        {
            manageQueue(q, p, false); // process not complete, remove and re-add
the process
```

```
        }
    }
    return 0;
}

int main()
{
    srand(time(NULL)); // initialize random generator with seed that changes every
second
    Queue* originalq=createQueue();
    //Process* pA = createProcess(1,5, 'A');
    addToQueueLast(originalq,createProcess(1,5,'A',10));
    addToQueueLast(originalq,createProcess(2,3,'B',5));
    addToQueueLast(originalq,createProcess(3,7,'C',20));

    Queue* q1 = cloneQueue(originalq);
    Process* temp = q1->first;
    printf("\n Pre-Scheduling Details:\n");
    while(temp)
    {
        printf(" Process %c     Size: %d    Tickets: %d\n", temp->name,temp-
>size,temp->tickets);
        temp = temp->next;
    }
    printf("\n");
    int c = lotteryScheduler(q1,3);
    freeQueue(q1); // Free memory after use
    printf("\nScheduling Completed!\n\n");

    Queue* q2 = cloneQueue(originalq);
    transferTickets(q2,'A','B', 5);
    inflateTickets(q2,'C', 10);
    temp = q2->first;
    printf("\n Pre-Scheduling Details:\n");
    while(temp)
    {
        printf(" Process %c     Size: %d    Tickets: %d\n", temp->name,temp-
>size,temp->tickets);
        temp = temp->next;
    }
    printf("\n");
    lotteryScheduler(q2, 5);
    freeQueue(q2);
    printf("\nScheduling Completed!\n\n");

    return 0;
}



// malloc - allocate memory, request a block of memory from OS
//      - returns pointer to beginning of allocated memory block, else return NULL
//      - allocates the number bytes on one segmnet and returns a pointer
//      - reserves a block of memory for the program
//      - sizeof() - requesting a specific size in bytes of the parameter
// struct creates a type
// free() - deallocate memory that was previously allocated
// srand() - seeds random number generator rand()
// time() - returns the current time
```

```
//      - time(NULL) - don't want to store the result in a variable, just return
value directly
```