```
#include <stdio.h>
#include <conio.h>
int Q[10],front=-1,rear=-1,n,run=-1;
// Pushing process in the RR queue
void Push(int p_id)
{
     if(front ==-1)  // Empty queue
     {
    //Make new process as the first process in Q
           front=rear=0;
           Q[front]=p_id;
     }
     else
     {
    // Insert new process at the end of Q behind existing processes
           rear++;
           Q[rear]=p_id;
     }
}
// Choosing the next process to execute
int Pop()
{
     int val,i;
     if(rear==-1) //Queue empty
          return -1;
     else if(rear==0) // Single element in queue
     {
          val=Q[front];
          front=rear=-1;
          return val;
     }
     else //More than one element in queue
     {
    // Retrieve the first process from front
          val=Q[front];
    // Rearrange the remaining elements by shifting the one position to their
left
          {
              for(i=0;i<=rear-1;i++)
                 Q[i]=Q[i+1];
          }
          rear--;
          return val;
     }
}
struct process
{
      int at;
      int st;
      int pr;
      // Whether a process completed or not, 0 - not completed and 1 - completed
      int status;
      int ft;
      // A new process should be entered into the RR queue only once when it
arrives for the first time
    int q;
}RR_QUEUE[10];

int dispatcher(int time)
{

      int i,pid;
      for(i=0;i<n;i++)
      {
```

```
      // Process already completed or not
        if(RR_QUEUE[i].status != 1)
          {
      // Process's arrival time within current time and it was not the running
process during the precious time quantum
            if(RR_QUEUE[i].at <= time && run !=i)
                {
      // The process has not been entered into the RR queue already
                {
                if(RR_QUEUE[i].q==0)
                     Push(i);  // Enters into RR queue
      // Prevent the process from getting reentered again
                     RR_QUEUE[i].q=1;
                }
            }

         }
      }
      // The process that was executed during the last time quantum must join at
the end of the RR queue
      if(run!=-1)
    {
               Push(run);
    // To prevent the entry of the same process multiple times during the
remaining iterations of the loop
               run=-1;
    }
     // Pick the first process in the RR queue for exeution
      pid=Pop();
      return pid;
}


int main()
{
      int i,cur_time,pid, tq;
      printf("Enter number of processes:");
      scanf("%d",&n);
      printf("Enter the time quantum:");
      scanf("%d",&tq);
      for(i=0;i<n;i++)
      {
            printf("Process %d\n",i+1);
            printf("***********\n");
            printf("Enter Arrival Time:");
            scanf("%d",&RR_QUEUE[i].at);
            printf("Enter Service Time:");
            scanf("%d",&RR_QUEUE[i].st);
         RR_QUEUE[i].status=0;
      // whether the process already entred into RR queue or not
            RR_QUEUE[i].q=0;
      }
      i=0; cur_time=0;
      while(i < n)
      {
             pid=dispatcher(cur_time);

             if(pid!=-1)
             {
      // The chosen process becomes the currently running process
              run=pid;
      // Process's pending service time is more than time quantum
              if(RR_QUEUE[pid].st > tq)
            {
```

```
                RR_QUEUE[pid].st=RR_QUEUE[pid].st - tq;
                cur_time=cur_time+tq;
            }
    // Process's pending service time is exactly equal to time quantum
            else if(RR_QUEUE[pid].st == tq)
            {
                RR_QUEUE[pid].st=0;
                cur_time=cur_time+tq;
                RR_QUEUE[pid].ft=cur_time;
                RR_QUEUE[pid].status=1;
    // Since the current process is completed it should not be added into the RR
queue again by the dispatcher
                run=-1;
                i++;
            }
    // Process's pending service time is less than time quantum
            else
            {
                cur_time=cur_time+RR_QUEUE[pid].st;
                RR_QUEUE[pid].st=0;
                RR_QUEUE[pid].ft=cur_time;
                RR_QUEUE[pid].status=1;
    // Since the current process is completed it should not be added into the RR
queue again by the dispatcher
                run=-1;
                i++;
            }
        }
    // No process is available at the moment and CPU gets idle
         else
             cur_time++;
      }
      printf("\n");
      printf("Process\t Arrival Time\t Service Time\t Finish Time\n");
      printf("*******\t ************\t ************\t ***********\n");
      for(i=0;i<n;i++)
      {
            printf("%d\t\t%d\t\t%d\t\t\t%d\
n",i,RR_QUEUE[i].at,RR_QUEUE[i].st,RR_QUEUE[i].ft);
      }
      getch();
}
```