The Deque ADT for Round Robin Scheduling:

A deque, short for "double-ended queue," is an abstract data type that allows insertion and deletion of elements from both the front and the back of the queue. It is similar to a stack or a queue, but with the added flexibility of allowing operations on both ends. Deques are useful in many situations where elements need to be inserted or deleted from both ends, such as implementing a sliding window network protocol, browser history caching, task scheduling, text editor caching for redo and undo operations.

In this mock test, we will build 2 implementations of the deque ADT for CPU processes and later use it to visualize the Round Robin scheduling algorithm.

Your task is to implement the Deque ADT for CPU processes in C as follows:

1) **linked_deque** data structure: a deque that is internally backed by a doubly linked list and allows for constant time insertions and deletions at both ends.
2) **array_deque** data structure: a deque that is internally backed by a resizable circular array buffer and allows for constant time insertions and deletions at both ends "on average" (i.e the resize operation is rare enough so that on average the time to access the ends is constant).

Question 1) Start by implementing the functions defined in linkedlist.h in the corresponding linkedlist.c file. The `print_linked_list()`, `create_node_for_process()` functions have already been completed for you. Remember to increment and decrement the size field when inserting or deleting from the linked list. Also remember to avoid memory leaks in the `remove()` functions.

Now, test your implementation via the linkedlisttest.c file. Feel free to modify it as you wish to add your own test cases. Use the following command to obtain the executable and run it in one line:

```
gcc linkedlist.c linkedlisttest.c -o test && ./test
```

[Expected Time Q1: 30 minutes]

Question 2) Now complete the `linked_deque` data structure by completing the functions through simple function calls to the underlying `linkedlist`. The code for none of the functions except `create_linked_process_deque()` exceeds a single line.

Now, test your implementation via the linkeddequetest.c file. Feel free to modify it as you wish. Use the following command to obtain the executable and run it in one line:

```
gcc linked_deque.c linkedlist.c linkeddequetest.c -o test2 &&
./test2
```

[Expected Time Q2: 10 minutes]

Question 3) Complete the **Round Robin Process visualizer** using the linked_deque implementation.
Algorithm: A process can enter into the deque from one of the ends and can be removed only for the other end of deque. The removed process should execute for a given time quantum and then should be inserted back into the deque after the quantum expires.

Take a few minutes to go through and trace the following example. Observe how Round Robin Scheduling does not allow a process to execute for more than the allocated TIME_QUANTUM. If the

process completes its execution before the current TIME_QUANTUM (TQ) expires, the scheduler immediately starts the next process instead of waiting for the TQ to expire.

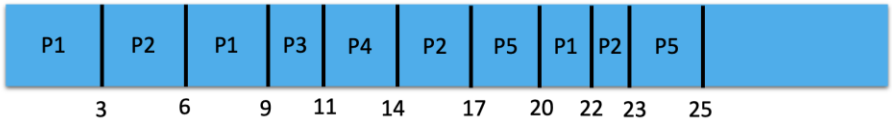
Also note that while a process is executing in its allocated time quantum, other processes may be added to the deque from the other end. In case a process finishes at the same time as another one arrives; the scheduler sends the new process into the queue first.

The same example is present in the file sample.txt in the format: process_name, process_id, arrival_time, cpu_burst. TQ is assumed to be 3.

Process Name	Process ID	Arrival Time	CPU Burst
P1	1	0	8
P2	2	1	7
P3	3	5	2
P4	4	6	3
P5	5	8	5

At t = 6, P2 finished its TQ and P4 arrived.
In such a situation P4 is added first to the deque
And P2, that has not yet completed,
is added just after that.

Round Robin Scheduling Timeline:



Turnaround time for P3 = Completion Time – Arrival Time = 11 – 5 = 6
Wait Time for P3 = Turnaround Time – Burst time = 6 – 2 = 4

`process struct` fields:

- `name`: the name of the process
- `pid`: the unique number associated with a process.
- `arrival`: the time at which the process arrives.
- `cpu_burst`: the time the process needs from the CPU to execute its instructions.
  Suppose the TQ is 3 and process has cpu_burst 6, then it will need 2 TQs from the Round Robin Scheduler
- `turnaround`: instant at which the process finished – arrival time.
- `wait`: time spent by the process idly waiting in the deque for its turn.
  = turnaround – burst
- `remaining_time`: the scheduler sets this field to determine if should put a process back in the deque from the other end. Scheduler only puts a process back in the deque if remaining time > 0.

As soon as a process's remaining time = 0, the scheduler calls the "print_stats" function to print that process. That process won't be added to the deque again.

The file handling part has already been done for you and the processes array is available in the visualize() function. Assume the processes array is sorted by arrival_time (lower first). Feel free to declare & define other static helper functions in the scheduler.c file. Now, test your simulation via the main.c & scheduler.c files. Use the following command to obtain the executable and run it in one line:

```
gcc scheduler.c main.c linked_deque.c linkedlist.c  -o ./test4 &&
./test4
```

Expected Output for sample.txt processes.

```
Number of processes: 5
Process   Arrival Time   Burst Time     Waiting Time   Turnaround Time
P3        5              2              4              6
P4        6              3              5              8
P1        0              8              14             22
P2        1              7              15             22
P5        8              5              12             17
```

[Expected Time Q3: 40 minutes]

Question 4) Now, complete the array_deque data structure which relies on a circular array buffer internally that doubles in size every time the deque is full in capacity and halves in size every time it's less than half full but greater than minimum capacity (`INITIAL_SIZE_ARRAY_DEQUE`). Ensure that all unfilled elements are set to 0 in the processes array so that there are no dangling pointers. Also ensure that you use calloc() during resize and initialization because it sets the fields in the allocated memory to 0 by default.

Now, test your implementation via the arraydequetest.c file. Feel free to modify it as you wish. Use the following command to obtain the executable and run it in one line:

```
gcc array_deque.c arraydequetest.c -o test5 && ./test5
```

[Expected Time Q4: 60 minutes]

Question 5) Test the scheduler works as expected by replacing linked_deque in your code with array_deque in the scheduler code. Which implementation (linked_deque or array_deque) do you believe achieves better performance and why?
[Optional] Test your hypothesis by measuring the time taken to execute the round robin scheduler in both cases using a randomly generated large test case and replacing the file_name in main() function.