**Round Robin Scheduler**

Just like when you played youth sports and played each other team in a division, a “round robin” is a way to divide equally resources. For a Round Robin Tournament, you give each team equal chance to win the greatest number of games in order to be named the victor. In computational use, round robins are often used to ensure that can be used to ensure that certain computational resources are equitably divided.

Often, when you have large jobs, you will send them to a server and use/purchase “CPU time” to run those jobs. Let’s say that three jobs, each taking an hour, arrive to the server at 6am. What the round robin scheduler does is to allow each job to use a specified amount of CPU time before moving onto the next job in the list for the same specified amount of time. This ensures that jobs that enter at the same time will, more or less, finish around the sa9792me time. If you have a scenario where jobs will not overlap in their CPU time needs, then the earlier jobs are allowed to finish as they won’t hold up any other jobs. Stated differently, jobs that arrive during a “busy” time have to share CPU time, but jobs that arrive during a “slow” time will not.

It turns out that linked lists provide a simple implementation for the algorithm described above. Specifically, circularly doubly-linked lists. During this project, you will be writing two classes: DoubleCircularList.java and DoubleNode.java. Each should be *generic* (more on this during class).

A DoubleNode<T> object consists of:

* A DoubleNode<T> previous
* A DoubleNode<T> next
* T data

DoubleCircularList.java will implement the DCListOps.java interface. Specifically, it should include the following methods:

|  |  |
| --- | --- |
| Method | Description |
| void add(T item); | Places a new object at the end of the list. The end of the list is defined as the node immediately preceding the current node. |
| T removeCurrent(); | Removes the current node, moves the current pointer to the next node, and returns a reference to the recently removed node. If the list is empty, this should return null. If there is only one item in the list, it should be removed and current should refer to null. |
| void next(); | Simply moves the current pointer to the next object in the list. If there is only one item in the list or the list is empty, nothing happens. |
| T getCurrent(); | Returns a reference to the current node. It **does not** remove the node from the list or move the current pointer. If the list is empty, null is returned. |
| boolean isEmpty(); | Returns true if the list is empty, false otherwise. |
| void printList(); | Goes through every node in the list, retrieves the data stored in the node, and calls the data's *toString()* method. It should always start with the current node and end with the last node in the list (thus, the ordering will change depending on where the current reference is). If the list is empty, nothing should get printed. This method should contain a line that looks something like this (it does not have to look exactly like this):  *System.out.println(tmp.data);*  This line should obviously be in some type of a loop. |