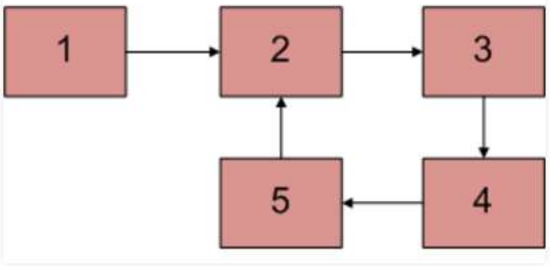
1. Write a program function to detect loop in a linked list

Given a linked list, check if the the linked list has loop or not. Below diagram shows a linked list with a loop.



2. Implement two stacks in an array

Create a data structure *twoStacks* that represents two stacks. Implementation of *twoStacks* should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

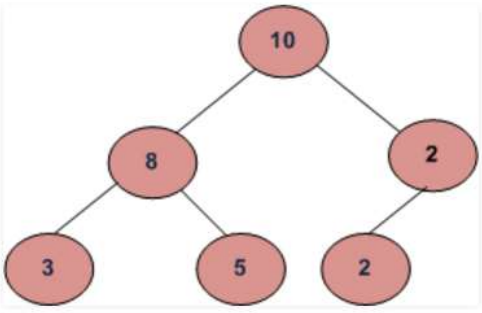
push1(int x) –> pushes x to first stack
push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element
pop2() –> pops an element from second stack and return the popped element

Implementation of *twoStack* should be space efficient.

3. Given a binary tree, print all root-to-leaf paths

For the below example tree, all root-to-leaf paths are:
10 –> 8 –> 3
10 –> 8 –> 5
10 –> 2 –> 2



Algorithm:
Use a path array path[] to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array path[]. When we reach a leaf node, print the path array.

4. Implement LRU Cache

How to implement LRU caching scheme? What data structures should be used?

We are given total possible page numbers that can be referred. We are also given cache (or memory) size (Number of page frames that cache can hold at a time). The LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which

Is not there in the cache.

We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size).The most recently used pages will be near front end and least recently pages will be near rear end.
2. **A Hash** with page number as key and address of the corresponding queue node as value.

When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue.

If the required page is not in the memory, we bring that in memory. In simple words, we add a new node to the front of the queue and update the corresponding node address in the hash. If the queue is full, i.e. all the frames are full, we remove a node from the rear of queue, and add the new node to the front of queue.

Note: Initially no page is in the memory.

5. Process an XML file using scala.

Process the **music.xml** file and display the data in an appropriate data structure so that we can fetch all songs for a given album for a given artist.