Linked List

Data Structures – CS2001 – Fall 2021

Dr. Syed Ali Raza

School of Computer Science

National University of Computing & Emerging Sciences

Karachi Campus

Outlines

- Arrays limitation.
- Liked list definition.
- Linked list types.
  - Singly linked list SLL
  - Doubly DLL
  - Circular Lists linked list CLL
  - Skip list and self organizing list

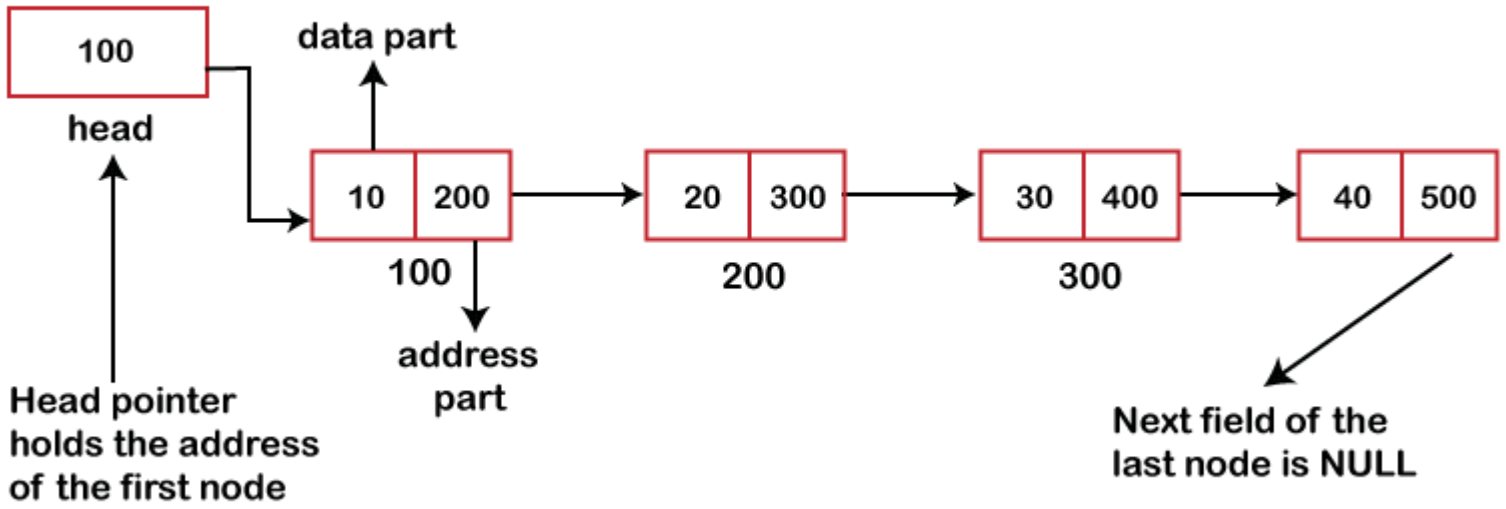
Linked list

Array limitations:

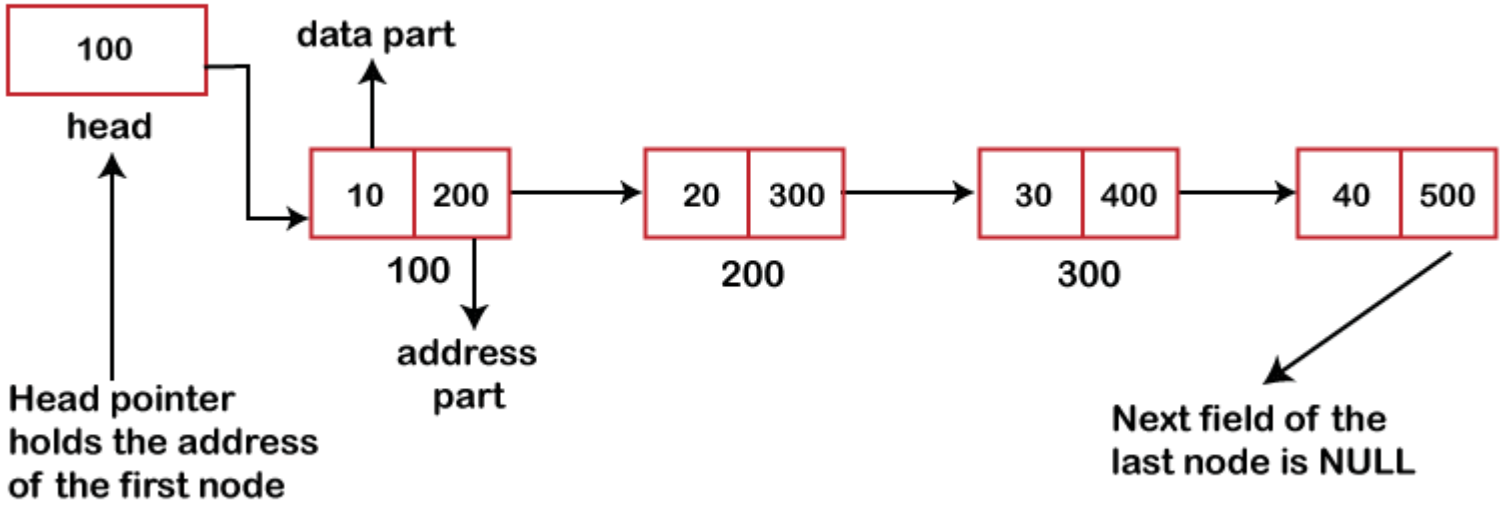
- Insertions and/or removals are expensive.
- Changing the size of the array requires creating a new array and then copying all data from the array with the old size to the array with the new size.
- The data in the array are next to each other sequentially in memory, which means that inserting an item inside the array requires shifting some other data in this array.
- This limitation can be overcome by using linked structures.

A linked list

- A linked structure is a collection of nodes storing data and links to other nodes.
- A linked list consists of:
  - A sequence of nodes



A linked list



Each node contains a value
and a link (pointer or reference) to some other node

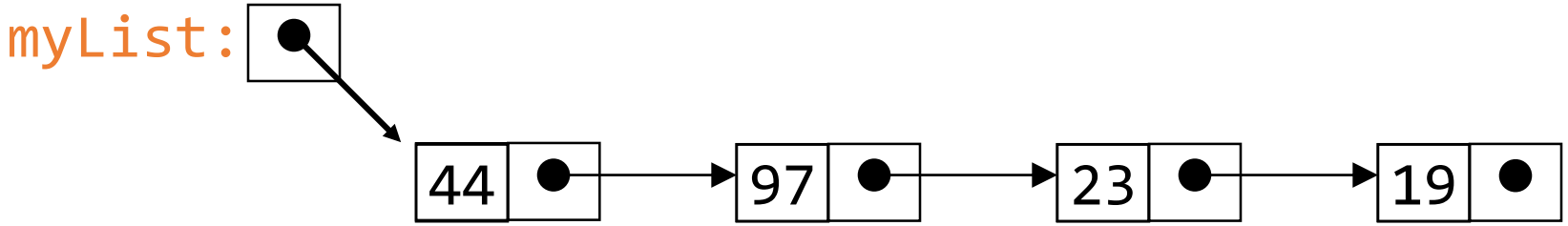
The last node contains a null link

The list may (or may not) have a header

More terminology

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

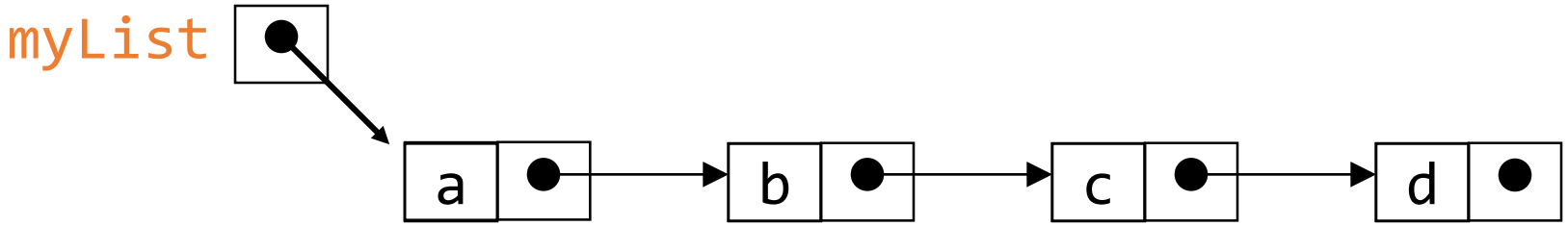
Creating Node Class




```
class Node{
      public:
      int data;
      Node* next;
       Node () {
              data = 0;
              next = NULL;
      }
      Node (int d, Node* n){
              data = d;
              next = n;
      }
};
```

Singly-linked lists

- singly-linked list (SLL):

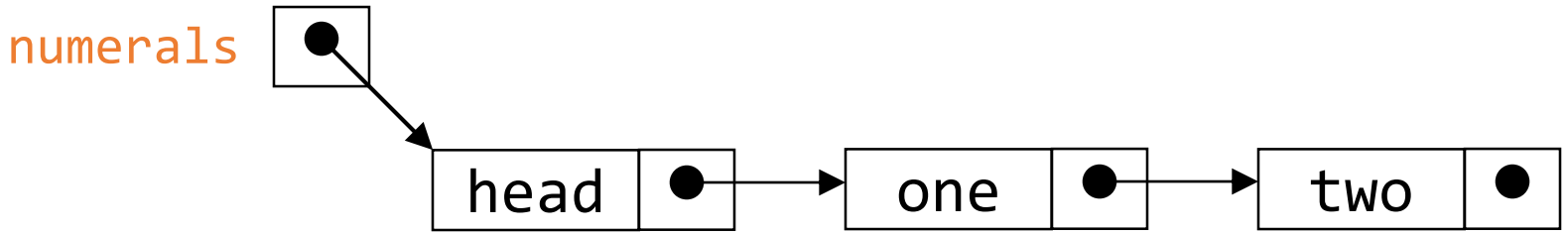


- A singly linked list is a data structure composed of nodes, each node holding some information and a reference to another node in the list. a node has a link only to its successor in

this sequence.

Using a header node

- A header node is just an initial node that exists at the front of every list, even when the list is empty
- The purpose is to keep the list from being null, and to point at the first element

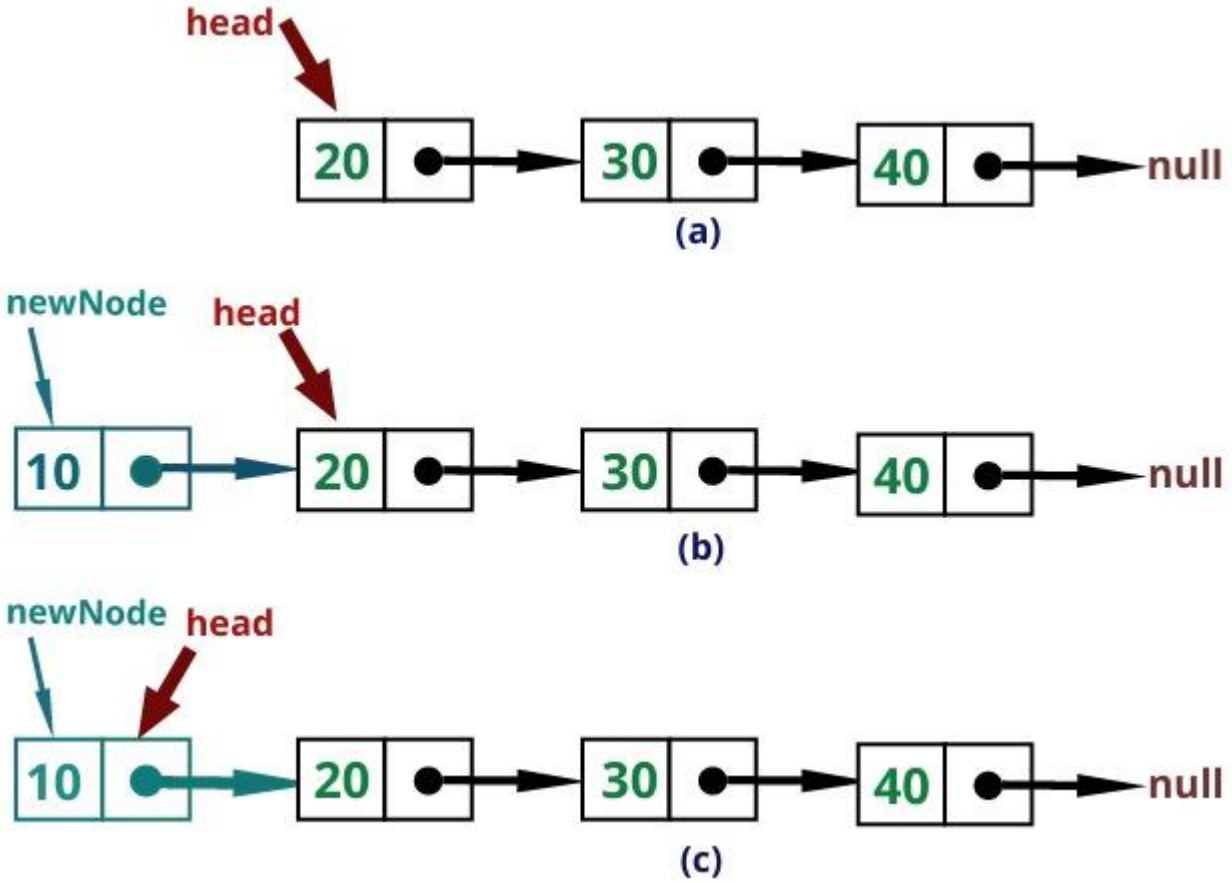


Inserting a node into a SLL

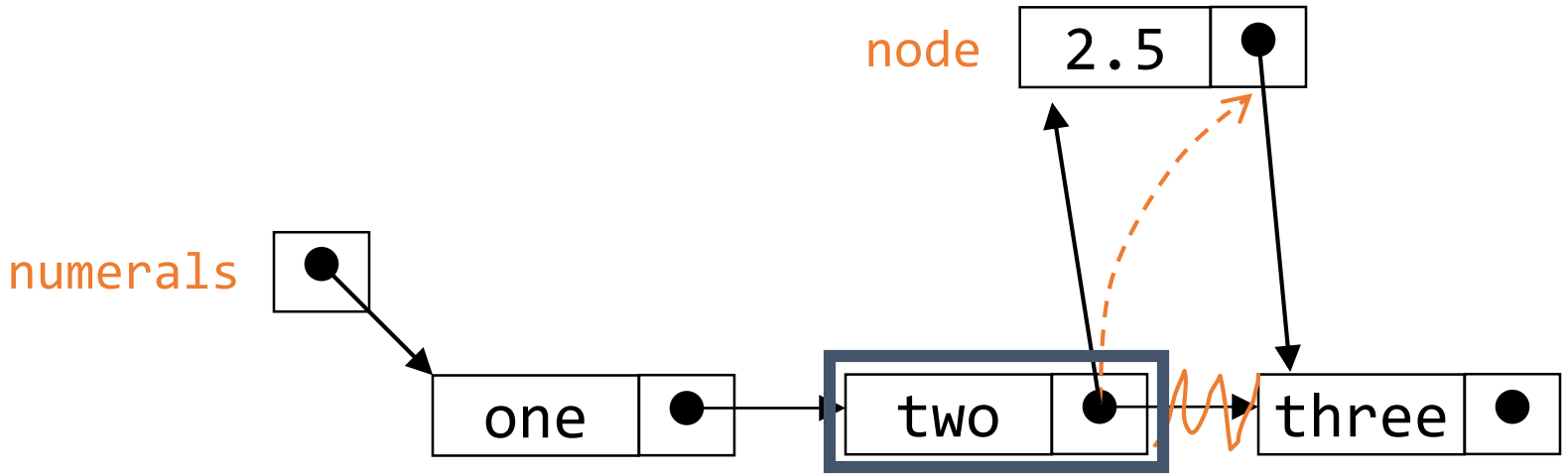
- There are many ways you might want to insert a new node into a list:
  - As the new first element
  - As the new last element
  - Before a given node (specified by a reference)
  - After a given node
  - Before a given value
  - After a given value
- All are possible, but differ in difficulty

Inserting at front (animation)

1. Make a new Node using the data provided
2. Assign head to new node's next.
3. Make new node the head.



Inserting after (animation)



Find the node you want to insert after

First, copy the link from the node that's already in the list

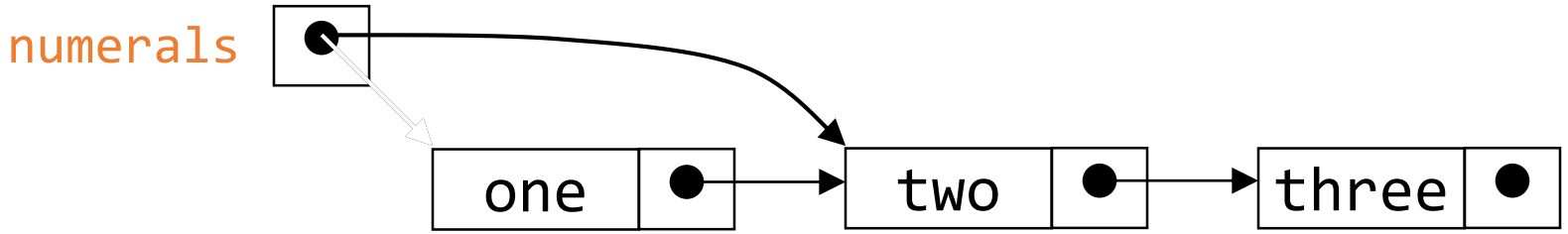
Then, change the link in the node that's already in the list

Deleting a node from a SLL

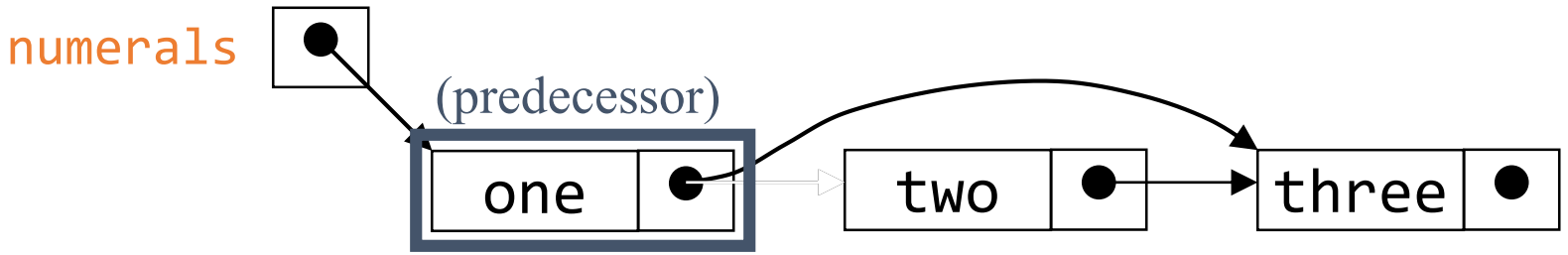
- In order to delete a node from a SLL, you have to change the link in its predecessor
- This is slightly tricky, because you can't follow a pointer backwards
- Deleting the first node in a list is a special case, because the node's predecessor is the list header

Deleting an element from a SLL

- To delete the first element, change the link in the header



- To delete some other element, change the link in its predecessor



Singly Linked List Functions (Implementated)

- Insert at the front
- Insert after a given node
- Insert at the end
- Delete from head
- delete node with a value (first occurrence)
- Delete from tail
- Delete List
- Copy constructor
- Destructor
- Search value in list
- reverse Linked List

Linked List

Arrays are preferred when,

- Need indexed/random access to elements.
- The number of elements in the array is known.
- Need fast iteration through all the elements in sequence.
- Memory is a constraint.

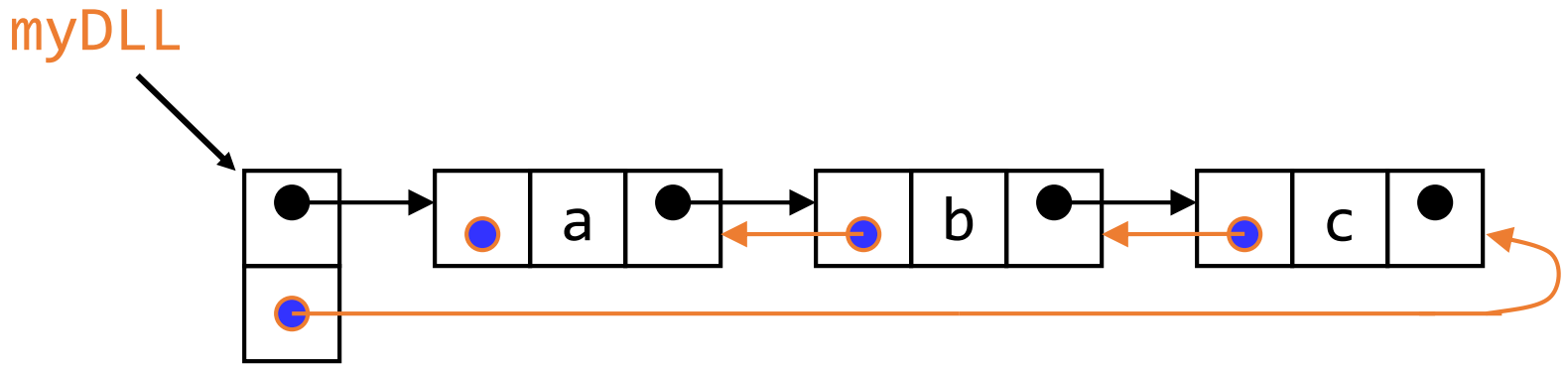
Linked List

Linked lists are preferred when,

- Need constant-time insertions/deletions.
- Number of items are unknown.
- Random access to any elements is not one of the main operations.
- Inserting items in the middle of the list is needed.

Doubly-linked lists

- Doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), and a link to its predecessor (if any)
- The header points to the first node in the list and to the last node in the list (or contains null links if the list is empty)

DLLs compared to SLLs

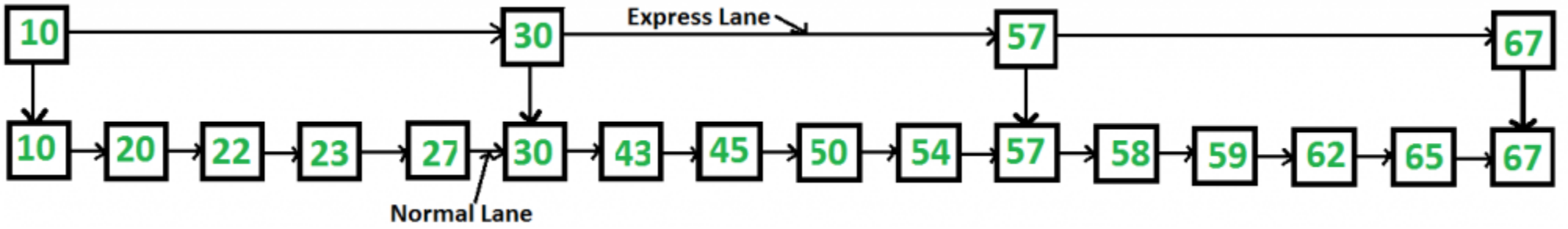
- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Circular Lists

- The list is finite and each node has a successor.
- An example of such a situation is when several processes are using the same resource for the same amount of time, and we have to assure that each process has a fair share of the resource.
- In an implementation of a circular singly linked list, we can use only one permanent reference, tail, to the list.
- Using tail we can add or remove nodes from front and back.
- However, if using only head reference then we need to traverse to the end (in Singly LL) to add or remove from back.

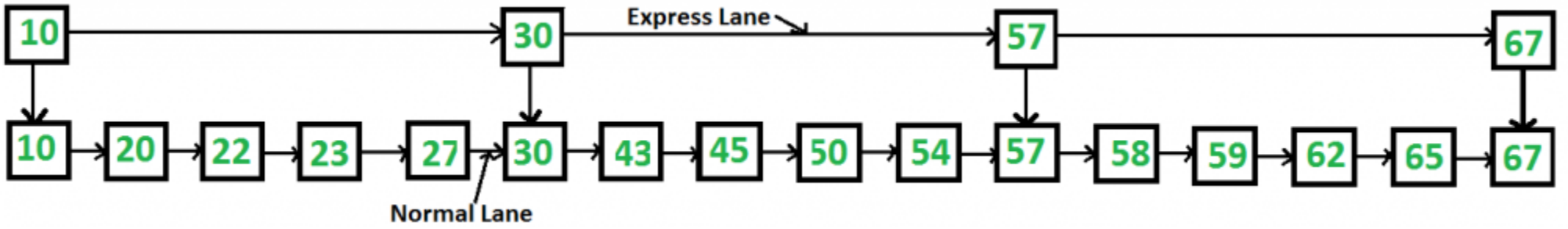
Skip List

- Even if we have sorted LL, we have to traverse the list node by node to search an element.
- Skip list provides a faster search approach for a sorted LL.
- In the Skip list, we operate on multiple layers. Each layer contains different number of total nodes. Therefore layers with fewer nodes allow faster traversal.



Skip List

- To search for 50, we start from first node of “express lane” and keep moving on “express lane” till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on “express lane”, we move to “normal lane” using pointer from this node, and linearly search for 50 on “normal lane”. In following example, we start from 30 on “normal lane” and with linear search, we find 50.



Self Organizing List

- A Self Organizing list reorders its nodes based on searches which are done.
- Idea is to place more frequently accessed items closer to head.
- This is helpful because in a typical database, there are few items which are frequently searched.
- If we have complete search sequence in advance then self organization can be done offline, otherwise we use online self organization of list (moving nodes based on every search operation).

Self Organizing List

1. ***Move-to-Front Method***: Any node searched is moved to the front. This strategy is easy to implement, but it may over-reward infrequently accessed items as it always move the item to front.
2. ***Count Method***: Each node stores count of the number of times it was searched. Nodes are ordered by decreasing count. This strategy requires extra space for storing count.
3. ***Transpose Method***: Any node searched is swapped with the preceding node. Unlike Move-to-front, this method does not adapt quickly to changing access patterns.