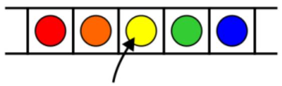
Linked Lists

By Sadaf Iqbal Behlim

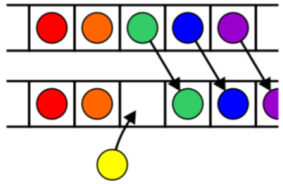
Operations

Operations at the k^{th} entry of the list include:

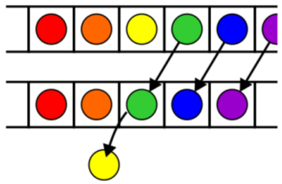
Access to the object



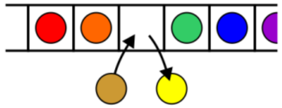
Insertion of a new object



Erasing an object

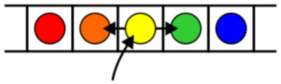


Replacement of the object



Operations

Given access to the k^{th} object, gain access to either the previous or next object



Given two lists, we may want to

– Concatenate the two lists

– Determine if one is a sub-list of the other

Linked Lists

- SLL – Singly Linked List
- DLL – Doubly Linked List
- CLL – Circular Linked List

Singly Linked List (SLList)

SLList = Sequence of nodes

```
class SLList {
    Node head;      // head node of list
    Node tail;      // tail node of list
    int n;          // number of items in list
}
class Node {
    T value;                  // value to store
    Node next;      // pointer to next node
}
```

Stack and Queues

Stack = Last in first out LIFO

Example = pile of plates after washing

Operations

- Push = Insert an element at head of stack
- Pop = Remove an element from head of stack

Queue = First in first out FIFO

Example = student waiting for submitting fees

Operations

- Add/Queue = Insert an element at tail of queue
- Remove = Remove an element from head of queue
- Dequeue = Remove an element from tail of queue

Examples

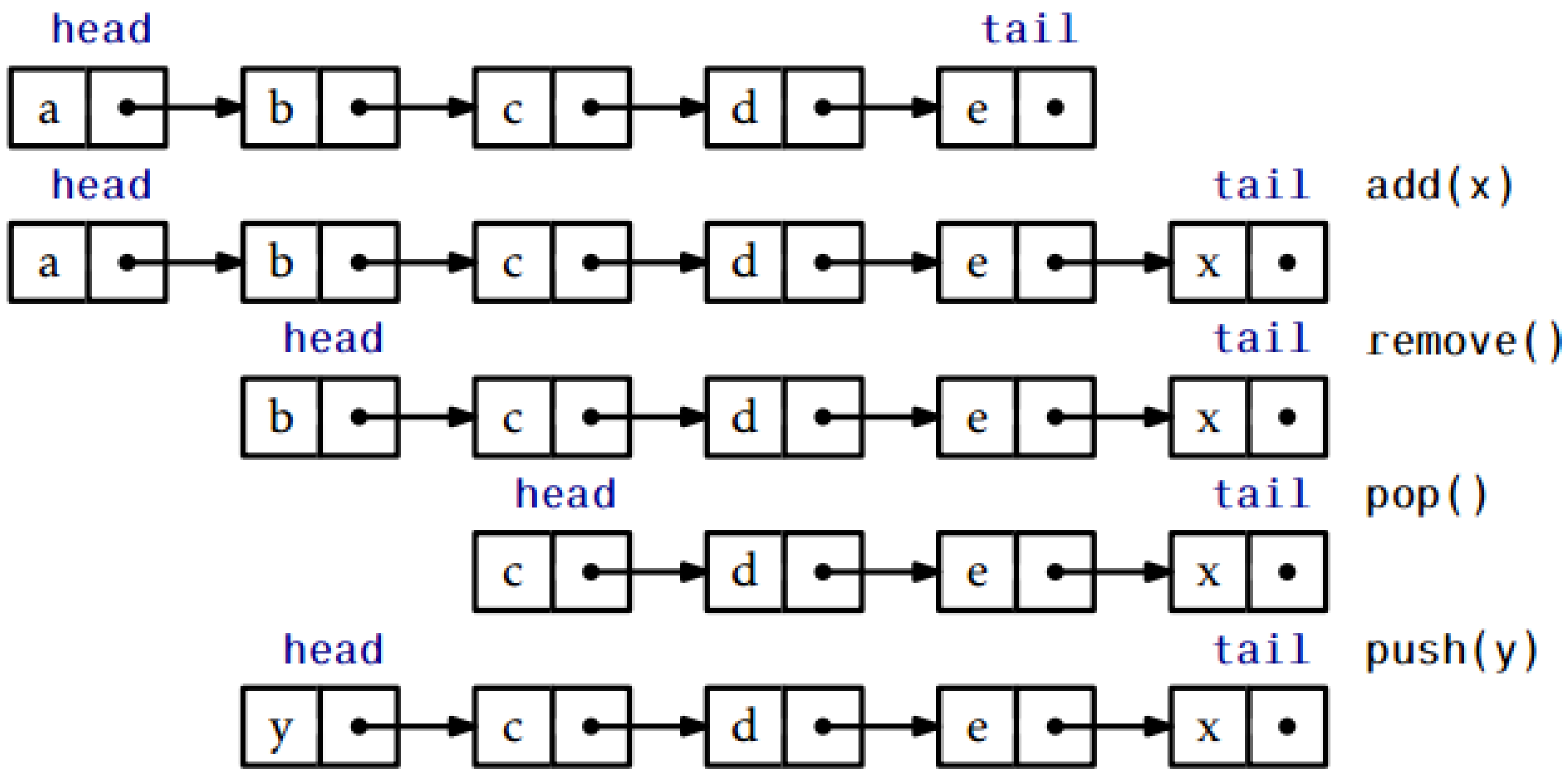


Figure 3.1: A sequence of Queue (add(x) and remove()) and Stack (push(x) and pop()) operations on an SLList.

Cost of Operations using SLList

- Push (add at head) = O(1)
- Pop (remove from head) = O(1)
- Add/Queue at tail= O(1)
- Dequeue/ Remove from head = O(1)
- Dequeue/ Remove from tail = O(n - 2)

Note:

- Cost = number of steps for a particular operation
- Complexity = number of steps for worst case operation

Push

```
T push(T x) {

    Node temp = new Node();
    temp.value = x;
    temp.next = head;
    head = u;
    if (n == 0)                  //if this is first element to insert
        tail = u;                //then tail and head will point to
                                 //same element

    n++;
    return x;
}
```

Pop / Remove

```
T pop() {
   if (n == 0) return null;          //nothing to pop
   T val = head.value;
   head = head.next;
   n= n-1;
   if (n == 0) tail = null;  //if after pop
                             //nothing in list
   return val;
   }
```

Add (Queue)

```
boolean add(T x) {
    Node temp = new Node();
    temp.value = x;
    if (n == 0) {              //if this is first element to add
        head = temp; //then head should be this element
    } else {                   //else
        tail.next = temp;      //add this element at tail
    }
    tail = temp;               //finally newly added element is at tail
    n++;
    return true;
    }
```

Doubly Linked List (SLList)

DLList = Sequence of nodes

```
class Node {
   T value;                  // value to store
   Node prev, next;          // pointer to previous and
                             //next node
}
```

Introducing Dummy Node

- Application:
  - Avoid checks (**marked in blue**) e.g. is list empty etc
- Dummy node contains no data
- Dummy node prev points to tail of list
- Dummy node next points to head of list

Initializing Dummy Node

```
class DLList {
    int n;
    Node dummy;
    DLList() {
        dummy = new Node();
        dummy.next = dummy;
        dummy.prev = dummy;
        n = 0;
    }
}
```

Example

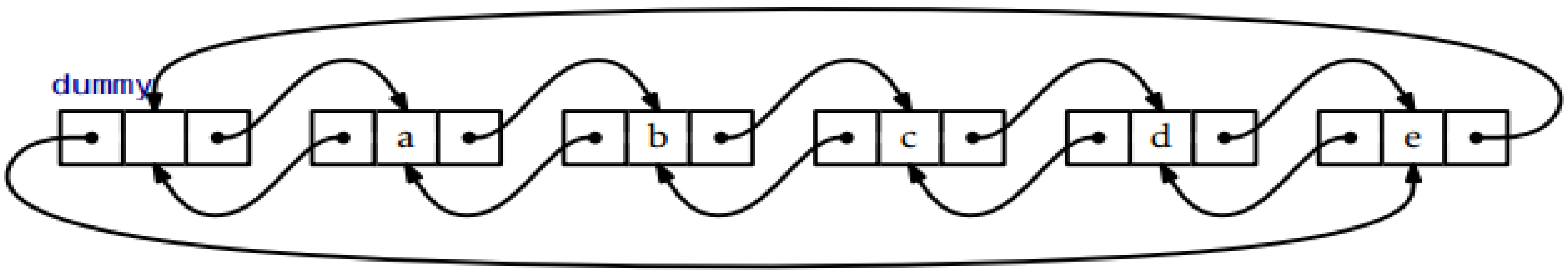
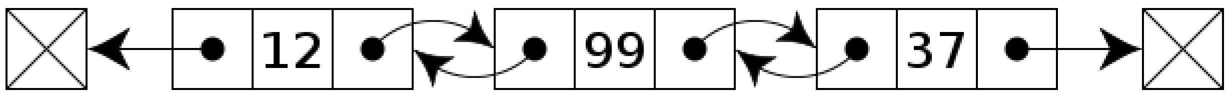


Figure 3.2: A DLList containing a,b,c,d,e.

- In above example by introduction of dummy node doubly linked list becomes circular. You can change this behavior.

Example



- Doubly linked list which is not circular

Cost of Operations using DLList

- get(i) i.e. get ith node = $O(1 + \min\{i, n - i\})$
- set(i,x) i.e. set x at ith node = $O(1 + \min\{i, n - i\})$
- add(i,x) i.e. set x at ith node = $O(1 + \min\{i, n - i\})$
- remove(i,x) i.e. set x at ith node = $O(1 + \min\{i, n - i\})$

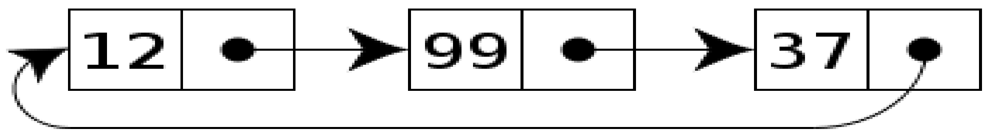
Implementation

View implementation of

- Node getNode(int i)
- T get(int i)
- T set(int i, T x)
- Node addBefore(Node w, T x)
- void add(int i, T x)
- void remove(Node w)
- T remove(int i)

Also notice that now there is no need of **n==0 type checks (list is empty)** due to dummy node

Circular Linked List



A circular linked list

Above is example of singly linked list which is circular.

Introduction of dummy node in previous example automatically made the list circular. You can modify implementation to make it non circular doubly linked list.

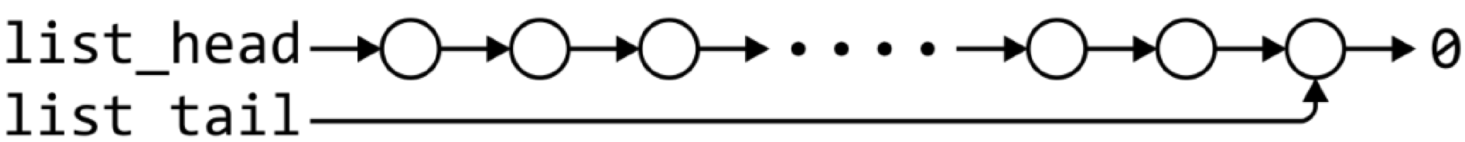
Practice Problems

1. Try list reversal Program 3.9, list insertion sort Program 3.10, Josephus problem Program 3.12 of Robert Sedgewick
2. Try End of chapter Exercise 3.35 to 3.47 of Robert Sedgewick
3. Try End of chapter Exercise 3.1-3.19 of open data structures in java

Singly linked list

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

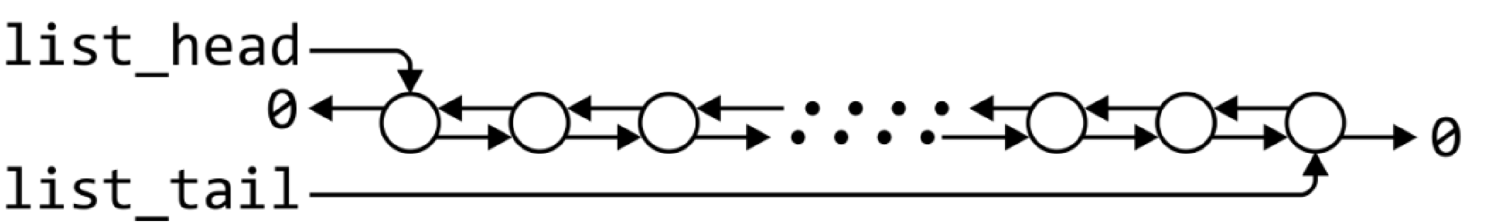
By replacing the value in the node in question, we can speed things up
– useful for interviews



Doubly linked lists

	Front/1st node	k^{th} node	Back/n^{th} node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

* These assume we have already accessed the k^{th} entry—an $O(n)$ operation



Doubly linked lists

Accessing the k^{th} entry is $\mathrm{O}(n)$

	k^{th} node
Insert Before	$\Theta(1)$
Insert After	$\Theta(1)$
Replace	$\Theta(1)$
Erase	$\Theta(1)$
Next	$\Theta(1)$
Previous	$\Theta(1)$