# Linked List

## 1. Linked List Intro

Linked list is a linear data structure that can grow or shorten based on the amount of data it holds. It is often compared with arrays. If the number of data items is fixed, arrays are better to store data than linked list. However, if the size of data changes quite often, arrays can be inflexible. Insertion at a specific index in array requires all items after i be right shifted. Deletion of items from array also needs to be handled in a special way (e.g. put a marker at deletion index, or left shift the items to remove the hole created by deletion). Arrays are often contiguous in memory. Linked list require extra memory to keep reference of another node. Nodes can be sparsely placed in memory.

class Node

{

int data;

Node next;

Node(int d) { data = d; }

}

## 2. Linked List Insertion

Linked lists can be inserted at the beginning (head), at a particular index or at the end of an existing linked list. It can also be inserted at a null node (so that it being the only node).

a. Insertion at end (could be the only element in the node)

Node insert(Node head, int d)

{

Node node = new Node(d);

Node cur = head;

if (head == null) {

head = node;

} else {

while (cur.next != null) {

cur = cur.next;

}

cur.next = node;

}

return head;

}

b. Insertion at a particular index

Traverse to the particular index or to the end of the node.

Assume: Insert at end, if end came before index reached k. Can be inserted both iterative or recursive way.

Node insert(Node head, int element, int k)

{

Node node = new Node(element);

Node cur = head;

if (head == null) {

head = node;

} else {

int count = 0;

Node prev = null;

while (cur != null && count < k) {

prev = cur;

cur = cur.next;

count++;

}

if (prev != null) {

prev.next = node;

node.next = cur;

} else {

node.next = head;

head = node;

}

}

return head;

}

## 3. Linked List Deletion

Deleting a given key.

4→5→7→9→ 8

a. Deleting 4 (head)

b. Deleting 8 (tail)

c. Deleting other nodes

1. Find the previous node, if it is null, we are still at head, return next node as head

2. Else prev.next = cur.next

Node delete(Node head, int key)

{

if (head == null)

return null;

if (head.data == key)

return head.next;

Node prev = null;

Node cur = head;

while (cur != null && cur.data != key) {

prev = cur;

cur = cur.next;

}

if (cur == null) {

return null;

} else {

prev.next = cur.next;

}

}

## 4. Linked List Deletion at specific index

1. Find the index, using previous and current node, delete the index.

2. If the index is 0, return next node of head.

3. If index is out of bounds, return node without deletion

Node delete(Node head, int k)

{

if (head == null)

return null;

Node cur = head;

Node prev = null;

int count = 0;

while (cur != null && count < k) {

prev = cur;

cur = cur.next;

count++;

}

if (count < k) {

return head;

} else {

if (prev == null) {

head.next;

} else {

prev.next = cur.next;

}

}

return head;

}

## 5. Find length of a linked list

Iterative

======

int count = 0;

while (cur != null) {

count++;

cur = cur.next;

}

return count;

// Recursive

if (head == null) {

return 0;

} else {

return 1 + count (head.next);

}

## 6. Swap nodes in linked list

1->2->3->4->5->6->null

1. One of the nodes could be head

2. Nodes may not be found

1→2→3→4→5→6→null 3,5

1→2→5→4→3→6->null

prevx, x, prevy, y

if (x == null || y == null)

return head;

if (prevx == null) {

head = y;

} else {

prevx.next = y;

}

if (prevy = null) {

head = x;

} else {

prevy.next = x;

}

// Swap pointers

Node next = x.next;

Node x.next = y.next;

y.next = next;

## 7. Reverse a linked list

1→2→3->4->5->null

5→4→3→2->1->null

1. If node == null, return null

2. Adjust current node’s next pointer to point to previous node

3. Move both previous and current forward

Node reverse(Node head)

{

Node prev = null;

Node cur = head;

while (cur != null }

Node next = cur.next;

cur.next = prev;

prev = cur;

cur = next;

}

return prev;

}

## 8. Merge Sort for Linked List