# 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

A: 1 → 5 → 7

B: 3 → 4 → 12 → 15

1. If A is null, return B else return A. It will take care of state of both being null

2. Divide the list into two halves

3. Call MergeSort on both halves

4. Merge both sorted halves.

Q: What it means for an entity to be sorted.

→ If there is only one element return that element

public class MergeSortExample

{

static Node

{

int data;

Node next;

public Node(int d) { data = d; }

}

Node MergeSort(Node head)

{

if (head == null || head.next == null)

return head;

Node mid = getHalf(head);

Node A = head;

Node B = mid.next;

mid.next = null; // break the chain

return Merge(MergeSort(head), MergeSort(B));

}

// Uses two runner technique

Node getHalf(Node head)

{

if (head == null)

return;

Node slow = head;

Node fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

Node Merge(Node A, Node B)

{

Node result = null;

if (A == null)

return B;

else if (B == null)

return A;

if (A.data < B.data) {

result = A;

result.next = Merge(A.next, B);

} else {

result = B;

result.next = Merge(A, B.next);

}

return result;

}

}

## 9. Reverse a linked list in given size

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

k = 3

3→2→1→6→5→4→8→7

reverse(node, k)

head.next = reverse(cur.next)

return previous

Node reverse(Node node, int k)

{

if (node == null || node.next == null)

return node;

Node cur = node;

Node prev = null;

int count = 0;

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

Node next = cur.next;

cur.next = prev;

prev = cur;

cur = next;

count++;

}

if (cur != null)

node.next = reverse(cur.next, k);

return prev;

}

## 9. Loop detect and remove

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8→4→5→6→7→8→4 …

a. If fast reaches null → good no cycle

b. If fast meets slow, there is a circle

c. When fast and slow meet, loop simultaneously one step from beginning and from the meet point. Where they meet is the loop point, remove it by setting the next before loop to null

void removeLoop(Node head)

{

if (head == null || head.next == null)

return;

Node slow = head;

Node fast = head;

while (fast != null && fast.next != null && fast != slow) {

fast = fast.next.next;

slow = slow.next;

}

if (fast == null || fast.next == null)

return;

Node start = head;

if (slow == fast) {

while (fast != slow) {

prev = fast;

fast = fast.next;

slow = slow.next;

}

prev.next = null; // remove the loop

}

}

## 10. Add two numbers

Reverse two nodes, until both nodes are null keep on adding values with carry into consideration. When one of the nodes gets null, add 0 value until both get to null. For each sum create new node, and put result % 10 as value and result/10 as carry. Track the head of the result node and finally, reverse the head and return the node.

Node add(Node A, Node B)

{

Node rA = reverse(A);

Node rB = reverse(B);

Node resultNode=null;

int carry = 0;

Node resultHead=null;

while (rA != null || rB != null) {

int result = (rA != null ? rA.data : 0) + (rB != null ? rB.data: 0) + carry;

if (resultNode == null) {

resultNode = new Node(result%10);

resultHead = resultNode;

} else {

resultNode.next = new Node(result%10);

resultNode = resultNode.next;

}

carry = result/10;

if (rA != null)

rA = rA.next;

if (rB != null)

rB = rB.next;

}

if (carry > 0) {

resultNode.next = new Node(carry);

}

if (resultHead != null)

return reverse(resultHead);

else

return null;

}

## 11. Rotate linked list

1→2→3→5→7

3

5→7→1→2->3

Get node at k (cur)

move forward but save the node at k (new head). Set next node of k as null (as last node)

at the end set next of (prev of) end as previous head node.

Node rotate(Node node, int k)

{

if (node == null || node.next == null)

return node;

if (k == 0)

return node;

Node headRef = node;

int count = 0;

while (node != null || count < k) {

count++;

node.next;

}

Node newhead = null;

if (node != null)

newhead = node.next;

Node cur = newhead;

while (cur.next != null) {

cur = cur.next;

}

cur.next = headRef;

return newhead;

}