public Node reverseInPair(){

Node previous =null, current, nextNode, temp;

if(head == null ) return null;

if(head.next == null ) return head;

previous = head;

current = head.next;

head = current;

while (true){

nextNode = current.next;

current.next = previous;

if(nextNode == null || nextNode.next == null){

previous.next = nextNode;

break;

}

previous.next = nextNode.next;

current = nextNode.next;

previous = nextNode;

}

return head;

}

==============================================================================

Self Organizing List | Set 1 (Introduction)

The worst case search time for a sorted linked list is O(n). With a Balanced Binary Search Tree, we can skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays.

One idea to make search faster for Linked Lists is [Skip List](http://www.geeksforgeeks.org/skip-list/). Another idea (which is discussed in this post) is to *place more frequently accessed items closer to head.*. There can be two possibilities. offline (we know the complete search sequence in advance) and online (we don’t know the search sequence).  
In case of offline, we can put the nodes according to decreasing frequencies of search (The element having maximum search count is put first). For many practical applications, it may be difficult to obtain search sequence in advance. A [Self Organizing list](http://en.wikipedia.org/wiki/Self-organizing_list) reorders its nodes based on searches which are done. The idea is to use locality of reference (In a typical database, 80% of the access are to 20% of the items). Following are different strategies used by Self Organizing Lists.

**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.

[**Competitive Analysis:**](http://en.wikipedia.org/wiki/Competitive_analysis_(online_algorithm))  
The worst case time complexity of all methods is O(n). In worst case, the searched element is always the last element in list. For [average case analysis](http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), we need probability distribution of search sequences which is not available many times.  
For online strategies and algorithms like above, we have a totally different way of analyzing them called*competitive analysis* where performance of an online algorithm is compared to the performance of an optimal offline algorithm (that can view the sequence of requests in advance). Competitive analysis is used in many practical algorithms like caching, disk paging, high performance computers. The best thing about competitive analysis is, we don’t need to assume anything about probability distribution of input. The Move-to-front method is 4-competitive, means it never does more than a factor of 4 operations than offline algorithm (See [the MIT video lecture](http://www.youtube.com/watch?v=2RxCCEHlEys) for proof).

## Merge a linked list into another linked list at alternate positions

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.  
For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers.

# Sorting and Searching Linked Lists in Java

|  |
| --- |
|  |

Dr. Dobbs Journal, May 1998, Algorthm Alley, by John Boyer.

// Merge subroutine used by the following

private void merge(Node before, Node F1, int N1,

Node F2, int N2, NodePair NP) {

Node first = null, last = null, temp = null;

int I,J;

first = last = F1.compareTo(F2) <= 0 ? F1 : F2;

for (I = J = 0; I < N1 || J < N2; ) {

if (I < N1 && (J >= N2 || F1.compareTo(F2) <= 0)) {

temp = F1; F1 = F1.next; I++; }

else {

temp = F2; F2 = F2.next; J++; }

last.next = temp;

last = temp;

}

if (before = null)

First = first

else

before.next = first;

last.next = F2;

NP.first = first;

NP.last = last;

}

// Simple non-recursive Merge Sort

private void mergesort() {

int i, j, k, N1, N2;

Node F1, F2, before;

NodePair NP = new NodePair();

for (i = 1; i < NumNodes; i <<= 1) { // the { is not required here

for (before = null, N1 = N2 = i, j = 0; j+n1<NumNode; j += i << 1) {

F1 = F2 = before == null ? First : before.next;

for (k = 0; k < N1; k++) F2 = F2.next; // move F2 to [N1]

if (N2 > NumNodes - j - N1) N2 = NumNodes - j - N1; // limit N2

merge(before, F1, N1, F2, N2, NP);

before = NP.last;

}

}

}

// Singly Linked List Recursive Merge Sort

private void mergesort(node before, Node F1, int N1, NodePair NP) {

if (N1 <= 1)

NP.first = NP.last = F1;

else {

Node F2;

int N2;

N2 = N1; N1 >>= 1; N2 -= N1;

mergesort(before, F1, N1, NP);

F1 = NP.first;

F2 = NP.last.next;

mergesort(NP.last, F2, N2, NP);

F2 = NP.first;

merge(before, F1, N1, F2, N2, NP);

}

}

// Singly Linked List Binary Search

public Node binarySearch(Object SearchKey) {

Node PartitionFirst = First, MidPtr = nul;

int Partition Size = NumNodes, Mid, I, Result;

while (PartitionSize > 0) {

Mid = PartitionSize / 2;

MidPtr = PartitionFirst;

for(I = 0; I < Mid; I++)

MidPtr = MidPtr.next;

Result = MidPtr.compareTo(SearchKey);

if (Result > 0)

PartitionSize = Mid;

else if (Result < 0) {

PartitionSize -= Mid;

PartitionFirst = MidPtr;

}

else return MidPtr;

}

return null;

}

## Linked list Quicksort

Uses first node's key as the pivot. Traverses forward through the list using two references called aNode and aNodePrev. Nodes with a key value greater than or equal to the pivot are ignored. When a node containing a lesser key is encountered, the code deletes aNode by connecting aNodePrev.next to aNode.next. Then aNode is pushed onto the front of the list, becoming the new first node and the remaining nodes of the list are traversed or transferred to the front in the same manner. At the end of this process,  the list is divided into two sublists, which can be recursivly sorted by applying this partitioning procedure to each sublist.

//Singly Linked List Quicksort

private Node QuickSort(Node before, Node first, int n) {

int Num1=0, Num2=n, i=1;

Node Pivot=first, aNode=first, aNodePrev=first;

// Pivot Advancement

for (i=1; i<n; i++, aNode=aNode.next) {

if (aNode.compareTo(aNode.next) > 0)

break;

if ((i&1)==0) { //every other time through the loop

Pivot=Pivot.next;

Num1++;

}

}

//Recognize sortedness in linear time

if (i == n) return first; //Pivot advanced through entire list and found it to be aleady sorted

// Partition list by unlinking nodes with values less

// than the pivot and pushing them onto front of list

for (aNodePrev = aNode; i < n; i++) {

aNodePrev.next = aNode.next;

if (Pivot.compareTo(aNode) > 0) {

aNode.next = first;

first = aNode;

Num1++;

}

else aNodePrev = aNode;

}

if (before!=null) before.next = first;

Num2 = n - Num1 - 1;

// Recurse to sort sublists

if (Num1 > 1) first = QuickSort(before, first, Num1);

if (Num2 > 1) QuickSort(Pivot, Pivot.next, Num2);

return first;

)

======================================================================== public static Node partition\_list(MyLinkedList list) {

Node node = list.head;

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

return node;

}

Node mark = node, iter = node, pivot = node;

Node temp, temp1;

while (iter.next != null) {

if (iter.next.data <= pivot.data) {

int lt = pivot.data - iter.next.data;

// System.out.println("mark " + mark.data);

if (iter != mark) {

temp1 = iter.next.next;

temp = mark.next;

mark.next = iter.next;

mark.next.next = temp;

iter.next = temp1;

} else {

iter = iter.next;

}

if(lt > 0)

mark = mark.next;

} else {

iter = iter.next;

}

}

if(mark != pivot){

temp = mark.next;

list.head = pivot.next;

mark.next = pivot;

pivot.next = temp;

}

return mark;

}

=======================================================================

public static Node reverseList\_recursive(Node node){

Node head =null;

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

Node previous=node, current = node.next;

head = recurse2(node.next);

current.next = previous;

previous.next = null;

return head; }