5-linked-lists

Contents

- Intorduction
- Creating
- Deleting
- Runner approach
- Previous & current approach
- Recursive approach
- List interweaving

Intorduction

Represents a sequence of nodes, where each node is linked to the next. This is called a singly linked list, while variations exist where each node might have a link to the node before it, making it a doubly linked list.

```
class Node {
    int value;
    Node next;
}
```

Creating

```
Node create(List<Integer> elements) {
    Node tail = null;
    Node head = null;
    for (Integer entry : elements) {
        if (head == null) {
```

```
            head = new Node();
            head.value = entry;
            tail = head;
        } else {
            Node next = new Node();
            next.value = entry;
            tail.next = next;
            tail = next;
        }
    }
    return head;
}
```

Deleting

Deleting from a list usually implies removing of a node link within the list, this is done by linking the parent of the node to be removed with the link to the next node after the one being deleted `prev.next = curr.next`. In the edge case where the `value` is contained in the node, we simply return the link to `node.next`.

```
Node delete(Node node, int value) {
    // in the case where the target node's value is the value to remove,
       simply
    // return the link to the next node, and bail
    if (node.value == value) {
        return node.next;
    }

    // start tracking the current node and the parent node, starting from the
    // next node, since we have already checked the input root node anyways
    Node curr = node.next;
    Node prev = node;

    // loop over until the end of the list is reached, curr is moving forward
    // to the tail of the list
    while (curr != null) {
        // check if the current node's value matches the value being looked
           for
        if (curr.value == value) {
            // since we start from node.next as curr we are guaranteed to
               have
            // prev / parent node always set
            prev.next = curr.next;
            break;
        }

        // move the two pointers forward, keeping track of the parent and the
        // current node, the loop will exit once curr becomes nil
        prev = curr;
        curr = curr.next;
    }
```

```
    // return the 'new' head of the list, caller should take that into
       account
    return node;
}
```

However some tasks with lists can also include normal shift base removal, where you do not remove the node, but you just remove the value by shifting from the left, by overriding the `curr.value` with `curr.next.value`, then deleting the tail node, as it would contain the last element two times after the process

That is a gotcha question sometimes. (e.g they want you to remove a `node` from the middle of the list, but you have only access to that node, well they likely mean that you have to just copy elements left to right, to "remove" the value not the node itself.

```
void delete(Node node) {
    // we are only given a starting node, which we want to remove, no access
    // to this node's parent or previous node
    Node curr = node;

    // move, shift values to the left, we are not really deleting nodes,
       simply
    // copying the values of the next node into the current one until the
    // current node has a next one. The last node that has a next is the one
    // right before the tail
    while (curr != null && curr.next != null) {
        curr.value = curr.next.value;

        // move to the next node, we know it exists due to the while loop
           check,
        // the last assignment here will happen right before the tail and the
            loop
        // will exit, therefore curr will point to the tail of the list
           outside of
        // loop, when the loop exits
        curr = curr.next;
    }

    // current will always point to the 'tail' of the list, the very last
       node,
    // being pedantic, simply set the value of the tail to some default value
    // as an example, we assume that to be 0, but it could be any 'value'
    curr.value = 0
}
```

The approach above does not really differ from a regular static array shift based removal, besides the fact that instead of indices we are using node links instead. But the approach is completely identical.

Runner approach

This is very important topic when talking about linked lists, there is a common technique called the runner. Where we loop through, or traverse a linked list forward or backwards, not with one, but two pointers.

This is very often expected approach to use when trying to detect loops in the list, where a list might have the same node (reference) more than once, forming a loop. Using the runner approach the fast pointer will move forward two elements ahead of the slow pointer, that way we can detect the loop.

Very CRUCIAL to know, that in a list with a loop, when the slow and fast nodes intersect, in other words when we detect the loop, fast and slow will point at a node within that loop, if we reset fast or slow to point to the original head, and we move both pointers together by one step, they will meet exactly at the start of the loop, the first node that forms the loop, that is GUARANTEED !

The general pseudo code for this approach looks like this

```
void runner(Node node) {
    // start both fast and slow at the same head of the list
    Node slow = node;
    Node fast = node;

    // make sure there is any nodes left
    while(thereAreMoreNodesToMoveTo) {
        // get the immediate next node, for the slow node
        slow = getNextNode(slow);

        // get the n-th node, from the last fast node
        fast = getNthNode(fast);

        // do something with fast and slow and repeat
    }
}
```

To detect if there is a loop within a linked list, we employ the runner approach, where we move into the list with two pointers, slow by one element, fast by 2 elements forward, if they ever intersect the list has a loop, a loop means that the same `node` by reference (not value) is present more than once in the linked list

```
void loop(Node node) {
    // start from the SAME initial starting point, this is VERY important, we
    // MUST start at the same starting positions, otherwise the loop
       detection
    // would not work
    Node slow = node;
    Node fast = node;

    // move both at different paces, if there is a loop in the list, they
       will
    // always meet at some point within this loop
    while (fast != null and fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;

        // they met, there is a loop in the list, the same exact node, by
        // reference is contained in the list twice
        if (slow == fast) {
            break;
        }
    }
```

```
}
```

Another example is interspersing items in a list, let us imagine that we have a linked list with the following structure where assume we have an even number elements such as the ones below

- `a1 - a2 - a3 - a4 - b1 - b2 - b3 - b4` - input
- `a1 - b1 - a2 - b2 - a3 - b3 - a4 - b4` - result

The runner approach would allow us to have two pointers one starting from the head of the list, the other starting from (size / 2) + 1. In that case the example has 8 elements, therefore the second pointer can start from the (head + 4 + 1) elements (or the fifth element in the list being `b1`), then move both forward with one element at a time.

```
public Node intersperse(Node start, Node midpoint) {
    // start the two pointers from the two positions, in the example above
    // 'start' would point at 'a1' and 'midpoint' would point at 'b1'
    Node ahead = start;
    Node bhead = midpoint;

    // loop from both heads, one starting from the actual list head, the
       other from the midpoint of interspersion,
    // attach the current midpoint node, to the current head, and move them
       along until the current midpoint's next node points to
    // nil, which means the actual tail of the list was reached
    while (ahead != null && bhead != null) {
        // remember where each of the two heads pointer's next nodes point to
        Node anext = ahead.next;
        Node bnext = bhead.next;

        // we at the list's tail ?
        if (bhead.next != null) {
            // when the bhead next is not nil, we have not reached the actual
                tail of the list, thus, attach the current bhead to
            // the ahead, and to bhead.next the anext
            ahead.next = bhead;
            bhead.next = anext;

            // move the two heads along forward
            ahead = anext;
            bhead = bnext;
        } else {
            // this means we have reached the actual end of the array, when
               bhead.next is nil, therefore only attach bhead to
            // ahead.next, the bhead.next here is set to nil for good measure
               , but it should be already nil
            ahead.next = bhead;
            bhead.next = null;

            // there is nothing more to move to
            ahead = null;
            bhead = null;
        }
    }
```

Previous & current approach

This is a variation of the runner approach where we keep the reference to the previous element in the list, the previous node, and the current one, usually to make it easier when we want to delete elements. This comes up a lot, both pointers are one off each other. The previous node is not always moving one step behind the current node, it might depend on the algorithm

Recursive approach

Another technique, which is very spread in linked list problems, if you are having trouble solving a linked list problem you should explore if a Recursive approach will work.

How can it help, in a linked list problem, having recursive calls traverse the list from head to tail, give us the inverse traversal tail to head, when the recursive call returns. This is not for free since the space complexity O(n) which is due to the call stack.

Example of this is to find the k-th to last element if a singly linked list. What can we do here ? Using recursion we can drill down to the tail, keeping the current count of elements in the list, at the bottom of the recursion we would have gone through the entire list, having counted all elements, and in the post recursive calls we can subtract from the total elements count in the list k, and compare to the current element's count, that way we can find the node exact node that is k elements behind the tail (or the last element)

```
// class or local / global variable, used for simplicity, and ease of
// understanding, stores the total number of elements in the list
int total = 0;

// example value, we are looking for the 3rd element from the end of the list
   ,
// meaning that if the list has 10 elements the one we are looking for is the
    7th
// element
int kth = 3;

void recursive(Node node, int accum) {
    // if that is the case we have reached the end of the list, meaning we
       are
    // at the element after the tail, in other words the function was called
       with
    // f(node.next), where node was the tail, and there are no more elements
       in
    // the list, therefore store whatever the value of 'accum' is as total,
       exit
    if(node == null) {
        // at the node 'after' the tail, we want to reach this point and not
        // early exit on the tail node, since the tail node also has to be
        // accounted for - accum + 1, now remember the total count & exit
        total = accum;
        return;
    }

    // pre-recursive call, local variable for tracking the current element
    // count, note that assign it to accum + 1, two reasons. First the curr
```

```
    // index/position is indeed current accum + 1, since at this point accum
    // stores the number of elements in the list 'so far', but the current
    // element is not accounted for yet. Second, this local variable is used
    // below to check curr element is k elements away from the total count
    int curr = accum + 1

    // this will drill down, till the tail, then start post recursive
    // unwinding of the function call stack, starting from the tail,
    // up to the head of the list, can be used to do operations in reverse on
    // the list, like print the n-th element before the tail for example
    recursive(node.next, curr);

    // post-recursive call, will print the k-th to last element in list
    if(total - kth  == curr) {
        println(node.value)
    }
}
```

List interweaving

List interweaving, a very important to understand algorithm, which weaves two lists into each other, effectively creating all possible permutations between the elements of two lists. However the relevant positions or order of elements in each list is retained the same, i.e. if one of the list is {1,2}, the order of elements after the weaving, will be such that 1 will always come before 2, no matter where the 1 and 2 are in the final result / weaved list

```
    void weave(LinkedList<Integer> first, LinkedList<Integer> second,
       LinkedList<Integer> prefix, List<LinkedList<Integer>> result) {
        if (first.isEmpty() || second.isEmpty()) {
            // in the case where one of the source lists are emtpy, meaning
               elements were moved to the prefix list, clone the prefix and
            // add to that the first and second lists, the prefix at any
               given time would contain elements from one, or both arrays, in
            // different order, i.e first elements from first array, then
               second, or vice-versa, but not mess the order of the elements
            // that come from the same array.
            LinkedList<Integer> cloned = (LinkedList<Integer>) prefix.clone()
               ;
            cloned.addAll(first);
            cloned.addAll(second);
            result.add(cloned);
            return;
        }

        // remove the front element from the first list
        Integer element = first.removeFirst();
        // add that element to the tail of the prefix list
        prefix.addLast(element);
        // drill down, depth first, into the first list first
        weave(first, second, prefix, result);
```

```
        // remove the last inserted element to the prefix
        prefix.removeLast();
        // restore the element back into the source list
        first.addFirst(element);

        // remove the front element from the second list
        element = second.removeFirst();
        // add that element to the tail of the prefix list
        prefix.addLast(element);
        // drill down, depth first, into the second list second
        weave(first, second, prefix, result);
        // remove the last inserted element to the prefix
        prefix.removeLast();
        // restore the element back into the source list
        second.addFirst(element);
    }
```

How does it work, we have to take a look at an example, lets take the following two lists {1,2} and {3,4}, and represent the recursive function calls in a stack order

The way this flow works, is by first exhausting the first list, down to the very last element, put everything into the prefix list, and append in the base case, now the recursive stack starts to unwind, elements are returned to the first list, one by one, and for each, we go depth first into the recurisive calls for the second list. Let us take the example {1,2} & {3,4}

```
    weave({1,2}, {3,4}, {}, {})
    weave({2}, {3,4}, {1}, {})
    weave({}, {3,4}, {1,2}, {})
            1,2,3,4
        weave({2}, {4}, {1,3}, {})
        weave({2}, {}, {1,3,4}, {})
            1,3,4,2
```