Objective:

- To make a strong grip on link based structure.

Note: In almost all the problems the link list is assumed to be linear single link list unless specified explicitly and the time bound for most of the functions is O(N) and you are not allowed to change the node links or nodes data unless specified.

Problem-1: *Union (Math: Set Operation) of two Link Lists.*

If list1 contain the elements 2, 3, 4, 5, 7, 9, 11 and list2 contain elements 3, 4, 10, 1, 5 then after calling this function the resultant list should contain elements 2, 3, 4, 5, 7, 10, 1, 9, 11

Not: if you are implementing it as a member function of class 'LSLL' then function prototype will be
Prototype:
LSLL doUnion(LSLL& list2);
List1 is represented by *this and list2 will be received in function.
Same pattern can be followed for rest of the problems

Problem-2: *Equality of Lists.*

This function determines that two given lists are equal or not (Math Set Operation: Equality).
Prototype:
int LSLL::isEqual(LSLL &list2); //list1 is *this and list2 is being received.

Problem-3: *Create List Clone.*

This function returns the deep copy (clone) of the list (*this).
Prototype:
LSLL LSLL::createClone();

Problem-4: *Delete Alternate nodes.*

This function will delete every second element from the start of the list e.g. if a list contains the elements 2, 3, 4, 5, 7, 9, 11 then after calling this function the list should contain elements 2, 4, 7, 11.
Prototype:
Void LSLL::delAlternate(); //delete alternate nodes of the calling object.

Problem-5: *Split List.*

This function will count the number of nodes in a linked list, and split the linked list form the middle on the basis of number of nodes. list1 and list2 be initially empty. If the original linked list contains 8 elements, then after the function call list1 will contain the first four elements and list2 will contain the last four elements. If the original list contains 7 elements, then list1 will contain 4 elements and list2 will contain 3 elements after the function call. Also, after the function has been called the original list will become empty. You are required to print all three lists after the function has been called.
Prototype:
void LSLL::splitList (LSLL & list1, LSLL & list2); //the original list means the calling object

Problem-6: *Remove all the duplicate nodes in a link list*

When this function will be called on some link list then after the execution of this function the list will not contain any node with its info repeating more then once. E.g. if list contains 23, 5, 4, 23, 6, 78, 4, 5 then after the list will contain only 23, 5, 4, 6, 78

Prototype:

void LSLL::removeDuplicates();

Problem-7: *Merge two sorted link list into a 3rd link list so that the resultant must also be sorted*

E.g. is list1 contains 1, 2, 3, 40, 50 and list2 contains 10, 40 60 the result list will contain 1, 2, 3, 40, 50, 60.

Prototype:

LSLL LSLL::mergeSortedLists(LSLL & list2); // list1 means *this

Problem-8: *Reverse print the link list iteratively and recursively.*

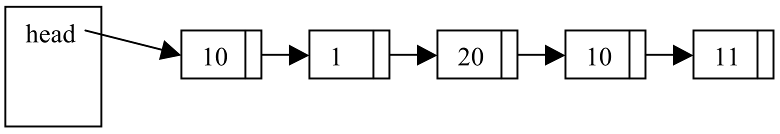
Prototype:

void LSLL::reversePrint(); // list means *this

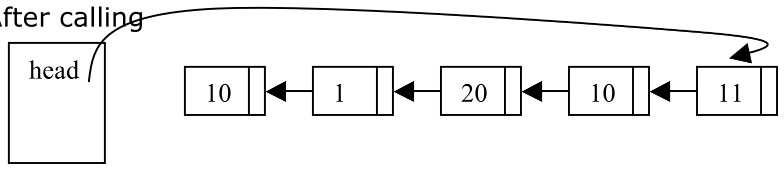
Problem-9: *Reverse the link list structure iteratively and recursively*

Note: No data swapping is being done only address swapping is being involved

List Before calling



List After calling



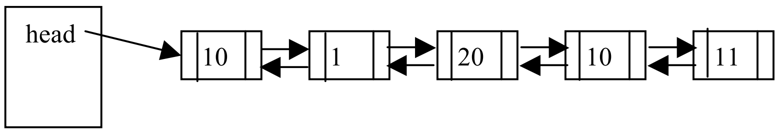
Prototype:

void LSLL::reverseList(); // list means *this

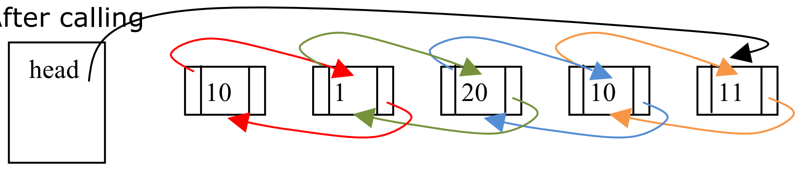
Problem-10: *Reverse the link list structure iteratively and recursively*

Note: No data swapping is being done only address swapping is being involved

List Before calling



List After calling



Prototype:

void LSLL::reverseList(); // list means *this

Problem-11: *Swap M^th^ and N^th^ number nodes*

Note: instead of doing data swapping, the node addresses must be swapped.

Problem-12: *Sort the link list on the bases of information stored in it.*

Note: Make two versions of it: one that adapts address swapping and the other one adapts data swapping.

Problem-13: *Can we apply the binary search on link list and still getting log(N) searching cost.*

Problem-14: *Null or Cycle*

You are given a linked list that is either NULL-terminated (acyclic), as shown in Figure 3.5, or ends in a cycle (cyclic), as shown in Figure 3.6.

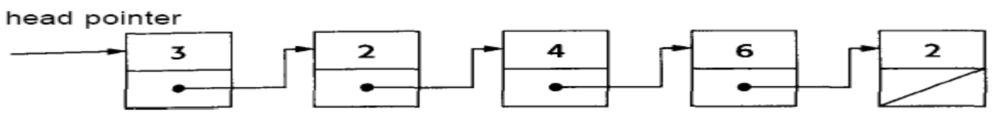


Figure 3.5 An acyclic list.

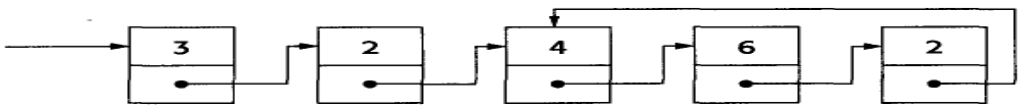


Figure 3.6 A cyclic list.

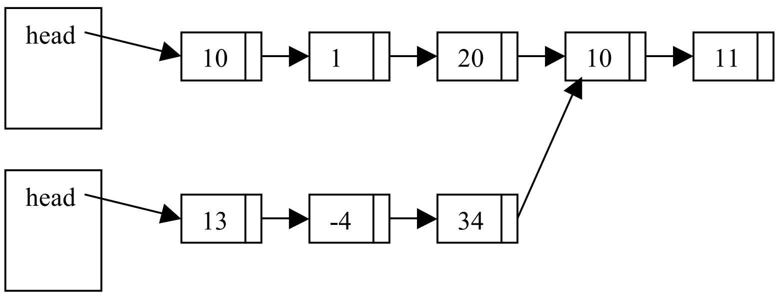
Write a O(N) time bound function that takes a pointer to the head of a list and determines if the list is cyclic or acyclic. Your function should return 0 if the list is acyclic and 1 if it is cyclic. You may not modify the list in any way.

Problem-15: *Count black and white nodes frequency*

A research student is given a singly-linked list. Each node of the list has a color, which is either "Black" or "White". He must find if there are more black nodes than white nodes, or vice versa. His advisor gives him 5,000Rs to buy a computer to do the work. He goes to the computer store and finds a slightly defective computer which costs a mere 3,000Rs. This computer has the small problem of **not being able to do arithmetic**. This means that he cannot use a counter to count the nodes in the list to determine the majority color. The computer is otherwise fully functional. He has the evil idea that he could buy the defective computer and somehow use it to do his work, so that he can use the rest of the money on enjoyment. Show how he can accomplish this amazing task. Write code for an algorithm called 'findMajorityColor' which takes a singly-linked list, L, with n nodes and returns the majority color among nodes of L. This algorithm should have the same asymptotic running time as counting the nodes (O(n)). **Note:** No arithmetic is allowed.

Problem-16: *Joining Point*

Write a function which return true if the two lists join at some point otherwise return false.

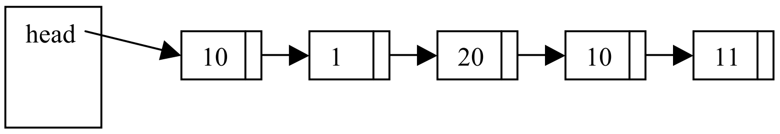


Prototype:
void LSLL::isJoining(LSLL &);

returns true for the above diagram.

Problem-17: *Middle Point*

Write a function which returns the value of the middle node in a given link list. Do it in single pass.



Prototype:
T LSLL::findMiddlePoint();

Returns 20 in case of above diagram

Problem-18: *List Flattening*

Start with a standard doubly linked list. Now imagine that in addition to next and previous pointers, each element has a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in Figure 3.3.

Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head and tail of the first level of the list. Each node is a C struct with the following definition:

```
struct nodeT
{

    struct nodeT *next;
    struct nodeT *prev;
    struct nodeT *child;
    int value;

} node;
```

This list-flattening question gives you plenty of freedom. You have simply been asked to flatten the list. There are many ways to accomplish this task. Each way results in a one-level list with a different node ordering-

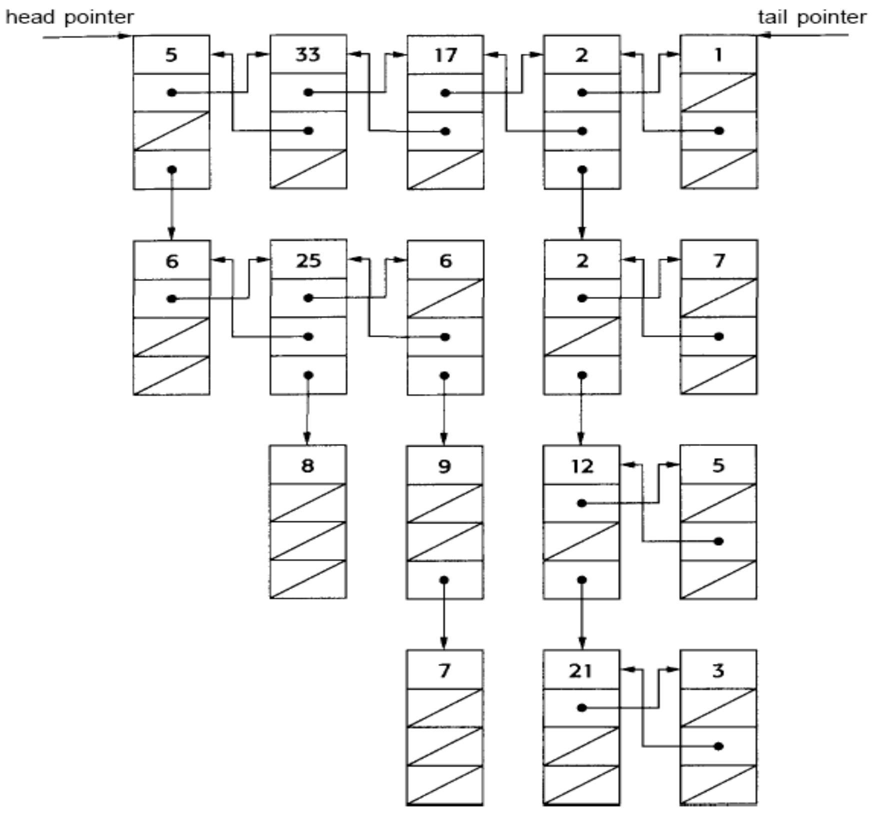


Figure 3.3 Multilevel data structure.

Start by considering several choices for algorithms and the node orders they would yield. Then implement the algorithm that looks easiest and most efficient.
Begin by looking at the data structure itself. This data structure is a little unusual for a list. It has levels and children—somewhat like a tree. A tree also has levels and children, but in a tree, no nodes on the same level are connected. You might try to use a common tree traversal algorithm and copy each node into a new list as you visit it as a simple way to flatten the structure.

The data structure is not exactly a normal tree, so any traversal algorithm you use will have to be modified. From the perspective of a tree, each separate child list in the data structure forms a single extended tree-node. This may not seem too bad: Where a standard traversal algorithm checks the child pointers of each tree-node directly, you just need to do a linked list traversal to check all the child pointers. Every time you check a node, you can copy it to a duplicate list. This duplicate list will be your flattened list. Before you work out the details of this solution, consider its efficiency.

Every node is examined once, so this is an O(n) solution. There is likely to be some overhead for the recursion or data structure required for the traversal. Also, you are making a duplicate copy of each node to create the new list. This copying is inefficient, especially if the structure is very large.

Therefore, you should search for a more efficient solution that doesn't require so much copying.
So far, the proposed solution has concentrated on an algorithm and let the ordering follow. Instead, try focusing on an ordering and then try to deduce an algorithm. You can focus on the data structure's levels as a source of ordering. It helps to define the parts of a level as *child lists.* Just as rooms in a hotel are ordered by level, you can order nodes by the level in which they occur. Every node is in a level and appears in an ordering within that level (arranging the child lists from left to right).

Therefore, you have a logical ordering just like hotel rooms. You can order by starting with all the first-level nodes, followed by all the second-level nodes, followed by all the third-level nodes, and so on. Applying these rules to the example data structure, you should get the ordering shown in Figure 3.4.

Now try to discover an algorithm that yields this ordering. One property of this ordering is that you never rearrange the order of the nodes in their respective levels. So, you could connect all the nodes on each level into a list and then join all the connected levels. But to find all the nodes on a given level so that you can join them, you would have to do a breadth-first search of that level. Breadth-first searching is inefficient, so you should continue to look for a better solution.

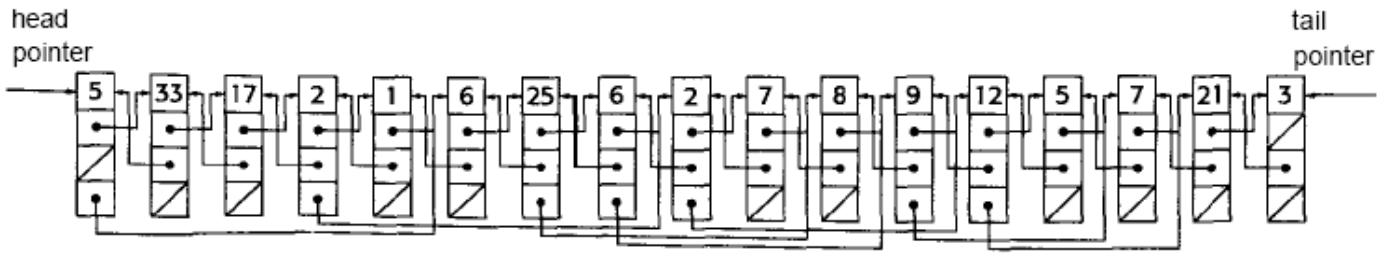


Figure 3.4 Ordering of nodes.

In Figure 3.3, the second level is composed of two child lists. Each child list starts with a different child of a first-level node. You could try to append the child lists one at a time to the end of the first level instead of combining the child lists.

To append the child lists one at a time, traverse the first level from the start, following the next pointers. Every time you encounter a node with a child, append the child (and thus the child list) to the end of the first level and update the tail pointer. Eventually, you will append the entire second level to the end of the first level. You can continue traversing the first level and arrive at the start of the old second level. If you continue this process of appending children to the end of the first level, you will eventually append every child list to the end and have a flattened list in the required order. More formally, this algorithm is as follows:

```
Start at the beginning of the first level
While you are not at the end of the first level
    If the current node has a child
        Append the child to the end of the first level
        Update the tail pointer
    Advance to next node
```

This algorithm is easy to implement because it's so simple. In terms of efficiency, every node after the first level is examined twice. Each node is examined once when you update the tail pointer for each child list and once when you examine the node to see if it has a child. The nodes in the first level get examined only once when you examine them for children because you had a first-level tail pointer when you began. So, there are no more than 2n comparisons in this algorithm, and it is an O(n) solution. This is the best time order you can achieve because every node must be examined.

Problem-19: *List Un-Flattening:*

Unflatten the list. Restore the data structure to its original condition before it was passed to Flatten-List.

You already know a lot about this data structure. One important insight is that you can create the flattened list by combining all of the child lists into one long level. Now, to get back the original list, you must separate the long flattened list back into its original child lists. First, try doing the exact opposite of what you did to create the list. When flattening the list, you traversed down the list from the start and added child lists to the end. To reverse this, you go backward from the tail and break off parts of the first level. You could break off a part when you encounter a node that was the beginning of a child list in the unflattened list. Unfortunately, this is more difficult than it might seem because you can't easily determine whether a particular node is a child (indicating that it started a child list) in the original data structure. The only way to determine whether a node is a child is to scan through the child pointers of all the previous nodes.

All this scanning would be inefficient, so you should examine some additional possibilities to find an efficient solution.

One way to get around the child node problem is to go through the list from start to end, storing pointers to all the child nodes in a separate data structure. Then you could go backward through the list and separate every child node. Looking up nodes in this way frees you from repeated scans to determine whether a node is a child or not. This is a good solution, but it still requires an extra data structure. Now try looking for a solution without an extra data structure.

It seems you have exhausted all the possibilities for going backward through the list, so try an algorithm that traverses the list from the start to the end. You still can't immediately determine whether a node is a child. One advantage of going forward, however, is that you can find all the child nodes in the same order that you appended them to the first level. You would also know that every child began a child list in the original list. If you separate each child node from the node before it, you get the unflattened list back.

You can't simply traverse the list from the start, find each node with a child, and separate the child from its previous node. You would get to the end of the list at the break between the first and second level, leaving the rest of the data structure untraversed. This solution is not too bad, though. You can traverse every child list, starting with the first level (which is a child list itself). When you find a child, continue traversing the original child list and also traverse the newly found child list. You can't traverse both at the same time, however. You can save one of these locations in a data structure and traverse it later. But, rather than designing and implementing a data structure, you can use recursion. Specifically, every time you find a node with a child, separate the child from its previous node, start traversing the new child list, and then continue traversing the original child list.

This is an efficient algorithm because each node gets checked at most twice, resulting in an O(n) running time. Again, an O(n) running time is the best you can do because you must check each node at least once to see if it is a child. In the average case, the number of function calls is small in relation to the number of nodes, so the recursive overhead is not too bad. In the worst case, the number of function calls is no more than the number of nodes. This solution is approximately as efficient as the earlier proposal that required an extra data structure, but somewhat simpler and easier to code. Therefore, this recursive solution would probably be the best choice in an interview. In outline form, the algorithm looks like the following:

```
Explore path:
    While not at the end
        If current node has a child
            Separate the child from its previous node
            Explore path beginning with the child
        Go onto the next node
```