COMS10016 | Week 05-07 | Summative Coursework 01:

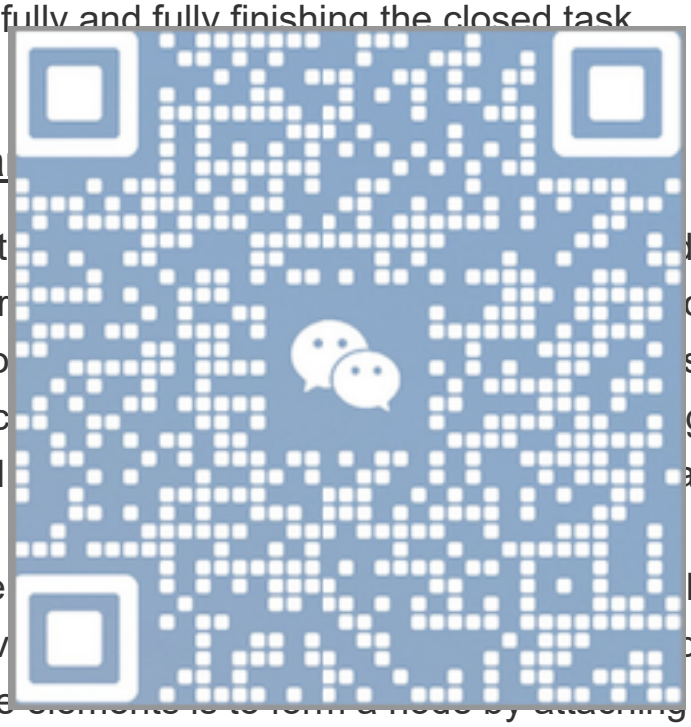
LIST CHALLENGE

This is your first assignment in imperative programming. This coursework runs over approximately the next two weeks **(Blackboard submission deadline Thursday 31st October 2024, well before 1pm)** and counts 20% towards your credit for the COMS10016 unit. Start the coursework as soon as possible and make sure you read the entire task first. Its main purpose is to show us what you have learned regarding fundamental aspects of C so far, that is writing basic syntactically and semantically correct programs with no memory leaks using pointers, strings, and bit operations. Our weekly lectures and sample code, labs and optional exercises with sample solutions built up to this point. This time you [redacted]ate with anyone or any AI. All code yo[redacted]ould be developed with some[redacted]Use our MS Teams channel or th[redacted]k questions. Mastering the course[redacted]s in your learning, show what you have[redacted] for many future assessments and viv[redacted]

Ask your questions o[redacted]hare or paste code snippets or solu[redacted] anywhere else with anyone. We will[redacted]s channel and help you along if there are questions to the assignment. Our 3h lab in Week 07 in MVB2.11/1.15 will provide in-person help with submissions and so far unanswered queries - the vast majority of the coursework should be done by this point. For someone new to C in this unit achieving an average result in this coursework may take between 5 and 15 hours if you are up-to-date in your learning, sometimes more. Since the coursework covers basics, a professional programmer may pass the coursework in well under one hour. However, to allow for inclusive assessment for students at all skill levels at this point, we have Consolidation Week upcoming for revising lectures and formative programs, and for taking as much time as needed for recap and getting up to speed with your programming and this coursework.

Again, do not copy or otherwise accept any code parts from peers or other sources and do not publish or make accessible parts of your own code anywhere. Do the right thing for yourself. The programs we may use for checking against the web, your peers and other sources are advanced. Unethical conduct and plagiarism are unprofessional and may result in 0 marks for a coursework, the entire unit, may lead to repeating a year or in repeated cases the forced end to your studies.

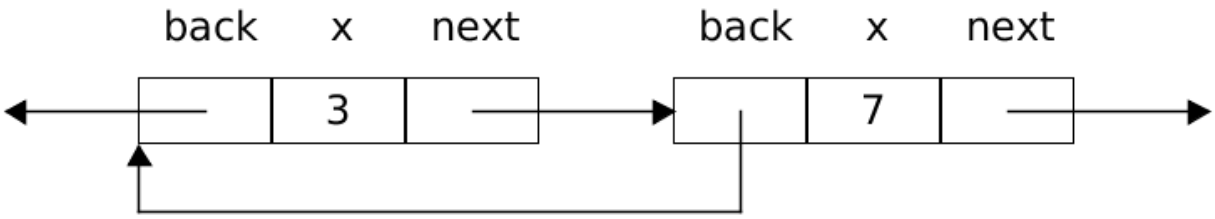
Use only standard libraries as given in the skeleton code for this task, so your code compiles and runs without linking any other libraries. Your task comes in two parts: a closed task worth the first 50% of your mark that comes with all tests so you can self-check progress and get feedback at any time, and an open-ended task. Backup your work regularly. Do not attempt the open-ended task before successfully and fully finishing the closed task.



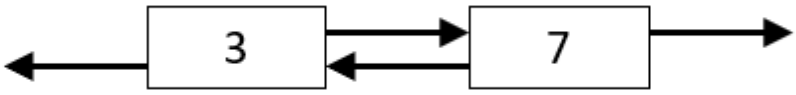
Step 1: Understa

Before you start on t derstood all lectures up to Lectur d understood all the code provided fo s. In particular, be sure you have run, c gram linkedlist.c from Lecture 15 and al exercises on Bits and Pointers.

Your task will evolve l node. Thus, let us understand and v oducing a list or sequence of variable elements is to form a node by attaching a pointer to each element, one that points to the next node and so forth. In essence, a doubly-linked list is made up of a sequence of nodes where neighbouring nodes point to each other. Each node is a structure that has a payload `x` (e.g. just an `int`) and two pointers: `back` and `next`. The `back` pointer always points to the predecessor node and the `next` pointer always points to the successor node. Two neighbouring nodes in a doubly-linked list can therefore be pictured like this:

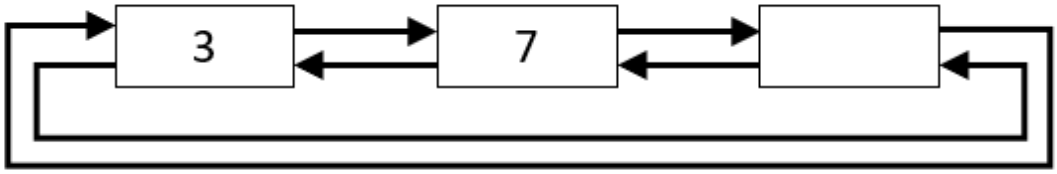


This emphasizes that a node structure contains three fields. However, for most purposes you can simplify the visualisation by depicting the above two nodes like this:

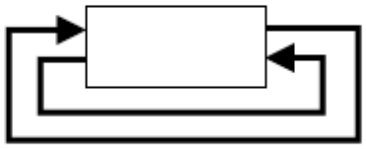


In this simplified visu node represents its back pointer, and inter. A pointer to anywhere on a node of the node.

The Sentinel Node. ement doubly-linked lists by keepin st (like the 3 node and 7 node in the pic point backward to NULL, and the last n ver, it turns out that adding an extra st implementations and For a circular, doubly-linked list our x) is linked in between the first and last payload in the list:

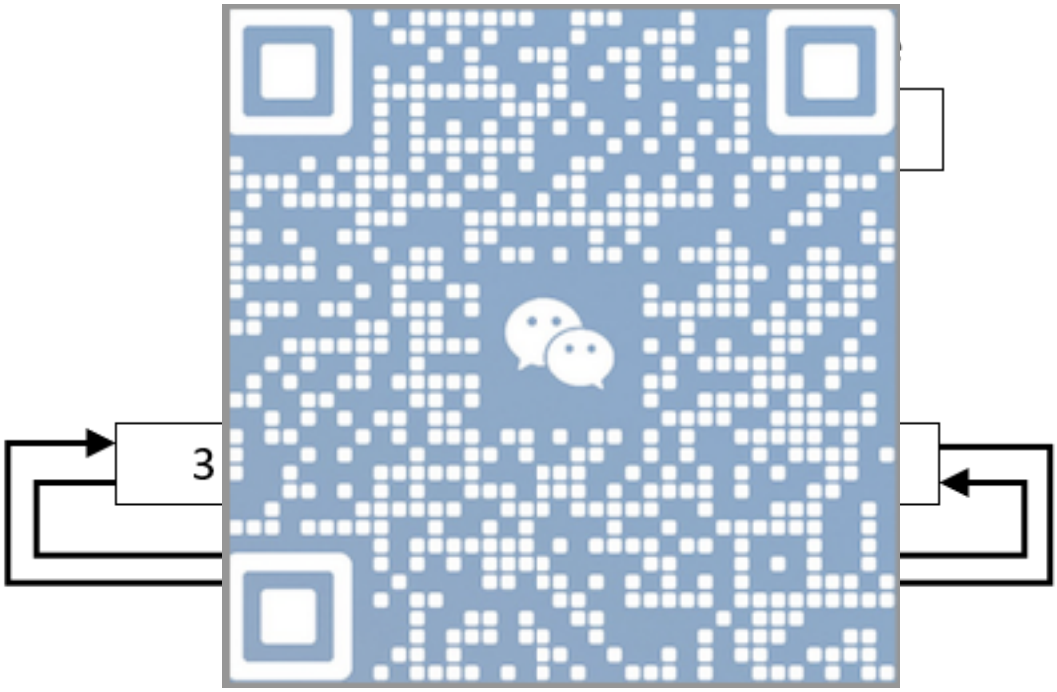


Using this idea, the nodes of a new 'empty' list with no payload nodes look like this:

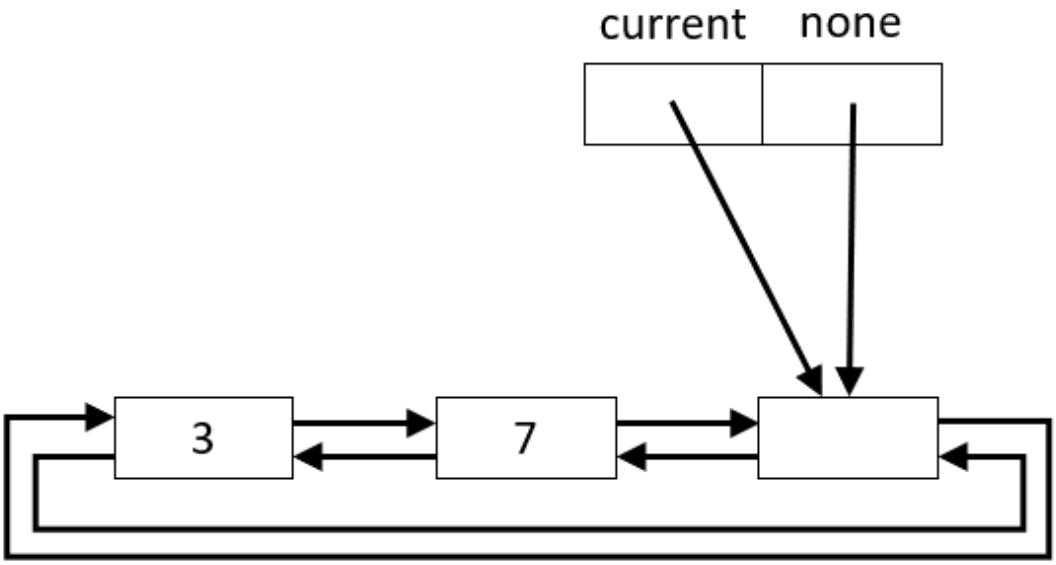


Here both the `back` and `next` pointers of the sentinel node simply point to the sentinel node itself.

The Structure `list`. To represent a list in a `list` data structure we need two node pointers: one fixed pointer to the sentinel node (called `none`) to access both list ends in constant time, and one `current` pointer that points to a current node in the list allowing for traversals:

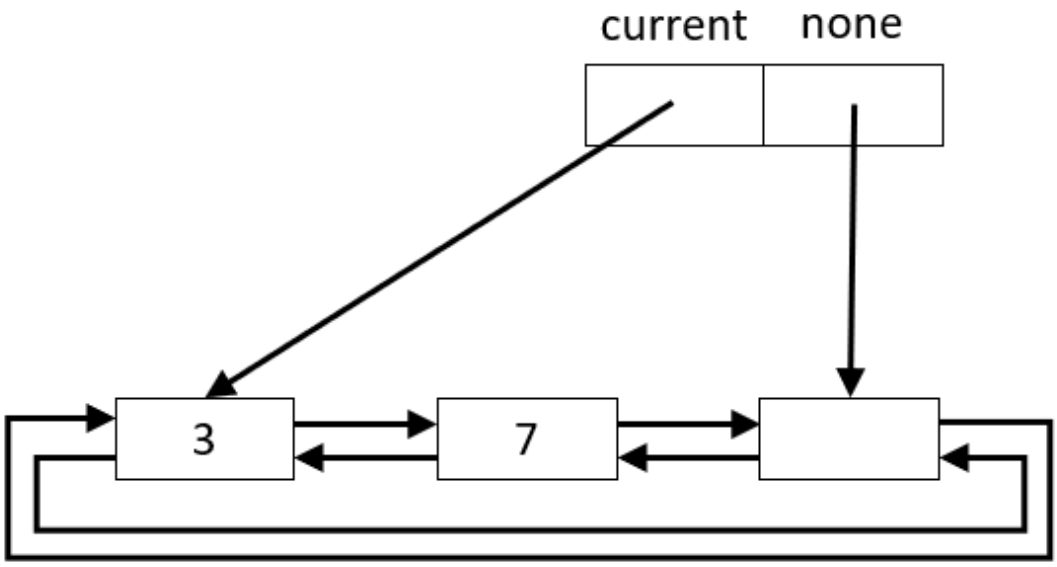


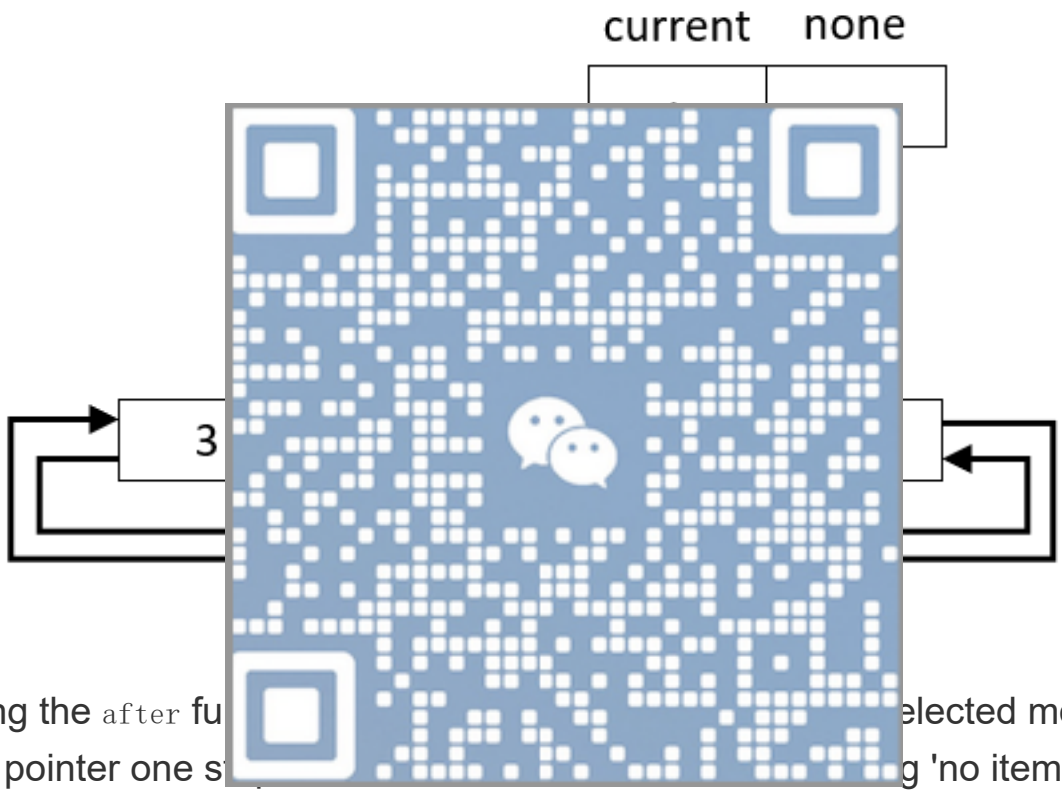
In the above image the `current` position is the node that holds 7, so payload 7 is selected. If the `current` pointer in a list points to `none` then we will interpret this as 'no payload is selected':



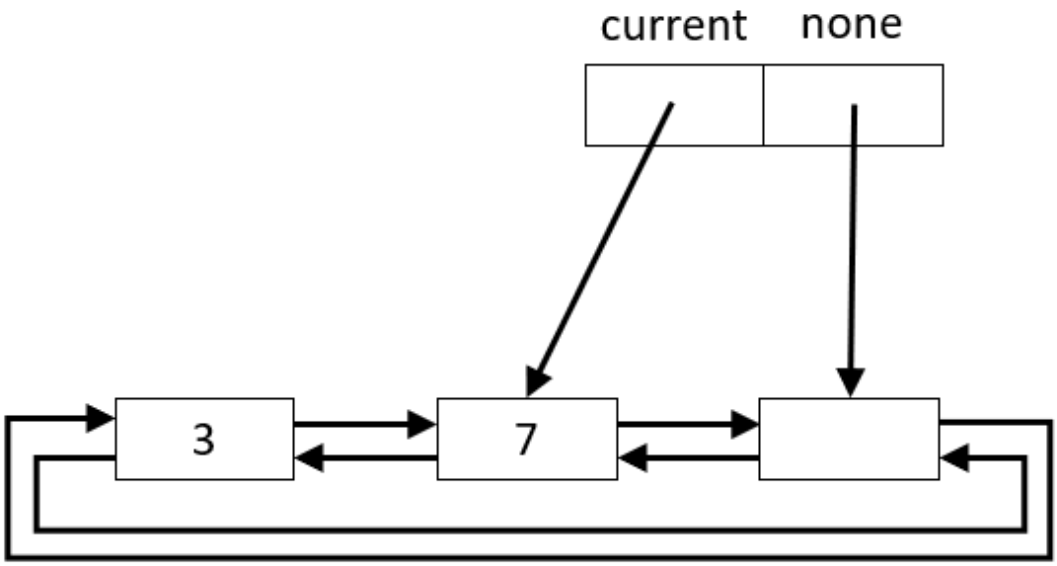
If there are payload nodes in our list, `none->next` should link to the first payload, and `none->back` should [illegible] access to both ends of the list and w[illegible]st or last node anywhere else.

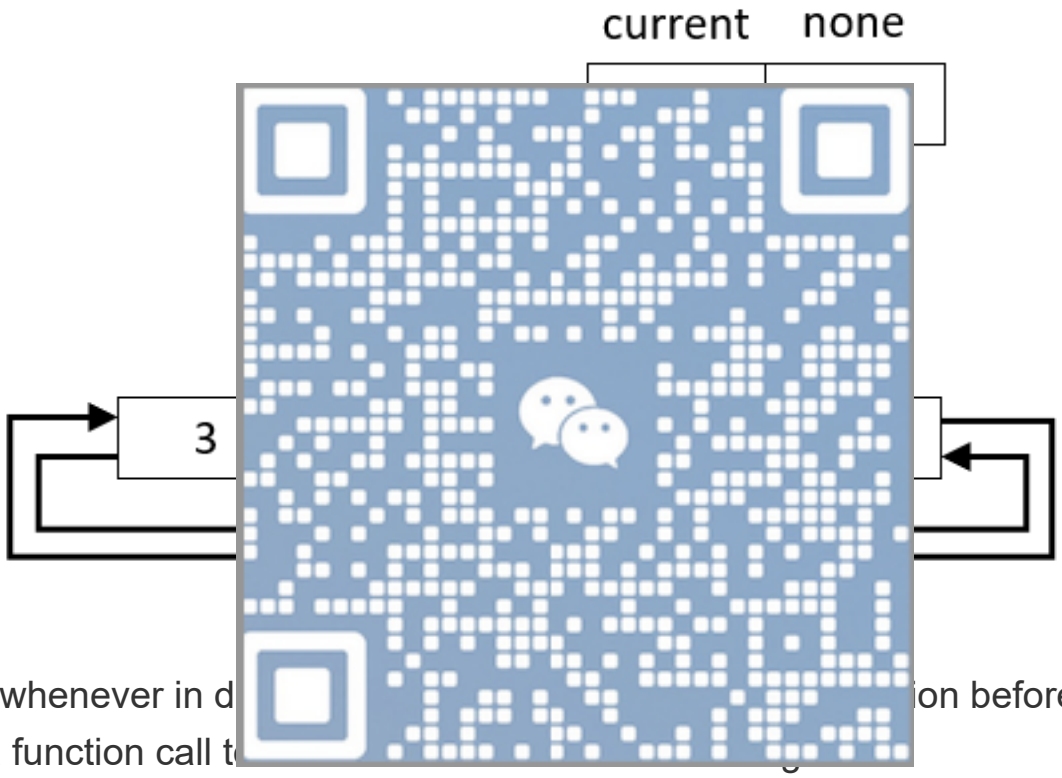
Picturing List Mani[illegible] oes to a `list`, draw a picture of the [illegible]other of the situation after the ca[illegible]er. If a payload is selected it moves the[illegible]hen applying the function to our list wi[illegible] the simple effect of moving the `current`





Applying the `after` fu elected moves the `current` pointer one s g 'no item' is selected after the call:





Thus, whenever in d ion before and after a function call t

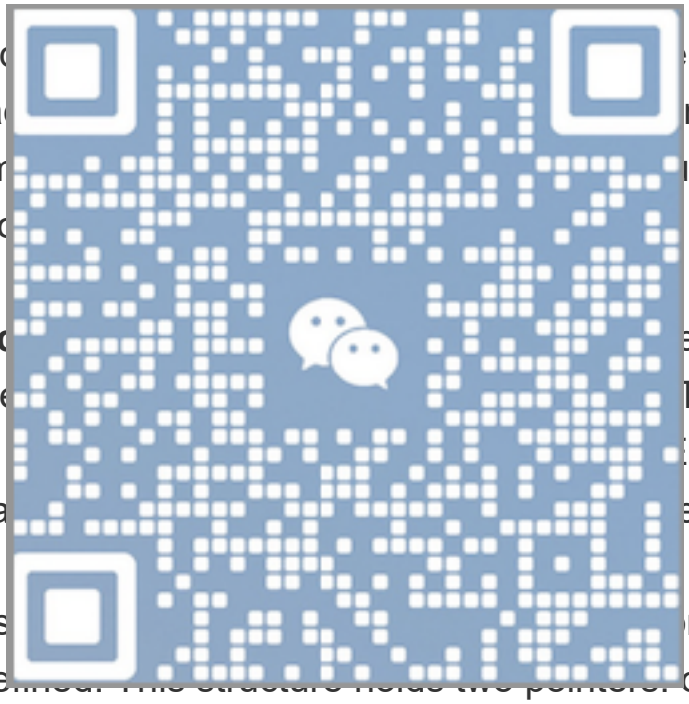
Step 2: Understand the Closed Task

Your closed task is to implement 14 missing procedures in the skeleton file list.c all of which manipulate circular doubly-linked lists with one sentinel node. The 14 missing procedures are described in detail in the header file list.h. You must use the provided files and are ***not allowed to alter any of the provided code***, only ***add*** the 14 missing functions where marked:

- list.h (header file)
- list.c (skeleton)
- Makefile

The header file `list.h` forms an API, which you should read ***carefully*** because the comments describe what the functions you will have to implement in `list.c` have to do. The `list.c` file has just two data structures `node` and `list` which you must use, and a lot of tests. The program as given to you will not compile initially. So, our first task is to produce a compiling skeleton by studying the signatures of the 14 missing procedures and accordingly defining some initial dummy functions.

Step 3: Create a Compiling Skeleton



For a start, downloa lopment machine that only you have a rn `list.c` into a full skeleton program nderstanding all of the technicalities

Two Key Data Struc e `node`s which make up the list is de The node structure `struct node` ach node is used to hold a payload `x` a es (`next` and `back`) which define the list `list` represents a list and is essential s mething to the user which is well de one to the sentinel node of the list and one to the currently selected node. Read the code comments about the `list` structure carefully. ***You will have to use the two data structures exactly as described to comply with the tests.***

Define Dummy Functions. Write a minimal dummy definition of each of the 14 functions mentioned in the header file in your file `list.c`. The safest way to do that is to copy-and-paste a function's declaration from `list.h`, then replace the semicolon by curly brackets. If the function returns anything other than `void`, add a return statement which returns the easiest temporary value of that type you can think of (e.g. `NULL` for a pointer, `false` for a boolean). For functions