Introduction

Linked lists are an important and fundamental data structure within computer science. They are a dynamic (S, 2024)type of data structure that can grow or shrink as needed, making them highly flexible for applications where the size of the data set is constantly changing. Linked lists are particularly useful in scenarios where memory management is crucial, as they allow for efficient insertion and deletion of elements without the need for continuous memory allocation, which is common with arrays.

Unlike arrays, which store elements in contiguous memory locations, linked lists store their elements in separate memory locations, with each element (or node) linked to the next via a pointer. This structure eliminates the need to shift elements when adding or removing data, making linked lists more efficient when dealing with dynamic datasets where frequent modifications occur.

This assignment focuses on designing and implementing support functions to the main biological simulation, for the simulation to be able to utilize linked lists. The goal is to implement support functions. The dynamic nature of the structure allowing for efficient modeling of population changes, the linked list structure can adjust accordingly without requiring the reorganization of memory.

The linked list is particularly well-suited for this simulation because the population size fluctuates over time. As the population increases or decreases due to bacterial replication and death, the linked list can dynamically allocate or deallocate memory as needed, making it an ideal tool for managing such a complex and evolving system.

Technical Background.

A linked list is a fundamental data structure. This structure consists of "nodes" that are connected through "pointers." Each node contains some data and a pointer to the next node in the list. The "pointer" points to a memory address in the memory where the linked list is stored. It is thus divided into two fields: one field holds a pointer to the next node, and the other field holds the data. Unlike "static" data structures, linked lists can grow or shrink as needed, without having to move elements around in memory. As a result, it can grow in step with the demand, which is much better compared to arrays, where a lot of "pre-setup" is required.

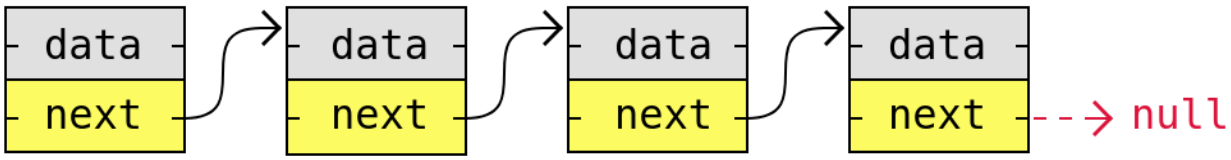

Figure 1: On w3schools.com, explains a linked list visually, especially the tail node, shown differently at the end with the null pointer. (w3schools, n.d.)

Design and Implementation

This experiment's design is based on using linked lists as the defining data structure. This represents the dynamic nature quite clearly. Each bacterium is a node within the list. The list itself shrinks or grows over time dependent on the deaths or life rate for the population.

A linked list has nodes, these nodes have the typical structure, of being divided in two, of a data pointer and a next-node pointer. Each node is represented by a struct that includes one pointer to the data's address and another pointer to the address of the next node or to NULL in the case it is the last element.

Linked lists iterates from the head node towards the tail node. The head node always points towards the next node, that node points towards the next node etc., eventually this ends at the tail node, which points towards NULL. The linked list ends here. The list is defined as a struct that contains two pointers, and the size of the list. We use pointers to define where the head and tail node lies.
The reason we only define these pointers is because, its linear, the head node points to the next node, and that node points to the node after that. Which after whatever number of nodes, eventually points towards the tail node. Thats the reason we only need those two pointers, which is to traverse the complete list. Size, however, is required by other functions, such as list_size. This function returns the size of the specific list in question.

Create Node function is a function that creates nodes, upon creation the specific node is assigns a pointer that points to the memory address of intended data. This function has a fail-check system. If the allocated memory was unsuccessful for the new node, it returns NULL. In the case memory is allocated, it assigns the pointer that points the memory address of the specified data. This node is then set to point to NULL, which allows it to be later set into a list. In the end it returns the memory address for the new node.

The “list create” function allocates memory for a new list if it is successful. In the case where it returns NULL, the allocation of data has been non-successful. If successful, however it sets the head and tail pointers to null and initializes the variable size to 0. Afterwards the newly initialized address for the empty list is returned.

The list destroy function frees up memory for specified lists. In full detail the function frees the memory, memory being the key point here because the free( ) (kamleshjoshi18, 2024) (Chandra, 4) function is used. The structure, and the allocated memory is freed, and the pointers no longer point towards any specific memory addresses, the data however still exists originally it was never stored lists it was pointed at.
This is done by using a local variable, in our case this variable is used to point to the current node. In addition to this it helps us to manage the while loop we use to free or de-allocate memory, which checks if the current variable has a node assigned and frees it if so, after which it selects the node after the current node. Which leads to the while loop continuing till the entire structure is clear, in the end it de-allocates the memory for the list.

The “list_addfirst” function creates a new node using the create node function and adds it to the front of the list. If successful, the new node's next pointer is set to the current head of the list, and the new node becomes the new head node. If the list is empty however, the new node additionally becomes the tail, and the size of the list is increases by one, in the size variable.

For a function that adds the node at the end of the list, we need a function named “list_addlast.” in theory it does the same as the former function, it creates a new node but adds it to the end of the list, the size of the list is again increased by 1 integer value. This is done by first having a failsafe section so that we may check the list size before making any changes, such as adding a node to the end of the list. As this happens, we check the size of the list, which determines whether we move forward or not. If the size is 0, this means the fail-safe function detects that there is no individual node existent, and therefore it must assign a node to the list as both the head and the tail node. If the size is bigger than 0 then the node is assigned after tail node, and then assigned to be the new tail node.

We also have a list_find function, this function allows the program to find the memory address for the specific node that the program is looking for. This function traverses through the whole list until it finds the node that points to the memory address of the data. This is done by using a local variable that sets the memory address for the focused node as the memory address for the local variable after this it checks if this node has the specific data, if it does not it does not return anything, and it continues the loop.

If it does not find the specified data in the node it is currently focused on it goes to the node after that. In the case of matching data, it returns the local variable with the memory address for the previous node which points to the node with the data we are looking for.

We have a function that handles the operation of destroying a node. This is done by a function called “list_remove” this function removes nodes based on what specific occurrence or situation we are in. if the head node matches the memory address of the

data the program is looking for it will set the next node to be the new head node and decrease the size by 1. However, in cases besides the head node, if the “list_find” function does not find a memory address, then it does not return anything. In the situation that the “list_find” function returns the memory address for the node it sets the pointer for the current node to the next-next node, and then the free () function deallocates the memory for the node after the current node.

The function that oversees sizing up lists, is called “list_size”. This function returns the list size using the information stored in the memory address of the list structure. This is done by collecting the struct and the list pointer information. After which returning the size variable.

An iterator traverses along the linked list, from one node to another. In the code we have functions that create iterators, these iterators require memory and for that we use the malloc function. We also must select which list we want to go through; this is done by selecting the memory address for the list and the head node which is stored in a struct. In the end the struct is returned.

We also need something to delete the iterator. We are using the free()function, this de-allocates the memory. The program uses free () to delete the information stored within the struct and the struct itself. An iterator scrolls through the linked lists, and this happens through a list next function. The iterator node is at the current node, which is updated to be on the next node in the list. Then we need a function that resets the alignment of the iterator, for this we have a function that starts the iterator again on the first node. By going inside the iterator's memory address, we can set the node to the first element or head node.

Discussion

Malloc () is how the memory was dynamically allocated, it made it possible for efficient memory management of the lists and the nodes. Using void* for data allowed for linked lists to be quite flexible allowing them to hold any kind of data, whether it be integers, characters, or floats. The function, list_addfirst function was quite efficient, however the list_addlast function could have been formed differently. Thankfully, since I made fail safes, I could safely ensure the program runs without memory failure.

On the topic of memory failure, it is however a flaw, to not have the list_addfirst and list_addlast function return NULL when results yield inconclusive of what is required. This means the error handling could be improved. Alongside this the code of “list_addfirst” and list_addlast could definitively be more efficient if refactored to improve, by avoiding needless repetition and ease of upkeep.

The utilization of linked lists in the assignment was quite handy, it gave me quite an insight into the theoretical of linked lists and what is behind them. Such as for how they allow dynamic memory allocation and ease of insertion and deletion.
For example, I noticed how valuable linked lists can be when it comes to managing collections of data, wherein the size changes frequently. Such as when removing or adding nodes without needing to reorganize that memory, unlike arrays. The flexibility of that memory usage was shown to be blatantly apparent during the implementation. However, I also realized the difference between data structures. Such as the slower access time to elements within linked lists compared to arrays. Traversing the nodes from head to tail. Overall, it made me really grasp my understanding of linked lists, in terms of both practical and theoretical foundation.

Conclusion

To sum up, this task has directly made it clearer to understand how linked lists can be properly utilized to store data and expand knowledge further of the topic. This furthered

comprehension of key concepts like dynamic memory management with the use of “malloc()”. Node traversal and list manipulation was also further explained. The execution of this simulation proved to be successful ergo validating the code and confirming the overall success.

Bibliography

Chandra, A. (4, May 2023). *scaler*. Retrieved from scaler.com: https://www.scaler.com/topics/free-in-c/

kamleshjoshi18. (2024, May 29). *geeksforgeeks*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/free-function-in-c-library-with-examples/

S, R. A. (2024, September 26). *simplilearn*. Retrieved from simplilearn.com:

https://www.simplilearn.com/tutorials/c-tutorial/singly-linked-list-in-c#:~:text=Linked%20List%20In%20C%20is,random%20address%20in%20the%20memory.

w3schools. (n.d.). Retrieved from w3schools: https://www.w3schools.com/dsa/dsa_theory_linkedlists.php