Temasek Junior College

2022 JC1 H2 Computing

Data Structures 4 – Linked Lists (I)

§4.1 Introduction to Linked Lists

We have thus far made use of static arrays to implement the abstract data types of stacks and queues. When implementing stacks and queues in Python, we simulated these static arrays using the list structure.

Such implementations of stacks and queues are premised upon an assumption that data items are stored in consecutive (adjacent) locations in the memory. However, this may not always be appropriate and/or not always possible.

An alternative strategy is to have individual data items stored in whatever available location in memory (as determined by the OS) and link these individual data items into an ordered sequence using pointers.

Such a strategy results in an abstract data type known as a **linked list**, which is a sequence of connected data items, where each data item contains the data itself together with a pointer to the next item. There may be an additional pointer to the previous item.

In a linked list, data items need not be stored in order in consecutive (adjacent) locations in memory. Despite so, data items can still be accessed in order.

Due to the nature of how a linked list is created, it is a **dynamic data structure**. There is no need to pre-allocate a set amount of memory for the linked list prior to program execution (at compile time). Instead, a linked list makes use of **dynamic memory allocation**. The size of a linked list is thus able to change at run-time depending on the quantity of data items to be added.

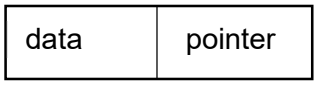
(See **Data Structures 1 – Static and Dynamic Data Structures** to recapitulate what is static memory allocation and dynamic memory allocation.)

You can imagine a linked list as a chain where each link is connected to the next one to form a sequence with a start and an end.

§4.2 Structure of a Linked List

Each element in a linked list is called a **node**. Each node stores:

- the **data** item(s) pertaining to the element,
- a **pointer**, which is a variable that stores the address of the location in memory of the node it points to.

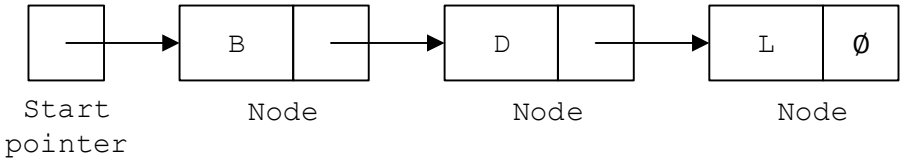


A pointer that does not point to any location in memory i.e. does not store any address, is a **null pointer**.

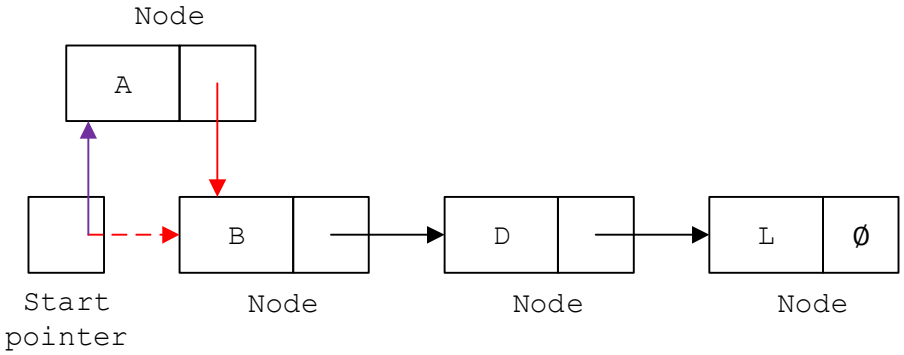
A pointer that stores the address of the location in memory of the first element is called a **start pointer** or **head pointer**. The first element of the linked list can also be referred to as the **head** of the list.

Example 1

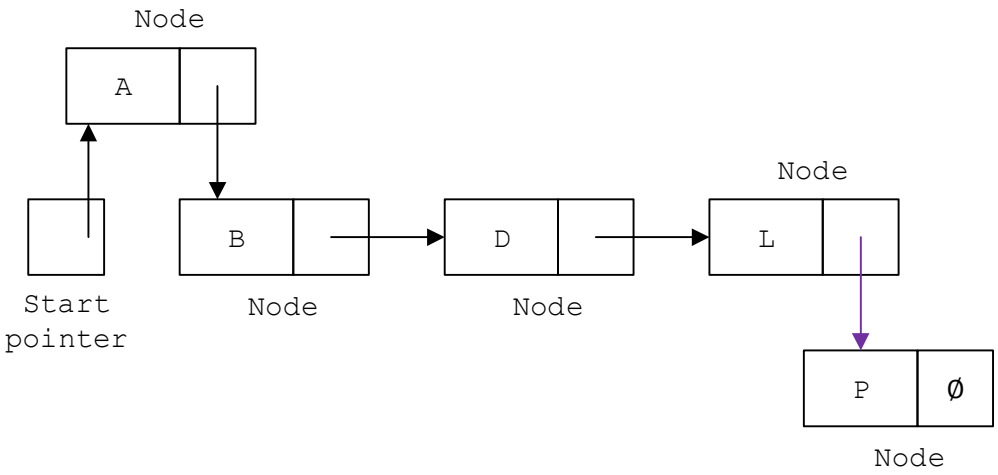
- This is a simplified illustration of a linked list structure.
- The letter in a node represents the data items associated with that node.
- There may be one or more pieces of data items associated with each node.
- The arrows represent the pointers.
- For simplicity, the pointer shown below does not have any values to represent the address of the location in memory where the node it is pointing to is stored. It only shows where it conceptually links to.
- The symbol ∅ represents the null pointer.



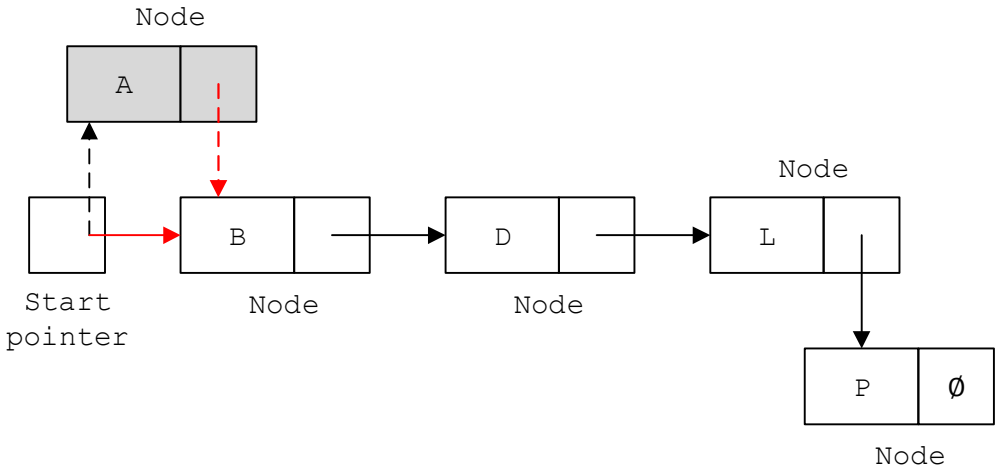
- Add new node A to the beginning of the linked list.
- The contents of the start pointer are copied into the pointer of the new node A.
- The start pointer is set to point to the address of the location in memory of node A.
- The pointer of the new node A will now point to the address of the location in memory of node B.



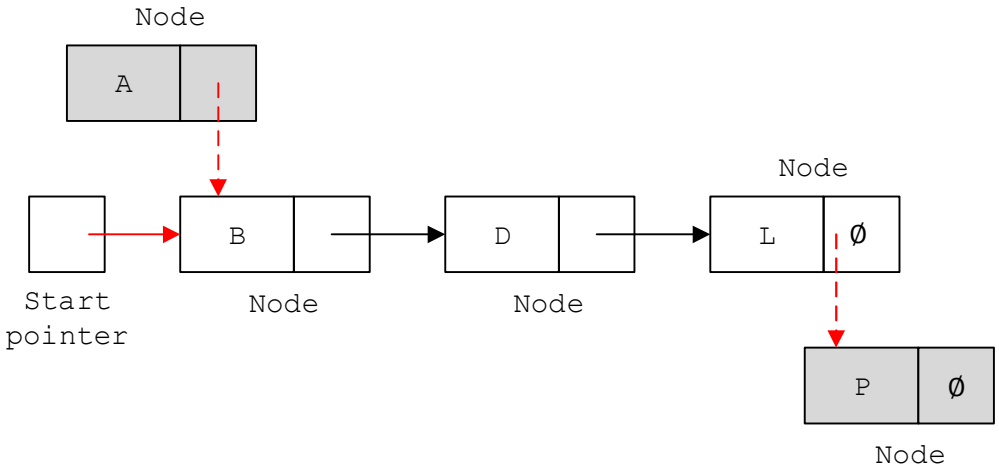
- Add new node P to the end of the linked list.
- The pointer of node L is set to point to the address of the location in memory of node P.
- Node P now contains the null pointer.



- Delete the current first node, node `A`.
- The contents of the pointer for the node to be deleted (node `A` in this case) are copied to the start pointer.
- The start pointer is set to point to the address of the location in memory of node `B`.



- To delete the last node, the pointer of the node before it is set to null.

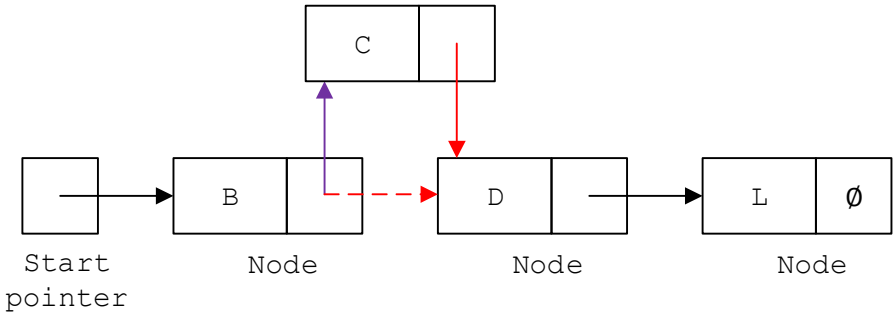


Example 1 provided schematic illustration of the structure of a linked list and how nodes can be added and deleted from the front and the end of the linked list.

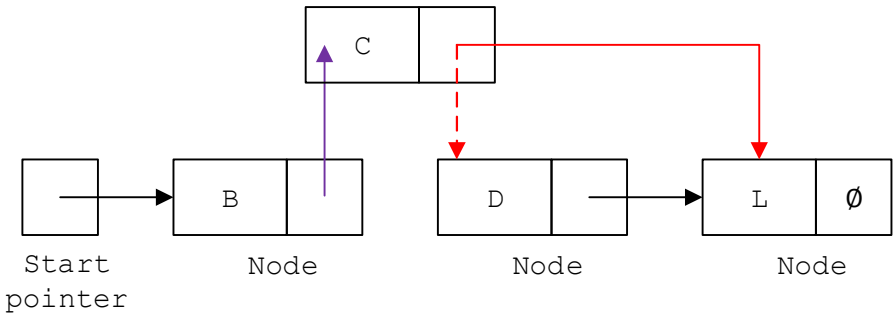
There may also be instances where a new node is required to be added or deleted from between two existing nodes.

Example 2

- Add a new node `C`, in between nodes `B` and `D` of the linked list in **Example 1**.
- The contents of the pointer in node `B` are copied into the pointer in node `C`.
- The pointer in node `B` is set to point to the address of the location in memory of node `C`.
- The pointer in node C will point to the address of the location in memory of node `D`.



- Delete node `D`.
- The contents of the pointer in node `D` are copied into the pointer of node `C`.
- The pointer of node `C` now points to the address of the location in memory of node `L`.



§4.3 Linked List vs. Lists

In actual applications, there might be a need to handle large volumes of data. When there is a requirement to store the data as a sequence, a linked list is preferred over a list.

When elements need to be added, deleted, or simply re-ordered, only pointers need to be changed in a linked list. In a linear list, all elements would have to be moved, which might not be the most efficient when the volume of data is large.

While efficiency is achieved in the movement of data in a linked list, additional memory is however incurred for the storage of the pointers.

Exercise

Implement Linked List using OOP in Jupyter Notebook.

References

Isaac Computer Science – Linked Lists
https://isaaccomputerscience.org/concepts/dsa_datastruct_linked_list?examBoard=all&stage=all&topic=data_structures