Understanding Memory Allocation in Linked Lists: How Nodes and Addresses Connect

Table of Contents

Linked lists are fundamental data structures in computer science, offering a flexible and dynamic way to organize data. Unlike arrays, which store elements in contiguous memory locations, linked lists consist of nodes scattered throughout memory, connected by pointers/references. This article covers various aspects of memory allocation in linked lists, exploring how nodes are created, stored, and linked together through memory addresses. Visual representation of how linked list nodes are stored in memory is also provided.

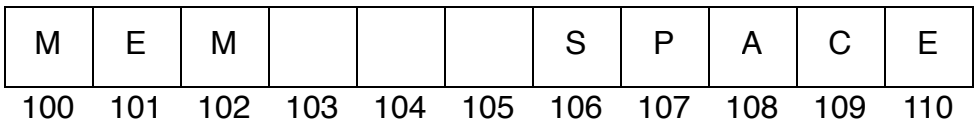
Prerequisite: Basics of linked list data structure.

Basics of Memory Allocation

Memory allocation in computer science deals with how an operating system or program manages and organizes its available memory resources. In this section we will cover the basics of memory allocation, focusing especially on the heap memory and how objects are stored and managed.

The Heap: A Sequence of Memory Slots

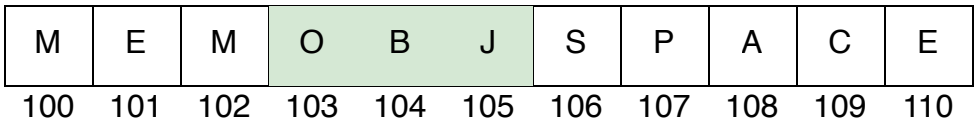
Think of the heap as a large, contiguous area of computer memory. We can visualize it as a long sequence of slots, each with its own unique address. Using the address of each memory slot, we can access the underlying data stored at the corresponding slot. These addresses are typically represented as numbers, with each subsequent slot having an address one higher than the previous one. For example below is a section of a heap address which slot addresses starting from 100:



In the image shown above, the letter `M` is stored at memory address 100, letter `E` is stored at memory address 101 and so on. For objects/data which require more than one slot to be stored in memory, the system allocates a series of consecutive slots based on the object's size. The address of the first slot is then returned as a reference to that object.

Storing and Retrieving Objects

Let's say we want to store an object that requires 3 memory slots. The system will scan the memory slots and look for 3 consecutive empty slots and then allocate the slots for the object to be stored. For example, the system might allocate slots 103, 104, and 105 for this object. The program would then receive the address 103 as a reference to the object's location in memory.



When the program needs to access or modify the object, it uses this address (103) as the starting point. Since the system knows the object's size, it can read all the necessary slots (103, 104, and 105) to retrieve or update the object's data.

Memory Allocation in Linked Lists

Node Creation

When creating a new node in a linked list, memory is allocated dynamically using heap memory. This allocation process involves reserving a continuous block of memory in heap that is large enough to store two key components:

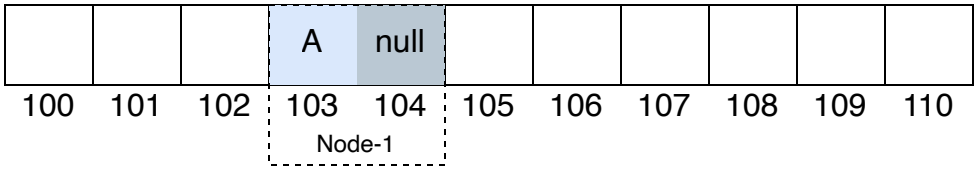
1. The data element of the node

2. A pointer to the next node

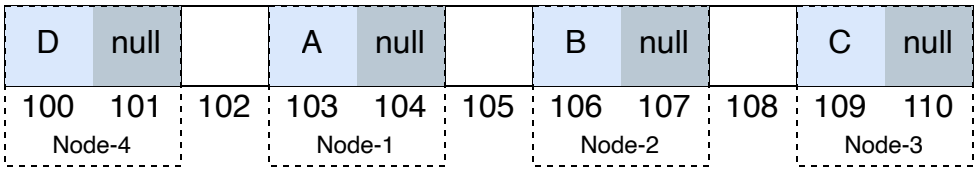
For example, if we're creating a linked list of alphabets/characters, each node would typically require memory for:

- An ASCII character value (usually 4 bytes)
- A pointer to the next node (usually 4 or 8 bytes, depending on the system architecture. Let's assume that this is a 32 bit system and 4 bits are required for storing pointer/reference address to the next node).

When a new node is created, the system allocates this continuous block of memory (in this case, 8 bytes or two slots of memory) on the heap. Initially, the address of the next node is set to `null`. In memory terms, `null` is represented by an invalid memory address (often `0x0` or `0`), indicating that the current node is not pointing to any other node yet. Below is how the newly created node looks like in memory:



Let's now allocate memory for 3 more nodes for storing characters `B`, `C` and `D`. Below is how all the linked list nodes are stored/represented in memory:



Notice that even though we want the nodes `A`, `B`, `C`, `D` in that sequence in the linked list, the created nodes need not be stored in the same sequence in memory. It can be in any order in memory. The actual order in the linked list depends on how we link these nodes.

Below is how to reserve memory and initialize linked list nodes in various programming languages:

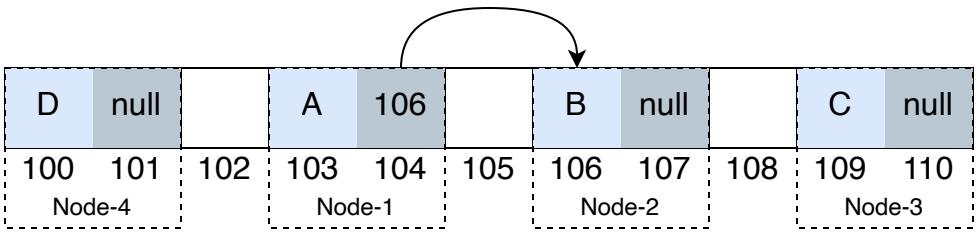
python c cpp javascript java

```
# Node class to store data and link to the next node
class Node:
```

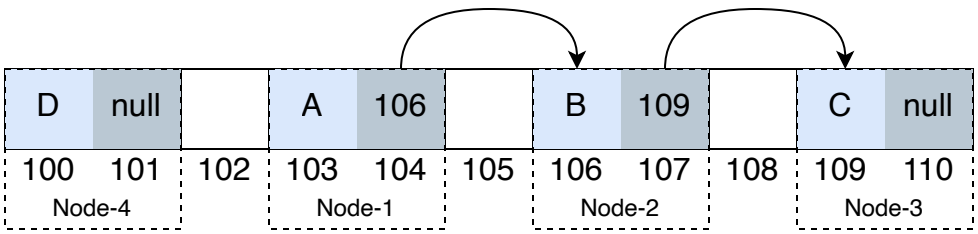
```
    def __init__(self, data):
      # Initialize the node with given data
      self.data = data
      # Set the next pointer to None initially
      self.next = None

# We need to create the nodes first before linking them
node1 = Node("A")
node2 = Node("B")
node3 = Node("C")
node4 = Node("D")
```

Linking Nodes

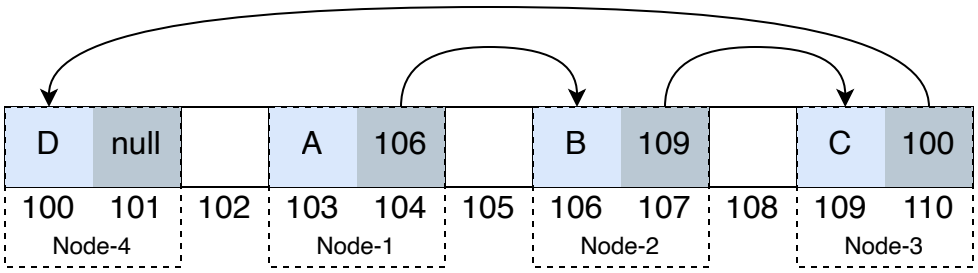
In the previous step, we looked at how we can allocate and fill memory required for each node. Now we can start connecting them to form a linked list. This process involves setting the "next" pointer of each node to the memory address of the subsequent node in the list. Let's first connect node `A` s next pointer to node `B` s address by using the expression `node1.next = address of node2` :



Now let's connect the node `B` s next pointer to node `C` s address using the expression `node2.next = address of node3` :



Now let's connect the node `C` s next pointer to node `D` s address:



Now, address of `A` (103) is stored as the head node of the linked list. Using this address we can access all the linked list nodes and perform all linked list operations. In the memory layout, although the sequence of linked list nodes is `D, A, B, C`, since we start from head node address (103), the sequence depends on how each node is linked, irrespective of how those nodes are arranged in the memory slots.

Below is how we programmatically link linked list nodes:

```
node1.next = node2 # A->B
node2.next = node3 # B->C
node3.next = node4 # C->A

# node1 is the head node in the linked list
head = node1 # Node A
```

Click here for a more detailed article on how to create nodes and connect them together to form a linked list.

Manual vs Automatic Memory management

When we allocate memory for linked list nodes on the heap, we should also make sure that the allocated memory is freed when the nodes are deleted or the linked list is no longer required. In some programming languages, the freeing of allocated memory needs to be manually done. In some languages with garbage collection, the freeing of allocated memory is done automatically.

For example, languages like C and C++ do not have garbage collection and memory needs to be managed manually. For cases like these, programmers are responsible for allocating and deallocating memory. Below is an example on how we do this in C language:

- **Allocation**: `Node* node = malloc(sizeof(Node));` - This allocates memory for storing a linked list node returns the address of allocated memory.

- **Deallocation**: `free(node);` - This releases the previously allocated memory for the node. This needs to be performed when the linked list node is no longer necessary.

Most of the bugs/issues with linked lists occur due to memory which is not deallocated properly for all the nodes which are removed from the linked list. This causes memory leaks and the program would crash over time due to high memory consumption.

Some programming languages, like Java or Python, use automatic memory management systems (often called garbage collectors). These systems track which objects are still in use by the program. So when an linked list node is no longer referenced by any other nodes/variables, the system automatically frees up the memory it occupied. This helps prevent memory leaks and makes programming easier.

◀ How to Create and Connect Nodes in a Linked List in Various Programming Languages

Different Types of Linked List Data Structure ▶

Home / Computer science / Data structures /

Arrays ▼

Linked Lists ▲

- Overview
- Basic Implementation
- Memory Management
- Different types
- Applications
- Basic Operations ▼
- Basic Problems ▼
- Advanced Problems ▼
- Advantages and Disadvantages
- Linked Lists vs Arrays
- Singly Linked List Implementations ▼

Hash Tables ▼

Stacks ▼