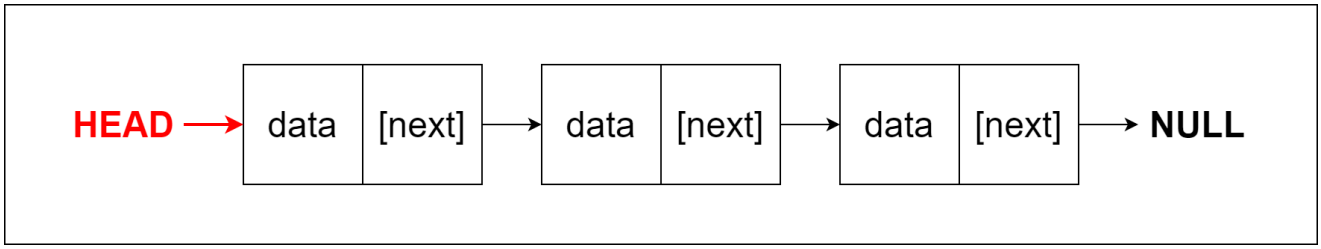
Alright, let's dive into the fascinating world of linked lists! A linked list is a linear data structure made up of a series of connected nodes. Each node contains two important pieces of information: the actual data we want to store and the address of the next node in the list.



Think of it like a treasure hunt game! Just like in the game, where each clue leads to the next one, in a linked list, each node contains data and a clue (address) that leads us to the next node in the sequence.

Now, to get started with our treasure hunt, we need to know where to begin. So, we give a special name to the address of the first node, and we call it the HEAD of the linked list. Similarly, the last node in the linked list can be easily identified because its "next" portion points to NULL, indicating that there are no more nodes after it.

Linked lists come in different types: singly linked lists, doubly linked lists, and circular linked lists. In this lesson, we will focus on the singly linked list, which means each node only points to the next node in the sequence.

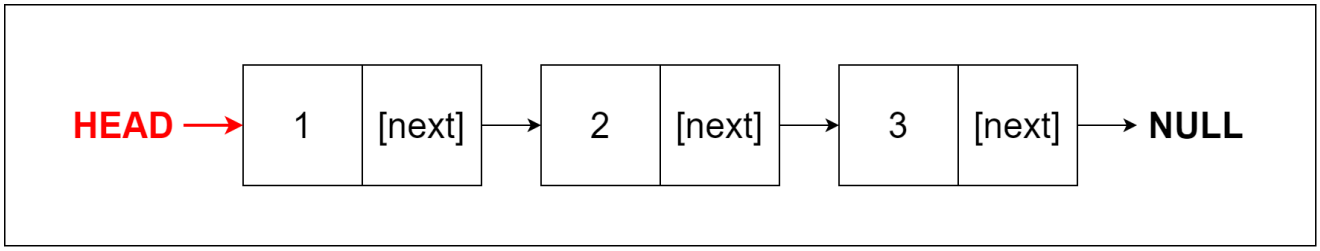
Just like in a treasure hunt, where you follow the clues one by one to find the ultimate prize, a singly linked list allows us to efficiently traverse through the data in a sequential manner. So, let's embark on this exciting journey of learning about singly linked lists!

Representation of Linked List

Let's take a closer look at how we represent each node in a linked list. Each node has two important components:

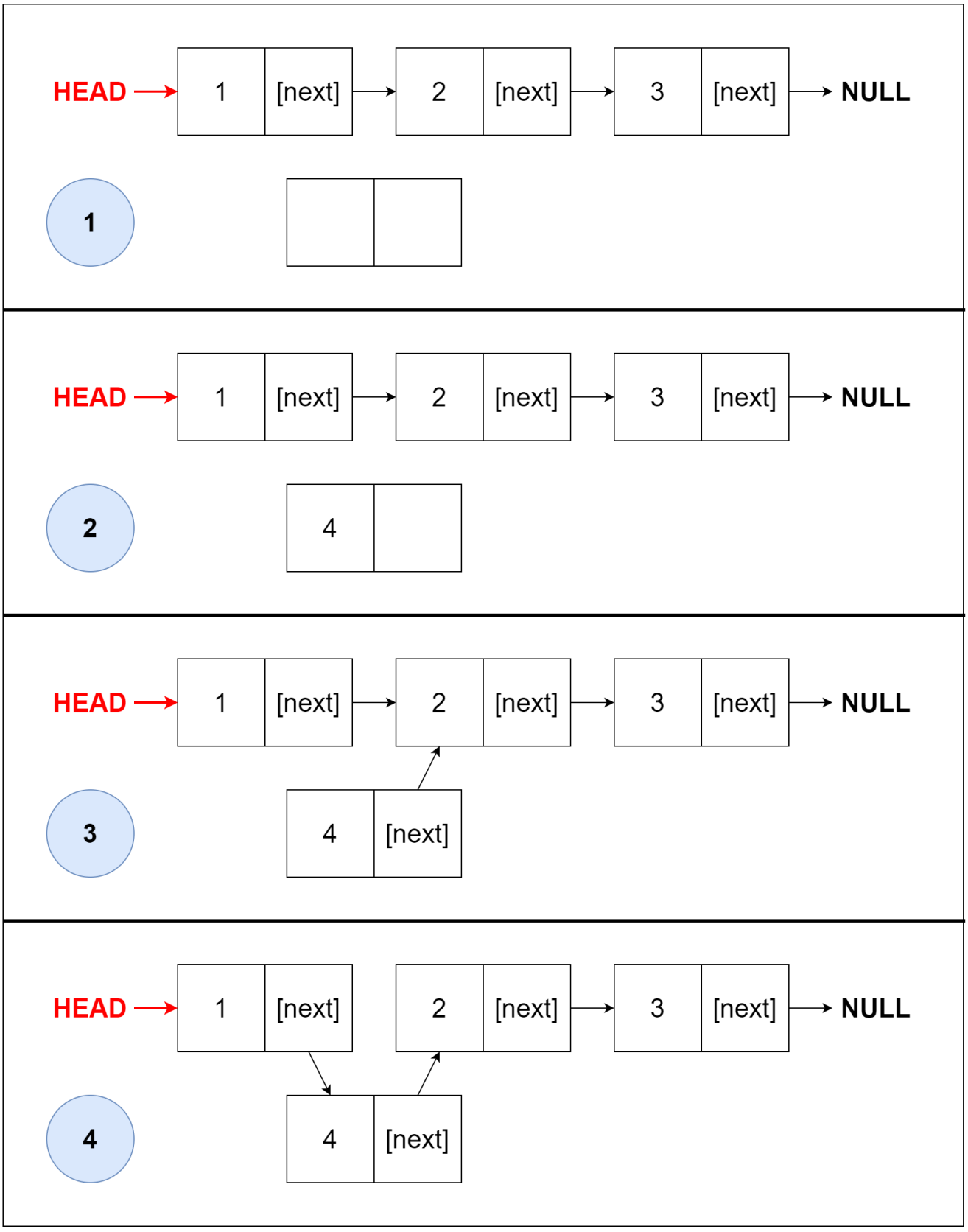
1. A data item: This is where we store the actual information we want to keep in the linked list. It could be anything, like a number, a word, or even a more complex data structure.
2. An address of another node: This is like a clue that tells us where the next node in the linked list is located. It's like following a trail of breadcrumbs that leads us to the next piece of data.

To better understand this, let's create a simple linked list with three items. Imagine we have three nodes, each containing a data item and a pointer to the next node.



Now, here's the real power of a linked list. Unlike an array, where elements are fixed in place, we can easily break the chain and rejoin it in a linked list. For example, suppose we want to insert an element with the value 4 between 1 and 2. Here's how we do it:

1. We create a new node and allocate memory for it.
2. We set its data value to 4.
3. We point its "next" pointer to the node containing 2 as its data value.
4. Finally, we change the "next" pointer of the node containing 1 to the new node we just created.



And that's it! We have successfully inserted the new element without needing to shift any other elements around, which is what we would have to do in an array.

Linked List Utility

Linked Lists are used in almost every programming language you can think of, such as C, C++, Python, Java, and C#. Linked lists are not only efficient but also a great way to understand how pointers work.

Now, what makes linked lists so special? Well, they provide a dynamic and flexible way to store and organize data. Unlike arrays, where elements are stored in fixed positions, linked lists allow us to create a chain of nodes, each holding a piece of data and pointing to the next node in the chain.

The best part is that linked lists can grow and shrink as needed. This makes them very versatile for different situations.

And here's another exciting thing about linked lists: they serve as a fantastic stepping stone to more advanced data structures, such as graphs and trees. By mastering how to manipulate linked lists and understand how pointers connect nodes, you'll be better prepared to tackle more complex data structures in the future.

Linked List Complexity

Now, let's talk about the complexity of linked lists. Complexity helps us understand how the performance of a data structure changes based on the size of the data it holds.

For linked lists, we have two main types of complexity to consider: time complexity and space complexity.

The time complexity for searching in a linked list is O(n) in both the worst and average cases. This means that if we have 'n' elements in the list, it may take, on average, n steps to find the element we're looking for. So, searching becomes less efficient as the list grows larger.

However, when it comes to inserting and deleting elements, linked lists shine. In both the worst and average cases, the complexity is O(1). This means that no matter how big the list is, adding or removing an element takes about the same amount of time – just one step! This makes linked lists great for dynamic data structures.

	Worst Case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Regarding space complexity, it's O(n). This means that as we add more elements to the list, it will occupy more memory in a linear manner – directly proportional to the number of elements.

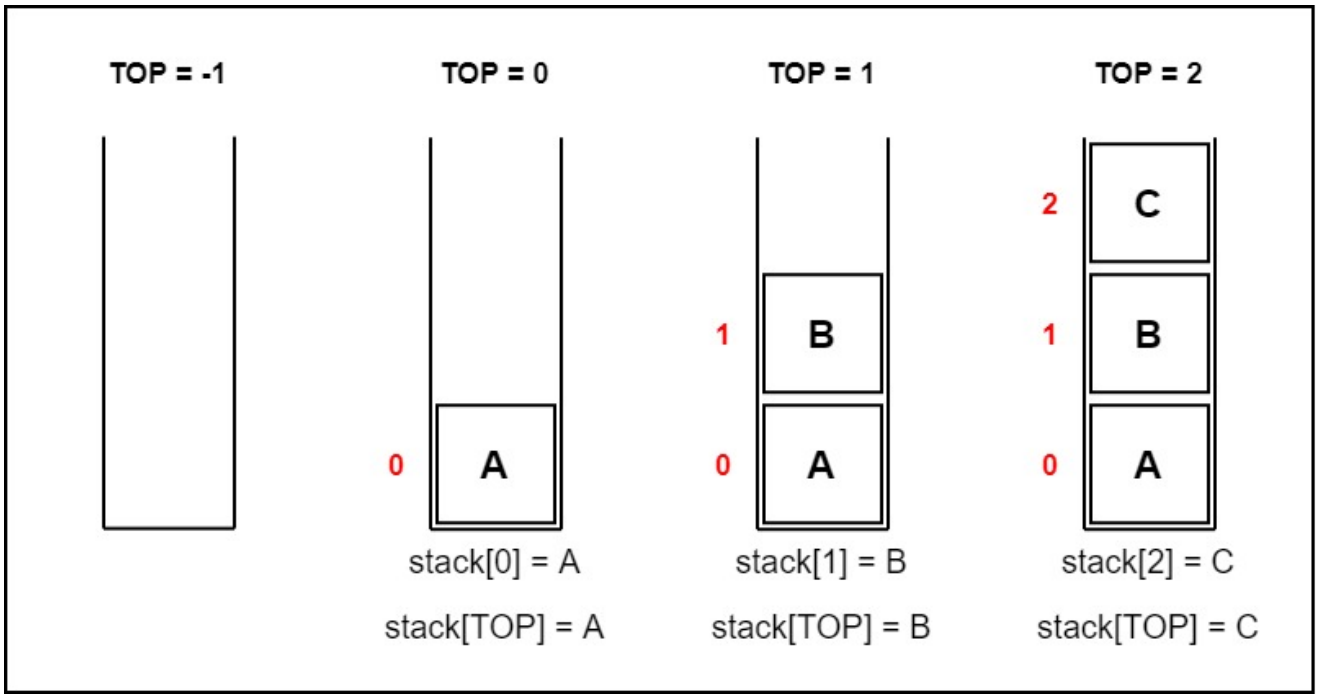
So, linked lists provide us with efficient insertion and deletion operations, but searching can take longer as the list grows. And remember, as we increase the number of elements, the memory needed will also increase in a linear fashion.

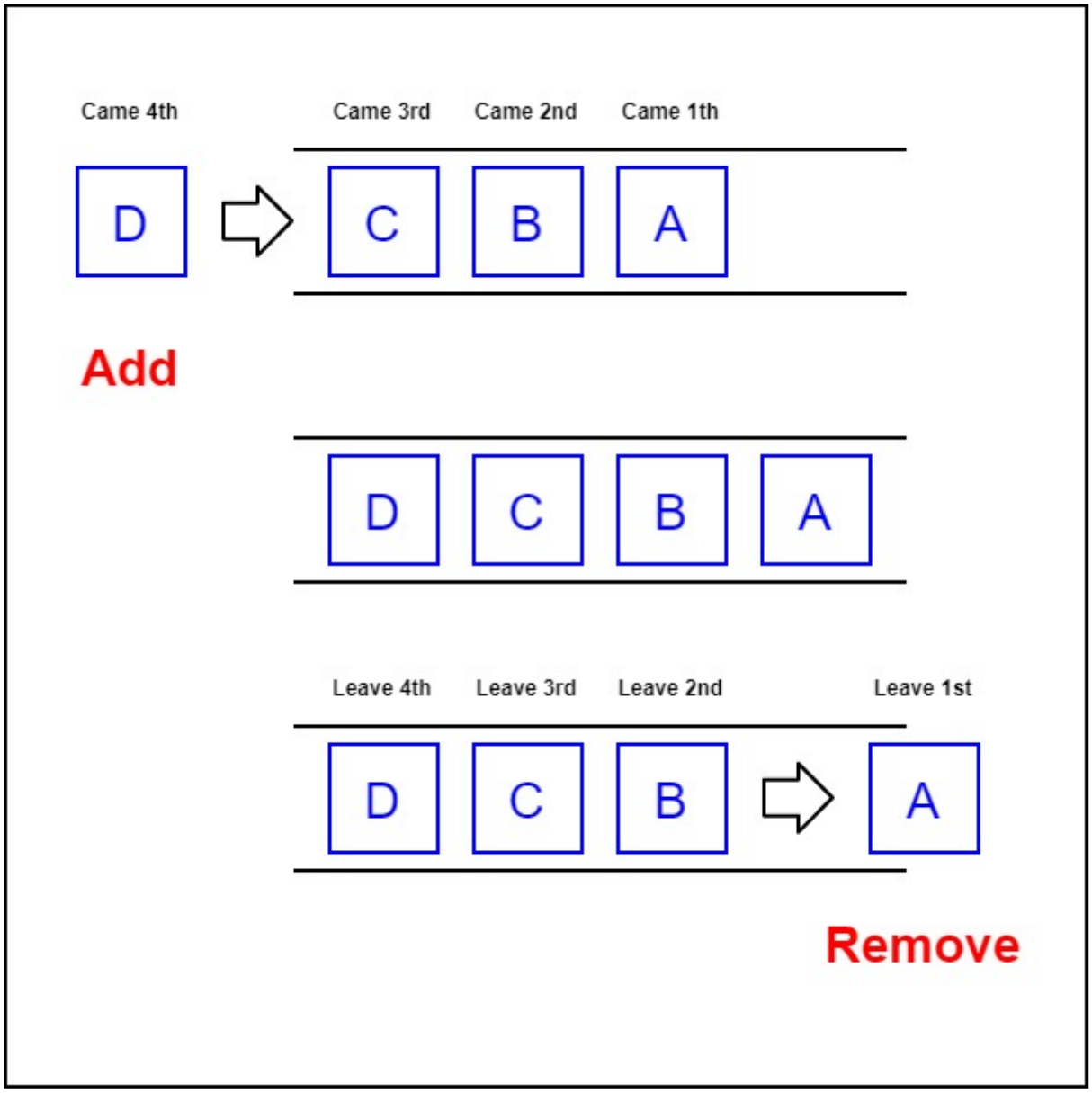
Linked List Applications

Let's now explore some interesting applications of linked lists. Linked lists find their use in various scenarios and can be quite versatile.

One of the key applications is dynamic memory allocation. Linked lists allow us to efficiently allocate memory for data during runtime, which means we can create and manage memory space as needed while the program is running.

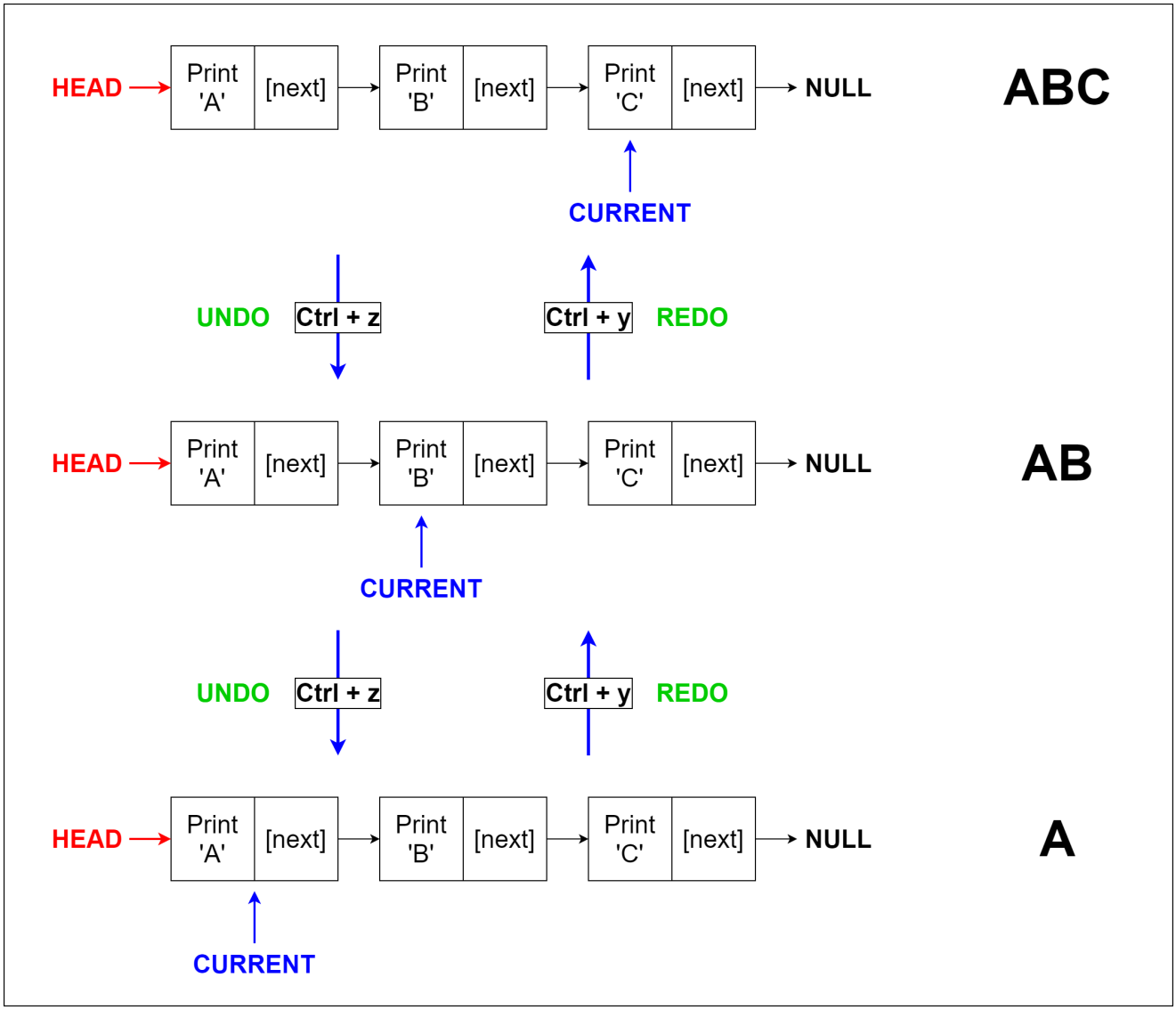
Another common application is in the implementation of stack and queue data structures. Linked lists provide an excellent foundation for building these essential tools, allowing us to perform push, pop, enqueue, and dequeue operations efficiently.



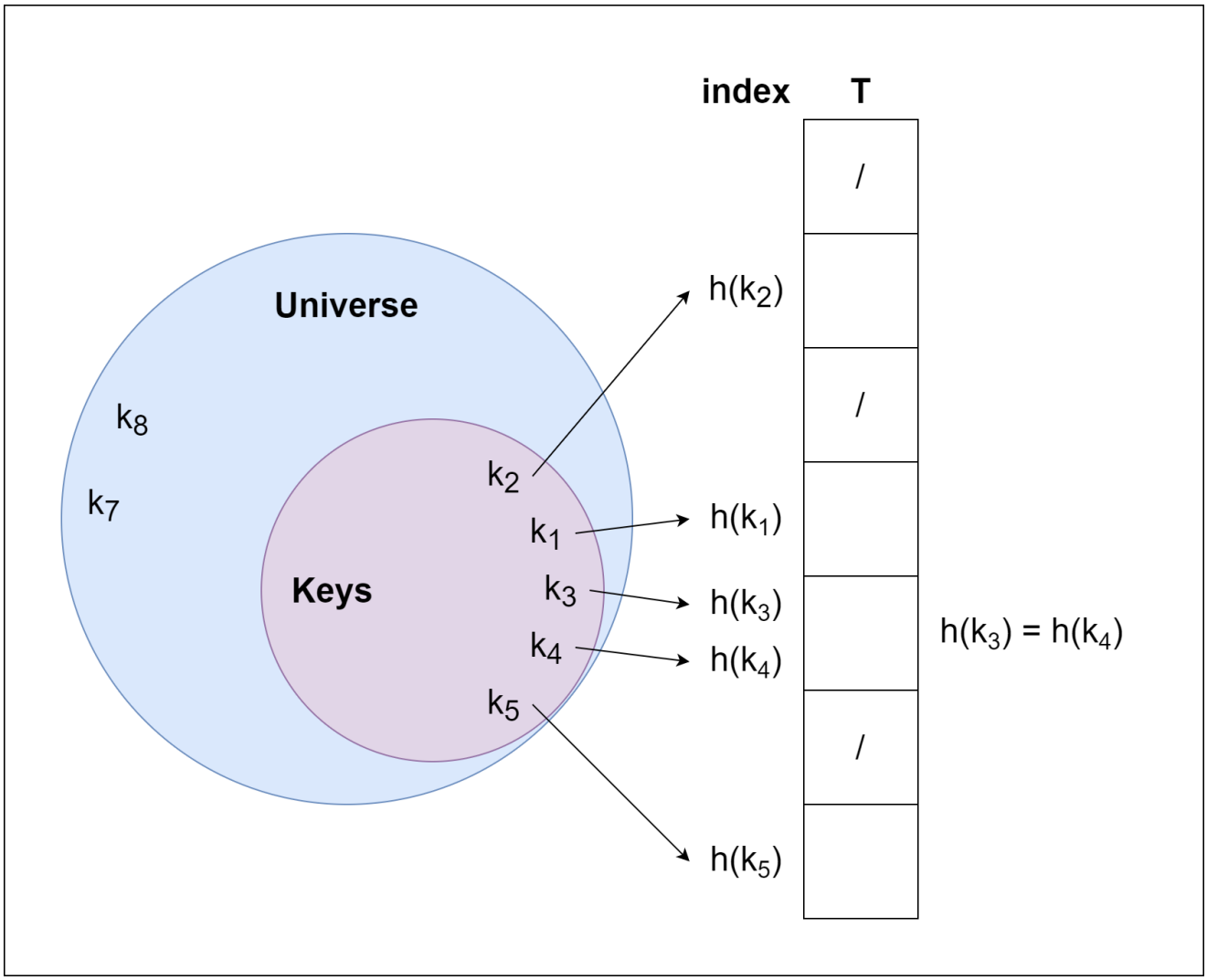


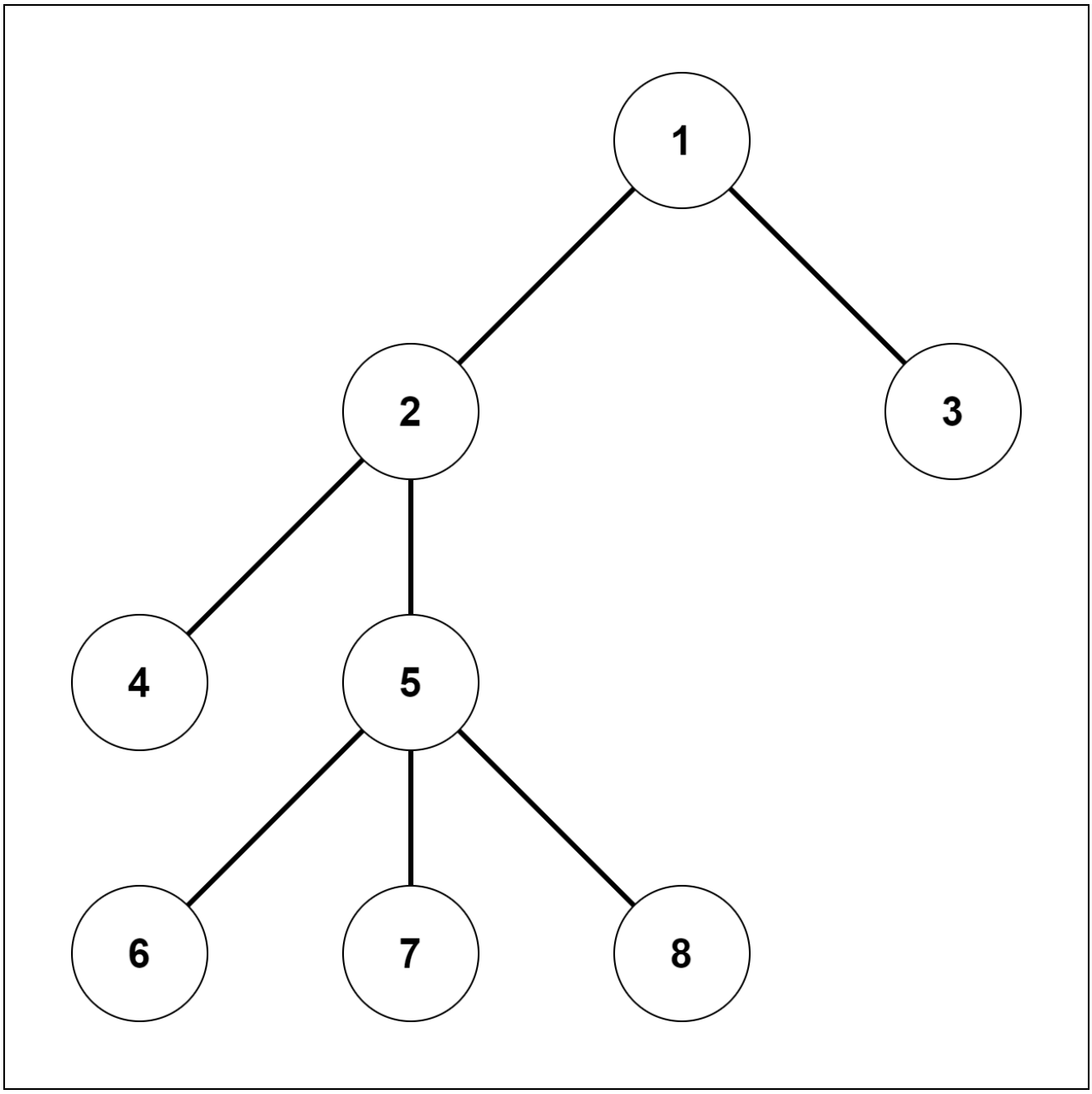
Linked lists are also used in the "undo - redo" functionality of software applications. You might have noticed that some programs allow you to undo the last action you performed. Linked lists

help track these actions, so we can reverse the changes made step by step.



Additionally, linked lists play a crucial role in hash tables and graphs. Hash tables use linked lists to handle collisions when multiple elements are mapped to the same hash value. Graphs, which represent connections between objects, often use linked lists to maintain relationships between nodes.


index
T
/
/
/
Universe
k8
k7
k2
k1
k3
k4
k5
Keys
h(k2)
h(k1)
h(k3)
h(k4)
h(k5)
h(k3) = h(k4)


1
2
3
4
5
6
7
8