INTERNATIONAL MULTI DISCIPLINARY JOURNAL FOR RESEARCH & DEVELOPMENT
eISSN 2394-6334 Impact factor: 8.854 Volume 13, issue 04 (2026)

ALGORITHMS FOR WORKING WITH LINKED LISTS: IMPLEMENTATION, ANALYSIS AND PRACTICAL APPLICATIONS

Ibragimov Ulugbek Muradilloyevich
Associate Professor, Asia International University

Abstract: Linked lists are one of the most fundamental and versatile dynamic data structures in computer science, forming the backbone of many complex algorithms and system-level implementations. This article provides a rigorous examination of singly linked lists, doubly linked lists, and circular linked lists, detailing their structural properties, algorithmic operations, time and space complexity analyses, and practical applications. The core operations — insertion, deletion, traversal, and searching — are analyzed both theoretically and through algorithmic pseudocode. The article demonstrates how linked lists serve as the underlying structure for stacks, queues, graphs, and hash tables, and examines their critical role in memory allocators, operating system kernel structures, and compiler symbol tables. A detailed comparative analysis between arrays and linked lists clarifies when each structure is optimal. Advanced topics including Floyd's cycle detection, list reversal, merging sorted lists, and the two-pointer technique are presented with full solutions to illustrate the practical depth of linked list mastery.

Keywords: Linked list, singly linked list, doubly linked list, circular linked list, node, pointer, dynamic data structure, traversal, insertion, deletion, Floyd's cycle detection, two-pointer technique, memory allocation, time complexity, algorithm analysis.

Аннотация: Связный список является одной из наиболее фундаментальных и универсальных динамических структур данных в информатике, лежащей в основе множества сложных алгоритмов и системных реализаций. В данной статье проводится строгое исследование односвязных, двусвязных и кольцевых связных списков: детально рассматриваются их структурные свойства, алгоритмические операции, анализ временной и пространственной сложности, а также практические области применения. Основные операции — вставка, удаление, обход и поиск — анализируются как теоретически, так и посредством алгоритмических псевдокодов. Показана роль связных списков в качестве базовой структуры для стеков, очередей, графов и хеш-таблиц, а также их ключевое значение в аллокаторах памяти, структурах ядра операционной системы и таблицах символов компиляторов. Представлен сравнительный анализ массивов и связных списков. Рассмотрены классические задачи: обнаружение цикла с помощью алгоритма Флойда, инвертирование списка, слияние отсортированных списков и метод двух указателей.

Ключевые слова: *Связный список, односвязный список, двусвязный список, кольцевой список, узел, указатель, динамическая структура данных, обход, вставка, удаление, алгоритм Флойда, метод двух указателей, аллокация памяти, временная сложность, анализ алгоритмов.*

In modern computer science, efficient data management lies at the heart of algorithm design and system performance. Among the most fundamental dynamic data structures is the linked list — a deceptively simple construct that powers everything from operating system kernel internals to high-level language standard libraries. Unlike arrays, linked lists manage memory dynamically, allocating and freeing nodes at runtime, which makes them indispensable in scenarios where the data size is unpredictable or where frequent insertions and deletions are required.

This article examines linked lists in depth, exploring their theoretical foundations, algorithmic properties, and real-world significance. To motivate the discussion, consider the following observations:

1) The Linux kernel represents all running processes as a doubly circular linked list (task_struct), enabling the CPU scheduler to insert and remove processes in O(1) time — a direct performance-critical application of linked list theory in production software used by billions of devices.

2) Every call to malloc() and free() in C, and every new and delete in C++, internally relies on a linked list of free memory blocks to track available heap segments — meaning that every program ever written in these languages depends on linked list algorithms at its core [1, 12].

These facts underscore that linked lists are not merely textbook abstractions — they are the invisible, load-bearing infrastructure of virtually all modern software systems.

1. Structure and Core Operations of Linked Lists

A linked list is a linear dynamic data structure in which each element, called a node, consists of two components: a data field storing the actual value, and a next pointer referencing the subsequent node in the sequence. Unlike arrays, nodes are not stored in contiguous memory locations; instead, logical order is maintained entirely through pointer linkage. The list is accessed via a head pointer that references the first node; the last node's next pointer holds a NULL value, signaling the end of the list.

This design has a profound implication: while arrays provide O(1) random access by index using arithmetic on a base address, linked lists require sequential traversal from the head to reach the i-th element, yielding O(n) access time. Conversely, once a reference to a node is held, insertion and deletion of adjacent nodes require only pointer reassignment — an O(1) operation — without the costly shifting of elements that arrays demand.

The fundamental operations of a linked list, along with their time and space complexities, are summarized in Table 1 below:

Table 1. Linked list operations and complexity analysis.

Operation	Description	Time Complexity	Space Complexity
insertFront(x)	Insert element at the head	O(1)	O(1)
insertEnd(x)	Insert element at the tail	O(n) / O(1)*	O(1)
insertAt(i, x)	Insert at position i	O(n)	O(1)
deleteFront()	Remove the head node	O(1)	O(1)

deleteAt(i)	Remove node at position i	O(n)	O(1)
search(x)	Find a value in the list	O(n)	O(1)
traverse()	Visit all nodes sequentially	O(n)	O(1)

** O(1) when a tail pointer is maintained.*

Array-based implementations offer superior cache performance due to spatial locality — elements reside in adjacent memory addresses, maximizing CPU cache utilization. Linked list nodes, by contrast, are scattered across the heap, leading to frequent cache misses during traversal. This makes arrays preferable for read-heavy workloads, while linked lists excel in write-heavy scenarios with unpredictable sizes [2, 11].

2. Types of Linked Lists and Their Structural Properties

Beyond the basic singly linked list, several important variants have been developed to address specific computational requirements. Each variant introduces structural modifications that unlock new algorithmic capabilities at the cost of increased memory or implementation complexity.

Singly Linked List: The simplest form, where each node stores data and a single next pointer. Traversal is unidirectional — from head to tail only. It is memory-efficient and serves as the natural underlying structure for stack and queue implementations. However, operations such as deletion of the previous node require traversing from the head, since no backward reference exists.

Doubly Linked List: Each node contains three fields: data, a next pointer to the successor, and a prev pointer to the predecessor. This bidirectional linkage enables O(1) deletion of any node given a direct reference, without requiring a traversal to locate the predecessor. Doubly linked lists underpin browser navigation history (forward and back buttons), text editor cursor movement, and the std::list container in C++.

Circular Linked List: The tail node's next pointer references the head instead of NULL, forming a ring structure. This eliminates the need to check for NULL during traversal and allows continuous cycling through elements — a property exploited in operating system round-robin process schedulers, where the CPU cycles through all ready processes indefinitely. Circular lists can be singly or doubly linked.

Doubly Circular Linked List: The most flexible variant, combining bidirectional linkage with a circular structure. The Linux kernel's implementation of the process list (struct list_head) uses precisely this structure, demonstrating its industrial-strength applicability. Every node can reach every other node in both directions without null-check termination.

Table 2 provides a comparative overview of the three primary variants:

Table 2. Comparative analysis of linked list variants.

Property	Singly Linked	Doubly Linked	Circular
Direction	Forward only	Bidirectional	Cyclic
Memory per node	data + next	data + prev + next	data + next
Backward traversal	Not possible	O(1) via prev	Full loop only
Tail insertion	O(n) or O(1)**	O(1) with tail ptr	O(1) with tail ptr
Typical use case	Stack, queue	Browser history, undo/redo	OS round-robin scheduling

*** O(1) when a tail pointer is explicitly maintained.*

3. Comparative Analysis: Linked Lists vs. Arrays

Although linked lists and arrays both represent linear collections of elements, they differ fundamentally in their memory layout, access semantics, and operational trade-offs. Selecting the appropriate structure is one of the most consequential decisions in algorithm and system design, with measurable impact on runtime performance.

An array stores elements in a contiguous block of memory. This layout enables constant-time random access — any element can be reached by computing its address as base + index × element_size — and maximizes CPU cache efficiency because prefetching and cache-line reads load adjacent elements automatically. The drawback is rigidity: inserting or deleting an element in the middle requires shifting all subsequent elements, incurring O(n) time, and resizing the array demands allocating a new block and copying all elements.

A linked list stores elements as independent nodes distributed across the heap. Insertion and deletion at a known position require only pointer reassignment — O(1) — and the list can grow or shrink at any point without reallocation. The cost is the absence of random access and poor cache behavior, since pointer dereferencing to traverse the list results in frequent cache misses, especially for large lists.

Table 3. Array vs. Linked List: detailed performance comparison.

Operation	Array	Linked List	Verdict
Random access	O(1)	O(n)	Array wins
Insert at head	O(n)	O(1)	List wins

Insert in middle	O(n)	O(n) + O(1)*	Equal
Delete element	O(n)	O(n) + O(1)*	Equal
Memory usage	n × sizeof(T)	n × (T + pointer)	Array wins
Resize	O(n) reallocation	O(1)	List wins
Cache performance	Excellent (locality)	Poor (scattered)	Array wins

** O(1) for the actual pointer reassignment once a reference to the target node is held.*

The practical recommendation is straightforward: choose arrays when the workload is dominated by random reads and the data size is known in advance; choose linked lists when the workload involves frequent insertions and deletions, data size is dynamic, or real-time memory behavior without reallocation is required [3, 10].

4. Algorithmic Applications and Classic Problem Solutions

The true power of linked lists becomes most apparent when applied to solving classical algorithmic problems. This section presents five representative problems that demonstrate the depth and versatility of linked list algorithms.

Cycle Detection — Floyd's Algorithm: Detecting whether a linked list contains a cycle is among the most celebrated problems in data structure theory. Floyd's Tortoise and Hare algorithm employs two pointers advancing at different speeds: the slow pointer moves one node per step, while the fast pointer moves two. If a cycle exists, the fast pointer will inevitably lap the slow pointer and they will meet within the cycle. If no cycle exists, the fast pointer reaches NULL. The algorithm runs in O(n) time and O(1) space — no auxiliary data structures required. It is applied in production systems for deadlock detection in resource allocation graphs and in linked list validation routines in standard libraries [4, 9].

List Reversal: Reversing a singly linked list in-place is achievable in O(n) time and O(1) space using three pointers: prev (initialized to NULL), current (initialized to head), and next (a temporary holder). At each step, current.next is redirected to prev, then all three pointers advance one position. When current reaches NULL, prev points to the new head. This algorithm underpins palindrome detection, certain sorting algorithms, and stack-based evaluations.

Merging Two Sorted Lists: Given two sorted linked lists, merging them into a single sorted list is a fundamental sub-routine of the Merge Sort algorithm. The procedure compares the heads of both lists at each step, appending the smaller node to the result list and advancing its source pointer. The algorithm runs in O(m + n) time, where m and n are the lengths of the two lists, and requires only O(1) additional space when performed iteratively. This problem appears as a foundational exercise on every major competitive programming platform.

Finding the Middle Node — Two-Pointer Technique: Without knowing the list's length in advance, the middle node can be located in a single pass using the slow-fast pointer technique:

when the fast pointer reaches the end of the list (NULL or the last node), the slow pointer is exactly at the middle. This O(n) single-pass approach is used as a preprocessing step in Merge Sort on linked lists and in determining the center of a list for palindrome checking [5, 8].

Finding the k-th Node from the End: Using two pointers separated by a fixed gap of k nodes, the k-th node from the tail can be found in a single O(n) traversal. The lead pointer is advanced k steps first; then both pointers advance together until the lead reaches NULL. This technique has direct applications in sliding-window network buffer management and in log file monitoring systems, where the last k entries must be retained without knowing the total entry count [6, 7].

5. Real-World Applications in Computing Systems

The practical significance of linked lists extends far beyond textbook algorithms into the foundations of modern software and hardware systems.

Operating System Kernels: The Linux kernel uses a doubly circular linked list (struct list_head, defined in <linux/list.h>) as its universal list primitive. The process scheduler maintains all runnable tasks in such a list, enabling O(1) insertion and removal of processes. The same structure is used for file descriptor tables, socket wait queues, module lists, and interrupt handler registrations — demonstrating that a single well-designed linked list abstraction can serve as the backbone of an entire operating system.

Dynamic Memory Allocation: The C standard library implementations of malloc() and free() (such as Doug Lea's dlmalloc and glibc's ptmalloc) maintain free memory blocks as a linked list. When memory is requested, the allocator traverses the free list to find a sufficiently large block; when memory is freed, the block is reinserted into the list with potential coalescing of adjacent free blocks. This mechanism operates transparently beneath every heap allocation in C and C++ programs, making linked list traversal one of the most frequently executed operations in any running system.

Compilers and Symbol Tables: Compilers use hash tables with chaining to resolve hash collisions in symbol tables — the data structure that stores variable names, function signatures, and type information during compilation. Each hash bucket is a linked list of symbol entries that share the same hash value. This chaining approach maintains O(1) average-case lookup while handling arbitrarily many collisions, and is the implementation strategy used in GCC, Clang, and the JVM.

Undo/Redo Mechanisms: Text editors, graphic design tools (Adobe Photoshop, GIMP), and version control systems implement undo/redo functionality using a doubly linked list of command objects. Each user action is a node; the current position pointer moves backward through prev links on undo and forward through next links on redo. Branching undo histories (as in Vim's undo tree) extend this to a doubly linked tree. This is a direct and elegant application of bidirectional linked list traversal in end-user software.

Artificial Intelligence and Pathfinding: Graph search algorithms at the core of AI planning — including A* and bidirectional BFS — maintain their open and closed sets as dynamically updated lists. In game development, the active object list (all currently alive game entities) is typically a linked list, since entities are frequently spawned and destroyed during gameplay. Insertion and deletion of entities in O(1) time, given a direct node reference, makes linked lists significantly more efficient than arrays for this use case.

Conclusion

Linked lists, despite their conceptual simplicity, represent indispensable tools in the arsenal of algorithmic thinking and systems design. This article has established several key findings: first, the dynamic memory management capability of linked lists confers a decisive advantage over arrays in scenarios involving unpredictable data size or frequent structural modification — the O(1) insertion and deletion at a known position is a guarantee that no array can match without costly reallocation; second, the three primary variants — singly, doubly, and circular linked lists — each offer a distinct set of capabilities suited to different problem domains, and the choice among them requires deliberate analysis of the access and modification patterns of the target application; third, advanced algorithmic techniques such as Floyd's cycle detection, the two-pointer method, and in-place reversal demonstrate that sophisticated problems can be solved on linked lists with optimal O(n) time and O(1) space, requiring no auxiliary data structures beyond the list itself; fourth, linked lists are not merely theoretical constructs — they are actively employed in the Linux kernel process scheduler, in C and C++ memory allocators, in compiler symbol table implementations, and in virtually every undo/redo system in interactive software. The practical recommendation for engineers and system designers is to first characterize the access pattern of the workload: if random access dominates, prefer arrays; if structural modification dominates and size is dynamic, prefer linked lists — and choose the specific variant based on whether bidirectional or circular traversal is required. Mastery of linked list algorithms and their trade-offs forms an essential cornerstone of efficient software engineering.

References:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.

3. Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.

4. Skiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer.

5. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Python. Wiley.

6. Shaffer, C. A. (2013). Data Structures and Algorithm Analysis (3rd ed.). Dover Publications.

7. Floyd, R. W. (1967). Nondeterministic algorithms. Journal of the ACM, 14(4), 636–644. DOI: 10.1145/321420.321422.

8. Weiss, M. A. (2012). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

9. McConnell, J. J. (2007). Analysis of Algorithms: An Active Learning Approach (2nd ed.). Jones & Bartlett Publishers.

10. U.M. Ibragimov, B. Ergashev. Important aspects of collecting Windows operating system data for the pentest process. Conference: "The role of digital technologies in the economy and education.". Uzbekistan (Samarqand). 2024. p. 30–33.

11. U.M. Ibragimov. Effectiveness and efficiency of the PROMETHEUS system. XVI Saginovsky Readings. Integration of Education, Science and Production. Kazakhstan (Karaganda). 2024. p. 237–239.

12. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Addison-Wesley.