Advanced Linked List Concepts

Data Structures & Algorithms

Class 04

Topic: Sentinel Nodes, Generalized Lists & Skip Lists

Learning Objectives

By the end of this class, you will be able to:

1. Understand the concept and benefits of sentinel nodes
2. Implement sentinel nodes to simplify boundary conditions
3. Understand generalized linked lists and their applications
4. Recognize when to use heterogeneous data structures
5. Understand skip lists as a probabilistic alternative to balanced trees
6. Analyze the time complexity of skip list operations
7. Choose between different advanced linked list variations

Recap: The Complete Journey

Class 01: Introduction to Linked Lists

- Basic nodes and pointers
- Why linked lists exist

Class 02: ADT and Basic Types

- Abstract Data Types (ADT) concept
- Singly and Doubly Linked Lists
- Visual operation walkthroughs

Class 03: Circular and Header Lists

- Circular Linked Lists - endless loops
- Header Nodes - special sentinel nodes

Today: Advanced concepts that power real-world systems!

Sentinel Nodes: Guards at the Boundaries

A Story to Begin

Imagine a train system where every route **must** have a starting station and an ending station, even if they're just empty platforms.

These "dummy stations" serve no passengers but make the system predictable:

- Train operators don't need to check "Does this route have a starting point?"
- Maintenance crews know where to begin and end inspections
- The system **never has a route with NULL endpoints**

This is the philosophy behind **Sentinel Nodes** - special dummy nodes that guard the boundaries and eliminate special cases!

What is a Sentinel Node?

A **Sentinel Node** (or **Dummy Node**) is a special node that:

- **Always exists** in the data structure
- **Doesn't contain meaningful data** (or contains special marker values)
- **Simplifies algorithms** by eliminating boundary checks
- **Serves as a guard** at list boundaries

Key Principle: "There is always at least one node" eliminates NULL checks!

Sentinel vs Header Node

Similar, but with subtle differences:

Aspect	Header Node	Sentinel Node
Purpose	Mark beginning of list	Guard boundaries (beginning AND/or end)
Count	Usually just one (at head)	Can be multiple (head, tail, both)
Data	May store metadata (count, max, etc.)	Typically just dummy/marker values
Philosophy	"Organization and metadata"	"Eliminate special cases"

In practice: All header nodes are sentinel nodes, but not all sentinel nodes are header nodes!

Types of Sentinel Node Implementations

Type 1: Single Sentinel at Head

Structure:

```
SENTINEL → [10|•] → [20|•] → [30|NULL]
    ↑          ↑
  Dummy   Real data starts here
```

C Implementation:

```c
struct Node* createList() {
    struct Node* sentinel = malloc(sizeof(struct Node));
    sentinel->data = -1;      // Marker value
    sentinel->next = NULL;    // Empty list
    return sentinel;
}
```

Benefit: No need to check if the list is empty before operations, as the sentinel node ensures the head pointer is never `NULL`, allowing the same logic to handle both the first and subsequent nodes.

Type 2: Dual Sentinels (Head and Tail)

Structure:

```
HEAD_SENTINEL ↔ [10] ↔ [20] ↔ [30] ↔ TAIL_SENTINEL
      ↑          Real data           ↑
   Dummy start                   Dummy end
```

Key Features:

- HEAD sentinel's `next` points to first real node (or TAIL if empty)
- TAIL sentinel's `prev` points to last real node (or HEAD if empty)
- Real data is always sandwiched between sentinels

Dual Sentinels - Empty List

Visual of Empty List:

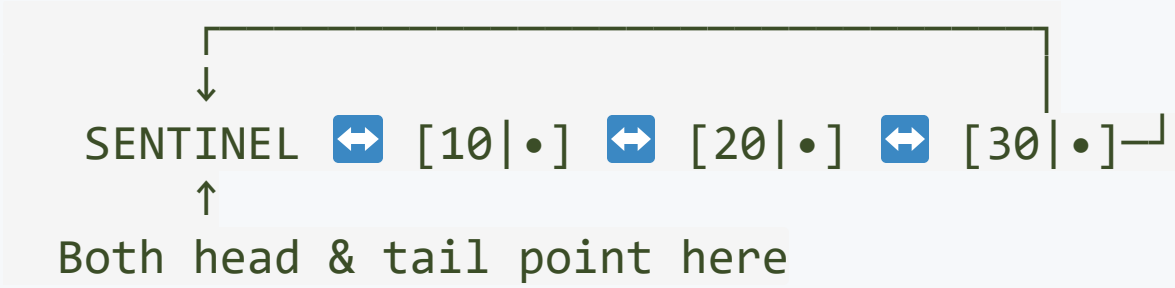
```
HEAD_SENTINEL ↔️ TAIL_SENTINEL
      ↑                ↑
   prev=NULL       next=NULL
```

Implementation:

```
struct List* createList() {
    struct List* list = malloc(sizeof(struct List));

    list->head_sentinel = malloc(sizeof(struct Node));
    list->tail_sentinel = malloc(sizeof(struct Node));

    // Connect them
    list->head_sentinel->next = list->tail_sentinel;
    list->tail_sentinel->prev = list->head_sentinel;

    list->head_sentinel->prev = NULL;
    list->tail_sentinel->next = NULL;
```

Type 3: Circular Sentinel

Structure:



Empty list: sentinel points to itself

```
struct Node* createCircularList() {
    struct Node* sentinel = malloc(sizeof(struct Node));
    sentinel->data = -1;
    sentinel->next = sentinel; // Points to itself
    sentinel->prev = sentinel;
    return sentinel;
}
```

Benefits of Sentinel Nodes

Benefit 1: Eliminates NULL Checks

Without Sentinel:

```
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);

    if (*head == NULL) {                // Special case!
        *head = newNode;
        return;
    }

    struct Node* current = *head;
    while (current->next != NULL) {    // NULL check
        current = current->next;
    }
    current->next = newNode;
}
```

Benefit 1: With Dual Sentinels

With Sentinel:

```
void insertAtEnd(struct List* list, int value) {
    struct Node* newNode = createNode(value);

    // Always insert before tail sentinel - no special cases!
    newNode->prev = list->tail_sentinel->prev;
    newNode->next = list->tail_sentinel;

    list->tail_sentinel->prev->next = newNode;
    list->tail_sentinel->prev = newNode;

    // Works even if list is empty!
}
```

Why it works: Empty or not, tail_sentinel always has a valid prev pointer!

Benefit 2: Uniform Deletion Logic

Without Sentinel:

```
void deleteNode(struct Node** head, struct Node* target) {
    if (*head == target) {              // Special case for head
        *head = target->next;
        free(target);
        return;
    }

    // Find previous node (O(n) operation!)
    struct Node* current = *head;
    while (current->next != target) {
        current = current->next;
    }
    current->next = target->next;
    free(target);
}
```

Benefit 2: With Dual Sentinels

With Sentinel:

```
void deleteNode(struct Node* target) {
    // Works for ANY node - no special cases!
    target->prev->next = target->next;
    target->next->prev = target->prev;
    free(target);
}
```

Benefit: Every node (including first and last) has a predecessor and successor (the sentinels)!

Benefit 3: Simplifies Iteration

Without Sentinel:

```
void printList(struct Node* head) {
    if (head == NULL) {                 // Empty check
        printf("Empty\n");
        return;
    }

    struct Node* current = head;
    while (current != NULL) {           // NULL check
        printf("%d ", current->data);
        current = current->next;
    }
}
```

Benefit 3: With Dual Sentinels

With Sentinel:

```
void printList(struct List* list) {
    struct Node* current = list->head_sentinel->next;

    // Stop when we hit tail sentinel
    while (current != list->tail_sentinel) {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

No NULL checks, no empty list special case!

Real-World Applications

Application 1: Linux Kernel Linked Lists

The Linux kernel uses circular doubly-linked list with sentinel pattern:

```
struct list_head {
    struct list_head *next, *prev;
};

// Every list has a sentinel "head"
struct list_head my_list;

// Initialize: points to itself
INIT_LIST_HEAD(&my_list);
```

Why?

- Uniform insertion/deletion (no special cases)
- Lockless safety: Sentinels provide stable memory addresses for atomic operations
- Generic: works with any data type

Application 2: Database B-Tree Implementations

A B-tree implementation is a self-balancing search tree data structure that maintains sorted data and allows for efficient operations by permitting nodes to have multiple keys and children.

Many B-tree implementations use sentinel keys to ensure that every search key always falls between two existing boundaries, simplifying the logic for node splitting and merging.

```
          [SENTINEL]
         /           \
   [Real Node]  [Real Node]
```

Benefits:

- Tree is never truly empty
- No NULL parent checks
- Rotation operations are uniform

Application 3: Java's LinkedList Class

Java's `LinkedList` uses dual sentinels internally:

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
}

private transient Node<E> first;  // Head sentinel
private transient Node<E> last;   // Tail sentinel
```

Benefits:

- Thread-safe operations simplified
- Iterator implementation cleaner
- No edge case bugs

Generalized Linked Lists

What is a Generalized Linked List?

A **Generalized Linked List** is a linked list where **nodes can contain different types of data**, including:

- Atomic values (integers, strings, etc.)
- **Other linked lists** (sublists or nested lists)
- Mixed types

Think of it as: A linked list that can represent hierarchical or nested structures!

The Need for Generalization

Problem with Simple Lists:

```
Simple list can only represent flat sequences:
[1] → [2] → [3] → [4] → NULL

But what if you need to represent:
- A book with chapters (each chapter has sections)
- An organization chart (departments with sub-teams)
- Polynomial: (x² + 2x)(y + 3)
```

Solution: Generalized Lists!

Structure of Generalized Lists

Each node can be one of two types:

Type 1: Atomic Node

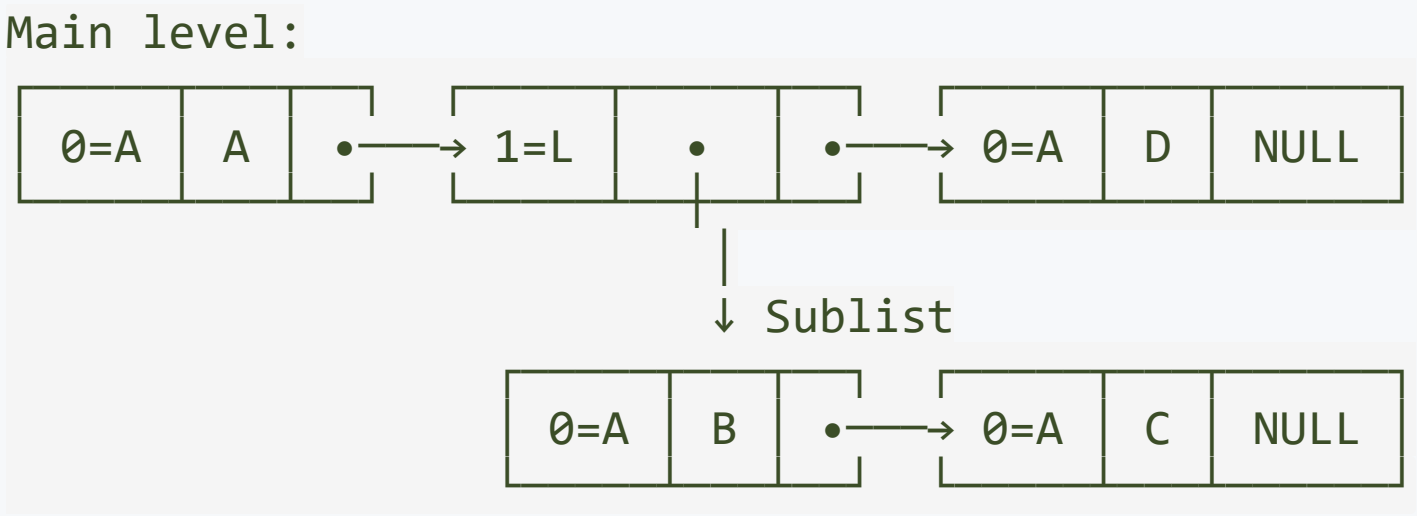

```
0 = atom
```

Type 2: List Node


```
1 = list    ↓
        Points to sublist
```

Visual Example: Nested Lists

Representing: `A → (B → C) → D`



Explanation:

- First node: Atomic, contains 'A'
- Second node: List type, points DOWN to sublist (B → C)
- Third node: Atomic, contains 'D'

C Implementation

```c
struct GLNode {
    int flag;  // 0 = atom, 1 = sublist

    union {
        int atom;              // If flag = 0
        struct GLNode* down;   // If flag = 1
    } data;

    struct GLNode* next;  // Next at same level
};

// Create atomic node
struct GLNode* createAtom(int value) {
    struct GLNode* node = malloc(sizeof(struct GLNode));
    node->flag = 0;
    node->data.atom = value;
    node->next = NULL;
    return node;
}
```

Using `union` saves memory - each node uses space for atom OR pointer, not both!

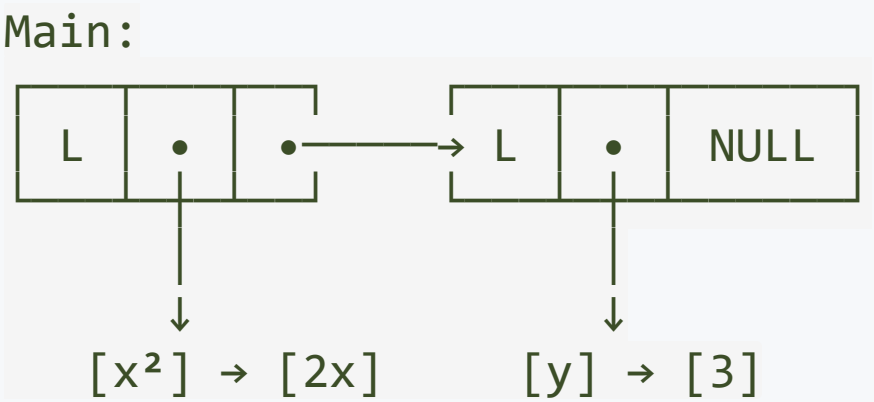
Traversing Generalized Lists

```
void printGeneralizedList(struct GLNode* head, int level) {
    struct GLNode* current = head;

    while (current != NULL) {
        // Print indentation
        for (int i = 0; i < level; i++) printf("  ");

        if (current->flag == 0) {
            // Atomic node
            printf("Atom: %d\n", current->data.atom);
        } else {
            // List node - go deeper!
            printf("List:\n");
            printGeneralizedList(current->data.down, level + 1);
        }

        current = current->next;
    }
}
```

Application: Polynomial Representation

Representing: `(x² + 2x)(y + 3)`

Product of two polynomials - perfect for generalized lists!



Benefits:

- Easy to multiply polynomials
- Natural representation of nested structure
- Can handle arbitrary complexity

Application: JSON/XML Representation

JSON:

```
{
  "name": "Alice",
  "scores": [95, 87, 92],
  "address": {
    "city": "NYC"
  }
}
```

As Generalized List:

```
Root List:
  → name (atom: "Alice")
  → scores (list: 95 → 87 → 92)
  → address (list: city (atom: "NYC"))
```

Use Cases: Configuration parsers, AST in compilers, DOM in browsers