**A Combined Use of Linked List and Binary Tree**

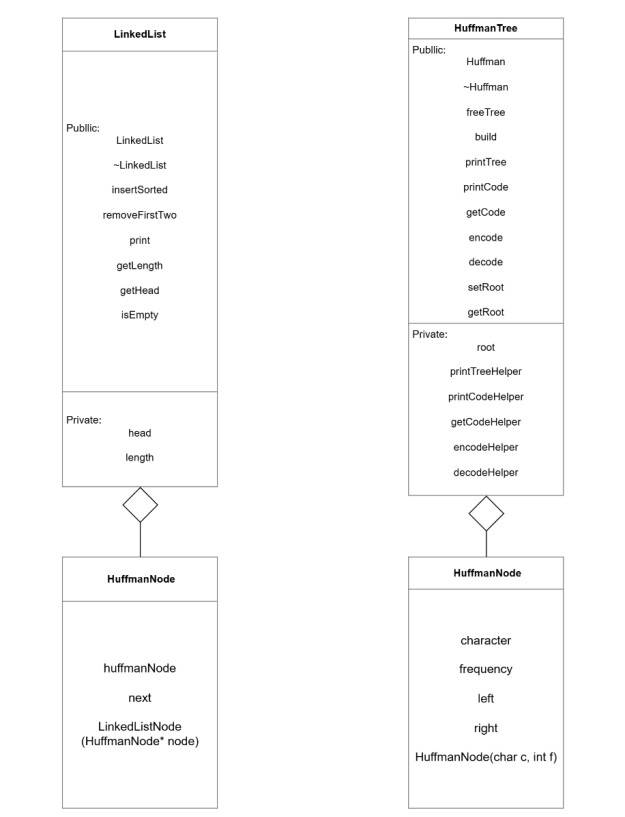
Group Number: Team Honor 3

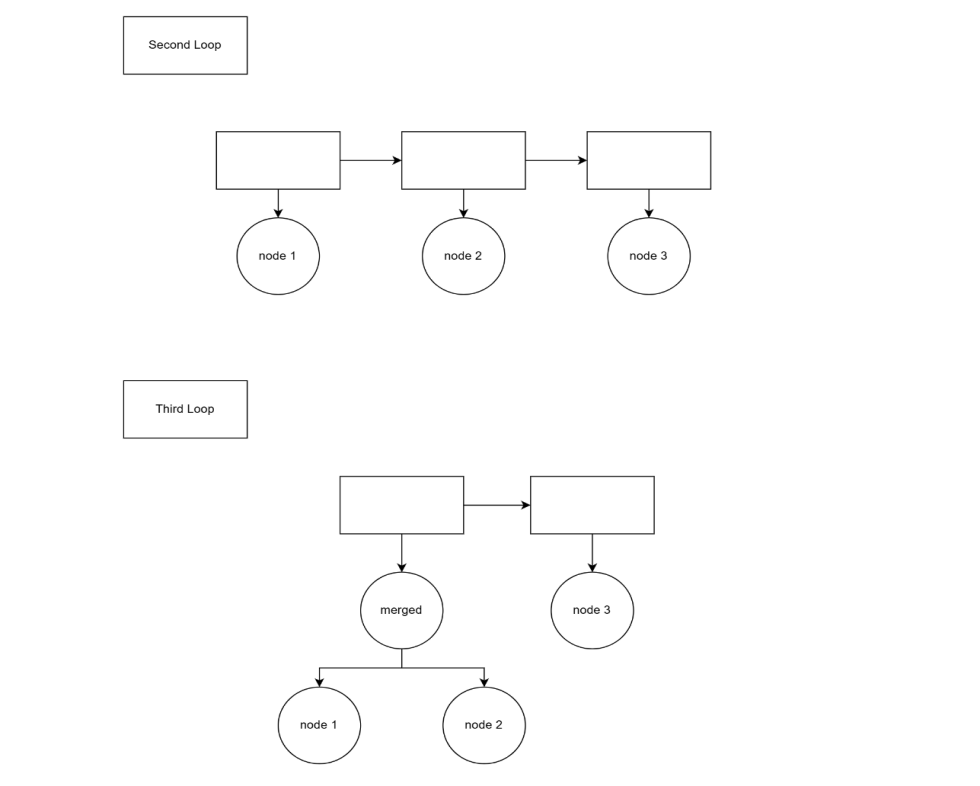
Team Members: Yankui Lee, Hengli Jia

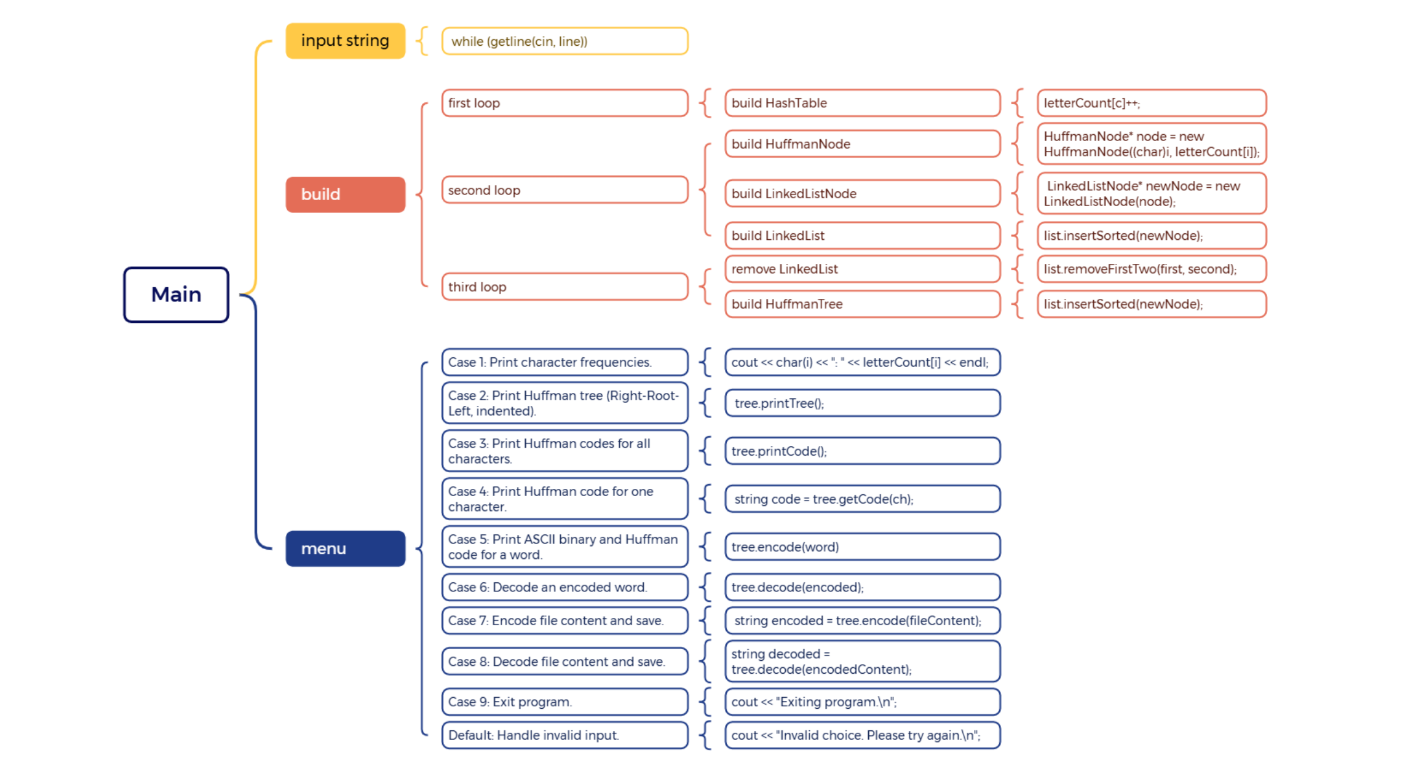
**Introduction**

Our project combines two data structures—linked lists and binary trees—to construct a Huffman tree. This tree structure is used to efficiently compress text by following a rule that minimizes storage space.

We first use the hash table concept to determine the frequency of each character. After we got each character and its frequency, we treated it as a node and built a binary search tree. The leaf nodes are the only nodes that contain characters. We use the two least frequent nodes to create a parent node that contains two nodes, but without a character inside. By continuing to do it, we can get a root node that only has the frequency, which is the length of the string. After completing the tree, we start from the root and go through the tree(0 for left, 1 for right) until we reach the leaf node and get the character. The higher the frequency, the easier the approach.

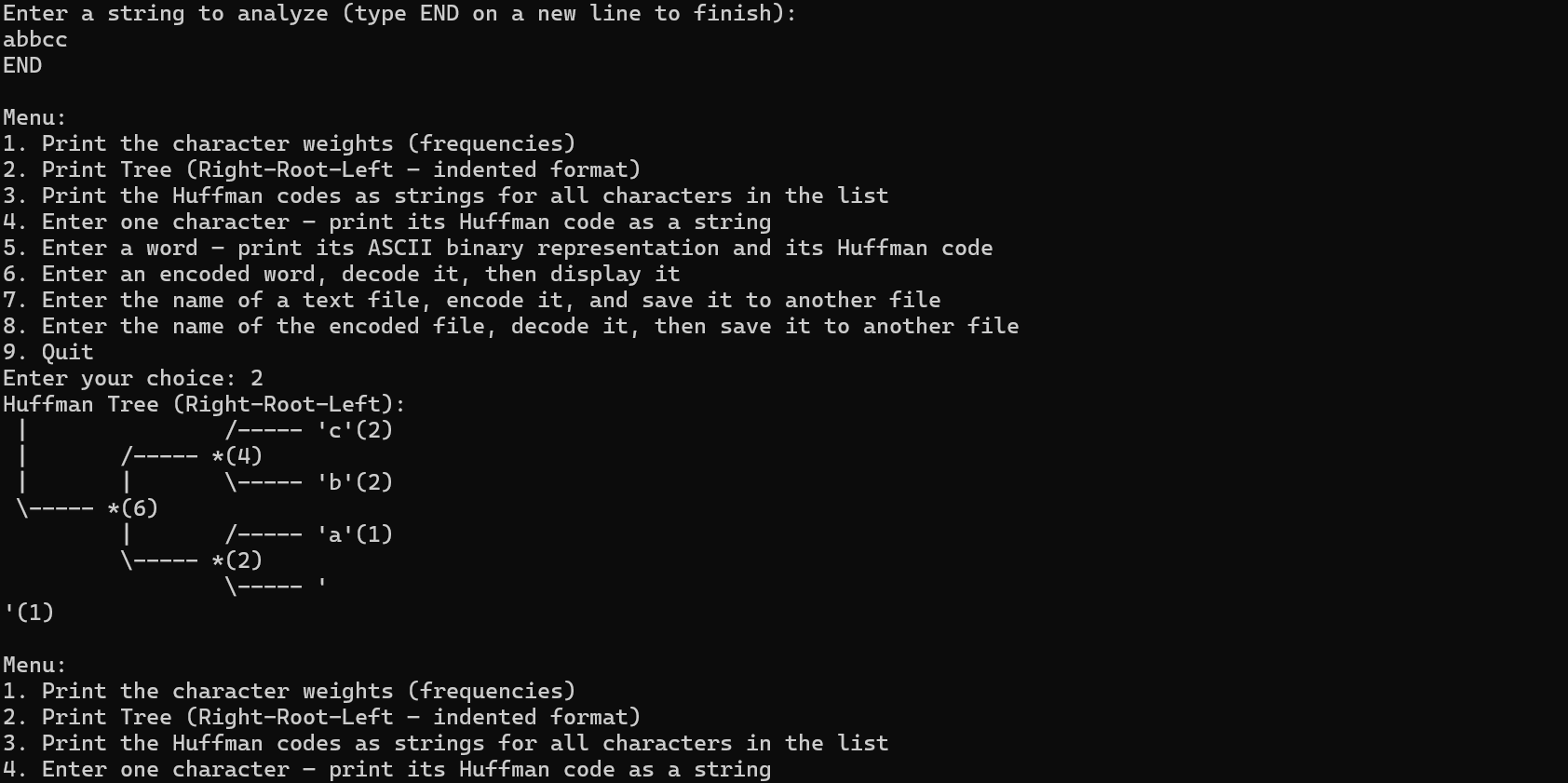
**Data Structures, Class Diagram, and Structure Chart**





**Sample Output Screen**

**Sample Screen Output**



**Major Debug Problems**

**First Problem**

The initial challenge we faced was not fully understanding how to structure the linked list. We rushed into implementation without a clear plan, resulting in a disorganized structure. This lack of preparation forced us to rebuild parts of the project, which consumed a significant amount of time.

Lesson Learned:  
To prevent similar issues in future projects, we should begin with a structure chart, a data structure diagram, and a class diagram. These visual tools help clarify the organization and implementation strategies for large projects, ensuring a smoother development process.

**Second Problem**

The second major issue involved a recursive dependency between Huffman.h and LinkedList.h. Specifically, Huffman.h required the LinkedList structure, while LinkedList.h also depended on elements from Huffman.h. This circular dependency resulted in compiler errors because C++ does not allow circular inclusions.

While forward declarations offer a temporary fix, the better long-term solution is to rethink the design. Instead of both headers relying on each other, we should redesign the structures to establish a clear containment relationship. For instance, one structure can contain the other, removing the need for mutual dependencies. This not only resolves the circular inclusion issue but also simplifies the design of the main function.

**Third Problem**

The third issue we encountered was the lack of a helper function to support our recursive method. Initially, we embedded all logic into a single recursive function, which made the parameters unclear and the function harder to understand and maintain. This also made it difficult to test and debug individual components of the recursion.

Lesson Learned:  
 In future projects, we should create a separate helper function for recursive algorithms. This helper function can handle the internal logic and parameter management, while the main function provides a cleaner interface. Doing so enhances code readability, separates concerns, and allows for easier debugging and extension.

**Application**

**File Compression**  
Huffman coding is widely used in file compression tools such as ZIP and GZIP to reduce file size by encoding frequently occurring characters with shorter bit strings.

**Multimedia Codecs**  
Formats like JPEG and MP3 use Huffman encoding to compress image and audio data, saving storage and transmission bandwidth.

**Fax Transmission (Group 3 Fax Standard)**  
Huffman encoding is used in fax machines to compress black and white image data for more efficient transmission.

**PNG Image Format**  
PNG images use a combination of LZ77 and Huffman coding to efficiently store pixel data with lossless compression.

**Assignment**

**Yankui Lee:**

Data Structure Design

main(), menu() and other related functions

Linked List algorithms

Integration, testing, and project documentation

**Hengli Jia:**

Data Structure Design

Binary Tree algorithms

Hash Table algorithms

Integration, testing, and project documentation

**Conclusions**

Our project successfully combined linked lists and binary trees to implement a Huffman tree for efficient text compression. By first using a hash table to calculate character frequencies, we created nodes representing these characters and organized them into a binary tree structure. The construction of the tree followed Huffman’s algorithm: repeatedly combining the two least frequent nodes into a new parent node until a single root node remained. Traversing the tree from root to leaf (using 0 for left and 1 for right) allowed us to assign binary codes to each character, with more frequent characters receiving shorter codes—thus optimizing compression.

Through this process, we faced and overcame two significant challenges. First, a lack of planning in our initial linked list structure led to a confusing and inefficient implementation. From this, we learned the importance of starting with visual planning tools such as structure charts, data structure diagrams, and class diagrams to guide development and avoid rework.

Second, we encountered a circular dependency between Huffman.h and LinkedList.h, causing compiler errors. This highlighted the importance of thoughtful design and modular code. While forward declarations can serve as a quick workaround, we learned that a cleaner solution is to establish a one-way containment relationship between structures to eliminate mutual dependencies and simplify the overall system architecture.

Ultimately, this project not only helped us understand how to implement Huffman coding using fundamental data structures, but also reinforced best practices in software design and project planning. These lessons will be valuable in future coding endeavors, especially when tackling complex, multi-module systems.