Objectives :

- Data structures
- Working with Files
- Bitmap and Wave Files

Data Structures :

When solving problems, we can observe that many problems share common structures and concepts. We can create data types that help us represent these structures effectively.
These special data types are the tools we use to implement data structures

Stacks and Queues :

Stacks :

The stack is one of the data structures that we can use to solve various problems. The fundamental principle of a stack is that it stores data in an array according to the LIFO (Last-In, First-Out) principle. This means that the last value added to the stack is the first one to be retrieved when reading data from the stack.
An example of this is a gun's charging system: the last bullet you insert into the magazine will be the first bullet fired.

Example :

```
#include <iostream>
#include <stack>
using namespace std;

#define MAX_BULLETS 20
typedef unsigned bullet;

void charge(stack<bullet> *m);
void fire(stack<bullet> *m);

int main(){
        stack<bullet> magazine;
        charge(&magazine);
        charge(&magazine);
        fire(&magazine);
        fire(&magazine);
        return 0;
}
void charge(stack<bullet> *m){
    if(m->size() > MAX_BULLETS){
        cout << "Magazine Full!\n";
        return ;
    }
    m->push(m->size() + 1);
    cout << "Charged bullet " << m->size() << endl;
    return ;
}

void fire(stack<bullet> *m){
    if (m->size() <= 0){
        cout << "Magazine Empty!\n";
        return ;
    }
    cout << "Fired  bullet " << m->top() << endl;
    m->pop();
    return ;
}
```

Queues :

Queues operate on a different principle than stacks. While stacks follow the LIFO (Last-In, First-Out) principle, queues adhere to the FIFO (First-In, First-Out) principle, meaning the first value added to the queue is the first value to be removed from it.
An example of this is a job application system where the first 'n' applicants who meet the requirements are typically offered the job

Example :

```
#include <iostream>
#include <queue>
using namespace std;

#define MAX_WORKERS 20
```

```
typedef unsigned Worker;

void apply(queue<Worker> *app);
void hire(queue<Worker> *app);

int main(){
        queue<Worker> application;
        apply(&application);
        apply(&application);
        hire(&application);
        hire(&application);
        return 0;
}
void apply(queue<Worker> *app){
    if(app->size() >= 20){
        cout << "Applications are full!\n";
        return ;
    }
    app->push(app->size() + 1);
    cout << "Person " << app->back() << " Applied to the job!\n";
    return ;
}

void hire(queue<Worker> *app){
    if (app->size() <= 0){
        cout << "Applications Empty !\n";
        return ;
    }
    cout << "Hired Worker " << app->front() << endl;
    app->pop();
    return ;
}
```

Linked List and Binary Search Trees :

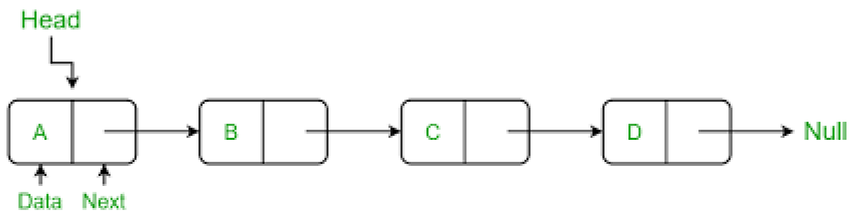
Linked List :

Linked Lists are another important data structure and one of the most frequently used ones. We've encountered dynamic arrays, which store data in contiguous blocks of memory. However, what if we don't have sufficient contiguous storage, or if the available storage is scattered in separate blocks? This is where the key power of linked lists lies.

A linked list is a collection of nodes, where each node contains two essential parts:

1. **Data:** The actual value stored within the node.
2. **Pointer (or Link):** A reference to the next node in the sequence.

This structure allows for flexible memory allocation, as nodes can be scattered throughout memory, and the list is maintained by following the chain of pointers from one node to the next



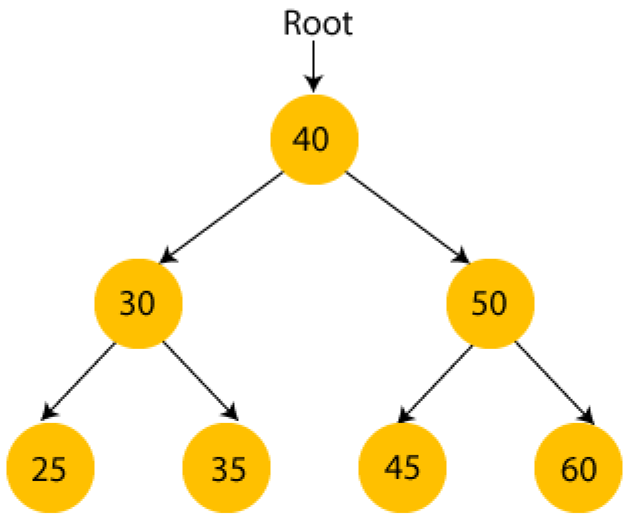
Example :

```
#include <iostream>
using namespace std;

using Node = struct Node{
    int data;
    struct Node* next;
};

void free_elements(Node* n);

int main(){
        Node Head;
        Node *cursor;
        cursor = &Head;
        int n;
        cout << "Enter Number of Elements : ";
        cin >> n;
        for (int i = 0; i < n; i++){
            cursor->next = new Node;
                cout << "Enter element " << i + 1 << " : ";
                cin >> cursor->data;
                cursor = cursor->next;
        }
        delete cursor->next;
        cursor->next = NULL;
        cursor = &Head;
        for (int i = 0; i < n; i++){
                cout << "Data in element " << i + 1 << " is : " << cursor->data <<
endl;
                cursor = cursor->next;
        }
        free_elements(&Head);
        return 0;

}
void free_elements(Node* n){
    if(n->next == NULL){
        delete n;
        return;
    }
    return free_elements(n->next);
    delete n;
}
```

Binary Search Trees :

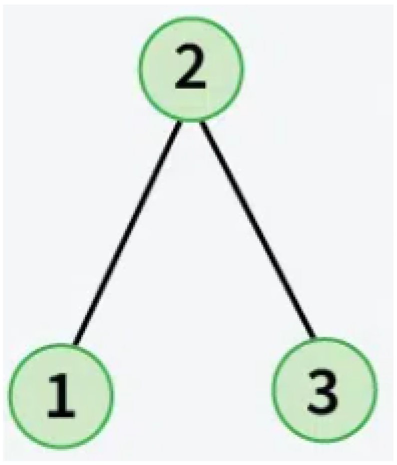
Binary Search Trees (BSTs) are a data structure used to store sorted data elements , the binary search trees is stuctured as folowing

- Each element in the tree is represented as a node.
- Each node contains:
  - A value
  - A left pointer: Points to the node containing a smaller value.
  - A right pointer: Points to the node containing a larger value.
  The head of the Binary Search Trees called root node



Example :

We will make this Binray Search Tree



```
#include <iostream>
using namespace std;



using node = struct node{
    int number;
    struct node *left;
    struct node *right;
};

void free_tree(node *root);
void print_tree(node *root);

int main(){
    node *tree = NULL;
    node *n = new node;

    n->number = 2;
    n->left = NULL;
    n->right = NULL;
    tree = n;

    n = new node;
    n->number = 1;
    n->left = NULL;
    n->right = NULL;
    tree->left = n;

    n = new node;
    n->number = 3;
    n->left = NULL;
    n->right = NULL;
    tree->right = n;

    print_tree(tree);
    free_tree(tree);
    return 0;
}

void free_tree(node *root){
    if (root == NULL){
        return;
    }
    free_tree(root->left);
    free_tree(root->right);
    delete root;
}

void print_tree(node *root){
    if (root == NULL){
        return;
    }
    print_tree(root->left);
    cout << root->number <<endl;
    print_tree(root->right);
}
```

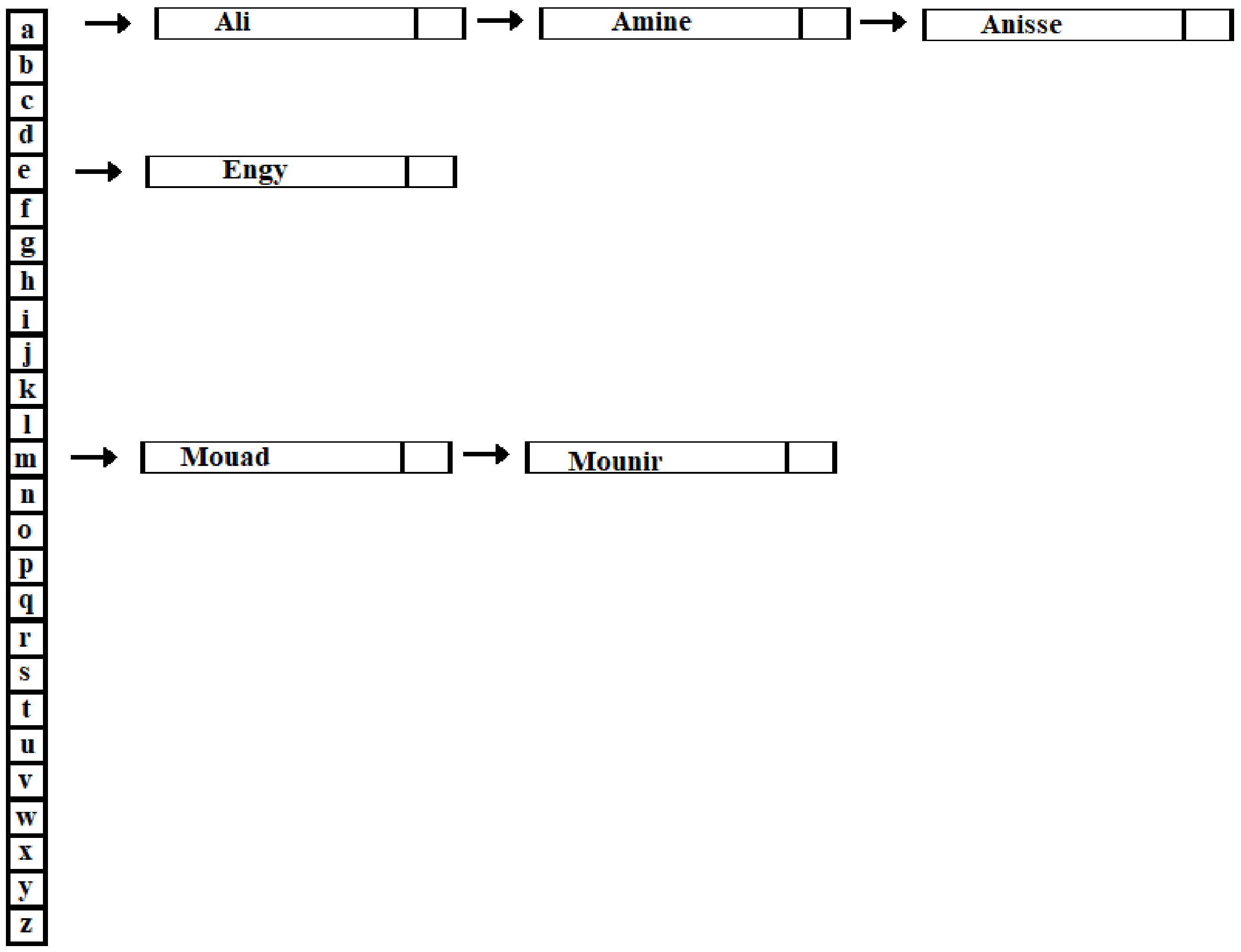
Dictionaries, Hash Tables and Tries :

Dictionary :

Dictionaries are data structures that store data in a key-value pair format, much like a real-world dictionary. Each key within a dictionary must be unique, and is associated with a corresponding value. Searching for data in a dictionary is incredibly fast, as you can directly access the value by simply providing its corresponding key. However, storing a large number of unique keys can consume significant memory space. Hashing is a common technique used to efficiently store and retrieve keys in a dictionary by mapping them to specific memory locations, thus improving the performance and memory efficiency of dictionary implementations

Hashing and Hash Tables :

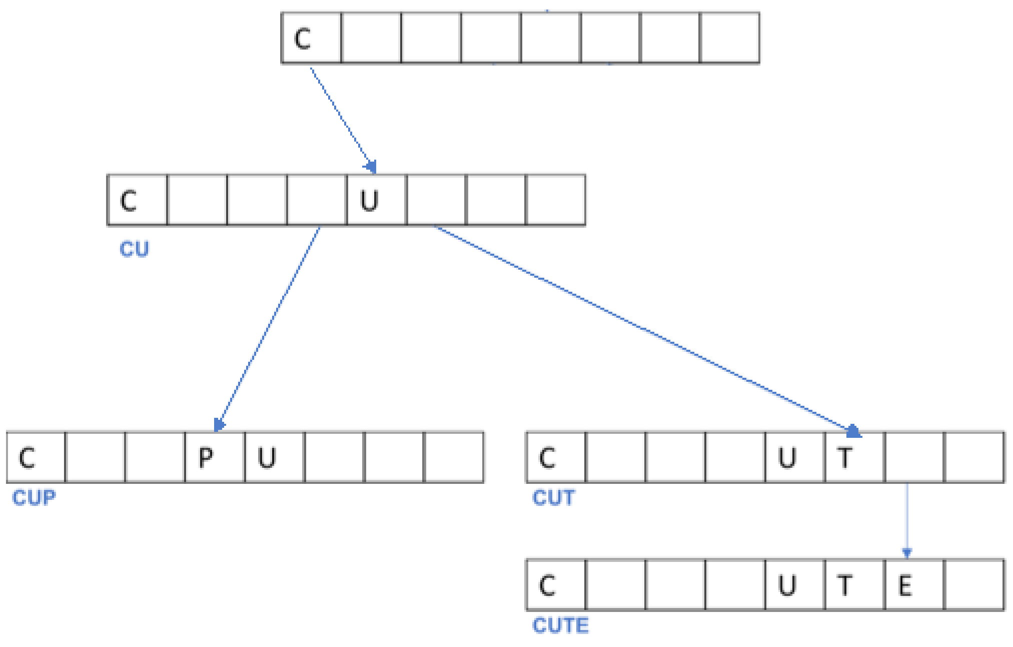
Hashing is a technique that transforms an input value into a shorter, fixed-size representation called a hash code. Hash tables utilize this concept by using an array of hash values, where each value can point to a node. Each node typically stores a value and a pointer to the next node (if there are collisions).
One of the most significant challenges in using hash tables lies in designing an effective hash function. A poorly designed hash function, which maps many different input values to the same hash code, can lead to frequent collisions. This results in longer linked lists at those specific array indices, significantly slowing down the search process. Conversely, an overly complex hash function might produce a large number of unique hash codes, leading to many empty slots in the hash table, which wastes memory space.
To illustrate this, consider using a hash table to store student names. If we simply use the first letter of each name as the hash code, we would likely encounter many collisions. For example, names starting with 'A' or 'J' might have a large number of entries, while other letters might have very few or none



we can fix this by using first and second letters as hash this will make searching for element faster but we will end up having 24*24 hash index instead of only 24 and many of them arn't used , that mean we will take more memory then we will use, and here it come to us, we can use simple hash function tat generated small indexes this will result less memory use and more time searching for value or we can use complexe hash map that will result less time in searching for value but more memory use.

Tries :

Tries are another form of data structure designed for efficient string storage and retrieval. One of their key advantages is that they allow searches to be performed in constant time, making them ideal for tasks like autocomplete or prefix matching. However, a downside of Tries is that they tend to consume a large amount of memory, as each node can have multiple children representing different characters, leading to significant memory usage, especially when storing many strings with few shared prefixes.



Time and Memory complexity :

When solving problems, it's crucial to consider not only the correctness of the solution but also its efficiency. Time and memory complexity are essential concepts for analyzing an algorithm's performance.

Example 1:

Calculating the sum of the first 'n' numbers.

- A simple **for loop** iterates through each number, resulting in O(n) time complexity.
- Utilizing the formula `sum = n * (n + 1) / 2` allows for constant-time calculation O(1) as it directly computes the sum without iteration.
  Solution 1:

```
#include <iostream>
using namespace std;

int main(){
        float n;
        float sum;
        cout << "Enter number : ";
        cin >> n;
        sum = n * (n + 1) / 2;
        cout << "Sum is " << sum << endl;
        return 0;
}
```

Solution 2:

```
#include <iostream>
using namespace std;

int main(){
        int n, sum = 0;
        cout << "Enter number : ";
        cin >> n;
        for(int i = 0; i <= n; i++){
                sum += i;
        }
        cout << "Sum is " << sum << endl;
        return 0;
}
```

Example 2:

Search algorithms:

- **Linear search** examines each element sequentially, resulting in a time complexity of O(n), where 'n' is the number of elements.
- **Binary search**, applicable to sorted lists, repeatedly divides the search space in half, achieving a significantly faster time complexity of O(log n).

Working with Files :

Introductions :

Computers store data in memory while they are running programs and performing calculations. However, this data is temporary. When a program finishes executing, all the variables and results stored in memory are lost.
To prevent this data loss, we can store it in files. This allows us to retain the information even after the program has completed its execution.

Writing and Appending to Text Files :

When working with files in C++, the first step is to include the `<fstream>` header, which stands for "file stream." This header provides the necessary classes and functions for file handling. Next, you need to choose the mode in which you want to open the file. The three primary classes for file operations are:

- **`ifstream`**: Used for reading from files.
- **`ofstream`**: Used for writing to files.
- **`fstream`**: Used for both reading and writing.
  After selecting the appropriate class, you need to create an object (variable) that will represent the file. This object is initialized by passing the file's path or name as an argument to the constructor of the chosen class.
  `ofstream file("output.txt"); // Opens 'example.txt' for writing`
  To write data to the file, you can use the insertion operator (`<<`). For instance:
  `file << "This is a new line of text." << endl;`
  Once you have finished working with the file, it is important to close it using the `.close()` method. This ensures that all data is properly saved and system resources are released:
  `file.close();`

Example :

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
        ofstream file("output.txt");
        file << "Hello my name is Ali" << endl;
        file << "nice to meet you All" << endl;
        file.close();
        return 0;
}
```

When writing to a file using `ofstream`, the default behavior is to overwrite the existing content of the file, starting from the beginning. This means that any previous data in the file is erased. To avoid this and **append new data to the end of the file** instead, you can use the **`ios::app`** mode (short for "append").
To enable append mode, pass `ios::app` as the second argument when creating the `ofstream` object.
`ofstream file("output.txt", ios::app);`
With this mode, any new data you write to the file will be added to the end, preserving the

existing content. For instance:
`file << "This line will be appended to the file." << endl;`
By using `ios::app`, you ensure that the file's original data remains intact while adding new information.

Example :

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
        ofstream file("output.txt", ios::app);
        file << "More data to the file" << endl;
        file << "Now bye :D " << endl;
        file.close();
        return 0;
}
```

Reading Files :

Reading files in C++ follows a process similar to writing them. First, include the `<fstream>` header to access the necessary file-handling classes. Next, create an `ifstream` object to open the file in reading mode. Pass the file name or path as an argument to the constructor of the `ifstream` object.
`ifstream file("input.txt");`
To read data from the file, you can use the `getline()` function. This function reads a line from the file each time it is called. You can use a `while` loop to repeatedly call `getline()` until the end of the file is reached. The loop ensures that you read all the data in the file and stop when there is nothing left to read.

```
string line;
while (getline(file, line)) {
    cout << line << endl;
}
```

Example :

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main(){
    string line;
    ifstream  file ("output.txt");
    while(getline(file,line)){
        cout << line << endl;
    }
    file.close();
    return 0;
}
```

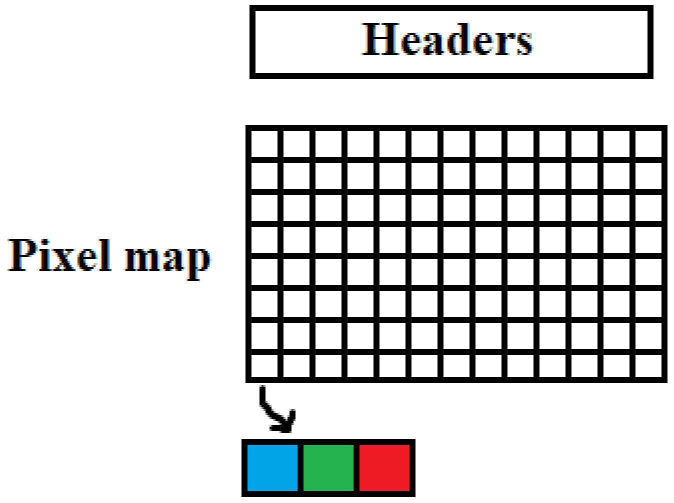
Bitmap and Wav Files :

Bitmap Files :

Bitmaps are a common file format for storing images. They leverage the principle that any color can be created by combining varying amounts of red, green, and blue light (RGB).

A bitmap image represents an image as a grid of pixels. Each pixel is typically represented by 24 bits, with 8 bits each for the intensity of red, green, and blue.
Before the actual pixel data, a bitmap file contains header information. This metadata is structured into two parts:

1. **BITMAPFILEHEADER:** This 14-byte header provides general information about the file, such as the file type (identified by the "BM" signature) and the offset to the pixel data within the file.
2. **BITMAPINFOHEADER:** This 40-byte header contains crucial information about the image itself, including its width, height, color depth, and compression method.
   Following these headers, the pixel data is stored as an array of bytes. Each pixel is represented by three consecutive bytes in the order Blue-Green-Red (BGR).



Example :

In this example, we will see how to grayscale an image. To achieve this, we need to iterate through each pixel in the image and calculate the average of its red, green, and blue color components.

```
#include <iostream>
#include <fstream>
using namespace std;

using BYTE = uint8_t;
using DWORD = uint32_t;
using LONG = int32_t;
using WORD = uint16_t;

#pragma pack(push, 1)
using BITMAPFILEHEADER =  struct {
WORD bfType;
DWORD bfSize;
WORD bfReserved1;
WORD bfReserved2;
DWORD bfOffBits; };
#pragma pack(pop)



#pragma pack(push, 1)
using  BITMAPINFOHEADER = struct {
DWORD biSize;
LONG biWidth;
LONG biHeight;
WORD biPlanes;
WORD biBitCount;
DWORD biCompression;
DWORD biSizeImage;
LONG biXPelsPerMeter;
LONG biYPelsPerMeter;
DWORD biClrUsed;
DWORD biClrImportant;  };
#pragma pack(pop)


#pragma pack(push, 1)
using pixel = struct {
BYTE B;
BYTE G;
BYTE R;
};
#pragma pack(pop)
int main(int argc, char *argv[]){
    BITMAPINFOHEADER binfo;
    BITMAPFILEHEADER bheader;
    pixel p;
    LONG h = 0, w = 0;
    ifstream input(argv[1], ios::binary);
    ofstream output(argv[2], ios::binary);
    input.read((char*)&bheader, sizeof(BITMAPFILEHEADER));
    output.write((char*)&bheader, sizeof(BITMAPFILEHEADER));
    input.read((char*)&binfo, sizeof(BITMAPINFOHEADER));
    output.write((char*)&binfo, sizeof(BITMAPINFOHEADER));
    w = abs(binfo.biWidth);
    h = abs(binfo.biHeight);
    WORD gray;
    for(LONG i = 0; i < w; i++){
        for(LONG j = 0; j < h; j++){
            input.read((char*)&p, sizeof(pixel));
            gray = (p.G + p.B + p.R) / 3;
            p.G = gray;
            p.R = gray;
            p.B = gray;
```

```
                output.write((char*)&p, sizeof(pixel));

            }
        }

        input.close();
        output.close();
        return 0;
}
```

Remarques :

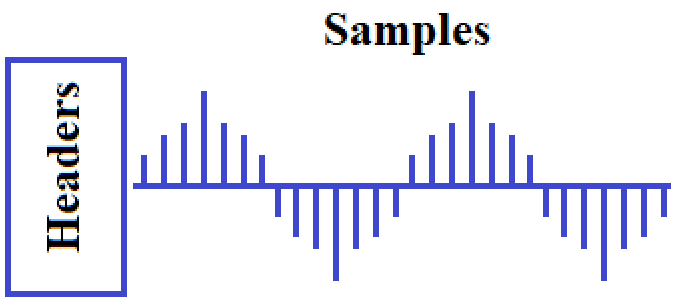
- We use `#pragma pack(push, 1)` and `#pragma pack(pop)` to instruct the compiler to minimize the amount of padding between members of a struct. This ensures that the struct is packed tightly in memory, which is useful for binary file operations or when working with hardware.
- To indicate that we are working with a binary file in C++, we pass `std::ios::binary` as the second argument when opening the file. This ensures that the data is read or written in raw binary format without any text transformations.
- To read from or write to a binary file, we use the `write()` and `read()` functions. Both functions take two arguments:
    1. The address of the data to be written (for `write()`) or the address where the read data will be stored (for `read()`).
    2. The size (in bytes) of the data being written or read.
- The `write()` and `read()` functions require a `char*` pointer as the first argument. To pass the address of our data, we cast it to a `char*` using by writing `(char*)&data`.

Wav Files :

Another common way to store data is the WAV file format. WAV files are used to store audio data. They begin with a 44-byte "header" that contains metadata about the file, such as the file size, the sampling rate (number of samples per second), and the bit depth (size of each sample, typically 16 bits or 2 bytes).
After the header, the WAV file contains a sequence of audio samples. Each sample is a single 2-byte (16-bit) integer that represents the amplitude of the audio signal at a specific point in time.
Scaling the value of each sample by a given factor directly affects the volume of the audio.

- Multiplying each sample value by 2.0 will double the amplitude, effectively increasing the volume.

- Multiplying each sample value by 0.5 will halve the amplitude, effectively decreasing the volume



Example :

To change the volume of a WAV file, we first need to obtain an integer value representing the factor by which to multiply each sample.
Next, we need to:

- Read the file header from the input WAV file and copy them to the output WAV file.
- Read the audio samples from the input WAV file.
- Multiply each sample by the volume factor.
- Write the modified samples to the output WAV file

```
#include <iostream>
#include <fstream>
using namespace std;

using BYTE = uint8_t;
using WORD = uint16_t;
using DWORD = uint32_t;

#pragma pack(push, 1)
using WAVHEADER = struct {
    BYTE   chunkID[4];
    DWORD  chunkSize;
    BYTE   format[4];
    BYTE   subchunk1ID[4];
    DWORD  subchunk1Size;
    WORD   audioFormat;
    WORD   numChannels;
    DWORD  sampleRate;
    DWORD  byteRate;
    WORD   blockAlign;
    WORD   bitsPerSample;
    BYTE   subchunk2ID[4];
    DWORD  subchunk2Size;
};
#pragma pack(pop)
int check_format(WAVHEADER header);
int get_block_size(WAVHEADER header);

int main(int argc, char *argv[]){
    int size = 0;
    int r = 0, f;
    WAVHEADER header;
    ifstream input(argv[1], ios::binary);
    if (!input.is_open()){
        cout << "Couldn't open file\n";
        return 1;
    }
    input.read((char*) &header, sizeof(WAVHEADER));
    if (!check_format(header)){
        cout << "Input is not a WAV file.\n";
        input.close();
        return 1;
    }
    ofstream output(argv[2], ios::binary);
    output.write((char*) &header, sizeof(WAVHEADER));
    cout << "Enter factor: ";
    cin >> f;
    size = get_block_size(header);
    uint8_t *buffer = new uint8_t[sizeof(uint8_t) * size];
    int seek = size;
    do{
        input.read((char*)buffer, size);

        for (int i = 0; i < size; i++){
        *(buffer + i) = *(buffer+i) * f;
        }
        output.write((char*) buffer, size);
        if (input.tellg() >= header.subchunk2Size + 44){
            break;
        }
    }while (1);
    delete[] buffer;
    input.close();
    output.close();
    return 0;
}
int check_format(WAVHEADER header){
    if (header.format[0] == 'W' && header.format[1] == 'A' && header.format[2] == 'V' && header.format[3] == 'E'){
        return 1;
    }
    return 0;
}

int get_block_size(WAVHEADER header){

    return header.bitsPerSample * header.numChannels / 8;
}
```

Tasks :

Task 1:

Rewrite the example of queue and stack by replacing fixed size array with linked list.

- **For the first example**: Each time we call the function `charge` element (bullet) get appended to the end of the linked list. When the function `fire` is called, the last element of the linked list is removed.
- **For the second example**: Each time someone applies for a job, they are added to the end of the linked list. When we assign a worker, the pointer moves to the second element of the list, making it the new head, and the first element is removed.

Task 2:

Write a program that converts a bitmap image into ASCII art:

- First, convert the image to grayscale to simplify it into shades of gray.
- Next, write characters to the `output.txt` file based on the gray values as follows:
  - ` ` **225 -> 255**.
  - `.` **200 -> 225**.

- `:` **175 -> 200**.
- `-` **150 -> 175**.
- `=` **125 -> 150**.
- `+` **100 -> 125**.
- `*` **75 -> 100**.
- `#` **50 -> 75**.
- `%` **25 -> 50**.
- `@` **0 -> 25**.

Hint :

Keep in mind that a bitmap saves pixel data in reversed order. The first row we see when we open the image is actually the last row in the file. Similarly, the top-left pixel in the image corresponds to the bottom-left pixel in the file.