1 Introduction

Data structure allows us to perform specific query or update in a more efficient way, along with some restriction on the retrieval / query / update. Therefore, it is of vital importance for us to choose the compatible data structure wisely for different problem.

Without specified mentioned, assume `arr` is declared as a signed integer array. The following code is added to the beginning of all C++ programs.

```cpp
#include <bits/stdc++.h>
using namespace std;
```

2 Address and Pointers

Obviously, a variable stores a value with its data type declared. Actually, variable are stored somewhere in the CPU, with its unique **address (地址)**. We can perform some unique operation with address.

Reference variable is a "reference" to an existing variable. It can be created with the "&" operator. The change of the reference variable will affect the value stored in the original variable being referenced.

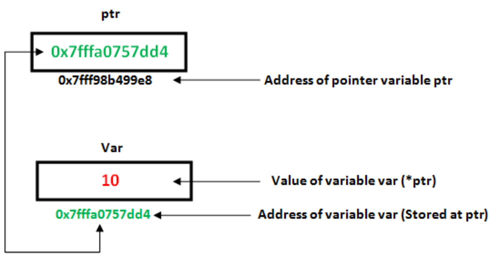
```
string school = "BSTC";
string &location = school;
location = "CTSB";
cout << school << endl; // output: CTSB
cout << location << endl; // output: CTSB
```

To retrieve the **address** of a variable, the "&" operator can be facilitated. For the code segment below, the output indicates the address of the variable, presented in the form of a hexadecimal value. It might **not be fixed** during each execution of the code segment.

```
string school = "BSTC";
cout << &school << endl; // output example: 0x7ffe4aa930a0
```

Pointers store address like a normal variable does. A normal variable stores some value, either mutable or immutable. A pointer variable stores an adress of a variable. You can refer to the visualization of pointers to the figure in the right.

The following showcase the declaration of a pointer.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

```
string school = "BSTC";
string* ptr = &school;
cout << ptr << endl; // output example: 0x7ffe4aa930a0
```

With a pointer variable, we can retrieve the value of variable stored with the address stored in the pointer variable, which is known as **dereferencing** the pointer. Deferencing can be done by the "*" operator.

```
string school = "BSTC";
string* ptr = &school;
cout << *ptr << endl; // output: BSTC
```

3 Structures and Unions

Structures (also called **structs**) are a way to group several related variables into a single place. Each variable in the structure is known as the **member** of the structure. It is *extremely useful* in complicated problem. Unlike an array, a structure can contain multiple data types. A structure can be declared as follows:

```
struct Node {
    int num;
    string s;
} myNode;
Node myAnotherNode;
myAnotherNode.num = 5;
myAnotherNode.s = "BSTC";
cout << myAnotherNode.num << endl; // output: 5
cout << myAnotherNode.s[2] << endl; // output: T
```

The structure above contains two members, an integer and a string. The name of the struct is `myNode`. The struct is named as a data type called `Node`. Declaration using a predefined structure as a data type is allowed.

To access the member of a structure, please refer to the code segment above.

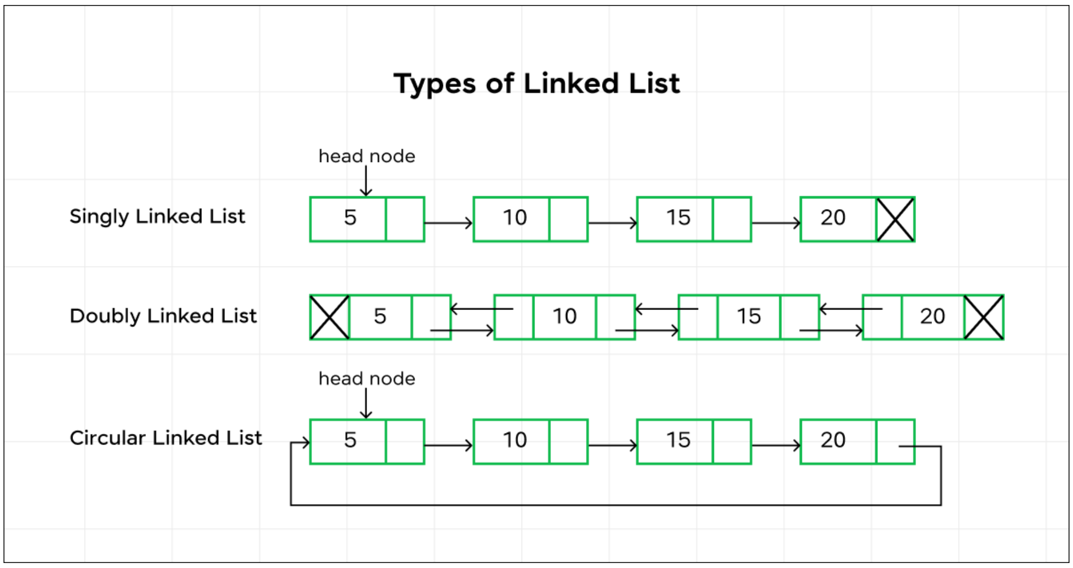
Pay heed to the special syntax of accessing the member of a structure under dereferencing.

```
struct Node {
    int num;
};
Node myNode; myNode.num = 5;
Node* ptr = &myNode;
cout << *ptr.num << endl; // ERROR!
cout << (*ptr).num << endl; // OK, output: 5
cout << ptr->num << endl; // OK, output: 5
```

Union is a special structre in C++. The class specifier for a union declaration is similar to the declaration of a normal structure. The syntax of union is basically the same as struct, by only replacing the keyword “`struct`” to “`union`”. However, one thing that makes it different from structures is that the member variables in a union share the same memory location, unlike a structure that allocates memory separately for each member variable. The size of the union is equal to the size of the largest data type. Memory space can be used by one member variable at one point in time, which means if we assign value to one member variable, it will automatically deallocate the other member variable stored in the memory which will lead to loss of data.

4 Linked list

A linked list is a linear data structure that allows us to store data in **non-contiguous** memory locations. A linked list is defined as a collection of nodes where each node consists of two members which represents its value and a next pointer which stores the address for the next node.



When a node contains a pointer storing the address of the next node, the list is called a **singly linked list**. When a node contains two pointers storing the address of the previous node and the next node respectively, the list is called a **doubly linked list**. When the list is found to be circular, the list is called a **circular linked list**.

Linked list has a high efficiency (constant time complexity) to insert or delete a node. However, it has an average lower efficiency (linear time complexity in the worst-case scenario) when accessing the element inside the linked list.

There are two major method to implement a linked list, through linear array or structures and pointers.

C++ Code Implementation

Here, we demonstrates the implementation of an integer singly linked list using structures and pointers, with some common operation of linked list. You can apply the same logic to implement doubly linked list and circular linked list.

```
struct Node {
    int val;
    Node* next;
};
// Sequentially traverse the list from head
void traverse_list(Node* head) {
    while (head != nullptr) {
        cout << head->val << endl;
        head = head->next;
    }
    cout << endl;
}
// Insert new_val at front of the linked list;
// Return the new head of the linked list
Node* insert_at_front(Node* head, int new_val) {
    Node new_node; new_node.val = new_val;
    Node* ptr = &new_node;
    new_node->next = head;
    return new_node
}
// Delete the head node of the linked list;
// Return the new head of the linked list
Node* delete_at_front(Node* head) {
    if (head == nullptr) return nullptr;
    Node* temp = head;
    head = head->next;
    delete temp; // Syntax: destroy objects from memory
    return head;
}
```

5 Monotonic Stack

Definition

A monotonic stack maintains its elements in a specific order inside a stack. Unlike traditional stacks, monotonic stacks ensure that elements inside the stack are arranged in an increasing or decreasing order based on their arrival time. In order to achieve the monotonic stacks, we have to enforce the push and pop operation depending on whether we want a monotonic increasing stack or monotonic decreasing stack.
To achieve a monotonic stack (e.g. increasing), we will compare the top of the stack with the element which is ready to be pushed inside the stack. The stack may be popped multiple times before actually pushing the element inside.

C++ Implementation (Monotonic Increasing Stack with Integers)

```
void mono_stack(vector<int>& nums) {
    stack<int> s;
    for (int i : nums) {
        while (!s.empty() && s.top() > i) s.pop();
        s.push(i);
    }
}
```

C++ Implementation (Monotonic Decreasing Stack with Integers)

```
void mono_stack(vector<int>& nums) {
    stack<int> s;
    for (int i : nums) {
        while (!s.empty() && s.top() < i) s.pop();
        s.push(i);
    }
}
```

6 Monotonic Queue

Definition

A monotonic queue maintains its elements in a specific order in a queue. Unlike traditional stacks, monotonic queues ensure that elements inside the queue are arranged in an increasing or decreasing order based on their arrival time. In order to achieve the monotonic stacks, we have to enforce a special data structure, **deque (double-ended queue)**, to facilitate pushing and popping from both ends.

To achieve a monotonic queue (e.g. increasing), we will compare the back of the queue with the element which is ready to be pushed inside the stack. The queue may be popped back multiple times before actually pushing the element inside.

C++ Implementation (Monotonic Increasing Queue with Integers)

```
void mono_stack(vector<int>& nums) {
    deque<int> q;
    for (int i : nums) {
        while (!q.empty() && q.back() > i) q.pop_back();
        q.push_back(i);
    }
}
```

C++ Implementation (Monotonic Decreasing Queue with Integers)

```
void mono_stack(vector<int>& nums) {
    deque<int> q;
    for (int i : nums) {
        while (!q.empty() && q.back() < i) q.pop_back();
        q.push_back(i);
    }
}
```