Student ID 2200520 NAME：Zhiqiang Geng

QUESTION ONE

(a) $Rt = n^2 + nlogn + n + 3$

The final time complexity is O(n2), since n^2 grows much faster than nlogn.

According to big O notation, which Big O specifically describes the worst case, an algorithm can be used to describe the required execution time or the space used (eg in memory or on disk).

(b) Use pseudocode to illustrate programming structures that have complexity O(1), O(n) andO(n2) and explain your answers

A:The following uses the pseudo code of several functions to illustrate.

```
void Func_O1()
 {
     int count = 0;
     for (int k = 0; k < 100; ++ k)
     {
         ++count;
     }
     // print something...
}
```

The time complexity of this algorithm function func_o1: O (1), because the number of cycles executes is constant times and the constant is 1.

```
void Func_On(int N)
{
     int count = 0;
     for (int k = 0; k < 2 * N ; ++ k)
     {
         ++count;
     }

     int M = 10;
     while (M--)
     {
         ++count;
       }
     // print something..
}
```

Analysis: The exact number of algorithm executions is 2 * n + 10; the time complexity is O(N) after using the Big O asymptotic notation rule;

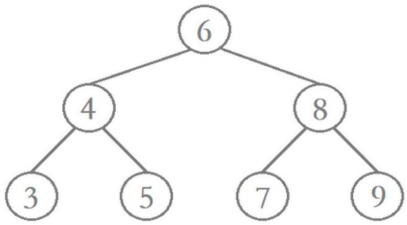
If the highest-order item exists and is not 1, then remove the constant multiplied by this item, and what remains will be O(N).

```
void Func_On2(int N) {
int count = 0;
//Two-layer loop nesting executes the outer loop n times, and the inner loop executes n times. The
//overall calculation is the number of executions of N*N
    for (int i = 0; i < N ; ++ i)
    {
        for (int j = 0; j < N ; ++ j)
        {
            ++count;
        }
    }
}
```

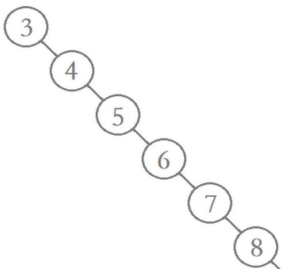
Analysis: The asymptotic notation time complexity of Big O is O(N2).

(c) Analyse the running time for a search operation in a binary search tree giving the best and worst case running times and show how these situations can occur.

A： As shown in the figure below, if the binary search tree is a complete binary tree, the search efficiency is very high, which is O(logN): .



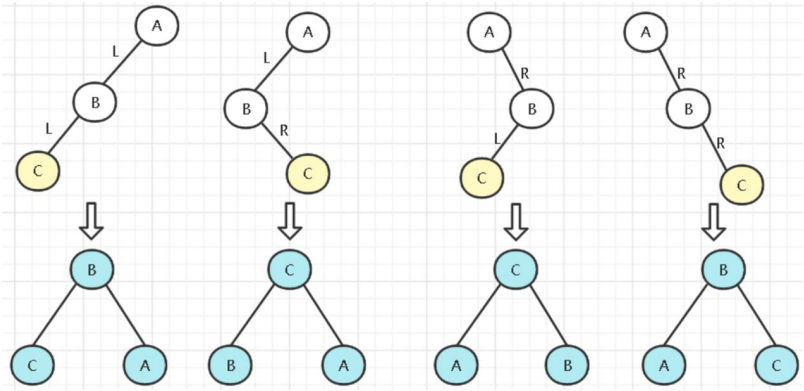
As shown in the figure below, there is still the worst case. In the worst case, the binary sorting tree degenerates into a single-branched tree, which can only be searched one by one from the root, and essentially becomes a sequential search.



(d) How does the use of a balanced tree improve upon a basic binary search tree?

A: If the height difference between the subtrees of the binary sorting tree is too large, the time complexity of the binary sorting tree operation will be upgraded to O(n). In order to avoid this situation and prepare for the worst case, use Balanced binary tree, also known as AVL tree, is used to solve the situation where the height of the binary sorting tree is uncertain.

The main operation of the AVL tree is to restore balance by rotation. Common operations are as follows:



QUESTION TWO

a) Explain the difference between static memory allocation and dynamic memoryallocation in the context of data structures.

A:

One is that the time is different. Static allocation occurs when the program is compiled and linked. Dynamic allocation occurs when the program is called and executed.

Second, the space is different. The heap is allocated dynamically, there is no statically allocated heap. There are two ways to allocate the stack: static allocation and dynamic allocation. Static allocation is done by the compiler, like the allocation of local variables. Dynamic allocation is allocated by the function malloc. However, the dynamic allocation of the stack is different from that of the heap. His dynamic allocation is released by the compiler without our manual implementation.

For the memory space of a process, it can be logically divided into three parts: code area, static data area and dynamic data area. The dynamic data area is generally "stack". "Stack" and "heap" are two different dynamic data areas, the stack is a linear structure, and the heap is a chain structure. Each thread of the process has a private "stack", so although the code of each thread is the same, the data of the local variables do not interfere with each other. A stack can be described by "base address" and "stack top" address. Global variables and static variables are allocated in the static data area, and local variables are allocated in the dynamic data area, namely the stack. The program accesses local variables through the base address and offset of the stack.

Generally, for variables modified with static, global variables are located in the static data area. The parameters, return address, EBP and local variables in the function call process are all stored in the stack.

In summary, static memory allocation is completed at compile time, the allocated memory space is fixed and cannot be changed dynamically, while dynamic memory allocation is performed at runtime, allowing programs to dynamically allocate and release memory as needed, but requires programmers to be responsible for managing memory allocation and release.

b) Explain with the aid of a diagram the term “node based linked list.” Identify and essential operations required manage this data structure. Provide reasons for your choices in your answer.

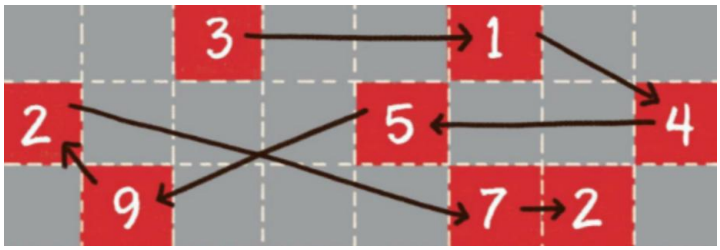
A: Linked list (linkedlist) is a physically discontinuous and non-sequential data structure, which is composed of several nodes (node); each node of the one-way linked list contains two parts, one part is the variable data for storing data, and the other part is A pointer to the next node next structure diagram:

Code：

```
private static class Node {
    int data;
    Node next;
}
```

The first node of the linked list is called the head node, and the last node is called the tail node, and the next pointer of the tail node points to nothing; unlike an array that randomly searches for elements according to subscripts, for one node A of the linked list, only Can find the next node B of the node according to the next pointer of node A, and then find the next node C according to the next pointer of node B... Level by level, single-line transmission! If you want each node to go back to its previous node, you can use a doubly linked list

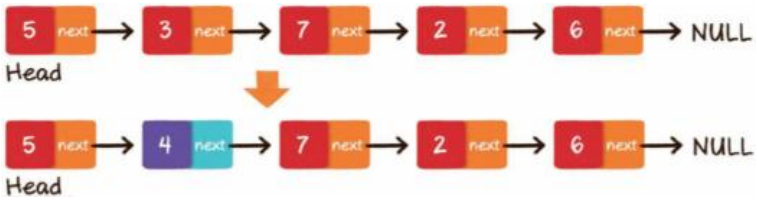
The memory allocation diagram of the linked list:



The basic operation of the linked list ——————

[1] Finding nodes: When searching for elements, unlike arrays, linked lists can be quickly located by subscripts, and can only be searched one by one from the head node backwards.

[2] Update nodes: If the process of finding nodes is not considered, the update process of the linked list will be as simple as an array, just replace the old data with new data



[2] Insert node linked list When inserting a node, there are three cases: ① Tail insertion Tail insertion just point the next pointer of the last node to the newly inserted node ② Head insertion Head insertion can be divided into two steps: 1. The first step is to point the next pointer of the new node to the original head node; the second step is to change the new node into the head node of the linked list ③ The middle insertion is also divided into two steps: the first step is the next pointer of the new node, which points to The node at the insertion position; Step 2, the next pointer of the preceding node at the insertion position points to the new node.

[4] The delete operation of the deleted element linked list is also divided into three cases: ①Delete the tail and delete the tail and set the next pointer of the penultimate node to empty. ②Delete the head and delete the head and set the head node of the linked list to the original head The next pointer of the node is enough. ③ Delete in the middle and delete in the middle. Just point the next pointer of the preceding node of the node to be deleted to the next node of the element to be deleted.

c) Using the ideas of dynamic memory allocation and the concept of a list node, describeusing pseudo-code a detailed algorithm for a function that will delete an item from adouble linked list. In your answer comment on any potential problems associated withyour implementation.

A: The following code describes the functional algorithm for deleting an item from a doubly linked list：

```cpp
struct Node {
    int data;
    Node* prev;
    Node* next;
};

void deleteItemFromDoublyLinkedList(Node*& head, Node*& tail, int item) {
    if (head == nullptr) {
        return;
    }

    Node* current = head;

    while (current != nullptr) {
        if (current->data == item) {
            if (current == head) {
                head = current->next;
                if (head != nullptr) {
                    head->prev = nullptr;
                } else {
                    tail = nullptr;
                }
            } else if (current == tail) {
                tail = current->prev;
                if (tail != nullptr) {
                    tail->next = nullptr;
                } else {
                    head = nullptr;
                }
            } else {
                current->prev->next = current->next;
                current->next->prev = current->prev;
            }

            delete current;  // Release the memory space of the deleted node
            return;
        }

        current = current->next;
    }

    // If the execution reaches here, it means that the item to be deleted is not found
    cout << "Item not found in the list." << endl;
}
```

There are mainly the following potential problems:

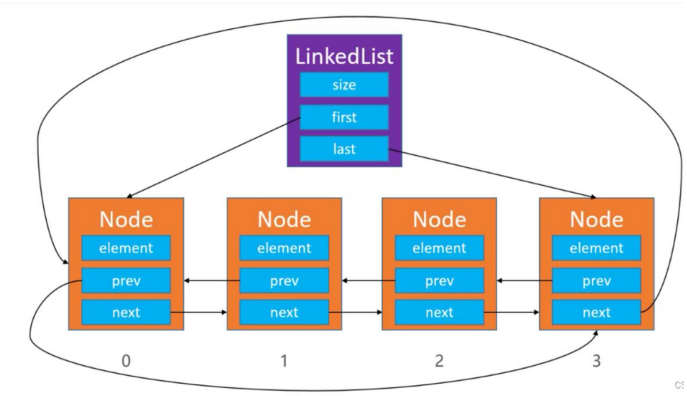
- The above code assumes that a structure named Node has been defined, which contains data items (data) and pointers to the previous node and the next node (prev and next).
- The parameters head and tail of the function represent the head pointer and tail pointer of the double linked list respectively. By passing references (Node*&), their values can be modified, and the head and tail pointers of the linked list can be updated after the delete operation.
- Deletion operations are divided into three cases: delete the head node, delete the tail node and delete the middle node. For each case, it is necessary to update the pointers of the predecessor and successor nodes of the corresponding node to ensure the continuity of the linked list.
- After deleting a node, use the delete operation to release the memory space of the deleted

node to prevent memory leaks.

- If the item to be deleted is not found, an appropriate error message is output.

d) Given a dynamic linearly linked list explain how it can be modified to implement acircular data structure

A: Take the doubly linked list as an example, ① First, find the tail node (ie the last node) of the linked list. ② Point the next pointer of the tail node to the head node of the linked list, and point the prev pointer of the head node to the tail node. This establishes a cyclic connection between the head and tail nodes. ③ Make sure that the head pointer of the linked list points to the new head node, that is, the original head node.



It should be noted that after modifying the linear linked list to the cycle list, you need additional conditions to control the cycle when traversing the linked list. For example, a special tag node (such as NullPtr) can be used to represent the end of the cycle, or the condition of the end of the cycle based on the application scenarios. Make sure that all related pointer operations are set correctly when modifying the linked list structure to avoid problems such as suspended pointers or memory leaks.

QUESTION THREE

a) Describe the stack and queue data structures including, with arguments for their inclusion, essential operations that are performed on each structure.

A：

The basic definition of a stack: a stack is a linear table limited to insert and delete operations only at the end of the table. For the stack, the end of the table has a special meaning, which is called the top of the stack, and the header section of the table is correspondingly the bottom of the stack. An empty list with no elements is called an empty stack. The modification of the stack is carried out according to the principle of last-in-first-out, so the stack becomes a linear list of last-in-first-out.

The data structure and basic operations of the stack.：

```
typedef int ElemType;
typedef struct Stack
{
	ElemType* data;
	int top;		// Stack top
	int capacity;	// Capacity
}Stack;

// Initialize the stack
void StackInit(Stack* ps)
{
	ps->data = (ElemType*)malloc(sizeof(ElemType) * Stack_Init_Size);
	if (NULL == ps->data)
	{
		printf("Failed to allocate memory\n");
		exit(0);
	}
	ps->capacity = Stack_Init_Size;
	ps->top = 0;
}

// Check for capacity and resize if necessary
void CapacityCheck(Stack* ps)
{
	assert(ps);
	if (ps->top >= ps->capacity)
	{
		ps->data = (ElemType*)realloc(sizeof(ElemType) * (ps->capacity + Stack_Add_Size));
		if (NULL == ps->data)
		{
			printf("Failed to allocate memory\n");
```

```
                exit(0);
            }
            ps->capacity += Stack_Add_Size;
        }
}

// Push an element onto the stack
void StackPush(Stack* ps, ElemType data)
{
        assert(ps);
        CapacityCheck(ps);
        ps->data[ps->top++] = data;
}

// Pop an element from the stack
void StackPop(Stack* ps)
{
        assert(ps);
        if (ps->top == 0)
        {
            printf("No elements in the current stack\n");
            return;
        }
        ps->top--;
}

// Get the top element of the stack
ElemType StackTop(Stack* ps)
{
        assert(ps);
        if (ps->top == 0)
        {
            printf("No elements in the current stack\n");
            return;
        }
        return ps->data[ps->top - 1];
}

// Get the number of valid elements in the stack
int StackSize(Stack* ps)
{
        return ps->top;
}
```

```
// Check if the stack is empty, return a non-zero value if empty, or 0 if not empty
int StackEmpty(Stack* ps)
{
	return ps->top == 0 ? 1 : 0;
}

// Destroy the stack
void StackDestroy(Stack* ps)
{
	assert(ps);
	if (ps->data)
	{
		free(ps->data);
		ps->data = NULL;
		ps->capacity = 0;
		ps->top = 0;
	}
}
```

Basic definition of queue：The queue is a linear data structure, and this data structure only allows insertion at one end and deletion at the other end, and direct access to all data other than these two ends is prohibited, and the queue is a first-in, first-out data structure.

The data structure and basic operations of the queue：

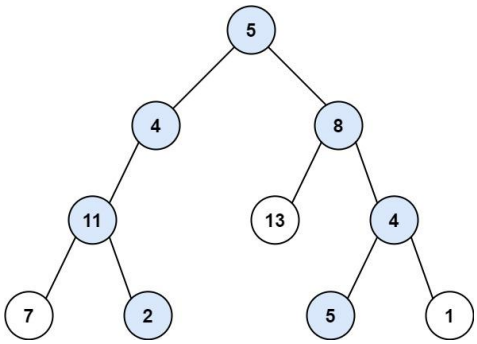
```
#define Queue_Init_Size 10

// Design the circular queue structure
typedef int ElemType;
typedef struct Queue
{
	ElemType* data;
	int front; // Index of the front element
	int rear;    // Index of the rear element
}Queue;

// Initialize the queue
void QueueInit(Queue* q)
{
	q->data = (ElemType*)malloc(sizeof(ElemType) * Queue_Init_Size);
	if (NULL == q->data)
	{
		printf("Failed to allocate memory\n");
		system("pause");
```

```
        exit(0);
    }
    q->front = 0;
    q->rear = 0;
}

// Enqueue at the rear of the queue
void QueuePush(Queue* q, ElemType data)
{
    assert(q);
    if ((q->rear + 1) % Queue_Init_Size == q->front)
    {
        printf("The current queue is full\n");
        return;
    }
    q->data[q->rear] = data;
    q->rear = (q->rear + 1) % Queue_Init_Size;
}

// Dequeue from the front of the queue
void QueuePop(Queue* q)
{
    assert(q);
    if (QueueEmpty(q))
    {
        printf("The current queue is empty\n");
        return;
    }
    q->front = (q->front + 1) % Queue_Init_Size;
}

// Get the front element of the queue
ElemType QueueFront(Queue* q)
{
    if (!QueueEmpty(q))
    {
        return q->data[q->front];
    }
}

// Get the rear element of the queue
ElemType QueueBack(Queue* q)
{
    if (!QueueEmpty(q))
```

```
    {
        return q->data[(q->rear - 1 + Queue_Init_Size) % Queue_Init_Size];
    }
}

// Get the number of valid elements in the queue
int QueueSize(Queue* q)
{
    return (q->rear - q->front + Queue_Init_Size) % Queue_Init_Size;
}

// Check if the queue is empty, return a non-zero value if empty, or 0 if not empty
int QueueEmpty(Queue* q)
{
    return q->front == q->rear;
}

// Destroy the queue
void QueueDestroy(Queue* q)
{
    assert(q);
    if (q->data != NULL)
    {
        free(q->data);
        q->data = NULL;
    }
    q->front = 0;
    q->rear = 0;
}
```

b) Provide an algorithm that implements an iterative depth first search using a stack datastructure. Analyse this algorithm and hence state its space and time complexity.

A：Here an example is used to explain the problem: given the root node of your binary tree and an integer targetSum and targetSum, find all paths from the root node to the leaf nodes whose sum is equal to the given target sum.



Here you can use the auxiliary stack to complete the DFS algorithm based on the iterative method: The relevant code is as follows:

```cpp
// Iterative implementation of depth-first search,
// using global variables for backtracking instead of parameters
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        vector<vector<int>> roads;
        vector<int> road;

        stack<pair<TreeNode*, bool>> st;
        if (root != nullptr) {
            st.push({root, false});
        }

        while (!st.empty()) {
            TreeNode* node = st.top().first;
            bool visited = st.top().second;
            st.pop();
            // If the node is visited, backtrack and remove the node's value from the current path
            if (visited) {
                targetSum += node->val;
                road.pop_back();
            } else {  // If the node is not visited, process the node and continue traversal
                targetSum -= node->val;
                road.push_back(node->val);
                // Check if the current path satisfies the target sum condition
                if (targetSum == 0 && node->left == nullptr && node->right == nullptr) {
                    roads.push_back(road);
                }

                // Push the node's children into the stack for traversal
                st.push({node, true});  // Mark the node as visited and backtrack later
                if (node->right != nullptr) {
                    st.push({node->right, false});
                }
                if (node->left != nullptr) {
                    st.push({node->left, false});
                }
            }
        }

        return roads;
    }
};
```

Correlation analyzes have been explained in detail in the Notes.

Time complexity analysis:

- The initialization part has a time complexity of O(1).
- In the while loop, we traverse the tree nodes to build the path and check if the path satisfies the target sum condition. Since it is a depth-first search, each node is visited at most once. Therefore, the time complexity of traversing the entire tree is O(N), where N is the total number of nodes in the tree.
- During the processing of each node, we perform some constant time operations such as updating the target sum, adding the node value to the path, backtracking, etc. The time complexity of these operations can be considered as O(1).
- Thus, the overall time complexity of the code snippet is O(N).

Space complexity analysis:

- Apart from the 2D array roads to store the resulting paths and the 1D array road to store the current path, we also use a stack st to assist in the iterative implementation of depth-first search. The stack can hold up to the number of nodes in the height of the tree simultaneously.
- In the worst case, if the tree is a complete binary tree, its height is O(logN). Therefore, the space complexity of the stack is O(logN).
- The space complexity of the 2D array roads depends on the number of valid paths stored, which is at most O(N).
- The space complexity of the 1D array road depends on the height of the tree, which is at most O(logN).
- Overall, the space complexity of the entire code snippet is O(logN).
- Note: The analysis assumes that no additional space is used apart from the input tree.

c) Apply your algorithm to the graph of figure Q3c, showing the contents of the stackafter each iteration of the loop. Explain how your depth first search algorithm might bemodified to perform, instead, a breadth first search.

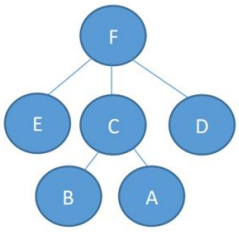


Figure Q3

A：

The content after each iteration

1st iteration：ECDF
2st iteration：ECDF
3st iteration：CDF
4st iteration：BACDF
5st iteration：ACDF
6st iteration：ACDF
7st iteration：CDF
8st iteration：CDF
9st iteration：DF
10st iteration：F

If it is very simple to change iterative DFS to BFS, change the stack to a queue, and then process all the elements in the queue at one time. code show as below:

```
int BFS(Node start, Node target) {
    queue<Node> q; // Core data structure
    unordered_set<Node> visited; // Avoid revisiting the same node
```

```
    q.push(start); // Add the starting point to the queue
    visited.insert(start);
    int step = 0; // Record the number of steps taken

    while (!q.empty()) {
        int sz = q.size();
        /* Spread all the nodes in the current queue in all directions */
        for (int i = 0; i < sz; i++) {
            Node cur = q.front();
            q.pop();
            /* Key point: Check if the target is reached here */
            if (cur == target)
                return step;
            /* Add adjacent nodes of cur to the queue */
            for (Node x : cur.adj()) {
                if (visited.count(x) == 0) {
                    q.push(x);
                    visited.insert(x);
                }
            }
        }
        /* Key point: Update the number of steps here */
        step++;
    }
}
```