# Linked List

The linked list problem is a favorite topic in C++ or Java, the key solution in such problem is to traverse the list, track the position of the target node and update the list while keeping its integrity. Because during the update it is easy to lose the link and the linked list become a corrupted list.

In real project we may maintain a double linked list, but in LeetCode, all the linked list is a singly linked list, which means only the next pointer will be used to maintain the list. There are some common tricks you may need to know in solving the linked list.

1. Create an empty head and add it to the front of the linked list. This will make some operation such as reverse works well.
2. Track the previous node of the target node is more important than keep the target node, because if you want to delete the target node, you should update the prev->next.
3. If you want to update a link such as prev->next, you should store it first, otherwise you will lose track on the remaining list.
4. If you want to know N positions ahead of the target node, you should use a slow traveler following the traversing node, with distance of N.

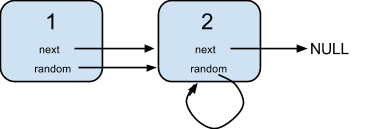
## 138. Copy List with Random Pointer

Medium

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list.

**Example 1:**

****

**Input:**

{"$id":"1","next":{"$id":"2","next":null,"random":{"$ref":"2"},"val":2},"random":{"$ref":"2"},"val":1}

**Explanation:**

Node 1's value is 1, both of its next and random pointer points to Node 2.

Node 2's value is 2, its next pointer points to null and its random pointer points to itself.

**Note:**

1. You must return the **copy of the given head** as a reference to the cloned list.

### Analysis:

This is a favorite problem for Microsoft and Amazon, to clone the linked list, we first ignore the random pointer but clone node by node and insert the new node right after the original node, which means the original node next point to the clone node, and the clone node next point to the next node in original list. The clone node now is side by side as the clone node.

Then we take care the random pointer, the original random pointer points to a random node, the clone node should point to the same node in the clone list, and we know it is right after the original random node.

In the last iteration we can disconnect the original list and the clone list by updating the next pointer. Actually, this step can be merged in the above when we update the random list.

/// <summary>

/// Leet code #138. Copy List with Random Pointer

///

/// A linked list is given such that each node contains an additional random

/// pointer which could point to any node in the list or null.

///

/// Return a deep copy of the list.

///

/// Example 1:

///

/// Input:

/// {"$id":"1","next":{"$id":"2","next":null,"random":{"$ref":"2"},"val":2},

/// "random":{"$ref":"2"},"val":1}

///

/// Explanation:

/// Node 1's value is 1, both of its next and random pointer points to Node 2.

/// Node 2's value is 2, its next pointer points to null and its random

/// pointer points to itself.

///

/// Note:

/// 1. You must return the copy of the given head as a reference to the

/// cloned list.

/// </summary>

RandomListNode \* LeetCodeLinkedList::copyRandomList(RandomListNode \*head)

{

RandomListNode \*node = head;

// clone the next link

while (node != nullptr)

{

RandomListNode\* new\_node = new RandomListNode(node->label);

new\_node->next = node->next;

node->next = new\_node;

node = new\_node->next;

}

// clone the random link

node = head;

while (node != nullptr)

{

RandomListNode\* new\_node = node->next;

if (node->random != nullptr)

{

new\_node->random = node->random->next;

}

node = new\_node->next;

}

// separate two link list

node = head;

RandomListNode \*new\_head = nullptr;

if (head != nullptr)

{

new\_head = head->next;

}

while (node != nullptr)

{

RandomListNode\* new\_node = node->next;

node->next = new\_node->next;

node = node->next;

if (node != nullptr)

{

new\_node->next = node->next;

}

}

return new\_head;

}

## 23. Merge k Sorted Lists

Hard

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

**Example:**

**Input:**

[

  1->4->5,

  1->3->4,

  2->6

]

**Output:** 1->1->2->3->4->4->5->6

### Analysis:

For every head node in the array, we put the negative value and its node in the priority queue and keep on popping from the priority queue.

/// <summary>

/// Leet code #23. Merge k Sorted Lists

///

/// Merge k sorted linked lists and return it as one sorted list. Analyze and

/// describe its complexity.

///

/// Example:

///

/// Input:

/// [

/// 1->4->5,

/// 1->3->4,

/// 2->6

/// ]

/// Output: 1->1->2->3->4->4->5->6

/// </summary>

ListNode\* LeetCodeLinkedList::mergeKLists(vector<ListNode\*>& lists)

{

ListNode \* head = nullptr;

ListNode\* curr = nullptr;

priority\_queue<pair<int, ListNode\*>> heap;

for (size\_t i = 0; i < lists.size(); i++)

{

ListNode\* node = lists[i];

if (node != nullptr)

{

heap.push(make\_pair(-node->val, node));

}

}

while (!heap.empty())

{

pair<int, ListNode\*> first = heap.top();

heap.pop();

if (first.second != nullptr && first.second->next != nullptr)

{

heap.push(make\_pair(-first.second->next->val, first.second->next));

}

if (head == nullptr)

{

head = first.second;

curr = head;

}

else

{

curr->next = first.second;

curr = curr->next;

}

if (curr != nullptr) curr->next = nullptr;

}

return head;

}

## 25. Reverse Nodes in k-Group

Hard

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

*k* is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes in the end should remain as it is.

**Example:**

Given this linked list: 1->2->3->4->5

For *k* = 2, you should return: 2->1->4->3->5

For *k* = 3, you should return: 3->2->1->4->5

**Note:**

* Only constant extra memory is allowed.
* You may not alter the values in the list's nodes, only nodes itself may be changed.

### Analysis:

We can add an empty head, and reverse K node, then move to the end of group and reverse next group. Before we start to reverse we first test if we have k node in the group, of course if we can also reverse again if we found there is less than k nodes in a group.

/// <summary>

/// Leet code #25. Reverse Nodes in k-Group

///

/// Given a linked list, reverse the nodes of a linked list k at a time and

/// return its modified list.

///

/// k is a positive integer and is less than or equal to the length of the

/// linked list. If the number of nodes is not a multiple of k then left-out

/// nodes in the end should remain as it is.

///

/// Example:

///

/// Given this linked list: 1->2->3->4->5

///

/// For k = 2, you should return: 2->1->4->3->5

///

/// For k = 3, you should return: 3->2->1->4->5

///

/// Note:

///

/// 1. Only constant extra memory is allowed.

/// 2. You may not alter the values in the list's nodes, only nodes itself

/// may be changed.

/// </summary>

ListNode\* LeetCode::reverseKGroup(ListNode\* head, int k)

{

ListNode\* fake\_head = new ListNode(0);

fake\_head->next = head;

ListNode\* prev = fake\_head;

ListNode\* last = head;

while (last != nullptr && last->next != nullptr)

{

// test if we have k nodes

ListNode\* next = prev;

for (int i = 0; i < k; i++)

{

if (next == nullptr) break;

next = next->next;

}

if (next == nullptr) break;

for (int i = 0; i < k - 1; i++)

{

// watch the order of operation.

next = last->next;

last->next = next->next;

next->next = prev->next;

prev->next = next;

}

prev = last;

last = last->next;

}

head = fake\_head->next;

delete fake\_head;

return head;

}

## 19. Remove Nth Node From End of List

Medium

Given a linked list, remove the *n*-th node from the end of list and return its head.

**Example:**

Given linked list: **1->2->3->4->5**, and ***n* = 2**.

After removing the second node from the end, the linked list becomes **1->2->3->5**.

**Note:**

Given *n* will always be valid.

**Follow up:**

Could you do this in one pass?

### Analysis:

We have a slow iterator which is N step after the fast iterator, to make it able to remove the node, we should record the previous node instead of the target node. For this case, we add a fake head.

/// <summary>

/// Leet code #19. Remove Nth Node From End of List

///

/// Given a linked list, remove the n-th node from the end of list and return

/// its head.

///

/// Example:

///

/// Given linked list: 1->2->3->4->5, and n = 2.

///

/// After removing the second node from the end, the linked list

/// becomes 1->2->3->5.

///

/// Note:

/// 1. Given n will always be valid.

///

/// Follow up:

///

/// 1. Could you do this in one pass?

/// </summary>

ListNode\* LeetCodeLinkedList::removeNthFromEnd(ListNode\* head, int n)

{

ListNode\* fake\_head = new ListNode(0);

fake\_head->next = head;

ListNode\* first = fake\_head;

ListNode\* last = fake\_head;

for (int i = 0; i < n; i++)

{

last = last->next;

}

while (last->next != nullptr)

{

first = first->next;

last = last->next;

}

ListNode \* next = first->next;

if (next != nullptr) first->next = next->next;

delete next;

head = fake\_head->next;

delete fake\_head;

return head;

}

## 143. Reorder List

Medium

Given a singly linked list *L*: *L*0→*L*1→…→*Ln*-1→*L*n,  
reorder it to: *L*0→*Ln*→*L*1→*Ln*-1→*L*2→*Ln*-2→…

You may **not** modify the values in the list's nodes, only nodes itself may be changed.

**Example 1:**

Given 1->2->3->4, reorder it to 1->4->2->3.

**Example 2:**

Given 1->2->3->4->5, reorder it to 1->5->2->4->3.

### Analysis:

We should position to the middle of the list first, then reverse the second half and finally merge them.

/// <summary>

/// Leet code #143. Reorder List

///

/// Given a singly linked list L: L0→L1→…→Ln-1→Ln,

/// reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→…

///

/// You may not modify the values in the list's nodes, only nodes itself

/// may be changed.

///

/// Example 1:

///

/// Given 1->2->3->4, reorder it to 1->4->2->3.

///

/// Example 2:

///

/// Given 1->2->3->4->5, reorder it to 1->5->2->4->3.

/// </summary>

void LeetCode::reorderList(ListNode\* head)

{

if ((head == nullptr) || head->next == nullptr) return;

ListNode\* fake\_head = new ListNode(0);

fake\_head->next = head;

ListNode\* slow = fake\_head;

ListNode \*fast = fake\_head;

while (fast->next != nullptr)

{

fast = fast->next;

if (fast->next != nullptr) fast = fast->next;

slow = slow->next;

}

// reverse second half

fast = slow->next;

while (fast->next != nullptr)

{

ListNode \* next = fast->next;

fast->next = next->next;

next->next = slow->next;

slow->next = next;

}

// merge the second half

fast = slow;

slow = head;

while (slow != fast)

{

ListNode \* next = fast->next;

fast->next = next->next;

next->next = slow->next;

slow->next = next;

slow = next->next;

}

delete fake\_head;

}

## 148. Sort List

Medium

Sort a linked list in *O*(*n* log *n*) time using constant space complexity.

**Example 1:**

**Input:** 4->2->1->3

**Output:** 1->2->3->4

**Example 2:**

**Input:** -1->5->3->4->0

**Output:** -1->0->3->4->5

### Analysis:

The straight forward sort for linked list is bubble sort, but it will not be O(nlog(n)). A stable O(nlog(n)) algorithm is merge sort, which is to sort two adjacent items then merge them, then merge 4 adjacent items, then 8… until you merge n/2 with n/2 to get n sorted list. This sorting algorithm is easier to implement than you think, you can use recursive call, so every time you just need to position the middle of the list which you suppose to sort, here we use fast traveler which traverse two nodes at a step and slow traveler which traverse one node at a step.

/// <summary>

/// Leet code #21. Merge Two Sorted Lists

///

/// Merge two sorted linked lists and return it as a new list. The new list

/// should be made by splicing together the nodes of the first two lists.

///

/// Example:

/// Input: 1->2->4, 1->3->4

/// Output: 1->1->2->3->4->4

/// </summary>

ListNode\* LeetCodeLinkedList::mergeTwoLists(ListNode\* l1, ListNode\* l2)

{

ListNode\* fake\_head = new ListNode(0);

ListNode\* head = nullptr;

ListNode\* prev = fake\_head;

while ((l1 != nullptr) || (l2 != nullptr))

{

if (l1 == nullptr)

{

prev->next = l2;

l2 = nullptr;

}

else if (l2 == nullptr)

{

prev->next = l1;

l1 = nullptr;

}

else if (l1->val < l2->val)

{

prev->next = l1;

l1 = l1->next;

}

else

{

prev->next = l2;

l2 = l2->next;

}

prev = prev->next;

}

head = fake\_head->next;

delete fake\_head;

return head;

}

/// <summary>

/// Leet code #148. Sort List

///

/// Sort a linked list in O(n log n) time using constant space complexity.

///

/// Example 1:

/// Input: 4->2->1->3

/// Output: 1->2->3->4

///

/// Example 2:

/// Input: -1->5->3->4->0

/// Output: -1->0->3->4->5

/// </summary>

ListNode\* LeetCodeLinkedList::sortList(ListNode\* head)

{

if ((head == nullptr) || (head->next == nullptr)) return head;

ListNode\* slow = head, \* fast = head;

while (fast != nullptr)

{

fast = fast->next;

if (fast != nullptr) fast = fast->next;

if (fast == nullptr) break;

slow = slow->next;

}

fast = slow->next;

slow->next = nullptr;

slow = sortList(head);

fast = sortList(fast);

return mergeTwoLists(slow, fast);

}

## 142. Linked List Cycle II

Medium

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

To represent a cycle in the given linked list, we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to. If pos is -1, then there is no cycle in the linked list.

**Note:** Do not modify the linked list.

**Example 1:**

**Input:** head = [3,2,0,-4], pos = 1

**Output:** tail connects to node index 1

**Explanation:** There is a cycle in the linked list, where tail connects to the second node.



**Example 2:**

**Input:** head = [1,2], pos = 0

**Output:** tail connects to node index 0

**Explanation:** There is a cycle in the linked list, where tail connects to the first node.



**Example 3:**

**Input:** head = [1], pos = -1

**Output:** no cycle

**Explanation:** There is no cycle in the linked list.



**Follow-up**:  
Can you solve it without using extra space?

### Analysis:

This is another typicla problem, we have a fast traveler which moves 2 nodes a step and a slow traveler which moves 1 node a step. If they meet, we know there is a cycle. Then we move the fast traveler from the head along with a slow traveler, both move 1 node a step, until they meet it is the start of cycle.

The theory is this, assume H steps before the cycle and L is the circle length and when they meet it is D steps ahead of start of cycle.

2\*(H + D) = L + D + H -> H = L -D.

Consider the following linked list, where E is the cylce entry and X, the crossing point of fast and slow.

H: distance from head to cycle entry E

D: distance from E to X

L: cycle length

\_\_\_\_\_

/ \

head\_\_\_\_\_H\_\_\_\_\_\_E \

\ /

X\_\_\_\_\_/

If fast and slow both start at head, when fast catches slow, slow has traveled H+D and fast 2(H+D).

Assume fast has traveled n loops in the cycle, we have:

2H + 2D = H + D + L --> H + D = nL --> H = nL - D

Thus if two pointers start from head and X, respectively, one first reaches E, the other also reaches E.

/// <summary>

/// Leet code #142. Linked List Cycle II

///

/// Given a linked list, return the node where the cycle begins. If there

/// is no cycle, return null.

///

/// To represent a cycle in the given linked list, we use an integer pos

/// which represents the position (0-indexed) in the linked list where tail

/// connects to. If pos is -1, then there is no cycle in the linked list.

///

/// Note: Do not modify the linked list.

///

///

/// Example 1:

/// Input: head = [3,2,0,-4], pos = 1

/// Output: tail connects to node index 1

/// Explanation: There is a cycle in the linked list, where tail connects to

/// the second node.

///

///

/// Example 2:

///

/// Input: head = [1,2], pos = 0

/// Output: tail connects to node index 0

/// Explanation: There is a cycle in the linked list, where tail connects to

/// the first node.

///

/// Example 3:

///

/// Input: head = [1], pos = -1

/// Output: no cycle

/// Explanation: There is no cycle in the linked list.

///

/// Follow-up:

/// Can you solve it without using extra space?

/// </summary>

ListNode\* LeetCodeLinkedList::detectCycle(ListNode\* head)

{

ListNode\* slow = head;

ListNode\* fast = head;

while (fast != nullptr)

{

slow = slow->next;

fast = fast->next;

if (fast != nullptr)

{

fast = fast->next;

if (slow == fast)

{

break;

}

}

}

if (fast == nullptr)

{

return nullptr;

}

else

{

fast = head;

while (slow != fast)

{

slow = slow->next;

fast = fast->next;

}

return slow;

}

}

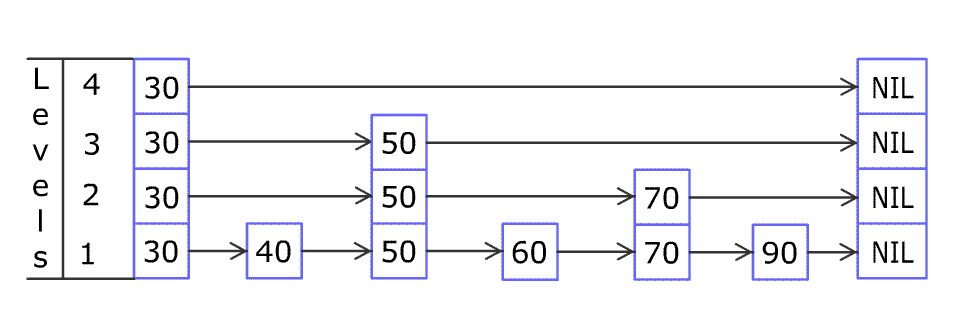
## 1206. Design Skiplist

Hard

Design a Skiplist without using any built-in libraries.

A Skiplist is a data structure that takes O(log(n)) time to *add*, *erase* and *search*. Comparing with treap and red-black tree which has the same function and performance, the code length of Skiplist can be comparatively short and the idea behind Skiplists are just simple linked lists.

For example: we have a Skiplist containing *[30,40,50,60,70,90]* and we want to add *80* and *45* into it. The Skiplist works this way:

  
Artyom Kalinin [CC BY-SA 3.0], via [Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Skip_list_add_element-en.gif)

You can see there are many layers in the Skiplist. Each layer is a sorted linked list. With the help of the top layers, *add* , *erase* and *search*can be faster than O(n). It can be proven that the average time complexity for each operation is O(log(n)) and space complexity is O(n).

To be specific, your design should include these functions:

* bool search(int target) : Return whether the target exists in the Skiplist or not.
* void add(int num): Insert a value into the SkipList.
* bool erase(int num): Remove a value in the Skiplist. If num does not exist in the Skiplist, do nothing and return false. If there exists multiple num values, removing any one of them is fine.

See more about Skiplist : <https://en.wikipedia.org/wiki/Skip_list>

Note that duplicates may exist in the Skiplist, your code needs to handle this situation.

**Example:**

Skiplist skiplist = new Skiplist();

skiplist.add(1);

skiplist.add(2);

skiplist.add(3);

skiplist.search(0); // return false.

skiplist.add(4);

skiplist.search(1); // return true.

skiplist.erase(0); // return false, 0 is not in skiplist.

skiplist.erase(1); // return true.

skiplist.search(1); // return false, 1 has already been erased.

**Constraints:**

* 0 <= num, target <= 20000
* At most 50000 calls will be made to search, add, and erase.

### Analysis:

This problem may be beyond the scope of interview, but we can take it as an exercise. The skip list has multi-layer a node can be promoted depending on the random probability, if it is 50% or less, it moves up a level. A delete will always delete the node from top to bottom to make our life easy, during the travel from top to bottom, we will always remember the previous node, so if we need to promote a node, we know after which node we can add.

/// <summary>

/// Leet code #1206. Design Skiplist

///

/// Design a Skiplist without using any built-in libraries.

///

/// A Skiplist is a data structure that takes O(log(n)) time to add, erase

/// and search. Comparing with treap and red-black tree which has the same

/// function and performance, the code length of Skiplist can be

/// comparatively short and the idea behind Skiplists are just simple linked

/// lists.

///

/// For example: we have a Skiplist containing [30,40,50,60,70,90] and we want

/// to add 80 and 45 into it. The Skiplist works this way:

///

/// Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

///

/// You can see there are many layers in the Skiplist. Each layer is a sorted

/// linked list. With the help of the top layers, add , erase and search can

/// be faster than O(n). It can be proven that the average time complexity

/// for each operation is O(log(n)) and space complexity is O(n).

///

/// To be specific, your design should include these functions:

///

/// bool search(int target) : Return whether the target exists in the Skiplist

/// or not.

///

/// void add(int num): Insert a value into the SkipList.

///

/// bool erase(int num): Remove a value in the Skiplist. If num does not exist

/// in the Skiplist, do nothing and return false. If there exists multiple num

//// values, removing any one of them is fine.

/// See more about Skiplist : https://en.wikipedia.org/wiki/Skip\_list

///

/// Note that duplicates may exist in the Skiplist, your code needs to handle

/// this situation.

///

/// Example:

///

/// Skiplist skiplist = new Skiplist();

///

/// skiplist.add(1);

/// skiplist.add(2);

/// skiplist.add(3);

/// skiplist.search(0); // return false.

/// skiplist.add(4);

/// skiplist.search(1); // return true.

/// skiplist.erase(0); // return false, 0 is not in skiplist.

/// skiplist.erase(1); // return true.

/// skiplist.search(1); // return false, 1 has already been erased.

///

/// Constraints:

///

/// 1. 0 <= num, target <= 20000

/// 2. At most 50000 calls will be made to search, add, and erase.

/// </summary>

class Skiplist

{

private:

struct SkipNode

{

int value;

SkipNode\* down;

SkipNode\* prev;

SkipNode\* next;

SkipNode(int v)

{

value = v;

down = nullptr;

prev = nullptr;

next = nullptr;

}

};

vector<SkipNode\*> skip\_layer;

SkipNode \* search\_next(SkipNode\* start, int target)

{

SkipNode\* node = start;

while (node->next != nullptr && target > node->next->value)

{

node = node->next;

}

return node;

}

SkipNode\* search\_down(int target, vector<SkipNode \*>& layer\_nodes)

{

SkipNode\* node = nullptr;

for (int i = skip\_layer.size() - 1; i >= 0; i--)

{

if (i == skip\_layer.size() - 1)

{

node = skip\_layer[i];

}

else

{

if (node != nullptr) node = node->down;

}

node = search\_next(node, target);

layer\_nodes.push\_back(node);

}

return node;

}

public:

Skiplist()

{

SkipNode\* prev = nullptr;

srand((unsigned int)time(0));

for (size\_t i = 0; i < 32; i++)

{

SkipNode\* node = new SkipNode(0);

skip\_layer.push\_back(node);

if (prev != nullptr)

{

node->down = prev;

}

prev = node;

}

}

~Skiplist()

{

for (size\_t i = 0; i < skip\_layer.size(); i++)

{

SkipNode\* node = skip\_layer[i];

while (node != nullptr)

{

SkipNode\* temp = node;

node = node->next;

delete temp;

}

}

}

bool search(int target)

{

SkipNode\* start = nullptr;

vector<SkipNode \*> layer\_nodes;

SkipNode\* node = search\_down(target, layer\_nodes);

if (node != nullptr && node->next != nullptr && node->next->value == target)

{

return true;

}

else

{

return false;

}

}

void add(int num)

{

vector<SkipNode\*> layer\_nodes;

SkipNode\* node = search\_down(num, layer\_nodes);

int n = 1;

SkipNode\* down = nullptr;

while (!layer\_nodes.empty())

{

// coin flip negative stop pop up

if (n == 0) break;

node = layer\_nodes.back();

layer\_nodes.pop\_back();

SkipNode\* new\_node = new SkipNode(num);

new\_node->next = node->next;

if (node->next != nullptr) node->next->prev = new\_node;

node->next = new\_node;

if (new\_node != nullptr) new\_node->prev = node;

if (new\_node != nullptr) new\_node->down = down;

down = new\_node;

n = rand() % 2;

}

}

bool erase(int num)

{

bool result = false;

vector<SkipNode\*> layer\_nodes;

SkipNode\* node = search\_down(num, layer\_nodes);

while (!layer\_nodes.empty())

{

node = layer\_nodes.back();

layer\_nodes.pop\_back();

if (node != nullptr && node->next != nullptr && node->next->value == num)

{

result = true;

SkipNode\* temp = node->next;

node->next = temp->next;

if (node->next != nullptr)

{

node->next->prev = node;

}

delete temp;

}

}

return result;

}

void print()

{

printf("#\n");

for (size\_t i = 0; i < skip\_layer.size(); i++)

{

SkipNode\* node = skip\_layer[i];

if (node == nullptr || node->next == nullptr) break;

node = node->next;

string message;

while (node != nullptr)

{

if (!message.empty())

{

message.append("->");

}

message.append(to\_string(node->value));

node = node->next;

}

printf("%s\n", message.c\_str());

}

}

};