中国科学院大学网络空间安全学院
数据结构与算法分析研讨课
实验报告

实验序号：1　　实验名称：单链表与跳表对比分析实验

1　单链表基本操作

1.1　创建单链表

```
  ListNode* CreatNode(int val){
    ListNode* Node=(ListNode*)malloc(sizeof(*Node));
    Node->val=val;
    Node->next=NULL;
    return Node;
}
  ListNode* CreatList(int size){
    ListNode* head =CreatNode(1);
    ListNode* ptr =head;
    for (int i = 2; i <=size; i++)
    {
        ListNode* NewNode=(ListNode*)malloc(sizeof(*NewNode));
        ptr->next=NewNode;
        NewNode->val= i;
        NewNode->next=NULL;
        ptr=NewNode;
    }
    return head;
}
```

1.2 插入节点

```
    ListNode* InsertList(ListNode*head,int position,int data){
    ListNode* ptr=head;
    ListNode*NewNode=(ListNode*)malloc(sizeof(ListNode));
    if (position==0)
    {
        NewNode->val=data;
        NewNode->next=head;
        return NewNode;
    }else{
    for (int i = 0; i < position-1 && ptr!=NULL; i++)
    {
        ptr=ptr->next;
    }
    if (ptr == NULL) return head;
    NewNode->val=data;
    NewNode->next= ptr->next;
    ptr->next=NewNode;
    }
    return head;
}
```

1.3 删除节点

```
ListNode* DeleteList(ListNode* head,int position){
    if (head == NULL) {
    printf("Error: List is empty!\n");
    return NULL;
    }
    if (position==0)
    {
        ListNode* temp=head;
        head=head->next;
        free(temp);
        return head;
    }

    ListNode* ptr =head;
```

```
    for (int i = 0; i < position-1 && ptr!=NULL; i++)
    {
        ptr=ptr->next;
    }
    if (ptr == NULL || ptr->next == NULL) return head;
    ListNode* temp=ptr->next;
    ptr->next=ptr->next->next;
    free(temp);
    return head;
}
```

1.4 查找节点

```
ListNode* FindList(ListNode* head,int data){
    ListNode* ptr=head;
    if (ptr == NULL) {
    printf("NOT FIND");
    return NULL;
    }
    while (ptr!=NULL&&ptr->val!=data)
    {
        ptr=ptr->next;
    }
    if (ptr==NULL)
    {
        printf("NOT FIND");
        return NULL;
    }else{
        return ptr;
    }
}
```

1.5 销毁链表

```
    ListNode* DestroyList(ListNode* head){
    ListNode* ptr=head;
    while (ptr!=NULL)
    {
        ListNode*temp=ptr;
```

```
        ptr=ptr->next;
        free(temp);
    }
    return NULL;
}
```

2 跳表基本操作

[1]

2.1 创建跳表

```
SkipListNode* CreatNode(int val,int level){
    SkipListNode* newnode=(SkipListNode*)malloc(sizeof(SkipListNode));
    newnode->val=val;
    newnode->forward=(SkipListNode**)malloc(sizeof(SkipListNode*)*(level+1));
    return newnode;
}

SkipList* CreatList(){
    SkipList* list=(SkipList*)malloc(sizeof(SkipList));
    list->level=0;
    list->head=CreatNode(INT_MIN,LEVEL);
    for (int i = 0; i < LEVEL; i++)
    {
        list->head->forward[i]=NULL;
    }
    return list;
}
```

2.2 插入节点

```
int randomlevel(){
    int level=0;
    while (rand()%2==0&&level<LEVEL)
    {
        level++;
    }
```

```
    return level;
}

void InsertNode(SkipList* list,int val){
    SkipListNode* update[LEVEL];
    SkipListNode* current=list->head;
    for (int i = list->level; i >= 0 ; i--)
    {
        while (current->forward[i]!=NULL&&current->forward[i]->val<val)
        {
            current = current->forward[i];
        }
        update[i]=current;

    }
    int newlevel=randomlevel();
    if(newlevel>list->level){
        for (int i = list->level+1; i <=newlevel; i++)
        {
            update[i]=list->head;
        }
        list->level=newlevel;
    }
    SkipListNode* newnode =CreatNode(val,newlevel);
    for (int i = 0; i <=newlevel; i++)
    {
        newnode->forward[i]=update[i]->forward[i];
        update[i]->forward[i]= newnode;
    }

}
```

2.3 删除结点

```
void freeNode(SkipListNode* node) {
    free(node->forward);
    free(node);
}

```

```
void DeleteNode(SkipList* list, int val)
{
    SkipListNode* update[LEVEL];
    SkipListNode* current = list->head;
    for (int i = list->level; i >= 0; i--)
    {
        while (current->forward[i] != NULL && current->forward[i]->val < val)
        {
            current = current->forward[i];
        }
        update[i] = current;
    }
    current = current->forward[0];
    if (current != NULL && current->val == val)
    {
        for (int i = 0; i <= list->level; i++)
        {
            if (update[i]->forward[i] != current)
                break;
            update[i]->forward[i] = current->forward[i];
        }
        while (list->level > 0 && list->head->forward[list->level] == NULL)
        {
            list->level--;
        }
        freeNode(current);
    }
    else
    {
        printf("Node with value %d not found.\n", val);
    }
}
```

2.4 查找节点

```
SkipListNode* SearchNode(SkipList* list,int val){
    SkipListNode* current = list->head;
    for (int i = list->level; i >=0; i--)
    {
```

```
        while (current->forward[i]!=NULL&&current->forward[i]->val<val)
        {
            current=current->forward[i];
        }

    }
    if (current->forward[0]!=NULL&& current->forward[0]->val==val)
    {
        return current->forward[0];
    }
    return NULL;
}
```

2.5 销毁跳表

```
void freeSkipList(SkipList* list) {
    SkipListNode* current = list->head->forward[0];
    SkipListNode* next;

    while (current != NULL) {
        next = current->forward[0];
        freeNode(current);
        current = next;
    }

    free(list->head->forward);
    free(list->head);
    free(list);
}
```

3 时间复杂度分析

3.1 各操作时间复杂度对比

对于跳表时间复杂度的具体分析详见第四部分。

3.1.1 查找操作

对于跳表，查找操作的时间复杂度为 O(log n)；对于单链表，查找操作的时间复杂度为 O(n)。

由于链表是单向的，所以在查找第 K 个元素时需要从头遍历整个链表，因此时间复杂度是 O(n)。

3.1.2 插入操作

对于跳表，插入操作的时间复杂度为 O(log n)；对于单链表，若已知第 K 个元素的位置，在第 K 个元素前插入元素，时间复杂度为 O(n)；在第 K 个元素后插入元素，时间复杂度为 O(1)。若采用置换的方式进行插入，则在第 K 个元素前插入元素，时间复杂度可优化为 O(1)。

3.1.3 删除操作

对于跳表，删除操作的时间复杂度为 O(log n)；对于单链表，若已知第 K 个元素的位置，采用从头遍历的方式删除第 K 个元素，则时间复杂度为 O(n)；若对删除操作进行优化，将第 K 个元素与它的后继置换，再删除，则时间复杂度可优化至 O(1)。

3.2 具体实验对比

时间测量函数代码如下：

跳表：

```
void measuretime(int node,int insertcount){
    printf("Node:%d\n",node);
    SkipList* list = CreatList();

    for (int i = 1; i <= node; i++)
    {
        InsertNode(list, i);
    }

    clock_t start = clock();
    for (int i = 1; i < 4000; i++)
    {
        DeleteNode(list,i);
    }
    clock_t end = clock();
    printf("Skip List deletion time: %.6f seconds\n", (double)(end - start)
       / CLOCKS_PER_SEC);

    start = clock();
    for (int i = 1; i <= insertcount; i++)
    {
```

```
        InsertNode(list, i);
    }
    end = clock();
    printf("Skip List insertion time: %.6f seconds\n", (double)(end - start)
        / CLOCKS_PER_SEC);

    SkipListNode* foundNode = SearchNode(list, 5000);
    printf("Found node with value: %d\n", foundNode->val);
    freeSkipList(list);
}
```

单链表:

```
void measuretime(int node,int insertcount){
    printf("Node:%d\n",node);
    ListNode*list= CreatList(node);

    clock_t start = clock();
    for (int i = 1; i < insertcount; i++)
    {
        list=DeleteList(list,i);
    }
    clock_t end = clock();
    printf("Single Linked List deletion time: %.6f seconds\n", (double)(end
        - start) / CLOCKS_PER_SEC);

    start = clock();
    for (int i = 0; i < insertcount; i++) {
        list = InsertList(list, i, 100 + i);
    }
    end = clock();
    printf("Single Linked List insertion time for %d nodes: %.6f
        seconds\n",insertcount,(double)(end - start) / CLOCKS_PER_SEC);

    list=DestroyList(list);

}
```

分别测量节点数为 15000、12000、10000、8000、5000 的单链表和跳表对 8000、6000、5000、4000、3000 个元素进行插入和删除操作所花费的时间，得到结果如下所示:

```
Node:15000
```

```
Single Linked List deletion time: 0.093000 seconds
Single Linked List insertion time for 8000 nodes: 0.083000 seconds
Node:12000
Single Linked List deletion time: 0.053000 seconds
Single Linked List insertion time for 6000 nodes: 0.052000 seconds
Node:10000
Single Linked List deletion time: 0.042000 seconds
Single Linked List insertion time for 5000 nodes: 0.037000 seconds
Node:8000
Single Linked List deletion time: 0.025000 seconds
Single Linked List insertion time for 4000 nodes: 0.024000 seconds
Node:5000
Single Linked List deletion time: 0.013000 seconds
Single Linked List insertion time for 3000 nodes: 0.013000 seconds
```

```
Node:15000
Skip List deletion time: 0.000000 seconds
Skip List insertion time: 0.002000 seconds
Found node with value: 5000
Node:12000
Skip List deletion time: 0.000000 seconds
Skip List insertion time: 0.001000 seconds
Found node with value: 5000
Node:10000
Skip List deletion time: 0.000000 seconds
Skip List insertion time: 0.001000 seconds
Found node with value: 5000
Node:8000
Skip List deletion time: 0.000000 seconds
Skip List insertion time: 0.001000 seconds
Found node with value: 5000
Node:5000
Skip List deletion time: 0.000000 seconds
Skip List insertion time: 0.001000 seconds
Found node with value: 5000
```

使用 Python 绘制对比图，如下图所示：

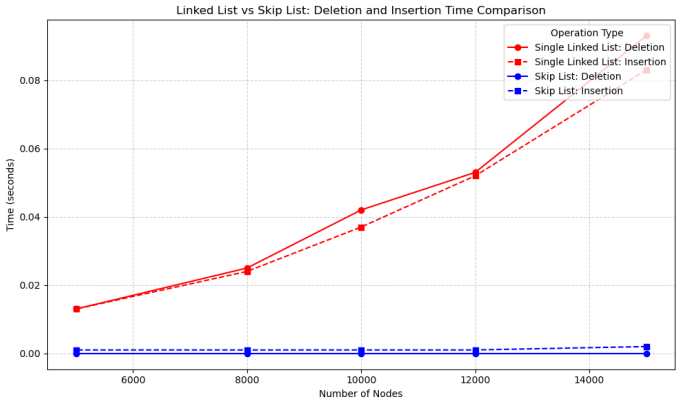

图 1: 单链表与跳表时间操作复杂度对比图

由图可以直观看出，跳表插入和删除操作所花费的时间明显小于单链表所花费的时间，这一实验结果也符合上述分析结果，即跳表时间复杂度为 O(log n)，单链表为 O(n)。

但同时实验仍存在一些问题，比如每次运行程序所得到的结果均有一些差别；而跳表每次运行后时间会变短很多，初步猜测可能是因为内存分配和缓存效应的原因；且跳表插入操作所需的时间可能是由于时间精度的限制，一直显示为 0。

4 跳表时间复杂度分析

4.1 查找操作

查找元素的过程是从最高级索引开始，一层一层遍历最后下沉到原始链表。所以，时间复杂度 = 索引的高度 * 每层索引遍历元素的个数。[2] 由于 k 级索引有 $\frac{n}{2^k}$ 个元素，而最高级 h 级索引一般有 2 个索引，故可由 $2 = \frac{n}{2^h}$ 求得 $h = log_2 n$。同时，如下图所示，每级索引中都是两个节点抽出一个节点作为上一级索引的结点，因此每一层最多遍历 3 个节点，故时间复杂度 $=O(3logn)$，省略常数得 $O(logn)$。

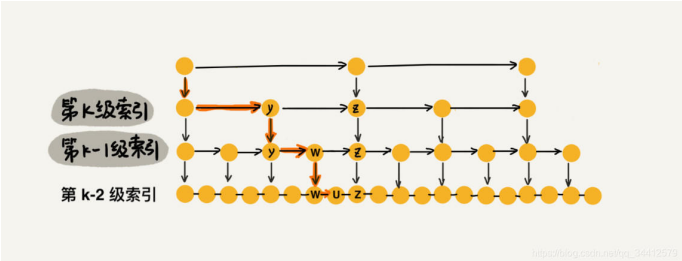

图 2: 每层最多遍历的节点 [2]

4.2 插入操作

插入操作需要在查找到元素应插入的位置后进行，因此插入操作的时间复杂度也是 $O(logn)$。

4.3 删除操作

删除操作包括两个部分，查找和删除。查找时间复杂度如上述为 $O(logn)$。而删除部分不仅是删除元素，还需要更新各层中指向该节点的前驱指针，因此时间复杂度也是 $O(logn)$。因此删除操作的总时间复杂度应该是 $O(2logn)$, 省略常数，时间复杂度为 O(log n)。

5 实验体会

在本次实验中，我通过实现单链表和跳表的基本操作，深入理解了两种数据结构在时间效率上的本质差异。当节点数量达到万级时，单链表的插入和删除耗时显著上升，例如删除 15000 个节点需要 0.093 秒，而跳表始终稳定在接近 0 秒的水平。这种直观的对比让我意识到，理论上的时间复杂度差异。

通过查阅资料，我对跳表的工作原理有了更深入的理解。跳表通过建立多层索引结构，以空间换时间，使得查找、插入和删除操作的时间复杂度大幅降低。这种空间换时间的策略看似简单，却需要精心设计索引概率与平衡逻辑。

实验的另一收获来自数据可视化。在实验过程中，我尝试使用 Python 进行数据可视化，将实验结果以图表的形式呈现，这不仅让数据对比更加直观，也让我掌握了新的技能。

此次实践让我认识到，数据结构的价值不仅在于实现功能，更在于适应场景。单链表虽简单易用，但面对海量数据时可能成为性能瓶颈；跳表的实现复杂度虽高，却能通过结构优化突破线性时间的限制。

参 考 文 献

[1] nanshaws. c 语言实现跳表（skiplist）, 2024. Accessed: 2025-04-10.

[2] fanru bigdata. 一文彻底搞懂跳表的各种时间复杂度、适用场景以及实现原理, 2019. Accessed: 2025-04-10.