

“数据结构”

课程设计报告

**设计题目**  SkipList的实现

**姓 名**

**学 号**

**专 业**  计算机科学与技术

**班 级**

**完成日期**

课程设计成绩评定表

|  |  |  |  |
| --- | --- | --- | --- |
| **设计题目** | SkipList的实现 | **成绩** |  |
| **内容简介** | 本设计题目的基本内容是构造并实现Skip List 的ADT，并能对其维护动态数据集合的效率进行一定的实验验证。  同时，本题要求实现SkipList初始化、插入、搜索、删除等操作的可视化，作者使用了SFML库来实现可视化。  另外，通过比较多层级(level>1)的SkipList与链表比较，来验证SkipList的高效性。 | | |
| **评语** | **教师签名：** | | |

**（一） 需求和规格说明**

本设计题目的基本内容是构造并实现Skip List 的ADT，并能对其维护动态数据集合的效率进行一定的实验验证。

（2）课程设计目的

认识并应用Skip List 数据结构，体会线性表结构的变形形式。

（3）基本要求

①ADT 中应包括初始化、查找、插入、删除等基本操作。

②分析各基本操作的时间复杂性。

③针对实现Skip List 上基本操作的动态演示（图形演示）。

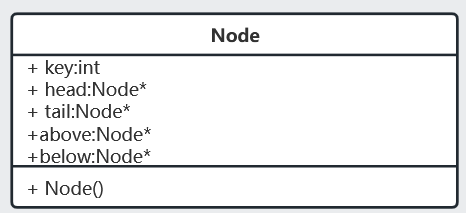
④能对Skip List 维护动态数据集合的效率进行实验验证，获得一定量的实验数据，如给定随机产生1000 个数据并将其初始化为严格Skip List，在此基础上进行一些列插入、删除、查找操作（操作序列也可以随机生成），获得各种操作的平均时间（或统计其基本操作个数）；获得各操作执行时间的变化情况，应该是越来越大，当大到一定程度后应该进行适当的整理，需设计相应的整理算法，并从数量上确定何时较为合适；能和其他简单线性数据结构，如排序数组上的折半查找进行各类操作效率上的数量对比。

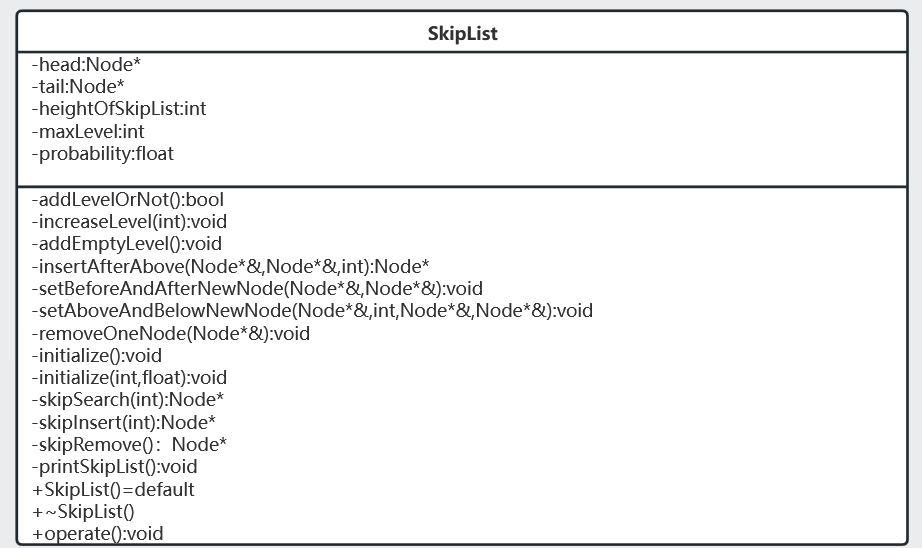
**（二） 设计**

根据上述需求，设计出结点结构Node和SkipList类，在SkipList类中实现题目要求的插入、删除、查找等算法。题目要求的演示使用SFML图形库完成。

SkipList中的各种操作通过操作接口operate()来调用，主函数中只需要实例化一个SkipList并调用其operate()接口即可。

**系统类图**

****

****

**属性和方法定义**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| Node | 属性 | int | key | 结点值 |
| Node\* | above | 此结点上层结点 |
| Node\* | below | 此节点下层结点 |
| Node\* | next | 此节点后继结点 |
| Node\* | prev | 此节点前驱结点 |
| 方法 |  | Node() | 结点构造函数 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| SkipList | 属性 | Node\* | head | SkipList头结点 |
| Node\* | tail | SkipList尾结点 |
| int | heightOfSkipList | 当前SkipList层数 |
| int | maxLevel | SkipList最大层数 |
| float | Probability | 生成一层的可能性 |
| 方法 | bool | addLevelOrNot() | 根据概率以及当前层数判断是否向上添加一层 |
| void | increaseLevel(int) | 在原来层的基础上添加一层 |
| void | addEmptyLevel() | 添加只有头和尾的空层 |
| Node\* | insertAfterAbove(Node\*&Node\*&,int) | 主要的插入函数,在结点之后插入一个新结点 |
| Node\* | setAboveAndBelowNewNode(Node \*&, int, Node \*&, Node \*&) | 设置新结点的上下指针 |
| void | removeOneNode(Node\*&) | 删除参数中的结点 |
| void | initialize | 初始化SkipList |
| void | initialize(int,float) | 含参初始化最大层次和增加一层的概率 |
| Node\* | skipSearch(int) | 查找值为key的结点 |
| Node\* | skipInsert(int) | 插入值为key的结点 |
| Node\* | skipRemove(int) | 删除值为key的结点 |
| void | printSkipList() | 打印SkipList |
| / | SkipList() | 构造函数，=default |
| / | ~SkipList() | 析构函数，释放内存 |
| void | operate() | 各种操作的接口 |
| void | increaseLevel(int ) | 向上添加一层 |

**（三） 用户手册**

为了满足题目的需求，给出了两个SkipList版本，一个是用于可视化的，另一个不具有可视化功能，用于测试时间效率。

可视化版本: 用户在控制台中根据提示输入信息，在window中即可看到相应的可视化结果。

非可视化版本: 用户在控制台依据提示输入，在控制台可以得到结果。

**（四） 调试及测试**

经典的SkipList添加结点时，在原始链表上添加多层索引链表，同时，每一层的结点都有指向下一层的指针，这些指针构成了层级结构，搜索时可以跳过一定数量的结点，从而快速接近目标结点。由于SkipList是有序的，所以搜索时可以按照二分查找的思想进行，故其搜索的平均时间复杂度是O(log N)级别的。

SkipList的插入、删除结点均以搜索结点为基础，故其平均时间复杂度也为O(log N)。

最坏情况是除了SkipList的底层，其余每一层都没有随机生成索引节点，此时SkipList就变成了单链表。故最坏时间复杂度为O(n).

SkipList的运行效率取决于SkipList支持的最大层数和在某个结点处向上增加一层的概率。下面是通过对拥有1000个元素的SkipList进行1000000次插入、搜索或删除等随机操作进行统计，得出来的结果。

由于SkipList层数限制在1-3时，效率与单链表差不多，所以直接从maxLevel=5

maxLevel=1时，SkipList即为单链表

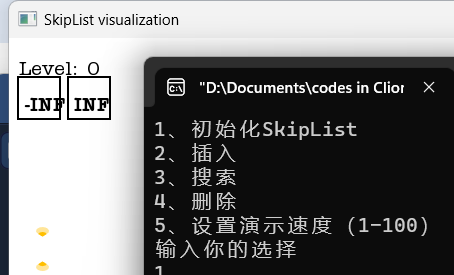
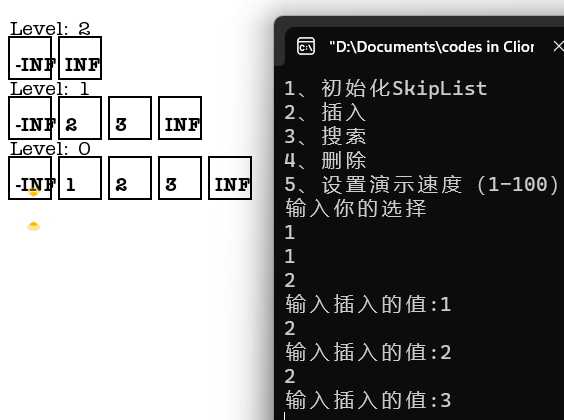
开始测试

|  |  |  |
| --- | --- | --- |
| **maxLevel** | **probability** | **time** |
| **1** | **/** | **5.5E-03** |
| **5** | **0.2 0.5 0.8** | **5E-03 5.1E-03 5.4E-03** |
| **9** | **0.2 0.5 0.8** | **5.4E-03 5E-03 5.2E-03** |
| **13** | **0.2 0.5 0.8** | **5.1E-03 5.2E-03 5.1E-03** |
| **17** | **0.2 0.5 0.8** | **5.7E-03 5.1E-03 5E-03** |
| **21** | **0.2 0.5 0.8** | **5.1E-03 5.2E-03 5.1E-03** |
| **25** | **0.2 0.5 0.8** | **5.4E-03 5.3E-03 5.2E-03** |
| **29** | **0.2 0.5 0.8** | **5.1E-03 5.6E-03 5.3E-03** |

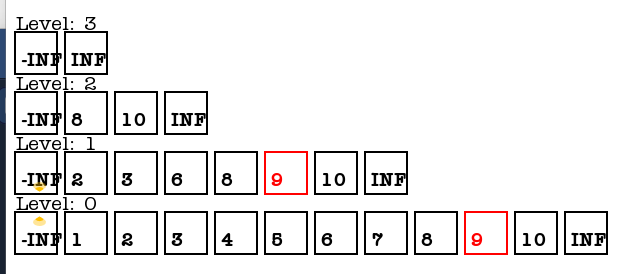
**综上可以大致看出，在maxLevel为9 – 21 之间，概率为0.5-0.8之间效率最高。SkipList的效率没有精确值，要根据实际应用场景设置probability和maxLevel。**

**（五） 运行实例：**

初始化 插入

搜索

删除需要当面演示。

**（六）进一步改进**

(1)SkipList的可视化实现中插入的过程可以做出一个简单的动画，这样演示的更清楚。还可以采用其他Ui库制作Ui，如imgui，使图形化完成度更高。

**（七）心得体会**

依据程序做图形化是实验的一个难点，因为之前没做过图形化，所以在学习图形化上花费了很长时间。同时，因为没尝试过图形化，不知道图形化如何做出演示的视觉效果，尝试了不同的GUI。最开始是Qt，最后才确定用SFML做，这个过程浪费了很长时间。

SkipList数据结构使得链表搜索、插入、删除等操作的时间复杂度从O(n)变为O(log n),当链表需要频繁执行搜索、插入、删除等操作时，可以将链表转化为SkipList，并根据自己的数据量确定maxLevel和probability,这将大大降低时间成本。

需求催生技术，通过这次实验我觉得有必要自己学习一些GUI技术了。使用ui界面将代码转化为图像，这样使得代码的逻辑可视化，特别简明。本次实验没有使用到更加优秀的GUI(Qt、imgui)等，有些遗憾，之后一定要学习并掌握一些GUI技术。

**（八）对课程设计的建议**

**（九）附录⎯⎯源程序**

**SkipList.h**

//

// Created by archi on 2023/6/21.

//

#ifndef SKIPLIST1\_1\_SKIPLIST\_H

#define SKIPLIST1\_1\_SKIPLIST\_H

#include <iostream>

#include <chrono>

#include <bits/stdc++.h>

using namespace std;

struct Node {

public:

Node\* above;

Node\* below;

Node\* next;

Node\* prev;

int key;

Node(int key) {

this->key = key;

this->above = nullptr;

this->below = nullptr;

this->next = nullptr;

this->prev = nullptr;

}

};

const int NEG\_INF=INT\_MIN, POS\_INF=INT\_MAX;

class SkipList {

private:

Node\*head,\*tail;//分别表示跳表表头和表尾

int heightOfSkipList=0;//当前跳表的高度

int maxLevel=4;//最大层级,默认为4

float probability=0.5f;//生成随机层次的概率 默认为0.5

//生成随机初始数据

void efficiencyVerification(int operationNum, int MaxLevel, float Probability);

// \*\* 搜索、插入、删除等所需函数\*\*//

bool addLevelOrNot(int currentLevel) const;//是否添加一层，概率

void increaseLevel(int level);//检查当前高度是否足以向上一层添加元素，不够则先添加一层空层，再向上添加一层

void addEmptyLevel();//添加空层，只有头结点和尾结点

Node\*insertAfterAbove(Node \*&position, Node \*&q, int key);//

void setAboveAndBelowNewNode(Node \*&position, int key, Node \*&newNode, Node \*&nodeBelowNewNode);

void removeOneNode(Node\*&nodeToRemove);

//\*\* 主要函数 \*\* //

void initialize();//初始化

void initialize(int maxLevel,float probability);//含参初始化

Node\*skipSearch(int key);//搜索

Node\*skipInsert(int key);//插入

Node\*skipRemove(int key);//删除

void printSkipList();//打印

public:

SkipList()=default;

~SkipList();

void operate();//操作接口

};

#endif //SKIPLIST1\_1\_SKIPLIST\_H

**SkipList.cpp**

#include "SkipList.h"

Node \*SkipList::skipSearch(int key) {

Node\*current=head;//从头开始查找

//从顶向下查找

while(current->below!= nullptr){

//如果一层查找完了，就向下一层查找

current=current->below;

//找到的位置的下一个位置值要比key大

while(key>=current->next->key){

current=current->next;

}

}

//返回找到的位置

//如果SkipList中不存在key,则返回最后一个小于key的位置 用于插入

//如果SkipList中存在key,则返回key所在位置

return current;

}

Node\* SkipList::skipInsert(int key) {

Node \*position = skipSearch(key);//找到插入的位置之前的结点，或是已经存在的值为key的结点

Node \*q;

int level = -1;

if (position->key == key) {

return position;

}

int currentLevel=0;//当前skipList结点的高度

//用do while是因为默认插入结点至少要在最底层插入一个

do {

level++;//增加一层层次+1

currentLevel++;//增加一层高度+1

//检查当前SkipList高度是否足以向上一层添加元素，不够先在上一层添加一层空层

increaseLevel(level);

q = position;//暂时保存当前位置（要插入元素的位置）

//这里是找到要插入新的一层的结点的位置

while (position->above == nullptr) {

position = position->prev;//向前查找插入的位置，也就是查找上一层最后一个结点位置

}

position=position->above;//position赋值为上面找到的位置

//在position位置插入新结点，也就是在原position位置(p指向的位置)的上面一层插入结点

q = insertAfterAbove(position, q, key);

} while (addLevelOrNot(currentLevel));//当前层已经插入，通过抛硬币决定是否继续向上添加层

//循环的逻辑是抛硬币（用概率）决定是否往上增加一层

return q;

}

bool SkipList::addLevelOrNot(int currentLevel) const {

//高度不能操作最大高度

if(currentLevel ==maxLevel-1)

return false;

random\_device rd;

mt19937 gen(rd());

uniform\_real\_distribution<float> dis(0.0f, 1.0f);

//如果随机生成的数小于probability,则添加新的一层

if(dis(gen)<probability)

return true;

else

return false;

}

void SkipList::increaseLevel(int level) {

if(level >= heightOfSkipList){

heightOfSkipList++;

addEmptyLevel();

}

}

//添加一层空层

void SkipList::addEmptyLevel() {

//生成新的空层的头结点为尾结点

Node\*newHeadNode = new Node(NEG\_INF);

Node\*newTailNode = new Node(POS\_INF);

//设置新结点的头、尾、上下层级

newHeadNode->next = newTailNode;

newHeadNode->below = head;

newTailNode->prev = newHeadNode;

newTailNode->below = tail;

//将新节点加入到原头、尾结点的上一层

head->above=newHeadNode;

tail->above=newTailNode;

//设置新的头尾结点为最上层结点

head = newHeadNode;

tail = newTailNode;

}

/\*

\* @brief 在q位置后插入结点新结点 同时处理新结点的上下结点问题

\* @position SkipList中新结点上一层

\*/

Node \*SkipList::insertAfterAbove(Node \*&position, Node \*&q, int key) {

Node\*newNode=new Node(key);

//将新结点插入到结点q之后

newNode->next=q->next;

newNode->prev=q;

q->next->prev=newNode;

q->next=newNode;

//q是position的下面一层的元素，需要找到插入位置position

Node\*nodeBelowNewNode=position->below->below;

setAboveAndBelowNewNode(position, key, newNode, nodeBelowNewNode);

return newNode;

}

/\*

\* @brief 设置新结点上下指针

\* @position 新结点的上层位置

\* @key 新结点值

\* @newNode 新结点指针

\* @nodeBelowNewNode 新结点的下层结点指针

\* @return void

\*/

void SkipList::setAboveAndBelowNewNode(Node \*&position, int key, Node \*&newNode, Node \*&nodeBelowNewNode) {

if(nodeBelowNewNode != nullptr){

//找到新结点下一层的对应结点

while(true){

if(nodeBelowNewNode->next->key != key){

nodeBelowNewNode=nodeBelowNewNode->next;

}

else{

break;

}

}

//设置新结点的下指针(也就是设置重复结点的上下指针)

newNode->below=nodeBelowNewNode->next;

nodeBelowNewNode->next->above=newNode;

}

//链接重复Key结点的上下位置

//例子: 第一次插入的时候在第二层未插入3，而第二次插入在第二层插入了3

//这样设置了新结点的上下左右结点之后，旧的值Key结点也能链接到上一层的新结点。

if(position!= nullptr){

if(position->next->key==key){

newNode->above=position->next;

}

}

}

/\*

\* @brief 删除值为key的结点

\* @key 要删除的结点值

\* @return Node\* 要删除的结点指针

\*/

Node\*SkipList::skipRemove(int key) {

Node \*nodeToRemove = skipSearch(key);

if (nodeToRemove->key != key) {

return nullptr;

}

removeOneNode(nodeToRemove);

while(nodeToRemove!= nullptr){

removeOneNode(nodeToRemove);

if(nodeToRemove->above!= nullptr){

nodeToRemove=nodeToRemove->above;

}

else{

break;

}

}

return nodeToRemove;

}

/\*

\* @brief 删除一个结点

\* @param 要删除的结点

\* @return void

\*/

void SkipList::removeOneNode(Node \*&nodeToRemove) {

Node \*afterNodeToRemove = nodeToRemove->next;

Node \*beforeNodeToRemove = nodeToRemove->prev;

beforeNodeToRemove->next = afterNodeToRemove;

afterNodeToRemove->prev = beforeNodeToRemove;

}

/\*

\* @brief 析构函数，删除操作过程中产生的结点

\* @param 无

\*

\*/

SkipList::~SkipList() {

queue<Node\*> nodeQueue;

nodeQueue.push(head);

//将所有层的头结点都放入队列

while(!head->above){

nodeQueue.push(head->above);

}

while(!nodeQueue.empty()){

Node\*current=nodeQueue.front();

Node\*next=current->next;

while(next){

delete current;

current=next;

next=next->next;

}

delete current;

}

}

/\*

\* @brief 初始化SkipList

\* @param 空

\* @return 空

\*/

void SkipList::initialize() {

this->heightOfSkipList=0;

head = new Node(NEG\_INF); // 表头初始化为负无穷大

tail = new Node(POS\_INF); // 表尾初始化为正无穷大

head->next = tail;

tail->prev = head;

}

/\*

\* @brief 初始化SkipList的最大层次和随机生成一层的可能性

\* @param MaxLevel SkipList最大层次

\* @param Probability SkipList随机生成一层的概率

\*/

void SkipList::initialize(int MaxLevel, float Probability) {

initialize();

this->maxLevel = MaxLevel;

this->probability = Probability;

}

/\*

\* @brief 显示操作菜单

\* @param void

\* @return void

\*/

void menu(){

cout<<"1、初始化SkipList"<<endl;

cout<<"2、插入"<<endl;

cout<<"3、搜索"<<endl;

cout<<"4、删除"<<endl;

cout<<"5、打印当前SkipList(非可视化用)"<<endl;

cout<<"6、计算时间效率并与排序数组二分查找对比"<<endl;

cout<<"输入你的选择"<<endl;

}

/\*

\* @brief 打印显示SkipList

\* @param void

\* @return void

\*/

void SkipList::printSkipList() {

cout << "\n当前SkipList如下:";

Node\* starting = head;

Node\* highestLevel = starting;

int level = heightOfSkipList;

while (highestLevel != nullptr) {

cout << "\nLevel: " << level << "\n";

while (starting != nullptr) {

cout << starting->key;

if (starting->next != nullptr) {

cout << " : ";

}

starting = starting->next;

}

highestLevel = highestLevel->below;

starting = highestLevel;

level--;

}

cout << endl;

}

/\*

\* @brief 非图形化操作接口

\* @param void

\* @return void

\*/

void SkipList::operate() {

menu();

bool isInitialized=false;

int select;

while(true){

cin>>select;

switch (select) {

case 1:{

cout<<"输入最大层级和生成层级的概率:";

int MaxLevel;

float Probability;

cin >> MaxLevel >> Probability;

initialize(MaxLevel, Probability);

printSkipList();

isInitialized= true;

cout<<"初始化成功!"<<endl;

break;

}

case 2:{

if(!isInitialized){

cout<<"请先初始化!"<<endl;

break;

}

cout<<"输入插入的值:";

int value;

cin>>value;

skipInsert(value);

printSkipList();

cout<<"插入成功!"<<endl;

break;

}

case 3:{

if(!isInitialized){

cout<<"请先初始化!"<<endl;

break;

}

cout<<"输入搜索的值:"<<endl;

int value;

cin>>value;

if(skipSearch(value)->key==value){

cout<<"该值存在"<<endl;

}

else{

cout<<"该值不存在"<<endl;

}

break;

}

case 4:{

if(!isInitialized){

cout<<"请先初始化!"<<endl;

break;

}

cout<<"输入删除的值:";

int value;

cin>>value;

skipRemove(value);

printSkipList();

break;

}

case 5:{

if(!isInitialized){

cout<<"请先初始化!"<<endl;

break;

}

printSkipList();

break;

}

case 6:{

cout<<"输入最大层级、生成层级的概率和随机操作数量:";

int opNum;

int MaxLevel;

float Probability;

cin >> MaxLevel >> Probability>>opNum;

efficiencyVerification(opNum,MaxLevel,Probability);

break;

}

default:{

cout<<"输入错误,请重新输入:";

break;

}

}

}

}

void SkipList::efficiencyVerification(int operationNum, int MaxLevel, float Probability) {

//以给定的最大层数和生成层次的可能性初始化

initialize(MaxLevel,Probability);

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dis(1, 10000); // 随机生成1到10000的整数

vector<int>\*dataSeries=new vector<int>(operationNum);

//生成1000个数据的SkipList

for (int i = 0; i < 1000; ++i) {

skipInsert(dis(gen));

}

//为各种操作提供operationNum个数据

for(int i=0;i<operationNum;i++){

dataSeries->emplace\_back(dis(gen));

}

//生成3种随机操作

vector<int>\* operations=new vector<int>(operationNum);

uniform\_int\_distribution<int>dis2(1,3);

for(int i=0;i<operationNum;i++){

operations->emplace\_back(dis2(gen));

}

// 开始计时

chrono::steady\_clock::time\_point start = std::chrono::steady\_clock::now();

//执行操作

int k=0;

for(int i=0;i<operationNum;i++){

if((\*operations)[i]==1){//插入

skipInsert((\*dataSeries)[i]);

}

else if((\*operations)[i]==2){//搜索

skipSearch((\*dataSeries)[i]);

}

else if((\*operations)[i]==3){

skipRemove((\*dataSeries)[i]);

}

k++;

}

// 结束计时

chrono::steady\_clock::time\_point end = chrono::steady\_clock::now();

// 计算时间间隔

chrono::duration<double> duration = chrono::duration\_cast<chrono::duration<double>>(end - start);

// 输出执行时间

cout << "执行时间: " << duration.count() << " seconds" << endl;

delete operations;

delete dataSeries;

}

**主函数**

#include "SkipList.h"

int main() {

SkipList skipList;

skipList.operate();

system("pause");

return 0;

}