

# 数据结构课程设计报告

设计题目：查找算法的教学演示

学生姓名：江龙

专 业：计算机科学与技术

班 级：计算机科学与技术一班

学 号：2019214658

指导教师：李培培

完成日期：2020年6月15日

# （一）需求和规格说明

**问题描述：**实现多种查找算法(顺序表、树表、散列表三种数据类型不小于

2类)，并进行对比分析。用户输入查找数据，界面动态演示基于查找过程的图形化显示。

**编程任务：**

1. 将图形化界面的基本布局设计好，根据选择的数据类型数目进行适当调整
2. 选取并完成所选取的查找算法，即在底层完成查找算法的代码逻辑，为后续的图形化做准备，这里我选取了5种算法，分别是：二分查找，索引表查找，平衡二叉树，B树，散列表的拉链法。
3. 根据QT的paintEvent事件的特性，完成底层算法的实时动态展示。
4. 完成4大按钮的功能，分别是：开始演示，暂停演示，终止演示，重新初始化。
5. 对用户可能出现的不规范操作进行处理准备，防止程序的崩溃。

# （二）设计

## 1．设计思想

首先此次设计重点除了对于不同结构种类的查找算法实现，还在于如何体现不同查找算法的查找过程。这里我选取的是QT来做可视化处理，利用重写QT的paint事件，该事件会在一开始就被自动调用，此后每一次更新需要自己手动更新该事件。利用此特性，我决定在某一块白板组件上展现算法的动态，将应的算法数据处理完后，在查找过程中同步调用更新函数，这样便达成了算法查找的同步图形化展示。

## 2. 设计表示

1. 部分共同成员数据

|  |  |  |
| --- | --- | --- |
| **成员数据类型** | **成员名** | **描述** |
| Int | length | 随机生成数据的长度 |
| Int | target | 查找的目标值 |
| int | currentIndex | 当前搜索点的值 |
| int \* | nums | 随机生成的数据 |
| int | count | 查询次数 |

1. 图形化界面

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| MainWindow | 函数 | void | generate () | 生成数据函数 |
| void | result () | 查询结果消息弹窗 |
| void | start(Search\* currentSearch,BtnGroup\* btngroup,int page) | 开始按钮功能的封装 |
| void | pause(Search\* currentSearch,BtnGroup\* btngroup) | 暂停按钮功能封装 |
| void | terminate(Search\* currentSearch,BtnGroup\* btngroup) | 终止按钮功能封装 |
| void | reinitialize(Search\* currentSearch,BtnGroup\* btngroup,int page) | 重新初始化功能封装 |
| void | initChcek() | 切换页面时初始化所有算法的选择 |
| void | setTarget() | 获取用户输入的查找值 |
| 数据 | bool | isProper | 查找目标是否合法 |
| bool | isready | 是否已生成数据 |
| int | randMax | 随机数据的范围最大值 |
| int | randMin | 随机数据的范围最小值 |
| int | max | 随机数据中的最小值 |
| int | min | 随机数据中的最大值 |
| int | randLengthMax | 随机数据长度的范围最大值 |
| int | randLengthMin | 随机数据长度的范围最小值 |

1. Search类：所有算法类的父类，用于存放一些所有算法类都会用到的封装函数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| Search | 函数 | void | repaint () | 更新绘画 |
| void | Sleep(int msec) | 自定义延时器 |
| 数据 | bool | ispause | 是否点击暂停按钮 |
| bool | isterminate | 是否点击终止按钮 |

1. HashSearch类：拉链法查找

|  |  |  |  |
| --- | --- | --- | --- |
| **结构体名称** | **成员数据类型** | **成员名** | **描述** |
| node | Int | data | 存储的单词 |
| node\* | next | 下一个结点 |
| bool | isSearched | 是否被搜索过 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| HashSearch | 函数 | void | paintEvent(QPaintEvent \*) | 绘画事件 |
| void | hashSearch() | 散列表查找对数据的处理 |
| void | getsignal(bool isHashSearch,int length,int target,int \*nums) | 获取图形化界面点击的信号，并对其解析处理 |
| void | searchBucket(bucket\* &t,int index) | 从buckets中找到指定的节点 |
| 数据 | bool | isfirst | 是否是第一次绘图 |
| bool | ishash | 是否选择了hash查找 |
| bucket\* | firstBucket | 指向第一个bucket的指针 |

1. ListSearch类：二分查找，索引表查找

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| ListSearch | 函数 | void | paintEvent(QPaintEvent \*) | 绘画事件 |
| void | binarySearch () | 二分查找对数据的处理 |
| void | getsignal(bool binary,bool indexed,int length,int target,int \*nums) | 获取图形化界面点击的信号，并对其解析处理 |
| void | indexedSearch() | 索引表查找对数据的处理 |
| 数据 | bool | isBinary | 是否选择的是二分查找 |
| bool | isIndexed | 是否选择的是索引查找 |
| int \* | indexList | 每个索引项其中的最大值 |
| int \* | starts | 每个索引项的起始索引值 |
| int \* | lengths | 每个索引项的长度 |

1. TreeSearch类：平衡树查找，B树查找

|  |  |  |  |
| --- | --- | --- | --- |
| **结构体名称** | **成员数据类型** | **成员名** | **描述** |
| bnode | Int | data | 存储数据 |
| bnode\* | leftC | 左儿子节点 |
| bnode\* | rightC | 右儿子节点 |
| int | level | 节点的层数 |
| int | bf | 平衡因子 |
| Bnode | int | keynum | 结点关键字个数 |
| int | key | 关键字数组，key[0]不使用 |
| Bnode \* | parent | 双亲结点指针 |
| Bnode \*[] | ptr | 孩子结点指针数组 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **类名** | **成员类别** | **类型** | **成员名** | **描述** |
| TreeSearch | 函数 | void | paintEvent () | 绘图事件 |
| void | balancedTreeSearch () | 平衡二叉树查找数据处理 |
| void | BTreeSearch () | B树查找数据处理 |
| void | getsignal(bool isBTree,bool isbalancedTree,int length,int target,int \*nums) | 接收信号，决定演示哪种算法 |
| int | SearchNode(bnode\* &target,int data) | 查找到data数据的父节点 |
| void | paint(int x,int y,bnode\* n) | 给当前节点画左右两个儿子节点的图 |
| void | Bpaint(int x,int y,int px,int py,int index,Bnode\* b) | 给B树的当前节点画儿子节点的图 |
| int | getHeight(bnode\* t) | 获取以当前节点为根的二叉树的高度 |
| void | setBF(bnode\* t) | 更新整棵树每个节点的平衡因子 |
| void | getBf(bnode\* &target) | 获取平衡因子绝对值超过1的最高层次的那个节点（注意，这里我没有采用找层次最低的平衡点，因为有个BUG实在无法定位修复，所以采取了消耗更大的找层次最高的平衡点） |
| void | correct(bnode\* begin,bnode\* target) | 判断当前错误属于什么型，并执行相应的修复函数 |
| void | getLevel(bnode\* t) | 更新所有节点的层次数 |
| void | setIndex(Bnode\* &b,int num) | 给新关键字找到合适的位置插入 |
| void | reset(Bnode\* &b) | 重置调整B树 |
| void | SplitBnode(Bnode \*&p,Bnode \*&q) | 将结点p分裂成两个结点,前一半保留,后一半移入结点q |
| void | dfs(double x,double y,double px,double py,int index,int level,Bnode\* b) | 深度优先遍历B树,并逐层调用绘画 |
| void | getMaxLevel(Bnode\* b) | 获取B树的最大深度 |
| void | LL(bnode\* t) | 平衡二叉树LL型问题 |
| void | RR(bnode\* t) | 平衡二叉树RR型问题 |
| void | LR(bnode\* t) | 平衡二叉树LR型问题 |
| void | RL(bnode\* t) | 平衡二叉树RL型问题 |
| 数据 | bool | isbalancedTree | 是否选择平衡二叉树按钮 |
| bool | isBTree | 是否选择B树按钮 |
| int | r | 圆的半径 |
| double | angle | 平衡二叉树儿子节点的初始偏转角度 |
| double | addangel | 每一层改变的偏转角度量 |
| QQueue<int> | q | 存储每个节点的x,y坐标 |
| QQueue<bnode\*> | node | 存储节点 |
| QQueue<bnode\*> | node\_2 | 存储节点，用于广度优先遍历 |
| int | w | B树关键字节点的宽度 |
| int | h | B树关键字节点的高度 |
| int | maxLevel | B树的最大深度 |

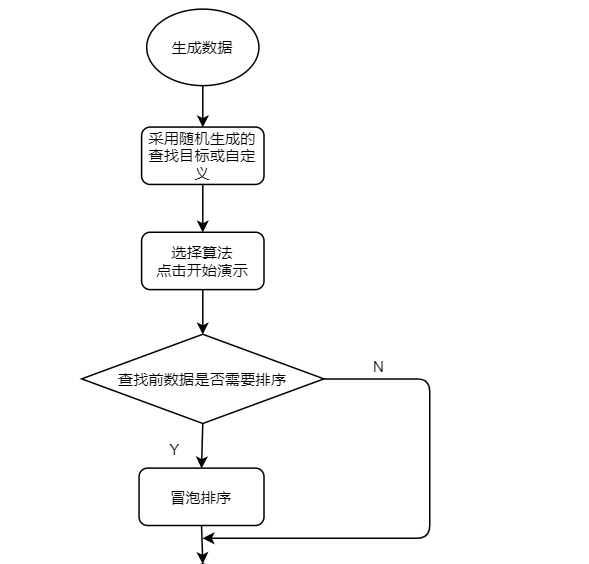
## 3. 核心算法

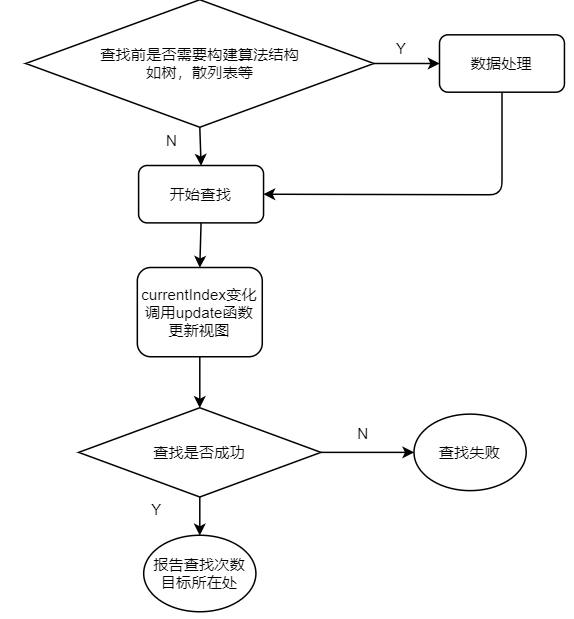
此次课设的核心算法主要是5个不同的查找算法和各个不同的算法类重写的paint绘图事件。

首先，用户在点击生成数据按钮后，会触发generate函数根据设定的参数随机生成数据，并记录下数据的最大值，最小值，和查找目标值。当然根据题目的要求，我们在图形界面展示出随机数据的长度，最大值，最小值和查找目标，因为必须兼容用户可以自己设定查找目标，用户可以在展示页面直接对查找目标进行更改。用户选择对应算法后，点击开始演示按钮，触发start函数，该函数根据用户选择的算法和生成的数据，触发不同算法类的getsignal函数。

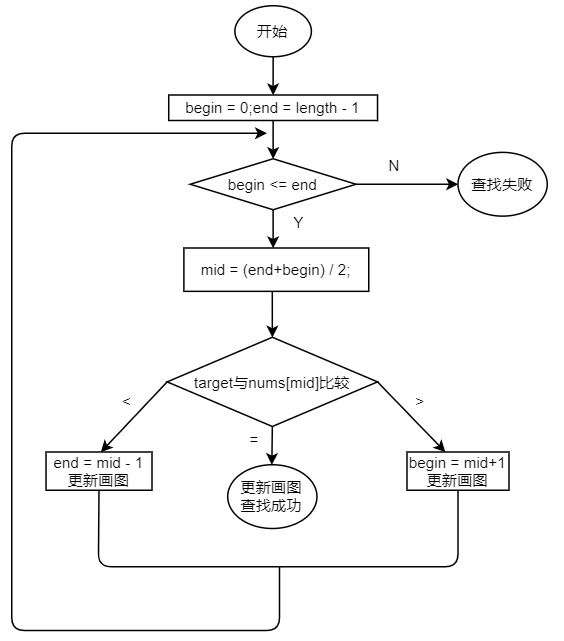
在触发对应的算法之后，因为像二分查找和索引表对数据要求必须是排序完的，所以就需要对数据进行排序处理，这里采用的冒泡排序。5种算法诸如binarySearch, balancedTreeSearch，BtreeSearch，indexedSearch，hashSearch函数便是对各自算法的所需对数据进行相应的处理，比如树类的查找算法就需要重头构建相应的树，都在对应的Search算法中进行数据处理。处理完后即开始查找，不同于普通的算法查找，为了动态图形化显示当前查找的进度，我们通过currentIndex来表示当前查找节点的值，同时只要currentIndex变化，我们就手动调用update函数，经过我们设置的短暂延迟便重新绘画整个图，如此便完成了动态实时展示查找过程的效果。

逻辑实现过程：

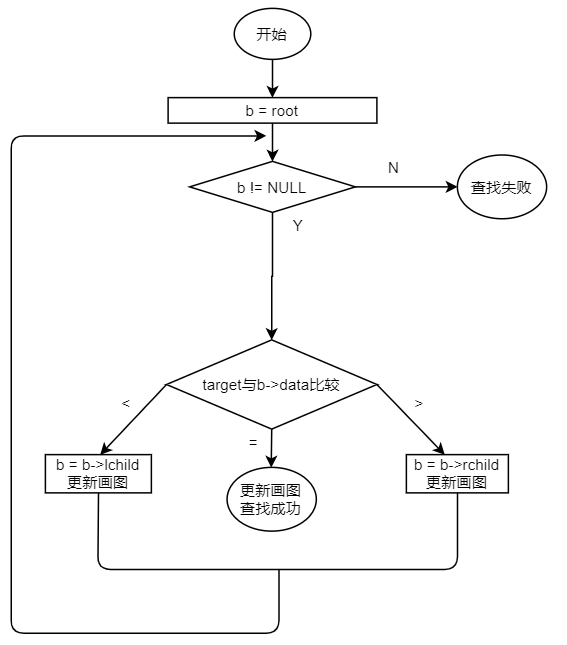




部分查找算法流程图如下所示：



**二分查找算法**



**平衡二叉树**

# （三）用户手册



程序运行将会显示出图形化界面

用户首先需要点击生成数据按钮，底层代码根据事先设定好的参数会随机生成数据，并且将其显示到图形化界面上。为了满足用户自定义查找目标的要求，除了查找目标可以自行设置，其他数据都是readOnly，不可更改。

然后，用户需要选择对应的数据类型，根据所选择的数据类型，左下角的算法会有不同的显示，比如选择树表，那么左下角就会显示平衡二叉树和B树查找，一共5种算法供用户选择。

生成完数据并且选择好对应的算法后，用户可以点击开始演示按钮，根据对应的算法选择，便会在对应的空白处显示不同的动态图形化显示。暂停演示按钮只有在点击开始演示按钮后才可以点击，它会暂停整个查找过程，使动态图形暂停演示。终止演示会直接情况整个图，变化空白状态。重新初始化则会先清空当前演示并自动重新生成数据一次，然后开始根据新的数据立马演示。

# （四）调试及测试

## 1. 测试数据：

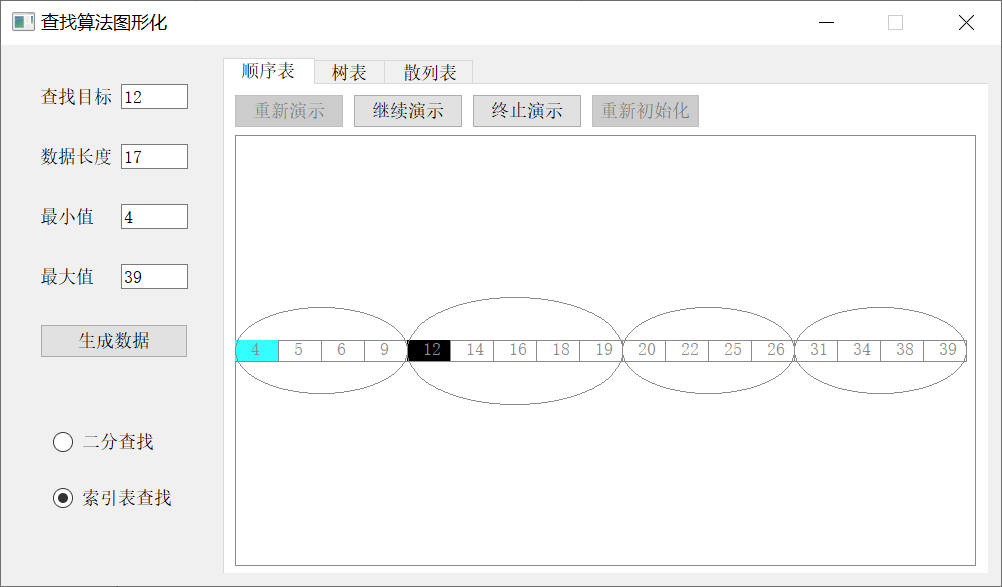


**点击生成数据按钮**

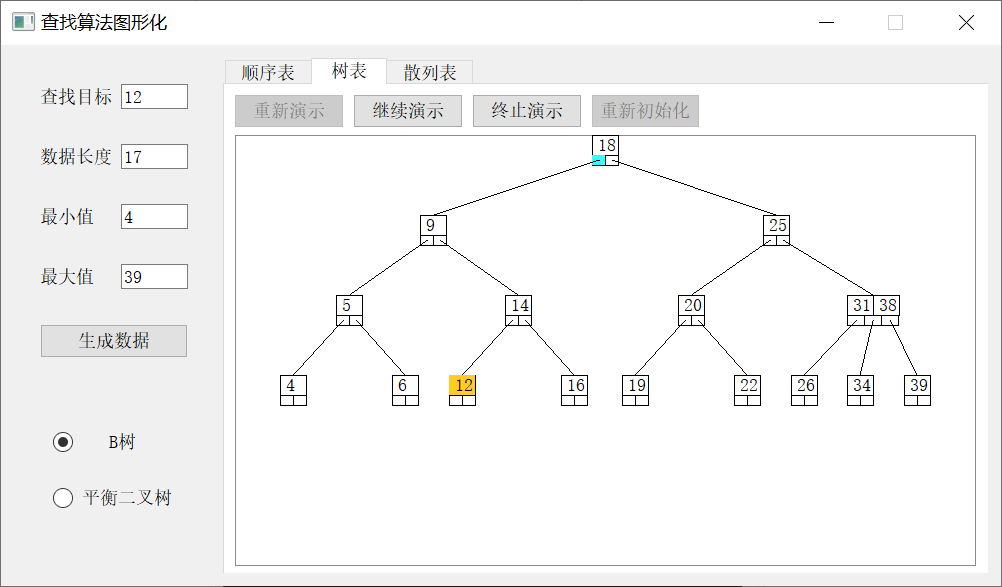
## 2. 演示示例：



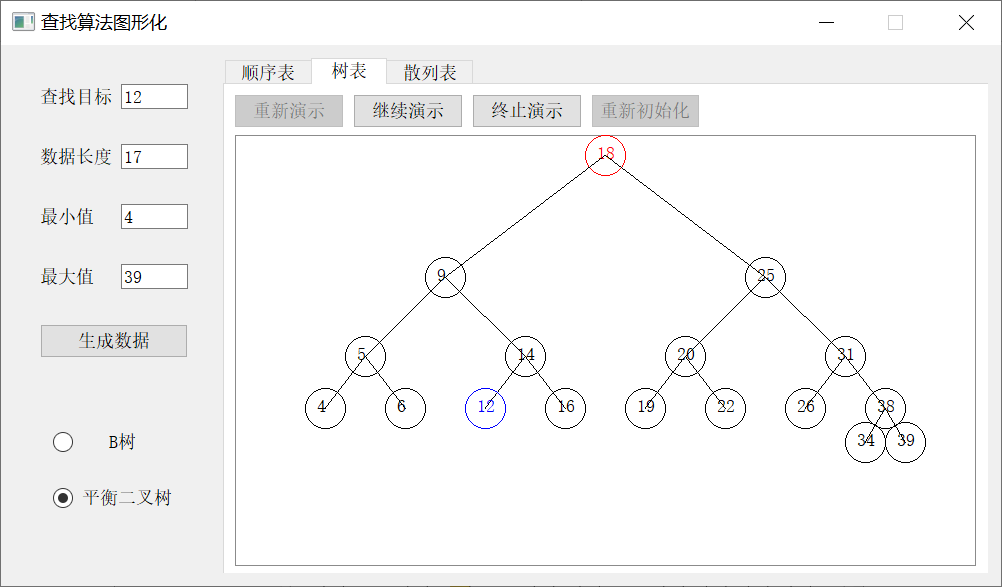
**二分查找**



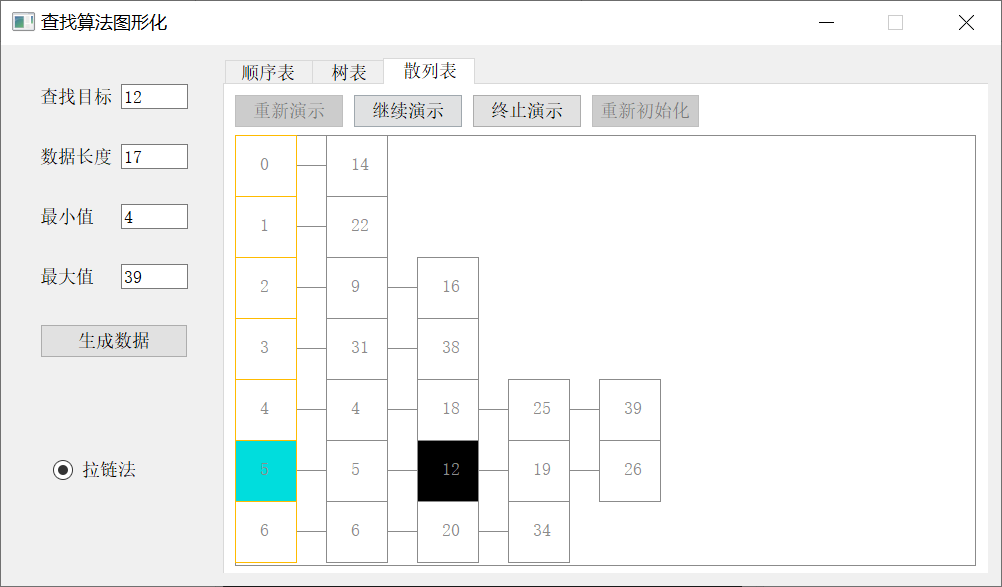
**索引表查找**



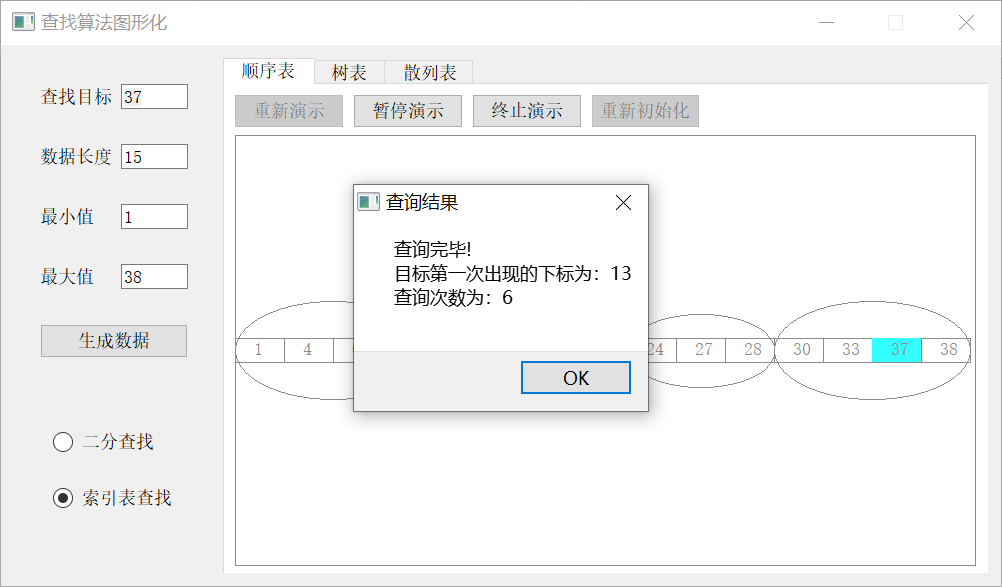
**B树查找**



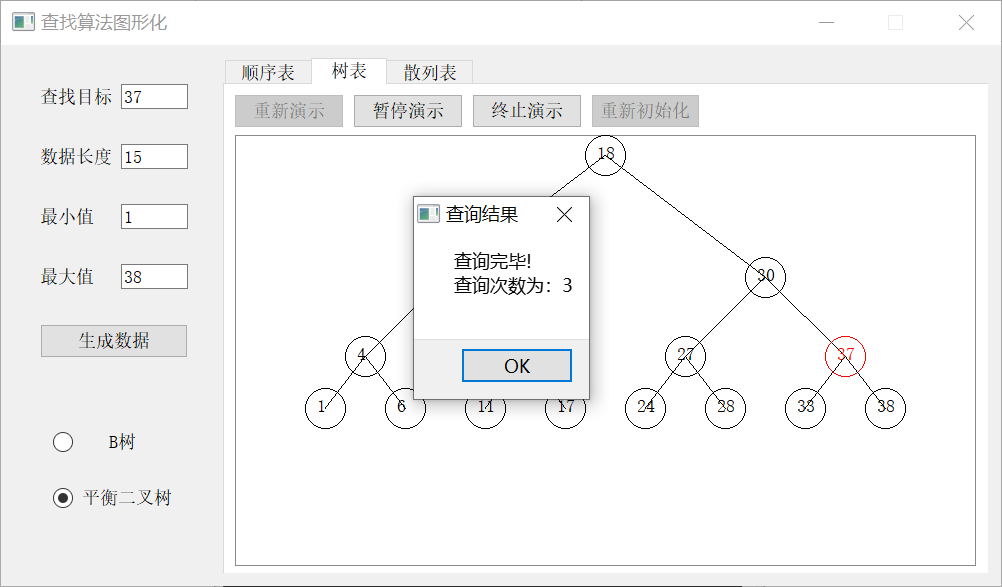
**平衡树查找**



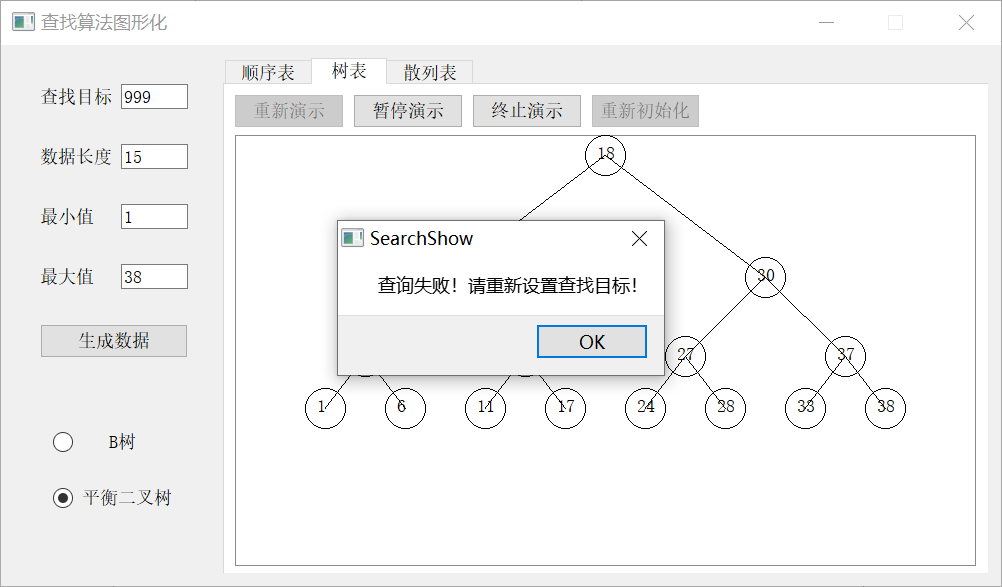
**散列表查找**



**查找成功，以索引表为例**



**查找成功，以平衡二叉树为例**



**查找失败，以平衡二叉树为例**

# (五) 感想

一开始写此课设的时候，自己有很多代码重复性比较大，但是发现的时候已经比较晚了，重新进行封装后，代码量大大减少，并且逻辑更加清晰，但是的确也费了不少力气。

对于此次设计，我最大的遗憾是没有结合多线程去实现，而没有了多线程，在实现一些功能上的确比较麻烦，例如终止演示，或重复点击相同的按钮等，这些操作会造成各种各样的bug，虽然经过我的改进，以单线程也可以完成这些功能，但总体比较遗憾，因为这应该是一次难得的多线程应用机会。

再者，因为一开始图形界面的设计问题，布局过于小，导致展示空间有限，这就使得我不得不限定随机生成数据的数目最大值，不能让用户去随意设置，因为再怎么优化展示的算法，空间始终无法满足太多的数据展示。

不过，此次课设我也收获颇多，首先是对于bfs，dfs的不同应用场景有了更深的理解，其次，自己去查了网上的资料，独立写出了平衡二叉树和B树查找，并且将他们图形化，让我对于这两种的算法的理解更加深刻，同时不再像以往一样对于树结构总有种陌生感，不熟悉各种操作。

并且，通过此次课设，我体会到了合理的函数封装与类的继承关系设计的重要性，如果只是一味的按照逻辑想到哪写到哪，那当代码量逐渐庞大起来时，便会造成自己也看不懂的困境。虽然此次对于自己的设计与封装还有很多很多的不足之处，但是经过此次的尝试，我相信自己能在下一次的程序设计种，设计出更合理的程序结构。

# 附录：

1. 头文件BtnGroup.h：

#ifndef BTNGROUP\_H

#define BTNGROUP\_H

#include <QWidget>

#include "qpushbutton.h"

namespace Ui {

class BtnGroup;

}

class BtnGroup : public QWidget

{

Q\_OBJECT

public:

explicit BtnGroup(QWidget \*parent = nullptr);

~BtnGroup();

void startBegin();

void startEnd();

QPushButton\* start;

QPushButton\* pause;

QPushButton\* terminate;

QPushButton\* reinitialize;

private:

Ui::BtnGroup \*ui;

};

#endif // BTNGROUP\_H

1. BtnGroup类：

#include "btngroup.h"

#include "ui\_btngroup.h"

BtnGroup::BtnGroup(QWidget \*parent) :

QWidget(parent),

ui(new Ui::BtnGroup)

{

ui->setupUi(this);

start = ui->start;

pause = ui->pause;

terminate = ui->terminate;

reinitialize = ui->reinitialize;

}

void BtnGroup::startBegin(){

ui->pause->setText("暂停演示");

ui->start->setText("重新演示");

ui->reinitialize->setEnabled(false);

ui->start->setEnabled(false);

ui->pause->setEnabled(true);

}

void BtnGroup::startEnd(){

ui->start->setEnabled(true);

ui->pause->setEnabled(false);

ui->reinitialize->setEnabled(true);

}

BtnGroup::~BtnGroup()

{

delete ui;

}

1. 头文件HashSearch.h：

#ifndef HASHSEARCH\_H

#define HASHSEARCH\_H

#include <QWidget>

#include "mainwindow.h"

class HashSearch : public Search

{

Q\_OBJECT

struct node{

int data = NULL;

node\* next = NULL;

bool isSearched = false;

};

struct bucket{

int index = NULL;

node\* node = NULL;

bucket\* next = NULL;

};

public:

explicit HashSearch(QWidget \*parent = nullptr);

void paintEvent(QPaintEvent \*);

void hashSearch();

void getsignal(bool isHashSearch,int length,int target,int \*nums);//接收信号，决定演示哪种算法

void searchBucket(bucket\* &t,int index);//根据index搜寻对应的bucket

private:

bool isfirst = true;//是否是第一次绘图

bool ishash = false;

int \*nums;

int buckedtLength = 7;

int bucketIndex;

int length;

int target;

int currentIndex;

int count;//查询次数

bucket \*firstBucket;

signals:

};

#endif // HASHSEARCH\_H

1. HashSearch类：

#include "hashsearch.h"

#include "QPainter"

#include "QDebug"

HashSearch::HashSearch(QWidget \*parent) : Search(parent)

{

}

//获取信号

void HashSearch::getsignal(bool isHashSearch,int length,int target,int \*nums){

ishash = isHashSearch;

currentIndex = -1;

count = 0;

this->nums = nums;

this->length = length;

this->target = target;

//散列表查找

if(ishash){

update();

hashSearch();

}

}

//散列表查找对数据的处理

void HashSearch::hashSearch(){

MainWindow m;

firstBucket = new bucket;

bucket\* p = firstBucket;

bucket\* last = NULL;

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

if(p!=firstBucket)

p = new bucket;

p->index = i;

p->node = new node;

p->node->isSearched = false;

if(last!=NULL)

last->next = p;

last = p;

p = p->next;

}

node\* t;

node\* currentNode;

bucket\* n;

//数据处理,为每个散列项加上后面的节点

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

int bucketIndex = nums[i];

while(bucketIndex>buckedtLength-1){

bucketIndex = bucketIndex%buckedtLength;

}

t = new node;

t->isSearched = false;

t->data = nums[i];

searchBucket(n,bucketIndex);

currentNode = n->node;

while(currentNode->next!=NULL){

currentNode = currentNode->next;

}

currentNode->next = t;

}

//确定目标所在的buckedIndex

bucketIndex = target;

while(bucketIndex>buckedtLength-1){

bucketIndex = bucketIndex % buckedtLength;

}

//第一次绘图，打上第一次绘图的标记

isfirst = true;

repaint();

//这里的currentIndex代表当前指向的节点

searchBucket(n,bucketIndex);

node\* index = n->node;

currentIndex = index->data;

count++;

while(index->data!=target){

index = index->next;

count++;

if(index==NULL)

break;

currentIndex = index->data;

repaint();

}

if(index==NULL){

currentIndex = FalseSearch;

}

if(currentIndex==FalseSearch&&isterminate==false){

m.result(1,count);

}

if(currentIndex!=FalseSearch&&isterminate==false){

m.result(0,count);

}

}

//从buckets中找到指定的节点

void HashSearch::searchBucket(bucket\* &t,int index){

bucket\* n = firstBucket;

while(n!=NULL){

if(n->index==index){

t = n;

return;

}

n = n->next;

}

}

//绘画事件

void HashSearch::paintEvent(QPaintEvent \*){

QPainter painter;

painter.begin(this);

//手动绘制边框

painter.setPen(QColor(139, 139, 139));

painter.drawLine(0, 0, this->width() - 1, 0);

painter.drawLine(0, 0, 0, this->height() - 1);

painter.drawLine(this->width() - 1, 0, this->width() - 1, this->height() - 1);

painter.drawLine(0, this->height() - 1, this->width() - 1, this->height() - 1);

//用于终止演示

if(isterminate){

return;

}

if(ishash){

int height = this->height()/buckedtLength;

int width = height;

int lineLength = width/2;

int x = 0;

int y = 0;

bucket \* n;

//判断当前节点是否为头节点

bool ishead = true;

//判断当前行是否已经完成当前节点的指向转移

bool isget = false;

//每循环一次绘制一行

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

searchBucket(n,i);

node\* t = n->node;

//每循环一次画一个矩形

while(t!=NULL){

QRect rect(x,y,width,height);

painter.save();

if(t->next!=NULL)

painter.drawLine(QPointF(x+width,y+height/2), QPointF(x+width+lineLength,y+height/2));

if(target==t->data)

painter.fillRect(rect,Qt::SolidPattern);

if(currentIndex==t->data&&isfirst==false&&isget==false&&t->data!=NULL&&t->isSearched==false){

painter.fillRect(rect,QColor("#00DDDD"));

t->isSearched = true;

isget = true;

}

if(bucketIndex==i&&isfirst){

isfirst = false;

painter.fillRect(rect,QColor("#00DDDD"));

}

if(ishead){

painter.drawText(x+width/2-5,y+height/2+5,QString::number(n->index));

painter.setPen(QColor("#FFBB00"));

ishead = false;

}else{

painter.drawText(x+width/2-5,y+height/2+5,QString::number(t->data));

}

painter.drawRect(rect);

painter.restore();

t = t->next;

x = x+width+lineLength;

}

ishead = true;

isget = false;

x = 0;

y = y+height;

}

}

painter.end();

}

1. 头文件ListSearch.h：

#ifndef BINARYSEARCH\_H

#define BINARYSEARCH\_H

#include <QWidget>

#include "mainwindow.h"

class ListSearch : public Search

{

Q\_OBJECT

public:

explicit ListSearch(QWidget \*parent = nullptr);

void paintEvent(QPaintEvent \*);

void binarySearch();

void indexedSearch();

void getsignal(bool binary,bool indexed,int length,int target,int \*nums);//接收信号，决定演示哪种算法

private:

bool isBinary = false;

bool isIndexed = false;

int length;

int target;

int \*indexList;//每个索引项其中的最大值

int \*starts;//每个索引项的起始索引值

int \*lengths;//每个索引项的长度

int \*nums;//数据组

int begin;

int end;

int currentIndex = 0;//当前选中的索引

int count;//查询次数

};

#endif // BINARYSEARCH\_H

1. ListSearch类：

#include "listsearch.h"

#include "QPainter"

#include "QDebug"

#include "mainwindow.h"

ListSearch::ListSearch(QWidget \*parent) : Search(parent)

{

}

//绘图事件

void ListSearch::paintEvent(QPaintEvent \*){

QPainter painter;

painter.begin(this);

//手动绘制边框

painter.setPen(QColor(139, 139, 139));

painter.drawLine(0, 0, this->width() - 1, 0);

painter.drawLine(0, 0, 0, this->height() - 1);

painter.drawLine(this->width() - 1, 0, this->width() - 1, this->height() - 1);

painter.drawLine(0, this->height() - 1, this->width() - 1, this->height() - 1);

//用于终止演示时

if(isterminate){

return;

}

//计算每个节点在图上出现的位置

int width = this->width()/length;

int height = width/2;

int x = 0;

int y = this->height()/2-height/2;

//当选择为二分查找算法时并开始运行时，开始绘图

if(isBinary){

//绘图

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

QRect rect(x,y,width,height);

painter.drawRect(rect);

if(target==nums[i])

painter.fillRect(rect,Qt::SolidPattern);

if(begin==i)

painter.fillRect(rect,QColor("#33FFFF"));

if(end==i)

painter.fillRect(rect,QColor("#FFCC22"));

//数据绘图

painter.drawText(x+width/2-5,y+height/2+5,QString::number(nums[i]));

x += width;

}

}

//索引表法的绘图

if(isIndexed){

int index = 0;

if(indexList[0]>10)

index++;

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

QRect rect(x,y,width,height);

painter.drawRect(rect);

if(target==nums[i])

painter.fillRect(rect,Qt::SolidPattern);

if(i==currentIndex)

painter.fillRect(rect,QColor("#33FFFF"));

//数据绘图

painter.drawText(x+width/2-5,y+height/2+5,QString::number(nums[i]));

//对应某个索引项的开始索引值的时候，画椭圆圈起来

if(i==starts[index]){

int h = width\*lengths[index]/2;

int w = width\*lengths[index];

int y = this->height()/2-h/2;

painter.drawEllipse(x,y,w,h);

index++;

}

x += width;

}

}

painter.end();

}

//接收开始按钮的信号,根据参数决定演示哪种查找

void ListSearch::getsignal(bool binary,bool indexed,int length,int target,int \*nums){

isBinary = binary;

isIndexed = indexed;

this->nums = nums;

this->length = length;

this->target = target;

//对数据进行排序，这里采用冒泡排序

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

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

if(nums[k]>nums[k+1]){

int t = nums[k];

nums[k] = nums[k+1];

nums[k+1] = t;

}

}

}

//二分查找

if(isBinary){

update();

binarySearch();

}

//索引表查询

if(isIndexed){

update();

indexedSearch();

}

}

//索引表查找的数据处理

void ListSearch::indexedSearch(){

MainWindow m;

//判断查找数据是否合法

currentIndex = 0;

indexList = new int[length];

starts = new int[length];

lengths = new int[length];

count = 0;

for (int k = 0;k<length;k++) {

lengths[k] = 0;

}

int t;

int last = 0;

int index = 0;

//索引表分表

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

t = nums[i];

if(t<10){

starts[index] = 0;

indexList[index] = nums[i];

lengths[index]++;

continue;

}

t = t/10;

if(t!=last){

index++;

starts[index] = i;

indexList[index] = nums[i];

lengths[index]++;

}else{

indexList[index] = nums[i];

lengths[index]++;

}

last = t;

}

if(target<nums[0]||target>nums[length-1]){

m.result(1,0);

return;

}

//搜寻合适的索引项

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

currentIndex = starts[i];

count++;

if(nums[currentIndex]>target){

m.result(1,0);

return;

}

repaint();

if(isterminate)

return;

if(target<=indexList[i]){

break;

}

}

//如果一开始就找到了，直接输出

if(target==nums[currentIndex]&&isterminate==false){

m.result(currentIndex,count);

}

//在某个索引项中开始查找目标

while(target!=nums[currentIndex]){

currentIndex++;

count++;

if(nums[currentIndex]>target){

m.result(1,0);

return;

}

repaint();

if(isterminate)

return;

if(target==nums[currentIndex]&&isterminate==false){

m.result(currentIndex,count);

break;

}

}

}

//二分查找的数据处理

void ListSearch::binarySearch(){

begin = 0;

end = length-1;

count = 0;

MainWindow m;

if(target>nums[end]||target<nums[begin]){

m.result(1,0);

return;

}

while(begin<=end){

count++;

repaint();

if(isterminate)

return;

if(target==nums[begin]){

m.result(begin,count);

break;

}

if(target==nums[end]){

m.result(end,count);

break;

}

if(target>nums[(end+begin)/2]){

begin = (end+begin)/2+1;

update();

continue;

}

if(target<nums[(end+begin)/2]){

end = (end+begin)/2-1;

update();

continue;

}

if(target==nums[(end+begin)/2]){

m.result((end+begin)/2,count);

break;

}

}

if(begin>end){

m.result(1,0);

}

}

1. Main类：

#include "mainwindow.h"

#include <QApplication>

int main(int argc, char \*argv[])

{

QApplication a(argc, argv);

MainWindow w;

w.show();

return a.exec();

}

1. 头文件MainWindow.h：

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

#include "search.h"

#include "btngroup.h"

QT\_BEGIN\_NAMESPACE

namespace Ui { class MainWindow; }

QT\_END\_NAMESPACE

class MainWindow : public QMainWindow

{

Q\_OBJECT

public:

MainWindow(QWidget \*parent = nullptr);

~MainWindow();

void generate();//生成数据函数

void result(int index,int count);//查询结果消息弹窗

void start(Search\* currentSearch,BtnGroup\* btngroup,int page);

void pause(Search\* currentSearch,BtnGroup\* btngroup);

void terminate(Search\* currentSearch,BtnGroup\* btngroup);

void reinitialize(Search\* currentSearch,BtnGroup\* btngroup,int page);

void initChcek();//切换页面时初始化所有算法的选择

void setTarget();//获取用户输入的查找值

private:

Ui::MainWindow \*ui;

int randMax = 40;//随机数据的范围最大值

int randMin = 1;//随机数据的范围最小值

int max;//随机数据中的最小值

int min;//随机数据中的最大值

int randLengthMax = 20;//随机数据长度的范围最大值

int randLengthMin = 15;//随机数据长度的范围最小值

int length;//数据长度

int target = randMax+1;//数据目标

int \*nums;//存放数据的数组

int isready = false;//是否已生成数据

bool isProper = false;//查找目标是否合法

};

#endif // MAINWINDOW\_H

1. MainWindow类：

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "listsearch.h"

#include "QTime"

#include "QMessageBox"

#include "QDebug"

MainWindow::MainWindow(QWidget \*parent)

: QMainWindow(parent)

, ui(new Ui::MainWindow)

{

ui->setupUi(this);

this->setWindowTitle("查找算法图形化");

this->setFixedSize(1000,541);

//初始将暂停按钮禁用

ui->btnGroup->pause->setEnabled(false);

ui->btnGroup\_2->pause->setEnabled(false);

ui->btnGroup\_3->pause->setEnabled(false);

//算法页面与对应的类型页面相匹配

connect(ui->tabWidget,&QTabWidget::currentChanged,ui->stackedWidget,[=](){

ui->stackedWidget->setCurrentIndex(ui->tabWidget->currentIndex());

initChcek();

terminate(ui->showBoard,ui->btnGroup);

terminate(ui->showBoard\_2,ui->btnGroup\_2);

terminate(ui->showBoard\_3,ui->btnGroup\_3);

});

//点击开始演示按钮时，发送信号

connect(ui->btnGroup->start,&QPushButton::clicked,[=](){

start(ui->showBoard,ui->btnGroup,1);

});

connect(ui->btnGroup\_2->start,&QPushButton::clicked,[=](){

start(ui->showBoard\_2,ui->btnGroup\_2,2);

});

connect(ui->btnGroup\_3->start,&QPushButton::clicked,[=](){

start(ui->showBoard\_3,ui->btnGroup\_3,3);

});

//点击暂停按钮

connect(ui->btnGroup->pause,&QPushButton::clicked,[=](){

pause(ui->showBoard,ui->btnGroup);

});

connect(ui->btnGroup\_2->pause,&QPushButton::clicked,[=](){

pause(ui->showBoard\_2,ui->btnGroup\_2);

});

connect(ui->btnGroup\_3->pause,&QPushButton::clicked,[=](){

pause(ui->showBoard\_3,ui->btnGroup\_3);

});

//点击生成数据按钮，生成数据

connect(ui->generate,&QPushButton::clicked,[=](){

generate();

ui->btnGroup->start->setText("开始演示");

ui->btnGroup\_2->start->setText("开始演示");

ui->btnGroup\_3->start->setText("开始演示");

});

//点击终止按钮时

connect(ui->btnGroup->terminate,&QPushButton::clicked,[=](){

terminate(ui->showBoard,ui->btnGroup);

});

connect(ui->btnGroup\_2->terminate,&QPushButton::clicked,[=](){

terminate(ui->showBoard\_2,ui->btnGroup\_2);

});

connect(ui->btnGroup\_3->terminate,&QPushButton::clicked,[=](){

terminate(ui->showBoard\_3,ui->btnGroup\_3);

});

//点击重新初始化按钮

connect(ui->btnGroup->reinitialize,&QPushButton::clicked,[=](){

reinitialize(ui->showBoard,ui->btnGroup,1);

});

connect(ui->btnGroup\_2->reinitialize,&QPushButton::clicked,[=](){

reinitialize(ui->showBoard\_2,ui->btnGroup\_2,2);

});

connect(ui->btnGroup\_3->reinitialize,&QPushButton::clicked,[=](){

reinitialize(ui->showBoard\_3,ui->btnGroup\_3,3);

});

}

//切换页面时清空所有算法的选择

void MainWindow::initChcek(){

//不关掉自动排除，则无法修改按钮的checked值

ui->binary->setAutoExclusive(false);

ui->binary->setChecked(false);

ui->binary->setAutoExclusive(true);

ui->index->setAutoExclusive(false);

ui->index->setChecked(false);

ui->index->setAutoExclusive(true);

ui->hash->setAutoExclusive(false);

ui->hash->setChecked(false);

ui->hash->setAutoExclusive(true);

ui->balancedTree->setAutoExclusive(false);

ui->balancedTree->setChecked(false);

ui->balancedTree->setAutoExclusive(true);

}

//开始按钮功能的封装

void MainWindow::start(Search\* currentSearch,BtnGroup\* btngroup,int page){

setTarget();

if(isready==false||isProper==false)

return;

currentSearch->ispause = false;

currentSearch->isterminate = false;

btngroup->startBegin();

if(page==1)

ui->showBoard->getsignal(ui->binary->isChecked(),ui->index->isChecked(),length,target,nums);

if(page==2)

ui->showBoard\_2->getsignal(ui->BTree->isChecked(),ui->balancedTree->isChecked(),length,target,nums);

if(page==3)

ui->showBoard\_3->getsignal(ui->hash->isChecked(),length,target,nums);

btngroup->startEnd();

}

//暂停按钮功能封装

void MainWindow::pause(Search\* currentSearch,BtnGroup\* btngroup){

currentSearch->ispause = !currentSearch->ispause;

if(currentSearch->ispause==true)

btngroup->pause->setText("继续演示");

else

btngroup->pause->setText("暂停演示");

}

//终止按钮功能封装

void MainWindow::terminate(Search\* currentSearch,BtnGroup\* btngroup){

currentSearch->isterminate = true;

currentSearch->ispause = false;

btngroup->pause->setText("暂停演示");

btngroup->pause->setEnabled(false);

btngroup->start->setEnabled(true);

btngroup->reinitialize->setEnabled(true);

//以白板覆盖,达到终止演示的效果

currentSearch->update();

}

//重新初始化功能封装

void MainWindow::reinitialize(Search\* currentSearch,BtnGroup\* btngroup,int page){

terminate(currentSearch,btngroup);

generate();

btngroup->reinitialize->setEnabled(false);

btngroup->start->setText("开始演示");

start(currentSearch,btngroup,page);

}

//随机生成数据

void MainWindow::generate(){

isready = true;

//初始化最大最小值，方便后面查询随机数据的最大最小值

min = randMax;

max = randMin;

qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));

//随机生成数据长度

length = qrand()%(randLengthMax-randLengthMin)+randLengthMin;

nums = new int[length];

//初始化数组

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

nums[i] = -1;

}

//随机生成数据,并对数据进行去重

int i = 0;

while(i<length){

bool isSame = false;

int num = qrand()%(randMax-randMin)+randMin;

for (int k = 0;nums[k]!=-1;k++) {

if(num==nums[k]){

isSame = true;

break;

}

}

if(isSame==true){

continue;

}

nums[i] = num;

if(max<nums[i])

max = nums[i];

if(min>nums[i])

min = nums[i];

i++;

}

//确定目标值

target = nums[qrand()%(length-1)];

//显示随机数据的一些信息

ui->target->setText(QString::number(target));

ui->length->setText(QString::number(length));

ui->max->setText(QString::number(max));

ui->min->setText(QString::number(min));

}

//通知查询结果的消息窗

void MainWindow::result(int index,int count){

QMessageBox msgBox;

if(index==1){

msgBox.setText("查询失败！请重新设置查找目标！");

msgBox.exec();

return;

}

QString s = "查询完毕!";

s.append("\n");

if(index!=0){

s.append("目标第一次出现的下标为：");

s.append(QString::number(index));

s.append(" ");

s.append("\n");

}

s.append("查询次数为：");

s.append(QString::number(count));

s.append(" ");

s.append("\n");

msgBox.setText(s);

msgBox.setWindowTitle("查询结果");

msgBox.exec();

}

//获取用户输入的查找值

void MainWindow::setTarget(){

isProper = false;

QString str = ui->target->text();

if(str=="")

return;

int target = str.toUInt();

if(target>0)

isProper = true;

if(isProper == true)

this->target = target;

else{

QMessageBox msgBox;

QString s = "您输入的查找目标不合法!";

s.append('\n');

s.append("查找目标值必须大于0!");

msgBox.setText(s);

msgBox.setWindowTitle("查询错误");

msgBox.exec();

}

}

MainWindow::~MainWindow()

{

delete ui;

}

1. 头文件Search.h：

#ifndef SEARCH\_H

#define SEARCH\_H

#include <QWidget>

#include "QTime"

#define FalseSearch -2147483648

class Search : public QWidget

{

Q\_OBJECT

public:

explicit Search(QWidget \*parent = nullptr);

bool ispause = false;

bool isterminate = false;

void repaint();//更新绘画

void Sleep(int msec);//自定义延时器

signals:

};

#endif // SEARCH\_H

1. Search类：

#include "search.h"

#include "QTime"

#include "QCoreApplication"

Search::Search(QWidget \*parent) : QWidget(parent)

{

}

//重新绘画

void Search::repaint(){

//将1.5s的绘画延迟重塑成15份0.1s,更加频繁的检测自身的状态,已解决程序未及时终止便开始新一轮绘图的情况

for (int i = 0;i<15;i++) {

Sleep(100);

while(ispause){

Sleep(100);

}

if(isterminate)

return;

}

update();

}

//自定义的延时器函数

void Search::Sleep(int msec)

{

QTime dieTime = QTime::currentTime().addMSecs(msec);

while( QTime::currentTime() < dieTime )

QCoreApplication::processEvents(QEventLoop::AllEvents, 100);

}

1. 头文件TreeSearch.h：

#ifndef TREESEARCH\_H

#define TREESEARCH\_H

#define ORDER 3

#include <QWidget>

#include "mainwindow.h"

#include "search.h"

#include "QTime"

#include <QQueue>

class TreeSearch : public Search

{

Q\_OBJECT

struct bnode{

int data;

bnode\* leftC = NULL;

bnode\* rightC = NULL;

int level = 0;//节点的层数

int bf = 0;//平衡因子

};

const int Max=ORDER-1; //结点的最大关键字数量

const int Min=(ORDER-1)/2; //结点的最小关键字数量

struct Bnode{ //B树和B树结点类型

int keynum; //结点关键字个数

int key[ORDER+1]; //关键字数组，key[0]不使用

struct Bnode \*parent; //双亲结点指针

struct Bnode \*ptr[ORDER+1]; //孩子结点指针数组

Bnode(){

for (int i = 0; i < ORDER+1 + 1; i++)

{

ptr[i] = NULL;

key[i] = -1;

}

keynum = 0;

parent = NULL;

}

};

public:

explicit TreeSearch(QWidget \*parent = nullptr);

void paintEvent(QPaintEvent \*);//绘图事件

void balancedTreeSearch();//平衡二叉树查找数据处理

void BTreeSearch();//B树查找数据处理

void getsignal(bool isBTree,bool isbalancedTree,int length,int target,int \*nums);//接收信号，决定演示哪种算法

int SearchNode(bnode\* &target,int data);//查找到data数据的父节点

void paint(int x,int y,bnode\* n);//给当前节点画左右两个儿子节点的图

void Bpaint(int x,int y,int px,int py,int index,Bnode\* b);//给B树的当前节点画儿子节点的图

int getHeight(bnode\* t);//获取以当前节点为根的二叉树的高度

void setBF(bnode\* t);//更新整棵树每个节点的平衡因子

void getBf(bnode\* &target);//获取平衡因子绝对值超过1的最高层次的那个节点（注意，这里我没有采用找层次最低的平衡点，因为有个BUG实在无法定位修复，所以采取了消耗更大的找层次最高的平衡点）

void correct(bnode\* begin,bnode\* target);//判断当前错误属于什么型，并执行相应的修复函数

void getLevel(bnode\* t);//更新所有节点的层次数

void setIndex(Bnode\* &b,int num);//给新关键字找到合适的位置插入

void reset(Bnode\* &b);//重置调整B树

void SplitBnode(Bnode \*&p,Bnode \*&q);//将结点p分裂成两个结点,前一半保留,后一半移入结点q

void dfs(double x,double y,double px,double py,int index,int level,Bnode\* b);//深度优先遍历B树,并调用绘画

void getMaxLevel(Bnode\* b);//获取B树的最大深度

void LL(bnode\* t);

void RR(bnode\* t);

void LR(bnode\* t);

void RL(bnode\* t);

private:

bnode\* root;//二叉树的根节点

Bnode\* Broot;//B树根节点

bool isbalancedTree = false;

bool isBTree = false;

int length;

int target;

int currentIndex;

int \*nums;

int count;//查询次数

int r = 20;//圆的半径

double angle = 30;//初始偏转角度

double addangel = 7.5;//每一层改变的偏转角度量

QQueue<int> q;//存储每个节点的x,y坐标

QQueue<bnode\*> node;//存储节点

QQueue<bnode\*> node\_2;//存储节点，用于广度优先遍历

int w = 26;//B树关键字节点的宽度

int h = 20;//B树关键字节点的高度

int maxLevel;//B树的最大深度

};

#endif // TREESEARCH\_H

1. TreeSearch类：

#include "treesearch.h"

#include "QPainter"

#include "math.h"

#include "QDebug"

#include <QQueue>

#include "math.h"

TreeSearch::TreeSearch(QWidget \*parent) : Search(parent)

{

}

//绘图事件

void TreeSearch::paintEvent(QPaintEvent \*){

QPainter painter;

painter.begin(this);

//手动绘制边框

painter.setPen(QColor(139, 139, 139));

painter.drawLine(0, 0, this->width() - 1, 0);

painter.drawLine(0, 0, 0, this->height() - 1);

painter.drawLine(this->width() - 1, 0, this->width() - 1, this->height() - 1);

painter.drawLine(0, this->height() - 1, this->width() - 1, this->height() - 1);

//用于终止演示时

if(isterminate){

return;

}

if(isbalancedTree){

int x = this->width()/2;

int y = r;

bnode \*n;

//画根节点

node.enqueue(root);

painter.setPen(Qt::black);

if(root->data==target){

painter.setPen(Qt::blue);

}

if(root->data==currentIndex){

painter.setPen(Qt::red);

}

painter.drawEllipse(QPoint(x,y),r,r);

painter.drawText(QPoint(x-8,y+4),QString::number(root->data));

q.enqueue(this->width()/2);

q.enqueue(r);

while(!q.isEmpty()){

n = node.dequeue();

x = q.dequeue();

y = q.dequeue();

paint(x,y,n);

}

}

if(isBTree){

Bnode\* b = Broot;

int x = this->width()/2 - w\*b->keynum/2;

int y = 0;

int px = x;

int py = y;

maxLevel = 0;

getMaxLevel(Broot);

dfs(x,y,px,py,0,1,b);

}

painter.end();

}

//给当前节点画左右两个儿子节点的图

void TreeSearch::paint(int x,int y,bnode\* n){

QPainter painter;

painter.begin(this);

int lx = x-8\*r/pow(2,n->level-1);

int ly = y+(x-lx)\*tan((angle+(n->level)\*addangel)\*3.1415/180);

int rx = x+8\*r/pow(2,n->level-1);

int ry = ly;

if(n->leftC!=NULL){

painter.drawLine(QPoint(x,y),QPoint(lx,ly));

if(n->leftC->data==target){

painter.setPen(Qt::blue);

}

if(n->leftC->data==currentIndex){

painter.setPen(Qt::red);

}

painter.drawEllipse(QPoint(lx,ly),r,r);

painter.drawText(QPoint(lx-8,ly+4),QString::number(n->leftC->data));

q.enqueue(lx);

q.enqueue(ly);

node.enqueue(n->leftC);

}

painter.setPen(Qt::black);

if(n->rightC!=NULL){

painter.drawLine(QPoint(x,y),QPoint(rx,ry));

if(n->rightC->data==target){

painter.setPen(Qt::blue);

}

if(n->rightC->data==currentIndex){

painter.setPen(Qt::red);

}

painter.drawEllipse(QPoint(rx,ry),r,r);

painter.drawText(QPoint(rx-8,ry+4),QString::number(n->rightC->data));

q.enqueue(rx);

q.enqueue(ry);

node.enqueue(n->rightC);

}

painter.end();

}

//接收开始按钮的信号,根据参数决定演示哪种查找

void TreeSearch::getsignal(bool isBTree,bool isbalancedTree,int length,int target,int \*nums){

this->isbalancedTree = isbalancedTree;

this->isBTree = isBTree;

this->nums = nums;

this->length = length;

this->target = target;

//平衡树

if(this->isbalancedTree){

update();

balancedTreeSearch();

}

//B树

if(this->isBTree){

update();

BTreeSearch();

}

}

//平衡二叉树查找数据处理

void TreeSearch::balancedTreeSearch(){

MainWindow m;

root = new bnode;

root->data = nums[0];

root->level = 1;

root->bf = 0;

bnode\* t;

count = 1;

int l;

//根据数据生成二叉树

for (int i = 1;i<length;i++) {

bnode\* begin = NULL;

bnode\* newNode = new bnode;

newNode->data = nums[i];

l = SearchNode(t,nums[i]);

if(nums[i]>t->data){

t->rightC = newNode;

}else{

t->leftC = newNode;

}

newNode->level = l+1;

setBF(root);

getBf(begin);

if(begin!=NULL){

correct(begin,newNode);

}

}

//更新所有节点的层次

getLevel(root);

update();

//查找开始

currentIndex = root->data;

repaint();

if(isterminate==true){

return;

}

bnode\* n = root;

while(target!=currentIndex&&n!=NULL){

if(target>n->data){

n = n->rightC;

}else if(target<n->data){

n = n->leftC;

}

if(n==NULL){

currentIndex = FalseSearch;

}else{

currentIndex = n->data;

}

count++;

repaint();

if(isterminate==true){

return;

}

}

if(currentIndex==FalseSearch&&isterminate==false){

m.result(1,count);

}

if(currentIndex!=FalseSearch&&isterminate==false){

m.result(0,count);

}

}

//查找到data数据的父节点

int TreeSearch::SearchNode(bnode\* &target,int data){

bnode\* t = root;

int level = 1;

while(t!=NULL&&t->data!=data){

target = t;

if(data>t->data){

t = t->rightC;

}else{

//如果节点值与数据相等，统一放右边

t = t->leftC;

}

level++;

}

return level;

}

//获取以当前节点为根的二叉树的高度

int TreeSearch::getHeight(bnode\* t){

if(t==NULL){

return 0;

}

return qMax(getHeight(t->leftC)+1,getHeight(t->rightC)+1);

}

//更新整棵树每个节点的平衡因子

void TreeSearch::setBF(bnode\* t){

if(t!=NULL){

t->bf = getHeight(t->leftC) - getHeight(t->rightC);

setBF(t->leftC);

setBF(t->rightC);

}

}

//获取平衡因子绝对值超过1的最高层次的那个节点（注意，这里我没有采用找层次最低的平衡点，因为有个BUG实在无法定位修复，所以采取了消耗更大的找层次最高的平衡点）

void TreeSearch::getBf(bnode\* &target){

bnode\* t;

node\_2.enqueue(root);

while(!node\_2.isEmpty()){

t = node\_2.dequeue();

if(qAbs(t->bf)>1){

target = t;

}

if(t->leftC!=NULL)

node\_2.enqueue(t->leftC);

if(t->rightC!=NULL)

node\_2.enqueue(t->rightC);

}

}

//判断当前错误属于什么型，并执行相应的修复函数

void TreeSearch::correct(bnode\* begin,bnode\* target){

if(begin->leftC==NULL)

qDebug()<<"";

if(target->data > begin->data){

if(begin->rightC!=NULL && target->data > begin->rightC->data){

RR(begin);

}else{

RL(begin);

}

}else{

if(begin->leftC!=NULL && target->data < begin->leftC->data){

LL(begin);

}else{

LR(begin);

}

}

}

void TreeSearch::LL(bnode\* t){

bnode\* begin = t;

bnode\* mid = begin->leftC;

if(mid->rightC!=NULL){

begin->leftC = mid->rightC;

mid->rightC = begin;

}else{

begin->leftC = NULL;

mid->rightC = begin;

}

//如果根节点参与了旋转，需要跳转root的指向

if(begin==root){

root = mid;

}else{

bnode\* p;

SearchNode(p,begin->data);

if(p->leftC==begin)

p->leftC = mid;

else

p->rightC = mid;

}

}

void TreeSearch::RR(bnode\* t){

bnode\* begin = t;

bnode\* mid = begin->rightC;

if(mid->leftC!=NULL){

begin->rightC = mid->leftC;

mid->leftC = begin;

}else{

begin->rightC = NULL;

mid->leftC = begin;

}

//如果根节点参与了旋转，需要跳转root的指向

if(begin==root){

root = mid;

}else{

bnode\* p;

SearchNode(p,begin->data);

if(p->leftC==begin)

p->leftC = mid;

else

p->rightC = mid;

}

}

void TreeSearch::LR(bnode\* t){

bnode\* begin = t;

bnode\* mid = begin->leftC;

bnode\* end = mid->rightC;

if(end->leftC!=NULL){

begin->leftC = end;

mid->rightC = end->leftC;

end->leftC = mid;

}else{

begin->leftC = end;

mid->rightC = NULL;

end->leftC = mid;

}

LL(begin);

}

void TreeSearch::RL(bnode\* t){

bnode\* begin = t;

bnode\* mid = begin->rightC;

bnode\* end = mid->leftC;

if(end->rightC!=NULL){

begin->rightC = end;

mid->leftC = end->rightC;

end->rightC = mid;

}else{

begin->rightC = end;

mid->leftC = NULL;

end->rightC = mid;

}

RR(begin);

}

//更新所有节点的层次数

void TreeSearch::getLevel(bnode\* t){

if(t==NULL){

return;

}

int level = 1;

bnode\* curNode = root;

while(curNode->data!=t->data){

if(t->data>curNode->data){

curNode = curNode->rightC;

}else{

curNode = curNode->leftC;

}

level++;

}

t->level = level;

getLevel(t->leftC);

getLevel(t->rightC);

}

//B树的数据处理

void TreeSearch::BTreeSearch(){

Broot = new Bnode;

Broot->key[1] = nums[0];

Broot->keynum = 1;

for (int i = 1;i<length;i++) {

Bnode\* b = Broot;

for (int j = 0;j<=ORDER+1;j++) {

if(((nums[i] < b->key[j+1]) || (nums[i] > b->key[j+1] && b->key[j+1]==-1)) && b->ptr[j]!=NULL){

b = b->ptr[j];

j = -1;

continue;

}

if(nums[i] < b->key[j+1]&&b->ptr[j]==NULL){

setIndex(b,nums[i]);

break;

}

if(b->key[j]==-1 && j!=0){

b->key[j]=nums[i];

b->keynum++;

break;

}

}

if(b->keynum == Max+1){

reset(b);

}

}

update();

//在B树中开始搜索

Bnode\* b = Broot;

MainWindow m;

count = 0;

for (int j = 0;j<=ORDER+1;j++) {

if(b==NULL){

currentIndex = -999;

repaint();

if(isterminate==true){

return;

}

m.result(1,count);

return;

}

if( b->key[j] < target && target < b->key[j+1] ){

currentIndex = b->key[j+1];

count++;

repaint();

if(isterminate==true){

return;

}

b = b->ptr[j];

j = -1;

continue;

}

if( b->key[j+1] < target && ( b->key[j+2]==-1 || j+1==ORDER )){

currentIndex = -b->key[j+1]-1;

count++;

repaint();

if(isterminate==true){

return;

}

b = b->ptr[j+1];

j = -1;

continue;

}

if(target==b->key[j+1]){

currentIndex = target;

count++;

repaint();

if(isterminate==true){

return;

}

m.result(0,count);

return;

}

}

}

//给新关键字找到合适的位置插入

void TreeSearch::setIndex(Bnode\* &b,int num){

for (int i = 1;i<ORDER+1;i++) {

if(b->key[i]>num&&b->key[i]!=-1){

for (int k = ORDER;k>=i;k--) {

b->key[k] = b->key[k-1];

}

b->key[i] = num;

break;

}

if(b->key[i]==-1){

b->key[i] = num;

break;

}

}

b->keynum++;

}

//重置调整B树

void TreeSearch::reset(Bnode\* &b){

Bnode\* newChild = new Bnode;

Bnode\* parent;

if(b->parent==NULL){

parent = new Bnode;

parent->key[1] = b->key[ORDER/2+1];

parent->keynum = 1;

SplitBnode(b,newChild);

parent->ptr[0] = b;

b->parent = parent;

parent->ptr[1] = newChild;

newChild->parent = parent;

Broot = parent;

}else{

parent = b->parent;

setIndex(parent,b->key[ORDER/2+1]);

SplitBnode(b,newChild);

for (int i = 0;i<ORDER+1;i++) {

if(parent->ptr[i]==b){

for (int k = ORDER; k>i; k--) {

parent->ptr[k+1] = parent->ptr[k];

}

parent->ptr[i+1] = newChild;

newChild->parent = parent;

break;

}

}

if(parent->keynum==Max+1){

reset(parent);

}

}

}

//将结点p分裂成两个结点,前一半保留,后一半移入结点q

void TreeSearch::SplitBnode(Bnode \*&p,Bnode \*&q){

int index = 0;

p->key[ORDER/2+1] = -1;

p->keynum--;

for (int i = ORDER/2+2;i<ORDER+1;i++) {

if(p->key[i]!=-1){

q->key[index+1] = p->key[i];

q->keynum++;

q->ptr[index] = p->ptr[i-1];

p->ptr[i-1] = NULL;

p->key[i] = -1;

p->keynum--;

index++;

}

}

q->ptr[index] = p->ptr[ORDER];

p->ptr[ORDER] = NULL;

//将新节点的儿子节点的父亲节点重置为新结点自己

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

if(q->ptr[i]!=NULL)

q->ptr[i]->parent = q;

}

}

//给B树的当前节点画儿子节点的图

void TreeSearch::Bpaint(int x,int y,int px,int py,int index,Bnode\* b){

QPainter painter;

painter.begin(this);

int ww;

int wh = h/2;

int wx;

int wy = y+h;

int nums = b->keynum+1;

ww = w\*b->keynum/nums;

wx = x;

if(b!=Broot){

int piancha = (w\*b->parent->keynum/(b->parent->keynum+1))/2;

painter.drawLine(px+w\*b->parent->keynum/(b->parent->keynum+1)\*(index+1)-piancha,py+h\*5/4,x+w\*b->keynum/2,y);

}

bool flag = false;

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

if(i!=nums-1){

QRect rect(x,y,w,h);

painter.drawRect(rect);

QString s = "#FFCC22";

if(target==currentIndex)

s = "#33FFFF";

if(target==b->key[i+1])

painter.fillRect(rect,QColor(s));

painter.drawText(x+6,y+h-4,QString::number(b->key[i+1]));

x += w;

}

QRect rect(wx,wy,ww,wh);

painter.drawRect(rect);

wx += ww;

if(target!=currentIndex){

if(currentIndex==b->key[i+1])

painter.fillRect(rect,QColor("#33FFFF"));

if(currentIndex + b->key[i+1] == -1){

flag = true;

continue;

}

if(flag){

painter.fillRect(rect,QColor("#33FFFF"));

}

}

}

painter.end();

}

//深度优先遍历B树,并调用绘画

void TreeSearch::dfs(double x,double y,double px,double py,int index,int level,Bnode\* b){

if(b==NULL){

return;

}

Bpaint(x,y,px,py,index,b);

px = x;

py = y;

y += 4\*h;

double newX;

double change;

level++;

if(maxLevel==3){

if(level==2){

change = (double)7.5\*w;

}

if(level==3){

change = (double)2.5\*w;

}

}

if(maxLevel==4){

if(level==2){

change = (6+(double)7/12)\*w;

}

if(level==3){

change = (3+(double)1/4)\*w;

}

if(level==4){

change = (2+(double)1/6)\*w;

}

}

for (int i = 0;i<b->keynum+1;i++) {

newX = x;

if(b->keynum==1){

if(i == 0){

newX = newX-change;

}

if(i == 1){

newX = newX+change;

}

}else{

if(i == 0){

newX = newX-change;

}

if(i == 2){

newX = newX+change;

}

}

dfs(newX,y,px,py,i,level,b->ptr[i]);

}

}

//获取B树的最大深度

void TreeSearch::getMaxLevel(Bnode\* b){

if(b!=NULL){

maxLevel++;

getMaxLevel(b->ptr[0]);

}

}