**操作系统——文件管理项目3**

**学号：2151298**

**姓名：杨滕超**

**班号：42036902**

**指导老师：张惠娟**

目录

[操作系统——文件管理项目3 1](#_Toc26689)

[1. 项目介绍 2](#_Toc11165)

[(1) 基本要求 2](#_Toc28994)

[(2) 项目目的 2](#_Toc4757)

[(3) 文件系统提供的操作 2](#_Toc13187)

[2. 运行方法 2](#_Toc4760)

[3. 开发环境 3](#_Toc7201)

[4. 算法设计 3](#_Toc904)

[(1) 空闲空间管理 3](#_Toc22779)

[(2) 物理结构管理 3](#_Toc5198)

[(4) 数据存储管理 4](#_Toc19384)

[5. 界面设计 4](#_Toc7467)

[6. 代码设计 5](#_Toc9837)

[(1) 空闲空间管理 5](#_Toc31732)

[(2) 物理空间管理 9](#_Toc26430)

[(3) 目录结构管理 11](#_Toc8953)

[(4) 数据存储管理 12](#_Toc598)

[(5) 主窗口管理 13](#_Toc27752)

[7. 功能展示 16](#_Toc21107)

[(1) 本项目相关信息 16](#_Toc29628)

[(2) 新建 16](#_Toc9308)

[(3) 打开文件 18](#_Toc1665)

[(4) 打开文件夹 18](#_Toc27657)

[(5) 删除 18](#_Toc13807)

[(6) 编辑并保存文件 19](#_Toc19030)

[(7) 再次打开编辑文件 20](#_Toc2634)

[(8) 重命名 20](#_Toc10869)

[(9) 格式化 21](#_Toc23205)

[(10) 保存数据 22](#_Toc1985)

[(11) 载入数据 22](#_Toc11699)

1. **项目介绍**
   1. **基本要求**

·在内存中开辟一个空间作为文件存储器，在其上实现一个简单的文件系统。

·退出这个文件系统时，需要该文件系统的内容保存到磁盘上，以便下次可以将其回复到内存中来。

·文件存储空间管理可采取链接结构（如FAT文件系统中的显式链接等）或者其他学过的方法。

·空闲空间管理可采用位图或者其他方法。

·文件目录采用多级目录结构，目录项目中应包含：文件名、物理地址、长度等信息

* 1. **项目目的**

·理解文件存储空间的管理。

·掌握文件的物理结构、目录结构和文件操作。

·实现简单文件系统管理。

·加深文件系统实现过程的理解。

* 1. **文件系统提供的操作**

·格式化

·创建子目录

·删除子目录

·显示目录

·更改当前目录

·创建文件

·打开文件

·关闭文件

·写文件

·读文件

·删除文件

·...

1. **运行方法**

进入publish文件夹，双击可执行文件FileTest.exe。进行新建文件或者载入历史数据均可。

1. **开发环境**

开发环境：Windows\_NT x64 10.0.19044

开发软件：Visual Studio 2019

开发语言：C#

1. **算法设计**
   1. **空闲空间管理**

本项目采用位图技术管理磁盘空闲块。位图是一种简单、紧凑和高效的数据结构，通过一串二进制位来标识磁盘的分配情况，具有较小的空闲占用量和高度压缩的特点，能够快速地反应磁盘中各个物理块的使用情况。当需要申请使用空闲物理块的使用，只需要在位图中查找可用的空闲磁盘块，并且返回显影的磁盘块地址。相比与其他复杂的数据结构，位图所需的存储空间更少，适合于不同的物理结构并且具有较好的描述能力和性价比。此外，由于计算机中位运算的速度很快，使用位图还可以节省查询空闲块所需的时间。但是使用位图也有缺点：对于大型磁盘分区，需要消耗大量的内存来存储位图；随着磁盘容量的增加，位图的大小也会增加，进而导致查询的时间变长；不支持动态扩展磁盘空间。

除了位图技术之外，还有一些其他方法可以应用与管理磁盘空闲块，例如：空闲链表法、成组链表法。

空闲链表指的是将磁盘空闲块组成一个链表，每个节点表示一个空闲块，链表的头指针指向第一个空闲块。当需要分配磁盘块时，算法会遍历空闲链表，并将其从链表中删除，返回给系统；当释放磁盘块时，算法会将被释放的块加入空闲链表中，以便后续使用。

成组链表法在Unix系统中采用。以50组为例，组的划分为从后往前顺次划分。其中，每组的第一块用来存放前一组中各块的块号和总块数。由于第一组的前面已无其他存在，因此，第一组的块数为49块。不过，由于存储设备的空间块不一定正好是50的整倍数，因而最后一组将不足50块，且由于该组后面已无另外的空闲块组，所以，该组的物理块号与总数只能放在管理文件存储设备用的文件资源表中。在成组链法对文件设备进行了上述分组之后，系统可根据申请者的要求进行空闲块的分配，并在释放文件时回收空闲块。

* 1. **物理结构管理**

本项目采用综合模式的索引表管理文件的物理结构，即文件的数据信息被存放在若干个可能物理上不连续的物理块当中，而这若干个物理块地址由该文件的多级索引表存储。该方式对每一个文件都建立一张索引表，而索引表中划分一部分索引直接存储指向数据物理块的地址，剩下部分的索引指向专门存放索引的物理块，即本项目所采用的二级索引。使用这种组织方式技能顺序存取又能随机存取，不仅能够对文件的占用空间进行动态更改，从而充分提高磁盘利用率，而且进一步扩大文件的最大容量。

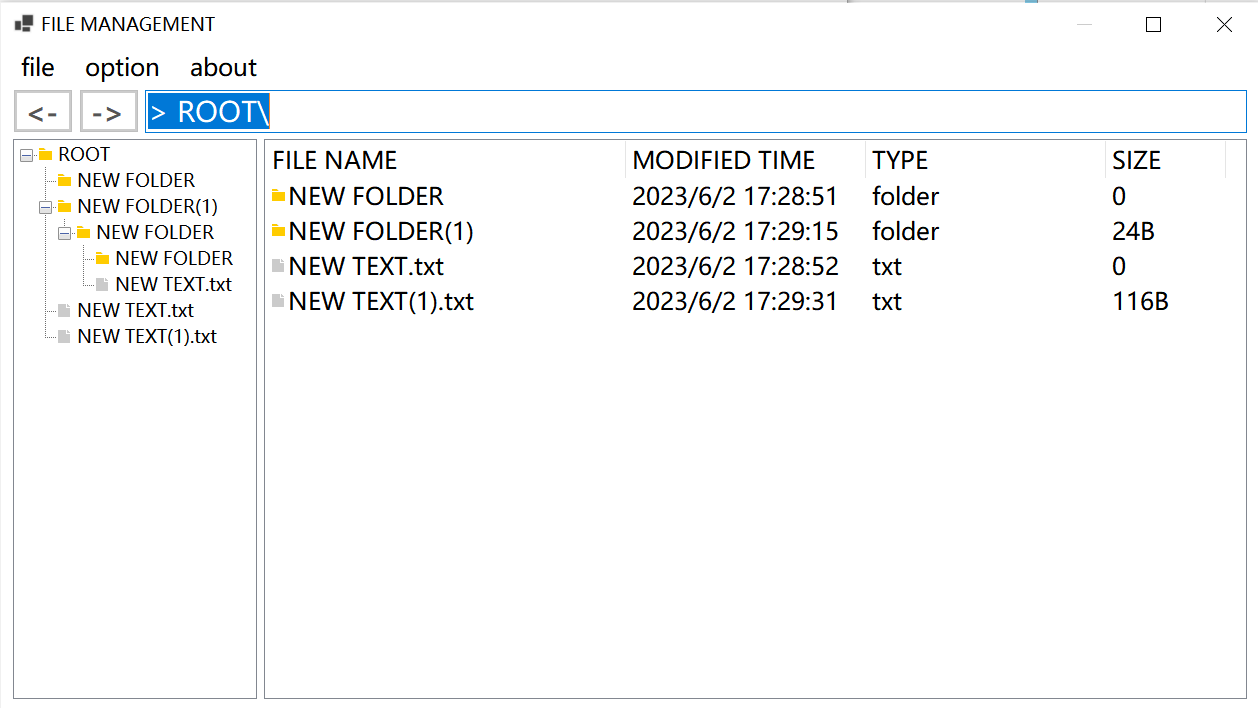
* 1. **目录结构管理**

本项目采用树形目录组织文件控制块。因此，为了节省目录组织大小，本项目将文件控制块分为符号目录项和基本目录项。其中符号目录项仅仅保留文件名、文件号等基本关键信息，用于组织属性结构和搜索，便于对文件的分类与管理。而基本目录项存储更多藐视文件的相关信息，如修改时间、地址等。每次查找文件的时候，首先对于树形结构搜索，查找对应的符号目录项，然后通过符号目录项找到对应的基本目录项，这里通过构建哈希表构建两者的映射关系。这种方法可以更方便地进行查询和管理目录结构，减少了查询文件所需要的I/O次数，节省存储空间，从而提高搜索效率。

* 1. **数据存储管理**

为了实现在程序上的操作结果在下一次启动程序时还能有作用，本项目设计将操作数据存储存储在电脑的文件中，主要采用C#的序列化方法，将程序中对象的数据保存在二进制文件中。在使用载入load功能时，从保存的文件中读取相关数据，然后反序列化为C#中的对象，从而可以恢复上一次保存的操作结果数据。

1. **界面设计**



本项目设计可视化界面，共分为三个部分：顶部菜单栏、左侧目录树视图以及右侧列表视图。

顶部菜单栏：该部分为用户提供该文件管理系统主要功能，通过点击不同的栏目可以查看该栏目下不同的功能。

·在file栏目中可以load、save即为将本地存储数据载入程序或将程序中的数据保存至本地。

·在option栏目中可以new、open、delete、reset、rename，即为新建文件、打开文件、删除文件、格式化、重命名。

·在about栏目中可以查看相关信息。

·点击前进、后退按钮可以进行目录的切换功能，切换到父目录或历史子目录。

·当前目录的地址同样显示在顶部。

左侧目录树视图：该部分为用户提供根目录下文件的目录的树形结构关系视图，帮助用户更好地了解到文件的父子关系。

右侧列表视图：显示当前目录下的所有文件。

1. **代码设计**
   1. **空闲空间管理**

管理器，该类名为Manager。该类负责磁盘块的管理，包括读取、写入和删除磁盘块的数据。属性包括：capacity：最多能存储多少个块；apacity：最多能存储多少个块；bitMap：位图数组，用于管理磁盘块的使用情况；bitIdx：管理位图的指针，执行下一个要检查的位图的位置。方法包括GetBlockData(int idx)：获取指定索引的磁盘块数据；GetBlock(int idx)：获取指定索引的磁盘块；AllocBlock()：在空闲块中找到一个可用的块，返回其索引；remove(int idx)：删除指定索引的磁盘块及其关联的索引块；remove(List<int> idx\_list)：批量删除指定索引列表的磁盘块及其关联的索引块；WriteData(string data)：将传入的数据写入磁盘块，并构建索引表来管理数据和索引块之间的关系。

public class Manager

{

// 最多多少个块

public static int capacity = 1000000;

// 磁盘块列表

private Block[] blocks;

// 位图管理空余空间

private bool[] bitMap;

// 管理位图的指针

private int bitIdx;

// 获取某一块的数据

public string GetBlockData(int idx)

{

return blocks[idx].ReadData();

}

// 获取某一快

public Block GetBlock(int idx)

{

return blocks[idx];

}

// 找到一个空闲块，返回空闲块索引

public int AllocBlock()

{

bitIdx %= capacity;

int tempIdx = bitIdx;

while (true)

{

// 有空闲块

if (bitMap[tempIdx])

{

// 找到空位申请新空间

blocks[tempIdx] = new Block();

// blocks[tempIdx].WriteData(data);

bitIdx = tempIdx + 1;

return tempIdx;

}

// 没有空闲块下一个

else

{

// 防止越界

tempIdx = (tempIdx + 1) % capacity;

}

// 找了一圈没找到

if (tempIdx == bitIdx)

{

break;

}

}

return -1;

}

// 删除块数据

public void remove(int idx)

{

System.Diagnostics.Debug.Assert(idx >= 0 && idx < capacity);

// 数据块直接清除，但索引块还要清除block中的index列表中的所有索引

bitMap[idx] = true;

foreach (int index in blocks[idx].GetIndex())

{

bitMap[index] = true;

}

}

public void remove(List<int> idx\_list)

{

foreach (int idx in idx\_list)

{

// System.Diagnostics.Debug.Assert(idx >= 0 && idx < capacity);

// bitMap[idx] = true;

remove(idx);

}

}

// 添加数据，构建索引表

public IndexTable WriteData(string data)

{

IndexTable table = new IndexTable();

while (data.Length > Block.capacity)

{

// 找到空闲块索引

int emptyIdx = AllocBlock();

// 保证索引在范围内

System.Diagnostics.Debug.Assert(emptyIdx >= 0 && emptyIdx < capacity);

// 写入数据

blocks[emptyIdx].WriteData(data.Substring(0, Block.capacity));

// 可以直接添加入数据索引

if (!table.DataFull())

{

table.AddDataIndex(emptyIdx);

}

// 加入索引的索引

else

{

// 检查是否添加索引成功

bool flag = false;

// 检查table中的idxIdxList指向的索引块

for (int i = 0; i < table.idxIdxList.Count(); ++i)

{

int idx = table.idxIdxList[i];

// 索引块满了下一个

if (blocks[idx].IdxFull())

{

continue;

}

// 没满继续添加

else

{

blocks[idx].AddIdx(emptyIdx);

flag = true;

break;

}

}

// 没添加成功，申请新的索引块，并加入table的索引列表

if (!flag)

{

int IdxEmptyIdx = AllocBlock();

// 索引块中存储指向数据块的索引

blocks[IdxEmptyIdx].AddIdx(emptyIdx);

// 将索引块的索引添加入table的索引块的索引列表中

System.Diagnostics.Debug.Assert(!table.IdxFull());

table.AddIdxIndex(IdxEmptyIdx);

}

}

data = data.Remove(0, Block.capacity);

}

// 剩下的肯定可以装进一个块

// 找到空闲块索引

int \_emptyIdx = AllocBlock();

// 保证索引在范围内

System.Diagnostics.Debug.Assert(\_emptyIdx >= 0 && \_emptyIdx < capacity);

// 写入数据

blocks[\_emptyIdx].WriteData(data);

// 可以直接添加入数据索引

if (!table.DataFull())

{

table.AddDataIndex(\_emptyIdx);

}

// 加入索引的索引

else

{

// 检查是否添加索引成功

bool flag = false;

// 检查table中的idxIdxList指向的索引块

for (int i = 0; i < table.idxIdxList.Count(); ++i)

{

int idx = table.idxIdxList[i];

// 索引块满了下一个

if (blocks[idx].IdxFull())

{

continue;

}

// 没满继续添加指向数据块的索引

else

{

blocks[idx].AddIdx(\_emptyIdx);

flag = true;

break;

}

}

// 没添加成功，申请新的索引块，并加入table的索引列表

if (!flag)

{

int IdxEmptyIdx = AllocBlock();

// 索引块中存储指向数据块的索引

blocks[IdxEmptyIdx].AddIdx(\_emptyIdx);

// 将索引块的索引添加入table的索引块的索引列表中

System.Diagnostics.Debug.Assert(!table.IdxFull());

table.AddIdxIndex(IdxEmptyIdx);

}

}

return table;

}

}

* 1. **物理空间管理**

索引结构，该类名为Index。它包含了一个属性 idxList（类型为 List<int>）和一个静态属性 capacity（类型为 int）。该类主要用于实现索引功能，通过将索引指向块（block）实现对块的管理。其中，capacity是一个静态属性，表示最多支持的块数上限；idxList是一个列表，用于存储当前索引所指向的块的编号。

AddIdx方法用于向idxList列表中添加一个新的块编号。在添加之前，会先判断idxList列表是否已满，如果已满则无法继续添加，返回false；否则将新的块编号添加到idxList列表中，并返回true。

Full方法用于判断idxList列表是否已满，如果已满则返回true，否则返回false。

public class Index

{

// 设计索引可以最多指向max\_index个块

public List<int> idxList;

public static int capacity = 256;

public Index ()

{

idxList = new List<int>();

}

// 添加索引指向块

public bool AddIdx(int newIdx)

{

// 先判断是否还有位置

if (Full())

{

return false;

}

idxList.Add(newIdx);

return true;

}

// 是否满了

public bool Full()

{

return idxList.Count() >= capacity;

}

}

索引表，该类名为IndexTable 的类，用于存储指向数据块和下一级索引表的索引。每个实例代表一个文件对应的索引表。该类有两个成员变量：dataIdxList 和 idxIdxList，分别表示指向数据块的索引列表和指向下一级索引块的索引列表。其中，dataIdxCapacity 和 idxIdxCapacity 分别表示 dataIdxList 和 idxIdxList 的容量限制。方法AddDataIndex(int idx)：将指向数据块的索引添加到 dataIdxList 中；DataFull()：检查dataIdxList是否已满；AddIdxIndex(int idx)：将指向索引块的索引添加到 idxIdxList 中;IdxFull()：检查 idxIdxList 是否已满;dxFull()：检查 idxIdxList 是否已满

// 一个文件对应一个索引表

// 存储指向数据的以及下一级索引表的索引

public class IndexTable

{

// 指向数据块的索引列表

public List<int> dataIdxList;

public static int dataIdxCapacity = 10;

// 指向索引块的索引列表

public List<int> idxIdxList;

public static int idxIdxCapacity = 3;

public IndexTable()

{

// 先只创建指向数据块的索引

dataIdxList = new List<int>(dataIdxCapacity);

idxIdxList = new List<int>();

}

[JsonConstructor]

public IndexTable(List<int> dataIdxList, List<int> idxIdxList)

{

this.dataIdxList = dataIdxList;

this.idxIdxList = idxIdxList;

}

public void AddDataIndex(int idx)

{

dataIdxList.Add(idx);

}

public bool DataFull()

{

return dataIdxList.Count() >= dataIdxCapacity;

}

public void AddIdxIndex(int idx)

{

idxIdxList.Add(idx);

}

public bool IdxFull()

{

return idxIdxList.Count() >= dataIdxCapacity;

}

// 返回所有数据块、索引块索引

public List<int> GetAllIndex()

{

if (idxIdxList.Count() != 0)

{

return dataIdxList.Concat(idxIdxList).ToList<int>();

}

else

{

return dataIdxList;

}

}

}

* 1. **目录结构管理**

目录管理符号目录项，该类名为SymFCB，用于存储符号目录项的信息，每个SymFCB实例代表一个文件或文件夹。其中包含：fileName文件或文件夹名称、fileType文件类型、fileId文件或文件夹的唯一标识，由静态变量fileCount分配、father指向父级 SymFCB 对象的引用，如果该对象是根节点，则为 null，children包含子结点SymFCB对象的列表。方法包括构造函数；AddChild：将一个 SymFCB 对象作为子节点添加到当前对象的 children 列表中，并设置其 father 属性为当前对象； SymFCB 对象作为子节点添加到当前对象的 children 列表中，并设置其 father 属性为当前对象；removeChild:移除某一个子节点。

// 存储树形结构

public class SymFCB

{

public static int fileCounter = 0;

public string fileName;

public string fileType;

public int fileId;

public SymFCB father;

public List<SymFCB> children;

public SymFCB()

{

fileType = "folder";

fileId = fileCounter++;

father = null;

children = new List<SymFCB>();

}

public SymFCB(string fileName, string fileType)

{

this.fileName = fileName;

this.fileType = fileType;

fileId = fileCounter++;

father = null;

children = new List<SymFCB>();

}

[JsonConstructor]

public SymFCB(string fileName, string fileType, int fileId, SymFCB father, List<SymFCB> children)

{

this.fileName = fileName;

this.fileType = fileType;

this.fileId = fileId;

this.father = father;

this.children = children;

}

public void AddChild(SymFCB child)

{

children.Add(child);

child.father = this;

}

public void removeChild(SymFCB sf)

{

children.Remove(sf);

}

* 1. **数据存储管理**

加载本地数据，首先检查路径中有无需要的文件，确保每个文件都有了才进行载入

private void loadToolStripMenuItem\_Click(object sender, EventArgs e)

{

// 先查看当前目录下有无需要的文件 转换为文件名

var jsonFileList = new List<string>(Directory.GetFiles(curPath, "\*.\*").Where(s => s.EndsWith(".dat") || s.EndsWith(".txt")).Select(Path.GetFileName));

//var jsonFileList = Directory.GetFiles(curPath, "\*.json").ToList();

string[] targetFile = new string[] { "fileDict.dat", "rootSymFCB.dat", "manager.dat", "fileCount.txt" };

foreach (string file in targetFile)

{

// 不包含任何一个目标函数

if (!jsonFileList.Contains(file))

{

// 弹出窗口提示并退出

MessageBox.Show("TARGET FILE NOT FOUND!", "WARNING");

return;

}

}

// 加载文件

LoadData();

// 检查循环引用

CheckFather();

// 更新界面

UpdateView();

}

* 1. **主窗口管理**

主窗口中调用上述数据结构，从而实现本项目中的文件管理系统。属性如下：

// 修改标识

bool changed;

// 当前symfcb

public SymFCB curSymFCB;

// 根symfcb

public SymFCB rootSymFCB;

// 字典fileId -> fcb的映射

public Dictionary<int, Pair> fileDict;

// 管理物理块

public Manager manager;

// 当前路径

public string curPath = Path.GetDirectoryName(Path.GetDirectoryName(Directory.GetCurrentDirectory()));

// 文件显示窗口

private Dictionary<int, ListViewItem> listTable;

// 文件树根结点

TreeNode rootNode;

// 保存前进方向

Stack<SymFCB> fileStack;

下面将挑选出几个重要的函数进行分析。

·检查重名函数：检查当前目录下是否存在与给定文件同名的文件或文件夹。如果存在同名，则在给定文件名后面添加括号和序号，从而生成新的文件名，防止重名。

具体来说，这个方法接受两个参数：fileName表示文件名，ext表示文件扩展名（如果有）。该方法首先定义了一个列表sameNameFile，用于记录同名文件的数量。然后遍历当前目录下的子节点（即文件和文件夹），对于每个子节点，它会判断其文件类型是否与指定扩展名相同（如果有），以及文件名是否与指定文件名相同。如果满足条件，它会使用正则表达式去掉子节点文件名中最后一个括号及其之后的内容，然后再去掉文件扩展名，得到子节点的基本文件名。如果子节点的基本文件名与指定文件名相同，则说明存在同名文件，此时它会使用正则表达式获取子节点文件名中最后一个圆括号中的数字，将数字转换为下标并标记为true。

接着，它会根据已经标记的同名文件数量以及当前目录下的子节点数量，找到第一个未标记的数字作为括号序号，将其添加到文件名后面，形成新的文件名。最后，它会将文件扩展名添加到新的文件名中（如果有扩展名），并返回新的文件名。

private string CheckSameName(string fileName, string ext="")

{

// 列表记录同名的共多少个

List<int> sameNameFile = new List<int>();

// 在当前目录下找是否有重名

foreach (SymFCB child in curSymFCB.children)

{

// 文件类型不同

if (!((child.fileType == "folder" && ext == "") || (child.fileType == ext)))

{

continue;

}

// 去掉最后一个圆括号及其之后的内容，[^\(]\* 表示任意不是左括号的字符可以出现 0 次或多次

string childFileName = Regex.Replace(child.fileName, @"\(\d+\)[^(\(\d+\))]\*$", "");

// 去掉后缀

childFileName = Regex.Replace(childFileName, String.Format(@"\.{0}", ext), "");

if (childFileName == fileName)

{

// 获得当前最后一个圆括号，匹配.之前的数字和圆括号

Match match1;

if (child.fileType == "folder")

match1 = Regex.Match(child.fileName, @"\(\d+\)$", RegexOptions.RightToLeft);

else

match1 = Regex.Match(child.fileName, @"\(\d+\)\.", RegexOptions.RightToLeft);

// 没有匹配的第0个

if (!match1.Success)

{

sameNameFile.Add(0);

continue;

}

// 获取括号中的数字

Match match2 = Regex.Match(match1.Value, @"\d+");

// 数字转换为下标

int idx = int.Parse(match2.Value);

// 标记为true

sameNameFile.Add(idx);

}

}

// 出现重名

for (int i = 0; i < curSymFCB.children.Count + 1; ++i)

{

if (sameNameFile.Contains(i))

continue;

// 找第一个缺的数字，i == 0就不用增加

if (i != 0)

fileName += "(" + i.ToString() + ")";

break;

}

return fileName + ((ext == "") ? "" : ("." + ext));

}

·读取文件数据：通过FCB中的索引表中的所有索引，然后遍历这些索引找到所指向的物理块，检查是数据块还是索引块。若是数据块直接将其中的数据读出，若是索引块便将其中的所有索引读出，通过二级索引继续寻找其指向的数据块，读取数据。

private void ShowText()

{

// 得到该文件的所有索引

List<int> indexList = bf.indexTable.GetAllIndex();

string text = "";

foreach (int idx in indexList)

{

// 数据块

if (manager.GetBlock(idx).GetIndex().Count == 0)

{

text += manager.GetBlockData(idx);

}

else

{

// 读索引块中索引对应数据块

foreach (int index in manager.GetBlock(idx).GetIndex())

{

text += manager.GetBlockData(index);

}

}

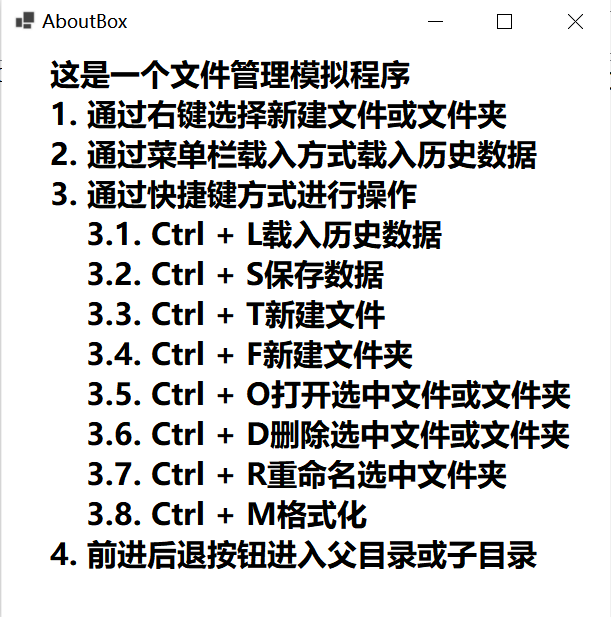
}

// 设置到文本

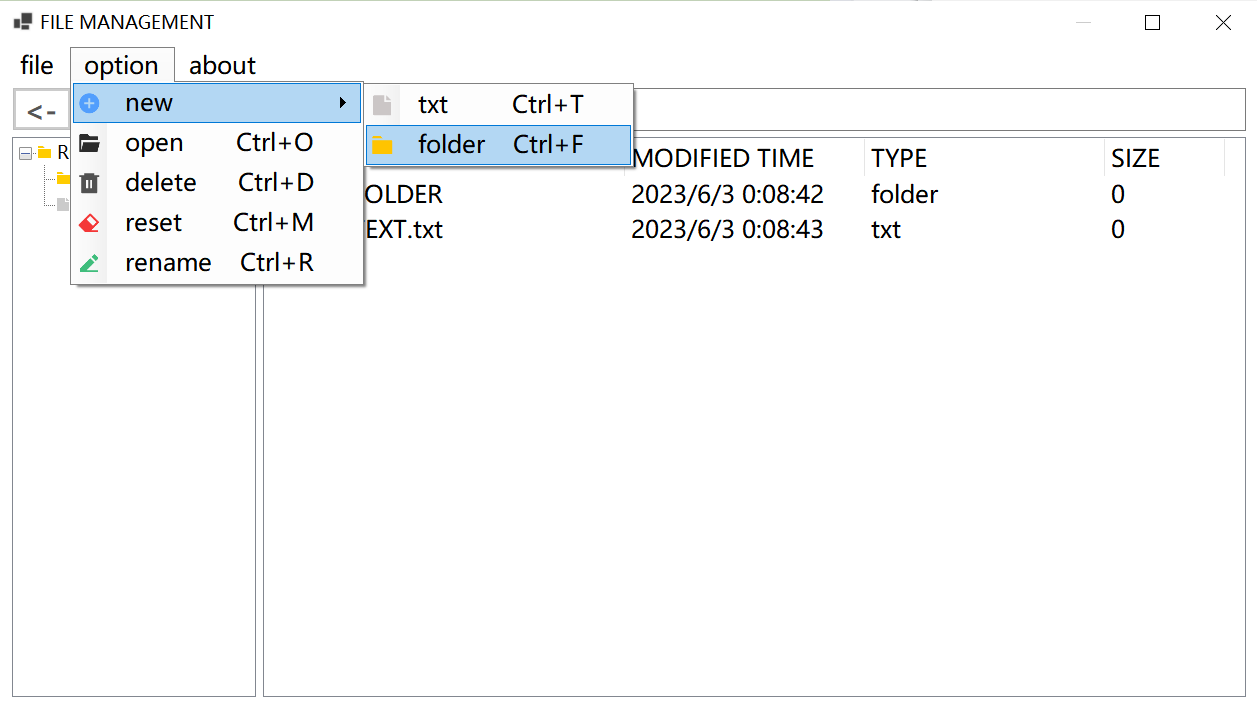
textBox.Text = text;

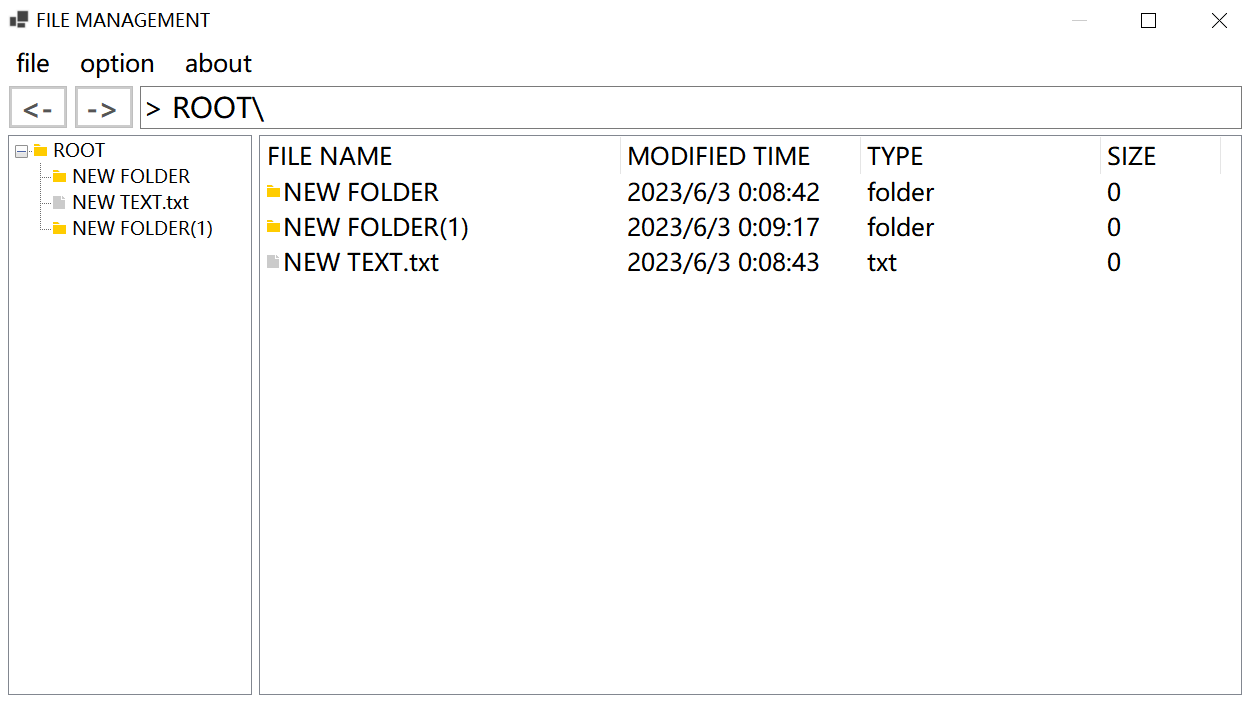
}

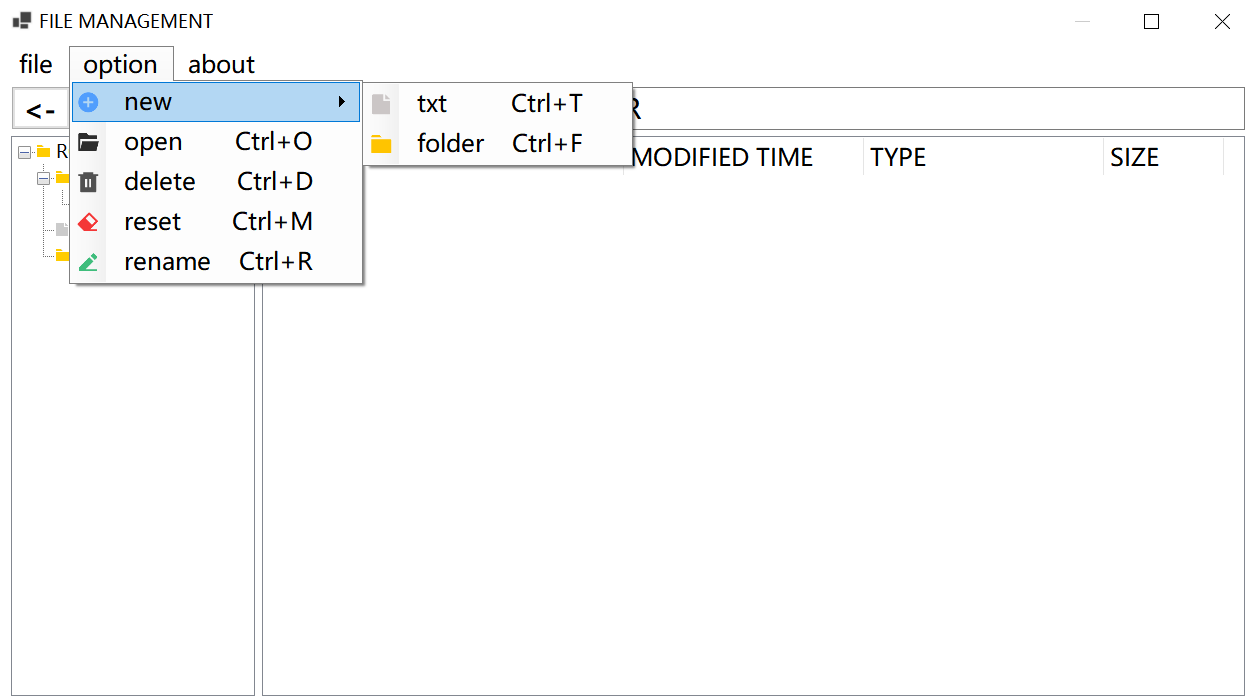
1. **功能展示**
   1. **本项目相关信息**

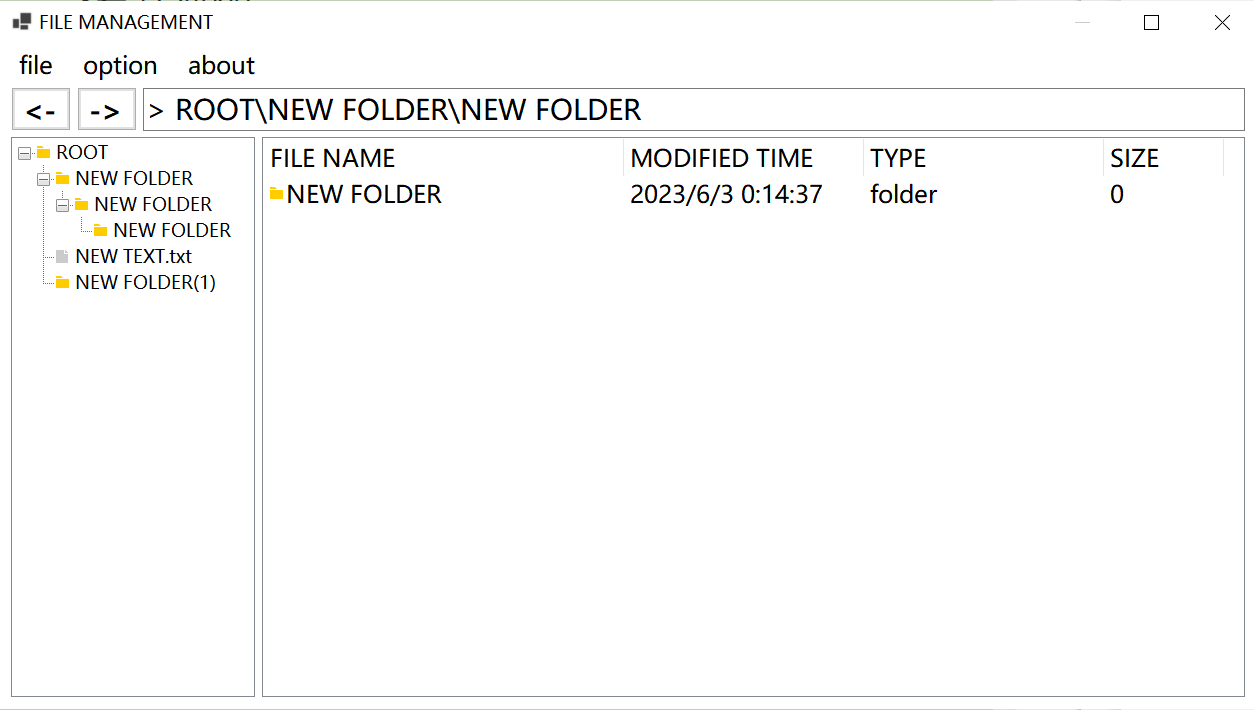


* 1. **新建**

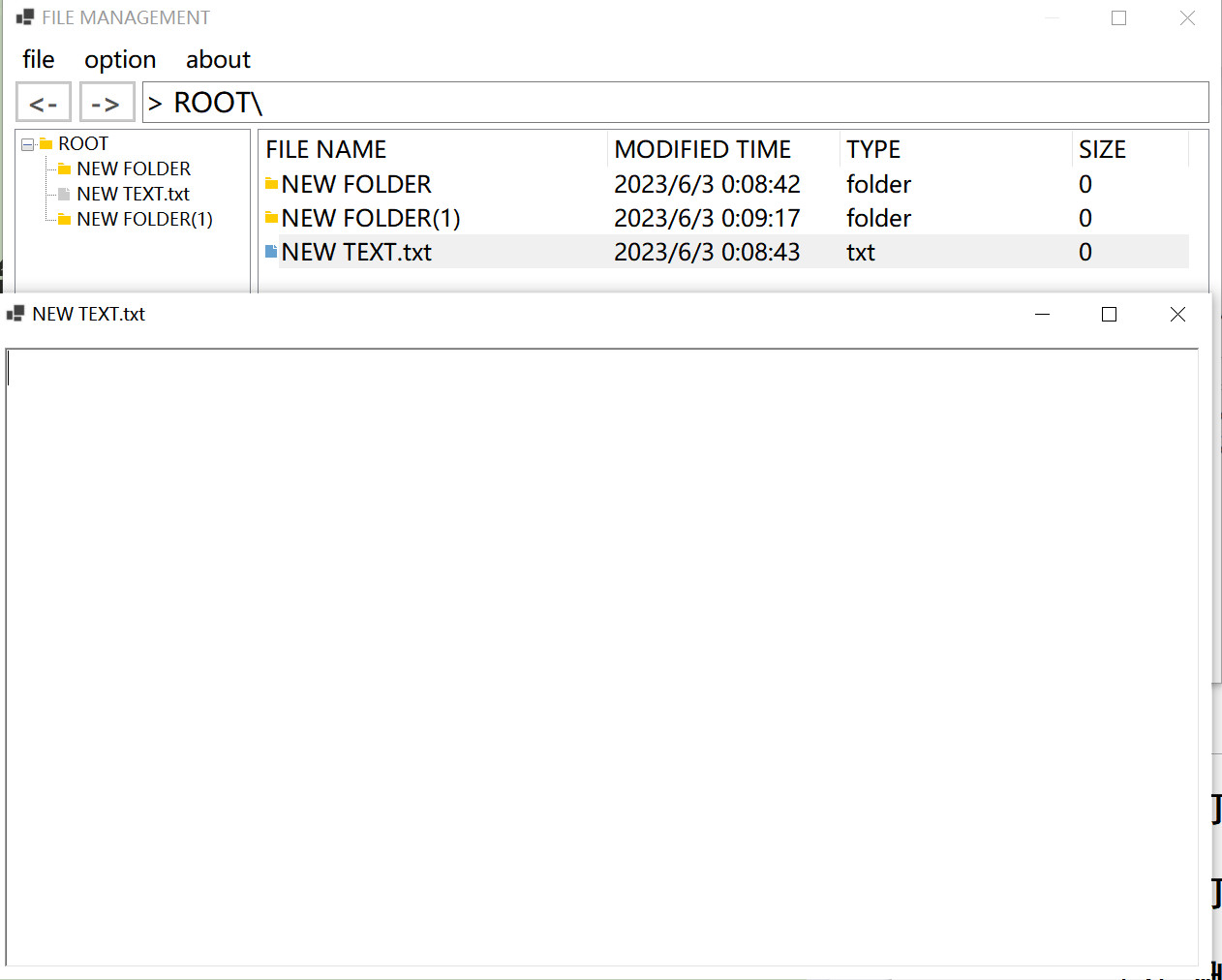




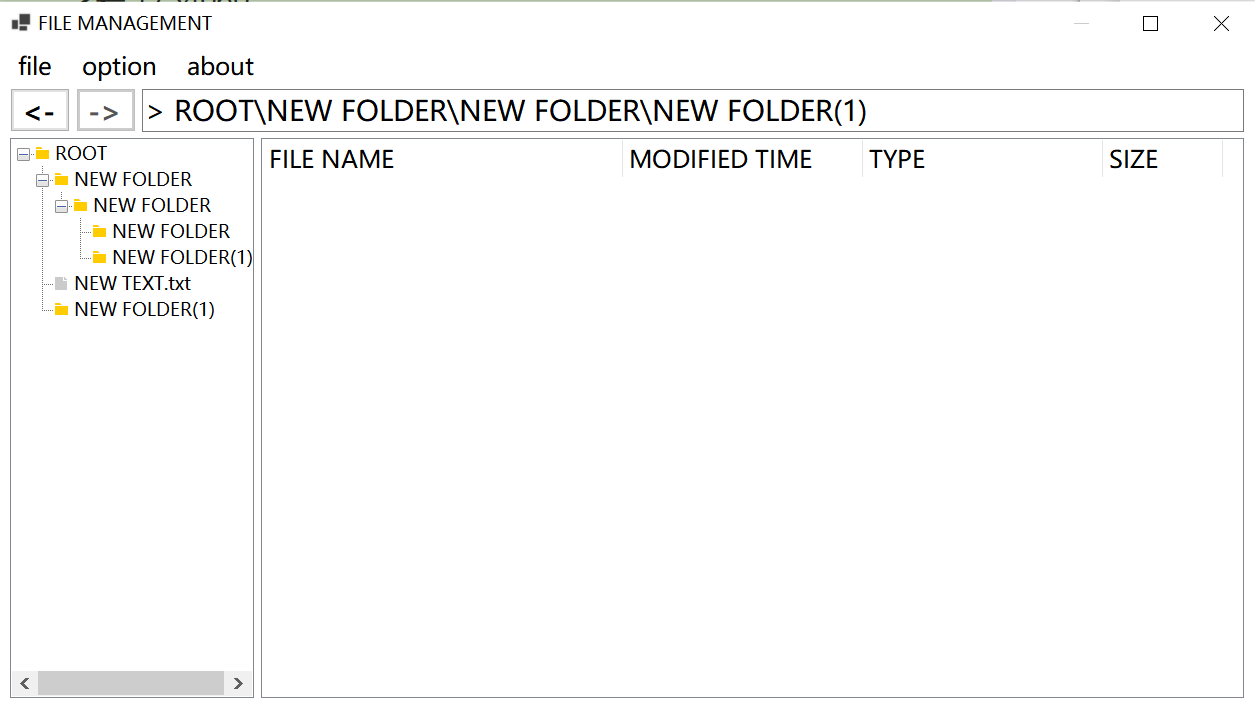




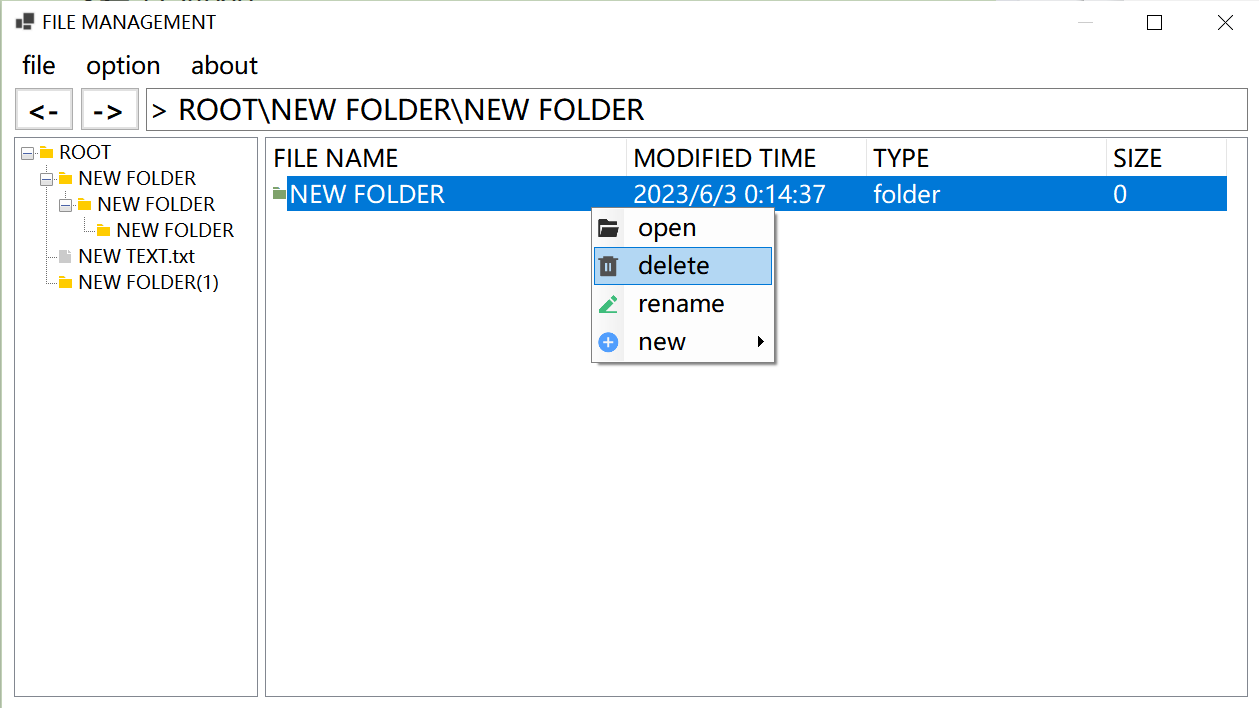
* 1. **打开文件**

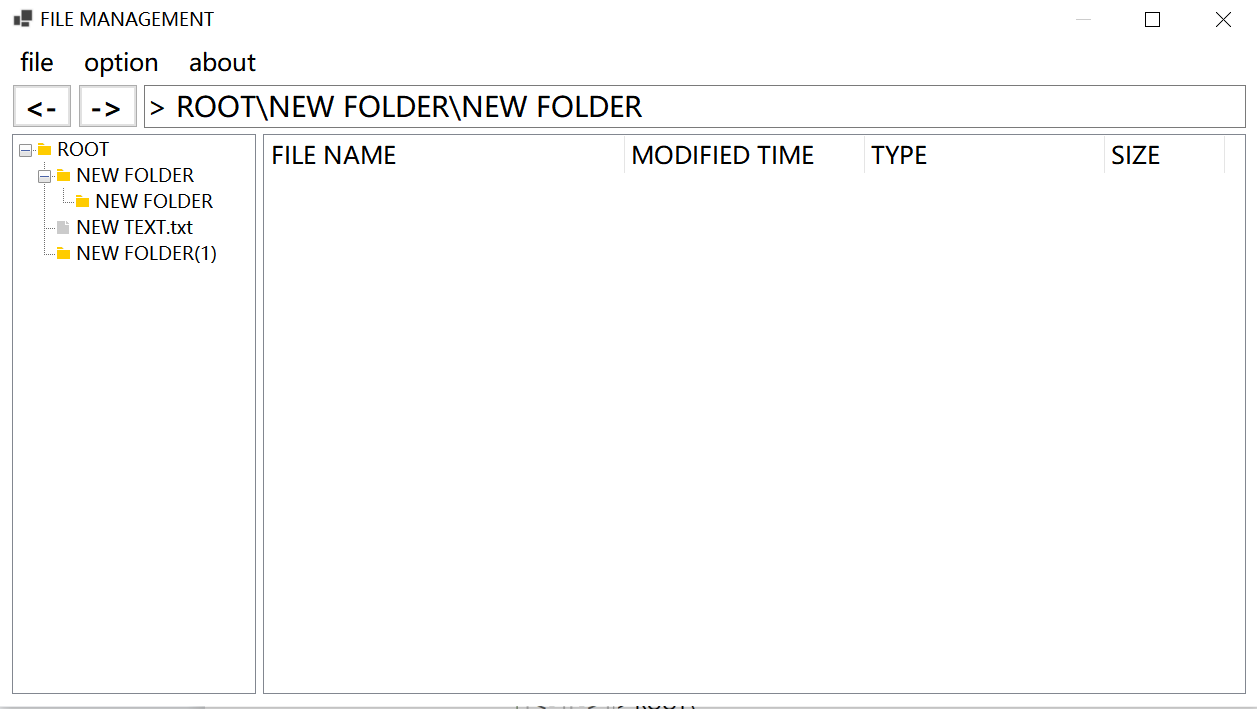


* 1. **打开文件夹**

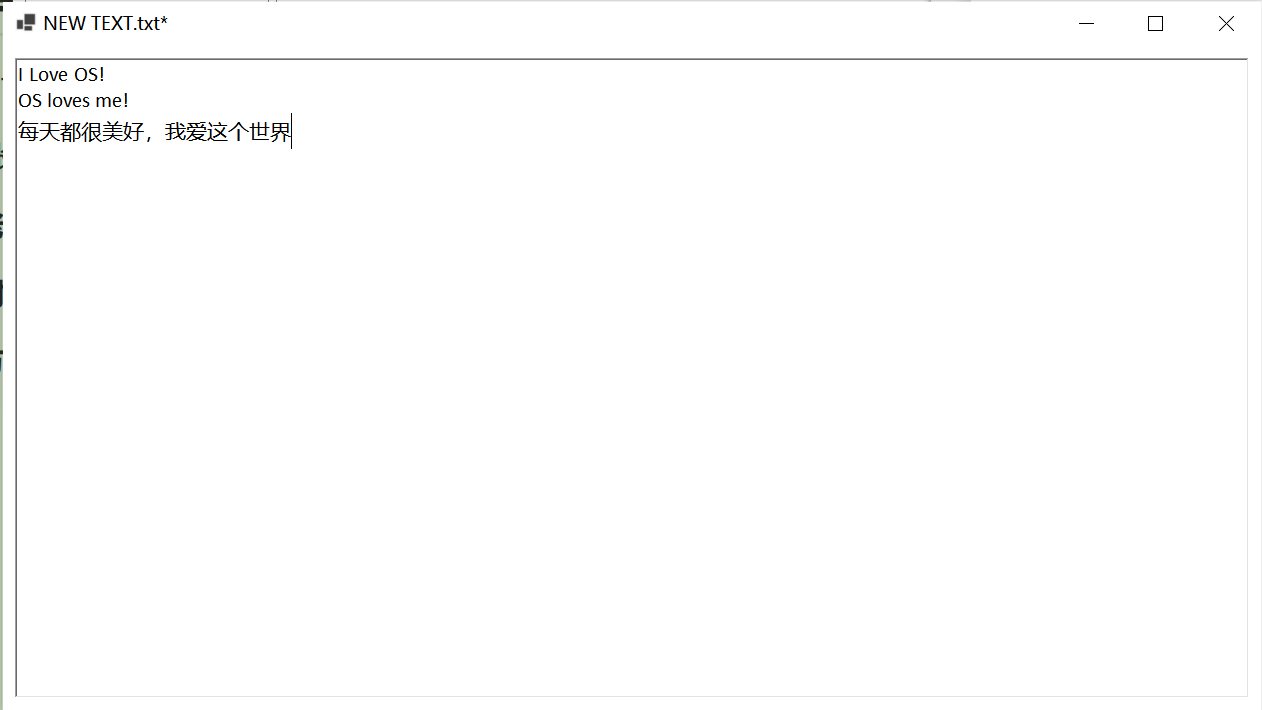


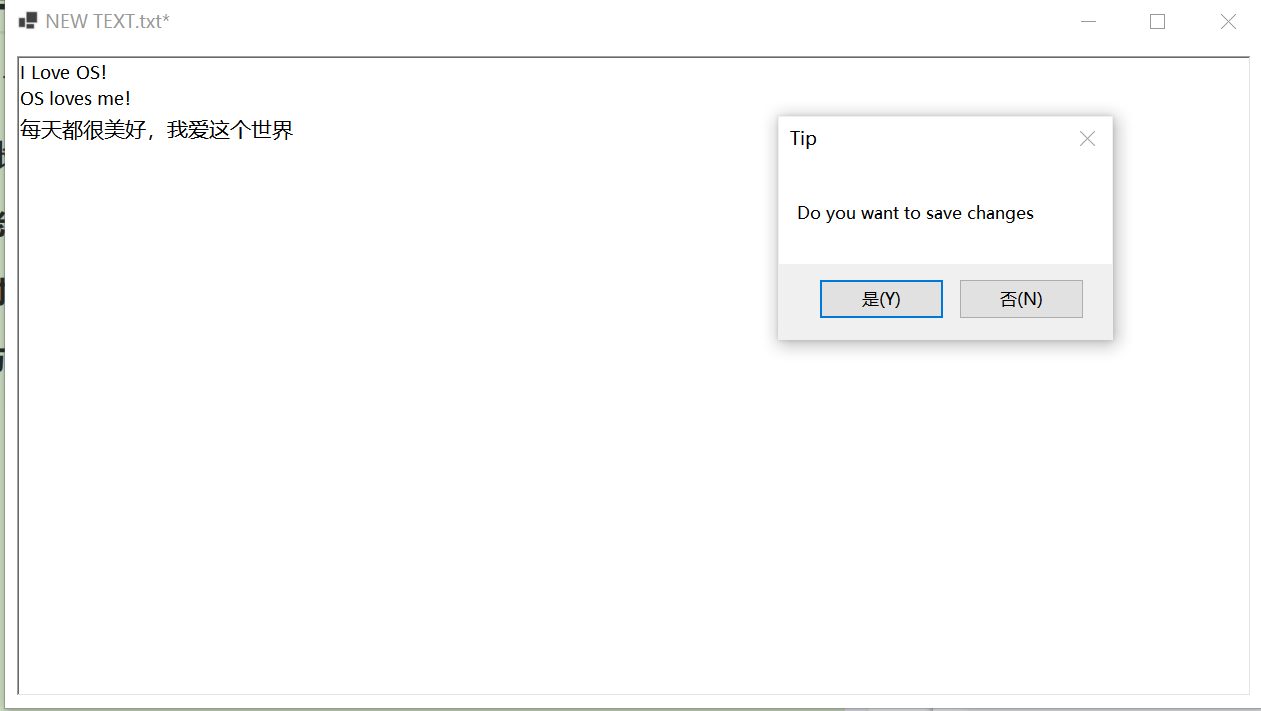
* 1. **删除**

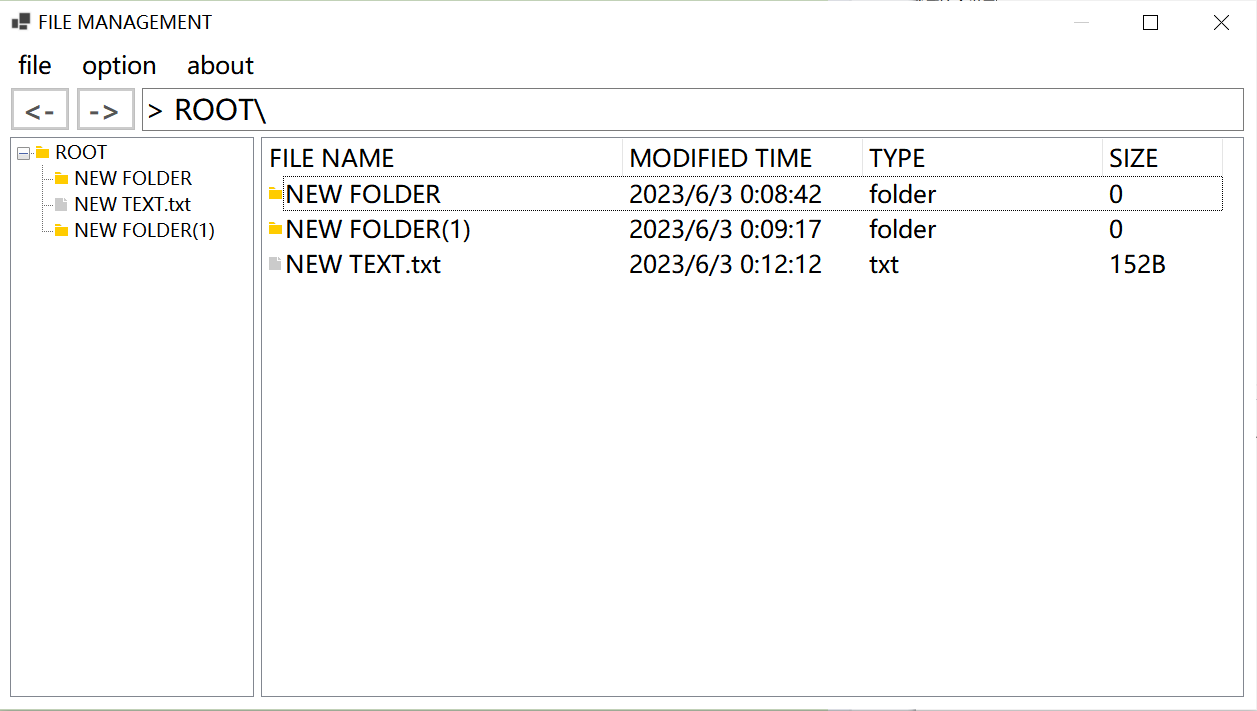




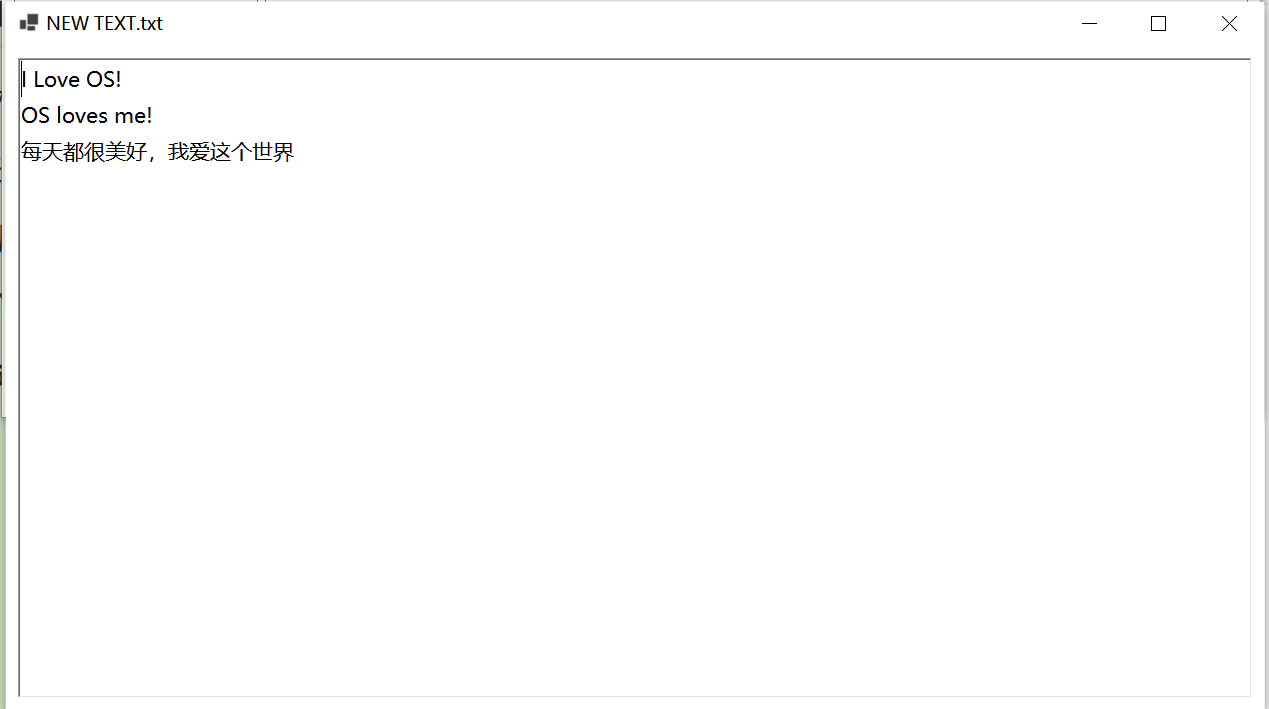
* 1. **编辑并保存文件**



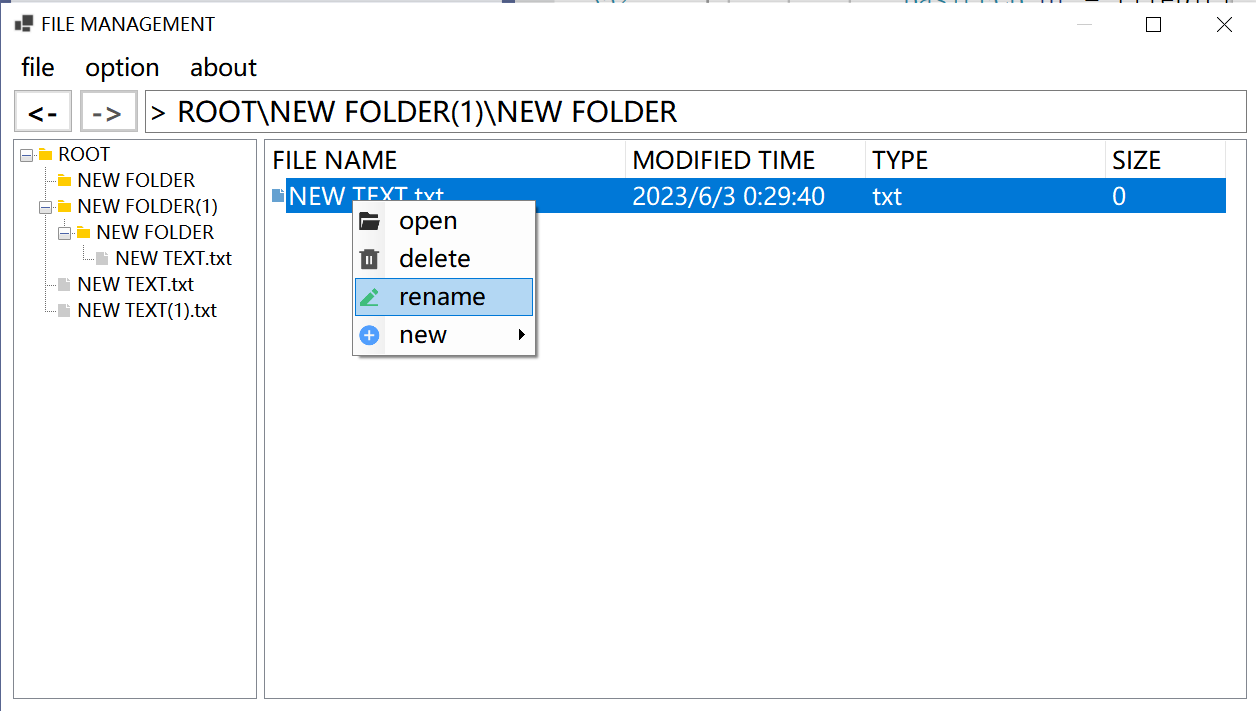


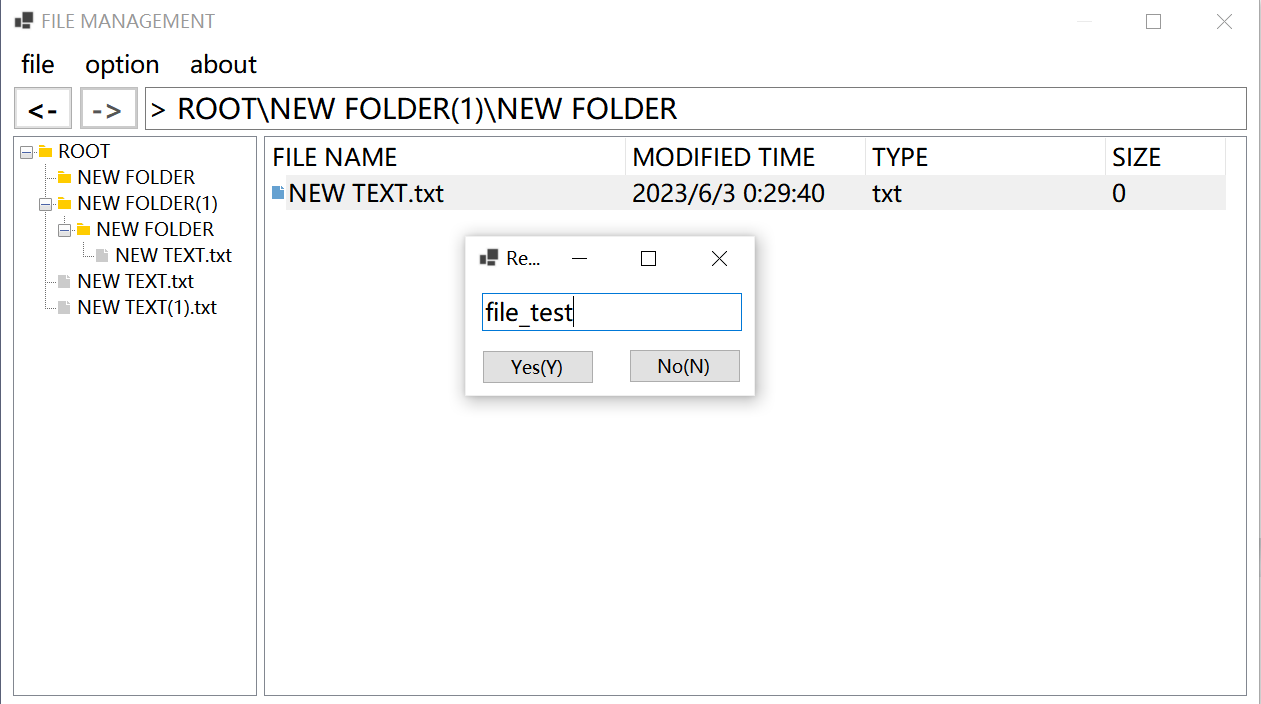


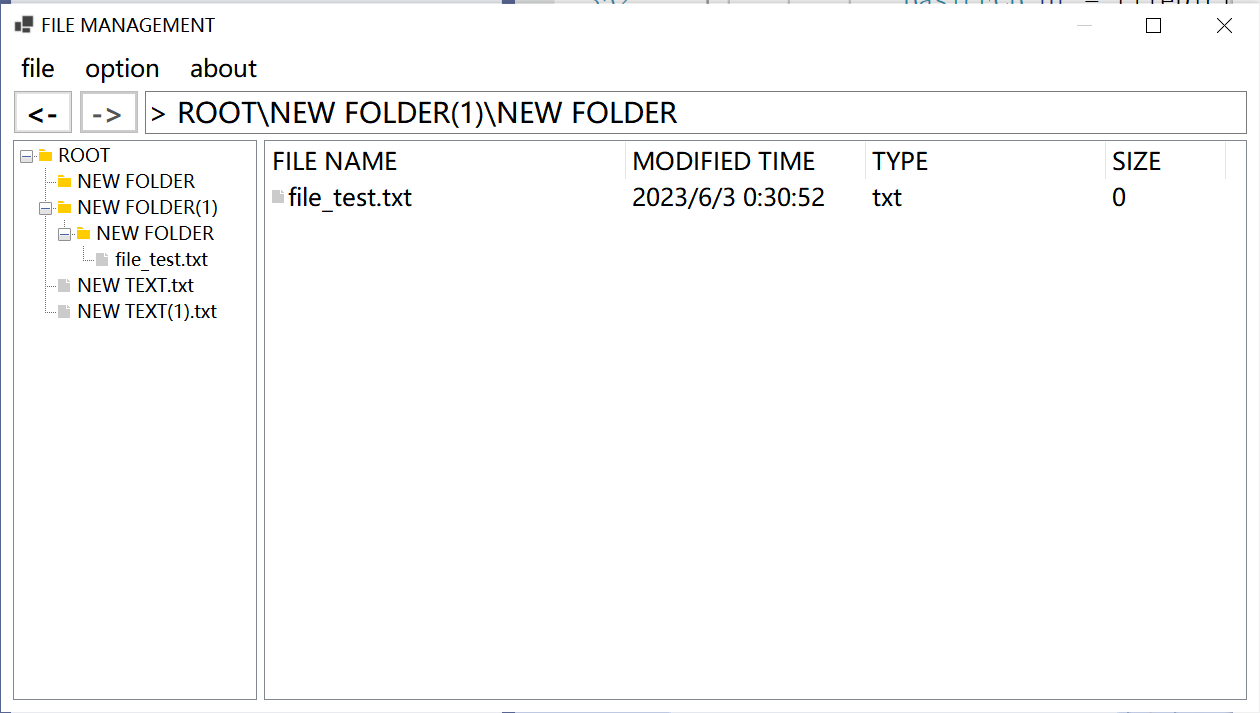
* 1. **再次打开编辑文件**



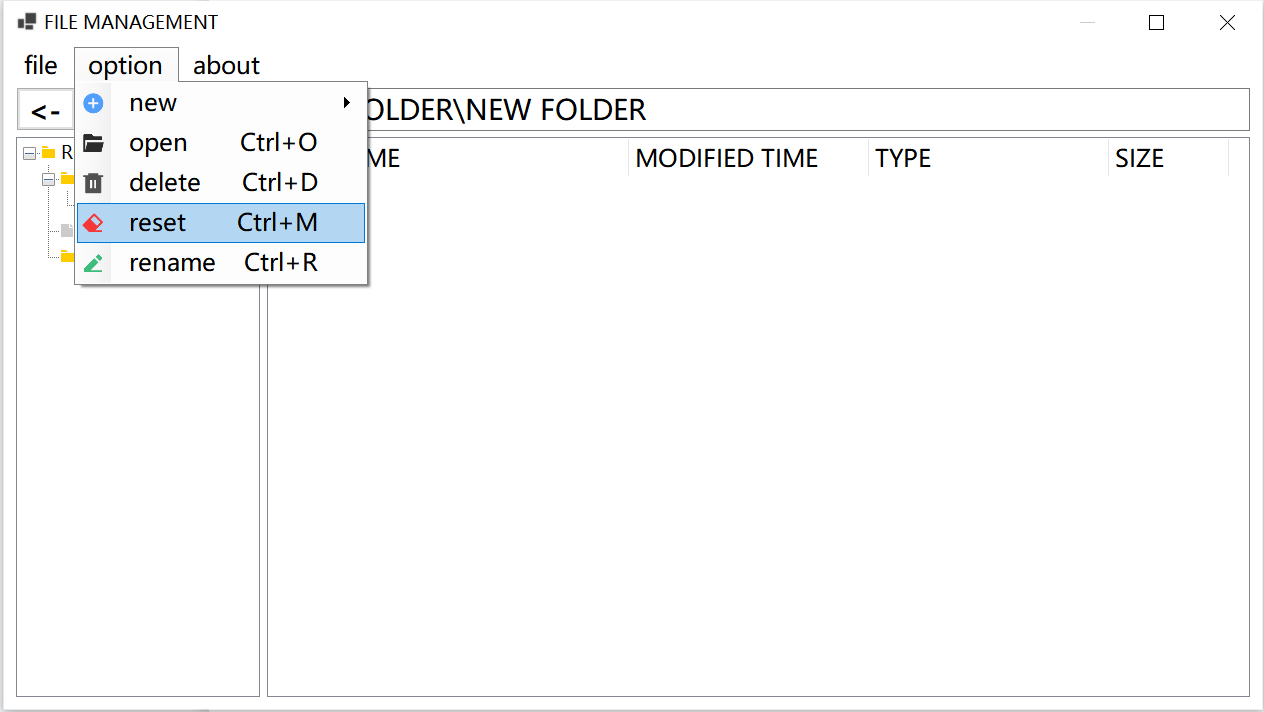
* 1. **重命名**

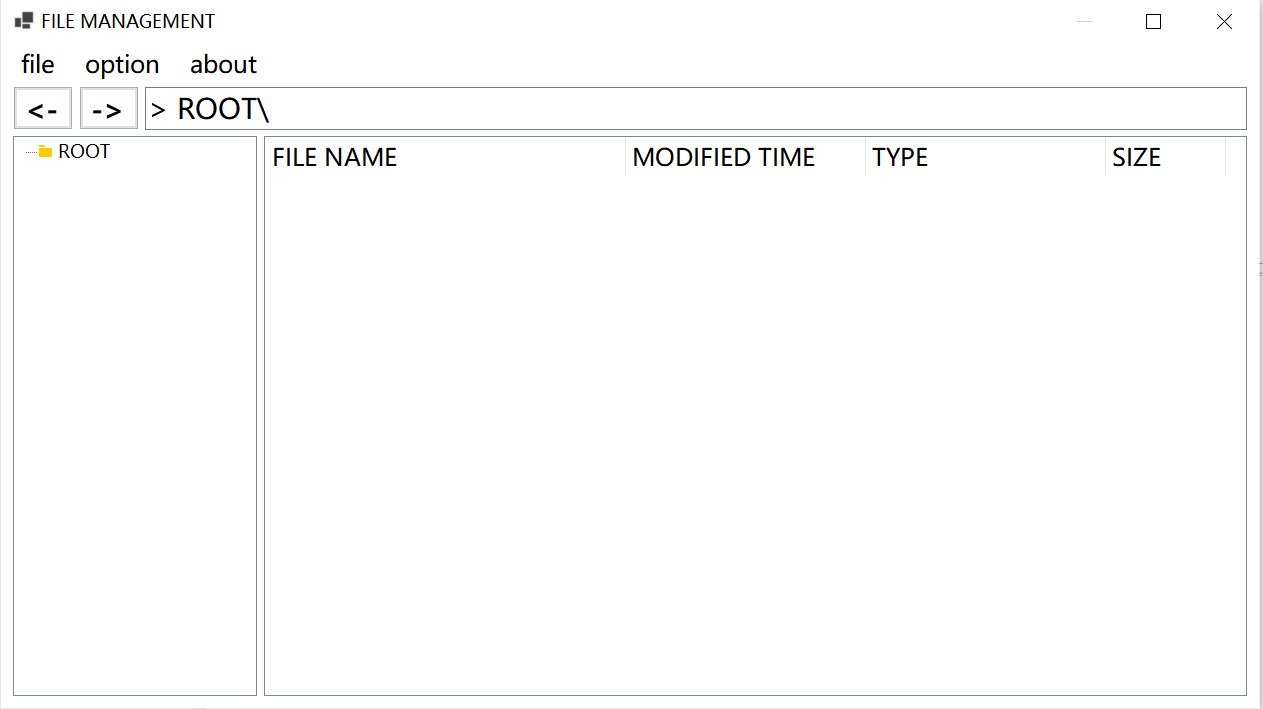




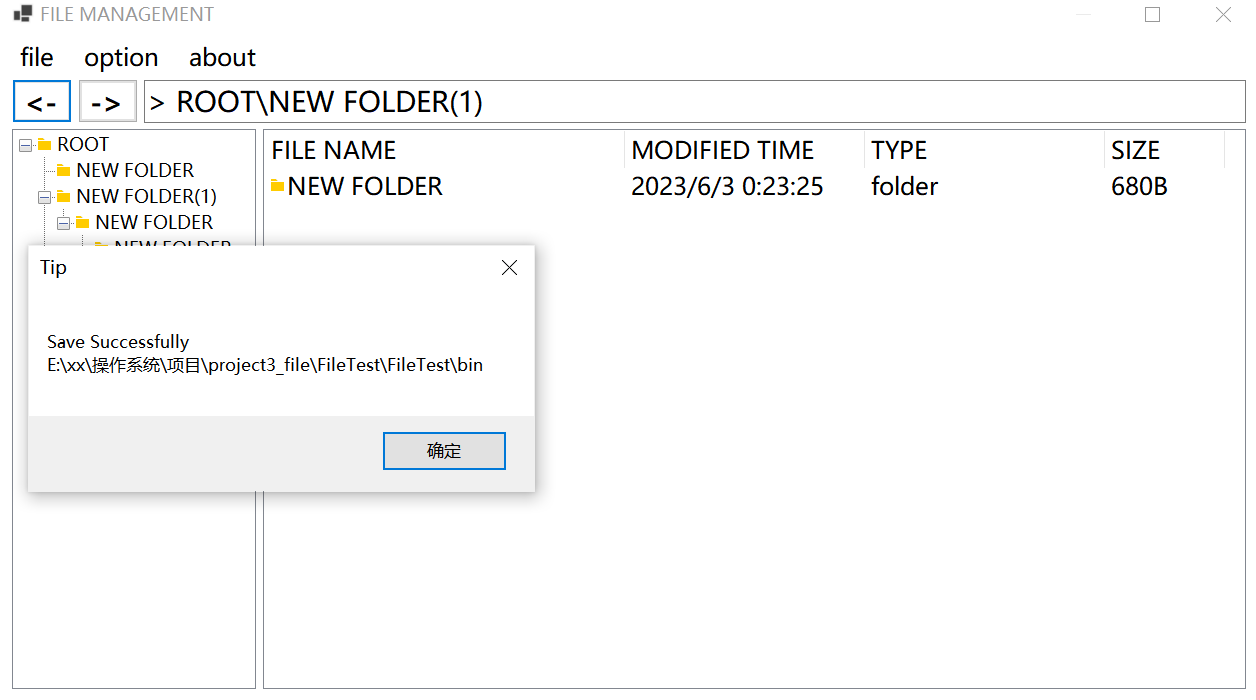


* 1. **格式化**





* 1. **保存数据**



* 1. **载入数据**

