Middle East Technical University | Department of Computer Engineering

CENG 310

DATA STRUCTURES

Fall 2023

Programming Assignment 1 - Simple File System Implementation via Linked Lists

Due: 6. November 2023 23:55

Submission: **via CengClass**

1 Objectives

In this programming assignment, you will first implement a ***doubly linked list with a dummy head node***. Then, using the linked list implementation, you will build a very simplified file system that supports a single directory with multiple files. The details of the assignment are explained in the following sections.

Keywords: C++, Data Structures, Linked List, Doubly Linked List with Dummy Nodes, Files, Directories.

2 Linked List Implementation

The linked list data structure used in this assignment is implemented as the class template LinkedList with the template argument T, which is used as the type of data stored in the nodes. The node of the linked list is implemented as the class template Node with the template argument T, which is the type of data stored in nodes. Node class is the basic building block of the LinkedList class. LinkedList class has a Node pointer in its private data field (namely head) which points to the dummy node of the linked list. It also has an int that keeps the number of nodes (apart from the dummy node) in the list, called size.

The LinkedList class has its definition and implementation in *LinkedList.h* file and the Node class has its definition and implementation in *Node.h* file.

2.1 Node Class

Node class represents nodes that are in the linked lists. A node instance keeps two pointers, called *prev* and *next*, that points to nodes that are previous and next nodes compared to the current node. It also keeps a data variable of type T, called *data*, to hold the data. The class has two constructors and the overloaded output operator (operator«) which are already implemented for you. You **must not** change anything inside of the *Node.h* file.

2.2 LinkedList Class

2.2.1 LinkedList()

This is the default constructor. You should make necessary initialization(s) in this function. Don't forget that our list will be a **doubly linked list with a dummy head node**.

2.2.2 LinkedList(const LinkedList &rhs)

This is the copy constructor. You should make necessary initializations and copy the nodes from rhs to the linked list accordingly.

2.2.3 ~LinkedList()

This is the destructor. You must deallocate all the memory that you had allocated before.

2.2.4 LinkedList<T> &operator=(const LinkedList &rhs)

This is the overloaded assignment operator. You should remove all the nodes in the linked list (if any) and then you should copy the nodes from rhs to the linked list accordingly.

2.2.5 int getSize() const

This function returns the number of nodes in the linked list (apart from the dummy node).

2.2.6 bool isEmpty() const

This function returns whether the linked list is empty or not. The linked list is empty if the only node present in the linked list is the dummy node. It returns true if the linked list is empty, otherwise it should return false.

2.2.7 bool containsNode(Node<T> *node) const

This function should return true if the linked list contains the given node (i.e., any prev/next in the nodes of the linked list matches with node). Otherwise, it should return false.

2.2.8 int getIndex(Node<T> *node) const

This function should return the index of the given node in the linked list. If the node is not in the list, it should return -1 instead. **Note that dummy node does not count as a node**. Therefore the first node (i.e., index 0) in the list is the next node of dummy node.

2.2.9 Node<T> *getFirstNode() const

This function should return a pointer to the first node in the linked list. If the linked list is empty, it should return NULL.

2.2.10 Node<T> *getLastNode() const

This function should return a pointer to the last node in the linked list. If the linked list is empty, it should return NULL.

2.2.11 Node<T> *getNode(const T &data) const

You should search the linked list for the node that has the same data with the given data and return a pointer to that node. You can use operator== to compare two T objects. If there exists multiple such nodes in the linked list, return a pointer to the first occurrence. If there exists no such node in the linked list, you should return NULL.

2.2.12 Node<T> *getNodeAtIndex(int index) const

You should search the linked list for the node at the given zero-based index (i.e., index=0 means the first node, index=1 means the second node, ..., index=size-1 means the last node) and return a pointer to that node. If there exists no such node in the linked list (i.e., index is out of bounds), you should return NULL.

2.2.13 void append(const T &data)

You should create a new node with the given data and insert it at the end of the linked list as the last node. Don't forget to make necessary pointer and size modifications.

2.2.14 void prepend(const T &data)

You should create a new node with the given data and insert it at the front of the linked list as the first node (after the dummy node). Don't forget to make necessary pointer and size modifications.

2.2.15 void insertAfterNode(const T &data, Node<T> *node)

You should create a new node with the given data and insert it after the given node as its next node. Don't forget to make necessary pointer and size modifications. If the given node is not in the linked list, do nothing.

2.2.16 void insertAtIndex(const T &data, int index)

You should create a new node with the given data and insert it at the given index (i.e., index=0 means the first node, index=1 means the second node, ..., index=size-1 means the last node). Don't forget to make necessary pointer and size modifications. If there exists no such index in the linked list (i.e., index is out of bounds), you should not insert the element.

2.2.17 void moveToIndex(int currentIndex, int newIndex)

This function should move the node at the currentIndex to newIndex. For this function, you are not allowed to just change the data in the given nodes. Also, you are not allowed to create new nodes. Do the moving by changing the pointers in the nodes of the linked list. If the newIndex is greater than the number of nodes in the list, then the node should be moved to the end of the list. If the currentIndex is greater than the number of nodes in the list, then do nothing.

2.2.18 void mergeNodes(int sourceIndex, int destIndex)

This function should add/concatenate the data of the node in the sourceIndex to the node in the destIndex and then delete the node in the sourceIndex. If either sourceIndex or destIndex is out of bounds (i.e., $0 >$ sourceIndex or sourceIndex $> size - 1$) then do nothing. **Note that this operation is only supported by data types that supports operator+. For string data type this would correspond to concatenation**.

2.2.19 void removeNode(Node<T> *node)

You should remove the given node from the linked list. Don't forget to make necessary pointer and size modifications. If the given node is not in the linked list (i.e., the linked list does not contain the given node), do nothing.

2.2.20 void removeNode(const T &data)

You should remove the node that has the same data with the given data from the linked list. Don't forget to make necessary pointer, head and size modifications. If there exists multiple such nodes in the linked list, remove all occurrences. If there exists no such node in the linked list, do nothing.

2.2.21 void removeNodeAtIndex(int index)

You should remove the node at the given index from the list. Don't forget to make necessary pointer and size modifications. If there exists no such index in the linked list (i.e., index is out of bounds), or if the list is empty, you should not do anything.

2.2.22 void removeAllNodes()

You should remove all nodes (apart from the dummy node) in the linked list so that the linked list becomes empty. Don't forget to make necessary pointer and size modifications.

2.2.23 void print() const

This function print the linked list to the standard output. It is provided for you.

3 File System Implementation

3.1 Block

The Block class represents a variable sized block from the memory. Each block object contains a std::string content and size_t size member variables. Its constructors, operator«, operator== and other accessor/-modifier functions are already implemented for you.

3.1.1 Block &operator+(const Block &rhs)

This is the only function that you need to implement in the Block class. This function should append the content of rhs to content of current Block object and arrange the size accordingly.

3.2 File

The File class represents a file in the directory. This class has a single private member variable called LinkedList<Block> blocks which keeps the blocks of the file. A file is made up by the blocks that it contains and the contents of the file can be obtained by sequential traversal of Blocks from the LinkedList object blocks. Public methods of the File class are explained below.

3.2.1 Block getBlock(int index) const

This function should return the block at the given index. If there is no block at the given index it must return an empty Block.

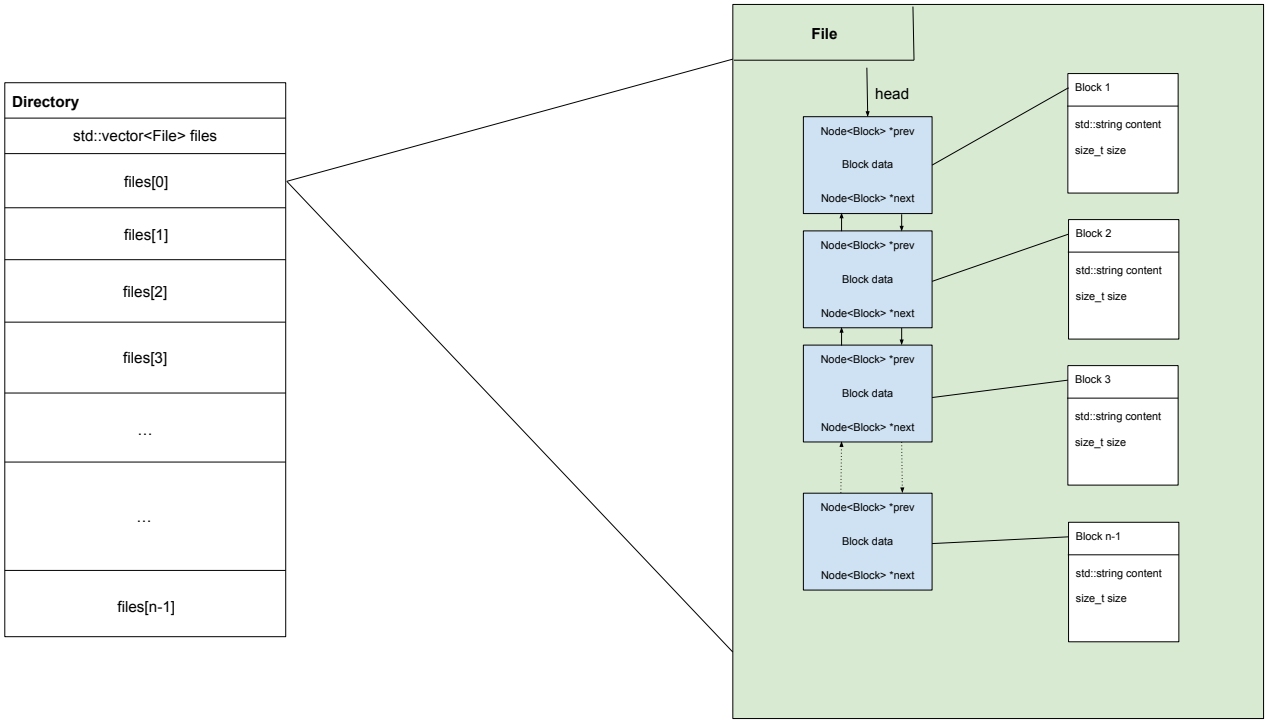

Figure 1: Organization of Blocks, Files and Directory

3.2.2 size_t fileSize() const

This function should return the cumulative size of the blocks resident in the File. If there are no Blocks in the File then you must return 0.

3.2.3 int numBlocks() const

This function should return the number of blocks in the File.

3.2.4 bool isEmpty() const

This function should return true if File is empty. Otherwise returns false.

3.2.5 bool operator==(const File &rhs) const

This function checks whether rhs is equal to the current object. **Note that you need to not only check the contents of the files but also the number of blocks resident in each file.**

3.2.6 void newBlock(const Block &block)

This function adds a new block at the end of the file.

3.2.7 void removeBlock(int index)

This function should remove the Block at the given index. If there exists no such index in the File, then do nothing.

3.2.8 void mergeBlocks(int sourceIndex, int destIndex)

This function should append the block given at sourceIndex into the block given at destIndex and then delete the block at sourceIndex. If any of the given indices are out of bounds do nothing.

3.2.9 void printContents() const

This function prints the contents of the file together with its size to the standard output. It is already implemented.

3.3 Directory

This class implements the simple directory structure. It has only a single private member variable called files which is of the std::vector<File>. While implementing this class you can use std::vector class's member functions. Public methods of Directory class are explained below.

3.3.1 File getFile(int index) const

This function returns the file at given index. If index is out of bounds then return an empty File.

3.3.2 size_t directorySize() const

This function should return the cumulative size of the Files resident in the Directory. If there are no Files in the Directory then you must return 0.

3.3.3 bool isEmpty() const

This function should return true if Directory is empty. Otherwise returns false.

3.3.4 size_t numFiles() const

This function should return the number of files in the Directory.

3.3.5 void newFile(const File &file)

This function adds a new file at the end of the Directory.

3.3.6 void removeFile(int index)

This function removes the file given at index. If given index is out of bounds do nothing.

3.3.7 void removeFile(const File &file)

This function removes the first occurrence of file from the directory. If file is not in the directory do nothing.

3.3.8 void removeBiggestFile()

This function should find the biggest file in the directory and remove it from the directory. In case of multiple files with biggest size you should delete the file with the smallest index. If the directory is empty do nothing.

4 Driver Programs

To enable you to test your LinkedList and Directory implementations, two driver programs, *main_ linkedlist.cpp* and *main_ directory.cpp* are provided.

5 Regulations

1. You will use C++ Programming Language.
2. Usage of Standard Template Library other than std::vector is not allowed.
3. External libraries other than the ones already provided is prohibited.
4. Those who do the operations (insert, remove, get) without utilizing the linked list will receive 0 grade.
5. Those who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive 0 grade.
6. Options used for g++ are "-ansi -Wall -pedantic-errors -O0". They are already included in the provided Makefile.
7. You can add private member functions whenever it is explicitly allowed.
8. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30 pm, you could use 2 late days to submit on Saturday by 11:30 pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
9. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
10. **Newsgroup:** You must follow ODTUClass for discussions and possible updates on daily basis.

6 Submission

- Submissions will be done via CengClass.
- Don't write a main function in any of your source files.
- A test environment will be ready in CengClass
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.