폴더 관리 프로그램

2018102238 조인화

**프로젝트 주제**

사진과 음원을 관리하기 위한 폴더 관리 프로그램을 만들어 보고자 한다. 이때, PC의 파일시스템과 유사한 구조의 파일시스템을 위해 자료구조를 설계할 것 이다. 또, PC의 window 탐색기와 유사한 파일 탐색기를 설계 및 구현하는 것이 목적이다.

**구현한 기능**

1. 각 폴더에는 다수의 서브 폴더와 파일들이 포함된다.
2. 파일 타입에는 이미지, 음악, 텍스트 파일 총 3가지가 있다.
3. 폴더 열기 (Root폴더에서부터 단계적으로 서브 폴더로 이동)
4. 상위 폴더로 이동
5. 폴더와 파일을 이름으로 검색 (a 검색 시 aa, ab, ac 모두 출력)
6. 새 폴더/파일 만들기
7. 폴더/파일 이름 바꾸기
8. 현재 폴더 속성 나타내기
9. 폴더/파일 삭제
10. 폴더/파일 복사
11. 폴더/파일 잘라내기
12. 폴더/파일 붙여 넣기
13. 파일 열기 – 파일 속성 나타냄. 연결프로그램도 표시
14. 자주 사용하는 항목
15. 최근 열어본 폴더 목록 보여주기

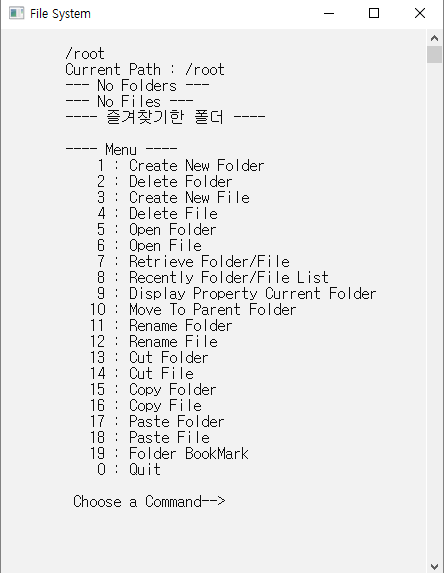
**추가 구현**

1. **보기 편한 콘솔 창을 위해 콘솔 관련 추가 구현을 하였다**

1) 콘솔 창 제목 바꾸기

2) 콘솔 글자 색 바꾸기

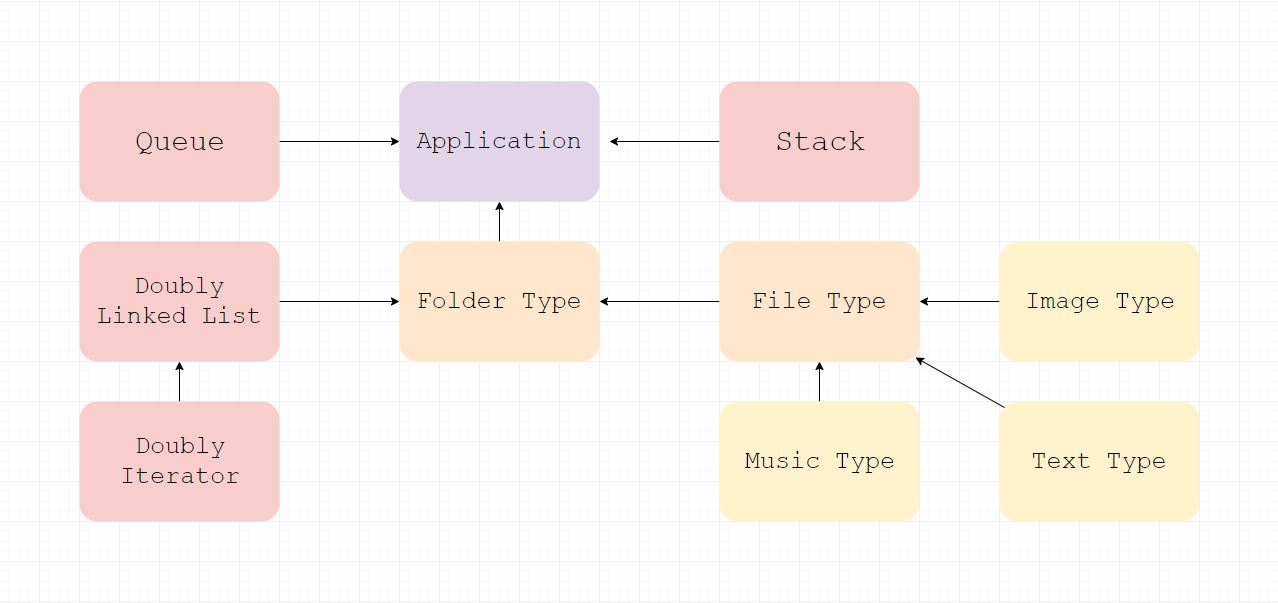
3) 콘솔 배경 색 바꾸기



1. **복사-붙여 넣기 기능 사용시 중복된 파일 이름 방지를 위해 복제된 파일 뒤 (n) 삽입**

****

**클래스 다이어그램**



**ADT**

1. **Application**

class Application

{

private:

FolderType m\_RootFolder; ///< item list.criptor.

FolderType \*m\_curFolder; ///< item list.

FolderType \*m\_fdClipBoard;

FileType \*m\_fClipBoard;

FolderType \*m\_fClipBoard\_Parent;

int state;

int pasteNum;

int m\_Command; ///< current command number.

CircularQueueType<FolderType>\* m\_RecentlyFolder; //최근 열어본 폴더 목록

CircularQueueType<FileType>\* m\_RecentlyFile;

Stack<FolderType\*> \*m\_Folder;

CircularQueueType<FolderType>\* m\_BookMark;

public:

//default constructor.

Application();

//destructor.

~Application();

/\*\*

\* @brief Program driver.

\* @pre program is started.

\* @post program is finished.

\*/

void Run();

/\*\*

\* @brief Display command on screen and get a input from keyboard.

\* @pre none.

\* @post none.

\* @return user's command.

\*/

int GetCommand();

/\*\*

\* @brief 현재폴더에 있는 서브폴더리스트의 이름을 출력. 서브폴더리스트에 폴더가 1개 이상일때만 실행됨.

\* @pre 폴더의 서브폴더갯수가 설정되어있어야함

\* @post none.

\*/

void DisplayCurFolderList();

/\*\*

\* @brief Add new record into list.

\* @pre list should be initialized.

\* @post new record is added into the list.

\* @return return 1 if this function works well, otherwise 0.

\*/

int NewFolder();

/\*\*

\* @brief 폴더를 삭제한다.

\* @pre 루트폴더에 폴더가 1개이상이어야한다.

\* @post 지정한 폴더가 삭제됨.

\* @return return 1 if this function works well, otherwise 0.

\*/

int DeleteFolder();

/\*\*

\* @brief Add new record into list.

\* @pre list should be initialized.

\* @post new record is added into the list.

\* @return return 1 if this function works well, otherwise 0.

\*/

void NewFile();

/\*\*

\* @brief 파일을 삭제한다.

\* @pre 루트폴더에 파일이 1개이상이어야한다.

\* @post 지정한 파일이 삭제됨.

\* @return return 1 if this function works well, otherwise 0.

\*/

int DeleteFile();

/\*\*

\* @brief 현재 가리키고있는 폴더를 연다.

\* @pre 폴더의 서브폴더가 설정되어있어야함.

\* @post none.

\*/

void OpenFolder();

/\*\*

\* @brief 파일을 연다.

\* @pre 폴더안에 파일이 생성되어있어야함.

\* @post none.

\*/

void OpenFile();

/\*\*

\* @brief 입력한 문자가 포함된 모든 폴더나 파일의 이름을 출력한다.

\* @pre 루트폴더에 폴더나 파일이 1개 이상이어야한다.

\* @post none.

\* @return return 1 if this function works well, otherwise 0.

\*/

int RetrieveFolderByName();

/\*\*

\* @brief 최근 열어본 폴더나 파일 10개가 화면에 출력된다.

\* @pre 최근 열어본 폴더나 파일이 있어야함.

\* @post none.

\*/

void RecentlyFolderList();

/\*\*

\* @brief 현재 폴더의 속성 출력

\* @pre 폴더의 속성이 설정되어있어야함.

\* @post none.

\*/

void DisplayProperty();

/\*\*

\* @brief 상위폴더 이동.

\* @pre Stack에 경로가 저장되어있어야함.

\* @post 상위폴더로 이동됨.

\*/

void MoveParentFolder();

/\*\*

\* @brief 폴더 이름 변경

\* @pre 변경할 폴더가 존재해야함

\* @post 폴더의 이름이 변경됨

\*/

void RenameFolder();

/\*\*

\* @brief 파일 이름 변경

\* @pre 변경할 파일이 존재해야함

\* @post 파일의 이름이 변경됨

\*/

void RenameFile();

/\*\*

\* @brief 폴더 잘라내기

\* @pre 폴더가 존재해야함

\* @post 임시저장공간에 폴더가 저장되고 리스트에서 폴더가 빠짐

\*/

void CutFolder();

/\*\*

\* @brief 파일 잘라내기

\* @pre 파일이 존재해야함

\* @post 임시저장공간에 파일이 저장되고 리스트에서 파일이 빠짐

\*/

void CutFile();

/\*\*

\* @brief 폴더 복사하기

\* @pre 폴더가 존재해야함

\* @post 임시저장공간에 폴더가 저장됨

\*/

void CopyFolder();

/\*\*

\* @brief 파일 복사하기

\* @pre 파일이 존재해야함

\* @post 임시저장공간에 파일이 저장됨

\*/

void CopyFile();

/\*\*

\* @brief 폴더 붙여넣기

\* @pre 없음

\* @post name(n)형태로 붙여넣어짐

\*/

void PasteFolder();

/\*\*

\* @brief 파일 붙여넣기

\* @pre 없음

\* @post name(n)형태로 붙여넣어짐

\*/

void PasteFile();

/\*\*

\* @brief 자주사용하는항목 저장

\* @pre 폴더가 존재해야함

\* @post 이름을 받아와 그이름과 동일한 폴더가 리스트에 저장됨

\*/

void BookMark();

/\*\*

\* @brief 자주사용하는항목 저장

\* @pre 폴더가 존재해야함

\* @post 이름을 받아와 그이름과 동일한 폴더가 리스트에 저장됨

\*/

};

1. **Folder Type**

class FolderType

{

private:

string fdName;

string fdPath;

string fdCreatTime;

int fdSubFolderNum;

int fdFileNum;

DoublySortedLinkedList<FolderType>\* fdFolderList; //서브폴더리스트

DoublySortedLinkedList<FileType>\* fdFileList; //서브파일리스트

public:

//default constructor.

FolderType();

//destructor.

~FolderType();

/\*\*

\* @brief 폴더의 이름을 받아온다

\* @pre 이름이 설정되어있어야함

\* @post 없음

\* @return 폴더의 이름 fdName

\*/

string GetName() { return fdName; }

/\*\*

\* @brief 폴더의 위치를 받아온다

\* @pre 폴더의 위치가 설정되어있어야함

\* @post 없음

\* @return 폴더의 위치 fdPath

\*/

string GetPath() { return fdPath; }

/\*\*

\* @brief 폴더의 생성날짜를 받아옴

\* @pre 없음

\* @post 폴더의 생성날짜가 설정됨

\* @return 폴더의 생성날짜 fdCreatTime

\*/

string GetCreatTime() { return fdCreatTime; }

/\*\*

\* @brief 폴더에 포함된 폴더 수를 받아옴

\* @pre 폴더에 포함된 폴더 수가 설정되어있어야함

\* @post 없음

\* @return 폴더에 포함된 폴더 수 fdFolderNum

\*/

int GetSubFolderNum() { return fdSubFolderNum; }

/\*\*

\* @brief 폴더에 포함된 파일 수를 받아옴

\* @pre 폴더에 포함된 파일 수가 설정되어있어야함

\* @post 없음

\* @return 폴더에 포함된 파일 수 fdFileNum

\*/

int GetFileNum() { return fdFileNum; }

/\*\*

\* @brief 폴더의 생성시간을 생성함

\* @pre 없음

\* @post 폴더의 생성시간이 생성됨

\* @return 폴더의 생성시간 curTime

\*/

string GetCurTime();

/\*\*

\* @brief 폴더의 이름을 설정.

\* @pre 없음.

\* @post 폴더의 이름이 설정된다.

\* @param fdName에 넣을 폴더이름을 받아올 파라미터가 name.

\*/

void SetName(string name) { fdName = name; }

/\*\*

\* @brief 폴더의 위치 설정

\* @pre 없음

\* @post 폴더의 위치가 설정됨

\* @param fdPath에 넣을 위치정보를 받아올 파라미터가 path

\*/

void SetPath(string path) { fdPath = path; }

/\*\*

\* @brief 폴더의 생성시간을 설정

\* @pre 없음

\* @post 폴더의 생성시간이 설정됨

\* @param fdCreatTime에 넣을 생성시간을 받아올 파라미터가 creatTime

\*/

void SetCreatTime(string creatTime) { fdCreatTime = creatTime; }

/\*\*

\* @brief 폴더에 포함된 서브폴더의 갯수를 설정

\* @pre 없음

\* @post 폴더에 포함된 서브폴더의 갯수가 설정됨

\* @param fdSubFolderNum에 넣을 위치정보를 받아올 파라미터가 subFolderNum

\*/

void SetSubFolderNum(int subFolderNum) { fdSubFolderNum = subFolderNum; }

/\*\*

\* @brief 폴더에 포함된 파일의 갯수를 설정

\* @pre 없음

\* @post 폴더에 포함된 파일의 갯수가 설정됨

\* @param fdFileNum에 넣을 위치정보를 받아올 파라미터가 FileNum

\*/

void SetFileNum(int FileNum) { fdFileNum = FileNum; }

/\*\*

\* @brief 화면에 폴더의 속성을 보여줌

\* @pre 폴더의 속성이 설정되어있어야함

\* @post 없음

\*/

void DisplayRecordOnScreen();

/\*\*

\* @brief 키보드로 폴더의 이름을 받아옴

\* @pre 없음

\* @post 폴더의 이름을 키보드로 받음

\*/

void SetNameFromKB();

/\*\*

\* @brief 키보드로 폴더의 속성을 받아옴

\* @pre 없음

\* @post 폴더의 속성을 키보드로 받음

\*/

void SetRecordFromKB();

/\*\*

\* @brief 두개의 폴더를 이름으로 비교함.

\* @pre two item types should be initialized.

\* @post the target file is included the new item record.

\* @param data target item for comparing.

\* @return return LESS if this.id < data.id,

\* return GREATER if this.id > data.id then,

\* otherwise return EQUAL.

\*/

RelationType CompareByID(const FolderType& data);

/\*\*

\* @brief data 폴더에 있는 속성들을 지정한 폴더 속성에 똑같이 입력

\* @pre 지정할 폴더가 만들어져있어야함

\* @post 지정한 폴더에 data폴더와 동일한 속성이 설정됨

\* @param 지정한 폴더에 옮겨넣을 속성을 가진 data폴더

\* @return 지정한 폴더를 리턴함

\*/

FolderType& operator= (const FolderType& data);

// SortedArrayList 관련 함수------------------------------------------

/\*\*

\* @brief 새폴더 정보를 키보드로부터 읽어서 폴더리스트에 추가. 처음폴더라면 sortedlist를 동적으로 할당하여 초기화하고 추가

\* @pre 없음

\* @post 새폴더가 생성됨

\*/

int AddFolder();

/\*\*

\* @brief 새폴더 정보를 키보드로부터 읽어서 폴더리스트에 추가. 처음폴더라면 sortedlist를 동적으로 할당하여 초기화하고 추가

\* @pre 없음

\* @post 새폴더가 생성됨

\*/

int AddFolder(FolderType\* data);

/\*\*

\* @brief 삭제할 폴더 정보를 읽어서 해당 폴더를 찾아 삭제

\* @pre 없음

\* @post 폴더가 삭제됨

\*/

int DeleteFolder();

/\*\*

\* @brief 삭제할 폴더 정보를 읽어서 해당 폴더를 찾아 삭제

\* @pre 없음

\* @post 폴더가 삭제됨

\*/

int DeleteFolder(FolderType\* data);

/\*\*

\* @brief 검색할 폴더정보를 화면에서 읽어 검색하여 결과 출력.

\* @pre 폴더정보가 설정되어있어야함.

\* @post 없음.

\*/

int RetrieveFolderByName();

/\*\*

\* @brief 현재폴더에 포함된 모든 서브폴더의 이름을 출력

\* @pre 현재폴더가 설정되어있어야함

\* @post 없음

\*/

void DisplayAllFolderName();

/\*\*

\* @brief 폴더 이름을 받아 그 폴더를 열어줌.

\* @pre 폴더가 설정되어있어야함

\* @post 서브폴더에 들어감과 동시에 그 서브폴더의 주소값을 반환해줌

\*/

FolderType\* OpenSubFolder();

/\*\*

\* @brief '<<' 오버로딩 함수.

\* @pre none.

\* @post 폴더명을 출력

\*/

friend ostream& operator << (ostream& os, const FolderType& p)

{

return os << p.fdName;

}

/\*\*

\* @brief 폴더타입 '==' 오버로딩 함수.

\* @pre none.

\* @post 폴더명이 같으면 true, 다르면 false 리턴.

\*/

bool operator== (const FolderType& data);

/\*\*

\* @brief 폴더타입 '>' 오버로딩 함수.

\* @pre none.

\* @post 폴더명이 더 크면 true, 작으면 false 리턴.

\*/

bool operator> (const FolderType& data);

/\*\*

\* @brief 폴더타입 '<' 오버로딩 함수.

\* @pre none.

\* @post 폴더명이 더 크면 true, 다르면 false 리턴.

\*/

bool operator< (const FolderType& data);

/\*\*

\* @brief 현재폴더에 포함된 모든 파일의 이름을 출력

\* @pre 현재폴더가 설정되어있어야함

\* @post 없음

\*/

void DisplayAllFileName();

/\*\*

\* @brief 새파일 정보를 키보드로부터 읽어서 추가.

\* @pre 없음

\* @post 새파일이 생성됨

\*/

int AddFile();

/\*\*

\* @brief 새파일 정보를 키보드로부터 읽어서 추가.

\* @pre 없음

\* @post 새파일이 생성됨

\*/

int AddFile(FileType\* data);

/\*\*

\* @brief 삭제할 파일 정보를 읽어서 해당 파일을 찾아 삭제

\* @pre 없음

\* @post 파일이 삭제됨

\*/

int DeleteFile();

/\*\*

\* @brief 삭제할 파일 정보를 읽어서 해당 파일을 찾아 삭제

\* @pre 없음

\* @post 파일이 삭제됨

\*/

int DeleteFile(FileType\* data);

/\*\*

\* @brief 이름을 받아 그 파일을 열어줌.

\* @pre 파일이 설정되어있어야함

\* @post 파일을 열어 파일 내용을 수정할 수 있다

\*/

FileType\* OpenFile();

/\*\*

\* @brief 폴더의 이름을 바꿔줌

\* @pre 변경할 폴더가 존재해야함

\* @post 폴더 이름이 변경됨

\*/

int RenameFolder();

/\*\*

\* @brief 파일의 이름을 바꿔줌

\* @pre 변경할 파일이 존재해야함

\* @post 파일 이름이 변경됨

\*/

int RenameFile();

};

1. **File Type**

class FileType

{

protected:

string fName; //파일 이름

string fPath; //저장 경로

string fCreatTime; //파일 생성 날짜

string fExtension; //파일 확장자

string fAccessTime; //엑세스한 날짜

public:

//default constructor.

FileType();

//destructor.

\*/

~FileType();

/\*\*

\* @brief 파일의 이름을 받아온다

\* @pre 이름이 설정되어있어야함

\* @post 없음

\* @return 파일의 이름 fName

\*/

string GetName() { return fName; }

/\*\*

\* @brief 파일의 위치를 받아온다

\* @pre 파일의 위치가 설정되어있어야함

\* @post 없음

\* @return 파일의 위치 fPath

\*/

string GetPath() { return fPath; }

/\*\*

\* @brief 파일의 생성날짜를 받아옴

\* @pre 없음

\* @post 파일의 생성날짜가 설정됨

\* @return 파일의 생성날짜 fCreatTime

\*/

string GetCreatTime() { return fCreatTime; }

/\*\*

\* @brief 파일의 연결프로그램을 받아옴

\* @pre 없음

\* @post 파일의 연결프로그램이 설정됨

\* @return 파일의 연결프로그램 fExtension

\*/

string GetExtension() { return fExtension; }

/\*\*

\* @brief 파일의 액세스날짜를 받아옴

\* @pre 없음

\* @post 파일의 액세스날짜가 설정됨

\* @return 파일의 액세스날짜 fAccessTime

\*/

string GetAccessTime() { return fAccessTime; }

/\*\*

\* @brief 파일의 생성시간을 생성함

\* @pre 없음

\* @post 파일의 생성시간이 생성됨

\* @return 파일의 생성시간 curTime

\*/

string GetCurTime();

/\*\*

\* @brief 파일의 이름을 설정.

\* @pre 없음.

\* @post 파일의 이름이 설정된다.

\* @param fName에 넣을 파일이름을 받아올 파라미터가 name.

\*/

void SetName(string name) { fName = name; }

/\*\*

\* @brief 파일의 위치 설정

\* @pre 없음

\* @post 파일의 위치가 설정됨

\* @param fPath에 넣을 위치정보를 받아올 파라미터가 path

\*/

void SetPath(string path) { fPath = path; }

/\*\*

\* @brief 파일의 생성시간을 설정

\* @pre 없음

\* @post 파일의 생성시간이 설정됨

\* @param fCreatTime에 넣을 생성시간을 받아올 파라미터가 creatTime

\*/

void SetCreatTime(string creatTime) { fCreatTime = creatTime; }

/\*\*

\* @brief 파일의 액세스시간을 설정

\* @pre 없음

\* @post 파일의 액세스시간이 설정됨

\* @param fAccessTime에 넣을 생성시간을 받아올 파라미터가 Time

\*/

void SetAccessTime(string Time) { fAccessTime = Time; }

/\*\*

\* @brief 파일의 연결프로그램 설정

\* @pre 없음

\* @post 파일의 연결프로그램이 설정됨

\* @param fExtension에 넣을 연결프로그램 정보를 받아올 파라미터가 extension

\*/

void SetExtention(string extension) { fExtension = extension; }

/\*\*

\* @brief 화면에 파일의 속성을 보여줌

\* @pre 파일의 속성이 설정되어있어야함

\* @post 없음

\*/

void DisplayRecordOnScreen();

/\*\*

\* @brief 키보드로 파일의 이름을 받아옴

\* @pre 없음

\* @post 파일의 이름을 키보드로 받음

\*/

void SetNameFromKB();

/\*\*

\* @brief 키보드로 파일의 속성을 받아옴

\* @pre 없음

\* @post 파일의 속성을 키보드로 받음

\*/

void SetRecordFromKB(string n);

/\*\*

\* @brief 두개의 파일을 이름으로 비교함.

\* @pre two item types should be initialized.

\* @post the target file is included the new item record.

\* @param data target item for comparing.

\* @return return LESS if this.id < data.id,

\* return GREATER if this.id > data.id then,

\* otherwise return EQUAL.

\*/

RelationType CompareByID(const FileType& data);

/\*\*

\* @brief data 파일에 있는 속성들을 지정한 파일 속성에 똑같이 입력

\* @pre 지정할 파일이 만들어져있어야함

\* @post 지정한 파일에 data파일와 동일한 속성이 설정됨

\* @param 지정한 파일에 옮겨넣을 속성을 가진 data파일

\* @return 지정한 파일를 리턴함

\*/

FileType& operator= (const FileType& data);

/\*\*

\* @brief '<<' 오버로딩 함수.

\* @pre none.

\* @post 파일명을 출력

\*/

friend ostream& operator << (ostream& os, const FileType& p)

{

return os << p.fName;

}

/\*\*

\* @brief 파일타입 '==' 오버로딩 함수.

\* @pre none.

\* @post 파일명이 같으면 true, 다르면 false 리턴.

\*/

bool operator== (const FileType& data);

/\*\*

\* @brief 파일타입 '>' 오버로딩 함수.

\* @pre none.

\* @post 파일명이 더 크면 true, 작으면 false 리턴.

\*/

bool operator> (const FileType& data);

/\*\*

\* @brief 파일타입 '<' 오버로딩 함수.

\* @pre none.

\* @post 파일명이 더 크면 false, 작으면 true 리턴.

\*/

bool operator< (const FileType& data)

{

if (this->fName < data.fName)

{

return true;

}

else

return false;

}

};

class ImageType : public FileType

{

public:

ImageType() : FileType() {}

/\*\*

\* @brief 파일명 설정

\* @pre none.

\* @post 파일명 변경

\*/

void SetName(string n);

/\*\*

\* @brief 연결프로그램 설정

\* @pre none.

\* @post 연결프로그램 변경

\*/

void SetExtention(string n);

/\*\*

\* @brief 키보드로 파일의 이름을 받아옴

\* @pre 없음

\* @post 파일의 이름을 키보드로 받음

\*/

void SetNameFromKB();

/\*\*

\* @brief 키보드로 파일의 속성을 받아옴

\* @pre 없음

\* @post 파일의 속성을 키보드로 받음

\*/

void SetRecordFromKB(string n);

};

class MusicType : public FileType

{

public:

MusicType() : FileType() {}

/\*\*

\* @brief 파일명 설정

\* @pre none.

\* @post 파일명 변경

\*/

void SetName(string n);

/\*\*

\* @brief 연결프로그램 설정

\* @pre none.

\* @post 연결프로그램 변경

\*/

void SetExtention(string n);

/\*\*

\* @brief 키보드로 파일의 이름을 받아옴

\* @pre 없음

\* @post 파일의 이름을 키보드로 받음

\*/

void SetNameFromKB();

/\*\*

\* @brief 키보드로 파일의 속성을 받아옴

\* @pre 없음

\* @post 파일의 속성을 키보드로 받음

\*/

void SetRecordFromKB(string n);

};

class TextType : public FileType

{

public:

TextType() : FileType() {}

/\*\*

\* @brief 파일명 설정

\* @pre none.

\* @post 파일명 변경

\*/

void SetName(string n);

/\*\*

\* @brief 연결프로그램 설정

\* @pre none.

\* @post 연결프로그램 변경

\*/

void SetExtention(string n);

/\*\*

\* @brief 키보드로 파일의 이름을 받아옴

\* @pre 없음

\* @post 파일의 이름을 키보드로 받음

\*/

void SetNameFromKB();

/\*\*

\* @brief 키보드로 파일의 속성을 받아옴

\* @pre 없음

\* @post 파일의 속성을 키보드로 받음

\*/

void SetRecordFromKB(string n);

};

1. **Doubly Linked List**

template <typename T>

struct DoublyNodeType

{

T data; ///< A data for each node.

DoublyNodeType\* prev; ///< A node pointer of previous node.

DoublyNodeType\* next; ///< A node pointer of next node.

};

/\*\*

\* Single sorted list class for managing items.

\*/

template <typename T>

class DoublySortedLinkedList

{

friend class DoublyIterator<T>;

public:

//default constructor.

DoublySortedLinkedList();

//destructor.

~DoublySortedLinkedList();

/\*\*

\* @brief Determines whether the list is full.

\* @pre List has been initialized.

\* @post None.

\* @return If there is not memory left, throws bad\_alloc exception and return true, otherwise false.

\*/

bool IsFull();

/\*\*

\* @brief Determines whether the list is empty.

\* @pre List has been initialized.

\* @post None.

\* @return Return true if the list is empty, otherwise false.

\*/

bool IsEmpty();

/\*\*

\* @brief Initialize list to empty state.

\* @pre None

\* @post List is empty.

\*/

void MakeEmpty();

/\*\*

\* @brief Get number of elements in the list.

\* @pre None.

\* @post Function value = number of elements in list.

\* @return Number of elements in this list.

\*/

int GetLength() const;

/\*\*

\* @brief Add item into this list.

\* @pre List is not full. item is not in list.

\* @post Item is inserted in this list.

\* @return 1 if this function works well, otherwise 0.

\*/

int Add(T item);

/\*\*

\* @brief Retrieve list element whose key matches item's key (if present).

\* @pre Key member of item is initialized.

\* @post If there is an element whose key matches with item's key then the element's record is copied to the item.

\* @return 1 if any element's primary key matches with item's, otherwise 0.

\*/

int Get(T& item);

/\*\*

\* @brief Primary key(학번)를 바탕으로 검색하여 해당하는 정보를 삭제함.

\* @pre 리스트가 존재/초기화 완료되어있는 상태여야 함.

\* @post 일치하는 데이터가 리스트에 있을 경우 삭제됨.

\* @param data primary key가 반드시 포함되어 있는 객체.

\* @return 삭제할 일치하는 데이터가 있으면 삭제 후 1, 없으면 0을 리턴.

\*/

int Delete(T data);

/\*\*

\* @brief Primary key(학번)를 바탕으로 검색하여 해당하는 정보를 새로 채워넣음.

\* @pre 리스트가 존재/초기화 완료되어있는 상태여야 함.

\* @post 일치하는 데이터가 리스트에 있을 경우 새로운 데이터로 교체됨.

\* @param data primary key가 반드시 포함되어 있는 객체.

\*/

void Replace(T data);

/\*\*

\* @brief 이름이 일치하는 데이터를 찾아줌

\* @pre 리스트가 존재/초기화 완료되어있는 상태여야 함.

\* @post 일치하는 데이터가 리스트에 있을 경우 그 데이터 주소값 반환

\* @param data primary key가 반드시 포함되어 있는 객체.

\*/

T\* GetCurNode(T data);

private:

DoublyNodeType<T>\* m\_pFirst; ///< Pointer for pointing a first node.

DoublyNodeType<T>\* m\_pLast; ///< Last node.

int m\_nLength; ///< Number of node in the list.

};

1. **Doubly Iterator**

template <typename T>

class DoublyIterator {

friend class DoublySortedLinkedList<T>;

public:

/\*\*

\* Initialize m\_List and m\_pCurPointer. (constructor overloading)

\*/

DoublyIterator(DoublySortedLinkedList<T>& list) : m\_List(list), m\_pCurPointer(list.m\_pFirst) {};

/\*\*

\* @brief 현재 노드를 가리키는 포인터가 null인지 아닌지 판단하는 메서드.

\* @pre Iterator has been initialized.

\* @post None.

\* @return nullptr true, otherwise false.

\*/

bool NotNull();

/\*\*

\* @brief 현재 노드 포인터의 다음 노드가 null인지 아닌지 판단하는 메서드.

\* @pre Iterator has been initialized.

\* @post None.

\* @return nullptr true, otherwise false.

\*/

bool NextNotNull();

/\*\*

\* @brief Get first data of the list.

\* @pre Iterator has been initialized.

\* @post Current pointer is moved to the first node.

\* @return Return first data of the list.

\*/

T First();

/\*\*

\* @brief Get next data of the current pointer.

\* @pre Iterator has been initialized.

\* @post Current pointer is moved to the next node.

\* @return Return next data of the list.

\*/

T Next();

/\*\*

\* @brief Get current data of the list and go to the next node. 현재 노드. 객체반환은 return by value

\* @pre Iterator has been initialized.

\* @post Current pointer is moved to next node.

\* @return Return current data of the list.

\*/

T GetCurrentNode();

/\*\*

\* @brief

\* @pre

\* @post

\* @return

\*/

//T\* GetAdress(T item);

private:

DoublySortedLinkedList<T>& m\_List; ///< 사용할 리스트의 참조 변수

DoublyNodeType<T>\* m\_pCurPointer; ///< Node pointer for pointing current node to use iteration.

};

1. **Stack**

/\*\*

\* @brief Exception class thrown by Push when stack is full.

\*/

class FullStack

{

public:

/\*\*

\* @brief Display a message for full stack on screen.

\*/

void print()

{

cout << "FullStack exception thrown." << endl;

}

};

/\*\*

\* @brief Exception class thrown by Pop and Top when stack is empty.

\*/

class EmptyStack

{

public:

/\*\*

\* @brief Display a message for full stack on screen.

\*/

void print()

{

cout << "EmtpyStack exception thrown." << endl;

}

};

/\*\*

\* @brief Stack class.

\* @details This class processes as Last In, First Out (LIFO).

\*/

template <typename T>

class Stack

{

public:

/\*\*

\* @brief MaxAllocate dynamic array whose size is maxStack.

\* @pre The maxStack is defined

\* @post Member of items points the allocated array.

\*/

Stack();

Stack(const Stack<T>& stack);

/\*\*

\* @brief Destruct the object. Free the array dynamically allocated.

\* @pre Release memory for stack pointer.

\* @post none.

\*/

~Stack();

/\*\*

\* @brief 스택이 차있는지 확인.

\* @pre 스택은 초기화 되어있다.

\* @post Function value = (stack is full)

\*/

bool IsFull() const;

/\*\*

\* @brief 스택이 비어있는지 확인.

\* @pre 스택은 초기화 되어있다.

\* @post Function value = (stack is empty)

\*/

bool IsEmpty() const;

/\*\*

\* @brief 스택의 top에 새로운 데이터 추가.

\* @pre 스택은 초기화 되어있다.

\* @post If (stack is full), FullStack exception is thrown; otherwise, newItem is at the top of the stack.

\*/

void Push(T item);

/\*\*

\* @brief 스택으로부터 top부분 제거.

\* @pre 스택은 초기화 되어있다..

\* @post If (stack is empty), EmptyStack exception is thrown; otherwise, top element has been removed from stack.

\*/

T Pop();

/\*\*

\* @brief 스택의 top 데이터를 복사해서 반환.

\* @pre 스택은 초기화 되어있다.

\* @post If (stack is empty), EmptyStack exception is thrown; otherwise, top element has been removed from stack.

\*/

T Top();

/\*\*

\* @brief 스크린에 스택의 모든 데이터 출력

\* @pre 스택은 초기화 되어있다.

\* @post 없음.

\*/

void Print();

/\*\*

\* @brief 대입연산자 오버로딩

\* @post 입력받은 스택이 딥카피된다.

\*/

Stack<T>& operator=(const Stack<T>& stack);

private:

int top; ///< Number of elements.

T \*items; ///< Pointer for a stack.

};

1. **Queue**

/\*\*

\* @brief Exception class thrown by Enqueue when queue is full.

\*/

class FullQueue

{

public:

/\*\*

\* @brief Display a message for full queue on screen.

\*/

void print()

{

cout << "\tFullQueue exception thrown." << endl;

}

};

/\*\*

\* @brief Exception class thrown by Dequeue when queue is empty.

\*/

class EmptyQueue

{

public:

/\*\*

\* @brief Display a message for empty queue on screen.

\*/

void print()

{

cout << "\tEmtpyQueue exception thrown." << endl;

}

};

/\*\*

\* @brief Queue class.

\* @details This class processes as First In, First Out (FIFO), 템플릿을 적용해 다양한 변수 타입을 저장할 수 있다.

\*/

template <typename T>

class CircularQueueType

{

public:

/\*\*

\* @brief Allocate dynamic array whose size is maxQueue.

\* @pre The maxQueue is defined

\* @post Member of items points the allocated array.

\*/

CircularQueueType();

/\*\*

\* @brief Allocate dynamic array whose size is max.

\* @pre none.

\* @post Member of items points the allocated array.

\*/

CircularQueueType(int max);

/\*\*

\* @brief Destruct the object. Free the array dynamically allocated.

\* @pre Release memory for queue pointer.

\* @post none.

\*/

~CircularQueueType();

/\*\*

\* @brief Determines whether the queue is full.

\* @pre Queue has been initialized.

\* @post Function value = (queue is full)

\*/

bool IsFull();

/\*\*

\* @brief Determines whether the queue is empty.

\* @pre Queue has been initialized.

\* @post Function value = (queue is empty)

\*/

bool IsEmpty();

/\*\*

\* @brief Makes the queue empty.

\* @pre Queue has been initialized.

\* @post m\_iFront와 m\_iRear is set same value as Constructer.

\*/

void MakeEmpty();

/\*\*

\* @brief Adds newItem to the last of the queue.

\* @pre Queue has been initialized.

\* @post If (queue is full), FullQueue exception is thrown; otherwise, newItem is at the last.

\*/

void EnQueue(T item);

/\*\*

\* @brief Removes first item from the queue.

\* @pre Queue has been initialized.

\* @post If (queue is empty), EmptyQueue exception is thrown; otherwise, first element has been removed from queue. item gets value of removed item.

\*/

void DeQueue(T &item);

/\*\*

\* @brief Print all the items in the queue on screen

\* @pre Queue has been initialized.

\* @post None.

\*/

void Print();

/\*\*

\* @brief 찾고자 하는 폴더가 현재 큐에 있는지 확인.

\* @pre None.

\* @post None.

\*/

int Check(T newitem, int x);

private:

int m\_iFront; //index of one infront of the first element.

int m\_iRear; //index of the last element.

int m\_nMaxQueue; //max size of the queue.

T\* m\_pItems; //pointer for dynamic allocation.

};

**개발 환경**

Microsoft Visual Studio 2017 (C++)