**Homework 10**

|  |
| --- |
| 그림입니다. 원본 그림의 이름: YU_UI_RGB-10.png 원본 그림의 크기: 가로 2256pixel, 세로 3047pixel 프로그램 이름 : Adobe ImageReady |

|  |  |
| --- | --- |
| 과목명 | 객체지향프로그래밍과자료구조 |
| 교수님 | 김영탁 교수님 |
| 이 름 | 김주환 |
| 학 번 | 21812158 |
| 일 자 | 2021.11.17.수 |

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* 균형 이진 탐색 트리  \* Programmed by J. H. Kim  \* Last updated : 2021-11-17 \*/  #include <iostream>  #include <fstream>  #include <iomanip>  #include <string>  #include "T\_BST.h"  #include "T\_Entry.h"  #include "T\_Array.h"  #include "Book.h"  #include "Date.h"  using namespace std;  #define NUM\_BOOKS 15  void main()  {  Book books[NUM\_BOOKS] =  {  //Book( string bk\_title, string bk\_author, Date dt)  Book(string("Book\_01"), string("Kim"), Date(2020, 1, 1)),  Book(string("Book\_02"), string("Kim"), Date(2010, 1, 1)),  Book(string("Book\_03"), string("Kim"), Date(2013, 1, 1)),  Book(string("Book\_04"), string("Lee"), Date(2011, 1, 1)),  Book(string("Book\_05"), string("Hwang"), Date(2001, 1, 1)),  Book(string("Book\_06"), string("Choi"), Date(2003, 1, 1)),  Book(string("Book\_07"), string("Park"), Date(2009, 1, 1)),  Book(string("Book\_08"), string("Brown"), Date(2012, 1, 1)),  Book(string("Book\_09"), string("Alpha"), Date(1980, 1, 1)),  Book(string("Book\_10"), string("Chalie"), Date(1970, 1, 1)),  Book(string("Book\_11"), string("Tango"), Date(1985, 1, 1)),  Book(string("Book\_12"), string("Yankee"), Date(1977, 1, 1)),  Book(string("Book\_13"), string("Zulu"), Date(2018, 1, 1)),  Book(string("Book\_14"), string("Foxtrot"), Date(2015, 1, 1)),  Book(string("Book\_15"), string("Delta"), Date(2000, 1, 1)),  /\* \*/  };  ofstream fout("output.txt");  if (fout.fail())  {  cout << "Fail to create output.txt for results !!" << endl;  exit;  }  fout << "Input books[] array : " << endl;  for (int i = 0; i < NUM\_BOOKS; i++)  {  fout << books[i] << endl;  }  fout << endl;  fout << endl << "Testing Binary Search Tree with Rebalancing" << endl;  T\_Entry<string, Book\*> entry\_title\_pBK;  T\_BST<string, Book\*> BBST\_BK\_keyTitle("BBST\_BK\_keyTitle");  T\_BSTN<string, Book\*>\* pRoot, \*\* ppBBST\_BK\_root;  ppBBST\_BK\_root = BBST\_BK\_keyTitle.getRootAddr();  for (int i = 0; i < NUM\_BOOKS; i++)  {  entry\_title\_pBK.setKey(books[i].getTitle());  entry\_title\_pBK.setValue(&books[i]);  //fout << "Insert inOrder (" << setw(3) << books[i] << ") into " << BBST\_BK\_keyTitle.getName() << endl;  BBST\_BK\_keyTitle.insertAndRebalance(entry\_title\_pBK);  }  fout << "\nEntries in " << BBST\_BK\_keyTitle.getName() << " (in order of Book Title) : " << endl;  BBST\_BK\_keyTitle.fprint\_inOrder(fout);  // BBST\_BK\_keyTitle.fprint\_with\_Depth(fout);  // fout << endl << "Balanced Binary Search Tree (BBST) with key book-author" << endl;  T\_Entry<string, Book\*> entry\_Author\_pBK;  T\_BST<string, Book\*> BBST\_BK\_keyAuthor("BBST\_BK\_keyAuthor");  T\_BSTN<string, Book\*>\*\* ppRoot\_BBST\_BK\_keyAuthor, \* pBBST\_BK\_keyAuthor;  ppRoot\_BBST\_BK\_keyAuthor = BBST\_BK\_keyAuthor.getRootAddr();  for (int i = 0; i < NUM\_BOOKS; i++)  {  entry\_Author\_pBK.setKey(books[i].getAuthor());  entry\_Author\_pBK.setValue(&books[i]);  //fout << "Insert inOrder (" << setw(3) << books[i] << ") into " << BBST\_BK\_keyTitle.getName() << endl;  BBST\_BK\_keyAuthor.insertAndRebalance(entry\_Author\_pBK);  }  fout << "\nEntries in " << BBST\_BK\_keyAuthor.getName() << " (in order of Book Author) : " << endl;  BBST\_BK\_keyAuthor.fprint\_inOrder(fout);  // BBST\_BK\_keyAuthor.fprint\_with\_Depth(fout);  // Testing Search on Binary Search Tree  string author = books[0].getAuthor();  Date d1, d2;  Book\* pBk;  T\_Array<Book\*> array\_pBook(1, string("Array\_Book"));  d1.setDate(2010, 1, 1);  d2.setDate(2015, 12, 31);  pBBST\_BK\_keyAuthor = BBST\_BK\_keyAuthor.searchBSTN(author);  BBST\_BK\_keyAuthor.traversal\_inOrder(pBBST\_BK\_keyAuthor, array\_pBook);  fout << endl << "Books of author (" << author << ") published during " << d1 << " ~ " << d2 << ":" << endl;  for (int i = 0; i < array\_pBook.size(); i++)  {  if (array\_pBook[i]->getAuthor() == author)  {  pBk = array\_pBook[i];  if ((pBk->getPubDate() >= d1) && (pBk->getPubDate() <= d2))  fout << \*(array\_pBook[i]) << endl;  }  }  // fout << endl << "Balanced Binary Search Tree (BBST) with key publication date" << endl;  T\_Entry<Date, Book\*> entry\_PubDate\_pBK;  T\_BST<Date, Book\*> BBST\_BK\_keyPubDate("BBST\_BK\_keyPubDate");  T\_BSTN<Date, Book\*>\*\* ppRoot\_BBST\_BK\_keyPubDate;  ppRoot\_BBST\_BK\_keyPubDate = BBST\_BK\_keyPubDate.getRootAddr();  for (int i = 0; i < NUM\_BOOKS; i++)  {  entry\_PubDate\_pBK.setKey(books[i].getPubDate());  entry\_PubDate\_pBK.setValue(&books[i]);  //fout << "Insert inOrder (" << setw(3) << books[i] << ") into " << BBST\_BK\_keyTitle.getName() << endl;  BBST\_BK\_keyPubDate.insertAndRebalance(entry\_PubDate\_pBK);  }  fout << "\nEntries in " << BBST\_BK\_keyPubDate.getName() << " (in order of Book Publication Date) : " << endl;  BBST\_BK\_keyPubDate.fprint\_inOrder(fout);  //BBST\_BK\_keyPubDate.fprint\_with\_Depth(fout);  /\* fout << "\nRemoving the root entry in sequence ..." << endl;  for (int i = 0; i < NUM\_BOOKS; i++)  {  pRoot = BBST\_BK\_keyTitle.getRoot();  entry\_title\_pBK = pRoot->getEntry();  fout << "\nremove " << entry\_title\_pBK << endl;  BBST\_BK\_keyTitle.eraseBSTN(&pRoot);  // BBST\_BK\_keyTitle.fprint\_with\_Depth(fout);  } \*/  fout << "\nClearing BBST\_BKs . . . " << endl;  BBST\_BK\_keyTitle.clear();  BBST\_BK\_keyAuthor.clear();  BBST\_BK\_keyPubDate.clear();  fout << "All BBST\_BKs cleared !! " << endl;  fout.close();  } |
| /\* Date.h \*/  #ifndef D\_H  #define D\_H  #include <iostream>  #include <iomanip>  using namespace std;  #define WEEKDAY\_AD01Jan01 MON // the weekday of AD Jan 1.  #define DAYS\_PER\_WEEK 7  class Date {  friend ostream& operator<<(ostream&, const Date&); // 출력  public:  // 생성자 & 소멸자  Date(); // default constructor  Date(int y, int m, int d); // constructor  ~Date(); // destructor  void setDate(int newYear, int newMonth, int newDay); // Date 설정  int getWeekDay(); // 요일 반환  int getElapsedDaysFromAD010101() const; // get elapsed days from AD 1. 1. 1.  int getElapsedDaysFromAD010101(Date) const;  // 비교연산자  bool operator<(const Date&) const;  bool operator<=(const Date&) const;  bool operator>(const Date&) const;  bool operator>=(const Date&) const;  bool operator==(const Date&) const;  private:  bool isValidDate(int y, int m, int d); // 유효한 Date 정보인지 판단  int year;  int month;  int day;  };  bool isLeapYear(int y); // check whether the given year y is leap year  int getYearDay(int year, int month, int day); // 해당 연도의 몇번째 일인지 반환  #endif // !D\_H |
| /\* Date.cpp\*/  #include "Date.h"  ostream& operator<<(ostream& fout, const Date& d) {  fout << "(" << setw(2) << d.year << "." << setw(2) << d.month << "." << setw(2) << d.day << ")";  return fout;  }  Date::Date()  : year(0), month(0), day(0) {  // cout << "constructor for Date" << endl;  }  Date::Date(int y, int m, int d)  : year(y), month(m), day(d) {  // cout << "constructor for Date" << endl;  }  Date::~Date() {  // cout << "destructor for Date" << endl;  }  void Date::setDate(int newYear, int newMonth, int newDay) {  if (isValidDate(newYear, newMonth, newDay))  year = newYear, month = newMonth, day = newDay;  else {  cout << "Invalid date (" << newYear << ", " << newMonth << ", " << newDay << ")";  cout << "Program aborted !!" << endl;  exit(1);  }  }  int Date::getWeekDay() {  int weekDay\_AD010101 = 1;  int weekDay;  int elapsedDays = 0;  elapsedDays = getElapsedDaysFromAD010101();  weekDay = (elapsedDays + weekDay\_AD010101 - 1) % 7; // 최초는 월욜이면 추가 안하면 대지않나?  // cout << ", Elapsed days from AD Jan. 1, 1 (" << elapsedDays << ")";  return weekDay;  }  int Date::getElapsedDaysFromAD010101() const {  int yearDay;  int elpsDay = 0;  for (int y = 1; y < this->year; y++) {  if (isLeapYear(y)) elpsDay += 366;  else elpsDay += 365;  }  yearDay = getYearDay(this->year, this->month, this->day);  elpsDay += yearDay;  return elpsDay;  }  int Date::getElapsedDaysFromAD010101(Date d) const {  return d.getElapsedDaysFromAD010101();  }  bool Date::operator<(const Date& d) const {  int tl, tr;  tl = getElapsedDaysFromAD010101();  tr = d.getElapsedDaysFromAD010101();  if (tl < tr) return true;  else return false;  }  bool Date::operator<=(const Date& d) const {  int tl, tr;  tl = getElapsedDaysFromAD010101();  tr = d.getElapsedDaysFromAD010101();  if (tl <= tr) return true;  else return false;  }  bool Date::operator>(const Date& d) const {  int tl, tr;  tl = getElapsedDaysFromAD010101();  tr = d.getElapsedDaysFromAD010101();  if (tl > tr) return true;  else return false;  }  bool Date::operator>=(const Date& d) const {  int tl, tr;  tl = getElapsedDaysFromAD010101();  tr = d.getElapsedDaysFromAD010101();  if (tl >= tr) return true;  else return false;  }  bool Date::operator==(const Date& d) const {  int tl, tr;  tl = getElapsedDaysFromAD010101();  tr = d.getElapsedDaysFromAD010101();  if (tl == tr) return true;  else return false;  }  bool Date::isValidDate(int y, int m, int d) {  int days\_month[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  if (isLeapYear(y)) days\_month[2] = 29;  if ((m >= 1) && (m <= 12) && (d >= 1) && (d <= days\_month[m])) return true; // 범위 내의 정보인지 확인  else {  cout << "Illegal date! (" << m << ", " << d << ") ==> Program aborted." << endl;  return false;  }  }  bool isLeapYear(int y) {  if ((y % 4 == 0 && y % 100 != 0) || y % 400 == 0)  return true;  else  return false;  }  int getYearDay(int year, int month, int day) {  int days\_month[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };  int yearDay = 0;  if (isLeapYear(year))  days\_month[2] = 29;  for (int i = 1; i < month; i++)  yearDay += days\_month[i];  return yearDay + day;  } |
| /\* Book.h \*/  #ifndef B\_H  #define B\_H  #include <iostream>  #include <iomanip>  #include"Date.h"  using namespace std;  class Book  {  friend ostream& operator<<(ostream& fout, Book& bk)  {  fout << "[";  fout.setf(ios::left);  fout << setw(8) << bk.title;  fout << ", " << setw(8) << bk.author;  fout << ", " << bk.pubDate;  fout << "]" ;  return fout;  }  public:  Book(string bk\_title, string bk\_author, Date dt) :  title(bk\_title), author(bk\_author), pubDate(dt) {}  string& getTitle() { return title; }  string getAuthor() { return author; }  Date getPubDate() { return pubDate; }  void setTitle(string bk\_title) { title = bk\_title; }  void setAuthor(string bk\_author) { author = bk\_author; }  private:  string title;  string author;  Date pubDate;  };  #endif // !B\_H |
| /\* T\_Entry.h \*/  #ifndef TE\_H  #define TE\_H  #include <fstream>  #include <iomanip>  using namespace std;  template<typename K, typename V>  class T\_Entry  {  friend ostream& operator<<(ostream& fout, T\_Entry<K, V>& entry)  {  if (entry.getValue() != NULL)  fout << "(" << setw(8) << entry.getKey() << ": " << \*(entry.getValue()) << ")";  return fout;  }  public:  T\_Entry(K key, V value) { \_key = key; \_value = value; }  T\_Entry() { } // default constructor  ~T\_Entry() {}  void setKey(const K& key) { \_key = key; }  void setValue(const V& value) { \_value = value; }  K getKey() const { return \_key; }  V getValue() const { return \_value; }  bool operator>(const T\_Entry<K, V>& right) const { return (\_key > right.getKey()); }  bool operator>=(const T\_Entry<K, V>& right) const { return (\_key >= right.getKey()); }  bool operator<(const T\_Entry<K, V>& right) const { return (\_key < right.getKey()); }  bool operator<=(const T\_Entry<K, V>& right) const { return (\_key <= right.getKey()); }  bool operator==(const T\_Entry<K, V>& right) const {  return ((\_key == right.getKey()) && (\_value ==  right.getValue()));  }  T\_Entry<K, V>& operator=(T\_Entry<K, V>& right);  void fprint(ostream& fout);  private:  K \_key;  V \_value;  };  template<typename K, typename V>  T\_Entry<K, V>& T\_Entry<K, V>::operator=(T\_Entry<K, V>& right)  {  this->\_key = right.getKey();  this->\_value = right.getValue();  return \*this;  }  template<typename K, typename V>  void T\_Entry<K, V>::fprint(ostream& fout)  {  fout << "[Key:" << setw(2) << this->getKey() << ", " << \*(this->getValue()) << "]";  }  #endif // !TE\_H |
| /\* Template class T\_Array.h \*/  #ifndef T\_Array\_H  #define T\_Array\_H  #include <iostream>  #include <iomanip>  using namespace std;  enum SortingOrder { INCREASING, DECREASING };  template<typename T>  class T\_Array {  public:  T\_Array(int n, string nm); // constructor  ~T\_Array(); // destructor  int size() { return num\_elements; }  string getName() { return name; }  void reserve(int new\_capacity);  void insert(int i, T element);  void insertBack(T element);  void remove(int i);  T& at(int i);  void set(int i, T& element);  void shuffle();  void selection\_sort(SortingOrder sortOrder = INCREASING);  void quick\_sort(SortingOrder sortOrder = INCREASING);  void fprint(ofstream& fout, int elements\_per\_line);  bool isValidIndex(int i);  T& operator[](int index) { return t\_array[index]; }  private:  T\* t\_array;  int num\_elements;  int capacity;  string name;  };  template<typename T>  T\_Array<T>::T\_Array(int new\_capacity, string nm) { // constructor  t\_array = (T\*) new T[new\_capacity];  if (t\_array == NULL) {  cout << "Error in creation of dynamic array of size (" << new\_capacity << ") !!" << endl;  exit;  }  capacity = new\_capacity;  num\_elements = 0;  name = nm;  }  template<typename T>  T\_Array<T>::~T\_Array() { // destructor  if (t\_array != NULL)  delete[] t\_array;  }  template<typename T>  bool T\_Array<T>::isValidIndex(int index) {  if ((index < 0) || (index > num\_elements))  return false;  else  return true;  }  template<typename T>  void T\_Array<T>::reserve(int new\_capacity) {  if (capacity >= new\_capacity)  return; // already big enough  T\* t\_newGA = (T\*) new T[new\_capacity];  if (t\_newGA == NULL)  {  cout << "Error in creation of dynamic array of size (" << new\_capacity << ") !!" << endl;  exit;  }  cout << this->getName() << " expands capacity to " << setw(3)  << new\_capacity << endl;  for (int i = 0; i < num\_elements; i++)  t\_newGA[i] = t\_array[i];  delete[] t\_array;  t\_array = t\_newGA;  capacity = new\_capacity;  }  template<typename T>  void T\_Array<T>::insert(int i, T new\_element) {  if (num\_elements >= capacity) // full ?  {  int new\_capa;  new\_capa = ((2 \* capacity) > 1) ? 2 \* capacity : 1;  reserve(new\_capa);  }  if (isValidIndex(i))  {  for (int j = num\_elements - 1; j >= i; j--)  t\_array[j + 1] = t\_array[j]; //shift up elements in one position  t\_array[i] = new\_element;  num\_elements++;  }  }  template<typename T>  void T\_Array<T>::insertBack(T new\_element) {  if (num\_elements >= capacity) // full ?  {  int new\_capa;  new\_capa = ((2 \* capacity) > 1) ? 2 \* capacity : 1;  reserve(new\_capa);  }  t\_array[num\_elements] = new\_element;  num\_elements++;  }  template<typename T>  void T\_Array<T>::remove(int i) {  if (isValidIndex(i))  {  for (int j = i + 1; j < num\_elements; j++)  t\_array[j - 1] = t\_array[j]; //shift down elements in one position  num\_elements--;  }  if (num\_elements < (capacity / 2))  {  int new\_capacity = capacity / 2;  T\* t\_newGA = (T\*) new T[new\_capacity];  if (t\_newGA == NULL)  {  return; // new memory allocation failed.  // Just return without modification.  }  cout << this->getName()  << " reduces capacity to " << setw(3)  << new\_capacity << endl;  for (int i = 0; i < num\_elements; i++)  t\_newGA[i] = t\_array[i];  delete[] t\_array;  t\_array = t\_newGA;  capacity = new\_capacity;  }  }  template<typename T>  T& T\_Array<T>::at(int i) {  if (isValidIndex(i))  return t\_array[i];  }  template<typename T>  void T\_Array<T>::set(int i, T& element) {  if (isValidIndex(i))  t\_array[i] = element;  }  template<typename T>  void T\_Array<T>::shuffle() {  srand(time(0));  int index1, index2;  int rand\_1, rand\_2;  T temp;  for (int i = 0; i < num\_elements; i++) {  rand\_1 = rand();  rand\_2 = rand();  index1 = ((rand\_1 << 15) | rand\_2) % num\_elements;  rand\_1 = rand();  rand\_2 = rand();  index2 = ((rand\_1 << 15) | rand\_2) % num\_elements;  temp = t\_array[index1];  t\_array[index1] = t\_array[index2];  t\_array[index2] = temp;  }  }  template<typename T>  void T\_Array<T>::selection\_sort(SortingOrder sortOrder) {  int index\_min, index\_max; // index of the element with minimum value  T minValue; // minimum value  T maxValue;  for (int i = 0; i < num\_elements - 1; i++)  {  if (sortOrder == INCREASING) { // sorting in increasing (non\_decreasing) order  index\_min = i;  minValue = t\_array[i];  for (int j = i + 1; j < num\_elements; j++)  {  if (t\_array[j] < minValue) // T must provide operator<() overloading !!  {  index\_min = j;  minValue = t\_array[j];  }  }  if (index\_min != i) // if a smaller element is found, then swap  {  /\* minValue is t\_array[min] \*/  t\_array[index\_min] = t\_array[i];  t\_array[i] = minValue;  }  }  else { // sorting in decreasing (non\_increasing) order  index\_max = i;  maxValue = t\_array[i];  for (int j = i + 1; j < num\_elements; j++)  {  if (t\_array[j] > maxValue) // T must provide operator>() overloading !!  {  index\_max = j;  maxValue = t\_array[j];  }  }  if (index\_max != i) // if a smaller element is found, then swap  {  /\* maxValue is t\_array[max] \*/  t\_array[index\_max] = t\_array[i];  t\_array[i] = maxValue;  }  }  } // end for  }  template<typename T>  int \_partition(T\* array, int size, int left, int right, int pivotIndex,  SortingOrder sortOrder = INCREASING) {  T pivotValue, temp; // pivot value  int newPI; // new pivot index  /\* place the pivot element at right-position \*/  pivotValue = array[pivotIndex];  array[pivotIndex] = array[right];  array[right] = pivotValue; // Move pivot to array[right]  newPI = left; // newPI is the index that points the position  // where pivot element will be finally re-located  for (int i = left; i <= (right - 1); i++) {  if (sortOrder == INCREASING) // sorting in increasing order  {  if (array[i] <= pivotValue) // T must provide operator<=() overloading !!  {  temp = array[i];  array[i] = array[newPI];  array[newPI] = temp;  newPI = newPI + 1;  // note: all elements in left of index newPI are equal or smaller than pivot\_value  }  }  else // sorting in decreasing (non\_increasing) order  {  if (array[i] > pivotValue) // T must provide operator>() overloading !!  {  temp = array[i];  array[i] = array[newPI];  array[newPI] = temp;  newPI = newPI + 1;  // note: all elements in left of index newPI are greater than pivot\_value  }  }  } // end for  // swap array[newPI] and array[right]; Move pivot element to its final place  temp = array[newPI];  array[newPI] = array[right];  array[right] = temp;  return newPI;  }  template<typename T>  void \_quick\_sort(T\* array, int size, int left, int right,  SortingOrder sortOrder = INCREASING) {  int pI, newPI; // pivot index  if (left >= right)  {  return;  }  else  {//select a pI (pivotIndex) in the range left ≤ pI ≤ right  pI = (left + right) / 2;  }  newPI = \_partition(array, size, left, right, pI, sortOrder);  if (left < (newPI - 1)) {  \_quick\_sort(array, size, left, newPI - 1, sortOrder);  // recursively sort elements on the left of pivotNewIndex  }  if ((newPI + 1) < right) {  \_quick\_sort(array, size, newPI + 1, right, sortOrder);  // recursively sort elements on the right of pivotNewIndex  }  }  template<typename T>  void T\_Array<T>::quick\_sort(SortingOrder sortOrder) {  int pI, newPI; // pivot index  \_quick\_sort(this->t\_array, num\_elements, 0, num\_elements - 1, sortOrder);  }  template<typename T>  void T\_Array<T>::fprint(ofstream& fout, int elements\_per\_line)  {  int count = 0;  while (count < num\_elements)  {  for (int i = 0; i < elements\_per\_line; i++)  {  fout << t\_array[count] << " ";  count++;  if (count % elements\_per\_line == 0)  fout << endl;  }  }  fout << endl;  }  #endif |
| /\* Template\_Binary\_Search\_Tree\_Node.h \*/  #ifndef T\_BSTN\_H  #define T\_BSTN\_H  #include "T\_Entry.h"  template<typename K, typename V>  class T\_BSTN { // a node of the tree  public:  T\_BSTN() : entry(), pPr(NULL), pLc(NULL), pRc(NULL) { } // default constructor  T\_BSTN(T\_Entry<K, V> e) : entry(e), pPr(NULL), pLc(NULL), pRc(NULL) { } // constructor  K getKey() { return entry.getKey(); }  V getValue() { return entry.getValue(); }  T\_Entry<K, V>& getEntry() { return entry; }  void setEntry(T\_Entry<K, V> e) { entry = e; }  T\_BSTN<K, V>\* getpPr() { return pPr; }  T\_BSTN<K, V>\* getpLc() { return pLc; }  T\_BSTN<K, V>\* getpRc() { return pRc; }  T\_BSTN<K, V>\*\* getppLc() { return &pLc; }  T\_BSTN<K, V>\*\* getppRc() { return &pRc; }  void setpPr(T\_BSTN<K, V>\* pTN) { pPr = pTN; }  void setpLc(T\_BSTN<K, V>\* pTN) { pLc = pTN; }  void setpRc(T\_BSTN<K, V>\* pTN) { pRc = pTN; }  T\_Entry<K, V>& operator\*() { return entry; }  private:  T\_Entry<K, V> entry; // element value  T\_BSTN<K, V>\* pPr; // parent  T\_BSTN<K, V>\* pLc; // left child  T\_BSTN<K, V>\* pRc; // right child  };  #endif // !T\_BSTN\_H |
| /\* Template\_Binary\_Search\_Tree.h \*/  #ifndef T\_BST\_H  #define T\_BST\_H  #include "T\_BSTN.h"  #include "T\_Array.h"  template<typename K, typename V>  class T\_BST  {  public:  T\_BST(string nm) : \_root(NULL), num\_entry(0), name(nm) {} // constructor  string getName() { return name; }  int size() const { return num\_entry; }  bool empty() const { return num\_entry == 0; }  void clear() {}  T\_BSTN<K, V>\* getRoot() { return \_root; }  T\_BSTN<K, V>\*\* getRootAddr() { return &\_root; }  T\_Entry<K, V>& getRootEntry() { return \_root->getEntry(); }  T\_BSTN<K, V>\* eraseBSTN(T\_BSTN<K, V>\*\* pp);  void insertInOrder(const T\_Entry<K, V> entry);  void insertAndRebalance(T\_Entry<K, V> e);  void traversal\_inOrder(T\_BSTN<K, V>\* p, T\_Array<V>& array\_value);  void traversal\_preOrder(T\_BSTN<K, V>\* pos, T\_Array<V>& array\_value);  void traversal\_postOrder(T\_BSTN<K, V>\* pos, T\_Array<V>& array\_value);  T\_BSTN<K, V>\* searchBSTN(K k);  V searchBST(K k);  T\_Entry<K, V>& minEntry();  T\_Entry<K, V>& maxEntry();  void fprint\_with\_Depth(ostream& fout);  void fprint\_inOrder(ostream& fout);  protected:  T\_BSTN<K, V>\* \_maxBSTN(T\_BSTN<K, V>\* subRoot);  T\_BSTN<K, V>\* \_minBSTN(T\_BSTN<K, V>\* subRoot);  T\_BSTN<K, V>\* \_insertInOrder(T\_BSTN<K, V>\*\* p, T\_BSTN<K, V>\* parenPos, const T\_Entry<K, V> e);  T\_BSTN<K, V>\* \_insertAndRebalance(T\_BSTN<K, V>\*\* ppTN, T\_BSTN<K, V>\* pPr, T\_Entry<K, V> e);  T\_BSTN<K, V>\* \_rotate\_LL(T\_BSTN<K, V>\* pCurSubRoot);  T\_BSTN<K, V>\* \_rotate\_RR(T\_BSTN<K, V>\* pCurSubRoot);  T\_BSTN<K, V>\* \_rotate\_RL(T\_BSTN<K, V>\* pCurSubRoot);  T\_BSTN<K, V>\* \_rotate\_LR(T\_BSTN<K, V>\* pCurSubRoot);  int \_getHeight(T\_BSTN<K, V>\* pTN);  int \_getHeightDiff(T\_BSTN<K, V>\* pTN);  T\_BSTN<K, V>\* \_reBalance(T\_BSTN<K, V>\*\* ppTN);  T\_BSTN<K, V>\* \_searchBSTN(T\_BSTN<K, V>\* pos, K k);  void \_fprint\_with\_Depth(T\_BSTN<K, V>\* pTN, ostream& fout, int depth);  void \_fprint\_inOrder(T\_BSTN<K, V>\* pTN, ostream& fout);  private:  T\_BSTN<K, V>\* \_root; // pointer to the root  int num\_entry; // number of tree nodes  string name;  }; // end of class T\_BST  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_insertInOrder(T\_BSTN<K, V>\*\* pp,  T\_BSTN<K, V>\* parenPos, const T\_Entry<K, V> entry)  {  T\_BSTN<K, V>\* newPos, \*\* pChildPos;  T\_BSTN<K, V>\* pos;  T\_Entry<K, V> ent;  if (pp == NULL) // 구성이 제대로 안됨  {  cout << "Error in creation of BinarySearchTree :";  cout << " address of the pointer to the Root Node is NULL !!₩n";  exit;  }  pos = \*pp;  if (pos == NULL) // 트리가 비어있다.  {  pos = new T\_BSTN<K, V>(entry);  if (parenPos == NULL) // 부모가 없다 => root  {  \_root = pos; // initialize the root node  }  pos->setpPr(parenPos);  \*pp = pos; // 위에서 동적할당한 노드의 주소  num\_entry++; // increment the number of elements  return pos;  }  ent = pos->getEntry();  if (entry < ent)  {  pChildPos = pos->getppLc();  newPos = \_insertInOrder(pChildPos, pos, entry);  if (newPos != NULL)  pos->setpLc(newPos);  return NULL; // only the leaf child is set correctly, while the intermediate node is skipped  }  else if (entry >= ent)  {  pChildPos = pos->getppRc();  newPos = \_insertInOrder(pChildPos, pos, entry);  if (newPos != NULL)  pos->setpRc(newPos);  return NULL; // only the leaf child is set correctly, while the intermediate node is skipped  }  }  template<typename K, typename V>  void T\_BST<K, V>::insertInOrder(const T\_Entry<K, V> entry)  {  \_insertInOrder(&\_root, NULL, entry);  }  template<typename K, typename V>  void T\_BST<K, V>::traversal\_inOrder(T\_BSTN<K, V>\* pos, T\_Array<V>& array\_value)  {  T\_BSTN<K, V>\* pLc, \* pRc;  T\_Entry<K, V> entry;  V value;  if (pos == NULL)  return;  pLc = pos->getpLc();  pRc = pos->getpRc();  traversal\_inOrder(pLc, array\_value);  entry = pos->getEntry();  value = entry.getValue();  array\_value.insertBack(value);  traversal\_inOrder(pRc, array\_value);  }  template<typename K, typename V>  void T\_BST<K, V>::traversal\_preOrder(T\_BSTN<K, V>\* pos, T\_Array<V>& array\_value)  {  T\_BSTN<K, V>\* pLc, \* pRc;  T\_Entry<K, V> entry;  V value;  if (pos == NULL)  return;  pLc = pos->getpLc();  pRc = pos->getpRc();  entry = pos->getEntry();  value = entry.getValue();  array\_value.insertBack(value);  traversal\_preOrder(pLc, array\_value);  traversal\_preOrder(pRc, array\_value);  }  template<typename K, typename V>  void T\_BST<K, V>::traversal\_postOrder(T\_BSTN<K, V>\* pos, T\_Array<V>& array\_value)  {  T\_BSTN<K, V>\* pLc, \* pRc;  T\_Entry<K, V> entry;  V value;  if (pos == NULL)  return;  pLc = pos->getpLc();  pRc = pos->getpRc();  traversal\_postOrder(pLc, array\_value);  traversal\_postOrder(pRc, array\_value);  entry = pos->getEntry();  value = entry.getValue();  array\_value.insertBack(value);  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_searchBSTN(T\_BSTN<K, V>\* pos, K k)  {  K ent\_k;  T\_BSTN<K, V>\* pos\_result = NULL;  if (pos == NULL)  return NULL;  ent\_k = pos->getKey();  if (ent\_k == k)  pos\_result = pos;  // given entry was found here !!  else if (ent\_k > k)  pos\_result = \_searchBSTN(pos->getpLc(), k);  else if (ent\_k < k)  pos\_result = \_searchBSTN(pos->getpRc(), k);  return pos\_result;  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::searchBSTN(K key)  {  T\_BSTN<K, V>\* pEntry;  pEntry = \_searchBSTN(\_root, key);  return pEntry;  }  template<typename K, typename V>  V T\_BST<K, V>::searchBST(K key)  {  T\_BSTN<K, V>\* pEntry;  V value;  pEntry = \_searchBSTN(\_root, key);  value = pEntry->getValue();  return value;  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::  \_minBSTN(T\_BSTN<K, V>\* subRoot)  {  T\_BSTN<K, V>\* pos, \* pLc;  if ((subRoot == NULL) ||  (NULL == subRoot->getpLc()))  return subRoot;  pos = subRoot;  while ((pos->getpLc()) != NULL)  pos = pos->getpLc();  return pos;  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::  \_maxBSTN(T\_BSTN<K, V>\* subRoot)  {  T\_BSTN<K, V>\* pos, \* pLc;  if ((subRoot == NULL) ||  (NULL == subRoot->getpRc()))  return subRoot;  pos = subRoot;  while ((pos->getpRc()) != NULL)  pos = pos->getpRc();  return pos;  }  template<typename K, typename V>  T\_Entry<K, V>& T\_BST<K, V>::minEntry()  {  T\_BSTN<K, V>\* pMin;  pMin = \_minBSTN(\_root);  return pMin->getEntry();  }  template<typename K, typename V>  T\_Entry<K, V>& T\_BST<K, V>::maxEntry()  {  T\_BSTN<K, V>\* pMax;  pMax = \_maxBSTN(\_root);  return pMax->getEntry();  }  template<typename K, typename V>  void T\_BST<K, V>::\_fprint\_inOrder(T\_BSTN<K, V>\* pTN, ostream& fout)  {  T\_BSTN<K, V>\* pRc, \* pLc;  if ((pLc = pTN->getpLc()) != NULL)  \_fprint\_inOrder(pLc, fout);  fout << pTN->getEntry() << endl;  if ((pRc = pTN->getpRc()) != NULL)  \_fprint\_inOrder(pRc, fout);  }  template<typename K, typename V>  void T\_BST<K, V>::fprint\_inOrder(ostream& fout)  {  T\_BSTN<K, V>\* root = getRoot();  if (num\_entry == 0)  {  fout << getName() << " is empty now !!" << endl;  return;  }  \_fprint\_inOrder(root, fout);  }  template<typename K, typename V>  void T\_BST<K, V>::\_fprint\_with\_Depth(T\_BSTN<K, V>\* pTN, ostream& fout, int depth)  {  T\_BSTN<K, V>\* pRc, \* pLc;  T\_Entry<K, V>\* pEntry;  if ((pRc = pTN->getpRc()) != NULL)  \_fprint\_with\_Depth(pRc, fout, depth + 1);  for (int i = 0; i < depth; i++)  {  fout << " ";  }  fout << pTN->getEntry() << endl;  if ((pLc = pTN->getpLc()) != NULL)  \_fprint\_with\_Depth(pLc, fout, depth + 1);  }  template<typename K, typename V>  void T\_BST<K, V>::fprint\_with\_Depth(ostream& fout)  {  T\_BSTN<K, V>\* root = getRoot();  if (num\_entry == 0)  {  fout << getName() << " is empty now !!" << endl;  return;  }  \_fprint\_with\_Depth(root, fout, 0);  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::eraseBSTN(T\_BSTN<K, V>\*\* pptoBeErased)  // remove BSTN considering balance of the BST  {  T\_BSTN<K, V>\* newSubRoot, \* temp, \* w, \* wlc;  T\_BSTN<K, V>\* toBeErased;  toBeErased = \*pptoBeErased;  if (toBeErased == NULL)  return NULL;  if ((toBeErased->getpLc() == NULL) && (toBeErased->getpRc() == NULL))  // no child  {  newSubRoot = NULL;  }  else if ((toBeErased->getpLc() != NULL) && (toBeErased->getpRc() == NULL))  // only left child  {  newSubRoot = toBeErased->getpLc();  newSubRoot->setpPr(toBeErased->getpPr());  }  else if ((toBeErased->getpLc() == NULL) && (toBeErased->getpRc() != NULL))  // only right child  {  newSubRoot = toBeErased->getpRc();  newSubRoot->setpPr(toBeErased->getpPr());  }  else  { /\* tree node to be deleted has both left child and right child \*/  int heightDiff = \_getHeightDiff(toBeErased);  T\_BSTN<K, V>\* parDel = toBeErased->getpPr();  T\_BSTN<K, V>\* lChild = toBeErased->getpLc();  T\_BSTN<K, V>\* rChild = toBeErased->getpRc();  T\_BSTN<K, V>\* ioSs = NULL, \* rcIoSs, \* parIoSs;  T\_BSTN<K, V>\* ioPd = NULL, \* lcIoPd, \* parIoPd;  if (heightDiff > 0)  // left subtree is higher, so put the ioPd in the place of the erased node  {  ioPd = \_maxBSTN(lChild); // in-order predecessor (ioPd)  lcIoPd = ioPd->getpLc();  parIoPd = ioPd->getpPr();  newSubRoot = ioPd;  if (ioPd->getpPr() != toBeErased)  {  newSubRoot->setpLc(lChild);  parIoPd->setpRc(lcIoPd);  if (lcIoPd != NULL)  lcIoPd->setpPr(parIoPd);  }  newSubRoot->setpRc(rChild);  newSubRoot->setpPr(toBeErased->getpPr());  }  else  // right subtree is higher, so put the ioSs in the place of the erased node  {  ioSs = \_minBSTN(rChild); // in-order successor (ioSs)  rcIoSs = ioSs->getpRc();  parIoSs = ioSs->getpPr();  newSubRoot = ioSs;  if (ioSs->getpPr() != toBeErased)  {  newSubRoot->setpRc(rChild);  parIoSs->setpLc(rcIoSs);  if (rcIoSs != NULL)  rcIoSs->setpPr(parIoSs);  }  newSubRoot->setpLc(lChild);  newSubRoot->setpPr(toBeErased->getpPr());  }  if (lChild != ioPd)  lChild->setpPr(newSubRoot);  if (rChild != ioSs)  rChild->setpPr(newSubRoot);  }  if (toBeErased == \_root)  \_root = newSubRoot;  num\_entry--; // decrement the number of entries in the BST  free(toBeErased);  \*pptoBeErased = newSubRoot;  return newSubRoot;  }  template<typename K, typename V>  int T\_BST<K, V>::\_getHeight(T\_BSTN<K, V>\* pTN)  {  int height = 0;  int height\_Lc, height\_Rc;  if (pTN != NULL)  {  height\_Lc = \_getHeight(pTN->getpLc());  height\_Rc = \_getHeight(pTN->getpRc());  if (height\_Lc > height\_Rc)  height = 1 + height\_Lc;  else  height = 1 + height\_Rc;  }  return height;  }  template<typename K, typename V>  int T\_BST<K, V>::  \_getHeightDiff(T\_BSTN<K, V>\* pTN)  {  int heightDiff = 0;  if (pTN == NULL)  return 0;  heightDiff = \_getHeight(pTN->getpLc())  - \_getHeight(pTN->getpRc());  return heightDiff;  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_rotate\_LL(T\_BSTN<K, V>\* pCurSubRoot)  {  T\_BSTN<K, V>\* pNewSubRoot, \* pBR, \* pCurParent;  pCurParent = pCurSubRoot->getpPr(); // 현재 서브루트의 부모를 현재부모로 설정  pNewSubRoot = pCurSubRoot->getpLc(); // 현재 서브루트의 왼쪽 자식을 새로운 서브루트로 설정  pBR = pNewSubRoot->getpRc(); // 새로운 서브루트의 오른쪽 자식을 BR로 설정  pCurSubRoot->setpLc(pBR); // 현재 서브루트의 왼쪽 자식을 BR로 설정  if (pBR != NULL)  pBR->setpPr(pCurSubRoot); // BR의 부모를 현재 서브루트로 설정  pNewSubRoot->setpRc(pCurSubRoot); // 새로운 서브루트의 오른쪽 자식을 현재 서브루트로 설정  pNewSubRoot->setpPr(pCurParent); // 새로운 서브루트의 부모를 현재 부모로 설정  pCurSubRoot->setpPr(pNewSubRoot); // 현재 서브루트의 부모를 새로운 서브루트로 설정  return pNewSubRoot; // 새로운 서브루트 반환  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_rotate\_RR(T\_BSTN<K, V>\* pCurSubRoot)  {  T\_BSTN<K, V>\* pNewSubRoot, \* pBL, \* pCurParent;  pCurParent = pCurSubRoot->getpPr(); // 현재 서브루트의 부모를 현재 부모로 설정  pNewSubRoot = pCurSubRoot->getpRc(); // 현재 서브루트의 오른쪽 자식을 새로운 서브루트로 설정  pBL = pNewSubRoot->getpLc();// 새로운 서브루트의 왼쪽 자식을 BL로 설정  pCurSubRoot->setpRc(pBL);// 현재 서브루트의 오른쪽 자식을 BR로 설정  if (pBL != NULL)  pBL->setpPr(pCurSubRoot);// BL의 부모를 현재 서브루트로 설정  pNewSubRoot->setpLc(pCurSubRoot);// 새로운 서브루트의 왼쪽 자식을 현재 서브루트로 설정  pNewSubRoot->setpPr(pCurParent);// 새로운 서브루트의 부모를 현재 부모로 설정  pCurSubRoot->setpPr(pNewSubRoot); // 현재 서브루트의 부모를 새로운 서브루트로 설정  return pNewSubRoot;// 새로운 서브루트 반환  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_rotate\_LR(T\_BSTN<K, V>\* pCurSubRoot)  {  T\_BSTN<K, V>\* pSubRoot, \* pNewSubRoot, \* pCurParent;  T\_BSTN<K, V>\* pA, \* pB, \* pC, \* pBL, \* pBR;  pC = pCurSubRoot; // 현재 서브루트  pCurParent = pCurSubRoot->getpPr(); // 현재부모  pA = pC->getpLc(); // A를 C의 왼쪽 자식으로 설정  pB = pA->getpRc(); // B를 A의 오른쪽 자식으로 설정  pBL = pB->getpLc(); // BL을 B의 왼쪽 자식으로 설정  pBR = pB->getpRc(); // BR을 B의 오른쪽 자식으로 설정  pSubRoot = \_rotate\_RR(pA); // A 기준 rotateRR을 해서 서브루트 반환  pCurSubRoot->setpLc(pSubRoot); // 현재 서브루트의 왼쪽 자식을 서브루트로 설정  pNewSubRoot = \_rotate\_LL(pC); // C 기준 rotateLL을 해서 서브루트 반환  pNewSubRoot->setpPr(pCurParent); // 새로운 서브루트의 부모를 현재 부모로 설정  pA->setpPr(pNewSubRoot); // A의 부모를 새로운 서브루트로 설정  pC->setpPr(pNewSubRoot); // C의 부모를 새로운 서브루트로 설정  if (pBL != NULL)  pBL->setpPr(pA); // BL의 부모를 A로 설정  if (pBR != NULL)  pBR->setpPr(pC); // BR의 부모를 C로 설정  return pNewSubRoot; // 새로운 서브루트 반환  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_rotate\_RL(T\_BSTN<K, V>\* pCurSubRoot)  {  T\_BSTN<K, V>\* pSubRoot, \* pNewSubRoot, \* pCurParent;  T\_BSTN<K, V>\* pA, \* pB, \* pC, \* pBL, \* pBR;  pA = pCurSubRoot; // 현재 서브루트  pCurParent = pCurSubRoot->getpPr(); // 현재 부모  pC = pA->getpRc(); // C를 A의 오른쪽 자식으로 설정  pB = pC->getpLc(); // B를 C의 왼쪽 자식으로 설정  pBL = pB->getpLc(); // BL을 B의 왼쪽 자식으로 설정  pBR = pB->getpRc(); // BR을 B의 오른쪽 자식으로 설정  pSubRoot = \_rotate\_LL(pC); // C 기준 rotateLL을 통해 서브루트 반환  pCurSubRoot->setpRc(pSubRoot); // 현재 서브루트의 오른쪽 자식을 서브루트로 설정  pNewSubRoot = \_rotate\_RR(pA); // A 기준 rotateRR을 통해 새로운 서브루트로 반환  pNewSubRoot->setpPr(pCurParent); // 새로운 서브루트의 부모를 현재 부모로 설정  pA->setpPr(pNewSubRoot); // A의 부모를 새로운 서브루트로 설정  pC->setpPr(pNewSubRoot); // C의 부모를 새로운 서브루트로 설정  if (pBL != NULL)  pBL->setpPr(pA); // BL의 부모를 A로 설정  if (pBR != NULL)  pBR->setpPr(pC); // BR의 부모를 C로 설정  return pNewSubRoot; // 새로운 서브루트 반환  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_reBalance(T\_BSTN<K, V>\*\* ppTN)  {  int heightDiff = 0;  heightDiff = \_getHeightDiff(\*ppTN);  if (heightDiff > 1) // left subtree is higher  {  if (\_getHeightDiff((\*ppTN)->getpLc()) > 0)  \*ppTN = \_rotate\_LL(\*ppTN);  else  \*ppTN = \_rotate\_LR(\*ppTN);  }  else if (heightDiff < -1) // right subtree is higher  {  if (\_getHeightDiff((\*ppTN)->getpRc()) < 0)  \*ppTN = \_rotate\_RR(\*ppTN);  else  \*ppTN = \_rotate\_RL(\*ppTN);  }  return \*ppTN;  }  template<typename K, typename V>  void T\_BST<K, V>::insertAndRebalance(T\_Entry<K, V> entry)  {  \_insertAndRebalance(&\_root, NULL, entry);  }  template<typename K, typename V>  T\_BSTN<K, V>\* T\_BST<K, V>::\_insertAndRebalance(T\_BSTN<K, V>\*\* ppTN,  T\_BSTN<K, V>\* pPr, T\_Entry<K, V> entry)  {  T\_BSTN<K, V>\* pTN, \*\* ppLc, \*\* ppRc;  if (\*ppTN == NULL) // attach a new tree node at the currently external node  {  pTN = new T\_BSTN<K, V>(entry);  \*ppTN = pTN;  if (pPr != NULL) // if not root  pTN->setpPr(pPr);  (\*ppTN)->setpLc(NULL);  (\*ppTN)->setpRc(NULL);  num\_entry++;  return \*ppTN;  }  T\_Entry<K, V> bstn\_entry;  bstn\_entry = (\*ppTN)->getEntry();  if (entry < bstn\_entry) // T\_Entry<K, V> must provide ‘<’ operator overloading !!  {  ppLc = (\*ppTN)->getppLc();  pTN = \_insertAndRebalance(ppLc, \*ppTN, entry);  if (ppTN != NULL)  {  (\*ppTN)->setpLc(pTN);  \*ppTN = \_reBalance(ppTN);  }  }  else // entry >= bstn\_entry  {  ppRc = (\*ppTN)->getppRc();  pTN = \_insertAndRebalance(ppRc, \*ppTN, entry);  if (ppTN != NULL)  {  (\*ppTN)->setpRc(pTN);  \*ppTN = \_reBalance(ppTN);  }  }  return \*ppTN;  }  #endif // !T\_BST\_H |
|  |