**Homework 8**

|  |
| --- |
| 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 | 2021.11.03.수 |

|  |
| --- |
| /\* main.cpp \*/  /\* Description  \* Heap priority queue 구현  \* Programmed by J. H. Kim  \* Last updated : 2021-11-03 \*/  #include <iostream>  #include <iomanip>  #include <fstream>  #include "HeapPrioQ.h"  #include <string>  #include <stdlib.h>  using namespace std;  #define INITIAL\_CBT\_CAPA 100  #define NUM\_DATA 15  void main() {  ofstream fout;  int size;  int\* pD;  int inputArray[NUM\_DATA] =  { 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };  fout.open("output.txt");  if (fout.fail())  {  cout << "Fail to open output.txt file for results !!" << endl;  exit;  }  HeapPrioQueue<int> HeapPriQ\_int(INITIAL\_CBT\_CAPA, string("Heap\_Priority\_Queue\_Int"));  for (int i = 0; i < NUM\_DATA; i++)  {  HeapPriQ\_int.insert(inputArray[i]);  fout << "Insert " << setw(3) << inputArray[i];  fout << " ==> Size of Heap Priority Queue : " << setw(3) << HeapPriQ\_int.size() << endl;  }  fout << "Final status of insertions : " << endl;  HeapPriQ\_int.fprintCBT\_byLevel(fout);  for (int i = 0; i < NUM\_DATA; i++)  {  fout << "\nCurrent top priority in Heap Priority Queue : ";  pD = HeapPriQ\_int.getHeapMin();  fout << setw(3) << \*pD << endl;  pD = HeapPriQ\_int.removeHeapMin();  fout << "RemoveMin (" << \*pD << ") from HeapPriQ\_int";  fout << " ==> " << HeapPriQ\_int.size() << " elements remains." << endl;  HeapPriQ\_int.fprintCBT\_byLevel(fout);  fout << endl;  }  fout.close();  } // end main(); |
| /\* TA\_Entry.h \*/  #ifndef TA\_E\_H  #define TA\_E\_H  #include <iostream>  using namespace std;  enum SortingDirection { INCREASING, DECREASING };  template<typename K>  class TA\_Entry {  public:  TA\_Entry(int n, string nm);  ~TA\_Entry();  int size() { return num\_elements; }  bool empty() { return num\_elements == 0; }  string getName() { return name; }  void reserve(int new\_capacity);  void insert(int i, K element);  void remove(int i);  K& at(int i);  void set(int i, K element);  K getMin(int begin, int end);  K getMax(int begin, int end);  void shuffle();  int sequential\_search(K search\_key);  int binary\_search(K search\_key);  void selection\_sort(SortingDirection sd);  void quick\_sort(SortingDirection sd);  void fprint(ofstream& fout, int elements\_per\_line);  void fprintSample(ofstream& fout, int elements\_per\_line, int num\_sample\_lines);  bool isValidIndex(int i);  K& operator[](int index) { return t\_GA[index]; }  protected:  K\* t\_GA;  int num\_elements;  int capacity;  string name;  };  template<typename K>  TA\_Entry<K>::TA\_Entry(int n, string nm) {  t\_GA = (K\*) new K[n];  if (t\_GA == NULL) {  cout << "Error in creation of dynamic array of size (" << n << ") !!" << endl;  exit;  }  capacity = n;  num\_elements = 0;  name = nm;  }  template<typename K>  TA\_Entry<K>::~TA\_Entry() {  if (t\_GA != NULL) delete[] t\_GA;  }  template<typename K>  void TA\_Entry<K>::reserve(int new\_capacity) {  if (capacity >= new\_capacity)  return; // already big enough  K\* t\_newGA = (K\*) new K[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\_GA[i];  delete[] t\_GA;  t\_GA = t\_newGA;  capacity = new\_capacity;  }  template<typename K>  void TA\_Entry<K>::insert(int i, K element) {  if (isValidIndex(i)) {  for (int j = num\_elements - 1; j >= i; j--)  t\_GA[j + 1] = t\_GA[j]; //shift up elements in one position  t\_GA[i] = element;  num\_elements++;  }  }  template<typename K>  void TA\_Entry<K>::remove(int i) {  if (isValidIndex(i))  {  for (int j = i + 1; j < num\_elements; j++)  t\_GA[j - 1] = t\_GA[j]; //shift down elements in one position  num\_elements--;  }  if (num\_elements < (capacity / 2))  {  int new\_capacity = capacity / 2;  K\* t\_newGA = (K\*) new K[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\_GA[i];  delete[] t\_GA;  t\_GA = t\_newGA;  capacity = new\_capacity;  }  }  template<typename K>  K& TA\_Entry<K>::at(int i) {  if (isValidIndex(i))  {  return t\_GA[i];  }  }  template<typename K>  void TA\_Entry<K>::set(int i, K element) {  if (isValidIndex(i))  {  t\_GA[i] = element;  }  }  template<typename K>  K TA\_Entry<K>::getMin(int begin, int end) {  K minValue;  int index\_min;  minValue = t\_GA[begin];  index\_min = begin;  for (int i = begin + 1; i <= end; i++)  {  if (t\_GA[i] < minValue) // T must provide operator<() overloading !!  {  minValue = t\_GA[i];  index\_min = i;  }  }  return t\_GA[index\_min];  }  template<typename K>  K TA\_Entry<K>::getMax(int begin, int end) {  K maxValue;  int index\_max;  maxValue = t\_GA[begin]; index\_max = begin; for (int i = begin + 1; i <= end; i++)  {  if (t\_GA[i] > maxValue) // T must provide operator>() overloading !!  {  maxValue = t\_GA[i];  index\_max = i;  }  }  return t\_GA[index\_max];  }  template<typename K>  void TA\_Entry<K>::shuffle() {  srand(time(0));  int index1, index2;  int rand\_1, rand\_2;  K 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\_GA[index1];  t\_GA[index1] = t\_GA[index2];  t\_GA[index2] = temp;  }  }  template<typename K>  int TA\_Entry<K>::sequential\_search(K search\_key) {  int index;  K key;  if (search\_key == "ST\_ID") { // student\_ID  for (int index = 0; index < num\_elements; index++) {  t\_GA[index].getKey(search\_key, &key);  if (key == search\_key)  return index;  }  }  return -1;  }  template<typename K>  int TA\_Entry<K>::binary\_search(K search\_key) {  K key;  int low, mid, high; int loop = 1;  low = 0; high = num\_elements - 1;  while (low <= high) {  cout << setw(2) << loop << "-th loop: current search range [" << setw(3) << low << ", " << setw(3) << high << "]" << endl;  mid = (low + high) / 2;  t\_GA[mid].getKey(search\_key, &key);  if (key == search\_key)  return mid;  else if (key > search\_key)  high = mid - 1;  else  low = mid + 1; loop++;  }  }  template<typename K>  void TA\_Entry<K>::selection\_sort(SortingDirection sd) {  int index\_min, index\_max; // index of the element with minimum value  K tempElement;  K minKey, maxKey, key;  for (int i = 0; i < num\_elements - 1; i++) {  if (sd == INCREASING) { // sorting in increasing (non\_decreasing) order  index\_min = i;  key = t\_GA[i].getKey(); // 키 반환  minKey = (K)key;  for (int j = i + 1; j < num\_elements; j++) {  key = t\_GA[j].getKey();  if ((K)key < minKey) {  index\_min = j;  minKey = (K)key;  }  }  if (index\_min != i) { // if a smaller element is found, then swap  tempElement = t\_GA[index\_min];  t\_GA[index\_min] = t\_GA[i];  t\_GA[i] = tempElement;  }  }  else { // sorting in decreasing (non\_increasing) order  index\_max = i;  key = t\_GA[i].getKey();  maxKey = (K)key;  for (int j = i + 1; j < num\_elements; j++) {  key = t\_GA[j].getKey();  if ((K)key > maxKey) {  index\_max = j;  maxKey = (K)key;  }  }  if (index\_max != i) { // if a smaller element is found, then swap  tempElement = t\_GA[index\_max];  t\_GA[index\_max] = t\_GA[i];  t\_GA[i] = tempElement;  }  }  } // end for  }  template<typename T>  void \_quick\_sort(T\* array, int size, int left, int right,  SortingDirection sd = 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, sd);  if (left < (newPI - 1)) {  \_quick\_sort(array, size, left, newPI - 1, sd);  // recursively sort elements on the left of pivotNewIndex  }  if ((newPI + 1) < right) {  \_quick\_sort(array, size, newPI + 1, right, sd);  // recursively sort elements on the right of pivotNewIndex  }  }  template<typename K>  void TA\_Entry<K>::quick\_sort(SortingDirection sd) {  \_quick\_sort(t\_GA, size, 0, num\_elements - 1, sd);  }  template<typename K>  void TA\_Entry<K>::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\_GA[count] << " ";  count++;  if (count % elements\_per\_line == 0)  fout << endl;  }  }  cout << endl;  }  template<typename K>  void TA\_Entry<K>::fprintSample(ofstream& fout, int elements\_per\_line, int num\_sample\_lines) {  string T\_type;  int last\_block\_start;  int count = 0;  T\_type = typeid(K).name();  for (int i = 0; i < num\_sample\_lines; i++)  {  for (int j = 0; j < elements\_per\_line; j++)  {  if (count >= num\_elements)  {  fout << endl;  return;  }  if ((T\_type == string("int")) || (T\_type == string("double")) ||  (T\_type == string("class std::basic\_string<char,struct std::char\_traits<char>,class std::allocator<char> > ")))  fout << setw(10) << t\_GA[count];  else  fout << t\_GA[count] << " ";  count++;  } fout << endl;  }  if (count < (num\_elements - elements\_per\_line \* num\_sample\_lines))  count = num\_elements - elements\_per\_line \* num\_sample\_lines;  fout << " . . . . . " << endl;  for (int i = 0; i < num\_sample\_lines; i++)  {  for (int j = 0; j < elements\_per\_line; j++)  {  if (count >= num\_elements)  {  fout << endl;  return;  }  if ((T\_type == string("int")) || (T\_type == string("double")) ||  (T\_type == string("class std::basic\_string<char, struct std::char\_traits<char>, class std::allocator<char> > ")))  fout << setw(10) << t\_GA[count];  else  fout << t\_GA[count] << " ";  count++;  } fout << endl;  } fout << endl;  }  template<typename K>  bool TA\_Entry<K>::isValidIndex(int i) {  if ((i < 0) || (i > num\_elements))  return false;  else  return true;  }  #endif // !TA\_E\_H |
| /\* CompleteBinaryTree.h \*/  #ifndef CBT\_H  #define CBT\_H  #include "TA\_Entry.h"  #define CBT\_ROOT 1  template<typename K>  class CompleteBinaryTree : public TA\_Entry<K> {  public:  CompleteBinaryTree(int capa, string nm);  int add\_at\_end(K& elem);  K& getEndElement() { return this->t\_GA[end]; }  K& getRootElement() { return this->t\_GA[CBT\_ROOT]; }  int getEndIndex() { return end; }  void removeCBTEnd();  void fprintCBT(ofstream& fout);  void fprintCBT\_byLevel(ofstream& fout);  protected:  void \_fprintCBT\_byLevel(ofstream& fout, int p, int level);  int parentIndex(int index) { return index / 2; }  int leftChildIndex(int index) { return index \* 2; }  int rightChildIndex(int index) { return index \* 2 + 1; }  bool hasLeftChild(int index) { return (index \* 2) <= end; }  bool hasRightChild(int index) { return (index \* 2 + 1) <= end; }  int end;  };  template<typename K>  CompleteBinaryTree<K>::CompleteBinaryTree(int capa, string nm)  :TA\_Entry<K>(capa, nm)  {  end = 0; // reset to empty  }  template<typename K>  void CompleteBinaryTree<K>::fprintCBT(ofstream& fout)  {  if (end <= 0)  {  fout << this->getName() << " is empty now !!" << endl;  return;  }  int count = 0;  for (int i = 1; i <= end; i++)  {  fout << setw(3) << this->t\_GA[i] << endl;  //if ((((count + 1) % 10) == 0) && (i != end))  //fout << endl;  count++;  }  }  template<typename K>  void CompleteBinaryTree<K>::\_fprintCBT\_byLevel(ofstream& fout, int index, int level)  {  int index\_child;  if (this->hasRightChild(index))  {  index\_child = this->rightChildIndex(index);  this->\_fprintCBT\_byLevel(fout, index\_child, level + 1);  }  for (int i = 0; i < level; i++)  fout << " ";  // this->t\_GA[index].fprint(fout);  fout << this->t\_GA[index];  fout << endl;  if (this->hasLeftChild(index))  {  index\_child = this->leftChildIndex(index);  this->\_fprintCBT\_byLevel(fout, index\_child, level + 1);  }  }  template<typename K>  void CompleteBinaryTree<K>::fprintCBT\_byLevel(ofstream& fout)  {  if (end <= 0)  {  fout << "CBT is EMPTY now !!" << endl;  return;  }  \_fprintCBT\_byLevel(fout, CBT\_ROOT, 0);  }  template<typename K>  int CompleteBinaryTree<K>::add\_at\_end(K& elem)  {  if (end >= this->capacity)  {  cout << this->getName() << " is FULL now !!" << endl;  return end;  }  end++;  this->t\_GA[end] = elem;  return end;  }  template<typename K>  void CompleteBinaryTree<K>::removeCBTEnd()  {  end--;  this->num\_elements--;  }  #endif // !CBT\_H |
| /\* HeapPrioQ.h \*/  #ifndef HPQ\_H  #define HPQ\_H  #include "CompleteBinaryTree.h"  template<typename K>  class HeapPrioQueue : public CompleteBinaryTree<K> {  public:  HeapPrioQueue(int capa, string nm);  ~HeapPrioQueue();  bool isEmpty() { return size() == 0; }  bool isFull() { return size() == this->capacity; }  int insert(K& elem);  K\* removeHeapMin();  K\* getHeapMin();  void fprint(ofstream& fout);  int size() { return this->end; }  private:  };  template<typename K>  HeapPrioQueue<K>::HeapPrioQueue(int capa, string nm)  :CompleteBinaryTree<K>(capa, nm)  { }  template<typename K>  HeapPrioQueue<K>::~HeapPrioQueue()  { }  template<typename K>  void HeapPrioQueue<K>::fprint(ofstream& fout)  {  if (size() <= 0)  {  fout << "HeapPriorityQueue is Empty !!" << endl;  return;  }  else  CompleteBinaryTree::printCBT(fout);  }  template<typename K>  int HeapPrioQueue<K>::insert(K& elem)  {  int index, parent\_index;  K temp;  if (isFull())  {  cout << this->getName() << " is Full !!" << endl;  return size();  }  index = this->add\_at\_end(elem);  /\* up-heap bubbling \*/  while (index != CBT\_ROOT)  {  parent\_index = this->parentIndex(index);  if (this->t\_GA[index] > this->t\_GA[parent\_index])  break;  else  {  temp = this->t\_GA[index];  this->t\_GA[index] = this->t\_GA[parent\_index];  this->t\_GA[parent\_index] = temp;  index = parent\_index;  }  }  return size();  }  template<typename K>  K\* HeapPrioQueue<K>::getHeapMin()  {  K\* pMinElem;  if (size() <= 0)  {  return NULL;  }  pMinElem = (K\*) new K;  \*pMinElem = this->getRootElement();  return pMinElem;  }  template<typename K>  K\* HeapPrioQueue<K>::removeHeapMin()  {  int index\_p, index\_c, index\_rc;  K\* pMinElem;  K temp, t\_p, t\_c;  int HPQ\_size = size();  if (HPQ\_size <= 0)  {  return NULL;  }  pMinElem = (K\*) new K;  \*pMinElem = this->getRootElement();  if (HPQ\_size == 1)  this->removeCBTEnd();  else {  index\_p = CBT\_ROOT;  this->t\_GA[CBT\_ROOT] = this->t\_GA[this->end];  this->end--;  /\* down-heap bubbling \*/  while (this->hasLeftChild(index\_p))  {  index\_c = this->leftChildIndex(index\_p);  index\_rc = this->rightChildIndex(index\_p);  if (this->hasRightChild(index\_p) && (this->t\_GA[index\_c] > this->t\_GA[index\_rc]))  index\_c = index\_rc;  t\_p = this->t\_GA[index\_p];  t\_c = this->t\_GA[index\_c];  if (t\_p > t\_c)  {  //swap(index\_u, index\_c);  temp = this->t\_GA[index\_p];  this->t\_GA[index\_p] = this->t\_GA[index\_c];  this->t\_GA[index\_c] = temp;  index\_p = index\_c;  }  else  break;  } // end while  }  return pMinElem;  }  #endif // !HPQ\_H |
|  |