객체지향프로그래밍과 자료구조

Ch 8. 템플릿 클래스
T_Array 응용 자료 구조

정보통신공학과
교수 김 영 탁

(Tel : +82-53-810-2497; E-mail : ytkim@yu.ac.kr)

Outline

- Template Class T_Array<T>
- Class T_Array<T> 기반 자료구조
  - LIFO 스택 (stack)
  - FIFO 큐, 환형 큐 (Circular Queue)
  - Deque (Double Ended Queue)
- 탐색 키를 포함하는 템플릿 class T_Entry<K, V>
- 완전이진트리 (Complete Binary Tree),
  class CompleBinaryTree<K, V>
- class CompleBinaryTree<K, V> 기반
  힙 우선 순위 큐 (Heap Priority Queue),
  class HeapPrioQ<K, V>
- Priority Queue의 응용 예: 우선순위 기반 Event 처리

Template Class T_Array<T>

Template class T_Array<T>

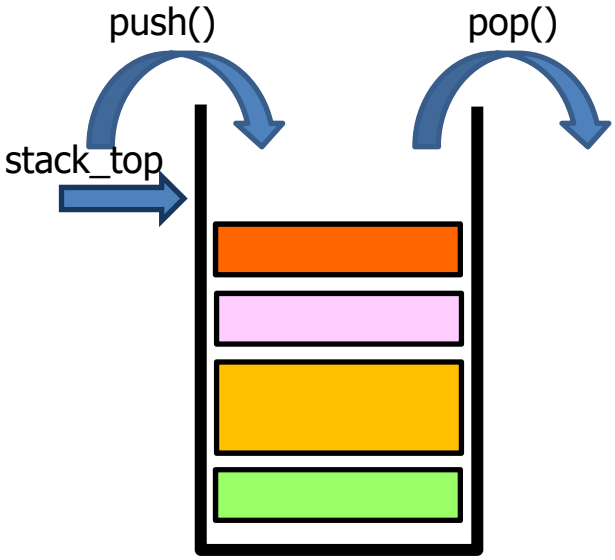
```
/* Template class T_Array.h (1) */
#ifndef T_Array_H
#define T_Array_H
#include <iostream>
#include <iomanip>
using namespace std;
enum SortingOrder { INCREASING, DECREASING };

template<typename T>
class T_Array
{
public:
    T_Array(int n, string nm); // constructor
    ~T_Array(); // destructor
    int size() { return num_elements; }
    bool empty() { return num_elements == 0; }
    string getName() { return name; }
    void reserve(int new_capacity);
    void insert(int i, T element);
    void insertBack(T element); // insert the new element at the back(end)
    void remove(int i);
```

```
/* Template class T_Array.h (2) */

    T& at(int i);
    void set(int i, T& element);
    T& getMin(int begin, int end);
    T& getMax(int begin, int end);
    void shuffle();
    int sequential_search(T search_key); // search and return the index; -1 if not found
    int binary_search(T search_key); // search and return the index; -1 if not found
    void selection_sort(SortingOrder sortOrder);
    void quick_sort(SortingOrder sortOrder);
    void merge_sort(SortingOrder sortOrder);
    void fprint(ofstream &fout, int elements_per_line);
    void fprintSample(ofstream &fout, int elements_per_line, int num_sample_lines);
    bool isValidIndex(int i);
    T& operator[](int index);
private:
    T *t_array;
    int num_elements;
    int capacity;
    string name;
};
```

Template Class TA_Stack<T>

First In Last Out (FILO) Stack

- The Stack stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) or first-in last –out (FILO) scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - **push(T elm)**: inserts an element
  - **T pop()**: removes the last inserted element
- Auxiliary stack operations:
  - **T top()**: returns the last inserted element without removing it
  - **int size()**: returns the number of elements stored
  - **bool empty()**: indicates whether no elements are stored



Applications of Stacks

Direct applications

- Page-visited history in a Web browser
  (e.g., keeping recently visited web site URLs)
- Undo sequence in a text editor
- Chain of method calls in the C++ run-time system

Indirect applications

- Auxiliary data structure for algorithms
- Stack is used as a component of other data structures

class TA_Stack<T>

```
template<typename T>
class TA_Stack : public T_Array<T>
{
public:
    TA_Stack(int capacity, string nm); // constructor
    ~TA_Stack() {} // destructor
    T* top(); // return the element at top of stack
    T* pop(); // pop the data block at top of the stack
    int push(const T& element); // push into the stack
    bool isEmpty();
    bool isFull();
    int size();
    void fprint(ostream& fout, int elements_per_line);
private:
    int stack_top; // index to stack_top
};
```

```
template<typename T>
int TA_Stack<T>::push(const T& element)
   // push into the stack
{
    if (isFull())
    {
       cout << "Stack is Full !₩n";
       return stack_top;
    }

    t_array[stack_top] = element;
    stack_top++;
    num_elements++;
    return stack_top;
}

template<typename T>
T* TA_Stack<T>::top()
   // return the pointer to the top of the stack
{
    if (isEmpty())
        return NULL;
    else
    {
        return &t_array[stack_top-1];
    }
}
```

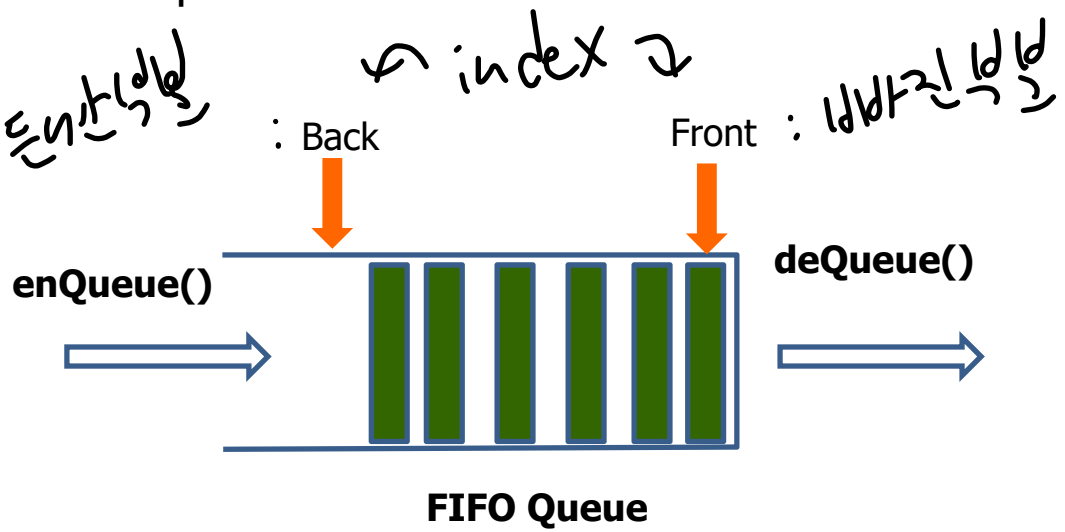
```
template<typename T>
T* TA_Stack<T>::pop()
   // pop the data block at top of the stack
{
    if (isEmpty())
        return NULL;
    else
    {
        T* pE;

        stack_top--;
        pE = &t_array[stack_top];
        num_elements--;
        return pE;
    }
}
```

Template Class TA_Queue<T>

First In First Out (FIFO) Queues

◆ The Queue stores arbitrary objects

- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the back(rear) of the queue and removals are at the front of the queue



Queue Operations

Main queue operations:

- enqueue(object): inserts an element at the end of the queue
- dequeue(): removes the element at the front of the queue

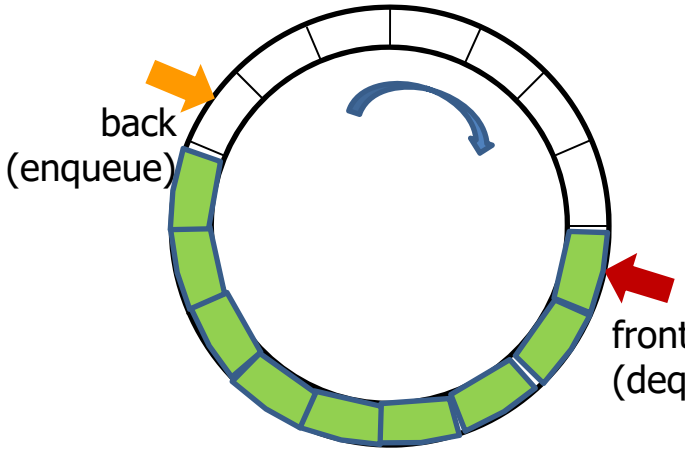
Auxiliary queue operations:

- object front(): returns the element at the front without removing it
- integer size(): returns the number of elements stored
- boolean empty(): indicates whether no elements are stored

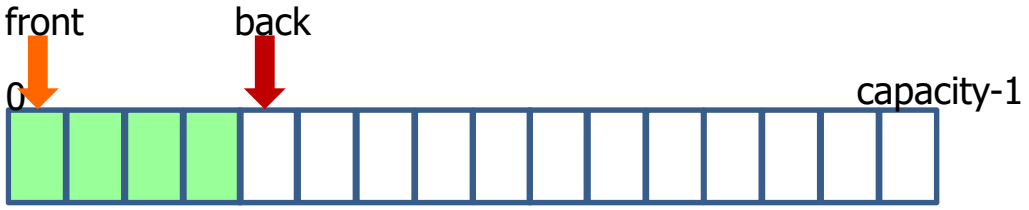
Exceptions

- Attempting the execution of dequeue or front on an empty queue throws an QueueEmpty

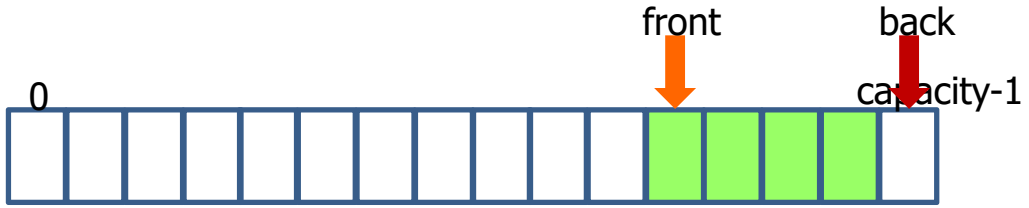
Implementation of Queue with Circular Buffer



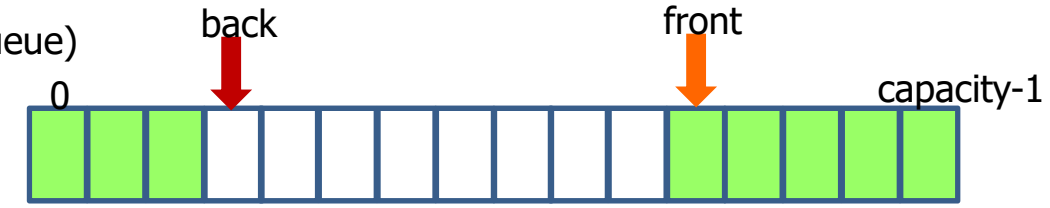
Operations in Circular Buffer



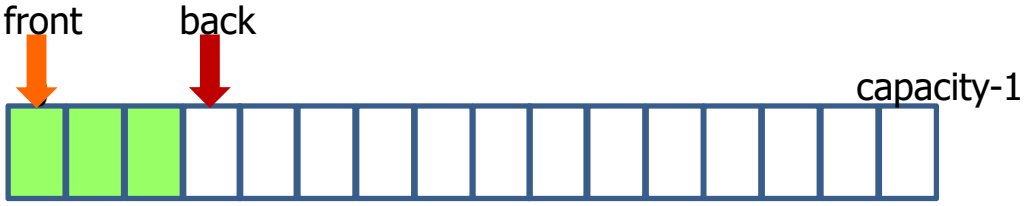
(a) Status of circular buffer after 4 enqueues after initialization



(b) Current status of circular buffer



(c) Circular buffer after 4 enqueues



(d) Circular buffer after 5 dequeues

class TA_Queue<T>

```
/* TA_Queue.h (1) */
#ifndef TA_QUEUE_H
#define TA_QUEUE_H
#include "T_Array.h"
template<typename T>
class TA_Queue : public T_Array<T>
{
private:
    int front; // index of queue_front
    int back;  // index of queue_back
public:
    class CirItor;
    TA_Queue(int capacity, string nm); // constructor
    ~TA_Queue() {} //destructor
    T* dequeue(); // return the element at front of queue
    T* enqueue(const T& element); // insert an element at the back of queue
    bool isEmpty();
    bool isFull();
    int size();
    void print(ostream& fout, int elements_per_line);
    CirItor begin() { return CirItor(&t_array[front], &t_array[0], &t_array[capacity - 1]); }
    CirItor end() { return CirItor(&t_array[back], &t_array[0], &t_array[capacity - 1]); }
```

```
//class TA_Queue : public T_Array<T>
public:
    class CirItor  // Circular Queue Iterator
    {
    private:
        T* pE;
        T* pBegin;
        T* pEnd;
    public:
        CirItor() {} // default constructor
        CirItor(T* p, T* b, T* e)
            : pE(p), pBegin(b), pEnd(e) {}
        T& operator*() { return *pE; }
        CirItor operator++()
        {
            if (pE == pEnd)
                pE = pBegin;
            else
                ++pE;
            return (*this);
        }
        CirItor operator--()
        {
            if (pE == pBegin)
                pE = pEnd;
            else
                --pE;
            return (*this);
        }

        bool operator==(const CirItor& p)
          { return (pE == p.pE); }

        bool operator!=(const CirItor& p)
          { return (pE != p.pE); }

    }; // end class CirItor of class TA_Deque

}; end of class TA_Deque
```

```
/* TA_Queue.h (2) */

template<typename T>
TA_Queue<T>::TA_Queue(int cap, string nm)
    :T_Array(cap, nm)
   // initialization section in constructor
{
    if (t_array == NULL)
    {
       cout << "Fail to create T_Array for "
            << nm << endl;
       exit;
    }
    front = back = 0;
    num_elements = 0;
}

template<typename T>
T* TA_Queue<T>::dequeue()
// dequeue the data at front of the queue
{
    if (isEmpty())
       return NULL;
    else
    {
       T* pE;

       pE = &t_array[front];
       front++;
       num_elements--;
       if (front >= capacity)
           front = front % capacity;
       return pE;
    }
}
```

```
/* TA_Queue.h (3) */

template<typename T>
T* TA_Queue<T>::enqueue(const T& element)
// push into the stack
{
   if (isFull())
   {
       cout << "Queue is Full !\n";
       return NULL;
   }

   T* pE;
   t_array[back] = element;
   pE = &t_array[back];
   back++;
   if (back >= capacity)
       back = back % capacity;
   num_elements++;
   return pE;
}
```

```
/* TA_Queue.h (4) */

template<typename T>
bool TA_Queue<T>::isEmpty()
{
   if (num_elements <= 0)
        return true;
   else
        return false;
}

template<typename T>
bool TA_Queue<T>::isFull()
{
   if (num_elements >= capacity)
      return true;
   else
      return false;
}
```

```
/* TA_Queue.h (5) */

template<typename T>
void TA_Queue<T>::print(ostream& fout,
   int elements_per_line)
{
    int count = 0;
    int index;
    if (num_elements <= 0)
    {
        fout << endl << this->getName() << " is
        Empty now !" << endl;
    }

    CirItor p;  // circular Iterator
    for (p = begin(); p != end(); ++p)
    {
        count++;
        fout << setw(5) << *p;
        if ((count % elements_per_line) == 0)
            fout << endl << "          ";
    }
}
#endif
```

```
/* main_T_Array_Queue.cpp  (1) */
#include <iostream>
#include <fstream>
#include <string>
#include <random>
#include "T_Array.h"
#include "TA_Queue.h"
using namespace std;

#define ELEMENTS_PER_LINE 10
#define SAMPLE_LINES 5
#define NUM_ELEMENTS 20
#define NUM_ELEMENTS_PER_ROUND 7
#define QUEUE_SIZE 10

void main()
{
    ofstream fout;
    int *pE;
    int data = 0;

    fout.open("output.txt");
    if (fout.fail())
    {
       cout << "Fail to open output.txt file for results !!" << endl;
       exit;
    }
```

```
/* main_T_Array_Queue.cpp  (2) */

    TA_Queue<int> TA_Queue_int(QUEUE_SIZE, string("TA_Queue of Integer"));
    for (int j = 0; j < 4; j++)
    {
        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "Enqueue (" << setw(3) << data << ") : ";
            TA_Queue_int.enqueue(data);
            TA_Queue_int.print(fout, 10);
            fout << endl;
            data++;
        }

        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "Dequeue (";
            pE = TA_Queue_int.dequeue();
            fout << setw(3) << *pE << ") : ";
            TA_Queue_int.print(fout, 10);
            fout << endl;
        }
    }
    fout << endl;
    fout.close();
}
```

◆ 실행 결과

```
Enqueue (  0) :     0
Enqueue (  1) :     0      1
Enqueue (  2) :     0      1      2
Enqueue (  3) :     0      1      2      3
Enqueue (  4) :     0      1      2      3      4
Enqueue (  5) :     0      1      2      3      4      5
Enqueue (  6) :     0      1      2      3      4      5      6
Dequeue (  0) :     1      2      3      4      5      6
Dequeue (  1) :     2      3      4      5      6
Dequeue (  2) :     3      4      5      6
Dequeue (  3) :     4      5      6
Dequeue (  4) :     5      6
Dequeue (  5) :     6
Dequeue (  6) :
TA_Queue of Integer is Empty now !

Enqueue (  7) :     7
Enqueue (  8) :     7      8
Enqueue (  9) :     7      8      9
Enqueue ( 10) :     7      8      9     10
Enqueue ( 11) :     7      8      9     10     11
Enqueue ( 12) :     7      8      9     10     11     12
Enqueue ( 13) :     7      8      9     10     11     12     13
Dequeue (  7) :     8      9     10     11     12     13
Dequeue (  8) :     9     10     11     12     13
Dequeue (  9) :    10     11     12     13
Dequeue ( 10) :    11     12     13
Dequeue ( 11) :    12     13
Dequeue ( 12) :    13
Dequeue ( 13) :
TA_Queue of Integer is Empty now !
```

Template Class TA_Deque<T>

class TA_Deque<T>

```
/* TA_Deque.h (1) */
#ifndef TA_DEQUE_H
#define TA_DEQU_H
#include "T_Array.h"
template<typename T>
class TA_Deque : public T_Array<T>
{
public:
    class CirItor;
    TA_Deque(int capacity, string nm); // constructor
    ~TA_Deque() {} //destructor
    T* push_front(T& element); // push the data block at the front of the Deque
    T* push_back(T& element); // push the data block at the back of the Deque
    T* pop_front(); // pop the data block at front of the Deque
    T* pop_back(); // pop the data block at front of the Deque
    bool isEmpty();
    bool isFull();
    int size();
    void fprint(ostream& fout, int elements_per_line);
private:
    int front; // index of deque_front
    int back;  // index of deque_end
```

```
public:
    class CirItor
    {
    private:
        T* pE;
        T* pBegin;
        T* pEnd;
    public:
        CirItor() {}
        CirItor(T* p, T* b, T* e) : pE(p), pBegin(b), pEnd(e) {}
        T& operator*() { return *pE; }
        CirItor operator++()
        {
            if (pE == pEnd)
                pE = pBegin;
            else
                ++pE;
            return (*this);
        }
        CirItor operator--()
        {
            if (pE == pBegin)
                pE = pEnd;
            else
                --pE;
            return (*this);
        }
        bool operator==(const CirItor& p) { return (pE == p.pE); }
        bool operator!=(const CirItor& p){ return (pE != p.pE); }
    }; // end class CirItor
};
```

```
/* TA_Deque.h (2) */
template<typename T>
TA_Deque<T>::TA_Deque(int cap, string nm)
:T_Array(cap, nm) // constructor
{
   if (t_array == NULL)
   {
      cout << "Fail to create T_Array for "
           << nm << endl;
      exit;
   }
   front = back = 0;
   num_elements = 0;
}

template<typename T>
bool TA_Deque<T>::isEmpty()
{
   if (num_elements <= 0)
      return true;
   else
      return false;
}

template<typename T>
bool TA_Deque<T>::isFull()
{
   if (num_elements >= capacity)
      return true;
   else
      return false;
}
```

```
/* TA_Deque.h (3) */
template<typename T>
T* TA_Deque<T>::push_front(T& element)
  // push at the front of Deque
{
     if (isFull())
     {
          cout << “Deque is Full !\n";
          return NULL;
     }
     front--;
     if (front < 0)
        front = capacity - 1;
     t_array[front] = element;
     T* pE = &t_array[front];
     num_elements++;
     return pE;
}

template<typename T>
T* TA_Deque<T>::push_back(T& element)
  // push at the back of Deque
{
   if (isFull())
   {
      cout << “Deque is Full !\n";
      return  NULL;
   }
   t_array[back] = element;
   T* pE = &t_array[back];
   back++;
   if (back >= capacity)
      back = back % capacity;
   num_elements++;
   return pE;
}
```

```
/* TA_Deque.h (4) */

template<typename T>
T* TA_Deque<T>::pop_front()
  // dequeue the data at front of the queue
{
   if (isEmpty())
   {
      return NULL;
   }
   else
   {
      T *pE = (T *) new T;

      *pE = t_array[front];
      front++;
      num_elements--;
      if (front >= capacity)
         front = front % capacity;
         //for circular queue operation
      return pE;
   }
}
```

```
/* TA_Deque.h (5) */

template<typename T>
T* TA_Deque<T>::pop_back()
  // dequeue the data at front of the queue
{
   if (isEmpty())
   {
      return NULL;
   }
   else
   {
      T *pE = (T *) new T;

      back--;
      if (back < 0)
          back = capacity - 1;
          //for circular queue operation
      *pE = t_array[back];
      num_elements--;
      return pE;
   }
}
```

```
/* TA_Deque.h (6) */

template<typename T>
void TA_Deque<T>::fprint(ostream& fout, int elements_per_line)
{
    int count = 0;
    int index;
    if (num_elements <= 0)
    {
        fout << this->getName()
            << " is Empty now !" << endl;
    }

    CirItor p;  // circular Iterator
    for (p = begin(); p != end(); ++p)
    {
        count++;
        fout << setw(5) << *p;
        if ((count % elements_per_line) == 0)
              fout << endl << "          ";
    }
}
#endif
```

```
/* main_T_Array_Deque.cpp (1) */
#include <iostream>
#include <fstream>
#include <string>
#include <random>
#include "T_Array.h"
#include "TA_Deque.h"

using namespace std;
#define ELEMENTS_PER_LINE 10
#define SAMPLE_LINES 5
#define NUM_ELEMENTS 20
#define NUM_ELEMENTS_PER_ROUND 7
#define DEQUE_SIZE 10

void main()
{
    ofstream fout;
    int *pE, elem;
    int data = 0;
    int* result;

    fout.open("output.txt");
    if (fout.fail())
    {
        cout << "Fail to open output.txt file for results !!" << endl;
        exit;
    }
```

```
/* main_T_Array_Deque.cpp (2) */

    TA_Deque<int> TA_Deque_int(DEQUE_SIZE, string("T_Array_Queue of Integer"));
    for (int j = 0; j < 4; j++)
    {
        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "TA_Deque::push_back (" << setw(3) << data << ") : ";
            result = TA_Deque_int.push_back(data);
            if (result == NULL)
                fout << "TA_Deque is FULL now !!" << endl;
            else
            {
                TA_Deque_int.fprint(fout, 10);
                fout << endl;
            }
            data++;
        }
        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "TA_Deque::pop_front (";
            pE = TA_Deque_int.pop_front();
            if (pE != NULL)
                fout << setw(3) << *pE << ") : ";
            else
                fout << "TA_Deque is Empty now !)" << endl;
            TA_Deque_int.fprint(fout, 10);
            fout << endl;
        }
```

```
/* main_T_Array_Deque.cpp (3) */

        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "TA_Deque::push_front(" << setw(3) << data << ") : ";
            result = TA_Deque_int.push_front(data);
            if (result == NULL)
                fout << "TA_Deque is FULL now !!" << endl;
            else
            {
                TA_Deque_int.fprint(fout, 10);
                fout << endl;
            }
            data++;
        }

        for (int i = 0; i < NUM_ELEMENTS_PER_ROUND; i++)
        {
            fout << "TA_Deque::pop_back  (";
            pE = TA_Deque_int.pop_back();
            if (pE != NULL)
                fout << setw(3) << *pE << ") : ";
            else
                fout << "TA_Deque is Empty now ! )" << endl;
            TA_Deque_int.fprint(fout, 10);
            fout << endl;
        }
    }
    fout << endl;
    fout.close();
}
```

◆ TA_Deque<T> 응용 프로그램의 실행 결과

```
TA_Deque::push_back (  0) :     0
TA_Deque::push_back (  1) :     0      1
TA_Deque::push_back (  2) :     0      1      2
TA_Deque::push_back (  3) :     0      1      2      3
TA_Deque::push_back (  4) :     0      1      2      3      4
TA_Deque::push_back (  5) :     0      1      2      3      4      5
TA_Deque::push_back (  6) :     0      1      2      3      4      5      6
TA_Deque::pop_front (  0) :     1      2      3      4      5      6
TA_Deque::pop_front (  1) :     2      3      4      5      6
TA_Deque::pop_front (  2) :     3      4      5      6
TA_Deque::pop_front (  3) :     4      5      6
TA_Deque::pop_front (  4) :     5      6
TA_Deque::pop_front (  5) :     6
TA_Deque::pop_front (  6) : T_Array_Queue of Integer is Empty now !

TA_Deque::push_front(  7) :     7
TA_Deque::push_front(  8) :     8      7
TA_Deque::push_front(  9) :     9      8      7
TA_Deque::push_front( 10) :    10      9      8      7
TA_Deque::push_front( 11) :    11     10      9      8      7
TA_Deque::push_front( 12) :    12     11     10      9      8      7
TA_Deque::push_front( 13) :    13     12     11     10      9      8      7
TA_Deque::pop_back  (  7) :    13     12     11     10      9      8
TA_Deque::pop_back  (  8) :    13     12     11     10      9
TA_Deque::pop_back  (  9) :    13     12     11     10
TA_Deque::pop_back  ( 10) :    13     12     11
TA_Deque::pop_back  ( 11) :    13     12
TA_Deque::pop_back  ( 12) :    13
TA_Deque::pop_back  ( 13) : T_Array_Queue of Integer is Empty now !
```

Template Class T_Entry<K, V>
Template Class TA_Entry<K, V>

탐색 키를 포함하는 class T_Entry<K, V>

```
/* T_Entry.h (1) */
#ifndef T_ENTRY_H
#define T_ENTRY_H
#include <fstream>

template<typename K, typename V>
class T_Entry
{
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& right) { return (_key > right.getKey()); }
    bool operator>=(const T_Entry& right) { return (_key >= right.getKey()); }
    bool operator<(const T_Entry& right) { return (_key < right.getKey()); }
    bool operator<=(const T_Entry& right) { return (_key <= right.getKey()); }
    bool operator==(const T_Entry& right)
        { return ((_key == right.getKey()) && (_value == right.getValue())); }
    T_Entry& operator=(const T_Entry& right);
    void fprint(ostream& fout);
private:
    K _key;
    V _value;
};
```

```
/* T_Entry.h (2) */

template<typename K, typename V>
T_Entry<K, V>&
T_Entry<K, V>::operator=(const T_Entry<K, V>& right)
{
    _key = right.getKey();
    _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
```

T_Entry의 일반화 배열
class TA_Entry<K, V>

```
template<typename K, typename V>
class TA_Entry
{
public:
    TA_Entry(int n, string nm); // constructor
    ~TA_Entry(); // destructor
    int size() { return num_elements; }
    bool empty() { return num_elements == 0; }
    string getName() { return name; }
    void reserve(int new_capacity);
    void insert(int i, T_Entry<K, V> element);
    void remove(int i);
    T_Entry<K, V>& at(int i);
    void set(int i, T_Entry<K, V>& element);
    void fprint(ofstream &fout, int elements_per_line);
    void fprintSample(ofstream &fout, int elements_per_line, int num_sample_lines);
    bool isValidIndex(int i);
    T_Entry<K, V>& operator[](int index);
protected:
    T_Entry<K, V> *t_array;
    int num_elements;
    int capacity;
    string name;
};
```

Heap(힙) /Priority Queue (우선순위큐), Complete Binary Tree (완전이진트리)

Priority Queue (우선 순위 큐)

- **A priority queue stores a collection of entries**
- **Typically, an entry is a pair (key, value), where the key indicates the priority**
- **Main methods of the Priority Queue ADT**
  - insert(e) : inserts an entry e
  - e = removeMin() : removes the entry with smallest key (highest priority)
- **Additional methods**
  - min() : returns, but does not remove, an entry with smallest key
  - size(), empty()
- **Applications:**
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue

◆ Functions supported in Priority Queue ADT

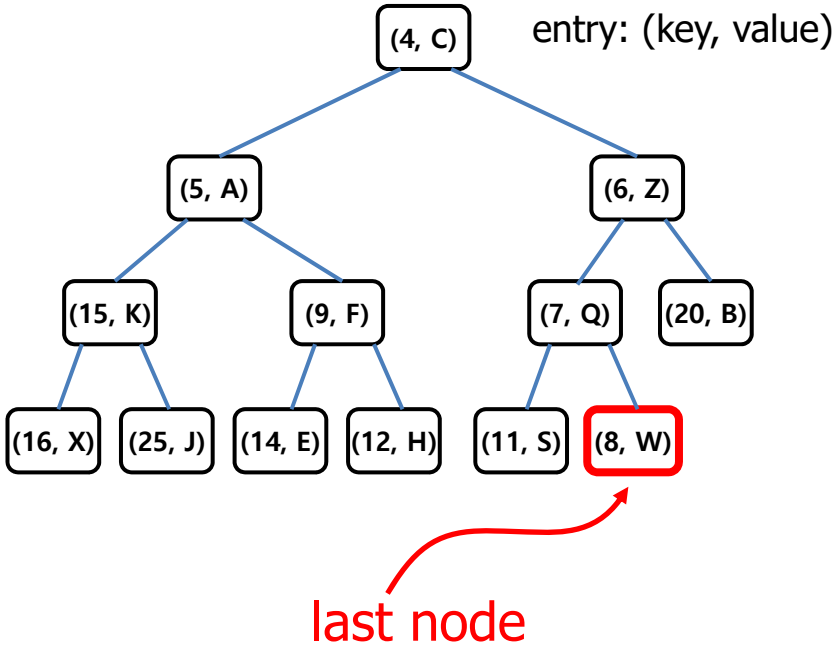
- size()
- empty()
- insert(e)
- min()
- removeMin()

Operation	Output	Priority Queue
insert(5)	-	{5}
insert(9)	-	{5, 9}
insert(2)	-	{2, 5, 9}
insert(7)	-	{2, 5, 7, 9}
min()	[2]	{2, 5, 7, 9}
removeMin()	-	{5, 7, 9}
size()	3	{5, 7, 9}
min()	[5]	{5, 7, 9}
removeMin()	-	{7, 9}
removeMin()	-	{9}
removeMin()	-	{}
empty()	true	{}
removeMin()	"error"	{}

Heap 내부 구조

- A heap is a complete binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every internal node v other than the root,
  $key(v) \geq key(parent(v))$
  $key(v) \leq key(child(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

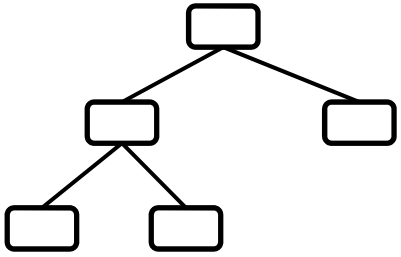
- The last node of a heap is the rightmost node of maximum depth



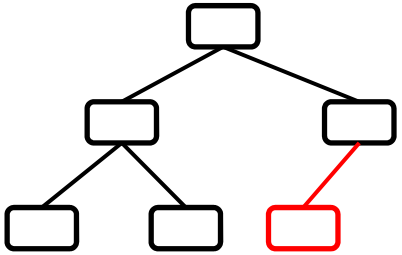
완전이진트리 (Complete Binary Tree)

◆ Complete Binary Tree 특성

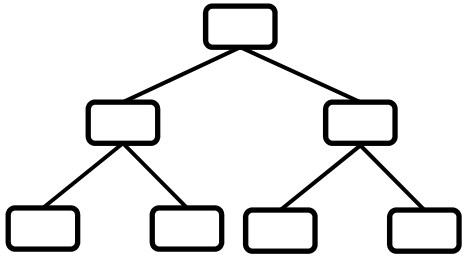
- a heap T with height h is a complete binary tree, that is, levels 0, 1, 2, …, h-1 of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \le i \le h-1$) and nodes at level h fill this level from left to right



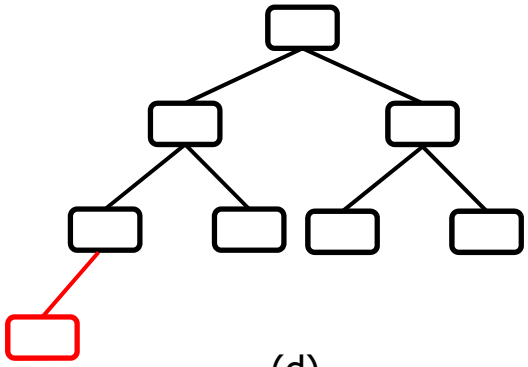
(a)



(b)



(c)



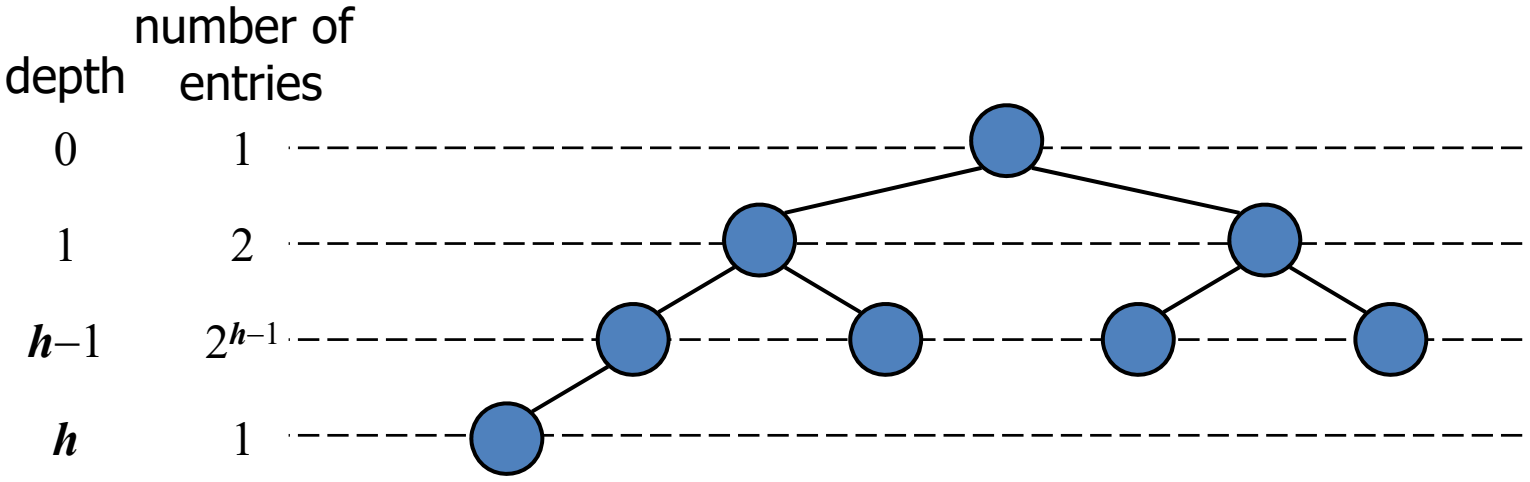
(d)

Height of a Heap

- **Theorem: A heap storing n keys has height $O(\log n)$**

  Proof: (we apply the complete binary tree property)

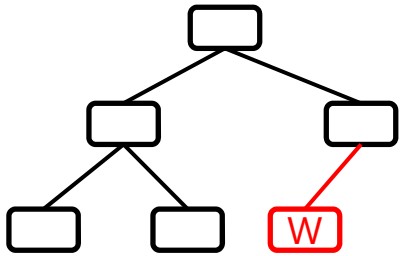
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log_2 n$



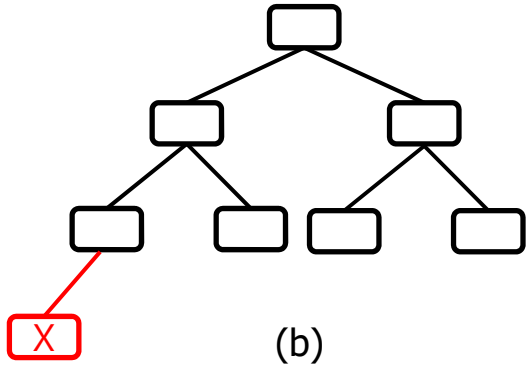
Array Representation of a Complete Binary Tree

◆ Array representation of a complete binary tree

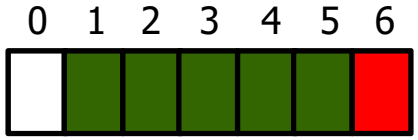
- if v is the root of CBT, then pos(v) = 1
- if lc is the left child of node u, then pos(lc) = 2 × pos(u)
- if rc is the right child of node u, then pos(rc) = 2 × pos(u) + 1



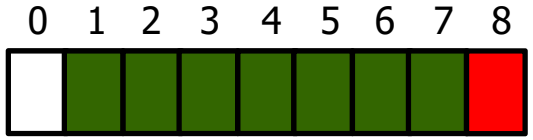
(a)



(b)



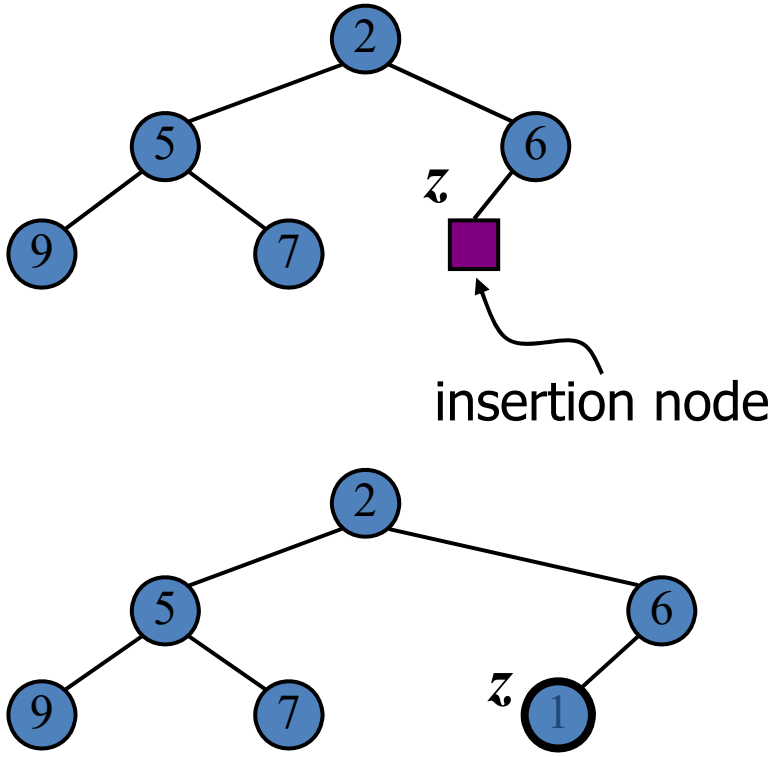
(c)



(d)

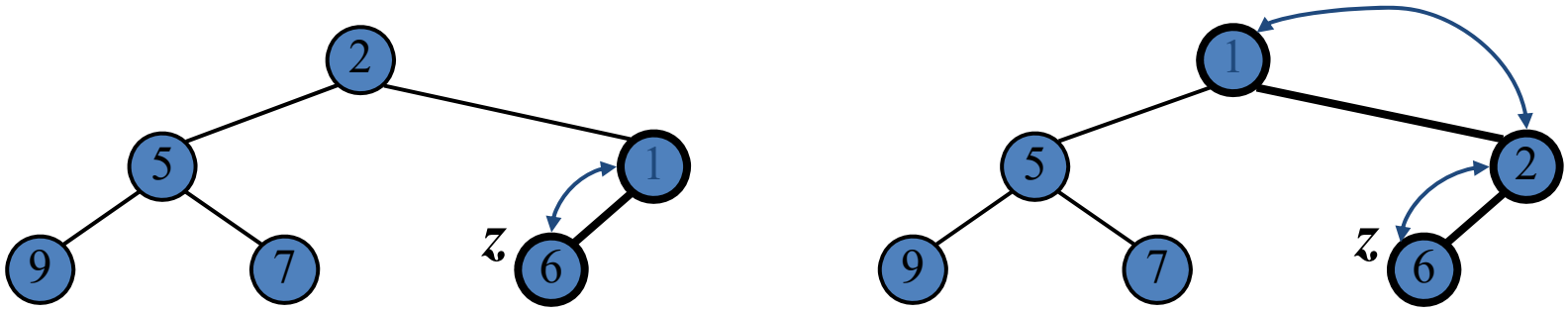
Insertion into a Heap

- Method insertItem() of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)

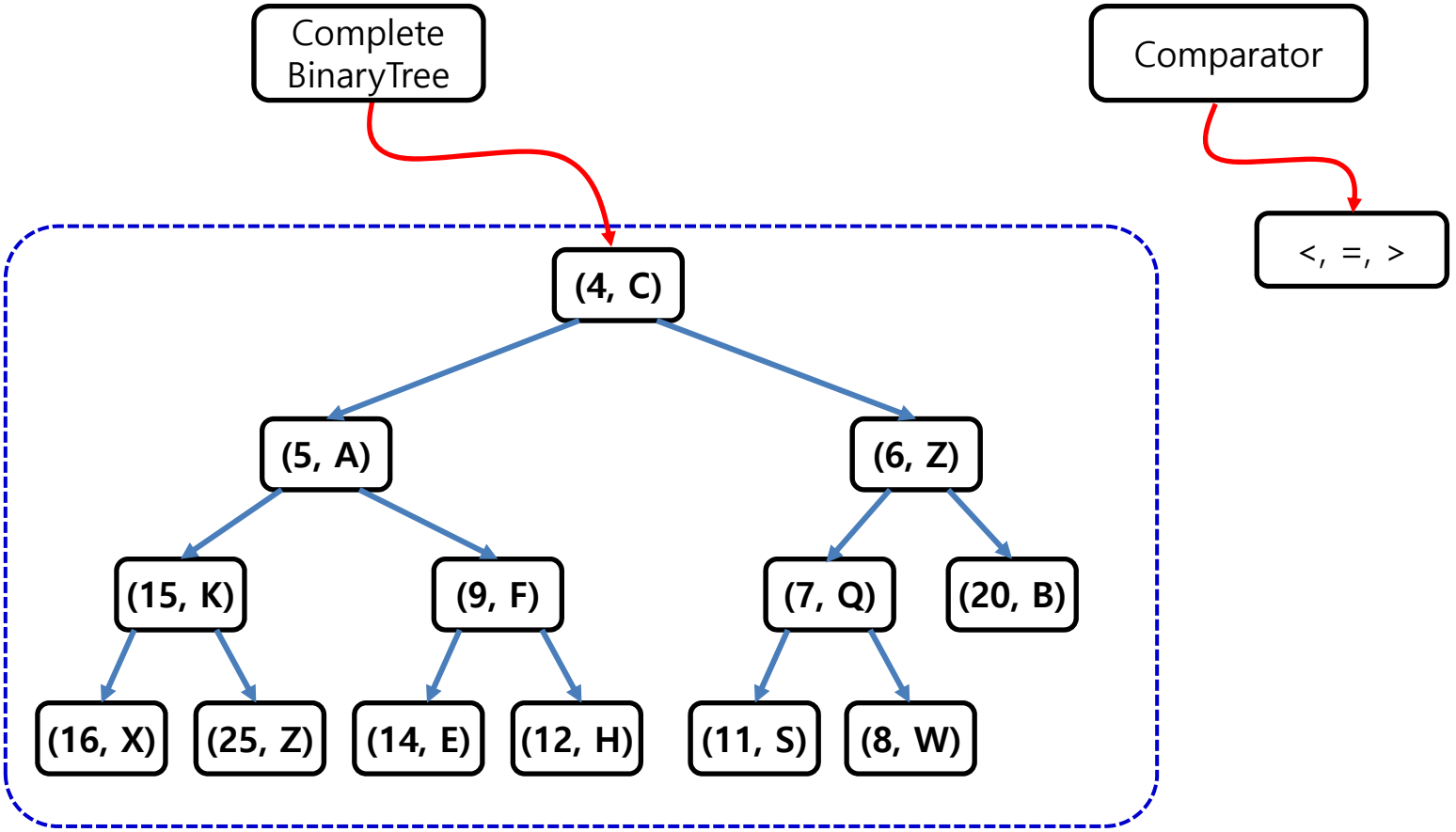


Up-heap Bubbling

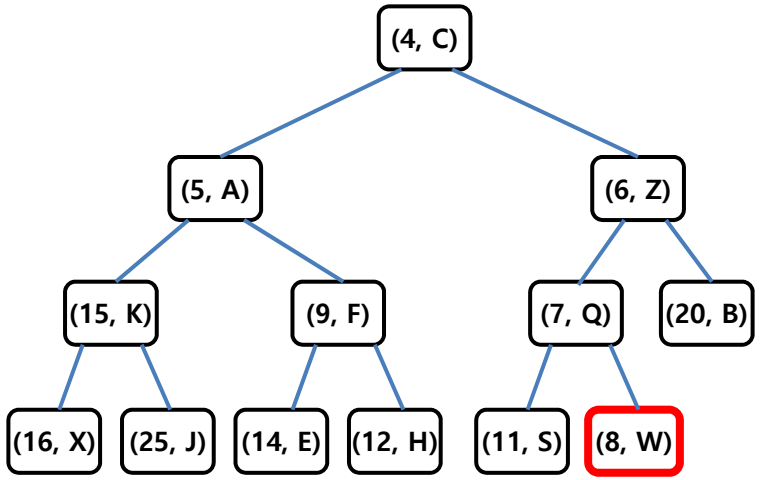
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log_2 n)$, upheap runs in $O(\log_2 n)$ time



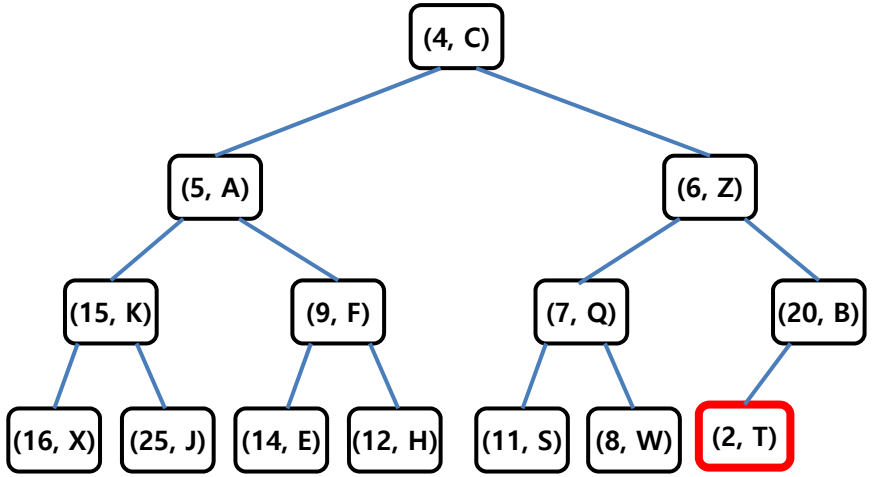
◆ Insertion with up-heap bubbling (1)



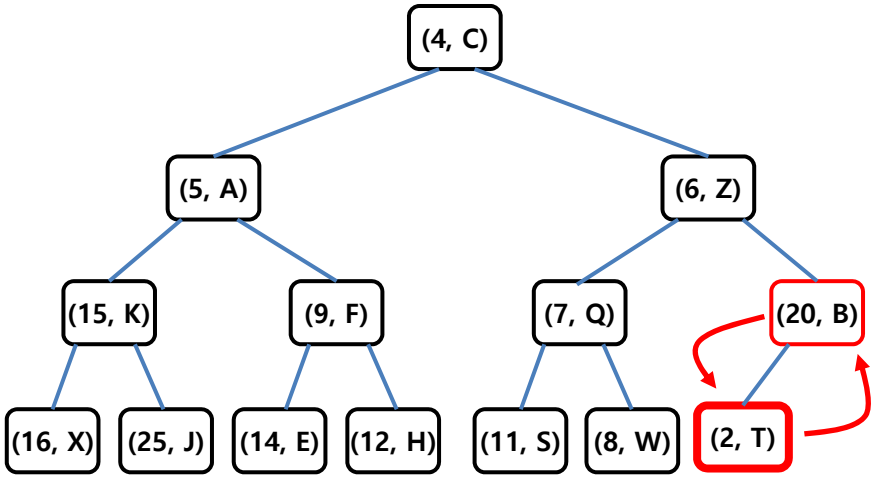
◆ Insertion with up-heap bubbling (2)



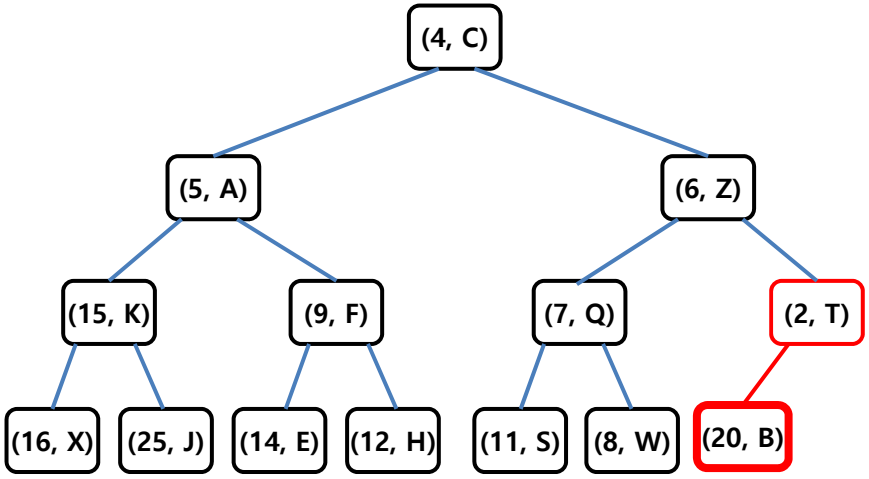
(a) last status



(b) insert a new node

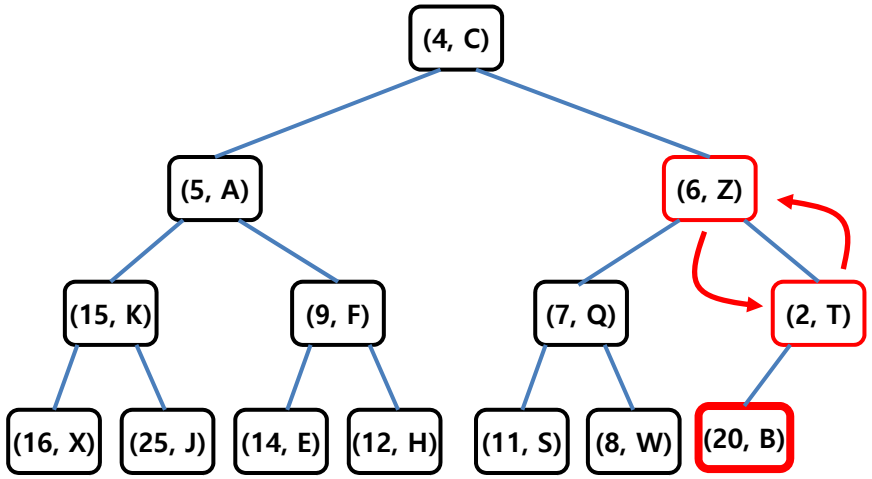


(c) swapping

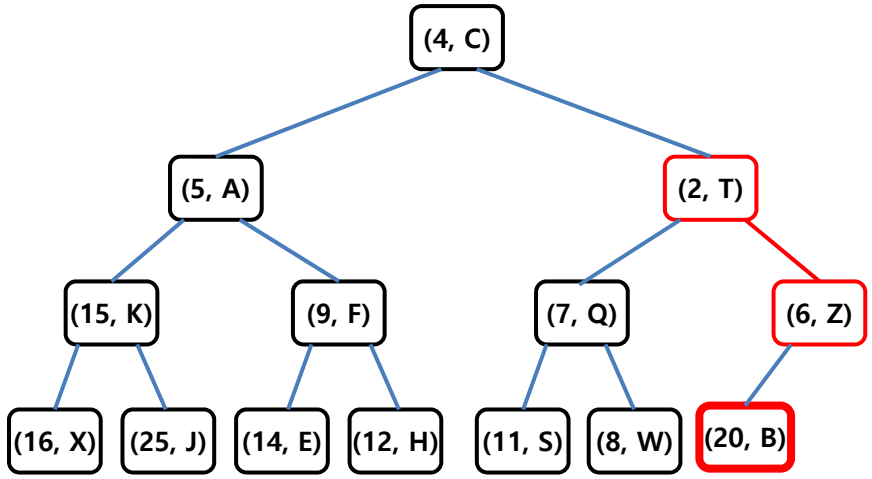


(d) after swapping

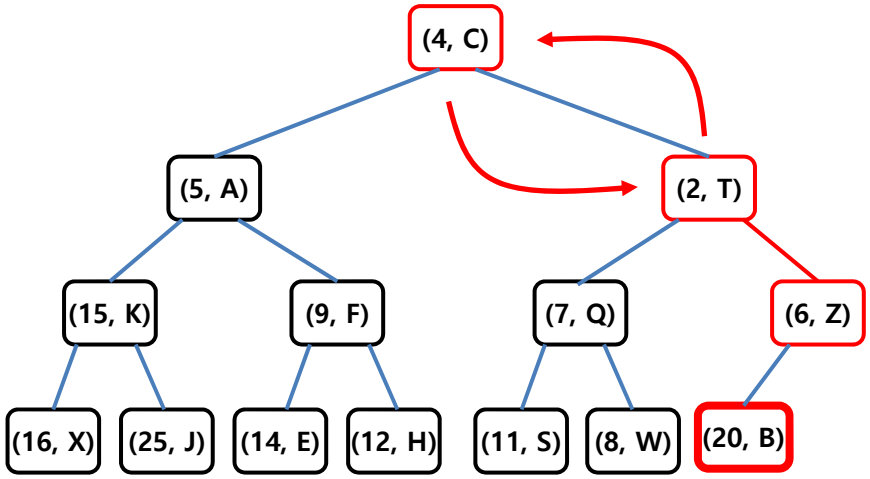
◆ Insertion with up-heap bubbling (3)



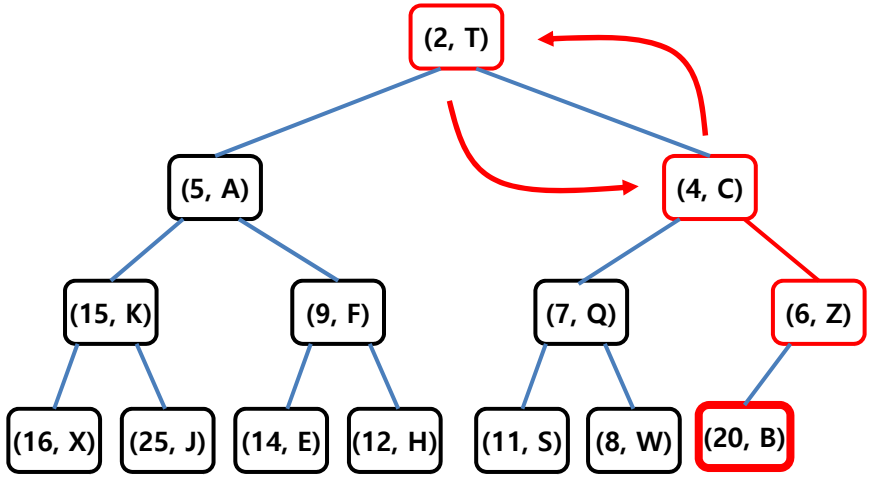
(e) swapping



(e) after swapping



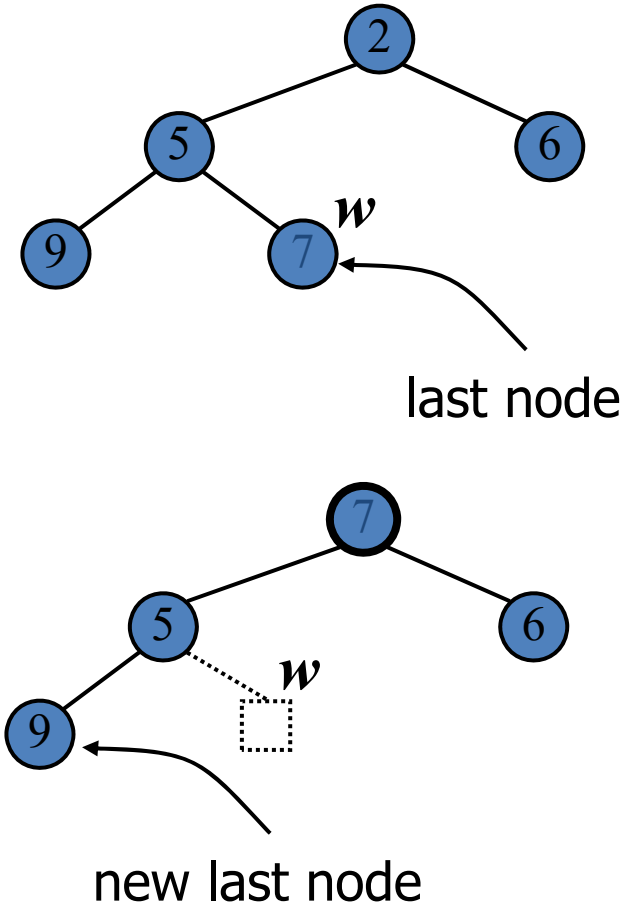
(e) swapping



(e) after swapping

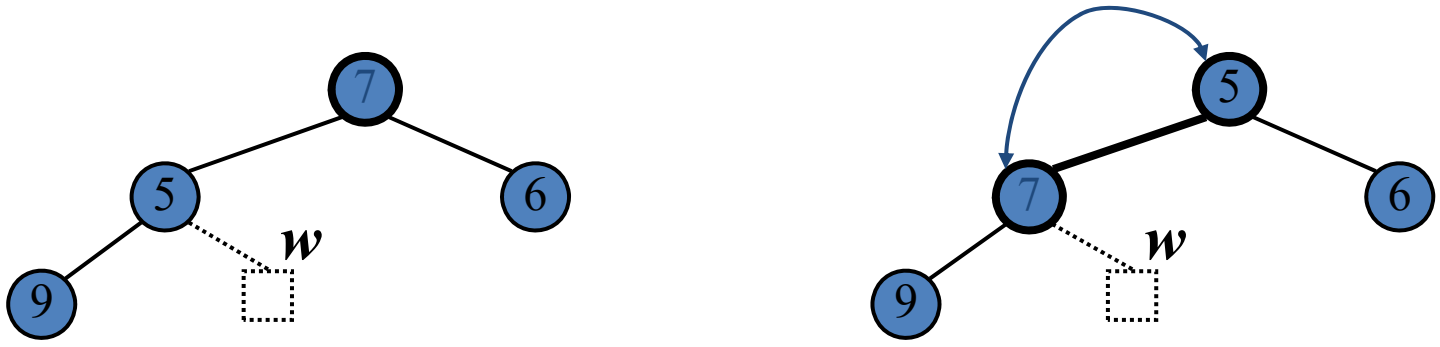
Removal from a Heap

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

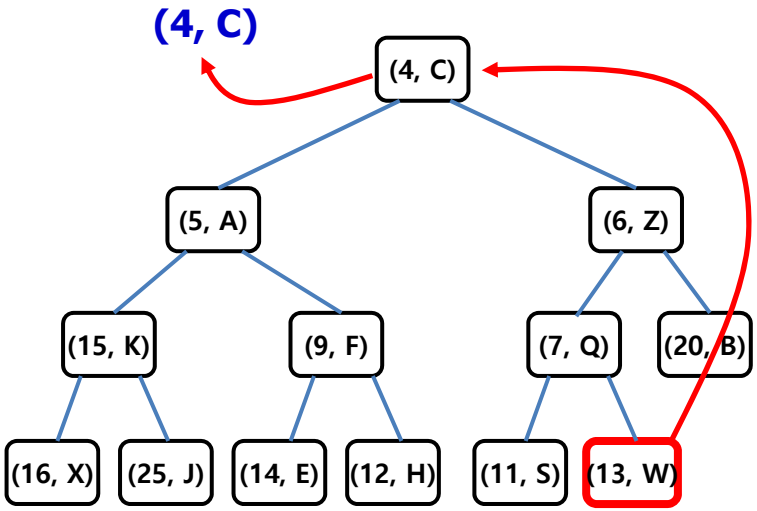


Down-heap Bubbling

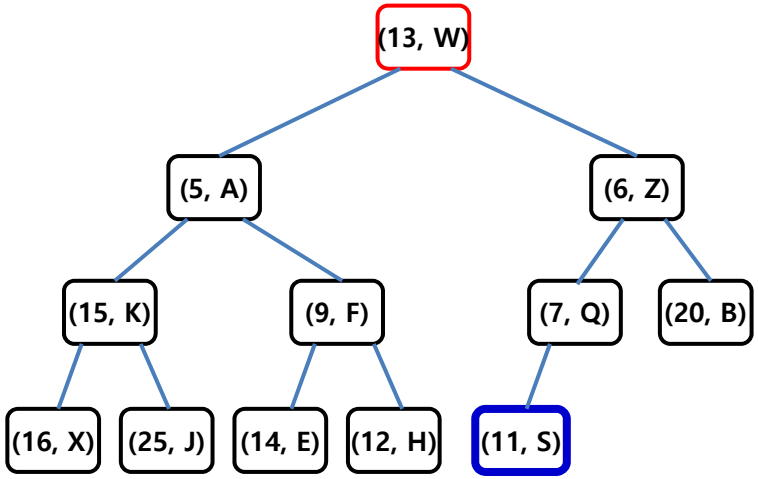
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm down-heap restores the heap-order property by swapping key k along a downward path from the root
- Down-heap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log_2 n)$, down-heap runs in $O(\log_2 n)$ time



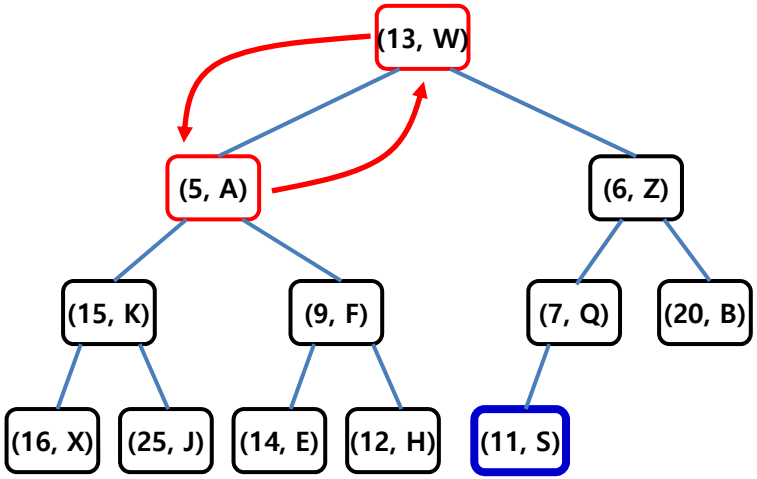
◆ Down-heap Bubbling after a RemoveMin() (1)



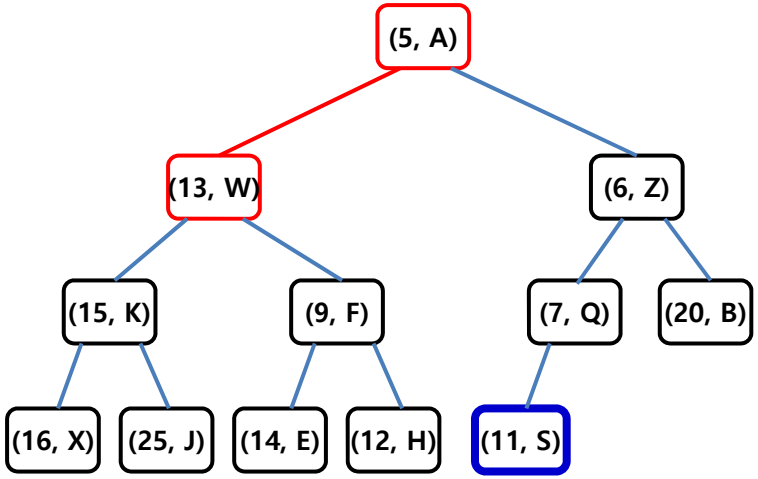
(a) last status



(b) after changing root by the last node

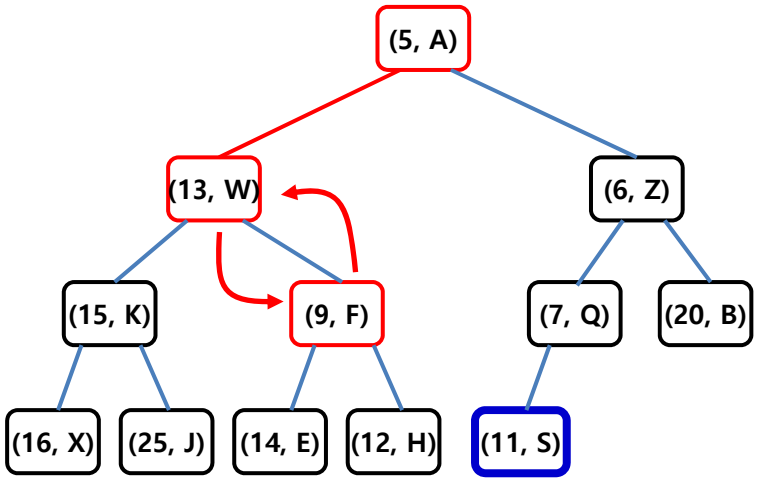


(c) compare and swap

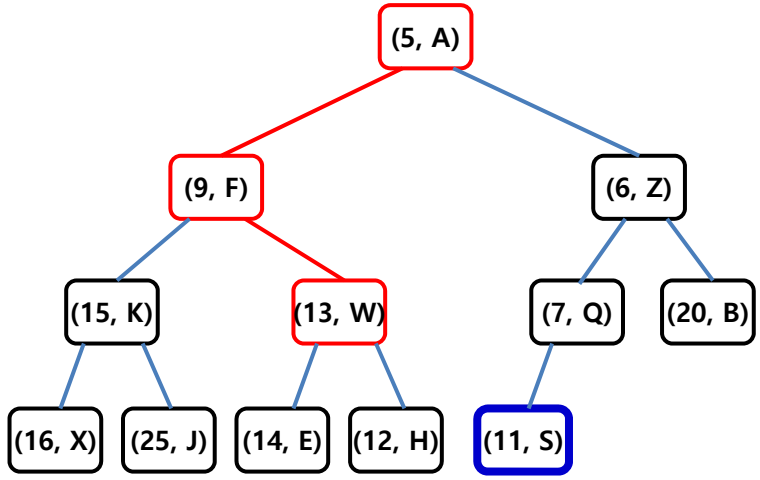


(d) after swapping

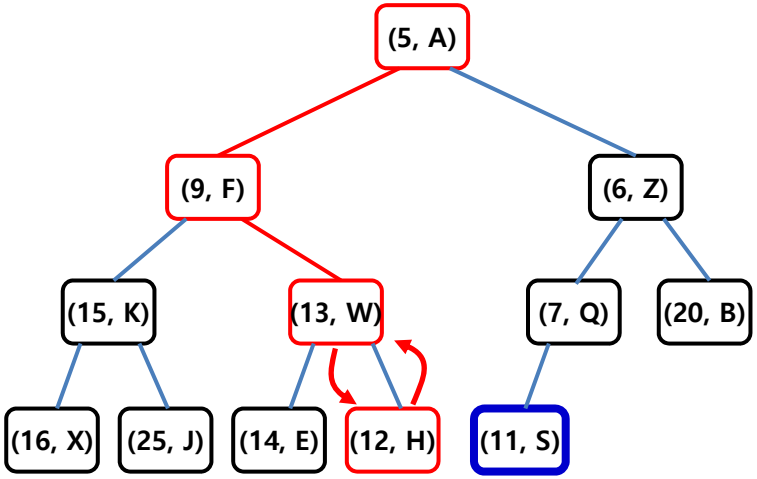
◆ Down-heap Bubbling after a RemoveMin() (2)



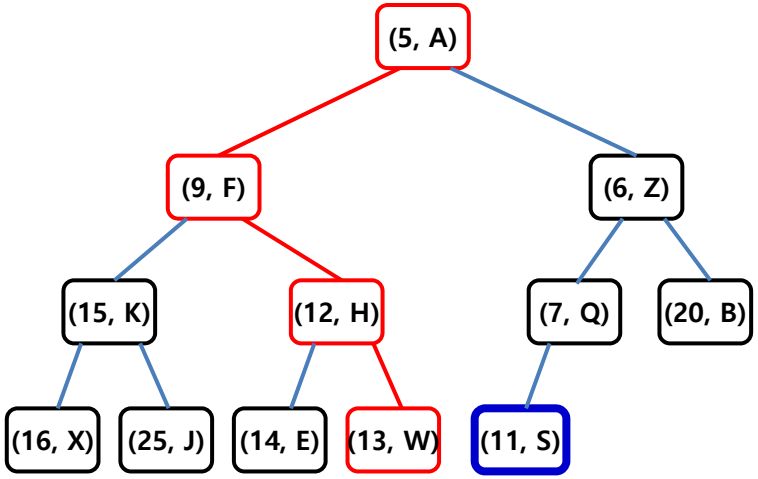
(e) compare and swap



(f) after swapping



(g) compare and swap



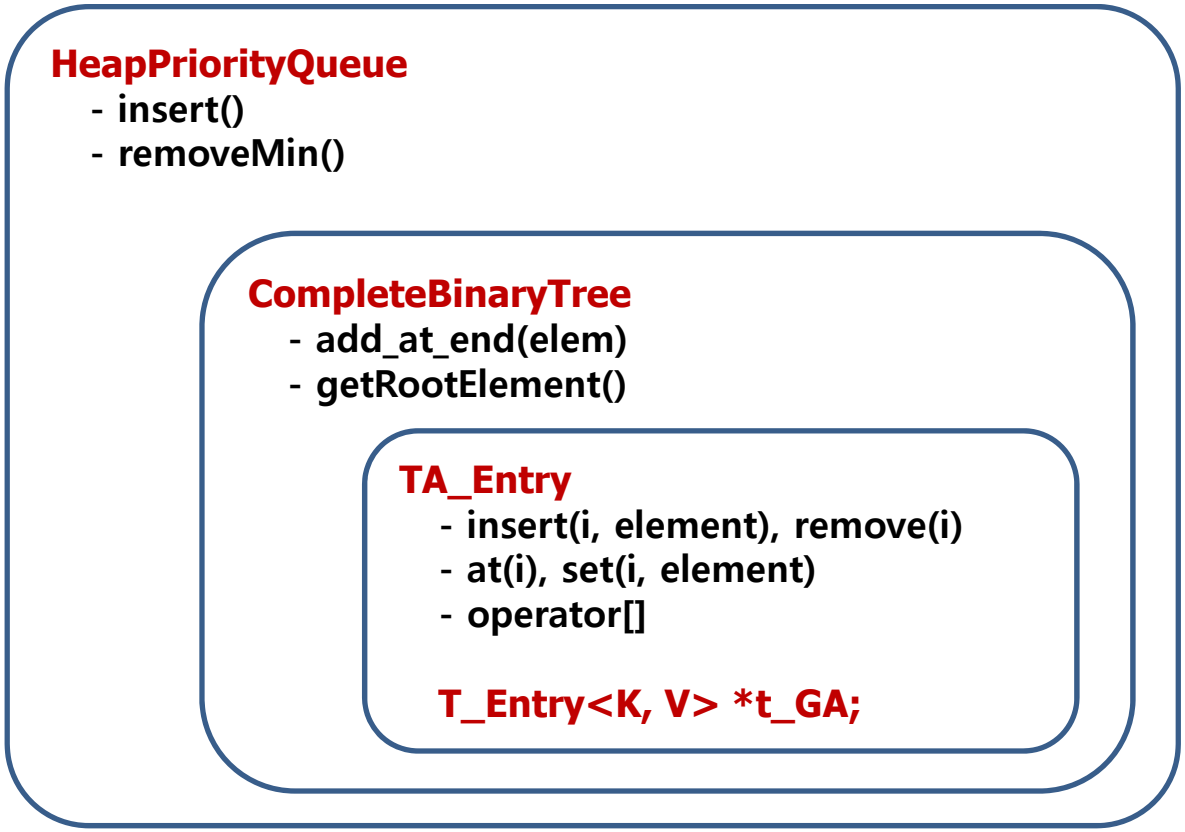
(h) after swapping

CompleteBinaryTree 기반
Heap Priority Queue 구현

CompleteBinaryTree 기반 Heap Priority Queue 구현

◆ Heap Priority Queue 구조

- class HeapPrioQueue는 class CompleteBinaryTree를 상속
- class CompleteBinaryTree는 class TA_Entry<K, V>를 상속



class CompleteBinaryTree<K, V>

```
/* CompleteBinaryTree.h (1) */
#ifndef COMPLETE_BINARY_TREE_H
#define COMPLETE_BINARY_TREE_H
#include "TA_Entry.h"
#include "T_Entry.h"
#define CBT_ROOT 1

template<typename K, typename V>
class CompleteBinaryTree : public TA_Entry<K, V>
{
public:
    CompleteBinaryTree(int capa, string nm);
    int add_at_end(T_Entry<K, V>& elem);
    T_Entry<K, V>& getEndElement() { return t_array[end]; }
    T_Entry<K, V>& getRootElement() { return t_array[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;
};
```

```
/* CompleteBinaryTree.h (2) */

template<typename K, typename V>
CompleteBinaryTree<K, V>::CompleteBinaryTree(int capa, string nm)
:TA_Entry<K, V>(capa+1, nm)
{
    end = 0; // reset to empty
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::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) << t_array[i] << endl;
        //if ((((count + 1) % 10) == 0) && (i != end))
        //fout << endl;
        count++;
    }
}
```

```
/* CompleteBinaryTree.h (3) */

template<typename K, typename V>
void CompleteBinaryTree<K, V>::_fprintCBT_byLevel(ofstream &fout, int index, int level)
{
    int index_child;
    if (hasRightChild(index))
    {
        index_child = rightChildIndex(index);
        _printCBT_byLevel(fout, index_child, level + 1);
    }

    for (int i = 0; i<level; i++)
       fout << "     ";
    t_array[index].fprint(fout);
    fout << endl;

    if (hasLeftChild(index))
    {
        index_child = leftChildIndex(index);
        _printCBT_byLevel(fout, index_child, level + 1);
    }
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::fprintCBT_byLevel(ofstream &fout)
{
    if (end <= 0)
    {
        fout << "CBT is EMPTY now !!" << endl;
        return;
    }
    _printCBT_byLevel(fout, CBT_ROOT, 0);
}
```

```
Final status of insertions :
                3
          2
                10
     1
                9
          4
                13
0
                7
          6
                12
     5
                11
          8
                14
```

```
/* CompleteBinaryTree.h (4) */

template<typename K, typename V>
int CompleteBinaryTree<K, V>::add_at_end(T_Entry<K, V>& elem)
{
    if (end >= capacity)
    {
       cout << this->getName() << " is FULL now !!" << endl;
       return end;
    }
    end++;
    t_array[end] = elem;

    return end;
}

template<typename K, typename V>
void CompleteBinaryTree<K, V>::removeCBTEnd()
{
    end--;
    num_elements--;
}
#endif
```

class HeapPrioQ<K, V>

```
/* HeapPrioQ.h (1) */

#ifndef HEAP_PRIO_QUEUE_H
#define HEAP_PRIO_QUEUE_H
#include "CompleteBinaryTree.h"

template<typename K, typename V>
class HeapPrioQueue : public CompleteBinaryTree<K, V>
{
public:
      HeapPrioQueue(int capa, string nm);
      ~HeapPrioQueue();
      bool isEmpty() { return size() == 0; }
      bool isFull() { return size() == capacity; }
      int insert(T_Entry<K, V>& elem);
      T_Entry<K, V>* removeHeapMin();
      T_Entry<K, V>* getHeapMin();
      void fprint(ofstream &fout);
      int size() {return end; }
private:
};
```

```
/* HeapPrioQ.h (2) */

template<typename K, typename V>
HeapPrioQueue<K, V>::HeapPrioQueue(int capa, string nm)
:CompleteBinaryTree(capa, nm)
{ }

template<typename K, typename V>
HeapPrioQueue<K, V>::~HeapPrioQueue()
{ }

template<typename K, typename V>
void HeapPrioQueue<K, V>::fprint(ofstream &fout)
{
    if (size() <= 0)
    {
       fout << "HeapPriorityQueue is Empty !!" << endl;
       return;
    }
    else
       CompleteBinaryTree::printCBT(fout);
}
```

```
/* HeapPrioQ.h (3) */

template<typename K, typename V>
int HeapPrioQueue<K, V>::insert(T_Entry<K, V>& elem)
{
    int index, parent_index;
    T_Entry<K, V> temp;
    if (isFull())
    {
        cout << this->getName() << " is Full !!" << endl;
        return size();
    }
    index = add_at_end(elem);

    /* up-heap bubbling */
    while (index != CBT_ROOT)
    {
        parent_index = parentIndex(index);
        if (t_array[index].getKey() >= t_array[parent_index].getKey())
            break;
        else
        {
            temp = t_array[index];
            t_array[index] = t_array[parent_index];
            t_array[parent_index] = temp;
            index = parent_index;
        }
    }
    return size();
}
```

```
/* HeapPrioQ.h (4) */

template<typename K, typename V>
T_Entry<K, V>* HeapPrioQueue<K, V>::getHeapMin()
{
    T_Entry<K, V>* pMinElem;
    if (size() <= 0)
    {
       return NULL;
    }
    pMinElem = (T_Entry<K, V>*) new T_Entry<K, V>;
    *pMinElem = getRootElement();
    return pMinElem;

}
```

```
/* HeapPrioQ.h (4) */

template<typename K, typename V>
T_Entry<K, V>* HeapPrioQueue<K, V>::removeHeapMin()
{
    int index_p, index_c, index_rc;
    T_Entry<K, V> *pMinElem;
    T_Entry<K, V> temp, t_p, t_c;
    int HPQ_size = size();

    if (HPQ_size <= 0)
    {
       return NULL;
    }
    pMinElem = (T_Entry<K, V>*) new T_Entry<K, V>;
    *pMinElem = getRootElement();
    if (HPQ_size == 1)
    {
       removeCBTEnd();
    }
    else
    {
        index_p = CBT_ROOT;
        t_array[CBT_ROOT] = t_array[end];
        end--;
```

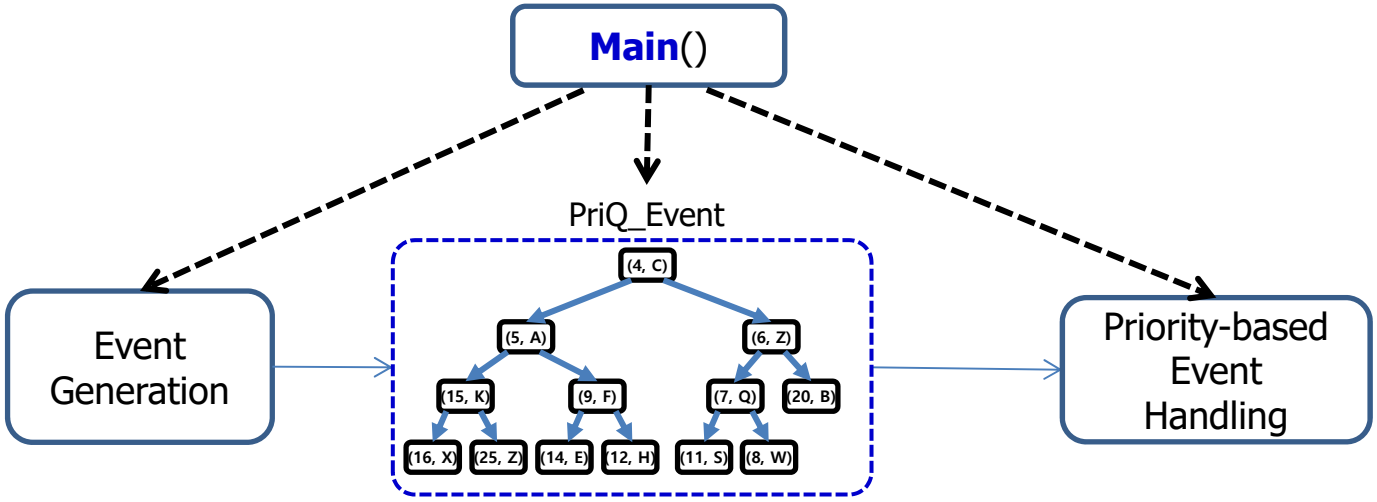
```
/* HeapPrioQ.h (5) */

        /* down-heap bubbling */
        while (hasLeftChild(index_p))
        {
            index_c = leftChildIndex(index_p);
            index_rc = rightChildIndex(index_p);
            if (hasRightChild(index_p) && (t_array[index_c] > t_array[index_rc]))
                index_c = index_rc;
            t_p = t_array[index_p];
            t_c = t_array[index_c];
            if (t_p > t_c)
            {
                //swap(index_u, index_c);
                temp = t_array[index_p];
                t_array[index_p] = t_array[index_c];
                t_array[index_c] = temp;
                index_p = index_c;
            }
            else
                 break;

        } // end while
    }
    return pMinElem;
}
#endif
```

Priority Queue의 응용 예제
Class Event
Class HeapPrioQ_Event

Priority Queue의 응용 예제

◆ Priority Queue 응용 예제

- Simple Simulation of Priority-based Event Handling
  - Event Generation
  - Event Handling
- Shared Priority Queue
  - PriQ_Event



class Event

```
/* Event.h (1) */

#ifndef EVENT_H
#define EVENT_H
#include <iostream>
#include <string>
#include <fstream>
#include <iomanip>
using namespace std;

enum EventStatus { GENERATED, ENQUEUED, PROCESSED, UNDEFINED };
#define MAX_EVENT_PRIORITY 100
#define NUM_EVENT_GENERATORS 10

class Event
{
    friend ostream& operator<<(ostream& fout, const Event& e);
public:
    Event() { } // default constructor
    Event(int event_id, int event_pri, int genAddr); //constructor
    void printEvent(ostream& fout);
    void setEventHandlerAddr(int evtHndlerAddr) { event_handler_addr = evtHndlerAddr; }
    void setEventGenAddr(int genAddr) { event_gen_addr = genAddr; }
```

```
/* Event.h (2) */

    void setEventNo(int evtNo) { event_no = evtNo; }
    void setEventPri(int pri) { event_pri = pri; }
    void setEventStatus(EventStatus evtStatus) { eventStatus = evtStatus; }
    int getEventPri() { return event_pri; }
    int getEventNo() { return event_no; }
    bool operator>(Event& e) { return (event_pri > e.event_pri); }
    bool operator<(Event& e) { return (event_pri < e.event_pri); }
private:
    int event_no;
    int event_gen_addr;
    int event_handler_addr;
    int event_pri;  // event_priority
    EventStatus eventStatus;
};

Event* genRandEvent(int evt_no);

#endif
```

```
/* Event.cpp (1) */
#include "Event.h"

Event::Event(int evt_no, int evt_pri, int evtGenAddr)
{
    event_no = evt_no;
    event_gen_addr = evtGenAddr;
    event_handler_addr = -1; // event handler is not defined at this moment
    event_pri = evt_pri;  // event_priority
    eventStatus = GENERATED;
}

Event* genRandEvent(int evt_no)
{
    Event *pEv;
    int evt_prio;
    int evt_generator_id;

    evt_prio = rand() % MAX_EVENT_PRIORITY;
    evt_generator_id = rand() % NUM_EVENT_GENERATORS;

    pEv = (Event *) new Event(evt_no, evt_prio, evt_generator_id);

    return pEv;
}
```

```
/* Event.cpp (2) */
#include "Event.h"

void Event::printEvent(ostream& fout)
{
    fout << "Event(pri:" << setw(3) << event_pri << ", gen:" << setw(3) << event_gen_addr;
    fout << ", no:" << setw(3) << event_no << ")";
}

ostream& operator<<(ostream& fout, const Event& evt)
{
    fout << "Event(pri:" << setw(3) << evt.event_pri << ", gen:" << setw(3) << evt.event_gen_addr;
    fout << ", no:" << setw(3) << evt.event_no << ")";

    return fout;
}
```

우선순위 기반 Event 처리 응용 프로그램

```
/* main() for Heap Priority Queue based on Complete Binary Tree (1) */
#include <iostream>
#include <fstream>
#include "Event.h"
#include "HeapPrioQ.h"
#include <string>
#include <stdlib.h>
using namespace std;

#define INITIAL_CBT_CAPA 100

void main()
{
    ofstream fout;
    string tName = "";
    char tmp[10];
    int priority = -1;
    int current_top_priority;
    int duration = 0;
    int size;
    int *pE;

    fout.open("output.txt");
    if (fout.fail())
    {
        cout << "Fail to open output.txt file for results !!" << endl;
        exit;
    }
```

```
/* main() for Heap Priority Queue based on Complete Binary Tree (2) */
    HeapPrioQueue<int, Event> HeapPriQ_Event(INITIAL_CBT_CAPA, string("Event_Heap_Priority_Queue"));
    Event *pEv;
    T_Entry<int, Event> entry_event, *pEntry_Event;

    for (int i = 10; i>0; i--)
    {
        pEv = genRandEvent(i);
        entry_event.setKey(pEv->getEventPri());
        entry_event.setValue(*pEv);
        HeapPriQ_Event.insert(entry_event);
        fout << "Insert ";
        pEv->printEvent(fout);
        fout << " ==> Size of Heap Priority Queue : " << setw(3) << HeapPriQ_Event.size() << endl;
    }
    fout << "Final status of insertions : " << endl;
    HeapPriQ_Event.fprintCBT_byLevel(fout);

    for (int i = 0; i<10; i++)
    {
        fout << "₩nCurrent top priority in Heap Priority Queue : ";
        pEntry_Event = HeapPriQ_Event.getHeapMin();
        pEntry_Event->fprint(fout);
        fout << endl;
        pEntry_Event = HeapPriQ_Event.removeHeapMin();
        fout << "Remove ";
        pEntry_Event->fprint(fout);
        fout << " ==> " << HeapPriQ_Event.size() << " elements remains." << endl;
        HeapPriQ_Event.fprintCBT_byLevel(fout);
        fout << endl;
    }
    fout.close();
} // end main();
```

```
Insert Event(pri: 41, gen:  7, no: 10) ==> Size of Heap Priority Queue :  1
Insert Event(pri: 34, gen:  0, no:  9) ==> Size of Heap Priority Queue :  2
Insert Event(pri: 69, gen:  4, no:  8) ==> Size of Heap Priority Queue :  3
Insert Event(pri: 78, gen:  8, no:  7) ==> Size of Heap Priority Queue :  4
Insert Event(pri: 62, gen:  4, no:  6) ==> Size of Heap Priority Queue :  5
Insert Event(pri:  5, gen:  5, no:  5) ==> Size of Heap Priority Queue :  6
Insert Event(pri: 81, gen:  7, no:  4) ==> Size of Heap Priority Queue :  7
Insert Event(pri: 61, gen:  1, no:  3) ==> Size of Heap Priority Queue :  8
Insert Event(pri: 95, gen:  2, no:  2) ==> Size of Heap Priority Queue :  9
Insert Event(pri: 27, gen:  6, no:  1) ==> Size of Heap Priority Queue : 10
Final status of insertions :
          [Key:81, Event(pri: 81, gen:  7, no:  4)]
     [Key:34, Event(pri: 34, gen:  0, no:  9)]
          [Key:69, Event(pri: 69, gen:  4, no:  8)]
[Key: 5, Event(pri:  5, gen:  5, no:  5)]
          [Key:41, Event(pri: 41, gen:  7, no: 10)]
               [Key:62, Event(pri: 62, gen:  4, no:  6)]
     [Key:27, Event(pri: 27, gen:  6, no:  1)]
               [Key:95, Event(pri: 95, gen:  2, no:  2)]
          [Key:61, Event(pri: 61, gen:  1, no:  3)]
               [Key:78, Event(pri: 78, gen:  8, no:  7)]

Current top priority in Heap Priority Queue : [Key: 5, Event(pri:  5, gen:  5, no:  5)]
Remove [Key: 5, Event(pri:  5, gen:  5, no:  5)] ==> 9 elements remains.
          [Key:81, Event(pri: 81, gen:  7, no:  4)]
     [Key:34, Event(pri: 34, gen:  0, no:  9)]
          [Key:69, Event(pri: 69, gen:  4, no:  8)]
[Key:27, Event(pri: 27, gen:  6, no:  1)]
          [Key:62, Event(pri: 62, gen:  4, no:  6)]
     [Key:41, Event(pri: 41, gen:  7, no: 10)]
               [Key:95, Event(pri: 95, gen:  2, no:  2)]
          [Key:61, Event(pri: 61, gen:  1, no:  3)]
               [Key:78, Event(pri: 78, gen:  8, no:  7)]
```

```
Current top priority in Heap Priority Queue : [Key:62, Event(pri: 62, gen:  4, no:  6)]
Remove [Key:62, Event(pri: 62, gen:  4, no:  6)] ==> 4 elements remains.
     [Key:81, Event(pri: 81, gen:  7, no:  4)]
[Key:69, Event(pri: 69, gen:  4, no:  8)]
     [Key:78, Event(pri: 78, gen:  8, no:  7)]
          [Key:95, Event(pri: 95, gen:  2, no:  2)]


Current top priority in Heap Priority Queue : [Key:69, Event(pri: 69, gen:  4, no:  8)]
Remove [Key:69, Event(pri: 69, gen:  4, no:  8)] ==> 3 elements remains.
     [Key:81, Event(pri: 81, gen:  7, no:  4)]
[Key:78, Event(pri: 78, gen:  8, no:  7)]
     [Key:95, Event(pri: 95, gen:  2, no:  2)]


Current top priority in Heap Priority Queue : [Key:78, Event(pri: 78, gen:  8, no:  7)]
Remove [Key:78, Event(pri: 78, gen:  8, no:  7)] ==> 2 elements remains.
[Key:81, Event(pri: 81, gen:  7, no:  4)]
     [Key:95, Event(pri: 95, gen:  2, no:  2)]


Current top priority in Heap Priority Queue : [Key:81, Event(pri: 81, gen:  7, no:  4)]
Remove [Key:81, Event(pri: 81, gen:  7, no:  4)] ==> 1 elements remains.
[Key:95, Event(pri: 95, gen:  2, no:  2)]


Current top priority in Heap Priority Queue : [Key:95, Event(pri: 95, gen:  2, no:  2)]
Remove [Key:95, Event(pri: 95, gen:  2, no:  2)] ==> 0 elements remains.
CBT is EMPTY now !!
```

Homework 8

Homework 8

8.1 템플릿 클래스 class HeapPriQ<int> 우선순위 큐를 사용한 정수형 난수 배열의 데이터 전달 기능 구현.

(1) 정수형 데이터 15개가 포함된 정수 배열 inputArray[NUM_DATA]를 준비하고, 14 ~ 0의 정수를 초기값으로 설정할 것.

(2) class HeapPrioQueue<int>를 기반으로 정수형 데이터를 위한 우선 순위 큐 HeapPrioQ_int를 구현할 것. 우선 순위 큐의 최초 크기는 1로 할 것.

(3) for-loop을 사용하여, inputArray[] 에 포함된 데이터를 차례로 읽어 HeapPrioQ_int에 insert 시킬 것.

(4) 15개의 데이터가 입력된 HeapPrioQ_int의 내부 상태를 fprintCBT_byLevel(fout) 멤버함수를 사용하여 출력할 것.

(5) for-loop을 사용하여 HeapPrioQ_int 에 포함되어 있는 데이터를 removeMin()을 사용하여 차례로 추출하고, 각 단계별로 HeapPrioQ_int의 내부 상태를 fprintCBT_byLevel(fout) 멤버함수를 사용하여 출력할 것.

```
/* main() for Heap Priority Queue based on Complete Binary Tree (1) */
#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;
    }
```

```
/* main() for Heap Priority Queue based on Complete Binary Tree (2) */

    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();
```

```
Final status of insertions :
                3
          2
                10
     1
                9
          4
                13
0
                7
          6
                12
     5
                11
          8
                14

Current top priority in Heap Priority Queue :   0
RemoveMin (0) from HeapPriQ_int ==> 14 elements remains.
          3
                10
     2
                9
          4
                13
1
                7
          6
                12
     5
                11
          8
                14
```

```
Current top priority in Heap Priority Queue :   1
RemoveMin (1) from HeapPriQ_int ==> 13 elements remains.
          10
     3
               9
          4
               13
2
               7
          6
               12
     5
               11
          8
               14

Current top priority in Heap Priority Queue :   2
RemoveMin (2) from HeapPriQ_int ==> 12 elements remains.
          10
     4
          9
               13
3
               7
          6
               12
     5
               11
          8
               14

Current top priority in Heap Priority Queue :   3
RemoveMin (3) from HeapPriQ_int ==> 11 elements remains.
          10
     9
          13
4
               7
          6
               12
     5
               11
          8
               14
```

```
Current top priority in Heap Priority Queue :  10
RemoveMin (10) from HeapPriQ_int ==> 4 elements remains.
     13
11
     12
          14

Current top priority in Heap Priority Queue :  11
RemoveMin (11) from HeapPriQ_int ==> 3 elements remains.
     13
12
     14

Current top priority in Heap Priority Queue :  12
RemoveMin (12) from HeapPriQ_int ==> 2 elements remains.
13
     14

Current top priority in Heap Priority Queue :  13
RemoveMin (13) from HeapPriQ_int ==> 1 elements remains.
14

Current top priority in Heap Priority Queue :  14
RemoveMin (14) from HeapPriQ_int ==> 0 elements remains.
CBT is EMPTY now !!
```