Problem 1 (R-8.11)

insertion sort는 정렬할 배열의 값을 하나 가져와서 정렬된 목록의 값들과 앞에서부터 비교하면서 넣을 위치를 찾아 삽입한다. 그러므로 정렬할 배열이 다음과 같은 경우에 최악의 경우가 될 것이다.

int arr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // 정렬할 원소의 개수 n이 10인 경우

즉, 이미 오름차순으로 정렬되어 있는 것을 다시 한 번 오름차순으로 정렬하려고 시도하는 경우 최악의 경우가 되는 것이다.

insertion sort의 과정을 살펴보면,

정렬된 값을 저장할 int sortedArr[10]; 을 선언하고

arr[0]을 가져와서 sortedArr 첫 칸에 넣는다.

arr[1]을 가져와서 sortedArr의 앞에서부터 크기를 비교하여 넣을 자리를 찾는다. 이 때 1과 비교를 한 번하고 뒤에 삽입하게 될 것이다.

arr[2]를 가져와서 sortedArr의 1과 비교하고 2와 비교한다. 그러고 그 뒤에 삽입한다.

이와 같은 과정이 arr[9]까지 계속 반복될 것이다.

arr[0]부터 arr[9]까지의 값을 가져 오는 것 : n

arr[0]의 값과 sortedArr의 값의 비교 : 1

arr[1]의 값과 sortedArr의 값의 비교 : 2

.

.

.

arr[n-1]의 값과 sortedArr의 값의 비교 : n

arr의 값들을 sortedArr에 저장 : n

모두 합하면 n + { n * (1 + n) } / 2 + n = $\frac{1}{2}n^2 + \frac{5}{2}n$

insertion sort가 Ω(n^2)임을 보이자.

insertion sort의 f(n) = $\frac{1}{2}n^2 + \frac{5}{2}n$

c>0이고 c는 실수, $n_0 \geq 1$이고 n_0는 정수일 때,

$f(n) \geq c*g(n)$, $n \geq n_0$ 를 만족하는 c와 n_0가 하나 이상씩 존재한다면,

f(n)는 big-Omega g(n) 이다.

c = 1/2 이라고 하자.

n = 1일 때 성립한다.

(5/2)*n은 항상 양수이므로 부등식이 n > 1에 대해 항상 성립한다.

따라서, f(n) is Ω(n^2) for n ≥ 1

Problem 2 (R-8.14)

heap sort는 정렬할 숫자들을 heap에 넣고 heap에서 하나씩 값을 가져와 마지막 index부터 첫 index까지 값을 넣는다. 이 때, max-heap을 이용하므로 큰 수가 우선 순위가 더 높다.

heap의 insertion : heap의 마지막 node에 추가한다. 단, 우선 순위가 만족되어야 하기 때문에 추가된 값이 있는 node와 그 node의 parent node를 우선 순위를 만족시키도록 값을 바꾼다. 그 과정을 root까지 반복한다.

heap의 removal : heap의 top element를 제거한다. 그리고 마지막 node를 root로 이동시킨다. 단, 우선 순위를 만족시켜야 하기 때문에 우선 순위가 가장 큰 child와 서로 값을 바꾼다. 그 과정을 external node까지 반복한다.

2	5	16	4	10	23	39	18	26	15

2를 heap에 넣는다.

5	2	16	4	10	23	39	18	26	15

5를 heap에 넣는다.

16	2	5	4	10	23	39	18	26	15

16을 heap에 넣는다.

16	4	5	2	10	23	39	18	26	15

4를 heap에 넣는다.

16	10	5	2	4	23	39	18	26	15

10을 heap에 넣는다.

23	10	16	2	4	5	39	18	26	15

23을 heap에 넣는다.

39	10	23	2	4	5	16	18	26	15

39를 heap에 넣는다.

39	10	23	10	4	5	16	2	26	15

18을 heap에 넣는다.

39	26	23	18	4	5	16	2	10	15

26을 heap에 넣는다.

39	26	23	18	15	5	16	2	10	4

15를 heap에 넣는다.

정렬할 숫자들을 heap에 모두 저장하고 난 다음, heap의 top element를 제거하며 마지막 칸에 값을 저장한다. 그 과정을 정렬할 원소의 개수만큼 반복하면 heap sort가 완료된다.

26	18	23	10	15	5	16	2	4	39
23	18	16	10	15	5	4	2	26	39
18	15	16	10	2	5	4	23	26	39
16	17	7	10	2	4	18	23	26	39
15	10	5	4	2	16	18	23	26	39
10	4	5	2	15	16	18	23	26	39
5	4	2	10	15	16	18	23	26	39
4	2	5	10	15	16	18	23	26	39
2	4	5	10	15	16	18	23	26	39
2	4	5	10	15	16	18	23	26	39

Problem 3 (R-8.23)

heap은 complete binary tree이다. 그러므로 배열 혹은 벡터로 값을 관리하는 것이 용이하다. 단, key 16으로 특정하였기에 벡터를 이용하지 않는 것이 더 간단할 것이지만, 알고리즘을 보이는 문제이고 임의의 key가 제시되고 그 key를 갖는 node를 제거하는 것이 목표가 될 수 있기 때문에 벡터를 이용하는 것이 더 직관적이고 간편해질 것이다.

node 하나에 여러 값(key, 실제 데이터)이 저장되어 있으므로 구조체 혹은 클래스를 이용한다.

BFS를 이용하여 위에서 아래로 왼쪽에서 오른쪽으로 root부터 값을 하나씩 벡터에 저장한다.

벡터에는 node의 주소를 저장한다.

그리고 key가 16인 node를 제거해야 한다. 그런데 key가 16인 node는 가장 마지막 node가 아니기 때문에 heap의 구조를 망가뜨린다. complete binary tree이므로 중간이 비어 있어서는 안 된다. 그래서 key가 16인 node를 제거하고 마지막 node를 가져온다. 이 때 우선 순위가 위배되면 안 되므로 up-heap bubbling과 같은 과정을 거친다.

정리하자면,

1. heap data structure의 모든 값들을 root부터 위에서 아래로 왼쪽에서 오른쪽으로 이동하며 하나씩 가져와서 벡터에 저장한다.

2. key가 16인 node를 벡터에서 찾아서 그 node로 가서 제거한다.

3. key가 16인 node가 있던 위치에 마지막 node를 삽입한다.

4. up-heap bubbling을 시행한다.

Problem 4 (C-8.31)

총 3가지 경우로 나누어 생각하였다. w는 last node를 칭한다.

(1) last node가 left child 일 때

- w의 parent의 right에 새로운 node를 추가하면 된다.

(2) last node가 right child 이고, 그것이 속한 level이 가득 차 있을 때

- 2번 경우임을 확인하기 위해 tree의 height값을 이용한다. height를 알면 node 개수의 최대값을 구할 수 있는데 그 최대값과 size가 일치하는 경우가 바로 2번 경우이다.

- preorder traversal을 하면서 처음으로 만난 external node에 새로운 node를 추가하고 종료

(3) last node가 right child 이고, 그것이 속한 level이 가득 차 있지 않을 때

- 2번 경우가 아닌 경우가 3번 경우이다.

- preorder traversal을 하면서 w(w도 external node이다)를 만나면 check를 0에서 1로 바꿔준다. 그리고 계속 traversal을 하면서 또다른 external node를 만나면 그 때 check가 1이므로 조건문이 수행되어서 그 node에 새로운 node를 추가하고 종료한다.

아래는 위의 내용을 대략적으로 코드화한 것이다.

```
#include<iostream>

int done = 0;

void preorder1(Node* v, int e){
	if(!done)
		return;

	if (v->left == NULL && v->right == NULL) {
		Node* temp = new Node;
		temp->elem = e;
		temp->parent = v;
		temp->left = NULL;
		temp->right = NULL;
		v->left = temp;
```

```
                done = 1;
                return;
        }
        preorder1(v->left, e);
        preorder1(v->right, e);
}


void preorder2(Node* v, int e, int* check) {
        if(!done)
                return;

        if (v->left == NULL && v->right == NULL) {
                if (v == w) { // external node가 last node라면
                        *check = 1;
                }
                if (*check) {
                        Node* temp = new Node;
                        temp->elem = e;
                        temp->parent = v;
                        temp->left = NULL;
                        temp->right = NULL;
                        done = 1;
                        v->left = temp;
                }
                return;
        }
        preorder2(v->left, e, check);
        preorder2(v->right, e, check
}
int height2(const Tree& T, const Position& p) {
 if(p.isExternal()) return 0;
 int h = 0;
 Position ch = p.children();
 for(Iterator q = ch.begin(); q != ch.end(); ++q) {
        int temp;
        temp = height2(T, *q)
        if ( h < temp )
                h = temp;
 }
```

```
return h + 1;
}
```

```
int main() {
	LinkedBinaryTree LBT;
	int success = 0;
	int check = 0;
	Node* root = LBT.root();
	int e;

	if (w->parent->left == w) {
		Node* z = new z;
		w->parent->right = z;
		z->elem = e;
	}
	else if (w->parent->right == w) {
		int exp = 2;
		int h;
		for (h = 0; h <= _HEIGHT; h++) {
			if (LBT.size() == exp - 1) {
				preorder1(root, e);
				success = 1;
			}
			exp = 2 * exp;
		}
		if (!success) { //
			preorder2(root, e, &check);
		}
	}
	return 0;
}
```

running time을 계산한다.

preorder는 O(n)

height2는 O(n)

2번 경우를 찾는 과정 O(height) = O(logn)

즉, 전체의 running time은 O(n)

Problem 5 (C-8.18)

heap-order property를 만족하는 두 개의 binary tree를 합쳐야 한다. heap-order property를 만족하는 tree 2개이므로 가장 우선 순위가 높은 것부터 하나씩 뽑아서 우선 순위를 비교해가며 새로운 트리 T에 넣으면 된다.

작은 수가 높은 우선 순위를 갖는다고 가정하면, 즉 오름차순으로 정렬되어 있다고 가정하면,

```
while(tree T1과 tree T2 둘 다 완전히 빌 때 까지) {
        if(T1.min() <T2.min())
          { T = T1.min(); T1.removeMin(); }
        else if(T1.min() > T2.min())
          { T = T2.min(); T2.removeMin(); }
        else // 두 개의 우선 순위 값이 같을 때
          { T = T1.min(); T1.removeMin(); T2.removeMin(); }
}
```

min()은 O(1)이고

removeMin()은 root에 있는 값을 제거한 후 가장 마지막 node를 root에 넣은 뒤 down-heap bubbling을 하는 것이므로 O(h)이다. 따라서 총 h = h1 + h2 이므로

전체적인 running time은 O(h1+h2)를 만족하게 된다.