5.8 Set 표현

set를 표현하는 데 tree를 사용한다. set이 disjoint하다고 가정할 때 그림 5.25처럼 S1 = {0, 6, 7, 8}, S2 = {1, 4, 9}, S3 = {2, 3, 5}를 tree로 표현한다. set을 만드는 이유는 두 개의 element가 있을 때 같은 set인지 다른 set인지를 구분하기 위함이다. set이 필요한 응용은 그래프의 minimal spanning tree와 shortest path를 구할 때 사용된다. 따라서 set의 표현을 tree 모양이지마는 배열로 구현하는 방법을 사용한다.

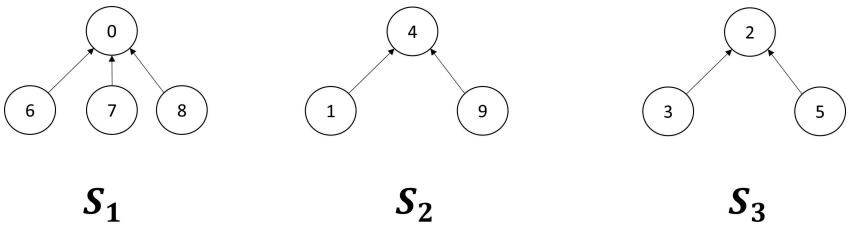


그림 5.25 Set의 트리 표현.

set operations[1]은 union과 find 함수이다. disjoint set union은 그림 5.26처럼 합집합을 만드는 함수이다. find(i)는 element i를 찾는 함수이다. find( )를 사용하여 2개의 element가 같은 집합인지 다른 집합인지 구분하기 위하여 find(i) == find(j) 을 사용한다.

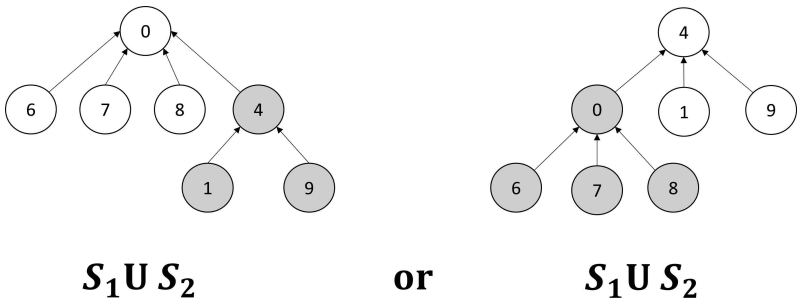


그림 5.26 합집합 처리.

Union은 2개의 set를 합치는 것으로 union(0, 4)을 사용한다. set s1의 root가 0이고 s2의 root가 4일 때 s2가 s1의 child로 합치거나 s1이 s2의 child로 합친다. union( )의 알고리즘에 따라 skewed tree가 될 수도 있고 balanced tree가 될 수가 있다.
set을 표현하는 tree node를 그림 5.27처럼 배열 parent[MaxElements]로 표현한다. parent[ ]의 값이 음수이면 root를 나타낸다. parent[1] = 4이고 parent[4] = -1이므로 {1,4}는 같은 set에 속하며 4가 root에 해당된다. 다음 set의 표현은 union( )하기 전의 set S1, S2, S3을 표현한 것이다.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

그림 5.27 집합의 배열 표현[1].

union(i, j)은 root가 i와 j인 2개의 tree를 합친다. find(i)는 element i를 포함하는 tree의 root를 return한다. set를 tree로 표현하였지마는 구현은 int * parent;로 배열로 처리한다. find(i)는 i를 포함하는 tree의 root를 return한다. 이때 root의 parent는 -1이다. union(i, j)는 i가 j의 subtree가 되도록 parent[i] = j로 구현한다. 소스코드 5.8에서 구현된 void Sets::SimpleUnion(int i, int j)은 set의 root의 값이 개수를 표현한 -count를 사용한다. 이때 i의 root를 찾아 i라고 하고, j의 root도 찾아서 i와 j가 다르면 합치며 -count을 더하여 갯수를 증가시킨다. SimpleUnion(i,j)는 갯수를 고려하지 않고 합치는 함수인데 WeightedUnion(i,j)를 구동하기 위하여 -count를 포함하도록 알고리즘을 수정하였다.

//소스 코드 5.8: Set 표현

```
/* sets - graph의 최단경로에서 사용
*/
#include <iostream>
#include <stdlib.h>
#include <iomanip>
using namespace std;

const int HeapSize = 100;

class Sets {
public:
        Sets(int);
        void display();
        void SimpleUnion(int, int);
        int SimpleFind(int);
        void WeightedUnion(int, int);
        int CollapsingFind(int);
private:
        int* parent;
        int n;
};

Sets::Sets(int sz = HeapSize)
{
        n = sz;
        parent = new int[sz + 1]; // Don't want to use parent[0]
        for (int i = 1; i <= n; i++) parent[i] = -1;  // 0 for Simple versions
}
```

```
void Sets::SimpleUnion(int i, int j)
// Replace the disjoint sets with roots i and j, i != j with their union
{
        // i,j는 임의 노드
        while (parent[i] > 0)
                i = parent[i];
        while (parent[j] > 0)
                j = parent[j];
        if (i == j)
                return;
        if (parent[i] < parent[j]) {
                parent[i] += parent[j];
                parent[j] = i;
        }
        else {
                parent[j] += parent[i];
                parent[i] = j;
        }
}

int Sets::SimpleFind(int i)
// Find the root of the tree containing element i
{
        while (parent[i] > 0) i = parent[i];
        return i;
}

void Sets::WeightedUnion(int i, int j)
// Union sets with roots i and j, i != j, using the weighting rule.
// parent[i]=-count[i] and parent[i]=-count[i].
{
        while (parent[i] > 0)
                i = parent[i];
        while (parent[j] > 0)
                j = parent[j];
        if (i == j)
                return;
        int temp = parent[i] + parent[j];
        if (parent[i] > parent[j]) { // i has fewer nodes
                parent[i] = j;
                parent[j] = temp;
        }
        else { // j has fewer nodes
```

```
                parent[j] = i;
                parent[i] = temp;
        }
}

int Sets::CollapsingFind(int i)
// Find the root of the tree containing element i.
// Use the collapsing rule to collapse all nodes from @i@ to the root
{
        int r;
        for (r = i; parent[r] > 0; r = parent[r]); // find root
        while (i != r) {
                int s = parent[i];
                parent[i] = r;
                i = s;
        }
        return r;
}



void Sets::display()
{
        cout << "display:index=  ";
        for (int i = 1; i <= n; i++) cout <<std::setw(3)<< std::setfill(' ')<< i;
        cout << "\n";
        cout << "display: value= ";
        for (int i = 1; i <= n; i++) cout << std::setw(3) << std::setfill(' ') << parent[i];
        cout << "\n";
}


int main(void)
{
        Sets m(20);
        m.SimpleUnion(7,1); m.SimpleUnion(2,3); m.SimpleUnion(4,5); m.SimpleUnion(6,7);
        m.SimpleUnion(4,2);             m.SimpleUnion(5,7);             m.SimpleUnion(8,10);
m.SimpleUnion(13,11);
        m.SimpleUnion(12,9); m.SimpleUnion(14,20); m.SimpleUnion(16,19);
        m.SimpleUnion(17,18); m.SimpleUnion(12, 19); m.SimpleUnion(13,15);
        cout << "SimpleUnion() 실행 결과::" << endl;
        m.display();
        cout << "find 5: " << m.CollapsingFind(5) << endl;
```

```
        m.WeightedUnion(1, 2); m.WeightedUnion(1, 4); m.WeightedUnion(3, 9);
m.WeightedUnion(7, 15);
        m.WeightedUnion(12, 18); m.WeightedUnion(4, 16);
        cout << "WeightedUnion() 실행 결과::" << endl;
        m.display();

        cout << "find 1: " << m.CollapsingFind(1) << endl;
        cout << "find 3: " << m.CollapsingFind(3) << endl;
        cout << "find 5: " << m.CollapsingFind(5) << endl;
        cout << "find 7: " << m.CollapsingFind(7) << endl;
        cout << "find 11: " << m.CollapsingFind(11) << endl;
        cout << "find 14: " << m.CollapsingFind(14) << endl;
        cout << "find 16: " << m.CollapsingFind(16) << endl;
        cout << "find 18: " << m.CollapsingFind(18) << endl;
        cout << "CollapsingFind() 실행후::" << endl;
        m.display();
        system("pause");
        return 0;
}
```

simpleUnion으로 n 개의 set (Si = {i}), parent[i] = -1에 대하여 union(0,1), union(1,2), union(2,3), ..., union(n-2,n-1)을 실행하면 degenerate tree가 된다. degenerate trees를 피하는 방법은 그림 5.28처럼 union(i,j)에 대하여 weighting-rule[1]를 사용하는 것이다. 2개의 set에 대한 노드 개수를 사용하여 노드 개수가 작은 set이 노드 개수가 많은 set에 child로 들어가는 것이다.

if (root i 의 node 숫자 < root j 의 node 숫자),

then j가 i의 parent
else i가 j의 parent

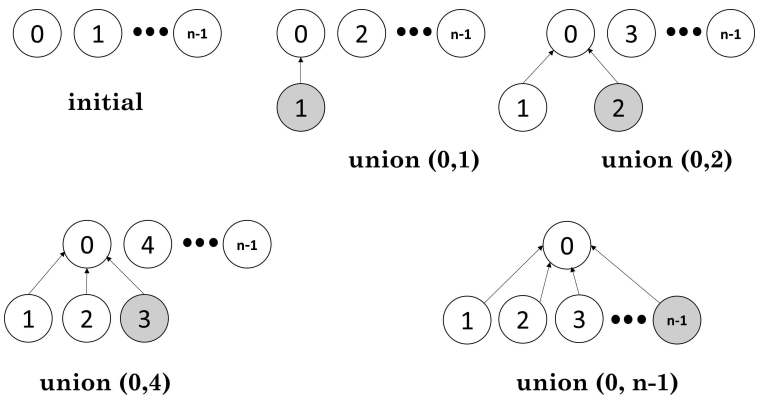
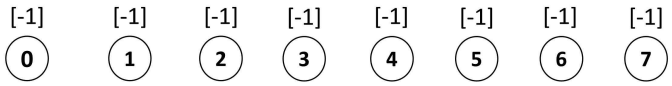


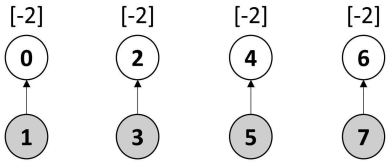
그림 5.28 Weighting rule를 사용한 합집합 처리.

tree의 root에 count field를 사용하여 weighting rule을 구현한다. root에 count field를 따로 두지 않고 그림 5.29처럼 parent[ ]의 값이 음수이면 root이면서 set의 노드 개수를 표현한다. void Sets::WeightedUnion(int i, int j)에서 parent[i]와 parent[j]을 비교하여 weighting rule을 적용한다.

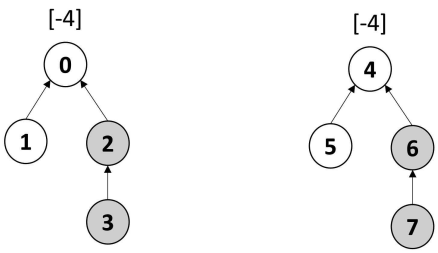
collapsing rule를 사용하여 find algorithm을 수정한다. Collapsing Rule은 j가 i로 부터 root로 가는 경로상의 node이고 parent[i] != root(i)이면 parent[j] = root(i)로 만들어 root로 가는 경로를 줄이는 방법이다. int Sets::CollapsingFind(int i) 함수가 collapsing rule을 사용한다.



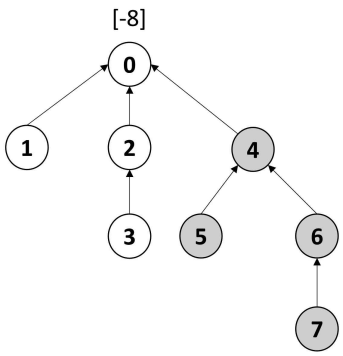
(a) 초기 Height -1 trees



(b) Height -2 trees는 *union* (0,1), (2,3), (4,5) and (6,7)의 결과



(c) Height -3 trees는 *union* (0,2), and (4,6)의 결과



(d) Height -4 tree는 *union* (0,4)의 결과

그림 5.29 union과 find 처리.