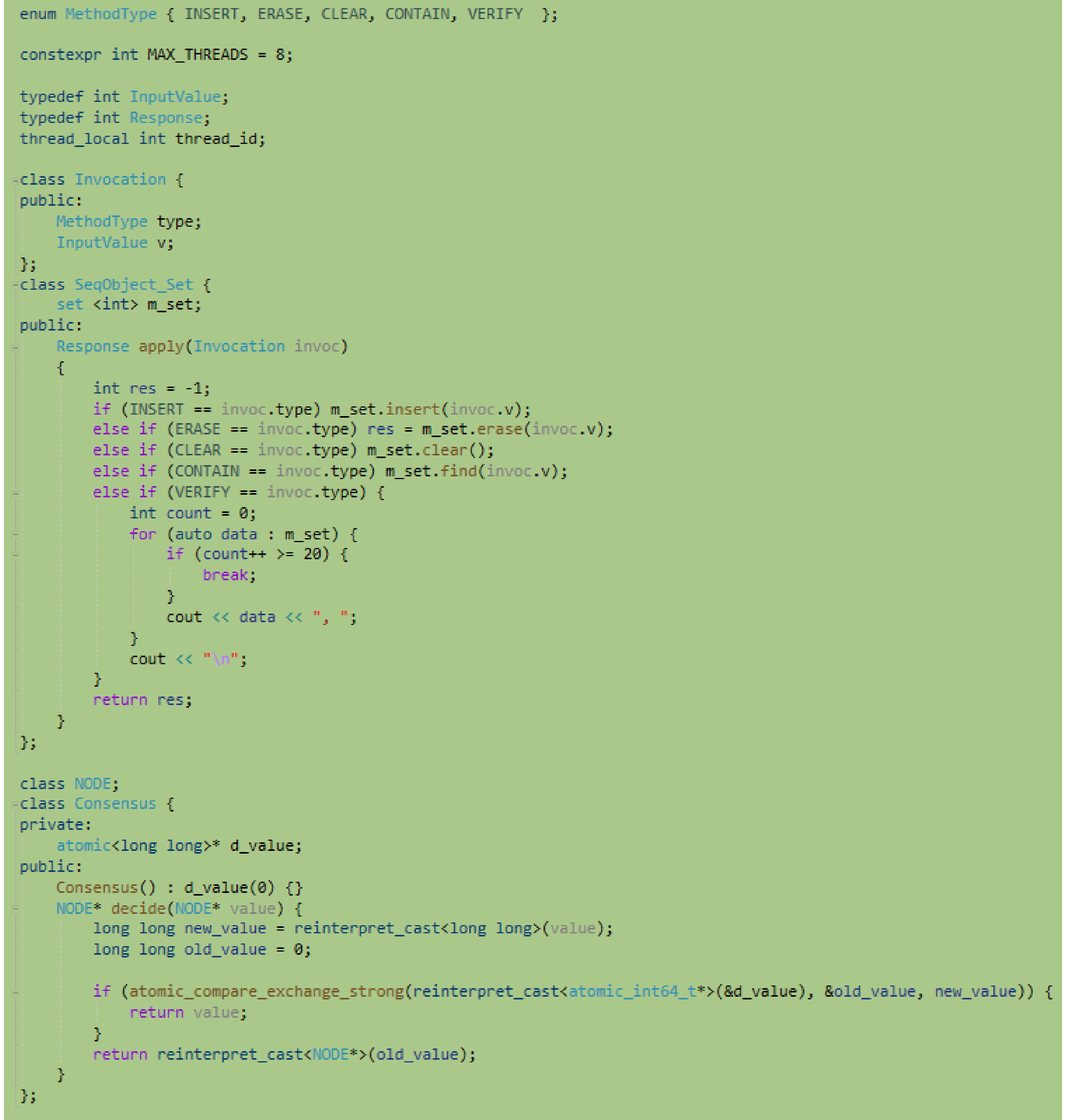
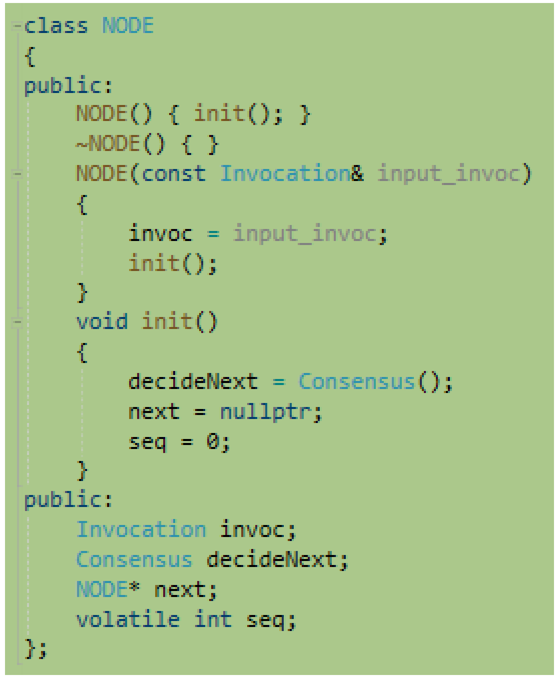
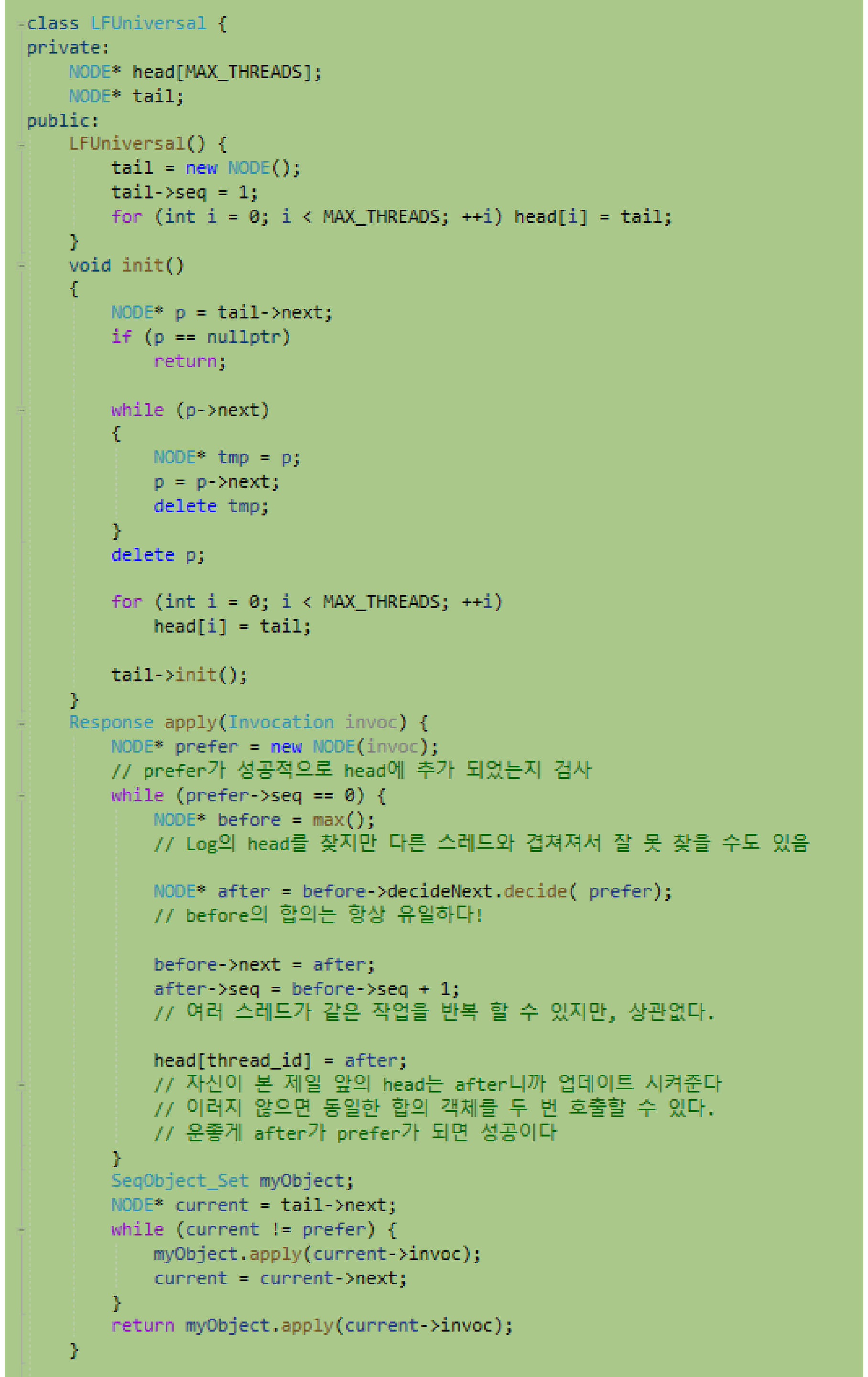


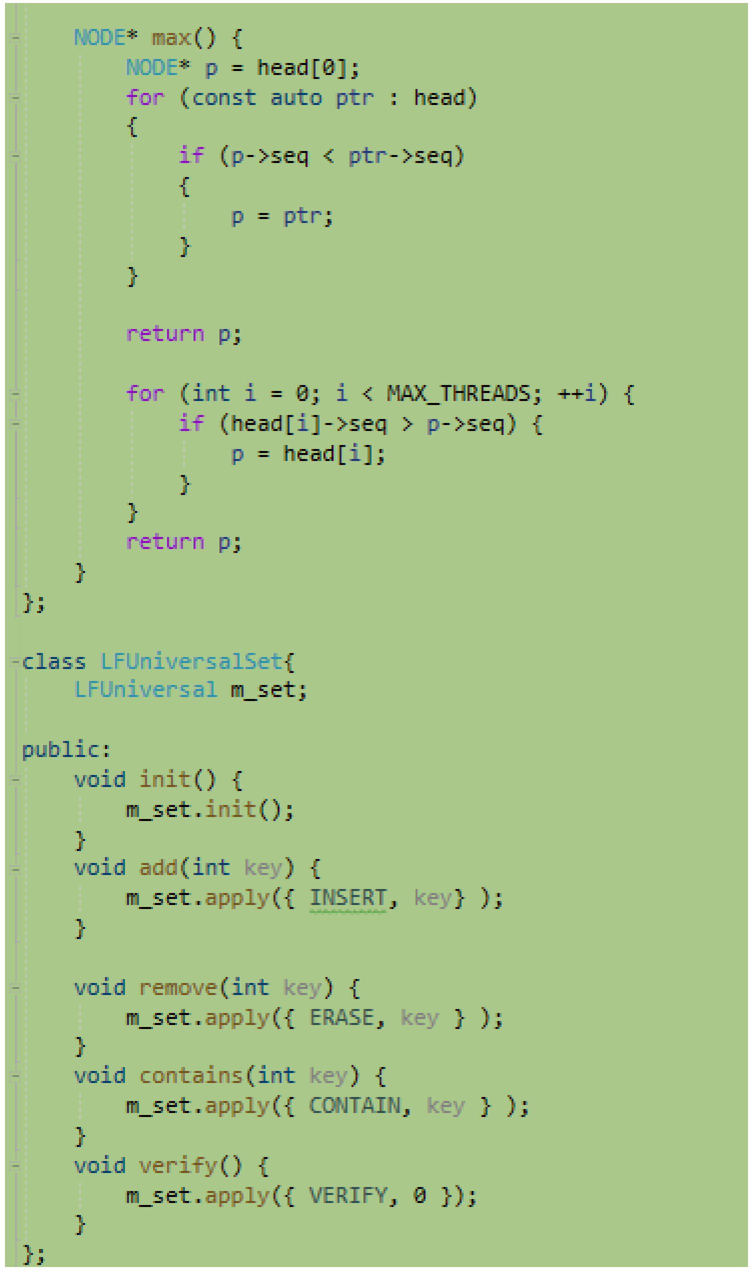
```cpp
enum MethodType { INSERT, ERASE, CLEAR, CONTAIN, VERIFY  };

constexpr int MAX_THREADS = 8;

typedef int InputValue;
typedef int Response;
thread_local int thread_id;

class Invocation {
public:
    MethodType type;
    InputValue v;
};
class SeqObject_Set {
    set <int> m_set;
public:
    Response apply(Invocation invoc)
    {
        int res = -1;
        if (INSERT == invoc.type) m_set.insert(invoc.v);
        else if (ERASE == invoc.type) res = m_set.erase(invoc.v);
        else if (CLEAR == invoc.type) m_set.clear();
        else if (CONTAIN == invoc.type) m_set.find(invoc.v);
        else if (VERIFY == invoc.type) {
            int count = 0;
            for (auto data : m_set) {
                if (count++ >= 20) {
                    break;
                }
                cout << data << ", ";
            }
            cout << "\n";
        }
        return res;
    }
};

class NODE;
class Consensus {
private:
    atomic<long long>* d_value;
public:
    Consensus() : d_value(0) {}
    NODE* decide(NODE* value) {
        long long new_value = reinterpret_cast<long long>(value);
        long long old_value = 0;

        if (atomic_compare_exchange_strong(reinterpret_cast<atomic_int64_t*>(&d_value), &old_value, new_value)) {
            return value;
        }
        return reinterpret_cast<NODE*>(old_value);
    }
};
```



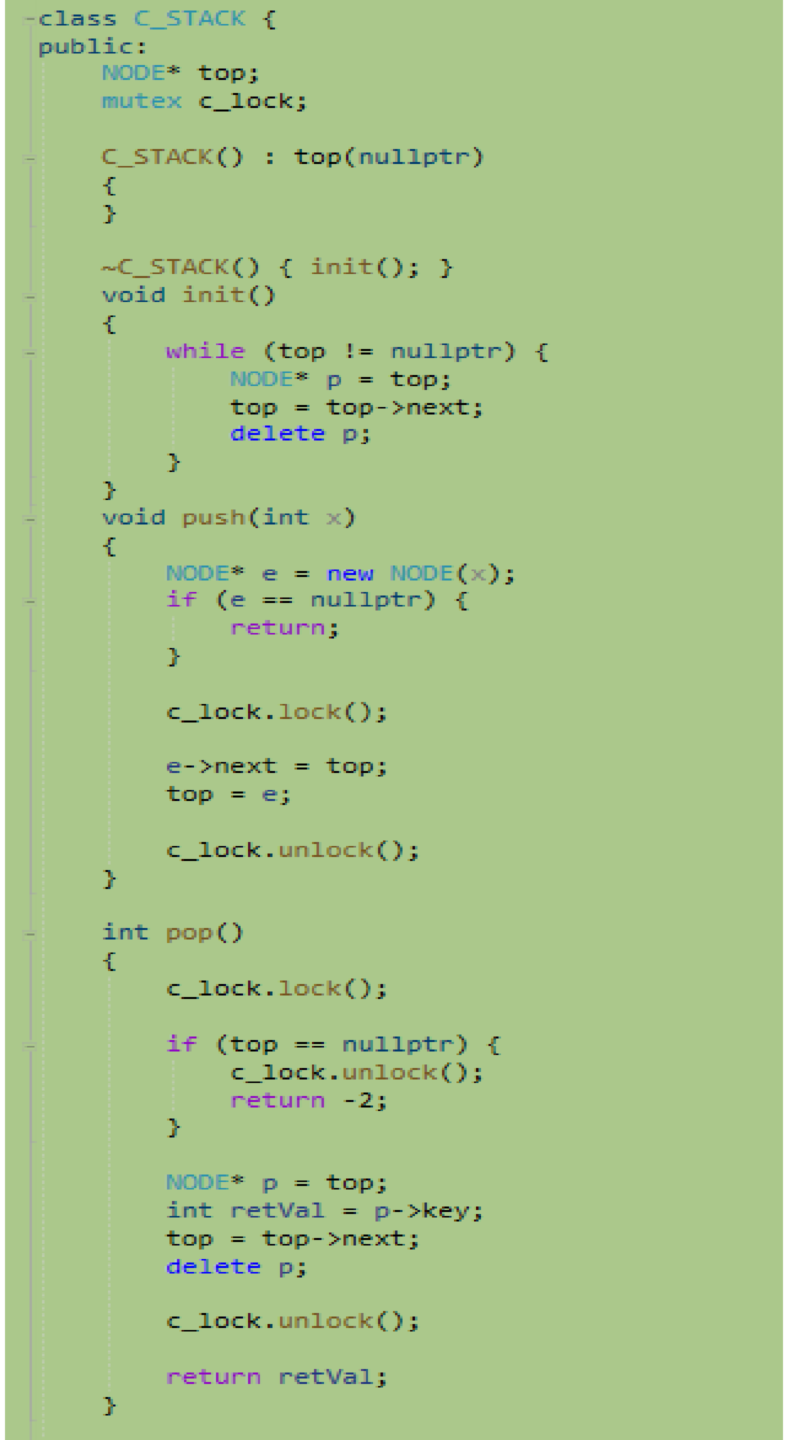
```cpp
class NODE
{
public:
    NODE() { init(); }
    ~NODE() { }
    NODE(const Invocation& input_invoc)
    {
        invoc = input_invoc;
        init();
    }
    void init()
    {
        decideNext = Consensus();
        next = nullptr;
        seq = 0;
    }
public:
    Invocation invoc;
    Consensus decideNext;
    NODE* next;
    volatile int seq;
};
```



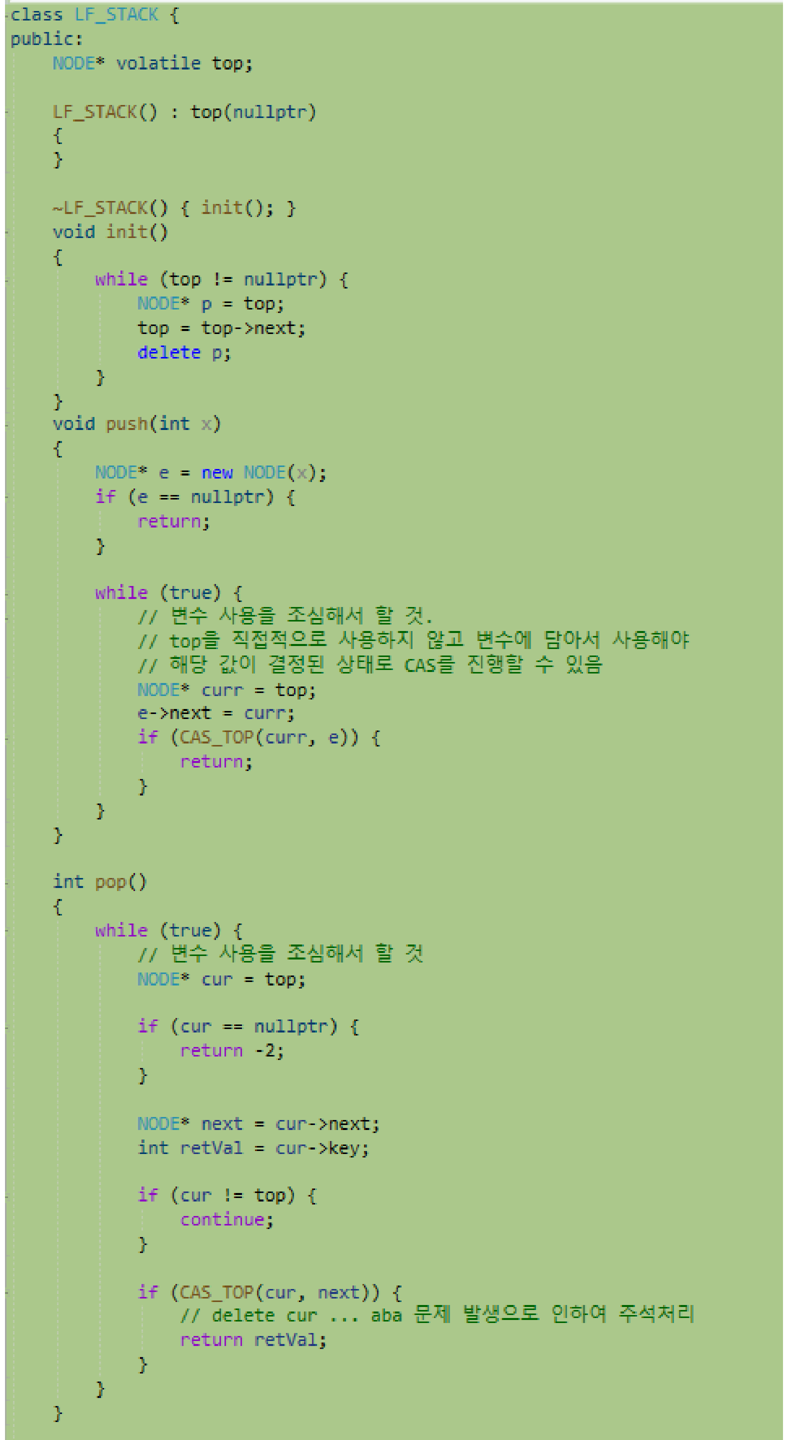
```
class LFUniversal {
private:
    NODE* head[MAX_THREADS];
    NODE* tail;
public:
    LFUniversal() {
        tail = new NODE();
        tail->seq = 1;
        for (int i = 0; i < MAX_THREADS; ++i) head[i] = tail;
    }
    void init()
    {
        NODE* p = tail->next;
        if (p == nullptr)
            return;

        while (p->next)
        {
            NODE* tmp = p;
            p = p->next;
            delete tmp;
        }
        delete p;

        for (int i = 0; i < MAX_THREADS; ++i)
            head[i] = tail;

        tail->init();
    }
    Response apply(Invocation invoc) {
        NODE* prefer = new NODE(invoc);
        // prefer가 성공적으로 head에 추가 되었는지 검사
        while (prefer->seq == 0) {
            NODE* before = max();
            // Log의 head를 찾지만 다른 스레드와 겹쳐져서 잘 못 찾을 수도 있음

            NODE* after = before->decideNext.decide( prefer);
            // before의 합의는 항상 유일하다!

            before->next = after;
            after->seq = before->seq + 1;
            // 여러 스레드가 같은 작업을 반복 할 수 있지만, 상관없다.

            head[thread_id] = after;
            // 자신이 본 제일 앞의 head는 after니까 업데이트 시켜준다
            // 이러지 않으면 동일한 합의 객체를 두 번 호출할 수 있다.
            // 운좋게 after가 prefer가 되면 성공이다
        }
        SeqObject_Set myObject;
        NODE* current = tail->next;
        while (current != prefer) {
            myObject.apply(current->invoc);
            current = current->next;
        }
        return myObject.apply(current->invoc);
    }
```



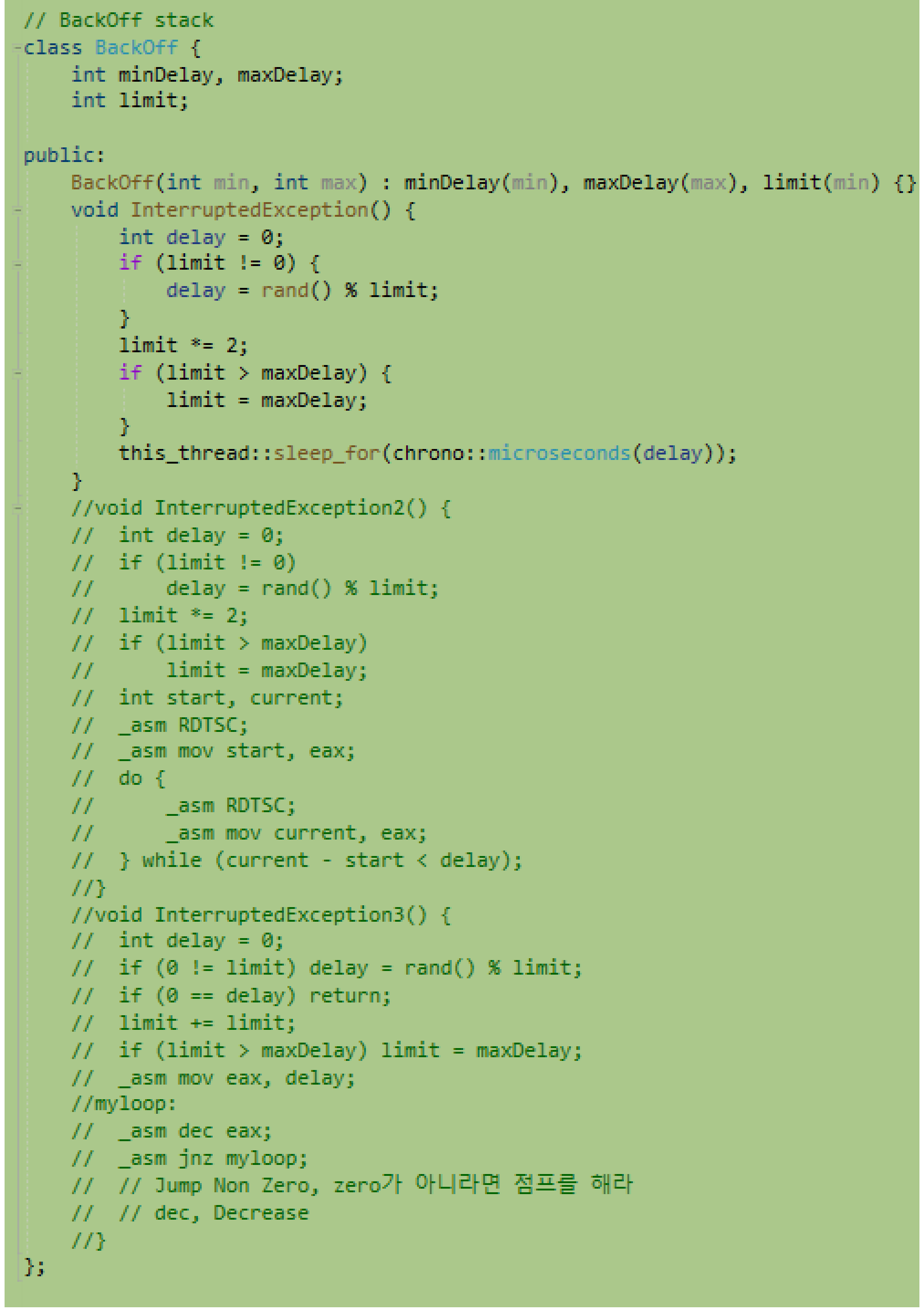
```
    NODE* max() {
        NODE* p = head[0];
        for (const auto ptr : head)
        {
            if (p->seq < ptr->seq)
            {
                p = ptr;
            }
        }

        return p;

        for (int i = 0; i < MAX_THREADS; ++i) {
            if (head[i]->seq > p->seq) {
                p = head[i];
            }
        }
        return p;
    }
};

class LFUniversalSet{
    LFUniversal m_set;

public:
    void init() {
        m_set.init();
    }
    void add(int key) {
        m_set.apply({ INSERT, key} );
    }

    void remove(int key) {
        m_set.apply({ ERASE, key } );
    }
    void contains(int key) {
        m_set.apply({ CONTAIN, key } );
    }
    void verify() {
        m_set.apply({ VERIFY, 0 });
    }
};
```



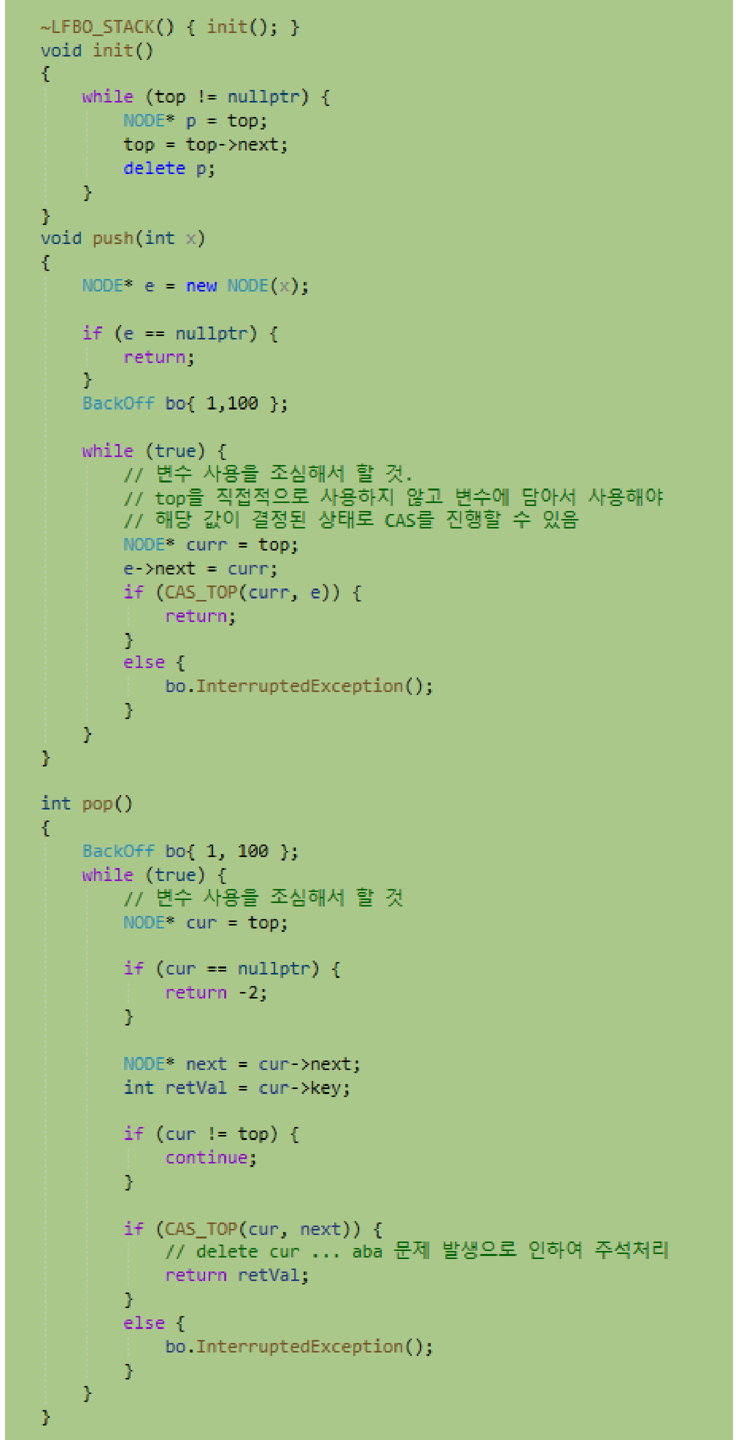
```cpp
class C_STACK {
public:
    NODE* top;
    mutex c_lock;

    C_STACK() : top(nullptr)
    {
    }

    ~C_STACK() { init(); }
    void init()
    {
        while (top != nullptr) {
            NODE* p = top;
            top = top->next;
            delete p;
        }
    }
    void push(int x)
    {
        NODE* e = new NODE(x);
        if (e == nullptr) {
            return;
        }

        c_lock.lock();

        e->next = top;
        top = e;

        c_lock.unlock();
    }

    int pop()
    {
        c_lock.lock();

        if (top == nullptr) {
            c_lock.unlock();
            return -2;
        }

        NODE* p = top;
        int retVal = p->key;
        top = top->next;
        delete p;

        c_lock.unlock();

        return retVal;
    }
```



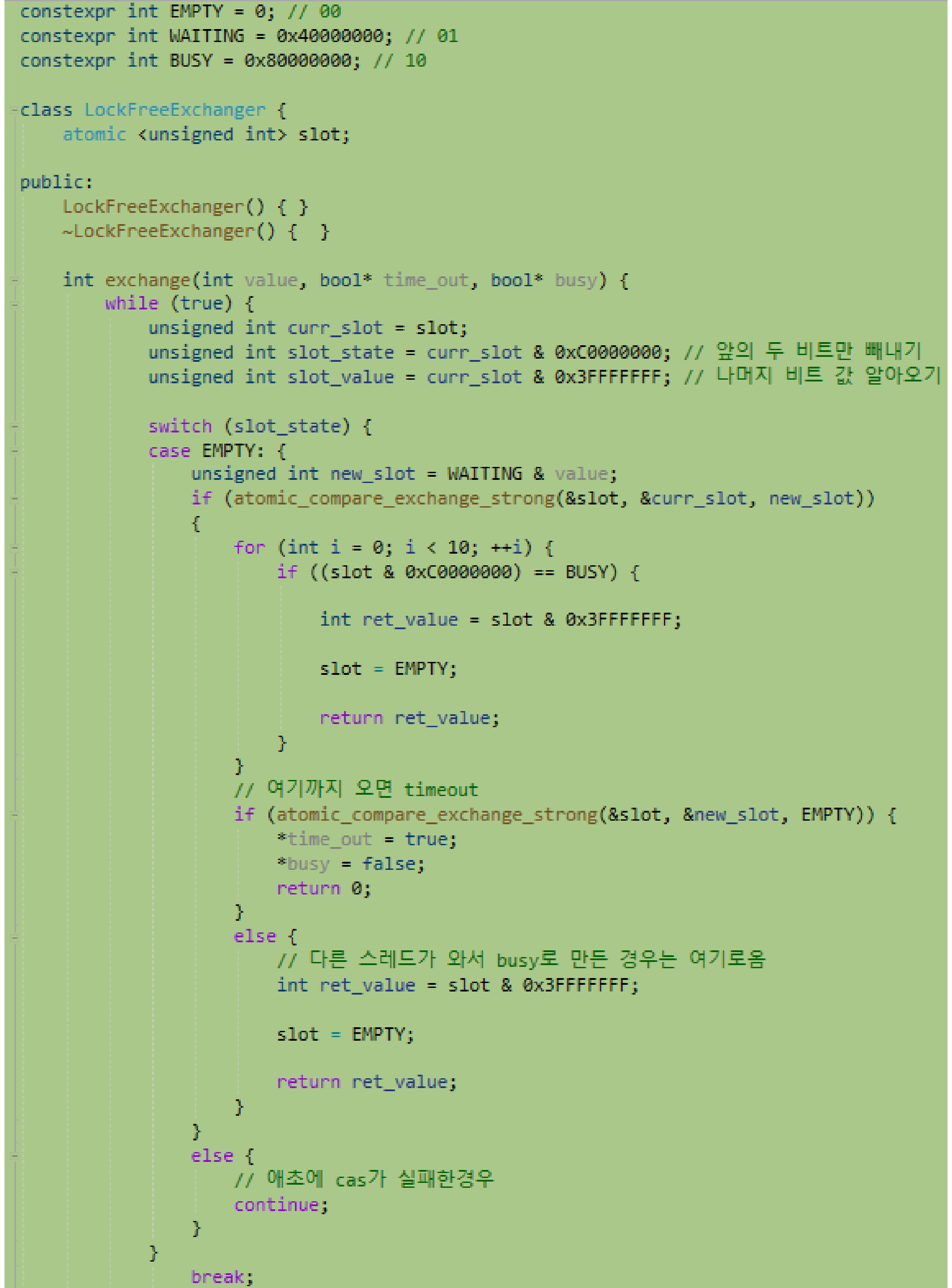
```cpp
class LF_STACK {
public:
    NODE* volatile top;

    LF_STACK() : top(nullptr)
    {
    }

    ~LF_STACK() { init(); }
    void init()
    {
        while (top != nullptr) {
            NODE* p = top;
            top = top->next;
            delete p;
        }
    }
    void push(int x)
    {
        NODE* e = new NODE(x);
        if (e == nullptr) {
            return;
        }

        while (true) {
            // 변수 사용을 조심해서 할 것.
            // top을 직접적으로 사용하지 않고 변수에 담아서 사용해야
            // 해당 값이 결정된 상태로 CAS를 진행할 수 있음
            NODE* curr = top;
            e->next = curr;
            if (CAS_TOP(curr, e)) {
                return;
            }
        }
    }

    int pop()
    {
        while (true) {
            // 변수 사용을 조심해서 할 것
            NODE* cur = top;

            if (cur == nullptr) {
                return -2;
            }

            NODE* next = cur->next;
            int retVal = cur->key;

            if (cur != top) {
                continue;
            }

            if (CAS_TOP(cur, next)) {
                // delete cur ... aba 문제 발생으로 인하여 주석처리
                return retVal;
            }
        }
    }
```



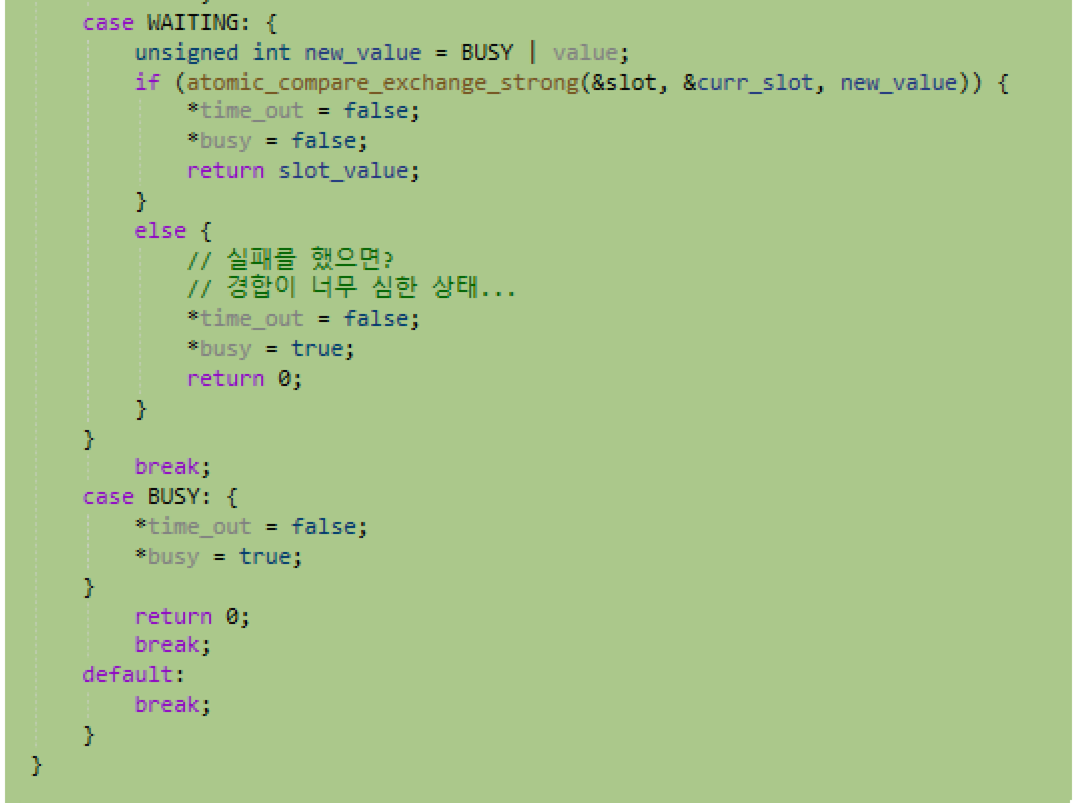
```
// BackOff stack
class BackOff {
    int minDelay, maxDelay;
    int limit;

public:
    BackOff(int min, int max) : minDelay(min), maxDelay(max), limit(min) {}
    void InterruptedException() {
        int delay = 0;
        if (limit != 0) {
            delay = rand() % limit;
        }
        limit *= 2;
        if (limit > maxDelay) {
            limit = maxDelay;
        }
        this_thread::sleep_for(chrono::microseconds(delay));
    }
    //void InterruptedException2() {
    //  int delay = 0;
    //  if (limit != 0)
    //      delay = rand() % limit;
    //  limit *= 2;
    //  if (limit > maxDelay)
    //      limit = maxDelay;
    //  int start, current;
    //  _asm RDTSC;
    //  _asm mov start, eax;
    //  do {
    //      _asm RDTSC;
    //      _asm mov current, eax;
    //  } while (current - start < delay);
    //}
    //void InterruptedException3() {
    //  int delay = 0;
    //  if (0 != limit) delay = rand() % limit;
    //  if (0 == delay) return;
    //  limit += limit;
    //  if (limit > maxDelay) limit = maxDelay;
    //  _asm mov eax, delay;
    //myloop:
    //  _asm dec eax;
    //  _asm jnz myloop;
    //  // Jump Non Zero, zero가 아니라면 점프를 해라
    //  // dec, Decrease
    //}
};
```



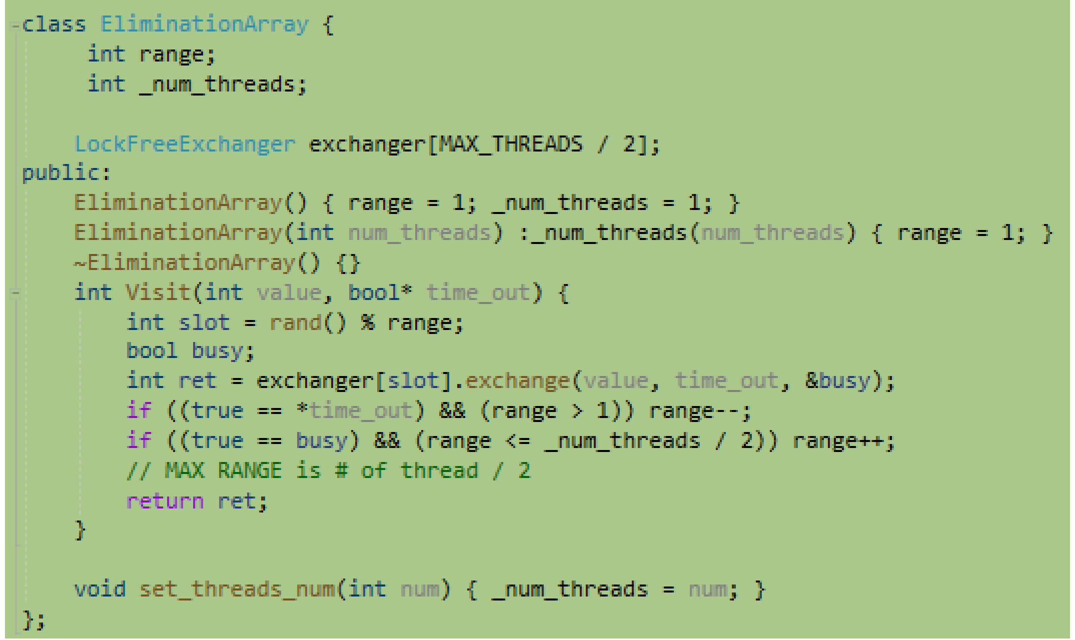
```
~LFBO_STACK() { init(); }
void init()
{
    while (top != nullptr) {
        NODE* p = top;
        top = top->next;
        delete p;
    }
}
void push(int x)
{
    NODE* e = new NODE(x);

    if (e == nullptr) {
        return;
    }
    BackOff bo{ 1,100 };

    while (true) {
        // 변수 사용을 조심해서 할 것.
        // top을 직접적으로 사용하지 않고 변수에 담아서 사용해야
        // 해당 값이 결정된 상태로 CAS를 진행할 수 있음
        NODE* curr = top;
        e->next = curr;
        if (CAS_TOP(curr, e)) {
            return;
        }
        else {
            bo.InterruptedException();
        }
    }
}

int pop()
{
    BackOff bo{ 1, 100 };
    while (true) {
        // 변수 사용을 조심해서 할 것
        NODE* cur = top;

        if (cur == nullptr) {
            return -2;
        }

        NODE* next = cur->next;
        int retVal = cur->key;

        if (cur != top) {
            continue;
        }

        if (CAS_TOP(cur, next)) {
            // delete cur ... aba 문제 발생으로 인하여 주석처리
            return retVal;
        }
        else {
            bo.InterruptedException();
        }
    }
}
```



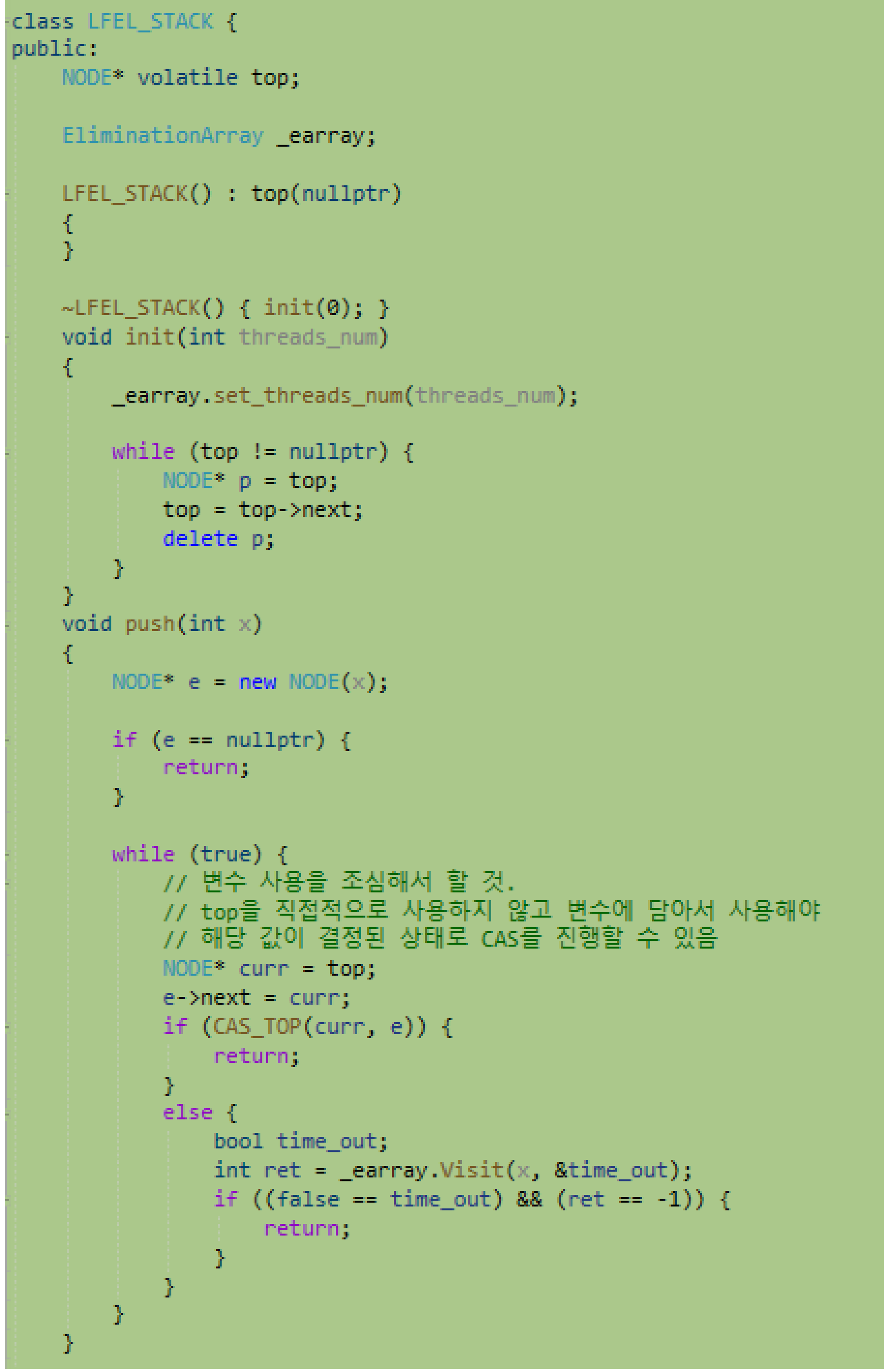
```
constexpr int EMPTY = 0; // 00
constexpr int WAITING = 0x40000000; // 01
constexpr int BUSY = 0x80000000; // 10

class LockFreeExchanger {
    atomic <unsigned int> slot;

public:
    LockFreeExchanger() { }
    ~LockFreeExchanger() {  }

    int exchange(int value, bool* time_out, bool* busy) {
        while (true) {
            unsigned int curr_slot = slot;
            unsigned int slot_state = curr_slot & 0xC0000000; // 앞의 두 비트만 빼내기
            unsigned int slot_value = curr_slot & 0x3FFFFFFF; // 나머지 비트 값 알아오기

            switch (slot_state) {
            case EMPTY: {
                unsigned int new_slot = WAITING & value;
                if (atomic_compare_exchange_strong(&slot, &curr_slot, new_slot))
                {
                    for (int i = 0; i < 10; ++i) {
                        if ((slot & 0xC0000000) == BUSY) {

                            int ret_value = slot & 0x3FFFFFFF;

                            slot = EMPTY;

                            return ret_value;
                        }
                    }
                    // 여기까지 오면 timeout
                    if (atomic_compare_exchange_strong(&slot, &new_slot, EMPTY)) {
                        *time_out = true;
                        *busy = false;
                        return 0;
                    }
                    else {
                        // 다른 스레드가 와서 busy로 만든 경우는 여기로옴
                        int ret_value = slot & 0x3FFFFFFF;

                        slot = EMPTY;

                        return ret_value;
                    }
                }
                else {
                    // 애초에 cas가 실패한경우
                    continue;
                }
            }
                break;
```



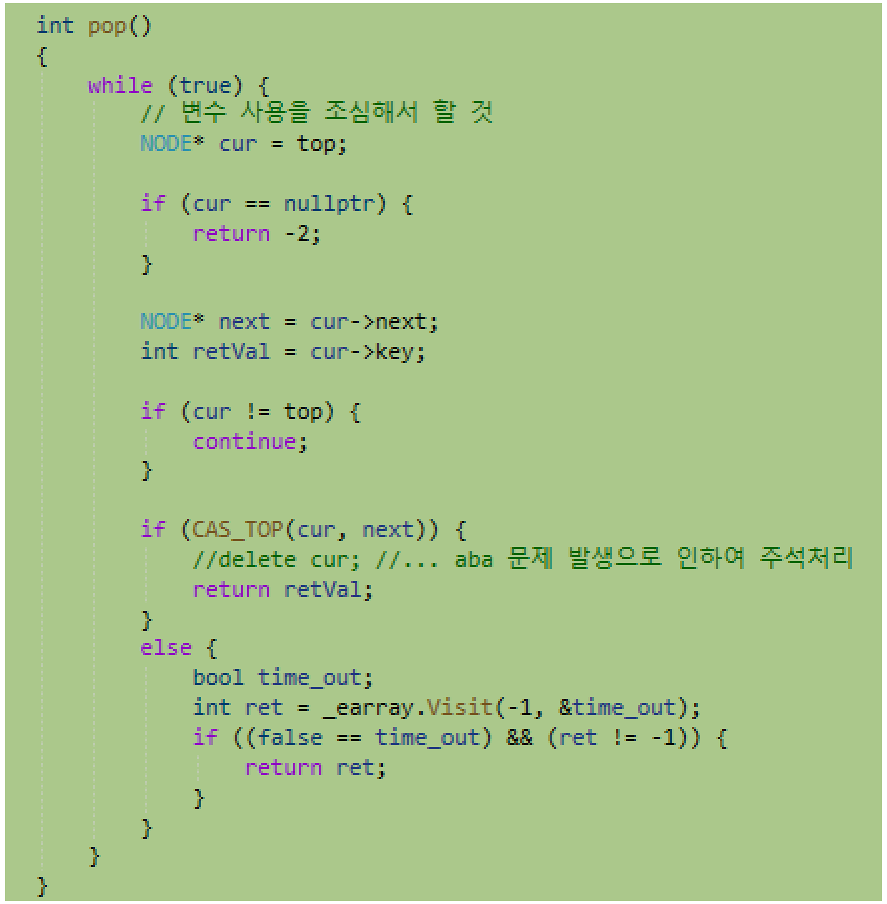
```
            case WAITING: {
                unsigned int new_value = BUSY | value;
                if (atomic_compare_exchange_strong(&slot, &curr_slot, new_value)) {
                    *time_out = false;
                    *busy = false;
                    return slot_value;
                }
                else {
                    // 실패를 했으면?
                    // 경합이 너무 심한 상태...
                    *time_out = false;
                    *busy = true;
                    return 0;
                }
            }
                break;
            case BUSY: {
                *time_out = false;
                *busy = true;
            }
                return 0;
                break;
            default:
                break;
            }
        }
```



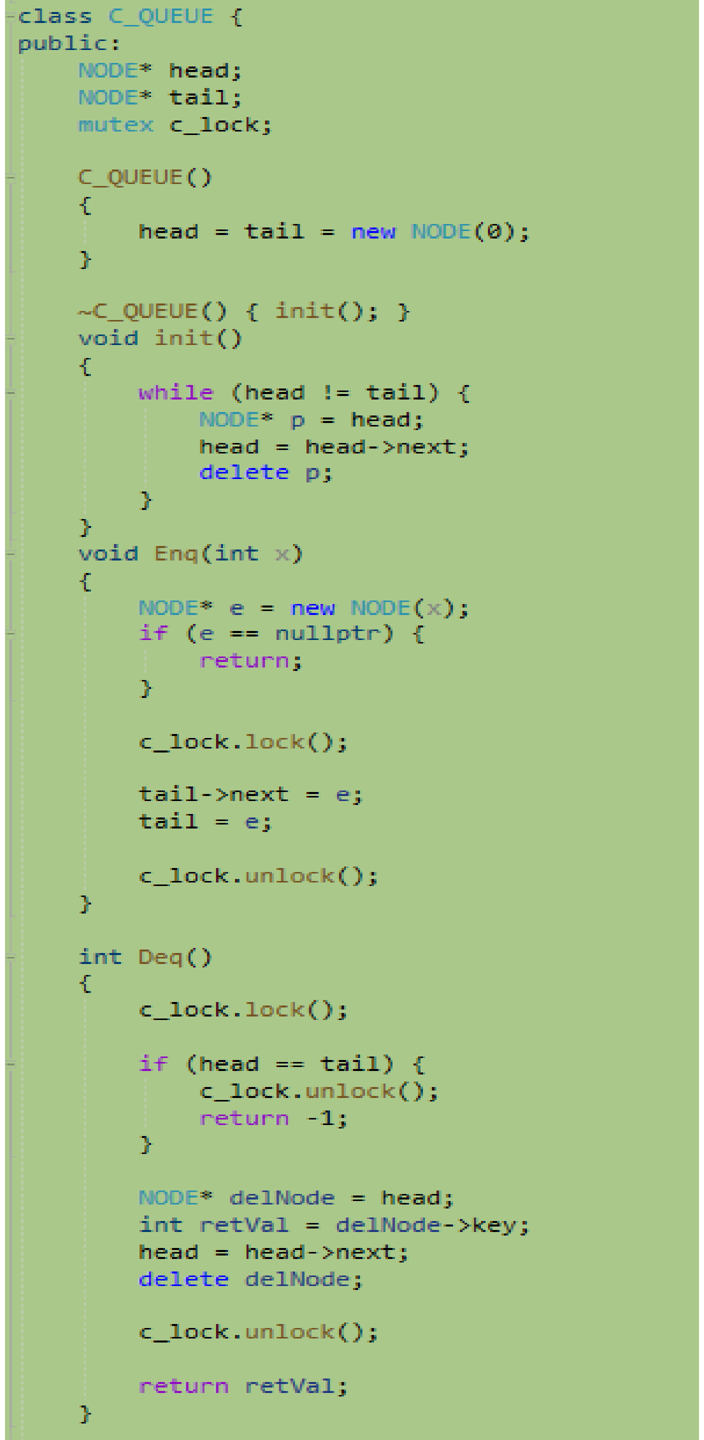
```
class EliminationArray {
     int range;
     int _num_threads;

    LockFreeExchanger exchanger[MAX_THREADS / 2];
public:
    EliminationArray() { range = 1; _num_threads = 1; }
    EliminationArray(int num_threads) :_num_threads(num_threads) { range = 1; }
    ~EliminationArray() {}
    int Visit(int value, bool* time_out) {
        int slot = rand() % range;
        bool busy;
        int ret = exchanger[slot].exchange(value, time_out, &busy);
        if ((true == *time_out) && (range > 1)) range--;
        if ((true == busy) && (range <= _num_threads / 2)) range++;
        // MAX RANGE is # of thread / 2
        return ret;
    }

    void set_threads_num(int num) { _num_threads = num; }
};
```



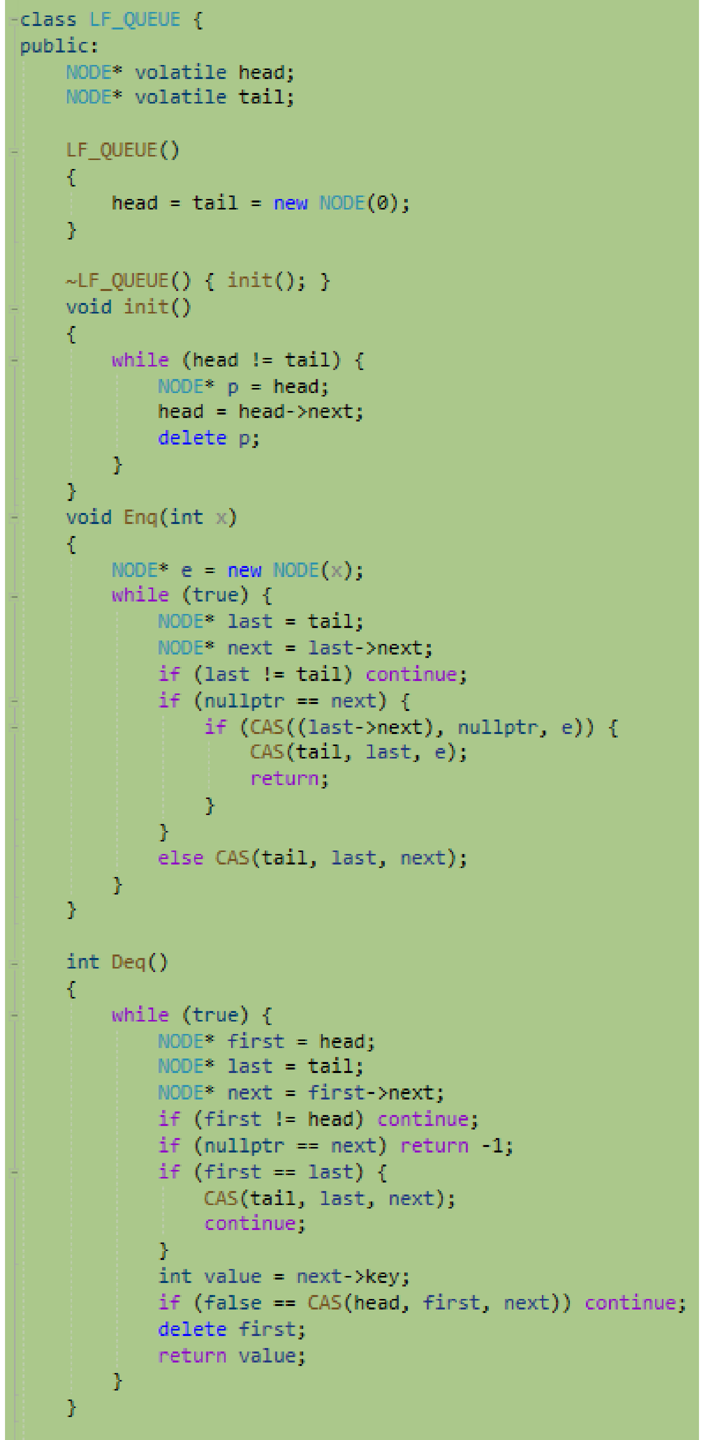
```
class LFEL_STACK {
public:
    NODE* volatile top;

    EliminationArray _earray;

    LFEL_STACK() : top(nullptr)
    {
    }

    ~LFEL_STACK() { init(0); }
    void init(int threads_num)
    {
        _earray.set_threads_num(threads_num);

        while (top != nullptr) {
            NODE* p = top;
            top = top->next;
            delete p;
        }
    }
    void push(int x)
    {
        NODE* e = new NODE(x);

        if (e == nullptr) {
            return;
        }

        while (true) {
            // 변수 사용을 조심해서 할 것.
            // top을 직접적으로 사용하지 않고 변수에 담아서 사용해야
            // 해당 값이 결정된 상태로 CAS를 진행할 수 있음
            NODE* curr = top;
            e->next = curr;
            if (CAS_TOP(curr, e)) {
                return;
            }
            else {
                bool time_out;
                int ret = _earray.Visit(x, &time_out);
                if ((false == time_out) && (ret == -1)) {
                    return;
                }
            }
        }
    }
```



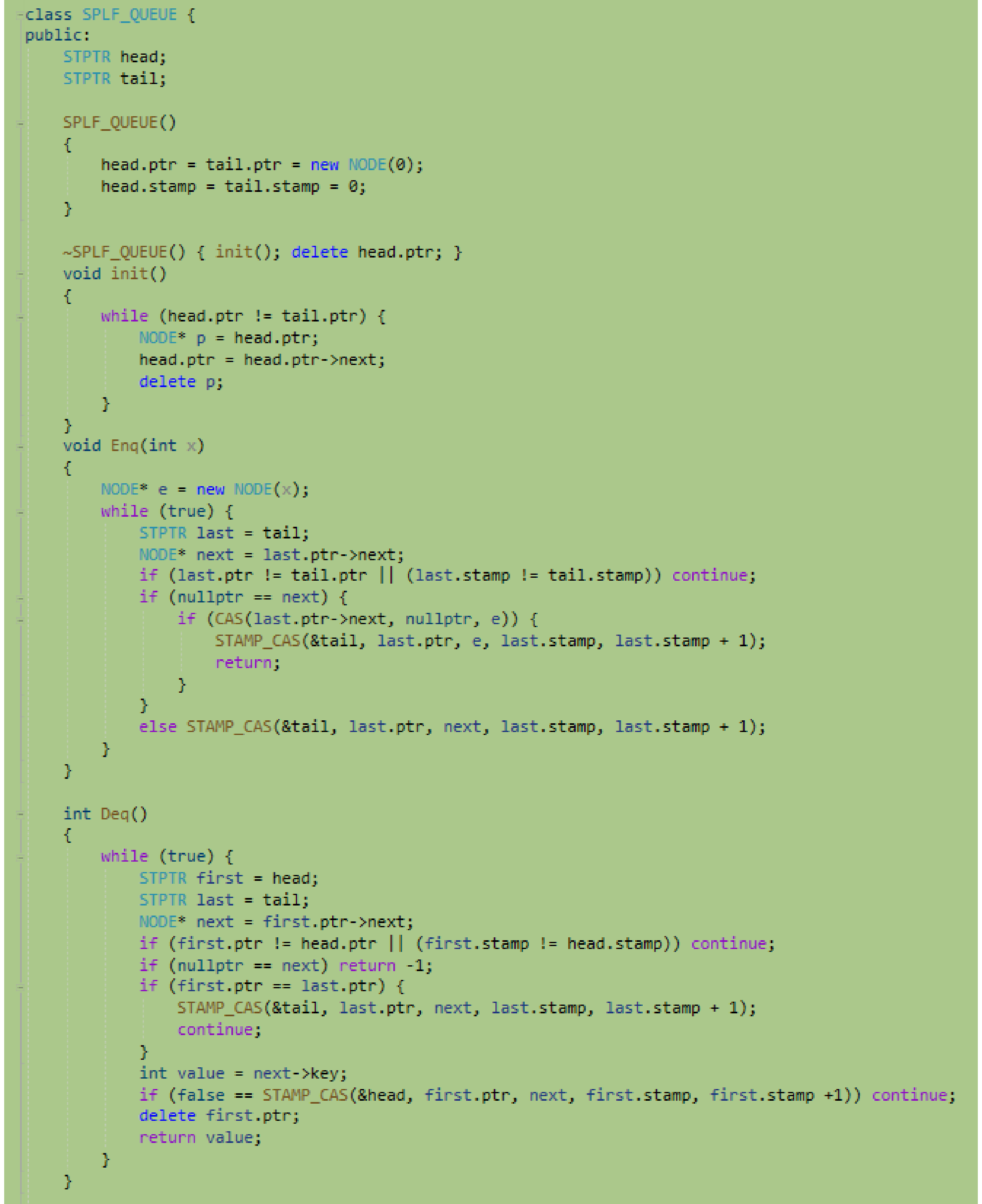
```
    int pop()
    {
        while (true) {
            // 변수 사용을 조심해서 할 것
            NODE* cur = top;

            if (cur == nullptr) {
                return -2;
            }

            NODE* next = cur->next;
            int retVal = cur->key;

            if (cur != top) {
                continue;
            }

            if (CAS_TOP(cur, next)) {
                //delete cur; //... aba 문제 발생으로 인하여 주석처리
                return retVal;
            }
            else {
                bool time_out;
                int ret = _earray.Visit(-1, &time_out);
                if ((false == time_out) && (ret != -1)) {
                    return ret;
                }
            }
        }
    }
```



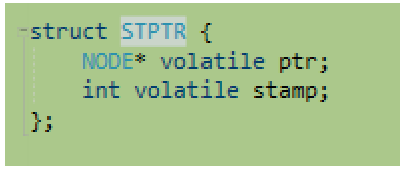
```
class C_QUEUE {
public:
    NODE* head;
    NODE* tail;
    mutex c_lock;

    C_QUEUE()
    {
        head = tail = new NODE(0);
    }

    ~C_QUEUE() { init(); }
    void init()
    {
        while (head != tail) {
            NODE* p = head;
            head = head->next;
            delete p;
        }
    }
    void Enq(int x)
    {
        NODE* e = new NODE(x);
        if (e == nullptr) {
            return;
        }

        c_lock.lock();

        tail->next = e;
        tail = e;

        c_lock.unlock();
    }

    int Deq()
    {
        c_lock.lock();

        if (head == tail) {
            c_lock.unlock();
            return -1;
        }

        NODE* delNode = head;
        int retVal = delNode->key;
        head = head->next;
        delete delNode;

        c_lock.unlock();

        return retVal;
    }
```



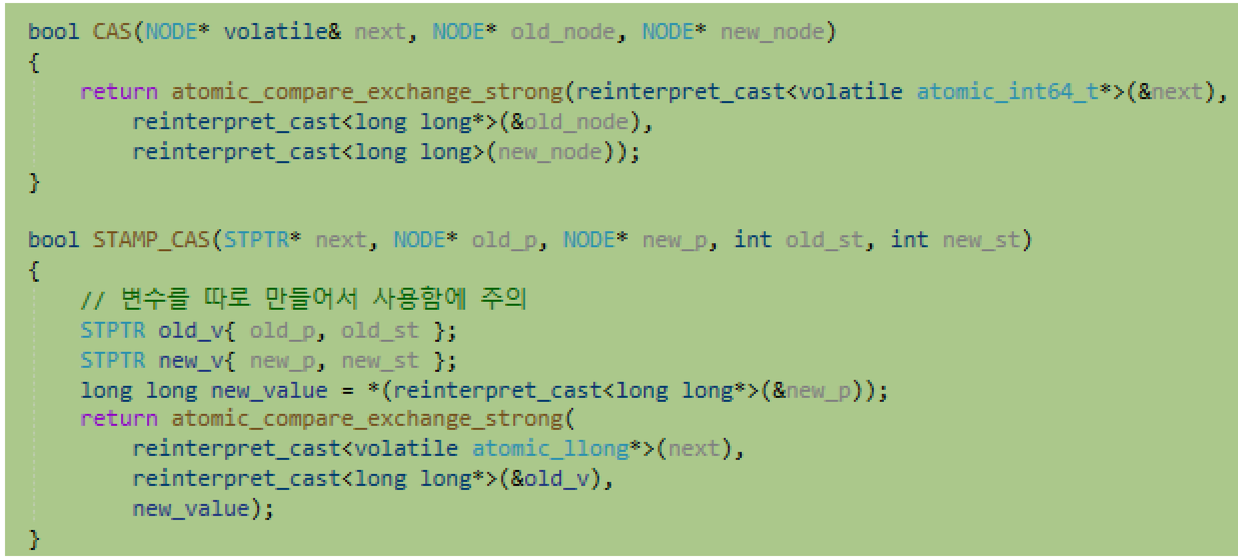
```
class LF_QUEUE {
public:
    NODE* volatile head;
    NODE* volatile tail;

    LF_QUEUE()
    {
        head = tail = new NODE(0);
    }

    ~LF_QUEUE() { init(); }
    void init()
    {
        while (head != tail) {
            NODE* p = head;
            head = head->next;
            delete p;
        }
    }
    void Enq(int x)
    {
        NODE* e = new NODE(x);
        while (true) {
            NODE* last = tail;
            NODE* next = last->next;
            if (last != tail) continue;
            if (nullptr == next) {
                if (CAS((last->next), nullptr, e)) {
                    CAS(tail, last, e);
                    return;
                }
            }
            else CAS(tail, last, next);
        }
    }

    int Deq()
    {
        while (true) {
            NODE* first = head;
            NODE* last = tail;
            NODE* next = first->next;
            if (first != head) continue;
            if (nullptr == next) return -1;
            if (first == last) {
                CAS(tail, last, next);
                continue;
            }
            int value = next->key;
            if (false == CAS(head, first, next)) continue;
            delete first;
            return value;
        }
    }
```



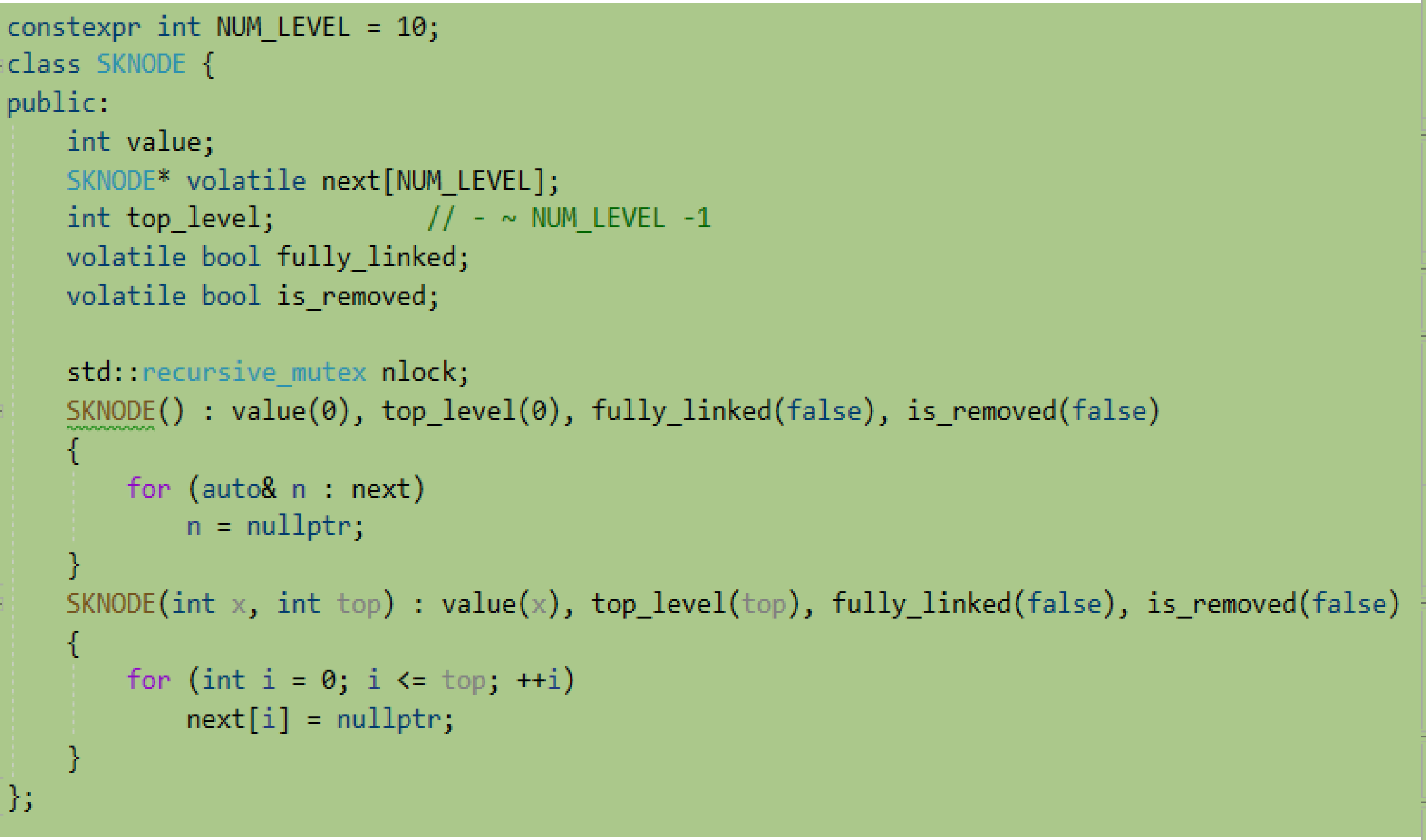
```
class SPLF_QUEUE {
public:
    STPTR head;
    STPTR tail;

    SPLF_QUEUE()
    {
        head.ptr = tail.ptr = new NODE(0);
        head.stamp = tail.stamp = 0;
    }

    ~SPLF_QUEUE() { init(); delete head.ptr; }
    void init()
    {
        while (head.ptr != tail.ptr) {
            NODE* p = head.ptr;
            head.ptr = head.ptr->next;
            delete p;
        }
    }
    void Enq(int x)
    {
        NODE* e = new NODE(x);
        while (true) {
            STPTR last = tail;
            NODE* next = last.ptr->next;
            if (last.ptr != tail.ptr || (last.stamp != tail.stamp)) continue;
            if (nullptr == next) {
                if (CAS(last.ptr->next, nullptr, e)) {
                    STAMP_CAS(&tail, last.ptr, e, last.stamp, last.stamp + 1);
                    return;
                }
            }
            else STAMP_CAS(&tail, last.ptr, next, last.stamp, last.stamp + 1);
        }
    }

    int Deq()
    {
        while (true) {
            STPTR first = head;
            STPTR last = tail;
            NODE* next = first.ptr->next;
            if (first.ptr != head.ptr || (first.stamp != head.stamp)) continue;
            if (nullptr == next) return -1;
            if (first.ptr == last.ptr) {
                STAMP_CAS(&tail, last.ptr, next, last.stamp, last.stamp + 1);
                continue;
            }
            int value = next->key;
            if (false == STAMP_CAS(&head, first.ptr, next, first.stamp, first.stamp +1)) continue;
            delete first.ptr;
            return value;
        }
    }
```



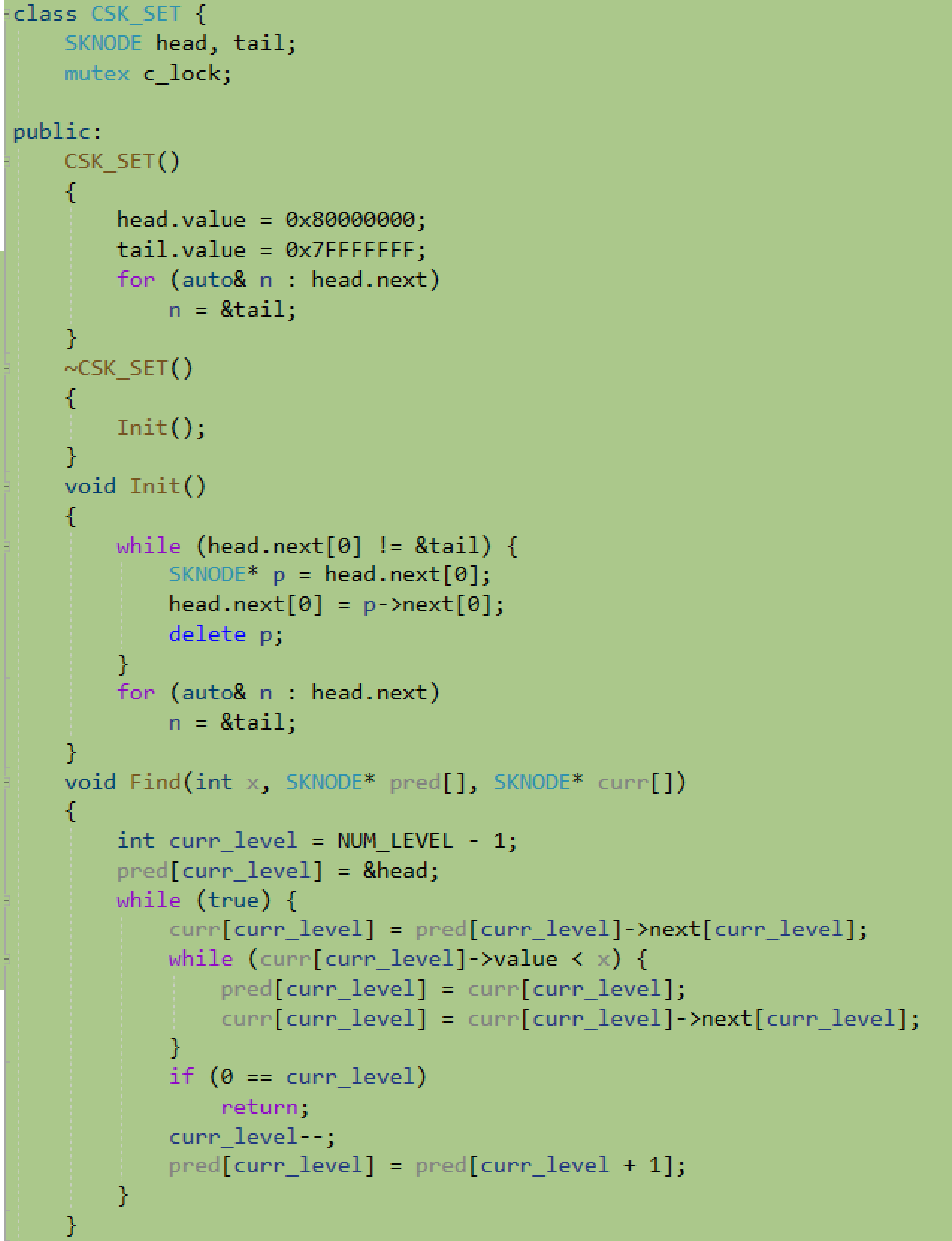
```
struct STPTR {
    NODE* volatile ptr;
    int volatile stamp;
};
```



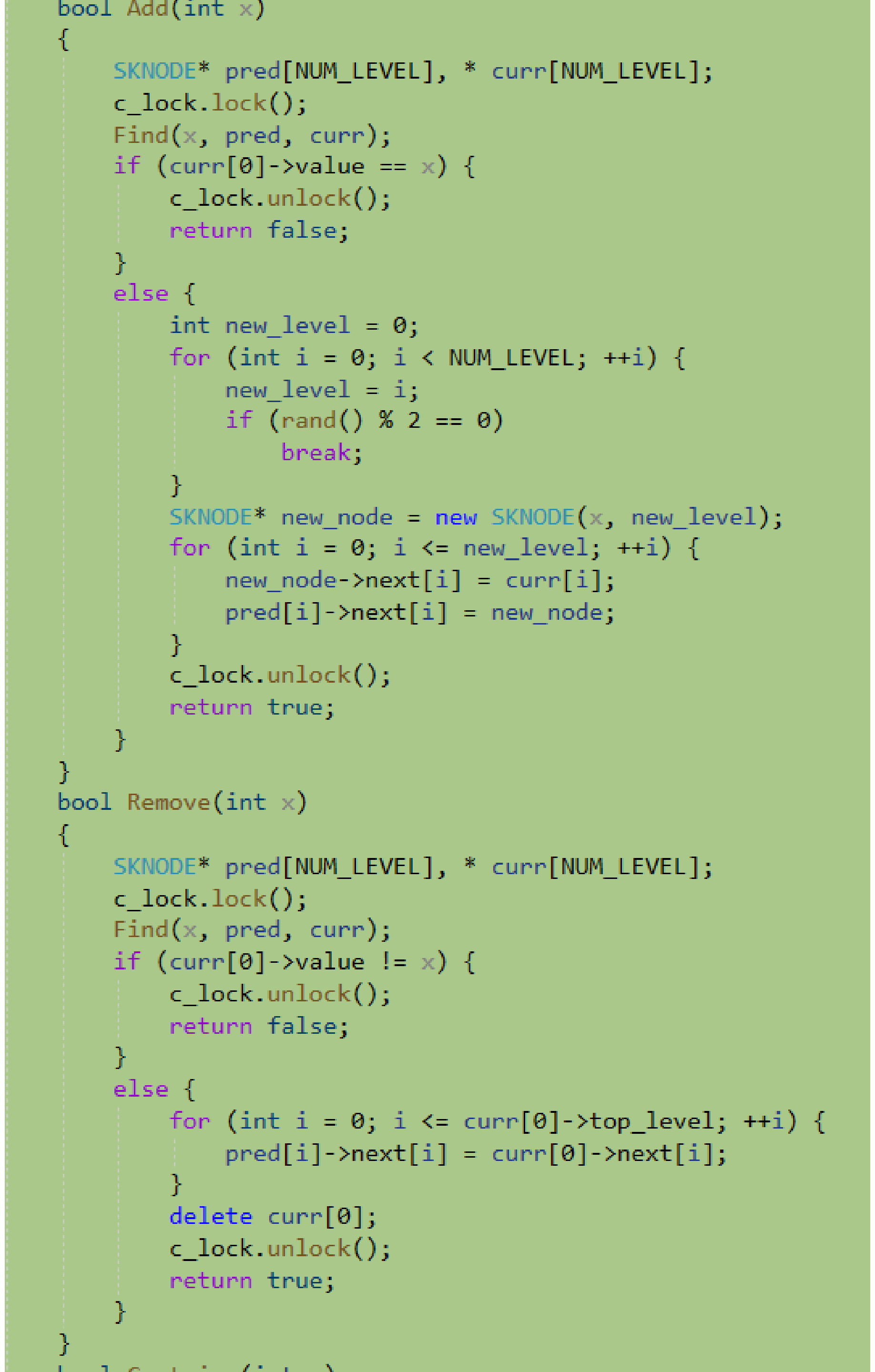
```
bool CAS(NODE* volatile& next, NODE* old_node, NODE* new_node)
{
    return atomic_compare_exchange_strong(reinterpret_cast<volatile atomic_int64_t*>(&next),
        reinterpret_cast<long long*>(&old_node),
        reinterpret_cast<long long>(new_node));
}

bool STAMP_CAS(STPTR* next, NODE* old_p, NODE* new_p, int old_st, int new_st)
{
    // 변수를 따로 만들어서 사용함에 주의
    STPTR old_v{ old_p, old_st };
    STPTR new_v{ new_p, new_st };
    long long new_value = *(reinterpret_cast<long long*>(&new_p));
    return atomic_compare_exchange_strong(
        reinterpret_cast<volatile atomic_llong*>(next),
        reinterpret_cast<long long*>(&old_v),
        new_value);
}
```



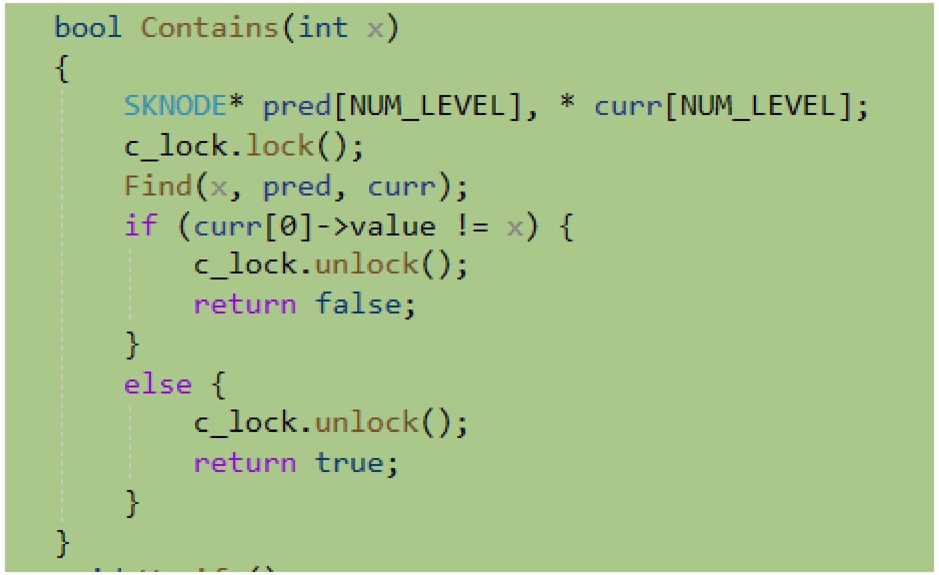
```cpp
constexpr int NUM_LEVEL = 10;
class SKNODE {
public:
    int value;
    SKNODE* volatile next[NUM_LEVEL];
    int top_level;          // - ~ NUM_LEVEL -1
    volatile bool fully_linked;
    volatile bool is_removed;

    std::recursive_mutex nlock;
    SKNODE() : value(0), top_level(0), fully_linked(false), is_removed(false)
    {
        for (auto& n : next)
            n = nullptr;
    }
    SKNODE(int x, int top) : value(x), top_level(top), fully_linked(false), is_removed(false)
    {
        for (int i = 0; i <= top; ++i)
            next[i] = nullptr;
    }
};
```



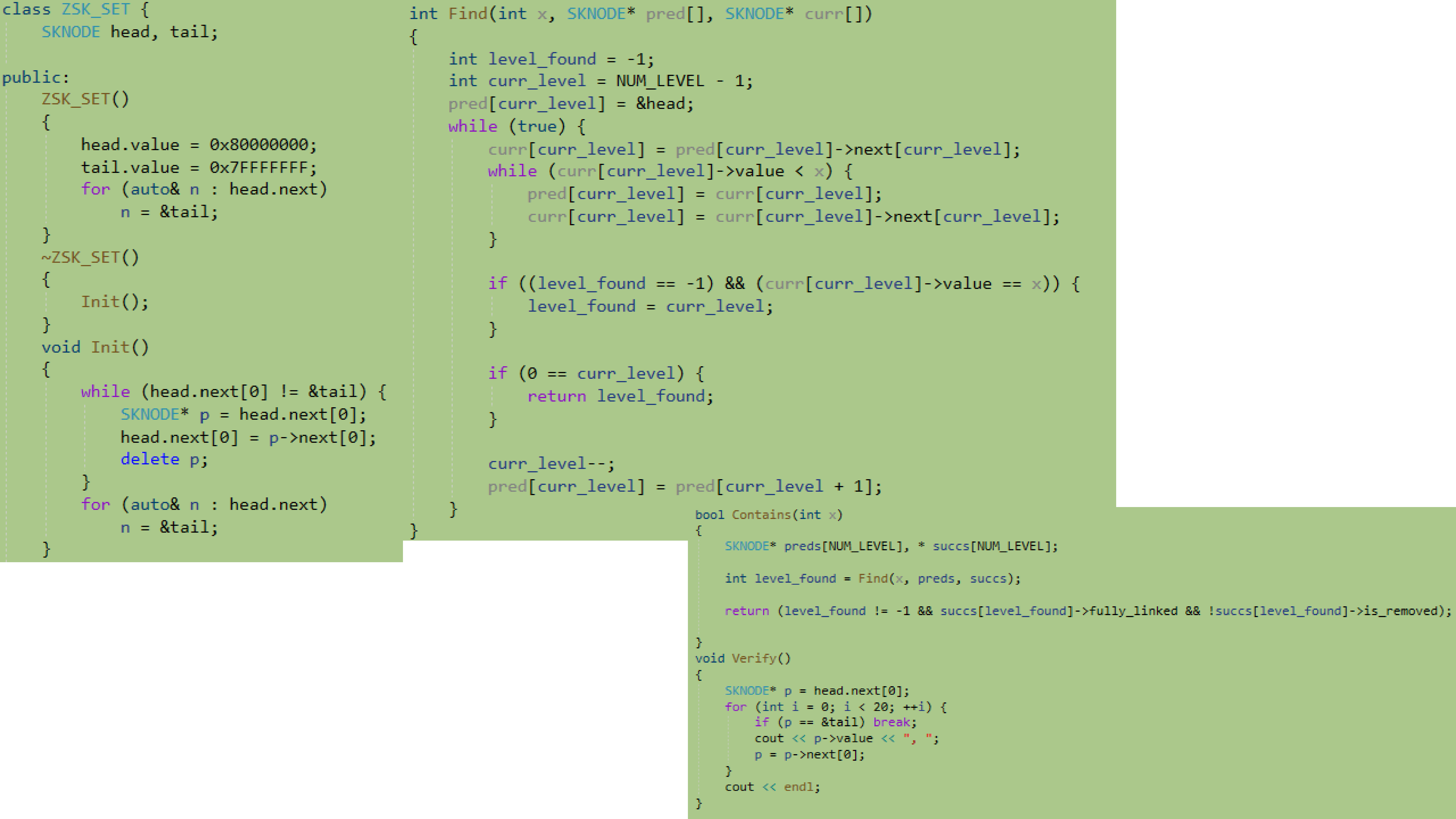
```cpp
class CSK_SET {
    SKNODE head, tail;
    mutex c_lock;

public:
    CSK_SET()
    {
        head.value = 0x80000000;
        tail.value = 0x7FFFFFFF;
        for (auto& n : head.next)
            n = &tail;
    }
    ~CSK_SET()
    {
        Init();
    }
    void Init()
    {
        while (head.next[0] != &tail) {
            SKNODE* p = head.next[0];
            head.next[0] = p->next[0];
            delete p;
        }
        for (auto& n : head.next)
            n = &tail;
    }
    void Find(int x, SKNODE* pred[], SKNODE* curr[])
    {
        int curr_level = NUM_LEVEL - 1;
        pred[curr_level] = &head;
        while (true) {
            curr[curr_level] = pred[curr_level]->next[curr_level];
            while (curr[curr_level]->value < x) {
                pred[curr_level] = curr[curr_level];
                curr[curr_level] = curr[curr_level]->next[curr_level];
            }
            if (0 == curr_level)
                return;
            curr_level--;
            pred[curr_level] = pred[curr_level + 1];
        }
    }
```



```
bool Add(int x)
{
    SKNODE* pred[NUM_LEVEL], * curr[NUM_LEVEL];
    c_lock.lock();
    Find(x, pred, curr);
    if (curr[0]->value == x) {
        c_lock.unlock();
        return false;
    }
    else {
        int new_level = 0;
        for (int i = 0; i < NUM_LEVEL; ++i) {
            new_level = i;
            if (rand() % 2 == 0)
                break;
        }
        SKNODE* new_node = new SKNODE(x, new_level);
        for (int i = 0; i <= new_level; ++i) {
            new_node->next[i] = curr[i];
            pred[i]->next[i] = new_node;
        }
        c_lock.unlock();
        return true;
    }
}
bool Remove(int x)
{
    SKNODE* pred[NUM_LEVEL], * curr[NUM_LEVEL];
    c_lock.lock();
    Find(x, pred, curr);
    if (curr[0]->value != x) {
        c_lock.unlock();
        return false;
    }
    else {
        for (int i = 0; i <= curr[0]->top_level; ++i) {
            pred[i]->next[i] = curr[0]->next[i];
        }
        delete curr[0];
        c_lock.unlock();
        return true;
    }
}
bool Contains(int x)
```



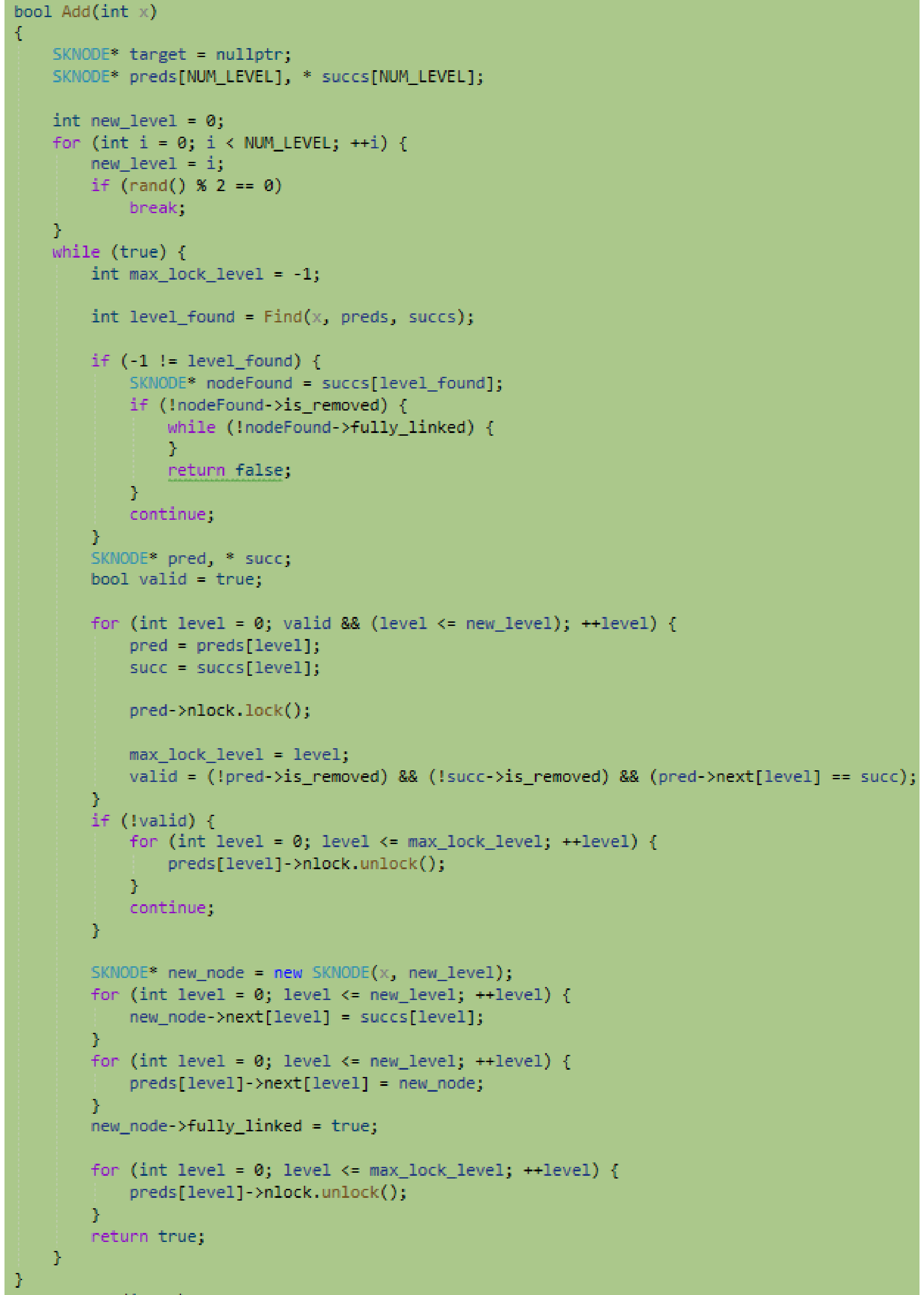
```
bool Contains(int x)
{
    SKNODE* pred[NUM_LEVEL], * curr[NUM_LEVEL];
    c_lock.lock();
    Find(x, pred, curr);
    if (curr[0]->value != x) {
        c_lock.unlock();
        return false;
    }
    else {
        c_lock.unlock();
        return true;
    }
}
```



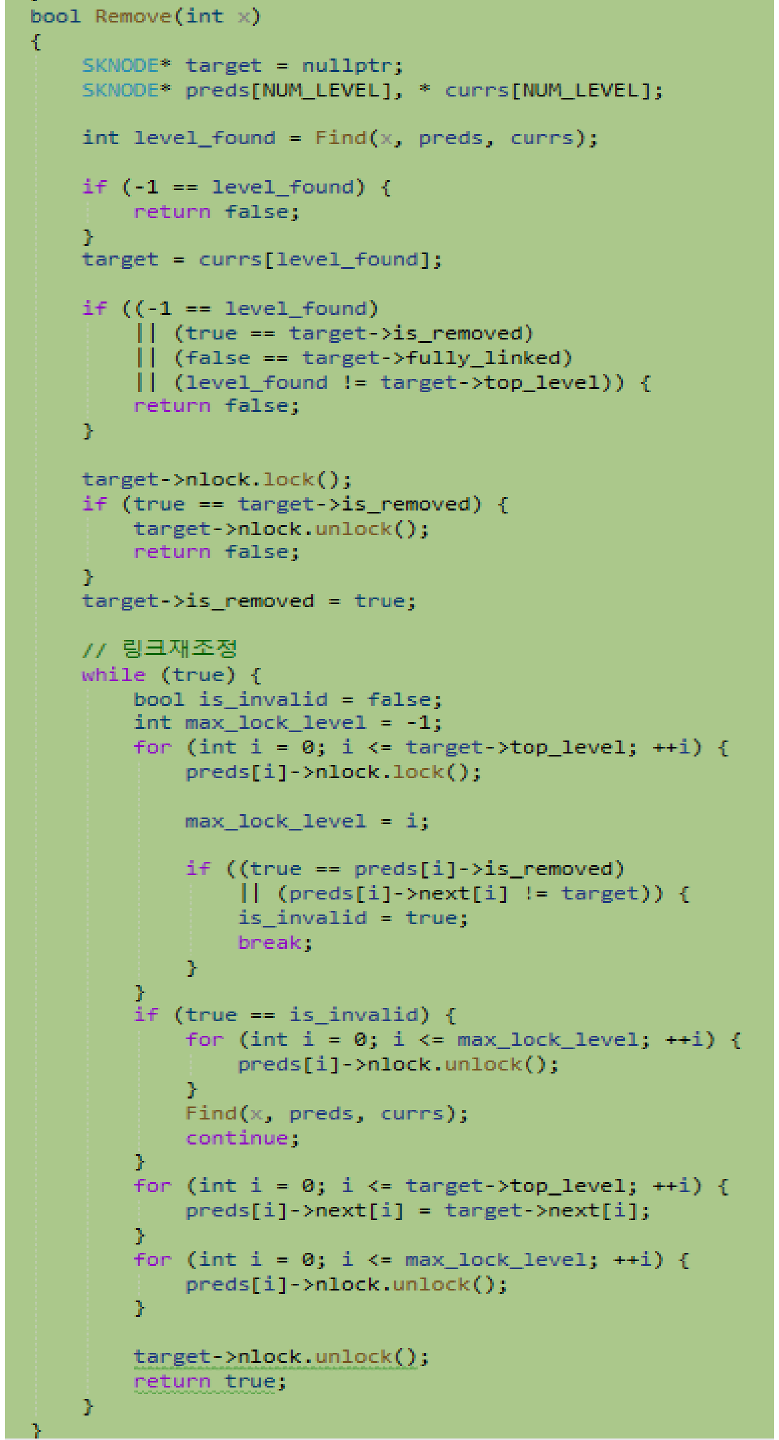
```
class ZSK_SET {
    SKNODE head, tail;

public:
    ZSK_SET()
    {
        head.value = 0x80000000;
        tail.value = 0x7FFFFFFF;
        for (auto& n : head.next)
            n = &tail;
    }
    ~ZSK_SET()
    {
        Init();
    }
    void Init()
    {
        while (head.next[0] != &tail) {
            SKNODE* p = head.next[0];
            head.next[0] = p->next[0];
            delete p;
        }
        for (auto& n : head.next)
            n = &tail;
    }
```

```
int Find(int x, SKNODE* pred[], SKNODE* curr[])
{
    int level_found = -1;
    int curr_level = NUM_LEVEL - 1;
    pred[curr_level] = &head;
    while (true) {
        curr[curr_level] = pred[curr_level]->next[curr_level];
        while (curr[curr_level]->value < x) {
            pred[curr_level] = curr[curr_level];
            curr[curr_level] = curr[curr_level]->next[curr_level];
        }

        if ((level_found == -1) && (curr[curr_level]->value == x)) {
            level_found = curr_level;
        }

        if (0 == curr_level) {
            return level_found;
        }

        curr_level--;
        pred[curr_level] = pred[curr_level + 1];
    }
}
```

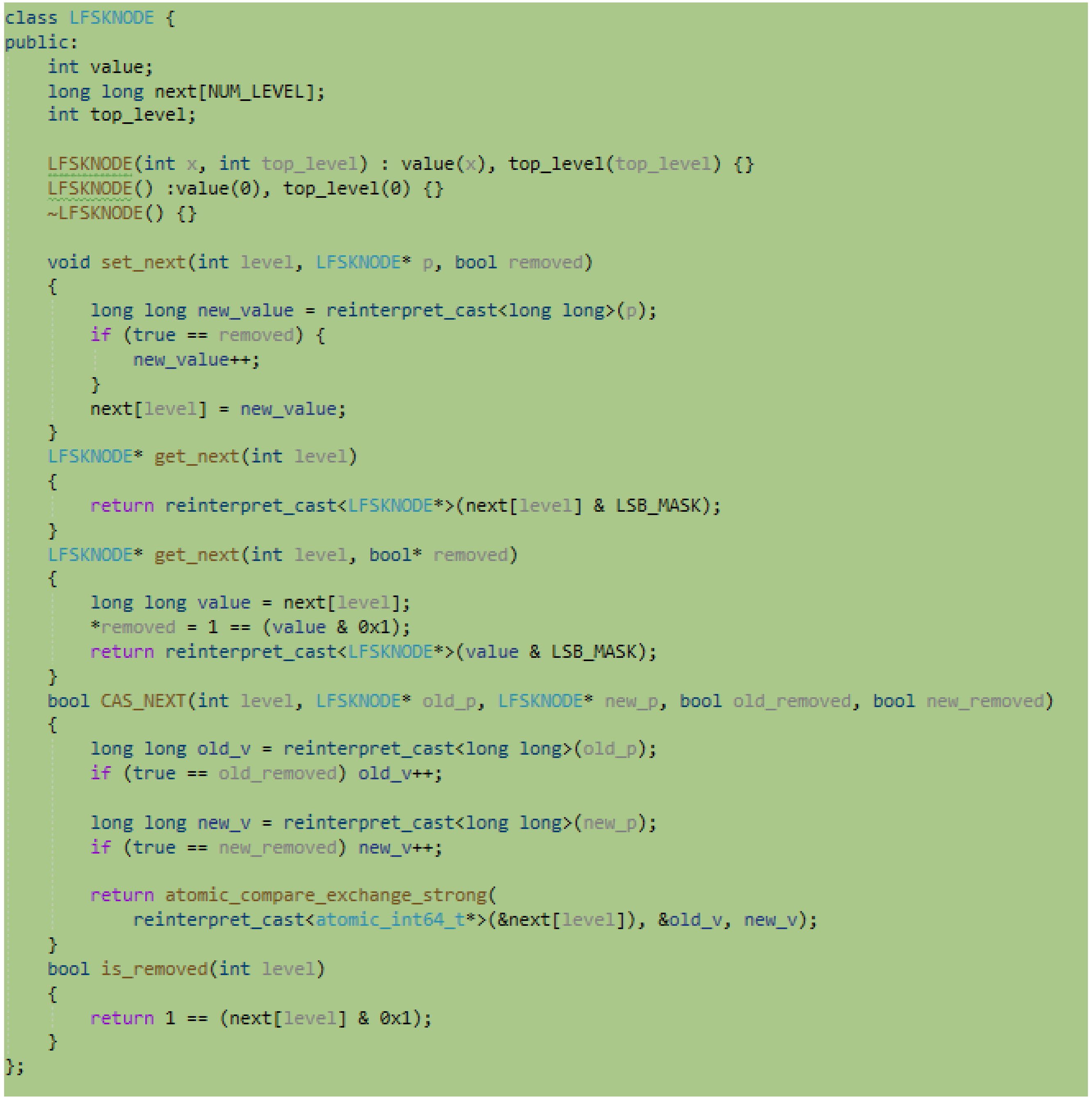
```
bool Contains(int x)
{
    SKNODE* preds[NUM_LEVEL], * succs[NUM_LEVEL];

    int level_found = Find(x, preds, succs);

    return (level_found != -1 && succs[level_found]->fully_linked && !succs[level_found]->is_removed);

}
void Verify()
{
    SKNODE* p = head.next[0];
    for (int i = 0; i < 20; ++i) {
        if (p == &tail) break;
        cout << p->value << ", ";
        p = p->next[0];
    }
    cout << endl;
}
```



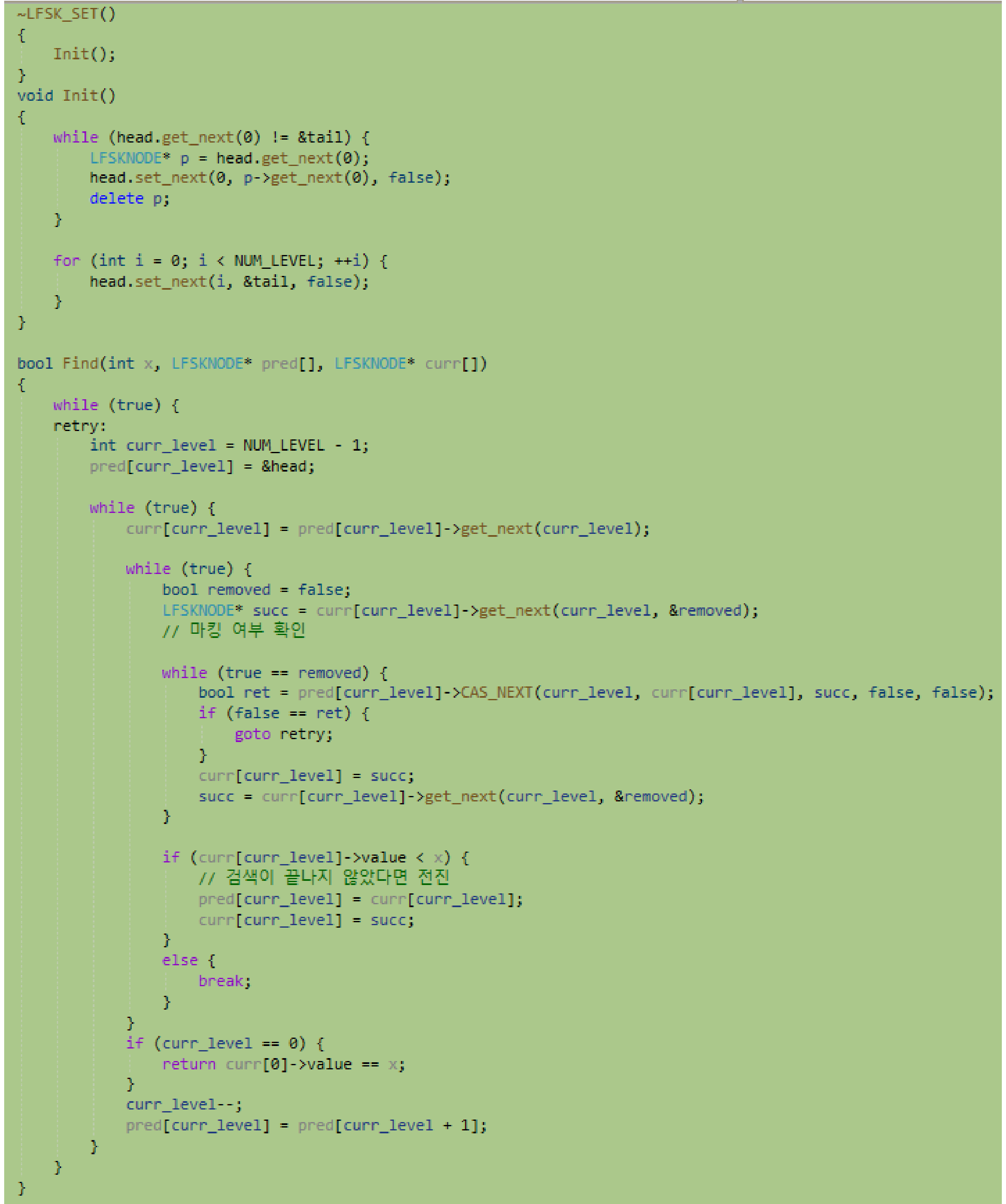
```
bool Add(int x)
{
    SKNODE* target = nullptr;
    SKNODE* preds[NUM_LEVEL], * succs[NUM_LEVEL];

    int new_level = 0;
    for (int i = 0; i < NUM_LEVEL; ++i) {
        new_level = i;
        if (rand() % 2 == 0)
            break;
    }
    while (true) {
        int max_lock_level = -1;

        int level_found = Find(x, preds, succs);

        if (-1 != level_found) {
            SKNODE* nodeFound = succs[level_found];
            if (!nodeFound->is_removed) {
                while (!nodeFound->fully_linked) {
                }
                return false;
            }
            continue;
        }
        SKNODE* pred, * succ;
        bool valid = true;

        for (int level = 0; valid && (level <= new_level); ++level) {
            pred = preds[level];
            succ = succs[level];

            pred->nlock.lock();

            max_lock_level = level;
            valid = (!pred->is_removed) && (!succ->is_removed) && (pred->next[level] == succ);
        }
        if (!valid) {
            for (int level = 0; level <= max_lock_level; ++level) {
                preds[level]->nlock.unlock();
            }
            continue;
        }

        SKNODE* new_node = new SKNODE(x, new_level);
        for (int level = 0; level <= new_level; ++level) {
            new_node->next[level] = succs[level];
        }
        for (int level = 0; level <= new_level; ++level) {
            preds[level]->next[level] = new_node;
        }
        new_node->fully_linked = true;

        for (int level = 0; level <= max_lock_level; ++level) {
            preds[level]->nlock.unlock();
        }
        return true;
    }
}
```



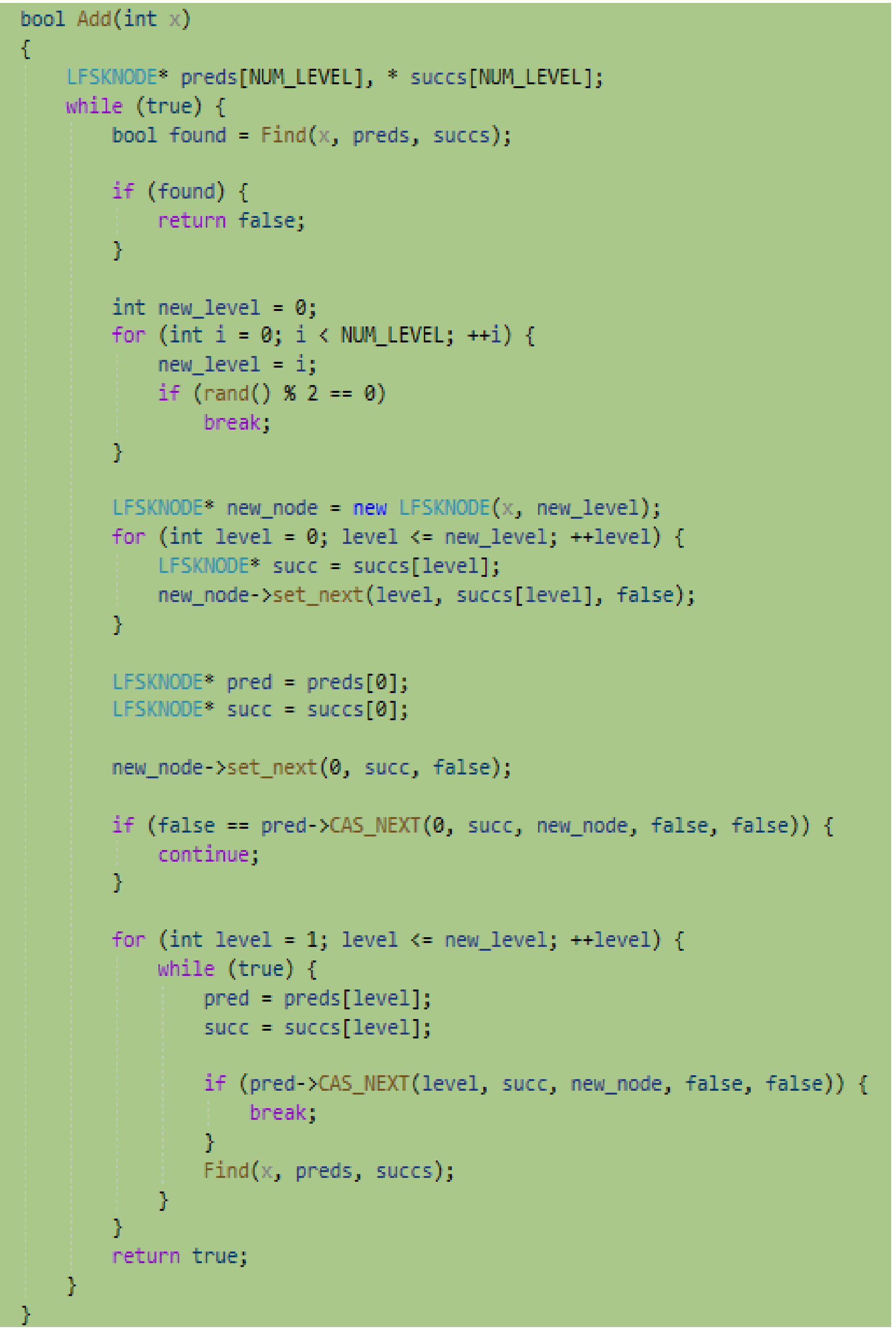
```
bool Remove(int x)
{
    SKNODE* target = nullptr;
    SKNODE* preds[NUM_LEVEL], * currs[NUM_LEVEL];

    int level_found = Find(x, preds, currs);

    if (-1 == level_found) {
        return false;
    }
    target = currs[level_found];

    if ((-1 == level_found)
        || (true == target->is_removed)
        || (false == target->fully_linked)
        || (level_found != target->top_level)) {
        return false;
    }

    target->nlock.lock();
    if (true == target->is_removed) {
        target->nlock.unlock();
        return false;
    }
    target->is_removed = true;

    // 링크재조정
    while (true) {
        bool is_invalid = false;
        int max_lock_level = -1;
        for (int i = 0; i <= target->top_level; ++i) {
            preds[i]->nlock.lock();

            max_lock_level = i;

            if ((true == preds[i]->is_removed)
                || (preds[i]->next[i] != target)) {
                is_invalid = true;
                break;
            }
        }
        if (true == is_invalid) {
            for (int i = 0; i <= max_lock_level; ++i) {
                preds[i]->nlock.unlock();
            }
            Find(x, preds, currs);
            continue;
        }
        for (int i = 0; i <= target->top_level; ++i) {
            preds[i]->next[i] = target->next[i];
        }
        for (int i = 0; i <= max_lock_level; ++i) {
            preds[i]->nlock.unlock();
        }

        target->nlock.unlock();
        return true;
    }
}
```



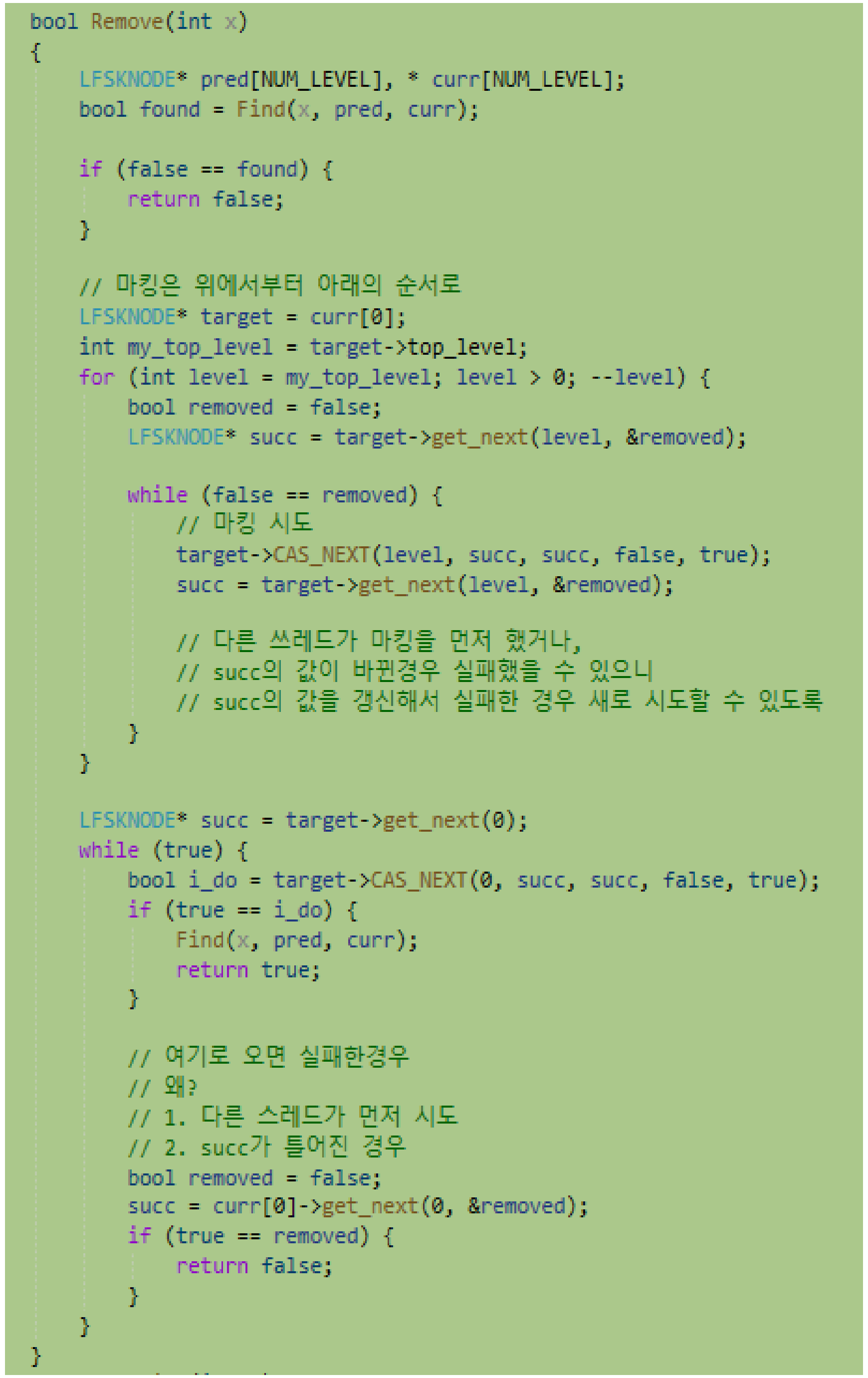
```
class LFSKNODE {
public:
    int value;
    long long next[NUM_LEVEL];
    int top_level;

    LFSKNODE(int x, int top_level) : value(x), top_level(top_level) {}
    LFSKNODE() :value(0), top_level(0) {}
    ~LFSKNODE() {}

    void set_next(int level, LFSKNODE* p, bool removed)
    {
        long long new_value = reinterpret_cast<long long>(p);
        if (true == removed) {
            new_value++;
        }
        next[level] = new_value;
    }
    LFSKNODE* get_next(int level)
    {
        return reinterpret_cast<LFSKNODE*>(next[level] & LSB_MASK);
    }
    LFSKNODE* get_next(int level, bool* removed)
    {
        long long value = next[level];
        *removed = 1 == (value & 0x1);
        return reinterpret_cast<LFSKNODE*>(value & LSB_MASK);
    }
    bool CAS_NEXT(int level, LFSKNODE* old_p, LFSKNODE* new_p, bool old_removed, bool new_removed)
    {
        long long old_v = reinterpret_cast<long long>(old_p);
        if (true == old_removed) old_v++;

        long long new_v = reinterpret_cast<long long>(new_p);
        if (true == new_removed) new_v++;

        return atomic_compare_exchange_strong(
            reinterpret_cast<atomic_int64_t*>(&next[level]), &old_v, new_v);
    }
    bool is_removed(int level)
    {
        return 1 == (next[level] & 0x1);
    }
};
```



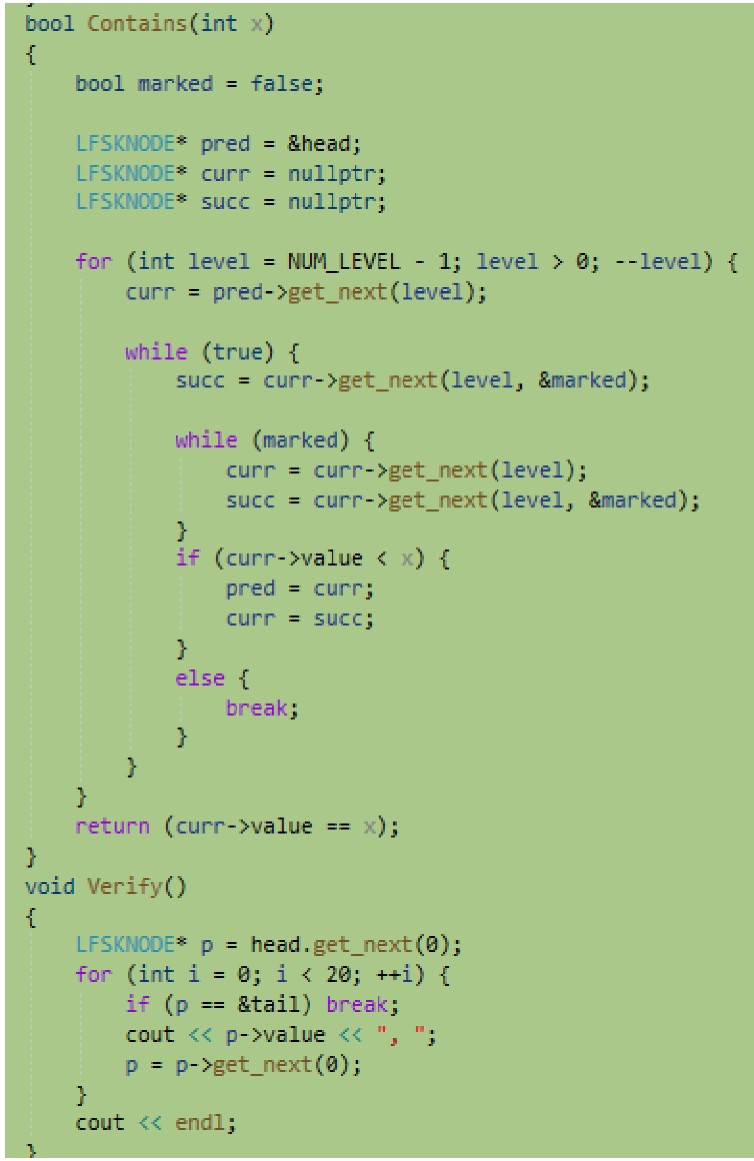
```
~LFSK_SET()
{
    Init();
}
void Init()
{
    while (head.get_next(0) != &tail) {
        LFSKNODE* p = head.get_next(0);
        head.set_next(0, p->get_next(0), false);
        delete p;
    }

    for (int i = 0; i < NUM_LEVEL; ++i) {
        head.set_next(i, &tail, false);
    }
}

bool Find(int x, LFSKNODE* pred[], LFSKNODE* curr[])
{
    while (true) {
    retry:
        int curr_level = NUM_LEVEL - 1;
        pred[curr_level] = &head;

        while (true) {
            curr[curr_level] = pred[curr_level]->get_next(curr_level);

            while (true) {
                bool removed = false;
                LFSKNODE* succ = curr[curr_level]->get_next(curr_level, &removed);
                // 마킹 여부 확인

                while (true == removed) {
                    bool ret = pred[curr_level]->CAS_NEXT(curr_level, curr[curr_level], succ, false, false);
                    if (false == ret) {
                        goto retry;
                    }
                    curr[curr_level] = succ;
                    succ = curr[curr_level]->get_next(curr_level, &removed);
                }

                if (curr[curr_level]->value < x) {
                    // 검색이 끝나지 않았다면 전진
                    pred[curr_level] = curr[curr_level];
                    curr[curr_level] = succ;
                }
                else {
                    break;
                }
            }
            if (curr_level == 0) {
                return curr[0]->value == x;
            }
            curr_level--;
            pred[curr_level] = pred[curr_level + 1];
        }
    }
}
```



```cpp
bool Add(int x)
{
    LFSKNODE* preds[NUM_LEVEL], * succs[NUM_LEVEL];
    while (true) {
        bool found = Find(x, preds, succs);

        if (found) {
            return false;
        }

        int new_level = 0;
        for (int i = 0; i < NUM_LEVEL; ++i) {
            new_level = i;
            if (rand() % 2 == 0)
                break;
        }

        LFSKNODE* new_node = new LFSKNODE(x, new_level);
        for (int level = 0; level <= new_level; ++level) {
            LFSKNODE* succ = succs[level];
            new_node->set_next(level, succs[level], false);
        }

        LFSKNODE* pred = preds[0];
        LFSKNODE* succ = succs[0];

        new_node->set_next(0, succ, false);

        if (false == pred->CAS_NEXT(0, succ, new_node, false, false)) {
            continue;
        }

        for (int level = 1; level <= new_level; ++level) {
            while (true) {
                pred = preds[level];
                succ = succs[level];

                if (pred->CAS_NEXT(level, succ, new_node, false, false)) {
                    break;
                }
                Find(x, preds, succs);
            }
        }
        return true;
    }
}
```



```cpp
bool Remove(int x)
{
    LFSKNODE* pred[NUM_LEVEL], * curr[NUM_LEVEL];
    bool found = Find(x, pred, curr);

    if (false == found) {
        return false;
    }

    // 마킹은 위에서부터 아래의 순서로
    LFSKNODE* target = curr[0];
    int my_top_level = target->top_level;
    for (int level = my_top_level; level > 0; --level) {
        bool removed = false;
        LFSKNODE* succ = target->get_next(level, &removed);

        while (false == removed) {
            // 마킹 시도
            target->CAS_NEXT(level, succ, succ, false, true);
            succ = target->get_next(level, &removed);

            // 다른 쓰레드가 마킹을 먼저 했거나,
            // succ의 값이 바뀐경우 실패했을 수 있으니
            // succ의 값을 갱신해서 실패한 경우 새로 시도할 수 있도록
        }
    }

    LFSKNODE* succ = target->get_next(0);
    while (true) {
        bool i_do = target->CAS_NEXT(0, succ, succ, false, true);
        if (true == i_do) {
            Find(x, pred, curr);
            return true;
        }

        // 여기로 오면 실패한경우
        // 왜?
        // 1. 다른 스레드가 먼저 시도
        // 2. succ가 틀어진 경우
        bool removed = false;
        succ = curr[0]->get_next(0, &removed);
        if (true == removed) {
            return false;
        }
    }
}
```



```cpp
bool Contains(int x)
{
    bool marked = false;

    LFSKNODE* pred = &head;
    LFSKNODE* curr = nullptr;
    LFSKNODE* succ = nullptr;

    for (int level = NUM_LEVEL - 1; level > 0; --level) {
        curr = pred->get_next(level);

        while (true) {
            succ = curr->get_next(level, &marked);

            while (marked) {
                curr = curr->get_next(level);
                succ = curr->get_next(level, &marked);
            }
            if (curr->value < x) {
                pred = curr;
                curr = succ;
            }
            else {
                break;
            }
        }
    }
    return (curr->value == x);
}
void Verify()
{
    LFSKNODE* p = head.get_next(0);
    for (int i = 0; i < 20; ++i) {
        if (p == &tail) break;
        cout << p->value << ", ";
        p = p->get_next(0);
    }
    cout << endl;
}
```