

# 7. 병렬 알고리즘 – STACK SKIP-LIST

멀티쓰레드 프로그래밍  
정내훈

# 목차

---

- Stack

- SKIPLIST

# 목차

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
  - 무잠금 교환자
  - 소거 배열

# 소개

---

- Stack
  - 후입선출(LIFO) 구조
  - Push(), Pop() 메서드를 제공

# 무제한 성긴 동기화 스택

- 무제한 성긴 동기화 스택은 연결리스트로 구성되고 top 필드가 첫 노드를 가리킨다.
  - 만약 스택이 비어있을 경우는 nullptr
  - -1을 스택에 추가하는 것은 고려하지 않는다.
  - Empty일 경우 Pop()은 -2을 리턴한다.

# 무제한 성긴 동기화 스택

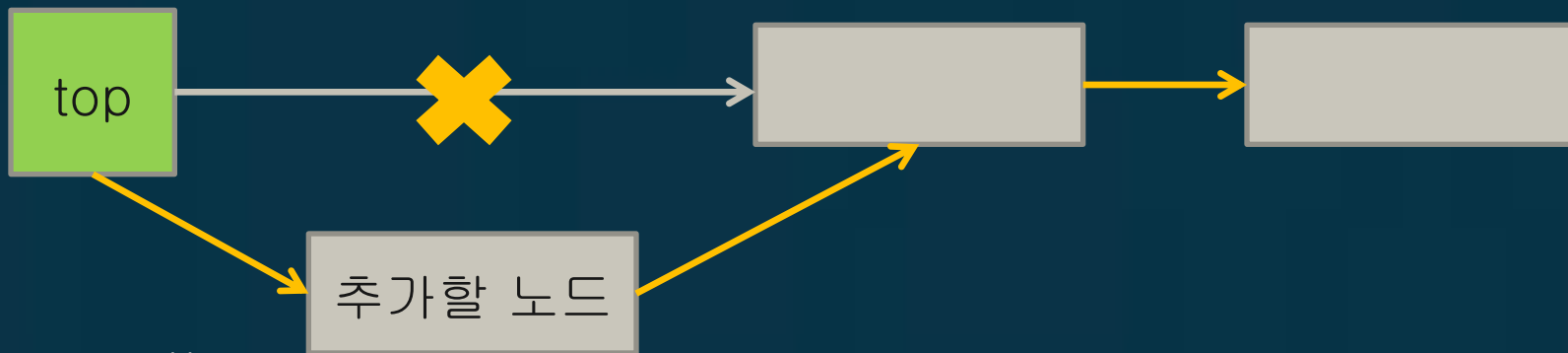
- Code

- class Node

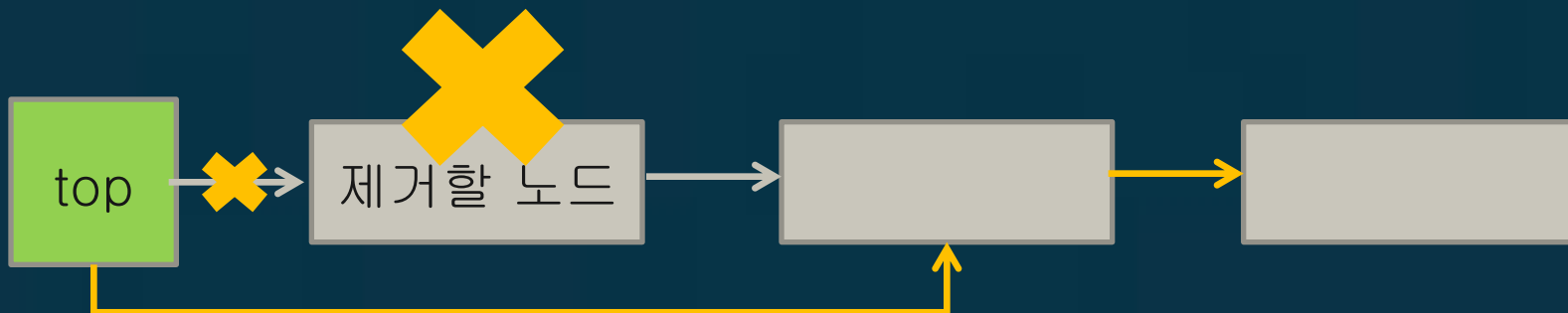
```
class Node
{
public:
    int value;
    Node *next;
    Node() { }
    ~Node() { }
    Node(int input_value) {
        value = input_value;
        next = nullptr;
    }
};
```

# 무제한 무잠금 스택

## ● Push()



## ● Pop()



# 무제한 성긴 동기화 스택

- Code

- Single Thread Code

```
int Pop()
{
    if (nullptr == top) return -2;
    int temp = top->key;
    NODE *ptr = top;
    top = top->next;
    delete ptr;
    return temp;
}
```

```
void Push(int x)
{
    NODE *e = new NODE{ x };
    e->next = top;
    top = e;
}
```



# 무제한 성긴 동기화 스택

## ● 실습 #20

- 성긴 동기화 스택을 구현하라.
- 1,2,4,8,16,32 쓰레드에서의 성능비교를 하라.
- 아래 벤치마크 프로그램을 사용하라.
  - NUM\_TEST는 1000,0000

```
void benchmark(const int num_thread)
{
    int key = 0;
    int loop_count = NUM_TEST / num_thread;
    for (auto i=0; i<loop_count; ++i) {
        if ((rand() % 2 == 0) || (i < 1000))
            my_stack.Push(key++);
        else
            my_stack.Pop();
    }
}
```

# 무제한 성긴 동기화 스택

## ● 과제 #14

- 성긴 동기화 스택을 구현하라.
- 1,2,4,8,16 쓰레드에서의 성능비교를 하라.
- 샘플 프로그램의 에러 검사기능을 사용하여 에러를 검출하라.
- 샘플 프로그램의 벤치마크 함수를 사용하라.
- 실습시간에 검사를 받은 수강생은 제출하지 말것.

# Stack

- 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	700			
2	1193			
4	1642			
8	2007			
16	1995			

# Stack

- 성능 (2023) 화금

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 1192ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	1905			
2	1409			
4	1793			
8	2200			
16	2131			

# 무제한 무잠금 스택

- 무잠금 스택은 CompareAndSet을 이용하여 구현한다.
- Top의 변환을 CAS를 사용하여 non-blocking으로 구현한다.
- ABA 문제가 있으므로 delete를 하지 않는다.

# 무제한 무잠금 스택

- 실습 #21 <<오동작 검출 프로그램 사용>>
  - 무제한 Lock-Free 스택을 구현하라.
  - 1,2,4,8 스레드에서의 성능비교를 하라.
  - 아래 벤치마크 프로그램을 사용하십시오.
  - 교재의 try-pop이니 try\_push은 무시하십시오.

```
void ThreadFunc(int threadNum)
{
    for (int i=1; i<NUM_TEST / threadNum; i++) {
        if ((rand() % 2) || (i < 1000)) {
            Stack.Push(i);
        } else {
            Stack.Pop();
        }
    }
}
```

# 무제한 무잠금 스택

## ● 숙제 #15

- 첨부 파일의 LF\_STACK 클래스를 수정하여 Lock Free Stack 을 완성하십시오.
  - Delete를 사용하지 않아도 됨 (메모리 릭 허용)
  - 첨부 파일에 있는 Error Check 기능은 건드리지 말 것.
    - 반드시 LF\_STACK class 만 수정할 것
    - benchmark\_test 함수도 수정하지 마시오.
- 실습시간에 검사를 받은 학생은 제출하지 마시오.
- 제출물
  - .cpp 파일
  - 실행속도 비교표 (Lock버전, Lock free)
  - CPU의 종류 (모델명, 코어 개수, 클럭)

# Stack

- 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	700	563		
2	1193	727		
4	1642	906		
8	2007	1034		
16	1995	1052		



# Stack

- 성능 (2024)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 1192ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	779	516		
2	1106	556		
4	1651	845		
8	2077	1078		
16	2060	1101		

# Stack

- 성능 (2023) 월금

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 935ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	1313	525		
2	1294	552		
4	1684	856		
8	2103	1015		
16	2069	1058		

# Stack

- 성능 (2022)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 930ms

Thread	Coarse Stack	Lock-Free Stack	STAMPED LF QUEUE	EBR LF QUEUE
1	1230	742		
2	1270	939		
4	1749	1027		
8	2187	1135		
16				

# 무제한 무잠금 스택

- 결론

- CAS를 사용하여 Lock-free 스택을 구현하였다.
- 메서드 호출은 스택의 top에 대해 성공한 CAS호출의 순서로 하나씩 진행되므로 순차병목현상이 나타날 수 있다.

- 메모리 문제

- new와 delete를 사용하면. ABA문제가 생긴다.
- Queue보다 문제가 생길 확률이 크다.

# BACK OFF 스택

- CAS 동기화의 문제

- 경쟁이 심할 경우 CAS 실패 시 계속 재시도하는 것은 전체 시스템에 악영향을 줌
  - Thread가 많아질 수록 경쟁이 심해짐
  - 경쟁이 심할 경우 CAS가 실패할 확률이 높음
    - 실패할 경우 성공할 때 까지 반복
  - CAS를 실행할 경우 같은 CPU의 모든 Core의 메모리 접근이 중단됨
    - Thread가 많아질 수록 잦은 메모리 접근 중단

# 목차

---

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
  - 무잠금 교환자
  - 소거 배열

# BACK OFF 스택

- BackOff

- 교재 7장에 나옴

- 경쟁이 심할 경우 경쟁을 줄이자.

- CAS의 실패는 경쟁이 심함을 뜻함.
    - CAS가 실패했을 경우 **적절한 기간** 동안 실행을 멈추었다가 재개 하자.
    - CAS의 실패 확률이 낮아짐 -> 메모리 접근 중단 감소

- **적절한 기간**

- 처음에는 짧게
    - 계속 실패하면 점점 길게
    - 첫 번 시도에 성공하면 짧게
    - Thread마다 기간을 다르게 해야 한다.

# BACK OFF 스택

- BackOff 객체

- CAS가 실패 했을 경우 다음 CAS를 시도하기 전에 사용.
- 사용될 때 마다. BackOff 시간 간격 증가.
- 시간 간격은 범위 안에서 랜덤하게



# BACK OFF 스택

- BackOff class

```
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));
    }
};
```

# Stack

- 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	LF_BO Stack	EBR LF QUEUE
1	700	563	593	
2	1193	727	658	
4	1642	906	699	
8	2007	1034	703	
16	1995	1052	800	

# BACK OFF 스택

- Better backoff

```
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);
```

# Stack

- 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	LF_BO Stack	LF_BO RDTSC
1	700	563	593	660
2	1193	727	658	701
4	1642	906	699	919
8	2007	1034	703	1103
16	1995	1052	800	1251

# Stack

- 성능 (2021-목화)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : -ms

Thread	Coarse Stack	Lock-Free Stack	Back Off Stack	Back Off RDTSC
1	-	1510	1794	1544
2	-	1383	1429	1372
4	-	1793	1731	1841
8	-	2213	2021	2197
16	-	2241	2069	2241

# Stack

- 성능

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 785ms

Thread	Coarse Stack	Lock-Free Stack	Backoff LF Stack	Backoff-2 LF STack
1	1252	773	801	800
2	1268	653	816	730
4	1358	761	757	883
8	1560	986	765	1207
16	1570	1027	748	1225

# BACK OFF 스택

- Better backoff – 2

- 절대 시간이 중요한가? CPU가 빠르면 Critical Section 처리 속도도 빠르다.
- 루프에서 메모리를 아예 접근하지 않는 것이 바람직하다.

```
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;
```

# Stack

- 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	LF_BO Stack	LF_BO RDTSC	LF_BO LOOP
1	700	563	593	660	679
2	1193	727	658	701	694
4	1642	906	699	919	919
8	2007	1034	703	1103	1090
16	1995	1052	800	1251	1115



# Stack

- 성능 (2021-목화)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : -ms

Thread	Coarse Stack	Lock-Free Stack	Back Off Stack	Back Off RDTSC	Back Off Reg Only
1	-	1510	1794	1544	1590
2	-	1383	1429	1372	1364
4	-	1793	1731	1841	1675
8	-	2213	2021	2197	1988
16	-	2241	2069	2241	2061

# Stack

- 성능 (수목반)

- Intel® Core™ i7-4770@ 2.20Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 877ms

Thread	Coarse Stack	Lock-Free Stack	BackOff LF Stack	BackOff-3 LF STack
1	1245	636	737	723
2	1545	797	643	635
4	1661	1139	783	807
8	1968	724	1031	1166
16	1960	730	1082	1186

# BACK OFF 스택

## ● 실습 #22

- 무제한 무잠금 BackOFF Lock-Free 스택을 구현하라.
- 1,2,4,8 쓰레드에서의 성능비교를 하라.
- 아래 벤치마크 프로그램을 사용하라.

```
void ThreadFunc(int threadNum)
{
    for (int i=1; i<25000000 / threadNum; i++) {
        if ((rand() % 2) || (i < 1000)) {
            Stack.Push(i);
        } else {
            Stack.Pop();
        }
    }
}
```

# BACK OFF 스택

- 숙제 12 : << 생략 >>

- Lock Free BackOff Stack의 구현

- 강의자료실에 있는 LFSTACK활용

- 제출물

- .cpp 파일
- 실행속도 비교표 (Lock버전, Lock free, Lock free BackOff)
- CPU의 종류 (모델명, 코어 개수, 클럭)

- 제출 : [nhjung@kpu.ac.kr](mailto:nhjung@kpu.ac.kr)

- 제목 : [2019 멀티코어 프로그래밍 숙제 12] 학번, 이름
- 11월 19일 화요일 오전 11시까지 제출

# BACK OFF 스택

- BackOff Stack 성능 (8 Core, 4 CPU XEON)

	1	2	4	8	16	32	64
BackOff STACK	1370	1158	1233	1198	1749	2977	2986
LF STACK	1392	1236	2298	2704	4739	6779	7081

# 목차

---

- 소개
- 무제한 무잠금 스택
- 백오프 스택
- 소거 백오프 스택
  - 무잠금 교환자
  - 소거 배열

# 소거

## ● 병행성 문제

– Queue나 Stack은 리스트의 말단 부분에서  
잡은 충돌이 생긴다.

- fine grain synchronization타입의 최적화가 불가능하다.

- Queue는  $\frac{1}{2}$ 로 충돌이 분산된다.

– Stack의 경우 충돌을 회피해서 병렬로  
처리하는 방법이 가능

- 아이디어 :

- push, pop이 동시에 발생하는 경우 꼭 Stack에 넣어야 하나?

# 소거

## ● 소거

- 많은 쓰레드가 서로 충돌할 경우 Stack에 넣지 않고 직접 Data를 주고 받도록 한다.
  - Stack을 통하지 말고 다른 객체를 통해 전달
- Lock-Free로 Data를 주고 받도록 한다.
- 높은 경쟁률에 대비하여 주고 받는 별도의 객체를 복수로 준비한다.

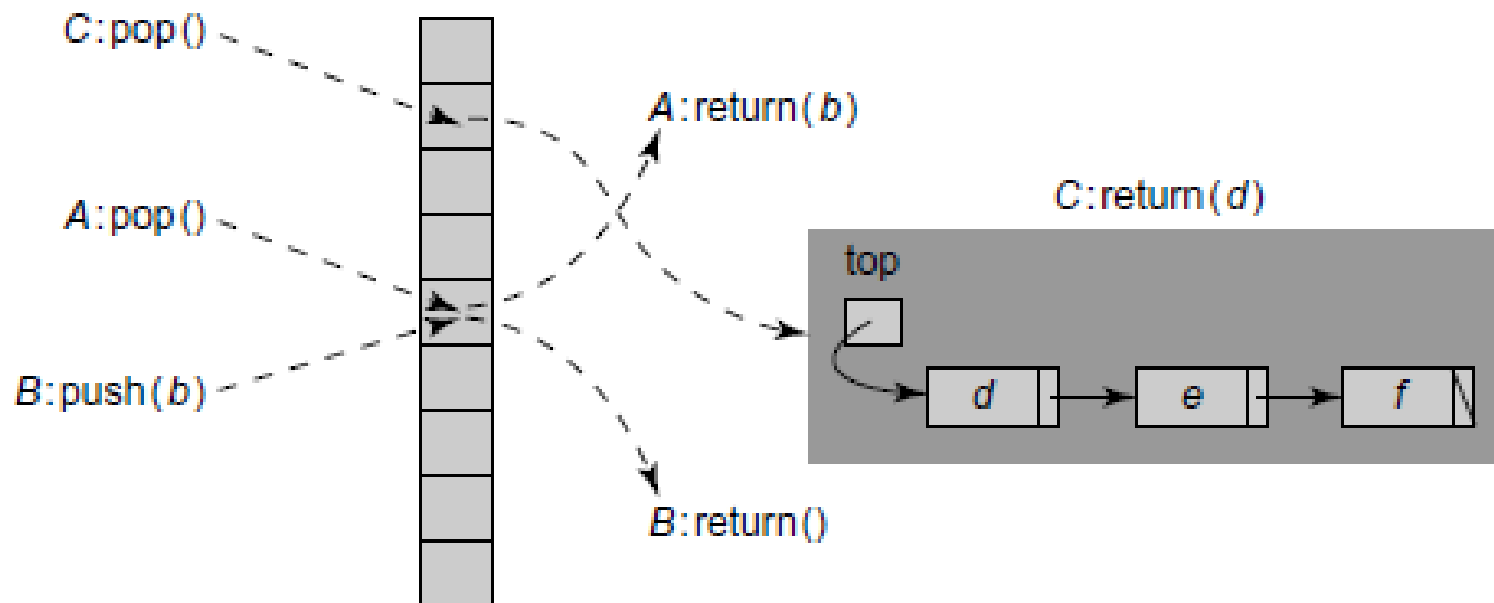


# 소거

- 만약 push와 pop이 거의 동시에 실행된다면 두 연산은 서로 취소되어 없어지고 스택에 접근하지 않는다.
- 이런 경우 push를 호출하는 스레드는 스택의 변동 없이 pop을 호출하는 스레드와 값을 교환할 수 있다.
  - 이 때 서로를 소거(eliminate)하게 된다고 한다.

# 소거

## ● 소거(elimination)



# 소거

- EliminationArray
  - 원소 : 자료교환의 장소 (교환자(Exchanger) 객체)
  - 여러 개의 원소를 갖고 부하를 분산
    - 서로 만나지 못할 경우도 있다.
    - 경쟁이 심할 경우 많은 장소가 효율적이고, 경쟁이 적을 경우는 장소가 적어야 한다.
- 스레드는 EliminationArray에서 임의의 항목을 골라서, 서로를 소거하려 시도
  - 잘못된 종류의 호출을 만난 경우는 소거 실패
    - 예) push와 push, pop과 pop
  - 임의로 하지 않고, 짝을 잘 맞추어 주면?
    - 정보 전달 자체가 오버헤드
    - Lock-free로 하는 것은 어렵다. Open-Problem

# 소거 백오프 스택

- 앞에서 구현했던 LockFreeStack에서 백오프를 사용하지 않고, 대신에 소거를 시도한다.
  - 소거 :
    - Elimination Array의 임의의 원소를 선택한다
      - 임의 : 협조 자체가 오버헤드
      - Array의 크기는 가변
    - 소거 시도 :
      - 아무도 없으면 기다림
      - 다른 스레드가 기다리고 있으면 교환
      - 이미 교환 중이면 재시도 [array size 증가 필요]
    - 주의점
      - Time Out 필요 [array size 감소 필요]
      - 서로 만난 스레드가 <push>, <pop>의 조합이라는 보장이 없음
        - 잘못된 만남은 실패

# 무잠금 교환자

- Elimination Array의 원소
- 두개의 쓰레드가 Lock free로 값을 교환할 수 있게 해주는 객체
- Exchange(myitem)의 메소드를 갖고 있으며 두개의 쓰레드가 Exchange()를 호출하면 서로의 입력 값을 리턴한다.
  - pop도 자기 값(-1)을 상대방에게 준다.
- 리턴 값은 교환값과 상태(성공, 실패, 타임아웃(-2)) 이다.

```

1 public class LockFreeExchanger<T> {
2     static final int EMPTY = ..., WAITING = ..., BUSY = ...;
3     AtomicStampedReference<T> slot = new AtomicStampedReference<T>(null, 0);
4     public T exchange(T myItem, long timeout, TimeUnit unit)
5         throws TimeoutException {
6         long nanos = unit.toNanos(timeout);
7         long timeBound = System.nanoTime() + nanos;
8         int[] stampHolder = {EMPTY};
9         while (true) {
10             if (System.nanoTime() > timeBound)
11                 throw new TimeoutException();
12             T yrItem = slot.get(stampHolder);
13             int stamp = stampHolder[0];
14             switch(stamp) {
15                 case EMPTY:
16                     if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
17                         while (System.nanoTime() < timeBound){
18                             yrItem = slot.get(stampHolder);
19                             if (stampHolder[0] == BUSY) {
20                                 slot.set(null, EMPTY);
21                                 return yrItem;
22                             }
23                         }
24                         if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
25                             throw new TimeoutException();
26                         } else {
27                             yrItem = slot.get(stampHolder);
28                             slot.set(null, EMPTY);
29                             return yrItem;
30                         }
31                     }
32                     break;
33                 case WAITING:
34                     if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY))
35                         return yrItem;
36                     break;
37                 case BUSY:
38                     break;
39                 default: // impossible
40                     ...
41             }
42         }
43     }

```

# 무잠금 교환자

- 구현

- Slot

- 처음 교환을 시도한 쓰레드의 입력값을 저장하는 공간
    - 두 번째 쓰레드가 자신의 값을 넘겨 주는 공간

- 교환기는 3개의 상태를 가진다.

- EMPTY

- 슬롯에 아무런 값도 저장되지 않은 상태

- WAITING

- 처음 온 쓰레드에서 슬롯에 값을 하나 저장한 상태
      - 다른 쓰레드가 와서 읽어가기를 기다리는 상태

- BUSY

- 나중에 온 쓰레드가 슬롯의 값을 읽어가고 자신의 값으로 변경한 상태
      - 처음 쓰레드는 아직 읽기를 완료하지 않음

# 무잠금 교환자

## ● 상태에 따른 구현 (1/2)

### — EMPTY

- CAS를 이용하여 슬롯에 자신의 아이템을 넣고 상태를 WAITING으로 바꾸기를 시도한다.

#### — 실패

- 다른 스레드가 이미 CAS를 성공한 경우이다.
- 다른 스레드가 waiting으로 변경
- 처음부터 다시 시도

#### — 성공

- 스핀을 하면서 다른 스레드가 교환을 끝내길 기다림
- 교환이 완료(BUSY) 된 후에는 대기 중이던 스레드는 아이템을 가져간 후 상태를 empty 으로 바꾼다.
  - Empty()로 전환하는 일은 CAS를 이용하지 않는다
  - 오직 한 스레드 만 작업하기 때문



# 무잠금 교환자

## ● 상태에 따른 구현 (2/2)

### — EMPTY

- Waiting으로 바꾼 후 다른 스레드를 기다림
- 다른 스레드가 나타나지 않는다면,
  - 대기중이던 스레드는 slot의 상태를 CAS를 호출하여 EMPTY로 전환
  - 성공시
    - 시간제한 예외 발생
  - 실패시
    - 다른 스레드가 나타나서 교환을 완료하고 Busy로 변경했다는 의미이므로 대기중이던 스레드는 교환을 완료한다.

# 무장금 교환자

## ● 구현

### —WAITING

- 어떤 스레드가 대기중이며 슬롯은 그 스레드의 아이템을 갖는다.
  - CAS를 이용하여 자신의 아이템을 적고 대기중인 스레드의 아이템을 얻는 것을 시도한다.(BUSY상태로 전환)
  - 실패시에는 처음부터 다시
  - 성공하면 얻은 아이템을 반환한다.

### —BUSY

- 현재 다른 두 스레드가 슬롯을 써서 값을 교환하고 있으므로 처음부터 다시.

# 무잠금 교환자

- 구현

- AtomicStampedReference의 구현

- 값의 교환과 상태 변경을 Atomic하게 구현해야 한다.
    - StampedQueue와 같은 구현?
      - InterlockedCompareExchange64
      - InterlockedCompareExchange128
    - 해결책
      - 32비트 int의 상위 2비트 사용
        - 교환할 수 있는 값의 최대값 감소

# 무잠금 교환자

## ● ABA 문제??

— 해당 알고리즘은 ABA문제를 일으키지 않는다.

- ABA현상이 발생한다.
- 하지만 그냥 진행해도 문제없다.
- 교환대상이 바뀐 것 뿐이고 값도 그대로이기 때문에 아무 문제 없다.
  - 사실은 CAS를 통해서 old값을 읽기 때문에 상태만 그대로이면 값이 바뀌는 것 까지도 문제가 없다.

## ● 타임아웃 문제

— 너무 짧은 교환 시간은 항상 실패하게 되므로 시간제한 기간을 고를 때 주의해야 한다.

# 소거 배열

- 소거 배열 (Elimination array)

- Exchange객체의 배열.

- capacity개의 원소를 갖는다.

- visit()

- 소거배열이 갖고 있는 exchange원소 중 하나를 랜덤하게 선택해서 교환을 시도한다.
    - range값을 통해 random값의 범위를 조정할 수 있다.

# 소거 배열

- Code

- Class Elimination

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

# 소거 배열

- EliminationBackoffStack

- LockFreeStack에서 CAS에 실패했을 경우 백오프를 하는 대신에 EliminationArray를 써서 값의 교환을 시도한다.

- Push()

- CAS(...)

- 성공하면 리턴

- 자신의 입력값을 인자로 Visit()를 호출

- timeout이 아니면

- 다른 스레드와의 교환이 성공

- pop과의 교환이었으면 리턴 (교환된 값이 -1)

- pop과의 교환이 아니었으면 다시 CAS로~~

- timeout이면

- 다시 CAS로~~~

# 소거 배열

## ● EliminationBackoffStack

### — Pop()

- CAS(...) 시도
  - 성공하면 return
- visit(-1)를 호출
  - timeout이 아닌 경우 : 교환 성공
    - push()호출과 교환했는지 검사 (교환값 != -1)
    - push()호출과의 교환이면 교환값을 리턴
    - 다시 CAS()로~~~
  - Timeout인 경우
    - 다시 CAS()로~~~.



# 소거 배열

- RangePolicy

- EliminationArray의 LoadBalancing
- EliminationArray에서 사용 중인 최대 인덱스 값을 갖는다.
- 경쟁이 심하면 값을 늘리고
  - 방문했을 때 Busy인 경우
- timeout이 발생하면 값을 줄인다.

# 소거 배열

## ● EliminationBackoffStack

— 만약 교환이 실패한다면 원인은?

- 처음부터 교환이 없었거나, 같은 종류의 연산과 값이 교환되었을 경우 실패한다.
- 이런 경우 다시 top에 대한 CAS를 재시도 한다.

# 소거 배열

## ● EliminationBackoffStack

— 교환자의 위치 선택 범위는?

- 범위의 크기와 충돌 확률은 반비례적
- 스레드의 수가 적을 경우 범위는 작아야하고 스레드수가 클 경우 범위가 넓어야 한다.
- RangePolicy객체를 이용하여 범위를 지정한다.
  - 간단하게는 time\_out 하면 범위를 줄이고 exchanger에 너무 많은 스레드가 몰리면 범위를 늘린다.

# 실습

- 실습 #21 : EliminationBackoffStack
  - Lock-Free 소거 백오프 Stack을 완성하라
    - 샘플 프로그램 참조 (버그가 있을 수 있음)
      - 오류 검사 그대로 사용하기.
    - 소거 성공 횟수를 기록해서 출력 할 것.
  - 1, 2, 4, 8, 16, 32 쓰레드에서의 성능비교를 하라.
  - 실습 #20과의 성능 비교를 하라.

# 속제

- 속제 16 :

- LockFree 소거 Stack의 구현
- 첨부한 stack.cpp를 수정할 것
- 성공적으로 교환된 횟수를 출력할 것
  - 스레드 1개, 2개일 때는 교환이 0이어야 함.
- 제출물
  - .cpp 파일
  - 실행속도 비교표 (Lock버전, Lock free, Lock free 소거)
  - CPU의 종류 (모델명, 코어 개수, 클럭) | 8

# Stack

## ● 성능 (2024-2학기)

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 690ms

Thread	Coarse Stack	Lock-Free Stack	LF_BO Stack	LF_BO RDTSC	EL Stack
1	700	563	593	660	467
2	1193	727	658	701	620
4	1642	906	699	919	940
8	2007	1034	703	1103	1105
16	1995	1052	800	1251	1182

# Stack

- 성능 (2023) 화금

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 1192ms

Thread	Coarse Stack	Lock-Free Stack	LF BackOff	LF EL Stack
1	1905	526	500	535
2	1409	537	512	696
4	1793	867	505	787
8	2200	1130	498	885
16	2131	1165	535	921

# Stack

- 성능 (2023) 월금

- Intel® Core™ i5-10210U@ 1.60Hz, 4Core, Hyperthread, 16GB Memory
- Single Thread : 935ms

Thread	Coarse Stack	Lock-Free Stack	LF-BO-STACK	LF-EL-STACK[
1	1313	525	566	517
2	1294	552	807	547
4	1684	856	785	656
8	2103	1015	921	832
16	2069	1058	775	866



# Stack

- 성능(화목)

- Intel® Core™ i7-7700@ 3.60Hz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 857 ms

Thread	Coarse Stack	Lock-Free Stack	Backoff-2 LF Stack	Elimination LF Stack
1	1077	732	692	
2	1191	573	644	
4	1316	616	661	
8	1491	737	752	
16	1505	766	753	

# Stack

- 성능(화목)

- Intel® it-4770HQ 2.2GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread :   ms

Thread	Coarse Stack	Lock-Free Stack	Backoff-2 LF Stack	Elimination LF Stack
1		954		763
2		640		776
4		785		799
8		1005		1145
16		1211		1242

# Stack

- 성능(화목-B조)

- Intel® it-4770HQ 2.2GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread :   ms

Thread	Coarse Stack	Lock-Free Stack	Backoff-2 LF Stack	Elimination LF Stack
1		733		776
2		619		753
4		937		867
8		1381		1362
16		1404		1335

# Stack

- 성능(2022)

- Intel® XEON, 2 CPU (18 X 2)core, Hyperthread
- Single Thread : 240 ms

Thread	Coarse Stack	Lock-Free Stack	Backoff-2 LF Stack	Elimination LF Stack
1	393	397	409	412
2	1153	1559	504	1327
4	2706	2341	979	2775
8	4091	3035	1136	3935
16	4464	3768	1419	4356
32	4671	3707	892	4693
64	6517	4618	1575	6350
128	7060	4882	1836	6932

# 결론

## ● EliminationBackoffStack

- LockFreeStack과 달리 확장될 가능성 있음
- 부하가 높아지면 성공적인 소거가 증가하고, 병렬적으로 여러 개의 연산이 완료된다
- 소거된 연산은 절대로 스택에 접근하지 않기 때문에 LockFreeStack에서의 경쟁이 줄어든다.

# 목차

---

- Stack

- SKIPLIST

# SKIP-LIST

---

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트

# 소개

- 지금까지의 자료구조
  - Queue, Stack
  - Set
    - Linked List, 검색시간  $O(n)$
- $O(n)$  검색시간?
  - 실제로 사용하기에는 너무 무리
  - 원소의 개수가 적을 경우에는 사용 가능
- 실제 사용하는 자료구조
  - Hash : 순서가 필요 없는 경우
  - Binary Tree 계열 (B+, Red-Black, Heap, AVL..)



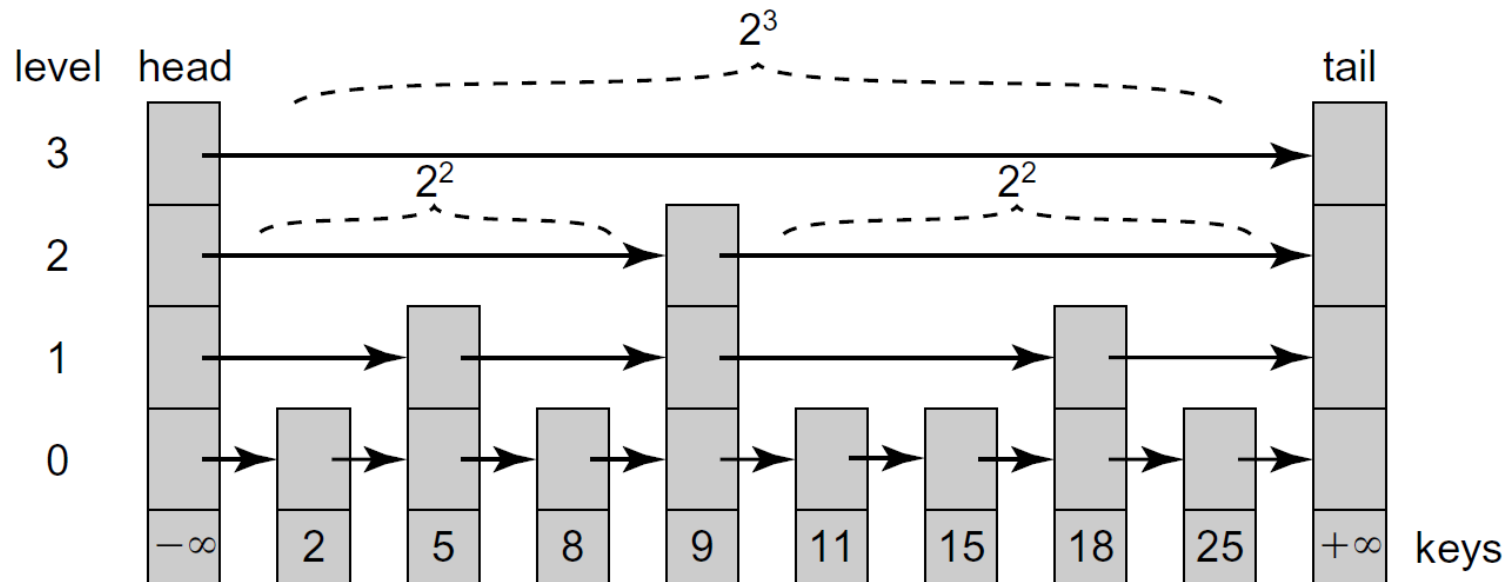
# 소개

- 일반적인 트리구조
  - 트리 깊이의 균형을 유지하기 위해서 정기적인 재균형(Rebalancing) 작업이 필요(AVL트리..)
    - 하지만 병행 구조에서는 재균형작업이 병목이나 경쟁상태를 유발할 가능성이 있다.
- SkipList
  - 평균  $O(\log n)$  검색시간을 갖는 자료 구조
  - 재균형작업이 필요 없음!!!
  - 랜덤 자료구조(Randomized algorithm, Probabilistic data structure)
  - Worst Case  $O(n)$  임

# 소개

## ● 스킵리스트의 예

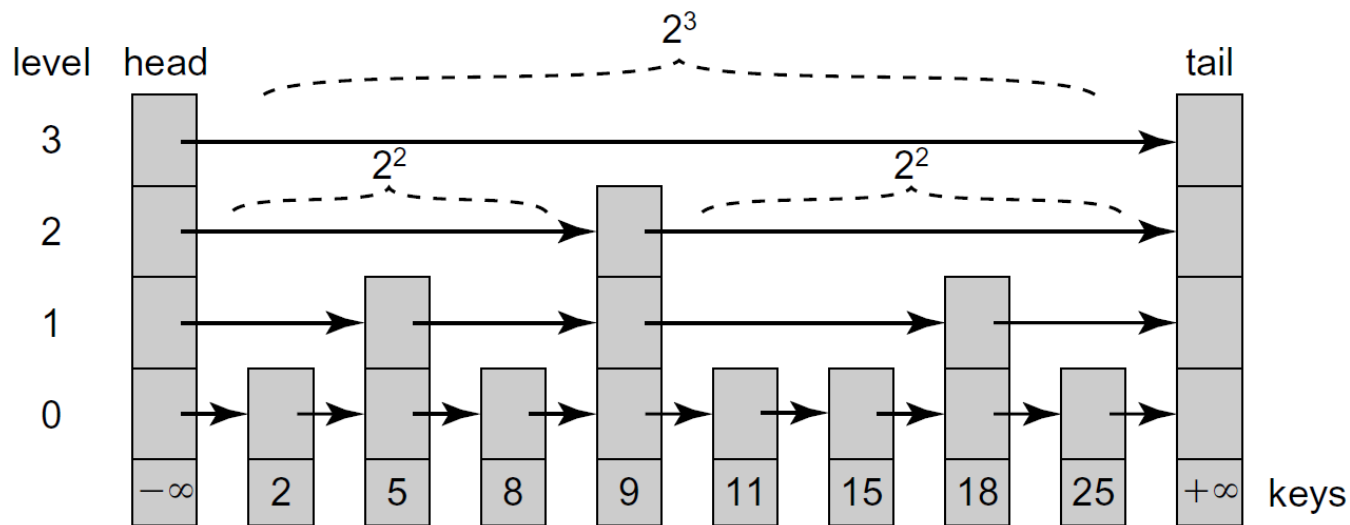
- Set을 구현할 때 사용한 연결리스트의 확장
- 노드들이 추가 포인터를 갖고 있으며, 같은 레벨의 포인터끼리 연결됨. ( $\text{next} \rightarrow \text{next}[n]$ )



# 소개

## ● 스킵리스트의 기본 아이디어

- 추가 링크 (레벨 1 이상) 지름길을 만든다.
  - 노드를 추가하거나 제거할 때 지름길을 유지한다.
- 검색을 할 때 지름길을 우선 이용한다.





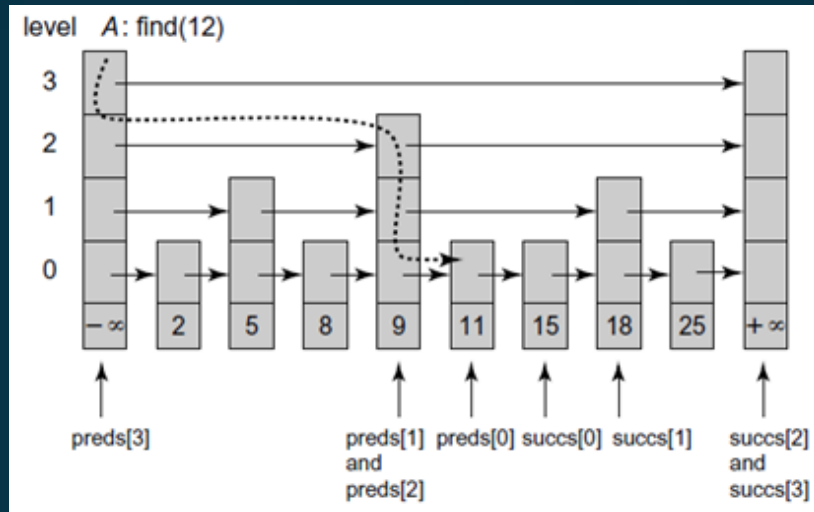
# SKIP-LIST

---

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트

# 순차 스킵리스트

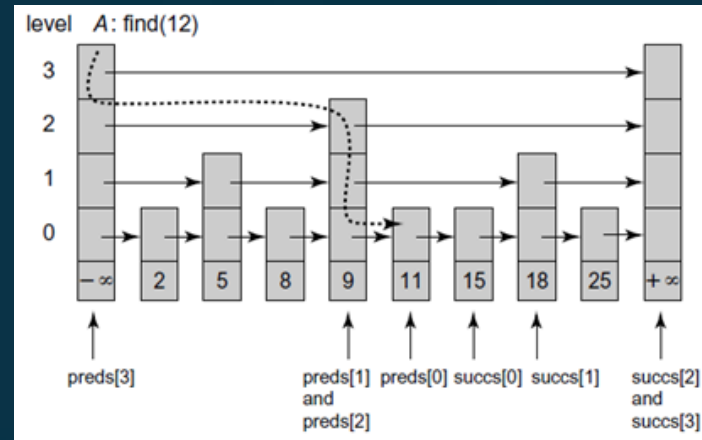
## ● 순차 스킵리스트 : Find(12)



- 언제나 시작은 head에서,
- 높은 레벨의 포인터부터 검색 시작
- 한 레벨의 검색이 끝나면 다음 레벨의 검색 시작
  - 레벨별 검색결과 저장
- 맨 아래 레벨에 도달할 경우 종료된다.

# 순차 스킵리스트

- 순차 스킵리스트
  - 검색의 구현 Find()

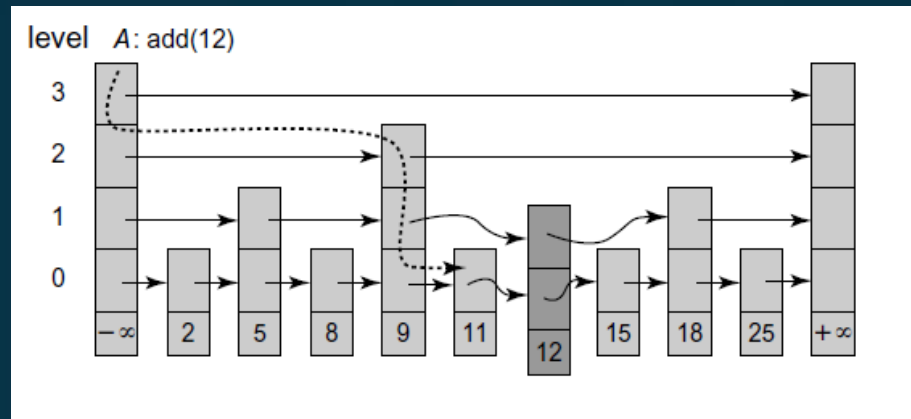


— 더 높은 층의 리스트는 낮은 층의 리스트에 대한 지름길이다.

- 계층의 각 연결은 바로 아래 계층의  $2^{i-1}$ 개의 노드를 건너 뛴다

# 순차 스킵리스트

## ● 순차 스킵리스트 : Add()



- 랜덤한 높이의 노드를 생성한다.
  - 높이가 높을 수록 노드의 개수는 감소 (단계 마다  $\frac{1}{2}$ )
- 검색한 위치에 추가한다.
- pred, curr 가 아닌 pred[], curr[]가 필요하다.
  - remove()도 마찬가지



# 순차 스킵리스트

## ● Node 구조

```
class NODE
{
public:
    int value;
    NODE *next[MAX_LEVEL + 1];
    int top_level;
    NODE()
    {
        value = 0;
        for (auto i = 0; i <= MAX_LEVEL; ++i)
            next[i] = nullptr;
        top_level = 0;
    }
    NODE(int x, int top)
    {
        value = x;
        for (auto i = 0; i <= MAX_LEVEL; ++i)
            next[i] = nullptr;
        top_level = top;
    }
};
```

# 순차 스킵리스트

## ● 공통함수 : Find

```
void Find(int key, SLNODE *preds[], SLNODE *currs[])
{
    preds[MAX_LEVEL] = &head;
    for (int cl = MAX_LEVEL; cl >= 0; --cl) {
        if (cl != MAX_LEVEL)
            preds[cl] = preds[cl + 1];
        currs[cl] = preds[cl]->next[cl];
        while (currs[cl]->key < key) {
            preds[cl] = currs[cl];
            currs[cl] = currs[cl]->next[cl];
        }
    }
}
```

# 리스트의 구현

## ● 실습 : #22

- 성긴 동기화 스킵리스트를 구현하시오
- E-Class에 올려놓은 Sample 프로그램의 벤치마킹 함수와 에러검사 함수를 사용하시오
- Remove()시 노드를 delete하시오.
- 쓰레드가 1개, 2개, 4개, 8개, 16개일 때의 속도를 비교하시오.
  - 각각 실행 전 Skip-List를 클리어 하시오

# 리스트의 구현

- 실습 : #22
  - test program

```
#define NUM_TEST    4000000
#define KEY_RANGE    1000
```

```
void *ThreadFunc(void *lpVoid)
{
    int key;

    for (int i=0;i < NUM_TEST / num_thread;i++) {
        switch (rand() % 3) {
            case 0: key = rand() % KEY_RANGE;
                    csklist.Add(key);
                    break;
            case 1: key = rand() % KEY_RANGE;
                    csklist.Remove(key);
                    break;
            case 2 : key = rand() % KEY_RANGE;
                    csklist.Contains(key);
                    break;
            default : printf("Error\n");
                    exit(-1);
        }
    }
}
```

# 속제

- 속제 13 :

- 성긴 동기화 SkipList의 구현

- SET구현할 때 사용한 Linked List 구현 프로그램에 SKIPLIST를 추가해서 구현하고, 기존의 오류검사와 벤치마크 프로그램을 사용해서 결과를 도출하시오

- 제출물

- .cpp 파일
- 실행속도 비교표 : 쓰레드 개수 별, Linked List 성긴동기화 Set과의 성능 비교
- CPU의 종류 (모델명, 코어 개수, 클럭)

- 제출 : eclass

# 성긴 동기화

## ● 결과

	성긴동기화	게으른 동기화	LF 동기화	C_SK LIST
1	1364	2710	2166	659
2	1620	2060	2242	814
4	2625	1278	1782	1115
8	12251	907	1335	1521
16	12057	914	1390	1524

싱글 쓰레드 : 1350

싱글 쓰레드 SK LIST: 688

# 성긴 동기화 SKIP-LIST

- 결과 (2024)

	성긴동기화	Lock Free	성긴 SKLIST	
1	1544	1469	653	
2	1730	1225	797	
4	2278	1006	1091	
8	12336	703	1444	

싱글 쓰레드 : 2639

싱글 쓰레드 SKLIST : 1199

# 성긴 동기화 SKIP-LIST

## ● 결과 (2023-화금)

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	Lock Free	성긴 SKLIST	게으른 SKLIST
1	2334				1543	1177	1873
2	2640				1331	1343	1050
4	4126				1013	1536	416
8	17938				760	2826	330

싱글 쓰레드 : 1991

싱글 쓰레드 SKLIST : 992



# 성긴 동기화 스킵리스트

- 성능(화목)

- Intel® i7-7700 3.6GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 1822ms(list), 658ms(skiplist)

Thread	Coarse List	Lazy List	LF List	Coarse SKLIST
1	1986	2208	2173	960
2	1857	1615	1780	882
4	1994	1016	1232	916
8	5716	643	895	1120
16	7297	645	888	1127

# SKIP-LIST

---

- 소개
- 순차 스킵 리스트
- 병행(Lazy) 스킵 리스트
- Lock-Free 스킵 리스트

# 잠금 기반의 병행 스킵리스트

- Non-Blocking 스킵리스트의 전 단계
- Mark field를 사용한 Lazy 방식의 스킵리스트
- 문제점 : 중간 단계의 증가
  - Linked List : Marking은 되었으나 링크에서 제거되지 않은 상태
  - Skip List :
    - (Remove) Marking은 되었으나 링크에서 **완전히** 제거되지 않은 상태
    - (Add) 0레벨은 연결되었으나 윗 레벨이 연결되지 않은 상태

# 잠금 기반의 병행 스킵리스트

- 중간단계에 대한 대처
  - 중간단계의 노드에 대한 명확한 정의 필요
    - ADD : 중간단계의 Node는 아직 Add 되지 않은 노드.
    - REMOVE : Marking이 되었으면 Remove된 노드
    - 중간 단계의 노드가 발견되면 완전히 처리될 때 까지 기다렸다가 Return하자.
  - 오동작 방지
    - Remove : 위에서 부터 아래로 링크를 제거한다.
      - Locking은 아래에서 부터 위로 한다.
    - Add : 아래에서 부터 위로 링크를 연결한다.
    - WHY?
      - 위의 링크가 연결되지 않아도 검색은 문제 없다.
        - 하지만 `pred[]`, `curr[]`에 잘못된 값이 들어갈 수 있다.
      - 아래의 링크가 연결되지 않았다면 `nullptr` 참조 에러

# 잠금 기반의 병행 스킵리스트

## ● 구현

- ADD : Linked List와는 달리 모든 링크가 연결되었는지를 나타내는 Flag 필요 => **fullylinked**
- REMOVE :
  - 완벽하게 prev[], curr[]를 locking하고 마킹하는 것은 비효율적
    - Head와 tail이 locking되면 전체가 멈춤.
  - 마킹 후 위의 링크에서 Invalidate가 발생했을 때 처리 필요
    - 다른 스레드에서 노드를 ADD 할 수 있음.
    - Find를 다시 해야 한다.

# 잠금 기반의 병행 스킵리스트

- 조감도
  - 노드
    - 개별적인 잠금
      - `std::recursive_mutex` 필요 : 하나의 노드를 여러 번 잠글 수 있음.
    - `Marked` 필드(`bool`)
      - `Remove()`시 논리적으로 제거하고 있는 중이라면 `true`
    - `next[n]`
      - 각 층에 해당하는 포인터의 배열
    - `fullyLinked` 필드(`bool`)
  - 보초노드 ( `head`, `tail` )
    - 초기(`SkipList`가 비어있을 경우)에는 `head`의 모든 층은 `tail`을 가리킨다.

# 잠금 기반의 병행 스킵리스트

- Find

- 찾지 못했을 경우 -1을 리턴
- 찾았을 경우 최고 레벨을 리턴
- 주의) 위쪽 링크가 다 연결되지 않았을 경우가 있으므로 검색이 끝난 후 최고 레벨을 리턴하면 곤란. (리턴 값과 최고 레벨을 비교하여 pred, curr가 모든 레벨의 링크를 담고 있는가 확인 가능)

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {  
    int key = x.hashCode();  
    int lFound = -1;  
    Node<T> pred = head;  
    for (int level = MAX_LEVEL; level >= 0; level--) {  
        Node<T> curr = pred.next[level];  
        while (key > curr.key) {  
            pred = curr; curr = pred.next[level];  
        }  
        if (lFound == -1 && key == curr.key) {  
            lFound = level;  
        }  
        preds[level] = pred;  
        succs[level] = curr;  
    }  
    return lFound;  
}
```

# 잠금 기반의 병행 스킵리스트

- Add()

- Add()는 항상 Find()를 호출

- Find()

- 스킵리스트를 순회하고 모든 층에 대해, 앞 노드와 뒤 노드를 반환

- 노드가 추가되는 동안 앞 노드가 변경되는 것을 방지하기 위해 앞 노드들을 잠근다.

- fullyLinked 필드

- 모든 층에서 추가된 노드에 제대로 참조를 설정할 때까지 논리적으로 집합에 있지 않다고 판단
    - 모든 층에 연결 될 경우에 true
    - False일 경우에 접근이 허용되지 않으며 true가 될 때까지 스킵



# 잠금 기반의

## • ADD

- 검색한다
- 존재 할 경우
  - marking되어 있으면 다시 시작 (invalid이므로)
  - fullyLinking를 기다린 후 false return
- 0레벨부터 valid 검사하면서 locking
- 0레벨부터 리스트에 연결
- unlocking
- 아직 추가되지 않았는데 false를 return하면 안됨.

```

boolean add(T x) {
    int topLevel = randomLevel();
    Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
    Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
    while (true) {
        int lFound = find(x, preds, succs);
        if (lFound != -1) {
            Node<T> nodeFound = succs[lFound];
            if (!nodeFound.marked) {
                while (!nodeFound.fulllyLinked) {}
                return false;
            }
            continue;
        }
        int highestLocked = -1;
        try {
            Node<T> pred, succ;
            boolean valid = true;
            for (int level = 0; valid && (level <= topLevel); level++) {
                pred = preds[level];
                succ = succs[level];
                pred.lock.lock();
                highestLocked = level;
                valid = !pred.marked && !succ.marked && pred.next[level] == succ;
            }
            if (!valid) continue;
            Node<T> newNode = new Node(x, topLevel);
            for (int level = 0; level <= topLevel; level++)
                newNode.next[level] = succs[level];
            for (int level = 0; level <= topLevel; level++)
                preds[level].next[level] = newNode;
            newNode.fulllyLinked = true; // successful add linearization point
            return true;
        } finally {
            for (int level = 0; level <= highestLocked; level++)
                preds[level].unlock();
        }
    }
}

```

# 잠금 기반의 병행 스킵리스트

## ● Remove()

- Find()를 호출해서 대상키를 가지고 있는 노드가 이미 리스트에 있는지 확인
  - 리스트에 있다면, 제거할 노드를 삭제할 수 있는지 확인
    - Marked, fullyLinked 필드를 이용해 확인
    - 삭제할 수 있다면, marked필드를 설정하여 논리적으로 삭제
    - 물리적인 삭제 방법은
      - 대상 노드의 앞 노드와 대상 노드를 잠그고, 각 층에서 하나씩 대상 노드를 제거(위층부터 제거)하고 다음 노드와 연결한다.

# 잠금 기반의 병

## ● Remove()

### — 변수

- victim : 제거하려는 노드
- isMarked : 제거하려는 노드에 marking을 했는가?

### — 마킹을 했거나 제거 조건이 만족되면 제외 시도

- 마킹을 하지 않았으면 마킹 실행

### — locking을 할 때 valid 검사

- marking되었다면 locking원상 복구 후 continue;

### — top\_level부터 링크를 끊는다.

```

95  boolean remove(T x) {
96      Node<T> victim = null; boolean isMarked = false; int topLevel = -1;
97      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
98      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
99      while (true) {
100         int lFound = find(x, preds, succs);
101         if (lFound != -1) victim = succs[lFound];
102         if (isMarked |
103             (lFound != -1 &&
104              (victim.fullyLinked
105               && victim.topLevel == lFound
106                && !victim.marked))) {
107             if (!isMarked) {
108                 topLevel = victim.topLevel;
109                 victim.lock.lock();
110                 if (victim.marked) {
111                     victim.lock.unlock();
112                     return false;
113                 }
114                 victim.marked = true;
115                 isMarked = true;
116             }
117             int highestLocked = -1;
118             try {
119                 Node<T> pred, succ; boolean valid = true;
120                 for (int level = 0; valid && (level <= topLevel); level++) {
121                     pred = preds[level];
122                     pred.lock.lock();
123                     highestLocked = level;
124                     valid = !pred.marked && pred.next[level] == victim;
125                 }
126                 if (!valid) continue;
127                 for (int level = topLevel; level >= 0; level--) {
128                     preds[level].next[level] = victim.next[level];
129                 }
130                 victim.lock.unlock();
131                 return true;
132             } finally {
133                 for (int i = 0; i <= highestLocked; i++) {
134                     preds[i].unlock();
135                 }
136             }
137         } else return false;
138     }
139 }

```

# 잠금 기반의 병행 스kip리스트

- Remove() : 다시

- 제거 조건 검사 (103 ~ 106)

- find에서 찾음, marking안됨, fullylinked임.  
pred/curr에 모든 링크가 담겨 있음
- 아니면 return false

- 조건에 맞으면 marking

- lock을 걸고 mark
- 다른 스레드에서 먼저 marking하면 return false

- Marking 후 링크 재설정

- invalid한 pred를 발견한 경우 다시 find 후 재설정

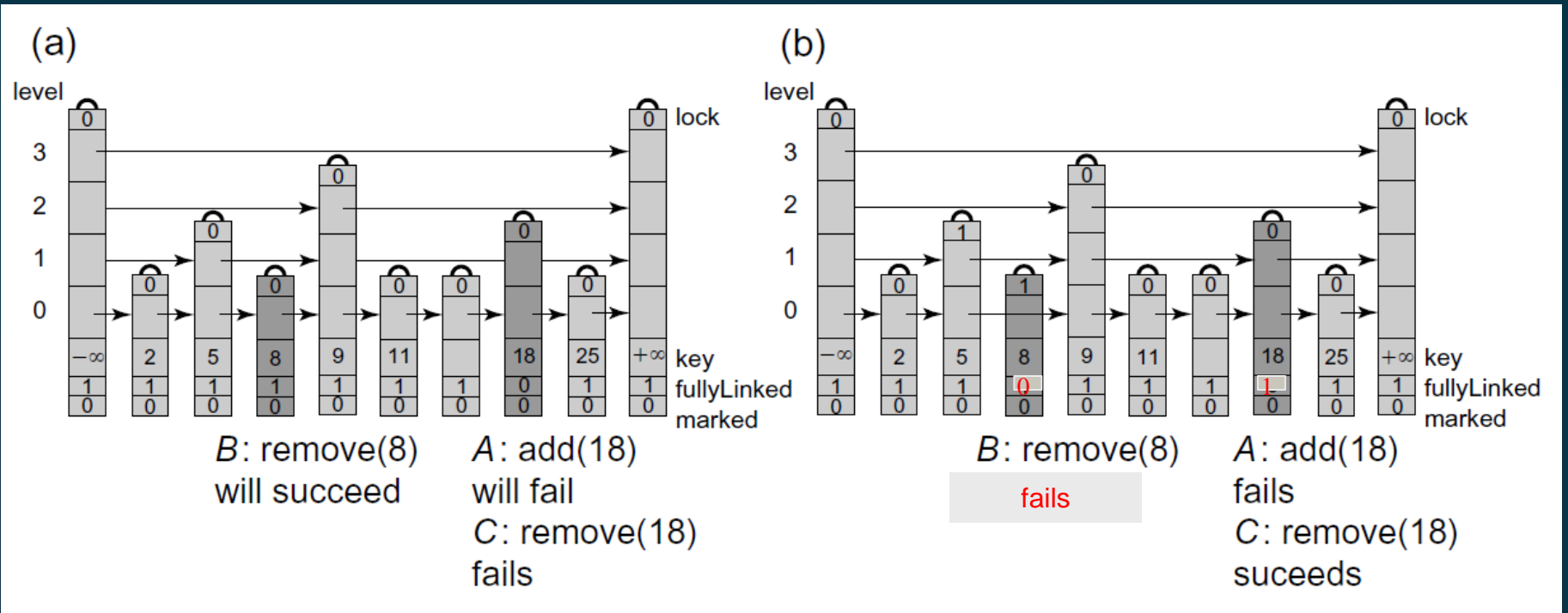
# 잠금 기반의 병행 스킵리스트

## ● Contains

```
140  boolean contains(T x) {  
141      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];  
142      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];  
143      int lFound = find(x, preds, succs);  
144      return (lFound != -1  
145              && succs[lFound].fullyLinked  
146              && !succs[lFound].marked);  
147  }
```

# 잠금 기반의 병행 스킵리스트

## ● 조감도



# 잠금 기반의 병행 스킵리스트

- 결과

- i7 - 920

- 연산횟수 32768

- 순차 SkipList(1Thread) : 0.37s

Thread	LazySkipLisy	Efficiency
1	0.094s	1
2	0.053s	0.885
4	0.026s	0.90
8	0.024s	0.48
16	0.020s	0.29
32	0.020s	0.14

# 실습

- 실습 : #23

- 게으른 동기화 스킵리스트를 구현하시오
- mutex로 `std::recursive_mutex`를 사용
  - 같은 스레드라면 여러 번 `lock()`을 호출할 수 있고, `lock()`을 호출한 회수 만큼 `unlock()`을 해줘야 잠금해제
- 지난 실습 (#22)에서 사용했던 벤치마크 프로그램을 사용하시오
- 스레드가 1개, 2개, 4개, 8개일 때의 속도를 비교하시오.
  - 각각 실행 전 Skip-List를 클리어 하시오
  - 각 단계마다 최초 20개의 원소를 출력하시오



# 숙제

- (SKIP) 숙제 14 :

- 게으른 동기화 SkipList의 구현

- 수업시간에 작성한 프로그램을 디버깅하여 완성하시오
      - 업로드한 샘플 참조 (버그가 많음)
    - mutex로 `std::recursive_mutex`를 사용
      - 같은 쓰레드라면 여러 번 `lock()`을 호출할 수 있고, `lock()`을 호출한 회수 만큼 `unlock()`을 해줘야 잠금해제

- 제출물

- .cpp 파일
    - 실행속도 비교표 (쓰레드 개수 별)
    - CPU의 종류 (모델명, 코어 개수, 클럭)

# 성긴 동기화 SKIP-LIST

- 결과 (2024)

	성긴동기화	Lock Free	성긴 SKLIST	Lazy SKLIST
1	1544	1469	653	1628
2	1730	1225	797	499
4	2278	1006	1091	324
8	12336	703	1444	347

싱글 쓰레드 : 2639

싱글 쓰레드 SKLIST : 1199

# 성긴 동기화

## ● 결과 (2023-월금)

	성긴 동기 화	세밀 한 동 기화	낙천적 동기화	게으른 동기화	Lock Free	성긴 SKLIST	게으른 SKLIST	LF SKLIST
1	2819	25310	4541	2787	1455	635	747	
2	2902	23125	3799	1934	1252	803	468	
4	3153	19000	2782	1298	1088	1144	363	
8	4111	14171	1709	904	760	1518	283	

싱글 쓰레드 : 2560

싱글 쓰레드 SKLIST : 590

# 스킵리스트

- 성능(화목)

- Intel® i7-4700HQ 2.2GHz, 4Core, Hyperthread, 16GB Memory

- Single Thread : 2029ms(list), 844ms(skiplist)

Thread	Coarse List	Lazy List	LF List	Coarse SKLIST	Lazy SKLIST	
1	2970		2532	984	1192	
2	2837		1737	1079	649	
4	3159		1251	1116	427	
8	3141		931	1418	339	
16	3137		954	1437	1937	

# 스킵리스트

- 성능(화목)

- Intel® i7-7700 3.6GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 1822ms(list), 658ms(skiplist)

Thr ead	Coarse List	Lazy List	LF List	Coarse SKLIST		
1	1986	2208	2173	960		
2	1857	1615	1780	882		
4	1994	1016	1232	916		
8	5716	643	895	1120		
16	7297	645	888	1127		

# 스킵리스트

- 성능(수목)

- Intel® i7-7700 3.6GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 1822ms(list), 658ms(skiplist)

Thr ead	Coarse List	Lazy List	LF List	Coarse SKLIST	Lazy SKLIST	
1	1986	2208	2173	960	1041	
2	1857	1615	1780	882	614	
4	1994	1016	1232	916	324	
8	5716	643	895	1120	249	
16	7297	645	888	1127	1552	

# 성긴 동기화 SKIP-LIST

## ● 결과 (2023-화금)

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	Lock Free	성긴 SKLIST	게으른 SKLIST
1	2334				1543	1177	1873
2	2640				1331	1343	1050
4	4126				1013	1536	416
8	17938				760	2826	330

싱글 쓰레드 : 1991

싱글 쓰레드 SKLIST : 992

# SKIP-LIST

---

- 소개
- 순차 스킵 리스트
- 병행 스킵 리스트
- Lock-Free 스킵 리스트



# LockFreeSkipList

- LockFreeSkipList

- 3 장의 LockFreeList를 사용
- 삽입과 삭제를 위해 CAS 사용
- 삭제 시에는 노드의 모든 next에 마킹을 해서 논리적으로 제거한다
- 물리적인 제거는 Find()에서 이루어 진다.
  - Find()는 리스트를 순회하면서 마킹된 노드를 만날 때마다 잘라내어서 마킹된 노드가 검색결과에 포함되지 않도록 한다.

# LockFreeSkipList

- 클래스 정의
  - LockFree List
    - 합성 포인터
  - 모든 next에 마킹을 추가
    - 고층의 마킹은 링크의 무효화를 뜻함.

```
1 public final class LockFreeSkipList<T> {
2     static final int MAX_LEVEL = ...;
3     final Node<T> head = new Node<T>(Integer.MIN_VALUE);
4     final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
5     public LockFreeSkipList() {
6         for (int i = 0; i < head.next.length; i++) {
7             head.next[i]
8                 = new AtomicMarkableReference<LockFreeSkipList.Node<T>>(tail, false);
9         }
10    }
11    public static final class Node<T> {
12        final T value; final int key;
13        final AtomicMarkableReference<Node<T>>[] next;
14        private int topLevel;
15        // constructor for sentinel nodes
16        public Node(int key) {
17            value = null; key = key;
18            next = (AtomicMarkableReference<Node<T>>[])
19                new AtomicMarkableReference[MAX_LEVEL + 1];
20            for (int i = 0; i < next.length; i++) {
21                next[i] = new AtomicMarkableReference<Node<T>>(null, false);
22            }
23            topLevel = MAX_LEVEL;
24        }
25        // constructor for ordinary nodes
26        public Node(T x, int height) {
27            value = x;
28            key = x.hashCode();
29            next = (AtomicMarkableReference<Node<T>>[])
30                new AtomicMarkableReference[height + 1];
31            for (int i = 0; i < next.length; i++) {
32                next[i] = new AtomicMarkableReference<Node<T>>(null, false);
33            }
34            topLevel = height;
35        }
36    }
37 }
```

# LockFreeSkipList

- Add()

- Find()를 이용하여 노드가 이미 리스트에 있는지 확인하고 자신의 앞노드(preds[])와 뒤노드(succs[])를 얻는다
- 추가되는 시점의 정의
  - 게으른 동기화 : 모든 리스트가 연결되면
  - LF 동기화 : 0 층이 연결되면
    - find를 할 때 0층 연결 되었는가의 여부를 true, false로 리턴
- 새 노드는 최하층 리스트에 연결하여 논리적으로 추가하고 그 다음에 순서대로 최상층까지 연결한다
  - 최상층
    - 새 노드가 고른 임의의 topLevel값이다.

# LockFreeSkipList

## ● Add()

```
36  boolean add(T x) {
37      int topLevel = randomLevel();
38      int bottomLevel = 0;
39      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
40      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
41      while (true) {
42          boolean found = find(x, preds, succs);
43          if (found) {
44              return false;
45          } else {
46              Node<T> newNode = new Node(x, topLevel);
47              for (int level = bottomLevel; level <= topLevel; level++) {
48                  Node<T> succ = succs[level];
49                  newNode.next[level].set(succ, false);
50              }
51              Node<T> pred = preds[bottomLevel];
52              Node<T> succ = succs[bottomLevel];
53              newNode.next[bottomLevel].set(succ, false);
54              if (!pred.next[bottomLevel].compareAndSet(succ, newNode,
55                                                         false, false)) {
56                  continue;
57              }
58              for (int level = bottomLevel+1; level <= topLevel; level++) {
59                  while (true) {
60                      pred = preds[level];
61                      succ = succs[level];
62                      if (pred.next[level].compareAndSet(succ, newNode, false, false))
63                          break;
64                      find(x, preds, succs);
65                  }
66              }
67              return true;
68          }
69      }
70  }
```

# LockFreeSkipList

## ● Remove()

- Find()를 호출하여 마킹되지 않고 대상키를 갖는 노드가 최하층리스트에 있는지 확인
- 최하층 리스트만 빼고 마킹을 해서 논리적으로 제거
- 최하층을 제외하고 전부 마킹했다면 최하층의 next에 마킹
  - 마킹에 성공하면 Find()를 통해 물리적 제거를 한다.
  - 실패하면 충돌했으므로 return false
    - 재시도??? 굳이??

# LockFreeSkipList

## ● Remove()

```

71  boolean remove(T x) {
72      int bottomLevel = 0;
73      Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
74      Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
75      Node<T> succ;
76      while (true) {
77          boolean found = find(x, preds, succs);
78          if (!found) {
79              return false;
80          } else {
81              Node<T> nodeToRemove = succs[bottomLevel];
82              for (int level = nodeToRemove.topLevel;
83                  level >= bottomLevel+1; level--) {
84                  boolean[] marked = {false};
85                  succ = nodeToRemove.next[level].get(marked);
86                  while (!marked[0]) {
87                      nodeToRemove.next[level].attemptMark(succ, true);
88                      succ = nodeToRemove.next[level].get(marked);
89                  }
90              }
91              boolean[] marked = {false};
92              succ = nodeToRemove.next[bottomLevel].get(marked);
93              while (true) {
94                  boolean iMarkedIt =
95                      nodeToRemove.next[bottomLevel].compareAndSet(succ, succ,
96                                                                      false, true);
97                  succ = succs[bottomLevel].next[bottomLevel].get(marked);
98                  if (iMarkedIt) {
99                      find(x, preds, succs);
100                     return true;
101                 }
102                 else if (marked[0]) return false;
103             }
104         }
105     }
106 }

```

# LockFreeSkipList

## ● Find()

- 스킵리스트를 순회하는데 각 층을 내려가며 리스트를 진행한다.
- 진행 중에 표시된 링크(marked)를 만나면 이 링크를 잘라내어서 표시된 노드의 키값을 절대 보지 않는다.
  - 모든 층의 링크를 다 잘라내지는 않으므로, 정확한 정의대로의 스킵리스트는 유지하지 못한다. 하지만 검색의 정확성에는 문제가 없다.

# LockFreeSkipList

## ● Find()

- LockFreeSkipList를 검색하고 대상키가 집합에 있을 때만 true를 반환
  - 0 레벨에 존재하고 마킹이 되지 않은 것을 집합에 존재한다고 정의
- 두가지 속성을 만족
  - 표시된 연결(marked)은 순회하지 않는다. 대신 해당 층의 리스트에서 표시된 연결이 참조된 노드를 제거



# LockFreeSkipList

- Find()

```
107 boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
108     int bottomLevel = 0;
109     int key = x.hashCode();
110     boolean[] marked = {false};
111     boolean snip;
112     Node<T> pred = null, curr = null, succ = null;
113     retry:
114     while (true) {
115         pred = head;
116         for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
117             curr = pred.next[level].getReference();
118             while (true) {
119                 succ = curr.next[level].get(marked);
120                 while (marked[0]) {
121                     snip = pred.next[level].compareAndSet(curr, succ,
122                                                             false, false);
123                     if (!snip) continue retry;
124                     curr = pred.next[level].getReference();
125                     succ = curr.next[level].get(marked);
126                 }
127                 if (curr.key < key) {
128                     pred = curr; curr = succ;
129                 } else {
130                     break;
131                 }
132             }
133             preds[level] = pred;
134             succs[level] = curr;
135         }
136         return (curr.key == key);
137     }
138 }
```

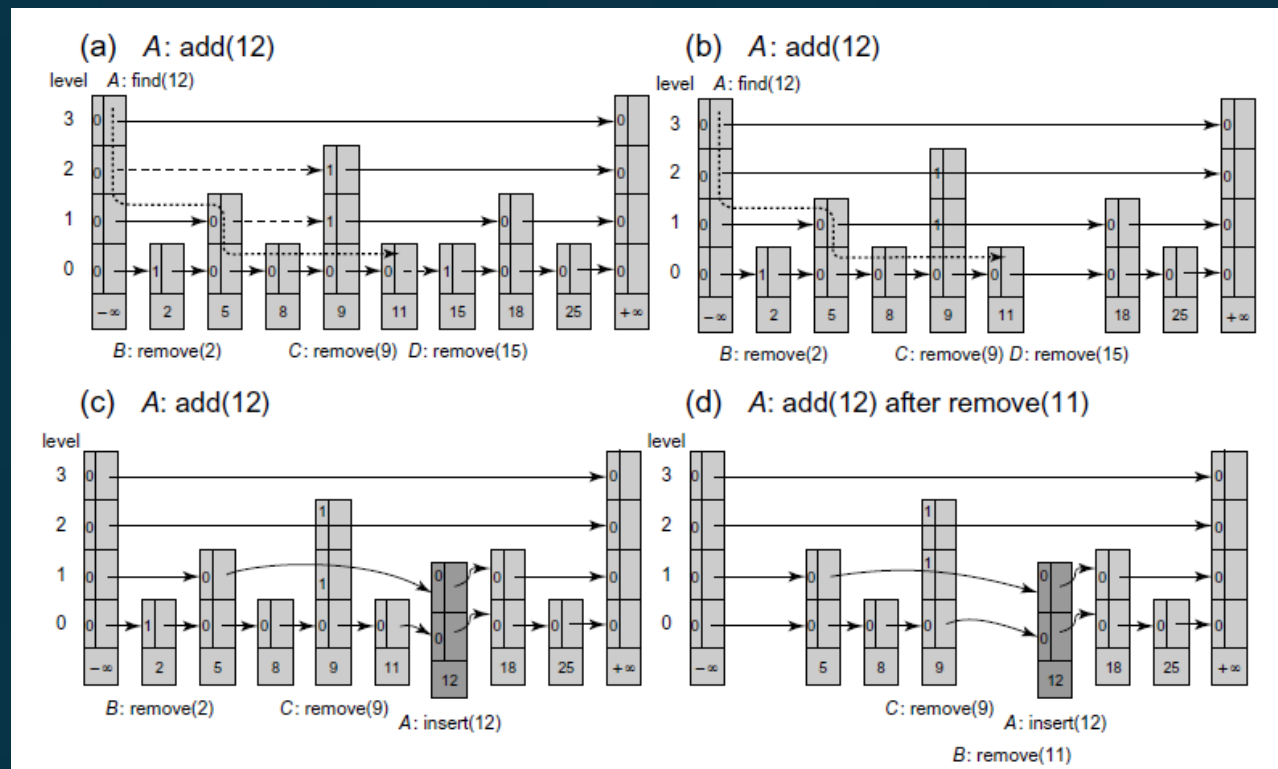
# LockFreeSkipList

## ● Contains()

```
139  boolean contains(T x) {
140      int bottomLevel = 0;
141      int v = x.hashCode();
142      boolean[] marked = {false};
143      Node<T> pred = head, curr = null, succ = null;
144      for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
145          curr = pred.next[level].getReference();
146          while (true) {
147              succ = curr.next[level].get(marked);
148              while (marked[0]) {
149                  curr = pred.next[level].getReference(); // curr.next로 변경
150                  succ = curr.next[level].get(marked);
151              }
152              if (curr.key < v) {
153                  pred = curr;
154                  curr = succ;
155              } else {
156                  break;
157              }
158          }
159      }
160      return (curr.key == v);
161  }
```

# LockFreeSkipList

## ● LockFreeSkipList



# LockFreeSkipList

- 실습 #24

- Lock Free Skip List 프로그램을 작성한다.
- 순차 Skip List와 속도 비교
- Lock Free 일반 List와 속도 비교
- 자료실의 샘플 프로그램 참조

# 숙제 #19

- Lock Free Skip List 프로그램을 완성한다.
  - 강의자료에 올려둔 미완성 소스코드를 완성한다.
    - 오류 검사 루틴 사용
- 성긴 동기화, 게으른 동기화 Skip List와 속도 비교
- Lock Free 일반 List와 속도 비교
- 실습시간에 확인받은 수강생은 제출하지 말것.

# 성긴 동기화 SKIP-LIST

- 결과 (2024)

	성긴동기화	Lock Free	성긴 SKLIST	Lazy SKLIST	LF SK LIST
1	1544	1469	653	1628	883
2	1730	1225	797	499	545
4	2278	1006	1091	324	387
8	12336	703	1444	347	287

싱글 쓰레드 : 2639

싱글 쓰레드 SKLIST : 1199

# 스킵리스트

- 성능(화목)

- Intel® i7-4700HQ 2.2GHz, 4Core, Hyperthread, 16GB Memory

- Single Thread : 2029ms(list), 844ms(skiplist)

Thread	Coarse List	Lazy List	LF List	Coarse SKLIST	Lazy SKLIST	LF SKLIST
1	2970		2532	984	1192	877
2	2837		1737	1079	649	527
4	3159		1251	1116	427	336
8	3141		931	1418	339	189
16	3137		954	1437	1937	192

# 스킵리스트

- 성능(수목)

- Intel® i7-7700 3.6GHz, 4Core, Hyperthread, 64GB Memory
- Single Thread : 1822ms(list), 658ms(skiplist)

Thread	Coarse List	Lazy List	LF List	Coarse SKLIST	Lazy SKLIST	LF SKLIST
1	1986	2208	2173	960	1041	806
2	1857	1615	1780	882	614	492
4	1994	1016	1232	916	324	262
8	5716	643	895	1120	249	186
16	7297	645	888	1127	1552	181



# 성긴 동기화

## ● 결과 (2023-월금)

	성긴 동기 화	세밀 한 동 기화	낙천적 동기화	게으른 동기화	Lock Free	성긴 SKLIST	게으른 SKLIST	LF SKLIST
1	2819	25310	4541	2787	1455	635	935	948
2	2902	23125	3799	1934	1252	803	583	622
4	3153	19000	2782	1298	1088	1144	429	428
8	4111	14171	1709	904	760	1518	358	320

싱글 쓰레드 : 2560

싱글 쓰레드 SKLIST : 590

# 성긴 동기화 SKIP-LIST

## ● 결과 (2023-화금)

	성긴동기화	세밀한 동기화	낙천적 동기화	게으른 동기화	Lock Free	성긴 SKLIST	게으른 SKLIST	Lock Free SKILIST
1	2334				1543	1177	1873	1197
2	2640				1331	1343	1050	641
4	4126				1013	1536	416	427
8	17938				760	2826	330	302

싱글 쓰레드 : 1991

싱글 쓰레드 SKLIST : 992

# EBR??

- 메모리 재사용

- EBR을 사용하면 된다.
- 교재에 있는 알고리즘은 오류 존재
  - Remove가 깔끔하게 되지 않는 오류 발생
  - 수정필요
    - 대학원 과정
    - Eclass에 EBR 적용된 코드 업로드

# 정리

- non-blocking Stack
  - Back Off의 유용성 (Core가 많은 머신에서)
  - 부하 분산テクニック
    - Elimination
- non-blocking 검색 자료구조
  - SKIP-LIST
    - 평균  $O(\log n)$ 의 검색 시간

# 총정리

- 왜 멀티쓰레드 프로그래밍을 해야 하는가?
- 멀티쓰레드 프로그램은 멀티코어 CPU에서 어떻게 실행되는가?
- 멀티쓰레드 프로그래밍 API는 무엇인가?
- 내가 작성한 멀티쓰레드 프로그램이 왜 죽는가?
- Lock을 쓰면 왜 느린가?
- Lock을 제거했더니 왜 죽는가?

# 총정리

- Lock을 사용하지 않을 때 스레드간의 동기화는 어떻게 구현 하는 것이 좋은가?
- Non-Blocking 알고리즘이 무엇인가?
  - 어떻게 구현해야 하는가?
  - Blocking보다 왜 좋은가? 어떤 경우에 좋은가?
- Lock-Free 알고리즘의 구현이 왜 어려운가?