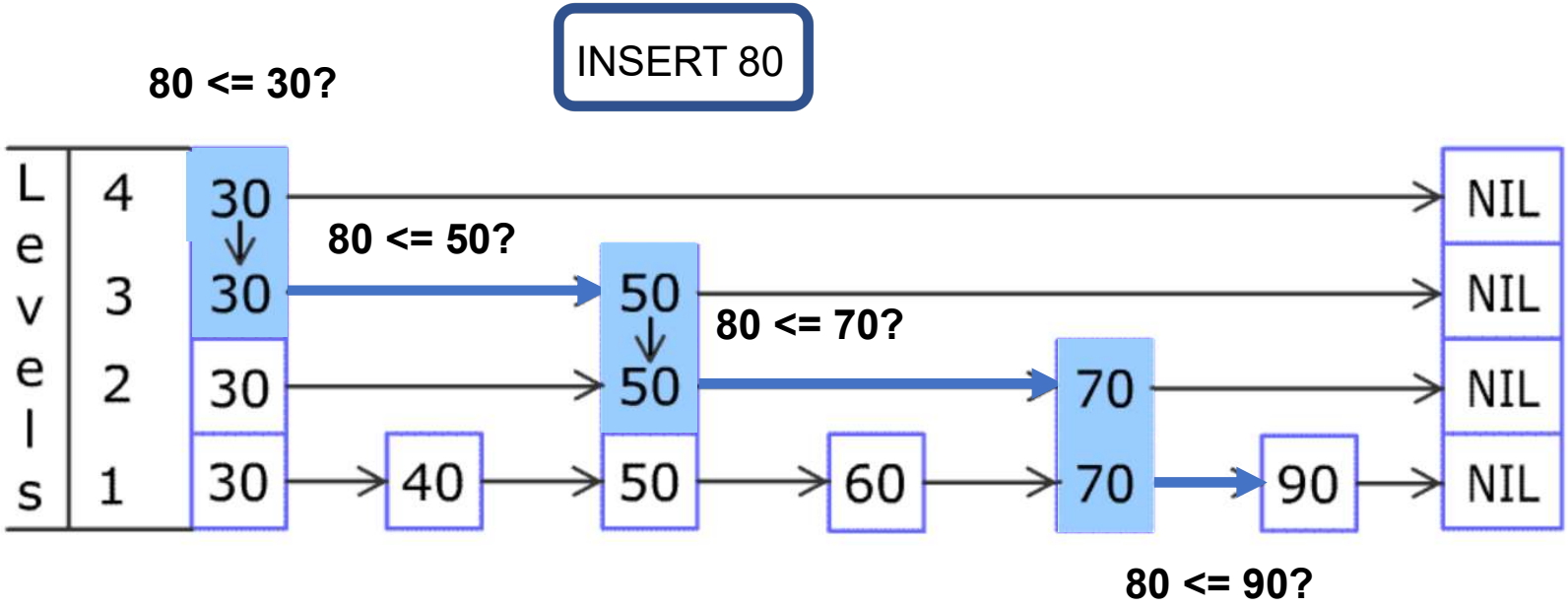
Memtable --Skiplist

Skiplist

- Node
- 변수
- Insert
- Find
- FindGreaterOrEqual

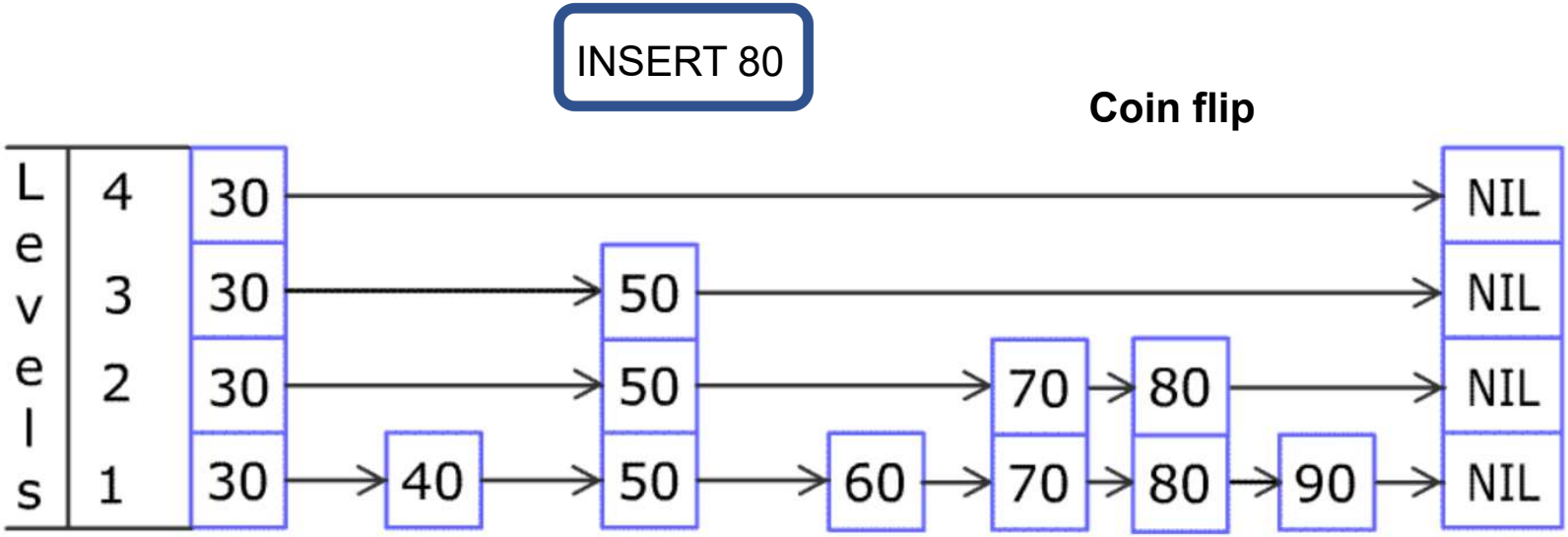
Skiplist

How to insert new element in skiplist?



Skiplist

How to insert new element in skiplist?



Skiplist --Node

```
template <typename Key, class Comparator>
struct SkipList<Key, Comparator>::Node {
  explicit Node(const Key& k) : key(k) {}

  Key const key;

  // Accessors/mutators for links.  Wrapped in methods so we can
  // add the appropriate barriers as necessary.
  Node* Next(int n) {
    assert(n >= 0);
    // Use an 'acquire load' so that we observe a fully initialized
    // version of the returned Node.
    return next_[n].load(std::memory_order_acquire);
  }
  void SetNext(int n, Node* x) {
    assert(n >= 0);
    // Use a 'release store' so that anybody who reads through this
    // pointer observes a fully initialized version of the inserted node.
    next_[n].store(x, std::memory_order_release);
  }

  // No-barrier variants that can be safely used in a few locations.
  Node* NoBarrier_Next(int n) {
    assert(n >= 0);
    return next_[n].load(std::memory_order_relaxed);
  }
```

Node에서는 key를 저장

지정한 level에서 해당node의 next node를 load

지정한 level에서 해당node의 next node를 x로 지정

Skiplist --변수

```
private:
 enum { kMaxHeight = 12 };
```

Skiplist의 내부 최대level

```
Comparator const compare_;
```

Skiplist의 내부 key크기를 compare

```
Arena* const arena_;  // Arena used for allocations of nodes
```

Memory부배

```
Node* const head_;
```

Skiplist의 sentinel node

```
std::atomic<int> max_height_;
```

현재skiplist가 사용중인 최대 level

Skiplist --insert

```
template <typename Key, class Comparator>
void SkipList<Key, Comparator>::Insert(const Key& key) {
  // TODO(opt): We can use a barrier-free variant of FindGreaterOrEqual()
  // here since Insert() is externally synchronized.
  Node* prev[kMaxHeight];
  Node* x = FindGreaterOrEqual(key, prev);

  // Our data structure does not allow duplicate insertion
  assert(x == nullptr || !Equal(key, x->key));

  int height = RandomHeight();
  if (height > GetMaxHeight()) {
    for (int i = GetMaxHeight(); i < height; i++) {
      prev[i] = head_;
    }
    // It is ok to mutate max_height_ without any synchronization
    // with concurrent readers.  A concurrent reader that observes
    // the new value of max_height_ will see either the old value of
    // new level pointers from head_ (nullptr), or a new value set in
    // the loop below.  In the former case the reader will
    // immediately drop to the next level since nullptr sorts after all
    // keys.  In the latter case the reader will use the new node.
    max_height_.store(height, std::memory_order_relaxed);
  }

  x = NewNode(key, height);
  for (int i = 0; i < height; i++) {
    // NoBarrier_SetNext() suffices since we will add a barrier when
    // we publish a pointer to "x" in prev[i].
    x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
    prev[i]->SetNext(i, x);
  }
}
```

→ Prev node선언. =insert할 key의 앞 node

→ Insert key보다 크거나 같은 키의 자리를 찾기

→ Insert key가 skiplist내에서 가질 level을 random으로 설정

→ ?Random으로 나온값이 GetMaxHeight보다 클때 큰 부분을 head에 연결?

→ Max_height를 갱신

→ Insert할 node를 생성

→ 먼저 x node의 next node를 prev의 next node로 지향

→ Prev를 x node에 지향

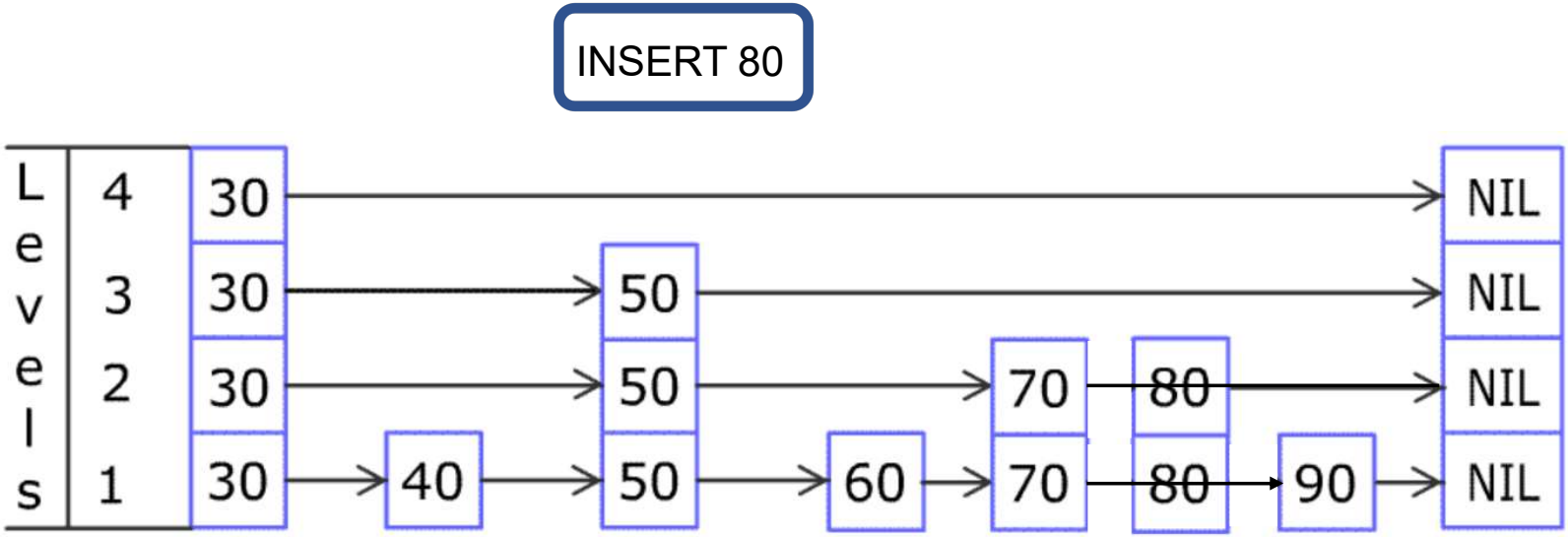
Skiplist --insert

```
x = NewNode(key, height);
for (int i = 0; i < height; i++) {
  // NoBarrier_SetNext() suffices since we will add a barrier when
  // we publish a pointer to "x" in prev[i].
  x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
  prev[i]->SetNext(i, x);
}
```

Insert할 node를 생성

먼저 x node의 next node를 prev의 next node로 지향

Prev를 x node에 지향



Skiplist --find

```
template <typename Key, class Comparator>
bool SkipList<Key, Comparator>::Contains(const Key& key) const {
  Node* x = FindGreaterOrEqual(key, nullptr);
  if (x != nullptr && Equal(key, x->key)) {
    return true;
  } else {
    return false;
  }
}
```

해당key보다 큰 key를 소지한 첫 번째node를 찾아 key를 비교

Skiplist --FindGreaterOrEqual

```
template <typename Key, class Comparator>
typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindGreaterOrEqual(const Key& key,
                                              Node** prev) const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    Node* next = x->Next(level);
    if (KeyIsAfterNode(key, next)) {
      // Keep searching in this list
      x = next;
    } else {
      if (prev != nullptr) prev[level] = x;
      if (level == 0) {
        return next;
      } else {
        // Switch to next list
        level--;
      }
    }
  }
}
```

Skiplist의 level은 0부터 시작

해당level의 next node로 지향

Key가 해당구간에 없다면 next로 계속 반복

Key가 해당구간에 있다면 level이 0 될 대까지 level – 1 & prev를 다시 지정해서 지정한 key 보다 큰 키를 가지고 있는 첫번째 node를 찾기

Q&A

감사합니다

Thank you!