CSCI-GA.3033-017
Special Topics:
Multicore Programming

Lecture 8
Synchronized Structures Part 2

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me

Homework 2 Review
Thread-Safe Aggregation

```
static int sum_stat_a = 0;
static int sum_stat_b = 0;
static int sum_stat_c = 0;
int aggregateStats(int stat_a, int stat_b, int stat_c) {
	sum_stat_a += stat_a;
	sum_stat_b += stat_b;
	sum_stat_c -= stat_c;
	return sum_stat_a + sum_stat_b + sum_stat_c;
}
void init(void) { }
```

1. Use a single pthread mutex to make this function thread-safe. Add global variables and content to the `init()` function as necessary.
2. Modify the original code from to make it thread-safe, but use two mutices this time, one each for `sum_stat_a`, `sum_stat_b`, and `sum_stat_b`.

Homework 2 Review
Aggregation (1 Lock)

```
pthread_mutex_t stats_mutex;
static double sum_stat_a = 0;
static double sum_stat_b = 0;
static double sum_stat_c = 10000;
int aggregateStats(double stat_a, double stat_b, double stat_c) {
        pthread_mutex_lock(&stats_mutex);
        sum_stat_a += stat_a;
        sum_stat_b += stat_b;
        sum_stat_c -= stat_c;
        int rval = sum_stat_a + sum_stat_b + sum_stat_c;
        pthread_mutex_unlock(&stats_mutex);
        return rval;
}
void init(void) {
        pthread_mutex_init(&stats_mutex, NULL);
}
```

Invariant: The returned value from aggregateStats() must be equal to exactly the sum of the statistics at exactly the moment in time when the new additions were aggregated into each sum.

Homework 2 Review
Aggregation (3 Locks)

Invariant: The returned value from aggregateStats() must be equal to exactly the sums of statistic A, B, and C, each taken at the moment in time when the new addition was aggregated into **each** sum. The aggregate sum may therefore represent a value covering three close (but different) time periods.

`sum_stat_a`: `stat_a_mutex`
`sum_stat_b`: `stat_b_mutex`
`sum_stat_c`: `stat_c_mutex`

Homework 2 Review
Aggregation (3 Locks)

```c
pthread_mutex_t stat_a_mutex;
pthread_mutex_t stat_b_mutex;
pthread_mutex_t stat_c_mutex;
static int sum_stat_a = 0;
static int sum_stat_b = 0;
static double sum_stat_c = 10000;

void init(void) {
    pthread_mutex_init(
        &stat_a_mutex, NULL);
    pthread_mutex_init(
        &stat_b_mutex, NULL);
    pthread_mutex_init(
        &stat_c_mutex, NULL);
}
```

```c
int aggregateStats(
    int stat_a, int stat_b, int stat_c)
{
    int rval = 0;
    pthread_mutex_lock(&stat_a_mutex);
    sum_stat_a += stat_a;
    rval += sum_stat_a;
    pthread_mutex_unlock(&stat_a_mutex);
    pthread_mutex_lock(&stat_b_mutex);
    sum_stat_b += stat_b;
    rval += sum_stat_b;
    pthread_mutex_unlock(&stat_b_mutex);
    pthread_mutex_lock(&stat_c_mutex);
    sum_stat_c -= stat_c;
    rval += sum_stat_c;
    pthread_mutex_unlock(&stat_c_mutex);
    return rval;
}
```

Outline

- `volatile` & Thread-Safe C++ Code
- A Lock-Free Hash Table

Interlude: `volatile` and Thread-Safe C++ Code

- What does `volatile` mean in C?
  - Don't assume this value can't be changed elsewhere
- `volatile` means the same thing in C++
  - Rabid arguments about using volatile in C++ eg: [`http://stackoverflow.com/questions/4557979`](http://stackoverflow.com/questions/4557979)
  - No memory fences, no order of execution
  - Some hardware may guarantee *tearing* does not occur
    - X86: Cache coherent
- Therefore: `volatile` useful only for rare single-word, many-write, "last to write wins" scenario w/ many cores

Building Block:
Concurrent Ordered List

Concurrent Ordered Lists

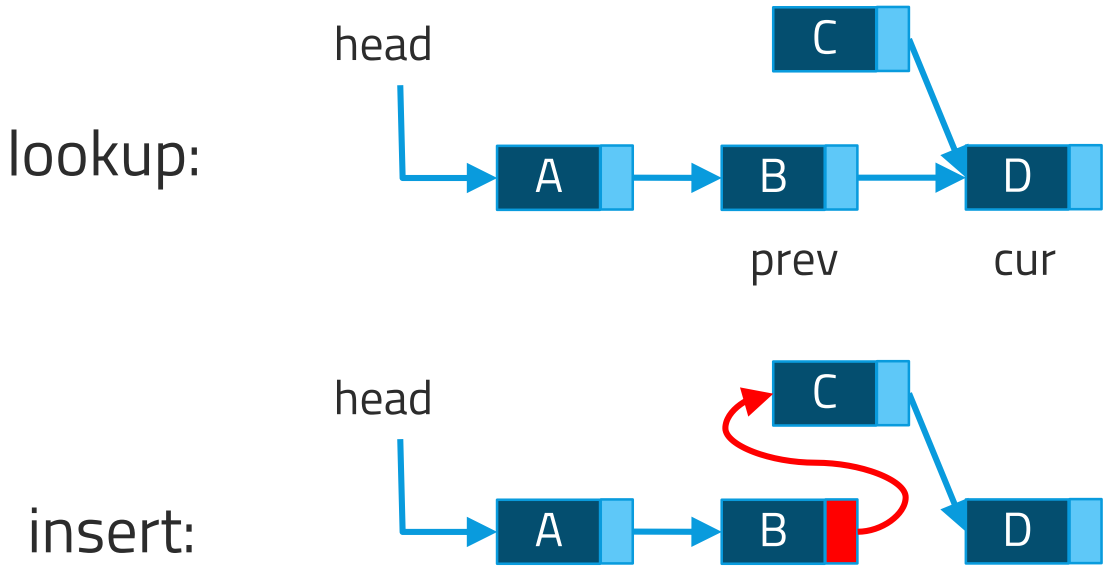
- Idea: Build more complex data structures using thread-safe ordered lists
- Want a finer grained lock than the whole list
  - What's so hard about concurrent operations over lists?
- Let's explore:
  1. A lock-based approach
  2. A lock-free approach, based on our queue

List-Based Set

- List is ordered
  - Inserts between any nodes
- Interface (looks roughly familiar?)
  - `bool insert(key)`: true if key didn't exist yet
  - `bool lookup(key)`: true if key exists already
  - `bool remove(key)`: true if key existed and was erased
- Each function needs to locate key first
  - Proposed:
    `(node* prev, node* cur, node* next, bool found) lookup(key)`

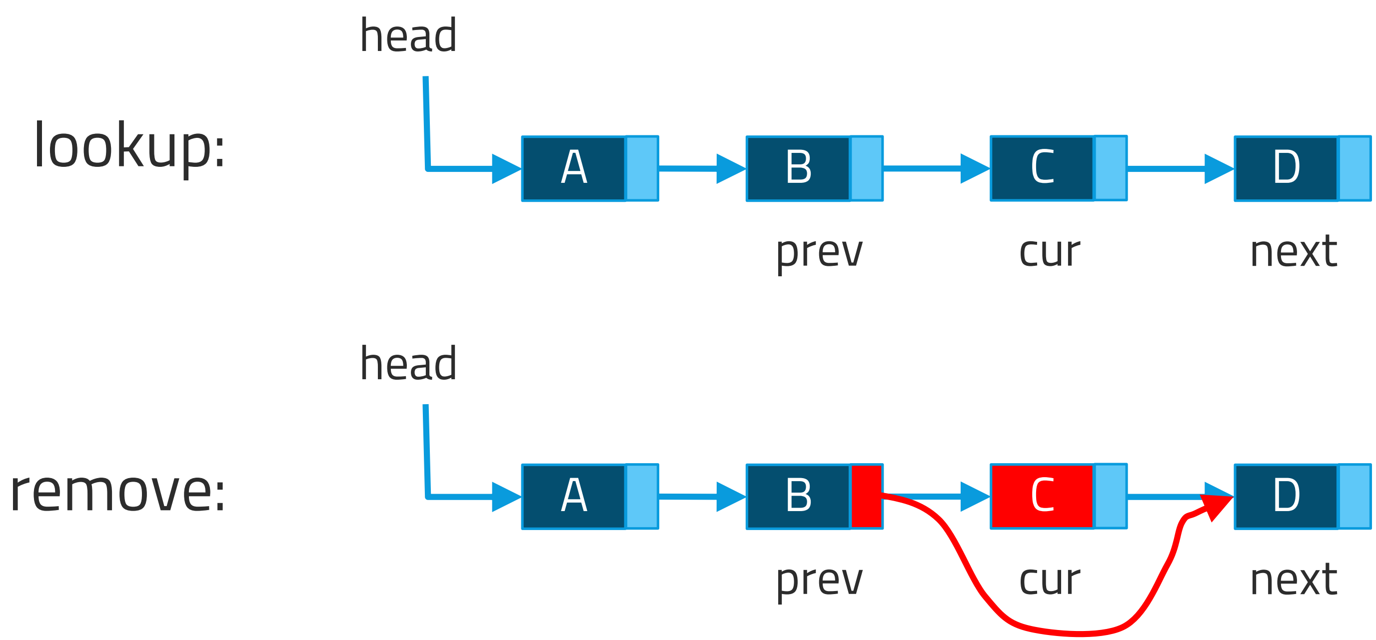
Lookup for Insertion

- Only prev's next really changes.
- Note: "head" is on the left, to make it clearer that this sorted list stores strictly increasing keys left-to-right



Lookup for Removal

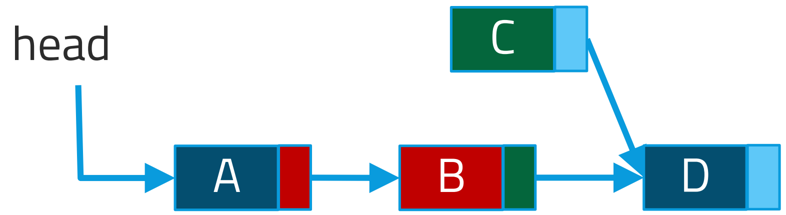
- Again, only prev’s next really changes
- Tempting: only lock prev for update



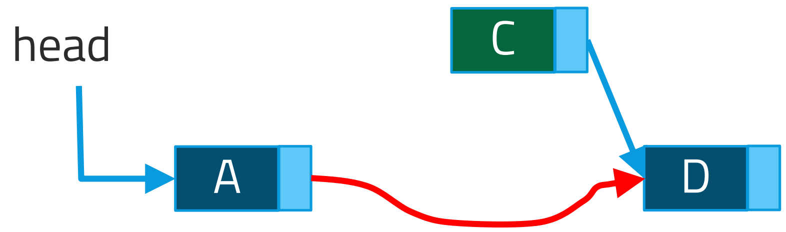
Hazard with Locks

- Concurrent deletion of ‘B’ and insertion of ‘C’
- Naïve approach: Incorrect result!
- Hazardous deletion/deletion hazard as well

lookup x2:



result:

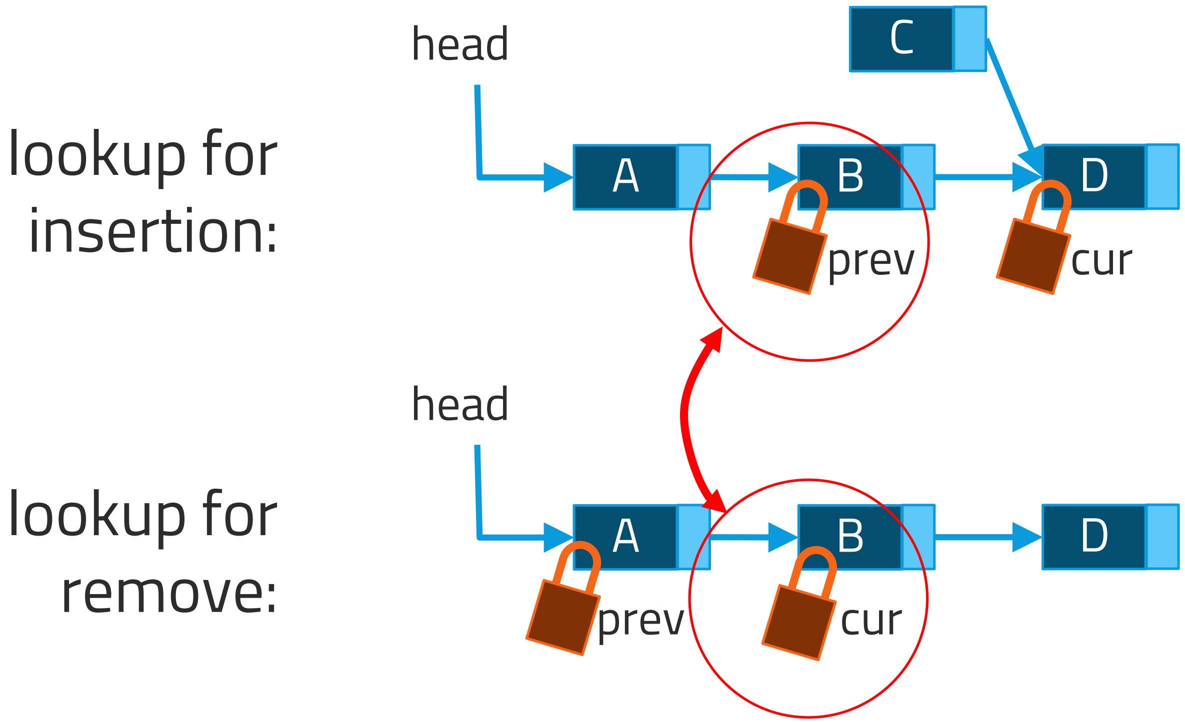


Lock Coupling

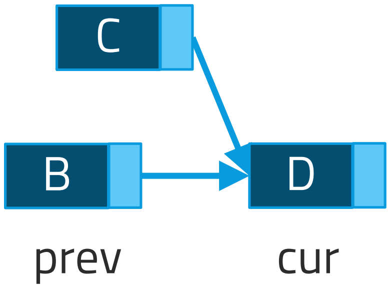
- Need to lock all nodes involved
  - On insertion, new node placed between prev and cur
  - On remove, prev node's pointer changes; cur disappears
- Acquire prev lock, then cur lock, during lookup.
- Lock on lookup, not the actual insertion/deletion!
- Still better than locking entire list

Hazard Removed with Lock Coupling

- Concurrent deletion of ‘B’ and insertion of ‘C’
- Lock-free version with CAS?



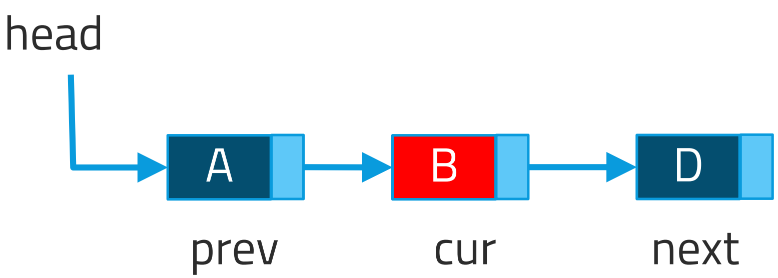
Lock-Free Insertion




```
while(true) {
    if find(list, key) {
        return false // sets prev,cur
    }
    new_node->next = cur;
    if CAS(&prev->next, cur, new_node) {
        return true
    }
}
```

Logical Deletion

1. Mark node deleted
2. Fix previous pointer

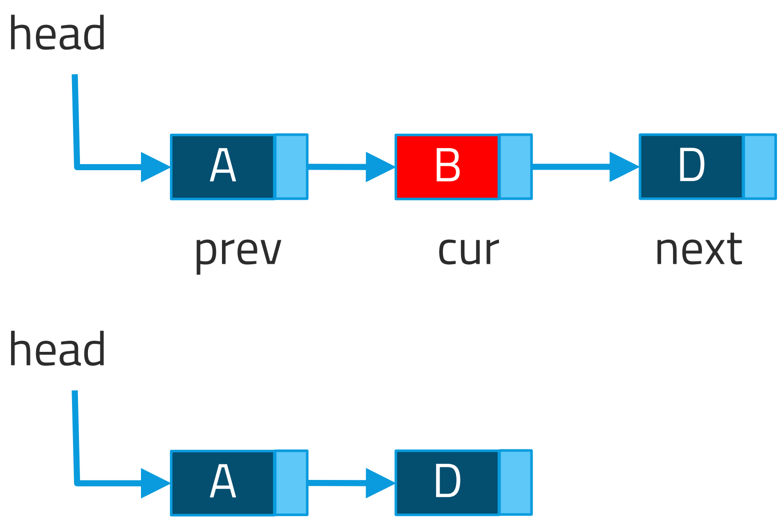


After step 1, want to make it that C can't be inserted after B since it's marked deleted. How?

Deletion Mark

- Combine next pointer and deletion mark (bit)
- All pointers are aligned!
  - Steal up to last 2 bits (for 32-bit pointers) or 3 bits (for 64-bit pointers)
- If the deletion bit is set a CAS against the original pointer (with low bit(s) reset) will fail
  - Works well for insert-delete hazards

Lock-Free Deletion




```
while(1) {
    if (!find(list, key)) {
        // sets prev,cur,next
        return false
     }
    if (CAS(&cur.mark, 0, 1)) {
        break;
     }
}
if (CAS(&prev->next, cur, next)) {
     dealloc(cur);
}
return true
```

- Decouples logical and physical deletions via marker hidden in pointer
- Future find()s may see logically deleted nodes!

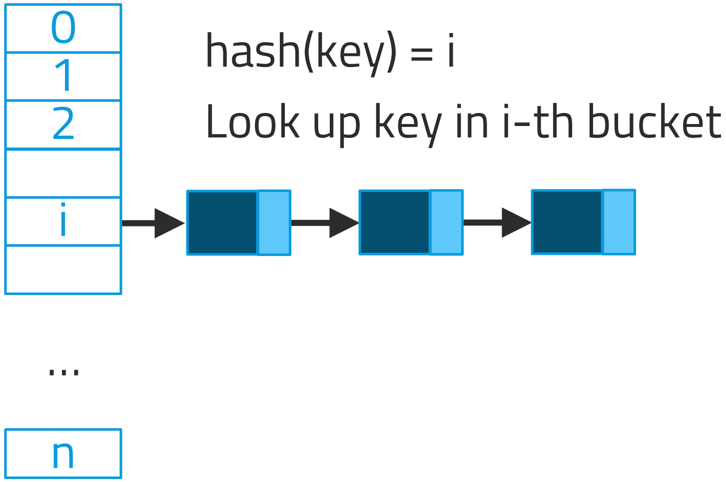
Lock-Free Ordered List Summary

- Now have a basic building block for more complex structures
- Memory management notes
  - ABA problem recurs
  - Same solution: pointer-counter pairs

Concurrent Hash Tables

Concurrent (Locked) Hashing

- Closed-addressing hashing schemes
  - Put colliding elements in the same bucket
  - May need to change bucket count as more elements are hashed
- Open-addressing works too, but beyond the scope today



Hash Table Lock Striping

- Approach: lock striping
  - Good results in practice
- Design pointers
  - Successful hash schemes maintain small buckets
  - No operation on the hash table affects all buckets
    - No *normal* operation (resizing…)
  - Each bucket could have its own lock (granularity)
  - Lookup needs locking! (why?)

Hash Table: Array of Lock-Free Lists

- Use lock-free list as a bucket
- Same-bucket concurrent operations allowed
- Lookup is lock-free because of atomic CAS use
  - But not completely "free"

Hashing and Resizing

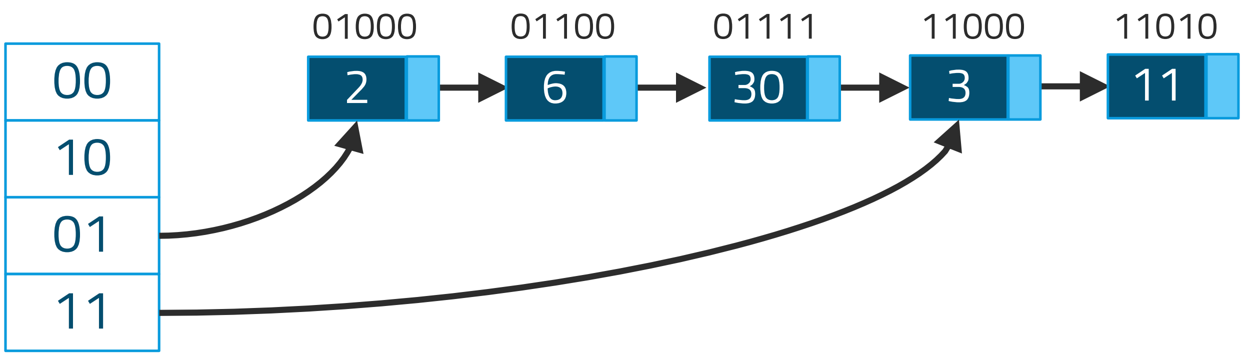
- With lock striping
  - Resizing changes the number of buckets so it requires rehashing of all members
  - Grab all locks for resizing
  - Good results in practice but we want to use our lock-free lists
- Lock-free list
  - Can't grab locks to resize

Hashing with Modulo

- First, hash keys using a good hashing algorithms
- Key hashes that share the same last $\log_2 K$ bits have the same value modulo K
  - Eg, let K = 4 ($\log_2 K = 2$).
  - 2 % 4 = 30 % 4 = 6 % 4 = 2 (binary 10)
  - 3 % 4 = 11 % 4 = 3 (binary 11)
- Ordering keys within a bucket
  - Reverse the bits!
  - Eg, bucket 0b10 with K=4, holding 2, 30, and 6
  - 01000 (2), 01100 (6), 01111 (30)

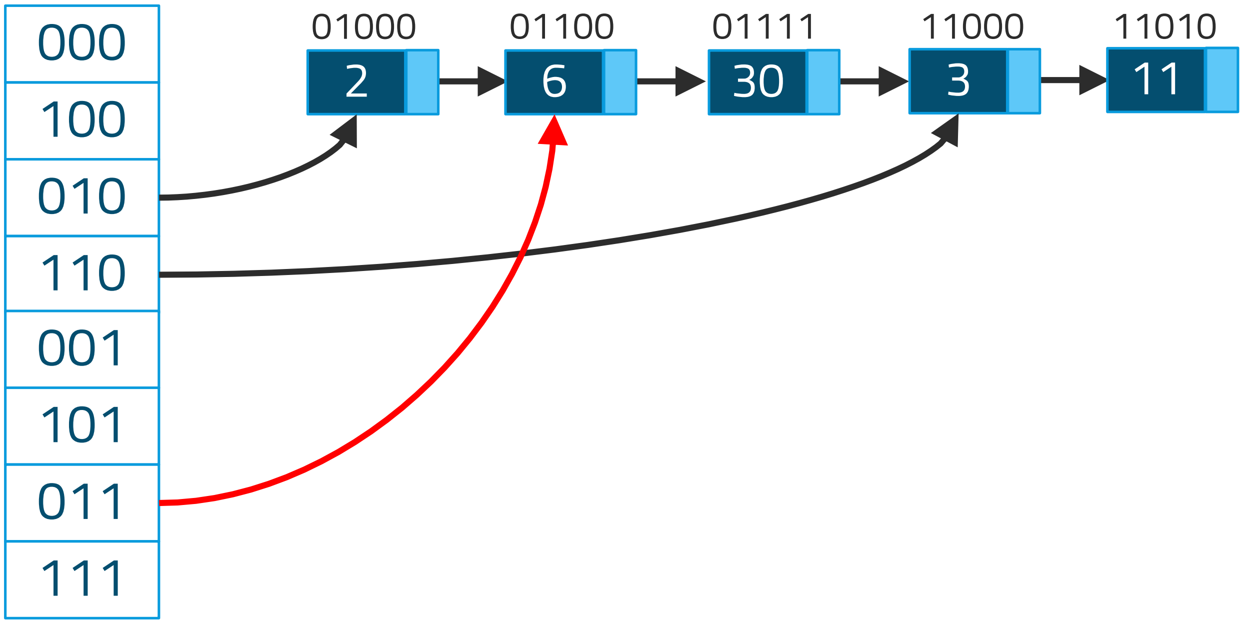
Hashing with Modulo: Split Ordered List

- Combine buckets into single ordered list
- Eg, buckets 0b10 and 0b11 for K=4
  - 0b10: 01000 (2), 01100 (6), 01111 (30)
  - 0b11: 11000 (3), 11010 (11)



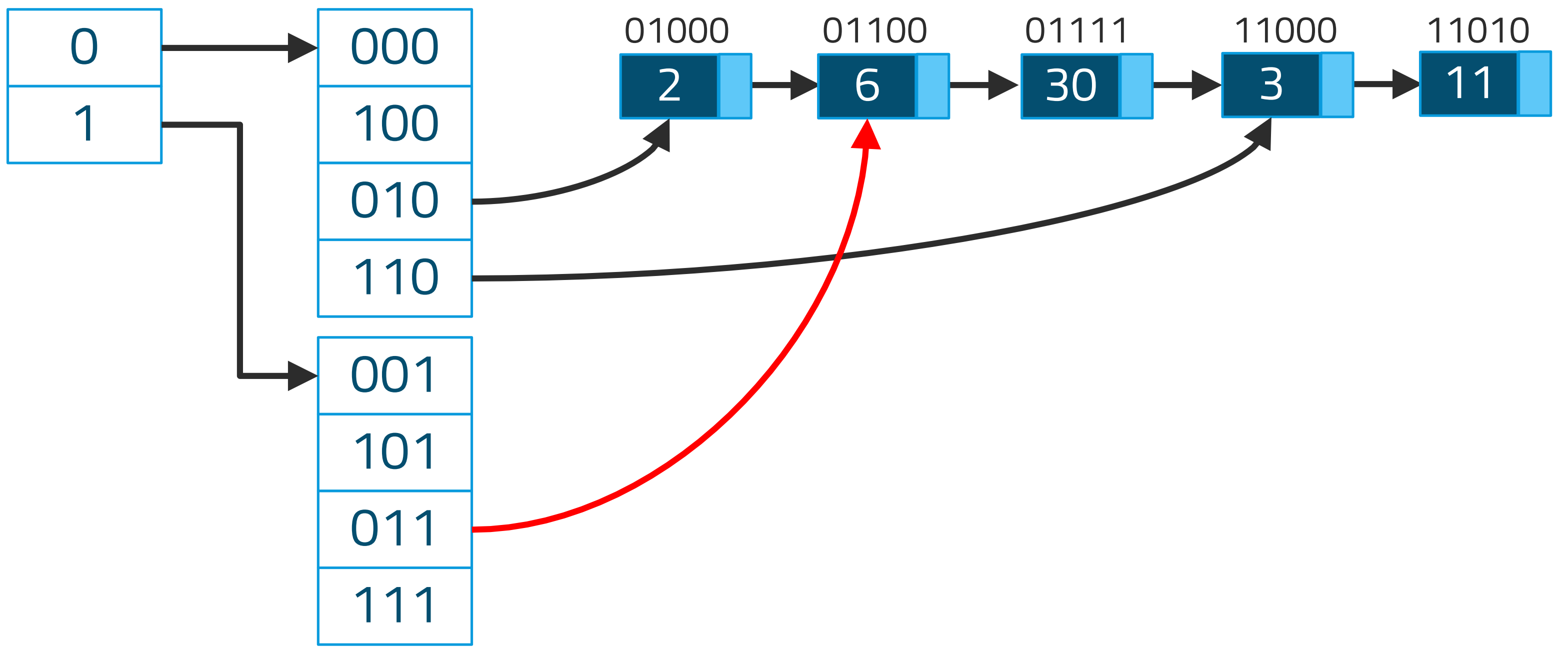
Resizing Split Ordered List

- Buckets can split, but the list stays the same
- Eg: K=4 -> K = 8: Use last $\log_2 8$ = 3 bits



Concurrency During Resize

- Add additional layer(s) of indirection
- Skiplist



Conclusion

- Concurrent ordered lists can be used for hashing and searching structures
- The ordered lists we saw today:
  - Decoupled deletion into two operations for lock-free operations
  - Allowed faster lookup but required locks for other operations
- Concurrent hash table
  - Different concurrency schemes can be combined in one structure
  - Difficult to show correctness