OS for Database systems
Linux and OS

Hojin Shin

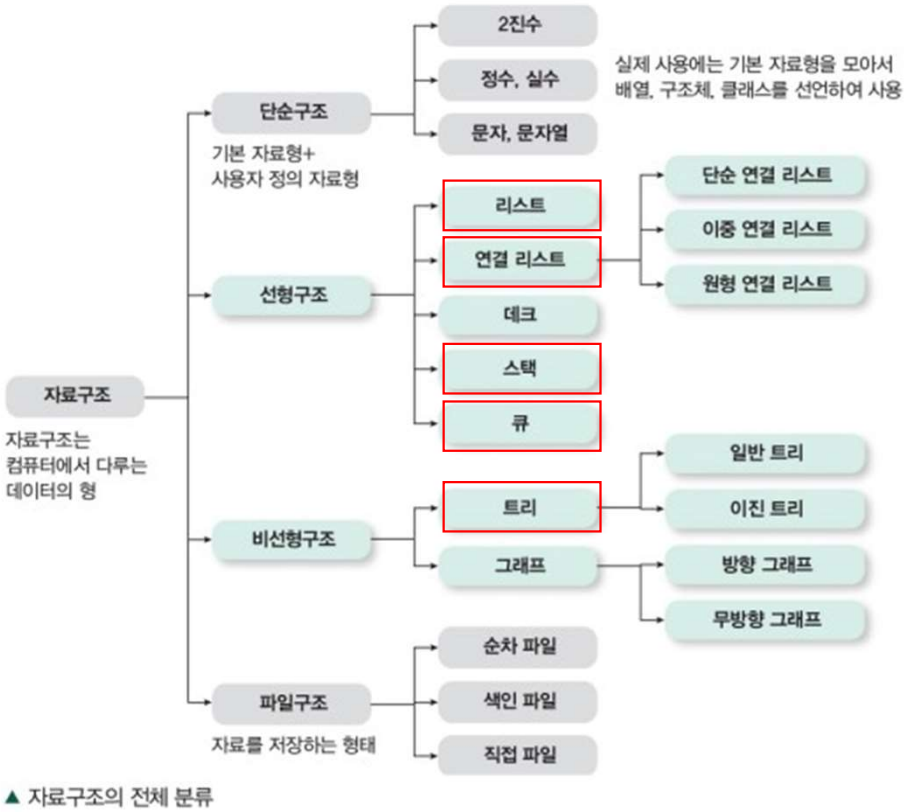
Dankook University

Index

- Lock Progamming
  - 동시성 자료구조
    - 실습 5: Hash, Queue
      - Single/Multi Thread
      - Coarse-grained, Fine-grained Lock
      - Lock overhead with experiment results

Lock: Data Structure

- Various Data Structures
  - Data vs Information
  - The way store data
  - Important to choose data structure what to use



▲ 자료구조의 전체 분류

Lock: Data Structure

- Time Complexity

	Search	Insertion	Deletion
Array	O(n)	O(n)	O(n)
Stack	O(n)	O(1)	O(1)
Queue	O(n)	O(1)	O(1)
Linked-List	O(n)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)
Binary Search Tree	O(logN)	O(logN)	O(logN)
B-Tree	O(logN)	O(logN)	O(logN)

Lock: Data Structure

Time Complexity VS Space Complexity

Lock: Data Structure

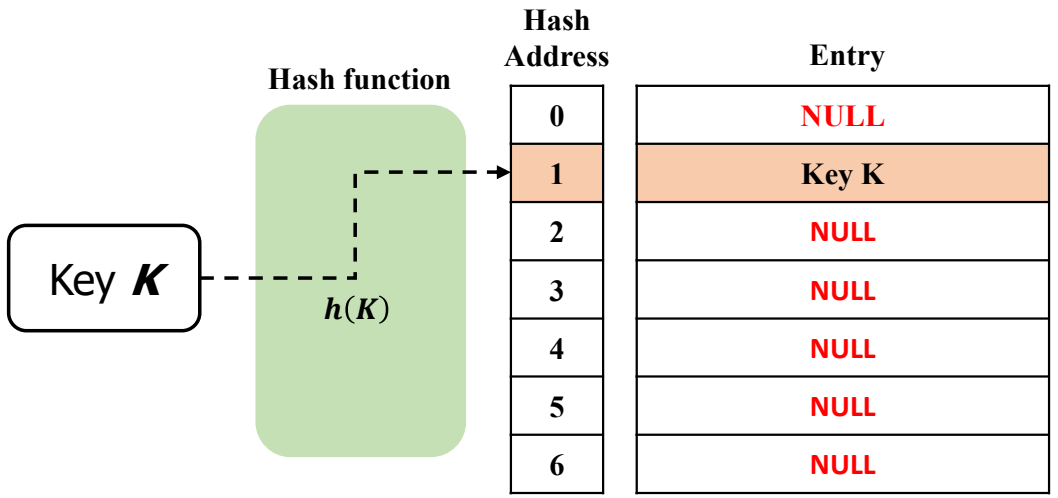
Time Complexity VS Space Complexity

Run Time

Memory Consumption

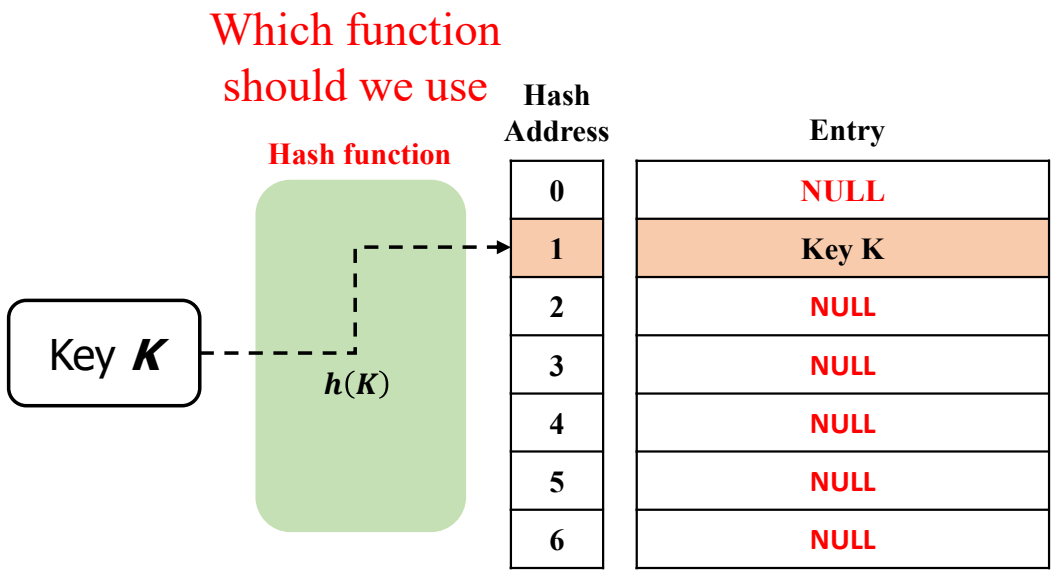
Lock: Hash

- Hash
  - Data management
  - Fast for store and lookup
  - O(1) time complexity



Lock: Hash

- Hash
  - Data management
  - Fast for store and lookup
  - O(1) time complexity



Lock: 실습

- Hash Function
  - Input: Data of any length
  - Output: Value of fixed length
  - Used to provide integrity

Hash Function: $h(x) = x \ mod \ m$

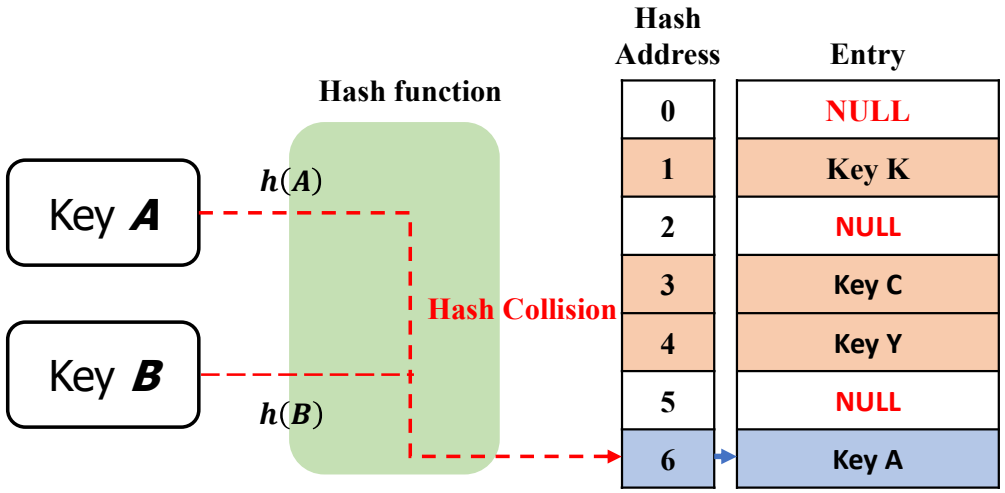
Input: x
Output: h(x)
mod = % operation

```
/hash# vim hash_function.c
```

```c
#include <stdio.h>

#define MSIZE 13

int hash_function(int key)
{
    return key % MSIZE;
}

int main(int argc, char *argv[])
{
    int key;
    int ret_index;

    printf("Please input your key: ");
    scanf("%d", &key);

    ret_index = hash_function(key);

    printf("This is index of your key: %d\n", ret_index);

    return 0;
}
```

Lock: Hash

- Hash Problem
  - There are two keys A and B
  - h(A) == h(B)
  - They have same location
  - Hash Collision



Lock: 실습

- Hash Collision

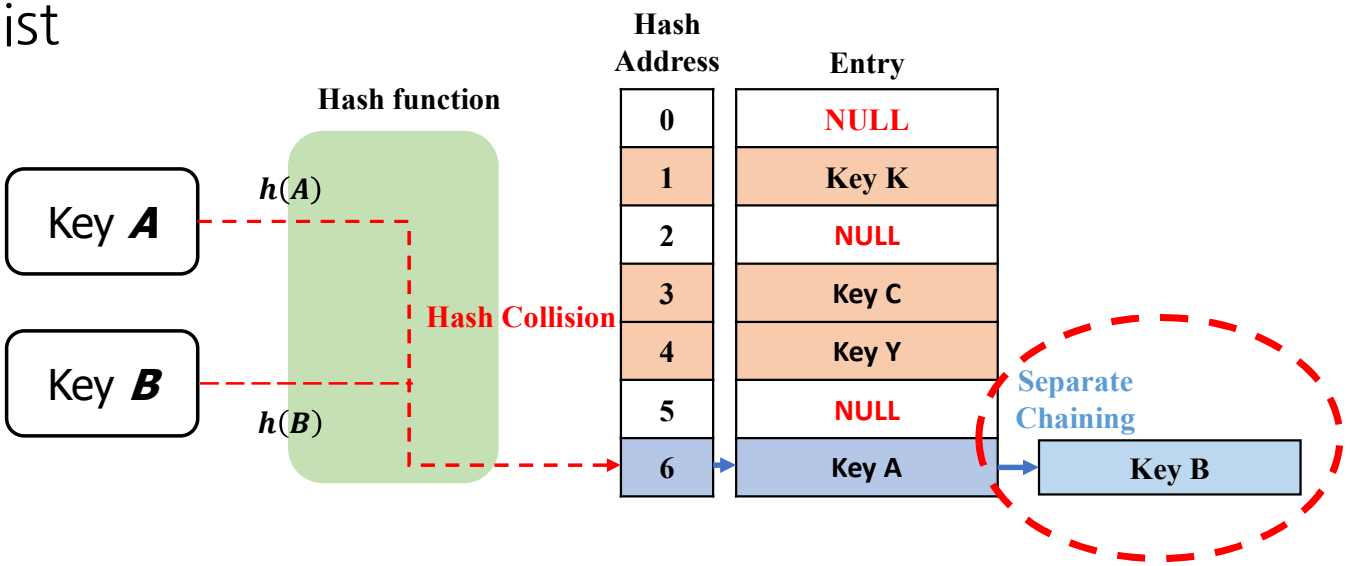
```
/hash# vim hash_collision.c
```

```c
#include <stdio.h>

#define MSIZE 13

int hash_function(int key)
{
    return key % MSIZE;
}
```

```c
int main(int argc, char *argv[])
{
    int key;
    int ret_index;
    int hash_array[100];

    for(int i = 0; i < 100; i++)
        hash_array[i] = 0;

    while(1)
    {
      printf("Please input your key: ");
      scanf("%d", &key);

      ret_index = hash_function(key);

      if(hash_array[ret_index] != 0)
      {
         printf("The data is already in %d hash index\n", ret_index);
         break;
      }
      hash_array[ret_index] = key;


      printf("This is index of your key: %d\n", ret_index);
    }

    return 0;
}
```

Lock: Hash

- Hash Solution (Chaining)
  - Solution ➔ Separate Chaining
  - Append Entry at collision location
  - Implement using linked list



Lock: 실습

- Hash Solution (Chaining)

```
/hash# vim hash_chaining.c
```

```c
#include <stdio.h>
#include <stdlib.h>

#define MSIZE 5

typedef struct _Node
{
    int index;
    struct _Node* nextIndex;
}Node;

Node* hashArray[MSIZE];
```

```c
int hash_function(int key)
{
    return key % MSIZE;
}

void AddHash(int key, Node* node)
{
    int ret_index = hash_function(key);
    if(hashArray[ret_index] == NULL)
    {
        hashArray[ret_index] = node;
    }
    else
    {
        node->nextIndex = hashArray[ret_index];
        hashArray[ret_index] = node;
    }
    printf("This is index of your key: %d\n", ret_index);
}
```

Lock: 실습

- Hash Solution (Chaining)

```
void ShowHash()
{
   printf("\n[Show Your Hash Data]\n");
   for(int i = 0; i < MSIZE; i++)
   {
        if(hashArray[i] != NULL)
        {
           printf("\n===Your hash index value = %d===\n", i);
           Node* node = hashArray[i];
           while(node->nextIndex)
           {
                printf("Hash value = %d\n", node->index);
                node = node->nextIndex;
           };
           printf("Hash value = %d\n", node->index);
        }
   }
}
```

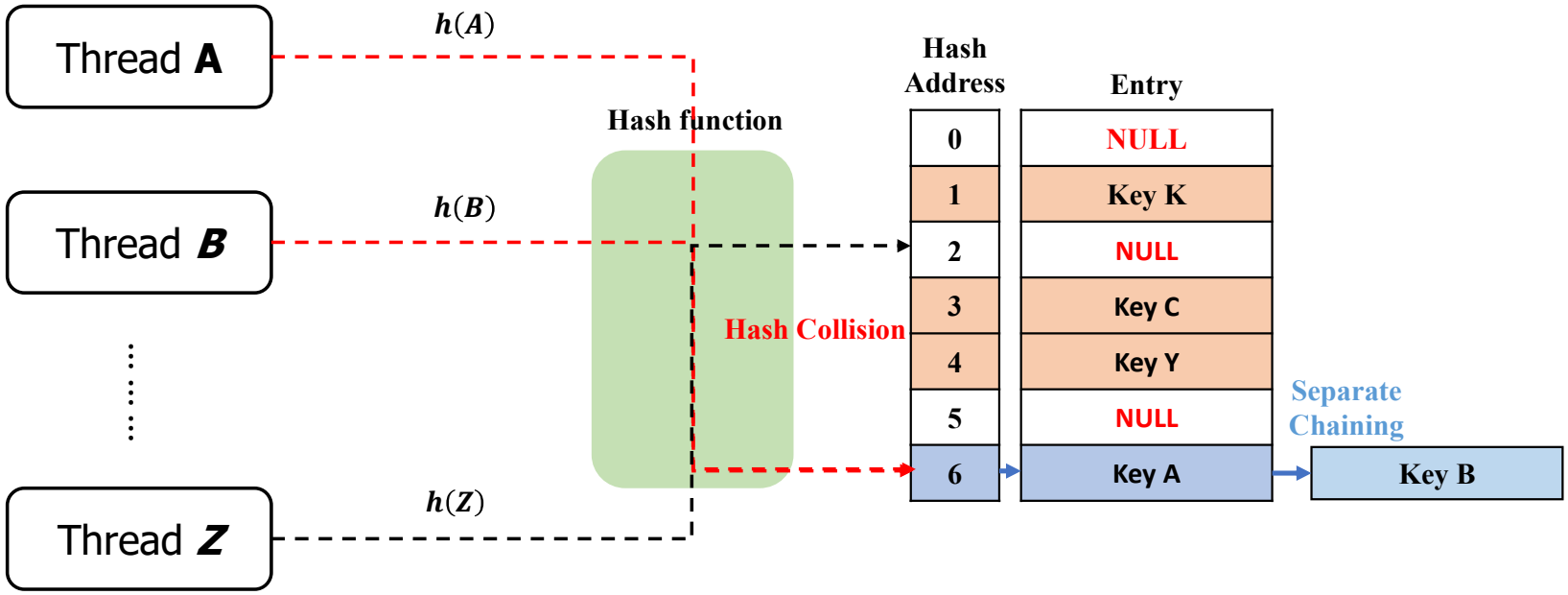
Lock: 실습

- Hash Solution (Chaining)

```
int main(int argc, char *argv[])
{
    int key;
    int count_run = 0;

    while(1)
    {
      printf("Please input your key: ");
      scanf("%d", &key);

      Node* node = (Node*)malloc(sizeof(Node));
      node->index = key;
      node->nextIndex = NULL;

      AddHash(node->index, node);
      count_run++;

      if(count_run == 20)
           break;
    }

    ShowHash();

    return 0;
}
```

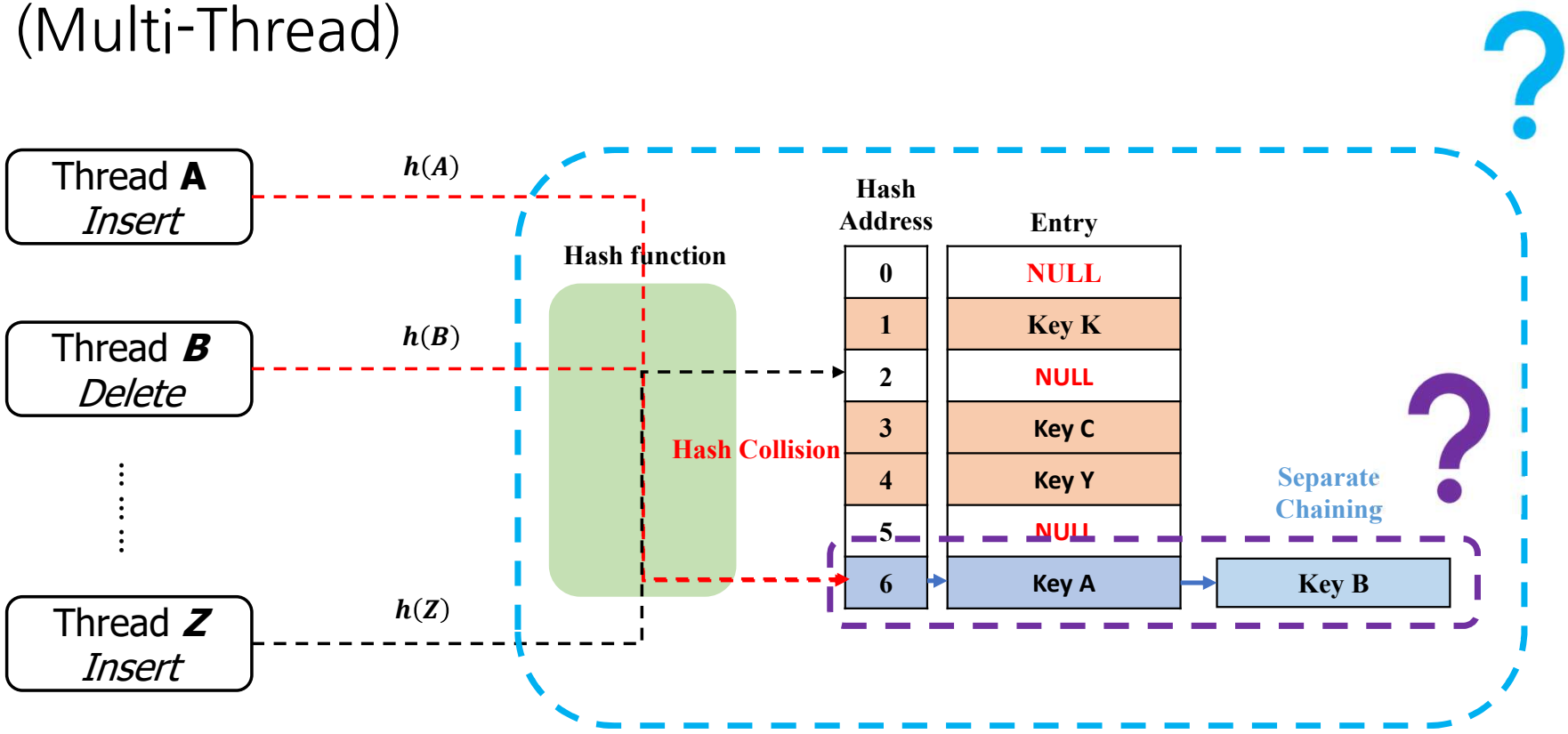
Lock: Hash

- Hash (Multi-Thread)



Lock: Hash

- Hash (Multi-Thread)



Lock: 실습

- Hash with Lock Problem

```
/hash# vim hash_lock.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <assert.h>

#define MSIZE 13

int nthread = 1;
int add_cnt = 1;

typedef struct _Node
{
    int index;
    struct _Node* nextIndex;
}Node;

Node* hashArray[MSIZE];

int hash_function(int key)
{
    return key % MSIZE;
}
```

```c
static void *AddHash(void *num)
{
    int num_cnt = 10;
    int th_num = (long)num;

    for(int i = 0; i < num_cnt; i++)
    {
        //printf("Your thread number : [%d]\n", th_num);
        srand(th_num);
        int key = rand() % 100;

        Node* node = (Node *)malloc(sizeof(Node));
        node->index = key;
        node->nextIndex = NULL;

        int ret_index = hash_function(key);

        if(hashArray[ret_index] == NULL)
        {
            hashArray[ret_index] = node;
        }
        else
        {
            node->nextIndex = hashArray[ret_index];
            hashArray[ret_index] = node;
        }
    }
}
```

Lock: 실습

- Hash with Lock Problem

```
void ShowHash()
{
   int cnt = 0;
   //printf("\n[Show Your Hash Data]\n");
   for(int i = 0; i < MSIZE; i++)
   {
        if(hashArray[i] != NULL)
        {
           //printf("\n===Your hash index value = %d===\n", i);
           Node* node = hashArray[i];
           while(node->nextIndex)
           {
                //printf("Hash value = %d\n", node->index);
                node = node->nextIndex;
                cnt++;
           };
           //printf("Hash value = %d\n", node->index);
           cnt++;
        }
   }
   printf("\n[Total number of data : %d]\n", cnt);
}
```

Lock: 실습

- Hash with Lock Problem

```c
int main(int argc, char *argv[])
{
   long i;
   pthread_t *th;

   if (argc < 2)
   {
      fprintf(stderr, "%s parameter : nthread\n", argv[0]);
      exit(-1);
   }

   nthread = atoi(argv[1]);

   th = malloc(sizeof(pthread_t) * nthread);

   for(i = 0; i < nthread; i++)
   {
      assert(pthread_create(&th[i], NULL, AddHash, (void *) i) == 0);
   }

   for(i = 0; i < nthread; i++)
   {
      assert(pthread_join(th[i], NULL) == 0);
   }

   ShowHash();

   return 0;
}
```

```
/hash# gcc hash_lock.c -lpthread -o hash_lock
```

Lock: 실습

- Hash with Lock Problem

```
/hash# cat hash_lock.c > hash_lock_2.c
```

```
/hash# vim hash_lock_2.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <assert.h>

#define MSIZE 13

int nthread = 1;
int add_cnt = 1;

pthread_mutex_t lock;
```

```c
static void *AddHash(void *num)
{
    int num_cnt = 10;
    int th_num = (long)num;

    for(int i = 0; i < num_cnt; i++)
    {
        //printf("Your thread number : [%d]\n", th_num);

        pthread_mutex_lock(&lock); // lock

        srand(th_num);
        int key = rand() % 100;

        Node* node = (Node *)malloc(sizeof(Node));
        node->index = key;
        node->nextIndex = NULL;

        int ret_index = hash_function(key);

        if(hashArray[ret_index] == NULL)
        {
            hashArray[ret_index] = node;
        }
        else
        {
            node->nextIndex = hashArray[ret_index];
            hashArray[ret_index] = node;
        }

        pthread_mutex_unlock(&lock); // unlock
    }
}
```

Lock: 실습

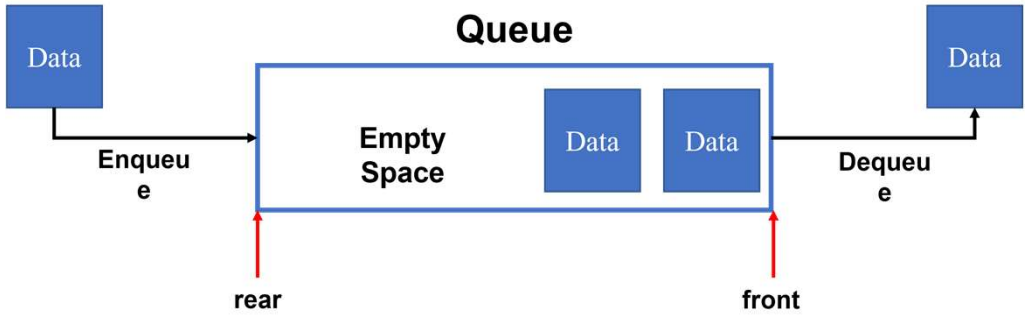
- Hash with Lock Problem

```c
int main(int argc, char *argv[])
{
   long i;
   pthread_t *th;

   if (argc < 2)
   {
      fprintf(stderr, "%s parameter : nthread\n", argv[0]);
      exit(-1);
   }

   nthread = atoi(argv[1]);

   th = malloc(sizeof(pthread_t) * nthread);

   pthread_mutex_init(&lock, NULL); //Initialize the lock

   for(i = 0; i < nthread; i++)
   {
      assert(pthread_create(&th[i], NULL, AddHash, (void *) i) == 0);
   }

   for(i = 0; i < nthread; i++)
   {
      assert(pthread_join(th[i], NULL) == 0);
   }

   ShowHash();

   return 0;
}
```

```
/hash# gcc hash_lock_2.c -lpthread -o hash_lock_2
```

Lock: Queue

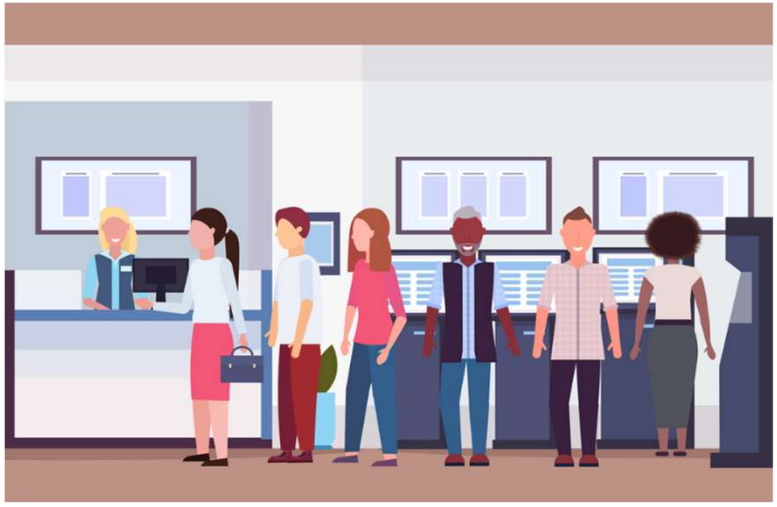
- Queue Data Structure
  - FIFO (First In First Out)
  - Add(Enqueue) at rear
  - Delete(Dequeue) at front
  - O(n) for Search
  - O(1) for Add and Delete



Lock: Queue

- Queue Data Structure
  - If queue is full, cannot use
  - Other types of queue:

    Circle Queue / Priority Queue



Lock: 실습

- Queue

```
/queue# vim queue.c
```

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 30

typedef struct Node
{
    int data;
    struct Node *next;
}Node;

typedef struct Queue
{
    Node *front;
    Node *rear;
    int count;
}Queue;

void initQueue(Queue *queue)
{
    queue->front = queue->rear = NULL;
    queue->count = 0;
}

int isEmpty(Queue *queue)
{
    return queue->count == 0;
}
```

```c
void enqueue(Queue *queue, int data)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;

    if (isEmpty(queue))
    {
        queue->front = newNode;
    }
    else
    {
        queue->rear->next = newNode;
    }
    queue->rear = newNode;
    queue->count++;

    printf("Enqueue data = [%d]\n", data);
}
```

Lock: 실습

- Queue

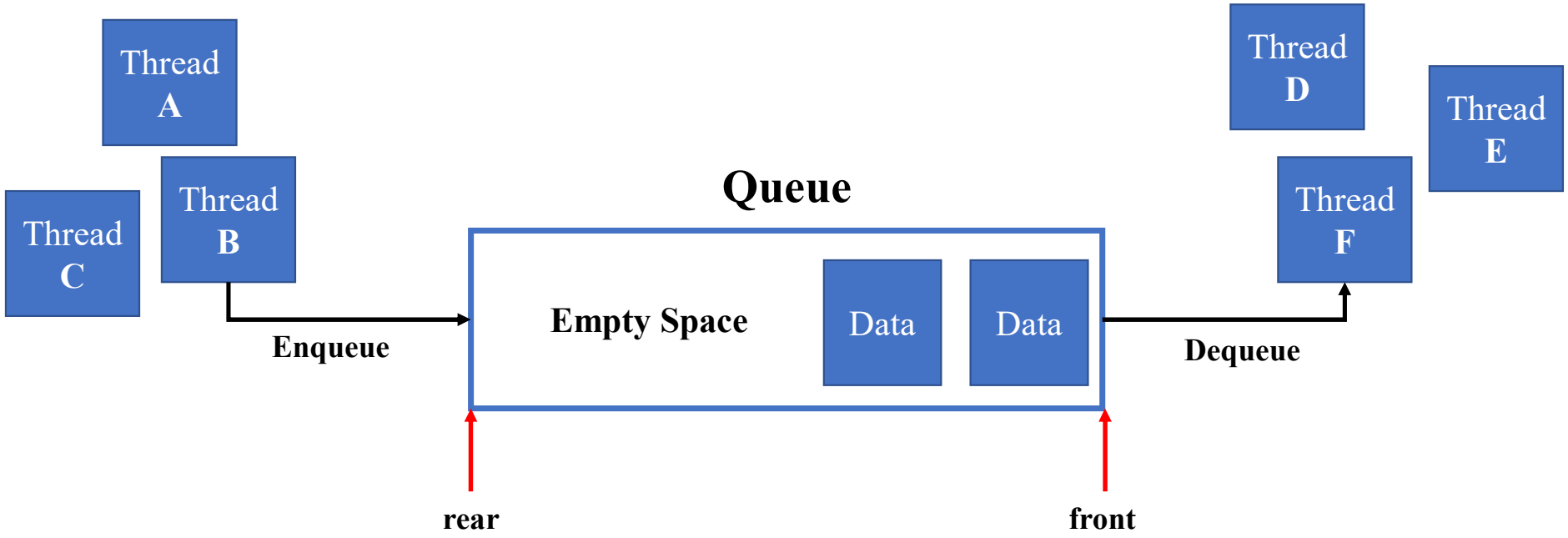
```
int dequeue(Queue *queue)
{
    int data;
    Node *loc;
    if (isEmpty(queue))
    {
        fprintf(stderr, "Queue is Empty. You cannot delete any data.\n");
        return 0;
    }
    loc = queue->front;
    data = loc->data;
    queue->front = loc->next;
    queue->count--;

    free(loc);
    return data;
}
```

```
int main(int argc, char *argv[])
{
    int i;
    Queue queue;

    initQueue(&queue);

    for(i = 0; i <= MAX_SIZE; i++)
    {
        enqueue(&queue, i);
    }
    while(!isEmpty(&queue))
    {
        printf("Dnqeueu data = [%d]\n", dequeue(&queue));
    }
    return 0;
}
```

```
/queue# gcc queue.c -o queue
```

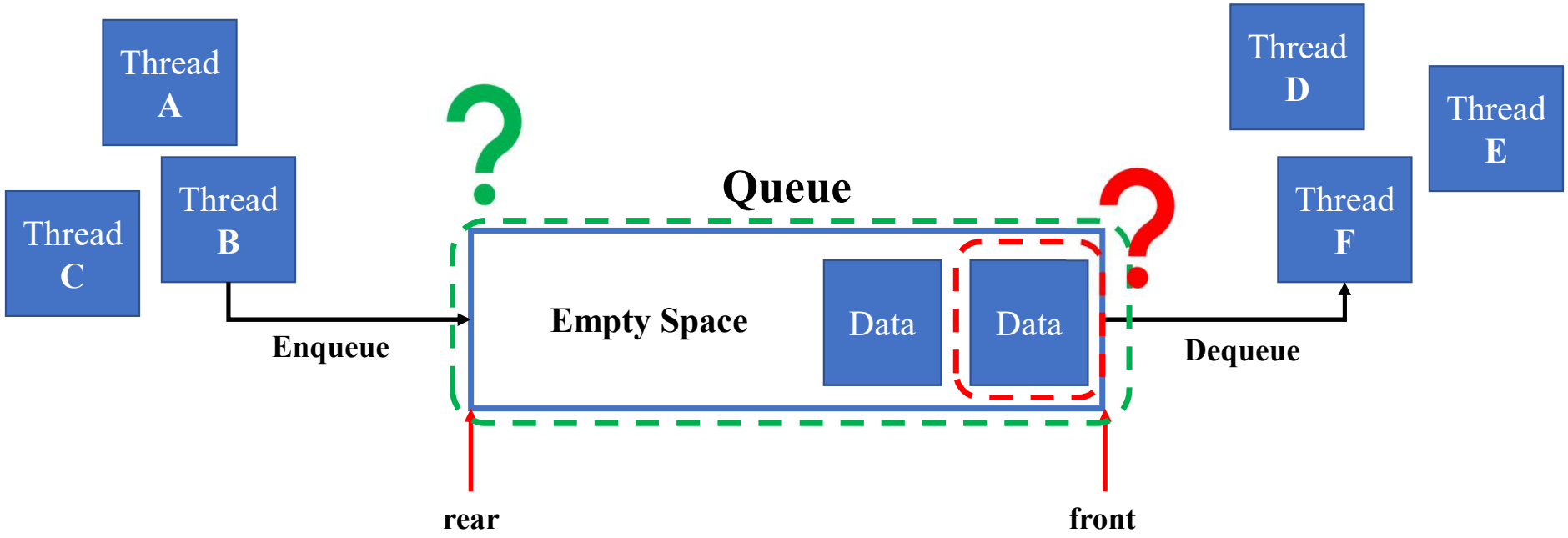
Lock: Queue

- Queue



Lock: Queue

- Queue



Lock: 실습

- Queue

```
/queue# vim queue_lock.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <pthread.h>

int nthread = 1;

#define MAX_SIZE 30

typedef struct Node
{
    int data;
    struct Node *next;
}Node;

typedef struct Queue
{
    Node *front;
    Node *rear;
    int count;
}Queue;

Queue* queue;
```

```c
void initQueue(Queue *_queue)
{
    _queue->front = _queue->rear = NULL;
    _queue->count = 0;
}

int isEmpty(Queue *queue)
{
    return queue->count == 0;
}
```

Lock: 실습

- Queue

```
void* enqueue(void *data)
{
    int _data = (long)data;
    for(int i = 0; i < MAX_SIZE; i++)
    {
        Node *newNode = (Node *)malloc(sizeof(Node));
        newNode->data = _data;
        _data = _data+1;
        newNode->next = NULL;

        if (isEmpty(queue))
        {
            queue->front = newNode;
        }
        else
        {
            queue->rear->next = newNode;
        }
        queue->rear = newNode;
        queue->count++;

    //printf("Enqueue data = [%d]\n", _data);
    }
}
```

```
void* dequeue(void *data)
{
    int _data;
    for(int i = 0; i < MAX_SIZE; i++)
    {
        Node *loc;
        if (isEmpty(queue))
        {
            fprintf(stderr, "Queue is Empty. You cannot delete any data.\n");
            return 0;
        }
        loc = queue->front;
        _data = loc->data;
        queue->front = loc->next;
        queue->count--;

        free(loc);
    }
}
```

Lock: 실습

- Queue

```c
int main(int argc, char *argv[])
{
    pthread_t *th;
    long i;

    if (argc < 2)
    {
        fprintf(stderr, "%s parameter : nthread\n", argv[0]);
    }

    nthread = atoi(argv[1]);

    th = malloc(sizeof(pthread_t) * nthread);
    queue = malloc(sizeof(Queue));

    initQueue(queue);

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_create(&th[i], NULL, enqueue, (void *) i) == 0);
    }

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_create(&th[i], NULL, dequeue, (void *) i) == 0);
    }

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_join(th[i], NULL) ==0);
    }

    return 0;
}
```

```
/queue# gcc queue_lock.c -lpthread -o queue_lock
```

Lock: 실습

- Queue with Lock Problem

```
/queue# gcc queue_lock_2.c -lpthread -o queue_lock_2
```

```
/queue# vim queue_lock_2.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <pthread.h>

int nthread = 1;

#define MAX_SIZE 30

pthread_mutex_t en_lock;
pthread_mutex_t de_lock;

typedef struct Node
{
    int data;
    struct Node *next;
}Node;

typedef struct Queue
{
    Node *front;
    Node *rear;
    int count;
}Queue;

Queue* queue;
```

```c
void initQueue(Queue *_queue)
{
    _queue->front = _queue->rear = NULL;
    _queue->count = 0;
}

int isEmpty(Queue *queue)
{
    return queue->count == 0;
}
```

Lock: 실습

- Queue with Lock Problem

```
void* enqueue(void *data)
{
    int _data = (long)data;
    for(int i = 0; i < MAX_SIZE; i++)
    {
        pthread_mutex_lock(&en_lock); // lock

        Node *newNode = (Node *)malloc(sizeof(Node));
        newNode->data = _data;
        _data = _data+1;
        newNode->next = NULL;

        if (isEmpty(queue))
        {
            queue->front = newNode;
        }
        else
        {
            queue->rear->next = newNode;
        }
        queue->rear = newNode;
        queue->count++;

      //printf("Enqueue data = [%d]\n", _data);

        pthread_mutex_unlock(&en_lock); // unlock
    }
}
```

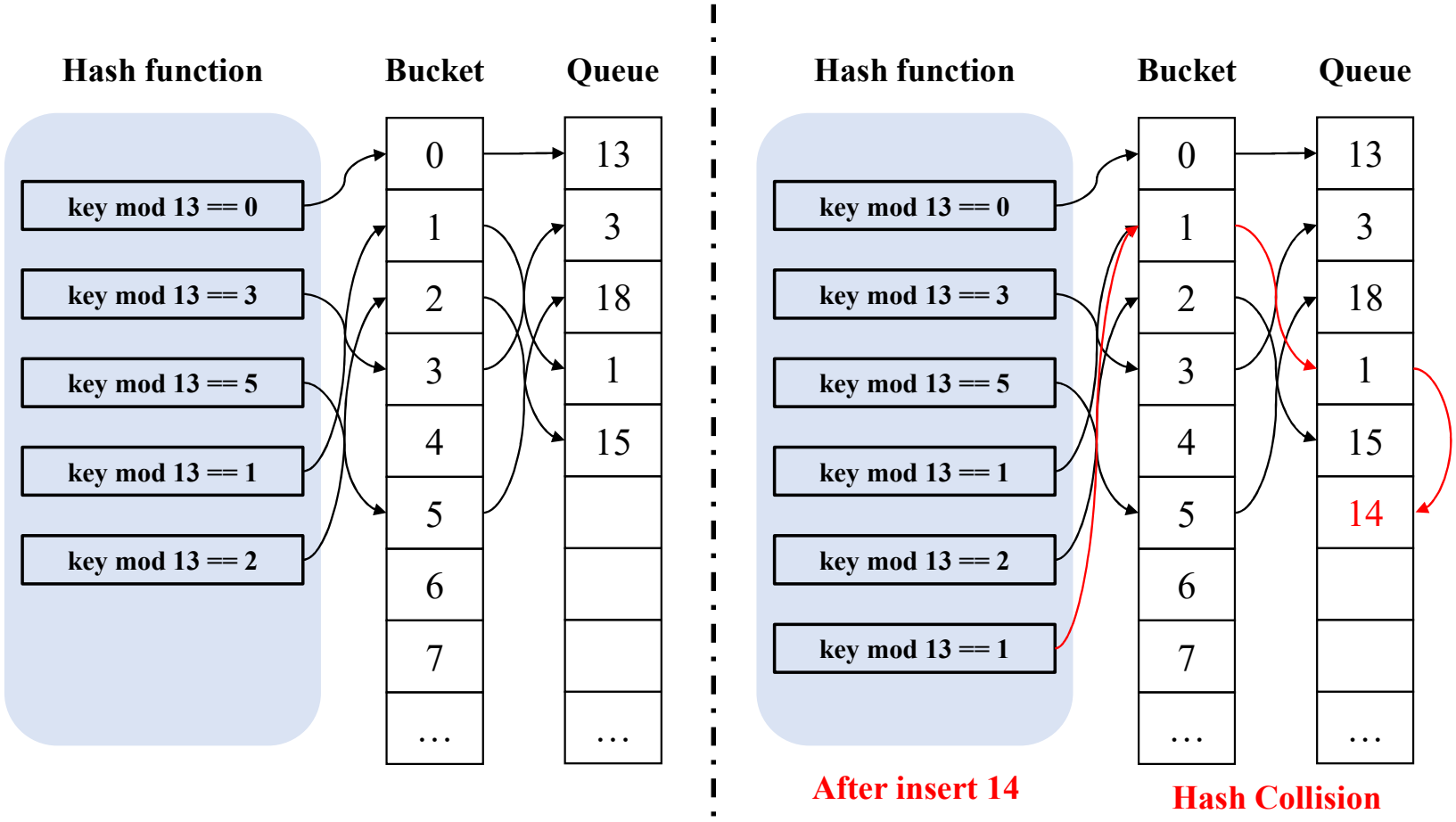
```
void* dequeue(void *data)
{
    int _data;
    for(int i = 0; i < MAX_SIZE; i++)
    {
        pthread_mutex_lock(&de_lock); // lock

        Node *loc;
        if (isEmpty(queue))
        {
            fprintf(stderr, "Queue is Empty. You cannot delete any data.\n");
            return 0;
        }
        loc = queue->front;
        _data = loc->data;
        queue->front = loc->next;
        queue->count--;

        free(loc);

        pthread_mutex_unlock(&de_lock); // unlock
    }
}
```

Lock: 실습

- Queue with Lock Problem

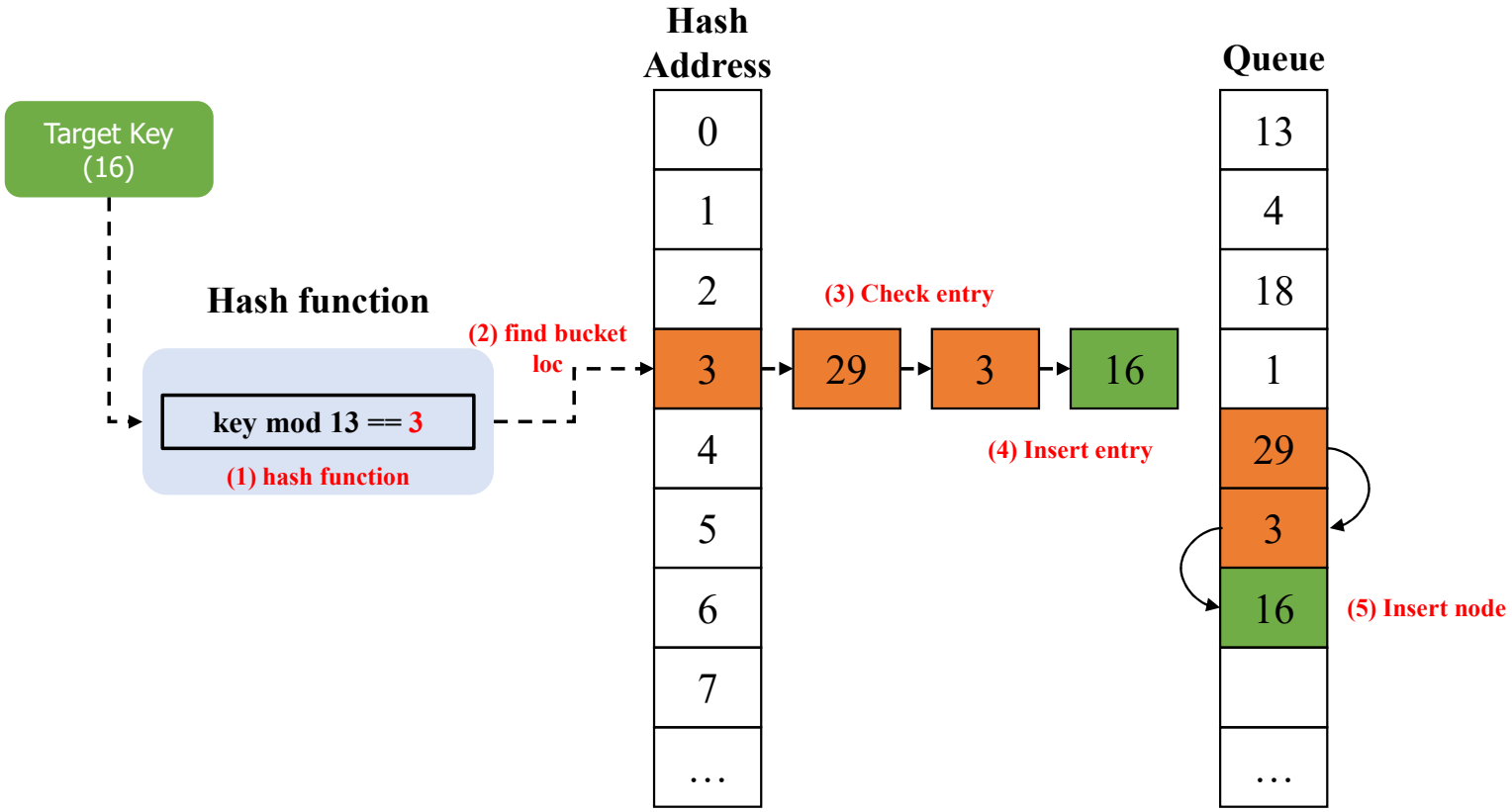
```
int main(int argc, char *argv[])
{
    pthread_t *th_en;
    pthread_t *th_de;
    long i;

    if (argc < 2)
    {
        fprintf(stderr, "%s parameter : nthread\n", argv[0]);
    }

    nthread = atoi(argv[1]);

    th_en = malloc(sizeof(pthread_t) * nthread);
    th_de = malloc(sizeof(pthread_t) * nthread);
    queue = malloc(sizeof(Queue));

    pthread_mutex_init(&en_lock, NULL);
    pthread_mutex_init(&de_lock, NULL);

    initQueue(queue);

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_create(&th_en[i], NULL, enqueue, (void *) i) == 0);
    }

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_create(&th_de[i], NULL, dequeue, (void *) i) == 0);
    }

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_join(th_en[i], NULL) == 0);
    }

    for(i = 0; i < nthread; i++)
    {
        assert(pthread_join(th_de[i], NULL) == 0);
    }

    return 0;
}
```

Lock: Hash-Queue Problem



Lock: Hash-Queue Problem

- Hash Queue Lock Problem (Insert)



Lock: Hash-Queue Problem

- Hash Queue Lock Problem (Delete)