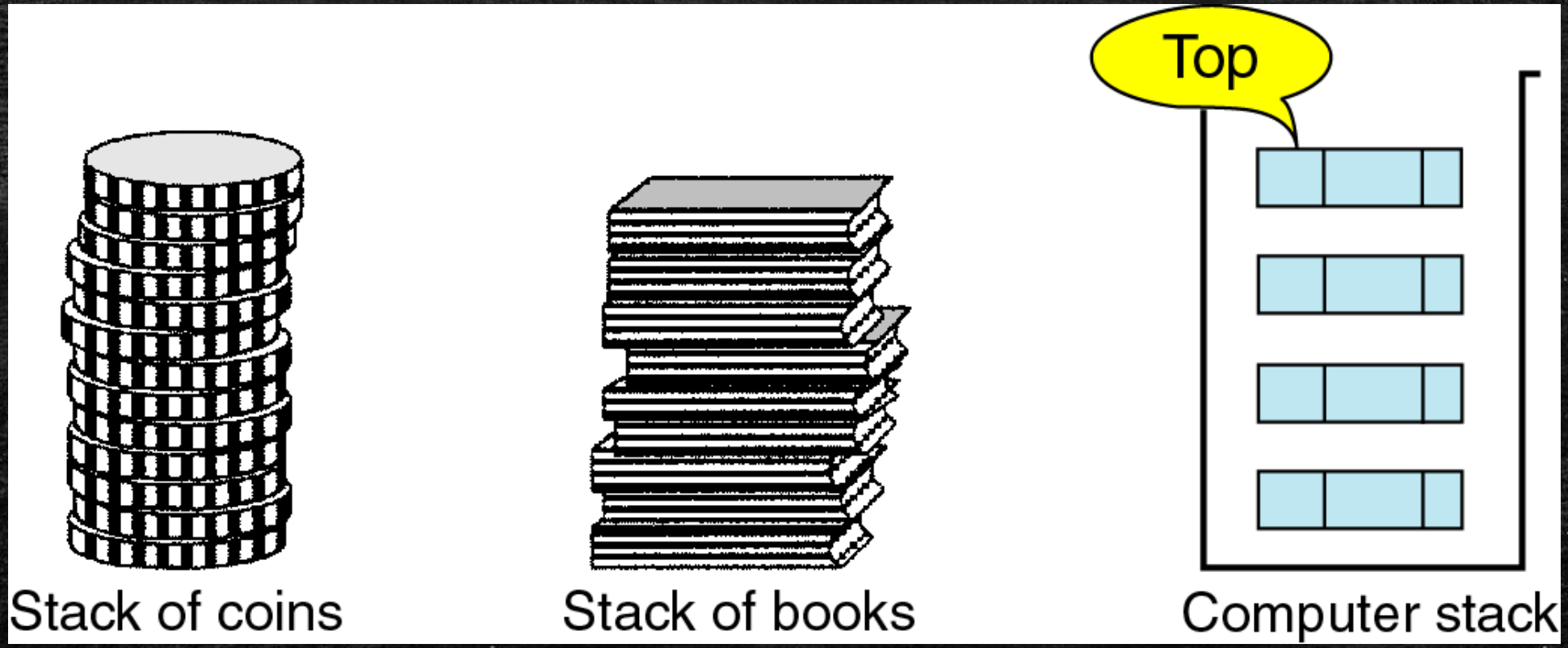
Design and Analysis of Data Structures and Algorithms

Warin Wattanapornprom Ph.D.

Stack

- Stack เป็นโครงสร้างข้อมูลแบบ LIFO (Last-In, First-Out)
- Operations พื้นฐานของ Stack ได้แก่
  - การนำข้อมูลเข้าสู่ Stack เรียกว่า Push
  - การนำข้อมูลออกจาก Stack เรียกว่า Pop
  - การเรียกใช้ข้อมูลจาก Stack เรียกว่า Top
- การสร้าง Stack
  - ใช้ Array แทน Stack
  - ใช้ Linked list แทน Stack

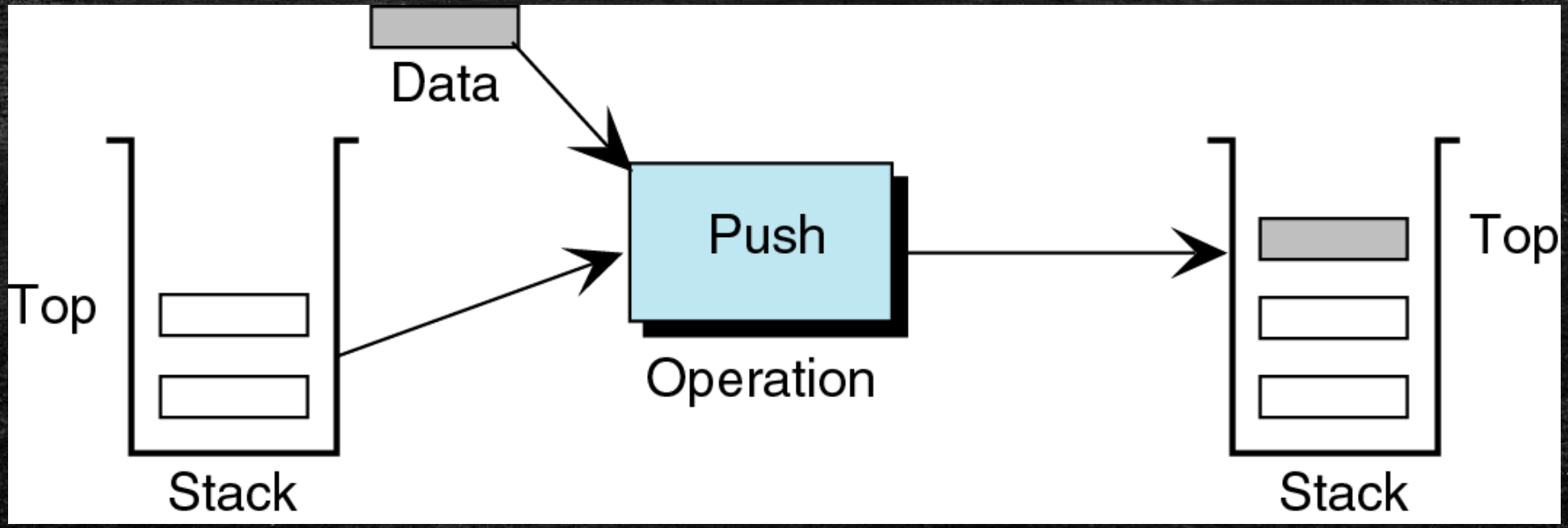
Stack



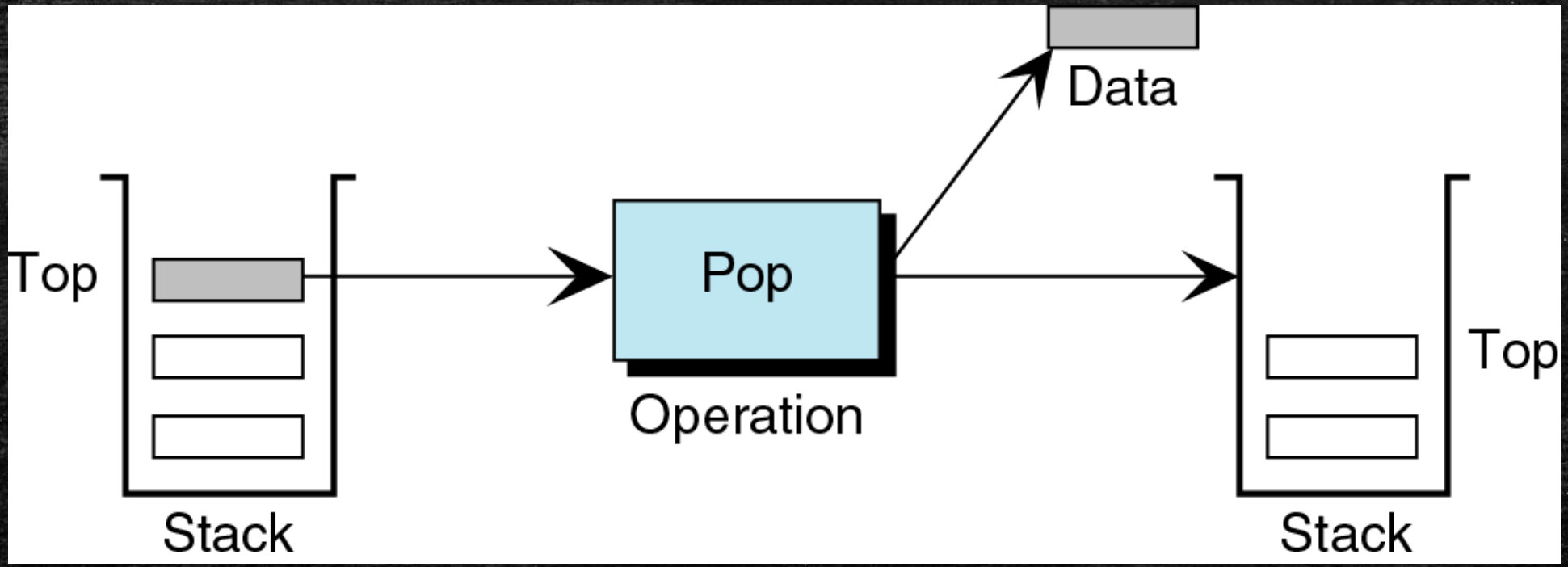
Operations พื้นฐานของ Stack

1. Create stack: สร้าง stack head node หรือ array
2. Push stack: เพิ่มรายการใน stack
3. Pop stack: ลบรายการใน stack
4. Stack top: เรียกดูรายการที่อยู่บนสุดของ stack
5. Empty stack: ตรวจสอบว่า stack ว่างเปล่าหรือไม่
6. Full stack: ตรวจสอบว่า stack เต็มหรือไม่
7. Stack count: ส่งค่าจำนวนรายการใน stack
8. Destroy stack: คืนหน่วยความจำของทุก node ใน stack ให้ระบบ

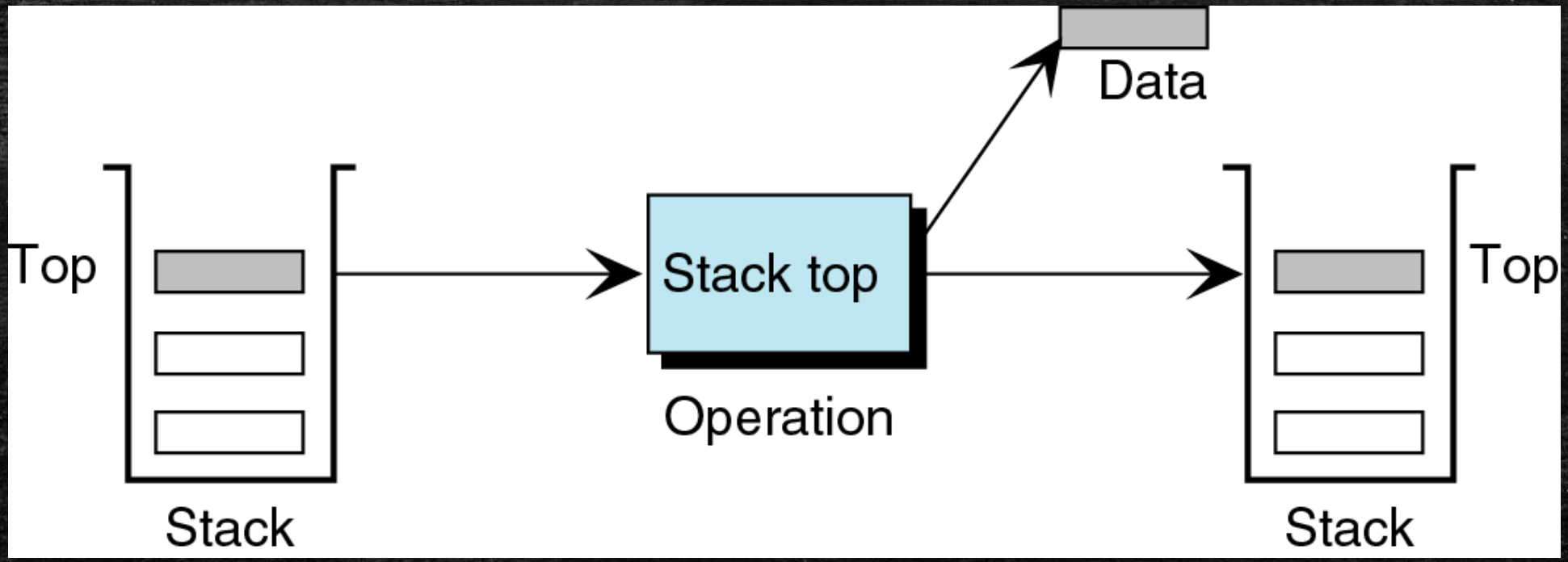
เพิ่มข้อมูลใน Stack: Push

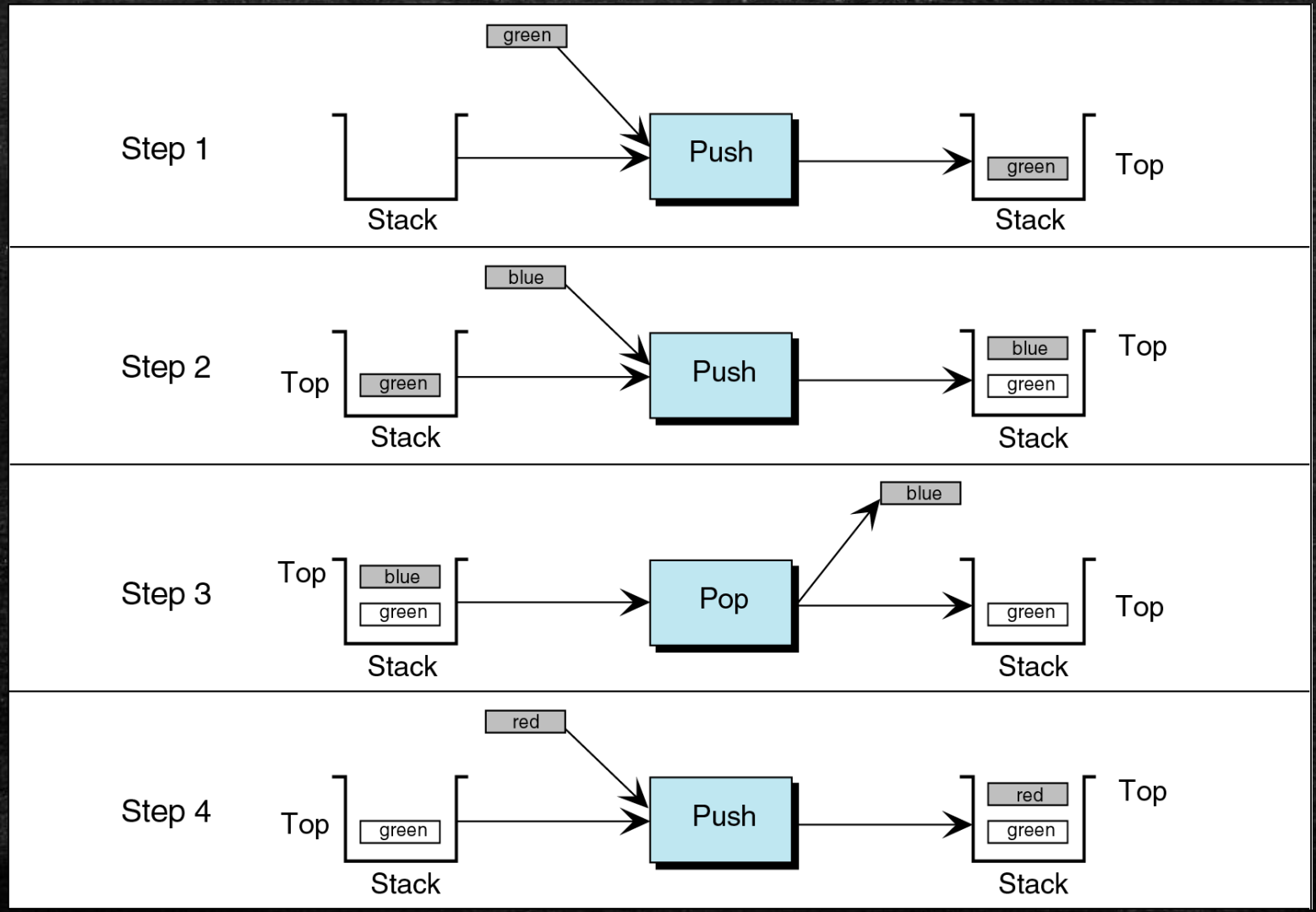


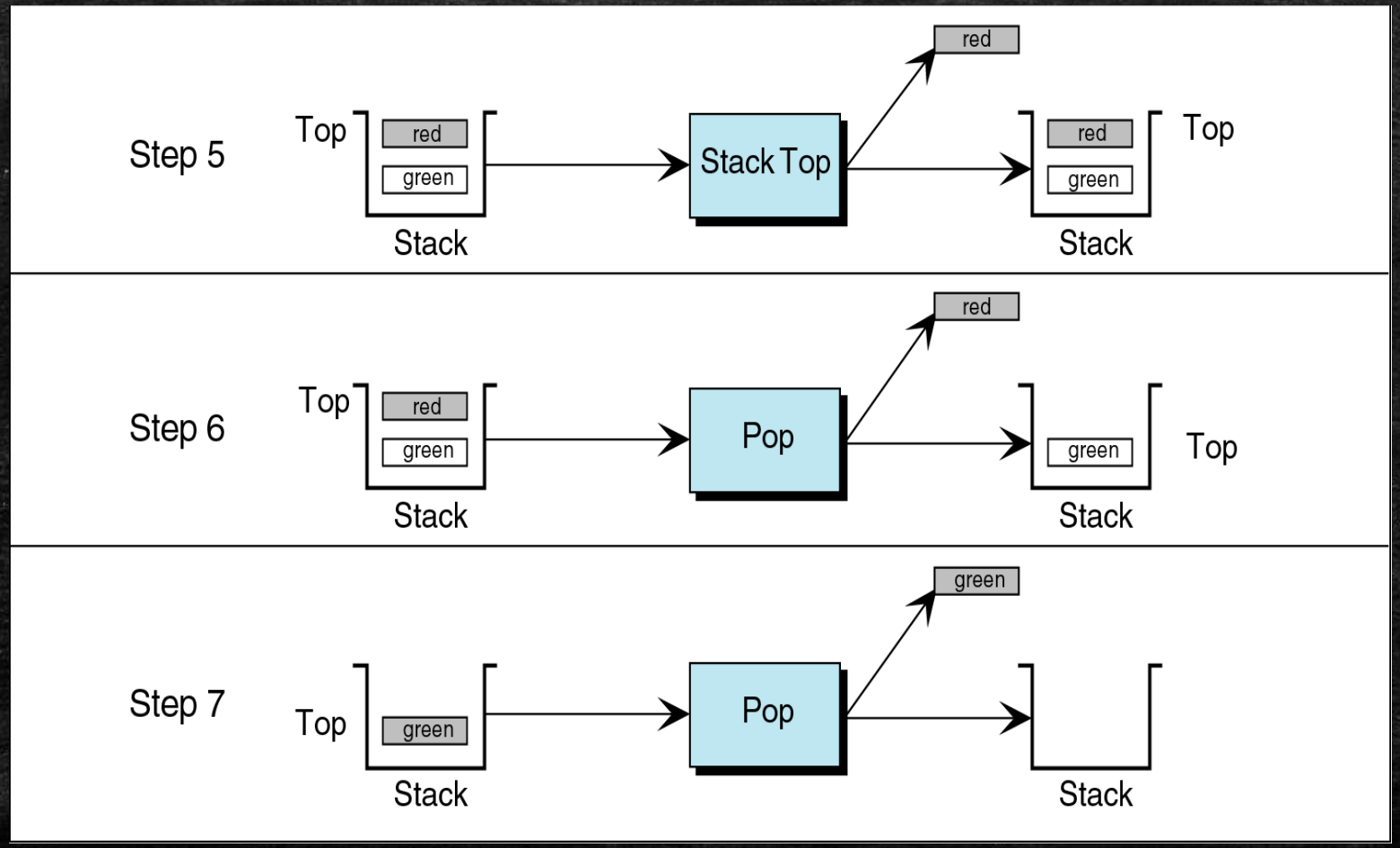
นำข้อมูลออกจาก Stack : Pop



เรียกใช้ข้อมูลใน Stack: Top




Step 1
green
Push
Stack
green
Top
Stack
Step 2
blue
Top
green
Push
Stack
blue
green
Top
Stack
Step 3
blue
Top
blue
green
Pop
Stack
green
Top
Stack
Step 4
red
Top
green
Push
Stack
red
green
Top
Stack


Step 5
Top
red
green
Stack
Stack Top
red
red
green
Stack
Top
Step 6
Top
red
green
Stack
Pop
red
green
Stack
Top
Step 7
Top
green
Stack
Pop
green
Stack

Array-Based Stack

```
class AStack{
private:
  int maxSize;                 // Maximum size of stack
  int top;                     // Index for top element
  int *listArray;             // Array holding stack elements

public:
  AStack(int size)   // Constructor
    { maxSize = size; top = 0; listArray = new int[size]; }

  ~AStack() { delete [] listArray; }  // Destructor

  void clear() { top = 0; }           // Reinitialize
```

Array-Based - Push

```
void push(int it) {          // Put "it" on stack
    if(top == maxSize)
        cout << "Stack is full" << endl;
    else
    listArray[top++] = it;
 }
```



```
//listArray[top+1]=it;
//top=top+1;
```

Array-Based - Pop

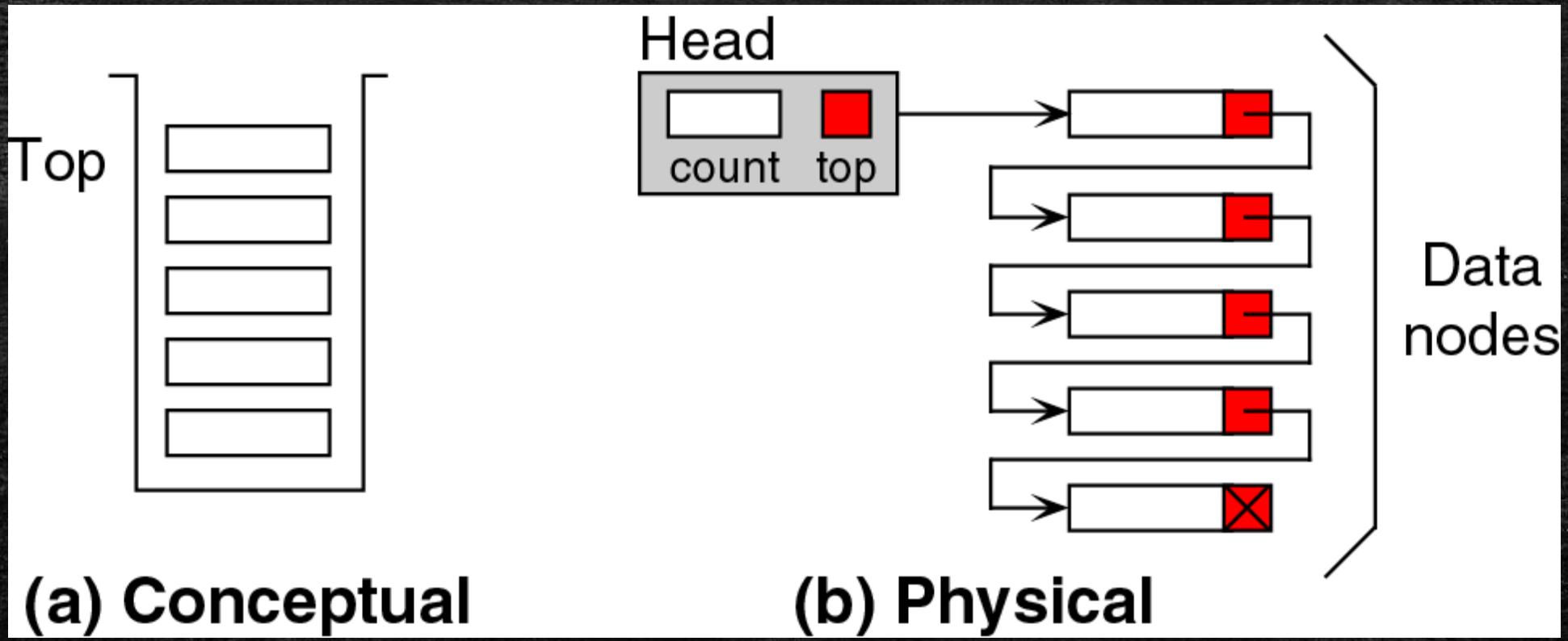
```
int pop() {                    // Pop top element
  if(top == 0)
      cout << "Stack is empty" << endl;
  else
  return listArray[--top];
}
```

```
//top=top-1;
//Return listArray[top+1];
```

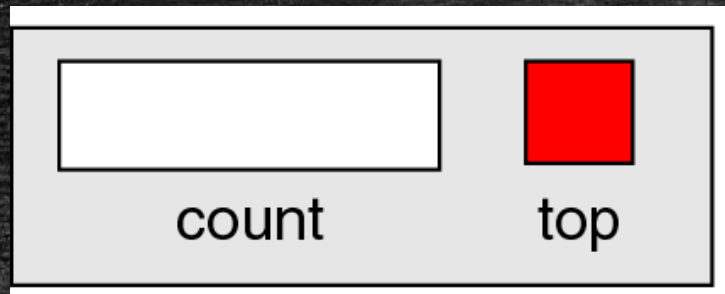
Array-Based - etc

```
 int topValue(){      // Return top element
    if(top == 0)
        cout << "Stack is empty" << endl;
    else
    return listArray[top-1];
    return 0;
  }

  int length() { return top; }  // Return length
};
```

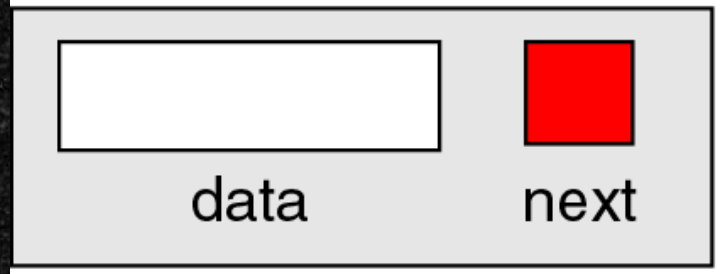
Linked list แทน Stack



Linked list แทน Stack

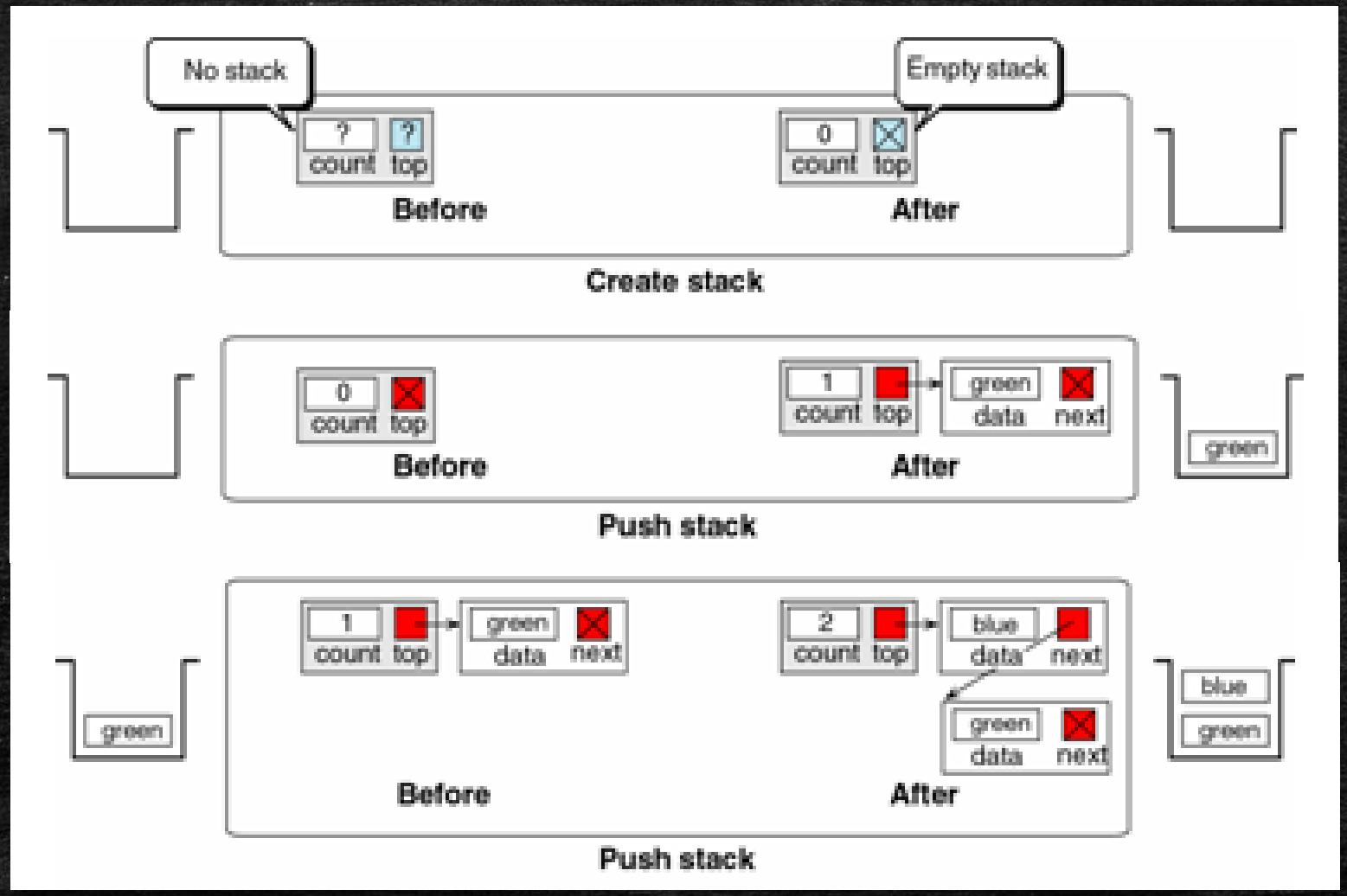


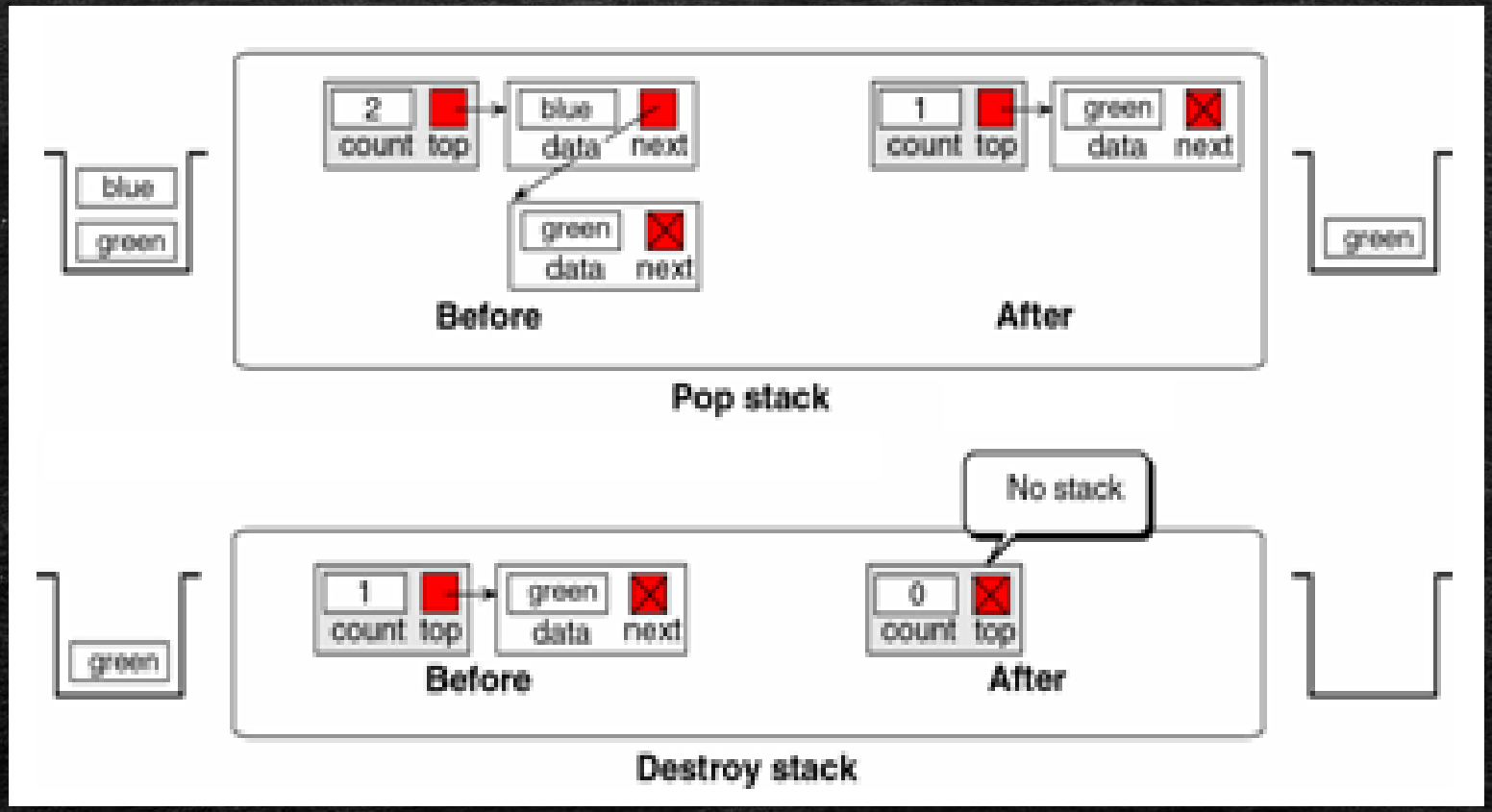
Stack head structure



Stack node structure

```
stack
  count  <integer>
  top    <node pointer>
end stack



node
  data   <dataType>
  next   <node pointer>
end node
```


No stack
?
count
top
Before
Empty stack
0
count
top
After
Create stack
0
count
top
Before
1
count
top
green
data
next
After
green
Push stack
1
count
top
green
data
next
Before
2
count
top
blue
data
next
green
data
next
After
green
blue
green
Push stack


blue
green
2
count
top
blue
data
next
green
data
next
Before
1
count
top
green
data
next
After
green
Pop stack
No stack
green
1
count
top
green
data
next
Before
0
count
top
After
Destroy stack

Linked Stack

What is the cost of the operations?

How do space requirements compare to the array-based stack implementation?

Linked Stack

```
class Link {
public:
  int element;       // Value for this node
  Link *next;         // Pointer to next node in list
  // Constructors
  Link(int elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }
};

class LStack{
private:
  Link* top;              // Pointer to first element
  int size;                   // Number of elements
```

Linked Stack - Clear

```
public:
  LStack() // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }              // Destructor

  void clear() {                      // Reinitialize
    while (top != NULL) {             // Delete link nodes
      Link* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }// Number of elements
```

Linked Stack - Push

```
void push(int it) { // Put "it" on stack
    top = new Link(it, top);
    size++;
  }
```

Linked Stack - Pop

```
int pop() {                    // Remove "it" from stack
    if(top == NULL)
     {          cout << "Stack is empty" << endl;
        return 0;
    }else{
    int it = top->element;
    Link* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
}
}
```

Linked Stack - etc

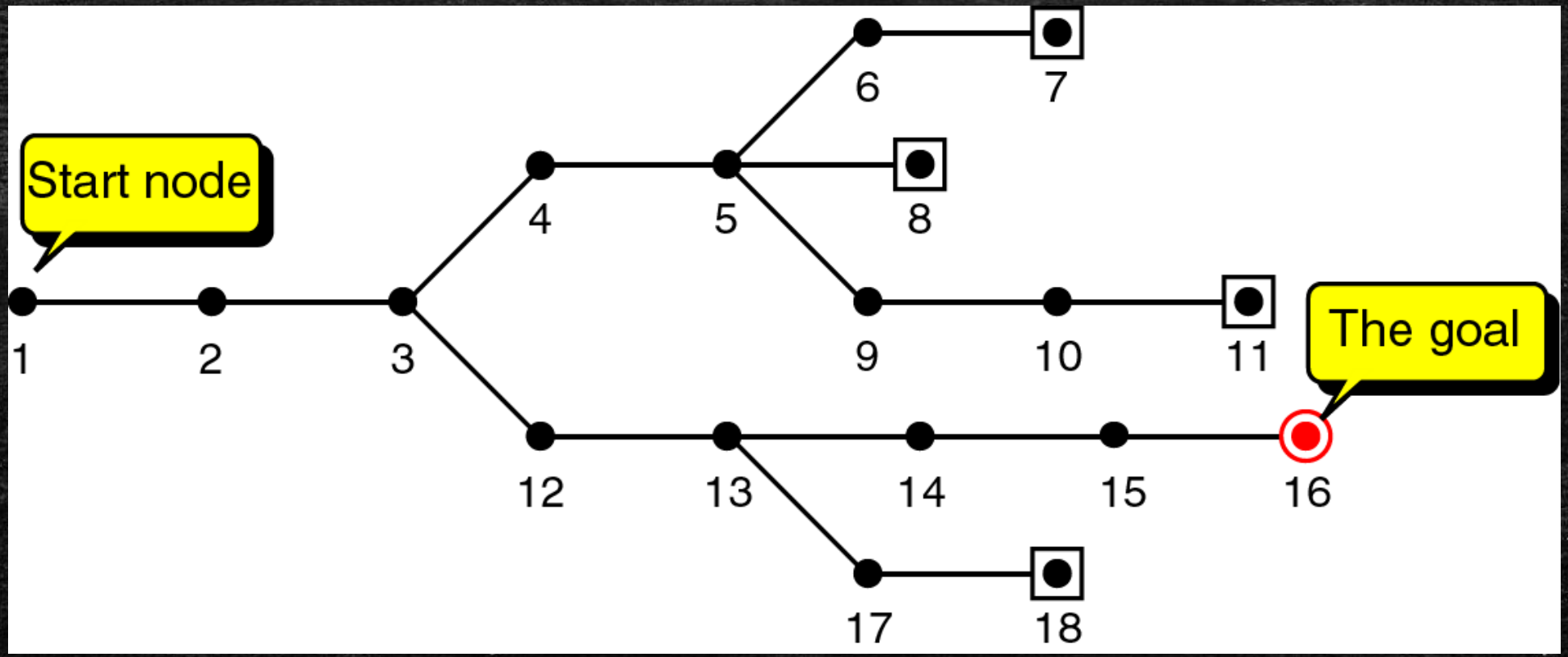
```
 int topValue() const { // Return top value
    if(top == 0){
        cout << "Stack is empty" << endl;
        return 0;
        } else
    return top->element;
}

  int length() const { return size; } // Return length
};
```

Backtracking

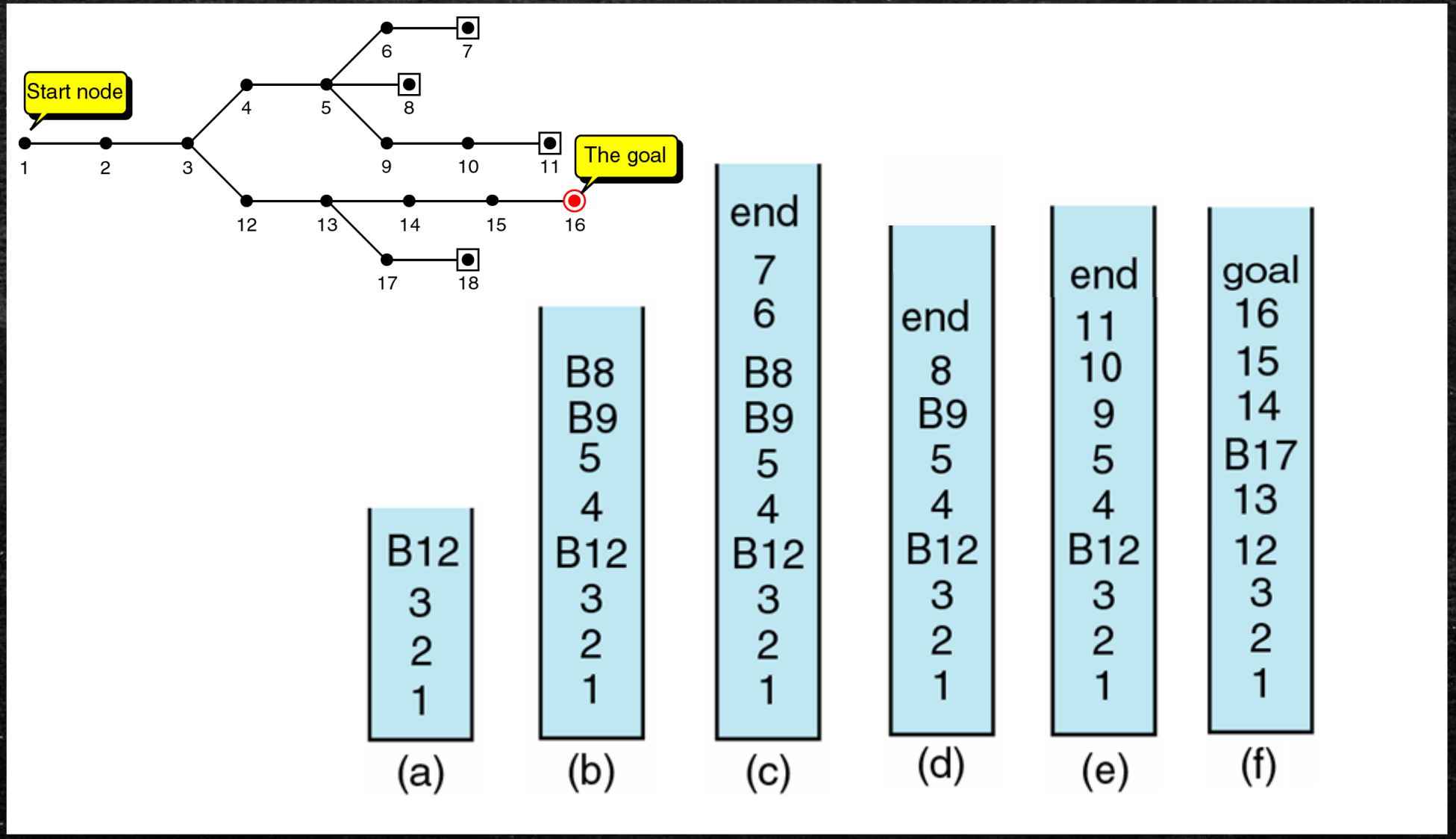
- Backtracking คือวิธีการหาคำตอบโดยเดินหน้าไปยังเป้าหมาย เมื่อถึงทางแยกก็จะต้องตัดสินใจเลือกเส้นทางใดเส้นทางหนึ่งเดินหน้าต่อไปเพื่อหาเป้าหมาย หากเดินไปจนสุดเส้นทางแล้วยังไม่พบเป้า ก็จะเดินย้อนกลับมายังจุดแยกครั้งสุดท้ายแล้วเลือกเส้นทางใหม่ที่ยังไม่เคยไป ทำเช่นนี้ไปเรื่อยๆจนกว่าจะพบเป้าหมาย หรือจนครบทุกเส้นทาง

- Backtracking เป็นการประยุกต์ใช้โครงสร้างข้อมูลแบบ Stack สำหรับการเขียนโปรแกรมประเภทเกมส์คอมพิวเตอร์ (computer gaming) การวิเคราะห์การตัดสินใจ(decision analysis) และระบบผู้เชี่ยวชาญ(expert system) ตัวอย่างปัญหาที่ใช้วิธี Backtracking เช่นปัญหาการค้นหาเป้าหมาย (goal seeking) และ ปัญหา 8 ราชินี (eight queens problem)

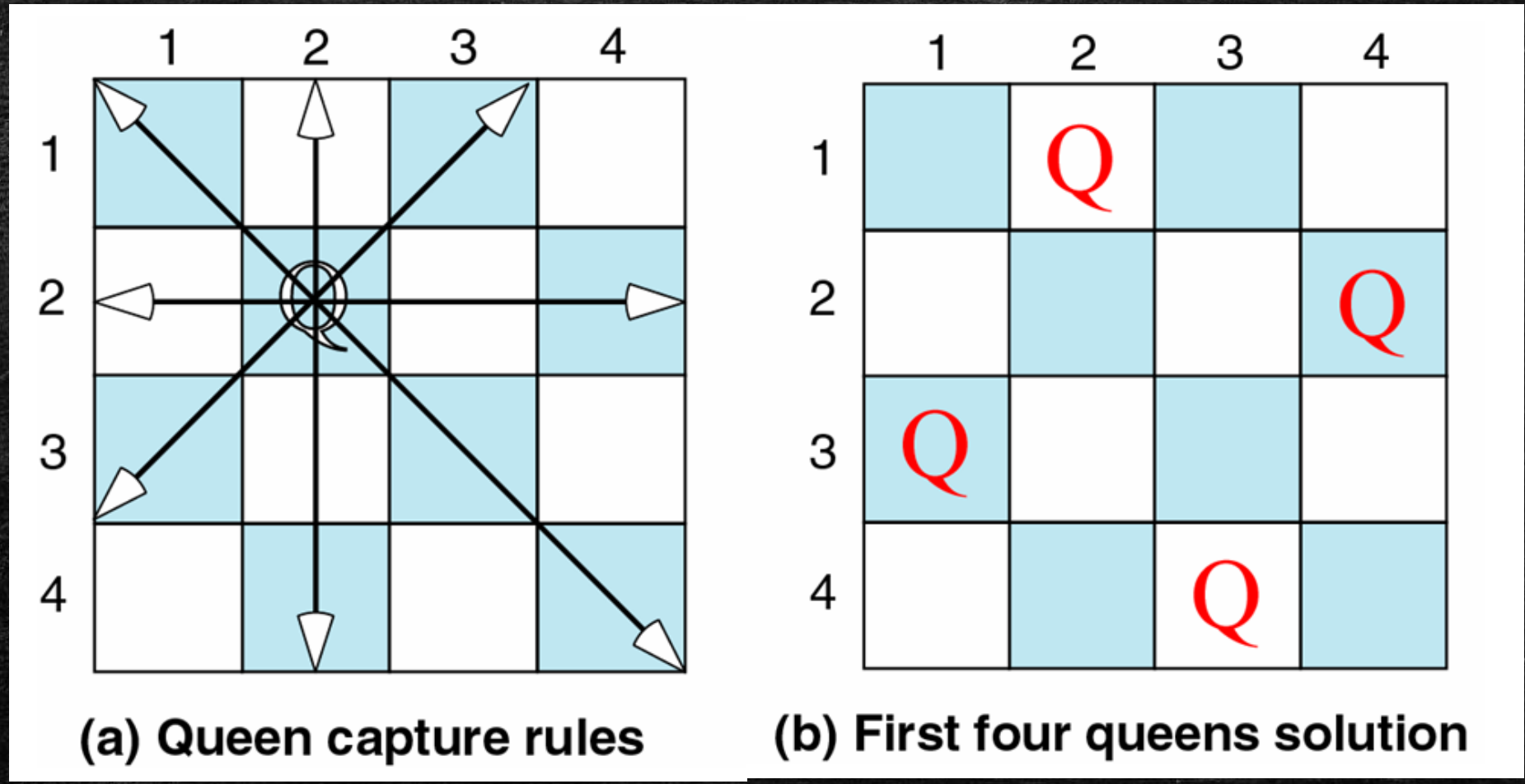
Stack Applications: Backtracking



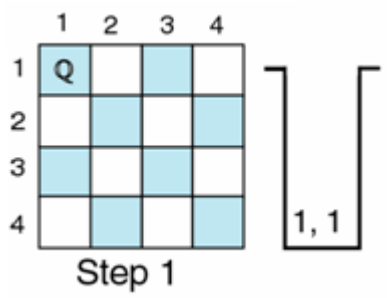
Stack Applications: Backtracking



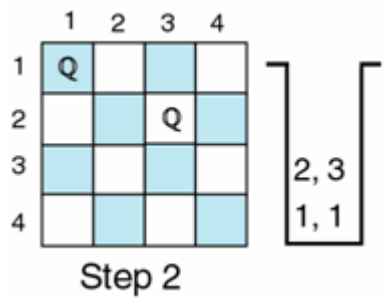
Stack Applications: Backtracking



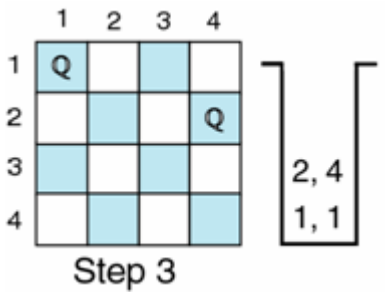
Stack Applications: Backtracking



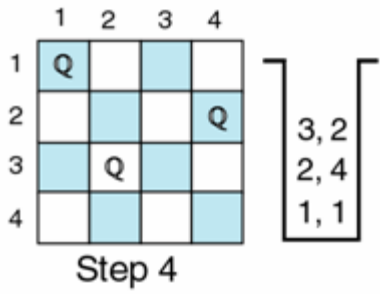
Step 1



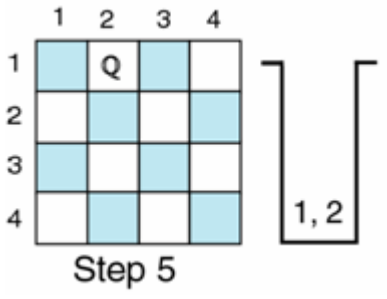
Step 2



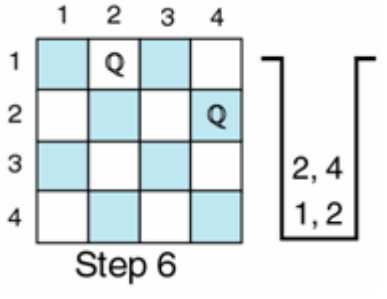
Step 3



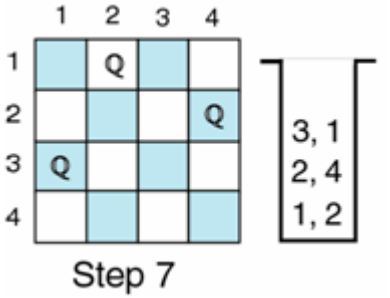
Step 4



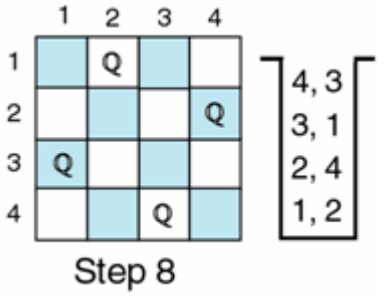
Step 5



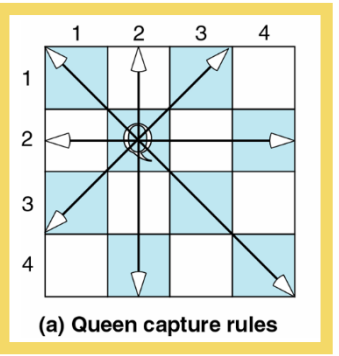
Step 6



Step 7



Step 8



(a) Queen capture rules

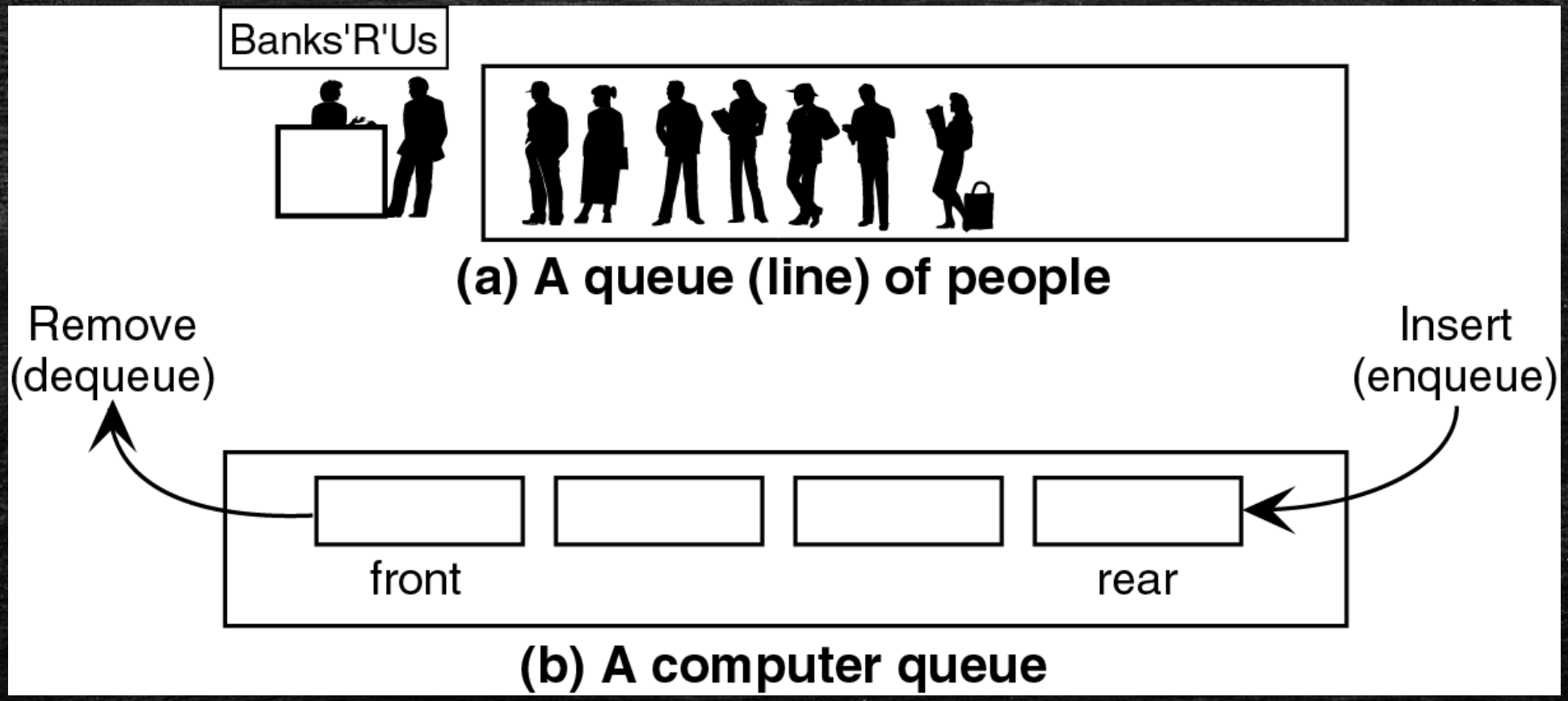
Queue

- Queue เป็นโครงสร้างข้อมูลแบบ FIFO (First-In, First-Out)
- Operations พื้นฐานของ Queue ได้แก่
  - การนำข้อมูลเข้าสู่ Queue เรียกว่า Enqueue
  - การนำข้อมูลออกจาก Queue เรียกว่า Dequeue
  - การเรียกใช้ข้อมูลจากหัวแถวของ Queue เรียกว่า Front
  - การเรียกใช้ข้อมูลจากท้ายแถวของ Queue เรียกว่า Rear
- การสร้าง Queue
  - ใช้ Array แทน queue
  - ใช้ Linked list แทน queue

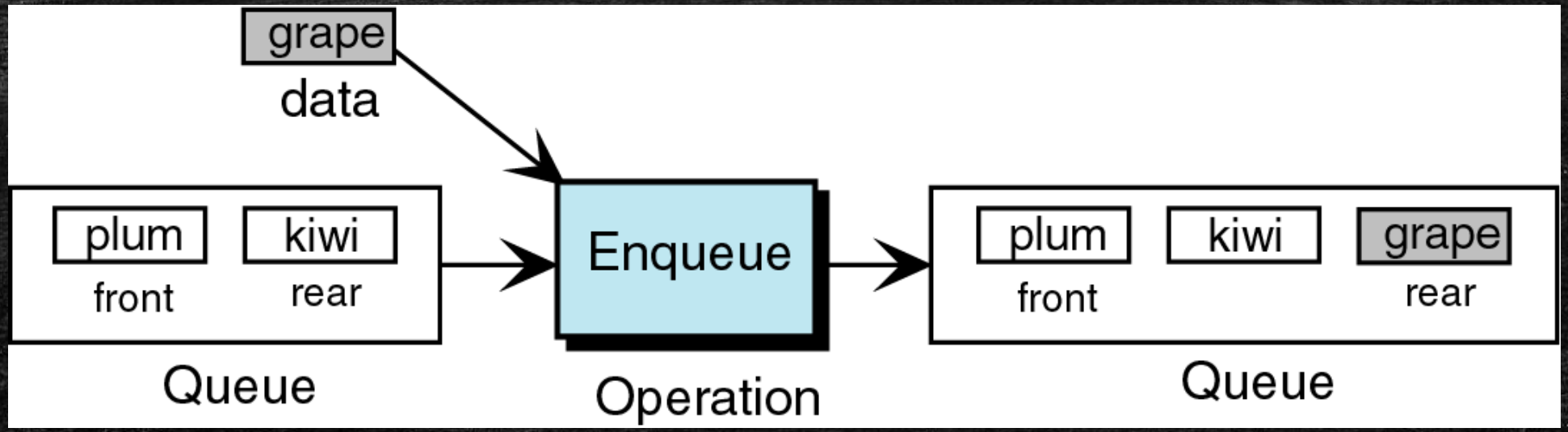
Operations พื้นฐานของ Queue

1. Create queue: สร้าง queue head จาก dynamic memory
2. Enqueue: เพิ่มรายการเข้าไปใน queue
3. Dequeue: ลบรายการออกจาก queue
4. Queue front: เรียกใช้ข้อมูลที่ด้านหน้าของ queue
5. Queue rear: เรียกใช้ข้อมูลที่ด้านหน้าของ queue
6. Empty queue: ตรวจสอบว่า queue ว่างหรือไม่
7. Full queue: ตรวจสอบว่า queue เต็มหรือไม่
   (มีหน่วยความจำ จัดให้ได้หรือไม่)
8. Queue count: บอกจำนวนรายการใน queue
9. Destroy queue: ลบข้อมูลทั้งหมดใน queue และคืนหน่วยความจำ
   ให้ระบบแล้วลบและคืนหน่วยความจำของ head

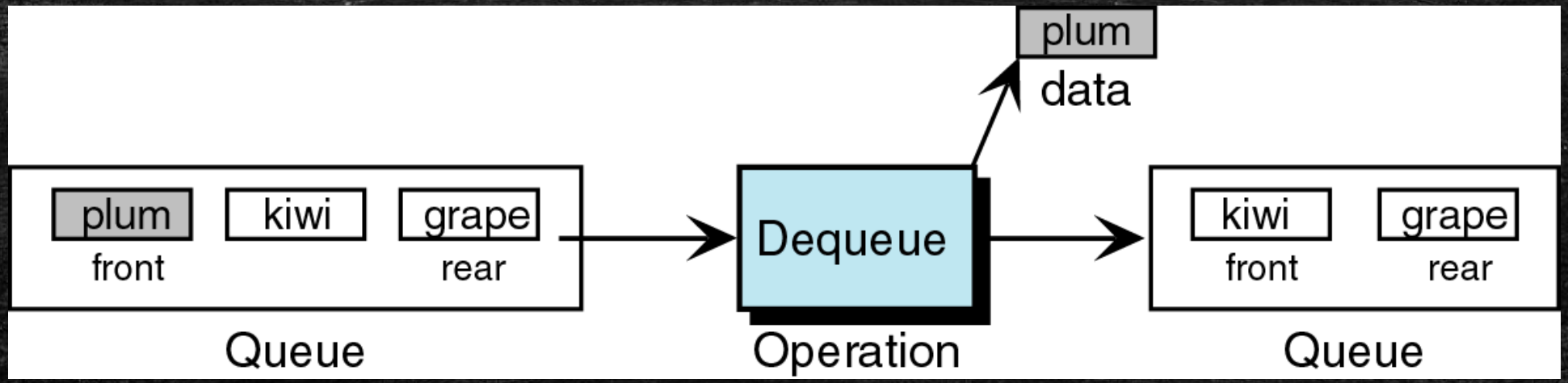
The Queue concept



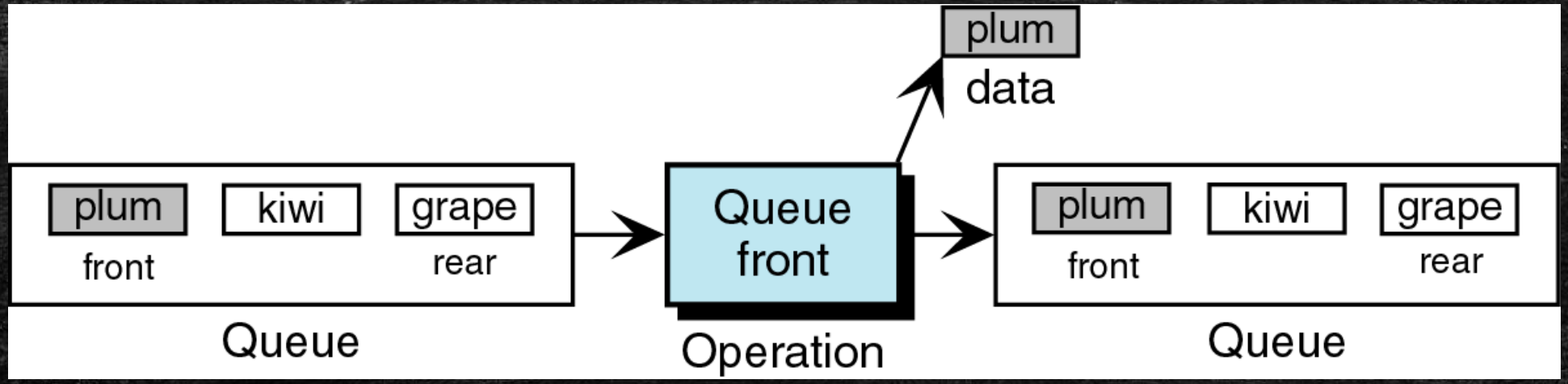
Operation Enqueue



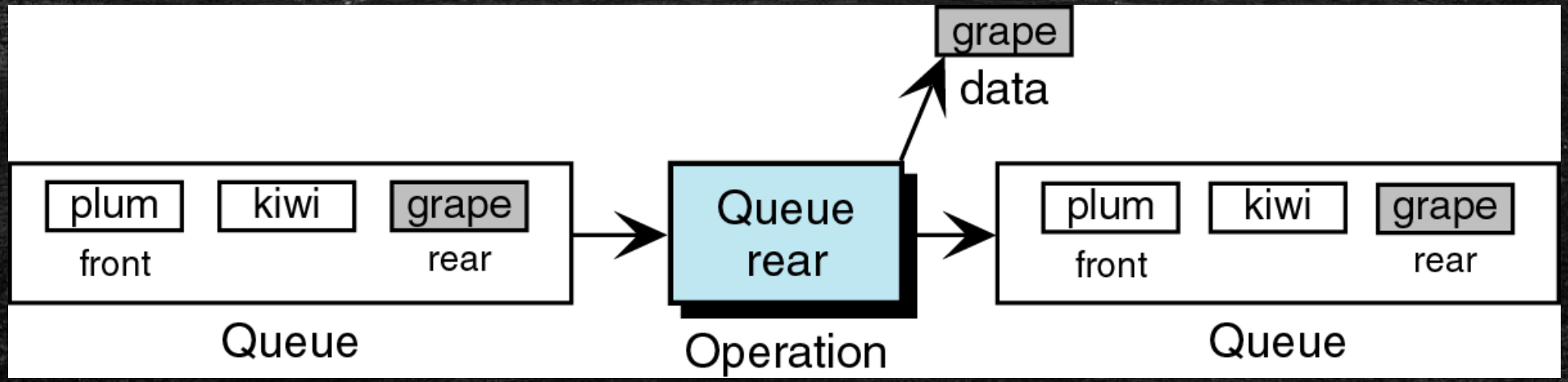
Operation Enqueue



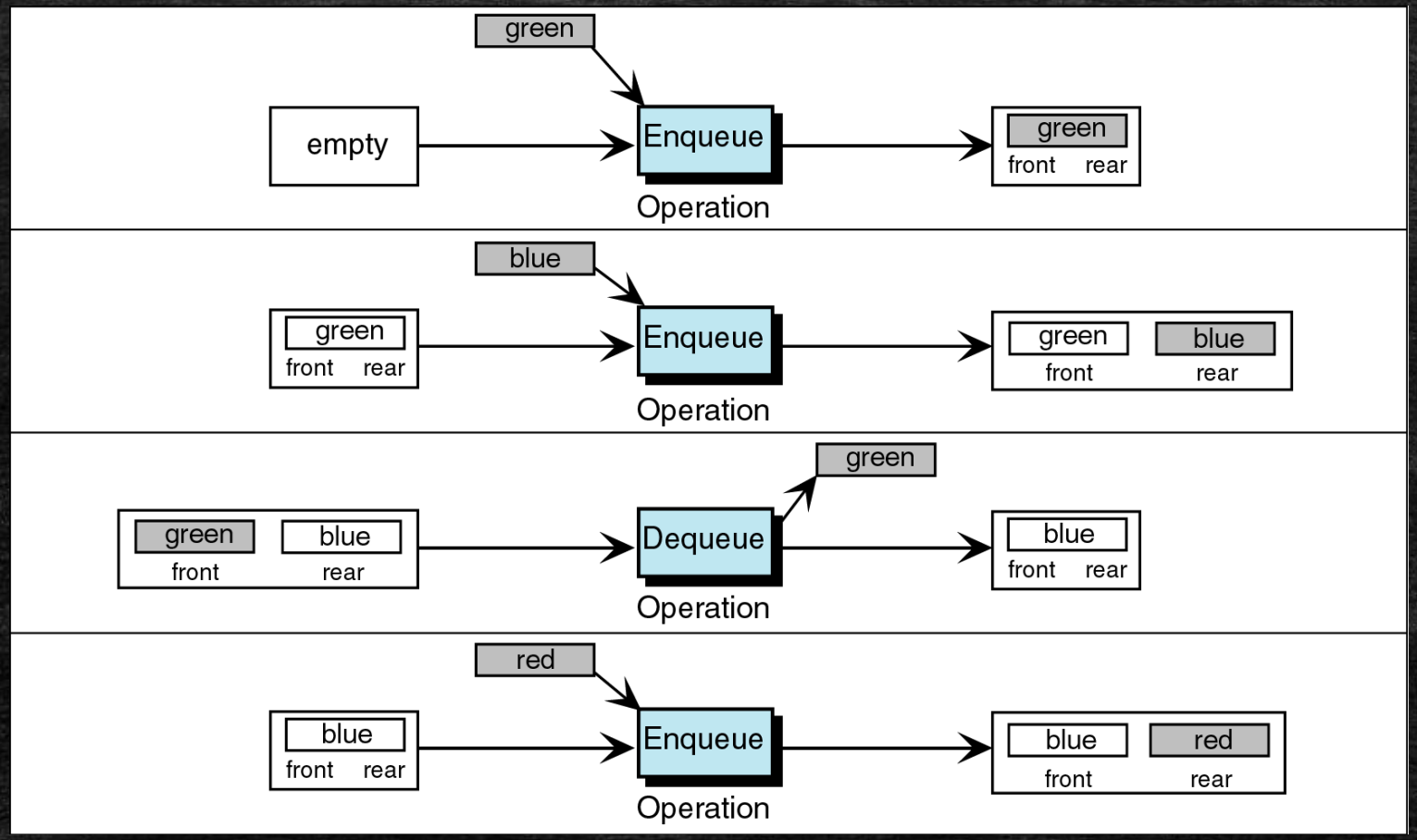
Operation Enqueue



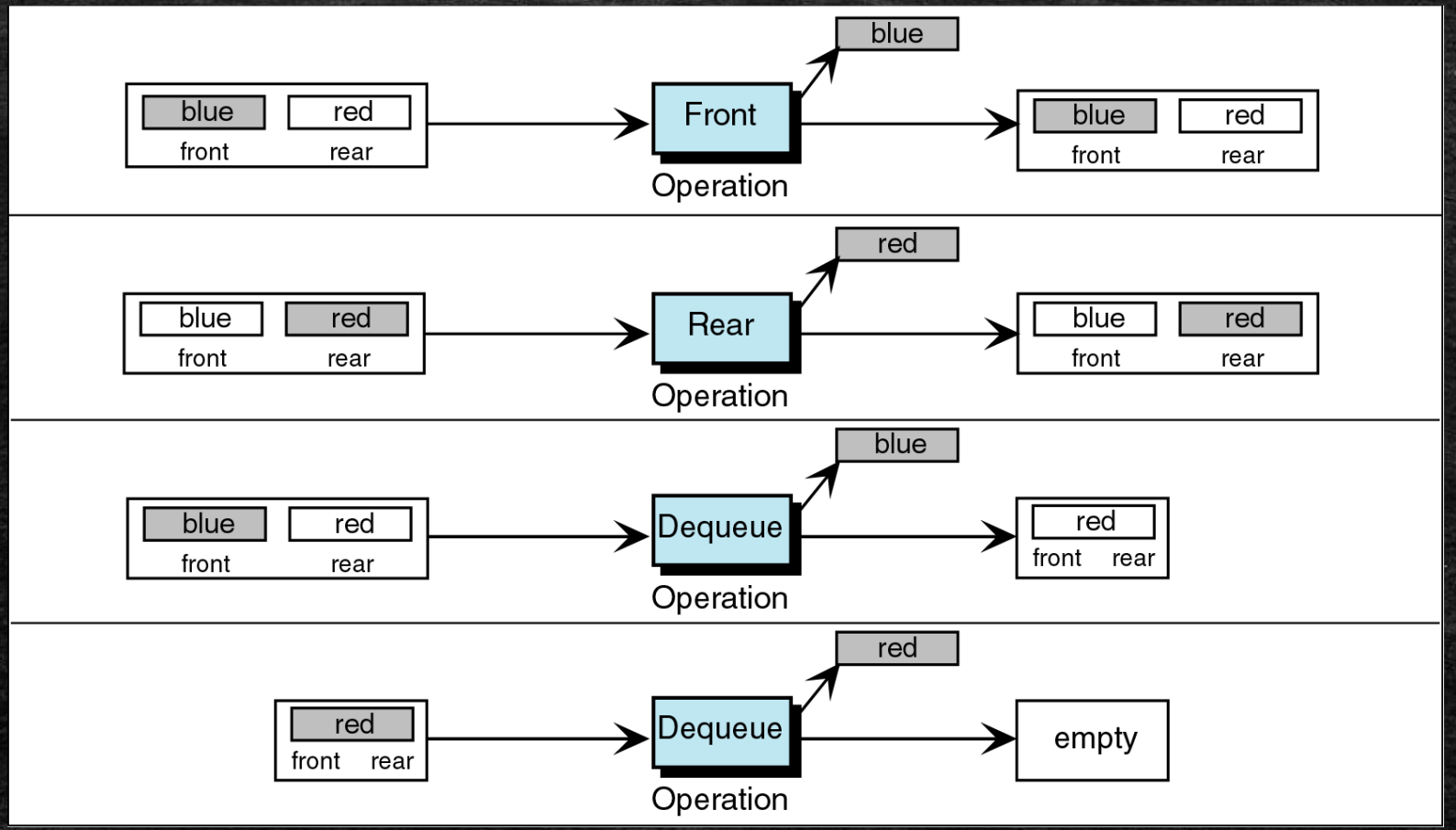
Operation Enqueue



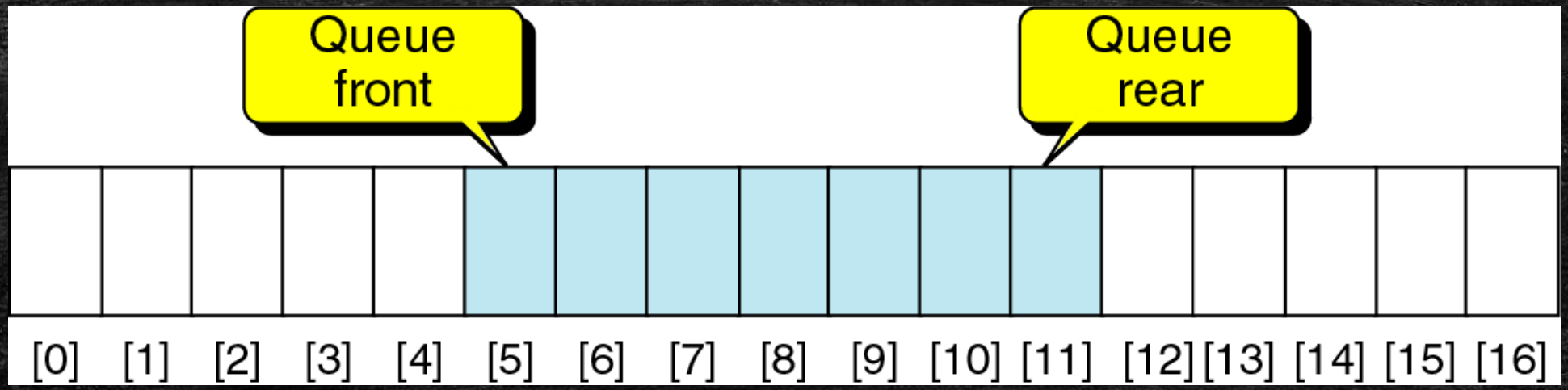
Operation Enqueue



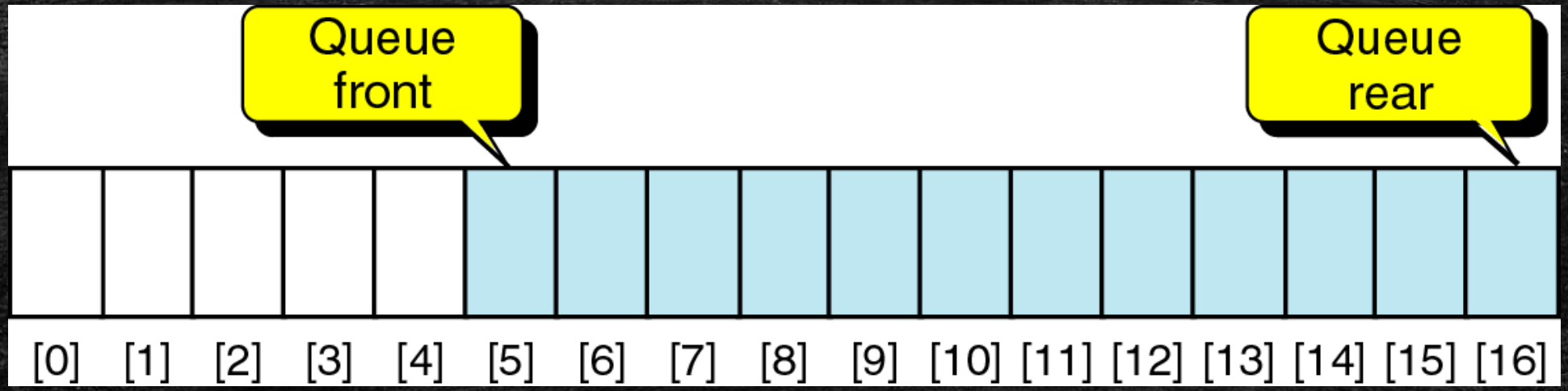
Operation Enqueue



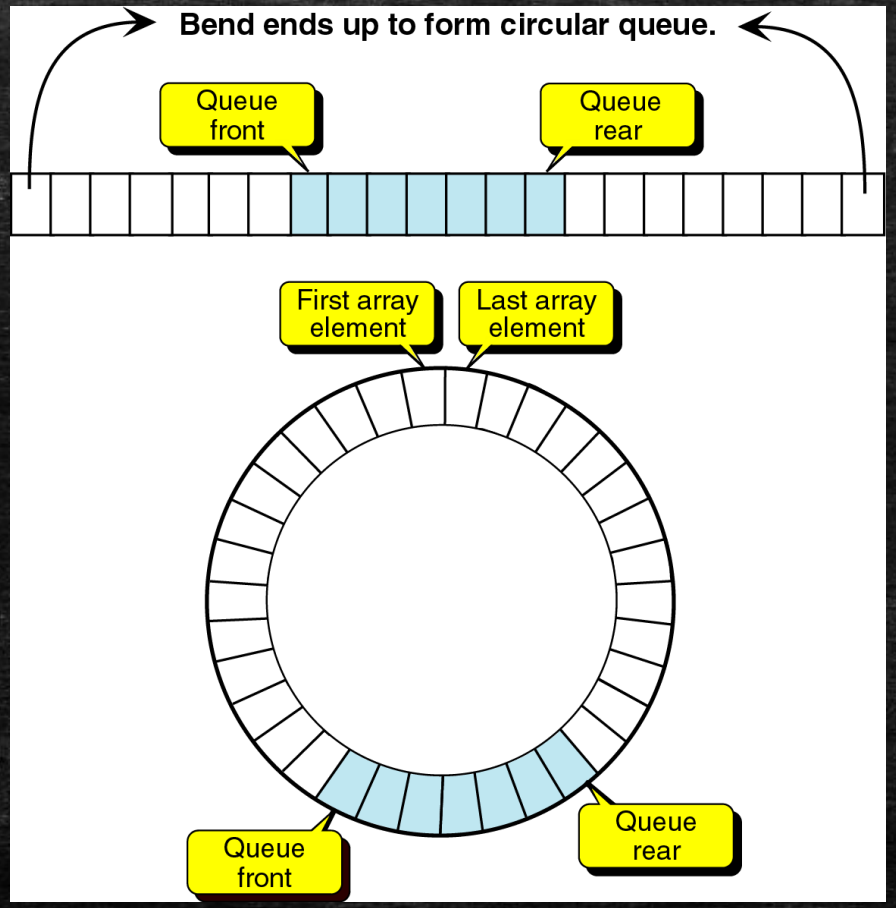
Queue with Array



Queue with Array



Circular Queue



Queue with Array

```
class AQueue{
private:
  int maxSize;                // Maximum size of queue
  int front;                  // Index of front element
  int rear;                   // Index of rear element
  int *listArray;            // Array holding queue elements

public:
  AQueue(int size) {  // Constructor
    // Make list array one position larger for empty slot
    maxSize = size+1;
    rear = 0;  front = 1;
    listArray = new int[maxSize];}

  ~AQueue() { delete [] listArray; } // Destructor

  void clear() { rear = 0; front = 1; } // Reinitialize
```

Queue with Array : Enqueue

```
void enqueue(int it) {      // Put "it" in queue
   if(((rear+2) % maxSize) == front)
       cout<< "Queue is full"<< endl;
   else{
   rear = (rear+1) % maxSize;        // Circular increment
   listArray[rear] = it;
   }
 }
```

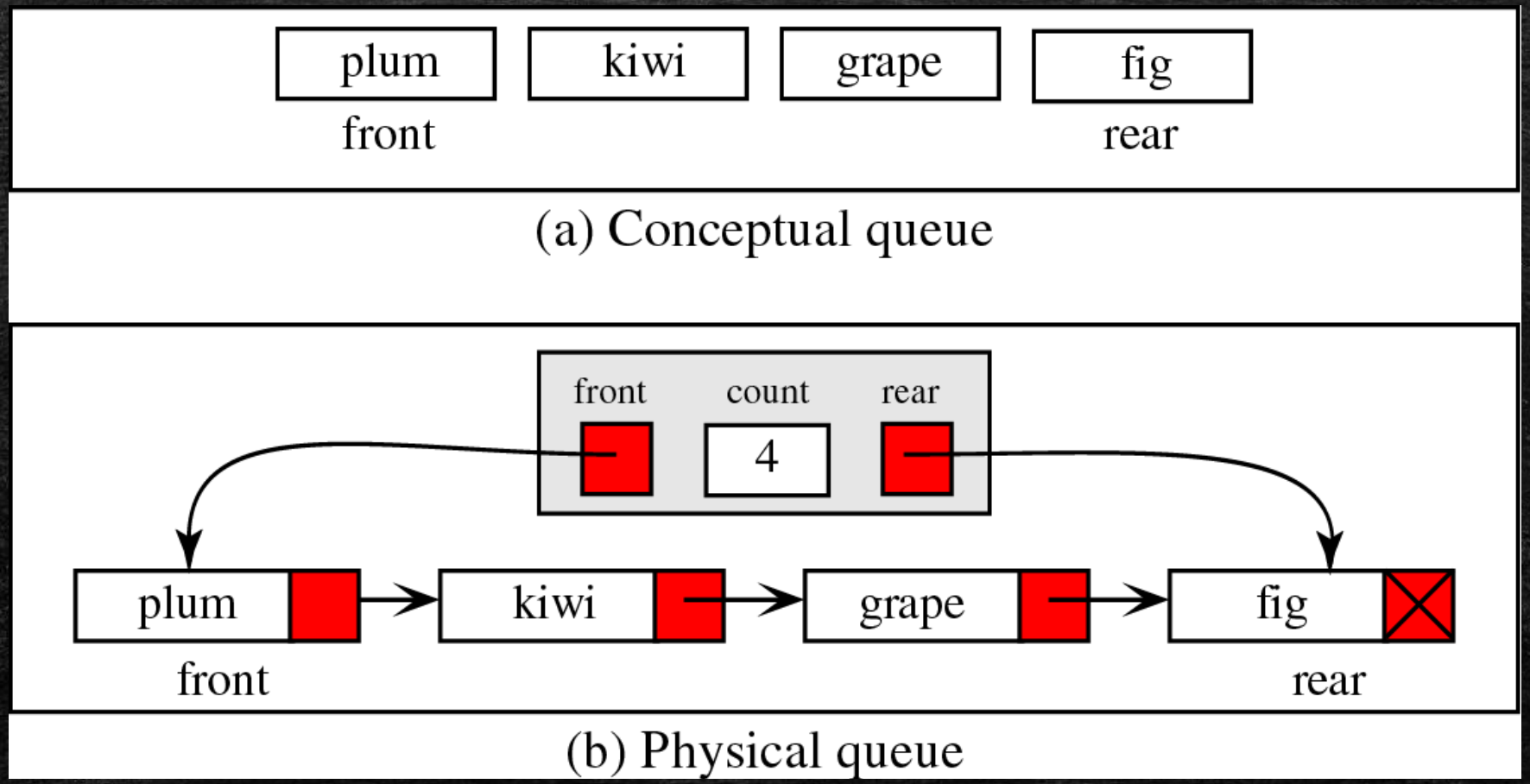
Queue with Array : Dequeue

```
int dequeue() {              // Take element out
    if(length() == 0){
        cout << "Queue is empty" << endl;
        return 0;
    }
    int it = listArray[front];
    front = (front+1) % maxSize;    // Circular increment
    return it;
 }
```

Queue with Array : etc

```
 int frontValue() {  // Get front value
    if(length() == 0)
    {
        cout << "Queue is empty" << endl;
        return 0;
    }
    return listArray[front];
  }

  int length()          // Return length
   { return ((rear+maxSize) - front + 1) % maxSize; }
};
```

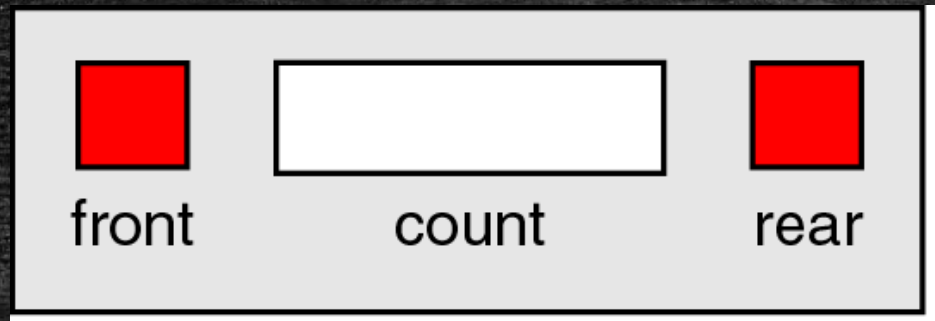
Linked Queue



(a) Conceptual queue

(b) Physical queue

Linked Queue

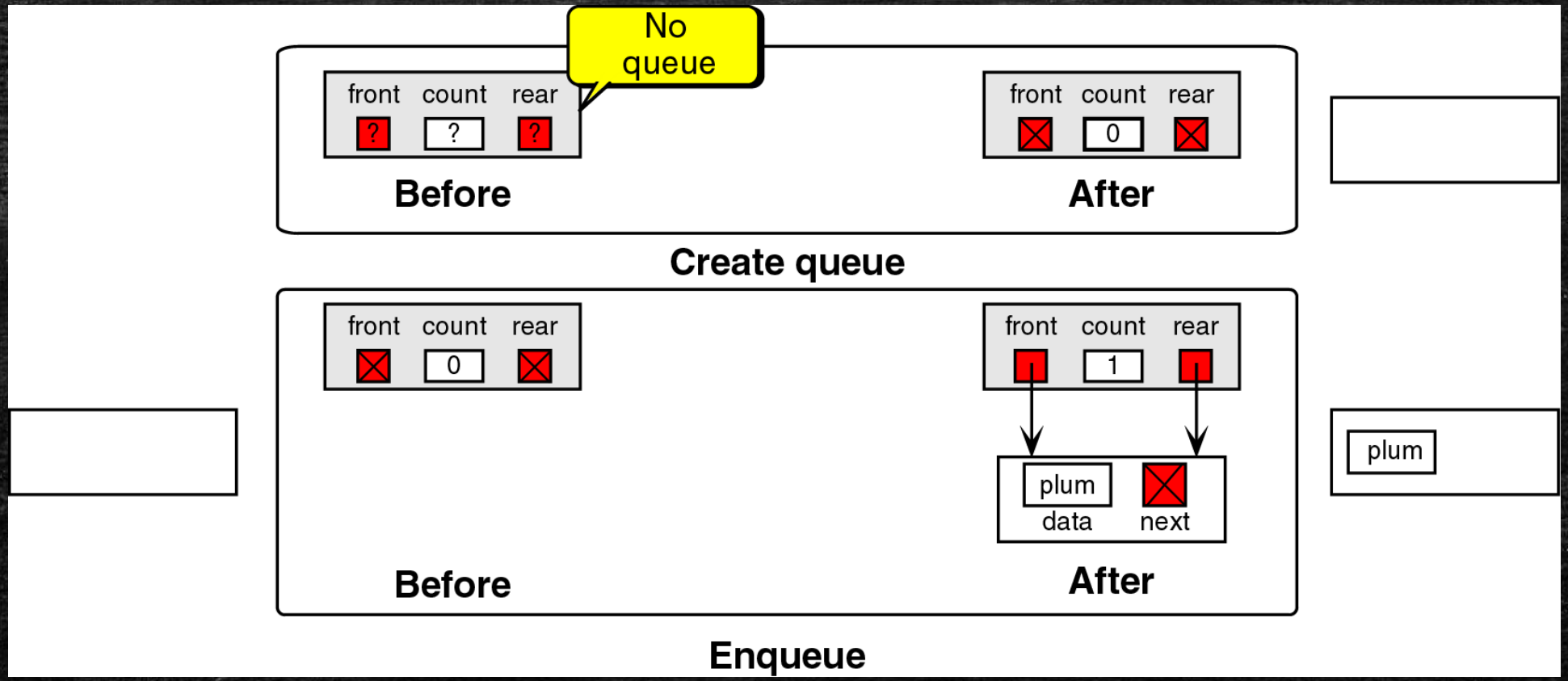


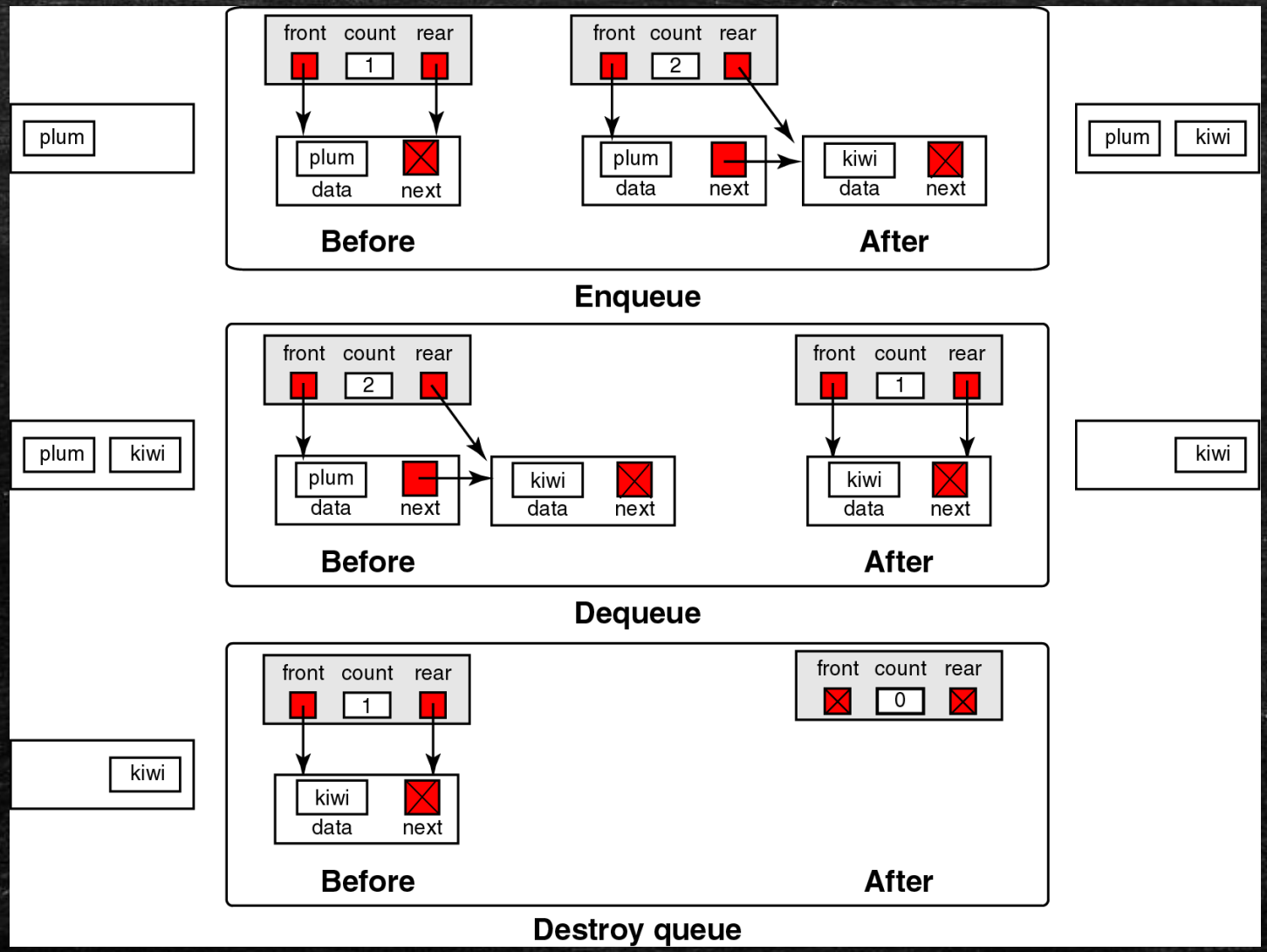
Head structure

Node structure

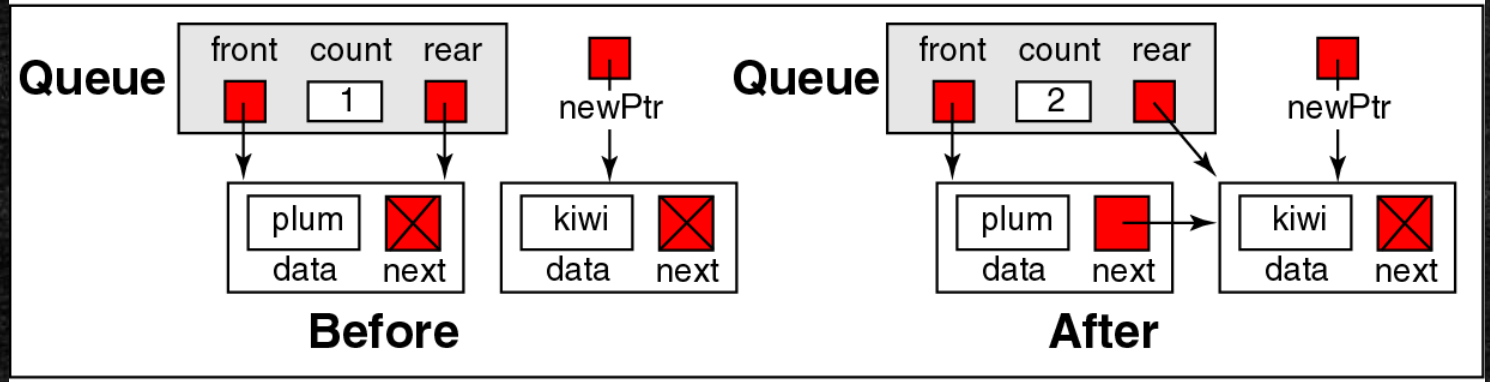
```
queueHead
 front     <node pointer>
 count     <integer>
 rear      <node pointer>
end queueHead


node
   data      <dataType>
   next      <node pointer>
end node
```

Linked Queue : Create and Enqueue



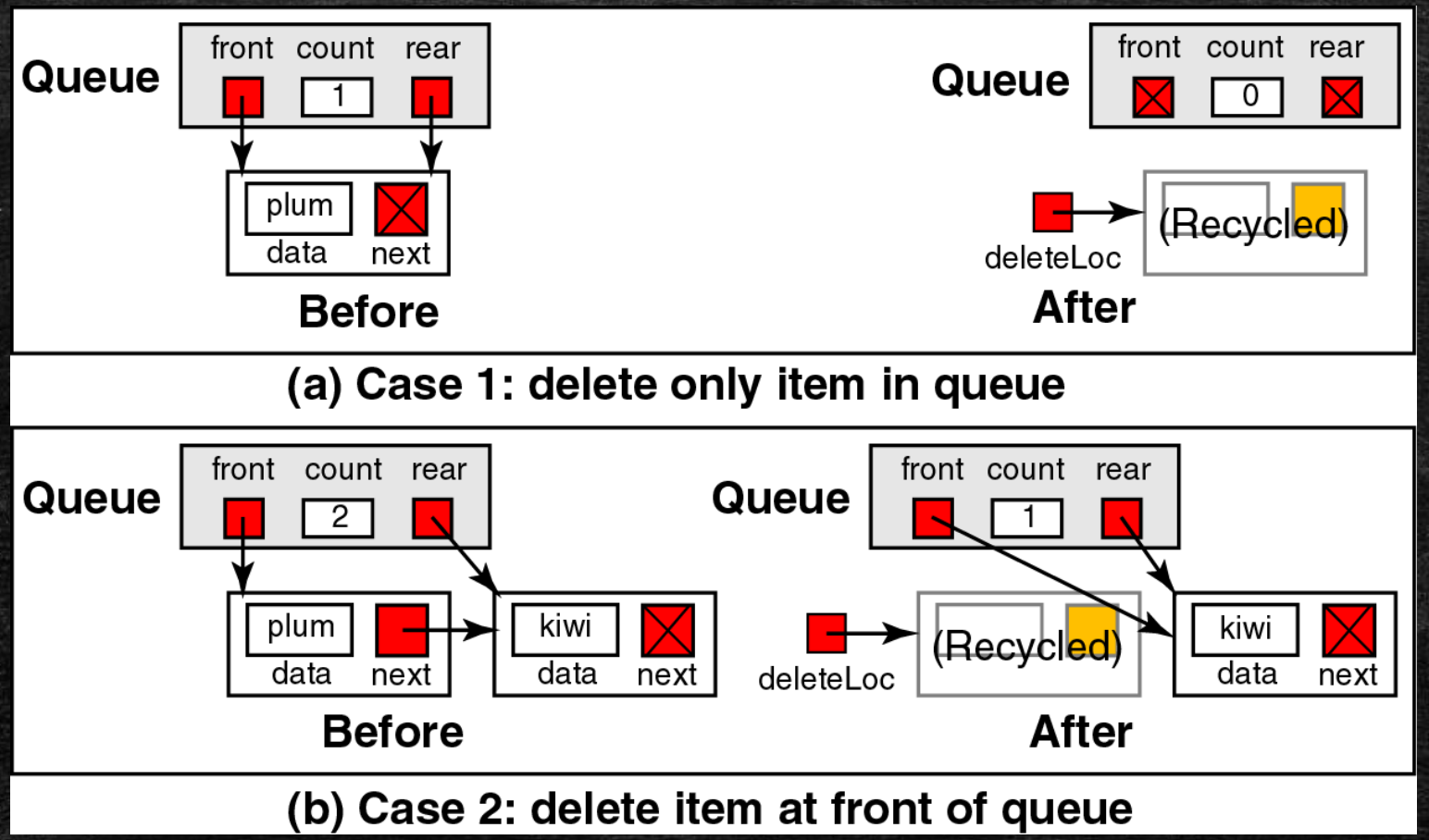

plum
front count rear
1
plum
data
next
Before
front count rear
2
plum
data
next
kiwi
data
next
After
plum kiwi
Enqueue
plum kiwi
front count rear
2
plum
data
next
kiwi
data
next
Before
front count rear
1
kiwi
data
next
After
kiwi
Dequeue
kiwi
front count rear
1
kiwi
data
next
Before
front count rear
0
After
Destroy queue

(a) Case 1: insert into null queue



(b) Case 2: insert into queue with data

Linked Queue : Dequeue



Linked Queue Class

```
class Link {
public:
  int element;       // Value for this node
  Link *next;         // Pointer to next node in list
  // Constructors
  Link(int elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }
};

class LQueue{
private:
  Link* front;        // Pointer to front queue node
  Link* rear;         // Pointer to rear queue node
  int size;          // Number of elements in queue
```

Linked Queue - Clear

```
public:
  LQueue() { front = rear = new Link(); size = 0; }

  ~LQueue() { clear(); delete front; }

  void clear() {              // Clear queue
    while(front->next != NULL) { // Delete link node
      rear = front;
      delete rear;
    }
    rear = front;
    size = 0;
  }
```

Linked Queue - Enqueue

```
void enqueue(int it) { // Put element on rear
    rear->next = new Link(it, NULL);
    rear = rear->next;
    size++;
 }
```

Linked Queue - Dequeue

```
int dequeue() {// Remove element from front
   if(size == 0)
   {
       cout << "Queue is empty" << endl;
       return 0;
   }
   int it = front->next->element;  // Store value
   Link* ltemp = front->next; // Hold dequeued link
   front->next = ltemp->next;       // Advance front
   if (rear == ltemp) rear = front; // Dequeue last
      delete ltemp;                 // Delete link
   size --;
   return it;                       // Return value
 }
```

Linked Queue - etc

```
int frontValue() { // Get front element
    if(size == 0)
    {
        cout << "Queue is empty" << endl;
        return 0;
    }

    return front->next->element;
  }

  int length() { return size; }
};
```

การใช้งานคิว

- พริ้นเตอร์
  - งานต่างๆที่ถูกส่งไปที่พริ้นเตอร์จะมาต่อคิวกัน (จริงๆมีการแซงคิวเพราะความสำคัญของงานไม่เท่ากัน นี่เรียกว่า priority queue)
  - ต่อแถวซื้อของ เราสามารถทำสถานการณ์จำลองเพื่อวิเคราะห์ว่าควรเพิ่มจำนวนแถวหรือไม่ นี่คือ simulation
  - การขอดูไฟล์บนไฟล์เซอร์ฟเวอร์ ผู้ใช้งานขอดูได้ในแบบต่อคิว

Stack Frame

Function call in C++

Memory Static , Stack , Heap

ใน C programming นั้นจะมีการใช้ Memory อยู่ 3 ชนิดคือ

- static : เก็บค่าตัวแปรชนิด global ซึ่งจะอยู่ถาวรจนกว่าจะจบโปรแกรม
- stack : เก็บค่าตัวแปรที่เป็น local หรือตัวแปรที่ประกาศใน function
- heap : dynamic storage (เป็นพื้นที่ความจำขนาดใหญ่การแบ่งใช้ไม่เป็นลำดับ)

STATIC Memory

- static memory จะอยู่ถาวรจนกว่าจะจบโปรแกรม ซึ่งโดยมากจะใช้เก็บค่าตัวแปร global หรือ ตัวแปรที่ประกาศนำด้วย static เช่น
  - **`int val ;`**
- ในหลายๆ ระบบ ตัวแปรนี้จะใช้พื้นที่ในหน่วยความจำ จำนวน 4 bytes หน่วยความจำดังกล่าวนี้อาจมาได้จาก 2 ที่ ถ้า ตัวแปรนี้ประกาศ นอก function ก็จะพิจารณาว่าเป็น global หมายถึง สามารถเข้าถึงจากที่ไหนก็ได้ในโปรแกรม และ ตัวแปร global เป็น static ซึ่งมีชุดเดียวตลอดในโปรแกรม

STATIC Memory

- ตัวแปรใน function เป็น local ซึ่งจะใช้พื้นที่ของ stack แต่ก็สามารถที่จะบังคับให้เป็น ตัวแปร static ได้โดยการนำหน้าด้วย static clause เช่นตัวแปรตัวเดียวกัน หากประกาศใน function แต่ถ้านำหน้าด้วย static ก็จะทำให้ถูก ใช้ในพื้นที่ ใน static memory ได้
  - **`static int val;`**

STACK MEMORY

- หน่วยความจำstack ใช้เก็บค่าตัวแปรใช้ภายใน function (รวมถึงใน main() function ด้วย) หน่วยความจำ stack มีโครงสร้างเป็นแบบ LIFO “Last-In-First-Out” ทุกครั้งที่มีการประกาศตัวแปรใน function ตัวแปรจะถูก Push ลงบนstack และเมื่อ function นั้นจบการทำงานตัวแปรทั้งหมด ที่อยู่ใน function จะถูกลบออกและหน่วยความจำนั้นก็จะว่างลง

STACK MEMORY

- Stack เป็นพื้นที่ของ หน่วยความจำที่พิเศษ ถูกจัดการด้วย OS ดังนั้นเราไม่จำเป็นจะต้องทำการ จอง (allocate) หรือ คืน (deallocate) พื้นที่หน่วยความจำ
- Stack ถูกแบ่งออกเป็น frame ต่อเนื่องกัน ในขณะที่ function ถูกเรียกนั้น stack frame ใหม่จะถูก จองใช้งาน

STACK MEMORY

- ข้อจำกัดของขนาดของหน่วยความจำ stack นั้นแตกต่างกันไปตาม OS (เช่น MacOS มีขนาด stack เท่ากับ 8MB) ถ้าโปรแกรม พยายามใช้ stack มากเกินไป จะเกิด stack overflow ซึ่งก็หมายถึง หน่วยความจำ stack ถูกจองใช้หมด นอกจากนี้การเขียน recursion ที่ไม่ถูกต้องก็เป็นอีกสาเหตุที่ทำให้ เกิด stack overflow ด้วย

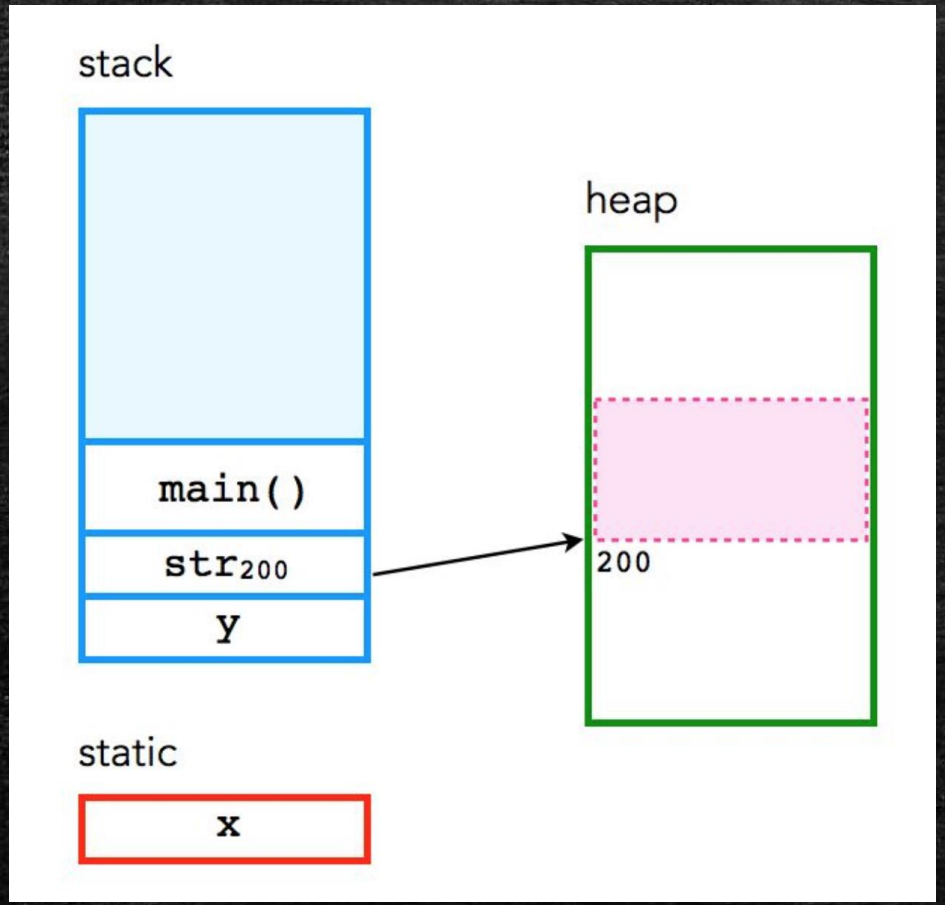
HEAP Memory

- เป็นพื้นที่สำหรับการเรียกใช้แบบ dynamic ตรงข้ามกับการใช้งานแบบ stack หรือเรียกอีกอย่างหนึ่งว่า “free store “ ผู้ใช้เป็นผู้จองใช้งานเองโดยผ่าน function เช่น malloc() และคืน เช่น free() การใช้งานโดยไม่คืน memory ทำให้เกิด memory leak ทำให้ process อื่น ไม่สามารถใช้งานได้
- ขนาดของ HEAP ซึ่งขึ้นอยู่กับ physical memory และตัวแปรที่สร้างใน heap สามารถเข้าถึงได้จากทุกที่ในโปรแกรม ผ่านการใช้งาน pointers

Memory: Static , Stack , Heap

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

int x;

int  main(void)
{
        int y;
        char *str;


        y=4;
        printf("stack memory: %d\n", y);


        str = malloc(100*sizeof(char));
        str[0] = 'm';
        printf("heap memory: %c\n", str[0]);
        free(str);

        return 0;
}
```

Memory: Static , Stack , Heap



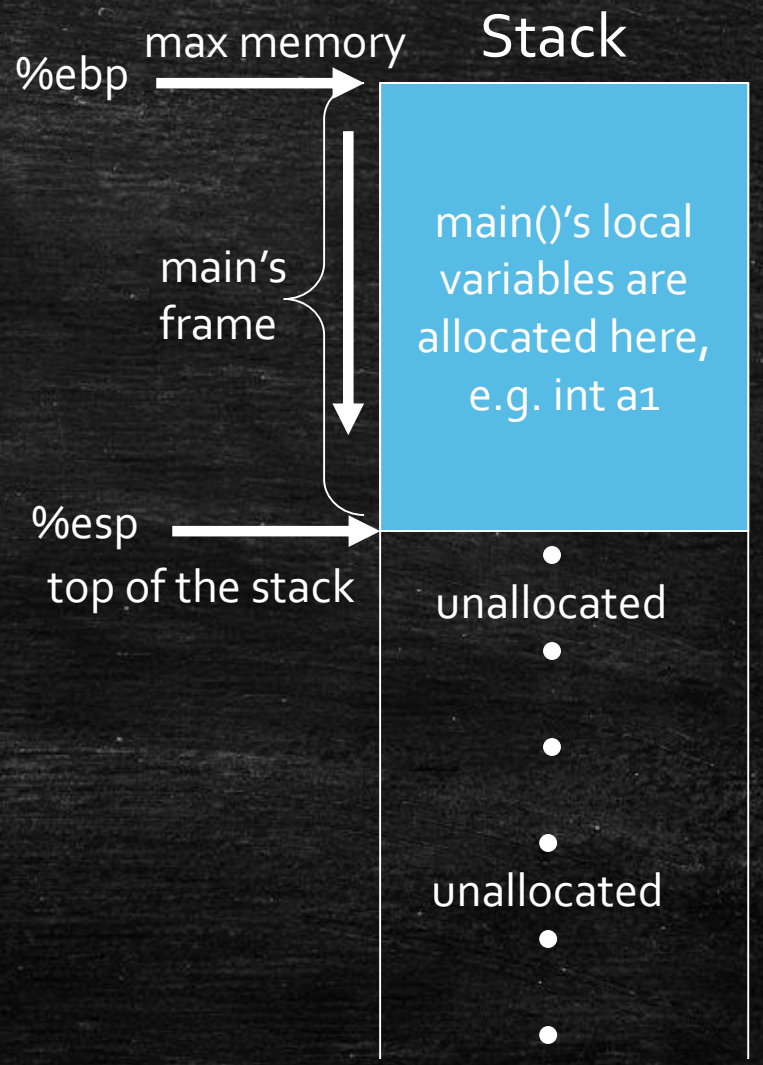
Memory: Static , Stack , Heap

- ตัวแปร x เป็น static เพราะว่า เป็นการประกาศแบบ global
- ตัวแปร y และ str เป็น stack ซึ่งจะถูกคืนพื้นที่เมื่อโปรแกรมจบ
- function malloc() นั้นใช้สำหรับการจองพื้นที่ 100 ช่อง ของ หน่วยความจำ heap แต่ละช่องมีขนาดตาม size ของ char ให้กับตัวแปร str
- function free() สำหรับคืนพื้นที่ ที่ ชี้โดย str

Calling a Function

```
main() {
    int a1;
    ...
    foo1(a1);
    ...
}
```

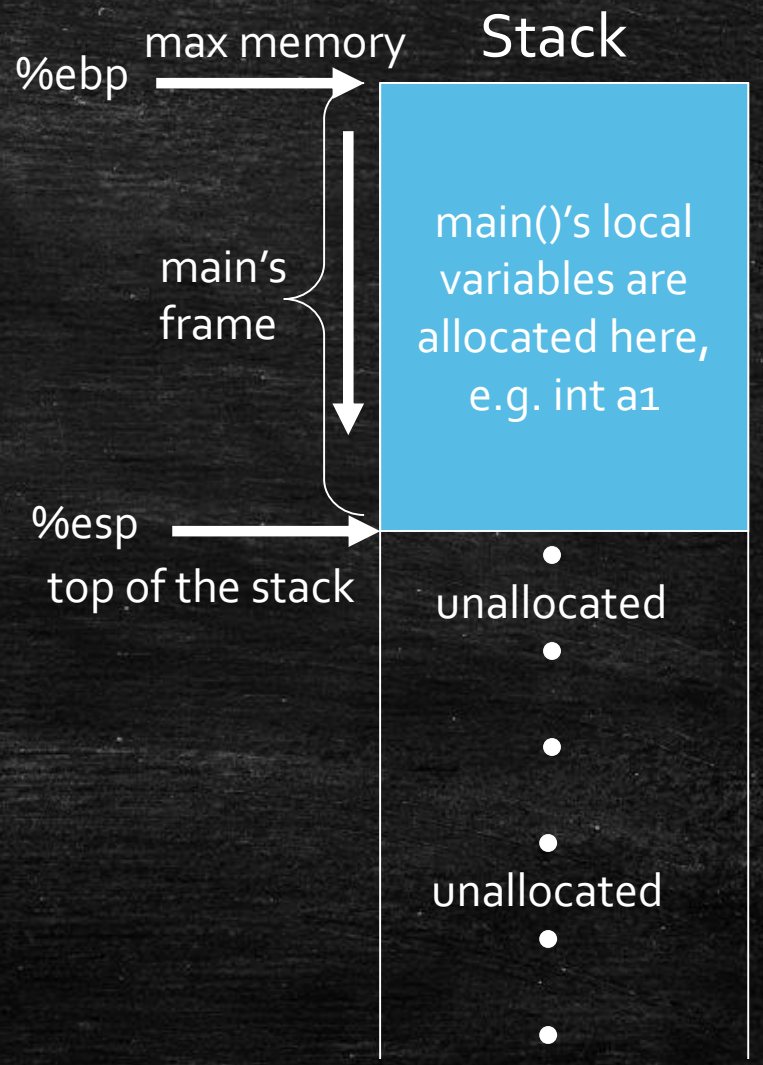
PC



Calling a Function

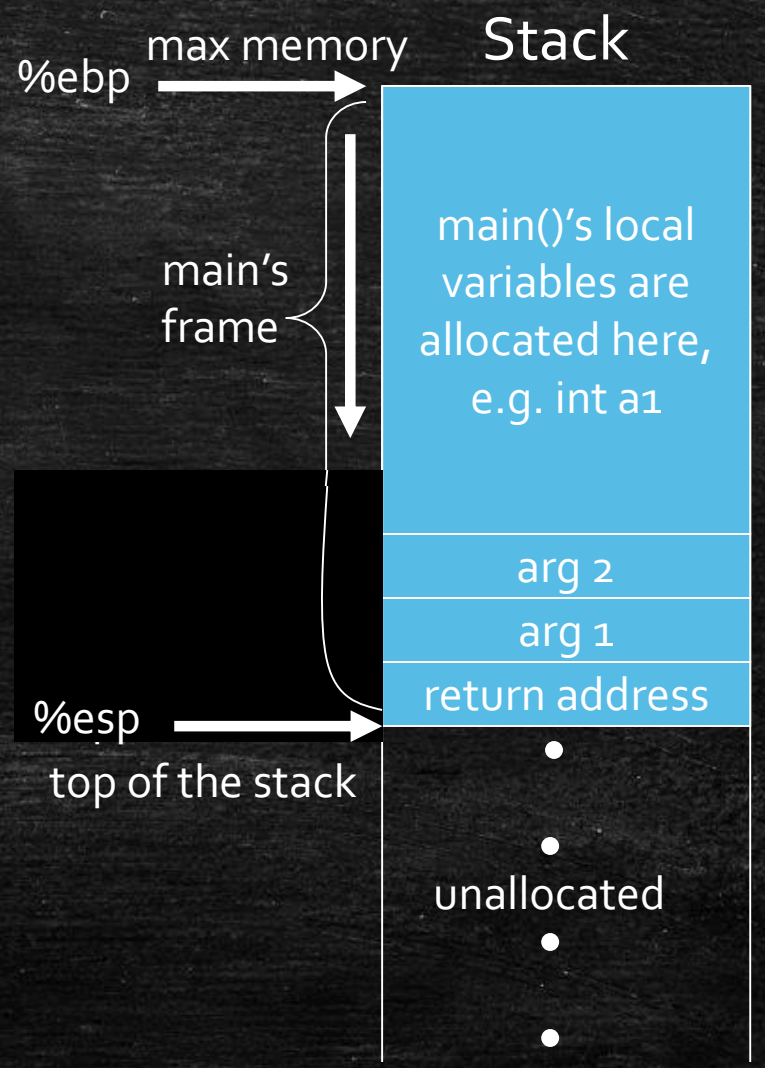
```
main() {
    int a1;
    ...


    foo1(a1);
    ...
}
```

PC



Calling a Function

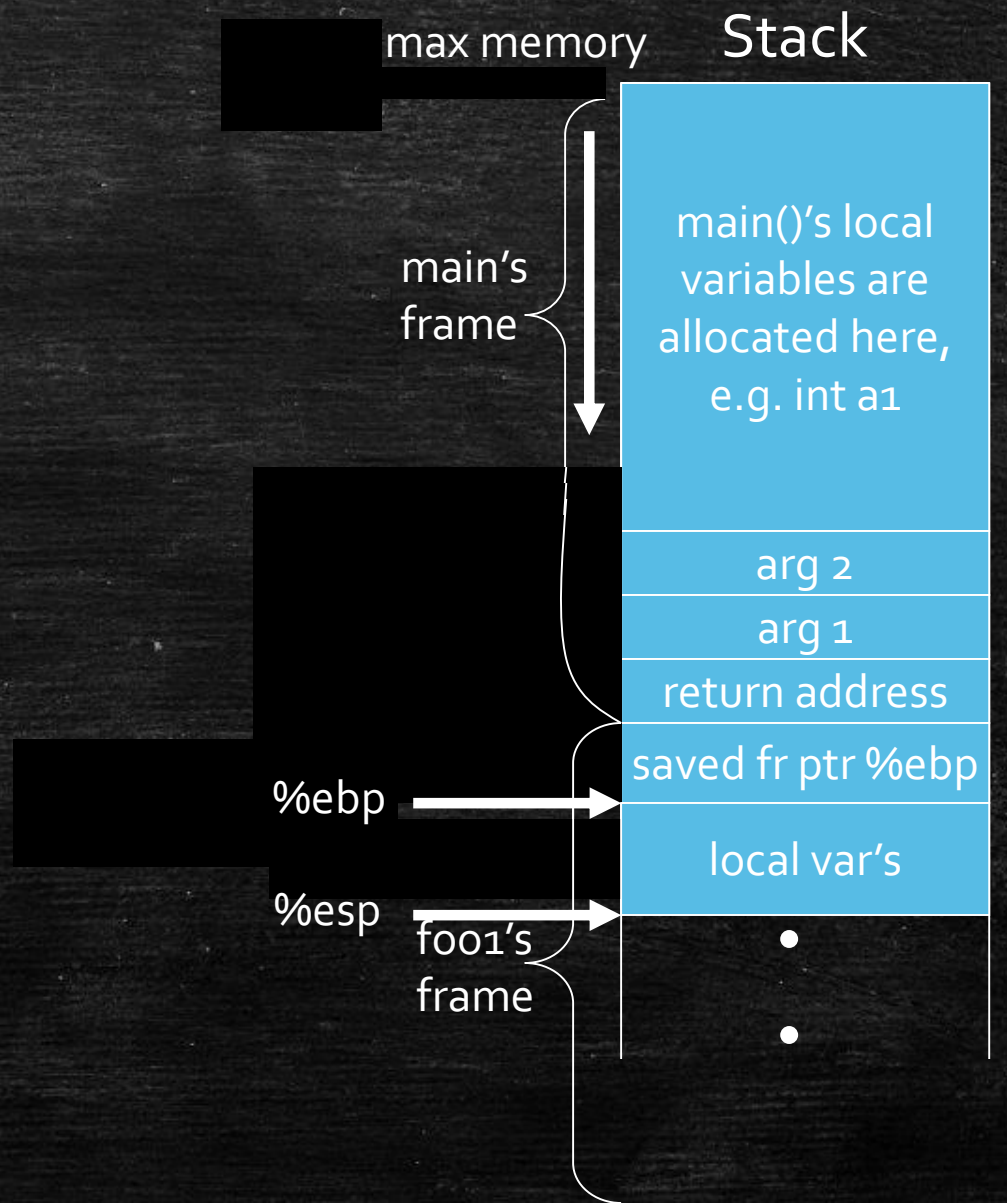
```
main() {
    int a1, b2;
    ...

PC →

    foo1(a1, b2);
    ...
}
```



Entering a Function

```
foo1(int v1, v2) {

    local var's

    ...

}
```

PC →



Exiting a Function

```
foo1(int v1, v2) {

    local var's
       ...
}
```

PC →