Data Structures: Lists: Stack and Queue 2

YoungWoon Cha
(Slide credits to Won Kim)

Spring 2022

Circular Queue

Problem of Non-Circular Queue

peach
apricot
melon
orange
dragon eye
pear
cherry
banana
apple

rear=8

front=7

garbage

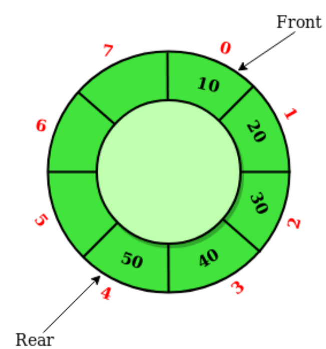
…

garbage

Wasted space !!

Circular Queue (Ring Buffer)

- The queue forms a circle (ring).
- The queue may be implemented using a linked list or an array.
- Once an insertion reached the end of the array, the element will be inserted into the start of the array.



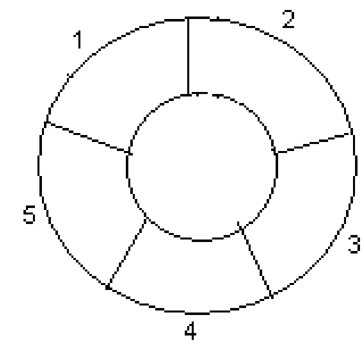
- A queue using a linked list needs Ring Buffer?

Circular Queue: Method1

- Array size N

- Initially, front = rear = 0
  - This means Empty Queue!

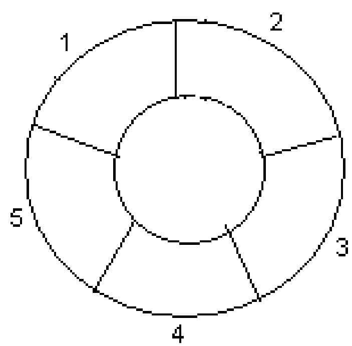
- This method does not use the array index 0.
  - [data_type] Queue[N+1]



N=5

Circular Queue: Method1

- Insertion (enqueue)
  - 1. next_rear = (rear % N) + 1
  - 2. If the queue is empty, set front=1 and go to step 4.
  - 3. If the queue is full, “circular queue overflow” (finish)
  - 4. Set rear = next_rear
  - 5. Store in rear of array
  - 6. Finish

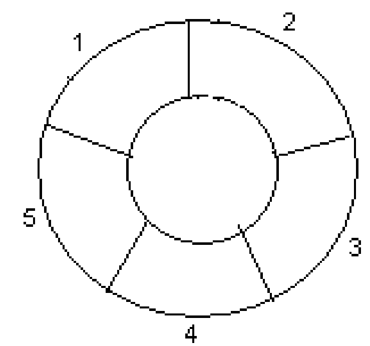


N=5

- Full state
  - front == next_rear
- Storing an item
  - queue[rear] = new_item

Circular Queue: Method1

- Deletion (dequeue)
  - 1. If the queue is empty, “circular queue underflow” (finish)
  - 2. get the front item
  - 3. If front == rear, set as empty queue and go to step 5.
  - 4. front = (front % N) + 1;
  - 5. return the item



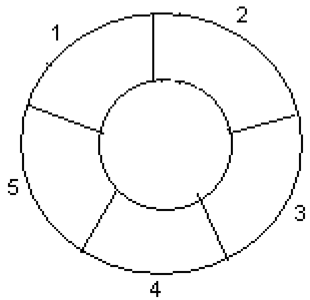
- Empty Queue
  - front = rear = 0

- Fetch an item
  - return_item = queue[front]

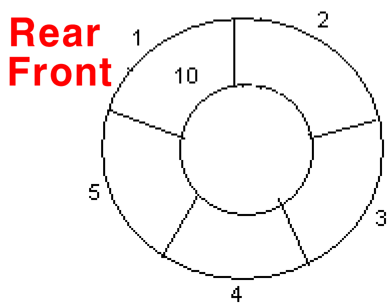
N=5

Example

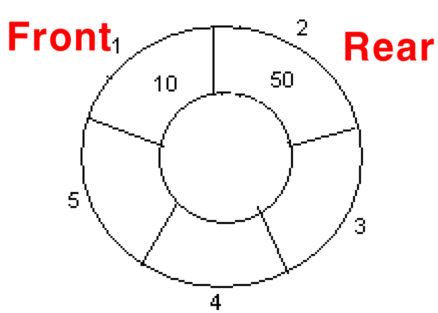
1. Initially, Rear = 0, Front = 0.



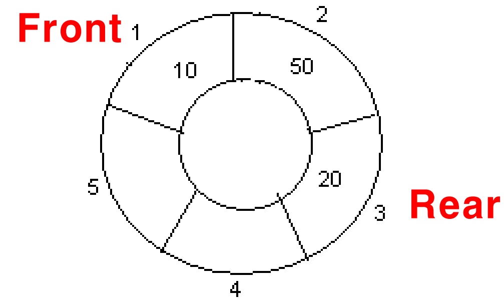
2. Insert 10, Rear = 1, Front = 1.



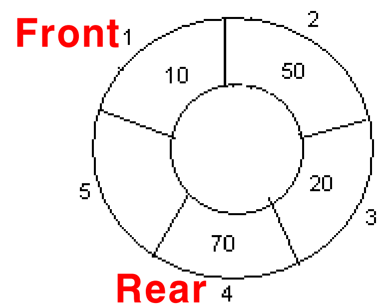
3. Insert 50, Rear = 2, Front = 1.



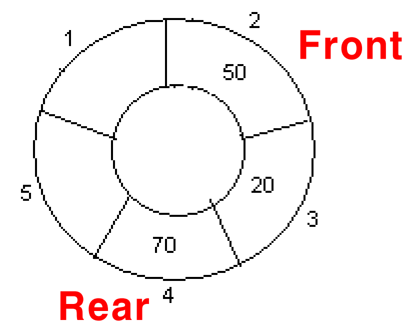
4. Insert 20, Rear = 3, Front = 1.



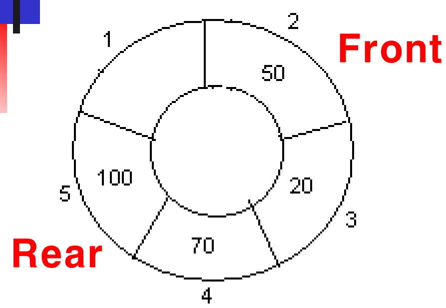
5. Insert 70, Rear = 4, Front = 1.



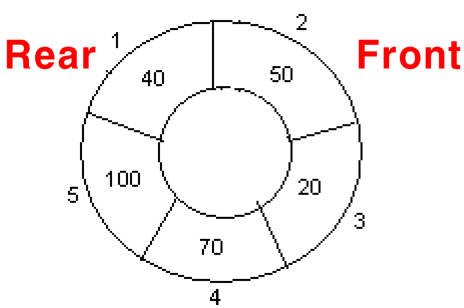
6. Delete front, Rear = 4, Front = 2.



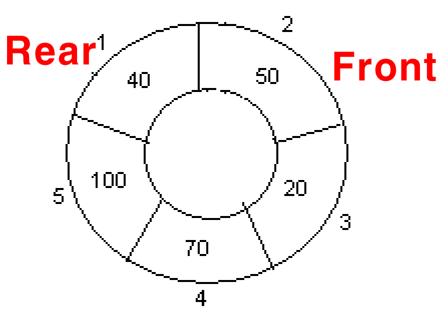
7. Insert 100, Rear = 5, Front = 2.



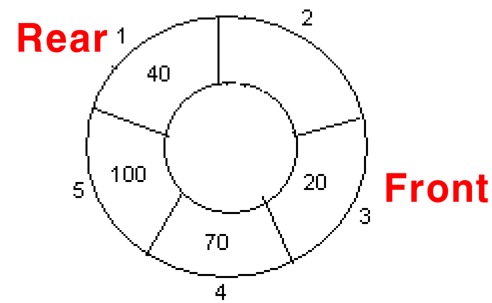
8. Insert 40, Rear = 1, Front = 2.



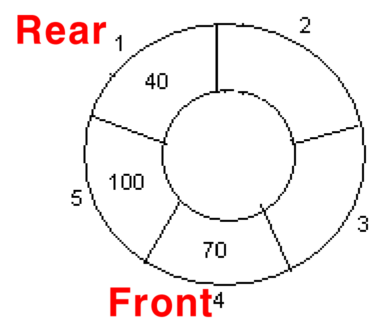
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



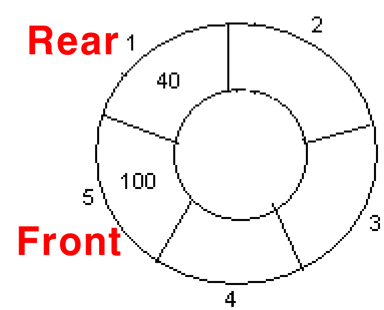
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.

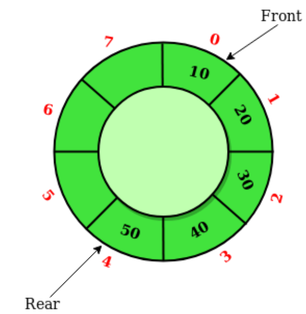


12. Delete front, Rear = 1, Front = 5.



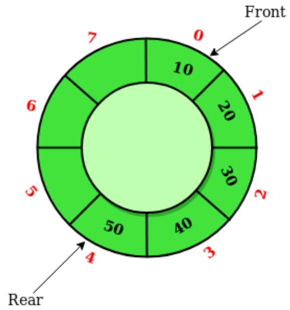
Circular Queue (Method 2)

```c
#include <stdio.h>

#define Q_SIZE 8
struct CIRCULAR_QUEUE
{
  int arr[Q_SIZE];// array to store queue elements
  int capacity;   // maximum number of elements
  int front;      // point to the front element in the queue
  int rear;       // point to the last element in the queue
  int count;      // the current number of elements
} Queue = { {0,}, Q_SIZE, 0, Q_SIZE -1, 0 };


bool q_isEmpty() {
  return (Queue.count == 0);
}
bool q_isFull() {
  return (Queue.count == Queue.capacity);
}
```



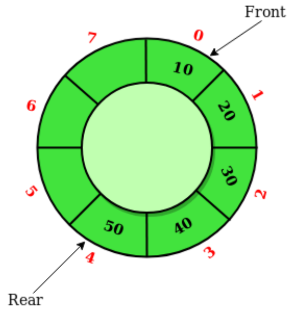
Circular Queue (Method 2)

```
void enqueue(int x)
{
  if (q_isFull()) // check for queue overflow
  {
    printf("Queue is Full\n");
    return;
  }
  Queue.rear = (Queue.rear + 1) % Queue.capacity;
  Queue.arr[Queue.rear] = x;
  Queue.count++;
 printf("%d is inserted into Q[%d]\n", x, Queue.rear);
}
```



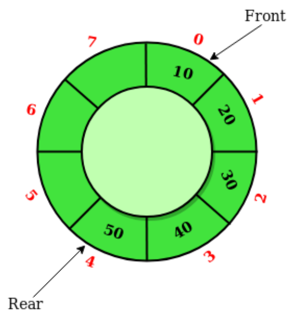
Circular Queue (Method 2)

```c
int dequeue()
{
  if (q_isEmpty()) // check for queue underflow
  {
    printf("Queue is Empty\n");
    return 0;
  }
  int x = Queue.arr[Queue.front];
  printf("%d is removed from Q[%d]\n", x, Queue.front);
  Queue.front = (Queue.front + 1) % Queue.capacity;
  Queue.count--;
  return x;
}
```



Circular Queue (Method 2)

```c
int main()
{
  int val;
  enqueue(1); // Q[0] <- 1
  enqueue(2); // Q[1] <- 2
  enqueue(3); // Q[2] <- 3

  val = dequeue(); // 1 <- Q[0]

  enqueue(4); // Q[0] <- 4

  val = dequeue(); // 2 <- Q[1]
  val = dequeue(); // 3 <- Q[2]
  val = dequeue(); // 4 <- Q[0]

  if (q_isEmpty()) // q is empty now
  {
    printf("The queue is empty\n");
  }
  else {
    printf("The queue is not empty\n");
  }
  return 0;
}
```

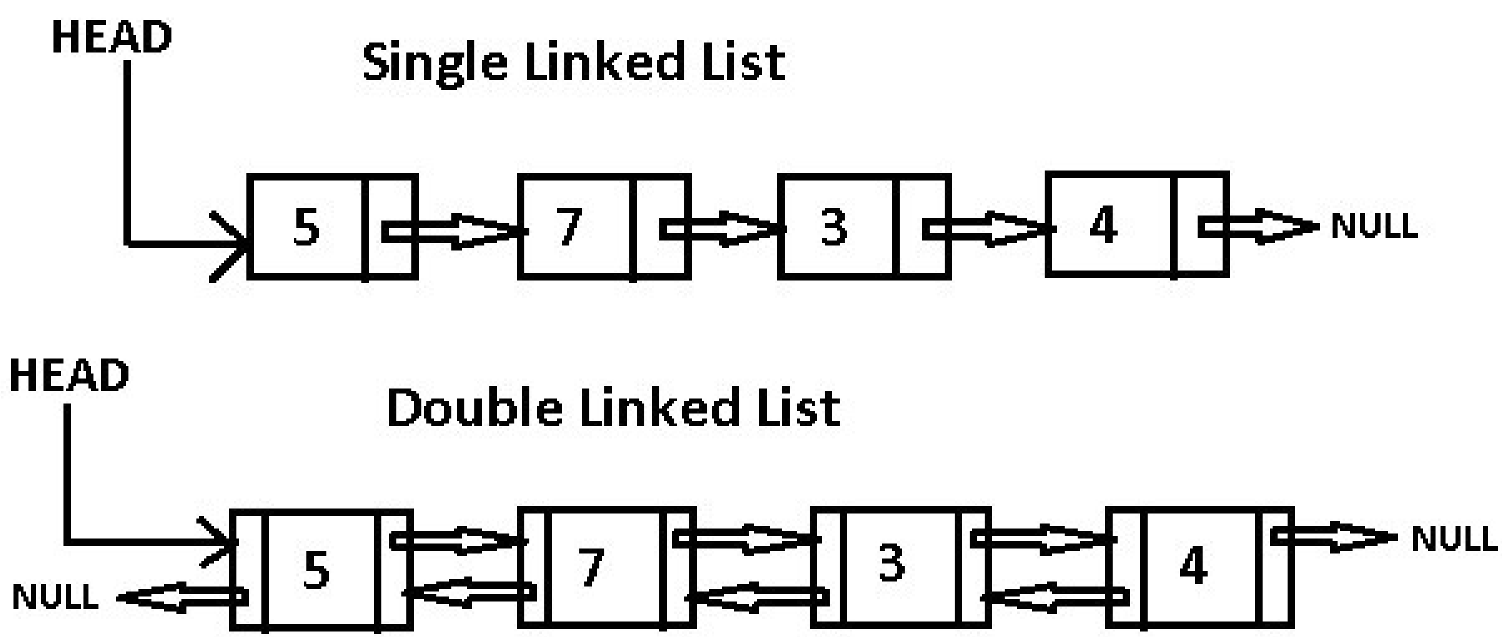


- What is the difference between the method 1 & 2?

Review on Linked List

Linked List

- Singly linked list
- Doubly linked list (not to be covered)



Defining a Data Node in C (Self-Referential Structure)

- One-way chain of data nodes
  - data node = (data, pointer to next data)

node

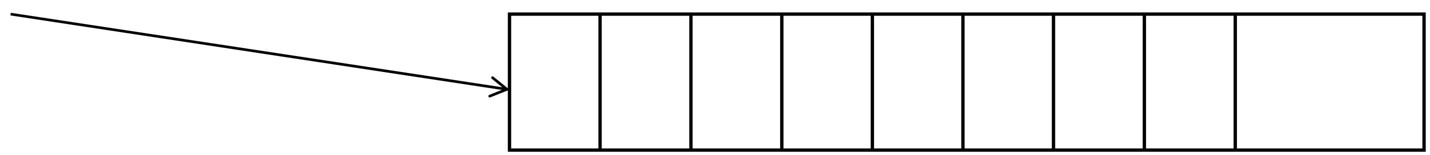
100	next

```
struct  NODE {
    int              key;
    struct NODE     *next;
} node;
```

Dynamic Memory Allocation: malloc( )

23406

char *cp;



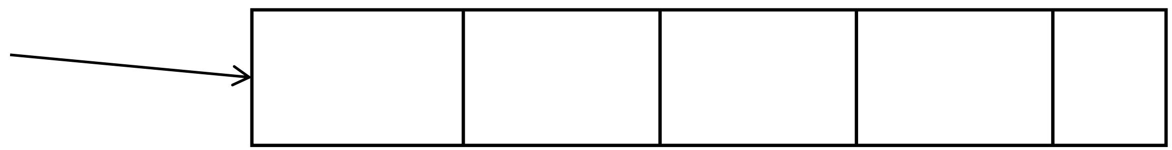
```
cp = (char *) malloc (1000);
      /* malloc returns pointer of type void.
         This needs to be type cast to desired type  */

if (cp == (char *) NULL) {
     printf ("malloc failed");
     exit(1);
}
/* malloc may fail to allocate memory  */
```

Allocating Memory for a struct Array

```
struct  NODE {
    int  key;
    struct NODE  *next;
    };
```

23406

```
struct NODE  *pNode;
```

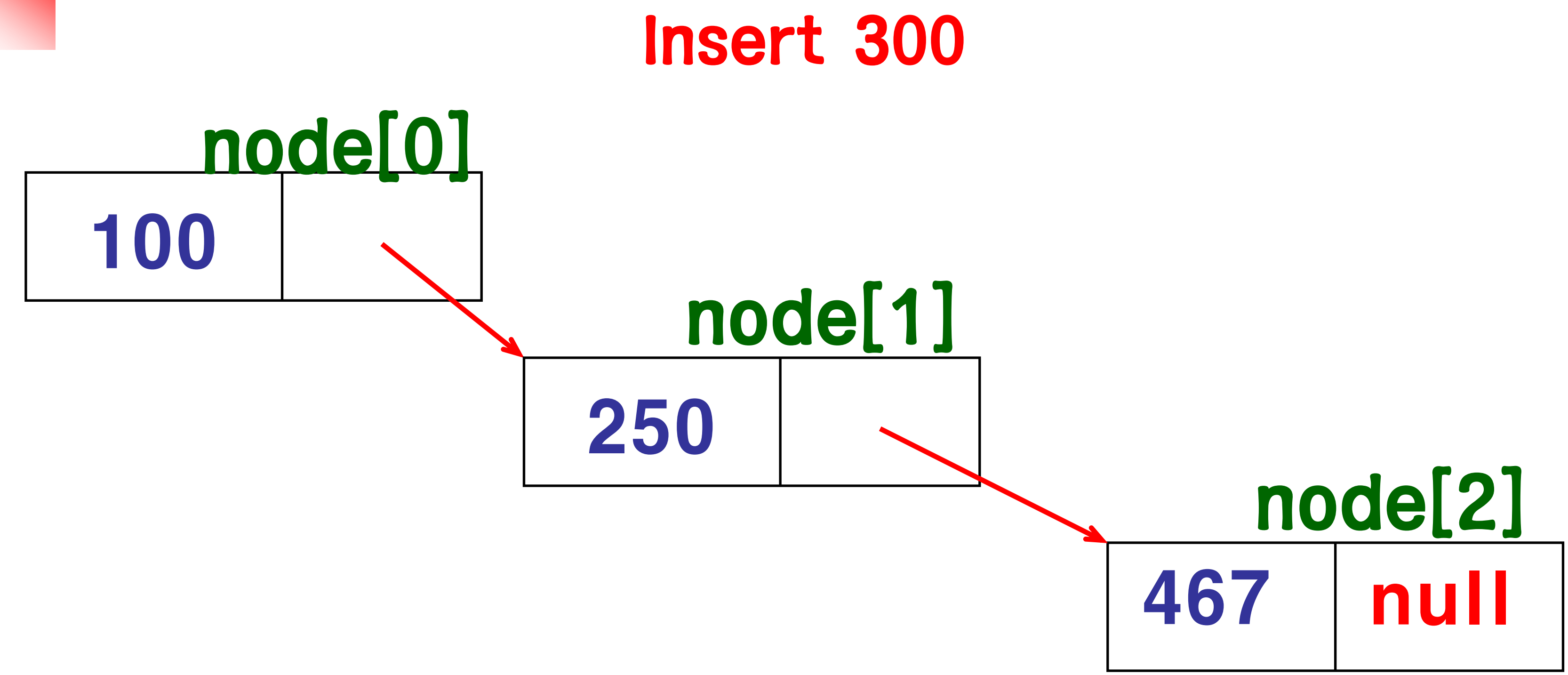


```
pNode = (struct NODE *) malloc (100 * sizeof(struct NODE));
```

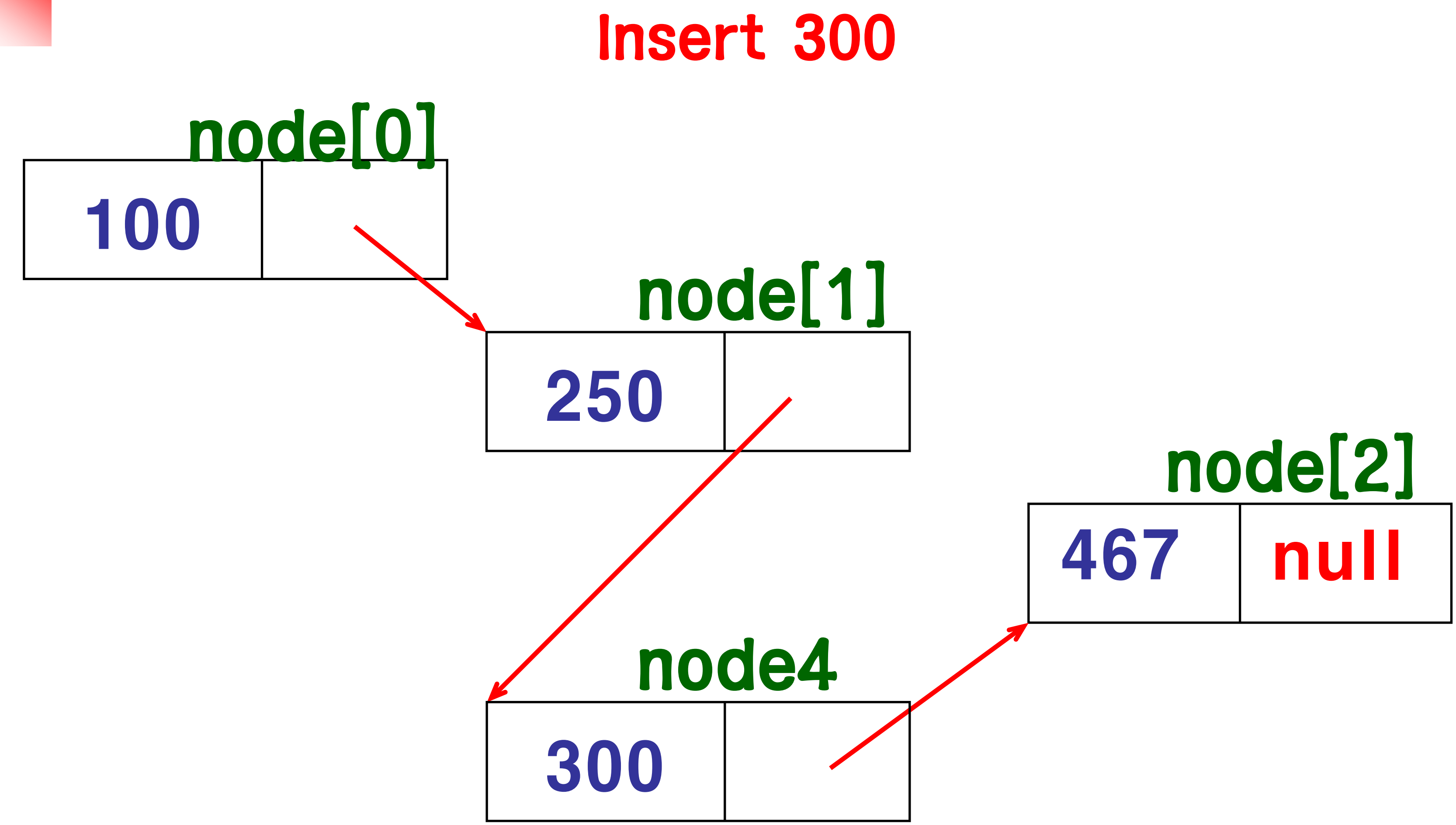
Dynamic Memory Deallocation: free( )

```
/* releases memory obtained by malloc  */
/* just pass the address of the memory block */
/* no need to specify the size of the memory block */

free (cp);
free (pNode);
```

Inserting a Data Node (While Maintaining an Order)

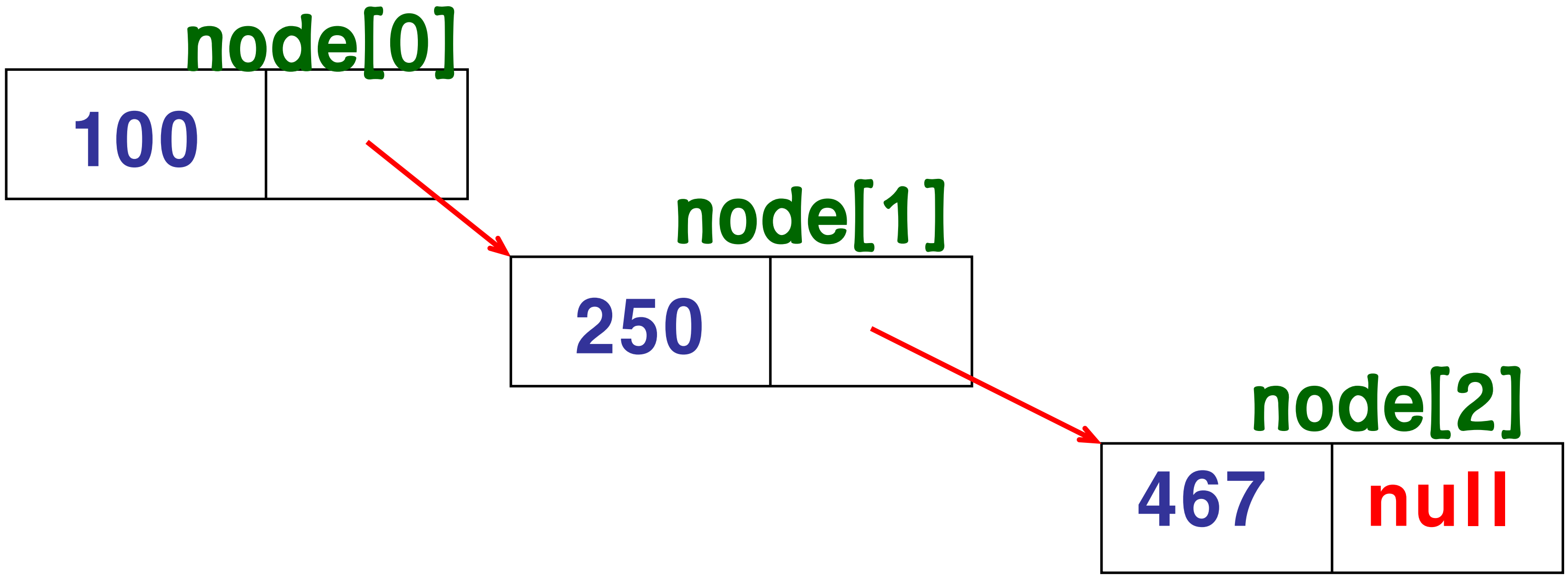


Inserting a Data Node (While Maintaining an Order): Result (how to get this? Later)



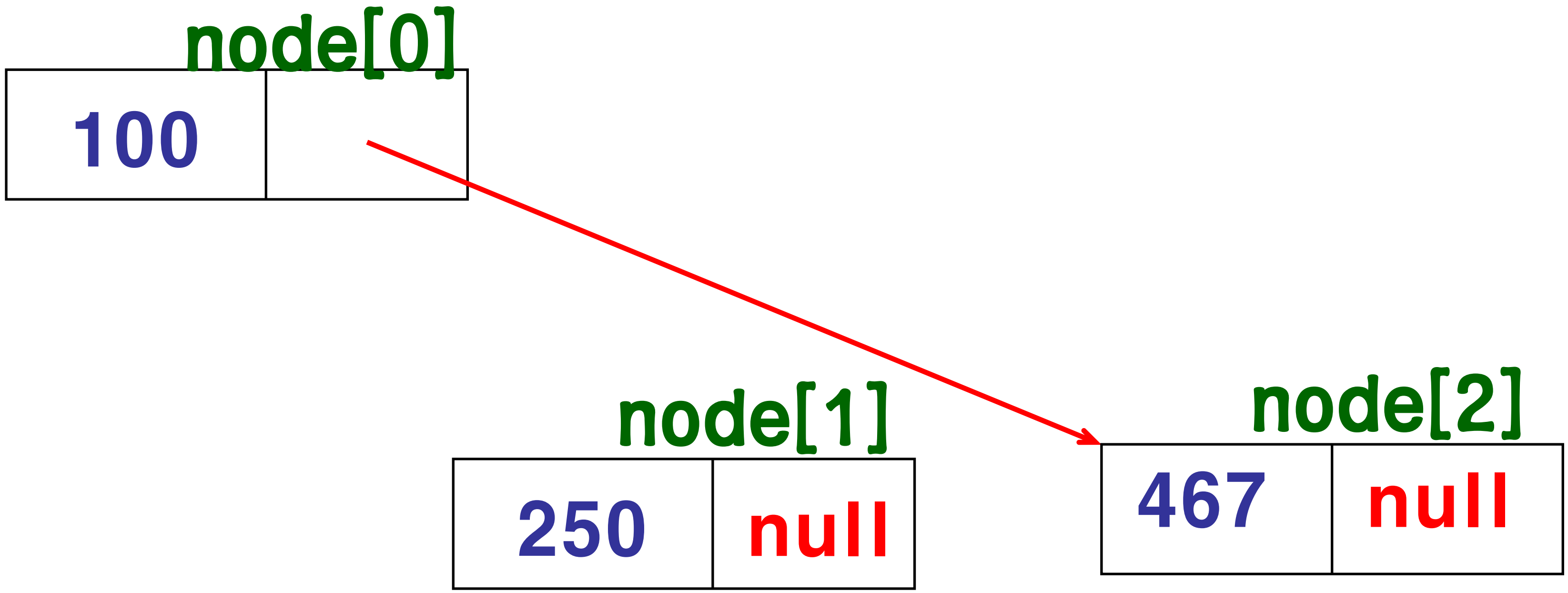
Deleting a Data Node (While Maintaining an Order)

Delete 250

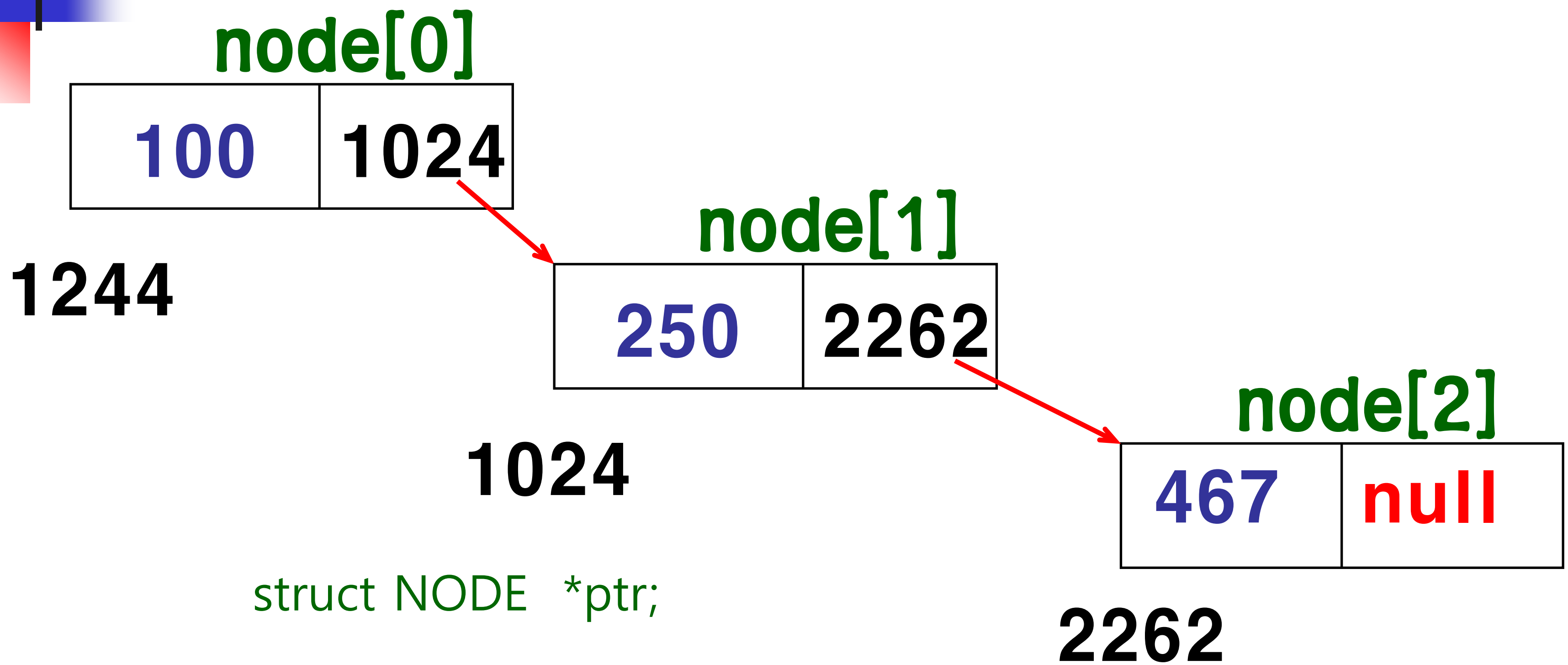


Deleting a Data Node (While Maintaining an Order): Result (how to get this? Later)

Delete 250



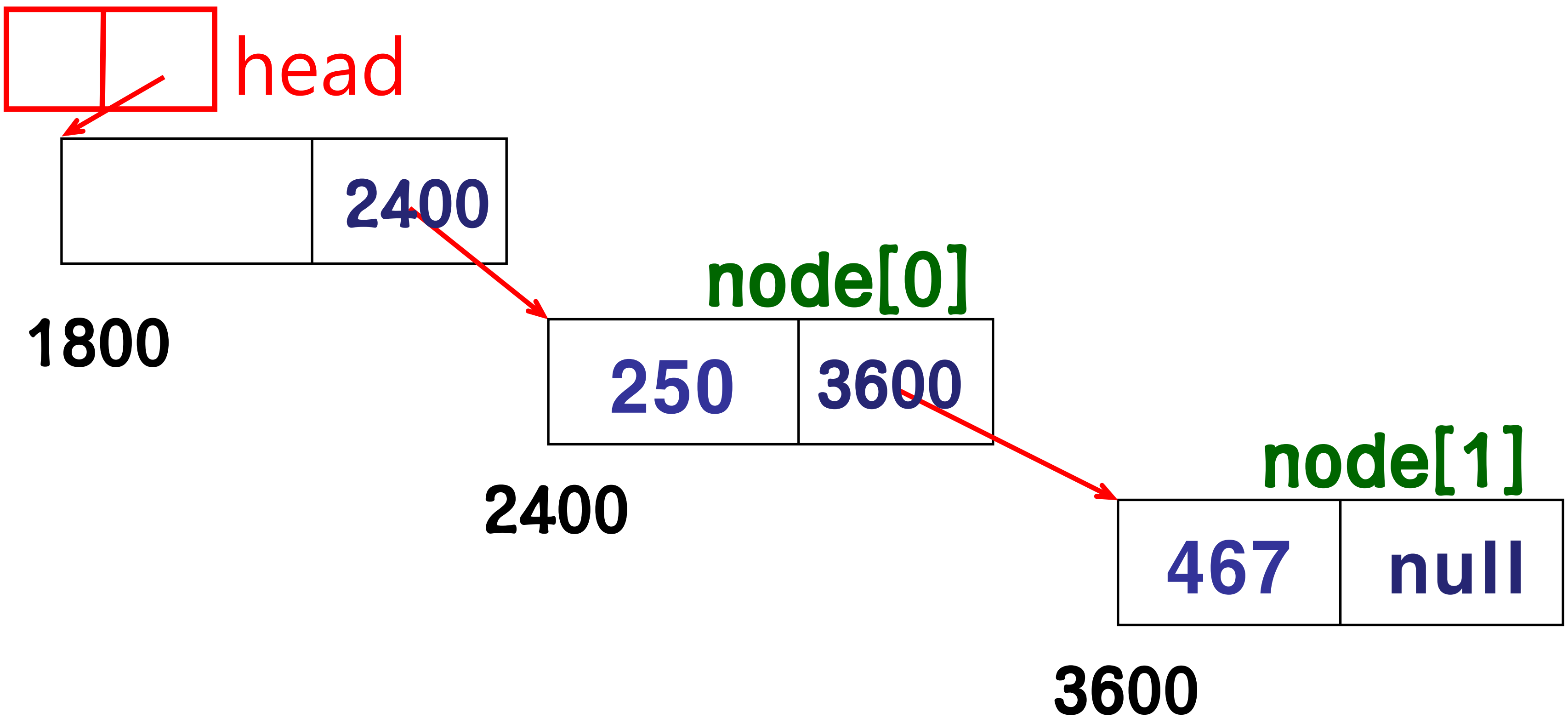
Searching for the Last Node (on a Linked List)




```
struct NODE   *ptr;

ptr = &node[0];

while (ptr != NULL)
      ptr = ptr->next;
```

Searching a Linked List: A Better Way

- Create a variable head (type struct *) to store the address of the first node of the linked list.
- head may also store information, such as the total number of nodes and the address of the last node.



Note

- The information in head must be updated, if
  - the current first node or last node is deleted, or
  - a new node is inserted as the new first node, or
  - the number of nodes on the linked list changes (due to insertion and deletion of nodes).

- Creating a head
  - Create a struct of struct node * type
  - The next pointer in head is set to NULL

Using **head** to Insert a New Node

```
void Insert( struct NODE *head, int value )
{
/* Start from head->next instead of head */
struct NODE *p = head->next, *prev = head;
struct NODE* new_node;

    while (p) {
        if ( p->key > value ) break;
        prev = p;
        p = p->next;
    }
    new_node = (struct NODE*)malloc(sizeof(struct NODE));
    new_node->key = value;
    prev->next = new_node;  /* adjust next pointers  */
    new_node->next = p;
}
```

Find the node to come after the new node

Get address of the node before the new node

Create a new node to insert

Set next pointers in before and new nodes

Using head to Delete a Node

```
Void Delete( struct NODE *head, int value )
{
  struct NODE *p = head->next,  *prev = head;

    while (p) {
         if ( p->key == value ) break;
        prev = p;
        p = p->next;
    }

    if (p) {
       prev->next = p->next;   /* node deleted   */
       free( p );     /* free memory */
    }
}
```

Find the node to delete

Mark the node before the node to delete

Adjust the next pointer in the before node

Free the memory used for the deleted node

Linked List for Stack & Queue

- Stack
  - Last-in, First-out
- Queue
  - First-in, First-out

- The elements in the list does not need to be ordered
- Linked list operations
  - Stack
    - Insert() to back
    - Delete() from back
  - Queue
    - Insert () to back
    - Delete () from front

Linked List for Stack & Queue

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

struct NODE
{
int  key;
struct NODE* next = NULL;
};

void PrintList(struct NODE* hea
d)
{
struct NODE* p = head->next;

int ind = 0;
while (p)
{
printf("node[%d] key: %d\n", in
d, p->key);
p = p->next;
ind++;
}
}
```

```c
void ClearMemory(struct NODE* h
ead)
{
struct NODE* p = head->next;
struct NODE* tmp;

int ind = 0;
while (p)
{
tmp = p;
p = p->next;
printf("node[%d] deleted..\n",
ind);
free(tmp);
ind++;
}
}
```

Linked List for Stack & Queue

```
void Insert_node_back(
struct NODE* head,
struct NODE* tail,
const int newkey)
{
struct NODE* current_node = head;
struct NODE* next_node = head->next;
struct NODE* new_node;

while (next_node)
{
current_node = next_node;
next_node = next_node->next;
}

// create a new node
new_node = (struct NODE*)malloc(sizeof(struc
t NODE));
new_node->key = newkey;
new_node->next = NULL;

// adjust next pointer
if (current_node == head) // empty now?
{
head->next = new_node;
}
else
{
current_node->next = new_node;
}
tail = new_node;

head->key += 1; // num elements sotred here
}
```

Linked List for Stack & Queue

```
int Delete_node_back(
struct NODE* head,
struct NODE* tail)
{
struct NODE* prev_node = head;
struct NODE* current_node = head;
struct NODE* next_node = head->next;

while (next_node)
{
prev_node = current_node;
current_node = next_node;
next_node = next_node->next;
}

if (current_node == head) // empty now?
{
printf("[Linked List Underflow..!!]\n");
return -1;
}

prev_node->next = NULL;
tail = prev_node;

int key = current_node->key;
free(current_node);
head->key -= 1; // num elements sotred he
re

return key;
}
```

Linked List for Stack & Queue

```
int Delete_node_front(
struct NODE* head,
struct NODE* tail)
{
struct NODE* current_node = head->nex
t;
if (current_node == NULL) // empty no
w?
{
printf("[Linked List Underflow..!!]\n
");
return -1;
}

struct NODE* next_node = current_node
->next;

head->next = next_node;
int key = current_node->key;
free(current_node);
head->key -= 1; // num elements sotre
d here

return key;
}
```

Linked List for Stack & Queue

```
int main()
{
struct NODE* head = (struct NODE*)mal
loc(sizeof(struct NODE));
head->key = 0;
head->next = NULL;

struct NODE* tail = (struct NODE*)mal
loc(sizeof(struct NODE));
tail->key = 0;
tail->next = NULL;

for (int i = 0; i < 5; i++)
{
printf("//---Inserting Key : %d \n",
i);
Insert_node_back(head, tail, i);
PrintList(head);
}

for (int i = 0; i < 6; i++)
{
int key = Delete_node_front(head, tai
l);
printf("//---First Key : %d \n", key)
;
PrintList(head);
}
```

```
for (int i = 0; i < 5; i++)
{
printf("//---Inserting Key : %d \n",
i);
Insert_node_back(head, tail, i);
PrintList(head);
}

for (int i = 0; i < 6; i++)
{
int key = Delete_node_back(head, tail
);
printf("//---Last Key : %d \n", key);
PrintList(head);
}


printf("\nDeleting all nodes in the l
inked list..\n");
ClearMemory(head);
free(head);
free(tail);
return 0;
}
```

Assignment 2

HW2-P1

- Using the method 1, create a ring buffer with an array of N=4 elements, and do the same sequence of inserts and deletes shown in pages 8-9.
- Draw your results in the ring buffer.
  - Use a scratch paper and take a photo.

HW2-P2

- Using the method 1, implement a ring buffer with an array of 5 elements.
- Test the program using the sequence of inserts and deletes shown in pages 8-9.

HW2-P3

- Implement and Test a Stack Program, Using a Singly Linked List.
  - for the same 4 functions (of Lab 1-1).
  - Does it need stack_full test??

HW2-P4

- Implement and Test a Queue Program, Using a Singly Linked List.
  - for the same 4 functions (of Lab 1-2).
  - Does it need queue_full test?

Submit to the CyberCampus

- # Assignment 2
  - Submit a single pdf file containing all the source codes, the compilation results, and the result screen captures

End of Class