Data Structure and Algorithm

Class 6

Seongjin Lee

Updated: 2017-03-06
DSA_2017_06

insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

1. Singly Linked Lists

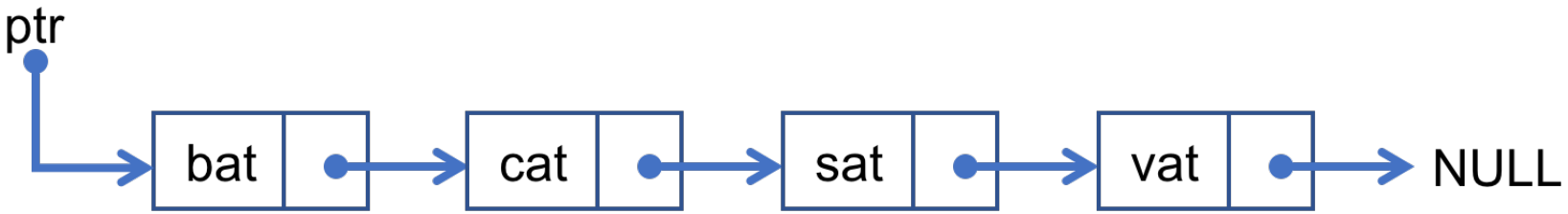
2. Dynamically Linked Stacks and Queues

3. Doubly Linked Lists

Singly Linked Lists

Singly Linked Lists

- compose of data part and link part
- link part contains address of the next element in a list
- non-sequential representations
- size of the list is not predefined
- dynamic storage allocation and deallocation

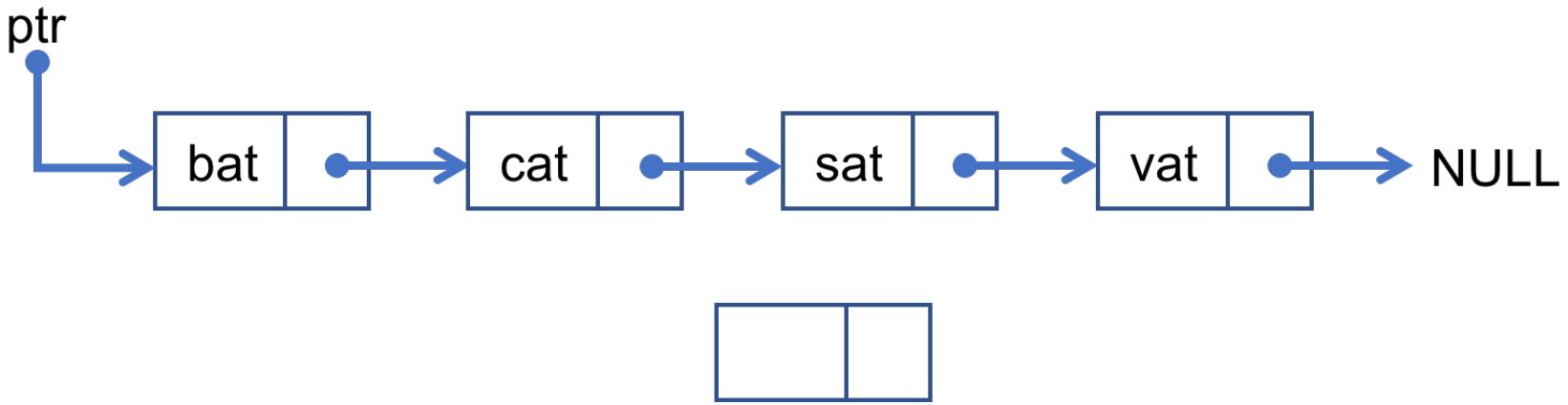


Singly Linked Lists: Insert

To insert the word `mat` between `cat` and `sat`

Singly Linked Lists: Insert

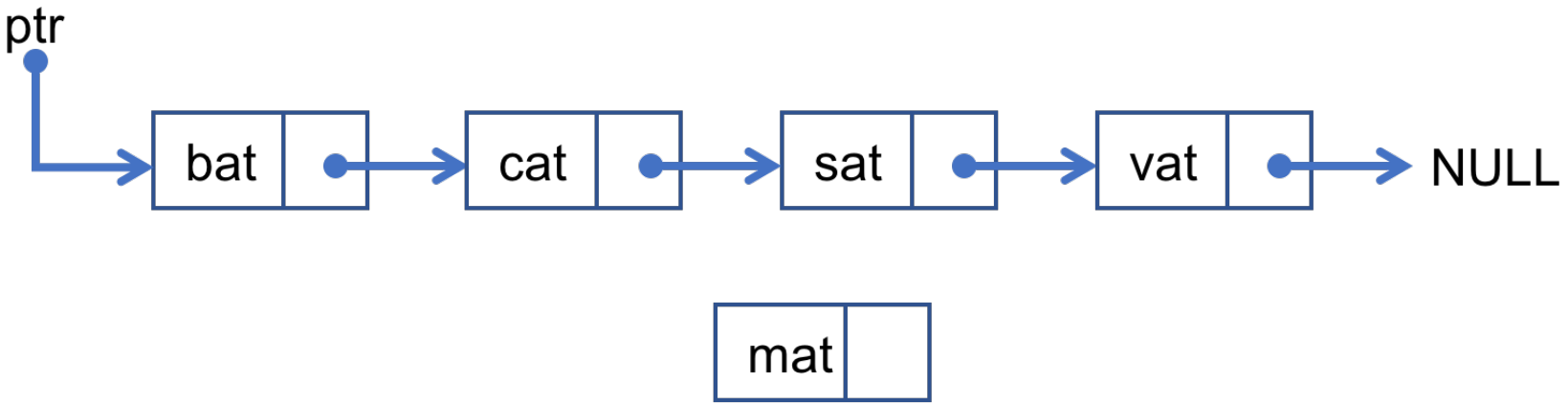
To insert the word `mat` between `cat` and `sat`



1. get a node currently unused (paddr)

Singly Linked Lists: Insert

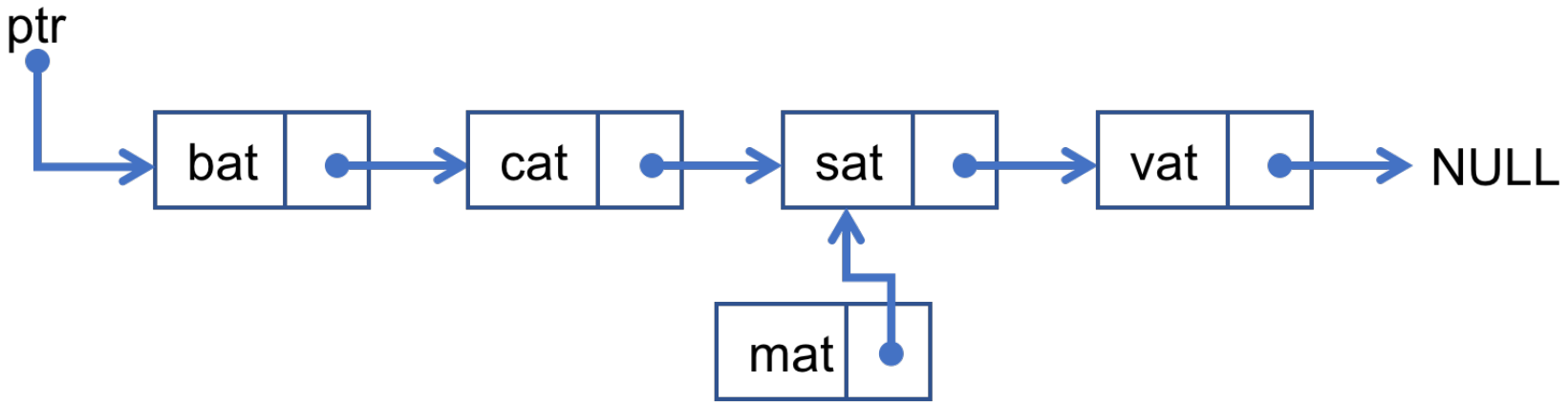
To insert the word `mat` between `cat` and `sat`



1. get a node currently unused (`paddr`)
2. set the data of this node to mat

Singly Linked Lists: Insert

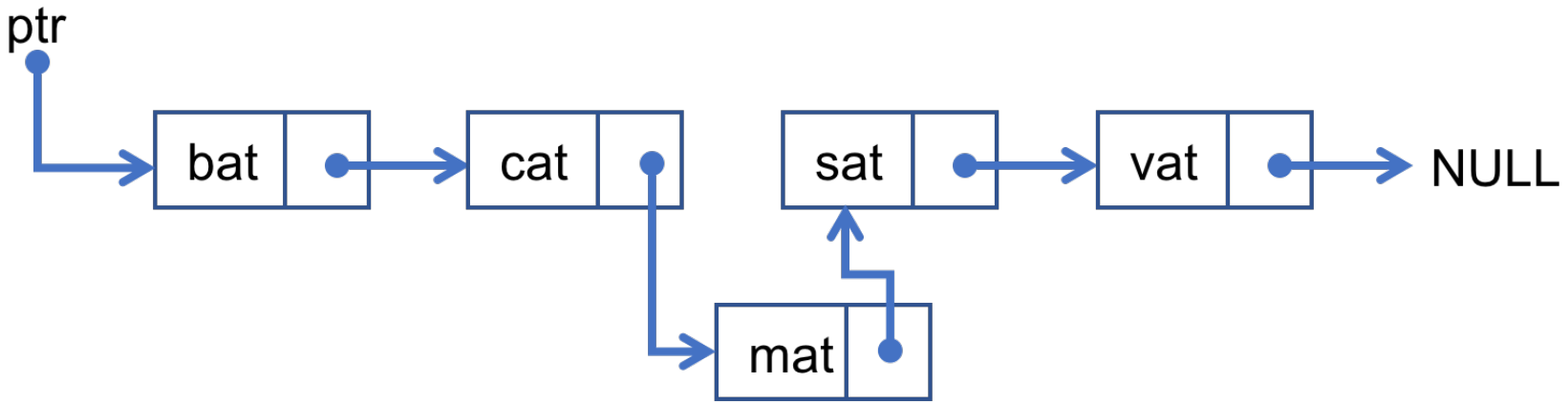
To insert the word `mat` between `cat` and `sat`



1. get a node currently unused (`paddr`)
2. set the data of this node to mat
3. set `paddr`'s link to point to the address found in the link of the node `cat`

Singly Linked Lists: Insert

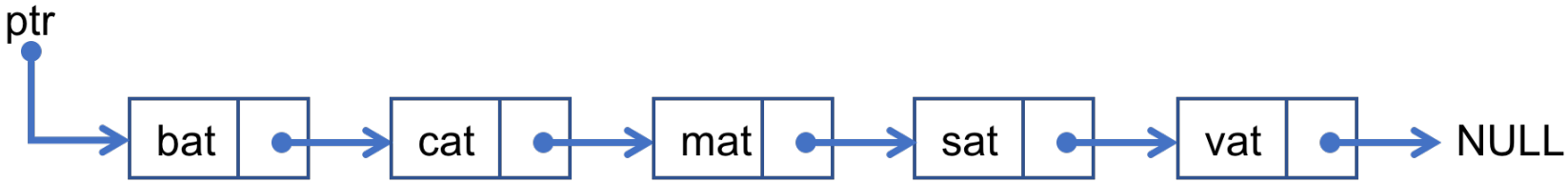
To insert the word `mat` between `cat` and `sat`



1. get a node currently unused (`paddr`)
2. set the data of this node to mat
3. set `paddr`'s link to point to the address found in the link of the node `cat`
4. set the link of the node cat to point to `paddr`

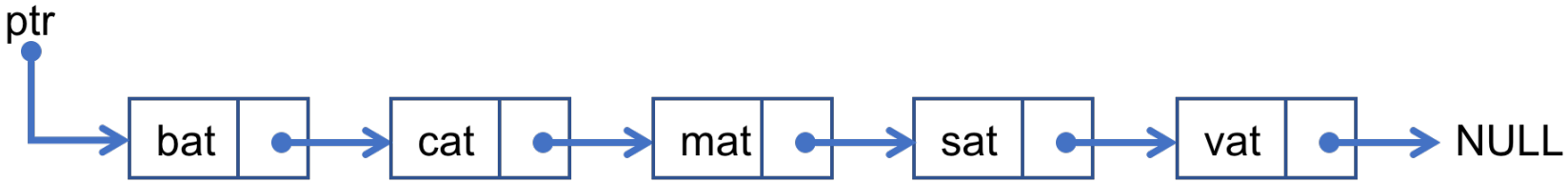
Singly Linked Lists: Delete

To delete `mat` from the list



Singly Linked Lists: Delete

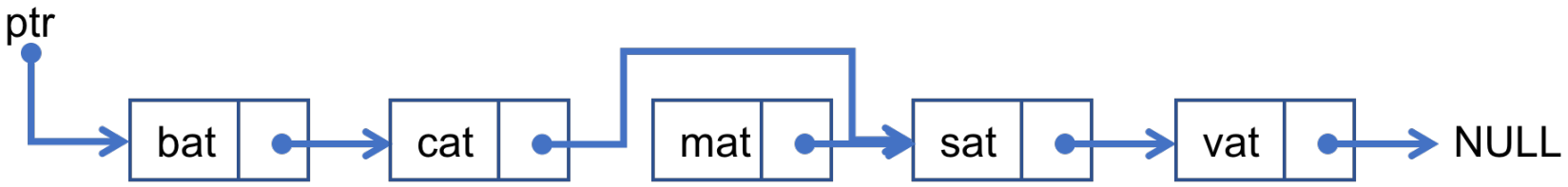
To delete `mat` from the list



1. find the element that immediately precedes `mat`, which is cat

Singly Linked Lists: Delete

To delete `mat` from the list



1. find the element that immediately precedes `mat`, which is cat
2. set its link to point to `mat`'s link

Singly Linked Lists: Delete

To delete `mat` from the list

1. find the element that immediately precedes `mat`, which is cat
2. set its link to point to `mat`'s link
3. free the `mat`'s node

Singly Linked List

Note

There is no data movement in insert and delete operation

Singly Linked List: Features

required capabilities to make linked representations possible

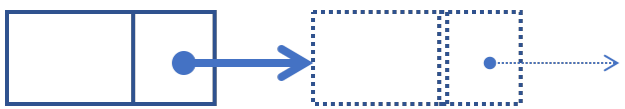
- a mechanism for defining a node's structure, that is, the field it contains
- a way to create new nodes when we need them
- a way to remove nodes that we no longer need

Singly Linked List: The data type

Define the node structure for the list

- **data field**: Character array
- **link field**: Pointer to the next node
  - Self-referential structure

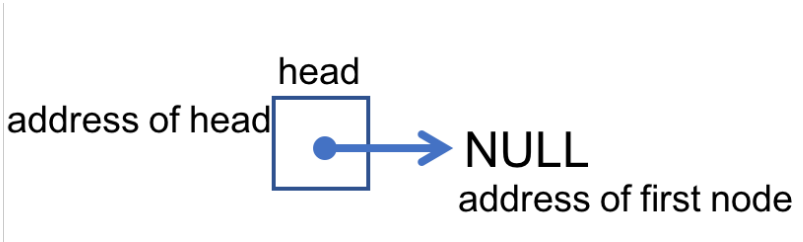
```
1  typedef struct node {
2      char data[4];
3      struct node *next; // self-referential structure
4  } node_t;
```



Singly Linked List: Creating a node

Create new nodes for our list then place the word bat into the list

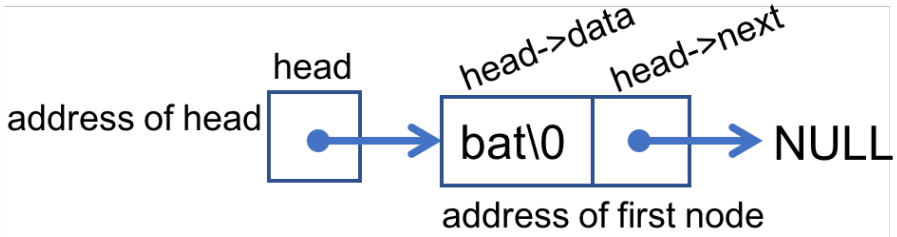
```
1  node_t *head = NULL;
2  head = malloc(sizeof(node_t));
3  if (head == NULL) {
4      return 1;
5  }
6  strcpy(head->data, "bat");
7  head->next = NULL;
```



Singly Linked List: Creating a node

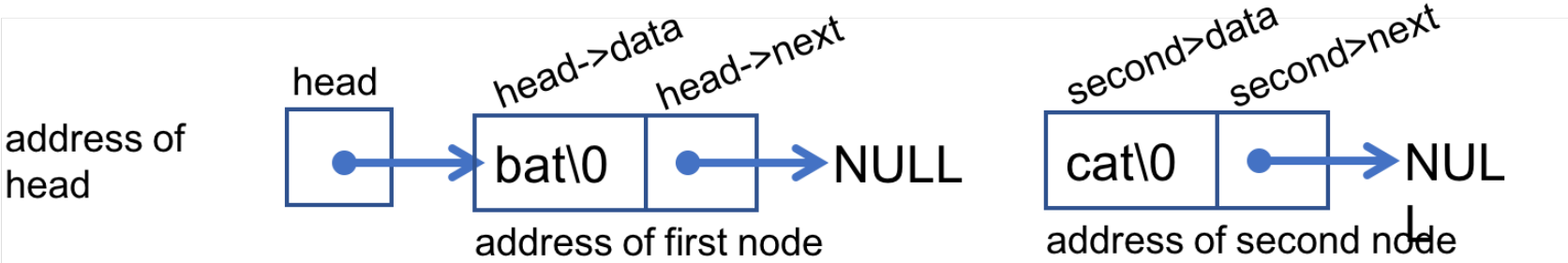
Create new nodes for our list then place the word bat into the list

```
1  node_t *head = NULL;
2  head = malloc(sizeof(node_t));
3  if (head == NULL) {
4      return 1;
5  }
6  strcpy(head->data, "bat");
7  head->next = NULL;
```



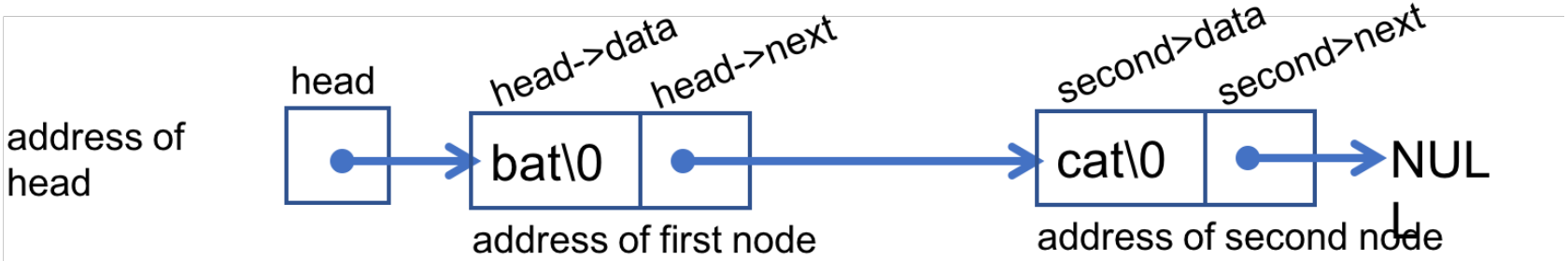
Singly Linked List: Adding a node

```
1  node_t second;
2
3  second = malloc(sizeof(node_t));
4
5  second->next=NULL;
6  strcpy(second->data, "cat");
7
8  head->next=second;
```



Singly Linked List: Adding a node

```
1  node_t second;
2
3  second = malloc(sizeof(node_t));
4
5  second->next=NULL;
6  strcpy(second->data, "cat");
7
8  head->next=second;
```



Singly Linked List: Implementation

- data structure
- insert a node
- delete the list
- remove a node
- search
- print

```
1  typedef struct node {
2      int data;
3      struct node *next;
4  } node_t;
```

Singly Linked List: Print

```
1  void print_list(node_t * head) {
2      node_t * current = head;
3
4      while (current != NULL) {
5          printf("%d\n", current->data);
6          current = current->next;
7      }
8  }
```

- Create a pointer that iterates the list
- print the data in the list.
- Repeat until the pointer reaches the end of the list (the next node in NULL)

Singly Linked List: Insert at the end

```
1  void insert_tail(node_t * head, int data) {
2      node_t * current = head;
3      while (current->next != NULL) {
4          current = current->next;
5      }
6
7      /* now we can add a new variable */
8      current->next = malloc(sizeof(node_t));
9      current->next->data = data;
10     current->next->next = NULL;
11 }
```

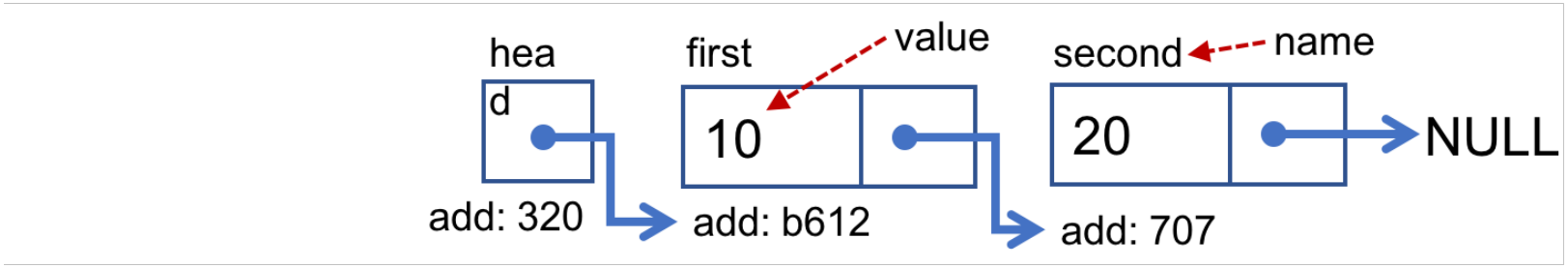
- Iterate the list using a pointer to find the end of the list
- add a new node at the end of the list
- add the data
- mark next node as `NULL`

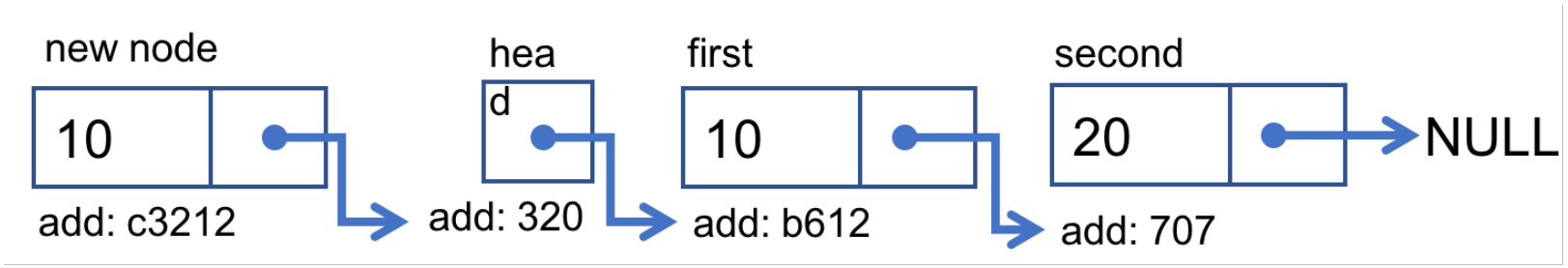
Singly Linked List: Insert at the beginning

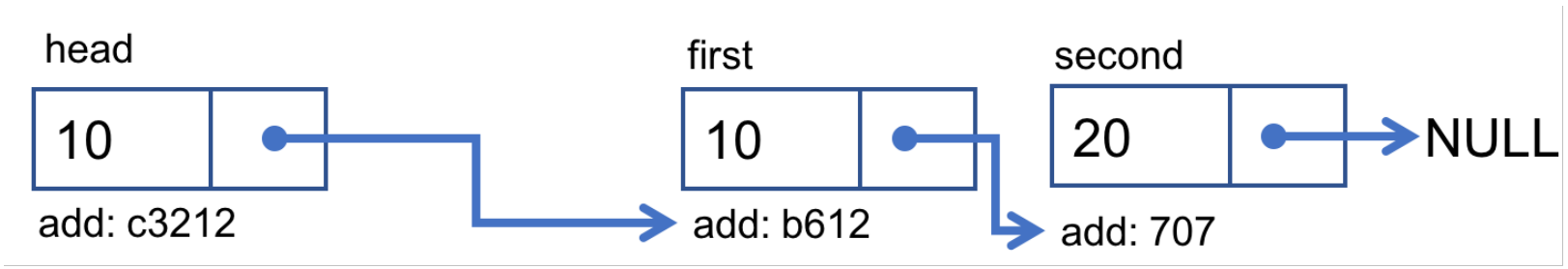
```
1  void insert_head(node_t ** head, int data) {
2      node_t * new_node;
3      new_node = malloc(sizeof(node_t));
4
5      new_node->data = data;
6      new_node->next = *head;
7      *head = new_node;
8  }
```

- create a new node with data
- link the new node to point to the head of the list
- set the head of the list with the new node

Singly Linked List: Insert at the beginning







Singly Linked List: Removing the first node

```
1  int remove_head(node_t ** head) {
2      int retval = -1;
3      node_t * new_head = NULL;
4      if (*head == NULL) {
5          return -1;
6      }
7      new_head = (*head)->next;
8      retval = (*head)->data;
9      free(*head);
10     *head = new_head;
11     return retval;
12 }
```

- select the next item that head points
- save it as a new head
- free the head
- use the new head as the head of the list

Singly Linked List: Removing the last node

```
1  int remove_tail(node_t * head) {
2      int retval = 0;
3      if (head->next == NULL) { // only one node in the list
4          retval = head->data;
5          free(head); // remove the head
6          return retval;
7      }
8
9      node_t * current = head;
10     while (current->next->next != NULL) { // go to the second last
           node
11         current = current->next;
12     }
13
14     retval = current->next->data; // get the last node's data
15     free(current->next); // free the last node
16     current->next = NULL; // set the next node NULL
17     return retval;
18
19 }
```

Singly Linked List: Removing a value

```
1  int remove_value(node_t ** head, int data) {
2      node_t *previous, *current;
3      if (*head == NULL)
4          return -1;
5
6      if ((*head)->data == data)
7          return pop(head);
8
9      previous = current = (*head)->next;
10     while (current != NULL) {
11         if (current->data == data) {
12             previous->next = current->next;
13             free(current);
14             return data;
15         }
16
17         previous = current;
18         current = current->next;
19     }
20     return -1;
21 }
```

Singly Linked List: Deleting the list

```
1  void delete_list(node_t *head) {
2      node_t *current = head,
3             *next = head;
4
5      while (current) {
6          next = current->next;
7          free(current);
8          current = next;
9      }
10 }
```

Dynamically Linked Stacks and Queues

Representing Stack and Queue by Linked List

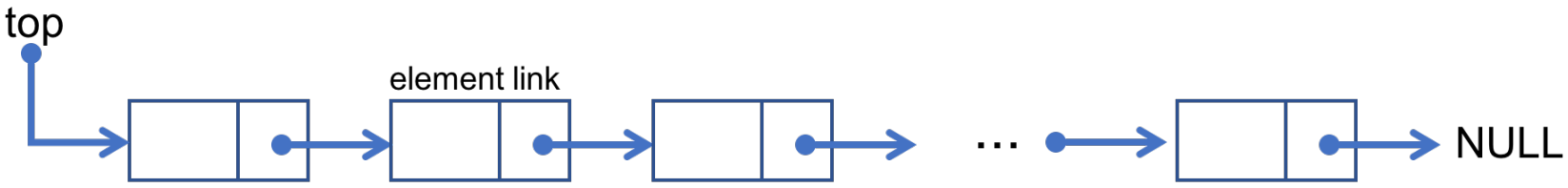


Figure: linked list stack

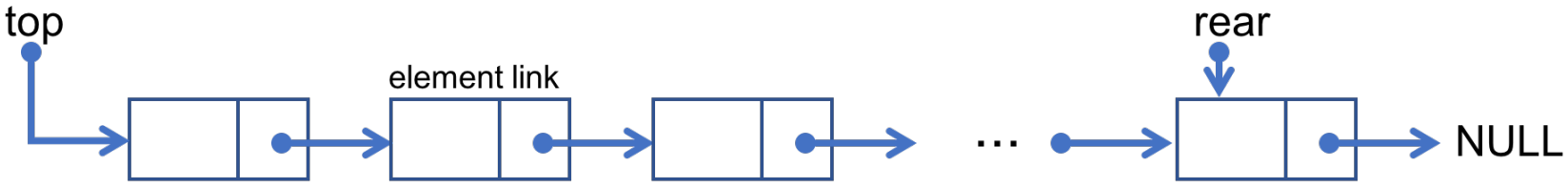


Figure: linked list queue

Representing stacks by linked list

```
1  /* n=MAX_STACKS=10 */
2  #define MAX_STACKS 10
3
4  typedef struct {
5      int key;
6      /* other fields */
7  } element;
8
9  typedef struct stack *stack_ptr;
10
11 typedef struct stack {
12     element item;
13     stack_ptr link;
14 };
15
16 stack_ptr top[MAX_STACKS];
```

- Initial condition for stacks
  - `top[i] = NULL`, `0 ≤ i ≤ MAX_STACKS`
- boundary condition for `n` stacks
  - empty condition
    - `top[i] = NULL` iff the i^{th} stack is empty
  - full condition
    - `IS_FULL(temp)` iff the memory is full

Pushing to a linked list stack

```
1  void push(stack_ptr *ptop, element item) {
2      stack_ptr temp = (stack_ptr)malloc(sizeof (stack));
3      if(IS_FULL(temp)) {
4          fprintf(stderr,‘‘The memory is full\n’’);
5          exit(1);
6      }
7      temp->item=item;
8      temp->link=*ptop;
9      *ptop = temp;
10 }
```

Poping a linked list stack

```
1  element pop(stack_ptr *ptop) {
2      stack_ptr temp = *ptop;
3      element item;
4      if(IS_EMPTY(temp)) {
5          fprintf(stderr,‘‘The stack is empty\n’’);
6          exit(1);
7       }
8       item=temp->item;
9       *ptop=temp->link;
10      free(temp);
11      return item;
12 }
```

Representing Queues by linked list

```
1  /* m=MAX_QUEUES=10 */
2  #define MAX_QUEUES 10
3
4  typedef struct queue *queue_ptr;
5
6  typedef struct queue {
7      element item;
8      queue_ptr link;
9  };
10
11 queue_ptr front[MAX_QUEUES],
12           rear[MAX_QUEUES];
```

- initial condition for queue
  - `front[i] = NULL`, $0 \leq i \leq$ `MAX_QUEUES`
- boundary condition for queues
  - empty condition
    - `front[i] = NULL` iff the i^{th} queue is empty
  - full condition
    - `IS_FULL(temp)` iff the memory is full

Add to the rear of a linked list queue

```
1  void insert(queue_ptr *pfront,
2              queue_ptr *prear,
3              element item) {
4      queue_ptr temp = (queue_ptr)malloc(sizeof(queue));
5
6      if(IS_FULL(temp)) {
7          fprintf(stderr, ‘‘The memory is full\n’’);
8          exit(1);
9      }
10
11     temp->item=item;
12     temp->link=NULL;
13
14     if (*pfront)
15         (*prear)->link=temp;
16     else
17         *pfront = temp;
18
19     *prear = temp;
20 }
```

Delete from the front of a linked list queue

```
1  element delete(queue_ptr *pfront) {
2      queue_ptr temp=*pfront;
3      element item;
4
5      if (IS_EMPTY(*pfront)) {
6          fprintf(stderr,‘‘The queue is empty\n’’);
7          exit(1);
8      }
9
10     item=temp->item;
11     *pfront=temp->link;
12     free(temp);
13
14     return item;
15 }
```

Representing Stack and Queues by Linked List

Advantages are

- no data movement is necessary: O(1) operation
- no full condition check is necessary
- size is growing and shrinking dynamically

Doubly Linked Lists

Doubly Linked Lists

Problems of singly linked lists

- move to only one way direction
- hard to find the previous node
- hard to delete the arbitrary node

doubly linked circular list

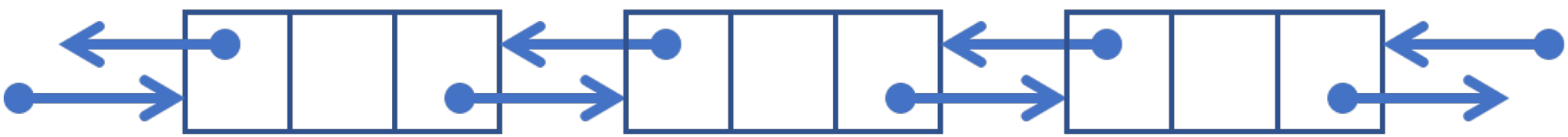
- doubly lists + circular lists
- allow two links
- two way direction

Doubly Linked Lists

```
1  typedef struct node *node_ptr;
2
3  typedef struct node {
4      node_ptr llink;
5      element item;
6      node_ptr rlink;
7  };
```

Doubly Linked Lists

```
1  typedef struct node *node_ptr;
2
3  typedef struct node {
4      node_ptr llink;
5      element item;
6      node_ptr rlink;
7  };
```



Doubly Linked Lists

suppose that `ptr` points to any node in a doubly linked list

```
1  ptr
2
3  ptr->llink->rlink
4
5  ptr->rlink->llink
```

Doubly Linked Lists

introduce dummy node, called, head

- to represent empty list
- make easy to implement operations
- contains no information in item field

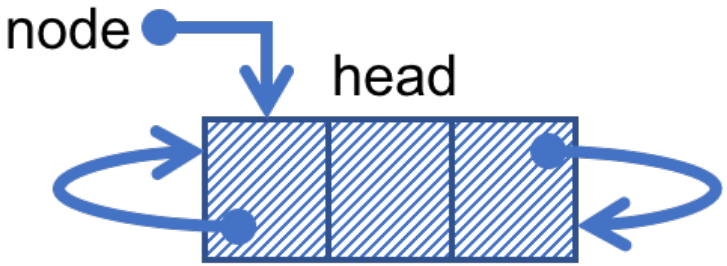


Figure: empty doubly linked circular list with head node

Doubly Linked Lists

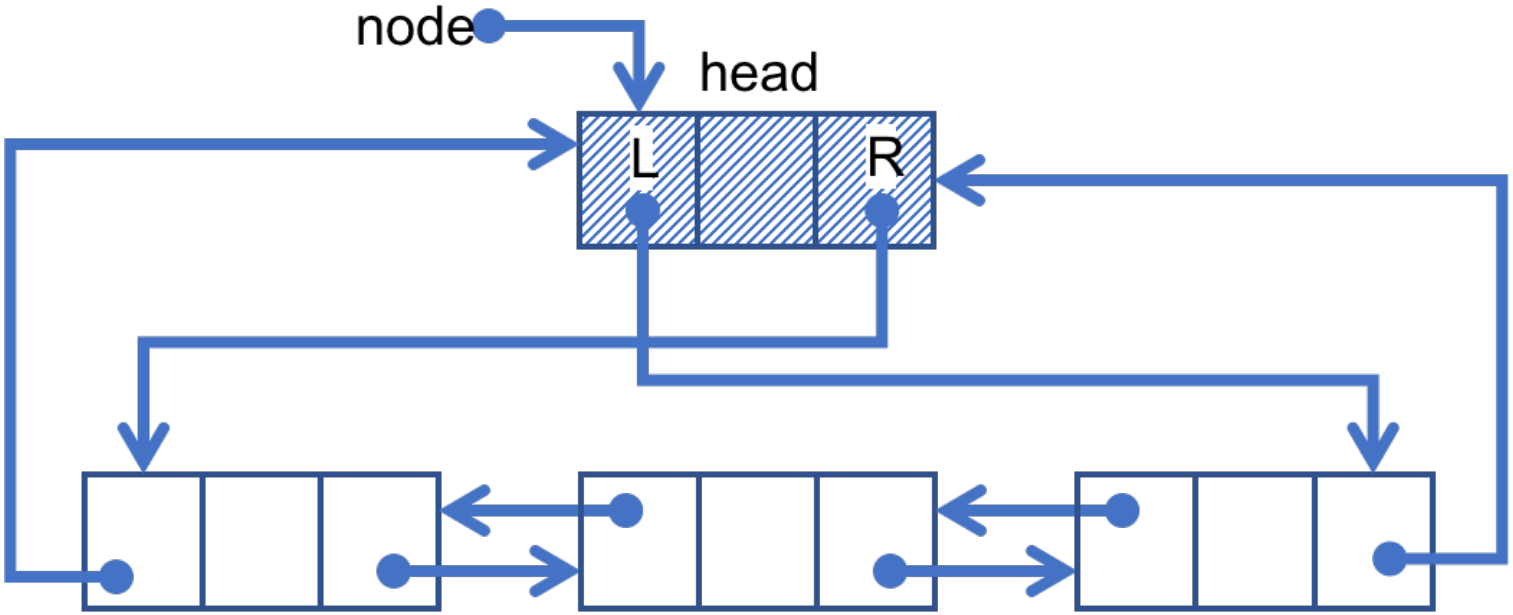


Figure: doubly linked circular list with head node

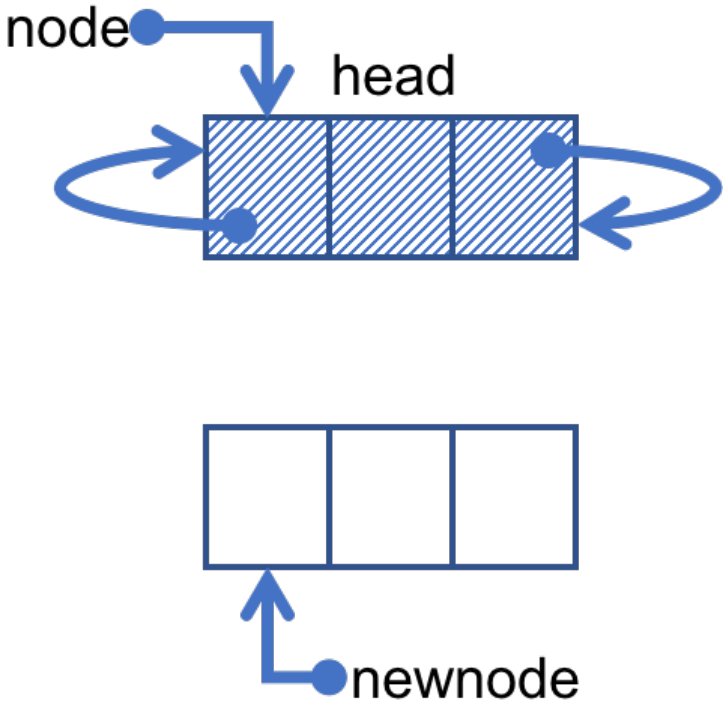
Insert into a doubly linked circular list

```
1  void d_insert(node_ptr node, node_ptr newnode) {
2      /* insert newnode to the right of node */
3      newnode->llink = node;
4      newnode->rlink = node->rlink;
5
6      node->rlink->llink = newnode;
7      node->rlink = newnode;
8  }
```

time complexity O(1)

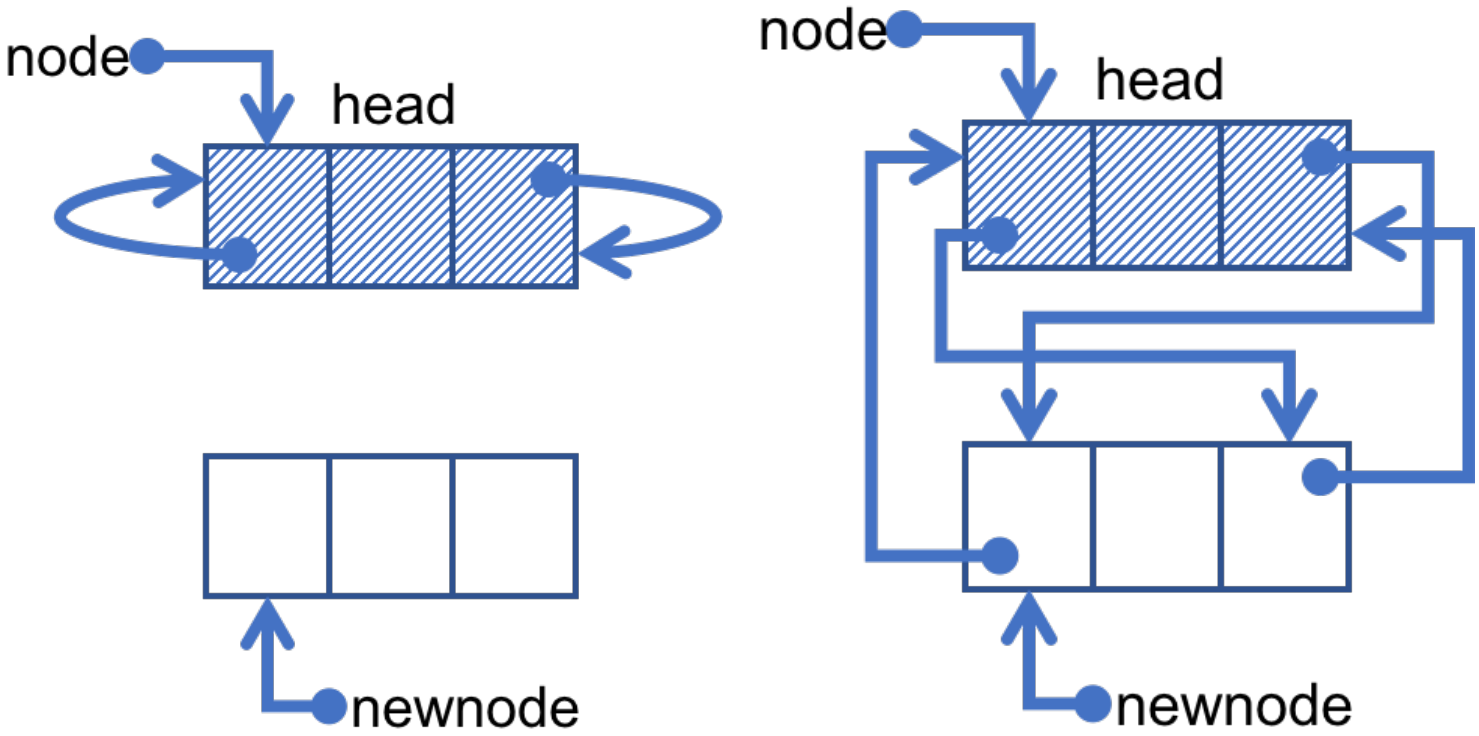
Insert into a doubly linked circular list

Insertion into an empty doubly linked circular list



Insert into a doubly linked circular list

Insertion into an empty doubly linked circular list



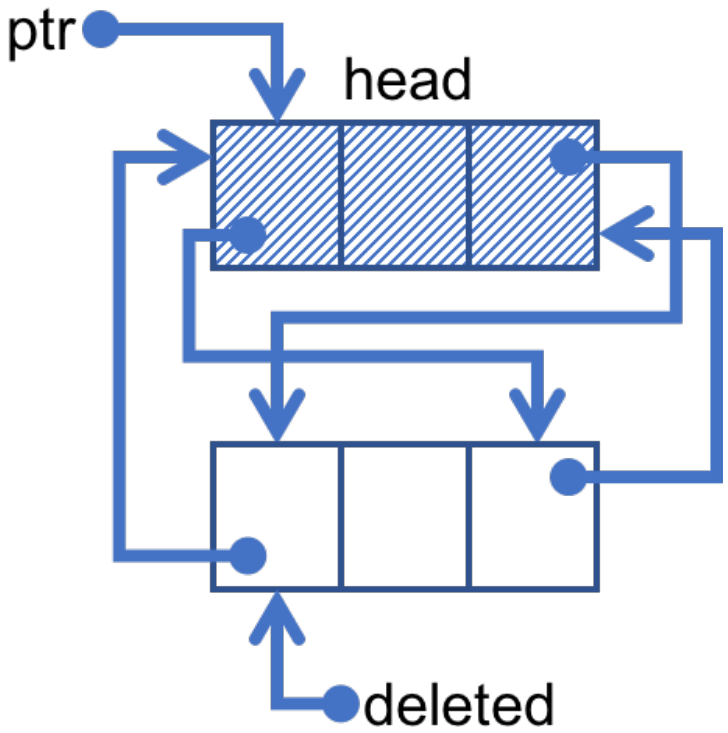
Deletion from a doubly linked circular list

```
1  void ddelete(node_ptr node, node_ptr deleted) {
2      /* delete from the doubly linked list */
3      if (node == deleted)
4          printf("Deletion of head node not permitted.\n");
5      else {
6          deleted->llink->rlink = deleted->rlink;
7          deleted->rlink->llink = deleted->llink;
8          free(deleted);
9      }
10 }
```

Time complexity: O(1)

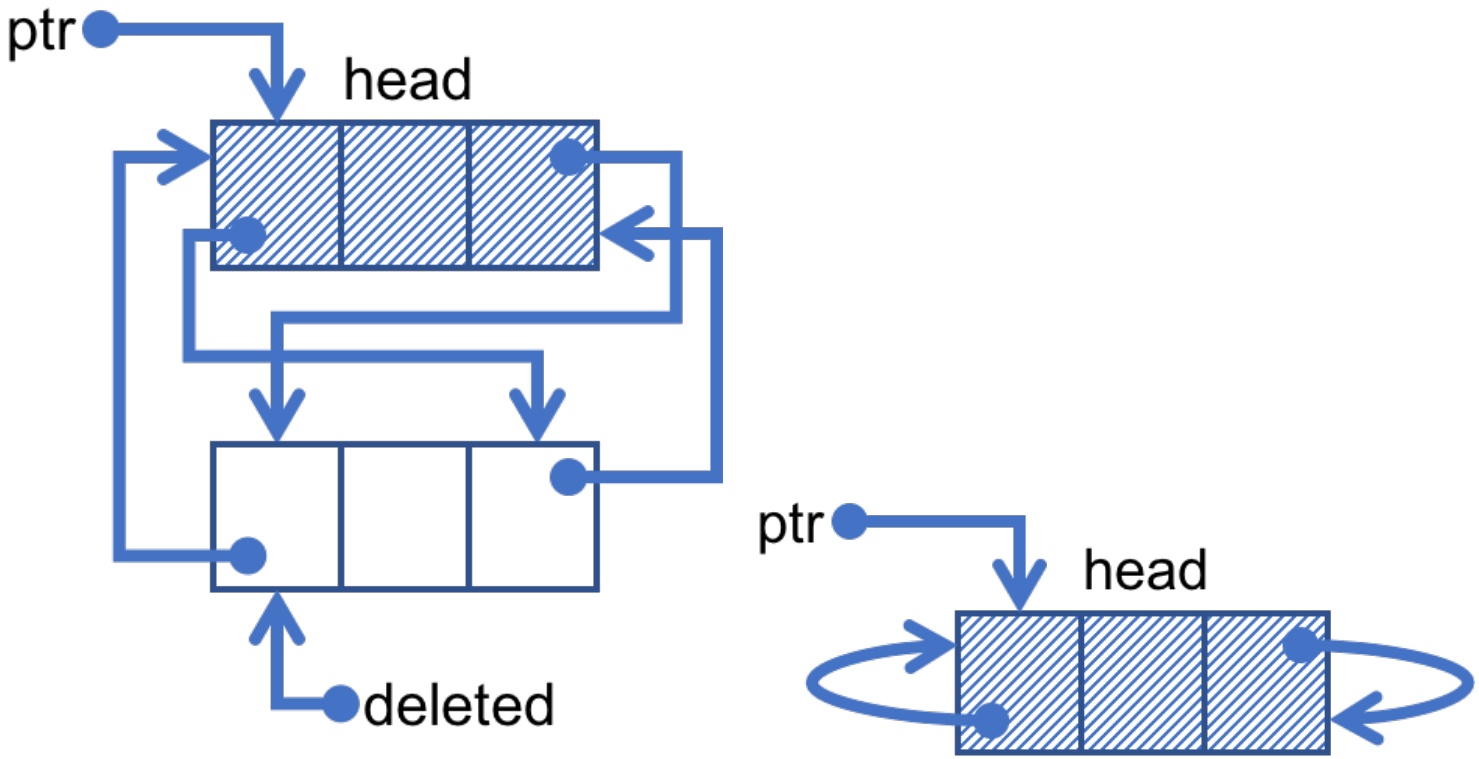
Deletion from a doubly linked circular list

Deletion in a doubly linked circular list



Deletion from a doubly linked circular list

Deletion in a doubly linked circular list



Notes on doubly linked circular list

- don't have to traverse a list - time complexity of O(1)
- insert(delete) front/middle/rear is all the same