Data Structure and Algorithm

Class 6

Seongjin Lee

DSA_06

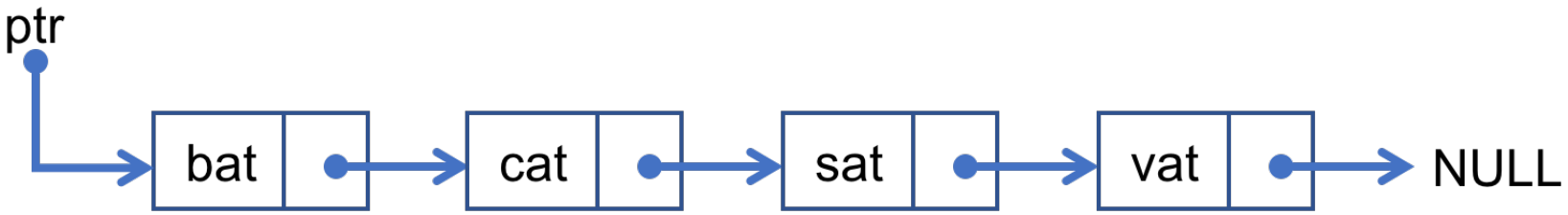
insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Singly Linked Lists

Singly Linked Lists

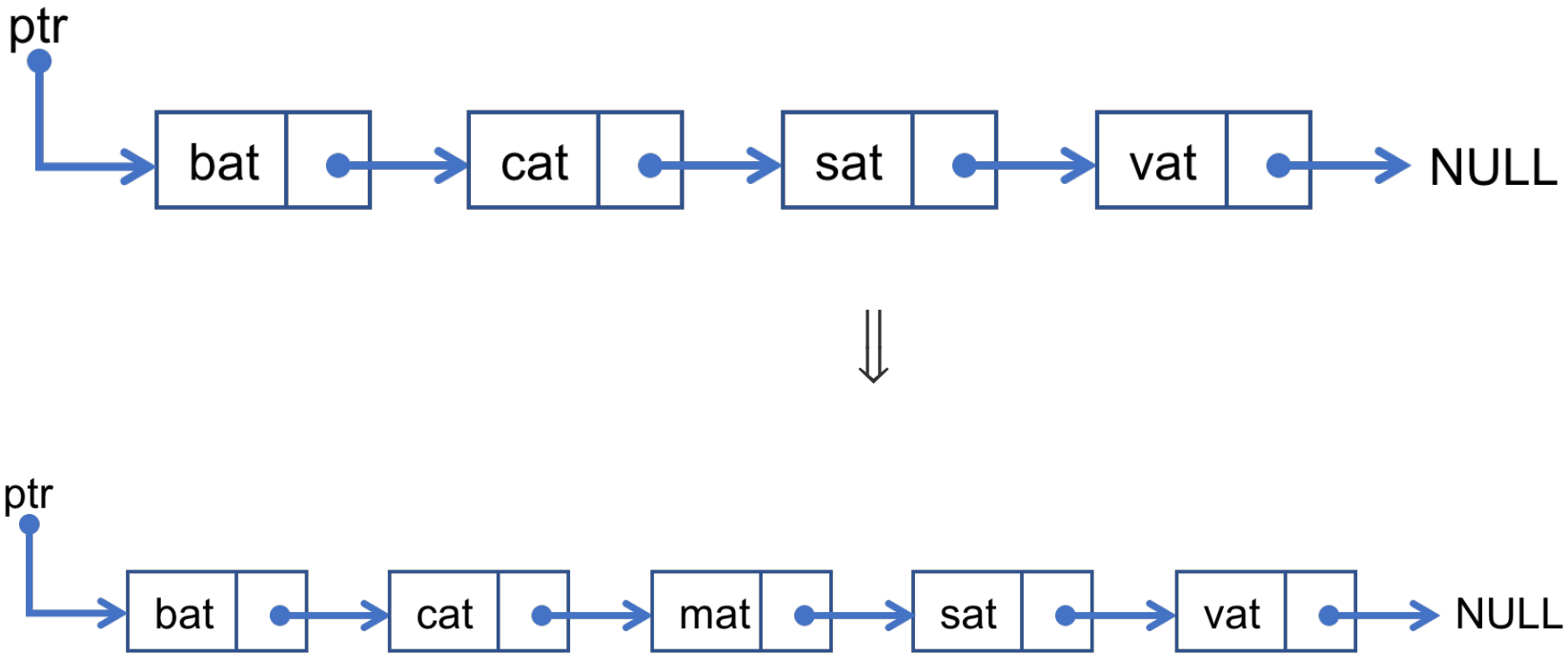
- Compose of data part and link part
- Link part contains address of the next element in a list
- Non-sequential representations
- Size of the list is not predefined
- Dynamic storage allocation and deallocation



Singly Linked Lists: Insert

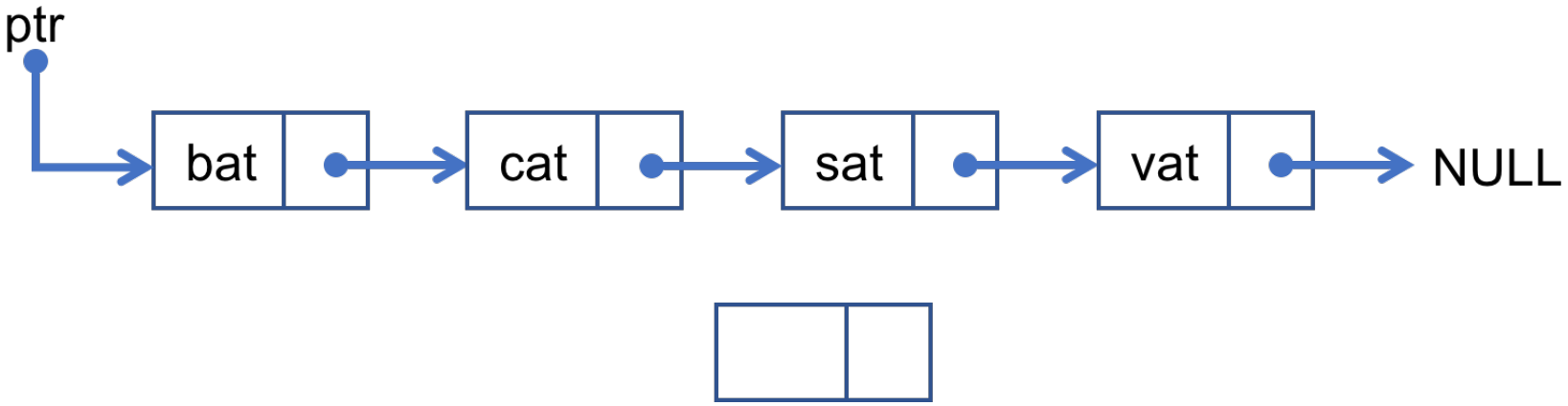
To insert the word `mat` between `cat` and `sat`

- Goal



Singly Linked Lists: Insert

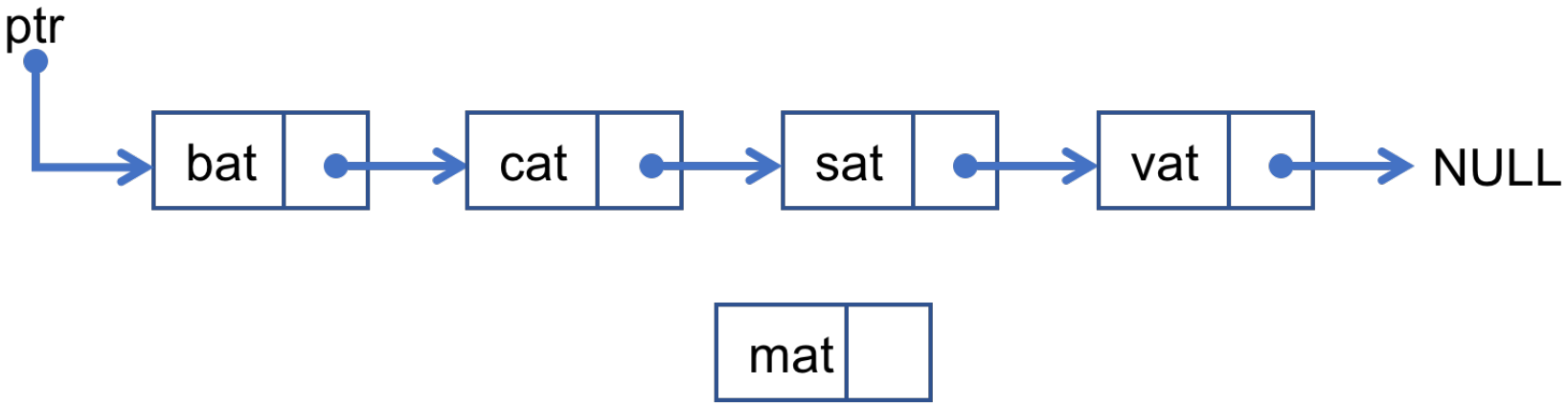
To insert the word `mat` between `cat` and `sat`



1. Get a node currently unused (`paddr`)

Singly Linked Lists: Insert

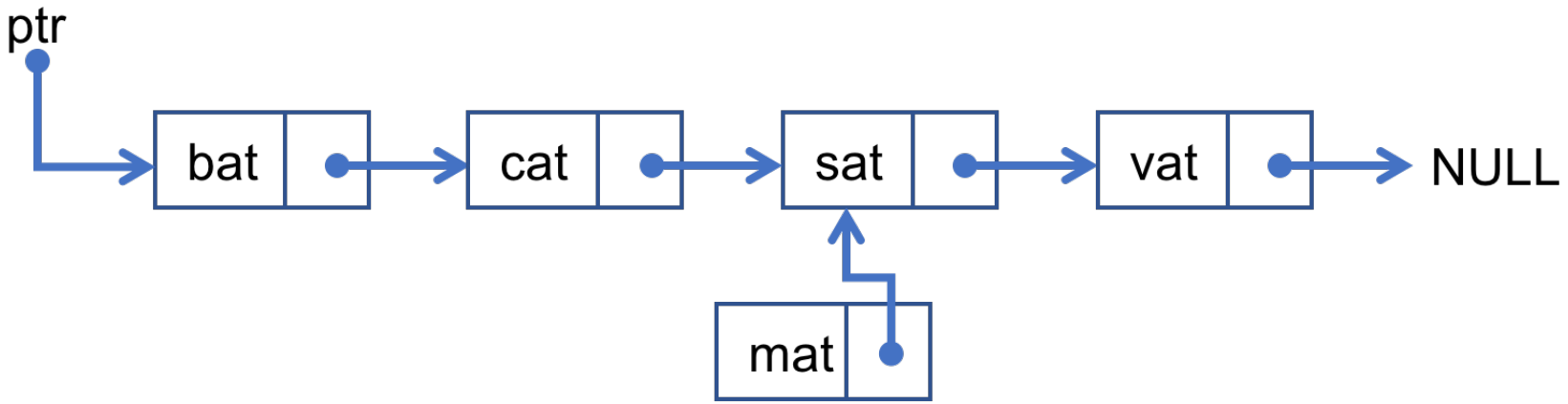
To insert the word `mat` between `cat` and `sat`



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

Singly Linked Lists: Insert

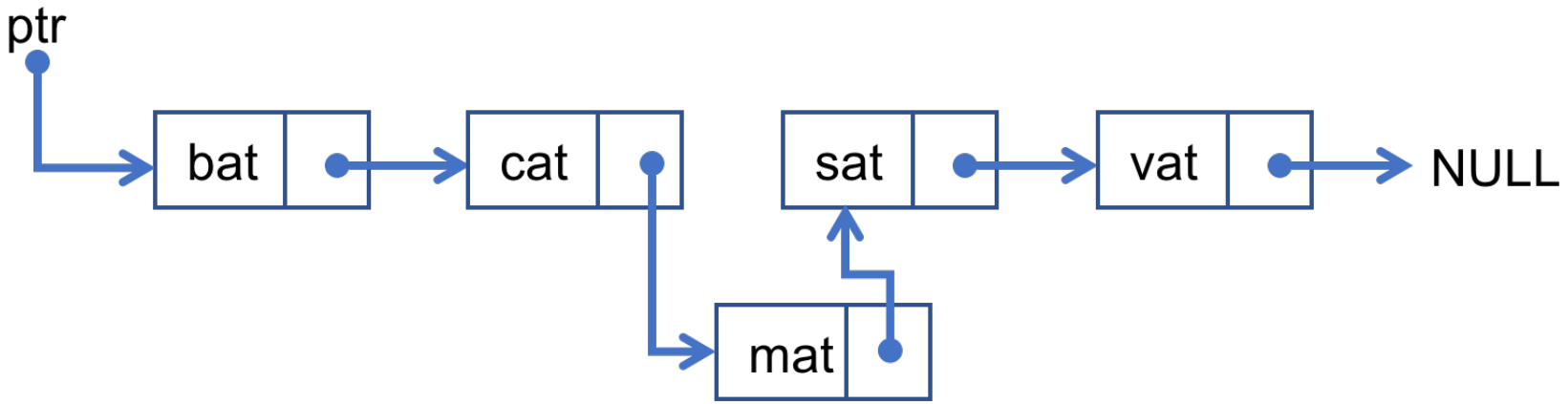
To insert the word `mat` between `cat` and `sat`



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`

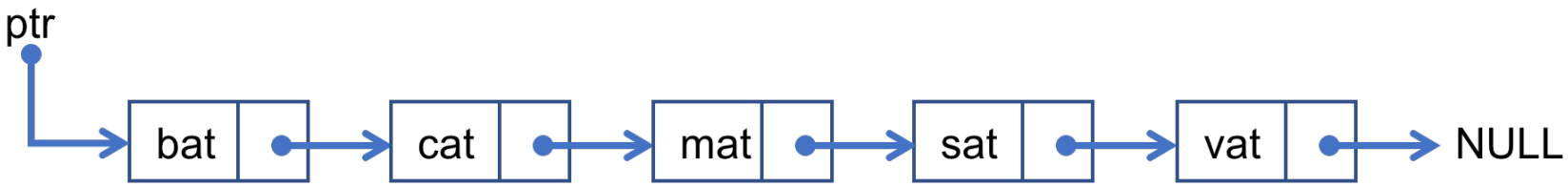


1. Get a node currently unused (`paddr`)
2. Set the data of this node to mat
3. Set `paddr`'s link to point to the address found in the link of the node `cat`
4. Set the link of the node cat to point to `paddr`

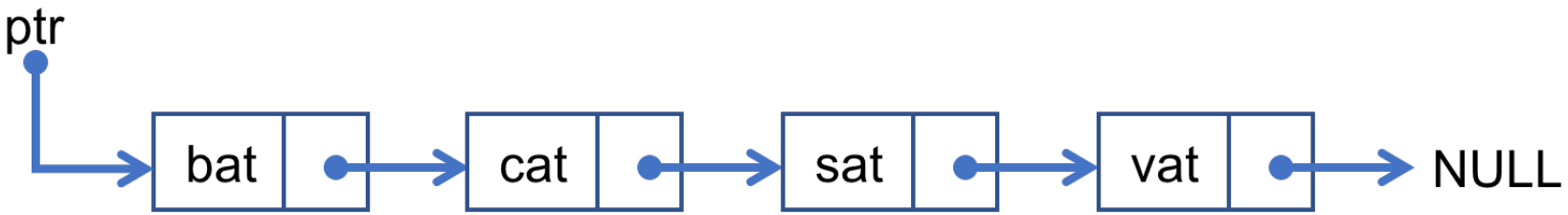
Singly Linked Lists: Delete

To delete `mat` from the list

- Goal

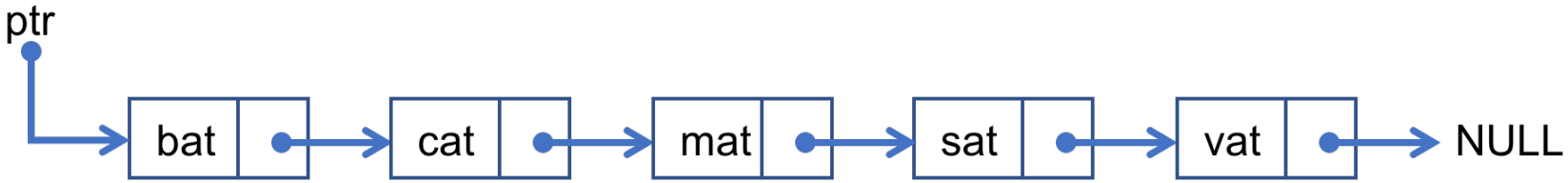


- Result



Singly Linked Lists: Delete

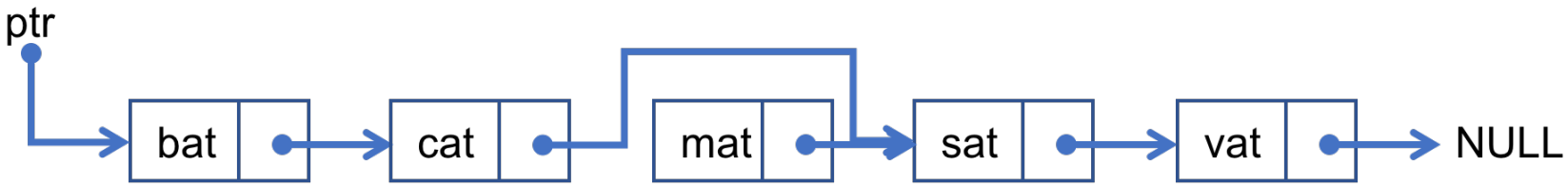
To delete `mat` from the list



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

Time Complexity

- O(1)

Singly Linked List: Features

Required capabilities to make linked representations possible
beginitemize

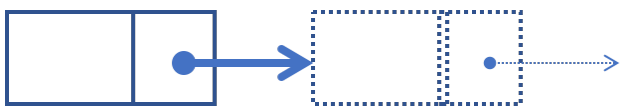
- A mechanism for defining a node's structure, that is, the field it contains
- A way to create new nodes when we need them
- A way to remove nodes that we no longer need

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

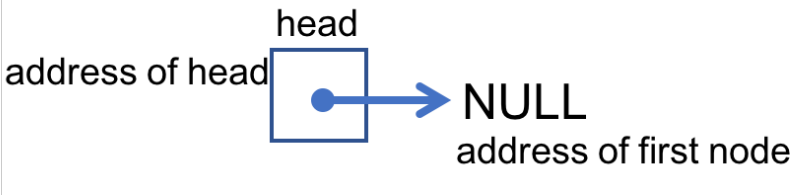
```
1  typedef struct node {
2      char data[4];
3      struct node *next; // self-referential structure
4  } node_t;
```



Singly Linked List: Creating a node

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

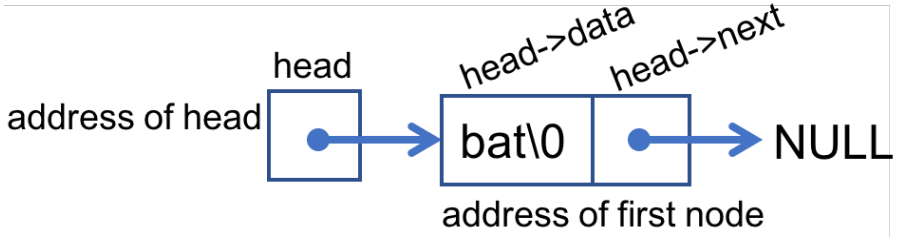
```
1  node_t *head = NULL;
2  head = malloc(sizeof(node_t));
3  if (head == NULL) {
4      return 1;
5  }
6  strcpy(head->data, "bat");
7  head->next = NULL;
```



Singly Linked List: Creating a node

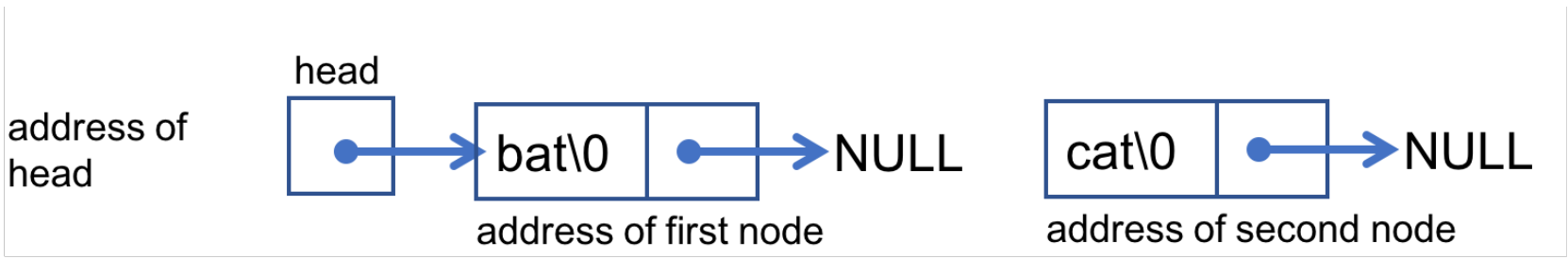
Create new nodes for our list then place the word bat into the list

```
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;
```



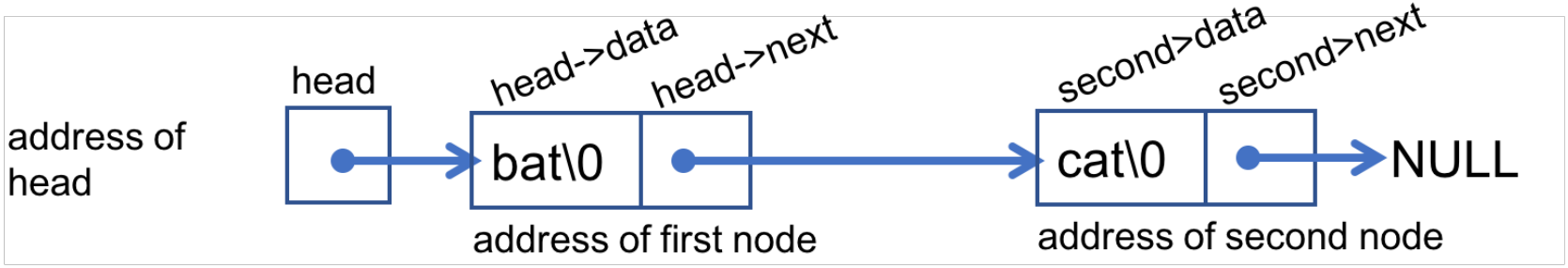
Singly Linked List: Adding a node

```
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 }
```

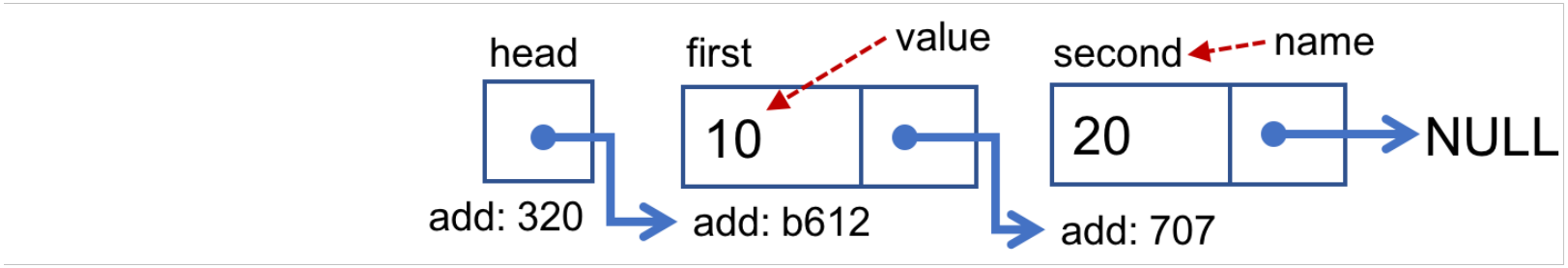
- Iterate the list using a pointer to find the end of the list
- Add a new node at the end of the list
- Add the data
- Mark next node as `NULL`

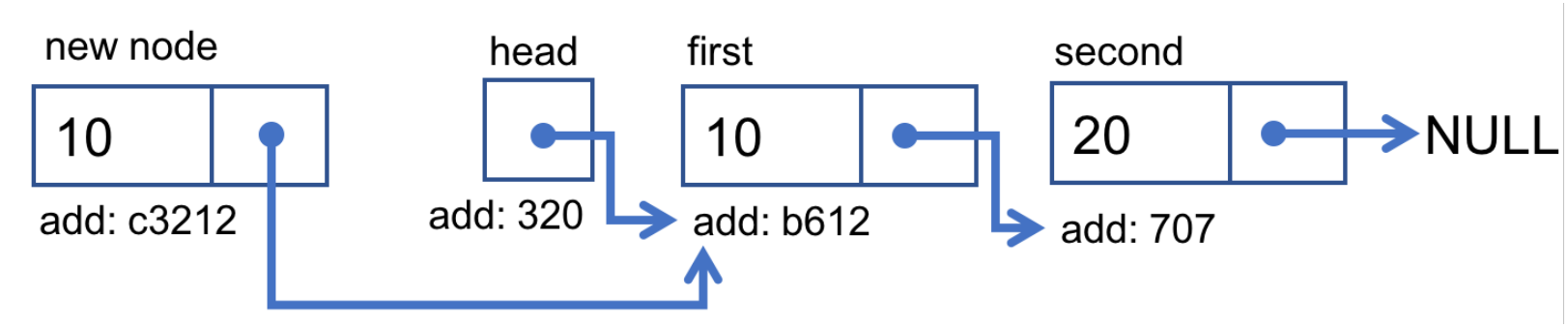
Singly Linked List: Insert at the beginning

```
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

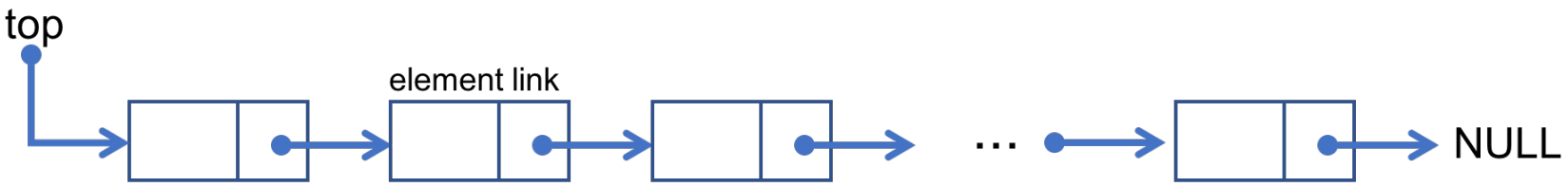


Figure: linked list stack

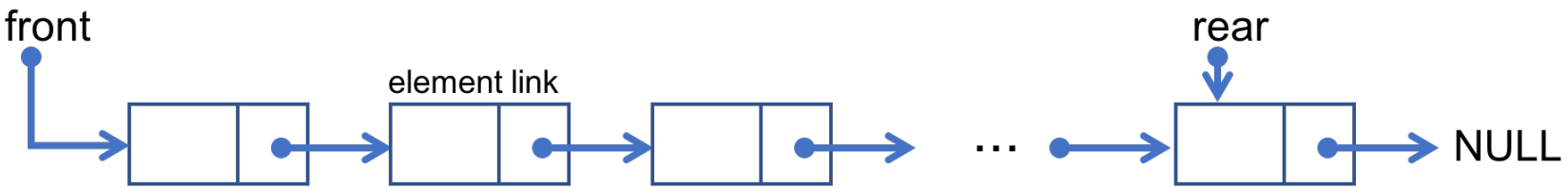


Figure: linked list queue

Representing stacks by linked list

```
1  typedef struct _stack_node {
2    int data;
3    struct _stack_node *next;
4  } stacknode;
5
6  typedef struct _stack_list {
7    stacknode *top;
8  } lstack;
9
10 void INIT(lstack *s) {
11   s->top = NULL;
12 }
13
14 int IS_FULL(lstack *s) {
15   return 0;
16 }
17
18 int IS_EMPTY(lstack *s) {
19   return (s->top == NULL);
20 }
21
22 void TOP(lstack *s) {
23   if (!IS_EMPTY(s))
24    printf("%d\n", s->top->data);
25 }
```

- Initial condition for stacks (No limits at the moment)
  - `top = NULL`,
    `0 ≤ i ≤ MAX_STACKS`
- Boundary condition for `n` stacks
  - Empty condition
    - `top = 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(lstack *s, int data) {
2      // IS_FULL(s) testing required
3      stacknode *t = (stacknode*)malloc(sizeof(stacknode));
4      if (t == NULL) {
5          printf("Failed to allocate\n");
6          return ;
7      }
8      t->data = data;
9      t->next = s->top;
10     s->top = t;
11 }
```

Poping a linked list stack

```
1  int POP(lstack *s) {
2    int data;
3    if(IS_EMPTY(s)) {
4      printf("Stack Empty\n");
5      return -1;
6    }
7    else {
8      stacknode *t = s->top;
9      data = t->data;
10     s->top = s->top->next;
11     free(t);
12     return data;
13   }
14 }
```

Representing Queues by linked list

```
1  struct _node {
2    int data;
3    struct _node *next;
4  } ;
5
6  typedef struct _node listnode;
7
8  struct _queue {
9    listnode *front;
10   listnode *rear;
11 } ;
12
13 typedef struct _queue queue;
```

- Initial condition for queue
  - `front = NULL`,
    `0 ≤ i ≤ MAX_QUEUES`
- Boundary condition for queues
  - Empty condition
    - `front = 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 enqueue(queue *q, int data)
2  {
3    listnode *newnode;
4    newnode = malloc(sizeof(listnode));
5    if(!newnode)
6    {
7      printf("failed to create node\n");
8      exit(-1);
9    }
10   newnode->data = data;
11   newnode->next = NULL;
12   if(q->rear)
13     q->rear->next = newnode;
14   q->rear = newnode;
15   if(q->front == NULL)
16     q->front = q->rear;
17   printf("enqueue %d\n", q->rear->data);
18 }
```

Delete from the front of a linked list queue

```
1  int dequeue(queue *q)
2  {
3    int data = 0;
4    listnode *temp;
5    if(IsEmptyQueue(q))
6    {
7      printf("queue is empty\n");
8      return 0;
9    }
10   else
11   {
12     temp = q->front;
13     data = q->front->data;
14     q->front = q->front->next;
15     free(temp);
16   }
17   printf("dequeue %d\n", data);
18   return data;
19 }
```

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

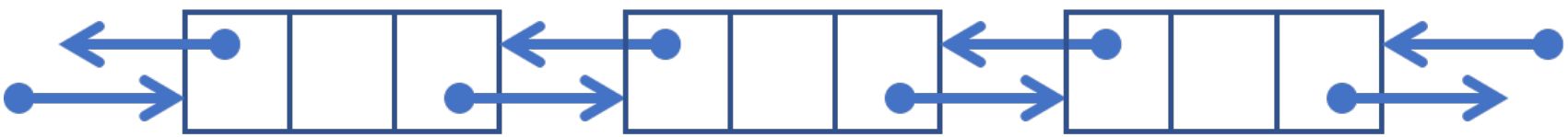
- Doubly lists + circular lists
- Allow two links
- Two way direction

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 Circular Lists

Introduce dummy node, called, head

- To represent empty list
- Make easy to implement operations
- Contains no information in item field

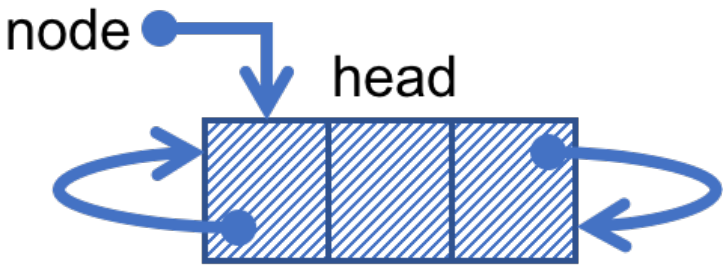


Figure: empty doubly linked circular list with head node

Doubly Linked Circular Lists

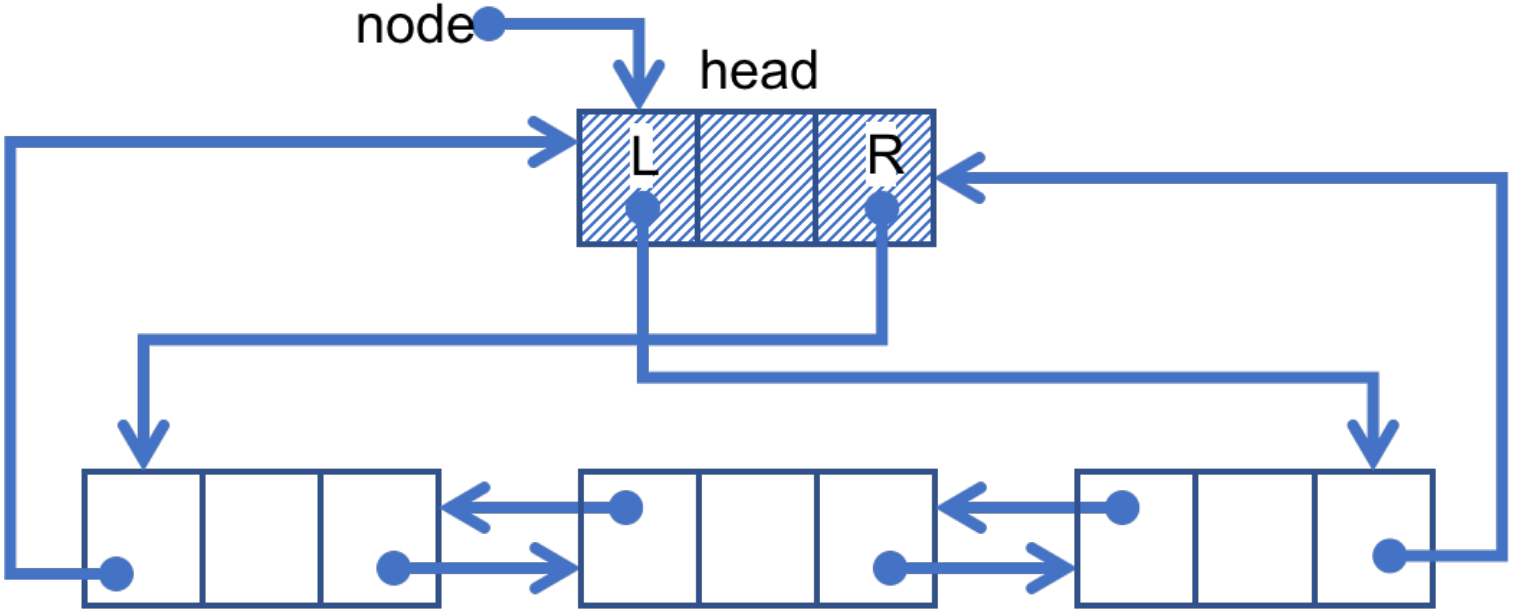


Figure: doubly linked circular list with head node

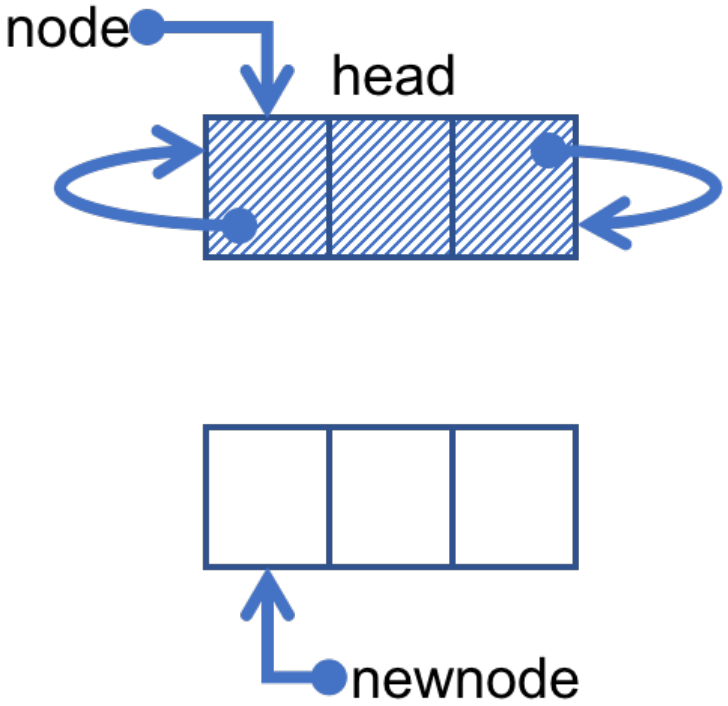
Insert into a doubly linked circular list

```
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)

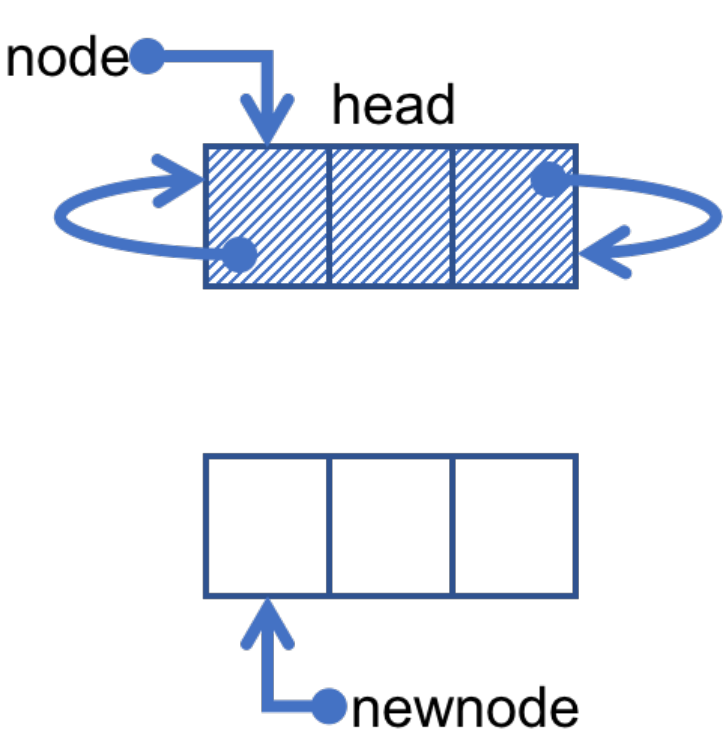
Insert into a doubly linked circular list

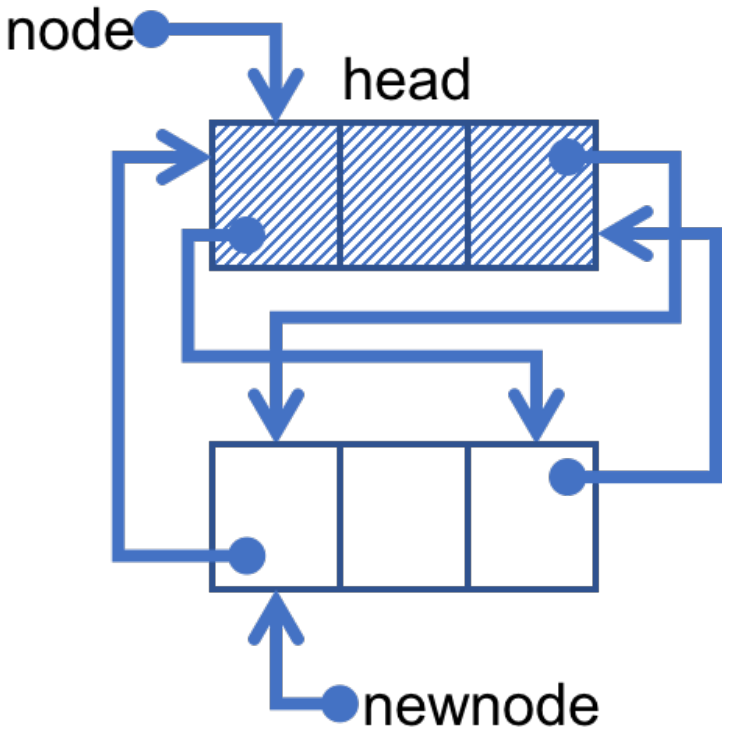
Insertion into an empty doubly linked circular list



Insert into a doubly linked circular list

Insertion into an empty doubly linked circular list





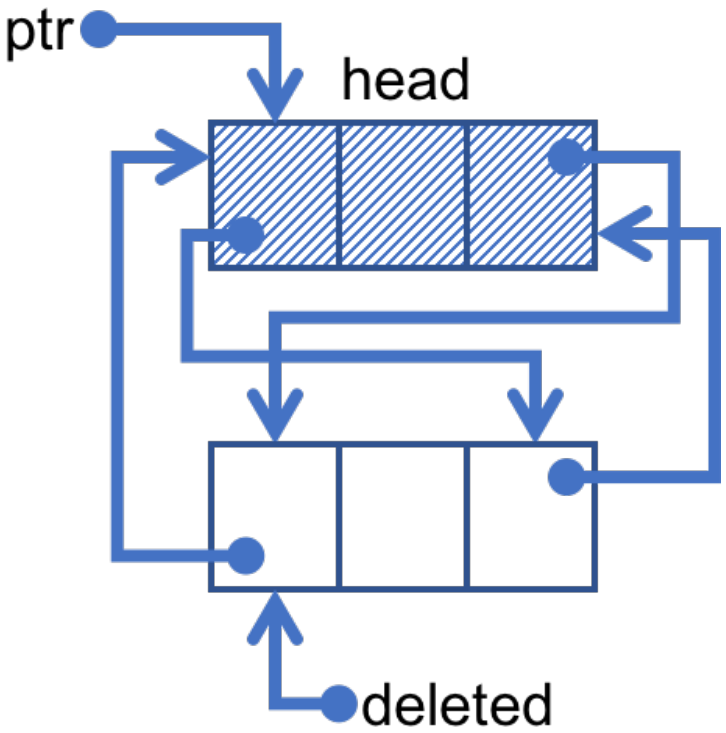
Deletion from a doubly linked circular list

```
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)

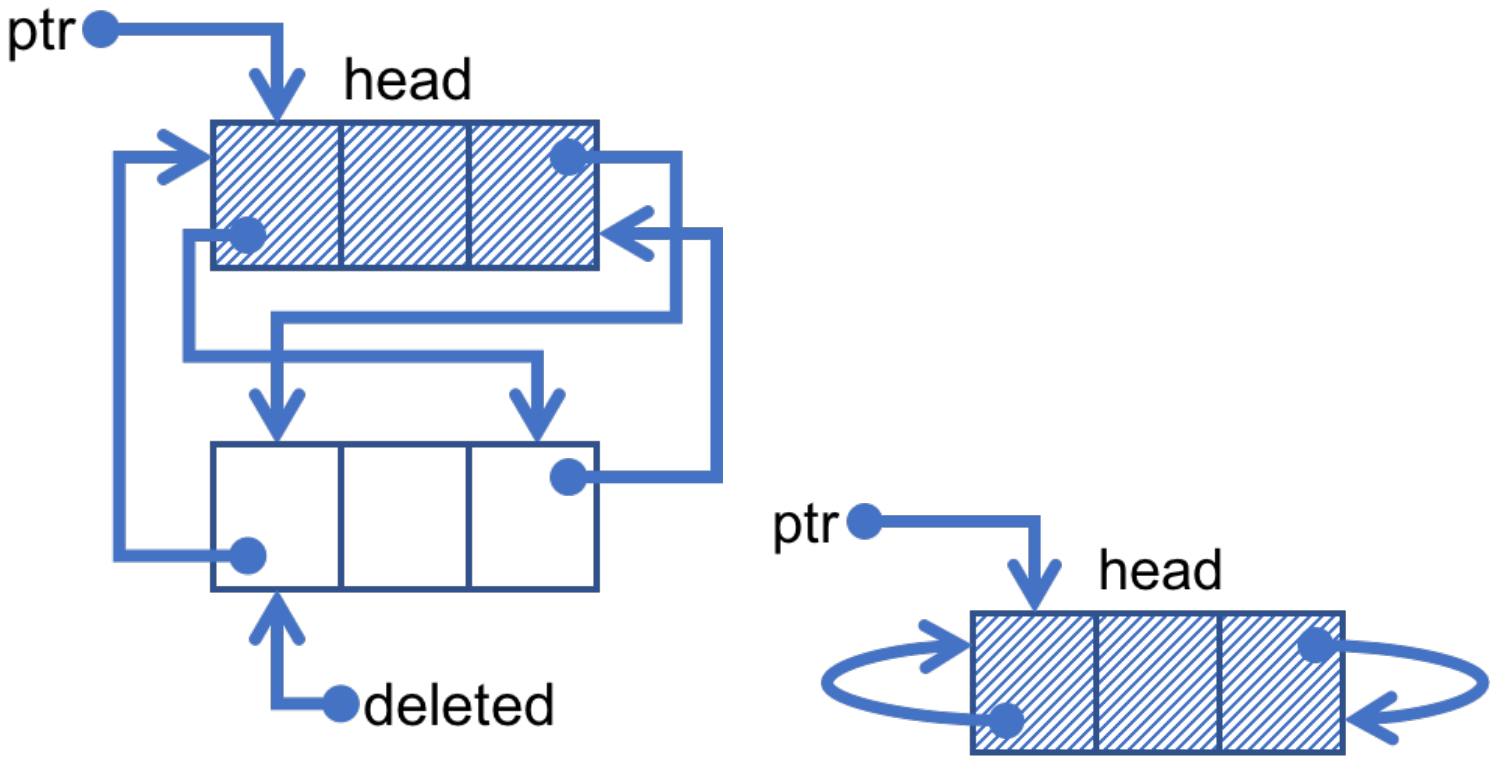
Deletion from a doubly linked circular list

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

Homework : Music Player

Function

- Play : Print current data
- Next : Move next music
- Back : Move back music
- Add Music : Add new music
- Delete Music : Delete current music
- Circular : If use the Next function with last music, move first music

Reference answer with cont'd skeleton code

Homework : Music Player I

```
1
2     typedef struct music
3     {
4     char musicName[15];
5     struct music *backMusic;
6     struct music *frontMusic;
7     } Music;
8
9     Music *headerPointer;
10    Music *currentMusic;
11
12    int main(void)
13    {
14    int i;
15
16    initMusic();
17    currentMusic = headerPointer;
18    char command[10];
19
20    //Code is required.
21
22    allFree();
23    return 0;
```

Homework : Music Player II

```
24      }
25
26      void initMusic()
27      {
28      headerPointer = (Music *)malloc(sizeof(Music));
29      headerPointer -> backMusic = headerPointer;
30      headerPointer -> frontMusic = headerPointer;
31      }
32
33      void playMusic()
34      {
35      //Code is required.
36      }
37
38      void Next()
39      {
40      //Code is required.
41      }
42
43
44      void Back()
45      {
46      //Code is required.
```

Homework : Music Player III

```
47      }
48
49
50      void addMusic()
51      {
52      char newdata[15];
53      scanf("%s", newdata);
54      Music *newMusic;
55      newMusic = (Music *)malloc(sizeof(Music));
56
57      //Code is required.
58      }
59
60
61      void deleteMusic()
62      {
63      if(headerPointer -> frontMusic == headerPointer)
64      {
65      printf("There is no music here\n");
66      }
67      else
68      {
69      Music *deletePointer;
```

Homework : Music Player IV

```
70      deletePointer = currentMusic;
71
72      //Code is required.
73
74      free(deletePointer);
75      }
76      }
77
78      void allFree()
79      {
80      Music *iter = headerPointer;
81      Music *iterNext = NULL;
82      do {
83      iterNext = iter -> frontMusic;
84      free(iter);
85      iter = iterNext;
86      } while(iter != headerPointer);
87
88      printf("____________________\n\n");
89      }
```