Data Structure and Algorithm

Class 6

Seongjin Lee

Updated: 2021-02-20
DSA_2017_06

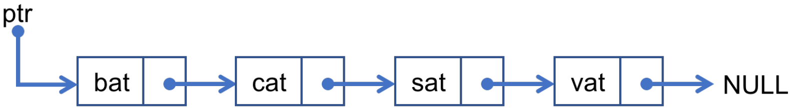
insight@gnu.ac.kr
http://resourceful.github.io
Systems Research Lab.
GNU

Table of contents

Singly Linked Lists

Singly Linked Lists

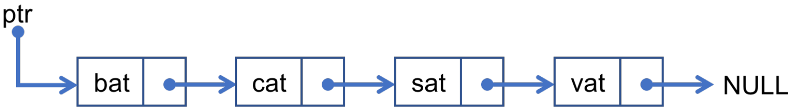
- Compose of data part and link part
- Link part contains address of the next element in a list
- Non-sequential representations
- Size of the list is not predefined
- Dynamic storage allocation and deallocation



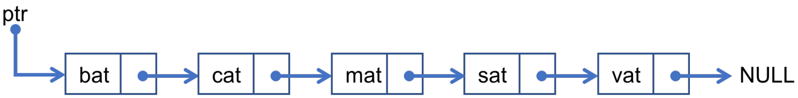
Singly Linked Lists: Insert

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

- Goal

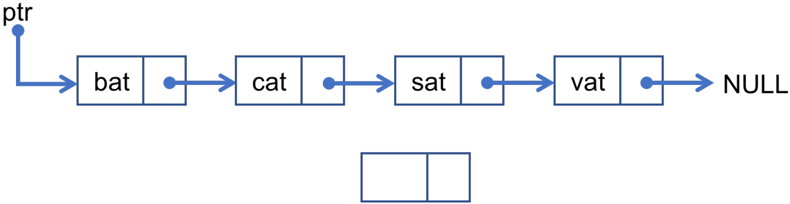


$\Downarrow$



Singly Linked Lists: Insert

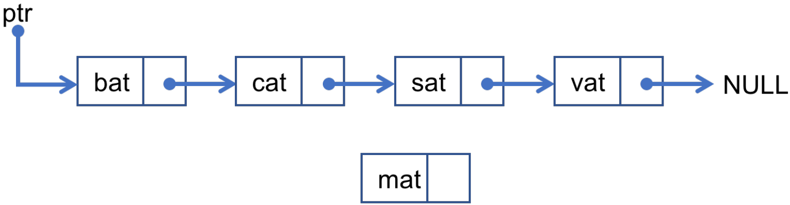
To insert the word `mat` between `cat` and `sat`



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

Singly Linked Lists: Insert

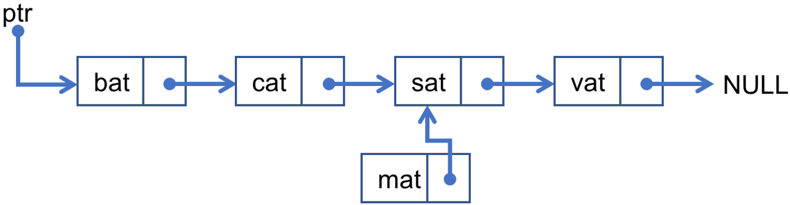
To insert the word `mat` between `cat` and `sat`



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

Singly Linked Lists: Insert

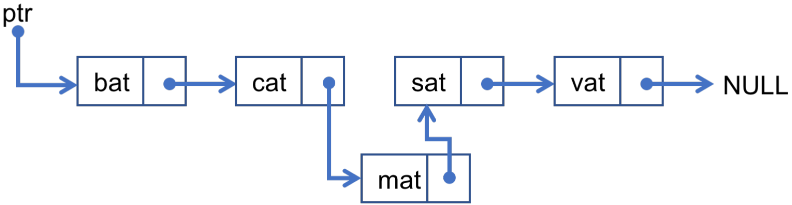
To insert the word `mat` between `cat` and `sat`



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`

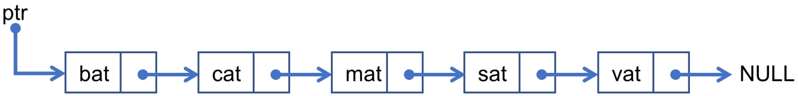


1. Get a node currently unused (`paddr`)
2. Set the data of this node to mat
3. Set `paddr`'s link to point to the address found in the link of the node `cat`
4. Set the link of the node cat to point to `paddr`

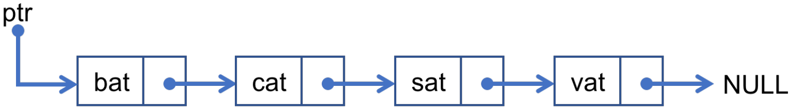
Singly Linked Lists: Delete

To delete `mat` from the list

- Goal

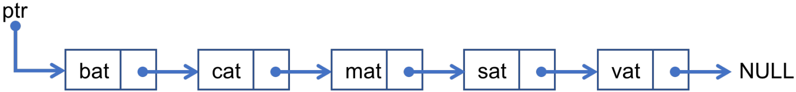


- Result



Singly Linked Lists: Delete

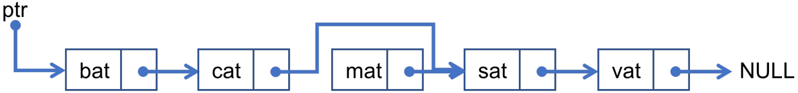
To delete `mat` from the list



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

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

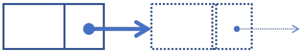
- A mechanism for defining a node's structure, that is, the field it contains
- A way to create new nodes when we need them
- A way to remove nodes that we no longer need

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

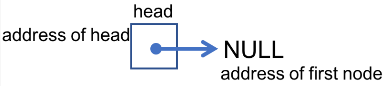
```
typedef struct node {
    char data[4];
    struct node *next; // self-referential structure
} node_t;
```



Singly Linked List: Creating a node

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

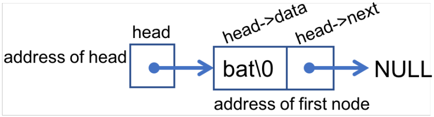
```
node_t *head = NULL;
head = malloc(sizeof(node_t));
if (head == NULL) {
    return 1;
}
strcpy(head->data, "bat");
head->next = NULL;
```



Singly Linked List: Creating a node

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

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



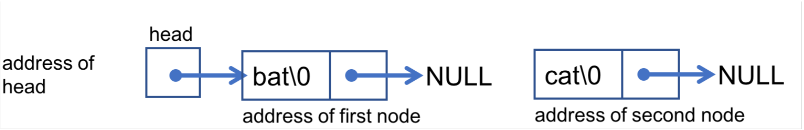
Singly Linked List: Adding a node

```
node_t second;

second = malloc(sizeof(node_t));

second->next=NULL;
strcpy(second->data, "cat");

head->next=second;
```



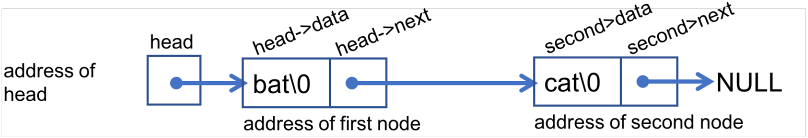
Singly Linked List: Adding a node

```
node_t second;

second = malloc(sizeof(node_t));

second->next=NULL;
strcpy(second->data, "cat");

head->next=second;
```



Singly Linked List: Implementation

- Data structure
- Insert a node
- Delete the list
- Remove a node
- Search
- Print

```
typedef struct node {
    int data;
    struct node *next;
} node_t;
```

Singly Linked List: Print

```
void print_list(node_t * head) {
    node_t * current = head;

    while (current != NULL) {
        printf("%d\n", current->data);
        current = current->next;
    }
}
```

- 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

```c
void insert_tail(node_t * head, int data) {
    node_t * current = head;
    while (current->next != NULL) {
        current = current->next;
    }

    /* now we can add a new variable */
    current->next = malloc(sizeof(node_t));
    current->next->data = data;
    current->next->next = NULL;
}
```

- 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`

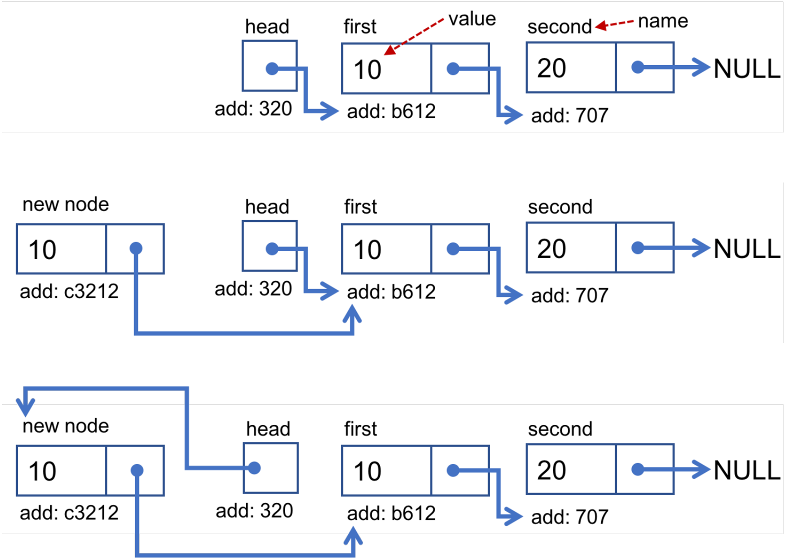
Singly Linked List: Insert at the beginning

```
void insert_head(node_t ** head, int data) {
    node_t * new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = data;
    new_node->next = *head;
    *head = new_node;
}
```

- 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

```c
int remove_head(node_t ** head) {
    int retval = -1;
    node_t * new_head = NULL;
    if (*head == NULL) {
        return -1;
    }
    new_head = (*head)->next;
    retval = (*head)->data;
    free(*head);
    *head = new_head;
    return retval;
}
```

- 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

```c
int remove_tail(node_t * head) {
    int retval = 0;
    if (head->next == NULL) { // only one node in the list
        retval = head->data;
        free(head); // remove the head
        return retval;
    }

    node_t * current = head;
    while (current->next->next != NULL) { // go to the second last node
        current = current->next;
    }

    retval = current->next->data; // get the last node's data
    free(current->next); // free the last node
    current->next = NULL; // set the next node NULL
    return retval;

}
```

Singly Linked List: Removing a value

```
int remove_value(node_t ** head, int data) {
    node_t *previous, *current;
    if (*head == NULL)
        return -1;

    if ((*head)->data == data)
        return pop(head);

    previous = current = (*head)->next;
    while (current != NULL) {
        if (current->data == data) {
            previous->next = current->next;
            free(current);
            return data;
        }

        previous = current;
        current = current->next;
    }
    return -1;
}
```

Singly Linked List: Deleting the list

```c
void delete_list(node_t *head) {
    node_t *current = head,
           *next = head;

    while (current) {
        next = current->next;
        free(current);
        current = next;
    }
}
```

Dynamically Linked Stacks and Queues

Representing Stack and Queue by Linked List

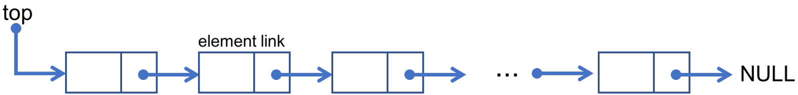


Figure: linked list stack

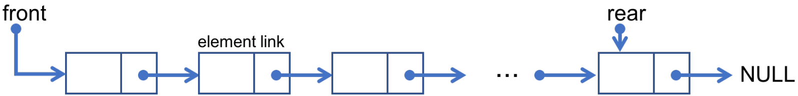


Figure: linked list queue

Representing stacks by linked list

```c
/* n=MAX_STACKS=10 */
#define MAX_STACKS 10

typedef struct {
    int key;
    /* other fields */
} element;

typedef struct stack *stack_ptr;

typedef struct stack {
    element item;
    stack_ptr link;
};

stack_ptr top[MAX_STACKS];
```

- Initial condition for stacks
  - `top[i] = NULL`, $0 \le$ `i` $\le$ `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

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

Poping a linked list stack

```
element pop(stack_ptr *ptop) {
    stack_ptr temp = *ptop;
    element item;
    if(IS_EMPTY(temp)) {
        fprintf(stderr,‘‘The stack is empty\n’’);
        exit(1);
     }
     item=temp->item;
     *ptop=temp->link;
     free(temp);
     return item;
}
```

Representing Queues by linked list

```
/* m=MAX_QUEUES=10 */
#define MAX_QUEUES 10

typedef struct queue *queue_ptr;

typedef struct queue {
    element item;
    queue_ptr link;
};

queue_ptr front[MAX_QUEUES],
          rear[MAX_QUEUES];
```

- Initial condition for queue
  - `front[i] = NULL`, $0 \le i \le$ `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

```
void insert(queue_ptr *pfront,
            queue_ptr *prear,
            element item) {
    queue_ptr temp = (queue_ptr)malloc(sizeof(queue));

    if(IS_FULL(temp)) {
        fprintf(stderr, ‘‘The memory is full\n’’);
        exit(1);
    }

    temp->item=item;
    temp->link=NULL;

    if (*pfront)
        (*prear)->link=temp;
    else
        *pfront = temp;

    *prear = temp;
}
```

Delete from the front of a linked list queue

```
element delete(queue_ptr *pfront) {
    queue_ptr temp=*pfront;
    element item;

    if (IS_EMPTY(*pfront)) {
        fprintf(stderr,‘‘The queue is empty\n’’);
        exit(1);
    }

    item=temp->item;
    *pfront=temp->link;
    free(temp);

    return item;
}
```

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

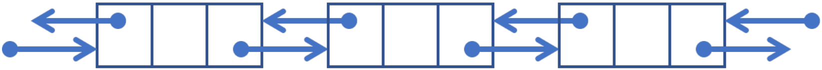
```
typedef struct node *node_ptr;

typedef struct node {
    node_ptr llink;
    element item;
    node_ptr rlink;
};
```

Doubly Linked Lists

```
typedef struct node *node_ptr;

typedef struct node {
    node_ptr llink;
    element item;
    node_ptr rlink;
};
```



Doubly Linked Lists

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

```
ptr

ptr->llink->rlink

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

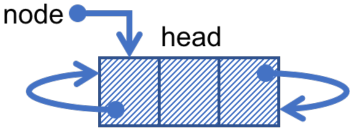


Figure: empty doubly linked circular list with head node

Doubly Linked Circular Lists

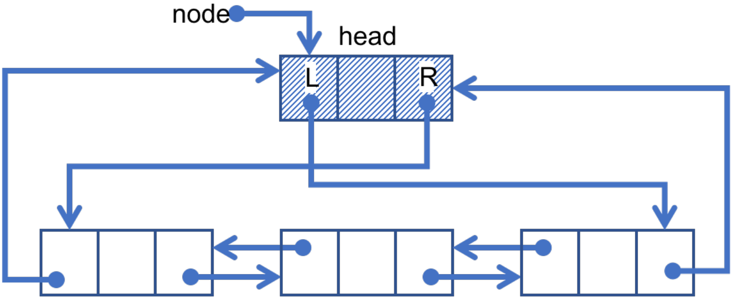


Figure: doubly linked circular list with head node

Insert into a doubly linked circular list

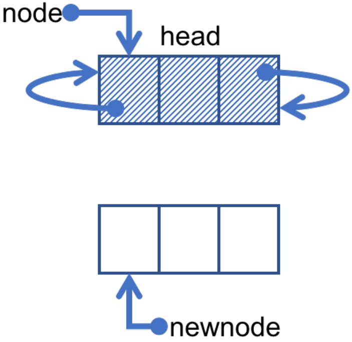
```
void d_insert(node_ptr node, node_ptr newnode) {
    /* insert newnode to the right of node */
    newnode->llink = node;
    newnode->rlink = node->rlink;

    node->rlink->llink = newnode;
    node->rlink = newnode;
}
```

Time complexity O(1)

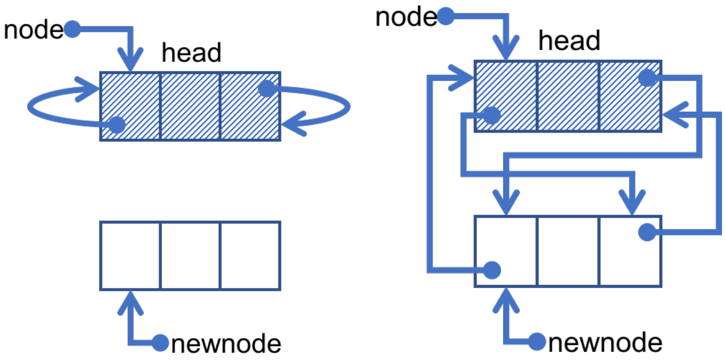
Insert into a doubly linked circular list

Insertion into an empty doubly linked circular list



Insert into a doubly linked circular list

Insertion into an empty doubly linked circular list



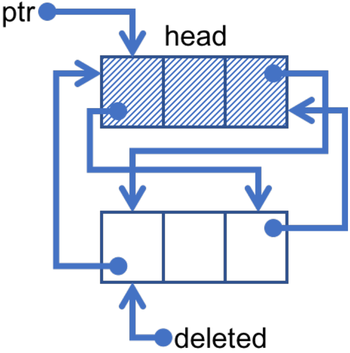
Deletion from a doubly linked circular list

```
void ddelete(node_ptr node, node_ptr deleted) {
    /* delete from the doubly linked list */
    if (node == deleted)
        printf("Deletion of head node not permitted.\n");
    else {
        deleted->llink->rlink = deleted->rlink;
        deleted->rlink->llink = deleted->llink;
        free(deleted);
    }
}
```

Time complexity: O(1)

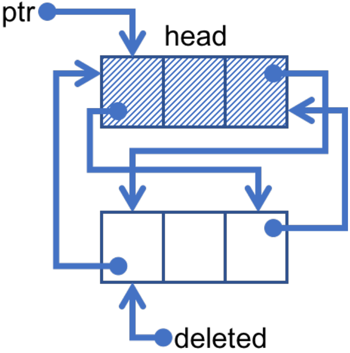
Deletion from a doubly linked circular list

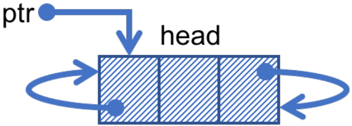
Deletion in a doubly linked circular list



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

```
typedef struct music
{
char musicName[15];
struct music *backMusic;
struct music *frontMusic;
} Music;

Music *headerPointer;
Music *currentMusic;

int main(void)
{
int i;

initMusic();
currentMusic = headerPointer;
char command[10];

//Code is required.

allFree();
return 0;
```

Homework : Music Player II

```
    }

    void initMusic()
    {
    headerPointer = (Music *)malloc(sizeof(Music));
    headerPointer -> backMusic = headerPointer;
    headerPointer -> frontMusic = headerPointer;
    }

    void playMusic()
    {
    //Code is required.
    }

    void Next()
    {
    //Code is required.
    }


    void Back()
    {
    //Code is required.
```

Homework : Music Player III

```
}


void addMusic()
{
char newdata[15];
scanf("%s", newdata);
Music *newMusic;
newMusic = (Music *)malloc(sizeof(Music));

//Code is required.
}


void deleteMusic()
{
if(headerPointer -> frontMusic == headerPointer)
{
printf("There is no music here\n");
}
else
{
Music *deletePointer;
```

Homework : Music Player IV

```
deletePointer = currentMusic;

//Code is required.

free(deletePointer);
}
}

void allFree()
{
Music *iter = headerPointer;
Music *iterNext = NULL;
do {
iterNext = iter -> frontMusic;
free(iter);
iter = iterNext;
} while(iter != headerPointer);

printf("____________________\n\n");
}
```