

Linked List

Singly Linked Lists, Chains, & Linked Stacks and Queues

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025



Outline

1 Singly Linked List and Chains

2 Representing Chains in C

3 Linked Stacks and Queues

- Linked Stacks
- Linked Queues

Outline

1 Singly Linked List and Chains

2 Representing Chains in C

3 Linked Stacks and Queues

- Linked Stacks
- Linked Queues

Definition

- We have learned **array** and **sequential mapping** (e.g., polynomial ADT).
 - Successive nodes of the data objects are stored in a fixed distance.

Definition

- We have learned **array** and **sequential mapping** (e.g., polynomial ADT).
 - Successive nodes of the data objects are stored in a fixed distance.
- **Issue:** When a sequential mapping is used for ordered lists:
 - no more available storage
 - waste of storage

Definition

- We have learned **array** and **sequential mapping** (e.g., polynomial ADT).
 - Successive nodes of the data objects are stored in a fixed distance.
- **Issue:** When a sequential mapping is used for ordered lists:
 - no more available storage
 - waste of storage
 - Excessive data movement is required for deletions and insertions.

Example

Alan	Bill	Carter	David	Elvis	Frank	
------	------	--------	-------	-------	-------	--

- Insert “Charlie” after Carter.

Example

Alan	Bill	Carter	David	Elvis	Frank	
------	------	--------	-------	-------	-------	--

- Insert “Charlie” after Carter.

Alan	Bill	Carter	Charlie	David	Elvis	Frank
------	------	--------	---------	-------	-------	-------

Example

Alan	Bill	Carter	David	Elvis	Frank	
------	------	--------	-------	-------	-------	--

- Insert “Charlie” after Carter.

Alan	Bill	Carter	Charlie	David	Elvis	Frank
------	------	--------	---------	-------	-------	-------

Three elements are moved.

Example

Alan	Bill	Carter	Charlie	David	Elvis	Frank
------	------	--------	---------	-------	-------	-------

- Delete “Carter” after Bill.

Example

Alan	Bill	Carter	Charlie	David	Elvis	Frank
------	------	--------	---------	-------	-------	-------

- Delete “Carter” after Bill.

Alan	Bill	Charlie	David	Elvis	Frank	
------	------	---------	-------	-------	-------	--

Example

Alan	Bill	Carter	Charlie	David	Elvis	Frank
------	------	--------	---------	-------	-------	-------

- Delete “Carter” after Bill.

Alan	Bill	Charlie	David	Elvis	Frank	
------	------	---------	-------	-------	-------	--

Four elements are moved.

Solution: linked presentation



- A linked list is comprised of nodes; each node has zero or more data fields and **one or more** link or pointer fields.
 - The nodes may be placed anywhere in memory.
 - The address of the next (or another) node in the list.

Singly Linked List

- In a singly linked list, each node has a pointer field.
- A singly linked list in which **the last node has a null link** is called a **chain**.

Singly Linked List

- In a singly linked list, each node has **exactly one** pointer field.
- A singly linked list in which **the last node has a null link** is called a **chain**.

Functions of Linked Lists (1/2)

- Insert (“Charlie”) after “Carter”.
 - ① Get an unused node a and set the data field of a to “Charlie”.
 - ② Set the link field of a to the node after “Carter”, which contains “David”.
 - ③ Set the link field of the node containing “Carter” to a .



Charlie

Functions of Linked Lists (1/2)

- Insert (“Charlie”) after “Carter”.

- ① Get an unused node a and set the data field of a to “Charlie”.
- ② Set the link field of a to the node after “Carter”, which contains “David”.
- ③ Set the link field of the node containing “Carter” to a .



Functions of Linked Lists (1/2)

- Insert (“Charlie”) after “Carter”.
 - ① Get an unused node a and set the data field of a to “Charlie”.
 - ② Set the link field of a to the node after “Carter”, which contains “David”.
 - ③ Set the link field of the node containing “Carter” to a .



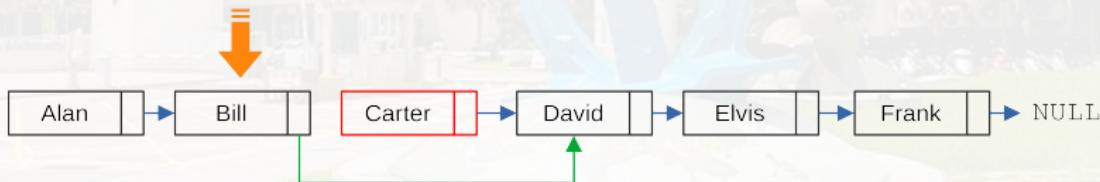
Functions of Linked Lists (2/2)

- Delete the node containing “Carter”.
 - ① Find the node *a* that **immediately precedes** the node containing “Carter”.
 - ② Set the link of *a* to point to “Carter”'s link.
 - ★ We don't need to move any data.
 - ★ If possible, free the memory space of node containing “Carter”.



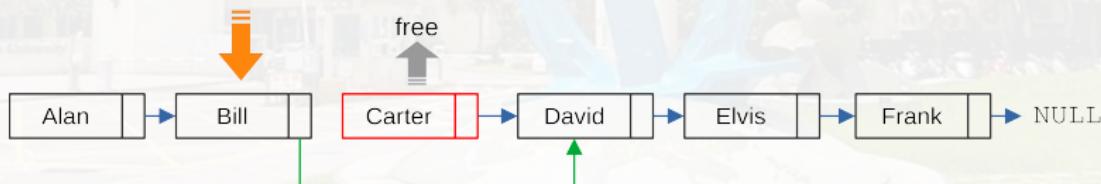
Functions of Linked Lists (2/2)

- Delete the node containing “Carter”.
 - Find the node *a* that immediately precedes the node containing “Carter”.
 - Set the link of *a* to point to “Carter”'s link.
 - We don't need to move any data.
 - If possible, free the memory space of node containing “Carter”.



Functions of Linked Lists (2/2)

- Delete the node containing “Carter”.
 - Find the node *a* that immediately precedes the node containing “Carter”.
 - Set the link of *a* to point to “Carter”'s link.
 - We don't need to move any data.
 - If possible, free the memory space of node containing “Carter”.



Outline

1 Singly Linked List and Chains

2 Representing Chains in C

3 Linked Stacks and Queues

- Linked Stacks
- Linked Queues

Pointers

- C provides extensive supports for pointers.

&: address operator

*: dereferencing (indirect) operator

```
int i, *pi; // i:integer variable; pi: a pointer to an integer.  
pi = &i;    // pi gets the address of i.  
i = 10;   // assign the value 10 to i  
*pi = 20; // assign the value 20 to i  
if (pi == NULL) ... // or if (!pi); test if the pointer is null.
```



Dynamically Allocated Storage

- C provides a mechanism, called **heap**, for allocating storage at run-time.

- **malloc** or **calloc**: dynamic memory allocation.
- **free**: free the memory previously (dynamically) allocated.

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *)malloc(sizeof(float));
*pi = 1024; *pf = 3.14;
free(pi);
free(pf);
```

Dynamically Allocated Storage

- How about C++?

- **new**: dynamic memory allocation.
- **delete**: free the memory previously (dynamically) allocated.

```
int i, *pi;  
float f, *pf;  
pi = new int;  
pf = new float;  
*pi = 1024; *pf = 3.14;  
delete pi;  
delete pf;
```

Using struct and typedef

```
struct employee {
    char name[4];
    struct employee *link;
};

typedef struct employee human; //usage: human h1, h2;
typedef struct employee *hPointer; // usage: hPointer link;
```

Variable or Structure?

```
struct {  
    char name[4];  
    int age;  
} person;
```

```
struct person {  
    char name[4];  
    int age;  
} human;
```

Variable or Structure?

```
struct {  
    char name[4];  
    int age;  
} person;
```

```
struct person {  
    char name[4];  
    int age;  
} human;
```

- The “structure” cannot be reused.
- The structure can be reused.

Self-Referential Structure

- Demo code.

```
struct Node {  
    int data;  
    struct Node *link;  
};
```

```
typedef struct Node Node;  
  
struct Node {  
    int data;  
    Node *link;  
};
```

Self-Referential Structure

- C allows us to create a pointer to a type that does not yet exist.

```
typedef struct listNode *listPointer; // listNode is still unknown!

struct listNode {
    char data[4];
    listPointer link;
};
```

More functions for linked lists

- To create a new empty list:
 - `listPointer first = NULL;`
- To test for an empty set:
 - `#define IS_EMPTY (first) (!(first))`
- To obtain a new node:
 - `first = (listPointer) malloc(sizeof(*first));`
- Enter “data” into the new node:
 - `strcpy(first->data, "data");`
 - `first->link = NULL;`

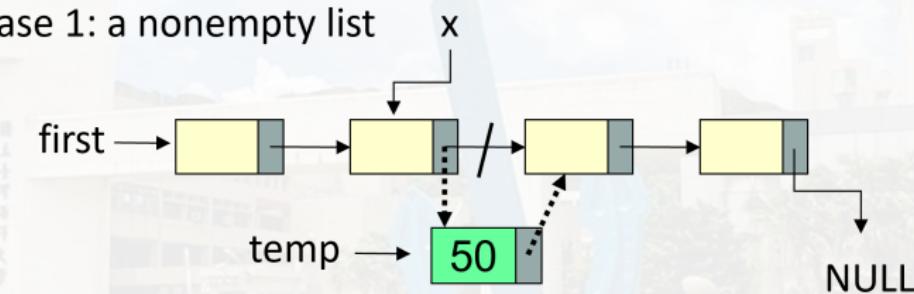
Further example: Create a two-node list

```
listPointer createTwo() {
    /* create a linked list with two nodes */
    listPointer first, second;
    first = (listPointer)malloc(sizeof(*first));
    second = (listPointer)malloc(sizeof(*second));
    second->link = NULL;
    second->data = 20; // or (*second).data = 20;
    first->data = 10;
    first->link = second;
    return first;
}
```

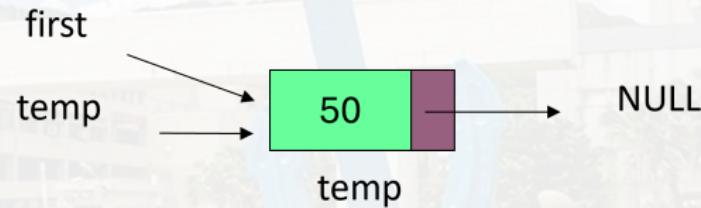
Simple insert into front of the list

```
void insert(listPointer *first, listPointer x) {
    /* insert a new node with data = 50 into the chain first after node x */
    listPointer temp;
    temp = (listPointer)malloc(sizeof(*temp));
    if(IS_FULL(temp)){ // check the capacity of the list first!
        printf("The memory is full\n");
        exit(1);
    }
    temp->data = 50; // get the data ready!
    if(*first) { //Case 1: nonempty list
        temp->link = x->link;
        x->link = temp;
    } else { //Case 2: empty list
        temp->link = NULL;
        *first = temp;
    }
}
```

Case 1: a nonempty list

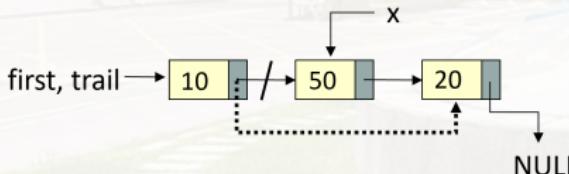
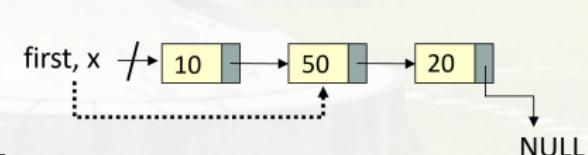


Case 2: an empty list



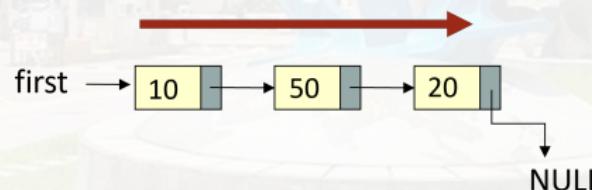
Delete a node from the list

```
void delete(listPointer first, listPointer trail, listPointer x) {  
    /* delete x from the list, trail points to the preceding node of x  
       and *first is the front of the list */  
    if (trail)          // Case 1: nonempty list  
        trail->link = x->link;  
    else                // Case 2:  
        *first = (*first)->link;  
    free(x);  
}
```

Case 1: $\text{trail} \neq \text{NULL}$ Case 2: $\text{trail} = \text{NULL}$ 

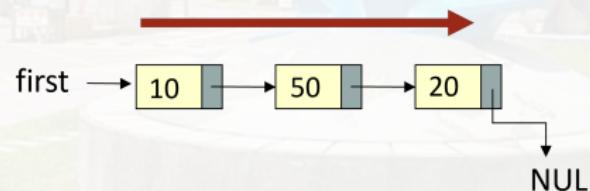
Printing a list

```
void printList(listPointer first) {  
    printf("The list contains: ");  
    for ( ; first ; first = first->link )  
        printf("%4d", first->data);  
    printf("\n");  
}
```



Printing a list

```
void printList(listPointer first) {
    printf("The list contains: ");
    while (first) {
        printf("%4d", first->data);
        first = first->link;
    }
    printf("\n");
}
```



Outline

1 Singly Linked List and Chains

2 Representing Chains in C

3 Linked Stacks and Queues

- Linked Stacks
- Linked Queues

Linked Stacks & Queues

- The links facilitate the implementation of stacks and queues.



(a) Linked stack



(b) Linked queue

Declarations & Initialization for Stacks

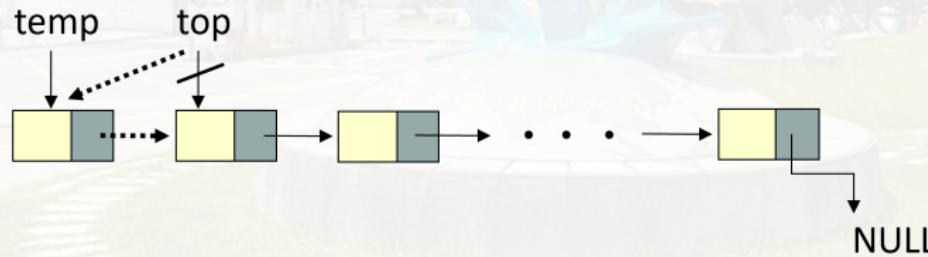
```
#define MAX_STACKS 10 /*maximum number of stacks*/
typedef struct {
    int key;
    /*other fields */
} element;

typedef struct stack* stackPointer;
struct stack {
    element data;
    stackPointer link;
};
stackPointer top[MAX_STACKS];
//Initialization
for (int i=0; i<MAX_STACKS; i++)
    top[i] = NULL;
```



Stack: push

```
void push(int i, element item) {  
    /* add item to the i-th stack */  
    stackPointer temp = malloc(sizeof(*temp));  
    temp->data = item;  
    temp->link = top[i];  
    top[i] = temp;  
}
```



Stack: pop

```
element pop(int i) { /* remove top element from the i-th stack*/
    stackPointer temp = top[i];
    element item;
    if (!temp)
        return stackEmpty();
    item = temp->data;
    top[i] = temp->link;
    free(temp); // Note: elements are dynamically allocated!
    return item;
}
```



A Simple Stack Implemented Using a Linked-List

Data: integers

- Refer to the codes [HERE](#).

```
typedef struct node {
    int data;
    struct node *link;
} Node;

typedef struct stack {
    Node *top;
} Stack;

Node stackEmpty() {
    printf("STACK EMPTY!\n");
    Node t;
    t.data = 0;
    t.link = NULL;
    return t;
}
```

```
void push(Stack *s, int item) {
/* add item to the stack */
    Node *temp = malloc(sizeof(Node));
    temp->data = item;
    temp->link = s->top;
    s->top = temp;
}

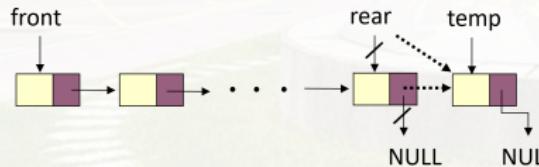
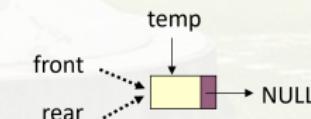
Node pop(Stack *s) {
/* remove top element from the stack */
    Node *temp = s->top;
    Node item;
    if (!temp) return stackEmpty();
    item.data = temp->data;
    s->top = temp->link;
    free(temp); // REMEMBER!
    return item;
}
```

Declarations & Initialization for Queues

```
#define MAX_QUEUES 10 /*maximum number of stacks*/  
  
typedef struct queue* queuePointer;  
  
struct queue {  
    element data;  
    queuePointer link;  
};  
queuePointer front[MAX_QUEUES], queuePointer rear[MAX_QUEUES];  
//Initialization  
for (int i=0; i<MAX_QUEUES; i++) {  
    front[i] = NULL; rear[i] = NULL;  
}
```

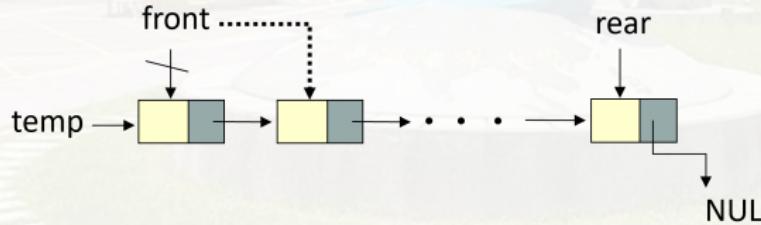
Queue: add (enqueue)

```
void add(i, item) { /* add item to the rear of queue i */
    queuePointer temp = malloc(sizeof(*temp));
    temp->data = item;
    temp->link = NULL;
    if (front[i])
        rear[i]->link = temp;
    else
        front[i] = temp;
    rear[i] = temp;
}
```

Case 1: $\text{front} \neq \text{NULL}$ Case 2: $\text{front} = \text{NULL}$ 

Queue: delete (dequeue)

```
element delete(int i) { /* delete an element from queue i */
    queuePointer temp = front[i];
    element item;
    if (!temp)
        return queueEmpty();
    item = temp->data
    front[i] = temp->link;
    free(temp);
    return item;
}
```



Discussions

