HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCTURES AND ALGORITHMS

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

7TH WEEK : LINKED LIST

ONE LOVE. ONE FUTURE.

CONTENTS

1. Basic concepts
2. Singly linked list
3. Operations on linked list

OBJECTIVES

After this lesson, students can:

1. Understand singly linked list data structure;
2. Build two basic operations on singly linked list data structures: Browse and Search

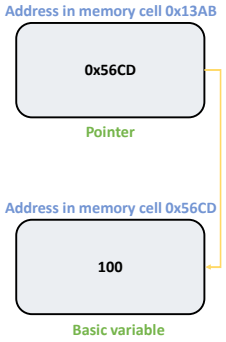
CONTENTS

1. Basic concepts
   1.1. Pointer
   1.2. Struct
2. Singly linked list
3. Operations on linked list
   3.1 Browse
   3.2 Search

1. BASIC CONCEPTS

1.1. Pointer

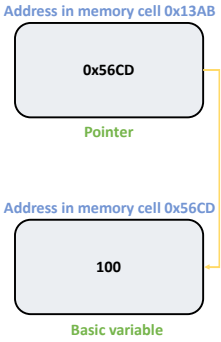
- Pointer is a basic concept in the C programming language, used to work with memory addresses.
- A pointer variable is also a variable, which also needs to be declared, initialized and used to store data.
- Pointer variables have their own addresses.
- A pointer variable does not store a value like a basic variable, it points to another address, which is an address in RAM.



1. BASIC CONCEPTS

1.1. Pointer

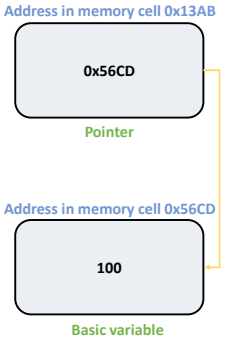
- Pointer is a basic concept in the C programming language, used to work with memory addresses.
- The data type of the pointer is the data type of the memory location to which it points.
- The value of the pointer contains the memory address to which the pointer points.
- The address of the pointer is the address of the pointer variable itself in RAM.



1. BASIC CONCEPTS

1.1. Pointer

- Declare: **`int *p;`**
  - Declare a pointer to point to an integer variable
  - The value of **p** determines the address of the variable
- Assign value: **`int a; int* p = &a;`**
  - **p** points to variable **a**



1. BASIC CONCEPTS

1.1. Pointer

- Example

```c
#include <stdio.h>
int main() {
    int* pc, c;

    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);

    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    *pc = 2;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c);
    return 0;
}
```

Print out the same memory address

Print value: 22

Print value : 22

Print value : 11

Print value : 2

Source: https://www.programiz.com/c-programming

1. BASIC CONCEPTS

1.2. Struct

- Struct is a data structure that is defined by the user (user defined datatype)
- A structure is a collection of variables, which can have different data types

```c
struct structureName {
  dataType member1;
  dataType member2;
  ...
};
```

```c
typedef struct  TNode{
          int a;
          double b;
          char* s;
      }TNode;
```


1. BASIC CONCEPTS

1.2. Struct

- A struct is a user-defined data type, consisting of a set of variables, which can have different data types.

```c
typedef struct  TNode{
          int a;
          double b;
          char* s;
      }TNode;
```

- **`TNode* q:`** **q** is a pointer that points to a variable of type **TNode**
- **Q→a:** access to member **a** of the struct type
- **`q = (TNode*)malloc(sizeof(TNode));`** allocate memory for a **TNode** type and **q** points to the allocated memory area

CONTENTS

1. Basic concepts
   1.1. Pointer
   1.2. Struct
2. Singly linked list
3. Operations on linked list
   3.1 Browse
   3.2 Search

2. SINGLY LINKED LIST

2.1. Introduction

- Singly linked list is an ordered list of elements; Elements are connected to each other through a link.
- ***Linked list and array***:

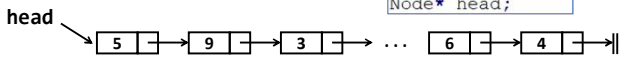
	Linked list	Array
Data structure type	Don't need to be the same	Must be the same
Allocate memory	Disperse	Continuous, side by side

2. SINGLY LINKED LIST

2.1. Introduction

- **Characteristic**:
  - Each element of the list consists of two parts: Data and pointer, which stores the address of the next element in the list;
  - In a singly linked list, each element only points to the next element;
  - A linked list has a first element: that calls **head**; and a last element, the pointer part of the last element is always null.

```
struct Node{
    int value;
    Node* next;
};
Node* head;
```



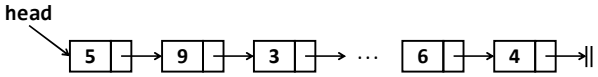
CONTENTS

1. Basic concepts
   1.1. Pointer
   1.2. Struct
2. Singly linked list
3. Operations on linked list
   3.1 Browse
   3.2 Search

3. OPERATIONS ON SINGLY LINKED LIST

3.1. Browse the list

- **Task:** Visit each element of the list exactly once
- **Idea:** Use the **next** pointer to access the next element



3. OPERATIONS ON SINGLY LINKED LIST

3.1.Browse the list

- **Task:** Visit each element of the list exactly once

```c
#include <stdio.h>

typedef struct Node{
    int value;
    struct Node* next;
}Node;

//Create a physical node
Node*makeNode(int v){
    Node* p = (Node*)malloc(sizeof(Node));
    p->value = v;
    p->next = NULL;
    return p;
}
```

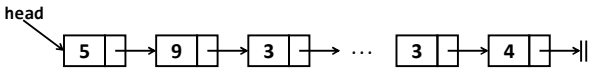
```c
//Print a list
void printList(Node* h){
  Node* p = h;
  while(p != NULL){
    printf("%d ",p->value);
    p = p->next;
  }
}

int main() {
    Node* head, *node1, *node2;
    head = makeNode(10);
    node1 = makeNode(20);
    node2 = makeNode(30);

    head->next = node1;
    node1->next = node2;

    printList(head);
    return 0;
}
```


3. OPERATIONS ON SINGLY LINKED LIST

3.2. Search

- **Task**: Find the first element of the list whose value is equal to the input value
- **Idea**: Use the **next** pointer to access the next element

For example, find the first element of the list with value 3



3. OPERATIONS ON SINGLY LINKED LIST

3.1. Search

- **Task**: Find the first element of the list whose value is equal to the input value

```c
#include <stdio.h>

typedef struct Node{
    int value;
    struct Node* next;
}Node;

//Create a physical node
Node*makeNode(int v){
    Node* p = (Node*)malloc(sizeof(Node));
    p->value = v;
    p->next = NULL;
    return p;
}

//Find a node with given value
Node * findFirst(Node * head, int val){
    Node* p = head;
    while(p != NULL){
        if(p->value == val)
            return p;
        p = p->next;
    }
    return NULL;
}
```

```c
int main() {
    Node* head, *node1, *node2;
    head = makeNode(10);
    node1 = makeNode(20);
    node2 = makeNode(30);

    head->next = node1;
    node1->next = node2;

    Node * res = findFirst(head, 20);
    if(res != NULL){
        printf("Found");
    }else{
        printf("Not Found");
    }

    return 0;
}
```

SUMMARY AND SUGGESTIONS

1. **Summary:**

   The lesson introduced singly linked lists and two basic operations on singly linked lists: browse and search

2. **Suggestions:**

   Design and implement other operations on the list

CONTENTS

1. **Insert an element at the beginning of the list**
2. **Insert an element at the end of the list**
3. **Insert an element before an element in the list**

OBJECTIVES

After this lesson, students can:

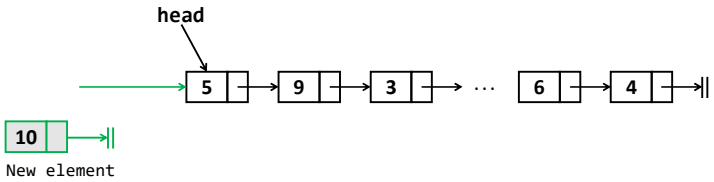
Understand the algorithm and successfully implement three basic operations on singly linked lists: inserting an element at the beginning, end, and before an element in a singly linked list.

CONTENTS

1. **Insert an element at the beginning of the list**
2. Insert an element at the end of the list
3. Insert an element before an element in the list

1. INSERT AN ELEMENT AT THE BEGINNING OF THE LIST

1.1. Idea



Step 1: Create a new element
Step 2: Update **head pointer** to point to the new element
Step 3: Update the **next pointer** of the new element: makes it points to the beginning of the old list, thus the new element becomes the first element of the list

1. INSERT AN ELEMENT AT THE BEGINNING OF THE LIST

1.2. Implement

```
Node* insertFirst(Node * head, int v){
    Node * new_node = makeNode(v);
    if(head == NULL) return new_node;
    else{
        new_node->next = head;
        head = new_node;
        return head;
    }
}
```

Create new element

If the current list is empty, returns the new element

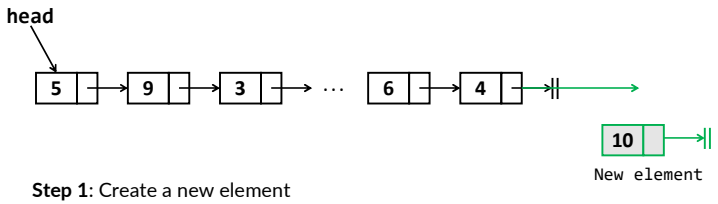
(1) Update **head pointer** to point to the new element
(2) Update the **next pointer** of the new element: makes it points to the beginning of the old list, thus the new element becomes the first element of the list

CONTENTS

1. Insert an element at the beginning of the list
2. Insert an element at the end of the list
3. Insert an element before an element in the list

2. INSERT AN ELEMENT AT THE END OF THE LIST

2.1. Idea



Step 1: Create a new element
Step 2: Find the last element of the list
Step 3: Update the **next pointer** of the last element to point to the new element

2. INSERT AN ELEMENT AT THE END OF THE LIST

2.2. Implement

```
Node * findLastNode(Node * head){
    Node* p = head;
    while(p != NULL){
        if(p->next == NULL) return p;
        p = p->next;
    }
    return NULL;
}
```

⇐ Use a loop to find the last element of a list

```
Node * insertLast(Node* head, int v){
    Node * new_node = makeNode(v);
    if(head == NULL) return new_node;
    else{
        Node * lastNode = findLastNode(head);
        lastNode->next = new_node;
        return head;
    }
}
```

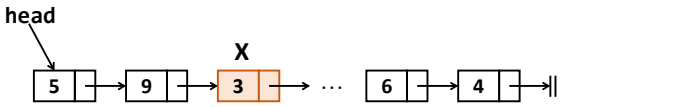
⇐ (1) Create new element
(2) Insert the new element after the last element of the list

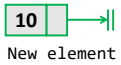
CONTENTS

1. Insert an element at the beginning of the list
2. Insert an element at the end of the list
3. Insert an element before an element in the list

3. INSERT AN ELEMENT BEFORE AN ELEMENT OF THE LIST

3.1. Idea





Insert anew element before the element X of the list:

Step 1: Create a new element

Step 2: Update the next pointer of the element before X as the new element

Step 3: Update the **next pointer** of new element as the X

3. INSERT AN ELEMENT BEFORE AN ELEMENT OF THE LIST

3.1. Implement

```
Node* prevNode(Node* head, Node* p){
  Node* q = head;
  while(q != NULL){
    if(q->next == p) return q;
    q = q->next;
  }
  return NULL;
}
```

⇐ Find the element preceding a given node

```
Node * insertBeforeNode(Node* head, Node* p, int v){
    Node* pp = prevNode(head, p);
    if(pp == NULL && p != NULL)  return head;
    Node* q = makeNode(v);
    if(pp == NULL){
        if(head == NULL)
            return q;
        q->next = head;
        return q;
    }
    q->next = p;
    pp->next = q;
    return head;
}
```

⇐ Insertion

SUMMARY AND SUGGESTIONS

1. **Summary:**

   Implement three operations to insert a new element into a singly linked list: **insert an element at the beginning, at the end, and before an element of the list**.

2. **Suggestions:**

   Design and implement other operations on the list

CONTENTS

1. **Delete an element of the list**
2. **Reverse the order of list elements**

OBJECTIVES

After this lesson, students can:

Understand the algorithm and successfully implement two basic operations on singly linked lists:

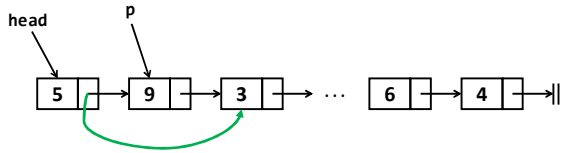
- remove an element from the list
- reverse the order of elements in a list

CONTENTS

1. **Delete an element of the list**
2. Reverse the order of list elements

1. DELETE AN ELEMENT OF THE LIST

1.1. Idea



- Check if the list and element to be deleted **p** are NULL?
- If the element to be deleted is at the beginning of the list → Simple
- Use recursion to delete

1. DELETE AN ELEMENT OF THE LIST

1.2. Implement

```
//Remove
Node* removeNode(Node* head, Node * p){
    if(head == NULL || p == NULL) return head;
    if(head == p){
        head = head->next;
        free(p);
        return head;
    }else{
        head->next = removeNode(head->next, p);
        return head;
    }
}
```

If the list and element to be deleted **p** are NULL, return head of the list

If the element to be deleted is the first element in the list, change the first element and delete the element that want to delete

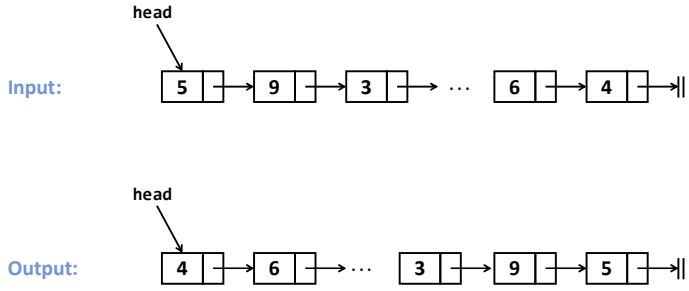
Apply recursive technique for deletion

CONTENTS

1. Delete an element of the list
2. **Reverse the order of list elements**

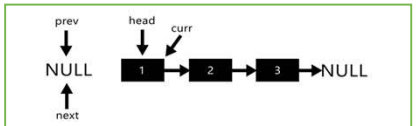
2. REVERSE THE ORDER OF THE LIST ELEMENTS

2.1. Problem



2. REVERSE THE ORDER OF THE LIST ELEMENTS

2.2. Idea




```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

Nguồn: https://www.geeksforgeeks.org/reverse-a-linked-list/

2. REVERSE THE ORDER OF THE LIST ELEMENTS

2.3. Implement

```
//Reverse
Node* reverse(Node* head){
    Node* cur = head;
    Node* next, *pre;
    pre = NULL;
    next = NULL;

    while(cur != NULL){
        //Get next
        next = cur->next;
        //Reverse
        cur->next = pre;
        //Move points ahead
        pre = cur;
        cur = next;
    }
    head = pre;
    return head;
}
```


SUMMARY AND SUGGESTIONS

1. Summary:

Implement two important operations on singly linked lists: **remove an element from the list and reverse the order of the list's elements**.

2. Suggestions:

- A singly linked list has only 1 link between 2 consecutive elements in the list. If there are 2 links, is it easier to operate on the list?

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !