Cấu trúc dữ liệu cơ bản

Mảng động, danh sách liên kết đơn, danh sách liên kết đôi, ngăn xếp, hàng đợi

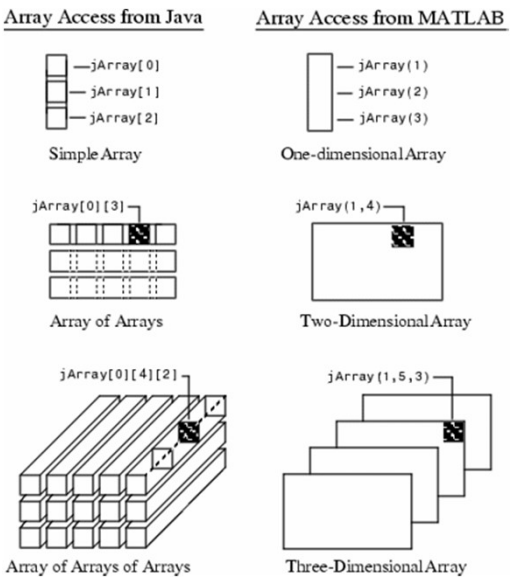
Nội dung

- Mảng động
- Danh sách liên kết đơn
- Danh sách liên kết đôi
- Danh sách tuyến tính
- Ngăn xếp – stack
- Hàng đợi – Queue

Cấu trúc dữ liệu

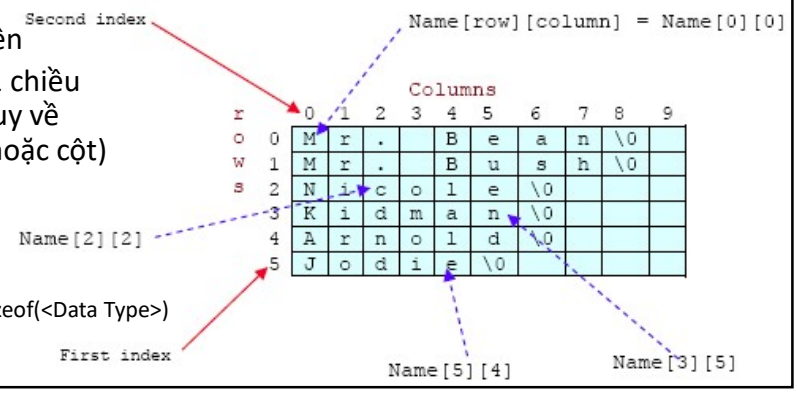
- Mô tả cách lưu trữ dữ liệu của bài toán vào trong máy tính
- Ảnh hưởng tới hiệu quả của thuật toán
- Các thao tác chính với một CTDL là
  - Duyệt
  - Tìm kiếm
  - Thêm phần tử
  - Xóa phần tử
  - Sắp xếp
  - Trộn
  - …

Array – Mảng



Mảng

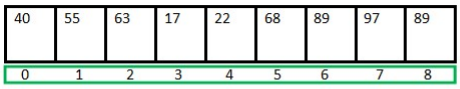
- **Mảng** - Array: là cấu trúc dữ liệu được cấp phát liên tục (liên tiếp) cơ bản
  - gồm các bản ghi có kiểu giống nhau, có kích thước cố định.
  - Mỗi phần tử được xác định bởi chỉ số (địa chỉ), là vị trí tương đối so với địa chỉ phần tử đầu mảng
  - Tên mảng = Hằng con trỏ trỏ tới địa chỉ phần tử đầu tiền
  - Trong máy tính chỉ có mảng 1 chiều mảng nhiều chiều sẽ được quy về mảng 1 chiều (ưu tiên hàng hoặc cột)



&Name[0][0] = Name
&Name[0][4] = Name + 4 * sizeof(<Data Type>)
&Name[i][j] = Name + (i*<column size> + j) * sizeof(<Data Type>)

Mảng

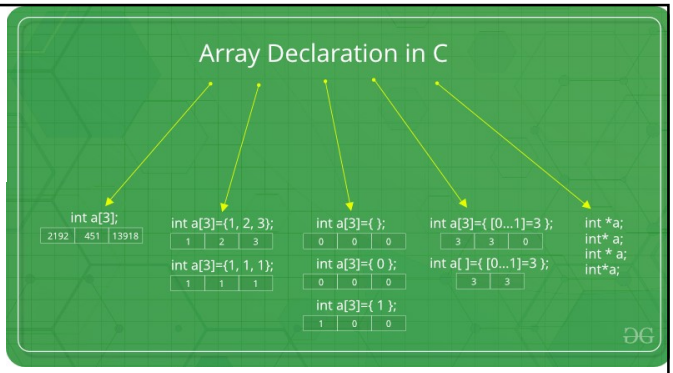
- **Ưu điểm** của mảng:
  - **Truy cập phần tử với thời gian hằng số $O(1)$**: vì thông qua chỉ số của phần tử ta có thể truy cập trực tiếp vào ô nhớ chứa phần tử.
  - **Sử dụng bộ nhớ hiệu quả**: chỉ dùng bộ nhớ để chứa dữ liệu nguyên bản, không lãng phí bộ nhớ để lưu thêm các thông tin khác.
  - **Tính cục bộ về bộ nhớ**: các phần tử nằm liên tục trong 1 vùng bộ nhớ, duyệt qua các phần tử trong mảng rất dễ dàng và nhanh chóng.
  - Các phần tử đặt dưới 1 tên chung nên dễ quản lý
- **Nhược điểm**:
  - không thể thay đổi kích thước của mảng khi chương trình đang thực hiện.
  - Các thao tác thêm/xóa phần tử mà dẫn đến phải dịch phần tử sẽ có chi phí lớn



Array Length = 9
First Index = 0
Last Index = 8

- Trong C/C++
  - Chỉ số bắt đầu từ 0
  - Có nhiều cách khai báo và khởi tạo mảng (chú ý phiên bản của C/C++)
  - KHÔNG check truy cập vượt ngoài phạm vi khai báo (các trình biên dịch có thể đưa ra cảnh báo với TH này)
  - KHÔNG check nếu khởi tạo quá số lượng

```
int arr[2] = { 10, 20, 30, 40, 50 };
```




```
int arr[2];

cout << arr[3] << " ";
cout << arr[-2] << " ";
```

https://www.geeksforgeeks.org/arrays-in-c-cpp/

Mảng trong C/C++

- Các phần tử sắp liên tiếp trong bộ nhớ

```
int arr[5], i;
cout << "Size of integer in this compiler is "
     << sizeof(int) << "\n";
for (i = 0; i < 5; i++)
    // The use of '&' before a variable name, yields
    // address of variable.
    cout << "Address arr[" << i << "] is " << &arr[i] << "\n";
```

```
Size of integer in this compiler is 4
Address arr[0] is 0x7ffe75c32210
Address arr[1] is 0x7ffe75c32214
Address arr[2] is 0x7ffe75c32218
Address arr[3] is 0x7ffe75c3221c
Address arr[4] is 0x7ffe75c32220
```

- Duyệt mảng

```
int arr[6]={11,12,13,14,15,16};
// Way -1
for(int i=0;i<6;i++)
    cout<<arr[i]<<" ";

cout<<endl;
  // Way 2
cout<<"By Other Method:"<<endl;
for(int i=0;i<6;i++)
    cout<<i[arr]<<" ";

cout<<endl;
```

```
11 12 13 14 15 16
By Other Method:
11 12 13 14 15 16
```

Mảng trong C/C++

- Mảng và con trỏ
  - Tên mảng = hằng con trỏ, trỏ tới ô nhớ đầu tiền cấp phát cho mảng (địa chỉ phần tử đầu tiên)
  - Không thể thay đổi địa chỉ mảng sau khi đã khai báo (KHÔNG thể gán 2 mảng trực tiếp)
  - Có thể dùng biến con trỏ để truy cập các phần tử trong mảng
  - Toán tử ++ và -- với con trỏ trỏ đến mảng để truy cập tới phần tử cách phần tử hiện tại 1 phần tử (về sau hoặc ở ngay trước)
- Vector trong C++
  - Có trong STL của C++
  - Không cần chỉ ra trước số lượng phần tử tối đa (tự điều chỉnh theo nhu cầu)
  - Hỗ trợ sẵn một số hàm thêm, xóa và tìm kiếm
  - Thời gian thêm/xóa KHÔNG còn là hằng số như trong mảng thường (VD. thêm cuối)

```
int arr[] = { 10, 20, 30, 40, 50, 60 };
int* ptr = arr;
cout << "arr[2] = " << arr[2] << "\n";
cout << "*(arr + 2) = " << *(arr + 2) << "\n";
cout << "ptr[2] = " << ptr[2] << "\n";
cout << "*(ptr + 2) = " << *(ptr + 2) << "\n";
```

```
arr[2] = 30
*(arr + 2) = 30
ptr[2] = 30
*(ptr + 2) = 30
```

Mảng trong C/C++

- Trong C luôn phải chỉ ra kích thước tối đa khi khai báo mảng, vậy có cách nào khác phục khi
  - Không biết trước số lượng phần tử tối đa
  - Muốn tối ưu bộ nhớ, tránh lãng phí (các phần tử khai báo mà không dùng đến)
- Mảng cấp phát động nhiều lần(mảng với kích thước biến đổi)
  - Hàm cấp phát động trong C: malloc, calloc, relloc, và free
  - Ban đầu cấp phát 1 mảng nhỏ (VD. MAX_SIZE = 10 phần tử)
  - Tùy theo nhu cầu, nếu cần chưa phần tử > kích thước tối đa hiện tại → tạo mảng mới với kích thước gấp đôi mảng cũ (VD. MAX_SIZE = 2 * MAX_SIZE). Copy các phần tử mảng cũ vào nửa đầu mảng mới.
  - Nếu số lượng phần tử thực sự trong mảng < ½ MAX_SIZE, tiến hành điều chỉnh co mảng với kích thước mảng mới MAX_SIZE = ½ MAX_SIZE để tránh lãng phí bộ nhớ
  - Hệ số co giãn mảng – Load Factor thường chọn là 0.75, 1 tùy NNLT

Mảng động với kích thước biến đổi

- Hệ số nạp $\lambda = \frac{n}{MAX\_SIZE}$
- Từ mảng 1 phần tử tới n phần tử, số lần phải thay đổi kích thước là $\log_2 n$
- Số phần tử phải di chuyển

$$M = \sum_{i=1}^{\log n} i * \frac{n}{2^i} = n * \sum_{i=1}^{\log n} \frac{i}{2^i} < n * \sum_{i=1}^{\infty} \frac{i}{2^i} = 2n$$

Thời gian để duy trì mảng chỉ là $O(n)$

- **Nhược điểm**: một số thời gian thực hiện một số thao tác không còn đúng là hằng số nữa

Mảng trong Java

- Mảng trong Java
  - Luôn được cấp phát động
  - Là kiểu object nên có sẵn 1 số hàm hỗ trợ. VD. length
  - Kích thước mảng bị giới hạn bởi giá trị int hoặc short int (<4G)
  - Kiểu phần tử có thể là kiểu cơ sở hoặc object
  - Trong JAVA luôn có check truy cập ngoài phạm vi của mảng

```
int intArray[];    //declaring array
intArray = new int[20];  // allocating memory to array
```

OR

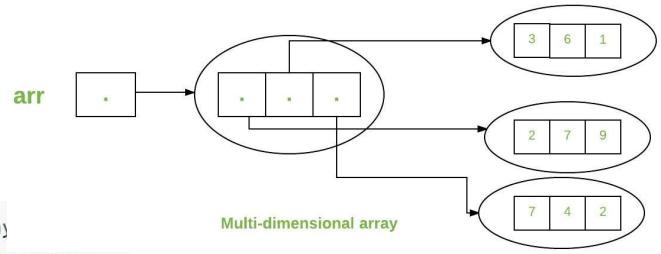
```
int[] intArray = new int[20]; // combining both statements in one
```

```
// both are valid declarations
int intArray[];
or int[] intArray;
```

```
public static void main (String[] args)
{
    int[] arr = new int[2];
    arr[0] = 10;
    arr[1] = 20;

    for (int i = 0; i <= arr.length; i++)
        System.out.println(arr[i]);
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
```

Mảng trong Java

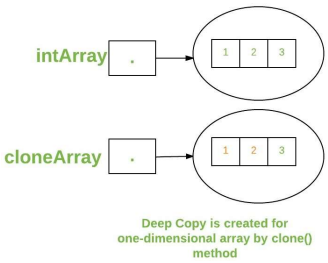


- Mảng nhiều chiều

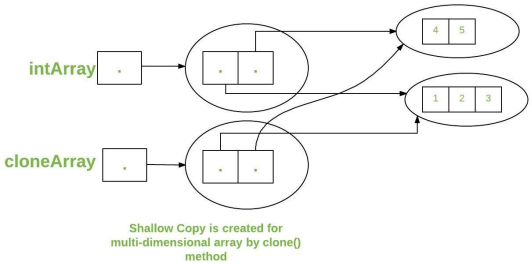
```
int[][] intArray = new int[10][20]; //a 2D array
int[][][] intArray = new int[10][20][10]; //a 3D array
```

Clone Array: Deep copy VS Shallow Copy

```
int intArray[] = {1,2,3};

int cloneArray[] = intArray.clone();
```

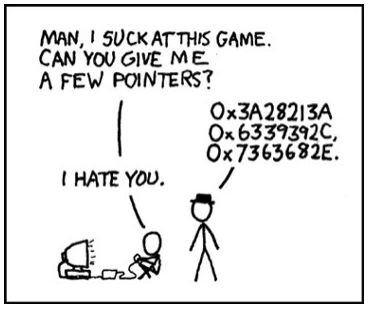



```
int intArray[][] = {{1,2,3},{4,5}};

int cloneArray[][] = intArray.clone();
```



Mảng - Array

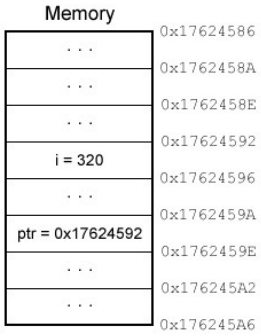
- VD 1. Viết chương trình xoay các phần tử của mảng đi k vị trí
- VD 2. Tìm phần tử trong mảng A có giá trị i*A[i] là lớn nhất
- VD 3. Viết chương trình sắp xếp lại mảng sao cho các phần tử âm ở đầu dãy và phần tử dương ở cuối dãy (không cần đúng thứ tự)
- VD 4. Cho 1 xâu chỉ chứa các ký tự là chữ số, hãy hoán đổi các ký tự trong xâu sao cho thu được biểu diễn của số có giá trị lớn nhất
- VD 5. Hãy viết chương trình xáo trộn mảng theo thứ tự ngẫu nhiên
- VD 6. Cho mảng A chứa n số nguyên, hãy tìm và in ra trung vị (median) của mảng
- VD 7. Cho mảng số thực A chứa n phần tử, tìm 2 số có tổng nhỏ nhất
- VD 8. Cho dãy chứa n phần tử, tìm dãy con độ dài k có giá trị trung bình nhỏ nhất
- VD 9. cho mảng n phần tử phân biệt và giá trị k, tìm xem có 2 phần tử trong mảng tổng bằng k



Cấu trúc liên kết

- Con trỏ và cấu trúc liên kết
- Danh sách liên kết đơn
- Các dạng khác của danh sách liên kết

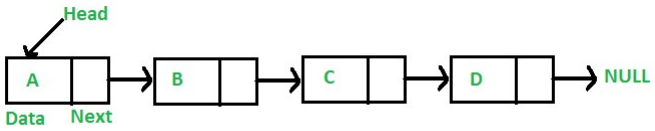
Con trỏ và cấu trúc liên kết




```
int i = 320;
int* ptr = &i;
```

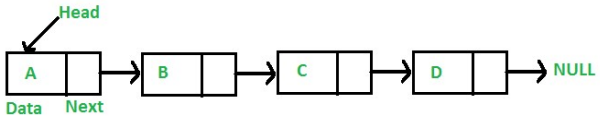
- **Con trỏ** lưu trữ địa chỉ của một vị trí trong bộ nhớ.
  VD. Visiting card có thể xem như con trỏ trỏ đến nơi làm việc của một người nào đó.
- Trong cấu trúc liên kết con trỏ được dùng để liên kết giữa các phần tử.
- Trong C/C++ :
  - ***ptr** chỉ **ptr** là một biến con trỏ
  - **&i** chỉ địa chỉ của biến **i** trong bộ nhớ (địa chỉ ô nhớ đầu tiên)
  - Con trỏ khi mới khởi tạo nhận giá trị **NULL** - con trỏ chưa được gán giá trị (không trỏ vào đâu cả)
  - Kiểu của con trỏ để xác định phạm vi bộ nhớ có thể truy cập
  - Các con trỏ có cùng kích thước trên 1 platform

Cấu trúc liên kết



- Cấu trúc liên kết
  - Các phần tử nằm rải rác trong bộ nhớ
  - Phần tử trước sẽ lưu lại địa chỉ phần tử tiếp theo (qua con trỏ)
  - Các phần tử chỉ có thể truy cập một cách tuần tự
  - Việc thêm/xóa phần tử đơn giản hơn so với cấu trúc liên tiếp (mảng)
  - Mỗi phần tử cần thêm ít nhất 1 con trỏ để duy trì liên kết trong cấu trúc (con trỏ được coi là bộ nhớ lãng phí dưới góc độ người dùng)
- Một số đại diện cấu trúc liên kết
  - Danh sách liên kết: danh sách liên kết đơn, danh sách liên kết đôi,…
  - Cây: cây nhị phân tổng quát, cây AVL, R-B tree, kD tree, prefix tree…
  - Đồ thị lưu bằng ma trận kề

Danh sách liên kết đơn



- Danh sách liên kết đơn
  - Là cấu trúc liên kết đơn giản nhất
  - Mỗi phần tử chỉ có thêm 1 con trỏ để lưu địa chỉ phần tử kế tiếp
- Ưu điểm so với mảng
  - Không cần khai báo trước số lượng tối đa
  - Dùng bao nhiêu, cấp phát đủ
  - Thêm/xóa các phần tử dễ dàng, không cần dịch (chỉ cần thay đổi giá trị con trỏ)
- Nhược điểm
  - Chỉ có thể truy cập phần tử một cách tuần tự
  - Mỗi phần tử tốn thêm 1 con trỏ

Cấu trúc liên kết – linked list

- Khai báo danh sách liên kết đơn (singly-linked list) :
  - Có 1 hay nhiều trường dữ liệu (item) chứa dữ liệu cần lưu trữ
  - Có ít nhất 1 con trỏ trỏ đến nút tiếp theo (next) → cần nhiều bộ nhớ hơn cấu trúc liên tục.
  - Cần 1 con trỏ lưu địa chỉ phần tử bắt đầu của cấu trúc.

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

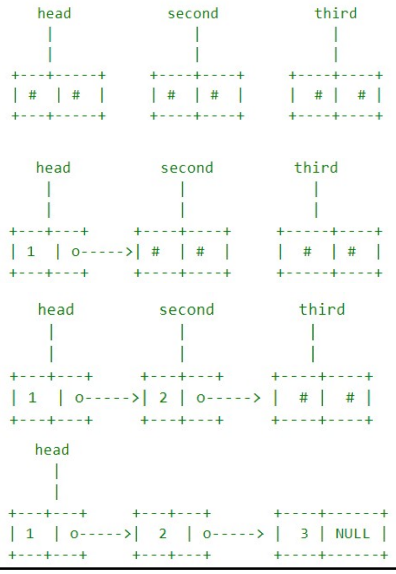
```
class Node {
public:
    int data;
    Node* next;
};
```

```
static class Node {
    int data;
    Node next;
    Node(int d)
    {
        data = d;
        next = null;
    } // Constructor
}
```

https://www.geeksforgeeks.org/linked-list-set-1-introduction/

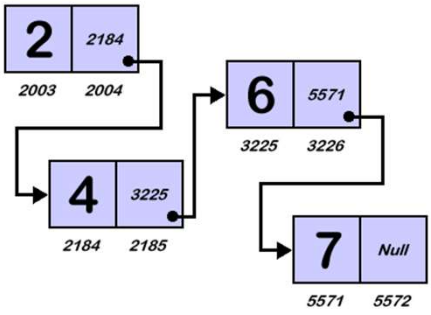
Danh sách liên kết đơn - Singly-linked list

```
struct Node* head = NULL;
struct Node* second = NULL;
struct Node* third = NULL;

// allocate 3 nodes in the heap
head = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third = (struct Node*)malloc(sizeof(struct Node));

head->data = 1; // assign data in first node
head->next = second; // Link first node with

second->data = 2;
second->next = third;

third->data = 3; // assign data to third node
third->next = NULL;
```



Danh sách liên kết đơn

- Một số thao tác thông dụng trên danh sách liên kết đơn
  - Duyệt danh sách (in danh sách, đếm số phần tử)
  - Chèn một phần tử mới
  - Xóa một phần tử
  - Tìm kiếm một phần tử



Danh sách liên kết đơn

- Duyệt danh sách

```
void printList(struct Node* n)
{
    while (n != NULL) {
        printf(" %d ", n->data);
        n = n->next;
    }
}
```

```
int size(Node* n)
{
    int count=0;
    while (n != NULL) {
        count++;
        n = n->next;
    }
    return count;
}
```

```
class LinkedList {

    Node head; // head of list
    static class Node {
        int data;
        Node next;
        Node(int d)
        {
            this.data = d;
            next = null;
        } // Constructor
    }
    public void printList()
    {
        Node n = head;
        while (n != null) {
            System.out.print(n.data + " ");
            n = n.next;
        }
    }
    public static void main(String[] args)
    {
        /* Start with the empty list. */
        LinkedList llist = new LinkedList();


        llist.head = new Node(1);
        Node second = new Node(2);
        Node third = new Node(3);


        llist.head.next = second; // Link first node with the second node
        second.next = third; // Link second node with the third node


        llist.printList();
    }
}
```

Duyệt danh sách

- Tìm kiếm
  - Tìm xem khóa key có xuất hiện trong danh sách
  - Đếm số lượng phần tử có giá trị bằng giá trị cho trước

$$T(n) = O(n)$$

Bài tập thêm
- Tìm phần tử ngay trước phần tử hiện tại?
- Tìm phần tử ở vị trí thứ k trong dãy
- Tìm phần tử ở giữa danh sách
- Đếm số lần xuất hiện của giá trị key

```
/* Checks whether the value x is present in linked list */
bool search(struct Node* head, int key)
{
    struct Node* current = head;  // Initialize current
    while (current != NULL)
    {
        if (current->key == key)
            return true;
        current = current->next;
    }
    return false;
}
```

```
/* Checks whether the value x is present in linked list */
struct Node* search(struct Node* head, int key)
{
    struct Node* current = head;  // Initialize current
    while (current != NULL)
    {
        if (current->key == key)
            return current;
        current = current->next;
    }
    return NULL;
}
```

Danh sách liên kết đơn

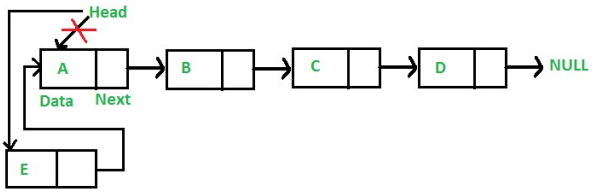
- Thêm một phần tử mới
  - Thêm vào đầu tiên - push
  - Thêm vào giữa (sau 1 vị trí nào đó) – insertAfter
  - Thêm vào cuối - append

Dùng khai báo minh họa 1 nút của DSLK đơn chỉ với 1 trường kiểu int

Trong thực tế KHÔNG nên khai báo DSLK đơn nếu dữ liệu lưu trữ chỉ là 1 trường int, TẠI SAO?

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

Thêm phần tử mới



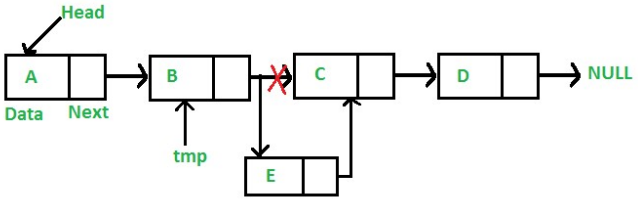
- **Thêm vào đầu tiên – push**
  - Danh sách ban đầu đang có A→B→C→D, chèn thêm phần tử E vào đầu để được danh sách mới E→A→B→C→D

Các bước cần làm gồm
- Cấp phát động lưu trữ phần tử mới
- Gán giá trị phần tử mới
- Cập nhật vị trí phần tử mới

Hàm push sẽ làm thay đổi giá trị con trỏ đầu danh sách , vì vậy trong hàm cần truyền vào là **head_ref

```
void push(struct Node** head_ref, int new_data)
{
    /* 1. cấp phát động nút mới */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. Cập nhật dữ liệu nút mới  */
    new_node->data = new_data;

    /* 3. Biến nút mới thành đầu của dãy hiện tại*/
    new_node->next = (*head_ref);

    /* 4. Cập nhật lại giá trị của con trỏ đầu dãy */
    (*head_ref) = new_node;
}
```

Thêm phần tử mới



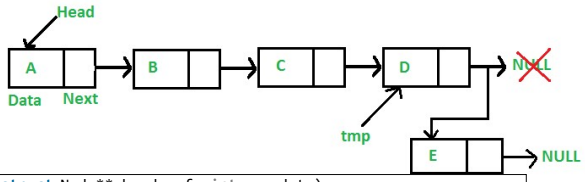
- **Thêm vào sau 1 nút cho trước – insertAfter**
  - Danh sách ban đầu A→B→C→D, Chèn E vào sau phần tử B để được A→B→E→C→D
  - Cần thêm con trỏ trỏ vào vị trí trước chèn prev_node
  - Trong hàm KHÔNG làm thay đổi giá trị của con trỏ prev_node nên không cần **

```
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. kiểm tra vị trí chèn prev_node có phải NULL */
    if (prev_node == NULL) return;

    /* 2. Cấp phát động nút mới */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));


    /* 3. Gán giá trị */
    new_node->data = new_data;


    /* 4. Cập nhật phần tử kế tiếp của phần tử mới */
    new_node->next = prev_node->next;


    /* 5. Gắn phần tử mới vào sau prev_node */
    prev_node->next = new_node;
}
```

Thêm phần tử mới

- **Thêm vào cuối dãy – append**
  - Dãy đang là A→B→C→D, chèn thêm E vào cuối dãy để được dãy mới A→B→C→D→E
  - Nếu dãy rỗng → thêm vào đầu = cuối
  - Ngược lại, phải duyệt tuần tự tìm cuối dãy
  - Nếu dãy rỗng ta sẽ phải cập nhật lại con trỏ đầu dãy → cần truyền vào **

$$T(n) = ?$$




```
void append(struct Node** head_ref, int new_data)
{
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    struct Node *last = *head_ref;

    new_node->data  = new_data;

    /* Nút mới là cuối dãy nên next sẽ trỏ vào NULL*/
    new_node->next = NULL;

    /* Nếu danh sách đang rỗng → cập nhật lại đầu */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* Ngược lại, tìm cuối dãy*/
    while (last->next != NULL)
        last = last->next;

    /* Gắn nút mới là cuối dãy*/
    last->next = new_node;
    return;
}
```

Thêm phần tử mới

- Thử nghiệm các thao tác thêm
  - Printf và cout với C/C++

```
void printList(struct Node *node)
{
  while (node != NULL)
  {
     printf(" %d ", node->data);
     node = node->next;
  }
}
```

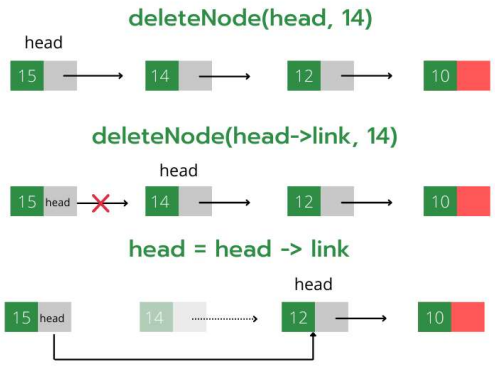
```
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    // Insert 6. So linked list becomes 6->NULL
    append(&head, 6);

    // Insert 7 at the beginning.
    // So linked list becomes 7->6->NULL
    push(&head, 7);

    // Insert 1 at the beginning.
    // So linked list becomes 1->7->6->NULL
    push(&head, 1);

    // Insert 4 at the end. So
    // linked list becomes 1->7->6->4->NULL
    append(&head, 4);

    // Insert 8, after 7. So linked
    // list becomes 1->7->8->6->4->NULL
    insertAfter(head->next, 8);

    cout<<"Created Linked list is: ";
    printList(head);

    return 0;
}
```

Danh sách liên kết đơn

- Xóa phần tử - delete
  - Xóa phần tử ở đầu
  - Xóa phần tử ở vị trí cho trước
  - Xóa phần tử ở cuối dãy
  - Xóa phần tử có khóa bằng key
  - Xóa toàn bộ dãy → giải phóng bộ nhớ

Xóa phần tử

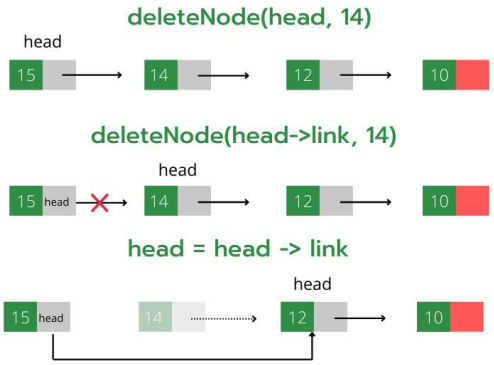
- Xóa phần tử ở đầu - removeFirst
  - Cập nhật lại con trỏ head
  - Gọi lệnh giải phóng bộ nhớ




```
void removeFirst(struct Node** head_ref)
{
    /* 1. Danh sách ban đầu rỗng thì không làm gì cả*/
    if(*head_ref==NULL) return;

    /* 2. Ghi nhận địa chỉ nút đầu cũ */
    struct Node* tmp = *head_ref;

    /* 3. Cập nhật lại đầu danh sách trỏ sang phần tử tiếp */
    *head_ref = (*head_ref)->next;

    /* 4. Giải phóng phần tử đầu cũ */
    free(tmp);
}
```

Xóa phần tử

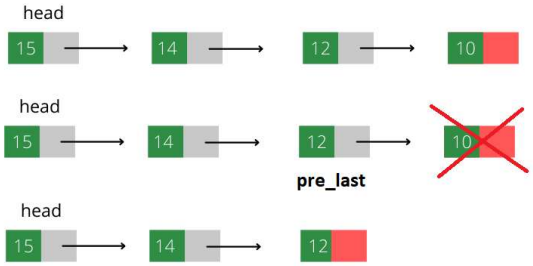
- Lấy phần tử ở đầu - pop
  - Cập nhật lại con trỏ head
  - Trả về phần tử




```
struct Node* pop(struct Node** head_ref)
{
    /* 1. Danh sách ban đầu rỗng thì không làm gì cả*/
    if(*head_ref==NULL) return NULL;

    /* 2. Ghi nhận địa chỉ nút đầu cũ */
    struct Node* tmp = *head_ref;

    /* 3. Cập nhật lại đầu danh sách trỏ sang phần tử tiếp */
    *head_ref = (*head_ref)->next;

    /* 4. Trả về*/
    return tmp;
}
```

Xóa phần tử

- **Xóa phần tử ở cuối**
  - Nếu danh sách rỗng → không làm gì cả
  - Danh sách có 1 phần tử ->Xóa đầu
  - Ngược lại → phải tìm phần tử gần cuối – pre_last
  - Xóa phần tử cuối từ pre-last, và cập nhật lại phần tử cuối

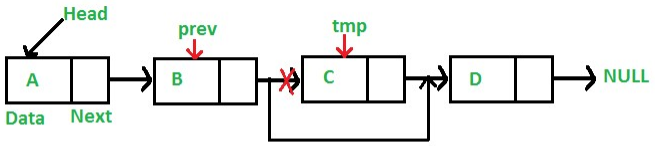



```
void removeLast(struct Node** head_ref)
{
  /* 1. Danh sách ban đầu rỗng thì không làm gì cả*/
  if(*head_ref==NULL) return;

  /* 2. Danh sách chỉ có 1 phần tử*/
  if((*head_ref)->next==NULL)
  {
    free(*head_ref);
    *head_ref = NULL;
    return;
  }
  /* 3. Tìm phần tử trước phần tử cuối cùng */
  struct Node* pre_last = *head_ref;
  while(pre_last->next->next !=NULL)
    pre_last = pre_last->next;

  /* 4. Xóa phần tử cuối cùng */
  free(pre_last->next);

  /* 5. Cập nhật lại con trỏ next của phần tử cuối dãy mới */
  pre_last->next = NULL;
}
```

Tại sao cần ** head_ref

$T(n) = ?$

Xóa phần tử

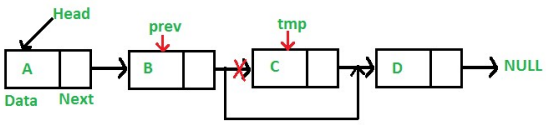


- Xóa phần tử ở vị trí cho trước
  - Phần tử trước vị trí cần xóa trỏ bởi con trỏ prev
  - Không rơi vào trường hợp đầu hoặc cuối
  - Phần tử cần xóa trỏ bởi tmp
- Nếu phần tử cần xóa là
  - Đầu/Cuối thì xử lý thế nào?
  - Xóa phần tử có giá trị bằng khóa K
  - Xóa toàn bộ dãy

```
void deleteNode(struct Node* prev)
{
   struct Node * temp = prev->next;

   // Unlink the node from linked list
   prev->next = temp->next;

   free(temp); // Free memory
}
```

Xóa phần tử

- Xóa phần tử có giá trị bằng khóa key
  - Key có thể là đầu
  - Có thể là giữa/ cuối
  - Luôn phải free bộ nhớ



Tại sao cần **head_ref ???

```
void deleteNode(struct Node** head_ref, int key)
{
   // Luu lai dia chi phan tu dau
   struct Node * temp = *head_ref, * prev;

   // phan tu can xoa chinh la dau day
   if (temp != NULL && temp->data == key) {
      * head_ref = temp->next; // Changed head
      free(temp); // free old head
      return;
   }

   // Tìm phan tu khoa key, luu lai phan tu truoc
   // phan tu can xoa la tmp, se la 'prev->next'
   while (temp != NULL && temp->data != key) {
      prev = temp;
      temp = temp->next;
   }

   // Neu danh sach không co phan tu bang key
   if (temp == NULL)
      return;

   // Xoa phan tu temp
   prev->next = temp->next;

   free(temp); // Free memory
}
```

Xóa phần tử

- Test code

```
void printList(struct Node *node)
{
  while (node != NULL)
  {
     printf(" %d ", node->data);
     node = node->next;
  }
}
```

```
int main()
{
   node* head = NULL;
   // tao day 15→14→12→10
   push(head, 10);
   push(head, 12);
   push(head, 14);
   push(head, 15);

   // original list, in ra 15,14,12,10
   print(head);

   pop(&head); // xoa phan tu dau tien
   print(head); // in ra 14,12,10

   deleteNode(head, 10);
   print(head); // in ra 14,12

   removeLast(&head);
   print(head); //in ra 14

   return 0;
}
```

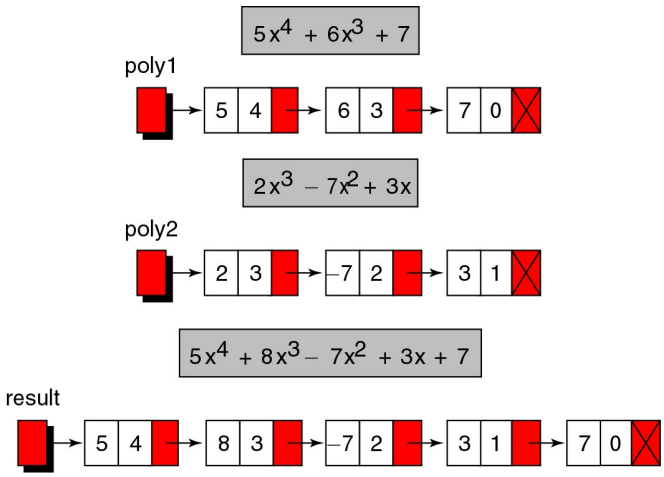
Danh sách liên kết đơn

- Bài tập thêm
  - Xóa toàn bộ dãy (cần giải phóng toàn bộ phần tử)
  - Xóa toàn bộ phần tử có khóa bằng key trong dãy
  - Xóa phần tử tại vị trí thứ i trong dãy (phần tử đầu tiên là vị trí 0)
  - Phát hiện vòng trong danh sách liên kết đơn
  - Loại bỏ các phần tử bị lặp trong danh sách liên kết đơn
  - Tìm giao của 2 danh sách liên kết đơn
  - Hoán đổi 2 phần tử ở vị trí i và j trong danh sách
  - Đảo ngược danh sách liên kết
  - Kiểm tra danh sách có phải đối xứng – palindrome
  - Copy danh sách liên kết đơn
  - Xóa phần tử hiện tại mà KHÔNG có địa chỉ phần tử đầu danh sách

https://www.geeksforgeeks.org/data-structures/linked-list

Danh sách liên kết đơn

- Bài toán biểu diễn đa thức



Danh sách liên kết đơn

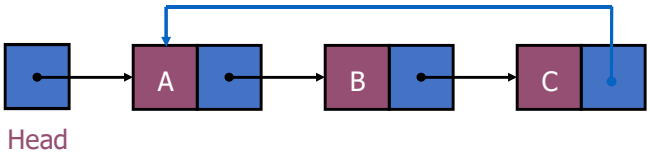
- **typedef struct poly{**
  float heSo;
  float soMu;
  struct poly *nextNode;
  } POLY;

- **Các thao tác:**
  - Nhập đa thức
  - Hiển thị
  - Cộng
  - Trừ
  - Nhân
  - Tính giá trị đa thức
  - Chia
  - ….

Danh sách liên kết

Một số dạng mở rộng khác của danh sách liên kết

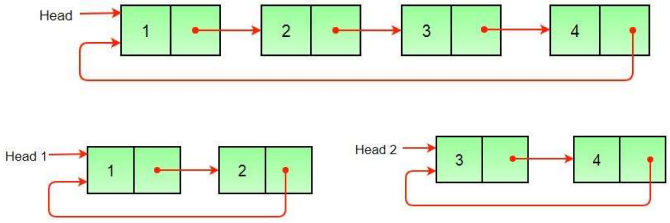
- **Danh sách liên kết đơn nối vòng**: Con trỏ của phần tử cuối trỏ về đầu danh sách



- Tác dụng:
  - có thể quay lại từ đầu khi đã ở cuối dãy
  - Kiểm tra ở cuối dãy : currentNode->next == head ?
  - Kiểm tra đang ở đầu dãy: currentNode == head

Danh sách liên kết đơn nối vòng

```
void printList(Node* head)
{
    Node* temp = head;

    // If linked list is not empty
    if (head != NULL) {

        // Print nodes till we reach first node again
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
    }
}
```




```
void splitList(Node *head, Node **head1_ref,
                           Node **head2_ref)
{
    Node *slow_ptr = head;
    Node *fast_ptr = head;

    if(head == NULL)
        return;

    /* If there are odd nodes in the circular list then
       fast_ptr->next becomes head and for even nodes
       fast_ptr->next->next becomes head */
    while(fast_ptr->next != head &&
          fast_ptr->next->next != head)
    {
        fast_ptr = fast_ptr->next->next;
        slow_ptr = slow_ptr->next;
    }

    /* If there are even elements in list
       then move fast_ptr */
    if(fast_ptr->next->next == head)
        fast_ptr = fast_ptr->next;

    /* Set the head pointer of first half */
    *head1_ref = head;

    /* Set the head pointer of second half */
    if(head->next != head)
        *head2_ref = slow_ptr->next;

    /* Make second half circular */
    fast_ptr->next = slow_ptr->next;

    /* Make first half circular */
    slow_ptr->next = head;
}
```

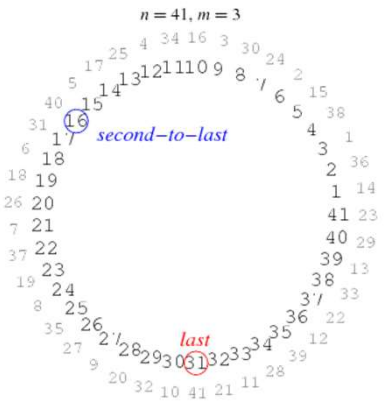
Danh sách liên kết đơn nối vòng

- Một số bài toán
  - Thêm phần tử vào danh sách liên kết đơn nối vòng
  - Xóa phần tử
  - Tìm phần tử giữa danh sách
  - Đếm số lượng phần tử trong danh sách
  - Bài toán vòng tròn Josephus
  - Chuyển danh sách liên kết đơn thường sang dạng nối vòng

Ứng dụng

Ví dụ. Bài toán Josephus

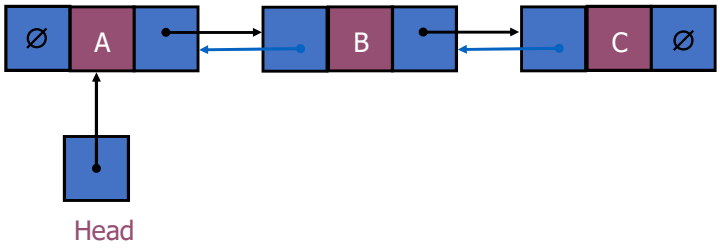
- Có một nhóm gồm *n* người được xếp theo một vòng tròn. Từ một vị trí bất kỳ đếm theo chiều ngược chiều kim đồng hồ và loại ra người thứ *m* trong vòng. Sau mỗi lần loại lại bắt đầu đếm lại vào loại tiếp cho đến khi chỉ còn lại 1 người duy nhất.



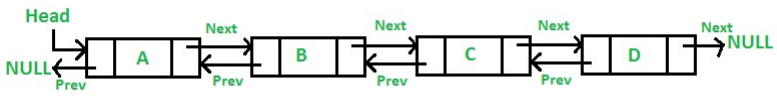
- **Cài đặt** : sử dụng danh sách liên kết đơn nối vòng

Danh sách liên kết mở rộng

- **Danh sách liên kết đôi:**
  - Mỗi nút có 2 con trỏ: con trỏ phải trỏ đến phần tử tiếp sau, con trỏ trái trỏ đến phần tử ngay trước.

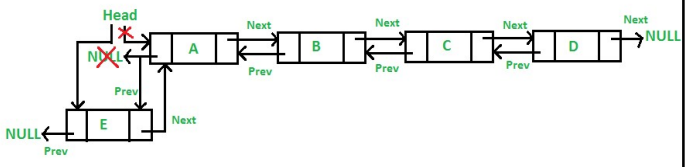


  - **Ưu điểm**: có thể duyệt danh sách theo cả hai chiều, thêm/xóa đơn giản hơn so với DS liên kết đơn
  - Kiểm tra cuối danh sách: con trỏ phải là NULL
  - Đầu danh sách: con trỏ trái là NULL
  - Tốn thêm 1 con trỏ để duy trì danh sách, thêm thao tác xử lý với con trỏ thứ 2



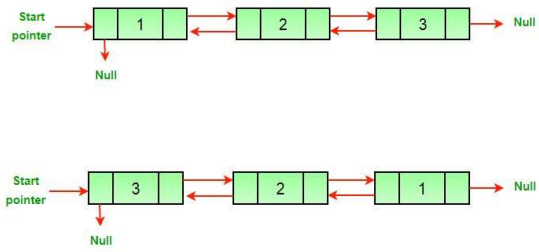

```
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head
    and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```



https://www.geeksforgeeks.org/doubly-linked-list/

Đảo ngược danh sách



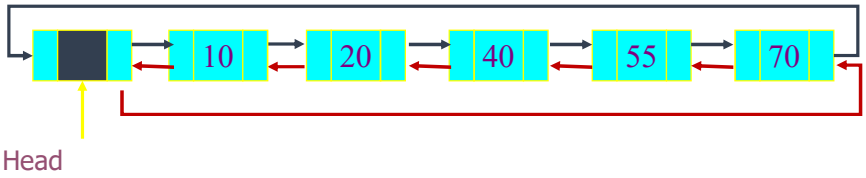

```
void reverse(Node **head_ref)
{
    Node *temp = NULL;
    Node *current = *head_ref;

    /* swap next and prev for all nodes of
    doubly linked list */
    while (current != NULL)
    {
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    /* Before changing the head, check for the cases like empty
        list and list with only one node */
    if(temp != NULL )
        *head_ref = temp->prev;
}
```

Danh sách liên kết đôi

- Một số bài toán
  - Thêm phần tử vào vị trí bất kỳ
  - Xóa phần tử giá trị Key trong danh sách
  - Loại bỏ phần tử trùng lặp
  - Xoay danh sách liên kết đôi
  - Biểu diễn số nguyên lớn với các toán tử +, -, *, / (cỡ hàng nghìn chữ số)
  - Tìm 2 số có tổng bằng giá trị k cho trước

Danh sách liên kết đôi

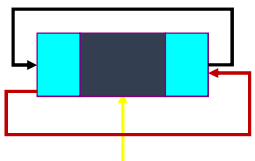
- **Danh sách liên kết đôi nối vòng**
  - Các con trỏ thừa ở cuối trỏ về đầu và con trỏ thừa ở đầu sẽ trỏ xuống cuối
  - Có thể dùng thêm nút giả để phân biệt đầu và cuối
  - Con trỏ pNext của nút cuối trỏ vào nút đầu và con trỏ pPrev của nút đầu trỏ vào nút cuối.
  - Nút đầu danh sách mà nút giả



Head

Danh sách liên kết đôi nối vòng

- **Ưu điểm**: có thể di chuyển theo hai chiều, và từ phần tử cuối (đầu) có thể nhảy ngay đến phần tử đầu (cuối) dãy.
- Kiểm tra danh sách rỗng: pNext, pPrev đều trỏ vào 1 phần tử Head.
- Kiểm tra phần tử cuối dãy: pNext trỏ tới Head
- Không dùng nút giả? Kiểm tra cuối và đầu phức tạp hơn 1 chút



Head

Danh sách rỗng

Cấu trúc dữ liệu – danh sách tuyến tính

- Danh sách tuyến tính – linear list
  - Các phần tử có cùng kiểu, và
  - Tuân theo thứ tự tuyến tính
  - Thao tác thêm/xóa cần giữ thứ tự của các phần tử đúng
- VD. Danh sách ban đầu A, B, C, D
  - Thêm E và vị trí thứ 2: A, E, B, C, D
  - Xóa A: E, B, C, D
  - Thêm G vào vị trí đầu tiên: G, E, B, C, D

Danh sách tuyến tính

- Các thao tác cơ bản của ADT danh sách tuyến tính
  - Thêm phần tử vào vị trí bất kỳ - insertAt
  - Thêm vào đầu: push
  - Thêm vào cuối: append
  - Lấy phần tử khỏi đầu: pop
  - Lấy phần tử ở vị trí bất kỳ: removeAt
  - Xóa toàn bộ danh sách
  - Tìm số lượng phần tử hiện tại
  - Tìm kiếm 1 giá trị có xuất hiện trong dãy
  - Kiểm tra danh sách rỗng
  - ….

Danh sách tuyến tính

- Cài đặt dùng mảng
  - Truy cập phần tử nhanh O(1)
  - Cần biết trước số lượng tối đa (hoặc dùng mảng động kích thước biến đổi)
  - Thêm/xóa phải dịch phần tử
- Cài đặt dùng danh sách liên kết đơn
  - Truy cập phần tử O(n)
  - Không cần biết trước số lượng tối đa
  - Không phải dịch phần tử khi thêm/xóa
  - Mỗi phần tử tốn thêm bộ nhớ phụ lưu trữ con trỏ

Ngăn xếp – Stack