ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

CƠ SỞ LẬP TRÌNH

Bài thực hành 08

LINKED LIST - STACK - QUEUE

Giảng viên thực hành: Lê Đức Khoan

Thành phố Hồ Chí Minh, 12/2025

Mục lục

1 Danh sách liên kết đơn

1.1 Khái niệm

Danh sách liên kết đơn (Singly Linked List) là một cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử (gọi là **node**) bao gồm:

- **Dữ liệu** (data)
- **Con trỏ** trỏ đến node kế tiếp trong danh sách

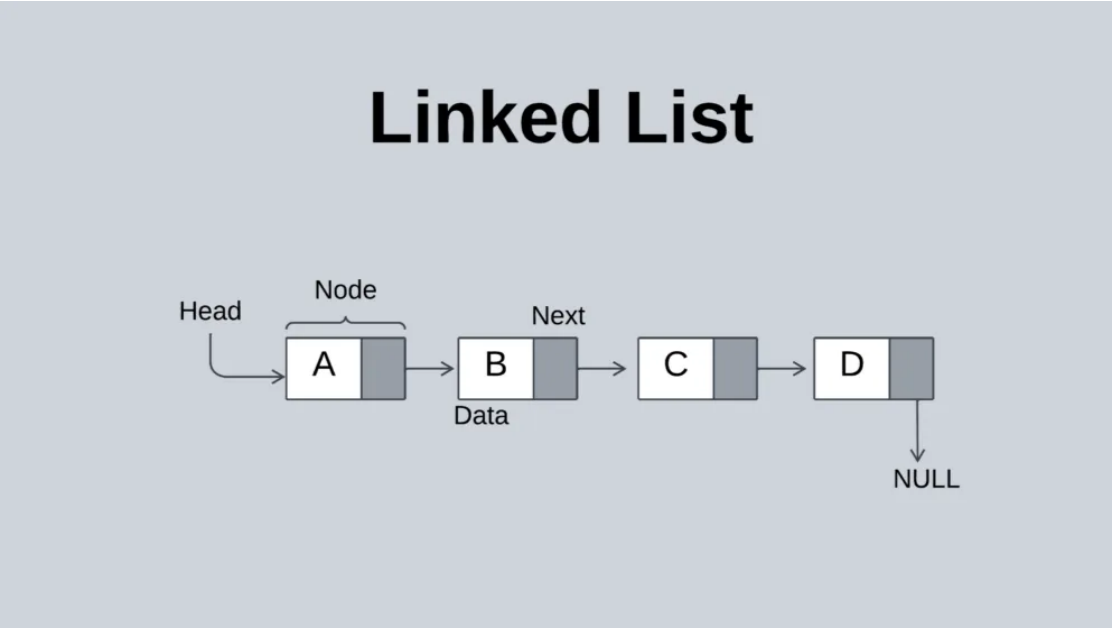

Figure 1: Danh sách liên kết đơn

Các node không nằm liên tiếp trong bộ nhớ như mảng, mà được liên kết với nhau thông qua các con trỏ.

1.2 Cấu trúc node

Mỗi node trong danh sách liên kết đơn thường được định nghĩa như sau:

```
1    struct Node {
2        int key;        // Dữ liệu
3        Node *pNext;    // Con trỏ đến node kế tiếp
4    };
```

Node cuối cùng trong danh sách có con trỏ `pNext = nullptr`.

1.3 Cấu trúc danh sách

Để quản lý toàn bộ danh sách, ta thường sử dụng một struct List gồm:

- `pHead`: trỏ đến node đầu tiên
- `pTail`: trỏ đến node cuối cùng

```
struct List {
    Node *pHead;
    Node *pTail;
};
```

1.4 Đặc điểm của danh sách liên kết đơn

- Kích thước danh sách có thể thay đổi linh hoạt trong quá trình chạy chương trình
- Dễ dàng chèn và xóa phần tử mà không cần dời các phần tử khác
- Không hỗ trợ truy cập ngẫu nhiên (không thể truy cập trực tiếp phần tử thứ i)

1.5 So sánh với mảng

	Mảng	**Danh sách liên kết**
Bộ nhớ	Liên tiếp	Không liên tiếp
Truy cập	Nhanh (O(1))	Chậm (O(n))
Chèn/Xóa	Tốn chi phí	Dễ dàng
Kích thước	Cố định	Linh hoạt

1.6 Các thao tác cơ bản

Danh sách liên kết đơn thường hỗ trợ các thao tác sau:

- Tạo node mới
- Chèn node vào đầu, cuối hoặc vị trí bất kỳ

- Xóa node
- Duyệt danh sách
- Đếm số lượng phần tử

Mã nguồn minh hoạ:

```
#include <iostream>
using namespace std;

struct Node {
    int key;
    Node *pNext;
};

struct List {
    Node *pHead;
    Node *pTail;
};

// Tạo node mới
Node* createNode(int data) {
    Node *p = new Node;
    p->key = data;
    p->pNext = nullptr;
    return p;
}

// Khởi tạo danh sách rỗng
void initList(List &L) {
    L.pHead = L.pTail = nullptr;
}

// Chèn node vào đầu danh sách
```

```
28  void addHead(List &L, int data) {
29      Node *p = createNode(data);
30      if (L.pHead == nullptr) {
31          L.pHead = L.pTail = p;
32      } else {
33          p->pNext = L.pHead;
34          L.pHead = p;
35      }
36  }
37
38  // Chèn node vào cuối danh sách
39  void addTail(List &L, int data) {
40      Node *p = createNode(data);
41      if (L.pTail == nullptr) {
42          L.pHead = L.pTail = p;
43      } else {
44          L.pTail->pNext = p;
45          L.pTail = p;
46      }
47  }
48
49  // Duyệt và in danh sách
50  void printList(List L) {
51      for (Node *p = L.pHead; p != nullptr; p = p->pNext) {
52          cout << p->key << " ";
53      }
54      cout << endl;
55  }
56
57  // Đếm số lượng phần tử
58  int countElements(List L) {
59      int count = 0;
60      for (Node *p = L.pHead; p != nullptr; p = p->pNext) {
```

```
61              count++;
62          }
63          return count;
64      }
```

2 Các lỗi thường gặp khi làm việc với danh sách liên kết đơn

2.1 Quên cấp phát bộ nhớ cho node mới

Lỗi:

```
1       Node *p;
2       p->key = 10;    // Lỗi: p chưa được cấp phát
```

Cách khắc phục:

```
1       Node *p = new Node;
2       p->key = 10;
```

2.2 Không kiểm tra danh sách rỗng

Truy cập `pHead` khi danh sách rỗng sẽ gây lỗi runtime.

Cách khắc phục: Luôn kiểm tra:

```
1       if (L == NULL || L->pHead == NULL) return;
```

2.3 Cập nhật con trỏ sai thứ tự

Cập nhật sai thứ tự con trỏ khi chèn node có thể làm mất danh sách.

Ví dụ sai:

```
1       L->pHead = newNode;
2       newNode->pNext = oldHead;
```

Ví dụ đúng:

```
1       newNode->pNext = oldHead;
2       L->pHead = newNode;
```

2.4 Quên giải phóng bộ nhớ

Trong danh sách liên kết, các node thường được cấp phát động bằng toán tử `new`. Nếu khi xóa node mà không sử dụng `delete` để thu hồi bộ nhớ, chương trình sẽ bị **rò rỉ bộ nhớ (memory leak)**.

Ví dụ sai (gây rò rỉ bộ nhớ)

```
1    void removeHead(List &L) {
2        if (L.pHead == nullptr) return;
3
4        Node *temp = L.pHead;
5        L.pHead = L.pHead->pNext;
6
7        // Quên delete temp
8    }
```

Trong đoạn code trên, node đầu danh sách không còn được sử dụng nhưng vùng nhớ của nó vẫn tồn tại trong heap, dẫn đến rò rỉ bộ nhớ.

Ví dụ đúng (giải phóng bộ nhớ).

```
1    void removeHead(List &L) {
2        if (L.pHead == nullptr) return;
3
4        Node *temp = L.pHead;
5        L.pHead = L.pHead->pNext;
6
7        delete temp; // Giải phóng bộ nhớ
8    }
```

Việc gọi `delete` đảm bảo vùng nhớ được thu hồi, giúp chương trình hoạt động ổn định và tránh lãng phí tài nguyên.

2.5 Truy cập con trỏ NULL

Ví dụ:

```
1    p = p->pNext->pNext; // Lỗi nếu pNext là NULL
```

Cách khắc phục: Luôn kiểm tra điều kiện trước khi truy cập.

3 Danh sách liên kết đôi

3.1 Khái niệm

Danh sách liên kết đôi (Doubly Linked List) là một cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử (node) chứa:

- Dữ liệu (data)
- Con trỏ trỏ đến node kế tiếp
- Con trỏ trỏ đến node đứng trước

Nhờ có hai con trỏ, danh sách liên kết đôi cho phép duyệt danh sách theo cả hai chiều: từ đầu đến cuối và từ cuối về đầu.

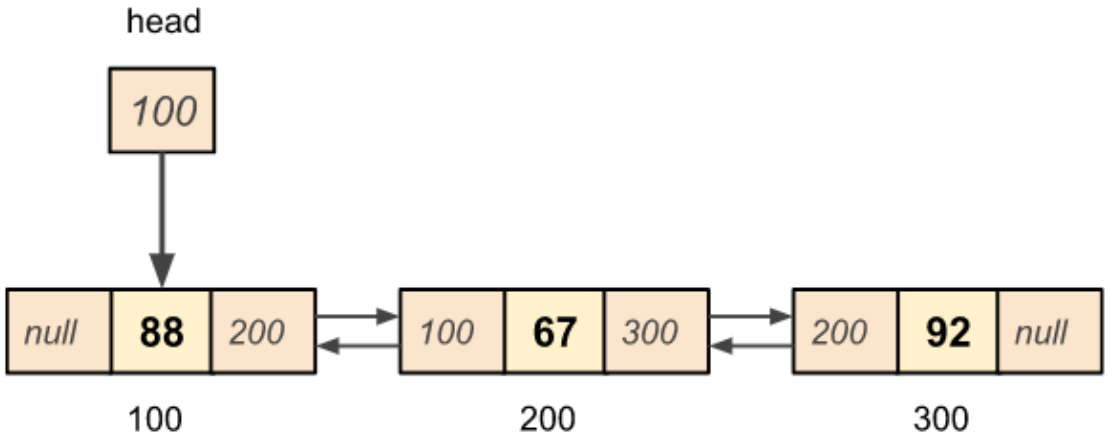

Figure 2: Danh sách liên kết đôi

3.2 Cấu trúc node

Một node trong danh sách liên kết đôi thường được định nghĩa như sau:

```
struct DNode {
    int key;          // Dữ liệu
    DNode *pNext;     // Con trỏ đến node kế tiếp
    DNode *pPrev;     // Con trỏ đến node đứng trước
};
```

3.3 Cấu trúc danh sách

Để quản lý danh sách liên kết đôi, ta sử dụng struct:

```
struct DList {
    DNode *pHead;      // Node đầu danh sách
    DNode *pTail;      // Node cuối danh sách
};
```

3.4 Đặc điểm của danh sách liên kết đôi

- Có thể duyệt danh sách theo cả hai chiều
- Chèn và xóa node ở giữa danh sách thuận tiện hơn DSLK đơn
- Tốn thêm bộ nhớ do phải lưu thêm con trỏ `pPrev`
- Việc cài đặt phức tạp hơn so với DSLK đơn

3.5 So sánh DSLK đơn và DSLK đôi

	DSLK đơn	**DSLK đôi**
Số con trỏ / node	1	2
Duyệt ngược	Không	Có
Xóa node giữa	Khó hơn	Dễ hơn
Bộ nhớ	Ít hơn	Nhiều hơn
Độ phức tạp cài đặt	Đơn giản	Phức tạp hơn

3.6 Các thao tác cơ bản

Danh sách liên kết đôi thường hỗ trợ các thao tác sau:

- Tạo node mới
- Chèn node vào đầu, cuối hoặc vị trí bất kỳ
- Xóa node
- Duyệt danh sách từ đầu hoặc từ cuối

Mã nguồn minh hoạ:

```cpp
#include <iostream>
using namespace std;

struct DNode {
    int key;
    DNode *pNext;
    DNode *pPrev;
};

struct DList {
    DNode *pHead;
    DNode *pTail;
};

// Tạo node mới
DNode* createNode(int data) {
    DNode *p = new DNode;
    p->key = data;
    p->pNext = nullptr;
    p->pPrev = nullptr;
    return p;
}

// Khởi tạo danh sách rỗng
void initList(DList &L) {
    L.pHead = L.pTail = nullptr;
}

// Chèn node vào đầu danh sách
void addHead(DList &L, int data) {
    DNode *p = createNode(data);
    if (L.pHead == nullptr) {
```

```
33              L.pHead = L.pTail = p;
34          } else {
35              p->pNext = L.pHead;
36              L.pHead->pPrev = p;
37              L.pHead = p;
38          }
39      }
40
41      // Chèn node vào cuối danh sách
42      void addTail(DList &L, int data) {
43          DNode *p = createNode(data);
44          if (L.pTail == nullptr) {
45              L.pHead = L.pTail = p;
46          } else {
47              p->pPrev = L.pTail;
48              L.pTail->pNext = p;
49              L.pTail = p;
50          }
51      }
52
53      // Duyệt danh sách từ đầu
54      void printForward(DList L) {
55          for (DNode *p = L.pHead; p != nullptr; p = p->pNext) {
56              cout << p->key << " ";
57          }
58          cout << endl;
59      }
60
61      // Duyệt danh sách từ cuối
62      void printBackward(DList L) {
63          for (DNode *p = L.pTail; p != nullptr; p = p->pPrev) {
64              cout << p->key << " ";
65          }
```

```
66          cout << endl;
67      }
```

3.7 Nguyên tắc cập nhật con trỏ

Khi chèn hoặc xóa node trong DSLK đôi, cần đảm bảo:

- `pNext` và `pPrev` của các node liên quan được cập nhật đầy đủ
- Không để con trỏ nào trỏ đến vùng nhớ đã bị giải phóng
- Node đầu có `pPrev = NULL`
- Node cuối có `pNext = NULL`

4 Các lỗi thường gặp khi làm việc với danh sách liên kết đôi

4.1 Quên cập nhật con trỏ pPrev hoặc pNext

Sinh viên thường chỉ cập nhật một chiều con trỏ khi chèn hoặc xóa node, dẫn đến danh sách bị hỏng.

4.2 Cập nhật con trỏ sai thứ tự

Ví dụ sai:

```
1     curr->pNext = newNode;
2     newNode->pPrev = curr;
3     newNode->pNext = curr->pNext; // Lỗi logic
```

Nguyên tắc đúng: Luôn lưu lại các con trỏ cũ trước khi thay đổi.

4.3 Truy cập con trỏ NULL

Ví dụ:

```
1     p->pPrev->pNext = p->pNext; // Lỗi nếu pPrev là NULL
```

Cách khắc phục: Luôn kiểm tra:

```
1     if (p->pPrev != NULL)
```

4.4 Quên giải phóng bộ nhớ

Không dùng `delete` khi xóa node gây rò rỉ bộ nhớ.

5 Stack

5.1 Khái niệm

Stack (ngăn xếp) là cấu trúc dữ liệu hoạt động theo nguyên tắc:

LIFO – Last In, First Out

Phần tử được đưa vào sau cùng sẽ là phần tử được lấy ra đầu tiên.

5.2 Minh họa nguyên lý

Stack có thể hình dung như:

- Một chồng sách
- Một ngăn xếp đĩa

Chỉ có thể:

- Thêm phần tử vào **đỉnh Stack**
- Lấy phần tử ra khỏi **đỉnh Stack**

5.3 Các thao tác cơ bản của Stack

Stack thường hỗ trợ các thao tác sau:

- `push`: thêm phần tử vào Stack
- `pop`: lấy và loại bỏ phần tử ở đỉnh Stack
- `top`: xem giá trị phần tử ở đỉnh Stack
- `isEmpty`: kiểm tra Stack rỗng
- `count`: đếm số lượng phần tử

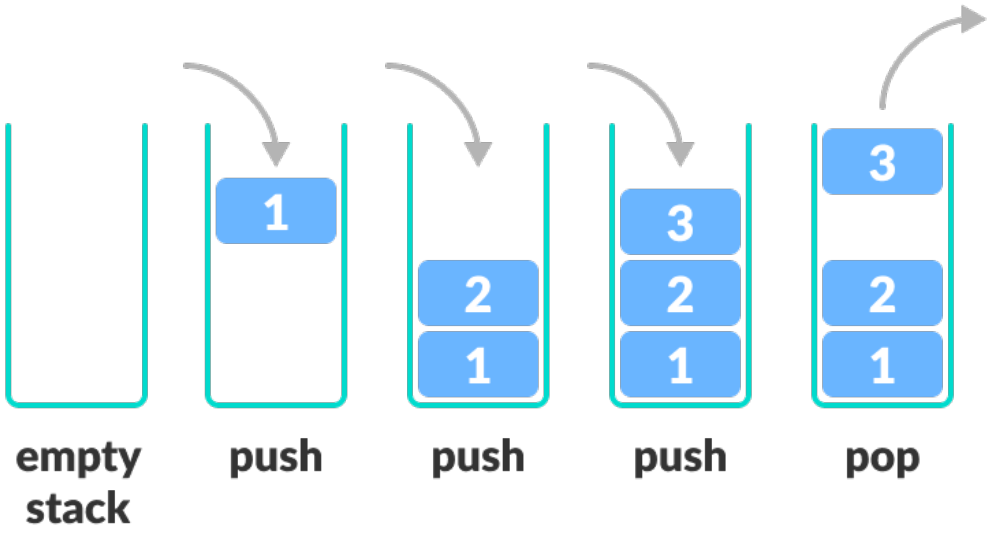

Figure 3: Ngăn xếp

5.4 Ứng dụng của Stack

Stack thường được sử dụng trong:

- Quản lý lời gọi hàm (call stack)
- Kiểm tra dấu ngoặc hợp lệ
- Chuyển đổi và tính toán biểu thức
- Undo / Redo trong phần mềm

6 Queue

6.1 Khái niệm

Queue (hàng đợi) là cấu trúc dữ liệu hoạt động theo nguyên tắc:

FIFO – First In, First Out

Phần tử được đưa vào trước sẽ được lấy ra trước.

6.2 Minh họa nguyên lý

Queue có thể hình dung như:

- Hàng người xếp hàng
- Hàng đợi xử lý tác vụ

Queue có hai đầu:

- **Front**: nơi lấy phần tử ra
- **Rear**: nơi thêm phần tử vào

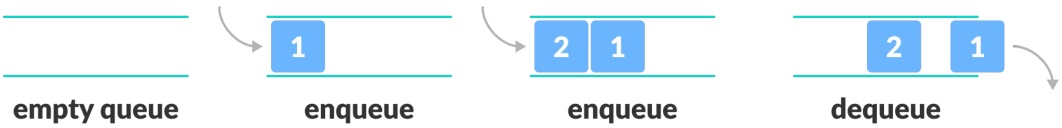

Figure 4: Hàng đợi

6.3 Các thao tác cơ bản của Queue

Queue thường hỗ trợ:

- `enqueue`: thêm phần tử vào cuối Queue
- `dequeue`: lấy và loại bỏ phần tử ở đầu Queue
- `front`: xem giá trị phần tử đầu Queue
- `isEmpty`: kiểm tra Queue rỗng
- `count`: đếm số lượng phần tử

6.4 Ứng dụng của Queue

Queue thường được sử dụng trong:

- Quản lý tiến trình (process scheduling)
- Hệ thống hàng đợi in ấn

- Thuật toán BFS (Breadth-First Search)
- Xử lý tác vụ theo thứ tự đến

7 Yêu cầu bài nộp

Sinh viên được cung cấp một file mã nguồn **linked_list.cpp, double_linked_list.cpp, stack_queue.cpp** chứa thư viện, các định nghĩa hàm. Mã nguồn đã bao gồm đầy đủ các định nghĩa hàm cho tất cả các bài tập trong Lab 08. Sinh viên thực hiện các đoạn mã còn thiếu trong từng hàm bắt đầu từ **TODO** và kết thúc ở **END TODO**.

Chú ý: Sinh viên không được tự ý sửa đổi bất kỳ thành phần nào của mã nguồn mẫu như **Thư viện, định nghĩa hàm**. Nếu mã nguồn biên dịch không thành công khi chấm thì sẽ nhận điểm 0 cho bài tập đó. Sinh viên có thể sửa đổi hàm **main** để có thể nhập xuất dữ liệu để kiểm thử các hàm.

Khi nộp bài sinh viên đổi tên file mã nguồn thành **MSSV.zip** với MSSV là mã số sinh viên được nhà trường cung cấp.

Tên file zip mẫu:

```
25123456.zip
```

Tất cả các trường hợp làm sai yêu cầu sẽ nhận điểm 0 cho bài thực hành. Vì thế sinh viên cần đọc kỹ và thực hiện đúng yêu cầu.

8 Hướng dẫn chạy file thực hành lab_08

Tổ chức thư mục của lab_08 như sau:

```
lab_08
   |-- single_linked_list_helper
   |           |-- single_linked_list.h
   |           |-- single_linked_list.cpp
   |
   |-- double_linked_list_helper
   |           |-- double_linked_list.h
   |           |-- double_linked_list.cpp
   |-- stack_queue_helper
   |           |-- stack_queue.h
   |           |-- stack_queue.cpp
   |-- main.cpp
   |-- Makefile
```

Vì có nhiều file được include vào trong file main để chạy chương trình do đó để nhanh chóng compile và chạy chương trình. Trong bài lab này chúng ta sẽ tiếp cận với giải pháp dùng Makefile để compile và chạy chương trình. Gõ vào terminal để kiểm tra version của Make:

```
make -v
```

Nếu không xuất hiện version thì máy chưa có Make. Khi đó, các bạn có thể tham khảo hướng dẫn cài đặt Make. Sau khi đã cài xong, các bạn đi đến thư mục đang chưa file **Makefile** và có thể thực hiện các câu lệnh sau:

- **make all**: Compile code ở tất cả các file để tạo ra file thực thi (**.exe**). Đây là default command nên nếu bạn gõ **make** thì nó sẽ tự động hiểu là **make all**.
- **make clean**: Để xoá bỏ file thực thi vừa tạo.
- **make run**: Thực hiện compile code và chạy chương trình.

Các câu lệnh trong Makefile bao gồm:

```
# Makefile for Lab 08: Linked List, Stack, Queue
# Compiler and flags
CXX = g++
CXXFLAGS = -std=c++17 -Wall -g -I linked_list_helper -I ...

# Name of execution file
TARGET = main

# List of source files
SRC = main.cpp linked_list_helper/linked_list.cpp ...
# Default command
all:
    $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET)

# Clean command
clean:
    rm -f $(TARGET)

# Compile and run command
run: all
    ./$(TARGET)
```

- **Câu lệnh số 2**: Khai báo compiler là **g++**.
- **Câu lệnh số 3**: Khai báo thêm các flags khi compile code. Với các tham số sau **-I** chỉ các thư mục chưa file **.h**. Nếu có thêm nhiều thư viện tự định nghĩa ta sẽ thêm tiếp các cặp **-I library_directory**. Mỗi thư mục sẽ đi với một ký tự **-I**.
- **Câu lệnh số 6**: Khai báo tên file **Thực thi** được tạo ra khi compile mã nguồn.
- **Câu lệnh số 13**: Compile mã nguồn và tạo ra file thực thi. Với câu lệnh tương tự trong những tuần trước: **g++ main.cpp -o main**. Câu lệnh này sẽ được chạy khi ta gọi: **make all** hoặc **make**.

- **Câu lệnh số 17**: Xoá file thực thi vừa được tạo ra. Lệnh được thực thi khi gọi: **make clean**.
- **Câu lệnh số 21**: Thực hiện compile và chạy file thực thi. Lệnh được thực thi khi gọi: **make run**. Khi gọi lệnh này sẽ thực hiện hai việc liên tục là **make all** và **./main**. Với **make all** đã được trình bày ở trên và **./main** là để chạy file thực thi.

9 Bài tập

9.1 Khai báo Node và List

Cho các struct định nghĩa một danh sách liên kết đơn và node tương ứng.

```
struct Node {
    int key;
    Node *pNext;
};

struct List {
    Node *pHead;
    Node *pTail;
};
```

9.2 Tạo một Node từ một số nguyên data cho trước

```
Node *createNode(int data);
```

9.3 Khởi tạo List từ một Node cho trước

```
List *createList(Node *pNode);
```

9.4 Chèn một số nguyên vào đầu List

```
bool addHead(List *&L, int data);
```

9.5 Chèn một số nguyên vào cuối List

```
bool addTail(List *&L, int data);
```

9.6 Chèn một số nguyên vào vị trí pos

```
bool addPos(List *&L, int val, int pos);
```

9.7 Chèn trước node mang giá trị val

```
bool addBefore(List *&L, int data, int val);
```

9.8 Chèn sau node mang giá trị val

```
bool addAfter(List *&L, int data, int val);
```

9.9 Xóa node đầu của List

```
void removeHead(List *&L);
```

9.10 Xóa node cuối của List

```
void removeTail(List *&L);
```

9.11 Xóa node tại vị trí pos

```
void removePos(List *&L, int pos);
```

9.12 Xóa toàn bộ các node trong List

```
void removeAll(List *&L);
```

9.13 Xóa node đứng trước node mang giá trị val

```
void removeBefore(List *L, int val);
```

9.14 Xóa node đứng sau node mang giá trị val

```
void removeAfter(List *L, int val);
```

9.15 In danh sách ra màn hình

```
void printList(List *L);
```

9.16 Đếm số lượng node trong List

```
int countElements(List *L);
```

9.17 Đảo ngược danh sách

```
List *reverseList(List *L);
```

9.18 Xóa các node trùng nhau

```
void removeDuplicate(List *&L);
```

9.19 Xóa các node có giá trị val

```
bool removeElement(List *&L, int val);
```

10 Danh sách liên kết đôi

10.1 Khai báo DSLK đôi

```
struct DNode {
    int key;
    DNode *pNext;
    DNode *pPrev;
};

struct DList {
    DNode *pHead;
    DNode *pTail;
};
```

10.2 Yêu cầu

Thực hiện tất cả các yêu cầu đã nêu ở phần Danh sách liên kết đơn cho DSLK đôi.

11 Stack và Queue

11.1 Khai báo Node

```
struct SNode {
    int key;
    SNode *pNext;
};
```

11.2 Stack

Cài đặt Stack bằng DSLK đơn và viết các hàm:

- Khởi tạo Stack
- Push một phần tử vào Stack
- Pop một phần tử ra khỏi Stack và trả về giá trị
- Đếm số lượng phần tử trong Stack
- Kiểm tra Stack có rỗng hay không

11.3 Queue

Cài đặt Queue bằng DSLK đơn và viết các hàm:

- Khởi tạo Queue
- Enqueue một phần tử vào Queue
- Dequeue một phần tử ra khỏi Queue và trả về giá trị
- Đếm số lượng phần tử trong Queue
- Kiểm tra Queue có rỗng hay không