** 📌 লিংকড লিস্ট **

Print Linked List in Original Order

```c
#include <stdio.h>
#include <stdlib.h>


struct node {
    int data;
    struct node *link;
};


void printList(struct node *head) {
    for (struct node *temp = head; temp != NULL; temp = temp->link) {
        printf("%d -> ", temp->data);
    }
    printf("NULL\n");
}

int main() {

    struct node *head = malloc(sizeof(struct node));
    head->data = 23;
    head->link = NULL;

    struct node *current = malloc(sizeof(struct node));
    current->data = 48;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 57;
    current->link = NULL;
    head->link->link = current;

    current = malloc(sizeof(struct node));
    current->data = 4;
    current->link = NULL;
    head->link->link->link = current;

    current = malloc(sizeof(struct node));
    current->data = 12;
    current->link = NULL;
    head->link->link->link->link = current;


    printList(head);
```

```
    return 0;
}
```

** 💠 লিংকড লিস্ট **

একটি **Singly Linked List** হলো এমন একটি ডাটা স্ট্রাকচার যেখানে প্রতিটি **নোডের** মধ্যে দুটি অংশ থাকে:
1️⃣ **ডাটা (data):** নোডের মধ্যে stored Value।
2️⃣ **লিংক (pointer to next node):** পরবর্তী নোডের Address।

লিংকড লিস্টের ক্ষেত্রে **মেমোরি ধারাবাহিক (contiguous) ভাবে বরাদ্দ হয় না**, বরং প্রতিটি **নোড আলাদাভাবে তৈরি হয় এবং পয়েন্টার দ্বারা সংযুক্ত হয়।**

📌 🎯 বাস্তব জীবনের উদাহরণ: ট্রেনের বগি

লিংকড লিস্টকে **একটি ট্রেনের মতো কল্পনা করুন**, যেখানে:

- **প্রতিটি বগি (coach)** হলো একটি **নোড**।
- **বগির মধ্যে যাত্রীদের (data)** বসানো হয়।
- **প্রতিটি বগির সাথে একটি সংযোগ (link)** থাকে, যা **পরবর্তী বগির সাথে সংযুক্ত থাকে**।
- **শেষ বগিটি (last node)** `NULL` নির্দেশ করে, কারণ তার পর আর কোনো বগি নেই।

💠 ট্রেনের লাইন:

```
বগি 1 → বগি 2 → বগি 3 → বগি 4 → বগি 5 → শেষ (NULL)
```

📌 কোড বিশ্লেষণ (Step-by-Step)

1️⃣ নোড স্ট্রাকচার তৈরি

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

- এখানে `struct node` হল একটি **স্ট্রাকচার (structure)**, যা **নোডের** জন্য **ব্লুপ্রিন্ট** হিসেবে কাজ করে।
- `int data;` → নোডের মধ্যে সংরক্ষিত ডাটা।
- `struct node *link;` → পরবর্তী নোডের ঠিকানা সংরক্ষণ করার জন্য পয়েন্টার।

💠 ট্রেন উদাহরণ:

একটি **ট্রেনের বগি (node)** যেমন একে অপরের সাথে **সংযুক্ত থাকে (linked)**, তেমনি এখানে প্রতিটি **নোড** `link` পয়েন্টারের মাধ্যমে **পরবর্তী নোডের ঠিকানা ধারণ করে**।

2️⃣ লিংকড লিস্ট প্রিন্ট করার ফাংশন

```
void printList(struct node *head) {
    for (struct node *temp = head; temp != NULL; temp = temp->link) {
        printf("%d -> ", temp->data);
    }
    printf("NULL\n");
}
```

- এটি **লিংকড লিস্ট ট্রাভার্স (traverse) বা ঘোরার** জন্য ব্যবহার করা হয়।
- প্রথম নোড (`head`) থেকে শুরু করে, প্রতিটি **নোডের ডাটা প্রিন্ট** করা হয় এবং **পরবর্তী নোডে যাওয়া হয়**।
- **শেষে** `NULL` **প্রিন্ট** করা হয়, কারণ এটি লিস্টের **শেষ নির্দেশ করে**।

◈ ট্রেন উদাহরণ:
একজন **ট্রেন পরিদর্শক (Inspector)** প্রতিটি **বগিতে ঢুকে** যাত্রীদের **সংখ্যা দেখে (data)** এবং পরবর্তী বগিতে চলে যায়।

3️⃣ লিংকড লিস্ট তৈরি (`main()` ফাংশনে)

প্রথম নোড তৈরি করা হচ্ছে:

```
struct node *head = malloc(sizeof(struct node));
head->data = 23;
head->link = NULL;
```

- প্রথম নোড (`head`) **ডাইনামিক মেমোরি** দ্বারা তৈরি করা হয় (`malloc` ব্যবহার করে)।
- `data = 23` সেট করা হয় এবং `link = NULL` রাখা হয় কারণ এটি বর্তমানে একমাত্র নোড।

◈ ট্রেন উদাহরণ:
প্রথম **বগি (coach)** তৈরি হলো যেখানে **২৩** জন যাত্রী আছে।

4️⃣ নতুন নোড সংযুক্ত করা হচ্ছে

```
struct node *current = malloc(sizeof(struct node));
current->data = 48;
current->link = NULL;
head->link = current;
```

- **নতুন নোড** তৈরি করা হলো, যেখানে `data = 48`।
- এটিকে **প্রথম নোডের সাথে সংযুক্ত করা হলো** (`head->link = current`)।

পরবর্তী **নোড সংযুক্ত করা হচ্ছে:**

```
current = malloc(sizeof(struct node));
current->data = 57;
current->link = NULL;
head->link->link = current;
```

- নতুন **নোড (data = 57)** তৈরি করা হলো এবং এটি **আগের নোডের সাথে সংযুক্ত হলো**।

এই ধাপে ধাপে **পুরো লিংকড লিস্ট তৈরি করা হয়েছে**:

```
23 → 48 → 57 → 4 → 12 → NULL
```

◈ **ট্রেন উদাহরণ:**
প্রতিটি **নতুন বগি** আগের **বগির সাথে সংযুক্ত** হচ্ছে এবং যাত্রী নিয়ে এগিয়ে চলেছে।

📌 মেমোরি স্ট্রাকচার (Memory Representation)

মেমোরি ঠিকানা	ডাটা	পরবর্তী নোডের ঠিকানা (link)
`0x101`	23	`0x102`
`0x102`	48	`0x103`
`0x103`	57	`0x104`
`0x104`	4	`0x105`
`0x105`	12	`NULL`

📌 ভিজুয়াল ডায়াগ্রাম

```
+-------+     +-------+     +-------+     +-------+     +-------+
|  23   | --> |  48   | --> |  57   | --> |  4    | --> |  12   | --> NULL
+-------+     +-------+     +-------+     +-------+     +-------+
   head         node2         node3         node4         node5
```

- প্রতিটি **বক্স** একটি **নোড** নির্দেশ করে।
- `NULL` মানে এটি **শেষ নোড**।

📌 আউটপুট

```
23 -> 48 -> 57 -> 4 -> 12 -> NULL
```

◈ **এটি পুরো লিংকড লিস্ট ট্রাভার্স (traverse) করার পর প্রিন্ট হবে।**

Reverse Order

```c
#include <stdio.h>
#include <stdlib.h>


struct node {
    int data;
    struct node *link;
};


void printReverse(struct node *head) {
    if (head == NULL) {
        return;
    }
    printReverse(head->link);
    printf("%d -> ", head->data);
}

int main() {

    struct node *head = malloc(sizeof(struct node));
    head->data = 23;
    head->link = NULL;


    struct node *current = malloc(sizeof(struct node));
    current->data = 48;
    current->link = NULL;
    head->link = current;

    current = malloc(sizeof(struct node));
    current->data = 57;
    current->link = NULL;
    head->link->link = current;

    current = malloc(sizeof(struct node));
    current->data = 4;
    current->link = NULL;
    head->link->link->link = current;

    current = malloc(sizeof(struct node));
    current->data = 12;
    current->link = NULL;
    head->link->link->link->link = current;


    printReverse(head);
```

```
    printf("NULL\n");

    return 0;
}
```

1️⃣ স্ট্রাকচার ডিফাইন করা (`struct node`)

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

- `data` → Value Store করে।
- `link` → পরবর্তী নোডের Address সংরক্ষণ করে।

এই স্ট্রাকচারটি **লিংকড লিস্টের নোডগুলি** তৈরি করতে ব্যবহৃত হয়।

2️⃣ রিভার্স প্রিন্টিং ফাংশন (`printReverse`)

```
void printReverse(struct node *head) {
    if (head == NULL) {
        return;
    }
    printReverse(head->link); // পরবর্তী নোডে যাওয়ার জন্য রিকারশন কল
    printf("%d -> ", head->data); // রিকারশন থেকে ফিরে এসে ডাটা প্রিন্ট
}
```

- ফাংশনটি **রিকারশন ব্যবহার করে লিংকড লিস্টের শেষ পর্যন্ত পৌঁছায়**।
- তারপর **রিকারশন থেকে ফিরে এসে ডাটা প্রিন্ট** করে (LIFO অর্ডার)।
- **বেস কেস (`head == NULL`) রিকারশন থামিয়ে দেয়** যখন শেষ নোডে পৌঁছায়।

3️⃣ লিংকড লিস্ট তৈরি করা (মেইন ফাংশন)

```
struct node *head = malloc(sizeof(struct node));
head->data = 23;
head->link = NULL;
```

- `head` তৈরি করা হয় এবং এতে **23** মান সংরক্ষিত হয়।
- লিংক `NULL` (এখন পর্যন্ত পরবর্তী নোড নেই)।

আরও নোড যোগ করা

```
struct node *current = malloc(sizeof(struct node));
current->data = 48;
current->link = NULL;
head->link = current;
```

- **নতুন নোড তৈরি করা হয় যার মান** `48`।
- `head->link` এতে এর ঠিকানা সংরক্ষণ করে, যা প্রথম নোডের সাথে **লিংক হয়ে যায়**।

এভাবে আরও নোড যোগ করা হয়েছে:

```
current = malloc(sizeof(struct node));
current->data = 57;
current->link = NULL;
head->link->link = current;
```

```
current = malloc(sizeof(struct node));
current->data = 4;
current->link = NULL;
head->link->link->link = current;
```

```
current = malloc(sizeof(struct node));
current->data = 12;
current->link = NULL;
head->link->link->link->link = current;
```

এভাবে আমাদের **লিংকড লিস্ট তৈরি হয়:**

```
23 -> 48 -> 57 -> 4 -> 12 -> NULL
```

4️⃣ `printReverse` ফাংশন কল করা

```
printReverse(head);
printf("NULL\n");
```

এটি **লিংকড লিস্টকে উল্টো (reverse) অর্ডারে প্রিন্ট করে**:

```
12 -> 4 -> 57 -> 48 -> 23 -> NULL
```

📌 কিভাবে রিকারশন কাজ করে

লিংকড লিস্ট:
`23 -> 48 -> 57 -> 4 -> 12 -> NULL`

রিকারশন কলের ক্রম

ফাংশন কল	একশন (যাচ্ছে নিচে)	একশন (ফিরে আসছে)
`printReverse(23)`	`printReverse(48)` কল করে	`23` প্রিন্ট
`printReverse(48)`	`printReverse(57)` কল করে	`48` প্রিন্ট
`printReverse(57)`	`printReverse(4)` কল করে	`57` প্রিন্ট
`printReverse(4)`	`printReverse(12)` কল করে	`4` প্রিন্ট
`printReverse(12)`	`printReverse(NULL)` কল করে	`12` প্রিন্ট
`printReverse(NULL)` (বেস কেস)	রিটার্ন	—

📌 Example

ভাবুন একটা **প্লেটের স্তূপ** 🍽️:

1️⃣ আপনি **প্লেটগুলো একে একে স্তূপ করে রাখছেন**:

- 23 (নিচে)
- 48
- 57
- 4
- 12 (উপরে)

2️⃣ **যখন উল্টো (reverse) প্রিন্ট করতে হয়, প্রথমে উপরের প্লেটটি সরানো হয়!**

- 12 (প্রথমে বের হবে)
- 4
- 57
- 48
- 23 (শেষে বের হবে)

এটাই **রিকারশন** → **Last In, First Out (LIFO)**!

📌 ডায়াগ্রাম রেপ্রেজেন্টেশন

🔹 মেমরিতে লিংকড লিস্ট

```
head → [23 | *] → [48 | *] → [57 | *] → [4 | *] → [12 | NULL]
```

প্রতিটি `[Data | Address]` একটি নোডকে নির্দেশ করে।

◈ রিকারশন কল স্ট্যাক (রিভার্স অর্ডার)

```
printReverse(23)
 └── printReverse(48)
      └── printReverse(57)
           └── printReverse(4)
                └── printReverse(12)
                     └── printReverse(NULL)  (Base Case)
```

রিকারশন থেকে ফিরে এসে প্রিন্ট হবে:

```
12 -> 4 -> 57 -> 48 -> 23 -> NULL
```

📌 সারাংশ

✅ **রিকারশন ব্যবহার করা হয়** শেষ নোডে পৌঁছানোর জন্য।
✅ **প্রিন্টিং হয় ফিরে আসার পথে**, অর্ডার উল্টে যায়।

**📖 রিকারশন ** 💥

আপনার কাজ হলো **সর্বশেষ কক্ষে পৌঁছানো**, তারপর **ফিরে আসার সময় সংখ্যাগুলো উল্টো ক্রমে লেখা**।

◈ গুহার নিয়ম

1. প্রতিটি কক্ষের **একটি দরজা আছে**, যা আপনাকে **পরবর্তী কক্ষে নিয়ে যায়** (শেষ কক্ষ ছাড়া)।
2. যদি সামনে আরেকটি কক্ষ থাকে, তাহলে **আপনাকে যেতে হবে**, পেছনে ফিরে আসা যাবে না।
3. যদি **একটি বন্ধ কক্ষ (শেষ কক্ষ)** পান, তখন **ফিরতে হবে এবং ফেরার সময় সংখ্যা লিখতে হবে**।

◈ গুহার অভিযান (রিকারশন কীভাবে কাজ করছে)

আপনি গুহায় প্রবেশ করলেন, আর কক্ষগুলোর **সংখ্যাগুলো এই ক্রমে সাজানো**:

কক্ষ	দেয়ালে লেখা সংখ্যা	পথ
কক্ষ ১	23	কক্ষ ২-এ যায়
কক্ষ ২	48	কক্ষ ৩-এ যায়
কক্ষ ৩	57	কক্ষ ৪-এ যায়
কক্ষ ৪	4	কক্ষ ৫-এ যায়
কক্ষ ৫	12	শেষ কক্ষ (আর কোনো দরজা নেই)

💡 **আপনার লক্ষ্য হলো সংখ্যাগুলোকে উল্টো করে লেখা (শেষ কক্ষ থেকে ফিরে আসা)।**

◈ ধাপে ধাপে রিকারশন কিভাবে কাজ করছে?

🔄 **প্রত্যেক কক্ষকে একটি রিকারশন ফাংশন কল হিসেবে ধরা যাক।**

ধাপ	সামনের দিকে এগোনো (ফাংশন কল)	রিকারশন ফাংশন কল
1️⃣	আপনি কক্ষ ১-এ প্রবেশ করলেন (সংখ্যা: 23)	`printReverse(23)`
2️⃣	আপনি দরজা খুলে কক্ষ ২-এ গেলেন	`printReverse(48)`
3️⃣	আপনি কক্ষ ৩-এ গেলেন	`printReverse(57)`
4️⃣	আপনি কক্ষ ৪-এ গেলেন	`printReverse(4)`
5️⃣	আপনি কক্ষ ৫-এ গেলেন (শেষ কক্ষ)	`printReverse(12)`
6️⃣	আপনি বুঝতে পারলেন সামনে আর পথ নেই (বেস কেস)	`printReverse(NULL) → ফিরে আসুন`

◈ ধাপে ধাপে উল্টো দিকে ফেরা (সংখ্যা লেখা)

এখন আপনি **ফিরতে শুরু করলেন**, এবং **সংখ্যাগুলো লিখতে লাগলেন**।

ধাপ	ফিরে আসা (ব্যাকট্র্যাকিং)	প্রিন্টিং
1️⃣	কক্ষ ৫ থেকে ফিরে এলেন	`প্রিন্ট 12`
2️⃣	কক্ষ ৪ থেকে ফিরে এলেন	`প্রিন্ট 4`
3️⃣	কক্ষ ৩ থেকে ফিরে এলেন	`প্রিন্ট 57`
4️⃣	কক্ষ ২ থেকে ফিরে এলেন	`প্রিন্ট 48`
5️⃣	কক্ষ ১ থেকে ফিরে এলেন	`প্রিন্ট 23`

✅ **শেষ আউটপুট (উল্টো ক্রমে):**

```
12 -> 4 -> 57 -> 48 -> 23 -> NULL
```

◈ রিকারশন কোডের ব্যাখ্যা

আপনার অভিযানের প্রতিটি ধাপ নিচের ফাংশনের মধ্যে **ঠিক যেভাবে ঘটছে, সেভাবেই কাজ করছে**:

🚀 রিকারশন ফাংশন বিশ্লেষণ

```
void printReverse(struct node *head) {
    if (head == NULL) {  // 🚪 বেস কেস: আর কোনো কক্ষ নেই
        return;
```

```
    }
    printReverse(head->link); // 🏃 সামনের কক্ষে যান
    printf("%d -> ", head->data); // 📝 ফিরে আসার সময় সংখ্যা লিখুন
}
```

◈ **সামনে যাওয়ার কাজ:** `printReverse(head->link)` কল করলে **নতুন কক্ষে প্রবেশ করা হয়**।
◈ **ফিরে আসার কাজ:** `printf("%d -> ", head->data)` **ফিরে আসার সময় প্রিন্ট হয়** (উল্টো ক্রমে)।

◈ সাধারণ ক্রমে প্রিন্ট করা (উল্টো নয়)

যদি আমরা **সংখ্যাগুলো সাধারণ ক্রমে প্রিন্ট করতে চাই**, তাহলে **প্রিন্টিং রিকারশন কলের আগে করতে হবে**:

```
void printNormal(struct node *head) {
    if (head == NULL) {
        return;
    }
    printf("%d -> ", head->data); // প্রিন্ট করা হবে আগেই
    printNormal(head->link);
}
```

এটি নিচের আউটপুট দেবে:

```
23 -> 48 -> 57 -> 4 -> 12 -> NULL
```

◈ মূল বিষয়গুলো

✅ **রিকারশন হলো একমুখী গুহা অভিযান → আগে একদিকেই যেতে হবে, তারপর ফিরে আসতে হবে।**
✅ **বেস কেস (head == NULL) → এটি শেষ কক্ষ (আর দরজা নেই)।**
✅ **ফিরে আসার সময় প্রিন্ট করলে উল্টো ক্রমে সংখ্যা পাওয়া যায়।**
✅ **রিকারশন স্ট্যাক ব্যবহার করে (LIFO – Last In, First Out) → শেষ কলটি আগে এক্সিকিউট হয় যখন ফিরে আসে।**

◈ `printReverse` ফাংশনের পার্থক্য ব্যাখ্যা (বাংলা)

আমরা `printReverse` ফাংশনে `printf` **কোথায় রাখা হয়েছে** সেটার উপর ভিত্তি করে আউটপুটের পার্থক্য দেখতে পাই।

📌 `printReverse` ফাংশন বিশ্লেষণ

```
void printReverse(struct node *head) {
    if (head == NULL) { // বেস কেস: লিস্টের শেষ পর্যন্ত গেলে থামবে
        return;
```

```
    }
    printReverse(head->link); // রিকার্সিভ কল দিয়ে সামনে এগিয়ে যাবে
    printf("%d -> ", head->data); // রিকার্সন রিটার্ন হওয়ার পরে প্রিন্ট করবে
}
```

◈ প্রধান পার্থক্য

কোডের অবস্থান	এক্সিকিউশন ফ্লো	আউটপুট অর্ডার
প্রথমে রিকার্সিভ কল (`printReverse(head->link)`)	শেষ নোড পর্যন্ত যায়, তারপর ব্যাক করে প্রিন্ট করে	**রিভার্স অর্ডার** (শেষ নোড আগে প্রিন্ট হয়)
প্রথমে প্রিন্ট (`printf("%d -> ", head->data); printReverse(head->link);`)	প্রতিটা নোড প্রিন্ট করে, তারপর রিকার্সিভ কল দেয়	**মূল অর্ডার** (প্রথম নোড আগে প্রিন্ট হয়)

◈ স্টেপ-বাই-স্টেপ ব্যাখ্যা

আমরা যদি নিচের লিংকড লিস্ট নিই:

```
23 -> 48 -> 57 -> 4 -> 12 -> NULL
```

1️⃣ যখন রিকার্সিভ কল আগে দেওয়া হয়

এক্সিকিউশন ধাপ:

1. `printReverse(23)` → `printReverse(48)` কে কল করবে (**এখনো প্রিন্ট হয়নি**)
2. `printReverse(48)` → `printReverse(57)` কে কল করবে (**এখনো প্রিন্ট হয়নি**)
3. `printReverse(57)` → `printReverse(4)` কে কল করবে
4. `printReverse(4)` → `printReverse(12)` কে কল করবে
5. `printReverse(12)` → `printReverse(NULL)` কে কল করবে এবং **স্টপ করবে**

এখন রিকার্সন রিটার্ন হবে এবং প্রিন্ট শুরু করবে:

- `printReverse(12)` → **`12 ->`** প্রিন্ট হবে
- `printReverse(4)` → **`4 ->`** প্রিন্ট হবে
- `printReverse(57)` → **`57 ->`** প্রিন্ট হবে
- `printReverse(48)` → **`48 ->`** প্রিন্ট হবে
- `printReverse(23)` → **`23 ->`** প্রিন্ট হবে

✅ আউটপুট (রিভার্স অর্ডার)

```
12 -> 4 -> 57 -> 48 -> 23 -> NULL
```

কারণ, **প্রিন্ট করা হয়েছে রিকার্সন ব্যাক করার সময়।**

2️⃣ যখন প্রিন্ট আগে করা হয়

```c
void printReverse(struct node *head) {
    if (head == NULL) {
        return;
    }
    printf("%d -> ", head->data); // আগে প্রিন্ট করবে
    printReverse(head->link); // তারপর রিকার্সিভ কল দেবে
}
```

এক্সিকিউশন ধাপ:

1. `printReverse(23)` → **`23 ->`** প্রিন্ট করবে, তারপর `printReverse(48)` কল করবে
2. `printReverse(48)` → **`48 ->`** প্রিন্ট করবে, তারপর `printReverse(57)` কল করবে
3. `printReverse(57)` → **`57 ->`** প্রিন্ট করবে, তারপর `printReverse(4)` কল করবে
4. `printReverse(4)` → **`4 ->`** প্রিন্ট করবে, তারপর `printReverse(12)` কল করবে
5. `printReverse(12)` → **`12 ->`** প্রিন্ট করবে, তারপর `printReverse(NULL)` কল করবে এবং **স্টপ করবে**

✅ আউটপুট (মূল অর্ডার)

```
23 -> 48 -> 57 -> 4 -> 12 -> NULL
```

কারণ, **প্রিন্ট করা হয়েছে রিকার্সনের আগে।**

📌 পার্থক্যের মূল কারণ

1. **প্রিন্ট আগে হলে → Normal order আউটপুট আসে**
2. **প্রিন্ট পরে হলে → রিকার্সন ব্যাক করার সময় আউটপুট আসে (যা উল্টো)**
3. **রিকার্সন আসলে স্ট্যাক ব্যবহার করে (LIFO - Last In First Out)**, তাই যখন আমরা **শেষ নোড পর্যন্ত গিয়ে ফিরে আসি**, তখন Reverse Order প্রিন্ট হয়।

📌 সারসংক্ষেপ

✅ **রিকার্সন প্রথমে শেষ নোড পর্যন্ত যায়।**
✅ **প্রিন্ট যদি কলের পরে করা হয়, তাহলে Reverse প্রিন্ট হবে।**

✅ **যদি প্রিন্ট আগে করা হয়, তাহলে Normal Order আউটপুট আসবে।**