有 array 和 linklist big O 時間表

Data Structures

LINKED LISTS (CHAPTER 4)

Todo List

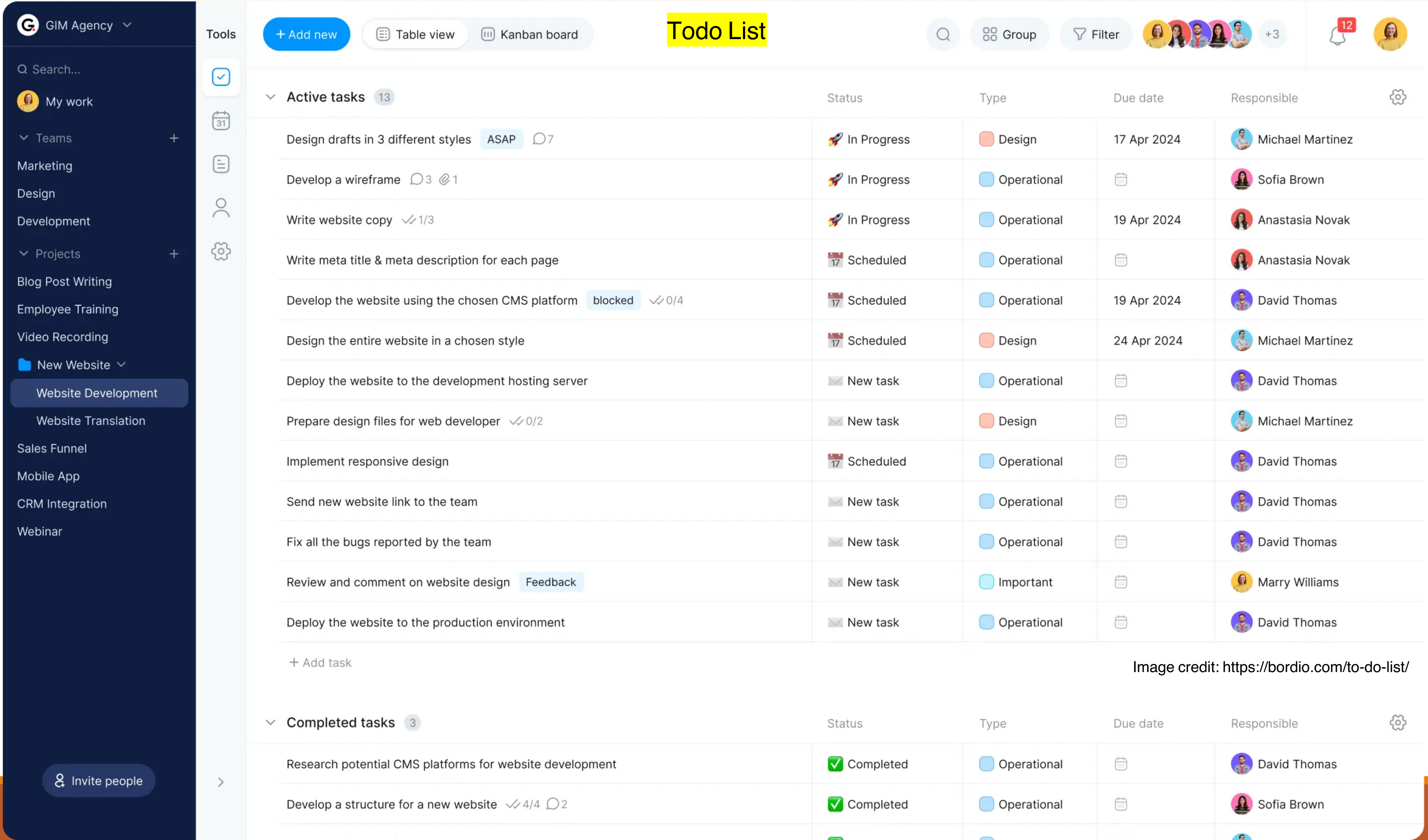


Image credit: https://bordio.com/to-do-list/

List

A number of connected items or names written or printed **consecutively**, typically one below the other.

For example:

- Check list
- Mailing list
- Todo list
- Wish list
- Memory management (OS)
- Data structures (stacks, queues, sets, hash tables, graphs)
- …

Efficiency Way for Insertion/Deletion

Efficiency way: O(1), O(n), O(n log n)?

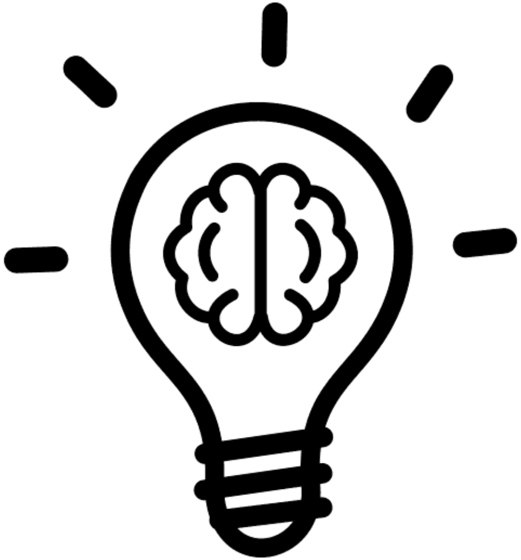


Image credit: https://uxwing.com/idea-icon/

Array

Array								
Integer Array	64	34	25	12	22	11	90	8

Array

Operations

- Insert an element into an array
- Delete an element from an array
- Costly shifts ⍰ improvement?

Efficiency Way for Insertion/Deletion

Traverse the entire data structures and insert/delete the target elements

Insertion

- Insert an elements before or after the target element

Deletion

- Delete the target element

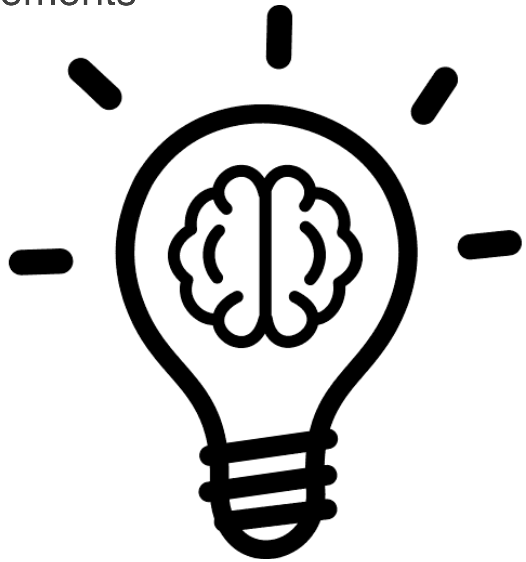


Image credit: https://uxwing.com/idea-icon/

What is This?

Image credit: https://www.britannica.com/technology/railroad-coupling

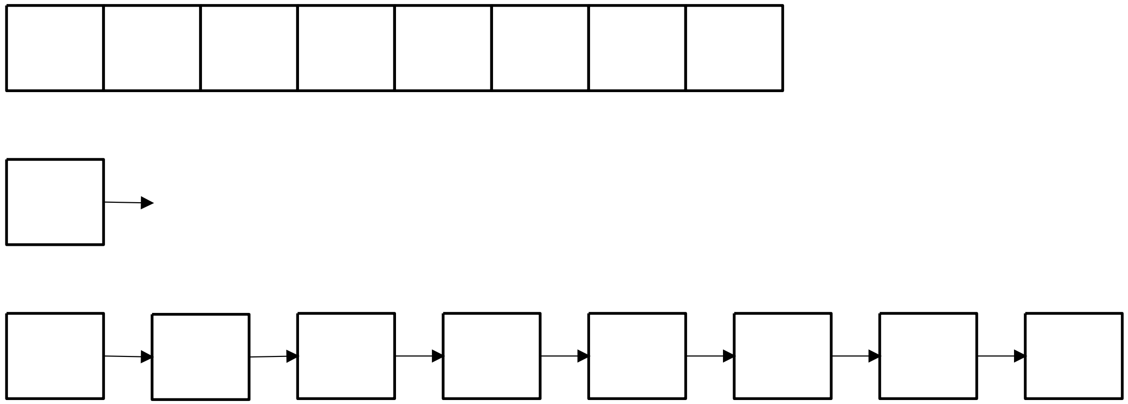
Linked List

Linked lists consist of a number of elements grouped, or linked, together in a specific order.

Array and Linked List

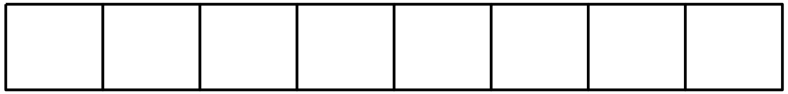
Something should be linked.

Add the linker will be easy to manipulate the order

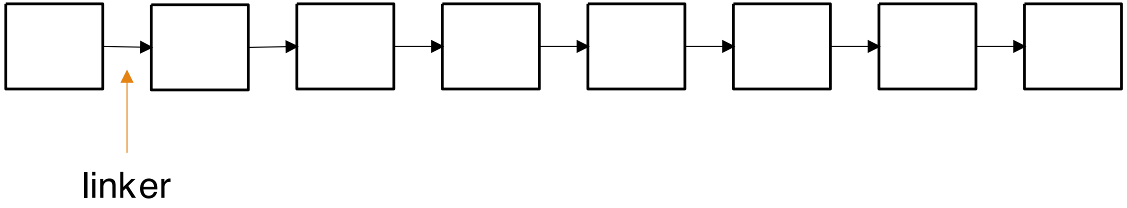


From Array to Linked List

1. Array



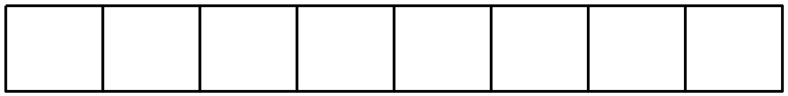
2. Linked list



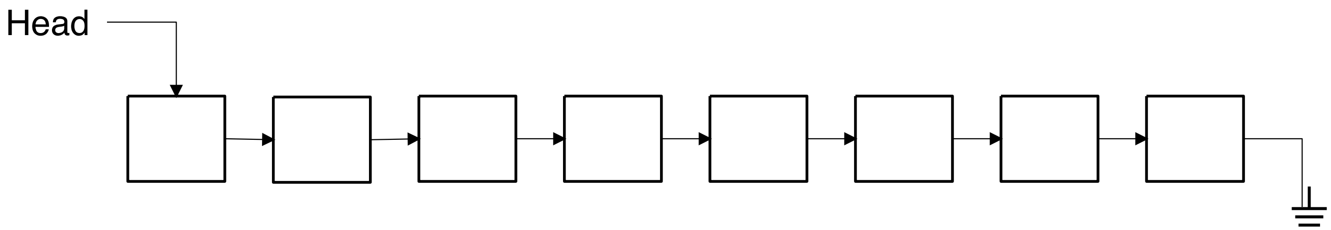
e.g. train

Something Missing?

1. Array



2. Linked list



Linked List

A linear data structure where elements (nodes) are stored in sequence, but not in contiguous memory locations.

Each node contains data and a reference (pointer) to the next node.

Unlike arrays, linked lists can grow or shrink during runtime, but accessing elements requires traversing from the head node.

ADT: Linked List

ADT *LinkedList* is

objects:

有限序列節點

A finite sequence of nodes, each containing:

- an item (data value from set item)

- a reference (pointer) to the next node 指向下一節點

The sequence terminates with a null reference.

functions:

for all $L \in LinkedList$, $x \in item$, $p \in position$

LinkedList Create()	::=	**return** an empty linked list
Boolean IsEmpty(*L*)	::=	**return** true **if** *L* has no nodes **else return** false
Integer Length(*L*)	::=	**return** the number of nodes in *L*
Item Retrieve(*L*, *p*)	::=	**if** position *p* is valid, **return** the item at position *p* **else return** error

ADT: Linked List

LinkedList Insert(*L*, *p*, *x*)	::=	**if** position *p* is valid, insert item *x* into list *L* at position *p* shift subsequent nodes if necessary
位置 insert值		**return** updated list
		else return error
LinkedList Delete(L, p)	::=	**if** position *p* is valid, remove the node at position *p*
		return updated list
		else return error
LinkedList Update(L, p, x)	::=	**if** position *p* is valid, replace the item at position *p* with x
		return updated list
		else return error
Position Search(L, x)	::=	**return** position of first occurrence of *x* in *L*
		else return error

end *LinkedList*

Linked List in C

```
struct list {
   int integerValue;
   struct list * nextPtr;
};

typedef struct list {
   int integerValue;
   struct list * nextPtr;
} IntegerNode;
```

Linked List in C++

Node Class

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

    Node(int value) {
        data = value;
        next = nullptr;
    }
};
```

Linked List in C++

```
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }
```

```
// Insert at end
void insert(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
```

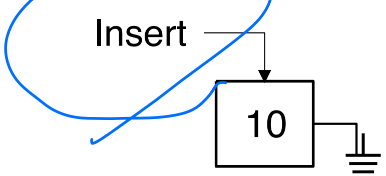
Linked List in C++

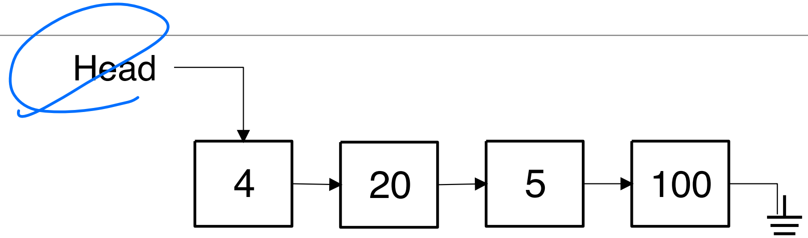
```cpp
// Print the list
void print() {
    Node* temp = head;
    while (temp != nullptr) {
        std::cout << temp->data << " -> ";
        temp = temp->next;
    }
    std::cout << "NULL" << std::endl;
}
```

```cpp
    // Destructor to free memory
    ~LinkedList() {
        Node* current = head;
        while (current != nullptr) {
            Node* nextNode = current->next;
            delete current;
            current = nextNode;
        }
    }
};
```

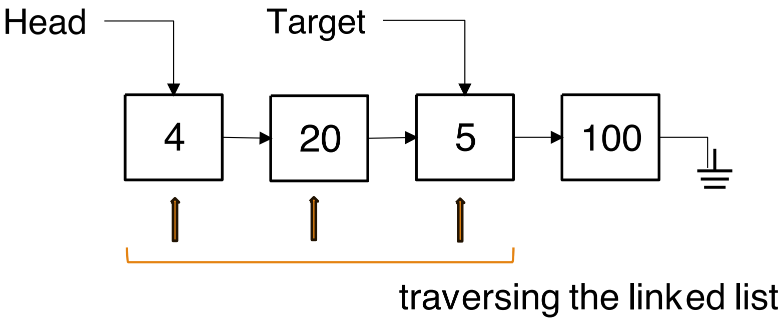
Operation: Insert Element (10) after the Target (5), integerLL

1 Initialize the node



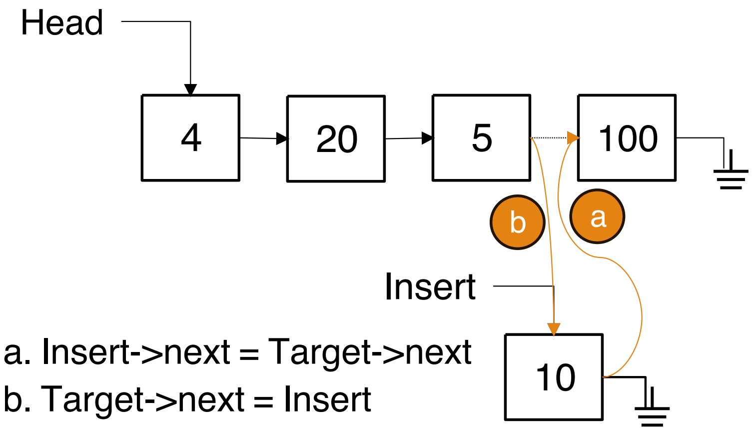


2 Traverse the linked list to find the target



Target = Head->next->next

3 Insert the Element (10)

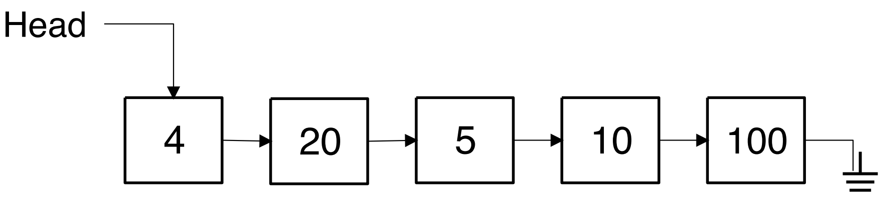


a. Insert->next = Target->next

b. Target->next = Insert

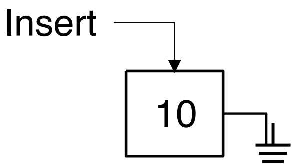
Operation: Insert Element (10) after the Target (5), integerLL

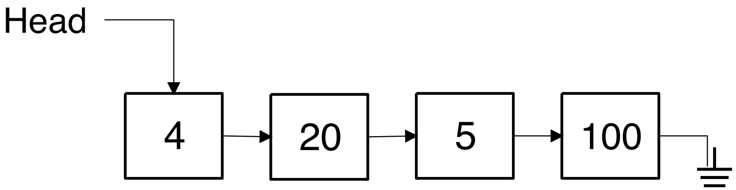
4 Final



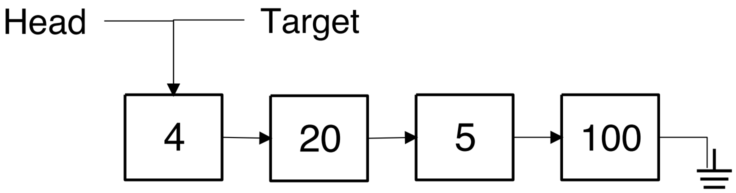
Operation: Insert Element (10) in the Beginning of the Linked List

1 Initialize the node



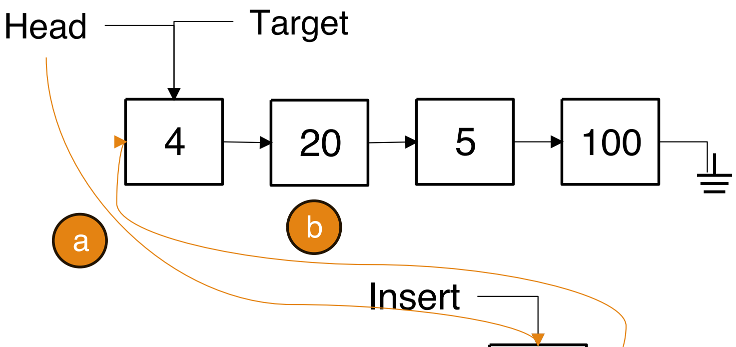


2 No need to traverse the linked list



Target = Head

3 Insert the Element (10)

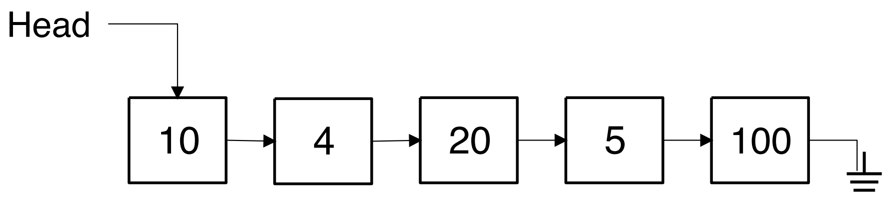


a. Head= Insert

b. Insert->next = Target

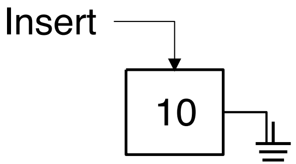
Operation: Insert Element (10) in the Beginning of the Linked List

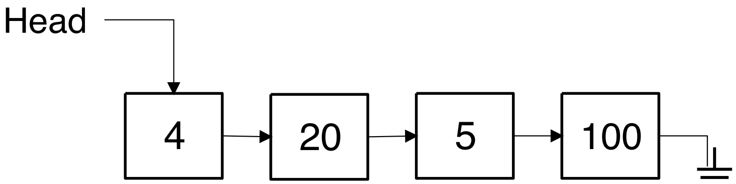
4 Final



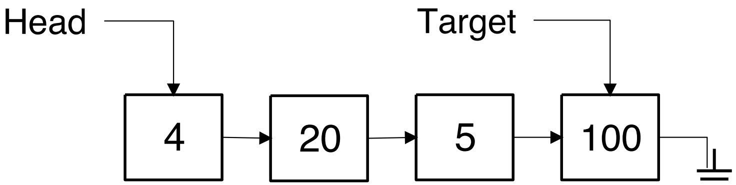
Operation: Insert Element (10) in the End of the Linked List

1 Initialize the node



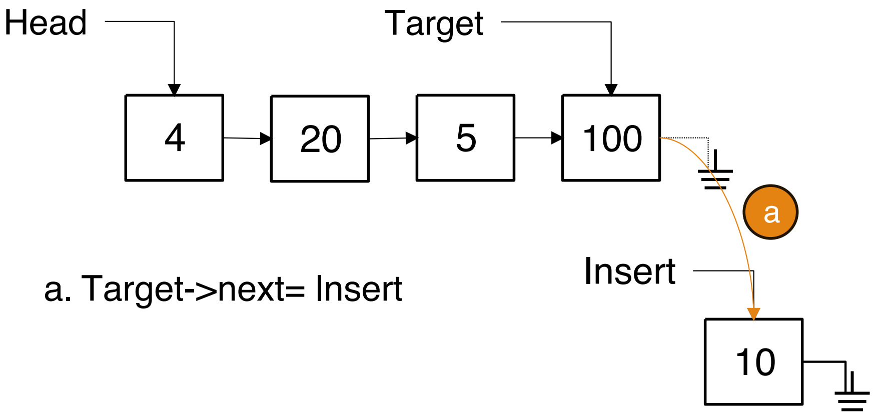


2 Traverse the linked list to find the end




```
Target = Head
While (Target->next != NULL){
        Target = Target->next;
}
```

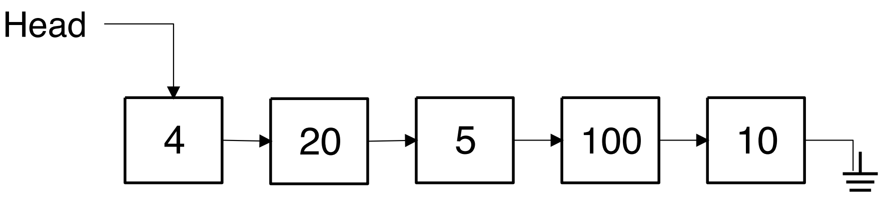
3 Insert the Element (10)



a. Target->next= Insert

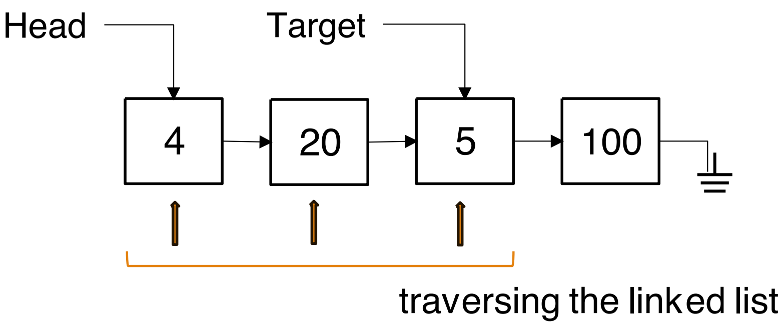
Operation: Insert Element (10) in the End of the Linked List

4 Final



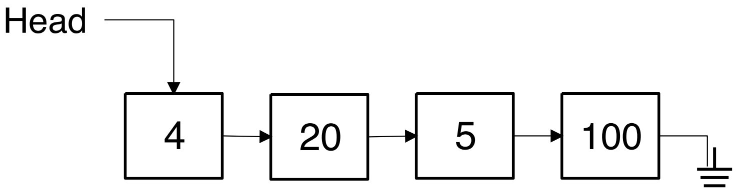
Operation: Delete the Target (5)

1 Traverse the linked list to find the target

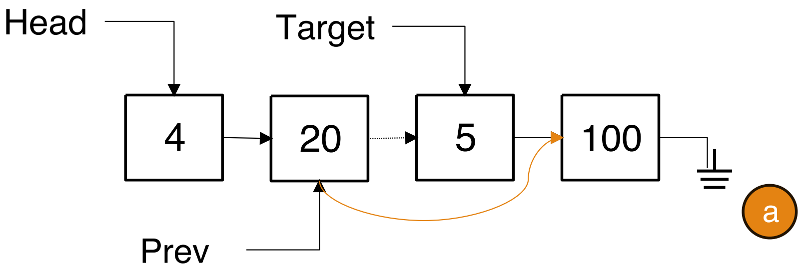


Target = Head
Prev = Head

While (Target->value !== Value){
    Prev = Target
    Target = Target->next
}



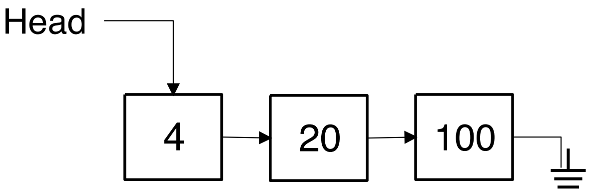
2 Delete the Element (5)



a. Prev>next= Target->next
b. free(Target)

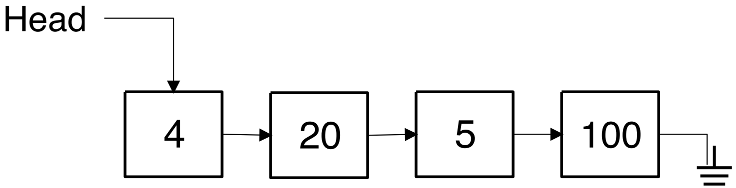
Operation: Delete the Target (5)

4 Final



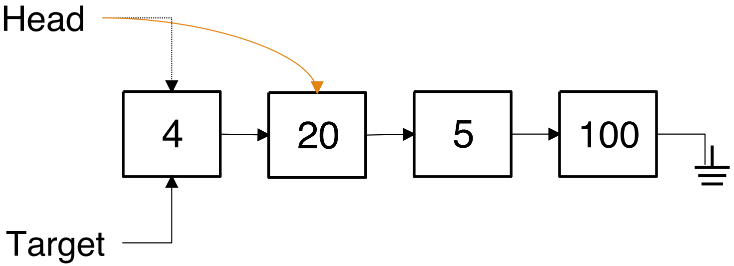
Operation: Delete the Target (First)

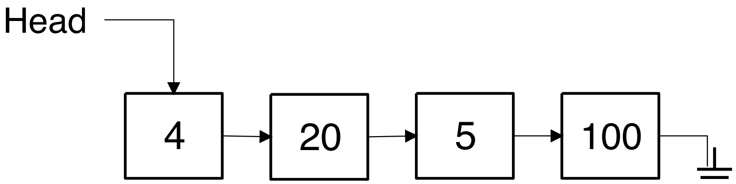
1. No need to traverse the linked list



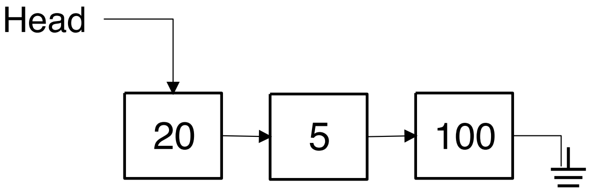
Target = Head
Prev = Head

2. Delete the first element (4)





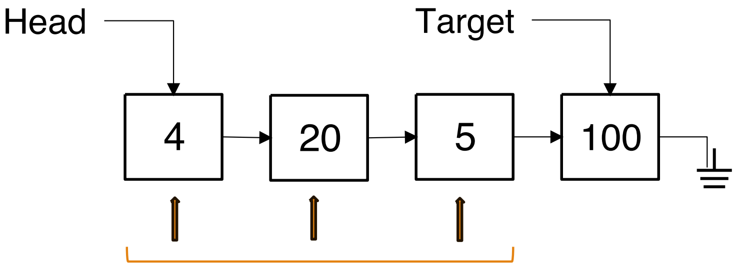
3. Final



Head = Target->next
free(target)

Operation: Delete the Target (Last)

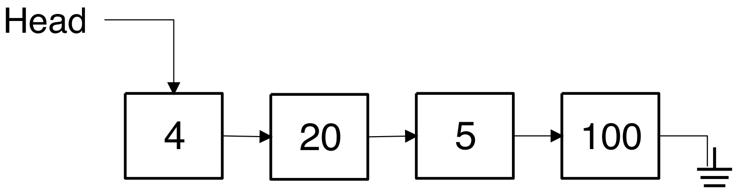
1 Traverse the linked list to fine the end



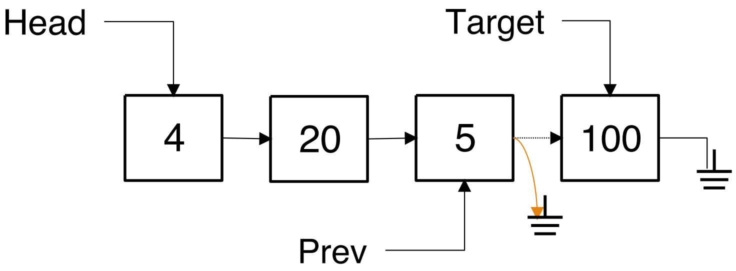
traversing the linked list

```
Target = Head
Prev = Head

While (Target->next !== NULL){
        Prev = Target
        Target = Target->next
}
```



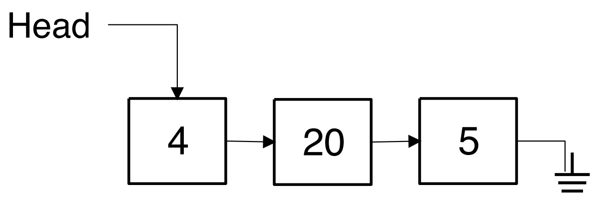
2 Delete the first element (4)




```
Prev->next = NULL
free(target)
```

Operation: Delete the Target (Last)

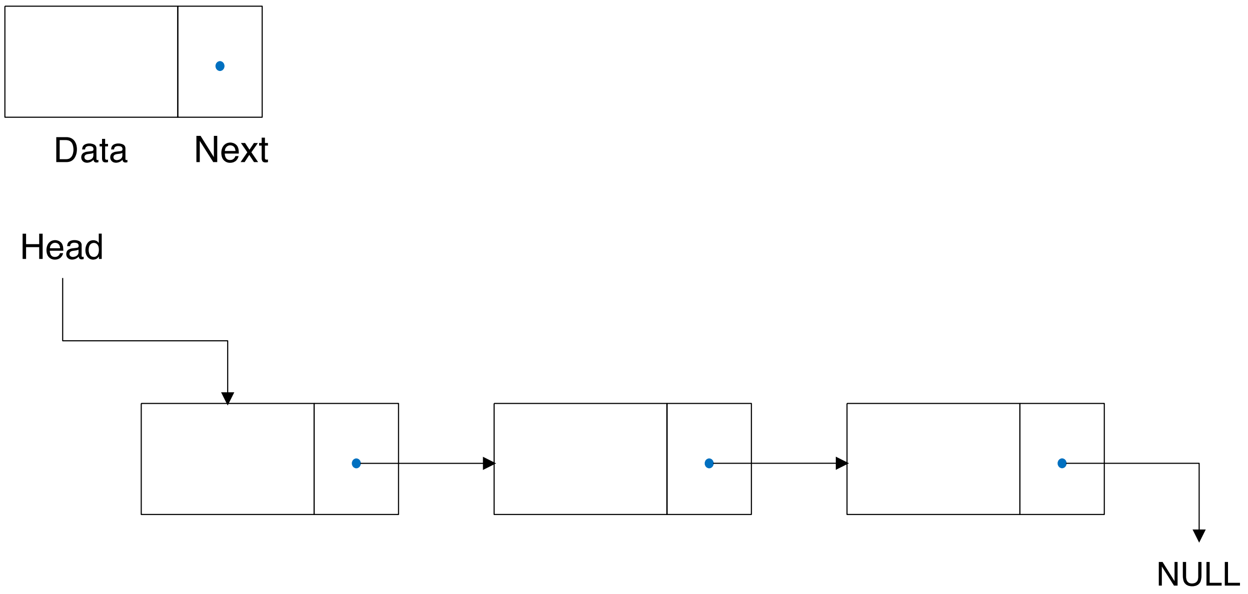
3 Final



Type of Linked List (Variation)

1. Singly linked list (basic form)
2. Doubly linked list (two-way navigation)
3. Circular linked list (wrap-around structure)
   - Singly linked List
   - Doubly linked list

Singly Linked List



Singly Linked List in C

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

Singly Linked List in C++

```
class SNode {
public:
    int data;
    SNode* next;
    SNode(int val) : data(val), next(nullptr) {}
};
```

Practice Problems: Traversing the Linked List and Observe the Memory Address

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

// Define the node structure
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

Practice Problems: Traversing the Linked List and Observe the Memory Address

```c
// Create a new node with given value
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));

    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

Practice Problems: Traversing the Linked List and Observe the Memory Address

```
// Traverse and print memory information
void traverseList(Node* head) {
    Node* current = head;
    int index = 0;

    while (current != NULL) {
        printf("Node %d: Value = %d, Address = %p, Next = %p\n",
               index, current->data, (void*)current, (void*)current->next);
        current = current->next;
        index++;
    }
}
```

Practice Problems: Traversing the Linked List and Observe the Memory Address

```
int main() {
    // Create linked list: 10 -> 20 -> 30 -> NULL
    Node* head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);

    // Traverse and print
    printf("Traversing the linked list:\n");
    traverseList(head);

    // Free memory
    Node* current = head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }

    return 0;
}
```

Doubly Linked List

next (指後面)

prev (指前面)

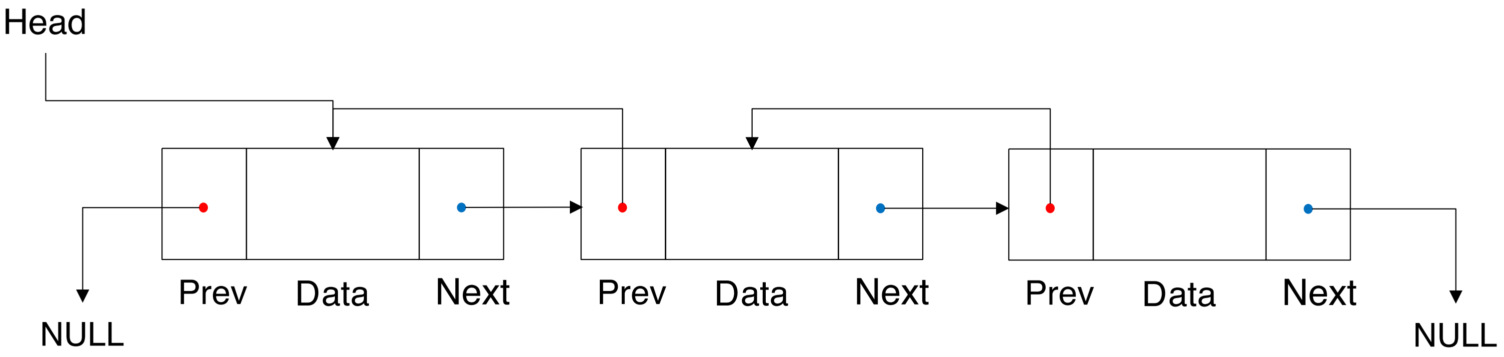
Doubly-linked lists, as their name implies, are composed of elements linked by two pointers.

Each element of a doubly-linked list consists of three parts: in addition to the data and the next pointer, each elements includes a pointer to the previous elements, called the prev pointer.

Improvement:

提供双向導航

Doubly Linked List



Doubly Linked List in C

```
struct DNode {
    int data;
    struct DNode *prev;
    struct DNode *next;
};
```

Doubly Linked List in C++

```
class DNode {
public:
    int data;
    DNode* prev;
    DNode* next;
    DNode(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

Circular Linked List

The circular linked list is another form of linked list that provides additional flexibility in traversing elements.

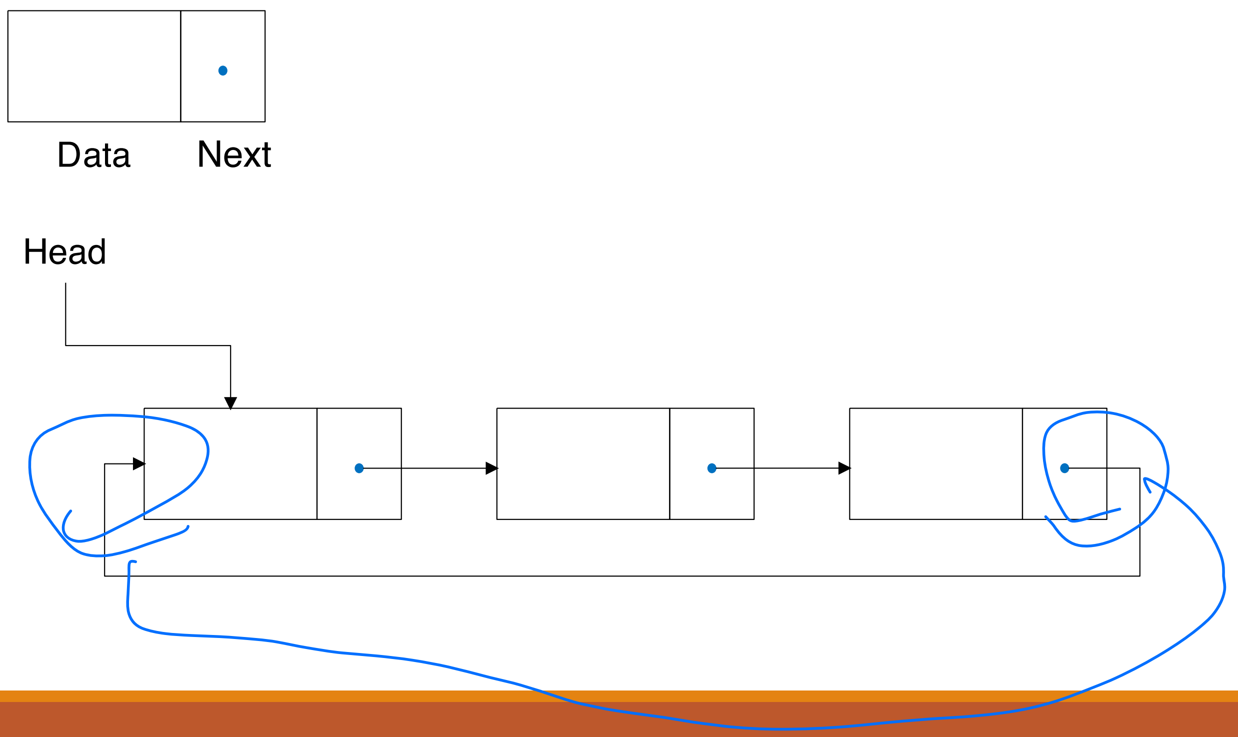
A circular linked may be singly-linked or doubly-linked, but its distinguishing feature is that it has no tail.

Improvement: 可是單向 (僅 next)

双向 (next , prev)

但結尾指向 head

Circular Linked List



Circular Linked List in C

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

Circular Linked List in C++

```
class CNode {
public:
    int data;
    CNode* next;
    CNode(int val) : data(val), next(nullptr) {}
};
```

Trade-offs: Variations of Linked List

1. traversal direction
2. insert/delete flexibility
3. memory overhead

Comparison: Array vs. Linked List

Aspect	Array	Linked List
Memory Allocation	Contiguous block of memory	Non-contiguous, each node allocated dynamically
Size	Fixed (static) – must be defined at declaration (in C)	Dynamic – can grow/shrink at runtime
存取 索引 **Access (Indexing)**	Direct access with index (`O(1)`)	Sequential access only (`O(n)`)
插入 刪除 **Insertion/Deletion**	Costly (`O(n)`) due to shifting elements	Efficient (`O(1)`) if pointer to node is known ↳ 如果 pointer 已知
Memory Usage	No extra overhead	額外 Requires extra memory for pointers
快取 **Cache Performance**	Better (contiguous memory → cache friendly)	分散 Worse (nodes may scatter in memory)
Implementation Simplicity	Easier to implement and use	More complex due to pointer handling
Use Cases	When size is known and frequent random access is needed	When frequent insertion/deletion is required and size is unpredictable

Comparison: Array vs. Linked List

Operation	Array (Dynamic)	Linked List
Access (by index)	O(1)	O(n) (must traverse)
Search	O(n)	O(n)
Insert at front	O(n)	O(1)
Insert at middle	O(n)	O(n)
Insert at end	O(1) amortized	O(n)* or O(1)**
Delete	O(n)	O(n) or O(1)***

- * O(n) unless tail pointer is used
- ** With tail pointer and singly linked list
- *** If node pointer is known, deletion is O(1)

std::list

Element access

`front`	access the first element (public member function)
`back`	access the last element (public member function)

Iterators

`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `crend` (C++11)	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`insert_range` (C++23)	inserts a range of elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`append_range` (C++23)	adds a range of elements to the end (public member function)
`pop_back`	removes the last element (public member function)
`push_front`	inserts an element to the beginning (public member function)
`emplace_front` (C++11)	constructs an element in-place at the beginning (public member function)
`prepend_range` (C++23)	adds a range of elements to the beginning (public member function)
`pop_front`	removes the first element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

Operations

`merge`	merges two sorted lists (public member function)
`splice`	transfers elements from another `list` (public member function)
`remove` `remove_if`	removes elements satisfying specific criteria (public member function)
`reverse`	reverses the order of the elements (public member function)
`unique`	removes consecutive duplicate elements (public member function)
`sort`	sorts the elements (public member function)

Key Observation of Linked List

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

```
struct Node* temp = head;
int i = 0;
while (temp != NULL) {
    printf("Node %d -> Value: %d, Address: %p, Next: %p\n", i, temp->data, (void*)temp,
(void*)temp->next);
    temp = temp->next;
    i++;
}
```

Key Observation of Linked List

Node Position

- head (start of the list)
- head->next
- head->next

Value of Each Node

- head->data → 0
- head->next->data → 1
- head->next->next->data → 2

Node Memory Location (Node Address)

- (void *) head
- (void *) head->next
- (void *) head->next->next

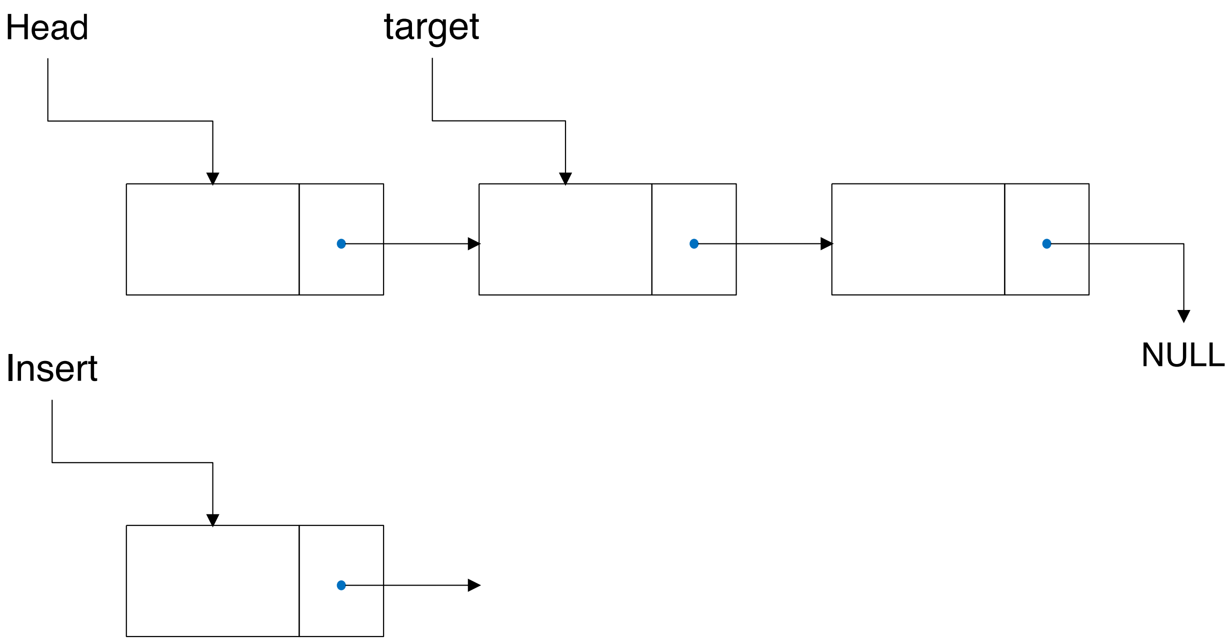
Next Pointer Address (Link to Next Node)

- (void *) head->next
- (void *) head->next->next

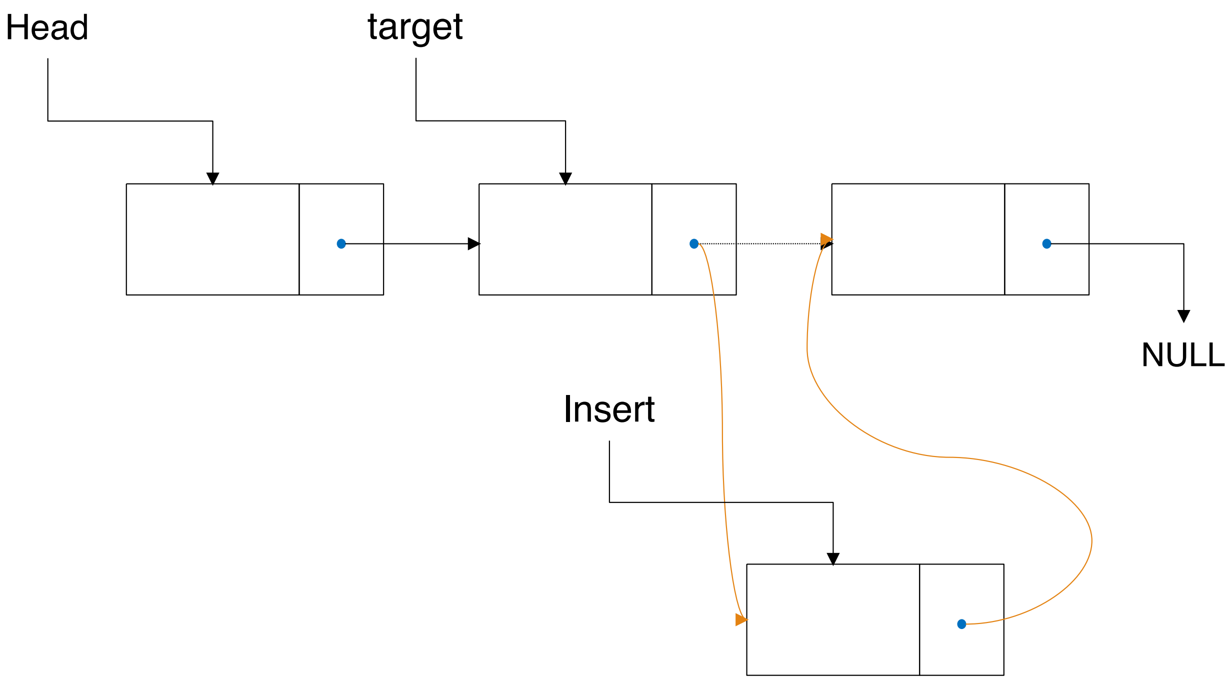
Summary

1. Create: always dynamic
   - Memory is allocated node-by-node at runtime.
   - Each node contains data + pointer to next node.
2. Retrieve: 只有在循環存取可以檢索
   - Sequential Access Only
   - Search the target
     - Unsorted
     - Sorted
3. Update
   - Target: Update data of a found node.
   - Insert
     - At head
     - At tail
     - In middle
   - Delete
     - Remove by value
     - Remove by position
     - Special cases: head/tail

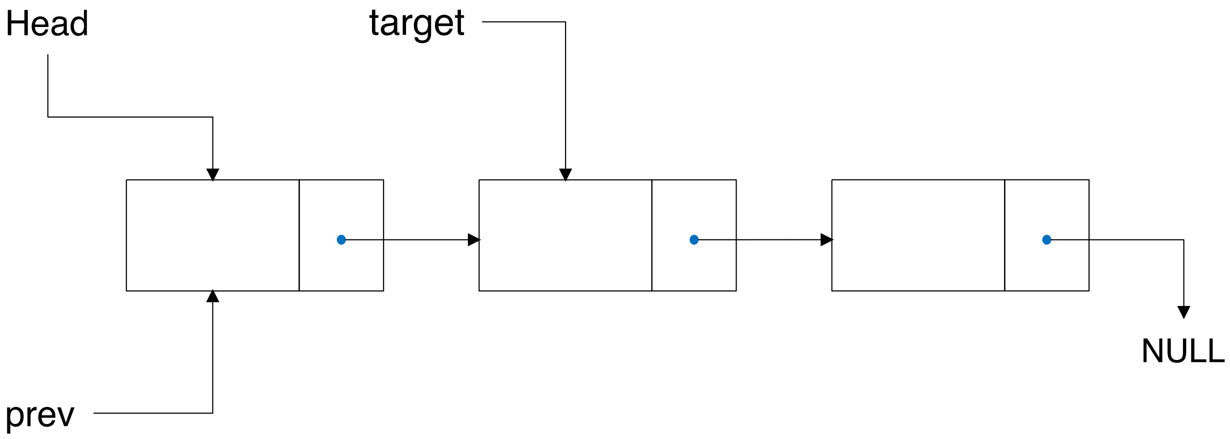
Linked List: Insert_after_Target()



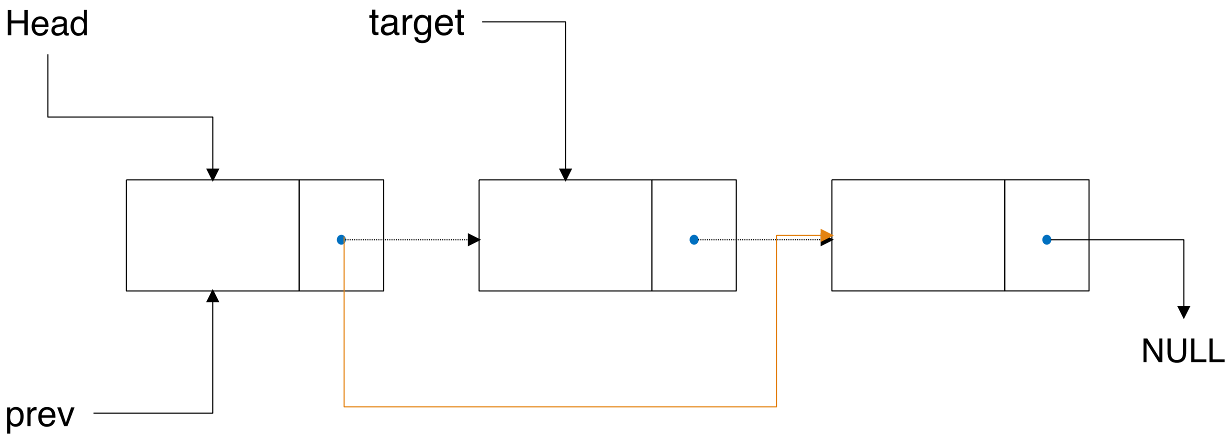
Linked List: Insert_after_Target()



Linked List: Delete()



Linked List: Delete()



Variation & Overhead

Extra pointer for quick operations: head & tail

Operations (ADTs)

- Insert before
- Insert after
- Insert_first
- Insert_last
- Delete_by_value
- Delete_by_position
- Delete_first
- Delete_last

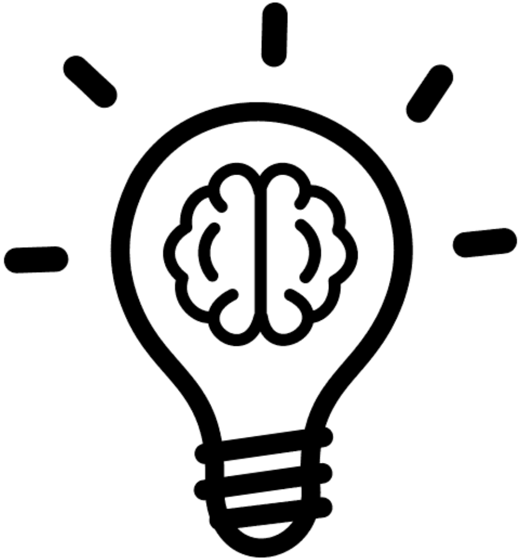
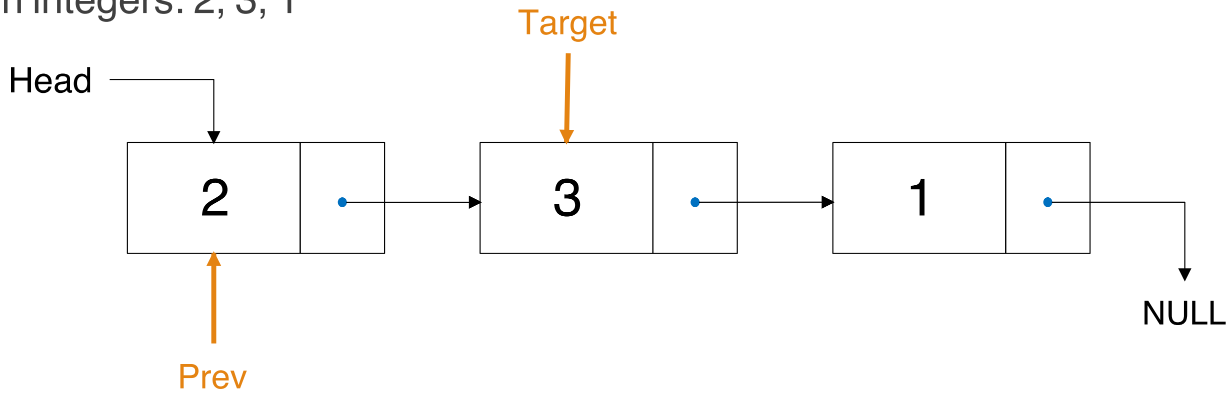


Image credit: https://uxwing.com/idea-icon/

Node Exchange in Linked List

Given integers: 2, 3, 1

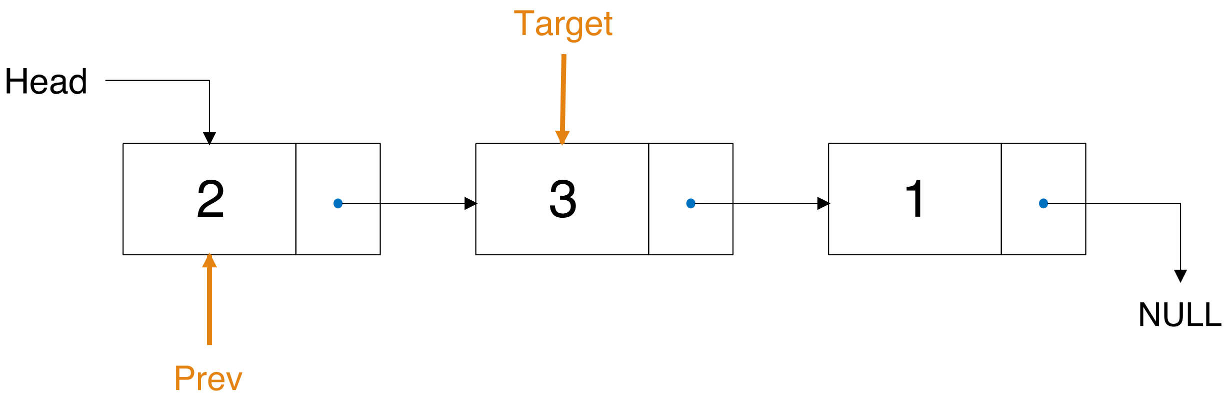


Question: Exchange Prev and Target

Prev: Prev
Prev->next

Target: Target
Target->next

Wrong



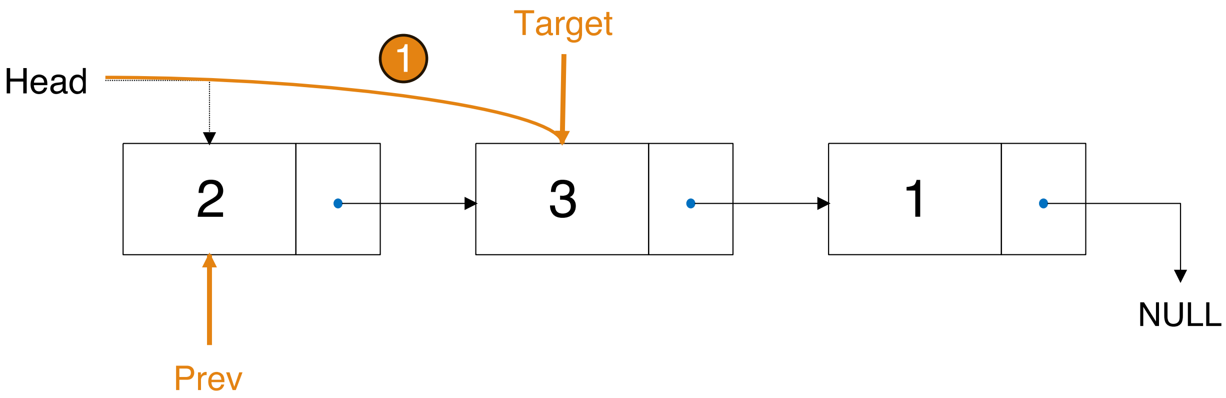
Prev: Prev
Prev->next

Target: Target
Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Target->next = Prev
3. Prev->next = Target->next (Prev)

Wrong

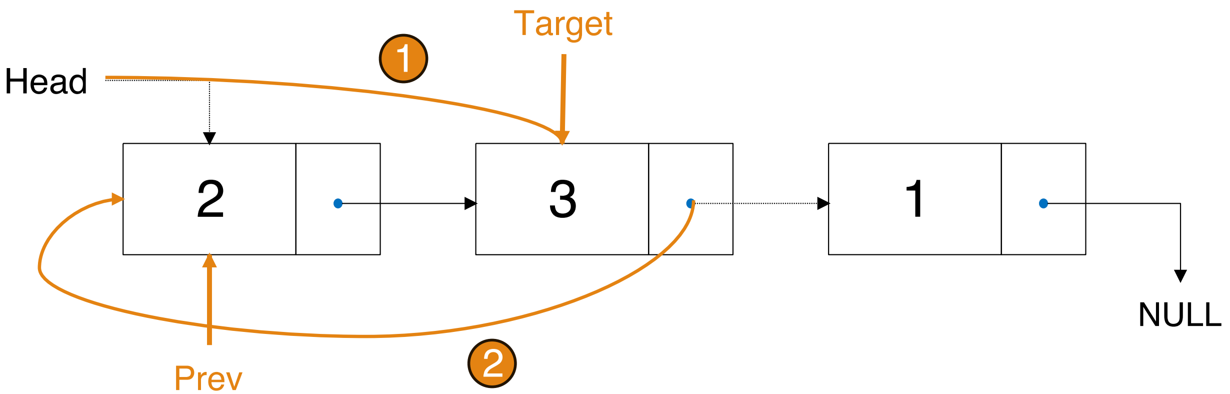


Prev: Prev
Prev->next
Target: Target
Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Target->next = Prev
3. Prev->next = Target->next (Prev)

Wrong



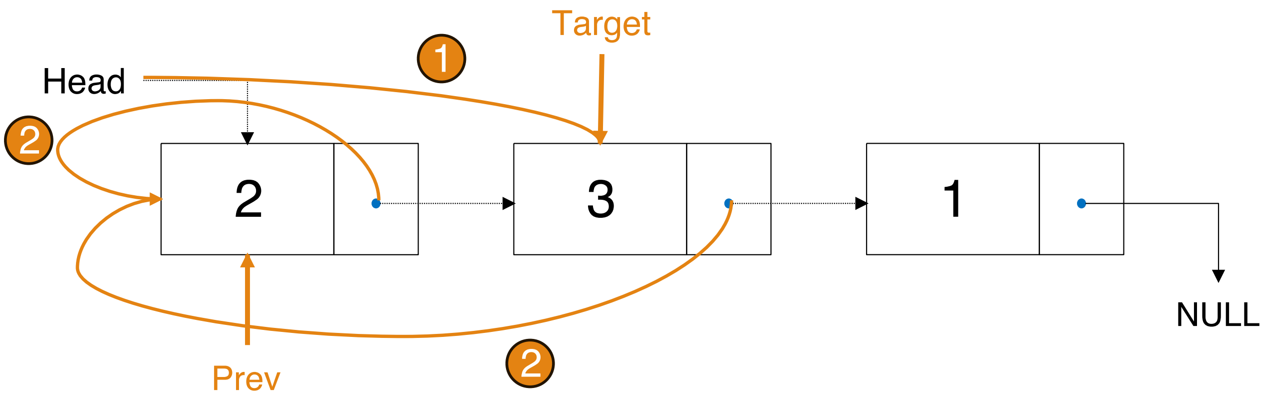
Prev: Prev
Prev->next

Target: Target
Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Target->next = Prev
3. Prev->next = Target->next (Prev)

Wrong



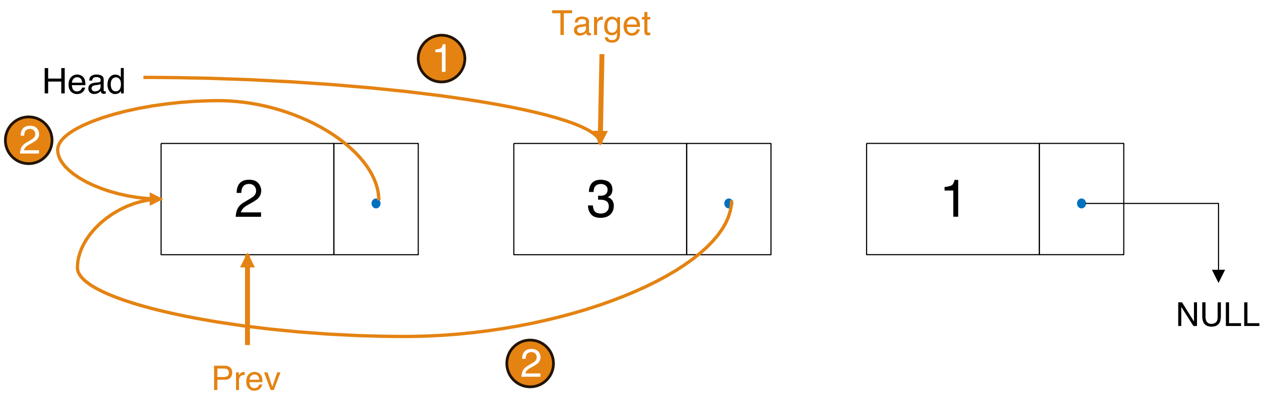
Prev: Prev
Prev->next

Target: Target
Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Target->next = Prev
3. Prev->next = Target->next (Prev)

Wrong (Final)

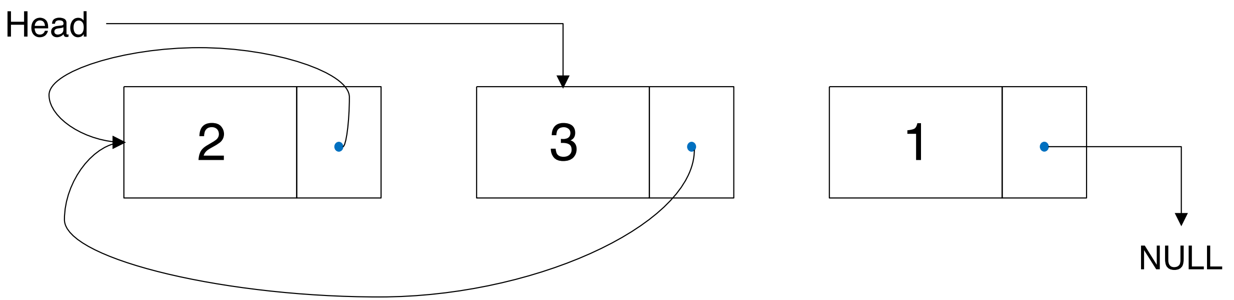


Prev:	Prev
	Prev->next
Target:	Target
	Target->next

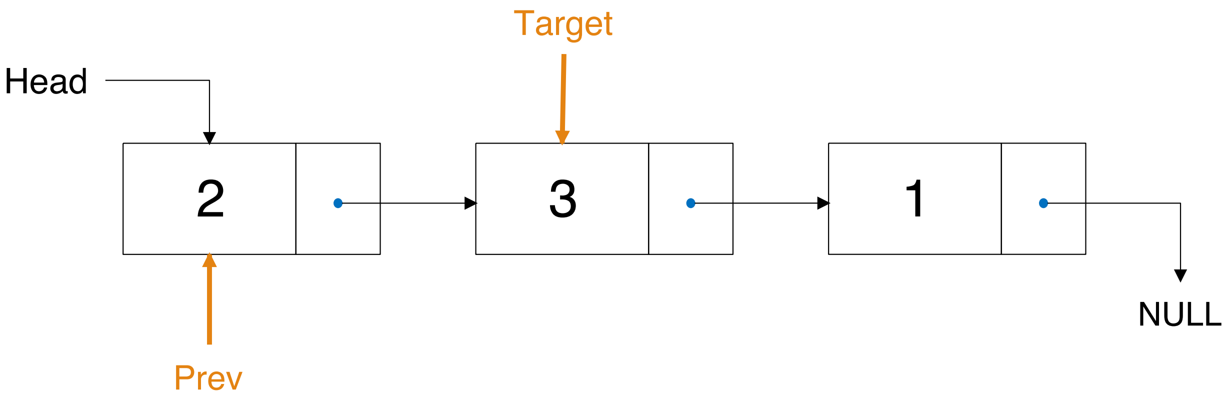
Steps

1. Head = Prev ? Head = Target
2. Target->next = Prev
3. Prev->next = Target->next (Prev)

Wrong (Final)



Correct

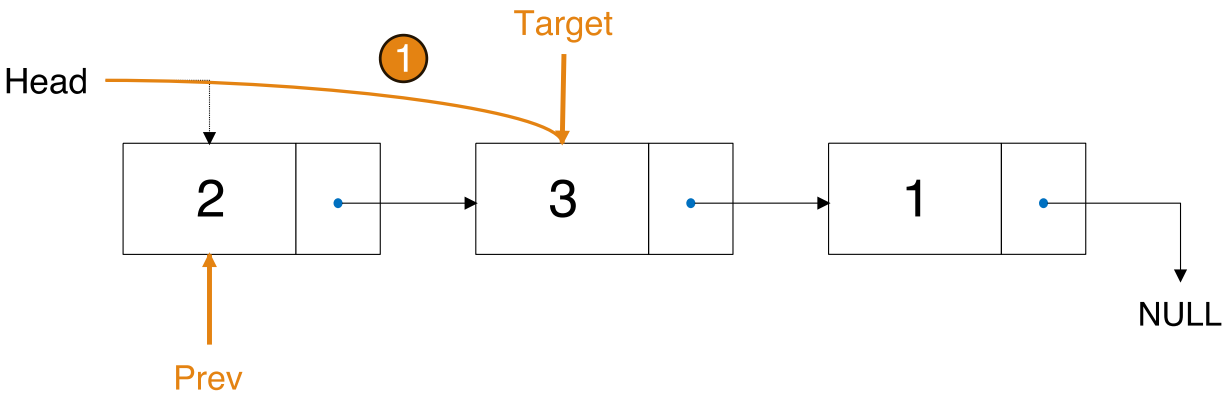


Prev: Prev
         Prev->next
Target: Target
         Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Prev->next = Target->next
3. Target->next = Prev

Correct



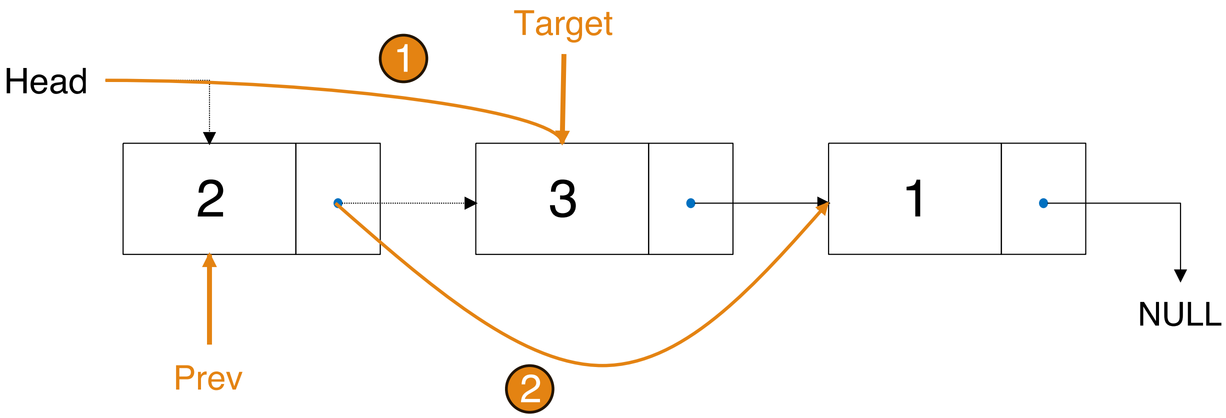
Prev: Prev
Prev->next

Target: Target
Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Prev->next = Target->next
3. Target->next = Prev

Correct

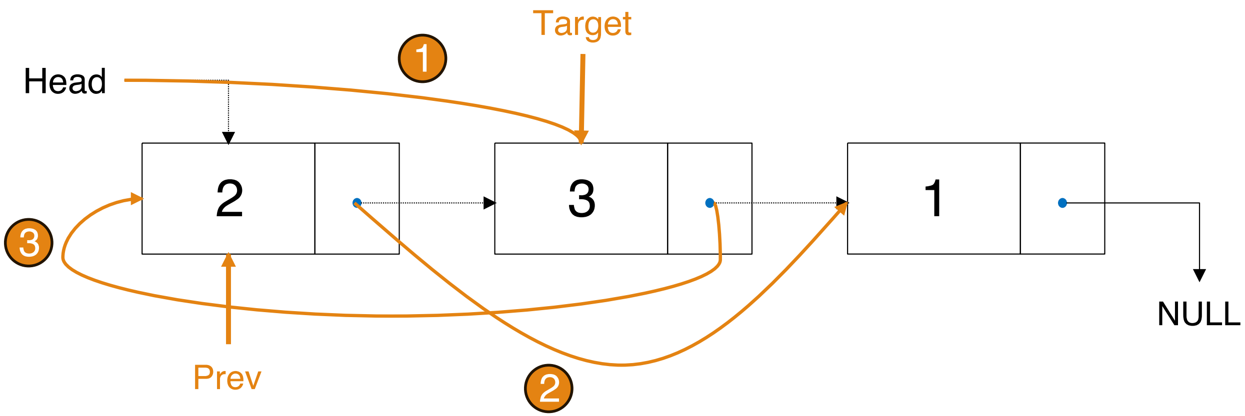


Prev:	Prev
	Prev->next
Target:	Target
	Target->next

Steps

1. Head = Prev → Head = Target
2. Prev->next = Target->next
3. Target->next = Prev

Correct

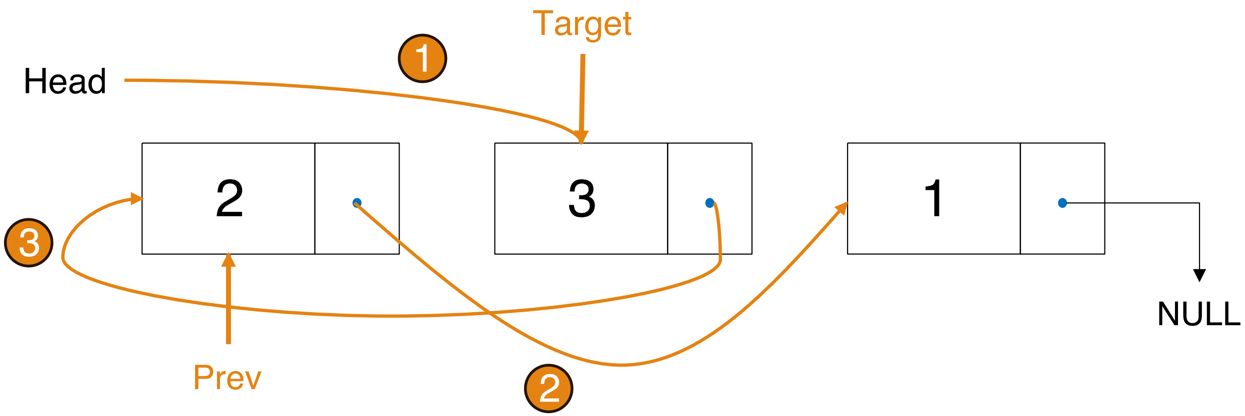


Prev:	Prev
	Prev->next
Target:	Target
	Target->next

Steps

1. Head = Prev ⍰ Head = Target
2. Prev->next = Target->next
3. Target->next = Prev

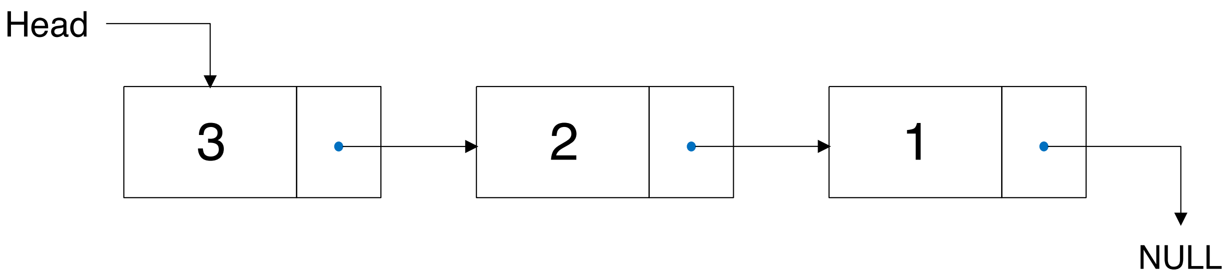
Correct (Final)



Prev:	Prev
	Prev->next
Target:	Target
	Target->next

Steps
1. Head = Prev ⍰ Head = Target
2. Prev->next = Target->next
3. Target->next = Prev

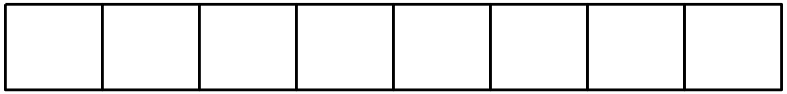
Correct (Final)

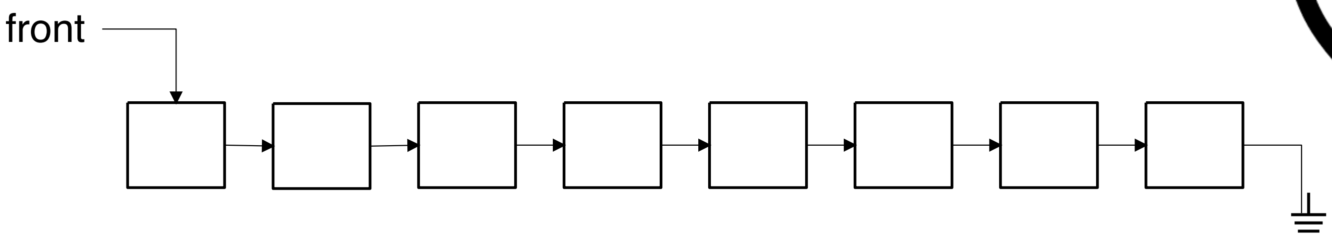


Node Exchange in Linked List

Correct vs. Wrong: pointer dependence

Think: the Integer Array, Sorting





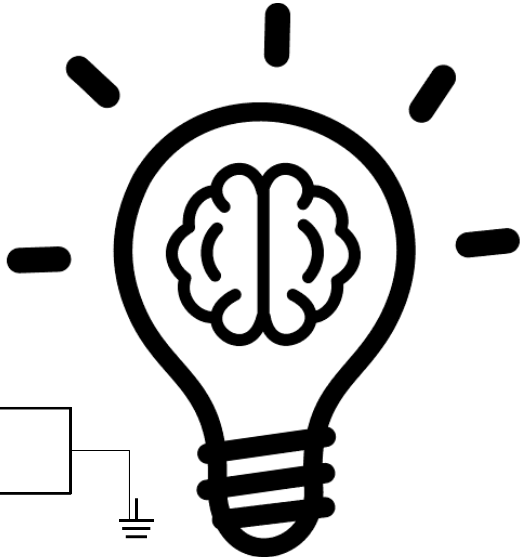
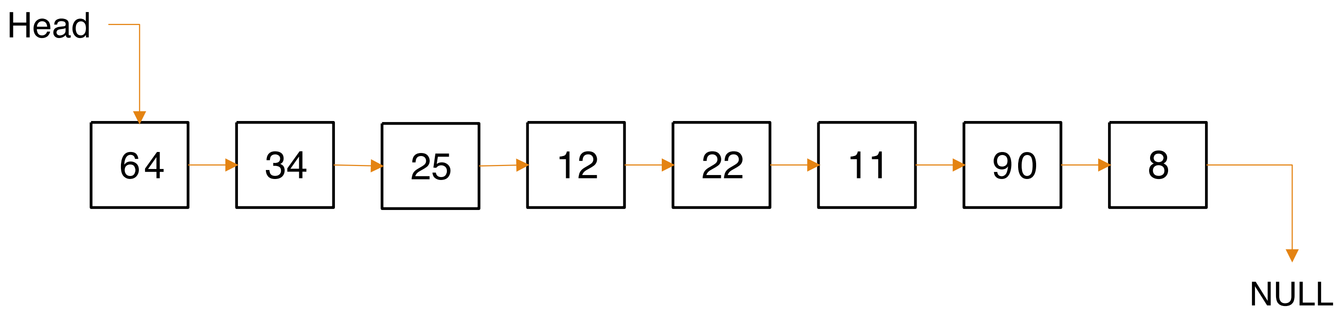


Image credit: https://uxwing.com/idea-icon/

Linked List Sorting

Think: the Integer Linked List

64	34	25	12	22	11	90	8



Selection Sort - Array

```
procedure selectionSort(A[1..n]):
    for i from 1 to n-1:
        minIndex = i
        for j from i+1 to n:
            if A[j] < A[minIndex]:
                minIndex = j
        swap A[i] and A[minIndex]
```

Selection Sort - Array

```
procedure selectionSort(A[1..n]):
    for i from 1 to n-1:
        minIndex = i
        for j from i+1 to n:
            if A[j] < A[minIndex]:
                minIndex = j
        swap A[i] and A[minIndex]
```

A0	64	34	25	12	22	11	90	8
A1	8	34	25	12	22	11	90	64
A2	8	11	25	12	22	34	90	64
A3	8	11	12	25	22	34	90	64
A4	8	11	12	22	25	34	90	64
A5	8	11	12	22	25	34	90	64
A6	8	11	12	22	25	34	90	64
A7	8	11	12	22	25	34	64	90

Selection Sort - Linked List

Linked List

- Swap the value
- Swap the pointer

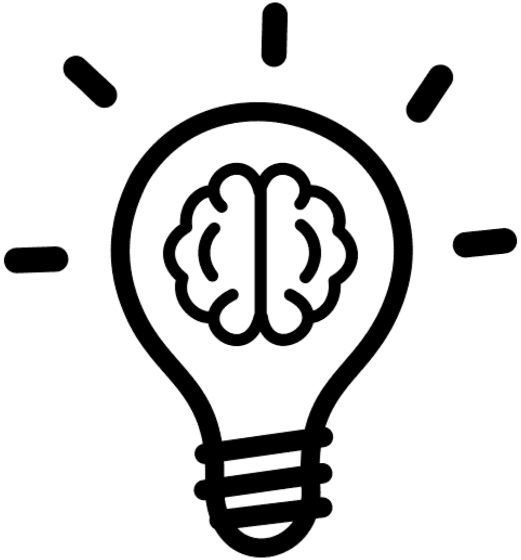


Image credit: https://uxwing.com/idea-icon/

Selection Sort - Linked List

Swap the value

```
void selectionSortValue(Node* head){
   for(Node* i=head; i && i->next; i=i->next){
      Node* minNode=i;
      for(Node* j=i->next; j; j=j->next){
         if(j->val < minNode->val) minNode=j;
      }
      if(minNode!=i){
         int t=i->val; i->val=minNode->val; minNode->val=t;
      }
   }
}
```

Selection Sort - Linked List

Swap the pointer

```
Node* selectionSortPointer(Node* head){
   if(!head) return head;
   Node dummy = {0, head};
   Node* sortedTail = &dummy;

   while(sortedTail->next){
      Node* minPrev = sortedTail;
      for(Node* p=sortedTail->next; p && p->next; p=p->next){
         if(p->next->val < minPrev->next->val) minPrev = p;
      }
      Node* minNode = minPrev->next;
      // detach minNode
      minPrev->next = minNode->next;
      // insert after sortedTail
      minNode->next = sortedTail->next;
      sortedTail->next = minNode;
      sortedTail = sortedTail->next;
   }
   return dummy.next;
}
```

Bonus: AI Prompt for Studying “Linked List” (1)

I understand basic singly linked lists and can create, traverse, and add nodes. Now I want to advance to intermediate level. Please help me with:

1. **Advanced List Types**：

   - Doubly linked lists - when and why to use them

   - Circular linked lists - practical applications

   - Singly vs doubly vs circular - trade-offs and comparisons

2. **Complex Operations**：

   - Insertion at any position (beginning, middle, end)

   - Deletion operations (by value, by position, specific cases)

   - Searching and finding nodes efficiently

   - Reversing a linked list (iterative and recursive approaches)

Bonus: AI Prompt for Studying “Linked List” (2)

3. **Algorithm Implementation**:

   - Merge two sorted linked lists
   - Detect cycles in linked lists (Floyd's cycle detection)
   - Find middle element efficiently
   - Remove duplicates from sorted/unsorted lists

4. **Memory Management & Error Handling**:

   - Proper memory allocation and deallocation
   - Handling edge cases (empty lists, single nodes)
   - Defensive programming techniques
   - Memory leak detection and prevention

Bonus: AI Prompt for Studying “Linked List” (3)

5. **Performance Analysis**：

- Time complexity of different operations
- Space complexity considerations
- When to use linked lists vs arrays vs vectors
- Performance trade-offs in real applications

6. **Advanced Techniques**：

- Using dummy/sentinel nodes to simplify code
- Two-pointer techniques (slow/fast pointers)
- Recursive vs iterative approaches
- Generic linked lists using templates

Bonus: AI Prompt for Studying “Linked List” (4)

7. **Practical Applications**：

- Implementing stacks and queues using linked lists
- Undo functionality in applications
- Hash table collision resolution with chaining
- Music playlist or browser history implementation