

Linked List

Inst. Nguyễn Minh Huy

Contents



- Linked List Concepts.
- Linked List Operations.
- Linked List Improvement.

Contents



- **Linked List Concepts.**
- Linked List Operations.
- Linked List Improvement.

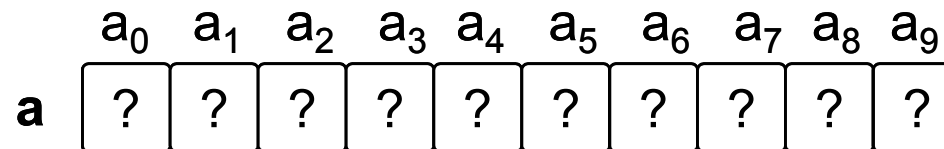
Linked List Concepts



■ Limitations of Array:

■ Characteristics:

- Continuous memory storage.



■ Advantages:

- Random element access by indexing.
- Very efficient for fix-sized storage.

■ Disadvantages:

- Resize array requires memory reallocation.
- Add or remove element need element shifted.
- Allocate large continuous memory is difficult.

Linked List Concepts



■ Linked list solution:

■ Locker renting problem:

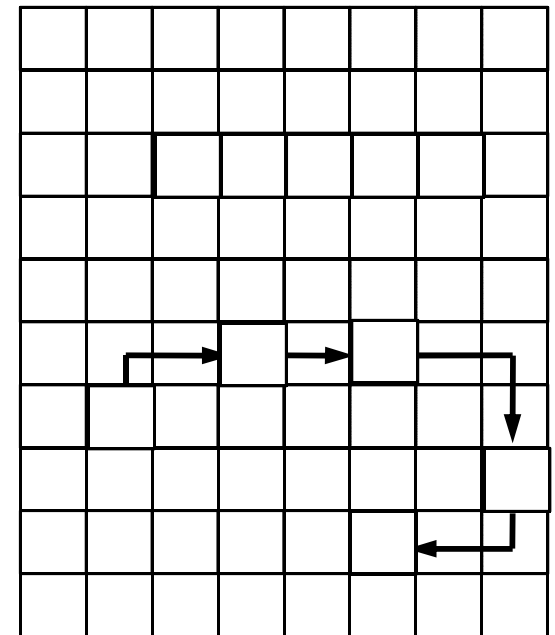
- Need to rent lockers to store N items.
- Each locker keeps only 1 item.

■ Continuous solution → Array.

■ Dis-continuous solution:

- Rent N arbitrary lockers.
- Each locker keeps:
 - 1 items.
 - Address of next locker.
- Only keep address of first locker.

Lockers



Linked List Concepts



■ Singly linked list:

- A discontinuous data structure.
- Element = Data + Link to next element.
- Last element link to NULL.
- Head: link to first element.



■ C declaration:

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

```
struct SList
{
    SNode *head;
};
```

Contents



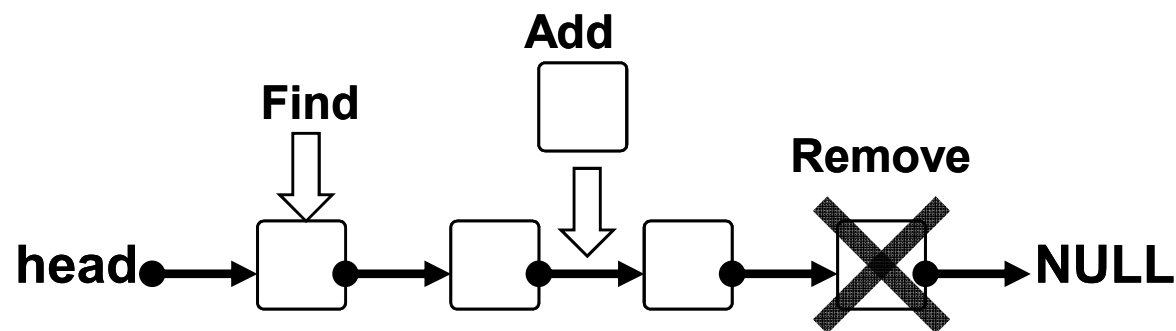
- Linked List Concepts.
- **Linked List Operations.**
- Linked List Improvement.

Linked List Operations



■ Operations on linked list:

- Initialize.
- Check empty.
- Find element.
- Add element.
- Remove element.



Linked List Operations



■ Initialize list:

- At first, list has no element.

head●→NULL

■ Check empty list:

- An empty list has no element.

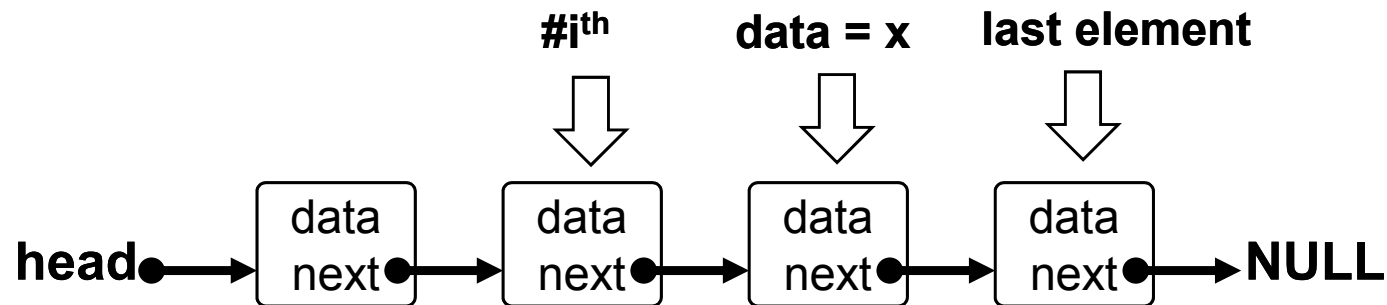
head●→NULL ???

Linked List Operations



■ Find element:

- Find $\#i^{\text{th}}$ element.
- Find element has data x.
- Find last element.

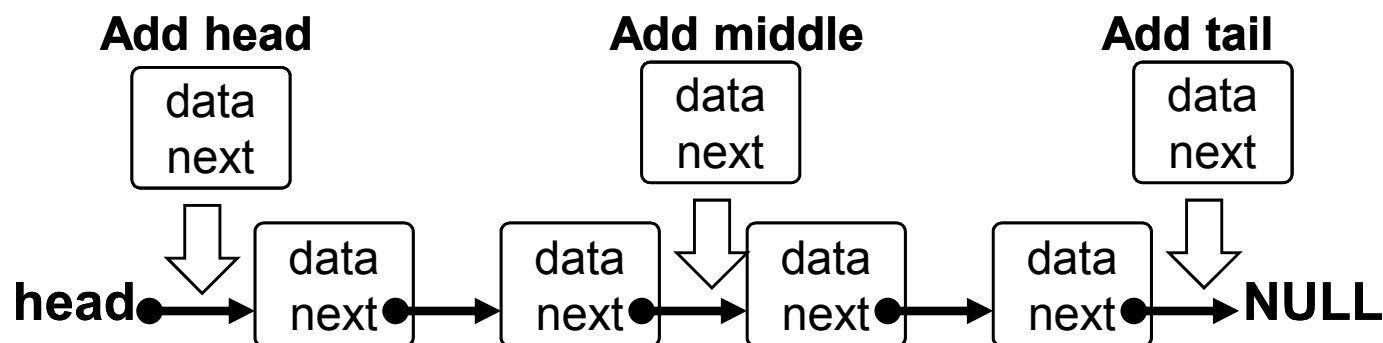


Linked List Operations



■ Add element:

- Add head.
- Add tail.
- Add middle:
 - After i^{th} element.
 - Keep order (ascending).

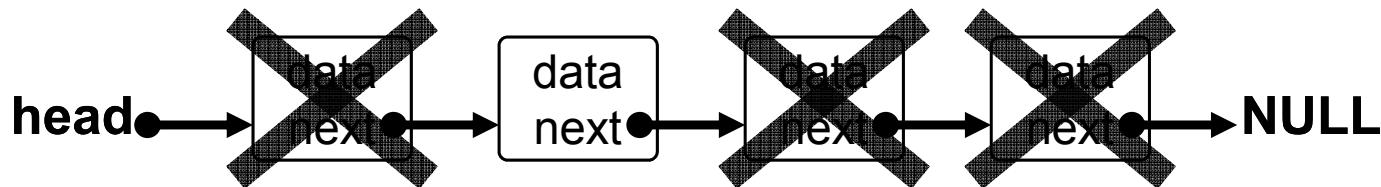


Linked List Operations

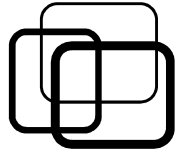


■ Remove element:

- Remove head.
- Remove tail.
- Remove middle:
 - #ith element.
 - Element has data x.
- Remove all.



Contents



- Linked List Concepts.
- Linked List Operations.
- **Linked List Improvement.**

Linked List Improvement



■ Singly linked list vs. Dynamic array:

	Dynamic array	Singly linked List
Organization	Continuous	Dis-continuous
Resize	Require re-allocation Complexity: $O(n)$	No re-allocation Complexity: $O(1)$
Access element	Random access Complexity: $O(1)$	Sequential access Complexity: $O(n)$
Search	Forward/Backward Complexity: $O(n)$	Forward only Complexity: $O(n)$
Add/Remove element	Require element shifting Complexity: $O(n)$	No shifting Complexity: $O(1)$
Memory cost	No extra memory	Require extra memory Cost: $4 * n$ bytes

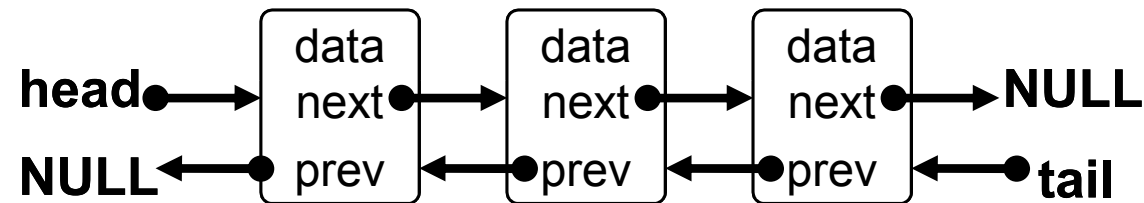
➔ Efficient way to store **sequential** and **variable-size** data.

Linked List Improvement



■ Doubly linked list:

- Element = Data + Link to next + Link to previous.
- Head: forward traverse.
- Tail: backward traverse.



■ C declaration:

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

```
struct DList
{
    DNode *head;
    DNode *tail;
};
```

Linked List Improvement



■ Doubly linked list vs. Dynamic array:

	Dynamic array	Doubly linked list
Organization	Continuous	Dis-continuous
Resize	Require re-allocation Complexity: $O(n)$	No re-allocation Complexity: $O(1)$
Access element	Random access Complexity: $O(1)$	Sequential access Complexity: $O(n)$
Search	Forward/Backward Complexity: $O(n)$	Forward/Backward Complexity: $O(n)$
Add/Remove element	Require element shifting Complexity: $O(n)$	No shifting Complexity: $O(1)$
Memory cost	No extra memory	Require extra memory Cost: $8 * n$ bytes

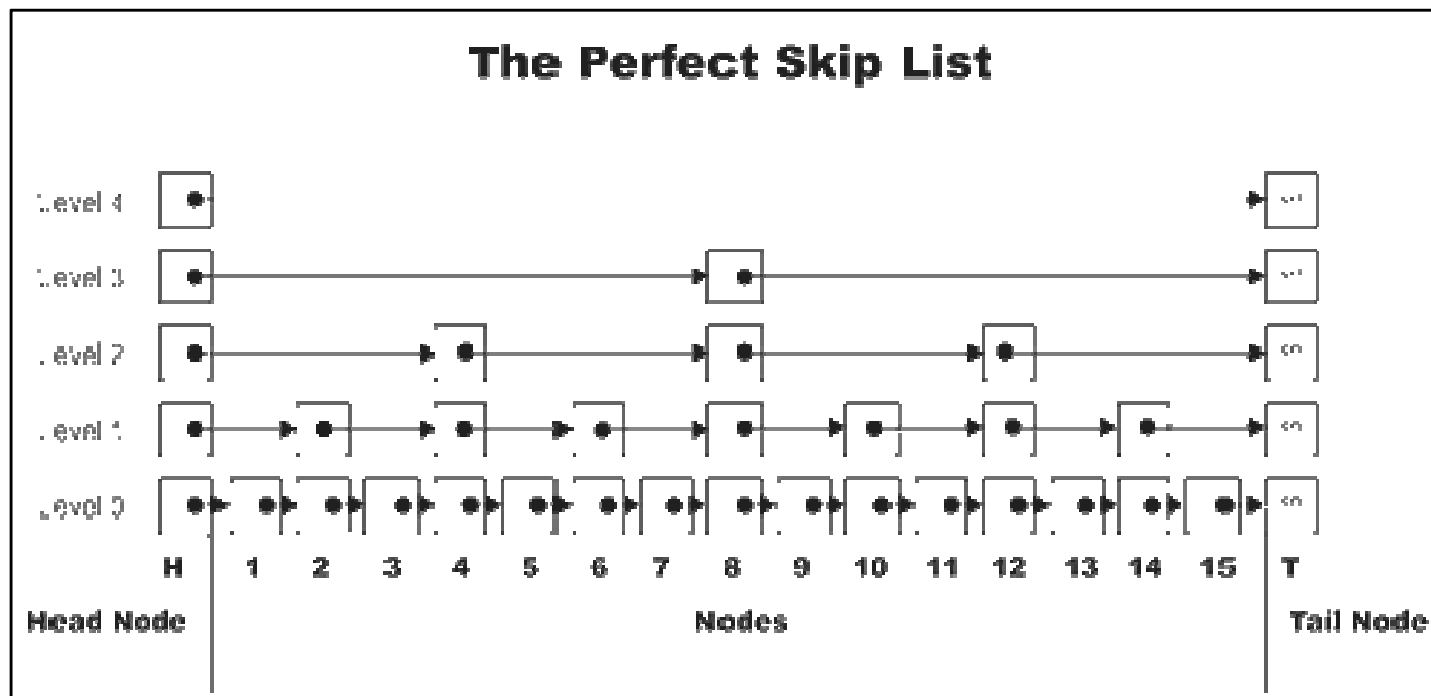
➔ Good for ***two-directional traverse, variable-size*** data.

Linked List Improvement



■ Skip list:

- Set of singly linked lists organized in layers.
- Lower layer: fine-grained nodes, slower “lane”.
- Higher layer: coarse-grained nodes, faster “lane”.



Linked List Improvement



■ Skip list vs. Ordered array:

	Ordered Array	Skip List
Organization	Continuous	Dis-continuous
Resize	Require re-allocation Complexity: $O(n)$	No re-allocation Complexity: $O(1)$
<i>Access element</i>	<i>Random access</i> Complexity: $O(1)$	<i>Sequential access</i> Complexity: $O(\log(n))$
<i>Binary search</i>	<i>Forward/Backward</i> Complexity: $O(n \cdot \log(n))$	<i>Forward only</i> Complexity: $O(n \cdot \log(n))$
Add/Remove element	Require element shifting Complexity: $O(n)$	No shifting Complexity: $O(1)$
Memory cost	No extra memory	Require extra memory Cost: $\log(n) * 4 * n$ bytes

➔ Efficient way to store **variable-size** and **ordered** data.

Summary



- Singly linked list concepts:
 - Dis-continuous storage.
 - Node = data + next.
 - Last node points to NULL.
- Singly linked list operations:
 - Initialize, check empty.
 - Find, add, remove.
- Singly linked list improvements:
 - Doubly linked list.
 - Circularly linked list.

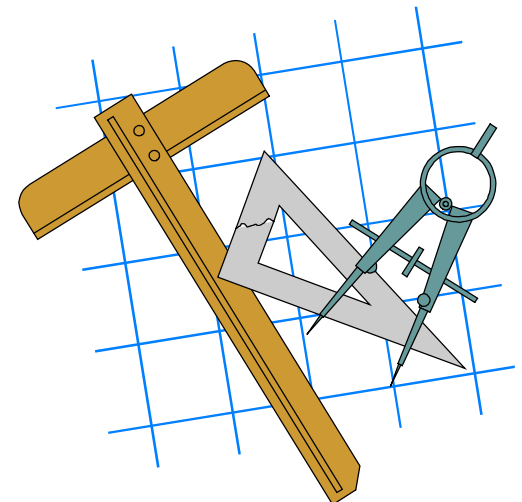


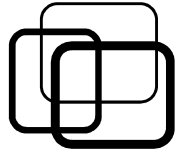
Practice



■ Practice 2.1:

Write C/C++ program to implement operations on Circularly linked lists.





■ Practice 2.2:

Write C/C++ program to implements the following operations on Singly linked list:

- Count nodes in list.
- Reverse list.
- Add new node (keep order).

