[SWE2015-41] Introduction to Data Structures (자료구조개론)

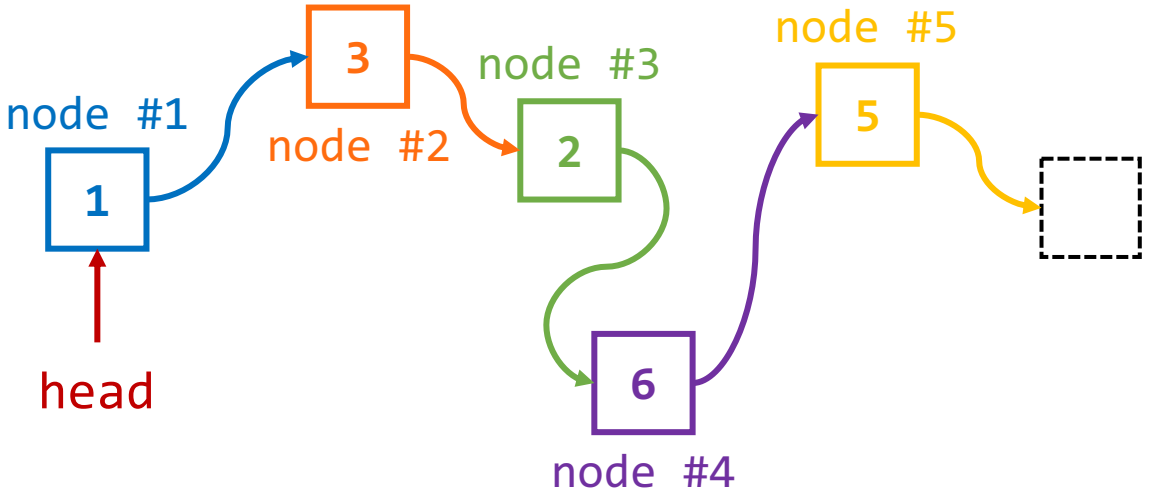
Variants of Linked Lists

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) Linked Lists

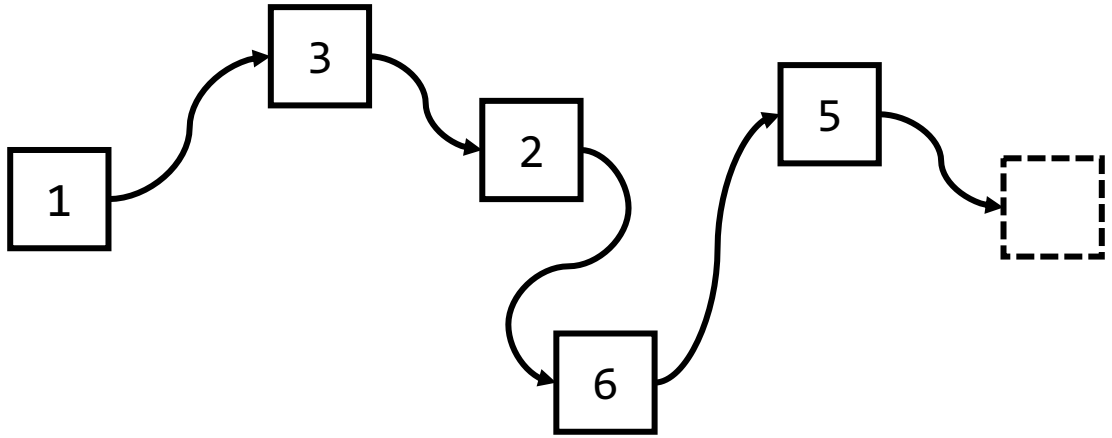
- **A linked list** is a collection of **sequentially-connected** elements
  - The elements are not required to be stored in contiguous memory




```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

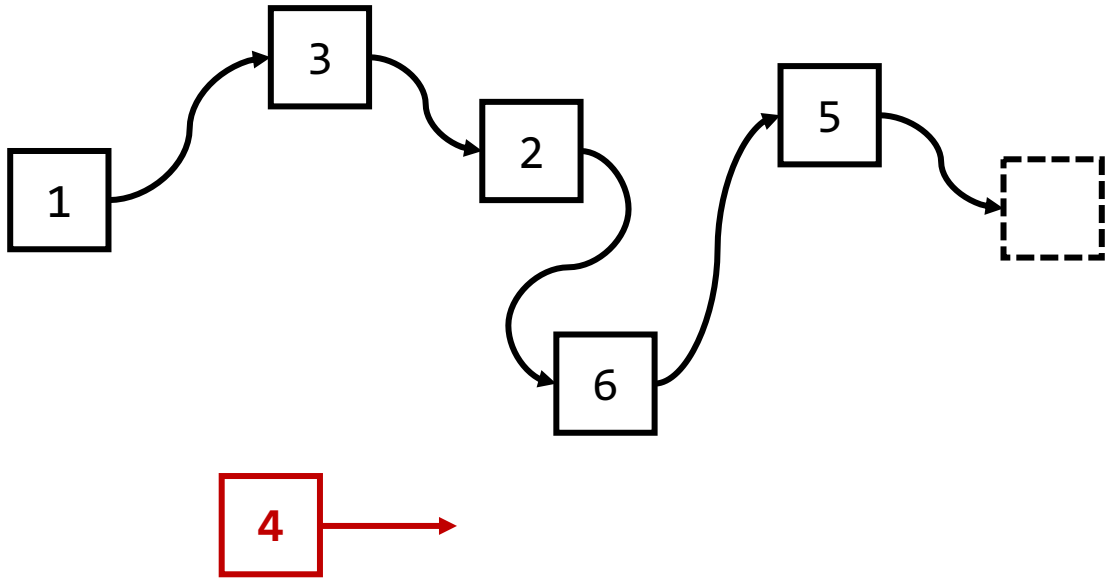
(Recap) Linked Lists - Insertion

- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



(Recap) Linked Lists - Insertion

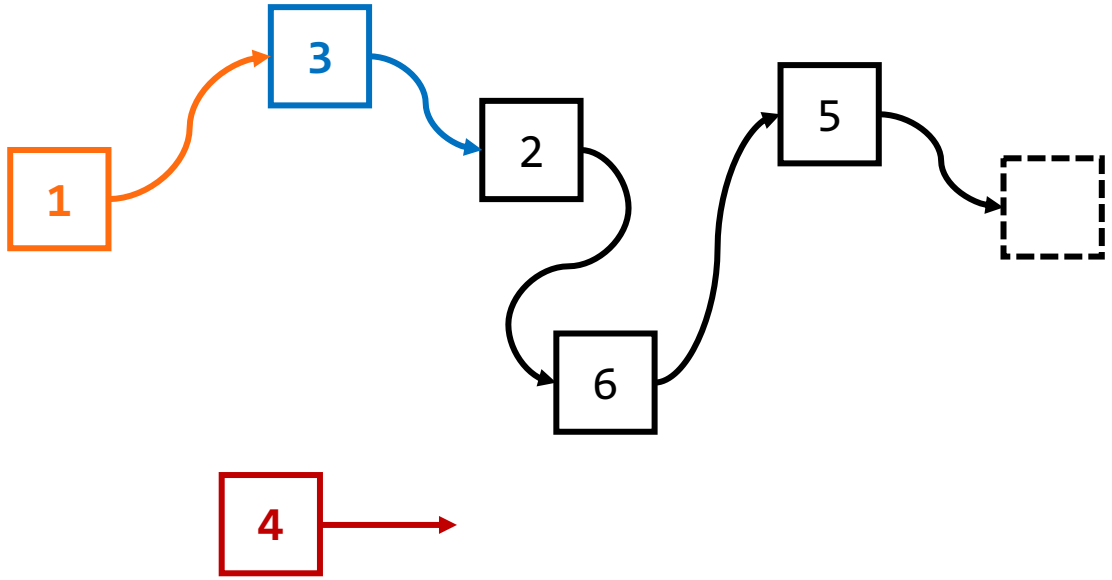
- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



1. Create new node

(Recap) Linked Lists - Insertion

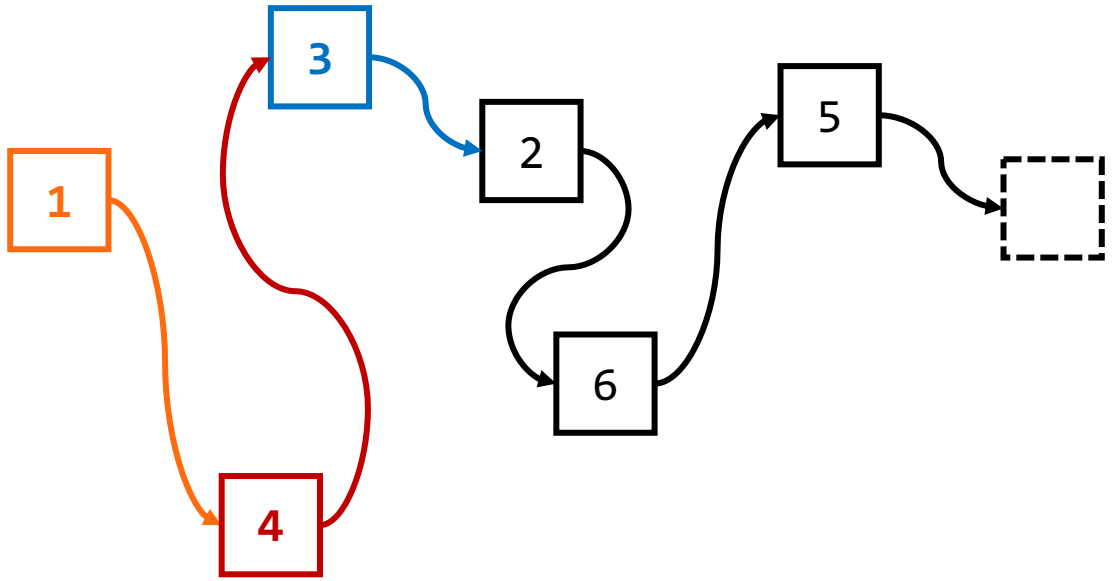
- How to **insert** an item at the middle of the linked list?
  - Example: Insert "4" at the 2nd position



1. Create new node
2. Find the 1st node and its next node

(Recap) Linked Lists - Insertion

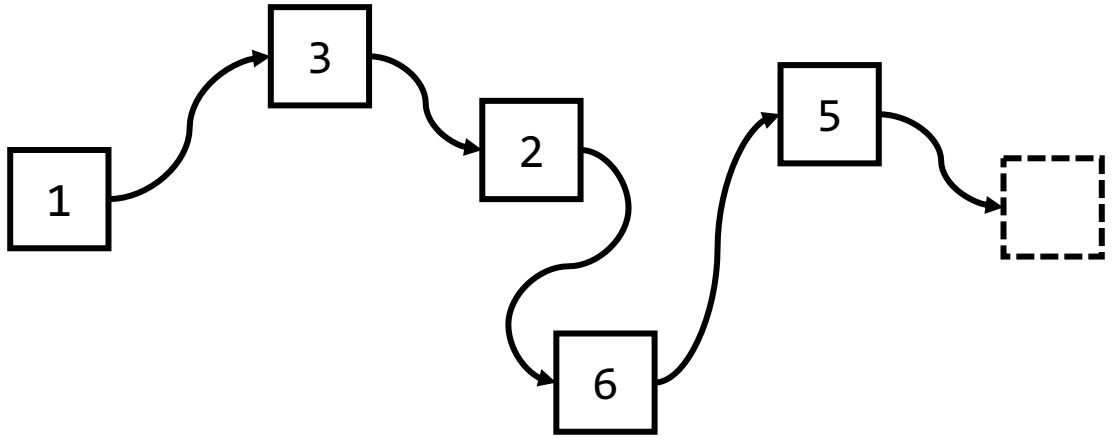
- How to **insert** an item at the middle of the linked list?
  - Example: Insert “4” at the 2nd position



1. Create new node
2. Find the 1st node and its next node
3. Connect them: node → node → node

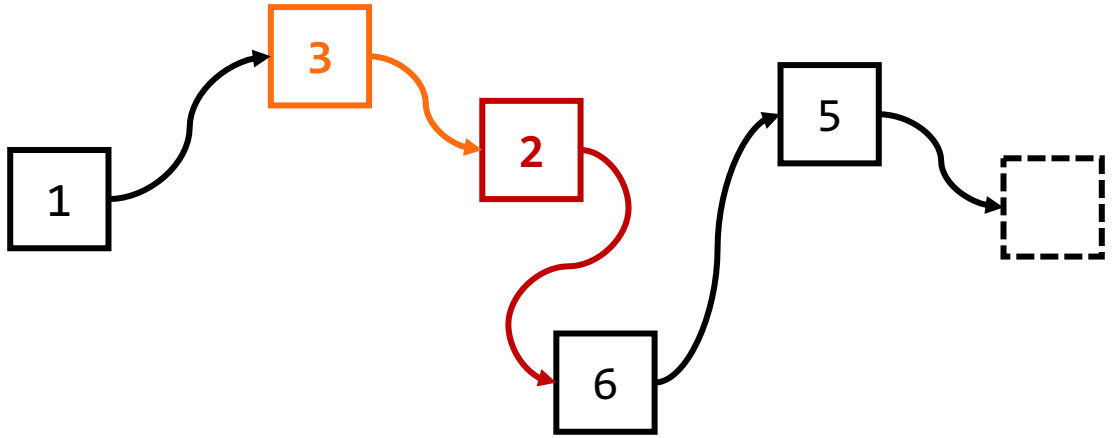
(Recap) Linked Lists - Deletion

- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



(Recap) Linked Lists - Deletion

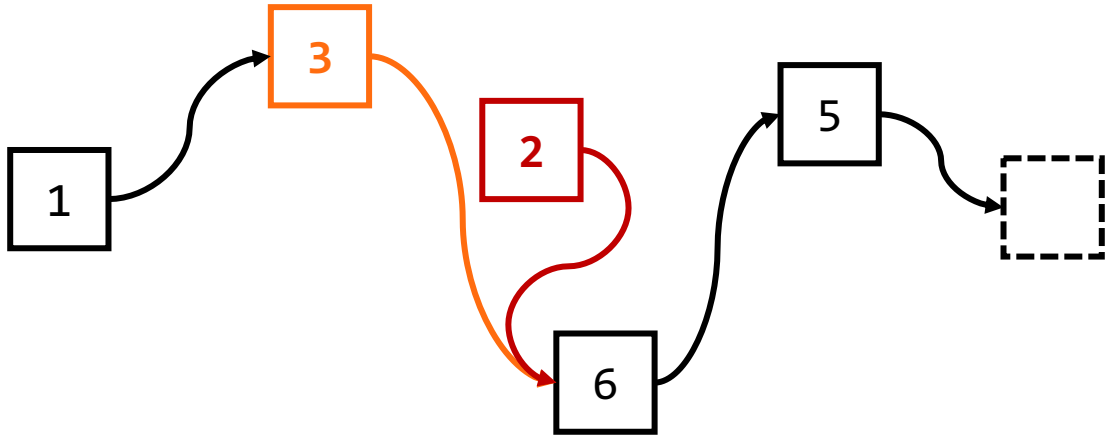
- How to **delete** an item from the linked list?
  - Example: delete “2” at the 3rd position



1. Find the 2nd node and its next node

(Recap) Linked Lists - Deletion

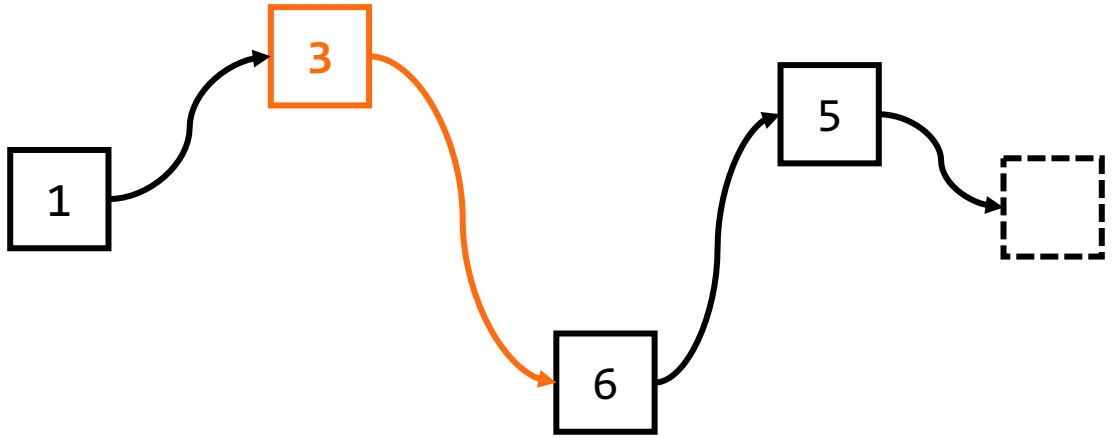
- How to **delete** an item from the linked list?
  - Example: delete "2" at the 3rd position



1. Find the 2nd node and its next node
2. Skip the connection between node and node

(Recap) Linked Lists - Deletion

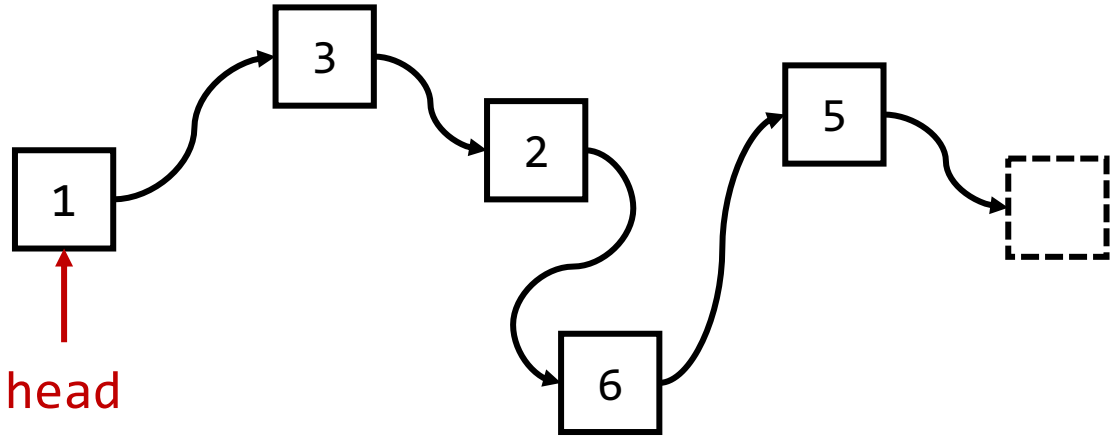
- How to **delete** an item from the linked list?
  - Example: delete "2" at the 3rd position



1. Find the 2nd node and its next node
2. Skip the connection between node and node
3. Free the node's memory

Variants of Linked Lists

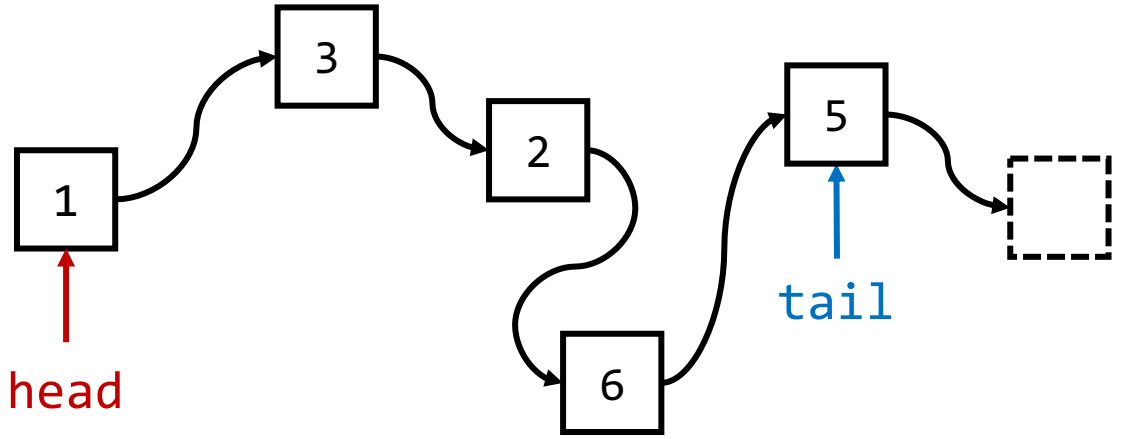
- Singly Linked Lists
  - The simplest linked list structure
  - One-directional → cannot go backward
  - $O(n)$ for insertion at the end



Variants of Linked Lists

- Singly Linked Lists with the tail pointer
  - The simplest linked list structure
  - One-directional → cannot go backward
  - $O(1)$ for insertion at the end

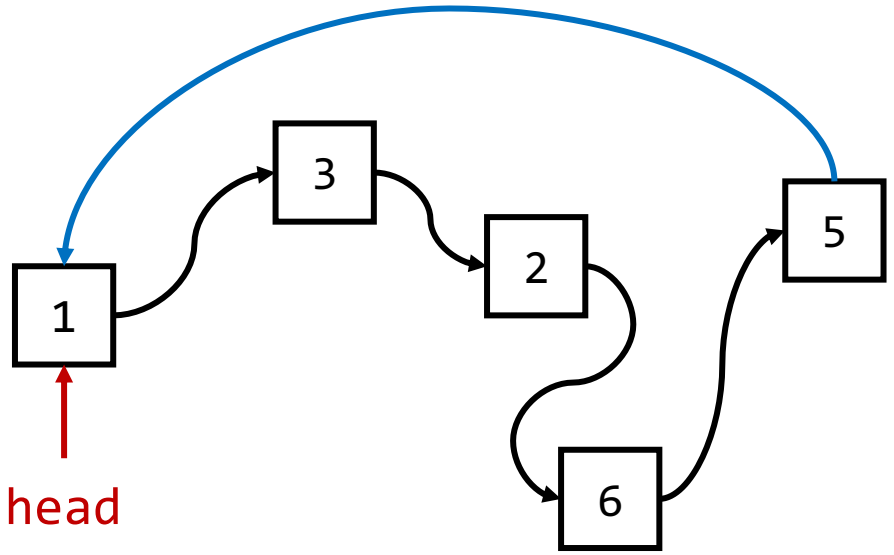
```
typedef struct _LinkedList {
    Node *head, *tail;
} LinkedList;
```



Variants of Linked Lists

- Circular Linked Lists
  - Every node has non-`NULL` `next` pointer → makes handling corner cases easier

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;
```

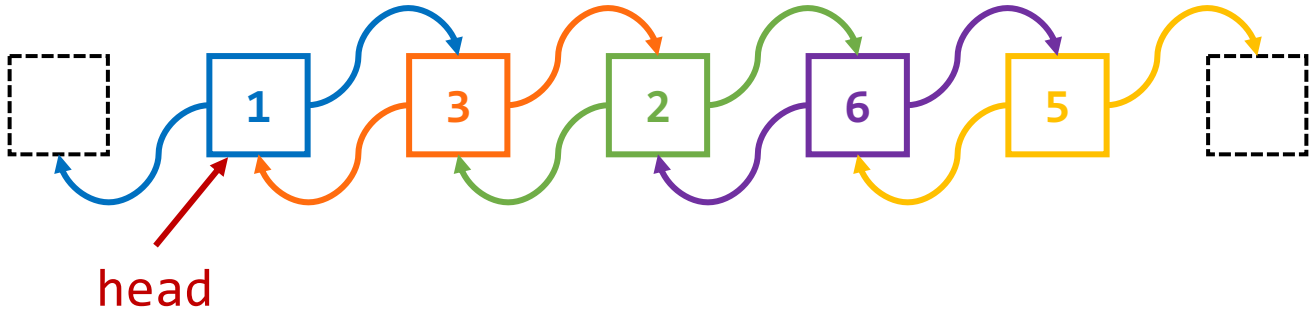


Variants of Linked Lists

- Doubly Linked Lists
  - Every node has `prev` and `next` pointers to point previous and next nodes

```
typedef struct _Node {
    int value;
    struct _Node *prev, *next;
} Node;
```

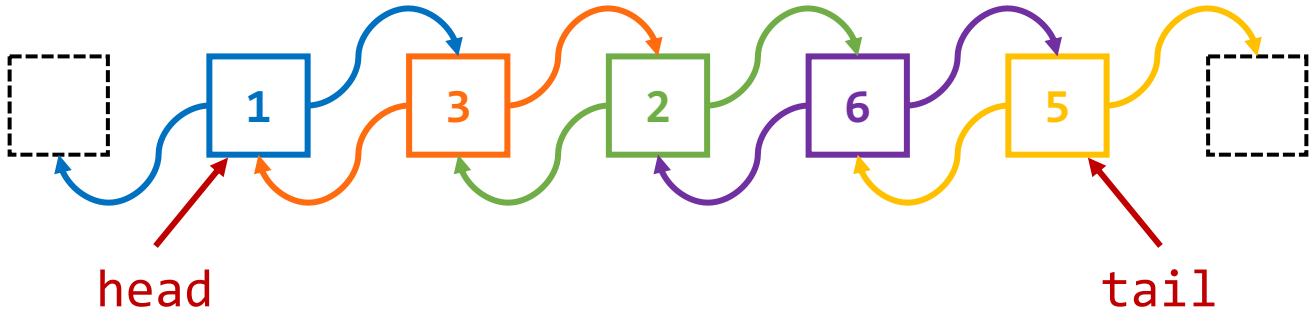
  - Two-directional → can move forward and backward



Doubly Linked Lists - Implementation

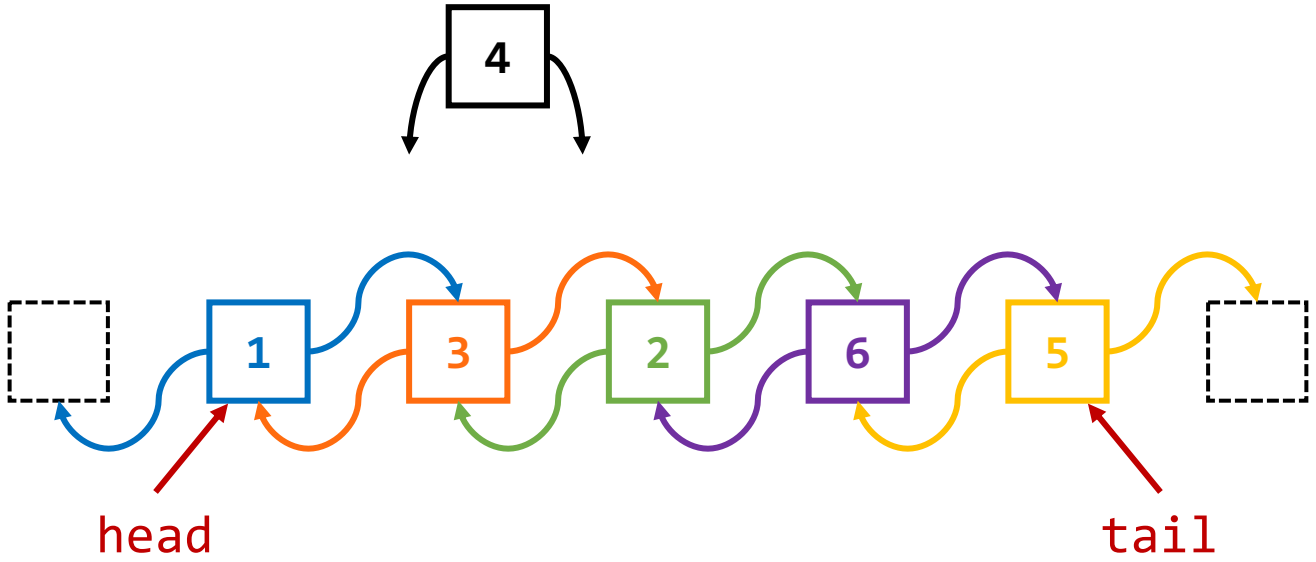
- Let's implement a doubly linked list with the tail pointer

```
typedef struct _Node {
    int value;
    struct _Node *prev, *next;
} Node;

typedef struct _LinkedList {
    Node *head, *tail;
} LinkedList;

void insert(LinkedList *list, int item, int index);
void delete(LinkedList *list, int index);
```



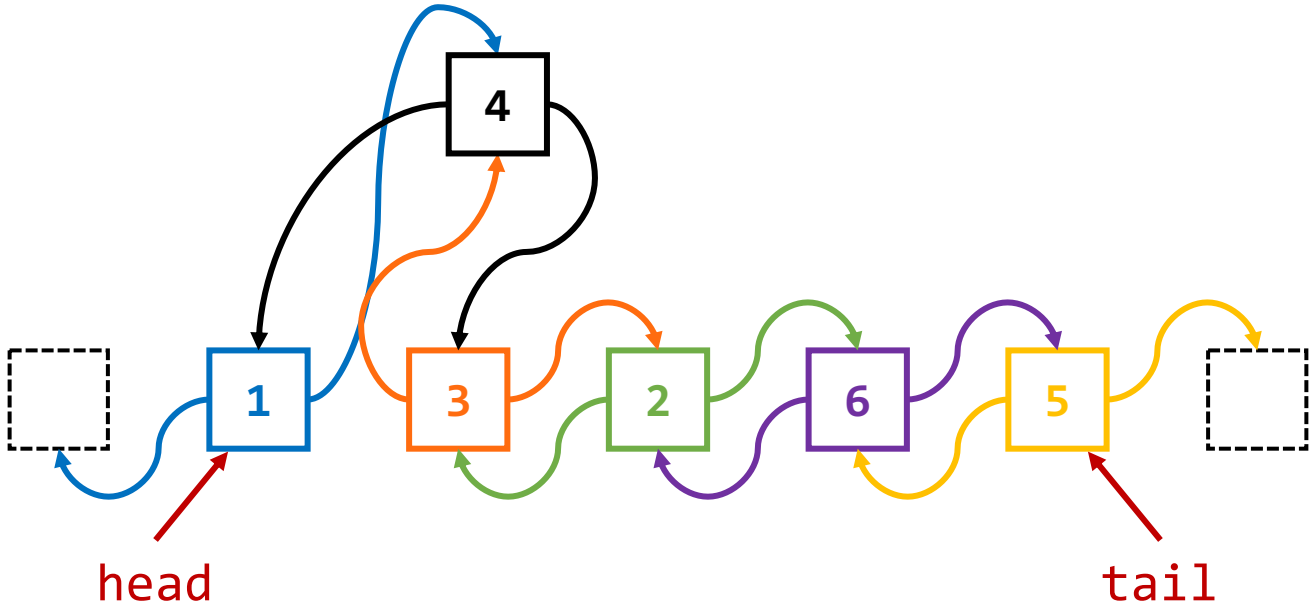
Doubly Linked Lists - Implementation

- Let's implement a doubly linked list with the tail pointer
  - How to insert an element **4** between two nodes **1** and **3**?



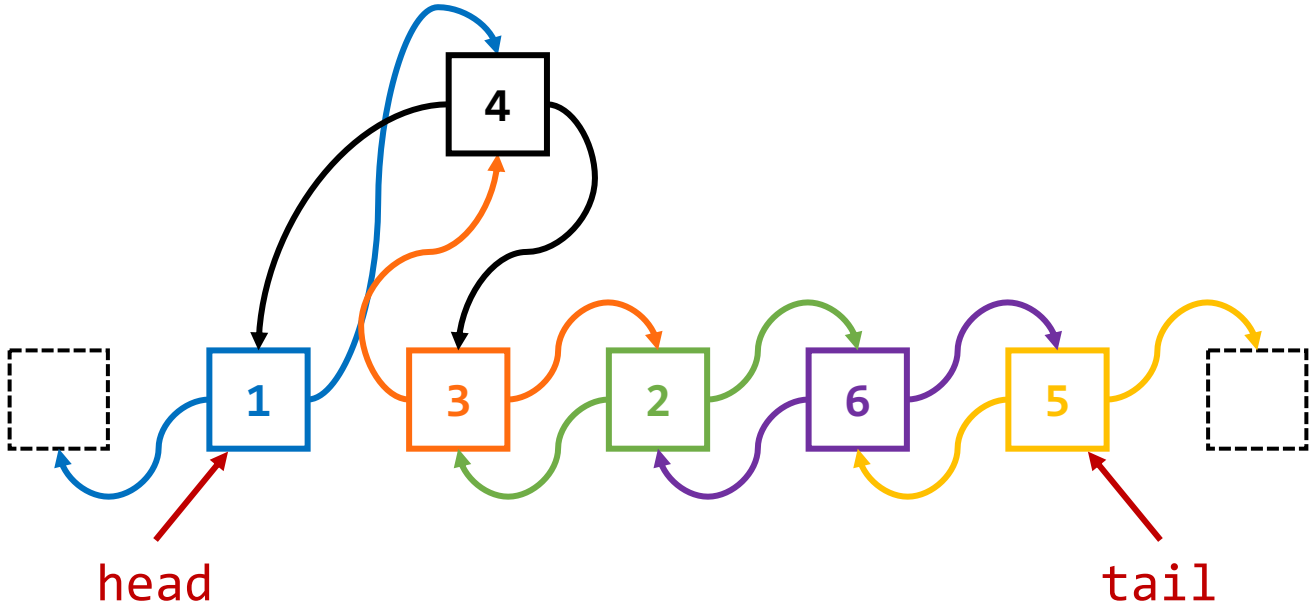
Doubly Linked Lists - Implementation

- Let's implement a doubly linked list with the tail pointer
  - How to insert an element **4** between two nodes **1** and **3**?



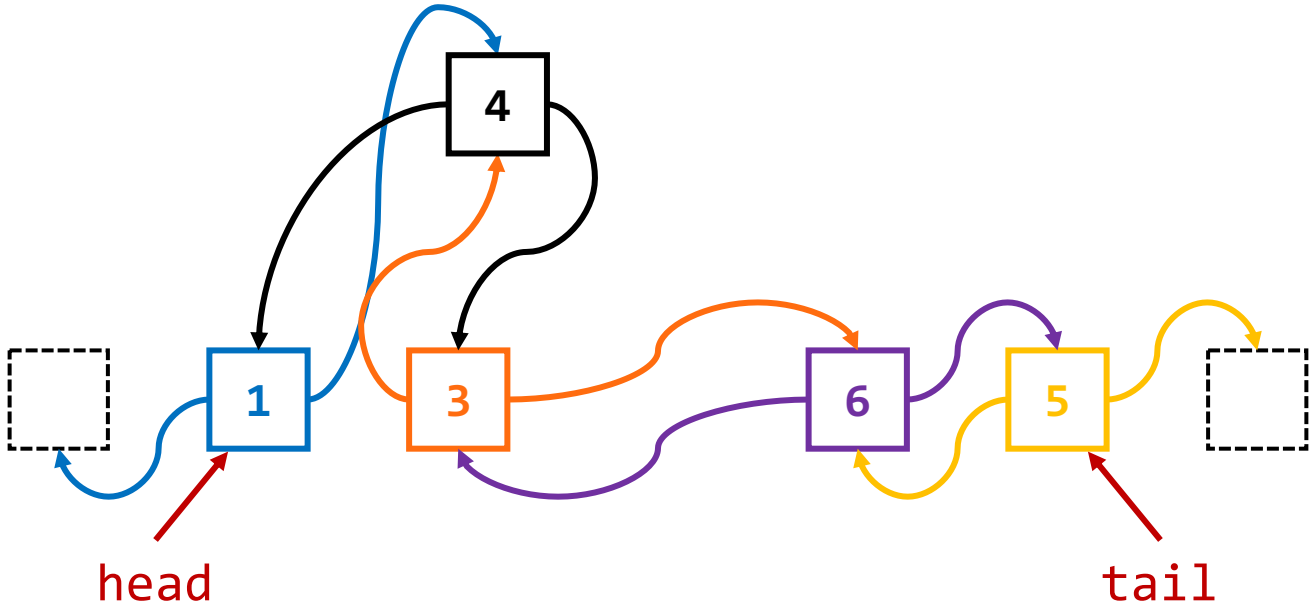
Doubly Linked Lists - Implementation

- Let's implement a doubly linked list with the tail pointer
  - How to insert an element **4** between two nodes **1** and **3**?
  - How to delete the element **2**?



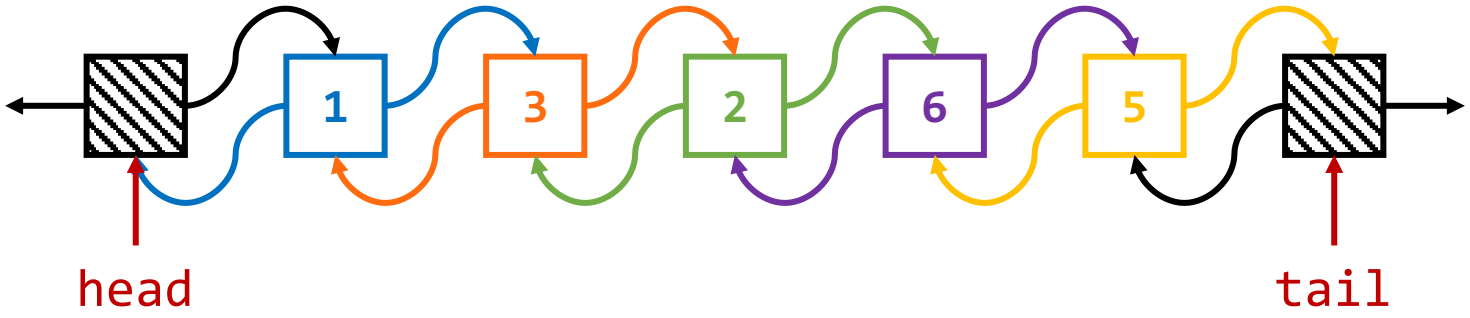
Doubly Linked Lists - Implementation

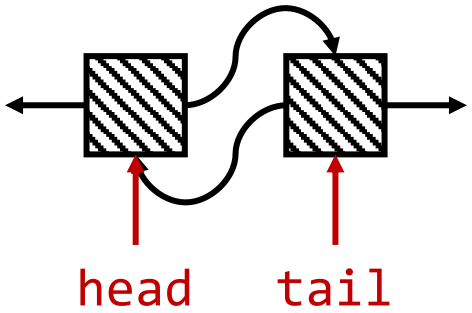
- Let's implement a doubly linked list with the tail pointer
  - How to insert an element **4** between two nodes **1** and **3**?
  - How to delete the element **2**?



Doubly Linked Lists - Implementation

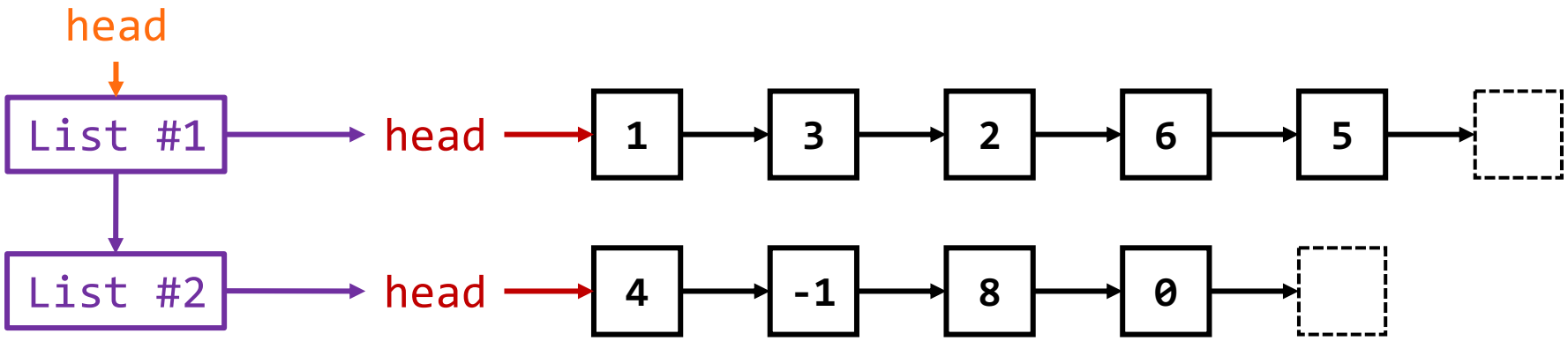
- A technique for easy implementation: **Dummy nodes**
  - This guarantees that `head` and `tail` are always non-`NULL` even if the list is empty
  - This guarantees that intermediate nodes have non-`NULL` `prev` and `next` pointers





Variants of Linked Lists

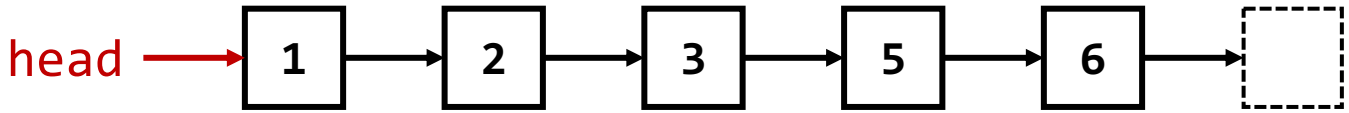
- Two-dimensional Linked Lists (like 2D arrays)




```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList {  // This represents a "list" node
    Node *head;
    struct _LinkedList *next; // This connects two linked lists
} LinkedList;

typedef struct _LinkedList2D {
    LinkedList *head;
} LinkedList2D;
```

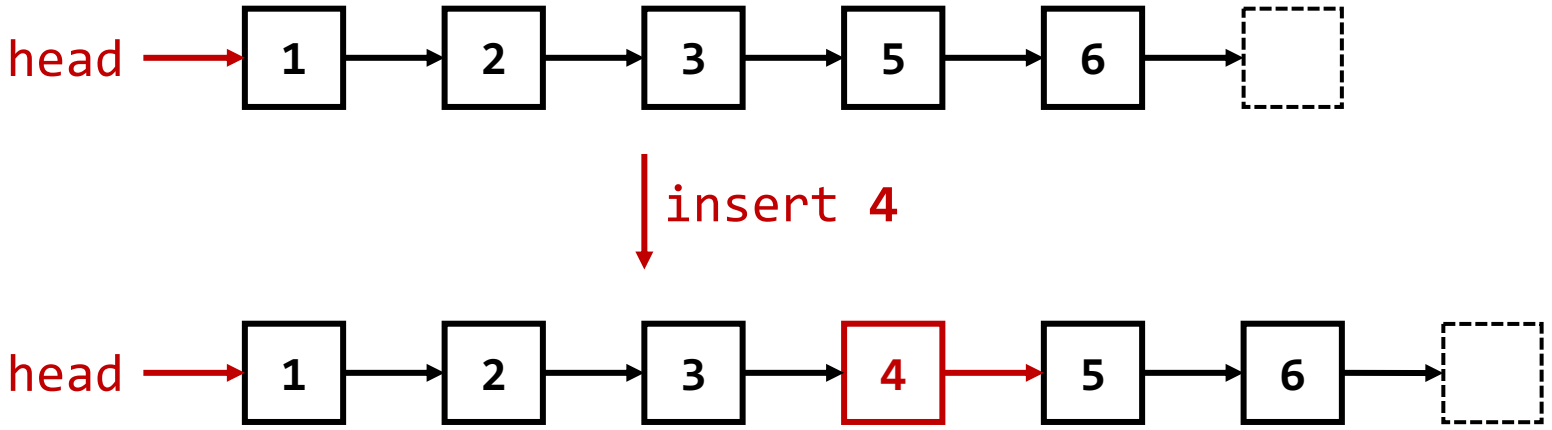
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



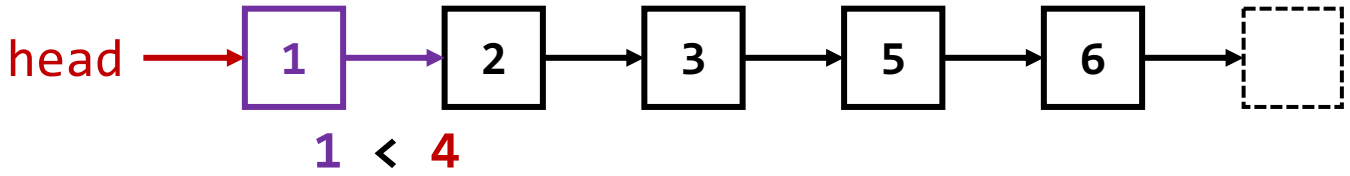
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



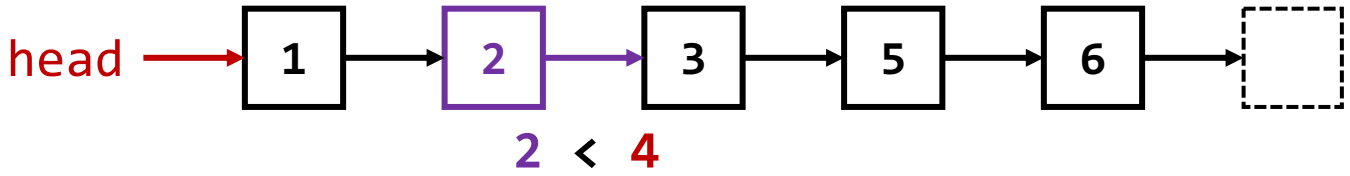
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



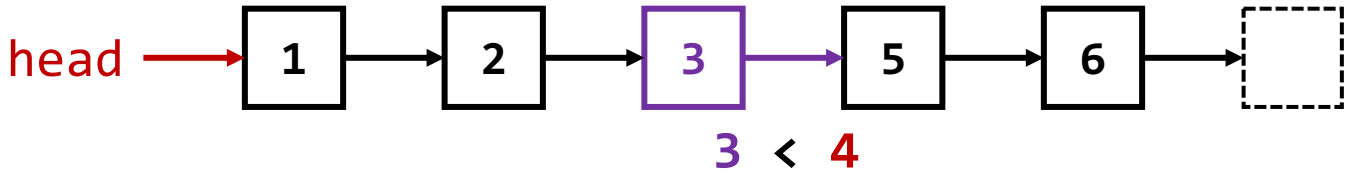
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



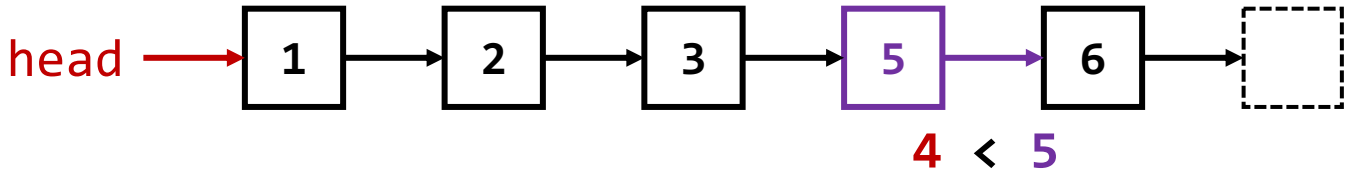
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



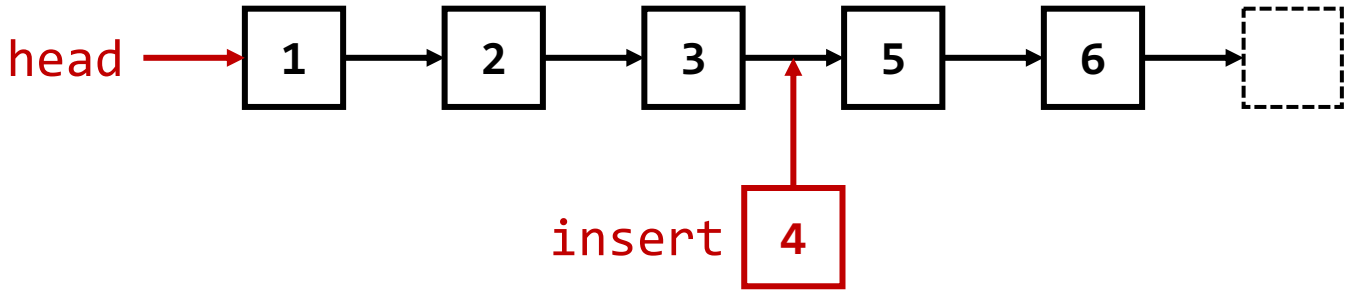
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



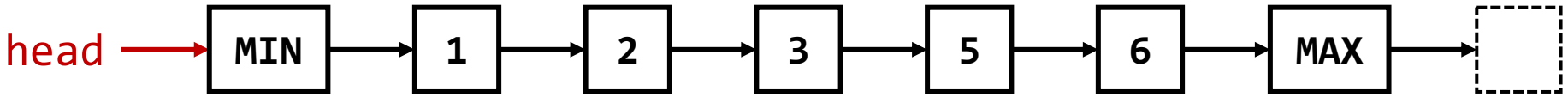
Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted

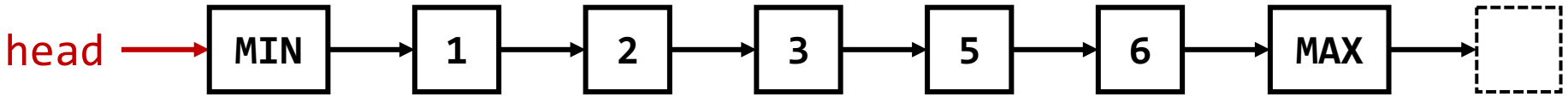


  - For easy implementation, one can consider dummy nodes, `MIN` and `MAX`

```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void search(LinkedList *list, int value);
void insert(LinkedList *list, int value);
void delete(LinkedList *list, int value);
```

Variants of Linked Lists

- Sorted Linked Lists
  - A linked list whose elements should be always sorted



  - For easy implementation, one can consider dummy nodes, `MIN` and `MAX`

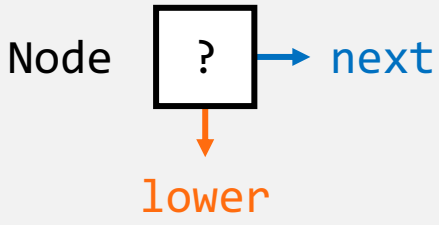
```
typedef struct _Node { int value; struct _Node *next; } Node;
typedef struct _LinkedList { Node *head; } LinkedList;

void search(LinkedList *list, int value);
void insert(LinkedList *list, int value);
void delete(LinkedList *list, int value);
```

  - The search, insertion, deletion operations require $O(n)$ time complexity

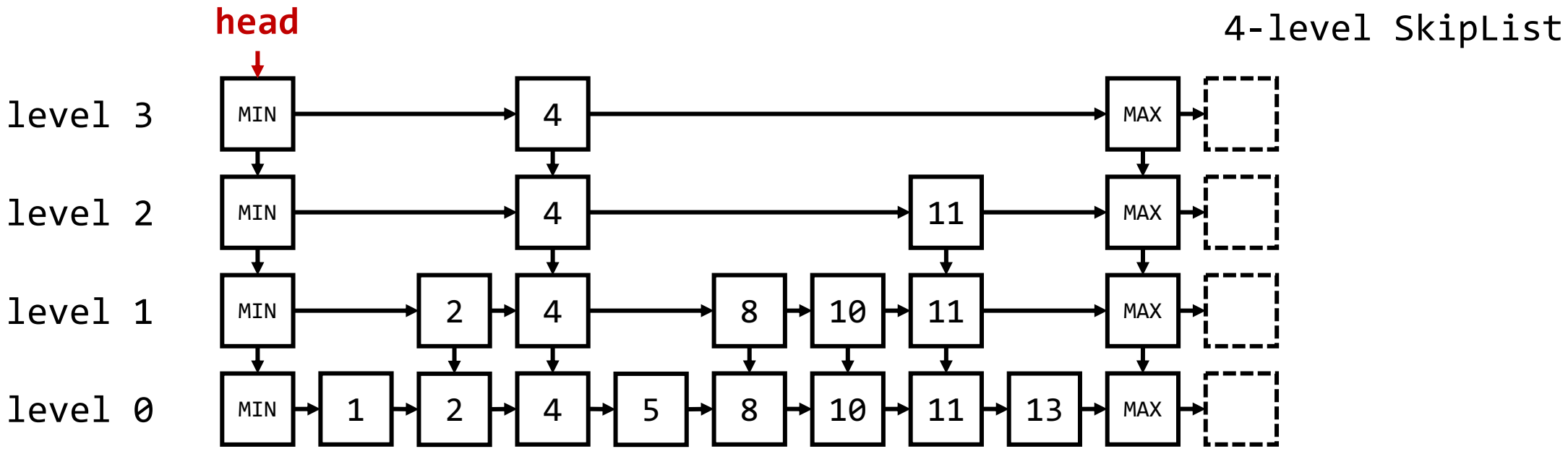
Variants of Linked Lists - Skip List

- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list

```
typedef struct _Node {
    int value;
    struct _Node *next, *lower;
} Node;
```

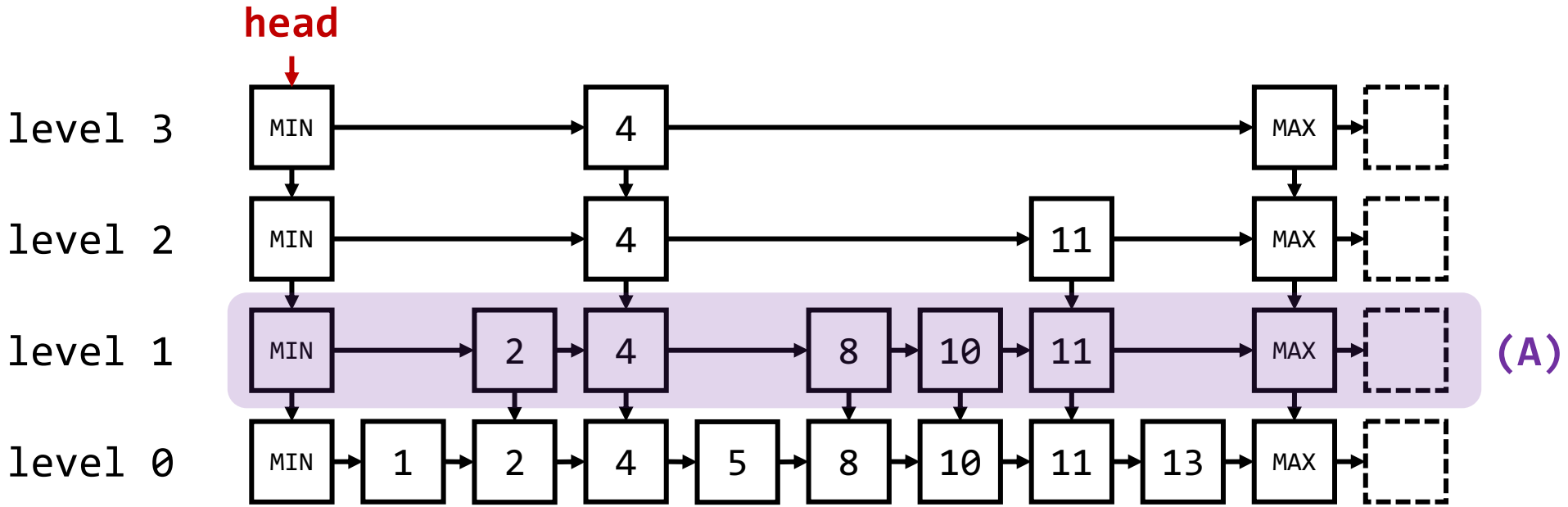



```
typedef struct _SkipList {
    Node *head;
    int num_levels;
} SkipList;
```



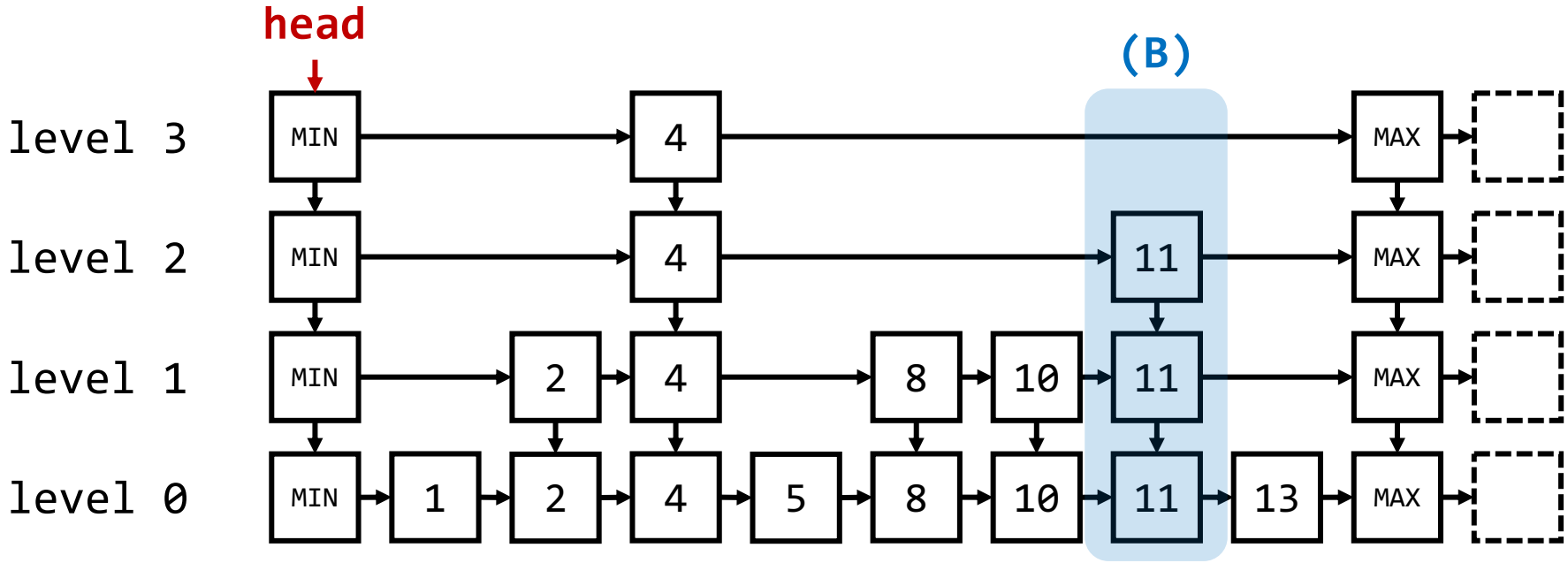
Variants of Linked Lists - Skip List

- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list
  - Two properties:
    (A) Elements at each level (i.e., row) are sorted



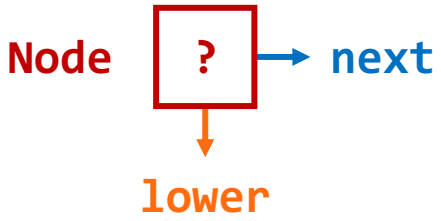
Variants of Linked Lists - Skip List

- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list
  - Two properties:
    - **(A)** Elements at each level (i.e., row) are sorted
    - **(B)** Elements at each column exist consecutively

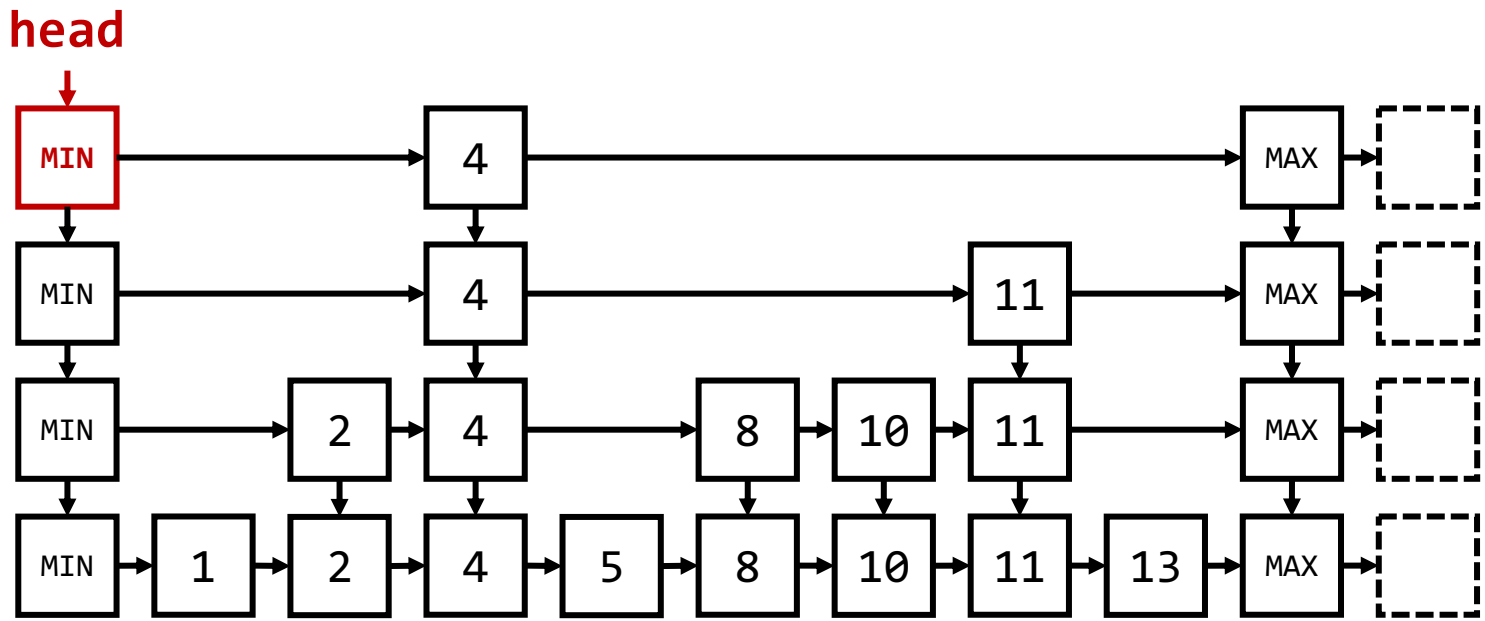


Variants of Linked Lists - Skip List

- The search procedure in **Skip Lists**
  1. Starting from **head**,

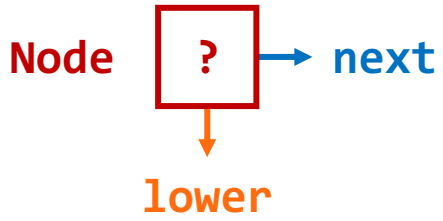


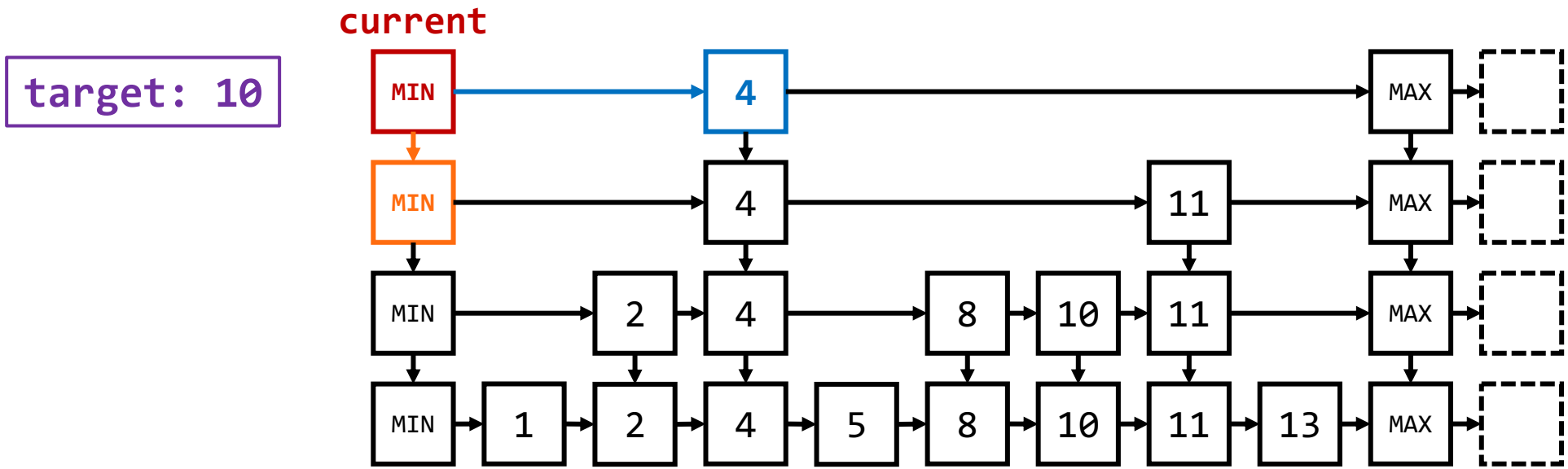
target: 10



Variants of Linked Lists - Skip List

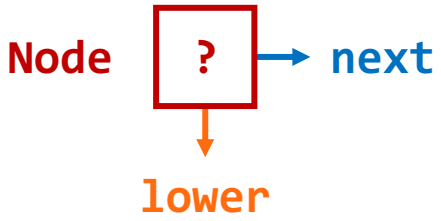
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node

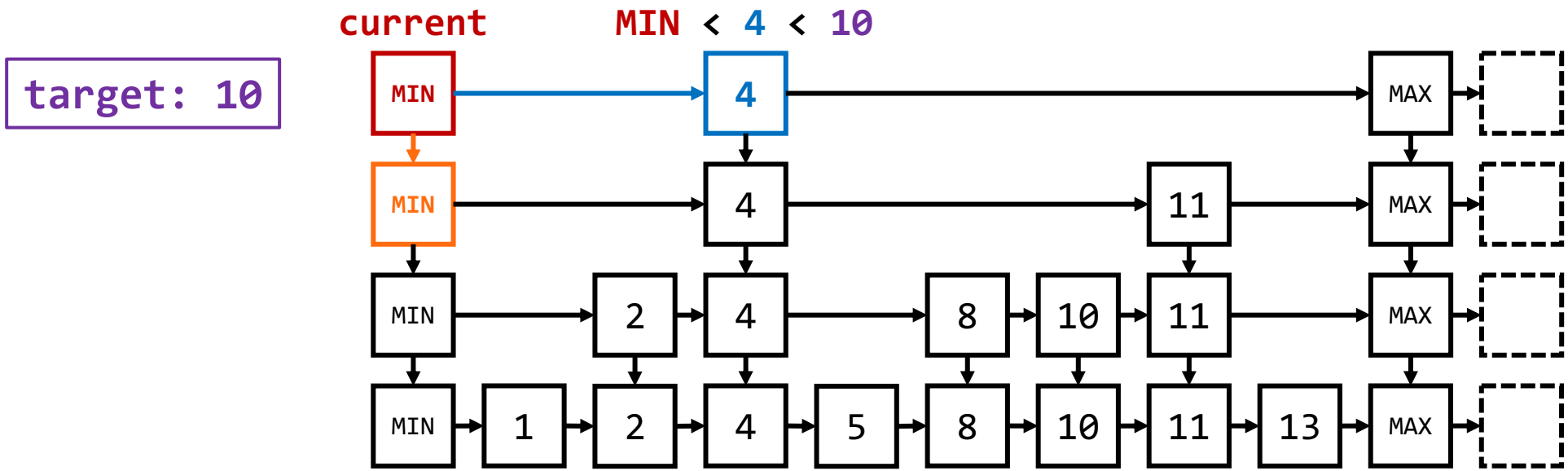




Variants of Linked Lists - Skip List

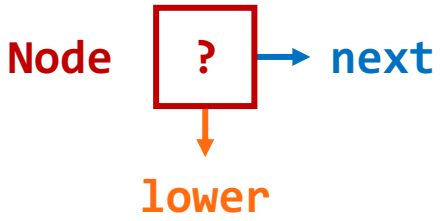
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node

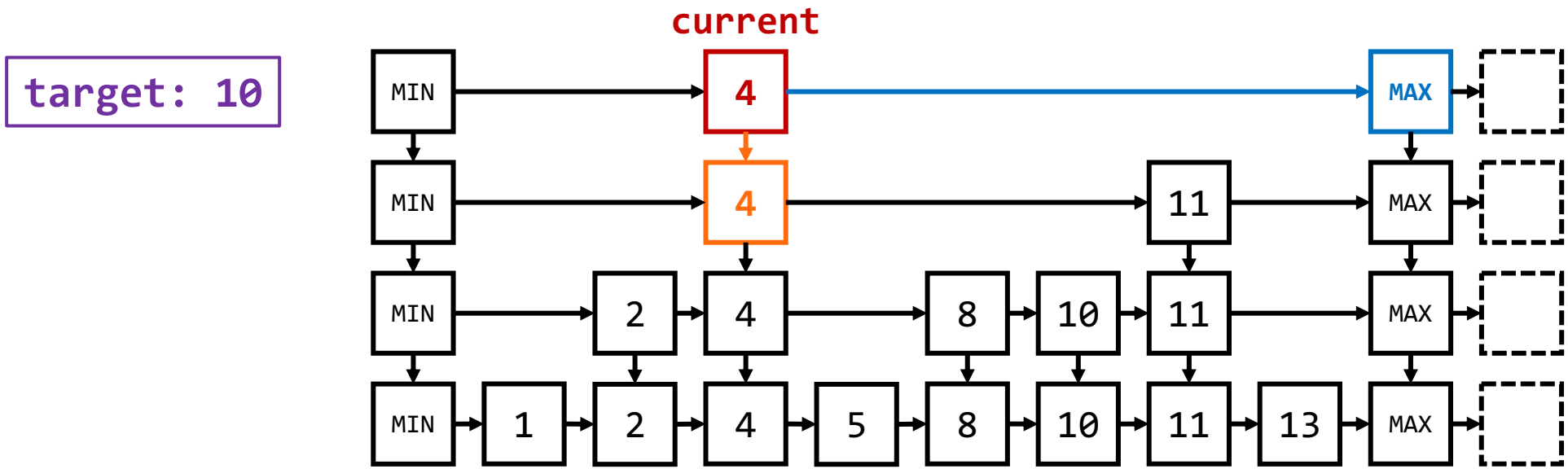




Variants of Linked Lists - Skip List

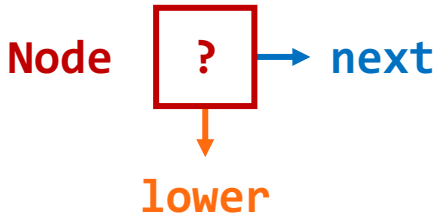
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node

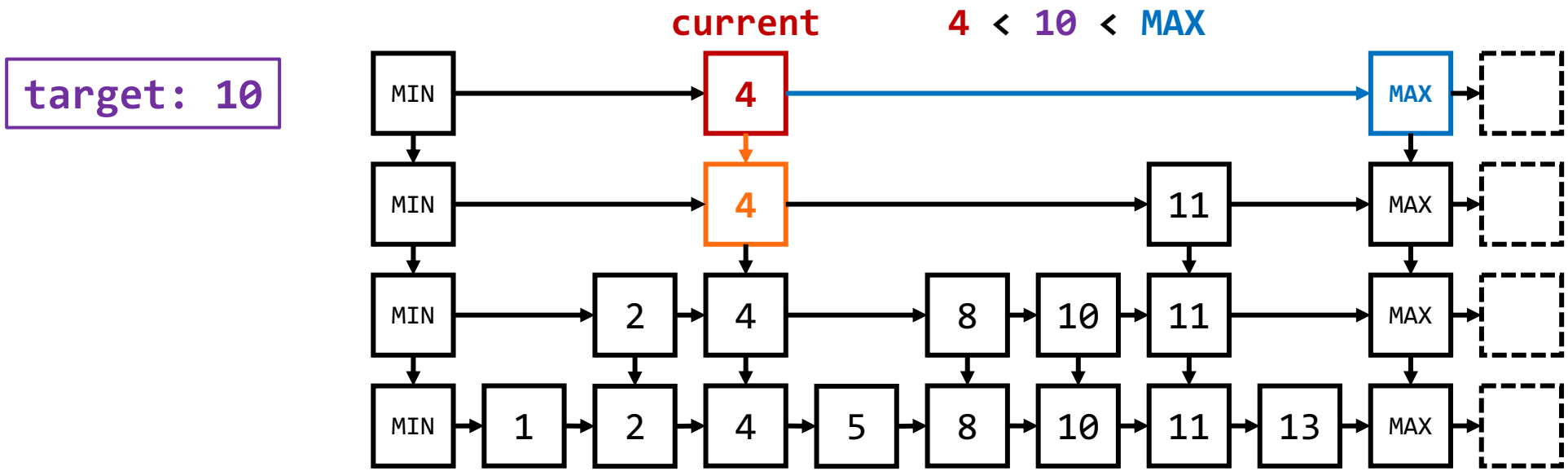




Variants of Linked Lists - Skip List

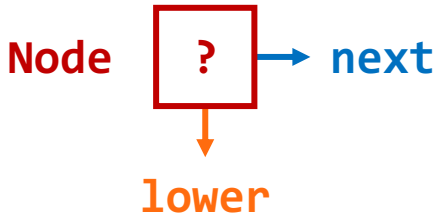
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node



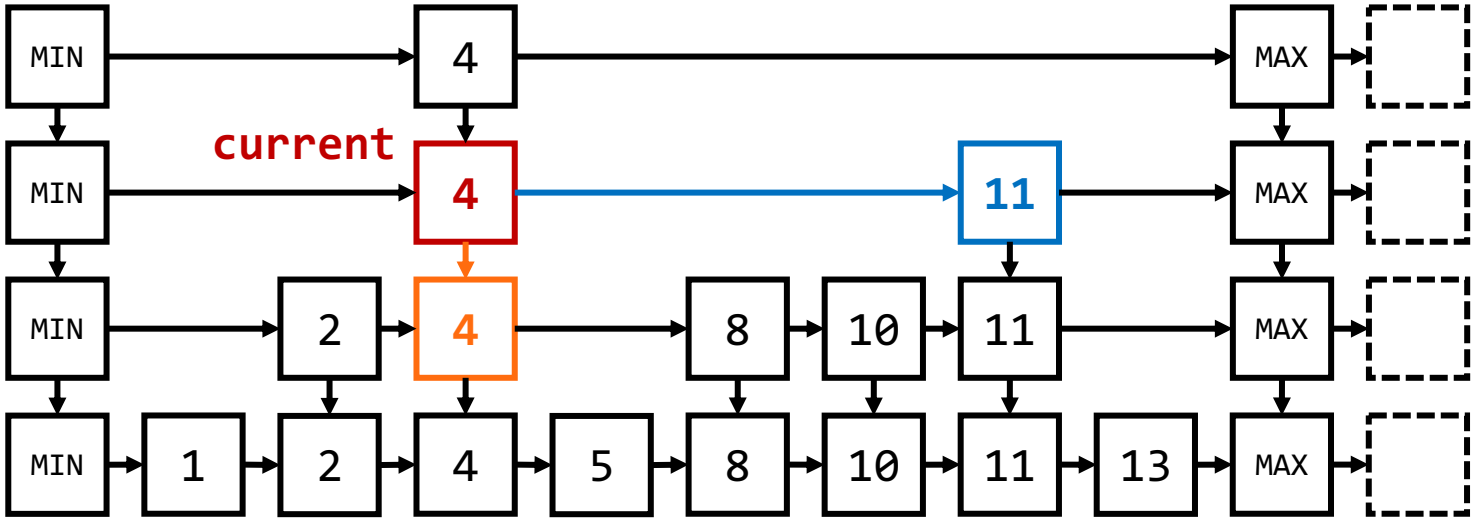


Variants of Linked Lists - Skip List

- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node

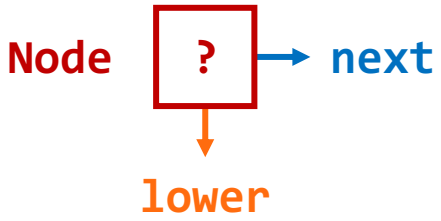


target: 10

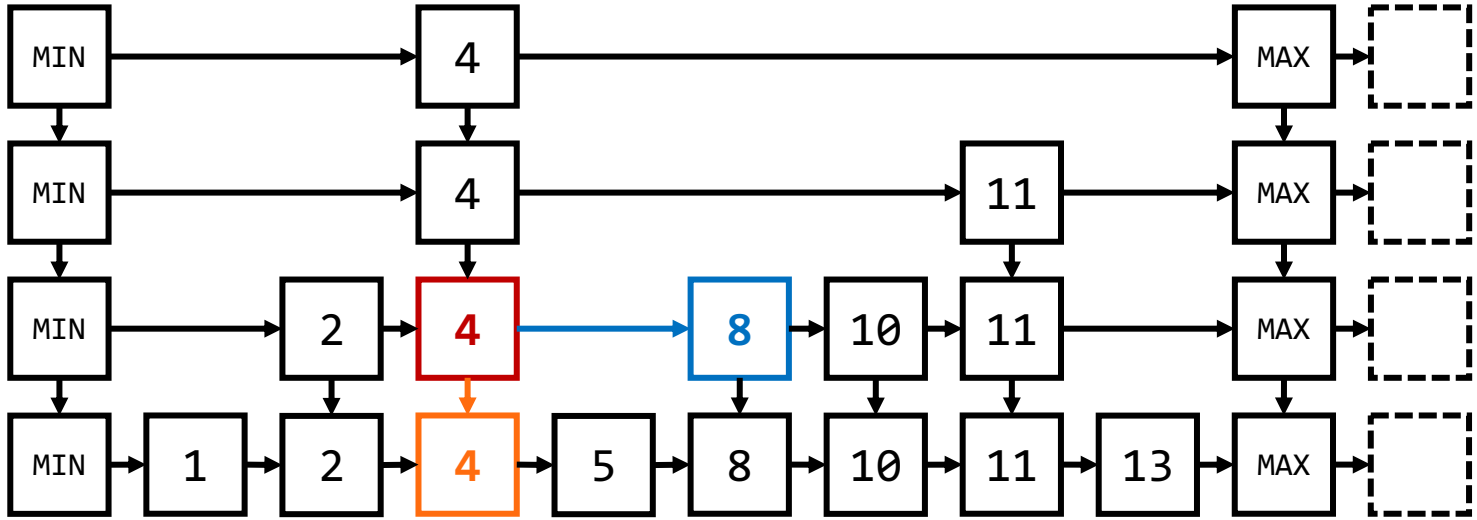


Variants of Linked Lists - Skip List

- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node

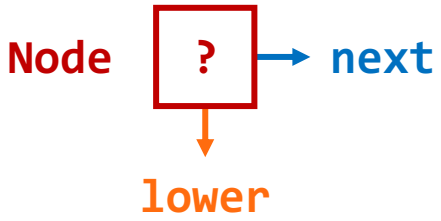


target: 10

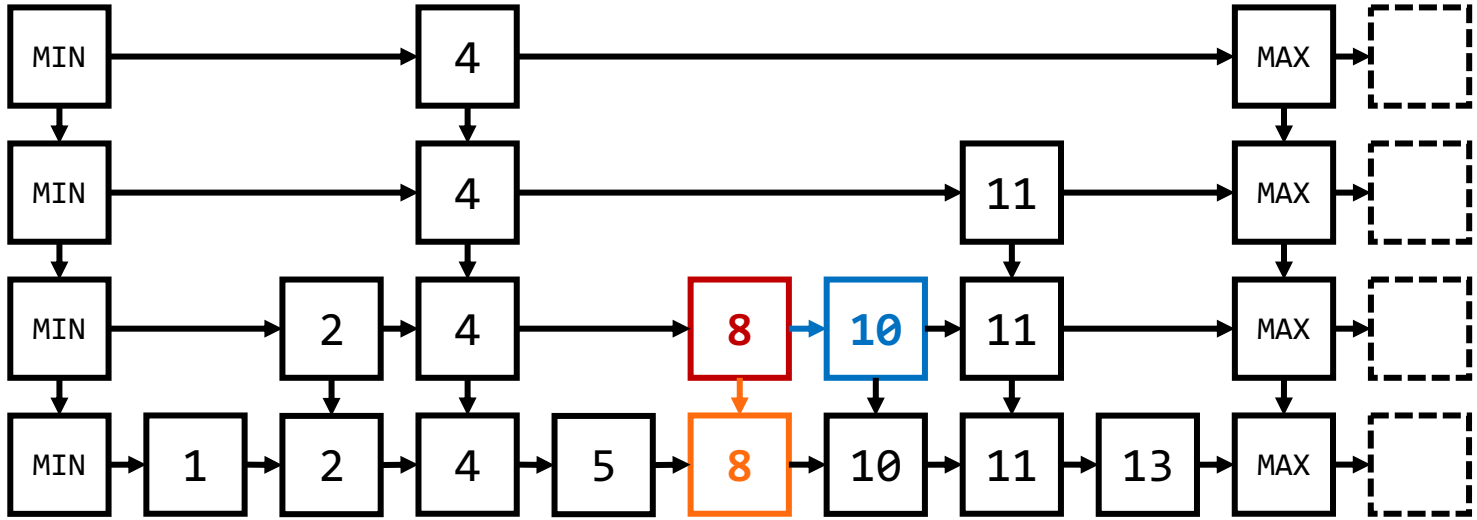


Variants of Linked Lists - Skip List

- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node

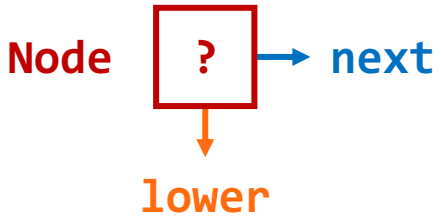


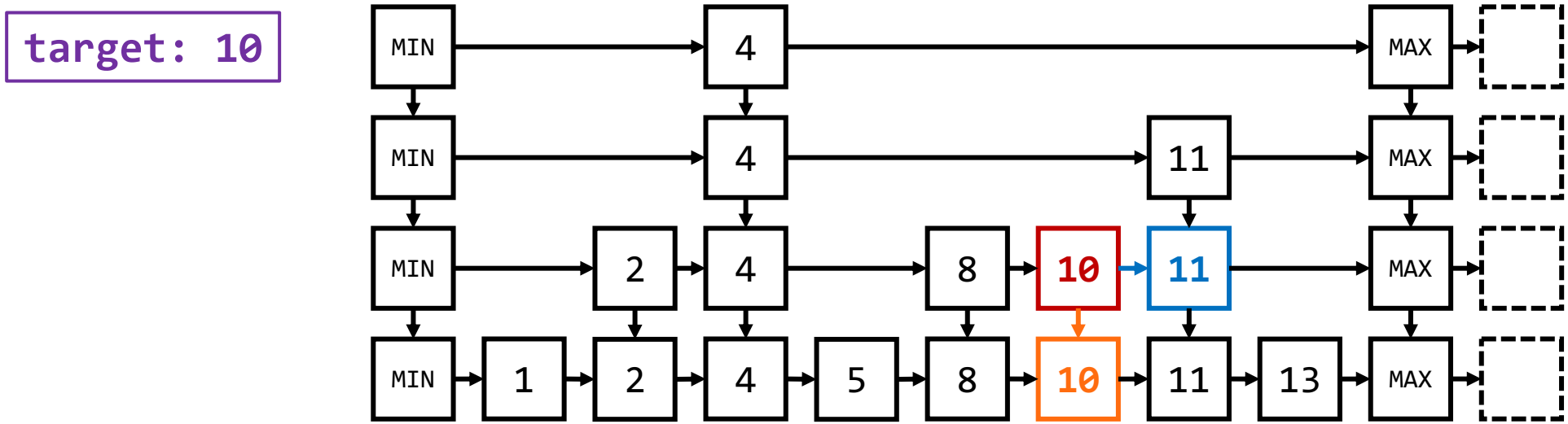
target: 10



Variants of Linked Lists - Skip List

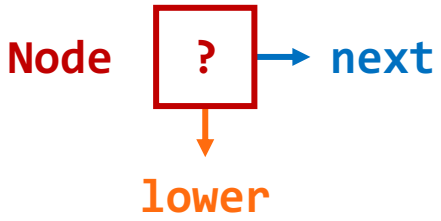
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node



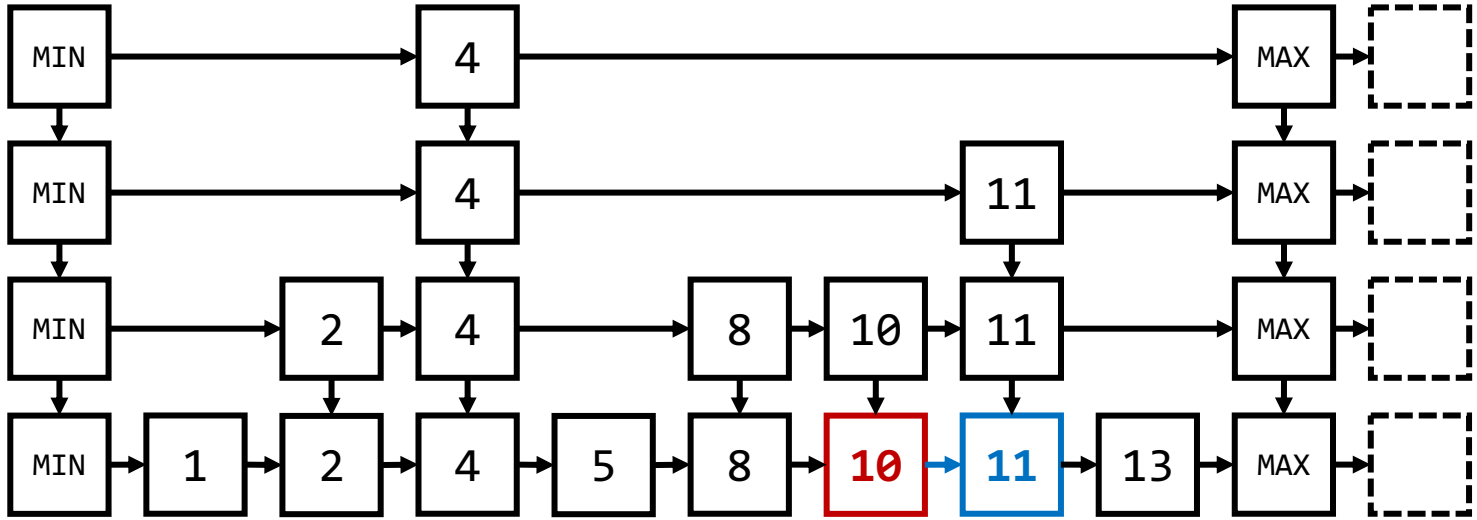


Variants of Linked Lists - Skip List

- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node

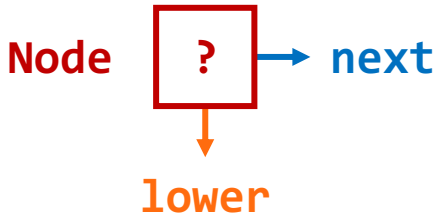


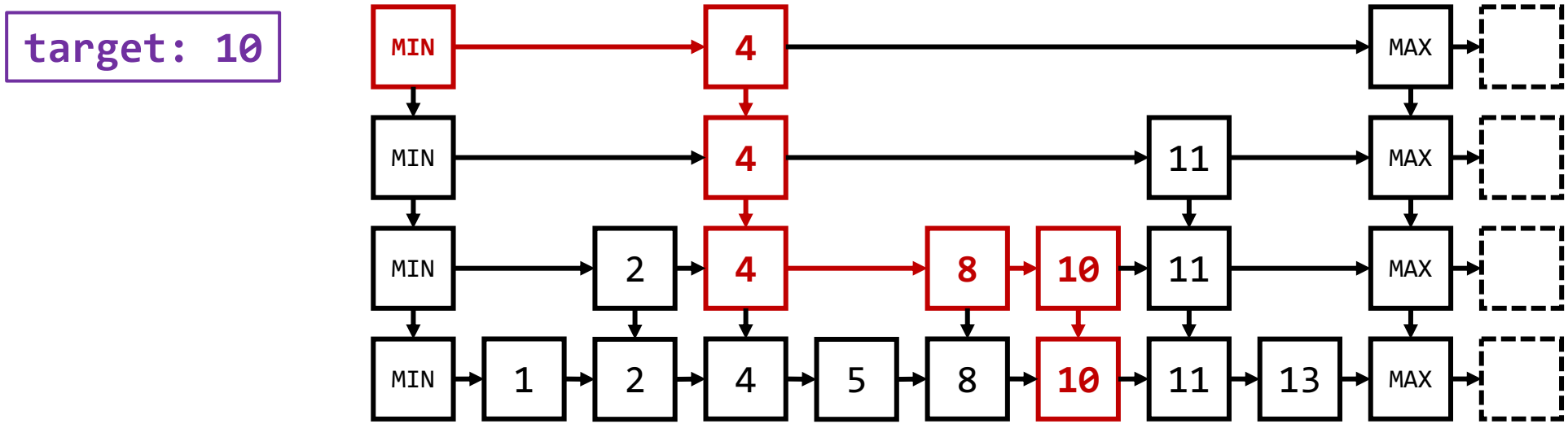
target: 10



Variants of Linked Lists - Skip List

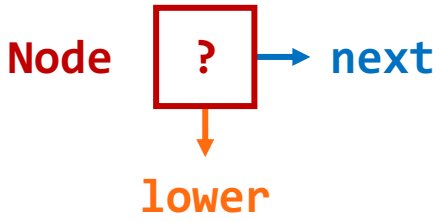
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current** < **next** <= **target**, then move to the **next** node
     2. If **current** <= **target** < **next**, then move to the **lower** node

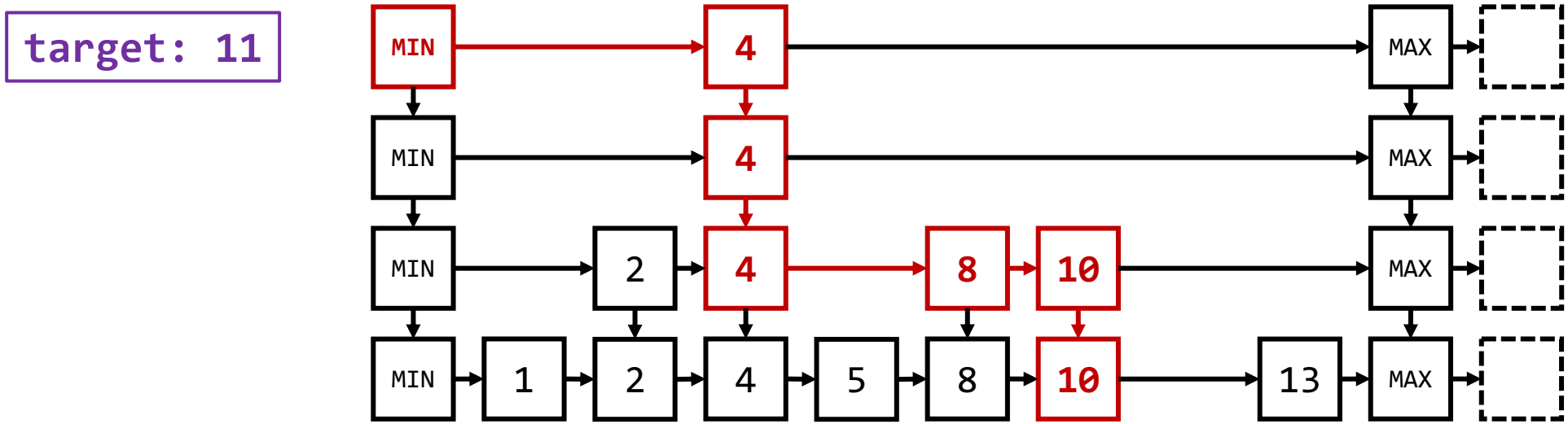




Variants of Linked Lists - Skip List

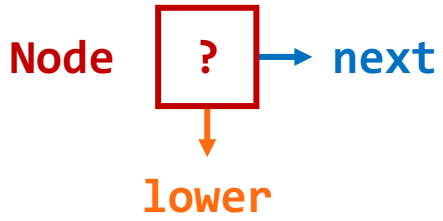
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**



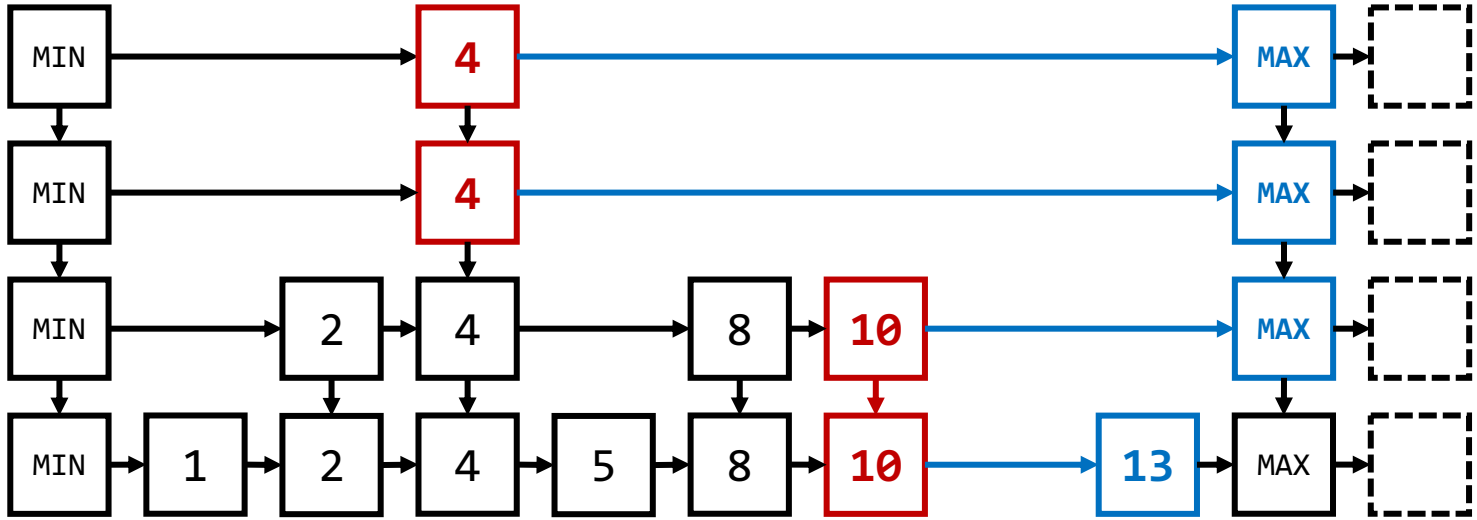


Variants of Linked Lists - Skip List

- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**

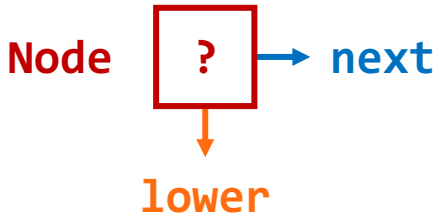


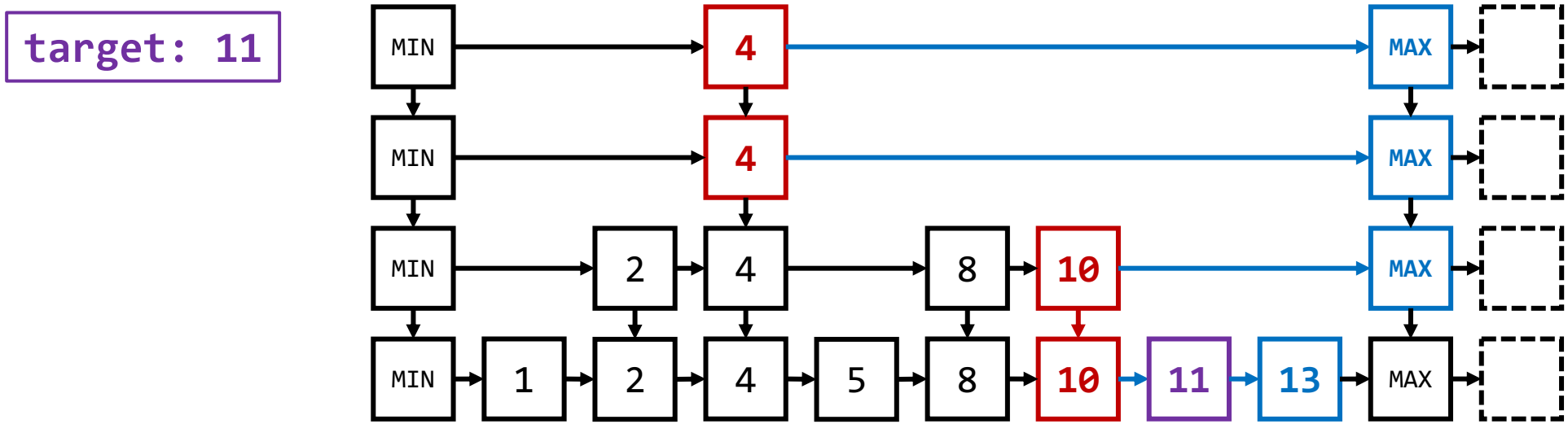
target: 11



Variants of Linked Lists - Skip List

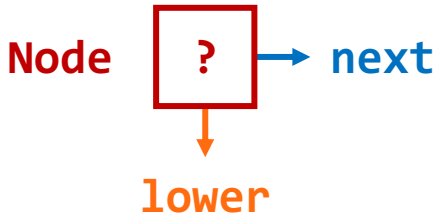
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top





Variants of Linked Lists - Skip List

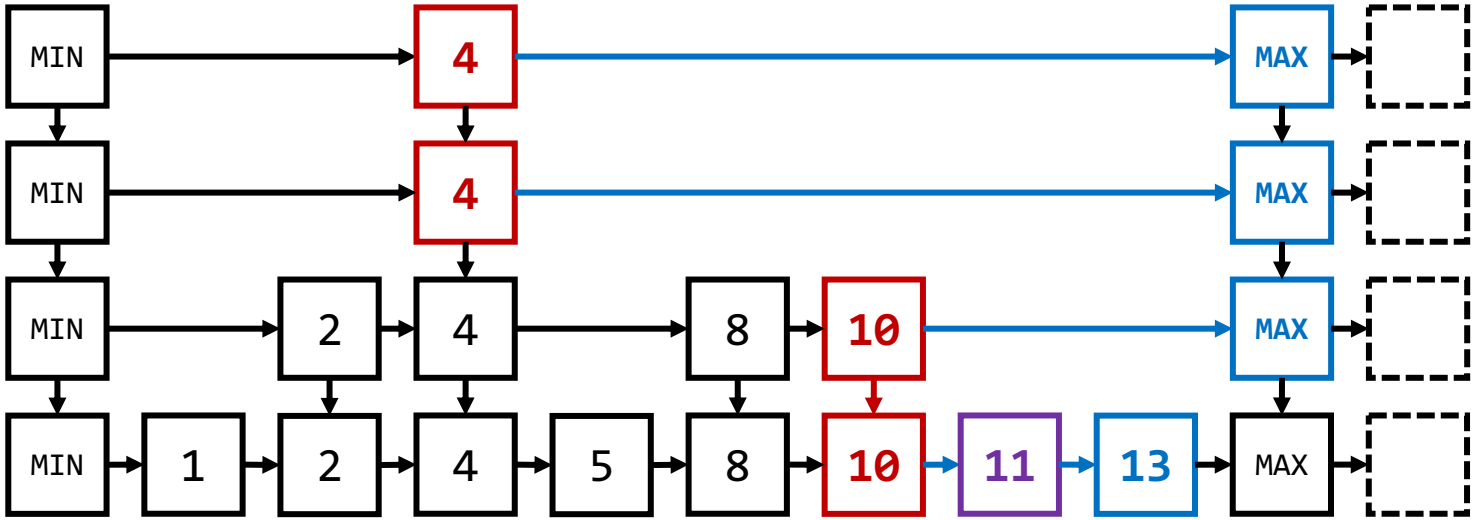
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top
     1. If the flipped coin shows head, continue to insert



Coin Flip

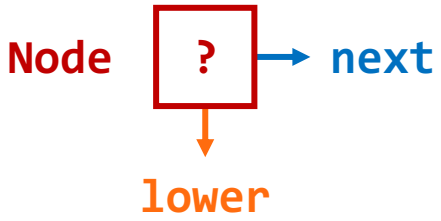
$p(\text{head}) = p(\text{tail}) = 0.5$

target: 11



Variants of Linked Lists - Skip List

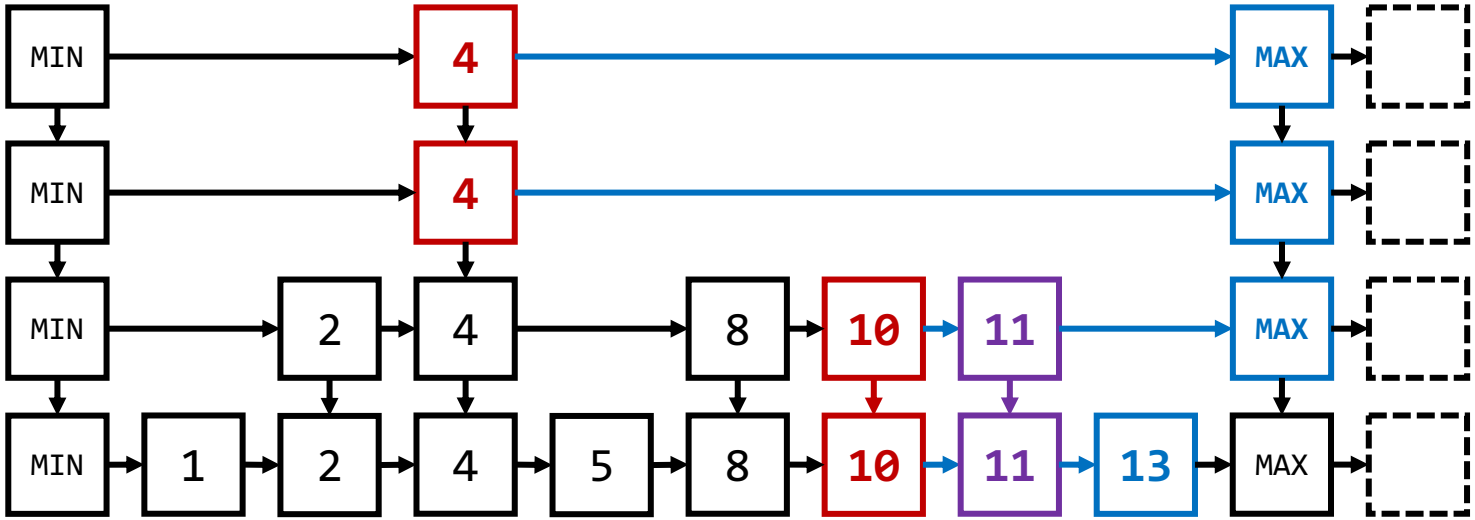
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top
     1. If the flipped coin shows head, continue to insert



Coin Flip

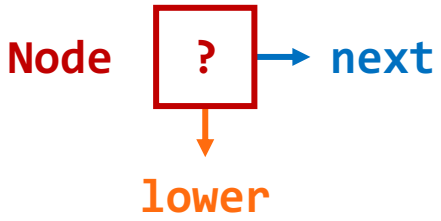
$p(\text{head}) = p(\text{tail}) = 0.5$

target: 11



Variants of Linked Lists - Skip List

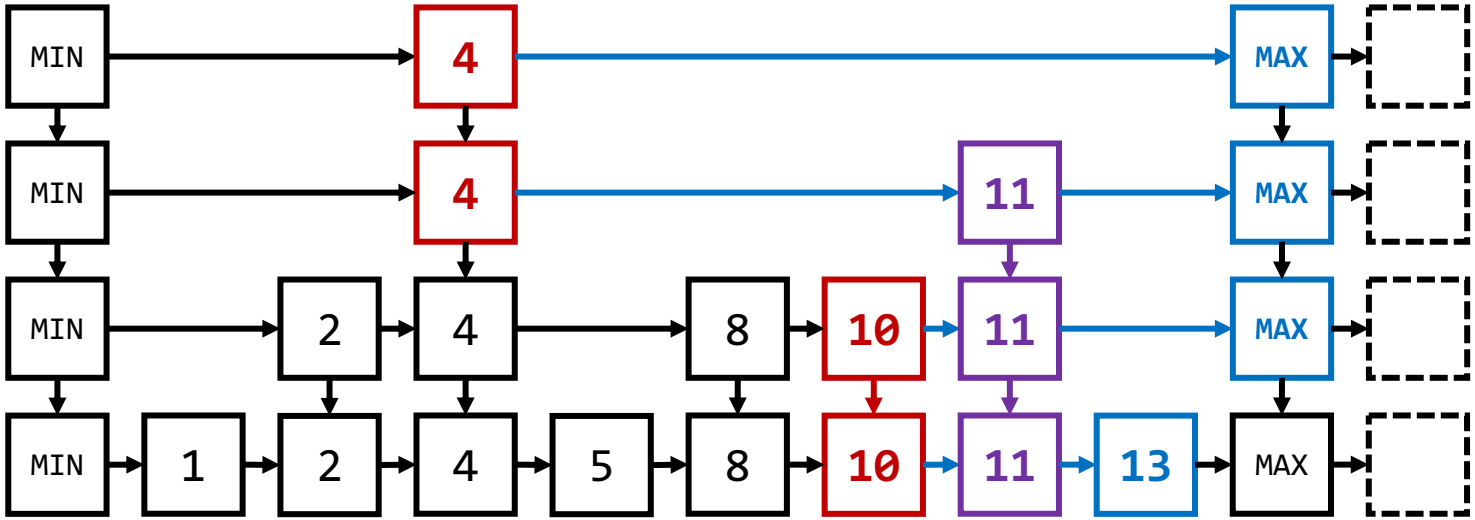
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top
     1. If the flipped coin shows head, continue to insert



Coin Flip

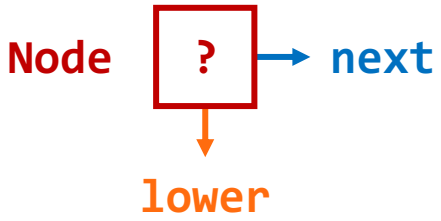
$p(\text{head}) = p(\text{tail}) = 0.5$

target: 11



Variants of Linked Lists - Skip List

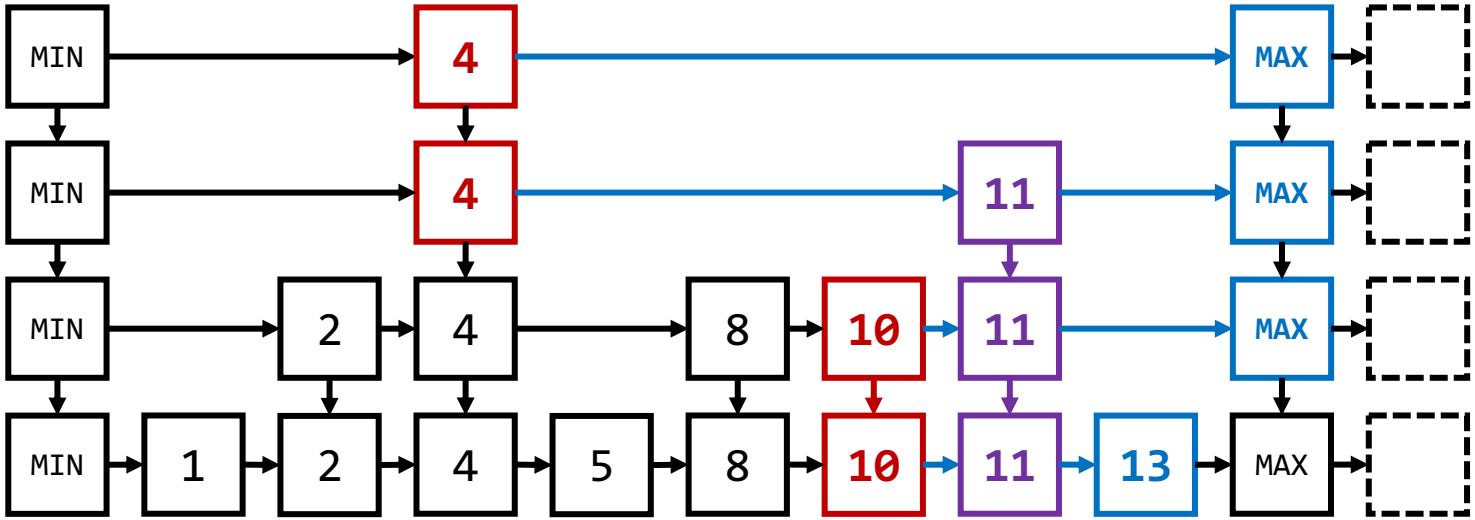
- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top
     1. If the flipped coin shows head, continue to insert
     2. If the flipped coin shows tail, stop insertion



Coin Flip

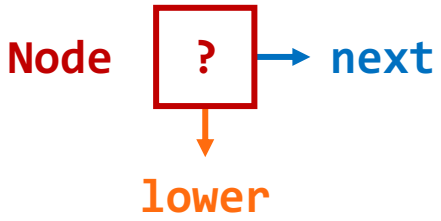
$p(\text{head}) = p(\text{tail}) = 0.5$

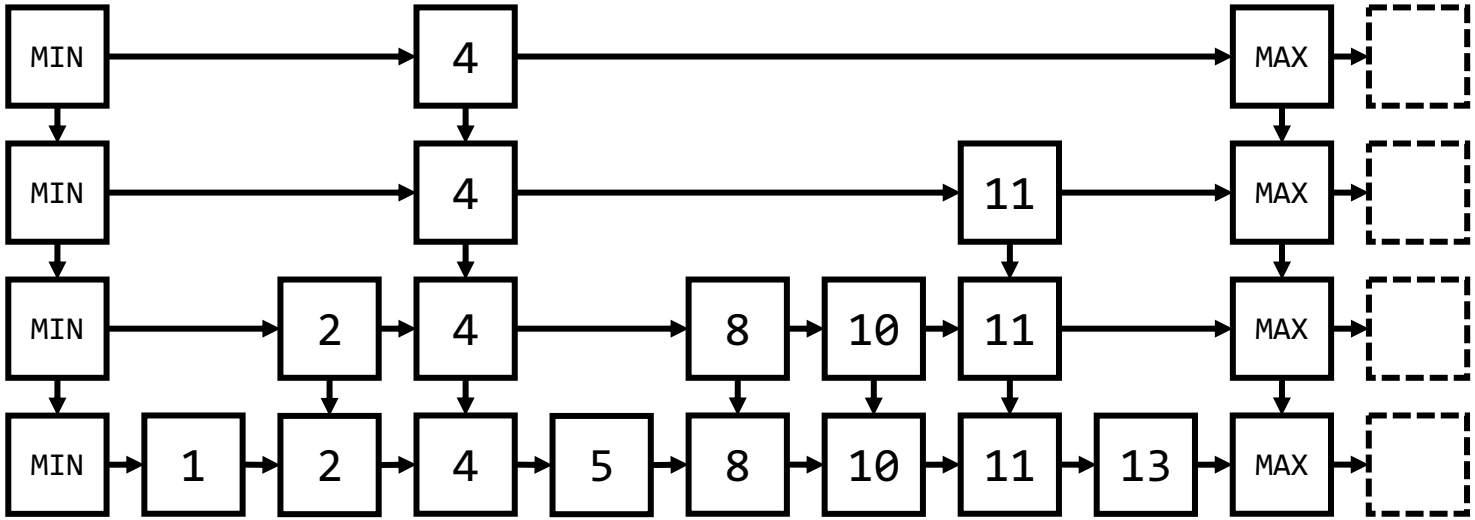
target: 11



Variants of Linked Lists - Skip List

- The **insertion** procedure in **Skip Lists**
  1. For all levels, find rightmost **nodes** < **target**
  2. Insert **nodes** from the bottom to the top
     1. If the flipped coin shows head, continue to insert
     2. If the flipped coin shows tail, stop insertion

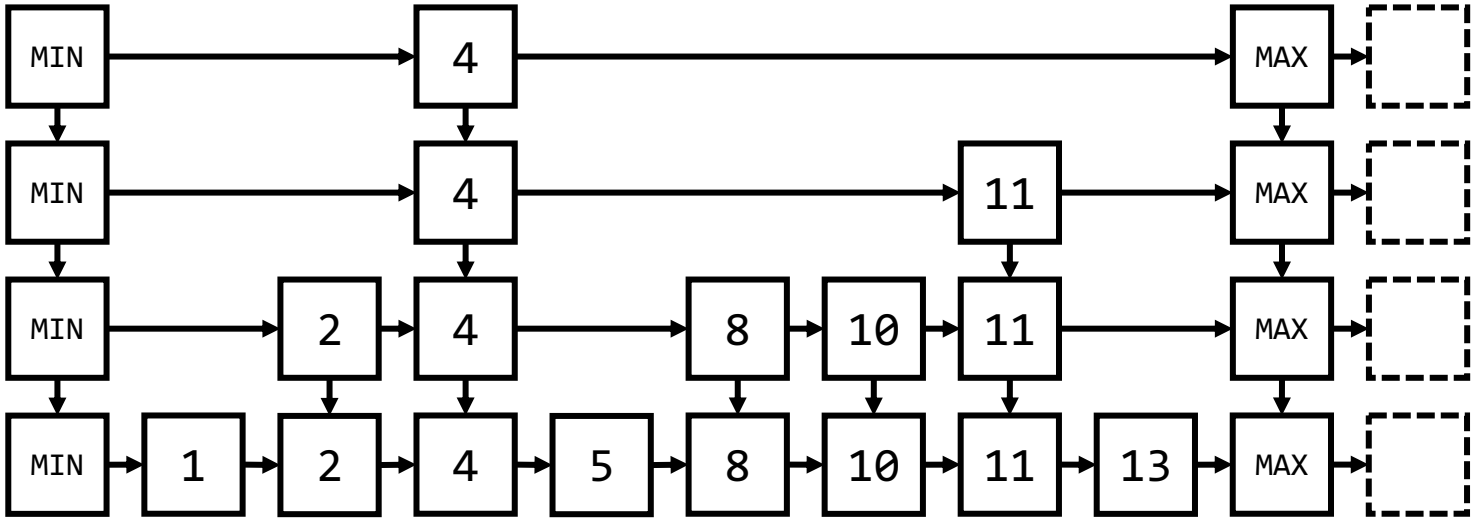




Variants of Linked Lists - Skip List

- Skip Lists - A **probabilistic** data structure for an ordered sequence
  - This allows $O(\log n)$ average time complexity for search
  - This allows $O(\log n)$ average time complexity for insertion/deletion while maintaining the order

  - # of nodes is roughly N + N/2 + N/4 + … = 2N
  - Therefore, **the space complexity** of the skip list is $O(n)$



Time & Space Complexities

Time Complexity

Operation	(standard) Array	(standard) Linked List	Skip List (average case)	Skip List (worst case)
Insertion	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
Deletion	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
Search by Index	$O(1)$	$O(n)$	$O(\log n)$	$O(n)$
Search by Value	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
Reversion	$O(n)$	$O(n)$	-	-

Space Complexity

(standard) Array	(standard) Linked List	Skip List (average case)	Skip List (worst case)
$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$

- **Note.** The array operations can be improved to $O(\log n)$ with advanced structures

Any Questions?