# Question

Design your implementation of the linked list. You can choose to use a singly or doubly linked list.  
A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node.  
If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are **0-indexed**.

Implement the MyLinkedList class:

* MyLinkedList() Initializes the MyLinkedList object.
* int get(int index) Get the value of the indexth node in the linked list. If the index is invalid, return -1.
* void addAtHead(int val) Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
* void addAtTail(int val) Append a node of value val as the last element of the linked list.
* void addAtIndex(int index, int val) Add a node of value val before the indexth node in the linked list. If index equals the length of the linked list, the node will be appended to the end of the linked list. If index is greater than the length, the node **will not be inserted**.
* void deleteAtIndex(int index) Delete the indexth node in the linked list, if the index is valid.

**Example 1:**

**Input**

["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]

[[], [1], [3], [1, 2], [1], [1], [1]]

**Output**

[null, null, null, null, 2, null, 3]

**Explanation**

MyLinkedList myLinkedList = new MyLinkedList();

myLinkedList.addAtHead(1);

myLinkedList.addAtTail(3);

myLinkedList.addAtIndex(1, 2); // linked list becomes 1->2->3

myLinkedList.get(1); // return 2

myLinkedList.deleteAtIndex(1); // now the linked list is 1->3

myLinkedList.get(1); // return 3

**Constraints:**

* 0 <= index, val <= 1000
* Please do not use the built-in LinkedList library.
* At most 2000 calls will be made to get, addAtHead, addAtTail,  addAtIndex and deleteAtIndex.

# Solution

#### **Interview Strategy**

[Linked List](https://en.wikipedia.org/wiki/Linked_list#Basic_concepts_and_nomenclature) is a data structure with zero or several elements. Each element contains a value and link(s) to the other element(s). Depending on the number of links, that could be singly linked list, doubly linked list and multiply linked list.

Singly linked list is the simplest one, it provides addAtHead in a constant time, and addAtTail in a linear time. Though doubly linked list is the most used one, because it provides both addAtHead and addAtTail in a constant time, and optimises the insert and delete operations.

Doubly linked list is implemented in Java as [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html). Since these structures are quite well-known, a good interview strategy would be to mention them during the discussion but not to base the code on them. Better to use the limited interview time to work with two ideas:

* [Sentinel nodes](https://leetcode.com/articles/remove-linked-list-elements/)

Sentinel nodes are widely used in the trees and linked lists as pseudo-heads, pseudo-tails, etc. They serve as the guardians, as the name suggests, and usually they do not hold any data.

Sentinels nodes will be used here to simplify insert and delete. We would apply this in both of the following approaches.

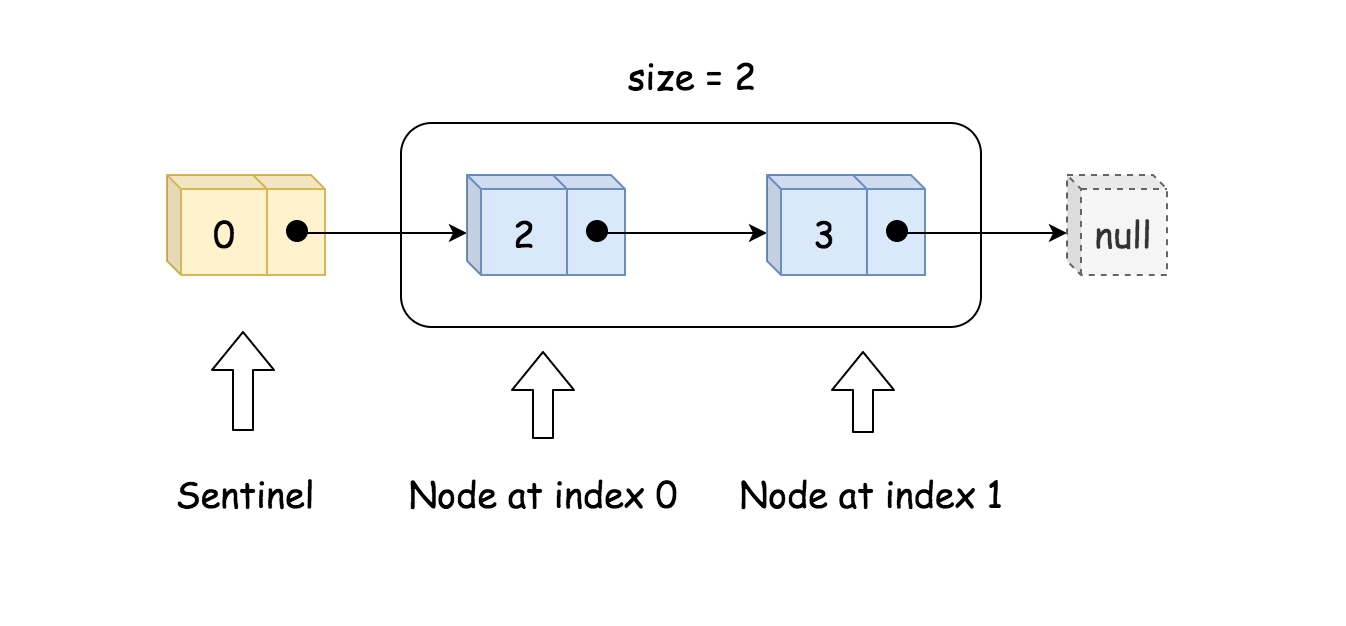
* Bidirectional search for doubly-linked list

Rather than starting from the head, we could search the node in a doubly-linked list from both head and tail.

If you are familiar with the concepts, you can start directly from the Approach #2. By the way, the Approach #2 is 90% of what you need to solve the problem of [LRU Cache](https://leetcode.com/articles/lru-cache/).

#### **Approach 1: Singly Linked List**

Let's start from the simplest possible MyLinkedList, which contains just a structure size and a sentinel head.



|  |
| --- |
| class MyLinkedList {  int size;  ListNode head; // sentinel node as pseudo-head  public MyLinkedList() {  size = 0;  head = new ListNode(0);  }  } |

Note, that sentinel node is used as a pseudo-head and is always present. This way the structure could never be empty, it will contain at least a sentinel head. All nodes in MyLinkedList have a type ListNode: value + link to the next element.

|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  } |

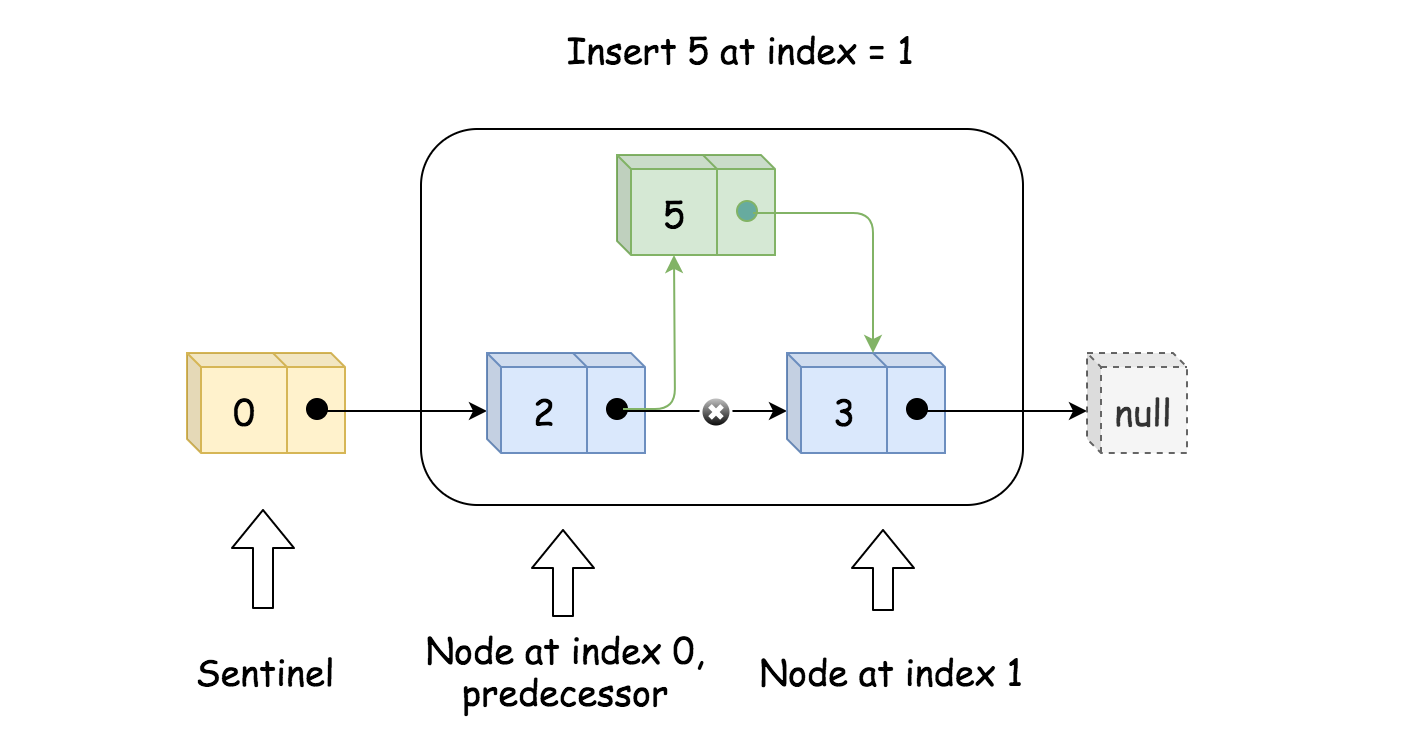
**Add at Index, Add at Head and Add at Tail**

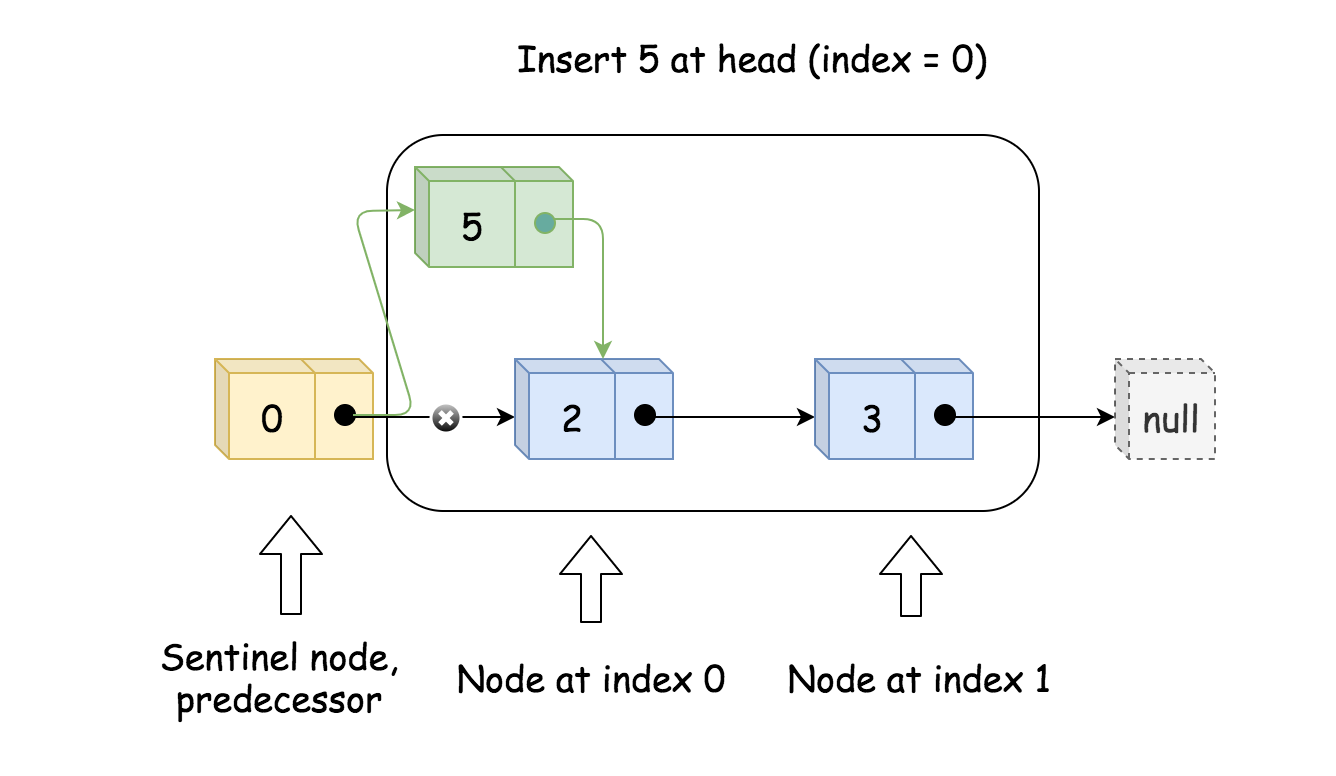
Let's first discuss insert at index operation, because thanks to the sentinel node addAtTail and addAtHead operations could be reduced to this operation as well.

The idea is straightforward:

* Find the predecessor of the node to insert. If the node is to be inserted at head, its predecessor is a sentinel head. If the node is to be inserted at tail, its predecessor is the last node.
* Insert the node by changing the link to the next node.

|  |
| --- |
| toAdd.next = pred.next;  pred.next = toAdd; |



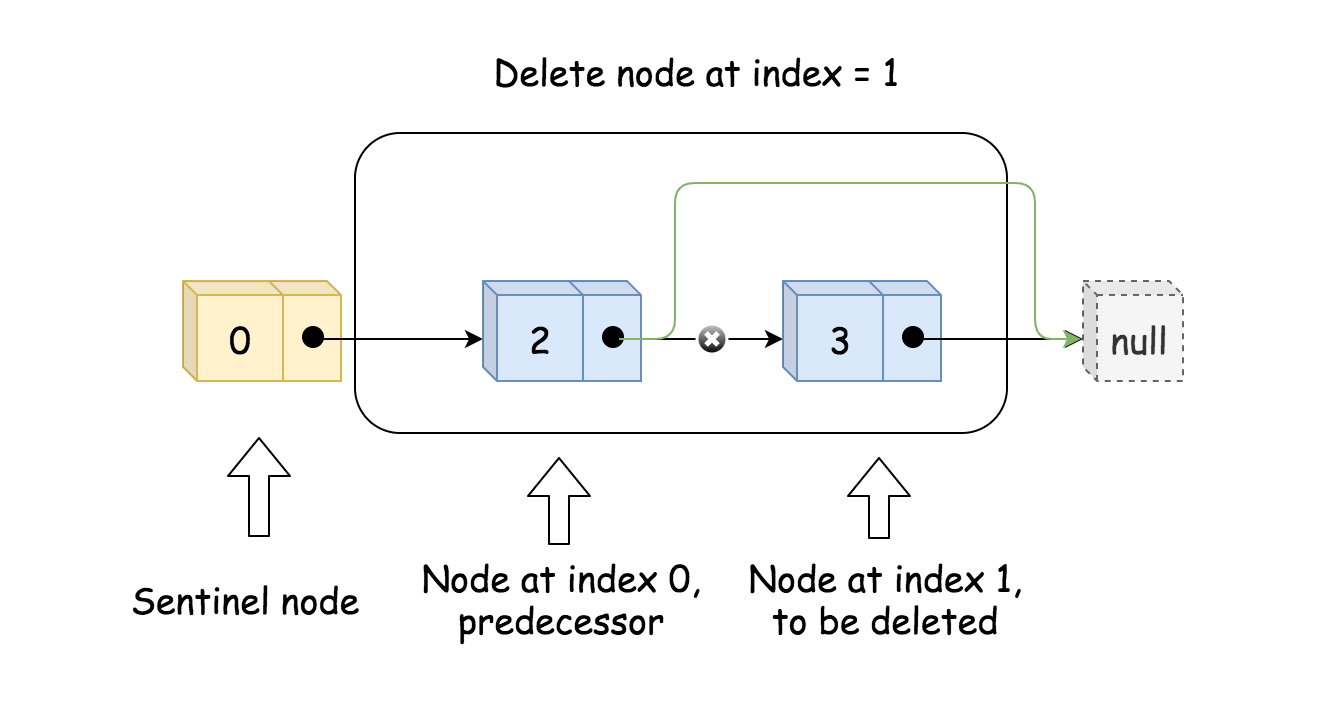


**Delete at Index**

Basically, the same as insert:

* Find the predecessor.
* Delete node by changing the links to the next node.

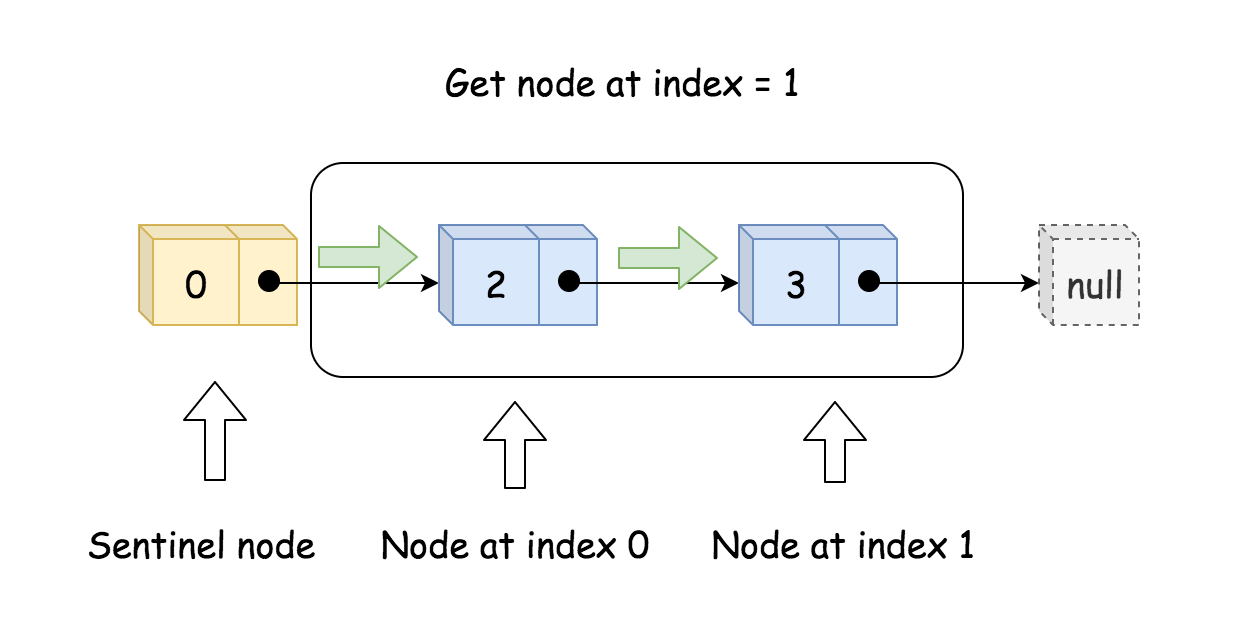
|  |
| --- |
| // delete pred.next  pred.next = pred.next.next; |



**Get**

Get is a very straightforward: start from the sentinel node and do index + 1 steps

|  |
| --- |
| // index steps needed  // to move from sentinel node to wanted index  for(int i = 0; i < index + 1; ++i) curr = curr.next;  return curr.val; |



**Implementation**

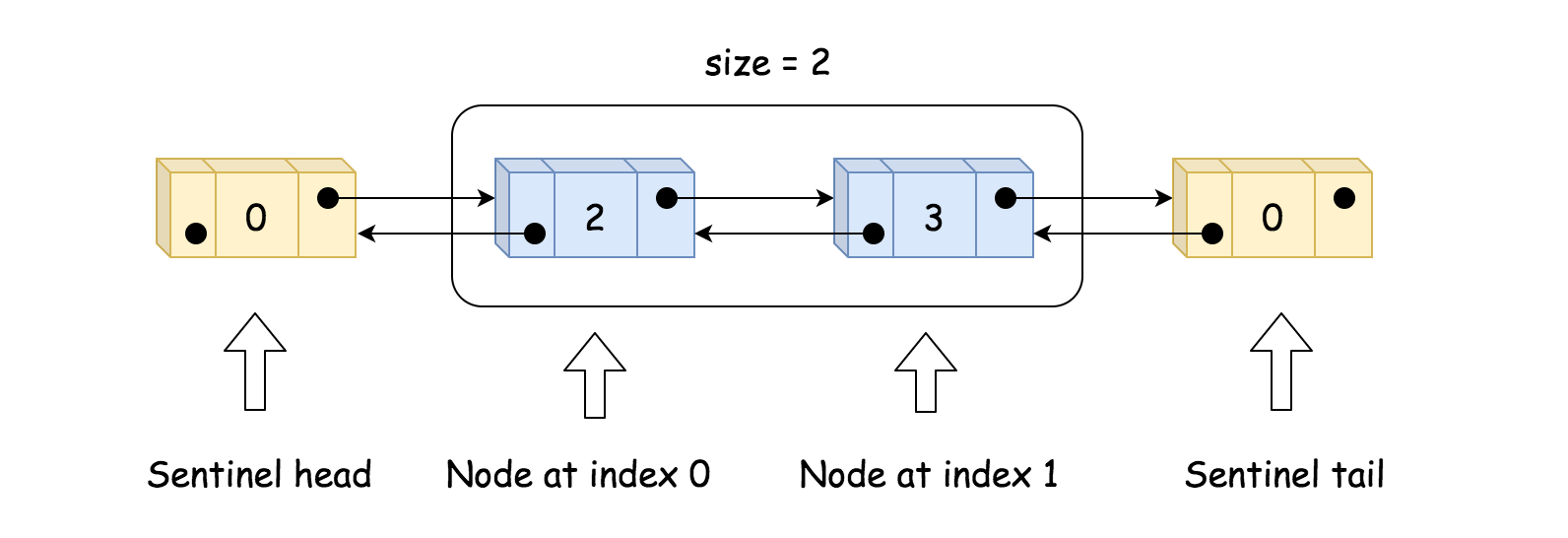
|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  }  class MyLinkedList {  int size;  ListNode head; // sentinel node as pseudo-head  public MyLinkedList() {  size = 0;  head = new ListNode(0);  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  // if index is invalid  if (index < 0 || index >= size) return -1;  ListNode curr = head;  // index steps needed  // to move from sentinel node to wanted index  for(int i = 0; i < index + 1; ++i) curr = curr.next;  return curr.val;  }  /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  addAtIndex(0, val);  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  addAtIndex(size, val);  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  // If index is greater than the length,  // the node will not be inserted.  if (index > size) return;  // [so weird] If index is negative,  // the node will be inserted at the head of the list.  if (index < 0) index = 0;  ++size;  // find predecessor of the node to be added  ListNode pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  // node to be added  ListNode toAdd = new ListNode(val);  // insertion itself  toAdd.next = pred.next;  pred.next = toAdd;  }  /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  // if the index is invalid, do nothing  if (index < 0 || index >= size) return;  size--;  // find predecessor of the node to be deleted  ListNode pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  // delete pred.next  pred.next = pred.next.next;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(1)O(1) for addAtHead. \mathcal{O}(k)O(*k*) for get, addAtIndex, and deleteAtIndex, where k*k* is an index of the element to get, add or delete. \mathcal{O}(N)O(*N*) for addAtTail.
* Space complexity: \mathcal{O}(1)O(1) for all operations.

#### **Approach 2: Doubly Linked List**

Time to implement DLL MyLinkedList, which is a much faster (twice faster on the testcase set here) though a bit more complex. It contains size, sentinel head and sentinel tail.



|  |
| --- |
| class MyLinkedList {  int size;  // sentinel nodes as pseudo-head and pseudo-tail  ListNode head, tail;  public MyLinkedList() {  size = 0;  head = new ListNode(0);  tail = new ListNode(0);  head.next = tail;  tail.prev = head;  }  } |

Note, that sentinel head and tail are always present. All nodes in MyLinkedList have a type ListNode: value + two links: to the next and to the previous elements.

|  |
| --- |
| class MyLinkedList {  int size;  // sentinel nodes as pseudo-head and pseudo-tail  ListNode head, tail;  public MyLinkedList() {  size = 0;  head = new ListNode(0);  tail = new ListNode(0);  head.next = tail;  tail.prev = head;  }  } |

**Add at Index, Add at Head and Add at Tail**

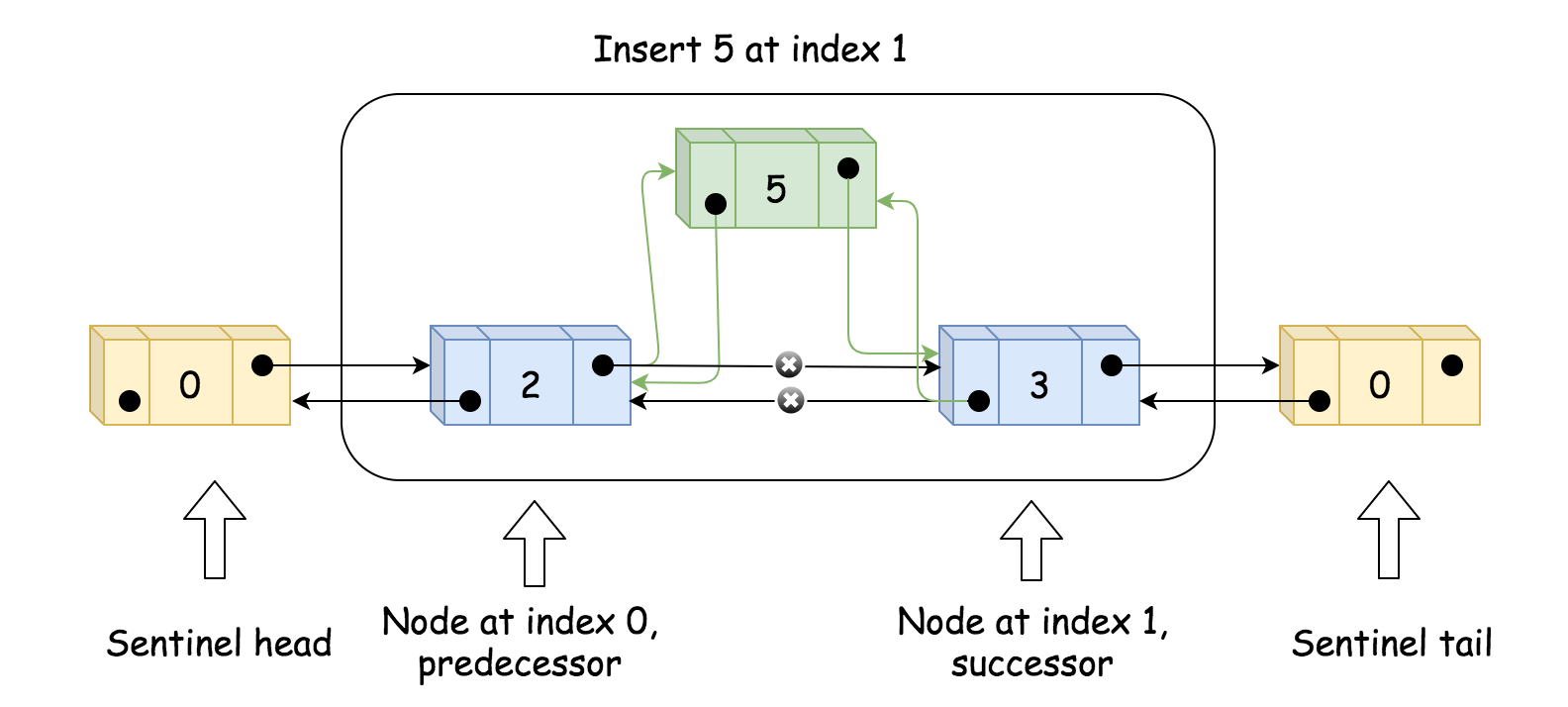
The idea is simple:

* Find the predecessor and the successor of the node to insert. If the node is to be inserted at head, its predecessor is a sentinel head. If the node is to be inserted at tail, its successor is a sentinel tail.

Use bidirectional search to perform faster.

* Insert the node by changing the links to the next and previous nodes.

|  |
| --- |
| toAdd.prev = pred  toAdd.next = succ  pred.next = toAdd  succ.prev = toAdd |

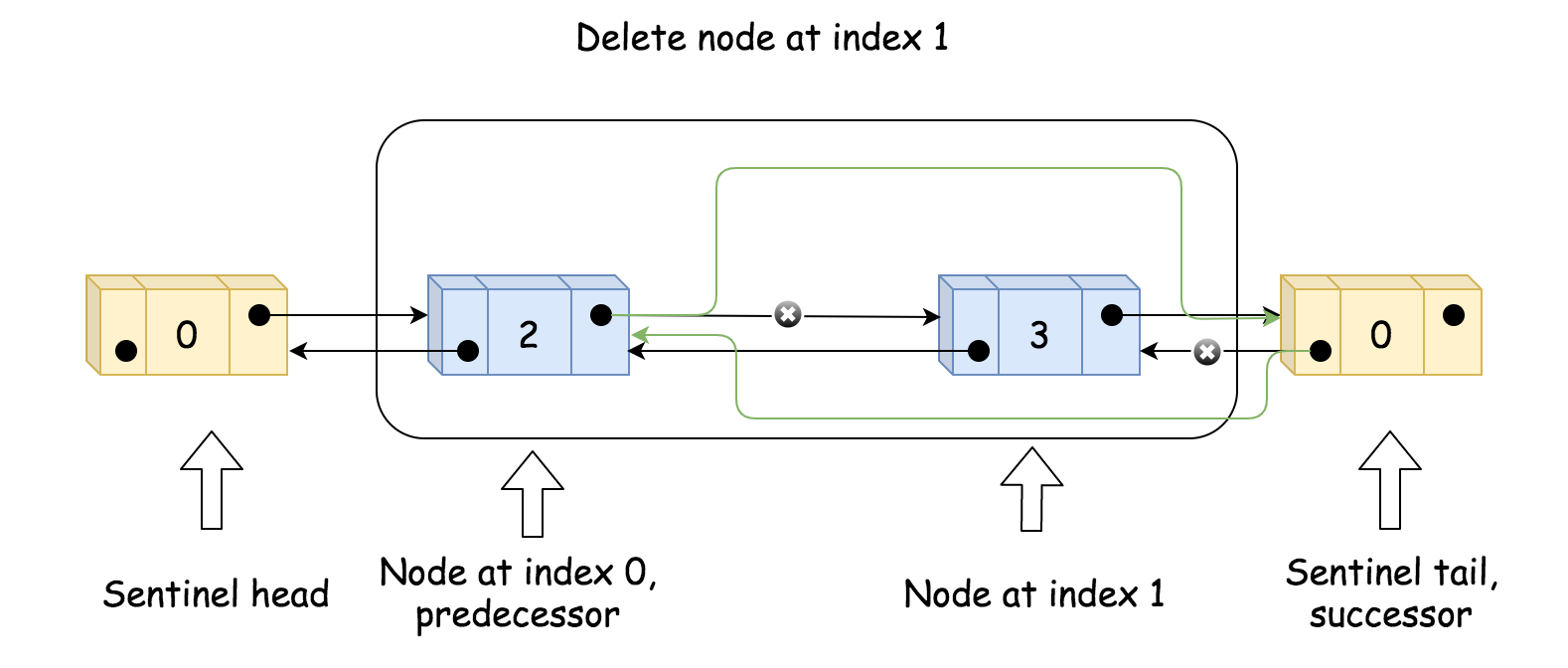


**Delete at Index**

Basically, the same as insert:

* Find the predecessor and successor.
* Delete node by changing the links to the next and previous nodes.

|  |
| --- |
| pred.next = succ  succ.prev = pred |

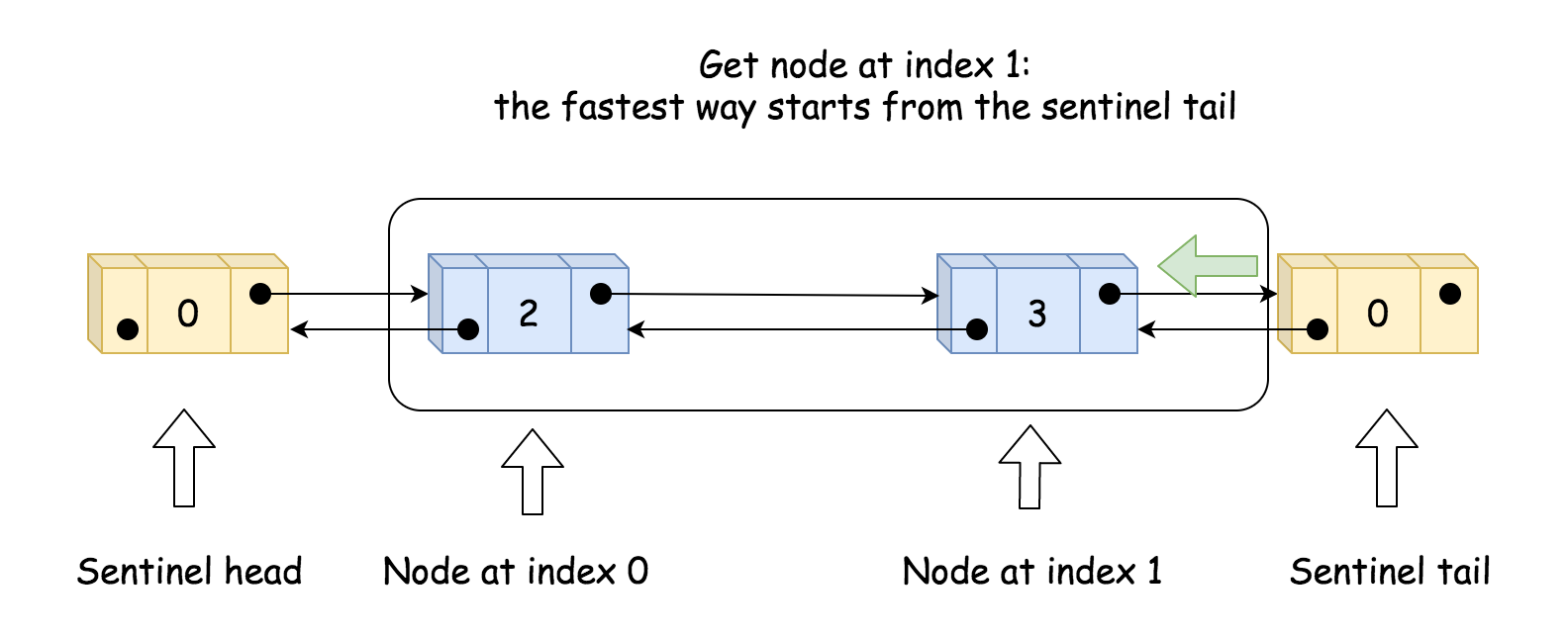


**Get**

Get is very straightforward:

* Compare index and size - index to define the fastest way: starting from the head, or starting from the tail.
* Go to the wanted node.

|  |
| --- |
| // choose the fastest way: to move from the head  // or to move from the tail  ListNode curr = head;  if (index + 1 < size - index)  for(int i = 0; i < index + 1; ++i) curr = curr.next;  else {  curr = tail;  for(int i = 0; i < size - index; ++i) curr = curr.prev;  } |



**Implementation**

|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode prev;  ListNode(int x) { val = x; }  }  class MyLinkedList {  int size;  // sentinel nodes as pseudo-head and pseudo-tail  ListNode head, tail;  public MyLinkedList() {  size = 0;  head = new ListNode(0);  tail = new ListNode(0);  head.next = tail;  tail.prev = head;  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  // if index is invalid  if (index < 0 || index >= size) return -1;  // choose the fastest way: to move from the head  // or to move from the tail  ListNode curr = head;  if (index + 1 < size - index)  for(int i = 0; i < index + 1; ++i) curr = curr.next;  else {  curr = tail;  for(int i = 0; i < size - index; ++i) curr = curr.prev;  }  return curr.val;  }  /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  ListNode pred = head, succ = head.next;  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  ListNode succ = tail, pred = tail.prev;  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  // If index is greater than the length,  // the node will not be inserted.  if (index > size) return;  // [so weird] If index is negative,  // the node will be inserted at the head of the list.  if (index < 0) index = 0;  // find predecessor and successor of the node to be added  ListNode pred, succ;  if (index < size - index) {  pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  succ = pred.next;  }  else {  succ = tail;  for (int i = 0; i < size - index; ++i) succ = succ.prev;  pred = succ.prev;  }  // insertion itself  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  // if the index is invalid, do nothing  if (index < 0 || index >= size) return;  // find predecessor and successor of the node to be deleted  ListNode pred, succ;  if (index < size - index) {  pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  succ = pred.next.next;  }  else {  succ = tail;  for (int i = 0; i < size - index - 1; ++i) succ = succ.prev;  pred = succ.prev.prev;  }  // delete pred.next  --size;  pred.next = succ;  succ.prev = pred;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(1)O(1) for addAtHead and addAtTail. \mathcal{O}(\min(k, N - k))O(min(*k*,*N*−*k*)) for get, addAtIndex, and deleteAtIndex, where k*k* is an index of the element to get, add or delete.
* Space complexity: \mathcal{O}(1)O(1) for all operations.