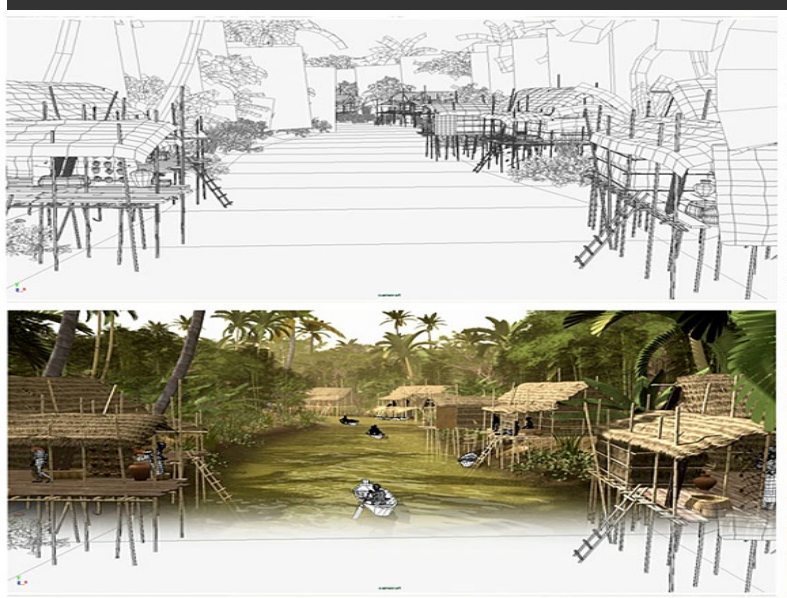
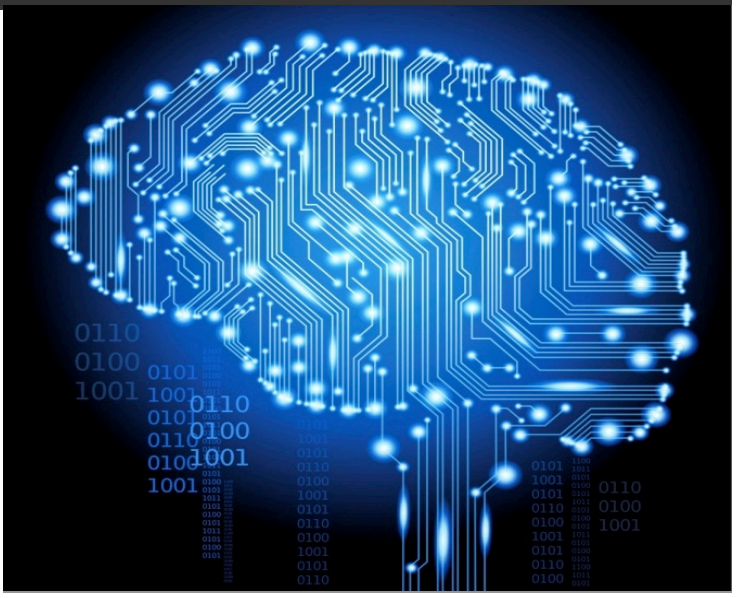
MONASH University

Information Technology

List ADT implemented with linked nodes

Prepared by:
Maria Garcia de la Banda





Objectives for this lesson

- **To understand the concept of Data Structures based on linked nodes**
- **To understand their use in implementing the List ADT**
- **To be able to:**
  - Implement, use and modify linked lists
  - Decide when is it appropriate to use them (as opposed to using the ones implemented with arrays)

MONASH University

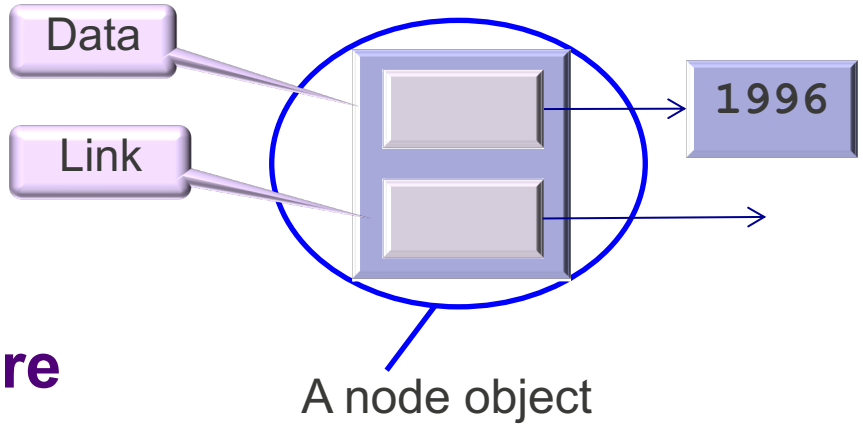
Linked Nodes

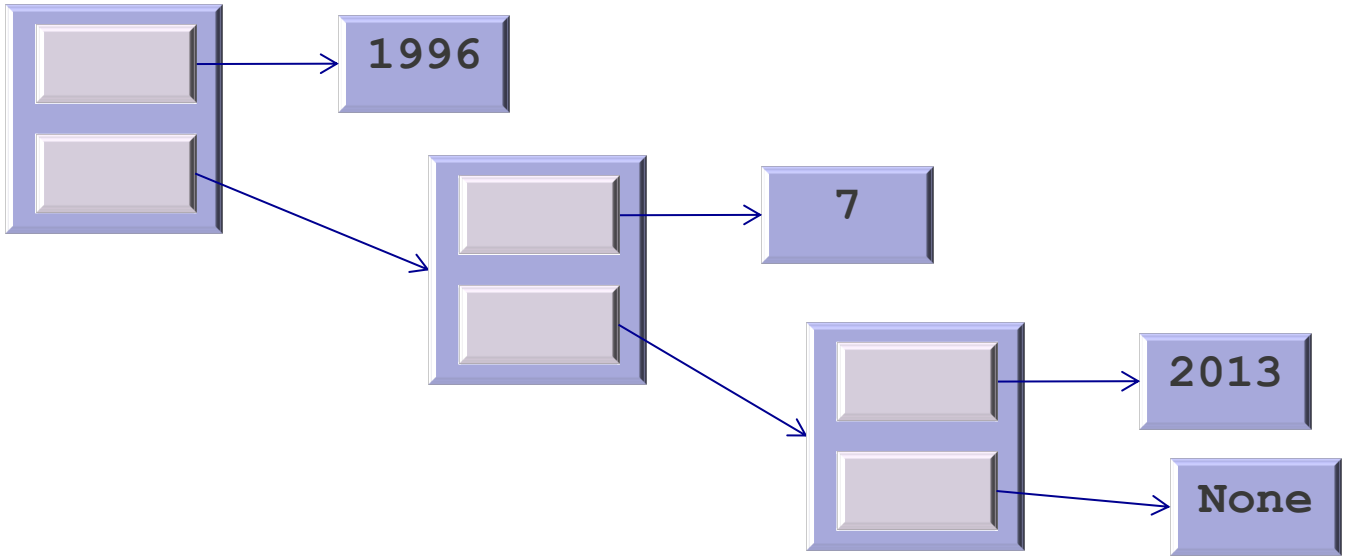
Motivation: recap of ADTs implemented with arrays

- **Recall array characteristics:**
  - Fixed size
  - Data items are stored sequentially
  - Each item occupies the same amount of space (in our case: stores a pointer)
- **Main advantages:**
  - Very fast access O(1)
  - Very compact representation if the array is relatively full
- **Main disadvantages:**
  - Expensive to resize: create a new array and copy all items
  - Slow operations if shuffling elements is required (insert/remove/delete_at_index)
- **Solution: linked data structures**

Linked node Data Structures

- **Collection of nodes**
- **Each node contains:**
  - One or more data items
  - One or more links to other nodes
- **Example: (simple) linked data structure**





As with arrays, linked node data structures can be used to implement stacks, queues and lists

Node class

- **How do we create a node?**
- **Easy! With a class that contains two instance variables**
  - The data
  - The link to the next node

Python's syntax for optional argument with default values. If the argument is not given, its value is assumed to be None.

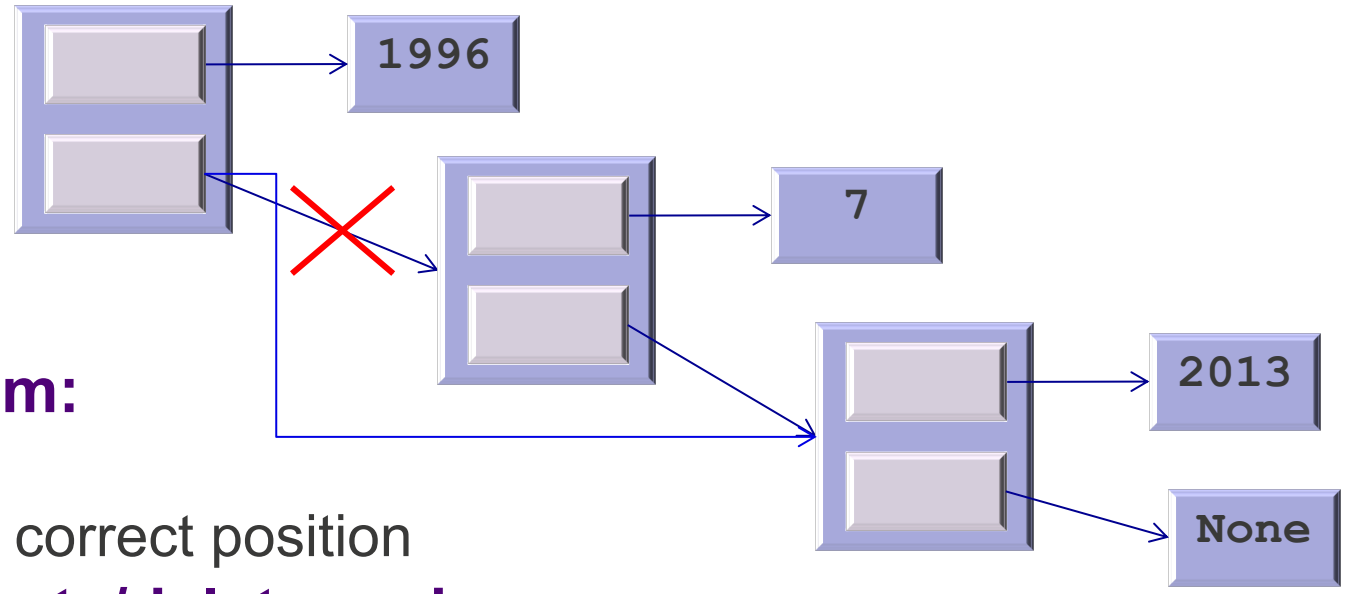
```
class Node(Generic[T]):
    def __init__(self, item: T = None) -> None:
        self.item = item
        self.link = None
```

- **If called as `Node(3)`, it is the usual**
- **If called as `Node()`, `item` defaults to `None`**
- **This avoids having to call with arguments set to `None`**
  - Users should not know about the class internals (default value of missing args)

Should have used it for the array capacity in ArrayList!

Linked node Data Structures: Advantages

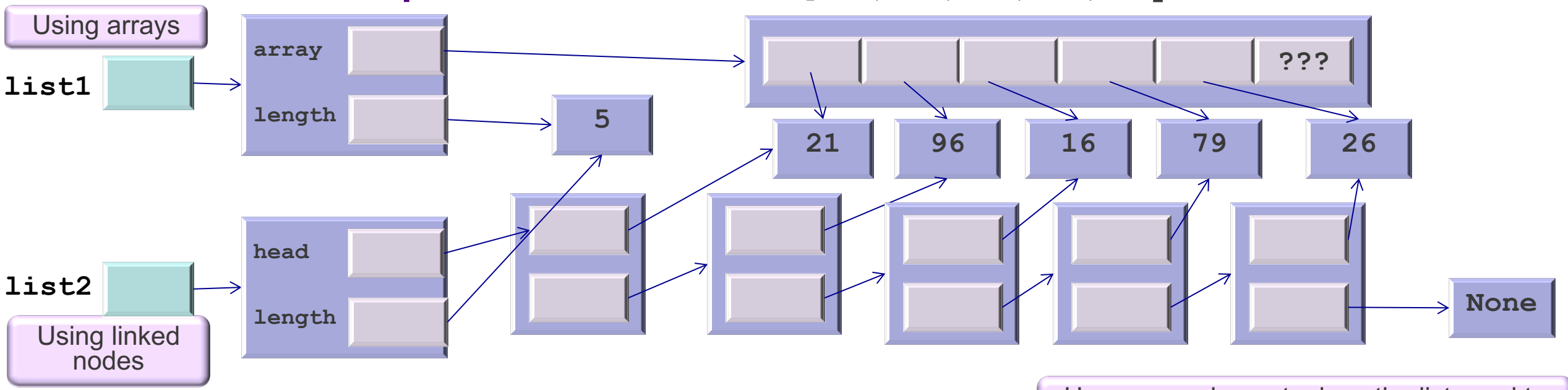
- **Fast deletion of an item (no need for reshuffling)**



- **Similar for adding an item:**
  - Create a node
  - Insert it (link it) into the correct position
- **Easily resizable: just create/delete node**
  - Full only if no more memory left to create the node
- **Less memory used than an array if the array is relatively empty**

Linked node Data Structures: Disadvantages

- **More memory used than an array if the array is relatively full**
  - Every data item has an associated link
- **Consider two implementations of list** `[21,96,16,79,26]`



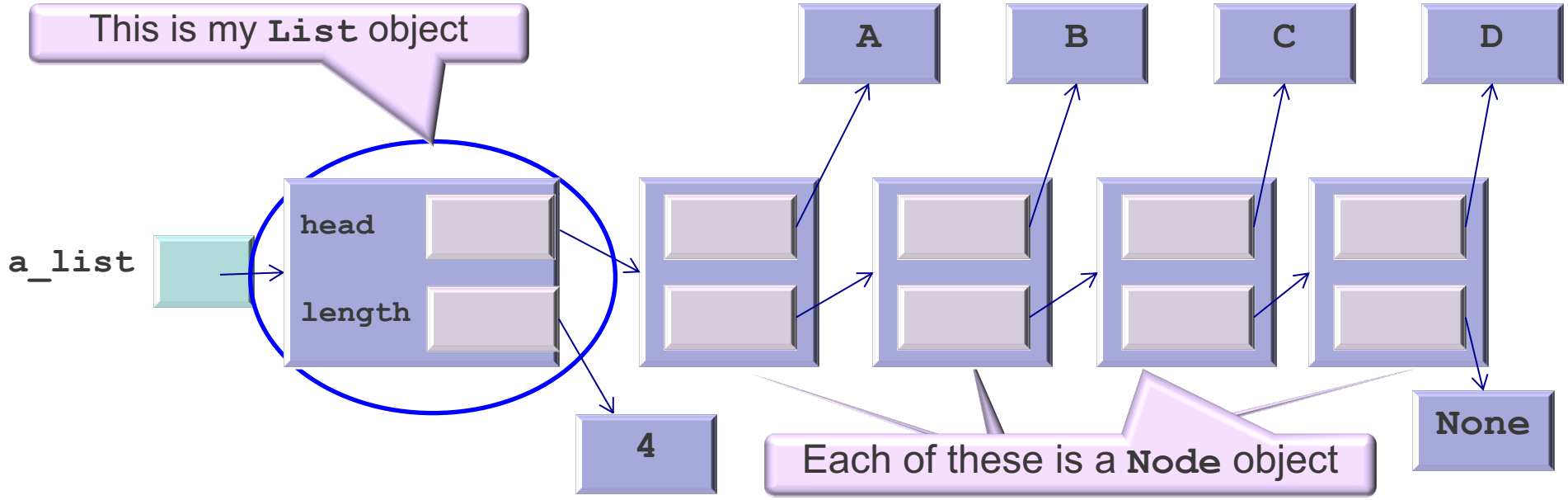
- **For some data types, the operations are more time consuming (no random access)**

How many elements does the list need to have for the array to use less space?

About half the size of the array

Linked Lists

- **What instance variables do we need for lists?**
  - Really only need one component: a reference to the head node
  - But since our base class also created the length, we will use it too (is useful!)
- **From the head we can access every other node**



MONASH
University

List ADT implemented with linked nodes part I

Remember: Abstract base List class

```
from abc import ABC, abstractmethod
from typing import TypeVar, Generic
T = TypeVar('T')

class List(ABC, Generic[T]):
    def __init__(self) -> None:
        self.length = 0

    @abstractmethod
    def __setitem__(self, index: int, item: T) -> None:
        pass

    @abstractmethod
    def __getitem__(self, index: int) -> T:
        pass

    def append(self, item: T) -> None:
        self.insert(len(self), item)

    @abstractmethod
    def insert(self, index: int, item: T) -> None:
        pass

    @abstractmethod
    def delete_at_index(self, index: int) -> T:
        pass

    @abstractmethod
    def index(self, item: T) -> int:
        pass
```

```
def remove(self, item: T) -> None:
    index = self.index(item)
    self.delete_at_index(index)

def __len__(self) -> int:
    return self.length

def is_empty(self) -> bool:
    return len(self) == 0

def clear(self):
    self.length = 0
```

Note: I have redefined append in terms of insert (reuse is your friend!)

Let's start implementing LinkList

- **Create the class and its __init__ constructor:**

```
from abstract_list import List, T

class LinkList(List[T]):
    def __init__(self) -> None:
        List.__init__(self)
        self.head = None
```

Empty list

- **Time complexity?**
  - Everything else is constant, including the parent __init__, so O(1)
- **Any properties of the list elements that affect big O?** No! so best = worst
- **Invariant: the number of valid elements is `self.length`, and are found via the node links starting from the head**

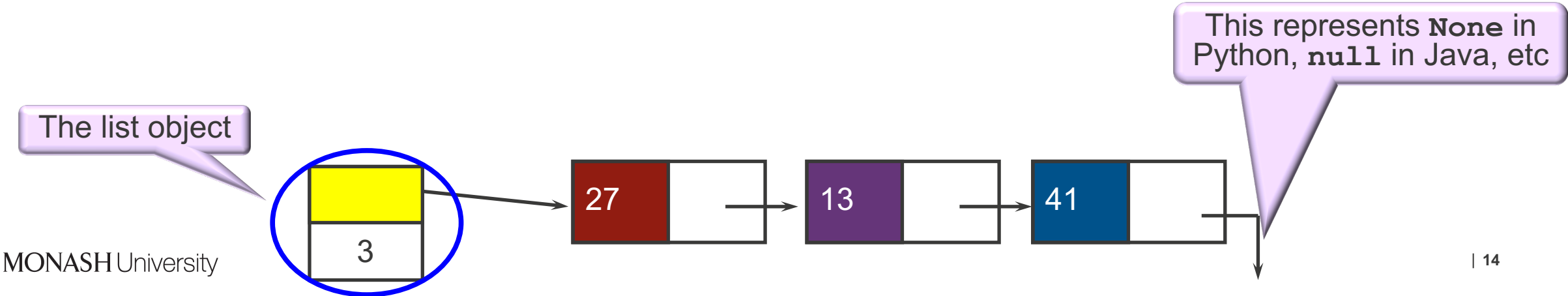
What about the other methods?

- **We do not have direct access to the elements**
- **Thus, we will need a lot of traversing**
  - All abstract methods need to traverse the list
- **Some need to traverse up to an index:**
  - `__setitem__`, `__getitem__`, `delete_at_index` and `insert`
- **Others need to traverse up to an item**
  - `index`
- **We will define an internal method to return the node at a given index**
  - `def __get_node_at_index(self, index: int) -> Node[T]:`
  - If the index is out of bounds, we will raise a `ValueError`

Let's see how to search for the node at index 2

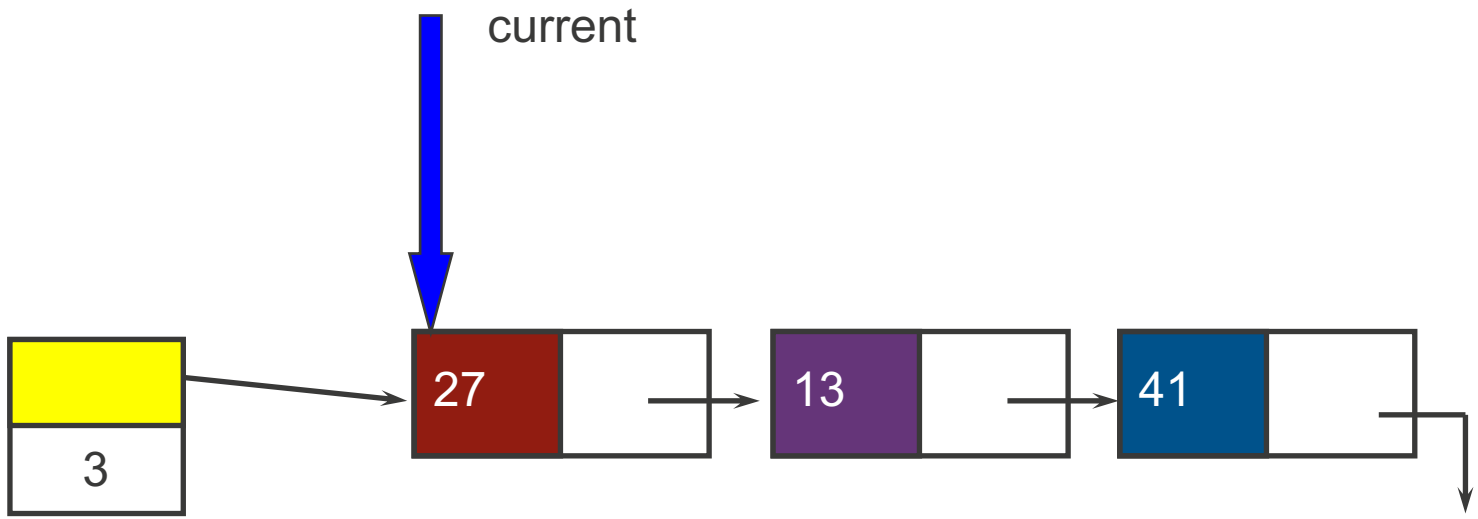
- **If the index is within the bounds (0 <= and < len(self))**

I will use this more abstract representation to explain the algorithm at a high-level, and the more detailed representation to follow the Python code



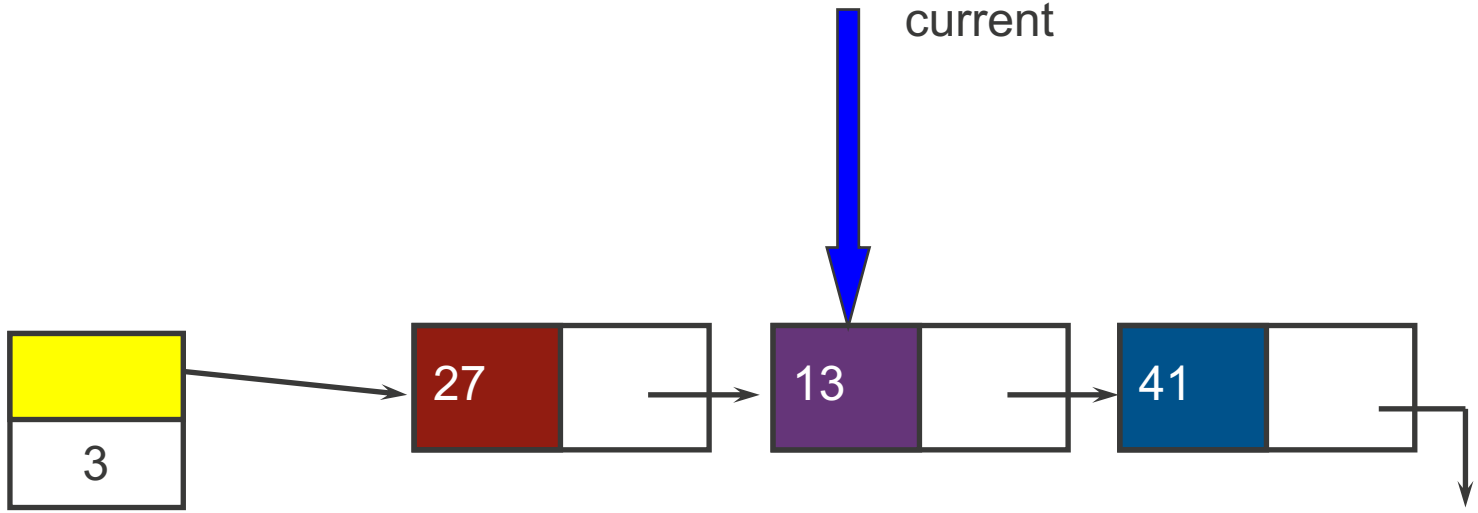
Let's see how to search for the node at index 2

- If the index is within the bounds (0 <= and < len(self))
- Start with the node at the head



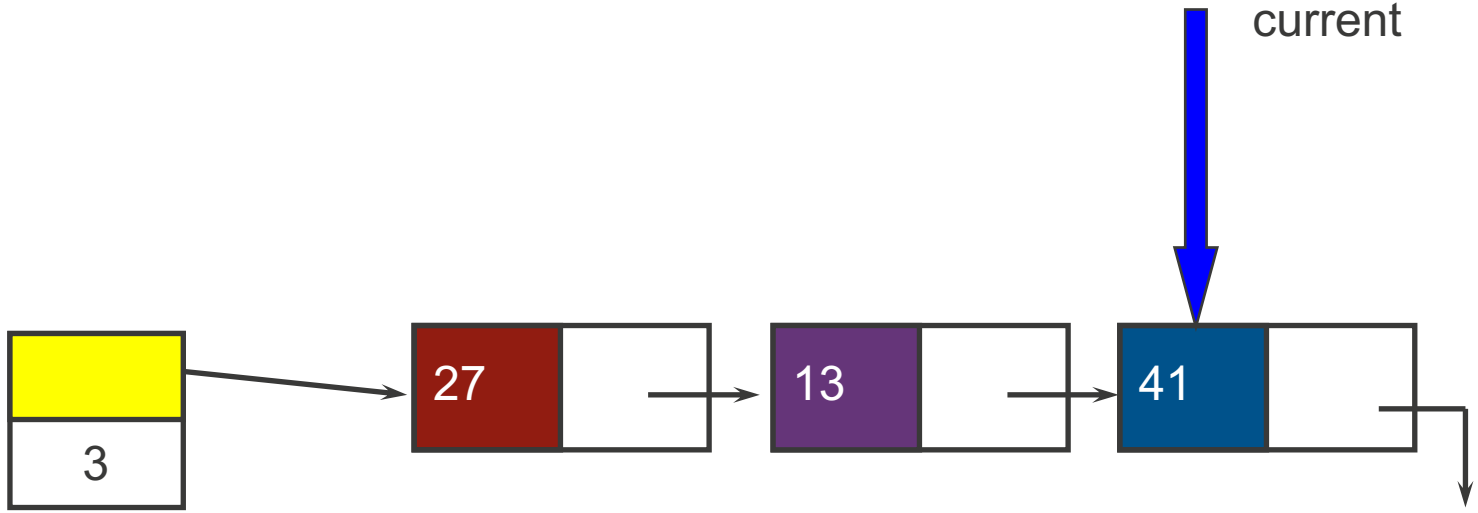
Let's see how to search for the node at index 2

- If the index is within the bounds (0 <= and < len(self))
- Start with the node at the head
- If not the index, follow the link to the next node



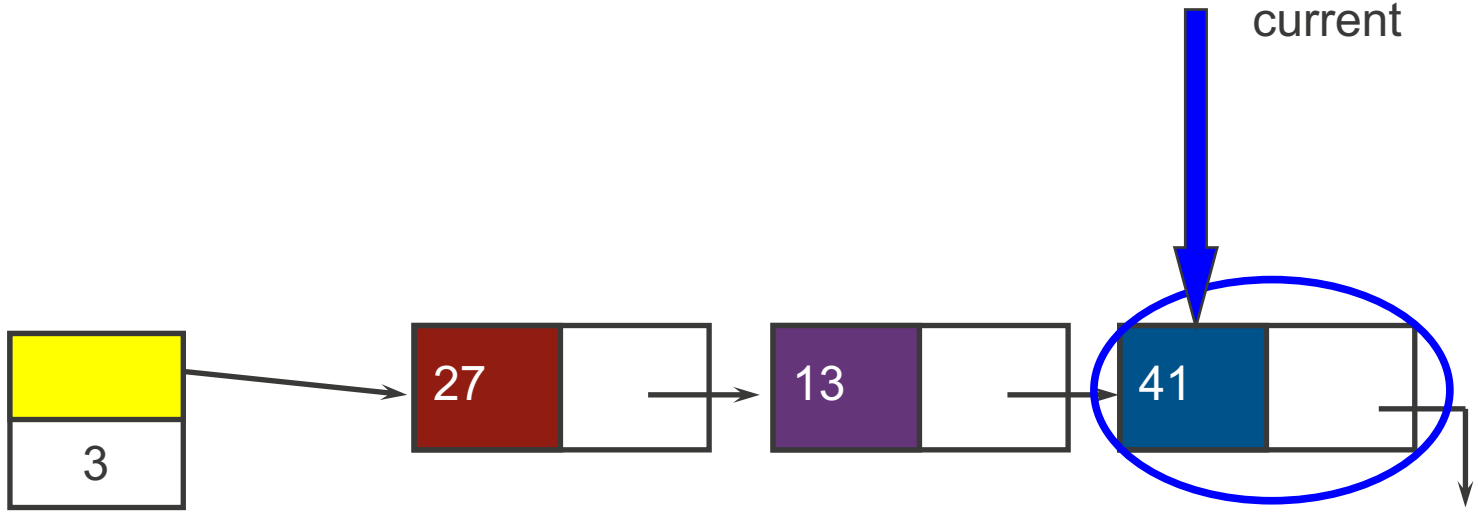
Let's see how to search for the node at index 2

- If the index is within the bounds (0 <= and < len(self))
- Start with the node at the head
- If not the index, follow the link to the next node
- Until we reach the index



Let's see how to search for the node at index 2

- If the index is within the bounds (0 <= and < len(self))
- Start with the node at the head
- If not the index, follow the link to the next node
- Until we reach the index
- Then, return (a pointer to) the node



__get_node_at_index method

```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

index times

- **Complexity?**
  - The loop executes index times always
    - Remember, erroneous situations (like index = -1) are not considered for Big O
  - All operations constant
  - Best = Worst = O(index)

__get_node_at_index method

```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

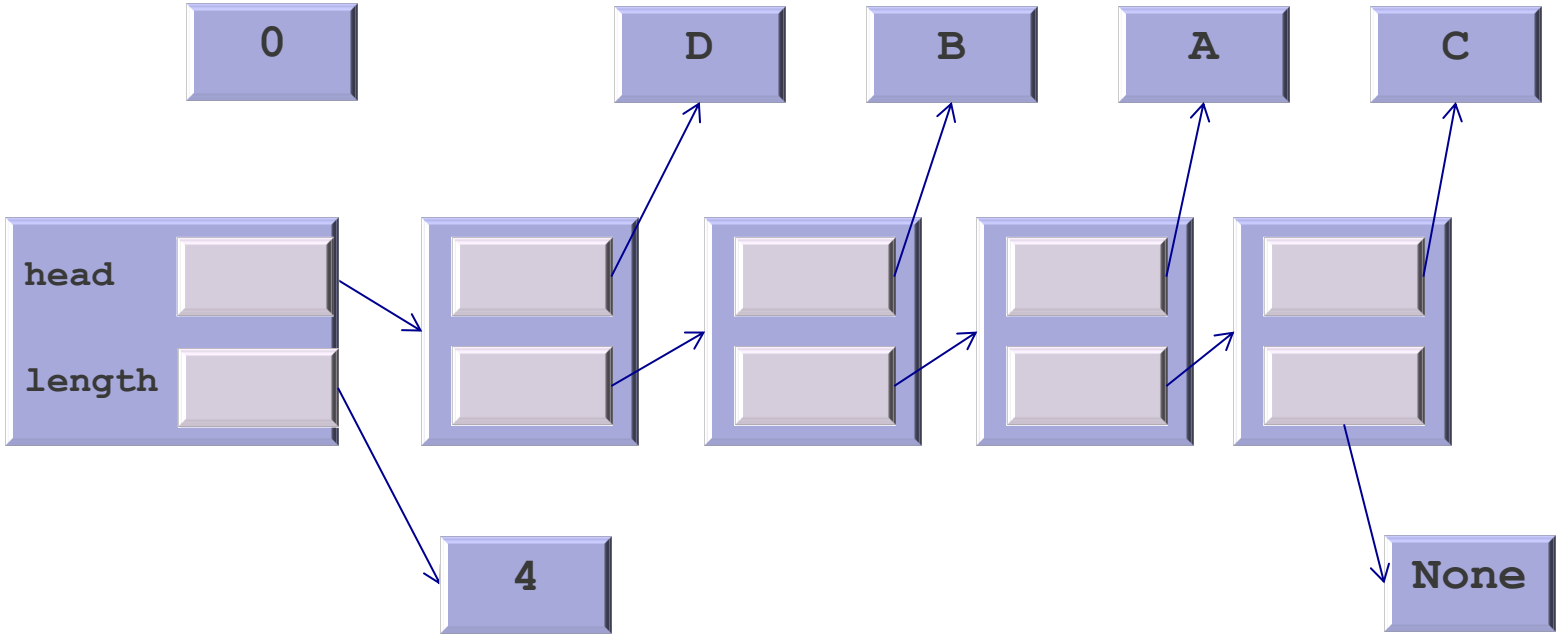
Main 3 steps for traversing any linked data structure

This is a classic linear search

- **Complexity?**
  - The loop executes index times always
    - Remember, erroneous situations (like index = -1) are not considered for BigO
  - All operations constant
  - Best = Worst = O(index)

Consider a list with four nodes whose items are `D,B,A,C.`

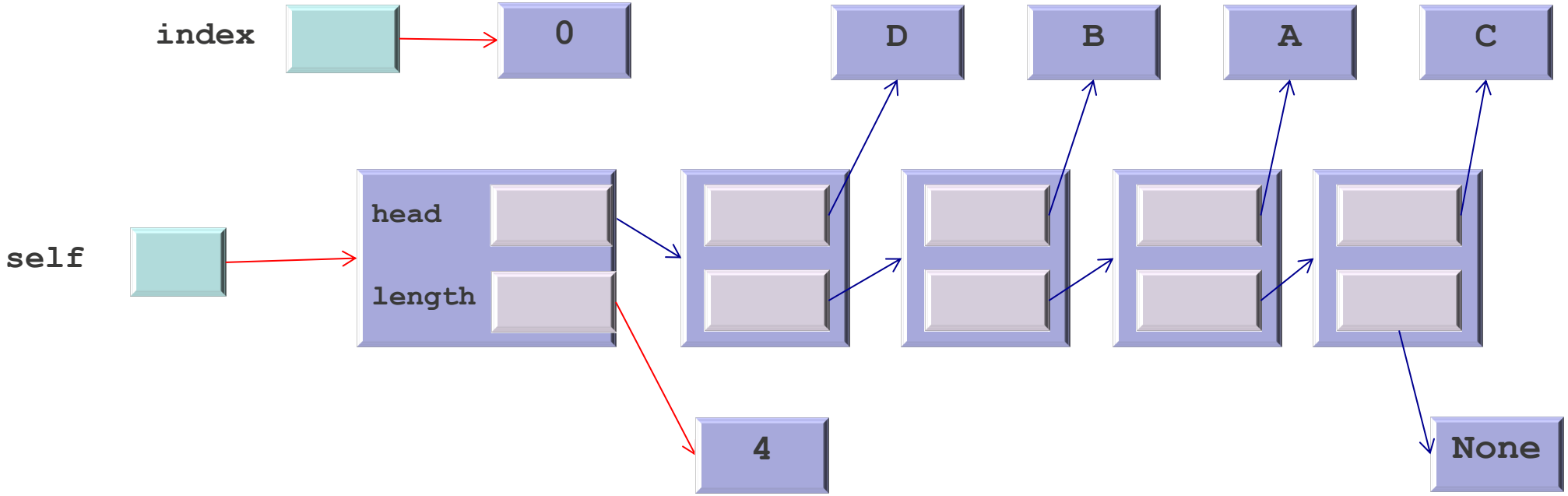
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

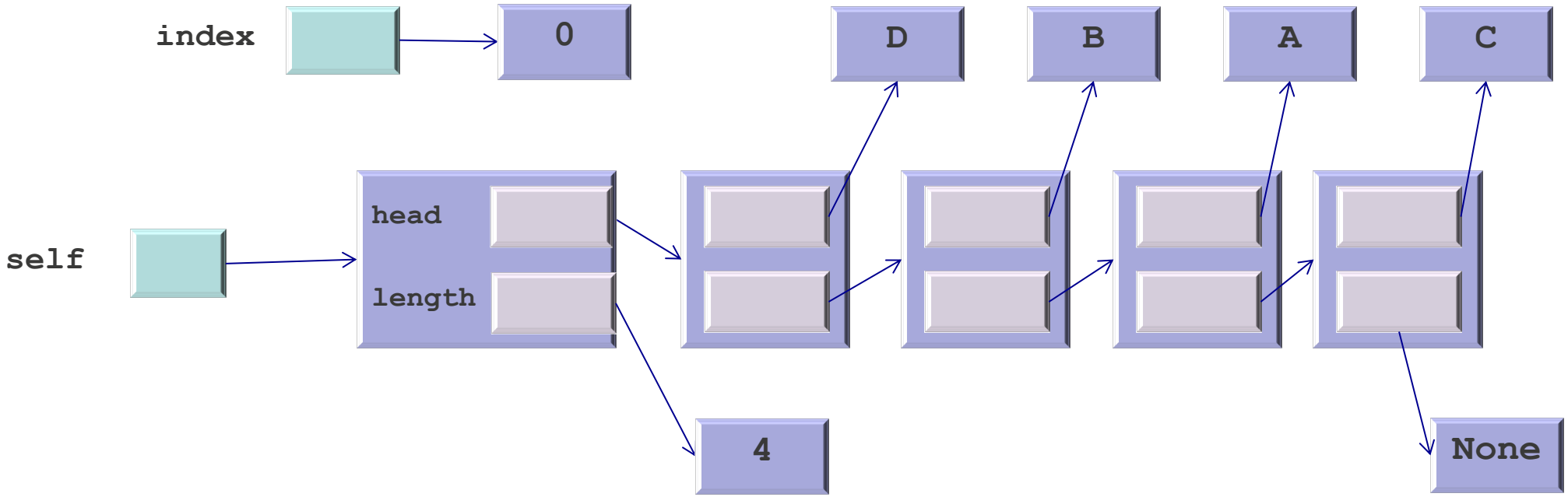
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

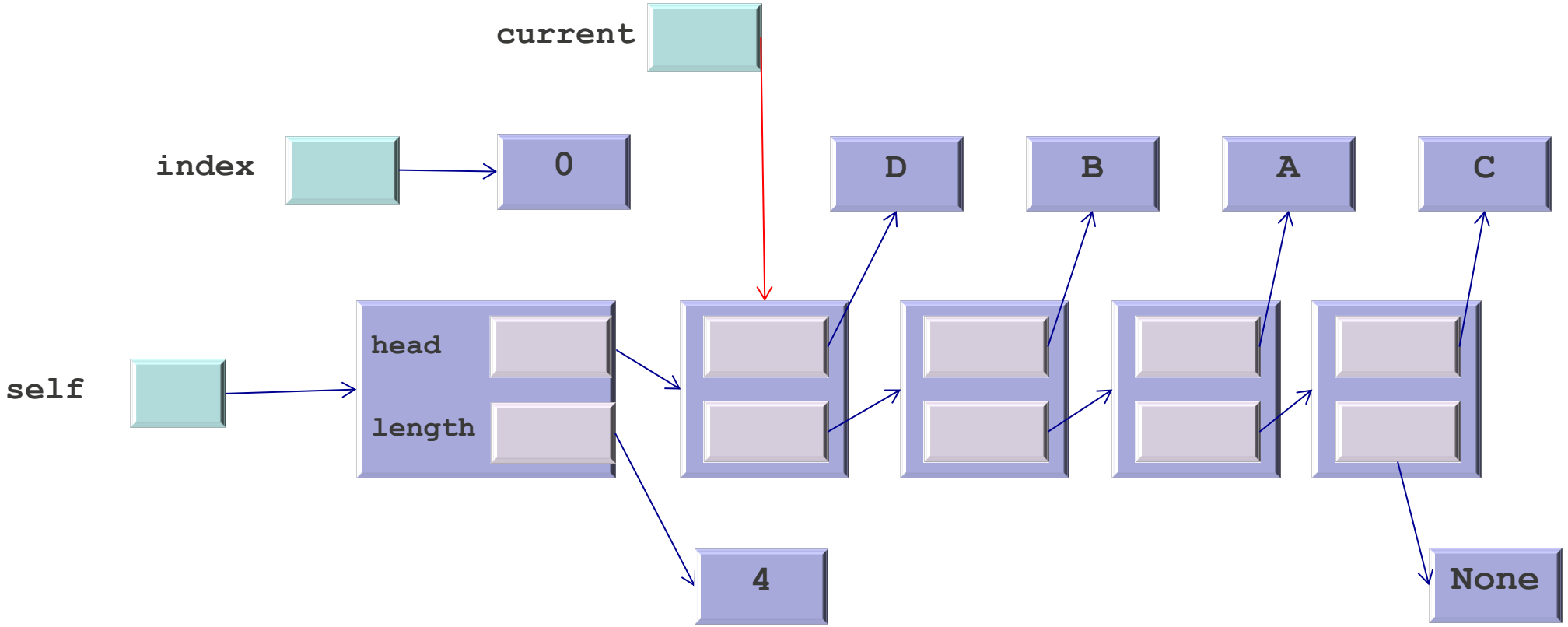
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

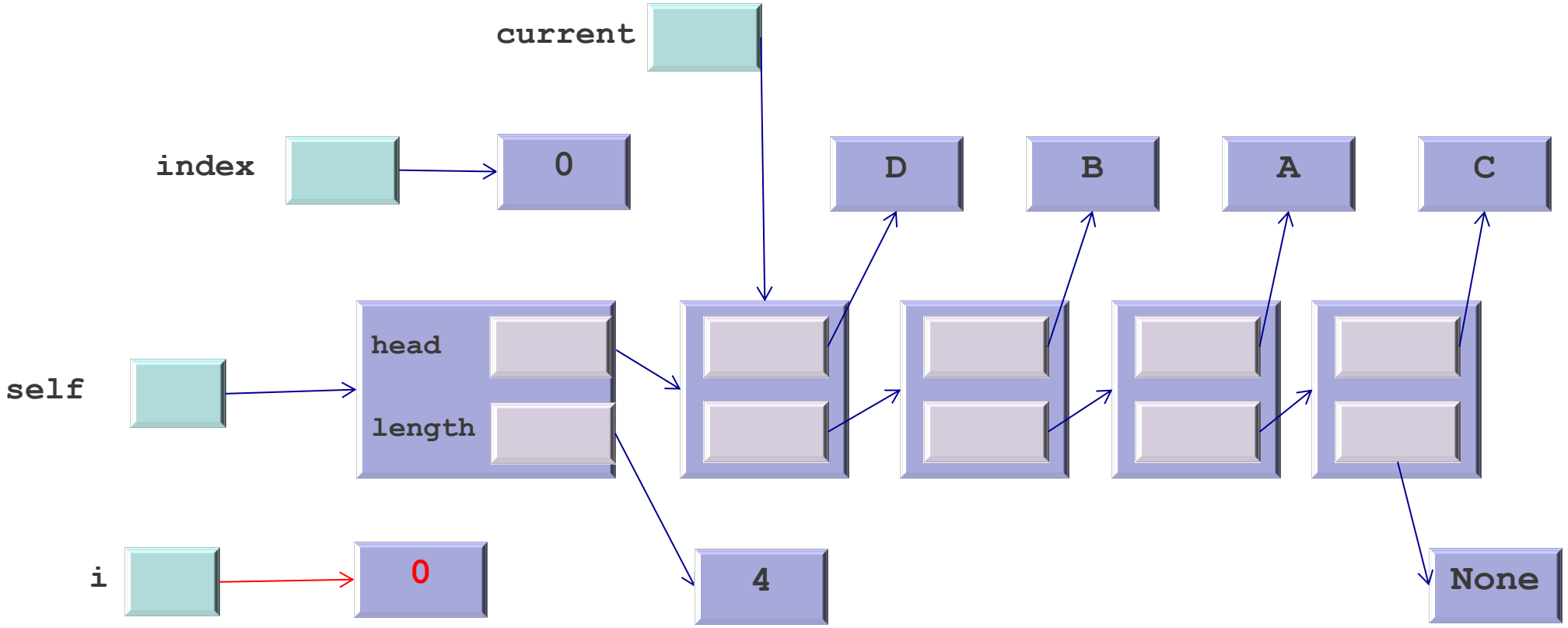
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

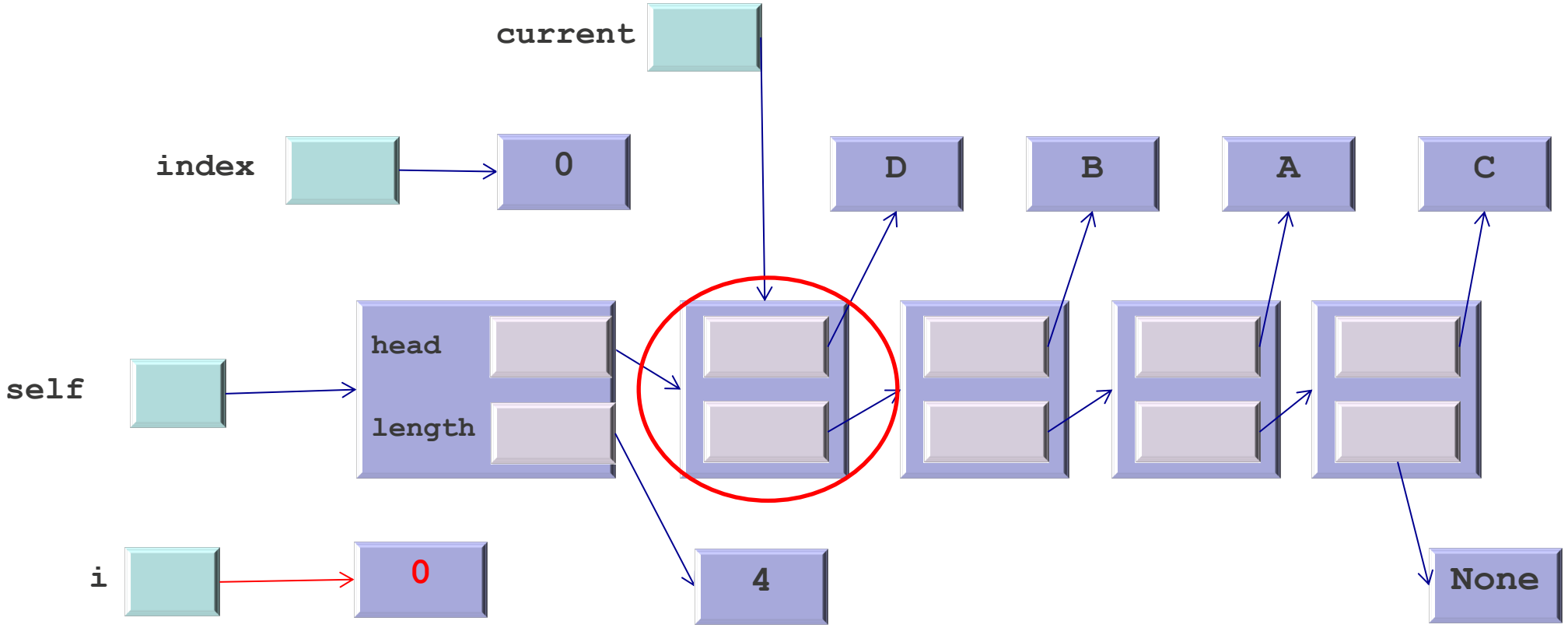
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

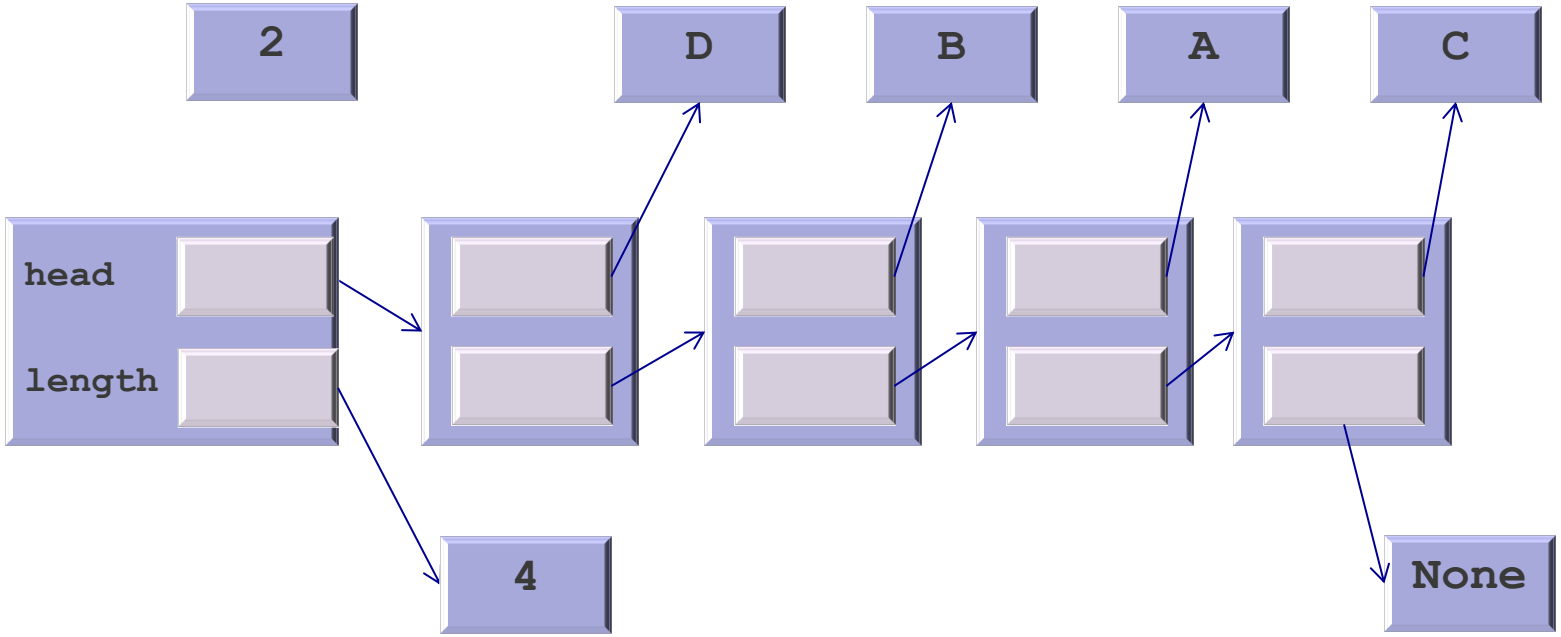
Let's search for index 0




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

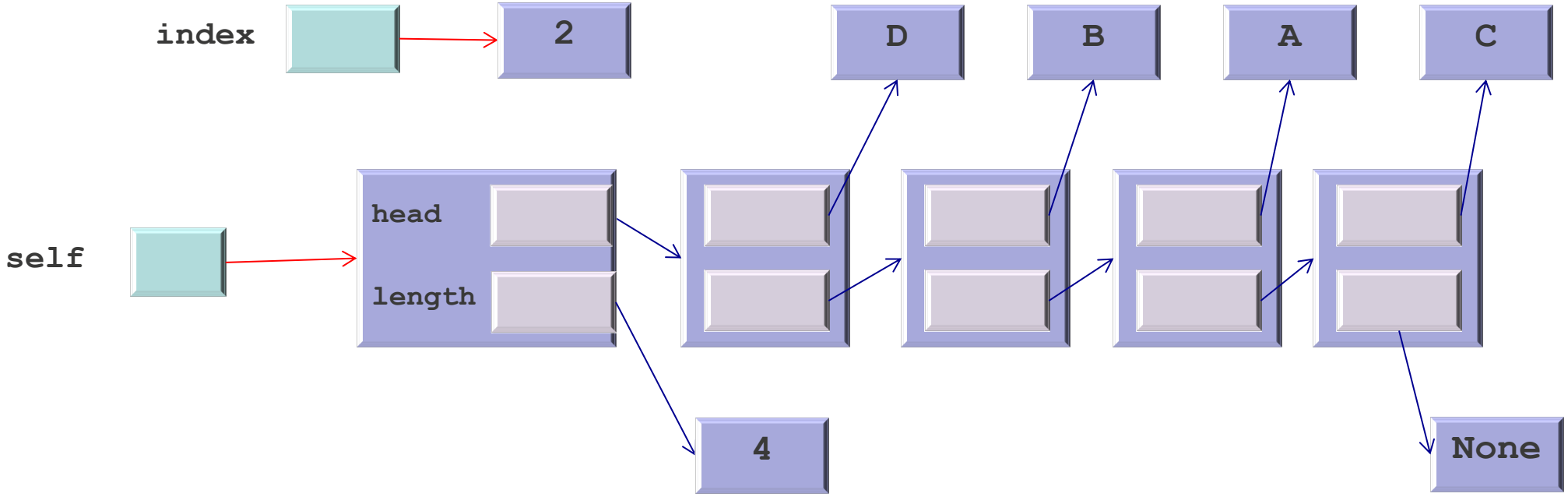
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

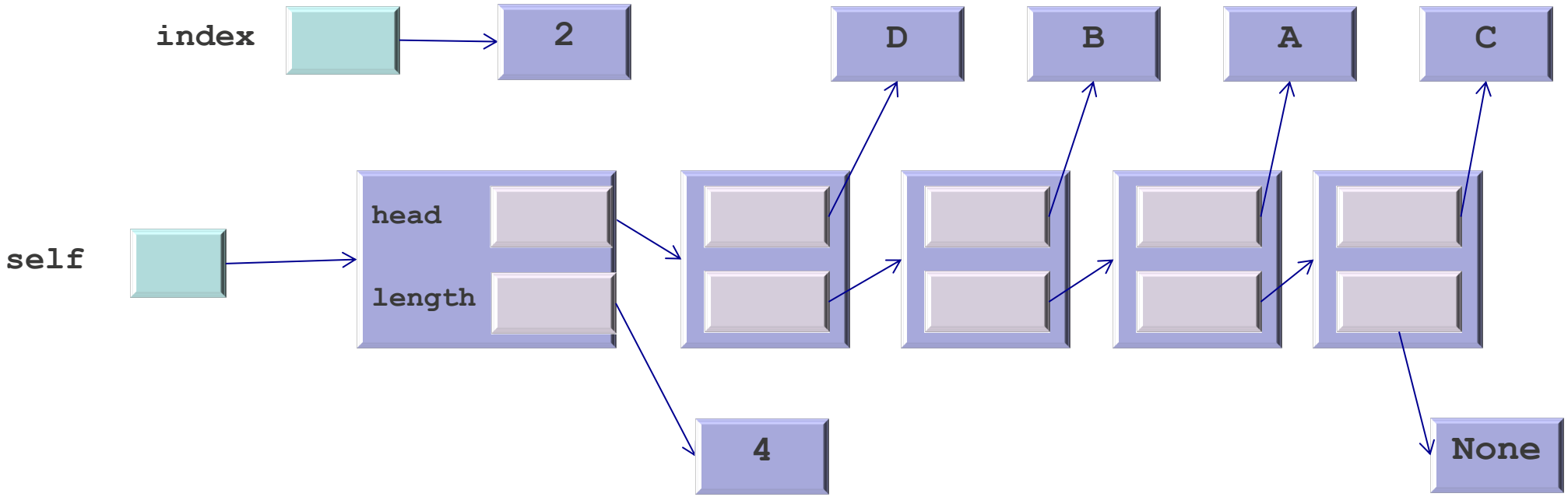
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

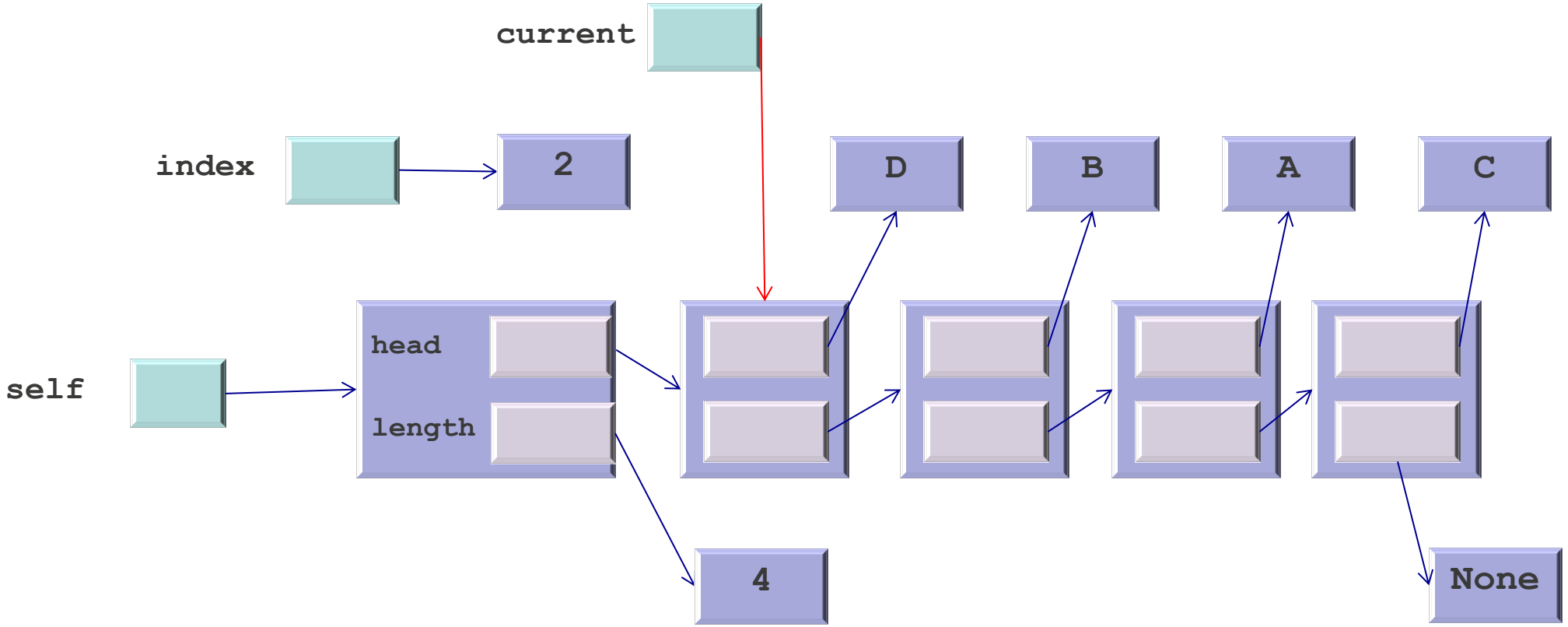
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

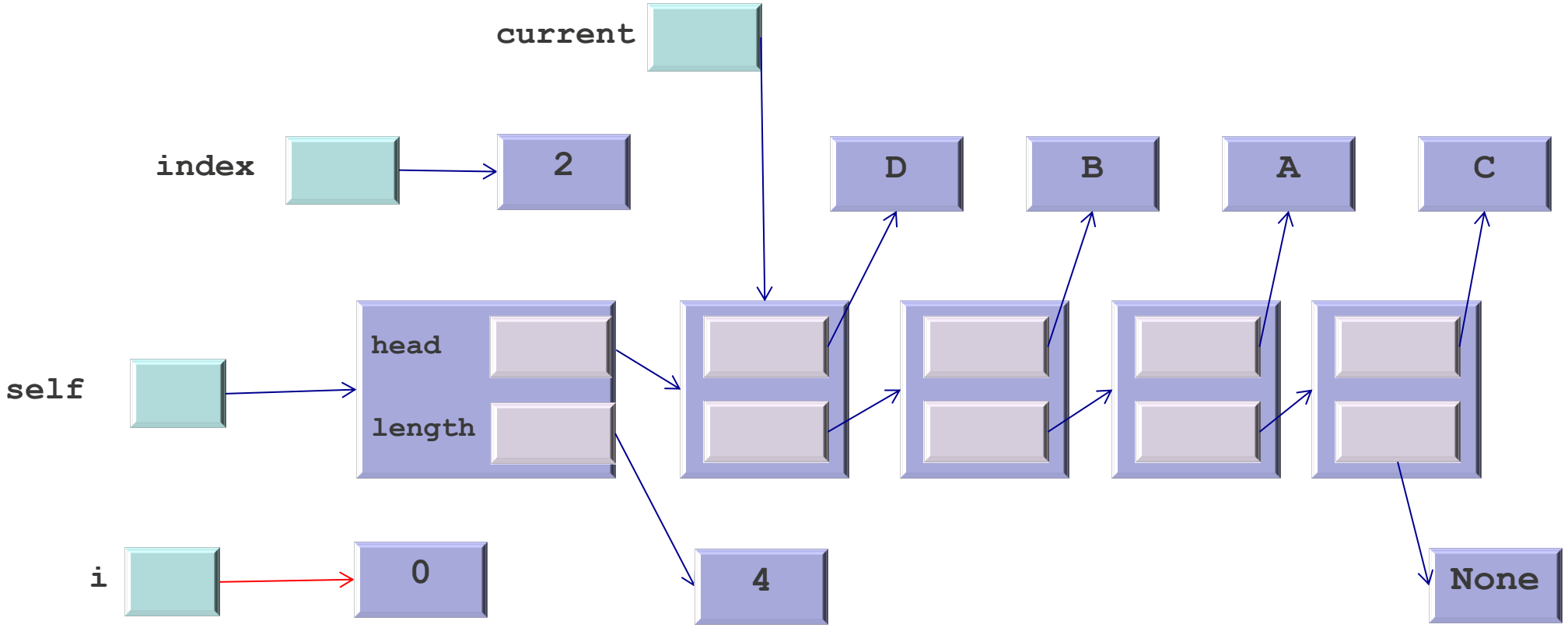
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

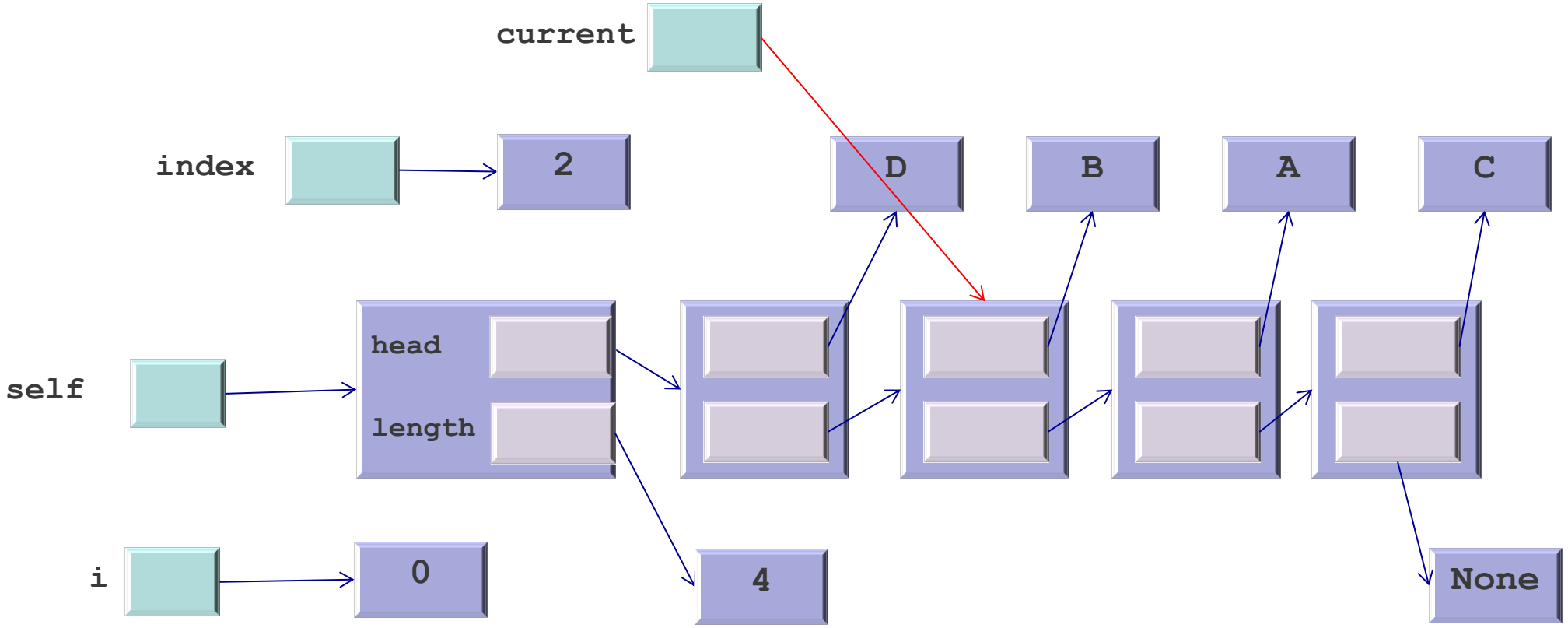
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

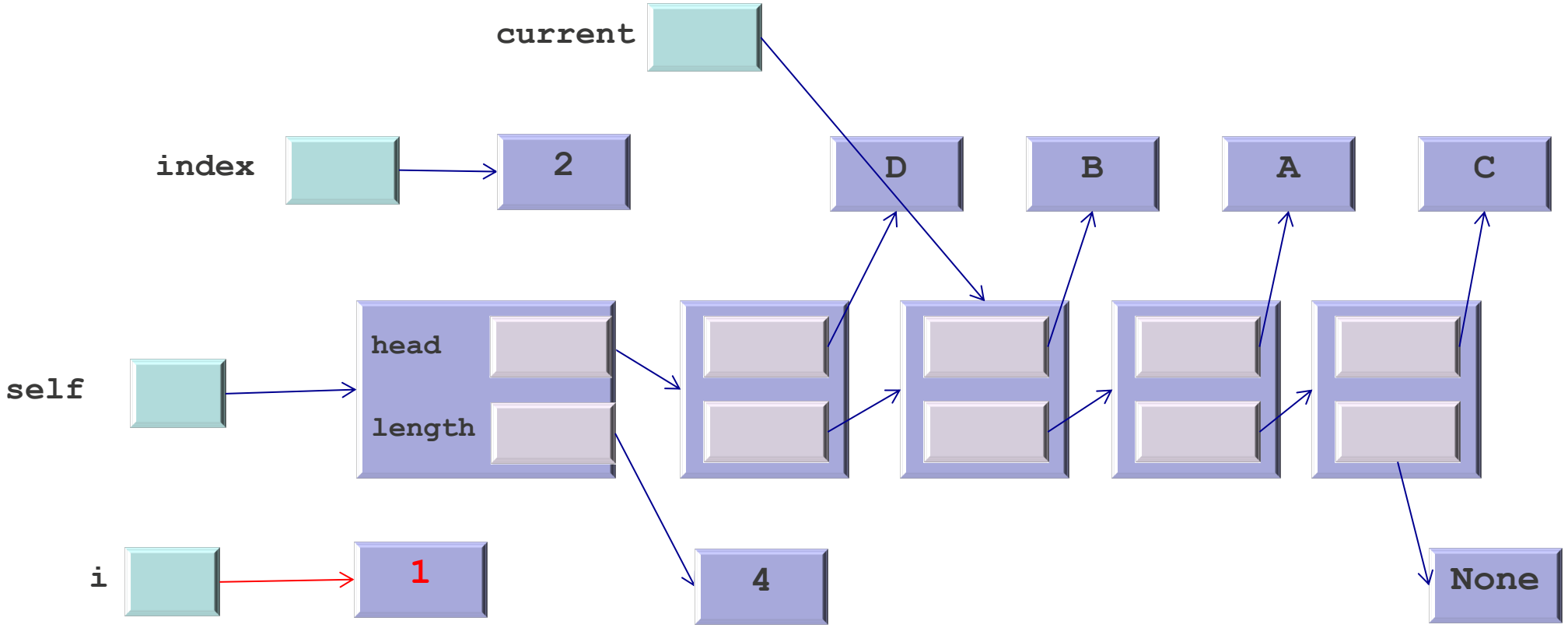
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

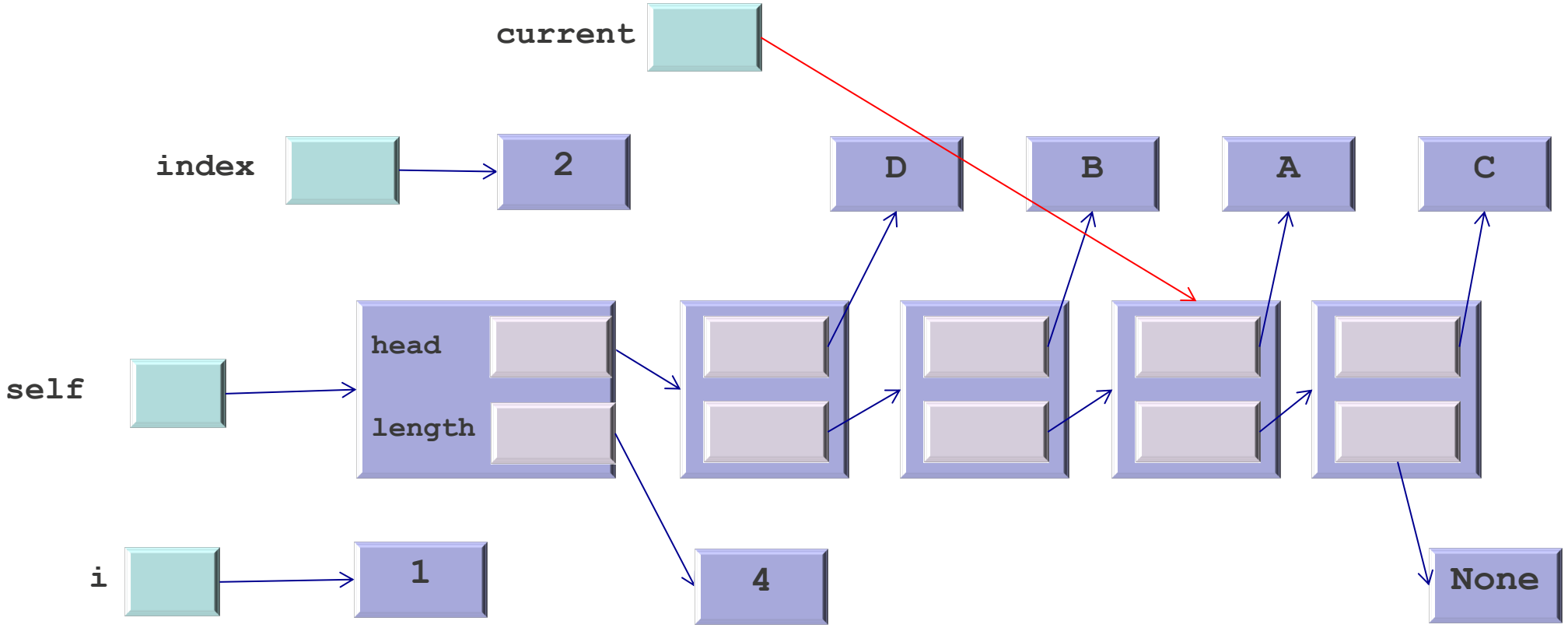
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

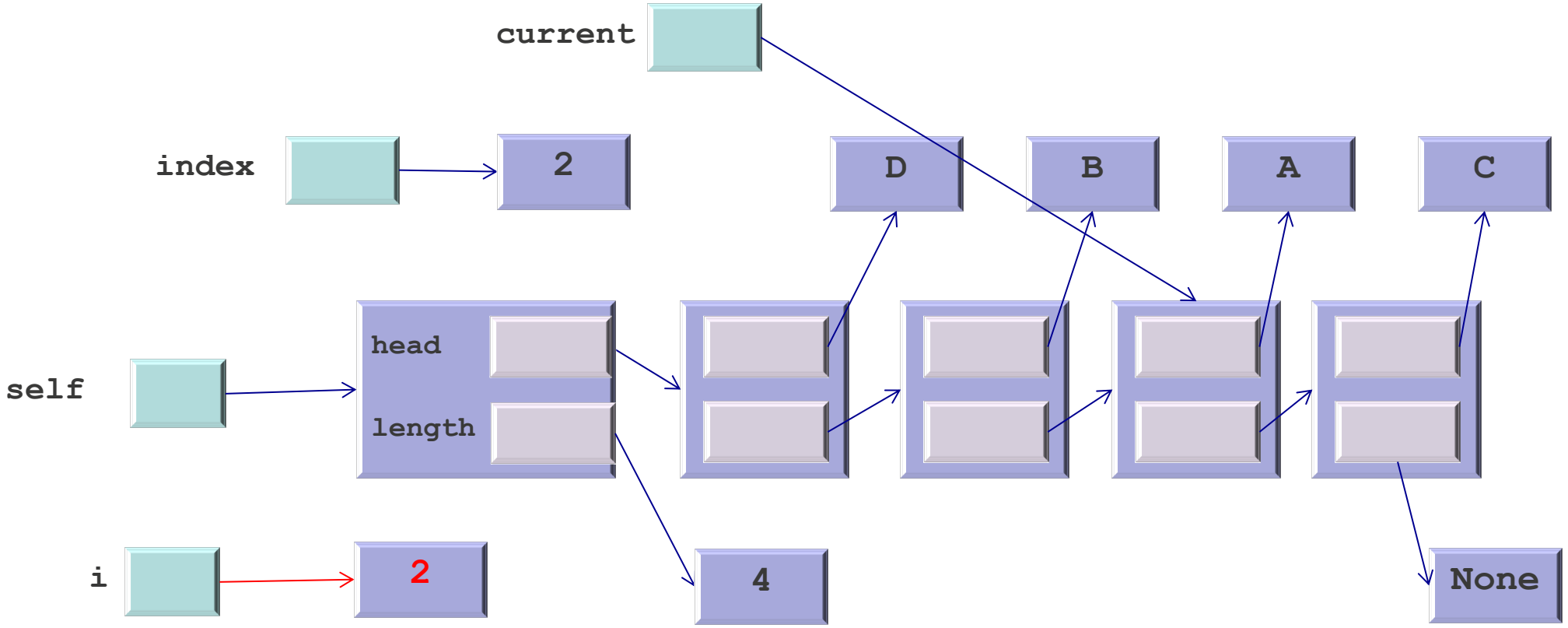
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

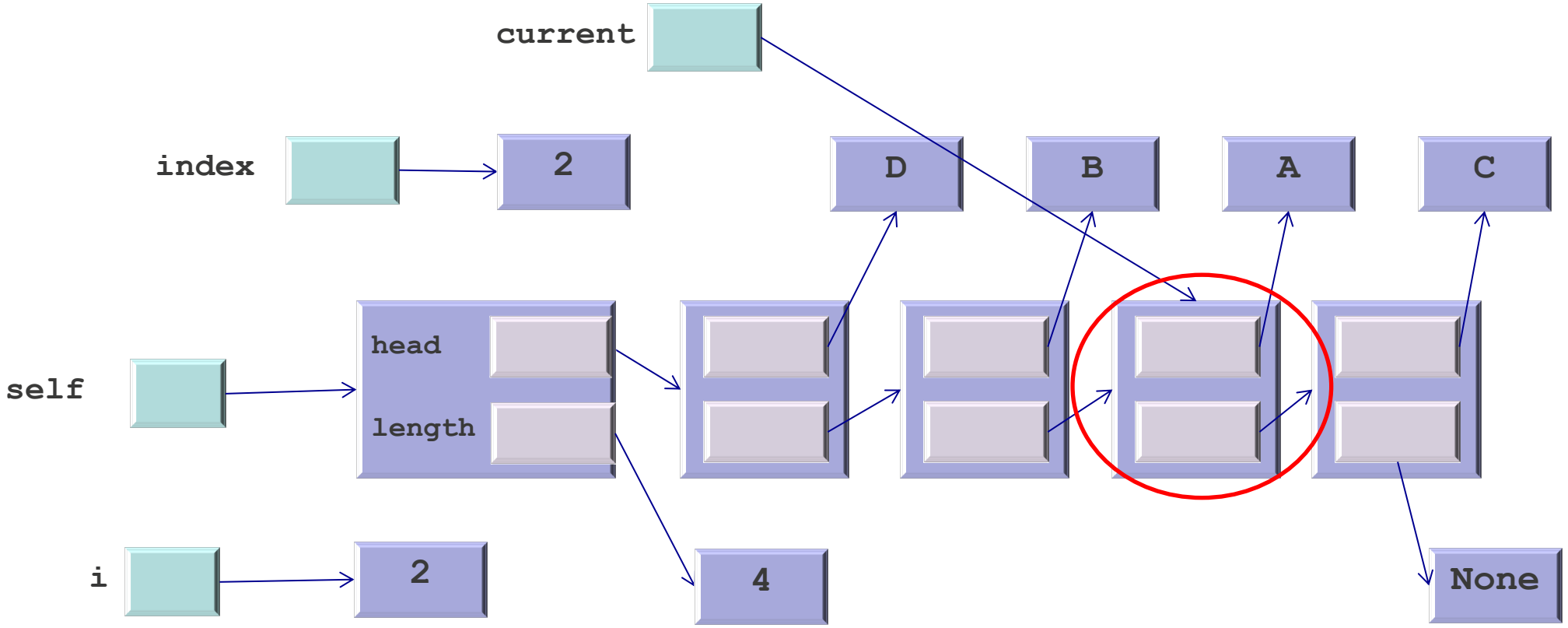
Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

Consider a list with four nodes whose items are `D,B,A,C.`

Let's search for index 2




```
def __get_node_at_index(self, index: int) -> Node[T]:
    if 0 <= index and index < len(self):
        current = self.head
        for i in range(index):
            current = current.link
        return current
    else:
        raise ValueError("Index out of bounds")
```

MONASH University

Implementing delete_at_index

When and how to we use our internal method

- **In the definition of many of our methods, such as:**

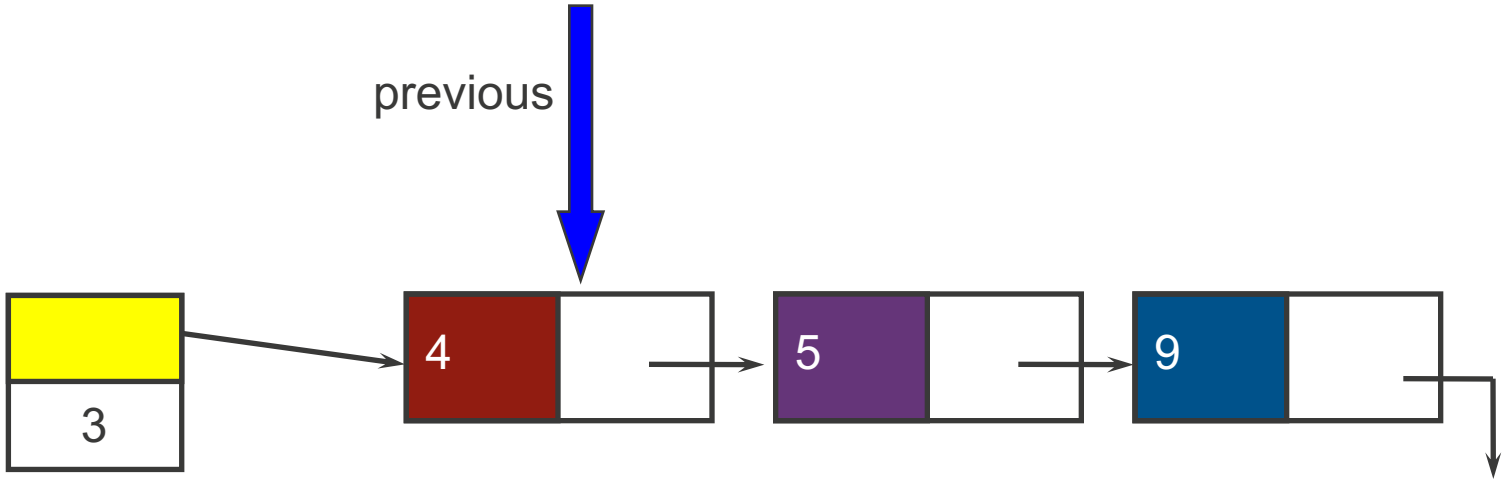
```
def __setitem__(self, index: int, item: T) -> None:
    node_at_index = self.__get_node_at_index(index)
    node_at_index.item = item


def __getitem__(self, index: int) -> T:
    node_at_index = self.__get_node_at_index(index)
    return node_at_index.item
```

- **Complexity?**
  - The same as `.__get_node_at_index`, that is O(index)
- **You can also use the internal method for insert and delete_at_index**

How would we use it for deleting the item at index 1?

- **Find the node previous to the index position**

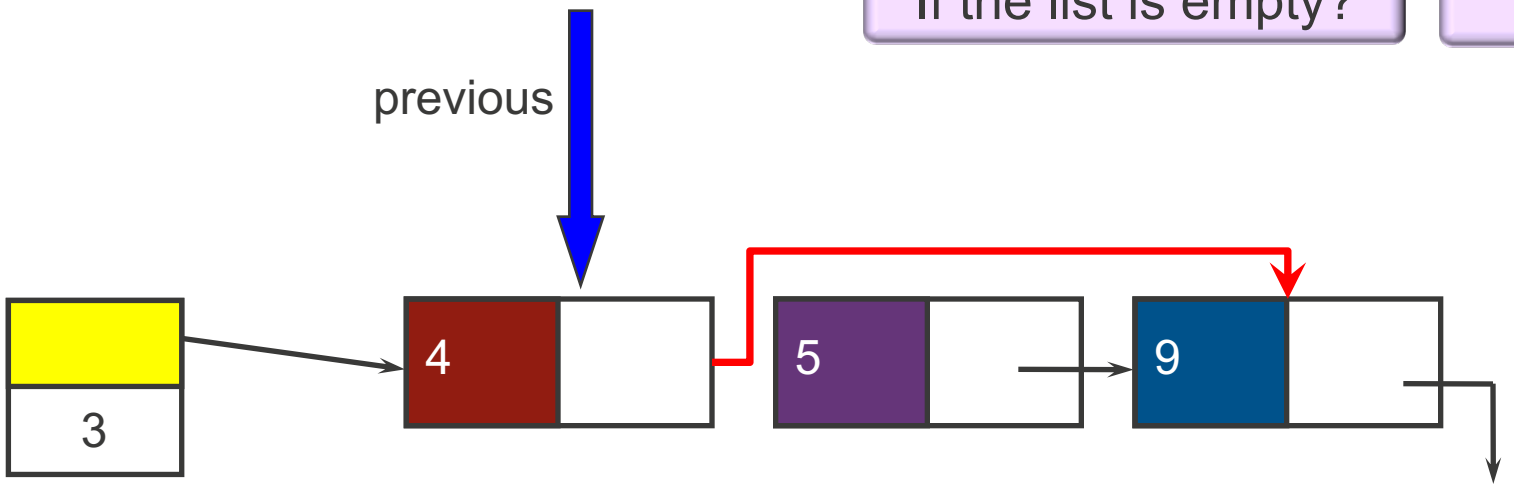


How would we use it for deleting the item at index 1?

- **Find the node previous to the index position**
- **Set the link of this previous node to the link of the next one**

If the index is 0? | Change the head

If the list is empty? | Raise ValueError



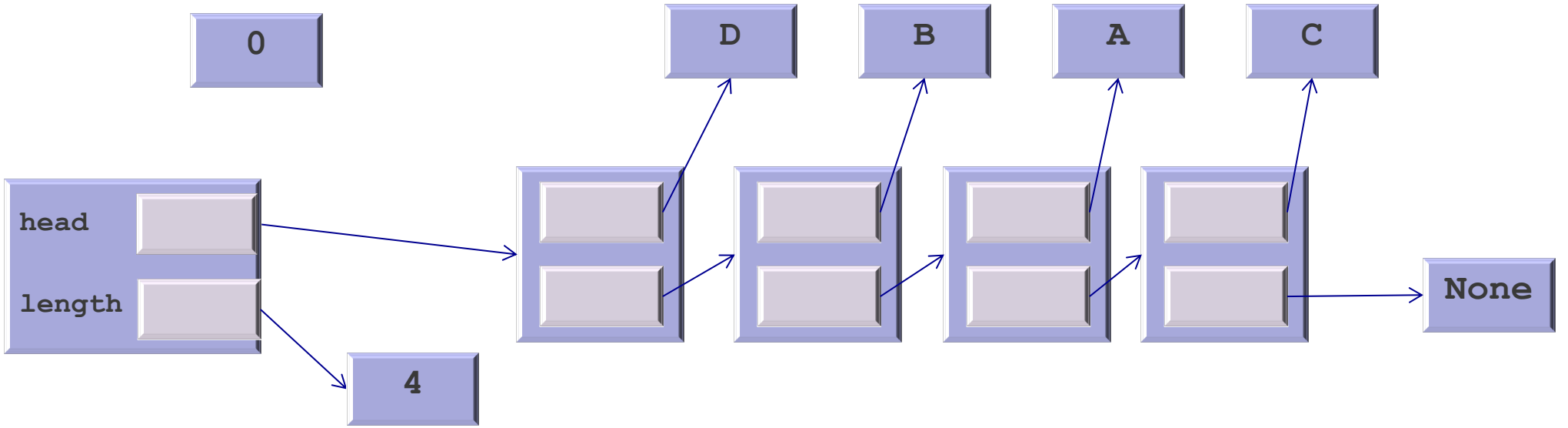
A possible implementation of delete at index

```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("List is empty")
```

- **Complexity?**
  - Everything is constant except **`__get_node_at_index`**, which is O(index)
  - Thus, it is O(index)

Consider a list with four nodes whose items are `D,B,A,C.`

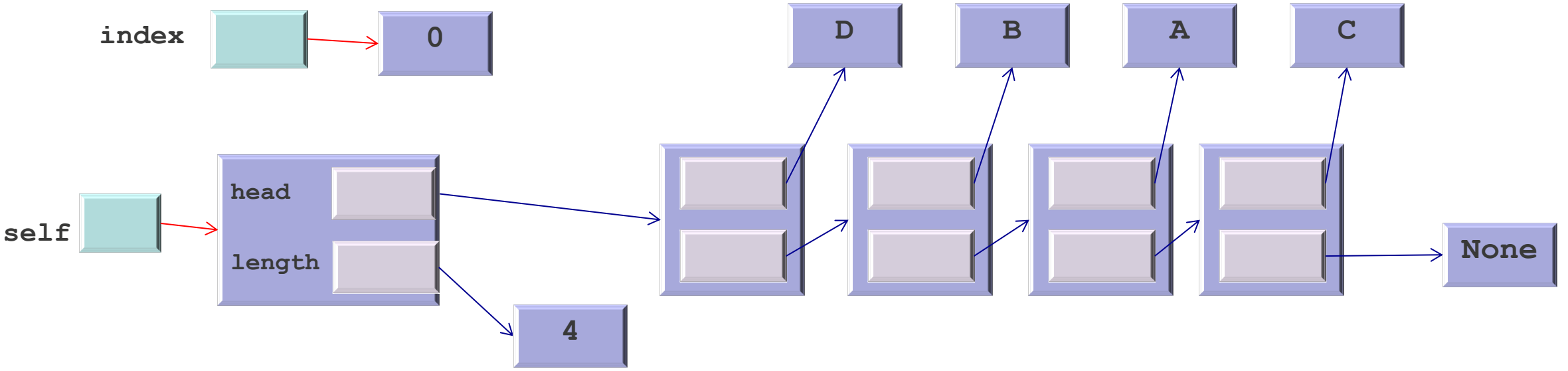
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

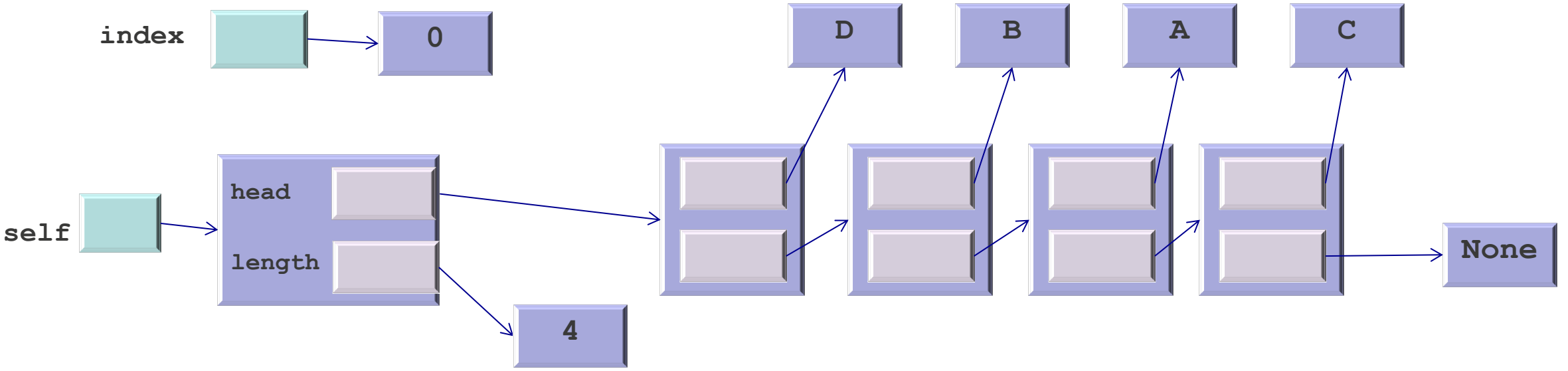
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

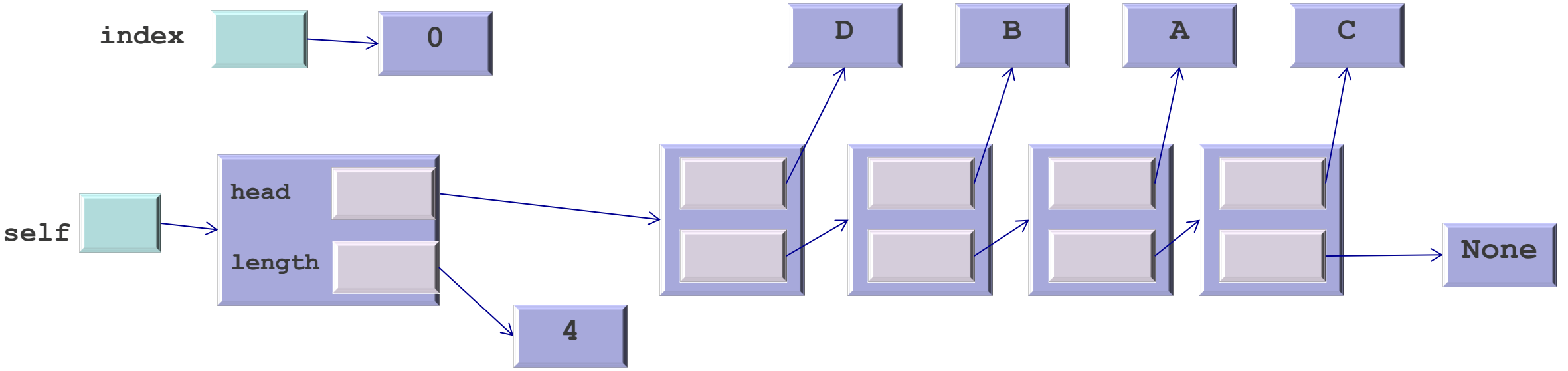
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

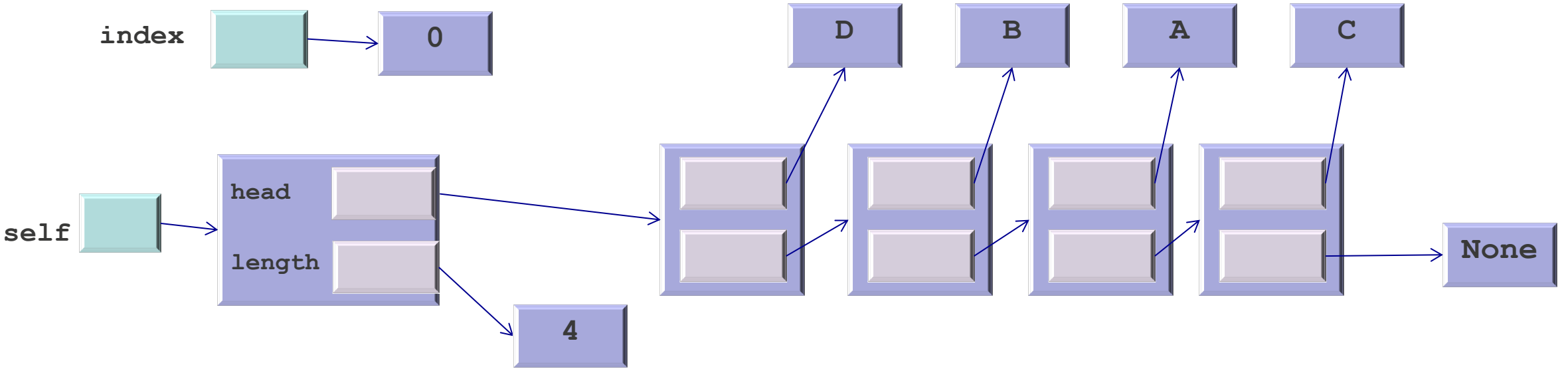
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

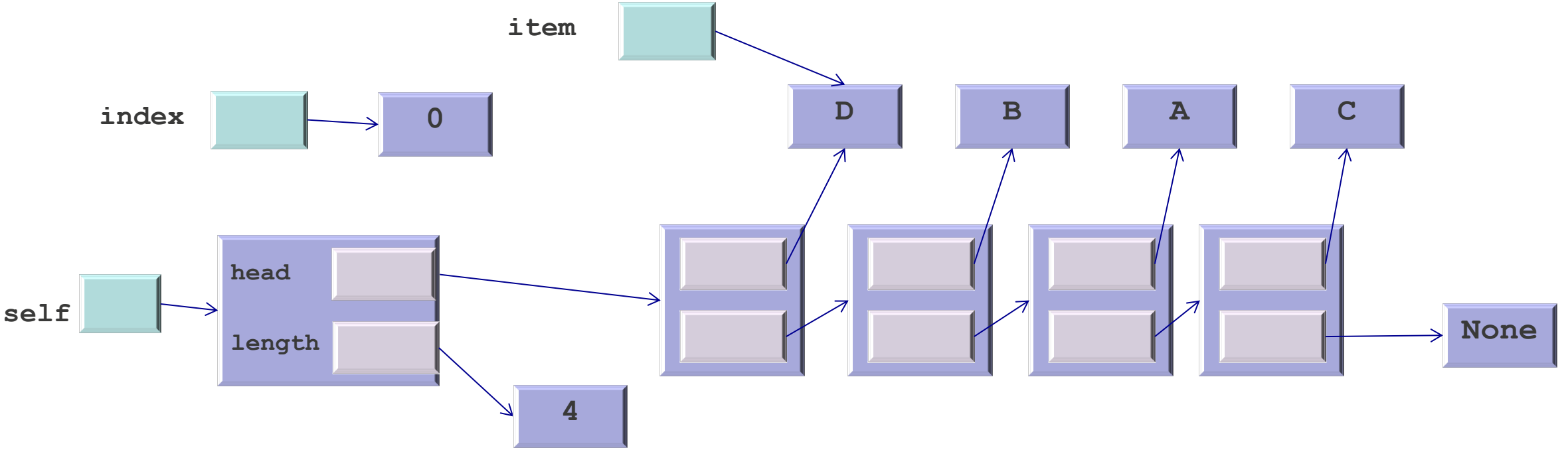
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

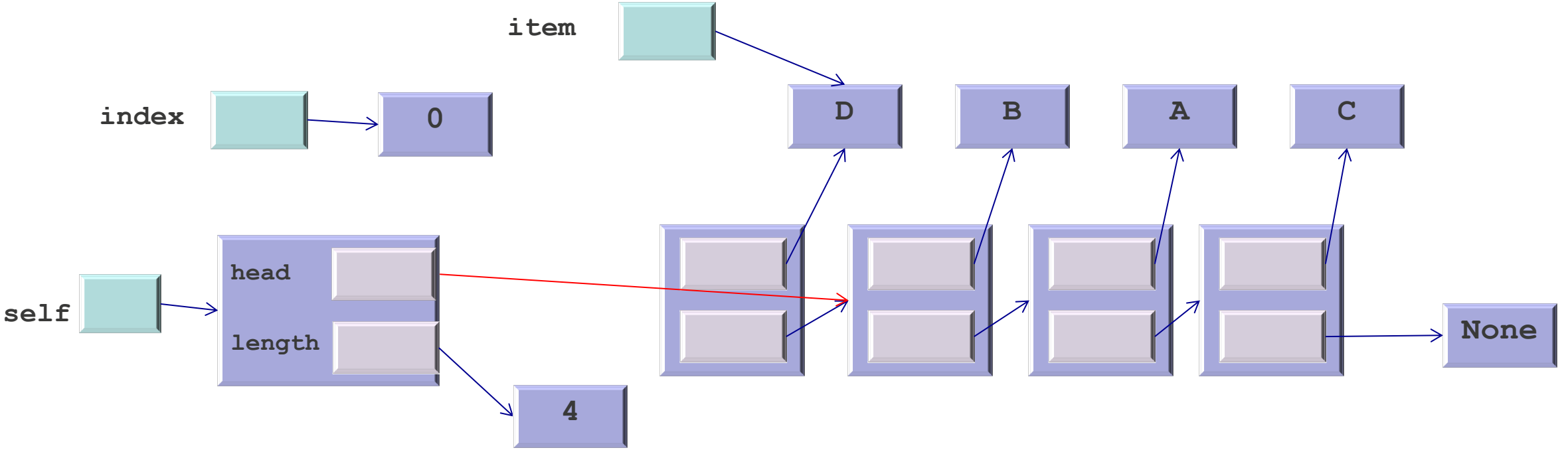
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

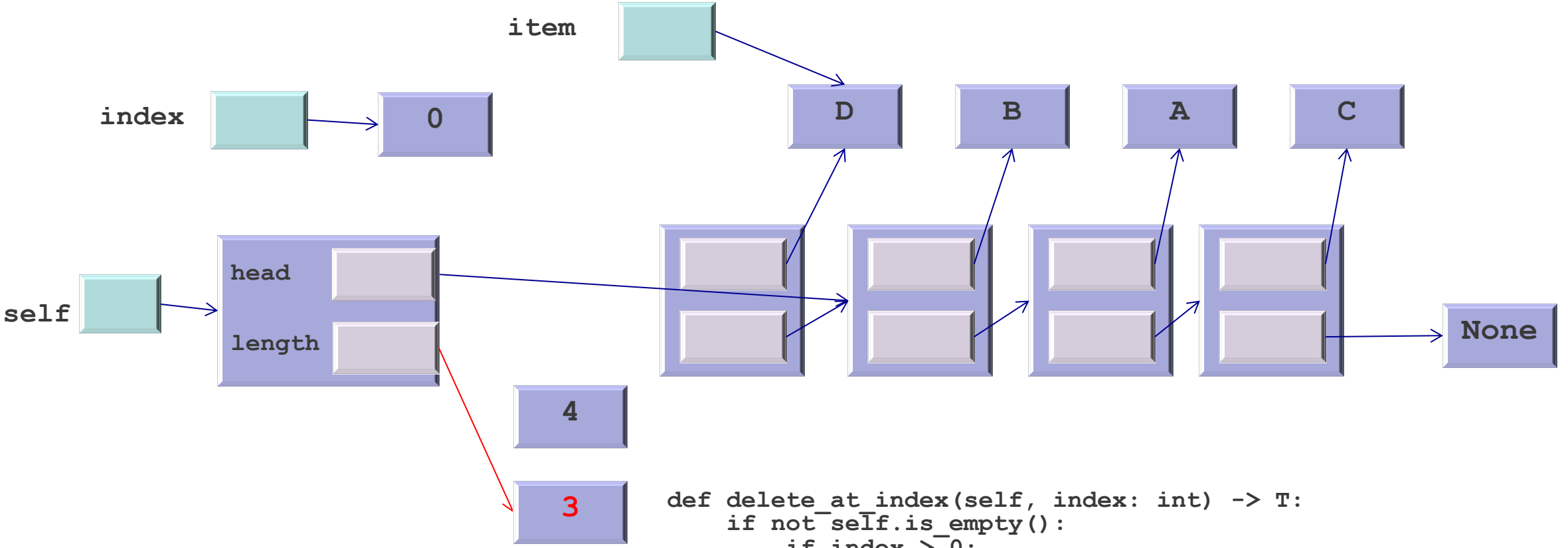
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

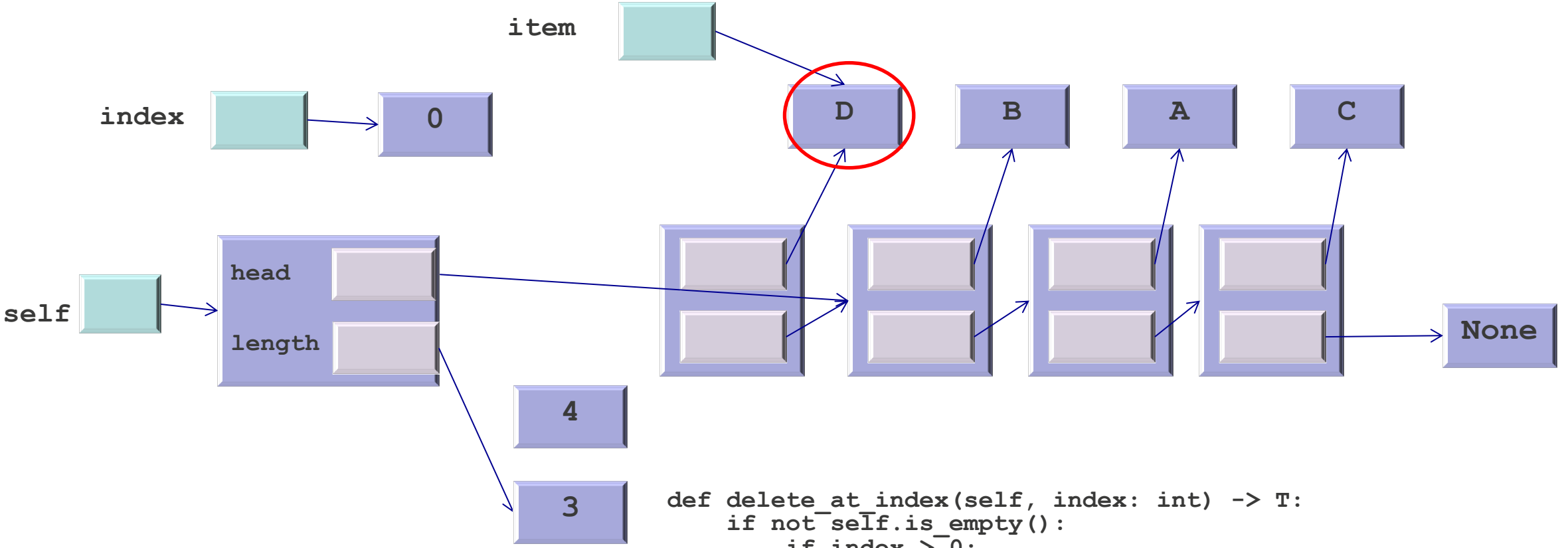
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

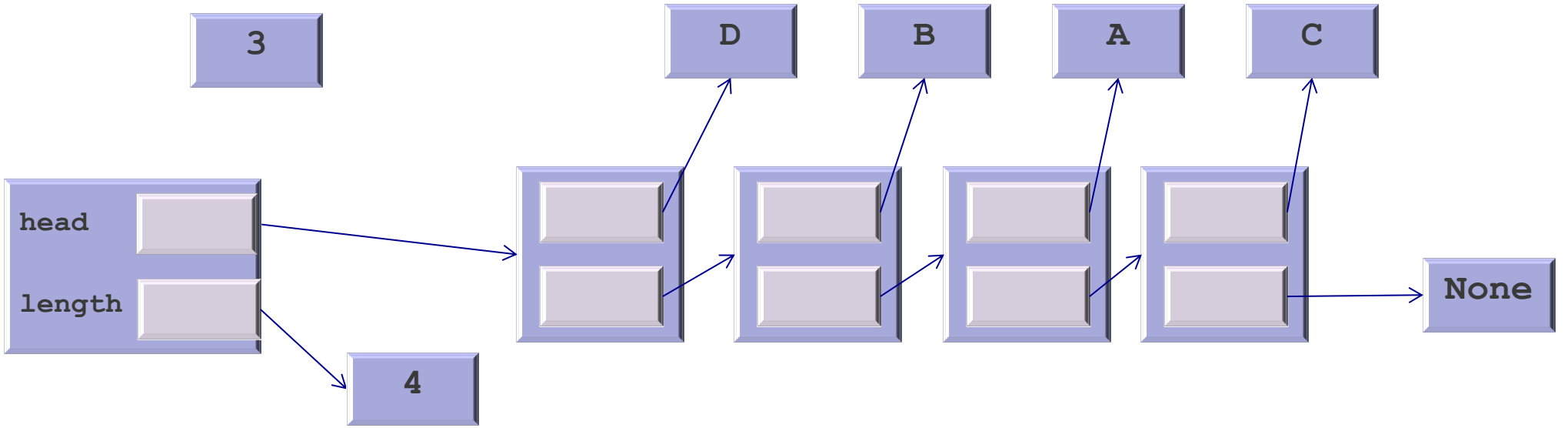
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

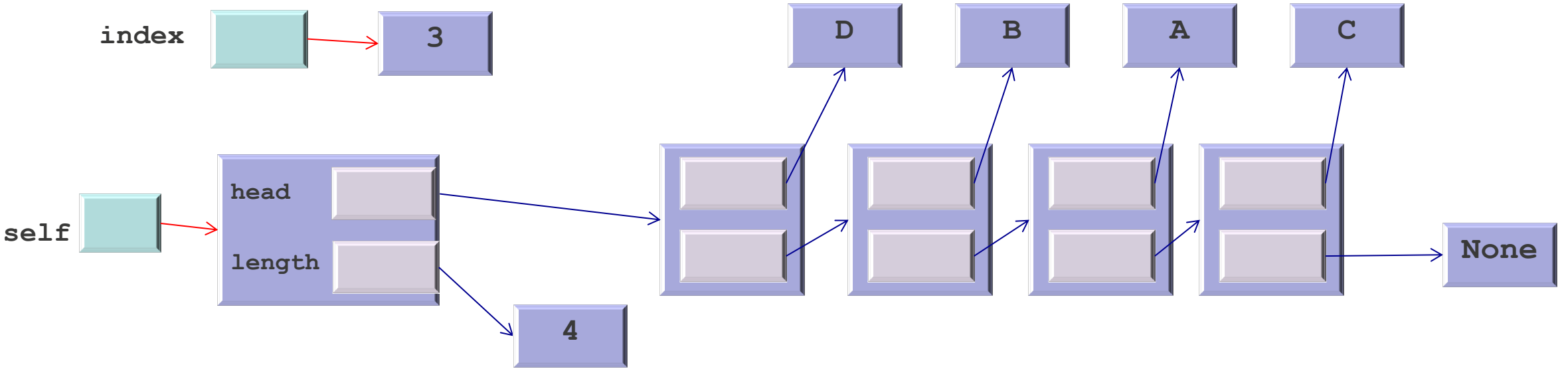
Let's delete at index 3




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

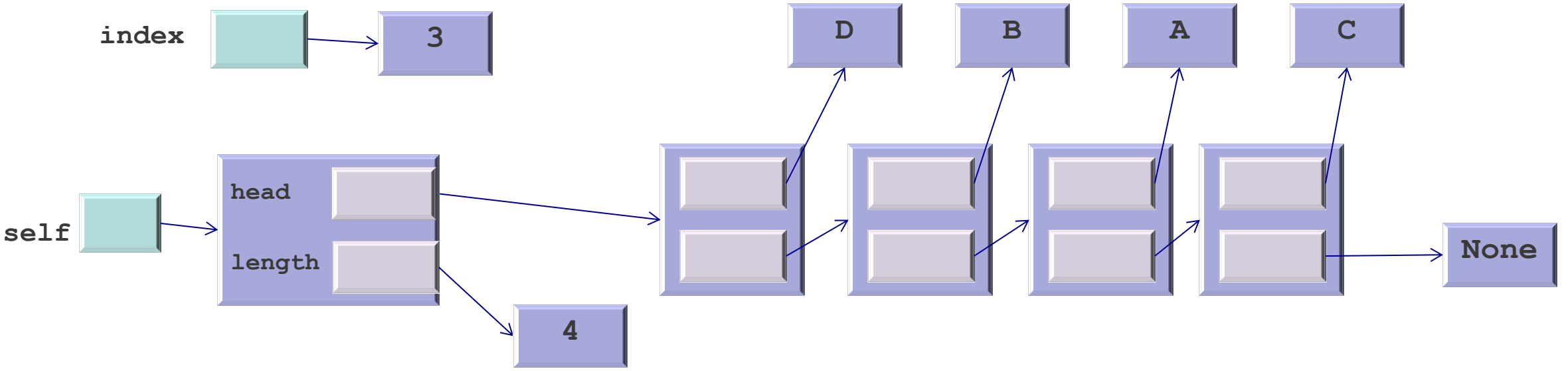
Let's delete at index 3




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

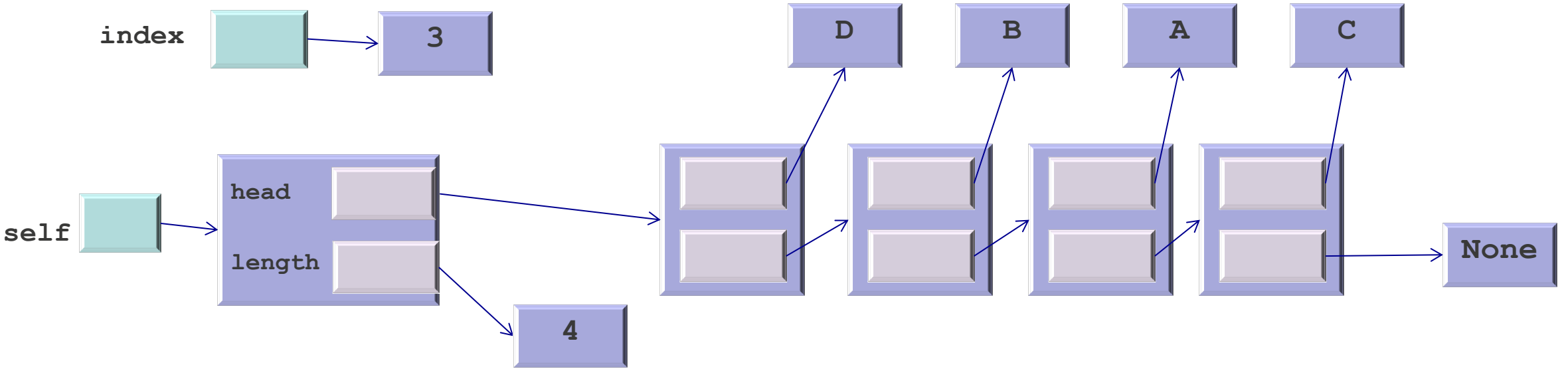
Let's delete at index 3




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

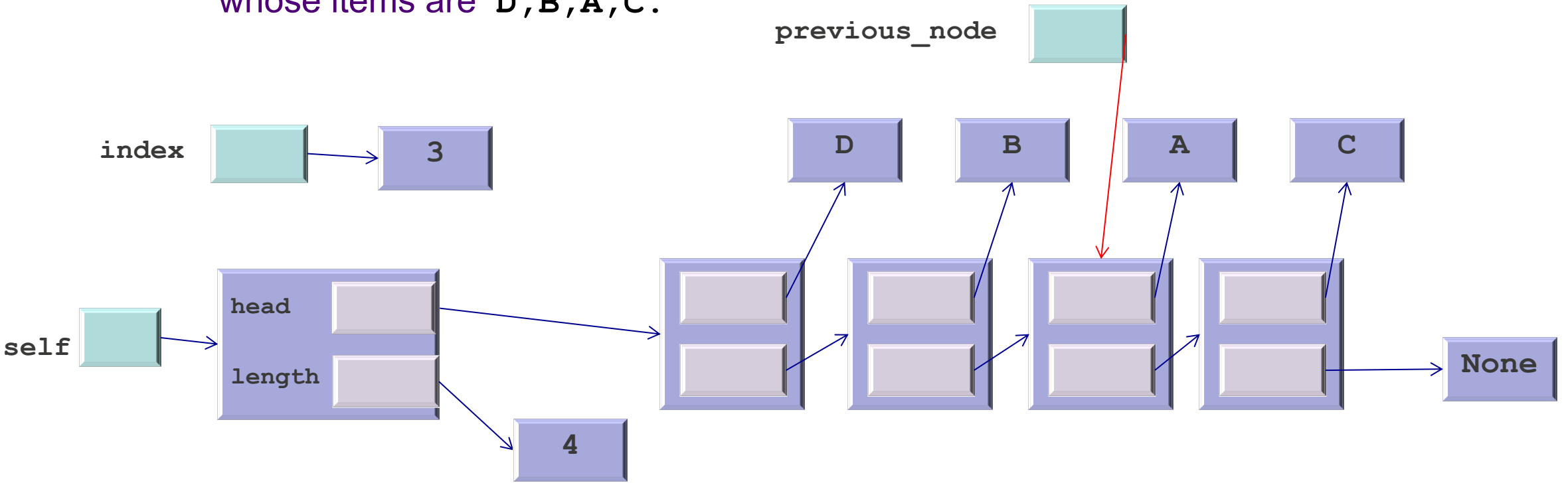
Let's delete at index 3




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

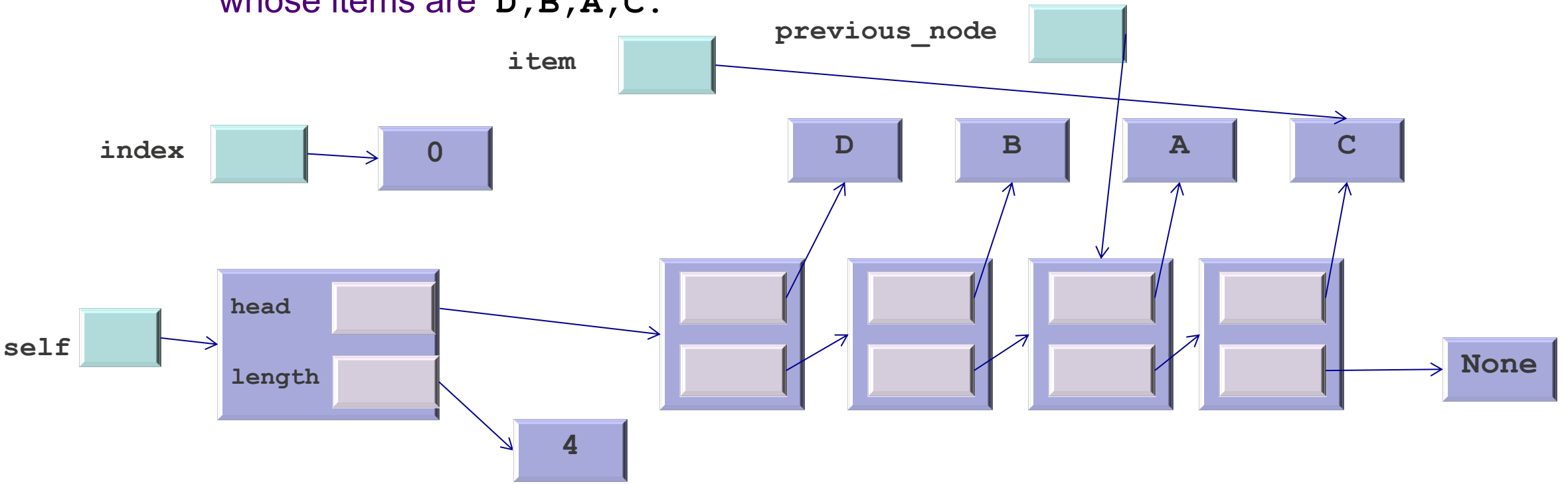
Let's delete at index 3




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

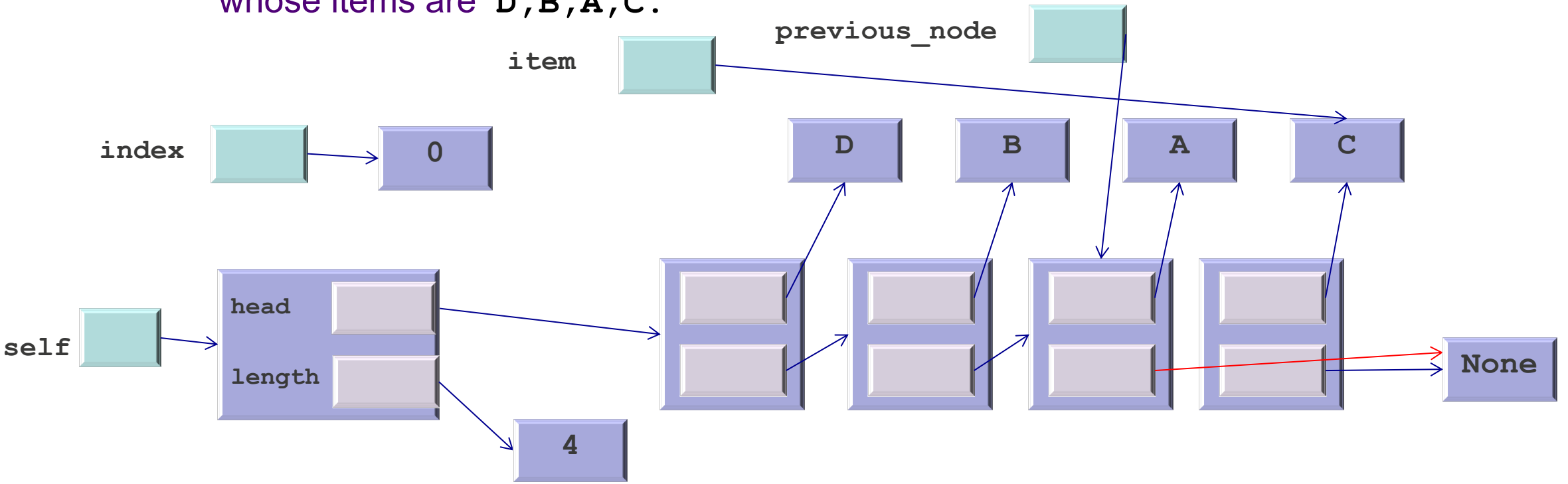
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

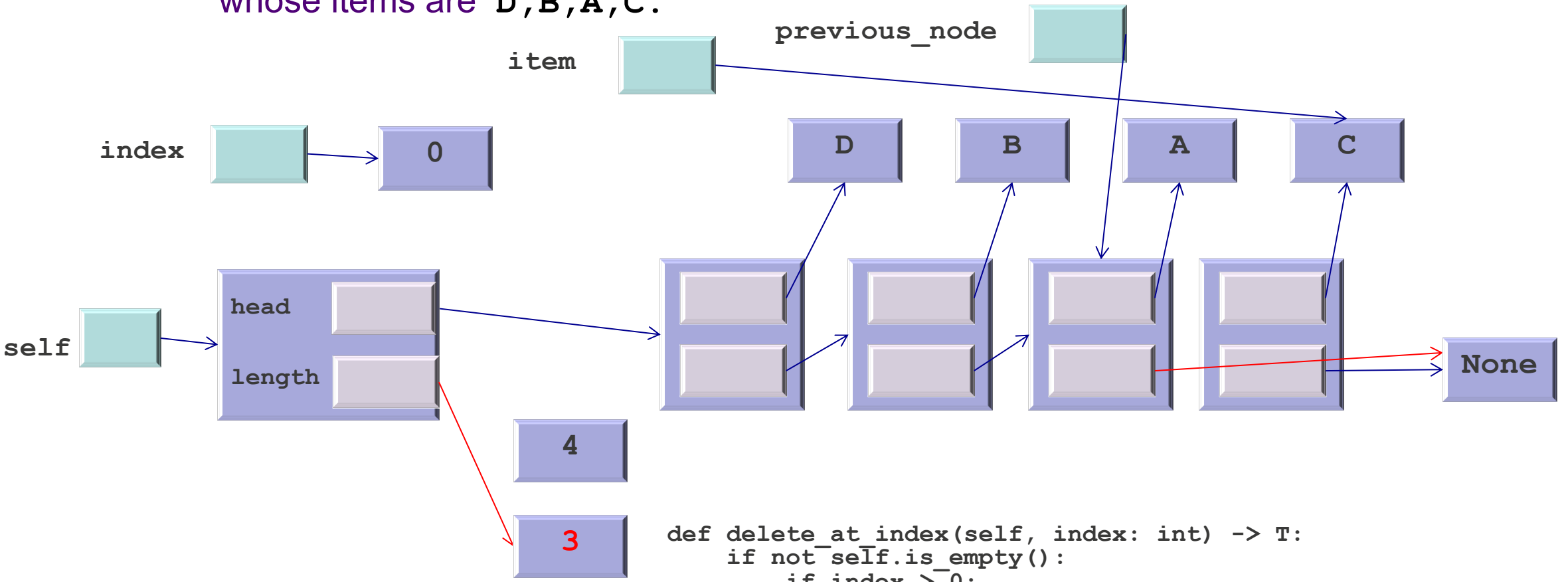
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

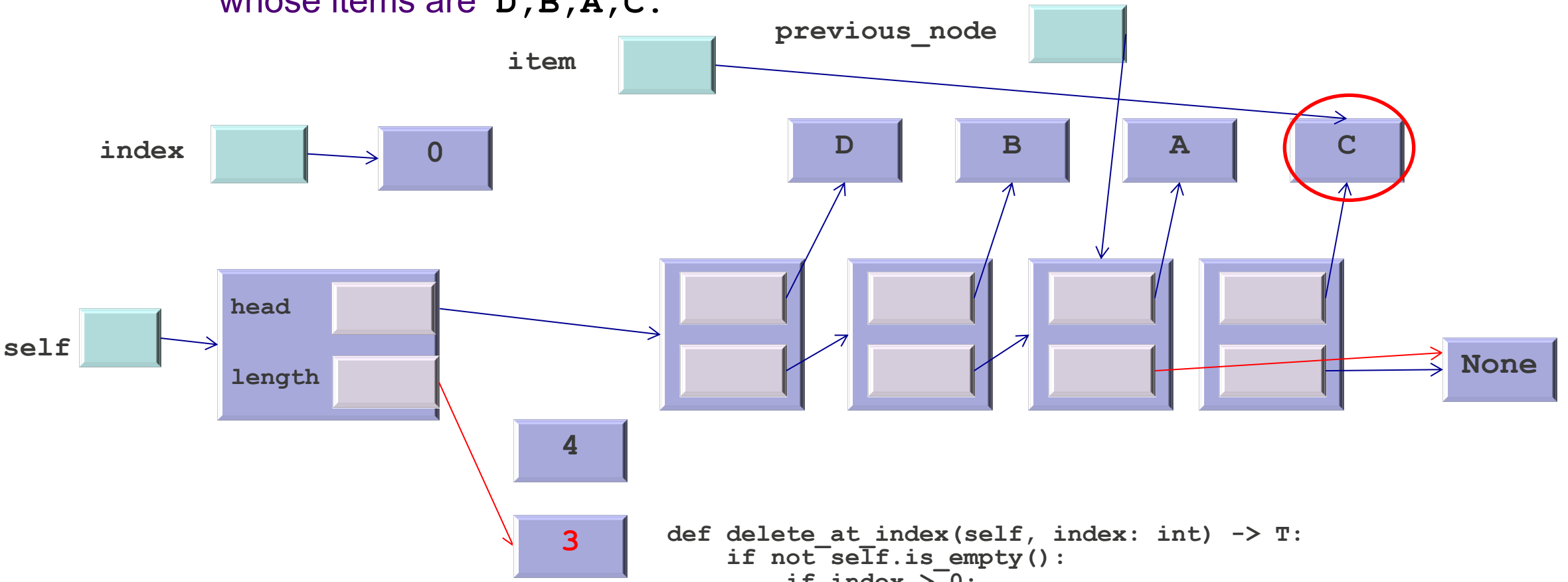
Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Consider a list with four nodes whose items are `D,B,A,C.`

Let's delete at index 0




```
def delete_at_index(self, index: int) -> T:
    if not self.is_empty():
        if index > 0:
            previous_node = self.__get_node_at_index(index-1)
            item = previous_node.link.item
            previous_node.link = previous_node.link.link
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise ValueError("Index out of bounds")
        self.length -= 1
        return item
    else:
        raise ValueError("Index out of bounds: list is empty")
```

Alternative implementation of delete at index

```
def delete_at_index(self, index: int) -> T:
    try:
        previous_node = self.__get_node_at_index(index-1)
    except ValueError as e:
        if self.is_empty():
            raise ValueError("List is empty")
        elif index == 0:
            item = self.head.item
            self.head = self.head.link
        else:
            raise e
    else:
        item = previous_node.link.item
        previous_node.link = previous_node.link.link
    self.length -= 1
    return item
```

It is more efficient because the most common case (index > 0, list not empty) has less number of conditions tested before executing

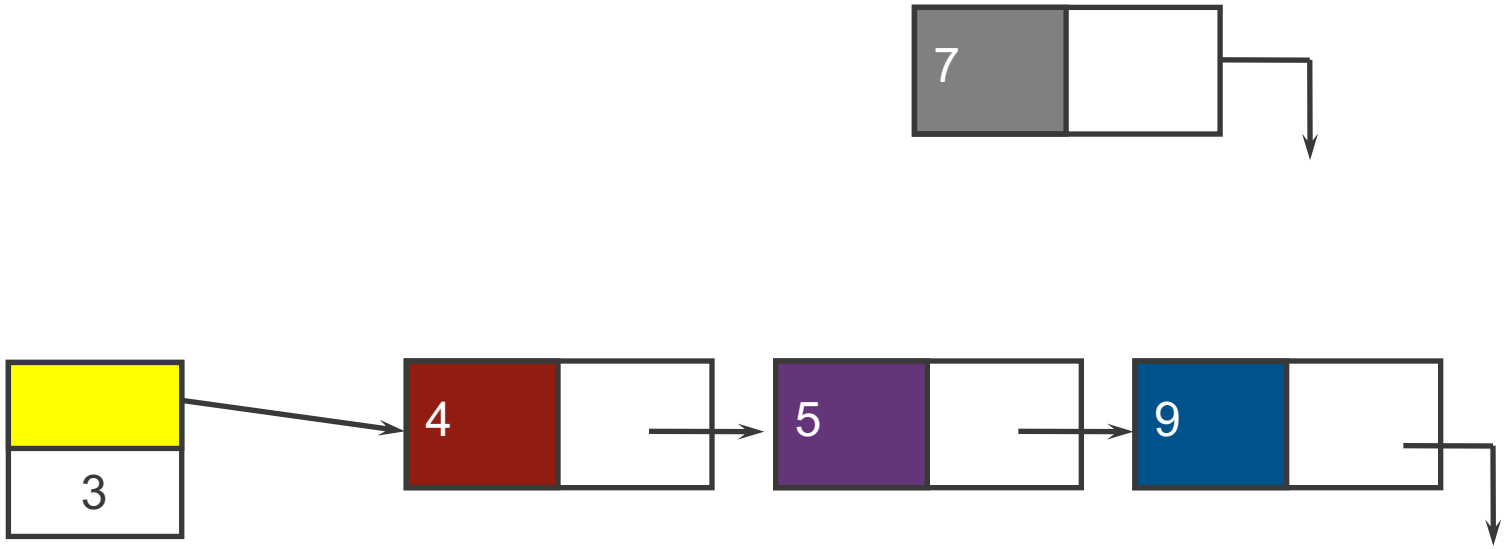
Perhaps less clear…

MONASH
University

Implementing insert

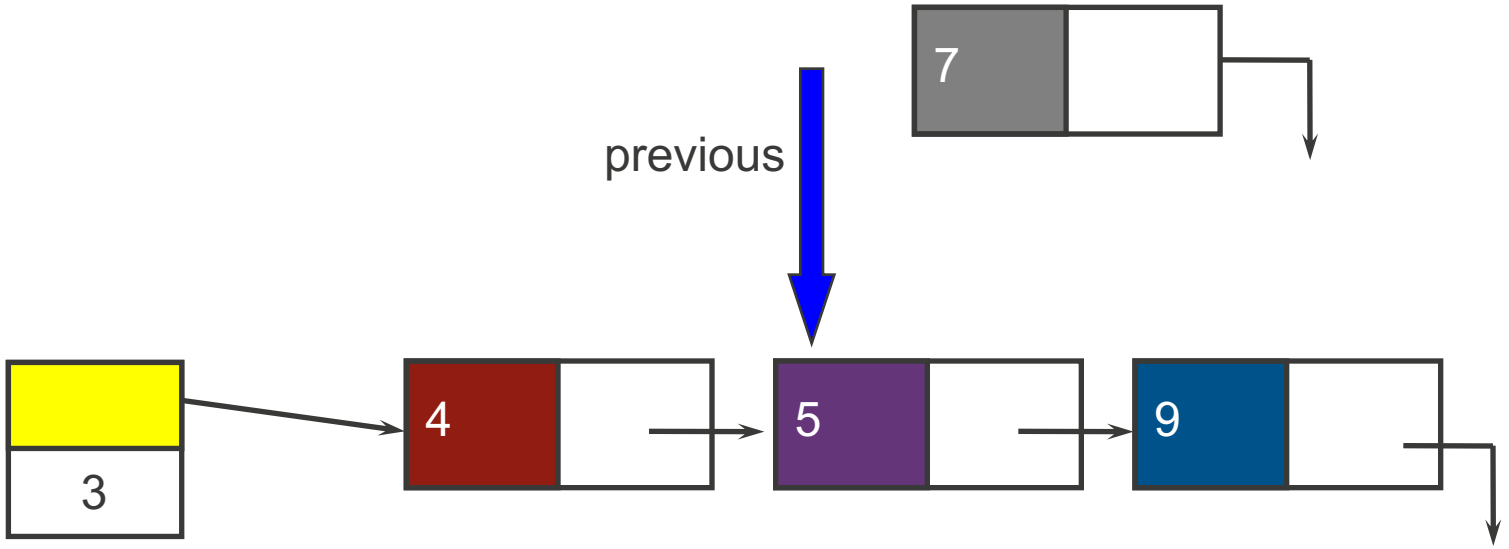
How would we use it for inserting item 7 at index 2?

- **Create a new node with the item**



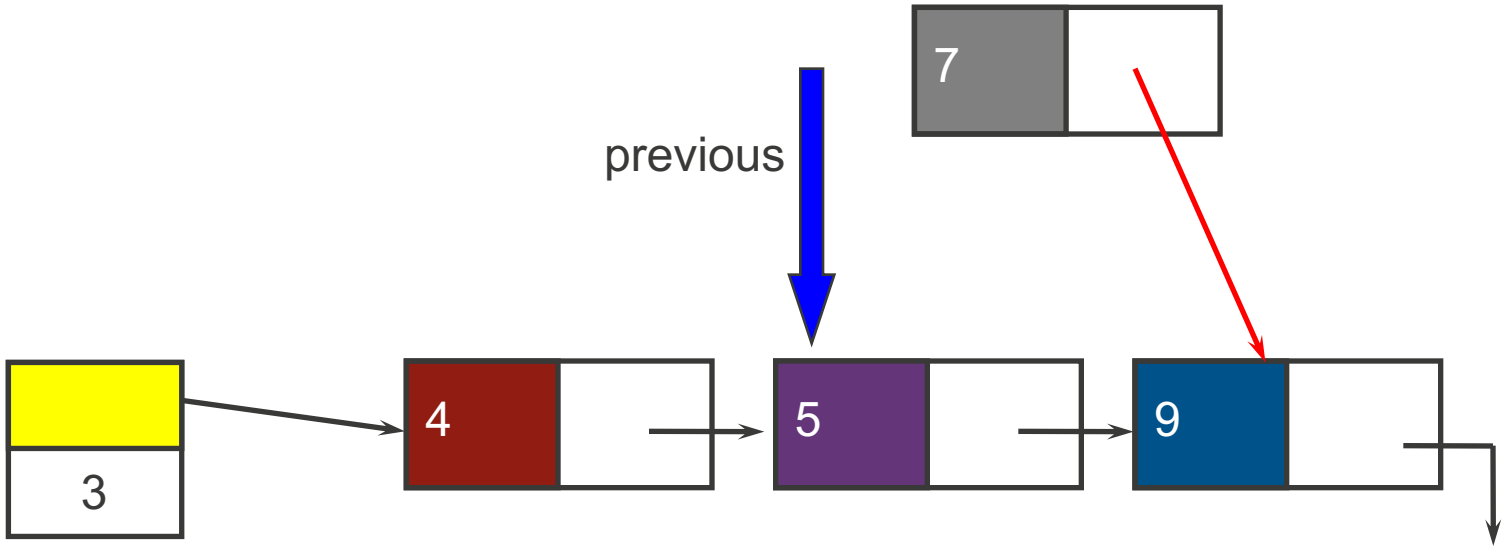
How would we use it for inserting item 7 at index 2?

- Create a new node with the item
- Find the node previous to the index



How would we use it for inserting item 7 at index 2?

- Create a new node with the item
- Find the node previous to the index
- Set the link of the new node to the link of previous

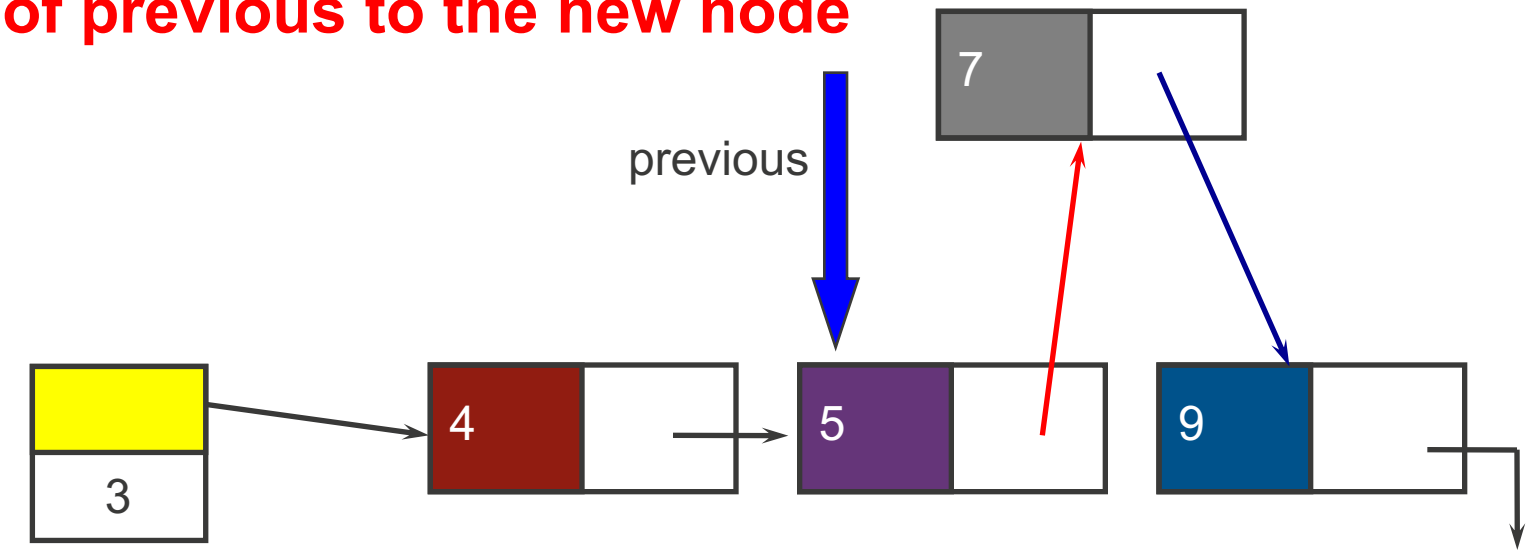


How would we use it for inserting item 7 at index 2?

- Create a new node with the item
- Find the node previous to the index
- Set the link of the new node to the link of previous
- Set the link of previous to the new node

If the index is 0?

Change the head



A possible implementation of insert at index

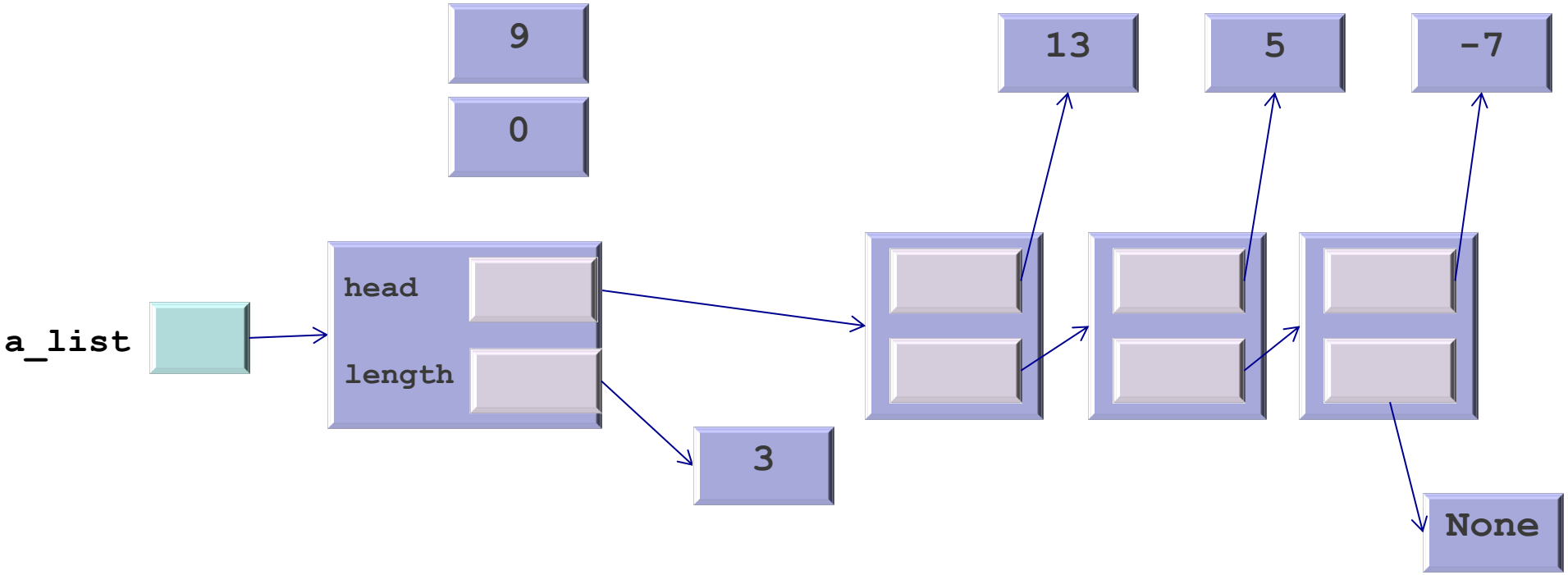
```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    self.length += 1
```

- **Complexity?**
  - Everything is constant except **`__get_node_at_index`**, which is O(index)
  - Thus, it is O(index)

Alternative implementation of insert at index

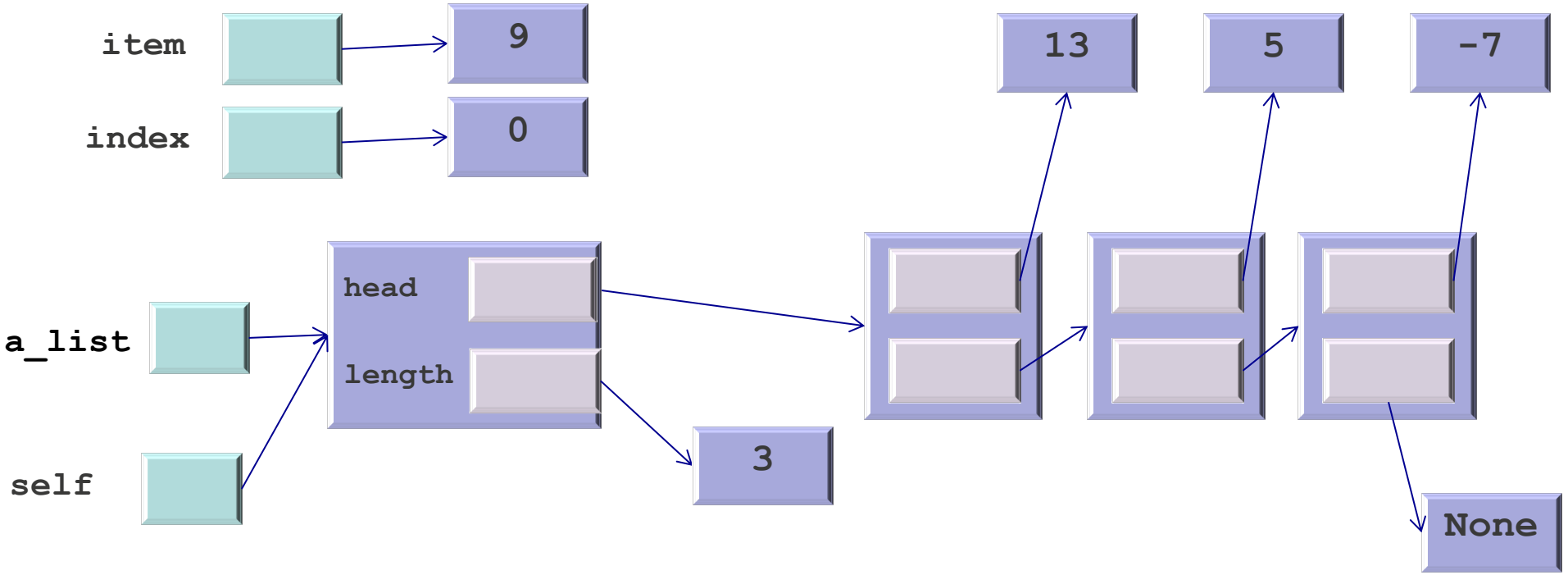
```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```

- **Is it better?**
  - It is more efficient (index will often be > 0, which means less jumps in assembler)
    - Think about your MIPS
  - It is longer and a bit more complex, so perhaps less maintainable
    - If your application is not efficiency critical, might be better to use the first version



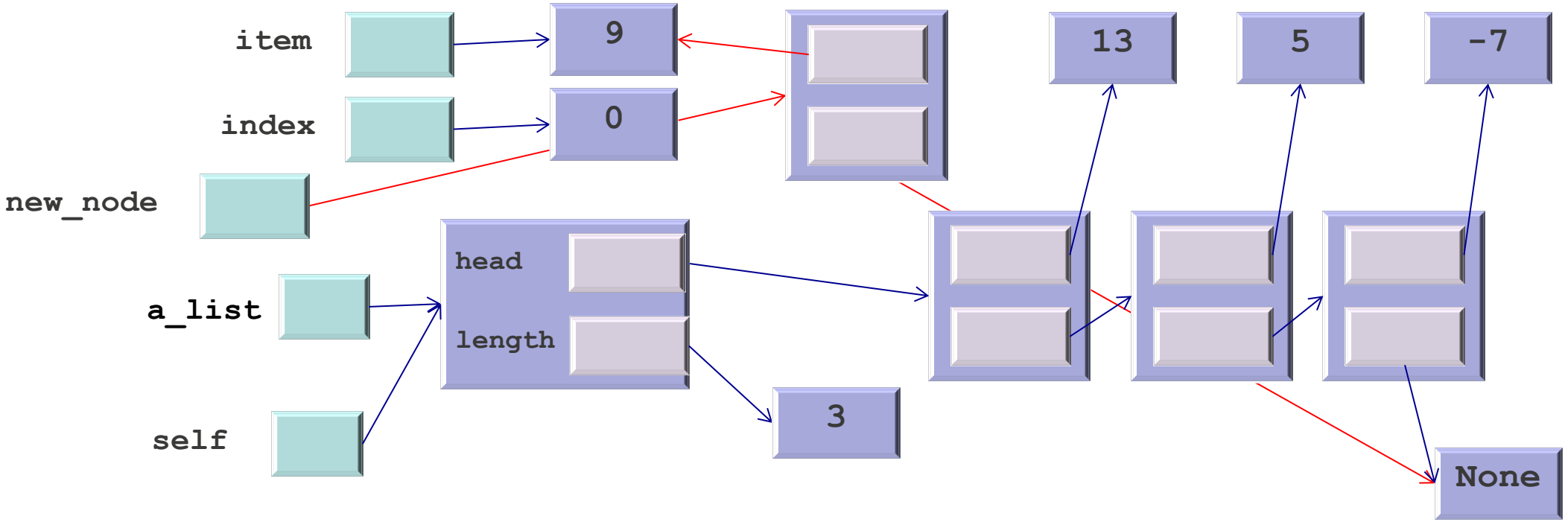
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



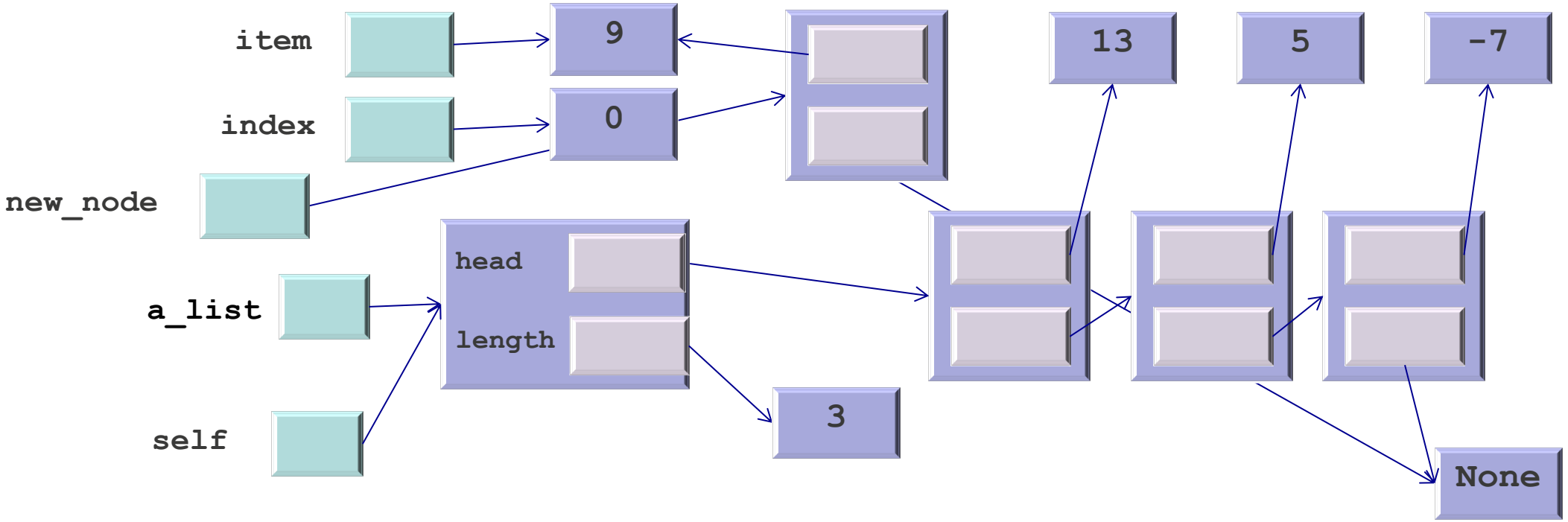
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



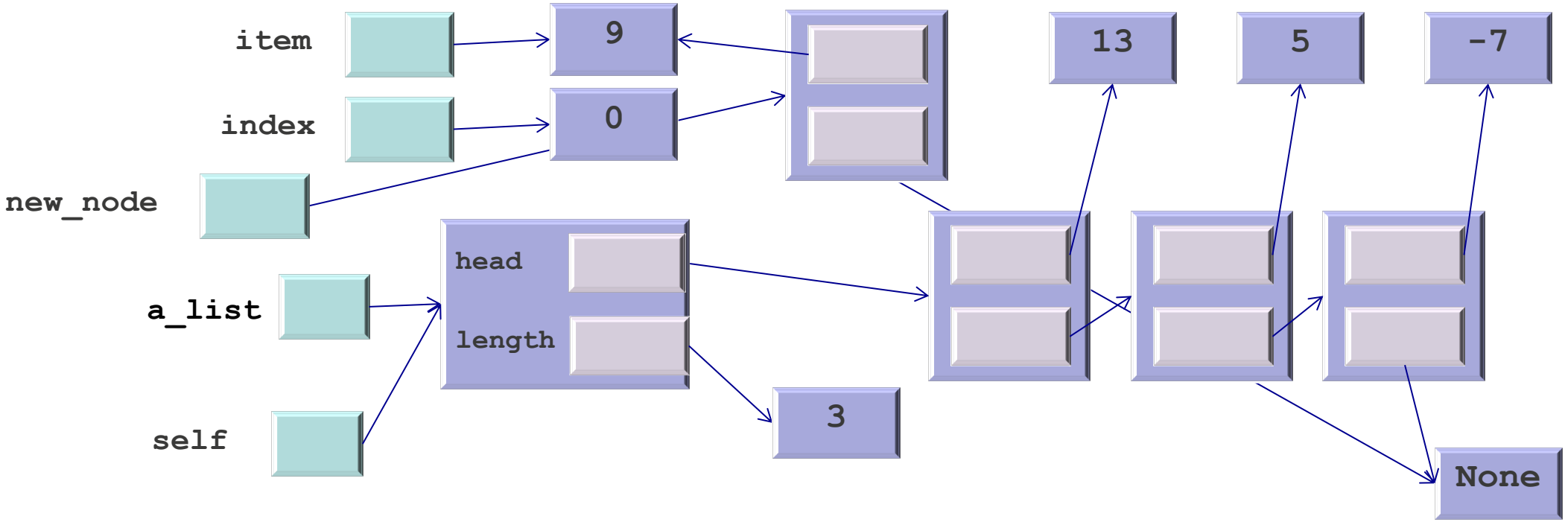
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



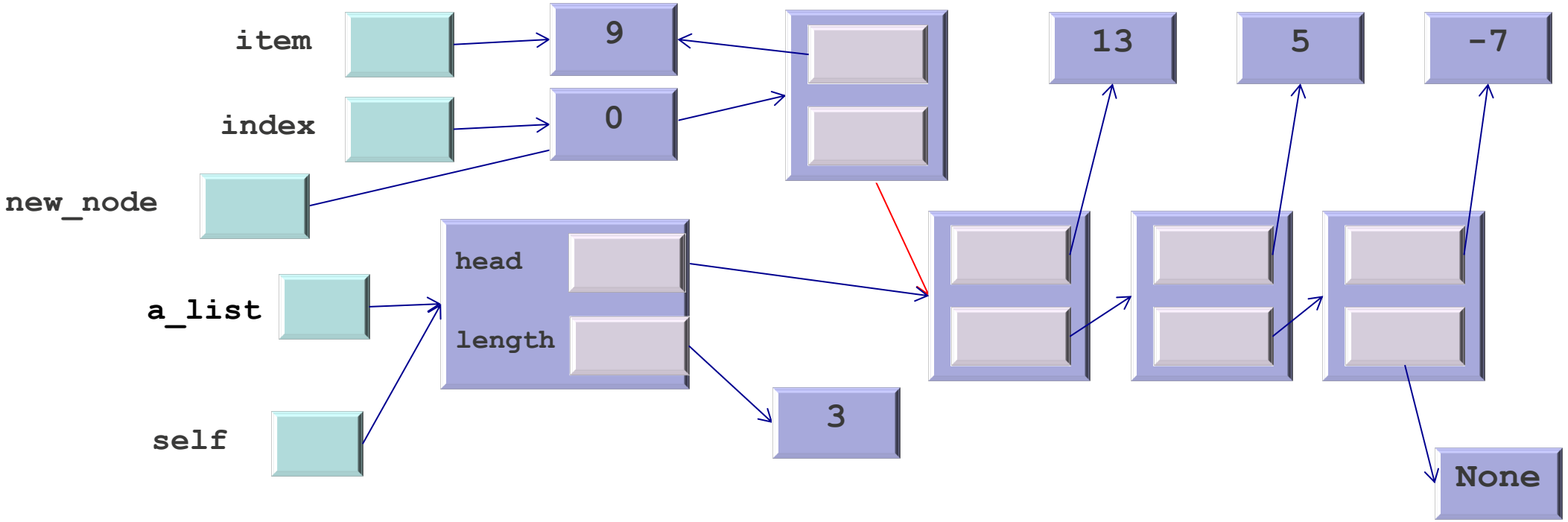
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



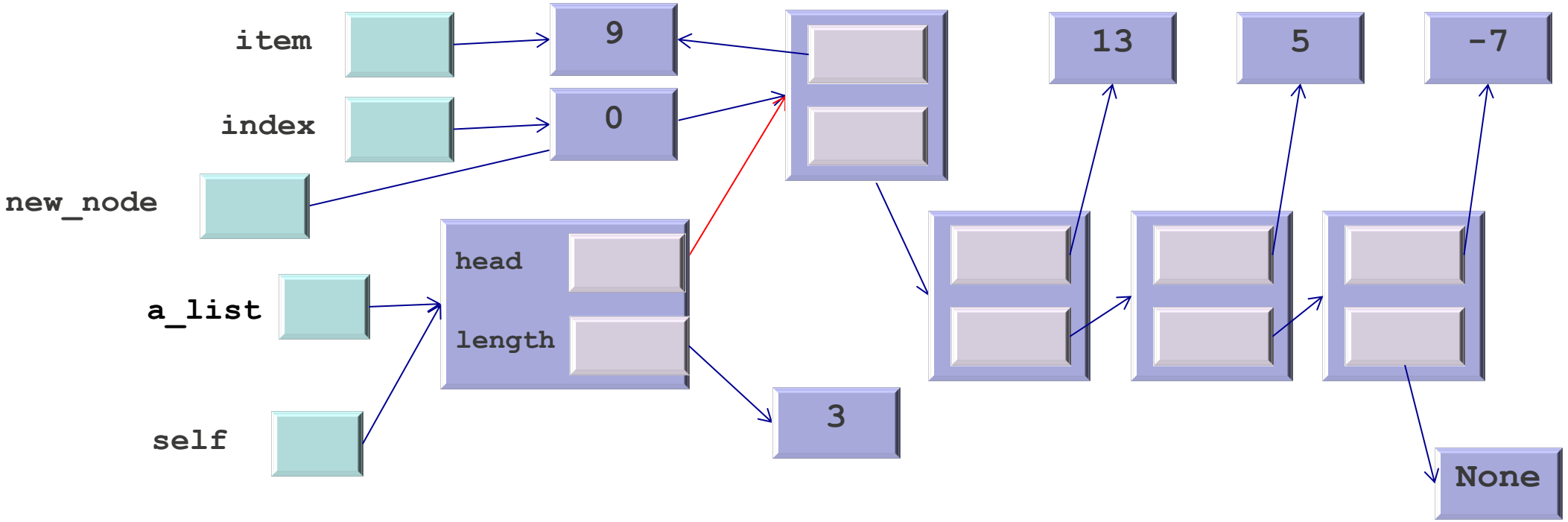
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



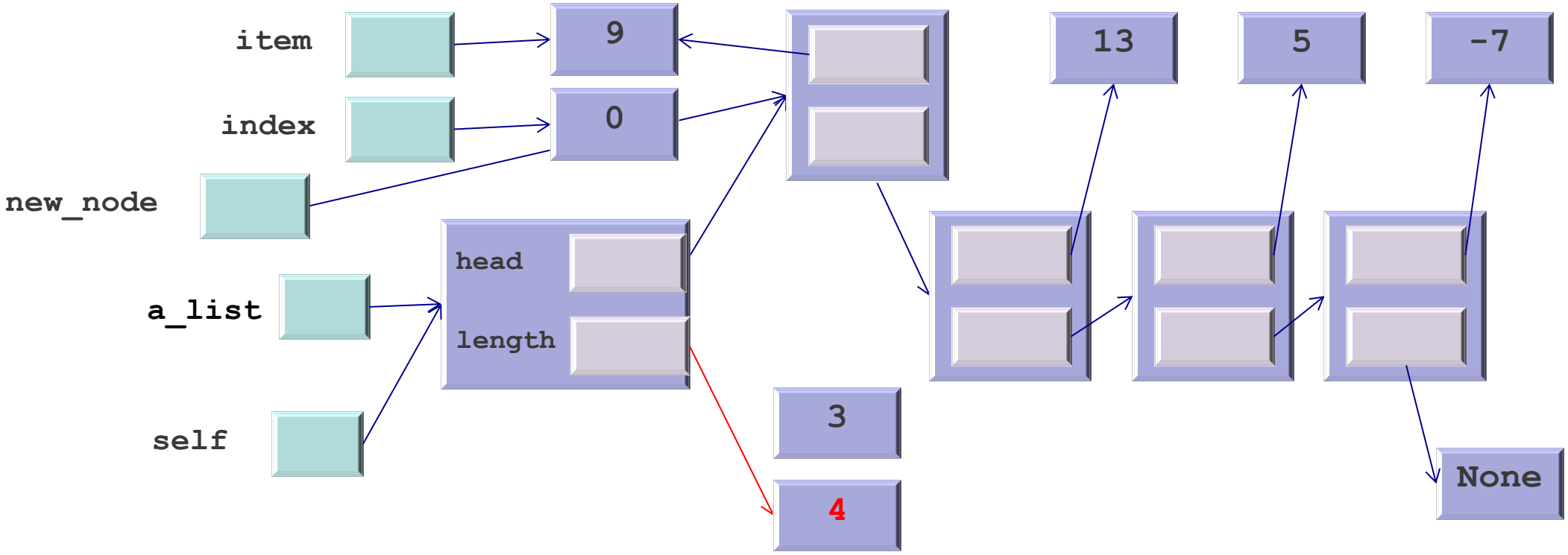
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



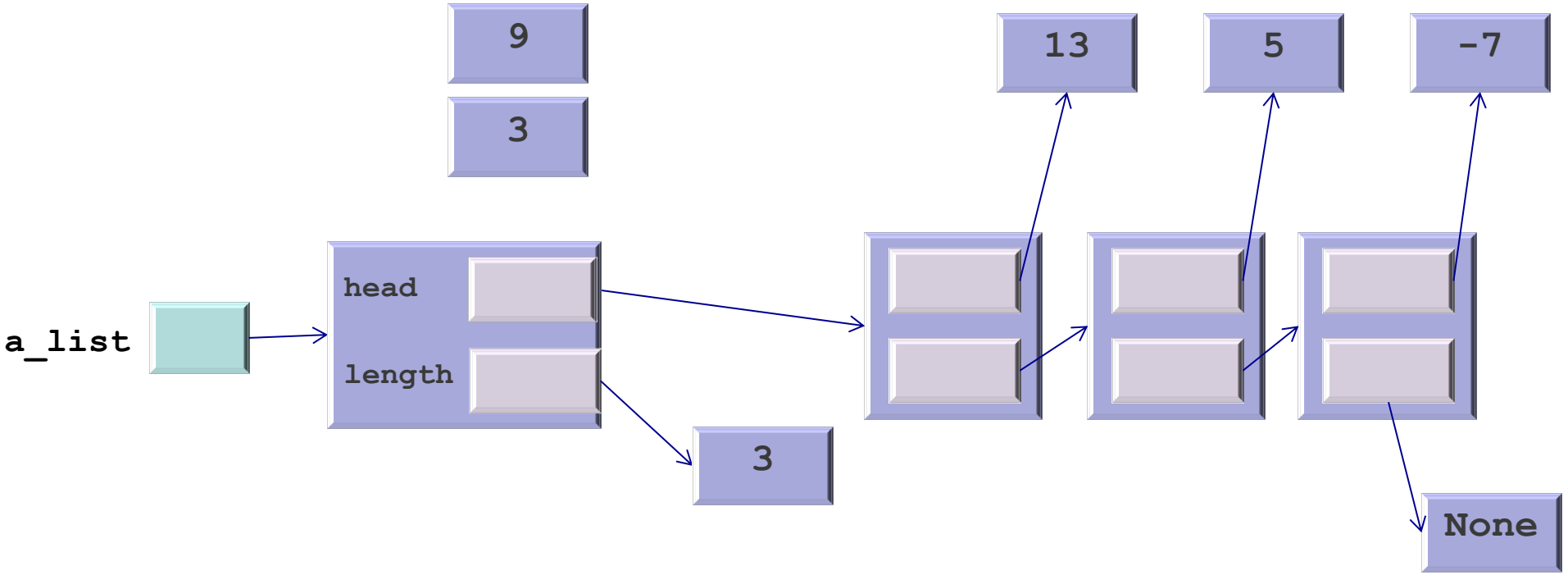
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



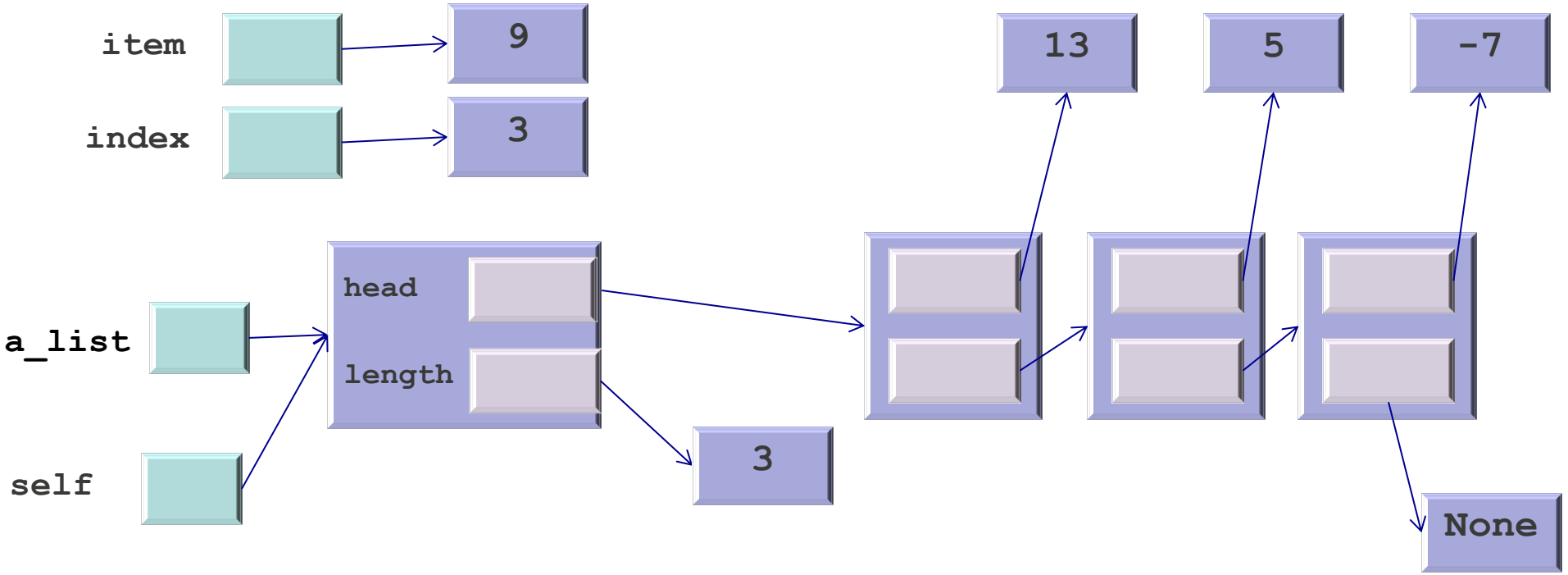
a_list.insert(0, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



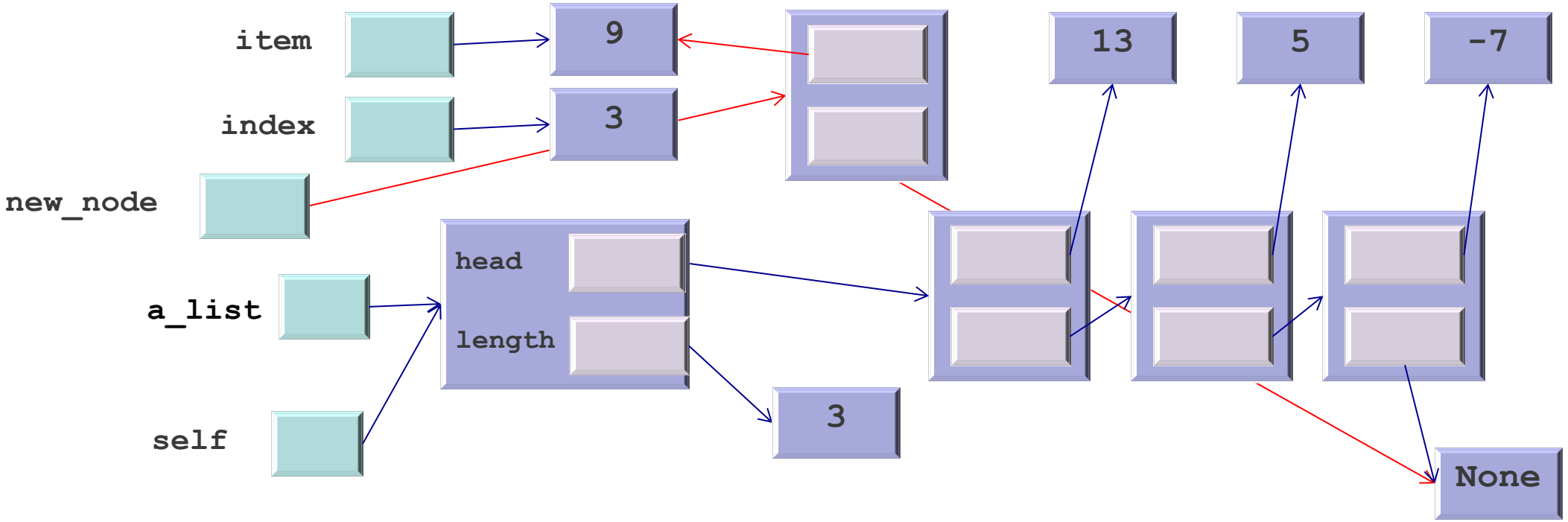
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



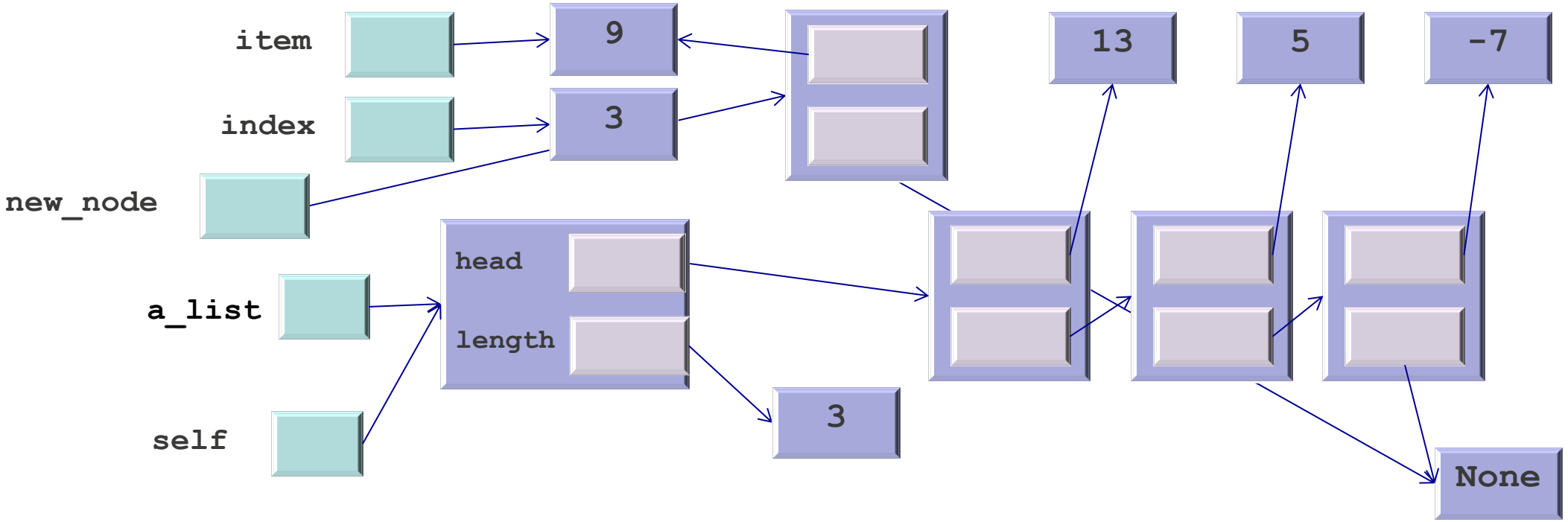
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



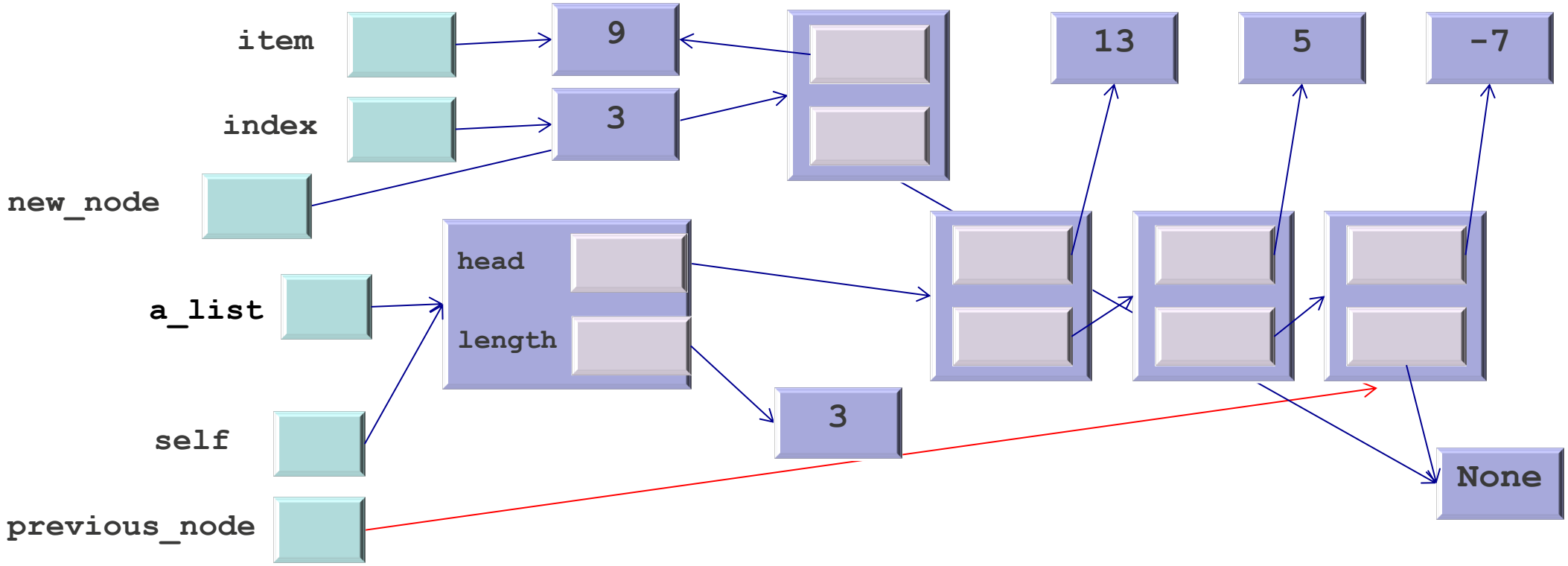
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



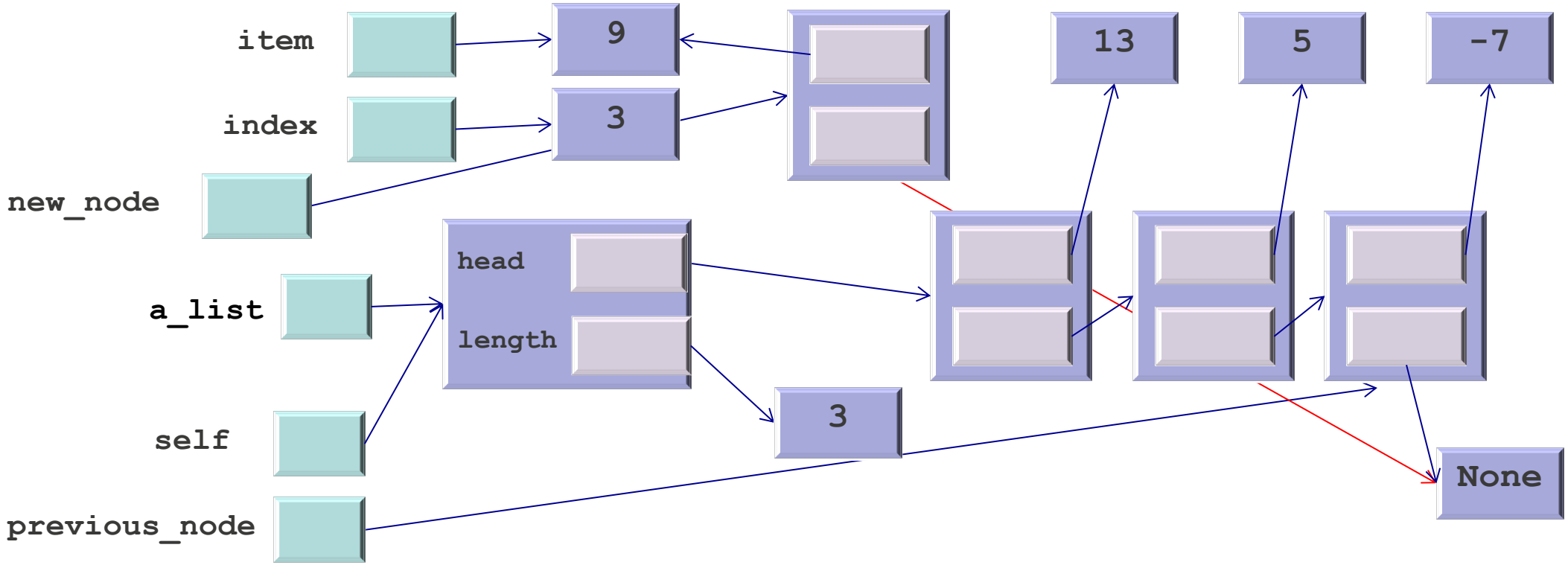
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



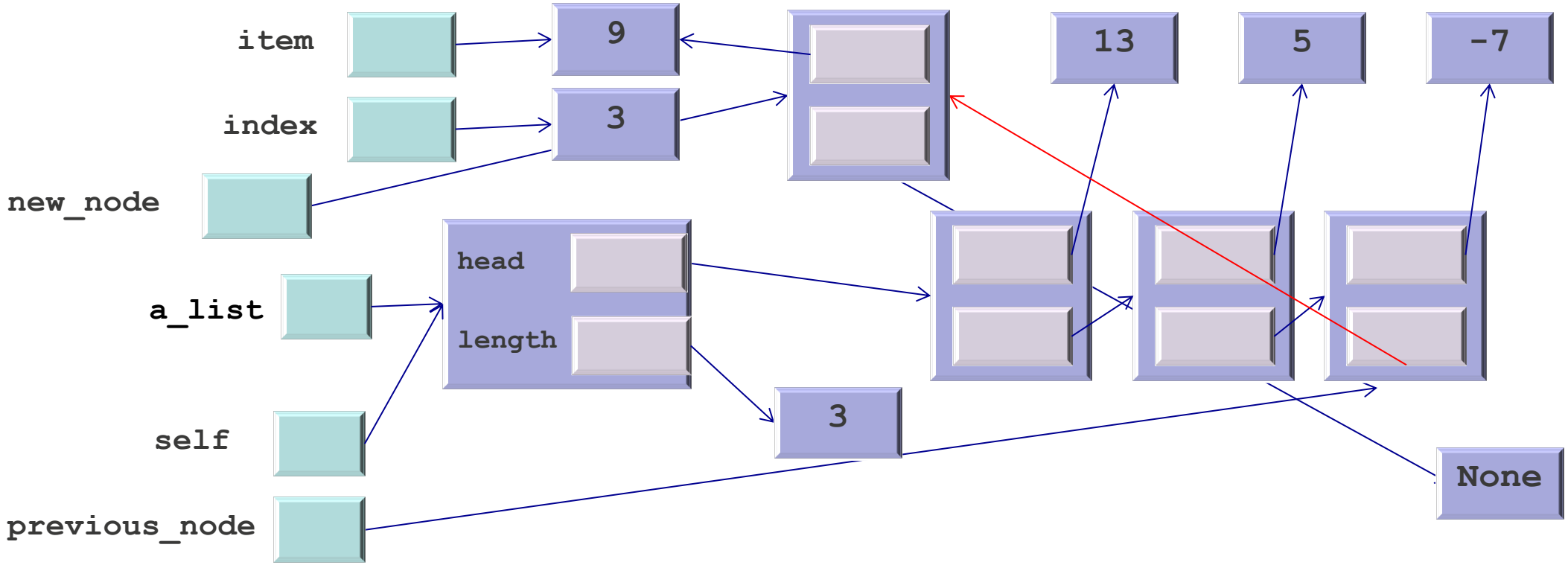
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



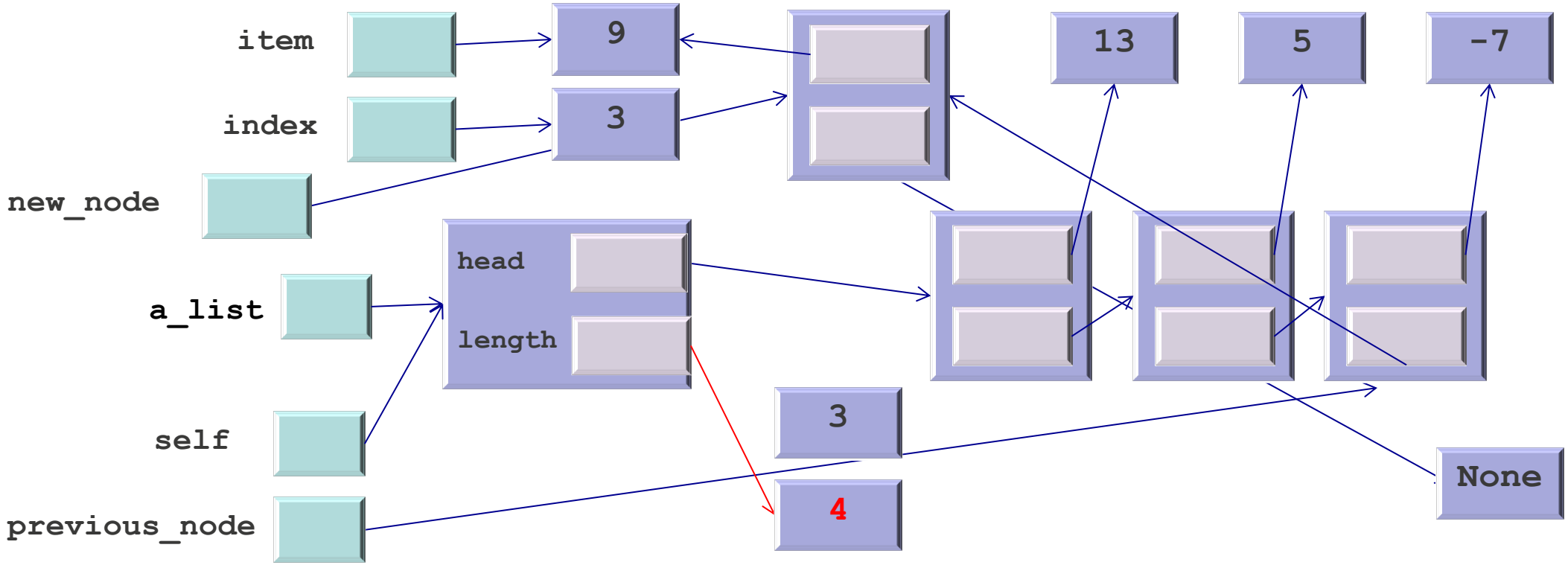
a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```



a_list.insert(3, 9)

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    if index > 0:
        previous_node = self.__get_node_at_index(index-1)
        new_node.link = previous_node.link
        previous_node.link = new_node
    elif index == 0:
        new_node.link = self.head
        self.head = new_node
    else:
        raise ValueError("Index out of bounds")
    self.length += 1
```

Yet another implementation of insert

```
def insert(self, index: int, item: T) -> None:
    new_node = Node(item)
    try:
        previous_node = self.__get_node_at_index(index-1)
    except ValueError as e:
        if index == 0:
            new_node.link = self.head
            self.head = new_node
        else:
            raise e
    else:
        new_node.link = previous_node.link
        previous_node.link = new_node
    self.length += 1
```

Again, it is more efficient because the most common case (index > 0) has less number of conditions tested before executing

Again, perhaps less clear…

MONASH University

Implementing index

And how do we find the index of an item?

- **We cannot use `__get_node_at_index` because we don't know the index**
- **The algorithm is quite easy (a mix of what we have seen):**
  - A linear search until we find the item or we reach the end of the list
  - That keeps an index to counts the number of times we move via links
- **It starts with the index at 0 and a pointer to the head node**

A possible implementation of index

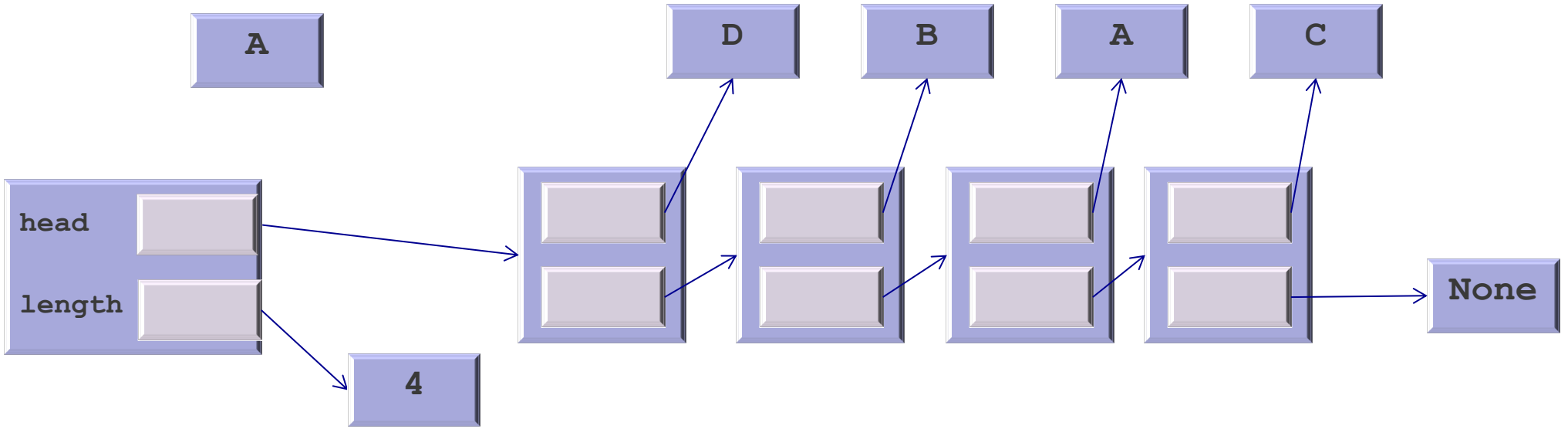
```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

? times

- **Complexity?**
  - Everything is constant except the comparison
  - The loop runs until it finds the item
  - Best case: item is in the head node $O(Comp_{==})$
  - Worst case: item is in the last node $O(Comp_{==} * len(self))$

Consider a list with four nodes whose items are `D,B,A,C.`

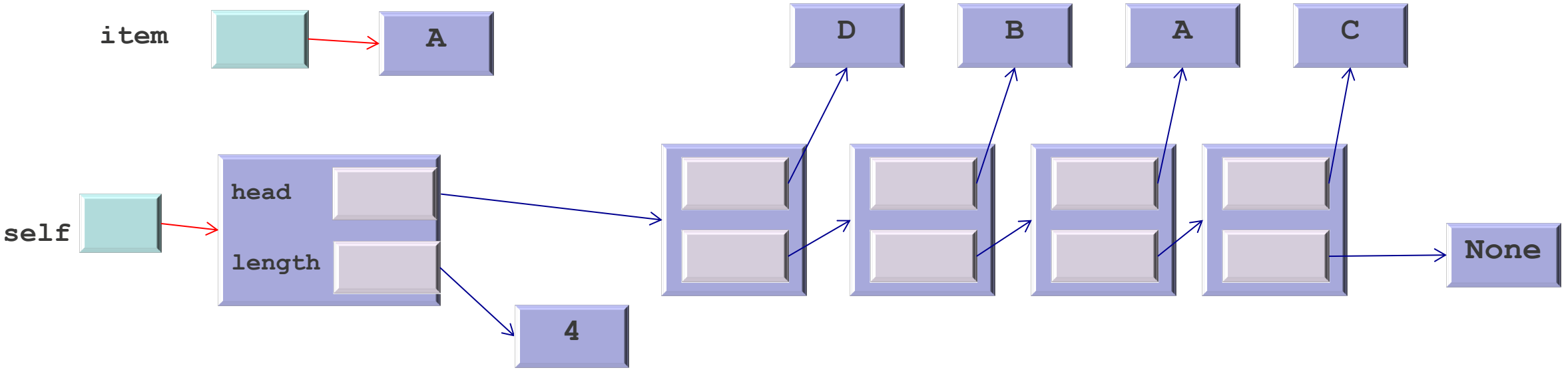
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

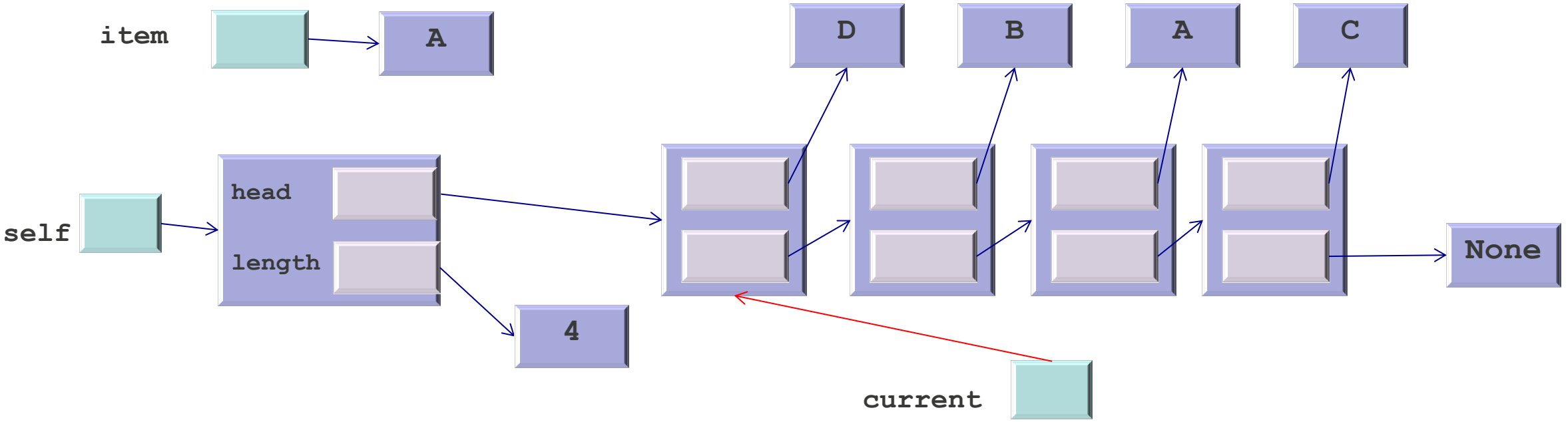
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

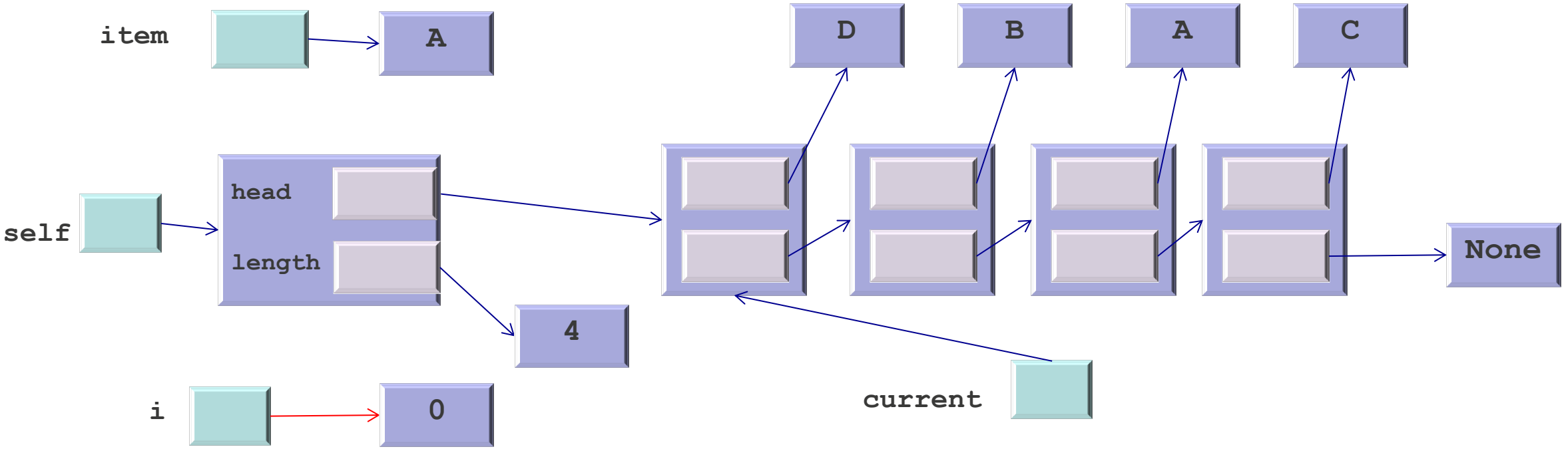
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

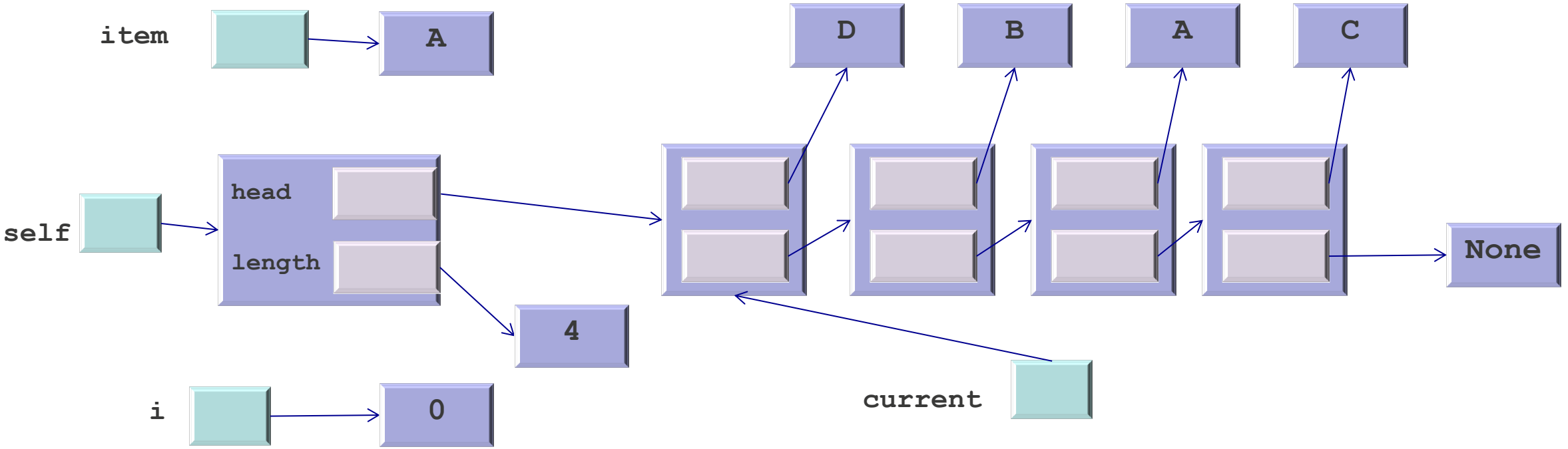
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

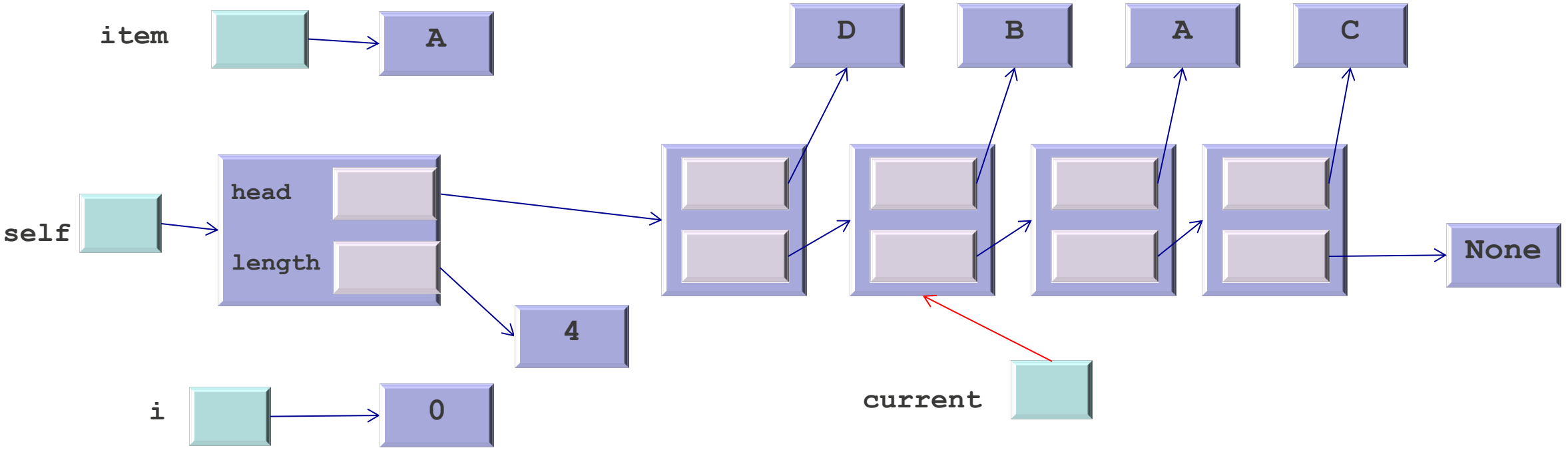
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

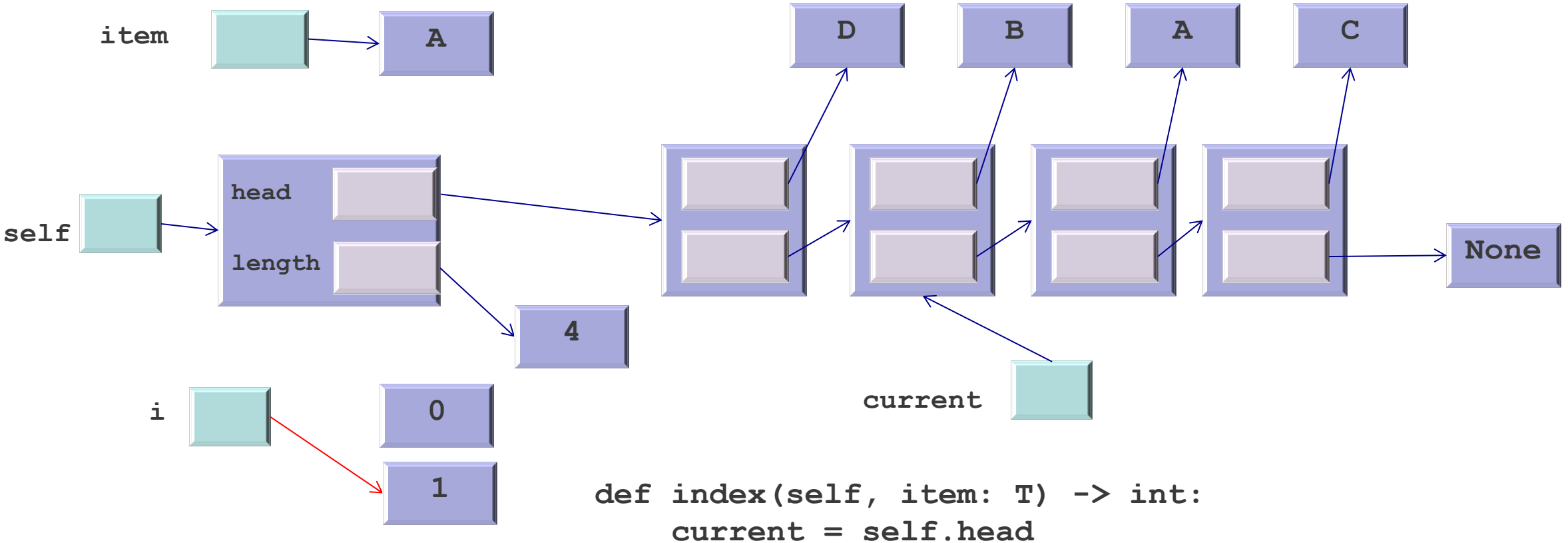
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

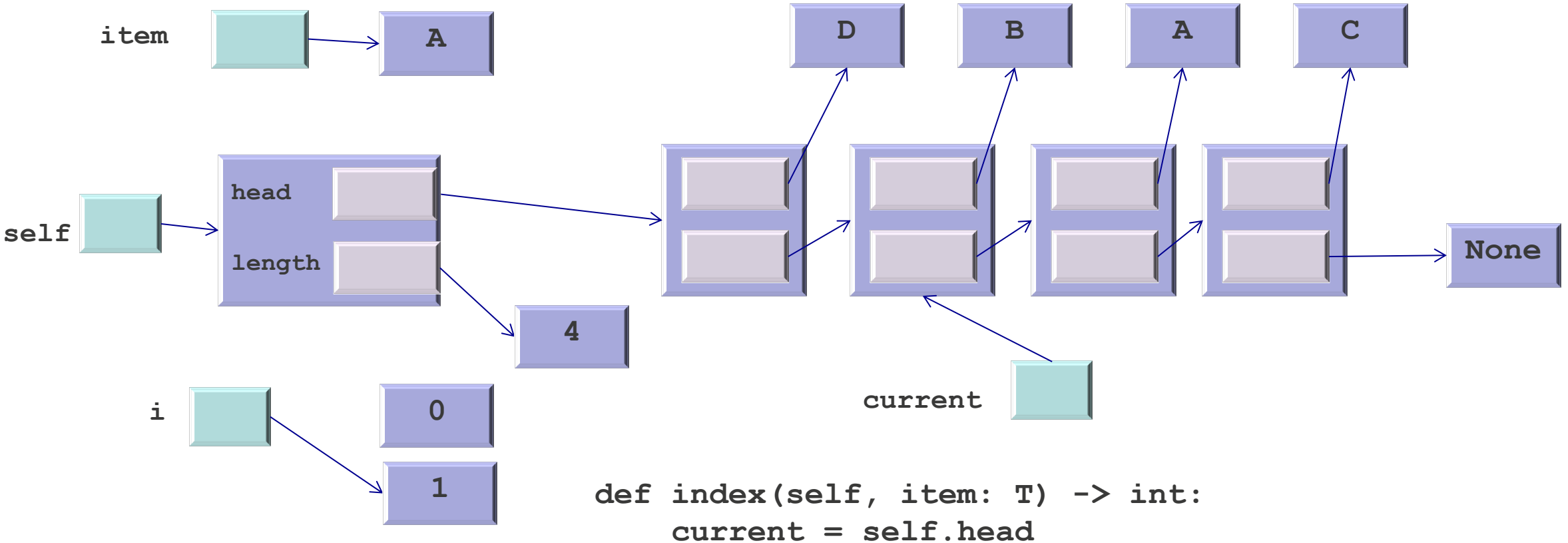
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

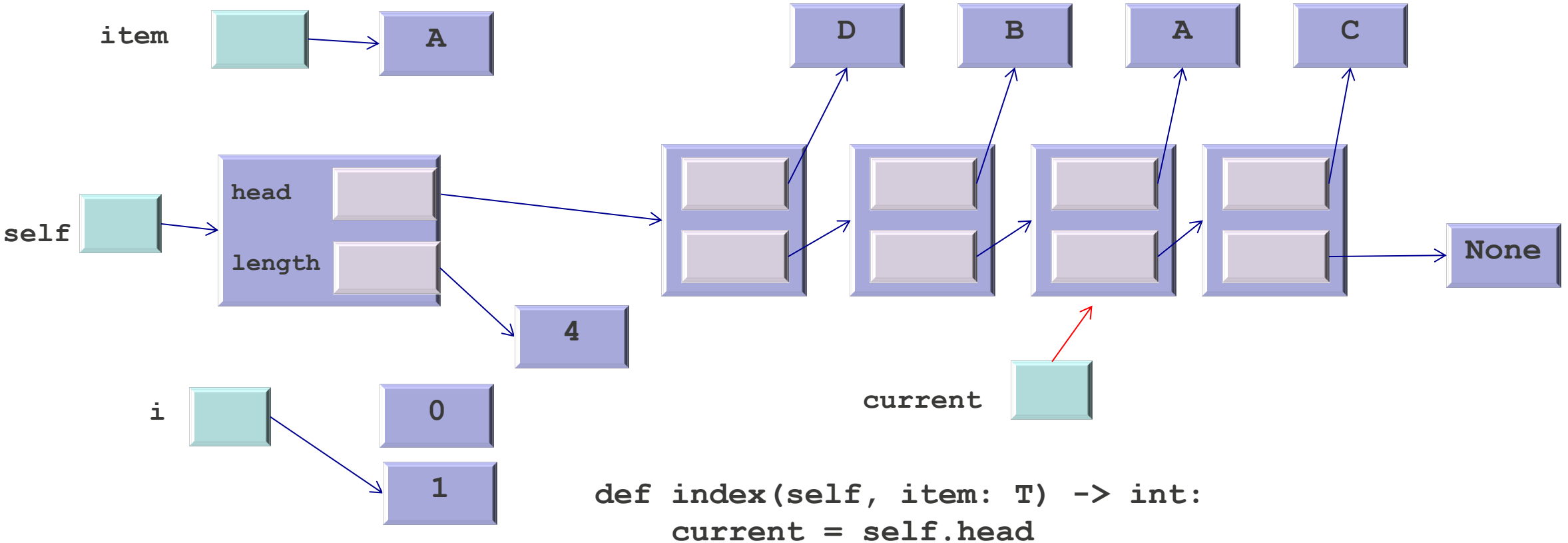
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

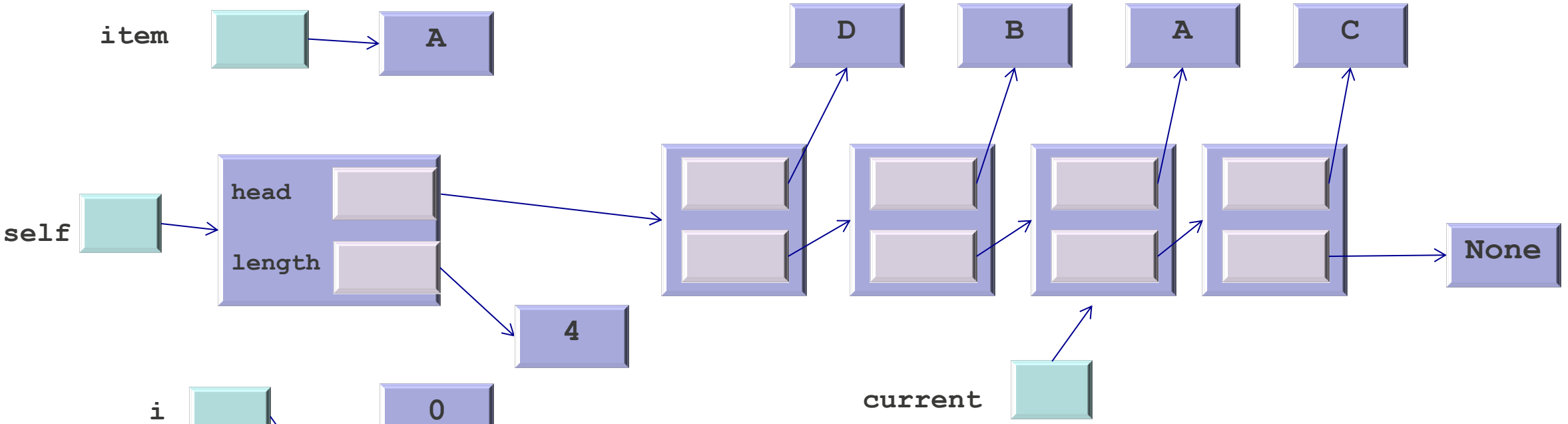
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

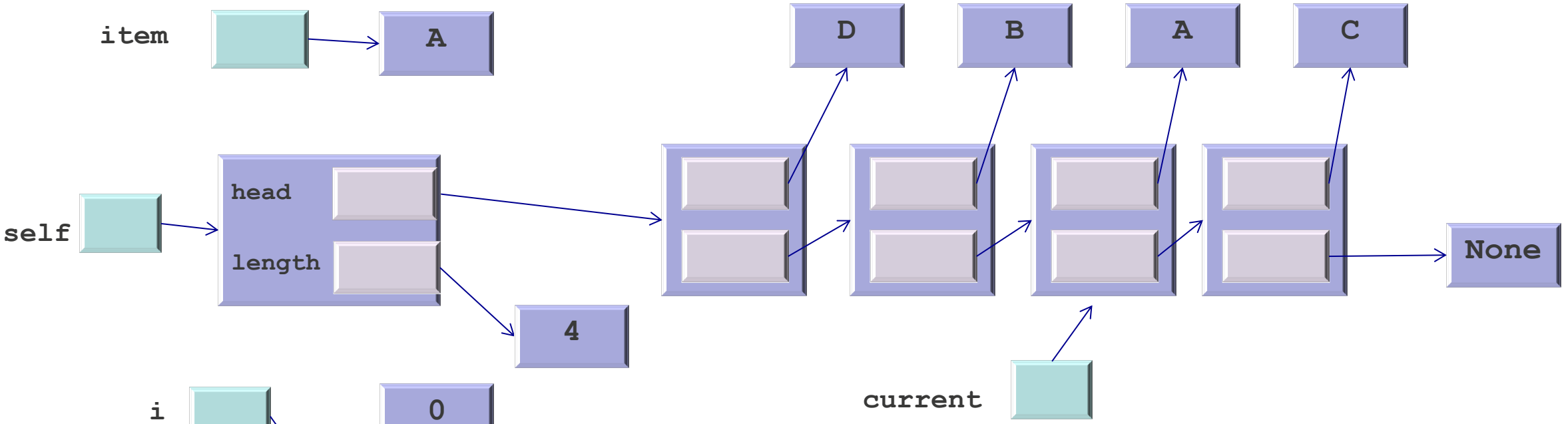
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

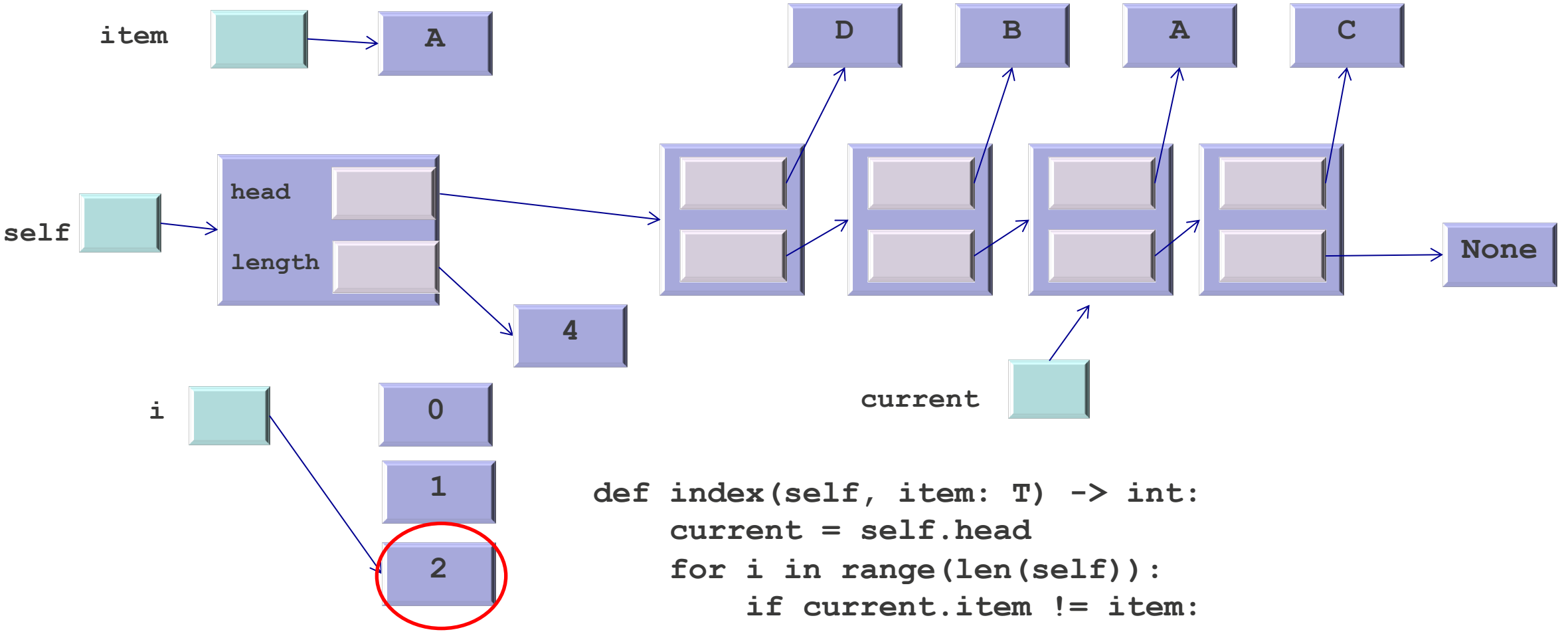
Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

Consider a list with four nodes whose items are `D,B,A,C.`

Let's search for A




```
def index(self, item: T) -> int:
    current = self.head
    for i in range(len(self)):
        if current.item != item:
            current = current.link
        else:
            return i
    raise ValueError("Item is not in list")
```

What about implementing Linked Sorted Lists?

- **Implementing index is very easy: modify to stop if `item < current.item`**
- **But there is no possibility of using binary search**
  - We don't have random access (i.e, constant-time access to any element)
  - And with linked nodes, we can only go in one direction
- **This reduces the usefulness of sorted linked lists**

Summary

- **We now understand the concept of Data Structures based on linked nodes**
- **We know how to use them in implementing the List ADT**
- **In particular, we are able to:**
  - Implement, use and modify linked lists as a derived class of our List[T] class
  - Decide when is it appropriate to use them (as opposed to using the ones implemented with arrays)