학교 공부/알고리즘

[알고리즘](Python) 리스트 자료구조

그리버 | 2023. 9. 11. 13:51 | 수정 | 삭제

참고 자료: 파이썬으로 배우는 자료 구조 핵심 원리(양태환), 이것이 자료구조 + 알고리즘이다 (박상현 저)

- 주요 개념

- 리스트의 개념
- 리스트와 배열의 차이점
- 연결 리스트
- 이중 연결 리스트
- 원형 링크드 리스트

4.1 연결 리스트 이해하기

리스트(List)

리스트는 목록 형태로 이뤄진 데이터 형식이다.

노드(Node)

이 목록을 이루는 개별 요소를 노드(Node)라고 부른다.
리스트의 길이는 노드의 개수와 같다.

특별히 리스트의 첫 노드를 헤드(Head), 마지막 노드를 테일(Tail)이라고 한다.

리스트의 연산

- Append
  : 리스트에 노드를 추가
- Insert
  : 노드 사이에 노드를 삽입
- Remove
  : 노드를 제거
- GetAt
  : 특정 위치에 있는 노드를 반환

리스트 vs. 배열

생성하는 시점에 크기를 정해줘야 하는 배열과 달리, 리스트는 유연하게 크기를 바꿀 수 있다.
데이터(노드)의 삽입과 삭제 시 배열의 성능은 O(n), 연결 리스트의 성능은 O(1)이다.
반대로 탐색 시 배열의 성능은 O(1), 연결 리스트의 성능은 O(n)이다.

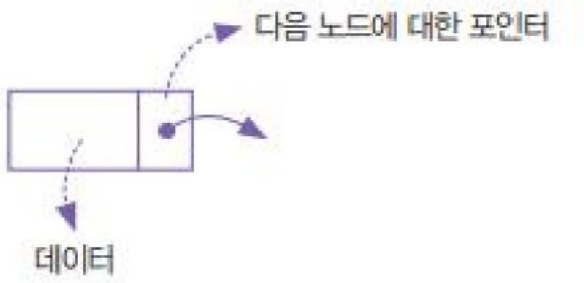
따라서 연결 리스트는 삽입과 삭제가 빈번하게 필요한 경우에 쓰면 좋다.

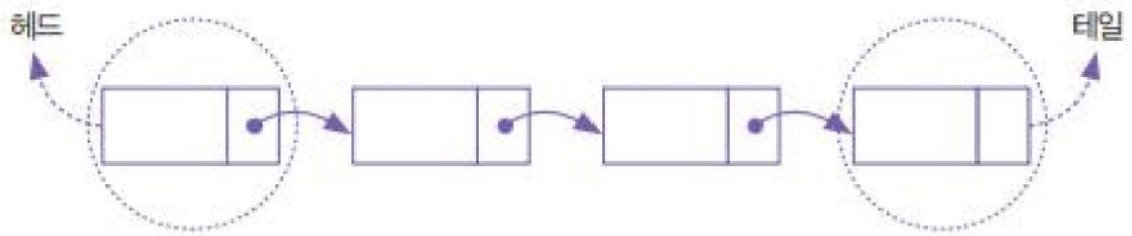
리스트의 다양한 구현 방법

1. 연결 리스트
2. 이중 연결 리스트
3. 원형 연결 리스트

연결 리스트

연결 리스트는 리스트의 구현 방법 중 가장 간단한 방법이다.
노드들을 연결한 리스트를 연결 리스트라고 한다. 각 노드들은 포인터(참조)로 서로 이어진다.
각각의 노드는 데이터가 들어가는 부분과 다음 노드를 가리키는 포인터로 이뤄진다.

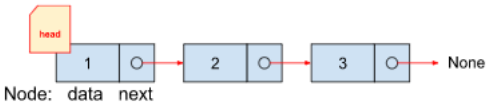




연결 리스트가 갖고 있는 참조가 한 개면 단순 연결 리스트, 두 개면 이중 연결 리스트이다.
단순 연결 리스트는 다음 노드를 가리키는 참조 하나만 가지지만, 이중 연결 리스트는 앞 노드와 다음 노드를 가리키는 참조를 모두 갖는다.

단순 연결 리스트

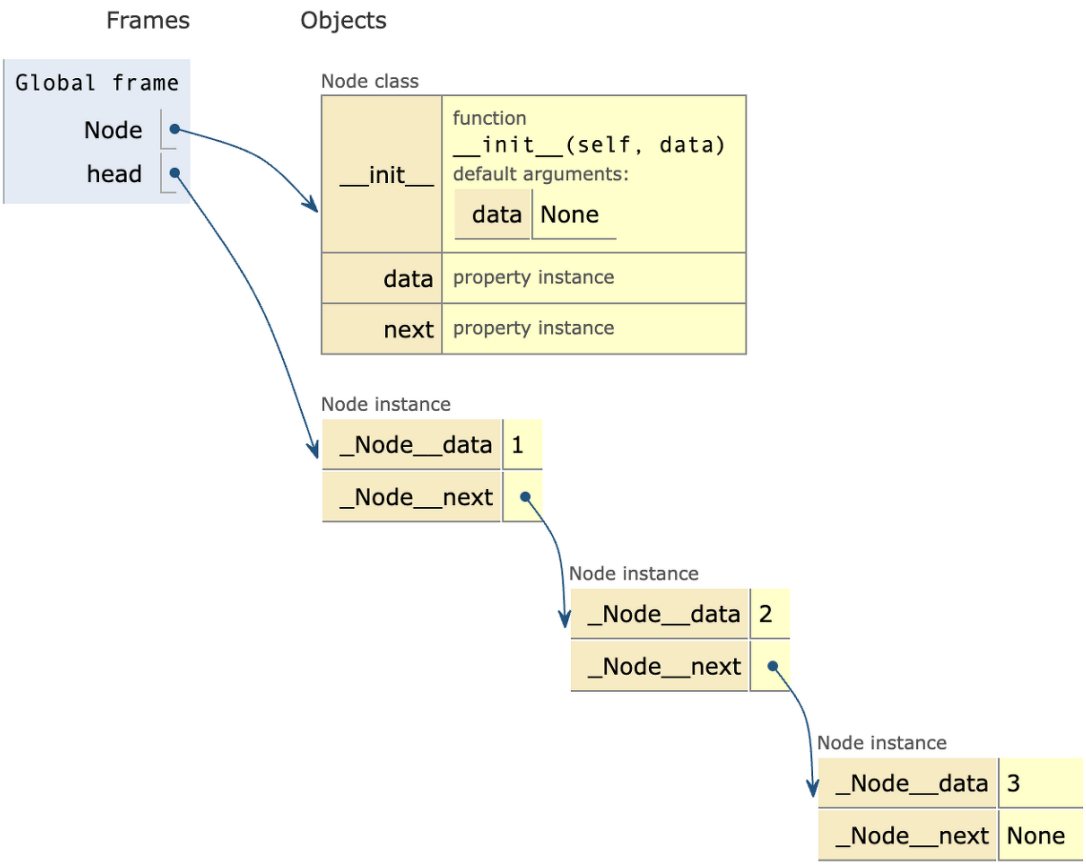
노드 표현



Python 문법으로 연결 리스트를 어떻게 구현할 수 있을까?
노드 클래스를 정의하고 data와 next 변수를 클래스 내에 만들어주면 된다.

```
class Node:
  def __init__(self,data=None):
    self.__data = data  # 값
    self.__next = None  # 다음 노드

  @property
  def data(self):
    return self.__data

  @data.setter
  def data(self, data):
    self.__data = data

  @property
  def next(self):
    return self.__next

  @next.setter
  def next(self, next):
    self.__next = next

head = Node(1) # 첫째 노드를 만들고 head에 첫째 노드를 할당한다.
head.next = Node(2) # 둘째 노드를 만들고 head의 next가 둘째 노드를 가리키도록 한다.
head.next.next = Node(3) # head의 next의 next가 셋째 노드를 가리키도록 한다.
```

위의 코드를 시각화 해보면 아래와 같이 연결되어 있는 것을 확인할 수 있다.



단순 연결 리스트는 주로 삽입, 삭제, 탐색 등의 연산과 함께 사용된다.

단순 연결 리스트의 연산

(1) 삽입

위와 같이 형성되어 있는 단순 연결 리스트에 새로운 노드를 삽입하는 연산이다.

```
(head) | 1 |   | --> | 2 |   | --> | 3 |   | --> | None |
```

이런 리스트에서 새로운 노드 | 4 | | 를 노드 1과 2 사이에 어떻게 삽입할 수 있을까?
참조하는 노드를 바꿔주면 된다.

새로운 노드의 next는 삽입하는 바로 다음 노드인 2를 가리키게 하고,
삽입하는 바로 전 노드인 1의 next는은 새로운 노드인 4를 가리키게 하면 된다.
그렇게 하면 결과적으로 아래와 같은 리스트가 만들어진다.

```
(head) | 1 |   | --> | 4 |   | --> | 2 |   | --> | 3 |   | --> | None |
```

(2) 삭제

삽입을 마친 위와 같은 리스트에서 다시 삽입한 노드를 삭제하려면 어떻게 하면 될까?
삽입 보다 더욱 쉽다.

```
(head) | 1 |   | --> | 4 |   | --> | 2 |   | --> | 3 |   | --> | None |
```

위의 리스트에서 노드 4를 삭제하려면,
노드 1의 next가 노드 2를 가리키게 하면 된다.

(C언어와 같은 경우에는 리스트에서 떨어진 노드 4를 프로그래머가 메모리에서 삭제해주어야 하지만 Python은 언어 차원에서 삭제해주기 때문에 따로 신경 쓸 필요가 없다. 이렇게 언어 차원에서 메모리를 사용자 대신 관리해주는 것을 가비지 컬렉션이라고 한다.
파이썬은 레퍼런스 카운트로 가비지 컬렉션을 해준다.이는 어떤 객체를 가리키는 객체의 개수가 0이 되면 해당 객체를 지우는 것이다. 노드 1이 노드 2를 가리키면서, 노드 4를 가리키는 노드가 없어졌기에 노드 4는 가비지 컬렉션으로 삭제된다.)

그렇게 하면 결과적으로 처음과 같은 리스트로 되돌릴 수 있다.

```
(head) | 1 |   | --> | 2 |   | --> | 3 |   | --> | None |
```

(3) 탐색

연결 리스트에서 어떤 노드를 찾기 위해서는 처음부터 끝까지 순회 방문하며 해당 노드를 찾아야 한다.
순회를 위해서는 우선 변수 cur을 둔다. 그리고 리스트의 각 노드를 차례대로 가리키며 찾고자 하는 값과 방문한 노드의 값을 비교하며 같은 값이 나올 때까지 반복한다.

```
        cur # 다음 노드로 이동하면서 비교
         |
(head) | 1 |    | --> | 2 |    | --> | 3 |    | --> | None |
```

연결 리스트와 배열의 성능 차이

4.3 더미 이중 연결 리스트

일반적인 연결 리스트로 사용되는 것이 바로 더미 이중 연결 리스트(dummy double linked list)이다.

```
### 더미 이중 연결 리스트의 ADT ###
DoubleLinkedList
# Operation
    1. empty() -> Boolean
       : 비어 있으면 True, 아니면 False 반환
    2. size() -> Integer
       : 노드 개수 반환
    3. add_first(data)
       : data를 리스트 맨 앞에 추가
    4. add_last(data)
       : data를 리스트 맨 마지막에 추가
    5. insert_after(data, node)
       : data를 node 다음에 삽입
    6. insert_before(data, node)
       : data를 node 이전에 삽입
    7. search_forward(target) -> node
       : target을 리스트의 맨 처음부터 찾으면서 있으면 node 반환, 없으면 None 반환
    8. search_backward(target) -> node
       : target을 리스트의 맨 뒤부터 찾으면서 있으면 node 반환, 없으면 None 반환
    9. delete_first()
       : 리스트의 첫 번째 노드 삭제
   10. delete_last()
       : 리스트의 맨 마지막 노드 삭제
   11. delete_node(node)
       : node 삭제
```

노드 구현

더미 이중 연결 리스트에서 노드를 아래와 같이 구현할 수 있다.
- class Node 멤버

- __data: 데이터 저장
- __prev: 이전 노드 참조
- __next: 다음 노드 참조

클래스의 변수 명 앞에 언더바(_)를 두 개 붙이면 private 변수로 사용할 수 있다.
클래스 밖 외부 코드에서 직접 접근과 수정까지 가능한 public 변수와 다르게 private 변수는 외부에서 접근이 불가능하다. 따라서 외부에서 클래스 멤버에 접근하기 위해서는 별도의 작업이 필요한데 이런 과정을 캡슐화라고 한다. 단순한 캡슐화는 멤버에 접근할 때마다 메쏘드를 호출하는 코드를 작성해야 하기 때문에 코드의 가독성이 좋지 않다. 따라서 프로퍼티를 이용한 캡슐화로 일반적인 멤버에 접근하듯이 사용할 수 있게 하였다.

이렇게 프로퍼티를 이용해 캡슐화를 하면,
변수를 private하게 사용하더라도 계속 메쏘드를 호출하는 번거로움 없이 멤버에 접근할 수 있다.

아래와 같이 코드를 작성하면 각각의 멤버에 외부에서 접근할 때 실제 변수 이름인 __data 대신 data로 사용할 수 있다.

```
class Node:
  # 생성자(Constructor)
  # : 객체 생성 시 초기화를 위해 호출되는 메쏘드
  def __init__(self, data=None):
    self.__data = data  # 값
    self.__prev = None  # 이전 노드
    self.__next = None  # 다음 노드

  # 소멸자(Destructor)
  # : 객체의 리소스가 삭제되어 소멸될 때 호출되는 메쏘드
  def __del__(self):
    print("data of {} is deleted".format(self.data))

  @property
  def data(self):
    return self.__data

  @data.setter
  def data(self, data):
```

```
    self.__data = data

  @property
  def prev(self):
    return self.__prev

  @prev.setter
  def prev(self, p):
    self.__prev = p

  @property
  def next(self):
    return self.__next

  @next.setter
  def next(self, n):
    self.__next = n
```

더미 이중 연결 리스트 구현

- class DoubleLinkedList 멤버

- head: 리스트의 맨 앞 노드(더미)
- tail: 리스트의 맨 뒤 노드(더미)
- d_size: 리스트에 있는 데이터의 개수

더미 이중 연결 리스트는 리스트의 맨 처음과 마지막은 실제 데이터를 저장하지 않는 더미 노드이다.
리스트의 맨 앞에 더미 노드를 만들고 이를 head라는 인스턴스 멤버로 가리키게 한다.
리스트의 맨 뒤에도 더미 노드를 만들고 이를 tail이라는 인스턴스 멤버로 가리키게 한다.
이 둘을 서로 가리키도록 연결하여 초기화 한다.
그리고 리스트의 있는 데이터 개수를 나타내는 인스턴스 멤버 d_size를 0으로 초기화 한다.

```
class DoubleLinkedList:
  def __init__(self):
    # 리스트 맨 앞, 뒤에 더미 노드를 만들고 head, tail로 가리킨다.
    self.head = Node()
    self.tail = Node()

    # head와 tail을 연결한다.
    self.head.next = self.tail
    self.tail.next = self.head

    self.d_size = 0 # 데이터 개수를 저장하는 변수
```

전체 연산 메서드 구현

```
class Node:
    # 생성자(Constructor)
    # : 객체 생성 시 초기화를 위해 호출되는 메쏘드
    def __init__(self, data=None):
        self.__data = data  # 값
        self.__prev = None  # 이전 노드
        self.__next = None  # 다음 노드

    # 소멸자(Destructor)
    # : 객체의 리소스가 삭제되어 소멸될 때 호출되는 메쏘드
    def __del__(self):
        print("data of {} is deleted".format(self.data))

    @property
    def data(self):
        return self.__data

    @data.setter
    def data(self, data):
        self.__data = data

    @property
    def prev(self):
        return self.__prev

    @prev.setter
    def prev(self, p):
        self.__prev = p

    @property
    def next(self):
        return self.__next

    @next.setter
    def next(self, n):
        self.__next = n


class DoubleLinkedList:
```

```
    def __init__(self):
        # 리스트 맨 앞, 뒤에 더미 노드를 만들고 head, tail로 가리킨다.
        self.head = Node()
        self.tail = Node()

        # head와 tail을 연결한다.
        self.head.next = self.tail
        self.tail.prev = self.head

        self.d_size = 0  # 데이터 개수를 저장하는 변수

    def empty(self):
        if self.d_size == 0:
            return True
        else:
            return False

    def size(self):
        return self.d_size

    def add_first(self, data):
        new_node = Node(data)

        new_node.next = self.head.next
        new_node.prev = self.head

        self.head.next.prev = new_node
        self.head.next = new_node

        self.d_size += 1

    def add_last(self, data):
        new_node = Node(data)

        new_node.prev = self.tail.prev
        new_node.next = self.tail

        self.tail.prev.next = new_node
        self.tail.prev = new_node

        self.d_size += 1

    def insert_after(self, data, node):
        new_node = Node(data)

        new_node.next = node.next
        new_node.prev = node

        node.next.prev = new_node
        node.next = new_node

        self.d_size += 1

    def insert_before(self, data, node):
        new_node = Node(data)

        new_node.prev = node.prev
        new_node.next = node

        node.prev.next = new_node
        node.prev = new_node

        self.d_size += 1

    def search_forward(self, target):
        cur = self.head.next

        while cur is not self.tail:
            if cur.data == target:
                return cur
            cur = cur.next
        return None

    def search_backward(self, target):
        cur = self.tail.prev
        while cur is not self.head:
            if cur.data == target:
                return cur
            cur = cur.prev
        return None

    def delete_first(self):
        if self.empty():
            return
        self.head.next = self.head.next.next
        self.head.next.prev = self.head

        self.d_size -= 1

    def delete_last(self):
        if self.empty():
            return
        self.tail.prev = self.tail.prev.prev
        self.tail.prev.next = self.tail

        self.d_size -= 1
```

```
    def delete_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

        self.d_size -= 1


def show_list(dlist):
    print('data size : {}'.format(dlist.size()))
    cur = dlist.head.next
    while cur is not dlist.tail:
        print(cur.data, end="  ")
        cur = cur.next
    print()
```

실행 코드

```
if __name__ == "__main__":
    dlist = DoubleLinkedList()
    print('*' * 60)
    print('데이터 삽입 - add_first')
    # dlist.add_first(1)
    # dlist.add_first(2)
    # dlist.add_first(3)
    # dlist.add_first(5)

    print('데이터 삽입 - add_last')
    dlist.add_last(1)
    dlist.add_last(2)
    dlist.add_last(3)
    dlist.add_last(5)
    show_list(dlist)

    print('데이터 삽입 - insert_after')
    dlist.insert_after(4, dlist.search_forward(3))
    show_list(dlist)

    print('데이터 삽입 - insert_before')
    dlist.insert_before(4, dlist.search_forward(5))
    show_list(dlist)

    print('데이터 탐색')
    target = 3
    #res=dlist.search_forward(target)
    res = dlist.search_backward(target)
    if res:
        print('데이터 {} 탐색 성공'.format(res.data))
    else:
        print('데이터 {} 탐색 실패'.format(target))
    res = None

    # print('데이터 삭제- delete_first')
    # dlist.delete_first()
    # dlist.delete_first()

    # print('데이터 삭제- delete_last')
    # dlist.delete_last()
    # dlist.delete_last()

    print('데이터 삭제- delete_node')
    dlist.delete_node(dlist.search_backward(5))

    show_list(dlist)

    print('*' * 60)
```

위의 코드를 실행하면 아래와 같은 결과를 얻을 수 있다.

```
************************************************************
데이터 삽입 - add_first
데이터 삽입 - add_last
data size : 4
1  2  3  5
데이터 삽입 - insert_after
data size : 5
1  2  3  4  5
데이터 삽입 - insert_before
data size : 6
1  2  3  4  4  5
데이터 탐색
데이터 3 탐색 성공
데이터 삭제- delete_node
data of 5 is deleted
data size : 5
1  2  3  4  4
************************************************************
data of None is deleted
data of 4 is deleted
data of None is deleted
data of 4 is deleted
data of 1 is deleted
```

```
data of 3 is deleted
data of 2 is deleted
```

1

'학교 공부 > 알고리즘' 카테고리의 다른 글

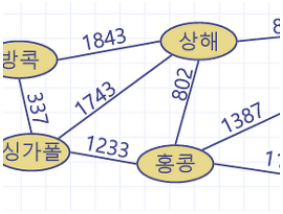
'학교 공부/알고리즘' Related Articles

NO IMAGE

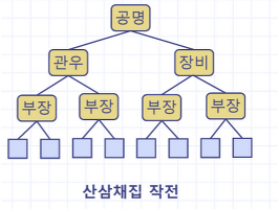
[알고리즘](Python) 백준 10828 - 스택

NO IMAGE

[알고리즘](Python) 백준 10845 - 큐



[알고리즘](C언어) 그래프 알고리즘



[알고리즘](C언어) 분할통치법 - 합병정렬

Secret

안녕하세요! 어떤 댓글이든 환영합니다!

댓글달기