Introduction to Python and Programming Language

Koscom Algorithm Lecture

신승우

Tuesday 4th September, 2018

Outline

Recursive Definition of List

리스트는 head, tail의 순서쌍이며, head는 임의의 데이터, tail은 다른 리스트를 말합니다. 즉, 리스트의 정의 자체가 이미 재귀적입니다. lisp 같은 언어에서 아주 적극적으로 활용되며, 다른 언어에서도 기본적으로 많이 쓰이는 ADT이자 데이터구조입니다.
재귀적인 구조를 가진 것은 리스트에 대한 많은 함수가 재귀적으로 작성될 수 있음을 의미합니다. 사실 리스트같은 경우 워낙 구조가 간단하기에 굳이 재귀를 사용할 필요가 없지만, 트리에서 사용될 재귀의 좋은 연습이 되기 때문에 알아두는 것이 좋습니다.

ADT List I

- __init__ : List Constructor
- __iter__ : 리스트의 원소를 한번씩 traverse하는 iterator.
- is_empty : 리스트가 비어 있으면 True, 아니면 False를 리턴.
- prepend / append : 리스트의 앞/뒤에 어떤 element를 넣는 함수. 기존 리스트를 변화시킴.

ADT List II

- add : 리스트의 주어진 위치에 element를 추가하는 함수. 기존 리스트를 변화시킴.
- head : 리스트의 첫 번째 원소를 리턴.
- tail : 리스트의 첫 번째 원소를 제외한 나머지 원소들을 리턴.
- sort : 리스트를 주어진 기준에 따라 소팅하여 새 리스트를 반환. 기존 리스트는 변하지 않음.
- max/min : 리스트의 원소 중 주어진 기준으로 볼 때 가장 큰/작은 원소 반환. 기존 리스트는 변하지 않음.

Dynamic Array Datastructure

Array는 연속된 메모리 블럭을 지칭하는 데이터구조입니다. 이 때 Dynamic이라는 말은, 할당된 array 구역이 다 이용되면 어레이 구역을 늘린다는 의미로 쓰입니다. Array의 시작 주소와 각 데이터의 크기를 안다는 가정하에서, array 내 임의의 원소에는 상수 시간에 접근이 가능합니다. 이것이 링크드리스트와 어레이의 가장 큰 차이입니다. 어레이를 이용하여 operation들을 구현해 보겠습니다.

prepend/append

- prepend : prepend의 경우, 어레이의 모든 원소를 원소 사이즈만큼 옆으로 옮긴 후 어레이의 처음 위치에 새 elem을 추가해야 합니다. 어레이의 모든 원소를 옆으로 옮기는 것이 O(n)이므로 그만큼의 시간이 듭니다.
- append : append의 경우, 어레이 맨 뒤의 원소의 주소의 끝에 elem을 붙이면 되기 때문에 O(1)입니다. 하지만, worst case의 경우 O(n)인데, 이 두 경우를 합쳐서 생각하면 평균은 O(1)이 됩니다. [1]

[1] Amortized Cost Analysis로 생각해볼 수 있습니다.

add(idx, elem)

add의 경우, idx에 따라서 옮겨야 하는 원소의 갯수가 다르지만 평균적으로 $\frac{n}{2}$개를 옮겨야 하므로 O(n)입니다.

head()/tail()

둘 다 O(1)입니다.

max()/min()

둘 다 O(n)입니다.

sort()

일반적인 소팅 알고리즘을 따릅니다. 어떤 알고리즘을 쓰냐에 따라서 달라집니다.

Linked List Datastructure

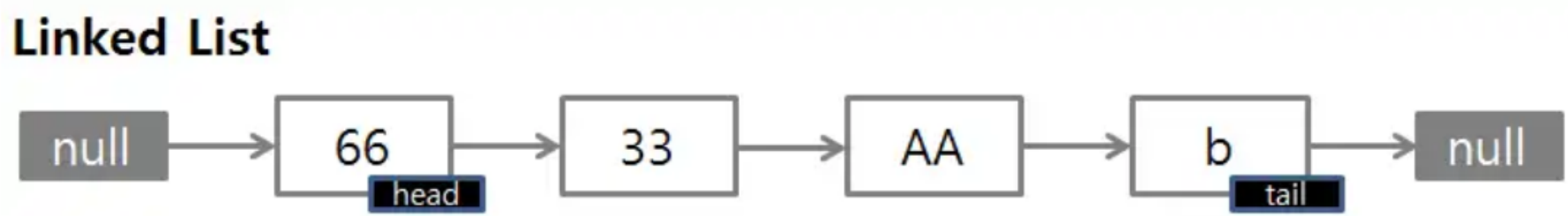

Figure: Linked List의 구조

Linked list는 위와 같이, 데이터와 그 다음 데이터로의 주소값으로 이루어진 데이터구조입니다. 맨 처음과 끝에 Null값을 붙일 수도, 안 붙일 수도 있습니다. Linked list는 list의 개념적인 정의에 매우 충실한 구조인데, 이는 head 뒤에 다음 element로의 포인터를 tail로 보면 tail 또한 linked list 이기 때문입니다.

prepend

prepend : prepend의 경우, 새 LinkedList를 만든 후, head에 item을 첨가합니다. 이후 새로 만든 LinkedList의 tail을 기존 LinkedList로 만든 후, 새로 만든 LinkedList를 반환합니다. 이 과정에서 tail은 언제나 None 아니면 LinkedList일 것입니다. 또한, 이 operation은 지금 리스트 안에 들어 있는 element의 갯수와 상관 없이 상수 시간이 걸릴 것입니다.

append

만약 리스트가 비어 있는 경우 prepend와 똑같이 시행합니다. 그렇지 않은 경우, 빈 LinkedList를 만들어 head에 item을 넣습니다. 그 후, 원 LinkedList의 tail에서 None을 발견할 때까지 LinkedList를 traverse한 후, tail이 None인 LinkedList를 발견하면 tail을 새로 만든 LinkedList로 대체합니다. 이 과정에서는 LinkedList를 한 번 끝까지 traverse 해야 하므로, 리스트 내의 element 갯수에 선형적으로 비례하는 시간이 걸립니다. [2]

[2] 이를 방지하기 위해서, tail 뿐만 아니라 앞쪽 방향의 list도 미리 저장하는 doubly linked list를 사용하기도 합니다.

add(idx, elem)

head에서 head.tail로 idx번 움직인 후, 그곳에서 prepend를 시행하면 되므로 O(n)입니다.

remove(elem)

head에서 elem을 발견할 때까지 움직인 후, 그곳에서 elem을 발견하면 제거하면 되므로 O(n)입니다.

head()/tail()

리스트의 첫 번째 원소를 head, 나머지를 tail로 리턴합니다. 둘 다 O(1) 입니다.

max()/min()

둘 다 O(n)입니다.

sort()

역시 소팅 알고리즘을 똑같이 따릅니다.