Welcome to Data C88C!

Lecture 16: Data Examples
Tuesday, July 22nd, 2025
Week 5
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Ants project is out!
- Mid-semester survey feedback: [link]
  - If 75% of the class completes this form by Monday July 28th at 11:59 PM, everyone will receive 1 point of extra credit! If this goal is not met, nobody will receive the extra point.

Lecture Overview

- More data examples
- Data structure overview thus far
  - Python builtins: list, dict
  - Linked list (`Link`)
- Why choose one data structure over another?
  -

Data structures in C88C (so far)

- Python built-ins: list, dict
- Linked list (`Link`)
- Why should we use one over the other?
- One answer: **performance**

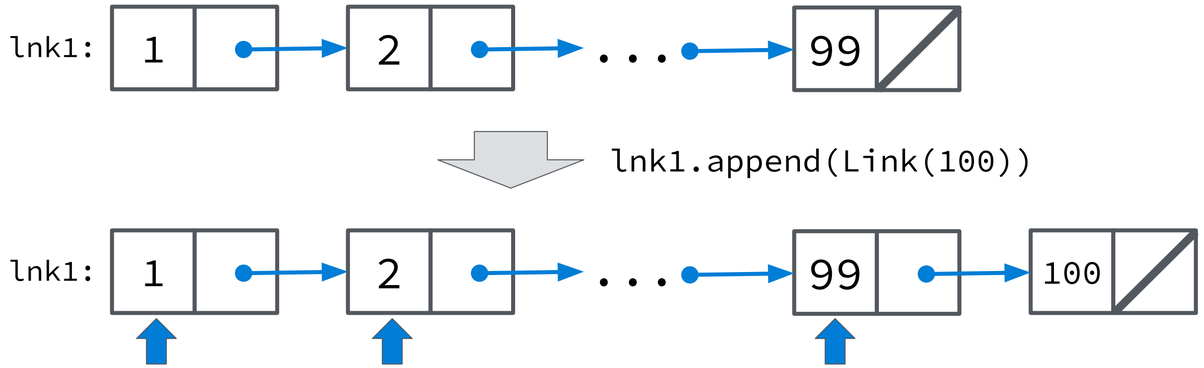
`Link`: operations

- Consider a linked list (`Link`) with length N.

Question: In terms of N, how long does it take to append a new element to the end of the linked list?

Let's use "`rest` pointer traversals" as our unit of time

Answer: it depends on the implementation of `lnk1.append()`. But, here's a simple implementation: start from the first `Link` instance, and follow all `rest` pointers until we reach the end. This would take N `rest` pointer traversals.




```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def append(self, lnk):
        last_lnk = get_last(self)
        last_lnk.rest = lnk

def get_last(lnk):
    if lnk.rest == Link.empty:
        return lnk
    return get_last(lnk.rest)
```

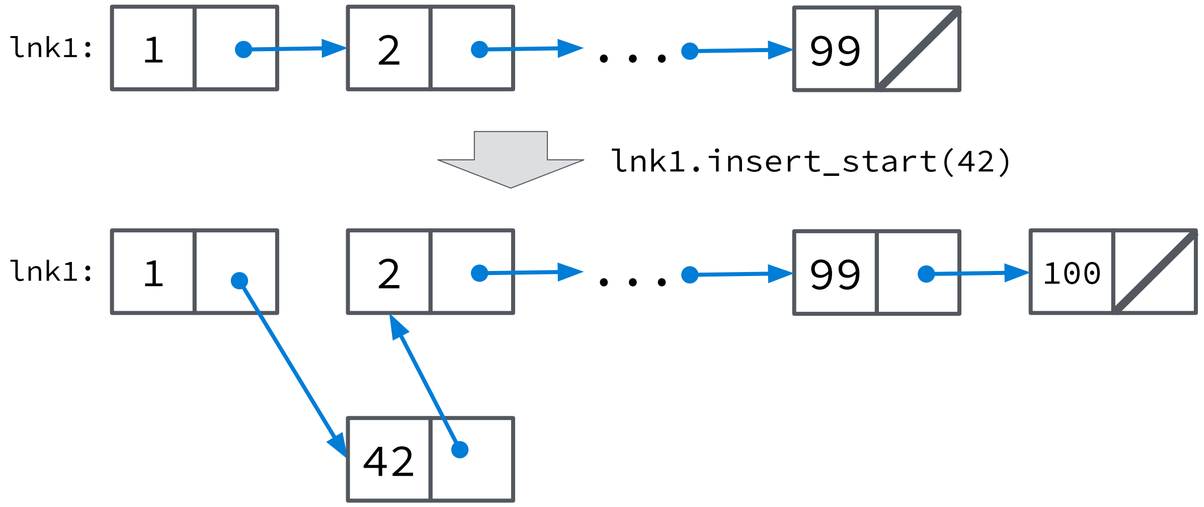
`Link`: operations

- Consider a linked list (`Link`) with length N.

Question: In terms of N, how long does it take to insert a new element at the beginning of the linked list?

Let's use "`rest` pointer traversals" as our unit of time

Answer: 0 `rest` pointer traversals. Can be implemented via a single pointer assignment:




```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def append(self, lnk):
        last_lnk = get_last(self)
        last_lnk.rest = lnk

    def insert_start(self, val):
        self.rest = Link(val, rest=self.rest)
```

Comparing data structures

- Using this methodology (count number of operations), we can quantify the performance of the `Link` class operations, and compare it to other data structures like: list, dict
- Note: Rather than saying "N operations", we'll use notation "O(N)" to loosely mean: proportional to N operations. "Big-O notation" [link]

Operation	**# operations (`Link`)**	**# operations (py list)**	**# operations (py dict)**
Append to end	O(N)	O(1)	N/A
Insert at beginning	O(1)	O(N)	N/A
Contains	O(N)	O(N)	O(1)*
Get item at index	O(N)	O(N)	O(1)*
Set item at index	O(N)	O(N)	O(1)*
...			

* it turns out for dict, the average case is O(1), but worst case is O(N). To learn more, read about hash tables

Some takeaways:

- `Link` is faster than list for inserting at the beginning, but slower for inserting at the end
- `dict` is great for lookup-type usage!

Verdict: the "best" data structure to use is dependent on your expected data usage patterns.

Tip: cs61B does a deep dive into this kind of stuff. It's a neat class!

Source for Python builtin data structs: [link]

Comparing data structures

Operation	**# operations (`Link`)**	**# operations (py list)**	**# operations (py dict)**
Append to end	O(N)	O(1)	N/A
Insert at beginning	O(1)	O(N)	N/A
Contains	O(N)	O(N)	O(1)*
Get item at index	O(N)	O(N)	O(1)*
Set item at index	O(N)	O(N)	O(1)*
...			

Question: how can we modify `Link`'s "Append to end" to reduce its # operations from O(N) to O(1)?

Answer: have the linked list keep track of both the beginning AND the end of the linked list ("head", "tail").

`Link`: keeping track of `head` and `tail`

Question: what's the best way to keep track of the "head" and the "tail"?

Answer: here's one way, add a new `tail` instance attribute to each linked list node. And, let's utilize inheritance too:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

O(N)

```
    def append(self, lnk):
        last_lnk = get_last(self)
        last_lnk.rest = lnk
```

```
class LinkWithTail(Link):
    def __init__(self, first, rest=Link.empty):
        super().__init__(first, rest)
        self.tail = get_last(self)
```

O(1)

```
    def append(self, lnk):
        self.tail.rest = lnk
        self.tail = lnk
```

Now, `LinkWithTail.append()` is fast: no need to traverse the linked list to reach the end. Neat!

Question: any downsides with `LinkWithTail`?

Answer: now, creating new `LinkWithTail` instances can be slow: each time we create a new instance, we have to traverse the rest of the linked list (`get_last()`).
Also, perhaps wasteful for each `LinkWithTail` instance to keep track of the tail...

Idea: rather than add `tail` as an instance variable to `LinkWithTail`, create a "wrapper" class that keeps track of the beginning and end nodes.

`Link`: keeping track of `head` and `tail`

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

O(1)

```
class LinkedList:
    def __init__(self, lnk):
        self.head = lnk
        self.tail = get_last(lnk)

    def append(self, lnk):
        self.tail.rest = lnk
        self.tail = lnk
```

O(1)

```
>>> lnk_lst = LinkedList(Link(1, Link(2, Link(3))))
>>> lnk_lst.append(Link(4, Link(5)))  # O(1)
>>> lnk_lst.head  # __repr__ code not shown here
<1 2 3 4 5>
```

With this approach, we get:

- Fast O(1) append, via `tail`
- Reduced memory usage (single `tail` per `LinkedList`, rather than a `tail` per each `Link` instance)
- Arguably a better software design

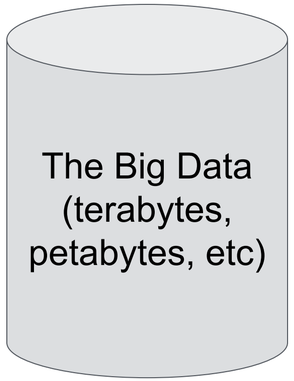
An example of where we used **composition** (`LinkedList` contains a `Link` as an attribute) rather than **inheritance** (`LinkWithTail`).

Takeaway for C88C: now that we are comfortable **writing** code, we can start **reasoning** about code. Things like: efficiency (Big-O notation) and software design (eg composition vs inheritance).

In cs61B, you will cover topics like this `LinkedList` rabbit hole, and more broadly study how fundamental data structures like list and dict are implemented, and their efficiency for different operations. It's a neat class, and was one of my favorite undergrad CS courses!

Data structures for Data Science

- One reason to care about efficiency of data structures: big data!
- In the data science / AI / ML world, datasets are often too large to fit on a single machine
  - Ex: a typical commercial laptop/desktop typically has ~16gb-32gb of CPU memory.
  - A big-data dataset can be **terabytes** (1000's of GB's) or even **petabytes** (millions of GB's) large!
- To effectively work on these datasets, we need two* techniques
  - Efficient data structures to store the data
    - Ex: "smart" file formats like parquet [link]
  - Distributed computing techniques to efficiently process the data
    - Idea: use a cluster of machines to process data
    - Ex: Hadoop MapReduce, Apache Spark



Examples: images/video, user engagement logs, The Internet, etc.

* The secret third technique: lots of $, either in building+maintaining your own compute cluster, or using cloud computing platforms like AWS EC2.