Linked-List Characteristics

- Recursive structure: a linked list (L) is a sequence of nodes that contain a data item (L.first) + reference to a linked list (L.rest). A special value (Link.empty) represents an empty list.
- Suggests that operations on it can be formulated as a tail recursion, or, equivalently, as an iteration.
- Complexity:
  - Fetch item or node \(#k\) for constant \(k\): \(\Theta(1)\).
  - Add node to the front: \(\Theta(1)\).
  - Add a node after a node, \(M\), in the middle of a list, assuming we have already found \(M\): \(\Theta(1)\).
  - Fetch item or node \(#k\) for arbitrary \(k\): \(\Theta(k)\) or (since \(k < N\) for \(N\) the length of the list) \(\Theta(N)\) in the worst case.
  - Find length of list or find a data item in the list: \(\Theta(N)\) worst case.
class Link:
    """A linked list node."""
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __repr__(self):
        """Return string denotation of SELF as Link(first, rest)."""
    def __str__(self):
        """Return string denotation of SELF as <item item ...>."""

def toLinked(L):
    """Returns a linked-list representation of the Python iterable L."""
    if len(L) == 0:
        return Link.empty
    result = last = Link(L[0], Link.empty)
    for item in L[1:]:
        last.rest = Link(item)
        last = last.rest
    return result
class Link:
    """A linked list node.""
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __repr__(self):
        """Return string denotation of SELF as Link(first, rest).""
    def __str__(self):
        """Return string denotation of SELF as <item item ...>.""

def toLinked(L):
    """Returns a linked-list representation of the Python iterable L.""
    if len(L) == 0:
        return Link.empty
    result = last = Link(L[0], Link.empty)
    for item in L[1:]:
        last.rest = Link(item)
        last = last.rest
    return result

#### Can you make this work for nested Python lists? ####
Exercise: Find Node \( k \) in List

```python
def split(L):
    
    """Returns (Mid, Last, Length), where Last is the last node in
    linked list L, Mid is the node at or (for even length) just before
    the middle, and Length is the length."""
```

- Do this with *one pass* through \( L \), with constant extra space (i.e.,
  iteratively with no auxiliary lists or other containers).
Exercise: Find Node $k$ in List

```python
def split(L):
    """Returns (Mid, Last, Length), where Last is the last node in
    linked list L, Mid is the node at or (for even length) just before
    the middle, and Length is the length.
    If L is empty, returns (empty, empty, 0).""
```

- Do this with `one pass` through L, with constant extra space (i.e.,
  iteratively with no auxiliary lists or other containers).

```
          Mid            Last
  5 10 15 20 25 30 35 ()
```

Last modified: Mon Apr 26 13:27:03 2021
def split(L):
    """Returns (Mid, Last, Length), where Last is the last node in linked list L, Mid is the node at or (for even length) just before the middle, and Length is the length. If L is empty, returns (empty, empty, 0)."""

• Do this with one pass through L, with constant extra space (i.e., iteratively with no auxiliary lists or other containers).
Exercise: Find Node $k$ in List

def split(L):
    """Returns (Mid, Last, Length), where Last is the last node in
    linked list L, Mid is the node at or (for even length) just before
    the middle, and Length is the length.
    If L is empty, returns (empty, empty, 0)."""

- Do this with *one pass* through L, with constant extra space (i.e.,
  iteratively with no auxiliary lists or other containers).
Exercise: Find Node $k$ in List

```python
def split(L):
    """Returns (Mid, Last, Length), where Last is the last node in linked list L, Mid is the node at or (for even length) just before the middle, and Length is the length.
    If L is empty, returns (empty, empty, 0)."""

• Do this with **one pass** through L, with constant extra space (i.e., iteratively with no auxiliary lists or other containers).
```

![Diagram of linked list with nodes 5, 10, 15, 20, 25, 30, 35, and arrows indicating Mid and Last nodes.]
def split(L):
    """Returns (Mid, Last, Length), where Last is the last node in linked list L, Mid is the node at or (for even length) just before the middle, and Length is the length. If L is empty, returns (empty, empty, 0)."""

• Do this with one pass through L, with constant extra space (i.e., iteratively with no auxiliary lists or other containers).
Exercise: Intersperse Lists (I)

We seek a recursive solution to the following:

```python
def intersperse(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose
values satisfy PRED (a one-argument, boolean function) are
followed by successive values from linked list INSERTS, until
INSERTS is exhausted. The function is non-destructive.

>>> data = toLinked([1, 2, 3, 4, 5])
>>> alt = toLinked([10, 11, 12, 13])
>>> print(intersperse(data, lambda x: x % 2 == 1, alt))
<1 10 2 3 11 4 5 12>
>>> print(intersperse(data, lambda x: True, alt))
<1 10 2 11 3 12 4 13 5>
"""
```
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we'll use a dummy sentinel node just before the beginning of the resulting list to cut down on special cases.

```python
def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
```
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we'll use a dummy sentinel node just before the beginning of the resulting list to cut down on special cases.

```python
def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
```

```
L

1 ─→ 2 ─→ 3 ─→ 4 ─→ 5 ()

inserts

10 ─→ 11 ()

sentinel ─→ last
```
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we'll use a dummy sentinel node just before the beginning of the resulting list to cut down on special cases.

```python
def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
```

![Diagram showing the intersperse process with sentinel node and linked list examples.](image)
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we’ll use a dummy sentinel node just before the beginning of the resulting list to cut down on special cases.

```python
def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
```

![Diagram of linked list operations](attachment:link_list_diagram.png)
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we'll use a dummy *sentinel node* just before the beginning of the resulting list to cut down on special cases.

```python
def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest
```

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
Exercise: Intersperse Lists (II)

This time, give an iterative solution. Here, we’ll use a dummy sentinel node just before the beginning of the resulting list to cut down on special cases.

def intersperse2(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is non-destructive.""
    sentinel = Link(None)
    # Code
    return sentinel.rest

intersperse2(toList(1, 2, 3, 4, 5), lambda x: x%2 == 1, toList(10, 11))
Exercise: Intersperse Lists (III)

This time, give a recursive, *destructive* solution. That is, we do not create any new Links, but instead modify existing ones as needed to create the result list, possibly destroying the original data in the \( L \) and \( \text{inserts} \) arguments.

```python
def dintersperse(L, pred, inserts):
    """Returns a copy of linked list \( L \) in which the items whose values satisfy \( \text{PRED} \) (a one-argument, boolean function) are followed by successive values from linked list \( \text{INSERTS} \), until \( \text{INSERTS} \) is exhausted. The function is destructive and creates no new Link nodes.""
```

Last modified: Mon Apr 26 13:27:03 2021
Exercise: Intersperse Lists (III)

This time, give a recursive, **destructive** solution. That is, we do not create any new Links, but instead modify existing ones as needed to create the result list, possibly destroying the original data in the `L` and `inserts` arguments.

```python
def dintersperse(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is destructive and creates no new Link nodes.""

data = toList(1, 2, 3, 4, 5); alt = toList(10, 11)
R = dintersperse(data, lambda x: x%2 == 1, alt)
```

Last modified: Mon Apr 26 13:27:03 2021
This time, give a recursive, *destructive* solution. That is, we do not create any new Links, but instead modify existing ones as needed to create the result list, possibly destroying the original data in the L and inserts arguments.

```python
def dintersperse(L, pred, inserts):
    """Returns a copy of linked list L in which the items whose values satisfy PRED (a one-argument, boolean function) are followed by successive values from linked list INSERTS, until INSERTS is exhausted. The function is destructive and creates no new Link nodes.""
```

data = toList(1, 2, 3, 4, 5); alt = toList(10, 11)
R = dintersperse(data, lambda x: x%2 == 1, alt)
```
Binary Trees

- We've looked at trees in lecture and homework.
- Let's consider a slight variation of what you've seen that is specialized to binary trees:

```python
class BTree(Tree):
    """A tree with exactly two branches, which may be empty."
    empty = None  # Placeholder

    def __init__(self, label, left=None, right=None):
        self.label = label
        self.left = left or BTree.empty
        self.right = right or BTree.empty

    def __repr__(self):
        return ...

BTree.empty = BTree(None)
```

- As it happens, I could simply have left `BTree.empty` as `None`. However, I wanted to make it a special (unique) `BTree` node. What might this be useful for, and why did I have to do it this way?
Binary Search Trees

• We just saw binary search trees on Wednesday in the context of Scheme.

• Of course, the same algorithms apply to Python.

• Today, we’ll use simple strings as labels. All labels in a left subtree of $T$ must be lexicographically less than $T$’s label, and all labels in the right subtree must be greater.

![Binary Search Tree Diagram]

- cow
  - bear
    - axolotl
    - bat
  - platypus
    - cougar
    - lion
    - viper
Binary Search Trees

- We just saw binary search trees on Wednesday in the context of Scheme.
- Of course, the same algorithms apply to Python.
- Today, we’ll use simple strings as labels. All labels in a left subtree of $T$ must be lexicographically less than $T$’s label, and all labels in the right subtree must be greater.

```python
def isIn(T, target):
    """Return True iff T contains the label TARGET."""
    ???
```
Binary Search Trees

- We just saw binary search trees on Wednesday in the context of Scheme.
- Of course, the same algorithms apply to Python.
- Today, we'll use simple strings as labels. All labels in a left subtree of $T$ must be lexicographically less than $T$'s label, and all labels in the right subtree must be greater.

```
def add(T, target):
    """Destructively modify T to add the label TARGET, if it is not already present. Return the resulting tree."""
    ???
```