UIT2201 Programming and Data Structures

Linked Lists

Chandrabose Aravindan
<AravindanC@ssn.edu.in>

Professor of Information Technology
SSN College of Engineering

June 25, 2022

Disadvantages of array-based lists

- Memory may be allocated more than necessary — the size grows exponentially and more memory may be unused in larger lists

Disadvantages of array-based lists

- Memory may be allocated more than necessary — the size grows exponentially and more memory may be unused in larger lists
- Amortized bounds for basic operations may be unacceptable in real-time systems

Disadvantages of array-based lists

- Memory may be allocated more than necessary — the size grows exponentially and more memory may be unused in larger lists
- Amortized bounds for basic operations may be unacceptable in real-time systems
- Arbitrary insertions and deletions are costly

Linked List

- Just-in-time allocation of memory of an object added to the list and release of memory when the object is removed

Linked List

- Just-in-time allocation of memory of an object added to the list and release of memory when the object is removed
- Allocated memory may not be contiguous! So, along with each object we should store reference to the next (and previous) object!

Linked List

- Just-in-time allocation of memory of an object added to the list and release of memory when the object is removed
- Allocated memory may not be contiguous! So, along with each object we should store reference to the next (and previous) object!
- We need a light-weight object (often referred to as a Node) to store an object along with other book-keeping information

Linked List

- Just-in-time allocation of memory of an object added to the list and release of memory when the object is removed
- Allocated memory may not be contiguous! So, along with each object we should store reference to the next (and previous) object!
- We need a light-weight object (often referred to as a Node) to store an object along with other book-keeping information
- A list can be implemented as a sequence of linked nodes

Node Structure

- The simplest possible node structure contains two fields: a reference to an object (stored in that node); and a reference to the 'next' node in the sequence

Node Structure

- The simplest possible node structure contains two fields: a reference to an object (stored in that node); and a reference to the 'next' node in the sequence
- The 'next' field will be 'None' where there is no next node

Node Class

- It should be easy to implement a Python class for nodes with single (next) link

Node Class

- It should be easy to implement a Python class for nodes with single (next) link

```
class Node(object):

    __slots__ = ['_item', '_next']

    def __init__(self, item=None, next=None):
        self._item = item
        self._next = next
```

Node Class

- The constructor should be sufficient, but we may define a few more methods

Node Class

- The constructor should be sufficient, but we may define a few more methods

```
class Node(object):

    def __str__(self):
        return str(self._item)

    def setItem(self, item=None):
        self._item = item

    def setNext(self, next=None):
        self._next = next

    def getItem(self):
        return self._item

    def getNext(self):
```

Singly Linked Lists

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object
- But then, 'end' position can not be defined!

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object
- But then, 'end' position can not be defined!
- Moreover, as we will see shortly, to manipulate a node in a linked list, we need the reference to the previous node!

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object
- But then, ‘end’ position can not be defined!
- Moreover, as we will see shortly, to manipulate a node in a linked list, we need the reference to the previous node!
- Considering these points, and also to simplify the implementation, we will adopt the notion of position as the reference to the previous node (and not the node containing the object)

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object
- But then, 'end' position can not be defined!
- Moreover, as we will see shortly, to manipulate a node in a linked list, we need the reference to the previous node!
- Considering these points, and also to simplify the implementation, we will adopt the notion of position as the reference to the previous node (and not the node containing the object)
- How do we then represent the 'begin' position?

Positions in Linked Lists

- It is easy to think of Position of an object as a reference to the node containing that object
- But then, ‘end’ position can not be defined!
- Moreover, as we will see shortly, to manipulate a node in a linked list, we need the reference to the previous node!
- Considering these points, and also to simplify the implementation, we will adopt the notion of position as the reference to the previous node (and not the node containing the object)
- How do we then represent the ‘begin’ position?
- We will introduce a dummy header node!

Singly linked list with a dummy header

Implementation of singly linked list

- We now know exactly how to construct an empty list

Implementation of singly linked list

- We now know exactly how to construct an empty list

```
from Node import Node

class LinkedList(object):

    def __init__(self):
        self._head = Node()    # Initialize an empty li
```

Explicit ‘end’ position?

- To iterate through a list, we need position operations ‘begin’, ‘end’, and ‘next’

Explicit 'end' position?

- To iterate through a list, we need position operations 'begin', 'end', and 'next'
- A simple analysis reveals that finding the 'end' position is not a constant time operation

Explicit 'end' position?

- To iterate through a list, we need position operations 'begin', 'end', and 'next'
- A simple analysis reveals that finding the 'end' position is not a constant time operation
- We may explicitly store the 'end' position, but that increases the coding effort

Explicit ‘end’ position?

- To iterate through a list, we need position operations ‘begin’, ‘end’, and ‘next’
- A simple analysis reveals that finding the ‘end’ position is not a constant time operation
- We may explicitly store the ‘end’ position, but that increases the coding effort
- Similar arguments apply for ‘len’ as well

Explicit ‘end’ position?

- To iterate through a list, we need position operations ‘begin’, ‘end’, and ‘next’
- A simple analysis reveals that finding the ‘end’ position is not a constant time operation
- We may explicitly store the ‘end’ position, but that increases the coding effort
- Similar arguments apply for ‘len’ as well

```
from Node import Node

class LinkedList(object):

    def __init__(self):
        self._head = self._end = Node() # Initialize a
        self._size = 0  # Initial size is 0 (list is e
```

Position Operations

- It should be easy now to implement the position operations ‘begin’, ‘end’, and ‘next’

Position Operations

- It should be easy now to implement the position operations ‘begin’, ‘end’, and ‘next’

```
class LinkedList(object):

    def begin(self):
        return self._head

    def end(self):
        return self._end

    def next(self, pos):
        return pos._next
```

isEmpty?

- We have several options to check if a list is empty!

isEmpty?

- We have several options to check if a list is empty!

```
class LinkedList(object):

    def isEmpty(self):
        return self._head == self._end
```

Linear Search

- Linear search is the only reasonable option for us to search for an object in a list

Linear Search

- Linear search is the only reasonable option for us to search for an object in a list

```
class LinkedList(object):

    def find(self, item):
        pos = self._head
        while pos._next is not None:
            if (pos._next._item == item):
                return pos
            else:
                pos = pos._next
        return None
```

Retrieve and Object

- How do we retrieve an object at a given position?

Retrieve and Object

- How do we retrieve an object at a given position?

```
class LinkedList(object):

    def retrieve(self, pos):
        if pos is None:
            return None
        elif pos._next is None:    # Can not retieve fr
            return None
        else:
            return pos._next._item
```

Inserting an Object

- How do we insert an object at a given position?

Inserting an Object

- How do we insert an object at a given position?

```
class LinkedList(object):

    def insert(self, item, pos=None):
        if (pos is None):
            pos = self._head
        pos._next = Node(item, pos._next)
        if (pos == self._end):
            self._end = self._end._next
        self._size += 1
        return self    # Return the updated list
```

Inserting an Object

- We can have a separate method for 'append', if it is frequently required

- We can have a separate method for ‘append’, if it is frequently required

```python
class LinkedList(object):

    def append(self, item):
        self._end._next = Node(item, self._end._next)
        self._end = self._end._next
        self._size += 1
        return self    # Return the updated list
```

Deleting an Object

- How do we delete an object at a given position?

Deleting an Object

- How do we delete an object at a given position?

```
class LinkedList(object):

    def delete(self, pos):
        if pos is None:
            return self
        if pos._next is None:  # Can not delete from '
            return self
        if (pos._next == self._end):
            self._end = pos
        pos._next = pos._next._next
        self._size -= 1
        return self
```

Iterator for our Linked List

- How do we implement an iterator for our linked list?

Iterator for our Linked List

- How do we implement an iterator for our linked list?
- Python automatically provides an iterator if both '__len__' and '__getitem__' are defined

Iterator for our Linked List

- How do we implement an iterator for our linked list?
- Python automatically provides an iterator if both '__len__' and '__getitem__' are defined
- Defining '__len__' is straight forward, but '__getitem__' does not make much sense

Iterator for our Linked List

- How do we implement an iterator for our linked list?
- Python automatically provides an iterator if both '__len__' and '__getitem__' are defined
- Defining '__len__' is straight forward, but '__getitem__' does not make much sense

```
class LinkedList(object):

    def __len__(self):
        """ Returns the length of the list
        """
        return self._size
```

Iterator for our Linked List

- Since the concept of index is not there, '__getitem__' does not make much sense
- So, we will implement an iterator ourself

Iterator for our Linked List

- Since the concept of index is not there, '__getitem__' does not make much sense
- So, we will implement an iterator ourself

```
class LinkedList(object):

    def __iter__(self):
        p = self._head
        while p._nextNode is not None:
            yield p._nextNode._itemNode
            p = p._nextNode
```

Copy Constructor

- It may be useful to have a copy constructor for our Linked List

Copy Constructor

- It may be useful to have a copy constructor for our Linked List

```
class LinkedList(object):

    def __init__(self, lst=None):
        # Initialize an empty list
        self._head = self._end = Node()
        self._size = 0
        # Copy constructor
        # Does not use internal data structure
        # So, 'lst' can be any Python sequence
        # or ArrayList or LinkedList
        # DRAWBACK: Linear time complexity
        if (lst is not None):
            for item in lst:
                self.append(item)
```

Decomposition of Linked List

- As discussed earlier, one standard way of decomposing a list is to define 'head' and 'tail' of a list

Decomposition of Linked List

- As discussed earlier, one standard way of decomposing a list is to define ‘head’ and ‘tail’ of a list
- ‘head’ returns the first object in the list (if the list is not empty)

Decomposition of Linked List

- As discussed earlier, one standard way of decomposing a list is to define 'head' and 'tail' of a list
- 'head' returns the first object in the list (if the list is not empty)
- 'tail' returns the list without the first element

Decomposition of Linked List

- As discussed earlier, one standard way of decomposing a list is to define ‘head’ and ‘tail’ of a list
- ‘head’ returns the first object in the list (if the list is not empty)
- ‘tail’ returns the list without the first element

```
class LinkedList(object):

    def head(self):
        """ Returns the first item in this list.
        Not defined for an empty list.
        This list is not modified.
        """
        if (self._head._nextNode is None):
            return None
        return self._head._nextNode._itemNode
```

Decomposition of Linked List

- Like ‘head’, we will implement ‘tail’ also as a non-mutating method that returns a copy of this list without the first element

Decomposition of Linked List

- Like ‘head’, we will implement ‘tail’ also as a non-mutating method that returns a copy of this list without the first element

```
class LinkedList(object):

    def tail(self):
        """ Returns 'tail', that is list without the f
        This list is not modified.
        """
        lst = LinkedList(self) # Create a copy of this
        if ( lst._head != lst._end ):
            lst._head = lst._head._nextNode
            lst._size -= 1
        return lst
```

Summary

- We have discussed link-based implementation of ADT List

What next?

- We will explore variations of link-based implementations of List and also consider link-based Stack and Queue ADTs
- We will look at some of the applications of Lists, Stacks, and Queues