The Ordered List Data Structure

Vivek Singh

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2018

The Ordered List

- An ordered list is another implementation of the List data structure.
- In this type, the items of the list maintain a relative position to each other, in that they are ordered.
- The ordering is typically ascending or descending based on the choice of implementation.
- The list items must have a meaningful comparison operation in place.
- Many of the operations of the ordered list are the same as the unordered list where the ordered characteristic is not involved.

Essential operations in an Ordered List

- Ability to create List instances.
- Ability to add items to the list while also making sure that the ordering is preserved. We assume the item is not already present in the list.
- Ability to remove an item from the list.
- Searching and finding an item.
- Ability to check if the list is empty or not.
- Find out the size of the list.
- Removing the last item in the list by popping it.
- Returning the index position of a list item.

Implementing an Ordered List

In order to implement the ordered list, we must follow certain rules such as,

- Keep the ordered structure intact.
- Use the linked structure and notion of Node to achieve the relative positioning of the items.
- Like in the unordered list, once again, an empty list will be denoted by a head reference to None.
- All the operations, except the search and add operations are the same as seen in unordered list.

The Add operation

The add method in ordered lists differ very much from the add method in the Unordered list.

- Using Link traversal, we must traverse the list to find out where the item could be inserted depending on the order.
- For example, in our list containing the following elements [17,23,45,68,78,79,90], if we have to add 75, the method should know that it goes in between 68 and 78.
- It is therefore helpful to perform this operation with an additional reference along with the current reference as seen in previous exercises.

```
def add(self, item):
        current = self.head
        tail = None
        stop = False
        while current != None and not stop:
                if current.get_data() > item:
                        stop = True
                else:
                        tail = current
                        current = current.get_next()

        temp = Node(item)
        if tail == None:
                temp.set_next(self.head)
                self.head = temp
        else:
                temp.set_next(current)
                tail.set_next(temp)
```

Search Operation

The search operation works pretty much similar to a search in unordered list.

- The same link traversal mechanism is used, but since, the items are sorted, the operation is more efficient here.
- For example, if in a list containing [17,23,45,68,78,79,90], we are looking for 48.
- When the links are traversed in search of the item, after 45 is passed, when the next node with value 68 is encountered, the comparison will result in a value greater than the item being searched for and therefore the search will be terminated right there.
- Because, the underlying characteristic of the list has already told us that going further is inconsequential.

```
def search(self, item):
        current = self.head
        found = False
        stop = False
        while current != None and not found and
                not stop:
                if current.get_data() == item:
                        found = True
                else:
                        if current.get_data() > item:
                                stop = True
                        else:
                                current = current.
                                get_next()
        return found
```

Summary

- Ordered lists are more or less similar to unordered lists except a few key operations such as add and search due to the unique underlying structure.
- Ordered lists offer significant improvements in search over the unordered counterparts because the items are ordered and mostly the average case performance is significantly higher.
- Most operations are similar in implementation and performance to the unordered lists.