Advanced Linked Lists

• Doubly Linked Lists
• Circular Linked Lists
• Multi-Linked Lists
Review

- The singly linked list:
  - consists of nodes linked in a single direction.
  - access and traversals begin with the first node.
- What if we want to traverse the nodes in reverse order?
Doubly Linked List

- A linked list in which each node contains a data component(s) and two links:
  - one pointing the next node and
  - one pointing to the preceding node.
Doubly Linked: Order

- Can be unsorted or sorted based on a key value.
  - Operations on the unsorted version are very similar to those of a singly linked list.
  - We limit our discussion to the sorted doubly linked list.
Doubly Linked Nodes

- The node storage class is similar to that of a singly linked list.

```python
class DListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```
Doubly Linked: Traversing

- Traversals can be in either order.
  - Forward is the same as with a singly linked list.
  - Reverse order is similar but the \texttt{prev} link is followed.

```python
def revTraversal( tail ):
    curNode = tail
    while curNode is not None :
        print( curNode.data )
        curNode = curNode.prev
```
Doubly Linked: Search

- Normal searching based on key value is the same as searching a singly linked list.
- Doubly linked lists provide an additional advantage.
  - We can search both forwards and backwards.
  - Use and maintain a probe reference.
Doubly Linked: Probing

- Use the probe reference for searching.
  - After a search, keep the reference where it left off.
  - On the next search, we can search either forwards or backwards based on the target value.

- empty list
- probe fall off
- probe forward
- probe backward
- extra tests: compare with first and last nodes in the list (good for large lists)
Doubly Linked: Probing

```python
# Make sure the list is not empty.
if head is None:
    return False

# If probe is null, initialize it to the first node.
elif probe is None:
    probe = head

# If the target comes before the probe node, we traverse backward.
if target < probe.data:
    while probe is not None and target <= probe.data:
        if target == probe.data:
            return True
        else:
            probe = probe.prev
    else:
        probe = probe.prev

else:  # otherwise traverse forward.
    while probe is not None and target >= probe.data:
        if target == probe.data:
            return True
        else:
            probe = probe.next

# If the target is not found in the list, return False.
return False
```
Doubly Linked: Insert

- Locate the position for the new node, then connect the links.
  - The first value larger than the new value.
  - There are three possible cases.
  - No need for a second temporary reference.
  - Connections must be made in a specific order.
Doubly Linked: Insert

- (1) Insert in the middle.
Doubly Linked: Insert

- Result after the insertion.
Doubly Linked: Insert

- (2) Insert at the front.
Doubly Linked: Insert

- (3) Insert at the end.
Doubly Linked: Insert

```python
# Add a new value given a head and tail reference
newnode = DListNode( value )
if head is None:  # empty list
    head = newnode
tail = head
elif value < head.data:  # insert before head
    newnode.next = head
    head.prev = newnode
    head = newnode
elif value > tail.data:  # insert after tail
    newnode.prev = tail
    tail.next = newnode
    tail = newnode
else:  # insert in the middle
    node = head
    while node is not None and node.data < value:
        node = node.next

    newnode.next = node
    newnode.prev = node.prev
    node.prev.next = newnode
    node.prev = newnode
```
Circular Linked List

- Another variation of the linked list in which the nodes form a continuous circle.
  - Allows for a complete traversal from any initial node.
  - Used with round-robin type applications.
  - The external reference can point to any node in the list. Common to reference “end” of the list.
Circular Linked List

- A circular linked list can also be doubly linked.

- We describe the operations for use with a sorted singly linked circular list.
Circular Linked: Traverse

- A traversal can start from any node, but it must visit every node.

```python
def traverse( listRef ):
    curNode = listRef
    done = listRef is None
    while not done :
        curNode = curNode.next
        print( curNode.data )
        done = curNode.data is listRef
```

- Flag the end of the traversal.
- Check the case with single node.
Circular Linked: Searching

- Searching a circular linked list is similar to the traversal operation.

```python
def searchCircularList(listRef, target):
curNode = listRef
done = listRef is None
while not done:
curNode = curNode.next
    if curNode.data == target:
        return True
    else:
        done = curNode is listRef or\
        curNode.data > target
return False:
```
Circular Linked: Inserting

- Adding nodes is very similar to that of the sorted singly linked list.
  - Unsorted list – common to add the node following the listRef.
  - Sorted list – new node is placed in proper position.
    - Can be divided into four cases.
Circular Linked: Inserting

- (1) Insert into an empty list.

```python
... if listRef is None :
    listRef = newNode
    newNode.next = newNode
```

![Diagram of list nodes and pointers.](image)
Circular Linked: Inserting

- (2) Insert at the "front" (one node past listRef)

```python
... 
if value < listRef.next.data :
    newNode.next = listRef.next
    listRef.next = newNode
```
Circular Linked: Inserting

- (3) Insert at the “end” (adjust listRef)

```python
... if value > listRef.data:
    newNode.next = listRef.next
    listRef.next = newNode
    listRef = newNode
```
Circular Linked: Inserting

- (4) Insert in the middle.
Circular Linked: Inserting

```python
newNode = ListNode( value )
if listRef is None :               # empty list
    listRef = newNode
    newNode.next = newNode
elif value < listRef.next.data :  # insert in front
    newNode.next = listRef.next
    listRef.next = newNode
elif value > listRef.data :       # insert in back
    newNode.next = listRef.next
    listRef.next = newNode
    listRef = newNode
else :                           # insert in the middle
    # Position the two pointers.
    predNode = None
    curNode = listRef
    done = listRef is None
    while not done :
        predNode = curNode
        curNode = curNode.next
        done = curNode is listRef or curNode.data > target

    # Adjust links to insert the node.
    newNode.next = curNode
    predNode.next = newNode
```
Multi-Linked Lists

- A linked list in which each node contains multiple link fields.
  - Used to create multiple chains within the same collection of nodes.
  - Each chain has its own head reference.

- The doubly linked list is a special case of the multi-linked list.
Multiple Chains

- Multiple keys and multiple lines are used to create chains through one set of nodes.
Multi-linked Nodes

- The storage class contains the data and one link field for each chain.

```python
class StudentMLListNode:
    def __init__(self, data):
        self.data = data
        self.nextById = None
        self.nextByName = None
```
Multi-Linked Operations

- Traversal and search
  - Can be performed on any chain.
  - Depends on the application.

- Adding nodes:
  - Create and initialize a single node.
  - Add the node to each chain.
Multi-Linked Example

- The chains do not all have to form complete lists.
- Common for at least one chain to contain all nodes in the list.