List ADT - Implementation Data Structures

- Big-O’s of List Implementations

<table>
<thead>
<tr>
<th>ADT Operations</th>
<th>Array</th>
<th>Singly-Linked List</th>
<th>Doubly-Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT (IN ORDER) (AT FRONT)</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>DELETE</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>FIND</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>FINDKTH</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>NEXT</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PREV</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>ISEMPTY</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Array NEXT and PREV assume that a current index is maintained as part of the list implementation.

- Array ISEMPTY assumes that the length of the list is maintained.

- INSERT (at current location) on a doubly-linked list is O(1). INSERT (at an arbitrary location) on a doubly-linked list is O(N).
Reversing a List

• If the list is implemented as an array...

• What is the Big-O of this algorithm?
  • There are n/2 swaps (n is the length of the list).
  • O(n)
Reversing a Linked List

- Given a singly-linked list, with List_p pointing to the head of the list (no header element, though this algorithm can easily be extended to use that implementation), assuming the list isn’t empty:

```c
curr = List_p;
next = List_p->next;
prev = NULL;
```

```plaintext
List_p → 1 → 2 → 3 → 4
```

```
prev
curr
next
```

```
while (next != NULL) {
    curr->next = prev;
    prev = curr;
    curr = next;
    next = curr->next;
}
```

```plaintext
After 1 iteration:
```

```plaintext
List_p → 1 → 2 → 3 → 4
```

```
prev
curr
next
```
Reversing a Linked List (cont’d)

curr->next = prev;
List_p = curr;

• The Big-O of this algorithm is O(n).

• What about Doubly-Linked Lists?
  • Just swap the PREV and NEXT pointers in each element. Also O(n).