CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Linear Data Structures

Department of Computer Science
University of Maryland, College Park
Linear Data Structures

- 1-to-1 relationship between elements
  - Each element has unique predecessor & successor
  - Results in total ordering over elements
  - For any two distinct elements x and y, either x comes before y or y comes before x
- Beginning of the list identified with first/head
- End of the list identified with last/tail
List Implementations

- Two basic implementation techniques for lists
  - Store elements in an array
  - Store as a linked list
    - Place each element in a separate object (node)
    - Node contains reference to other node(s)
    - Link nodes together

```java
class Node {
    Object data;
    Node next;
}
```

- Node head ➔ points to first node
Array vs. LinkedList Implementations

- Arrays
  - Advantages
    - Can efficiently access element at any position (O(1))
    - Efficient use of space (space just to hold reference to each element)
  - Disadvantages
    - Expensive to grow / shrink array
      - Can amortize cost (grow / shrink in spurts)
    - Expensive to insert / remove elements in middle (O(n))
    - Tricky to insert / remove elements at both ends
- LinkedList
  - Advantages
    - Can efficiently insert / remove elements anywhere
  - Disadvantages
    - Cannot efficiently access element at any position
      - Need to traverse list to find element (O(n))
    - Less efficient use of space
      - 1-2 additional references per element
  - See LinkedList code distribution
Linked List – Insert (After Cursor)

1. Original list & new element temp

```
before  I1    I2    I3
```

2. Modify temp.next → cursor.next
Linked List – Insert (After Cursor)

3. Modify `cursor.next → temp`

4. Modify `cursor → temp`
### Linked List – Delete (Cursor)

1. Find **before** such that **before.next** = **cursor**

```
   l1  →  l2  →  l3
     ▼        ▼
    before  cursor
```

2. Modify **before.next** → **cursor.next**

```
   l1  →  l2  →  l3
     ▼        ▼
    before  cursor
```
Linked List – Delete (Cursor)

3. Delete cursor

4. Modify cursor → before.next
Maintaining List Sorted

• One approach to maintain a linked list sorted with every insertion is
  • If the list is empty
    • Just make the element the first of the list (insertion is trivial)
  • Otherwise
    • Traverse the list until you find an element (B) larger than the one you want to insert (A)
    • Once you find B, insert A before B
    • If you don’t find B, A will become the last element of the list
Doubly Linked List

- Linked list where element has predecessor & successor

Structure
Class Node {
  Object data;
  Node next;
  Node previous;
}

Issues
- Easy to find preceding / succeeding elements
- Extra work to maintain links (for insert / delete)
- More storage per node
Doubly Linked List – Insertion

- Example

- Must update references in both predecessor and successor nodes
Restricted Abstractions

- Restricting the operations an abstraction supports can be a good thing
  - Efficiently supporting only a few operations efficiently is easier
  - If limited abstraction is sufficient, easier to reason about limited abstraction than a more general one
- Restricted list abstractions
  - Stack (aka LIFO queue)
  - Queue (aka FIFO queue)
  - Dequeue (aka double ended queue)
Stack

- Properties
  - Elements removed in opposite order of insertion
  - Last-in, First-out (LIFO)
- A restricted list where
  - Access only to elements at one end
  - Can add / remove elements only at one end
- Stack operations
  - Push = add element (to top)
  - Pop = remove element (from top)

(a) A three-element stack
(b) After a \texttt{pop}() operation
(c) After a \texttt{push}(W) operation
Stack Implementations

• Linked list
  • Add / remove from head of list

  top \rightarrow Z
  \ Y
  \ X

  (a) Logical view of the stack

  head \rightarrow Z \rightarrow Y \rightarrow X

  (b) Its linked list implementation

• Array
  • Increment / decrement Top pointer after push / pop

  X \ Y \ Z \ \ ...

  Top
Queue

- Properties
  - Elements removed in order of insertion
  - First-in, First-out (FIFO)
- A restricted list where
  - Access only to elements at beginning / end of list
  - Add elements only to end of list
  - Remove elements only from front of list
  - Alternatively, can add to front & remove from end
- Queue operations
  - Enqueue = add element (to back)
  - Dequeue = remove element (from front)
- Example

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Y</th>
<th>Z</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
<td>^</td>
</tr>
<tr>
<td>front</td>
<td>back</td>
<td>front</td>
<td>back</td>
<td>front</td>
<td>back</td>
</tr>
</tbody>
</table>

(a) Three-element queue  
(b) After deletion of X  
(c) After insertion of W
Queue Implementations

• Linked list
  • Add to tail (back) of list
  • Remove from head (front) of list

- Circular array
Queue – Circular Array Implementation

• Inherent problem for queue of size \( N \)
  • Only \( N \) possible (Front – Back) pointer locations
  • \( N+1 \) possible queue configurations
    • Queue with 0, 1, … \( N \) elements

• Solutions
  • Maintain additional state information
    • Use state to recognize empty / full queue
  • Examples
    • Record Size
    • Record QueueEmpty flag
  • Leave empty element in queue
  • Store marker in queue