Stack versus LinkedList mplementations

– static versus dynamic memory allocation

- **Array-based implementation of a data structure such as a list or stack**
  - restricts the number of items in the data structure.
  - is the array size is sufficient (can you predict in advance the number of items needed).
  - would you waste storage if the structure remains only partially full.
  - could resize the array with *new* – may still overspecify the size with many unused spaces.

Stack versus LinkedList mplementations

– static versus dynamic memory allocation

- **Reference-based implementation:**
  - allocate the memory dynamically with *new* only as much storage as is needed.
  - However, the order of the items in the data structure can affect the outcome.
    - In an array the position of the next item is ***implicit*** (i+1). In a linked list must use a reference – the primary difference between the two implementations
    - so the array does not have to store implicit information thus requiring less memory. Array-based implementation also provides direct access (position 4 is item 3) so access time is constant.
    - With linked list must traverse with *next* pointer – access for the ith node depends on i.

Stack versus LinkedList mplementations

– static versus dynamic memory allocation

- A list implementation with an array (the ADT list) requires the shifting of elements when you insert or delete from the list – say delete item 20 (i-1 shifts).
- No such shifts with a linked list reference-based implementation. Add and remove require the same effort – depends on the time to find the item.