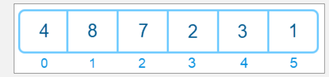
Data Structures

There are two main kinds of data structures –

- *contiguous structures* occupy consecutive memory locations
  - e.g. arrays
- *linked structures* consist of separate chunks of memory connected by references or pointers
  - e.g. linked lists, many implementations of trees, graphs

Array Refresher

array concepts



– insert/remove element involves shifting elements (order-preserving)
– dynamic array requires resizing when full

```
/**
 * Add the element at the specified index in the array, preserving the order
 * of the remaining elements.
 *
 * @param array
 *          the array
 * @param elt
 *          element to add
 * @param index
 *          index where to add the element
 */
public static void insert ( int[] array, int elt, int index ) {

}
```

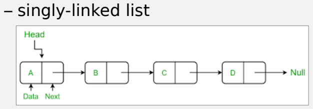
strategy – use examples!
- draw before and after pictures
- identify what changes
- get convenient references for those things
- update the values

be sure to consider several cases to make sure your solution works in general

be sure to consider typical special cases such as the first and last things, empty or one-element array

Linked List Refresher

linked list concepts
– singly-linked list



– insert/remove node involves re-linking rather than shifting or changing elements

```
/**
 * Remove the specified node from the list.
 *
 * @param head
 *          head of the list
 * @param todel
 *          node to remove
 * @return the head of the list after the removal
 */
public static ListNode remove ( ListNode head, ListNode todel ) {
  return null;
}
```

```
public class ListNode {

  // what goes here?

}
```

strategy – use examples!
- draw before and after pictures
- identify what changes
- get convenient references for those things
- update the values

be sure to consider several cases to make sure your solution works in general

be sure to consider typical special cases such as the first and last things, empty or one-element list, `null` values

Linked List Refresher

```
/**
 * Remove the node todel from the list.
 *
 * @param head
 *          head of the list
 * @param todel
 *          node to remove
 * @return the head of the list after the removal
 */
public ListNode remove ( ListNode head, ListNode todel ) {

  if ( todel == head ) {
    // removing the first node - only the head changes
    return head.getNext();
  }

  // removing not the first node
  ListNode before; // the node before todel
  for ( before = head ; before.getNext() != todel ; before =
      before.getNext() ) {}

  ListNode after = todel.getNext(); // the node after todel

  // do the removal
  before.setNext(after);

  return head;
}
```

```
/**
 * Singly-linked list node with int elements.
 */
public class ListNode {

  private int elt_;
  private ListNode next_;

  public ListNode ( int elt ) {

  public ListNode ( int elt, ListNode next ) {

  public int getElt () {

  public void setElt ( int elt ) {

  public ListNode getNext () {

  public void setNext ( ListNode next ) {

}
```

common problems – implementing linked list operations

– recursive rather than iterative (loop) solution (there's nothing wrong with using recursion, just that loops are generally easier to understand)

– attempting to use the size of the list

– doubly-linked nodes instead of singly-linked

– working with elements instead of nodes (moving elements instead of re-linking nodes, comparing elements rather than nodes)

– missing or incorrectly handling special cases (potential special cases: empty list, one-node list, `todel` null, `todel` last, `todel` first, `todel` not actually in the list)

– mixing up `.equals` and `==`

– creating a new node object instead of only a new node variable

style considerations

– loop body shouldn't contain steps only done at the very beginning or the very end

```
/**
 * Remove the node todel from the list.
 *
 * @param head head of the list
 * @param todel node to remove
 * @return the head of the list after the removal
 */
public ListNode remove ( ListNode head, ListNode todel ) { ... }
```

what if this was a doubly-linked list?

```
/**
 * Singly-linked list node with int elements.
 */
public class ListNode {

  private int elt_;
  private ListNode next_;

  public ListNode ( int elt ) {

  public ListNode ( int elt, ListNode next ) {

  public int getElt () {

  public void setElt ( int elt ) {

  public ListNode getNext () {

  public void setNext ( ListNode next ) {

}
```


Characteristics and Tradeoffs

arrays	linked structures
access time is constant given the index (efficient random access)	access time depends on position relative to the beginning (inefficient random access)
space efficiency – no overhead (links, end-of-record markers) beyond the data elts themselves, though to efficiently handle resizing up to O(n) empty slots are allowed	overhead of at least one pointer per data value
memory locality – iterating through involves access to nearby memory blocks which can be efficiently loaded into a cache	no memory locality
fixed size – must resize or waste space dynamic arrays support resizing (when doubled in size) in O(1) amortized time and still O(n) space, but O(n) worst case insert and ~2x constant factors	dynamic size, no resizing needed growing is O(1) when the insert position is known
insert, remove other than at the end requires shifting (O(n))	insert, remove at any position O(1), given a node pointer

Basic Implementation of Containers

How to use the data structure to realize the ADT operations?

- decide how container elements will be arranged in the data structure
  - use linear order of array or linked list to store the linear order of the container
- options: forward or reverse?
  - beginning of array / head of linked list can match beginning / top / front of container
  - end of array / head of linked list can match beginning / top / front of container
- outline how each operation is carried out in terms of these choices
  - give enough detail so correctness and running time can be assessed, but no more

Basic Implementation of Containers

How to use the data structure to realize the ADT operations?

- decide how container elements will be arranged in the data structure
  - use linear order of array or linked list to store the linear order of the container
- options: forward or reverse?
  - beginning of array / head of linked list can match beginning / top / front of container
  - end of array / head of linked list can match beginning / top / front of container

Vector / List / Sequence	classic array vs linked list tradeoffs • insert/remove not at the end requires shifting in the array (O(n)), but access by rank (index) is O(r) for linked list • dynamic array has overhead in time (resizing) and space (empty slots), linked list has overhead in space (pointers)
Stack	O(1) push, pop with array and linked list • top of stack = end of array, head of linked list choice of array vs linked list is largely determined by whether there is an upper bound on the size of the stack that is known in advance (static array) or not (dynamic array or linked list – time vs space overhead tradeoff)
Queue	O(1) enqueue or dequeue, O(n) for other with array, linked list

Basic Implementation of Containers

Vector / List / Sequence	classic array vs linked list tradeoffs • insert/remove not at the end requires shifting in the array (O(n)), but access by rank (index) is O(r) for linked list • dynamic array has overhead in time (resizing) and space (empty slots), linked list has overhead in space (pointers)
Stack	O(1) push, pop with array and linked list • top of stack = end of array, head of linked list choice of array vs linked list is largely determined by whether there is an upper bound on the size of the stack that is known in advance (static array) or not (dynamic array or linked list – time vs space overhead tradeoff)
Queue	O(1) enqueue or dequeue, O(n) for other with array, linked list

Can we do better?

- Vector/List/Sequence – tradeoff is due to the nature of the data structures (random access vs sequential access)
- Stack – can't beat O(1)
- Queue – …