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

The book gives C code for recursive implementatio
linked list search, insert, and delete operations. Fill
body of the function below with an iterative implen
of deletion.

```
/**
* 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 ) { .
```

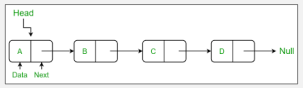
Use the following definition for `ListNode`:

```
public class ListNode {
  public ListNode ( int value ) { ... }
  public ListNode ( int value, ListNode next ) { ... }

  public int getValue () { ... }
  public void setValue ( int value ) { ... }

  public ListNode getNext() { ... }
  public void setNext ( ListNode next ) { ... }
}
```

linked list concepts
– singly-linked list



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

common problems
– 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

The book gives C code for recursive implementat
linked list search, insert, and delete operations. F
body of the function below with an iterative impl
of deletion.

```
/**
* 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 ) { ... }
```

Use the following definition for `ListNode`:

```
public class ListNode {
  public ListNode ( int value ) { ... }
  public ListNode ( int value, ListNode next ) { ... }

  public int getValue () { ... }
  public void setValue ( int value ) { ... }

  public ListNode getNext() { ... }
  public void setNext ( ListNode next ) { ... }
}
```

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

```
/**
 * 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;
}
```

```
/**
* 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?

Characteristics and Tradeoffs

arrays	linked structures
access – constant time 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	no overflow, 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

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 – …

Improving an Implementation – Queue

Consider the linked list implementation with the head of the queue at the beginning of the list.

- enqueue(x) is O(n) – inserting at the end of the list requires finding the last node
- dequeue() is O(1) – removing from head just involves updating pointers

enqueue is slow because we have to find the tail of the list.

Can we store a tail pointer instead?

- enqueue – O(1) to locate the node before the insertion point (current tail), O(1) to create new node and link to current tail, O(1) to update current tail to new node
- dequeue is not affected (unless the last element is removed – tail becomes `null`)

→ O(1) enqueue and dequeue using a linked list with a tail pointer

Improving an Implementation – Queue

Consider the array implementation with the head of the queue at the beginning of the array.

- enqueue(x) is O(1) – insert at the end of the array
- dequeue() is O(n) – removing from head of array requires shifting

Do we have to shift?

- we shift to keep the head of the queue at 0 and the tail position based on the size

Can we store the head and tail positions instead?

- O(1) to locate head/tail
- O(1) to update head/tail – new value is next position
  - "next position" at the end of the array wraps around to 0

→ O(1) enqueue and dequeue using a circular array

Doing Better

- if the slowness is because of having to find or compute something, can you store it instead?
  - must consider the cost of updating the stored info
  -
- if the slowness is the result of not storing something, can you store it instead?
  - must consider the cost of updating the additional info stored