Iterator ADT

Dynamic Array and Linked List
Goals

• Why do we need iterators?
• Iterator ADT
• Linked List and Dynamic Array Iterators
Iterating through a Linked List

Write a code segment to print the elements of our LinkedListDeque implementation.
Problem: How do you provide a user of a container access to the elements, without exposing the inner structure?

Think of two developers: one writing the container (that’s you!!!), the other using the container (that’s someone using your library to build an application where they need, for example, a stack implementation)
Traversing a Container – as Developer

For example, within the Linked List container you (the developer) wrote a loop such as the following:

```c
struct LinkedList *list;
struct Link *l;
...
/* Initialize list. */
l = list->frontSentinel->next;
while(l!=list->backSentinel){
    ...do something...
    l=l->next;
}
```

This is fine within the container library code itself, but we don’t want users of the container library to have to know about links...or worse yet, be able to manipulate links!
• Chapter 5: Hide the implementation details behind a simple and easy to remember interface (ie. abstraction mechanism)

• Users should *not* know about links, arrays, size, capacity, etc.

• Users should know and use the public abstractions: push, pop, contains, remove, etc.
How do we abstract away loops...

```c
for(i = 0; i < sizeDynArr; i++)
    val = getDynArr(i);

while( cur != backSent)
    do something
    cur = cur -> next

do {
    something
    cur = cur->next;
}while (cur != backSent
```
Solution: define an interface that provides methods for writing loops

```c
int hasNextIter(struct Iter *itr);
TYPE nextIter(struct Iter *itr);
void removeIter(struct Iter *itr);
void changeIter(struct Iter *itr, TYPE val);
void addIter(struct Iter *itr, TYPE val);
```
Iterator: Typical Usage

```
TYPE cur; /* current collection val */
struct linkedList *list;
linkedListIterator *itr;
list = createLinkedListList(...) 
itr = createLinkedListListIter(list)

while (hasNextListIter(itr)) {
  cur = nextListIter(itr);
  if (cur ...)
    removeListIter(itr);
}
```
• Function **next** and **hasNext** must be interleaved
• Call **remove** after **next**
• Cannot call **remove** twice in a row without a calling **hasNext**
struct linkedListIterator {
    struct linkedList *lst;
    struct dlink *currentLink;
}

struct linkedListIterator *createLinkedListIterator ( struct linkedList *lst) {
    struct linkedListIterator *itr;
    itr = malloc(sizeof(struct linkedListIterator));
    itr->lst = lst;
    itr->currentLink = lst->frontSentinel;
}
After Initialization
Strategy

- **HasNext**
  - Returns T (F) to the user if there is (is not) another value remaining to be enumerated
  - Updates current

- **Next**
  - Return the next value to be enumerated

- **Remove**
  - Removes the value (and link) that was *last returned* by call to Next()
while (hasNextListIter(itr)) {
    curVal = nextListIter(itr);
    ...
}
while (hasNextListIter(itr)) {
    curVal = nextListIter(itr);
    ...
}
while (hasNextListIter(itr)) {
    curVal = nextListIter(itr);
    ...
}
while (hasNextListIter(itr)) {
    curVal = nextListIter(itr);
    ...
}
After Two Iterations – then Remove

Remove the last value enumerated (10)
Where should current be after the removal?

while (hasNextListIter(itr)) {
    curVal = nextListIter(itr);
    if (curVal == 10)
        removeListIter(itr);
}
After Two Iterations – then Remove

Remove the last value enumerated (10)
Where should current be after the removal?
Your Turn

Worksheet#24 Linked List Iterator