Iterator ADT
Dynamic Array and Linked List
Goals

• Why do we need iterators?
• Iterator ADT
• Linked List and Dynamic Array Iterators
• 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)
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 class itself, but we don’t want users of the container 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: push, pop, contains, remove, etc.
Iterators to the Rescue

• So, how do we allow them to loop through the data without manipulating links?
• Provide a “facilitator object”
• Maintain encapsulation
  – Hide the details away from the user. Allow them to work at an abstract level.
• Also helps with code maintenance: a loop is essentially the same for a user whether using a LinkedList or DynamicArray
Solution: define an interface that provides methods for writing loops

```c
void createIter(container *l);
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

```c
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);
}
```
Simplifying Assumptions

- Function `next` and `hasNext` are 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->current = lst->frontSentinel;
}
After Initialization
Strategy

• HasNext
  – Returns T/F to the user if a value does/does not remain 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()
After Initialization
After One Iteration: hasNext, next
After Two Iterations

- list
- ltr
- current
- backSent
- frontSent
- prev
- next
- front

List

Nodes:
- 5
- 10
- 15
We’ve enumerated all elements
Subsequent calls to HasNext will evaluate to false
Remove the last value enumerated (10). Where should current be after the removal?
Remove the last value enumerated (10)
Where should current be after the removal?
Your Turn

Worksheet#24 Linked List Iterator