CS261 – Data Structures

Iterator ADT

Dynamic Array and Linked List

Goals

- Why do we need iterators?
- Iterator ADT
- Linked List and Dynamic Array Iterators

Iterator Concept

- 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)

Encapsulation

- 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.

Two Developers

Developer Sees…	End-User Sees…
a[size] = value	addLast(value)
return a[size]	getLast()

Traversing a Container

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

```
struct LinkedList *list;
struct Link *l;
...    /* Initialize list. */
for (l=list->head; l!=null; l=l->next)
or
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

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.

Iterator ADT

Solution: define an interface that provides methods for writing loops

```
void    initListIter(struct LinkedList *l,
                     struct ListIter *itr);

int  hasNextListIter(struct ListIter *itr);

TYPE    nextListIter(struct ListIter *itr);

void  removeListIter(struct ListIter *itr);

void  changeListIter(struct ListIter *itr,
                     TYPE val);

Void     addListIter(struct ListIter *itr,
                     TYPE val);
```

Iterator: Typical Usage

```
TYPE cur; /* current collection val */
Struct LinkedList *list;
Iterator *itr;
list = createList(…)
itr = createIter(list)

while (hasNextListIter(itr)) {
  cur = nextListIter(itr);
  if (cur ...) removeListIter(itr);
}
```

Information Hiding

- Notice that the iterator loop says nothing about the inner workings of the container
- The inner structure of the container is effectively encapsulated → the information is hidden

```
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`**

Iterators & Object Oriented Programming

- Iterators are common in OOP languages, where you have polymorphism, interfaces, etc
- But idea can be used in any language
- Very intuitive and easy to understand interface, easy to adapt

Your Turn

Worksheet#24 Linked List Iterator