Iterators

THE PURPOSE AND PRACTICALITY OF ITERATORS

Encapsulation

```
struct linked_list {
    struct node* first;
    struct node* last;
    size_t size;
} linked_list;

struct node {
    struct node* prev;
    struct node* next;
    value_t value;
};
```

- Members are encapsulated.
  - *Private members are not a C language feature.*
- Access/mutation controlled by interface.
- Linked lists are opaque to programmers: only behavior is important, not implementation.
- Flexible implementation, consistent interface.

Iteration

HOW DO WE ITERATE OVER A LINKED LIST?

Iteration – Traditional Method

```c
node* n = list->first;

while (n != NULL) {
        value_t value = n->value;
        // use value
        n = n->next;
}
```

- Breaks encapsulation.
- Efficient, O(n) runtime.
- Allows for in-place insertion and removal during iteration.
  - *Provided the current element being referred to is not being erased.*

Iteration – Through Interface

```
for (size_t idx = 0; idx < size(list); idx++) {
        value_t value = get(list, idx);
        // use value
}
```

- Does not break encapsulation.
- Inefficient, O(n^2) runtime.
- Does not allow for in-place insertion and removal during iteration.
    - *Adjustments (at the cost of performance) can be made to allow for this.*

Resolution

Extend the interface to support iteration.

- Introduce **for_each**?
  - Efficient but not flexible and requires a callback.
- Caching of **get/set** to optimize access?
  - Introduces state: excess overhead and increased complexity of our interface.
  - Still not flexible.
- Iterators?
  - Efficient, flexible, stateless, and does not break encapsulation.

Iterators

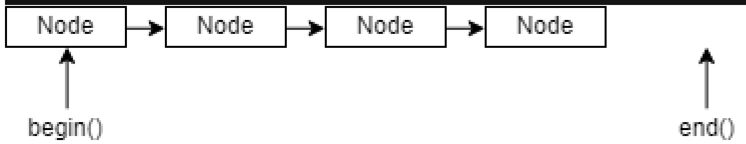
```c
typedef struct node* iter_t;
```

- **Opaque**: a programmer does not need to understand implementation.
- **Flexible:** allow for traversal bidirectionally and with a given number of advancements.
- **Efficient:** advancing the iterator does not depend on the number of elements.
- **Useful:** access, insert, and remove elements anywhere.

Iterators – begin/end

```c
iter_t linked_list_begin(const linked_list* list);

iter_t linked_list_end(const linked_list* list);
```



- The **begin** iterator is associated with the first node.
- The **end** iterator is associated with one after the last node.
- By associating the **end** iterator as such, **insert** will be able to insert a node anywhere in the linked list (before any given iterator).
  - *Insertion **anywhere** is not possible otherwise.*

Iterators - advance

```
iter_t linked_list_advance(const linked_list* list, iter_t iter, int i);
```

- The iterator can be advanced by **i steps** without the need to break encapsulation.
- The number of steps can be negative, referring to advancement towards **begin()**.
- The iterator returned can be in the range **[begin(), end()]**.

- **Example:** *advance(list, begin(list), 1)* will return an iterator to the ***second*** element in the list.
- **Example:** *advance(list, end(list), -1)* will return an iterator to the ***last*** element in the list.

Iteration – Through Iterators

```
for (iter_t iter = begin(list); iter != end(list); advance(list, iter, 1)) {
        value_t value = fetch(iter);
        // use value
}
```

- Efficient, O(n) runtime (as **i** is always 1).
- Encapsulation not broken.
- Allows for in-place insertion and removal during iteration.
  - *Provided the current element being referred to is not being erased.*

Iterators – insert/erase

- Efficient, O(1) insertion/removal time from anywhere.
- Only erased iterators become invalid.
- Linked lists are now more versatile.