CSE 373: Lists

Chapter 3

What is a List?

List:
List Components

- Lists are composed of
  - values
    - of arbitrary, but fixed type (Object)
  - at positions
    - some notion of placement/order within a list
    - type not necessarily known by user
      (ListItr<Object>)
    - type depends on implementation

List Operations

Iteration operations (List methods):
ListItr<Object> first();
ListItr<Object> kth(int);
ListItr<Object> last();

Iteration operations (ListItr methods):
Object retrieve();
void advance();
bool IsLast();
bool isPastEnd();
void previous();
bool IsFirst();
bool isPastStart();
**List Operations (cont’d)**

Main operations (*List* methods):

- `ListItr<Object> find(Object);`
- `void insert(Object, ListItr<Object>);`
- `void add(Object, ListItr<Object>);`
- `void remove(Object);`
- `bool isEmpty();`

Creation/Deletion (*List* methods):

- `List();`
- `~List();`
- `void makeEmpty();`

**Array-based List Implementation**

Store data in a normal C array:

```cpp
template <class Object>
class List {
    private:

    }
```

How would *ListItr<Object>* be implemented?
Array-based Insertion/Deletion

L.insert(5, 3);

L.remove(5);

Evaluating Lists

What’s the worst-case performance of...

Array-based

Other advantages/disadvantages?
**Linked List Implementation**

Store data in dynamically allocated nodes:

```cpp
template <class Object>
class ListNode {
private:
    Object data;
    ListNode *next;
};
```

How would `ListItr<Object>` be defined?

---

**Linked Insertion/Deletion**

```cpp
L.insert(3);
```

![Diagram showing insertion of 3 between 2 and 5]

```cpp
L.remove(3);
```

![Diagram showing removal of 3 between 2 and 5]
Coding Tips for Lists

- Implementation is conceptually straightforward, but it’s easy to make mistakes
- Testing strategy
  - “normal” case (as in pictures)
  - boundary cases:
    - empty list (full list?)
    - first element in list
    - last element in list
  - illegal cases

Design Decision: Header Node

VS.
Design Decision: Doubly-Linked

```
2 ---> 3 ---> 5 ---> 7
```

vs.

```
2 <-> 3 <-> 5 <-> 7
```

Design: Iterative vs. Recursive

Some list operations (e.g., `find()`) have obvious recursive implementations:

```cpp
ListItr<Object> find(Object val) {
    return findHelp(L.first(), val);
}
ListItr<Object> findHelp(ListItr<Object> pos, Object val) {
    if (pos.retrieve() == val) {
        return pos;
    } else if (pos.isFastEnd()) {
        return NULL;
    } else {
        pos.advance();
        return findHelp(pos, val);
    }
}
```

- Is this a reasonable use of recursion? A good use?
Applications

- Everything
  - class list
  - compilers: list of functions in a program, statements in a function
  - graphics: list of triangles to be drawn to the screen
  - operating systems: list of programs running

Reconsidering Array-Based Lists

- The book implies that the maximum size must be known in advance
- This isn’t technically true:
  - allocate an initial (default) size
  - if we run out of space:
    - allocate a larger array
    - copy data values from original to new array
    - delete original array
    - swap pointers so that new array is used