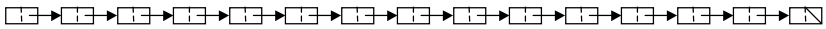
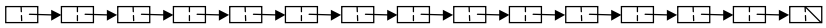


CSE 373: Lists

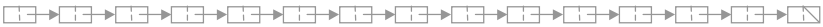
Chapter 3



What is a List?

List:

List Components



- Lists are composed of
  - *values*
    - of arbitrary, but fixed type (`ElementType`)
  - at *positions*
    - some notion of placement/order within a list
    - type not necessarily known by user (`Position`)
    - type depends on implementation

List Operations

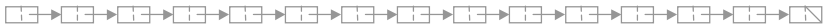
Main operations:

```
void Insert(List, Position, ElementType);
void Delete(List, ElementType);
Position Find(List, ElementType);
ElementType Retrieve(List, Position);
ElementType FindKth(List, int);
```

Creation/Deletion:

```
List NewList(List);
List MakeEmpty(List);
void DeleteList(List);
```

List Operations (cont'd)



Iteration operations:

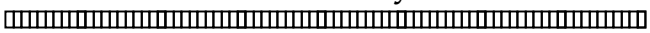
```
int IsEmpty(List);
Position First(List);
Position Next(List, Position);
int IsLast(List, Position);
Position Last(List);
Position Previous(List, Position);
int IsFirst(List, Position);
```

Questionable operations:

```
Position FindPrevious(List, ElementType);
Position Header(List);
```

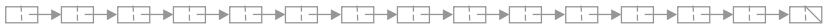

Array-based List Implementation

Store data in a normal C array:

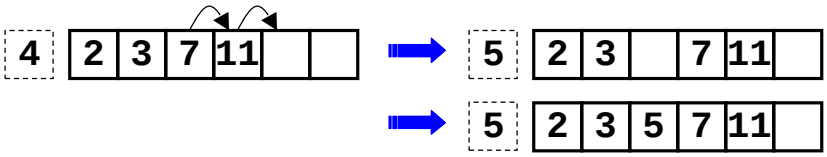


What is the type of **`Position`**?

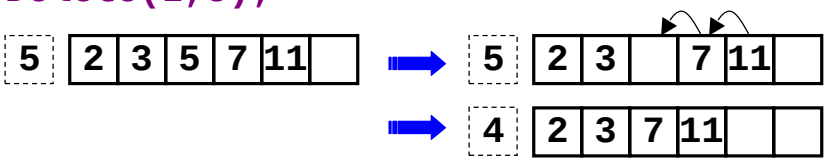
Array-based Insertion/Deletion



```
Insert(L,3,5);
```




```
Delete(L,5);
```



Evaluating Lists

What's the worst-case performance of…

Array-based

```
void Insert()
void Delete()
Position Find()
ElementType Retrieve()
ElementType FindKth()
```

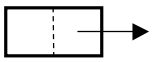
Other advantages?

Disadvantages?

Linked List Implementation

Store data in dynamically allocated nodes:

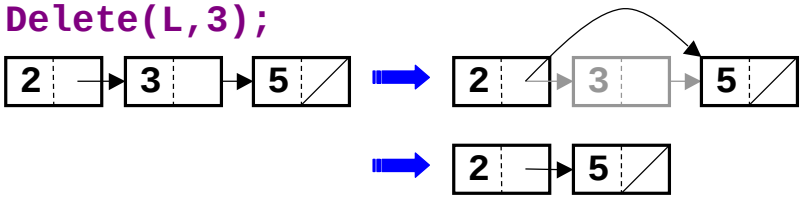
```
typedef struct _Node {
  ElementType data;
  struct _Node *next;
} Node;

typedef Node *Position;
typedef Node *List;
```



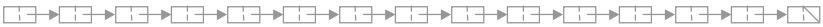
Linked Insertion/Deletion

```
Insert(L,,3);
```

```
Delete(L,3);
```

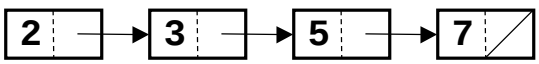


Coding Tips for Lists

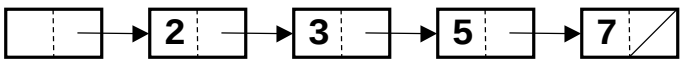


- Implementation is conceptually straightforward, but it's easy to make mistakes
- Testing strategy
  - "normal" case (as in pictures)
  - *boundary cases:*
    - empty list (full list?)
    - first element in list
    - last element in list
  - illegal cases

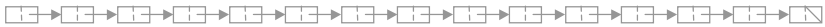
Design Decision: Header Node

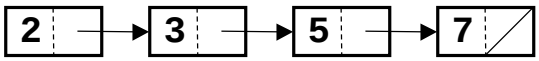


vs.

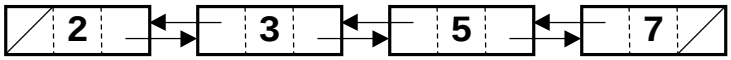


Design Decision: Doubly-Linked





vs.



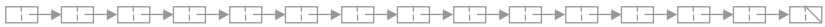
Design: Iterative *vs.* Recursive

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

```
Position Find(List L, ElementType val) {
  return FindHelp(First(L),val);
}
Position FindHelp(Position P,ElementType val){
  if (P->data == val) {
    return P;
  } else {
    return FindHelp(P->next,val);
  }
}
```

– Is this a good use of recursion?

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:

```
realloc() -- resize a memory area returned
             by malloc()
```

- Idea: **`realloc()`** array as we need more (or fewer) elements