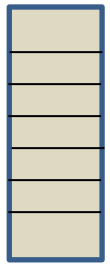
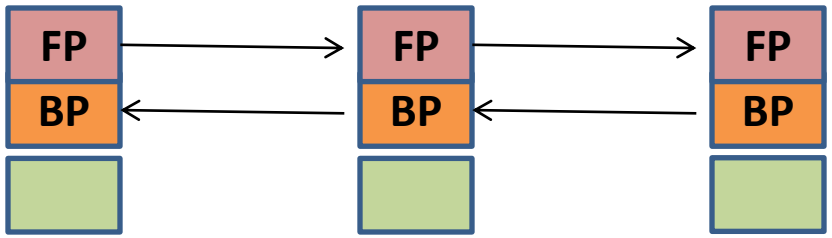
Linked Lists

Data Collection Structures

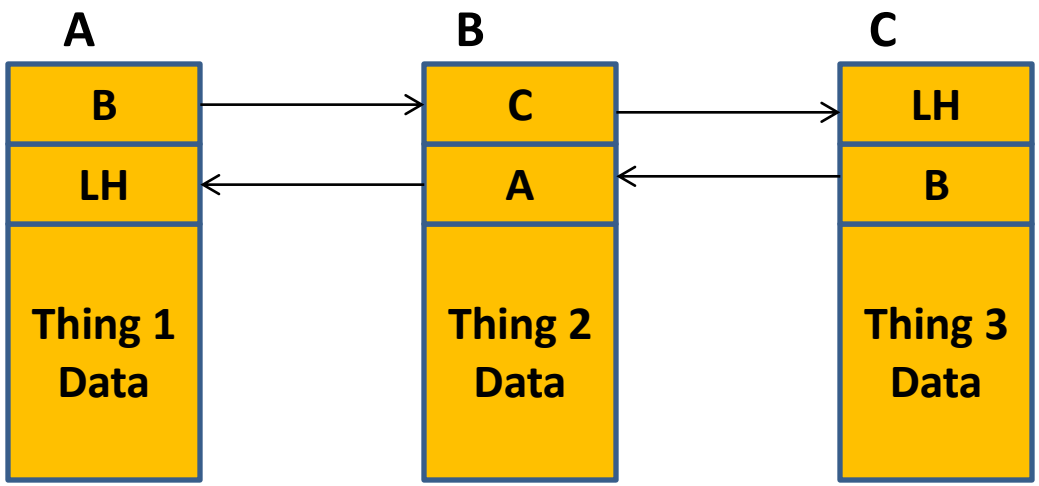
- Enhanced Arrays
  - Elements consecutively stored
  - Elements accessible by numeric key (Ex: a[5])

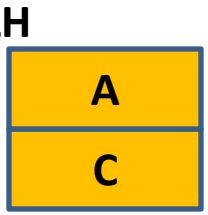


- **Linked Lists**
  - Elements “daisy chained” together (hooks to neighbors)
  - Access only by “walking the chain
  - Why would anyone want to collect elements this way?



Linked List: Implementation Issues






```
Class Thing
{
   private:
          Thing * fp;
          Thing *bp;
```

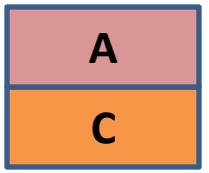
// ?? Object must not assume its collection class!

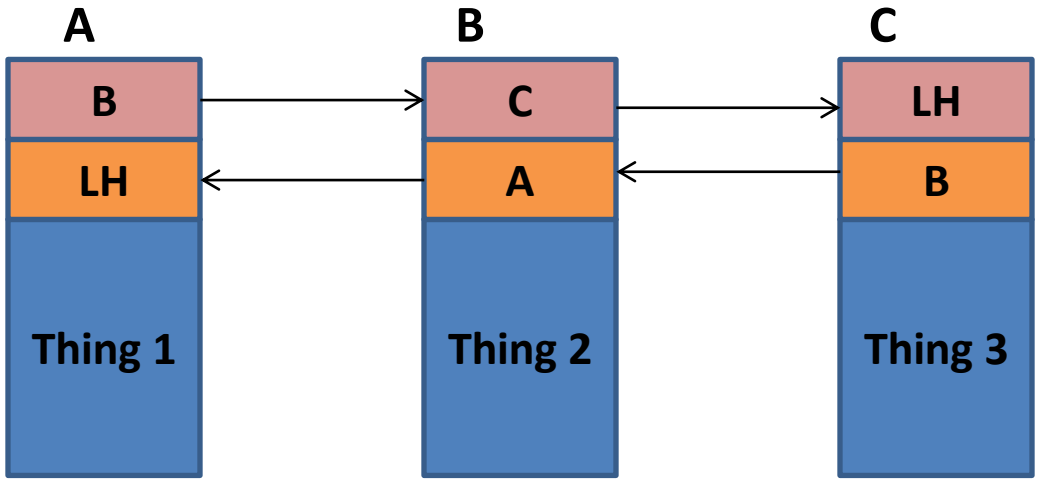
```
public:
        Thing *getNext();
        Thing *getPrev();
        bool insertNext (Thing *);
        bool insertPrevious (Thing *);
```

// ?? Who "*remembers*" the current position in the linked list to support these "next / previous" functions?

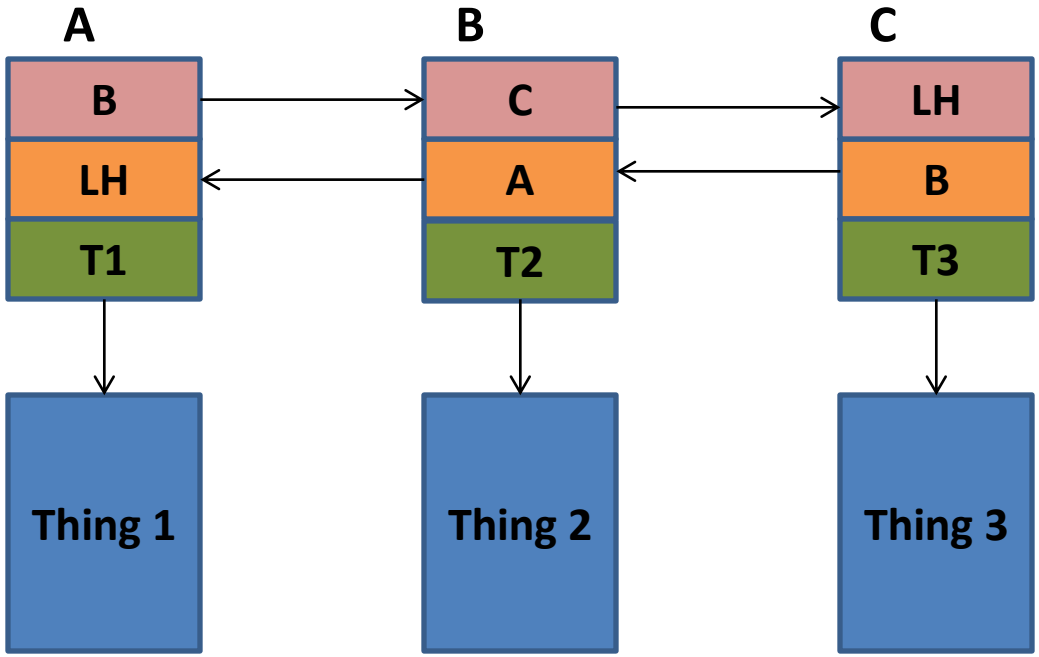
The Client? The List? The Element?

Linked List: The Internal "Node"






```
Class Thing  {
  private:
          Thing  *fp;
          Thing *bp;

// Object must not assume
its collection class!
```



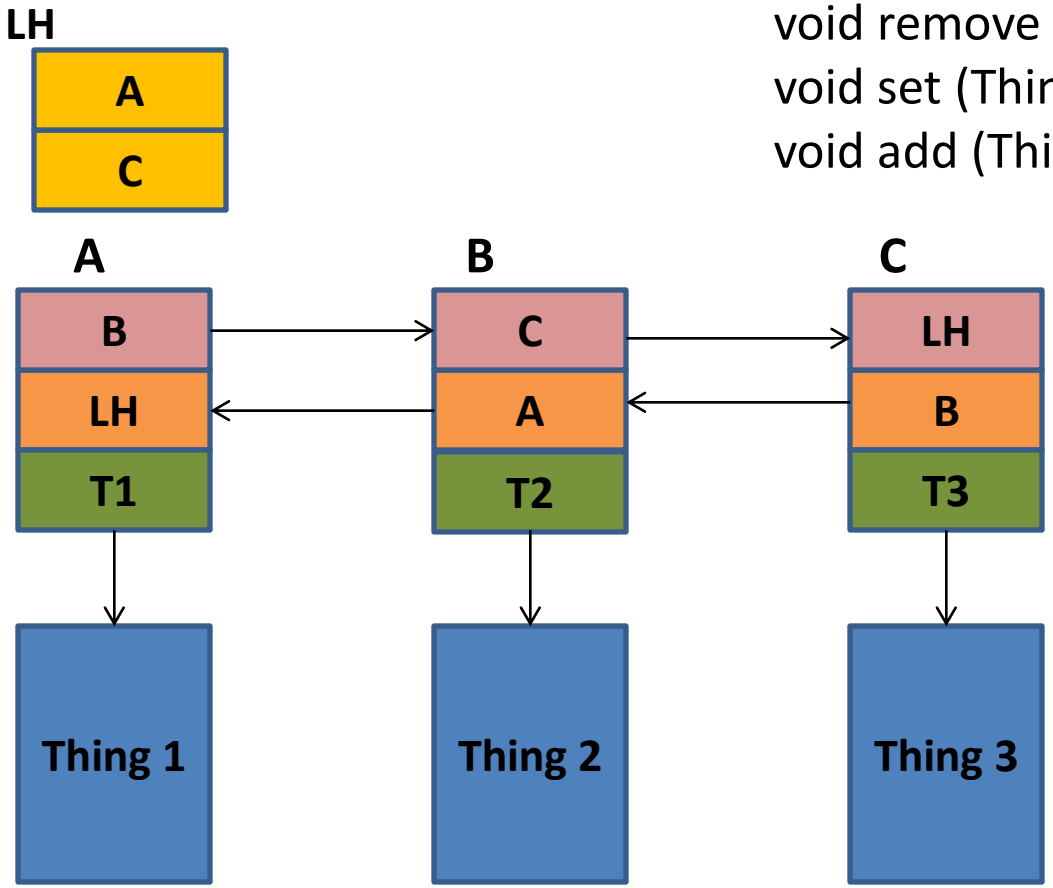

```
Class ThingNode
{  // Internal List class to support
   // a Thing object in a Linked List
   private:
          ThingNode  *fp;
          ThingNode  *bp;
           Thing * tp;
```

➔ Thing collected in a Linked List but doesn't know it!

Linked List: The External "Iterator"

```
Class ThingLinkedList  {
  public:
     bool isEmpty();
     ThingIterator getIterator();
```

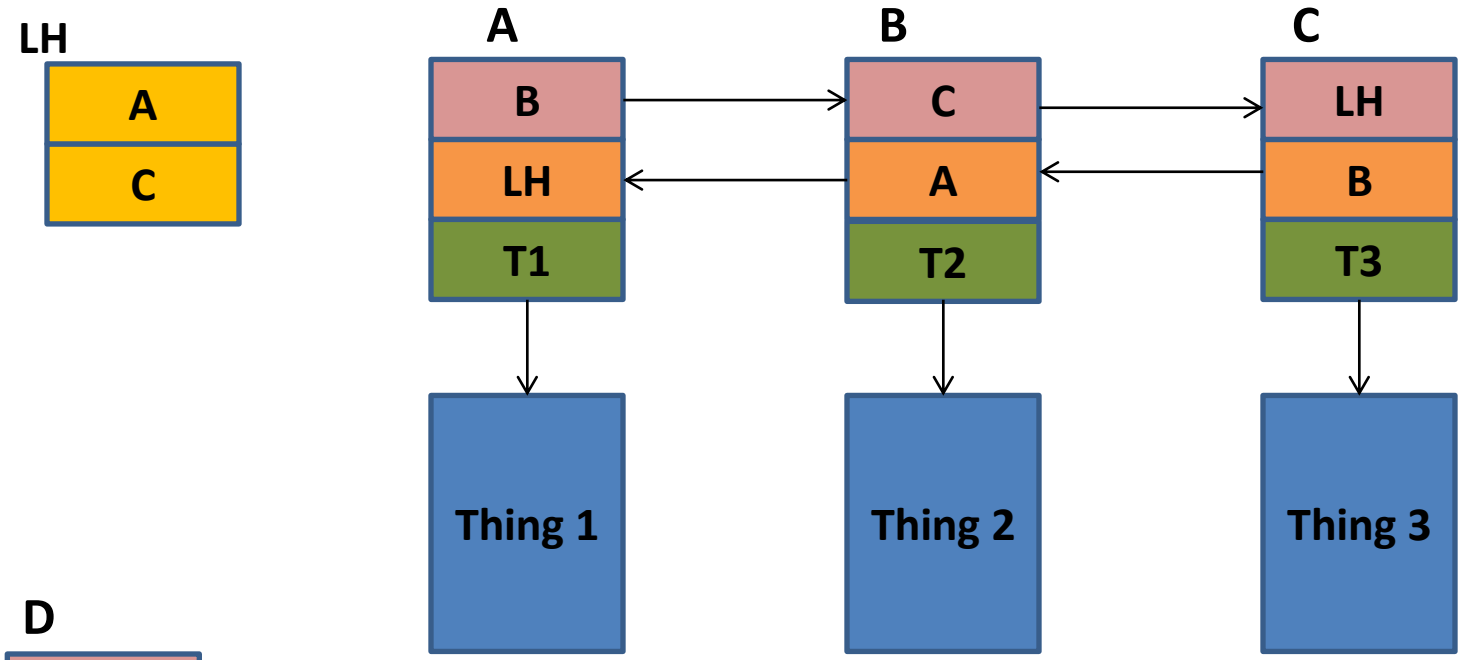
```
Class ThingIterator
{ // Works with LinkedList to "walk" element Nodes
   public:
    bool hasNext ();  Thing& getNext();
     bool hasPrevious (); Thing& getPrevious();
    void remove ();  // Delete current element
    void set (Thing&); // Replace current element
    void add (Thing&);   // Insert element "here"
```

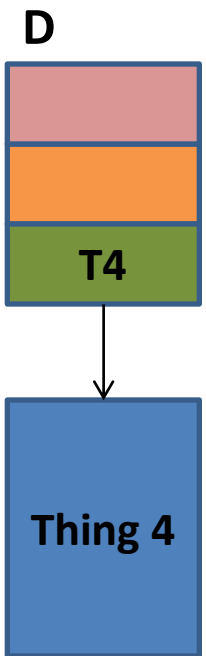



```
Class ThingNode
{  // Internal List class to support
   // a Thing object in a Linked List
   private:
              ThingListEntry * fp;
              ThingListEntry *bp;
               Thing * tp;
                    .
                    .
```

➔ Thing collected in a Linked List but doesn't know it!

Linked List: Insertion



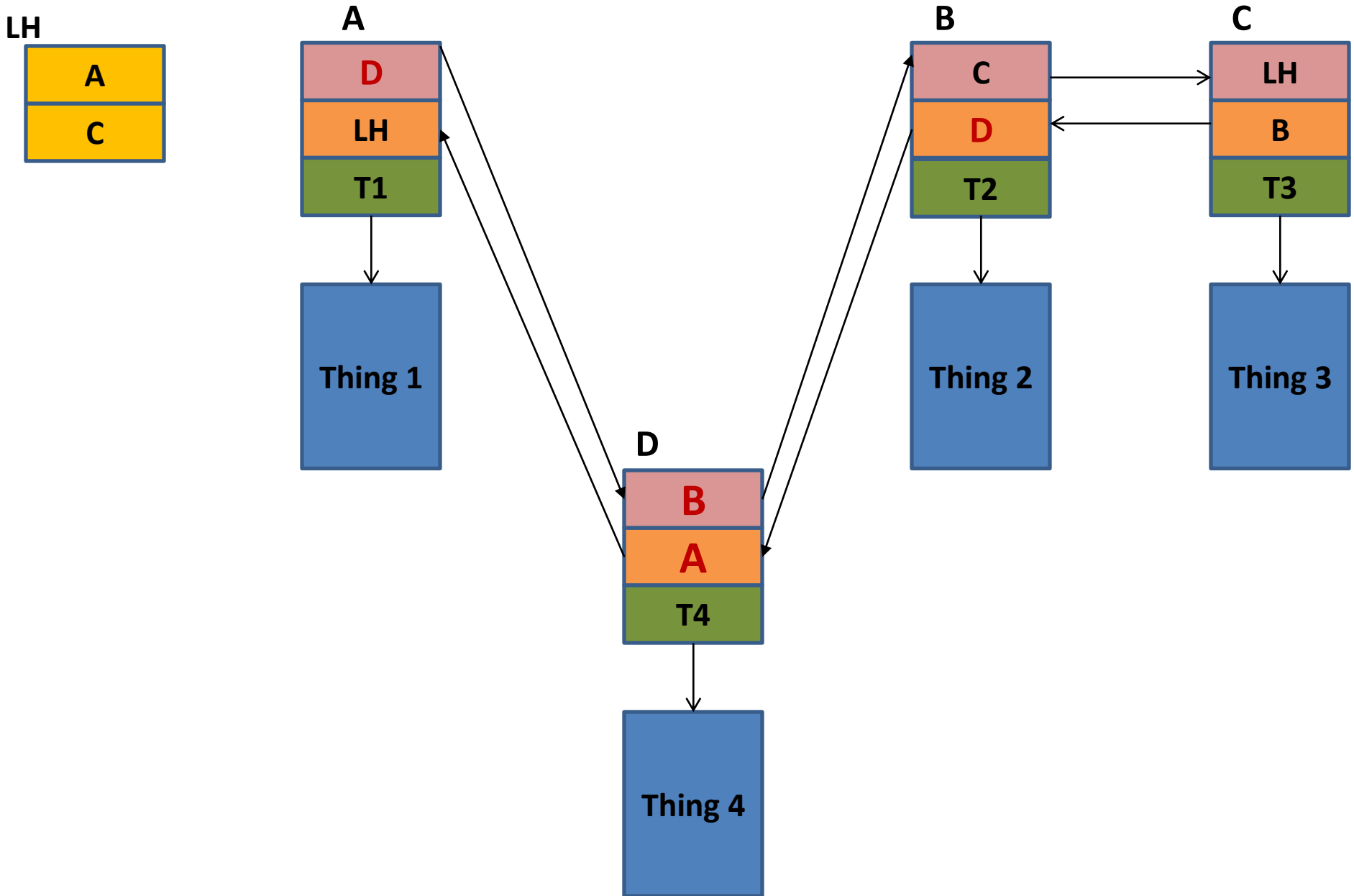


Insert Thing 4 between Thing 1 and 2: → Adjust FP / BPs in adjacent nodes

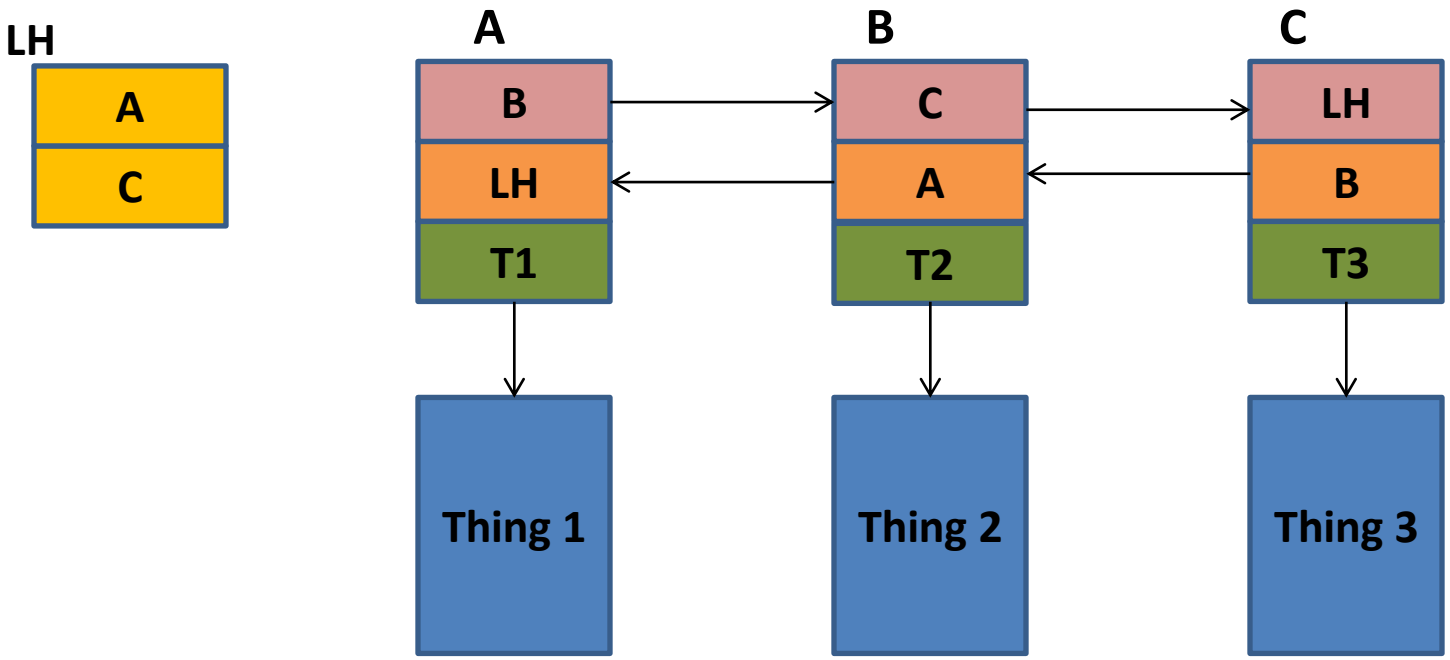
1. Set D node's FP to B and its BP to A. It is ready to be inserted.
2. To actually insert it:
   Set A's FP to D. // D now comes "after" A
   Set B's BP to D // D now comes "before" B

No matter how large the list, **this is all you have to do**.
(Extended Array: Everything past insertion must be moved down 1 slot)

Linked List: Element Insertion



Linked List: Element Removal

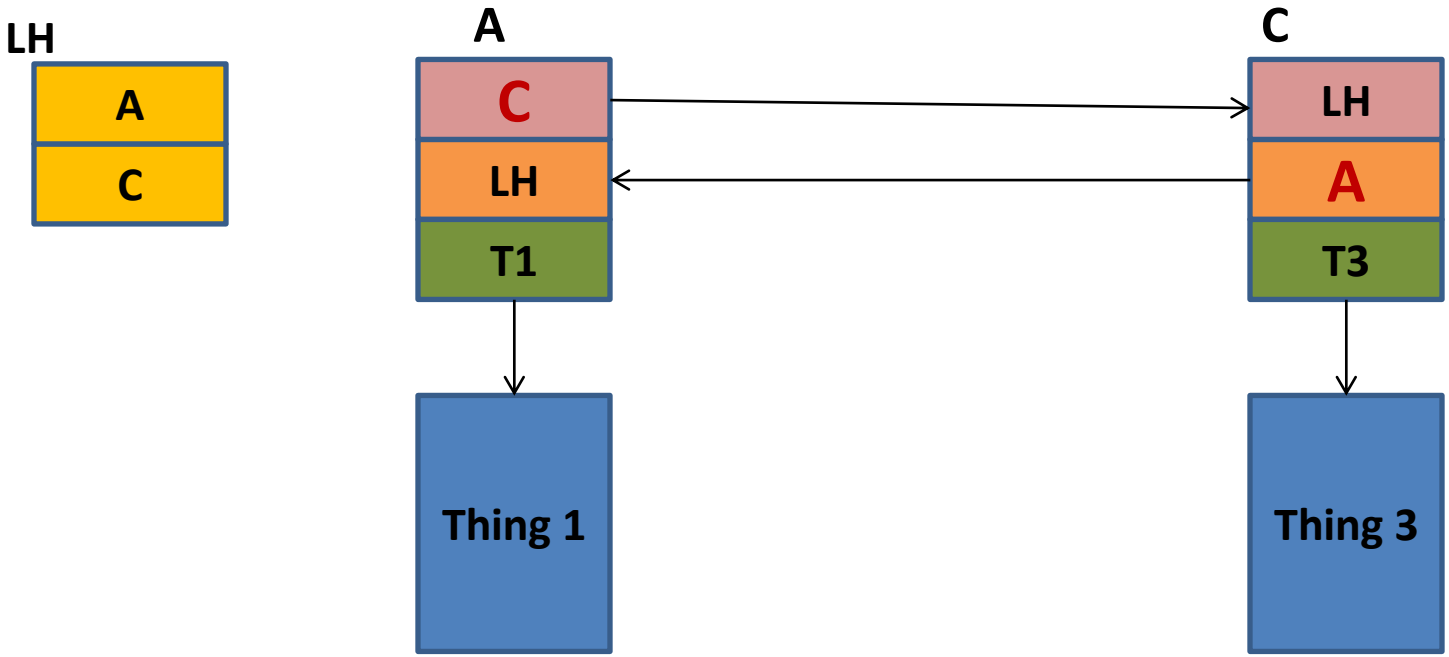


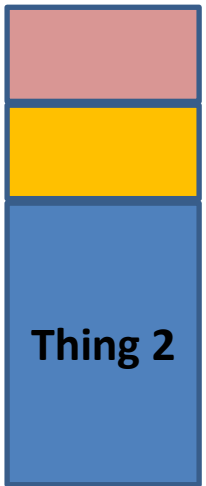
Delete Thing 2 from the Lise: → Adjust FP / BPs in adjacent nodes

Set A's FP to C. // B is off the forward chain
Set C's BP to A // B is off the backward chain

No matter how large the list, **this is all you have to do**
(Extended Array: Everything past insertion must be moved up 1 slot).

Linked List: Element Removal





Linked List Advantages / Disadvantages

- **Advantage**
  - Lists perform generally better in inserting & extracting elements in any position within the container **to which an iterator is already positioned**.
- **Disadvantages**
  - Requires 3 classes to implement (List, Node, Iterator)
  - Lists lack direct access to the elements by their position, so the list must be walked to get into position.

➔ **Ex: To access the nth element in a list, the client first has to iterate from a known position (like the beginning or the end) to that position, which takes linear time.**

Linked List (LL) Usage Guidelines

1. An LL should never be considered when the contained elements do not need to be ordered.

2. An LL might be considered if element ordering is required, access to the "sorted collection" is frequent, and the number of elements changes frequently: (inserts +deletes) > # retrieves.

3. An LL should seriously be considered if the elements being collected are objects and not object pointers (avoids copy constructor calls on each element move).

Linked List vs. Ordered Array

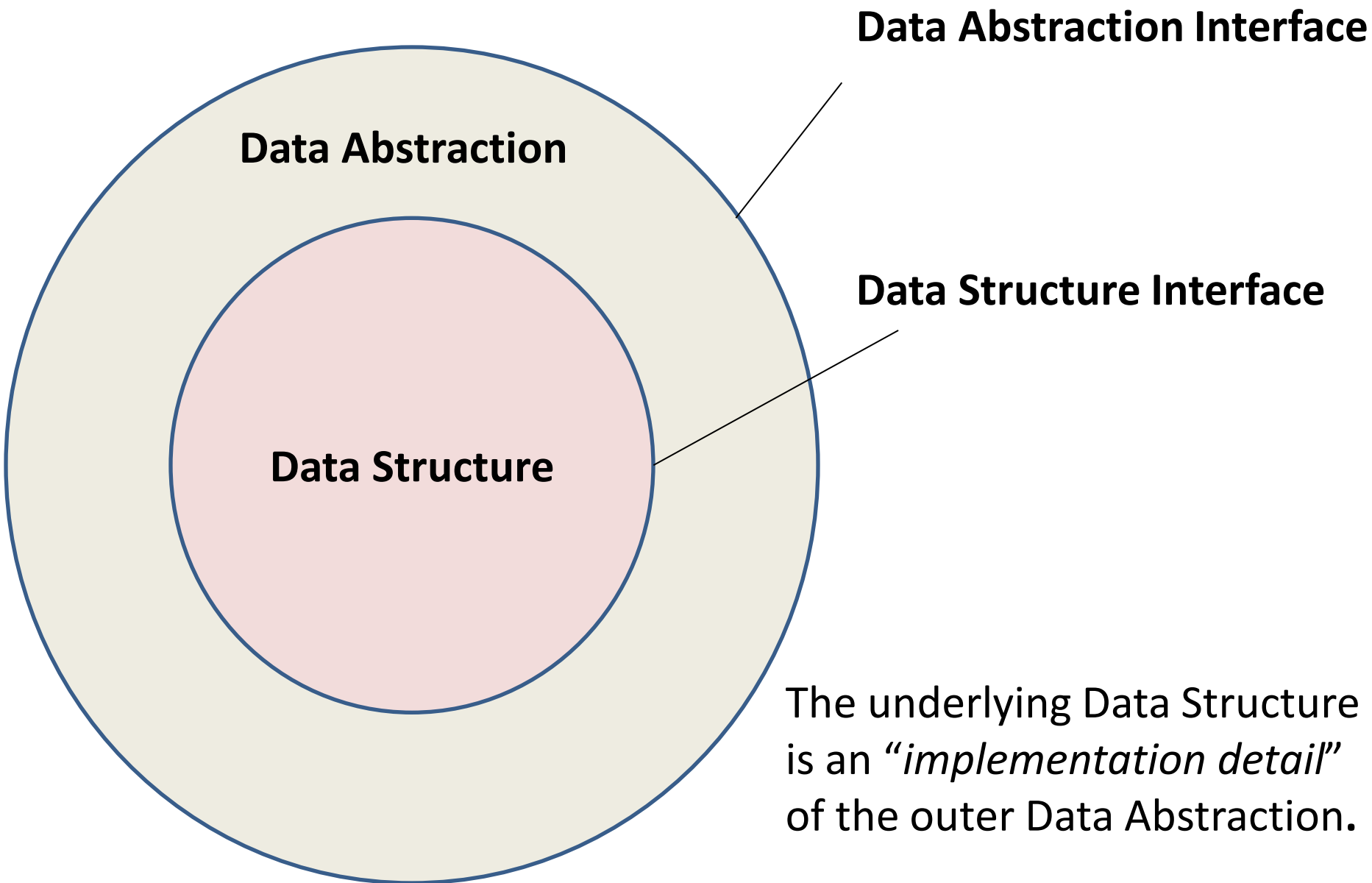
Comparisons (N elements)	Linked List	Ordered
Element "Find"	Scan all elements until match **O(N/2)**	Binary search for specific element **O(log N)**
New Element Addition	Find O(N/2) + Add O(1) = **O(N/2)**	Find O(log N) + push every element past insertion point back one = **~O(N/2)**
Element Deletion (once there)	Find O(N/2) + Remove O(1) = **O(N/2)**	Find O(Log N) + push every element past remove point forward one = **~O(N/2)**
Element "Ordering"	Ordered at Element insertion. **O(1)**	Ordered at Element insertion. **O(1)**
Reporting	All elements are present in order **O(N)**	All elements are present in order **O(N)**

"Ordered" Containers Comparison: Usage

- **Number of elements in Collection <100**
  - Preallocate Array and self-manage
- **Collecting objects rather than object PTRS**
  - Use LL to avoid object "moves" on insert/delete
- **Integer UID range < 10 times number of elements**
  - Use Sparse array (ID = Index)
- **Ratio of Retrievals / Updates** (Inserts & Deletes)
  - Low (R<<U) → LL or OA.
  - High (R>>U) → OA … or Unordered Array + Sorting
    - LL "Find" times much worse

Data Abstractions
(creating a “false front” for a Data Structure)

Data Abstraction “wraps” Data Structure



The underlying Data Structure is an “*implementation detail*” of the outer Data Abstraction.

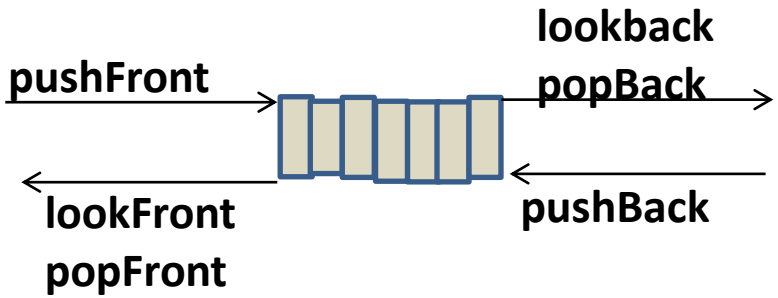
Some Data Collection Abstractions

- **Dequeue (double Q)**
- **FIFO Queue**
- **LIFO Stack**
- **Priority Queue**

- Hash Tables
- Trees
- Graphs
- Heaps

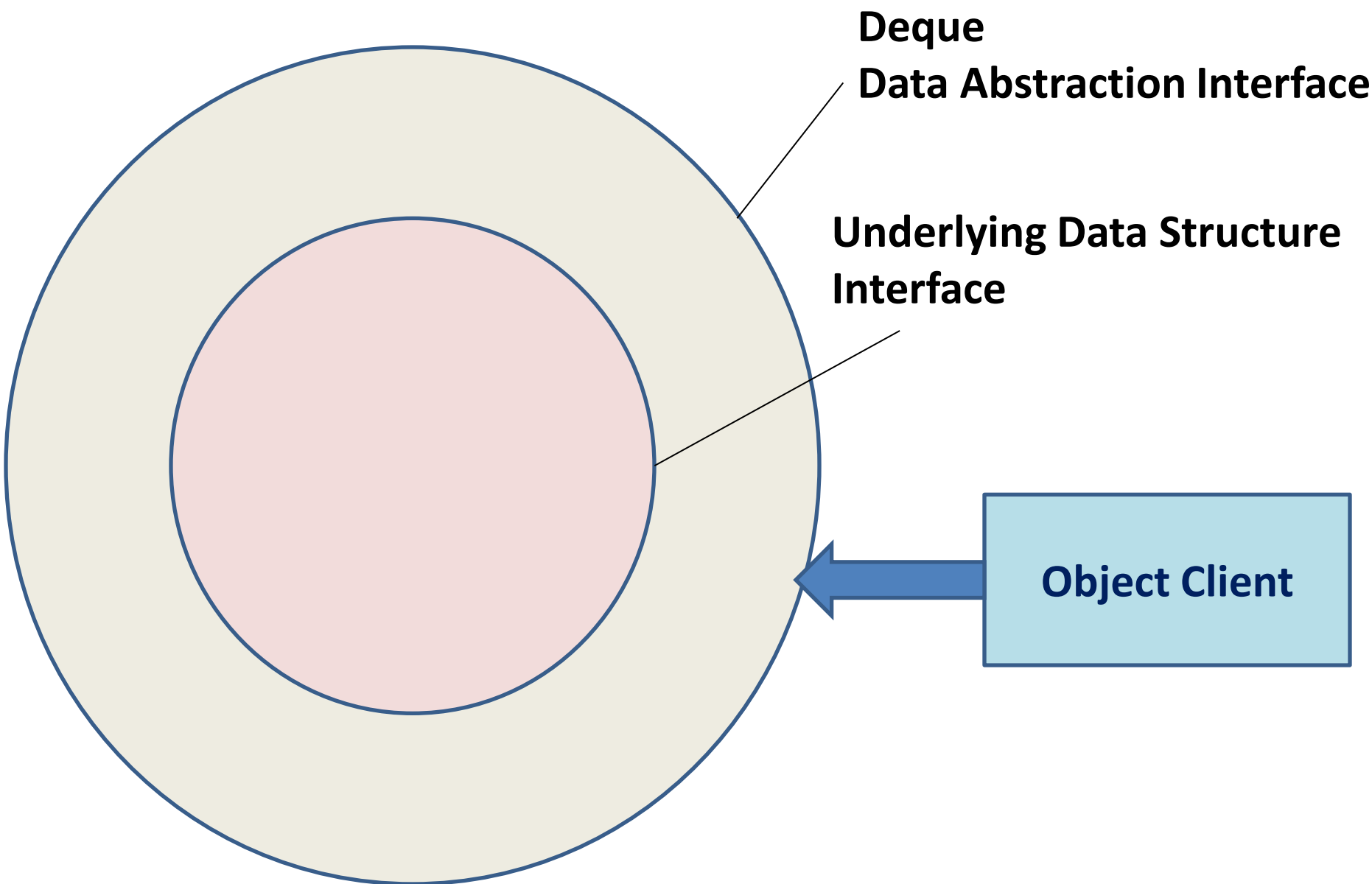
DeQue (DQ)Abstraction (underlying Data Structure TBD)

Deque for "Thing"




```
class DQ
{  public:
      DQ (unsigned int maxSize);
      DQ (DQ&); ~DQ();

     void pushFront (Thing &);
     Thing& lookback ();
      Thing& popBack ();

     void pushBack (Thing &);
     Thing & lookFront ();
     Thing & popFront ();

     bool isEmpty ();
      unsigned int getNumElements();
};
```

“Data Abstraction “wraps” Data Structure



Wrapper Design Pattern

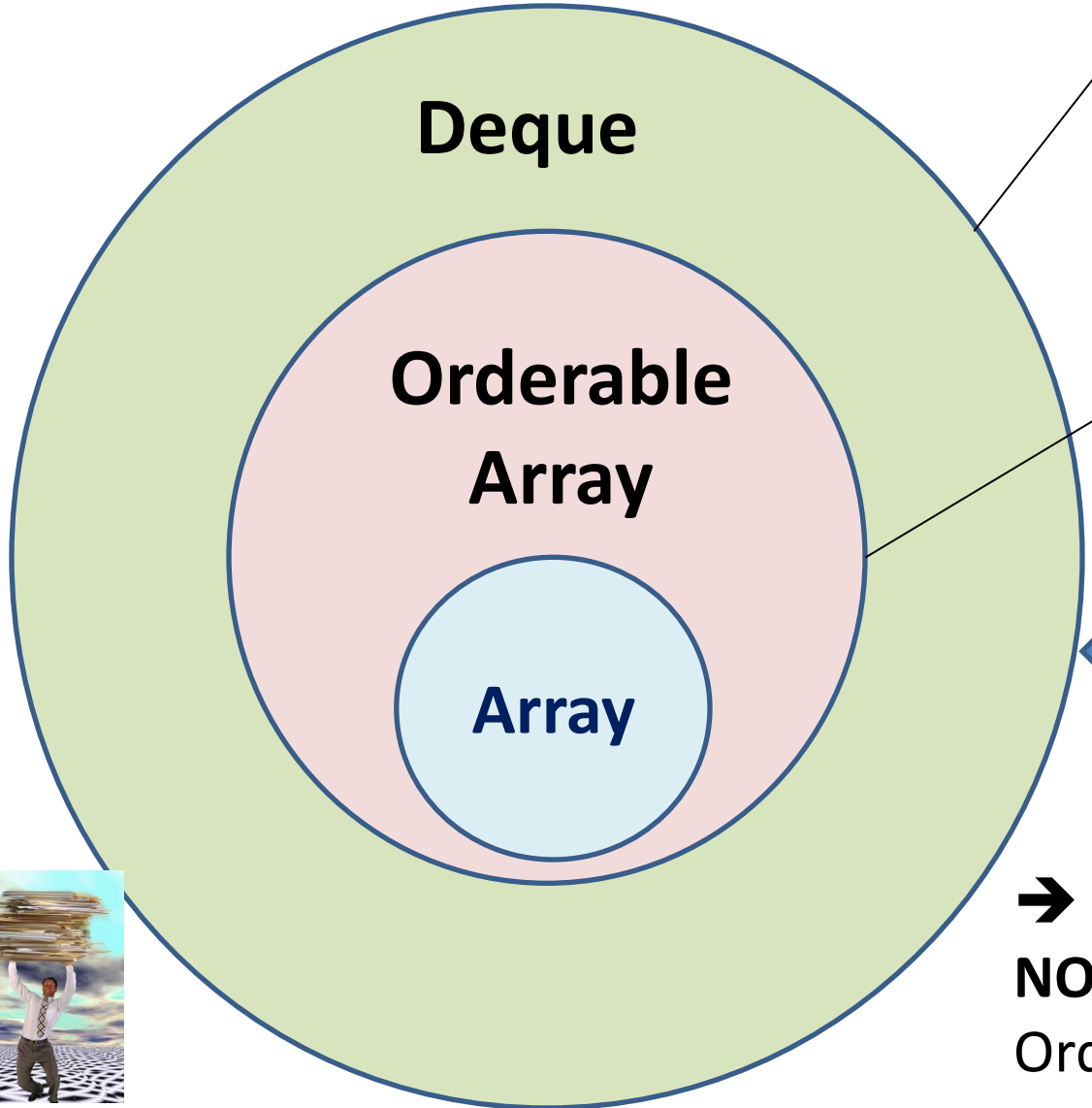
Object Type A "contains" **Object Type B** (or a Ptr to one) in its private data and uses it to deliver most / all of its functionality. In this "Wrapper" case:

1. Does the Client see both interfaces? If not, which interface does the Client see?
2. What should the underlying Data Structure for Deque Data Abstraction be? **Why?**
3. How can the underlying Data Structure be optimally used in supporting the Deque Data Abstraction?

Data Abstraction Wrapper

- **Which interface does the Client see?**
  - The Data Abstraction ("wraps" inner Data Structure)
- **What should the underlying Data Structure for Deque be? Why? Orderable Array**
  - Element insertion / deletion is either at front or back
  - Strength of LL is insertion / deletion at middle
- **How can the underlying Data Structure be optimally used in supporting the Deque?**
  - DQ Constructor creates internal OA

Deque “wraps” OA



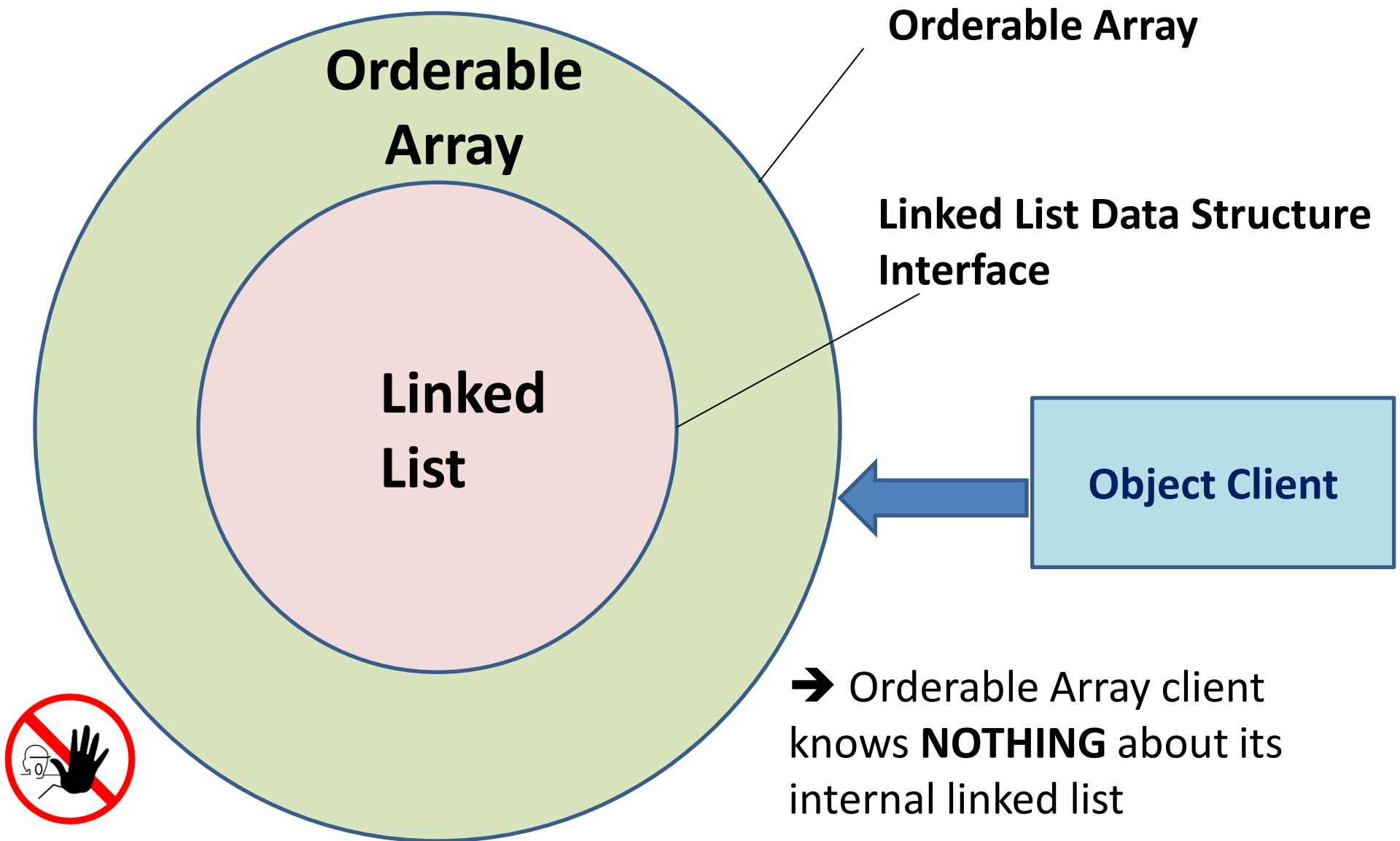
Deque
Data Abstraction Interface

Orderable Array

Object Client

➔ Deque Client knows **NOTHING** about its internal Orderable Array

Could an Orderable Array Wrap a LL?



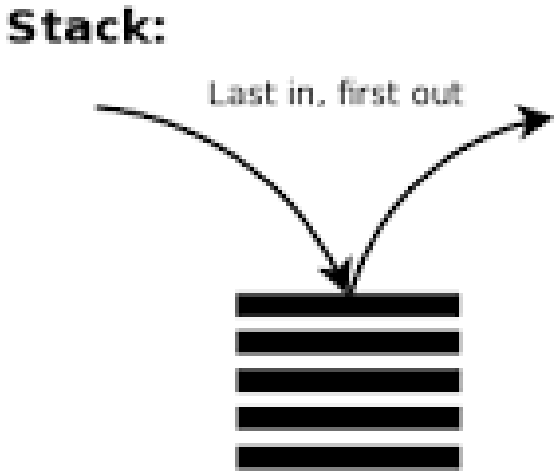
➔ Orderable Array client knows **NOTHING** about its internal linked list

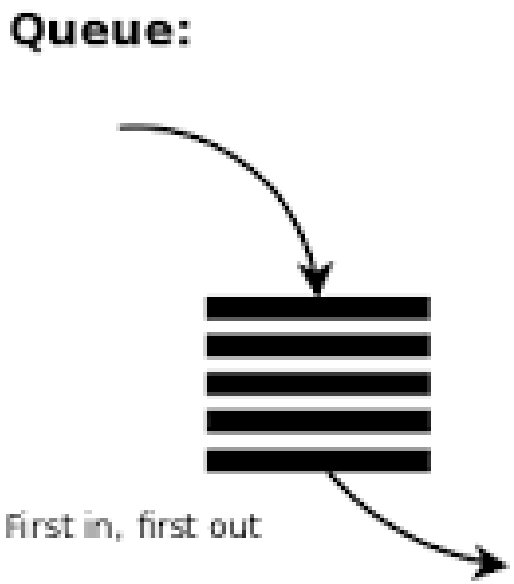
Could an OA Wrap a LL?

Link List Iterator API	Orderable Array API
Thing& getNext();	Thing& get(int index);
Thing& getPrevious();	
void remove ();	int remove(int index);
void set (Thing&);	int set (Thing&, int index);
void add (Thing&);	int insert (Thing&, int index);
	int append (Thing&);

➔ **Yes! Process the index argument by traversing that many nodes in the LL. But why??**

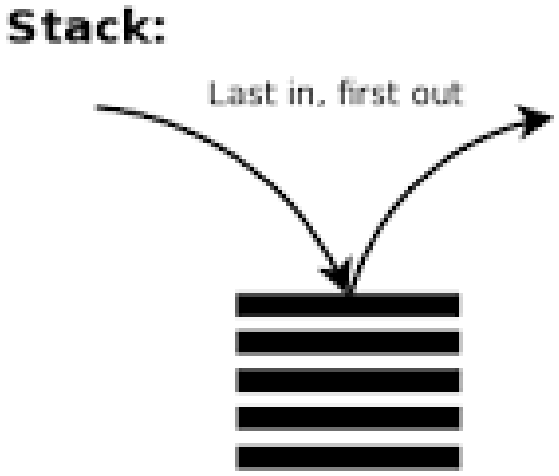
Stacks & Queues

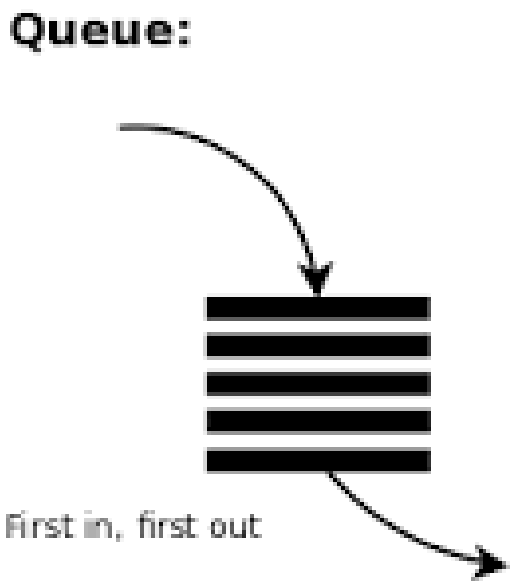




Real World Examples??

Stacks & Queues



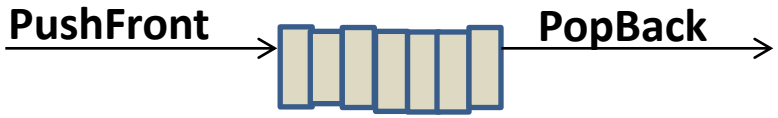


Employed Teachers when layoffs happen.

Employed Teachers when assigning available Sections.

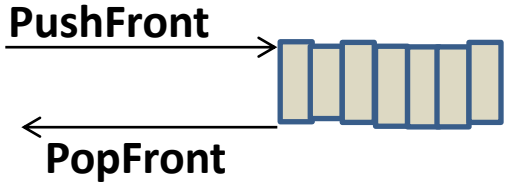
Creating FIFO Queues and LIFO Stacks

FIFO Queue




```
class FQ
{  public:
      void pushFront (Thing &);
      Thing & popBack ();
};
```

LIFO Stack




```
class LS
{  public:
      void pushFront (Thing &);
      Thing & popFront ();
};
```

Relationship: FQ with DQ

1. Wrapper: DQ "wraps" 2 FQ's

- One supports pushBack(), popFront()
- One supports pushFront(), popBack()

2. Inheritance (DQ is a child of FQ)?

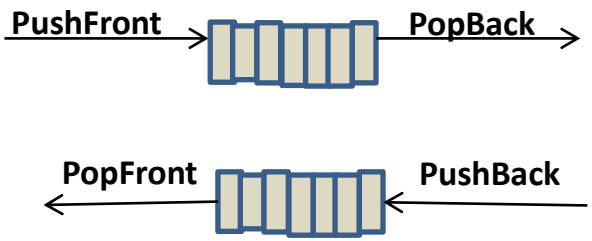
- FQ has 2 of the 4 public functions of DQ
- DQ inherits the FQ interf / impl ops, adds the other 2
  - Why couldn't FQ be a child of DQ?

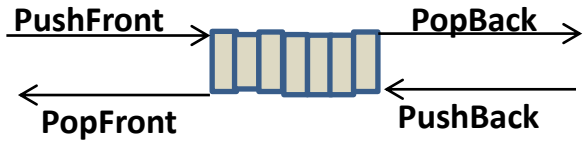
3. Wrapper: FQ "wraps" DQ

- pushFront() & popBack() invoke DQ equivalents
- Remaining functions unsupported in public interface

1. DQ wraps 2 FQs: Fails

2 Single FIFO Queues **Are NOT equivalent to** **1 Deque**





** **Problem:** A Thing inserted via **PushFront (or PushBack)** can be popped off **EITHER** the front **or** the back. If DQ was a collection of 2 FQ's this would not be possible!!

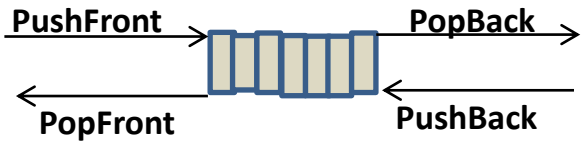
➔ DQ cannot wrap 2 separate FQ's because even taken together, they cannot implement the public function requirements of a DQ.

2. DQ inherits from FQ: Fails

FIFO Queue

CANNOT be parent of

1 DQ



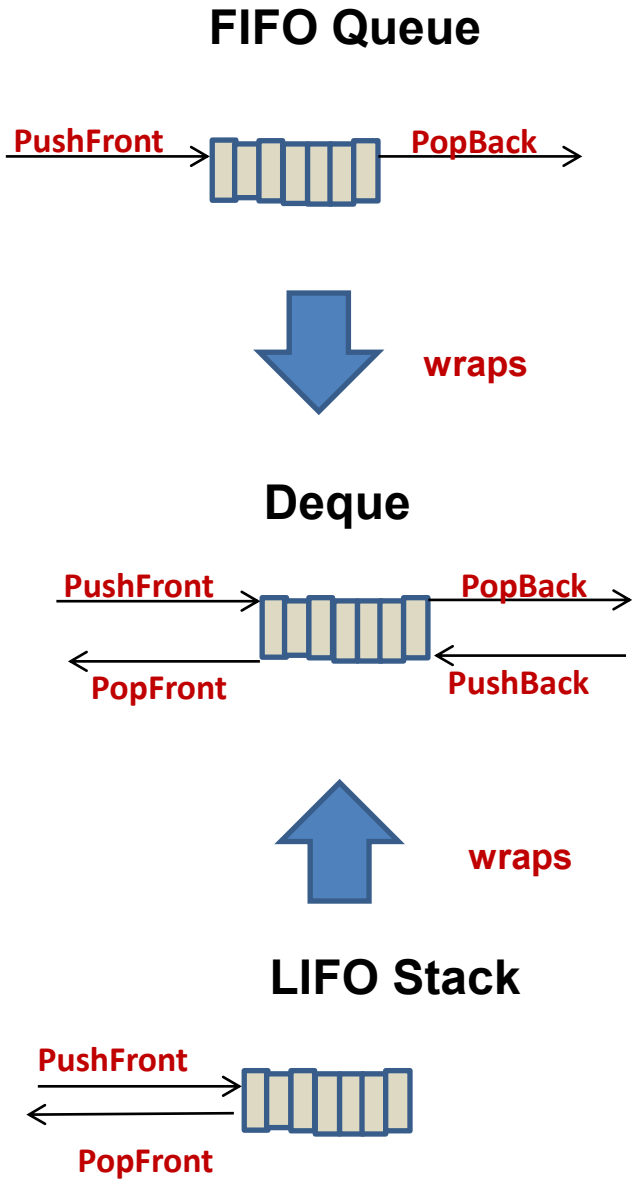
** **Problem:** How could the DQ implementations of PushBack and PopFront learn where the array used internally by FQ methods PushFront and PopBack was, how many entries were in it, what was the current front and back … ??

➔ **As a child of a FIFO Queue, a DQ will inherit the implementation of 2 of the 4 functions it needs, but it cannot use them to add the other 2 functions it must support!**

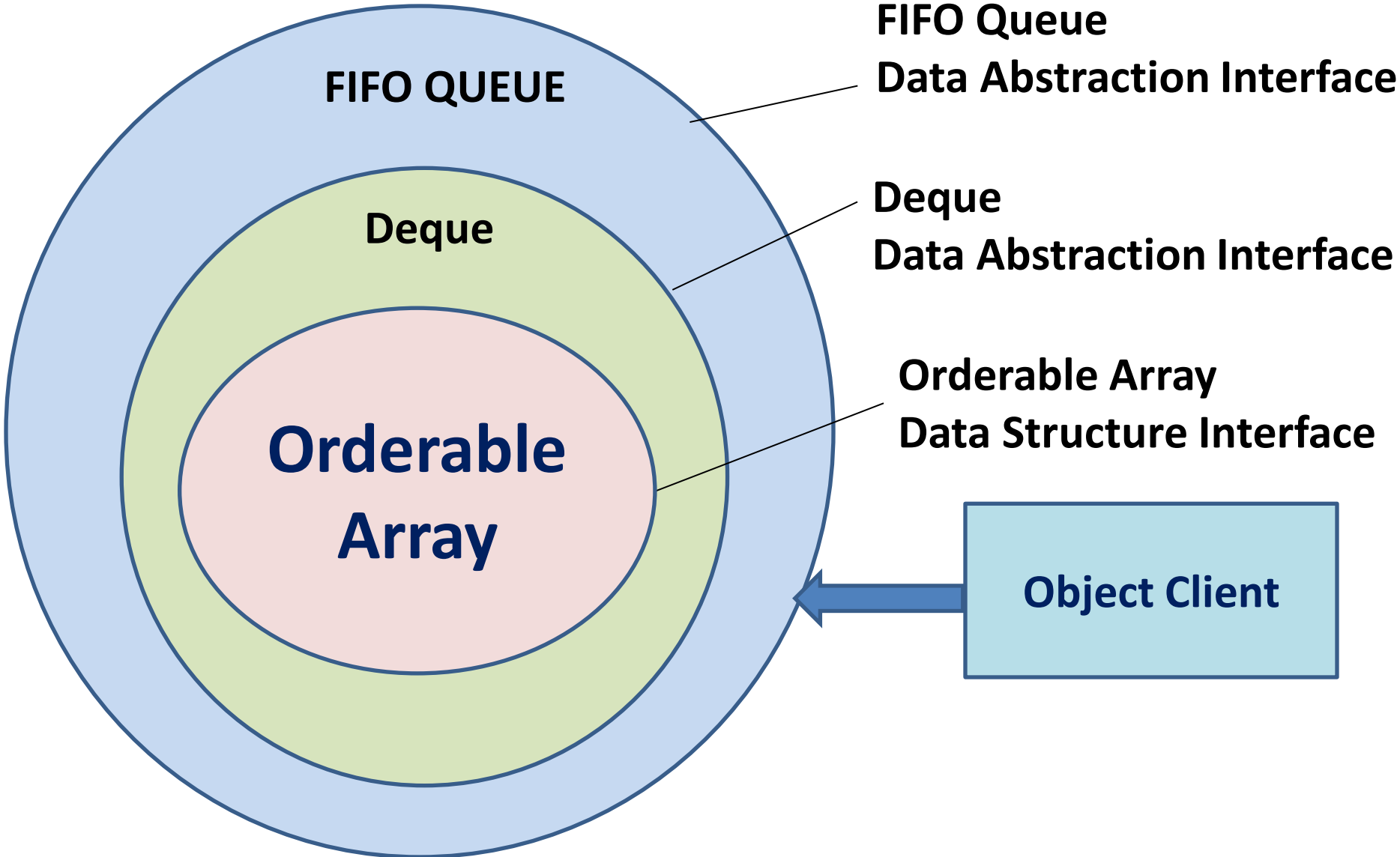
3. FQ (and Lstack) each wrap DQ: Success!

Once a Deque is implemented, it can be wrapped by FIFO Queue (and LIFO Stack) to do all their required functions!

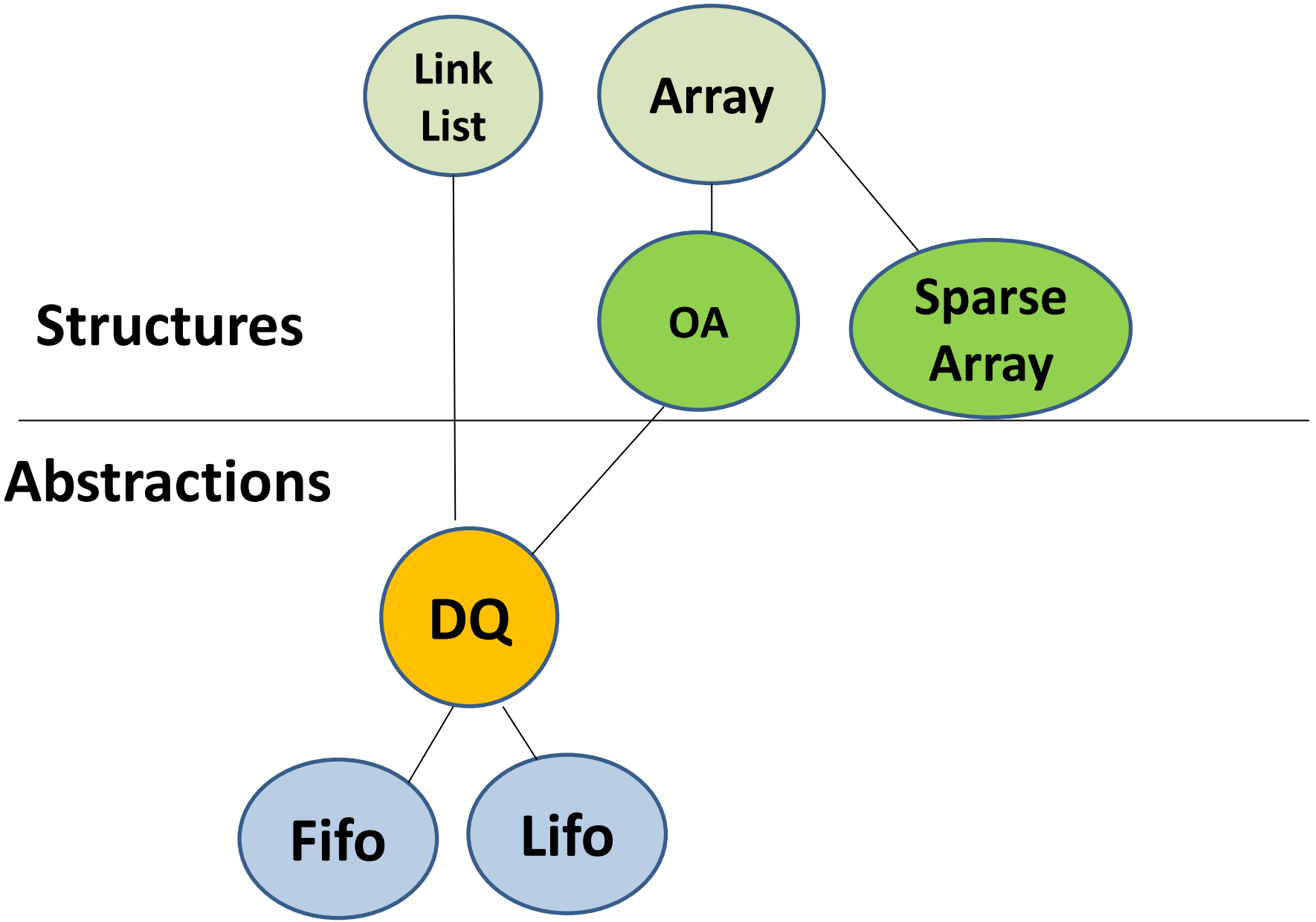
```
class Fifo {
   private
        DQ dq;  // Created at construction
   public:
        void pushFront (Thing& x)
        { return (dq.pushFront (x)); }

        Thing & popBack()
         { return (dq.popBack()); }
};
```



FQ / LS “wrap” DQ


FIFO QUEUE
Deque
Orderable Array
FIFO Queue
Data Abstraction Interface
Deque
Data Abstraction Interface
Orderable Array
Data Structure Interface
Object Client

Relationship of Collections



Questions?