CS416

Introduction to Computer Science II

Spring 2018

7 Introduction to ADTs and Data Structures (Cont'd)

Chapter 14

Previously in 415-416

- Stacks and Queues

List

- A linear collection of objects such that you can *add anywhere* and *remove anywhere*, but can only *access sequentially*
  - prioritized list of tasks to complete
    - a new task may get inserted anywhere in the list
  - alphabetized list of names
    - a new name can go anywhere
  - just about anything that needs ordering or re-ordering

List ADT

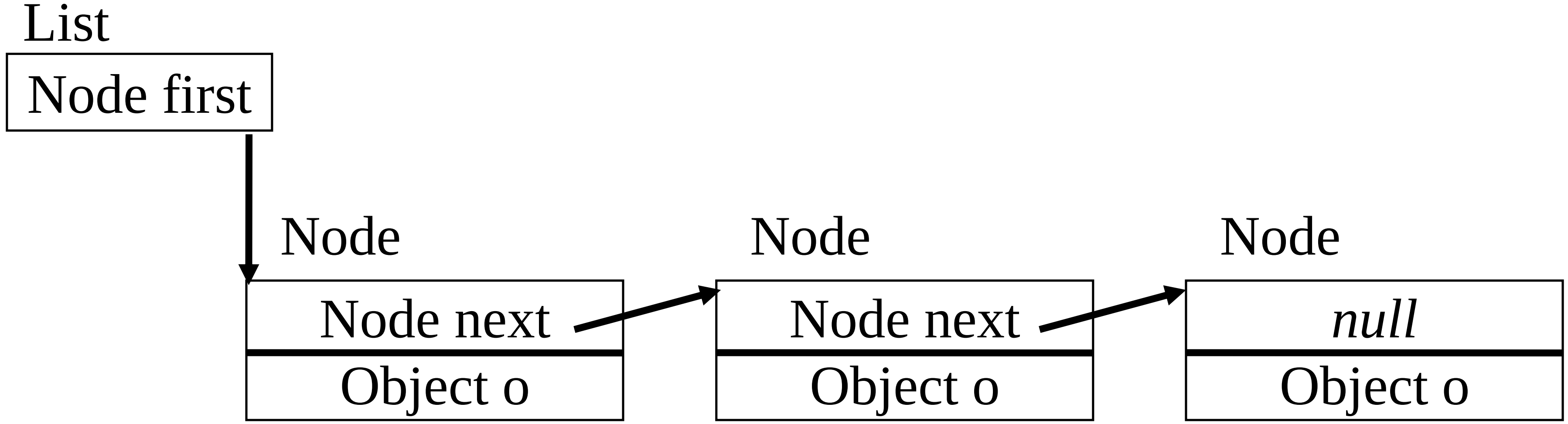
- *add(Element)*: several options
  - List includes algorithm to insert at the “right” place.
  - Application code traverses the list until it finds where the new object belongs; the *add* method inserts the argument “here”, i.e. at the “current” position.
- Deletion
  - *Element remove()*
    - remove and return the element at the “current” position.
  - *void remove(Element)*
    - search for *Element* in list, remove if found

List ADT (cont)

- Access: many possible access options
  - *first()*
  - *next()*
  - *find(key)*
  - *find(Object)*
  - *get(int) // get a specific entry based on position*
- Utility behavior
  - Test if there is anything in the list: *isEmpty()*
  - If list has a fixed size, need: *isFull()*
  - Determine number of current entries: *int size()*
  - Determine maximum possible size: *int capacity()*

Linked Lists

- Traditional *List* implementation is done with a *Linked List* (a concrete data structure)
  - Each object is *in* a *Node*
  - There is a reference to the first *Node* on the list
  - Each *Node* has a reference to the next *Node*



LinkedList Class

```
public class LinkedList<T>
{
   protected Node<T> _head;
   public LinkedList()
   {
      _head = null;
   }
   public T first()
   {  if ( _head == null )
         return null;
      return _head.data;
   }
   public void add( T adder )
   {
      _head = new Node<T>( adder, _head );
   }
   public T find( String key )
   {
      Node<T> next = _head;
      while ( next != null
            && !next.compare( key ) )
         next = next.next;
      if ( next == null )
         return null;     // not found
      else
         return next.object;
   }
   protected class Node  ....
}
```

Note: This class <u>does</u> <u>not</u> implement the “hard” behavior of the *ListADT* (like doing a sort).

`_head references first Node`

`first()` `returns the object, not a Node`

`Add at front, let Node class update the next field`

`find()` `returns the object, not a Node`

LinkedList:Node

```
protected class Node<T>
{
    public Node<T> next = null;
    public T data = null;

   public Node( T adder, Node n )
   {
     data = adder;
     next = n;
   }
   public boolean compare( String key )
   {   /*.???..*/
       return data.equals( key );
       // is equals(String) a T method?
   }
}
```

- Node is an inner class of LinkedList.
- Designed for a list where each new entry becomes the *head* and its *next* points at the old *head*.

Consider the add sequence:

1. create L — LinkedList
2. add(A) — LinkedList → A
3. add(C) — LinkedList → C → A
4. add(E) — LinkedList → E → C → A

Note: in this version *Node* fills in the *next* link; this need not be the case; the *LinkedList* class often does that.

LinkedList Implementation

- Can write *Node* as an inner class of a *LinkedList* class.
  - *Node* being *private* or *protected*, however, limits the functionality
- Textbook describes a public *Node* class
  - That has other disadvantages, especially exposing the implementation
- *Package* visibility is probably best in this case

DataList class

- Special purpose list that can only contain objects of type *DataNode*, an external class
- *DataNode* contains an object of type *Data*, with **public** *String* (key) and *int* (value) fields

```
        DataList Public Interface
DataList()          - create empty list
DataNode head()     - return 1st node
DataNode tail()     - return last node
void addHead( Data ) - add at start
void addTail( Data ) - add to end
void add( Data )     - add to end
int  size()          - size of list
DataNode find( String k ) - search list
          for 1st node with key, k;
          return DataNode or null
```

```
public class DataNode
{
  private DataNode _next = null;
  private Data     _data = null;

  public DataNode( Data d, DataNode n
)
  {
    _data = d;
    _next = n;
  }
  public DataNode next()
  {
    return _next;
  }
  public Data data()
  {
    return _data;
  }
  public void setNext( DataNode n )
  {
    _next = n;
  }
}
```

Array ADTs

- Many variations possible for an Array ADT
  - Abstract Data Type for a basic Java array
    - *add*: Array constructor adds <u>all</u> elements to the array at once, none can be added later
    - *delete*: Array elements cannot be deleted (but their *values* can be changed).
    - *access*: by position *index.*
    - Of course, Java arrays are built into the language, not as a pre-defined class, but as an integral language feature.
  - Suppose we want to create an *Array* class with these semantics?

Array ADT

- Define *Array* class with *Java's* basic array semantics
  - *add* element
    - only at construction!
  - *remove* element: can't easily
  - *access*
    - random: by index with *get/set*
    - sequential: not supported by the class, but can program it

```
Array<String> names = new Array<String>(4);
```

```
String check = names.get( i );
```

```
names.set( i, "Name" );
```

```
for ( i=0; i < names.length; i++)
   print( names.get( i ));
```

ArrayList ADT

```
ArrayList<String> names = new ArrayList<String>();
```

- *add* element
  - to end of list; or insert at specified index

```
names.add( "Smith" );
```

```
names.add( index, "Smith" );
```

- *remove* element
  - at a specified index; or first element that *equals* a given element

```
names.remove( index );
```

```
names.remove( "Smith" );
```

- *access*
  - random: by index with *get*
  - sequentially: via *iterator()* and its *next()*

```
for ( i=0; i < names.length; i++)
   print( names.get( i ));
```

```
Iterator<String> iter;
iter = names.iterator();
while ( iter.hasNext() )
   print( iter.next() );
```

StringDictionary

```
StringDictionary names = new StringDictionary();
```

- *add* element (*insert*)
  - in alphabetical order

```
void names.insert( "Smith" );
```

- *remove* element (*delete*)
  - based on String match

```
boolean names.delete( "Smith" );
```

- *access*
  - by String match (*search*)

```
boolean names.search( "Smith" );
```

  - sequentially:
    - via *first(), next()* or
    - via *iterator()* and its *next()*

```
Iterator<String> iter;
iter = names.iterator();
while ( iter.hasNext() )
   print( iter.next() );
```

GenericDictionaryADT, v1

- Naive conversion to generic doesn't work
  - need to create a complete T object to search or delete
  - what is *alphabetic order* for T?

```
public interface StringDictionary
{
  public void insert( String item );
  public boolean search( String item );
  public boolean delete( String item );

  public boolean isEmpty();
}
```

```
public interface DictionaryADT<T>
{
  public void insert( T item );
  public boolean search( T item );
  public boolean delete( T item );

  public boolean isEmpty();
}
```

GenericDictionaryADT, v2

- Searching for T elements
  - Suppose T elements have a *String* “key” that identifies them
  - Can pass *String* to *search* and *delete*

```
public interface DictionaryADT<T>
{
  public void insert( T item );
  public boolean search( String key );
  public boolean delete( String key );

  public boolean isEmpty();
}
```

- Want key to access entire T object and return it
- But that's not enough!
  - How do we *get* T's *String* key?

```
public interface DictionaryADT<T>
{
  public void insert( T item );
  public T    search( String key );
  public T    delete( String key );

  public boolean isEmpty();
}
```

GenericDictionaryADT, v2

- How does *search* and *delete* get T's key?
  - Need to specify an interface that defines how to access a *String* key
  - Need to require a type to support that interface if you want to use it with *DictionaryADT*

```
public interface StringKey
{
  public String getStringKey( );
}
```

Note the use of the keyword **extends**; in this context it actually means either *extends* or *implements*.

```
public interface DictionaryADT<T extends StringKey>
{
  public void insert( T item );
  public T    search( String key );
  public T    delete( String key );

  public boolean isEmpty();
}
```

Called a ***bounded generic***

But, what if T <u>doesn't</u> have *String* key?

GenericDictionaryADT, v3

- Searching for T elements
  - define a key type, K, that can be used to identify T element
- Add a type parameter, K, to the generic specification
  - use K in search/delete
- Define a key access interface
  - Specify a *bounded generic* that restricts T to classes that support *KeyProvider<K>*

```
public interface DictionaryADT<K, T>
{
  public void insert( T item );
  public T search( K key );
  public T delete( K key );

  public boolean isEmpty();
}
```

```
public interface KeyProvider<K>
{
  public K getKey();
}
```

```
public interface DictionaryADT
      <K, T extends KeyProvider<K> >
{
  public void insert( T item );
  public T    search( K key );
  public T    delete( K key );

  public boolean isEmpty();
}
```

GenericDictionaryADT, v3

- But, we're still not done!
  - How do we compare K objects?
  - Java has a *Comparable<T>* interface
  - We need to bound the K data type to ones that implement *Comparable<K>*

```
public interface DictionaryADT
      <K, T extends KeyProvider<K>>
{
  public void insert( T item );
  public T    search( K key );
  public T    delete( K key );

  public boolean isEmpty();
}
```

```
public interface DictionaryADT
      <K extends Comparable<K>,
       T extends KeyProvider<K>>
{
  public void insert( T item );
  public T    search( K key );
  public T    delete( K key );

  public boolean isEmpty();
}
```

```
public interface Comparable<T>
{
  public int compareTo( T item );
}
```

Dictionary Class

- The *Dictionary* class implements the *DictionaryADT* interface using a linked list
- It uses recursion for all 3 major methods:
  - search, insert, delete
  - Each has its own “helper” method that does the recursion: *searchNode*, *insertNode*, *deleteNode* (these are called *searchAux*, etc. in the book)
  - The “helpers” are started with the head node and recurse down the list until they find the “right” place

Dictionary.search

- Note:

For simplicity, using my Node class with public fields for *data* and *next*

```
public T search( K key )
{
  Node<T> found = searchNode( key, _first );
  if ( found == null )
    return null;
  else
    return found.data;
}
```

```
private Node<T> searchNode
       ( K key, Node<T> cur )
{
  if ( cur == null )  // base case 1
    return null;
  K curKey = cur.data.getKey();
  int c = curKey.compareTo(key);
  if ( c == 0 ) // keys match, base 2
    return cur;
  else if ( c < 0 ) // curKey < key
    return searchNode( key, cur.next );
  else              // key not in list
    return null;    // base case 3
}
```

`cur == null`: Got to end of list without finding the node; return null

If keys match, we've found the node; return it.

This key < search key; need to check rest of the list; recurse.

This key > search key; search key cannot be on the list; return null

Dictionary.delete

- Key idea:

delNode returns what previous node's *next* field (or _first) should become.

```
public T delete( K key )
{
  _deleted = null; // set to deleted
  _first = delNode( key, _first );
  if ( _deleted == null )
     return null;
  else return _deleted.data;
}
```

```
private Node<T> delNode
       ( K key, Node<T> cur )
{
  if ( cur == null )
    return cur;
  K curKey = cur.data.getKey();
  int c = curKey.compareTo(key);
  if ( c == 0 ) {// keys match
     _deleted = cur; // save deleted
     return cur.next;
  }
  else if ( c < 0 ) {// curKey < key
    cur.next = delNode(key, cur.next);
    return cur;
  }
  else               // key not in list
    return cur;
}
```

`cur == null`: Got to end of list without finding the node to delete; previous node's **next** field was *null*; it should still be *null*.

Keys match; this is node to delete; return this node's **next**; previous node will skip this one!

curKey < key: the node with the search key could still be later in the list, so recurse; result is what this node's **next** should be and previous nodes **next** is unchanged (cur)

curKey > key: key is not on the list, stop looking; return **cur** since previous node's **next** remains unchanged.

Dictionary.insert

- Key idea:

insNode returns what previous node's *next* field (or _first) should become.

```
public void insert( K key )
{
  Node<T> add = new Node<T>( T );
  _first = insNode( add, _first );
}
```

```
private Node<T> insNode
      ( Node<T> add, Node<T> cur )
{
  if ( cur == null )
    return add;
  K curKey = cur.data.getKey();
  K addKey = add.data.getKey();
  int c = addKey.compareTo( curKey );
  if ( c == 0 ) // keys match
    return cur; // if no duplicates!
  else if ( c < 0 ) // addKey < curKey {
    cur.next = insNode( add, cur.next );
    return cur;
  }
  else  {              // insert here
    add.next = cur;
    return add;
  }
}
```

`cur == null`: Got to end of list without finding the node's place; so it goes at the end and the previous node's **next** field should point to the new node.

Keys match; this version doesn't allow duplicates, so it's done; no change to previous node's **next**; it should still point to this node.

addKey < curKey: the spot for the new node could still be later in the list, so recurse; result is what this node's **next** should be and previous nodes **next** is unchanged (cur)

addKey > curKey: key is not on the list, this is where it belongs; new node points at **cur** and previous node's **next** should be new one

Iterative Implementation

- Typical iterative implementation of a sorted list
  - *search* is a simple loop
  - *delete* and *insert*
    - could keep a pair of references while traversing the list: *cur* and *prev*
      - once find element to delete or spot to insert, *prev* is there when you need it
    - use *cur.next* for testing
      - when find element or spot, you still have *cur*
  - Still requires 3 <u>separate</u> similar methods

LinkedList Implementation Notes

- Three separate, similar search methods
- Code complexities dealing with head/tail
  - Updating the head and tail variables
  - Handling cases where the head or tail is deleted
- LinkedList variations
  - 2-way linked list
  - 1-way ring
  - 2-way ring
  - sentinels

Two-Way Linked Lists

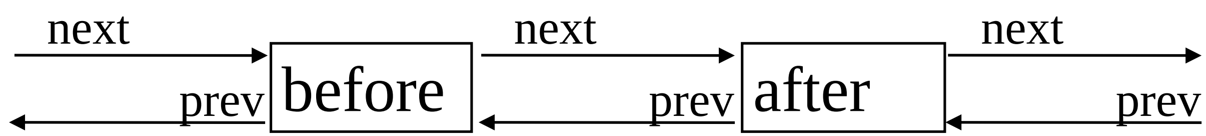
- Suppose every *Node* contains a reference to the *previous* node on list as well as the *next* one
- *delete* and *search* could share code
  - A *searchNode* method would return the *Node* with the matching key. That node has *prev* and *next* links that *delete* can use to detach the node.

```
public T search( K key )
{
  Node<T> found = searchNode( key );

  if ( found != null )
    return found.data;
  else
    return null;
}
```

```
public T delete( K key )
{
  Node<T> found = searchNode( key );

  if ( found != null )
  {
    if ( found.prev != null )
      found.prev.next = found.next;
    if ( found.next != null )
      found.next.prev = found.prev;
    // check head/tail
    return found.data;
  }
  else
    return null;
}
```

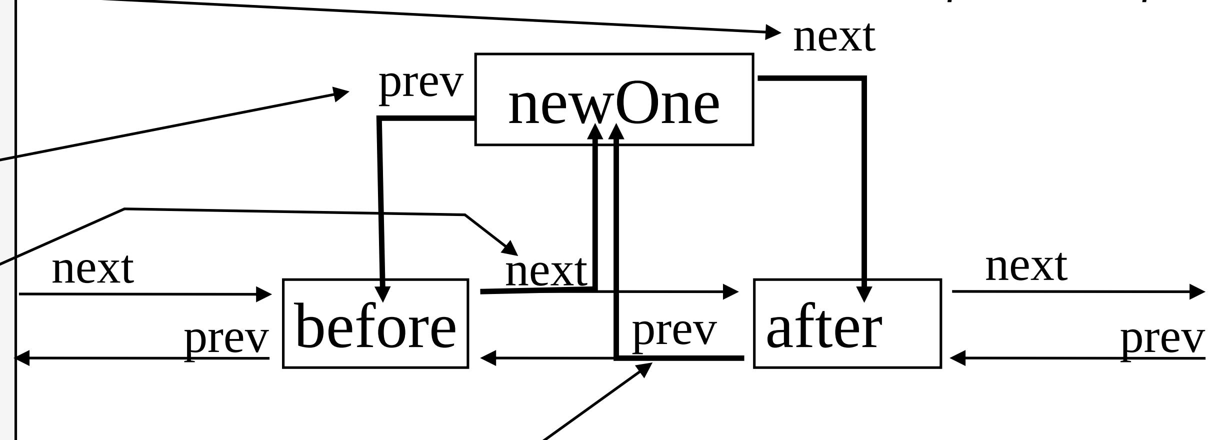
Two-Way Linked List Insert

```
class List<T> // package class
{
  public addB4Node( T t,
                    Node<T> after )
  {
    Node before = null;
    Node newOne = new Node( t );
    newOne.next = after;
    if ( after != null )
    {
       before = after.prev;
       newOne.prev = before;
       if ( before != null )
          before.next = newOne;
       else
          head = newOne;
       after.prev = newOne;
    }
    else // adding after tail
      ...
  }
  .....
}
```



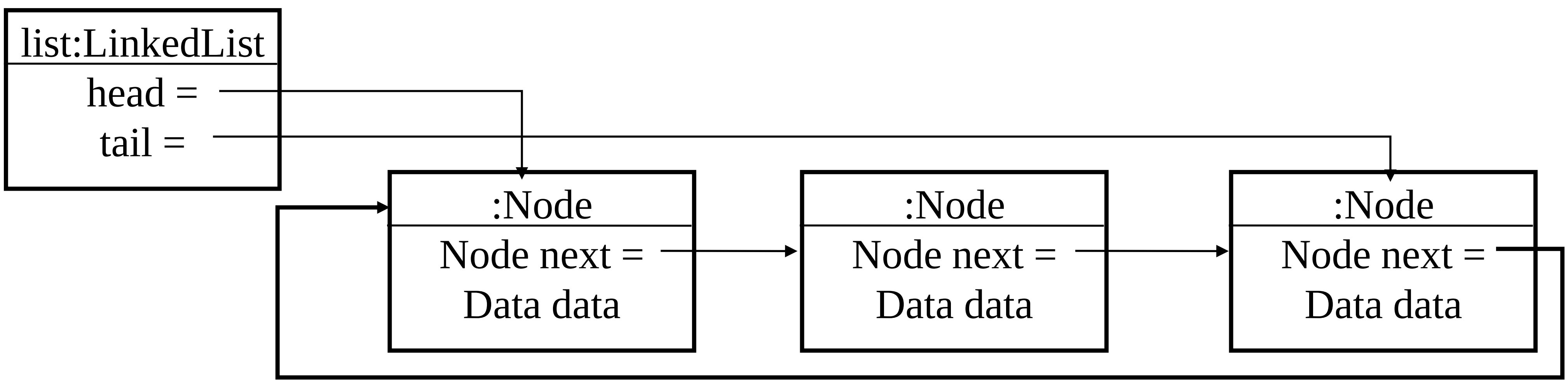
In this example, the links are filled by the *List* class, rather than the *Node* class -- most common approach.

New node, *newOne*, to be inserted between *before* and *after*



One-Way Ring

- Suppose "normal" search of a particular list finds a <u>subset</u> of the nodes in the order they occur on list
  - can start next search where previous left off
- Rather than tail.next = null, set tail.next = head
  - called a <u>*one-way ring*</u>



Ring Implementation Options

- Moving *head*: every search updates *head*
  - useful if some searches are <u>not</u> in order so each search might "wrap around"

```
// Moving head implementation
Node find( String key )
{
  if ( head == null )
    return null;
  Node start = head; // local variable
  Node found = null;
  do {
    if ( head.key.equals( key ))
      found = head;
    else
    {
      tail = head;
      head = head.next;
    }
  }
  while ( found == null && start != head );
  return found;
}
```

This is a good (rare) example where *do-while* makes sense

Ring Implementation Options

- Add a *cur* reference
  - all searches start at *cur* and stop at *head*
  - need a *reset* to set *cur* back to *head*

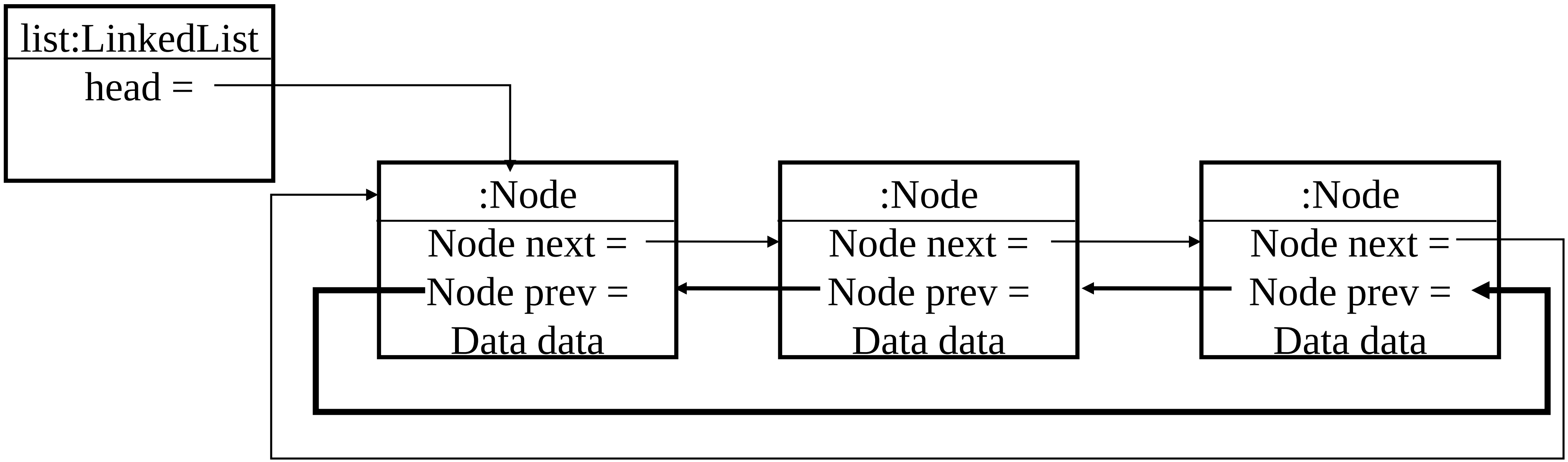
```
// cur implementation
Node find( String key )
{
  if ( cur == null )
    return null;
  Node node = cur; // local variable
  Node found = null;
  do {
    if ( node.key.equals( key ))
      found = node;
    else
      node = node.next;
  }
  while ( found == null && node != head );
  if ( found != null )
    cur = found;
  return found
}
```

This is a good (rare) example where *do-while* makes sense

Implementation with *cur*: starts at *cur*, doesn't update *head*, does update *cur* to *found.next* but only if found.

Two-Way Ring

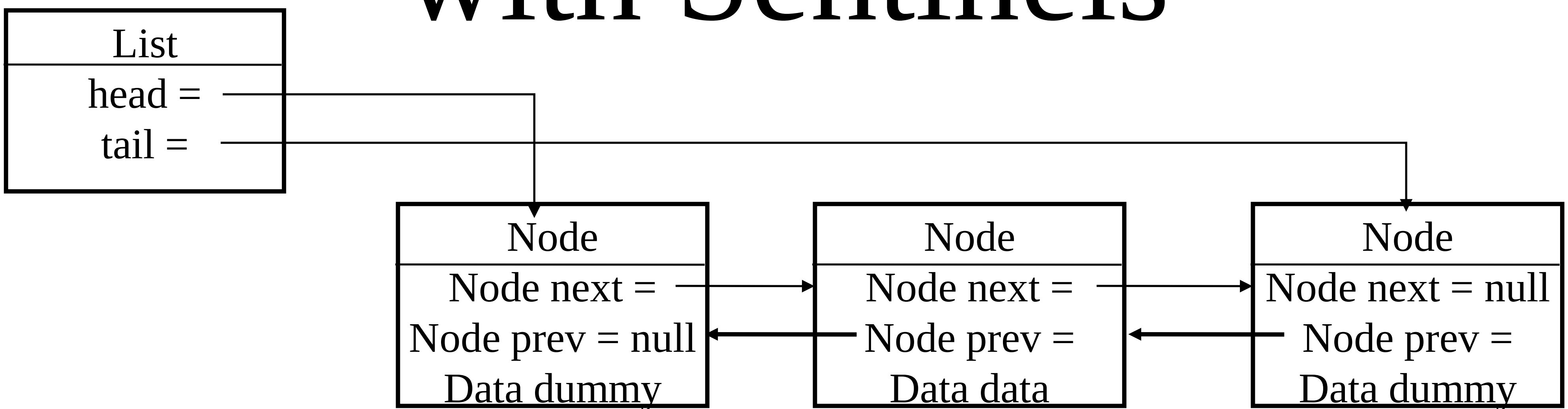
- Can make 2-way list into a 2-way ring
  - Don't need explicit *tail* reference; it's trivial to get to tail
    - *tail = head.prev*



Sentinels

- Linked list code has several special cases that complicate the code:
  - adding/deleting head
  - adding/deleting tail
- Can simplify code considerably if add "dummy" or *sentinel* nodes at both ends
  - start a search at *head.next* instead of at *head*
  - end search when *cur == tail* (before processing *cur* node)

Two-Way List with Sentinels




```
public T delete( K key )
{
  Node<T> found = searchNode( key );

  if ( found != null )
  {
      found.prev.next = found.next;
      found.next.prev = found.prev;

      return found.data;
  }
  else
    return null;
}
```

The same 2 lines of code work for **all** cases, even when deleting the last element

Insertion code is similarly simplified

Sentinels created in constructor

```
public List()
{
  head = new Node( ... );
  tail = new Node( ... );
  head.next = tail;
  tail.prev = head;
}
```

<u>IMPORTANT</u>

No “real” data is ever stored in the sentinels!!!

ADT Implementation Notes

- Different *concrete data structures* can be used to implement the same ADT
  - Choice depends on how the data will be used
  - Java arrays are most efficient for basic access, but expensive in terms of data insertion/deletion
  - *ArrayList* and *Vector* make insertion/deletion more convenient, but still expensive
  - Linked lists are most efficient for insertion/deletion, but expensive for searching