CS 351
Design of Large Programs
Complex Data Structures

Brooke Chenoweth

University of New Mexico

Spring 2024

What is a Data Structure?

A data structure is defined by:

- an organization of the data being stored
- a set of operations for effective access to this data

Algorithmic complexity (efficiency) is directly tied to data organization

- searching for an element in an unsorted array takes linear time: $2^{32} = 4,294,967,296$
- searching for an element in a sorted array takes logarithmic time: 32

Key to computing efficiency is reducing algorithmic complexity

Relation to Abstract Data Types

Abstract Data Type specifications are given in terms of:

- an abstract data representation
- a set of operations over the abstract representation
- signature or interface
- semantics

A data structure is a concrete realization of the ADT

- it should preserve encapsulation
- it can be analyzed with respect to performance
  - at the formal level – time and space complexity
  - at the execution level – runtime and resource usage

Common Data Structures

- Some basic data structures are built into the language
  - arrays
- Some data structures are provided in standard libraries
  - linked lists
  - hash tables
  - search trees
- Other data structures need to be explicitly coded
  - trees
  - graphs
- Generic types facilitate generality and reuse
- Java collections expand the range of ready to use common data structures
  - designed, coded, and optimized

Illustration: Linked List

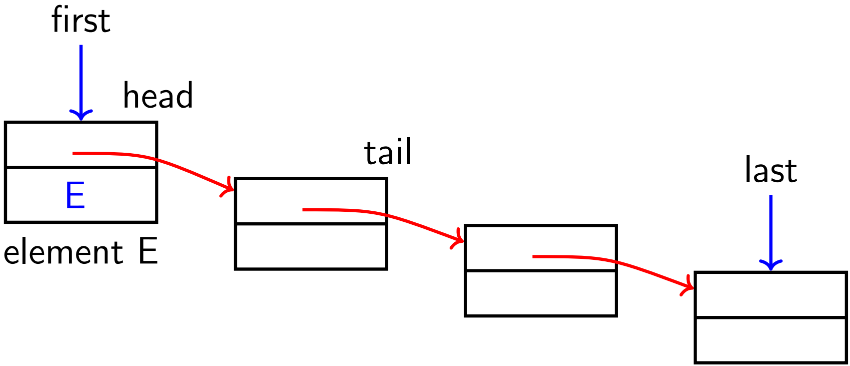


Illustration: Defining a Linked List in Java

```
public class LinkedList {
  private Node first;
  private Node last;

  public LinkedList() {
    last = first = null;
  }
  public void add(Object element) {
    // add element to the end
  }
  public void remove() {
    // remove element from the front
  }
  public Object getHead() {
    // return the head of the list
  }
  public LinkedList getTail() {
    // return the tail of the list
  }
}
```

Illustration: Defining a Node

```
public class Node {
  private Node next;
  private Object element;

  public Node(Object element, Node next) {
    this.element = element;
    this.next = next;
  }

  public Object getElement() { return element; }
  public void setElement() {
    this.element = element;
  }
  public Node getNext() { return next; }
  public void setNext(Node next) {
    this.next = next;
  }
}
```

Even better: Generics!

```
public class LinkedList<T> {
  private Node<T> first;
  private Node<T> last;

  //...
```

```
public class Node<T> {
  private Node<T> next;
  private T element;

  //...
```

Design Concerns: Memory Leaks

- Memory leaks are serious programming errors – hard to debug
- They happen when references are maintained to objects no longer in use
- The garbage collector cannot reclaim the space

Illustration: Memory Leaks

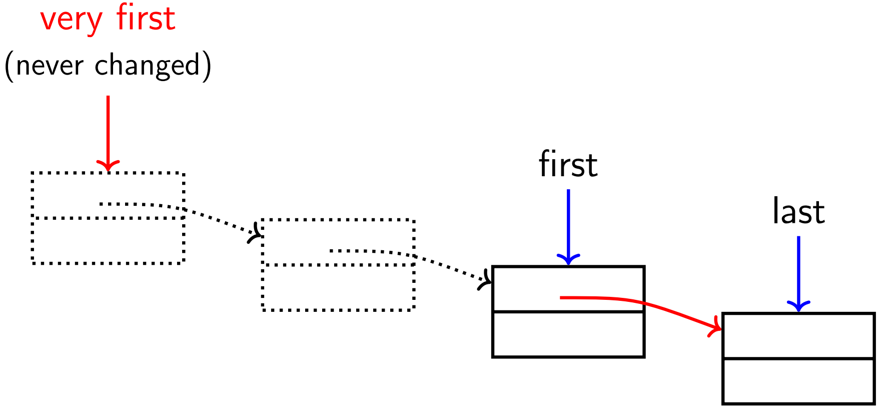
- Consider a new constructor creating a very first node

```
public LinkedList(Object element) {
  Node veryFirst = new Node(element, null);
  first = last = veryFirst;
}
```

- What if very first is a private field of LinkedList?

```
public LinkedList(Object element) {
  veryFirst = new Node(element, null);
  first = last = veryFirst;
}
```

Illustration: Memory Leaks



Design Concerns: Entanglement

- Assignment statements such as x = y result in both x and y referring to the same object
- changes carried out by invoking a method x.set(v)
  - alter the object
  - are visible to y when invoking y.get()
- Often this is not the desired outcome

Object Cloning

Cloning is a powerful, but controversial, feature in Java

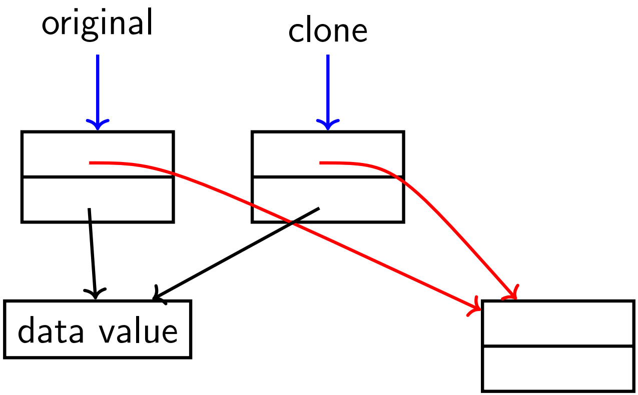
- Java provides two forms of cloning
  - *shallow* – a new object is created and the fields of the original are copied without change
    Default strategy provided by `Object.clone()`
  - *deep* – a new object is created and cloning is applied recursively to each field
    Must override `clone()` to implement this strategy
- Great care needs to be exercised to apply it correctly
- It is generally recommended to provide a *copy constructor*

Illustration: Incorrect Cloning

```
@Override
public Node clone() {
  try {
    return (Node) super.clone();
  }
  catch (CloneNotSupportedException e) {
    throw new InternalError(e.toString());
  }
}
```

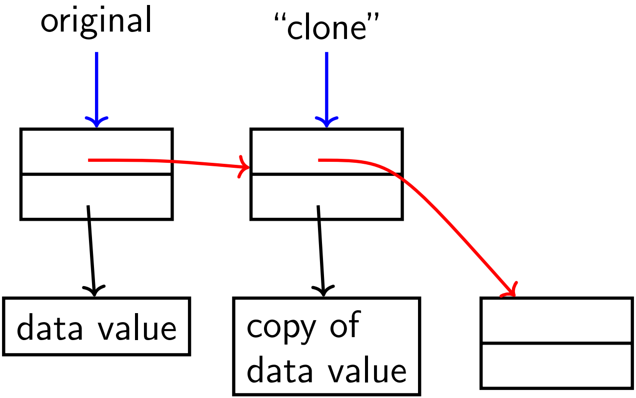
Using default clone implementation will just copy all fields. (This can be okay if all fields are primitives and/or immutable objects.)

Illustration: Incorrect Clone Result



Fields are just blindly duplicated.

Illustration: Desired Copy Action



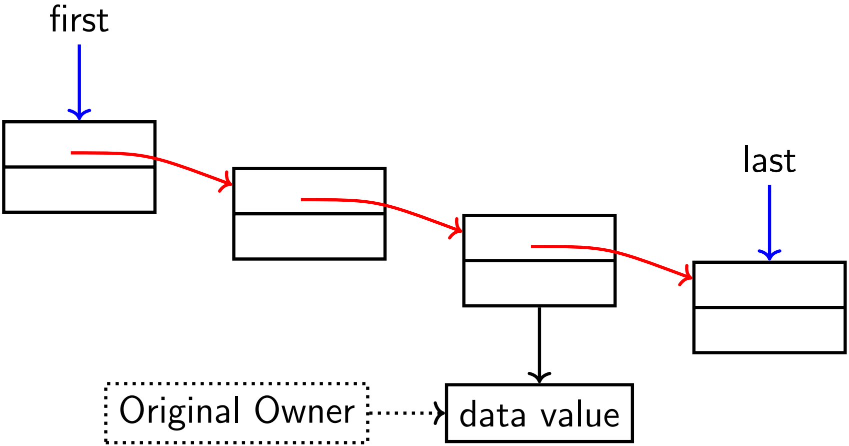
This is not cloning, rather tailored object duplication.

Design Concerns: Encapsulation Cracks

The object creator retaining access:

- exposes access to the data structure internals
- introduces side effects that break the class contract
- cloning/copying is one way to avoid this design flaw

Illustration: Creator Retaining Access



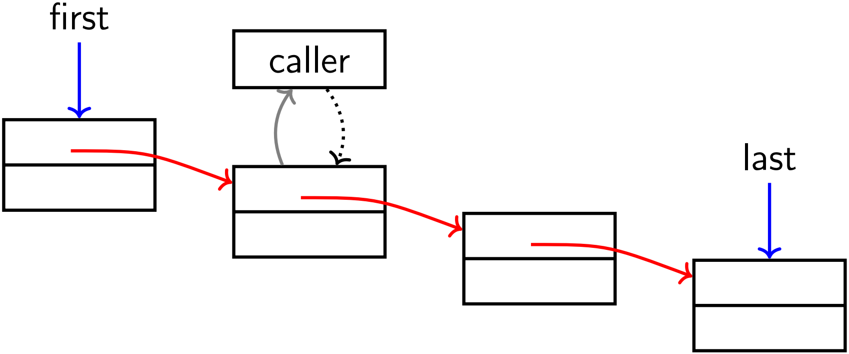
Original owner of data may still have access to it.

Design Concerns: Encapsulation Cracks

An internal object is returned to the caller

- providing access to the data structure internals
- enabling side effects that break the class contract
- creating the potential to break the integrity of the data structure

Illustration: Returning Internal Object



Returned node becomes accessible to caller.

Collections

Collections or *containers*:

- allow programmers to hold and organize sets of objects
- . . . in useful and efficient ways
- . . . as part of consistent and flexible framework
- Java provides us with many standard collection interfaces and implementations

Core Collection Interfaces

- Collection – root of the collection hierarchy
- Set – no duplicates
- List – ordered collection, sequence
- Queue – holds elements for processing
- Deque – double ended queue
- Map – maps keys to values

Collection Interface

- size, isEmpty
- contains
- add, addAll
- remove, removeAll, retainAll, clear
- toArray
- iterator (because Collection implements Iterable)

Optional methods that are not supported by a specific implementation throw `UnsupportedOperationException` (read the documentation!)

List

- add, addAll – add to end of the list
- remove – removes first occurrance
- iterator, listIterator
- indexOf, lastIndexOf – find index of elememt
- get, set – access element at given index
- subList – view portion of list as a List

List

- add, addAll – add to end of the list
- remove – removes first occurrance
- iterator, listIterator
- indexOf, lastIndexOf – find index of elememt
- get, set – access element at given index
- subList – view portion of list as a List

What data structure could you use to implement a List?

Queue

- Insert – add, offer
- Remove – remove, poll
- Examine – element, peek

Queues usually use FIFO order. `PriorityQueue` will use natural ordering or a Comparator.

Queue

- Insert – add, offer
- Remove – remove, poll
- Examine – element, peek

Queues usually use FIFO order. `PriorityQueue` will use natural ordering or a Comparator.

What data structure could you use to implement a Queue?

Map

Maps keys to values. Does not implement `Collection` itself, but has three *collection views*

- `keySet` – Set of the keys
- `values` – Collection of values
- `entrySet` – Set of key-value mappings.

Beware of using mutable objects as keys!

- put – associate a key with a value
- get – get value associated with key
- remove – remove key/value mapping
- containsKey, containsValue

Map

Maps keys to values. Does not implement `Collection` itself, but has three *collection views*

- `keySet` – Set of the keys
- `values` – Collection of values
- `entrySet` – Set of key-value mappings.

Beware of using mutable objects as keys!

- put – associate a key with a value
- get – get value associated with key
- remove – remove key/value mapping
- containsKey, containsValue

What data structure could you use to implement a Map?

Iterator and Iterable

Iterator has three methods

- hasNext – Are there more elements?
- next – Return the next element
- remove – Remove the last element returned (optional)

Iterable only has one method

- iterator – returns an Iterator
- Implementing this interface allows object to be target of for-each

Iterator

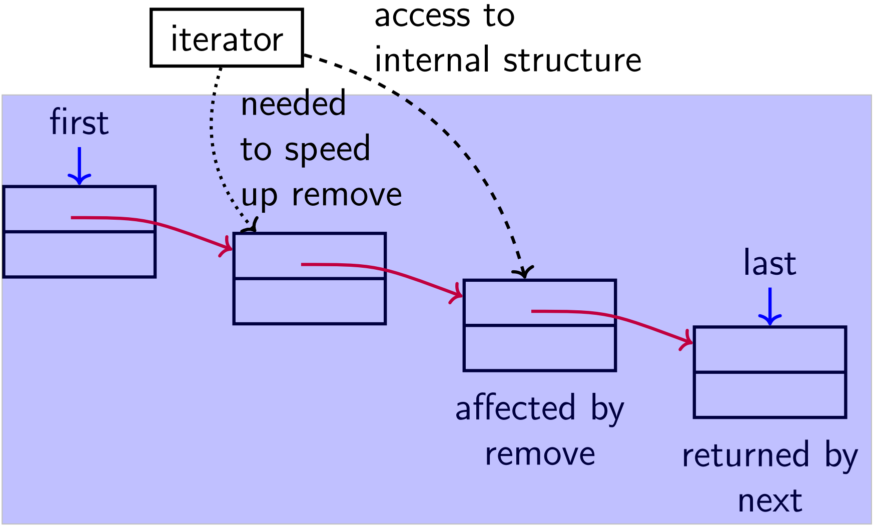
- The ability to examine an entire collection of objects is a helpful feature
- Java provides the Iterable interface specifically for this purpose
- A specific iterator may or may not take a snapshot first
- If not, use of other methods may affect the results

Iterator: Protecting Encapsulation

The iterator:

- is made available outside the collection
- does not reveal anything about the internal organization
- has access to the internal organization of the data

Iterator: Protecting Encapsulation



Data Structure Design

Research has led to the development of an arsenal of specialized data structures

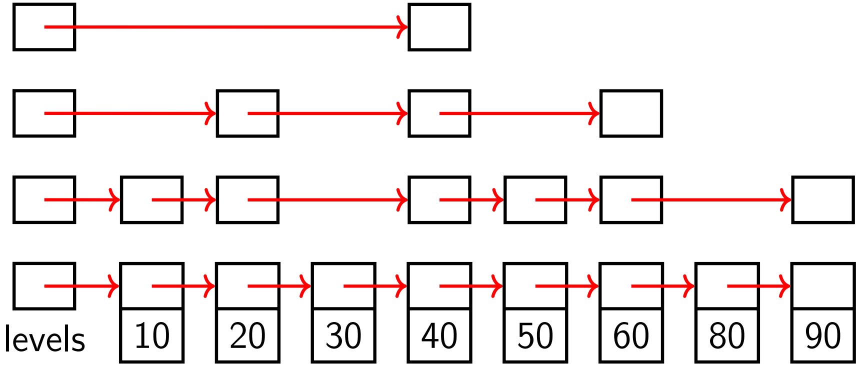
- Search and use them when developing a new program
- Don't reinvent the wheel

Example: Skip List

One such data structure is a Skip List:

- optimized for fast search in ordered lists
- multiple linked lists form express lanes supporting list traversal
- stops along each lane are determined using some probability p

Illustration: Skip List



Example: Game of Life

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970.

- Given a finite, two-dimensional, orthogonal grid of square cells.
- Each cell is in one of two possible states, Live or Dead.
- The initial pattern constitutes the 'seed' of the system.

Example: Game of Life

- Every cell interacts with its eight neighbors.
- At each step in time:
    - any live cell with fewer than two live neighbors dies, as if by loneliness
    - any live cell with more than three live neighbors dies, as if by overcrowding
    - any dead cell with exactly three live neighbors comes to life
- A new generation is created by applying the rules simultaneously to every cell.

Implementing Game of Life

How do we represent the game?

Implementing Game of Life

How do we represent the game?

- Simple implementation: Two dimensional boolean array N by N

Implementing Game of Life

How do we represent the game?

- Simple implementation: Two dimensional boolean array N by N
- What if N is very large?

Implementing Game of Life

How do we represent the game?

- Simple implementation: Two dimensional boolean array N by N
- What if N is very large?
- What if game is very sparse?