Announcements

- Tutorials: TBA
- Office hours: Thursday 2:00pm - 4:00pm

Data structures

- Abstract Data Types (ADTs)
  - An ADT is a type representing a data-structure, with some operations on its elements.
  - Separating interface from implementation: A given ADT may be implemented using different underlying data-structures. For example, a *set* could be implemented as an array, a Vector, a *linked-list*, etc.

- A *dynamic data-structure* is a data-structure which can change.

- A *collection* is an ADT which supports operations for adding and removing elements (hence it is a dynamic data-structure.)

- A dynamic data-structure has a variable size, in contrast with an array or an object which have a fixed size.

- "Adding" to an array modifies the array, but it doesn't change its overall structure. "Growing" an array changes its overall structure.

The List ADT

- A list is an abstract data-type
- A list is a collection
- A list is a dynamic data-structure
- List operations:
  - Adding an element
  - Removing an element
  - Obtaining an element
  - Length
- Possible implementations
  - Arrays
  - Growing arrays
  - Vectors
  - Linked-lists

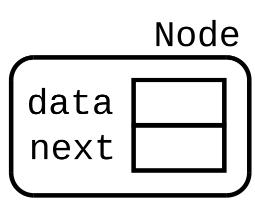
Linked Lists

- A *linked-list* is a dynamic data-structure consisting of a sequence of objects called *nodes*, where each node has a reference or link to the next node in the sequence.

- A linked-list is a collection.

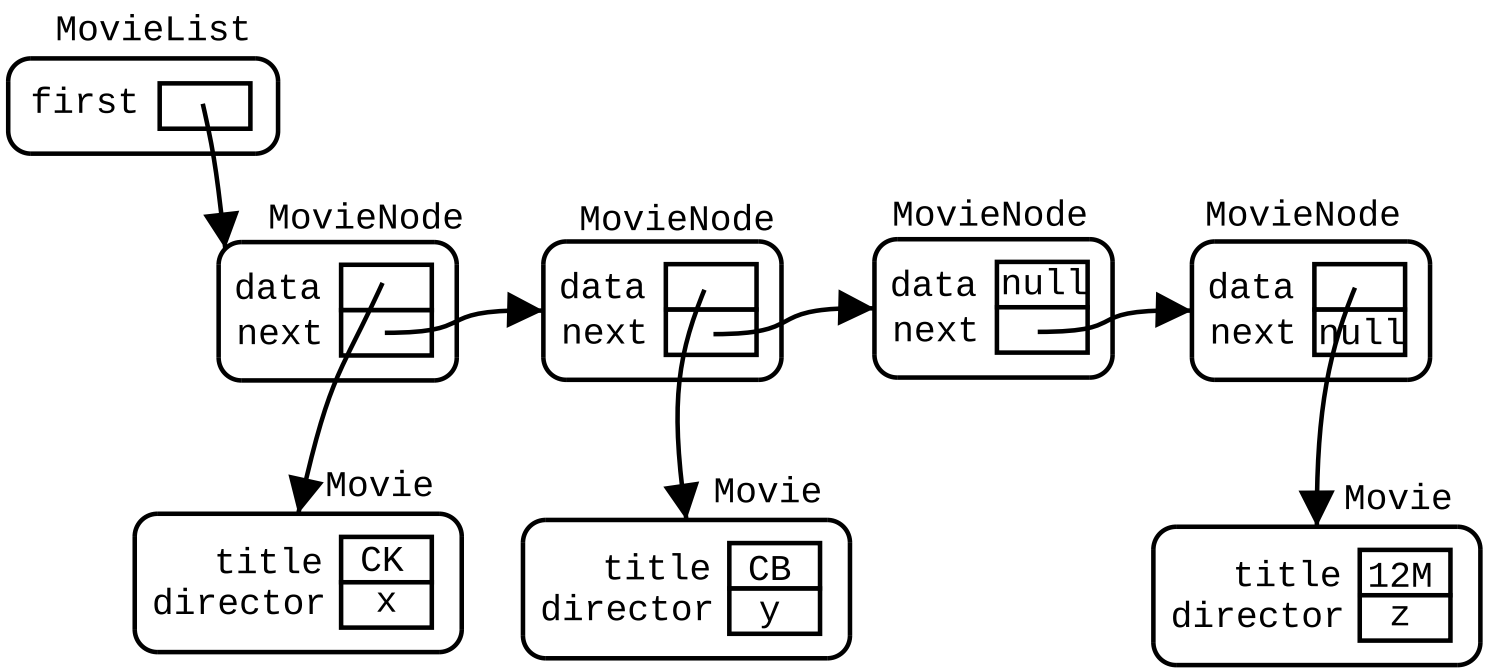
- Nodes are a recursive data-structure

```
class Node {
  String data;
  Node next;
}
```



- A recursive data-structure has references to objects of its own type

Linked Lists



Linked Lists

```
class MovieNode {
  private Movie data;
  private MovieNode next;

  public MovieNode(Movie m, MovieNode n) {
    data = m;
    next = n;
  }
  public Movie get_movie() { return data; }
  public MovieNode get_next() { return next; }
  public void set_movie(Movie m)
  {
    data = m;
  }
  public void set_next(MovieNode n)
  {
    next = n;
  }
}
```

Linked Lists

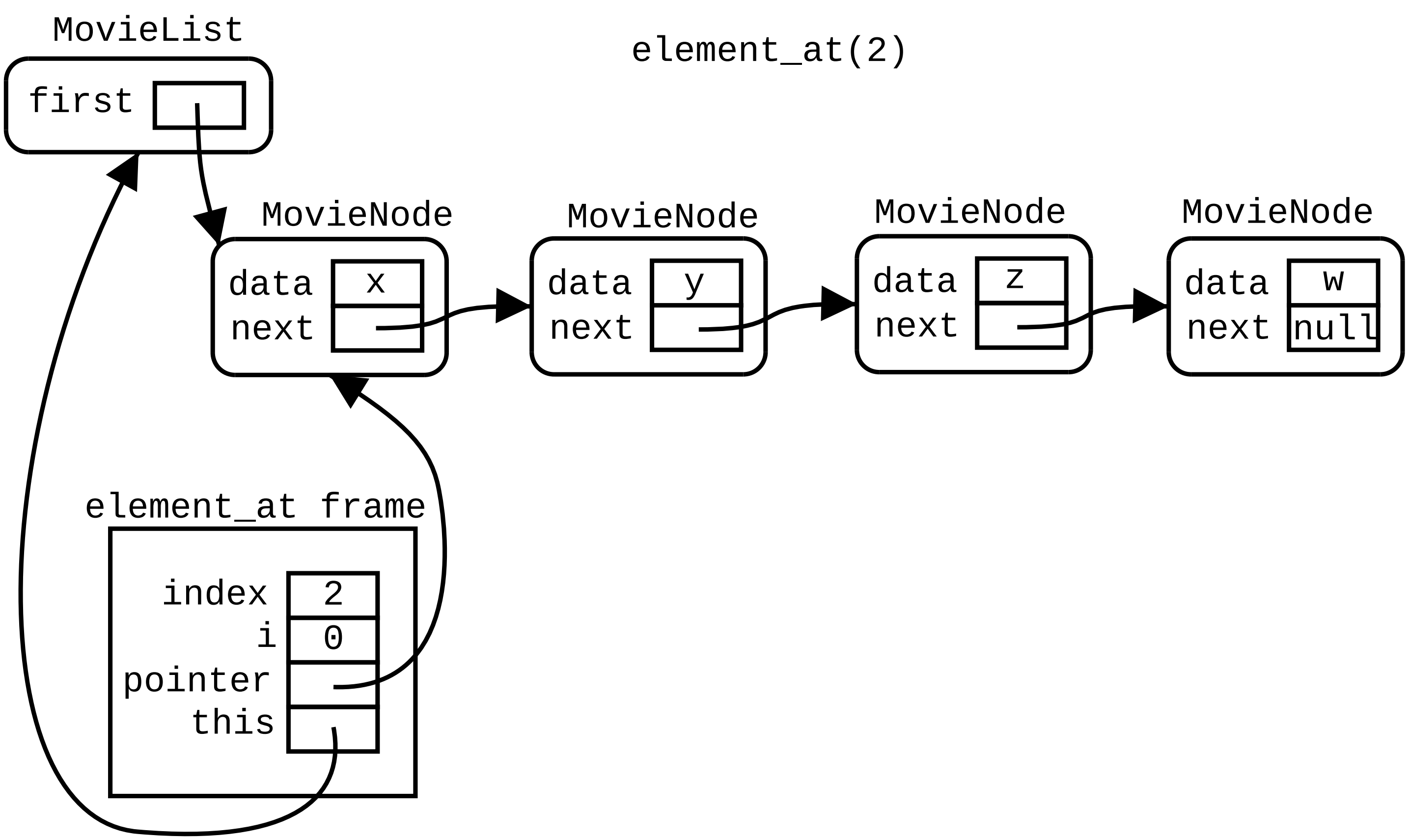
```
class MovieList {
  private MovieNode first;
  //...
  public Movie element_at(int index)
  {
    // ...
  }
}
```

- To return the element at the given index:
  - Jump from node to node
  - until we have counted up to the given index.

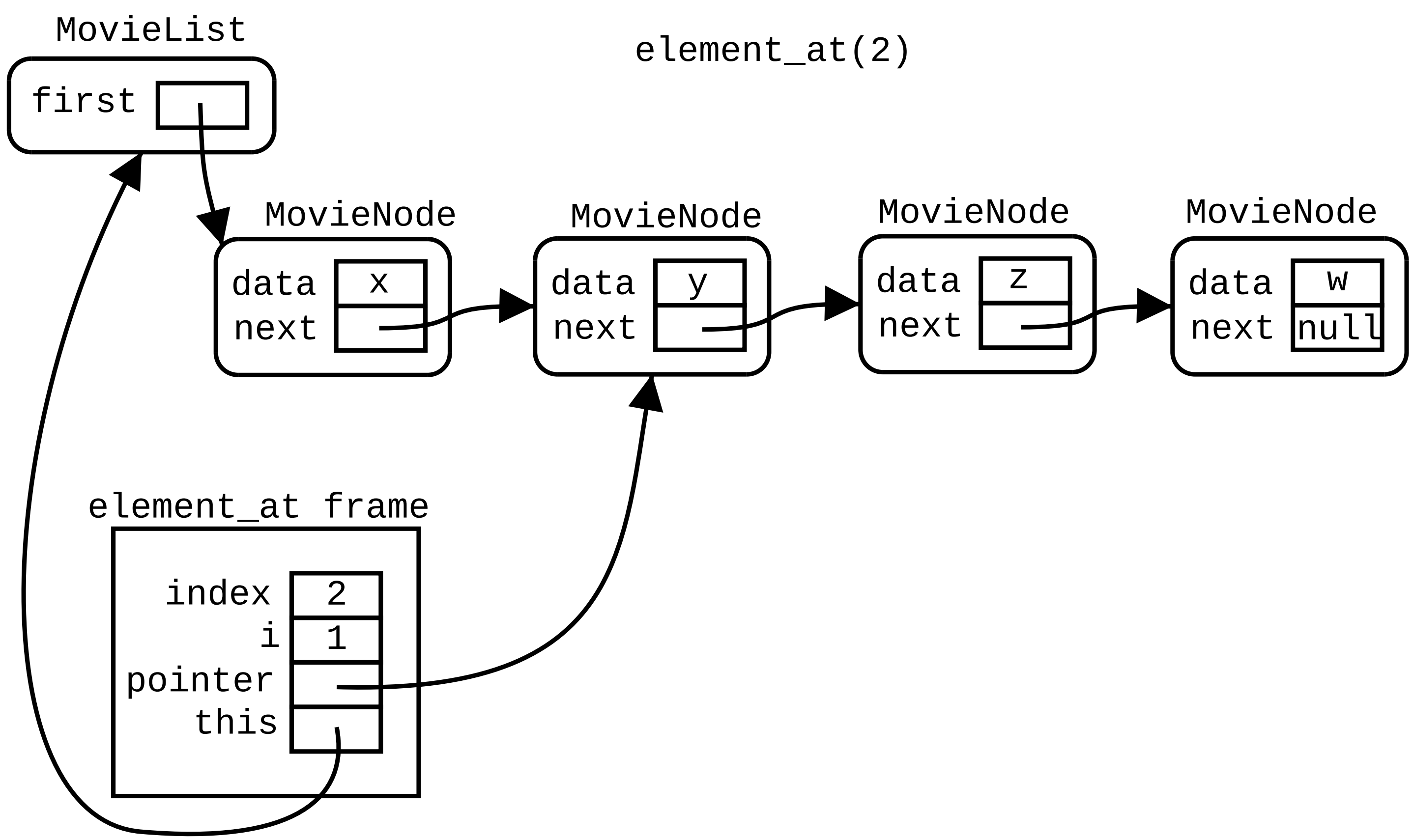
Linked Lists

```
class Test {
  public static void main(String[] args)
  {
    MovieList l = new MovieList();
    Movie w = new Movie(“abc”,“def”);
    Movie x = new Movie(“bca”,“efd”);
    Movie z = new Movie(“cba”,“fef”);
    Movie y = new Movie(“xxx”,“yyy”);
    l.add(w);
    l.add(z);
    l.add(y);
    l.add(x);
    Movie m = l.element_at(2);
  }
}
```

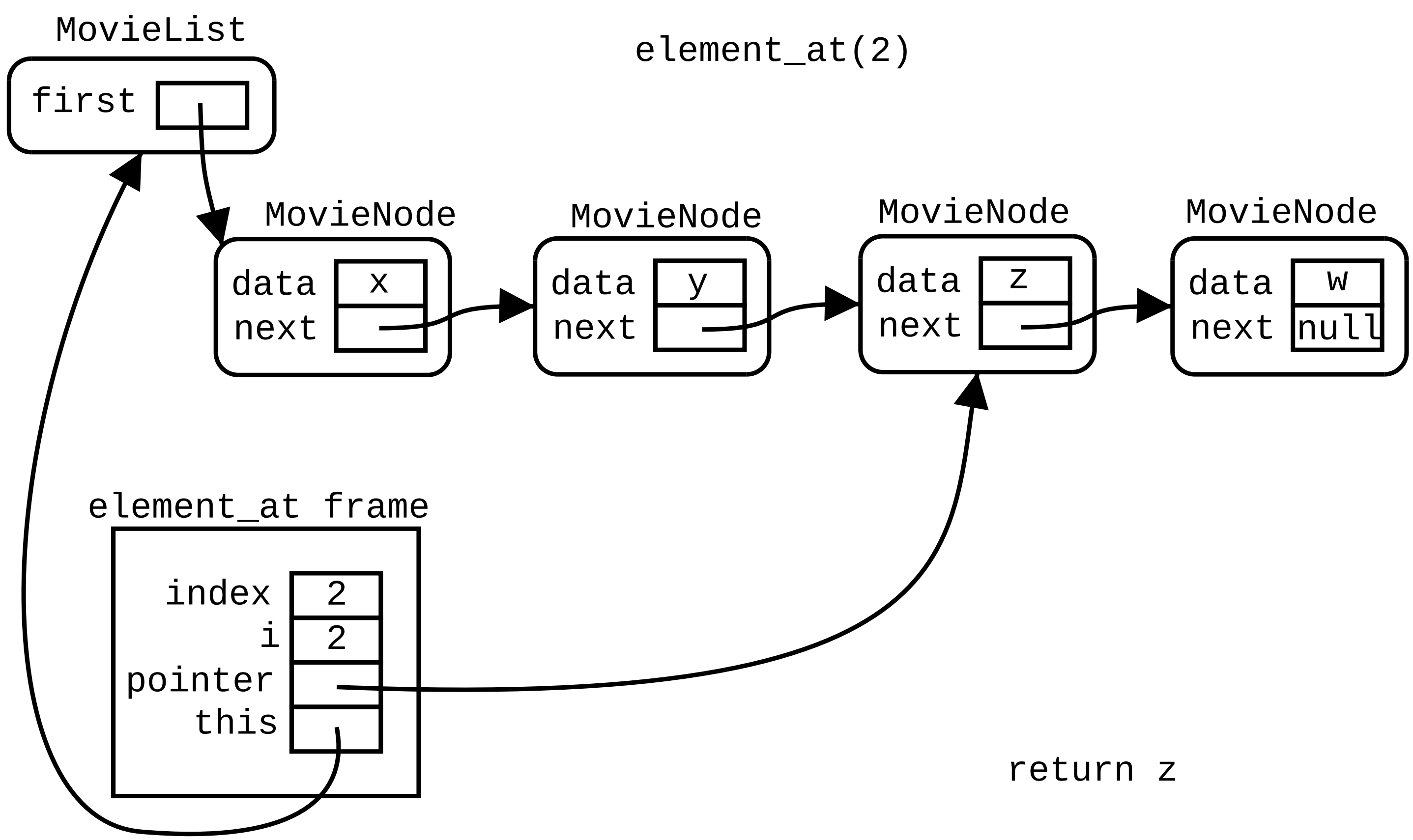
Linked Lists



Linked Lists



Linked Lists



Linked Lists

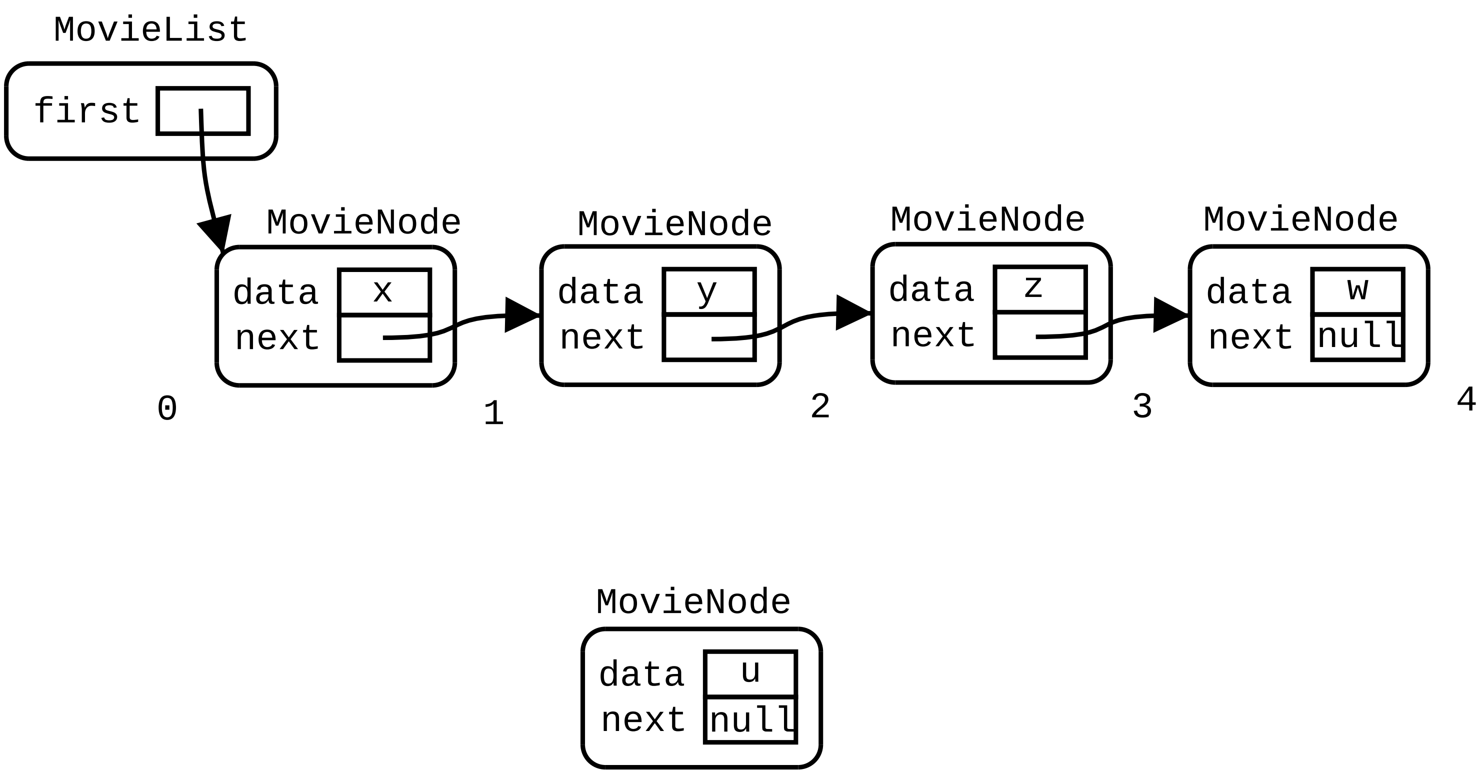
```
class MovieList {
  private MovieNode first;
  //...
  public Movie element_at(int index)
  throws IndexOutOfBoundsException
  {
    if (index < 0)
      throw new IndexOutOfBoundsException();
    int i = 0;
    MovieNode pointer = first;
    while (pointer != null && i < index) {
      pointer = pointer.get_next();
      i++;
    }
    if (pointer == null)
      throw new IndexOutOfBoundsException();
    return pointer.get_movie();
  }
}
```

Linked Lists

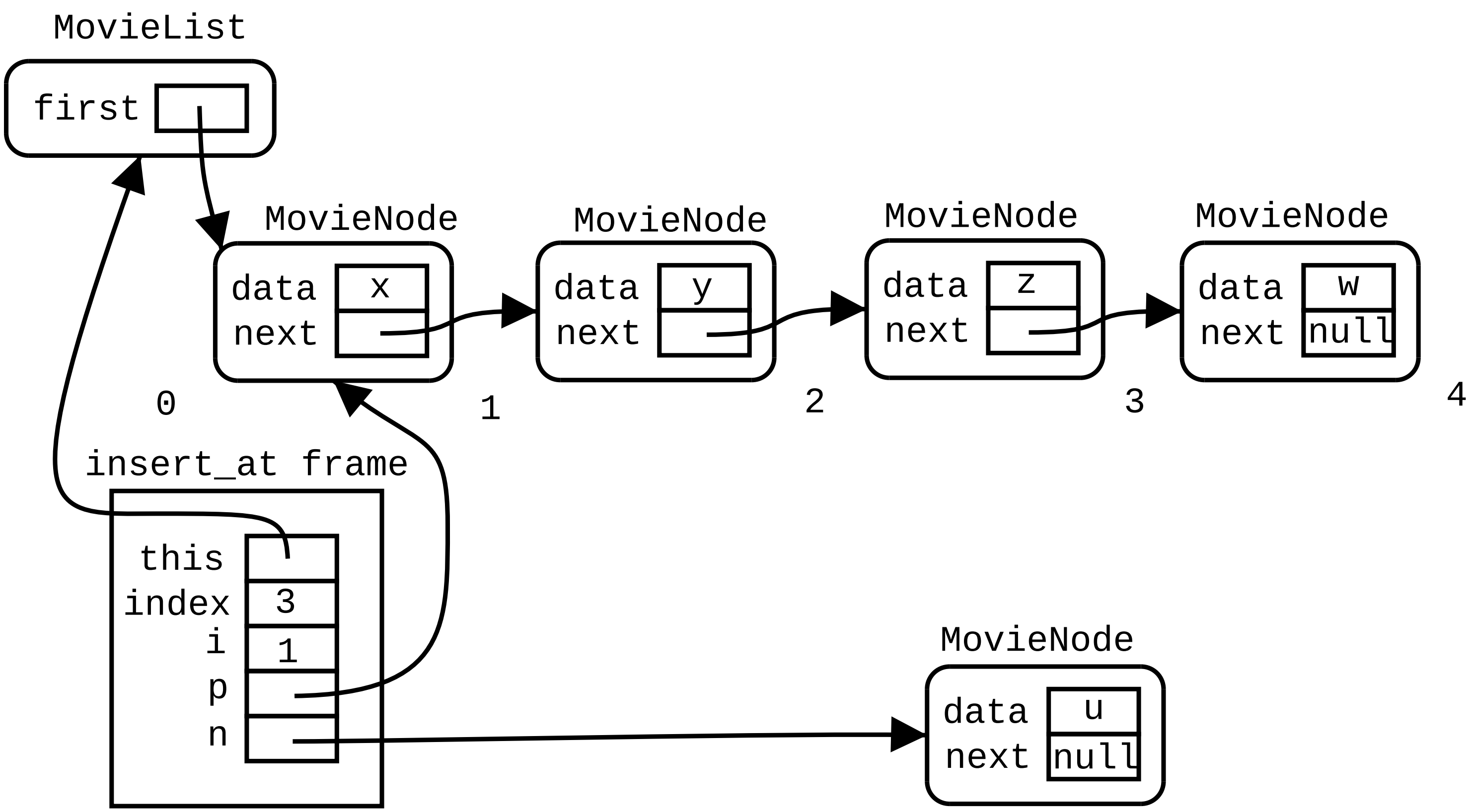
```
class MovieList {
  private MovieNode first;
  //...
  public void insert_at(Movie m, int index)
  {
    // ...
  }
}
```

- To insert the element at the given index:
  - Jump from node to node
  - until we have counted up to the given index,
  - remembering the "previous" node, and
  - updating the "next" of the "previous" node and the "new" node.

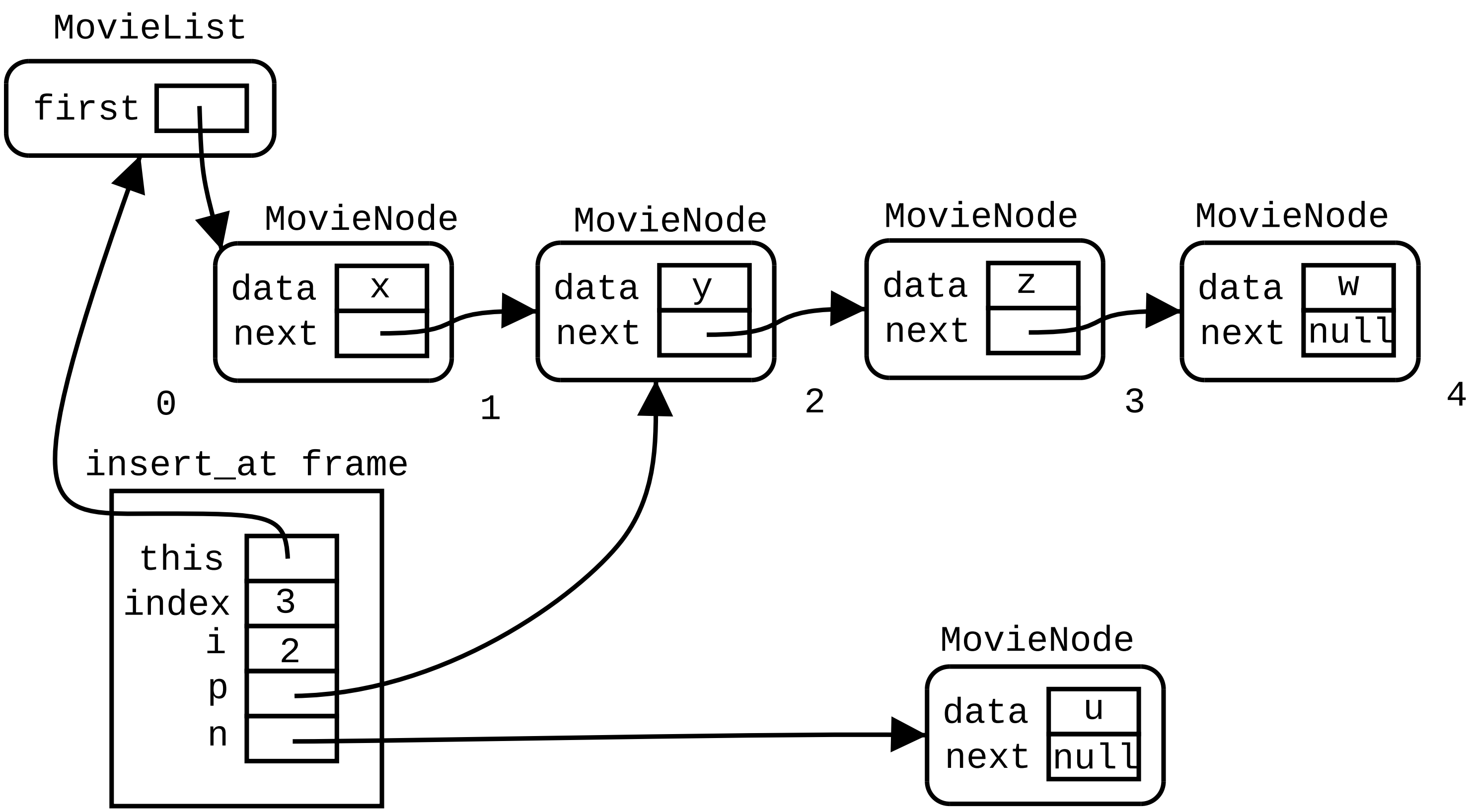
Linked-lists



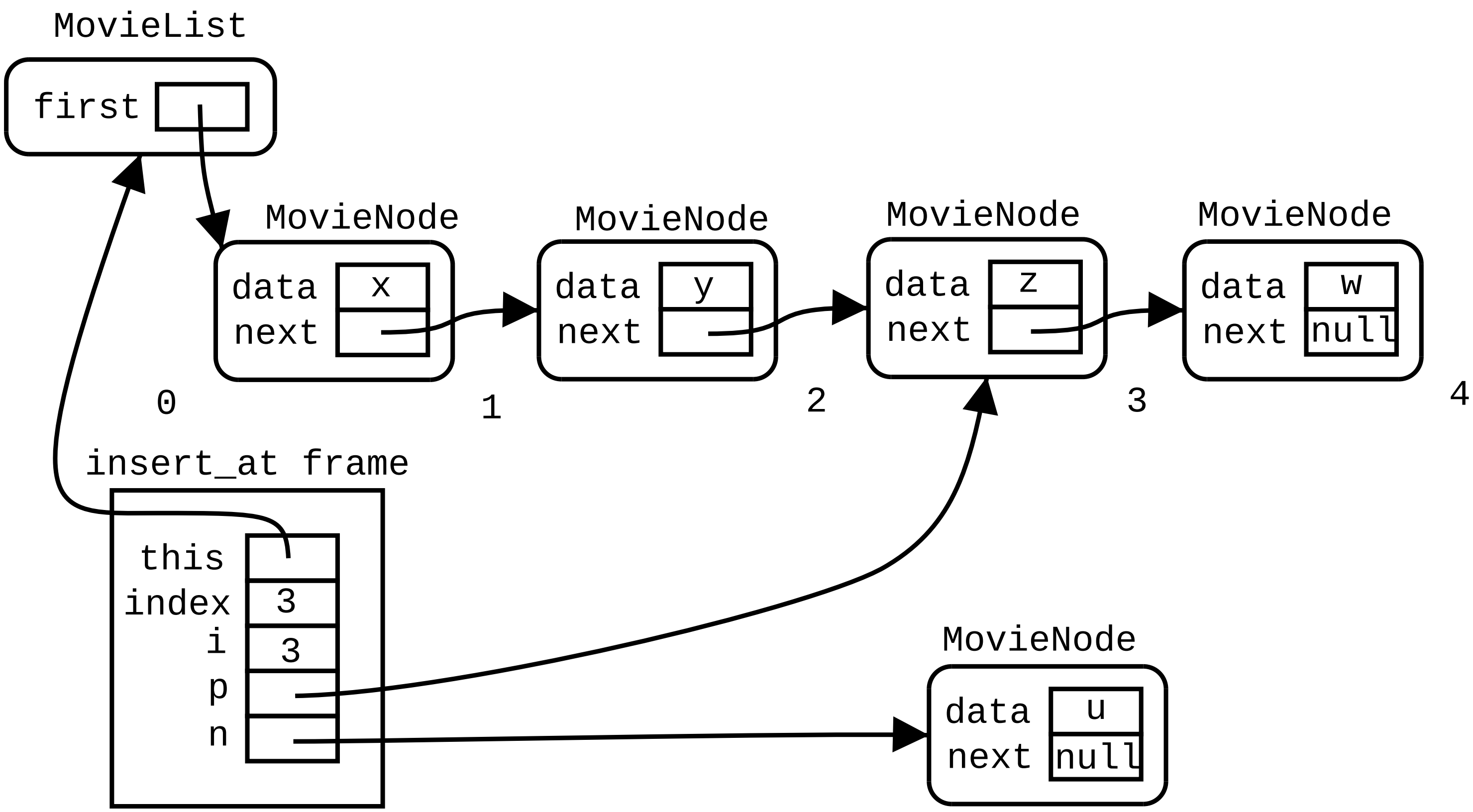
Linked-lists



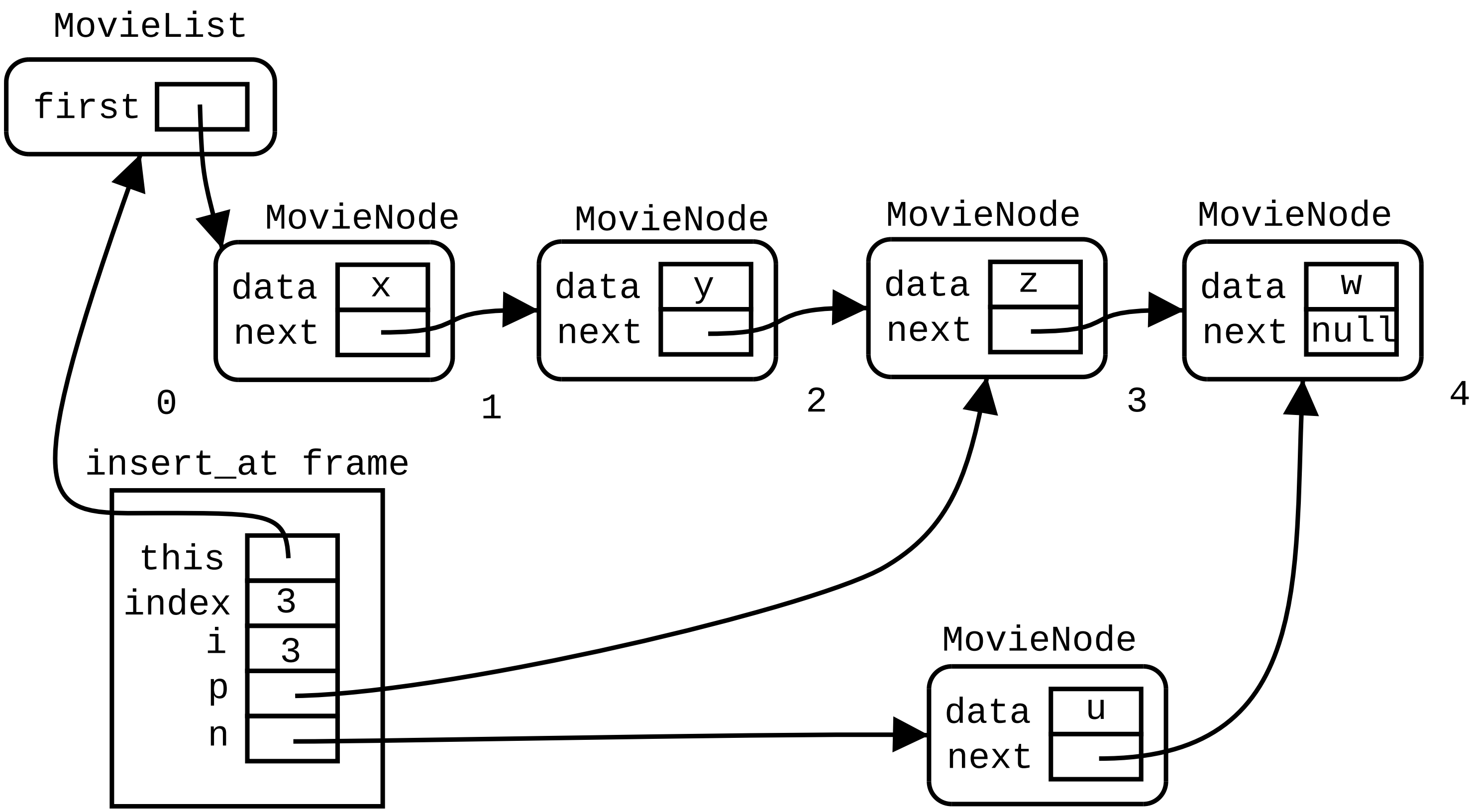
Linked-lists



Linked-lists

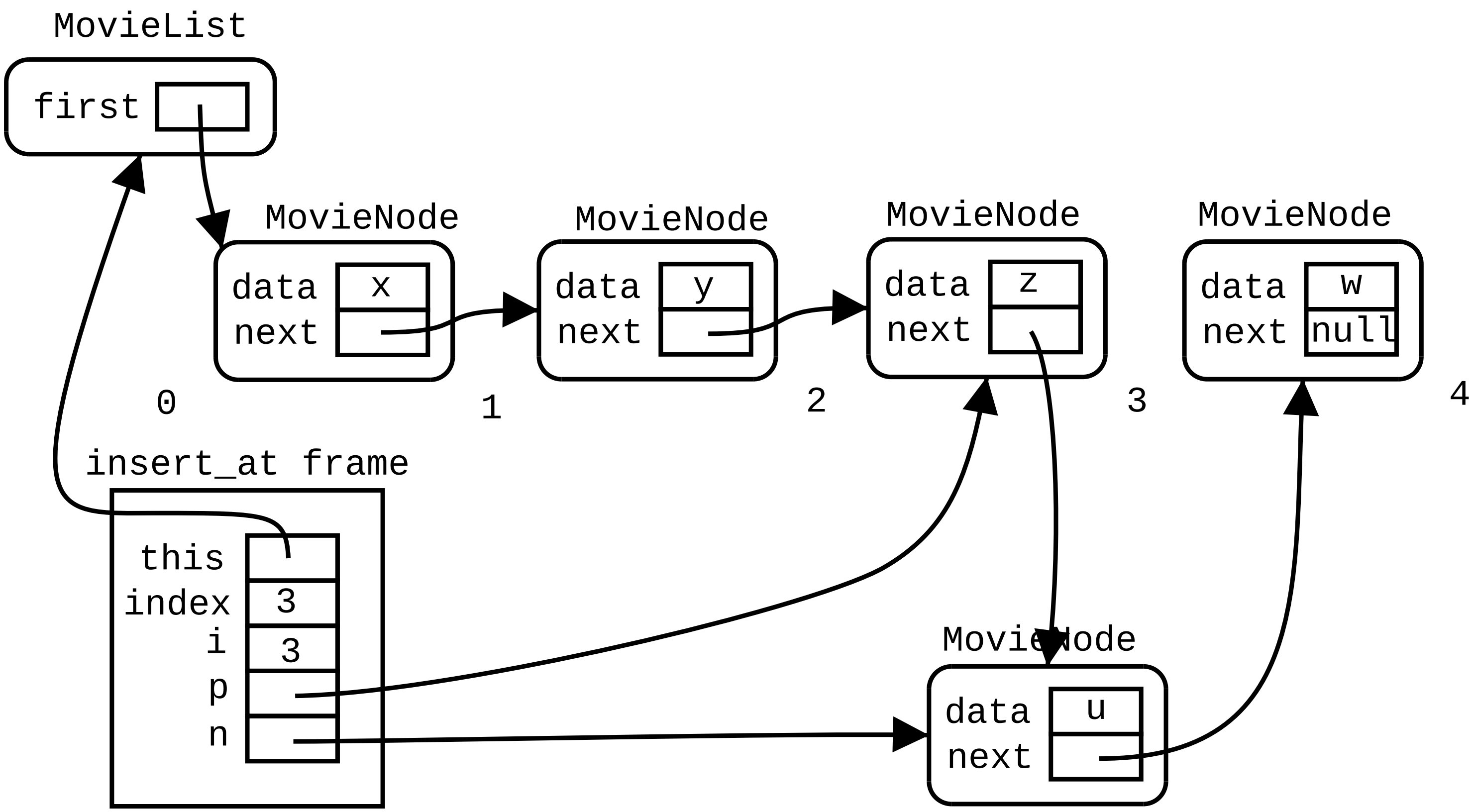


Linked-lists

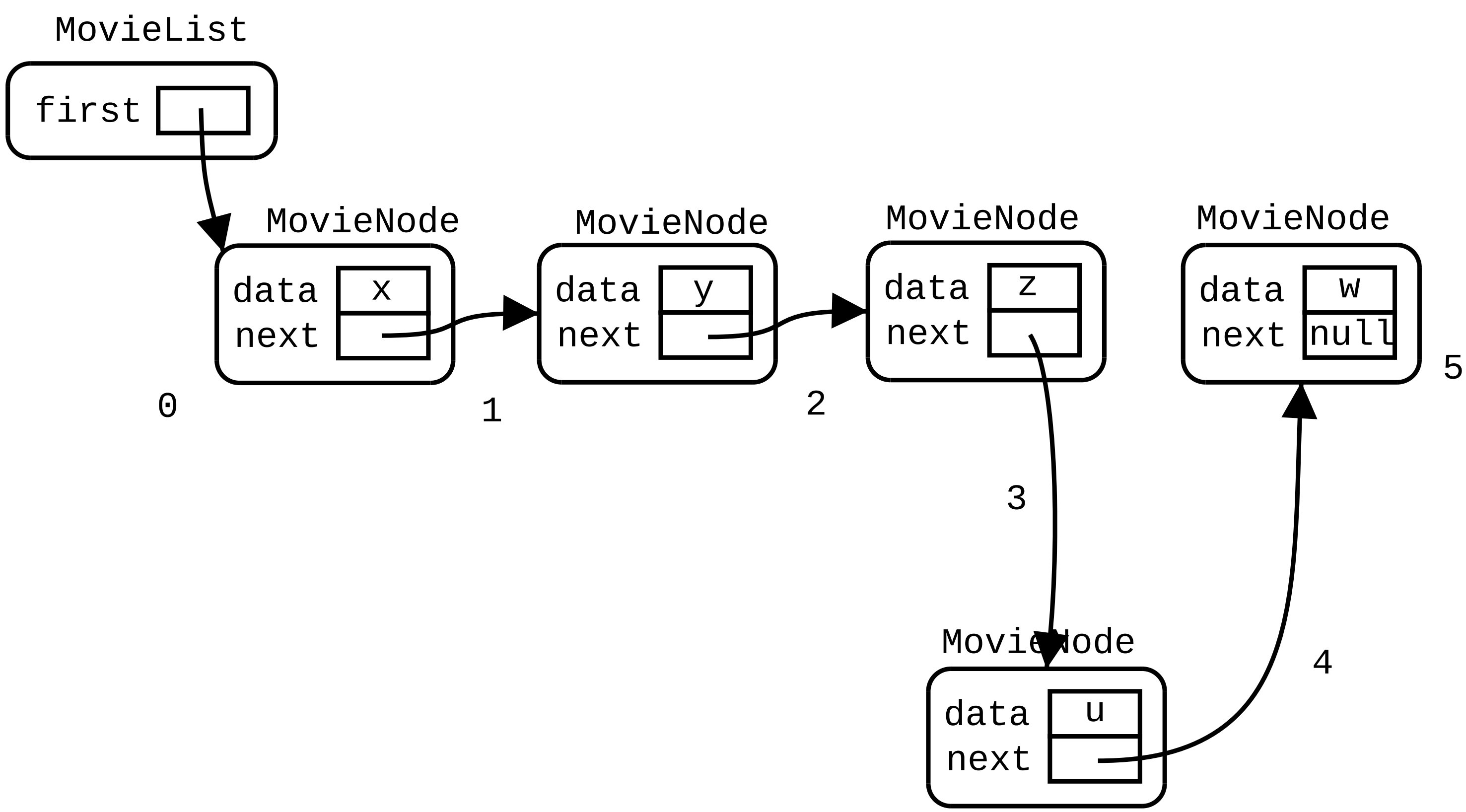


McGill

Linked-lists



Linked-lists



Linked-lists

```
public void insert_at(Movie m, int index)
throws IndexOutOfBoundsException {
  if (index < 0)
    throw new IndexOutOfBoundsException();
  MovieNode n = new MovieNode(m, null);
  if (index == 0) {
    n.set_next(first);
    first = n;
  }
  else {
    MovieNode p = first;
    int i = 1;
    while (i < index && p != null) {
      p = p.get_next();
      i++;
    }
    if (p == null)
      throw new IndexOutOfBoundsException();
    n.set_next(p.get_next());
    p.set_next(n);
  }
}
```

Review

- Linked-lists: nodes with data and pointer to the next
- Traversing a linked-list

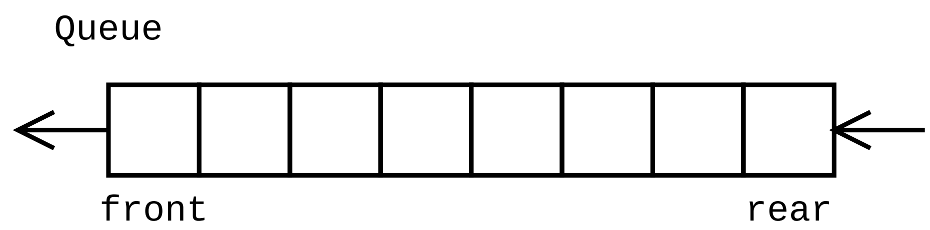
```
Node p = first;
while (p != null && ...) {
  //...
  p = p.get_next();
}
```

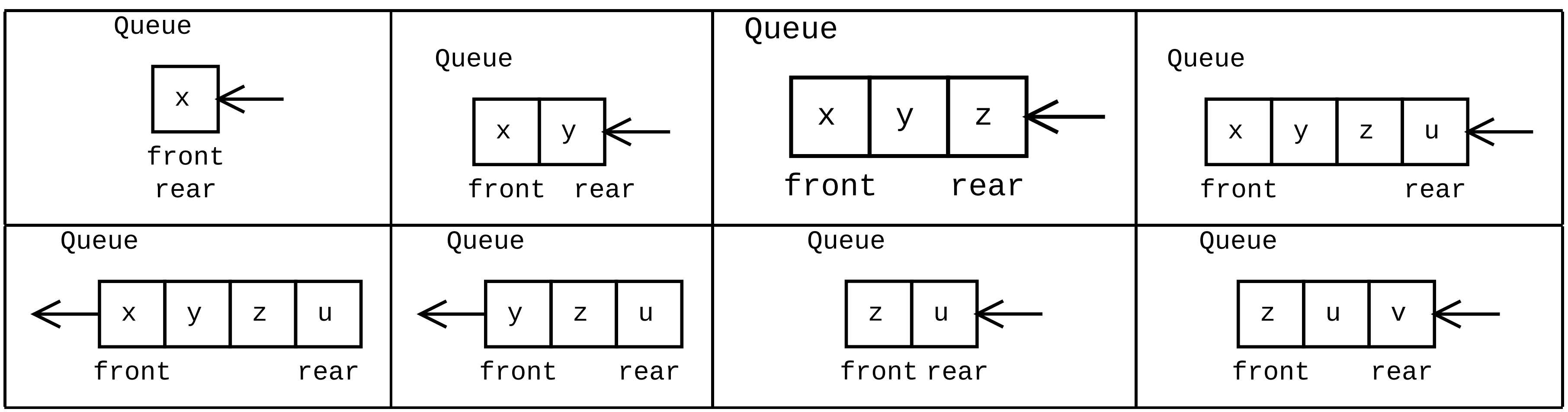
- Difference between a linked-list and an array
  - An array has a fixed size
  - A linked-list is dynamic: its size increases each time we add a new element and we don't have to worry about running out of space

Queues and Stacks

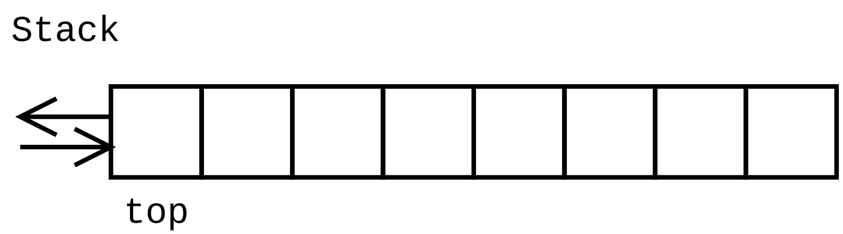
- Queues and Stacks are ADTs representing linear collections with particular operations
- A *queue* (FIFO) is a (dynamic) linear collection with (at least) the following operations:
  - *enqueue*: adds an item at the end of the sequence
  - *dequeue*: removes the first item of the sequence
  - *peek*: gets the first item of the sequence without removing it
  - *isempty*: returns true if the sequence has no items
- A *stack* (LIFO, or FILO) is a (dynamic) linear collection with (at least) the following operations:
  - *push*: adds an item at the "top" of the sequence
  - *pop*: removes the "top" item of the sequence
  - *top*: returns the top item without removing it
  - *isempty*: returns true if the sequence has no items

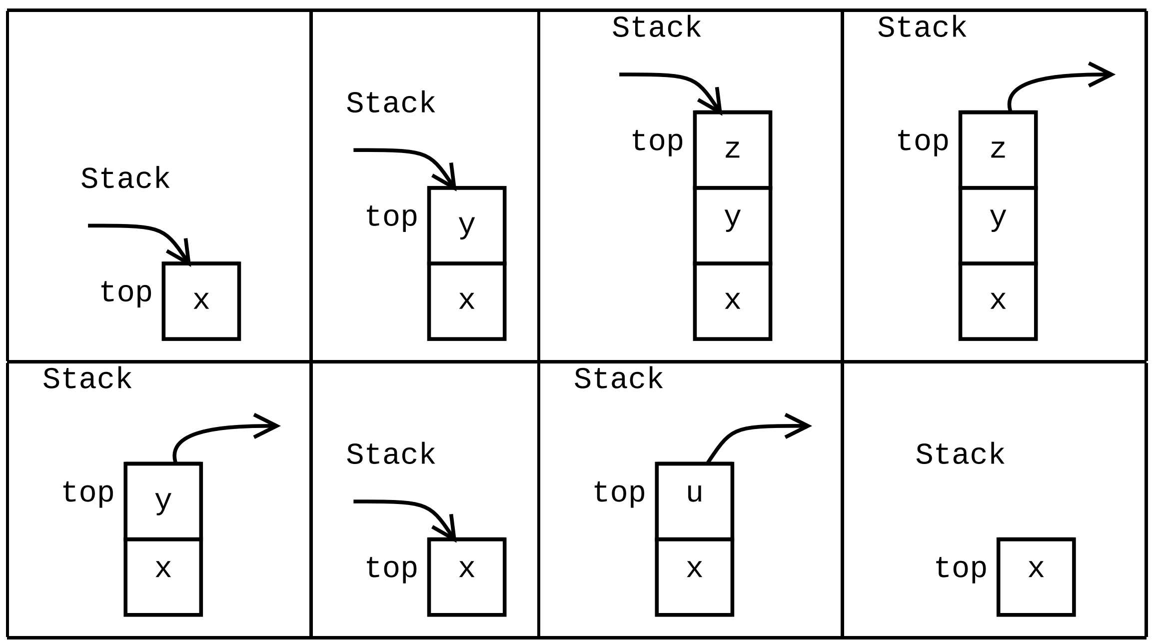
Queues


Queue
front
rear


Queue
x
front
rear
Queue
x y
front rear
Queue
x y z
front rear
Queue
x y z u
front rear
Queue
x y z u
front rear
Queue
y z u
front rear
Queue
z u
front rear
Queue
z u v
front rear

Stacks


Stack
top


Stack
top
x
Stack
top
y
x
Stack
top
z
y
x
Stack
top
z
y
x
Stack
top
y
x
Stack
top
x
Stack
top
u
x
Stack
top
x

Queues and Stacks

- Queues and Stacks are ADTs so they can be implemented in many different ways
- For example, they could be implemented using
  - Linked-lists
  - Arrays
  - Doubly-linked-lists
  - Linked-lists with front and rear pointers
  - Vectors/Growing arrays
  - "Circular" arrays
- Each implementation has advantages and disadvantages with respect to efficiency
- Queues implemented as fixed-size circular arrays are commonly called *buffers*

Implementing Queues

```
class LinkedList {
  //...
  public LinkedList() { ... }
  public void insert_at(Object o, int index) { ...
  public void remove_at(int index) { ... }
  public Object element_at(int index) { ... }
  public int length() { ... }
}
```

Implementing Queues

```
class Queue {
  private LinkedList list;
  public Queue() { list = new LinkedList(); }
  public void enqueue(Object obj)
  {
    list.insert_at(obj, list.length());
  }
  public void dequeue()
  {
    list.remove_at(0);
  }
  public Object peek()
  {
    return list.element_at(0);
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Stacks

```
class Stack {
  private LinkedList list;
  public Stack() { list = new LinkedList(); }
  public void push(Object obj)
  {
    list.insert_at(obj, 0);
  }
  public void pop()
  {
    list.remove_at(0);
  }
  public Object top()
  {
    return list.element_at(0);
  }
  public boolean isempty()
  {
    return list.length() == 0;
  }
}
```

Implementing Stacks

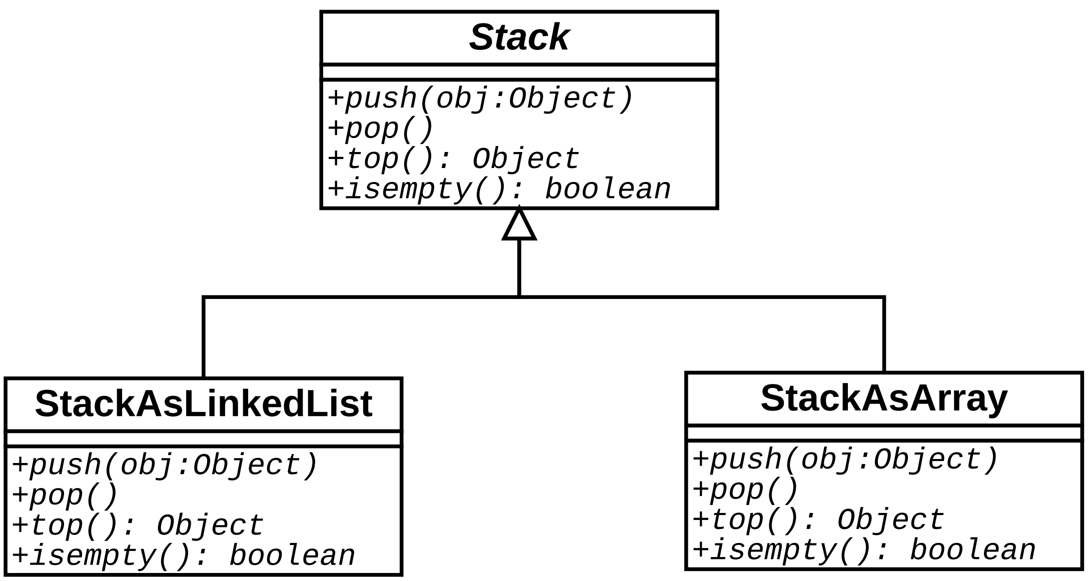
```
class Stack {
  private Object[] list;
  private int top;

  public Stack()
  {
    list = new Object[1000];
    top = 0;
  }
  public void push(Object obj)
  {
    if (top >= list.length)
      grow_array(100);
    list[top] = obj;
    top++;
  }
```

Implementing Stacks

```
  public void pop()
  {
    top--;
  }
  public Object top()
  {
    return list[top];
  }
  public boolean isempty()
  {
    return top == 0;
  }
  private void grow_array(int n)
  {
    ...
  }
}  // End of Stack
```

ADTs and abstract classes

- Stacks and queues constrain the operations on a list
- An ADT should be declared as an interface or abstract class, and the concrete implementations should implement the interface or extend the abstract class



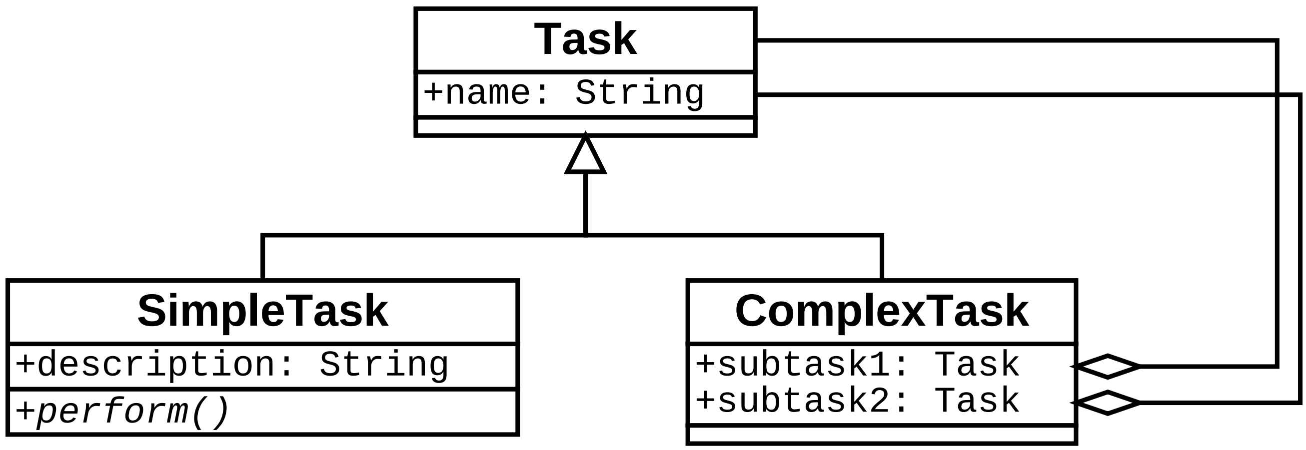
Applications (Simulation)

```
class Customer { ... }
class SuperMarket {
  Queue line;
  SuperMarket() { line = new Queue(); }
  void process(Customer c) { ... }
  void run()
  {
    while (true) {
      int coin = (int)(Math.random() * 2);
      if (coin == 1) {
        Customer first = line.peek();
        process(first);
        line.dequeue();
      }
      else {
        line.enqueue(new Customer());
      }
    }
  }
}
```

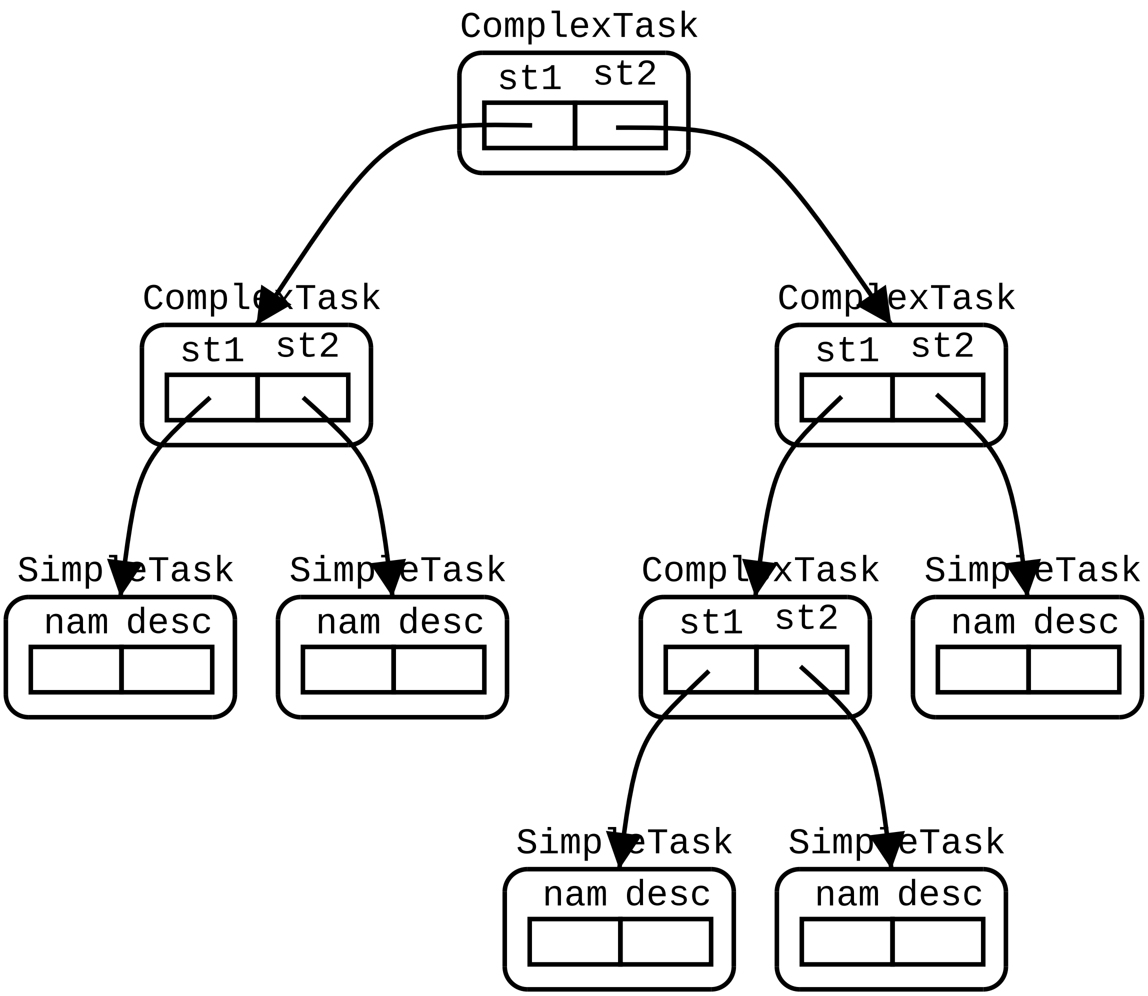
Applications (reverse)

```
static String reverse(String s)
{
  String r = "";
  Stack stack = new Stack();
  int i = 0;
  while (i < s.length()) {
    stack.push(new Character(s.charAt(i)));
    i++;
  }
  while (!stack.isempty()) {
    Character c = (Character)stack.top();
    r = r + c.charValue();
    stack.pop();
  }
  return r;
}
```

Binary Trees



Binary Trees



Binary Trees

```
abstract class Task {
  String name;
}

class SimpleTask extends Task {
  String description;
  void perform()
  {
    System.out.println(name+“:”+description);
    //...
  }
}

class ComplexTask extends Task {
  Task subtask1, subtask2;
}
```

Binary Trees

- Processing trees using recursion

```
class Worker {
  void work(Task t)
  {
    if (t instanceof SimpleTask) {
      ((SimpleTask)t).perform();
    }
    else if (t instanceof ComplexTask) {
      work(((ComplexTask)t).subtask1);
      work(((ComplexTask)t).subtask2);
    }
  }
}
```

Binary Trees

- Processing trees using stacks

```
class Worker {
  void work(Task t)
  {
    Stack s = new Stack();
    s.push(t);
    while (!s.isempty()) {
      Task temp = s.top();
      s.pop();
      if (temp instanceof SimpleTask) {
        ((SimpleTask)t).perform();
      }
      else {
        s.push(((ComplexTask)temp).subtask2);
        s.push(((ComplexTask)temp).subtask1);
      }
    }
  }
}
```

Data structures zoo

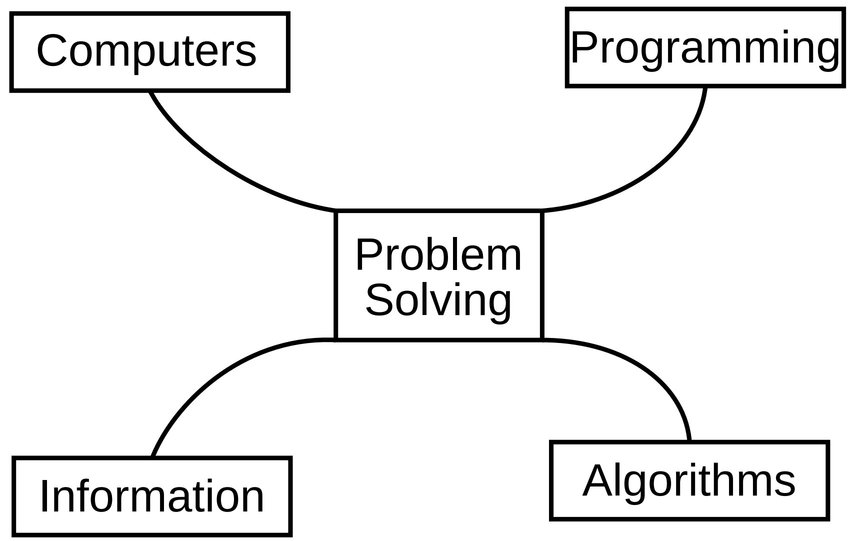
- Other data-structures: sets, bags, priority queues, heaps, binary trees, n-ary trees, red-black trees, AVL trees, graphs, hyper-graphs, hi-graphs, dictionaries/mappings, etc.

- The selection of data-structure has a major impact on the efficiency of an algorithm.

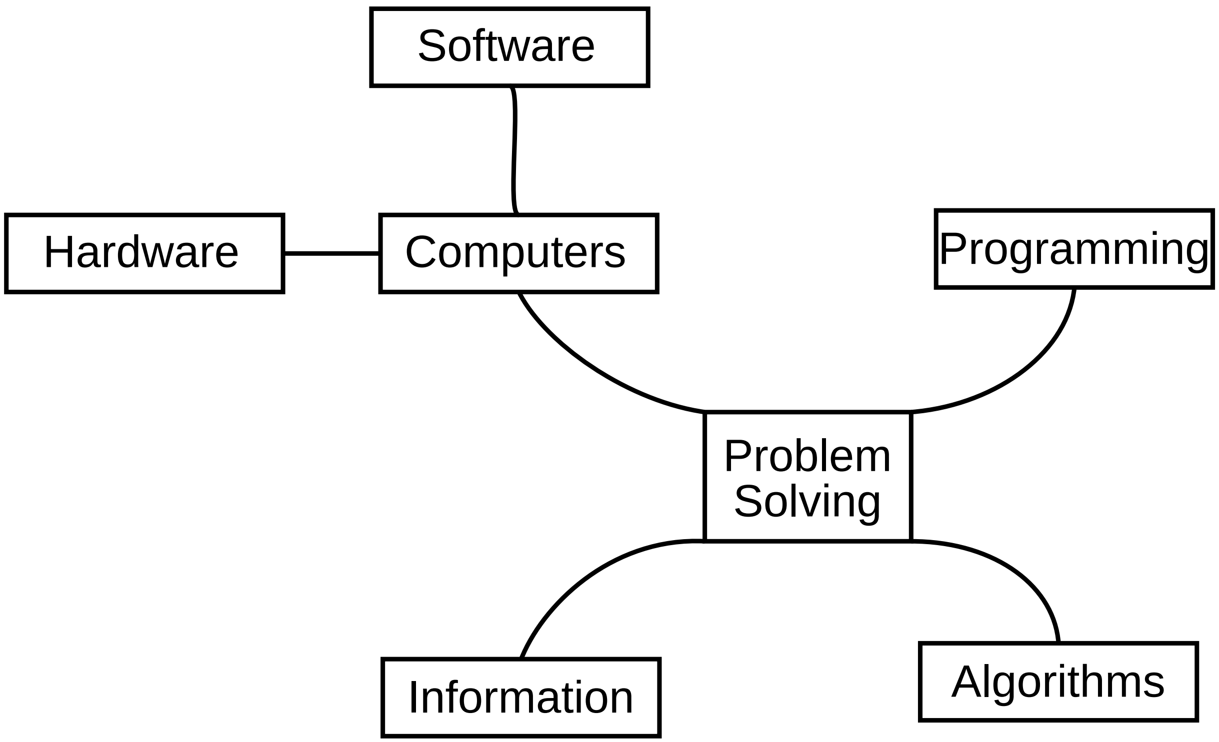
The big picture

Problem
Solving

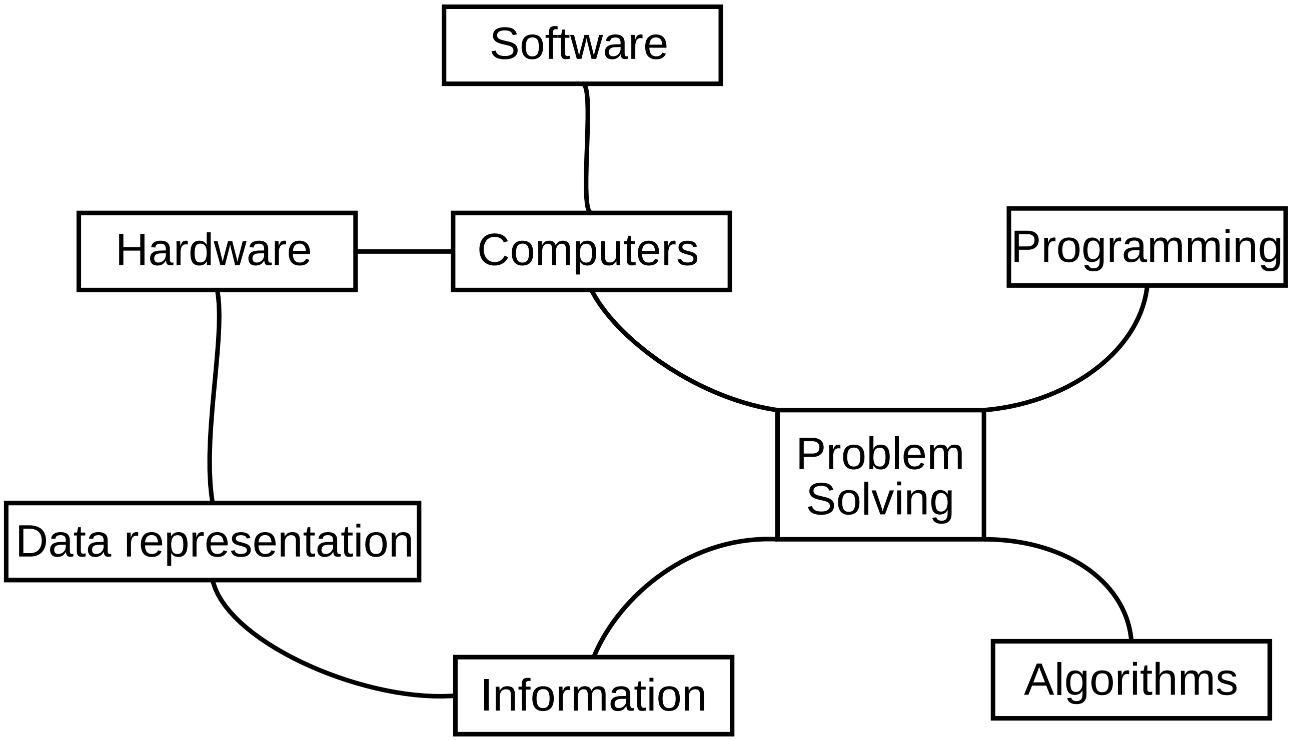
The big picture



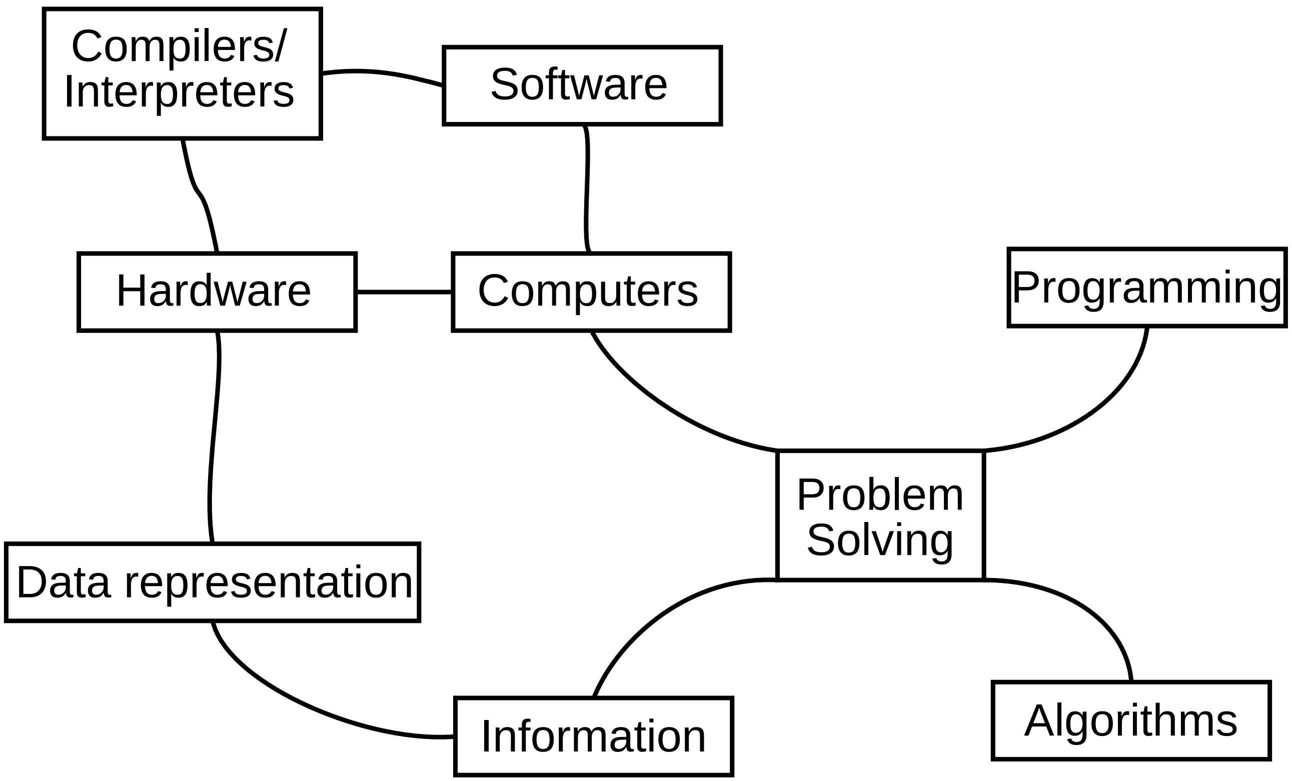
The big picture



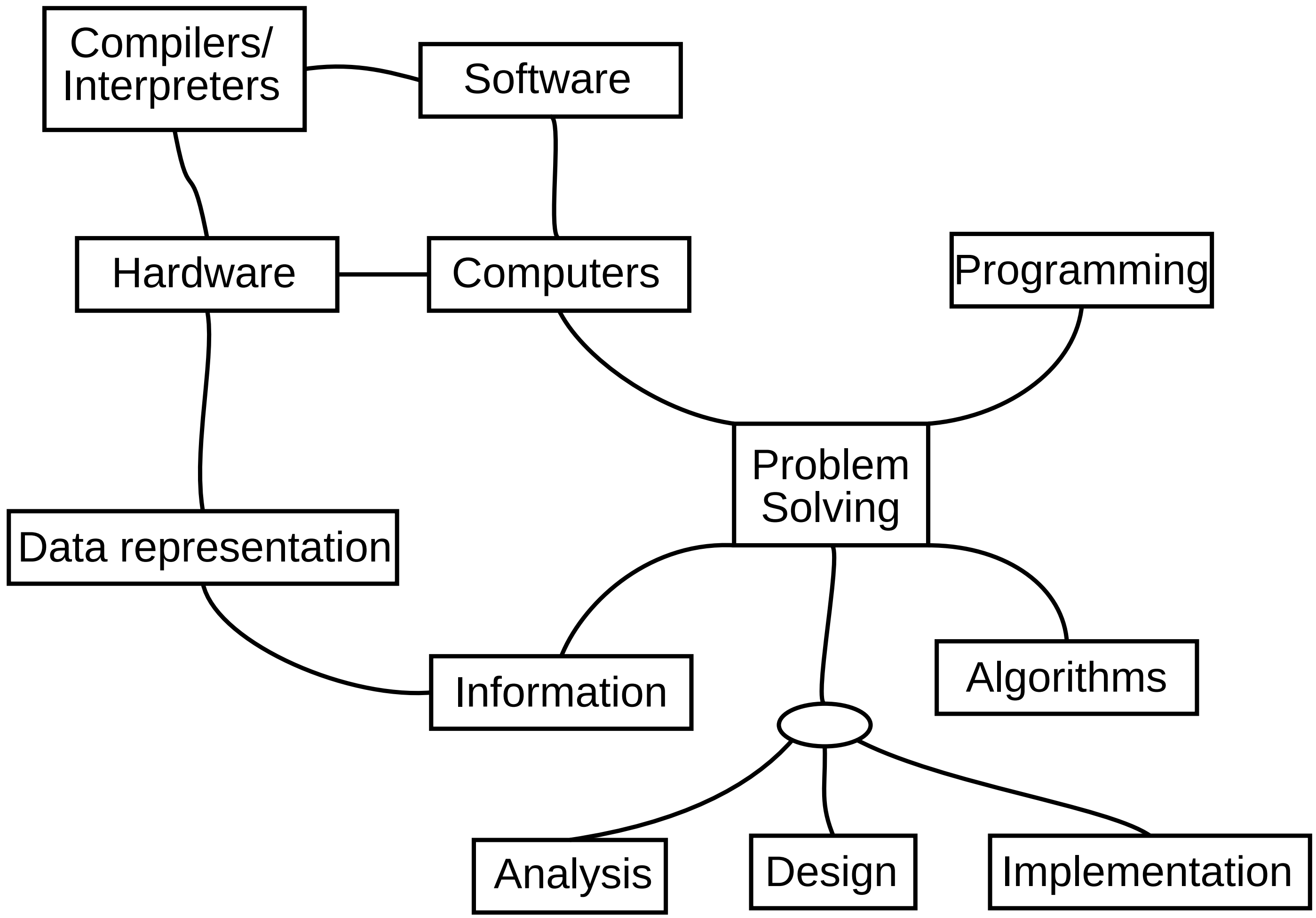
The big picture



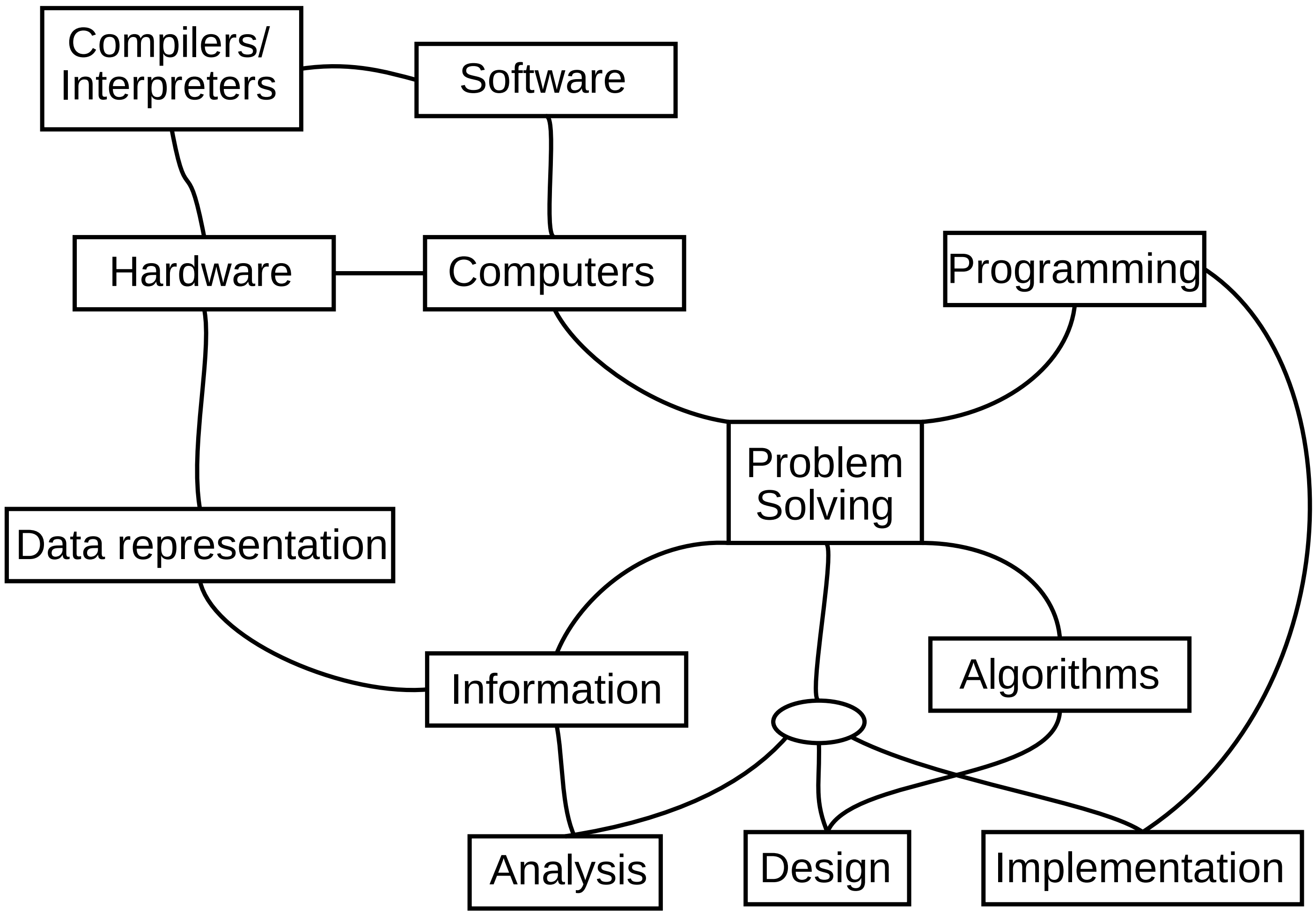
The big picture



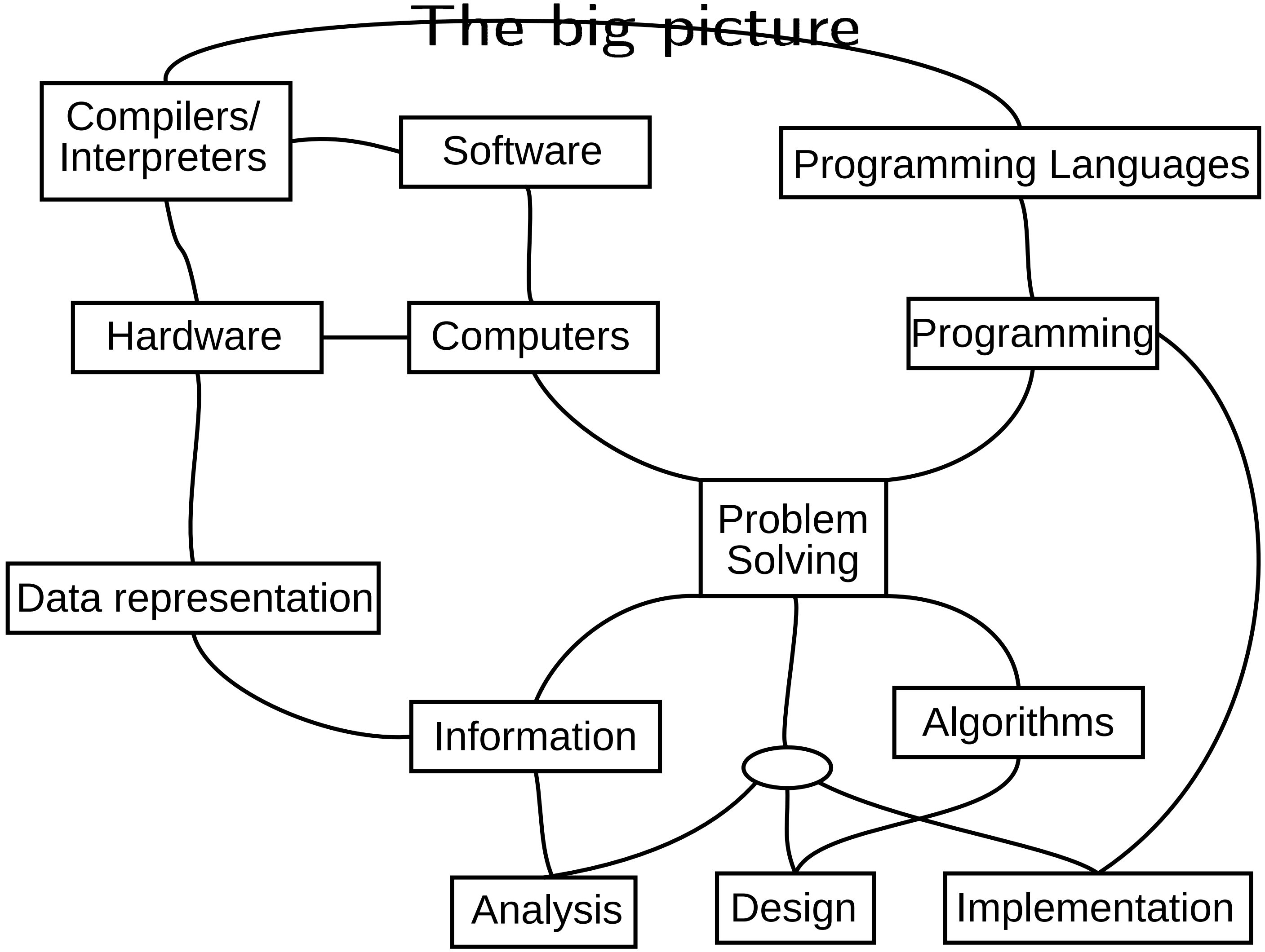
The big picture



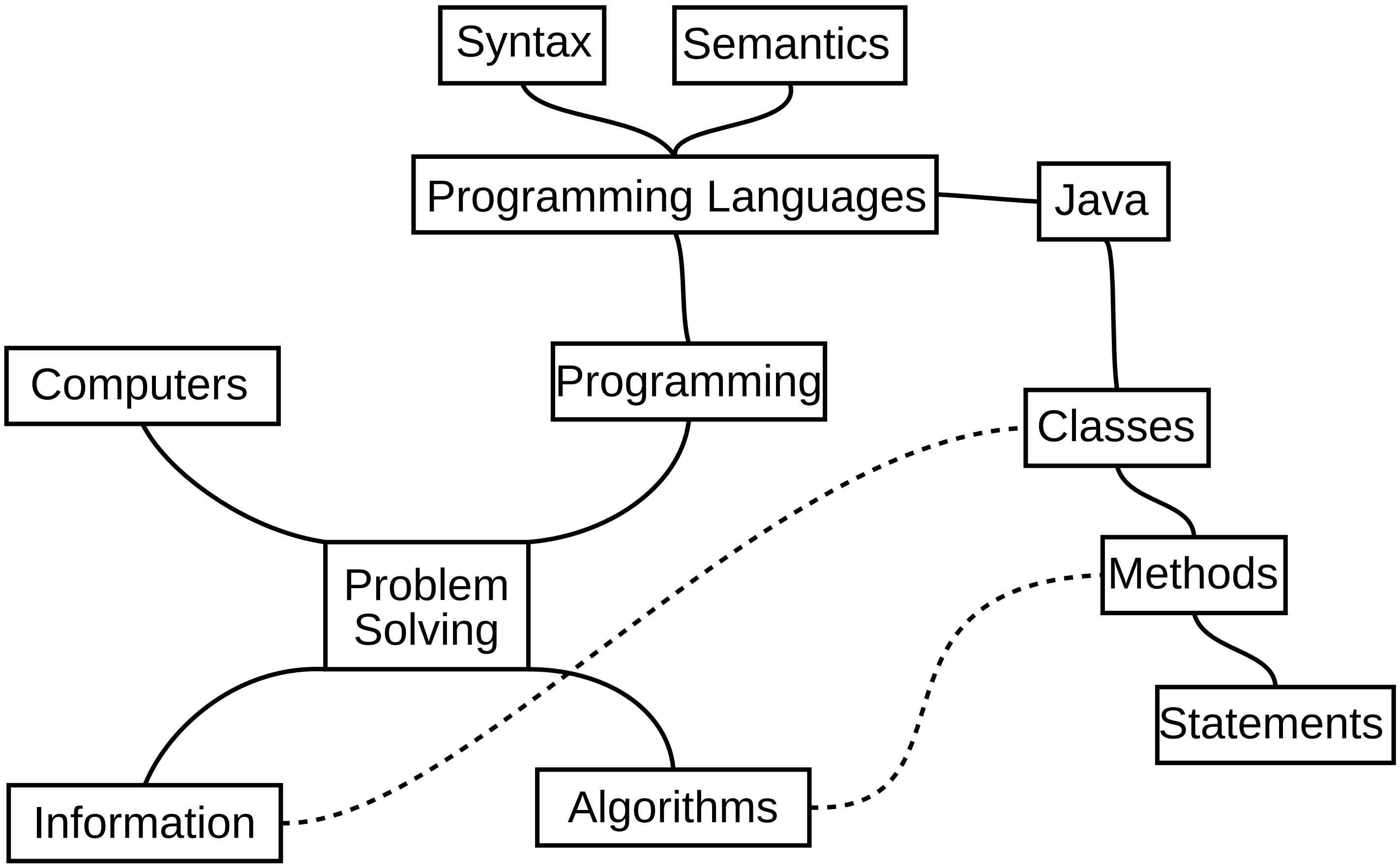
The big picture



The big picture



The big picture



Object-Oriented Programming

- The execution of an OO program consists of
  - Creation of objects
  - Interaction between objects (message-passing)

- Defining features of an OO language:
  - Class definitions (describing the types of objects and their structure,)
  - Object instantiation (creation,)
  - Message-passing (invoking methods,)
  - Aggregation (object structure, has-a relationships)
  - Encapsulation (objects as abstract units, hiding,)
  - Inheritance,
  - Polymorphism

Other tools and techniques

- Arrays
- Sorting
- Recursion
- Exceptions
- I/O
- Data Structures

The end