Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 2: this Friday @ 11:59 pm
  - Lab 1: next Monday @ 11:59 pm
  - Programming Assignment 1: Friday Oct. 7th
- Draft slides and handouts available on Canvas
- Lecture recordings are available under Panopto Video on Canvas
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- ADT Bag Implementations
  - Fixed-size array: ArrayBag
    - getFrequencyOf(T), contains(T), remove(), remove(T)
  - Resizable array: ResizableArrayBag

Muddiest Points

- **Q: The book shows that the return type of "remove anEntry" is a boolean. Why? The code we worked on in class returned the Entry**
- A: There are different ways of *designing* a method. The book uses a Boolean to return false when the entry is not found; we return the removed entry (not possible in the book's design) with null returned when the entry is not found.
- **Q: I don't understand why we need to do unchecked casts for arrays of type T when type erasure happens. Why would we not declare an array of type Object instead?**
- A: We want to make the compiler happy when it does type checking. Type checking happens **before** type erasure.

Muddiest Points

- **Q: I'm still confused about <T> vs <?> vs <Object>, what the hierarchy is and when you would want to use each one.**
- A:
  - ArrayBag<T> is a subtype of ArrayBag<?>.
  - ArrayBag<Object> is a subtype of ArrayBag<?>
  - ArrayBag<T> is NOT a subtype of ArrayBag<Object>
  - All of the above are subtypes of Object.
  - Let's say that we are designing a method **void display** that takes an ArrayBag as a parameter.
    - **void display(ArrayBag<T>) →** we need to use T in the method declaration and/or method body
    - **void display(ArayBag<?>) →** we don't need to use T
    - **void display(ArrayBag<Object>) →** This is limited to only receive ArrayBag<Object>, not for example ArrayBag<Integer>

Muddiest Points

- **Q: Why does the order of an arraybag not matter**
- A: This is the definition of the ADT Bag. The client (user of a Bag) doesn't expect a Bag to keep its items in a particular order

Muddiest Points

- **Q: The hardest thing for me to understand is what happens to everything/where all the stuff goes during the remove method.**
- A: The steps we took in remove
  - find the index of an item that we want to remove
  - if not found, return null
  - if found,
    - save the found item by **result = bag[index]**
    - replace the item by the last item of the array
      - **bag[index] = bag[numberOfItems-1]**
    - remove the last item in the array **bag[numberOfItems-1] = null**
    - decrement the logical size of the array **numberOfItems--**

Muddiest Points

- **Q: Why do we care if the one problem method was public? Does revealing we are using an array even matter?**
- A: Yes, it does! A Bag maintains its items in no particular order. Allowing the client to get the index of an item implicitly promises the client that the item will remain at that index; which is not guaranteed.

Today's Agenda

- ADT Bag Implementations
  - Fixed-size array: ArrayBag
    - copy constructor
  - Resizable array: ResizableArrayBag
    - add
  - Linked implementation

Copy constructor for ArrayBag<T>

- deep copy! (not deeper though; why?)
- how would you make it a shallow copy?

```java
//Copy constructor
public ArrayBag(ArrayBag<T> other){
  checkCapacity(other.bag.length);
  @SuppressWarnings("unchecked")
  T[] temp = (T[]) new Object[other.bag.length];
  for(int i=0; i<other.size; i++){
    temp[i] = (T)other.bag[i];
  }
  bag = temp;
  size = other.size;
  initialized = true;
}
```

ResizableArrayBag: add

- Should we double the capacity before or after adding the item to the array?

```java
public boolean add(T item) {
  checkIntegrity();
  boolean result = false;
  if(!isFull()){
    bag[size] = item;
    size++;
    result = true;
  }

  if(size == bag.length){
    doubleCapacity();
  }
  return result;
}
```

ResizableArrayBag: doubleCapacity

```java
private void doubleCapacity(){
    int capacity = bag.length;
    checkCapacity(2*capacity);

    @SuppressWarnings("unchecked")
    T[] temp = (T[])new Object[2*capacity];
    for(int i=0; i<size; i++){
        temp[i] = bag[i];
    }
    bag= temp;
    //bag = Arrays.copyOf(bag, 2*capacity);
}
```

ResizableArrayBag

- Can we still have the final keyword for the underlying array?

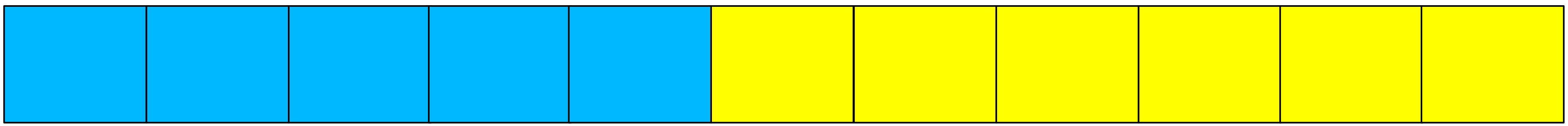
```
private T[] bag;
```

Pros and Cons of Using an Array

- Pros:
  - Adding an entry to the bag is fast
  - Removing an unspecified entry is fast
- Cons:
  - May be wasteful of memory
    - for example, we have 10 items now in the Bag, but we *expect* that we will have 1,000,000 items later
      - how big should the array be?
  - Increasing the size of the array requires time to copy its entries

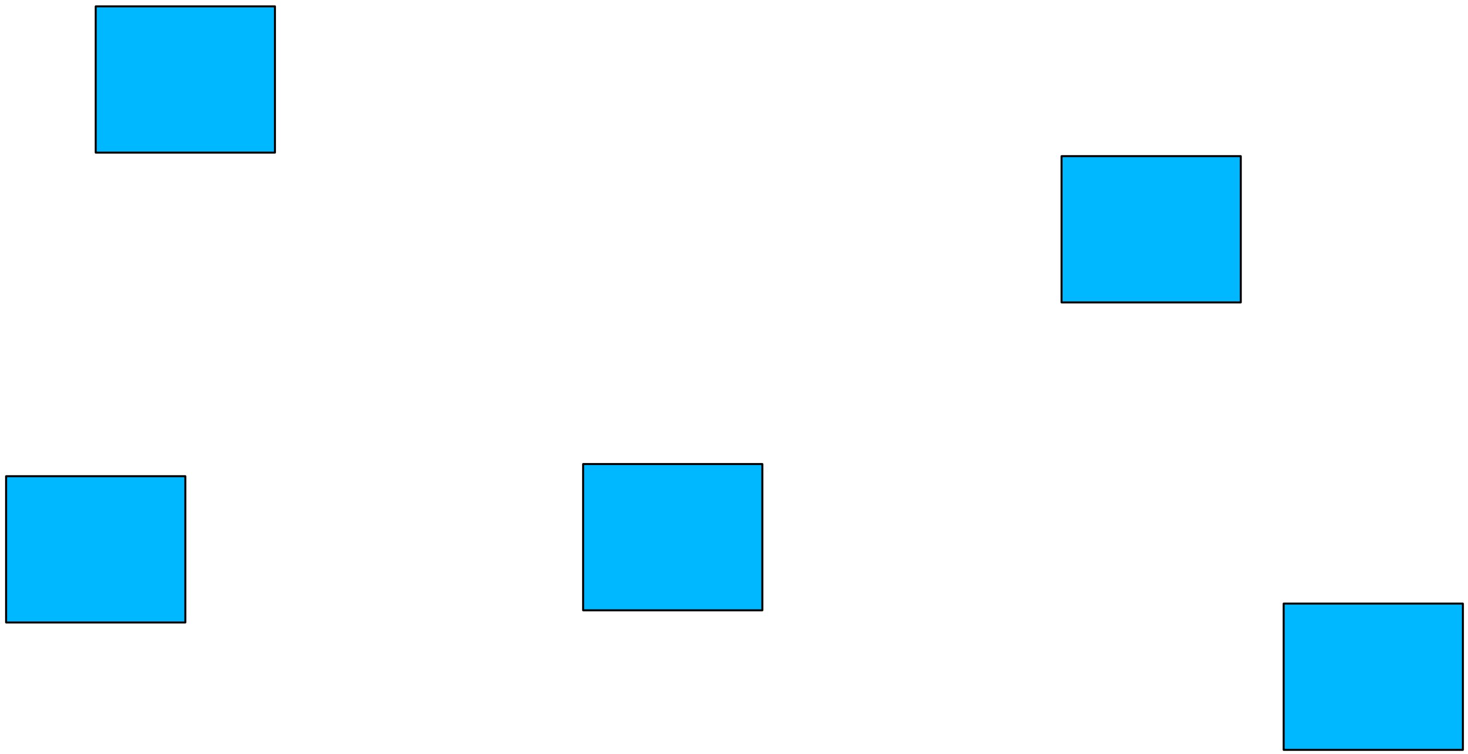
Linked Implementation

Array



Linked Implementation

Linked Chain

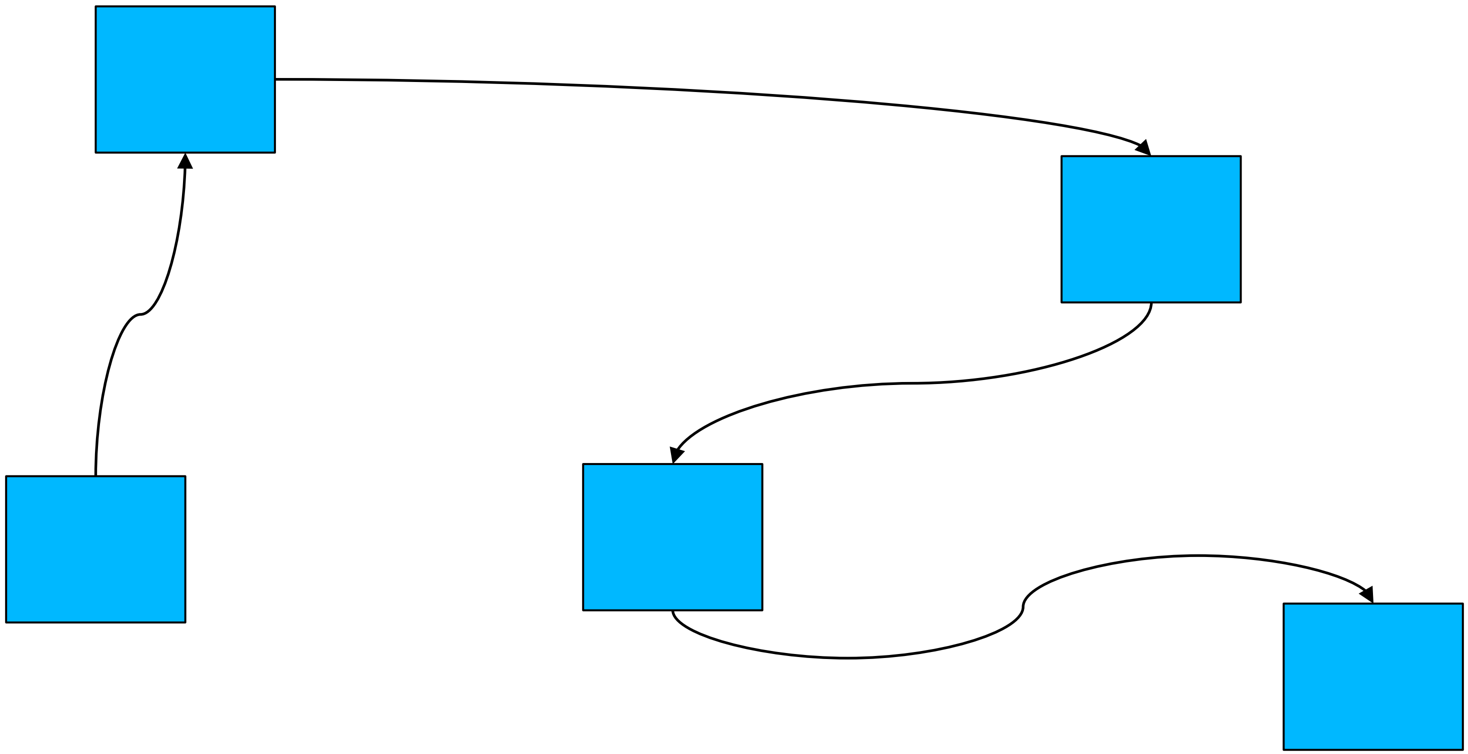


filled entry

empty entry

Linked Implementation

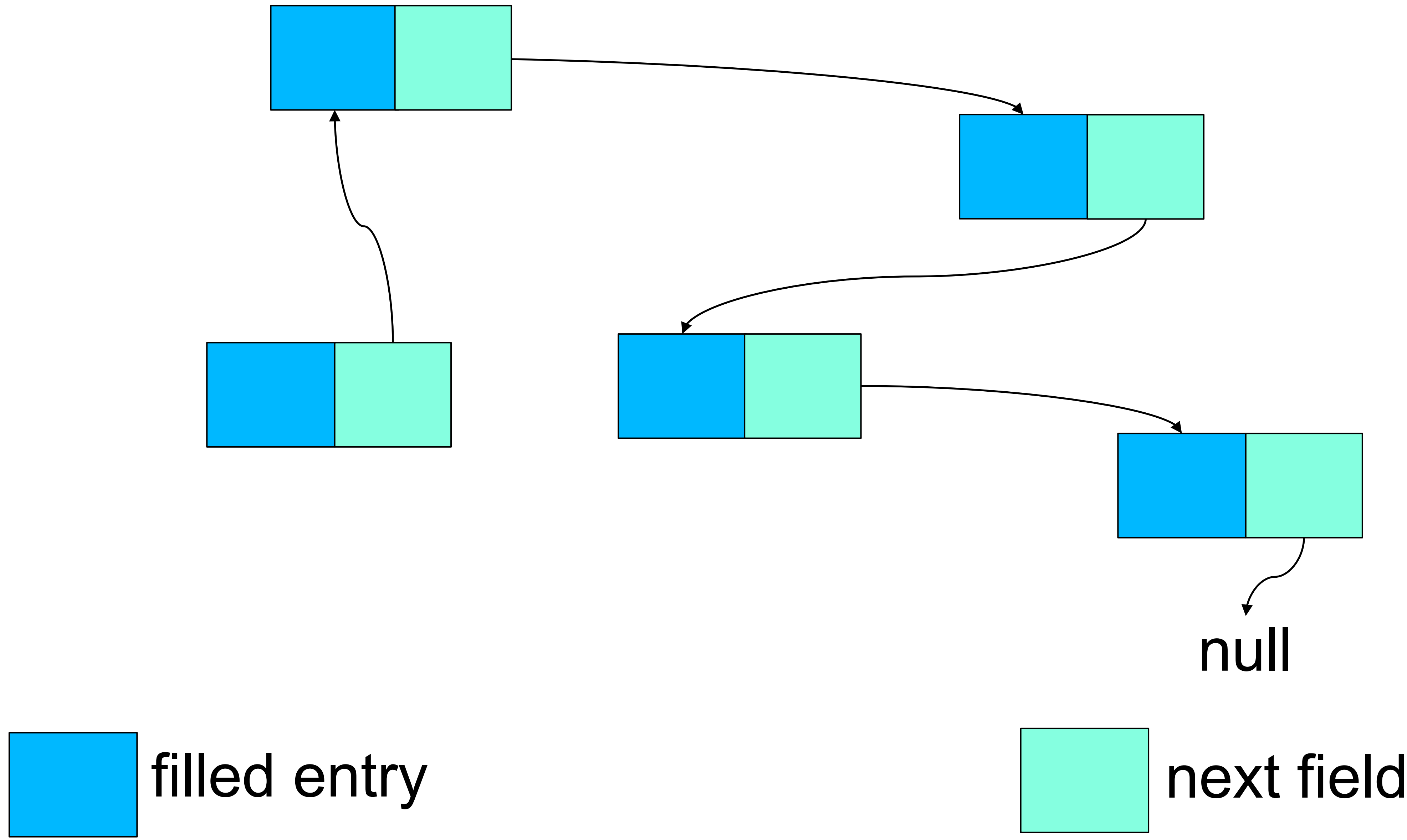
Linked Chain



filled entry

Linked Implementation

Linked Chain



Pros of Using a Linked Chain

- Bag can grow and shrink in size as necessary.
- Remove and recycle nodes that are no longer needed
  - Using Java's garbage collection

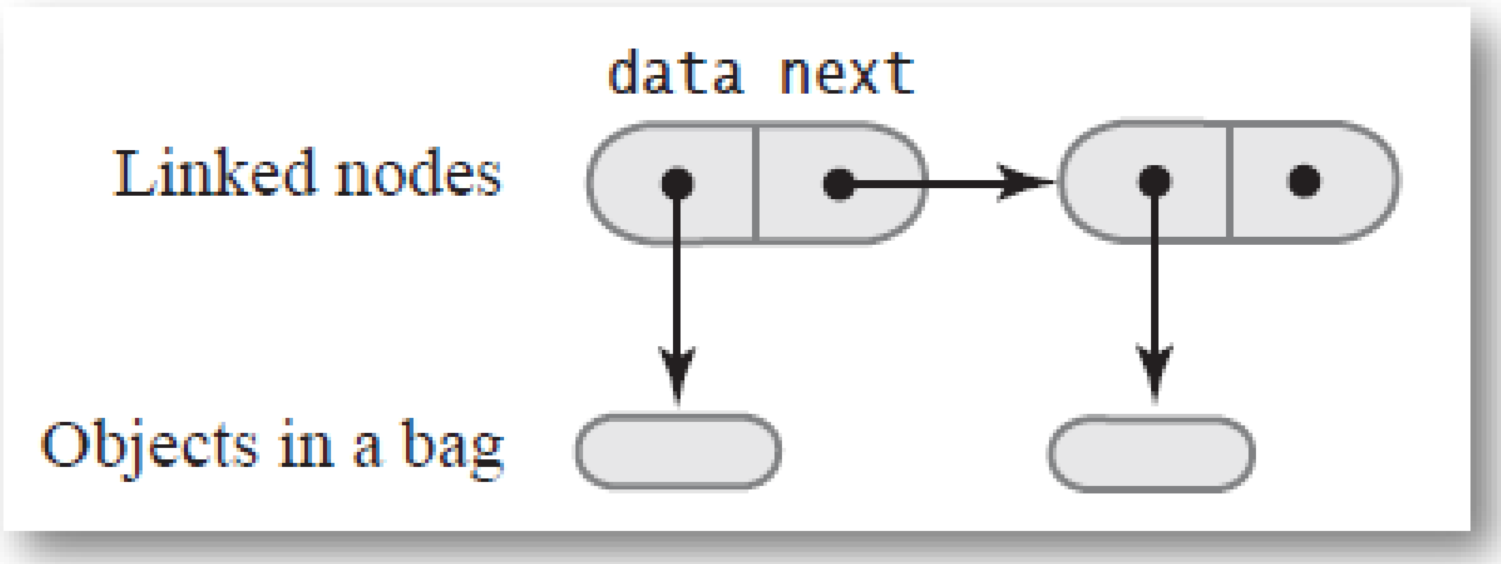
The Private Class **Node**

- The private inner class **Node**

```
private class Node
{
    private T    data; // Entry in bag
    private Node next; // Link to next node

    private Node(T dataPortion)
    {
        this(dataPortion, null);
    } // end constructor

    private Node(T dataPortion, Node nextNode)
    {
        data = dataPortion;
        next = nextNode;
    } // end constructor
} // end Node
```

The Private Class **Node**

Two linked nodes that each **references** object data



Class `LinkedBag`

We need to keep track of only the first node in the chain!

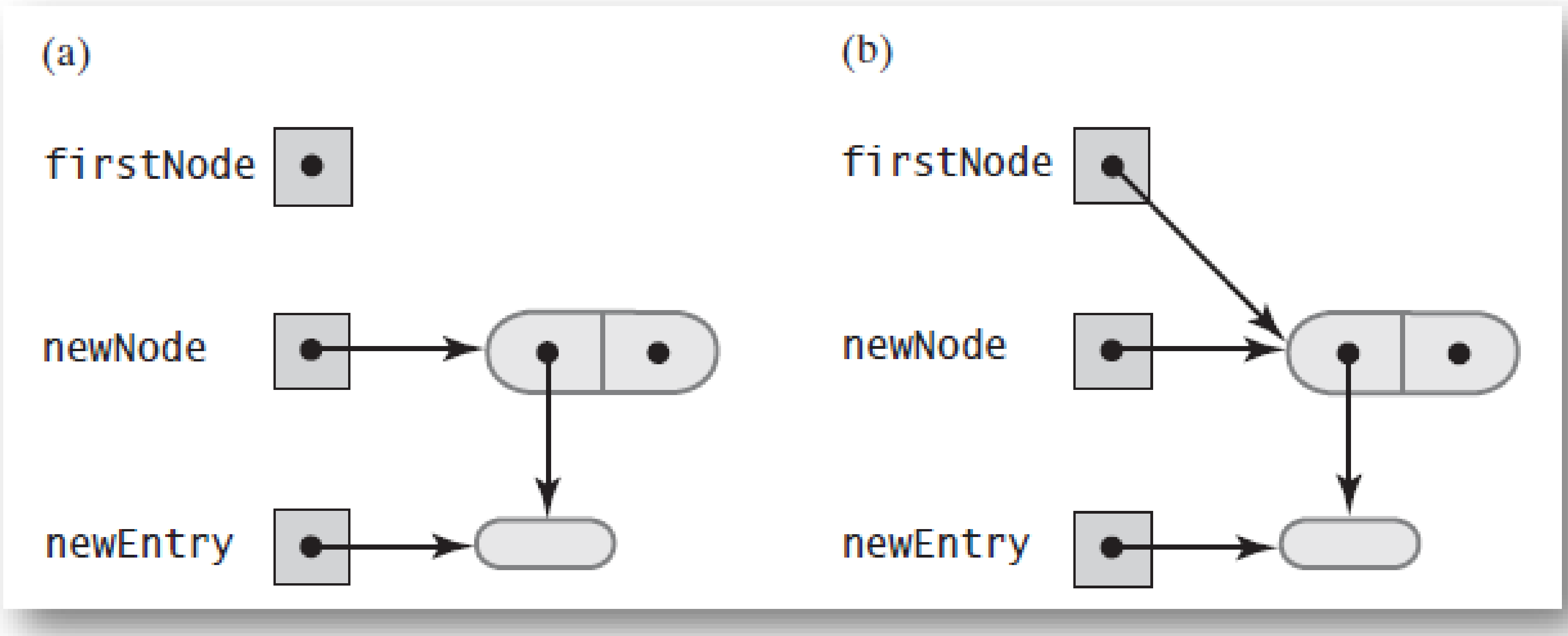
```
/**
    A class of bags whose entries are stored in a chain of linked nodes.
    The bag is never full.
    @author Frank M. Carrano
*/
public final class LinkedBag<T> implements BagInterface<T>
{
   private Node firstNode;       // Reference to first node
   private int  numberOfEntries;

   public LinkedBag()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end default constructor

   < Implementations of the public methods declared in BagInterface go here. >
```

Class `LinkedBag`

```
        numberOfEntries = 0;
    } // end default constructor

    < Implementations of the public methods declared in BagInterface go here. >

    . . .

    private class Node // Private inner class
    {
        < See Listing 3-1. >
    } // end Node
} // end LinkedBag
```

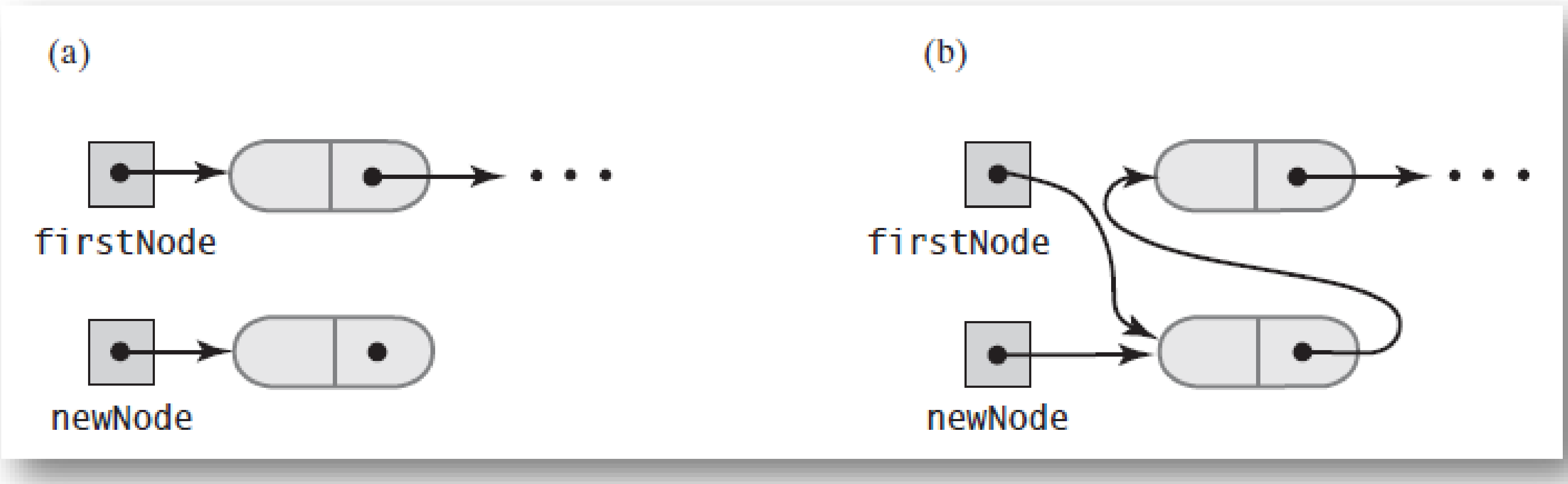
Beginning a Chain of Nodes

(a) An empty chain and a new node; (b) after adding a new node to a chain that was empty



Beginning a Chain of Nodes

A chain of nodes (a) just prior to adding a node at the beginning; (b) just after adding a node at the beginning



LinkedBag.add

- The method **add**

```
/** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry.
    @return  True. */
public boolean add(T newEntry) // OutOfMemoryError possible
{
   // Add to beginning of chain:
   Node newNode = new Node(newEntry);
   newNode.next = firstNode;    // Make new node reference rest of chain
                                // (firstNode is null if chain is empty)
   firstNode = newNode;    // New node is at beginning of chain
   numberOfEntries++;

   return true;
} // end add
```

Method `toArray`

- The method **`toArray`** returns an array of the entries currently in a bag by **traversing** the chain

```java
/** Retrieves all entries that are in this bag.
    @return  A newly allocated array of all the entries in the bag. */
public T[] toArray()
{
   // The cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast

   int index = 0;
   Node currentNode = firstNode;
   while ((index < numberOfEntries) && (currentNode != null))
   {
      result[index] = currentNode.data;
      index++;
      currentNode = currentNode.next;
   } // end while

   return result;
} // end toArray
```

Method `getFrequencyOf`

- Counts the number of times a given entry appears
- Also traverses the chain

```java
/** Counts the number of times a given entry appears in this bag.
    @param anEntry  The entry to be counted.
    @return  The number of times anEntry appears in the bag. */
public int getFrequencyOf(T anEntry)
{
   int frequency = 0;
   int loopCounter = 0;
   Node currentNode = firstNode;

   while ((loopCounter < numberOfEntries) && (currentNode != null))
   {
      if (anEntry.equals(currentNode.data))
         frequency++;

      loopCounter++;
      currentNode = currentNode.next;
   } // end while

   return frequency;
} // end getFrequencyOf
```

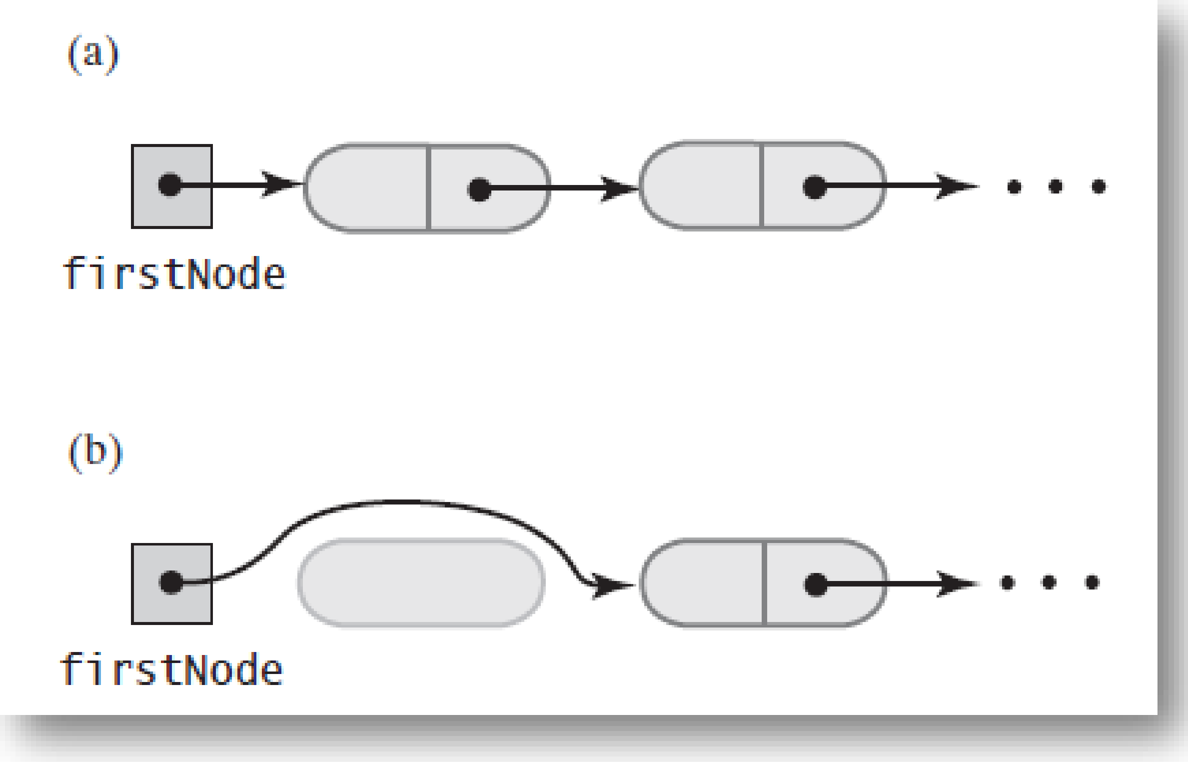
Method `contains`

- Determine whether a bag contains a given entry

```
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;
   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.data))
         found = true;
      else
         currentNode = currentNode.next;
   } // end while
   return found;
} // end contains
```

Removing an unspecified item

A chain of nodes (a) just prior to removing the first node; (b) just after removing the first node



Removing a specified item

- Note need for private method `getReferenceTo`
- Returns a reference to a node that references an object that equals anEntry

```
// Locates a given entry within this bag.
// Returns a reference to the node containing the entry, if located,
// or null otherwise.
private Node getReferenceTo(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.data))
         found = true;
      else
         currentNode = currentNode.next;
   } // end while

   return currentNode;
} // end getReferenceTo
```

Method `remove`

- Similar trick to what we did in ArrayBag.remove(T)
  - replace data by data of first item
- Note use of method `getReferenceTo`

```
public boolean remove(T anEntry)
{
   boolean result = false;
   Node nodeN = getReferenceTo(anEntry);

   if (nodeN != null)
   {
      nodeN.data = firstNode.data; // Replace located entry with entry
                                   // in first node
      firstNode = firstNode.next;  // Remove first node
      numberOfEntries--;
      result = true;
   } // end if

   return result;
} // end remove
```

Node as a Public class

- **Node** can be implemented as an independent class
- Needs to be generic!

```java
class Node<T>
{
   private T       data;
   private Node<T> next;

   Node(T dataPortion) // The constructor's name is Node, not Node<T>
   {
      this(dataPortion, null);
   } // end constructor
```

Node as a Public class

- Need setters and getters

```
    Node(T dataPortion, Node<T> nextNode)
    {
        data = dataPortion;
        next = nextNode;
    } // end constructor

    T getData()
    {
        return data;
    } // end getData

    void setData(T newData)
    {
        data = newData;
    } // end setData
```

Node as a Public class

```
    void setData(T newData)
    {
        data = newData;
    } // end setData

    Node<T> getNextNode()
    {
        return next;
    } // end getNextNode

    void setNextNode(Node<T> nextNode)
    {
        next = nextNode;
    } // end setNextNode
} // end Node
```

When Node is a Public class

```
public class LinkedBag<T> implements BagInterface<T>
{
   private Node<T> firstNode;
   . . .
   public boolean add(T newEntry)
   {
      Node<T> newNode = new Node<T>(newEntry);
      newNode.setNextNode(firstNode);
      firstNode = newNode;
      numberOfEntries++;

      return true;
   } // end add
   . . .
} // end LinkedBag
```

This occurrence of T is optional

Cons of Using a Chain

- Removing specific entry requires search of array or chain
- Chain requires more memory than array of same logical size
  - why?

Why do we care about efficient code?

- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

Example

- Consider the problem of summing: computing the sum
  1 + 2 + . . . + n for an integer n > 0

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + ... + n$$

One solution

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

```
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);
```

Another solution

Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

```
// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
    for (long j = 1; j <= i; j++)
        sum = sum + 1;
} // end for
System.out.println(sum);
```

And a third solution

Algorithm C

```
sum = n * (n + 1) / 2
```

```
// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

Which is “best”?

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- The study of time and space complexities of algorithms is called analysis of algorithms

Counting Basic Operations

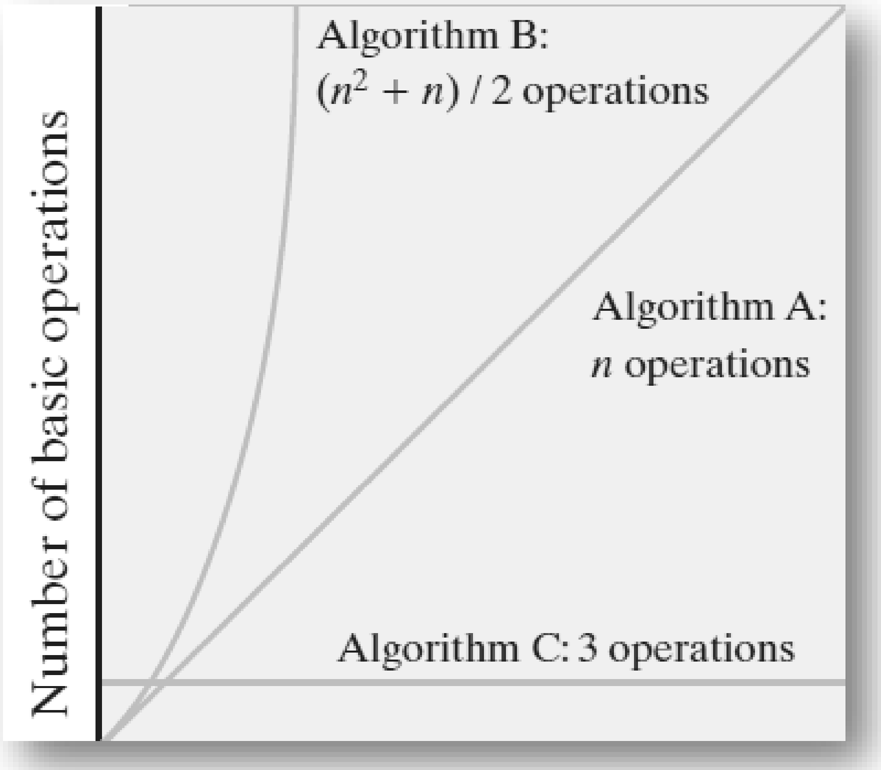
- A basic operation of an algorithm
  - The most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	**3**

- The number of basic operations required by the sum algorithms

Counting Basic Operations

- The number of basic operations required by the sum algorithms as a function of *n*



Counting Basic Operations

- Typical growth-rate functions evaluated at increasing values of *n*

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Best, Worst, and Average Cases

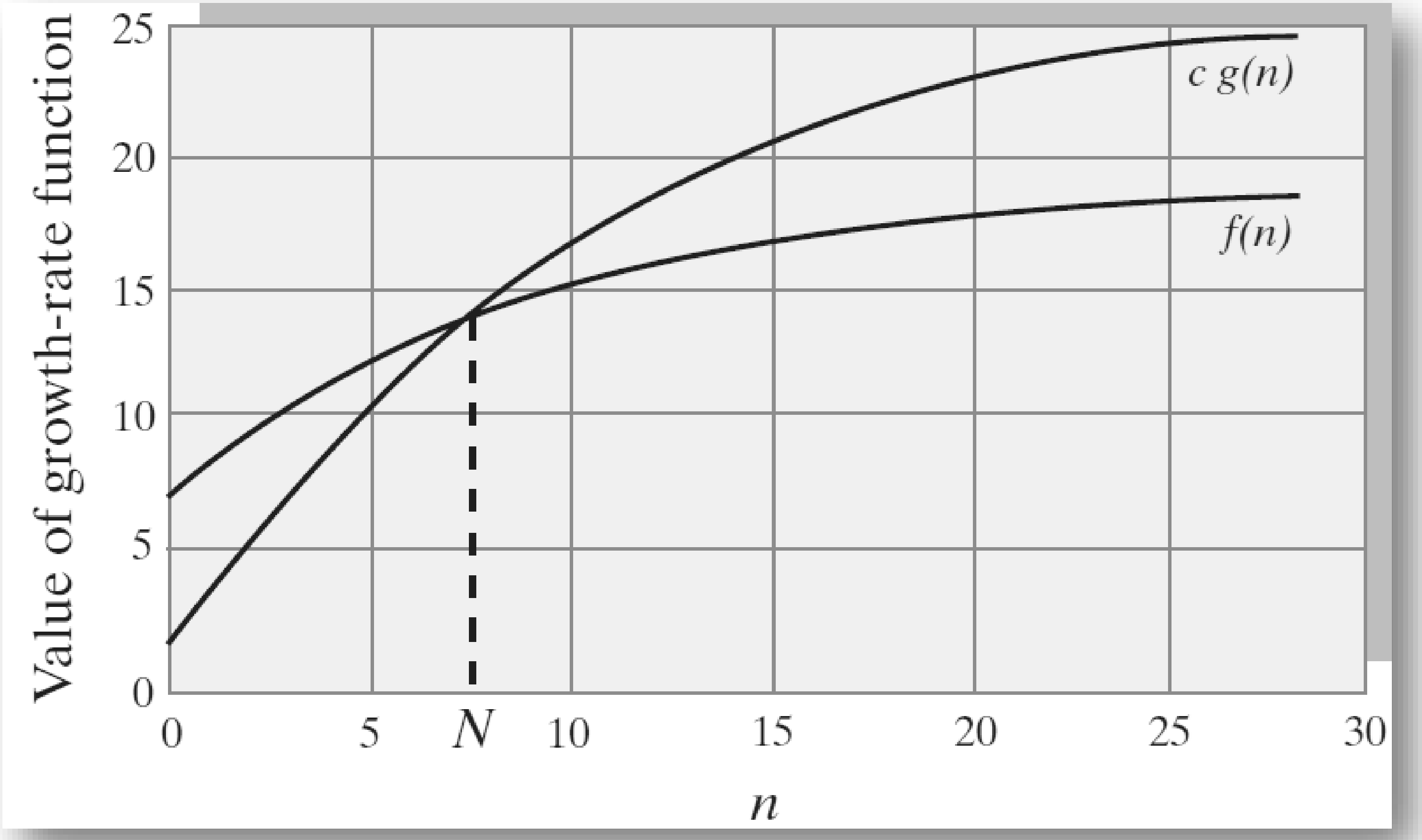
- For some algorithms, execution time depends only on size of data set
- Other algorithms depend on the nature of the data itself
  - Here we seek to know best case, worst case, average case

Big Oh Notation

- A function f(n) is of order at most g(n)
- That is, f(n) is O(g(n))—if
  - A positive real number c and positive integer N exist …
  - Such that $f(n) \le c * g(n)$ for all $n \ge N$
  - That is, $c * g(n)$ is an upper bound on $f(n)$ when n is sufficiently large

Big Oh Notation

- An illustration of the definition of Big Oh



Big Oh Notation

- Identities for Big Oh Notation

The following identities hold for Big Oh notation:

$$O(k\,g(n)) = O(g(n)) \text{ for a constant } k$$

$$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$$

$$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$$

$$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n)))$$

$$O(\max(g_1(n), g_2(n), \ldots, g_m(n))) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$$

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is $4n^2 + 50n - 10$,

$$\begin{aligned} O(4n^2 + 50n - 10) &= O(4n^2) && \text{by ignoring the smaller terms} \\ &= O(n^2) && \text{by ignoring the constant multiplier} \end{aligned}$$

Complexities of Program Constructs

Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are g_1 and g_2, respectively	$O(\mathit{condition}) + \max(O(g_1), O(g_2))$
A loop that iterates m times and has a body whose growth-rate function is g	$m \times O(g(n))$

Time complexity of an algorithm

- Count the number of <u>**executed**</u> steps (basic operations or just lines)
  - sum = 0

    for i = 1 to n

      sum = sum + i
  - Number of executed lines is 2n + 2
- Let f(n) = the number of executed steps
  - n is the problem size
  - Usually it is the input size (very roughly, the number of keyboard presses needed to enter the input)
  - f(n) may depend only on *n* or on the actual values of the input
    - In the latter, need to find f(n) for best, average, worst cases

Time complexity of an algorithm

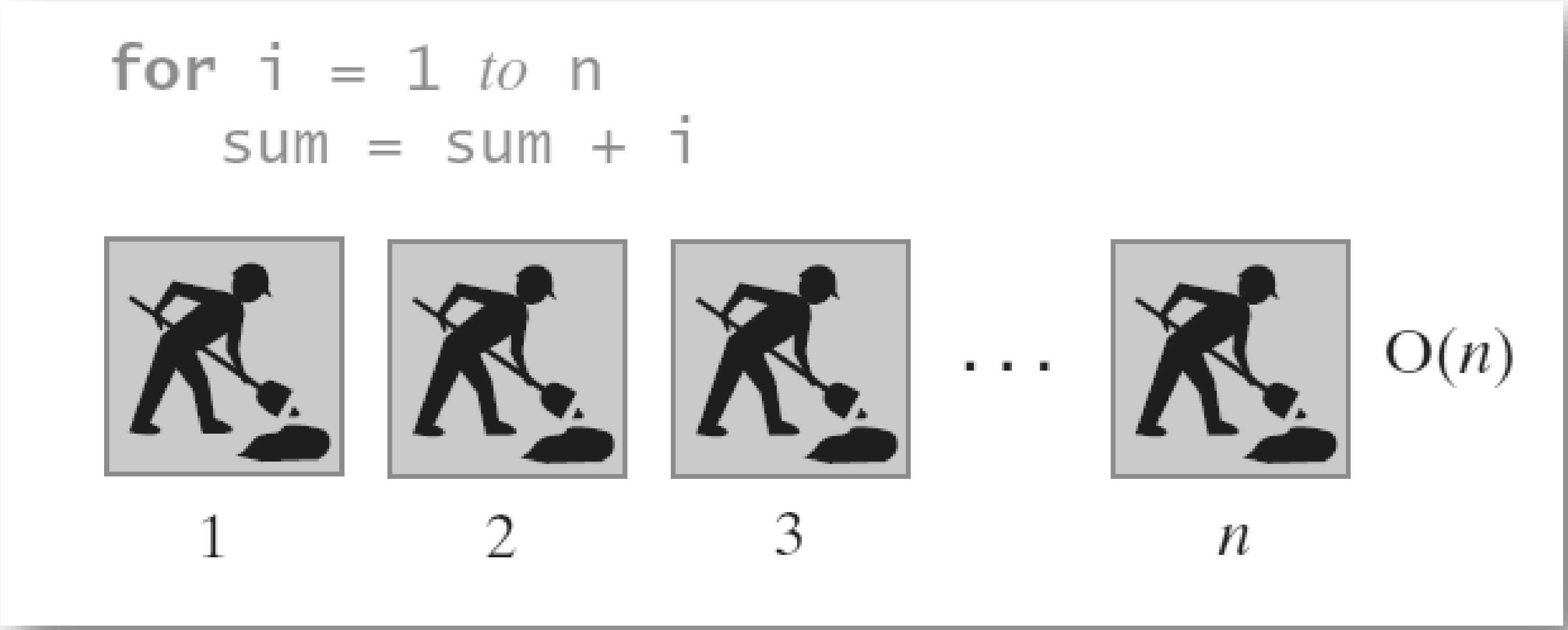
- Convert the function f into the Big-Oh notation
  - Ignore lower order terms
    - e.g., constant < log log n < log n < $\log^2 n$ < n < n log n < n^2 < n^3 < 2^n < n!
    - e.g., $n^2 + \log n = O(n^2)$
  - Ignore constant factors
    - cn => O(n), where c is a constant (doesn't depend on n)
    - 2^{cn} is **not** $O(2^n)$
  - $f(n) = 2n + 2 = O(2n) = O(n)$

The Big-Oh Family

- Big Omicron: O ≈ <=
  - $n = O(n)$
  - $n = O(n!)$
- Little Omicron: o ≈ <
  - $n \neq o(n)$
  - $n = o(n^2)$
- Big Omega: Ω ≈ >=
  - $n = \Omega(n)$
  - $2^n = \Omega(n)$
- Little Omega: ω ≈ >
  - $n \neq \omega(n)$
  - $n = \omega(1)$
- Theta: θ ≈ =
  - $5n = \theta(n)$ (has to be O and Ω)

Picturing Efficiency

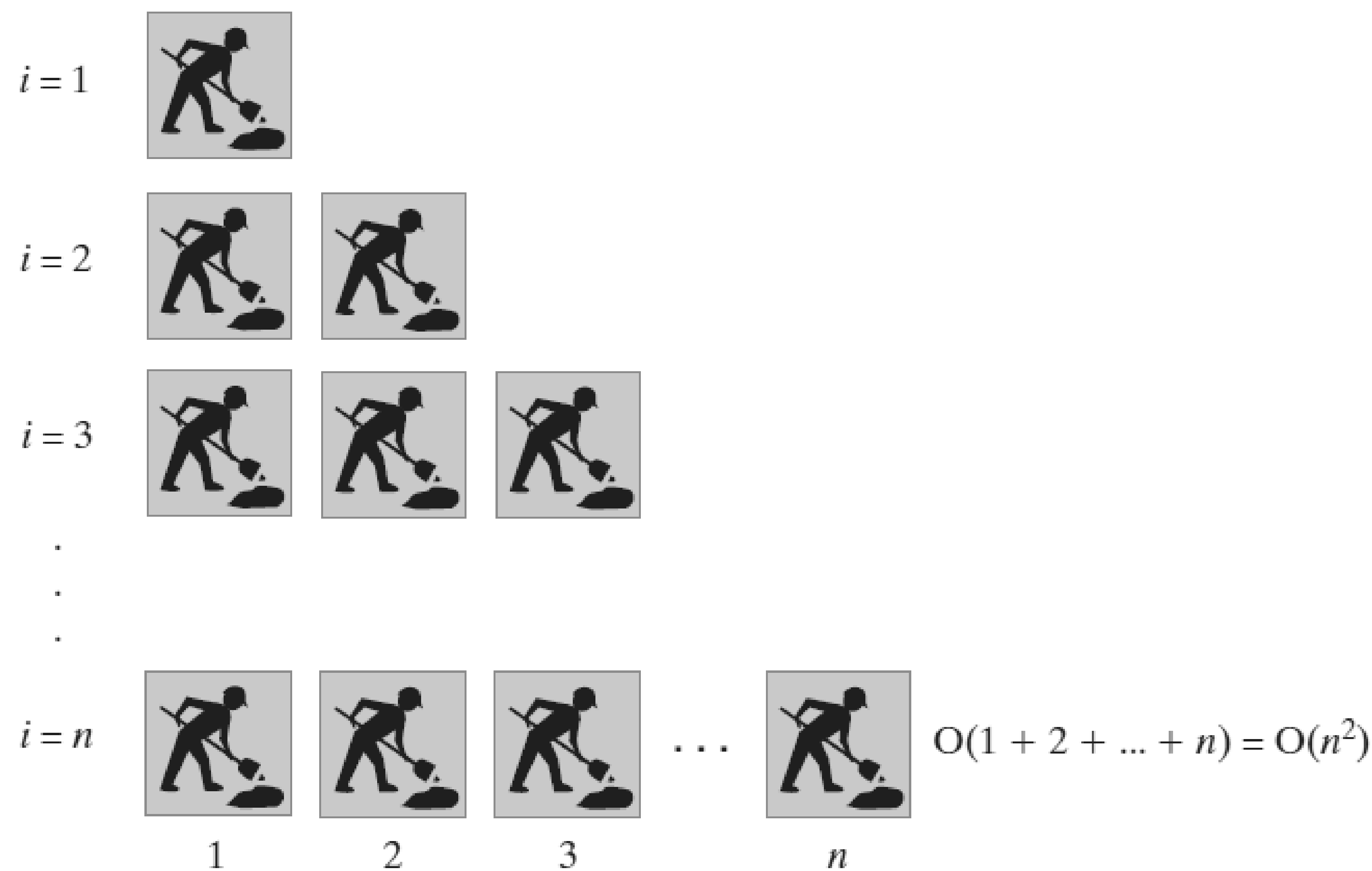
- An O(n) algorithm



Picturing Efficiency

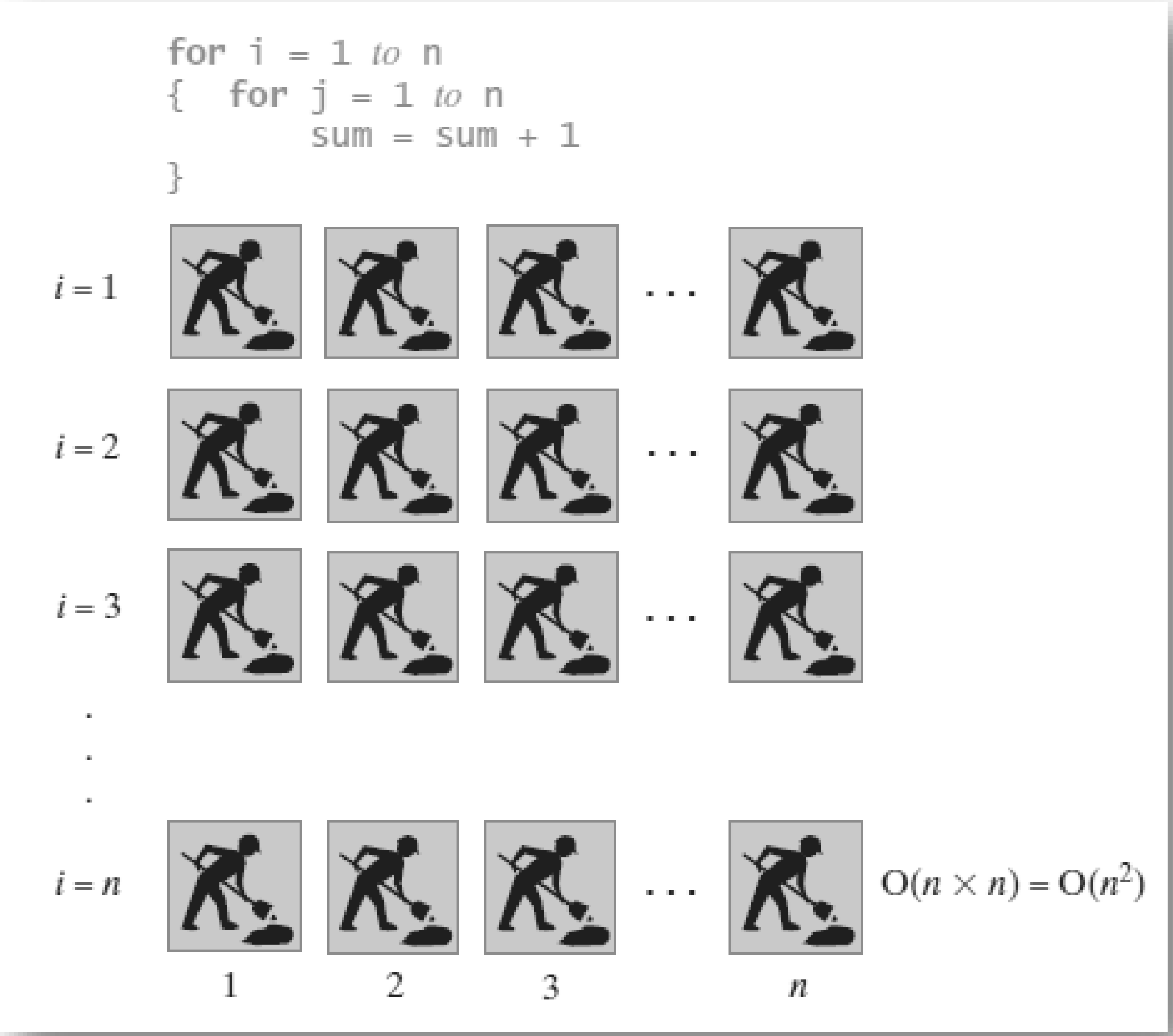
- An $O(n^2)$ algorithm

```
for i = 1 to n
{  for j = 1 to i
        sum = sum + 1
}
```



Picturing Efficiency

- Another O(n^2) algorithm



Picturing Efficiency

- The effect of doubling the problem size on an algorithm's time requirement

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size $2n$ Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

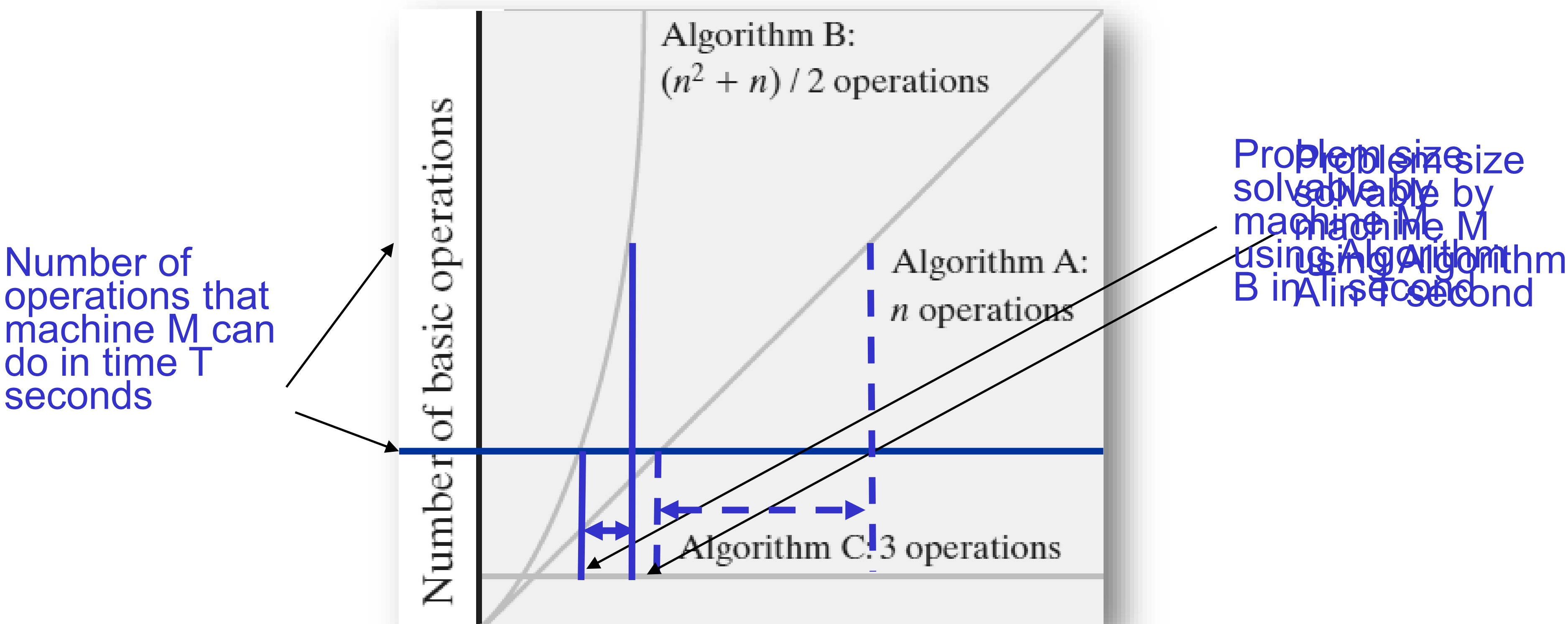
Picturing Efficiency

- The time required to process one million items by algorithms of various orders at the rate of one million operations per second

Growth-Rate Function g	$g(10^6) / 10^6$
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Riding Moore's law

- Writing an efficient algorithm (with less time complexity) is important
  - Such algorithm rides the exponentially-growing curve of hardware-speed ``better”



Efficiency of Implementations of ADT Bag

- The time efficiencies of the ADT bag operations for two implementations, expressed in Big Oh notation

Operation	Fixed-Size Array	Linked
`add(newEntry)`	$O(1)$	$O(1)$
`remove()`	$O(1)$	$O(1)$
`remove(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`clear()`	$O(n)$	$O(n)$
`getFrequencyOf(anEntry)`	$O(n)$	$O(n)$
`contains(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`toArray()`	$O(n)$	$O(n)$
`getCurrentSize(), isEmpty()`	$O(1)$	$O(1)$