

Data Structures

Lesson 3

BSc in Computer Science
University of New York, Tirana

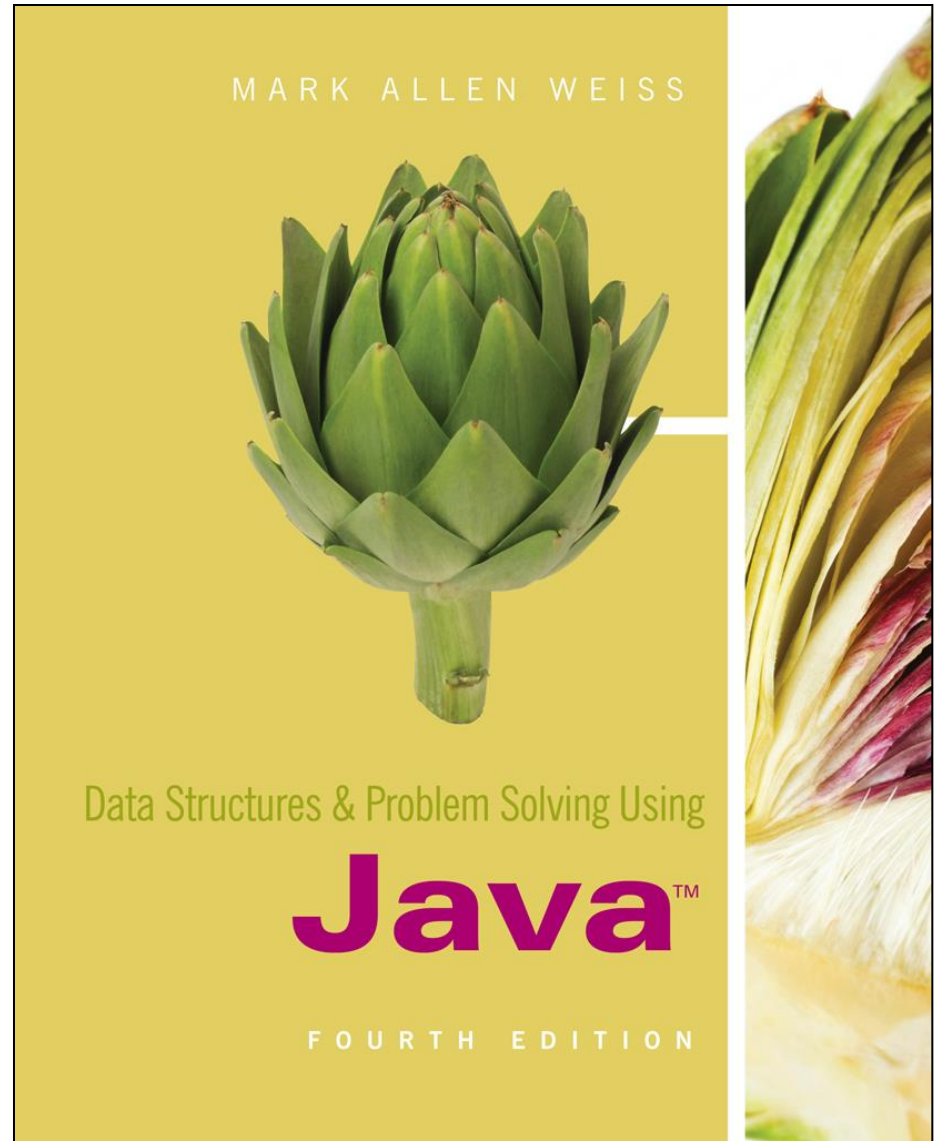
Assoc. Prof. Marenglen Biba

Outline

- PART I: Algorithm analysis
- PART II: Linked Lists

Chapter 5

Algorithm Analysis



Addison-Wesley
is an imprint of

PEARSON

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Running times and algorithms

- When we run a program on large amounts of input, we must be certain that the program terminates within a **reasonable amount of time**.
- Although the amount of running time is somewhat **dependent** on the programming language we use, and to a smaller extent the methodology we use (such as procedural versus object-oriented), often those factors are **unchangeable constants** of the design.
- Even so, the running time is most strongly correlated with the **choice of algorithms**.

Algorithm

- An algorithm is a **clearly specified set of instructions** the computer will follow to solve a problem.
- Once an algorithm is given for a problem and determined to be correct, the next step is to **determine the amount of resources, such as time and space**, that the algorithm will require.
- This step is called **algorithm analysis**.
- An algorithm that requires several gigabytes of main memory is **not useful** for most current machines, even if it is **completely correct**.

Some questions

- How to estimate the **time** required for an algorithm
- How to use techniques that drastically **reduce** the running time of an algorithm
- How to use a **mathematical framework** that more rigorously describes the running time of an algorithm

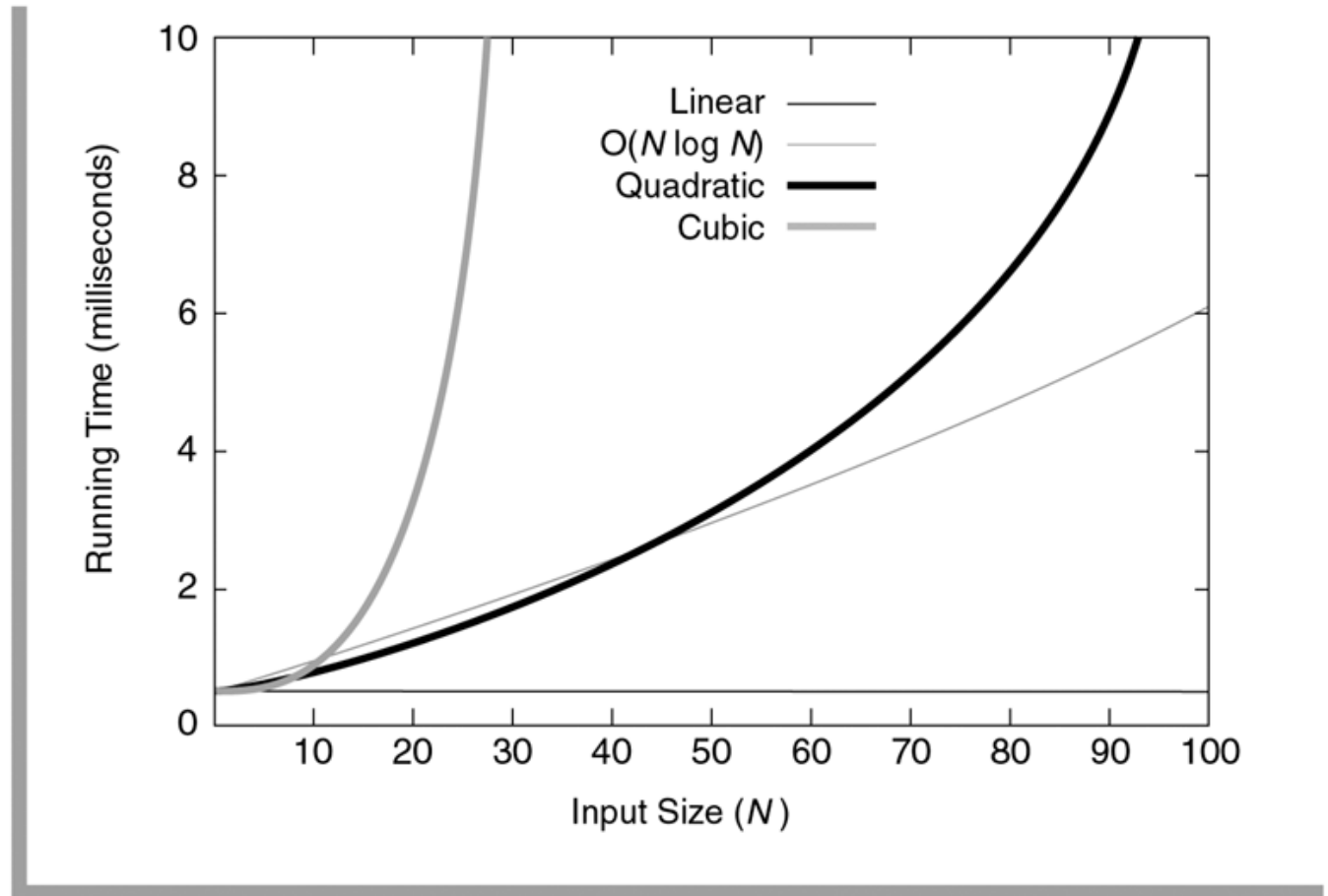
Algorithms and Input size

- The amount of time that any algorithm takes to run almost always **depends on the amount of input** that it must process.
- We expect, for instance, that sorting 10,000 elements requires more time than sorting 10 elements.
- The running time of an algorithm is thus a **function of the input size**.
- The **exact value** of the function depends on many factors, such as the speed of the **host machine**, the quality of the **compiler**, and in some cases, the quality of the **program**.

Running times for small inputs

figure 5.1

Running times for small inputs



Example of linear algorithm

- An example is the problem of downloading a file over the Internet.
- Suppose there is an initial 2-sec delay (to set up a connection), after which the download proceeds at 1.6 K/sec.
- Then if the file is N kilobytes, the time to download is described by the formula $T(N) = N/1.6 + 2$. This is a **linear function**.
- Downloading an 80K file takes approximately 52 sec, whereas downloading a file twice as large (160K) takes about 102 sec, or roughly twice as long.
- This property, in which time essentially is **directly proportional to amount of input**, is the signature of a **linear algorithm**, which is the most efficient algorithm.

Important issues

- Is it always important to be on the most efficient curve?
- How much better is one curve than another?
- How do you decide which curve a particular algorithm lies on?
- How do you design algorithms that avoid being on less-efficient curves?

Running times for moderate inputs

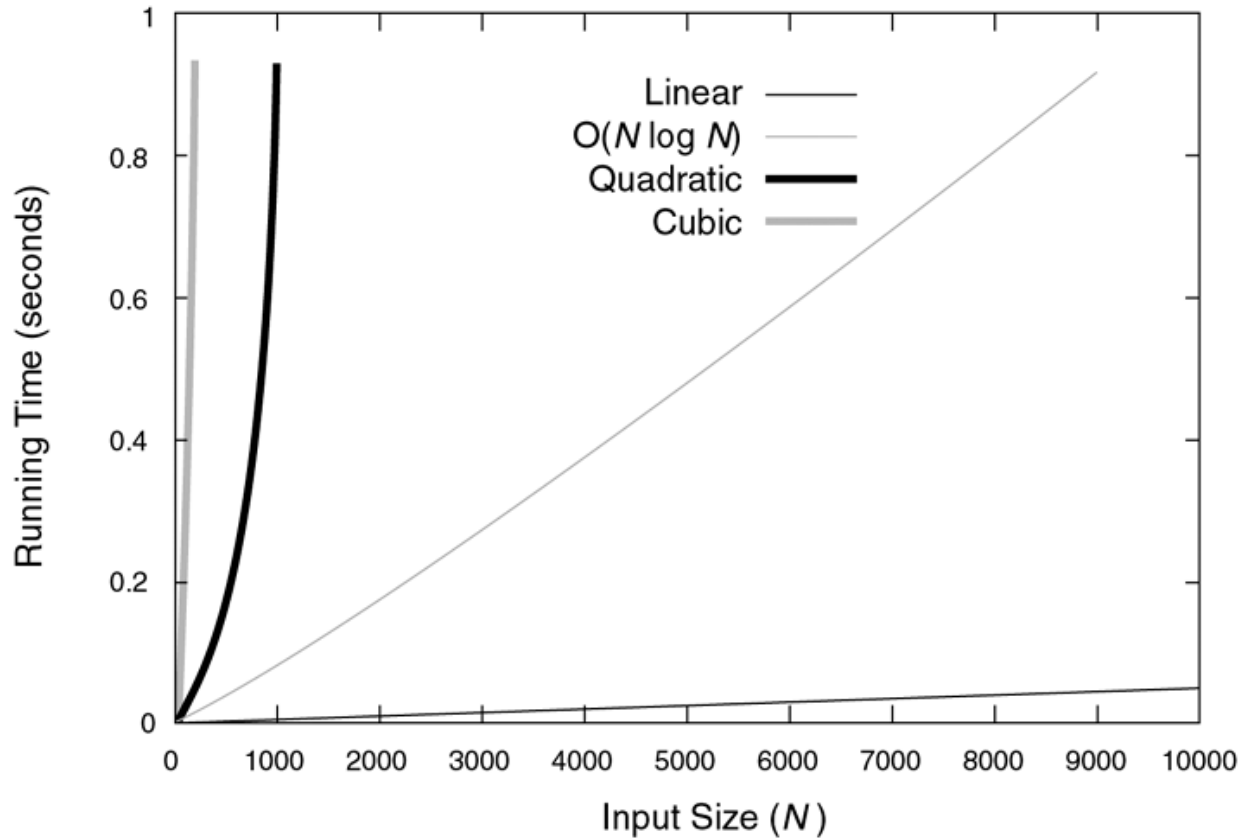


figure 5.2

Running times for moderate inputs

Cubic function

- A **cubic function** is a function whose dominant term is some constant times N^3 . As an example, $10N^3 + N^2 + 40N + 80$ is a cubic function.
- Similarly, a **quadratic function** has a dominant term that is some constant times N^2 , and a linear function has a dominant term that is some constant times N .
- The expression $O(N \log N)$ represents a function whose dominant term is **N times the logarithm of N** .
- The logarithm is a **slowly growing function**; for instance, the logarithm of 1,000,000 (with the typical base 2) is only 20. The logarithm **grows more slowly** than a square or cube (or any) root.

Growth rate of a function

- Either of two functions may be smaller than the other at any given point, so claiming, for instance, that $F(N) < G(N)$ does not make sense.
- Instead, we measure the **functions' rates of growth**.
- This is justified for three reasons.
- First, for cubic functions such as the one shown in Figure 5.2, when N is 1,000 the value of the cubic function is almost **entirely determined by the cubic term**. In the function $10N^3 + N^2 + 40N + 80$, for $N = 1,000$, the value of the function is 10,001,040,080, of which 10,000,000,000 is due to the $10N^3$ term.
- If we were to use only the cubic term to estimate the entire function, **an error of about 0.01 percent** would result.
- For sufficiently large N , the value of a function is **largely determined by its dominant term** (the meaning of the term sufficiently large varies by function).

Growth rate of a function

- The second reason we measure the functions' growth rates is that **the exact value of the leading constant** of the dominant term is **not meaningful** across different machines
 - For instance, the **quality of the compiler** could have a large influence on the leading constant.
- The third reason is that **small values of N generally are not important**. For $N = 20$, Figure 5.1 shows that all algorithms terminate within 5 ms.
 - The difference between the best and worst algorithm is less than a blink of the eye.

Big-Oh notation

- We use Big-Oh notation **to capture the most dominant term** in a function and to represent the growth rate.
- For instance, the running time of a quadratic algorithm is specified as $O(N^2)$ (pronounced "order en-squared").
- Big-Oh notation also allows us to **establish a relative order** among functions by comparing dominant terms.

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

figure 5.3

Functions in order of increasing growth rate

Small inputs

- For small amounts of input, making comparisons between functions is difficult because leading constants become very significant.
- The function $N + 2,500$ is larger than N^2 when N is less than 50.
- Eventually, the linear function is always less than the quadratic function.
- Most important, **for small input sizes the running times are generally inconsequential**, so we need not worry about them.
- Figure 5.1 shows that when N is less than 25, all four algorithms run in less than 10 ms.
- **Consequently, when input sizes are very small, a good rule of thumb is to use the simplest algorithm.**

Optimizing the algorithm

- The most striking feature of these curves is that the quadratic and cubic algorithms are **not competitive** with the others for reasonably large inputs.
- We can code the quadratic algorithm in **highly efficient machine language** and do a poor job coding the linear algorithm, and the quadratic algorithm will **still lose badly**.
- Even the most clever programming **tricks** cannot make an **inefficient** algorithm fast.
- Thus, before we waste effort attempting to optimize code, **we need to optimize the algorithm**.

Examples of algorithm running times

Minimum element in an array

- Given an array of N items, find the smallest item.
- The minimum element problem is fundamental in computer science. It can be solved as follows:
 1. Maintain a variable `min` that stores the minimum element.
 2. Initialize `min` to the first element.
 3. Make a sequential scan through the array and update `min` as appropriate.
- The running time of this algorithm will be $O(N)$, or linear, because we will repeat a fixed amount of work for each element in the array.

Closest points in the plane

- Given N points in a plane (that is, an x-y coordinate system), find the pair of points that are closest together.
- The closest points problem is a fundamental problem in graphics that can be solved as follows:
 1. Calculate the distance between each pair of points.
 2. Retain the minimum distance.
- This calculation is expensive, however, because there are $N(N - 1)/2$ pairs of points.
- Thus there are roughly N^2 pairs of points. Examining all these pairs and finding the minimum distance among them takes quadratic time.
- (Beyond the scope of this course: A better algorithm runs in $O(N \log N)$ time and works by avoiding the computation of all distances. There is also an algorithm that is expected to take $O(N)$ time.)

Analysis of algorithms

- Problem
 - The maximum contiguous subsequence sum problem
- Given (possibly negative) integers A_1, A_2, \dots, A_N , find (and identify the sequence corresponding to) the maximum value of $\text{SUM}(A_k)$.
 - The maximum contiguous subsequence sum is zero if all the integers are negative.
- As an example, if the input is $\{-2, \mathbf{11}, \mathbf{-4}, \mathbf{13}, -5, 2\}$, then the answer is 20, which represents the contiguous subsequence encompassing items 2 through 4 (shown in boldface type).
- As a second example, for the input $\{1, -3, \mathbf{4}, \mathbf{-2}, \mathbf{-1}, \mathbf{6}\}$, the answer is 7 for the subsequence encompassing the last four items.

Algorithms that solve the problem

- There are lots of drastically different algorithms (in terms of efficiency) that can be used to solve the maximum contiguous subsequence sum problem.

Cubic: a brute force algorithm

```
1  /**
2   * Cubic maximum contiguous subsequence sum algorithm.
3   * seqStart and seqEnd represent the actual best sequence.
4   */
5  public static int maxSubsequenceSum( int [ ] a )
6  {
7      int maxSum = 0;
8
9      for( int i = 0; i < a.length; i++ )
10         for( int j = i; j < a.length; j++ )
11             {
12                 int thisSum = 0;
13
14                 for( int k = i; k <= j; k++ )
15                     thisSum += a[ k ];
16
17                 if( thisSum > maxSum )
18                     {
19                         maxSum = thisSum;
20                         seqStart = i;
21                         seqEnd   = j;
22                     }
23             }
24
25     return maxSum;
26 }
```

figure 5.4

A cubic maximum
contiguous
subsequence sum
algorithm

**The dominant
term**



Analysis of the algorithm

- Four expressions that are repeatedly executed:
 1. The initialization $k = i$
 2. The test $k \leq j$
 3. The increment $\text{thisSum} += a[k]$
 4. The adjustment $k++$
- The number of times expression 3 is executed makes it **the dominant term among the four expressions.**

Analysis of the algorithm

- The number of times line 15 is executed is exactly equal to the number of ordered triplets (i, j, k) that satisfy $1 < i < k < j < N$.
- The reason is that the index i runs over the entire array, j runs from i to the end of the array, and k runs from i to j .
- A quick and dirty estimate is that the number of triplets is somewhat less than $N \times N \times N$, or N^3 , because i , j , and k can each assume one of N values.

Theorem 5.1

- The number of integer-ordered triplets (i, j, k) that satisfy $1 < i < k < j < N$ is $N(N+1)(N+2)/6$.
- The result of Theorem 5.1 is that the innermost for loop accounts for cubic running time.

Big-Oh estimation

- The previous combinatorial argument allows us to obtain **precise calculations** on the number of iterations in the inner loop.
- For a Big-Oh calculation, this is **not really necessary**; we need to know only that the **leading term** is some constant times N^3 .
- Looking at the algorithm, we see a **loop that is potentially of size N inside a loop that is potentially of size N inside another loop that is potentially of size N**.
- This configuration tells us that the triple loop has the potential for $N \times N \times N$ iterations.
- This potential is only about six times higher than what our precise calculation of what actually occurs.

An improved algorithm: quadratic

figure 5.5

A quadratic maximum contiguous subsequence sum algorithm

```
1  /**
2   * Quadratic maximum contiguous subsequence sum algorithm.
3   * seqStart and seqEnd represent the actual best sequence.
4   */
5  public static int maxSubsequenceSum( int [ ] a )
6  {
7      int maxSum = 0;
8
9      for( int i = 0; i < a.length; i++ )
10     {
11         int thisSum = 0;
12
13         for( int j = i; j < a.length; j++ )
14         {
15             thisSum += a[ j ];
16
17             if( thisSum > maxSum )
18             {
19                 maxSum = thisSum;
20                 seqStart = i;
21                 seqEnd   = j;
22             }
23         }
24     }
25
26     return maxSum;
27 }
```

How about a linear algorithm?

- We need to remove another loop.
- The only way we can attain a subquadratic bound is to find a clever way to eliminate from consideration a large number of subsequences, **without actually computing their sum and testing** to see if that sum is a new maximum.

A linear algorithm

figure 5.8

A linear maximum contiguous subsequence sum algorithm

```
1  /**
2   * Linear maximum contiguous subsequence sum algorithm.
3   * seqStart and seqEnd represent the actual best sequence.
4   */
5  public static int maximumSubsequenceSum( int [ ] a )
6  {
7      int maxSum = 0;
8      int thisSum = 0;
9
10     for( int i = 0, j = 0; j < a.length; j++ )
11     {
12         thisSum += a[ j ];
13
14         if( thisSum > maxSum )
15         {
16             maxSum = thisSum;
17             seqStart = i;
18             seqEnd   = j;
19         }
20         else if( thisSum < 0 )
21         {
22             i = j + 1;
23             thisSum = 0;
24         }
25     }
26
27     return maxSum;
28 }
```

One for loop



General big-oh rules

definition: (Big-Oh) $T(N)$ is $O(F(N))$ if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$.

definition: (Big-Omega) $T(N)$ is $\Omega(F(N))$ if there are positive constants c and N_0 such that $T(N) \geq cF(N)$ when $N \geq N_0$.

definition: (Big-Theta) $T(N)$ is $\Theta(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is $\Omega(F(N))$.

definition: (Little-Oh) $T(N)$ is $o(F(N))$ if and only if $T(N)$ is $O(F(N))$ and $T(N)$ is not $\Theta(F(N))$.

Interpretation of notations

figure 5.9

Meanings of the various growth functions

Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is \leq growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is \geq growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$.

Observed running times

N	Figure 5.4 $O(N^3)$	Figure 5.5 $O(N^2)$	Figure 7.20 $O(N \log N)$	Figure 5.8 $O(N)$
10	0.000009	0.000004	0.000006	0.000003
100	0.002580	0.000109	0.000045	0.000006
1,000	2.281013	0.010203	0.000485	0.000031
10,000	NA	1.2329	0.005712	0.000317
100,000	NA	135	0.064618	0.003206

figure 5.10

Observed running times (in seconds) for various maximum contiguous subsequence sum algorithms

Ooopss...

The Searching Problem

- Static searching problem
 - Given an integer X and an array A , return the position of X in A or an indication that it is not present.
 - If X occurs more than once, return any occurrence. The array A is never altered.

Sequential search

- When the input array is not sorted, we have little choice but to do a **linear sequential search**, which steps through the array sequentially until a match is found.
- The complexity of the algorithm is analyzed in three ways.
 - First, we provide the **cost of an unsuccessful search**.
 - Then, we give the **worst-case cost of a successful search**.
 - Finally, we find the **average cost of a successful search**.

Worst cases

- An **unsuccessful search** requires the examination of every item in the array, so the time will be $O(N)$.
- In the worst case, a successful search, too, requires **the examination of every item in the array** because we might not find a match until the last item.
 - Thus the worst-case running time for a successful search is also **linear**.
- On average, however, we search only half of the array.
- However, $N/2$ is still $O(N)$.

Binary search

- If the input array has been sorted, we have an alternative to the sequential search, the **binary search**, which is performed from the middle of the array rather than the end.

Binary search

figure 5.12

Binary search using
two-way comparisons

```
1  /**
2   * Performs the standard binary search
3   * using one comparison per level.
4   * @return index where item is found or NOT_FOUND.
5   */
6  public static <AnyType extends Comparable<? super AnyType>>
7      int binarySearch( AnyType [ ] a, AnyType x )
8  {
9      if( a.length == 0 )
10         return NOT_FOUND;
11
12     int low = 0;
13     int high = a.length - 1;
14     int mid;
15
16     while( low < high )
17     {
18         mid = ( low + high ) / 2;
19
20         if( a[ mid ].compareTo( x ) < 0 )
21             low = mid + 1;
22         else
23             high = mid;
24     }
25
26     if( a[ low ].compareTo( x ) == 0 )
27         return low;
28
29     return NOT_FOUND;
30 }
```

the number of iterations
will be $O(\log N)$.

Limitations of Big-oh

- Big-Oh analysis is a very effective tool, but it does have limitations.
- As already mentioned, its use is **not appropriate for small amounts of input**.
- For small amounts of input, **use the simplest algorithm**.
- Also, for a particular algorithm, the **constant** implied by the Big-Oh may be **too large to be practical**.
- For example, if one algorithm's running time is governed by the formula $2N \log N$ and another has a running time of $1000N$, then the first algorithm would most likely be **better**, even though **its growth rate is larger**.

Limitations of Big-oh

- Large constants can come into play when an algorithm is **excessively complex**.
- They also come into play because our analysis disregards constants and thus cannot differentiate between things like **memory access** (which is cheap) and **disk access** (which typically is many thousand times more expensive).
- Our analysis assumes **infinite memory**, but in applications involving **large data sets**, lack of sufficient memory can be a severe problem.

End of PART I

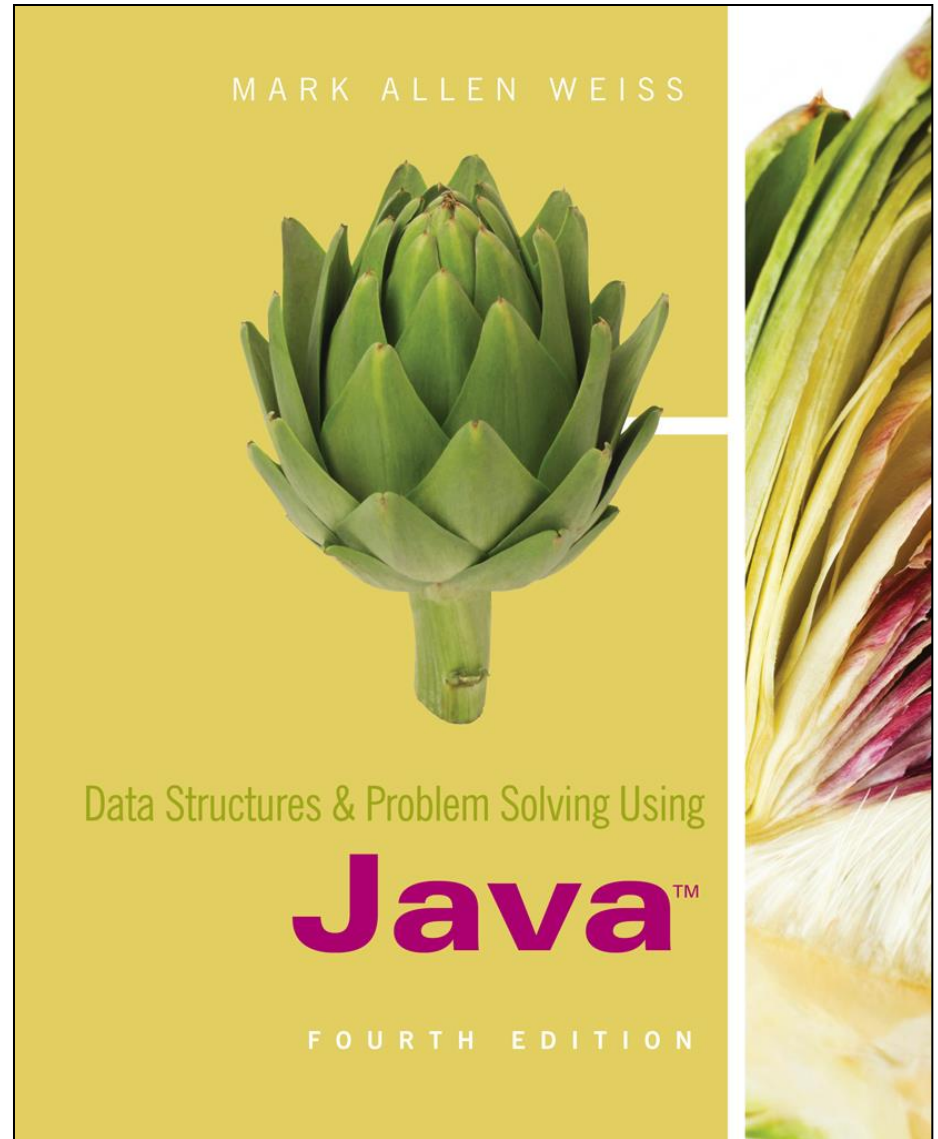
- Readings
 - Chapter 5

PART II

- Linked Lists

Chapter 17

Linked Lists



Addison-Wesley
is an imprint of

PEARSON

Copyright © 2010 Pearson Education, publishing as Addison-Wesley. All rights reserved

Linked Lists

- With stacks and queues we have demonstrated that linked lists can be used to store items noncontiguously.
- The linked lists used until now were simplified, with all the accesses performed at one of the list's two ends.
- We will now show:
 - How to allow access to any item by using a general linked list
 - The general algorithms for the linked list operations
 - How the iterator class provides a safe mechanism for traversing and accessing linked lists.

Singly linked list

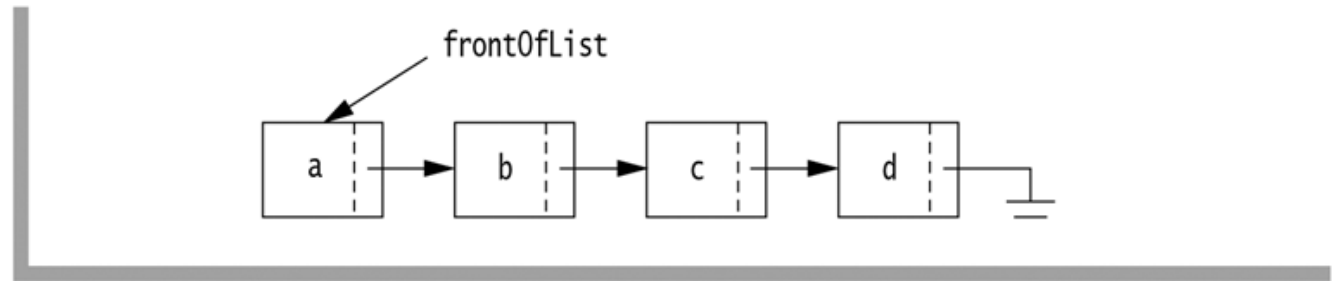
- The basic linked list consists of a collection of connected, dynamically allocated nodes.
- In a singly linked list, each node consists of the data element and a link to the next node in the list.
- The last node in the list has a null next link.

ListNode

```
class ListNode
{
    Object element;
    ListNode next;
}
```

Basic linked list

figure 17.1
Basic linked list

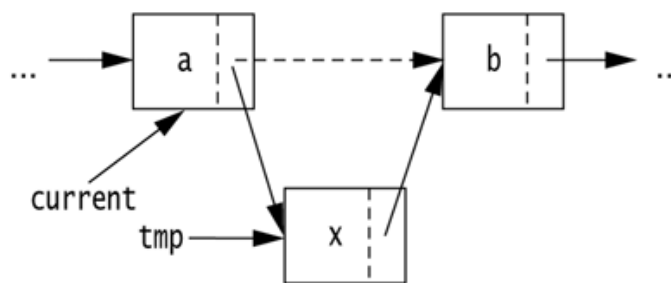


We can print or search in the linked list by starting at the first item and **following the chain of next links**. The two basic operations that must be performed are **insertion** and **deletion** of an arbitrary item x.

Insertion

figure 17.2

Insertion in a linked list: Create new node (tmp), copy in x, set tmp's next link, and set current's next link.



In code:

```
tmp = new ListNode( ); // Create a new node
tmp.element = x; // Place x in the element member
tmp.next = current.next; // x's next node is b
current.next = tmp; // a's next node is x
```

Insertion

- We can simplify the code if the ListNode has a constructor that initializes the data members directly. In that case, we obtain:

```
tmp = new ListNode( x, current.next );    // Create new node
current.next = tmp;                       // a's next node is x
```

- We now see that tmp is no longer necessary. Thus we have the one-liner

```
current.next = new ListNode( x, current.next );
```

Deletion

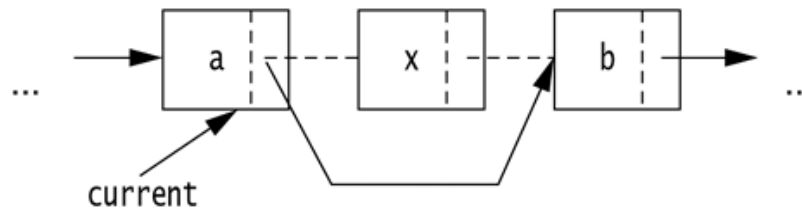


figure 17.3

Deletion from a linked list

To remove item *x* from the linked list, we set **current** to be the node prior to *x* and then have current's next link bypass *x*.

In code:

```
current.next = current.next.next;
```

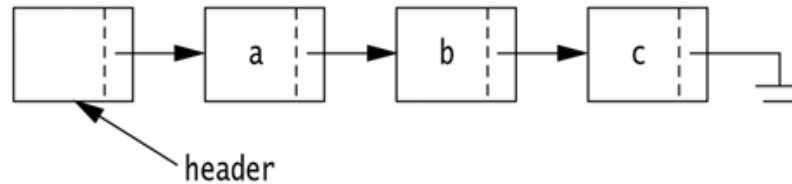
Header nodes

- There is one problem with the basic description: It assumes that whenever an item x is **removed**, some **previous item** is always present to **allow a bypass**.
- Consequently, removal of the first item in the linked list becomes a **special case**.
- Similarly, the insert routine does not allow us to insert an item to be the **new first element in the list**.
- The reason is that insertions must follow some existing item.
- So, although the basic algorithm works fine, some annoying special cases must be dealt with.

Header node

figure 17.4

Using a header node
for the linked list



A header node holds no data but serves to satisfy the **requirement that every node have a previous node**.

A header node allows us to avoid special cases such as insertion of a new first element and removal of the first element.

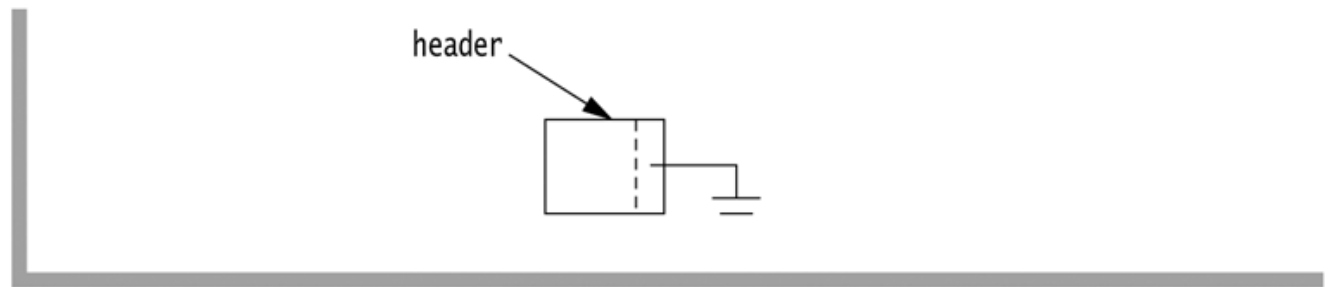
Header node

- Referring to Fig. 17.4, note that **a** is no longer a special case.
- It can be deleted just like any other node by having **current reference the node before it**.
- We can also add a new first element to the list by setting **current equal to the header node** and calling the insertion routine.
- By using the header node, we greatly simplify the code — with a negligible space penalty.

Use of header in isEmpty

figure 17.5

Empty list when a header node is used



With a dummy header node, a list is empty if `header.next` is null.

Implementation

The current position

- By storing a **current position** in a list class, we ensure that access is controlled.
- As all access to the list goes through the class methods, we can be certain that **current** always represents a node in the list, the header node, or null.

Iterators

- The scheme with **current** has a problem: With only one position, the case of **two iterators needing to access the list independently** is left unsupported.
- One way to avoid this problem is to define a **separate iterator class**, which maintains a notion of its current position.
- A list class would then **not maintain any notion of a current position** and would only have methods that treat the list as a unit, such as `isEmpty` and `makeEmpty`, or that accept an iterator as a parameter, such as `insert`.

Iterators

- Routines that depend only on an iterator itself, such as the advance routine that advances the iterator to the next position, would **reside in the iterator class**.
- Access to the list is granted by making the iterator class either **package-visible or an inner class**.
- We can view each instance of an iterator class as one in which **only legal list operations**, such as advancing in the list, are allowed.

Example: Use of iterator

```
1  // In this routine, LinkedList and LinkedListIterator are the
2  // classes written in Section 17.2.
3  public static <AnyType> int listSize( LinkedList<AnyType> theList )
4  {
5      LinkedListIterator<AnyType> itr;
6      int size = 0;
7
8      for( itr = theList.first(); itr.isValid(); itr.advance() )
9          size++;
10
11     return size;
12 }
```

figure 17.6

A static method that returns the size of a list

We initialize itr to the first element in theList (skipping over the header, of course) by referencing the iterator given by the **List.first()**.

The test **itr.isValid()** attempts to mimic the test **p!=null** that would be conducted if p were a visible reference to a node. Finally, the expression **itr.advance()** mimics the conventional idiom **p=p.next**.

```

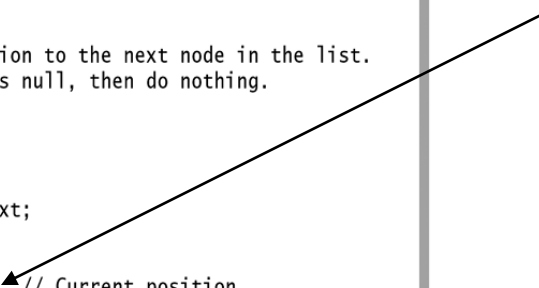
1 package weiss.nonstandard;
2
3 // LinkedListIterator class; maintains "current position"
4 //
5 // CONSTRUCTION: Package visible only, with a ListNode
6 //
7 // *****PUBLIC OPERATIONS*****
8 // void advance( )      --> Advance
9 // boolean isValid( )   --> True if at valid position in list
10 // AnyType retrieve     --> Return item in current position
11
12 public class LinkedListIterator<AnyType>
13 {
14     /**
15      * Construct the list iterator
16      * @param theNode any node in the linked list.
17      */
18     LinkedListIterator( ListNode<AnyType> theNode )
19     { current = theNode; }
20
21     /**
22      * Test if the current position is a valid position in the list.
23      * @return true if the current position is valid.
24      */
25     public boolean isValid( )
26     { return current != null; }
27
28     /**
29      * Return the item stored in the current position.
30      * @return the stored item or null if the current position
31      * is not in the list.
32      */
33     public AnyType retrieve( )
34     { return isValid( ) ? current.element : null; }
35
36     /**
37      * Advance the current position to the next node in the list.
38      * If the current position is null, then do nothing.
39      */
40     public void advance( )
41     {
42         if( isValid( ) )
43             current = current.next;
44     }
45
46     ListNode<AnyType> current; // Current position
47 }

```

figure 17.7

The
LinkedListIterator
class

**Current: helps to
maintain the notion
of the current
position**



```

1 package weiss.nonstandard;
2
3 // LinkedList class
4 //
5 // CONSTRUCTION: with no initializer
6 // Access is via LinkedListIterator class
7 //
8 // *****PUBLIC OPERATIONS*****
9 // boolean isEmpty( )    --> Return true if empty; else false
10 // void makeEmpty( )    --> Remove all items
11 // LinkedListIterator zeroth( )
12 //                      --> Return position to prior to first
13 // LinkedListIterator first( )
14 //                      --> Return first position
15 // void insert( x, p )   --> Insert x after current iterator position p
16 // void remove( x )     --> Remove x
17 // LinkedListIterator find( x )
18 //                      --> Return position that views x
19 // LinkedListIterator findPrevious( x )
20 //                      --> Return position prior to x
21 // *****ERRORS*****
22 // No special errors
23
24 public class LinkedList<AnyType>
25 {
26     public LinkedList( )
27     { /* Figure 17.9 */ }
28
29     public boolean isEmpty( )
30     { /* Figure 17.9 */ }
31     public void makeEmpty( )
32     { /* Figure 17.9 */ }
33     public LinkedListIterator<AnyType> zeroth( )
34     { /* Figure 17.9 */ }
35     public LinkedListIterator<AnyType> first( )
36     { /* Figure 17.9 */ }
37     public void insert( AnyType x, LinkedListIterator<AnyType> p )
38     { /* Figure 17.14 */ }
39     public LinkedListIterator<AnyType> find( AnyType x )
40     { /* Figure 17.11 */ }
41     public LinkedListIterator<AnyType> findPrevious( AnyType x )
42     { /* Figure 17.13 */ }
43     public void remove( Object x )
44     { /* Figure 17.12 */ }
45
46     private ListNode<AnyType> header;
47 }

```

Header node

figure 17.8

The LinkedList class skeleton

```

1  /**
2   * Construct the list
3   */
4  public LinkedList( )
5  {
6      header = new ListNode<AnyType>( null );
7  }
8
9  /**
10 * Test if the list is logically empty.
11 * @return true if empty, false otherwise.
12 */
13 public boolean isEmpty( )
14 {
15     return header.next == null;
16 }
17
18 /**
19 * Make the list logically empty.
20 */
21 public void makeEmpty( )
22 {
23     header.next = null;
24 }
25
26 /**
27 * Return an iterator representing the header node.
28 */
29 public LinkedListIterator<AnyType> zeroth( )
30 {
31     return new LinkedListIterator<AnyType>( header );
32 }
33
34 /**
35 * Return an iterator representing the first node in the list.
36 * This operation is valid for empty lists.
37 */
38 public LinkedListIterator<AnyType> first( )
39 {
40     return new LinkedListIterator<AnyType>( header.next );
41 }

```

figure 17.9

Some LinkedList
class one-liners

static printList(List)

```
1  // Simple print method
2  public static <AnyType> void printList( LinkedList<AnyType> theList )
3  {
4      if( theList.isEmpty( ) )
5          System.out.print( "Empty list" );
6      else
7      {
8          LinkedListIterator<AnyType> itr = theList.first( );
9          for( ; itr.isValid( ); itr.advance( ) )
10             System.out.print( itr.retrieve( ) + " " );
11      }
12
13     System.out.println( );
14 }
```

figure 17.10

A method for printing the contents of a LinkedList

find(X)

```
1  /**
2   * Return iterator corresponding to the first node containing x.
3   * @param x the item to search for.
4   * @return an iterator; iterator isPastEnd if item is not found.
5   */
6  public LinkedListIterator<AnyType> find( AnyType x )
7  {
8      ListNode<AnyType> itr = header.next;
9
10     while( itr != null && !itr.element.equals( x ) )
11         itr = itr.next;
12
13     return new LinkedListIterator<AnyType>( itr );
14 }
```

figure 17.11

The find routine for the LinkedList class

remove(X)

```
1  /**
2   * Remove the first occurrence of an item.
3   * @param x the item to remove.
4   */
5  public void remove( AnyType x )
6  {
7      LinkedListIterator<AnyType> p = findPrevious( x );
8
9      if( p.current.next != null )
10         p.current.next = p.current.next.next;  // Bypass deleted node
11  }
```

figure 17.12

The remove routine for the LinkedList class

findPrevious(X)

```
1  /**
2   * Return iterator prior to the first node containing an item.
3   * @param x the item to search for.
4   * @return appropriate iterator if the item is found. Otherwise, the
5   * iterator corresponding to the last element in the list is returned.
6   */
7  public LinkedListIterator<AnyType> findPrevious( AnyType x )
8  {
9      ListNode<AnyType> itr = header;
10
11      while( itr.next != null && !itr.next.element.equals( x ) )
12          itr = itr.next;
13
14      return new LinkedListIterator<AnyType>( itr );
15  }
```

figure 17.13

The findPrevious routine—similar to the find routine—for use with remove

insert (X)

```
1  /**
2   * Insert after p.
3   * @param x the item to insert.
4   * @param p the position prior to the newly inserted item.
5   */
6  public void insert( AnyType x, LinkedListIterator<AnyType> p )
7  {
8      if( p != null && p.current != null )
9          p.current.next = new ListNode<AnyType>( x, p.current.next );
10 }
```

figure 17.14

The insertion routine for the `LinkedList` class

End of PART II

- Readings
 - Chapter 17

Lab exercises

Perform the following on the given implementation:

- Modify the find routine in the nonstandard LinkedList class to return the last position of item x.
- Modify remove in the nonstandard LinkedList class to remove all occurrences of x.
- Clone: replicate a list in another list.
- FindMinimum: find the smallest element in the list
 - Hint: you need to use the following signature:
public AnyType findMinimum(Comparator<AnyType> cmp)
 - Hint: Define a comparator similar to the queue exercise
 - Hint: define BookComparator and the method compare that returns -1, 0, or 1.
- Test all these in a testing class

