Chapter 7 Searching

数据科学与计算机学院　　　黄方军

data_structures@163.com

东校区实验中心B502

7.1 Searching: Introduction and Notation

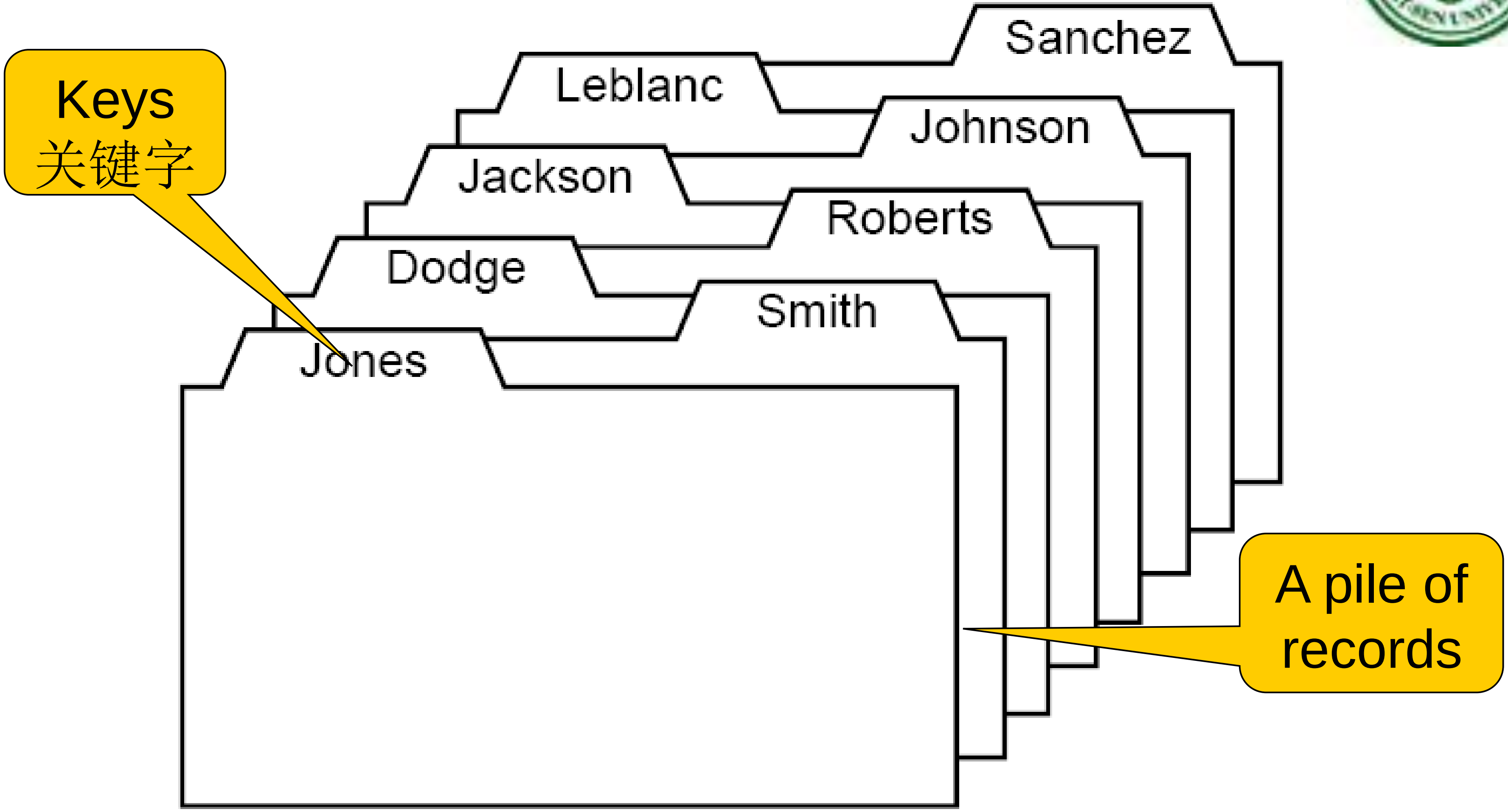


Figure 7.1. Records and their keys

7.1 Searching: Introduction and Notation

The records that are stored in a list being searched must conform to the following minimal standards:

- Every record is associated to a key.
- Keys can be compared for equality or relative ordering.
- Records can be compared to each other or to keys by first converting records to their associated keys.

In applications the conversion operation could be one of the following:

- A method of the **class** Record, with the declaration **operator** Key( ) **const**;
- A constructor for the class Key, with the declaration Key(**const** Record &);
- Or, if the classes Key and Record are identical, no conversion needs to be defined, since any Record is automatically a Key.

```
typedef int Key;
typedef int Record;
```

```
typedef Key Record;
```

7.1 Searching: Introduction and Notation

```
class Key{
  public:
    //    Add any constructors and methods for key data.
  private:
    //    Add declaration of key data members here.
};
//    Declare overloaded comparison operators for keys.
bool operator ==  (const Key &x, const Key &y);
bool operator >   (const Key &x, const Key &y);
bool operator <   (const Key &x, const Key &y);
bool operator >=  (const Key &x, const Key &y);
bool operator <=  (const Key &x, const Key &y);
bool operator !=  (const Key &x, const Key &y);
```

7.1 Searching: Introduction and Notation

```
class Record{
  public:
    operator Key(); // implicit conversion from Record to Key.
  //    Add any constructors and methods for Record objects.
  private:
  //    Add data components.
};
```

7.2 Sequential Search

```
Error_code sequential_search(const List<Record> &the_list,
                                  const Key &target, int &position)
{
   int s = the_list.size();
   for (position = 0;  position < s;  position++) {
      Record data;
      the_list.retrieve(position, data);
      if (data ==  target) return success;
   }
   return not_present;
}
```

The average number of key comparisons (*):

$$\frac{1+2+3+\cdots+n}{n} = \frac{1}{2}(n+1).$$

7.2 Sequential Search

```
class Key {
    int key;
public:
    static int comparisons;
    Key (int x = 0);
    int the_key( ) const;
};
bool operator ==  (const Key &x, const Key &y);
bool operator >   (const Key &x, const Key &y);
bool operator <   (const Key &x, const Key &y);
bool operator >=  (const Key &x, const Key &y);
bool operator <=  (const Key &x, const Key &y);
bool operator !=  (const Key &x, const Key &y);
```

7.2 Sequential Search

```
int Key::comparisons = 0;

int Key::the_key() const
{
   return key;
}
Key::Key (int x)
{
   key = x;
}
bool operator ==(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() == y.the_key();
}
```

7.2 Sequential Search

```
bool operator !=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() != y.the_key();
}
bool operator >=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() >= y.the_key();
}
bool operator <=(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() <= y.the_key();
}
```

7.2 Sequential Search

```
bool operator >(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() > y.the_key();
}

bool operator <(const Key &x, const Key &y)
{
   Key::comparisons++;
   return x.the_key() < y.the_key();
}
```

7.2 Sequential Search

```
typedef Key Record;
…
int main()
{
   int list_size = 20, searches = 100;
   cout << "Timing and testing of sequential …”<<
endl;
   int i;
   List<Record> the_list;
   for (i = 0; i < list_size; i++)
      if (the_list.insert(i, 2 * i + 1) != success) {
         cout << " Overflow in filling list." << endl;
      }
}
```

7.3 Binary Search

- 基于上一章的List程序来完成；
- List中所有原始数据均已排序；
- 了解继承、函数覆盖、函数重载等。

7.3.1 Ordered Lists

```
class Ordered_list: public List<Record>{
public:
   Ordered_list();
   Error_code insert(const Record &data);
   Error_code insert(int position, const Record &data);
   Error_code replace(int position, const Record &data);
};
```

7.3.1 Ordered Lists

```
Ordered_list::Ordered_list()
{
}

Error_code Ordered_list::insert(const Record &data)
{
   int s = size();
   int position;
   for (position = 0; position < s; position++) {
      Record list_data;
      retrieve(position, list_data);
      if (data <= list_data) break;
   }
   return List<Record>::insert(position, data);
}
```

7.3.1 Ordered Lists

```
Error_code  Ordered_list::insert(int position, const
Record &data)
{
   Record list_data;
   if (position > 0) {
      retrieve(position - 1, list_data);
      if (data < list_data)
         return fail;
   }
   if (position < size()) {
      retrieve(position, list_data);
      if (data > list_data)
         return fail;
   }
   return List<Record>::insert(position, data);
}
```

7.3.2 Algorithm Development

The method dates back at least to 1946, but the first version free of errors and unnecessary restrictions seems to have appeared only in 1962.

The target key, provided it is present in the list, will be found between the indices bottom and top, inclusive.

To do binary search, we first calculate the index mid halfway between bottom and top as

mid = (bottom + top)/2

Next, we note that binary search should terminate when top<=bottom; that is, when the remaining part of the list contains at most one item, providing that we have not terminated earlier by finding the target.

Trace of Binary Search

item = 84

15	26	38	57	62	78	84	91	108	119
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
first				middle					last

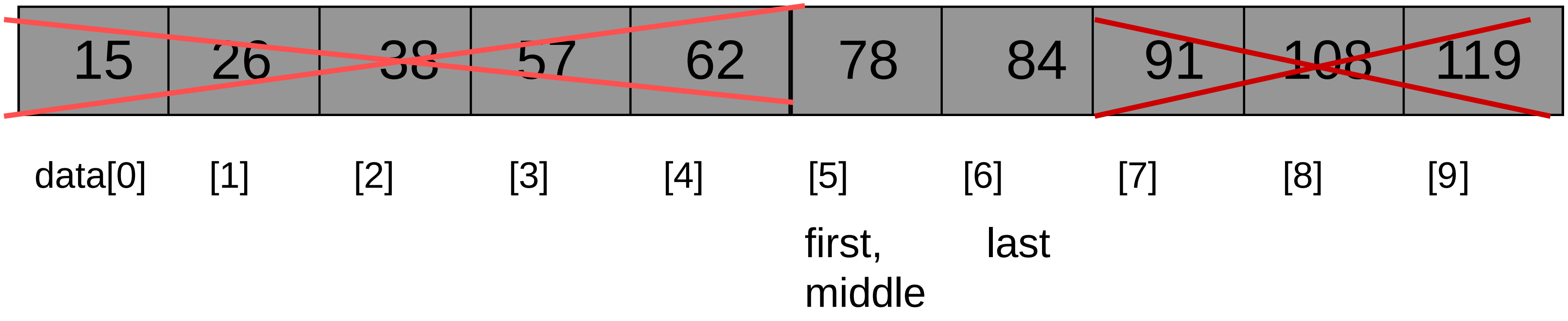
item > data [middle] ⇨ first = middle + 1

~~15~~	~~26~~	~~38~~	~~57~~	~~62~~	78	84	91	108	119
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
					first		middle		last

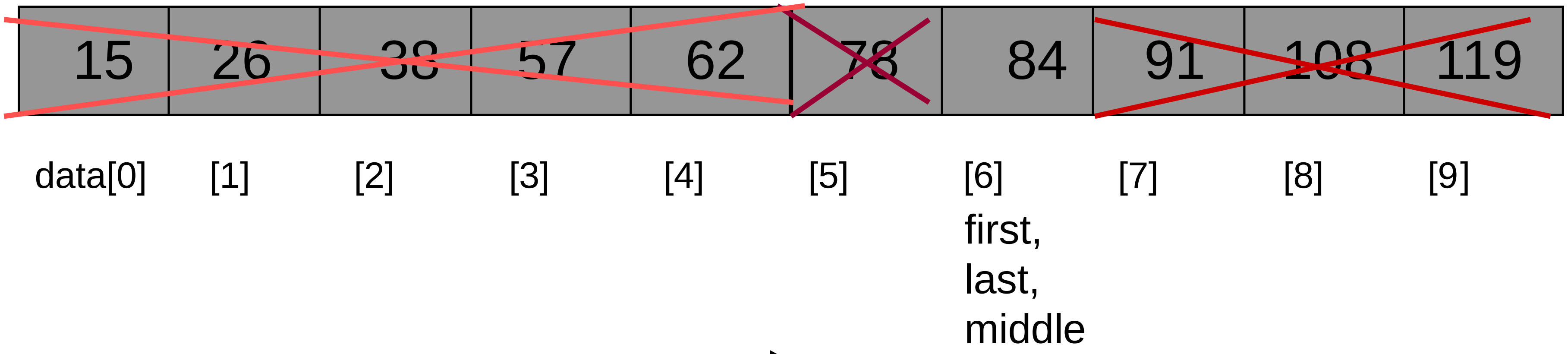
item < data [middle] ⇨ last = middle - 1

Trace continued

item = 84



item > data [middle] → first = middle + 1



item == data [middle] → found = true

Another Binary Search Trace

item = 45

15	26	38	57	62	78	84	91	108	119
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
first				middle					last

item < data [middle] ⇒ last = middle - 1

15	26	38	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
first	middle		last						

item > data [middle] ⇒ first = middle + 1

Trace continued

item = 45

~~15~~	~~26~~	38	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		first, middle	last						

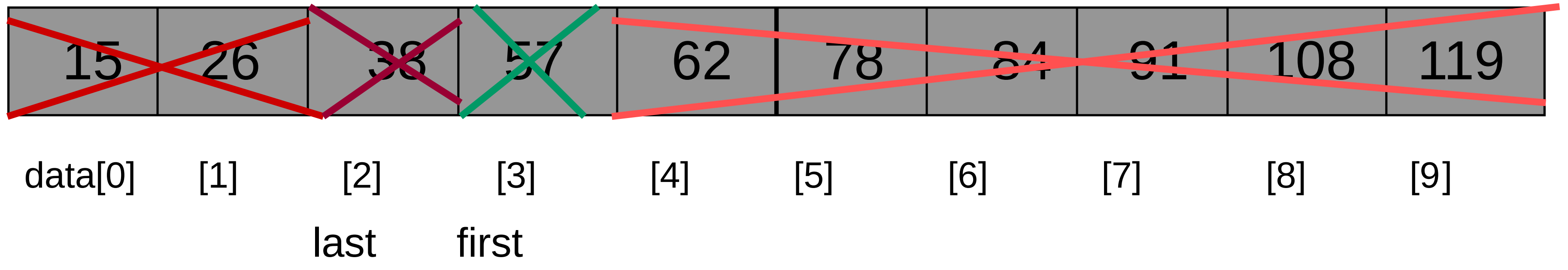
item > data [middle] ⇒ first = middle + 1

~~15~~	~~26~~	~~38~~	57	~~62~~	~~78~~	~~84~~	~~91~~	~~108~~	~~119~~
data[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
			first, middle, last						

item < data [middle] ⇒ last = middle - 1

Trace concludes

item = 45



first > last → found = false

7.3.3 The Forgetful Version

```
 Error_code recursive_binary_1(const Ordered_list &the_list, const Key
&target, int bottom, int top, int &position)
{
   Record data;
   if (bottom < top) {              // List has more than one entry.
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data < target)  // Reduce to top half of list.
         return recursive_binary_1(the_list, target, mid + 1, top, position);
      else                          // Reduce to bottom half of list.
         return recursive_binary_1(the_list, target, bottom, mid, position);
   }
   else if (top < bottom)
      return not_present;           // List is empty.
   else {                           // List has exactly one entry.
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
```

7.3.3 The Forgetful Version

```
 Error_code run_recursive_binary_1(const
Ordered_list &the_list, const Key &target, int
&position)
{
   return recursive_binary_1(the_list, target, 0,
                                       the_list.size() - 1, position);
}
```

7.3.3 The Forgetful Version

```
Error_code binary_search_1 (const Ordered_list &the_list,
                            const Key &target, int &position)
{
  Record data;
  int bottom = 0, top = the_list.size() - 1;
  while (bottom < top)    {
    int mid = (bottom + top) / 2;
    the_list.retrieve(mid, data);
    if (data < target)
      bottom = mid + 1;
    else
      top = mid;       }
  if (top < bottom)   return not_present;
  else {
    position = bottom;
    the_list.retrieve(bottom, data);
    if (data == target) return success;
    else return not_present;
  }
}
```

7.3.4 Recognizing Equality

```
Error_code recursive_binary_2(const Ordered_list &the_list, const Key
&target, int bottom, int top, int &position)
{
   Record data;
   if (bottom <= top) {
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data == target) {
         position = mid;
         return success;
      }
      else if (data < target)
         return recursive_binary_2(the_list, target, mid + 1, top, position);
      else
         return recursive_binary_2(the_list, target, bottom, mid - 1, position);
   }
   else return not_present;
}
```

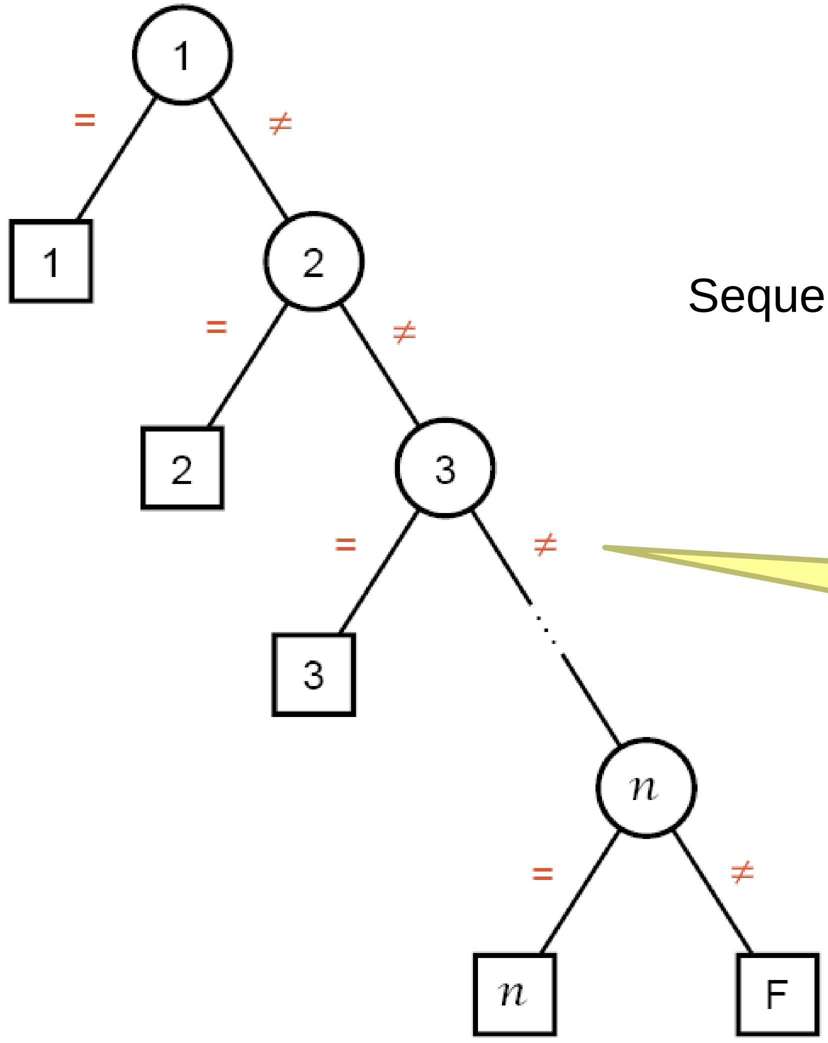
7.3.4 Recognizing Equality

```
Error_code run_recursive_binary_2(const Ordered_list
&the_list, const Key &target, int &position)
{
   return recursive_binary_2(the_list, target, 0,
                                             the_list.size() - 1, position);
}
```

7.3.4 Recognizing Equality

```
Error_code binary_search_2(const Ordered_list
&the_list, const Key &target, int &position)
{
   Record data;
   int bottom = 0, top = the_list.size() - 1;
   while (bottom <= top) {
      position = (bottom + top) / 2;
      the_list.retrieve(position, data);
      if (data == target) return success;
      if (data < target) bottom = position + 1;
      else top = position - 1;
   }
   return not_present;
}
```

7.4 Comparison Trees



Sequential Search

平均比较次数5.5次

Figure 7.2. Comparison tree for sequential_search

7.4.1 Analysis for n = 10

平均比较
次数4.4次

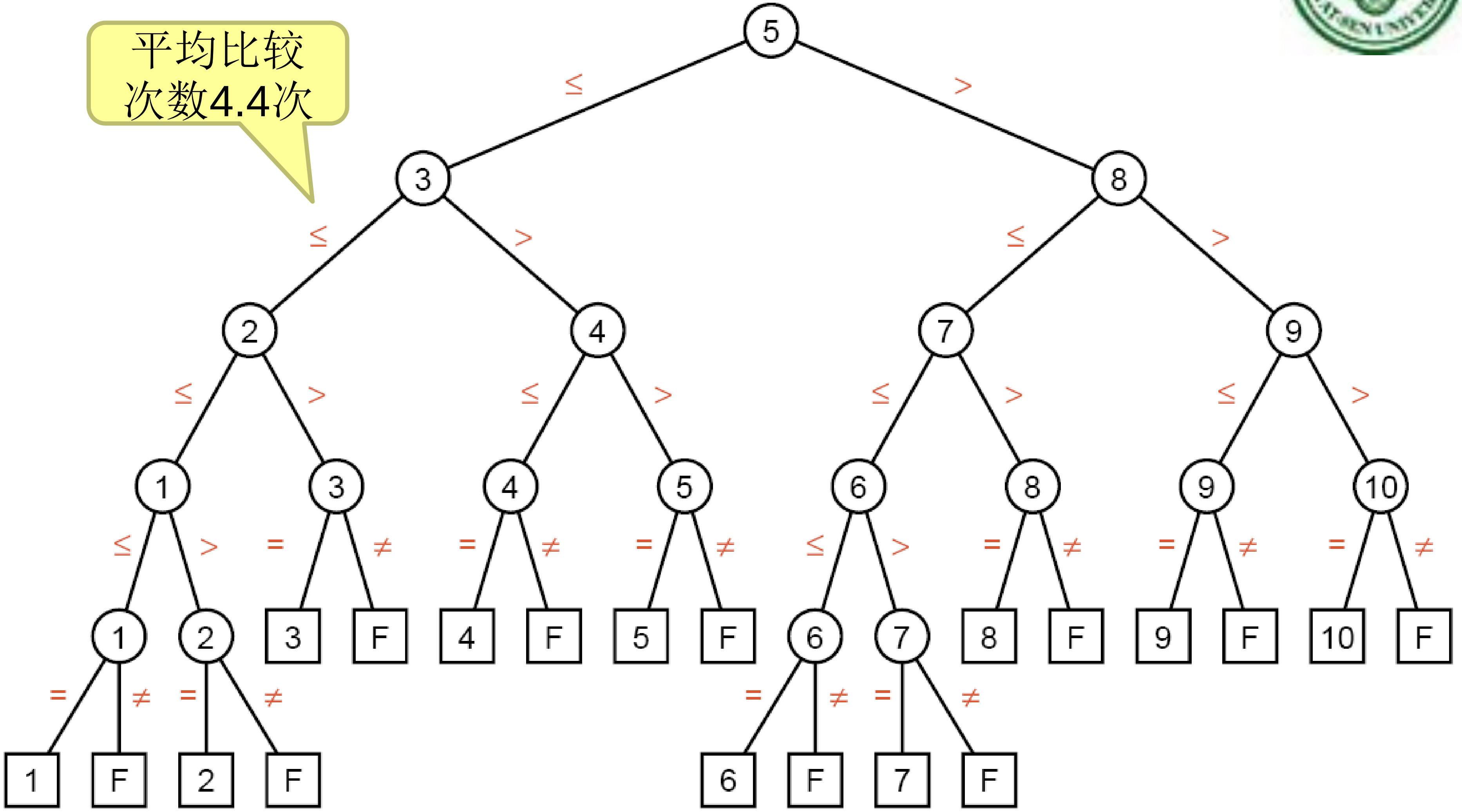


Figure 7.3. Comparison tree for binary_search_1, $n = 10$

7.4.1 Analysis for n = 10

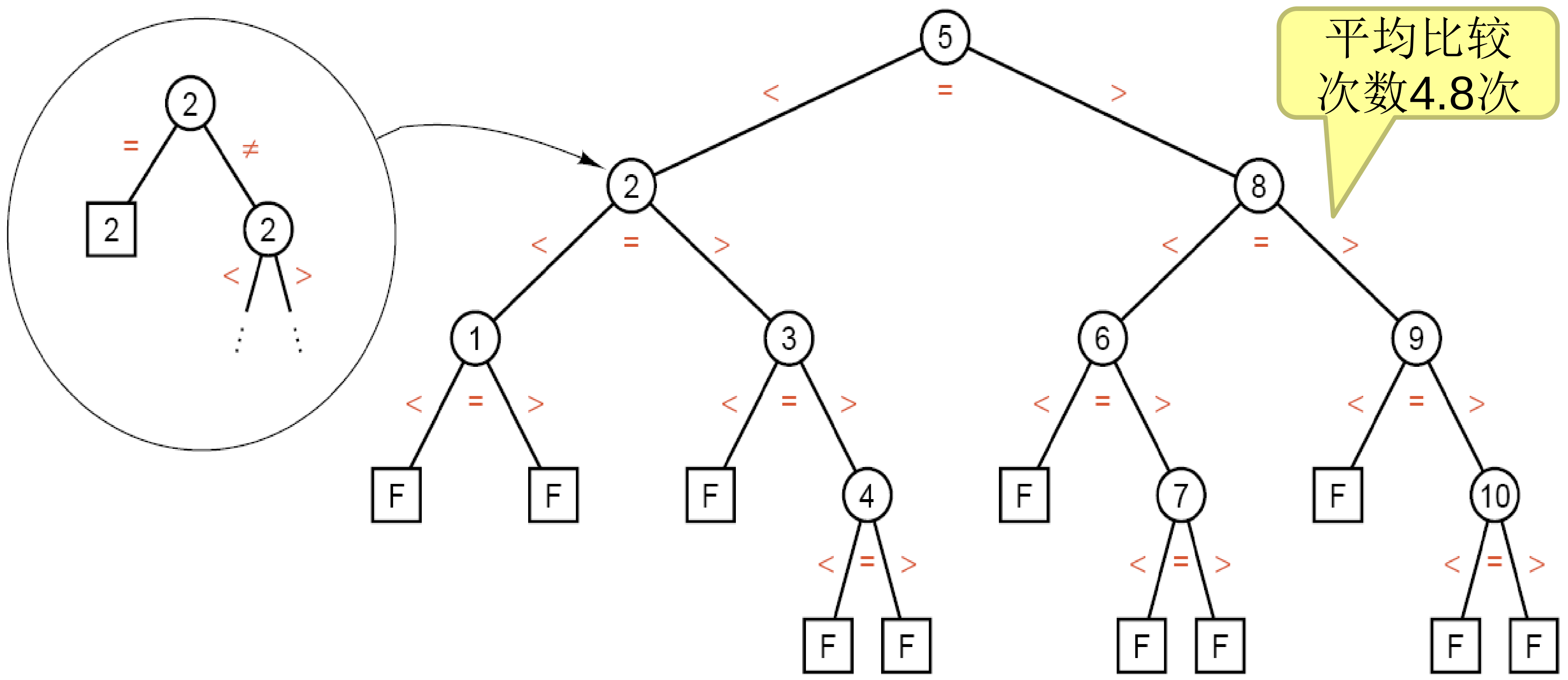
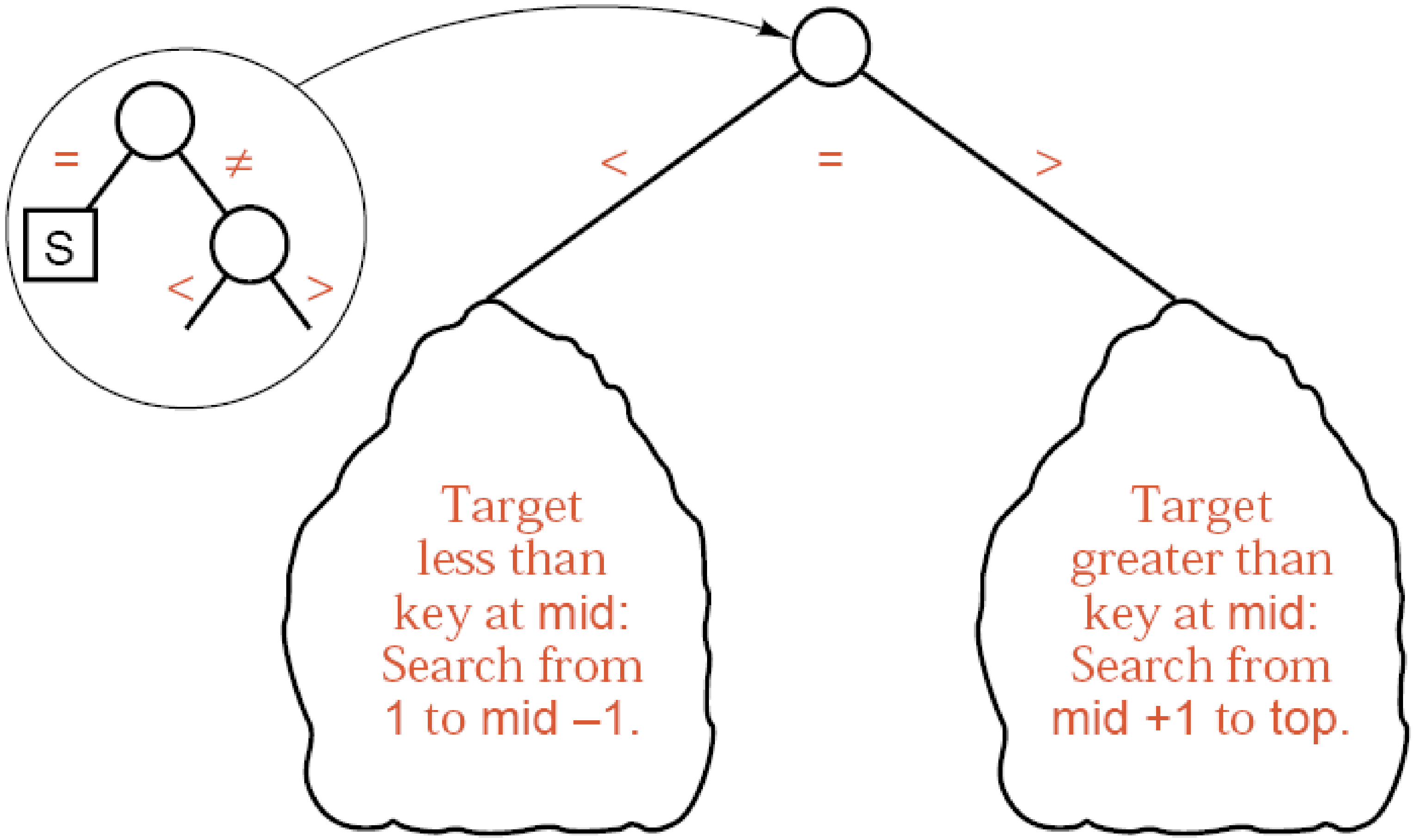


Figure 7.4. Comparison tree for binary_search_2, $n = 10$

7.4.1 Analysis for n = 10



=
≠
S
<
>
<
=
>
Target less than key at mid: Search from 1 to mid −1.
Target greater than key at mid: Search from mid +1 to top.

Figure 7.5. Top of the comparison tree, recursive binary_search_2

7.4.2 Generalization

1. 2-Trees

As ***2-tree*** is a tree in which every vertex except the leaves has exactly two children.

Lemma 7.1 The number of vertices on each level of a 2-tree is at most twice the number on the level immediately above. Hence, in a 2-tree, the number of vertices on level t is at most 2^t for $t \geq 0$.

Lemma 7.2 If a 2-tree has k vertices on level t, then $t \geq \lg k$, where lg denotes a logarithm with base 2.

since 2^t>k

7.4.2 Generalization

2. Analysis of binary_search_1

Some facts:

➢ The last step is a check with the target, hence successful and unsuccessful searches terminate at leaves, and there are $2n$ leaves.

➢ All the leaves are on the same level or on two adjacent levels.

➢ let t be the maximum level, then $t = \lceil \lg 2n \rceil$, this is because $2^t >= 2n$ and $2^{(t-1)} < 2n$ for 2-trees, hence

$$t >= \lg(2n) \quad \text{and } t < \lg(2n) + 1;$$

7.4.2 Generalization

2. Analysis of binary_search_1

- The maximum number of key comparisons is $t = \lceil \lg 2n \rceil$, approximately $\lg(n) + \lg(2) = \lg(n) + 1$.
- The average number of comparisons is about $\lg(n)$.

7.4.2 Generalization

3. Analysis of binary_search_2, Unsuccessful Search

- For binary_search_2, all the leaves correspond to unsuccessful searches, so thee are exactly n+1 leaves, corresponding to the n+1 unsuccessful outcomes.

- Since these leaves are all at the bottom of the tree, Lemma 7.1 implies that the number of leaves is approximately 2^h, where h is the height of the tree.

- Since in binary_search_2, two comparisons of keys are performed for each internal vertex,

The number of comparisons done in an unsuccessful search by binary_search_2 *is approximately* $2\lg(n+1)$.

7.4.2 Generalization

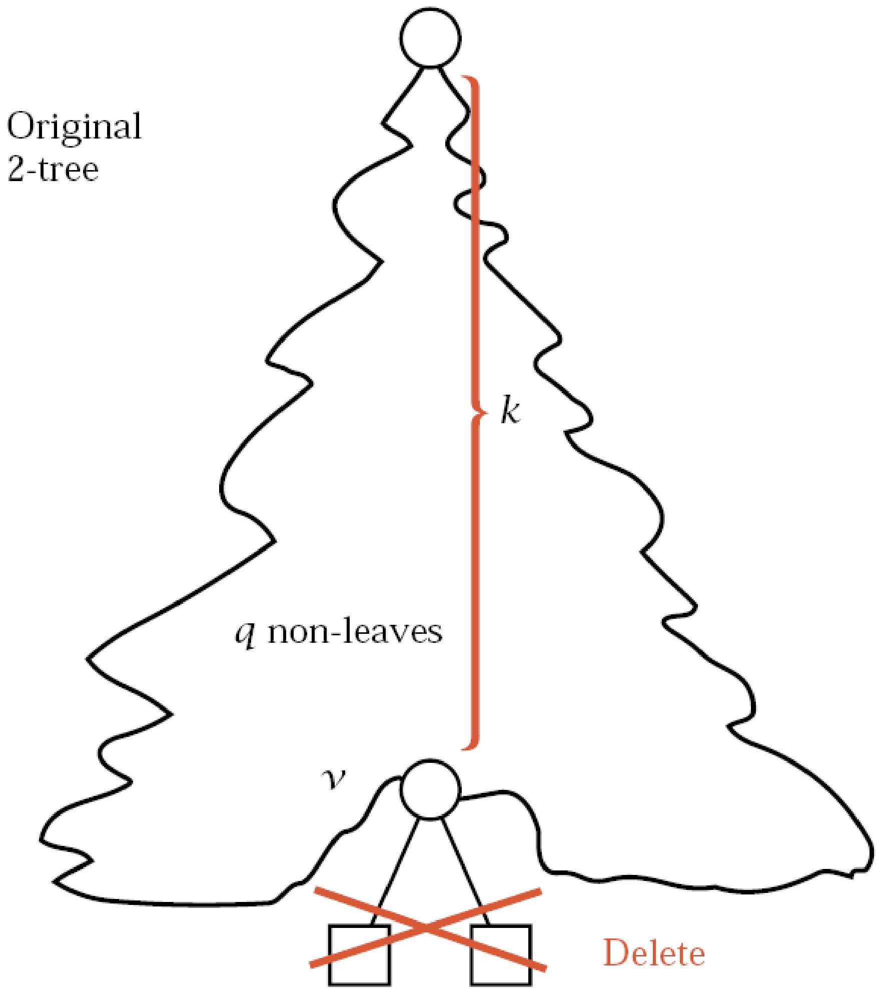
4. The Path-Length Theorem

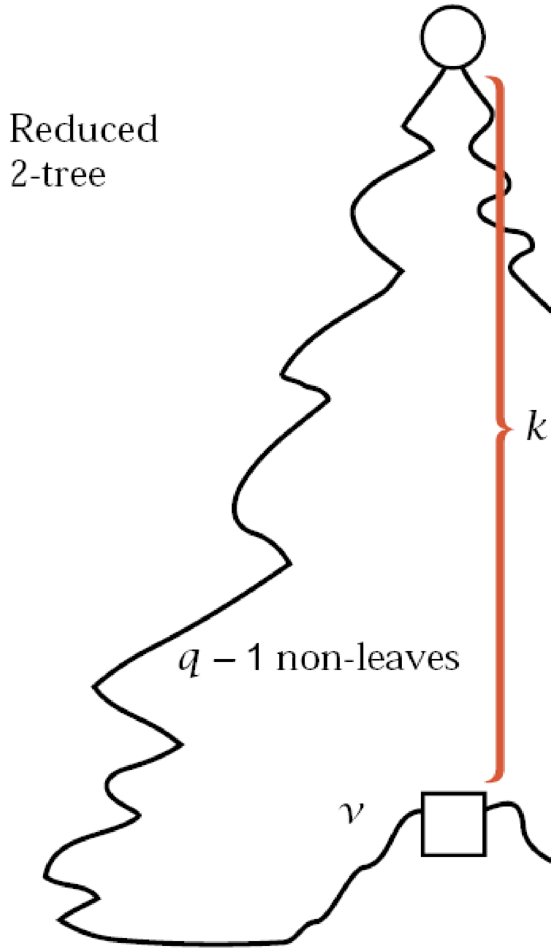
Denote the external path length of a 2-tree by E, the internal path length by I, and let q be the number of vertices that are not leaves. Then

$$E = I + 2q.$$

7.4.2 Generalization

4. The Path-Length Theorem





E-2(k+1)+k = E-k-2
将E = I +2q代入得，
E'=I+2q-k-2
= (I-k)+2(q-1)
=I'+2q',符合定义

7.4.2 Generalization

5. Analysis of binary_search_2, Successful Search

In the comparison tree of binary_search_2, the distance to the leaves is $\lg(n+1)$, as we have seen. The number of leaves is $n+1$, so the external path length is about

$$(n+1)\lg(n+1).$$

Theorem 7.3 then shows that the internal path length is about

$$(n+1)\lg(n+1)-2n.$$

To obtain the average number of comparisons done in a successful search, we must first divide by n (the number of non-leaves) and then add 1 and double, since two comparisons were done at each internal node. Finally, we subtract 1, since only one comparison is done at the node where the target is found. The result is:

In a successful search of a list of n entries, binary_search_2 *does approximately*

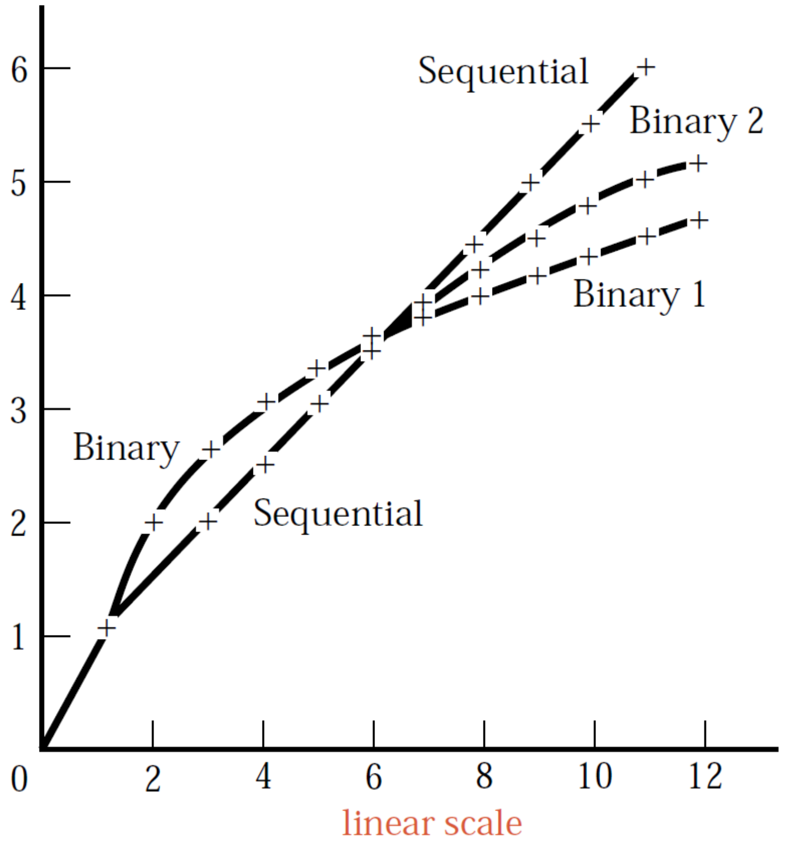
$$\frac{2(n+1)}{n}\lg(n+1)-3$$

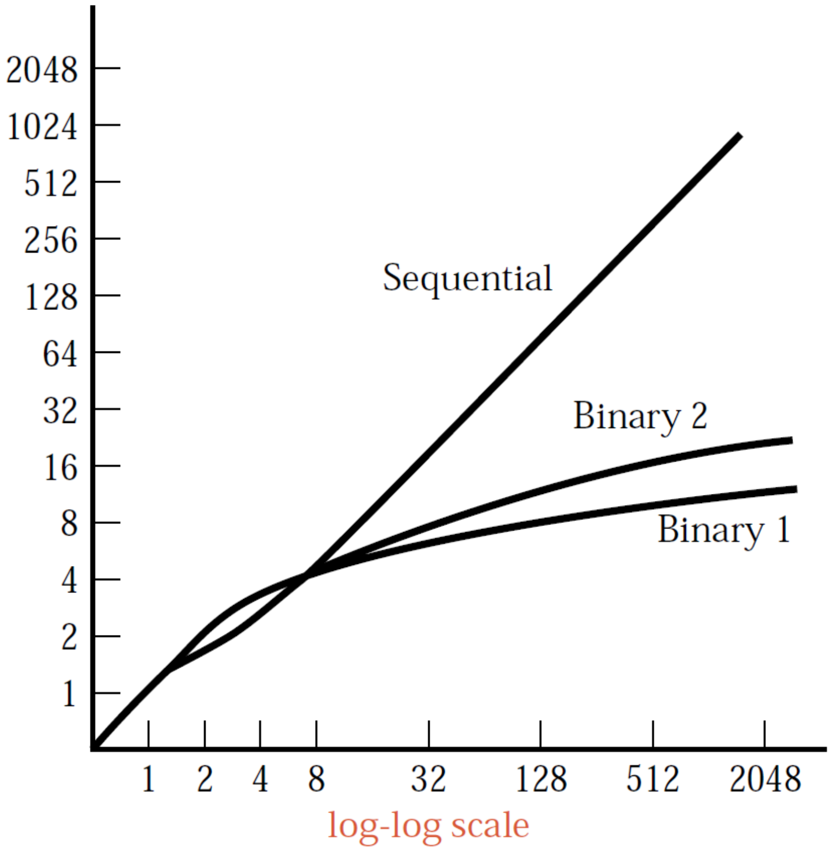
comparisons of keys.

7.4.3 Comparison of Methods

	Successful search	*Unsuccessful search*
binary_search_1	$\lg n + 1$	$\lg n + 1$
binary_search_2	$2 \lg n - 3$	$2 \lg n$

7.4.3 Comparison of Methods





7.4.3 Comparison of Methods

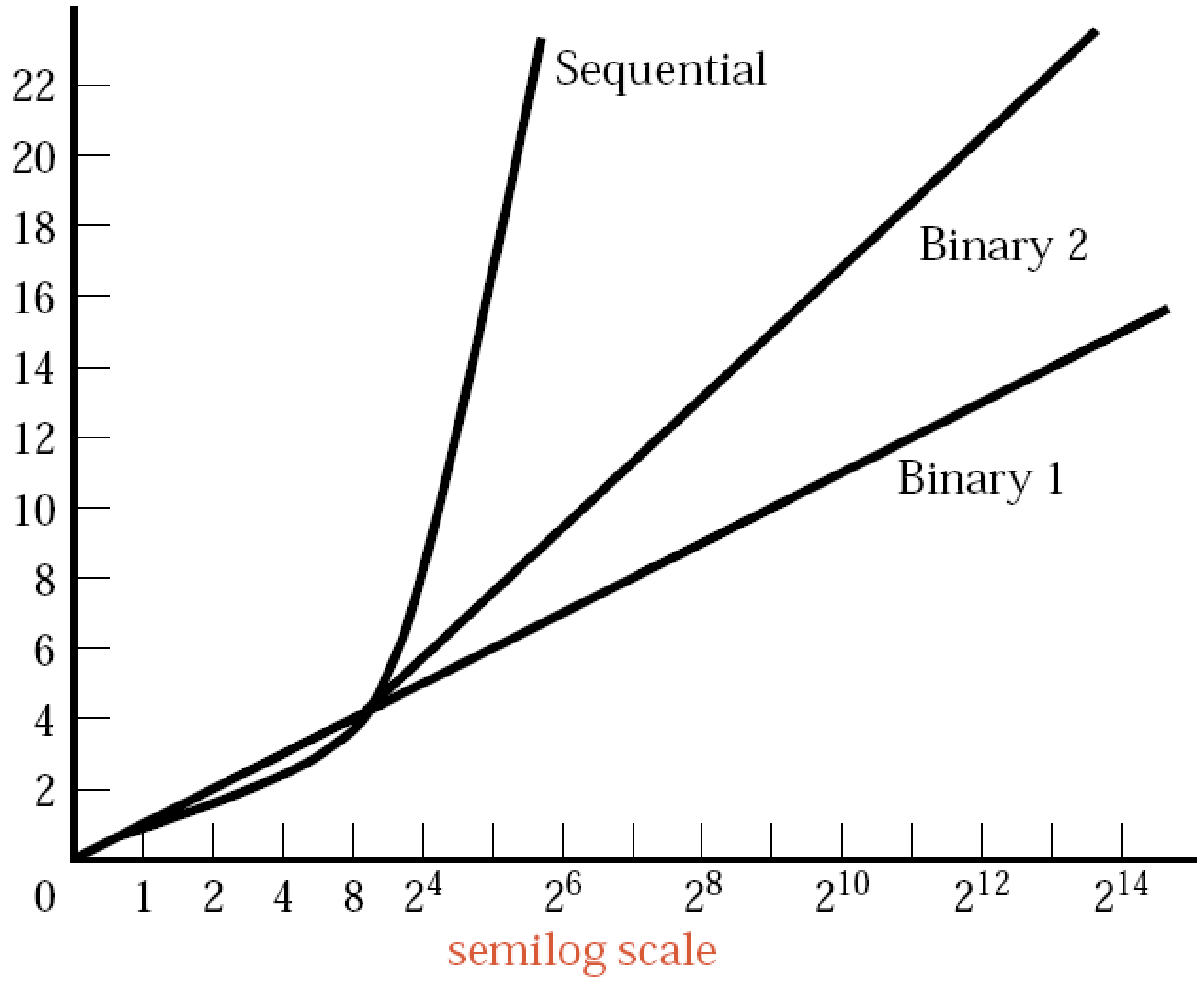


Figure 7.7. Numbers of comparisons for average successful searches

7.4.4 General Relationship

binary_search_2

we know that the average number of comparisons in a successful search is

$$S = 2\left(\frac{I}{n} + 1\right) - 1 = \frac{2I}{n} + 1$$

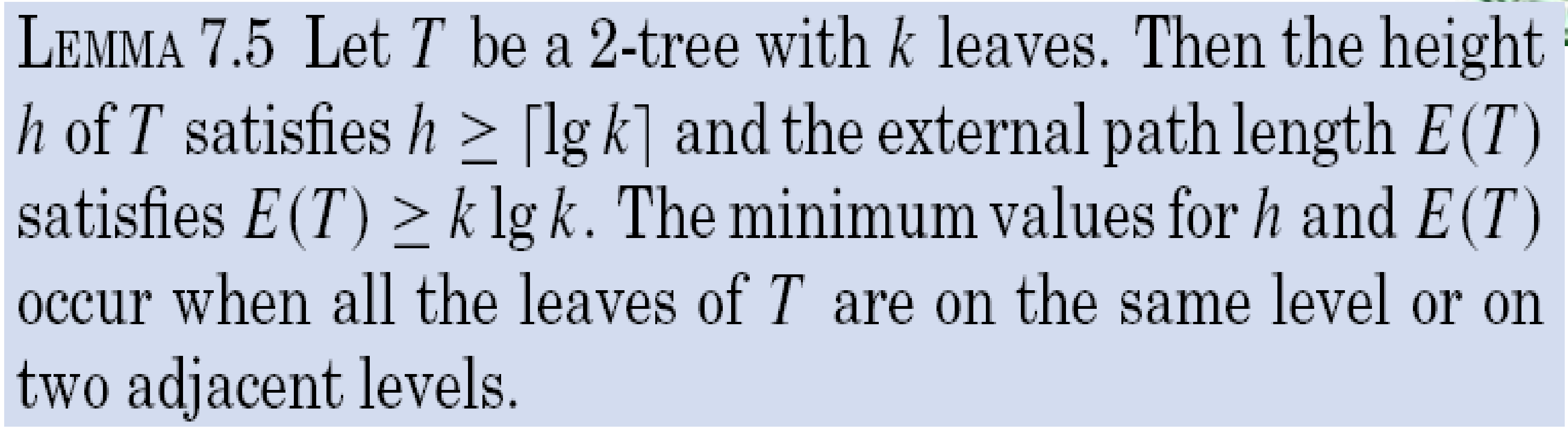
and the average number for an unsuccessful search is $U = 2E/(n+1)$. By Theorem 7.3, $E = I + 2n$. Combining these expressions, we can therefore conclude that

Theorem 7.4

Under the specified conditions, the average numbers of key comparisons done in successful and unsuccessful searches are related by

$$S = \left(1 + \frac{1}{n}\right)U - 3.$$

7.5 Lower Bounds


Lemma 7.5 Let T be a 2-tree with k leaves. Then the height h of T satisfies $h \geq \lceil \lg k \rceil$ and the external path length $E(T)$ satisfies $E(T) \geq k \lg k$. The minimum values for h and $E(T)$ occur when all the leaves of T are on the same level or on two adjacent levels.

1) 如果所有的叶子节点均位于h层，显然有 $k \leq 2^h$；如果有部分叶节点位于h-1层，那么每一个h-1层上的节点对应将h层上的节点减2，显然$k \leq 2^h$依然保持，固有$h \geq \lg k$.

2) 假定有x个叶节点在h-1层，那么有k-x个叶节点在h层，这k-x个叶节点对应1/2(k-x)个h-1层的节点，显然在h-1层上满足$1/2(k-x)+x \leq 2^{h-1}$, 可得到$x \leq 2^h - k$.

又已知

$$E(T) = (h-1)x+h(k-x) = kh-x \ \geq kh-(2^h - k) = k(h+1)- 2^h$$

又已知$2^{h-1} \leq k \leq 2^h$，可假定$h = \lg k+q$ $(0\leq q \leq 1)$,代入上式可得到

$$E(T) \geq k(\lg k+1+q- 2^q)$$

当$0\leq q \leq 1$时， $0\leq 1+q- 2^q \leq 0.0816$，故得证。

7.5 Lower Bounds

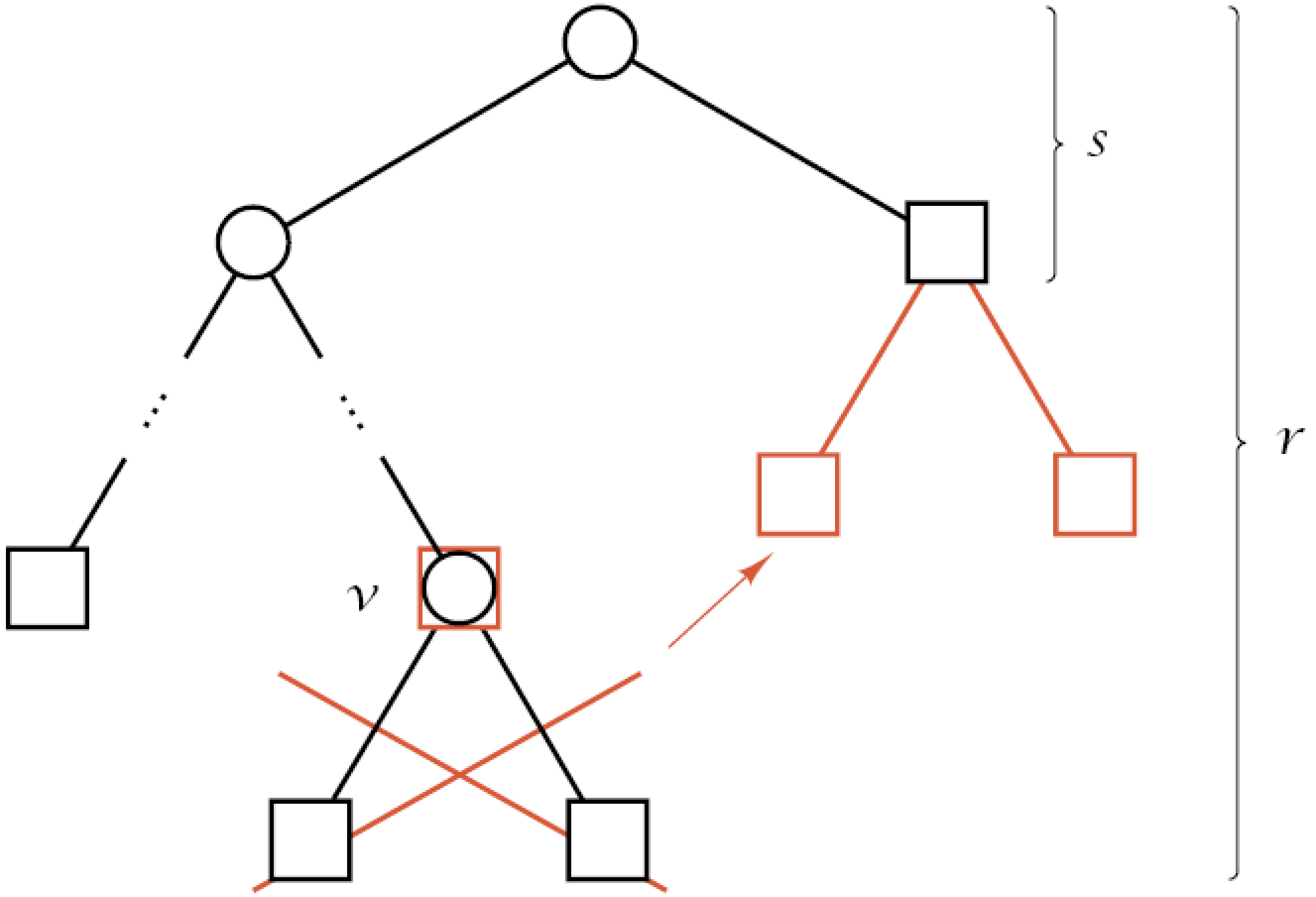


Figure 7.8. Moving leaves higher in a 2-tree

$$E(T') = E(T) - 2r + (r-1) - s + 2(s+1) = E(T) - r + s + 1 < E(T)$$

7.5 Lower Bounds

Theorem 7.6

Suppose that an algorithm uses comparisons of keys to search for a target in a list. If there are k possible outcomes, then the algorithm must make at least $\lceil \lg k \rceil$ comparisons of keys in its worst case and at least $\lg k$ in its average case.

Corollary 7.7

binary_search_1 *is optimal in the class of all algorithms that search an ordered list by making comparisons of keys. In both the average and worst cases,* binary_search_1 *achieves the optimal bound.*

7.6 Asymptotics

$g(n) = 1$	Constant function
$g(n) = \log n$	Logarithmic function
$g(n) = n$	Linear function
$g(n) = n^2$	Quadratic function
$g(n) = n^3$	Cubic function
$g(n) = 2^n$	Exponential function

7.6.2 Orders of Magnitude

1. Definitions

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then:

$f(n)$ has ***strictly smaller order of magnitude*** than $g(n)$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is finite and nonzero then:

$f(n)$ has ***the same order of magnitude*** as $g(n)$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then:

$f(n)$ has ***strictly greater order of magnitude*** than $g(n)$.

7.6.2 Orders of Magnitude

2. Assumptions

- We assume that $f(n) > 0$ and $g(n) > 0$ for all sufficiently large n.
- We assume that $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists.

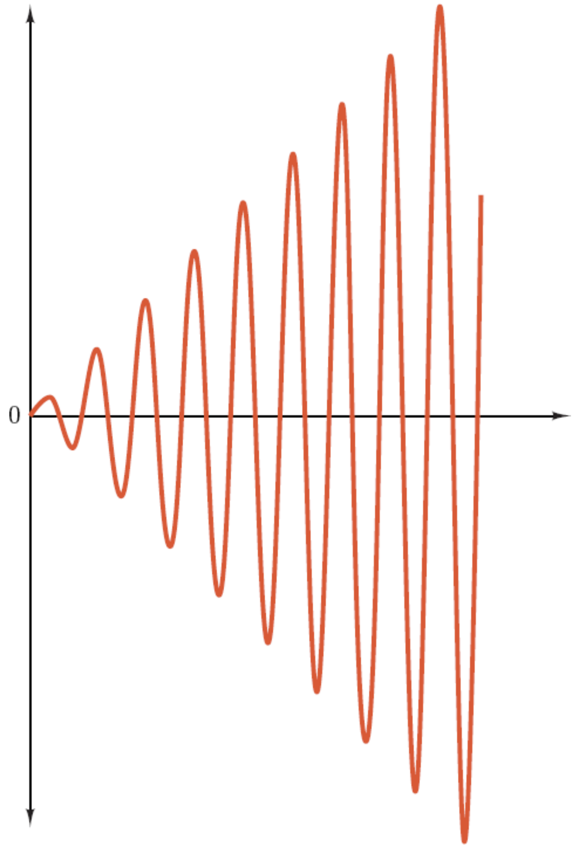


Figure 7.9. Graph of $x \sin x$

7.6.2 Orders of Magnitude

3. Polynomials

polynomials

If $f(n)$ is any polynomial in n with degree r, then $f(n)$ has the same order of magnitude as n^r.

powers of n

If $r < s$, then n^r has strictly smaller order of magnitude than n^s.

7.6.2 Orders of Magnitude

4. Logarithms and L'Hopital's Rule

The order of magnitude of a logarithm does not depend on the base for the logarithms.

To see why this is true, let $\log_a n$ and $\log_b n$ be logarithms to two different bases $a > 1$ and $b > 1$. As observed in Section A.2.6, $\log_b n = (\log_b a)(\log_a n)$. Hence,

$$\lim_{n\to\infty} \frac{\log_b n}{\log_a n} = \lim_{n\to\infty} \frac{(\log_b a)(\log_a n)}{\log_a n} = \log_b a,$$

which is a nonzero constant, so $\log_b n$ has the same order of magnitude as $\log_a a$, which was to be shown.

7.6.2 Orders of Magnitude

4. Logarithms and L'Hopital's Rule

L'Hôpital's Rule *Suppose that:*

- $f(x)$ *and* $g(x)$ *are differentiable functions for all sufficiently large* x, *with derivatives* $f'(x)$ *and* $g'(x)$, *respectively.*
- $\lim_{x\to\infty} f(x) = \infty$ *and* $\lim_{x\to\infty} g(x) = \infty$.
- $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ *exists.*

Then $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ *exists and* $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$.

When we apply L'Hôpital's Rule to $f(x) = \ln x$ and $g(x) = x^r$, $r > 0$, we have $f'(x) = 1/x$ and $g'(x) = rx^{r-1}$, and hence

$$\lim_{x\to\infty} \frac{\ln x}{x^r} = \lim_{x\to\infty} \frac{1/x}{rx^{r-1}} = \lim_{x\to\infty} \frac{1}{rx^r} = 0$$

since $r > 0$. Since the base for logarithms doesn't matter, we have:

$\log n$ *has strictly smaller order of magnitude than any positive power* n^r *of* n, $r > 0$.

7.6.2 Orders of Magnitude

5. Exponential Functions

Specifically, let $f(x) = a^x$, where $a > 1$ is a real number, and let $g(x) = x^r$, where r is a positive integer.

微分后得到

$$f'(x) = \frac{d}{dx}a^x = \frac{d}{dx}\left(e^{\ln a}\right)^x = \frac{d}{dx}e^{(\ln a)x} = (\ln a)e^{(\ln a)x} = (\ln a)a^x$$

$$g'(x) = rx^{r-1}.$$

继续运用L'Hopital's

$$\lim_{x\to\infty}\frac{f(x)}{g(x)} = \lim_{x\to\infty}\frac{f'(x)}{g'(x)} = \cdots = \lim_{x\to\infty}\frac{f^{(r)}(x)}{g^{(r)}(x)} = \lim_{x\to\infty}\frac{(\ln a)^r a^x}{r!} = \infty.$$

Any exponential function a^n for any real number $a > 1$ has strictly greater order of magnitude than any power n^r of n, for any positive integer r.

7.6.2 Orders of Magnitude

6. Common Orders

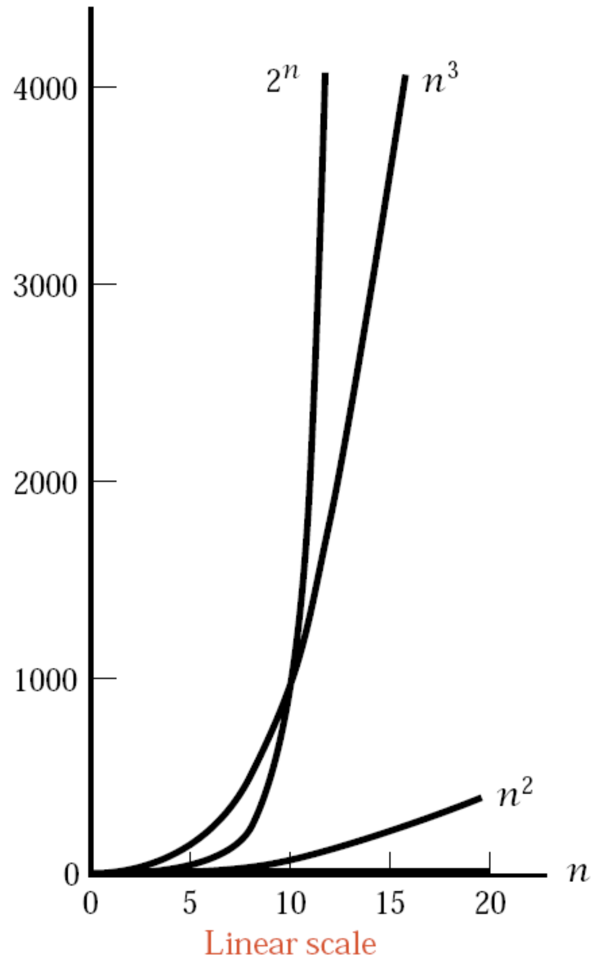
If $h(n)$ is any function for which $h(n) > 0$ for all sufficiently large n, then the order of magnitude of $f(n)h(n)$ is related to the order of $g(n)h(n)$ in the same way (less than, equal to, or greater than) as the order of $f(n)$ is related to the order of $g(n)$.

The proof of this is simply the observation that

$$\lim_{n\to\infty} \frac{f(x)h(x)}{g(x)h(x)} = \lim_{n\to\infty} \frac{f(x)}{g(x)}.$$

7.6.2 Orders of Magnitude

6. Common Orders



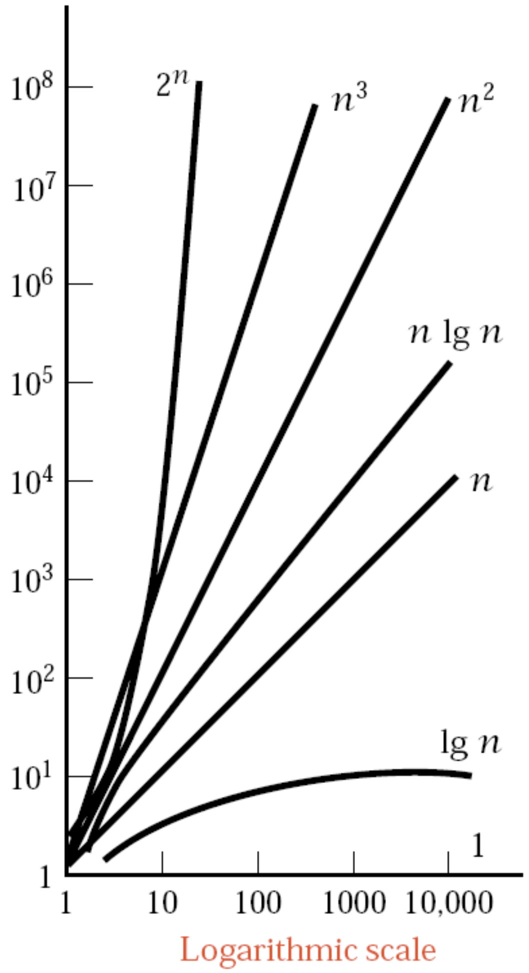


Figure 7.10. Growth rates of common functions

7.6.3 The Big-O and Related Notations

Notation: $f(n)$ is	Pronounce: $f(n)$ is	Meaning: Order of f compared to g is		Value of $\lim_{n\to\infty}(f(n)/g(n))$
$o(g(n))$	*little oh of* $g(n)$	$<$	*strictly smaller*	0
$O(g(n))$	*Big Oh of* $g(n)$	$\leq$	*smaller or equal*	*finite*
$\Theta(g(n))$	*Big Theta of* $g(n)$	$=$	*equal*	*nonzero finite*
$\Omega(g(n))$	*Big Omega of* $g(n)$	$\geq$	*larger or equal*	*nonzero*

- On a list of length n, sequential search has running time $\Theta(n)$.
- On an ordered list of length n, binary search has running time $\Theta(\log n)$.
- Retrieval from a contiguous list of length n has running time $O(1)$.
- Retrieval from a linked list of length n has running time $O(n)$.
- Any algorithm that uses comparisons of keys to search a list of length n must make $\Omega(\log n)$ comparisons of keys (Theorem 7.6).
- Any algorithm for the Towers of Hanoi (see Section 5.1.4) requires time $\Omega(2^n)$ in order to move n disks.

7.6.4 Keeping the Dominant Term

$$f(n) = g(n) + O(h(n))$$

- For a successful search in a list of length n, sequential search has running time $\frac{1}{2}n + O(1)$.
- For a successful search in an ordered list of length n, binary search has running time $2\lg n + O(1)$.
- Retrieval from a contiguous list of length n has running time $O(1)$.
- Retrieval from a simply linked list of length n has average running time $\frac{1}{2}n + O(1)$.

$$n^2 + 4n - 5 = n^2 + O(n) \text{ and } n^2 - 9n + 7 = n^2 + O(n)$$