CS416

Introduction to Computer Science II

Spring 2018

7 Introduction to ADTs and Data Structures (part 3)

Chapter 14

Dictionary Search Performance

- Given a *key*, how expensive is it to find the record containing that key?
  - if records stored in an unordered list or unordered array?
  - if records stored in a sorted list?
  - if records stored in a sorted array?

Searching an Unordered Array or List

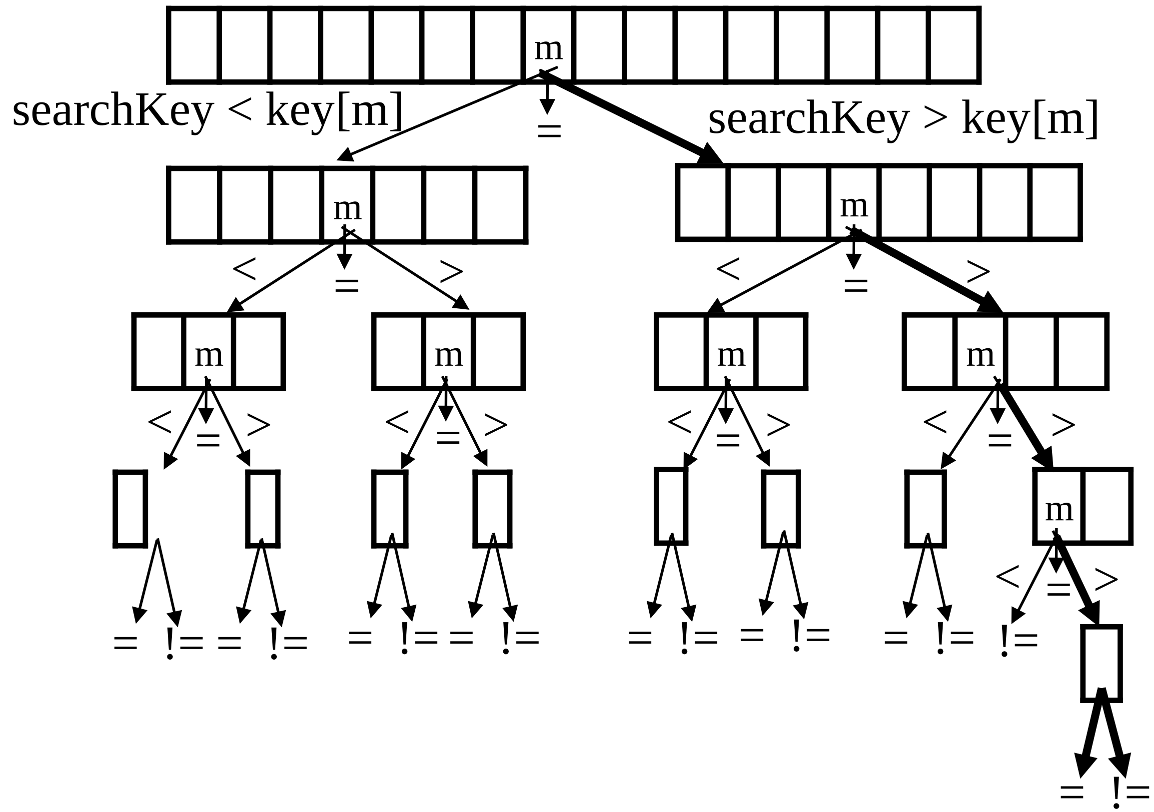
- Must test every entry until key matches or get to end
  - if there are *n* records in the list or array
    - assuming random order and random key distribution, *on average* must compare key to *n/2* entries before finding the right one; if the key is not in the data collection, you have to test all *n*
    - with 1000 records you can expect to compare "on the order of" 1000 keys to find the desired record, or determine it is not there
    - this is called an order *n* algorithm, *O(n)*
      - *order* is short for *order of magnitude*
      - an algorithm is *O(n)* if its time is related to *kn* for any constant *k*
        - So, *n/2, 2n, n* comparisons all result in an *O(n)* algorithm
        - the constant *k* can be important, but less so far larger *n*

Searching a Sorted List

- Must test every entry until key matches or pass where the key should be on the list
  - if there are *n* records in the list or array
    - assuming random order and random key distribution, *on average* you have to compare the key to 1/2 of the entries before finding the right one -- that's *n/2* comparisons
    - if the key is <u>not</u> in the data collection, *on average* you'll also have to make *n/2* comparisons to find where it should have been.
    - this is still an order *n* algorithm, *O(n)*

Searching a Sorted Array

- Use a *binary search algorithm* (Lab 8)
- Let's use a 16 entry array
- Each test eliminates about ½ the array
- One final test for equality
- What's the <u>longest</u> path to an answer?
  - 5
- What is algorithm complexity?



Binary Search Complexity

- Analysis for *n* where *n* is a power of 2
  - 1st test eliminates *1/2* the array (*n/2* elements)
  - 2nd test eliminates *1/2* the remaining (*n/4* elements)
  - *k*-th test eliminates $n/2^k$ elements
  - We've tested the entire array when

    $$n = n/2 + n/4 + ... + n/(2^k) \text{ for some } k$$

    k is easy to find since the last step leaves 1 element; so

    $$1 = n/2^k\text{, i.e., when } n = 2^k\text{, i.e., when } k = log_2 n$$
  - So, max number tests is $log_2 n + 1$, but we ignore "+1"
    - algorithm is $O(\ log_2 n\ )$ or just $O(\ log\ n\ )$
    - for *n* around 8,000 *k* is just 13; for 16,000, it's 14, etc.
- For other *n*, order of magnitude is same as for the smallest power of 2 that is greater than *n*.

Key as Array Index

- If the record's *key* is a unique (small) positive integer, can use it as an array index for storing the record.
- Record "search" is just an array entry access:
  - it takes just 1 operation to access the desired record
  - this is O(1) or *constant time* algorithm (the time to access the element does not depend on the total number of elements in the data structure)
- This is ideal performance, but its not realistic
  - "real" keys are strings, or large integers, or other even more complex objects

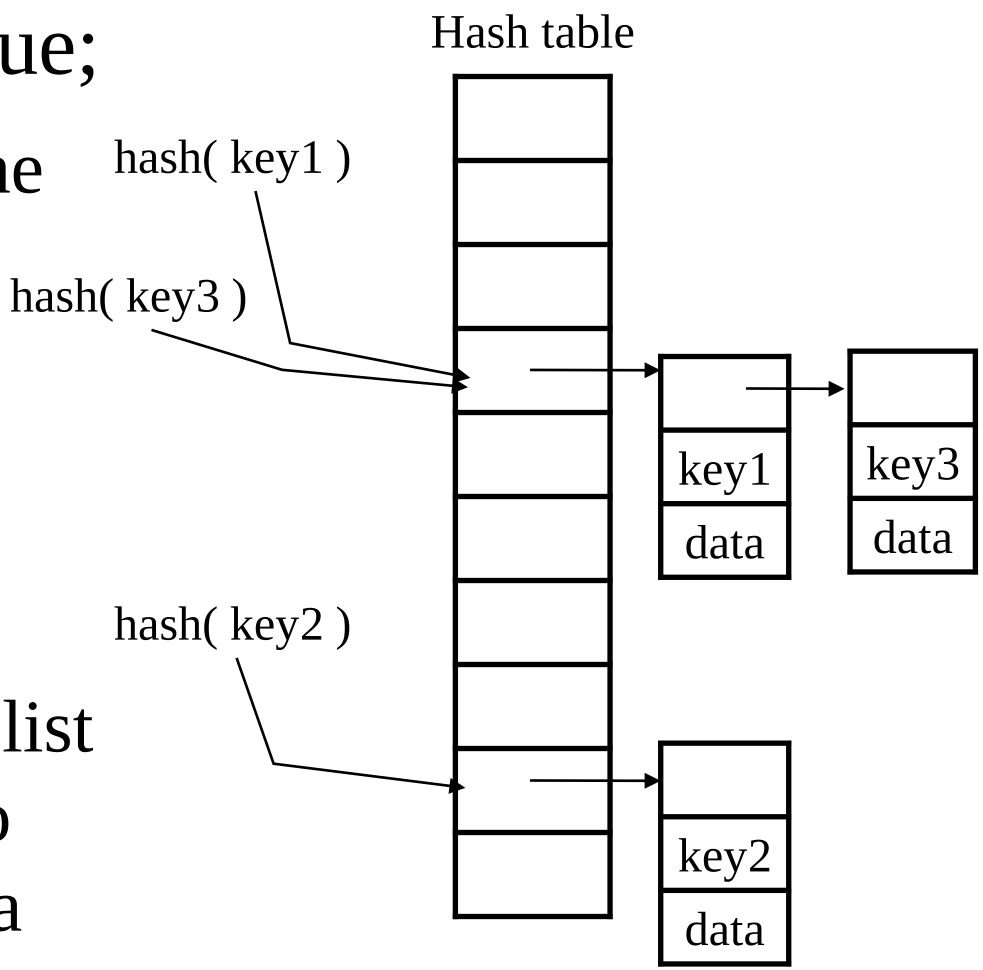
Hashing

- *Hashing* transforms a “real” key into a small positive integer, with the goal of getting an O(1) search algorithm
- For example, a simple *hash function* for a String could be:
  - sum up the internal *int* values of all the characters in the String and compute the *remainder* from dividing by the size of the array used to store the data

```
public int hash( String s )
{
  int sum = 0;
  for (int i=0; i<s.length; i++)
  {
     sum += s.charAt( i );
  }
  return sum % arrayLength;
}
```

Hash Collisions

- Except in very rare cases, *hash* function values are <u>not</u> unique;
  - many different keys map to the same hash table index
  - these are called *collisions*
- Simplest collision-handling strategy:
  - Each array entry references a list of records whose keys hash to that array index; often called a *bucket* or *bucket list*



Hash Function Issues

- Hash functions should *distribute* hash values reasonably uniformly over the array indexes
  - Best to separate computation of a hash value from the mapping of that value to a particular array size. All Java objects have *int hashCode()* method that does the former
  - We could use the first letter of the word to index into an array of lists; this is a primitive (and poor) hash function: 1st characters of words aren't evenly distributed
  - The sum of character values isn't very good either: any words with the same letters hash to the same index
  - A more reasonable algorithm:
    - int hashValue = sum(charAt(i) * 10^i))
    - this generates integer overflow, but we just ignore lost digits

Hash Table Search

- Complexity of hash table search using *bucket* lists
  - The number of tests is on the order of the average list size
  - With good distribution of *n* records over *k* array entries, complexity is *O(n/k)*, which is still *O(n)*
    - We didn't get to *O(1)* but we can keep the "constants" low
  - We can also increase *k* as *n* increases, E.g., let k = n / 100
    - Now it's *O(n/k) = O(n/(n/100)) = O(100) = O(1)*
    - Of course, there is a limit to both *n* and *k* in terms of the memory available to the program.
    - At some point, need to come up with strategies for searching data stored on disk -- *database technology*

Java Hash Tables

- *java.util.HashSet<T>*
  - *add(T)*: based on the *Object* method *hashCode*
  - *T get(T)*: retrieves object if it is in the *HashSet*
- *java.util.HashMap<K,V>*
  - *put(K key, V value)*: the *hashCode* for *key* is used to store the *value* object
  - *V get (K key)*: retrieves *V* object based on *ke*y
- *java.util.Hashtable<K,V>*
  - *synchronized* version of *HashMap*
  - similar to *Vector/ArrayList* relationship

Iterators

- Java *Collection* classes implement the *Iterable* interface
  - which means they can generate *Iterators*
  - An *Iterator* provides common access to elements of *Collections* <u>without exposing internal structure</u>

```
public interface Iterable<T>
{
  public  Iterator<T> iterator();
}
```

```
public interface Iterator<T>
{
  public boolean hasNext();
  public T       next();

  public void    remove();
                 // removes last T
                 // referenced by next
                 // from Collection!
}
```

Using Iterators

```
Iterator<String>  iter
```

```
ArrayList<String> sa ...
iter = sa.iterator();
```

```
Vector<String> sa ...
iter = sa.iterator();
```

```
LinkedList<String> sa ..
iter = sa.iterator();
```

- With *while*

```
while ( iter.hasNext() )
{
   String str = iter.next();
   System.out.println( str );// do something with str
}
```

- Alternatively, use the new "for each" feature without an explicit iterator (it uses the *for* keyword)

```
for ( String str: sa )  // for each String, str, in sa
{
   System.out.println( str );// do something with str
}
```

Creating an Iterator

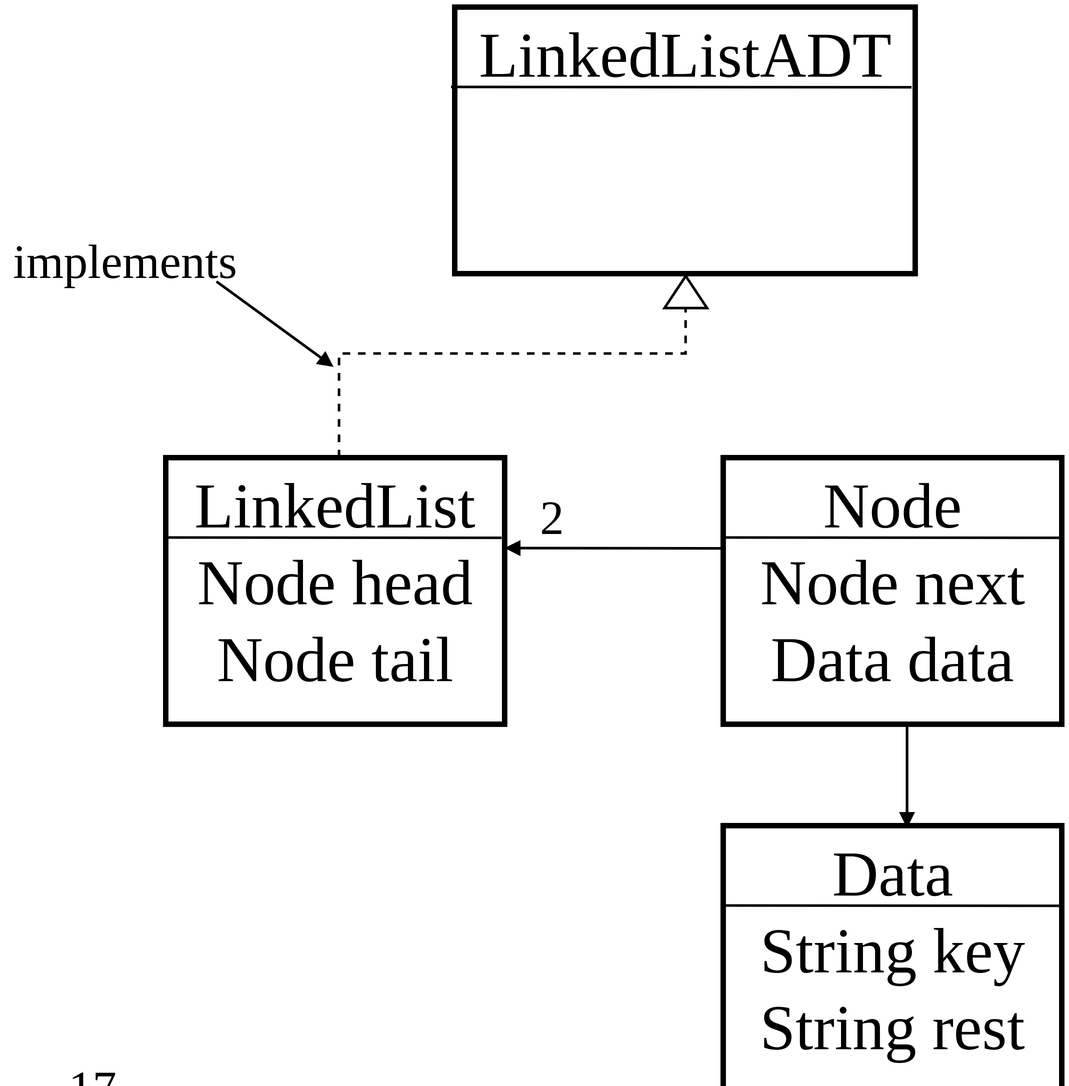
- Usually an *Iterator* is an inner class of the *Collection* object.
  - It has access to private information in the *Collection*
  - It has its own copies of position information, so multiple different iterations can be taking place over the same *Collection* at the same time
    - Unlike the *first(), next()* iteration methods of our *LinkedList*
  - It cannot guarantee correctness if elements are deleted or added during its execution -- except for its own *remove()*

Iterator for LinkedList

```
public class MyListIterator<T>
{
   public Node<T> cur = null;
   public MyListIterator()
   {
      cur = _head;
   }
   public boolean hasNext()
   {
      return cur != null;
   }
   public T next()
   {
      // throw exception if cur is null
      T temp = cur.data;
      cur = cur.next;
      return temp;
   }
 }
```

LinkedList Related Class Diagram

- UML Class diagram
  - Shows *static* relationships between classes
  - Does not show how objects of the class relate during execution
    - No insight re complexity of the object interactions
- UML Instance Diagram!



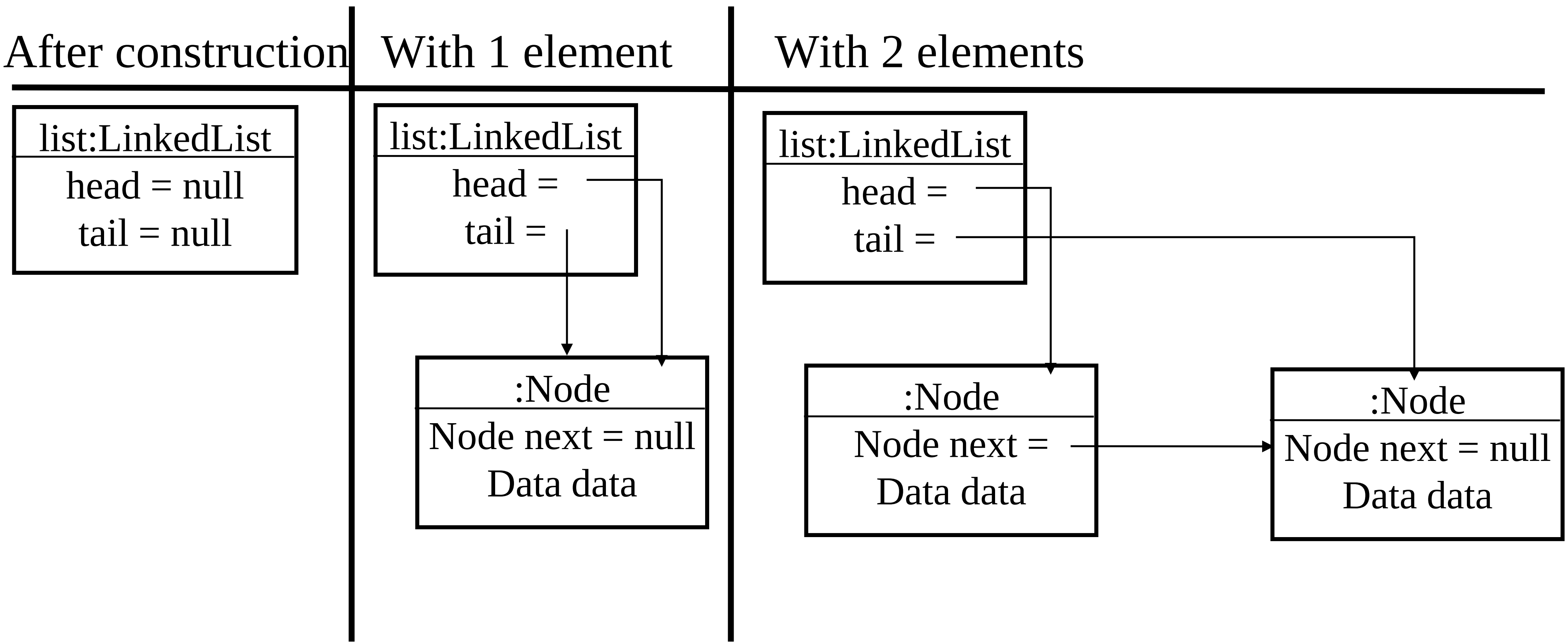
Instance Diagram

- UML <u>*instance diagrams*</u> show the status of a set of objects at a point in time during execution
- Name field is *objectName:ClassName*
  - *objectName* is optional and often missing
- Attribute fields
  - *Type name = value*
    - *value* field is the actual value at the point in time, if relevant
- We’ve used simplified versions of these diagrams all along

:Data
String key = “able” String rest = “?”~

Instance Diagram Example

- A particular example shows only those objects “of interest” for goals of the particular diagram



Review

- Abstract Data Type: *specification*
- Concrete data structures: *implementation*
- Arrays, stacks, queues, linked lists
- Recursion
- Linked lists: 1-way, 2-way, rings, 2-way rings
- Implementation/performance issues
- Hash tables

Next, in 416

- Trees
- More recursion