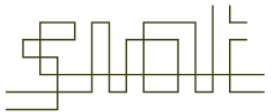


# IAT 265

## Java Data Structure and Sorting Algorithm



SCHOOL OF INTERACTIVE  
ARTS + TECHNOLOGY

# Today's Topics

- Overview of Data structures
- Data Structures and runtime reduction
  - Binary Tree & HashTable
- Search
  - Binary search
- Sorting
  - Quick sort

# Data Structures

- **Data structure** is a particular way of storing and organizing data in computer so that it can be used efficiently
  - Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address
- Common data structures include: **array, linked list, B-tree, hash-table, heap, ...**

# Operations on Data Structure

- With a collection of data, we often want to do many things
  - Store and organize data
  - Go through the list of data and do **X** per item
  - **Add** new data
  - **Delete** unwanted data
  - **Search** for data of some value
    - “Give me Smith’s phone number”

# Arrays

0	1	2	3	4	5	6
---	---	---	---	---	---	---

- Store data

```
arr[3] = 33
```

- Sort data

- Bubble sort

- Iterate (*go thru*):

```
for( i = 0 ; i < 10 ; i++)  
    sum = sum + arr[i] ;
```

- Search for 42

```
for( i = 0 ; i < last ; i++ )  
    if( arr[i] == 42 )  
        SUCCESS ;
```

# Using Java ArrayList

Declare:

```
ArrayList aList = new ArrayList(6);
```

- ArrayList allows you to *add* or *delete*:

```
SomeClass newItem, existingItem ;
```

```
aList.add( newItem );
```

```
aList.remove( existingItem );
```

# Runtime

- Often, we care about the time that computation takes
- It's ok if unimportant things run slow
- **Important** stuff needs to go fast
  - Stuff that gets run all the time

# What's Important

- Runtime is important
- If you're only going to do it once or twice
  - Use any old algorithm that will work
- If you're going to run it millions of times
  - Think about the algorithm!

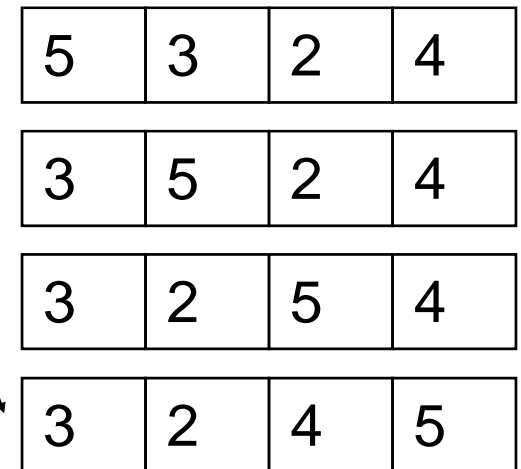


# E.g. Bubble sort

## ■ Code is small:

```
for (int i=arr.length-1; i>0; i--) {  
    for (int j=0; j<i; j++) {  
        if (arr[j] > arr[j+1]) {  
            temp = arr[j];  
            arr[j] = arr[j+1];  
            arr[j+1] = temp;  
        }  
    }  
}
```

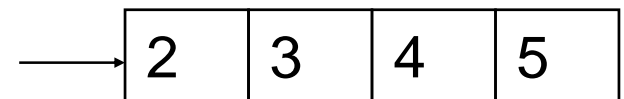
end of one inner loop



5	3	2	4
3	5	2	4
3	2	5	4
3	2	4	5

5 'bubbled' to the correct position

Nested loops put remaining elements in place



2	3	4	5
---	---	---	---

# Runtime of Bubble Sort

- Suppose we have  $n$  values in the array, then with the nested loops:
  - For **each iteration** of outer loop, inner loop run  $n$  times
  - The total runtime:  $n*n \rightarrow O(n^2)$
  - **This is very slow!!**
- Searching, insertion & deletion need to consider how to reduce runtime as well

# Data Structures for runtime reduction

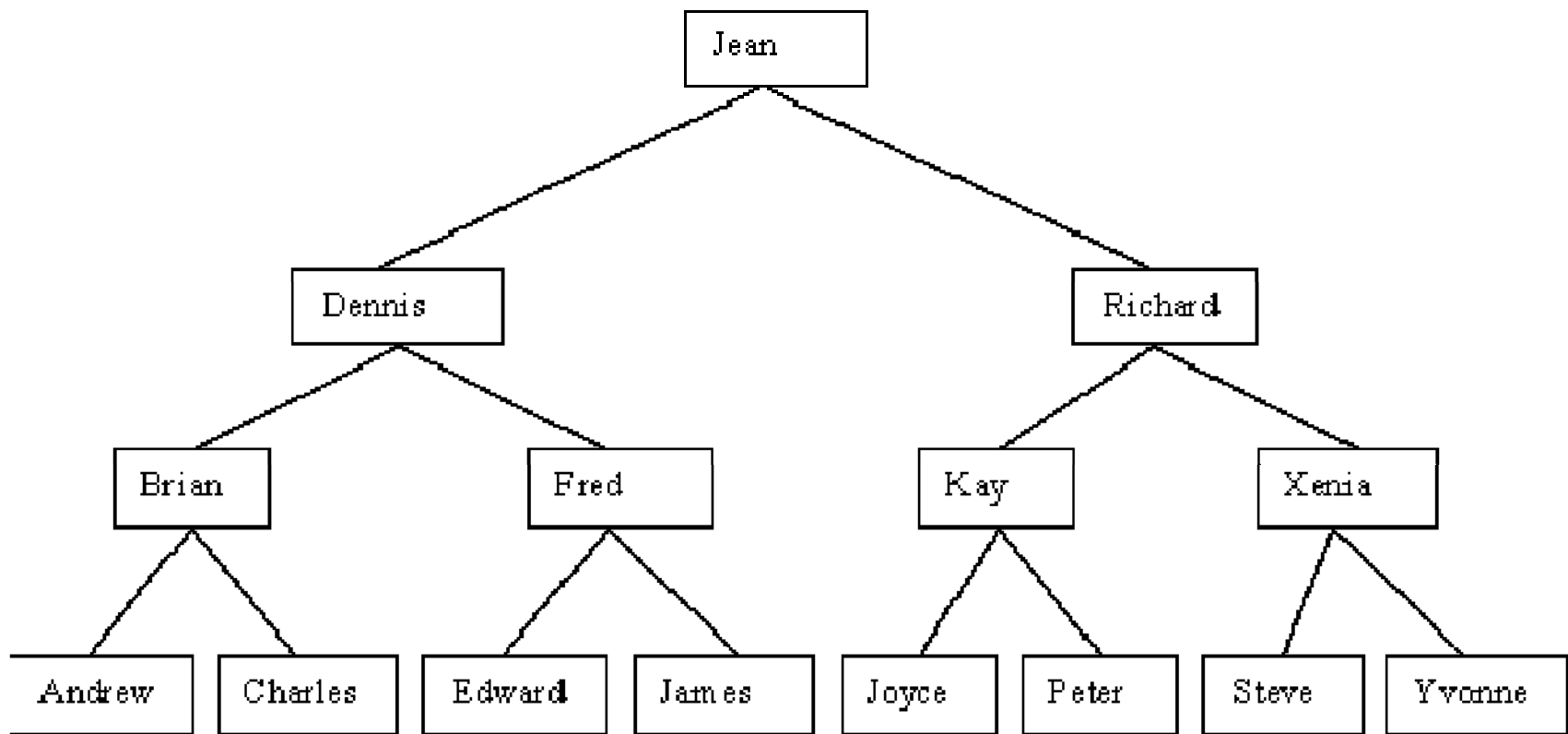
- Some are built to enable fast searching
- Some enable fast insertion/deletion

— What	Tree	HashTable
— Add	$O(\lg N+)$	$O(1)$
— Delete	$O(\lg N+)$	$O(1)$
— Search	$O(\lg N)$	$O(1)$

# Trees

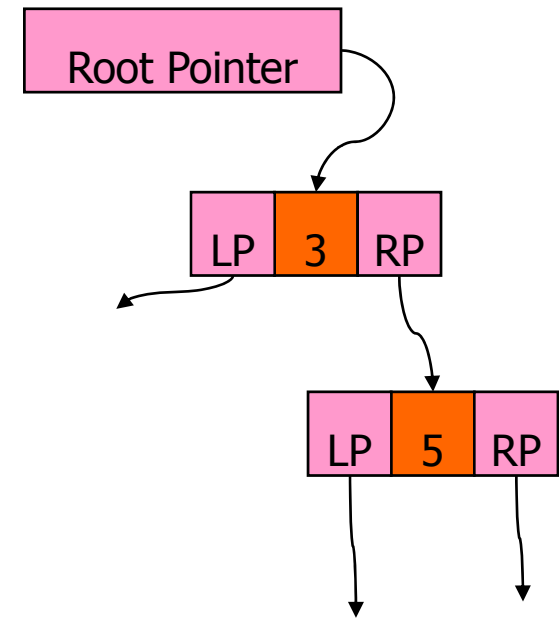
- A Tree is a node-link structure
- It is built to enable fast searching
  - What Runtime
  - Add A little slower
  - Delete A little slower
  - Search Much faster

# Binary Search Tree

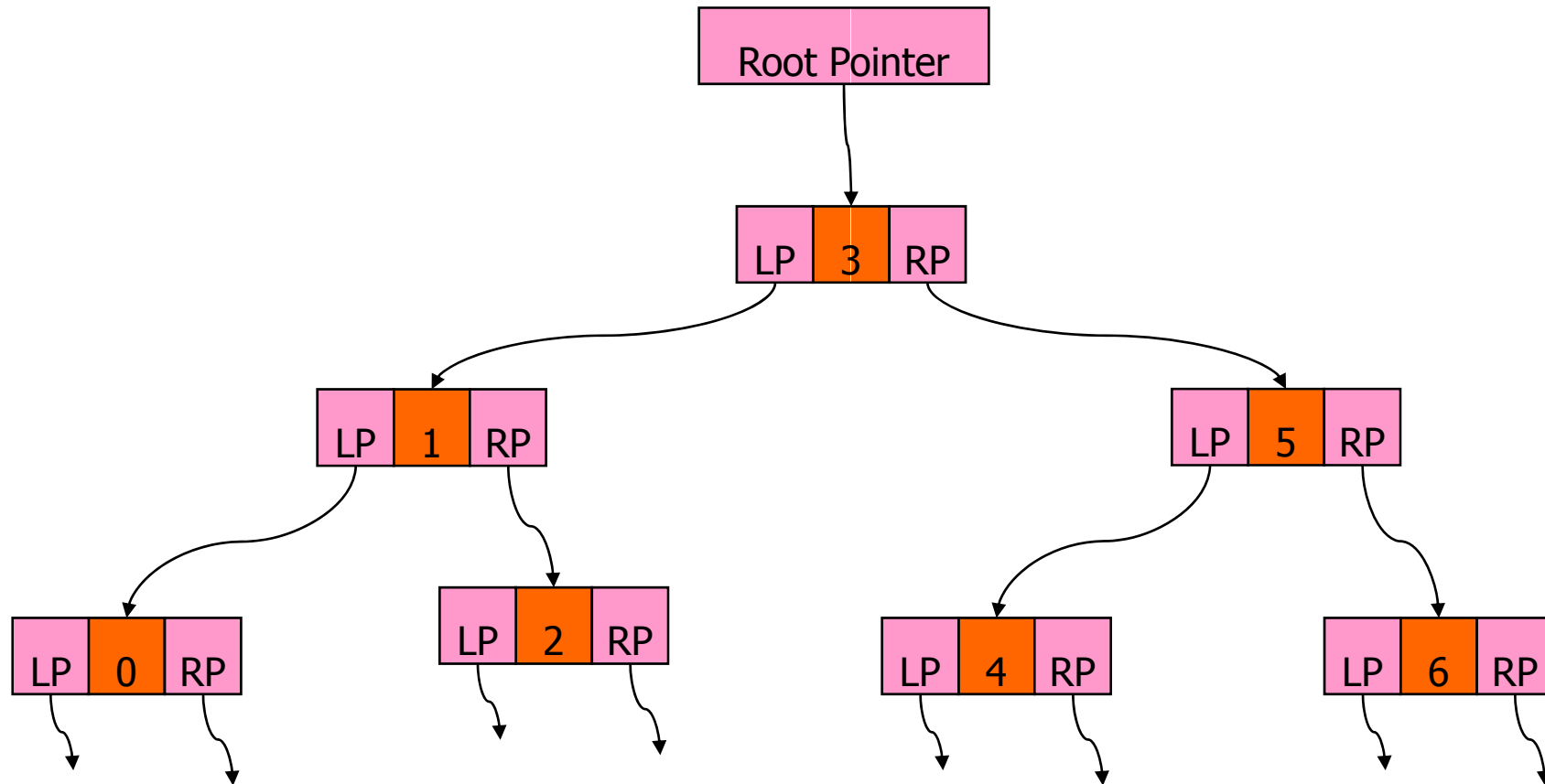


# Binary Search Tree

- Each **Node** has **two** links
  - **left** and **right** child
  - Top node is **root**
  - Node without children is **leaf**
- **Binary** meaning **two** children
- **Search tree** because of the node arrangement



# Binary Search Tree



# Binary Search Tree

- For **Any** node **n**

- To the **left**

- All child values are **Less Than n**

- To the **right**

- All child values are **Greater Than n**

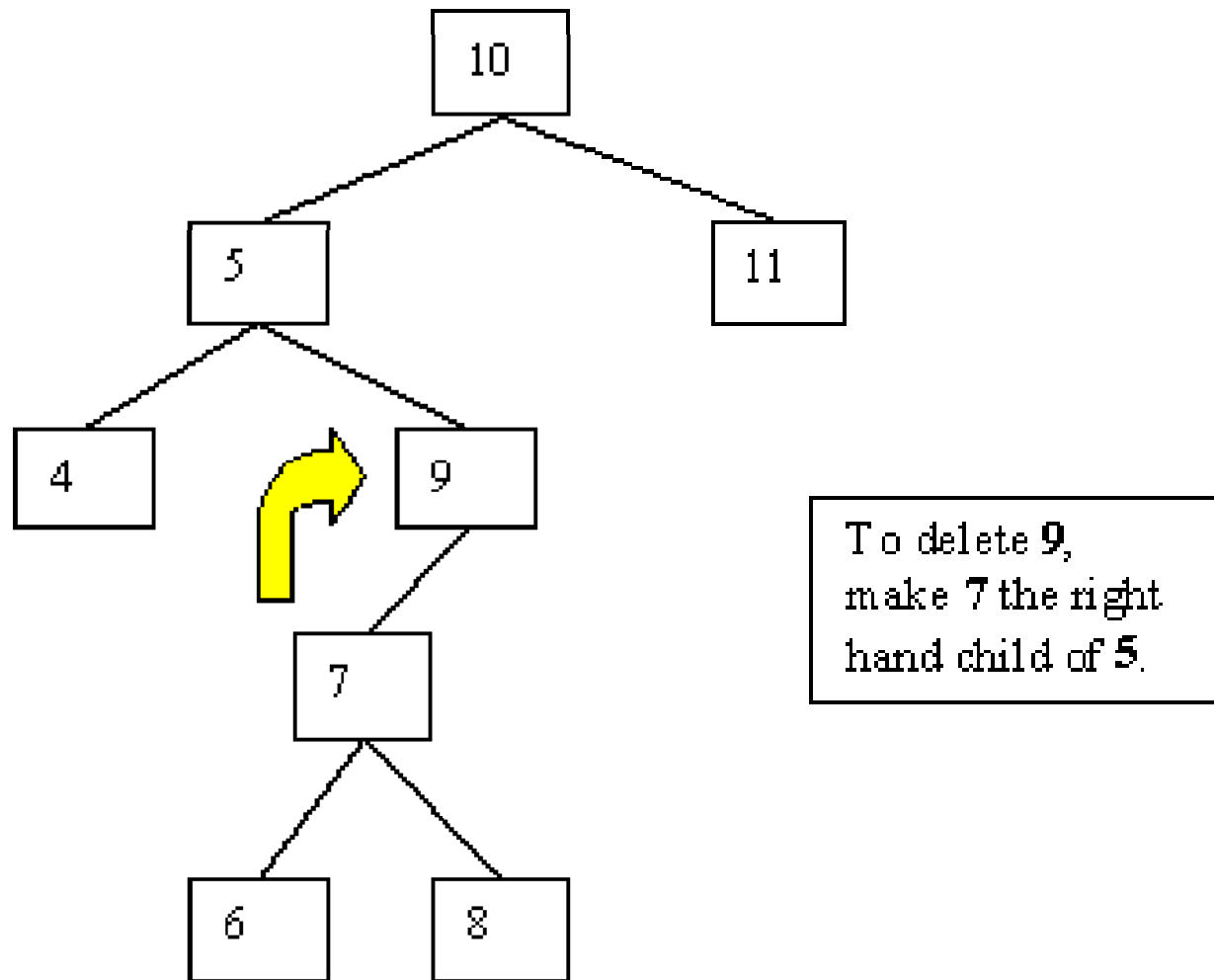
- [http://www.csanimated.com/animation.php?t=Binary\\_search\\_tree](http://www.csanimated.com/animation.php?t=Binary_search_tree)



# Binary Search Tree Search

```
class BNode {  
    int    key ;  
    BNode left, right ;  
}  
BNode root, cursor ;  
  
cursor = root ;  
while( cursor != null )    // search for SEARCH  
{  
    if( SEARCH == cursor.key )  
        SUCCESS ;  
    if( SEARCH > cursor.key )  
        cursor = cursor.right ;  
    else  
        cursor = cursor.left ;  
}
```

# Delete



# Trees in Java

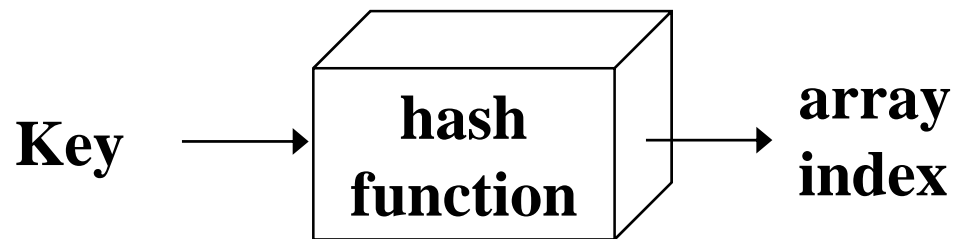
- Use the **TreeSet** class to get this functionality
  - You don't have to build the structure yourself
  - Like ArrayList, it stores **Objects**

```
TreeSet t1 = new TreeSet();  
t1.add( "aaa" );
```

```
Iterator it1 = t1.iterator();  
while( it1.hasNext() ) {  
    Object o1 =it1.next();  
    System.out.println( o1 );  
}
```

# Hash Table

- An array in which items are ***not*** stored consecutively - their place of storage is calculated using the key and a *hash function*



- *Hashed key*: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array

	key	entry
4		
10		
123		

# Hashing example

- 10 stock details, 10 table positions
- Stock numbers between 0 and 1000
- Use *hash function*: stock no. / 100
- What if we now insert stock no. 350?
  - Position 3 is occupied: there is a *collision*
- *Collision resolution strategy*: insert in the next free position (*linear probing*)
- Given a stock number, we find stock by using the hash function again, and use the collision resolution strategy if necessary

	key	entry
0	85	apples
1		
2		
3	323	guava
4	462	pears
5	350	oranges
6		
7		
8		
9	912	papaya

# Hashing

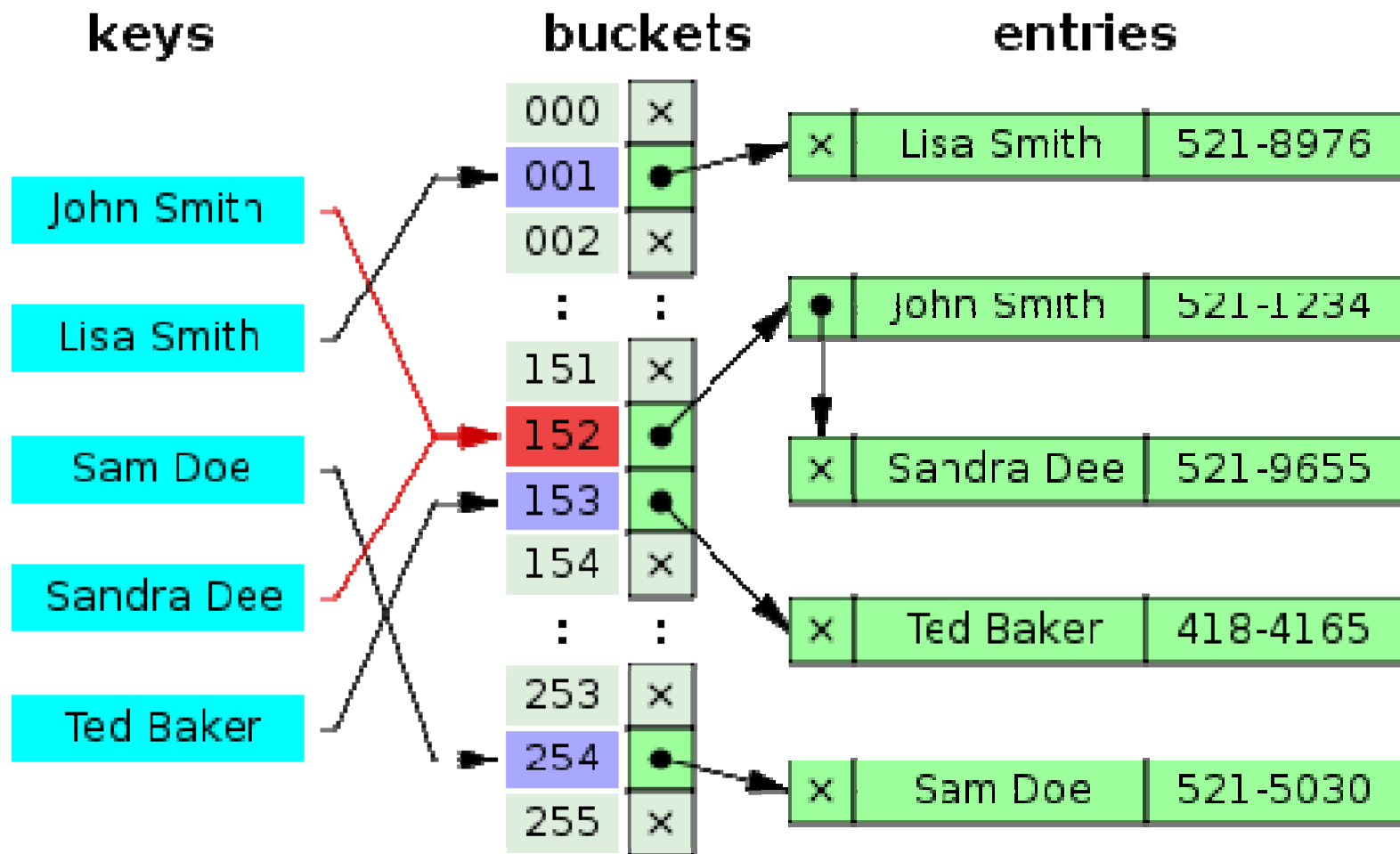
- **insert:** compute location, insert TableNode;  $O(1)$
- **search:** compute location, retrieve entry;  $O(1)$
- **remove:** compute location, set it to null;  $O(1)$

	key	entry
4		
10		
123		

# Hash Function

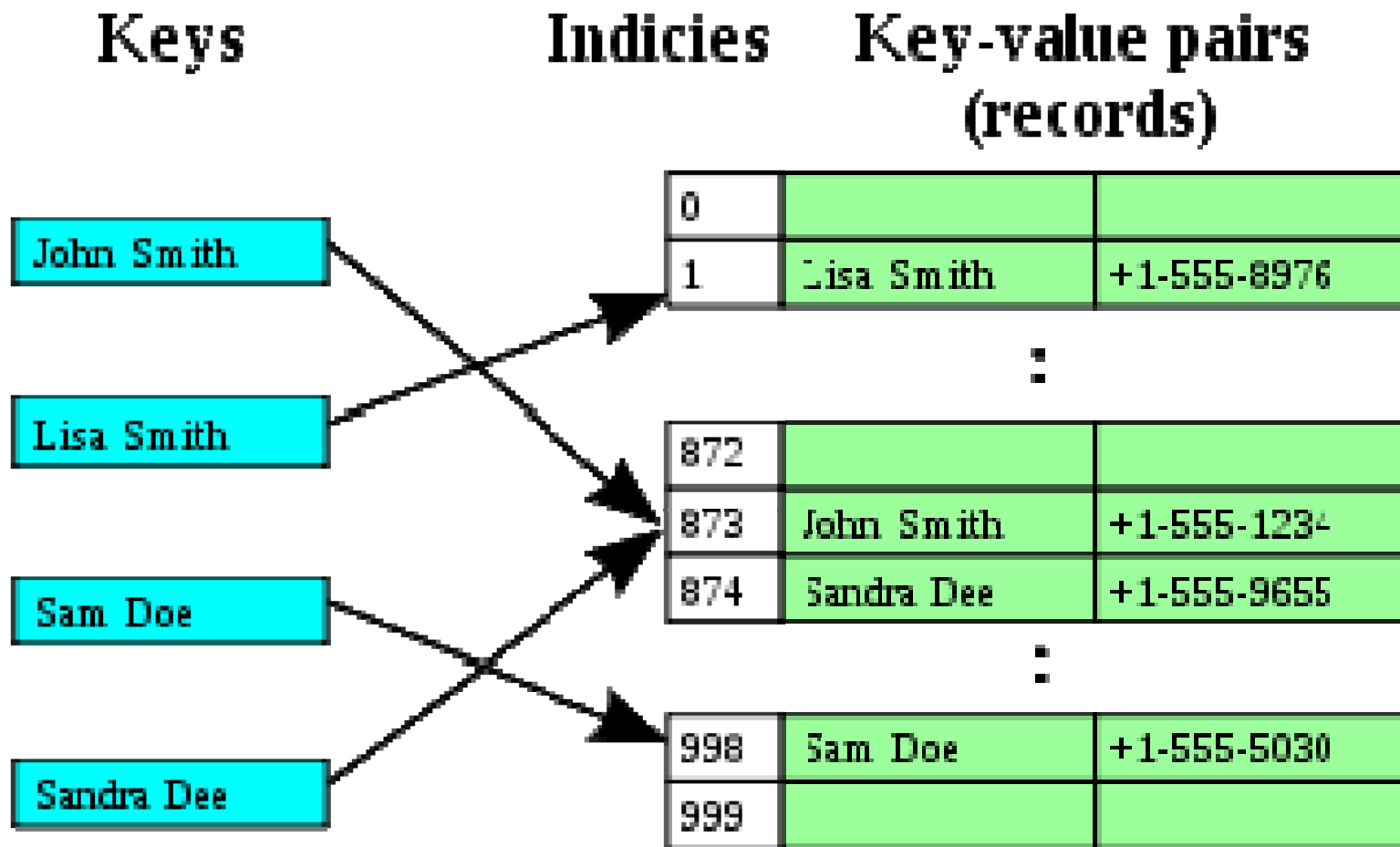
- The hash function:  $index = f(key, arrayLength)$ 
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should minimize *collisions*
- The collision resolution strategy
  - *Separate chaining*: chain together several keys/entries in each position
  - *Open addressing*: store the key/entry in a different position

# Hash collision resolved by *separate chaining*





# Hash collision resolved by *Open addressing*



# Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for online spelling checkers — an entire dictionary can be hashed and words checked in constant time
- A hash table is used for **database index** that improves the speed of data retrieval operations on a database table

# When to use hashing?

## ■ Good if:

- Need many searches in a reasonably stable table

## ■ Not So Good if

- Many insertions and deletions,
- If table traversals are needed
- Need things in sorted order
- More data than available memory
  - Use a tree and store leaves on disk

# Using Java HashMap

```
HashMap hm = new HashMap();  
hm.put( "Ava", "555-8976" );  
hm.put( "Mary", "555-1238" );  
hm.put( "Joe", "555-2121" );  
String phone = (String)hm.get( "Mary" );  
Iterator iter = hm.entrySet().iterator();  
while(iter.hasNext() ) {  
    Map.Entry me = (Map.Entry) iter.next();  
    System.out.print(me.getKey() + "'s phone#: " );  
    System.out.println(me.getValue() );  
}
```

# Search

- Often want to search for an item in a list

- In an unsorted list, must search linearly

12	5	31	62	9	4	26	15
----	---	----	----	---	---	----	----

- In a sorted list...

4	5	9	12	15	26	31	62
---	---	---	----	----	----	----	----

# *Binary search*

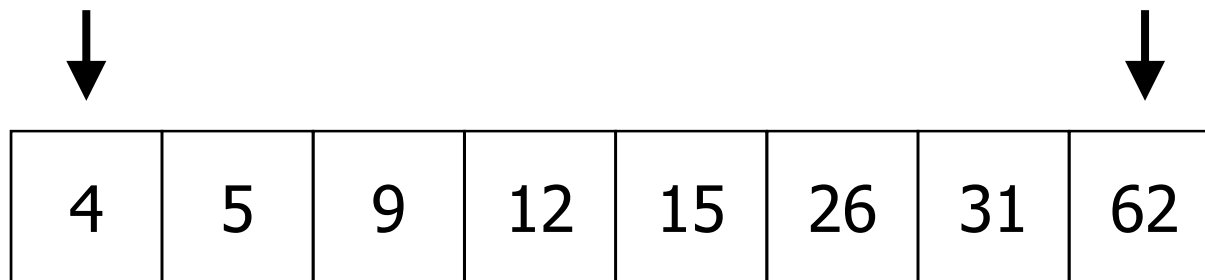
- *Binary search* is a procedure for searching an element in an array
  - The array has to be sorted in ascending order
  - The basic idea is to compare the middle element of the array with the target element, if not found, chop off half of the array, compare the middle element of remaining array with the target element...

# *Binary search*

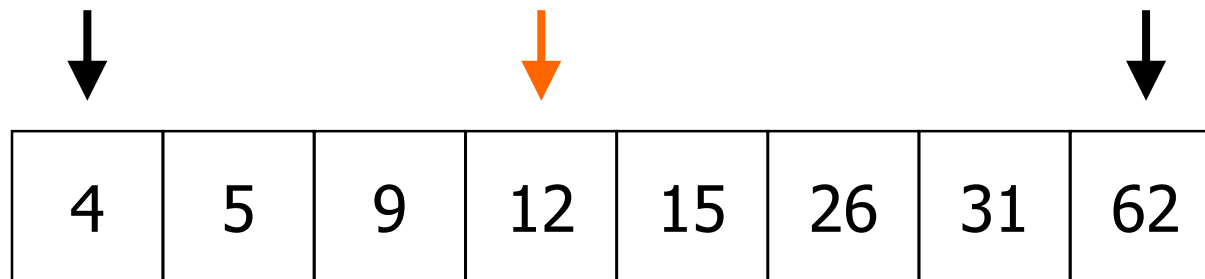
- The comparison can yield three results:
  - The middle element is the *target element* → search is successful and returns the index
  - The *target* element is *greater* than the middle element → the search is then continued *recursively to the right* of the element
  - The *target* element is *smaller* than the middle element → the search is then continued *recursively to the left* of the element

# Binary Search

- Start with index pointer at start and end



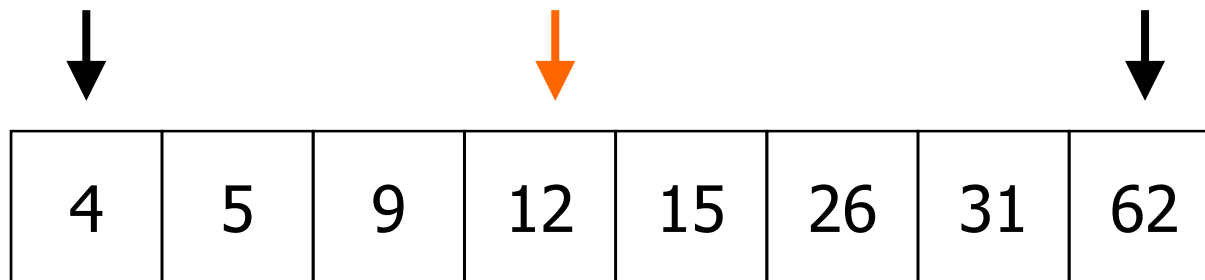
- Compute index between two end pointers



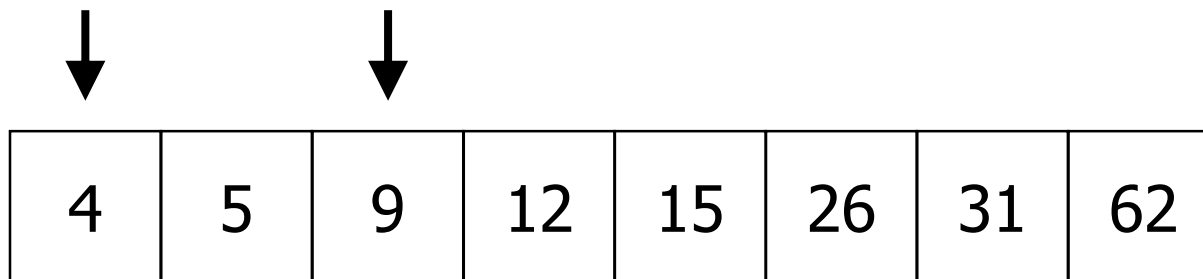


# Binary Search

- Compare middle item to search item



- If  $\text{search} < \text{mid}$ : move end to mid - 1



# Binary Search

```
int[] Arr = new int[8] ;
```

```
<populate array>
```

```
int search = 4 ;
```

```
int start = 0, end = Arr.length, mid ;
```

```
mid = (start + end)/2 ;
```

```
while( start <=end )
```

```
{
```

```
    if(search == Arr[mid] )
```

```
        SUCCESS ;
```

```
    if( search < Arr[mid] )
```

```
        end = mid - 1 ;
```

```
    else
```

```
        start = mid + 1 ;
```

```
}
```

# Binary Search

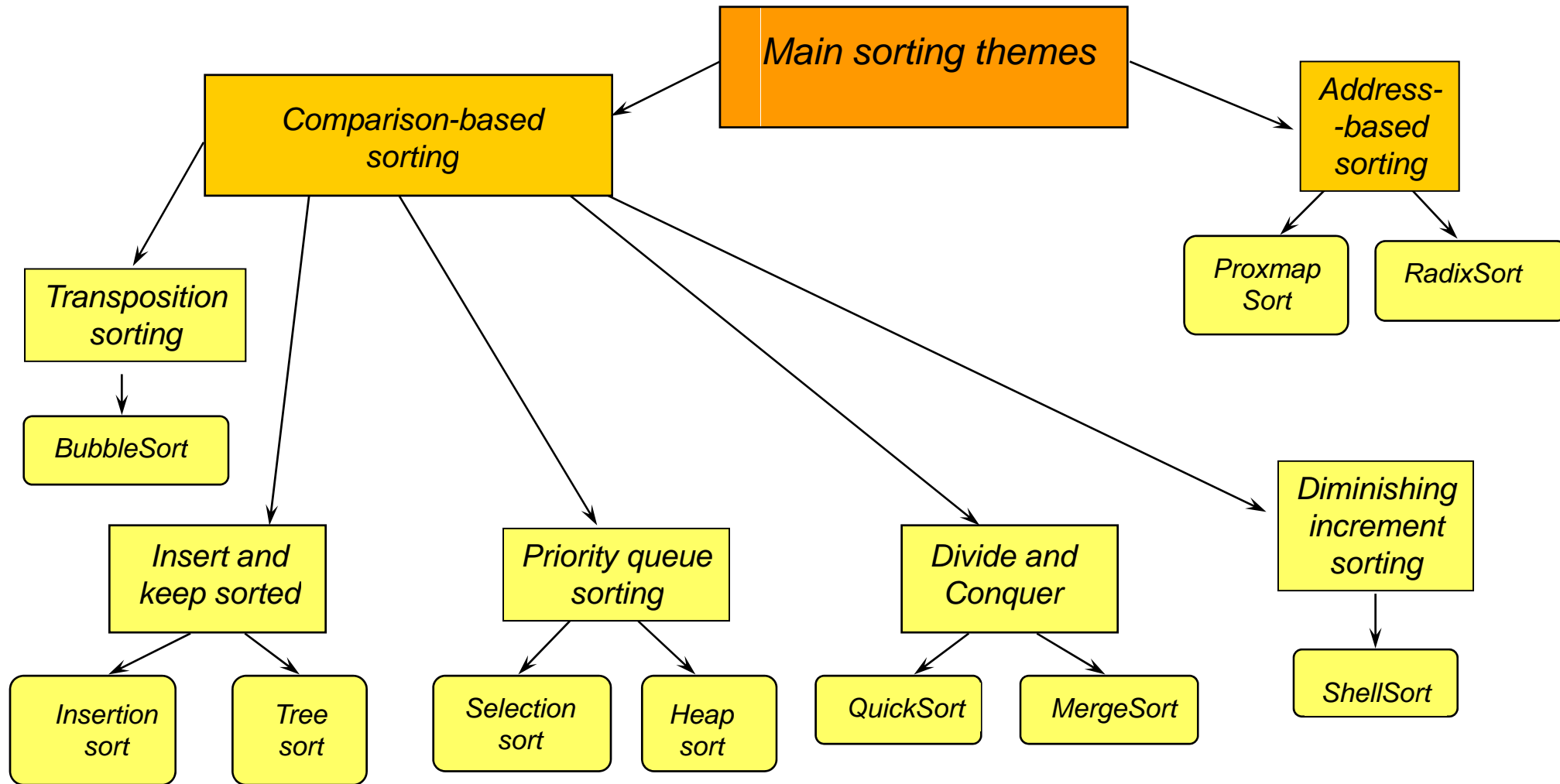
## ■ Run Time

- $O(\log(N))$
- Every iteration chops the list in half

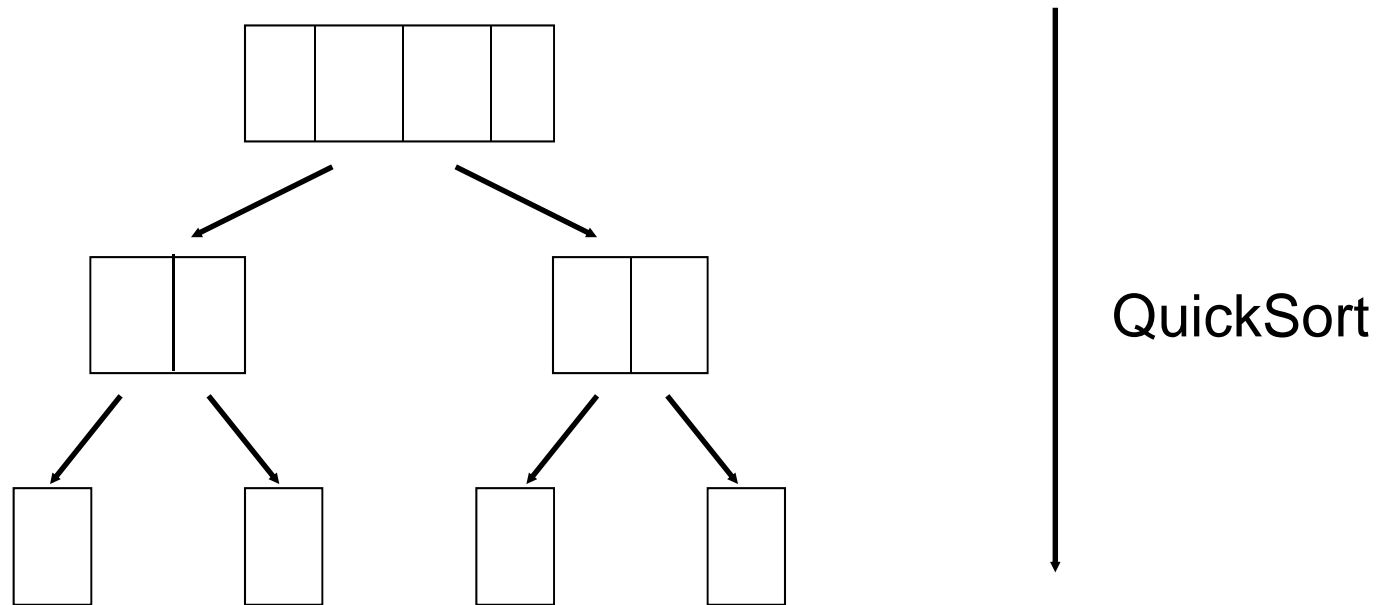
# Sorting

- Need a sorted list to do binary search
- Numerous sort algorithms

# The family of sorting methods



# Divide and Conquer sorting



As its name implies, *QuickSort* is the fastest known sorting algorithm *in practice*. Its average running time is  $O(n \log n)$

# QuickSort [divide and conquer sorting]

■ The idea is as follows:

1. If the number of elements to be sorted is 0 or 1, then return
2. **Pick** any element,  $v$  (called the *pivot*)
3. **Partition** the other elements into two disjoint sets,  $S_1$  of elements  $\leq v$ , and  $S_2$  of elements  $> v$
4. **Return** QuickSort ( $S_1$ ) followed by  $v$  followed by QuickSort ( $S_2$ )

# QuickSort example

5	1	4	2	10	3	9	15	12
---	---	---	---	----	---	---	----	----

Pick the middle element as the pivot, i.e., 10

Partition into the two subsets below

5	1	4	2	3	9
---	---	---	---	---	---

15	12
----	----

Sort the subsets (recursive process)

1	2	3	4	5	9
---	---	---	---	---	---

12	15
----	----

Recombine with the pivot

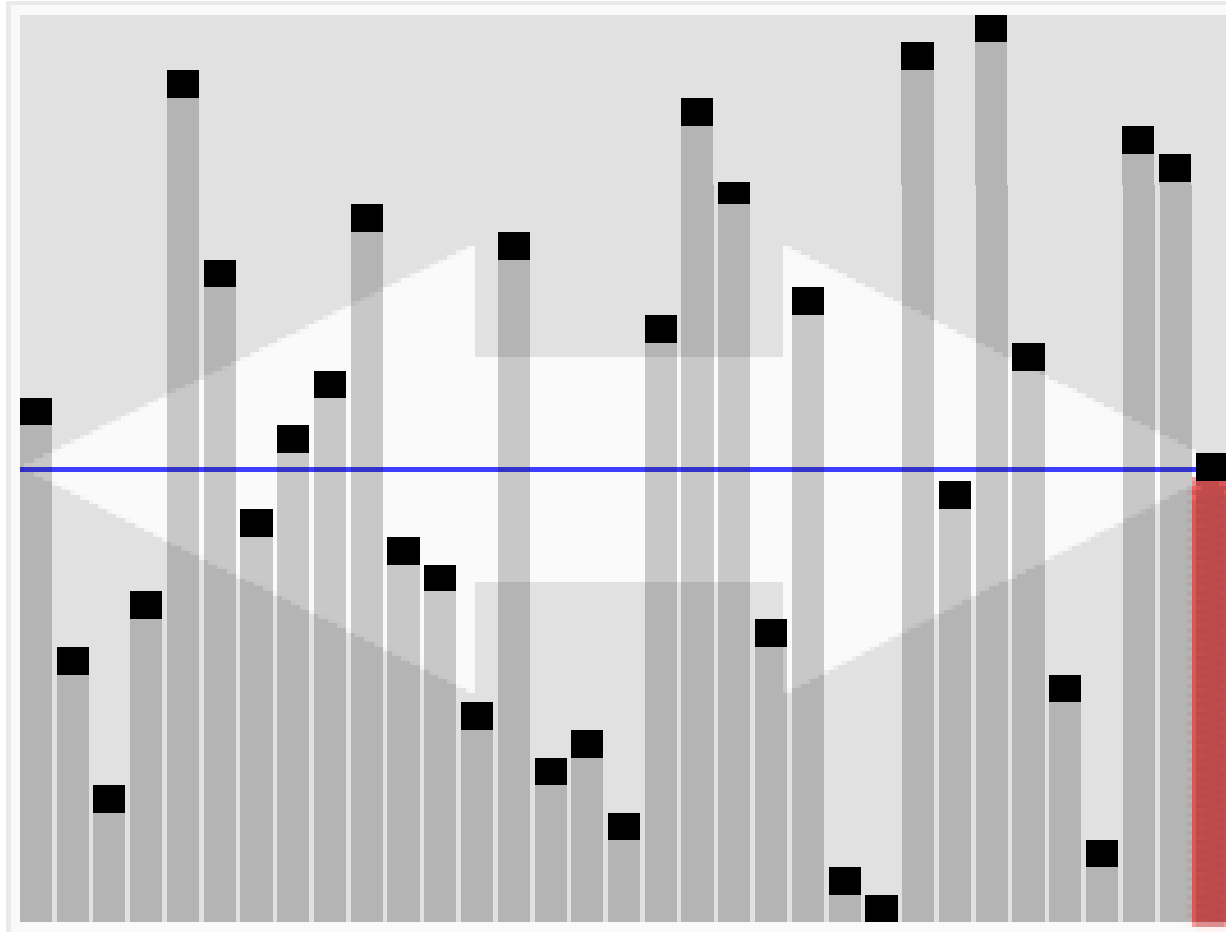
1	2	3	4	5	9	10	12	15
---	---	---	---	---	---	----	----	----



# Pseudocode for Quicksort

```
function quicksort (array)
  var list less, greater
  if length(array) ≤ 1
    return array //base case
  select a pivot value pivot from array
  for each x in array
    if  $x \leq \text{pivot}$  then append x to less
    else append x to greater
  return concatenate(quicksort(less), pivot, quicksort(greater))
```

# Visualization of the quicksort algorithm



# Java

## ■ Sort and binary search provided on Arrays

- *void sort(int[] a) , void sort(float[] a)*

- ints, floats,...

- *int binarySearch(int[] a, int key)*

- *int binarySearch(float[] a, float key)*

- ints, floats...

# Summary

- Overview of Data structures
- Data Structures and runtime reduction
  - Binary Tree & HashTable
- Search
  - Binary search
- Sorting
  - Quick sort