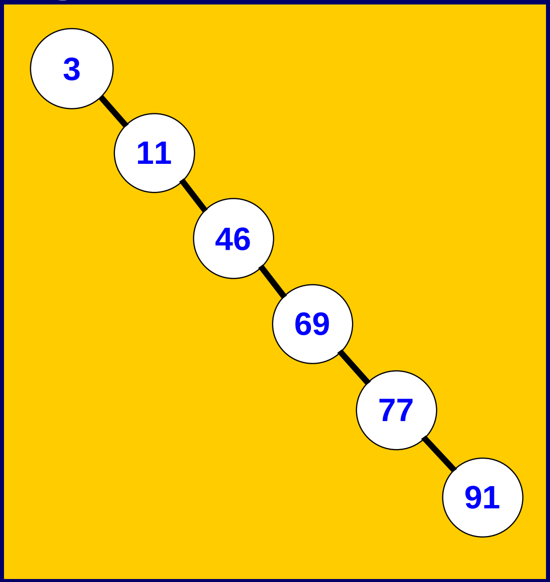
BST search efficiency

- Q: what determines the average time to find a value in a tree containing n nodes?
- A: average path length from root to nodes.
  - Q: how long is that?
  - Path lengths ("depths"): 1 (root) at depth 0, 2 at depth 1, 4 at depth 2, 8 at depth 3, …, log *n* levels in full tree

$$average = \frac{1}{n} \cdot \sum_{i=0}^{\log n} 2^i \cdot i \approx \log n$$

- But …
  - … tree must be balanced!
  - Or complexity can reach O(n)

→



Insert to a BST

- Same general strategy as find operation:

```
if (info < current node) insert to left;
else if (info > current node) insert to right;
else – duplicate info – abort insert;
```

  – Need a way to signal "unsuccessful" insert
    - Project 3 ADT – insert method returns a boolean value – true if successful, false otherwise
- Use either iterative or recursive approach
- 2 potential base cases for recursive version:
  – Already in tree – so return false; do not insert again
  – An empty tree where it should go – so set parent's link

Insertion order affects the tree?

- Try inserting these values *in this order*:
  `6, 4, 9, 3, 11, 7`
- Q: does the insertion order matter?
- A: yes!
  – Proof – insert same values in this order:
    `3, 4, 6, 7, 9, 11`
- Moral: sorted order is bad, random is good.
  – Note: cheaper to insert randomly, than try to set up self-balancing trees (see AVL trees)

Deleting a node (outline)

- First step: find node (keep track of parent)
- Rest depends on how many children it has
  - No children: no problem – just delete it (by setting appropriate parent link to null)
  - One child: still easy – just move that child "up" the tree (set parent link to that child)
  - Two children: more difficult – strategy is to replace the node with (either) largest value in its left subtree (or smallest in right subtree) – may lead to one more delete
- Generally, deleteNode method will return a node pointer – to replace the child pointer of parent

deleteNode algorithm

- Pseudocode for an external method:

```
TreeNode deleteNode(Comparable item,
                    TreeNode node)  {
    if (item is less than node's item)
        // delete from left subtree (unless there is no left subtree)
        // return result of delete (or null if no left subtree)

    else if (item is greater than node's item)
        // same as above, but substitute right subtree

    else // node contains the item to be deleted
        // return result of delete this node  ;
}
```

Actually removing a node

- More pseudocode (with strategic real code mixed in):

```
TreeNode deleteThis(TreeNode node) {

    if (node is a leaf)
        // return a null result

    else if (node has just one child )
        // return that child

    else { // node has two children
        // find "greatest" node in left subtree
        // copy item of greatest node in left subtree to node.item
        // deleteNode(item, node.left);
        return node;
    }
}
```

greatestNode, & other utilities

- Greatest node in BST is *all the way* to the right
  - So it is easy to find with recursion:

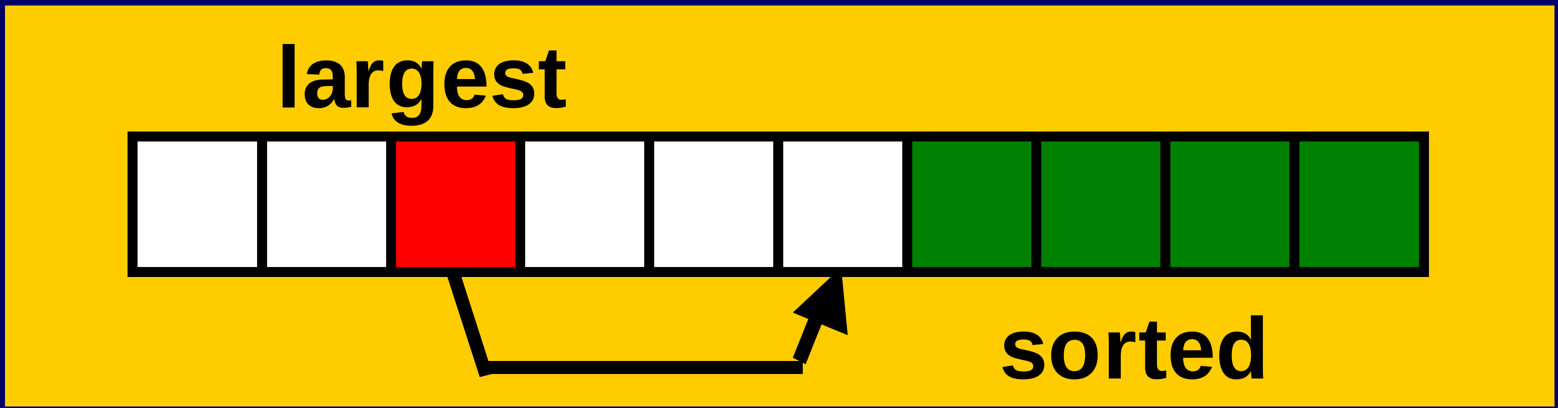
```
TreeNode greatestNode(TreeNode node) {
    if (node.right == null)
        return node;
    else return greatestNode(node.right);
}
```

- Use recursion to calculate height too
  - At any node: `1 + maximum(`*left height, right height*`)`
- To count: “traverse” the nodes – add 1 at each visit
- Other methods from Project 3, part 2:
  - **Think recursively!**

Sorting

- Probably *the* most expensive common operation
- Problem: arrange `a[0..n-1]` by some ordering
  - e.g., in ascending order: `a[i-1]<=a[i], 0<i<n`
- Two general types of strategies
  - Comparison-based sorting – includes most strategies
    - Apply to any comparable data – (key, info) pairs
    - Lots of simple, inefficient algorithms
    - Some not-so-simple, but more efficient algorithms
  - Address calculation sorting – rarely used in practice
    - Must be tailored to fit the data – not all data are suitable

Selection sort



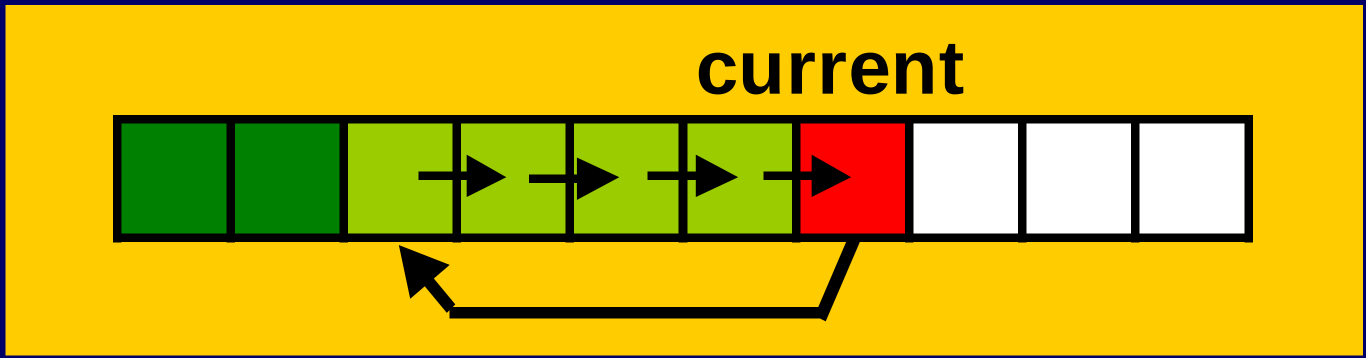
- Idea: build sorted sequence at end of array
- At each step:
  - *Find largest* value in not-yet-sorted portion
  - *Exchange* this value with the one at end of unsorted portion (now beginning of sorted portion)
- Complexity is $O(n^2)$– but simple to program
  - Also – best way to find k^{th} largest, or top k values

Heap sort

- Another priority queue sorting algorithm
  - Note about selection sort: unsorted part of array is like a priority queue – remove greatest value at each step
  - Also recall that heaps make faster priority queues
- Idea: create heap out of unsorted portion, then remove one at a time and put in sorted portion
- Complexity is `O(n log n)`
  - `O(n)` to create heap + `O(n log n)` to remove/reheapify
- Note proof: `O(n log n)` is the fastest possible class of any *comparison-based* sorting algorithm
  - But constants do matter – so some are faster than others

Insertion sort

- Generally "better" than other simple algorithms
- Inserts one element into sorted part of array
  - Must move other elements to make room for it



- Complexity is $O(n^2)$ (code)
  - But runs faster than selection sort and others in class
  - Really quick on *nearly sorted* array
- Often used to supplement more sophisticated sorts

Divide & conquer strategies

- Idea: (1) divide array in two; (2) sort each part; (3) combine two parts to overall solution
- e.g., mergeSort

```
if (array is big enough to continue splitting) →
   divide array into left half and right half;
   mergeSort(left half);
   mergeSort(right half);
   merge(left half and right half together);
else → sort small array in a simpler way
```

  - Need `2n` space, and `O(n)` step to merge two halves
  - Overall complexity is `O(n log n)`
  - The best sort for large files (especially if too big for memory)
- Used in `java.util.Arrays.sort(Object[] a)`
  - `Collections.sort(`*a list*`)` copies to array, uses Arrays.sort

Quick sort

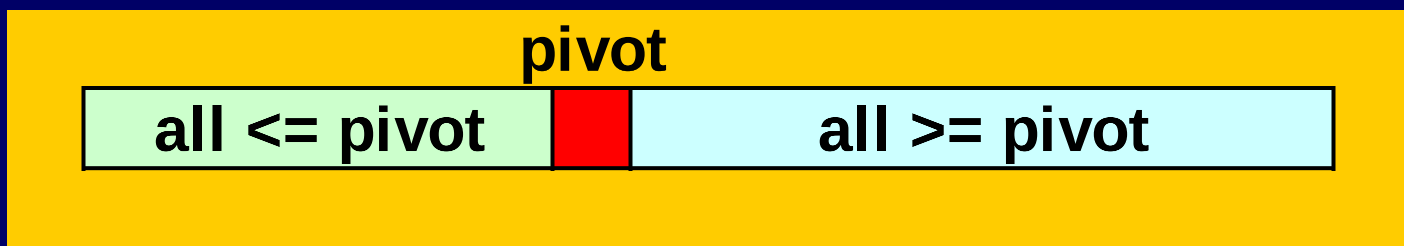
- Invented in 1960 by C.A.R. Hoare
  - Studied extensively by many people since
  - Probably used more than any other sorting algorithm
- Basic (recursive) quicksort algorithm:

```
if (there is something to sort)
{    partition array;
     sort left part;
     sort right part;  }
```

  - All the work is done by partition function
  - So there is no need to merge anything at the end

Partitioning (for quickSort)

- Arrange so elements in the two sub-arrays are on correct side of a pivot element
  - Also means pivot element ends up in its final position



- Done by performing two series of “scans”

```
scan from (i = left) until a[i] >= pivot;
scan from (j = right) until a[j] <= pivot;
swap a[i] and a[j], and continue both scans;
stop scanning when i >= j;
```

(code)

Quick sort (cont.)

- Complexity is `O(n log n)` on average
  - Fastest comparison-based sorting algorithm
  - But overkill, and not-so-fast with small arrays
    - Um … what about a small partition?!
  - One optimization applies insertion sort for partitions smaller than than 7 elements
- Also worst case is $O(n^2)$!
  - Depends on initial ordering and choice of pivot
- Used in `Arrays.sort(`*primitive array*`)`

A table ADT (a.k.a. a Dictionary)

```
interface Table {
    // Put information in the table, and a unique key to identify it:
    boolean put(Comparable key, Object info);
    // Get information from the table, according to the key value:
    Object get(Comparable key);
    // Update information that is already in the table:
    boolean update(Comparable key, Object newInfo);
    // Remove information (and associated key) from the table:
    boolean remove(Comparable key);
      // Above methods return false if unsuccessful (except get returns null)
    // Print all information in table, in the order of the keys:
    void printAll();
}
```

Table implementation options

- Many possibilities – depends on application
  - And how much trouble efficiency is worth
- Option 1: use a BST
  - To put: insertTree using key for ordering
  - To update: deleteTree, then insertTree
  - To printAll: use in-order traversal
- Option 2: sorted array with binary searching
- Option 3: implement as a “hash table”
  - Hashing – later

Recursive binary searching

- Start with *sorted* array of items: `a[0..n-1]`

  ```
  public class Item implements Comparable<Item> {…}
  ```

- Binary searching algorithm is naturally recursive:

```
int bsearch(Item key, Item a[], int left, int right) {
   // first call is for left=0, and right=n-1
   if (left > right) return -1; // unsuccessful search
   int middle = (left + right) / 2;  // location of middle item
   int comp = key.compareTo(a[middle]);
   if (comp == 0) return middle;// success
   if (comp > 0)  // otherwise search one half or the other
      return bsearch(key, a, middle+1, right);
   else return bsearch(key, a, left, middle-1);
}
```

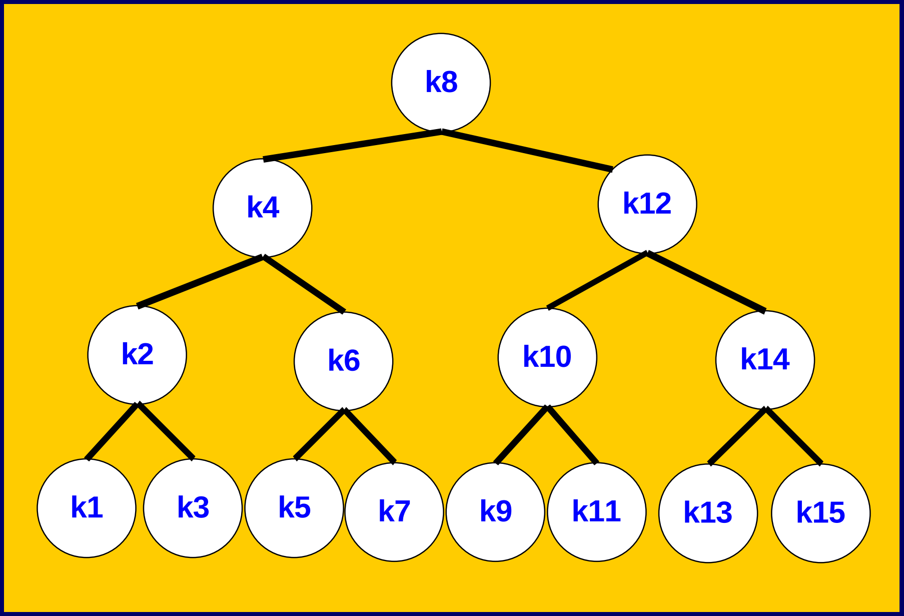
Iterative binary searching

```
int bsearch(Item key, Item a[]) {
   int low = 0, high = a.length-1, middle;
   while (low <= high) {
      middle = (low + high) / 2;
      int comp = key.compareTo(a[middle]);
      if (comp == 0) return middle; // success
      if (comp > 0) low = middle + 1;
      else high = middle – 1;
   }
   return -1; // unsuccessful search
}
```

- Both versions are same complexity class (next slide)
  - Interpolation search, by the way, is in a faster class
    - Trick is to calculate middle more intelligently

Complexity of binary search

- Say array has 15 elements, $k_1..k_{15}$: `a[0..14]`
  - If key is at k_8 (`a[7]`) then found by 1 comparison
  - If key is at k_4 or k_{12}, takes 3 comparisons …
- i.e., it's just like searching a BST



- Problem size is halved at each step
  - So complexity class is `O(log n)`
- Interpolation search reduces more quickly
  - Class is `O(log log n)`

Hashing ideas and concepts

- Idea: transform arbitrary key domain (e.g., strings) into "dense integer range"
  - Then use result as index to table
  - `int index = hash(key);` // transform key to int
- Collisions: `hash(k1)==hash(k2), k1 != k2`
  - Usually impossible to avoid ("perfect hashing" rare)
  - Therefore, must have a way to handle collisions
    - e.g., if using "open addressing" techniques - `while (`*occupied*`(index)) index = probe(key);`
- Concept: insertion/searching is quick – but only until the table starts to get filled up
  - Then collisions start happening too often!

Implementing a hash table

- Constructor allocates memory for array of items, and initializes all items to “empty” key
  - `size` is size of array
  - `n` is the number of items in the table
  - Load factor is `n / size`
- `put` method uses `hash(key)` (and `probe(key)`if open address hashing) to find empty slot for new item
  - May be necessary to *resize* array
    - If so, also necessary to *rehash* existing items
    - If open address hashing, resize when load factor > 50%

Open address hashing

- `get` & `update` methods use `hash(key)` and `probe(key)` in *exact same sequence* as `put`
  - To find existing info where it was put
- `remove` is more complicated
  - Cannot just remove an item – future probes for get and update might terminate prematurely at empty slot
    - Common trick is to have “deleted” key
      - Problem with that is table can seem full prematurely
    - Inefficient alternative rehashes all items when any removed
- Note: to `printAll` in key order – must sort first
  - So `O(n log n)` at best!

Hash functions

- Goal: uniform distribution of keys
  - Means each index of table is equally likely
  - Important for reducing collisions
- Common approach is a *restricted transformation*
  - Step 1 – transform key to *large* integer
  - Step 2 – restrict integer to 0…size-1
    - Usually done with modulus operator - %
- Lots of variations – partly depends on key type
  - General observation: hard to find a good hash function
  - Note: should be "cheap" to compute too – e.g., division is slower on most CPUs than addition

Resolving collisions

- Simplest open address approach is linear probing
  - If (`index = hash(key)`) is not empty, try `index+1`, then `index+2`, …, until empty slot
    - Note: searching for first "open address"
  - Leads to "primary clusters" – collisions bunch up
- Quadratic probing – vary probe, like 1, 3, 6, …
  - Leads to "secondary clusters" but not as quickly
- Double hashing – `probe(key)` varies by key
  - Best open addressing approach for avoiding clusters
- Or completely different approach – "chaining"

Chaining

- Constructor allocates memory for array of Lists, and creates an empty list for each element of the array
- `put` method uses `hash(key)` and appends to end of list at that index of array
  - Still should resize when load factor approaches 80%
    - Clustering is not a problem, but long lists slow performance
- `remove` method is easier now – just delete from list
- But lots more overhead than open addressing
  - Must store node links as well as key and info
  - Use list method calls instead of direct array access

Compare 3 table implementations

Table operation	Hash table	BST	Sorted array
create (new table)	`O(n)`	`O(1)`	`O(n)`
get, update	`O(1)`	`O(log n)`	`O(log n)`
put	`O(1)`	`O(log n)`	`O(n)`
remove	`O(1)`	`O(log n)`	`O(n)`
printAll	`O(n log n)`	`O(n)`	`O(n)`

- Conclusion: choice depends on table purpose and size of `n`
- Q. Ever want to use a sorted array?
  - A. It *depends*!