54 Balanced search trees, and sorting odds and ends *(3 activities)*

54.1 Explore techniques for balancing a search tree.(15 steps)

54.1.1 (Display page) Balanced search trees: an overview

Balanced search trees: an overview

Over the past several weeks, we have analyzed the performance of algorithms for access and insertion into binary search trees under the assumption that the trees were *balanced*. Informally, that means that the paths from root to leaves are all roughly the same length, and that we won't have to worry about lopsided trees in which search is linear rather than logarithmic. This balancing doesn't happen automatically, and we have seen how to insert items into a binary search tree to produce worst-case search behavior. There are two approaches to tree balancing: incremental balance, where at each insertion or deletion we do a bit of work to keep the tree balanced; and all-at-once balancing, where we don't do anything to keep the tree balanced until it gets too lopsided, then we completely rebalance the tree. In the activities of this segment, we start by analyzing some tree balancing code. Then we explore how much work is involved in maintaining *complete balance*. We'll move on to explore two kinds of balanced search trees, *AVL trees* and *2-3 trees*. Finally, we'll investigate *tries*, which provide a mechanism for improving on log N access time.

54.1.2 (Display page) Code for rebalancing a tree

Code for rebalancing a tree

Consider the following code for building a balanced tree out of items in a LinkedList.

```
public BinaryTree (LinkedList list) {
    myRoot = ll2tree (list.iterator ( ), list.size ( ));
}

private TreeNode ll2tree (Iterator iter, int n) {
    if (n == 0) {
        return null;
    }
    if (n == 1) {
        return new TreeNode (iter.next ( ));
    }
    TreeNode child = ll2tree (iter, n/2);
    TreeNode root = new TreeNode (iter.next ( ));
    root.myLeft = child;
    root.myRight = ll2tree (iter, n - n/2 - 1);
    return root;
}
```

The next few steps ask questions about this code. You may wish to test the code (by incorporating it into your BinaryTree class) prior to answering the questions.

54.1.3 (Brainstorm) Comments for ll2tree

Provide a good comment for the ll2tree method that specifies preconditions and invariant relations involving the iter and n arguments, and describes the returned tree in terms of iter and n.

54.1.4 (Brainstorm) Runtime estimate for ll2tree

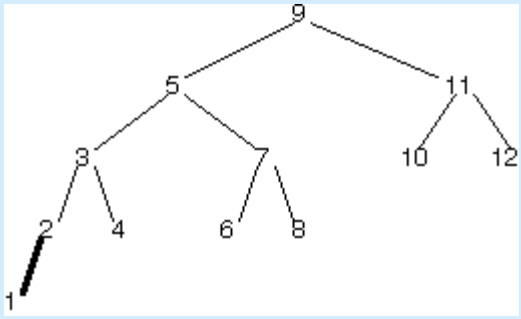
Estimate ll2tree's running time, and briefly justify your estimate.

54.1.5 (Brainstorm) Shape of the tree ll2tree builds

Is the tree built by ll2tree heap-shaped? Briefly explain why, or give a counterexample.

54.1.6 (Brainstorm) Worst case for maintaining complete balance

Suppose we wished to maintain a heap-shaped binary search tree after each insertion. Insertion of 1 into the

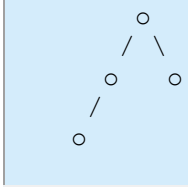
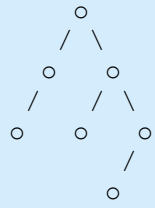
tree below, as marked in bold, is a worst case. 

How many TreeNodes would change as a result of inserting 1 into the above tree? Briefly explain your answer.

54.1.7 (Display page) "Almost-balanced" trees

"Almost-balanced" trees

The preceding step suggests that maintaining complete balance requires too much effort. A solution is to allow more flexibility via "almost-balanced" trees. In a *height-balanced* tree, the heights of the two subtrees differ by at most 1 for each non-leaf node in the tree. Here are some examples.

	
Height of left subtree is 2, height of right subtree is 1. Balance requirements also hold for the children.	Height of left subtree is 2, height of right subtree is 3. Balance requirements also hold for the children.

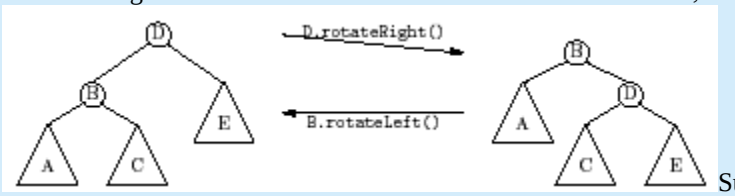
54.1.8 (Brainstorm) Question about the definition

The definition of height-balanced trees requires that the heights of subtrees differ by at most 1 in *every node of the tree*. Why is this extra condition necessary? (We might instead require only that the heights of the *root's* left and right subtrees differ by at most 1.)

54.1.9 (Display page) AVL trees

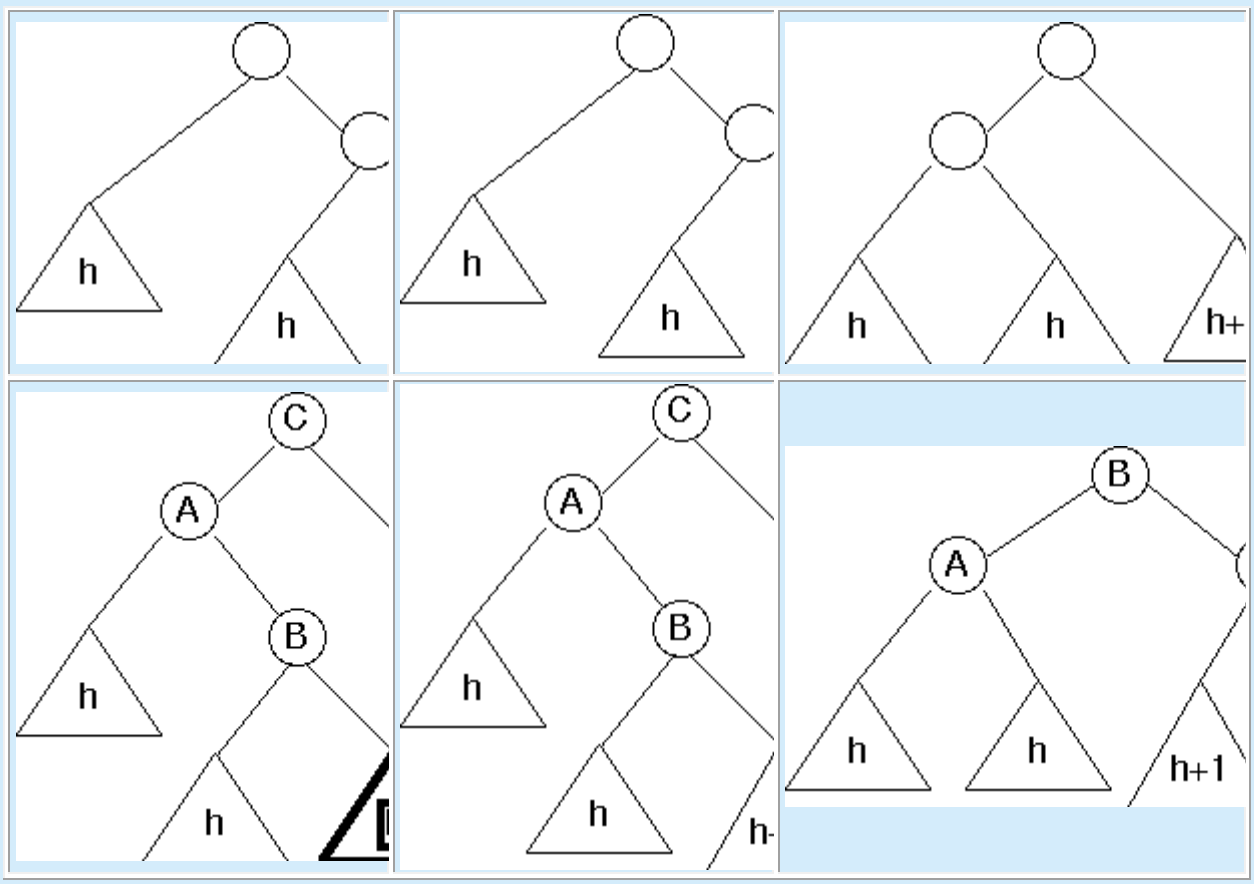
AVL trees

AVL trees (named after their Russian inventors, Adel'son-Vel'skii and Landis) are height-balanced binary search trees, in which information about tree height is stored in each node along with the myItem information. Restructuring of an AVL tree after insertion is done via *rotations*, an example of which is shown below.



Such a rotation preserves the order relationship among tree elements, but changes the levels of some of the nodes. Most insertions would not require any special handling. There are essentially two cases that do require tree restructuring, however. These are shown below. In each case, a insertion is done that increases the height of the subtree outlined in bold (from h to h+1).

before insertion	after insertion	after rotation

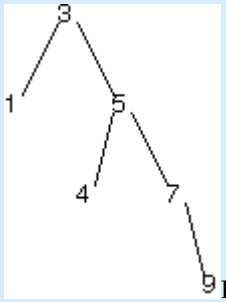


The second case is essentially a left rotation followed by a right rotation. Other cases are mirror images of the ones shown. The resulting tree may itself be too high compared to its sibling, so an insertion potentially propagates to the top of the tree, producing a worst case proportional to log N (where N is the number of tree elements).

54.1.10 (Self Test) Assignment statements involved in a rotation

Estimate the number of operations required to do a single rotation. Assume that N is the number of items in the tree.

54.1.11 (Brainstorm) Balancing a tree

Consider the following tree. 

Describe the result of applying a rotation to balance the tree.

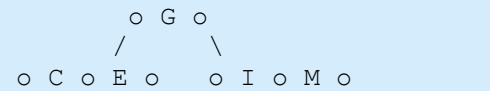
54.1.12 (Display page) 2-3 trees

2-3 trees

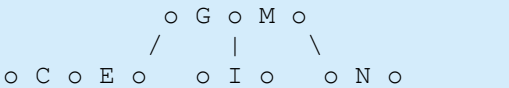
Another approach to "almost-balanced" trees is to relax the requirement that a node has only two subtrees. In a 2-3 tree, each nonleaf stores either one key and two subtree pointers, or two keys and three subtree pointers. A one-key, two-subtree node is like its BST counterpart; in a node with two keys, all the values in the left subtree are less than or equal to the first key, all the values in the middle subtree are between the two keys, and all the

values in the right subtree are greater than or equal to the second key. Thus search proceeds essentially as in a BST, with an extra comparison possibly required at each two-key node. All paths from the root to a leaf are the same length. This means search is requires at worst time proportional to log N, where N is the number of keys in the tree. Insertion into a 2-3 tree occurs at the bottom, just as in a BST. We first consider the simplest case of an insertion for which there is plenty of room, for example, inserting "E" into the tree

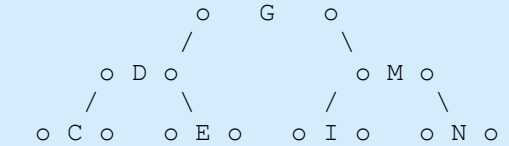
Following the link, we find that "E" belongs to the right of "C", so we just add a key for the corresponding node:



Now we insert "N". It goes to the right of "M", but there's no more room in that node. We must *split* the node into two nodes, each with one key ("I" in the first, "M" in the second), and send the *middle* key back up the tree to be inserted above. Since the "G" node has room, "M" is inserted there.



A third case arises when we insert into a full node and the nodes are full all the way back up the tree. In this case we must split the root node and increase the height of the tree. (Thus 2-3 trees grow "up" at the root rather than "down" at the leaves.) This case arises from inserting "D" into the tree above. It doesn't fit in the node with "C" and "E", so that node is split and "D" sent up a level. It also doesn't fit in the node with "G" and "M", so that node is split and "G" sent up a level. The following tree results.



After an insertion that propagates all the way up the tree, it will normally be a long time before another such general restructuring is necessary.

54.1.13 (Brainstorm) Growing a 2-3 tree

Suppose the keys 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are inserted sequentially into an initially empty 2-3 tree. Which insertion causes the *second* split to take place? Briefly explain your answer.

54.1.14 (Display page) B-trees

B-trees

A 2-3 tree is a special case of a structure called a *B-tree*. What varies among B-trees is the number of keys/subtrees per node. B-trees with lots of keys per node are especially useful for organizing storage of disk files, for example, in an operating system or database system. Since retrieval from a hard disk is quite slow compared to computation operations involving main memory, we generally try to process information in large chunks in such a situation, in order to minimize the number of disk accesses. The directory structure of a file system could be organized as a B-tree so that a single disk access would read an entire node of the tree. This would consist of (say) the text of a file or the contents of a directory of files. The java.util.TreeMap and TreeSet classes are implemented as "red-black trees", which are a variation of B-trees.

54.1.15 (Display page) Tries

Tries

When we covered hashing, we worked with a hashCode function for words that returned the internal code of the word's first letter. Though this was not a very good hash function, it suggests a method for organizing words in a tree structure. Here's an illustration, involving the storage of the words "a", "abase", "abash", "abate", "abbas", "axe", "axolotl", "fabric", and "facet". We first classify the words by their first letter:

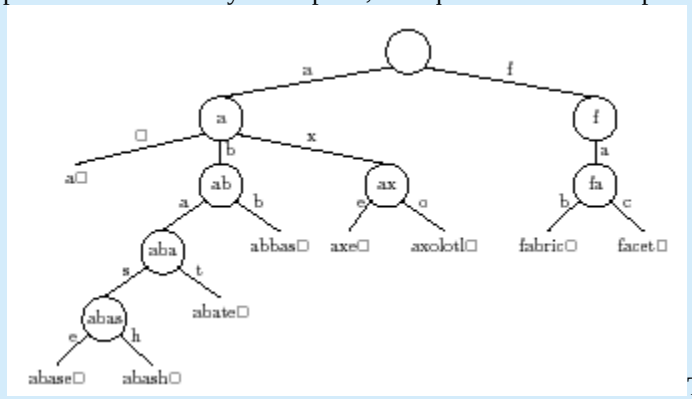
words starting with "a"	words starting with "f"

a abase abash abate abbas axe axolotl	fabric facet

Then, within each group, we form groups based on the second letter:

words starting with "a"			words starting with "f"
words with only "a"	words starting with "ab"	words starting with "ax"	words starting with "fa"
a	abase abash abate abbas	axe axolotl	fabric facet

We can continue this process, forming the tree shown below. The internal nodes are labeled to show the string prefixes to which they correspond; the square "character" represents the end of a word.



This structure is called a *trie* (pronounced "try"). If the nodes are implemented in a way that allows quick access to each child node, the time required to find a word in the structure will be at worst proportional to the number of letters in the word. Depending on how many words we're storing, this can provide a dramatic improvement over structures we've considered so far. Suppose, for example, that we want to store most of the 26^5 words consisting of five lower-case letters. Putting them in an array and using binary search requires over 20 comparisons to find a given word, as compared to 5 using a trie. The big disadvantage of using a trie is that we trade space efficiency for time efficiency.

54.2 Do a few more sorting-related activities.(9 steps)

54.2.1 (Display page) Finding the kth largest element

Finding the kth largest element

Suppose we want to find the kth largest element in an array. We could just sort the array to do this. Finding the kth item ought to be easier, however, since it asks for less information than sorting, and indeed finding the smallest or largest requires just one pass through the collection. You may recall the activity of finding the kth largest item (1-based) in a binary search tree augmented by size information in each node. The desired item was found as follows:

1. If the left subtree has k-1 items, then the root is the k item, so return it.
2. If the left subtree has k or more items, find the kth largest item in the left subtree.
3. Otherwise, let j be the number of items in the left subtree, and find the k-j-1st item in the right

subtree.

A binary search tree is similar to the recursion tree produced by the Quicksort algorithm. The root of the BST corresponds to the divider element in Quicksort; a lopsided BST, which causes poor search performance, corresponds exactly to a Quicksort in which the divider splits the elements unevenly. We use this similarity to adapt the Quicksort algorithm to the problem of finding the kth largest element. Here was the Quicksort algorithm.

1. If the collection to be sorted is empty or contains only one element, it's already sorted, so return.
2. Split the collection in two by *partitioning* around a "divider" element. One collection consists of elements greater than the divider, the other of elements less or equal to the divider.
3. Quicksort the two subcollections.
4. Concatenate the sorted large values with the divider, and then with the sorted small values.

The adaptation is as follows.

1. If the collection contains only one element, that's the one we're looking for, so return it.
2. Otherwise, partition the collection as in Quicksort, and let j be the number of values greater than the divider.
3. If k (1-based) is less than or equal to j, the item we're looking for must be among the values greater than the divider, so return the result of a recursive call.
4. If k = j, the kth largest item is the divider, so return it.
5. Otherwise, the item we're looking for is the k-j-1 largest item among the values less than or equal to the divider, so return the result of a recursive call.

The worst-case running time for this algorithm, like that for Quicksort, occurs when all the dividers produce splits as uneven as possible. On the average, however, it runs in time proportional to N, where N is the number of items in the collection being searched. Here's a brief explanation of why the algorithm runs in linear time in a good case. Partitioning is where all the work comes in. With optimal dividing, we partition the array of N elements, then a subarray of approximately N/2 elements, then a subarray of N/4 elements, and so on down to 1. Suppose that partitioning of j elements require P*j operations, where P is a constant. Then the total amount of partitioning time is approximately

```
P*N + P*N/2 + P*N/4 + ... + P*1
= P * (N + N/2 + N/4 + ... + 1)
= P * (N + N-1)
= 2*P*N - P
```

which is proportional to N.

54.2.2 (Brainstorm) Selection sort using a LinkedList

Here's an implementation of selection sort using LinkedLists.

```
public static LinkedList selectionSort (LinkedList values) {
    LinkedList sorted = new LinkedList ( );
    while (!values.isEmpty ( )) {
        Iterator iter = values.iterator ( );
        Comparable max = (Comparable) values.getFirst ( );
        while (iter.hasNext ( )) {
            Comparable obj = (Comparable) iter.next ( );
            if (max.compareTo (obj) < 0) {
                max = obj;
            }
        }
        values.remove (max); // removes first occurrence of max
        sorted.add (max);
    }
    return sorted;
```

```
}
```
In the version of selection sort we saw last week, we maintained a pointer to the max-so-far. Here, we keep track only of the maximum element; the call to remove must then do a linear search for the item to remove from the list. Does this linear search add more than a linear factor to the running time of selection sort? Briefly explain your answer.

54.2.3 (Display page) Sorting with multiple keys

Sorting with multiple keys

So far, we've been concerned with structuring or sorting using only a single set of keys. Much more common is the situation where a key has multiple components, and we'd like the keys to be sorted using *any* of those components. An example is the collection of files in a UNIX directory. The command

```
ls
```

lists the files sorted by name. The command

```
ls -s
```

lists the files sorted by size.

```
ls -t
```

lists the files sorted by the time of the last modification.

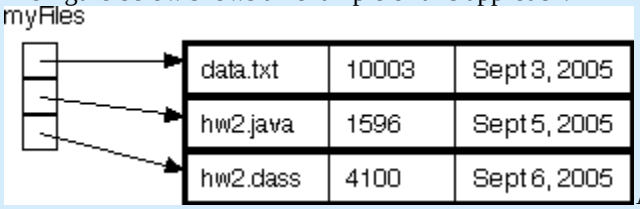
54.2.4 (Display page) An example

An example

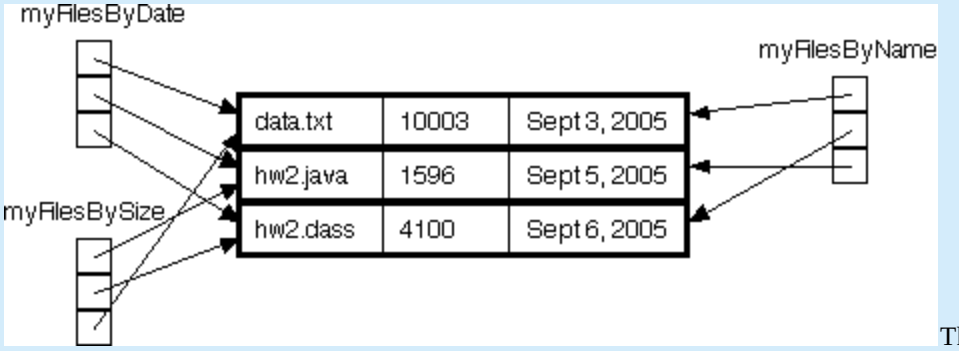
Consider the following example, that represents a flat (nonhierarchical) directory of file entries.

```
public class Directory {

  private FileEntry [ ] myFiles;

  private class FileEntry {
    public String myName;
    public int mySize;
    public GregorianDate myLastModifDate;
      ...
  }

  public void listByName ( ) ...
  public void listBySize ( ) ...
  public void listByLastModifDate ( ) ...
  public void add (FileEntry f)  ...
  public void remove (FileEntry f)  ...
}
```

One approach to supporting all three "listBy..." methods would be to have one copy of the directory's file entries—the myFiles array—and to sort the file entries in the array into an appropriate sequence at each call. The figure below shows an example of this approach.



Another way, which trades memory efficiency for time efficiency in the case where the directory doesn't change very often, is to have a separate array for each list order, as shown below.



Those of you with experience using data base programs may recognize this technique. Each entry in the data base typically contains a bunch of *fields*, and the data base program maintains *index arrays* that allow the entries to be listed by one field or another.

54.2.5 (Brainstorm) Implementing the "add" method

Briefly explain how to implement the add method that uses the myFilesByName, myFilesBySize, and myFilesByDate as just described.

54.2.6 (Display page) Stable sorting

Stable sorting

A sorting method is referred to as *stable* if it maintains the relative order of entries with equal keys. Consider, for example, the following set of file entries:

```
    data.txt    10003   Sept 3, 2005
    hw2.java     1596   Sept 5, 2005
    hw2.class    4100   Sept 5, 2005
```

Suppose these entries were to be sorted by date. The entries for hw2.java and hw2.class have the same date; a stable sorting method would keep hw2.java before hw2.class in the resulting ordering, while an unstable sorting method might switch them.

54.2.7 (Brainstorm) Stability of selection sort?

Here is the code for a version of selection sort that sorts an array of file entries into chronological order using the date of a file entry as the key for sorting. The earlier method returns true exactly when its first argument date is earlier than its second.

```
public static void selectionSort(FileEntry list[]) {
  for (int j=list.length-1; j>0; j--) {
    int latestPos = 0;
    for (int k=1; k<=j; k++) {
      if (earlier (list[latestPos], list[k])) {
        latestPos = k;
      }
    }
    if (j != latestPos) {
      FileEntry temp = list[j];
      list[j] = list[latestPos];
      list[latestPos] = temp;
    }
  }
}
```

Is this sorting method stable? Explain why or why not.

54.2.8 (Brainstorm) Stability of insertion sort?

Here is code for a version of insertion sort that sorts file entries into chronological order using dates as keys as in the previous example. Again, the earlier method returns true exactly when its first argument date is earlier than its second.

```
public static void insertionSort (FileEntry list[]) {
```

```
  for (int j=1; j<list.length; j++) {
    FileEntry temp = list[j];
    int k = j;
    while( k > 0 && earlier (temp, list[k-1]) ) {
      list[k] = list[k-1];
      k--;
    }
    list[k] = temp;
  }
}
```

Is this sorting method stable? Explain why or why not.

54.2.9 (Display page) Linear-time sorting with radix sort

Linear-time sorting

All the sorting methods we've seen so far are comparison-based, that is, they use comparisons to determine whether two elements are out of order. One can prove that any comparison-based sort needs *at least* O(N log N) comparisons to sort N elements. However, there are sorting methods that don't depend on comparisons that allow sorting of N elements in time proportional to N! A description of one such algorithm, radix exchange sort, follows. The radix of a number system is the number of values a single digit can take one. Binary numbers form a radix-2 system; decimal notation is radix 10. Any radix sort examines elements in passes, one pass for (say) the rightmost digit, one for the next-to-rightmost digit, and so on. Radix exchange sort is used to sort elements of an array. Here's how it works.

1. You look at the array elements one digit at a time, starting with the high-order (most significant, leftmost) digit. You go through the array adding elements to "buckets", one bucket for each possible digit value (i.e. bucket[0], bucket[1], ..., bucket[9]). Now you've divided the array into ten pieces, bucket[0] to bucket[9], although the order of numbers within each piece is unknown.
2. Copy the values back to the array, keeping track of the boundaries between the values in the various buckets. Now you recursively sort each of the ten buckets, only you do the division based on the next highest digit.
3. When you're down to the low-order digit, you're finished.

The number of passes through the array is the number of digits in the largest value. If this is known already, it becomes a constant, and the time for the whole sorting process is proportional to that constant times the number of array elements. This algorithm, along with several others, is described in the Wikipedia entry for radix sort. Another radix sort that processes digits in the opposite order A nice demo of this algorithm is available online. Why not use radix sort all the time? Fast performance requires a large number of keys that aren't too long, for which it is relatively straightforward to identify "digits".

54.3 Homework*(1 step)*

54.3.1 (Display page) Project checkoffs

Project checkoffs

In the next lab, you will be asked to defend your hash function for trays. This involves answering three questions:

- Do equal trays have the same hash code?
- Is your hash function computable quickly?
- Does it mix up the trays sufficiently?

The InstrumentedHashMap class that you used in lab may be helpful in generating evidence that your hash function is minimizing collisions. You will also be asked to demo your program on the medium puzzles.

55 Quiz 27	*(1 activity)*

55.1 (Quiz) Review sorting.*(1 step)*

55.1.1 (Student Assessment) Quiz questions

1. Someone has asked you to sort a linked list of elements that are almost in order. Describe a method you might use and a method you should not use. Explain your choices briefly, giving estimates for the number of comparisons each of the methods you listed would make.