GIUSEPPE TURINI

CS-203 COMPUTING AND ALGORITHMS 3
LESSON 06
TRANSFORM-AND-CONQUER

INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS (3RD EDITION), **CHAP 6, 201-252**.

HIGHLIGHTS

Introduction to Transform-and-Conquer
Presorting
Gaussian Elimination
LU Decomposition, Matrix Inverse, and Matrix Determinant
Balanced Search Trees
AVL Trees, and 2-3 Trees
Heaps and Heapsort
Horner's Rule and Binary Exponentiation
Problem Reduction
Computing the Least Common Multiple, and Counting Paths in a Graph
Reduction of Optimization Problems, and Linear Programming
Reduction to Graph Problems

STUDY GUIDE

Study Material

- This slides.
- "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 6, pp. 201-252.

Selected Exercises

- Exercises: 6.1.1-6.1.4, 6.1.7-6.1.8, 6.2.1-6.2.3, 6.2.6-6.2.7, 6.2.9, 6.3.1-6.3.10, 6.4.1-6.4.11, 6.5.1-6.5.10, 6.6.2, 6.6.5-6.6.6.

Additional Resources

- visualgo.net/en

INTRODUCTION TO TRANSFORM-AND-CONQUER 1

Now we discuss about design methods based on the idea of transformation.

This general technique is called **transform-and-conquer** because it works as a **two-stage procedure**:

1. in the **transformation stage**, the instance of the problem is modified to be, for one reason or another, easier to solve; then

2. in the **conquering stage**, the instance of the problem is solved.

INTRODUCTION TO TRANSFORM-AND-CONQUER 2

There are 3 variations of this idea that differ by what we transform a given instance to:

- **instance simplification:** the transformation to a simpler or more convenient instance of the same problem;
- **representation change:** the transformation to a different representation of the same instance;
- **problem reduction:** the transformation to an instance of a different problem for which an algorithm is already available.

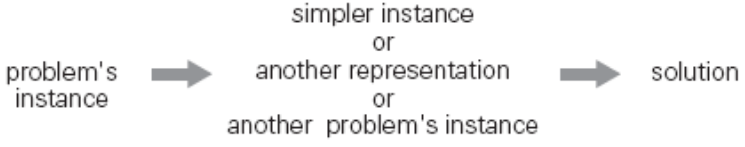


FIGURE 6.1 Transform-and-conquer strategy.

PRESORTING 1

The idea of **presorting** is to simplify a problem (**instance simplification**) considering that many algorithmic problems are easier to solve if their input is sorted.

Of course, the benefits of sorting should more than compensate for the time spent on it; otherwise, we would better off dealing with an unsorted input directly.

Obviously, the time efficiency of algorithms that involve sorting may depend on the efficiency of the sorting algorithm being used. So far we have discussed:

- **3 elementary sorting algorithms:** selection sort, bubble sort, and insertion sort, that are quadrating in the worst-case and average-case; and

- **2 advanced sorting algorithms:** mergesort (always in $\Theta(n \log n)$), and quicksort ($\Theta(n \log n)$ in average-case, and quadratic in worst-case).

PRESORTING 2

Example: Suppose we need to **check if an array contains only unique elements**.

We have already seen a **brute-force** approach to solve this problem which consisted in comparing pairs of the array elements until either 2 equal elements were found or no more pairs were left. This algorithm is quadratic in the worst-case.

Alternatively, we can sort the array first, and then check only its consecutive elements: if the array has equal elements, they must be next to each other, and viceversa.

```
boolean presortCheckElementUniqueness( int[] A ) {
   mergesort(A);
   for( int i = 0; i < A.length - 1; i++ ) {
      if( A[i] == A[i+1] ) { return false; } }
   return true;
}
```

PRESORTING 3

Example (continued): The running time of this algorithm is the sum of the time spent on sorting and the time spent on checking consecutive elements:

1. the **sorting** by mergesort requires **n log n** comparisons, and

2. the **checking** of consecutive elements needs no more than **n-1** comparisons.

So, it is the sorting part that will determine the overall efficiency of the algorithm:

$$T(n) = T_{sorting}(n) + T_{checking}(n) \in \Theta(\, n \log n\,) + \Theta(n) = \Theta(\, n \log n\,)$$

PRESORTING 4

Example: Let us consider the problem of **computing a mode**. A **mode** is a value that occurs most often in a given list of numbers. If several different values occur most often, any of them can be considered a mode.

The **brute-force** approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.

In order to implement this idea, we can store the values already encountered, along with their frequencies, in a separate list. On each iteration, the $\mathbf{i^{th}}$ element of the original list is compared with the values already encountered by traversing this auxiliary list.

If a matching value is found, its frequency is incremented; otherwise, the current element is added to the list of distinct values seen so far with a frequency of **1**.

PRESORTING 5

Example (continued): It is easy to see that the **worst-case** input for this algorithm is a list with no equal elements. For such a list, its **i^{th}** element is compared with **i-1** elements of the auxiliary list of distinct values seen so far before being added to the list with a frequency of **1**.
As a result, the **worst-case number of comparisons C(n)** made by this algorithm in creating the frequency list is:

$$C_{worst}(n) = \sum_{i=1}^{n} (i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)\,n}{2} \in \Theta(n^2)$$

Note: The additional **n-1** comparisons needed to find the largest frequency in the auxiliary list do not change the quadratic worst-case efficiency class of this algorithm.

PRESORTING 6

Example (continued): As an alternative, let us use **presorting**. If we first sort the input, then all equal value will be adjacent to each other. So, to compute the mode, we need to find the longest sequence of adjacent equal values in the sorted array.

```
int presortComputingAMode( int[] A ) {
   mergesort(A);
   int i = 0; int modeFreq = 0; int modeValue = 0;
   while( i <= A.length-1 ) {
      int runNum = 1; int runVal = A[i];
      while( ( i+runNum <= A.length-1 ) && ( A[i+runNum] == runVal ) ) { runNum++; }
      if( runNum > modeFreq ) { modeFreq = runNum; modeValue = runVal; }
      i = i+runNum; }
   return modeValue;
}
```

In this case the time efficiency analysis is dominated by the time spent on sorting (**n log n**), since the remainder of the algorithm takes linear time (**n**).

PRESORTING 7

Example: Consider the **search for a given value v** in an array of **n** sortable items.

The **brute-force** solution (**sequential search**) needs **n** comparisons in **worst-case**.

If we use **presorting**, we can sort the array at first, and then we can apply the **binary search**, which requires only $\lfloor \log_2 n \rfloor + 1$ comparisons in the **worst-case**. So:

$$T_{worst}(n) = T_{sorting}(n) + T_{searching}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$$

This worst-case efficiency is inferior to the efficiency of the sequential search, so **in this case the presorting is not useful**.

Note: If we need to perform multiple searches on the array, the time spent on sorting might well be justified!

GAUSSIAN ELIMINATION 1

Here we introduce one of the most important algorithms in applied mathematics: **Gaussian elimination**. This algorithm solves a system of linear equations by first transforming it to another system (**representation change**) with a special property that makes finding a solution quite easy.

The following is a **system of 2 linear equations in 2 unknowns**:

$$\begin{cases} a_{11}\,x + a_{12}\,y = b_1 \\ a_{21}\,x + a_{22}\,y = b_2 \end{cases}$$

Note: Unless the coefficients (e.g. $\mathbf{a_{11}}$ and $\mathbf{a_{12}}$) of one equation are proportional to the coefficients of the other equation (e.g. $\mathbf{a_{21}}$ and $\mathbf{a_{22}}$), the system has a unique solution.

GAUSSIAN ELIMINATION 2

The **standard method** to find the unique solution of a system of linear equations, is:

1. use either equation to express the 1st variable as a function of the other; and then
2. substitute the result into the other equation, yielding another linear equation;
3. solve this new linear equation to find the value of the 2nd variable.

In many applications, we need to solve a **system of n equations in n unknowns**:

$$\begin{cases} a_{11}\,x_1 + a_{12}\,x_2 + \cdots + a_{1n}\,x_n = b_1 \\ a_{21}\,x_1 + a_{22}\,x_2 + \cdots + a_{2n}\,x_n = b_2 \\ \vdots \\ a_{n1}\,x_1 + a_{n2}\,x_2 + \cdots + a_{nn}\,x_n = b_n \end{cases}$$

where **n** is a large number.

GAUSSIAN ELIMINATION 3

Theoretically, we can solve such a system by **generalizing the substitution method** for solving systems of 2 linear equations; however, the resulting algorithm would be extremely cumbersome.

Question: What algorithm design technique would such a method be based upon?
Answer: Decrease and conquer (decrease-by-one).

Fortunately, there is a much more elegant algorithm for solving systems of linear equations called **Gaussian elimination**.

The idea of Gaussian elimination is to transform a system of **n** linear equations in **n** unknowns to an equivalent system (i.e. a system with the same solution as the original one) with an **upper-triangular coefficient matrix**, a matrix with all 0s below its main diagonal.

GAUSSIAN ELIMINATION 4

From a **system of n linear equations in n unknowns** to an equivalent system:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases} \Rightarrow \begin{cases} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n = b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n = b'_2 \\ \vdots \\ a'_{nn}x_n = b'_n \end{cases}$$

In matrix notation, we can write this as: $Ax = b \quad \Rightarrow \quad A'x = b'$ where

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \quad A' = \begin{bmatrix} a'_{11} & \cdots & a'_{1n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & a'_{nn} \end{bmatrix} \quad b' = \begin{bmatrix} b'_1 \\ \vdots \\ b'_n \end{bmatrix}$$

GAUSSIAN ELIMINATION 5

Question: Why is the system with the upper-triangular coefficient matrix better than a system with an arbitrary coefficient matrix?
Answer: Because we can easily solve the system with an upper-triangular coefficient matrix by back substitutions as follows:

1. first, we can immediately find the value of $\mathbf{x_n}$ from the last linear equation;

2. then, we can substitute this value into the next to last linear equation to get $\mathbf{x_{n-1}}$;

3. and so on, until we substitute the known values of the last **n-1** variables into the **1st** linear equation, from which we find the value of $\mathbf{x_1}$.

So, how we can get from a system with an arbitrary coefficient matrix A to an equivalent system with an upper-triangular coefficient matrix A'?

GAUSSIAN ELIMINATION 6

We can transform a system with an arbitrary coefficient matrix **A** to an equivalent system with an upper-triangular coefficient matrix **A'**, through a series of the so-called **elementary operations**:

- **exchanging 2 linear equations** of the system;
- **replacing a linear equation** with its non-zero multiple;
- **replacing a linear equation** with a sum or difference of this linear equation and some multiple of another linear equation.

Since no elementary operation can change a solution to a system, **any system that is obtained through a series of elementary operations will have the same solution as the original system**.

GAUSSIAN ELIMINATION 7

Let us see how we can get to a system with an upper-triangular matrix:

1. First we use $\mathbf{a_{11}}$ as **pivot** to make all $\mathbf{x_1}$ coefficients **0**s in the linear equations below the 1st one. To do this, we replace the 2nd linear equation with the difference between it and the 1st linear equation multiplied by $\mathbf{a_{21}/a_{11}}$ to get a linear equation with a **0** coefficient for $\mathbf{x_1}$.
2. We do the same for all the other linear equations (with multiples $\mathbf{a_{31}/a_{11}}$, $\mathbf{a_{41}/a_{11}}$, ... , $\mathbf{a_{n1}/a_{11}}$ of the 1st linear equation, respectively) making all the coefficients of $\mathbf{x_1}$ below the 1st linear equation equal to **0**.
3. Then we set to **0** all the coefficients of $\mathbf{x_2}$ by subtracting an appropriate multiple of the 2nd linear equation from each of the linear equations below the 2nd one.
4. Repeating this elimination for each of the first **n-1** variables ultimately yields a system with an upper-triangular coefficient matrix **A'**.

GAUSSIAN ELIMINATION 8

Note: We can perform the Gaussian elimination **augmenting the coefficient matrix** of a system, adding another column (i.e. the **n+1st** column) with the right-hand side values (i.e. the **constant terms**) of the linear equations. In this way, we don't need to write explicitly the variable names and the addition/subtraction signs.

Example: We can solve the following system by Gaussian elimination:

$$\begin{cases} 2\,x_1 - x_2 + x_3 = 1 \\ 4\,x_1 + x_2 - x_3 = 5 \\ x_1 + x_2 + x_3 = 0 \end{cases} \Rightarrow \begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

system of linear equations augmented coefficient matrix

GAUSSIAN ELIMINATION 9

Example (continued): These are the elementary operations to transform the augmented coefficient matrix into an upper-triangular augmented coefficient matrix:

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix} \Rightarrow \begin{matrix} 1^{\text{st}}\ \text{row} \\ 2^{\text{nd}}\ \text{row} - 4/2\ \ 1^{\text{st}}\ \text{row} \\ 3^{\text{rd}}\ \text{row} - 1/2\ \ 1^{\text{st}}\ \text{row} \end{matrix} \Rightarrow \begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 3/2 & 1/2 & -1/2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 3/2 & 1/2 & -1/2 \end{bmatrix} \Rightarrow \begin{matrix} 1^{\text{st}}\ \text{row} \\ 2^{\text{nd}}\ \text{row} \\ 3^{\text{rd}}\ \text{row} - 1/2\ \ 2^{\text{nd}}\ \text{row} \end{matrix} \Rightarrow \begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Now we can obtain the solution by back substitutions:

$$x_3 = -2/2 = -1, \quad x_2 = (3 - (-3)x_3)/3 = 0, \quad x_1 = (1 - x_3 - (-1)x_2)/2 = 1.$$

GAUSSIAN ELIMINATION 10

The following is the Java implementation of the 1st stage, called **forward elimination**, of the Gaussian elimination algorithm:

```
float[][] forwardElimination( float[][] A, float[] b ) {
   float AFrom[][] = augmentMatrix( A, b ); // Merge A and b into augmented coeff matrix.
   float ATo[][] = copyMatrix(AFrom); // Warning: auxiliary matrix to store temp results!
   for( int i = 0; i < A.length-1; i++ ) {
      for( int j = i+1; j < A.length; j++ ) {
         for( int k = i; k < A.length+1; k++ ) {
            ATo[j][k] = AFrom[j][k] - AFrom[i][k] * AFrom[j][i] / AFrom[i][i]; } }
      AFrom = copyMatrix(ATo); } // Warning: we need to swap the matrices!
   return ATo;
}
```

GAUSSIAN ELIMINATION 11

Note: If **AFrom[i][i]=0**, we cannot perform the division, and in such a case we should use the 1st elementary operation to **exchange** the i^{th} row with some row below it that has a non-zero coefficient in the i^{th} column.

Note: Another potential difficulty is the possibility that **AFrom[i][i]** is so small and consequently **AFrom[j][i]/AFrom[i][i]** so large that **ATo[j][k]** might be distorted by a **round-off error** (i.e. subtraction of 2 numbers of greatly different magnitudes).

To avoid this issue, we can look for a row with largest absolute value of coefficient in column **i**, exchange it with row **i**, and then use the **AFrom [i][i]** as pivot of iteration **i**.

This method, called **partial pivoting**, ensures that the magnitude of the scaling factor will never exceed 1.

GAUSSIAN ELIMINATION 12

The following is the Java implementation of the **forward elimination using partial pivoting** for the Gaussian elimination algorithm:

```
float[][] forwardEliminationWithPartialPivoting( float[][] A, float[] b ) {
   float AFrom[][] = augmentMatrix( A, b ); // Merge A and b into augmented coeff matrix.
   float ATo[][] = copyMatrix(AFrom); // Warning: auxiliary matrix to store temp results!
   for( int i = 0; i < A.length-1; i++ ) {
      int pivotRow = i;
      for( int j = i+1; j < A.length; j++ ) {
         if( Math.abs(AFrom[j][i]) > Math.abs(AFrom[pivotRow][i]) ) { pivotRow = j; } }
      for( int k = i; k < A.length+1; k++ ) {
         // Swap AFrom[i][k] with AFrom[pivotRow][k].
         float tmp=AFrom[i][k]; AFrom[i][k]=AFrom[pivotRow][k]; AFrom[pivotRow][k]= tmp;}
      for( int j = i+1; j < A.length; j++ ) {
         float tmp = AFrom[j][i] / AFrom[i][i]; // Warning: compute div once per row!
         for( int k = i; k < A.length+1; k++ ) { ATo[j][k]=AFrom[j][k]-AFrom[i][k]*tmp;}}
      AFrom = copyMatrix(ATo); } // Warning: we need to swap the matrices!
   return ATo;
}
```

GAUSSIAN ELIMINATION 13

Let us find the time efficiency of the **forward elimination using partial pivoting** for the Gaussian elimination.

The innermost loop of this algorithm consists of a single line:

```
ATo[j][k] = AFrom[j][k] – AFrom[i][k] * tmp;
```

which contains 1 multiplication and 1 subtraction. We can choose the multiplication as the basic operation, obtaining:

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1$$

GAUSSIAN ELIMINATION 14

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{k=i}^{n+1} 1 = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (\, n+1-i+1\,) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (\, n+2-i\,) =$$

$$= \sum_{i=1}^{n-1} (\, n+2-i\,)(\, n-(\, i+1\,)+1\,) = \sum_{i=1}^{n-1} (\, n+2-i\,)(\, n-i\,) =$$

$$= (\, n+1\,)(\, n-1\,) + n\,(\, n-2\,) + \cdots + 3 \cdot 1 =$$

$$= \sum_{a=1}^{n-1} (\, a+2\,)\, a = \sum_{a=1}^{n-1} a^2 + \sum_{a=1}^{n-1} 2a =$$

$$= \frac{(n-1)\, n\, (2n-1)}{6} + 2\,\frac{(n-1)\, n}{2} = \frac{n\,(n-1)(2n+5)}{6} \approx \frac{n^3}{3} \in \Theta(n^3)$$

GAUSSIAN ELIMINATION 15

Since the 2nd stage (i.e. **back substitution**) of Gaussian elimination is in $\Theta(n^2)$, the running time is dominated by the cubic elimination stage, making the entire algorithm cubic as well.

Theoretically, Gaussian elimination always yields an exact solution to a system of linear equations when the system has a unique solution, or discovers that no such solution exists (i.e. either no solutions or infinitely many of them).

Example: Examples of application of the Gaussian elimination are:

- **LU decomposition** (aka **lower-upper decomposition** or **LU factorization**);
- computing a **matrix inverse**;
- computing a **matrix determinant**.

BALANCED SEARCH TREES 1

We have already discussed about how we can use a binary search tree to implement a dictionary. This transformation from a **set** of words, to a **binary search tree (BST)** storing the same words is an example of a **representation change** technique.

Question: What do we gain by such transformation compared to the straightforward implementation of a dictionary by using an array?
Answer: We gain in the time efficiency of searching, insertion, and deletion, which are all in **Θ(log n)** in the average-case (in the worst-case all these operations are in **Θ(n)** since the BST can be unbalanced)

The next data structures **preserve good properties of BST** (sorted items, and logarithmic efficiency of insert/delete/search) but have **better worst-case efficiency**:

BALANCED SEARCH TREES 2

- The 1st approach is an **instance simplification**: an unbalanced BST is transformed into a balanced BST (a **self-balancing BST**). Implementations of self-balancing BST differ by their definition of **balance**. For example an **AVL tree** (or **Adelson-Velsky & Landis tree**) defines balance as the heights of left and right subtrees of every node never exceeds 1, whereas **red-black trees** tolerate the height of a subtree being twice as large as the other subtree of the same node. If an insertion or deletion of a new node creates a tree with a violated balance requirement, the BST is restructured by a transformation in a family of special transformations called **rotations** that restore the balance required.

- The 2nd approach is a representation change: we allow more than 1 element to be stored in a node of a BST. Specific cases of such trees are **2-3 trees**, **2-3-4 trees**, and **B-trees**. They all differ in the number of elements admissible in a single node of a BST, but all are perfectly balanced.

BALANCED SEARCH TREES 3

BALANCED SEARCH TREES: AVL TREES A

Definition: An **AVL tree** is a **binary search tree** in which the **balance factor** of every node (defined as the difference between the heights of the left and right subtrees of a node) is either **0** or **+1** or **-1**. (The height of the empty tree is defined as **-1**).

Example: The BST in the figure **(a)** is an AVL tree, but the BST in figure **(b)** is not.

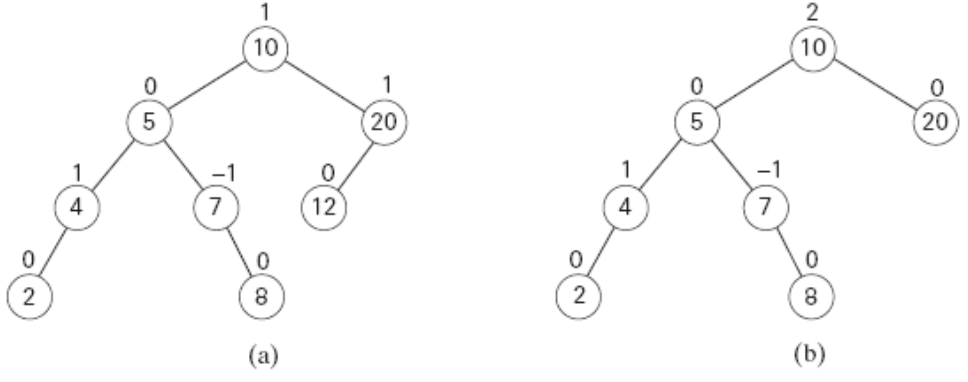


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

BALANCED SEARCH TREES 4

BALANCED SEARCH TREES: AVL TREES B

If an insertion of a new node makes an AVL tree unbalanced, we "rotate" the AVL tree.

A **rotation** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either **+2** or **-2**. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

There are only 4 types of rotations: **R**, **L**, **LR**, and **RL**.

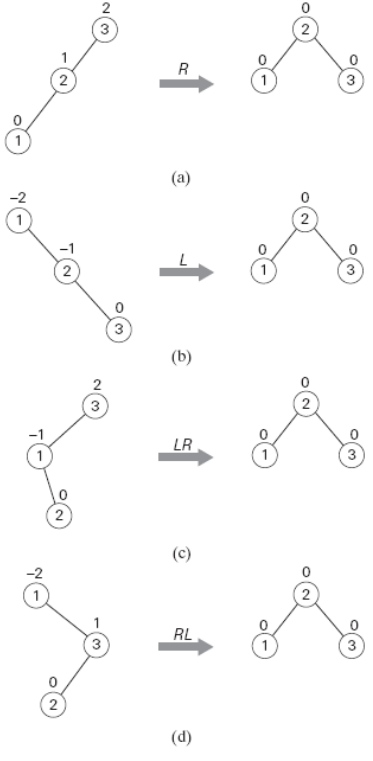


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

BALANCED SEARCH TREES 5

BALANCED SEARCH TREES: AVL TREES C

The 1st rotation type is the **single right rotation**, or **R-rotation**. Rotate the edge connecting the root **r** and its left child to the right (see figure **(a)**).
Note: An **R-rotation** is performed after a new key is inserted into the **left subtree of the left child** of a tree whose root had balance **+1** before the insertion.

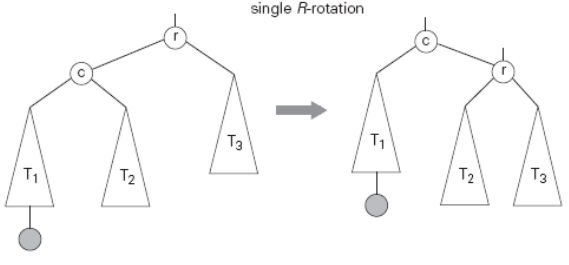


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

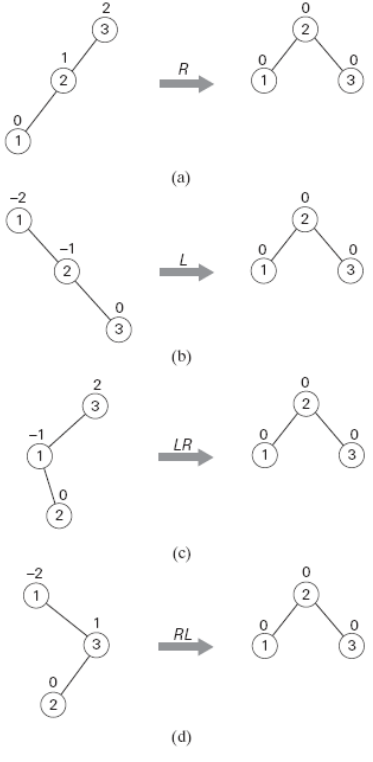


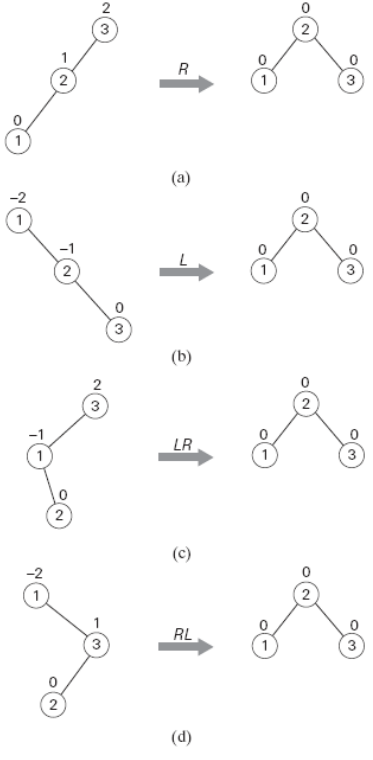
FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

BALANCED SEARCH TREES 6

BALANCED SEARCH TREES: AVL TREES D

The 2nd rotation type is called **single left rotation**, or **L-rotation**. This rotation is the mirror image of an R-rotation (see figure **(b)**).

Note: An **L-rotation** is performed after a new key is inserted into the **right subtree of the right child** of a tree whose root had the balance of **-1** before the insertion.



IGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

BALANCED SEARCH TREES 7

BALANCED SEARCH TREES: AVL TREES E

The 3rd rotation type is the **double left-right rotation**, or **LR-rotation**: an L-rotation of the left subtree of root **r** an R-rotation of the new tree rooted at **r** (see figure **(c)**). **Note:** An **LR-rotation** is performed after a new key is inserted into the **right subtree of the left child** of a tree whose root had balance **+1** before the insertion.

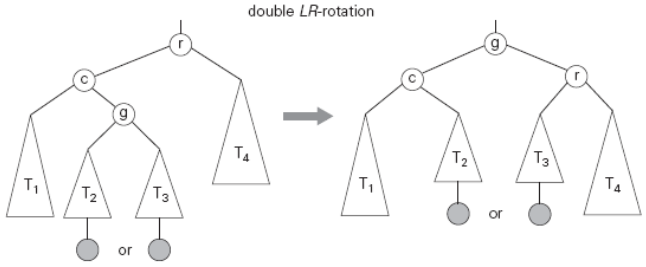


FIGURE 6.5 General form of the double *LR*-rotation in the AVL tree. A shaded node is the last one inserted. It can be either in the left subtree or in the right subtree of the root's grandchild.

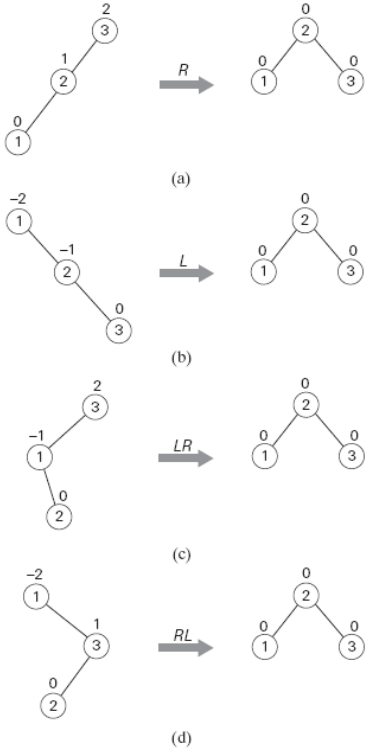


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

BALANCED SEARCH TREES 8

BALANCED SEARCH TREES: AVL TREES F

The 4th rotation type is the **double right-left rotation**, or **RL-rotation**. This rotation is a combination of 2 rotations: an R-rotation of the right subtree of root **r**, followed by an L-rotation of the new tree rooted at **r** (see figure **(d)**)

Note: An **RL-rotation** is performed after a new key is inserted into the **left subtree of the right child** of a tree whose root had the balance of **-1** before the insertion.

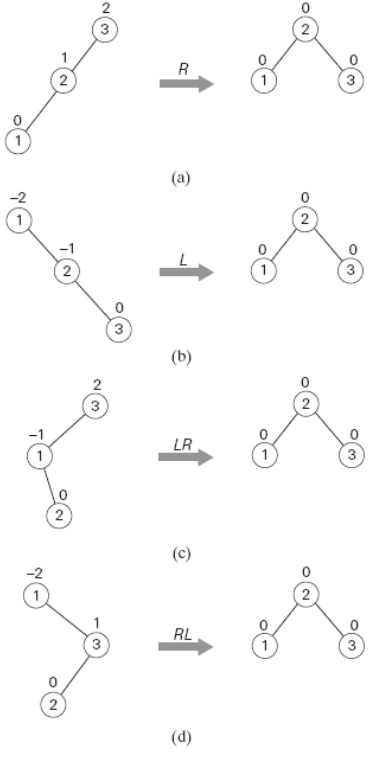


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

BALANCED SEARCH TREES: AVL TREES G

Note: These **rotations** are not trivial transformations, but luckily they can be done in **constant time**. Not only they should guarantee that a resulting tree is balanced, but they should also preserve the basic requirements of a binary search tree.

Example: In the left BST in figure, all keys of subtree $\mathbf{T_1}$ are < than **c**, which is < than all keys of subtree $\mathbf{T_2}$, which are < than **r**, which is < than all keys of subtree $\mathbf{T_3}$. The same relationships among keys hold for the BST after the rotation.

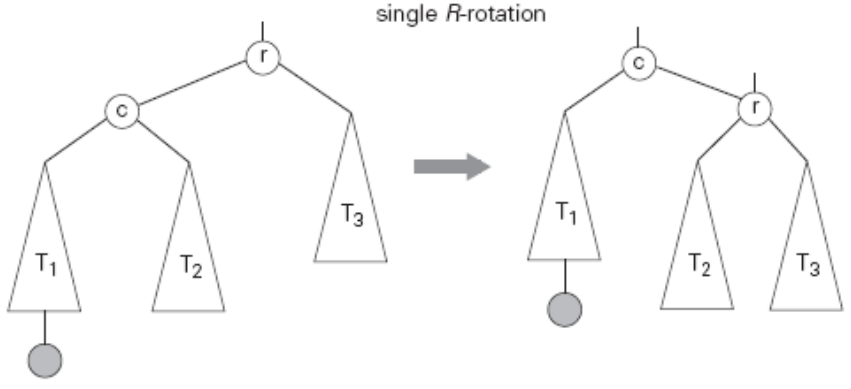


FIGURE 6.4 General form of the *R*-rotation in the AVL tree. A shaded node is the last one inserted.

BALANCED SEARCH TREES

BALANCED SEARCH TREES: AVL TREES

Example: In figure you can see the construction of an AVL tree for the list **{ 5, 6, 8, 3, 2, 4, 7 }** by successive insertions.

Question: How efficient are AVL trees?

Answer: The key to analyze the efficiency is the height of the tree, that satisfied the following:

$$\lfloor \log_2 n \rfloor \leq h < 1.440 \log_2(n+2) - 1.327$$

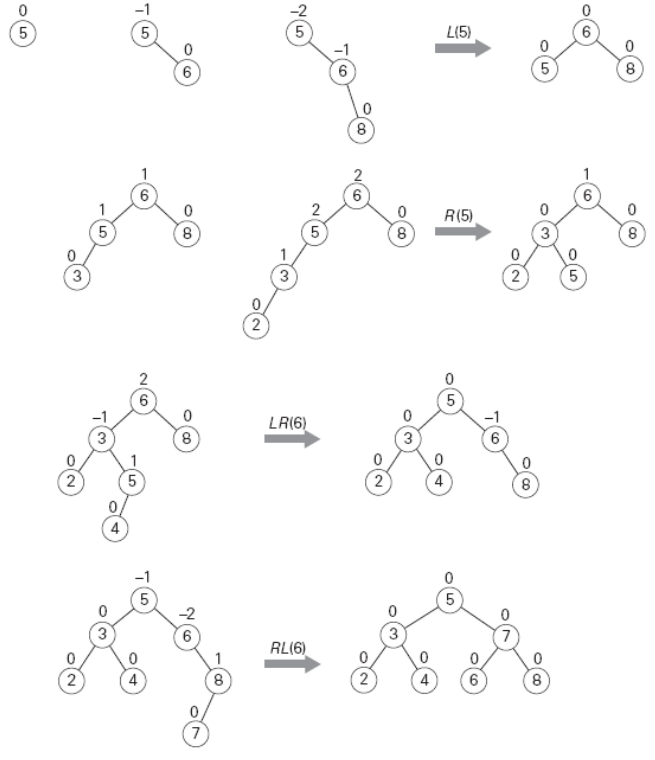


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

BALANCED SEARCH TREES: AVL TREES

The inequalities bounding the height **h** of the AVL tree, immediately imply that the operations of **searching** and **insertion** are **Θ(log n)** in the **worst-case**.

Experimental results have shown that in the **average-case** the height **h** of an AVL tree is about **1.01 log2 n + 0.1**. So, searching in an AVL tree requires, on average, the same number of comparisons as searching in a sorted array by binary search.

The operation of key **deletion** in an AVL tree is in the same efficiency class as insertion: i.e. logarithmic.

BALANCED SEARCH TREES 12

BALANCED SEARCH TREES: 2-3 TREES A

The second idea of balancing a search tree is to allow more than 1 key in the nodes of the search tree. The simplest implementation of this idea is 2-3 trees.

Definition: A **2-3 tree** is a tree that can have nodes of 2 types:

- a **2-node** is a node that contains a single key **K** and has 2 children: the left child serves as the root of a subtree whose keys are **< K**, and the right child serves as the root of a subtree whose keys are **> K** (as in a standard BST);
- a **3-nodes** is a node that contains 2 ordered keys $\mathbf{K_1}$ and $\mathbf{K_2}$ ($\mathbf{K_1 < K_2}$) and has 3 children: the leftmost child serves as the root of a subtree with keys $\mathbf{< K_1}$, the middle child serves as the root of a subtree with keys between $\mathbf{K_1}$ and $\mathbf{K_2}$, and the rightmost child serves as the root of a subtree with keys $\mathbf{> K_2}$.

Finally, all the **leaves** in a 2-3 tree must be **on the same level**.

BALANCED SEARCH TREES 13

BALANCED SEARCH TREES: 2-3 TREES B

Searching for a key **K** in a 2-3 tree is quite simple. We start the search at the root:

- if the root is a **2-node**, we act as if it were a BST;
- if the root is a **3-node**, we need 2 key comparisons to understand if we need to stop the search or in which of the 3 subtrees we should continue the search.

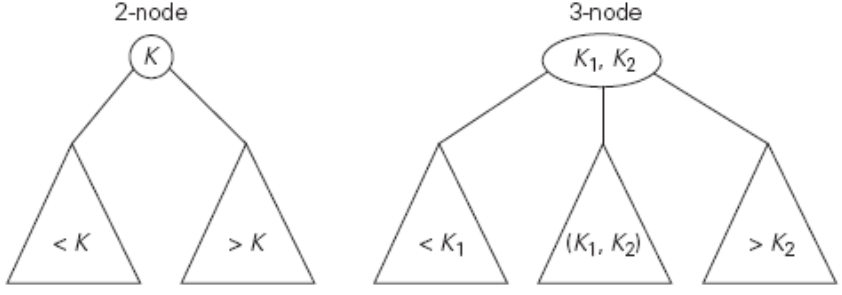


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

BALANCED SEARCH TREES 14

BALANCED SEARCH TREES: 2-3 TREES C

Inserting a new key **K** in a 2-3 tree is done as follows:

1. first, note that we always insert a new key **K** in a leaf (except for the empty tree);
2. the proper leaf to perform the insertion is found by performing a search for **K**:
    a. if the leaf is a **2-node**, we insert **K** there as either 1st or 2nd key of the leaf;
    b. if the leaf is a **3-node**, we split the leaf in 2:
        i. the **smallest** of the 3 keys is put in the 1st leaf,
        ii. the **largest** of the 3 keys is put in the 2nd leaf,
        iii. the middle key is promoted to the parent of the leaf (causing sometimes a node split chain).

BALANCED SEARCH TREES 15

BALANCED SEARCH TREES: 2-3 TREES D

Example: The construction of a 2-3 tree for list **{ 9, 5, 8, 3, 2, 4, 7 }** is given in figure.

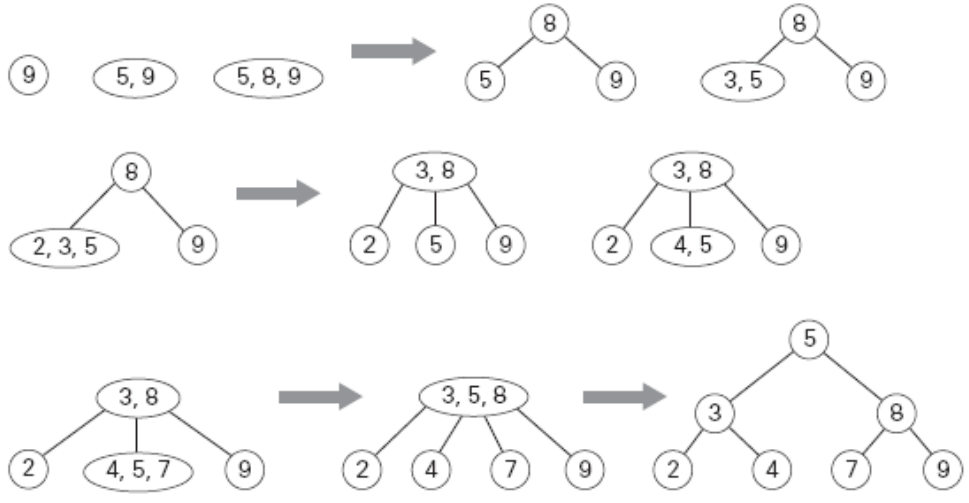


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

BALANCED SEARCH TREES 16

BALANCED SEARCH TREES: 2-3 TREES E

In order to analyze the efficiency of 2-3 trees, let us consider its height:

$$\underbrace{n}_{\text{nodes in a 23 tree}} \geq \underbrace{1 + 2 + \cdots + 2^h = 2^{h+1} - 1}_{\text{nodes in a 23 tree with only 2nodes}}$$

$$\Downarrow$$

$$h \leq \log_2(n+1) - 1 \quad \text{maximum height of 23 tree with n nodes}$$

$$\underbrace{n}_{\text{nodes in a 23 tree}} \leq \underbrace{2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2\left(3^{h+1} - 1\right)}_{\text{nodes in a 23 tree with only 3nodes}}$$

$$\Downarrow$$

$$h \geq \log_3\left(\frac{n}{2} + 1\right) - 1 \quad \text{minimum height of 23 tree with n nodes}$$

BALANCED SEARCH TREES 17

BALANCED SEARCH TREES: 2-3 TREES F

In summary in a 2-3 tree:

$$\log_3\left(\frac{n}{2}+1\right)-1 \leq h \leq \log_2(n+1)-1$$

So, the time efficiencies of searching, inserting, and deleting in 2-3 trees are all in **Θ(log n)** in both the **worst-case** and **average-case**.

HEAPS AND HEAPSORT 1

The data structure called **heap** is definitely not a disordered pile of items as the dictionary definition might suggest. Rather, it is a clever, **partially ordered data structure that is especially suitable for implementing priority queues**.

Definition: A **priority queue** is a multiset of items with an orderable characteristic called **item priority**, with the following operations:

- **finding** an item with the highest priority;
- **deleting** an item with the highest priority;
- **adding** a new item to the multiset.

It is primarily an efficient implementation of these operations that makes the heap both interesting and useful.

HEAPS AND HEAPSORT 2

Definition: A **heap** can be defined as a **binary tree** with keys assigned to its nodes, **1** key per node, provided the following two conditions are met:

1. The **shape property**: the binary tree is **essentially complete** (or simply **complete**), i.e. all its levels are full except possibly the last level, where only some rightmost leaves may be missing.
2. The **parental dominance** or **heap property**: the key in each node is greater or equal than the keys in its children. (This condition is considered automatically satisfied for all leaves.)

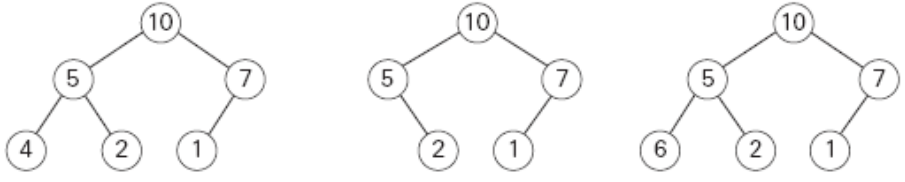


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

HEAPS AND HEAPSORT 3

Note: The **key values in a heap are ordered top down**: i.e. a sequence of values on any path from the root to a leaf is decreasing (non-increasing, if equal keys are allowed). However, **there is no left-to-right order of key values in a heap**.

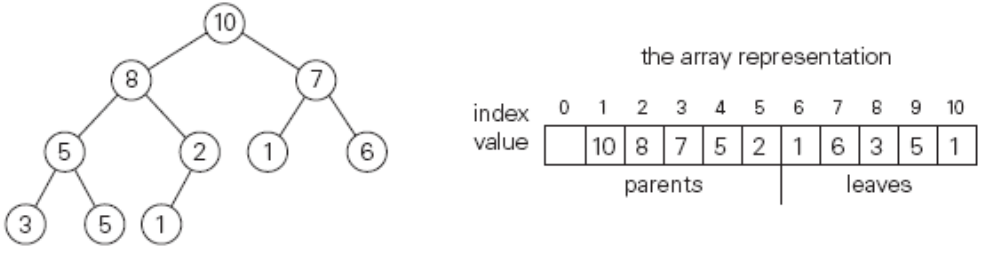


FIGURE 6.10 Heap and its array representation.

HEAPS AND HEAPSORT 4

The following are important properties of heaps:

1. There **exists exactly 1 essentially complete binary tree with n nodes**. Its height is equal to $\mathbf{\lfloor \log_2 n \rfloor}$.
2. The **root of a heap** always contains its **largest element**.
3. A **node of a heap** considered with all its descendants **is also a heap**.
4. A **heap can be implemented as an array** by recording its elements in the top-down, left-to-right fashion. It is convenient to store the elements of the heap in positions **1** through **n** of such array, leaving element in position **0** either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation:
    a. the parental node keys will be in the first $\mathbf{\lfloor n/2 \rfloor}$ positions of the array, while the leaf keys will occupy the last $\mathbf{\lfloor n/2 \rfloor}$ positions;
    b. if a key is stored in position **i** ($\mathbf{2 \le i \le \lfloor n/2 \rfloor}$), its children will be stored in position **2i** and **2i+1**, and its parent will be stored in position $\mathbf{\lfloor i/2 \rfloor}$.

HEAPS AND HEAPSORT 5

Thus, we could also define a heap as an array **H[1…n]** in which every element in position **i** in the **1st half** of the array is **≥** than the elements in position **2i** and **2i+1**:

$$H[i] \geq \max\{\, H[\,2i\,], H[\,2i+1\,]\,\}, \quad \text{for } i = 1 \ldots \lfloor n/2 \rfloor.$$

Note: While the ideas behind most of the algorithms dealing with heaps are easier to understand if we think of heaps as binary trees, their actual implementations are usually much simpler and more efficient using arrays.

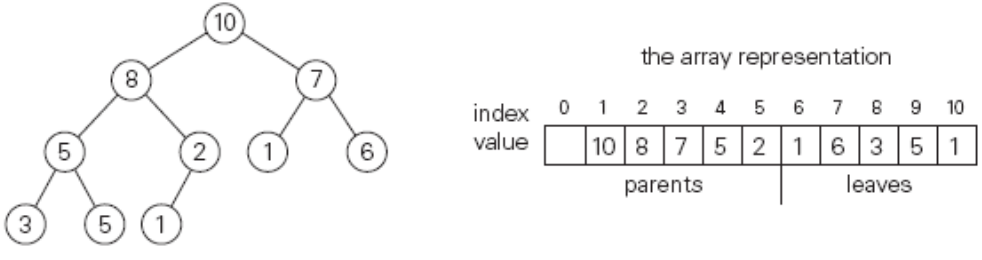


FIGURE 6.10 Heap and its array representation.

HEAPS AND HEAPSORT 6

How we can construct a heap for a given list of keys?

There are two principal alternatives for doing this:

- the **bottom-up heap construction** algorithm, and
- the **top-down heap construction** algorithm.

HEAPS AND HEAPSORT: BOTTOM-UP HEAP CONSTRUCTION A

The bottom-up heap construction algorithm initializes the essentially complete binary tree with **n** nodes by placing keys in the order given and then **"heapifies"** the tree.

HEAPS AND HEAPSORT 7

HEAPS AND HEAPSORT: BOTTOM-UP HEAP CONSTRUCTION B

The **"heapification"** of the binary tree is performed as follows:

1. Starting with the last parental node, the algorithm checks whether the **parental dominance holds for the key** in the current parental node:
   a. if it does not, the algorithm exchanges the key **K** of the current node with the larger key of its children, and checks if the parental dominance holds for **K** in its new position;
   b. this process continues until the parental dominance for **K** is satisfied;
   c. (eventually, it has to because it holds automatically for any key in a leaf).
2. After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the immediate predecessor of the current parental node.
3. The algorithm stops after this (**step 1**) is done for the root of the tree.

HEAPS AND HEAPSORT: BOTTOM-UP HEAP CONSTRUCTION C

```
void bottomUpHeapConstruction( int[] H ) {
   // Note: Heap element at position 0 (i.e. H[0]) is unused.
   for( int i = ( H.length - 1 ) / 2; i >= 1; i-- ) {
      int k = i; int v = H[k]; boolean heap = false;
      while( ( !heap ) && ( ( 2 * k ) <= ( H.length - 1 ) ) ) {
         int j = 2 * k;
         // There are 2 children?
         if( j < ( H.length - 1 ) ) {
            if( H[j] < H[j+1] ) {
               j++; } }
         if( v >= H[j] ) {
            heap = true; }
         else {
            H[k] = H[j];
            k = j; } }
      H[k] = v; }
}
```

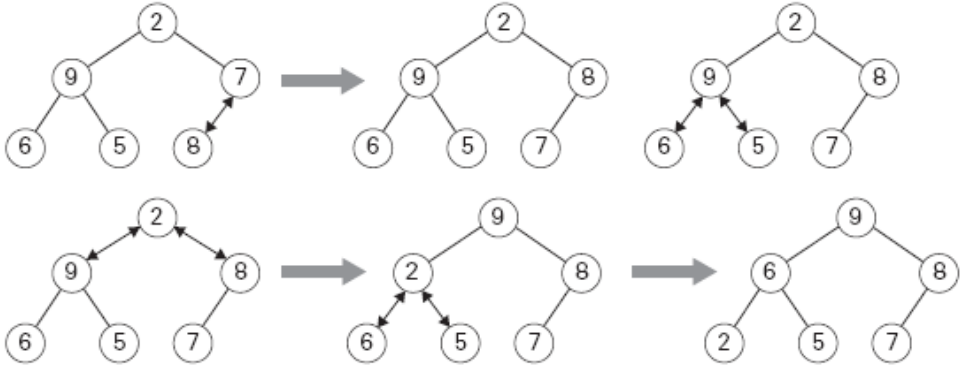


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

HEAPS AND HEAPSORT 9

HEAPS AND HEAPSORT: BOTTOM-UP HEAP CONSTRUCTION D

How efficient is this algorithm in the **worst-case**?

Assume, for simplicity, that $\mathbf{n=2^k-1}$ so that the binary tree of the heap is full: i.e. the largest possible number of nodes occurs on each level.

Let **h** be the height of the binary tree. According to the first property of heaps, $\mathbf{h = \lfloor \log_2 n \rfloor}$ (or $\mathbf{\lceil \log_2(n+1) \rceil - 1 = k-1}$ for the specific values of **n** we are considering).

In the worst-case, each key on level **i** of the tree will travel to the leaf level **h**.
Since moving to the next level down requires **2** comparisons (step **(a)**: **1** to find the largest child, and **1** to determine whether the exchange is required), the total number of key comparisons involving a key on level **i** will be **2(h-i)**.

HEAPS AND HEAPSORT: BOTTOM-UP HEAP CONSTRUCTION E

So, in the **worst-case**, the total number of key comparisons is:

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2\,(\,h-i\,) = \sum_{i=0}^{h-1} 2\,(\,h-i\,)\,2^i = 2\,(\,n - \log_2(\,n+1\,)\,)$$

Thus, with this bottom-up heap construction algorithm, a heap of size **n** can be constructed with fewer than **2n** comparisons.

HEAPS AND HEAPSORT 11

HEAPS AND HEAPSORT: TOP-DOWN HEAP CONSTRUCTION A

The **top-down heap construction** works by **successive insertions of a new key**:

1. First, attach a new key **K** after the last leaf of the existing heap, then:
    a. compare **K** with the key of its current parent:
        i. if the parent key **≥ K**, stop since the new binary tree is already a heap;
        ii. otherwise (i.e. parent key **< K**), swap these 2 keys and go to step (a);
        iii. this swapping continues until **K ≤** than its parent or **K** reaches the root.

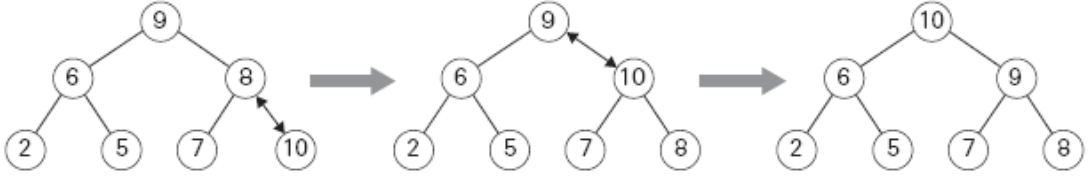


FIGURE 6.12 Inserting a key (10) into the heap constructed in Figure 6.11. The new key is sifted up via a swap with its parent until it is not larger than its parent (or is in the root).

HEAPS AND HEAPSORT: TOP-DOWN HEAP CONSTRUCTION B

Obviously, this **insertion operation** cannot require more key comparisons than the height of the heap. Since the height of a heap with **n** nodes is about $\log_2 n$, the time efficiency of insertion is in $O(\log_2 n)$.

HEAPS AND HEAPSORT: TOP-DOWN HEAP CONSTRUCTION C

The **removal** of a key needs to be implemented in different ways depending on the node to be deleted.

We consider here only the case of **deleting the root**:

1. exchange the root with the last key **K** of the heap;
2. decrease the heap size by **1** (i.e. removing the last element);
3. then **"heapify"** the smaller tree by moving **K** down the tree exactly in the same way we did it in the bottom-up heap construction algorithm:
    a. verify the parental dominance for **K**;
    b. if it holds, we are done;
    c. if it does not hold, swap **K** with the larger of its children and repeat the swap until the parental dominance condition holds for **K** in its new position.

HEAPS AND HEAPSORT 14

HEAPS AND HEAPSORT: TOP-DOWN HEAP CONSTRUCTION D

The efficiency of **deletion** is determined by the number of key comparisons needed to "heapify" the binary tree after the swap has been made and the size of the tree is decreased by 1.

Since this cannot require more key comparisons than twice the height of the heap, the time efficiency of deletion is in **O(log n)** as well.

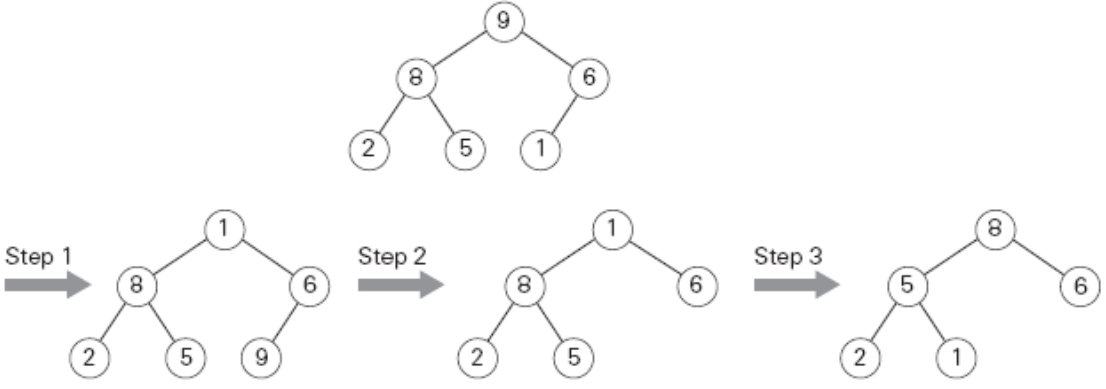


FIGURE 6.13 Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

HEAPS AND HEAPSORT 15

HEAPS AND HEAPSORT: HEAPSORT A

The following is the description of the **heapsort**: an interesting two-stage sorting algorithm that works as follows.

1. **heap construction:** construct a heap for a given array;

2. **maximum deletions:** delete the root **n-1** times from the remaining heap.

Note: The array elements are eliminated in decreasing order. But since under the array implementation of heaps an element being deleted is placed last, the resulting array will be exactly the original array sorted in increasing order.

HEAPS AND HEAPSORT 16

HEAPS AND HEAPSORT: HEAPSORT B

Example: Sorting array **{ 2, 9, 7, 6, 5, 8 }** by heapsort.

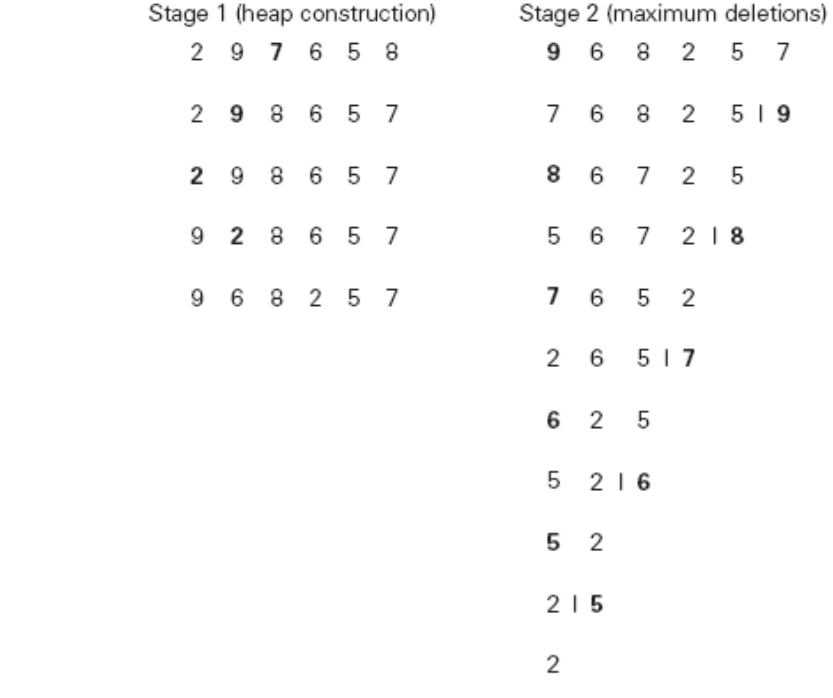


FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

HEAPS AND HEAPSORT

HEAPS AND HEAPSORT: HEAPSORT C

Since we already know that the **heap construction** stage of the algorithm is in **O(n)**, we have to investigate only the time efficiency of the **maximum deletions**.

For the number of key comparisons, **C(n)**, needed for eliminating the roots from the heaps of diminishing sizes from **n** to **2**, we have:

$$C(n) \leq 2\,\lfloor \log_2(n-1) \rfloor + 2\,\lfloor \log_2(n-2) \rfloor + \cdots + 2\,\lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i \leq$$

$$\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2\,(n-1)\log_2(n-1) \leq 2\,n \log_2 n$$

HEAPS AND HEAPSORT 18

HEAPS AND HEAPSORT: HEAPSORT D

This means that **C(n)** is in **O(n log n)** for the 2nd stage of heapsort.

For both stages, we get **O(n) + O(n log n) = O(n log n)**.

A more detailed analysis shows that heapsort is in **Θ(n log n)** in both the worst-case and average-case.

So, heapsort time efficiency is in the same class as that of mergesort, but heapsort is **in-place**.