L21: Divide and Conquer Problems

Integer multiplication

How fast we can multiply
Normally we require n^2 complexity to multiply two numbers
Multiply two n-bit integers I and J.
Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

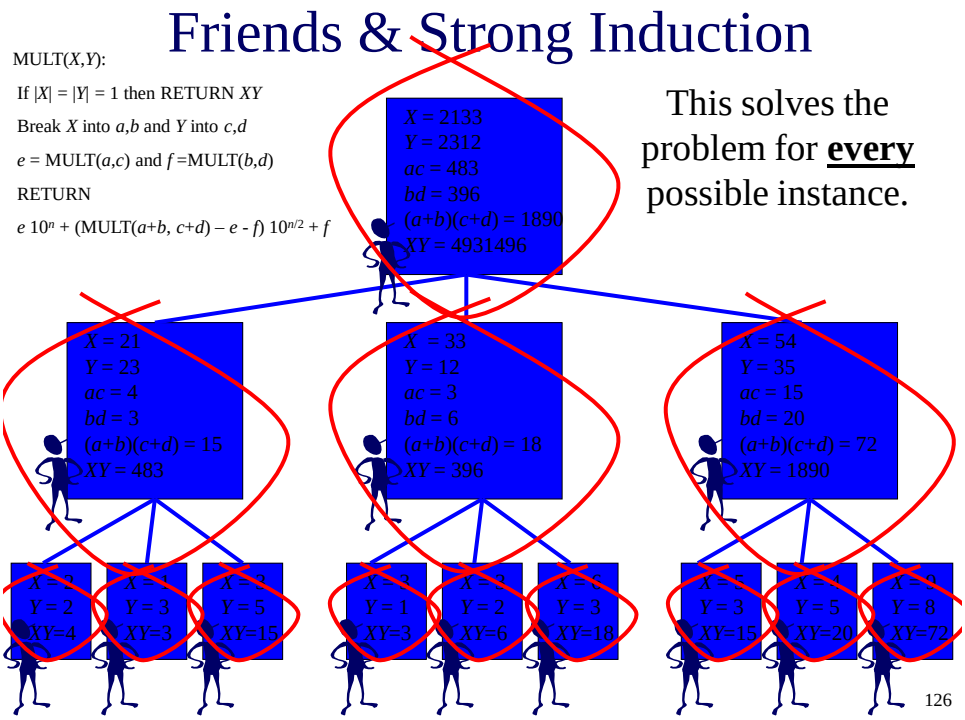
We can then define I*J by multiplying the parts and adding:

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

So, T(n) = 4T(n/2) + n, which implies T(n) is $O(n^2)$.
But that is no better

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$
$$I * J = I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$
$$= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$
$$= I_h J_h 2^n + (I_h J_l + I_l J_h) 2^{n/2} + I_l J_l$$

So, T(n) = 3T(n/2) + n, which implies T(n) is $O(n^{\log_2 3})$, by the Master Theorem.
Thus, T(n) is $O(n^{1.585})$.



Matrix Multiplication

Suppose we want to multiply two matrices of size *N* x *N*: for example *A* x *B* = *C*.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$C_{11} = a_{11}b_{11} + a_{12}b_{21}$
$C_{12} = a_{11}b_{12} + a_{12}b_{22}$
$C_{21} = a_{21}b_{11} + a_{22}b_{21}$
$C_{22} = a_{21}b_{12} + a_{22}b_{22}$
2x2 matrix multiplication can be accomplished in 8 multiplication.
$T(n)= 8T(n/2) + n \quad = (2^{\log_2 8} = 2^3)$
Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. .
$T(n) = 7T(n/2) + n \quad = (2^{\log_2 7} = 2^{2.807})$
This reduction can be done by Divide and Conquer Approach
Divide matrices into sub-matrices: A_0 , A_1, A_2 etc
Use blocked matrix multiply equations
Recursively multiply sub-matrices
Terminate recursion with a simple base case
P1 = (A11+ A22)(B11+B22)
P2 = (A21 + A22) * B11
P3 = A11 * (B12 - B22)
P4 = A22 * (B21 - B11)
P5 = (A11 + A12) * B22
P6 = (A21 - A11) * (B11 + B12)
P7 = (A12 - A22) * (B21 + B22)
C11 = P1 + P4 - P5 + P7
C12 = P3 + P5
C21 = P2 + P4
C22 = P1 + P3 - P2 + P6
$C_{11} = P_1 + P_4 - P_5 + P_7$
$= (A_{11}+ A_{22})(B_{11}+B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22}+(A_{12} - A_{22}) * (B_{21} + B_{22})$
$= A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} – A_{22} B_{11} -A_{11} B_{22} -A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} – A_{22} B_{21} – A_{22} B_{22}$
$= A_{11} B_{11} + A_{12} B_{21}$
Strassen beats normal matrix multiplication for n ≥ 25 approx.

Infinite Wall Problem

----------*----------||----------
You have an infinite wall on both sides where you are standing and it has a gate somewhere in one direction, you have to find out the gate,
no design-infinite time
incremental design- you go one step in one direction , come back go to other direction one step, come back and then go 2 steps in other direction
1+2.1+1
2+2.2+2
3+2.3+3
:
:
n-1+2.(n-1)+n-1
n+2n
$4\sum i+3n$
$4n(n-1)/2+3n=2n^2+n$

You go 2^0 step in one direction, come back go 1 step in other then 2^1direction in one way then come back and then up to 2^k

$2^0+2.2^0+2^0$

$2^1+2.2^1+2^1$

:

$2^{K-1}+2.2^{K-1}+2^{K-1}$

3.2^K

:

$4(2^{K-1}+2^{K-2}+\ldots\ldots1)+ 3.2^K$

$4(2^K-1)+ 3.2^K$

$7.2^K -4$

7N-4

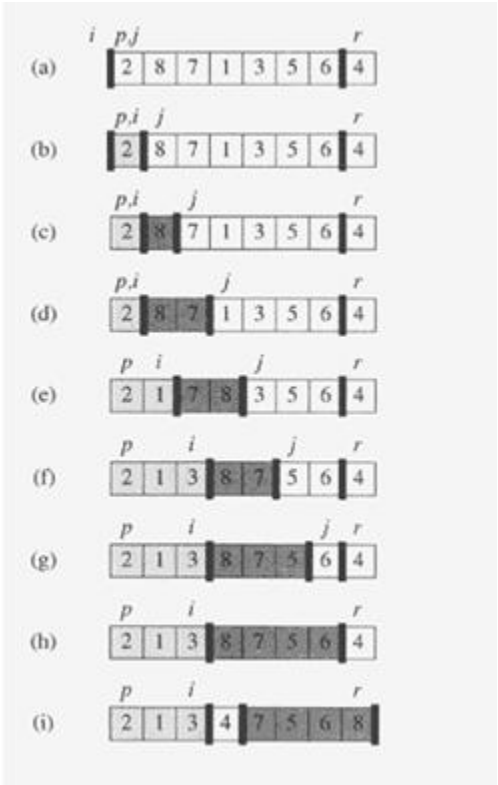
L22: Quick Sort

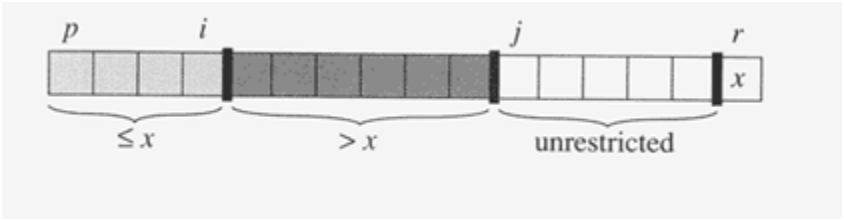
It is a divide and conquer algorithm which relies on a partition operation: we choose a pivot, move all smaller elements before it, and greater elements after it. Quick sort is an unstable and in place. It uses O(log *n*) additional space due to log n recursive calls. The most complex issue is choosing a good pivot .Poor choices of pivots can result in O(n^2) performance, but if at each step we choose the median as the pivot then it works in O(*n* log *n*). Finding the median, is an O(n) operation on unsorted lists. Quick sort make excellent usage of the memory hierarchy, taking perfect advantage of virtual memory and available caches. It can be easily parallelized due to its divide and conquer nature. Combine Step is trivial in Quick sort.

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Initial call is QUICKSORT(A, 1, n).

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```





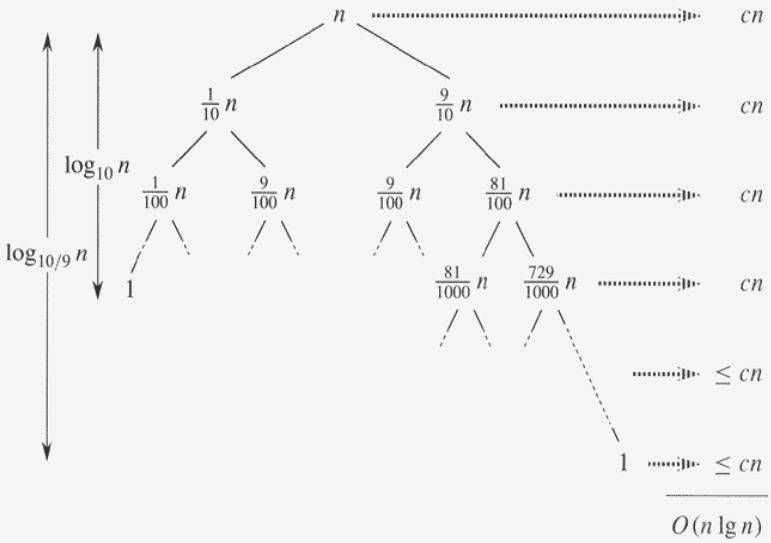


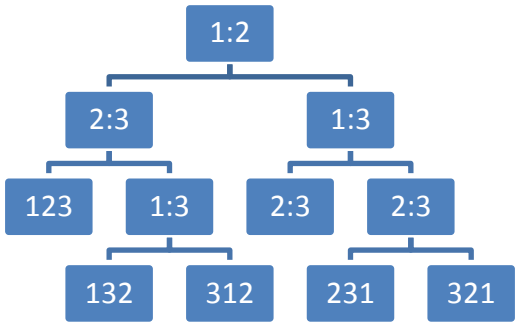
Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Quick sort complexity

T(n) = T(i-1) + T(n-i) +n
if (i=1) then
T(n) = T(n-1)+n
= O(n^2)
if i=n/2 then
T(n) = 2T(n/2) +n
= O(nlogn)

Lower bound on comparison sorting algorithms

Sort<a1,a2,a3>



Each leaf of the tree gives a permutation

Tree lists comparison along all possible instruction traces

Running time (#comparisons) = length of the path

Worst case run time = height of the tree

For n elements number of leaves will be n!

A tree with height h can have maximum 2^h leaves.

$n! = 2^h$

taking log on both sides

log n! = h

$h=\lg(n/e)^n$ (By Stirling formula)

h= nlg(n/e)

h= n(lgn-lge)

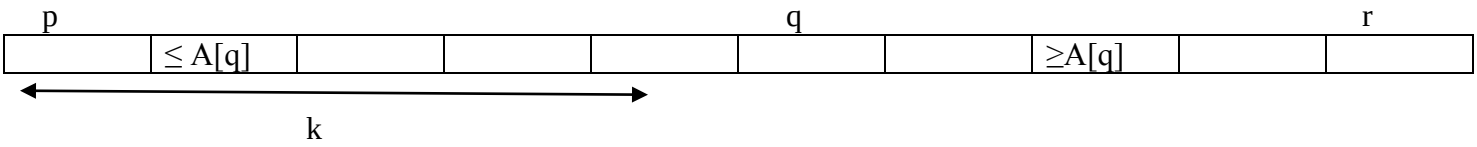
$= \Omega$ nlgn

Merge sort & heap sort are asymptotically optimal in comparison models

L23: Randomized and Worst Order Statistics

Returns ith smallest element in A[p..r]

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i = k          ▷ the pivot value is the answer
6     then return A[q]
7  elseif i < k
8     then return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```



let i=7

6	10	13	5	8	3	2	11

pivot

2	5	3	6	8	13	10	11

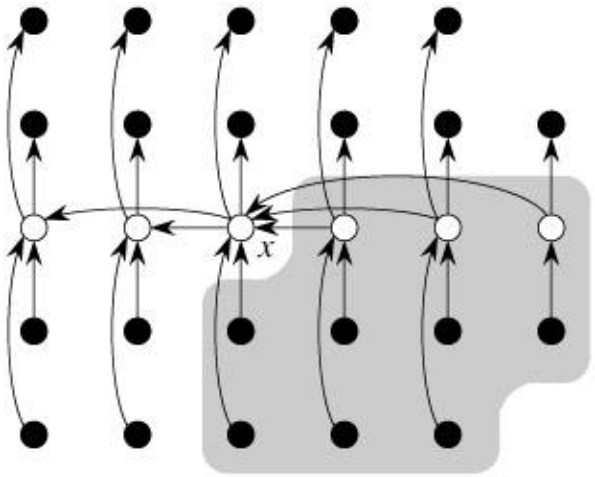
q=4

Now we are looking for 7-4=3rd rank in the right list
Analysis (assume all are distinct)

lucky case: any constant fraction split
$T(n)= T(9n/10) + \theta(n)$
from master theorem $n^{\log_{10/9} 1} = n^0 = 1$ so complexity= $\theta(n)$
worst case analysis $T(n)=T(n-1)+ \theta(n) = \theta(n^2)$

Analysis of expected time
T(n) be the random variable for running time of randselect on an input of size n of random numbers. Assume random numbers are independent.
-define indicator random variable x_k for k=0,1…n-1
x_k = 1 if partition generates k : n-k-1 split
0 otherwise
$T(n) \le T_{max}\{0, n-1\} + \theta(n)$ if 0 …n-1 split
$T_{max}\{1, n-2\} + \theta(n)$ if 1 …n-2 split
:
$T_{max}\{n-1, 0\} + \theta(n)$ if n-1 …0 split
= Sigma (K=0 to n-1) $x_k (T(\max\{k,n-k-1\}) + \theta(n))$
Take Expectation of both sides
$E[T(n)]= E[\text{Sigma (K=0 to n-1)} \; x_k (T(\max\{k,n-k-1\}) + \theta(n))]$
$E[T(n)]= \text{Sigma (K=0 to n-1)} \; E[x_k (T(\max\{k,n-k-1\}) + \theta(n))]$ by linearity of expectation
$E[T(n)]= \text{Sigma (K=0 to n-1)} \; E[x_k] E[(T(\max\{k,n-k-1\}) + \theta(n))]$ Expectation of product of two functions is product of expectations of individual functions if functions are independent
Linearity of Expectation $E[x_k]=1/n$
$E[T(n)]= 1/n \text{ Sigma (K=0 to n-1)} \; E[(T(\max\{k,n-k-1\})] + 1/n \text{ Sigma (K=0 to n-1)} \quad \theta(n)$
Upper terms appear twice
$E[T(n)]= 2/n \text{ Sigma (K=n/2 to n-1)} \; E[(T(k)] + \theta(n)$

By substitution method we guess that E[T(n)] ≤ cn for sufficient large constant > 0
From induction method assume it is true for 1…n-1
E[T(n)] ≤ 2/n Sigma (K=n/2 to n-1) ck + θ(n)
≤ 2c/n ($3n^2/8$) + θ(n)
= cn – (c/4(n- θ(n))
This is ≤ cn for sufficiently large value of c
So randomized select expected running time is θ(n)



Worst Case linear time order statistics

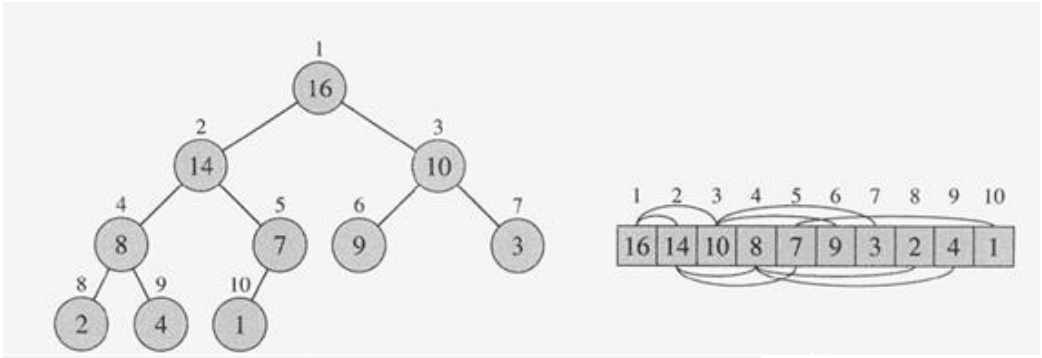
1. Divide the n elements into (n/5) groups of 5 elements each
   Find the median of each group (can be done in θ(n) time)
2. Recursively select median x of the (n/5) group medians
(can be done in T(n/5) time)
3. Partition with x as pivot let k = rank(x)
4. if i=k then return x
   if i<k then recursively select the ith smallest element in the lower left part of the array
   else recursively select (i-k)th element in the upper right part of the array

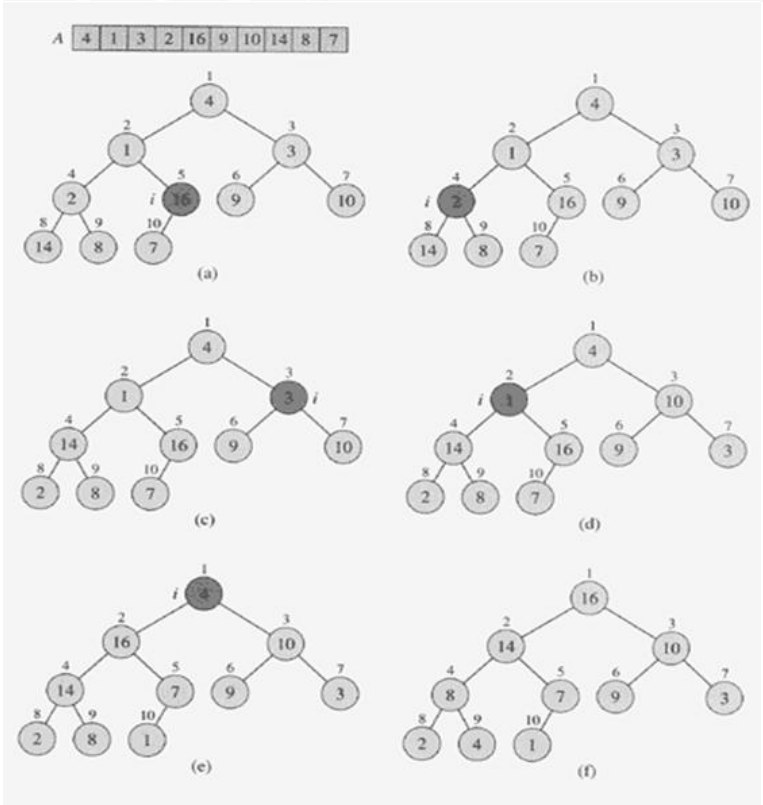
At least half of the medians found in step 2 are greater than or equal to the median-of-medians x. Thus, at least half of the ceil(n/5) groups contribute 3 elements greater than x, except, possibly, for one group containing less than 5 elements and the group containing x itself. Therefore, the number of elements > x is at least
3n/10 Similarly, the number of elements < x is at least 3n/10
For n>=50 we have 3n/10>= n/4
Therefore for n>= the recursive call to Select is executed on <= 3n/4 elements
Thus the recurrence for running time takes at most T(3n/4) for this step.
For n<50 T(n)= θ(1)
So step wise it becomes θ(n) + T(n/5) + θ(n)+T(3n/4)
By Substitution T(n)<= cn
we have
T(n) <= 1/5 cn + ¾ cn+ θ(n)
=19cn/20+ θ(n)
= cn-(cn/20 - θ(n))n
<=cn
if c is chosen large enough to handle both the θ(n) and the initial conditions.

L24:Heap Sort

Heap A (not garbage-collected storage) is a nearly complete binary tree.
· Height of node = # of edges on a longest simple path from the node down to a leaf.
· Height of heap = height of root = _(lg n).
· A heap can be stored as an array A.
· Root of tree is A[1].
· Parent of A[i] = A[i/2].
· Left child of A[i] = A[2i].
· Right child of A[i] = A[2i + 1].
· Computing is fast with binary representation implementation of a max-heap. [Arcs above and below the array on the right go between parents and children.

Heap property
For max-heaps (largest element at root), for all nodes i, excluding root, A [Parent(i)] ≥ A[i].
For min-heaps (smallest element at root), for all nodes i, excluding the root, A[Parent(i)] ≤ A[i].
By induction and transitivity of ≤, the max-heap property guarantees that the maximum element of a max-heap is at the root. In general, heaps can be k-ary tree instead of binary

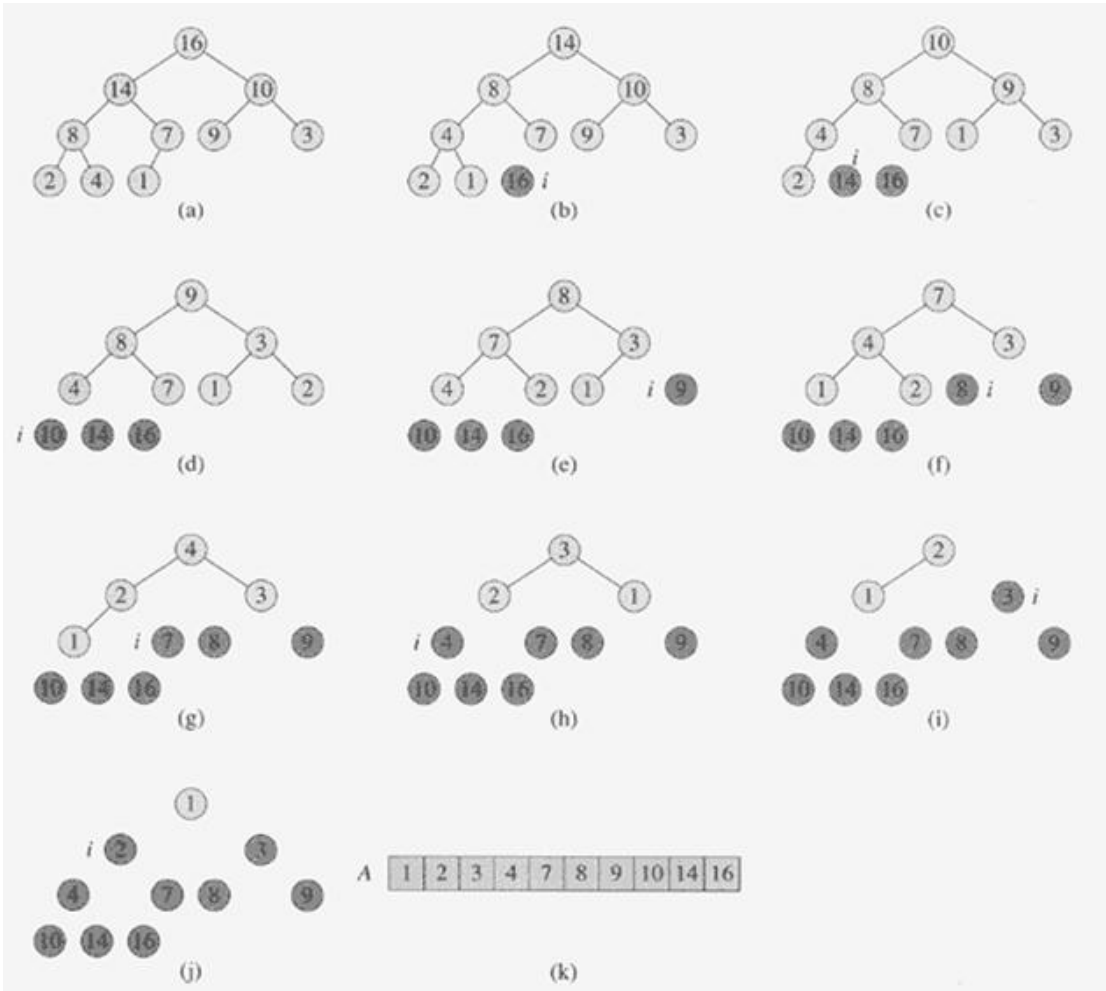





```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

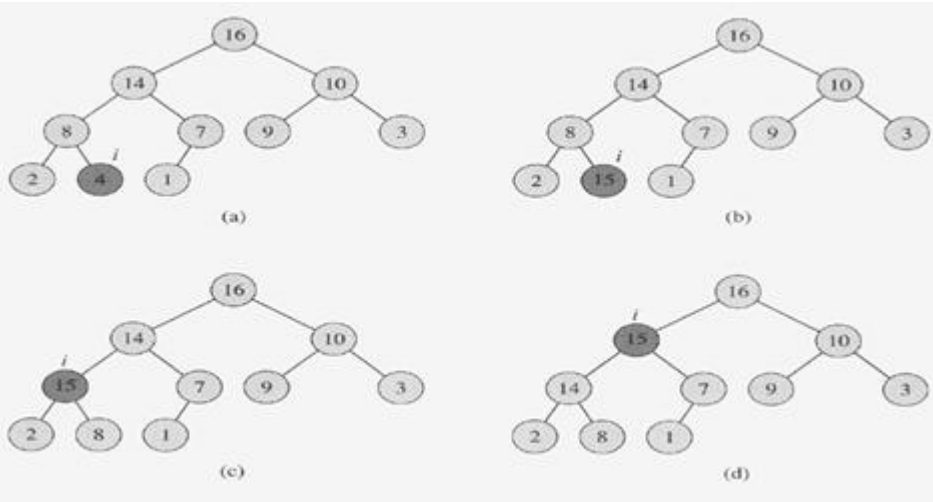
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3       do exchange A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          MAX-HEAPIFY(A, 1)
```



Priority Queue

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2     then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

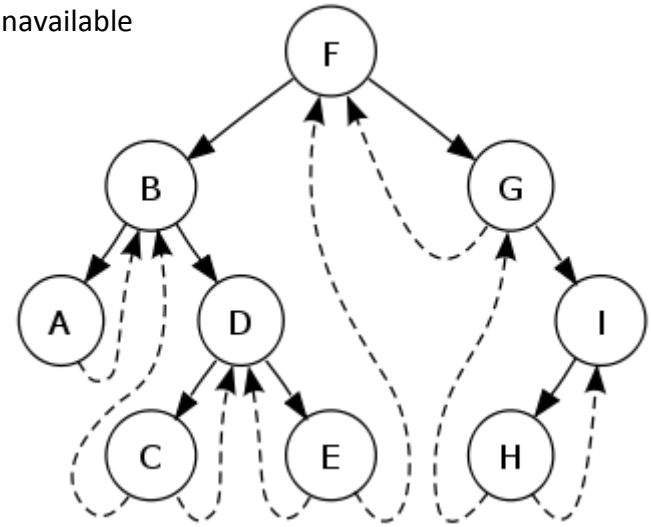
Increasing a number



L25: AVL Trees

Threaded Binary Tree

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.
A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.
It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable



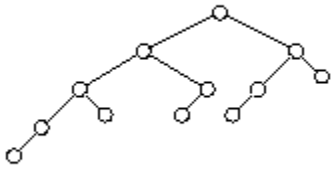
AVL Trees

These are also called height balanced trees. AVL tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1.

The height of an AVL tree T storing n keys is O(logn)
Proof: the minimum number of nodes in an AVL tree of height h
for n(1) = 1
for n(2) = 2
For h≥3 an AVL tree of height h contains the root node, one AVL subtree of height h-1 and the other AVL subtree of height h-2
n(h)= 1+ n(h-1)+n(h-2) we know that n(h-1) ≥n(h-2)
n(h)>2n(h-2)
>4n(h-4)
>8n(h-6)
> 2^in(h-2i)

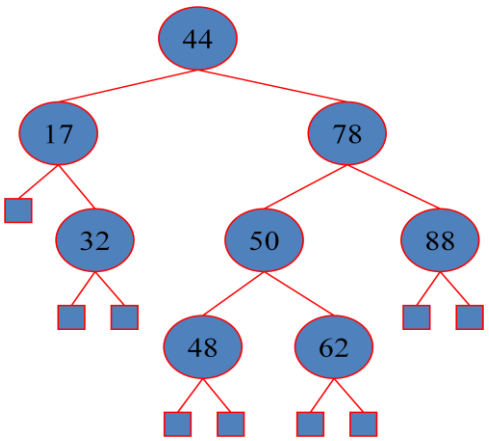
suppose i=h/2-1
n(h)>$2^{h/2-1}$ n(2) = $2^{h/2}$ is the minimum number of elements
h< 2log n(h) so height is O(logn)

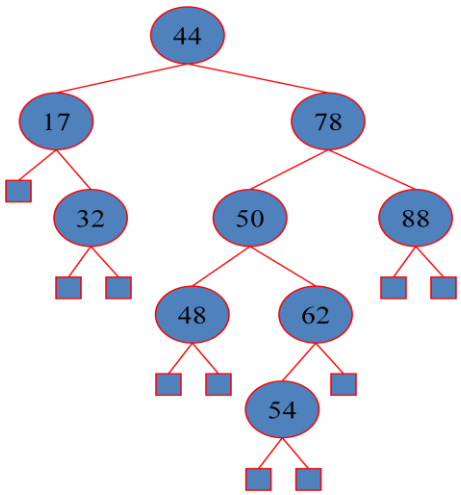
A more complex proof will show that AVL tree of n nodes has height atmost $\log_{1.63}n$

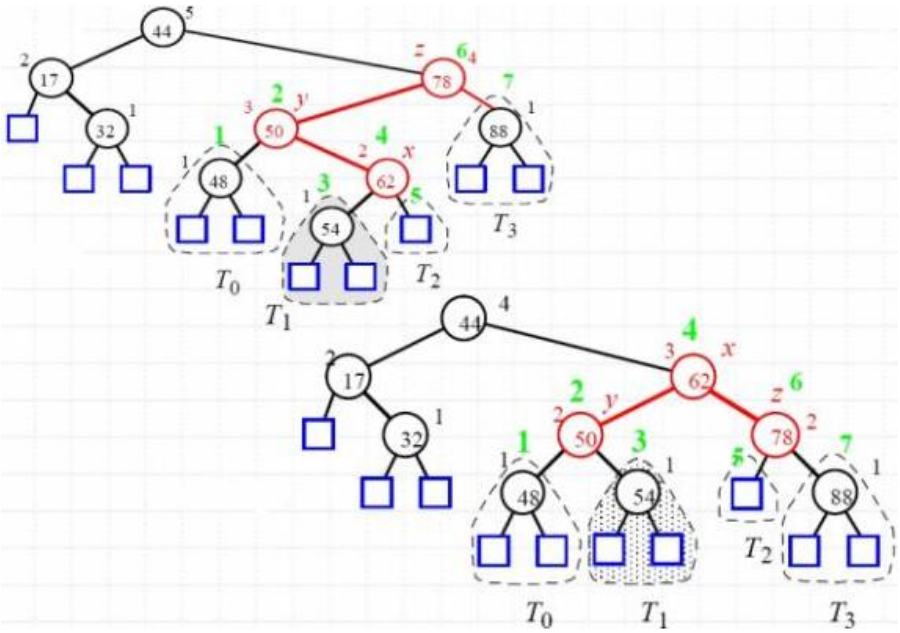


If closest leaf is at level k, all nodes at levels 1 .. k-2 have 2 children. In an AVL tree of height h, the leaf closest to the root is at level (h+1)/2. On the first (h-1)/2 levels the AVL tree is a complete binary tree. After (h-1)/2 levels the AVL tree may start thinning out. Number of nodes in the AVL tree is at least $2^{(h-1)/2}$ and at most 2^h. Search is same as BST.

Insertion is done similar to insertion in an binary tree. If insertion causes T to become unbalanced. We travel up the tree from the newly created node until we find the first node X such that its grandparent z is unbalanced node. Let Y be the parent of node X.
The only nodes whose heights can increase are the ancestors of node v.





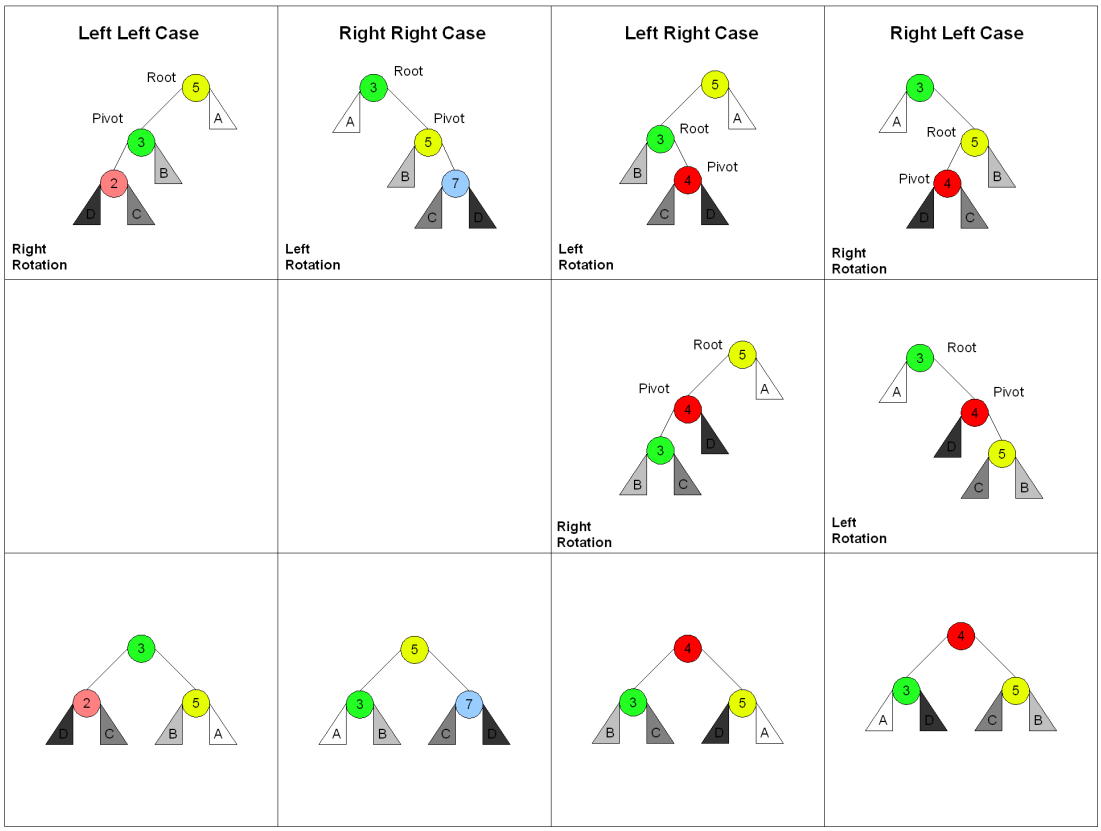


Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
When Deleting in a BST , We either delete a leaf or a node with only one child. In an AVL tree if a node has only one child, then that child is a leaf. So in an AVL tree we either delete a leaf or a parent of a leaf.

Let *z* be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
We perform restructure(x) to restore balance at z.
As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached
a single restructure is O(1)
using a linked-structure binary tree
find is O(log n)
height of tree is O(log n), no restructures needed
insert is O(log n)
initial find is O(log n)
Restructuring up the tree, maintaining heights is O(log n)
remove is O(log n)
initial find is O(log n)
Restructuring up the tree, maintaining heights is O(log n)

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



L26: Amortized and Competitive Analysis

Random Numbers

1. The random numbers should be uniformly distributed.
2. They should be statically independent.
3. Though the stream of random numbers will repeat depending on the parameters used in their generation, the stream length should be sufficiently larger than the desired length for a particular application. The generation of random numbers should be faster.

There are various tests to check the randomness of numbers like Frequency or equidistribution test, Gap Test, Run Tests, Chii-square test, Partition test, Permutation test, Collision test, Kolmogorov-smirnov test etc

The mid square method of random number generation, an initial seed is assumed and that number is squared. The middle four digits of the squared value are taken as the first random number. Next, the random number which is generated most recently is again squared and the middle most four digits of this squared value are assumed as the next random number.
This is to be repeated to generate the required number of random numbers

Relatively slow
- Statistically unsatisfactory
- Sample of random numbers may be too short
- There is no relationship between the initial seed and the length of the sequence

The Linear Congruential Method

By far the most popular random number generators in use today are special cases of the following scheme, introduced by D. H. Lehmer in 1949. We choose four "magic numbers:
m, the modulus; m > 0.
a, the multiplier; $0 \leq a < m$.
c, the increment; $0 \leq c < m$.
X_0, the starting value; $0 \leq X_0 < m$.

The desired sequence of random numbers (X_n) is then obtained by setting

$X_{n+1} = (a X_n + c) \bmod m, n \geq 0$ ..... (1)
This is called a linear congruential sequence.

LCGs are fast and require minimal memory (typically 32 or 64 bits) to retain state. This makes them valuable for simulating multiple independent streams. · Linear congruential random number generators are widely used in simulation and Monte Carlo calculations.

Quadratic Congruential Method

It is proposed by R.R.Coveyou. It is used to get more random numbers.
$Xn+1 = (d(Xn)^2 + aXn + c) \bmod m$

Fibonacci Generator

The simplest sequence in which Xn+1 depends on more than one of the preceding values is the Fibonacci sequence, $Xn+1 = (Xn + Xn-1) \bmod m$.

Combined Multiple Recursive Generator (CMRG)

An approach is to combine two or more multiplicative congruential generators to have good statistical properties and longer periods.

$X_n = (X_n + X_{n-k}) \bmod m$ ..... (6)

When k is a comparatively large value.

Amortized analysis

Analyze a sequence of operations on a data structure.

Goal is to show that although some individual operations may be expensive, on average the cost per operation is small. Average in this context does not mean that we are averaging over a distribution of inputs.

No probability is involved. We are talking about average cost in the worst case.

There are 3 methods: Aggregate analysis Accounting method Potential method

Binary counter

k-bit binary counter A[0 . . k − 1] of bits, where A[0] is the least significant bit and A[k − 1] is the most significant bit. Counts upward from 0.

Value of counter is Sigma (i=0 to k-1) $A[i] \cdot 2^i$

Initially, counter value is 0, so A[0 . . k − 1] = 0.

Increment(A, k)

i ⟵ 0

while i < k and A[i] = 1

do A[i] ⟵ 0

i ⟵i + 1

if i < k

then A[i] ⟵ 1

Example: k = 3

counter value	A 2 1 0	cost
0	0 0 0	0
1	0 0 1	1
2	0 1 0	3
3	0 1 1	4
4	1 0 0	7
5	1 0 1	8
6	1 1 0	10
7	1 1 1	11
0	0 0 0	14
...	...	15

Cost of INCREMENT = # of bits flipped

Analysis: Each call could flip k bits, so n INCREMENTs takes O(nk) time.

Observation is that not every bit flips every time.

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	n/2
2	1/4 the time	n/4
...		
i	1/2i the time	n/2i

$i \geq k$ never 0

Therefore, total # of flips = Sigma (i=0 to k-1) $n/2^i$

< n Sigma (i=0 to infinity) $1/2^i$

= n(1/1 − ½)

= 2n .

Average cost per operation = O(1).

Accounting Method

Assign different charges to different operations.
Some are charged more than actual cost and some are charged less.
Amortized cost = amount we charge.
When amortized cost > actual cost, store the difference on specific objects in the data structure as credit.
Use credit later to pay for operations whose actual cost > amortized cost.
Differs from aggregate analysis:
In the accounting method, different operations can have different costs.
In aggregate analysis, all operations have same cost.
Need credit to never go negative.
Otherwise, have a sequence of operations for which the amortized cost is not an upper bound on actual cost.
Let c_i = actual cost of i th operation ,
c_i cap = amortized cost of i th operation .
Then require
Sigma (i=1 to n $c_i$cap) >= Sigma (i=1 to n c_i)
for all sequences of n operations.
Total credit stored = Sigma (i=1 to n $c_i$cap) - Sigma (i=1 to n c_i)>=0
Binary Counter
Charge $2 to set a bit to 1. $1 pays for setting a bit to 1.$1 is prepayment for flipping it back to 0.
Have $1 of credit for every 1 in the counter.
Therefore, credit ≥ 0.
Amortized cost of increment: Cost of resetting bits to 0 is paid by credit.
At most 1 bit is set to 1. Therefore, amortized cost ≤ $2. For n operations, amortized cost = O(n).

Potential Method

Like the accounting method, but think of the credit as potential stored with the entire data structure.
• Accounting method stores credit with specific objects.
• Potential method stores potential in the data structure as a whole.
• Can release potential to pay for future operations. • Most flexible of the amortized analysis methods.
Let D_i = data structure after i th operation ,
D_0 = initial data structure , c_i = actual cost of i th operation , $capc_i$ = amortized cost of i th operation .
Potential function Φ : Di → R
Φ (Di) is the potential associated with data structure Di.
cap c_i = c_i + Φ (D_i) −Φ (D_{i-1})
Increase in potential due to i th operation
Total amortized cost = Sigma (i=1 to n $c_i$cap)
= Sigma (i=1 to n c_i)+ ΦD_i −ΦD_{i-1} (telescoping sum: every term other than D0 and Dn is added once and subtracted once)
= Sigma (i=1 to n c_i)+ ΦD_n −ΦD_0
<= Sigma (i=1 to n c_i)
If we require that Φ (D_i) ≥Φ (D_0) for all i , then the amortized cost is always an upper bound on actual cost.
In practice: Φ (D0) = 0, Φ (Di) ≥ 0 for all i .
Binary counter
Φ = bi = # of 1's after i th Increment
Suppose i th operation resets ti bits to 0.
c_i ≤ t_i + 1 (resets ti bits, sets ≤ 1 bit to 1)
• If bi = 0, the i th operation reset all k bits and didn't set one, so
b_i−1 = t_i = k ⇒b_i = b_i−1 − t_i.

• If $b_i > 0$, the i th operation reset t_i bits, set one, so
$b_i = b_i{-}1 - t_i + 1$.
• Either way, $b_i \leq b_i{-}1 - ti + 1$.
• Therefore, deltaΦ (D_i) ≤ (bi−1 − ti + 1) − bi−1
= 1 − t_i .
capci = ci + deltaΦ (Di)
≤ (t_i + 1) + (1− t_i)
= 2.
If counter starts at 0, Φ (D0) = 0. Therefore, amortized cost of n operations = O(n).

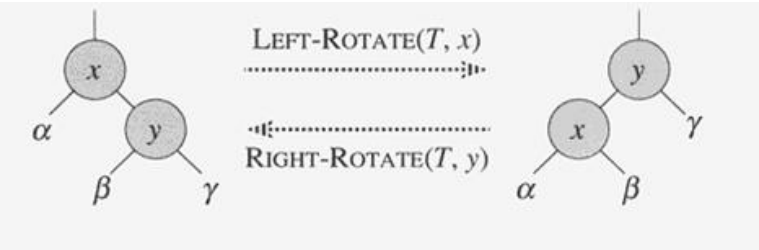
L27:Red Black Trees

It is a BST data structure with extra information in each node called the color filed satisfying properties
1. Every Node is either red or black.
2. Root and leaves are all black.
3. Every red node has black parent (means two red nodes cannot appear consecutively while black can appear)
4. All simple paths from node n to a descendent leaf of n has same number of black nodes called black height(x).

Red black tree with n keys has height h≤ 2 lg(n+1)
2-4 Trees : All can have 1,2 or 3 keys that means either 2 or 3 or 4 children respectively except leaf which has no children. Depth of all leaves is same. The Height of the 2-4 tree is black height of the red-black tree created. Any 2-4 tree can be converted to a red black tree and Vice versa.
To convert RB Tree into 2-4 tree Merge each red node into its black node. Now all leaves have same depth in the tree which is black height of root.
To convert 2-4 Tree into RB tree replace a node of the 2-4 tree with one black node and 0/1/2 red nodes which are children of the black node. The Height of the 2-4 tree is the black height of the red-black tree created.
Every red node has a black child.

leaves=number of internal nodes+1

$2^h \le$ number of leaves in 2-3-4 trees is $\le 4^h$
$2^h \le n+1$
$h \le \lg(n+1)$ (This is for 2-3-4 trees)
For RB trees We know that less than half nodes on any root to the leaf path are red $h \le 2\lg(n+1)$
There can be only two types of problems in the tree after Insertion or deletion.
1. Black height does not remain Uniform
2. Double Red Problem
Insertion: we insert like a BST and color the newly inserted node red. Since inserted node is colored red, the black height of the tree remains unchanged. However, if the parent of the inserted node is also red then we have a double red problem. In case of insertion we will never have a black height problem and in case of deletion we will never have double red problem.
Insertion Case 1:



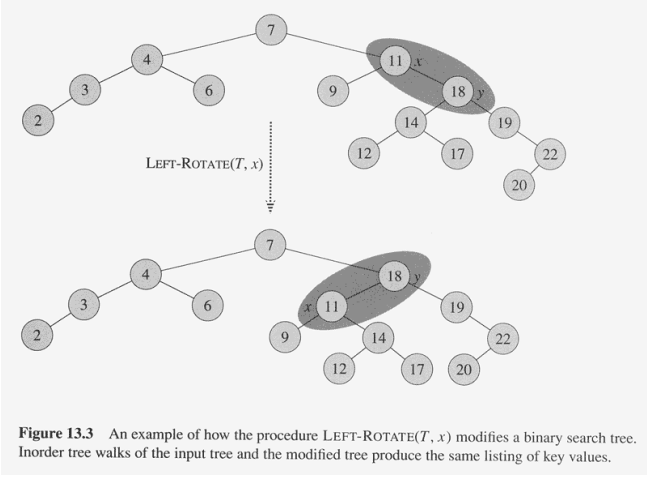


Figure 13.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

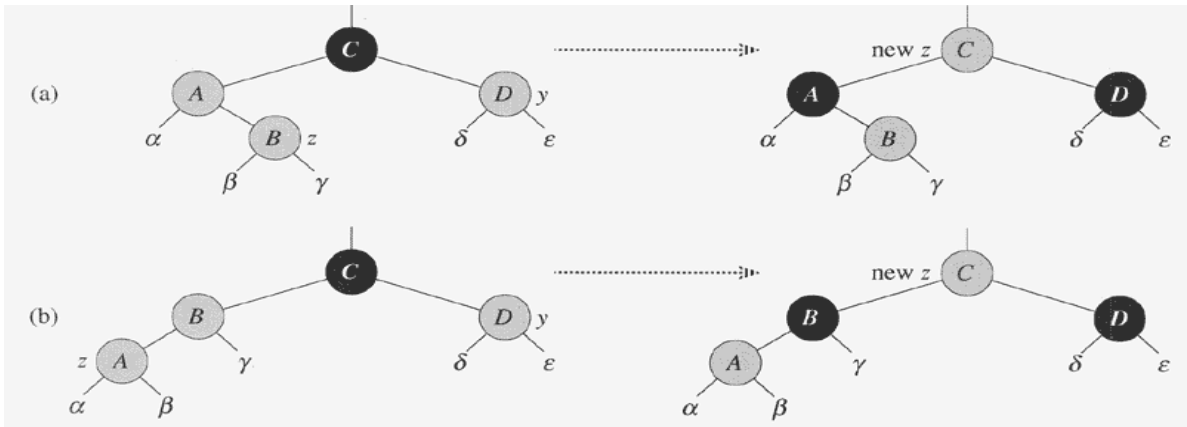


Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent $p[z]$ are both red. The same action is taken whether **(a)** z is a right child or **(b)** z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node z's grandparent $p[p[z]]$ as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

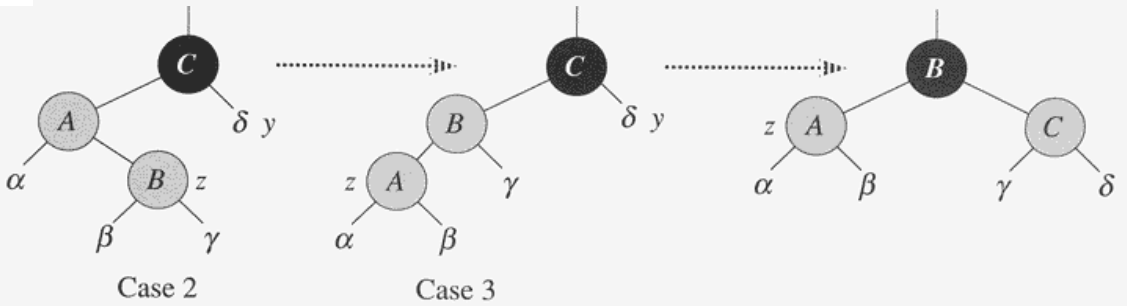


Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent $p[z]$ are both red. Each of the subtrees α, β, γ, and δ has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The **while** loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.

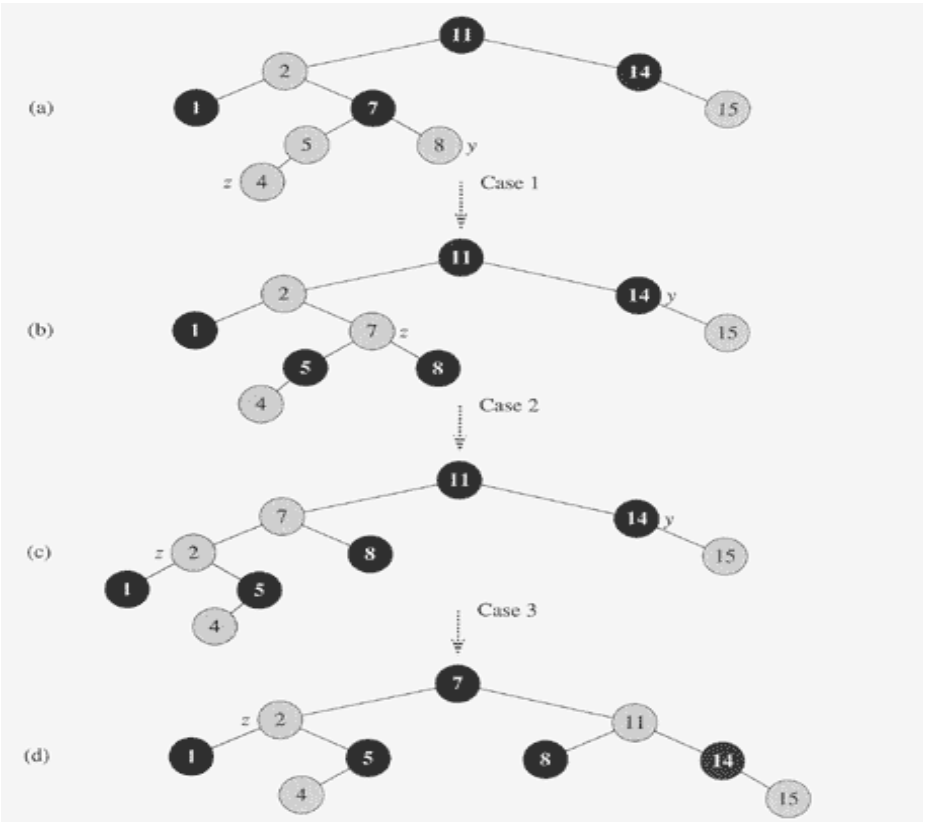
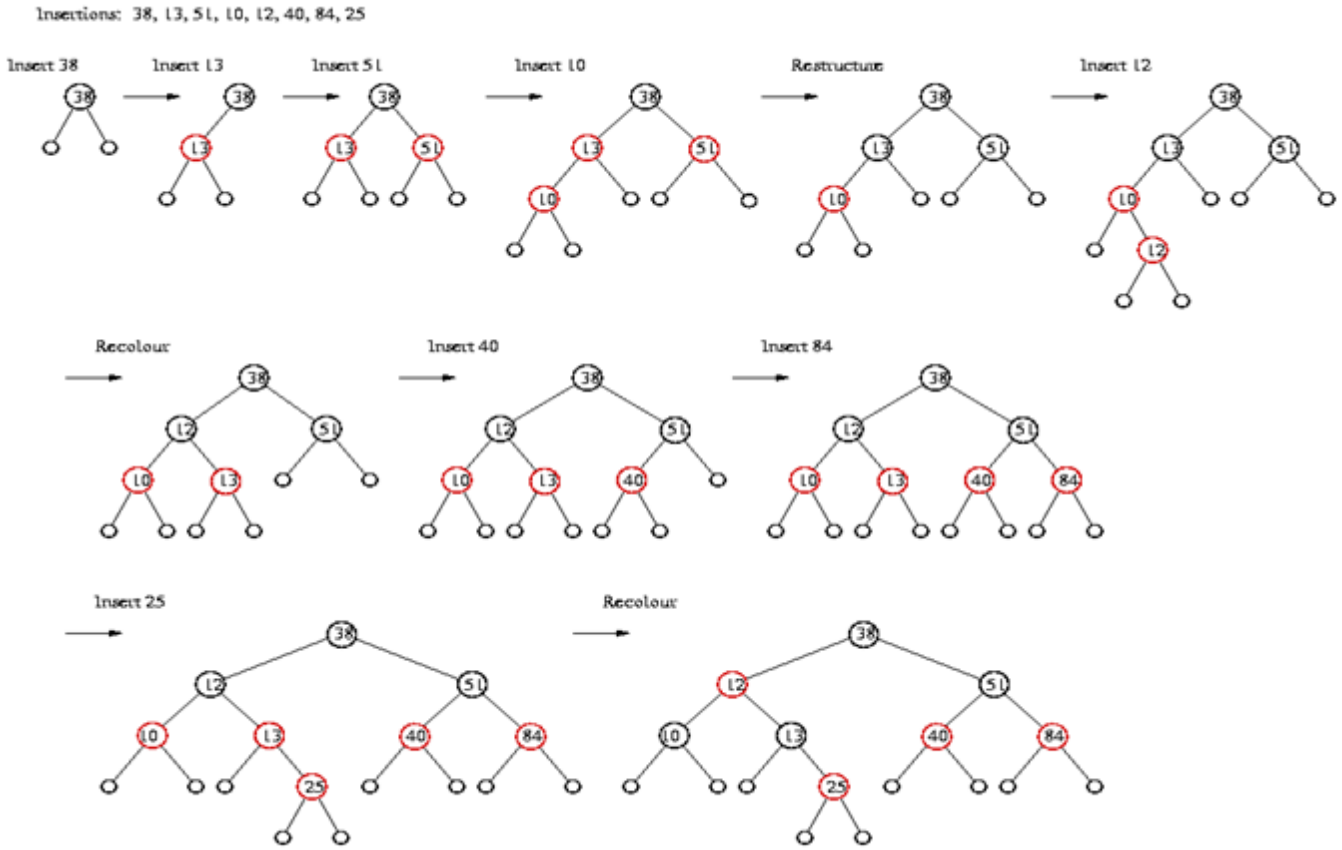
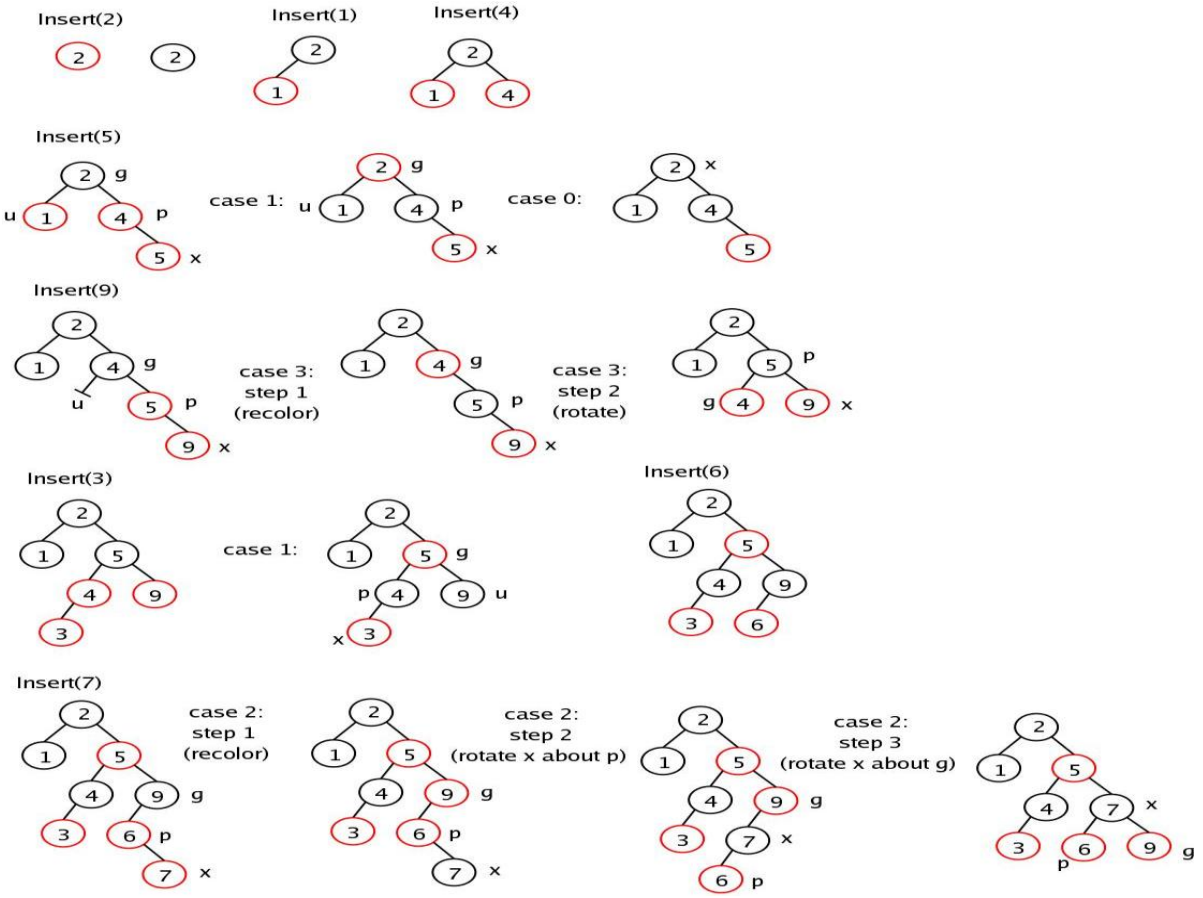


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Since z and its parent $p[z]$ are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $p[z]$, case 2 can be applied. A left rotation is performed, and the tree that results is shown in **(c)**. Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in **(d)**, which is a legal red-black tree.


Insertions: 38, 13, 51, 10, 12, 40, 84, 25
Insert 38
Insert 13
Insert 51
Insert 10
Restructure
Insert 12
Recolour
Insert 40
Insert 84
Insert 25
Recolour


Insert(2)
Insert(1)
Insert(4)
Insert(5)
case 1:
case 0:
Insert(9)
case 3:
step 1
(recolor)
case 3:
step 2
(rotate)
Insert(3)
case 1:
Insert(6)
Insert(7)
case 2:
step 1
(recolor)
case 2:
step 2
(rotate x about p)
case 2:
step 3
(rotate x about g)

For Deletion see Printout pages of Coreman Instructor Manual

L28:Splay Trees

Splay Trees are Binary Search Trees
Searching in a Splay Tree: Starts the Same as in a BST
splay trees are much simpler then AVL or Red Black trees. Their average performance is comparable and they provide additional properties not found in AVL or RB trees. It has working set and dynamic finger property.
a *splay tree* is a binary search tree where a node is splayed after it is accessed (for a search or update)
deepest internal node accessed is splayed
splaying costs O(h), where h is height of the tree – which is still O(n) worst-case
O(h) rotations, each of which is O(1)
Thus, amortized cost of any splay operation is **O(log n).**
In fact, the analysis goes through for any reasonable definition of rank(x).
This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases
Which nodes are splayed after each operation?

findElement	if key found, use that node if key not found, use parent of ending external node
insertElement	use the new node containing the item inserted
removeElement	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Sequential Access Property: We can do inorder traversal of a BST in O(1) amortized time.
Let the numbers of the set {1,2,…n} be keys stored in a binary search tree. Let $\{x_1,x_2,\ldots x_m\}$ be keys of an access pattern xi.
It is known that some access patterns are very easy . For example if x_i=k for all i, then placing k at the root will be O(1) access time. Similarly inorder access pattern will also give us constant amortized time.
Static Optimality of BST
let k be a key which appears in the access sequence with probability pk. Then the per operation cost of access is
O(Sigma pi(lg(1/pi))
This is because an element which appears with probability pi can be stored at height lgpi. This is the strict lower bound for static sets.
Dynamic finger property : if the last access was to key xi-1 and the present access is to key xi then the cost of the present access is $O(\lg|x_i-x_i-1|)$
This property performs well on access patterns which exhibit spatial locality
Working Set property: Let t_j the number of distinct keys between the present operation and the last time the same key was accessed. Let tj be the number of distinct keys in $\{x_i..x_j\}$ then the cost of accessing xj is $O(\lg t_j+2)$
Data structures with this property perform well on access patterns which exhibit temporal locality.
Unified Property: Let t_{ij} be the number of distinct keys in $\{x_i....x_j\}$ then the cost of accessing x_j is
$O(\lg \min_i (|x_i-x_j| + t_{ij} + 2)$
It says that access to keys close to keys which have been recently accessed is cheap.

L28: B-Trees or Multi Way search Trees

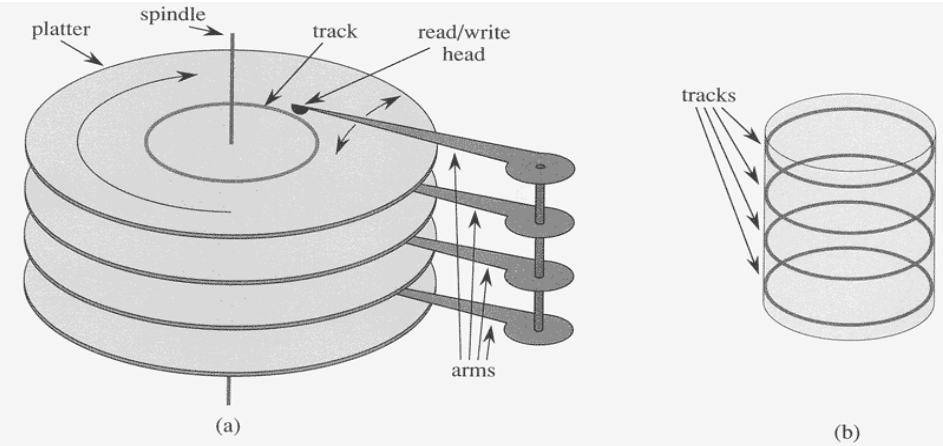


Figure 18.2 **(a)** A typical disk drive. It is composed of several platters that rotate around a spindle. Each platter is read and written with a head at the end of an arm. The arms are ganged together so that they move their heads in unison. Here, the arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when it is stationary. **(b)** A cylinder consists of a set of covertical tracks.

A ***B-tree*** T is a rooted tree (whose root is $root[T]$) having the following properties:

1. Every node x has the following fields:

a. $n[x]$, the number of keys currently stored in node x,

b. the $n[x]$ keys themselves, stored in nondecreasing order, so that $key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$,

c. $leaf\ [x]$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $n[x]+1$ pointers $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i\ [x]$, then $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le key_{n[x]}[x] \le k_{n[x]+1}$.

4. All leaves have the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:

a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t$ – 1 keys

B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and file systems.

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full.

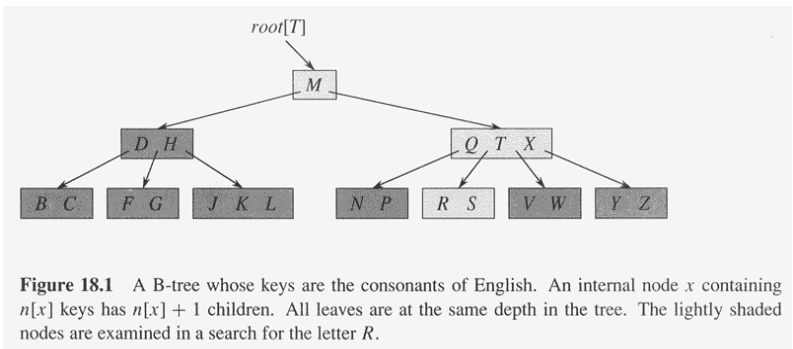


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

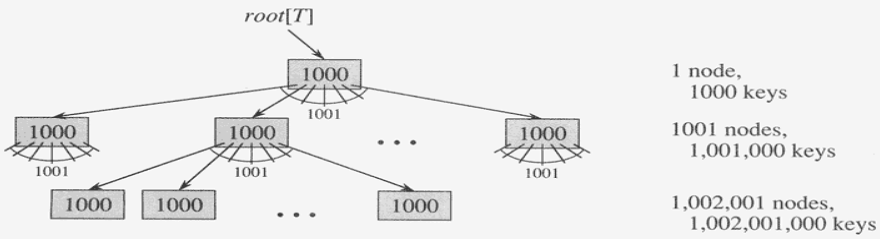


Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

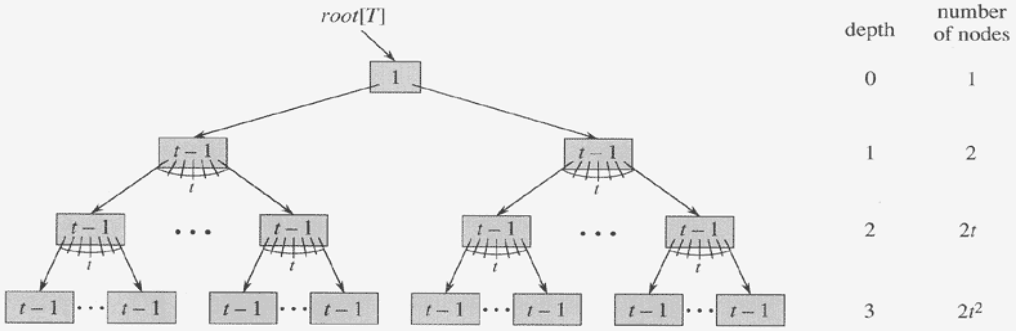


Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $n[x]$.

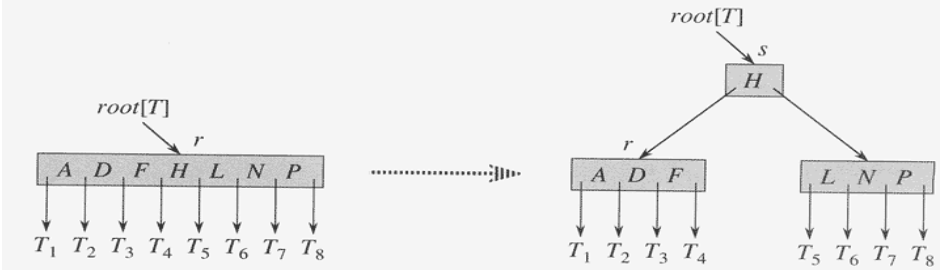


Figure 18.6 Splitting the root with $t = 4$. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

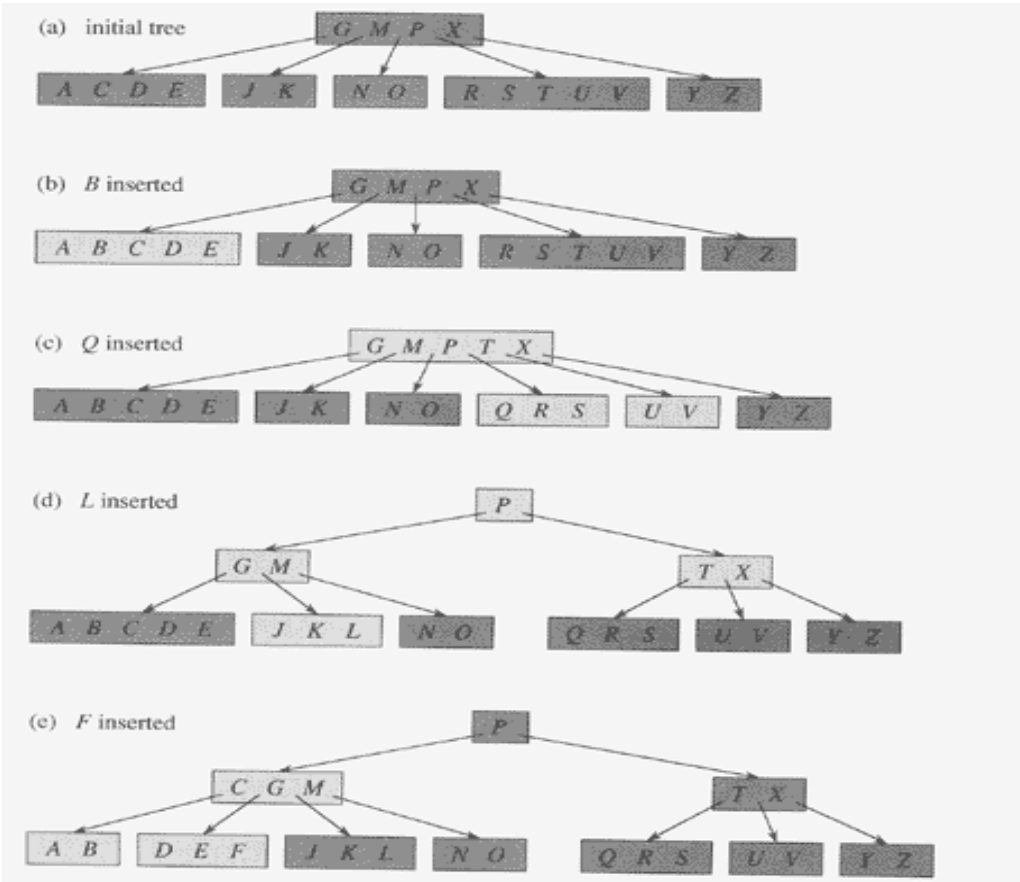
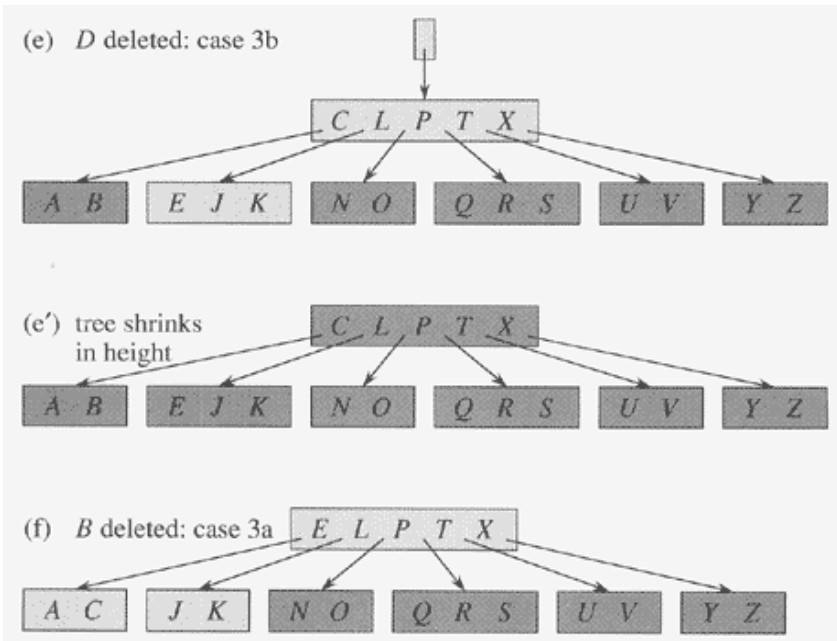


Figure 18.7 Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting B into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting Q into the previous tree. The node $RSTUV$ is split into two nodes containing RS and UV, the key T is moved up to the root, and Q is inserted in the leftmost of the two halves (the RS node). **(d)** The result of inserting L into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK. **(e)** The result of inserting F into the previous tree. The node $ABCDE$ is split before F is inserted into the rightmost of the two halves (the DE node).



1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.
a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)
b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)
c. Otherwise, if both y and z have only t - 1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t - 1 keys. Then, free z and recursively delete k from y.
3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only t - 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.
a. If $c_i[x]$ has only t - 1 keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.
b. If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have t - 1 keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node

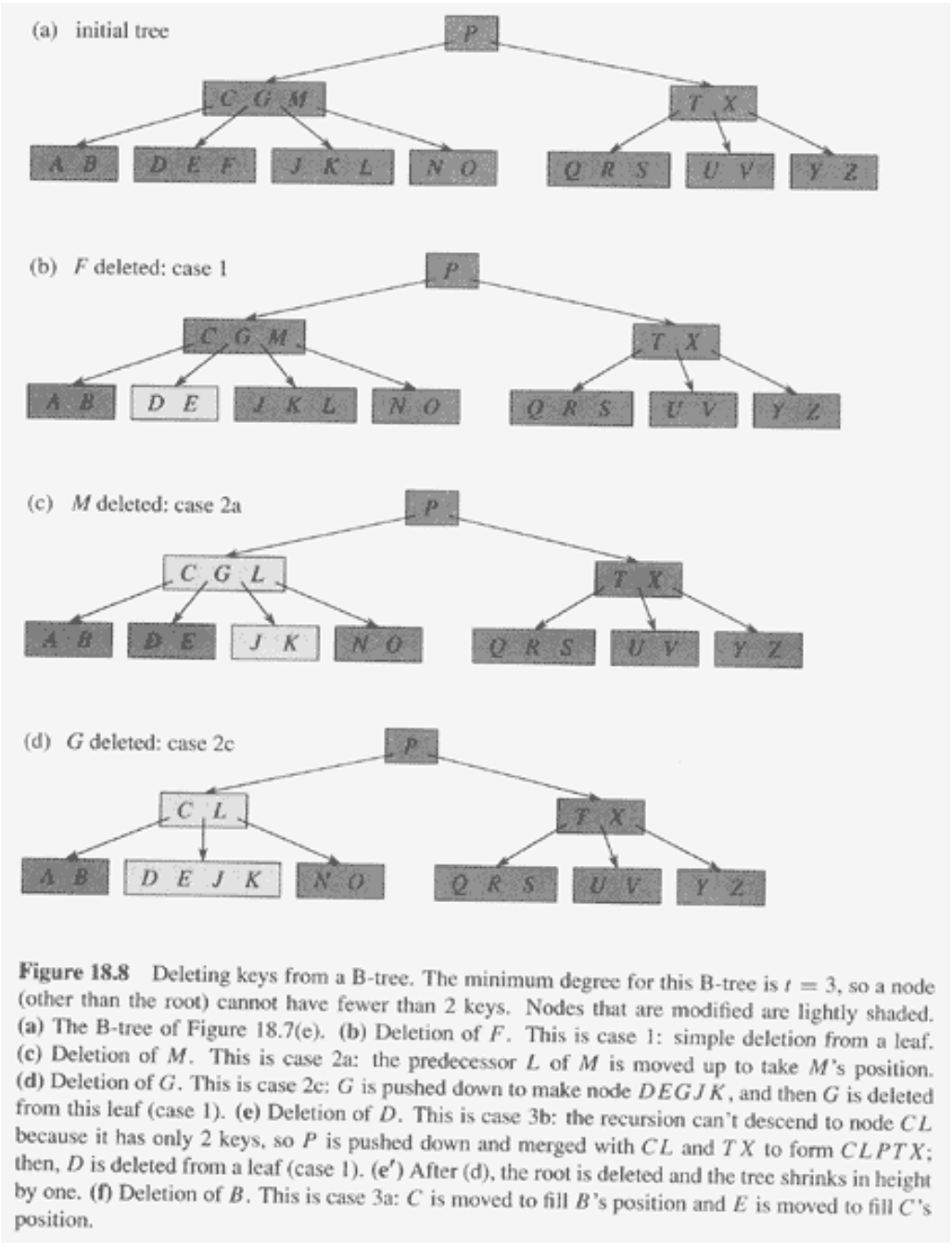


Figure 18.8 Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M is moved up to take M's position. **(d)** Deletion of G. This is case 2c: G is pushed down to make node $DEGJK$, and then G is deleted from this leaf (case 1). **(e)** Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and TX to form $CLPTX$; then, D is deleted from a leaf (case 1). **(e′)** After (d), the root is deleted and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.