SEARCHING TECHNIQUES

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

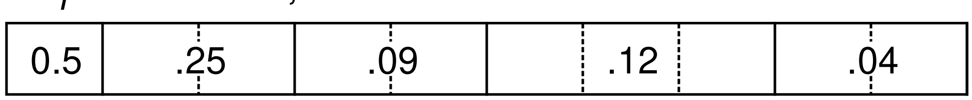
SEARCHING

- Simplest technique is to look at every record until finding the one we are looking for (known as *sequential search*)
- Speeding up sequential search:
  1. table-lookup
     - assumes existence of 1-1 mapping from dataset (i.e., one of the key values) to a memory address
     - frequently 1-1 mapping does not exist and use hashing to calculate a good starting point for the search
  2. preprocess data by sorting it
     - search is implemented by the repeated application of a partitioning process to the set of key values until locating the desired record
       a. tree-based
          - partitions the set of key values
          - e.g., binary search trees, AVL trees, B-trees, …
       b. trie-based
          - partitions based on the digits or characters that comprise the domain of the key values
          - e.g., digital searching, radix-trees, most quadtree methods

SEQUENTIAL SEARCHING

- Simplest way to search
- Two tests:
  1. has every record in the file been examined?
  2. is the current record the one we want?
- Expected cost of success is $2 \cdot n/2 = n$ tests
- Expected cost of failure is $2n$ tests
- Speed up sequential search by inserting the record with the desired key value k at the end of the file
  - eliminates need to test if all records have been examined
  - $n/2$ tests for success and $n+1$ tests for failure
  - halve the execution time at the cost of one additional location
  - assumes we know location of the end of the file for insertion
- For greater speedups, records need to be sorted
  1. sort enables us to know when to stop the search
  2. average $n/2$ comparisons for success and failure

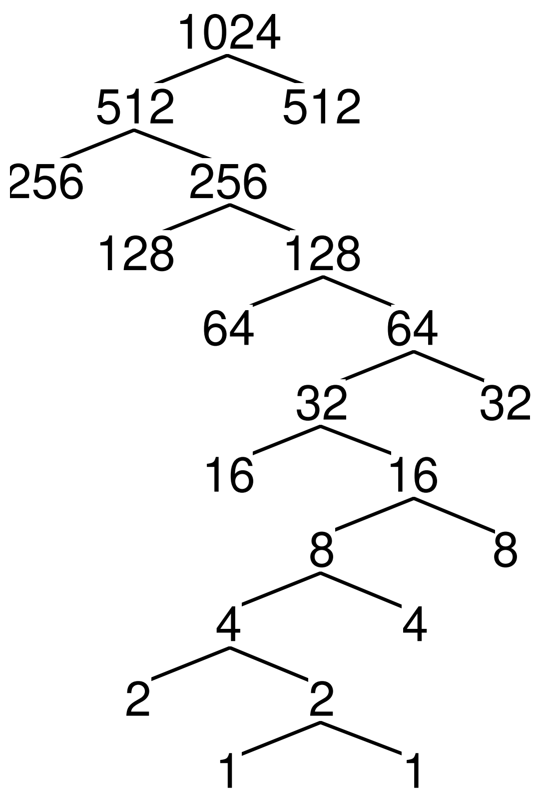
SELF-ORGANIZED FILES

- Sort by frequency of access rather than by value
- Two variants:
  1. all records of equal length
     - each time a record is accessed, it is moved to the start of the file
     - Ex: 44 55 12 42 94 06 67 18
     - drawback is that it rewards rare accesses
     - overcome by interchanging the accessed element with the immediately preceding element
     - Ex: 44 55 12 42 94 06 67 18
     - problem: if always access the same two elements in alternating order
  2. all records of varying length
     - retrieval time may depend on length (e.g., system tape)
     - goal: minimize average retrieval time
     - assume record i has length L_i and probability p_i of being retrieved
       a. average retrieval time =
          $p_1 \cdot L_1 + p_2 \cdot (L_1 + L_2) + \ldots + p_n \cdot (L_1 + L_2 + \ldots + L_n)$
       b. minimized when $p_1 / L_1 \geq p_2 / L_2 \geq \ldots \geq p_n / L_n$
          - as p_i increases, the record moves to the front
          - as L_i increases, the record moves to the rear
     - Ex: 

     - results in sorting on the basis of probability per unit length of a record

BINARY SEARCHING

- Analogous to searching through the phone book
- Repeatedly halve the data set while determining which partition to search next
- Reduces search time from $O(n)$ to $O(\log_2 n)$
- Ex: searching through $2^{10} = 1024$ elements requires just 10 comparisons



- Requires being able to identify the proper half of the list where the search is to be continued

  1. easy with sequential allocation
  2. linked allocation requires an intermediate array of pointers to the linked list

BINARY SEARCH ALGORITHM

- Search for `k` through `table` which is sorted
- Flag `found` (initially 0) returns the index to the entry in `table` containing `k`

```
integer procedure BINARY_SEARCH_TABLE(k,table,n);
begin
  value integer k,n;
  value record array table[1:n];
  integer low,high,m;
  low←1;
  high←n;
  while high≥low do
    begin
      m←(low+high)÷2;
      if k<KEY(table[m]) then high←m-1
      else if k=KEY(table[m]) then return(m)
      else low←m+1;
    end;
  return(0);
end;
```

- Ex: search for 44 in [06 12 18 42 44 55 67 94]

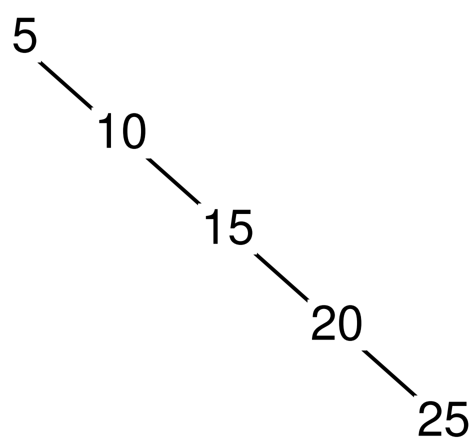
  1. low=1 high=8 m=4 [44 55 67 94]
  2. low=5 high=8 m=6 [44 55]
  3. low=5 high=5 m=5 [44]

- Advantage: search time is reduced from $O(n)$ to $O(\log_2 n)$
- Disadvantages:
  1. records must always be kept sorted
  2. insertion and deletion may be expensive as may need to move records around

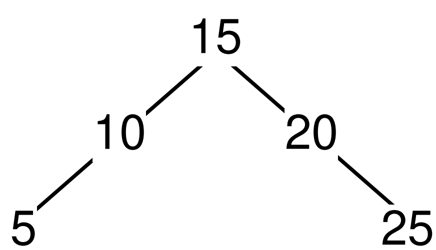
BINARY SEARCH TREES

- Drawbacks of binary searching (e.g., insertion and deletion) are a property of the implementation of binary search using sequential allocation
- Insertion and deletion are cheap when using a tree instead of a list
- Price is the extra space needed for the links
- $O(\log_2 n)$ search time assumes a balanced binary search tree
- Binary search trees are inefficient when the data is inserted in sorted order

  Ex: 5 10 15 20 25

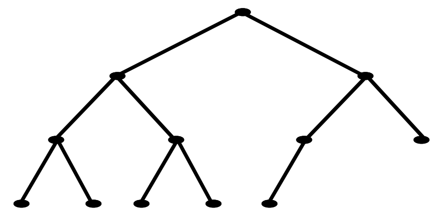


- Can overcome by applying a balancing operation

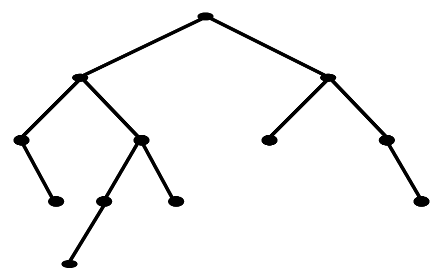


BALANCED BINARY SEARCH TREES

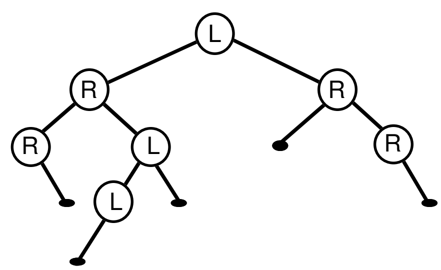
1. Global balance: all leaf nodes are at the maximum depth or at a depth one less than the maximum



2. Local balance: difference between the maximum depth of the left and right subtrees of nonleaf node t is at most 1



   - e.g., AVL trees
   - use balance factor L, E, R to indicate the left subtree being one level deeper, equal, or less, respectively than the right subtree



   - maximum depth is ≈ 1.44 $\log_2 n$ implying $O(\log_2 n)$ search, insertion, and deletion times

3. A Fibonacci tree T_d is an AVL tree of depth d having a minimum number of nodes
   - the subtrees of a Fibonacci tree T_d are Fibonacci trees of depth $d-1$ and $d-2$

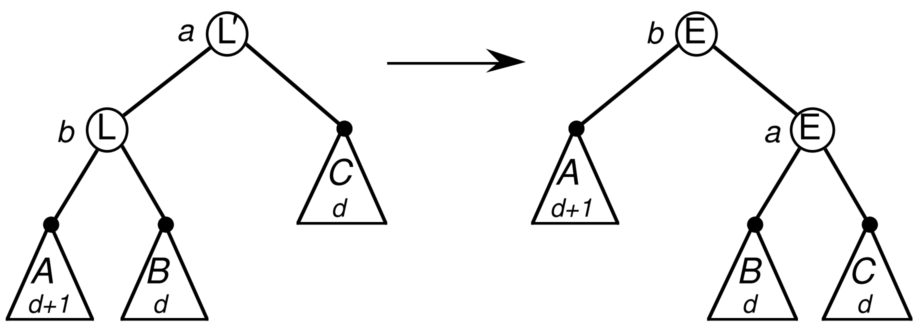
INSERTION INTO AVL TREES

- Analogous to binary search tree insertion
- Need to update balance factors
  1. a node with balance factor L whose left subtree became deeper
  2. a node with balance factor R whose right subtree became deeper
- Two cases are handled in an analogous manner by use of rotations
- Rotations involve changing links and no search — i.e., $O(1)$ time

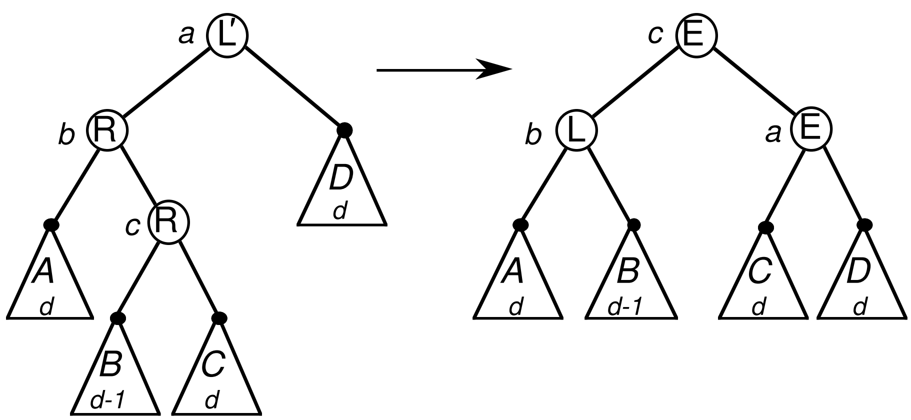
EXAMPLE ROTATIONS IN AVL TREES

- Node *a* with balance factor L whose left subtree became deeper

  1. balance factor of left son *b* is L
     - single rotation to the right (involves one level)



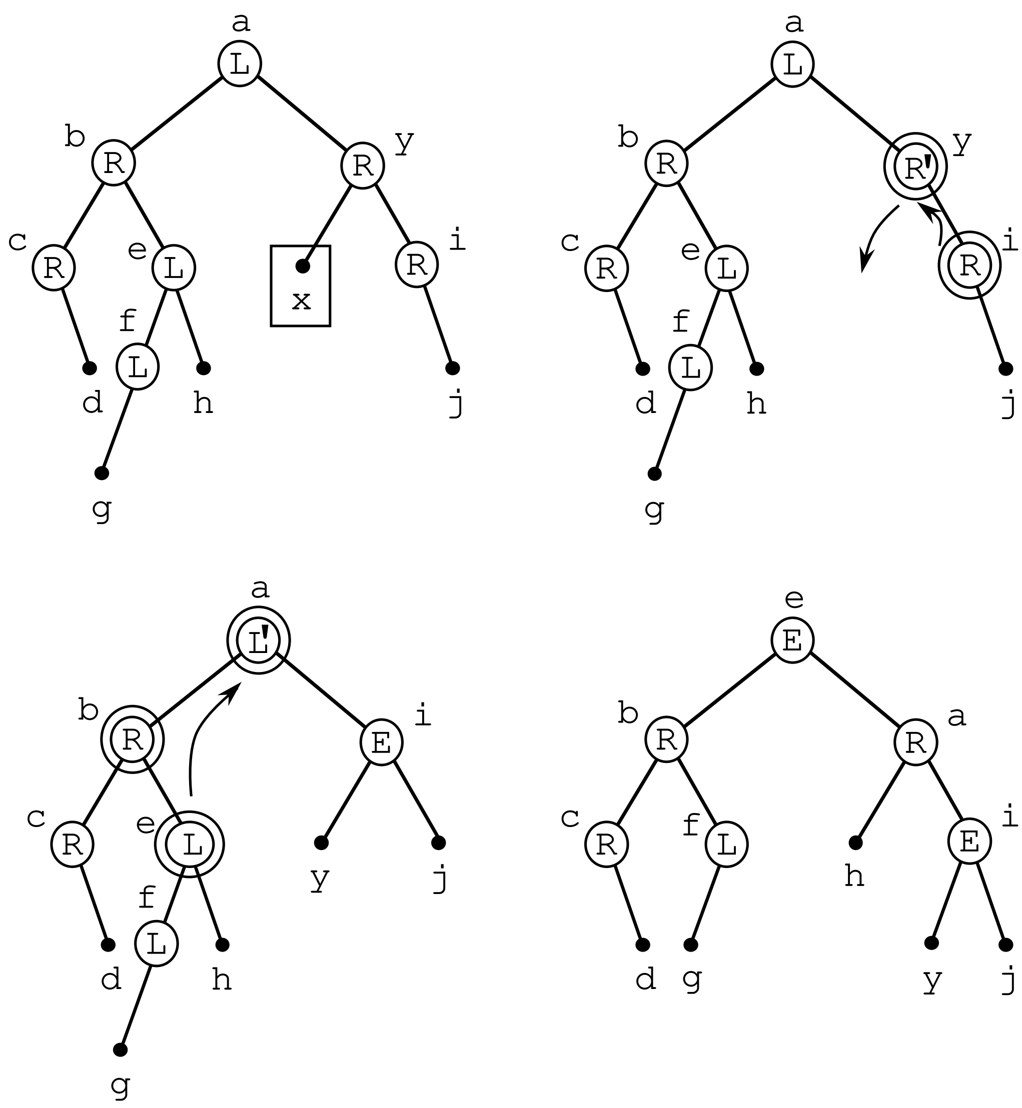
  2. balance factor of left son *b* is R
     - double rotation to the right (involves two levels)



  3. balance factor of left son *b* is E
     - impossible as the insertion caused the left subtree of *a* to be two deeper than the right subtree of *a*
     - thus the new record had to go into either the left or right subtrees of *b* whose depth must have increased thereby contradicting the fact that they are equal after the insertion

DELETION FROM AVL TREES

- Analogous to insertion
- Difference is that the number of rotations needed may be as high as the depth of the node being deleted
- Ex: delete *x*



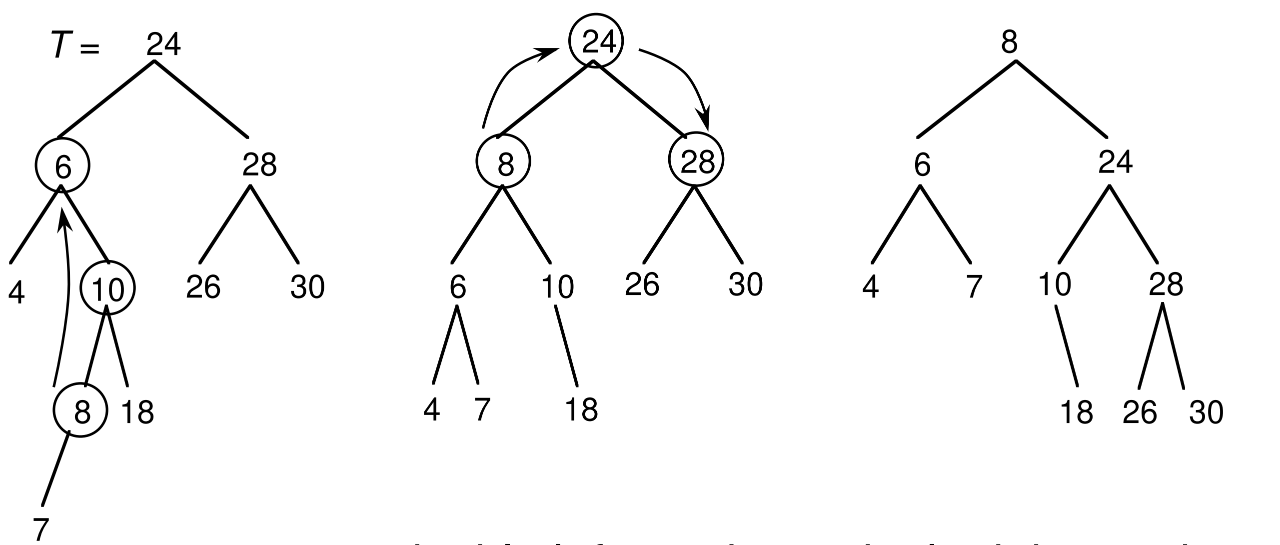
1. depth of right subtree of *y* exceeds that of the left subtree by more than 1 so need to rotate *y* and *i* to the left
2. depth of left subtree of root exceeds that of right subtree by more than 1 so need a double rotation of *e*, *b*, and *a* to the right

AMORTIZED ANALYSIS

1. An amortized analysis is the cost of performing a sequence of operations (e.g., on a data structure) averaged over the number of operations performed
   - e.g., if n operations can be performed in total of $O(n \log n)$ time, then the amortized cost of each operation is $O(\log n)$
   - of course, any single operation could take more time
2. Similar to average-case analysis as both involve averaging
   - average case
     a. expected cost of each operation is derived under the assumption that the operations are sampled from some known distribution
     b. because it is an expectation, in rare circumstances it may be possible to perform the same excessively costly operation over and over again
     c. however, this cannot happen in a data structure with good amortized performance
   - amortized
     a. bounds on the cost are derived by averaging over any sequence of operations
     b. even though a single operation in a sequence may be costly, the entire sequence cannot consist predominantly of excessively costly operations.
3. Several methods of obtaining an amortization analysis
   - Ex: aggregate method: establish an upper bound on total cost of sequence of n operations $T(n)$
   - properties:
     a. amortized cost of an operation is $T(n)/n$
     b. same amortized cost for each type of operation
     c. amortized cost may be different for other methods

SELF-ADJUSTING BINARY SEARCH TREES (SPLAY TREES)

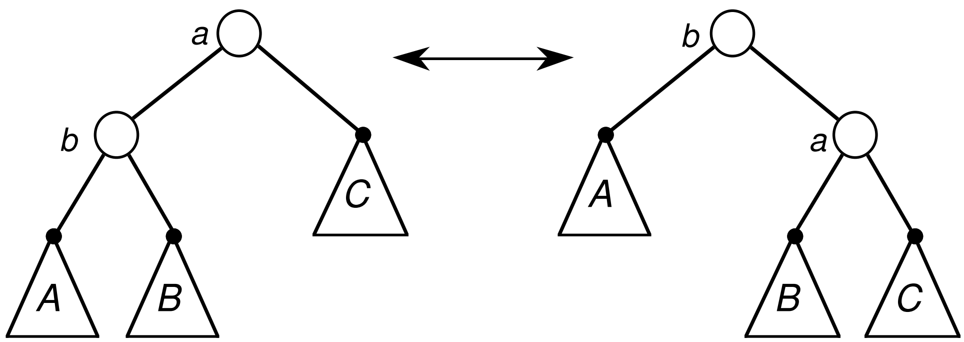
- Simple binary search trees
- No need for balance factors as in AVL trees
- Search, insertion, and deletion algorithms have an amortized cost of $\log_2 n$
- Like a self-organizing file in that whenever a record is accessed it is moved to the root of the tree, if it is not already there
- Termed a *splay* operation and occurs on all updates (i.e., insertion, deletion, search)
- If tree T does not contain key k, then splay operation moves k's immediate predecessor or successor to the root
- Ex: *splay*$(8,T)$



double left rotation   single right rotation

IMPLEMENTATION OF SPLAY OPERATIONS

- Use rotations to bring splay value to the root
  1. single rotations (2) raise the splay value one level at a time

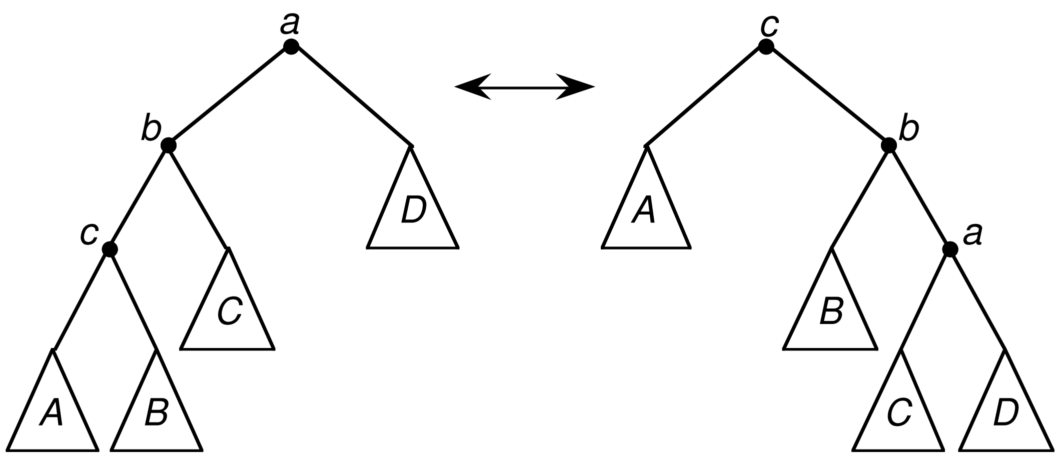


  2. double rotations (4) raise the splay value two levels at a time
  3. triple rotations (8) raise the splay value three levels at a time

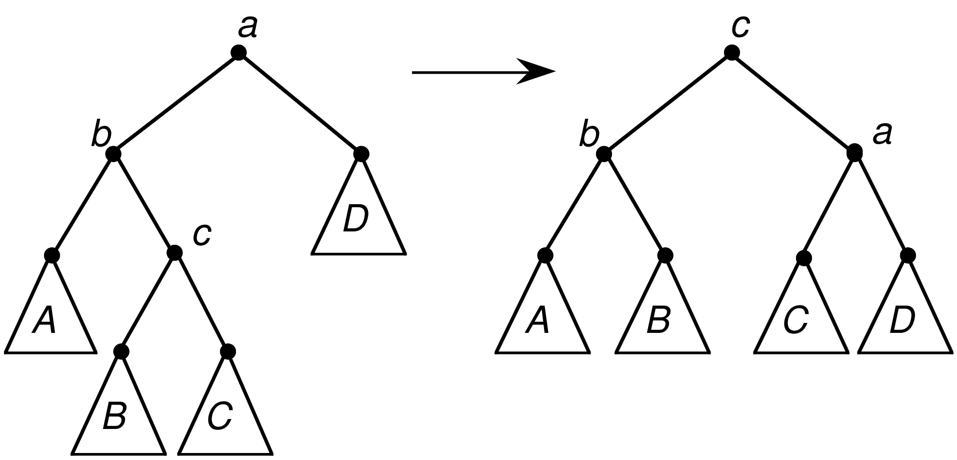
- Use as many double routations as possible
  1. at most one single rotation
  2. enables proof that the splay operation has an amortized cost of $O(\log_2 n)$

DOUBLE ROTATIONS

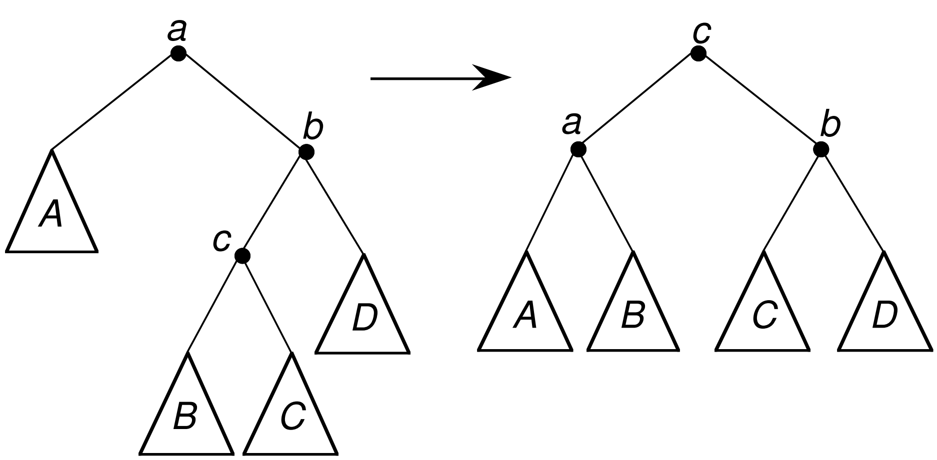
- First two symmetric cases:



- Third case:

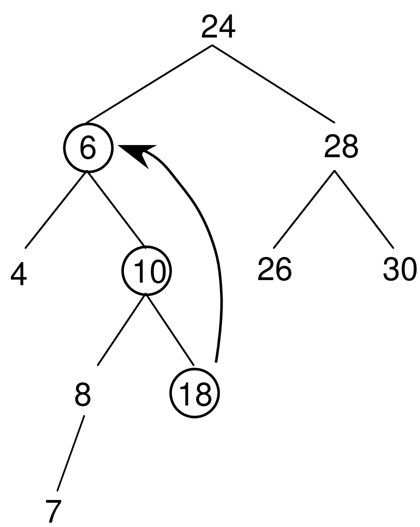


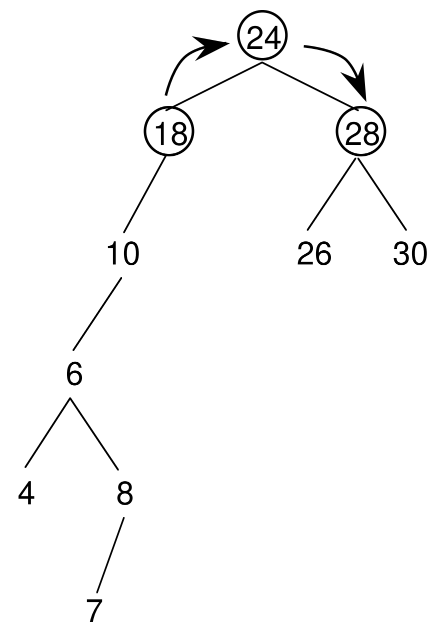
- Fourth case:



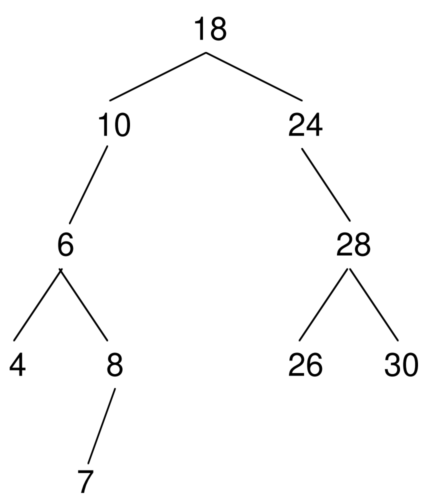
INSERTION INTO A SPLAY TREE

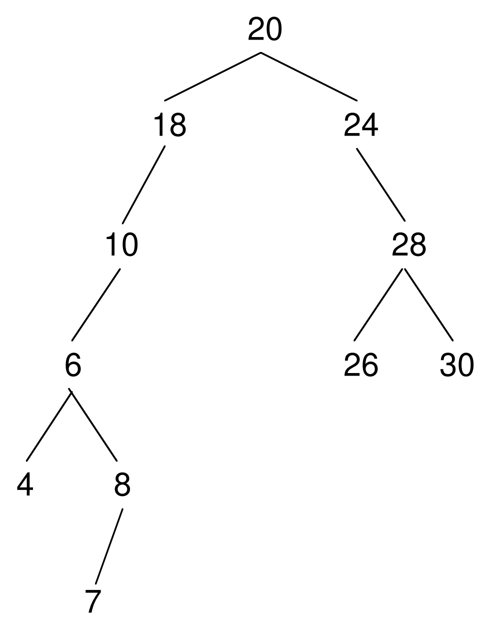
- Assume that *splay*(k,T) yields the immediate predecessor of k
- Ex: insert 20





double left rotation of 18





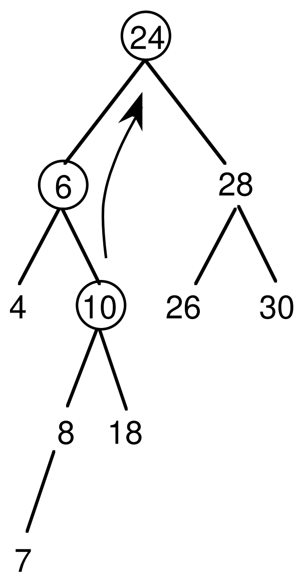
single right rotation of 18

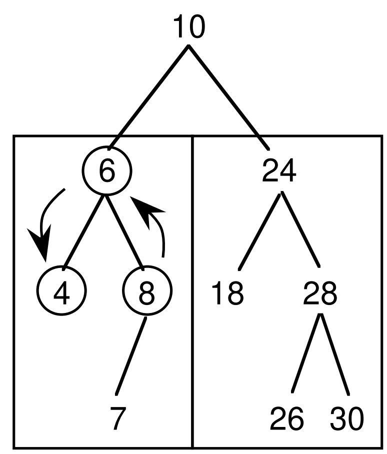
insert 20 between 18 and 24

DELETION FROM A SPLAY TREE

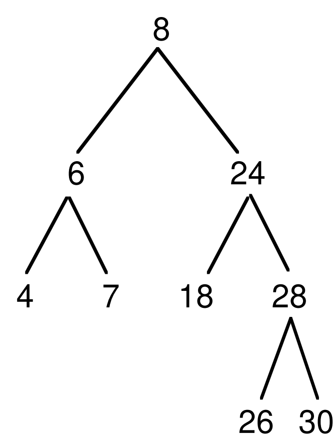
1. Execute *splay*(k,T) yielding T' with L and R as the left and right subtrees, respectively
2. Apply *splay*(∞,L) yielding T'' (equivalent to *splay*(k,L) as $k >$ everything in the left subtree)
3. Make R the right subtree of T''

Ex: delete 10





- splay around 10 via a double right rotation of 10
- splay left subtree around ∞ via a single left rotation of 8



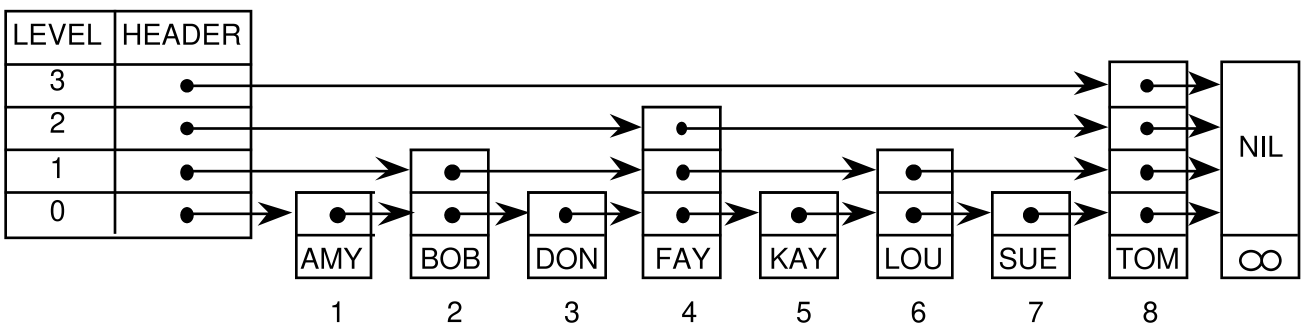
- make the right subtree resulting from the first splay a right subtree of 8

COMPARISON

- Shortcoming of binary search is that we must know size of the dataset and where the partitions start so that we can repeatedly perform the partitions (i.e., know where they start)
  1. can avoid problem by storing dataset in sequential locations
  2. problem is that updates require moving data
     - can avoid by using a linked list, but now can't access dataset at random as is needed to perform the partitions
- Solutions:
  1. binary search tree
     - good expected behavior but bad for certain permutations of the data (e.g., sorted)
  2. AVL trees
     - worst-case logarithmic behavior but need extra storage for balance factors, and rotations are complex
  3. self-adjusting binary search tree
     - amortized logarithmic behavior but high constant factors for primitive operations (splay via rotations)
     - no need for space for balance factors as in AVL trees
  4. hierarchy of linked lists called a *skip list* serves as a compromise between:
     - conventional list (sequential or list implementation)
     - binary search tree (including self-adjusting ones)
     - balanced binary search trees

LIST ARRAY

- Hierarchy of linked lists
  1. level 0 contains the elements of the dataset
  2. level 1 contains pointers to every other element at level 0
  3. level 2 contains pointers to every fourth element at level 0
  4. continue until a list of just one element
- List header is an array with pointers to the start of each level's linked list

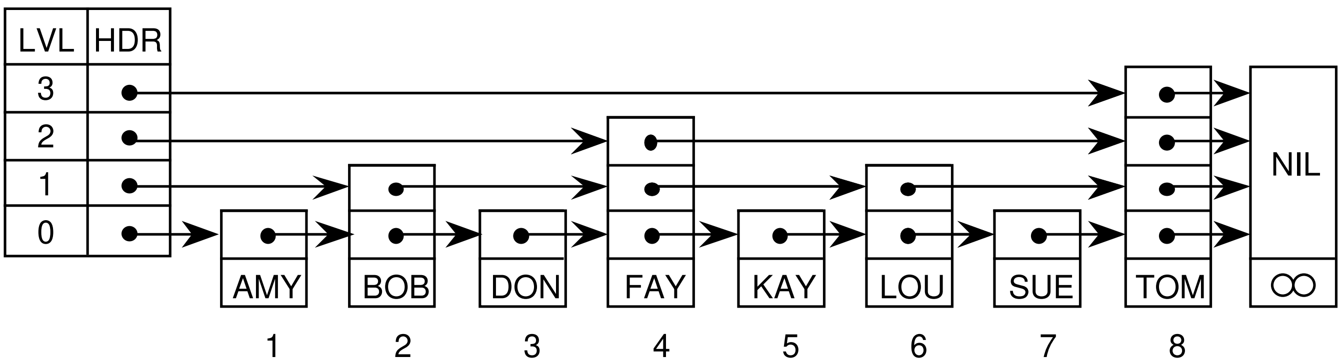


- Close resemblance to a complete binary tree
- Similar to a range tree but some differences:
  1. in the list array, the value of a node at every level corresponds to one of the elements of the dataset, and nodes are linked to each other at every level
  2. in the range tree, values of the elements of the dataset are only stored in the leaf nodes, and only the leaf nodes are linked to each other

SEARCHING FOR *k* IN A LIST ARRAY

1. Start at highest level
2. At each level scan the nodes in sequence until encountering a value ≥ *k*
3. If value equals *k*, then halt
4. Else descend a level and repeat steps 2-4 unless at the lowest level in which case the search is unsuccessful

- Ex: search for RAY



level 3: starting at the list header, TOM at position 8 is greater than RAY, so backup to the list header and descend to level 2

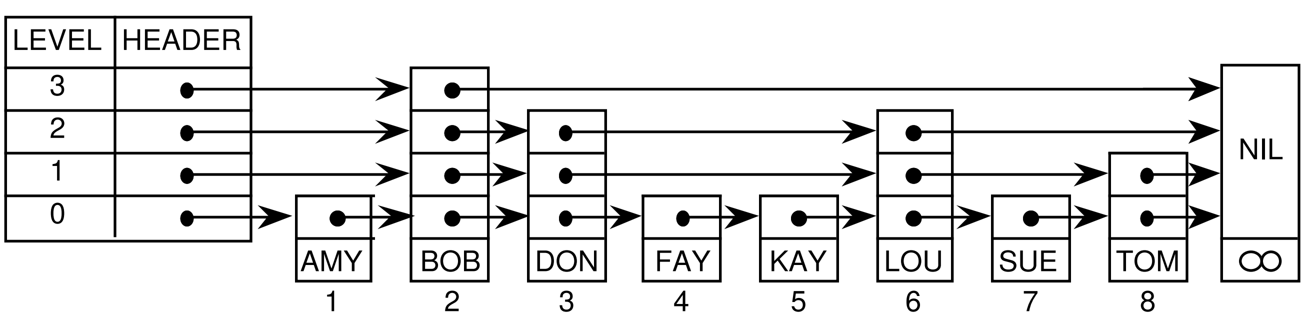
level 2: starting at the list header, FAY at position 4 is less than RAY, but TOM at position 8 is greater than RAY, so backup to position 4 and descend to level 1

level 1: starting at position 4, LOU at position 6 is less than RAY, but TOM at position 8 is greater than RAY, so backup to position 6 and descend to level 0

level 0: starting at position 6, SUE at position 7 is greater than RAY, which means that RAY is not in the list, and we exit with failure

SKIP LIST

- List array
  1. good for searching
  2. insertion and deletion may force its reorganization
     - e.g., if insert before the first element, then must rebuild list structure at level 1 and above
- Skip list
  1. based on idea that there is no need to restrict nodes at successive levels of the list array to skip exactly one node in the list at the immediately lower level
  2. expected performance is the same as long as can guarantee that the expected number of nodes skipped at the immediately lower level is one
  3. data structure
     - a list array where the skip increment for the immediately lower level is generated at random
     - expected behavior is made the same as that of the list array by ensuring that the insertion routine will be $1/p$ times as likely to generate a skip at level i (or depth j) as it is at level i+1 (or depth j–1)

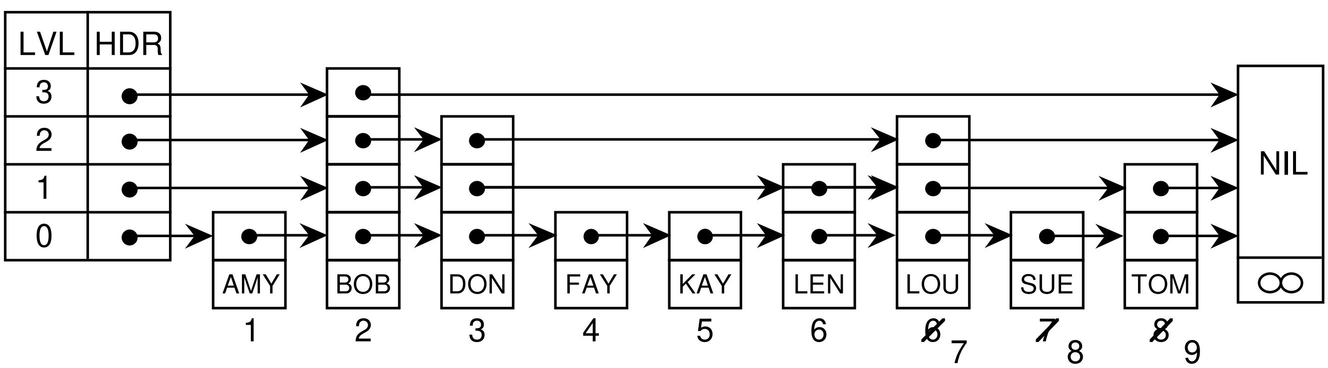


     - not unique as the skips are generated at random

INSERTING *k* INTO A SKIP LIST

1. Search for *k* as in the list array, and remember the search path so that the link fields can be properly set upon the insertion
2. If search is unsuccessful, then found the position at level 0 where *k* is to be inserted
3. Make the insertion
4. Generate a random number *a* in [0,1)
5. If $a < 0.5$, then exit
6. Else if $a \geq 0.5$, then ascend a level, say to $i+1$, and insert *k*
7. Repeat steps 4–6 as is appropriate

- Ex: insert LEN assuming a probability of 0.5 of ascending a level upon an insertion



1. search for LEN and insert it between KAY and LOU
   - set the level 0 link field of KAY to LEN
   - set the level 0 link field of LEN to LOU
   - remember the level 1 and 2 link fields of DON and the level 3 link field of BOB which are on the path to LEN
2. generate a random number, say 0.7, which is >0.5 and thus insert LEN at level 1
   - update level 1 link of DON to point at LEN
   - update level 1 link of LEN to point at LOU
3. generate a random number, say 0.4, which is <0.5 and stop the insertion

SKIP LIST ANALYSIS BASICS

- Assumptions:
  1. n records
  2. probability p of ascending a level upon insertion
  3. set a bound b on the number of levels
  4. b is the level where the expected number of nodes is 1
     - b is $\log_{1/p} n$
     - obtained by observing that probability of having a node at level i is p^i and solving $n \cdot p^i = 1$ for i
- Expected cost, in terms of number of nodes examined, of search, insertion, and deletion is proportional to $\log_{1/p} n$

SKIP LIST ANALYSIS MECHANICS

1. Show expected length of path from the root to reach a node at level 0 is proportional to $\log_{1/p} n + c$, where c is a constant
2. Trace search path backwards from a node at level i and compute its expected length $C(j)$ as climb j levels in an infinite list
   - $C(j)$ is the expected number of transitions that are made
   - two possibilities:
     a. proceed backwards to a node at level i, from where j levels must still be climbed
     b. climb up one level to level i+1 from where j–1 levels must still be climbed
   - solve recurrence relation
     $C(j) = (1-p) \cdot (1+C(j)) + p \cdot (1+C(j-1))$ with $C(0)=0$
     yielding $C(j) = j/p$
3. Infinite list assumption is a worst-case assumption
   - if list is not infinite, then as soon as encounter list header, continue climbing up without any leftward moves
   - cost to go from level 0 to $\log_{1/p} n$ is $(\log_{1/p} n)/p$
   - expected number of left moves remaining at level $\log_{1/p} n$ is bounded by 1
   - move up from level $\log_{1/p} n$ to expected maximum level
     a. probability of climbing exactly one more level: $(1-p) \cdot p$
        - probability p of climbing one level
        - probability $(1-p)$ of climbing no more levels
     b. expected number of levels to be climbed is
        $\sum_{i=1}^{\infty} i \cdot (1-p) \cdot p^i = p/(1-p)$
4. Total cost:
   $$(\log_{1/p} n)/p + 1 + p/(1-p) = (\log_{1/p} n)/p + 1/(1-p)$$

CHOOSING P IN A SKIP LIST IMPLEMENTATION

- Expected number of pointers per node is $1/(1-p)$
  1. probability of exactly one pointer is $1-p$
  2. probability of exactly two pointers is $(1-p) \cdot p$
  3. probability of exactly j pointers is $(1-p) \cdot p^{j-1}$
  4. expected number of pointers per node is
     $$\sum_{i=1}^{\infty} i \cdot (1-p) \cdot p^{i-1} = \frac{1-p}{p} \cdot \sum_{i=1}^{\infty} i \cdot p^{i} = \frac{1}{1-p}$$
- Let $s=1/p$ denoting how many nodes are skipped on the average at each level in the skip list
- Cost of searching a skip list of n records is $s \log_s n + 1$ assuming that search is started at level where the expected number of nodes is 1
- Cost is minimized for $s = e$ and is the same for $s=2$ and $s=4$ corresponding to probabilities of 1/2 and 1/4, respectively
- Usually use $s=2$ and $s=4$ as they make it easy to generate the random number for the insertion process
  1. can just pick $\log_2 s$ bits at a time from a randomly generated stream of bits
  2. as s increases, less space is needed as average number of pointers is $s/(s-1)$, and thus $s=4$ is a good choice

SUMMARY OF SKIP LISTS

1. Advantage over binary search trees is that worst-case cost is independent of the data
2. Worst-case cost of binary search tree arises when the data is sorted
3. Worst-case cost of skip list depends on the tree generation process
   - arises when all skip lists with exception of the one at the lowest level are empty
   - search will take n steps
   - assuming p=1/2, probability of all insertions at the lowest level is $1/2^n$ — a rare event
   - bad case does not depend on the values of the keys
   - there are no bad datasets — just bad random number generator sequences
   - no guarantee against worst-case performance using skip lists
4. Worst-case of skip list is unknown (based on result of a random number generator) and unlikely to occur, instead of being known, undesirable, and far more likely to occur than its probability of occurring were it generated at random (as is the case with binary search trees)

EXTERNAL SEARCHING

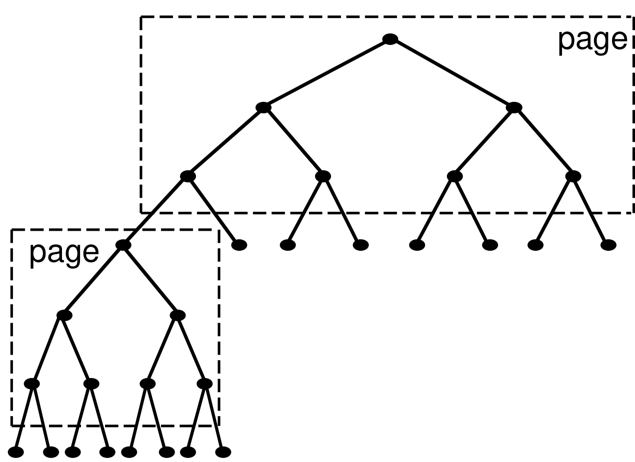
1. When data volume is high, tree is too large to fit in memory

2. Binary search tree is stored on disk pages

3. Branches in a binary search tree contain disk addresses instead of links to other nodes

4. Each branch corresponds to a disk seek operation
   - *n* records imply $\log_2 n$ disk accesses
   - $n = 2$ million implies 21 disk accesses

5. Solution: aggregate results of the comparisons by delaying the branch to the missing page
   - Ex: form groups of 7 key values at every 3 levels of the tree

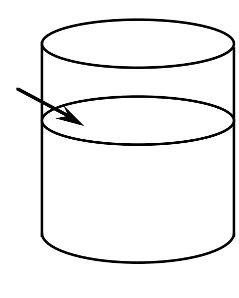


   - search is three times as fast as only need 1/3 as many disk accesses
   - basis of indexed-sequential file organization (ISAM) – 3 level tree

     a. level 1 = cylinder

     b. level 2 = track

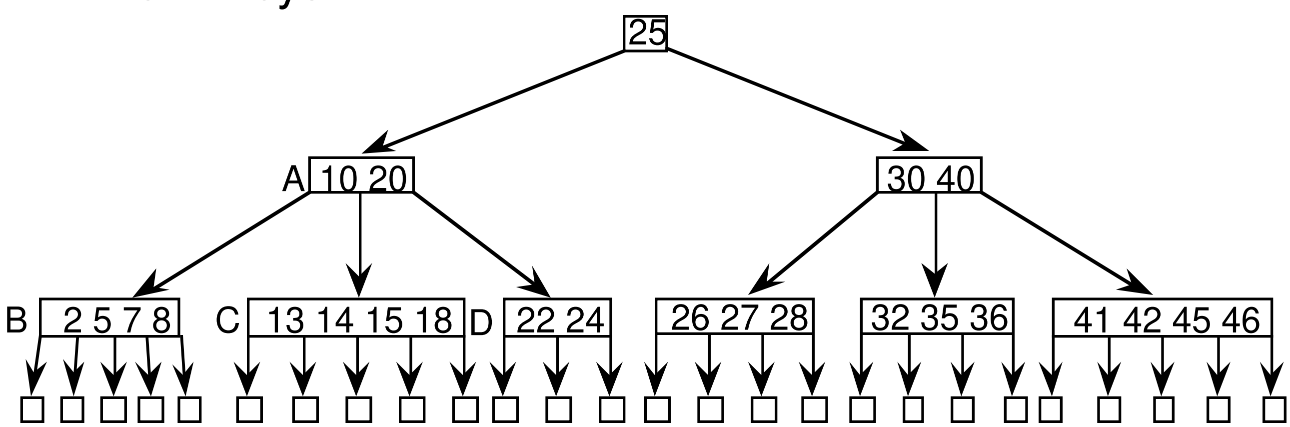
     c. level 3 = pointer to record



B-TREES

- A B-tree of order m has the following properties:
  1. every node has $\leq m$ sons
  2. every node except for the root and leaves has $\geq \lceil m/2 \rceil$ sons
  3. the root has at least 2 sons (unless it is a leaf)
  4. all leaves appear at the same level and carry no keys
     - usually omitted from the drawings
  5. a non-leaf node with k sons contains k–1 keys
- Guarantee that each node of size m is at least 50% full

  Ex: m=5 with 3 levels
  all nodes with the exception of the root contain 2, 3, or 4 keys



A non-terminal node contains j keys and j+1 pointers:

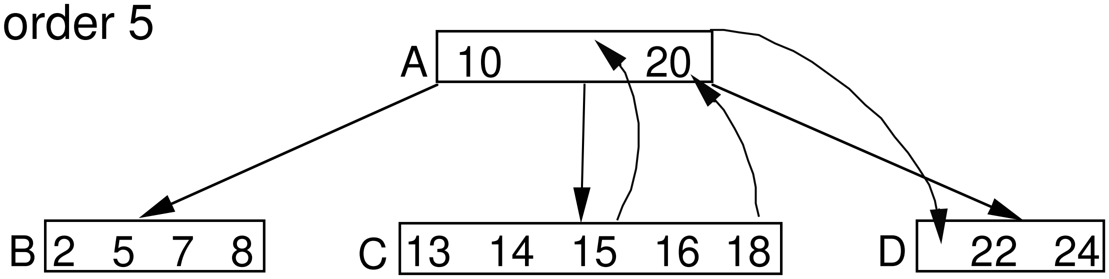
P_0 K_1 P_1 K_2 P_2 K_3 K P_{j-1} K_j P_j

$K_1 < K_2 < K_3 <$ K $< K_j$

- m is usually the size of a page and a 1–1 correspondence between nodes and pages
- Search in a B-tree is similar to that in a binary search tree

B-TREE INSERTION

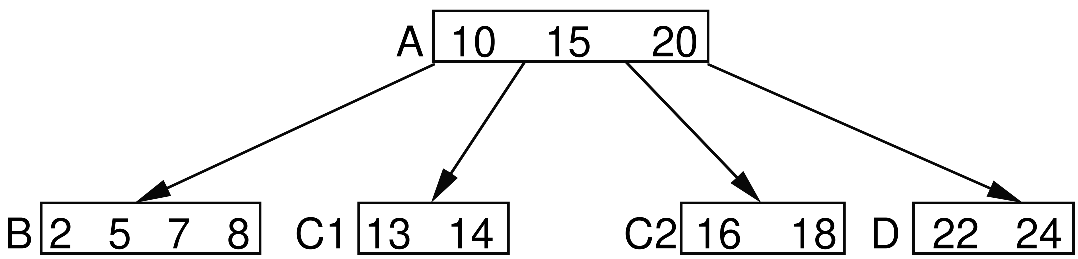
Ex: order 5



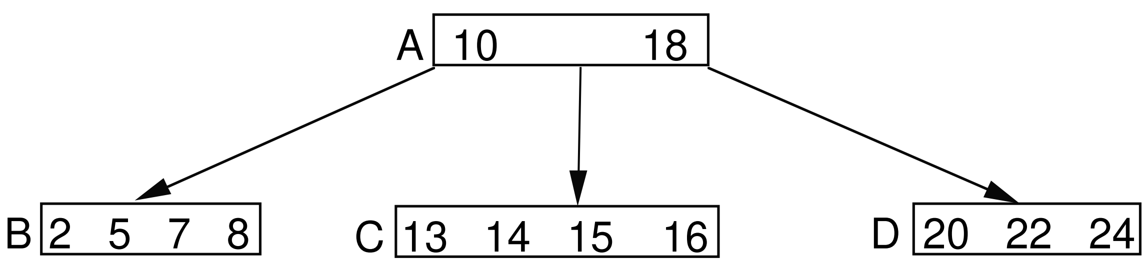
Insert 16

Node C becomes too full (overflow)

Therefore, split node C and promote 15



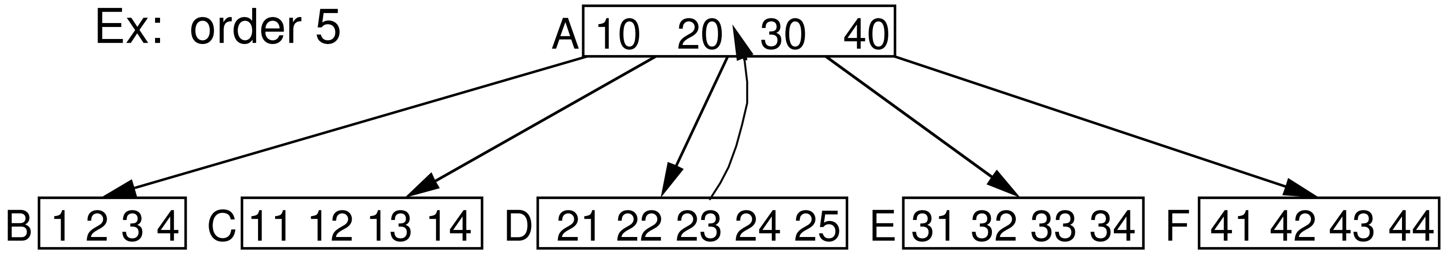
Alternatively, we can apply rotation (i.e., promote 18 to replace 20 in node A and demote 20 to node D)



Note that we cannot promote 13 and rotate 10 to the left (to node B) because node B is full

HOW DO B-TREES GROW?

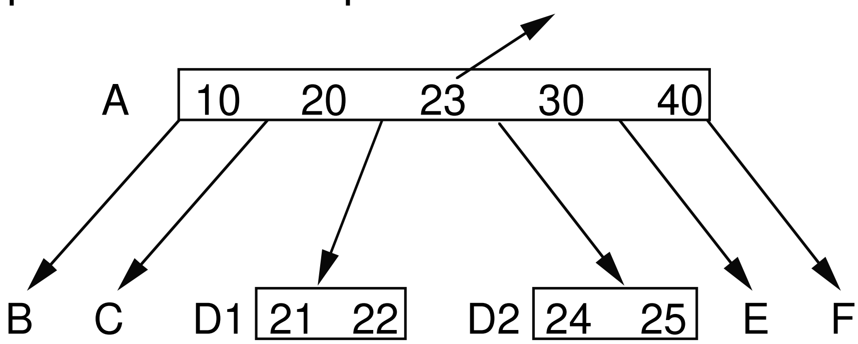
Ex: order 5



Insert 24

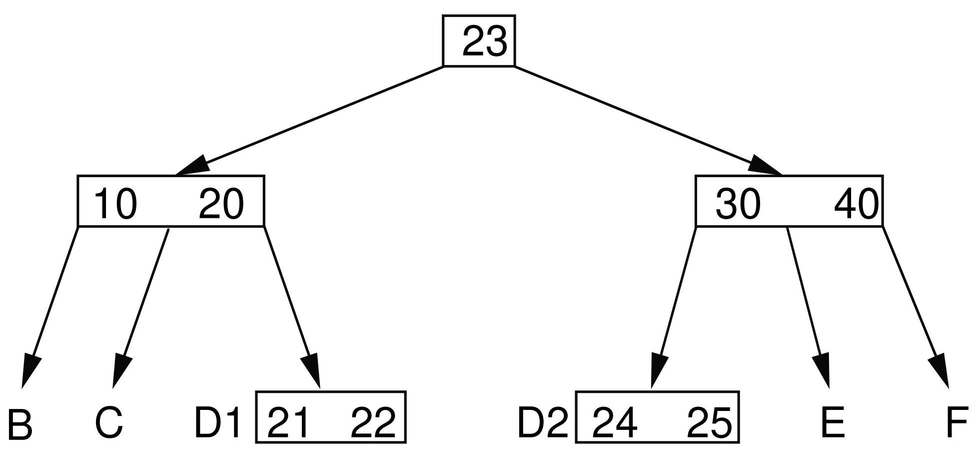
Node D becomes too full (overflow)

Therefore, split node D and promote 23 to node A



Now, node A is too full

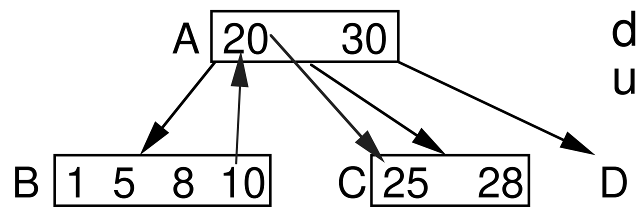
Split node A and promote 23 creating a new node and the tree has grown by one level



B-TREE DELETION

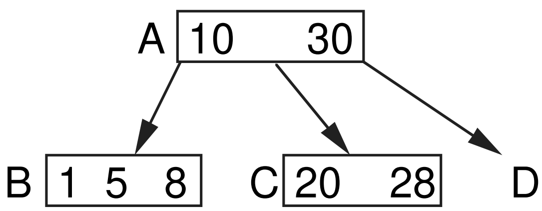
1. To delete a record *R* with key *K* start at the root node and search for *K*
2. Once found, locate the next record *S* with smaller or larger key (in terminal node *N*)
3. Replace *R* with *S* and delete *S* from *N*
4. If *N* contains < $\lceil m/2 \rceil$ records, then underflow

Ex: order 5

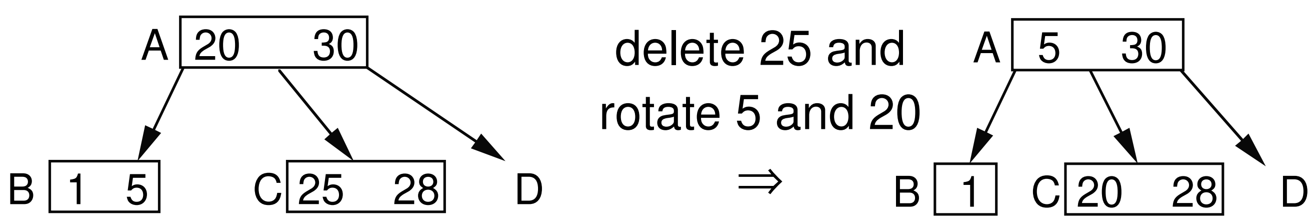


deletion of 25 causes underflow in node C

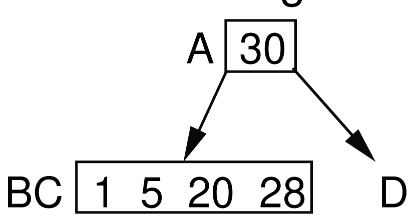
Promote 10 from B to A and demote 20 to C



Problem: Suppose there is underflow in adjacent nodes?



Solution: Merge nodes B and C

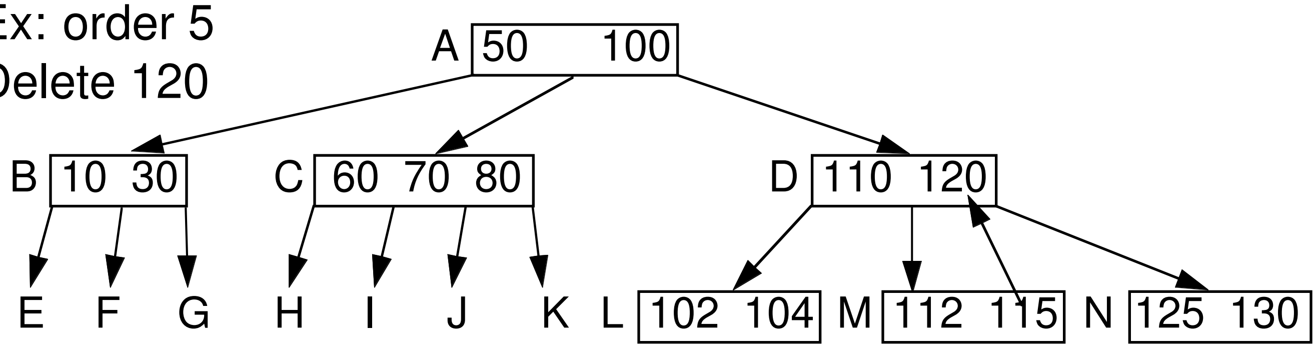


The same algorithm is recursively applied to node A which may underflow if it is not a root node

COMPLEX B-TREE DELETION

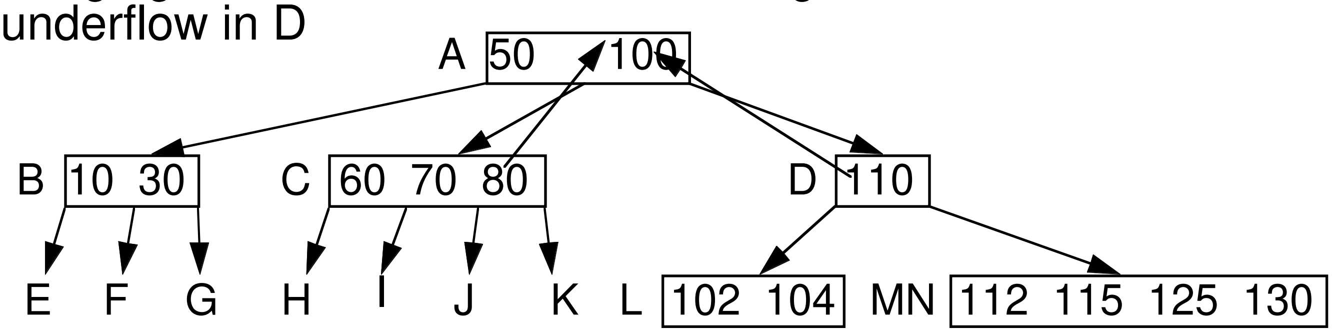
Ex: order 5

Delete 120

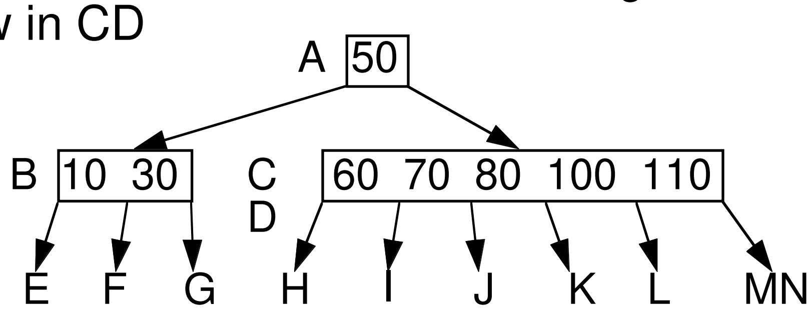


Replacing 120 with 115 causes node M to underflow

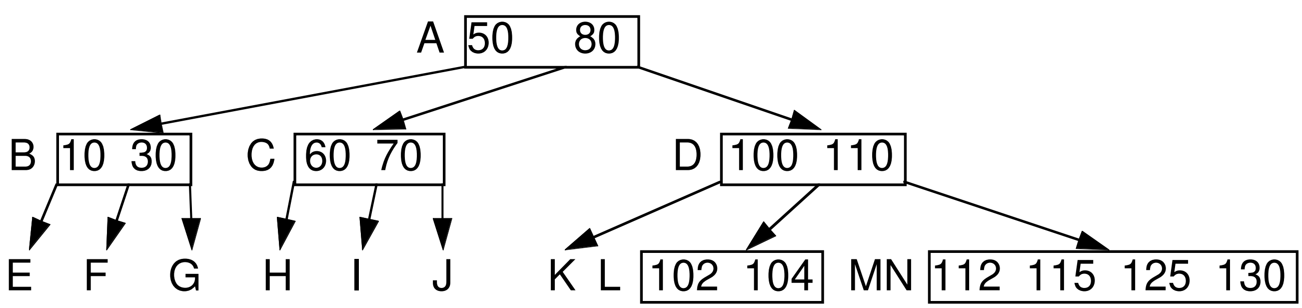
Merging nodes M and N and demoting 115 leads to underflow in D



Merging nodes C and D and demoting 100 leads to overflow in CD



Splitting CD and promoting 80 leads to the desired result



Note: rotation could also have been applied (promote 80 and demote 100)

SUMMARY OF B-TREES

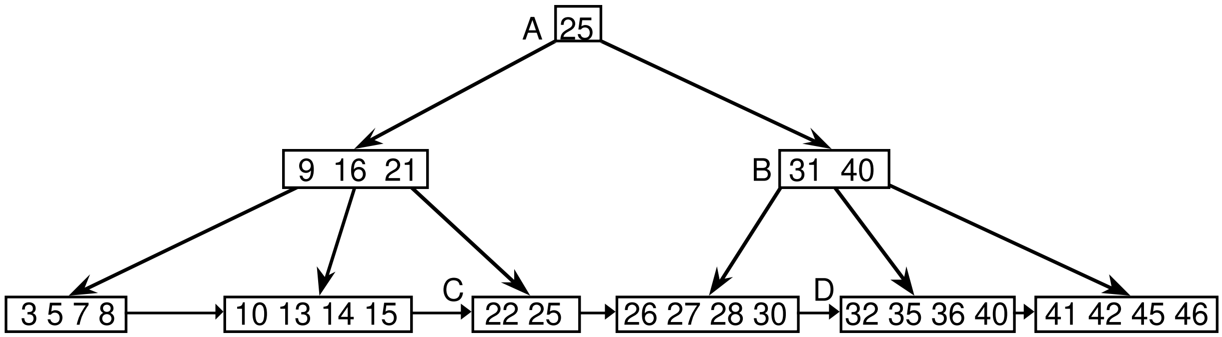
Advantages of B-trees

- Storage utilization is at least 50% and often much better
- Storage is allocated and released as the file grows and shrinks
- No congestion problem as is common with hashing
- The natural order among the keys is maintained and processing along that order is possible (unlike hashing)
- If requests are batched, then can use a prepaging scheme after sorting the transactions on their keys

Variations

- Eliminate wasted space in terminal nodes
- Increase the utilization of non-terminal nodes by only storing the key there instead of the entire record (e.g., B+-tree)
- Use different values of m for each level
- Improve storage utilization to $n/(n+1) \cdot 100\%$ by stipulating that when node splitting occurs, n adjacent nodes are split to create n+1 nodes.
- Currently, one node is split to create 2 nodes.

B+-TREES



1. Limit information stored in nodes to be just a key value and a pointer to a node that contains the rest of the record's data

2. No need for empty leaf nodes

3. Only nodes at the deepest level (called *leaf nodes*) contain key values
   - contain between $\lceil m/2 \rceil -1$ and $m-1$ key values
   - space for pointers to leaf nodes can now be used for pointers to nodes containing the rest of the data associated with the key

4. Nonleaf nodes are used just to provide an index to enable locating the leaf node containing the desired record
   - at times, nonleaf nodes may contain some of the same key values that are stored in the leaf nodes, but this need not be so
   - Ex: 25 appears in nonleaf node A and leaf node C, while 40 appears in nonleaf node B and leaf node D

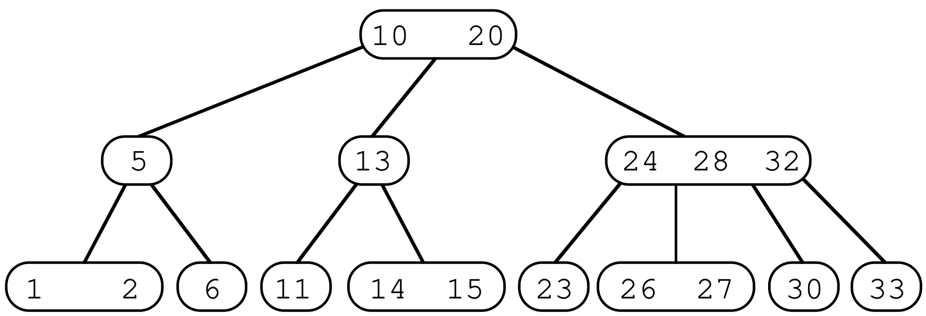
5. Can link up all leaf nodes in the tree to enable traversal without ever accessing the nonleaf nodes
   - efficient support of both random and sequential access

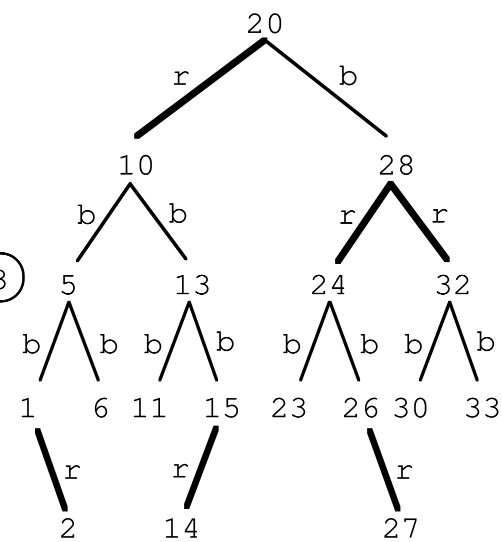
PREFIX B+-TREE

- When keys are long strings, nonleaf nodes only need to store enough characters to enable differentiating between the key values in the subtrees
- To differentiate between ‘jefferson’ and ‘jones’, ‘je’ is enough rather than ‘jefferson’ which is longer

RED-BLACK TREES

- B-tree is a generalization of a 2-3 tree where each node has 2 or 3 sons — that is, an order 3 B-tree
- 2-3-4 tree: each node contains between 1 and 3 key values (i.e., 2 to 4 sons)
- Red-black trees enable the implementation of 2-3 and 2-3-4 trees as binary trees in a manner analogous to the natural correspondence between a forest and a binary tree



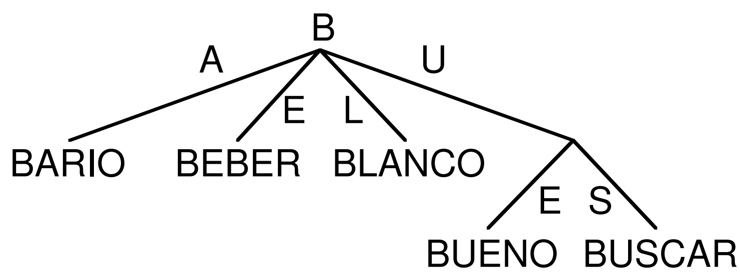


1. black edges connect nodes of the 2-3 and 2-3-4 trees
2. nodes of the 2-3 and 2-3-4 tree are represented as one level binary trees consisting of red edges
3. implementation needs an additional bit to indicate if the incoming edge from the father is red or black

- Observations
  1. 2-3 tree: one-level binary tree has at most 1 red edge
  2. 2-3-4 tree: one-level binary tree has at most 2 red edges
  3. impossible to have 2 successive red edges in a 2-3 or 2-3-4 tree

TRIE-BASED OR DIGITAL SEARCHING

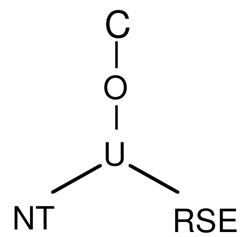
- Uses a trie structure
  1. digits or characters that comprise the domain of the key values control the branching process
  2. each character has M possible values
  3. a node at depth i in the trie represents an M-way branch depending on the i^{th} character or digit
- Analogous to a "thumb-index" in a dictionary
- Ex:



- Organizes the embedding space from which the data is drawn rather than the data itself as is the case in tree-based searching
- For n records and M-valued digits, $\log_M n$ digit positions must be examined during a random search
- Used in command decoders and file systems to recognize incompletely specified commands and file names
- Ex: LO$ yields LOGIN DI$ yields DIRECTORY

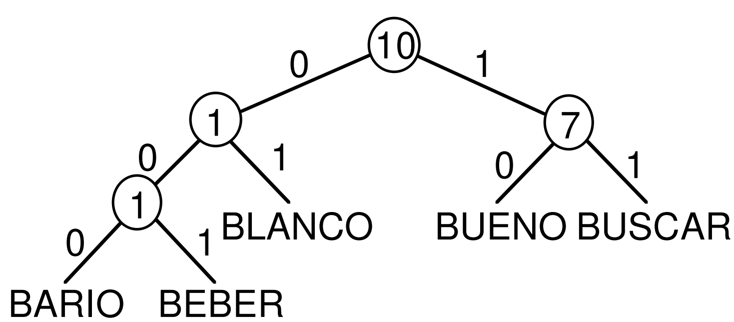
PATRICIAN TRIES

- Practical Algorithm to reTRieve Information Coded In Alphanumeric
- Digital searching can be made more efficient when 1-way branches are avoided



- Instead of an *M*-way branch on the digits, use a 2-way branch on the bits comprising the digits
- Use a SKIP field with each branch node to indicate which bit is to be tested next (i.e., how many bits are to be skipped)
- Ex: use ASCII codes

Word	Letter 1	Letter 2	Letter 3	Letter 4	Letter 5	Letter 6
BARIO	1000010	1000001	1010010	1001001	1001111	
BEBER	1000010	1000101	1000010	1000101	1010010	
BLANCO	1000010	1001100	1000001	1001110	1000011	1001111
BUENO	1000010	1010101	1000101	1001110	1001111	
BUSCAR	1000010	1010101	1010011	1000011	1000001	1010010



1. skip to bit position 10 and test
2. skip to bit position 10+1 on the left and 10+7 on the right
3. skip to bit position 10+1+1 on the left of the left

ADVANTAGES OF PATRICIAN TRIES

1. Standard node size
2. Only examine differences between nodes (no 1-way branches)
3. Good for long keys — e.g., book titles, etc.
4. Data structure is analogous to a program with an interpreter

LITERARY APPLICATION OF DIGITAL SEARCHING

- Ex: English dictionary
  1. 100,000 words
  2. only 309 of the possible $26^2=676$ initial 2-letter combinations are used
  3. only store stems, prefixes, and suffixes of words
- Lookup algorithm:
  1. search for the word
  2. if not found, then identify a prefix or suffix
  3. remove prefix and/or suffix and lookup the word
- Caution must be used not to misidentify the suffixes and prefixes as this may change the meaning of the word