Chapter Sixteen

B-Trees and Related Works

B-Trees

- Large degree B-trees used to represent very large dictionaries that reside on disk.
- Smaller degree B-trees used for internal-memory dictionaries to overcome cache-miss penalties.

AVL Trees

- $n = 2^{30} = 10^9$ (approx).
- $30 <= height <= 43$.
- When the AVL tree resides on a disk, up to 43 disk access are made for a search.
- This takes up to (approx) 4 seconds.
- Not acceptable.

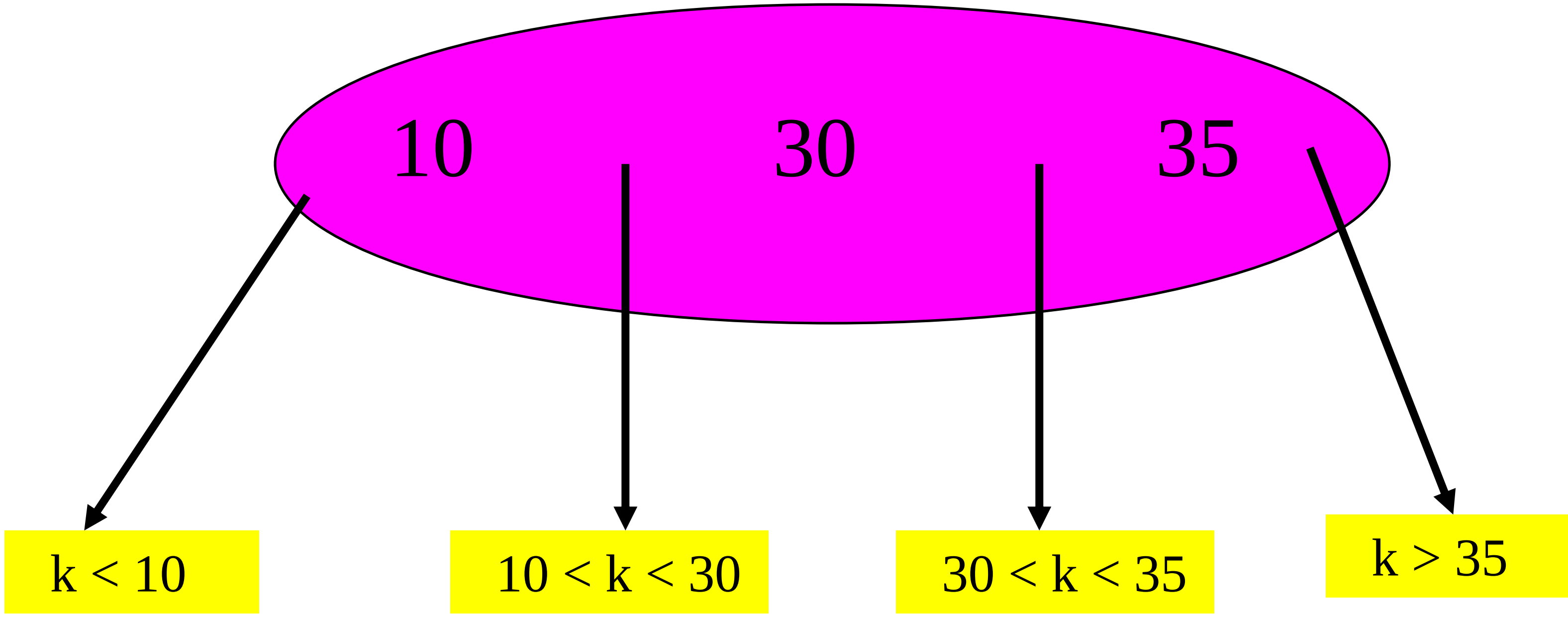
Red-Black Trees

- $n = 2^{30} = 10^9$ (approx).
- 30 <= height <= 60.
- When the red-black tree resides on a disk, up to 60 disk access are made for a search.
- This takes up to (approx) 6 seconds.
- Not acceptable.

m-way Search Trees

- Each node has up to $m - 1$ pairs and m children.
- $m = 2$ => binary search tree.

4-Way Search Tree



Maximum # Of Pairs

- Happens when all internal nodes are m-nodes.
- Full degree m tree.
- # of nodes $= 1 + m + m^2 + m^3 + \ldots + m^{h-1}$
  $= (m^h – 1)/(m – 1)$.
- Each node has $m – 1$ pairs.
- So, # of pairs $= m^h – 1$.

Capacity Of m-Way Search Tree

	m = 2	m = 200
h = 3	7	$8 * 10^{6} - 1$
h = 5	31	$3.2 * 10^{11} - 1$
h = 7	127	$1.28 * 10^{16} - 1$

Definition Of B-Tree

- Definition assumes external nodes (extended m-way search tree).
- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

2-3 And 2-3-4 Trees

- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

- 2-3 tree is B-tree of order 3.
- 2-3-4 tree is B-tree of order 4.

B-Trees Of Order 5 And 2

- B-tree of order m.
  - m-way search tree.
  - Not empty => root has at least 2 children.
  - Remaining internal nodes (if any) have at least ceil(m/2) children.
  - External (or failure) nodes on same level.

- B-tree of order 5 is 3-4-5 tree (root may be 2-node though).
- B-tree of order 2 is full binary tree.

Minimum # Of Pairs

- n = # of pairs.
- # of external nodes $= n + 1$.
- Height $= h$ => external nodes on level $h + 1$.

level	# of nodes
1	1
2	>= 2
3	>= 2*ceil(m/2)
h + 1	>= $2*\text{ceil}(m/2)^{h-1}$

$$n + 1 >= 2*\text{ceil}(m/2)^{h-1},\ h >= 1$$

Minimum # Of Pairs

$$n + 1 >= 2*ceil(m/2)^{h-1}, h >= 1$$

- m = 200.

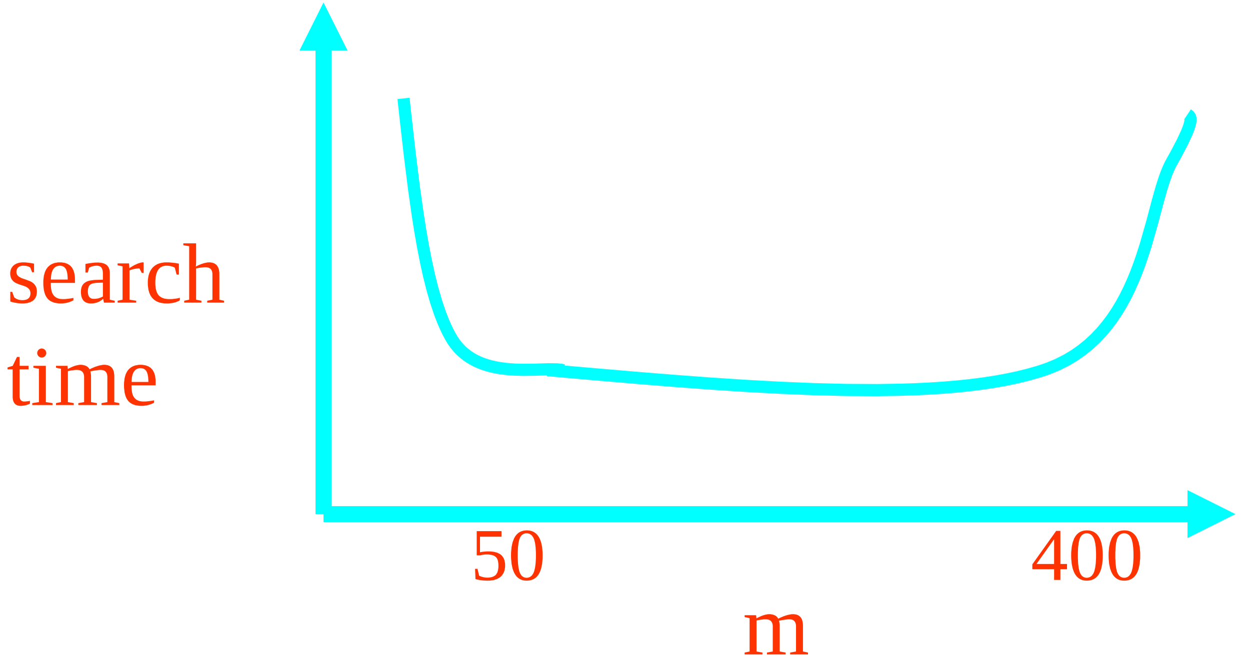
height	# of pairs
2	$>= 199$
3	$>= 19,999$
4	$>= 2 * 10^6 - 1$
5	$>= 2 * 10^8 - 1$

$$h <= \log_{ceil(m/2)} [(n+1)/2] + 1$$

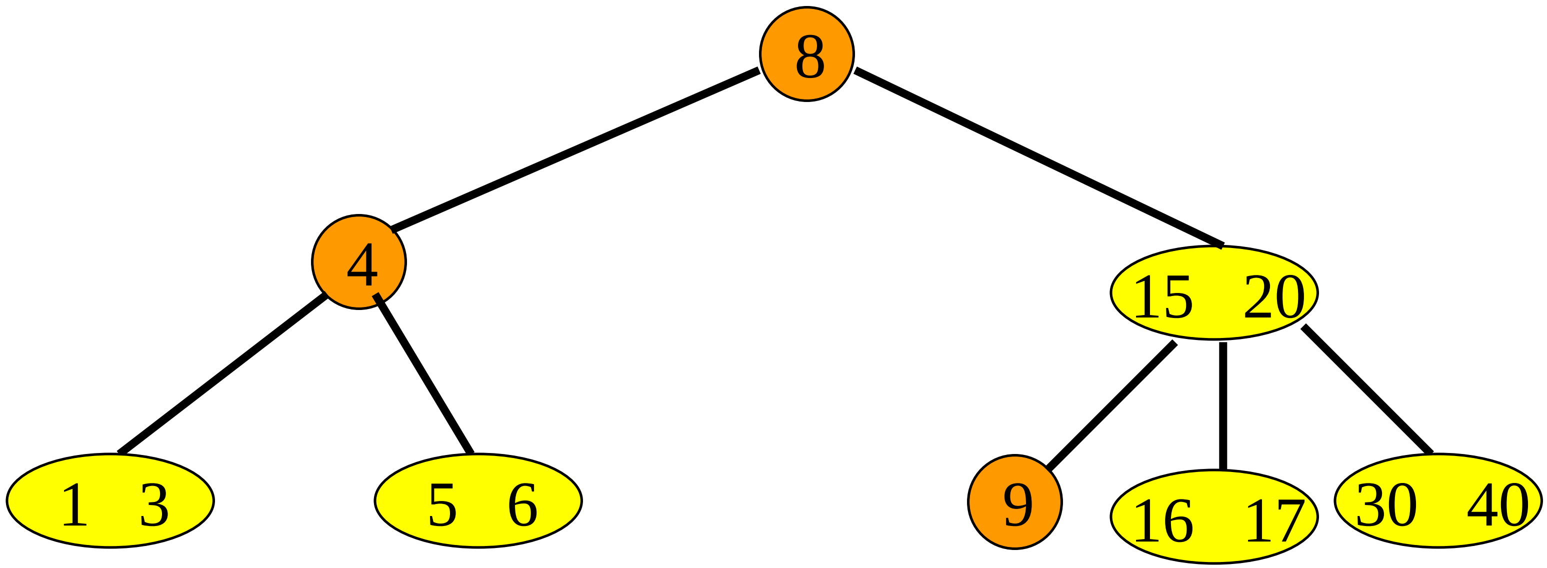
Choice Of m

- Worst-case search time.
  - (time to fetch a node + time to search node) * height
  - $(a + b*m + c * \log_2 m) * h$

    where a, b and c are constants.



Insert



Insertion into a full leaf triggers bottom-up node splitting pass.

Split An Overfull Node

$m\ a_0\ p_1\ a_1\ p_2\ a_2\ \ldots\ p_m\ a_m$

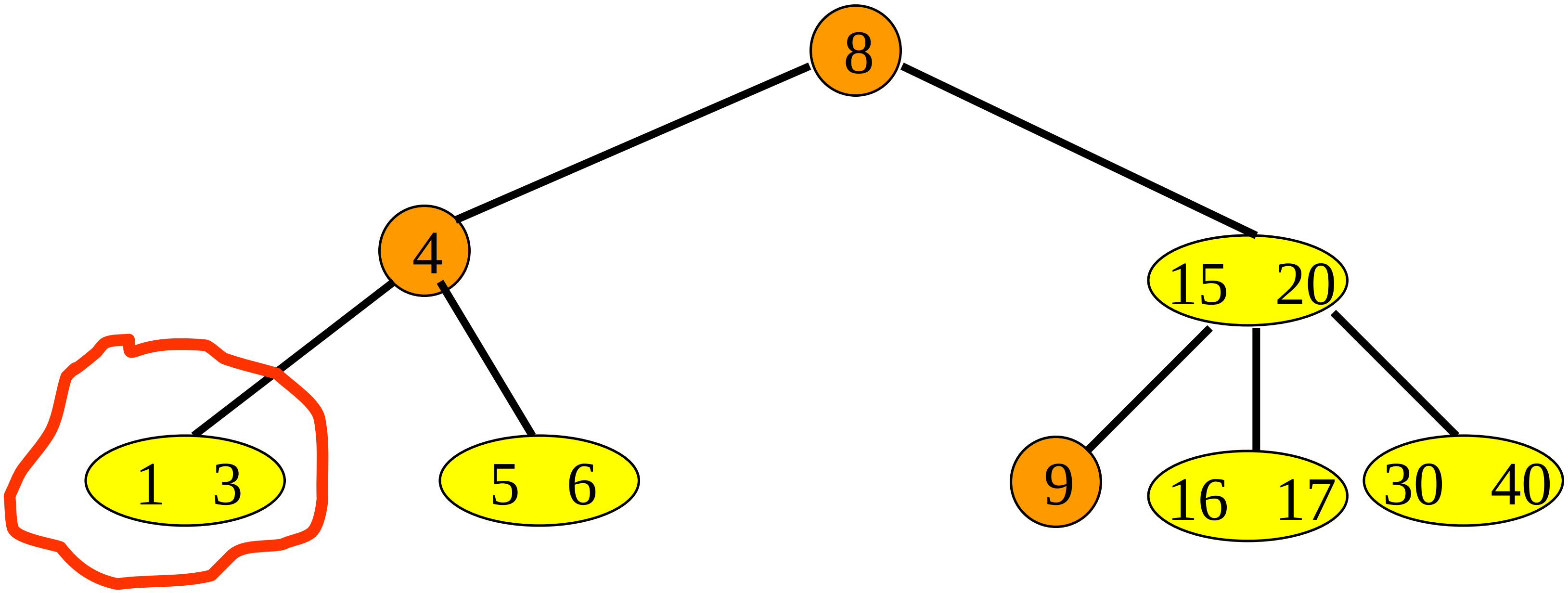
- a_i is a pointer to a subtree.
- p_i is a dictionary pair.

$ceil(m/2)-1\ a_0\ p_1\ a_1\ p_2\ a_2\ \ldots\ p_{ceil(m/2)-1}\ a_{ceil(m/2)-1}$

$m-ceil(m/2)\ a_{ceil(m/2)}\ p_{ceil(m/2)+1}\ a_{ceil(m/2)+1}\ \ldots\ p_m\ a_m$

- $p_{ceil(m/2)}$ plus pointer to new node is inserted in parent.

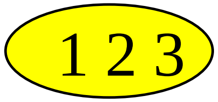
Insert



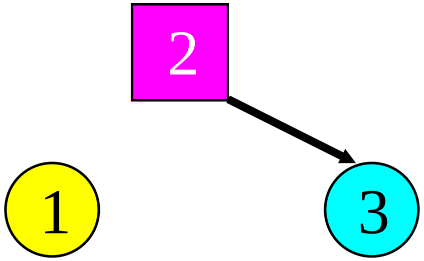
- Insert a pair with key = 2.
- New pair goes into a 3-node.

Insert Into A Leaf 3-node

- Insert new pair so that the 3 keys are in ascending order.

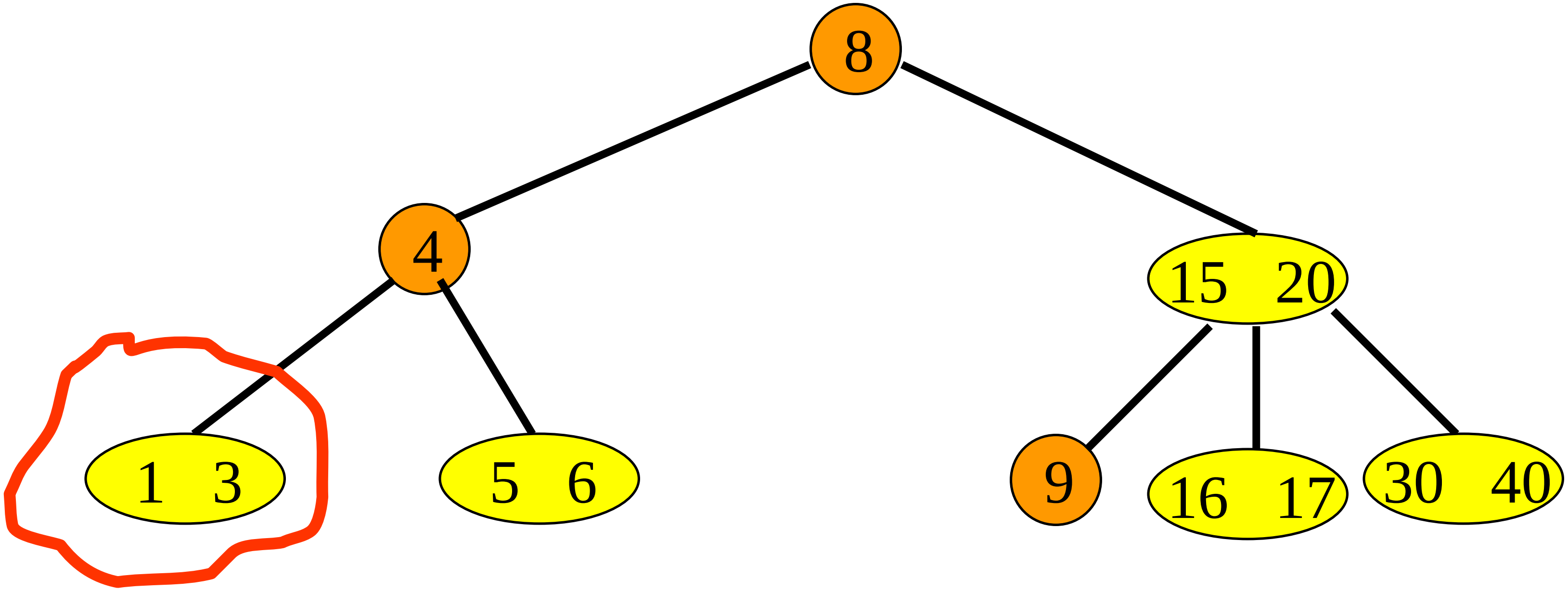


- Split overflowed node around middle key.



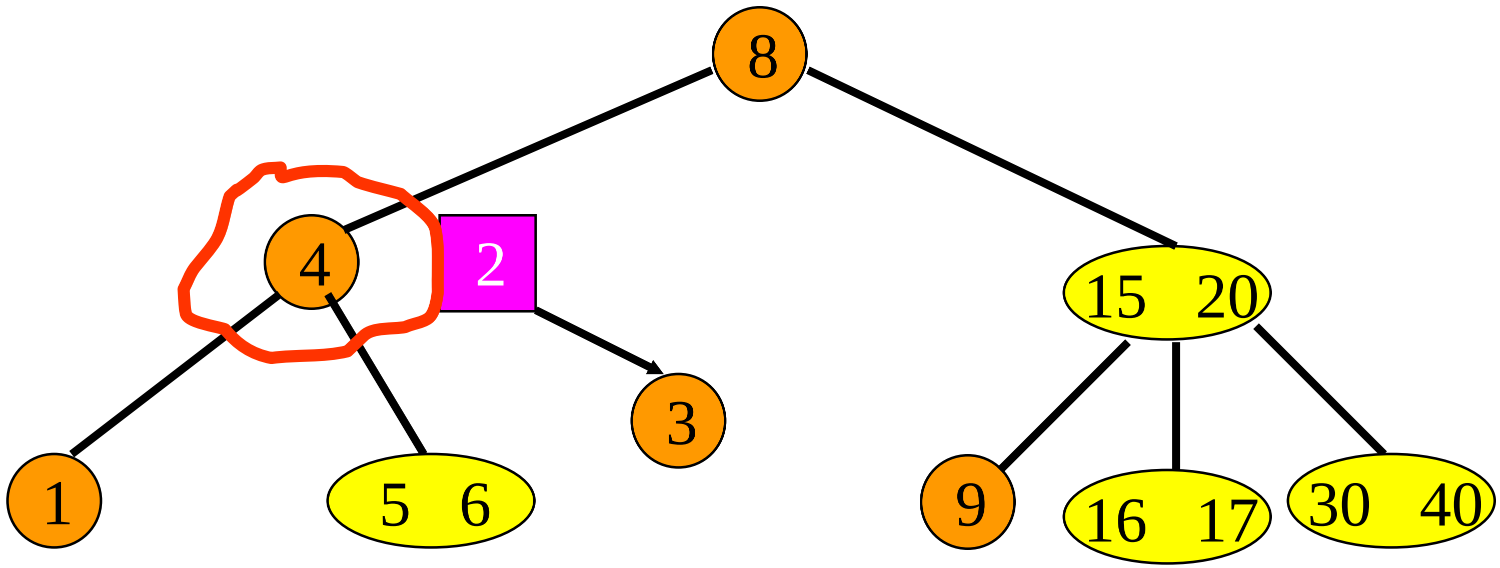
- Insert middle key and pointer to new node into parent.

Insert



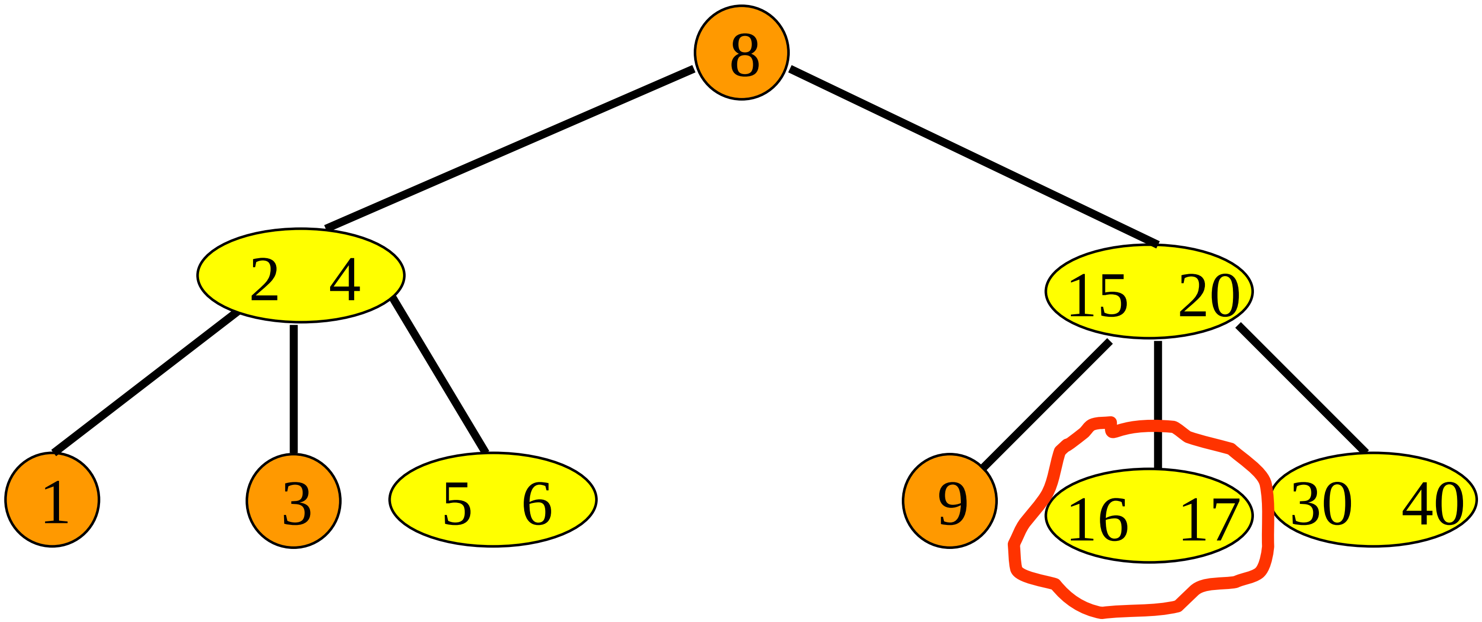
- Insert a pair with key = 2.

Insert



- Insert a pair with key = 2 plus a pointer into parent.

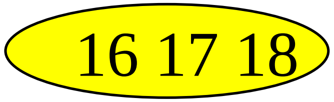
Insert



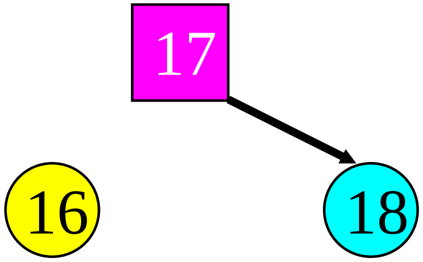
- Now, insert a pair with key = 18.

Insert Into A Leaf 3-node

- Insert new pair so that the 3 keys are in ascending order.

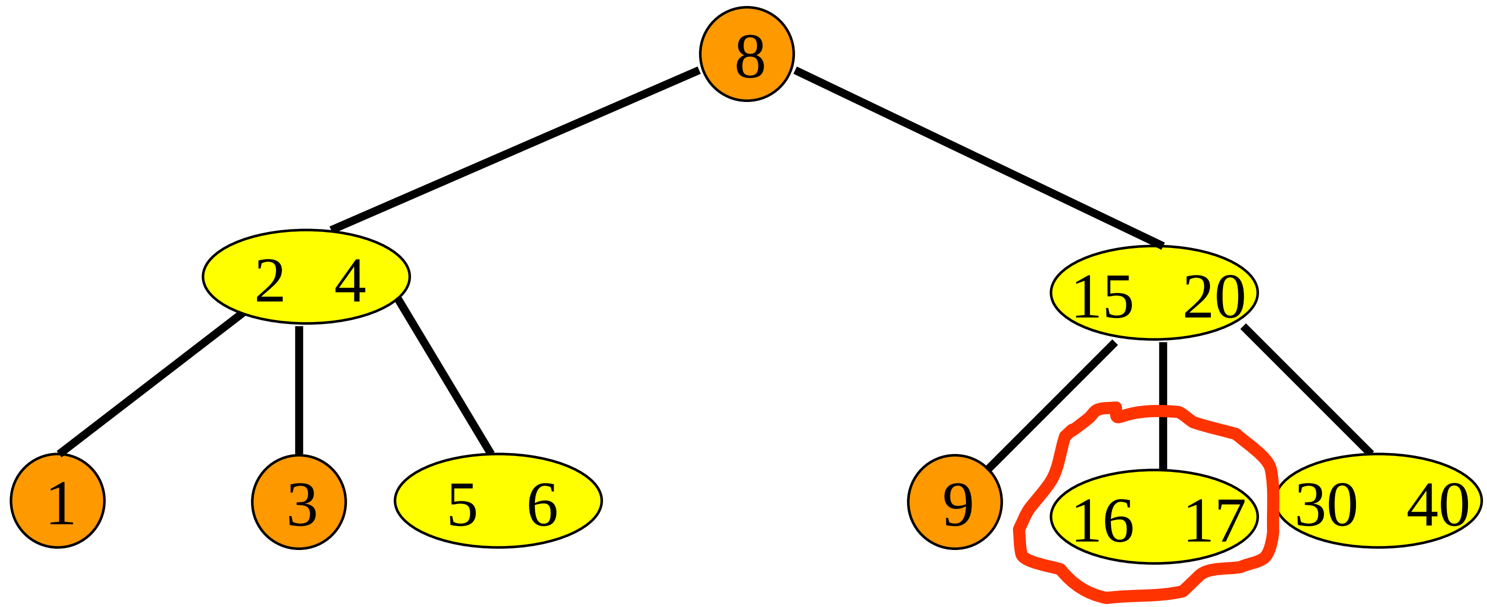


- Split the overflowed node.



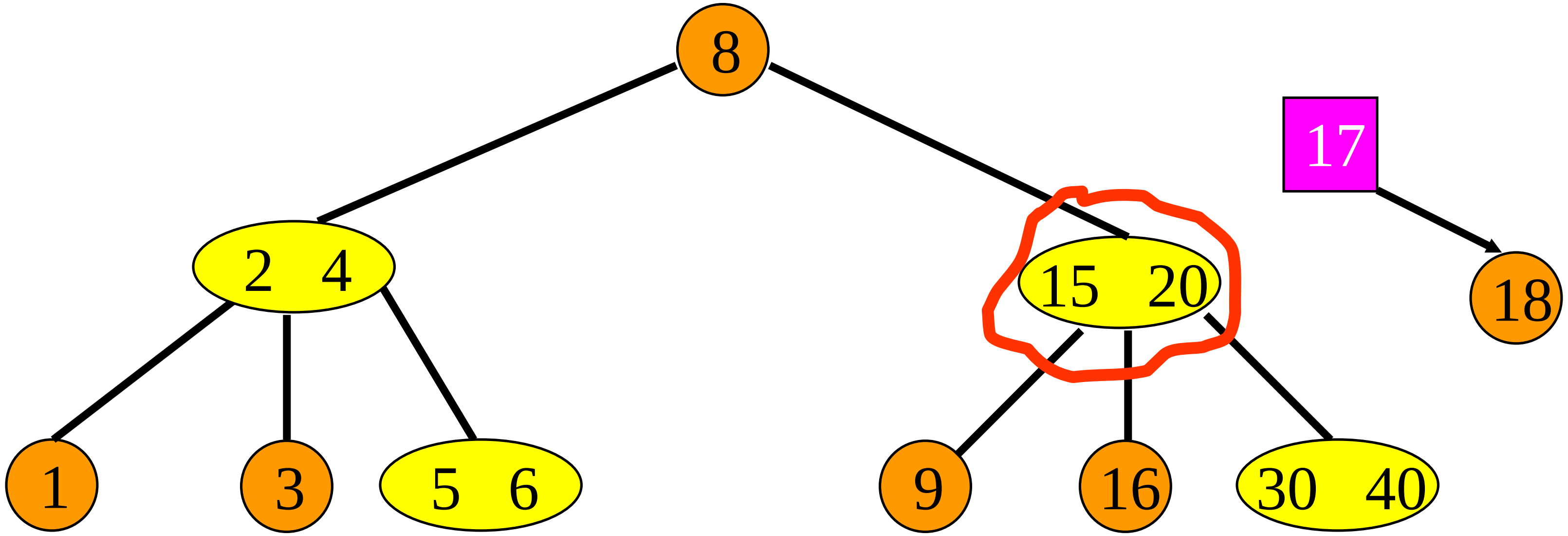
- Insert middle key and pointer to new node into parent.

Insert



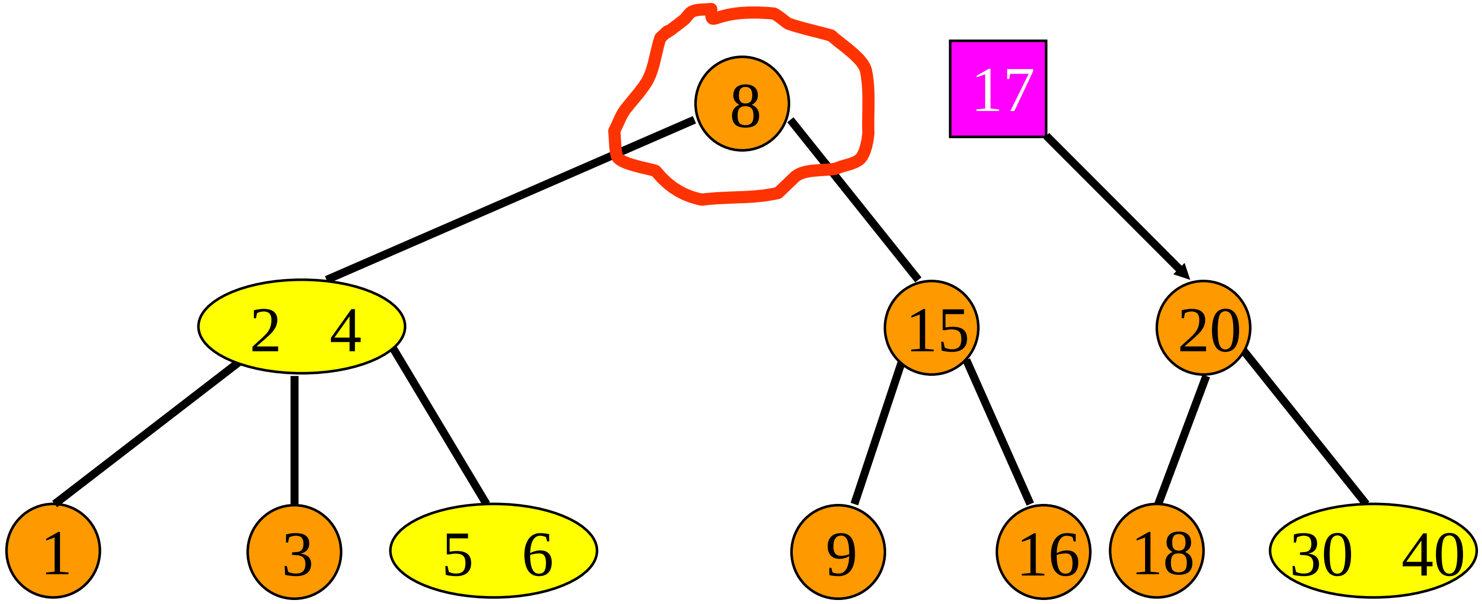
- Insert a pair with key = 18.

Insert



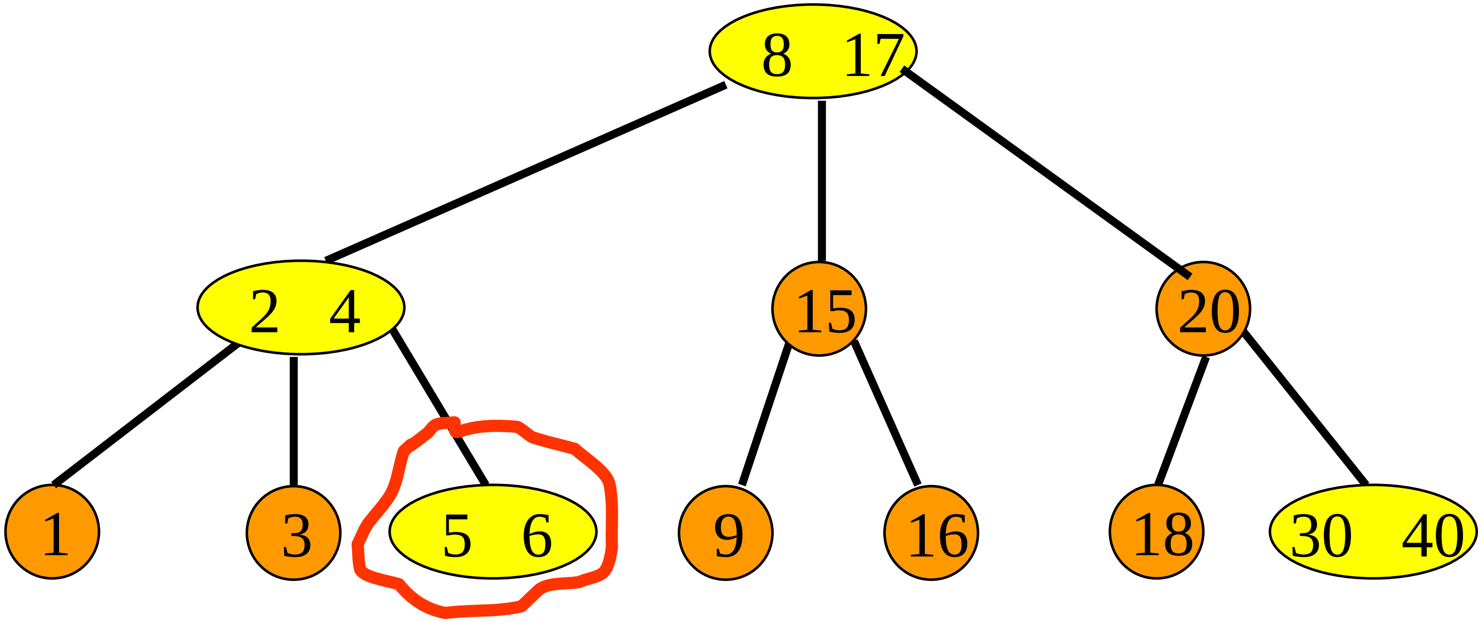
- Insert a pair with key = 17 plus a pointer into parent.

Insert



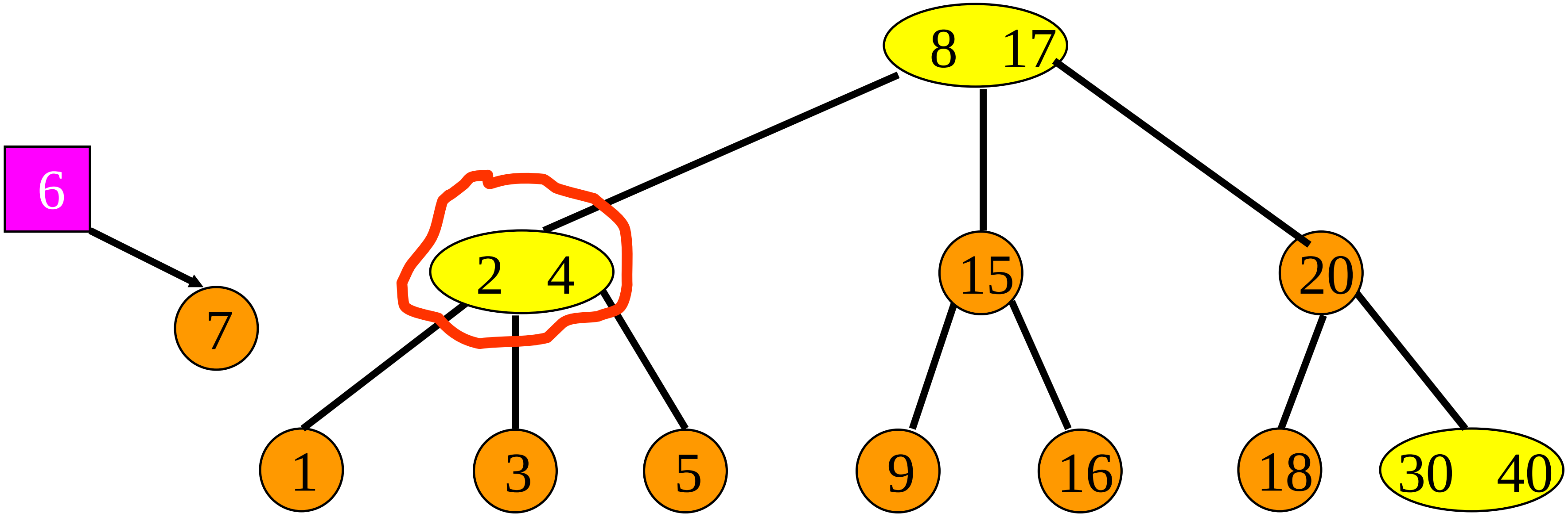
- Insert a pair with key = 17 plus a pointer into parent.

Insert



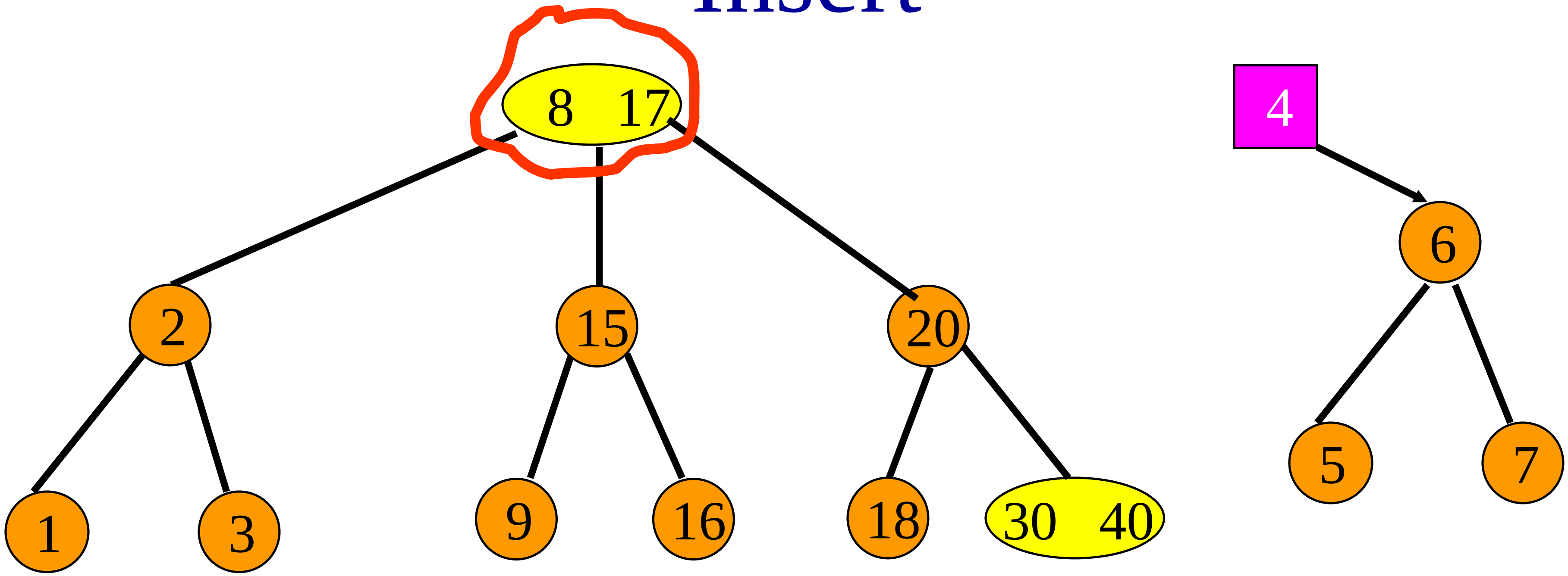
- Now, insert a pair with key = 7.

Insert



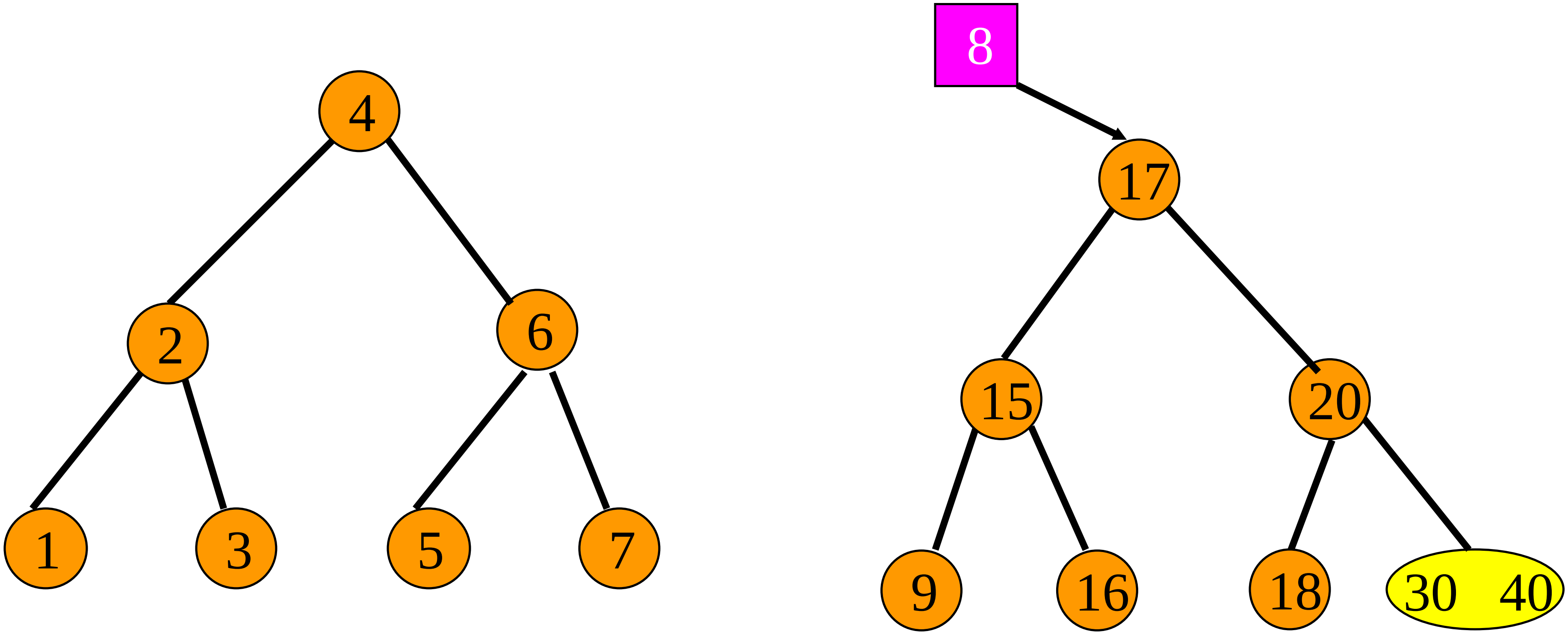
- Insert a pair with key = 6 plus a pointer into parent.

Insert



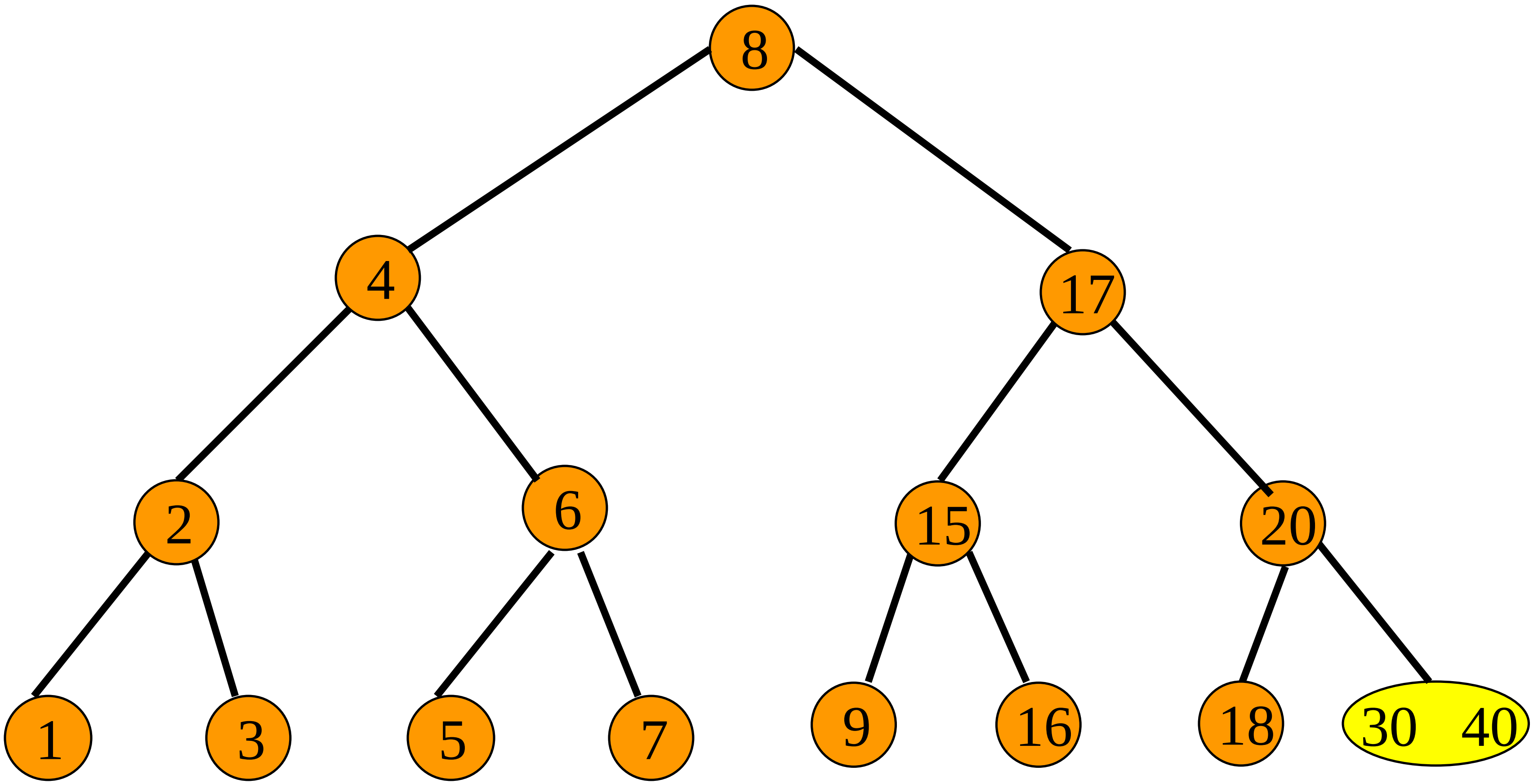
• Insert a pair with key = 4 plus a pointer into parent.

Insert



- Insert a pair with key = 8 plus a pointer into parent.
- There is no parent. So, create a new root.

Insert

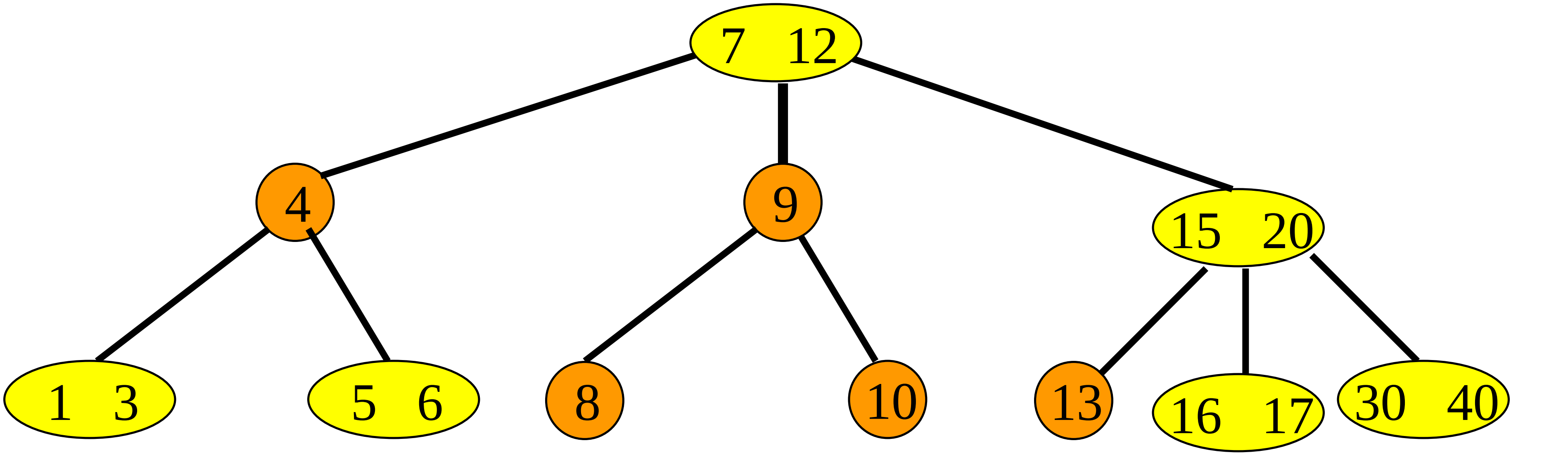


- Height increases by 1.

B-Trees (continued)

- Analysis of worst-case and average number of disk accesses for an insert.
- Delete and analysis.
- Structure for B-tree node

Worst-Case Disk Accesses



Insert 14.

Insert 2.

Insert 18.

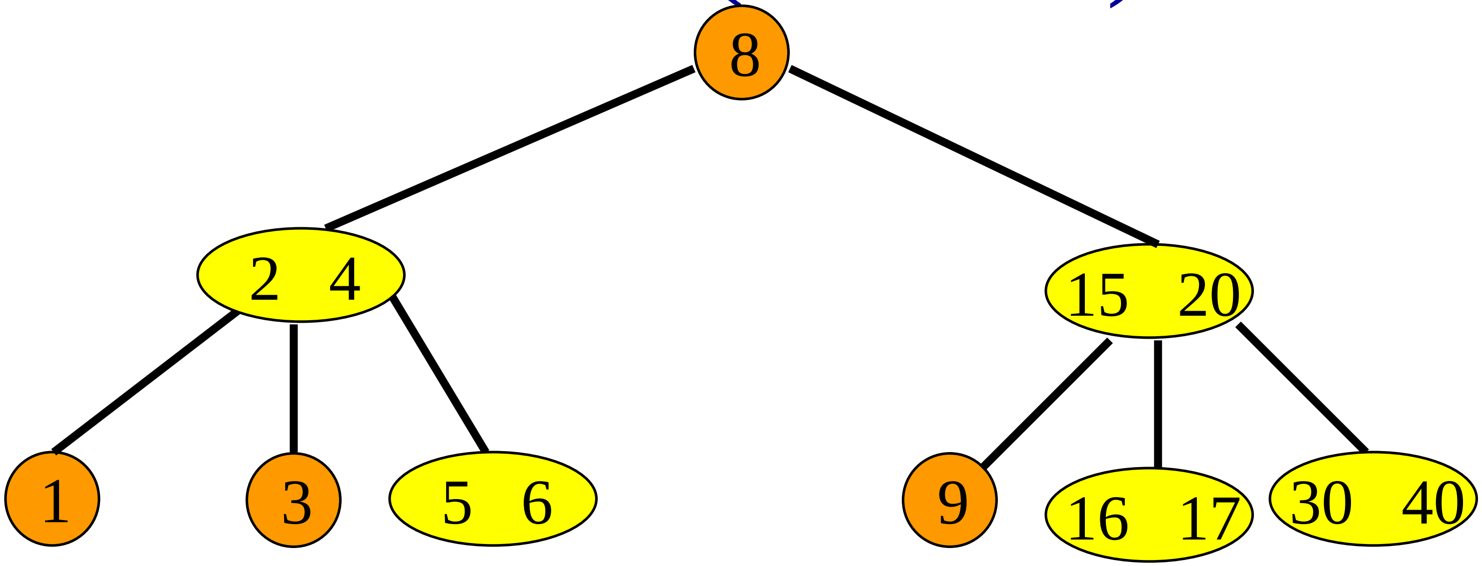
Worst-Case Disk Accesses

- Assume enough memory to hold all h nodes accessed on way down.
- h read accesses on way down.
- 2s+1 write accesses on way up, s = number of nodes that split.
- Total h+2s+1 disk accesses.
- Max is 3h+1.

Average Disk Accesses,

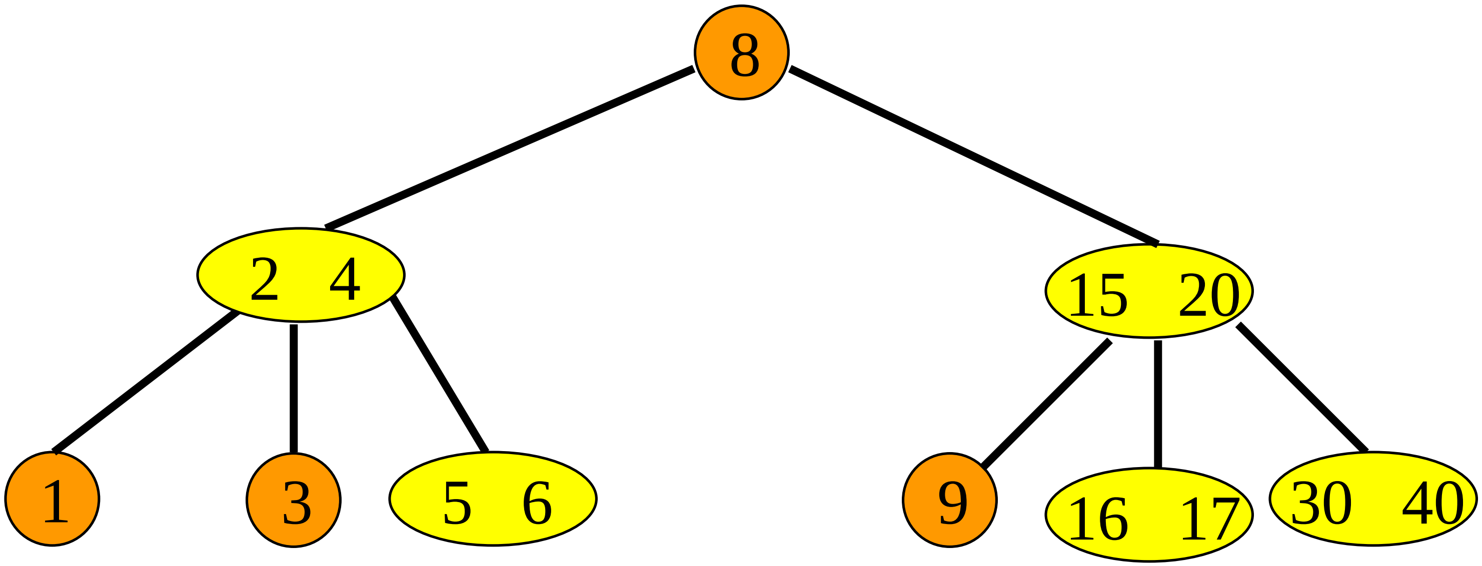
- Start with empty B-tree.
- Insert n pairs.
- Resulting B-tree has p nodes.
- # splits $<= p - 2$, $p > 2$.
- # pairs $>= 1+(\mathrm{ceil}(m/2) - 1)(p - 1)$.
- $s_{avg} <= (p - 2)/(1+(\mathrm{ceil}(m/2) - 1)(p - 1))$.
- So, $s_{avg} < 1/(\mathrm{ceil}(m/2) - 1)$.
- $m = 200 \Rightarrow s_{avg} < 1/99$.
- Average disk accesses $< h + 2/99 + 1 \sim h + 1$.
- Nearly minimum.

Delete (2-3 tree)



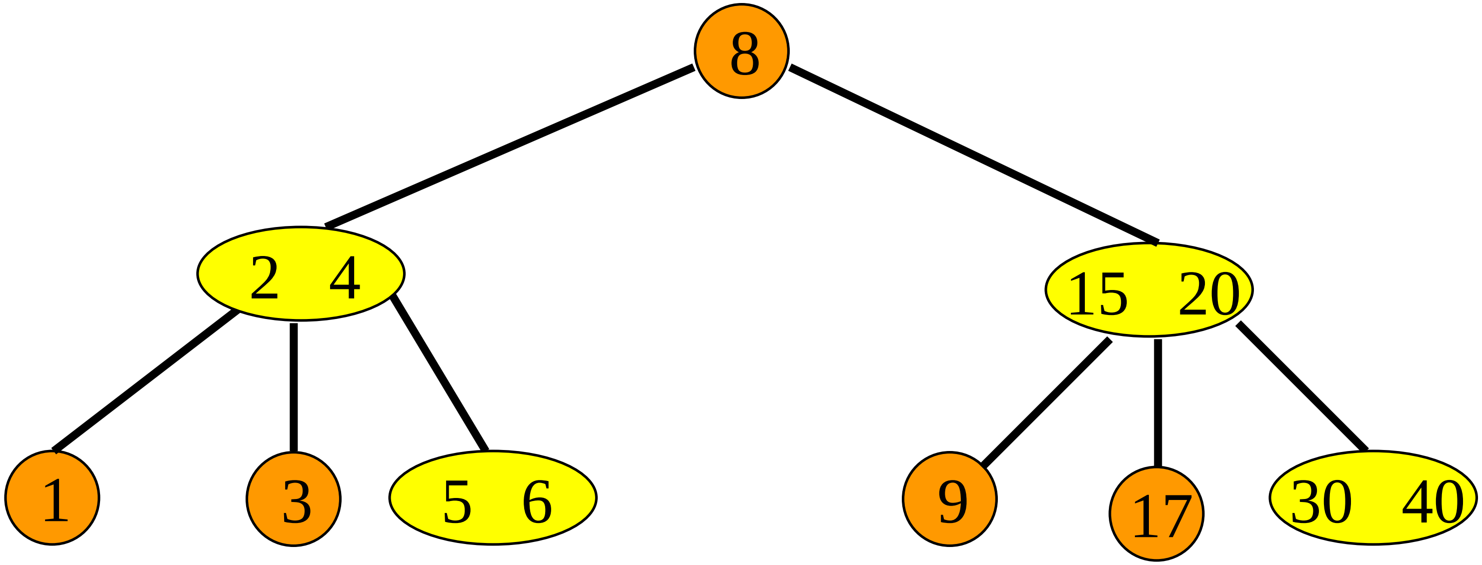
- Delete the pair with key = 8.
- Transform deletion from interior into deletion from a leaf.
- Replace by largest in left subtree.

Delete From A Leaf



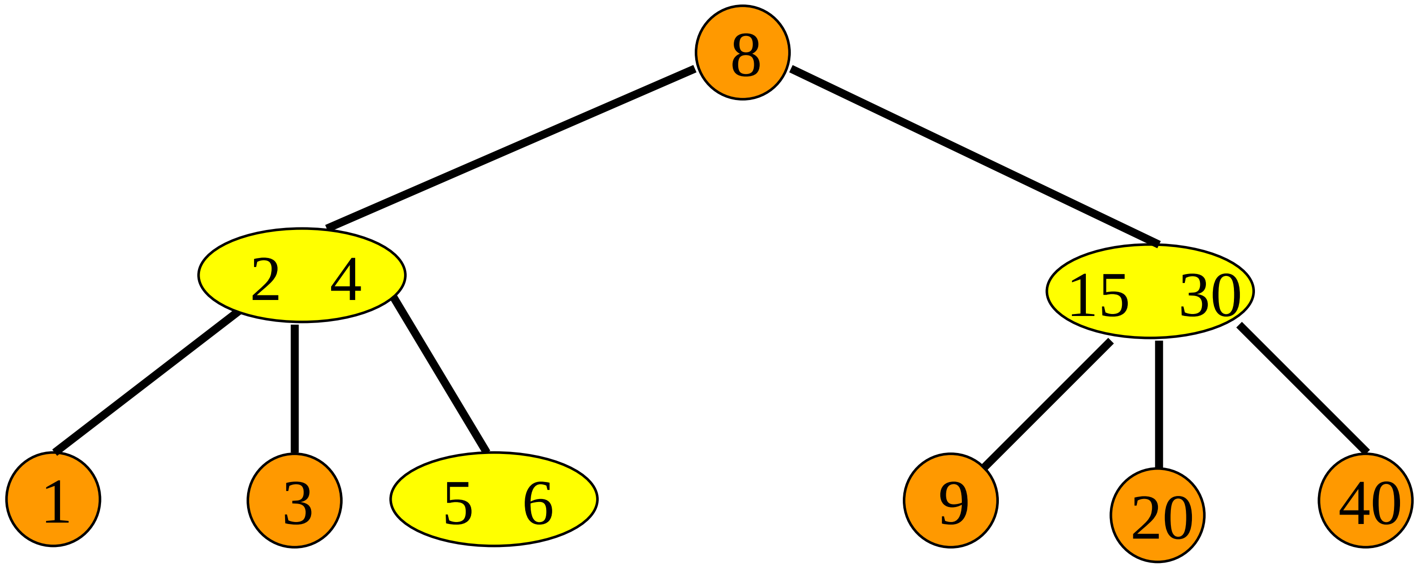
- Delete the pair with key = 16.
- 3-node becomes 2-node.

Delete From A Leaf



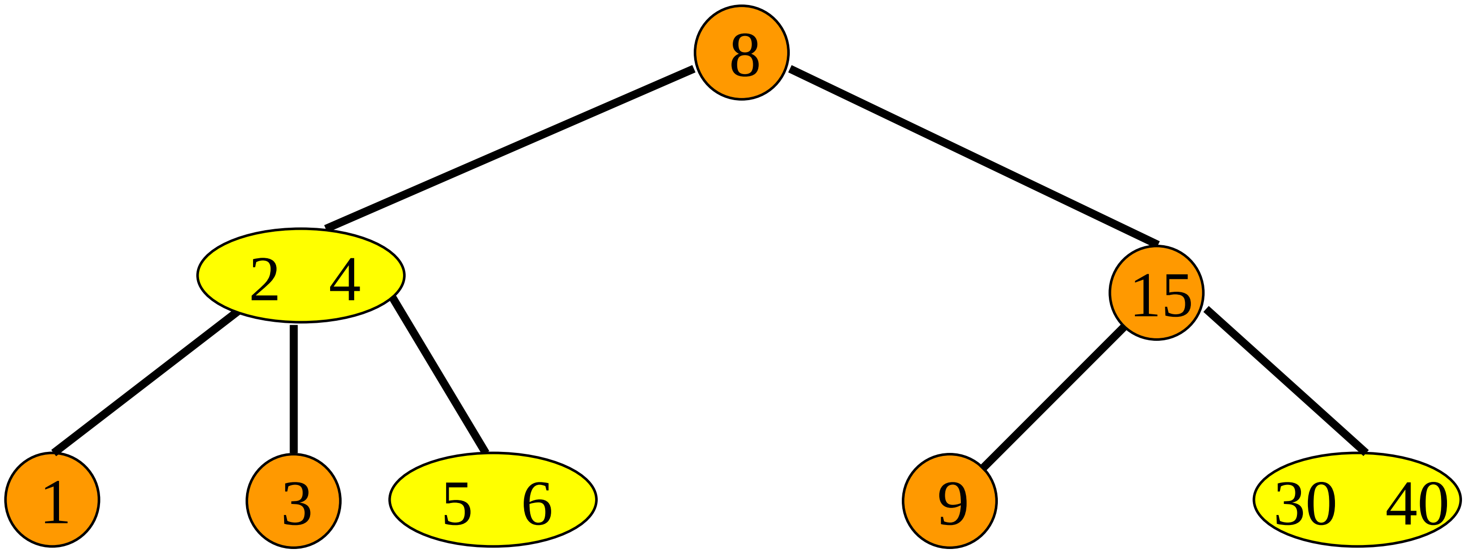
- Delete the pair with key = 17.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

Delete From A Leaf



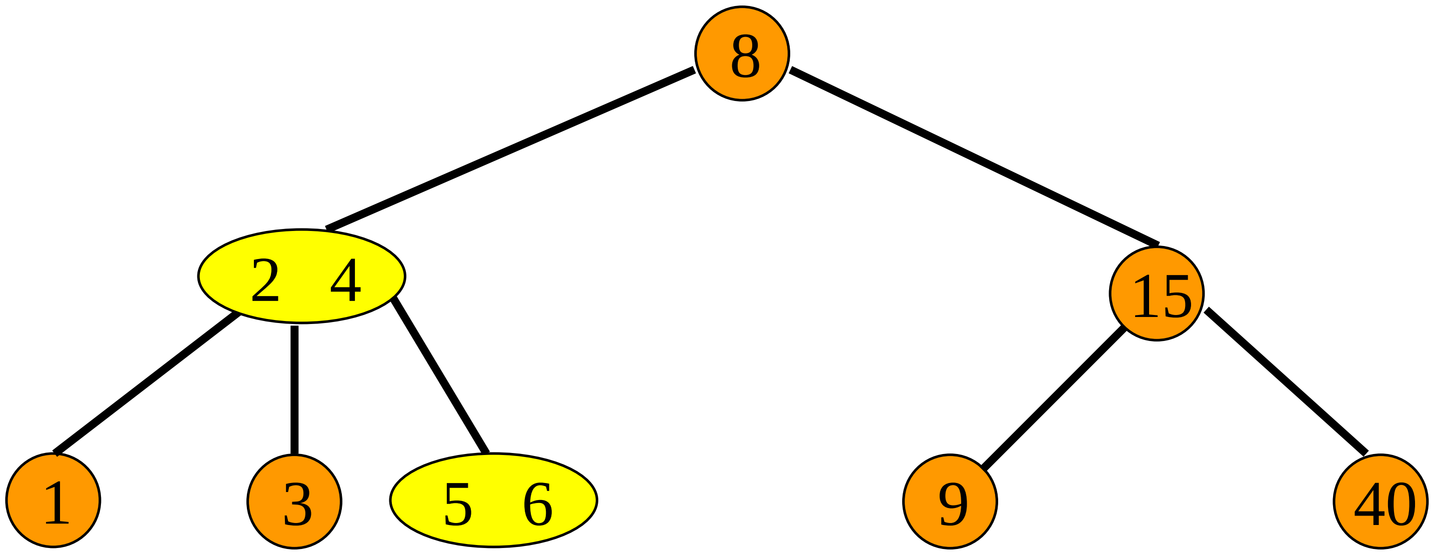
- Delete the pair with key = 20.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



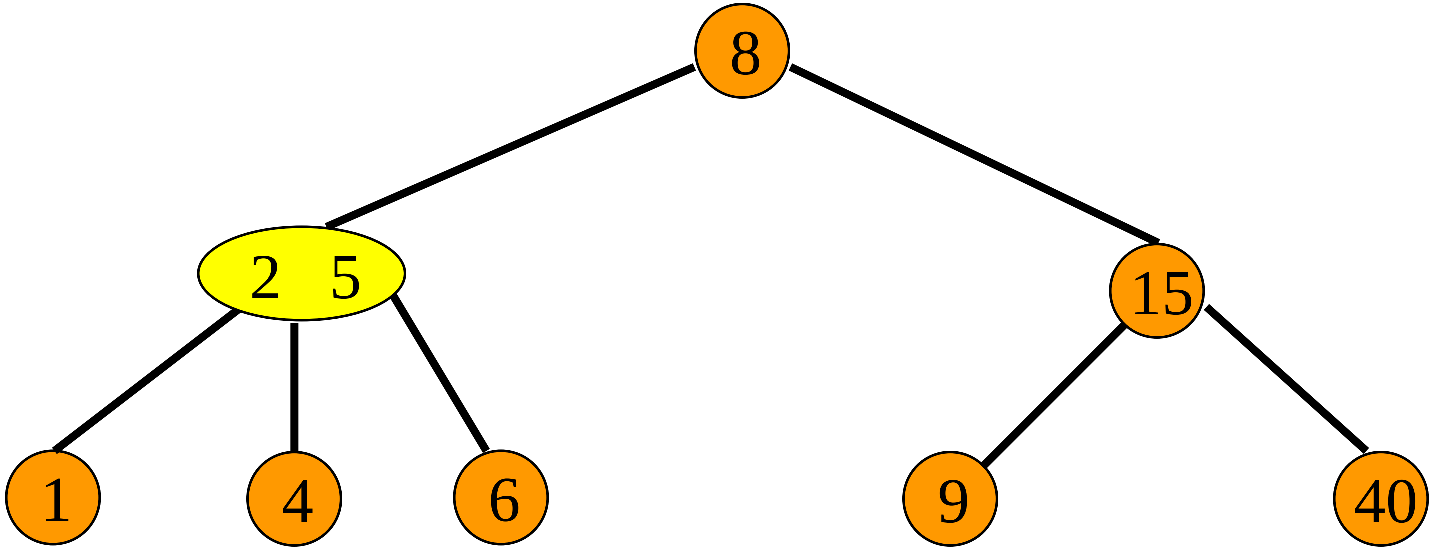
- Delete the pair with key = 30.
- Deletion from a 3-node.
- 3-node becomes 2-node.

Delete From A Leaf



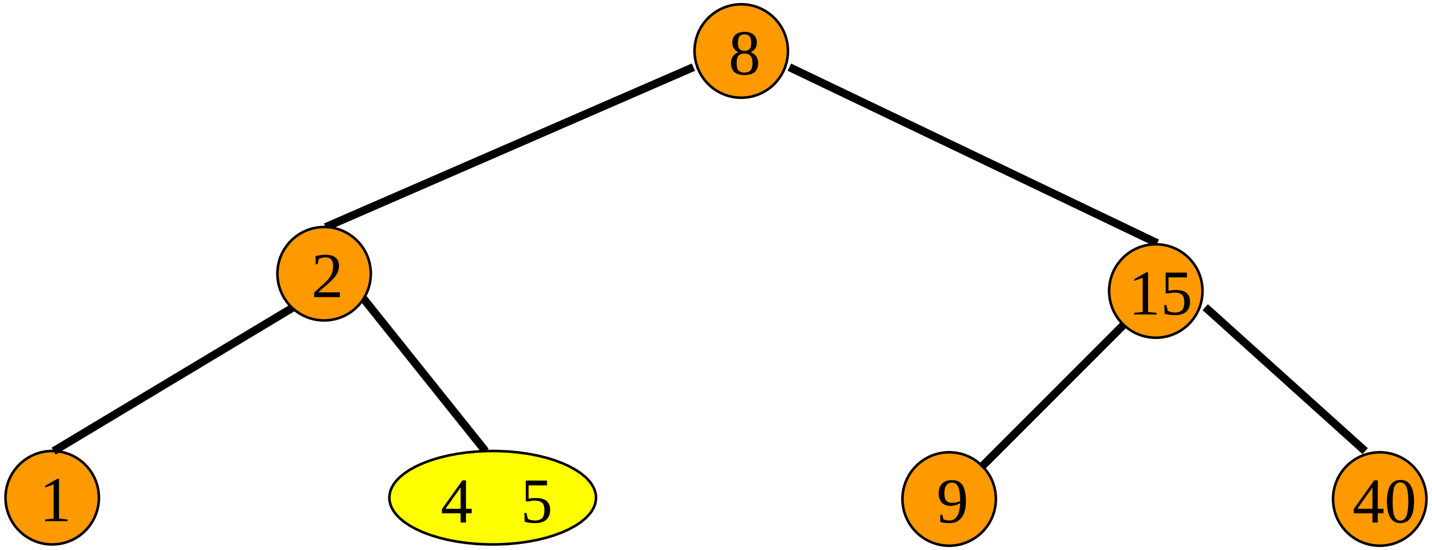
- Delete the pair with key = 3.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If so borrow a pair and a subtree via parent node.

Delete From A Leaf



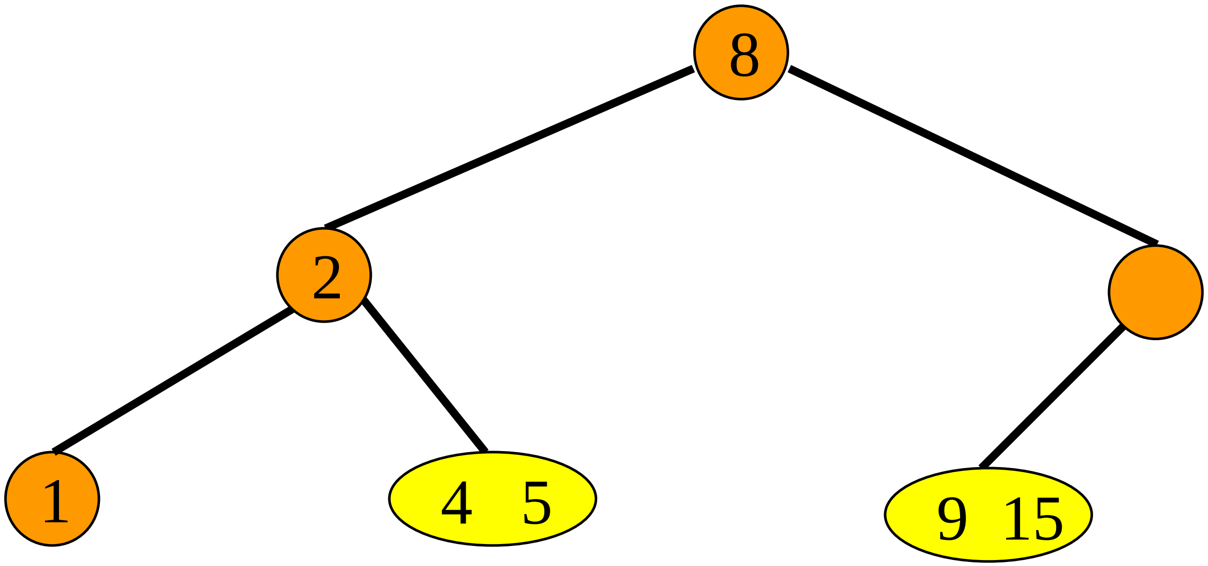
- Delete the pair with key = 6.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



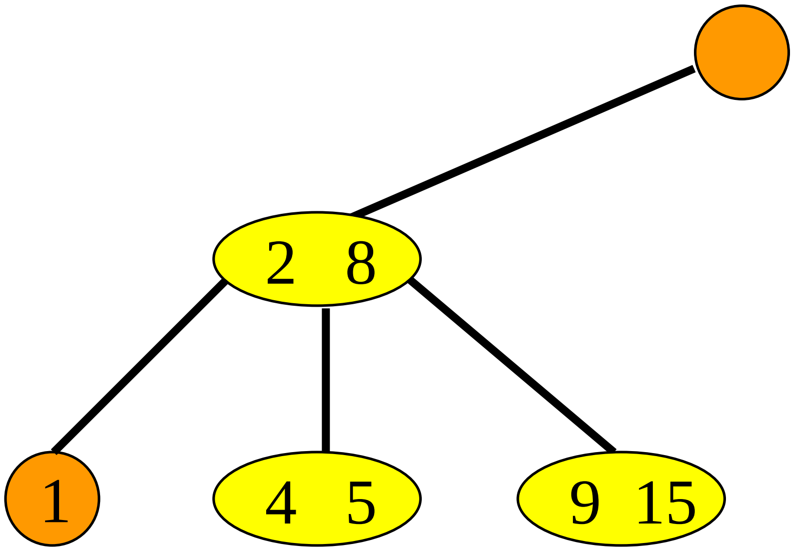
- Delete the pair with key = 40.
- Deletion from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



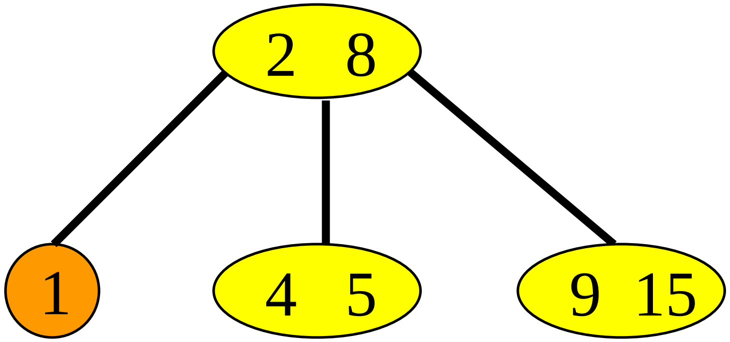
- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- If not, combine with sibling and parent pair.

Delete From A Leaf



- Parent pair was from a 2-node.
- Check one sibling and determine if it is a 3-node.
- No sibling, so must be the root.
- Discard root. Left child becomes new root.

Delete From A Leaf

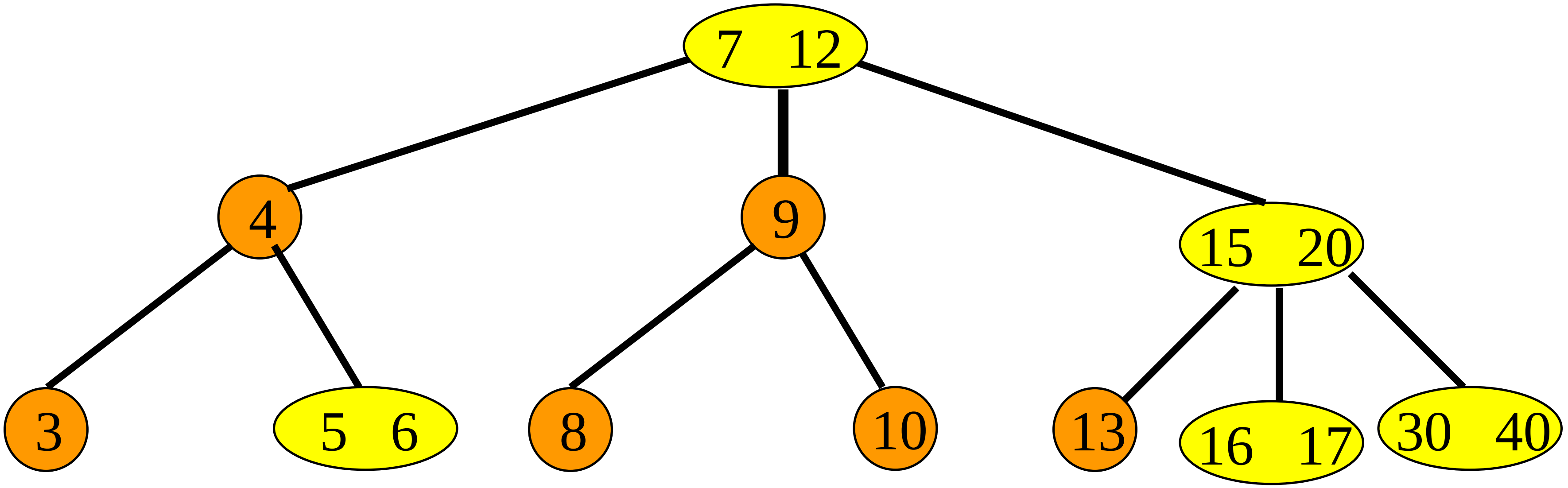


- Height reduces by 1.

(** Delete A Pair, not included **)

- Deletion from interior node is transformed into a deletion from a leaf node.
- Deficient leaf triggers bottom-up borrowing and node combining pass.
- Deficient node is combined with an adjacent sibling who has exactly ceil(m/2) – 1 pairs.
- After combining, the node has [ceil(m/2) – 2] (original pairs) + [ceil(m/2) – 1] (sibling pairs) + 1 (from parent) <= m –1 pairs.

Disk Accesses



Minimum.

Borrow.

Combine.

Worst-Case Disk Accesses

- Assume enough memory to hold all h nodes accessed on way down.
- h read accesses on way down.
- h – 1 sibling read accesses on way up.
- h – 2 writes of combined nodes on way up.
- 3 writes of root and level 2 nodes for sibling borrowing at level 2.
- Total is 3h.

Average Disk Accesses

- Start with B-tree that has n pairs and p nodes.
- Delete the pairs one by one.
- $n >= 1+(\text{ceil}(m/2) – 1)(p – 1)$.
- $p <= 1 + (n – 1)/(\text{ceil}(m/2) – 1)$.
- Upper bound on total number of disk accesses.
  - Each delete does a borrow.
  - The deletes together do at most $p – 1$ combines/merges.
- # accesses $<= n(h+4) + 2(p – 1)$.

Average Disk Accesses

- Average # accesses $<= [n(h+4) + 2(p - 1)]/ n \sim h + 4$.
- Nearly minimum.

Worst Case

- Alternating sequence of inserts and deletes.
- Each insert does h splits at a cost of $3h + 1$ disk accesses.
- Each delete moves back up to root at a cost of $3h$ disk accesses.
- Average for this sequence is $3h + 1$ for an insert and $3h$ for a delete.

Internal Memory B-Trees

- Cache access time vs main memory access time.
- Reduce main memory accesses using a B-tree.

Node Structure

$q \; a_0 \; p_1 \; a_1 \; p_2 \; a_2 \; \ldots \; p_q \; a_q$

- Node operations during a search.
  - Search the node for a given key.

Node Operations For Insert

- Insert a dictionary pair and a pointer (p, a).

$m\ a_0\ p_1\ a_1\ p_2\ a_2 \ldots p_m\ a_m$

$\text{ceil}(m/2)-1\ a_0\ p_1\ a_1\ p_2\ a_2 \ldots p_{\text{ceil}(m/2)-1}\ a_{\text{ceil}(m/2)-1}$

$m-\text{ceil}(m/2)\ a_{\text{ceil}(m/2)}\ p_{\text{ceil}(m/2)+1}\ a_{\text{ceil}(m/2)+1} \ldots p_m\ a_m$

- Find middle pair.
- 3-way split around middle pair.

Node Operations For Delete

- Delete a dictionary pair.
- Borrow.
  - Delete, replace, insert.
- Combine.
  - 3-way join.

Node Structure

- Each B-tree node is an array partitioned into indexed red-black tree nodes that will keep one dictionary pair each.
- Indexed red-black tree is built using simulated pointers (integer pointers).

Complexity Of B-Tree Node Operations

- Search a B-tree node … O(log m).
- Find middle pair … O(log m).
- Insert a pair … O(log m).
- Delete a pair … O(log m).
- Split a B-tree node … O(log m).
- Join 2 B-tree nodes … O(m).
  - Need to copy indexed red-black tree that represents one B-tree node into the array space of the other B-tree node.

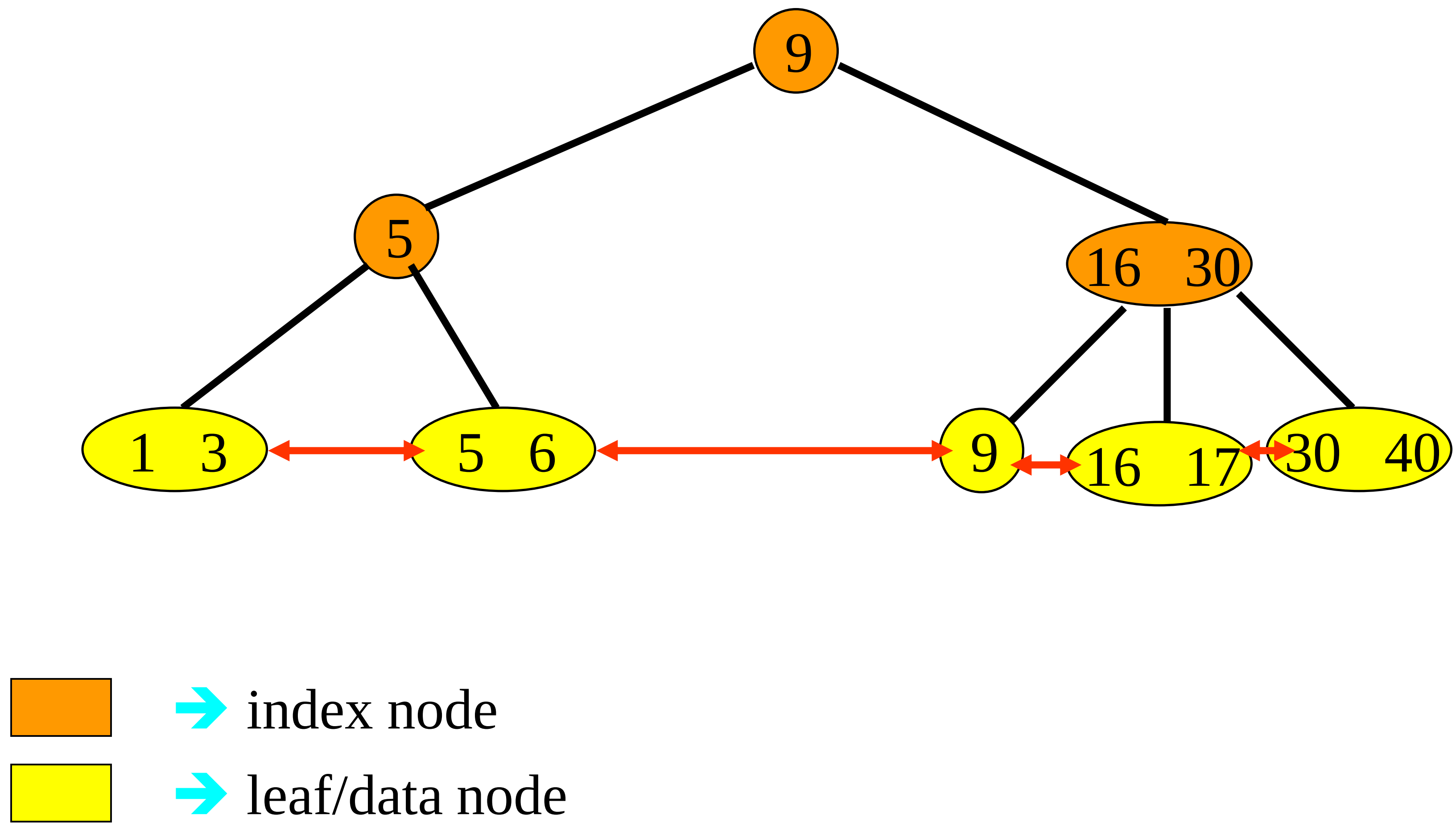
B^+-Trees

- Same structure as B-trees.
- Dictionary pairs are in leaves only. Leaves form a doubly-linked list.
- Remaining nodes have following structure:

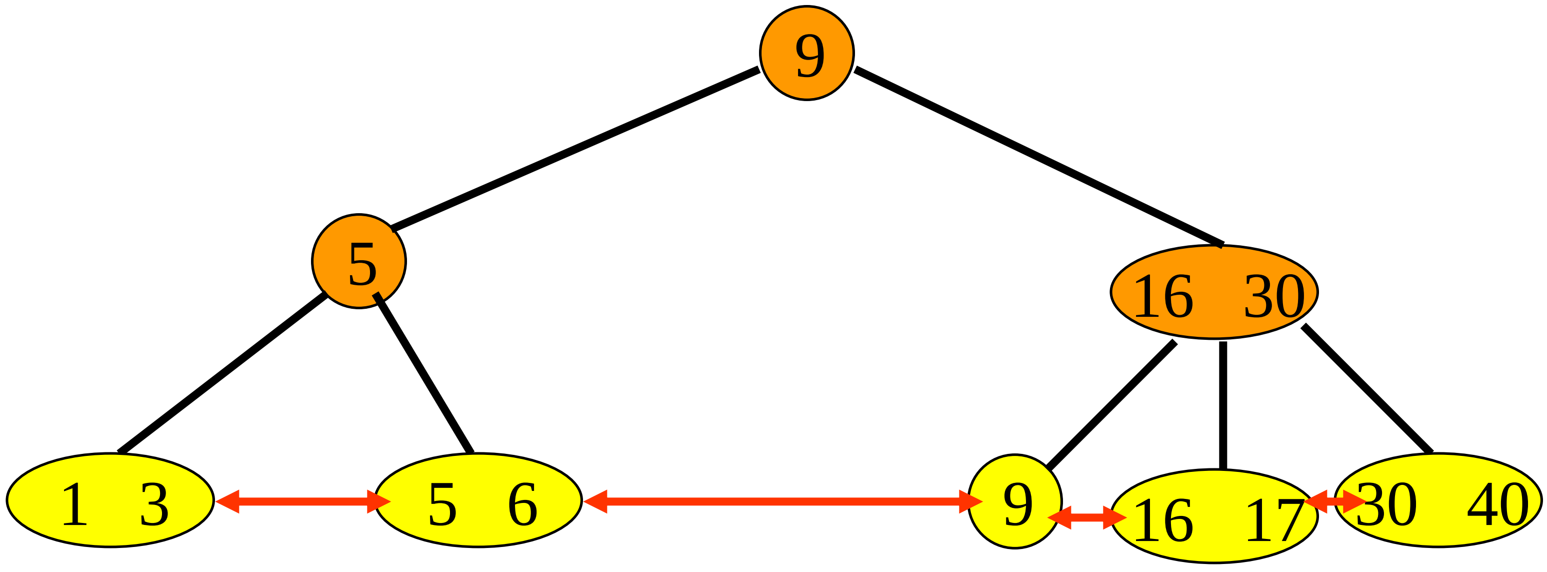
$$j \ a_0 \ k_1 \ a_1 \ k_2 \ a_2 \ \ldots \ k_j \ a_j$$

- j = number of keys in node.
- a_i is a pointer to a subtree.
- k_i <= smallest key in subtree a_i and > largest in a_{i-1}.

Example B+-tree



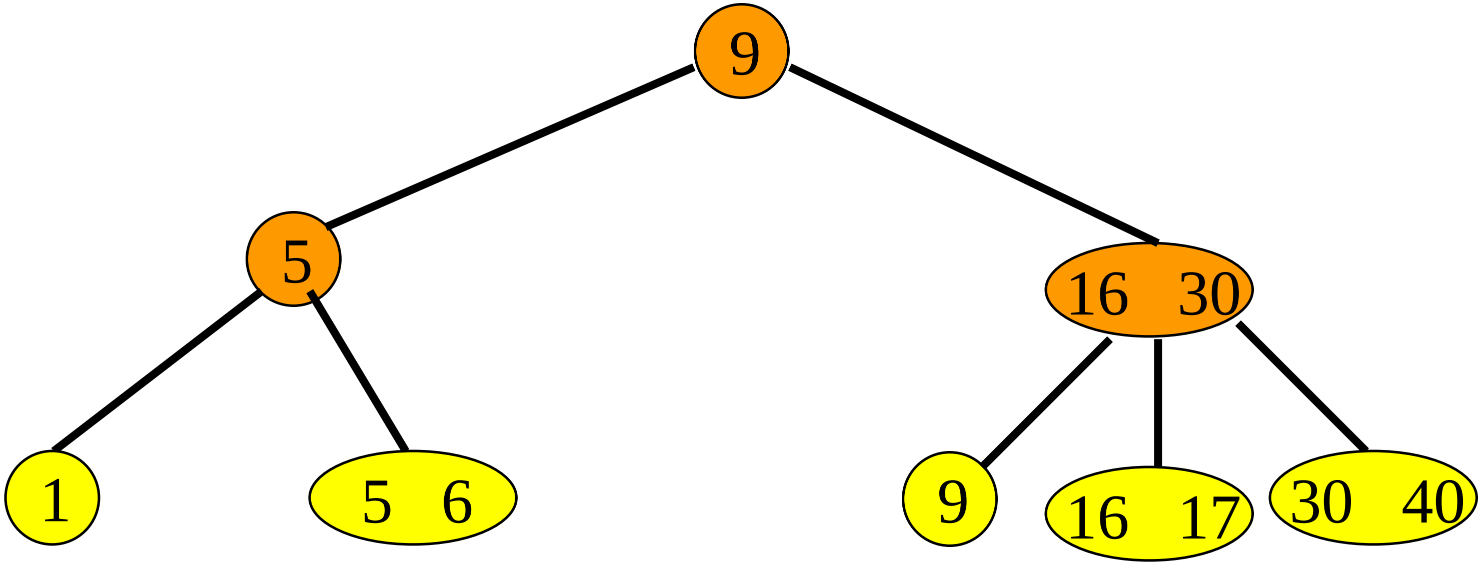
B+-tree—Search



key = 5

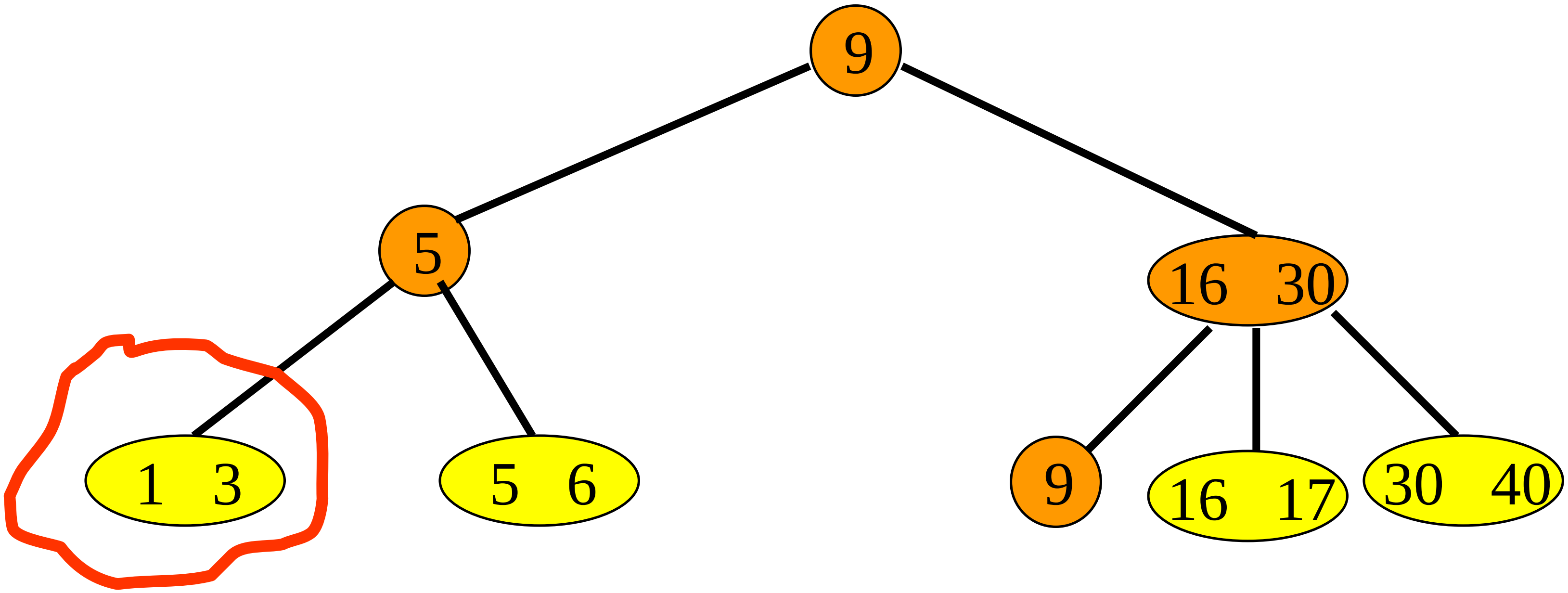
6 <= key <= 20

B+-tree—Insert



Insert 10

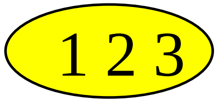
Insert



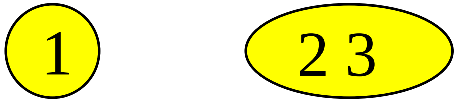
- Insert a pair with key = 2.
- New pair goes into a 3-node.

Insert Into A 3-node

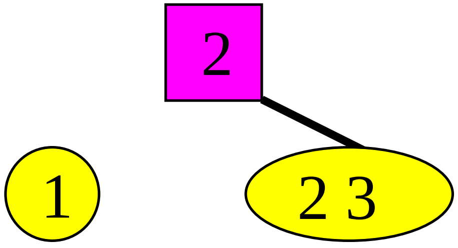
- Insert new pair so that the keys are in ascending order.



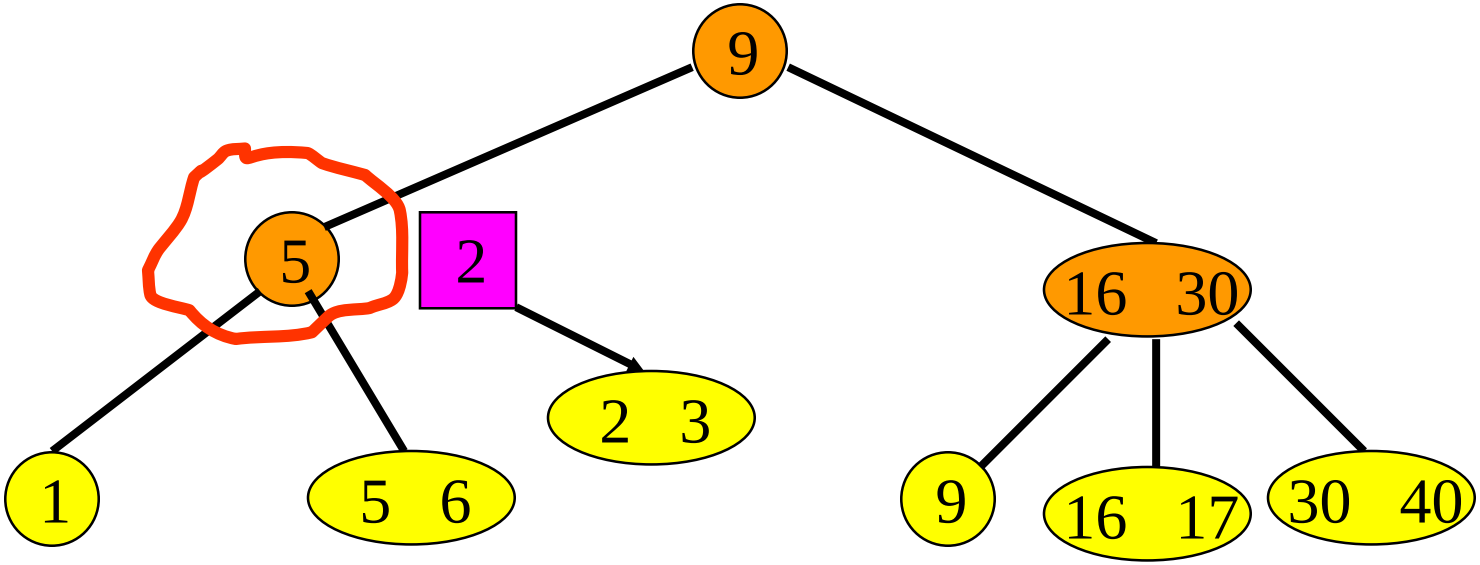
- Split into two nodes.



- Insert smallest key in new node and pointer to this new node into parent.

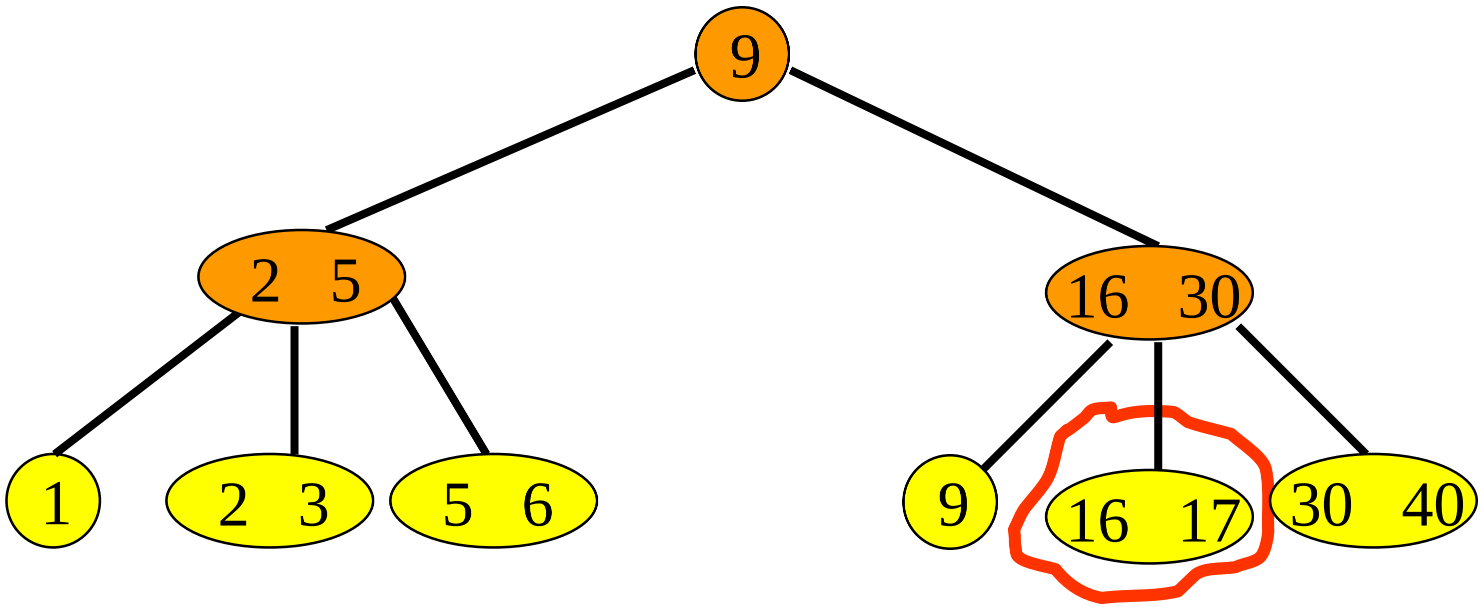


Insert



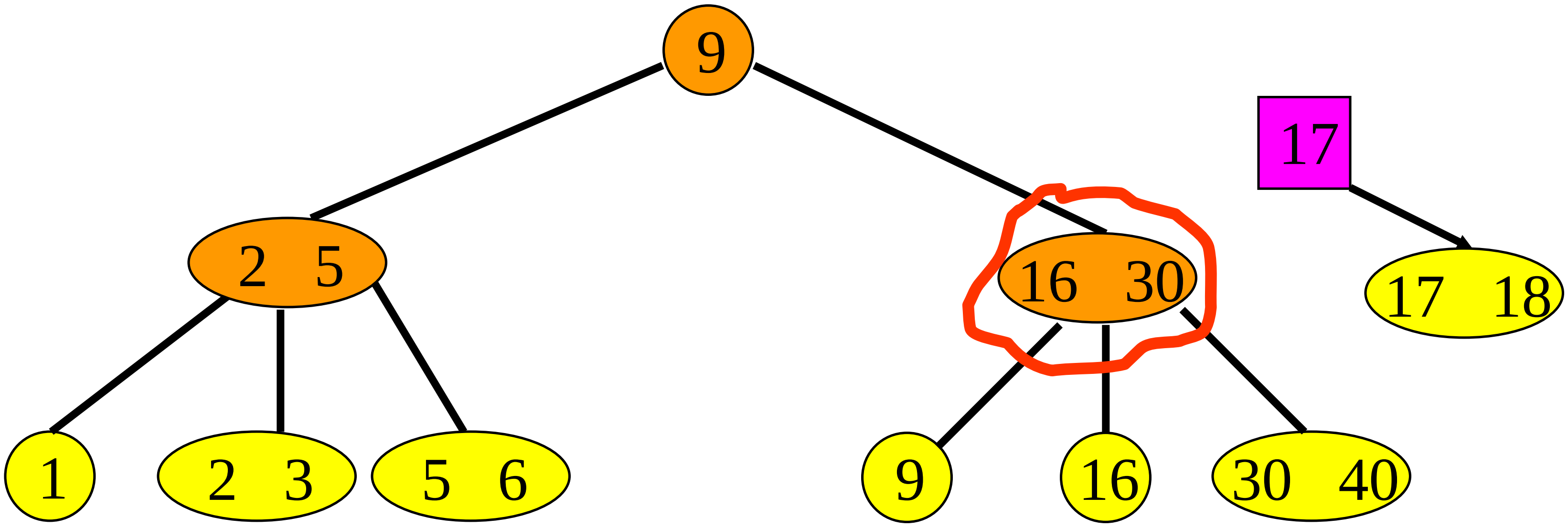
- Insert an index entry 2 plus a pointer into parent.

Insert



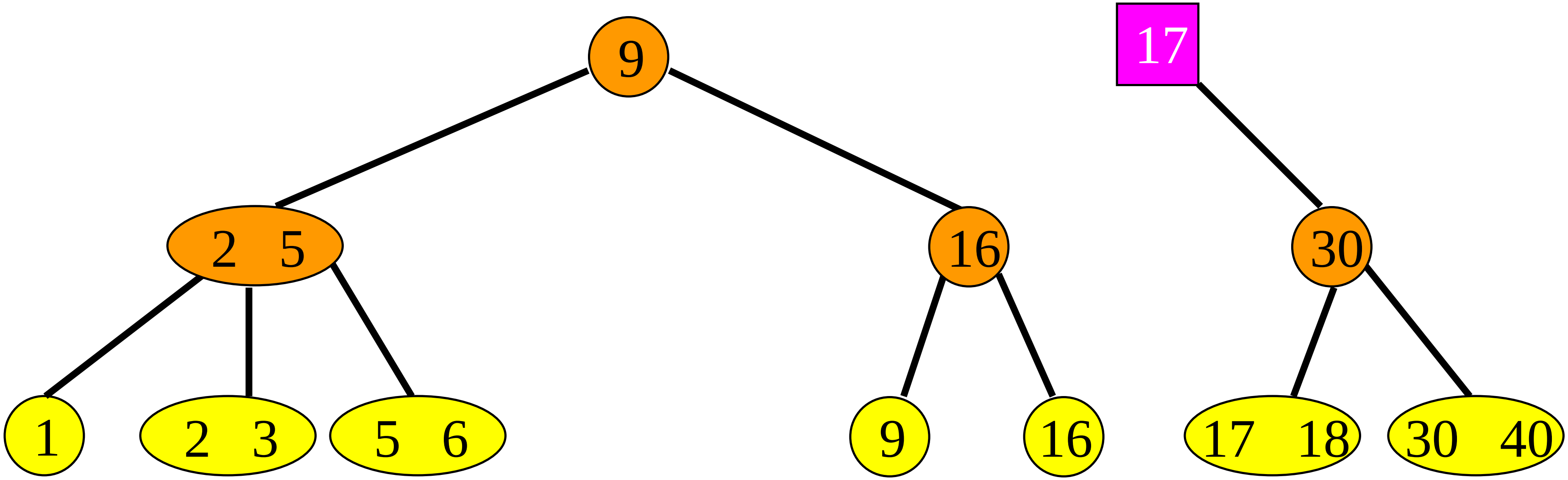
- Now, insert a pair with key = 18.

Insert



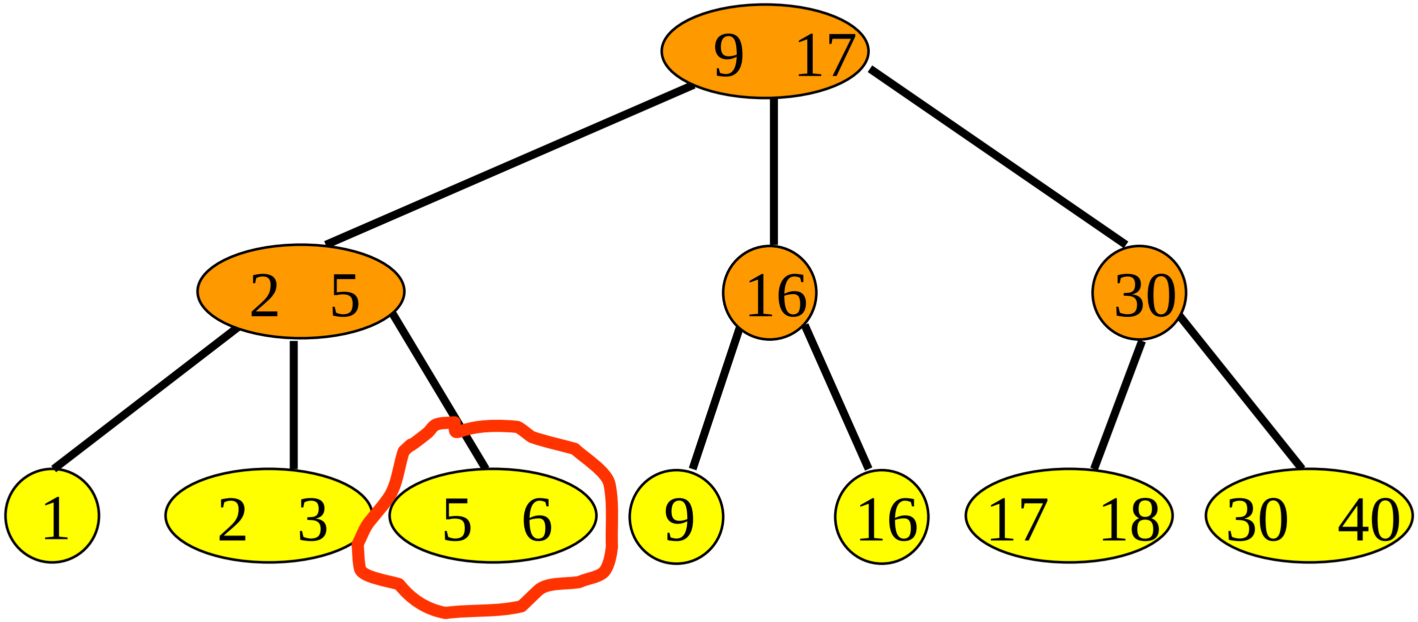
- Now, insert a pair with key = 18.
- Insert an index entry 17 plus a pointer into parent.

Insert



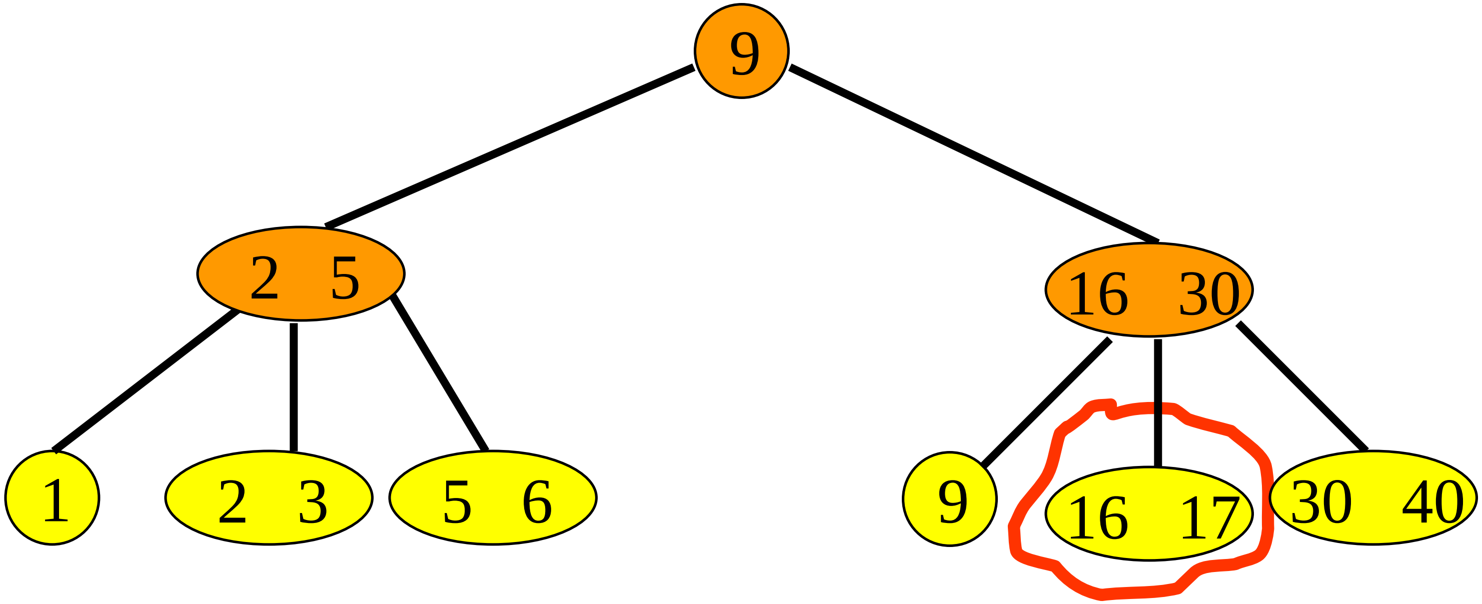
- Now, insert a pair with key = 18.
- Insert an index entry17 plus a pointer into parent.

Insert



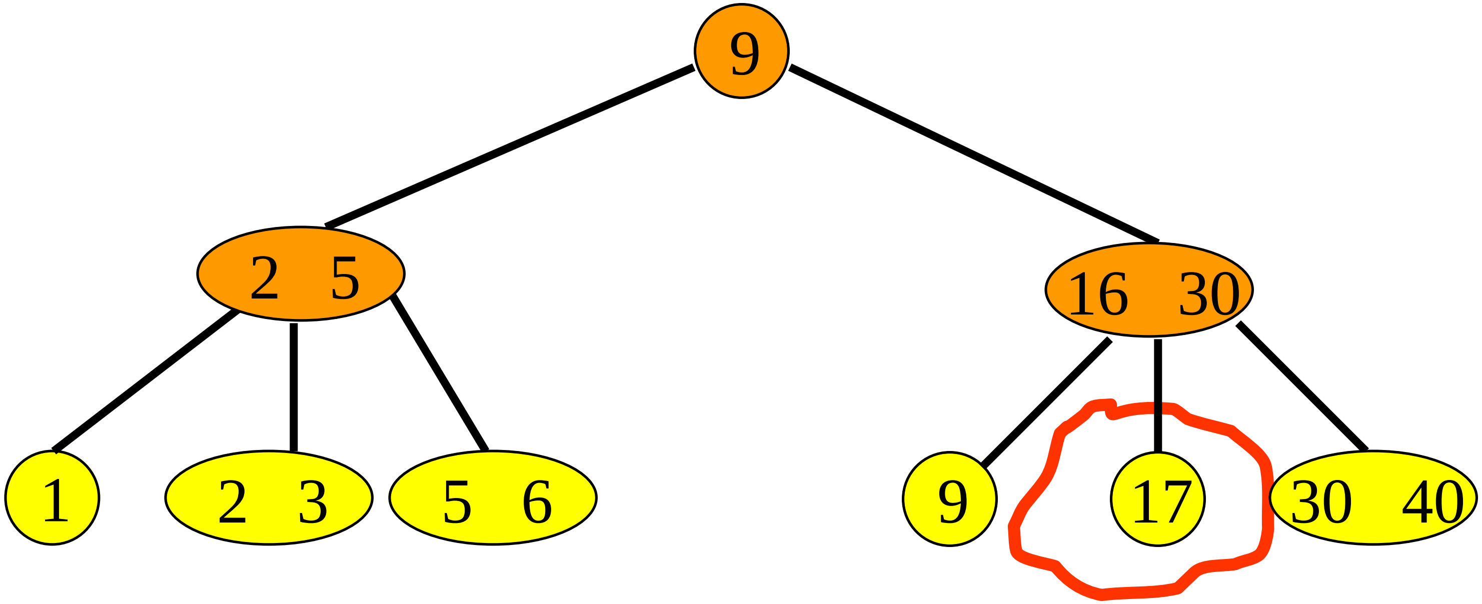
- Now, insert a pair with key = 7.

Delete



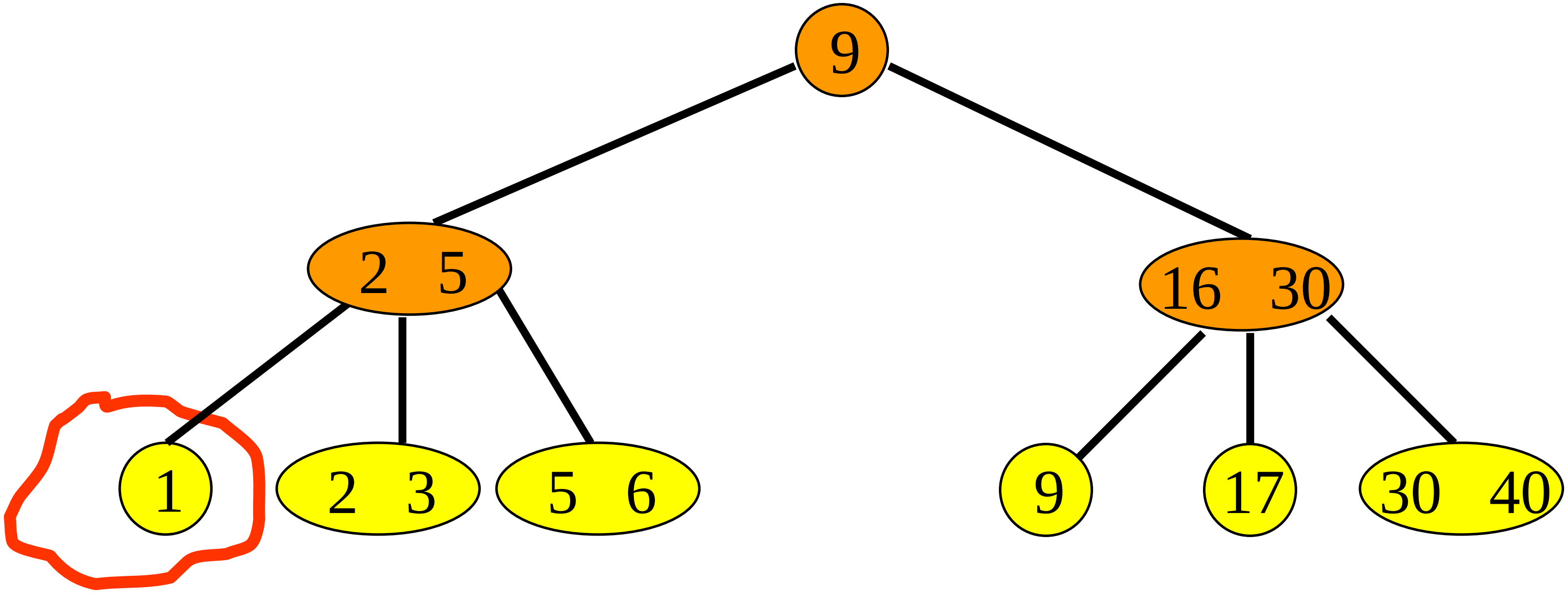
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

Delete



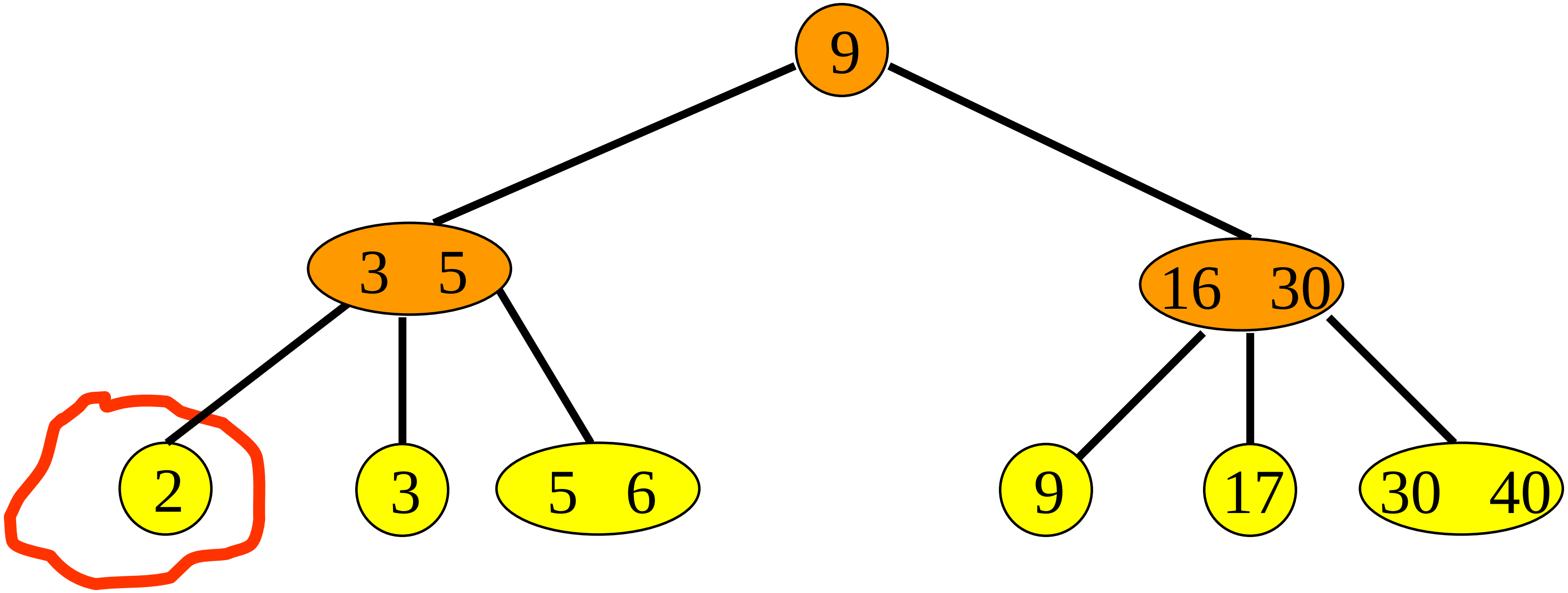
- Delete pair with key = 16.
- Note: delete pair is always in a leaf.

Delete



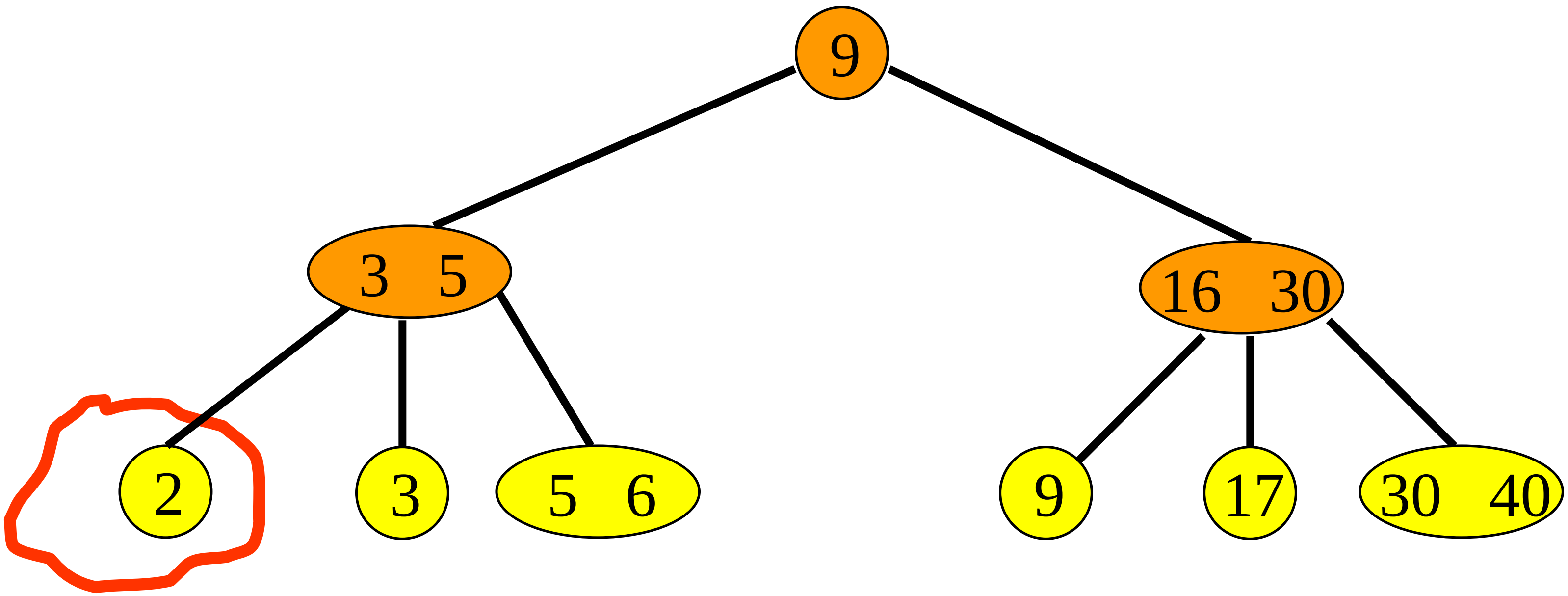
- Delete pair with key = 1.
- Get >= 1 from sibling and update parent key.

Delete



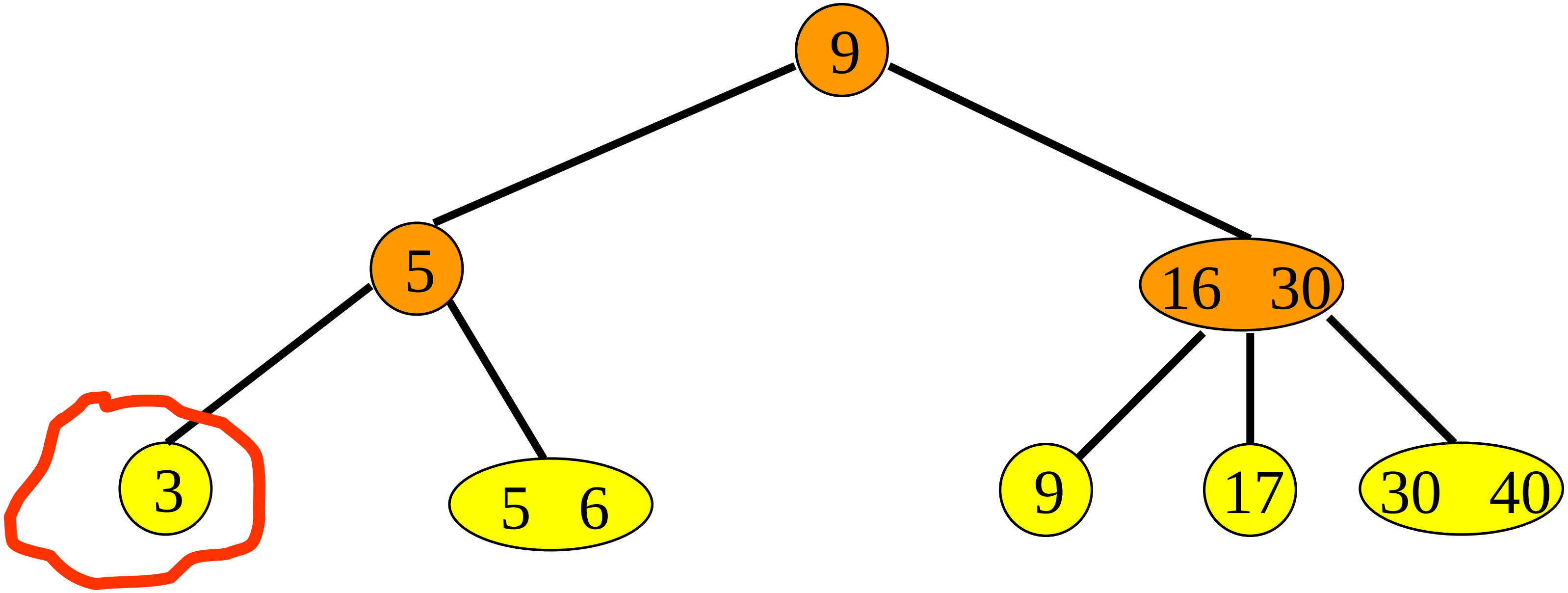
- Delete pair with key = 1.
- Get >= 1 from sibling and update parent key.

Delete



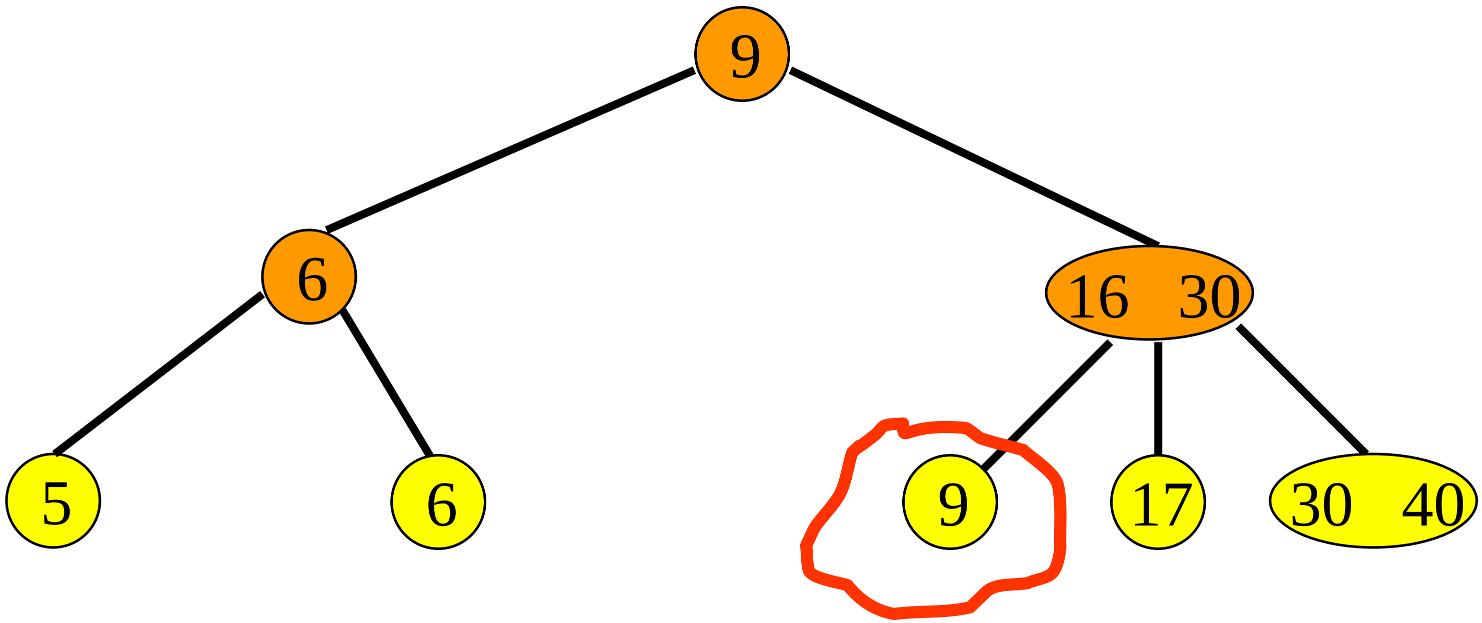
- Delete pair with key = 2.
- Merge with sibling, delete in-between key in parent.

Delete



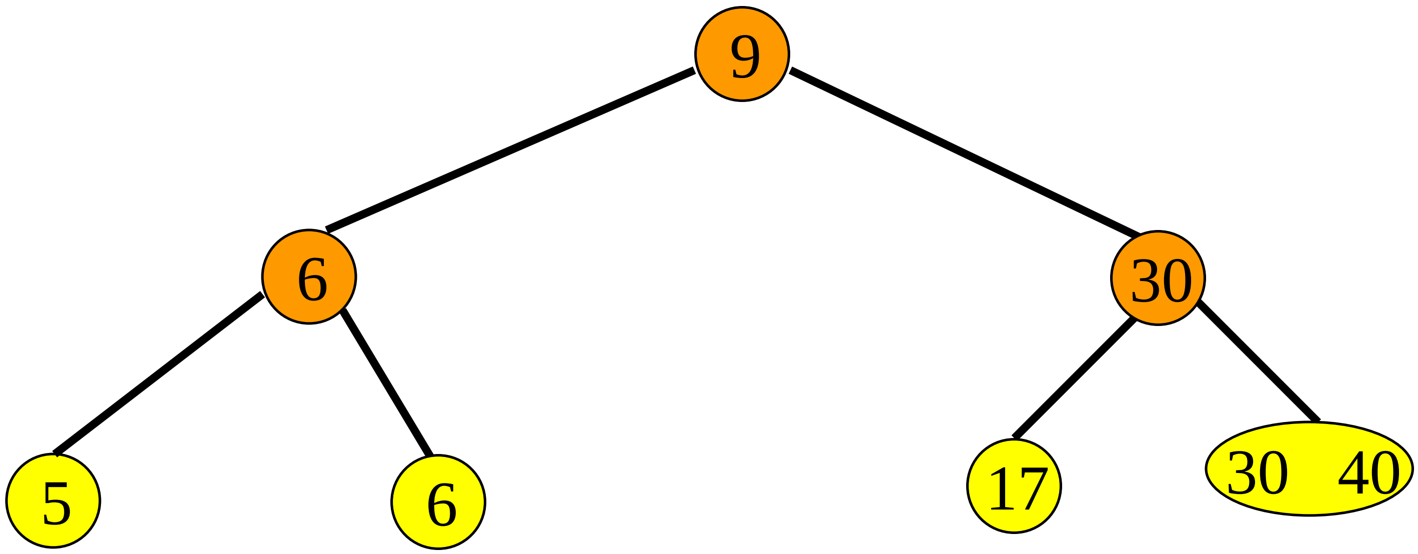
- Delete pair with key = 3.
- Get >= 1 from sibling and update parent key.

Delete

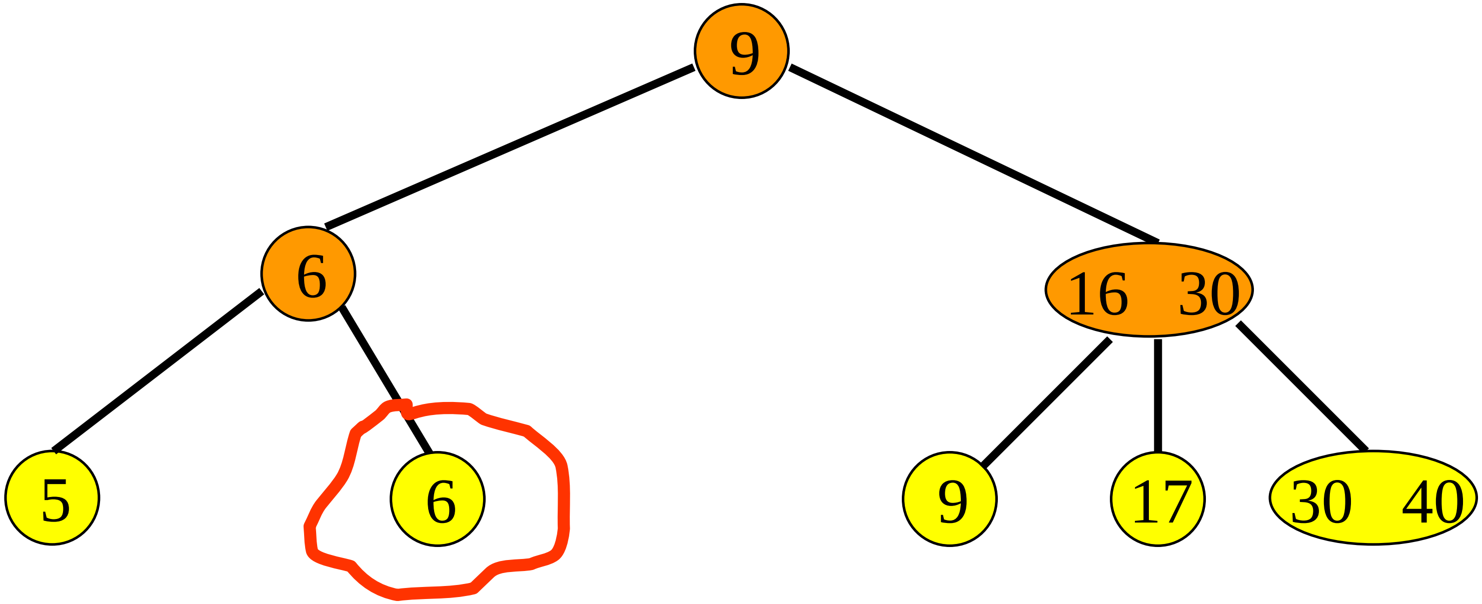


- Delete pair with key = 9.
- Merge with sibling, delete in-between key in parent.

Delete

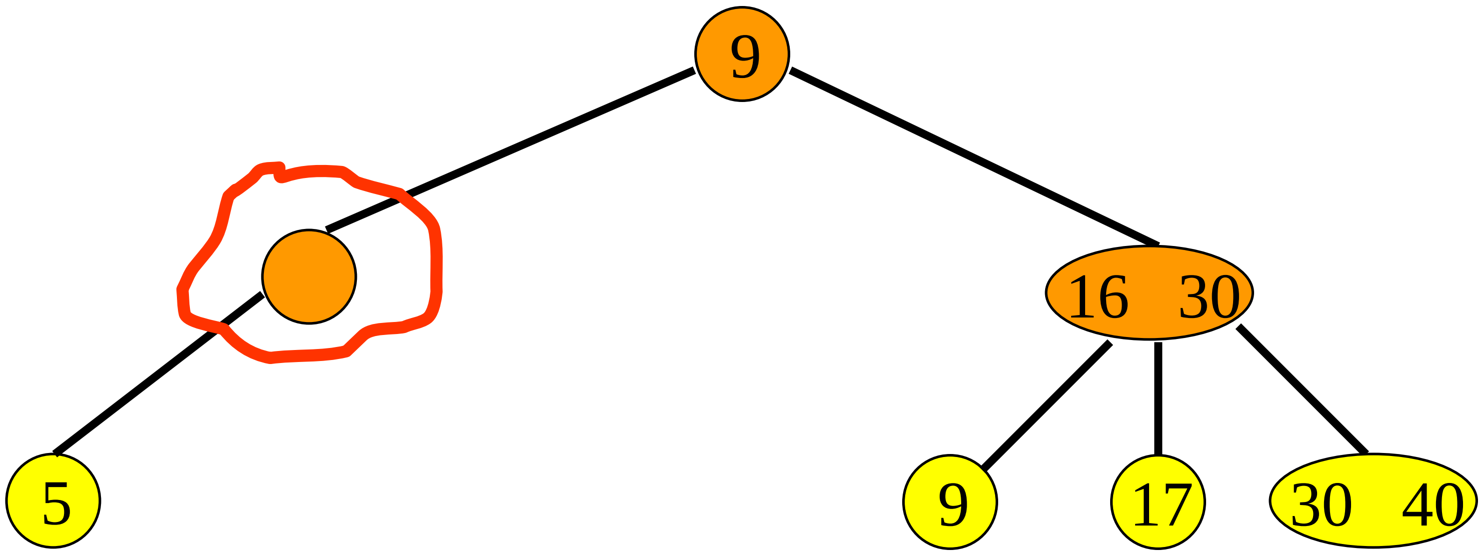


Delete



- Delete pair with key = 6.
- Merge with sibling, delete in-between key in parent.

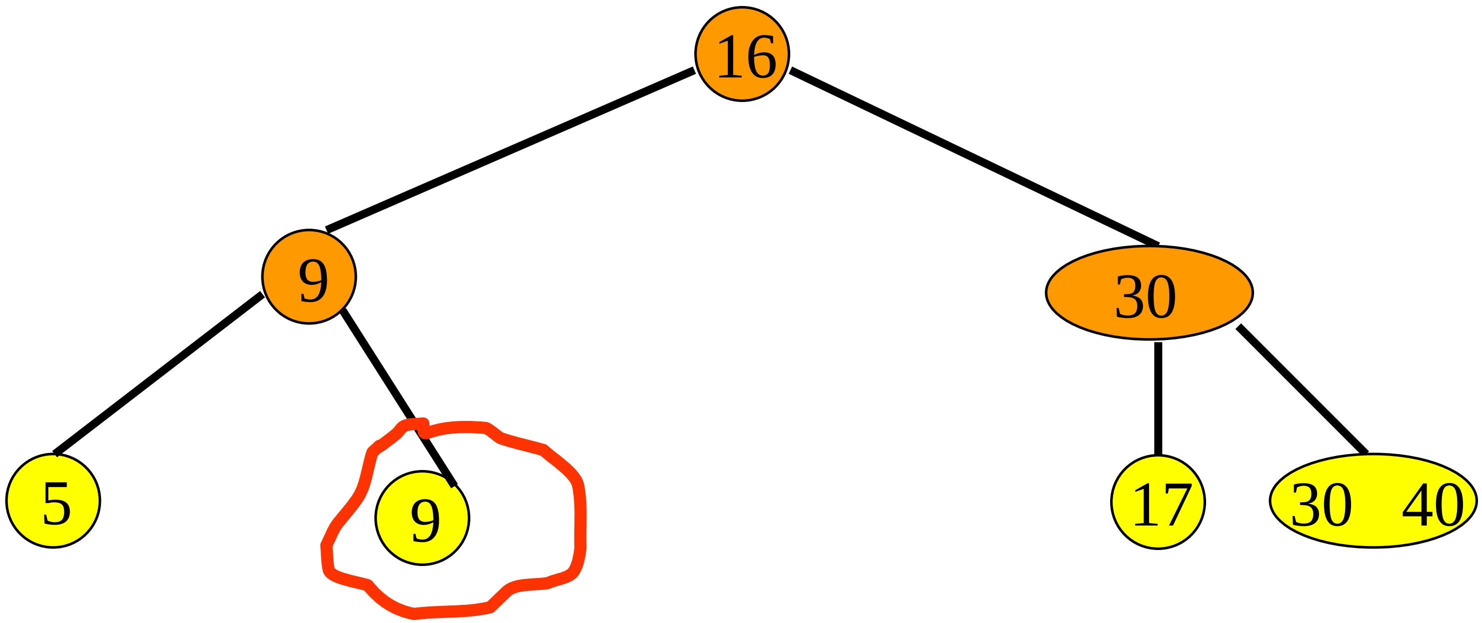
Delete



- Index node becomes deficient.

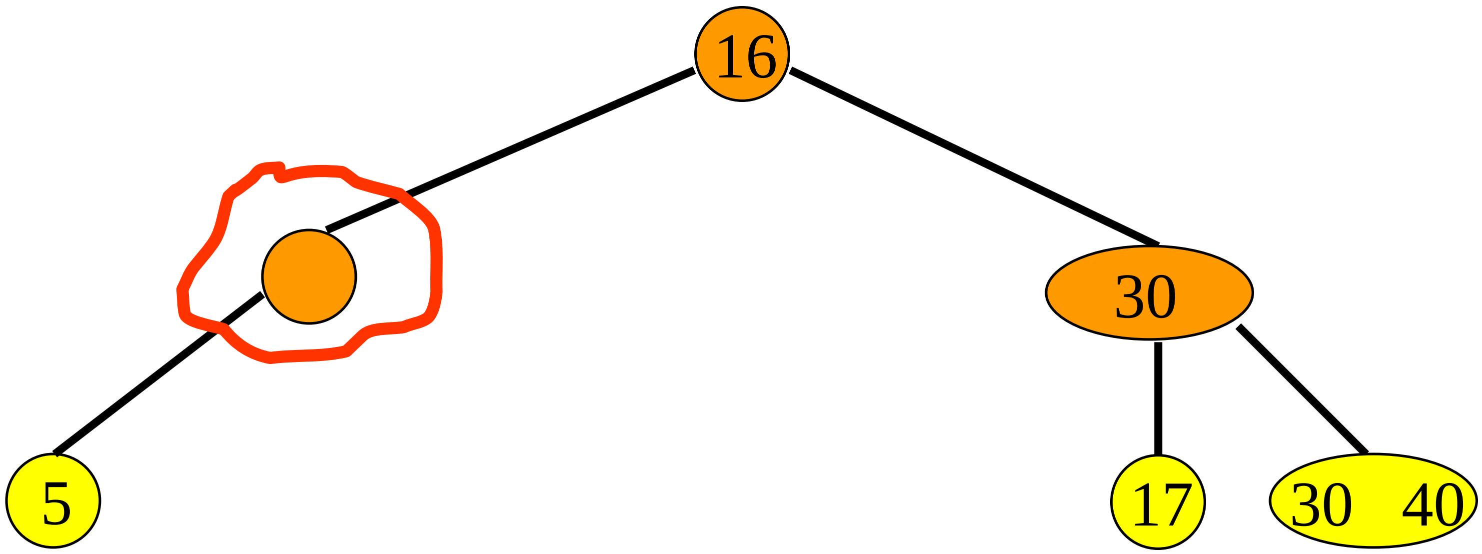
- Get >= 1 from sibling, move last one to parent, get parent key.

Delete



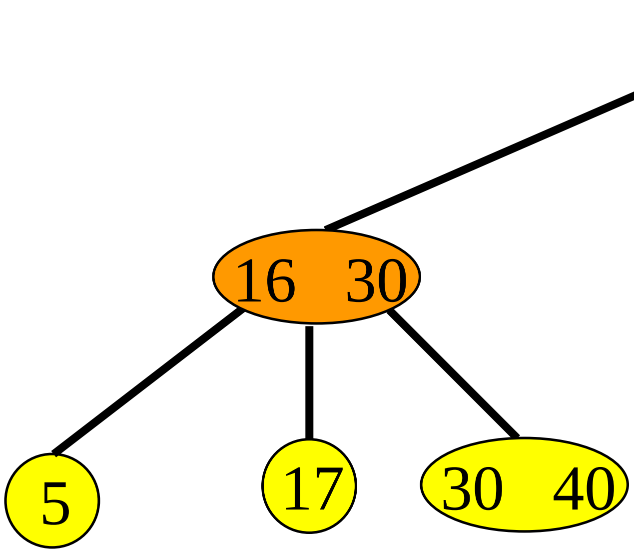
- Delete 9.
- Merge with sibling, delete in-between key in parent.

Delete



- Index node becomes deficient.
- Merge with sibling and in-between key in parent.

Delete



- Index node becomes deficient.
- It’s the root; discard.

(*Below not covered this year *)
B*-Trees

- Root has between 2 and 2 * floor((2m – 2)/3) + 1 children.
- Remaining nodes have between ceil((2m – 1)/3) and m children.
- All external/failure nodes are on the same level.

Insert

- When insert node is overfull, check adjacent sibling.
- If adjacent sibling is not full, move a dictionary pair from overfull node, via parent, to nonfull adjacent sibling.
- If adjacent sibling is full, split overfull node, adjacent full node, and in-between pair from parent to get three nodes with floor((2m – 2)/3), floor((2m – 1)/3), floor(2m/3) pairs plus two additional pairs for insertion into parent.

Delete

- When combining, must combine 3 adjacent nodes and 2 in-between pairs from parent.
  - Total # pairs involved = 2 * floor((2m-2)/3) + [floor((2m-2)/3) – 1] + 2.
  - Equals 3 * floor((2m-2)/3) + 1.
- Combining yields 2 nodes and a pair that is to be inserted into the parent.
  - m mod 3 = 0 => nodes have m – 1 pairs each.
  - m mod 3 = 1 => one node has m – 1 pairs and the other has m – 2.
  - m mod 3 = 2 => nodes have m – 2 pairs each.

Digital Search Trees & Binary Tries

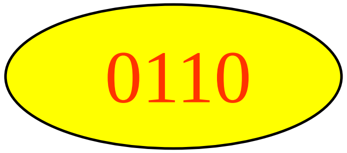
- Analog of radix sort to searching.
- Keys are binary bit strings.
  - Fixed length – 0110, 0010, 1010, 1011.
  - Variable length – 01, 00, 101, 1011.
- Application – IP routing, packet classification, firewalls.
  - IPv4 – 32 bit IP address.
  - IPv6 – 128 bit IP address.

Digital Search Tree

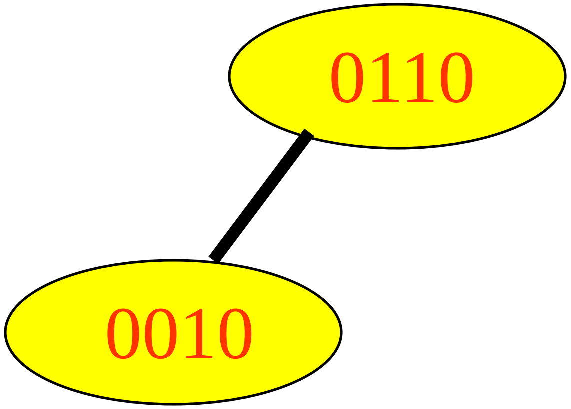
- Assume fixed number of bits.
- Not empty =>
  - Root contains one dictionary pair (any pair).
  - All remaining pairs whose key begins with a 0 are in the left subtree.
  - All remaining pairs whose key begins with a 1 are in the right subtree.
  - Left and right subtrees are digital subtrees on remaining bits.

Example

- Start with an empty digital search tree and insert a pair whose key is 0110.

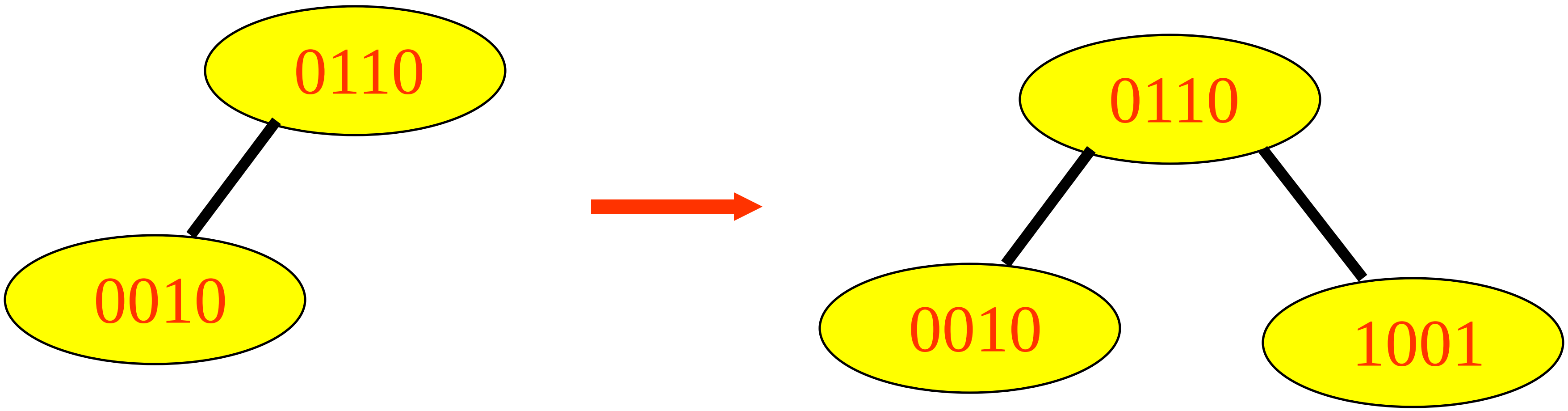


- Now, insert a pair whose key is 0010.



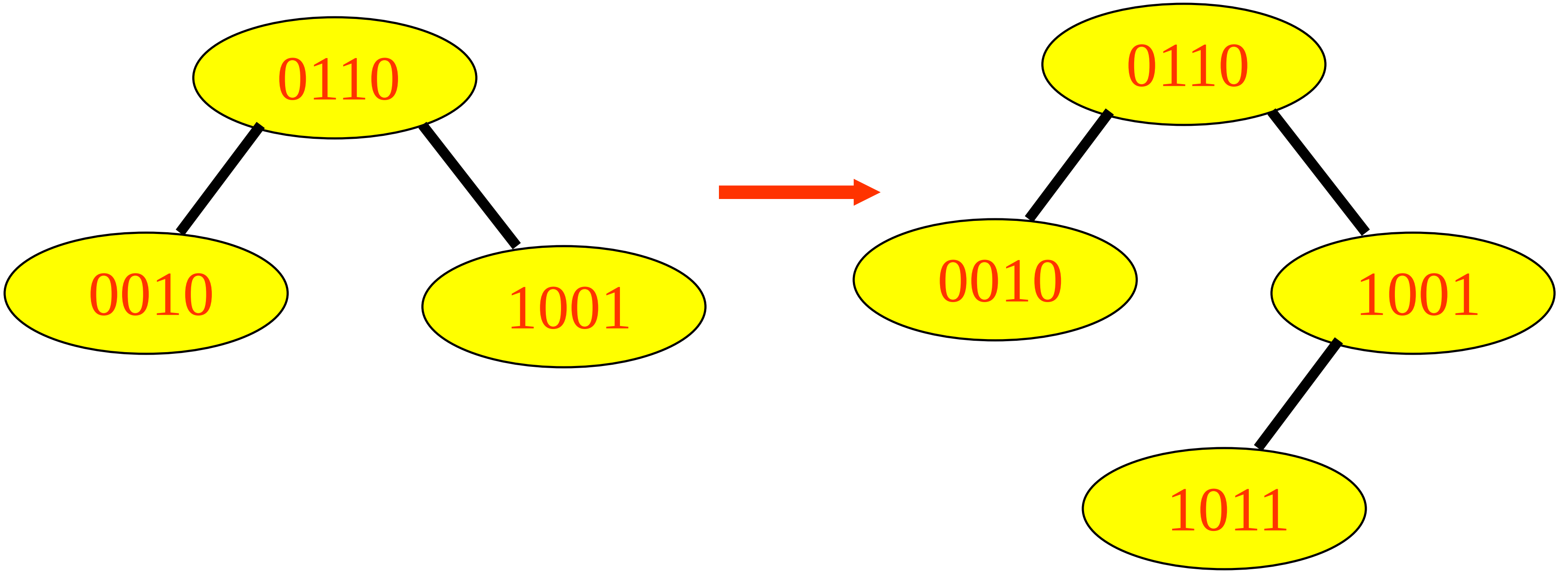
Example

- Now, insert a pair whose key is 1001.



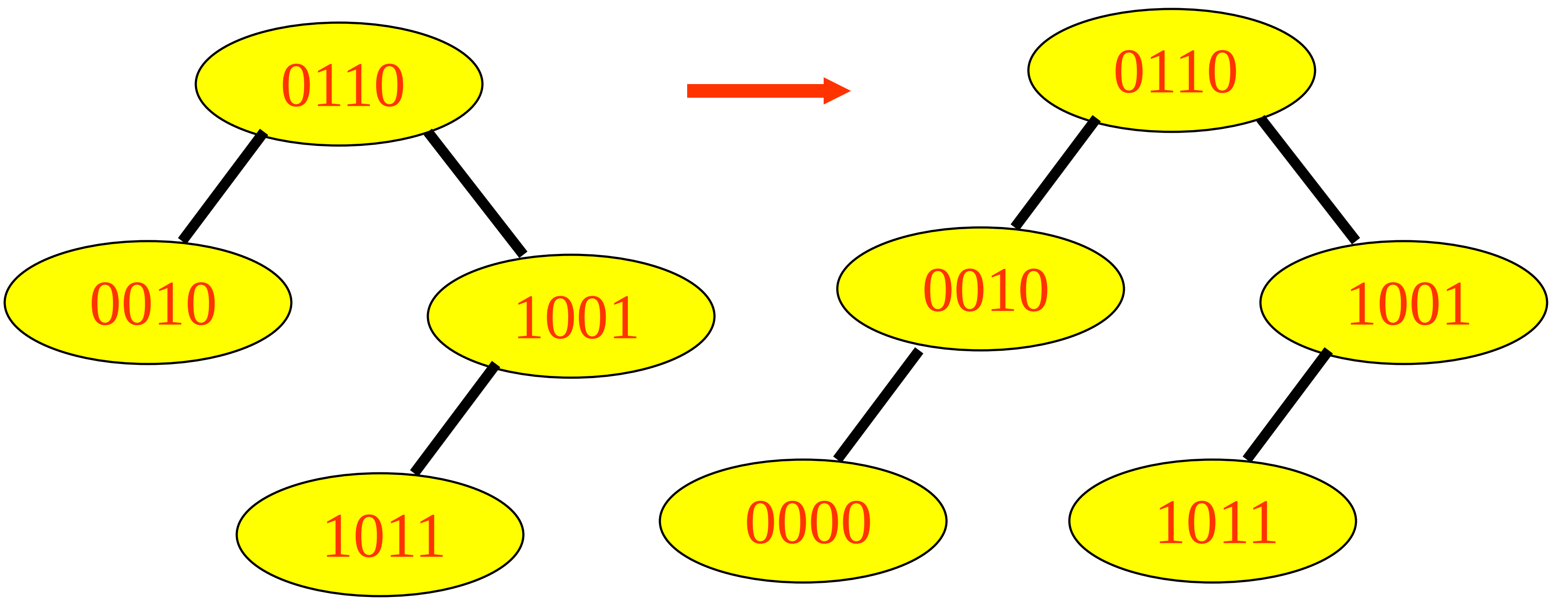
Example

- Now, insert a pair whose key is 1011.

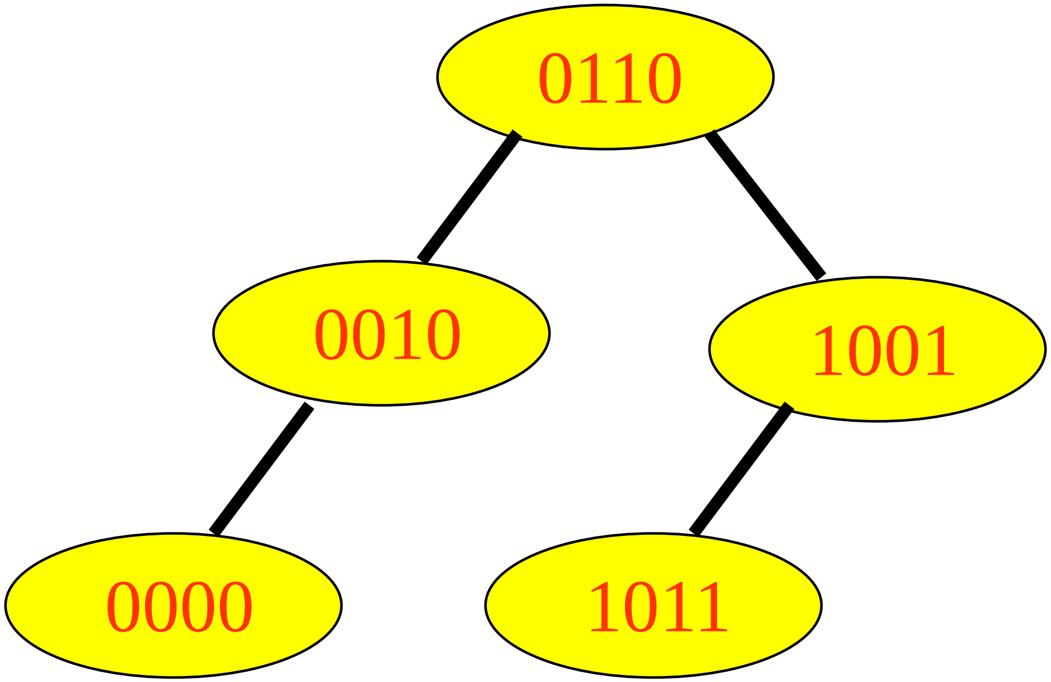


Example

- Now, insert a pair whose key is 0000.



Search/Insert/Delete

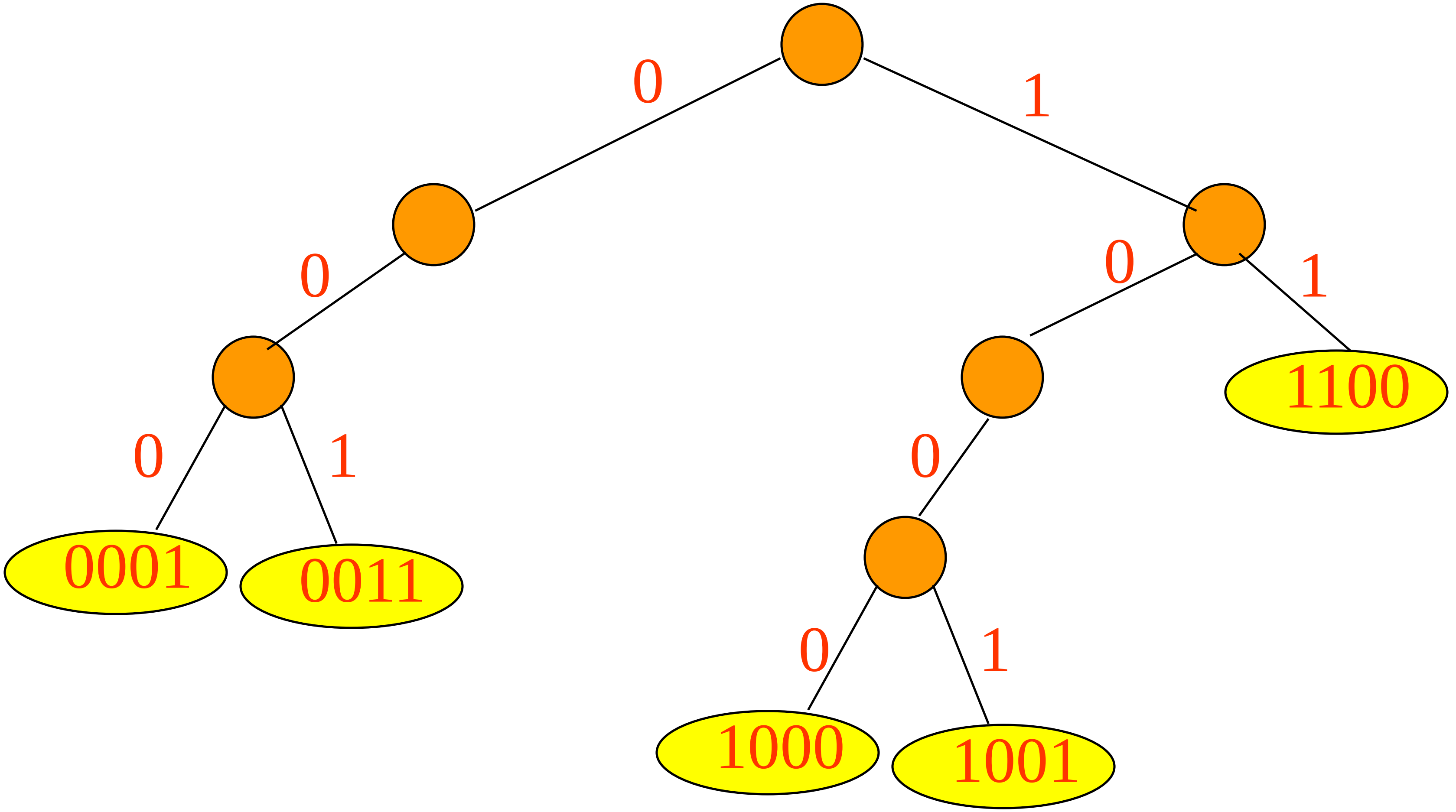


- Complexity of each operation is O(#bits in a key).
- #key comparisons = O(height).
- Expensive when keys are very long.

Binary Trie

- Information Retrieval.
- At most one key comparison per operation.
- Fixed length keys.
  - Branch nodes.
    - Left and right child pointers.
    - No data field(s).
  - Element nodes.
    - No child pointers.
    - Data field to hold dictionary pair.

Example

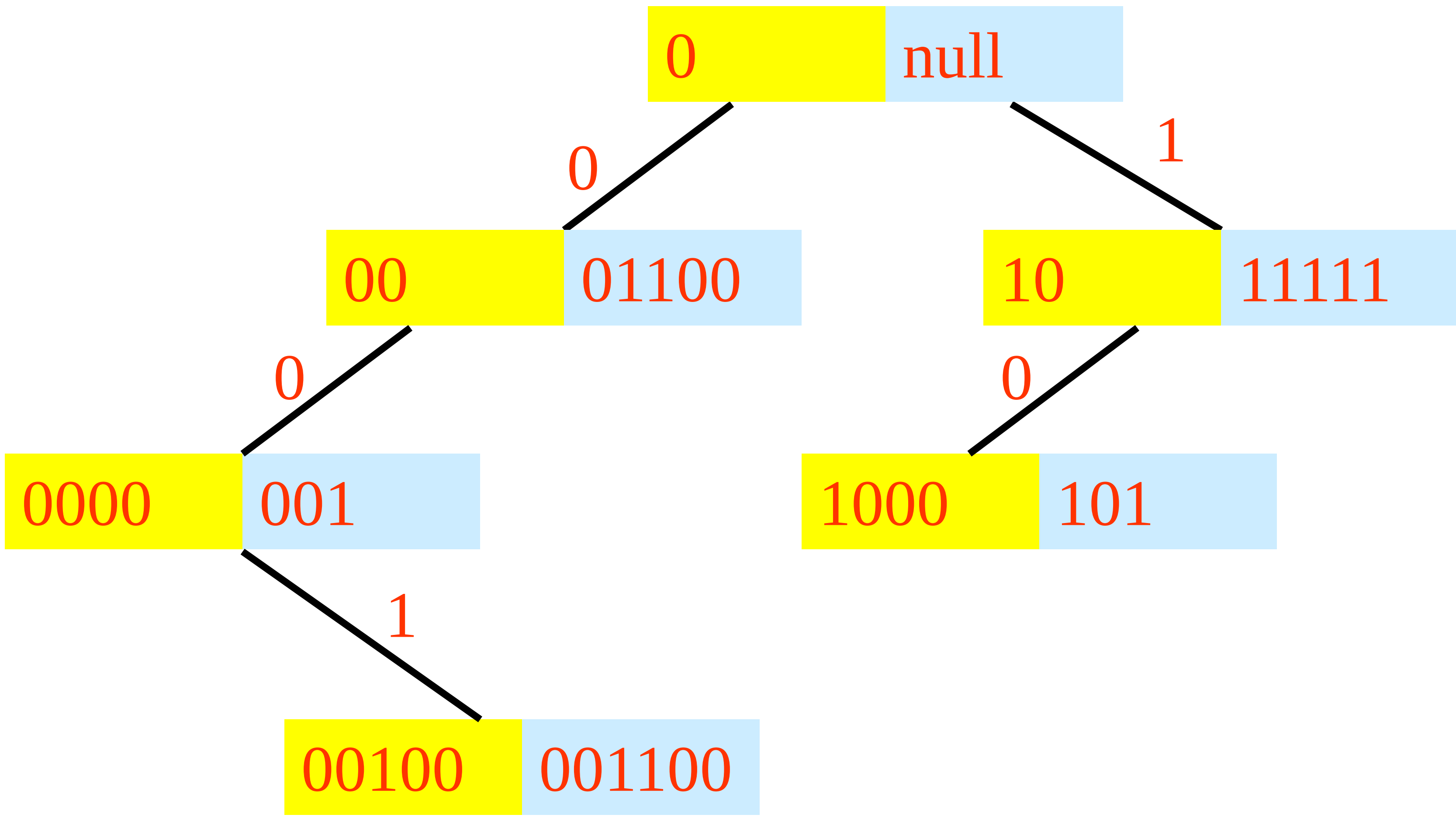


At most one key comparison for a search.

Variable Key Length

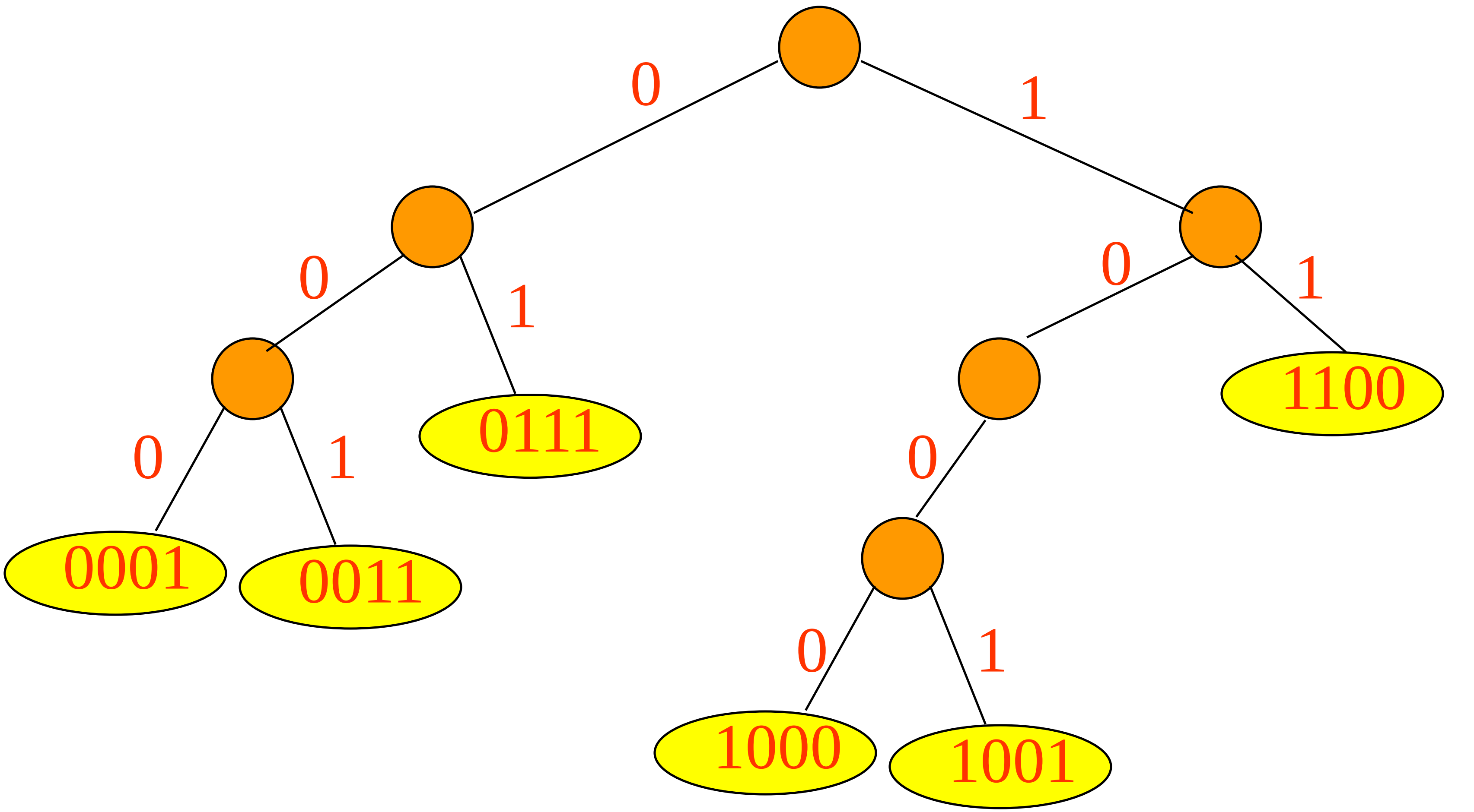
- Left and right child fields.
- Left and right pair fields.
  - Left pair is pair whose key terminates at root of left subtree or the single pair that might otherwise be in the left subtree.
  - Right pair is pair whose key terminates at root of right subtree or the single pair that might otherwise be in the right subtree.
  - Field is null otherwise.

Example



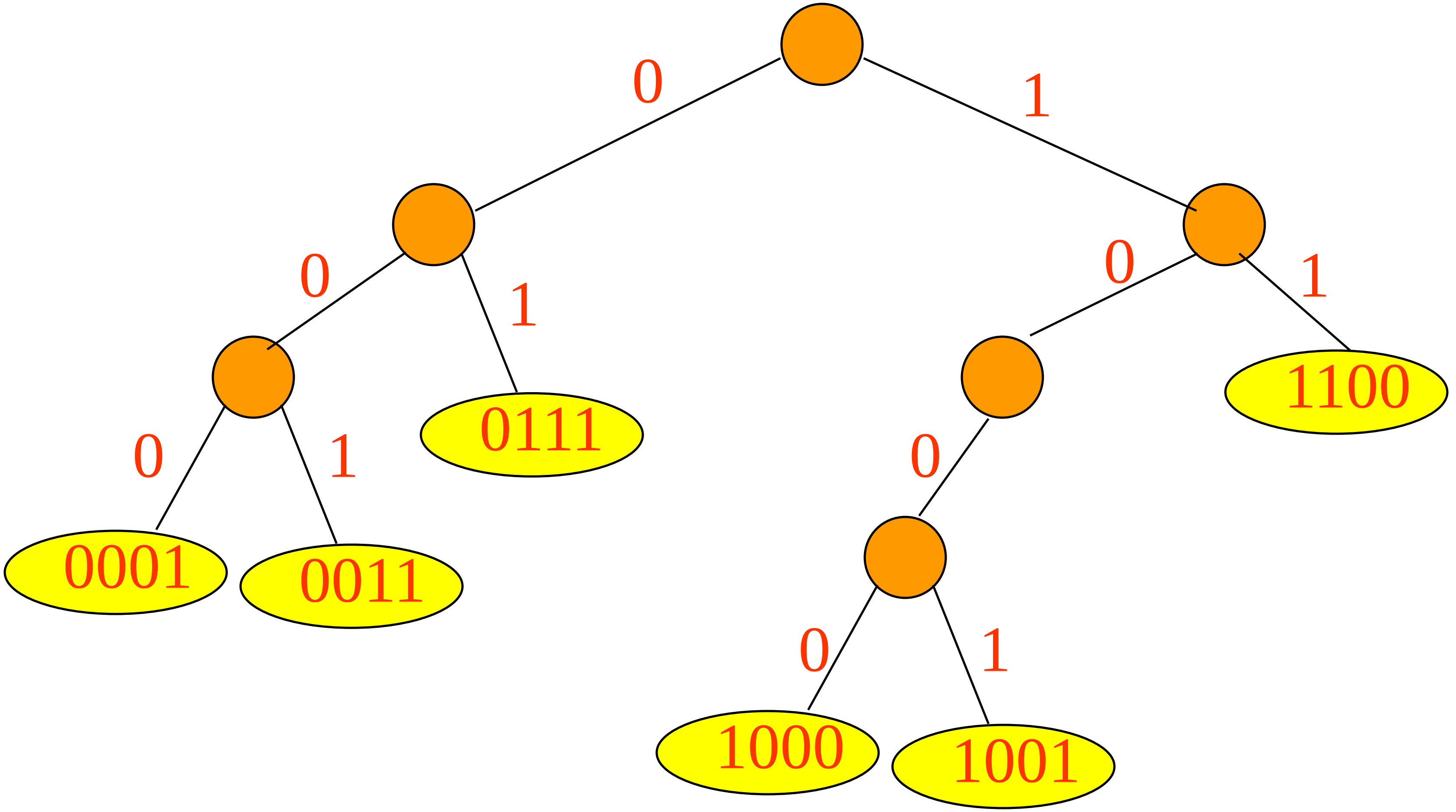
At most one key comparison for a search.

Fixed Length Insert



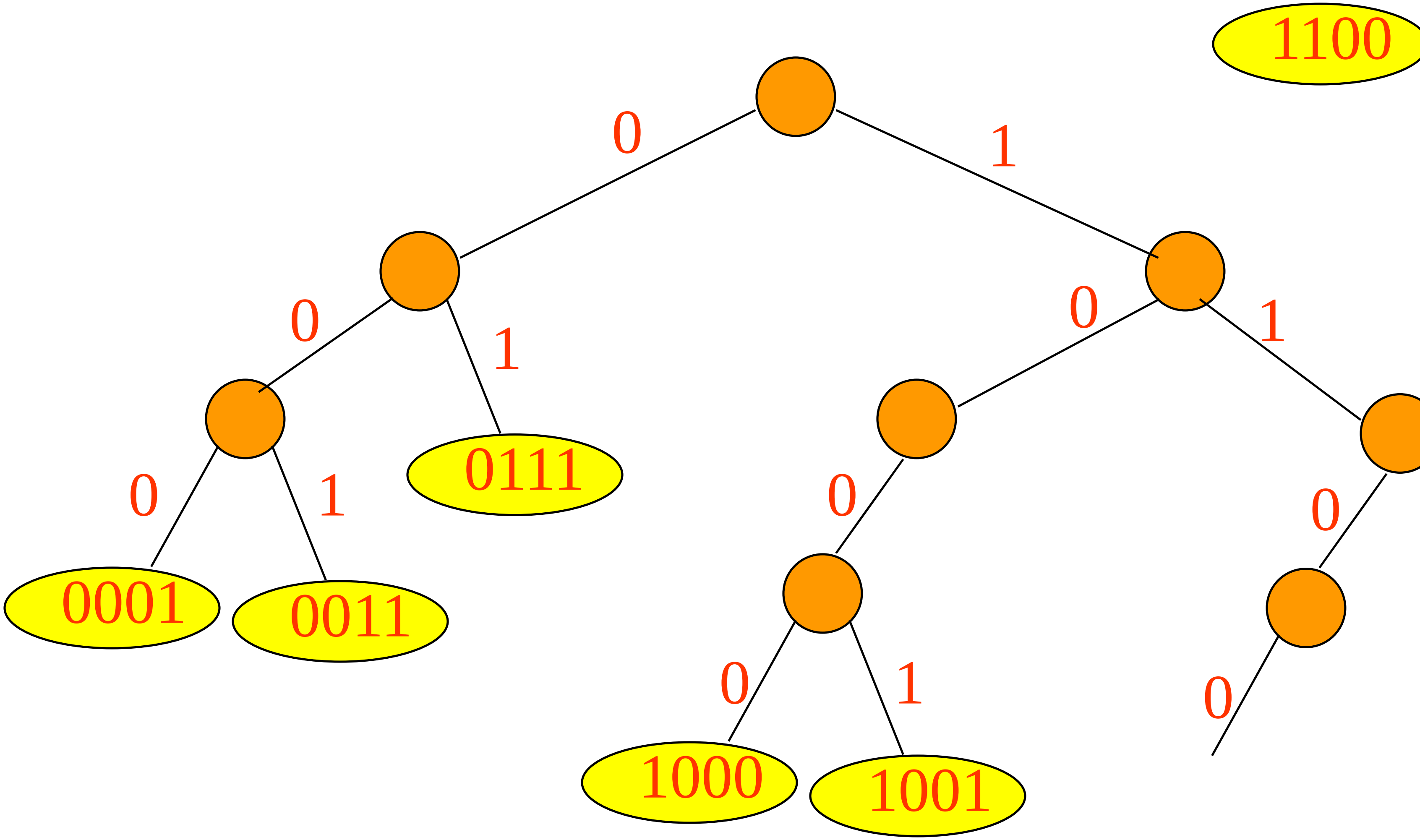
Insert 0111. Zero compares.

Fixed Length Insert



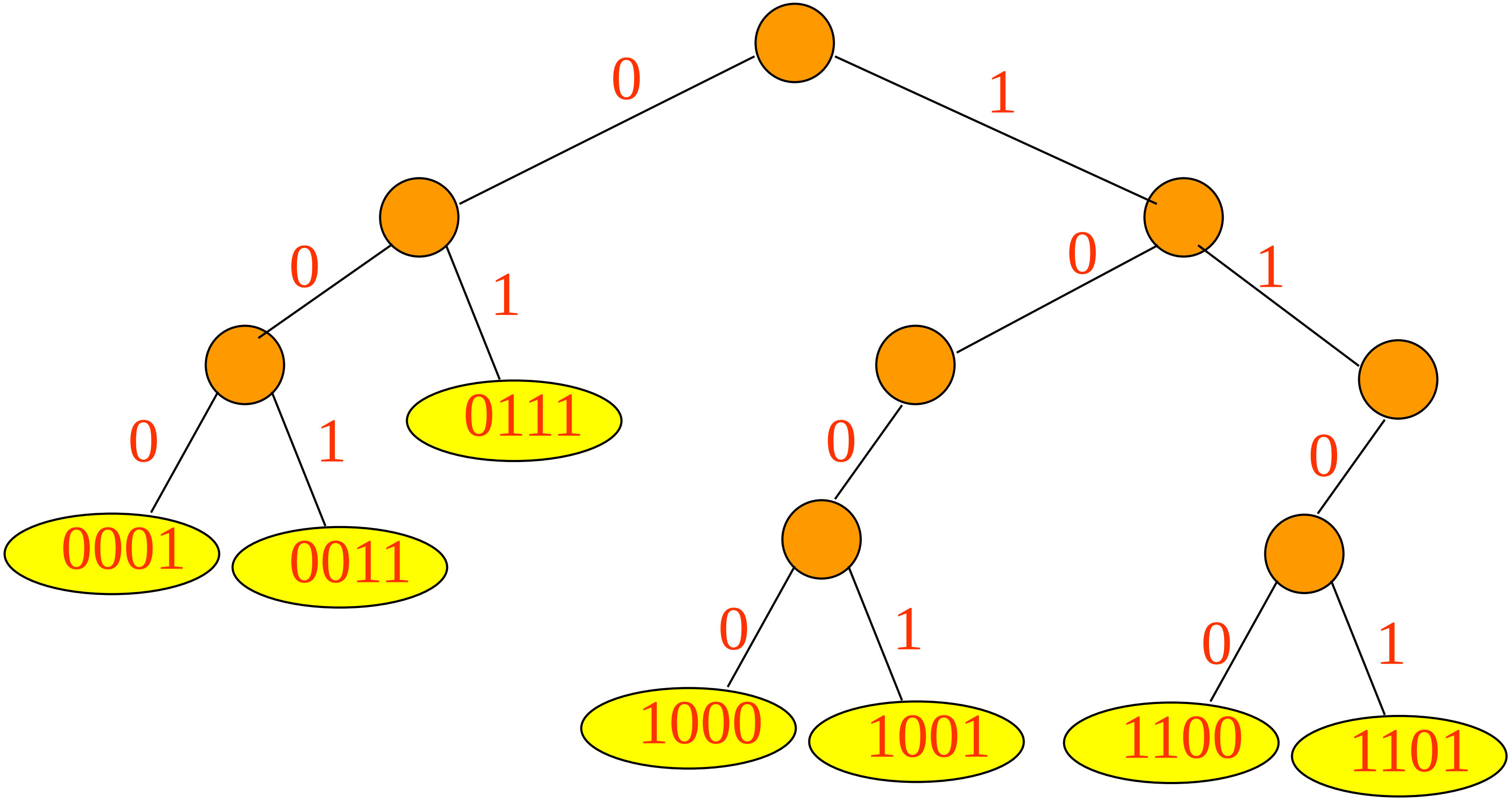
Insert 1101.

Fixed Length Insert



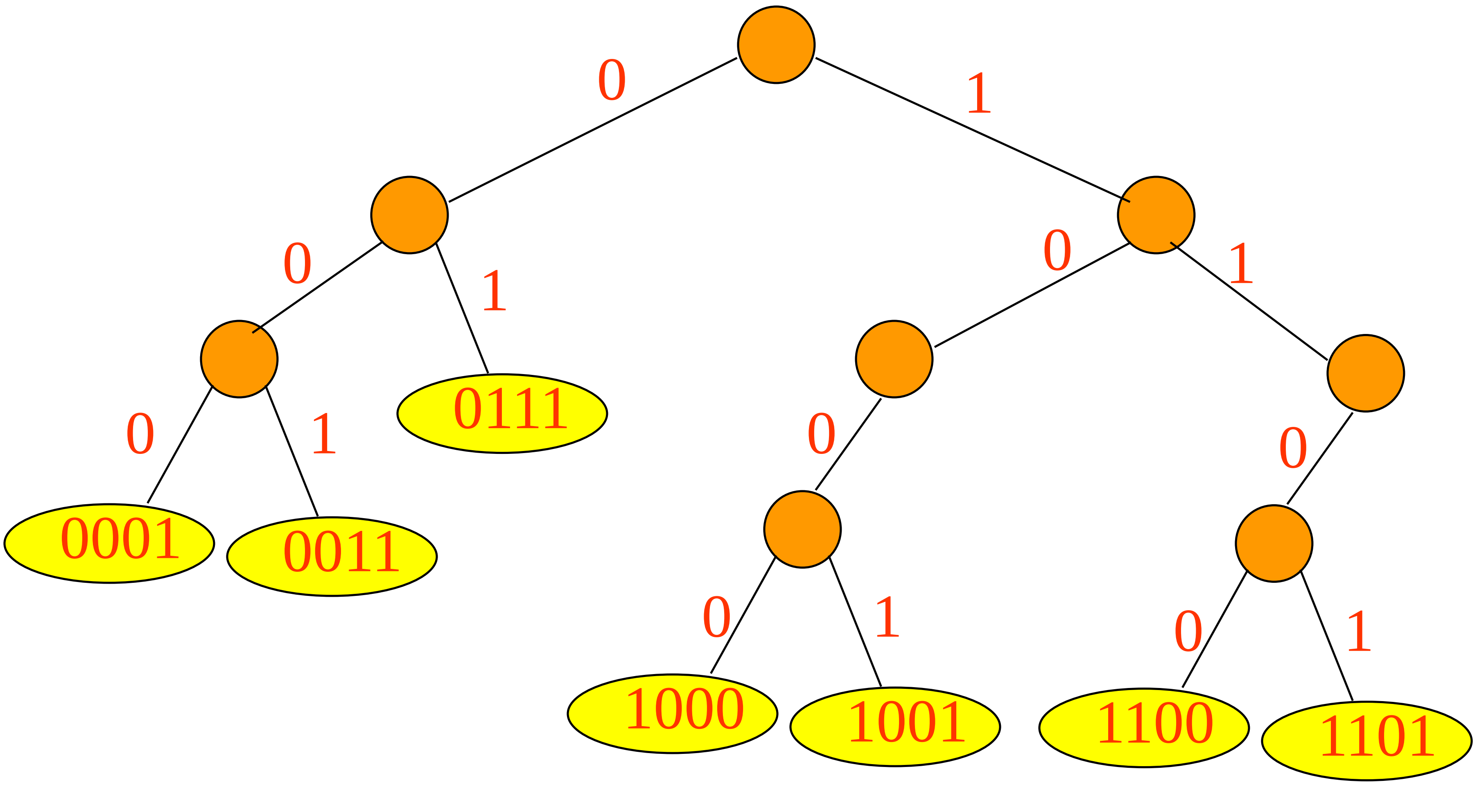
Insert 1101.

Fixed Length Insert



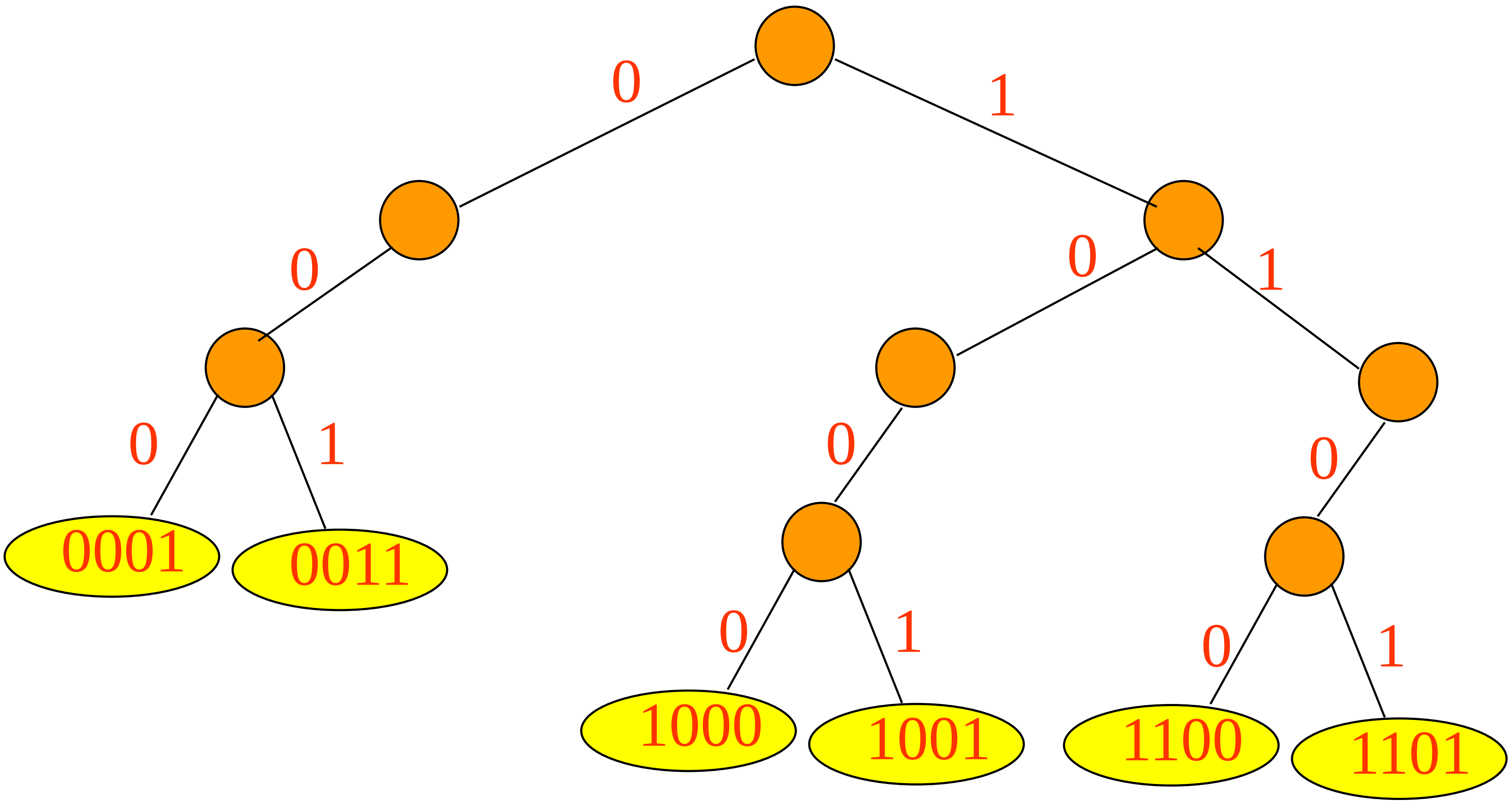
Insert 1101. One compare.

Fixed Length Delete



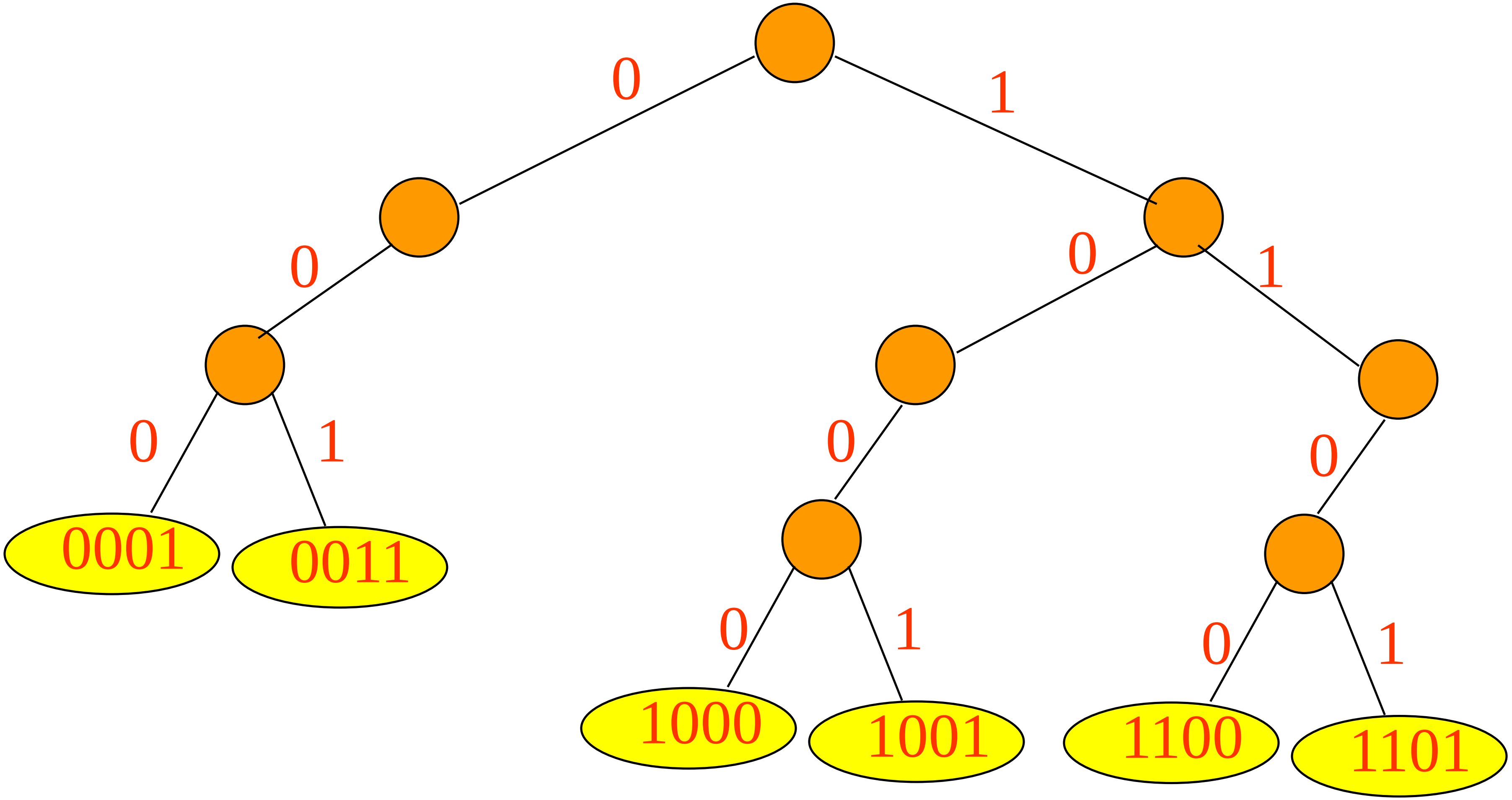
Delete 0111.

Fixed Length Delete



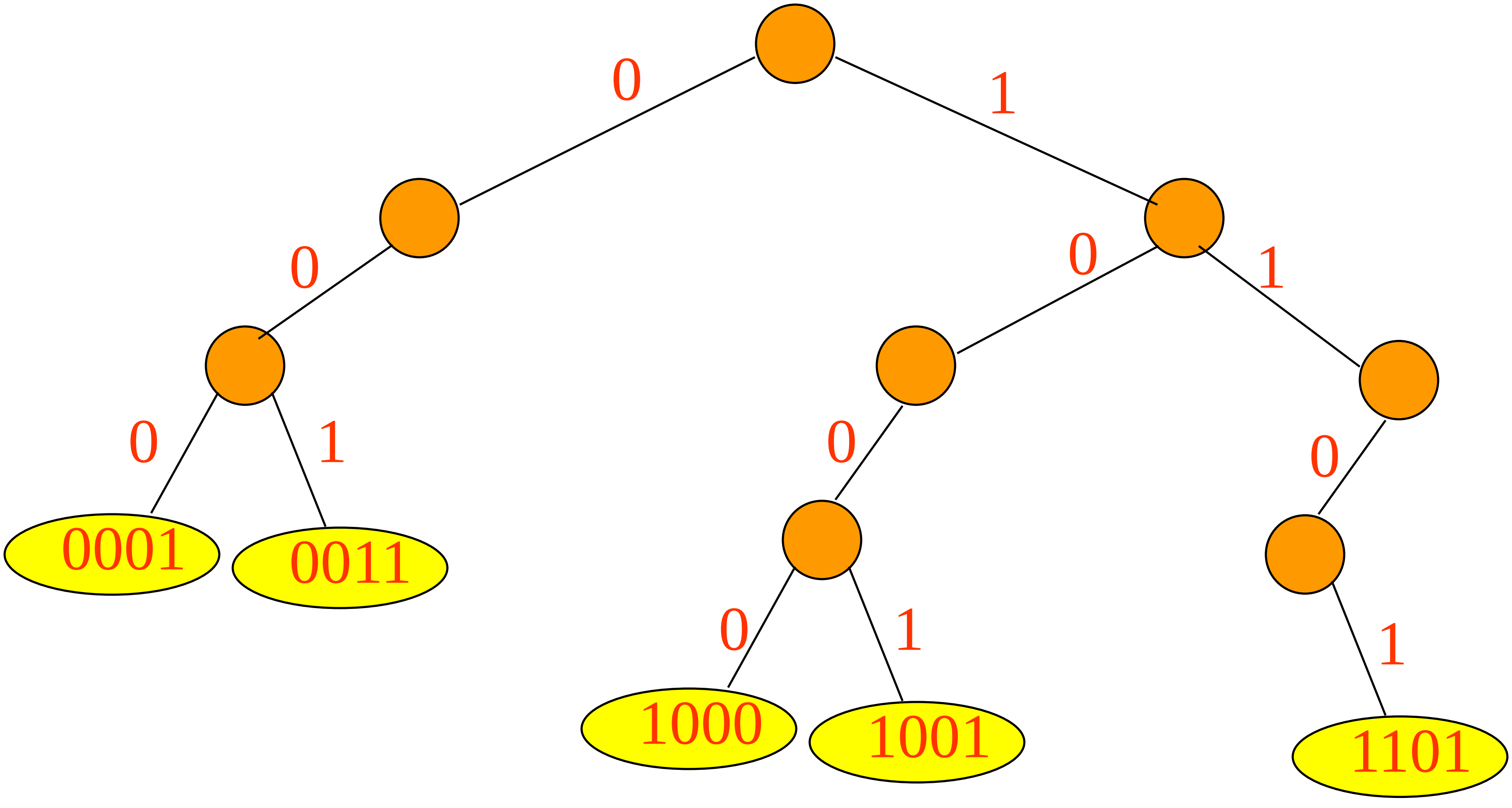
Delete 0111. One compare.

Fixed Length Delete



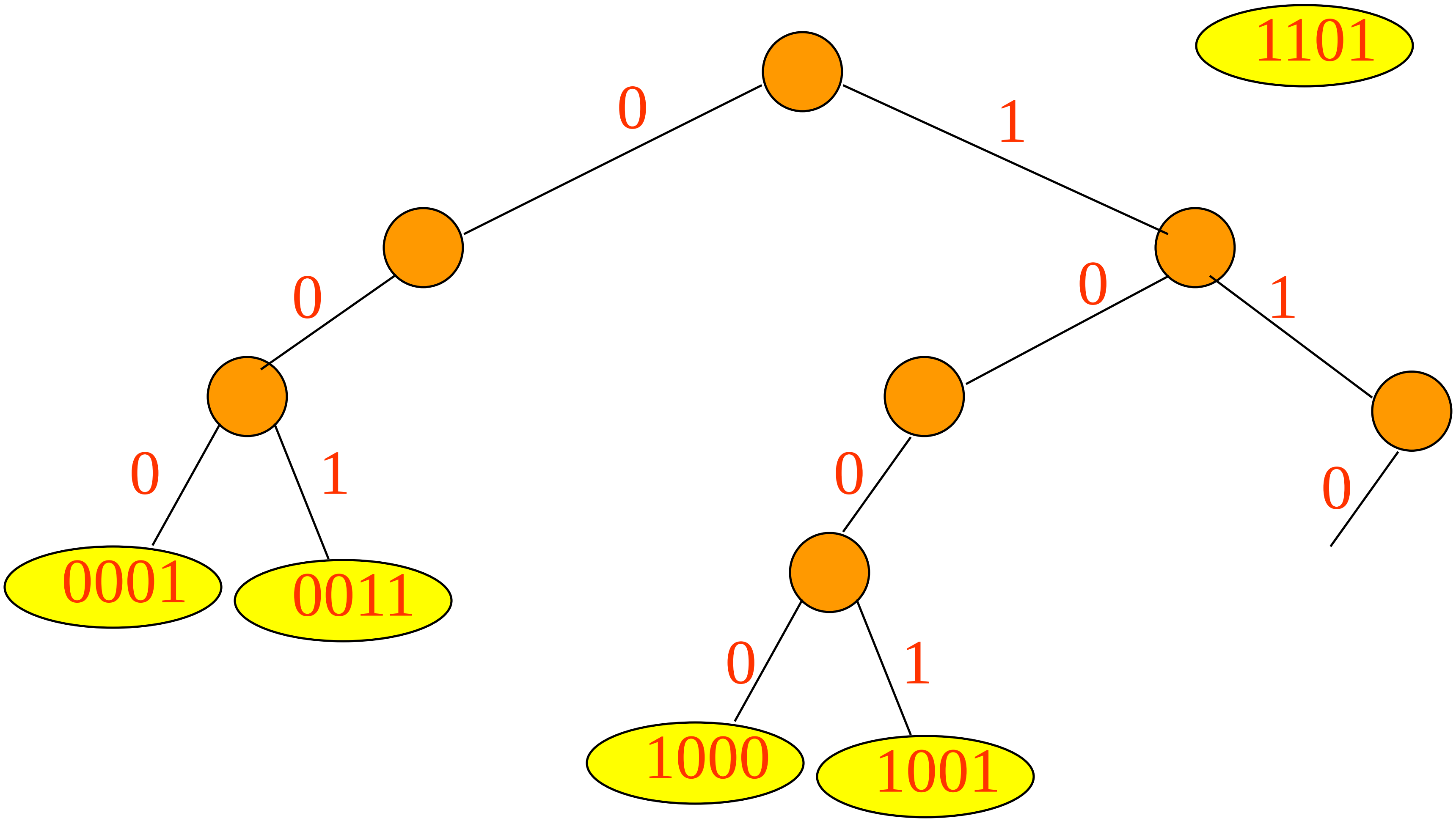
Delete 1100.

Fixed Length Delete



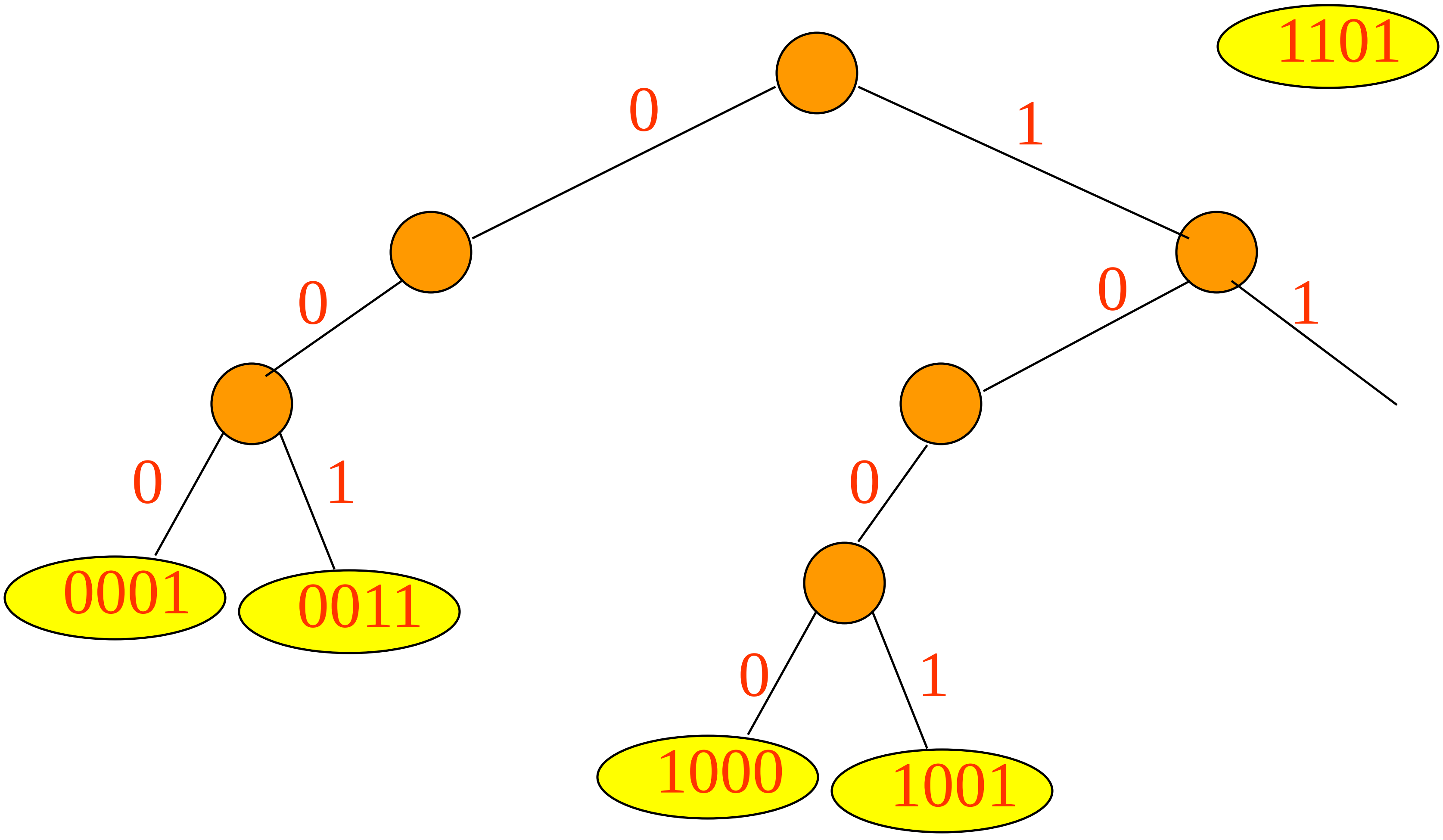
Delete 1100.

Fixed Length Delete



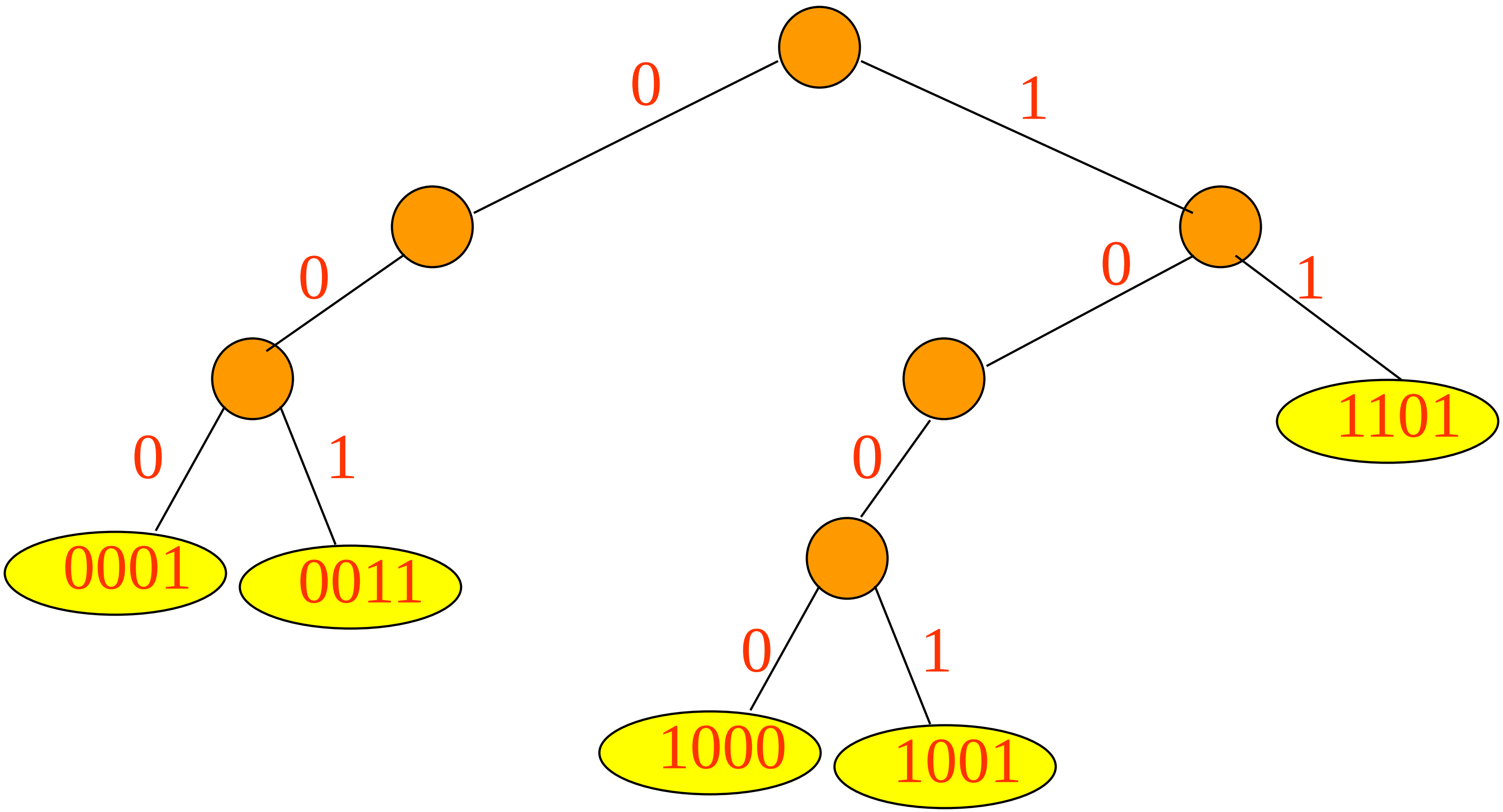
Delete 1100.

Fixed Length Delete



Delete 1100.

Fixed Length Delete



Delete 1100.

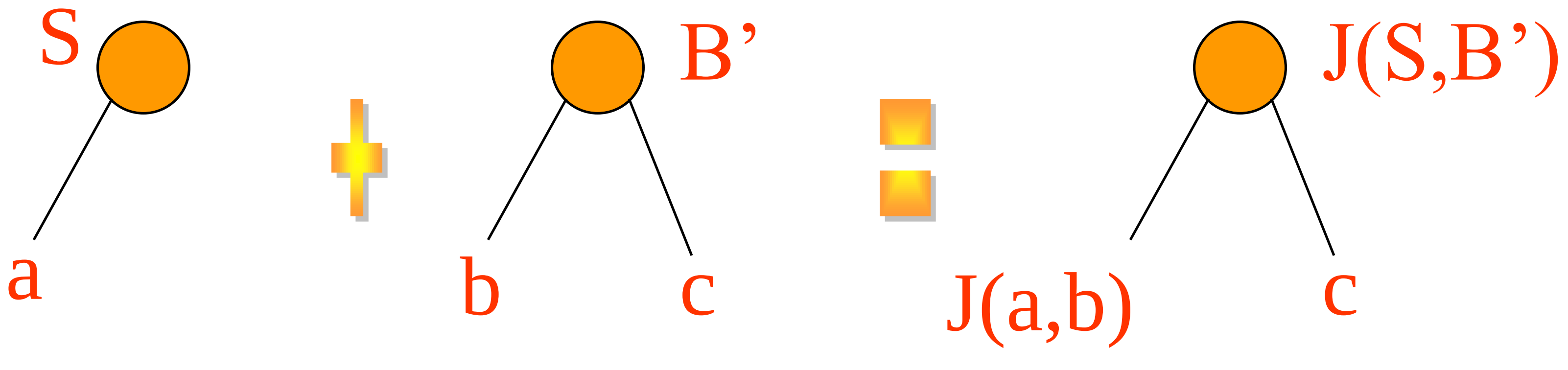
One compare.

Fixed Length Join(S,m,B)

- Insert m into B to get B’.
- S empty => B’ is answer; done.
- S is element node => insert S element into B’; done;
- B’ is element node => insert B’ element into S; done;
- If you get to this step, the roots of S and B’ are branch nodes.

Fixed Length Join(S,m,B)

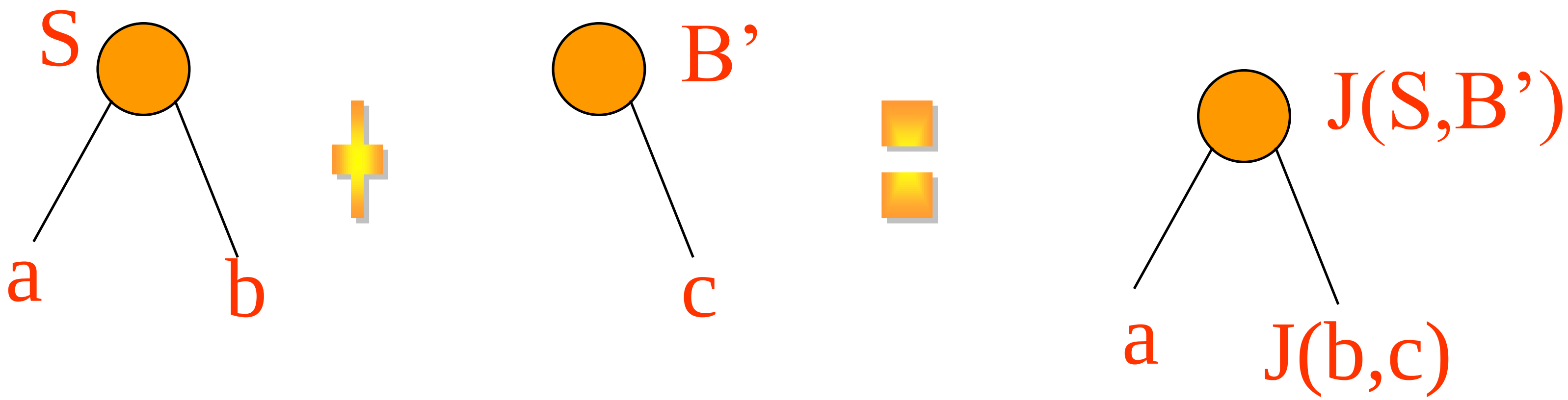
- S has empty right subtree.



J(X,Y) ⇨ join X and Y, all keys in X < all in Y.

Fixed Length Join(S,m,B)

- S has nonempty right subtree.
- Left subtree of B' must be empty, because all keys in B' > all keys in S.



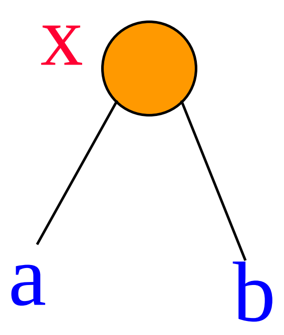
Complexity = O(height).

Binary Tries (continued)

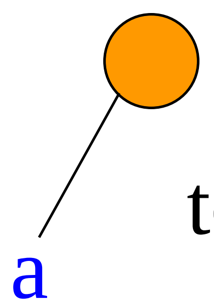
- split(k).
- Similar to split algorithm for unbalanced binary search trees.
- Construct S and B on way down the trie.
- Follow with a backward cleanup pass over the constructed S and B.

Forward Pass

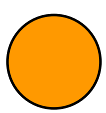
- Suppose you are at node x, which is at level j of the input trie.



- If bit j of k is 1, move to root of b and add

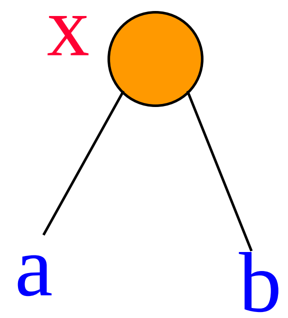


to level j of S and

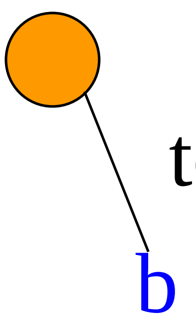


to level j of B.

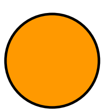
Forward Pass



- If bit j of k is 0, move to root of a and add



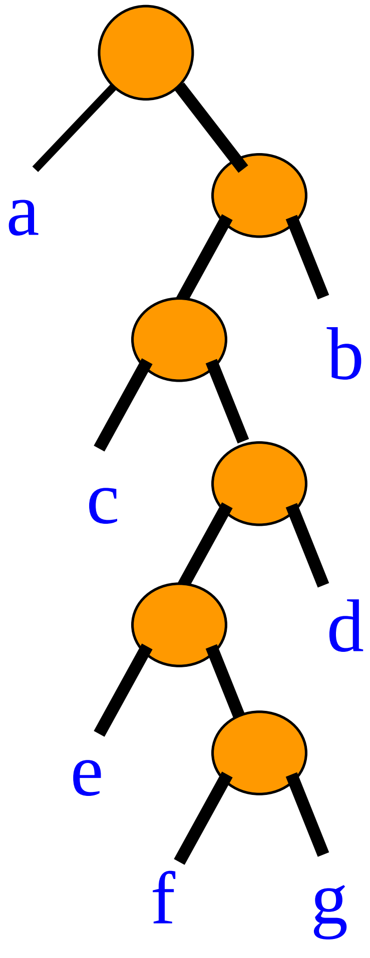
to level j of B and



to level j of S.

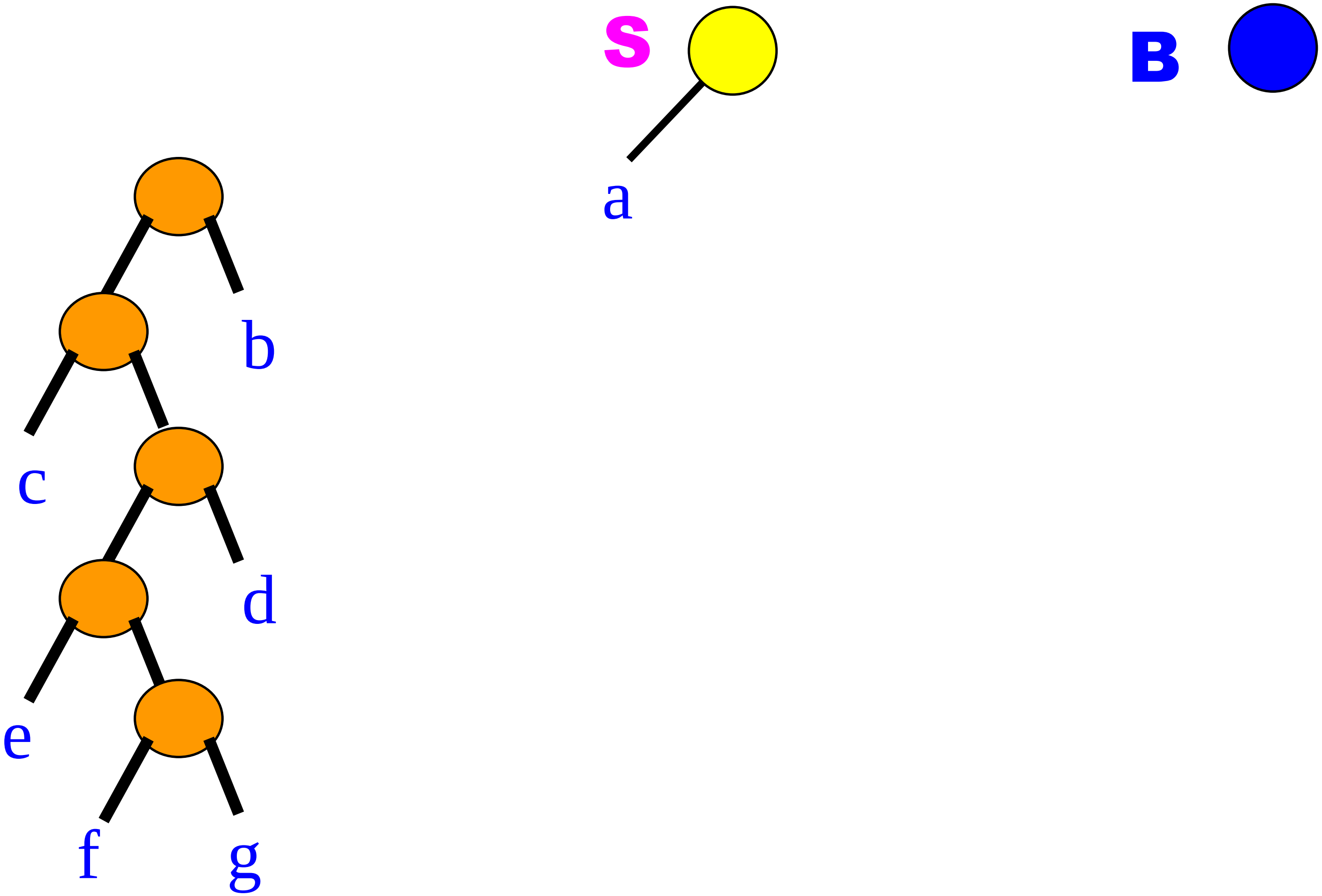
Forward Pass Example

S = null **B = null**



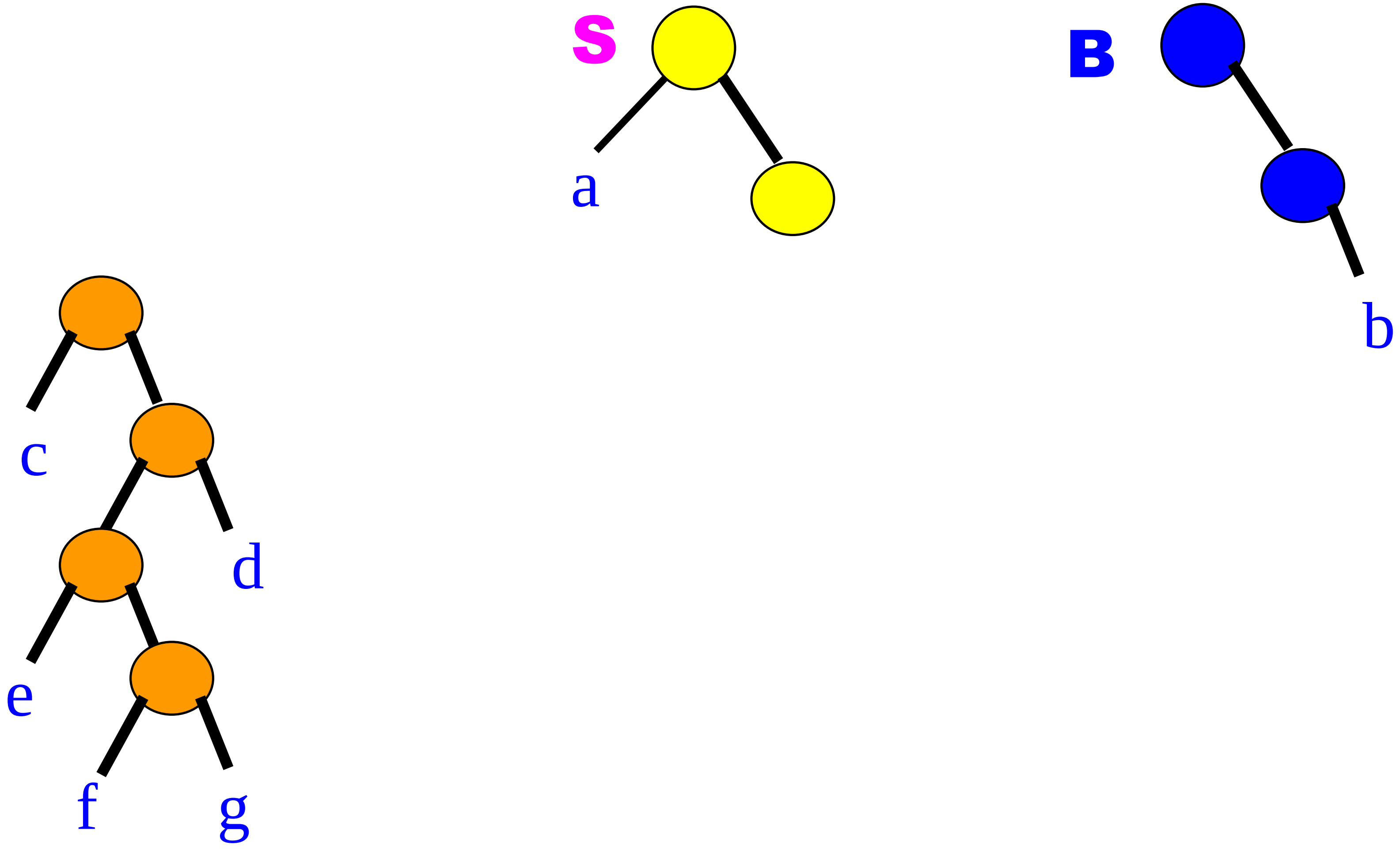
k = g.key = 101011

Forward Pass Example



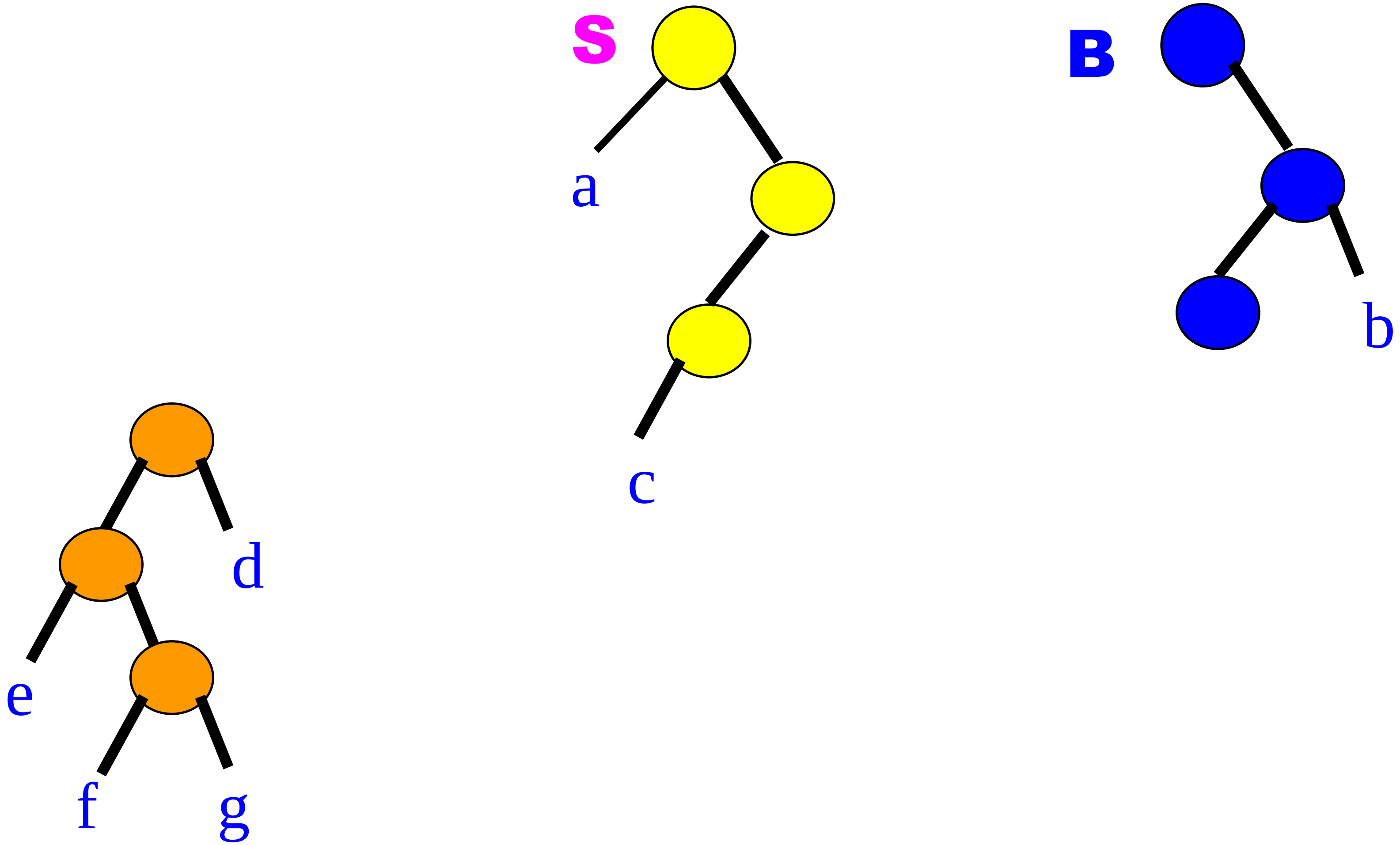
k = g.key = 101011

Forward Pass Example



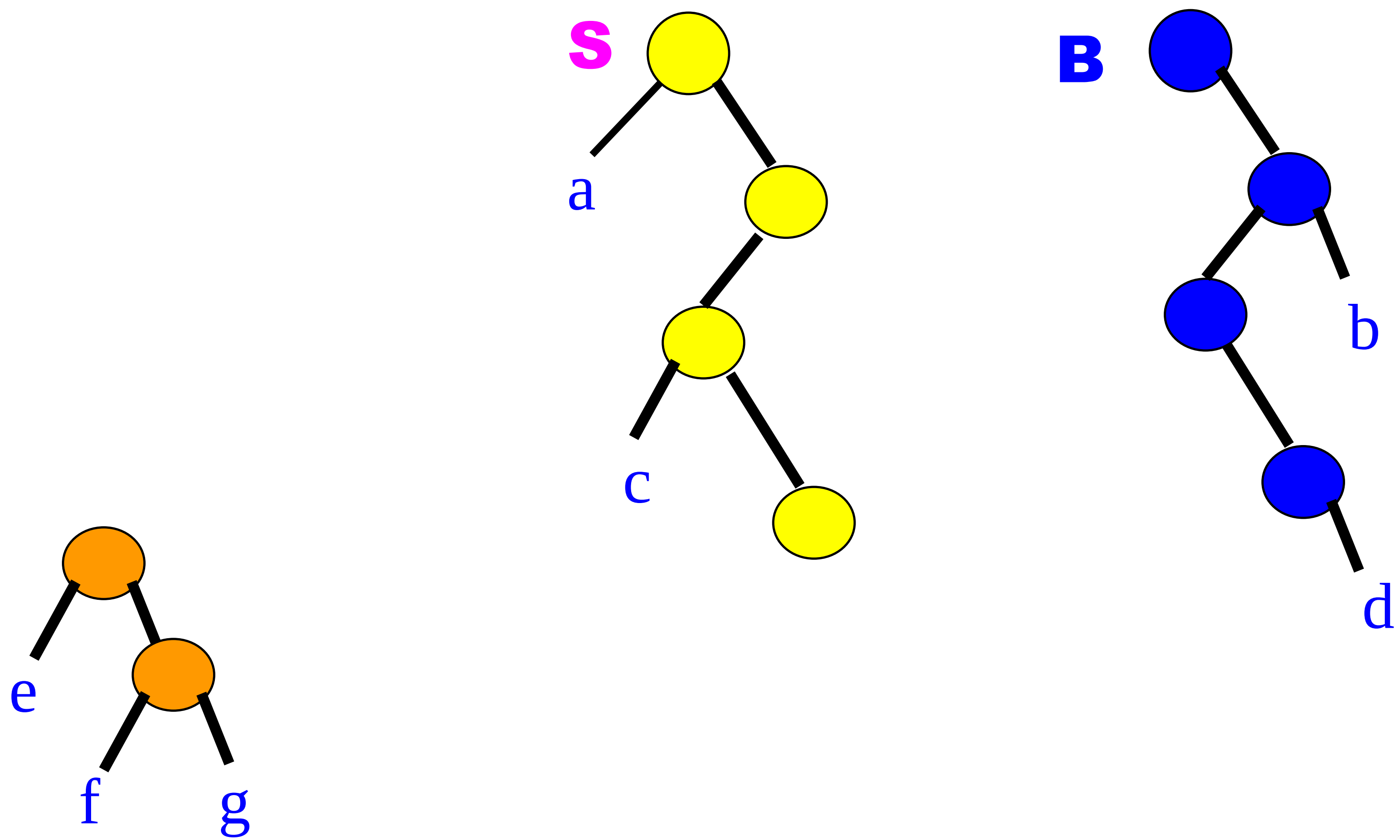
k = g.key = 101011

Forward Pass Example



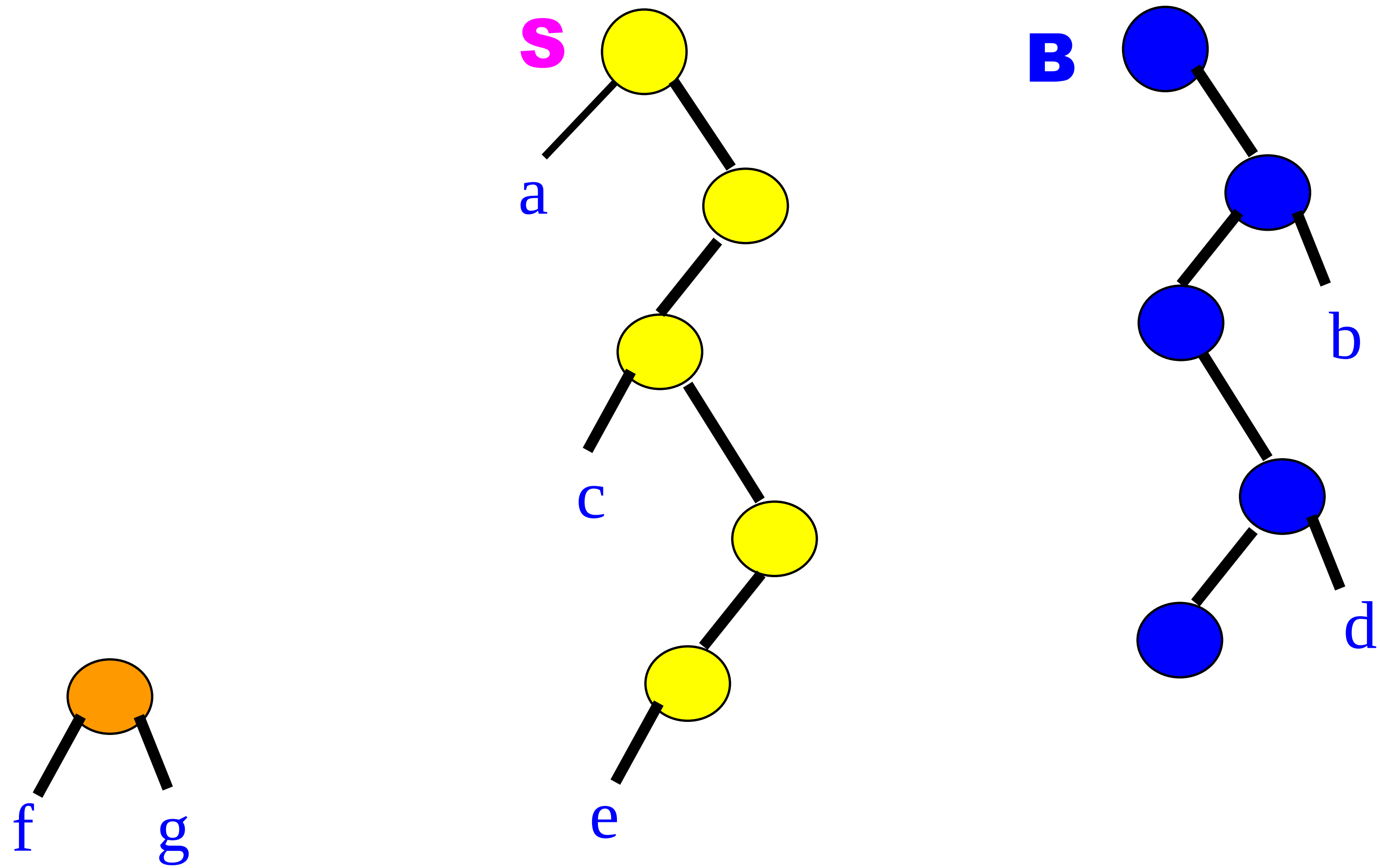
k = g.key = 101011

Forward Pass Example



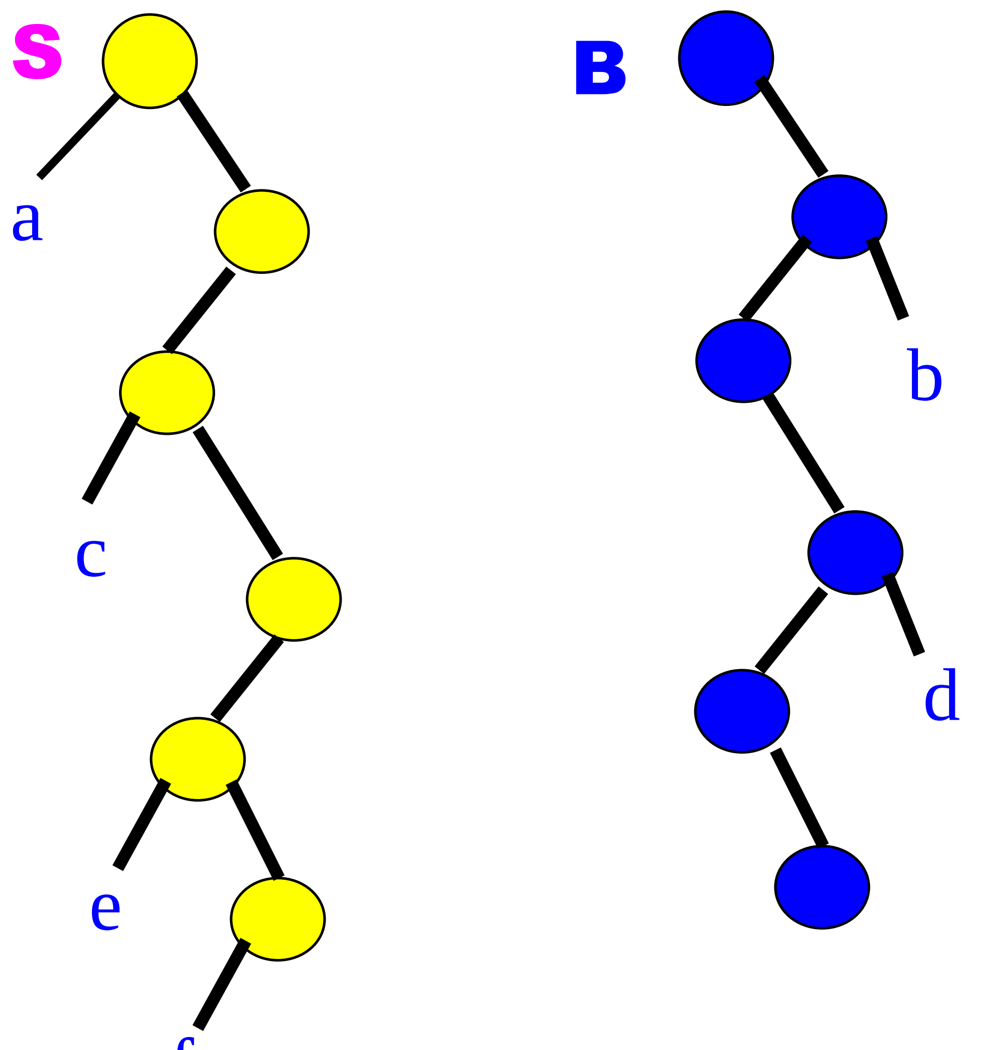
k = g.key = 101011

Forward Pass Example



k = g.key = 101011

Forward Pass Example



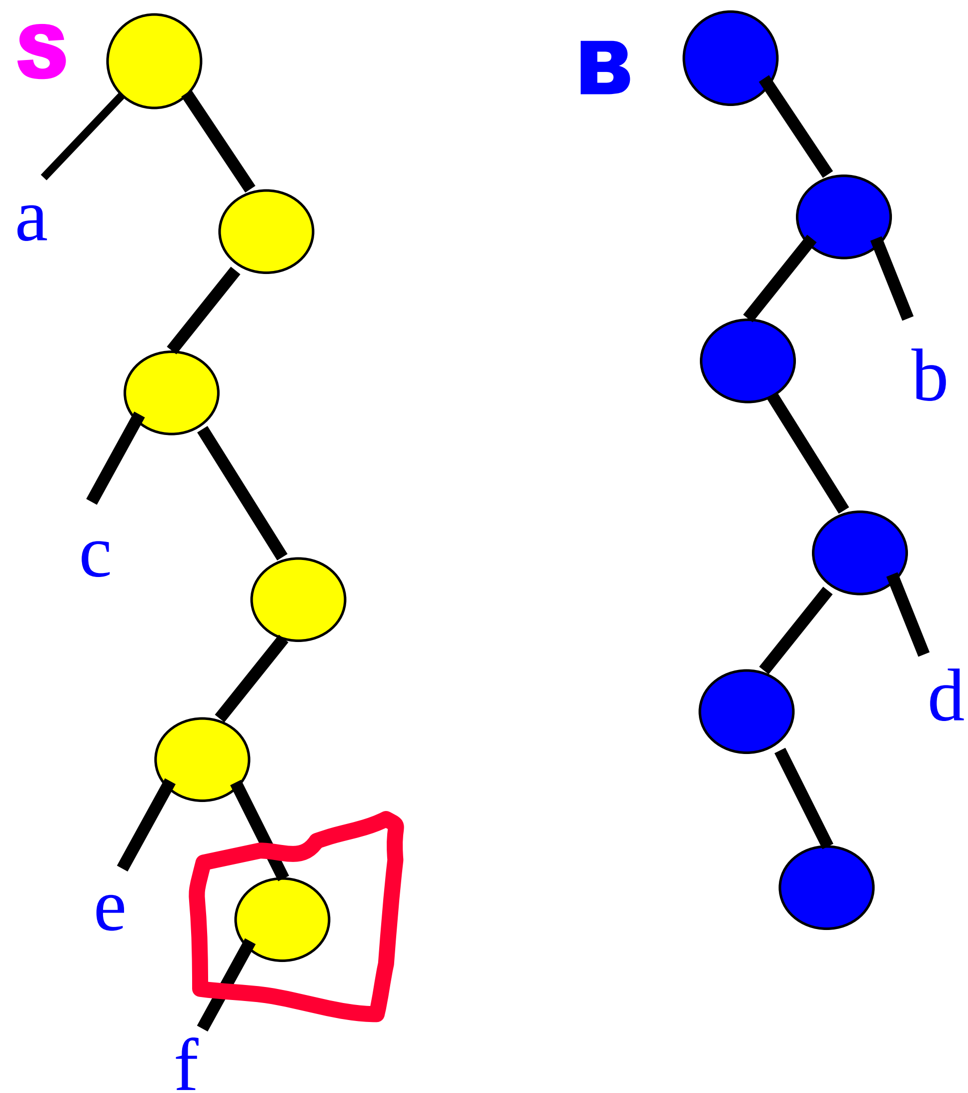
k = g.key = 101011

Backward Cleanup Pass

- Retrace path from current nodes in S and B toward roots of respective tries.
- Eliminate branch nodes that are roots of subtries that have fewer than 2 dictionary pairs.

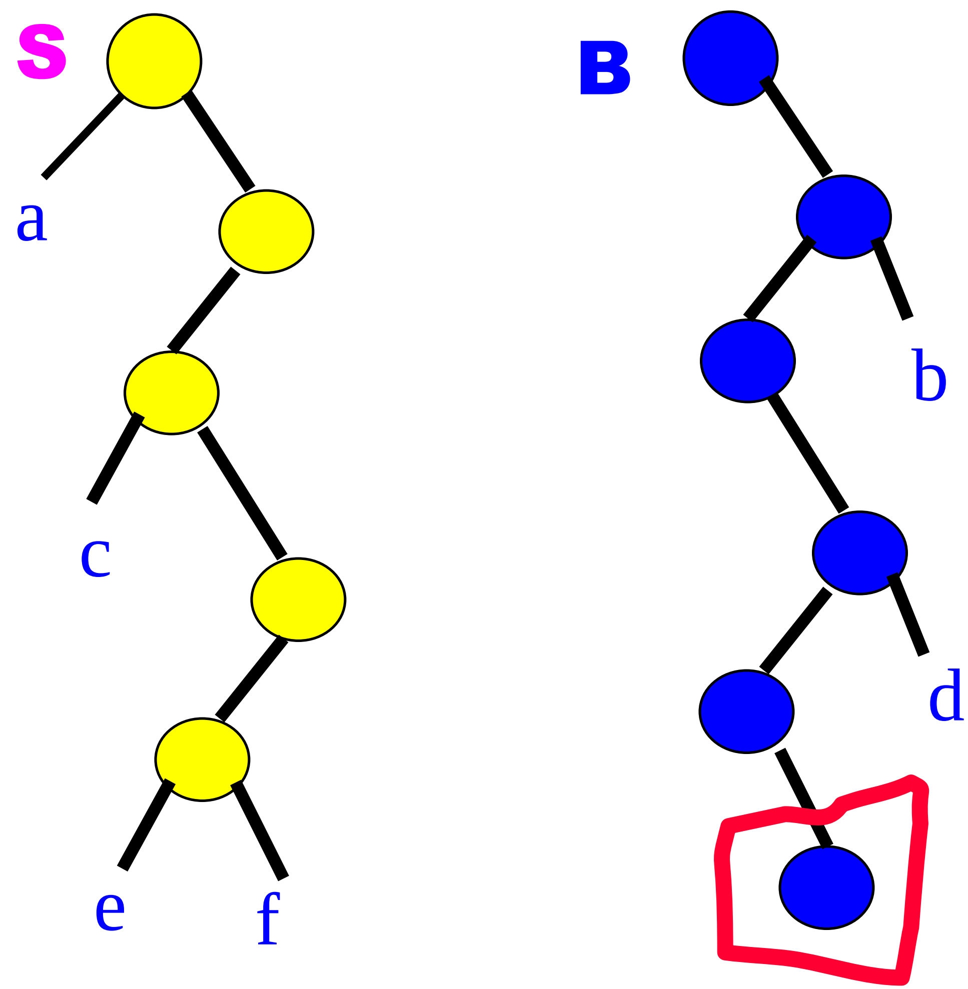
Backward Cleanup Pass Example

f is an element node.



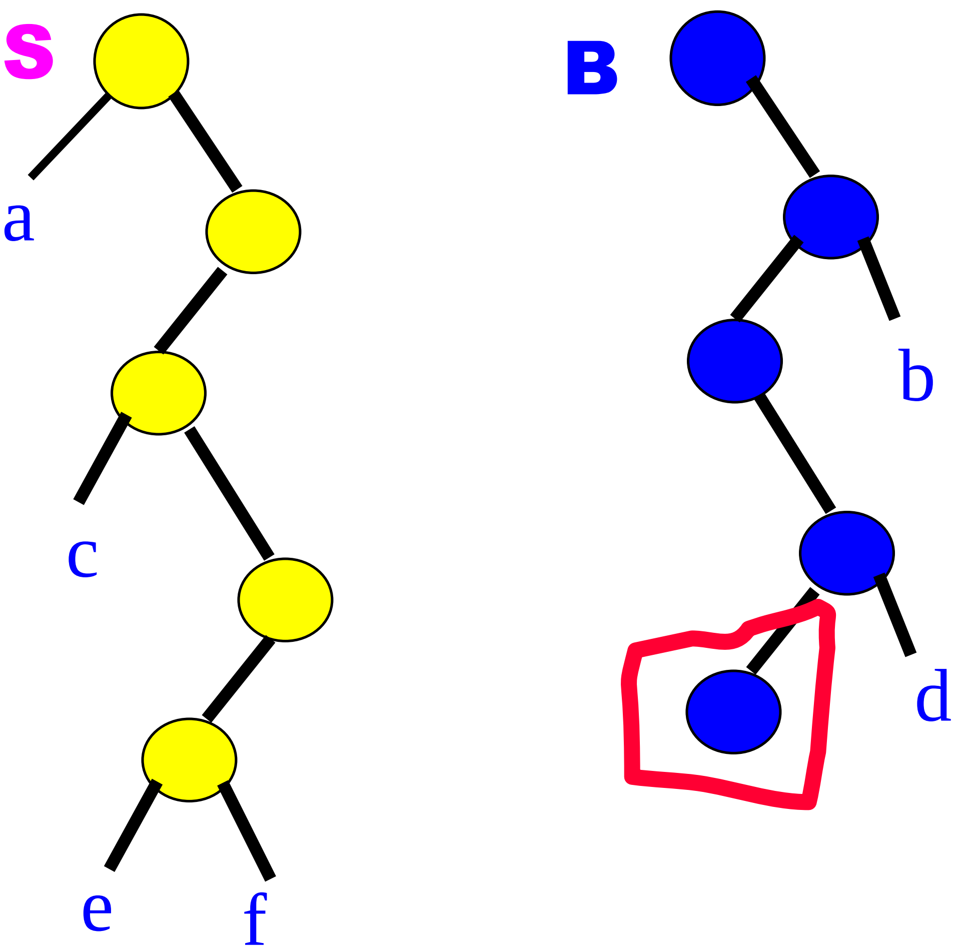
Backward Cleanup Pass Example

Now backup on B.



Backward Cleanup Pass Example

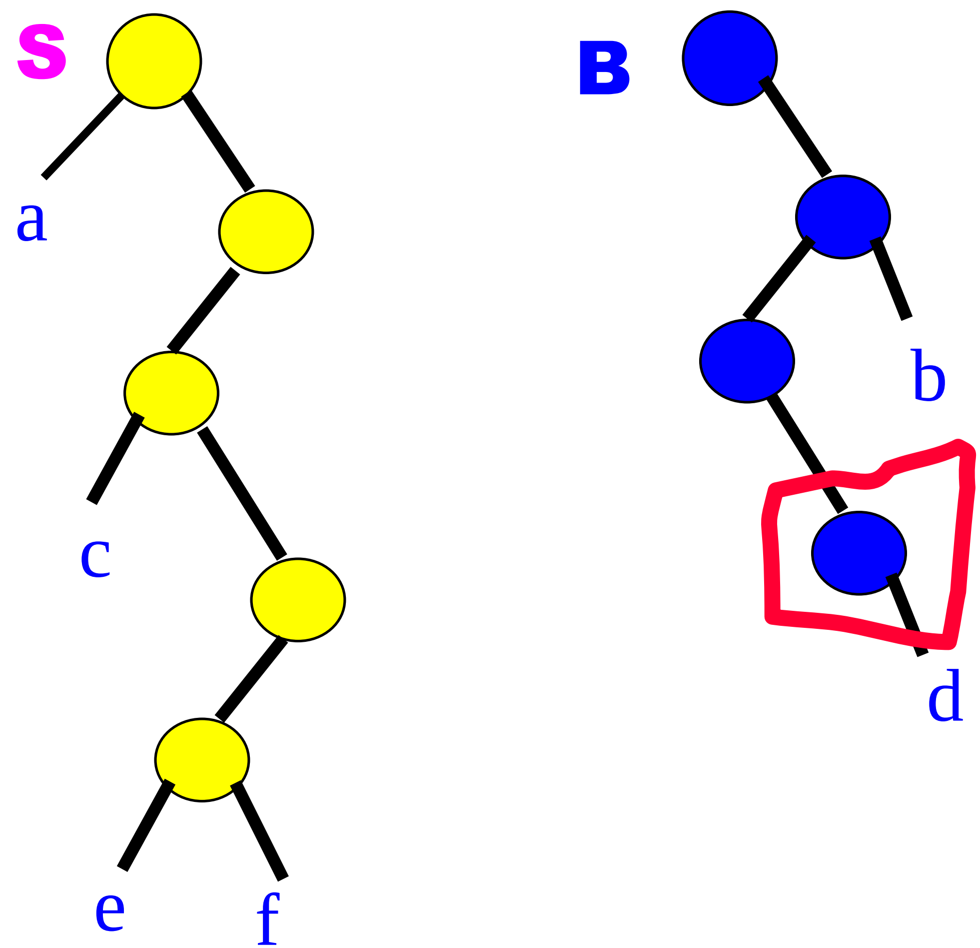
Now backup on B.



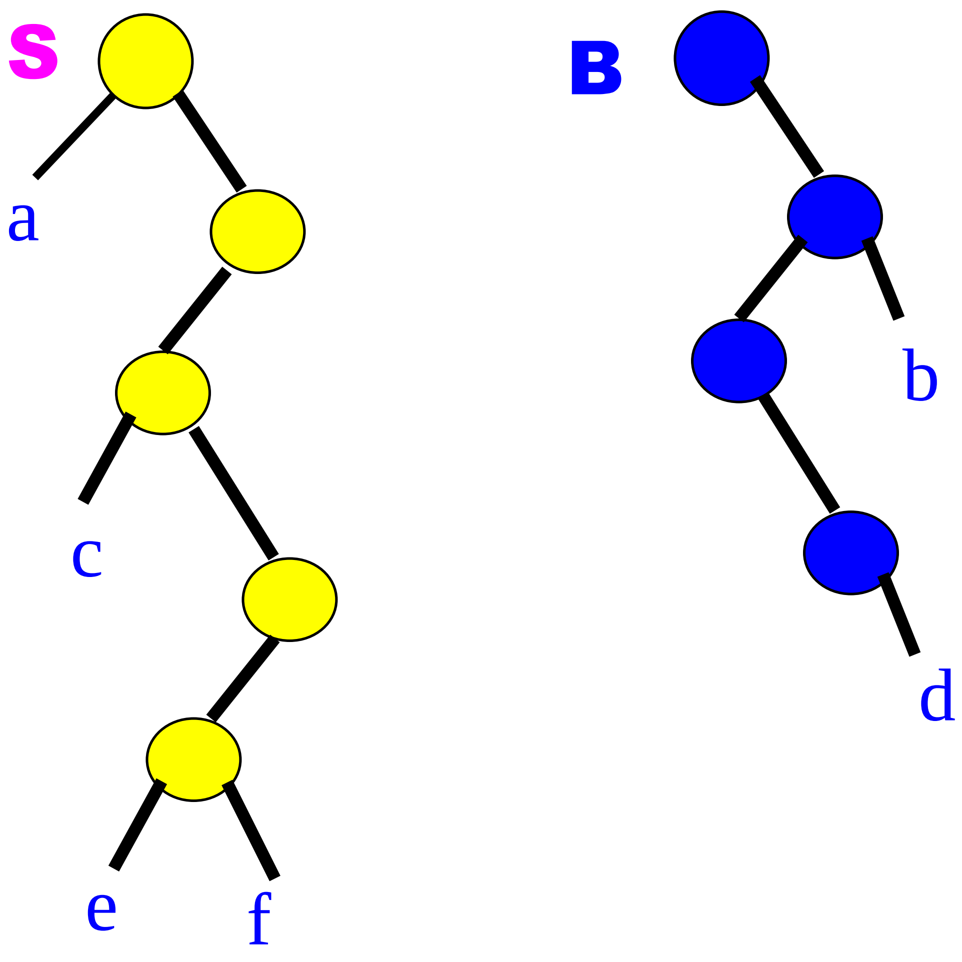
Backward Cleanup Pass Example

Now backup on B.

Assume root of d is a branch node.



Backward Cleanup Pass Example

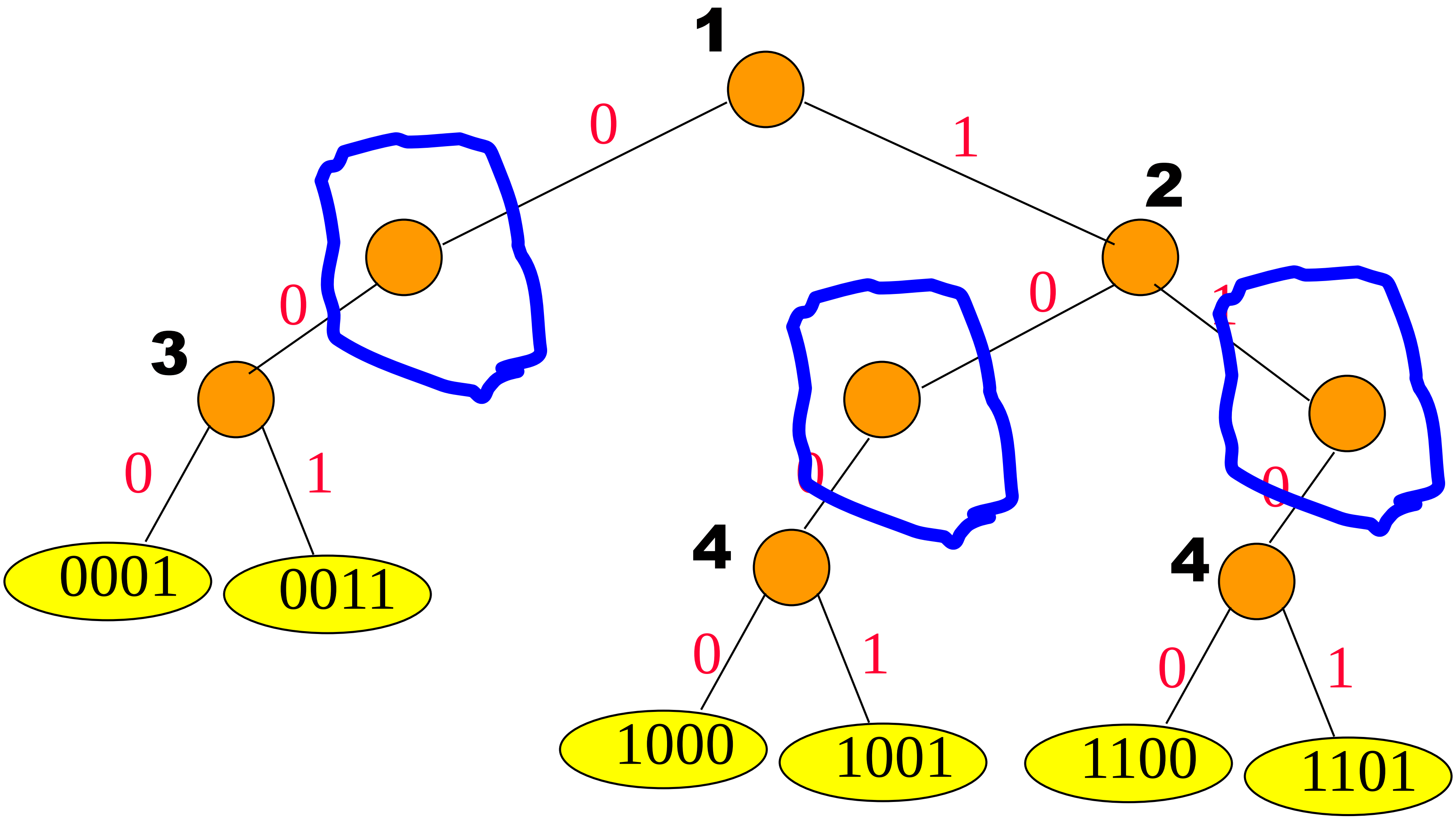


Complexity of split is O(height).

Compressed Binary Tries

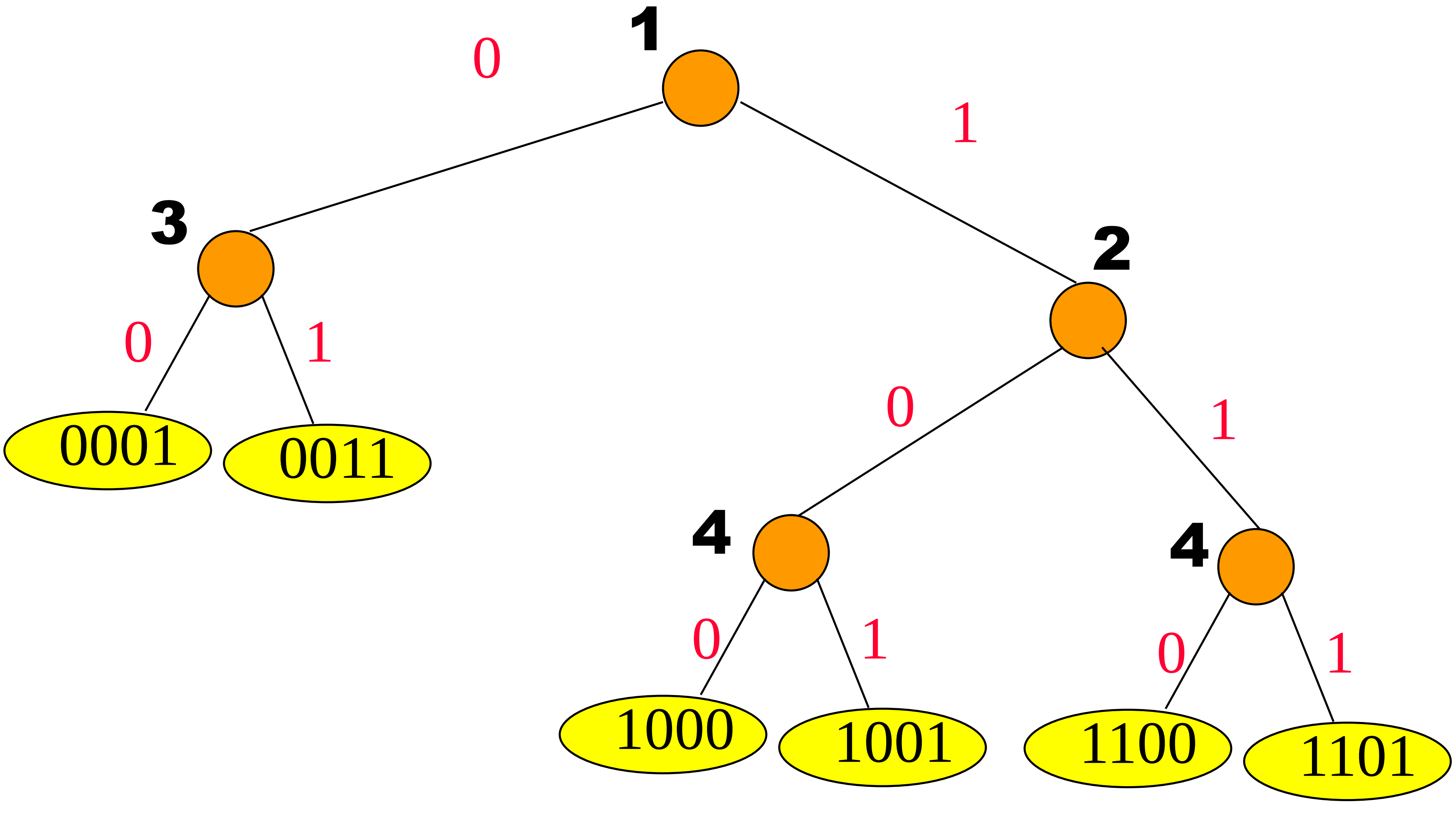
- No branch node whose degree is 1.
- Add a bit# field to each branch node.
- bit# tells you which bit of the key to use to decide whether to move to the left or right subtrie.

Binary Trie



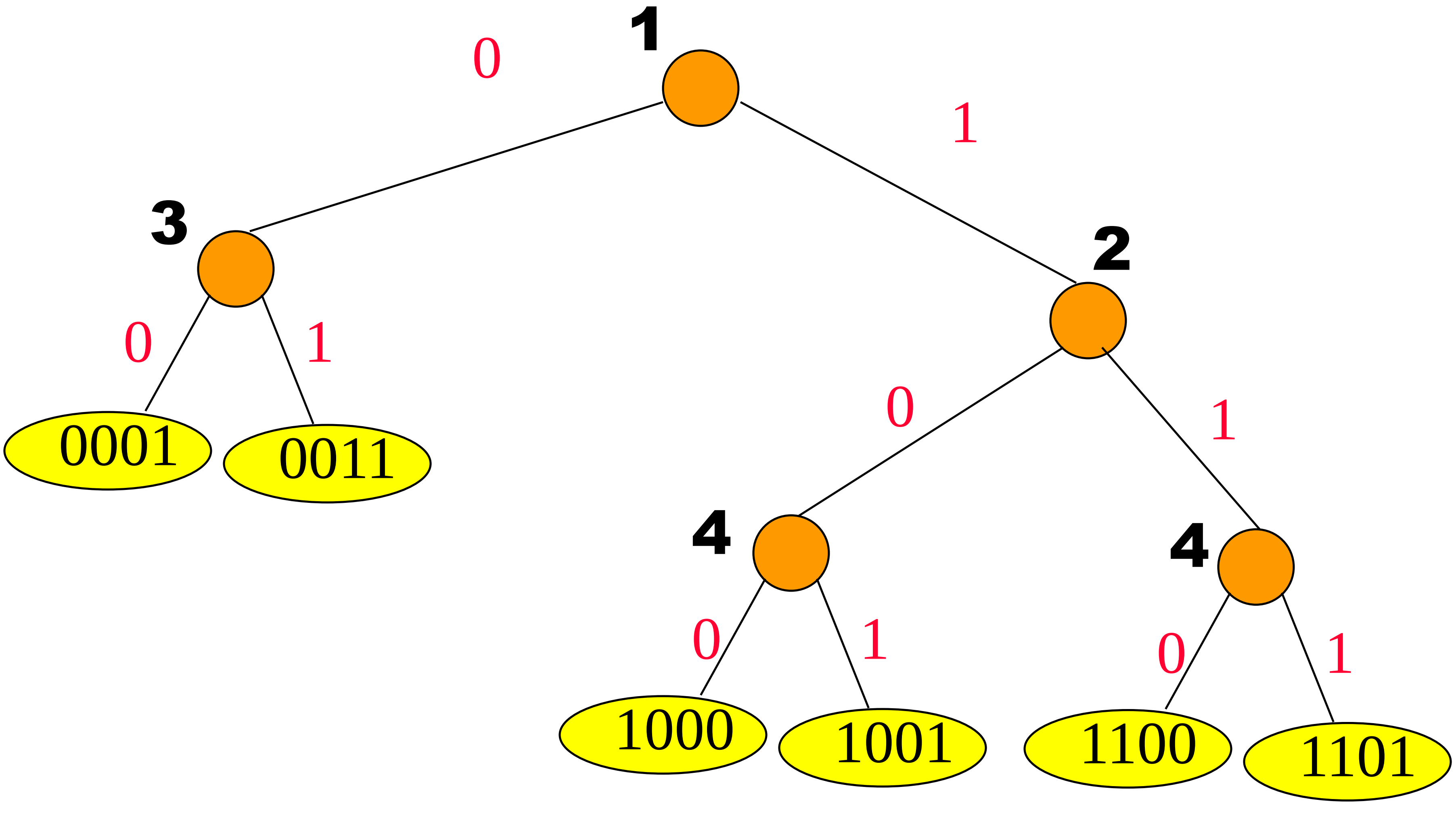
bit# field shown in black outside branch node.

Compressed Binary Trie



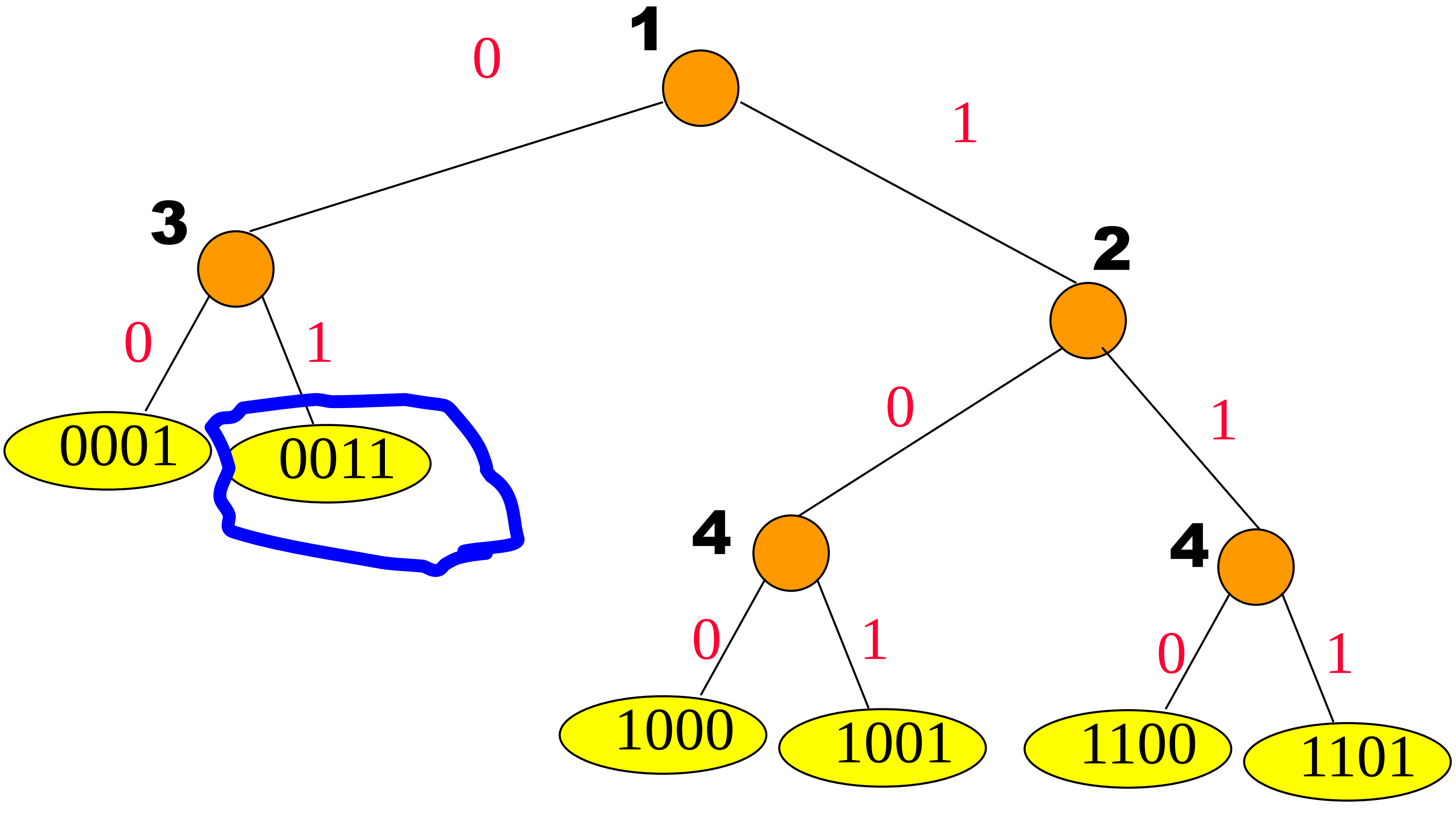
bit# field shown in black outside branch node.

Compressed Binary Trie



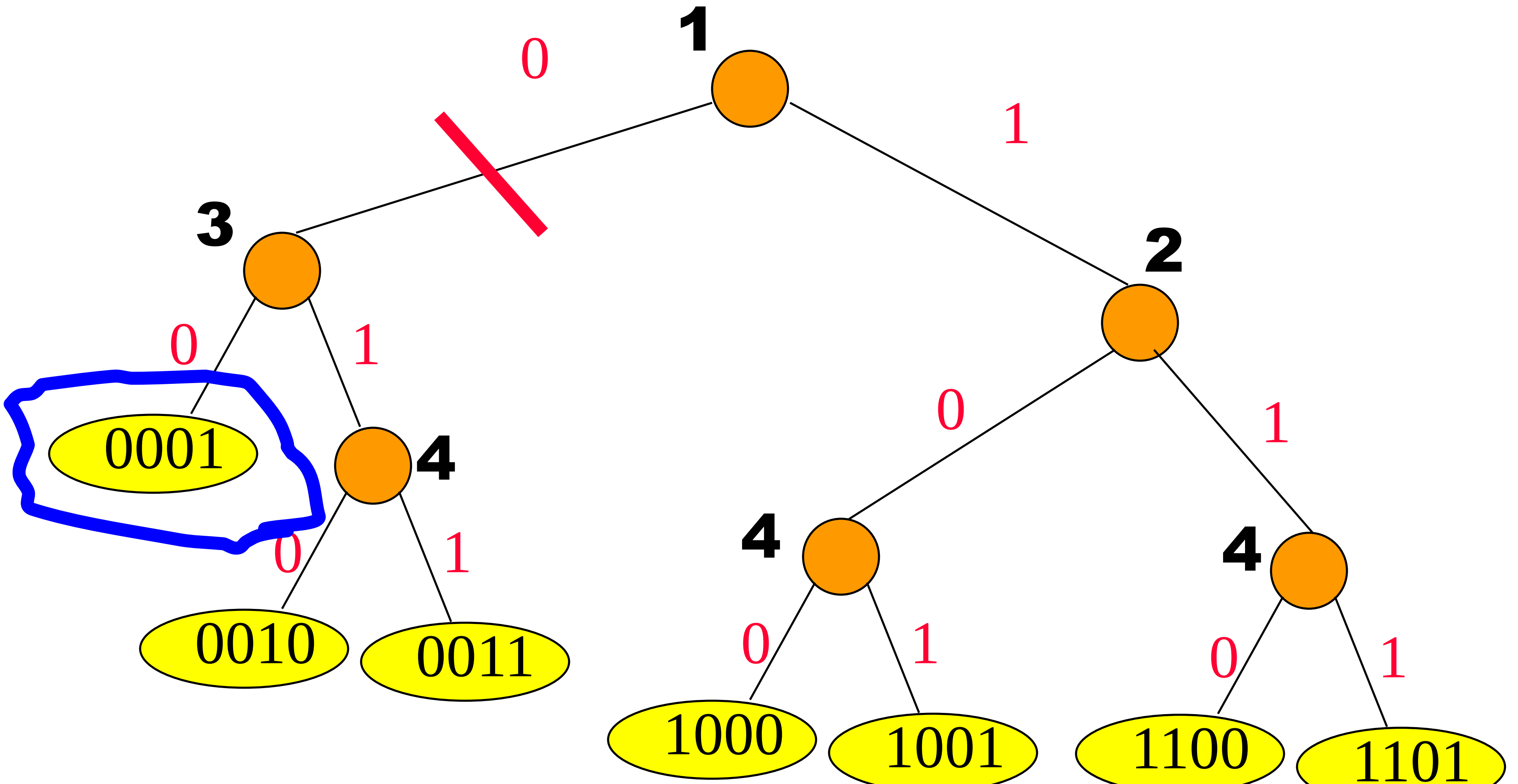
#branch nodes = n – 1.

Insert



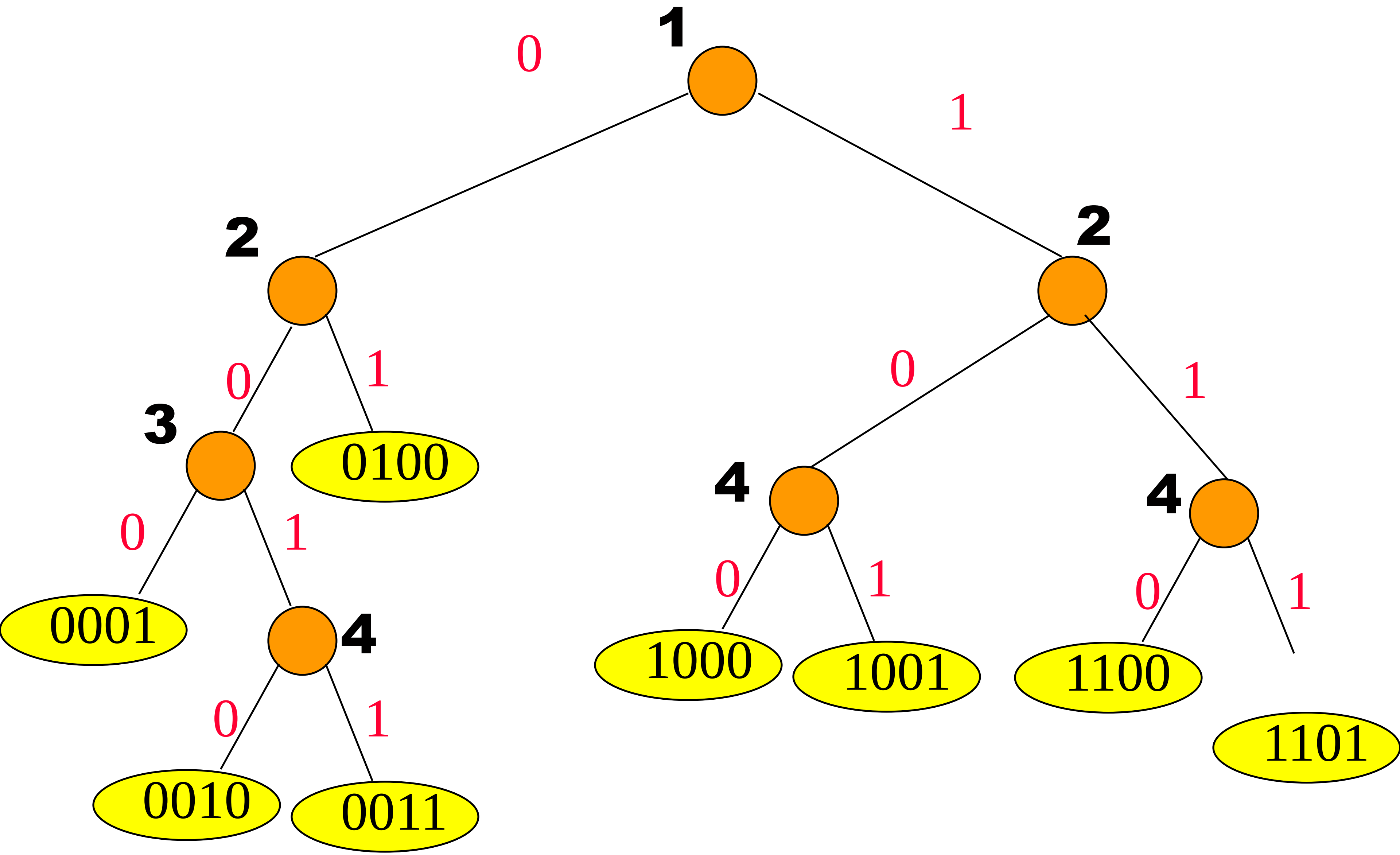
Insert 0010.

Insert



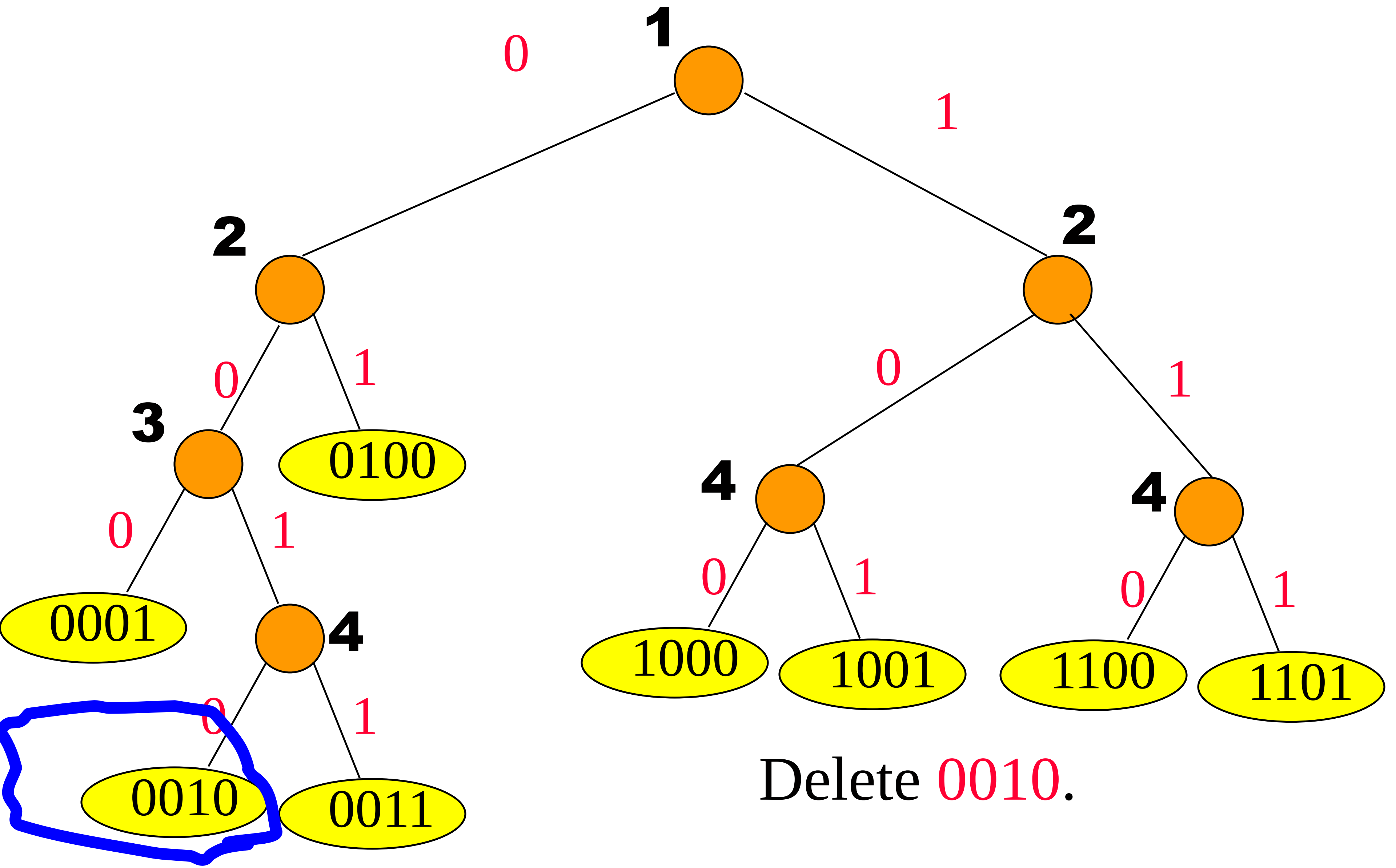
Insert 0100.

Insert



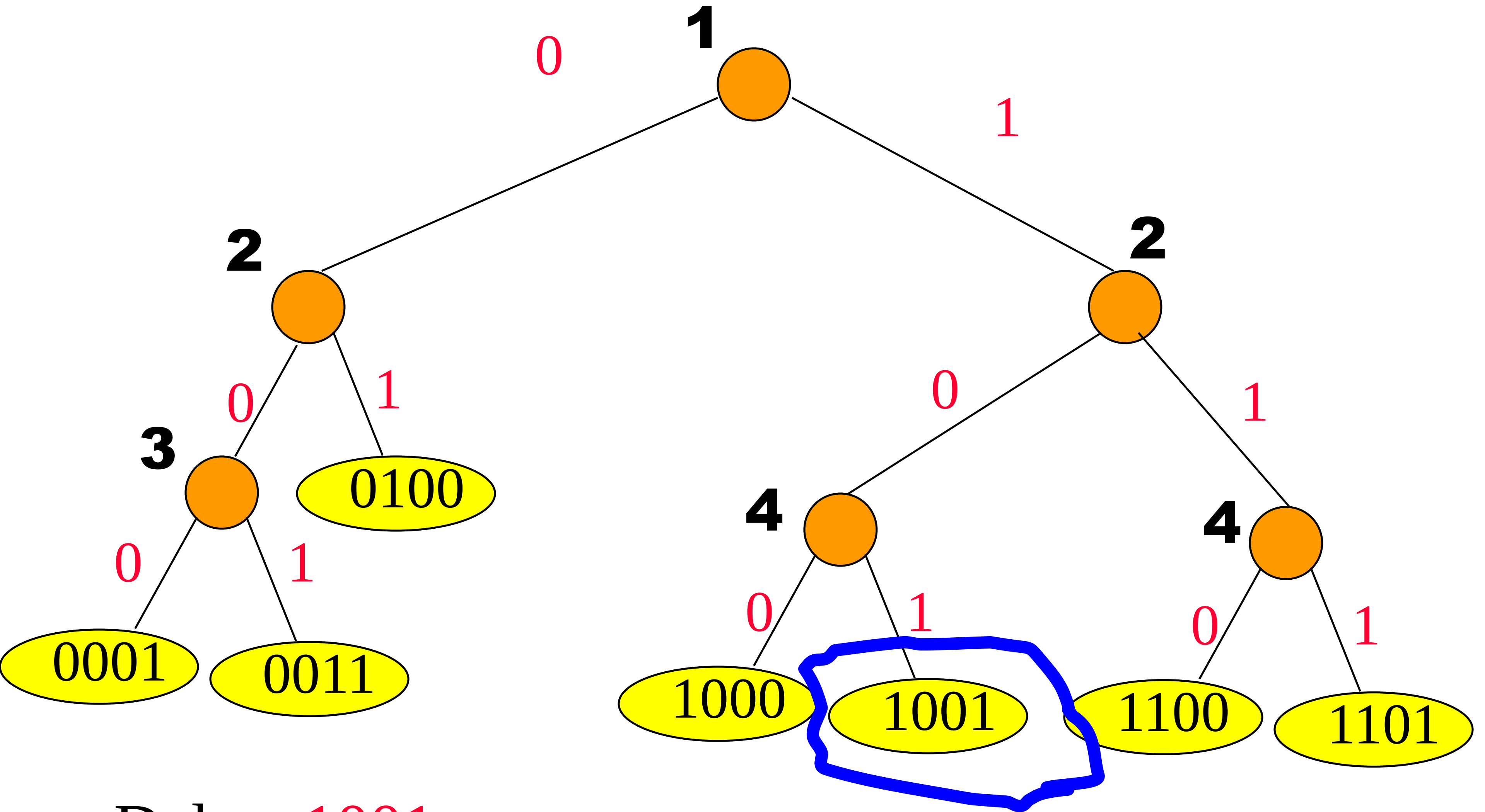
1
0
1
2
2
0
1
0
1
3
0100
0
1
4
4
0
1
0001
4
0
1
0
1
1000
1001
1100
0
1
0010
0011
1101

Delete



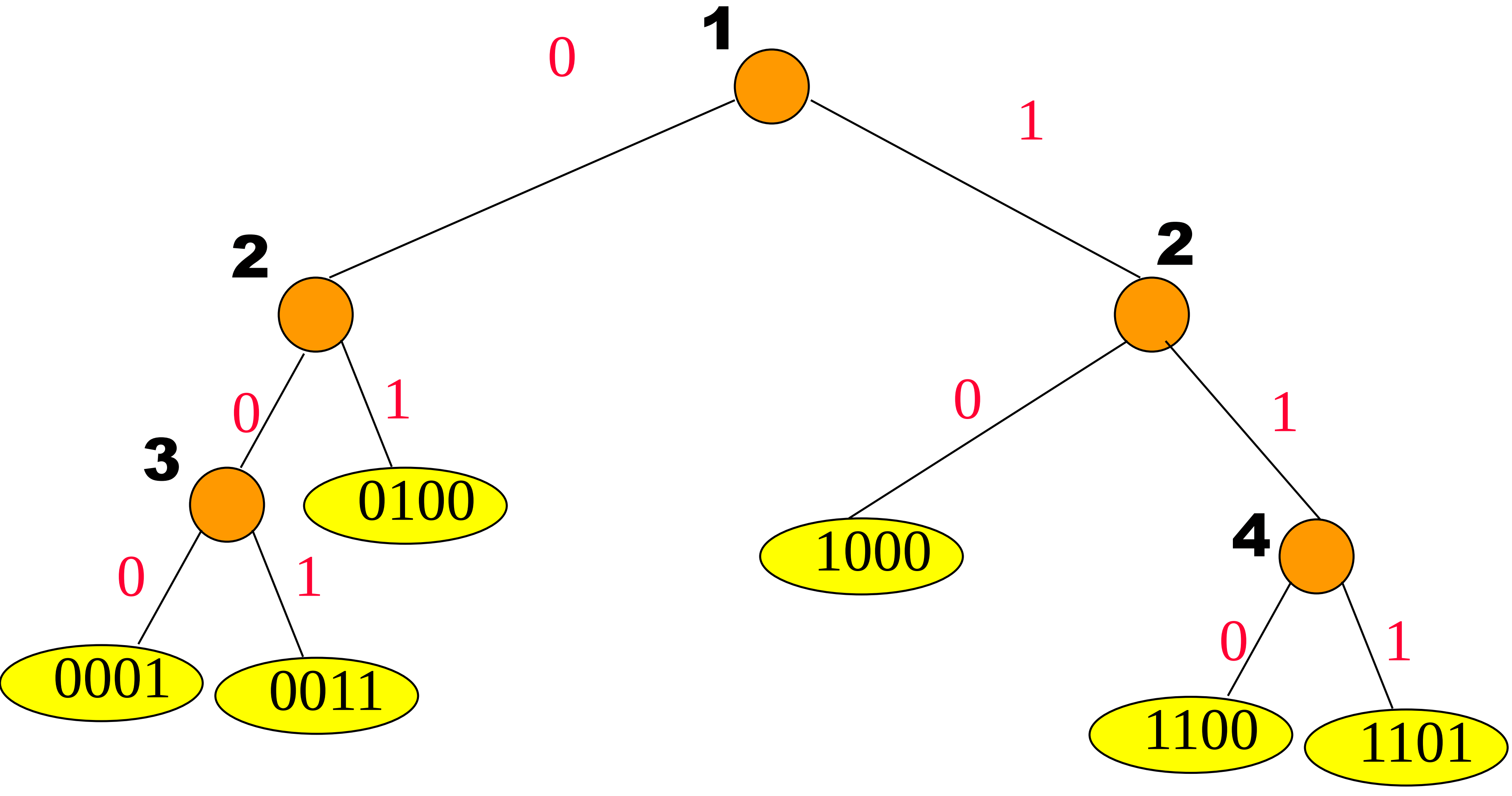
Delete 0010.

Delete



Delete 1001.

Delete



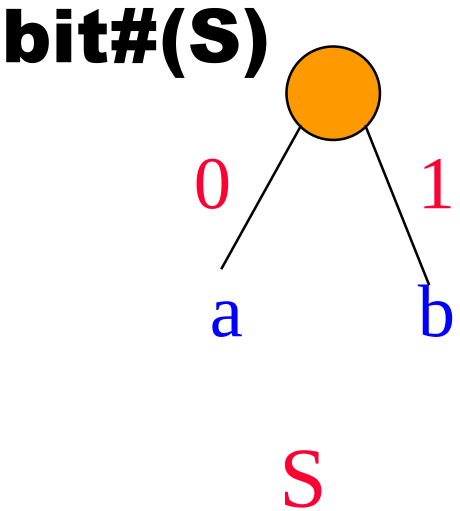
Split(k)

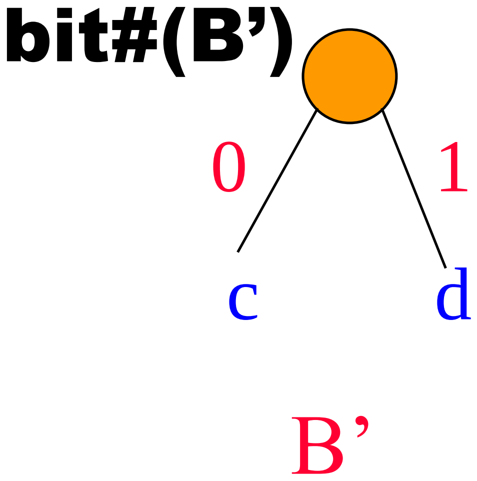
- Similar to splitting an uncompressed binary trie.

Join(S,m,B)

- Insert m into B to get B'.
- $|S| <= 1$ or $|B'| = 1$ handled as special cases as in the case of uncompressed tries.
- When $|S| > 1$ and $|B'| > 1$, let S_{max} be the largest key in S and let B'_{min} be the smallest key in B'.
- Let d be the first bit which is different in S_{max} and B'_{min}.

Cases To Consider

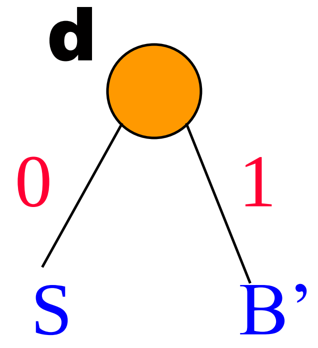




- $d < \min\{bit\#(S), bit\#(B')\}$
- $bit\#(S) = bit\#(B')$
- $bit\#(S) < bit\#(B')$
- $bit\#(S) > bit\#(B')$

$d < \min\{bit\#(S), bit\#(B')\}$

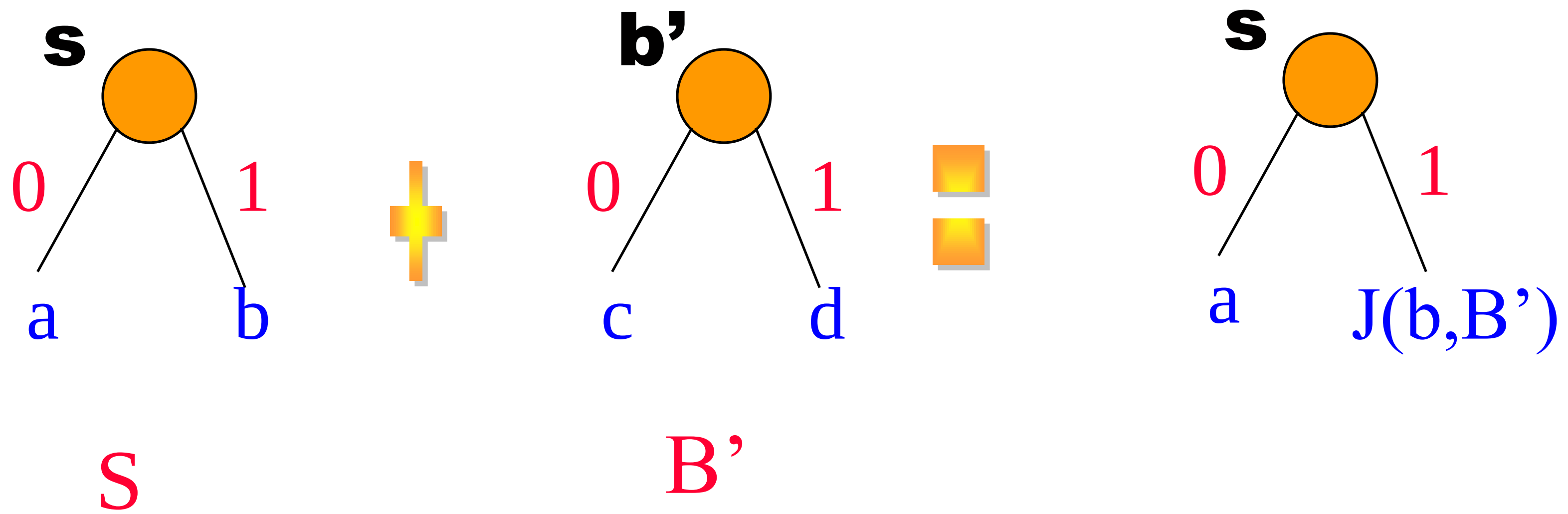
Bit d of S_{max} must be 0.



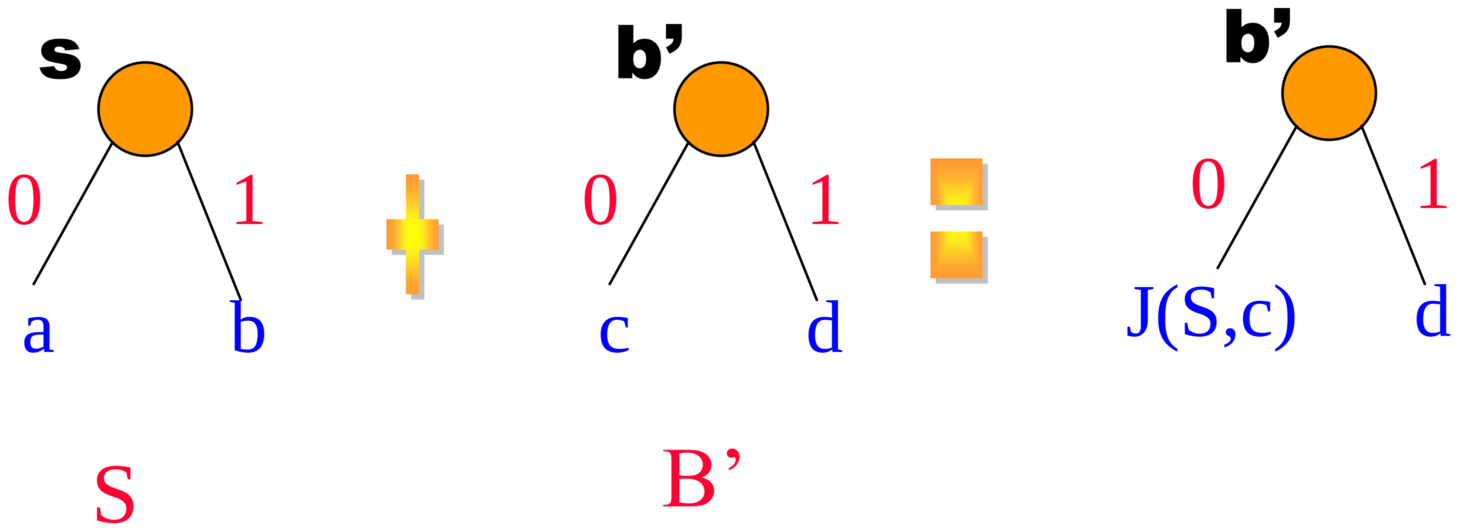
bit#(S) = bit#(B’)

- Not possible, because keys in b are larger than those in c.
- However, all keys in S are supposed to be smaller than those in B’.

bit#(S) < bit#(B’)



bit#(S) > bit#(B')



Complexity is O(max{height(S), height(B)}).

S_{max} and B'_{min} are found just once.