Multi-Way Search Trees

Manolis Koubarakis

Multi-Way Search Trees

- **Multi-way trees** are trees such that each internal node can have many children.
- Let us assume that the **entries** we store in a search tree are pairs of the form (k, x) where k is the **key** and x the **value** associated with the key.
- **Example**: Assume we store information about students. The key can be the student ID while the value can be information such as name, year of study etc.

Definitions

- A tree is **ordered** if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, the second, third and so on.
- Let v be a node of an ordered tree. We say that v is a $\boldsymbol{d}$**-node** if v has d children.

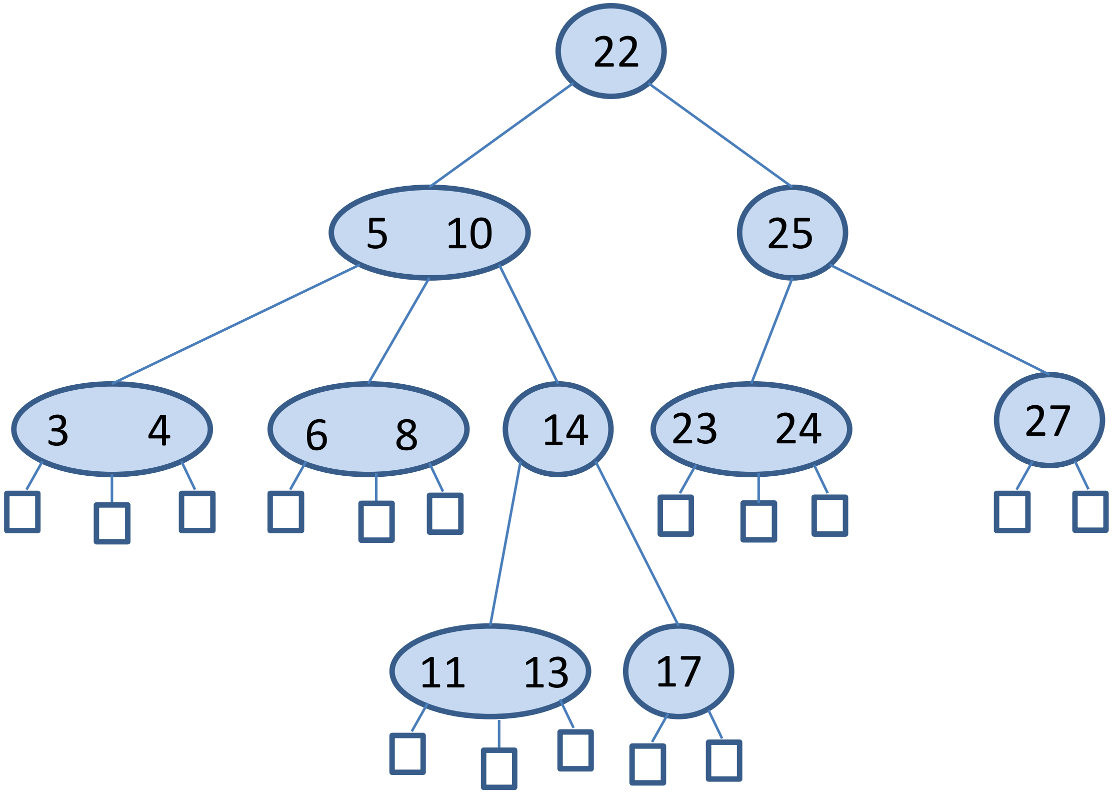
Definitions (cont'd)

- A **multi-way search tree** is an ordered tree T that has the following properties:
    - Each internal node of T has at least 2 children. That is, each internal node is a d-node such that $d \geq 2$.
    - Each internal d-node of T with children $v_1, \cdots, v_d$ stores an ordered set of $d-1$ key-value entries $(k_1, x_1), \cdots, (k_{d-1}, x_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
    - Let us conveniently define $k_0 = -\infty$ and $k_d = +\infty$. For each entry (k, x) stored at a node in the subtree of v rooted at $v_i, i = 1, \cdots, d$, we have that $k_{i-1} \leq k \leq k_i$.

Definitions (cont’d)

- By the above definition, the external nodes of a multi-way search tree do not store any entries and are “dummy” nodes (i.e., our trees are extended trees).
- When $m \geq 2$ is the maximum number of children that a node is allowed to have, then we have an $\boldsymbol{m}$**-way search tree.**
- A **binary search tree** is a special case of a multi-way search tree, where each internal node stores one entry and has two children (i.e., $m = 2$).

Example Multi-Way Search Tree ($m = 3$)



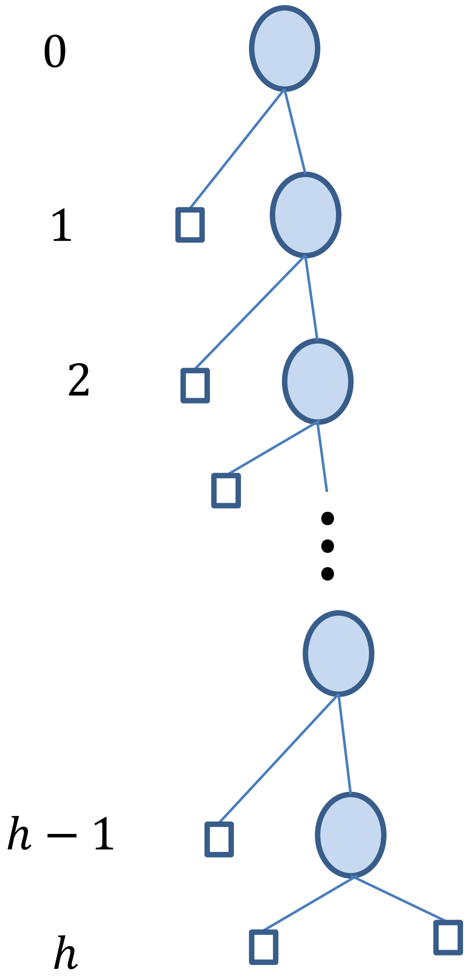
Proposition

- Let T be an m-way search tree with height h, n entries and n_E external nodes. Then, the following inequalities hold:
  1. $h \leq n \leq m^h - 1$
  2. $\log_m(n + 1) \leq h \leq n$
  3. $n_E = n + 1$
- Proof?

Proof

- We will prove (1) first.
- The lower bound can be seen by considering an m-way search tree like the one given on the next slide where we have one internal node and one entry in each node for levels $0, 1, 2, \cdots, h-1$ and level h contains only external nodes.

Proof (cont’d)



Proof (cont'd)

- For the upper bound, consider an m-way search tree of height h where each internal node in the levels 0 to $h-1$ has exactly m children (the external nodes are at level h).
- These internal nodes are $\sum_{i=0}^{h-1} m^i = \frac{m^h-1}{m-1}$ in total.
- Since each of these nodes has $m-1$ entries, the total number of entries in the internal nodes is $m^h - 1$.

Proof (cont'd)

- To prove the lower bound of (2), rewrite (1) and take logarithms in base m. The upper bound in (2) is the same as the lower bound in (1).

Proof (cont'd)

- We will prove (3) by induction on the height h of the tree.

- **Base case:** Let $h = 1.$ Then there is a single root node with n entries and $n + 1$ external nodes and the proposition holds.

Proof (cont'd)

- **Inductive step:** Let $h > 1$. If the root stores m entries then it has $m + 1$ subtrees for which the inductive hypothesis holds. Therefore, each such subtree i has p_i entries and $p_i + 1$ external nodes.

Therefore the tree has $A = m + (\sum_{i=1}^{m+1} p_i)$ entries and

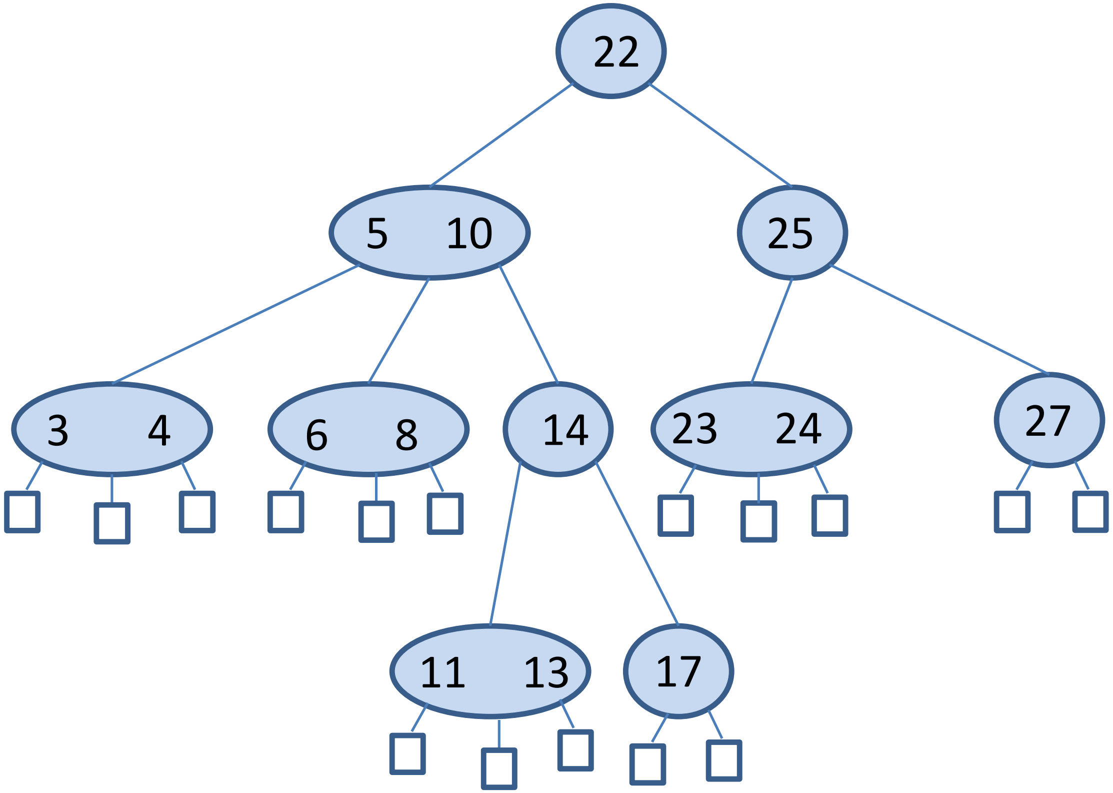
$$B = \sum_{i=1}^{m+1}(p_i+1) = m + 1 + (\sum_{i=1}^{m+1} p_i) = A + 1$$

external nodes.

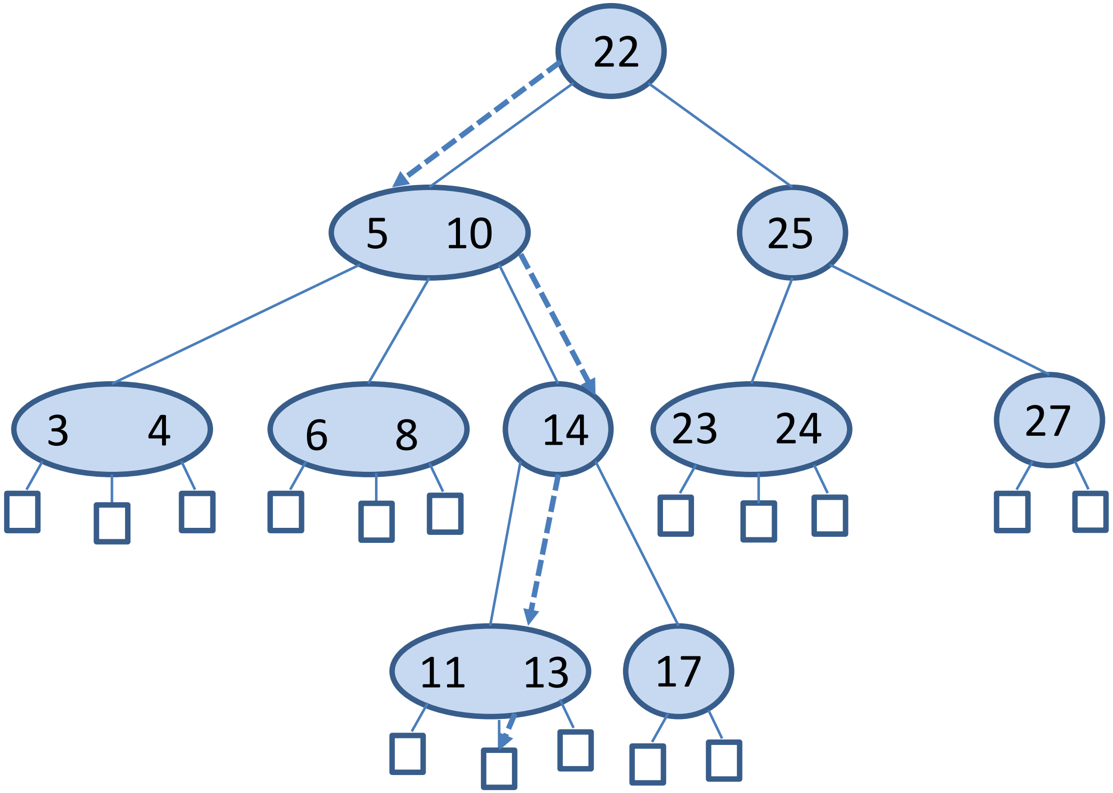
Searching in a Multi-Way Search Tree

- Let T be a multi-way search tree and k be a key.
- The algorithm for searching for an entry with key k is simple.
- We trace a path in T starting at the root.
- When we are at a d-node v during the search, we compare the key k with the keys $k_1, \cdots, k_{d-1}$ stored at v.
- If $k = k_i$, for some i, the search is successfully completed. Otherwise, we continue the search in the child v_i of v such that $k_{i-1} < k < k_i$.
- If we reach an external node, then we know that there is no entry with key k in T.

Example Multi-Way Search Tree

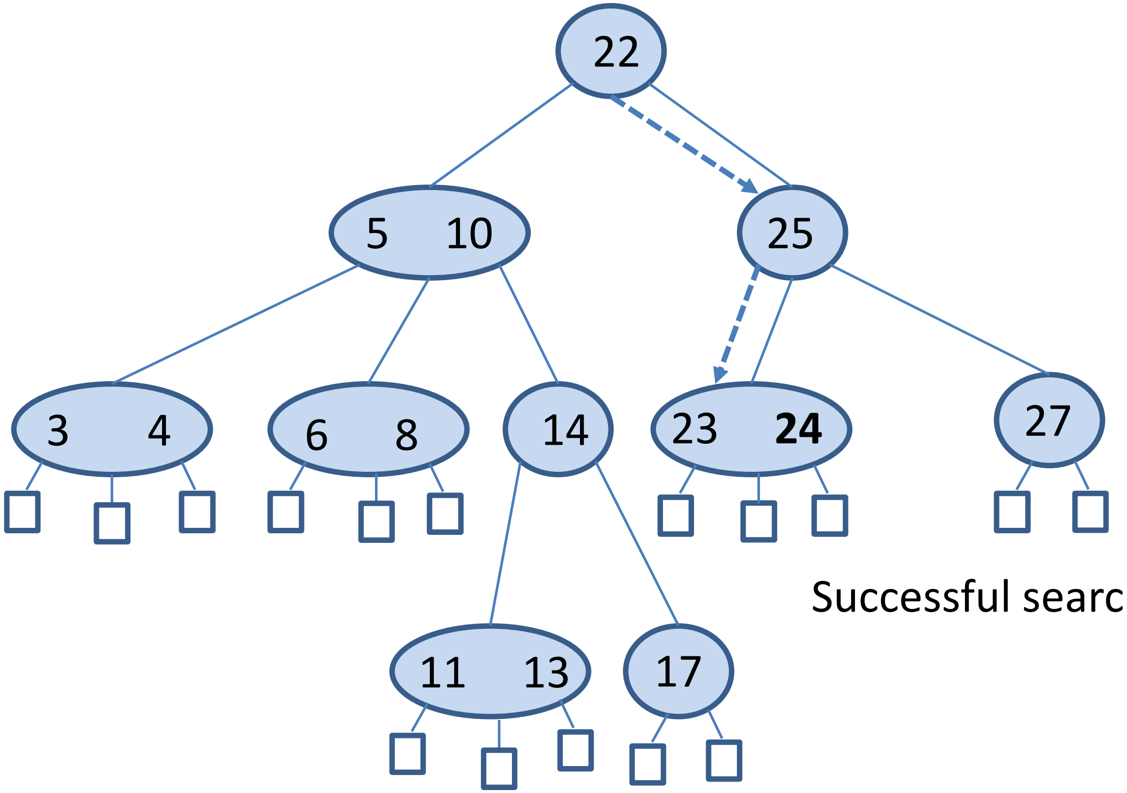


Search for Key 12



Unsuccessful search

Search for Key 24

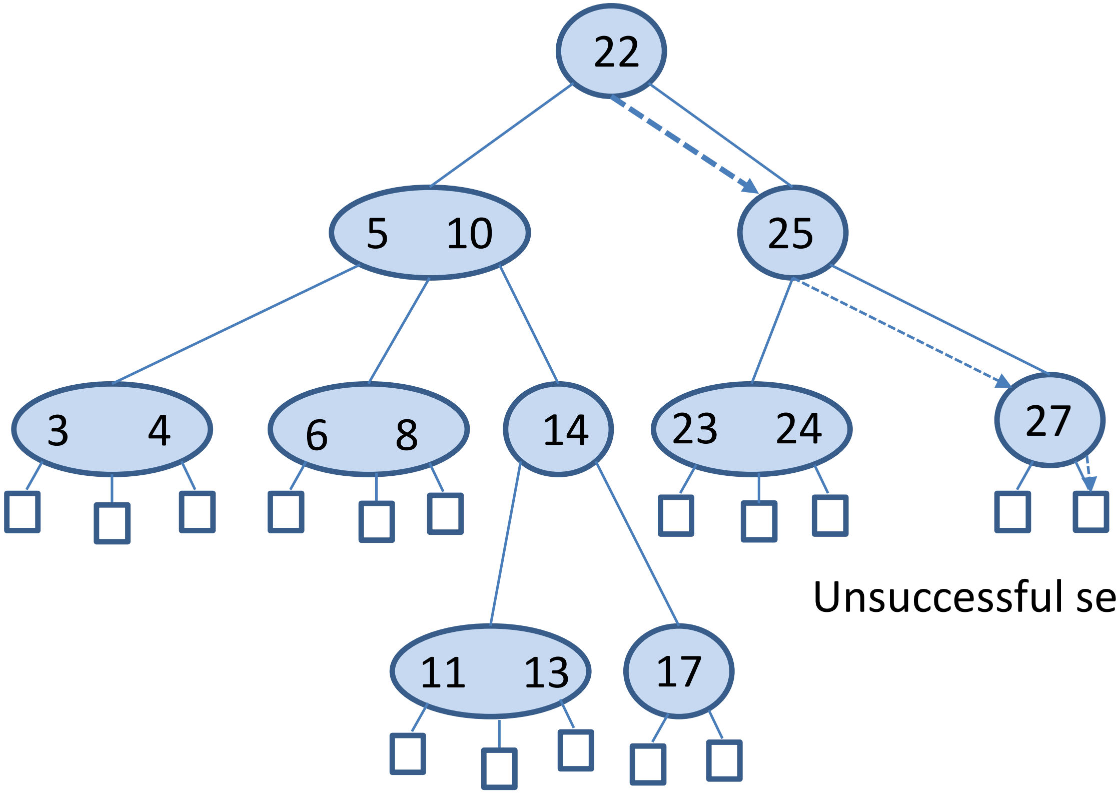


Successful search

Insertion in a Multi-Way Search Tree

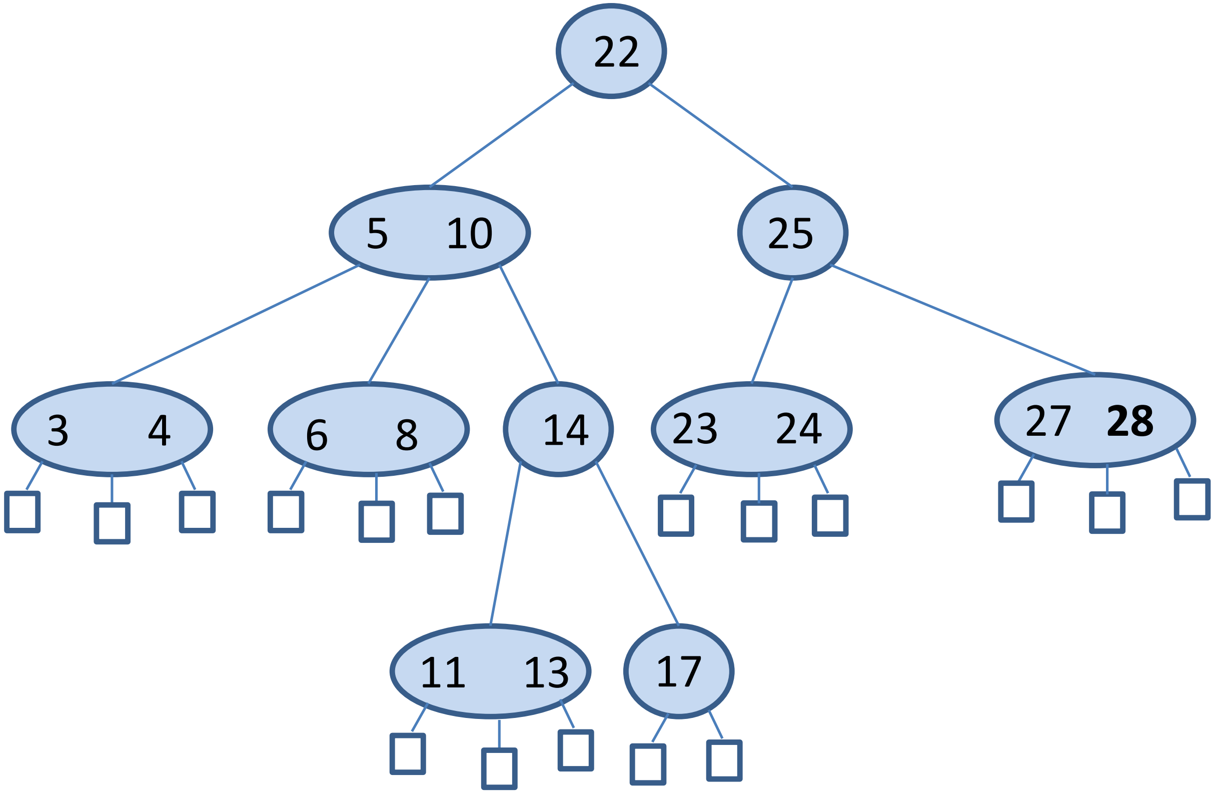
- If we want to insert a new pair (k, x) into a multi-way search tree, then we start by searching for this entry.
- If we find the entry, then we do not need to reinsert it.
- If we end up in an external node, then the entry is not in the tree. In this case, we return to the parent v of the external node and attempt to insert the key there.
- If v has space for one more key then we insert the entry there. If not, we create a new node, we insert the entry in this node and make this node a child of v in the appropriate position.

Insert Key 28 ($m = 3$)

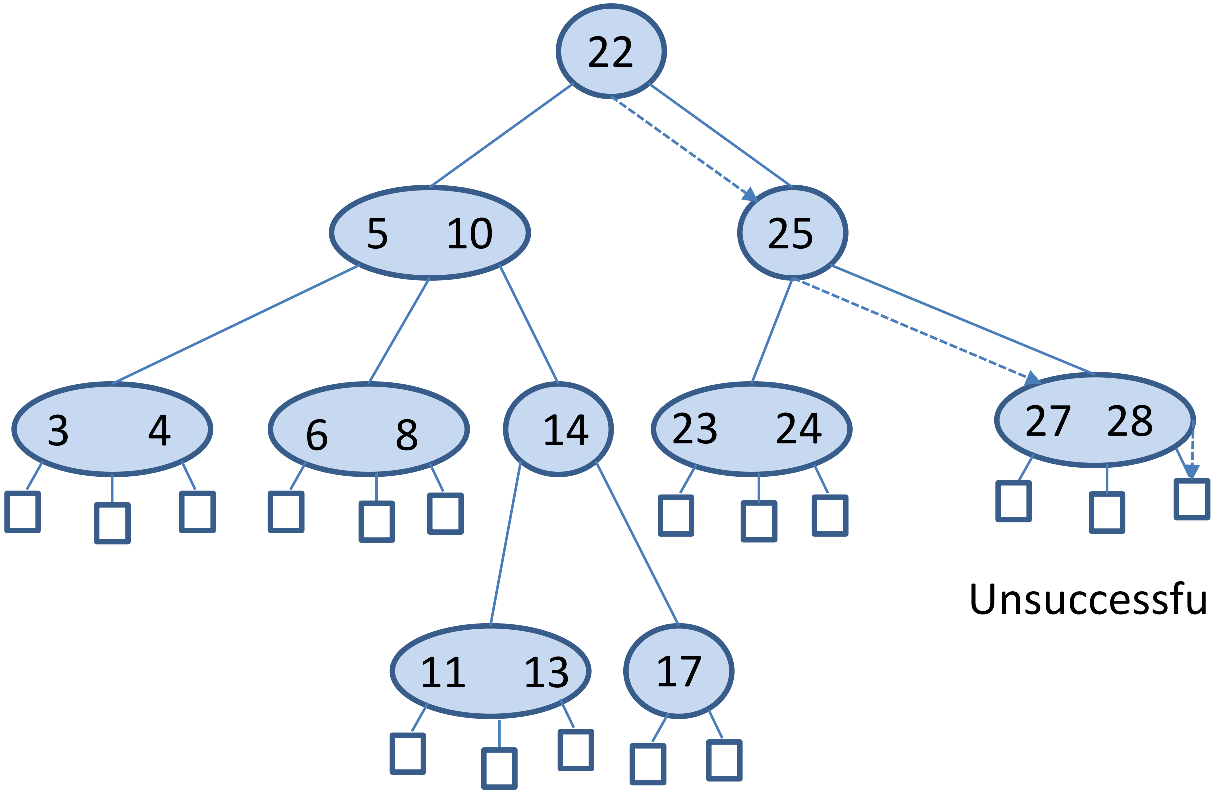


Unsuccessful search

Key 28 Inserted

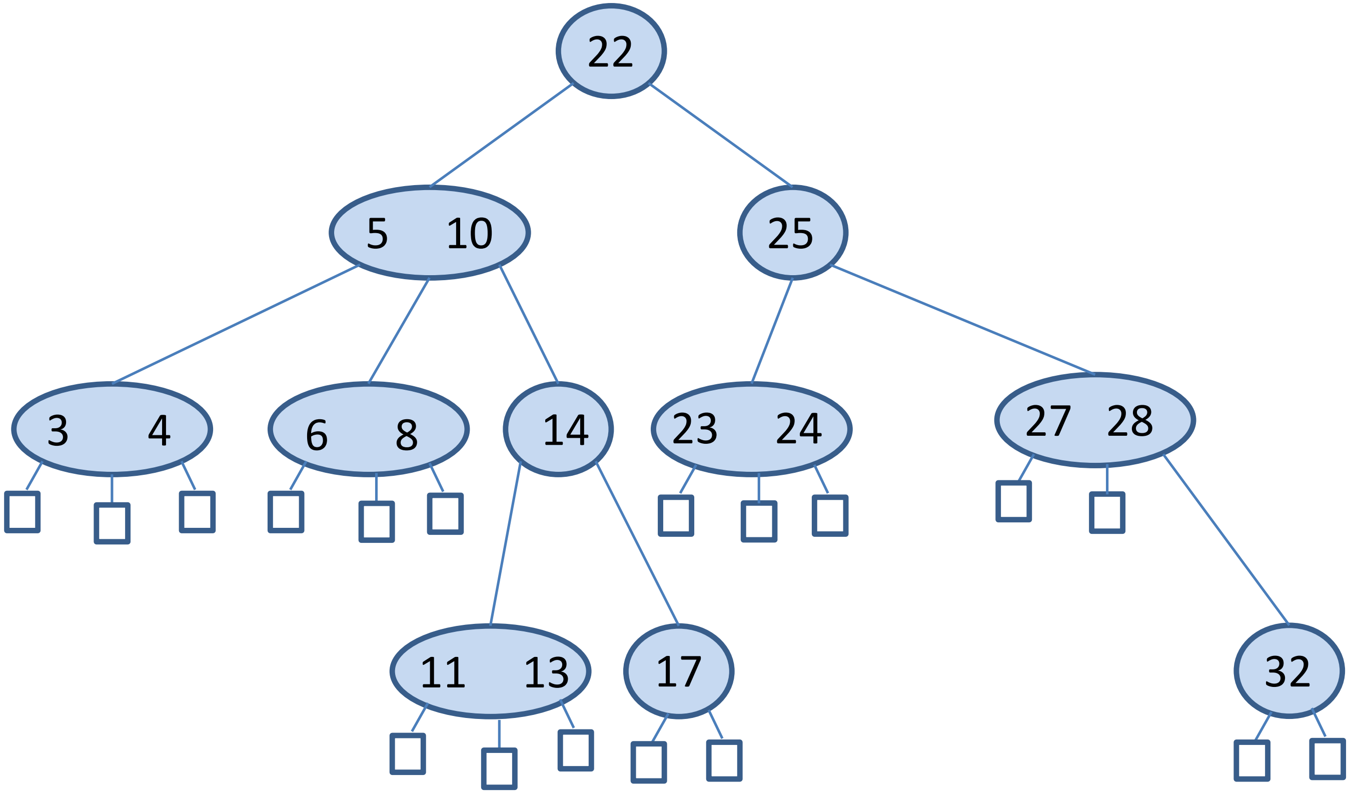


Insert Key 32

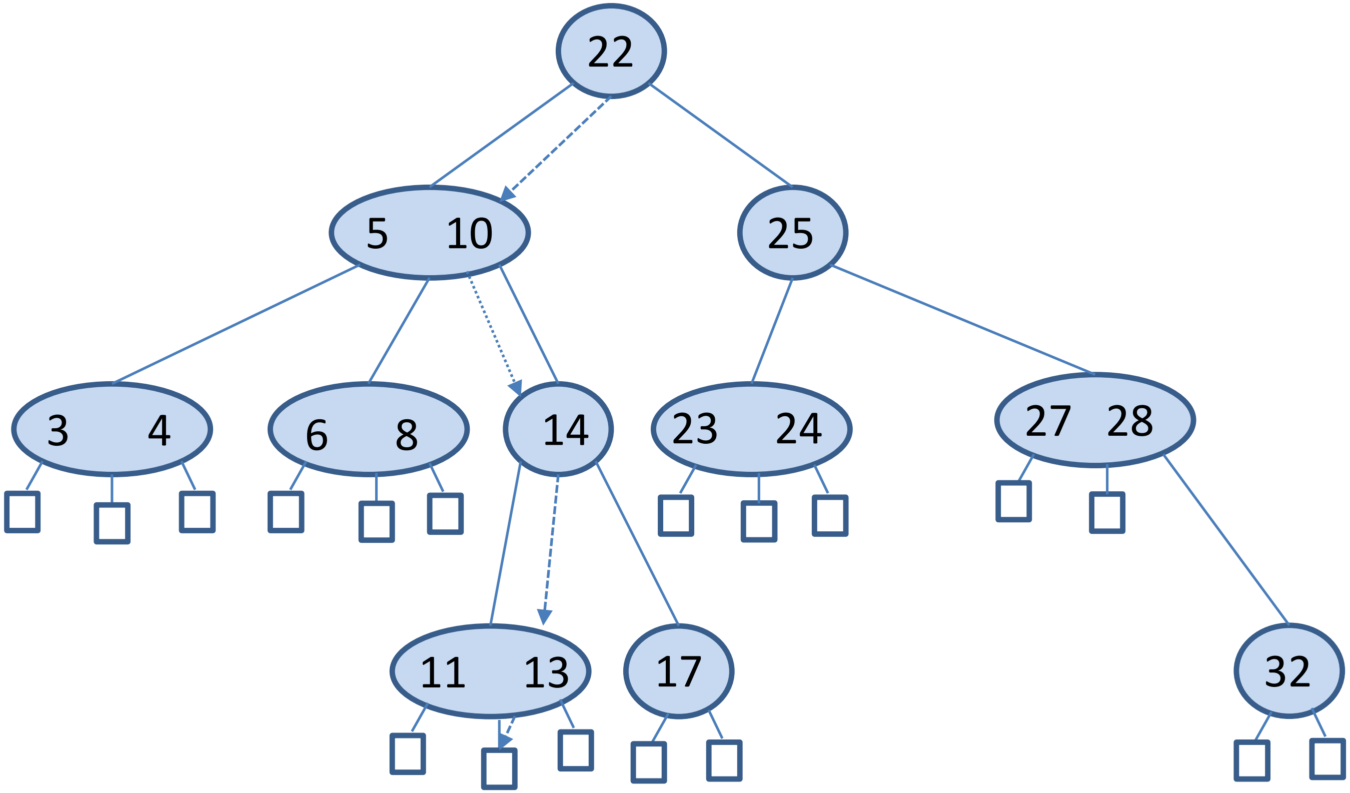


Unsuccessful search

Key 32 Inserted

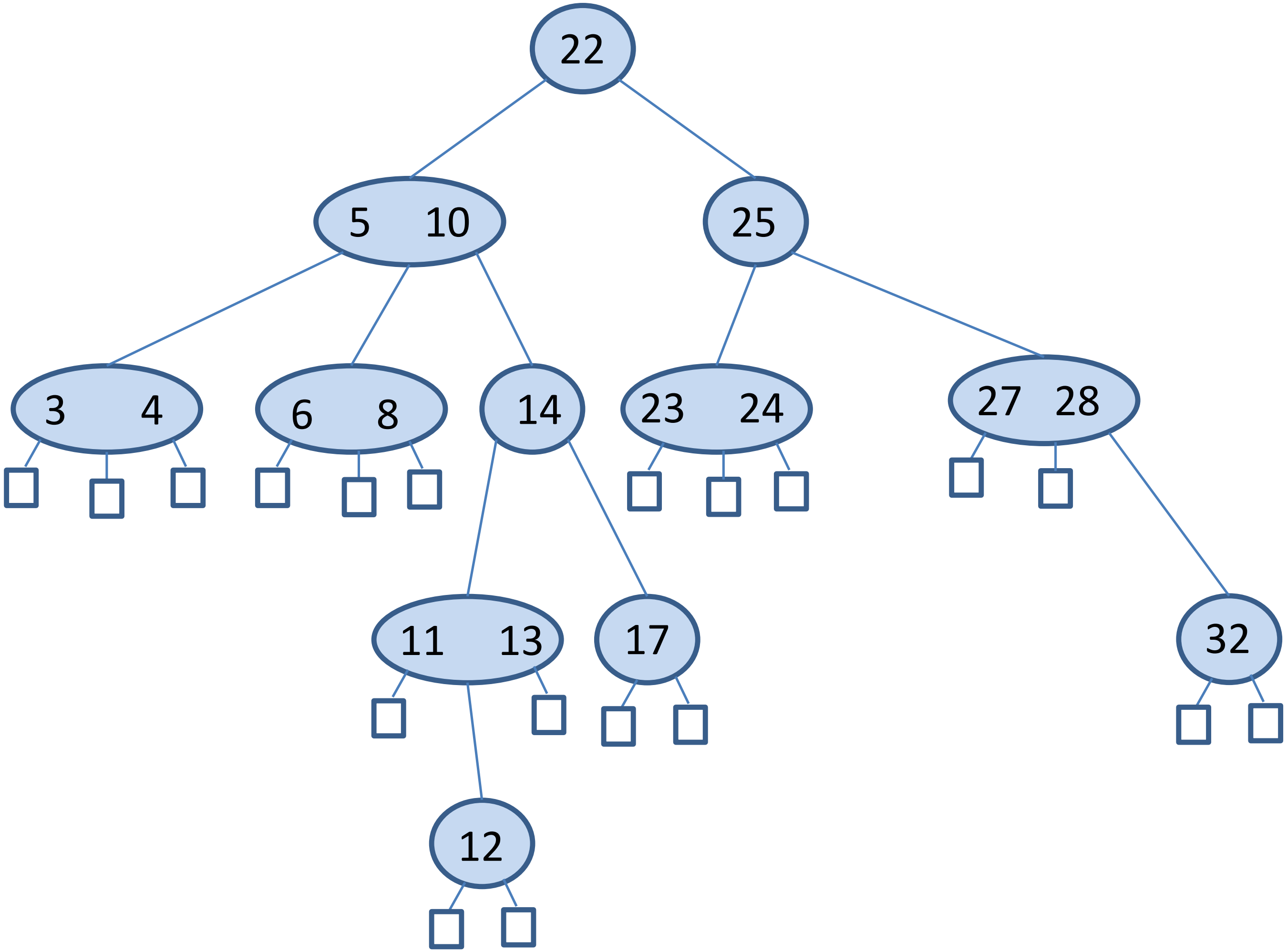


Insert Key 12



Unsuccessful Search

Key 12 Inserted



Deletion from a Multi-Way Search Tree

- The algorithm for deletion from a multi-way search tree is left as an exercise.

Complexity of Operations

- Let us consider the time to search a m-way search tree for a given key.
- The time spent at a d-node depends on the implementation of the node. If we use a sorted array then, using binary search, we can search a node in $O(\log d)$ time.
- Thus the time for a search operation in the tree is $O(h \log m)$.
- The complexity of insertion and deletion is also $O(h \log m)$.

Efficiency Considerations

- We know that **maintaining perfect balance** in binary search trees yields shortest average search paths, but the attempts to maintain perfect balance when we insert or delete nodes can incur costly rebalancing in which every node of the tree needs to be rearranged.
- AVL trees showed us one way to solve this problem by abandoning the goal of perfect balance and adopt the goal of keeping the trees “almost balanced”.

Efficiency Considerations (cont’d)

- Multi-way search trees give us another way to solve this problem.
- The primary efficiency goal for a multi-way search tree is to **keep the height as small as possible but permit the number of keys at each node to vary.**
- We want the height of the tree h to be a logarithmic function of n, the total number of entries stored in the tree.
- A search tree with logarithmic height is called a **balanced search tree**.

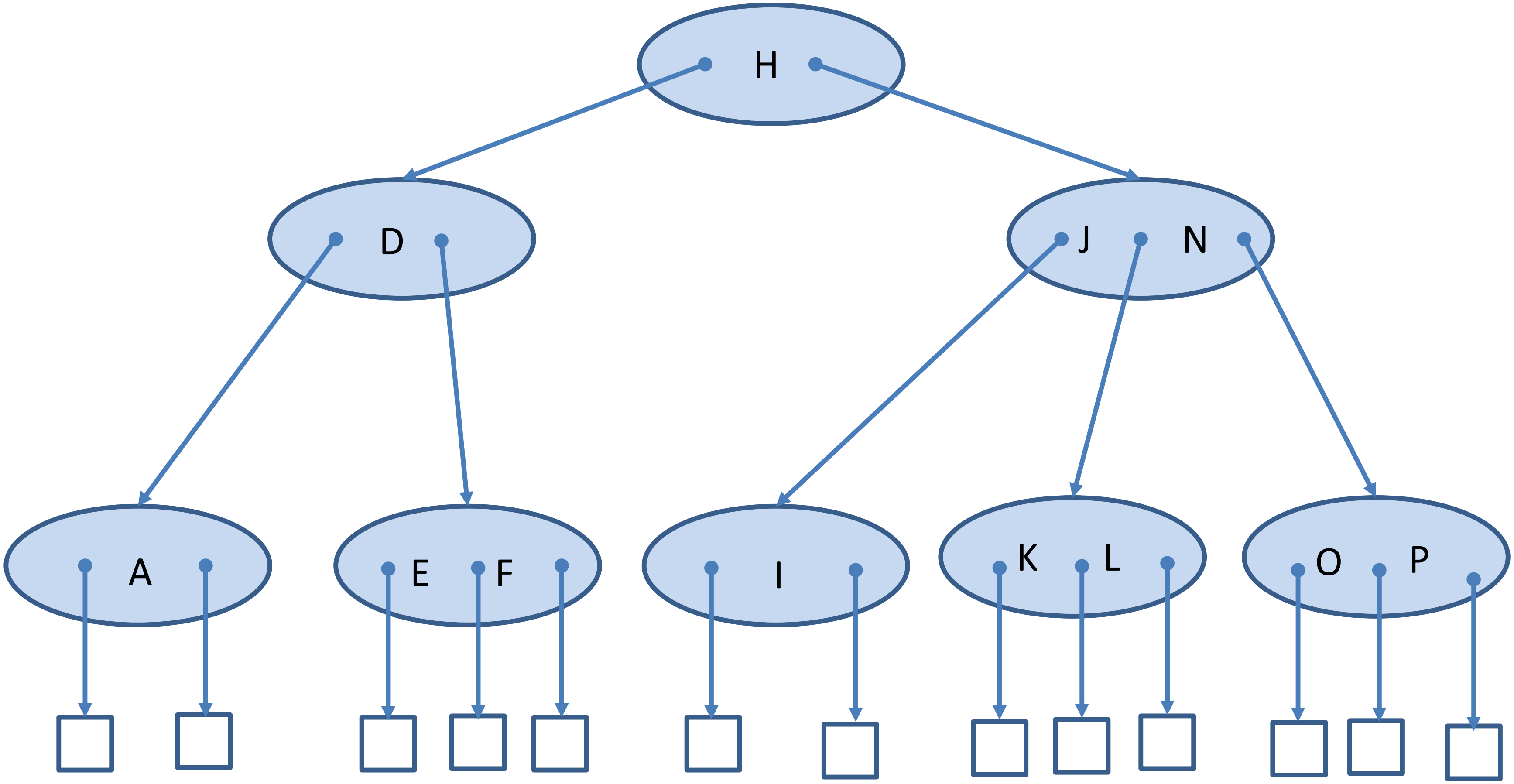
Balanced Multi-way Search Trees

- We will study two kinds of balanced multi-way search trees:
  - **2-3 trees**
  - **2-3-4 trees** or **(2,4) trees**.

2-3 Trees

- A **2-3 tree** is a multi-way search tree which has the following properties:
- **Size property**: Each internal node contains one or two entries, and has either two or three children.
- **Depth property**: All leaves of the tree are empty trees that have the same depth (lie on a single bottom level).

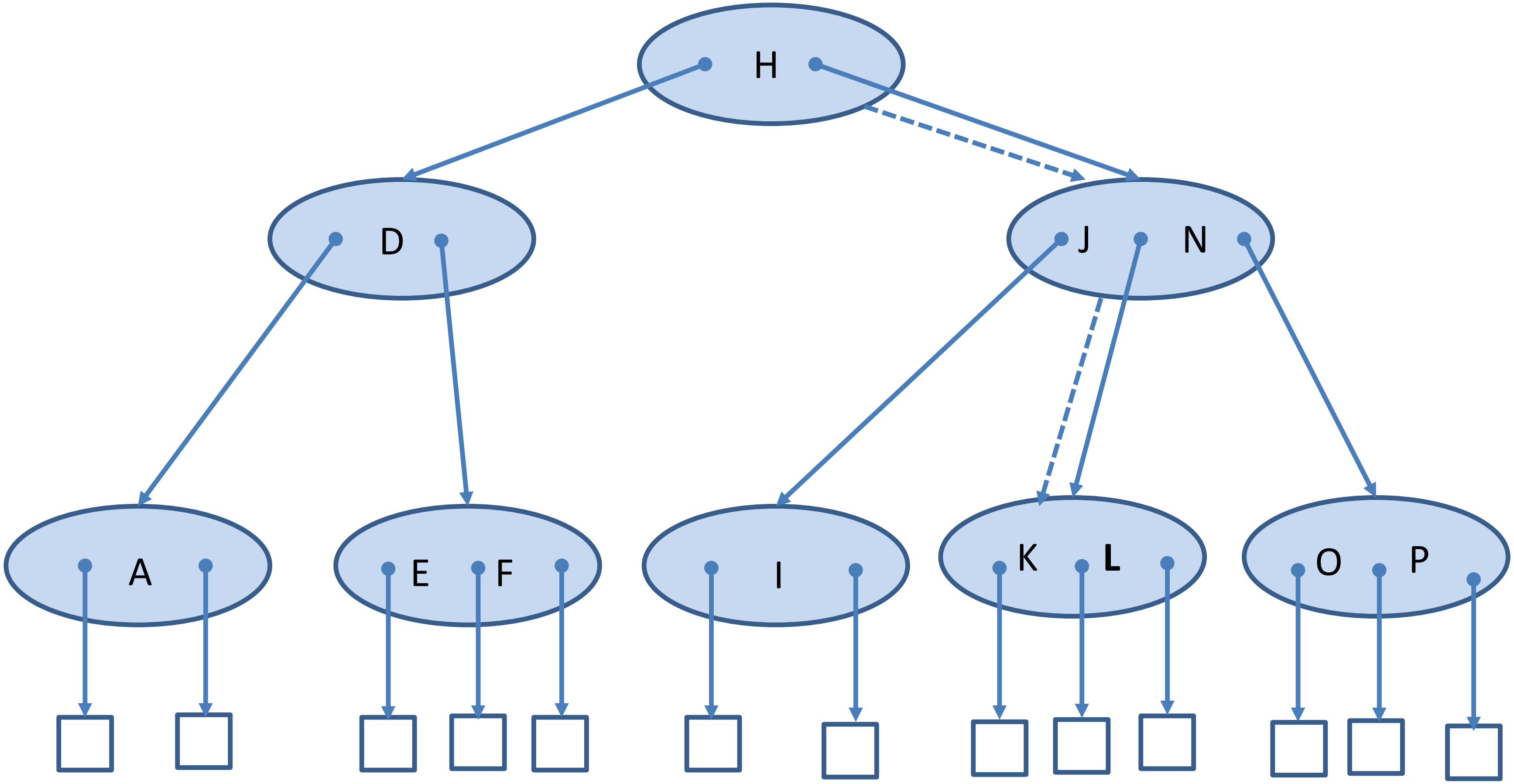
Example of 2-3 Tree



Searching in 2-3 Trees

- To search for the key L, for example, we start at the root and since L > H, we follow the pointer to the right subtree of the root node.
- Now we note that L lies between J and N, so we follow the middle pointer between the nodes J and N to the node containing the keys K and L.
- L is found and the search terminates.

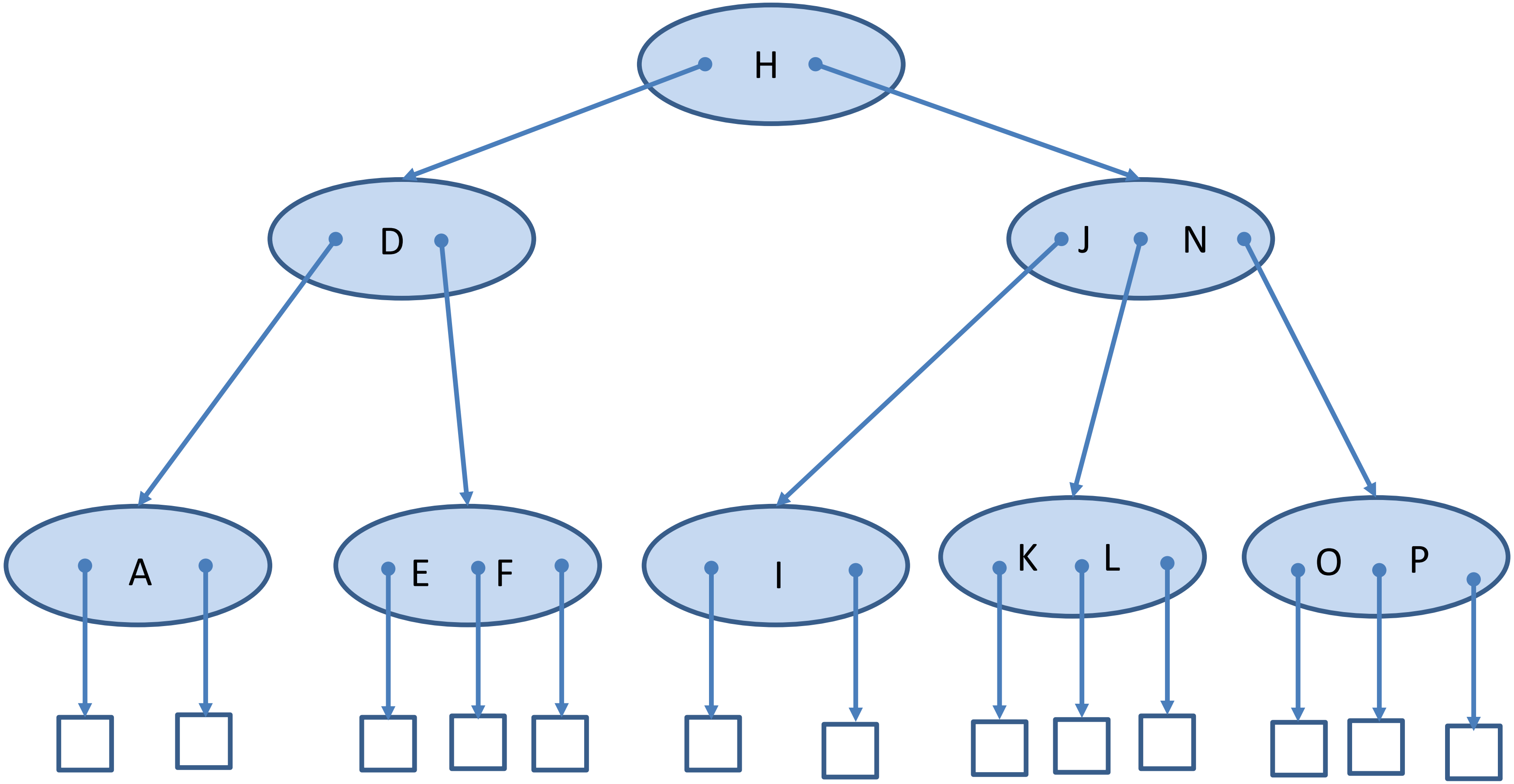
Search for Key L



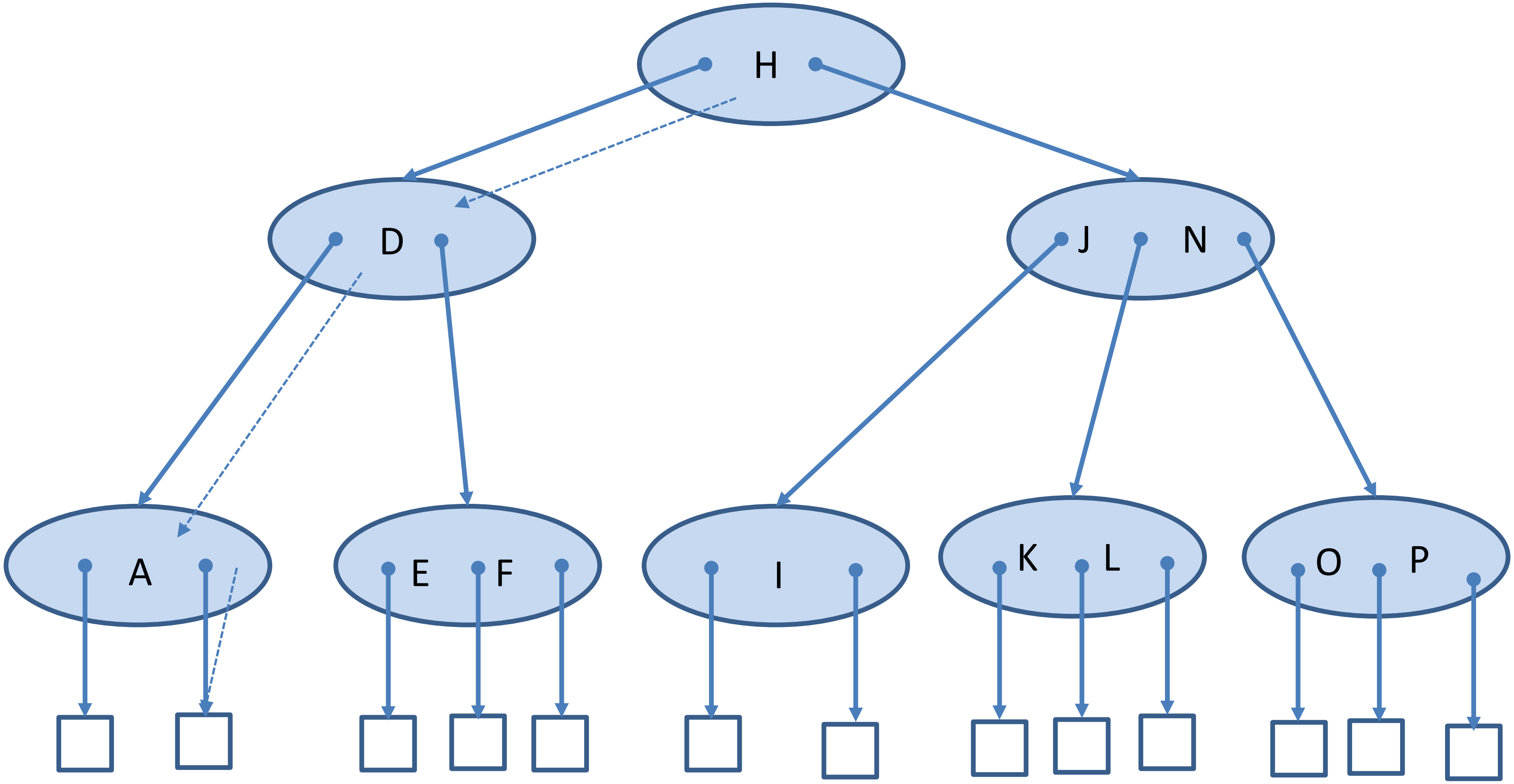
Insertion of New Keys

- Suppose we want to insert the new key B into the tree.
- Since B<H, we follow the left pointer from the root node to the node containing D in the second row.
- Then we follow the left pointer of D’s node to the node containing A.
- Since B>A, we follow the right pointer of A’s node which lead us to an **empty tree** (an external node).
- Then we go back to the parent of the external node and try to store the new key there.
- The node containing A has room for one more key so we store key B there. We also add a new empty child to this node.

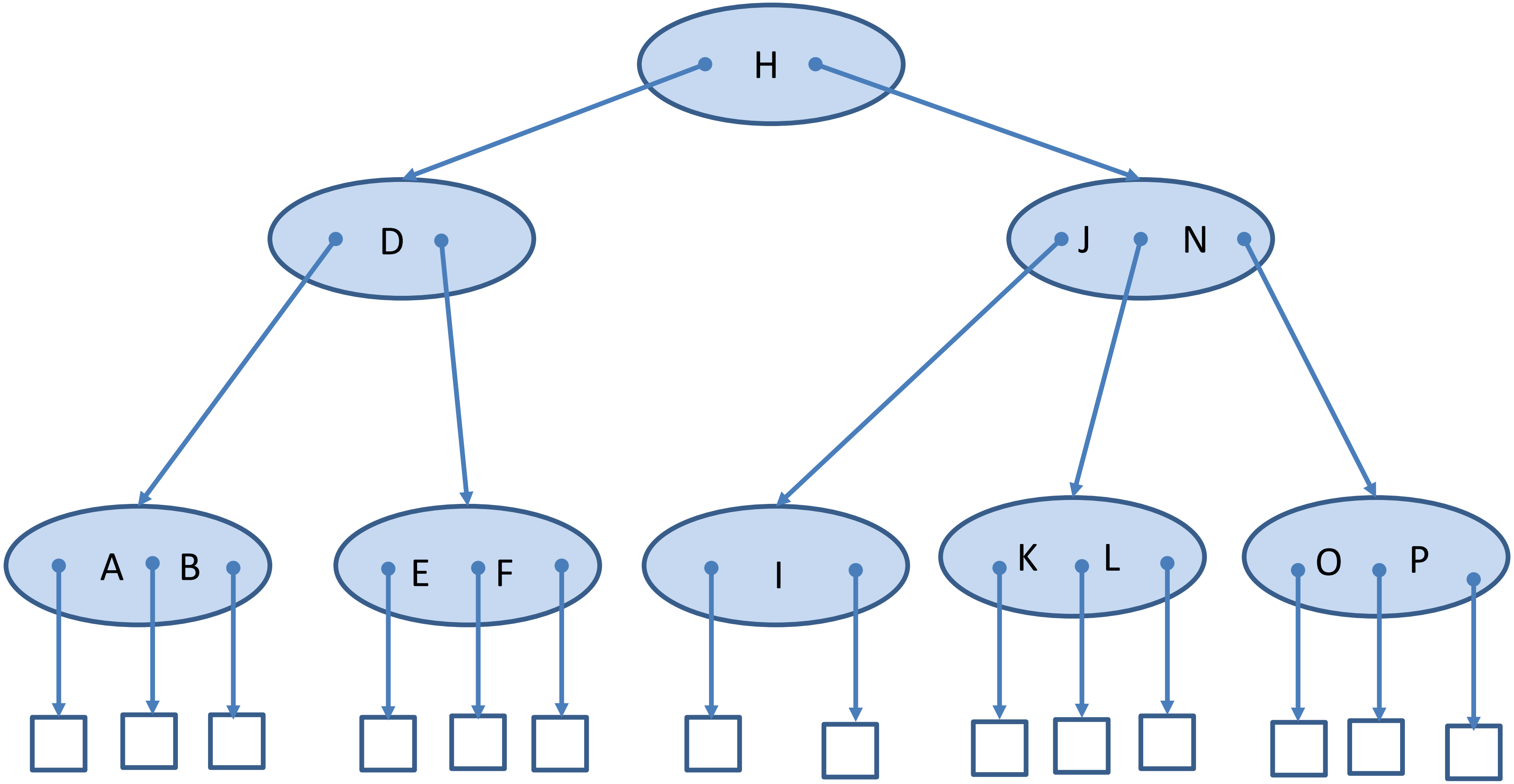
Example: Insert B



Example: Insert B



Example: Result



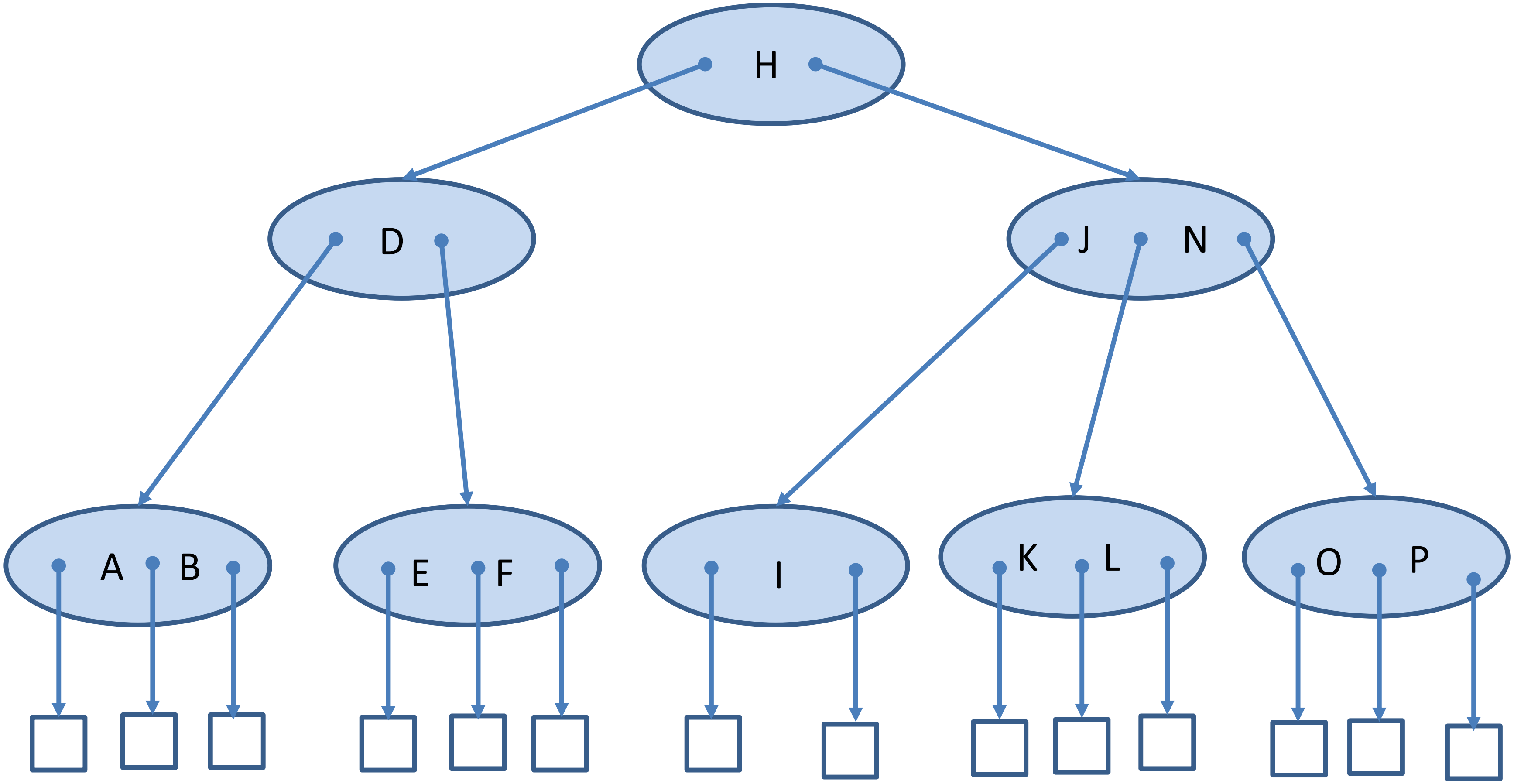
Insertion of New Keys (cont’d)

- Let us now insert key M. This leads to the attempt to add M to the node containing K and L which now **overflows** with keys K, L and M.
- The strategy for such cases is to **split the overflowed node into two nodes and pass the middle key to the parent.**
- Hence we split the overflowed node into two new nodes containing K and M respectively.
- We also pass the middle key L to the parent node containing J and N.

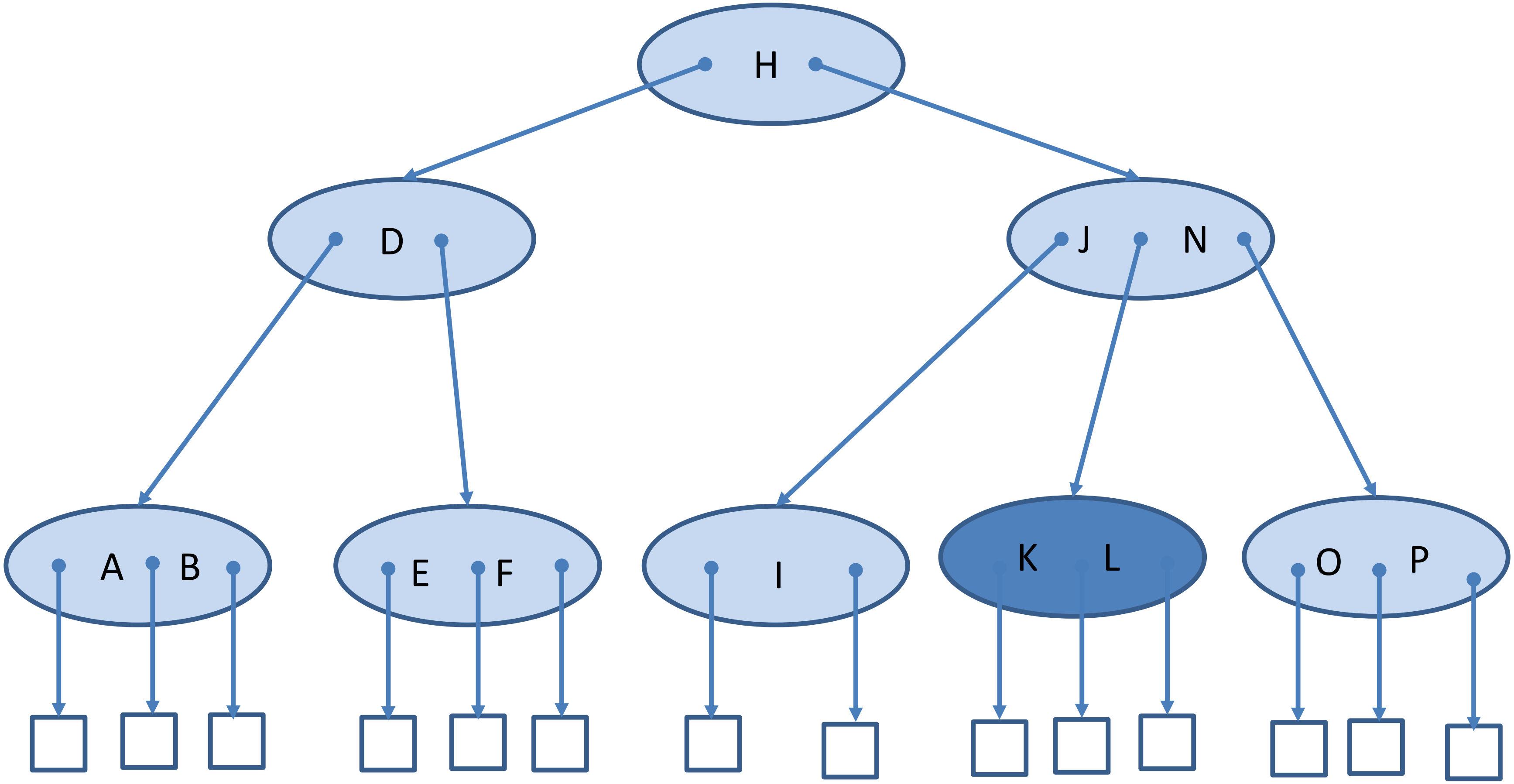
Insertion of New Keys (cont’d)

- The attempt to add L to this parent node results in a new overflowed node in which the key L lies between keys J and N.
- So we split this parent node into new nodes containing J and N respectively, and we pass the middle key L up to the root.
- The root has room for L so we store it there.

Example: Insert M

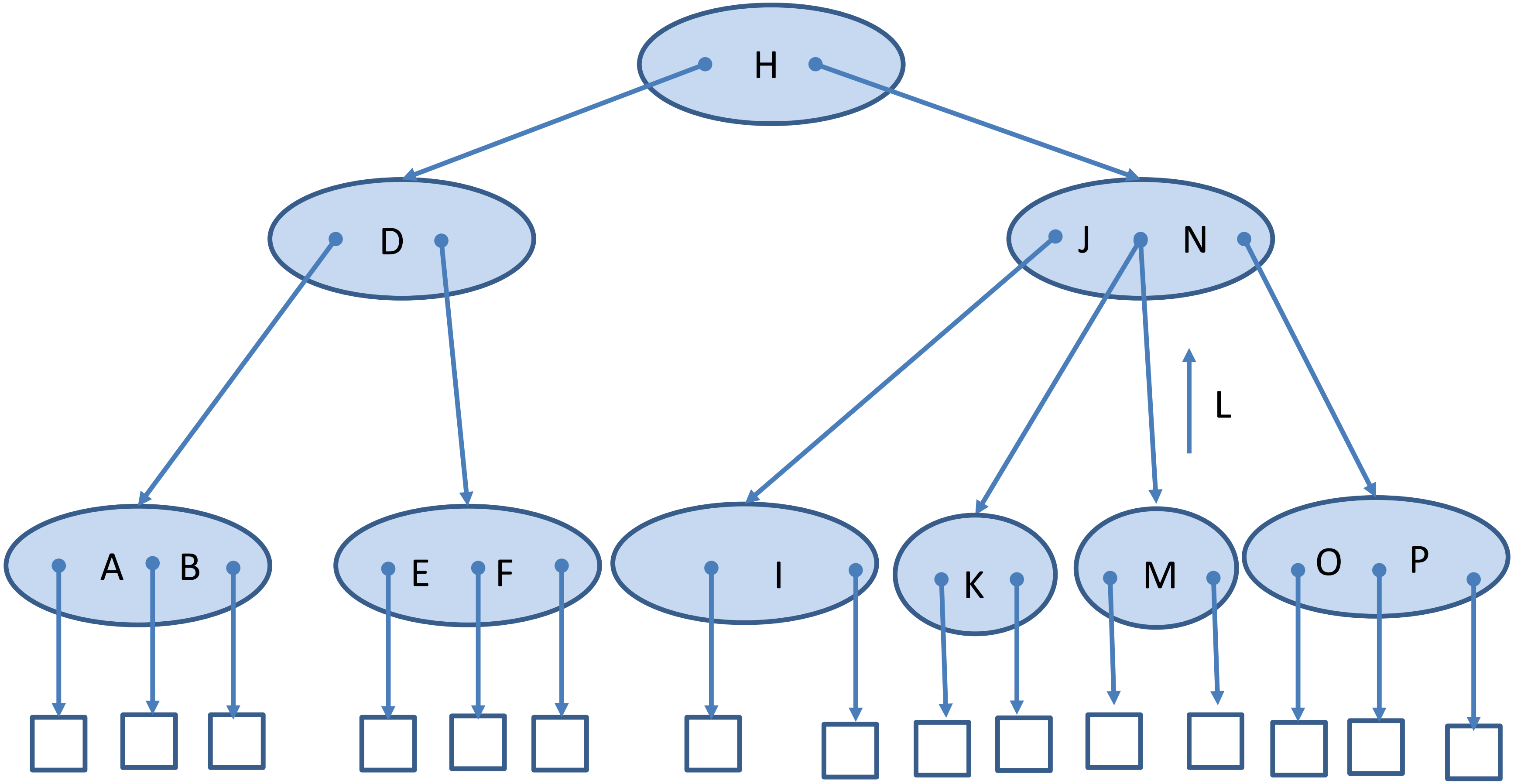


Example: Insert M (cont'd)



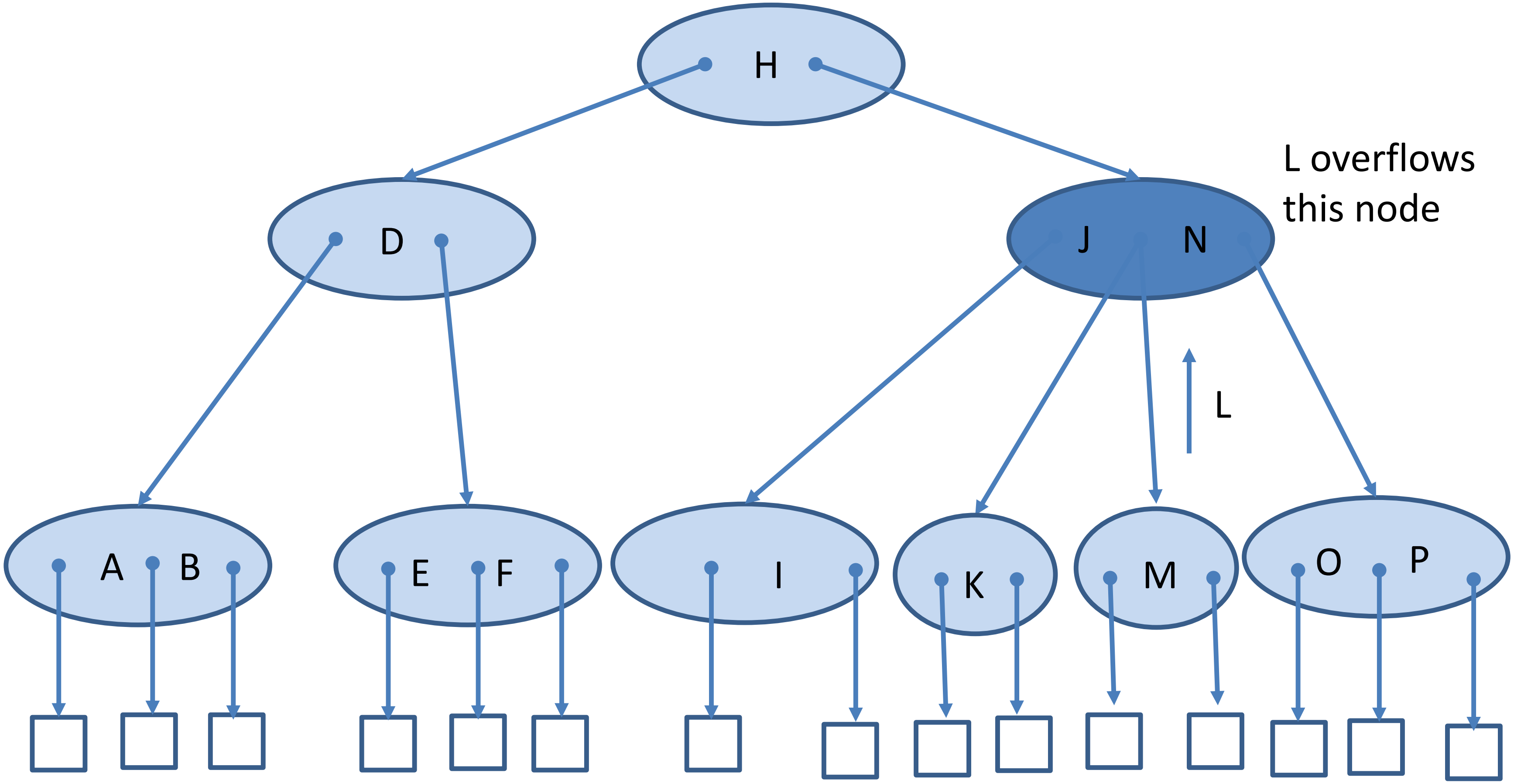
M overflows this node

Example: Insert M (cont'd)



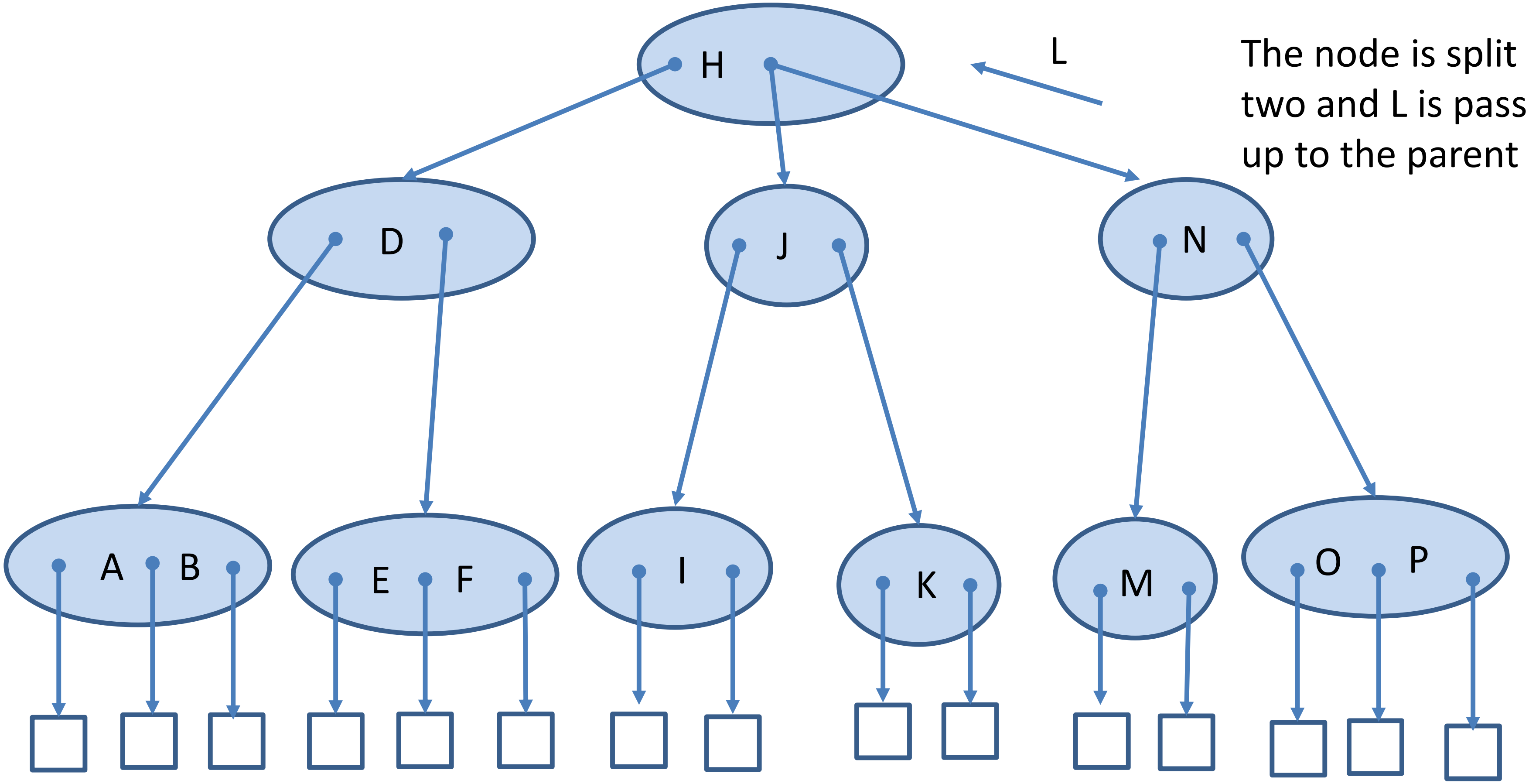
The node is split in two and L is passed to the parent node

Example: Insert M (cont'd)



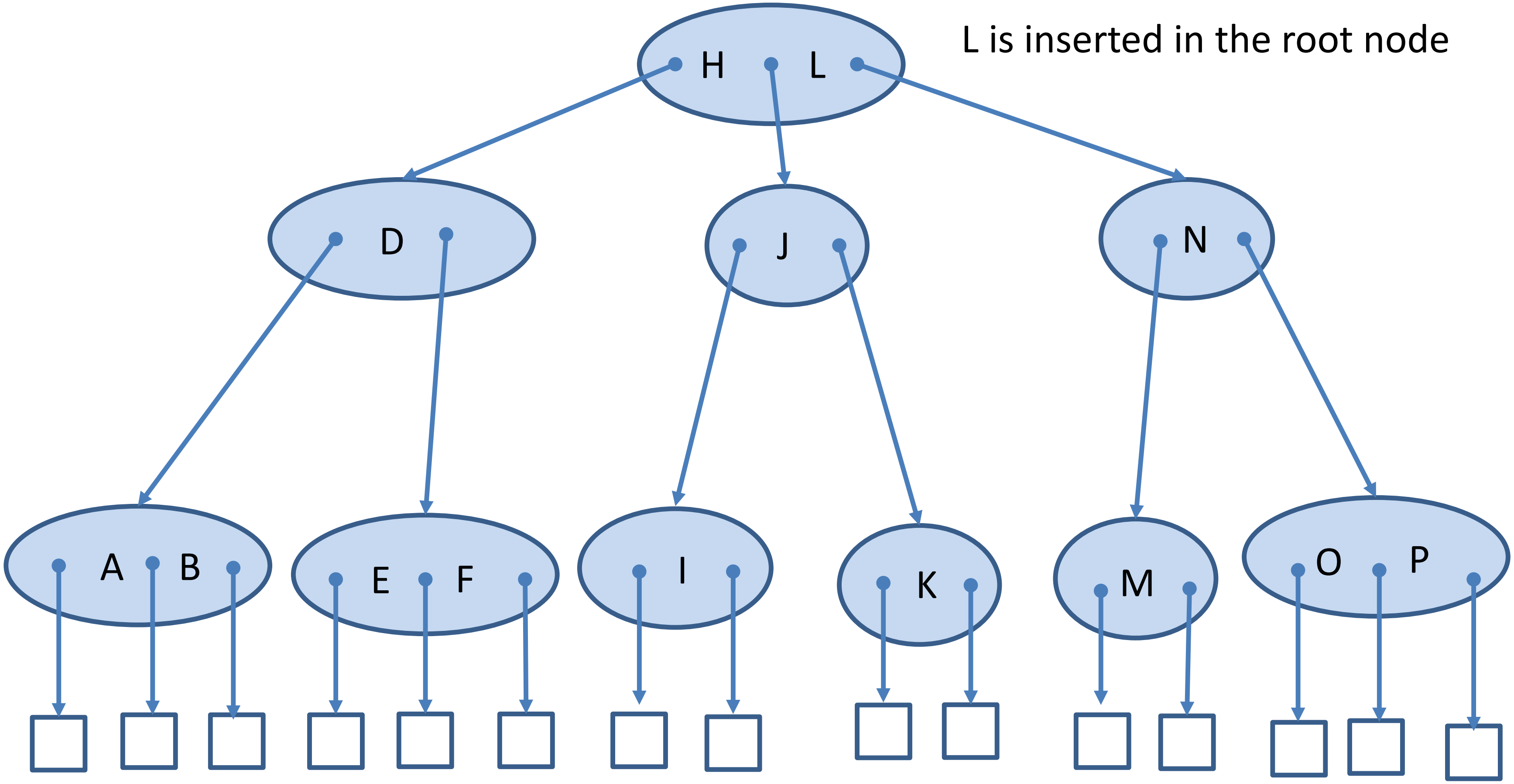
L overflows this node

Example: Insert M (cont'd)



The node is split in two and L is passed up to the parent

Example: Result

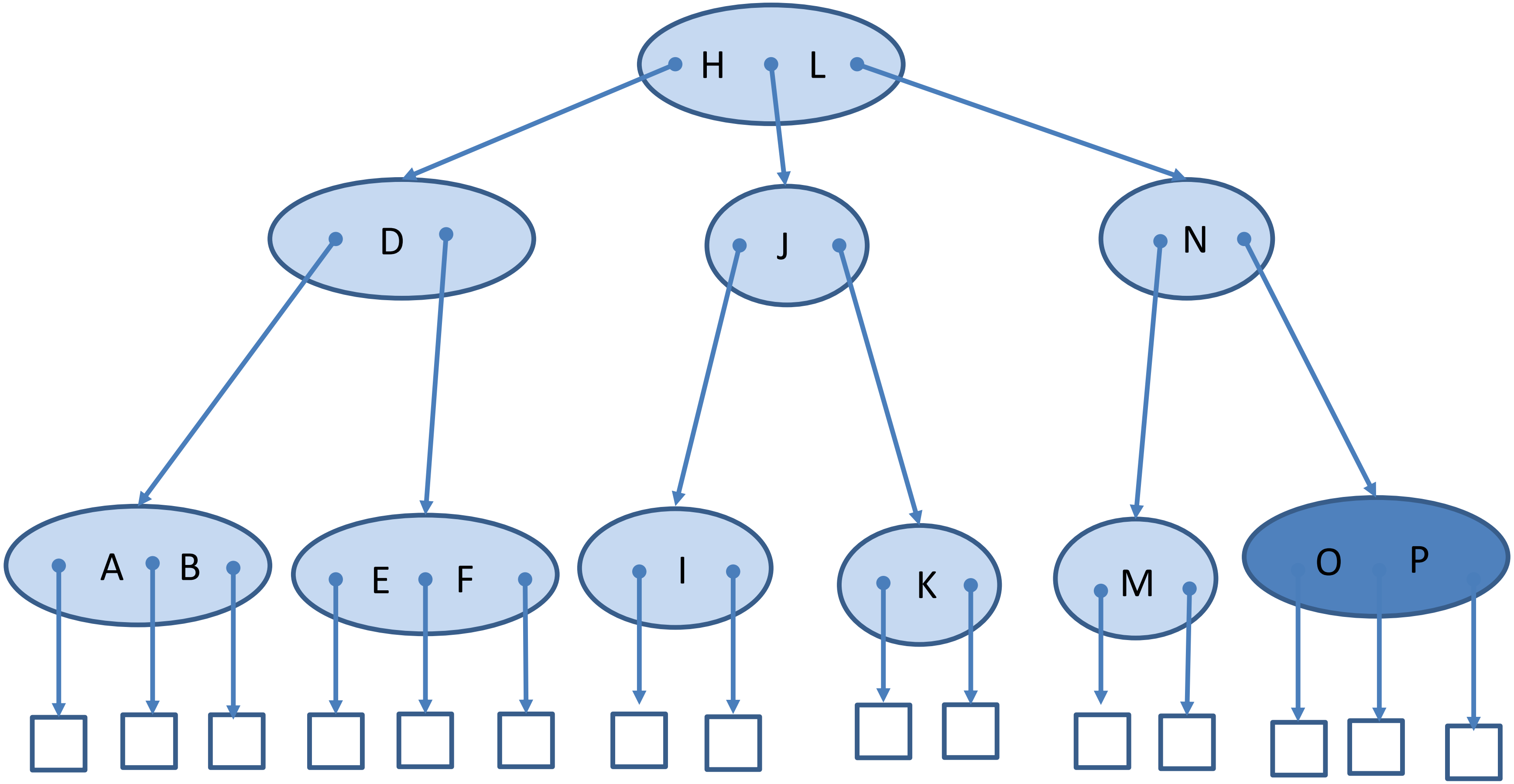


L is inserted in the root node

Insertion of New Keys (cont’d)

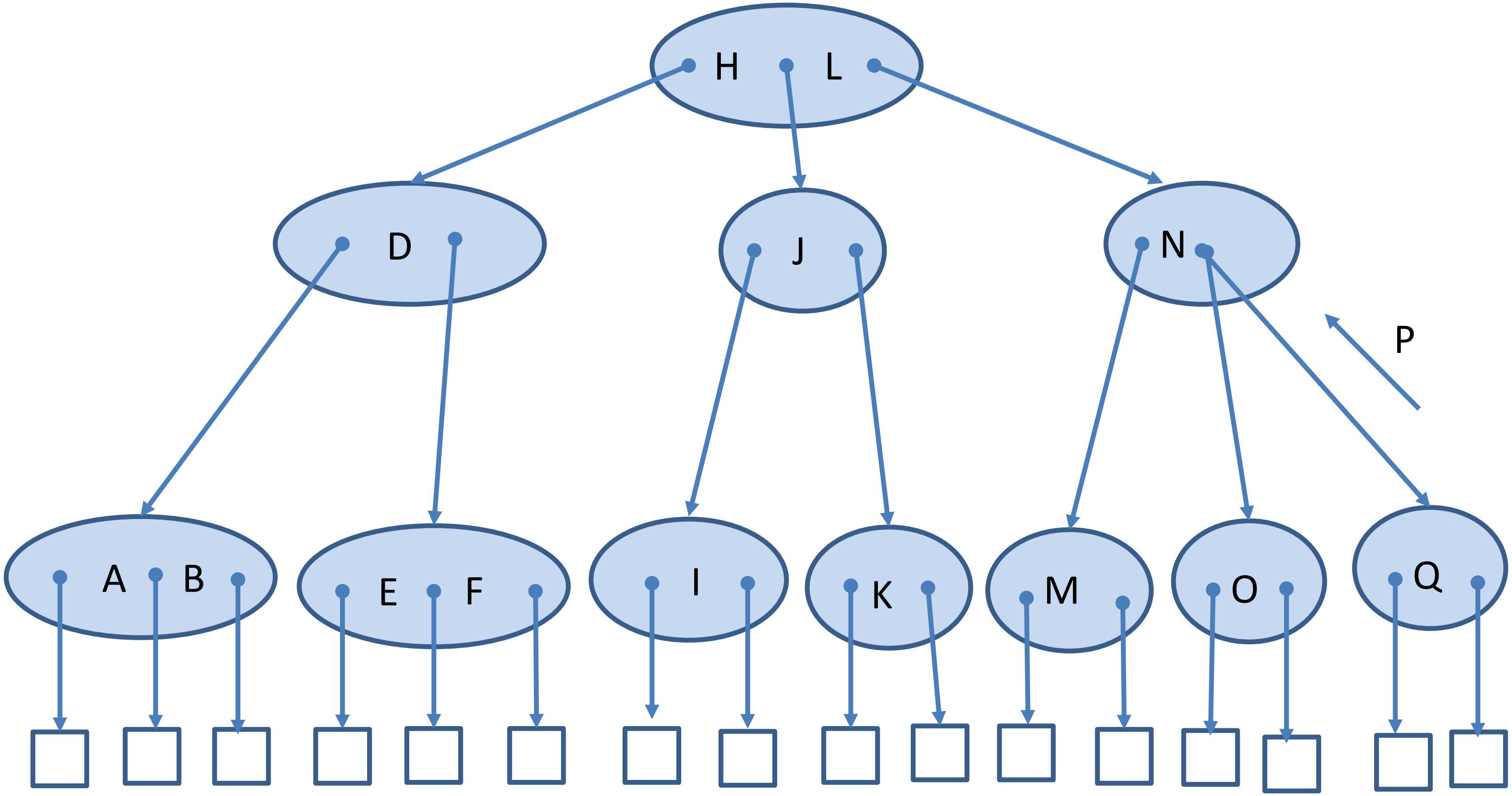
- Let us now insert key Q. Q should be entered in the node containing O and P which now overflows.
- Thus, this node is split up to two nodes one containing O and the other containing R, and the middle key is passed up towards the parent node.
- The parent node has only key N so there is space for P and it is inserted there.

Example: Insert Q



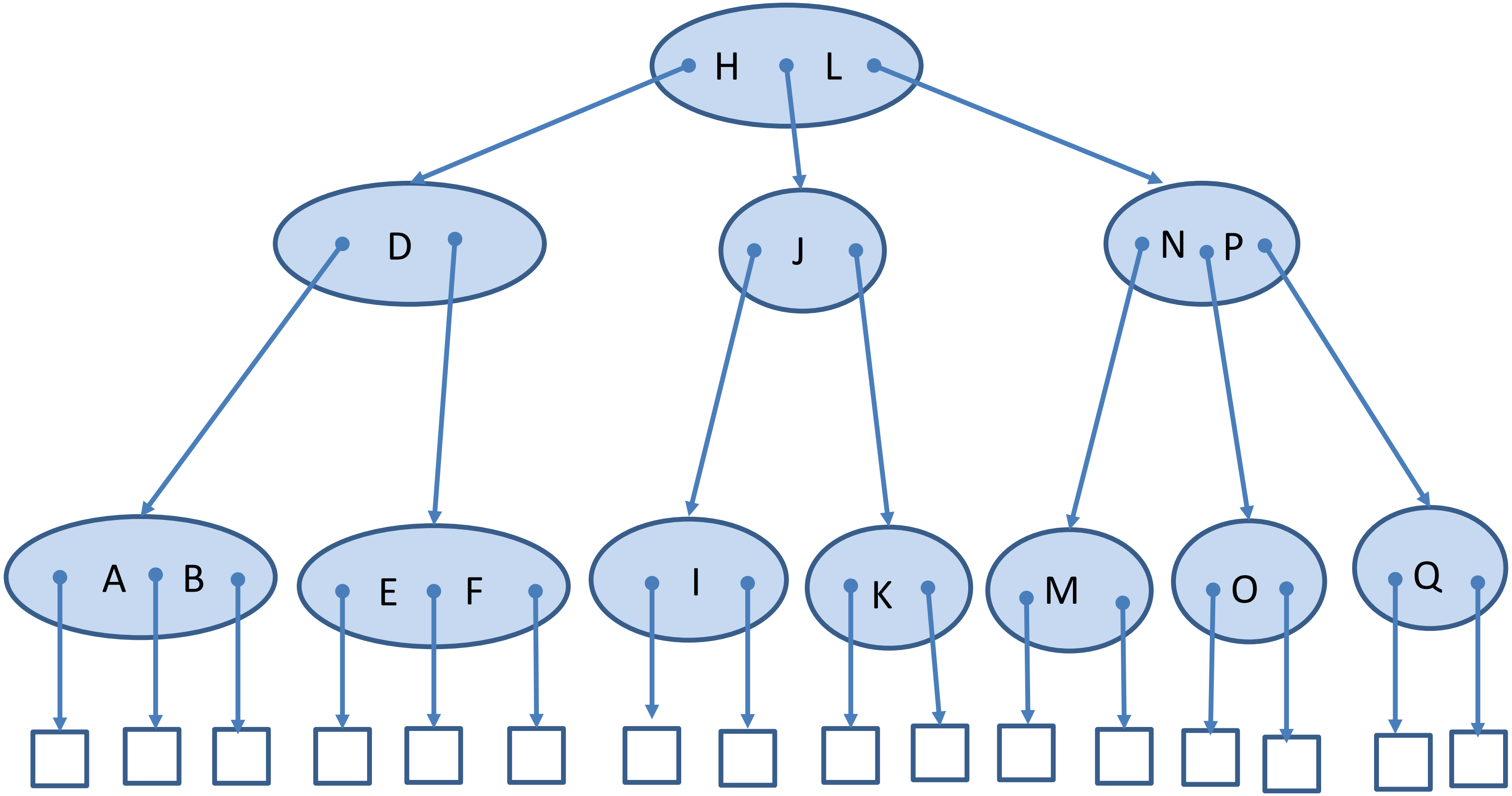
Q overflows this node

Example: Insert Q (cont’d)



This node is split up and P is passed up

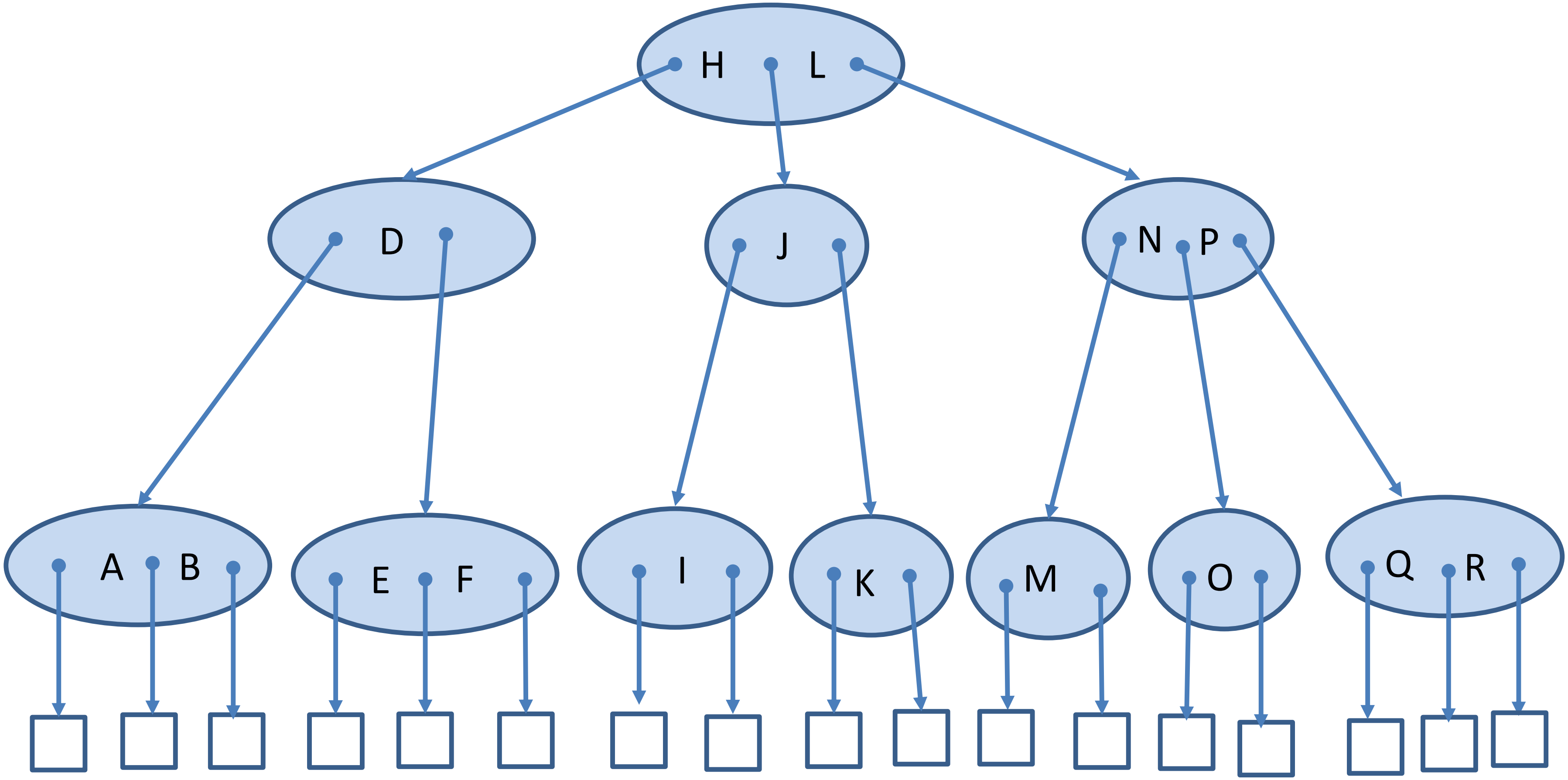
Example: Result



Insertion of New Keys (cont’d)

- Let us now insert key R. R is inserted in the node with Q where there is space.

Example: Insert R



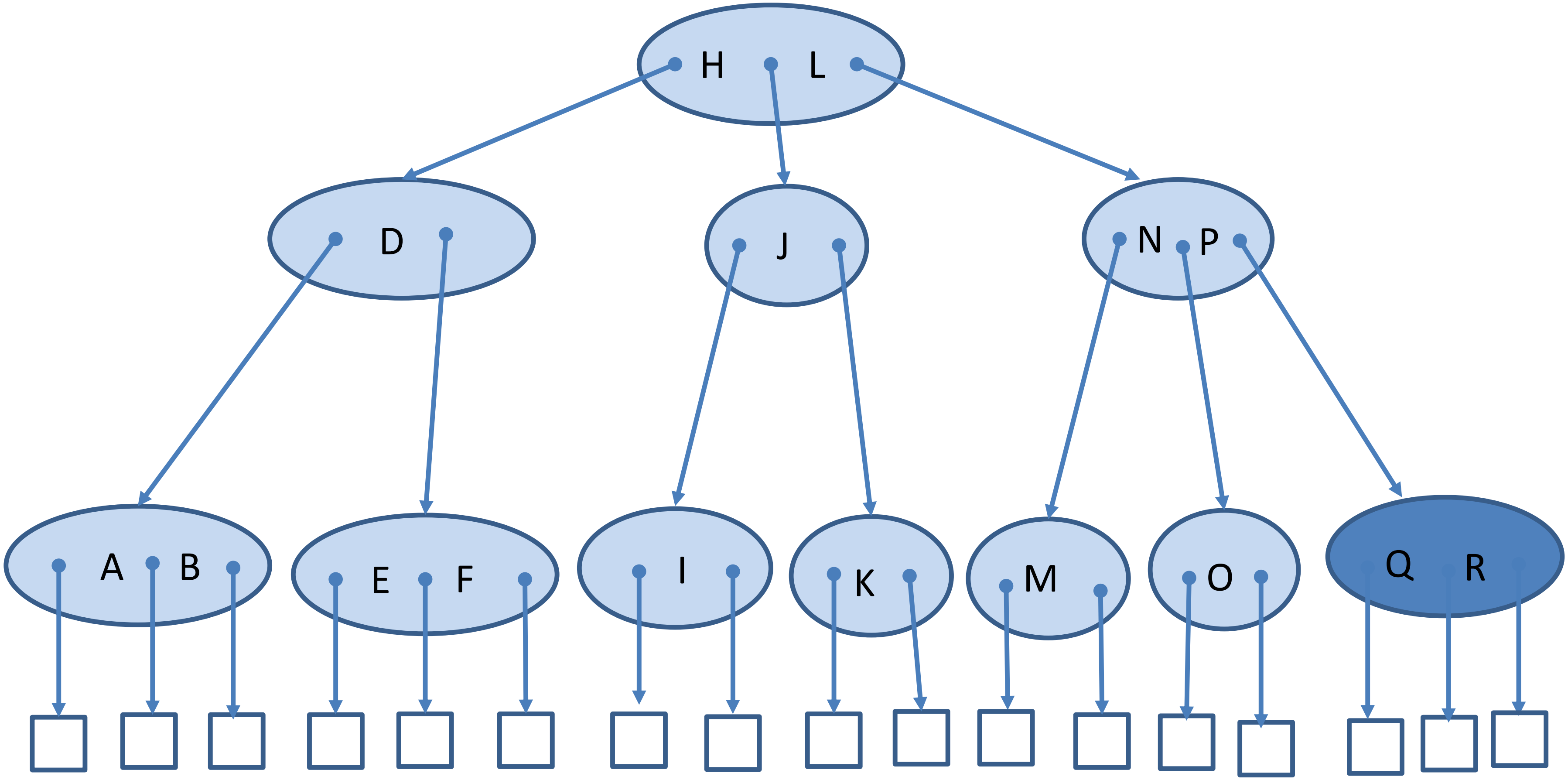
Inserting New Keys (cont'd)

- Let us now insert key S. S should be inserted in the node with Q and R.
- This node overflows. Thus, it is split into two nodes one containing Q and the other containing S and R is passed up to the parent node.
- R now oveflows this node where N and P are also stored. Thus, this node is split into two nodes one containing N and the other containing R and the middle key P is sent up to the parent (the root).

Inserting New Keys (cont’d)

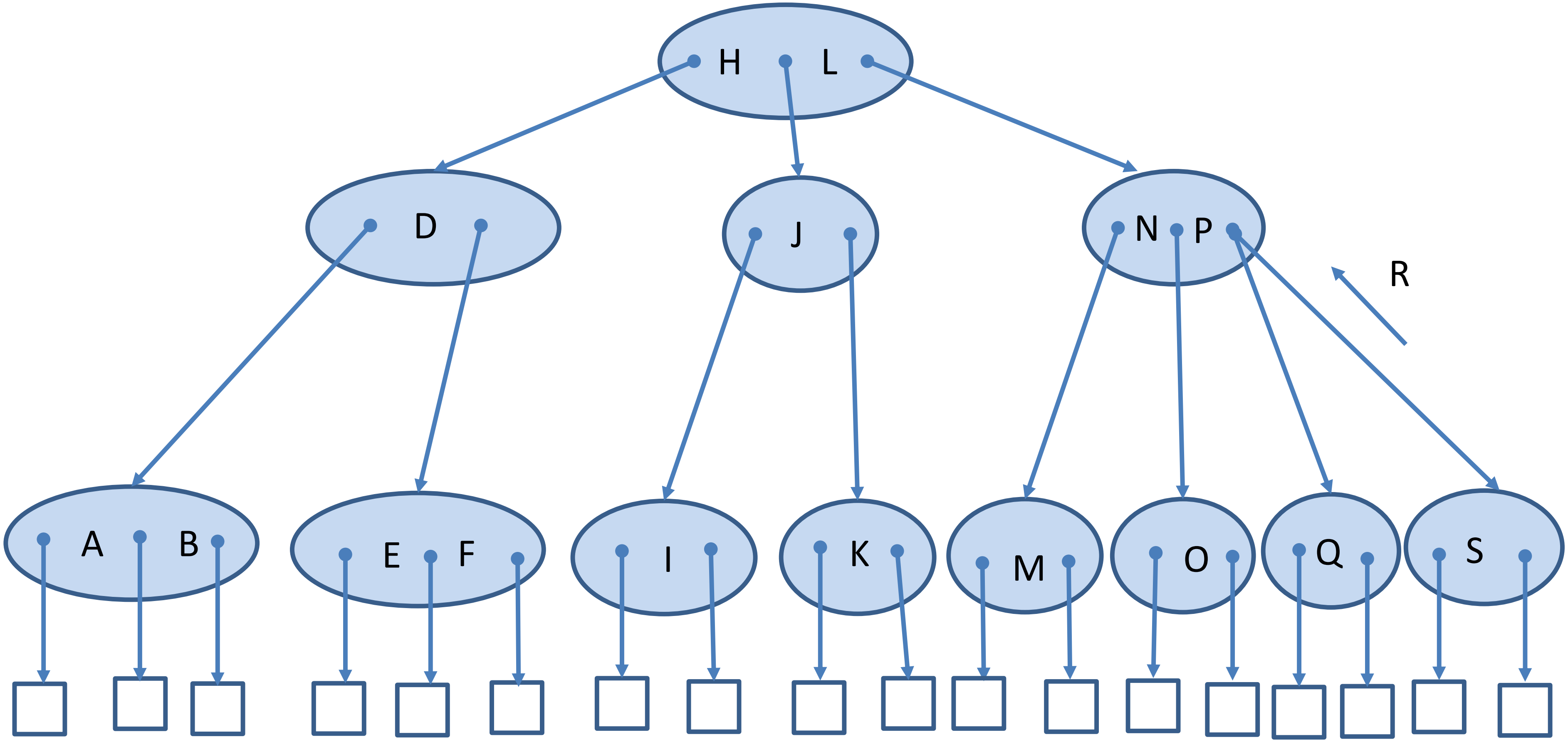
- The root now overflows with the addition of P so it is split into two nodes one containing H and the other containing P and the middle key L is used to **create a new root**.
- Thus, we have added **one more level** to the tree.

Example: Insert S



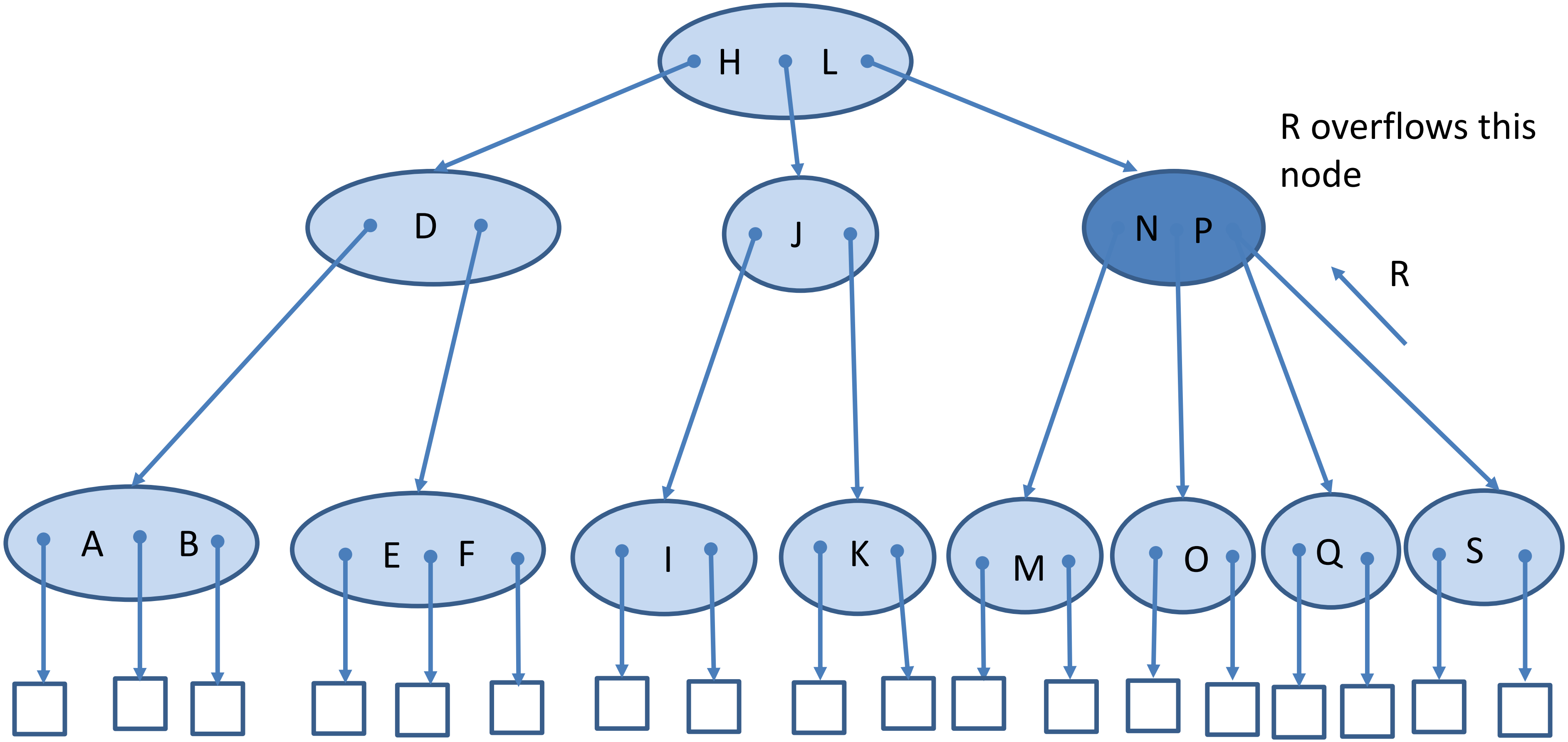
S overflows this node

Example: Insert S (cont'd)

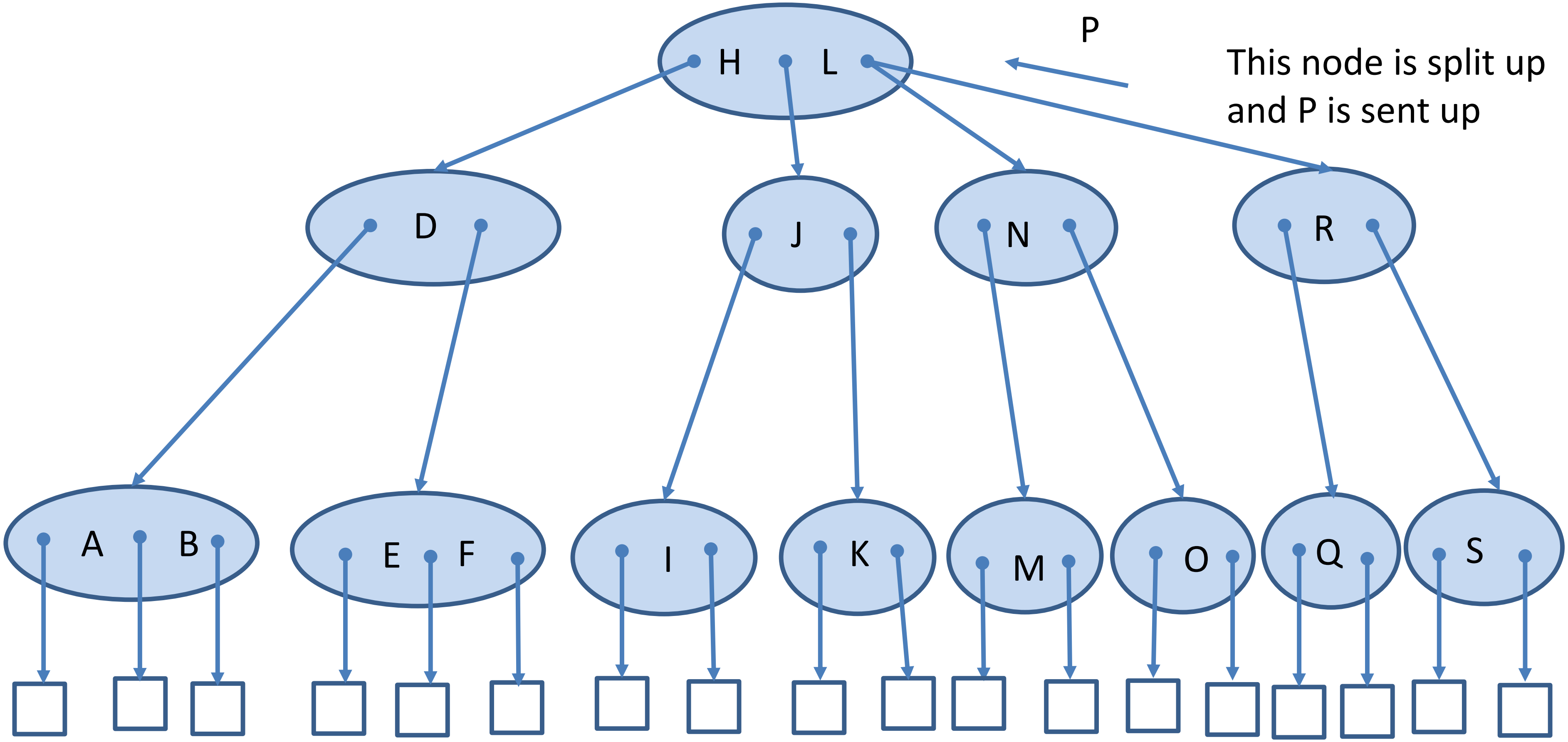


This node is split and R is sent up

Example: Insert S (cont’d)



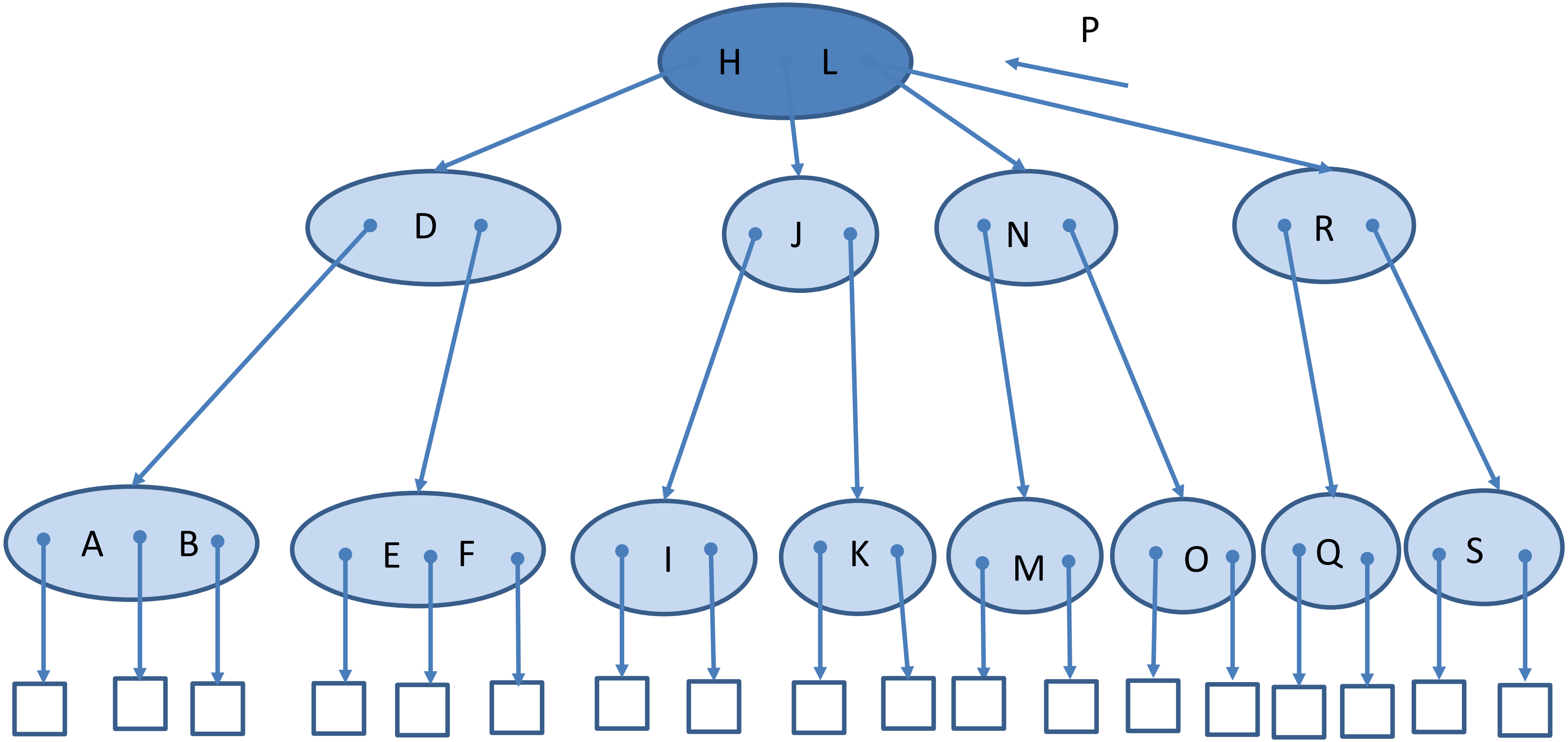
Example: Insert S (cont'd)



This node is split up and P is sent up

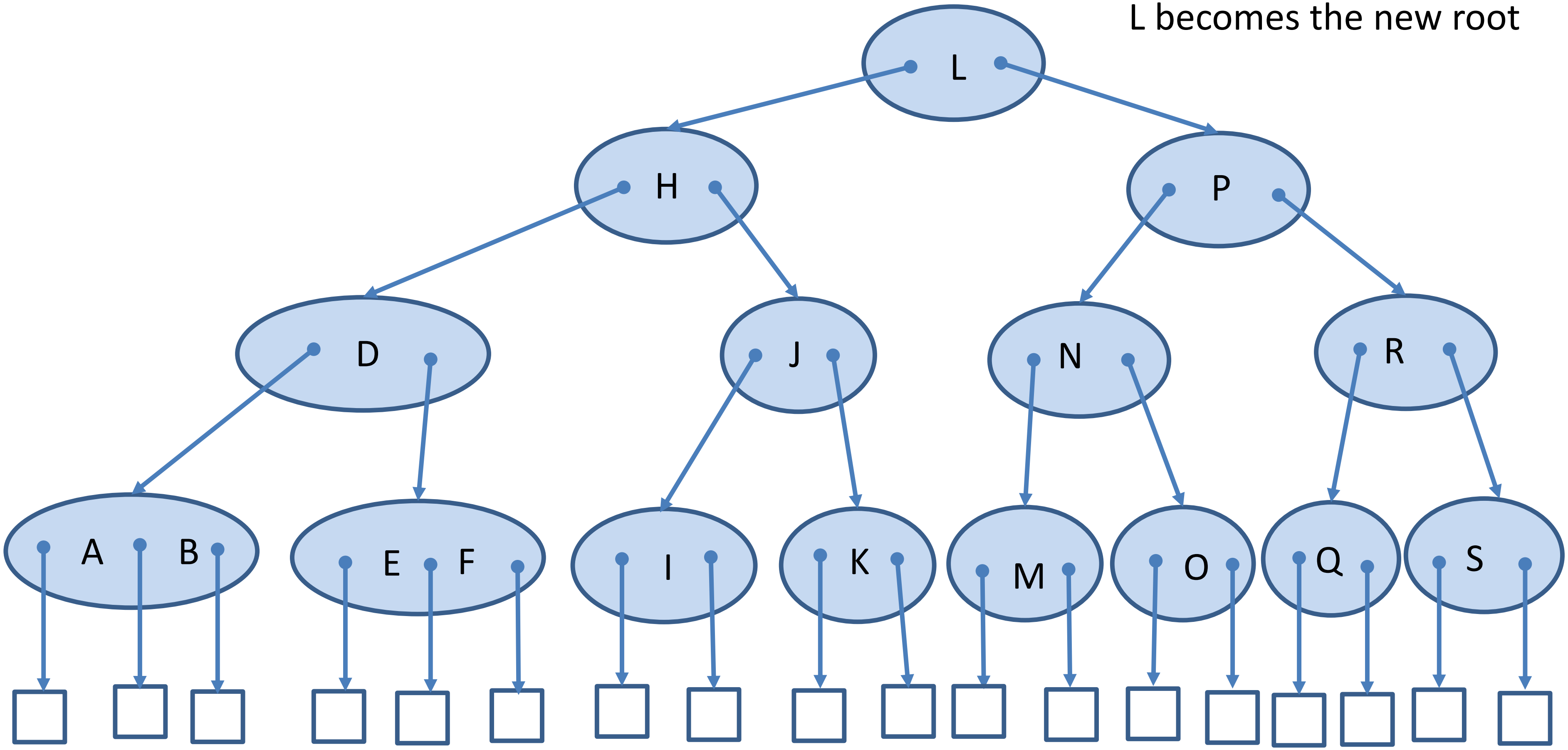
Example: Insert S (cont'd)

P overflows the root



Example: Result

The root splits and
L becomes the new root



Complexity of Insertion in 2-3 Trees

- When we insert a key at level k, in the worst case we need to split $k+1$ nodes (one at each of the k levels plus the root).
- A 2-3 tree containing n keys with the **maximum number of levels** takes the form of a **binary tree where each internal node has one key and two children**.
- In such a tree $n = 2^{k+1} - 1$ where k is the number of the lowest level.
- This implies that $(k+1) = \log(n+1)$ from which we see that the splits are in the worst case $O(\log n)$.
- So insertion in a 2-3 tree takes at worst $\boldsymbol{O(\log n)}$ **time**.
- Similarly we can prove that searches and deletions take $\boldsymbol{O(\log n)}$ **time**.

(2,4) Trees

- A **(2,4) tree** or **2-3-4 tree** is a multi-way search tree which has the following two properties:
  - **Size property**: Every internal node contains at least one and at most three keys, and has at least two and at most four children.
  - **Depth property**: All the external nodes are empty trees that have the same depth (lie on a single bottom level).

Result

- **Proposition**. The height of a (2,4) tree storing n entries is $O(\log n)$.
- **Proof**: Let h be the height of a (2,4) tree T storing n entries. We justify the proposition by showing that

$$\frac{1}{2}\log(n+1) \leq h$$

and

$$h \leq \log(n+1).$$

Result (cont'd)

- Note that by the size property, we have at most 4 nodes at depth 1, at most 4^2nodes at depth 2, and so on. Thus, the number of external nodes of T is at most 4^h.
- Similarly, by the size property, we have at least 2 nodes at depth 1, at least 2^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at least 2^h.
- We also know that the number of external nodes is $n + 1$.

Result (cont'd)

- Therefore, we obtain

$$2^h \leq n + 1$$

  and

$$n + 1 \leq 4^h.$$

- Taking the logarithm in base 2 of each of the above terms, we get that

$$h \leq \log(n + 1)$$

  and

$$\log(n + 1) \leq 2h.$$

- These inequalities prove our claims.

Insertion in (2,4) Trees

- To insert a new entry (k, x), with key k, into a (2,4) tree T, we first perform a search for k.
- Assuming that T has no entry with key k, this search terminates unsuccessfully at an external node z.
- Let v be the parent of z. We insert the new entry into node v and add a new child (an external node) to v on the left of z.

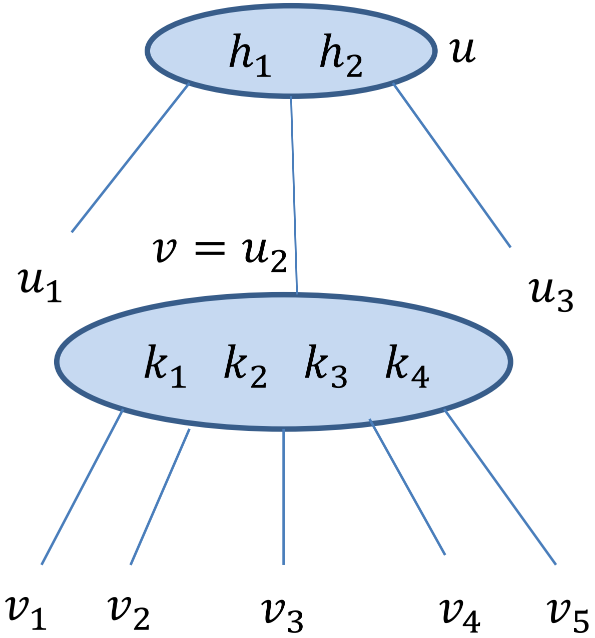
Insertion (cont’d)

- Our insertion method **preserves the depth property**, since we add a new external node at the same level as existing external nodes.
- But it might **violate the size property**. If a node v was previously a 4-node, then it may become a 5-node after the insertion which causes the tree to longer be a (2,4) tree.
- This type of violation of the size property is called an **overflow** node at node v, and it must be resolved in order to restore the properties of a (2,4) tree.

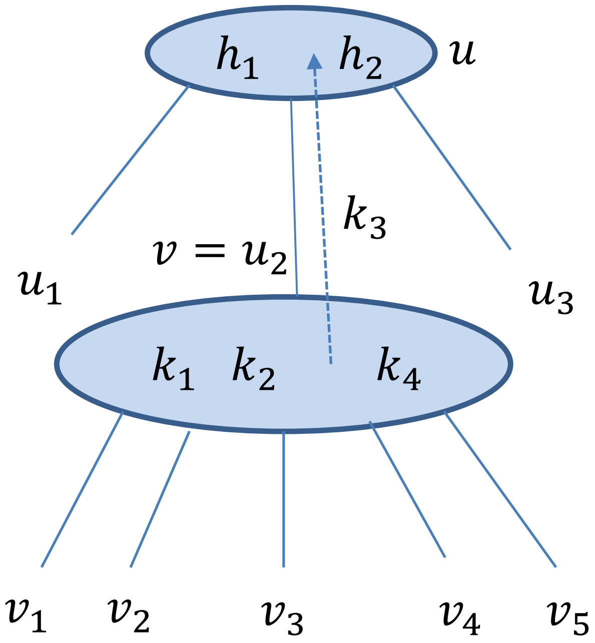
Dealing with Overflow Nodes

- Let $v_1, \cdots, v_5$ be the children of v, and let $k_1, \cdots, k_4$ be the keys stored at v. To remedy the overflow at node v, we perform a **split** operation on v as follows.
- Replace v with two nodes v' and v'', where
  - v' is a 3-node with children v_1, v_2, v_3 storing keys k_1 and k_2
  - v'' is a 2-node with children v_4, v_5, storing key k_4.
- If v was the root of T, create a new root node u. Else, let u be the parent of v.
- Insert key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i + 1$ of u, respectively.

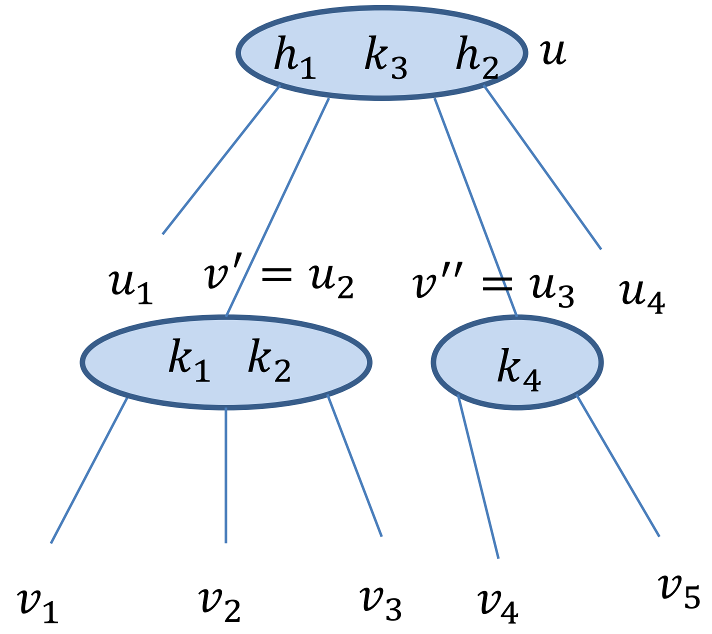
Overflow at a 5-node



The third key of v inserted into the parent node u



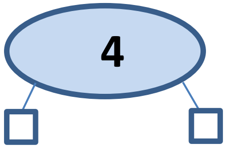
Node v replaced with a 3-node v' and a 2-node v''



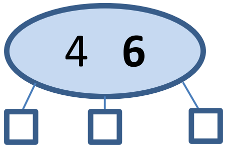
Example

- Let us now see an example of a few insertions into an initially empty (2,4) tree.

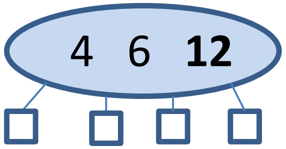
Insert 4



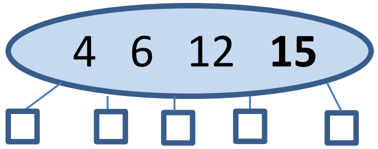
Insert 6



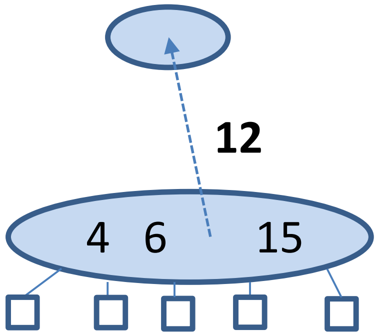
Insert 12



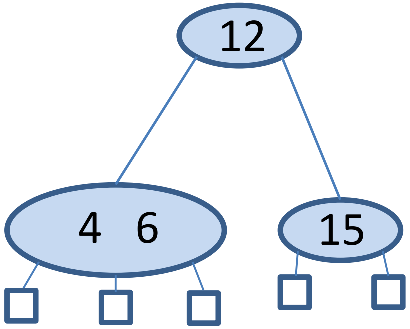
Insert 15 - Overflow



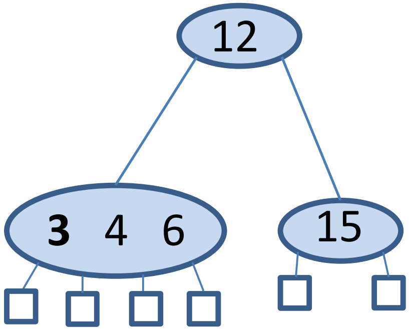
Creation of New Root Node



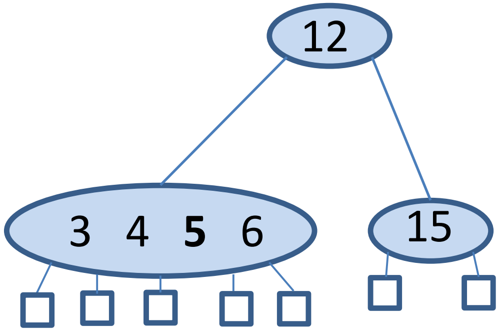
Split



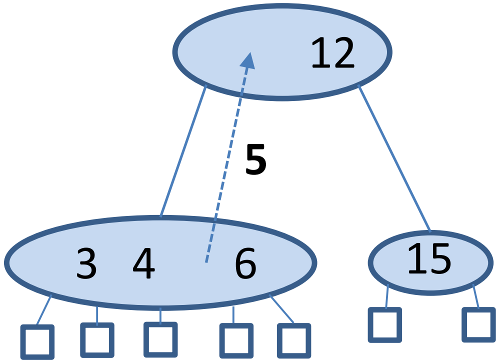
Insert 3



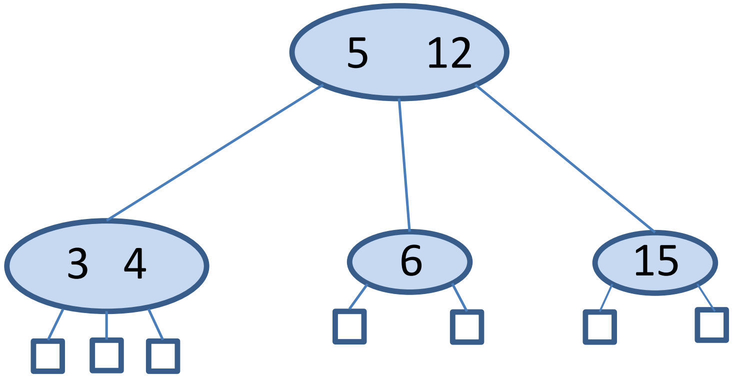
Insert 5 - Overflow



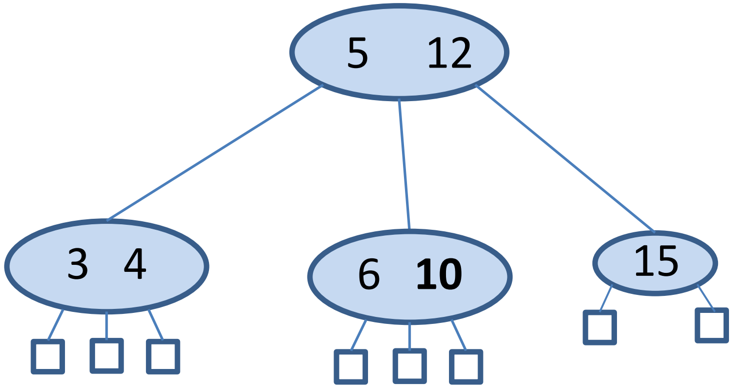
5 is Sent to the Parent Node



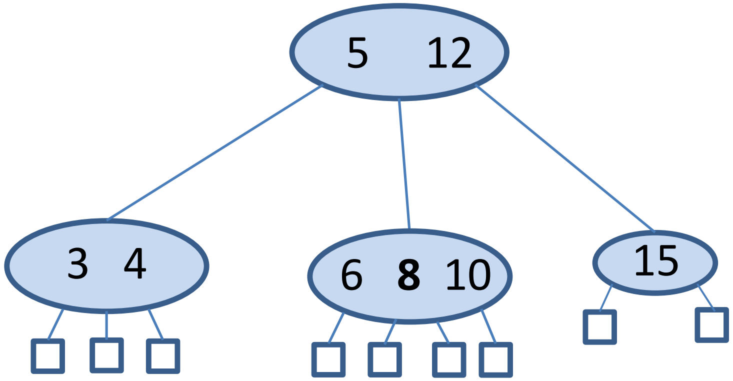
Split



Insert 10



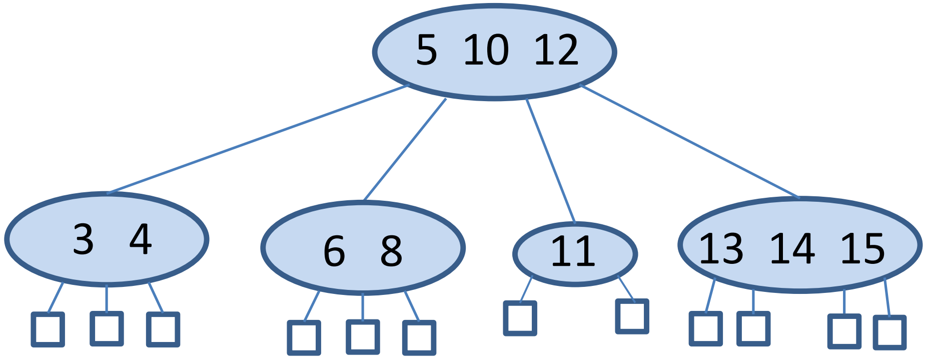
Insert 8



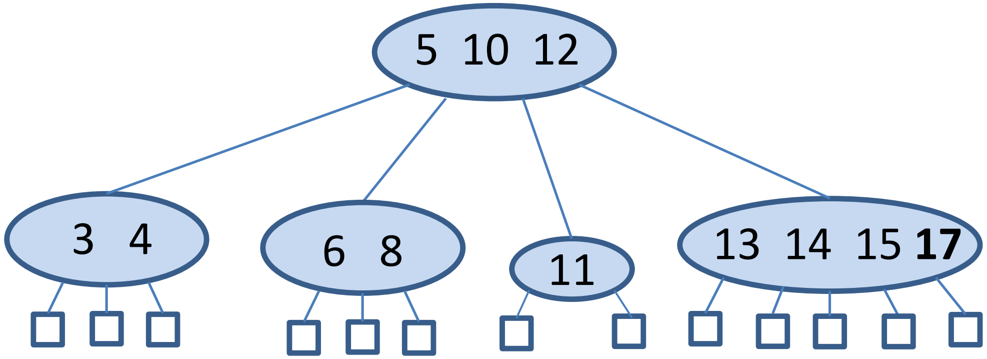
Insertion (cont’d)

- Let us now see a more complicated example of insertion in a (2,4) tree.

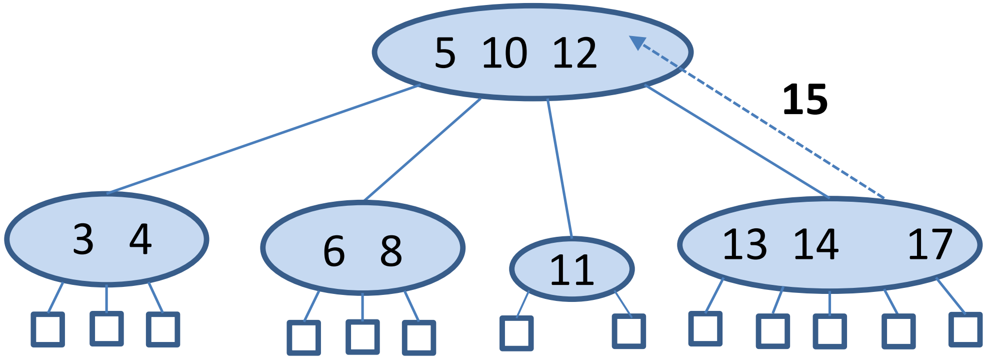
Initial Tree



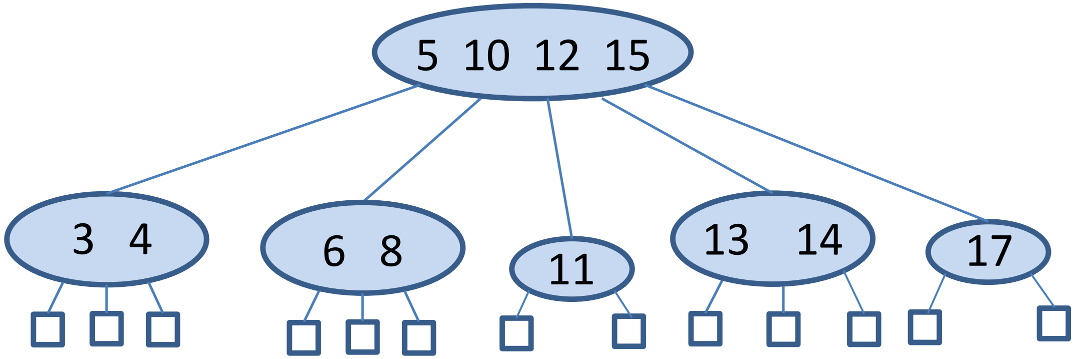
Insert 17 - Overflow



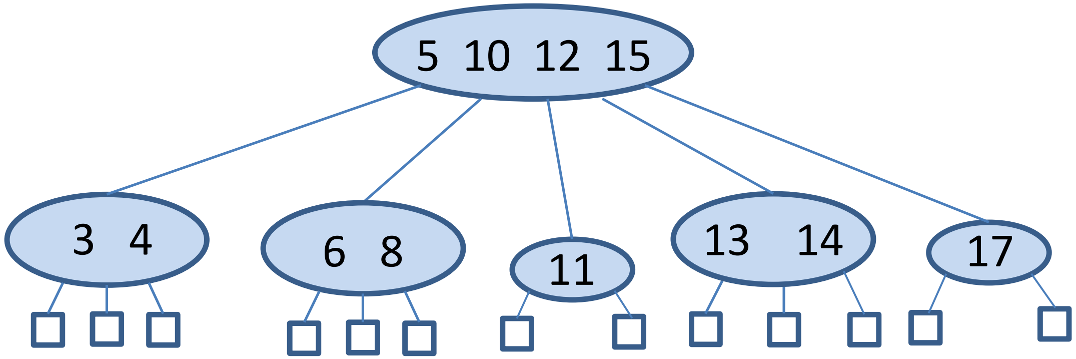
15 is Sent to the Parent Node



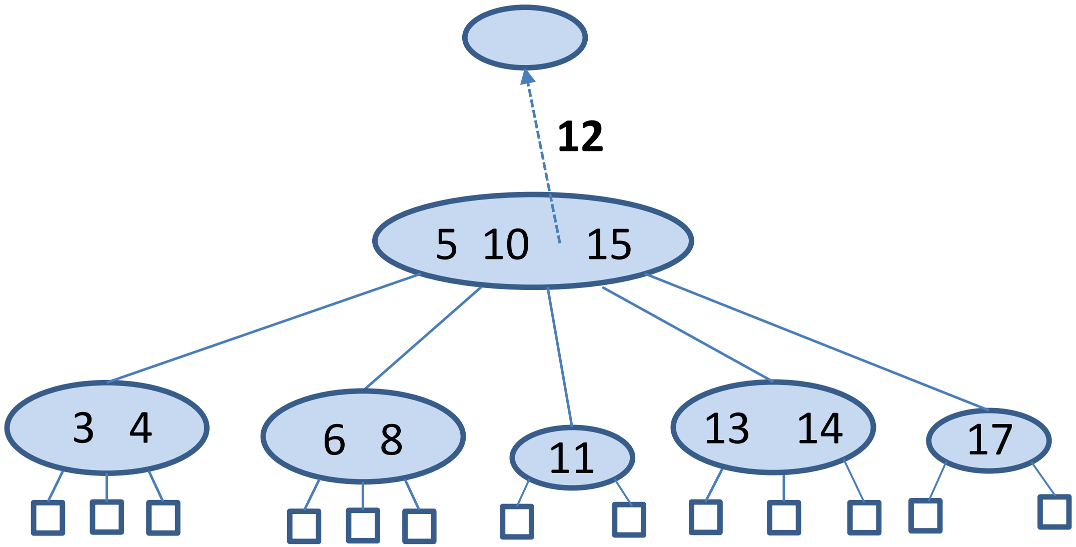
Split



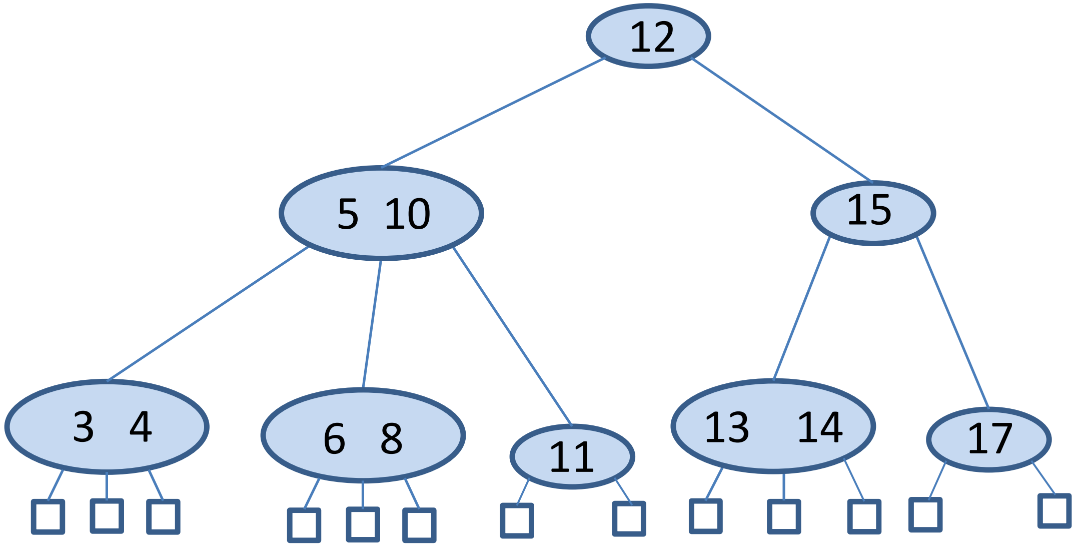
Overflow at the Root



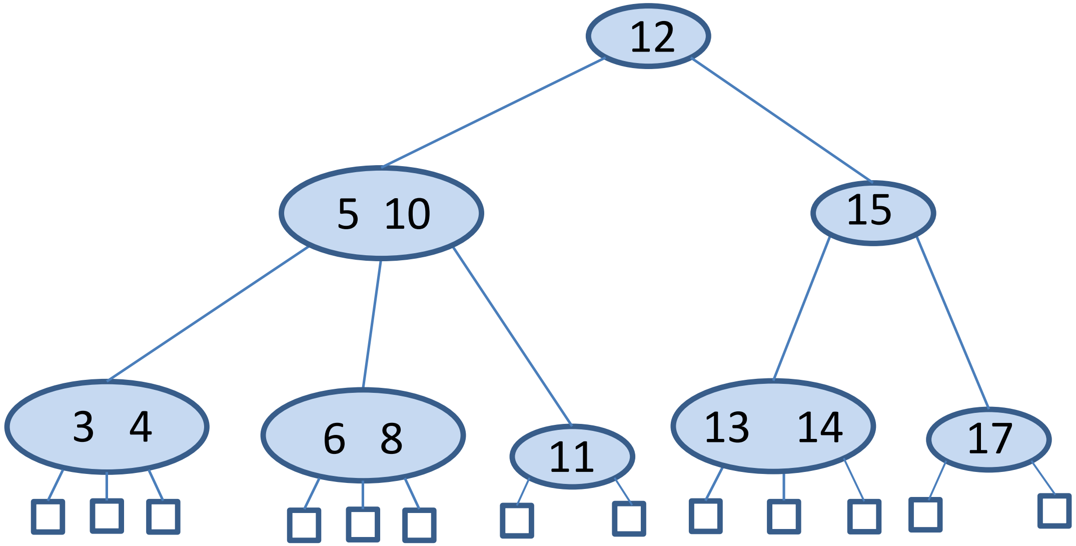
Creation of New Root



Split



Final Tree



Complexity Analysis of Insertion

- **A split operation affects a constant number of nodes** of the tree and a constant number of entries stored at such nodes. Thus, it can be implemented in $\boldsymbol{O(1)}$ **time**.
- As a consequence of a split operation on node v, a new overflow may arise at the parent u of v. A split operation either eliminates the overflow or it propagates it into the parent of the current node. Hence, **the number of split operations is bounded by the height of the tree**, which is $O(\log n)$.
- Therefore, the total time to perform an insertion is $\boldsymbol{O(log\,n)}$.

Removal in (2,4) Trees

- Let us now consider the removal of an entry with key k from a (2,4) tree T.
- Removing such an entry can always be reduced to the case where the entry to be removed is stored at a node v whose children are external nodes.
- Suppose that the entry we wish to remove is stored in the i-th entry (k_i, x_i) at a node z that has only internal nodes as children. In this case, we swap the entry (k_i, x_i) with an appropriate entry that is stored at a node v with external-node children as follows:
  - We find the right-most internal node v in the subtree rooted at the i-th child of z, noting that the children of node v are all external nodes.
  - We swap the entry (k_i, x_i) at z with the last entry of v. The key of this entry is the **predecessor** of k_i in the natural ordering of the keys of the tree.

Removal (cont’d)

- Once we ensure that the entry to be removed is stored at a node v with only external-node children, we simply remove the entry from v and remove the i-th external node of v.
- Removing an entry from a node v **preserves the depth property**, because we always remove an external node child from a node v with only external-node children.
- However, we might **violate the size property** at v.

Removal (cont’d)

- If v was previously a 2-node, then, after the removal, it becomes a 1-node with no entries.
- This type of violation of the size property is called an **underflow** node at v.
- To remedy an underflow, **we check whether an immediate sibling of $\boldsymbol{v}$ is a 3-node or a 4-node.** If we find such a sibling w, then we perform a **transfer** operation, in which we move a child of w to v, a key of w to the parent u of v and w, and a key of u to v.
- If v has only one sibling and this sibling is a 2-node, or if both immediate siblings of v are 2-nodes, then we perform a **fusion** operation, in which we merge v with a sibling, creating a new node v', and move a key from the parent u of v to v'.

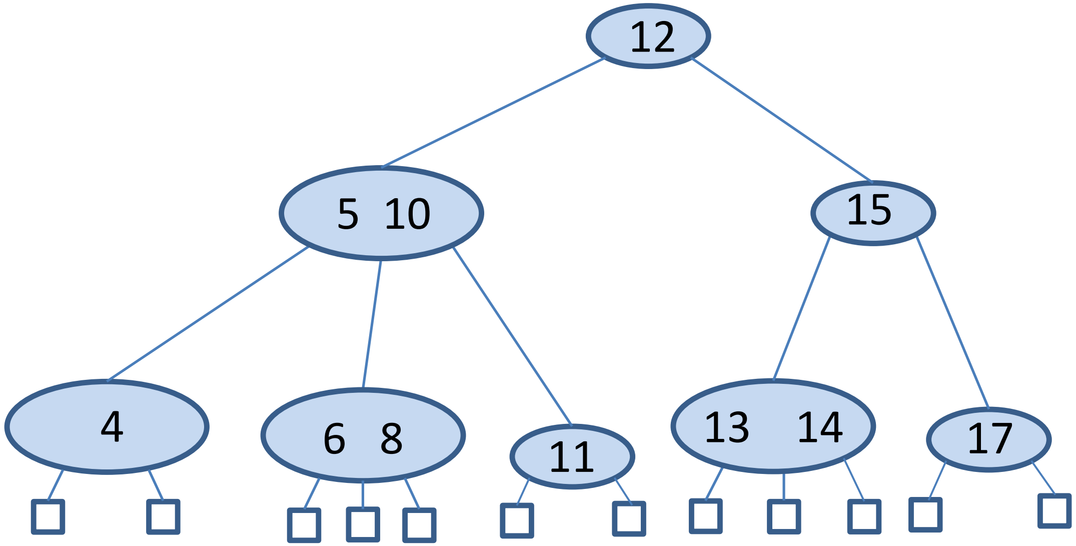
Removal (cont’d)

- A fusion operation at a node v may cause a new underflow to occur at the parent u of v, which in turn triggers a transfer or fusion at u.
- Hence, the number of fusion operations is bounded by the height of the tree which is $O(\log n)$.
- Therefore, a removal operation can take $\boldsymbol{O(log\, n)}$ **time in the worst case.**
- If an underflow propagates all the way up to the root, then the root is simply deleted.

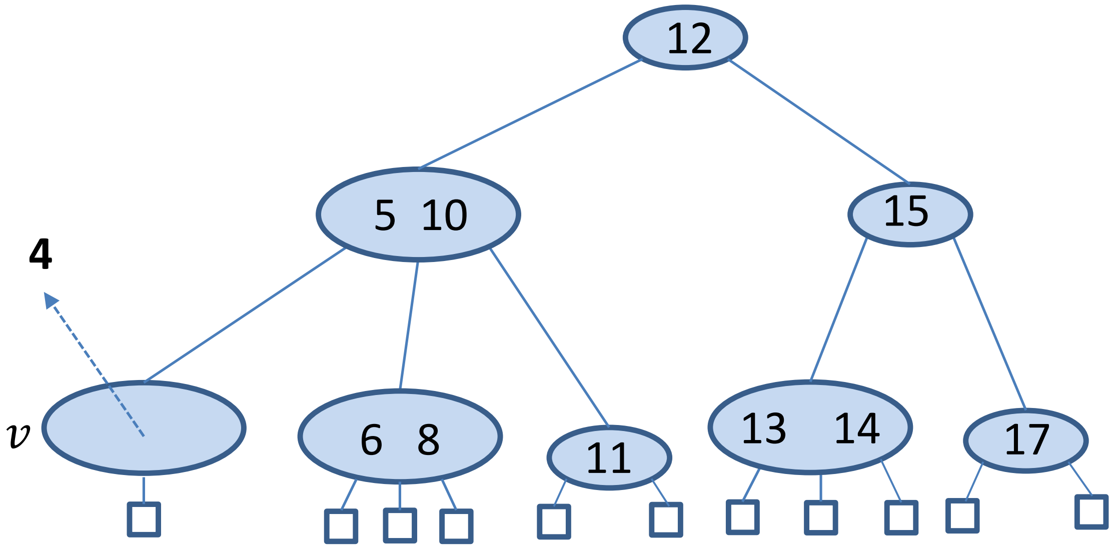
Examples

- Let us now see some examples of removal from a (2,4) tree.

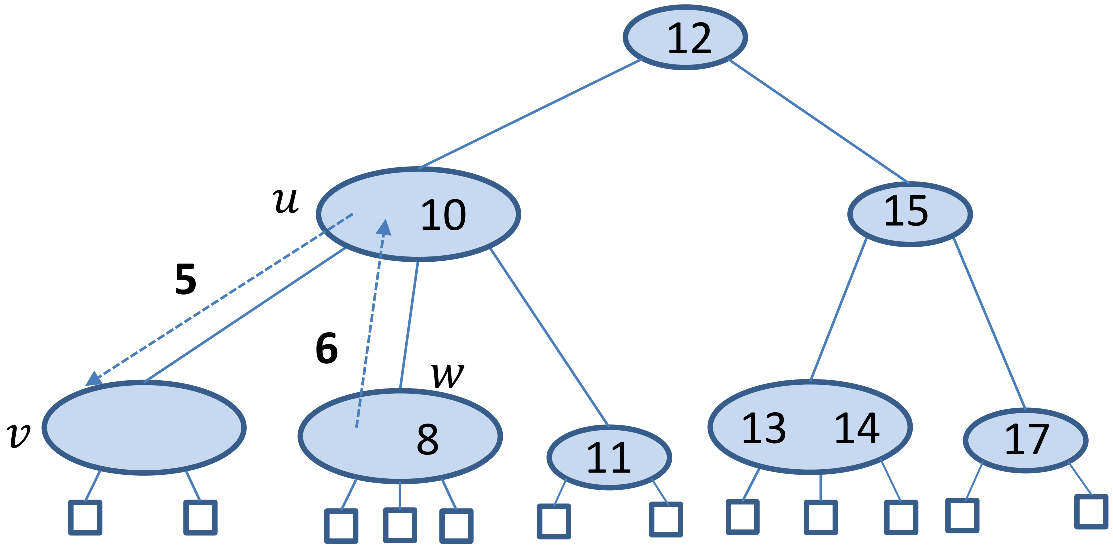
Initial Tree



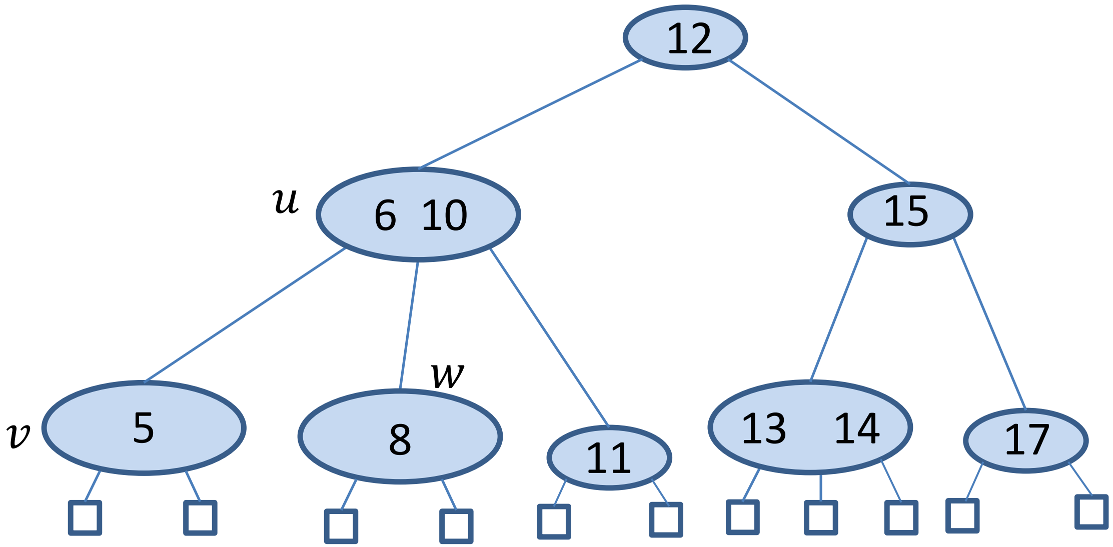
Remove 4



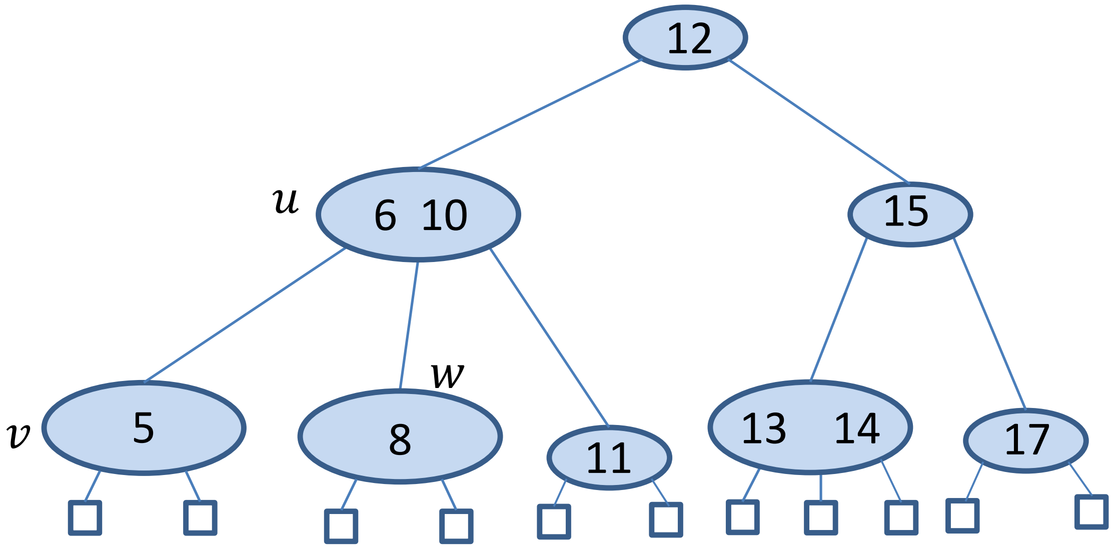
Transfer



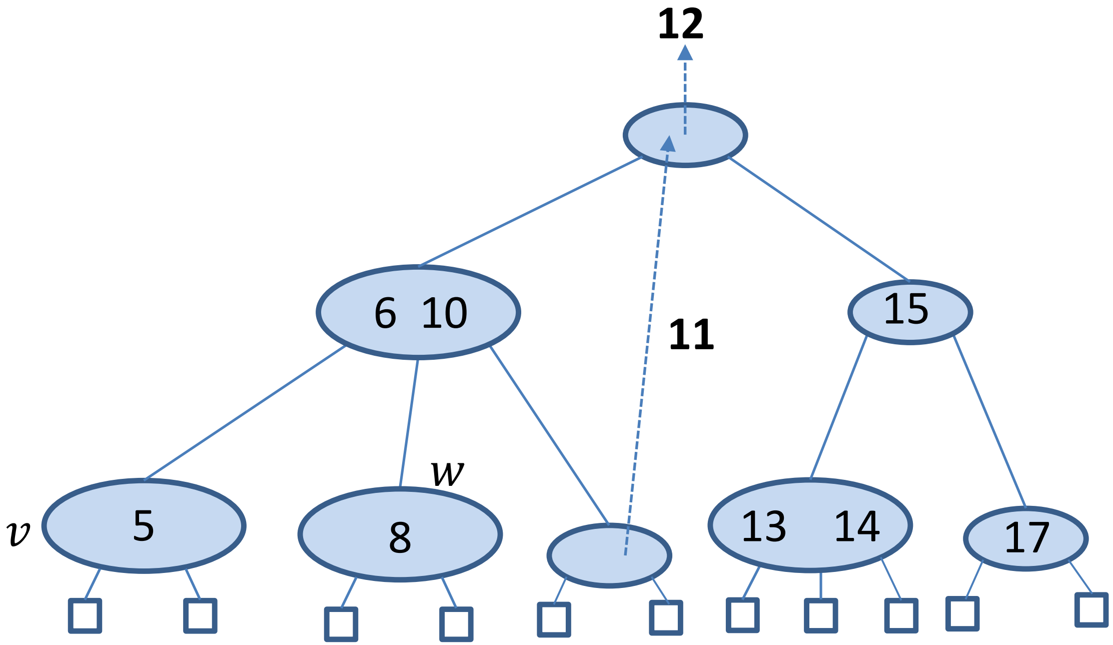
After the Transfer



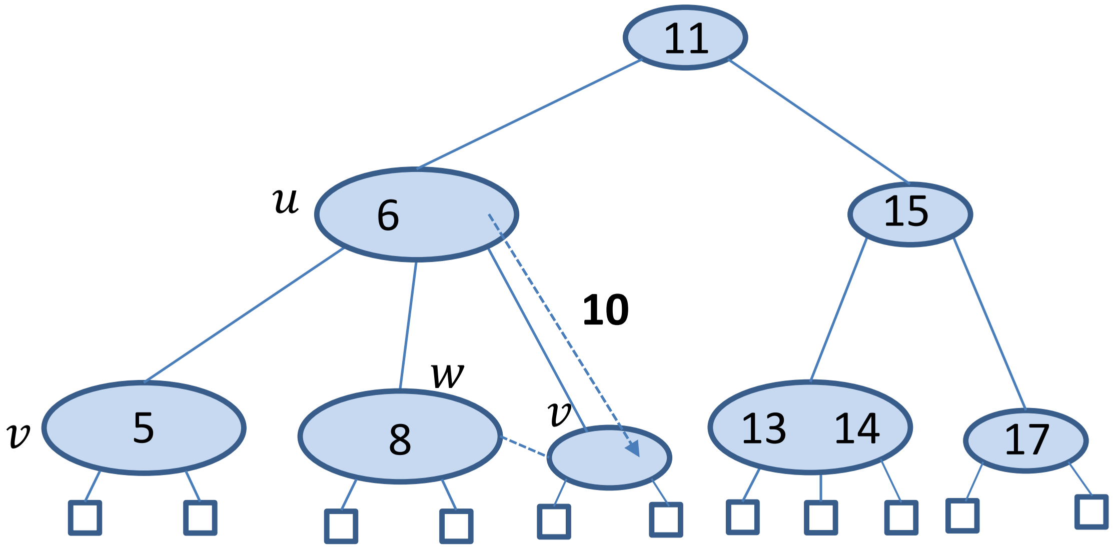
Remove 12



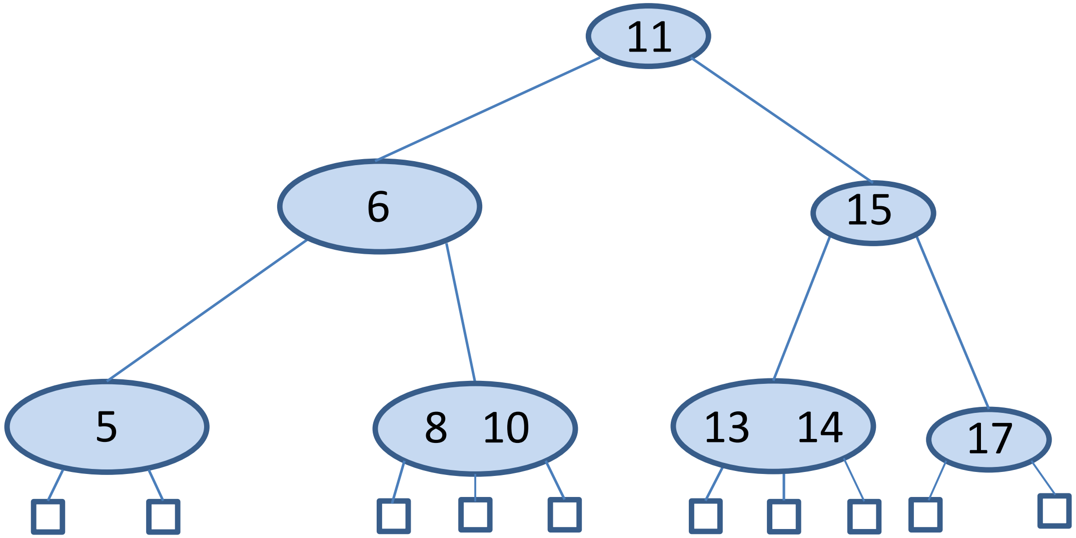
Remove 12



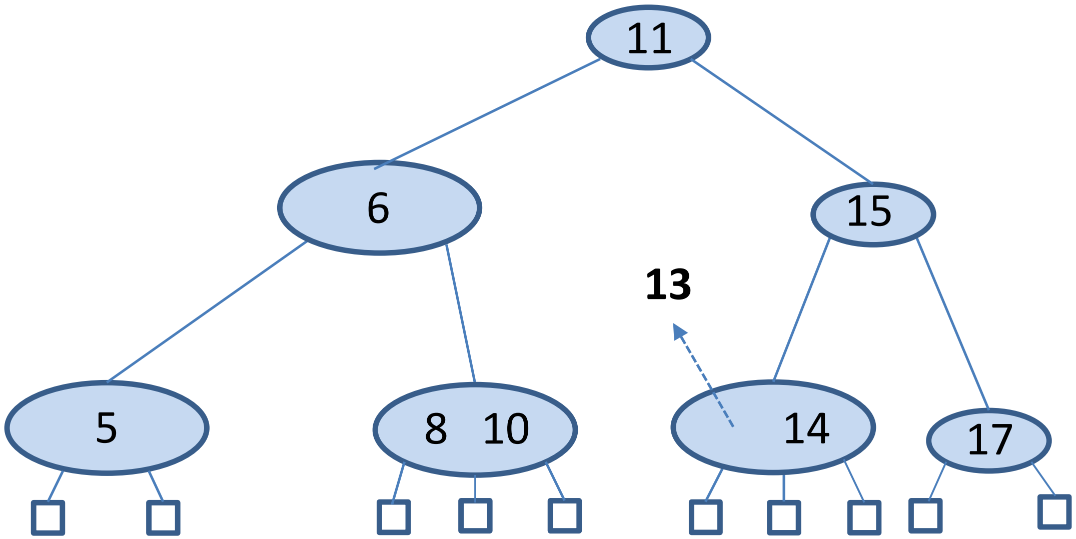
Fusion of w and v



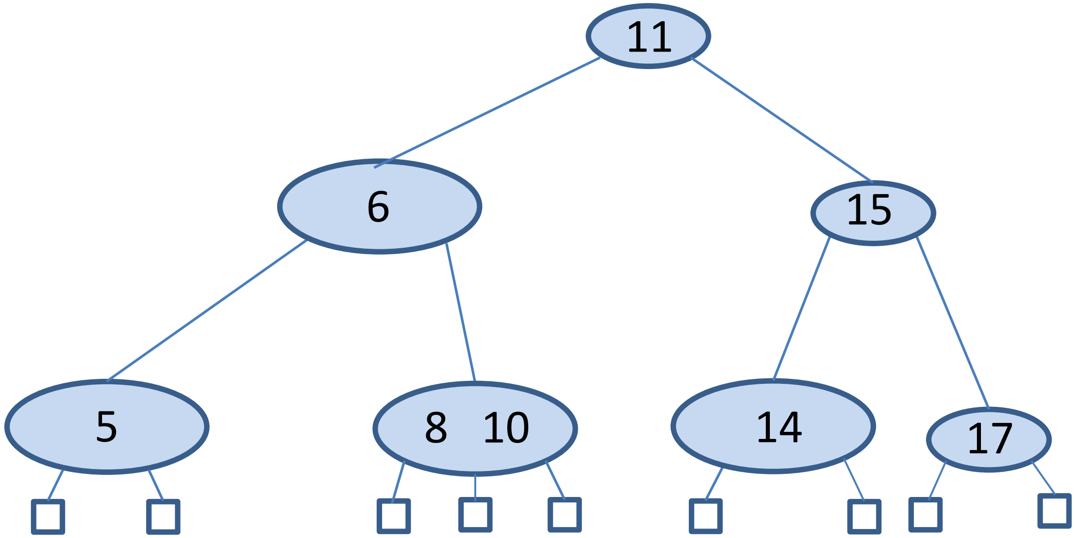
After the Fusion



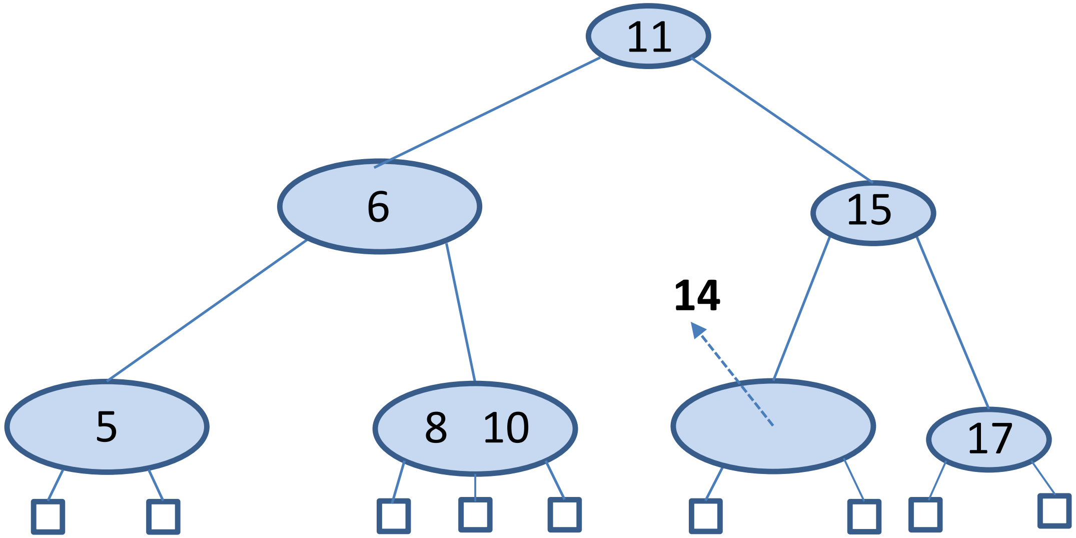
Remove 13



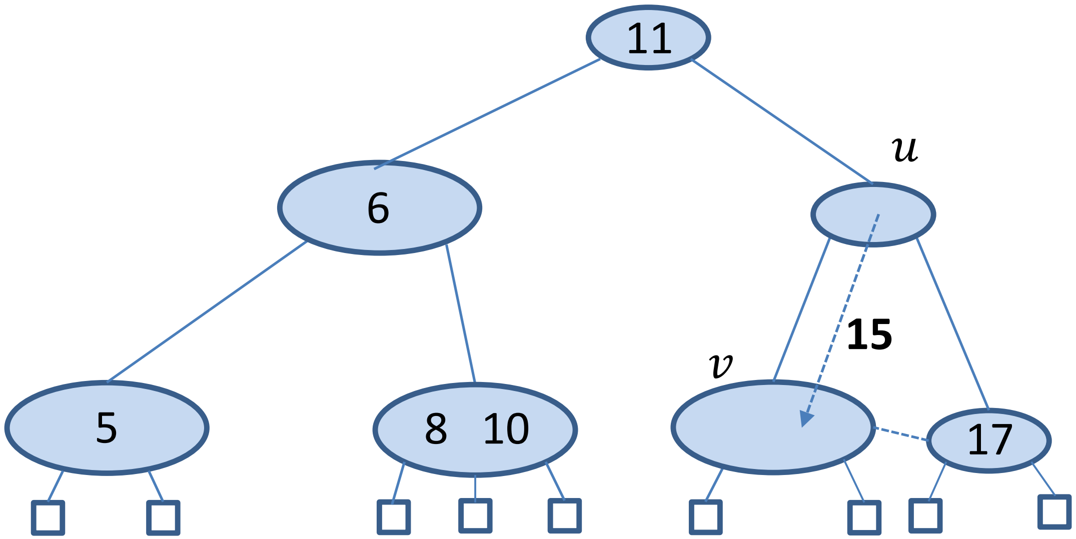
After the Removal of 13



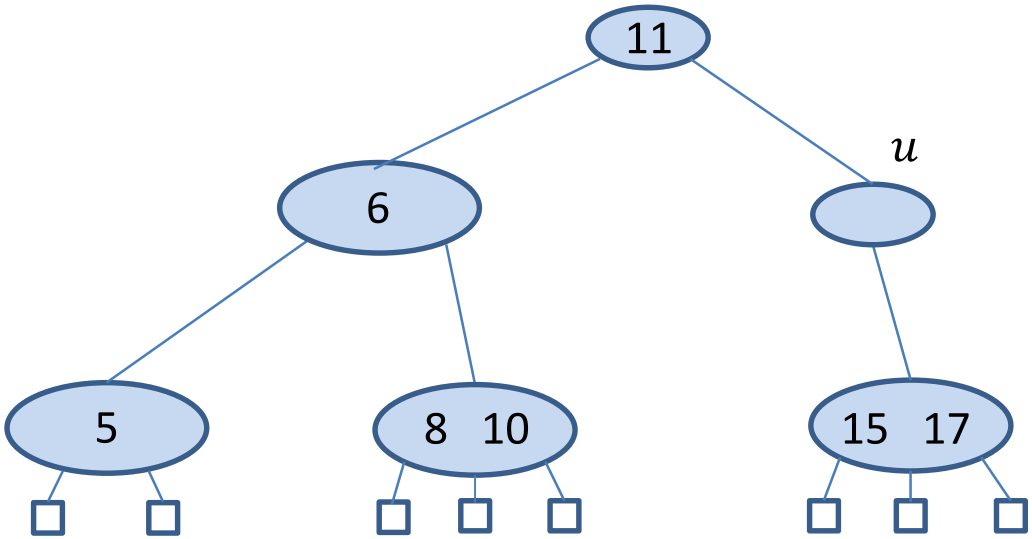
Remove 14 - Underflow



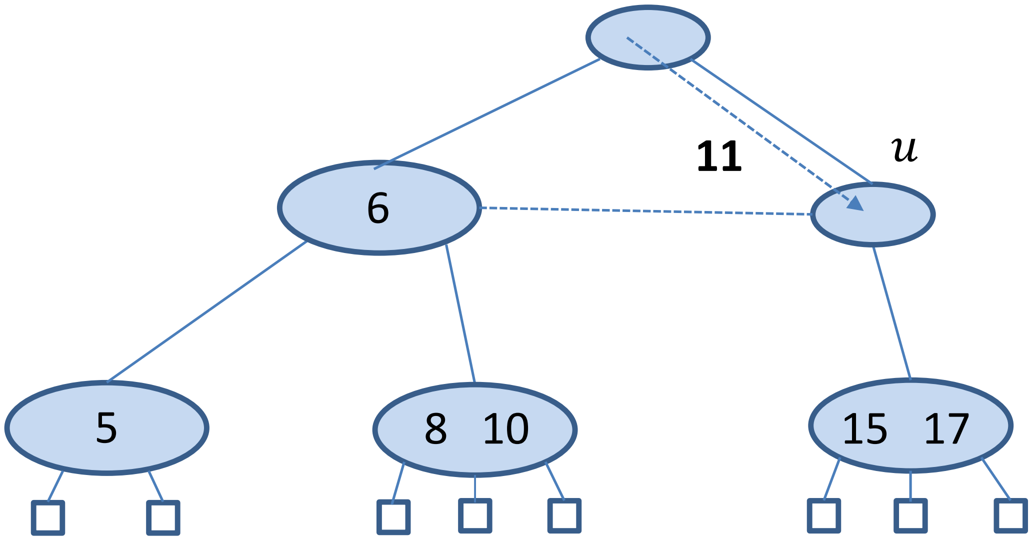
Fusion



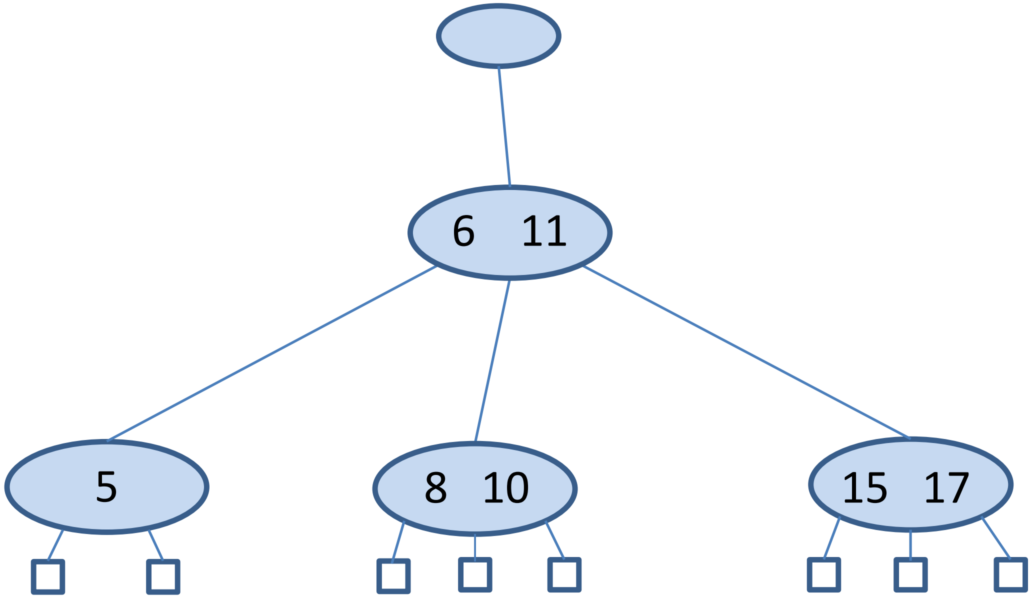
Underflow at u



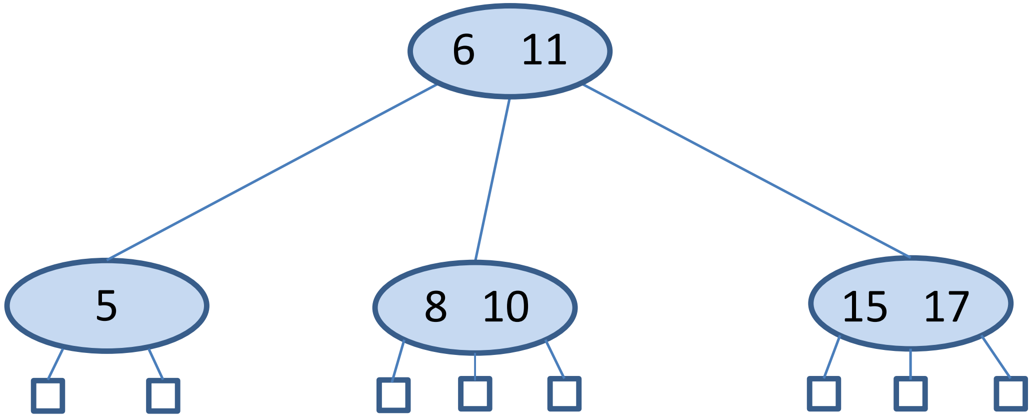
Fusion



Remove the Root



Final Tree



Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
  - Section 9.9
- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++.*
  - Section 10.4
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
  - Section 13.3