Multi-Way Search Trees

Manolis Koubarakis

Multi-Way Search Trees

- **Multi-way trees (δένδρα πολλών δρόμων)** are trees such that each internal node can store more than one key and can have more than two children.
- As in binary search trees, we assume that the **entries** we store in a multi-way search tree are **pairs of the form** $(\boldsymbol{k}, \boldsymbol{x})$ 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 (διατεταγμένο)** 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.

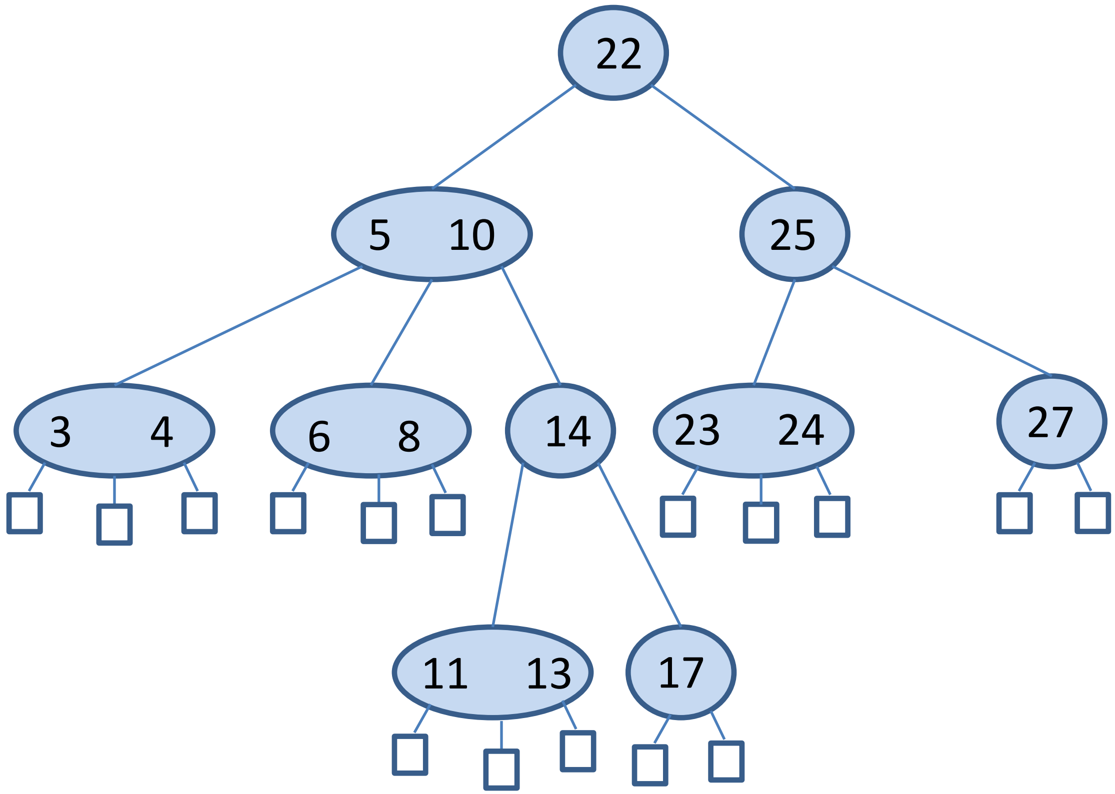
Definitions (cont'd)

- A **multi-way search tree (δένδρο αναζήτησης πολλών δρόμων)** 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 $k_1 < \cdots < k_{d-1}$.
  - 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 $k_{i-1} < k < k_i$.

Definitions (cont'd)

- 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 (δένδρο αναζήτησης** $\boldsymbol{m}$ **δρόμων**).
- 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., $m = 2$).
- Since no duplicates are allowed by the previous definition, multi-way search trees are an appropriate data structure for implementing **maps**.

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



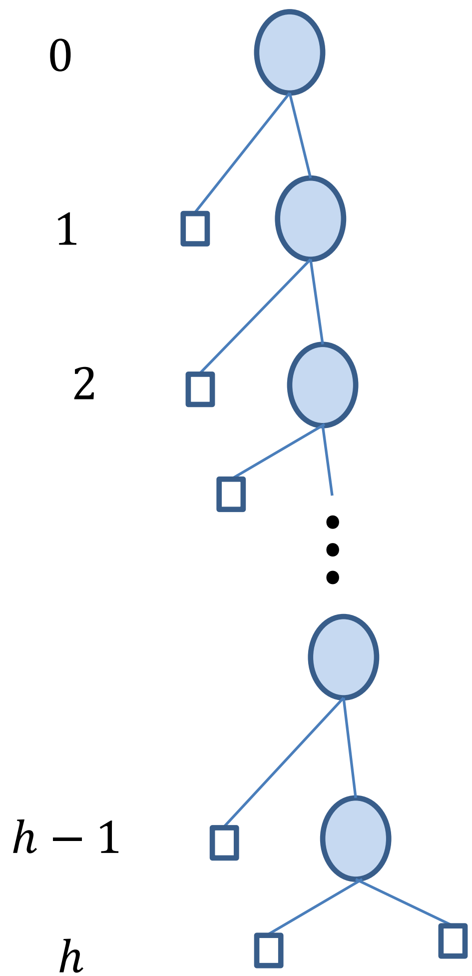
Proposition

- 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

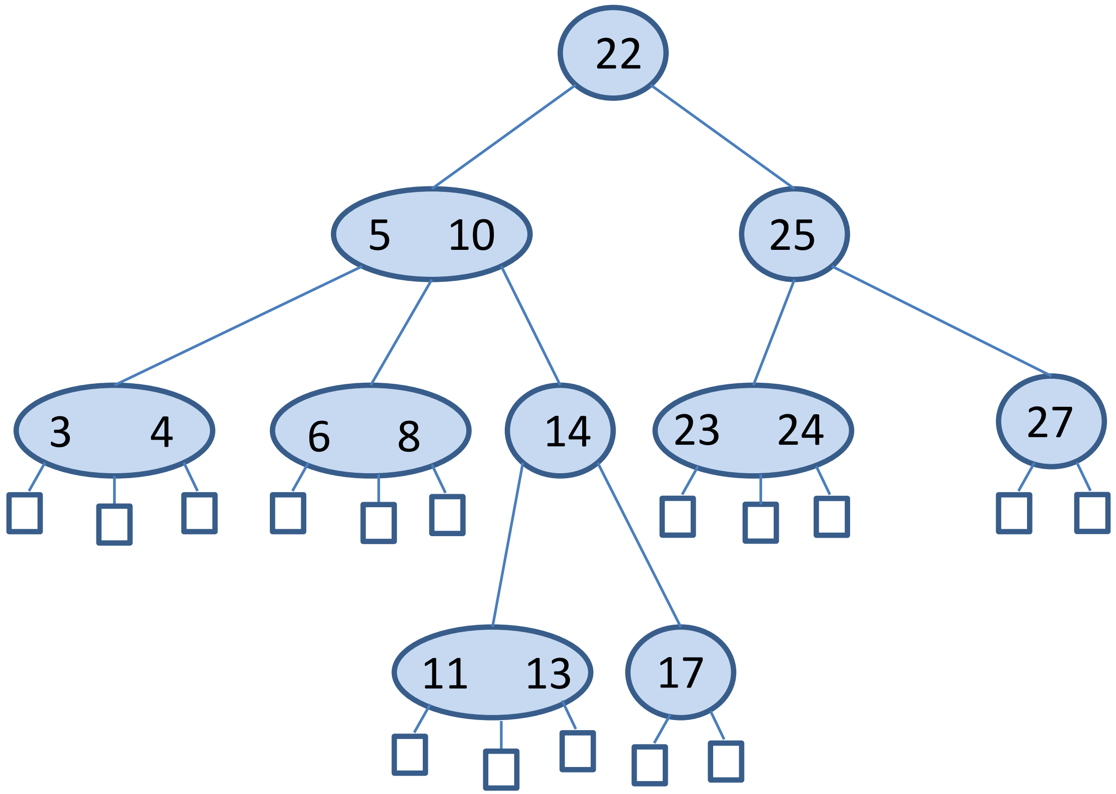
$$B = \sum_{i=1}^{m+1}(p_i + 1) = m + 1 + (\sum_{i=1}^{m+1} p_i) = A + 1$$

external nodes.

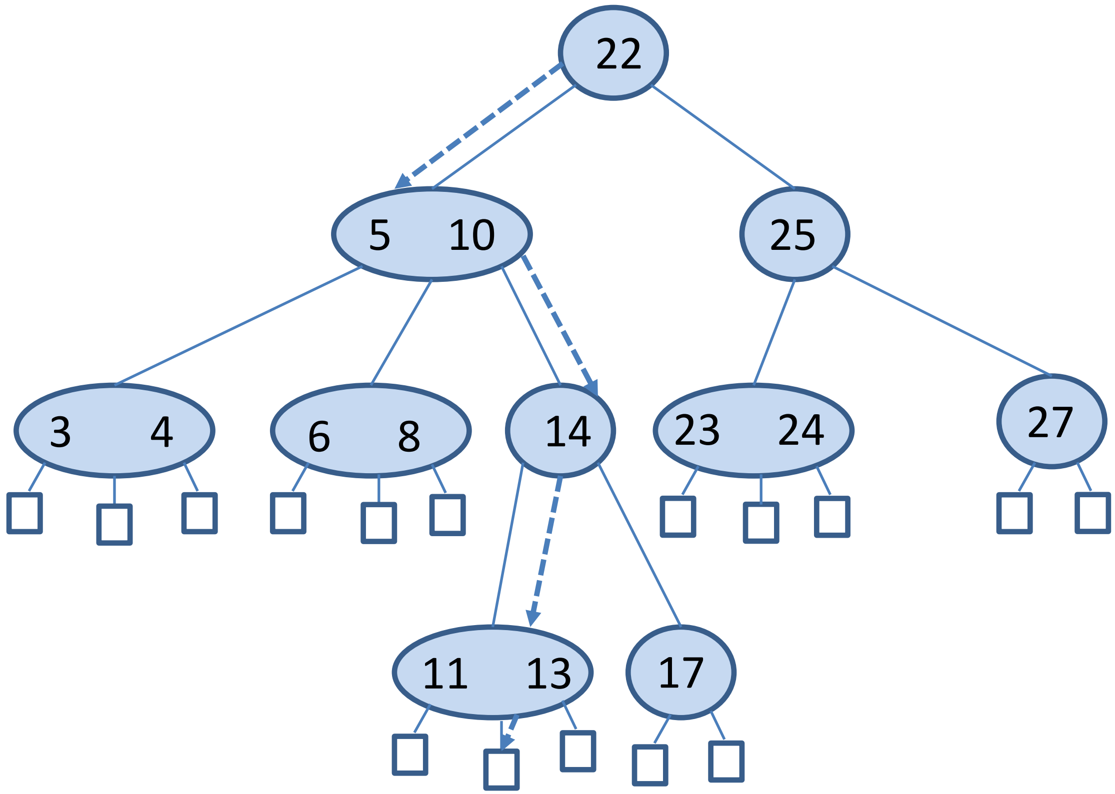
Searching in a Multi-Way Search Tree

- 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

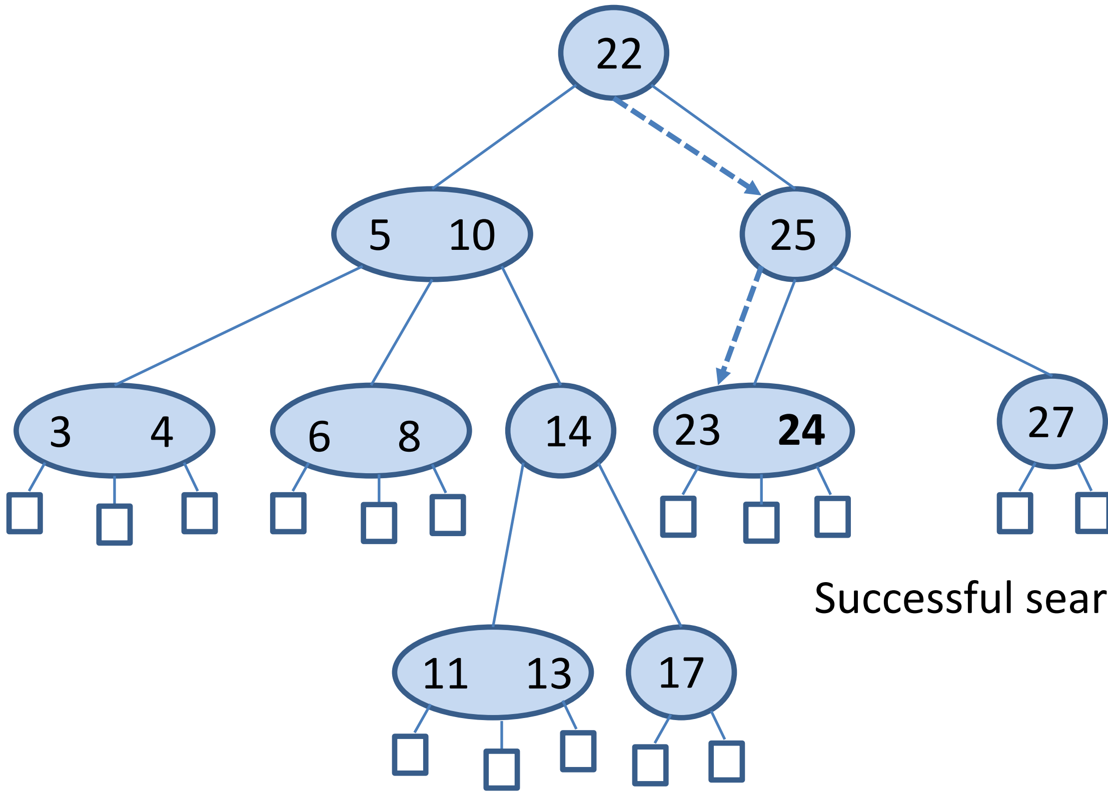


Search for Key 12



Unsuccessful search

Search for Key 24

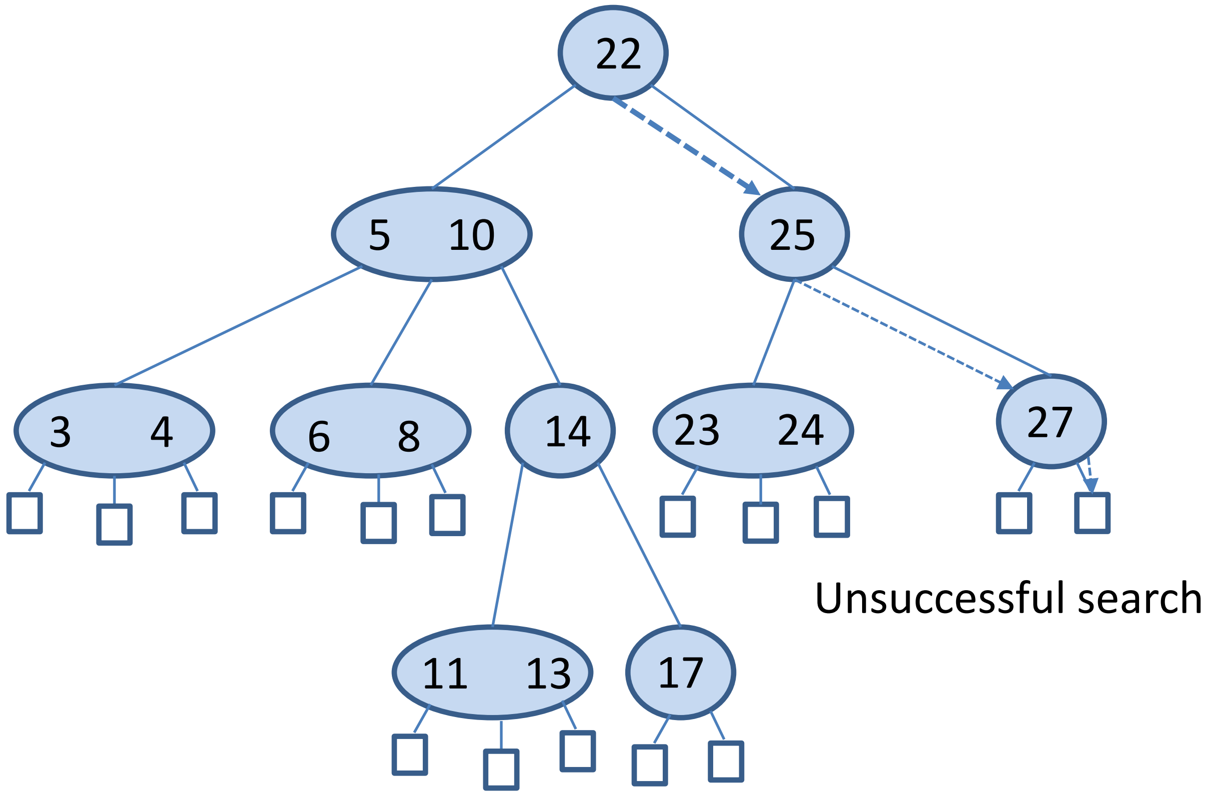


Successful search

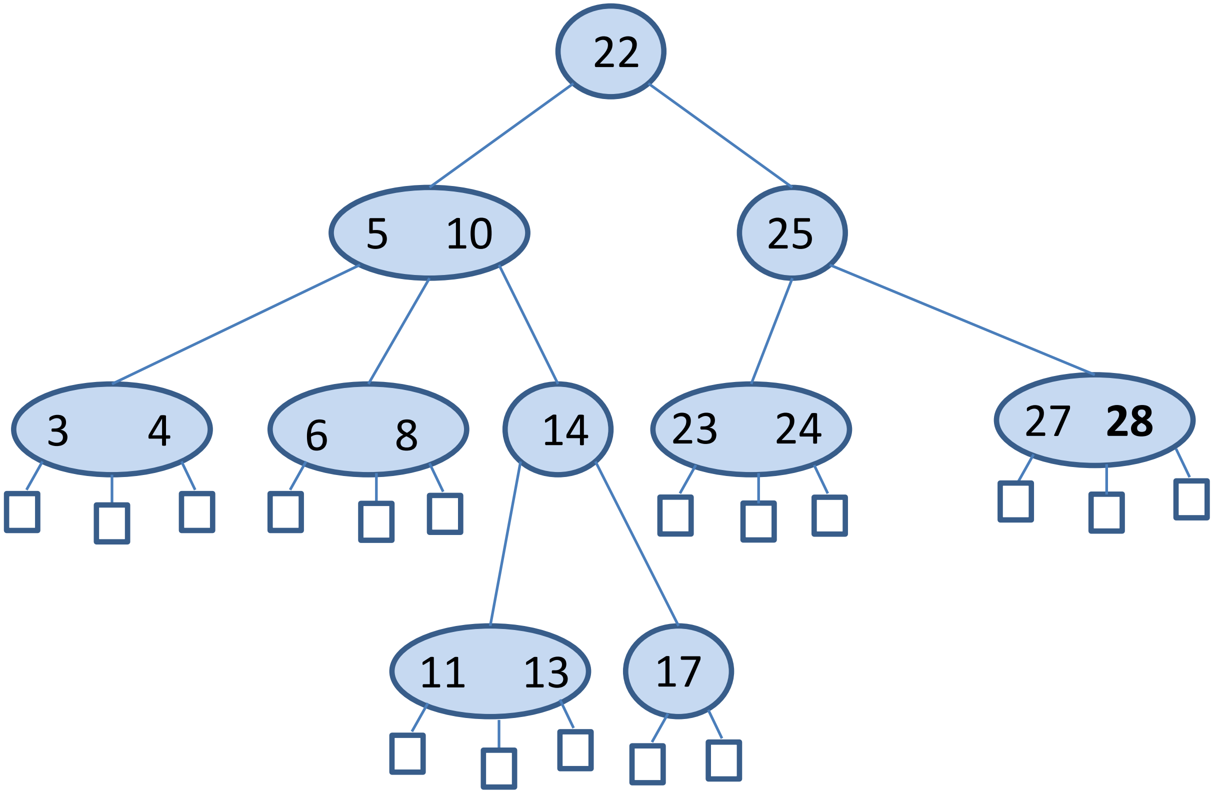
Insertion in a Multi-Way Search Tree

- 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 (no duplicates are allowed).
- 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.

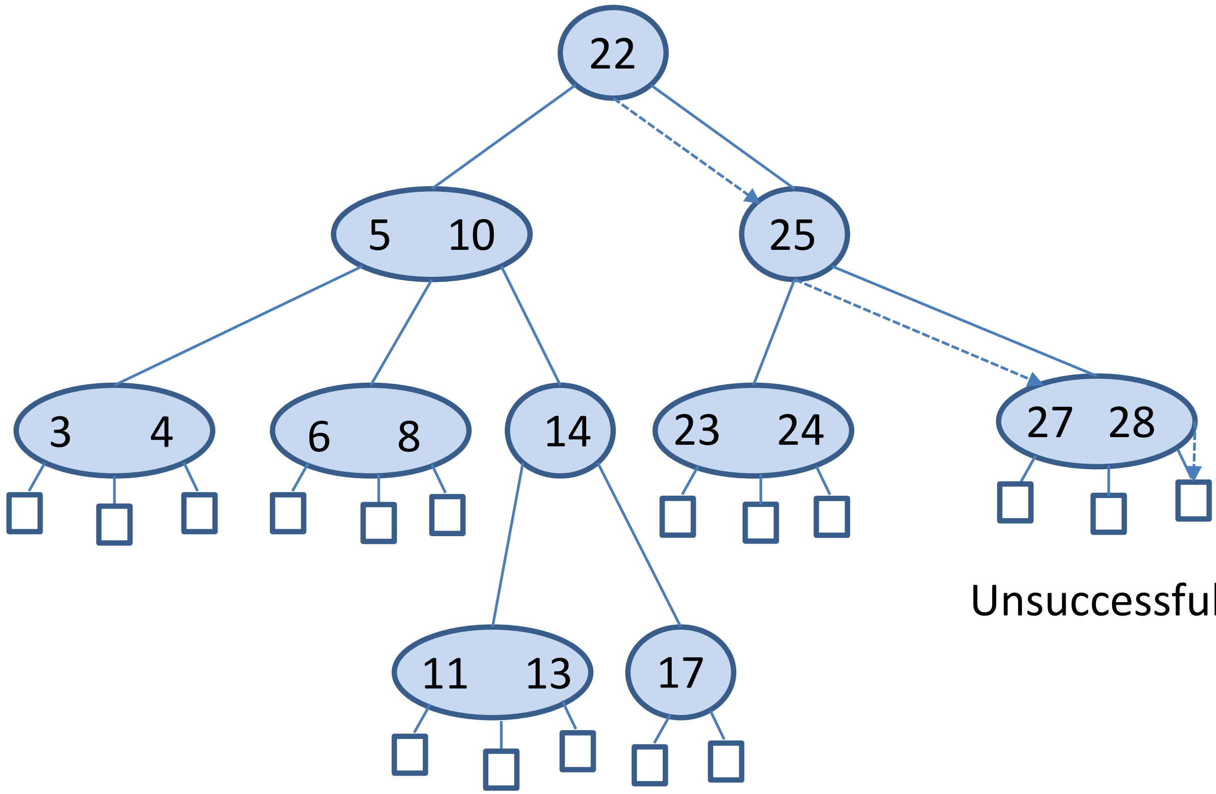
Insert Key 28 ($m = 3$)



Key 28 Inserted

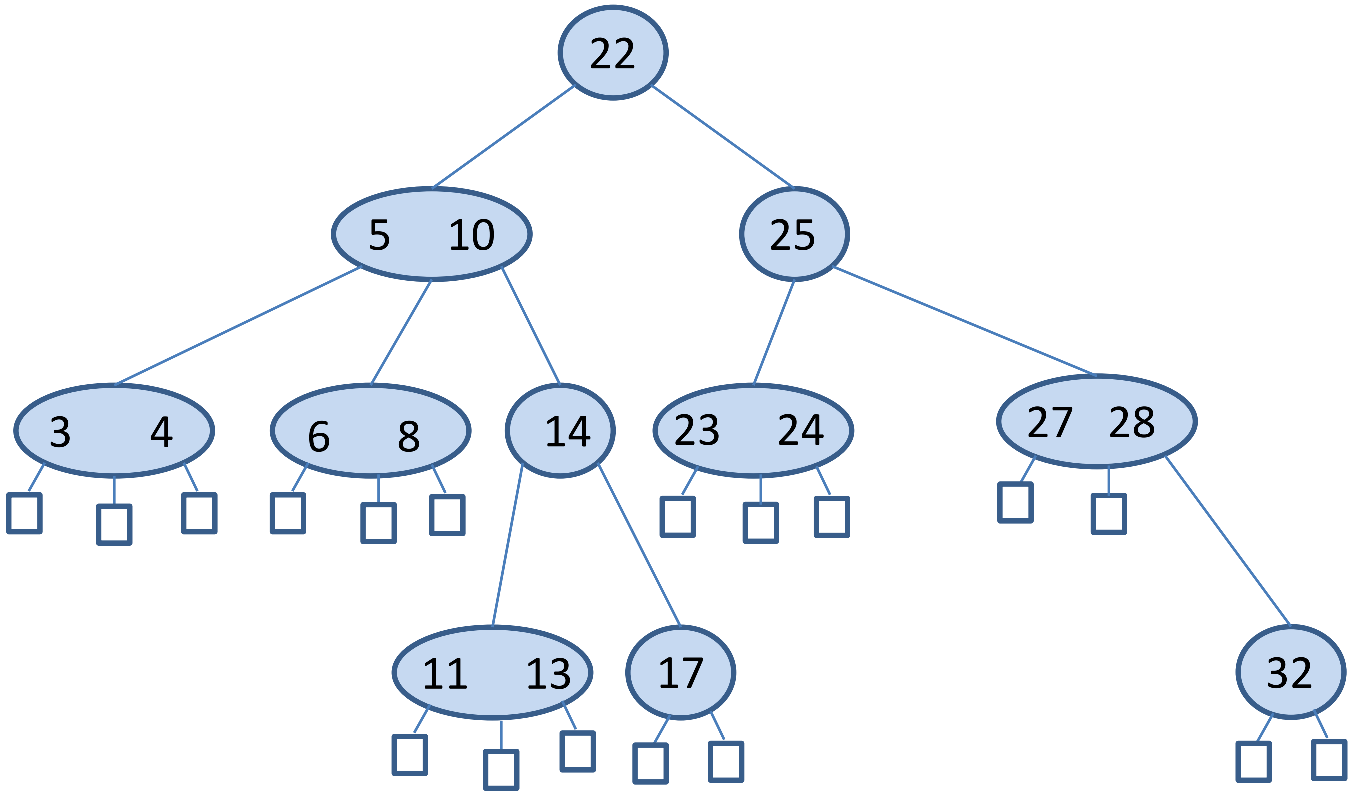


Insert Key 32

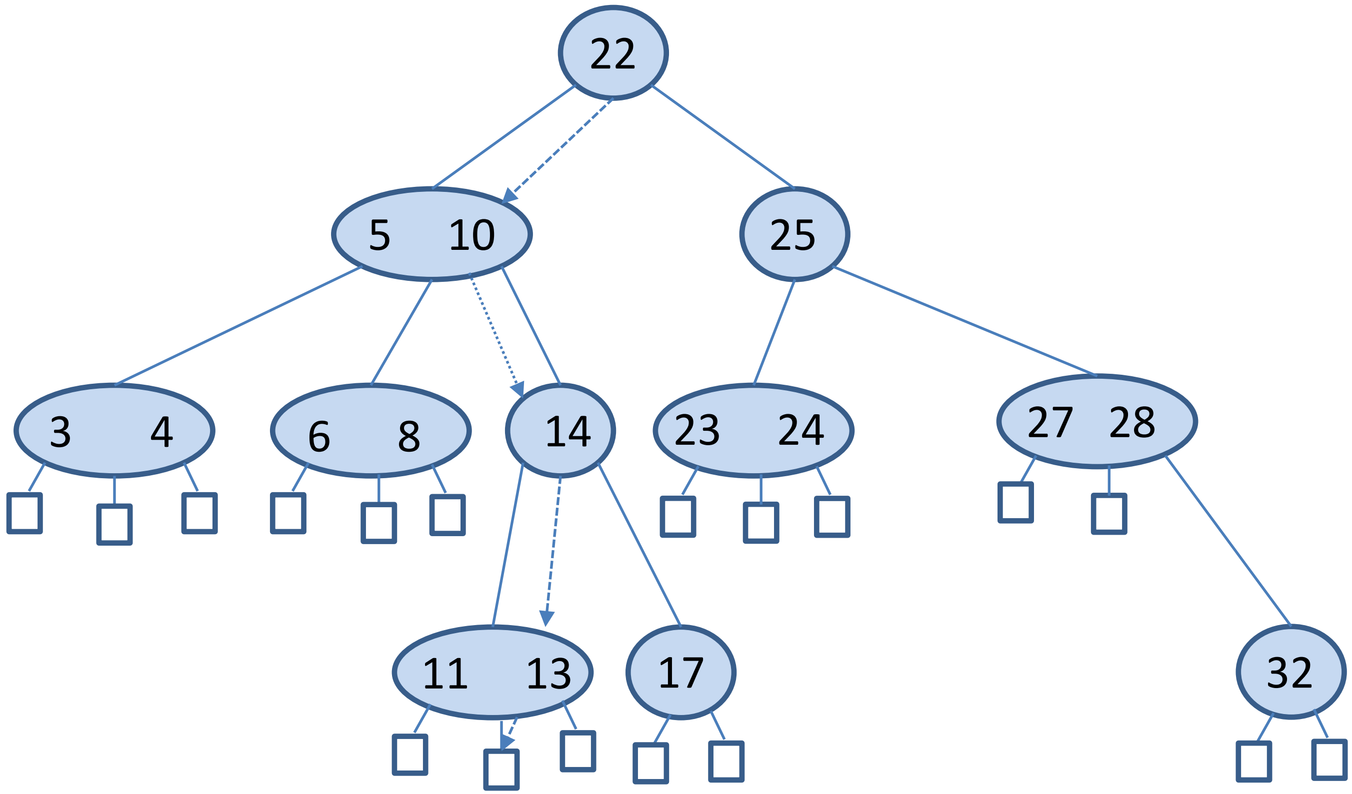


Unsuccessful search

Key 32 Inserted

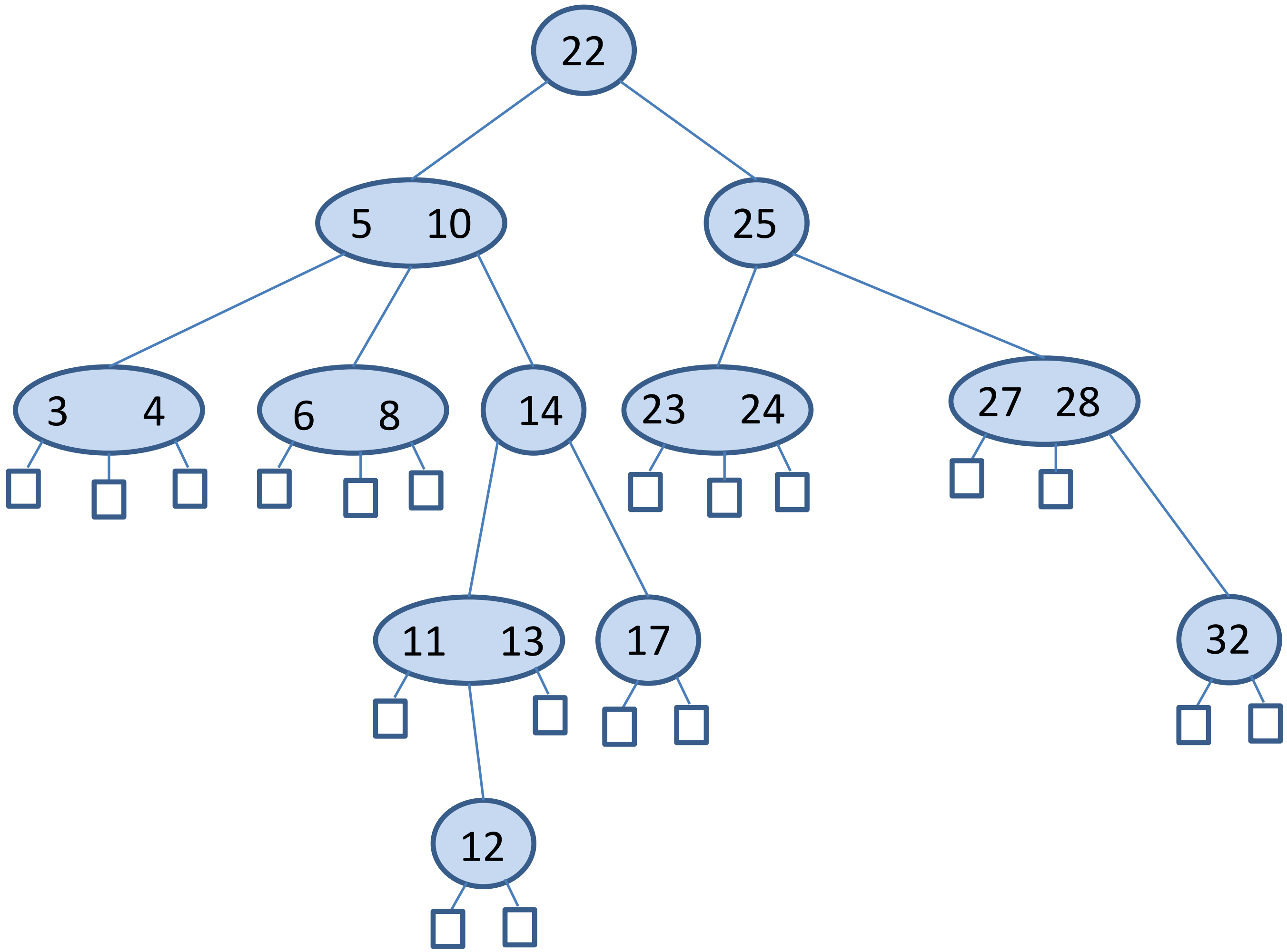


Insert Key 12



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 (ισορροπημένο ή ισοζυγισμένο δένδρο αναζήτησης).**

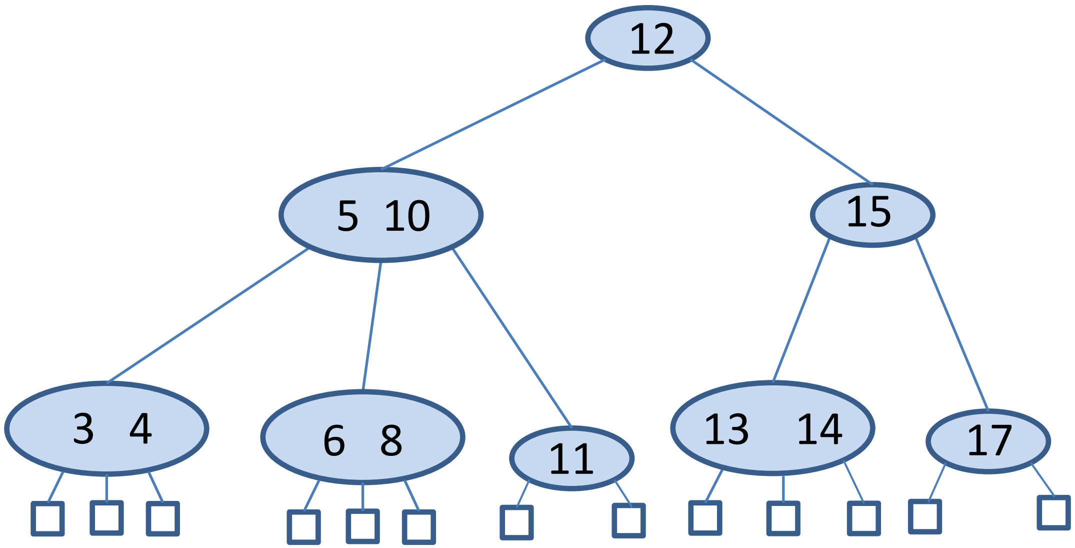
Balanced Multi-way Search Trees

- We will now study the following kinds of balanced multi-way search trees:
  - **(2,4) trees** (this lecture)
  - **Red-black trees** (forthcoming lectures)
  - **(a,b) trees** (forthcoming lectures)
  - **B-trees** (forthcoming lectures)

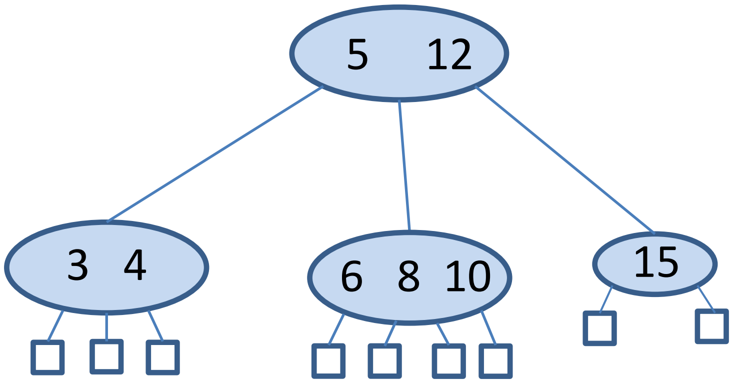
(2,4) Trees

- 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).

Example



Example



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.

Search in (2,4) Trees

- The algorithm for searching for an entry with key k in a (2,4) tree is the same as the algorithm we presented for multi-way trees.

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.

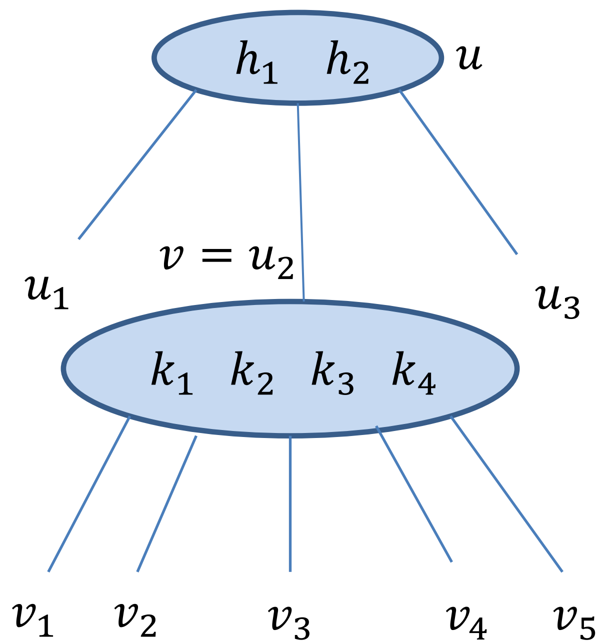
Insertion (cont'd)

- 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 no longer be a (2,4) tree.
- This type of violation of the size property is called an **overflow (υπερχείλιση)** node at node v, and it must be resolved in order to restore the properties of a (2,4) tree.

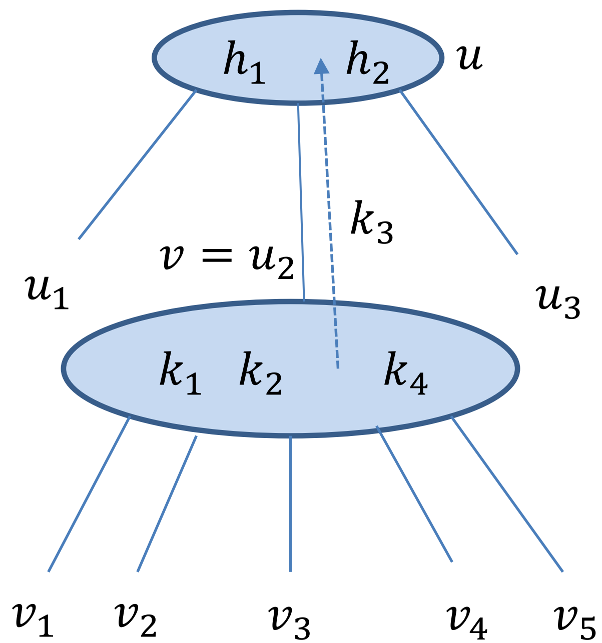
Dealing with Overflow Nodes

- 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 (διαίρεση) 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.

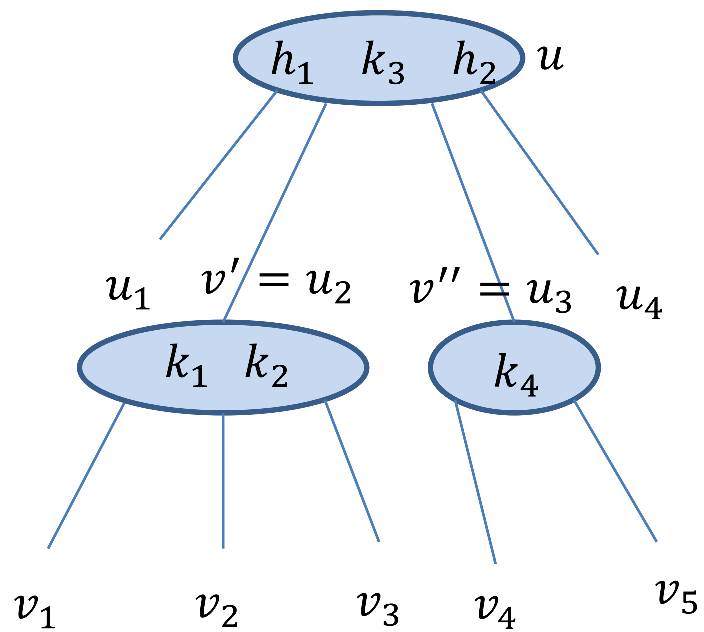
Overflow at a 5-node



The third key of v inserted into the parent node u



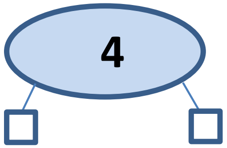
Node v replaced with a 3-node v' and a 2-node v''



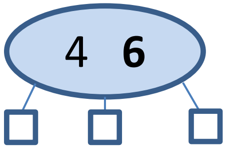
Example

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

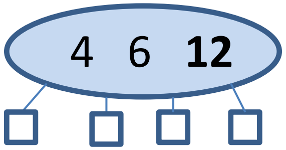
Insert 4



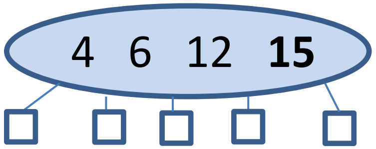
Insert 6



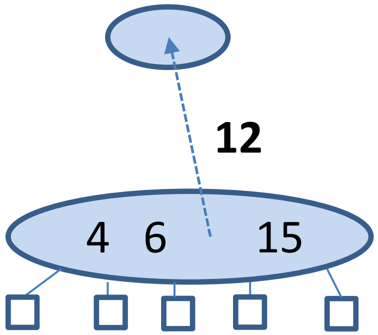
Insert 12



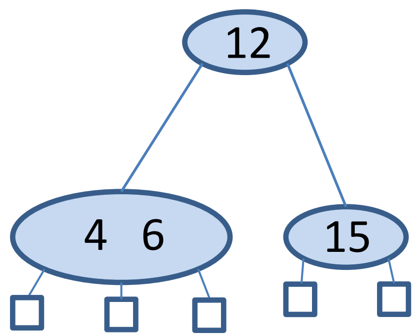
Insert 15 - Overflow



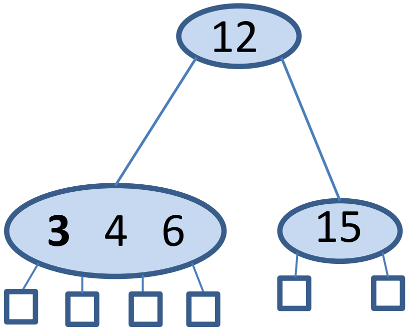
Creation of New Root Node



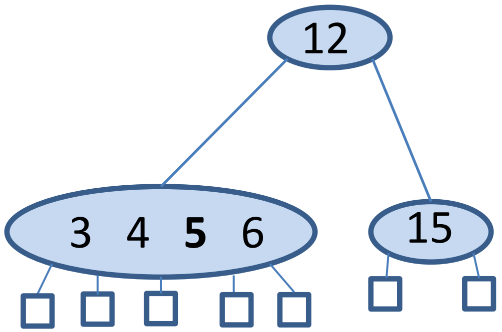
Split



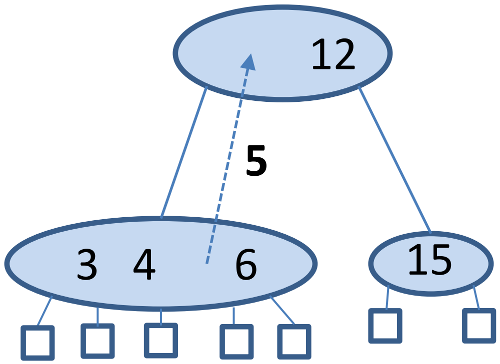
Insert 3



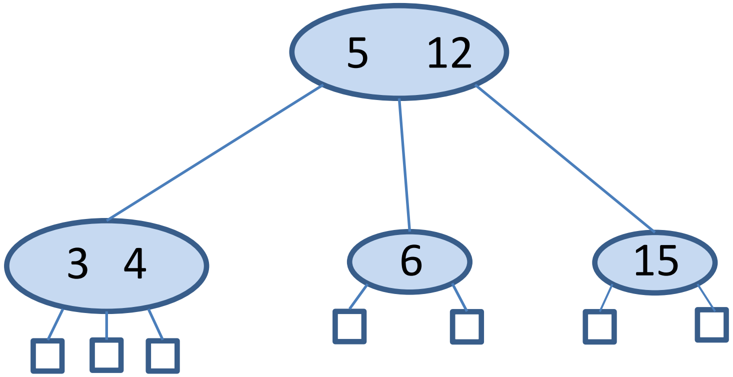
Insert 5 - Overflow



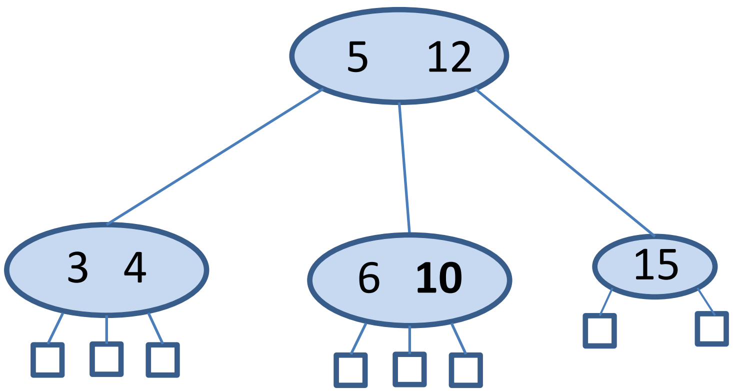
5 is Sent to the Parent Node



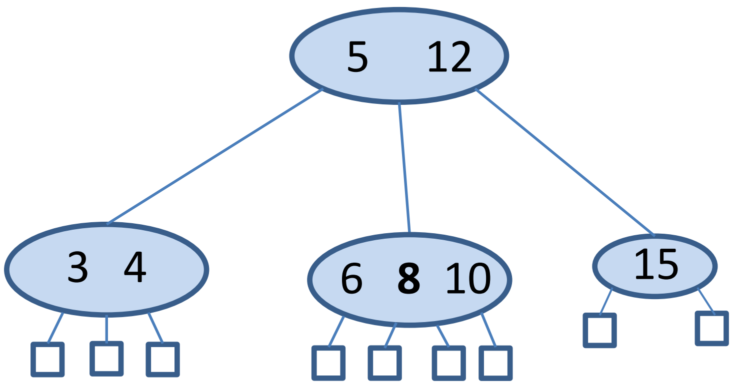
Split



Insert 10



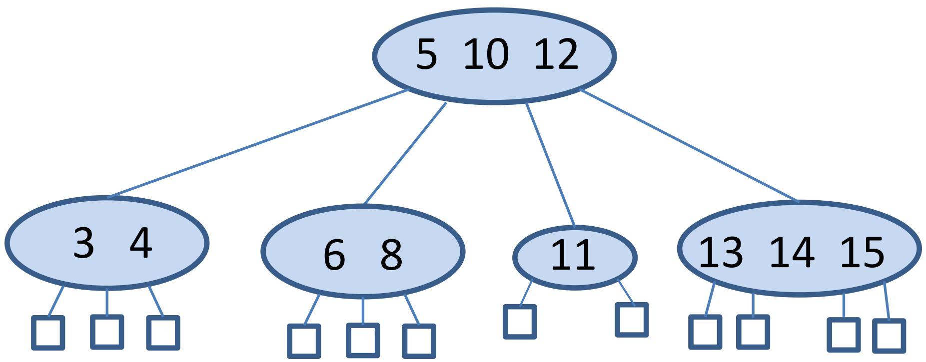
Insert 8



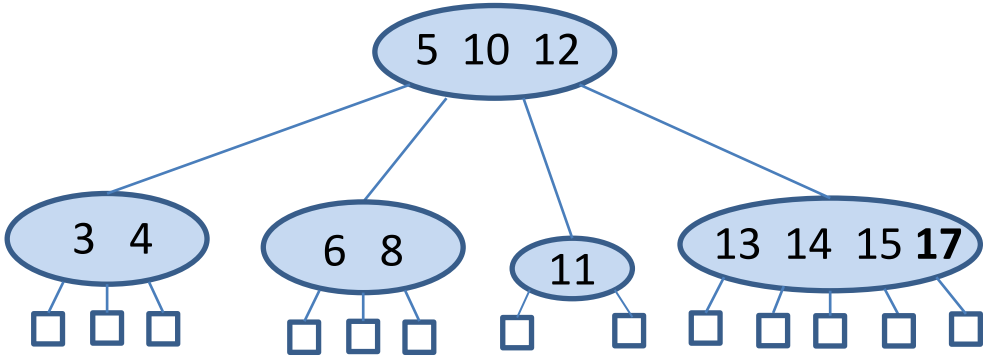
Insertion (cont'd)

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

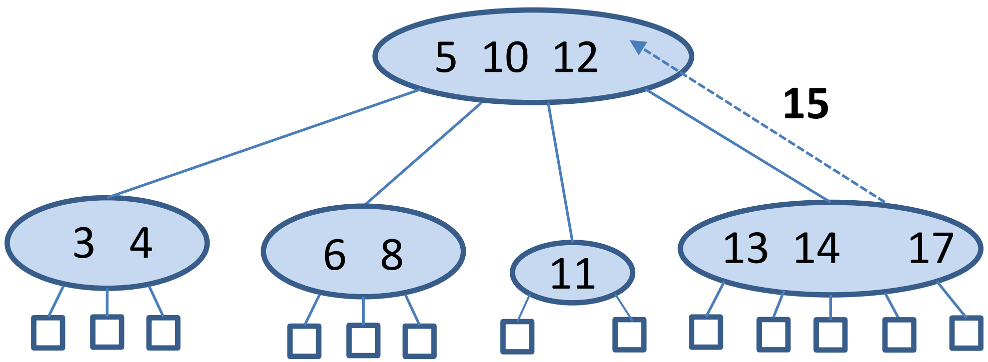
Initial Tree



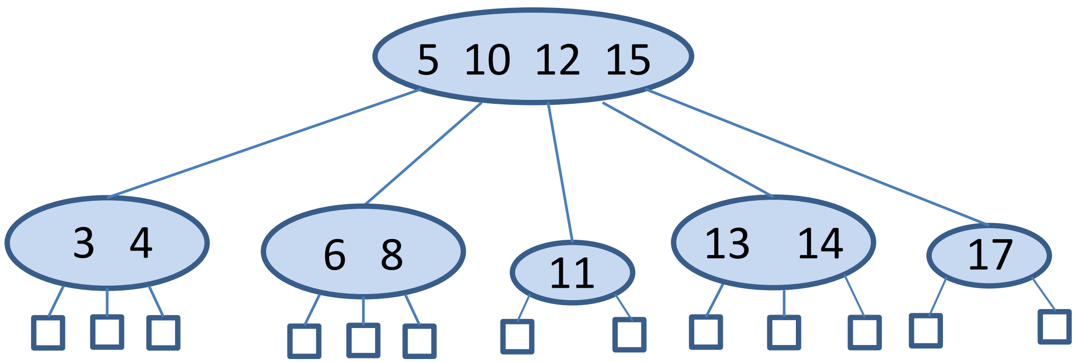
Insert 17 - Overflow



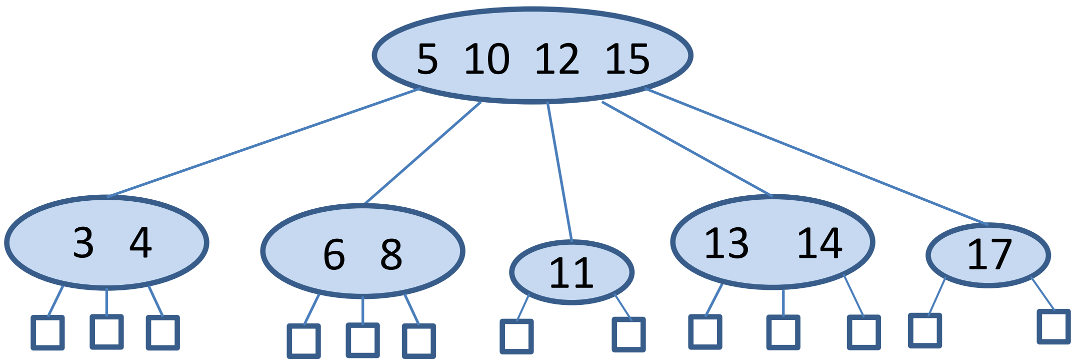
15 is Sent to the Parent Node



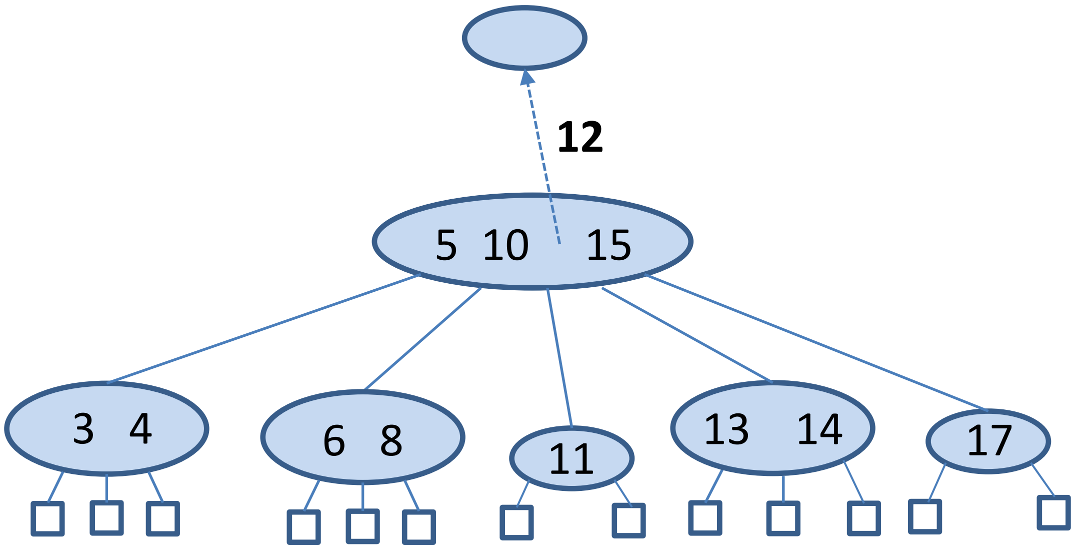
Split



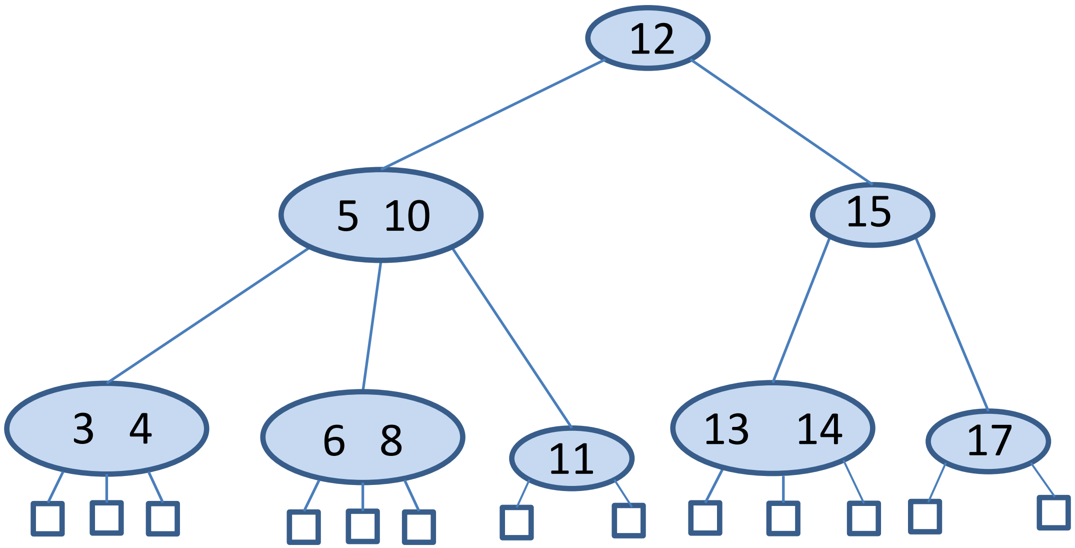
Overflow at the Root



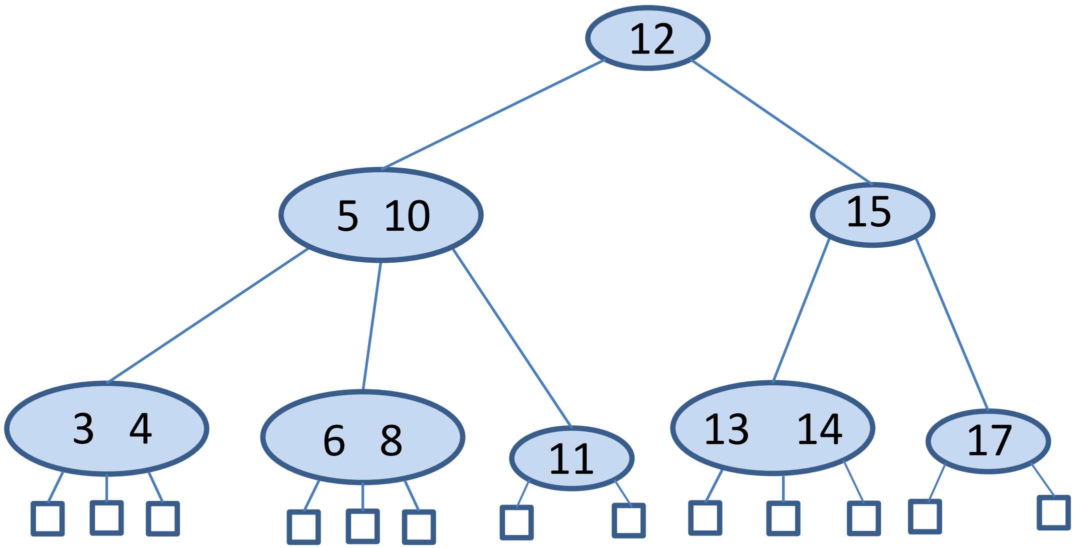
Creation of New Root



Split



Final Tree



Complexity Analysis of Insertion

- The insertion algorithm consists of a **downward phase** where we find the node of the tree where the new entry will be inserted and, possibly, of an **upward phase** consisting of split operations.
- The downward phase has time complexity $O(h) = O(\log n)$.
- **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.
- The first step of the algorithm is to **search** for key k as we have done for multi-way search trees.
- If the entry to be removed is found to be the i-th entry (k_i, x_i) (where $k_i = k$) in a node v with **only external-node children**, we simply remove the entry from v and remove the i-th external node of v.

Removal (cont'd)

- 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)

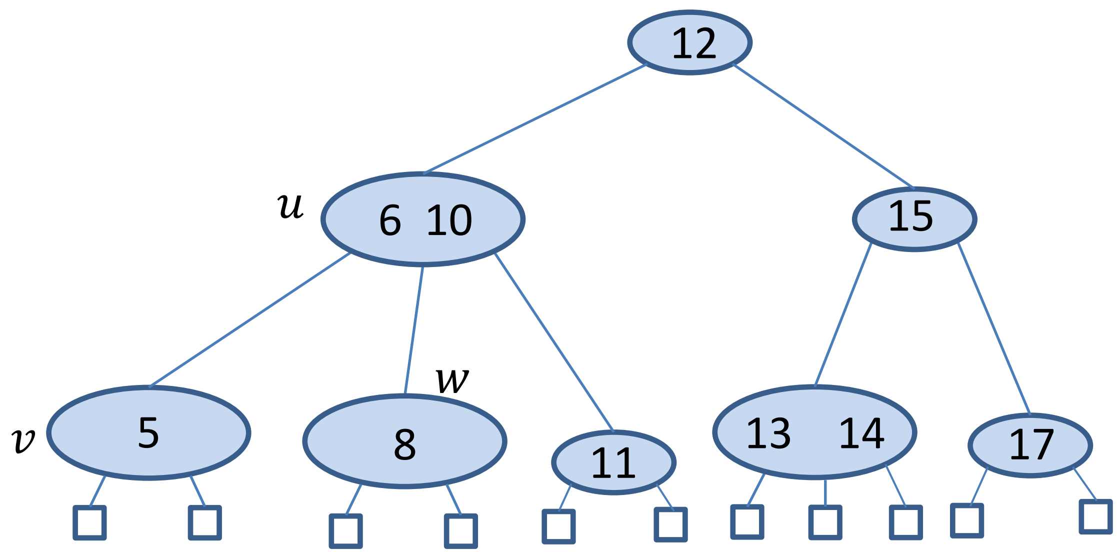
- If v was previously a 2-node, then, after the removal, it becomes a **1-node with no entries**.
- This type of violation of the size property is called an **underflow** node at v.
- To remedy an underflow, **we check whether an immediate sibling of v is a 3-node or a 4-node.** If we find such a sibling 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'.
- If an underflow propagates all the way up to the root, then the root is deleted.

Removal (cont'd)

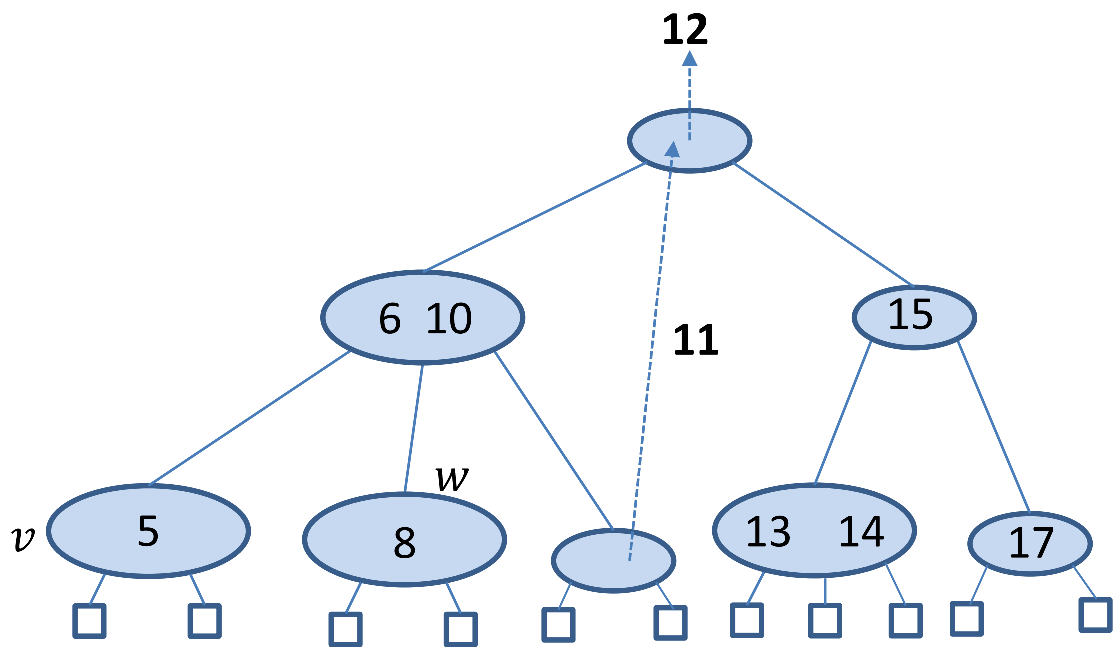
- Suppose now that the entry we wish to remove is stored in the i-th entry (k_i, x_i) at a node z that has **only internal nodes as children.**
- Removing such an entry **can always be reduced** to the case where the entry to be removed is stored at a node v whose children are external nodes.
- In this 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 which is deleted from v. The key of this last entry is the **predecessor** of k_i in the natural ordering of the keys of the tree.

- Then, the algorithm proceeds as in the previous case by doing transfer and fusion operations if necessary.

Example: Remove 12



Key 11 is Moved to the Place of 12

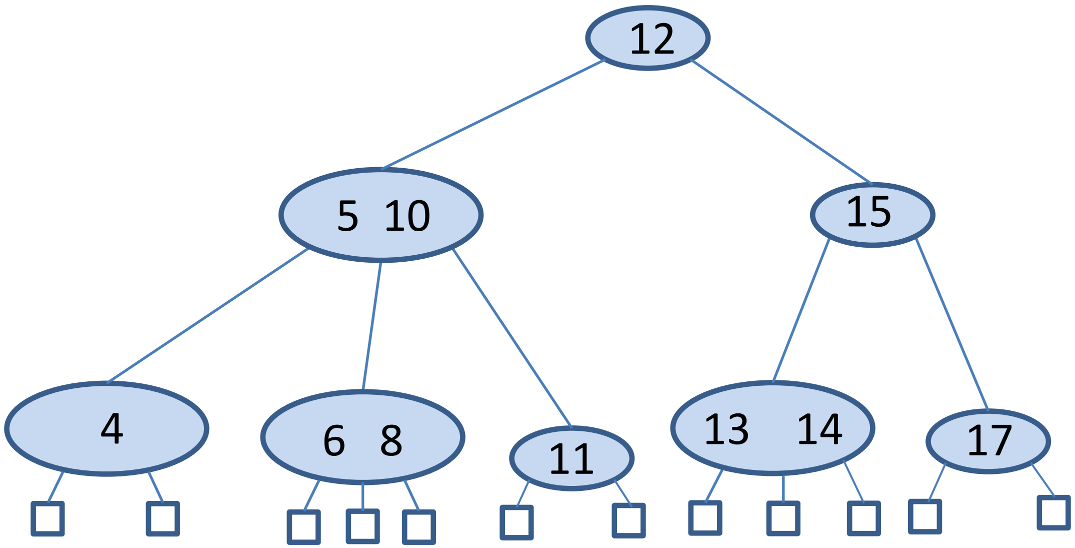


- We could have also used 13 instead of 11 (namely, the **successor** of 12 in the natural ordering of the keys).

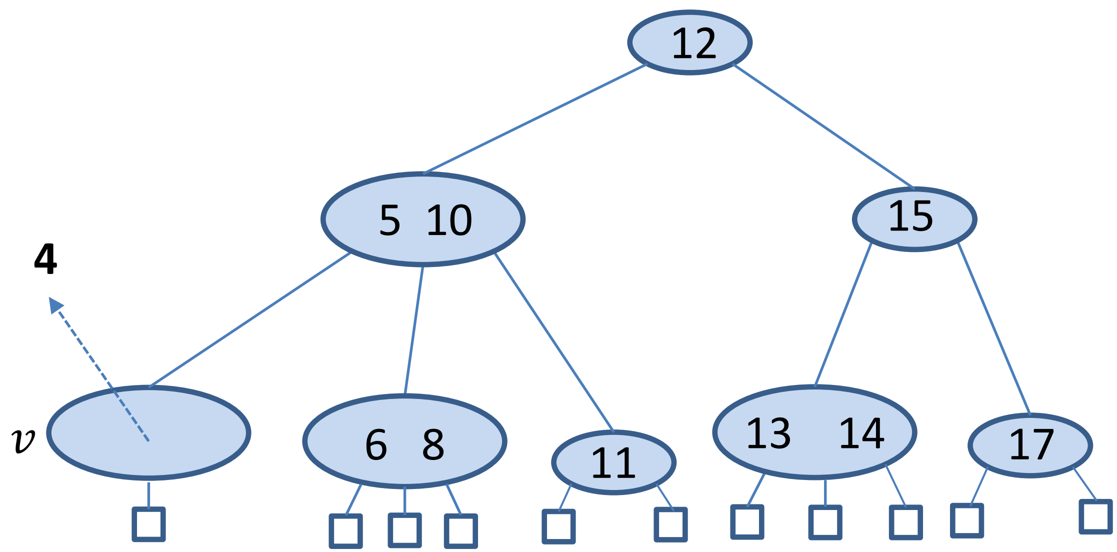
Examples

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

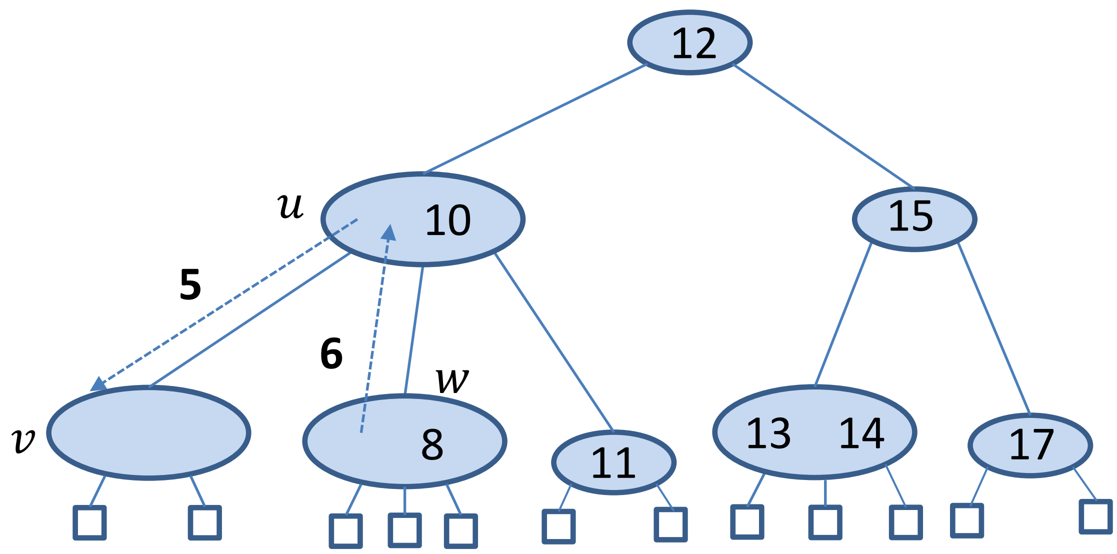
Initial Tree



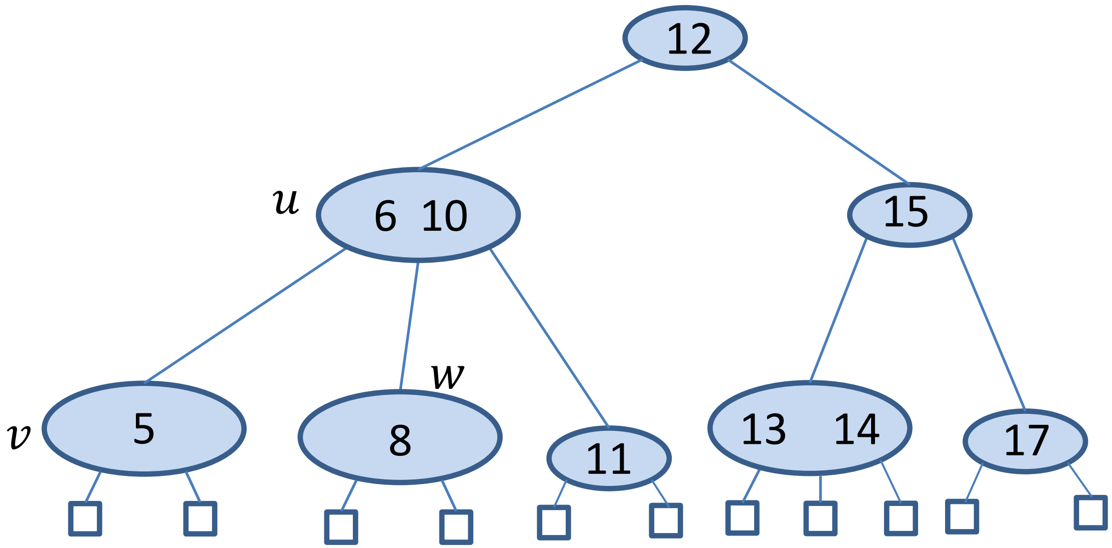
Remove 4



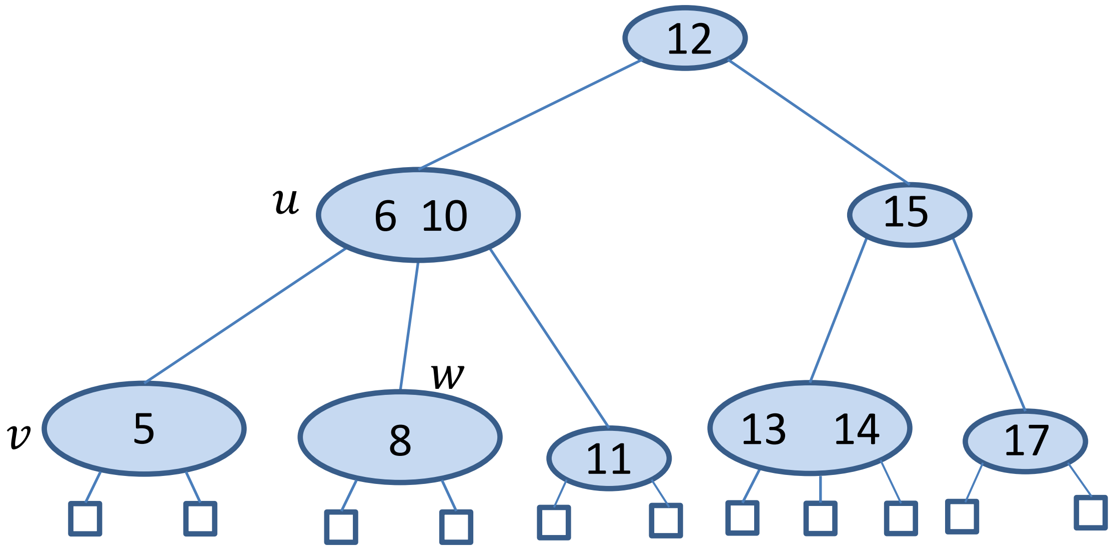
Transfer



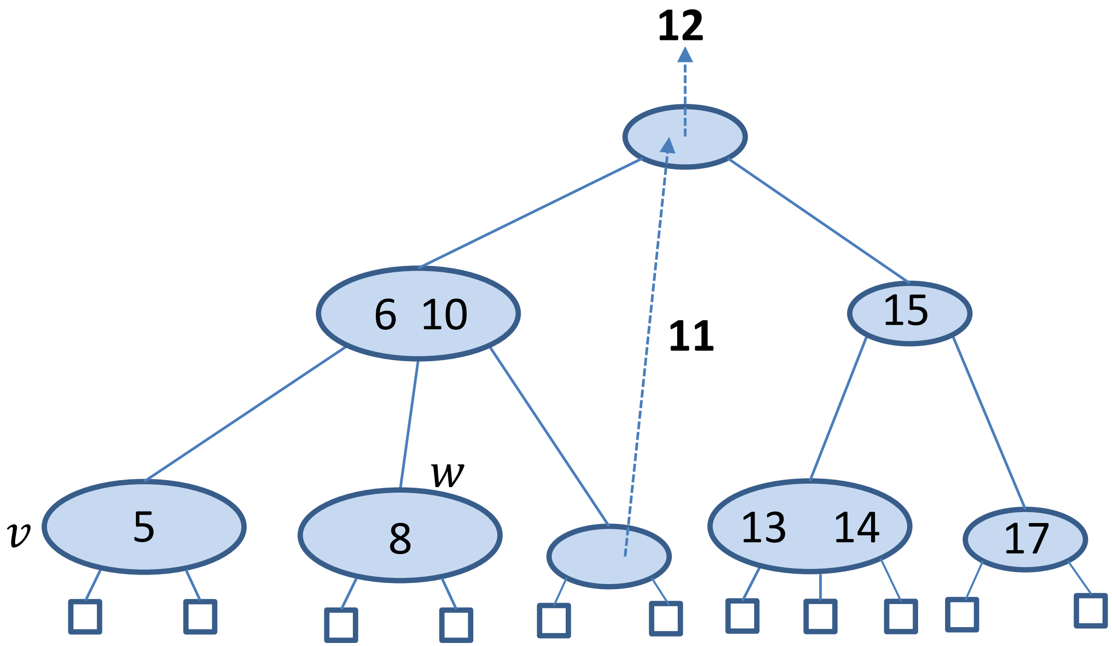
After the Transfer



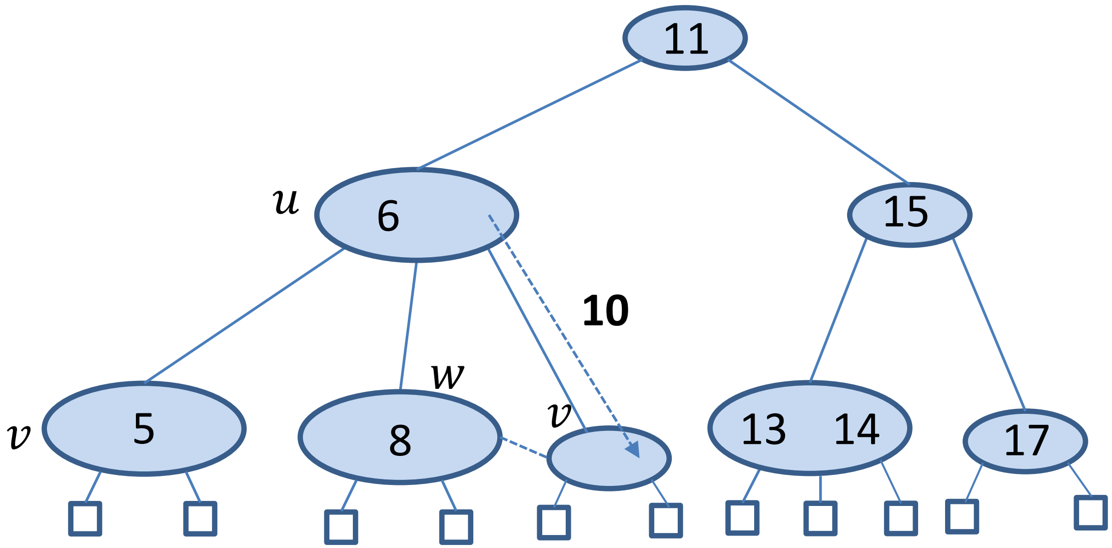
Remove 12



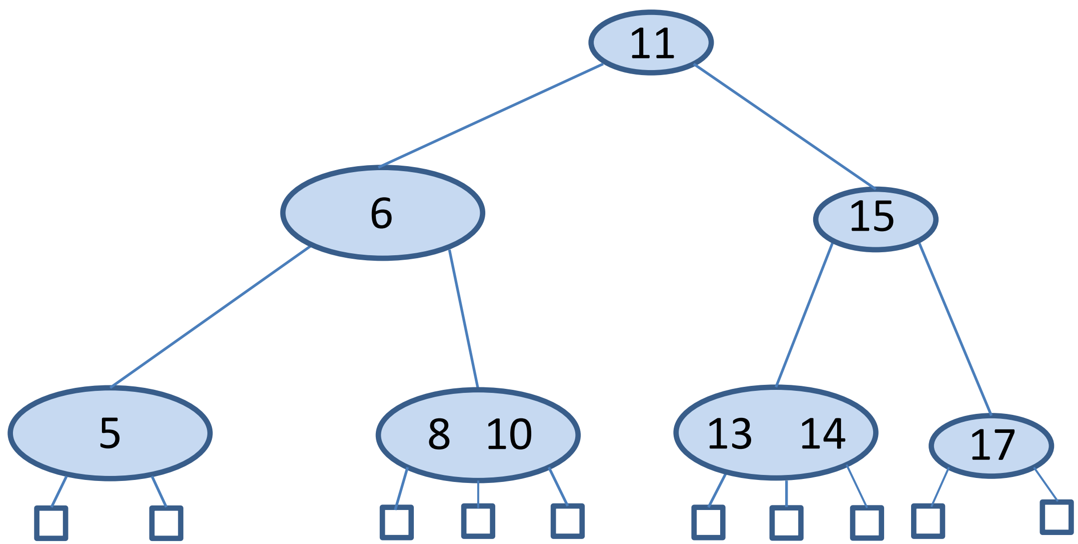
Remove 12



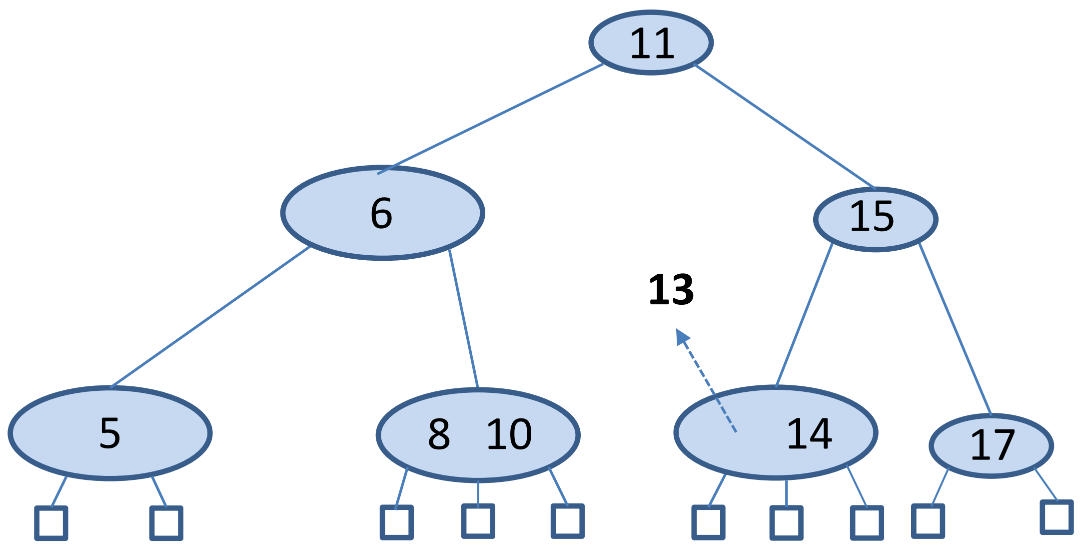
Fusion of w and v



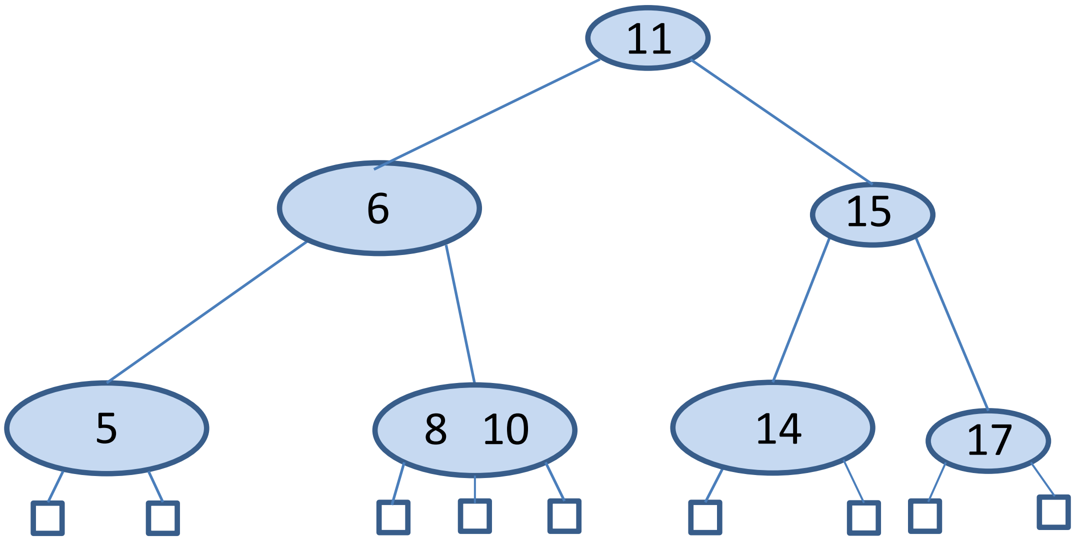
After the Fusion



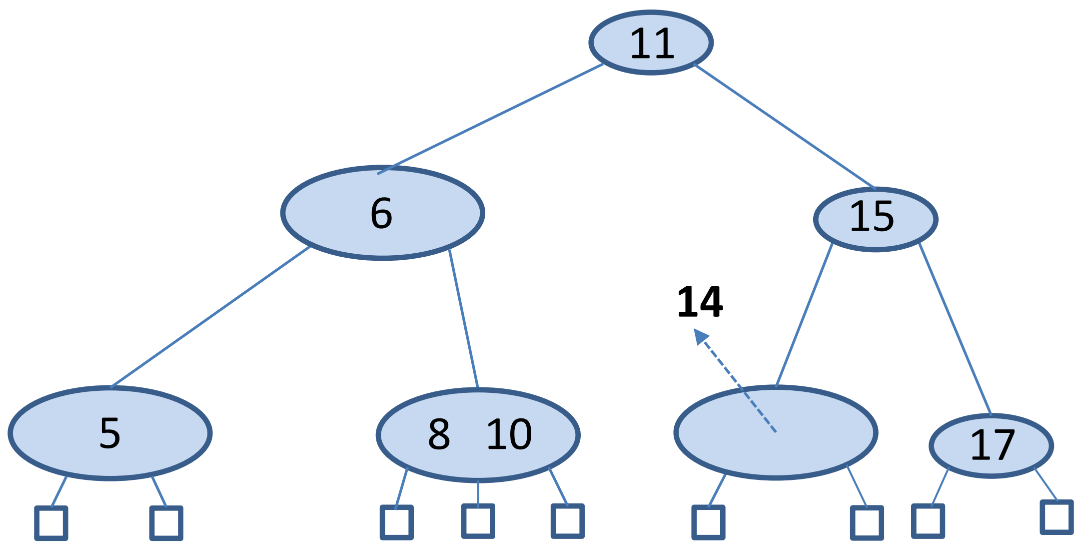
Remove 13



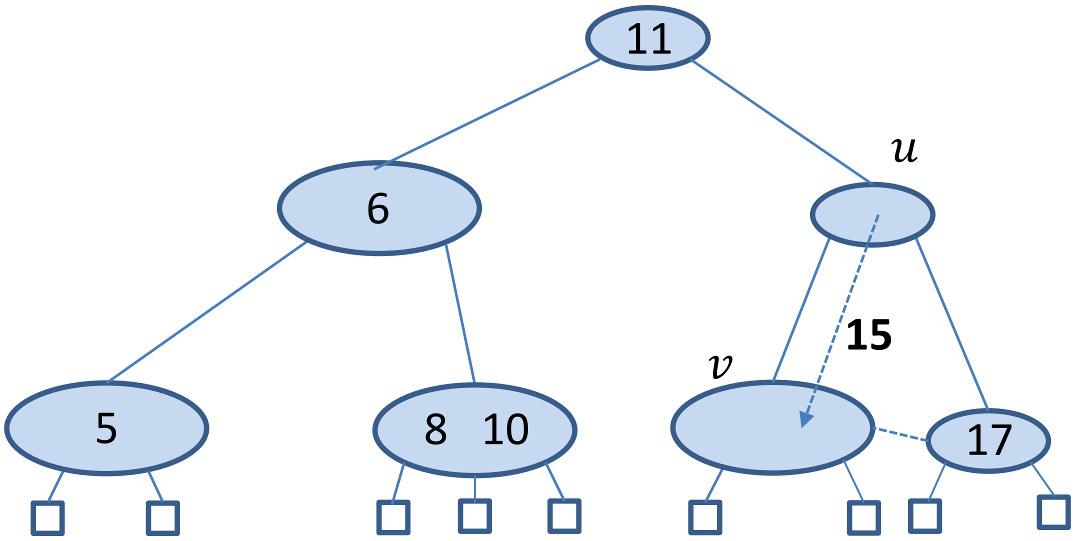
After the Removal of 13



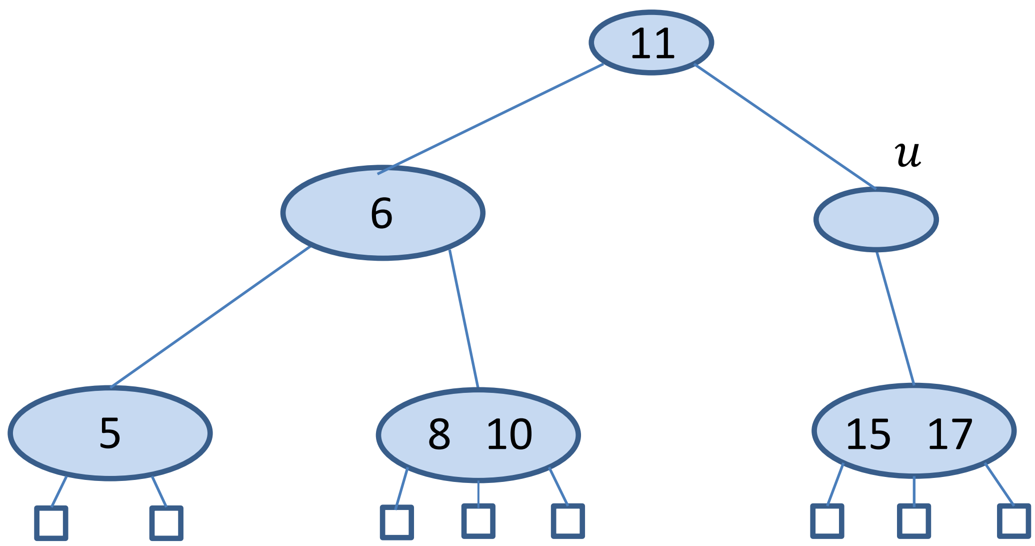
Remove 14 - Underflow



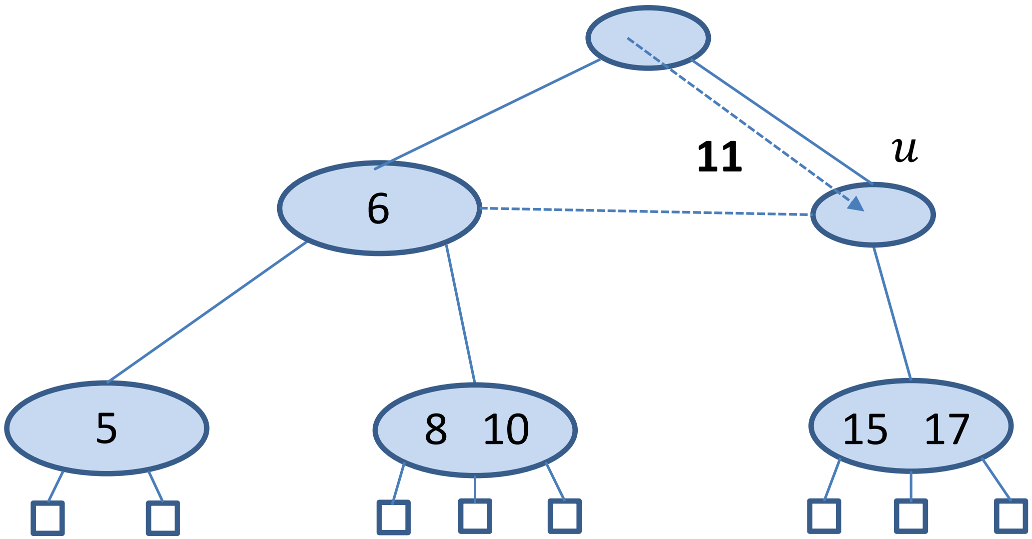
Fusion



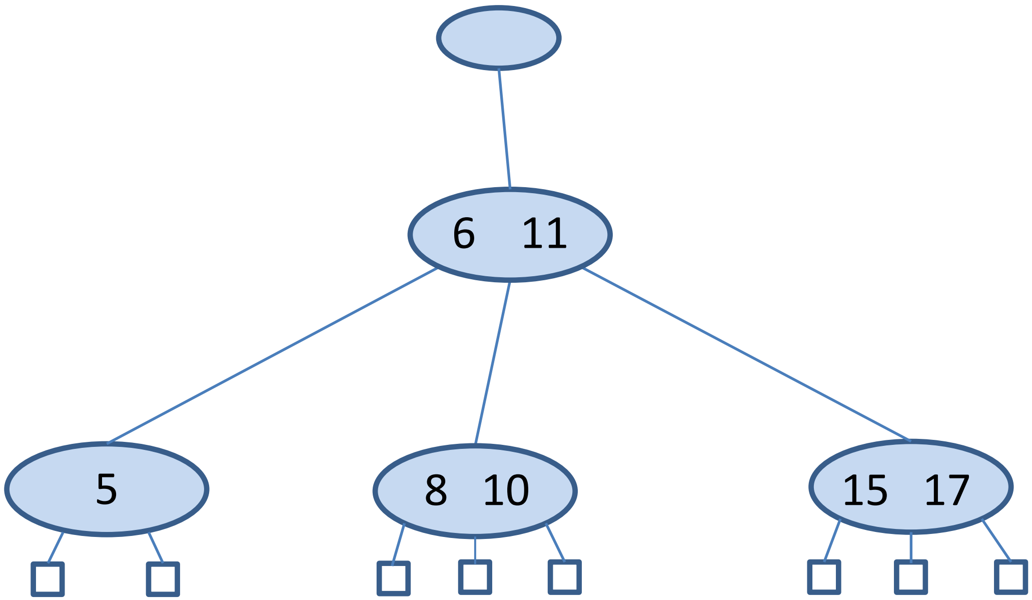
Underflow at u



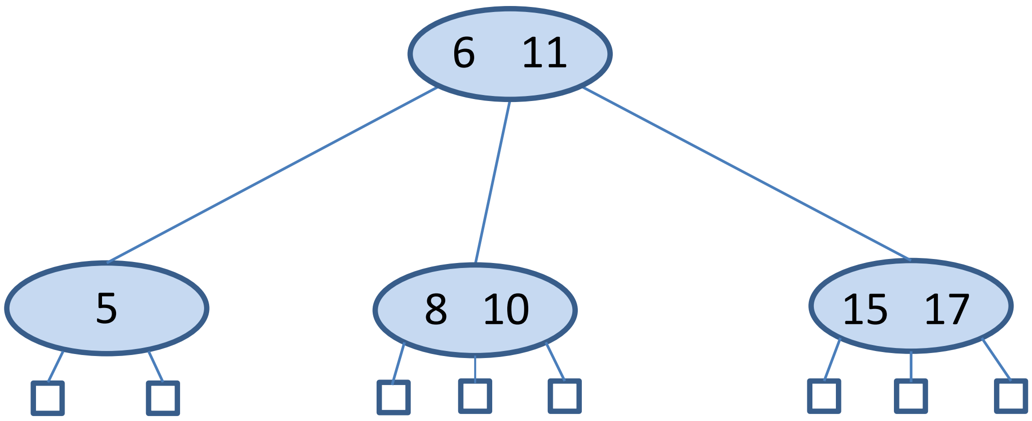
Fusion



Remove the Root



Final Tree



Complexity of Removal

- The removal of an entry consists of a **downward phase** for finding the entry to be removed and, possibly, of an **upward phase** of transfer and/or fusion operations.
- The downward phase has time complexity $O(h) = O(\log n)$.
- A **transfer** operation is local to three nodes hence it takes $O(1)$ time.
- 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.**

Readings

- M. T. Goodrich, R. Tamassia and Michael H. Goldwasser. *Data Structures and Algorithms in Java. 6th* edition. John Wiley and Sons, 2014.
  - Section 11.5
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
  - Section 13.3