CSC 226

Algorithms and Data Structures: II

2-3 Trees

Tianming Wei
twei@uvic.ca
ECS 466

Why balanced BSTs?

- Reminder: Definition of Binary Search Tree (BST)
- Search
- Insertion
- Deletion

Definition: Binary Search Tree (BST)

- A *binary search tree* (*BST*) is a binary tree where
  - each node has a (comparable) key and
  - satisfies the restriction that the key in any node is
    - larger than the keys in all nodes in that node's left subtree and
    - smaller than the keys in all nodes in that node's right subtree

Convention

- In a binary search tree, keys are stored in internal nodes only
- Every internal node in a binary search tree contains a key/an element with a key
- Every node has exactly two children (one or two of which can be leaves)

Search

- recursive
- follows structure of tree
- return the key's associated value if search successful; null otherwise

Insertion of new key

- Perform search (ends in leaf)
- replace leaf with new node containing the new key

Deletion

- Search key
  - key not found
  - key found

Deletion of existing key (key found)

- Three cases
  1. The node containing the key is a parent of leaves (null) only
     - simply remove the node and replace it by a leaf
  2. The node containing the key is a parent of one internal node only
     - Remove the internal node and replace it with the child that is an internal node

Deletion of existing key

3. The node *x* containing the key is parent of two internal nodes
   - Identify the node *y* that is *x*'s in-order successor
   - Replace the content of *x* with the content of *y*
   - Delete key of *y* in subtree rooted by node *y*
     - *Note:* node *y* will have at most one internal child node and thus case 1 or 2 will apply

Properties of binary search trees

- Height $O(n)$
- Worst-Case Time complexity
  - Search $O(n)$
  - Insertion $O(n)$
  - Deletion $O(n)$

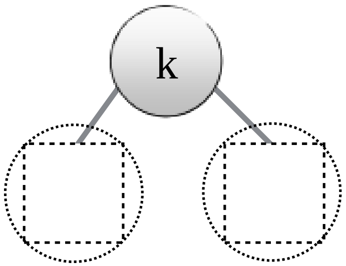
Balanced Search Trees

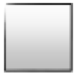
- Why balanced search trees?
  - unbalanced search trees are not efficient due to height O(n)
- Examples
  - AVL trees
  - 2-3 trees & red-black trees

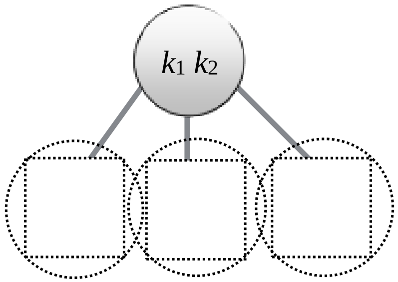
2-3 trees

- Not binary
  - In contrast to AVL trees and red black trees
- How to guarantee balance?
  - Nodes can hold more than one key
- 2-node: holds one key, has two children
- 3-node: holds two keys, has three children
- Assumption: all keys different

Definition (2-3 tree)



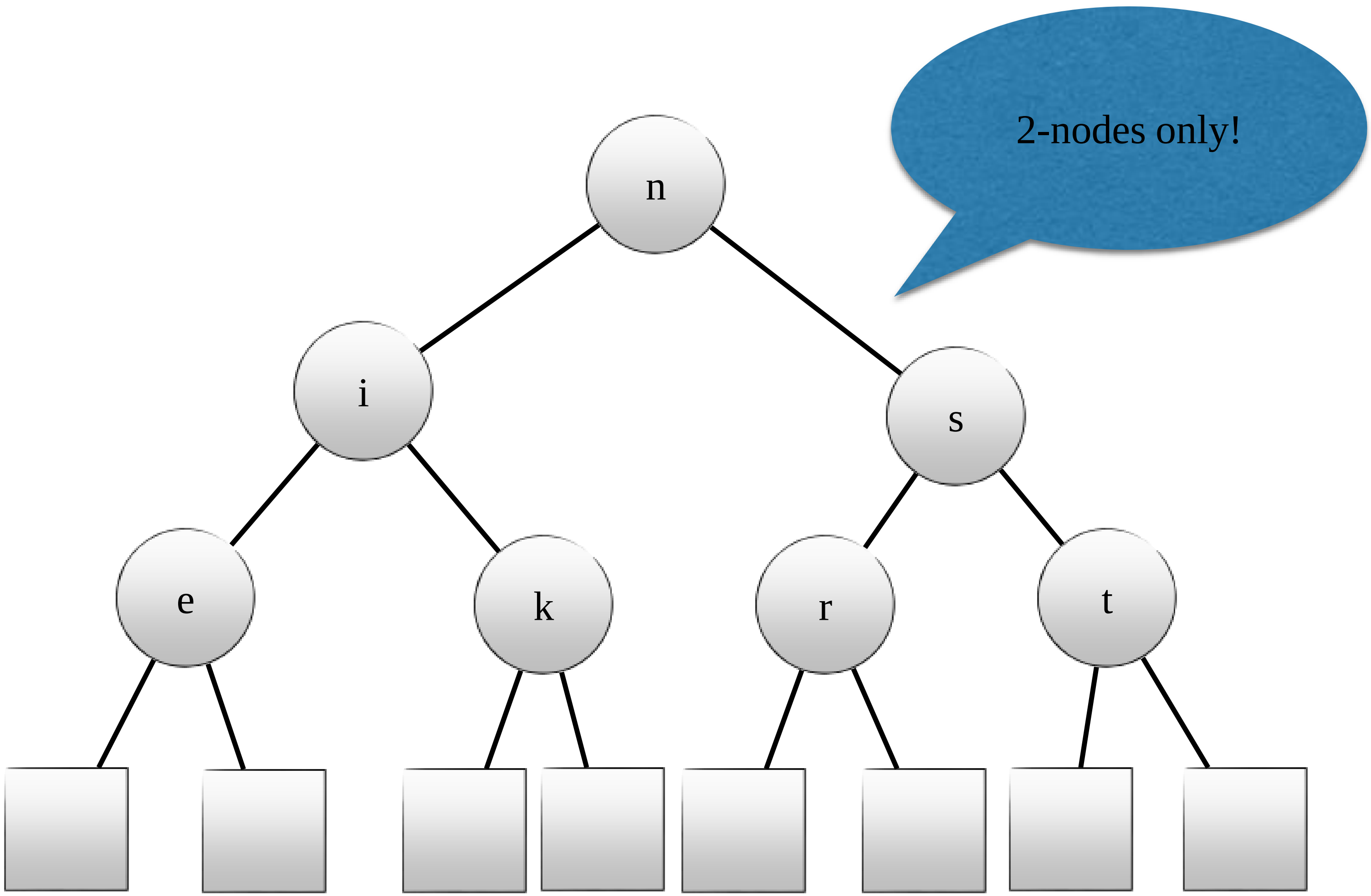
- A *2-3 search tree* is a tree that is
  - either empty 
  - or a *2-node*, with one key k (and associated value) and two links: a left link to a 2-3 search tree with keys smaller than k, and a right link to a 2-3 search tree with keys larger than k
  - or a *3-node*, with two keys $k_1 < k_2$ (and associated values) and three links: a left link to a 2-3 search tree with keys smaller than k_1, a middle link to a 2-3 search tree with keys larger than k_1 and smaller than k_2, and a right link to a 2-3 search tree with keys larger than k_2



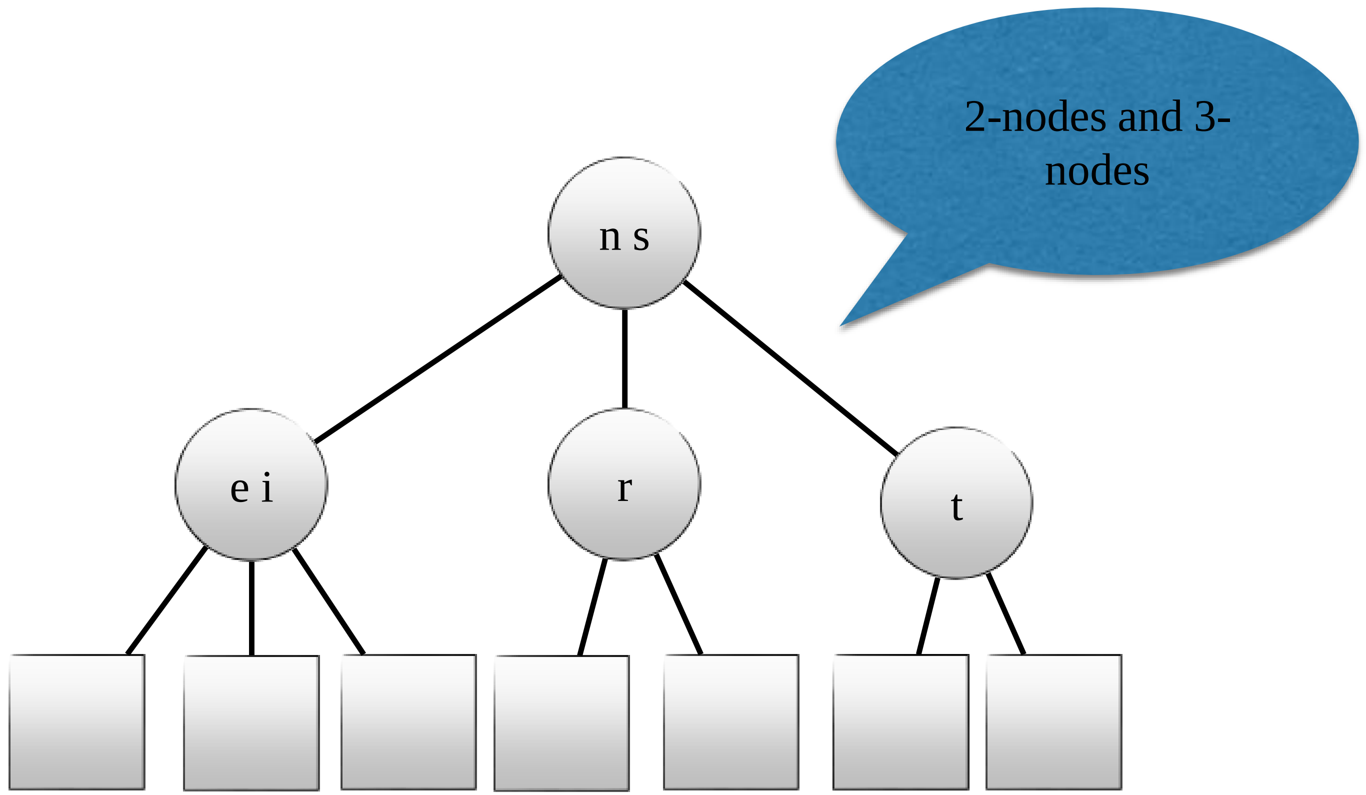
Definition (2-3 tree, continued)

- A link to an empty tree is called a *null link* or *leaf*.
- A *2-3 tree* is a *perfectly balanced* 2-3 search tree, which is one where all null links are the same distance from the root (i.e same depth.)

Example of a 2-3 tree

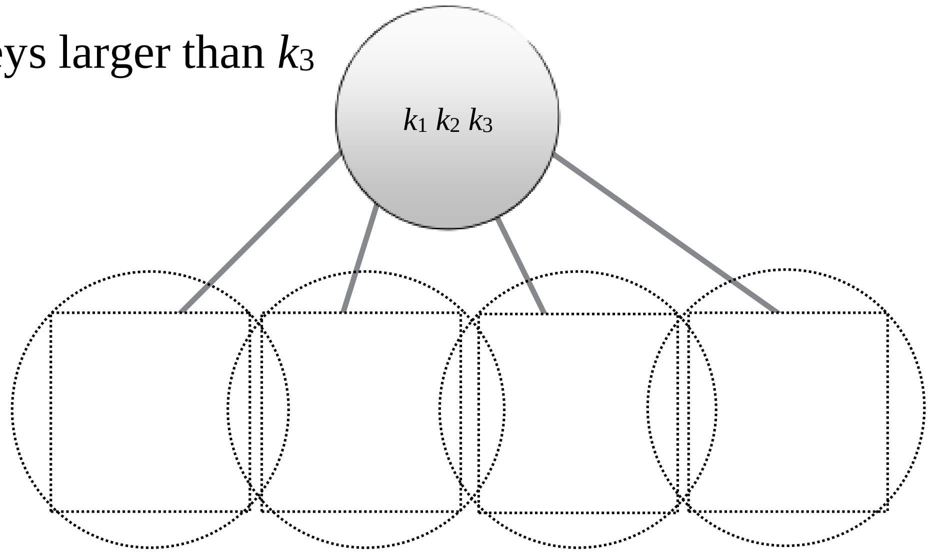


Example of a 2-3 tree



auxiliary nodes: 4-nodes

- On a temporary bases when working with 2-3 trees we will make use of 4-nodes:
- a 4-*node*, with three keys $k_1 < k_2 < k_3$ (and associated values) and four links
  - ➢a left link to a 2-3 search tree with keys smaller than k_1,
  - ➢a left middle link to a 2-3 search tree with keys larger than k_1 and smaller than k_2,
  - ➢a right middle link with keys larger than k_2 and smaller than k_3, and
  - ➢a right link to a 2-3 search tree with keys larger than k_3



Supported methods

- Search a key
- Insert an element/key and associated value
- Delete an element/key and associated value

2-3 trees: search

- Generalization of binary search
- **If root node is a 2-node then** compare search key *s* against root key *k*
  - If $s = k$ then return element with key *k*
  - Else if $s < k$ then recurse on left subtree
  - Else if $s > k$ then recurse on right subtree

2-3 tree: search (continued)

- **If root node is 3-node then** compare search key s with 3-node keys k_1 and k_2
  - If $s = k_1$ then return element with key k_1
  - If $s = k_2$ then return element with key k_2
  - If $s < k_1$ then recurse on left subtree
  - If $s > k_2$ then recurse on right subtree
  - Else recurse on middle subtree

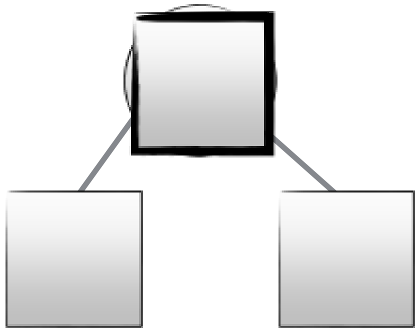
2-3 tree: search (continued)

- **If root is empty/leaf then** search key is not contained in the 2-3 tree

2-3 trees: insertion of an element with key k

- We only insert if key k is not yet in the tree. The search for key k returns a leaf.
- **Case 1.** If the leaf is root, then the tree is empty and the leaf (root node) is replaced by a 2-node with key k
- **Otherwise**, the search terminates in a leaf with parent node v.
- We distinguish two cases
  - **Case 2.** v is a 2-node
  - **Case 3.** v is a 3-node

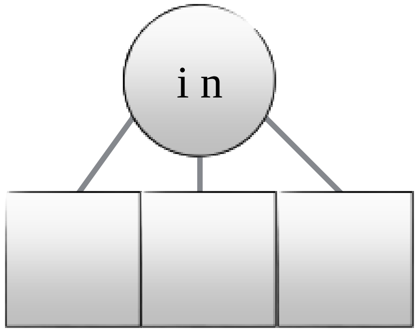
Case 1. Inserting key *i* into an empty tree



Case 2. *v* is a 2-node

- Replace *v* with a 3-node containing both its original key and the new key to be inserted
- Note: The tree remains perfectly balanced and satisfies the search-tree properties

Case 2. Inserting key *n*



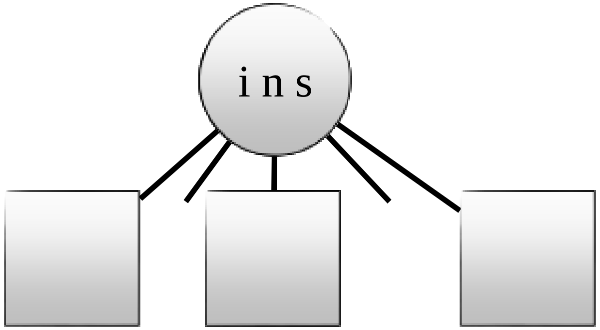
Case 3. *v* is a 3-node

- We distinguish the following cases
  - **Case 3.1** *v* is root
  - **Case 3.2** *v*'s parent is a 2-node
  - **Case 3.3** *v*'s parent is a 3-node
  - These are all cases since the search tree is perfectly balanced.

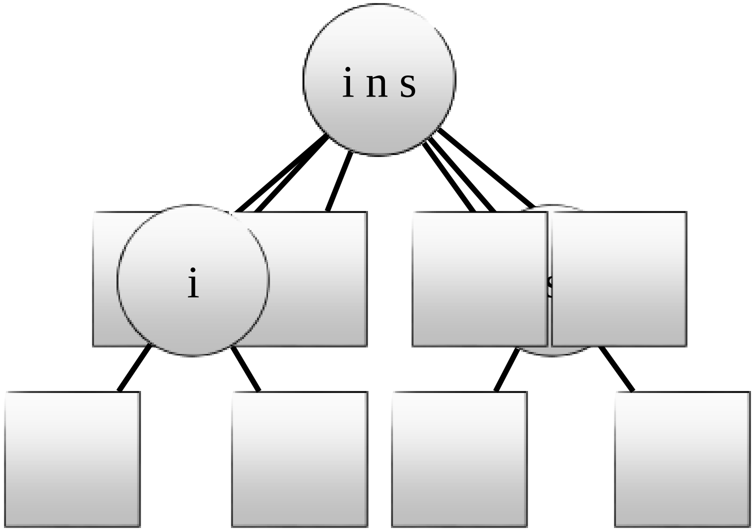
Case 3.1 *v* is root

- *v* is parent of leaves only
- Temporarily replace *v* by a 4-node with original keys and inserted new key
- Convert this tree rooted by the 4-node into a 2-3 tree consisting of three 2-nodes as follows:
  - The new root contains key k_2.
  - The left child of the root contains key k_1
  - The right child of the root contains key k_3
  - The children of the 2-nodes containing k_1 and k_3 are all leaves.

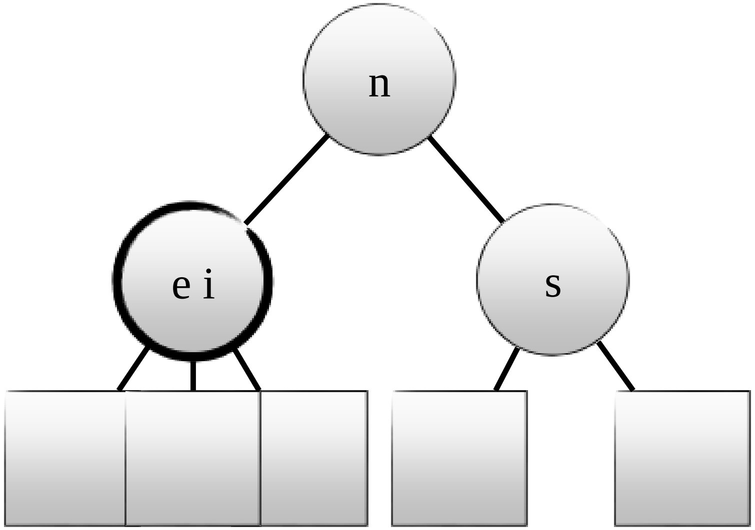
Case 3.1. Insert key *s*



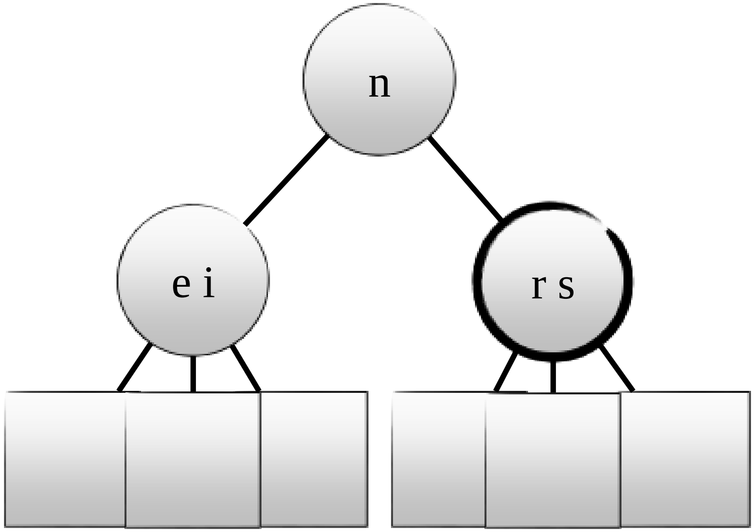
Case 3.1. Insert key *s*



Case 2. Insert key *e*



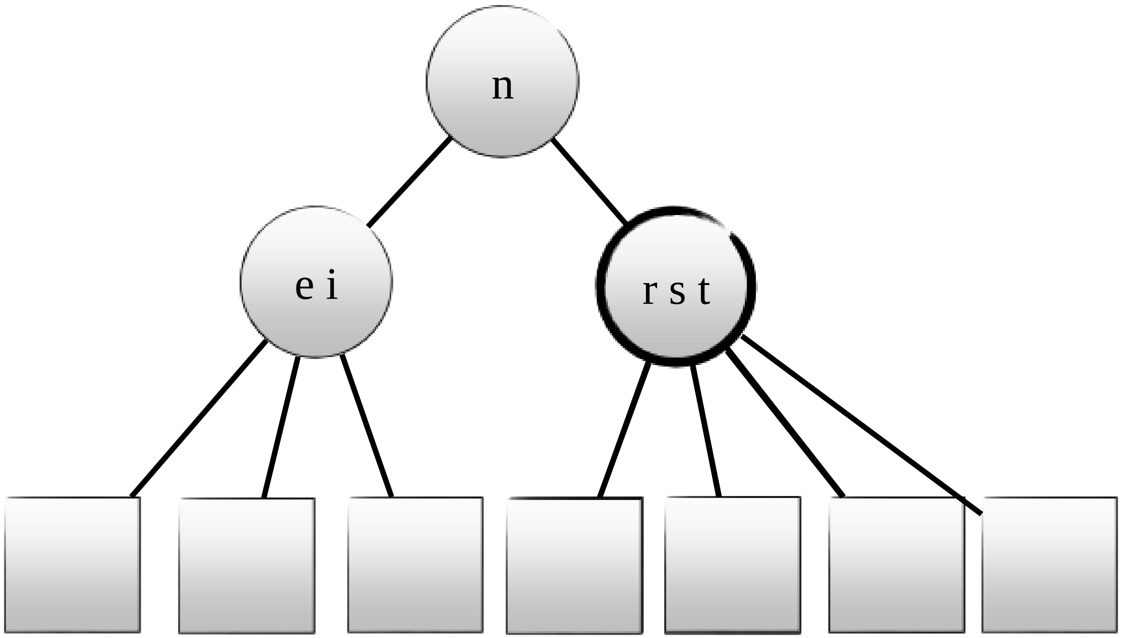
Case 2. Insert key *r*



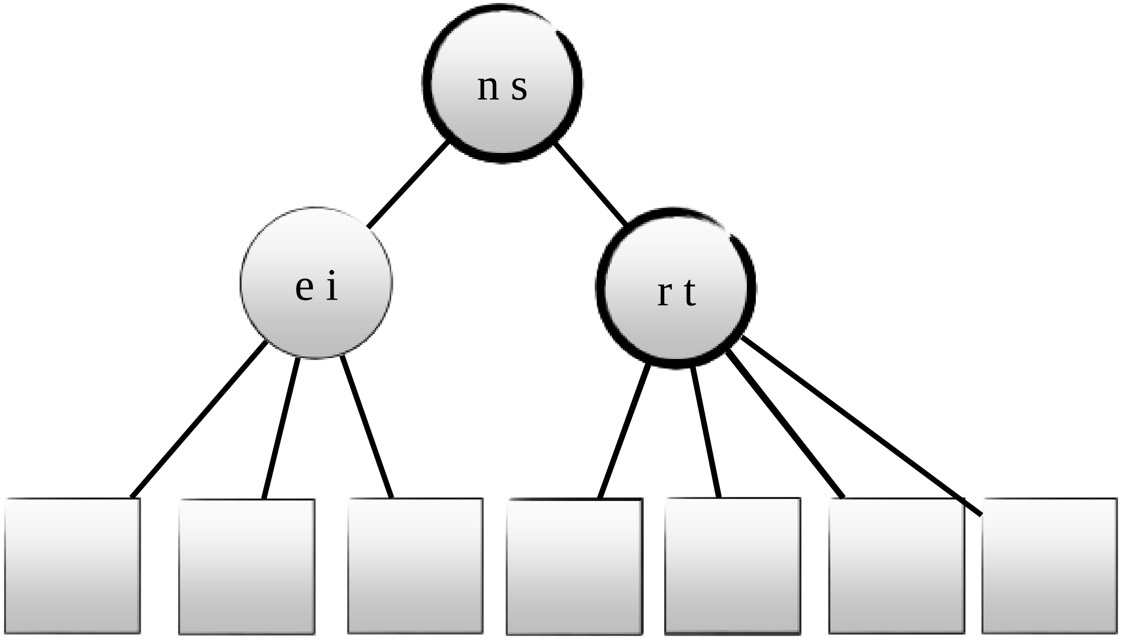
Case 3.2. The parent of *v* is a 2-node

- We replace *v* (temporarily) by a 4-node that contains the original keys of *v* and the new key to be inserted.
- Then, the middle key, k_2, is removed from the 4-node and inserted into the parent 2-node *y* (making it into a 3-node), and splitting the 4-node with its two remaining keys, k_1 and k_2, into two 2-nodes with parent *y*.

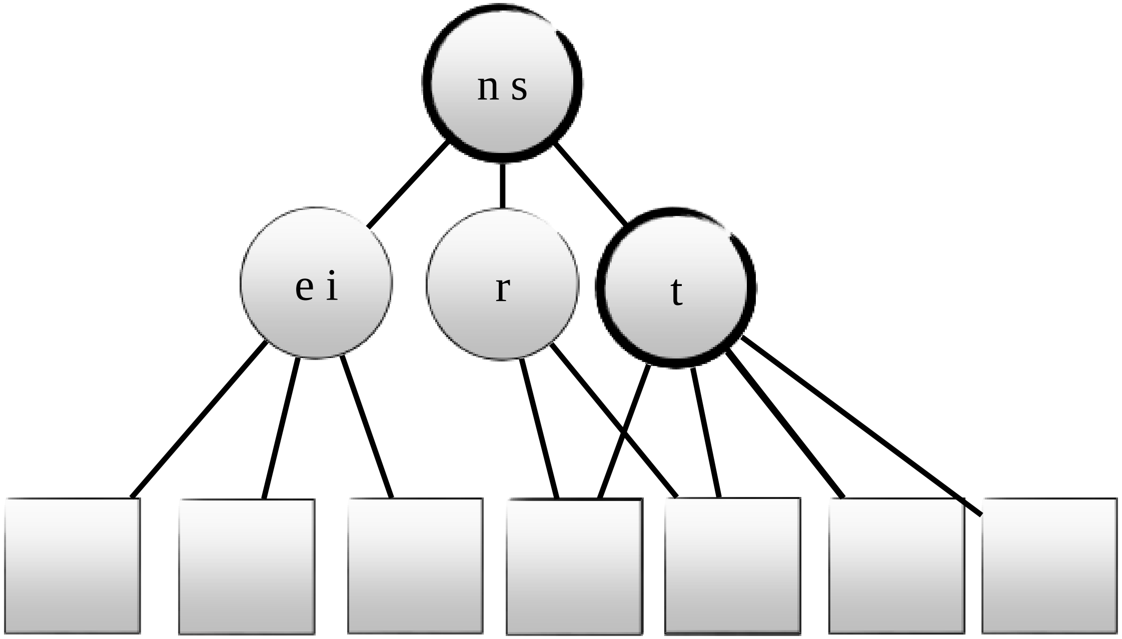
Case 3.2. Insert key *t*



Case 3.2.Insert key *t*



Case 3.2.Insert key *t*



Case 3.3. The parent *y* of 3-node *v* is a 3-node

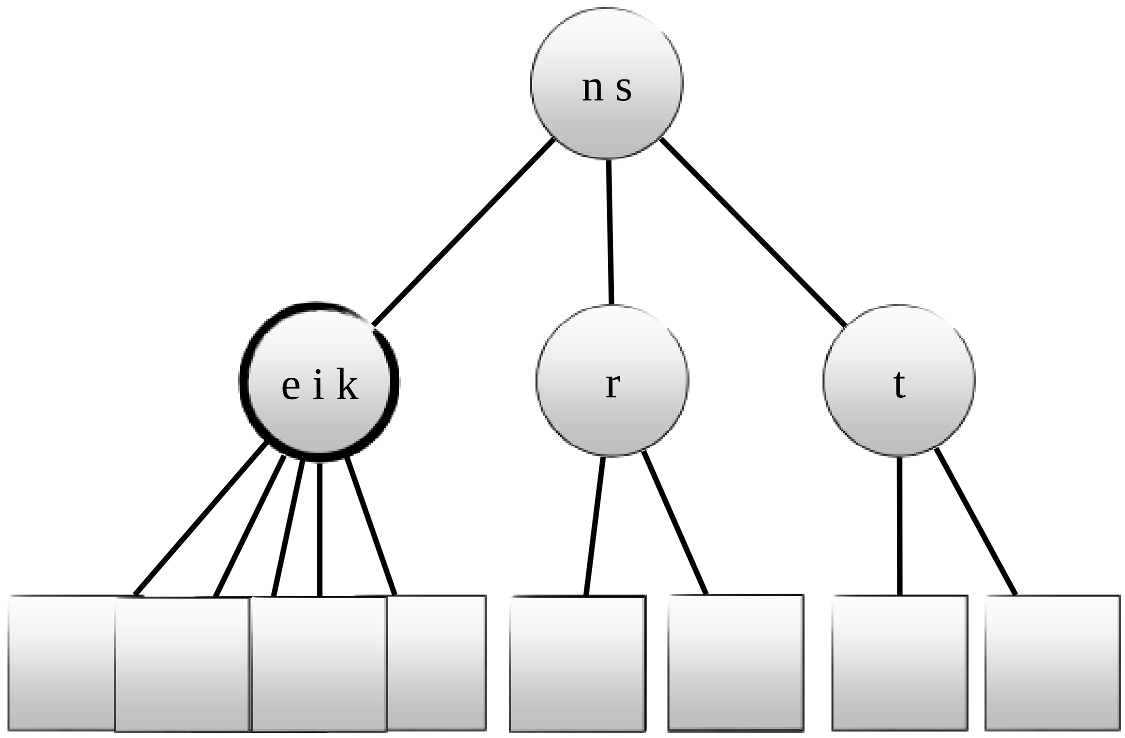
- We replace 3-node *v* (temporarily) by a 4-node that contains the original keys of *v* and the new key to be inserted.

- We then move the middle key up and insert it into the parent, creating a temporary 4-node at parent *y*.

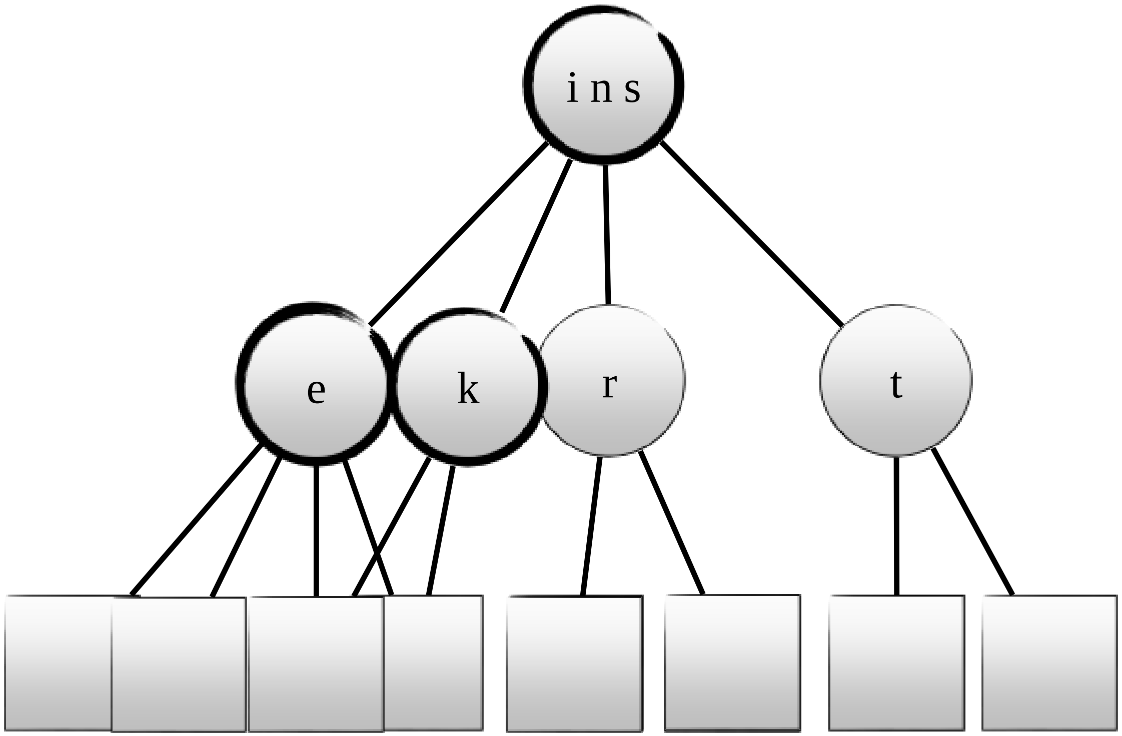
- This 4-node is either the root, has a 2-node as parent or has a 3-node as parent.

- The first case is discussed next: *splitting the root*. In the second case we continue as in Case 3.2. In the last case, we continue to move the middle key up the tree as above (Case 3.3).

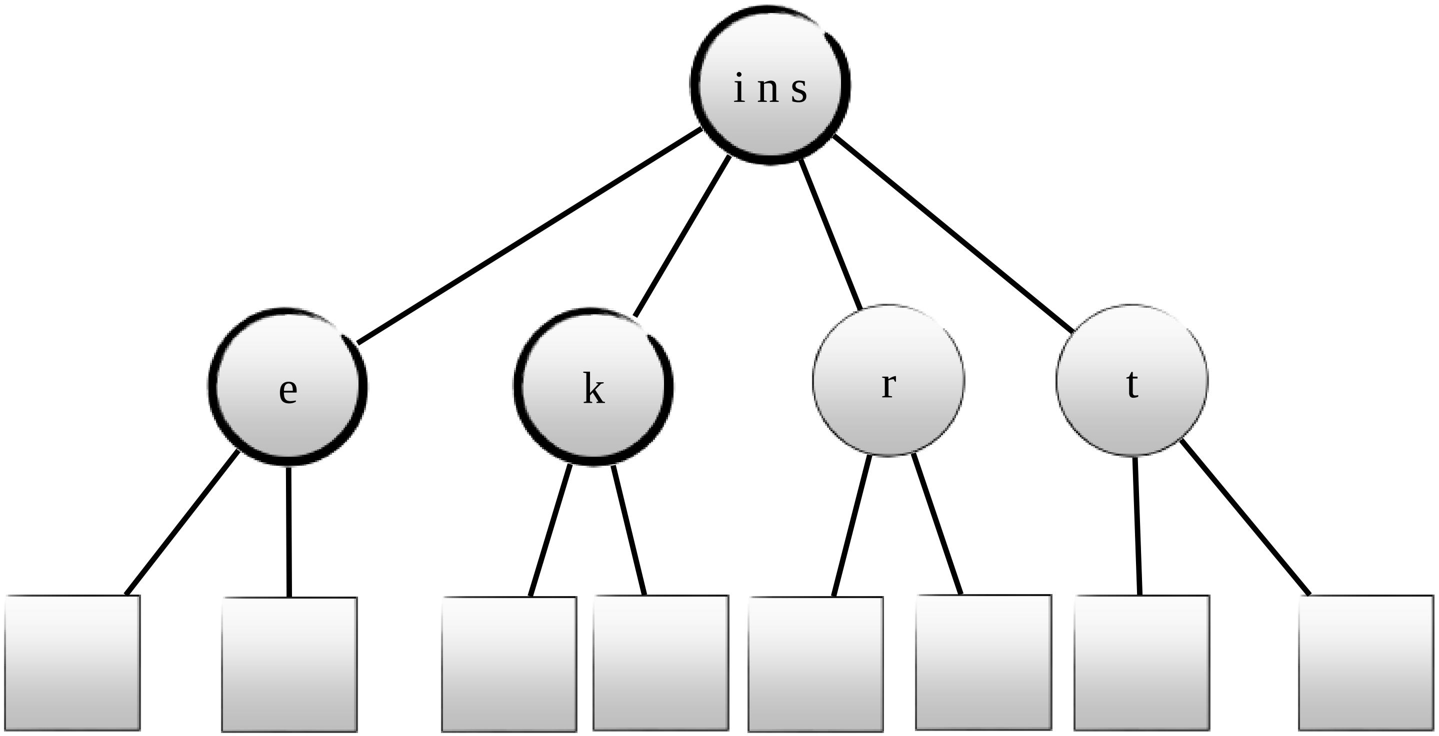
Case 3.3. Insert key *k*



Case 3.3. Insert key *k*



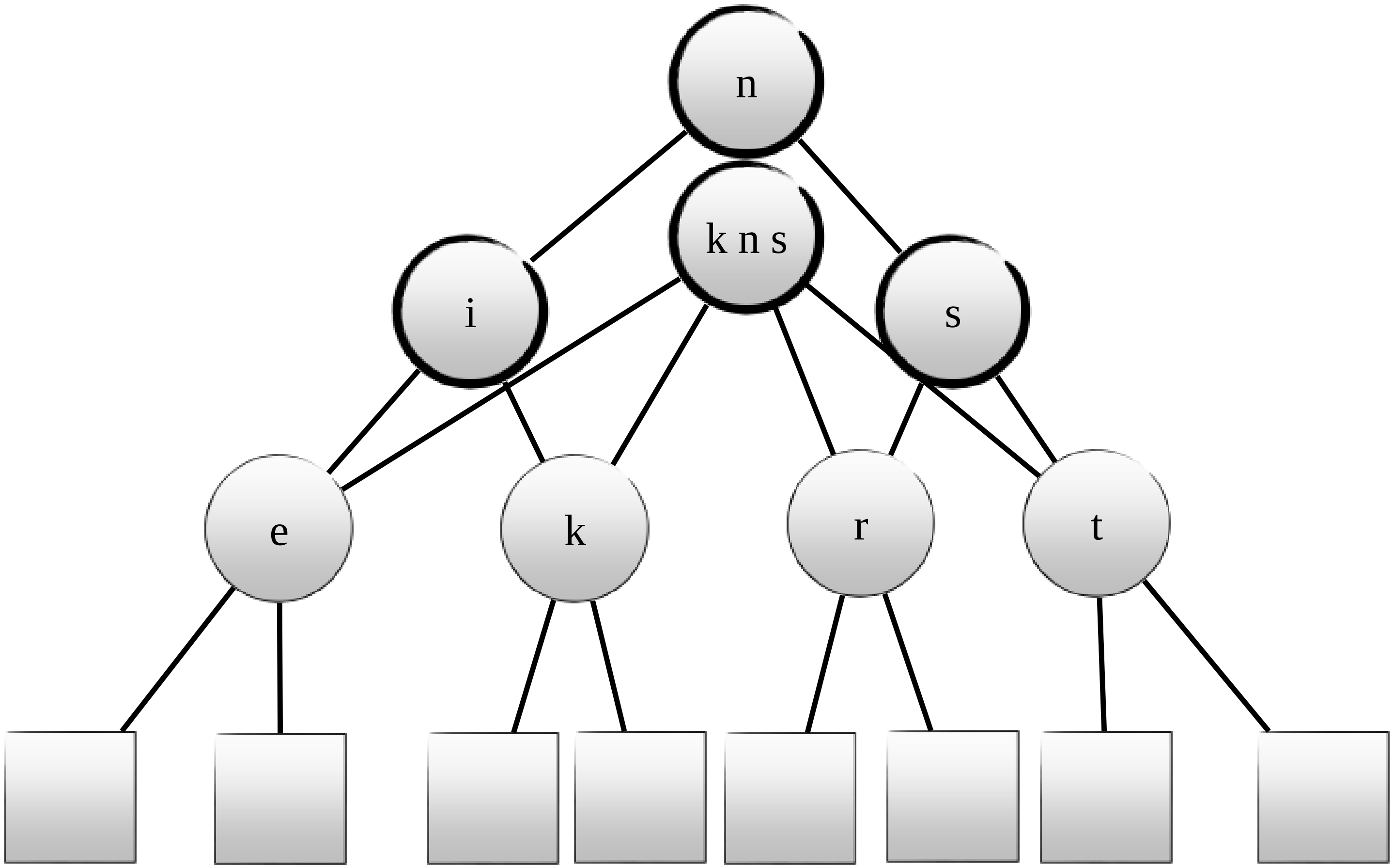
Case 3.3. Insert key *k*



Splitting the root

- Split the root into three 2-nodes (this increases the height of the tree by one), leaving the tree perfectly balanced
  - k_2 is the root key
  - k_1 the key of the root's left child; its two children are the two leftmost children of the 4-node
  - k_3 the key of the root's right child; its two children are the two rightmost children of the 4-node

Insert key *k*



Theorem

- 2-3 Search and Insertion is $O(\log n)$

Proof

- We show
  - ➢ The height of a 2-3 tree is O(log n)
  - ➢ After inserting a key k into a 2-3 tree with keys k_1, ..., k_n by the steps discussed, the resulting tree is a 2-3 tree containing keys k_1, ..., k_n, k.

Reminder: Definition 2-3 tree

- A *2-3 tree* is a *perfectly balanced* 2-3 search tree, which is one where all leaves have the same distance from the root.

Insertion: cases 1,2 & 3

- For each case we show: after insertion
  - 2-3 search tree
  - perfectly balances

After inserting a key k into a 2-3 tree with keys $k_1, \ldots, k_n$ the resulting tree is a 2-3 tree containing keys $k_1, \ldots, k_n, k$

- Recall, when inserting a key, the search for key k returns a leaf
- Case 1. If the leaf is root, then the tree is empty and the leaf (root node) is replaced by a 2-node with key k
- Otherwise, the search terminates in a leaf with parent node v
- We distinguish the cases where v is a 2-node (Case 2) and where v is a 3-node (Case 3)

Proof

- To show: After inserting a key into a 2-3 tree the tree remains

  A. a 2-3 search tree

  B. the tree is perfectly balanced

- Note that the internal node the search terminates in is always a parent of leaves only.
- Case 1. Inserting into an empty tree
- Case 2. Search terminates in a 2-node
- Case 3. Search terminates in a 3-node
  - ➢Case 3.1. Search terminates at root
  - ➢Case 3.2. Parent: 2-node
  - ➢Case 3.3. Parent: 3-node

Case 1. Inserting into an empty tree

- To show: After inserting a key into a 2-3 tree the tree remains
  A. a 2-3 search tree
  B. the tree is perfectly balanced
- After inserting a key into an empty key, the key consists of a single 2-node. Properties A and B are satisfied

Proof

- To show: After inserting a key into a 2-3 tree the tree remains

  A. a 2-3 search tree

  B. the tree is perfectly balanced

- Case 2. Search terminates in a 2-node
- The number of internal nodes does not change. The node where the key is inserted is added a third leaf, keeping the tree perfectly balanced.
- Inserting the new key into the 2-node will maintain the search tree property: The search determined the right subtree for the key to be inserted. Inserting the key to the left of the 2-node key if smaller and to the right if larger will complete the insertion maintaining the search tree property.

Proof

- To show: After inserting a key into a 2-3 tree the tree remains

  A. a 2-3 search tree

  B. the tree is perfectly balanced

- Case 3. Search terminates in a 3-node

  - Case 3.1. Search terminates at root

  - Case 3.2. Parent: 2-node

  - Case 3.3. Parent: 3-node

Proof

- Finally, we show that the height, h, of any 2-3 tree is $O(\log n)$
- How many external nodes are there in a 2-3 tree with n keys? Induction.
- What is the lower bound on the number of external nodes in terms of the height, h?