Lecture: Trees-2

TDRK12 Data Structures, 7.5 credits

Vladimir Tarasov

21 February 2022

School of Engineering, Jönköping University

JÖNKÖPING UNIVERSITY

Table of contents

M-way Search Trees

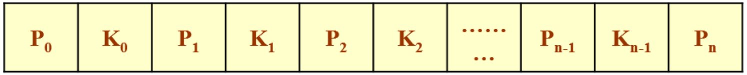
M-way Search Trees

Definition

- In an **M-way search tree** every internal node consists of pointers to M sub-trees and contains $M - 1$ keys, where $M > 2$.
- M is called the *order of the tree*.

- In a binary search tree every node contains one value and two pointers, to the node's left and right sub-trees.
- In a BST $M = 2$.

Structure of an M-way Search Tree



- In the structure of an M-way search tree node
  - $P_0, P_1, P_2, \ldots, P_n$ are pointers to the node's sub-trees
  - $K_0, K_1, K_2, \ldots, K_{n-1}$ are the key values of the node
- All the key values are stored in ascending order: $K_i < K_{i+1}$ for $0 \leq i \leq n-2$.

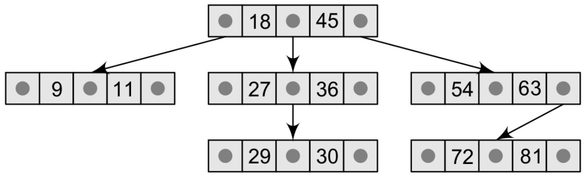
M-way search tree node

```c
// MAX is the B-tree order
struct node
{
	struct node *children[MAX];
	int keys[MAX - 1];
	int key_count;
};
struct node *root;
```

Structure of an M-way Search Tree

- In an M-way search tree, it is not compulsory that every node has exactly $(M - 1)$ values and have exactly M sub-trees (where $M > 2$).
  - The node can have from 2 to $(M - 1)$ values.
  - M is a *fixed upper limit* that defines how much key values can be stored in the node.
  - The number of sub-trees may vary from 0 (for a leaf node) to $1 + k$, where k is the number of key values in the node.

The Basic Properties of an M-way Search Tree



- In the example $M = 3$
  - A node can contain a maximum of two key values and three pointers to sub-trees.
- All the key values in the sub-tree pointed by P_i are less than K_i, where $0 \leq i \leq n - 1$
- All the key values in the sub-tree pointed by P_i are greater than K_{i-1}, where $1 \leq i \leq n$.

B-Trees

B-Trees

- A B-tree is a *specialized M-way tree* that is widely used for disk access.
- A B-tree may contain a large number of key values and pointers to sub-trees.
- Storing a large number of keys in a single node keeps *the height of the tree* relatively small.
- A B-tree is designed to store *sorted data* and allows search, insert, and delete operations to be performed in $O(\log n)$ time.

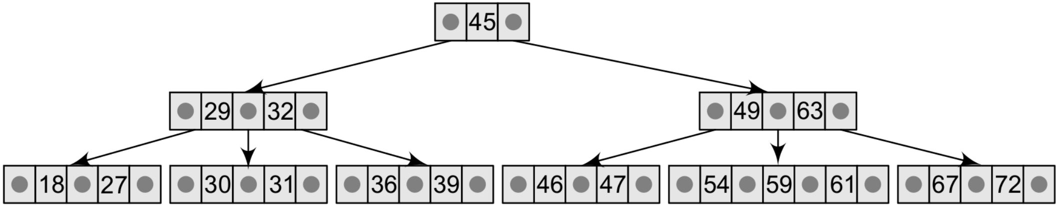
B-Trees

Definition

A B-tree of order m is a specialized M-way search tree with the following *additional* properties:

1. *Every node* in the B tree has *maximum m* children.
2. *Every node* in the B tree except the root node and leaf nodes has *minimum* $\lceil m/2 \rceil$ (ceiling of $m/2$) children (thus between $\lceil m/2 \rceil - 1$ and $m - 1$ keys).
    - This helps to keep the tree bushy so that the *path from the root node to the leaf is very short* even in a tree that stores a lot of data.
3. *The root node* has *at least* two children if it is not a terminal (leaf) node.
4. All leaf nodes are at the *same level*.

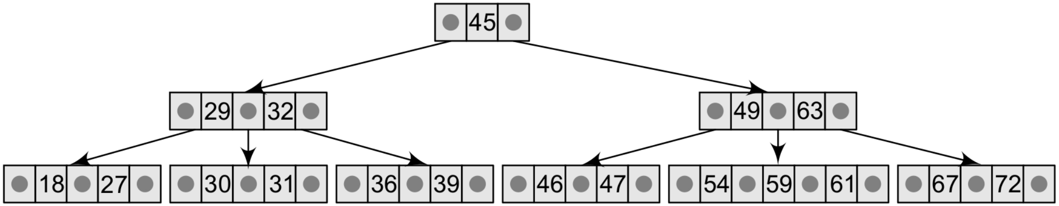
Example of a B-Tree



B-tree of order 4

- An *internal node* in the B-tree can have n number of children, where $0 \leq n \leq m$.
- Every node does not have to have the same number of children.
- The only restriction is that the node should have *at least* $\lceil m/2 \rceil$ children.

Searching for an Element in a B-Tree



- Searching for an element in a B-tree is similar to search in BST.
- The search time is $O(\log n)$ since it depends upon the height of the tree.

Searching for key 59 (in B-tree of order 4)

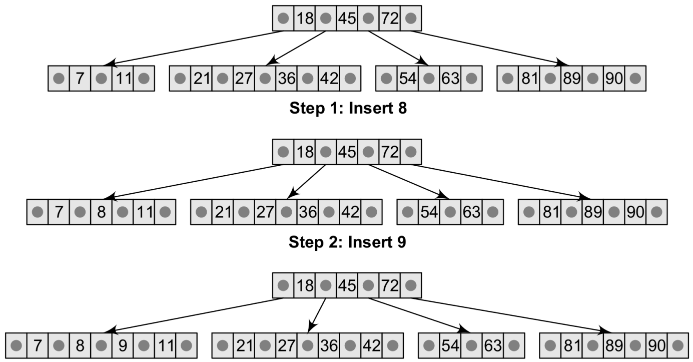
1. Since the root value $45 < 59$, traverse in the *right sub-tree*.
2. Since $49 \leq 59 \leq 63$, traverse the *right sub-tree of 49* or the *left sub-tree of 63*.
3. On finding the value 59, the search is successful.

Inserting a New Element in a B-Tree

Algorithm to insert an element in a B-tree

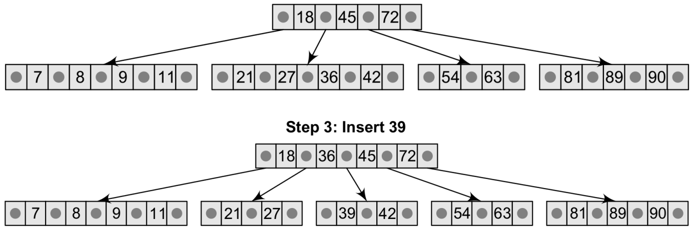
1. **Search** the B-tree to find the *leaf node* where the new key value should be *inserted*.
2. If the leaf node contains less than $m - 1$ key values, **insert** the new key in the node keeping the node's *keys ordered*.
3. If the leaf node already contains $m - 1$ key values (*full*):
   - 3.1 **insert** the new value *in order* into the existing set of keys,
   - 3.2 **split** the node at *its median* into *two nodes* (note that the split nodes are half full),
   - 3.3 **push** the *median* element up to *its parent's node*. If the parent's node is already full, then split the parent node by following the same steps.

Example of Insertion in a B Tree of order 5



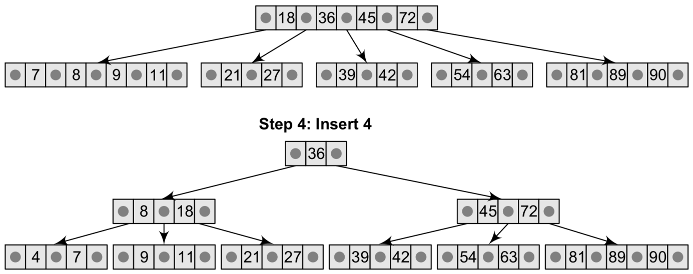
1. Since **8** $<$ 18, traverse in the *left sub-tree*.
2. The leaf node contains $<$ (5 – 1) key values, so insert the new element *keeping the element ordering*.
3. Insert **9** following the same steps.

Example of Insertion in a B Tree of order 5



1. The node in which **39** should be inserted is already full, so split the node.
2. Before splitting, arrange the key values *in order*: 21, 27, 36, 39, 42.
3. The median value is 36, so push it into its parent's node and split the leaf nodes.

Example of Insertion in a B Tree of order 5



1. The node in which **4** should be inserted is already full so split the node.
2. Before splitting, arrange the key values *in order*: 4, 7, 8, 9, 11.
3. The median value is 8, so push it into its parent's node and split the leaf nodes.
4. But the parent's node is already full, so split the parent node using the same procedure.

Deleting an Element from a B-Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion:

1. A *leaf node* has to be deleted
2. An *internal node* has to be deleted

Deleting an Element from a B-Tree

Algorithm to delete a *leaf node* from a B tree

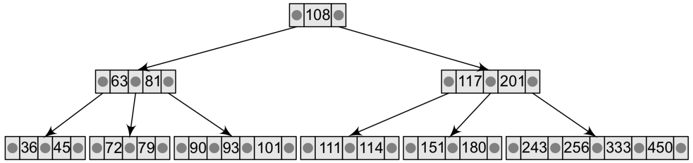
1. **Locate** the *leaf node* which has to be deleted.
2. If the leaf node *contains more than the minimum* of keys ($> \lceil m/2 \rceil - 1$), **delete** the value.
3. Else if the leaf *node does not contain* $> \lceil m/2 \rceil - 1$ keys, **fill** the node by taking a key from either the left or right sibling.
   3.1 If the *left sibling has more than the minimum of keys*, **push** its *largest key* into its parent and **pull down** the *intervening element* from the parent to the leaf where the key is deleted.
   3.2 Else, if the *right sibling has more than the minimum of keys*, **push** its *smallest key* into its parent and **pull down** the *intervening element* from the parent to the leaf where the key is deleted.
4. Else, if both left and right siblings contain *only the minimum of keys* ($\lceil m/2 \rceil - 1$),
   4.1 **Create** a new leaf by *combining* the two leaves and the intervening element of the parent (ensuring that the number of keys does not exceed the maximum ($m - 1$).
   4.2 If pulling the intervening element from the parent leaves it with less than the minimum of keys, **propagate** the (combination) process upwards possibly *reducing the height* of the B tree.

Deleting an Element from a B-Tree

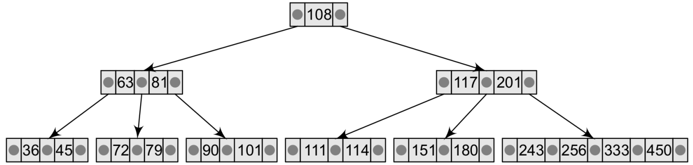
Algorithm to delete an *internal node* from a B tree

1. **Promote** the *successor or predecessor* of the key to be deleted to **occupy** the position of the deleted key.
   - This predecessor or successor will always be in the leaf node.
2. **Continue** the processing *as if* a value (predecessor or successor) from the *leaf node* has been **deleted**.

Example of Deletion in a B-Tree of order 5

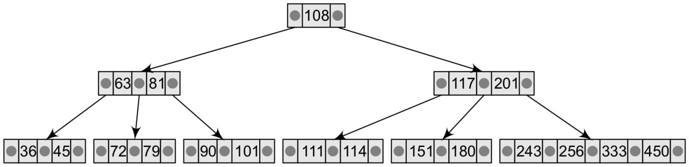


Step 1: Delete 93

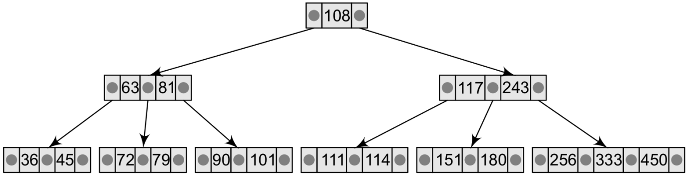


1. Since **93** < 108, traverse in the *left sub-tree*.
2. Since $93 > 81$, traverse in the *right sub-tree*, which is a **leaf node**.
3. The leaf node contains 3 keys ($> \lceil 5/2 \rceil - 1$), so delete the value (93).

Example of Deletion in a B-Tree of order 5

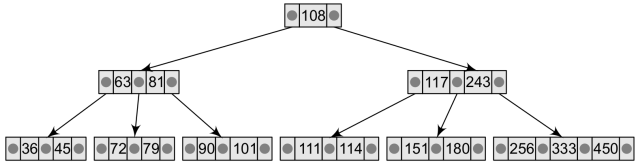


Step 2: Delete 201

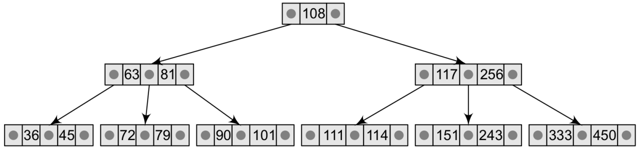


1. Since **201** $>$ 108, traverse in the *right sub-tree*, which is an **internal node**.
2. Promote the *successor* (243) of the key to be deleted to occupy the position of the deleted key (201).
3. Continue the processing by deleting the value of the successor (243) from the leaf node. The **leaf node** contains 4 keys ($> \lceil 5/2 \rceil - 1$), so delete the value (243).

Example of Deletion in a B-Tree of order 5

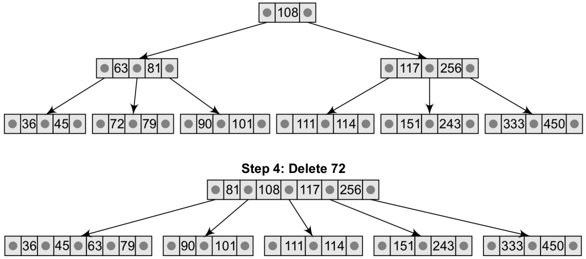


Step 3: Delete 180



1. Since **180** $>$ 108, traverse in the *right sub-tree* and then traverse the *left sub-tree* of **243** since 117 $\leq$ 180 $\leq$ 243.
2. The **leaf node** contains exactly 2 keys (not more than $\lceil 5/2 \rceil - 1$), so fill the node by taking an element from the *right sibling* after deleting 180.
   - 2.1 It has more than the minimum of keys ($\lceil 5/2 \rceil - 1$), so push its smallest key (256) into its parent, and
   - 2.2 Pull down the *intervening element* (243) from the parent to the leaf where the key (180) is deleted.

Example of Deletion in a B-Tree of order 5



1. Since **72** < 108, traverse in the *left sub-tree* of 108 and then traverse the *right sub-tree* of **63** since $63 \leq 72 \leq 81$.
2. The **leaf node** does not contain more than $\lceil 5/2 \rceil - 1$ keys and *both siblings* contain only the minimum of keys:
   - 2.1 After deleting 72 create a *new leaf* by combining the siblings and the *intervening element* of the parent (63): 36, 45, 63, 79 (the number of keys is the allowed maximum $4 = 5 - 1$).
   - 2.2 Pulling the intervening element leaves the parent with one key ($< \lceil 5/2 \rceil - 1$), so propagate the combination upwards *reducing the tree height*.
     - Create an *internal node* by combining the parent and its siblings: 81, 108, 117, 256 (the number of keys is $4 = 5 - 1$).

B+ Trees

B+ Trees

- A B+ tree is a variant of a B tree which stores *sorted data*—records identified by a *key*.
  - This allows for efficient insertion, retrieval and removal.
- A B+ tree, stores all records at the leaf level of the tree; only keys are stored in internal nodes.
  - In contrast, a B tree can store both keys and records in its internal nodes.

B+ Trees

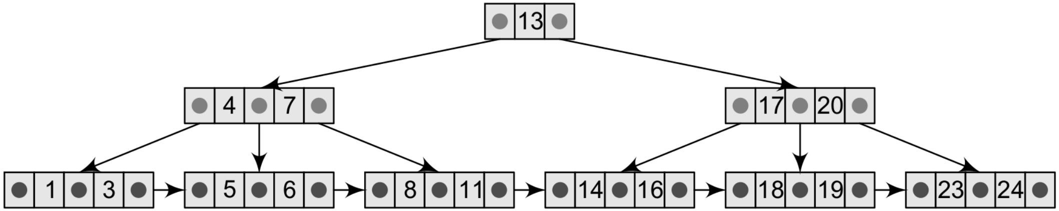
Definition

- B+-tree stores **data** only in the *leaf nodes*. All other nodes (internal nodes) are called *index nodes* or *i-nodes* and store **index values**.
- The *leaf nodes* of the B+ tree are *often linked* to one another in a linked list.

- Linking nodes has an added advantage of making the queries simpler and more efficient.
- B+ trees are used to store large amounts of data:
  - The secondary storage is used to store the *leaf nodes* of the tree.
  - The *internal nodes* of the tree are stored in the main memory.

B+ Trees Structure

- This structure allows us to traverse the tree from the root down to the leaf node that stores the desired data item.
- Many database systems use such a structure because of its simplicity.
    - Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.



Example of a B+ tree of order 3.

The Advantages of B+ Ttrees

1. Records can be fetched in equal number of disk accesses.
2. It can be used to perform search and deletion easily as the data can be found in leaf nodes only.
3. Height of the tree is less and balanced.
4. Supports both random and sequential access to records.
5. Keys are used for indexing.

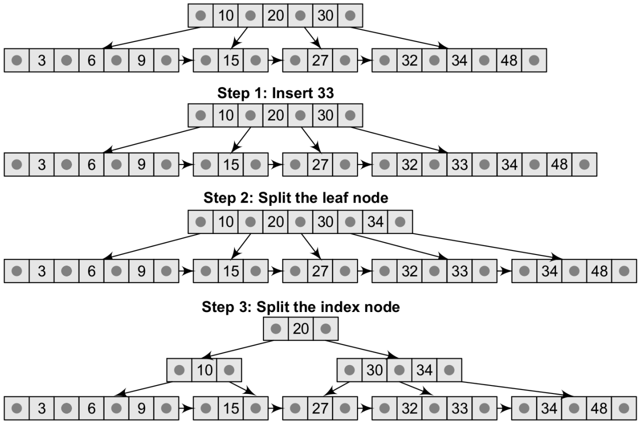
Inserting a New Element in a B+ Tree

Algorithm to insert a node in a B+ Tree

1. **Insert** the new node in the *leaf node*.
2. If the leaf node *overflows*, **split** the node and copy the *middle element* to **next index node**.
3. If the index node *overflows*, **split** that node and move the *middle element* to **next index node**.

- If the data node in the tree is full, then node is split into two nodes.
  - A new index value is added in the parent index node.
- The parent node maybe, in turn, split as well.
  - All the nodes on the path from a leaf to the root may split.
- If the root splits, a new internal node is created and the tree grows by one level.

Inserting 33 in a B+ Tree of Order 4



1. Traverse right and insert **33** in the leaf node.
2. The leaf node *overflows* $(4 > 3)$, so split it and copy the *middle element* (34) to next index node (*root*).
3. The root index node *overflows* $(4 > 3)$, so split it and move the *middle element* (20) to next index page (*the new root*).
    - The tree grows by one.

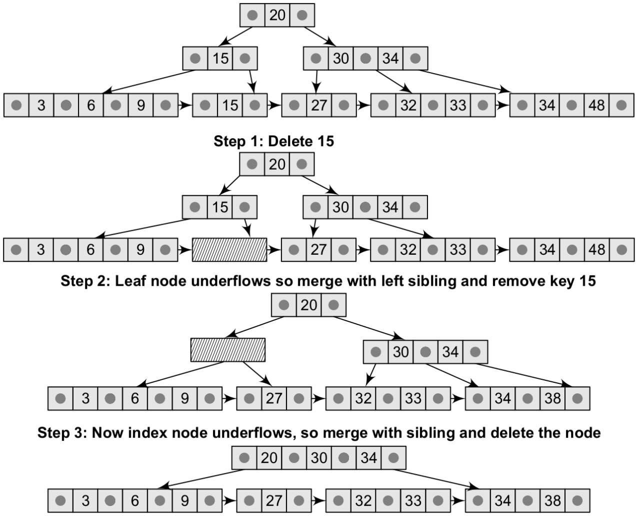
Deleting an Element from a B+ Tree

Algorithm to delete a node fro a B+ Tree

1. **Delete** the key and data from the *leaves*.
2. If the leaf node *underflows*, **merge** that node with the *sibling* and **delete** the *key in between them*.
3. If the index node *underflows*, **merge** that node with the *sibling* and **move down** the *key in between them*.

- If the leaf is empty after deleting, the neighbouring nodes are merged with the underfull node.
- An index value from the parent node may be, in turn, deleted as well.
- A merge-delete wave may be run on the path from a leaf node to the root.
    - This leads to shrinking of the tree by one level.

Deleting 15 in a B+ Tree of Order 4


Step 1: Delete 15
Step 2: Leaf node underflows so merge with left sibling and remove key 15
Step 3: Now index node underflows, so merge with sibling and delete the node

1. Traverse left-left to delete the key (**15**) and its data from the *leaf node*.

2. The leaf node underflows, so merge it with the left sibling and delete the key in between them (15) from the *index*.

3. The *index node* (old 15) underflows, so merge it with the right sibling and move down the key in between them (old root, 20).

   - The tree shrinks by one.

Other Trees

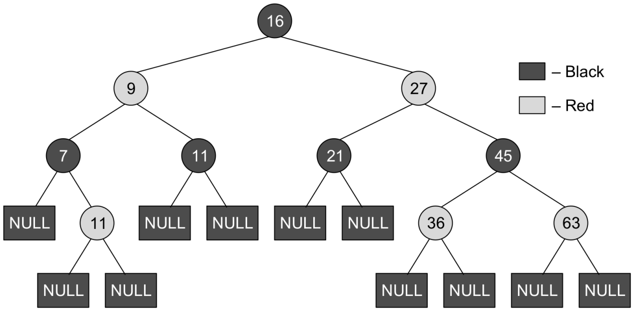
Red Black Trees

Definition

A red-black tree is a binary search tree, in which *every node has a color*, and that conforms to the additional requirements:

1. The color of a node is either red or black.
2. The color of the *root* node is always *black*.
3. All *leaf nodes* are *black*.
4. Every *red node* has both *children* colored in *black*.
5. Every *simple path* from a given node to any of its leaf nodes has *equal number of black nodes*.

Red Black Trees



In the example of a red-black tree:

- The root node is *black*.
- The leaf nodes are *black*.
- Every *red* node has both *children* colored in *black*.
- Every path from a node to a leaf has equal number of *black* nodes.

Red Black Trees

- These constraints enforce a critical property of red-black trees.
    - The longest path from the root node to any leaf node is no more than twice as long as the shortest path from the root to any other leaf in that tree.
- A red-black tree is a self-balancing binary search tree.
- It performs insertion, search and deletion operations in $O(\log n)$ time.
    - Red-black trees are valuable in time-sensitive applications such as real-time applications.
    - AVL tres are more rigidly balanced than red-black trees, thereby resulting in slower insertion and removal but faster retrieval of data.

Insertion in Red Black Trees

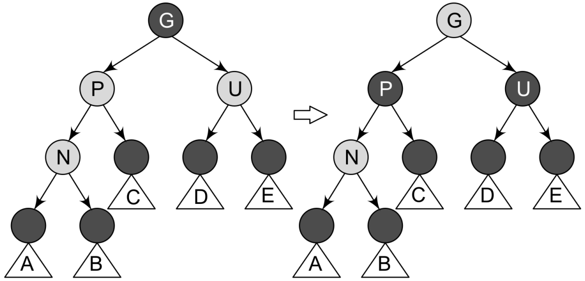
- The insertion starts in the same way as in BST
- However, instead of adding the new node as a leaf node, a red internal node is added that has two black leaf nodes.
    - Because in a red-black tree, leaf nodes contain no data.
- Insertion may violate the properties of a red-black tree.
    - To restore their property, we will check for certain cases and will do restoration of the property depending on the case that turns up after insertion.

Insertion in Red Black Trees

- **Case 1:** The new node N is added as the root of the tree.
    - N is repainted black
- **Case 2:** The new node's parent P is black
- **Case 3:** If both the parent (P) and the uncle (U) are red
- **Case 4:** The parent P is red but the uncle U is black and N is the right child of P and P is the left child of G.
- **Case 4:** The parent P is red but the uncle U is black and N is the right child of P and P is the left child of G.

Case 3: If both the parent (P) and the uncle (U) are red

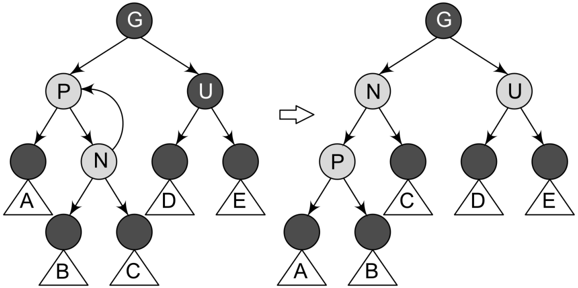
- Both nodes (P and U) are repainted black and the grandparent G is repainted red.
- The grandparent G may now violate property 2 which says that the root node is always black or property 4 which states that both children of every red node are black.
    - To fix this problem, this entire procedure is recursively performed on G from case 1.



Insertion in Red Black Trees

Case 4: The parent P is red but the uncle U is black and N is the right child of P and P is the left child of G.

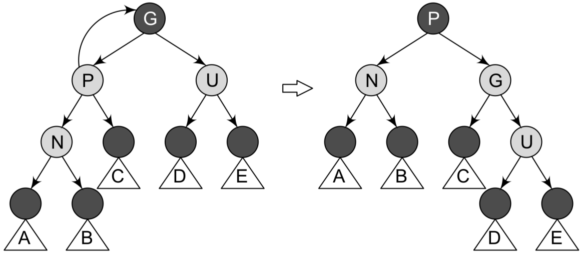
- A left rotation is done to switch the roles of the new node N and its parent P.
- Then case 5 is called to deal with the new node's parent.
    - Because property 4 which says both children of every red node should be black is still violated.



Insertion in Red Black Trees

Case 5: The parent P is red but the uncle U is black and the new node N is the left child of P, and P is the left child of its parent G.

- A right rotation on G (the parent of parent of N) is performed.
    - Now the former parent P is now the parent of both the new node N and the former grandparent G.
- Switch the colors of P and G to satisfy property 4 which states that both children of a red node are black.



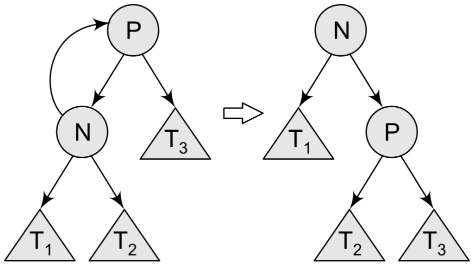
Splay Trees

- A splay tree is a **self-balancing binary search** tree with an *additional property* that *recently accessed elements* can be re-accessed fast.
- It performs insertion, search and deletion operations in $O(\log n)$ time.
- For many *non-uniform sequences of operations*, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
- This advantage is particularly useful for implementing caches and garbage collection algorithms.

Splay Trees

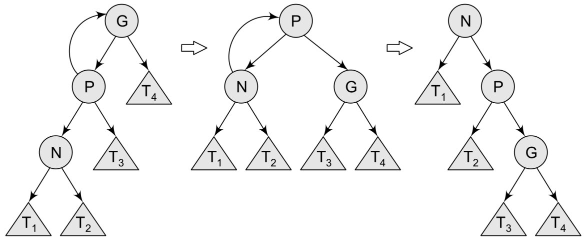
- When a node in a splay tree is accessed, it is rotated or ”splayed” to the root.
- Since the most frequently accessed nodes are always moved closer to the starting point of the search (or the root node), those nodes are located faster.
- Operations like insertion, search and deletion are combined with one basic operation called *splaying*.

Splaying



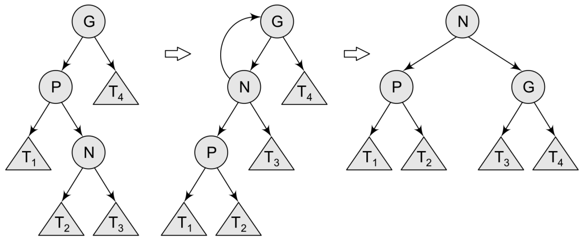
- **Zig step:** is done when P (the parent of N) *is the root* of the splay tree.
- It is usually performed as the last step in a splay operation and only when N has an odd depth at the beginning of the operation.
- The tree is rotated on the edge between N and P.

Splaying



- **Zig-zig Step**: is performed when P *is not the root.*
- N and P are either both right children or are both left children of their parent's.
- First the tree is rotated on the edge joining P and its parent G, and then again rotated on the edge joining N and P.

Splaying



- **Zig-zag Step:** is performed when P *is not the root.*
- In addition to this, N is a right child of P and P is a left child of G or vice versa.
- The tree is first rotated on the edge between N and P, and then rotated on the edge between P and G.

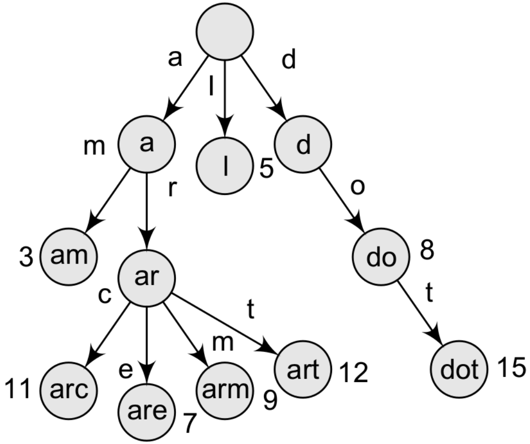
Trie

Definition

- The *position* of a node in the trie *represents the key* associated with that node instead of storing the key directly.
- All the descendants of a node have *a common prefix of the string* associated with that node.
- The root is associated with the empty string.

- The term trie has been taken from the word ’retrieval’.
- Trie stores keys that are usually strings
- Searching time:
  - The worst case: $O(m)$, m is the maximum string length
  - The average case: $O(\log_m n)$, n is the total number of keys and m is the maximum string length

Trie data structure



- Keys are listed in the nodes and the values below them.
- Any path from the root to a leaf represents a word.
- Each complete English word is assigned an arbitrary integer value.
- Following a path within the trie yields the associated Value for the given string key.

Applications of Tries

- A trie is very commonly used to store a dictionary (for ex, on a mobile telephone).
- These applications take advantage of a trie's ability to quickly search, insert, and delete entries.
    - In a balanced BST, search may take $O(m \log_2 n)$ in the worst case, where n is the total number of nodes and m *is* the maximum string length.
- Tries are also used to implement approximate matching algorithms, including those used in spell checking software.

QUESTIONS ❓