CS 2124: DATA STRUCTURES
Spring 2024

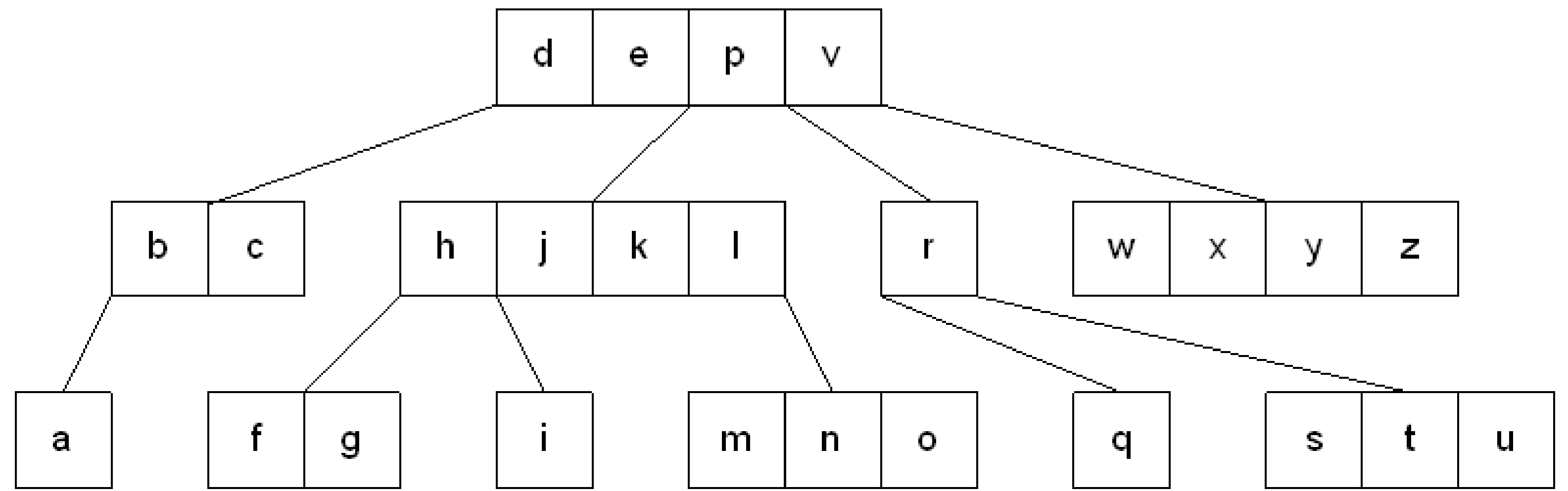
Topics: **B-Trees**

Topics

- Multiway Tree (M-way tree)
    - Multiway Search Tree
- B-Tree (Height-balanced m-way tree/Balanced Tree)
    - Application of B-tree
    - B-Tree Properties
    - B-Tree Order
    - Creating a B-Tree
    - B-Tree Operations
    - Advantages and Disadvantages of B-tree
- RB Tree (Red Black Tree)
    - RB Tree Properties
    - Creating RB Tree
    - RB Tree Operations
    - RB Tree Rotations

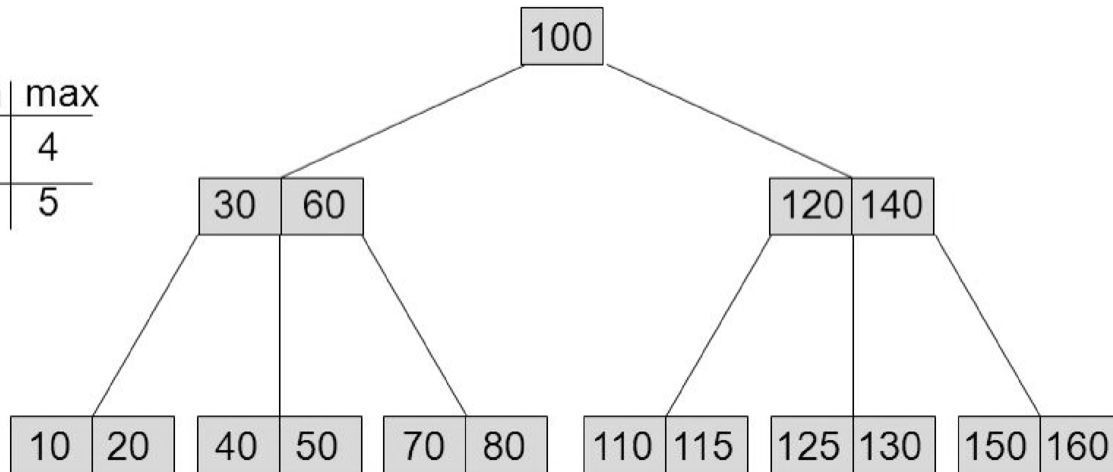
Multiway Trees (M-way Tree)

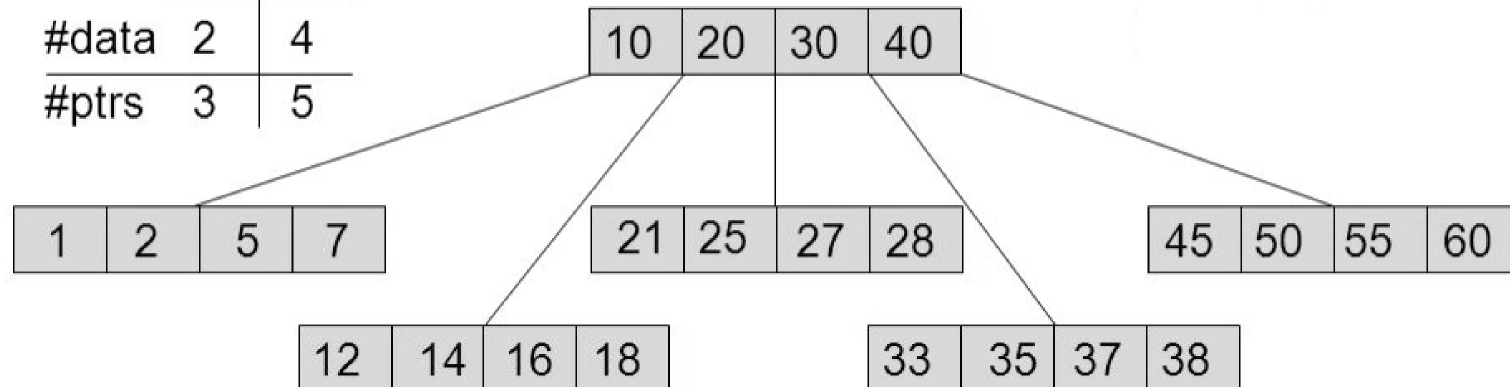
- A multiway tree is a tree that can have more than two children/nodes.
- A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.
- As with the other trees that have been studied, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.
- Example: Following is a multiway tree of order 5
  1. All leaf nodes are at the same level.
  2. All non-leaf nodes (except the root) have at most 5 and at least 2 children.



Source

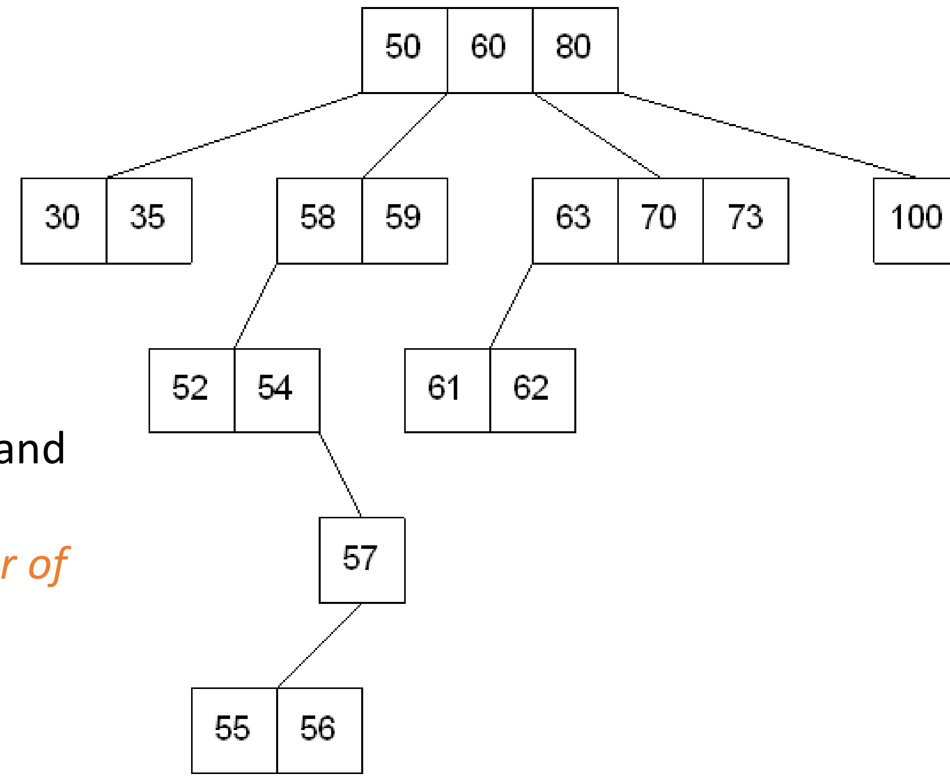
Multiway Trees (M-way Tree of order 5)





Multiway Search Trees

- M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees they provide fast information retrieval and update.
- However, they also have the same problems that binary search trees had they can become **unbalanced,** which means that the construction of the tree becomes of vital importance.



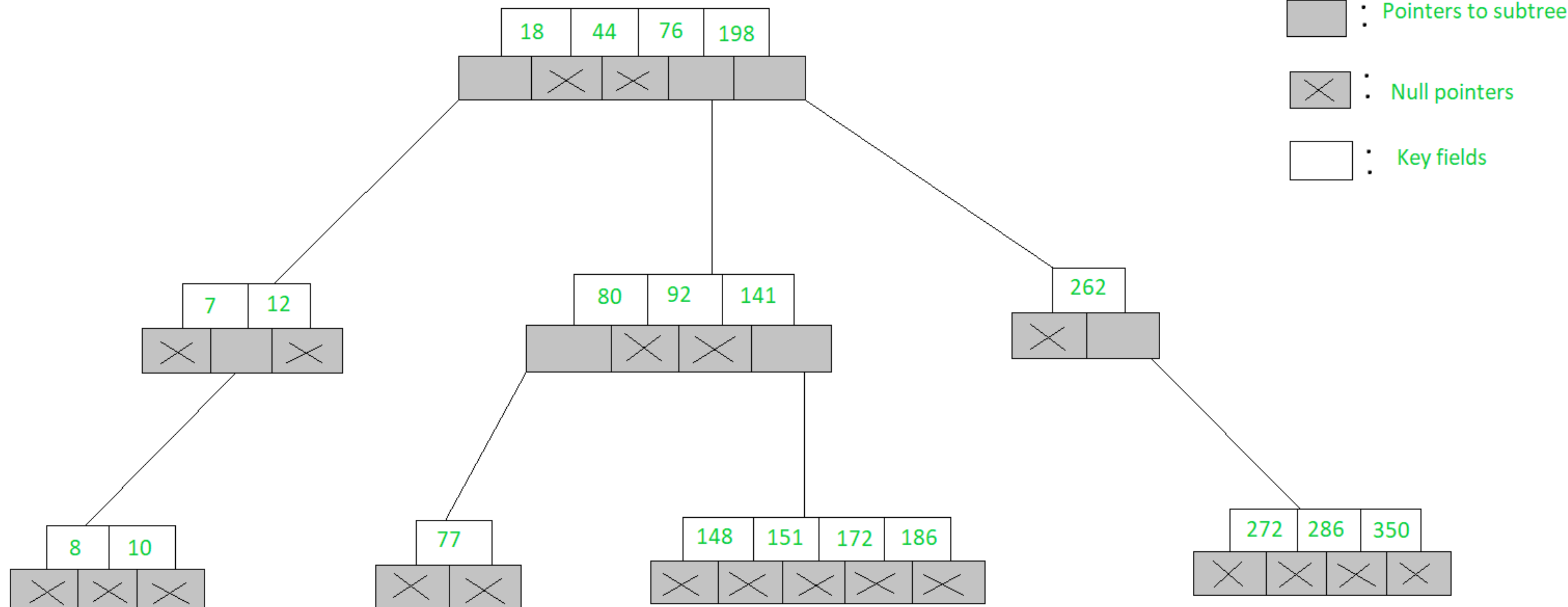
- When a binary tree is balanced, the elements can be accessed, inserted, and removed in the most efficient way possible.
- This is because the depth of the tree is minimized, which *reduces the number of comparisons needed to find a specific node.*

Multiway Search Trees

- The m-way search trees are multi-way trees which are generalized versions of binary trees where each node contains multiple elements.
- In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.
- An example of a 5-Way search tree is shown in the figure below.
- Observe how each node has at most 5 child nodes & therefore has at most 4 keys contained in it.

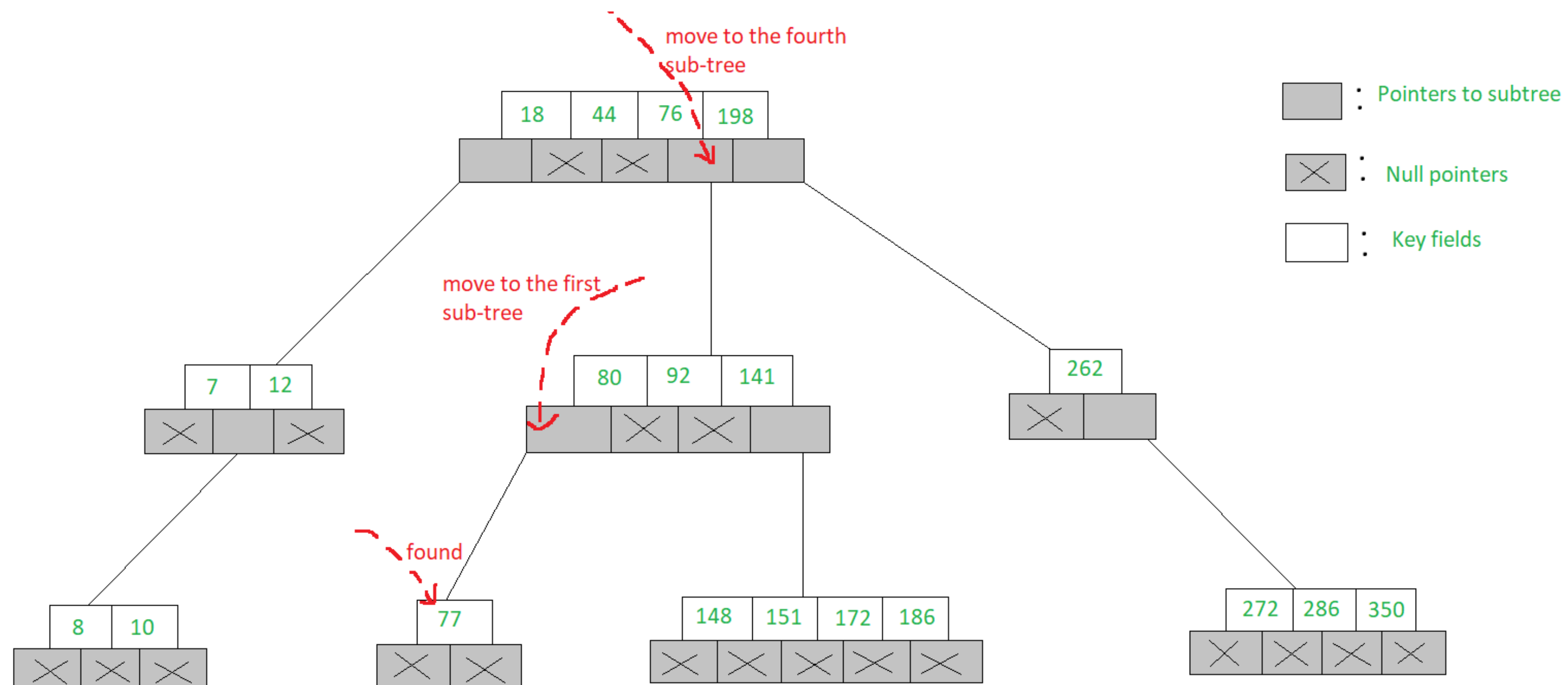
M=5

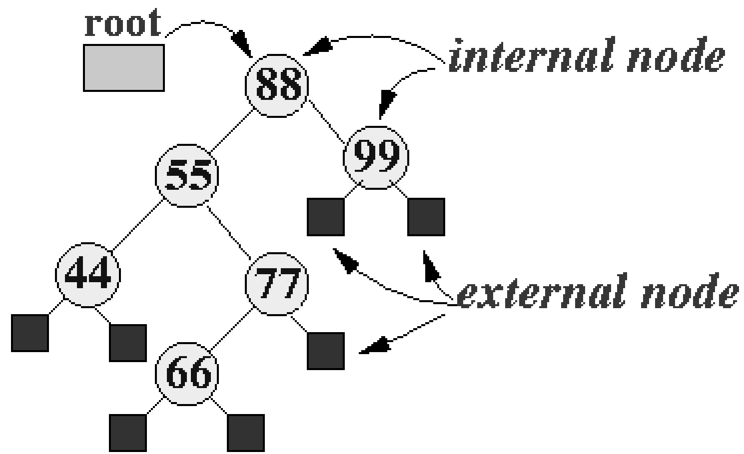
	min	max
#data	2	4
#ptrs	3	5



Multiway Search Trees

- Searching for a key in an m-Way search tree is similar to that of binary search tree
- To search for 77 in the 5-Way search tree (shown in the figure). We begin at the root & as 77 > 76 > 44 > 18, move to the fourth sub-tree
- In the root node of the fourth sub-tree, 77 < 80 & therefore we move to the first sub-tree of the node.
- Since 77 is available in the only node of this sub-tree, we claim 77 was successfully searched



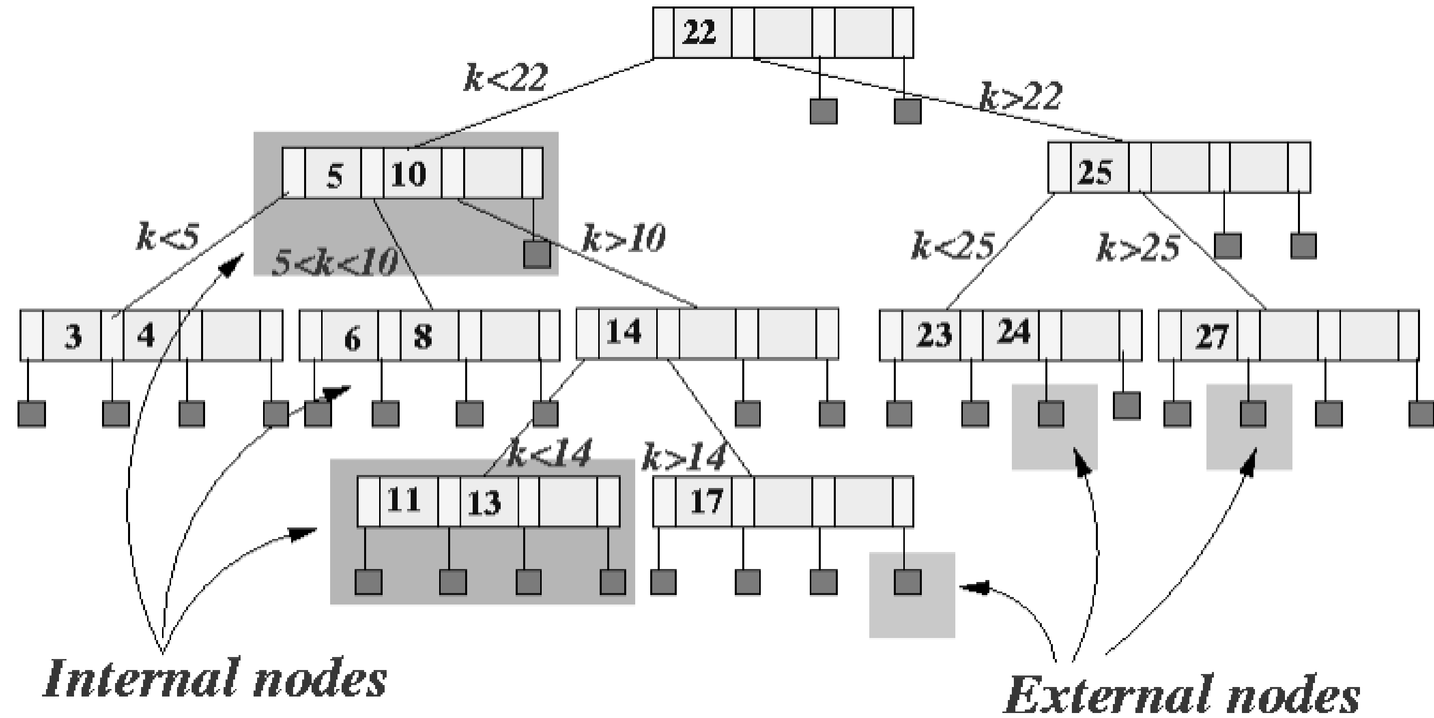


A **BST** is an ordered tree where each internal node has at most 2 children nodes

- Each internal node has exactly 2 (internal or external) child nodes
- External nodes have 0 children nodes (they are "terminal nodes")
- External nodes are often simply a null value

A **multiway (m-way)** search tree is a generalization of the binary search tree.

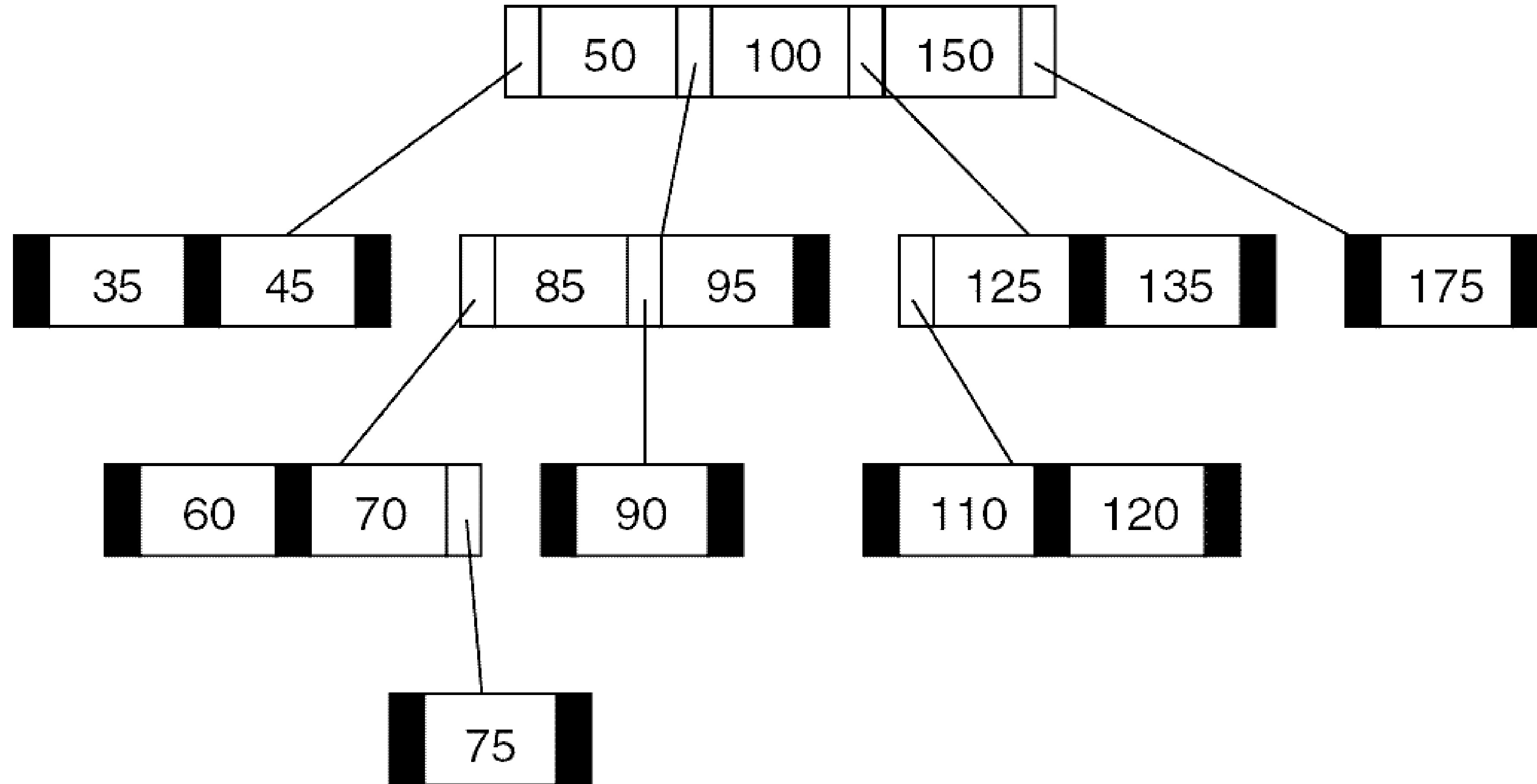
- Each internal node of a m-way tree has exactly m (internal or external) children nodes
- External nodes have 0 children nodes (they are "terminal nodes")
- External nodes are often simply a null value



4-way tree

- Each internal node has 4 (internal or external) child nodes
- External nodes has no children

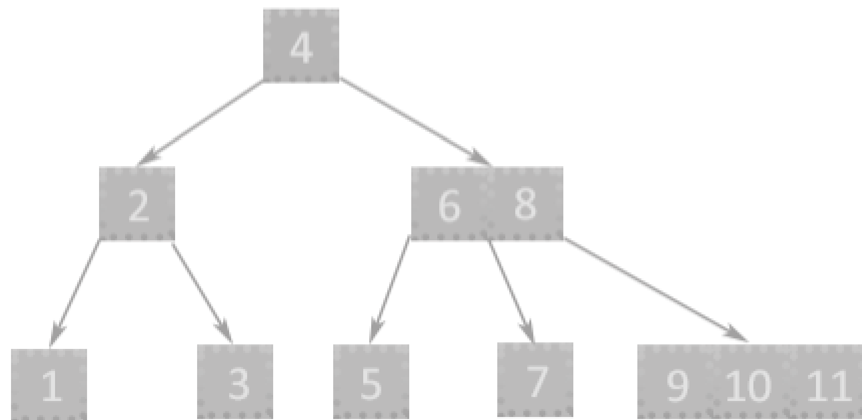
Multiway Search Trees

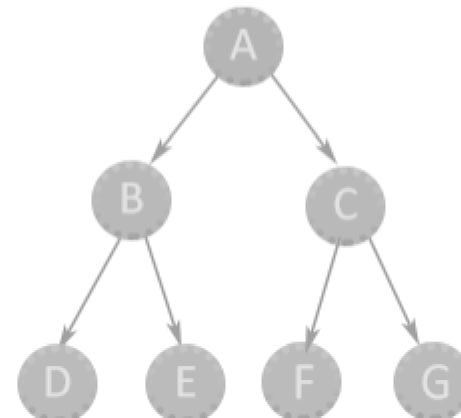


Multi-way search tree – order 4

B-Tree Introduction

- The limitations of traditional binary search trees (BST) can be frustrating (i.e. storing data, imbalance issue etc.).
- Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease.
- When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage.
- B-Trees, also known as Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

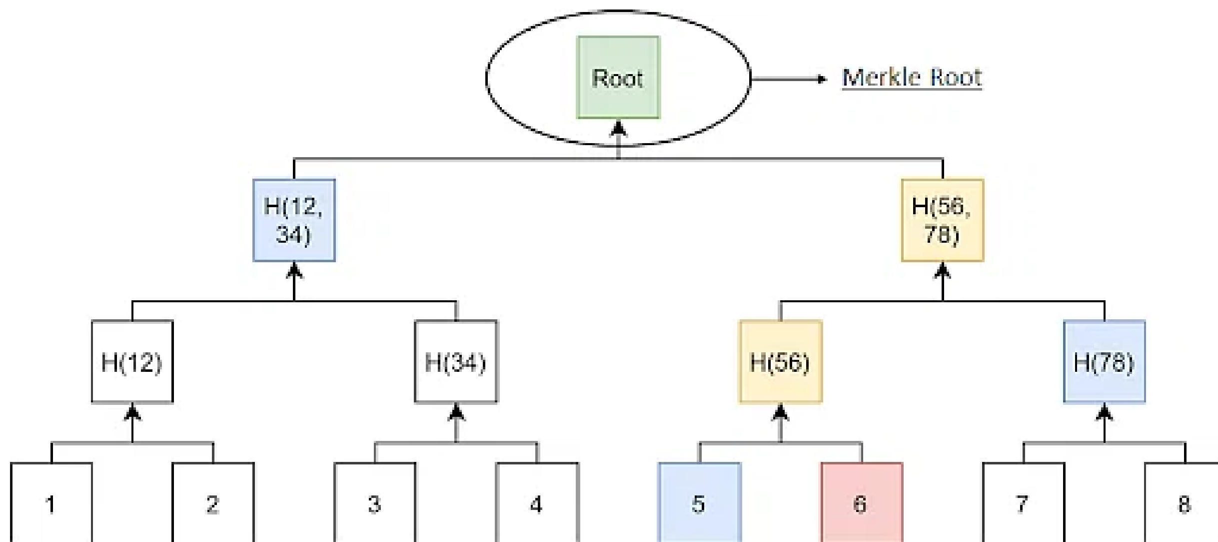




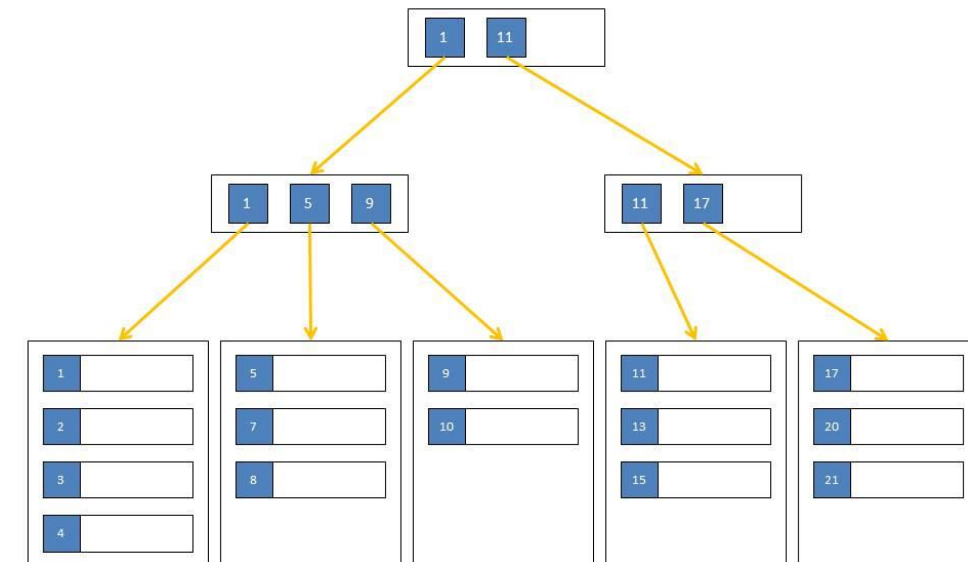
Source

Applications of B-Trees

1. It is used in large databases to access data stored on the disk
2. Searching for data in a data set can be achieved in significantly less time using the B-Tree
3. With the indexing feature, multilevel indexing can be achieved.
   a) Most of the servers also use the B-tree approach.
4. B-Trees are used in CAD (computer-aided design) systems to organize and search geometric data.
5. B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

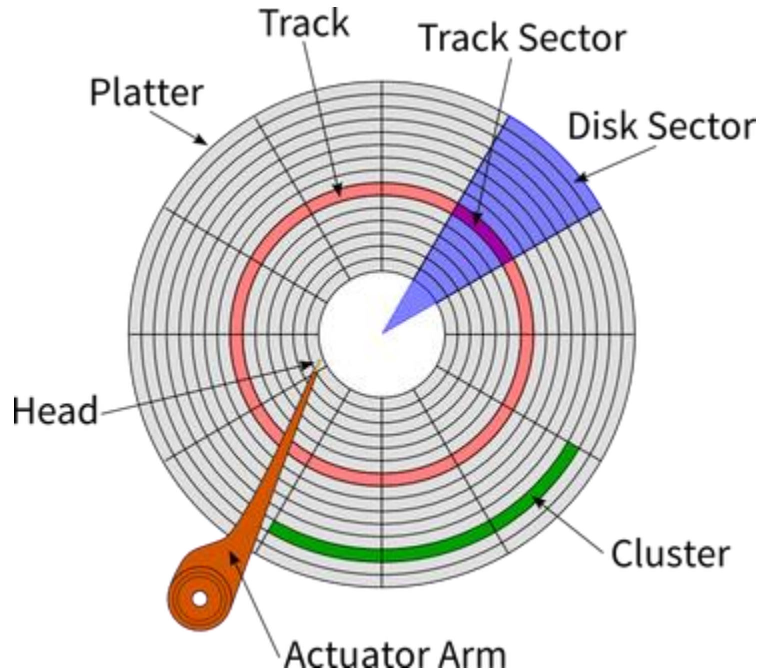


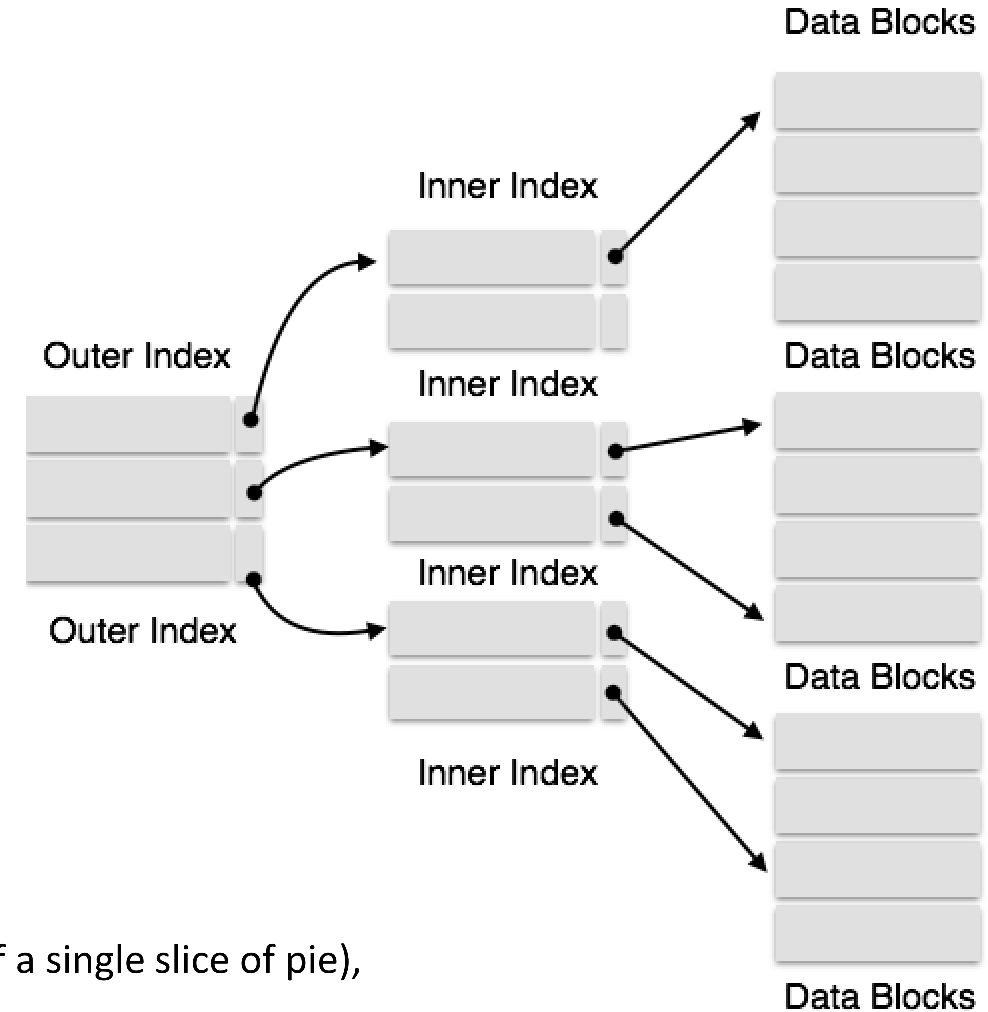
Cryptography (Hashing) using B-trees



SQL server indexing using B-trees

Applications of B-Trees

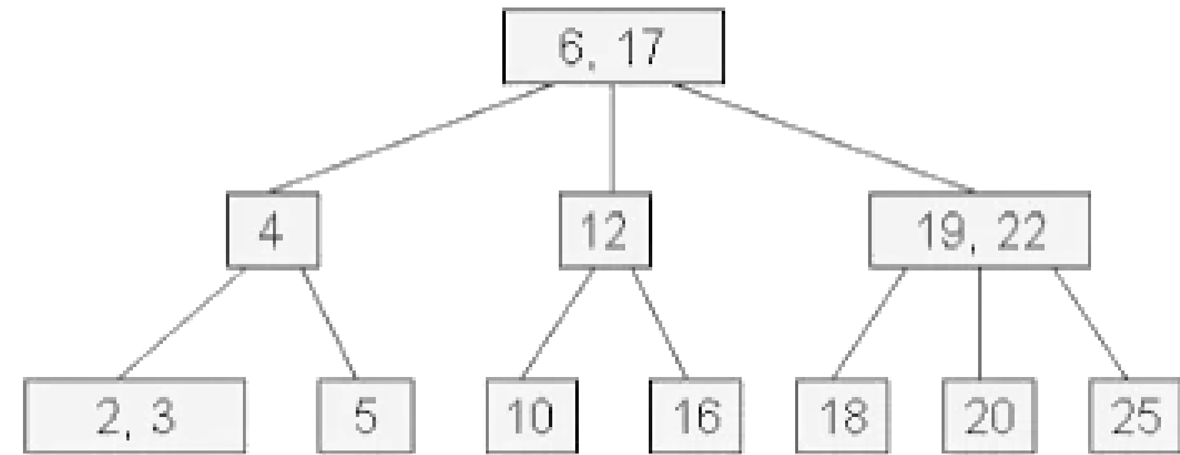




- Disks are made up of tracks (a circular ring, like the outer crust of a pie),
- Disk sectors (a section of the disk, like a single slice of pie),
- A track sector or block (the intersection of a track and a sector, like the outer crust of a single slice of pie),
- And an actuator arm.
- A block is the smallest storage unit on a disk and is typically 512 bytes in size.

Note: Not part of quiz/exam

B-Tree (Properties)

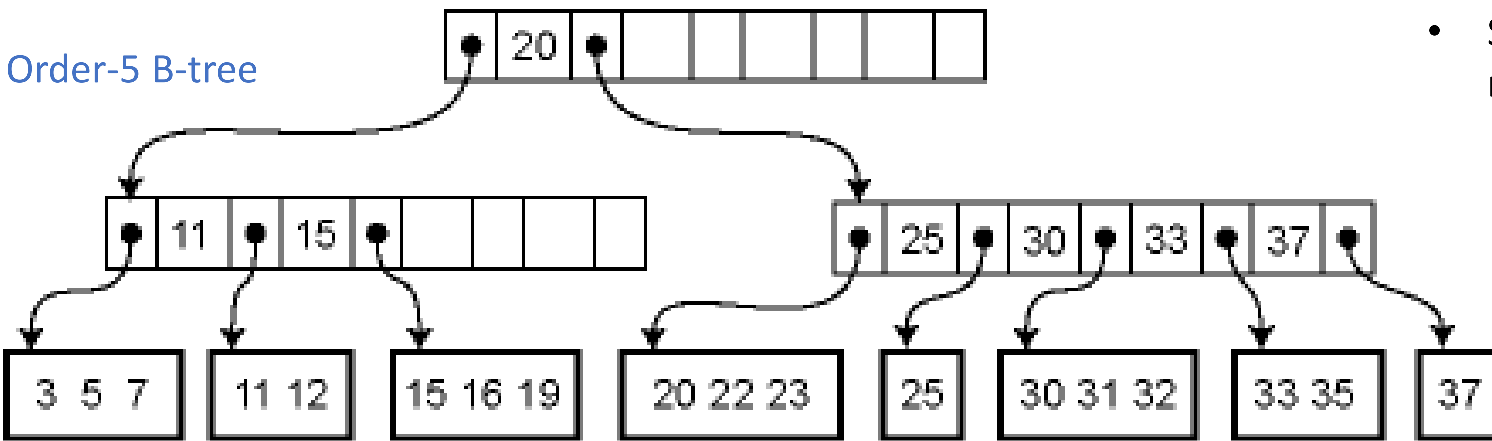


B-Tree of order 3

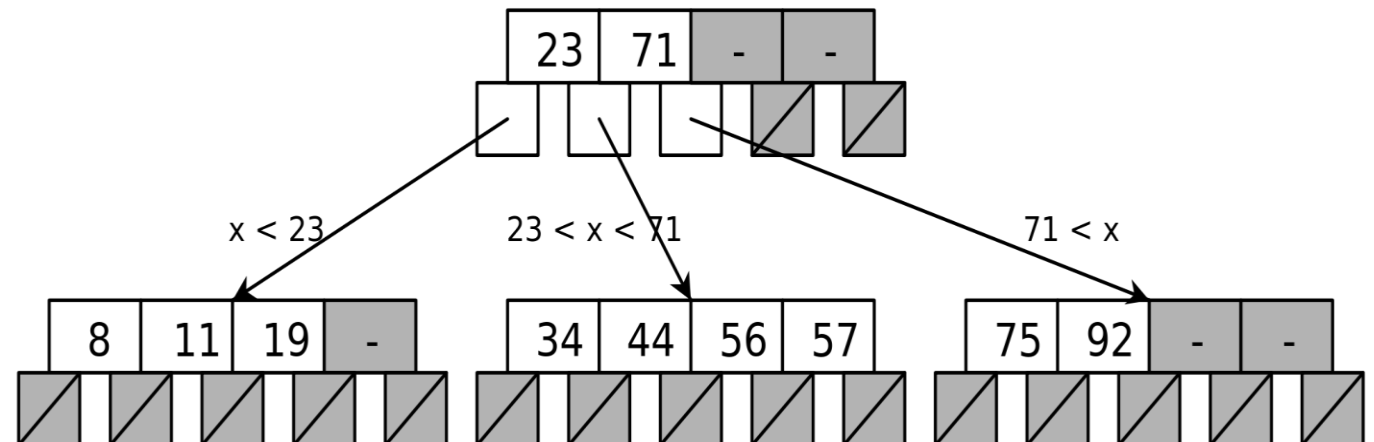
1. All leaves are at the same level.
2. B-Tree is defined by the term minimum degree ‘t‘.
   a) The value of ‘t‘ depends upon disk block size.
3. Every node except the root must contain at most t-1 keys.
   a) The root may contain a minimum of 1 key.
4. Number of children of a node is equal to the number of keys in it plus 1.
5. All keys of a node are sorted in increasing order.
   a) The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
6. B-Tree grows and shrinks from the root which is unlike Binary Search Tree.
   a) Binary Search Trees grow downward and also shrink from downward.
7. Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
8. Insertion of a Node in B-Tree happens only at Leaf Node.

B-Tree (Order)

- For example, the following is an order-5 B-tree (m=5) where the leaves have enough space to store up to 3 data records:



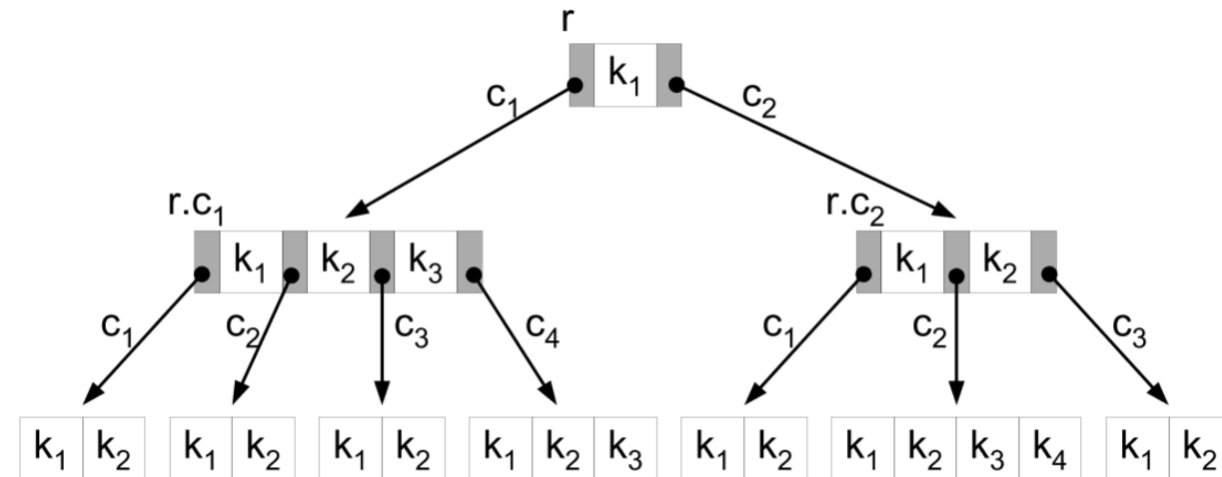
- The **order** of the tree represents the maximum number of children a tree's node could have.
- So when we say we have a B-Tree of order , It means every node of that B-Tree can have a maximum of. children.



Source

B-Tree

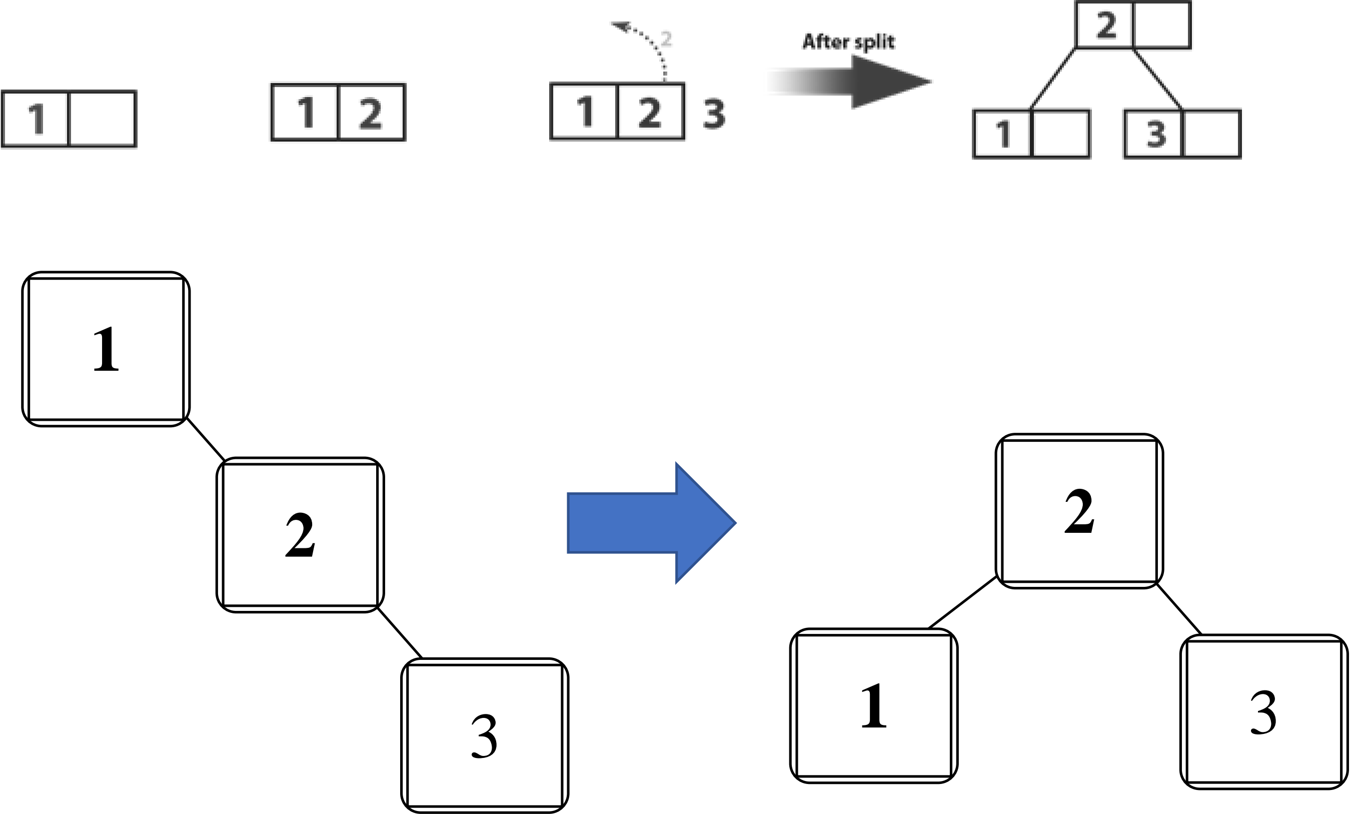
- Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height.
- This shallow height leads to less disk I/O, which results in faster search and insertion operations.
- B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory etc.
- B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced.
- This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.



Creating a B-Tree

Create B-tree of order 3
Elements: 1 to 10

Every node except the root must contain at most t-1 keys

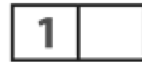


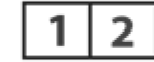
Creating a B-Tree

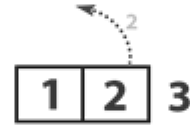
Create B-tree of order 3
Elements: 1 to 10

Every node except the root must contain at most t-1 keys

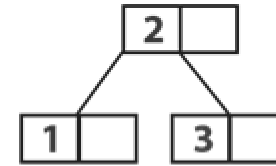
Insertion of a Node in B-Tree happens only at Leaf Node

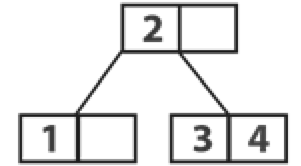




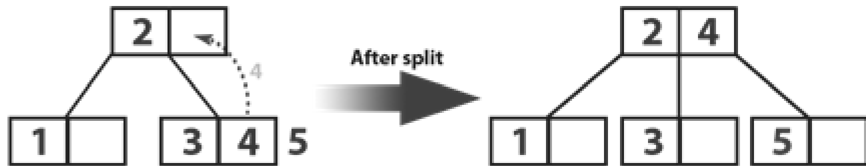


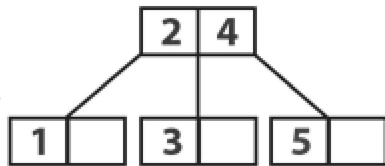
After split



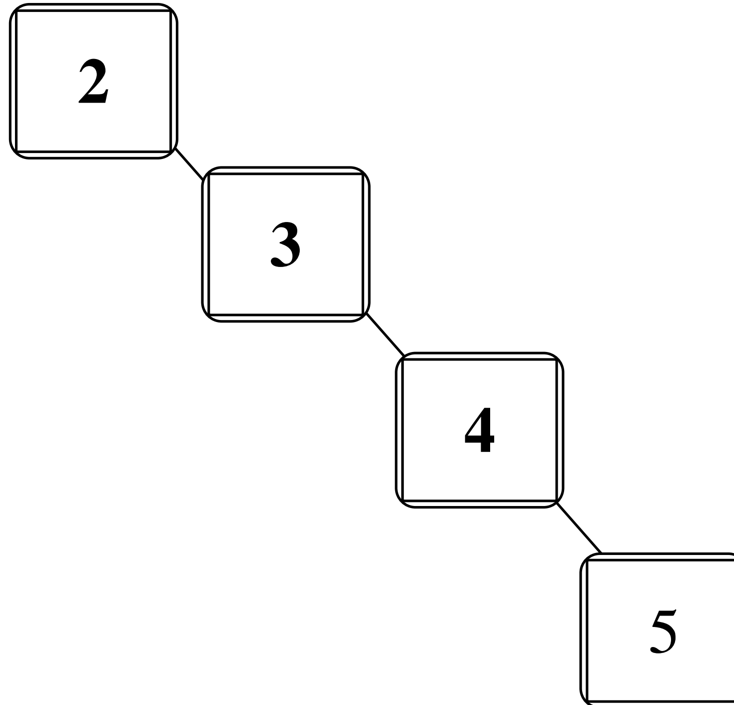


All keys of a node are sorted in increasing order.





Number of children of a node is equal to the number of keys in it plus 1.



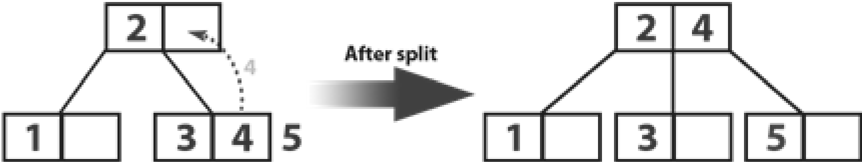
Creating a B-Tree

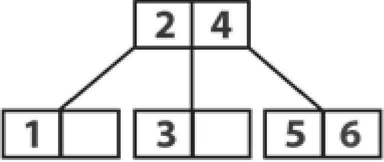
Create B-tree of order 3
Elements: 1 to 10

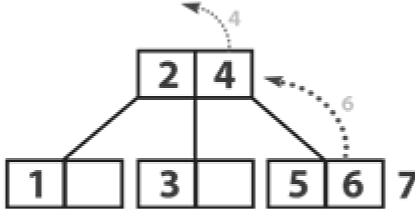
Every node except the root
must contain at most t-1 keys

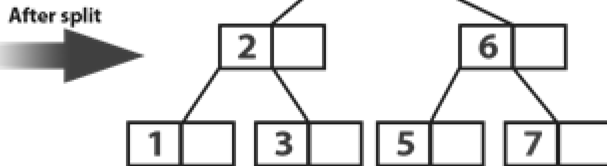
Insertion of a Node in B-Tree
happens only at Leaf Node

All keys of a node are sorted in increasing order.



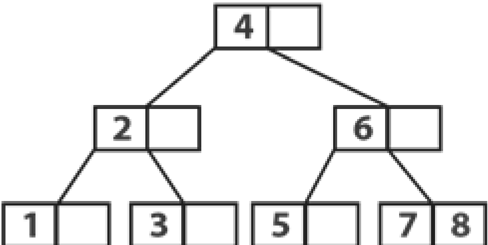


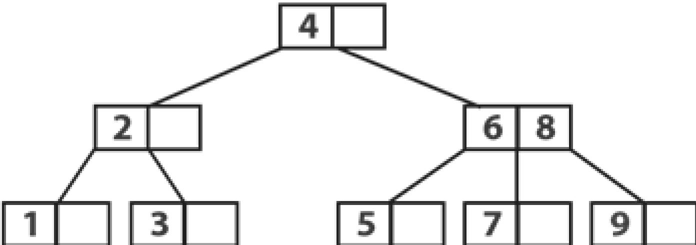




Number of children of a node is equal
to the number of keys in it plus 1.

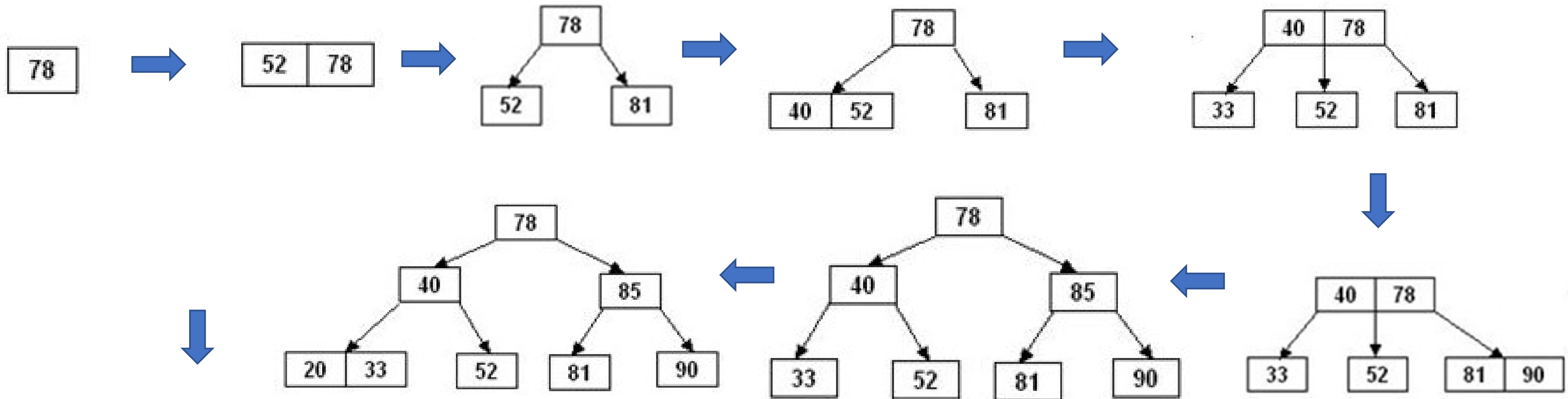
Grows and shrinks from the root





Creating a B-Tree

Create B-tree of order 3
Elements: 78, 52, 81, 40, 33, 90, 85, 20, 38



Website to visualize the creation of B-tree: Link

B-Tree

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}
```

Note: This Code is not part of quiz/exam

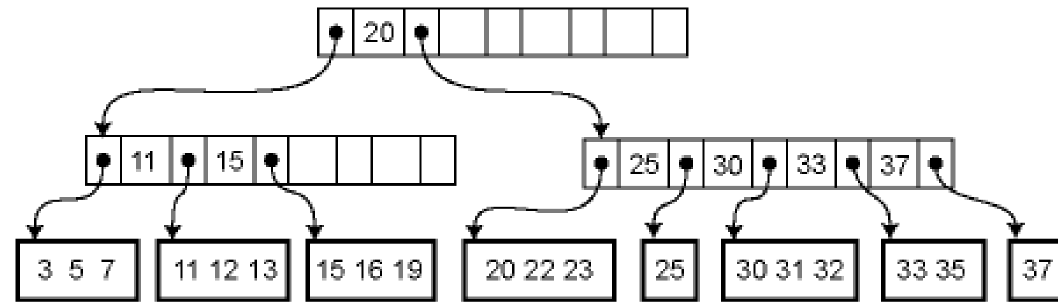
Source

B-Tree (Introduction to operations)

- **Lookup** in a B-tree is straightforward.
  - Given a node to start from, we use a simple linear or binary search to find whether the desired element is in the node, or if not, which child pointer to follow from the current node.
- **Insertion** and **deletion** from a B-tree are more complicated; in fact, they are notoriously difficult to implement correctly.
- **Insertion:**
  - We first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree).
  - If there is already room in the node, the new element can be inserted simply.
  - Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element.
  - The parent is then updated to contain a new key and child pointer.
  - If the parent is already full, the process ripples upwards, eventually possibly reaching the root.
  - If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one.

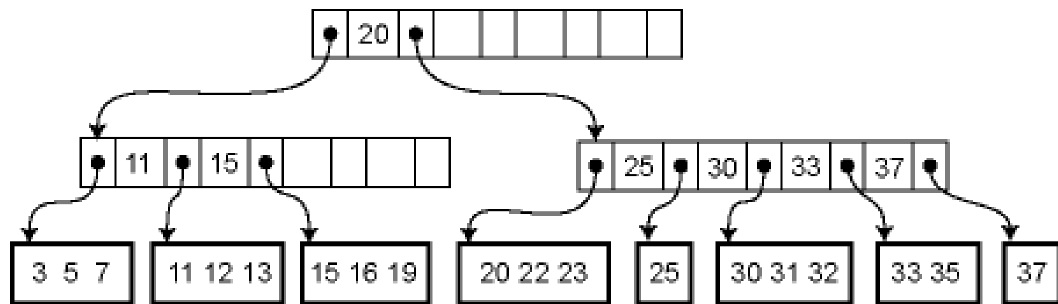
B-Tree (Insertions)

- For example, here is the effect of a series of **insertions**.

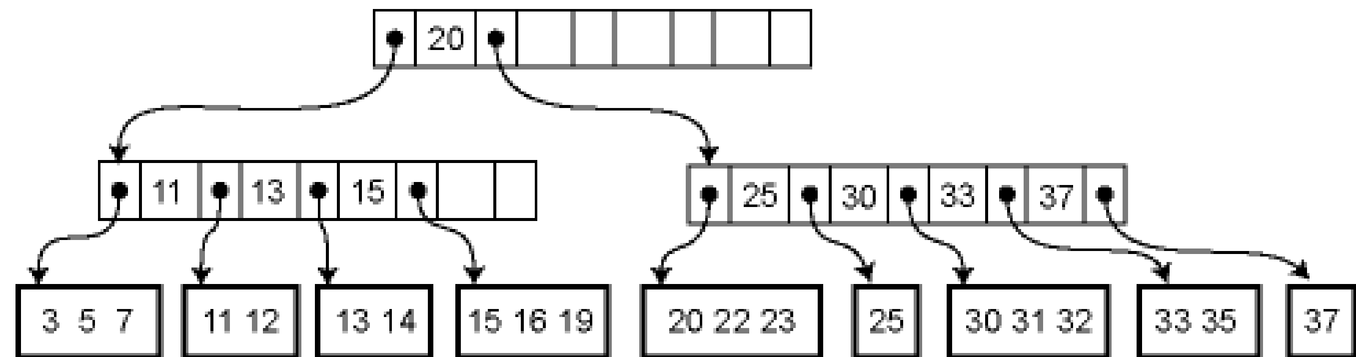


B-Tree (Insertions)

- For example, here is the effect of a series of **insertions**.
- The first insertion (13) merely affects a leaf.

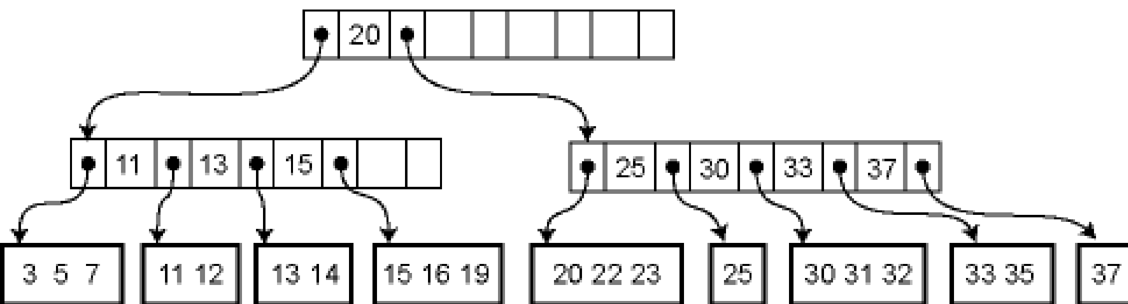


Every node except the root must contain at most t-1 keys



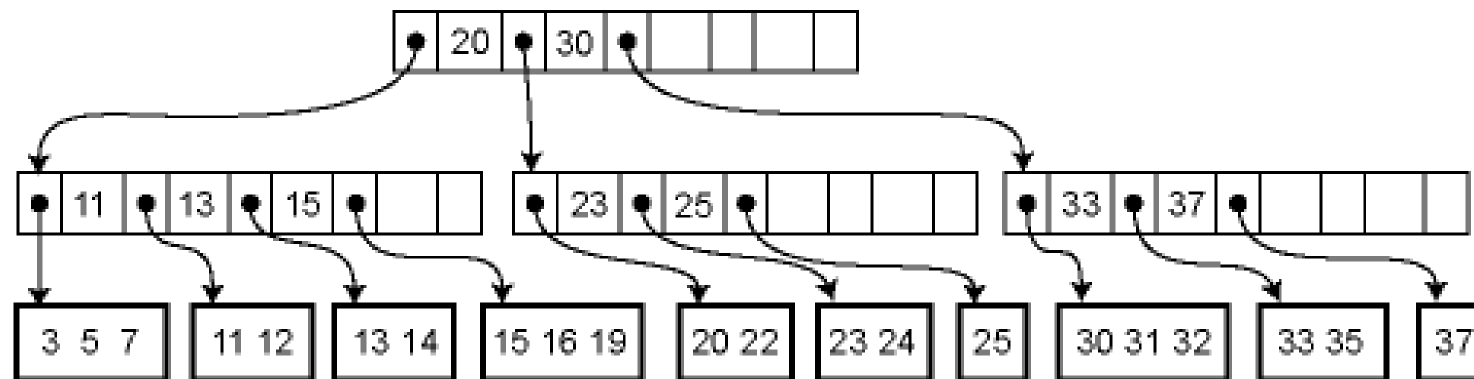
B-Tree (Insertions)

- For example, here is the effect of a series of **insertions**.
- The first insertion (13) merely affects a leaf.
- The third insertion (24) propagates all the way to the root.



Adding 24 to the root:
Keys in root = its child nodes

Every node except the root must contain at most t-1 keys



Steps on next slide >>>

B-Tree (Insertions)

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
- **Step 3 -** If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- **Step 4 -** If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- **Step 5 -** If that leaf node is already full, **split** that leaf node by sending middle value to its parent node.
  - Repeat the same until the sending value is fixed into a node.
- **Step 6 -** If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

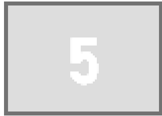
B-Tree (Insertions)

- **Node does not consist of more than the MAXIMUM number of keys** - We will insert the key in the node at its proper place.
- **A node consists of a MAXIMUM number of keys** - We will insert the key to the full node, and then a split operation will take place, splitting the full node into three parts.
- The second or median key will move to the parent node, and the first and the third parts will act as the left and right child nodes, respectively.
- This process will be repeated with the parent node if it also consists of a MAXIMUM number of keys.

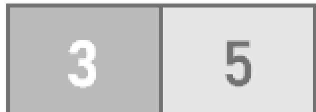
B-Tree (Insertions)

- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.

- Since m is equal to 3, the maximum number of keys for a node = m-1 = 3-1 = 2.
- We will start by inserting 5 in the empty tree.

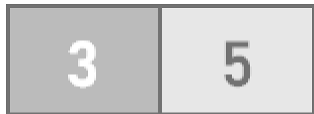


We will now insert 3 into the tree. Since 3 is less than 5, we will insert 3 to the left of 5 in the same node.

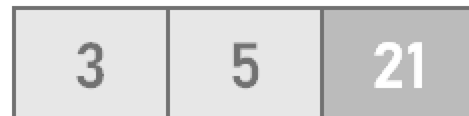


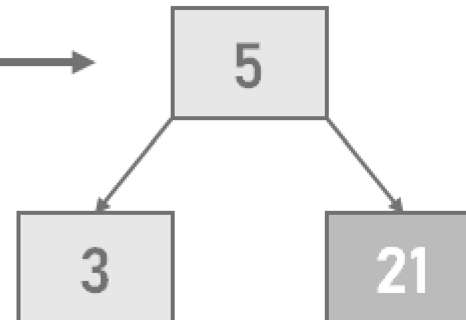
B-Tree (Insertions)

- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.



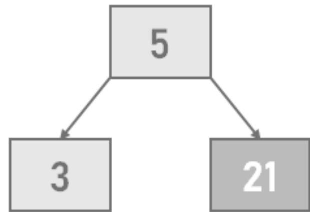
- We will now insert 21 into the tree, and since 21 is greater than 5, it will be inserted to the right of 5 in the same node.
- However, as we know that the maximum number of keys in the node is 2, one of these keys will be moved to a node above in order to split it.
- Thus, 5, the middle data element, will move up, and 3 and 21 will become its left and right nodes, respectively.





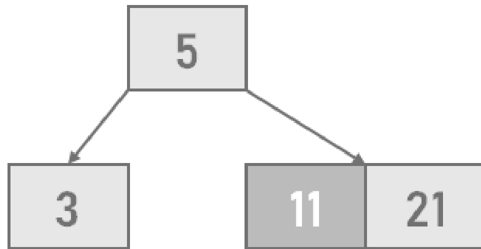
B-Tree (Insertions)

- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.



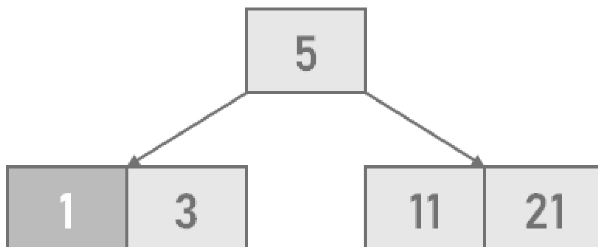
- Now it is time to insert the next data element, i.e., 11, which is greater than 5 but less than 21.
- Therefore, 11 will be inserted as a key on the left of the node consisting of 21 as a key

Insert 11



- Similarly, we will insert the next data element 1 into the tree, and as we can observe, 1 is less than 3; therefore, it will be inserted as a key on the left of the node consisting of 3 as a key.

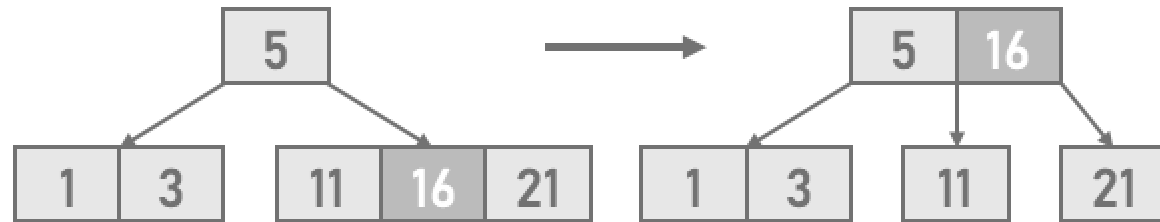
Insert 1

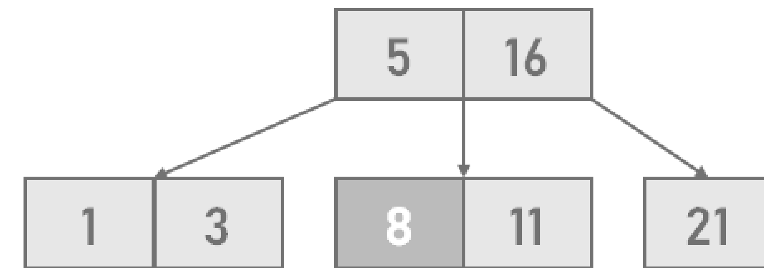


- Now, we will insert data element 16 into the tree, which is greater than 11 but less than 21.
- However, the number of keys in that node exceeds the maximum number of keys.
- Therefore, the node will split, moving the middle key to the root.
- Hence, 16 will be inserted to the right of the 5, splitting 11 and 21 into two separate nodes.

B-Tree (Insertions)

- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.

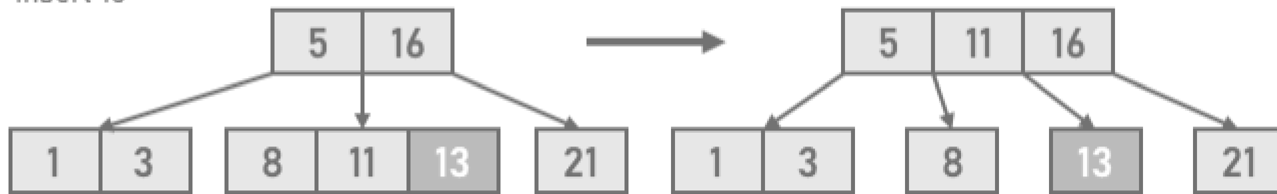


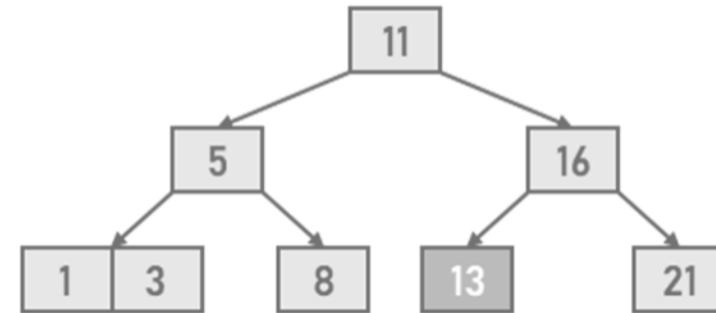


- We will insert 13 into the tree, which is less than 16 and greater than 11.
- Therefore, data element 13 should be inserted to the right of the node consisting of 8 and 11.
- However, since the maximum number of keys in the tree can only be 2, a split will be occurred, moving the middle data element 11 to the above node and 8 and 13 into two separate nodes.
- Now, the above node will consist of 5, 11, and 16, which again exceeds the maximum key count.
- Therefore, there will be another split, making the data element 11 the root node with 5 and 16 as its children

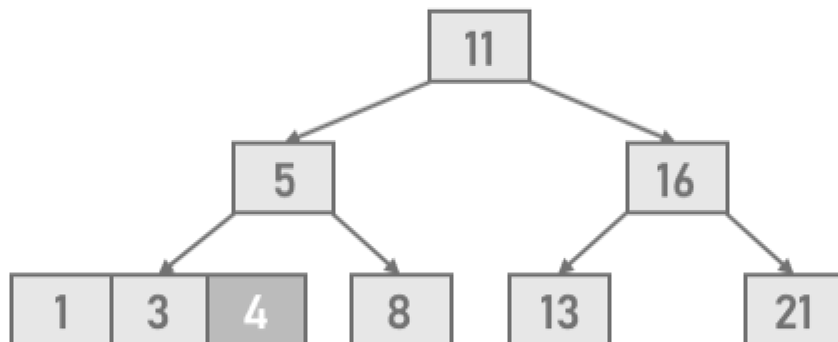
B-Tree (Insertions)

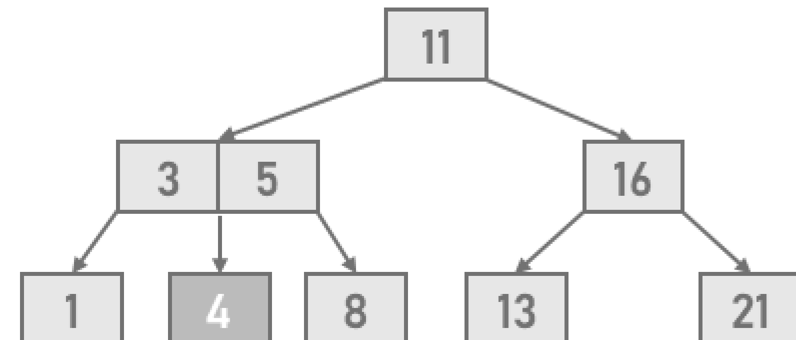
- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.





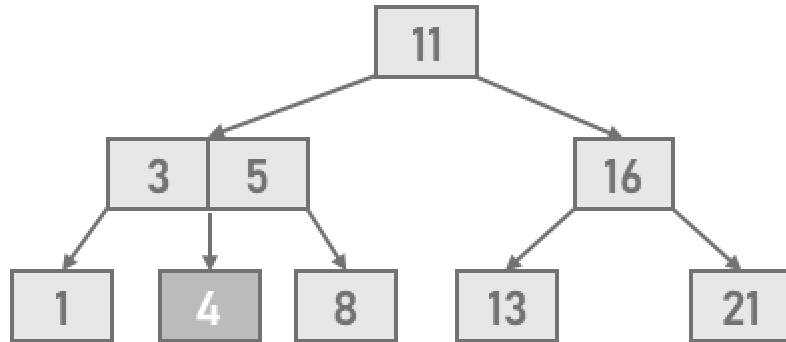
- Since the data element 4 is less than 5 but greater than 3, it will be inserted to the right of the node consisting of 1 and 3, which will exceed the maximum count of keys in a node again.
- Therefore, a spill will occur again, moving the 3 to the upper node beside 5.



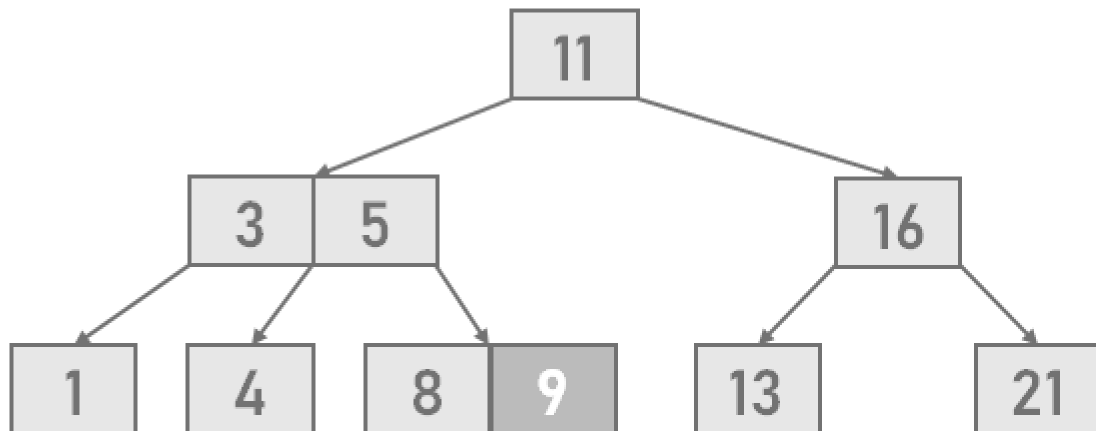


B-Tree (Insertions)

- Create a B Tree of order 4. The data elements needed to be inserted into the B Tree are 5, 3, 21, 11, 1, 16, 8, 13, 4,and 9.



- At last, the data element 9, which is greater than 8 but less than 11, will be inserted to the right of the node consisting of 8 as a key.



B-Tree (Insertions)

```c
// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;
}
```

```c
// Split node
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
         struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;
  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;
  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}
```

Note: This Code is not part of quiz/exam

B-Tree (Insertions)

```c
69  // Set the value
70  int setValue(int val, int *pval,
71               struct BTreeNode *node, struct BTreeNode **child) {
72    int pos;
73    if (!node) {
74      *pval = val;
75      *child = NULL;
76      return 1;
77    }
78    if (val < node->val[1]) {
79      pos = 0;
80    } else {
81      for (pos = node->count;
82         (val < node->val[pos] && pos > 1); pos--)
83        ;
84      if (val == node->val[pos]) {
85        printf("Duplicates are not permitted\n");
86        return 0;
87      }
88    }
89    if (setValue(val, pval, node->link[pos], child)) {
90      if (node->count < MAX) {
91        insertNode(*pval, pos, node, *child);
92      } else {
93        splitNode(*pval, pval, pos, node, *child, child);
94        return 1;
95      }
96    }
97    return 0; }
```

```c
99  // Insert the value
100 void insert(int val) {
101   int flag, i;
102   struct BTreeNode *child;
103
104   flag = setValue(val, &i, root, &child);
105   if (flag)
106     root = createNode(i, child);
107 }
```

```c
131 // Traverse then nodes
132 void traversal(struct BTreeNode *myNode) {
133   int i;
134   if (myNode) {
135     for (i = 0; i < myNode->count; i++) {
136       traversal(myNode->link[i]);
137       printf("%d ", myNode->val[i + 1]);
138     }
139     traversal(myNode->link[i]);
140   }
141 }
```

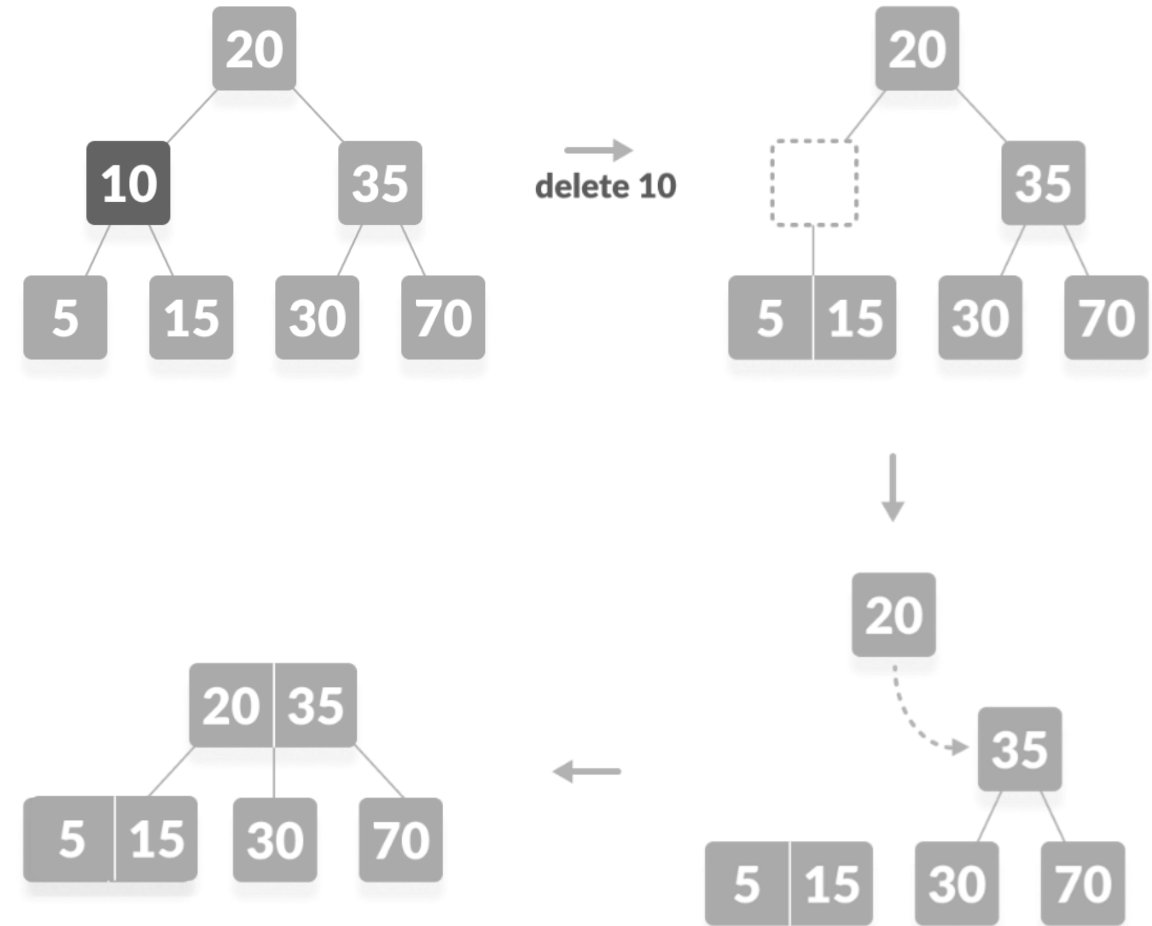
Note: This Code is not part of quiz/exam

B-Tree (Operations)

- **Deleting:**
- An element on a B-tree consists of three main events.
    1. Searching the node where the key to be deleted exists,
    2. Deleting the key and balancing the tree if required.
    3. While deleting a tree, a condition called underflow may occur.
- Underflow occurs when a node contains less than the minimum number of keys it should hold.
    - i.e. Number of children of a node is equal to the number of keys in it plus 1.

B-Tree (Deletion)

- **Deletion** works in the opposite way: the element is removed from the leaf.
- If the leaf becomes empty, a key is removed from the parent node.
- The keys of the parent node and its immediate right (or left) sibling are reappointed among them so that invariant 3 is satisfied.
- If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possible causing a ripple all the way to the root.
- If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.



B-Tree (Deletion)

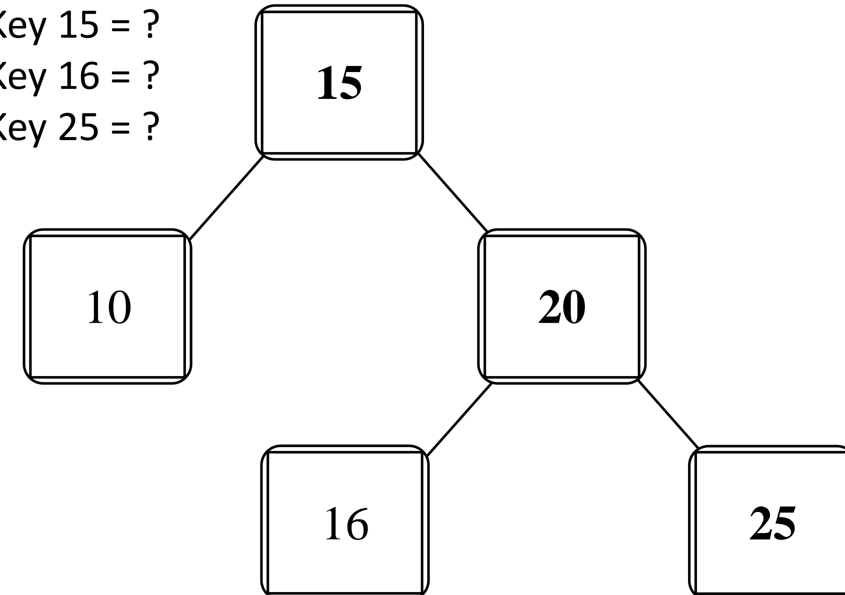
- While deleting an element from the tree, a condition known as Underflow may occur. Underflow occurs when a node consists of less than the minimum number of keys; it should hold.
- Some terms required to be understood before visualizing the deletion/removal operation:
- **In-order Predecessor:** The most significant key on the left child of a node is known as its in-order predecessor.
- **In-order Successor:** The minor key on the right child of a node is known as its in-order successor.

The following are **three prominent cases** of the deletion operation in a B Tree

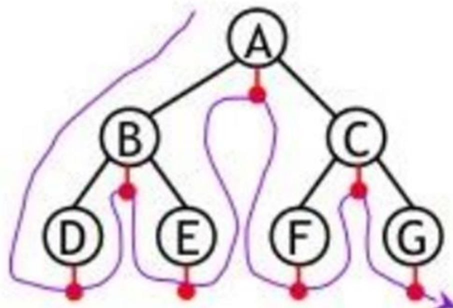
In-order Predecessor

- The inorder predecessor of a node p is the node q that comes just before p in the binary tree's inorder traversal.
- Given the root node of a binary search tree and the node p , find the inorder predecessor of node p .
- If it does not exist, return null .

Delete:
Key 15 = ?
Key 16 = ?
Key 25 = ?



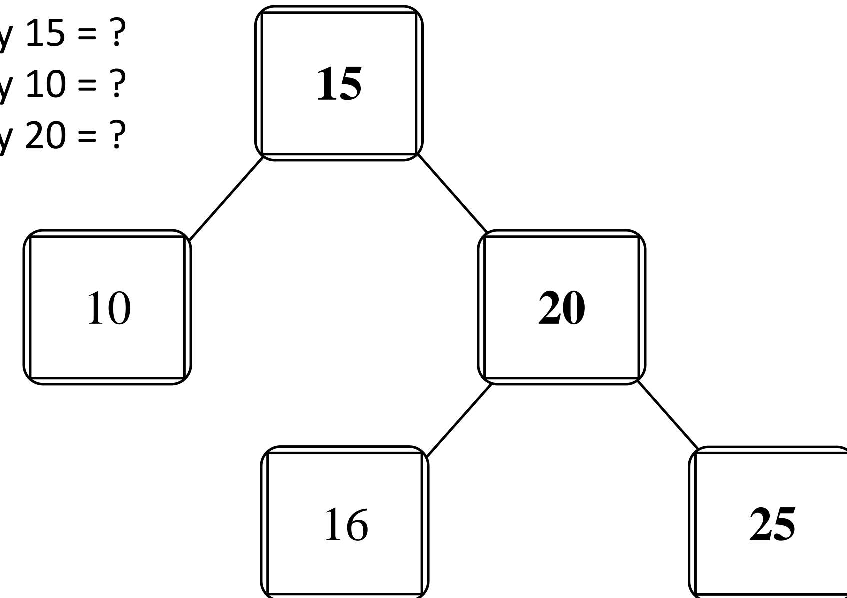
In-order
1. Left Subtree,
2. Root,
3. Right Subtree



In-order Successor

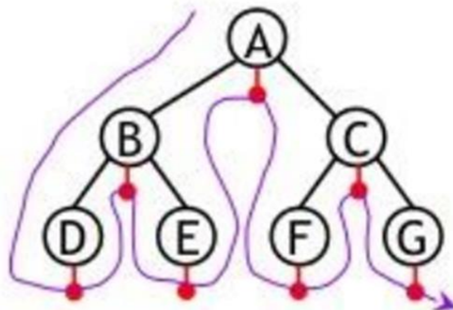
- Inorder successor of a node is the next node in Inorder traversal of the Binary Tree.
- Inorder Successor is NULL for the last node in Inorder traversal.
- In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of the input node.

Delete:
Key 15 = ?
Key 10 = ?
Key 20 = ?



In-order

1. Left Subtree,
2. Root,
3. Right Subtree



B-Tree
(Deletion – Case 1/3)

Case 1: The Deletion/Removal of the key lies in the Leaf node - This case is further divided into two different cases:

1. The deletion/removal of the key does not violate the property of the minimum number of keys a node should hold.

2. The deletion/removal of the key violates the property of the minimum number of keys a node should hold. In this case, we must borrow a key from its proximate sibling node in the order of Left to Right.
    - Firstly, we will visit the proximate Left sibling. If the Left sibling node has more than a minimum number of keys, it will borrow a key from this node.
    - Else, we will check to borrow from the proximate Right sibling node.

B-Tree
(Deletion – Case 2/3)

- **Case 2:** The Deleting/Removal of the key lies in the non-Leaf node - This case is further divided into following cases:
    1. The non-Leaf/Internal node, which is removed, is replaced by an in-order predecessor if the Left child node has more than the minimum number of keys.
    2. The non-Leaf/Internal node, which is removed, is replaced by an in-order successor if the Right child node has more than the minimum number of keys.
    3. If either child has a minimum number of keys, then we will merge the Left and the Right child nodes.

B-Tree
(Deletion – Case 3/3)

- **Case 3:** In the following case, the tree's height shrinks.
- If the target key lies in an Internal node, and the removal of the key leads to fewer keys in the node (which is less than the minimum necessitated), then look for the in-order predecessor and the in-order successor.
- If both the children have a minimum number of keys, then borrowing can't occur.
  - This leads to Case 2(3), i.e., merging the child nodes.
- We will again look for the sibling to borrow a key.
- However, if the sibling also consists of a minimum number of keys, then we will merge the node with the sibling along with the parent node and arrange the child nodes as per the requirements (ascending order).