B-Tree

Alok Kumar Jagadev

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency

Motivation for B-Trees

- Assume an AVL tree to store about 20 million records
  - The tree will end up with a very deep binary tree with lots of different disk accesses;
  - ≈ $\log_2 20{,}000{,}000$ is about 24
  - so this takes about 0.2 seconds

- Problem:
  - Not possible to improve on the log *n* lower bound to search in a binary tree
- Solution: To use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Binary vs. higher-order tree

- Binary trees:
  - Designed for in-memory searches
  - Try to minimize the number of memory accesses

- Higher-order trees:
  - Designed for searching data on block devices
  - Try to minimize the number of device accesses
    - Searching within a block is cheap!

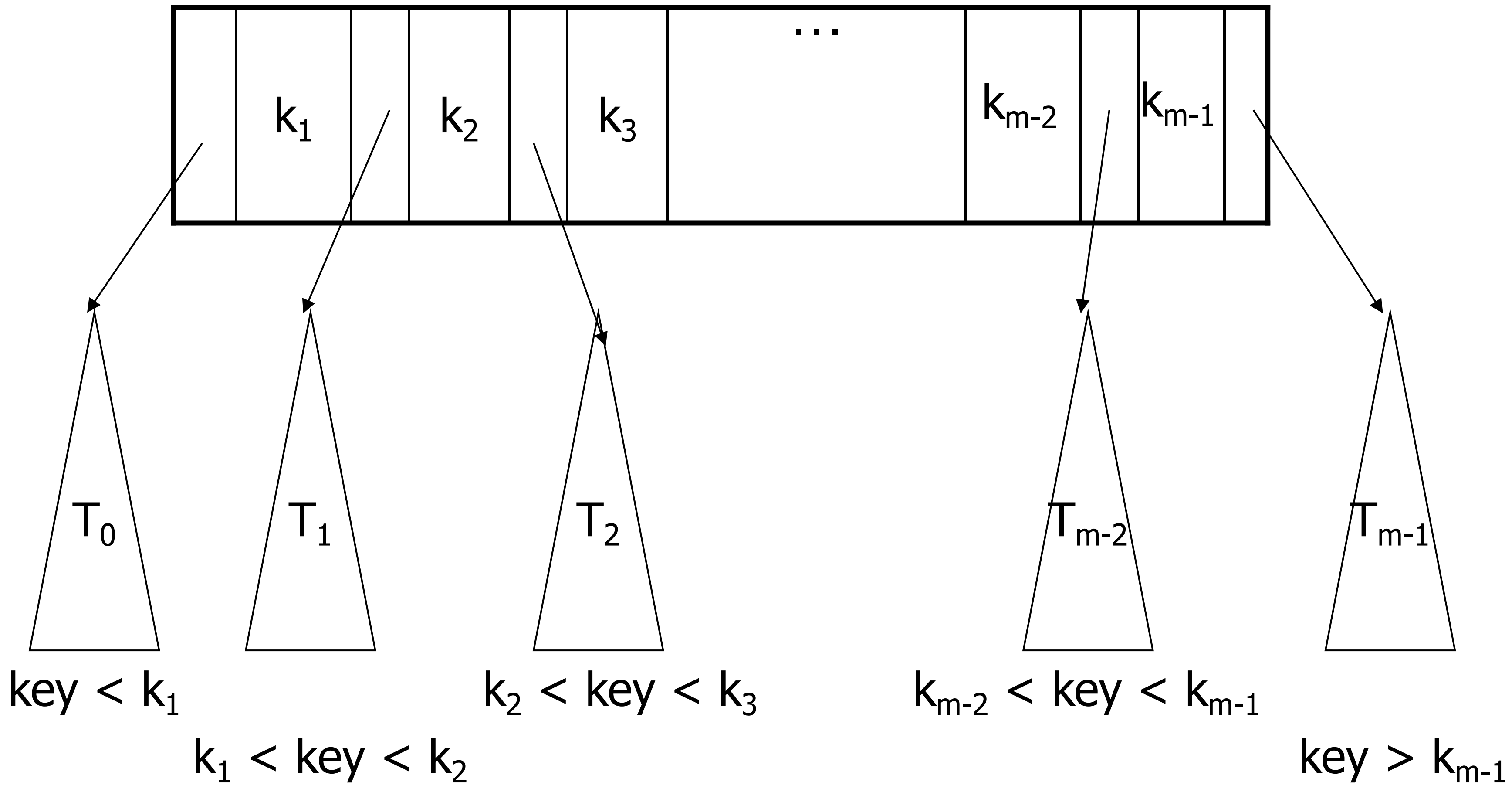
What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most **m** subtrees, where the subtrees may be empty.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.

- The keys and subtrees of a non-leaf node are ordered as:

  $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:
  - All keys in subtree T_0 are less than k_1.
  - All keys in subtree T_i , $1 \leq i \leq m - 2$, are greater than k_i but less than k_{i+1}.
  - All keys in subtree T_{m-1} are greater than k_{m-1}

Multi-way tree



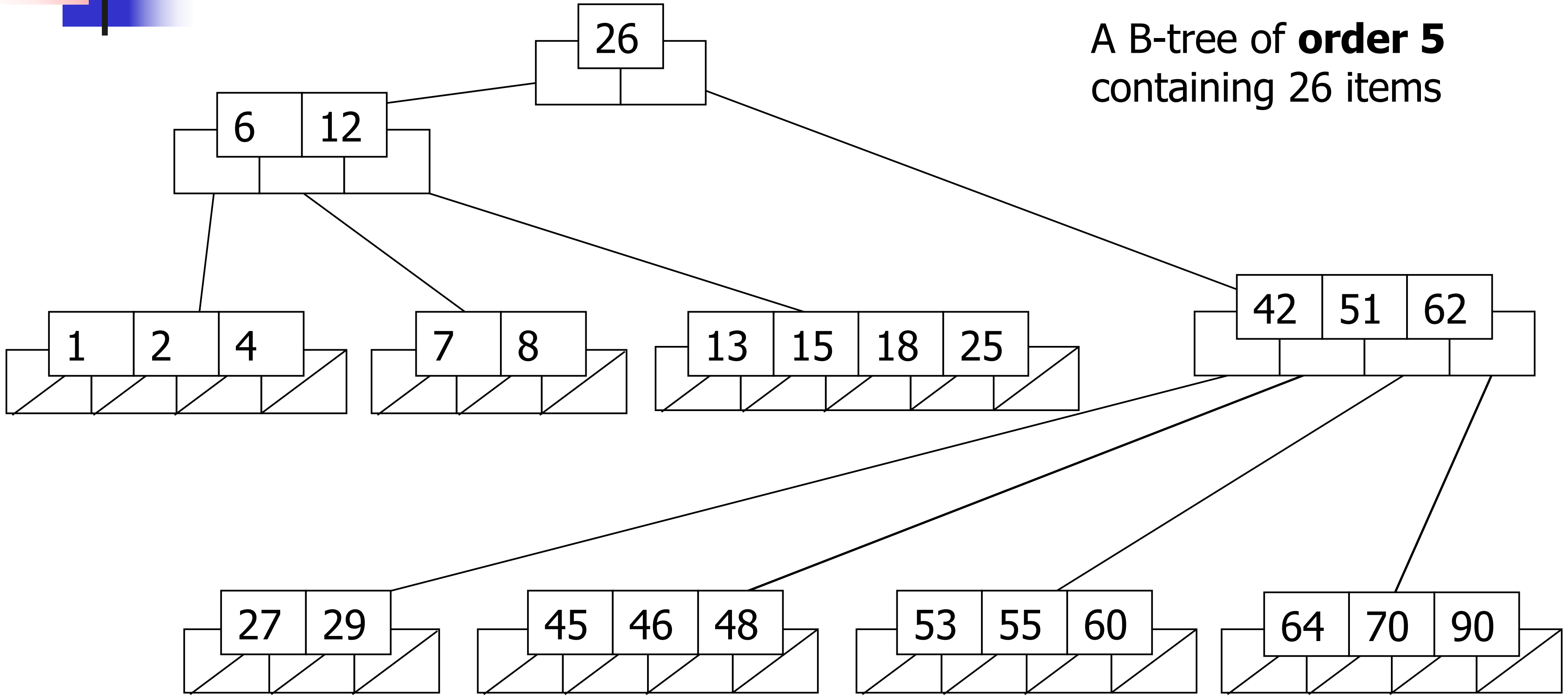
B-trees

- Generalization of binary search trees
  - Not binary trees
  - The B stands for Boeing, Balanced, and Bayer (suggested).
- Designed for searching data stored on block-oriented devices

B-tree

- A B-tree of order *m* is an *m*-way tree.
- Properties:
  - For each node x, the keys are stored in increasing order.
  - If m is the order of the tree, each internal node can contain at most m - 1 keys along with a pointer to each child.
  - Each node except root can have at most m children and at least m/2 children.
  - All leaves have the same level (balanced tree).
  - The root has at least 2 children and contains a minimum of 1 key.
- The number *m* should always be odd

An Example: B-Tree



A B-tree of **order 5** containing 26 items

Note: All the leaves are at the same level

Searching in B-Tree

- Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree.
- The steps are:

1. Starting from the root node, compare k with the first key of the node.
   i. If k = the first key of the node, return the node and the index.
   ii. If k.leaf = true and the key is not found, return NULL (i.e. not found).
   iii. If k < the first key of the root node, search the left child of this key recursively.
2. If there is more than one key in the current node and k > the first key, compare k with the next key in the node.

   If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).

   Else, search the right child of the key.

5. Repeat steps 1 to 2 until the leaf is reached.

Algorithm for Searching an Element

```
BtreeSearch(root, key)
    i = 1
    while i ≤ n[root] and key > $key_i$[root]   // n[root] means number of keys in root
        do i = i + 1
    if i ≤ n[root] and key == $key_i$[root]
        then return (root, i)
    if root is a leaf node and the key is not found
        then return NULL
    else
        return BtreeSearch($child_i$[root], key)
```

Inserting into a B-Tree

- Check whether tree is Empty.
- If tree is Empty, create a new node with new key value and insert it into the tree as a root node.
- Traverse the B-Tree in order to find the appropriate leaf node at which the node can be inserted.
  - If the leaf node contain less than m-1 keys then insert the element in the increasing order.
  - Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at middle.
    - Push the middle element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by these same steps.
- If the root will be split then the height of the tree is increased by one.

Constructing a B-tree

- Construct a B-tree of order 5, starting with an empty B-tree where keys are arriving in the order: 1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

Constructing a B-tree

- First four items go into the root: 1 12 8 2

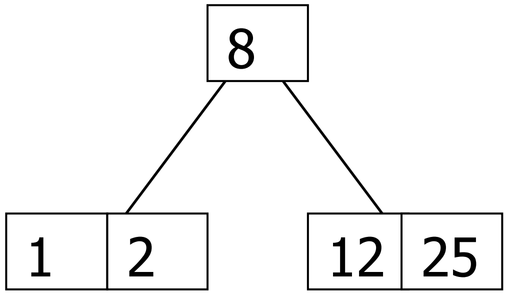
1	2	8	12

- Inserting the fifth item, 25, in the root would violate condition 5

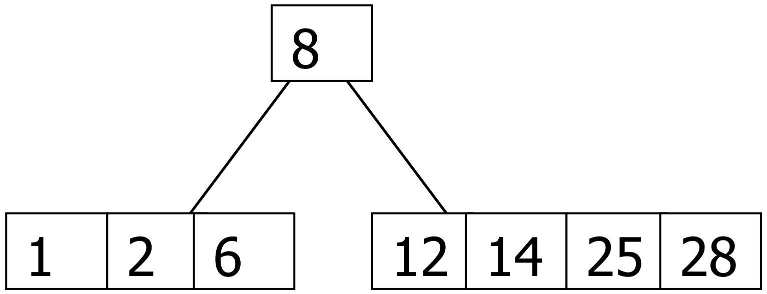
1	2	8	12	25

- Therefore, after arrival of 25, pick the middle key to make a new root

Constructing a B-tree (contd.)

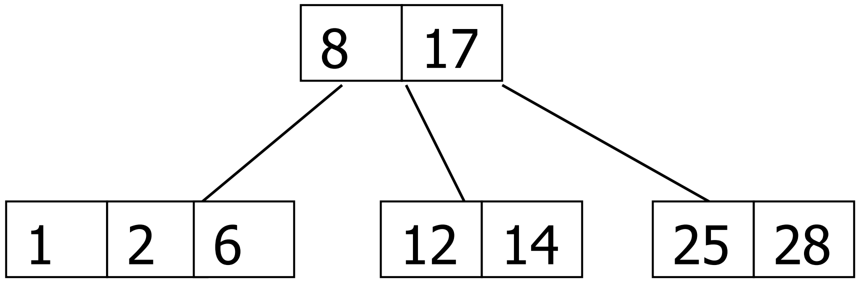


6, 14, 28 get added to the respective leaf nodes:

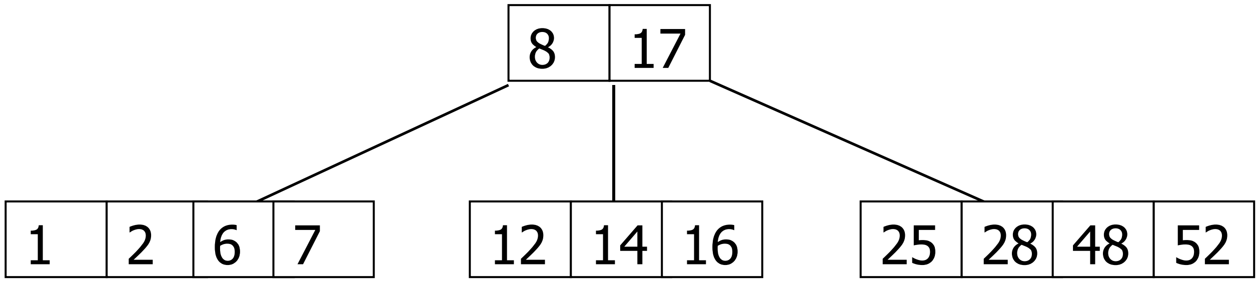


Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so take the middle key, promote it (to the root) and split the leaf

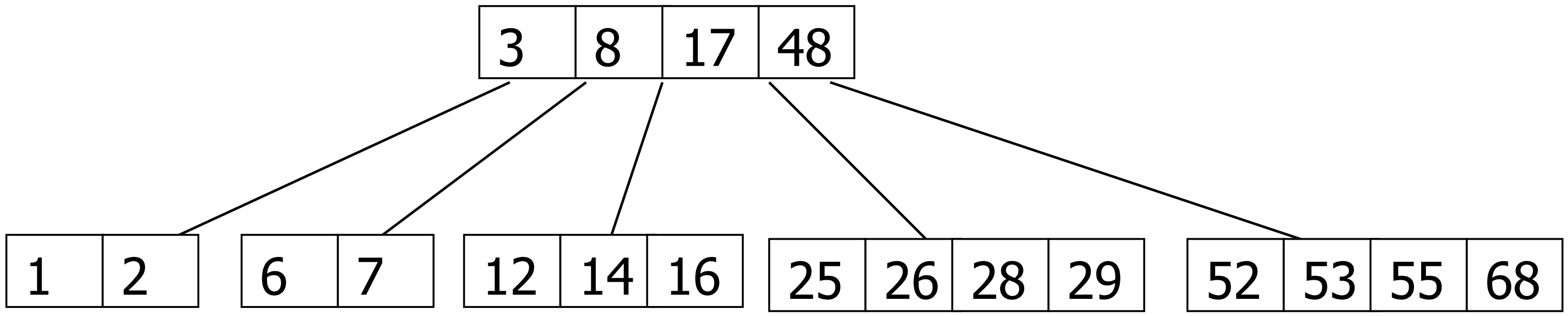


7, 52, 16, 48 get added to the leaf nodes

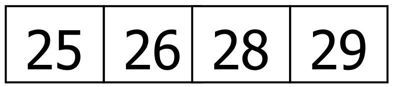


Constructing a B-tree (contd.)

- Adding 68 causes to split the right most leaf, promoting 48 to the root, and
- Adding 3 causes to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

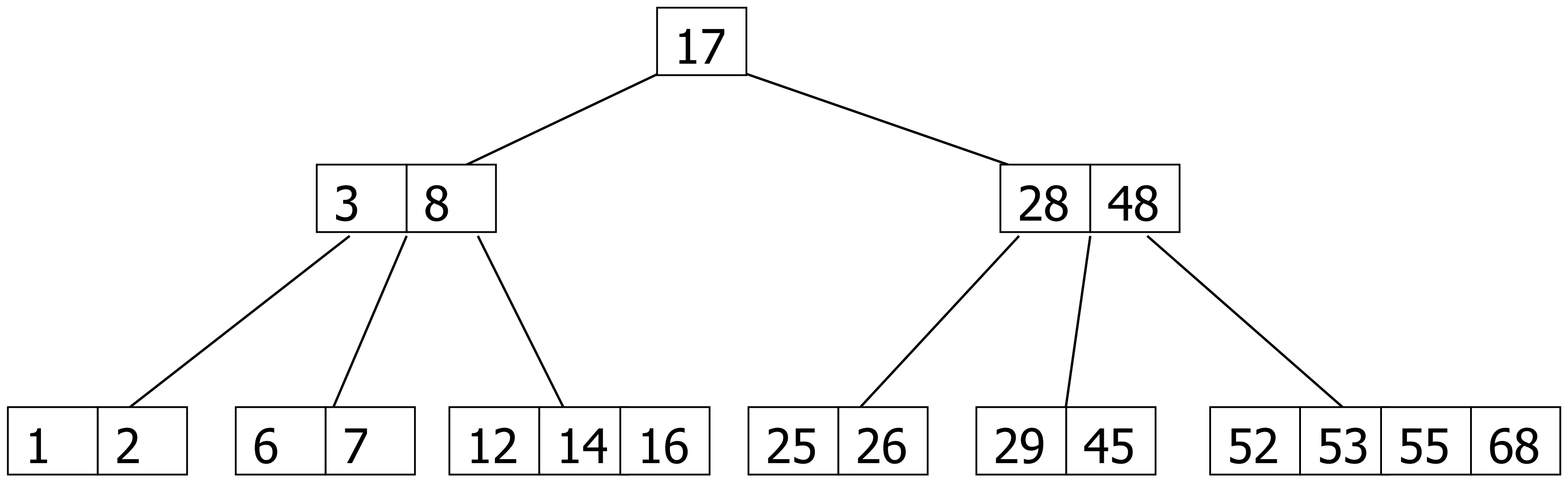


- Adding 45 causes a split of



- and promoting 28 to the root then causes the root to split

Constructing a B-tree (contd.)



Exercise: Construct a B-Tree

- Insert the following keys to a 5-way B-tree with arrival of keys: 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Delete key from a B-tree

- Deleting an element on a B-tree consists of three main events:
  i. searching the node where the key to be deleted exists
  ii. deleting the key and
  iii. balancing the tree if required.

- While deleting a node from a tree, an underflow condition may occur.
  - Underflow occurs when a node contains less than the minimum number of keys it should hold.

- The terms to be understood before studying deletion operation are:
  - Inorder Predecessor: largest key on the left child of a node
  - Inorder Successor: smallest key on the right child of a node

Delete key from a B-tree

Before going through the deletion steps, refresh the facts about a B-tree of order m.

- A node can have at most m children.
- A node can contain a maximum of m-1 keys.
- A node should have at least ⌈ m/2⌉ children.
- A node (except root node) should contain at least ⌈ m/2⌉ -1 keys.

Delete key from a B-tree

There are three main cases for deletion operation in a B-tree.

- Case I: The key to be deleted lies in the leaf.
- Case II: If the key to be deleted lies in the internal node.
- Case III: If the key to be deleted lies in the root node.

Delete key from a B-tree

Case I: The key to be deleted lies in the leaf.

There are two cases for it.

1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.
   - Simply delete the key updating the desired links

Delete key from a B-tree

2. Deletion of the key: violates the minimum number of keys a node should hold.
   - Borrow a key from its immediate neighboring sibling node in the order of left to right.
     - left sibling: contains more than minimum (m/2) number of keys,
       - push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
     - right sibling: contains more than minimum (m/2) number of keys,
       - push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
   - If both the immediate sibling nodes already have a minimum number of keys:
     - merge the node with either the left sibling node or the right sibling node.
     - This merging is done through the parent node.

Delete key from a B-tree

Case II: If the key to be deleted lies in the internal node, the following cases occur.

a) The key is replaced by an inorder predecessor if the left child has more than the minimum number of keys.

b) The key is replaced by an inorder successor if the right child has more than the minimum number of keys.

c) If either child has exactly a minimum number of keys then, merge the left and the right children.

- After merging: if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

Delete key from a B-tree

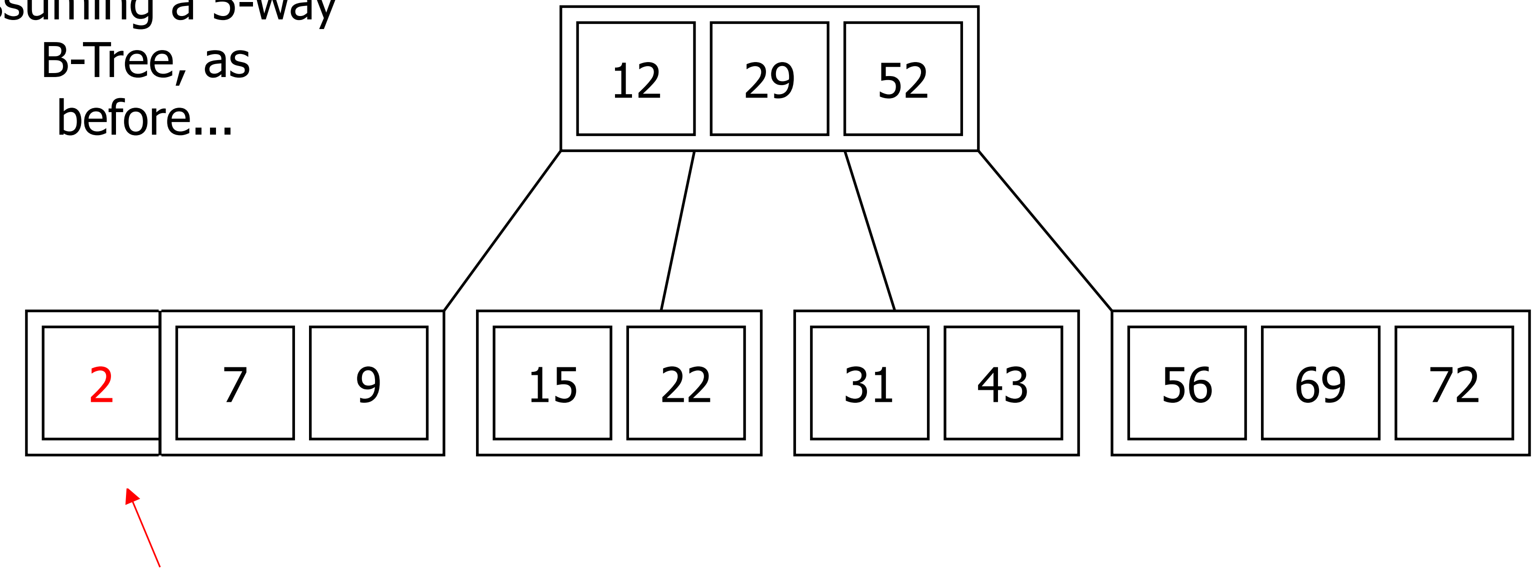
If the target key lies in an internal node, and the deletion of the key leads to a fewer the minimum keys required,

- look for the inorder predecessor and the inorder successor.
- If both the children contain a minimum number of keys then,
  - borrowing cannot take place.
  - This leads to Case II(c) i.e. merging the children.
- In this case, the height of the tree shrinks.

- Again, look for the sibling to borrow a key.
- If the sibling also has only a minimum number of keys
  - merge the node with the sibling along with the parent.
- Arrange the children accordingly (increasing order).

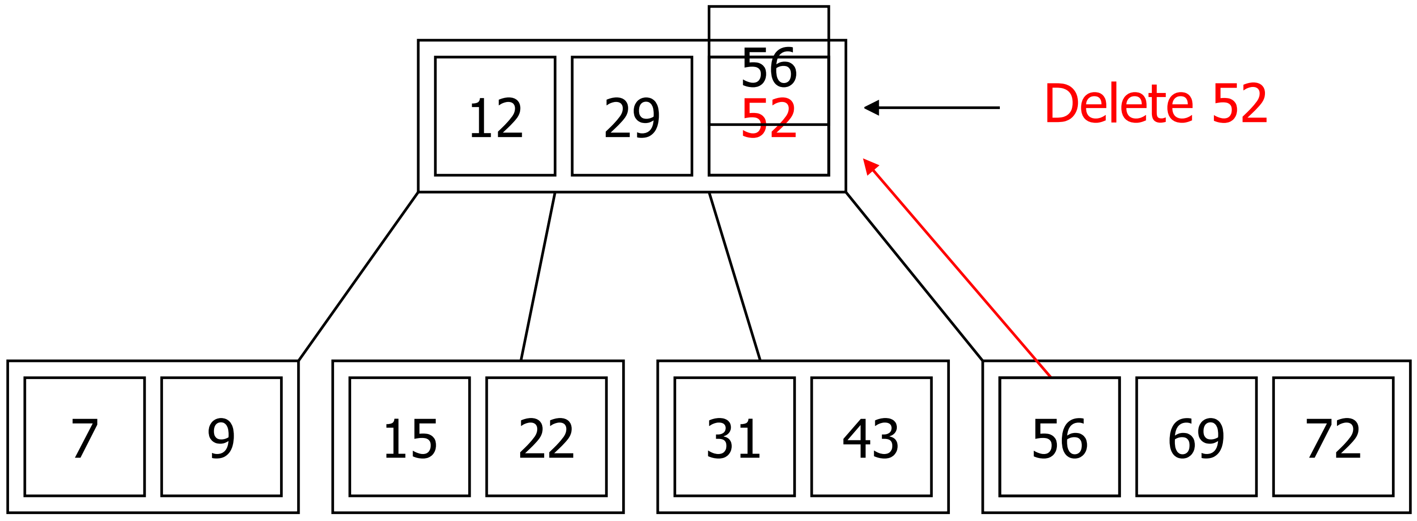
Simple leaf deletion

Assuming a 5-way B-Tree, as before...



Delete 2: Since there are enough keys in the node, just delete it

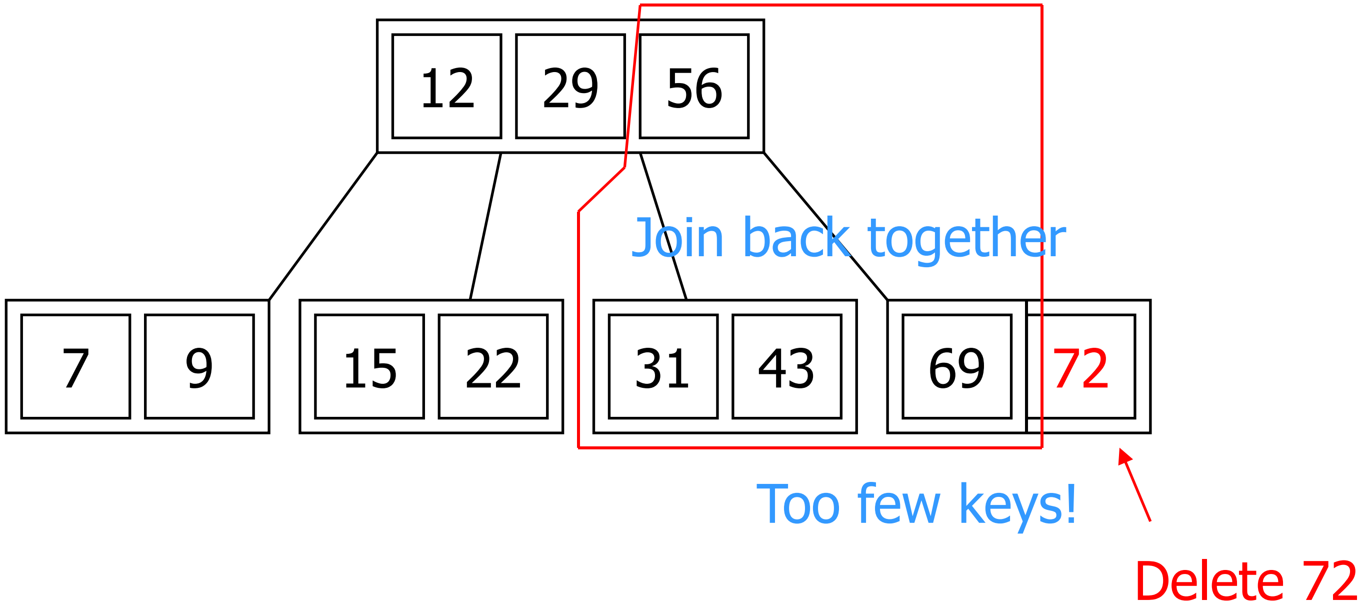
Simple non-leaf deletion



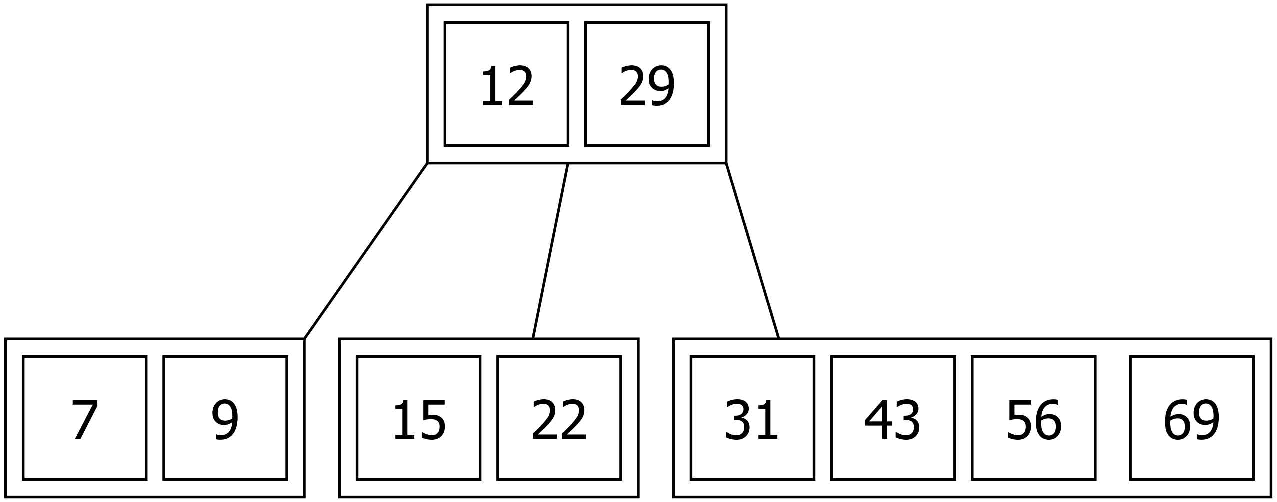
Delete 52

Borrow the predecessor or (in this case) successor

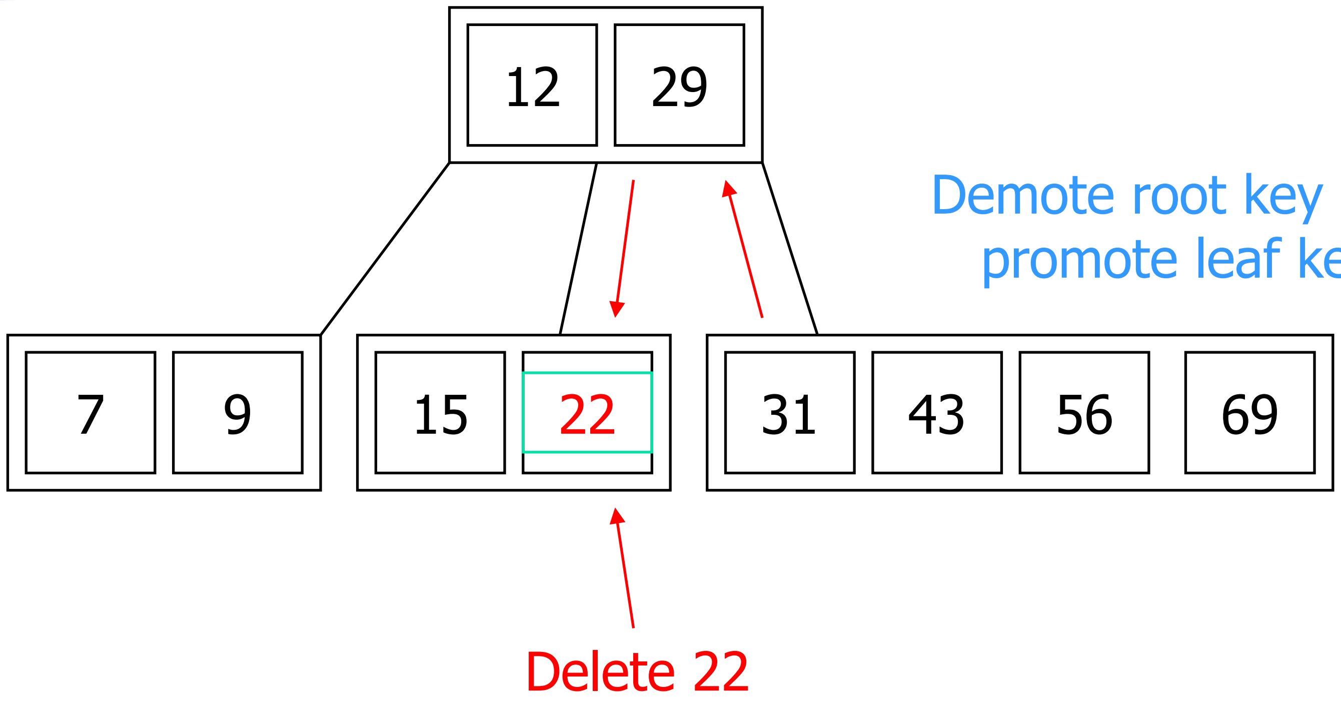
Too few keys in node and its siblings



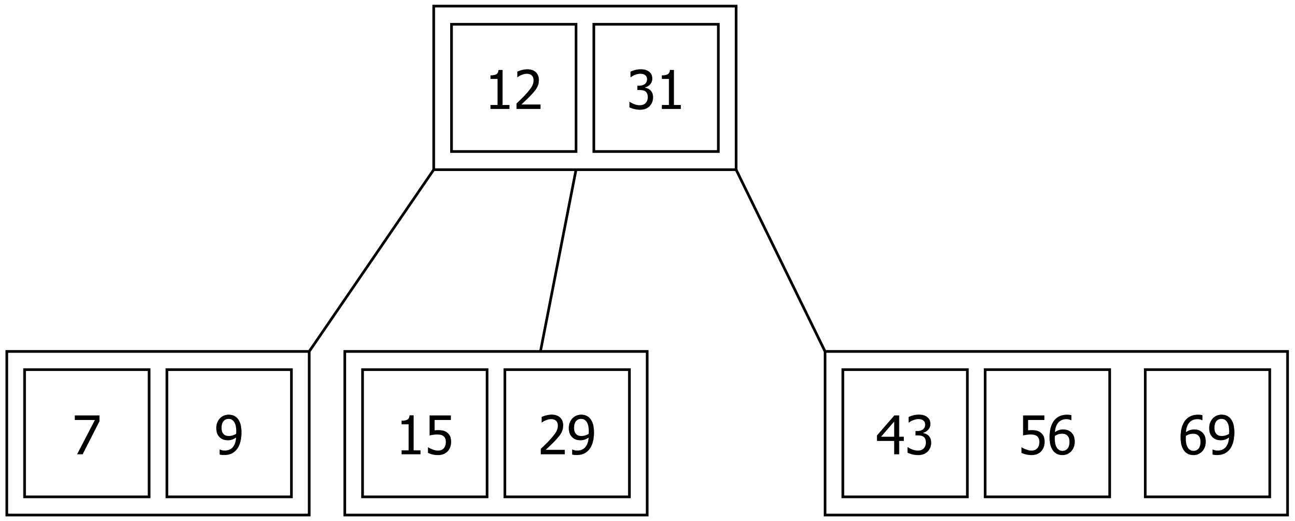
Too few keys in node and its siblings (Contd...)



Enough siblings



Enough siblings



Exercise in Removal from a B-Tree

- Given 5-way B-tree created by these data:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

- Add these further keys: 2, 6,12

- Delete these keys: 4, 5, 7, 3, 14

Exercise

- B-Tree of order 5
  - Each node has at most 4 pointers and 3 keys, and at least 2 pointers and 1 key.
- Insert: 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8
- Delete: 2, 21, 10, 3, 4

Analysis of B-Trees

- The maximum number of keys in a B-tree of order m and height h:

root	$m - 1$
level 1	$m\ (m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$
$$[(m^{h+1} - 1)/\ (m - 1)]\ (m - 1) = \underline{m^{h+1} - 1}$$

- When m = 5 and h = 2 this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- Data is stored on the disk blocks,
  - this data is brought into the main memory when required.
- In-case of huge data, searching one record in the disk requires reading the entire disk;
  - this increases time and main memory consumption due to high disk access frequency and data size.
- To overcome this, index tables are created that saves the record reference of the records based on the blocks they reside in.
  - This drastically reduces the time and memory consumption.
- Since there will be huge data, it can create multi-level index tables.
- Multi-level index can be designed by using B-Tree for keeping the data sorted in a self-balancing fashion.

Reasons for using B-Trees

- If B-tree of order 101 is used, then each node can be transferred in one disk read operation
- A B-tree of order 101 and height 3 can hold $101^4 - 1$ keys (approximately 100 million) and any key can be accessed with 3 disk reads (assuming the root is hold in memory)

Comparing Trees

- Binary search trees
    - Can become unbalanced and lose their good time complexity (big O)
    - AVL trees are strict binary trees that *overcome the balance problem*
    - Heaps remain balanced but only prioritize (not order) the keys

- Multi-way trees
    - B-Trees can be *m*-way, they can have any (odd) number of children
    - One B-Tree, the 2-3 (or 3-way) B-Tree

B+ Tree

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B-Tree, keys and records both can be stored in the internal as well as leaf nodes.
  - whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree

- B+ Tree are used to store the large amount of data which cannot be stored in the main memory.
- Due to the fact that, size of main memory is always limited,
  - the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas,
  - leaf nodes are stored in the secondary memory.

Advantages of B+ Tree

- Records can be fetched in equal number of disk accesses.
- Height of the tree remains balanced and less as compare to B-tree.
- Can access the data stored in a B+ tree sequentially as well as directly.
- Keys are used for indexing.
- Faster search queries as the data is stored only on the leaf nodes.

B-Tree vs B+ Tree

- Search keys cannot be repeatedly stored.
- Data can be stored in leaf nodes as well as internal nodes
- Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.
- Deletion of internal nodes are so complicated and time consuming.
- Leaf nodes cannot be linked together.

- Redundant search keys can be present.
- Data can only be stored on the leaf nodes.
- Searching is comparatively faster as data can only be found on the leaf nodes.
- Deletion will never be a complex process since element will always be deleted from the leaf nodes.
- Leaf nodes are linked together to make the search operations more efficient.

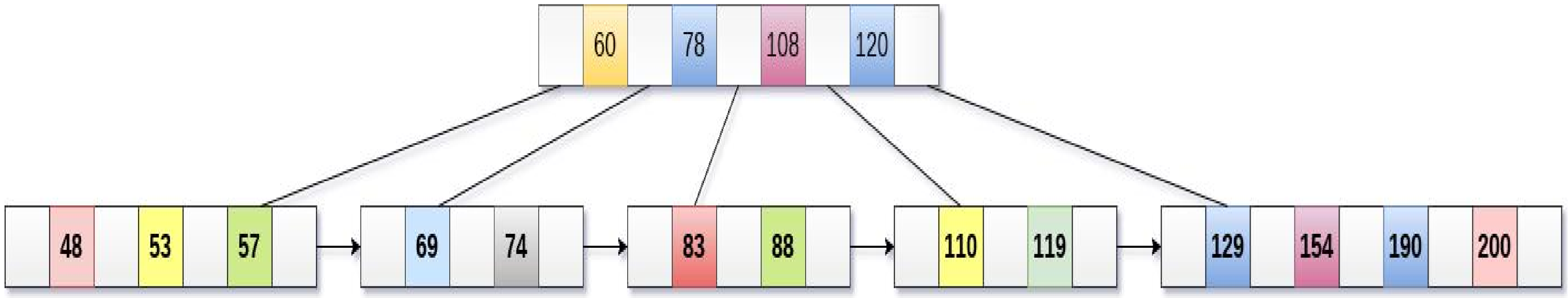
Insertion in B+ Tree

Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

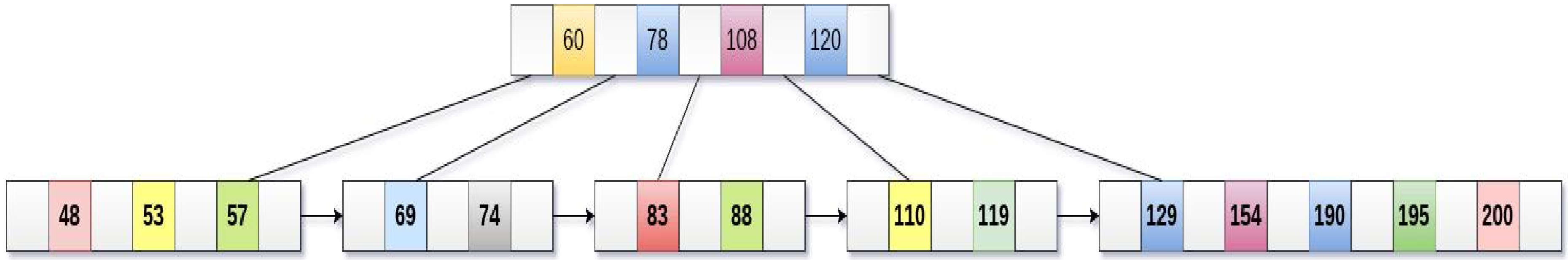
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example



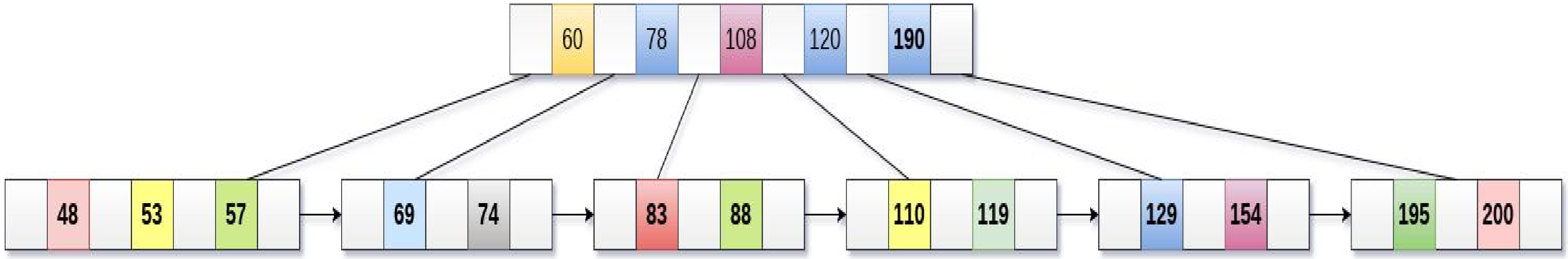
Example (Contd…)

- Insert the value 195 into the B+ tree of order 5.
- 195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position



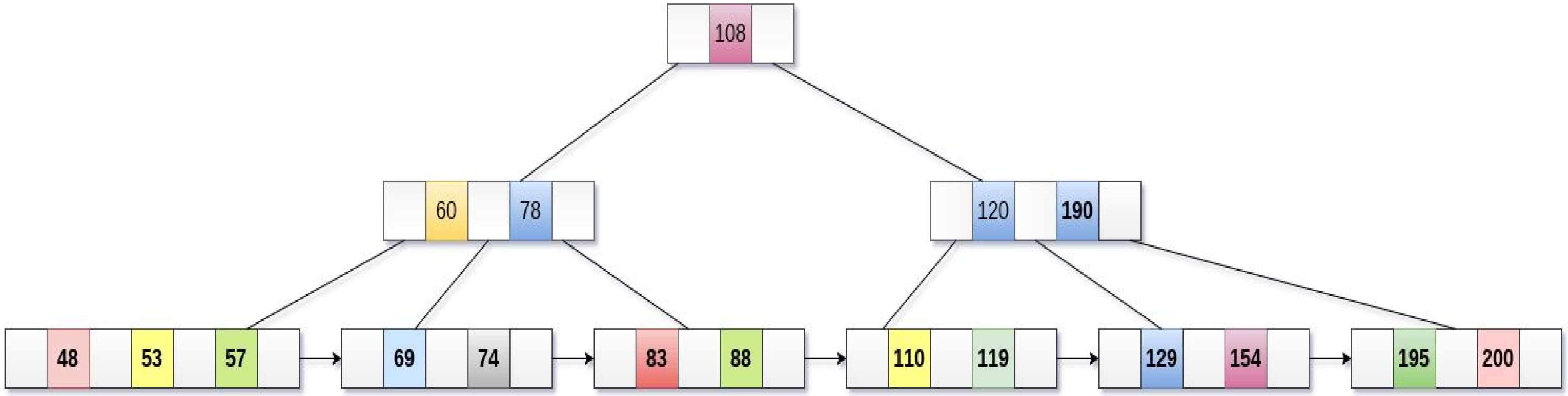
Example (Contd…)

- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the middle node up to the parent.



Example (Contd…)

- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



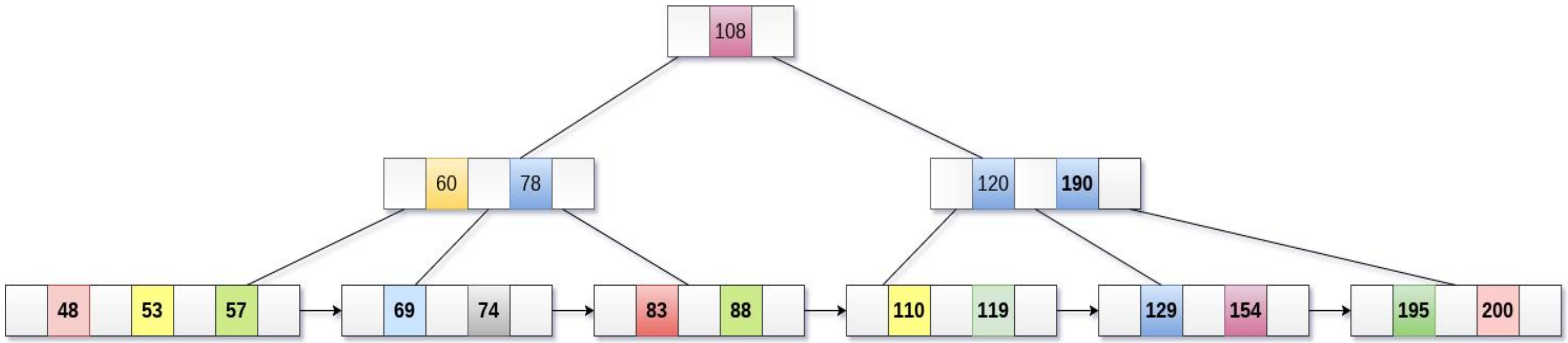
Deletion in B+ Tree

Step 1: Delete the key and data from the leaves.

Step 2: If the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

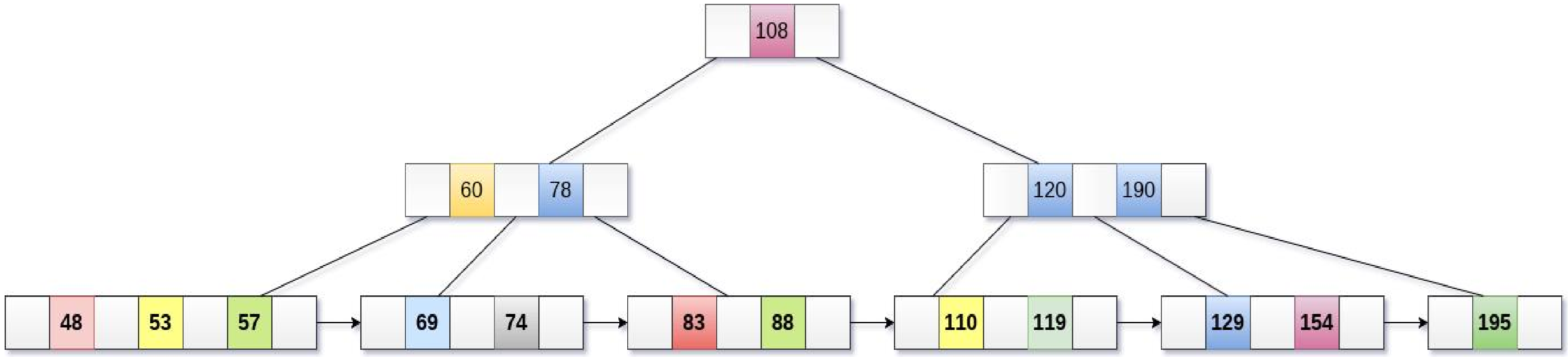
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example



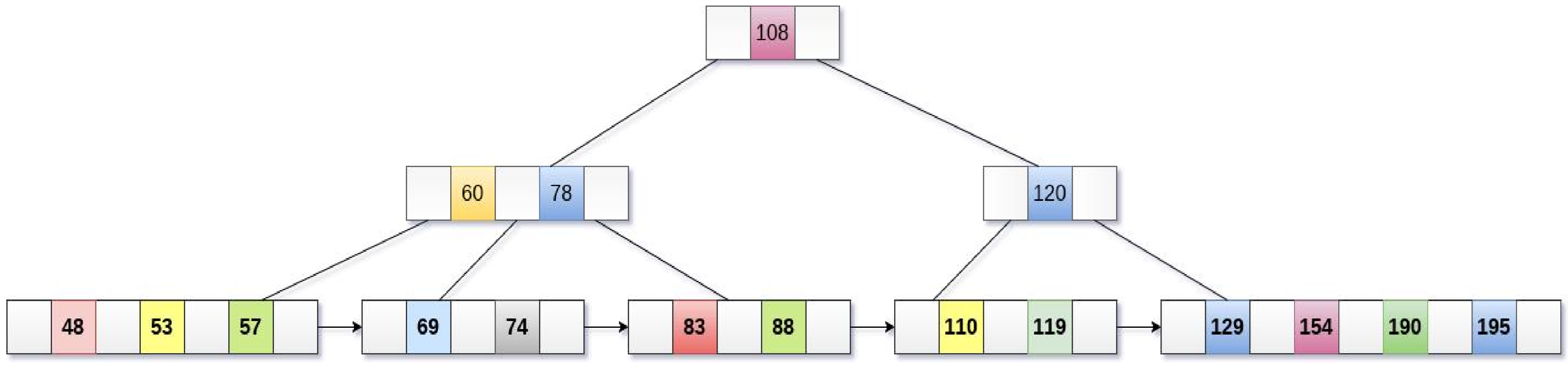
Example (Contd...)

- Delete the key 200 from the B+ Tree.
- 200 is present in the right sub-tree of 190, after 195. delete it.



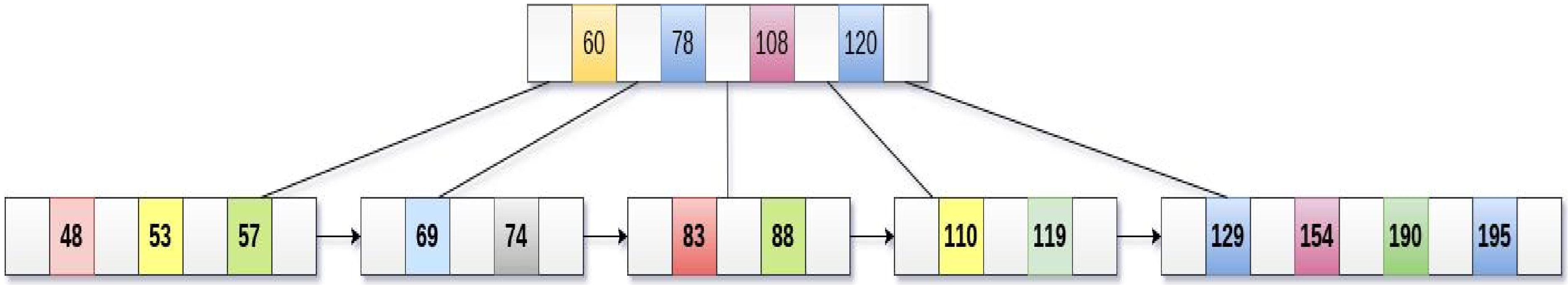
Example (Contd...)

- Merge the two nodes by using 195, 190, 154 and 129.



Example (Contd…)

- Element 120 is the single element present in the node which is violating the B+ Tree properties.
- Merge it by using 60, 78, 108 and 120.
- So, the height of B+ tree will be decreased by 1.



Priority Queue ADT

1. PQueue data: collection of data with priority
2. PQueue operations:
   - insert
   - deleteMin
3. PQueue property: for two elements in the queue, *x* and *y*, if *x* has a lower priority value than *y*, *x* will be deleted before *y*

Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

Potential Implementations

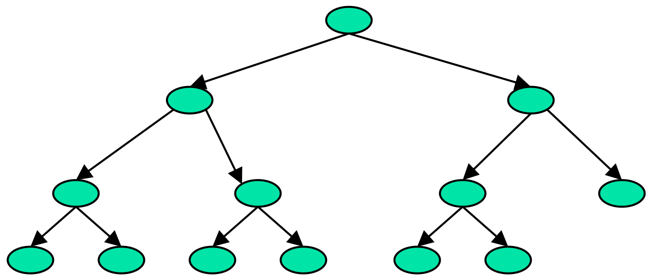
	insert	deleteMin
Unsorted list (Array)	O(1)	O(n)
Unsorted list (Linked-List)	O(1)	O(n)
Sorted list (Array)	O(n)	O(1)
Sorted list (Linked-List)	O(n)	O(1)

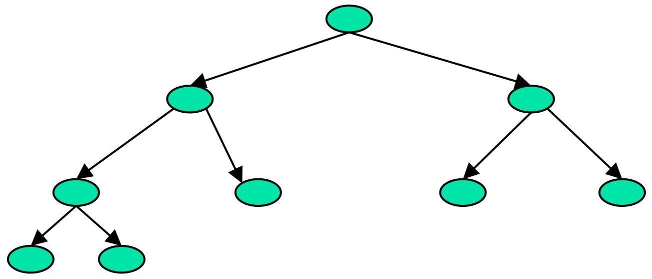
Binary Heap Properties

1. Structure Property
2. Ordering Property

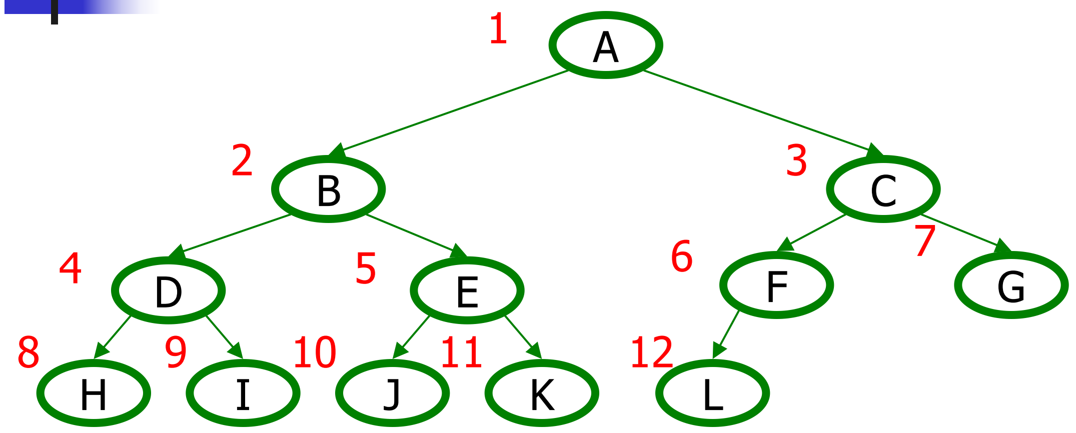
Heap Structure Property

- A binary heap is a *complete* binary tree.
- Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.
- Examples:





Representing Complete Binary Trees in an Array



From node **i**:

left child at: $2 \times i$
right child at: $2 \times i + 1$
parent at: $\lfloor i/2 \rfloor$
Heapsize(A) ≤ length[A]

implicit (array) implementation:

	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

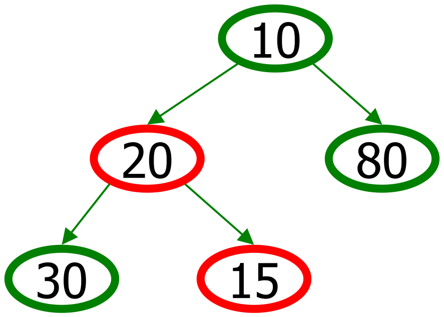
Heap Types

- Max-heaps (largest element at root), have the *max-heap property*:
  - for all nodes i, excluding the root: A[parent(i)] ≥ A[i]

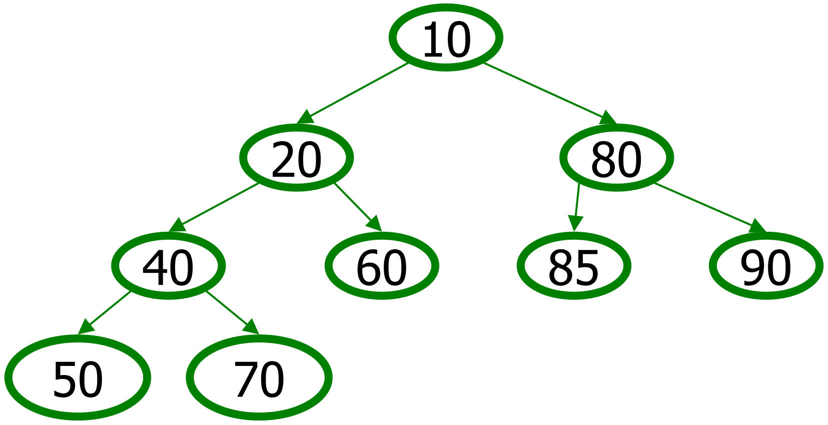
- Min-heaps (smallest element at root), have the *min-heap property*:
  - for all nodes i, excluding the root: A[parent(i)] ≤ A[i]

Heap Order Property

- Heap order property: the value stored at each node is smaller or equal to the values stored at the children.

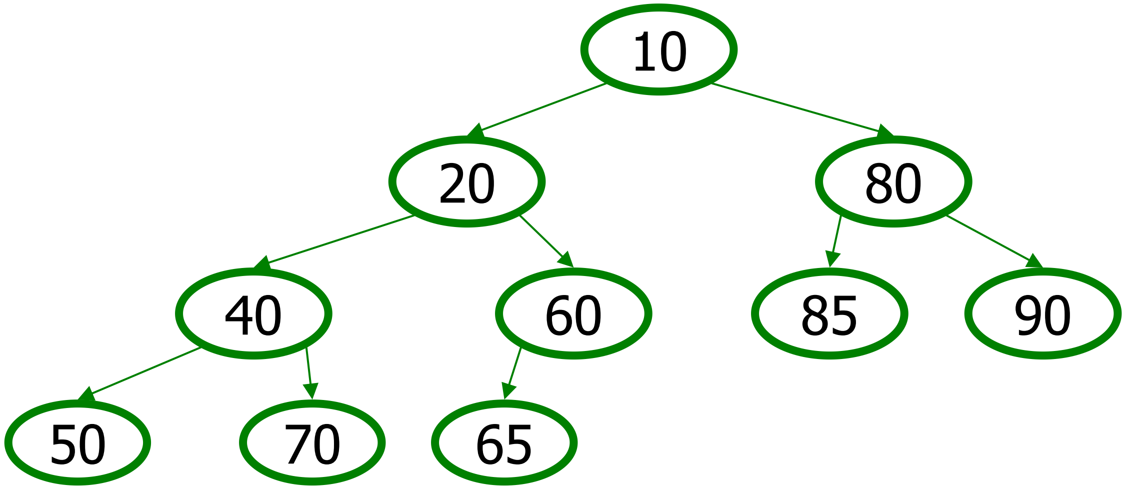


not a heap



Heap Operations

- findMin:
- insert(val)
- deleteMin

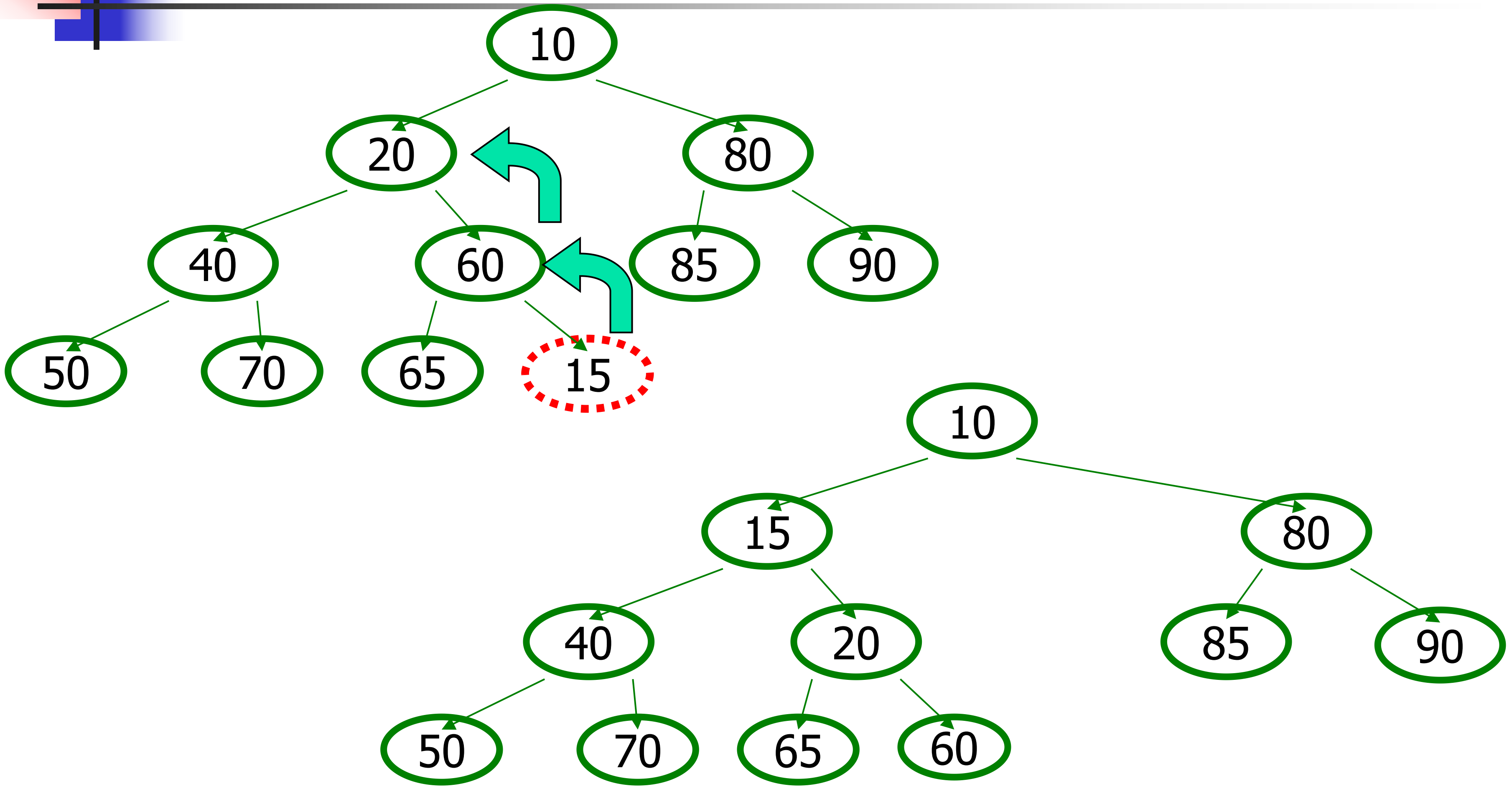

$$\begin{tikzcd}
 & & & 10 \arrow[dl] \arrow[dr] & & \\
 & & 20 \arrow[dl] \arrow[d] & & 80 \arrow[d] \arrow[dr] & \\
 & 40 \arrow[dl] \arrow[d] & 60 \arrow[d] & & 85 & 90 \\
50 & 70 & 65 & & &
\end{tikzcd}$$

Heap – Insert(val)

Basic Idea:

1. Put val at "*next*" leaf position
2. Heapify() repeatedly exchanging node until no longer needed

Insert: Heapify

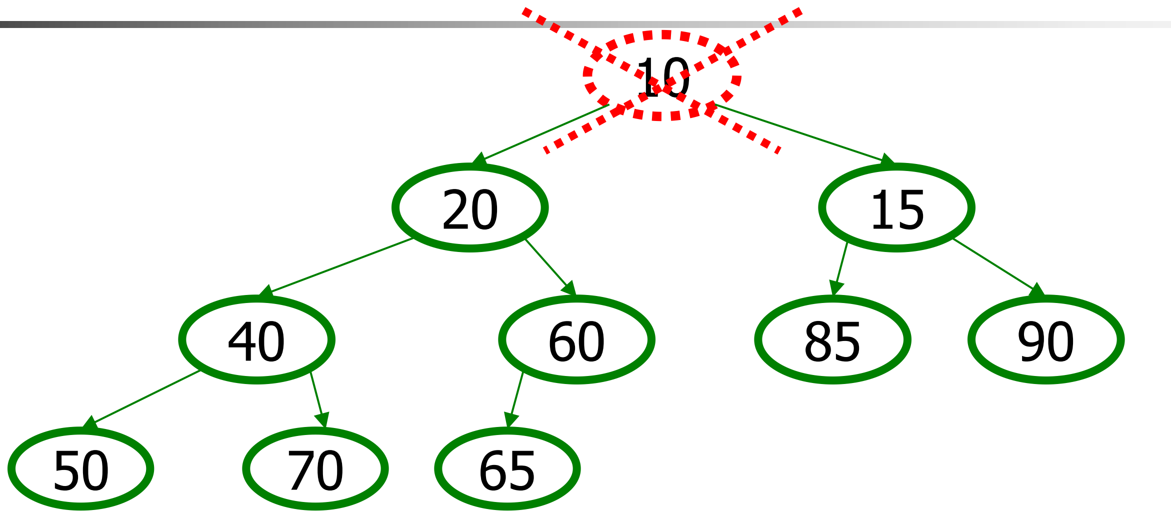


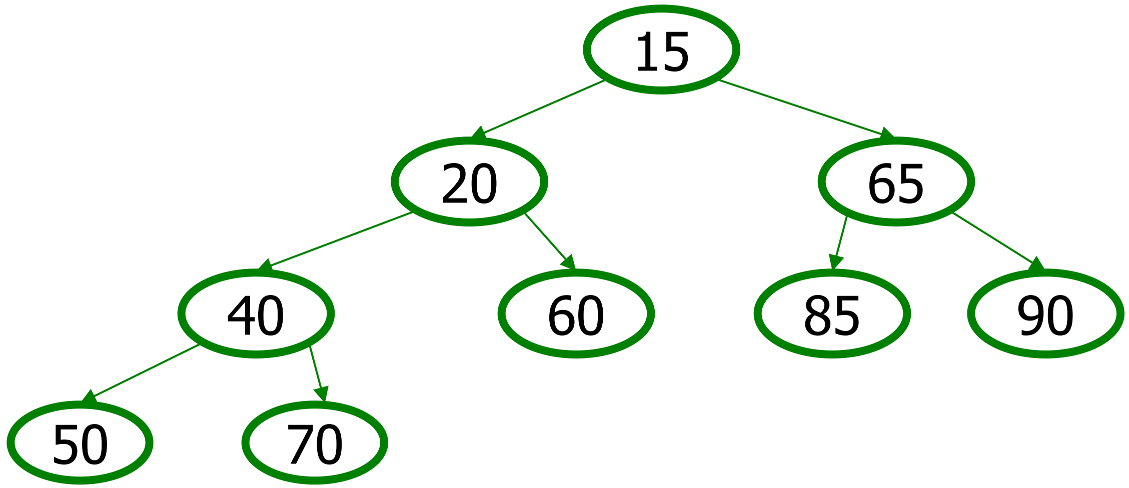
Heap – Deletemin

Basic Idea:

1. Remove root (that is always the min!)
2. Put "last" leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

DeleteMin





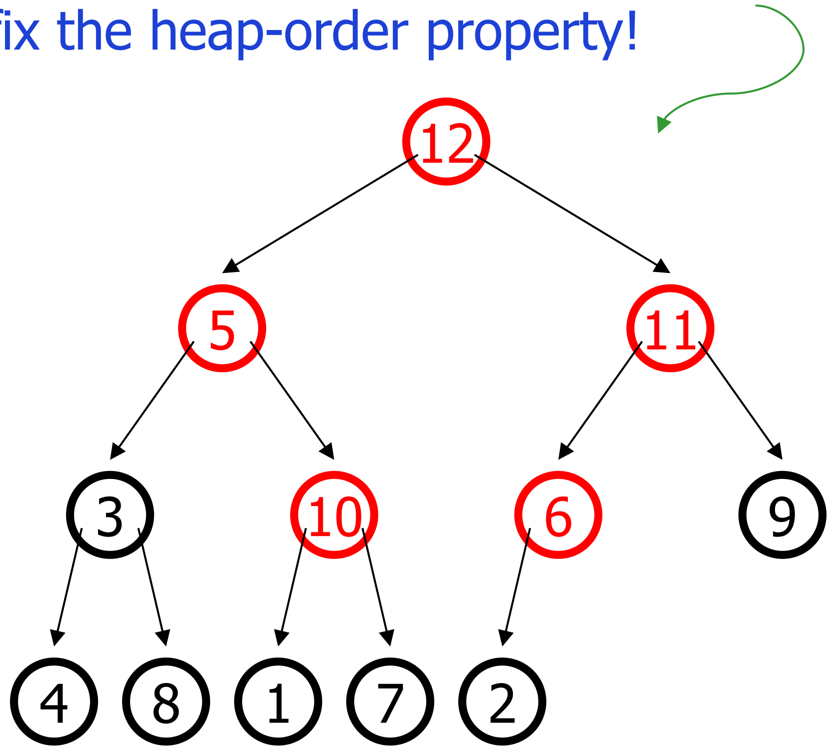
Working on Heaps

- What are the two properties of a heap?
  - Structure Property
  - Order Property
- How do we work on heaps?
  - Fix the structure
  - Fix the order

BuildHeap: Floyd’s Method

12	5	11	3	10	6	9	4	8	1	7	2

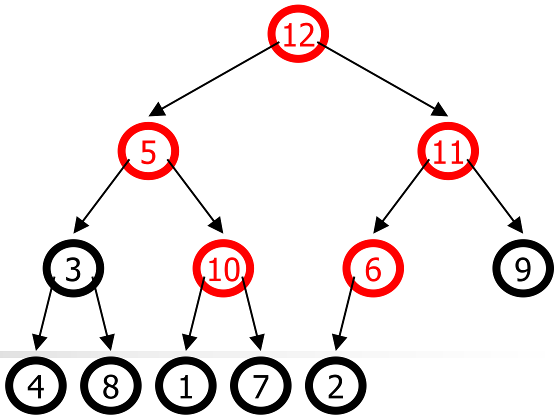
Add elements arbitrarily to form a complete tree.
Pretend it’s a heap and fix the heap-order property!



Buildheap pseudocode

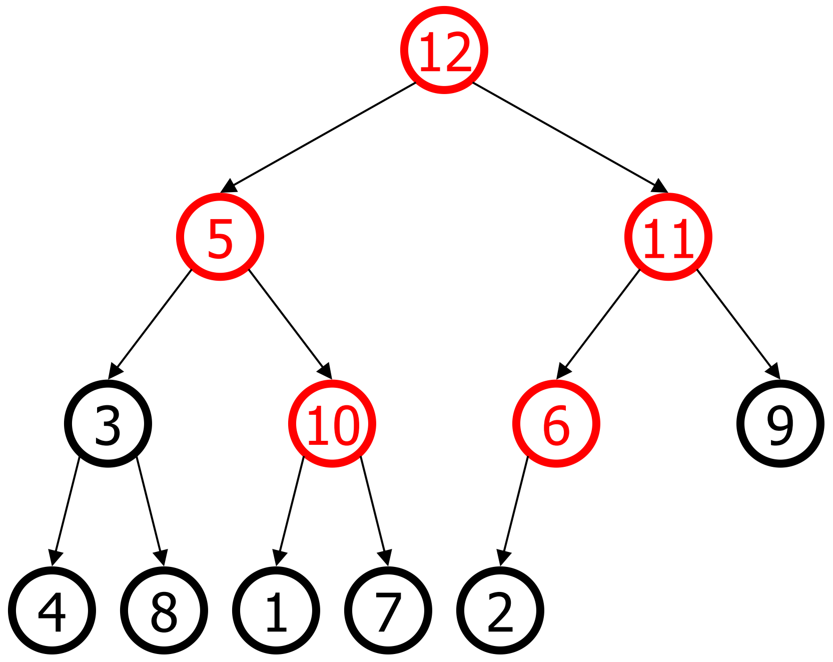
```
void buildHeap(A[], size) {
    for ( int i = (size-1)/2; i >= 0; i-- )
        heapify(A, size, i);
}
```

Heapify pseudocode

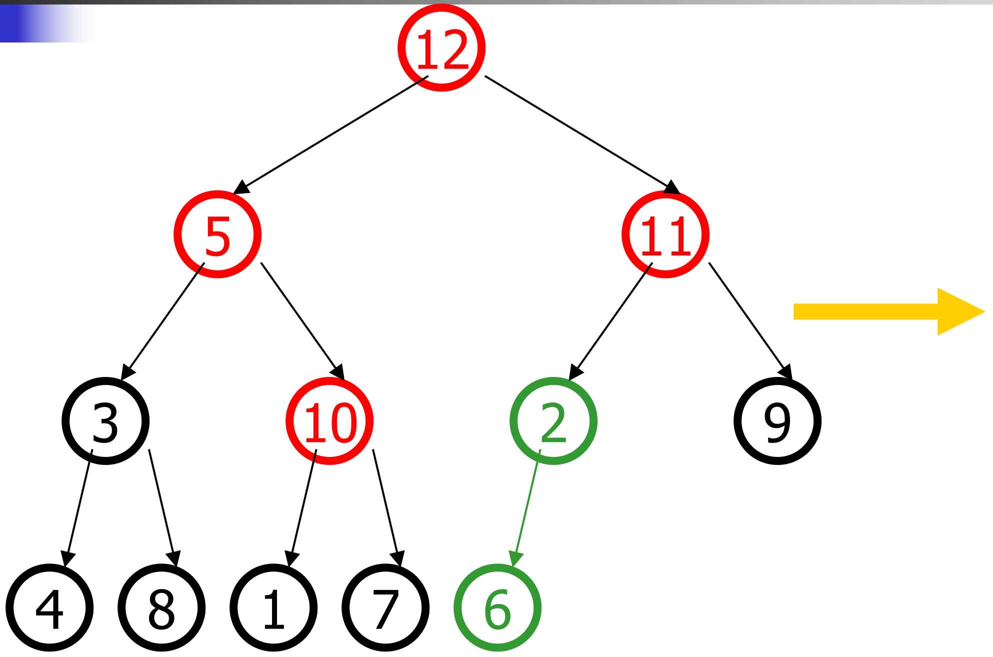


```
void heapify(int A[], int n, int i) {
    int min = i;                          // Initialize min as root
    int l = 2*i + 1;                      // left = 2*i + 1
    int r = 2*i + 2;                      // right = 2*i + 2
                                   // If left child is larger than root
    if (l < n && A[l] < A[min])
        min = l;
                                   // If right child is larger than largest so far
    if (r < n && A[r] < A[min])
        min = r;
                                          // If largest is not root
    if (min != i) {
        swap(A[i], A[min]);
                                   // Recursively heapify the affected sub-tree
        heapify(A, n, min);
    }
}
```

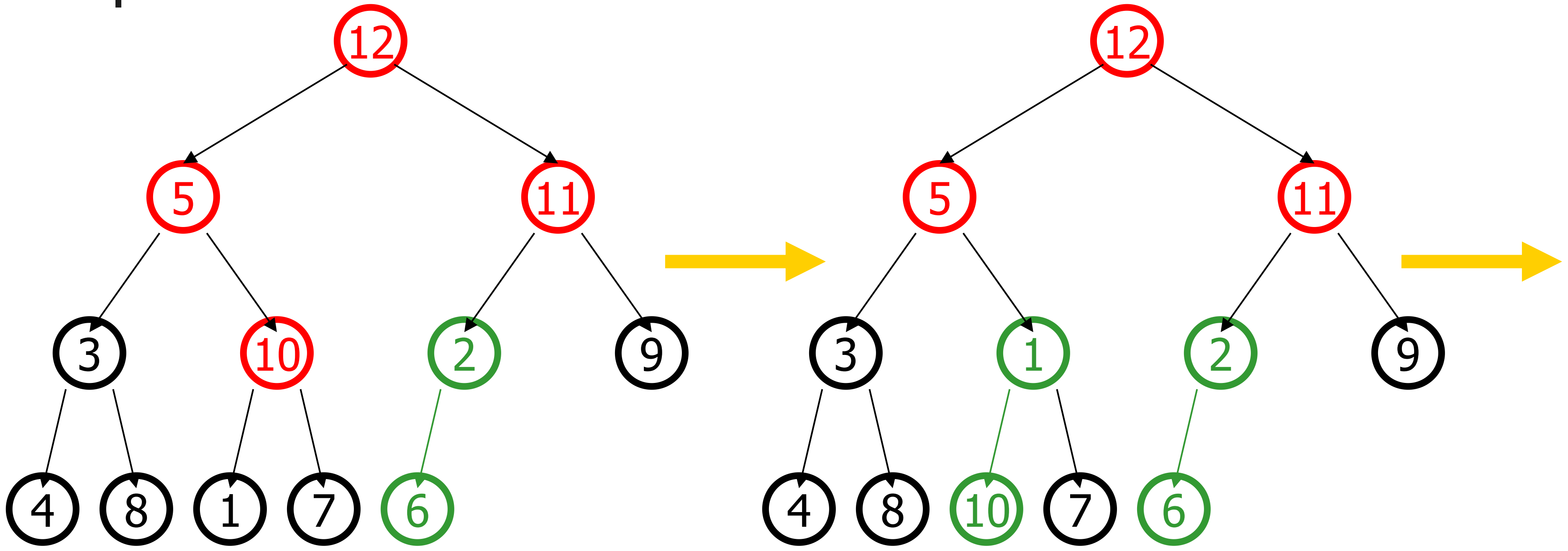
BuildHeap: Floyd's Method



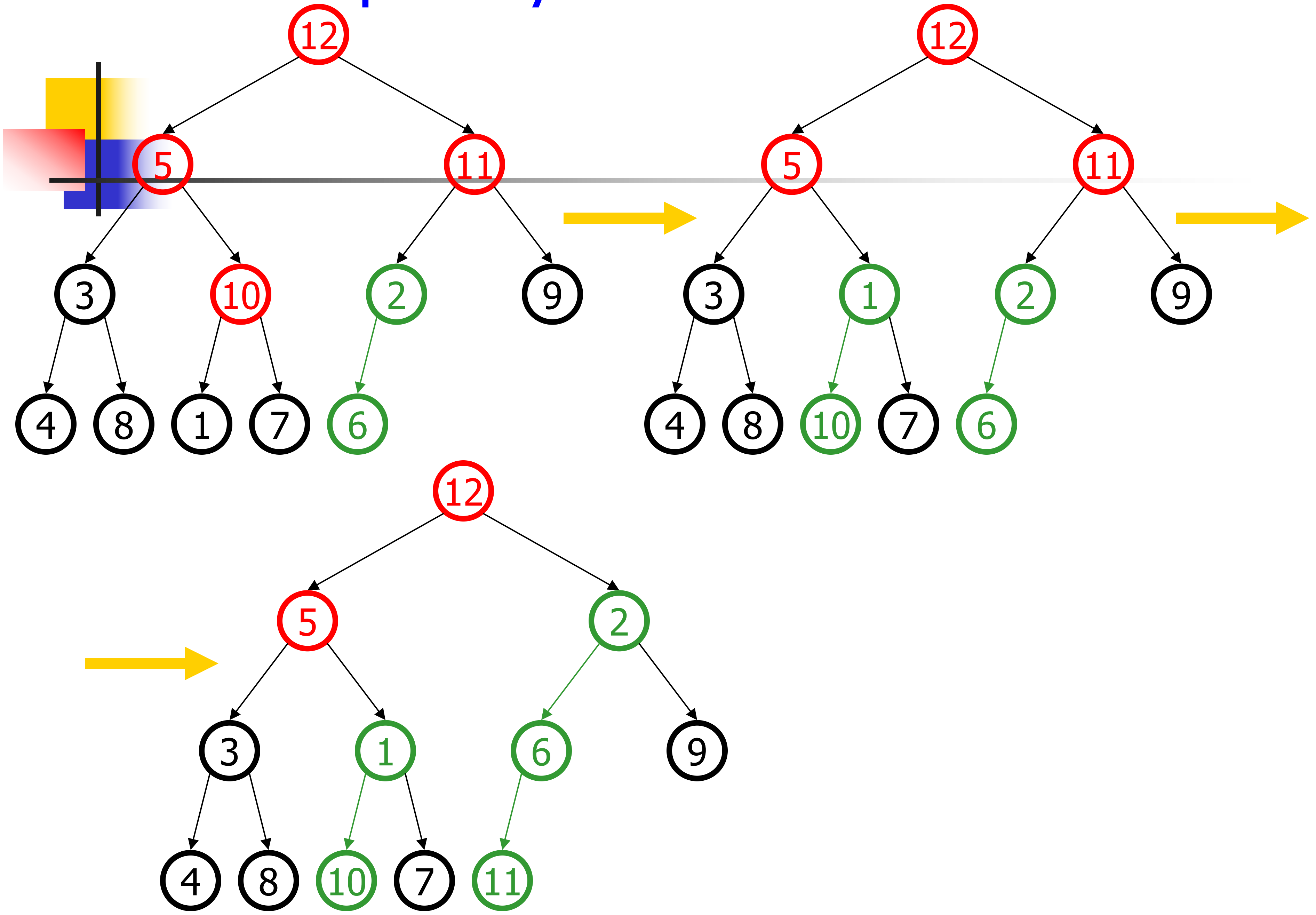
BuildHeap: Floyd's Method



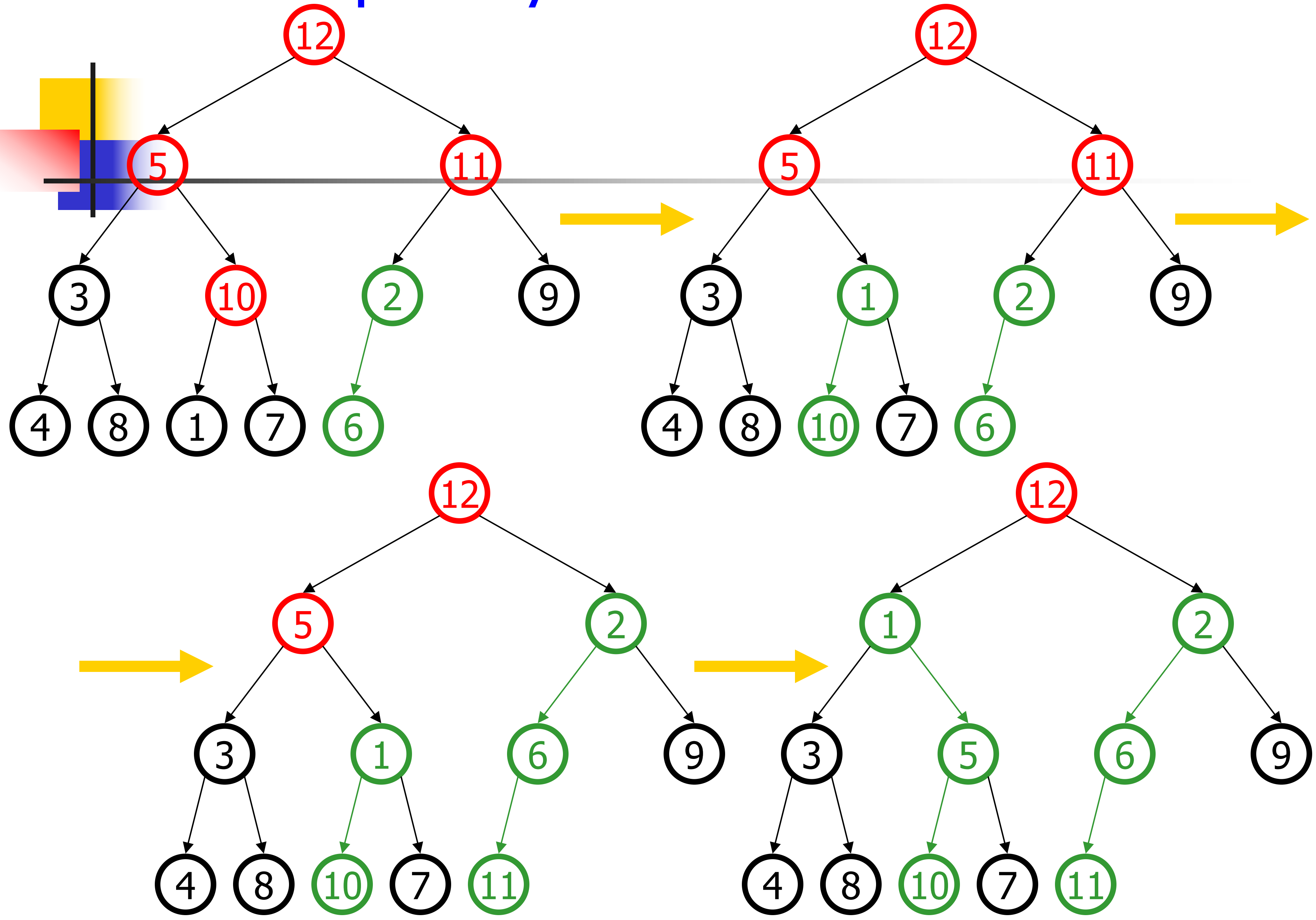
BuildHeap: Floyd's Method



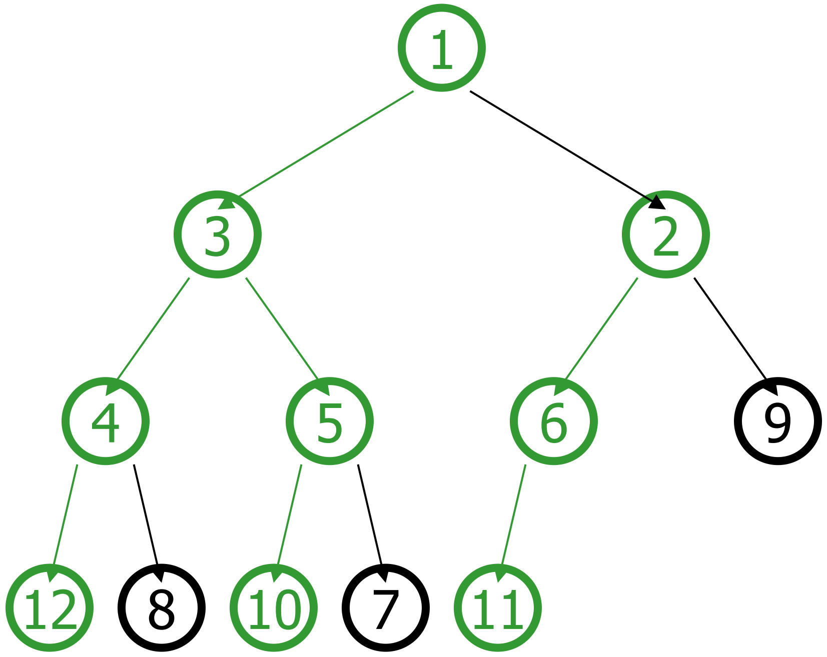
BuildHeap: Floyd's Method



BuildHeap: Floyd's Method

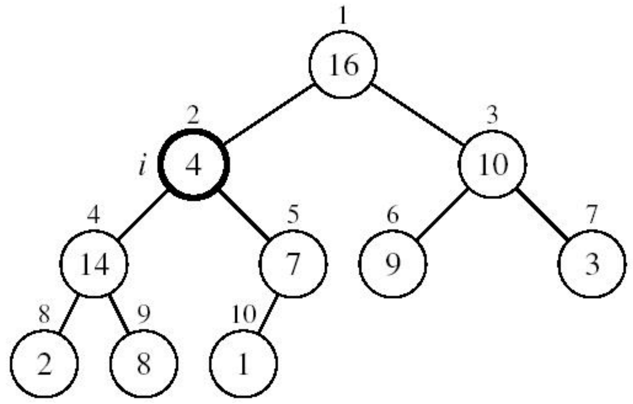


Finally…



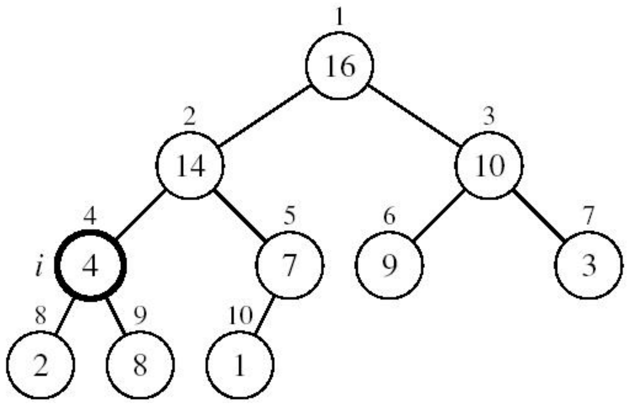
Example: MaxHeap

MAX-HEAPIFY(A, 2, 10)



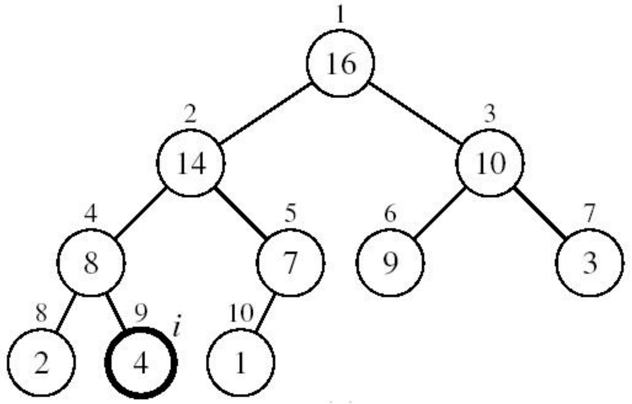
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]



Heap property restored

Exercise

Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

a) 25, 14, 16, 13, 10, 8, 12

b) 25, 12, 16, 13, 10, 8, 14

c) 25, 14, 12, 13, 10, 8, 16

d) 25, 14, 13, 16, 10, 8, 12

Exercise

Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?

a) 25, 14, 16, 13, 10, 8, 12
b) 25, 12, 16, 13, 10, 8, 14
c) 25, 14, 12, 13, 10, 8, 16
d) 25, 14, 13, 16, 10, 8, 12

Ans: a

Exercise

Consider the following max heap: 50, 30, 20, 15, 10, 8, 16

Delete a node with value 50.