Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

DS221: Introduction to Scalable Systems

Topic: Algorithms and Data Structures

CDS
The Department of Computational and Data Science

L4: Fast Searching

Search Trees, B-Tree, Hashmap

Dictionary Abstract Data Structure

- Store <key,value> as a pair
- *Lookup* the value for a given key
- Goal: Lookup has to fast
- Different implementations
  - Ordered List
  - Hash table (or Hash Map)
  - Binary Search Tree

Dictionary using List

- Dictionary stored as a List of <key,value> items
  - Unsorted Linked List and Array
  - Insertion time? Searching time?

- Dictionary stored as an *Ordered* List of <key,value> elements, ordered by key
  - Linked List vs. Array
  - What's the advantage?

Dictionary as a Sorted Array

- Idea: **Divide and Conquer**
- Narrow down the search range by half at each stage
- E.g. **find (8)**
- Start with `floor(|search space| / 2)`
- `2 5 8` $\boxed{\mathbf{9}}$ `11 17 20 22`
- `2` $\boxed{\mathbf{5}}$ `8 9 11 17 20 22`
- `2 5` $\boxed{\mathbf{8}}$ `9 11 17 20 22`

Binary search over array
Takes $O(\log_2(n))$ searches

Dictionary as a Sorted List

```
int bsearch(KVP[] list, int start, int end, int k) {
   if (end < start) return -1  // No match!
   i = start+(end-start)/2     // midpoint
   if (list[i].key == k)  // Found!
      return list[i].value
   if (list[i].key < k)   // check 2nd half
      return bsearch(list, i+1, end, k)
   else                   // check 1st half
      return bsearch(list, start, i-1, k)
}
```

Usual problem with arrays!

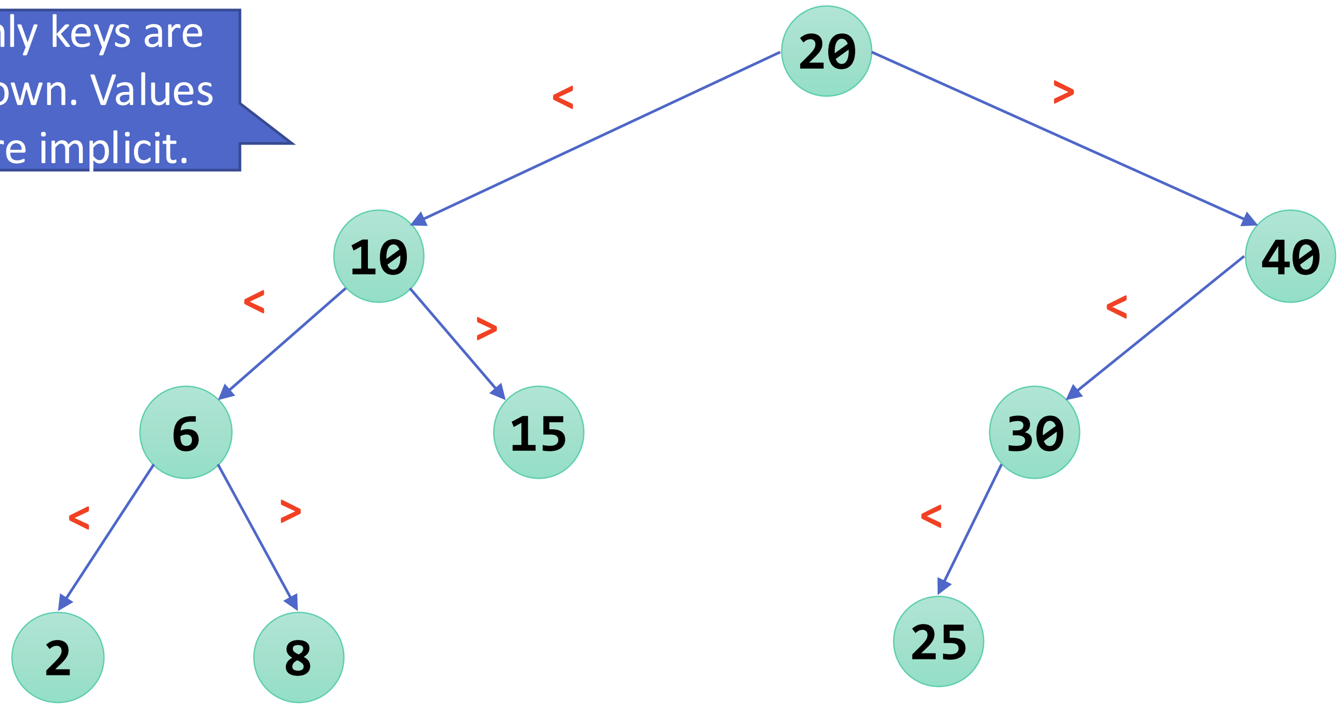
- Unused capacity
- Costly to update and maintain sorted list…many shifts

Dictionary using Binary Search Tree (BST)

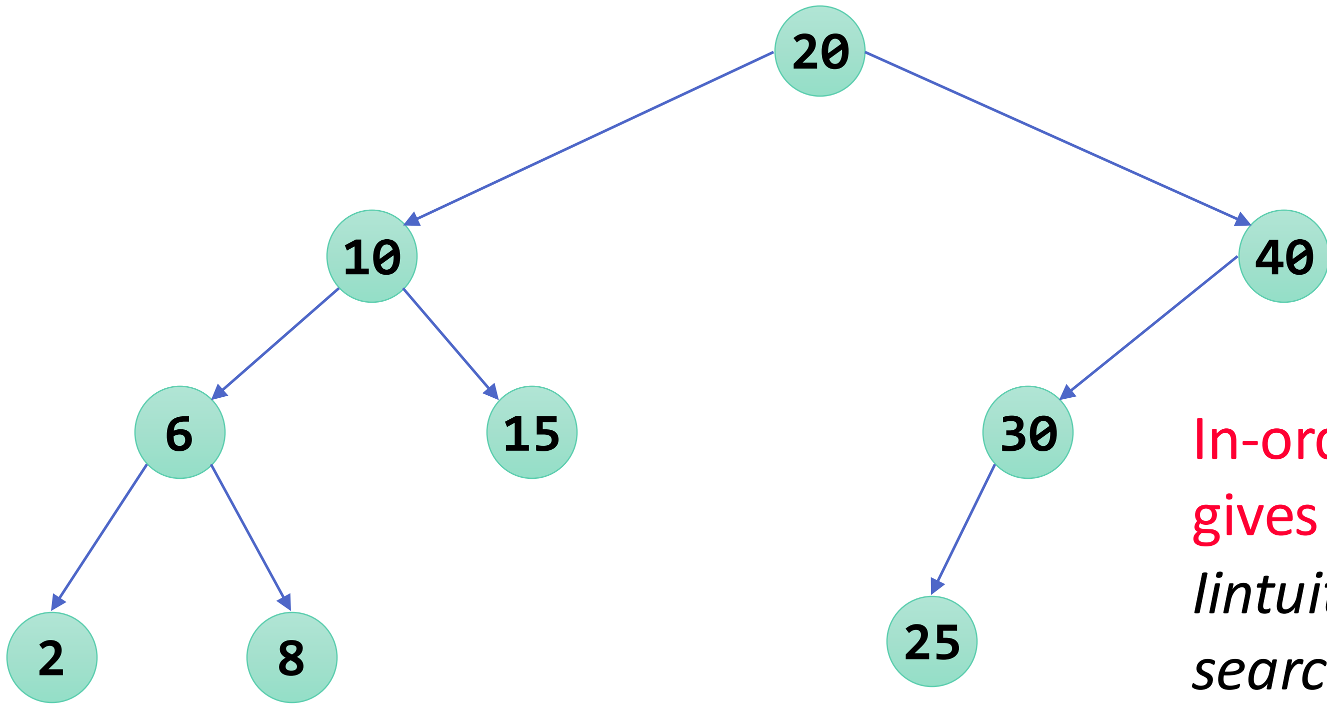
- Combining speed of binary search over array with dynamic capacity of a linked list
- A binary tree with each node having a **(key, value)** pair
- For each node x,
  - All keys in the *left subtree* of x are *smaller* than the key of x
  - All keys in the *right subtree* of x are *greater* than the key of x
- Dictionary Operations
  - `find(key)`
  - `insert(key, value)`
  - `delete(key)`

Example Binary Search Tree

Only keys are shown. Values are implicit.



The Operation find()



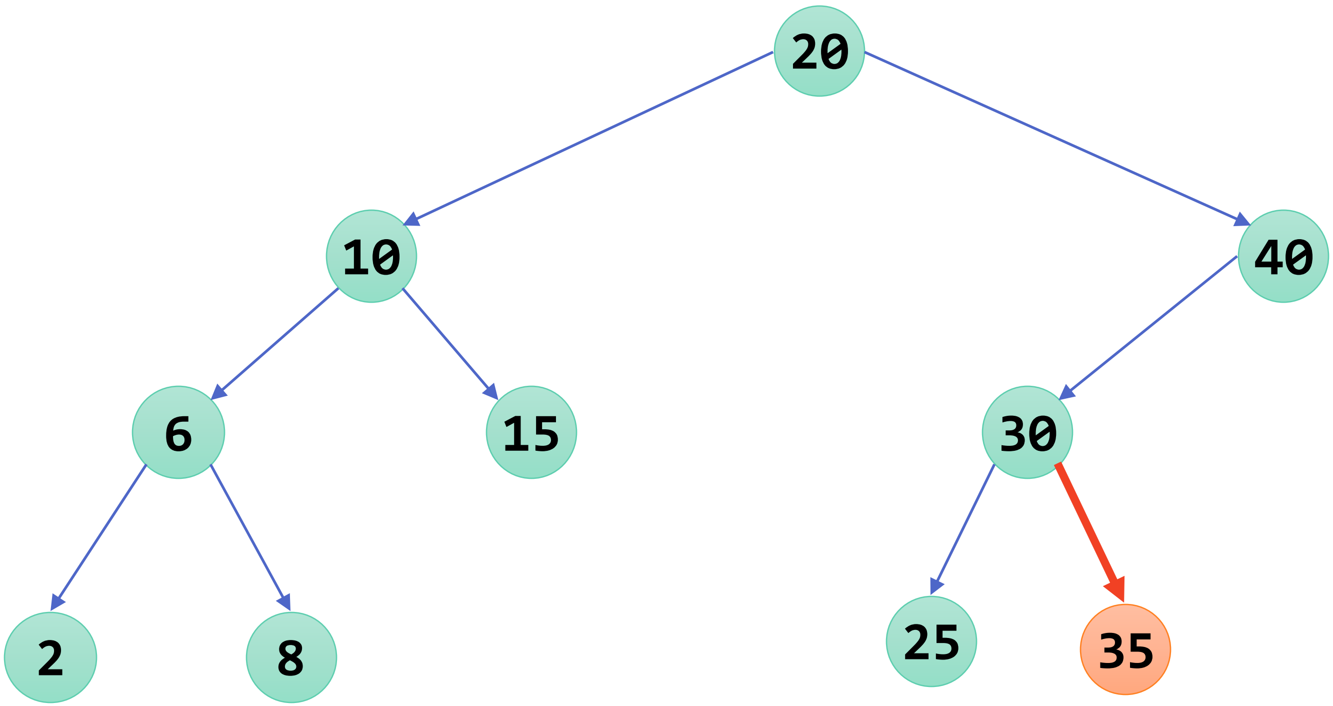
In-order traversal of BST gives a sorted array.

Iintuition behind "binary search" by partitioning into two.

2	6	8	10	-	15	-	20	25	30	-	40	-	-	-

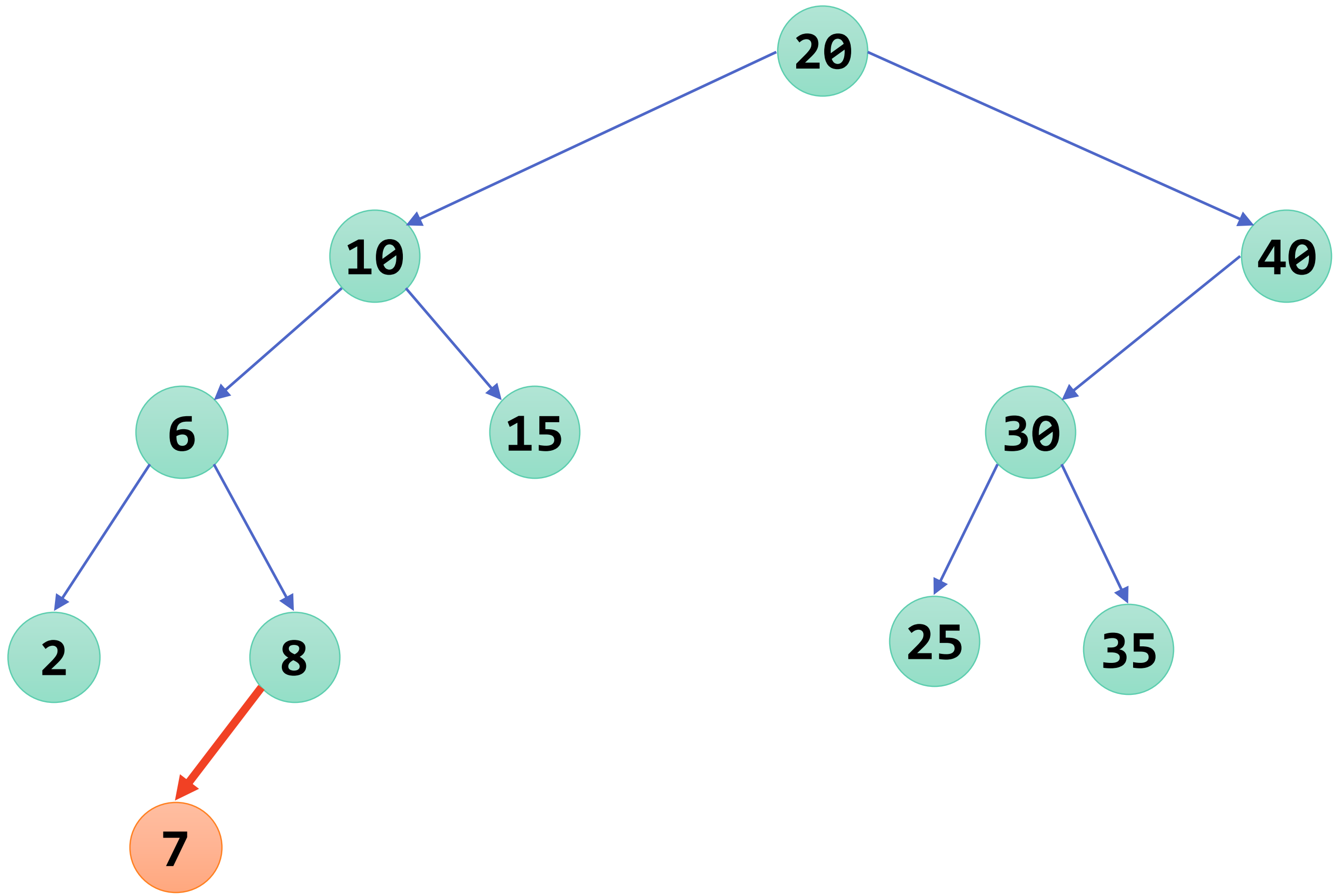
Complexity is O(height) = O(n), where n is the number of nodes/elements.

The Operation insert()



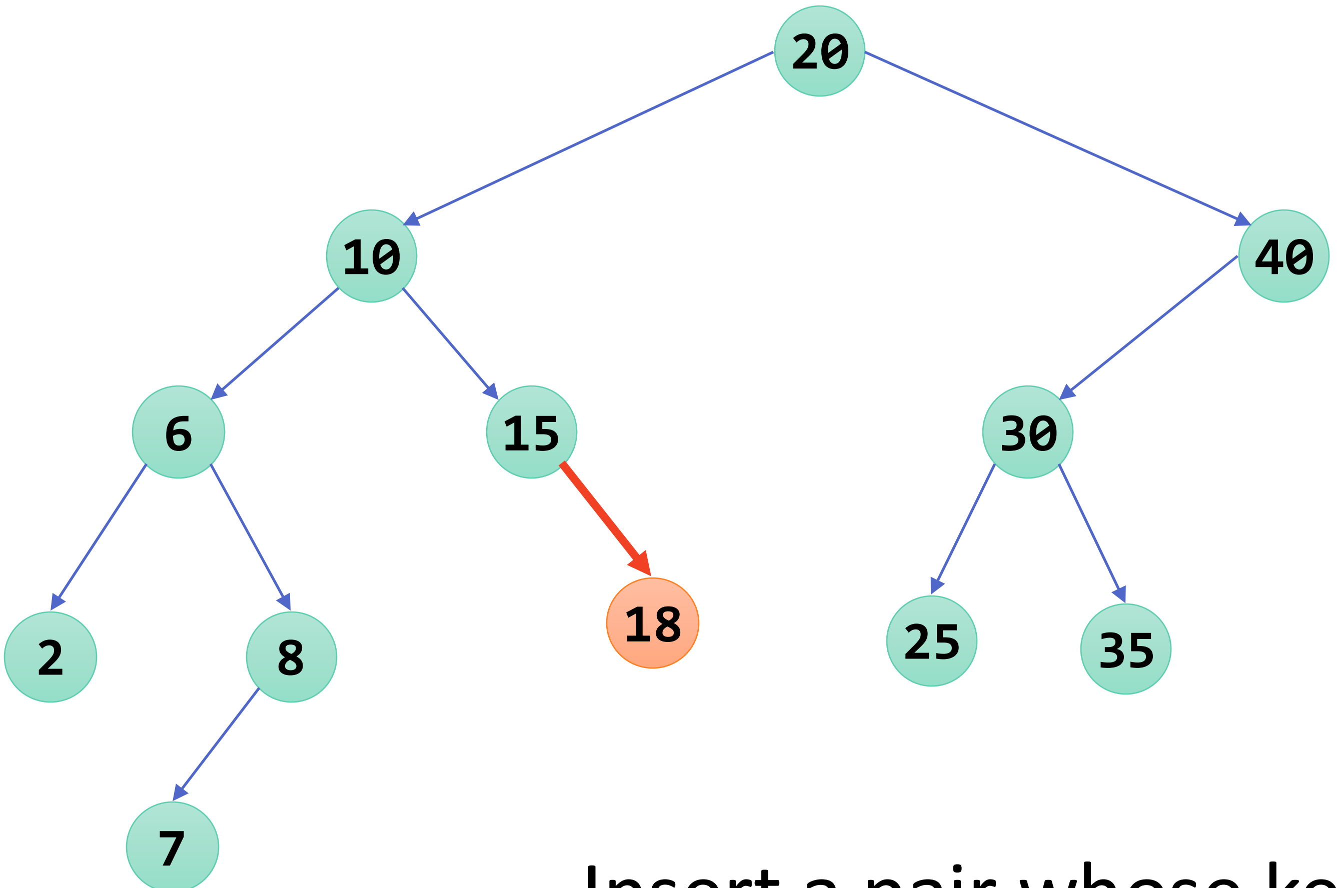
Insert a pair whose key is 35.

The Operation insert()



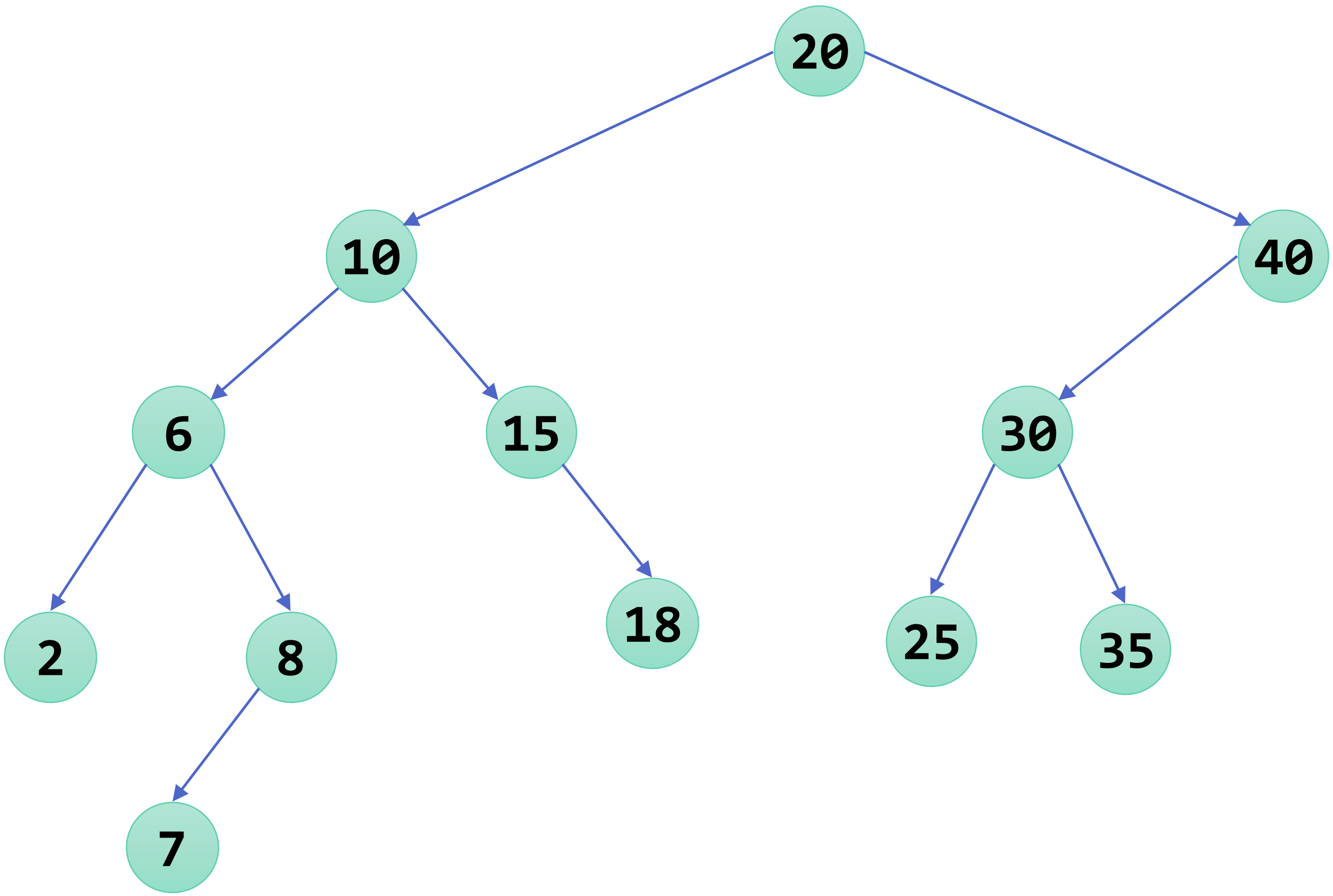
Insert a pair whose key is 7.

The Operation insert()



Insert a pair whose key is 18.

The Operation insert()



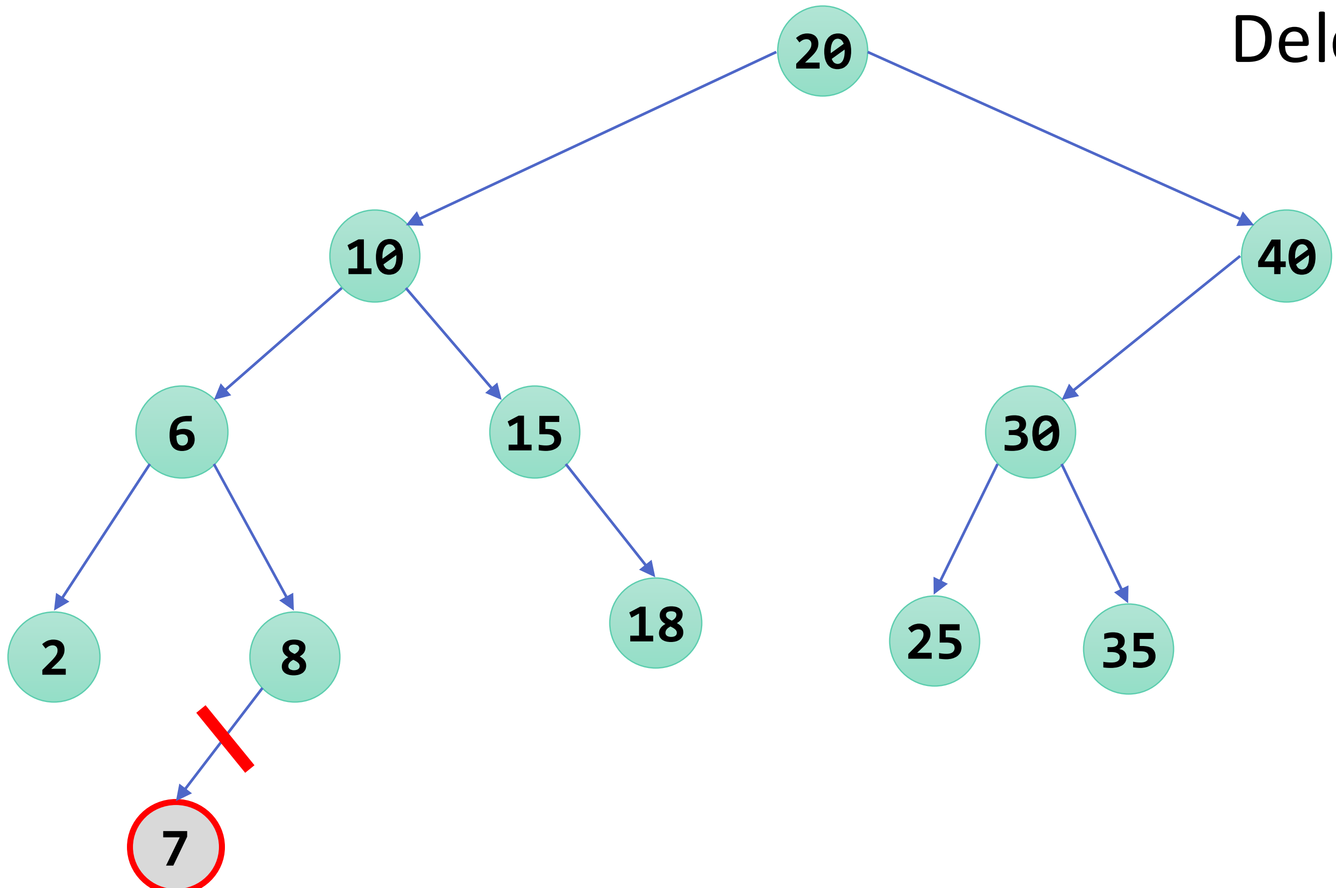
Complexity of insert() is O(height).

The Operation delete()

- Three cases:
  - Element is in a leaf.
  - Element is in a degree 1 node.
  - Element is in a degree 2 node.

Delete From A Leaf

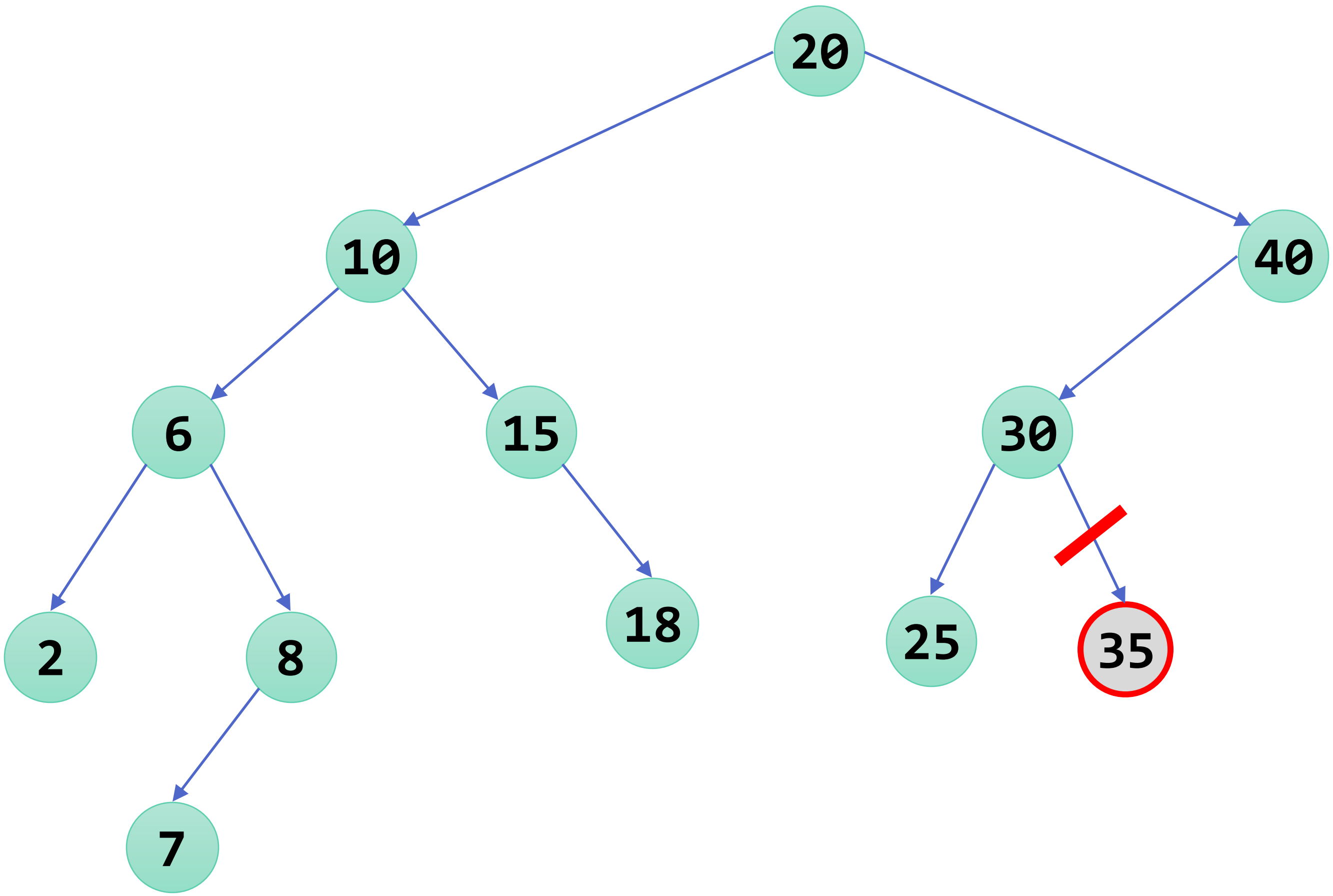
Delete key = 7



Delete a leaf element.

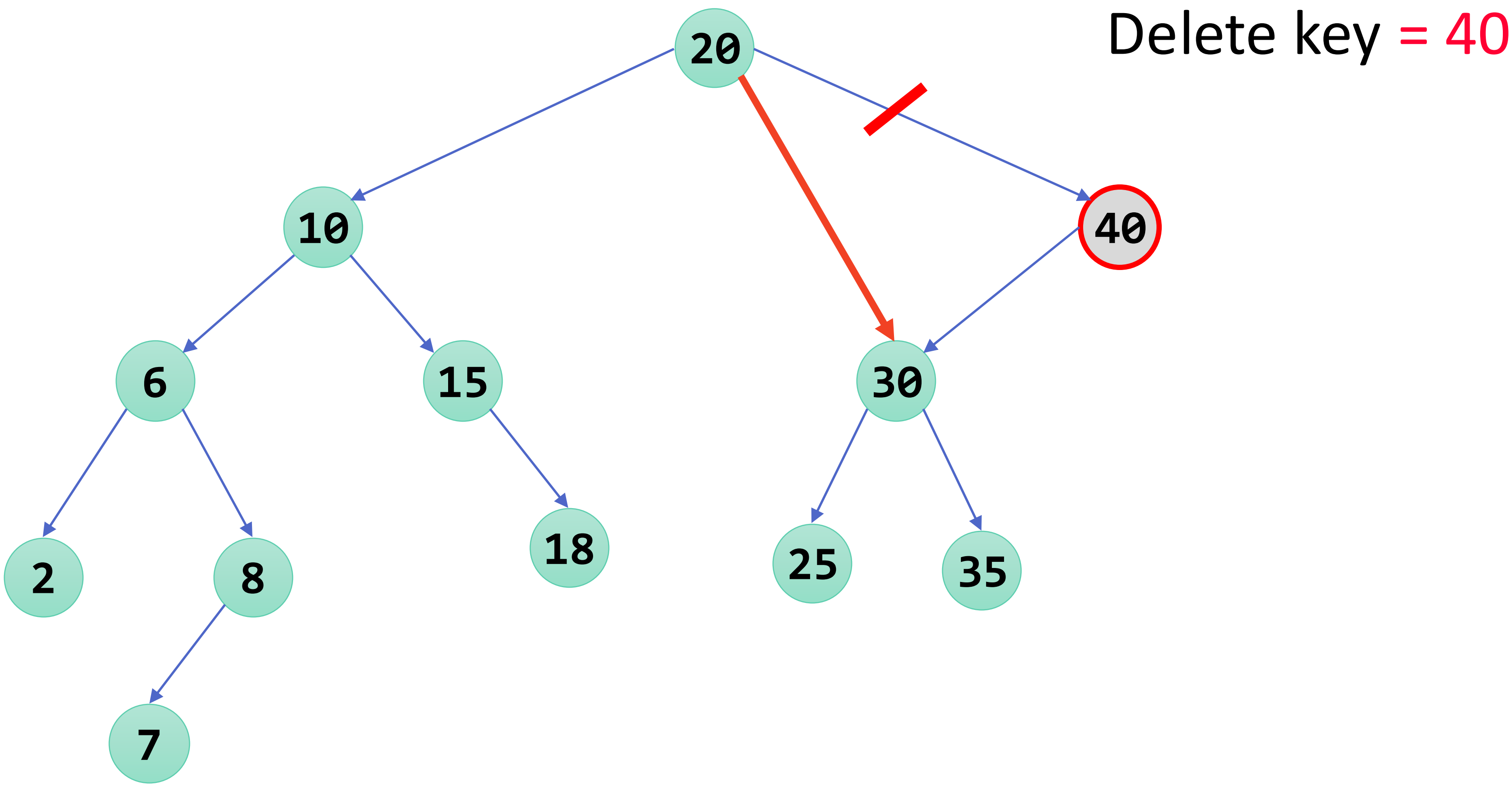
Set parent to NULL

Delete From A Leaf



Delete a leaf element. key = 35

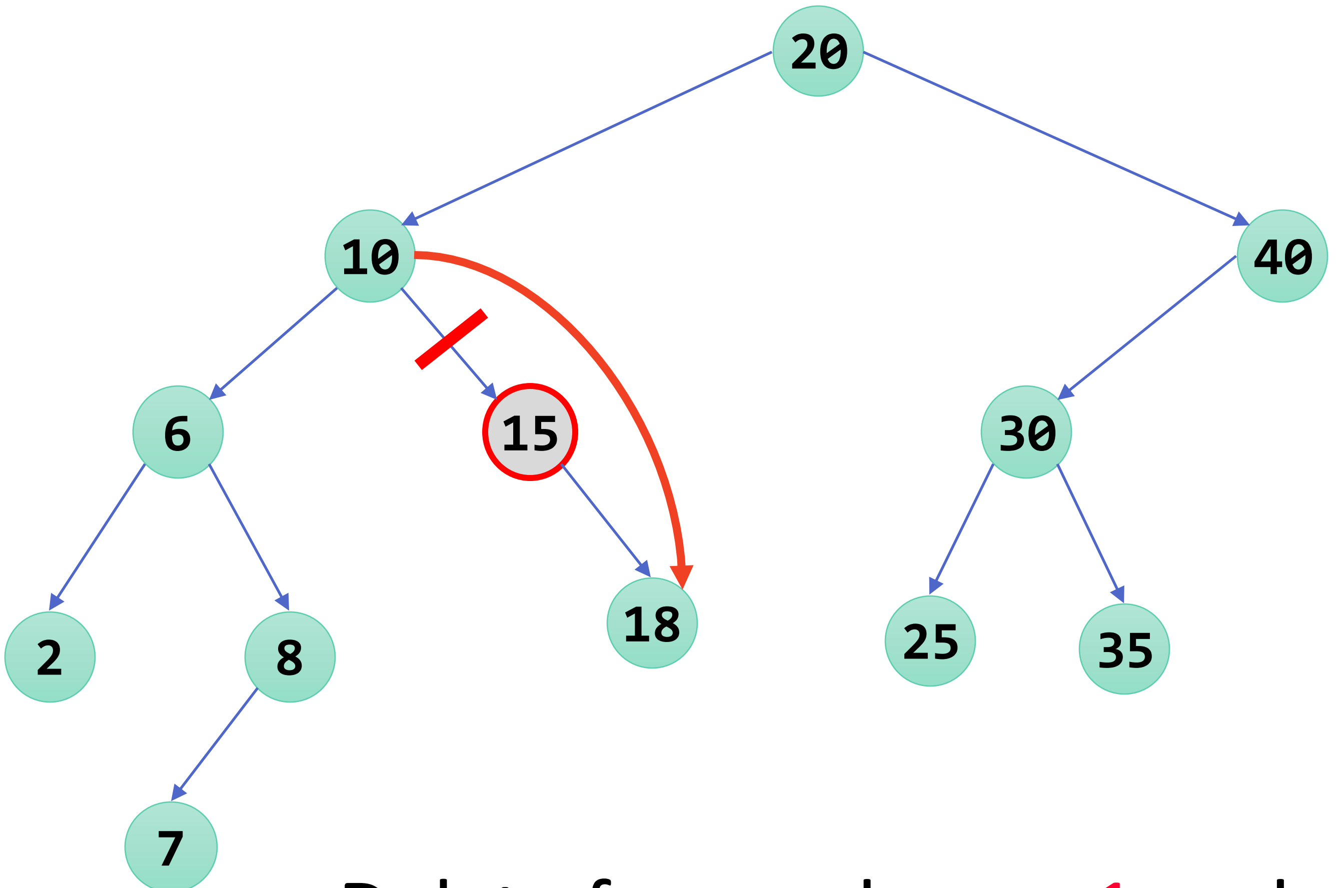
Delete From Degree 1 Node



Delete from a degree 1 node.

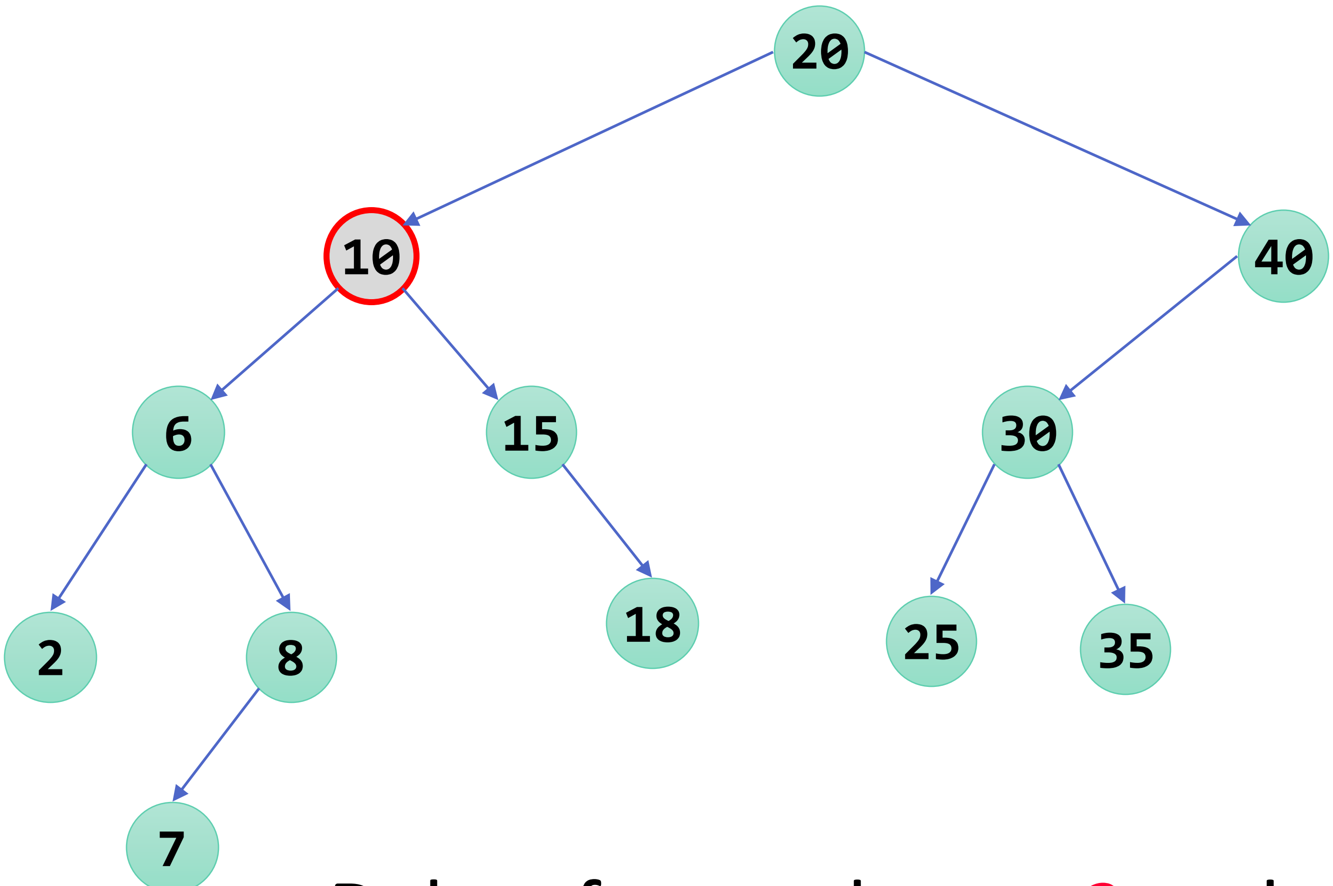
Point parent to child.

Delete From Degree 1 Node



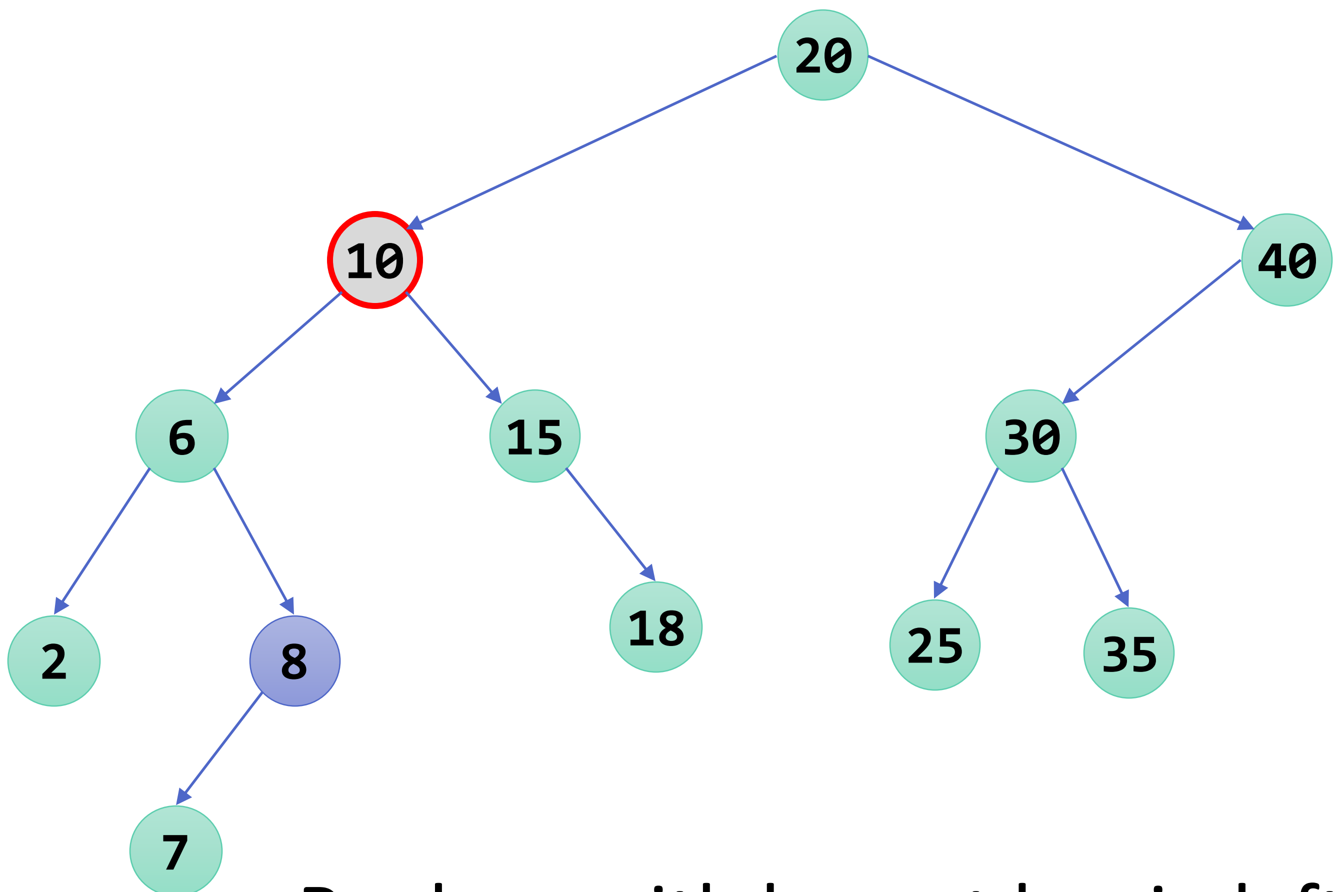
Delete from a degree 1 node. key = 15

Delete From Degree 2 Node



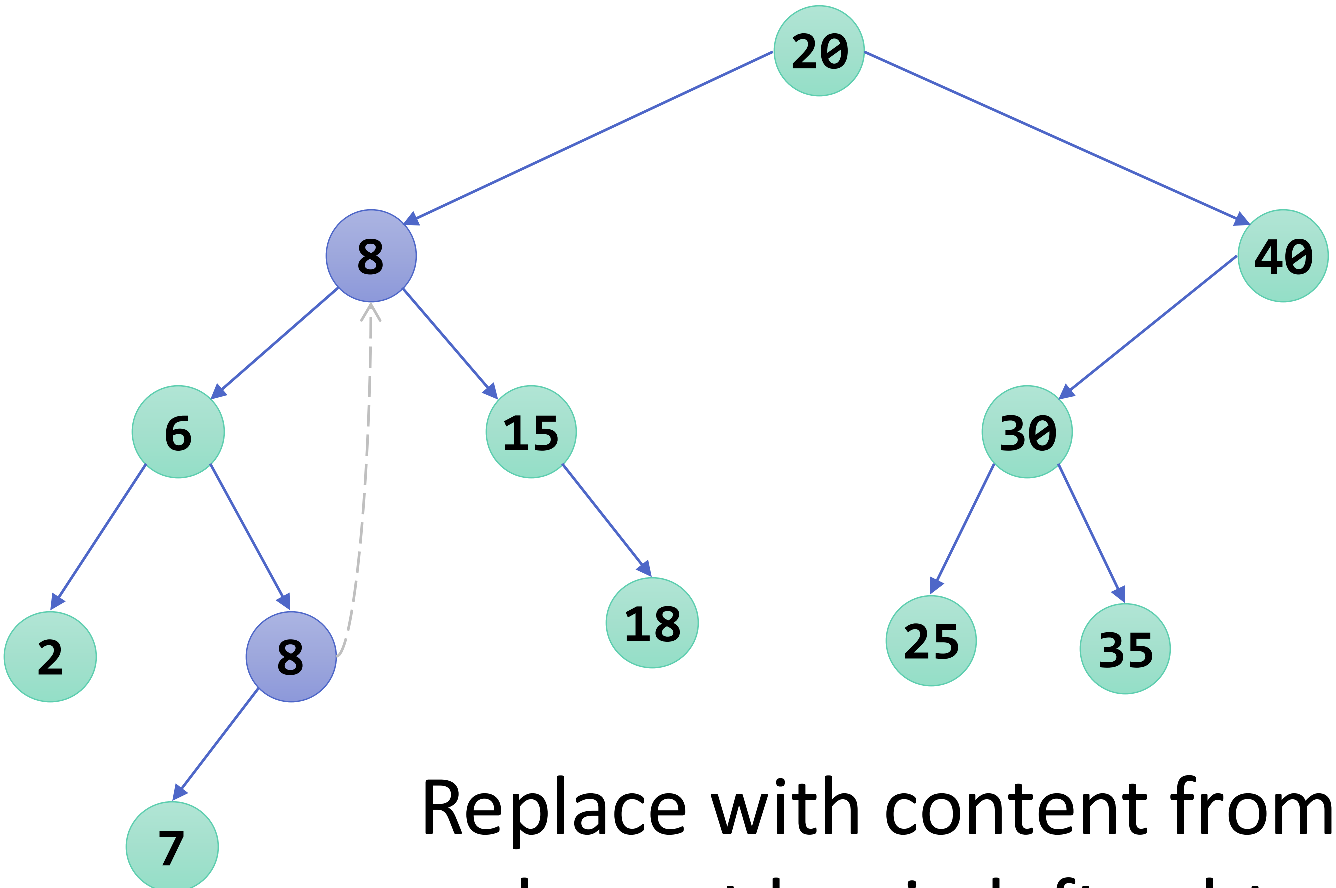
Delete from a degree 2 node. key = 10

Delete From Degree 2 Node



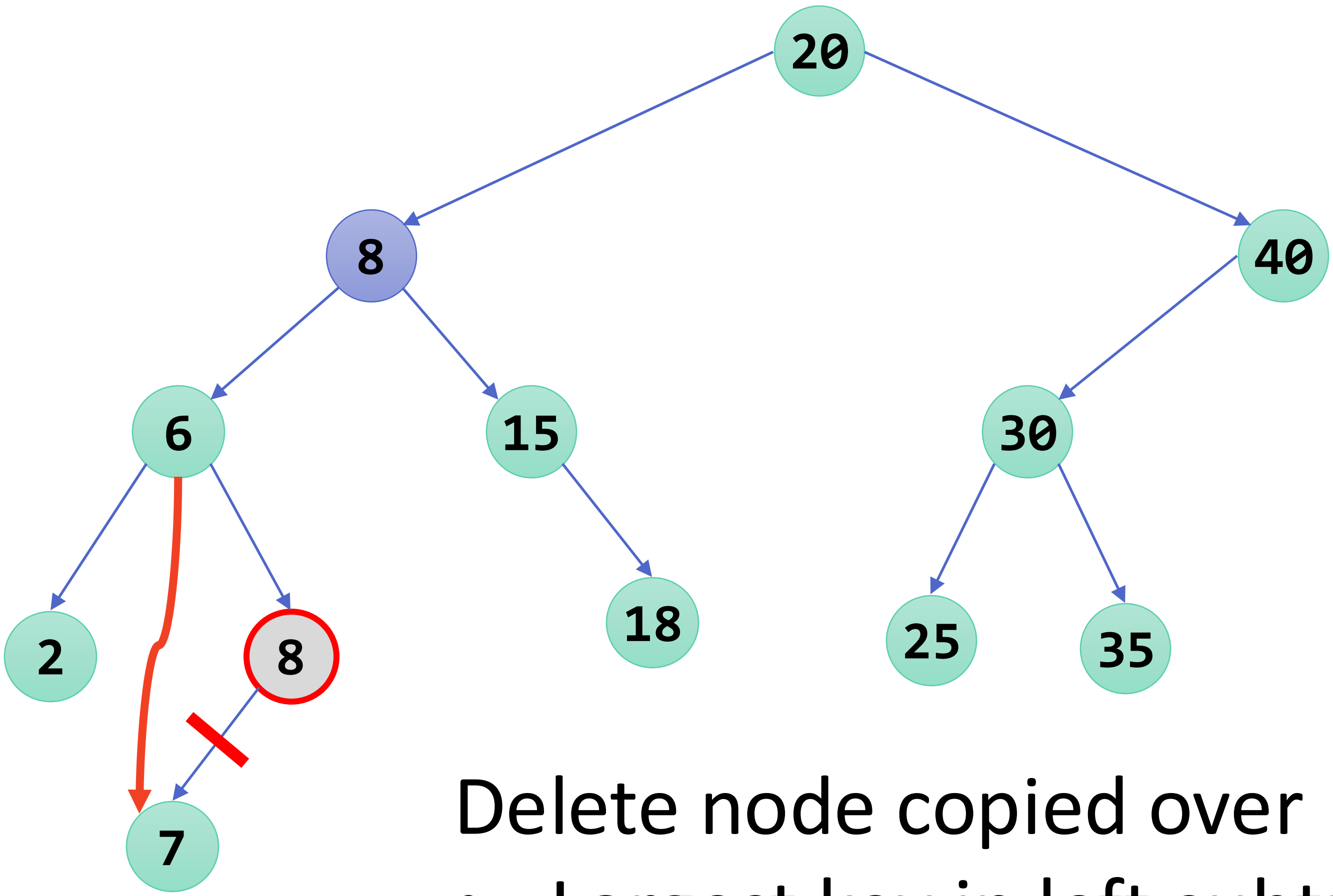
Replace with largest key in left subtree (or smallest in right subtree).

Delete From Degree 2 Node



Replace with content from
- <u>largest</u> key in <u>left</u> subtree, or
- <u>smallest</u> in <u>right</u> subtree

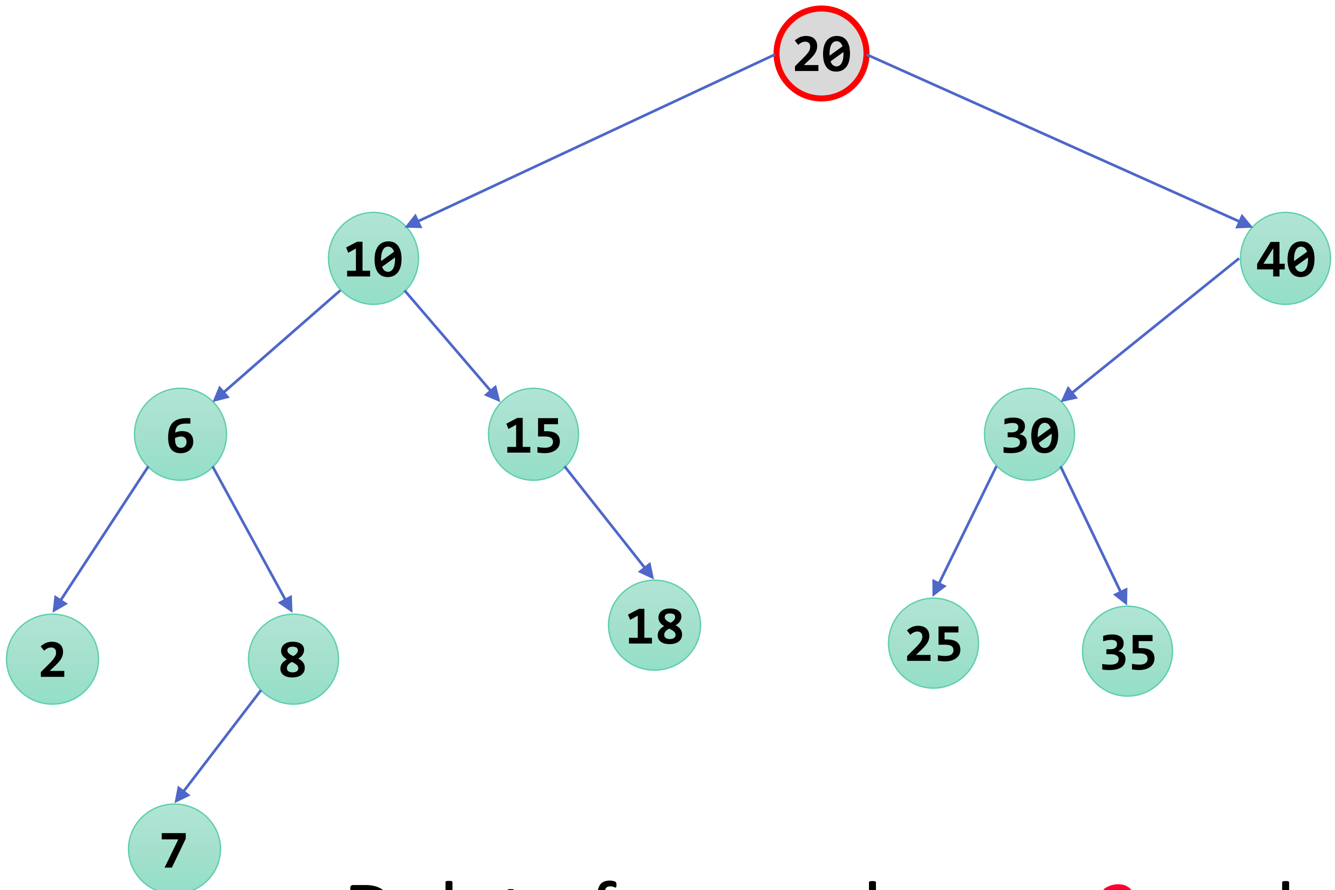
Delete From Degree 2 Node



Delete node copied over

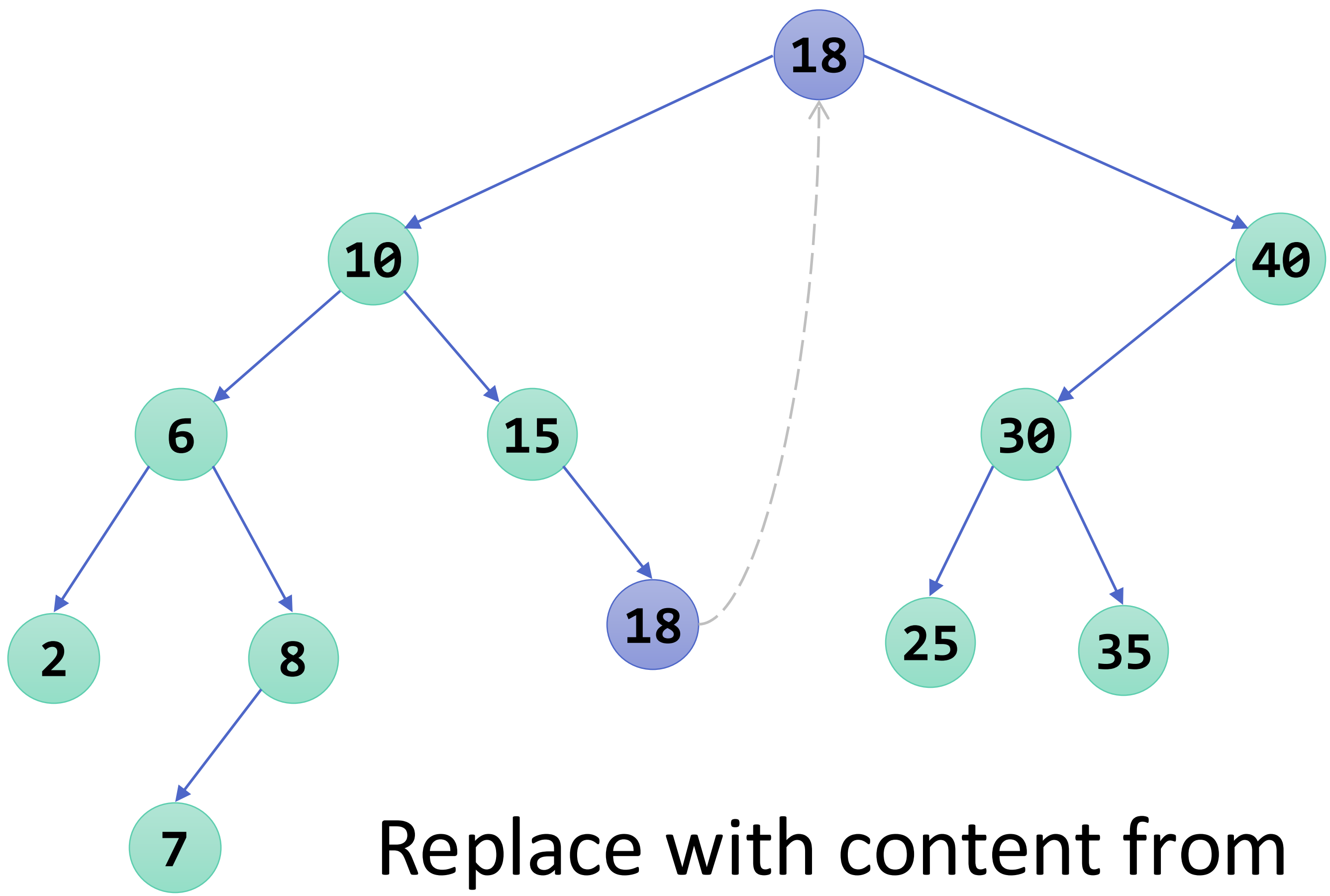
- Largest key in left subtree will be a leaf, or degree 1 node.

Delete From Degree 2 Node



Delete from a degree 2 node. key = 20

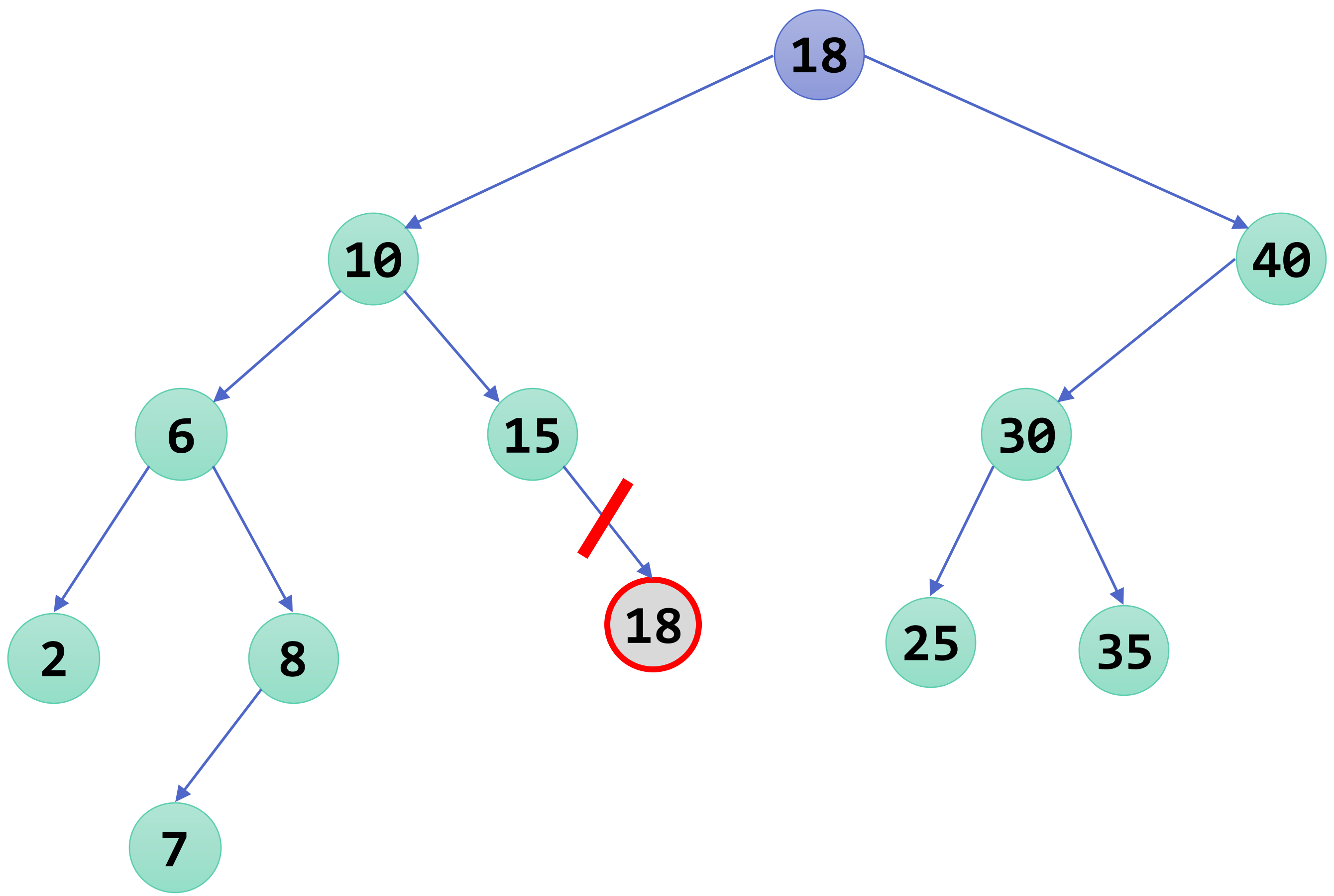
Delete From Degree 2 Node



Replace with content from

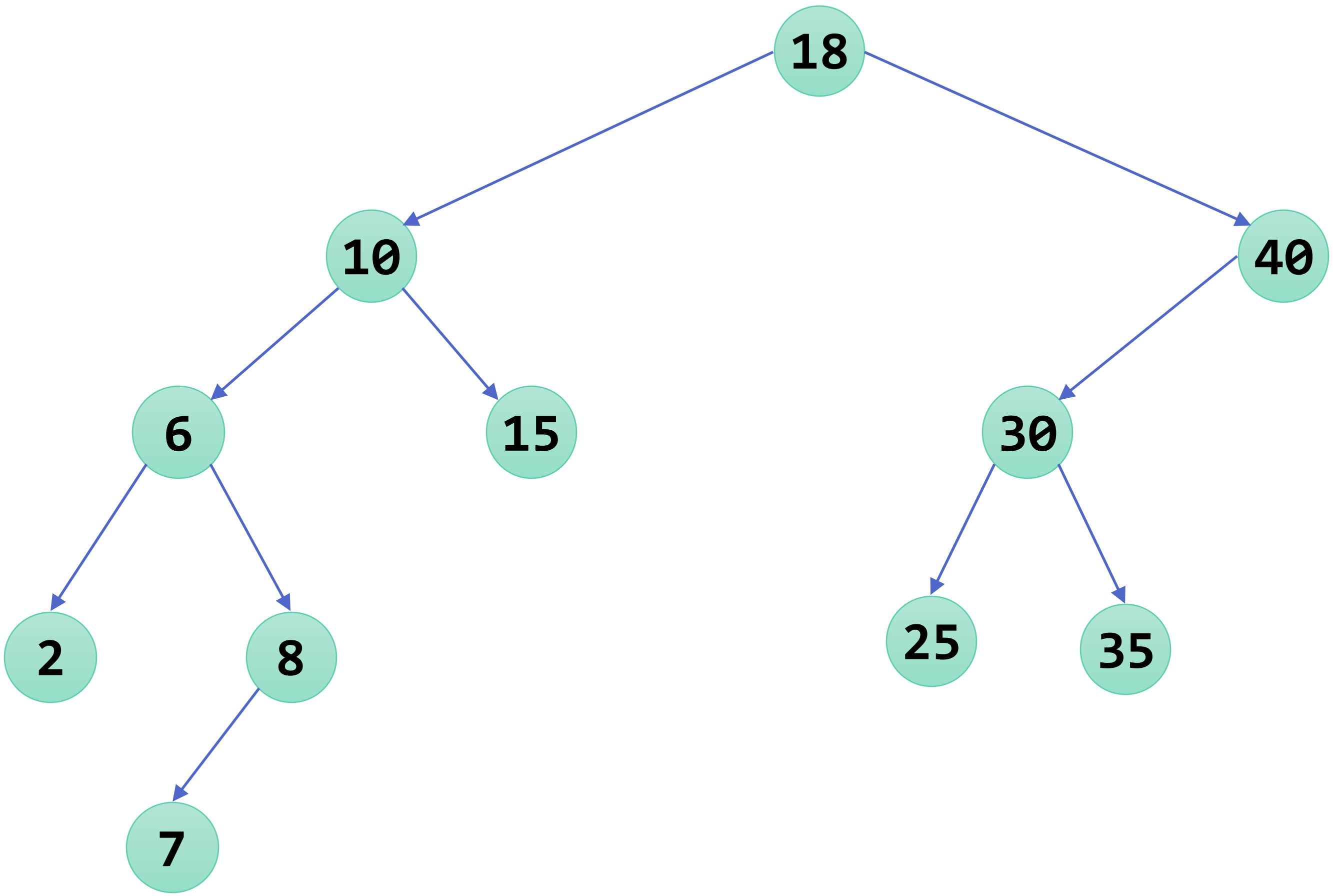
- <u>largest</u> key in <u>left</u> subtree, or
- <u>smallest</u> in <u>right</u> subtree

Delete From Degree 2 Node



Delete node copied over

Delete From Degree 2 Node



Complexity is O(height)

Tree Imbalances

- Similarities between **BST vs. Sorted Array**
- Inserting and Deleting in specific orders can cause tree to be imbalanced
  - E.g. insert in sorted ascending/descending order
  - Height of left and right subtrees are very different, skewed
- Causes complexity to tend to O(n) rather than O(log(n))
- Periodically *rebalance* if skew greater than a threshold
  - *Balanced* BST, e.g., AVL Tree, Red-Black Tree, etc.

Hash Table

- Uses a 1D array (or table) table[0:b-1]
  - Each position of this array is a **bucket**
  - Number of buckets is b
  - A bucket can normally hold only one dictionary pair: <key, value>
    - But larger capacity allowed per bucket as well
    - Bucket sizes can be unbounded as well
- Uses a hash function h that converts each key k into an index in the range [0, b-1].
  - h(k) is the "home bucket" for key k.
- Every dictionary pair is stored in its home bucket

  table[h(item.key)] = item

Ideal Hashing Example

- KVPs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is **table[0:7]**, b = 8.
- Hash function **h=key/11**
- Pairs are stored in table as below

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Lookup, Insert and Delete are done similarly
  - Apply hash, find bucket, perform op.
  - Take O(1) time to apply hash and do array access

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Where does (99,k) go?
- Hash function causes us to go beyond table size
- **Simple fix**: do a "mod" with the bucket size by default
- **h = (k / 11) % 8**

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Where does (26,g) go?
- Keys 22 and 26 have the same *home bucket*, are synonyms with respect to the hash function used
  - This is a **collision**
- The home bucket for (26,g) is already occupied
  - And capacity of bucket is only 1 item
  - This is called an **overflow**

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

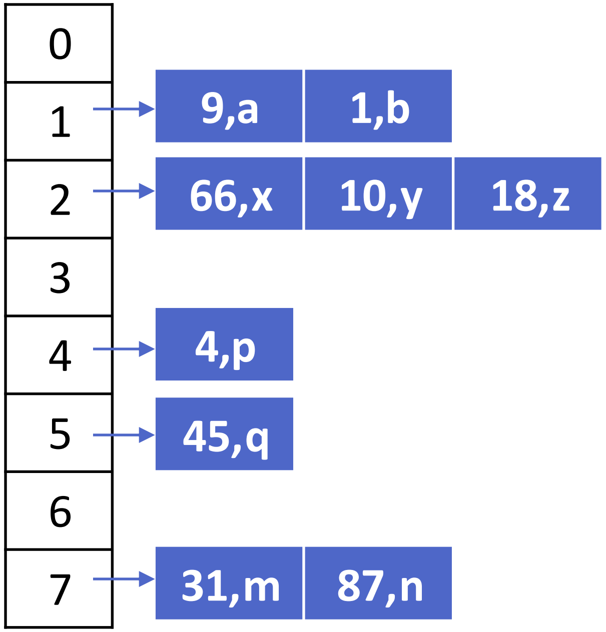
- A **collision** occurs when the home bucket for a new pair is occupied by a pair with a different key
- An **overflow** occurs when there is no space in the home bucket for a new pair
  - E.g. if each bucket has capacity to hold two values for same key, and more than 2 values for the key are inserted
- If a bucket has a capacity of 1, *collisions and overflows occur together*
  - *Can we allow buckets to hold multiple item? Unbounded items?*
  - Using a linked list for each bucket item is called “Chaining”
- Need a method to handle overflows

Designing/Selecting a Hash Table

- Choice of hash function
  - Should be **fast** to compute
  - Distributes keys **uniformly** throughout the table
  - Each bucket should have an **equal probability** of receiving a key
  - E.g. **h=k%b** is a *uniform hash function* for keys in the range [0..r] … *assuming all keys have equal probability of occurrence*
  - Each buckets gets about **ceil(r/b)** or **floor(r/b)** keys
- Size (number of buckets) of hash table
  - Decides frequency of collision
- Overflow handling method

Hash Table using Array & Linked List

- Buckets with unbounded capacity
  - Bucket as a linked list
- Hash function gives array index
- Array contains pointer to head of linked list
  - Items are <key,value> pairs
- Traverse list to lookup element
- What if key not present?
- Time complexity for Insert? Lookup?



Open Addressing to handle Overflows

- All elements are stored in the hash table
  - Elements to store <= capacity of table
- Each table entry contains either a <key,value> element or *null*
- While inserting an element **systematically probe** table slots if overflow occurs
- While searching for an element **systematically probe** table slots if bucket does not match key

Open Addressing

- Modify the hash function to take the *probe number **i*** as second parameter
  - `h: K x {0,1,…b-1} → {0,1,…b-1}`
- Hash function, *h*, also determines the sequence of slots "probed" for a given key
- Probe sequence for a given key ***k*** is the series of buckets `h(k,0),h(k,1),…,h(k,b-1)`
  - Use `h(k,0)` as bucket if no overflow
  - Else probe each bucket from successive hash fns., i.e. a permutation of `<0,1,…b-1>`

Linear Probing

- If the current location is occupied, try the next location

```
LPInsert(k)
  If (table is full) return error
  probe = h(k)
  while (table[probe] is occupied)
    probe = (probe+1) mod b
  table[probe]=k
```

Linear Probing – Example

- Home bucket h(k) = k mod 17
- Insert keys: 6, 12, 34, *29*, 28, *11*, *23*, *7*, *0*, 33, *30*, *45*

0				4				8				12				16
34	*0*					6	*23*	*7*			28	12	*29*	*11*	*30*	33

0				**4**				**8**				**12**				**16**
34	*0*	*45*				6	*23*	*7*			28	12	*29*	*11*	*30*	33

Lookup in Linear Probing

- Search for a key: Go to *(k mod 17)* and continue looking at successive locations till we find *k* or reach empty location.
  - Longer (unsuccessful) lookup time
  - Deletion?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

Deletion

- Shift all elements to previous location?
  - Costly
- Instead, place flag at vacated location
  - `neverUsed=false`
- Lookup continues till `neverUsed=true`
- Insert puts element in first location with `neverUsed=true`, sets it to `false`
- Too many markers degrade performance
  - Perform Rehashing

Complexity Of Dictionary Operations find(), insert()

- Given **n** elements in the dictionary

Data Structure	**Worst Case**	**Expected**
Hash Table (linear probing)	O(n)	O(1) *
Binary Search Tree	O(n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)

* Assumptions: (i) Each key's hash is uniform and independent over the 'b' buckets and (ii) the *load factor* (= n/b) is a constant strictly less than one

For a proof, see https://en.wikipedia.org/wiki/Linear_probing

Demo

- Demo: Chaining vs Linear probing vs BST

https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture_5

B Tree

B-Tree: Searching External Storage

- Main memory (RAM) is fast, but has limited capacity
- Different considerations for in-memory vs. on-disk data structures for search
- Problem: Database too big to fit memory
  - Disk reads are slow
- Example: 1,000,000 records on disk
- Binary search might take 20 disk reads
  - $\log_2(1M)$ ~= 20

www.cs.nott.ac.uk/~psznza/G52ADS/btrees2.pdf
http://www.cs.carleton.edu/faculty/jgoldfea/cs201/spring11/inclass/Tree/BTreefinalNew.pdf

Searching External Storage

- But disks are accessed "block at a time" by OS
- Blocks are typically 1KiB–4KiB in size
  - Can span multiple "sectors" on HDD
  - Access time per block
  - 10-15 ms for HDD
  - <1ms for SSD
- Say 1KiB block, and say each block can hold 100 records
  - 10,000 blocks for 1M records

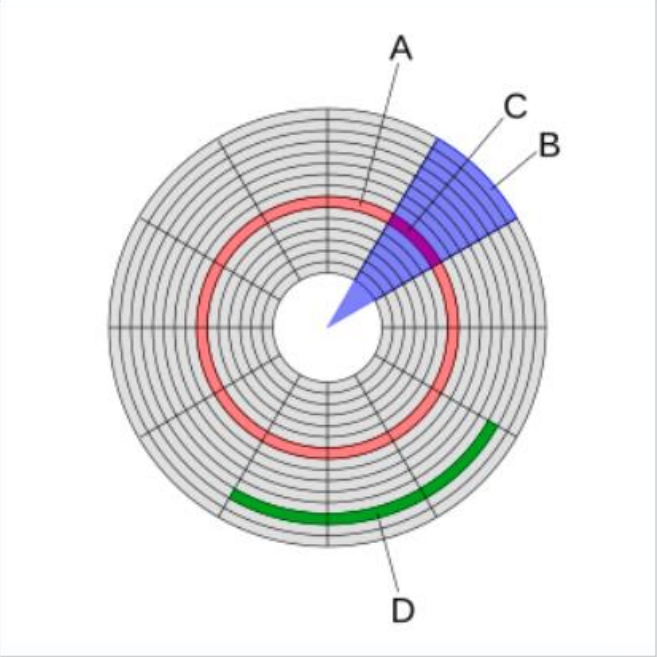
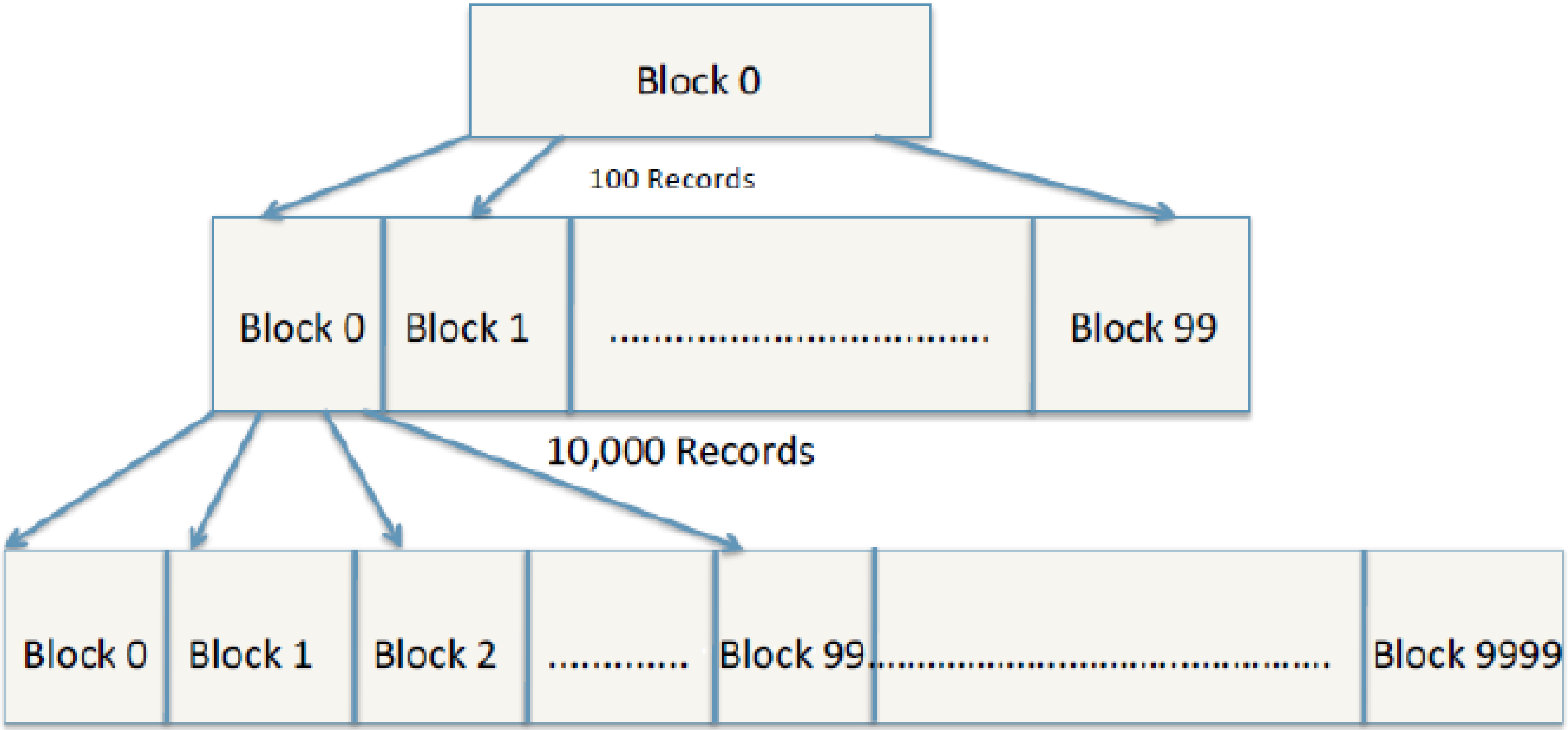


Figure 1: Disk structures:
(A) Track
(B) Geometrical sector
(C) Disk sector
(D) Cluster

Source: Wikipedia

Searching External Storage



1,000,000 Records

B-Trees

- Data structures for external memory, not main memory
  - Goal is to reduce number of block accesses, not number of comparisons
- Similar to *binary* search tree
  - But allow more than 1 value and 2 children per node
  - Each node is one disk block with data records plus block addresses of children
- B-Trees
  - Proposed by R. Bayer and E. M. McCreigh in 1972.
  - “Bayer”, “Balanced”, Bushy”, “Boeing” trees?
  - Different from **binary** trees
- NOTE
  - An in-memory data structure generally outperforms an on-disk one. Thus, an in-memory binary tree will be faster than an on-disk B-tree.
  - However, when working on disk, a B-tree is more efficient than a binary tree.

B-Trees

- Like a BST, nodes contain alternating records (keys & values) and child pointers (blocks).
  - A node with *k* records has *(k + 1)* children.
- Key ordering rule:
  - Keys in a node are sorted in increasing order.
  - Every key is greater than all keys in its left child's subtree and smaller than all keys in its right child's subtree.
- Bounds on minimum and maximum number of children in a node. For an 'order m' tree:
  - Each node can have **at most** 'm' children
  - Each internal node (except root) must have **at least** $\lceil m/2 \rceil$ children

E.g. Largest-sized node in order 5 B-Tree

Blk id	<k,v>	Blk id	<k,v>	Blk id	<k,v>	Blk id	<k,v>	Blk id

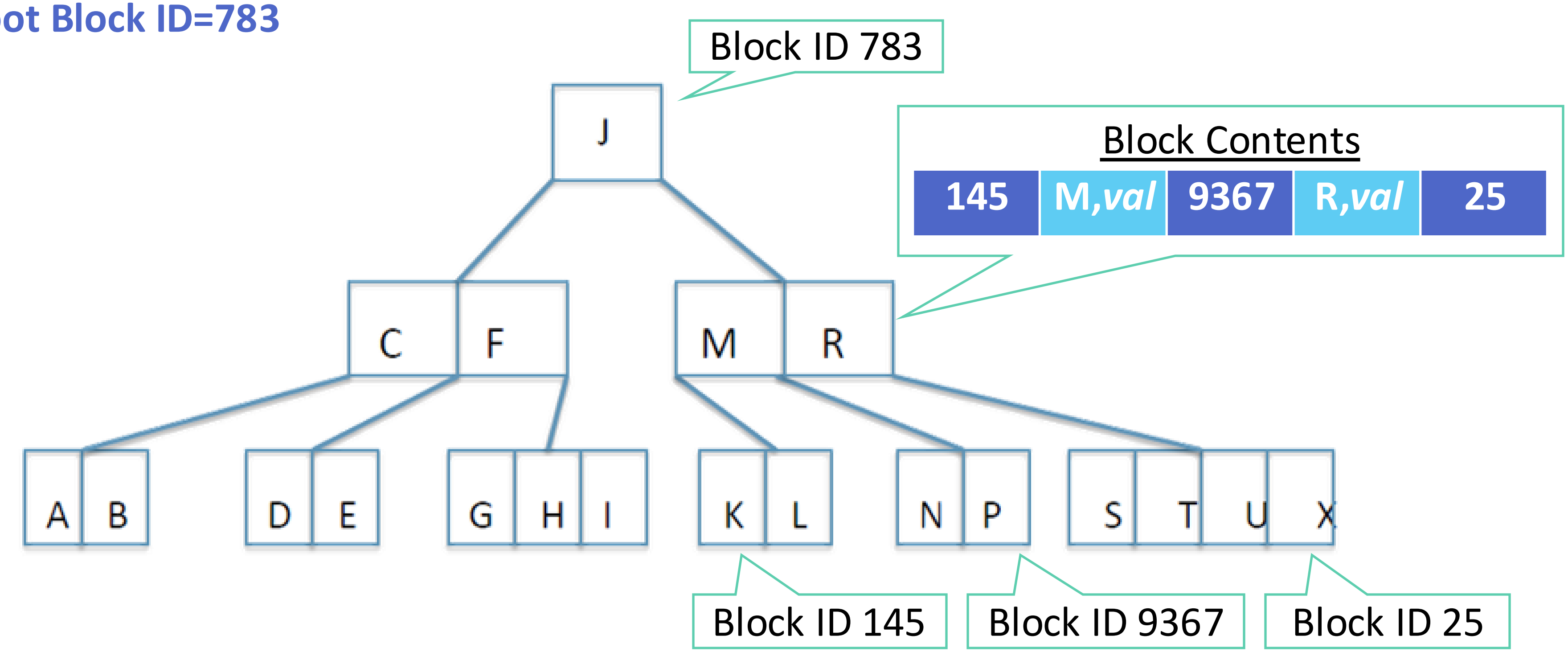
Blk id	<k,v>	Blk id

…and its smallest-sized node

B-Tree Search (Order 5)

A G F B K D H M J E S I R X C L N T U P

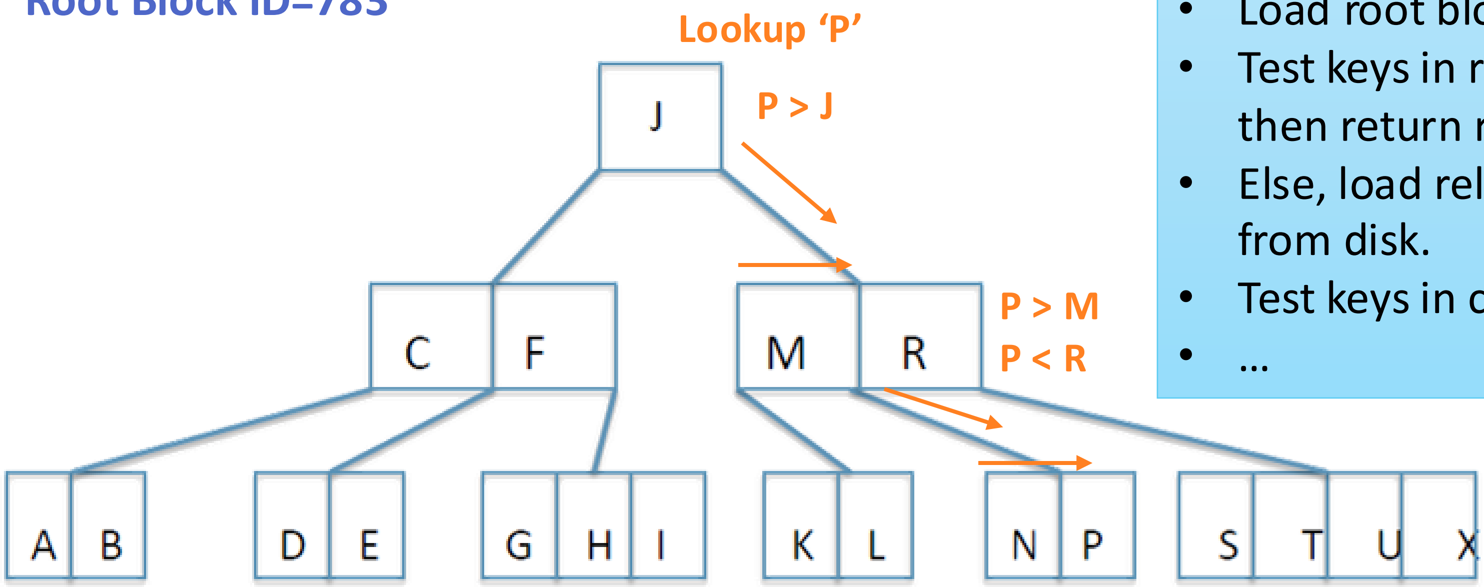
Root Block ID=783



B-Tree Search (Order 5)

A G F B K D H M J E S I R X C L N T U P

Root Block ID=783



Lookup is similar to BST

- Load root block from disk
- Test keys in root block. If match, then return record.
- Else, load relevant child block from disk.
- Test keys in child block…
- …

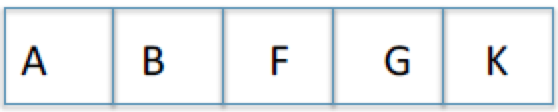
B-Tree Creation

A G F B K D H M J E S I R X C L N T U P

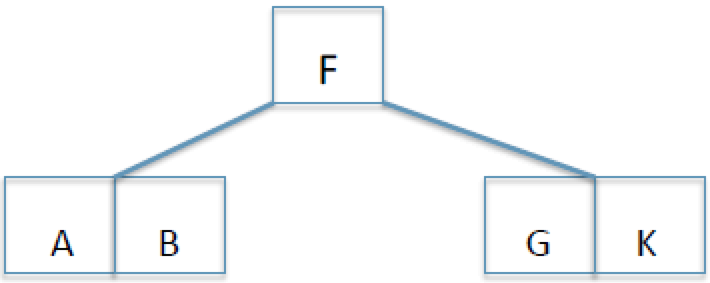
A	B	F	G	K

B-Tree Creation

A G F B K D H M J E S I R X C L N T U P

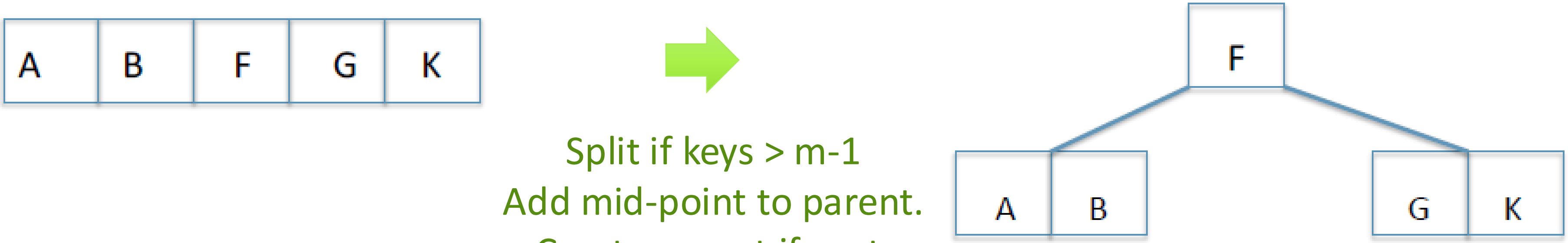


Split if keys > m-1
Add mid-point to parent.
Create parent if root.



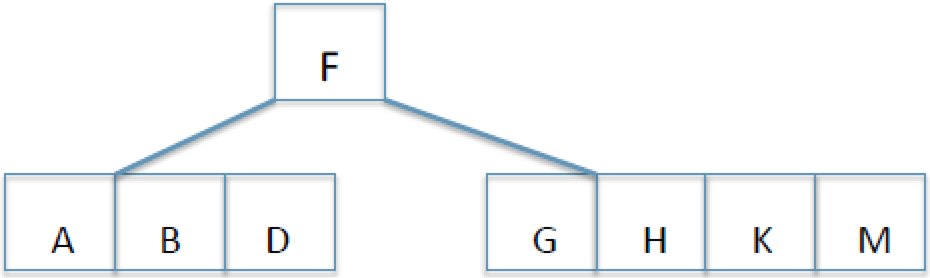
B-Tree Creation

A G F B K D H M J E S I R X C L N T U P



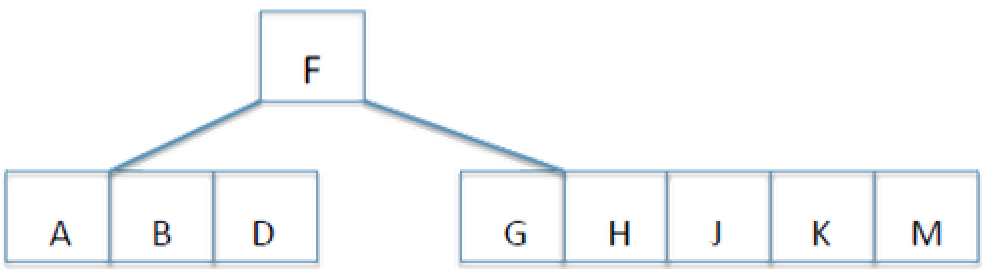
Split if keys > m-1
Add mid-point to parent.
Create parent if root.

A G F B K D H M J E S I R X C L N T U P

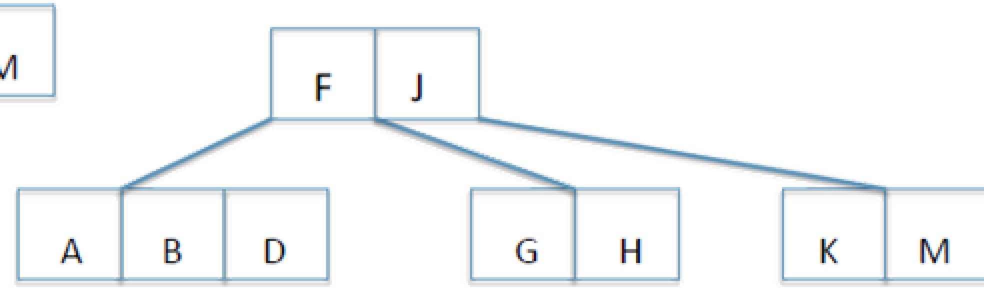


B-Tree Creation

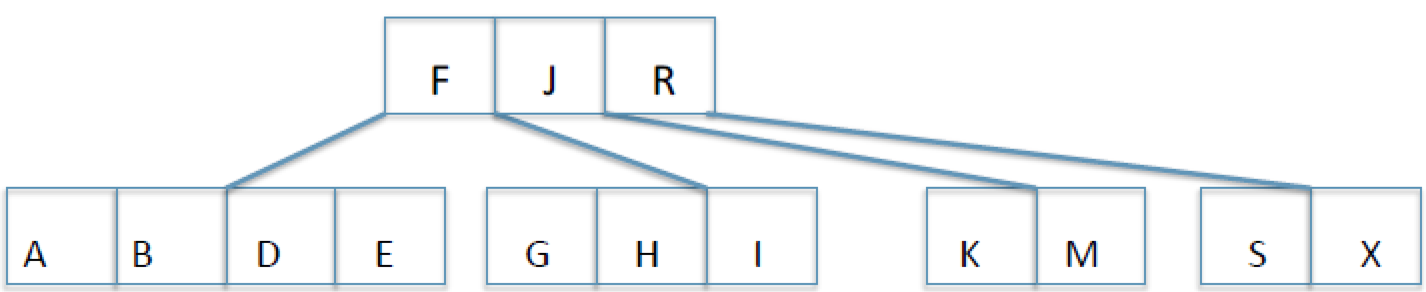
A G F B K D H M J E S I R X C L N T U P



Split if keys > m-1
Add mid-point key to parent.

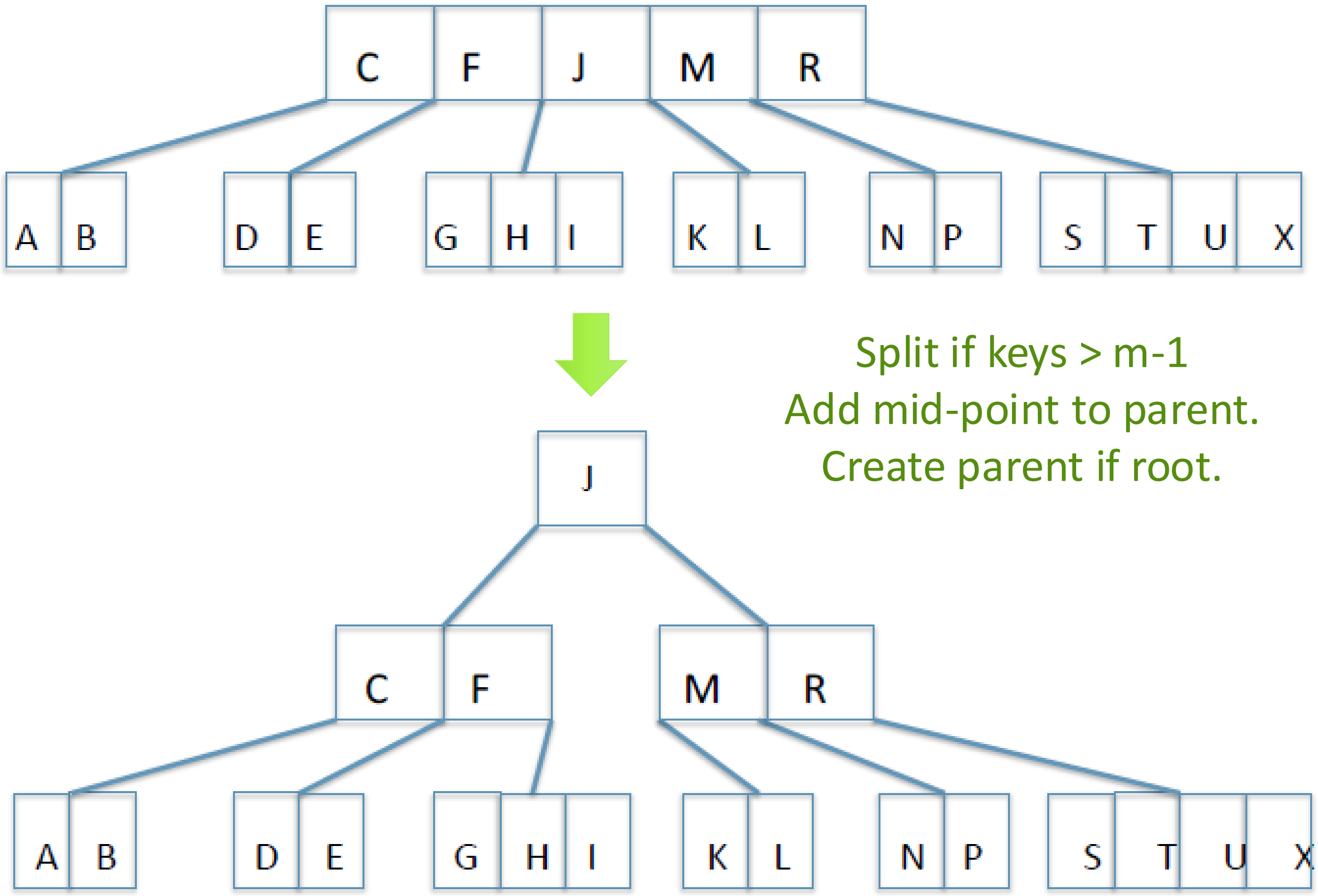


A G F B K D H M J E S I R X C L N T U P



B-Tree Creation

A G F B K D H M J E S I R X C L N T U P



Split if keys > m-1
Add mid-point to parent.
Create parent if root.

Efficiency of B-trees

- If a B-tree has order **m**, then each node (apart from the root) has at least **m/2 children**
- So the depth of the tree is at most **log $_{m/2}$ (size)+1**
  - These many blocks have to be loaded from disk
- In the worst case, we have to make **m-1 comparisons** in each node
  - Linear search, but (m-1) is a *constant factor* and *in-memory scan cost* is lower

Tasks

- Self study (Sahni Textbook)
  - Chapter 10.5, Hashing from textbook
  - Chapter 11.0-11.6, Trees & Binary Trees from textbook
  - B Trees (online sources)