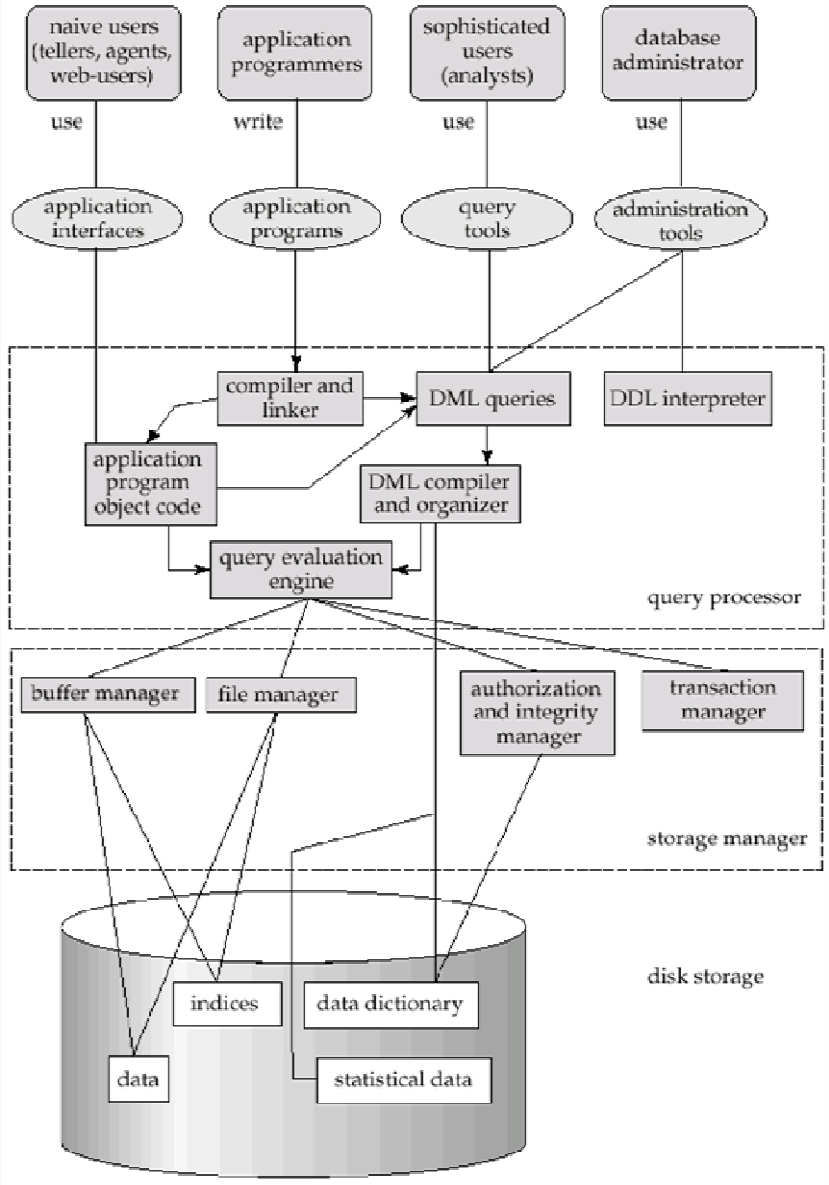
File Storage and Indexes

CS 361: Database Systems

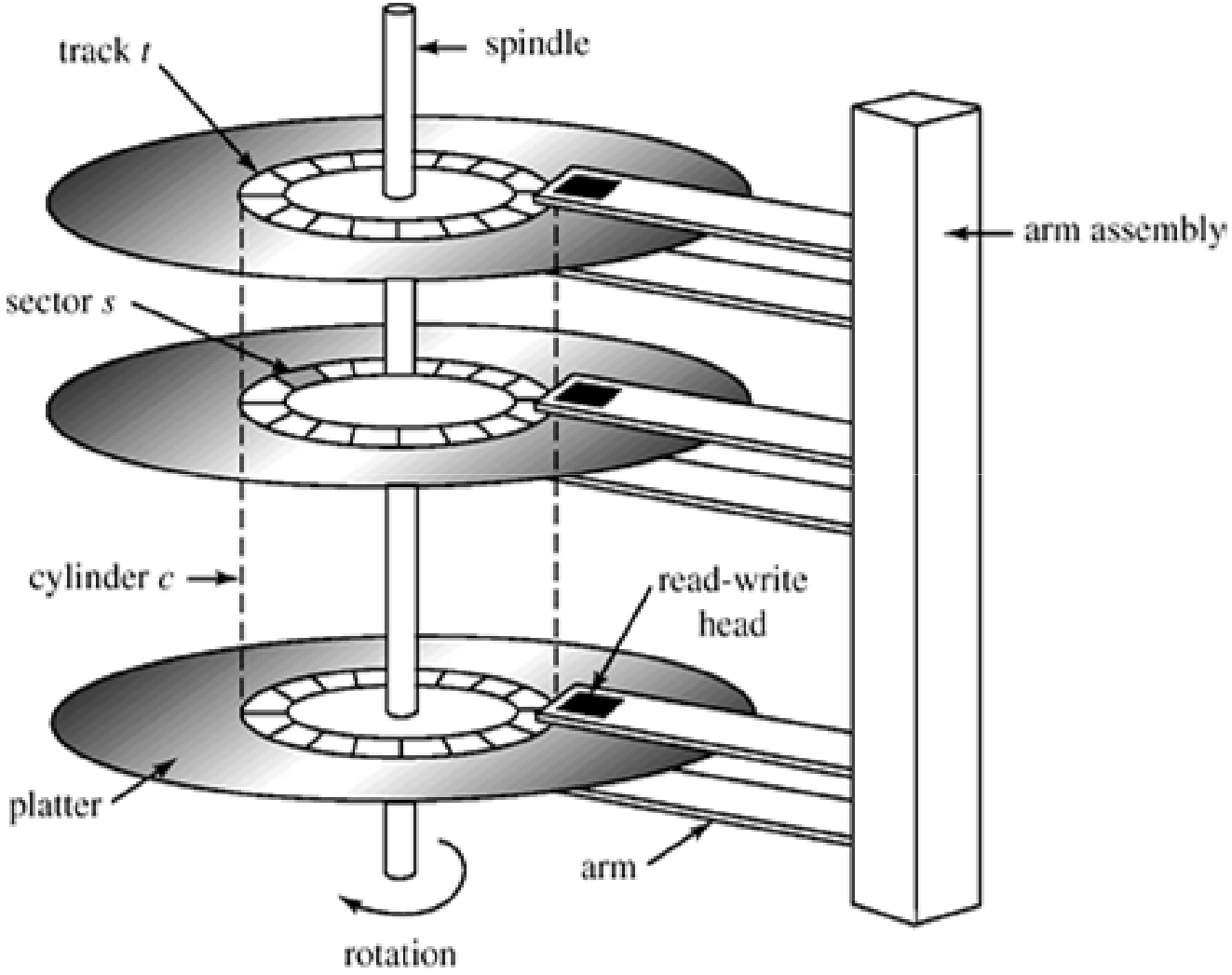
Recall DB Design steps

Real-world → E/R model → Relational schema →
Better relational schema → Relational DBMS →
Applications for Users

- Recall our original representation of DBMS and the people involved.
- Who is the Database "person" missing in this picture?



Hard Drive Architecture

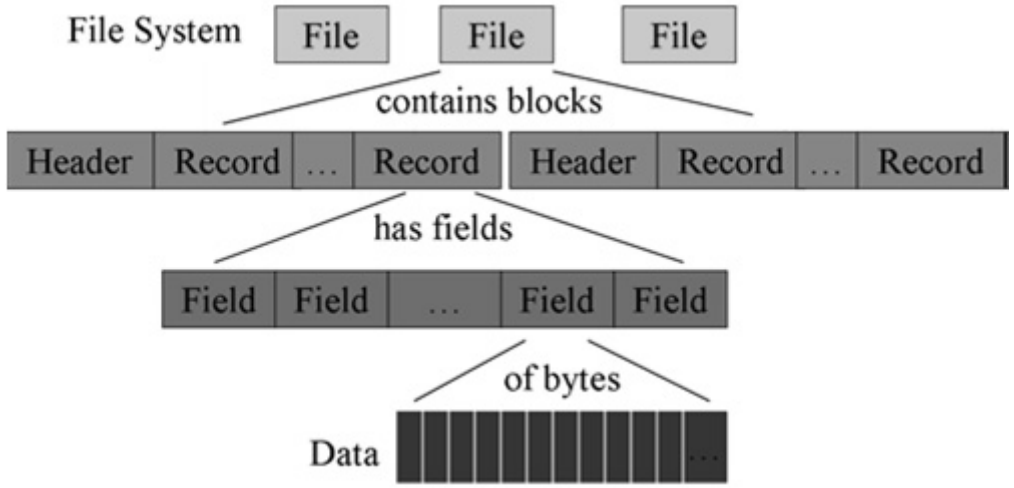


Disk Storage Devices

- A track is divided into smaller **blocks** or **sectors**
  - The block size B is fixed for each system: 512-4096 bytes
  - Whole blocks are transferred between disk and main memory for processing.
- A **read-write head** moves to the track that contains the block to be transferred.
  - Disk rotation moves the block under the read-write head for reading or writing.
- A physical disk block (hardware) address consists of:
  - a cylinder number (imaginary collection of tracks of same radius from all recorded surfaces)
  - the track number or surface number (within the cylinder)
  - and block number (within track).
- Reading or writing a disk block is time consuming because of the *seek time* and *rotational latency*.
  - Double buffering can be used to speed up the transfer of contiguous disk blocks.
- Disk access time is the time required to transfer data.

  = seek time + rotational latency + *block transfer time*
  - Minimize this!

- A database is made up of one or more files.
  - Usually, each table is a file
  - Each file contains one or more blocks.
  - Each block has a header and contains one or more records.
  - Each record contains one or more fields.
  - Each field is a representation of a data item in a record
- Basically, a file is a *sequence* of records
- A block (a.k.a. *page*) is the unit of data transfer between disk and main memory



Storing Records in Blocks

- Fixed length vs. Variable length
  - optional, repeating groups, variable length fields
- Separation - how do we separate adjacent records?
- Spanning - can a record cross a block boundary?
  - spanned vs. unspanned blocking
- Clustering - can a block store multiple record types?
  - why?
- Allocating file blocks on disk:
  - contiguous, linked, clusters, indexed

To search for a record on disk

- One or more blocks copied into main memory buffers
- Programs search for desired record(s) within buffers using info. from block headers
- If address of block containing record not known
  - linear search through all blocks!
- **Goal of good file organization:** locate block containing desired record with minimum number of block transfers
- File operations grouped into
  - Retrieval operations
  - Update operations

Unordered Files

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records.

Ordered Files

- Also called a **sequential** file.
- File records are kept sorted by the values of an ordering field.
- Insertion is expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered overflow (or transaction) file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its ordering field value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Indexing

- An **index** is a data structure that allows for fast lookup of records in a file.
  - may also allow records to be retrieved in sorted order.
  - important for file systems and databases as many queries require only a small set of the data in a file
- Terminology:
  - The **data file** is the file that actually contains the records.
  - The **index file** is the file that stores the index information.
  - The **search key** is the set of attributes stored by the index to find the records in the data file.
    - does not have to be unique
  - An **index entry** is one index record that contains a search key value and a pointer to the location of the record with that value.

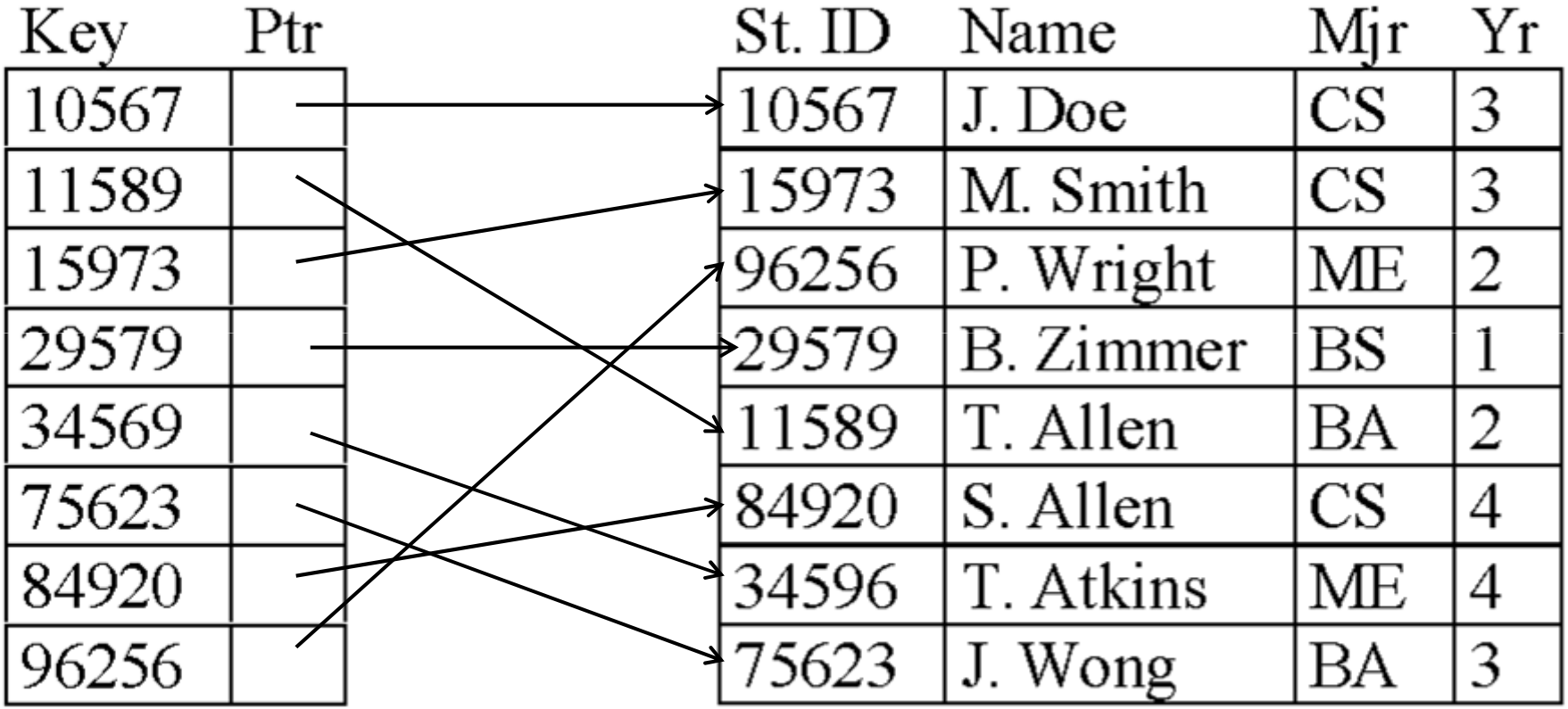
Evaluating Index Methods

- Index methods can be evaluated for functionality, efficiency, and performance.
- The **functionality** of an index can be measured by the types of queries it supports. Two query types are common:
  - exact match on search key
  - query on a range of search key values
- The **performance** of an index can be measured by the time required to execute queries and update the index.
  - Access time, update, insert, delete time
- The **efficiency** of an index is measured by the amount of space required to maintain the index structure.

Types of Indexes

- Indexes on ordered versus unordered files
  - An ordered file is sorted on the search key. Unordered file is not.
- Dense versus non-dense indexes
  - A dense index has an index entry for every record in the data file.
  - A non-dense index has index entries for only some of the data file records (often indexes by blocks).
- Primary (clustering) indexes versus secondary indexes
  - A primary index sorts the data file by its search key. The search key does not have to be the same as the primary key.
  - A secondary index does not determine the organization of the data file.
- Single-level versus multi-level indexes
  - A single-level index has only one index level.
  - A multi-level index has several levels of indexes on the same file.

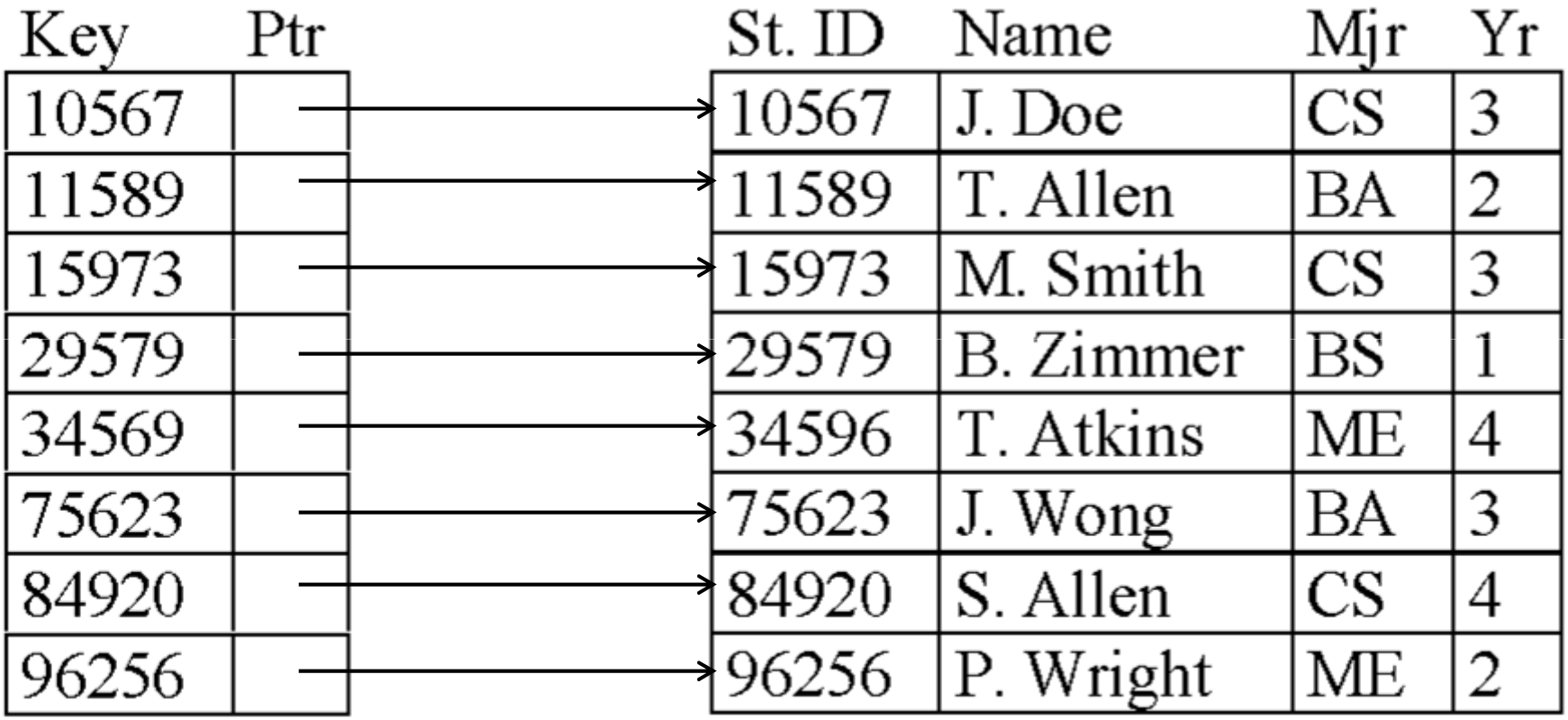
Dense single-level (secondary) index on unordered file



Key	Ptr
10567	
11589	
15973	
29579	
34569	
75623	
84920	
96256	

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
15973	M. Smith	CS	3
96256	P. Wright	ME	2
29579	B. Zimmer	BS	1
11589	T. Allen	BA	2
84920	S. Allen	CS	4
34596	T. Atkins	ME	4
75623	J. Wong	BA	3

Dense single-level primary index on ordered file



Key	Ptr
10567	
11589	
15973	
29579	
34569	
75623	
84920	
96256	

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

Index on Unordered/Ordered Files

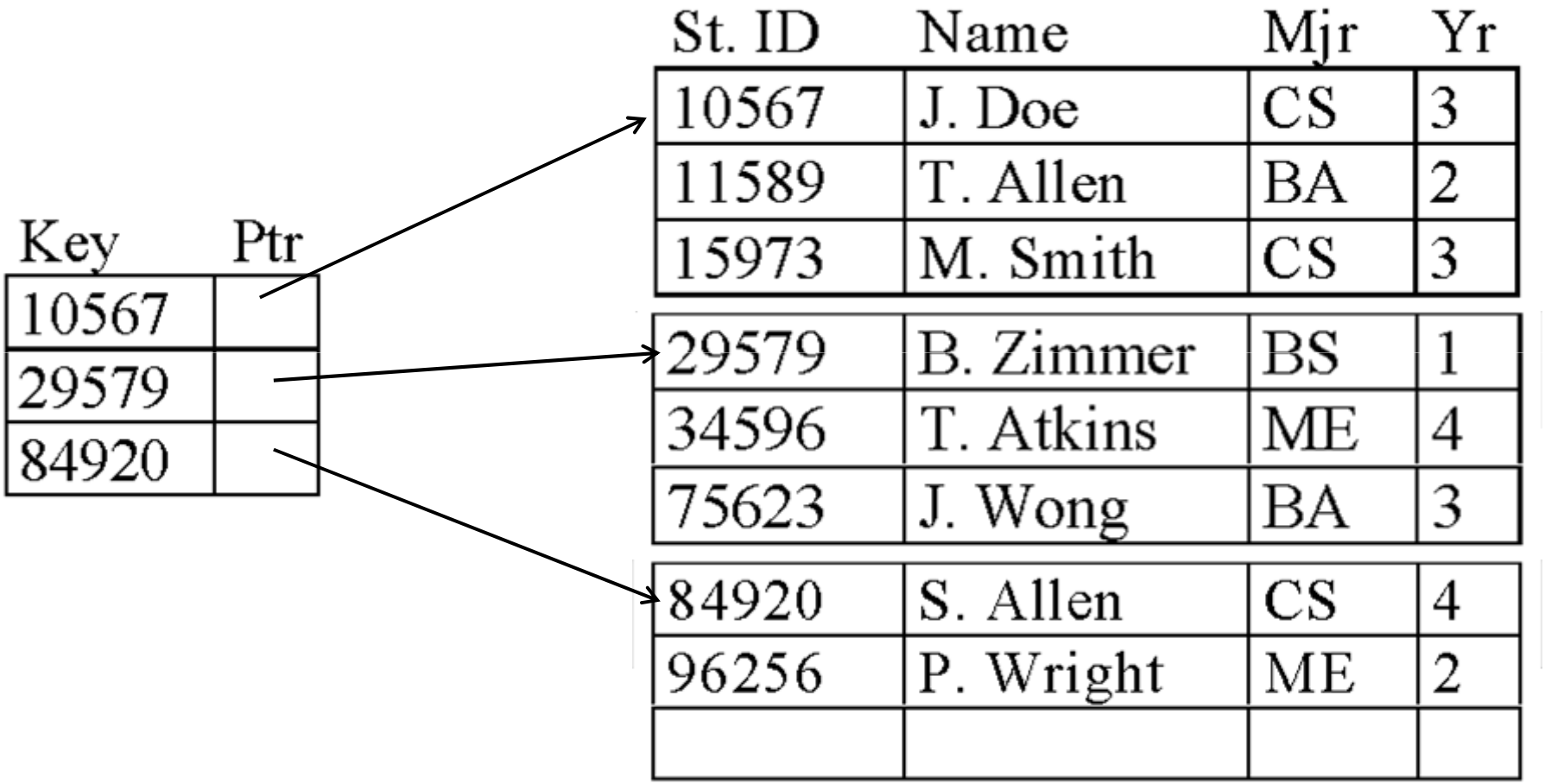
- An index on an unordered file allows us to access the file in sorted order without maintaining the records in sorted order.
  - Insertion/deletion are more efficient for unordered files.
  - Append record at end of file or move record from end for delete.
  - Must only update index after data file is updated.
  - Searching for a search key can be done using binary search on the index.

- What advantage is there for a primary index on an ordered file?
  - Less efficient to maintain an ordered file PLUS we must now also maintain an ordered index!

Sparse Index on Ordered Files

- A sparse index only contains a subset of the search keys that are in the data file.
  - better choice for an ordered file is a sparse index since we can take advantage of the fact that the data file is already sorted.
  - The index will be smaller as not all keys are stored.
  - Fewer index entries than records in the file.
  - Binary search over index can be faster as fewer index blocks to read than unordered file approach.
  - Generally store one search key per block of the data file.

Sparse Index on Ordered File



Sparse Index versus Dense Index

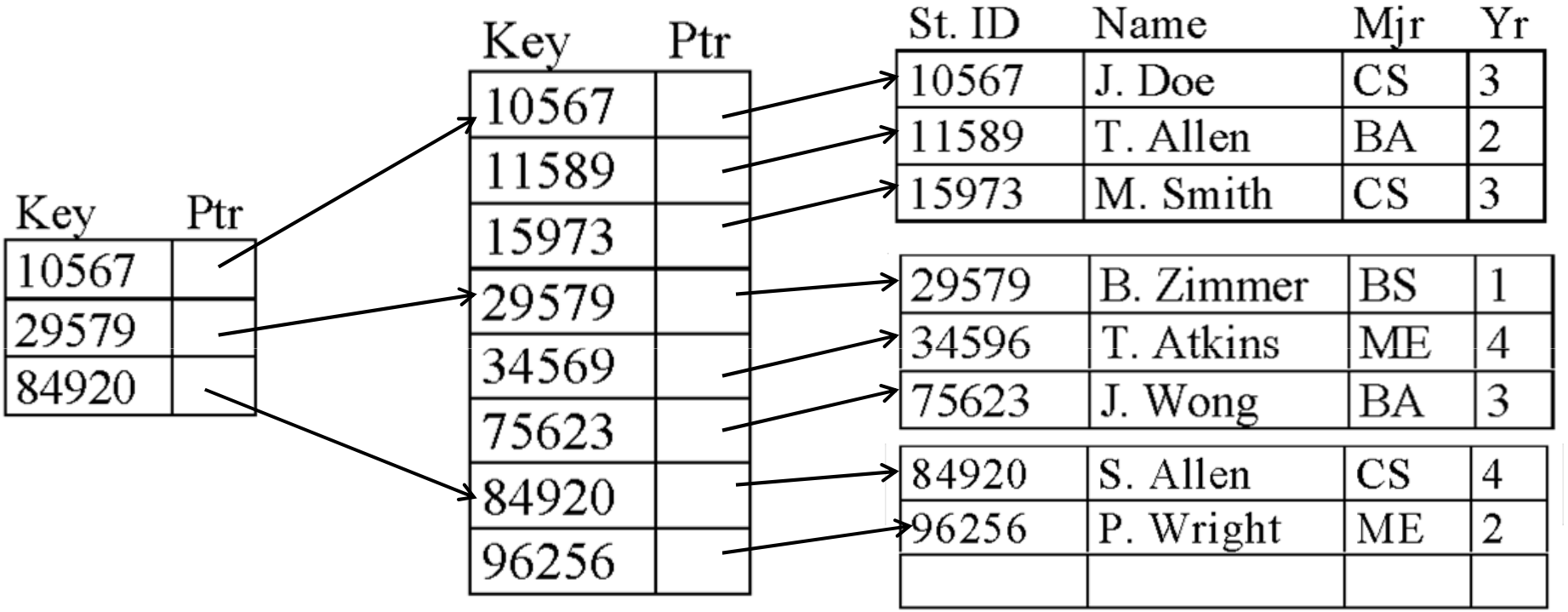
- A sparse index is much more space efficient than a dense index because it only stores one search key per block.
  - If a block can store 10 data records, then a sparse index will be 10 times smaller than a dense index!
  - This allows more (or all) of the index to be stored in main memory and reduces disk accesses if the index is on disk.

- A dense index has an advantage over a sparse index because
  - it can answer queries like "does search key K exist?" without accessing the data file (by using only the index).
  - Finding a record using a dense index is easier as the index entry points directly to the record.
  - For a sparse index, the block that may contain the data value must be loaded into memory and then searched for the correct key.

Multi-level Index

- A multi-level index has more than one index level for the same data file.
  - Each level of the multi-level index is smaller, so that it can be processed more efficiently.
  - The first level of a multi-level index may be either sparse or dense, but all higher levels must be sparse.
- Having multiple levels of index increases the level of indirection, but is often quicker because the upper levels of the index may be stored entirely in memory.
  - However, index maintenance time increases with each level.

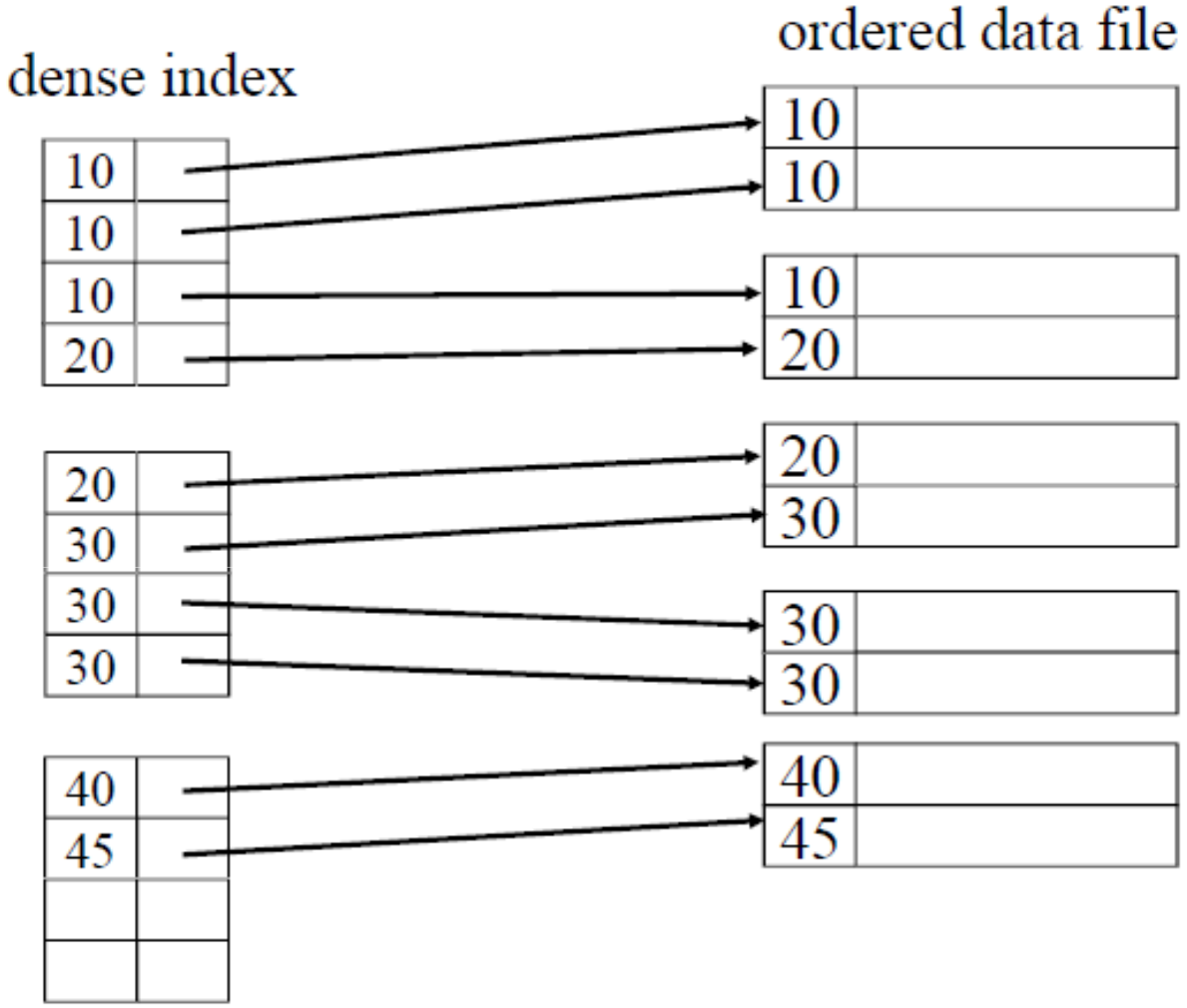
Multi-level index on ordered file

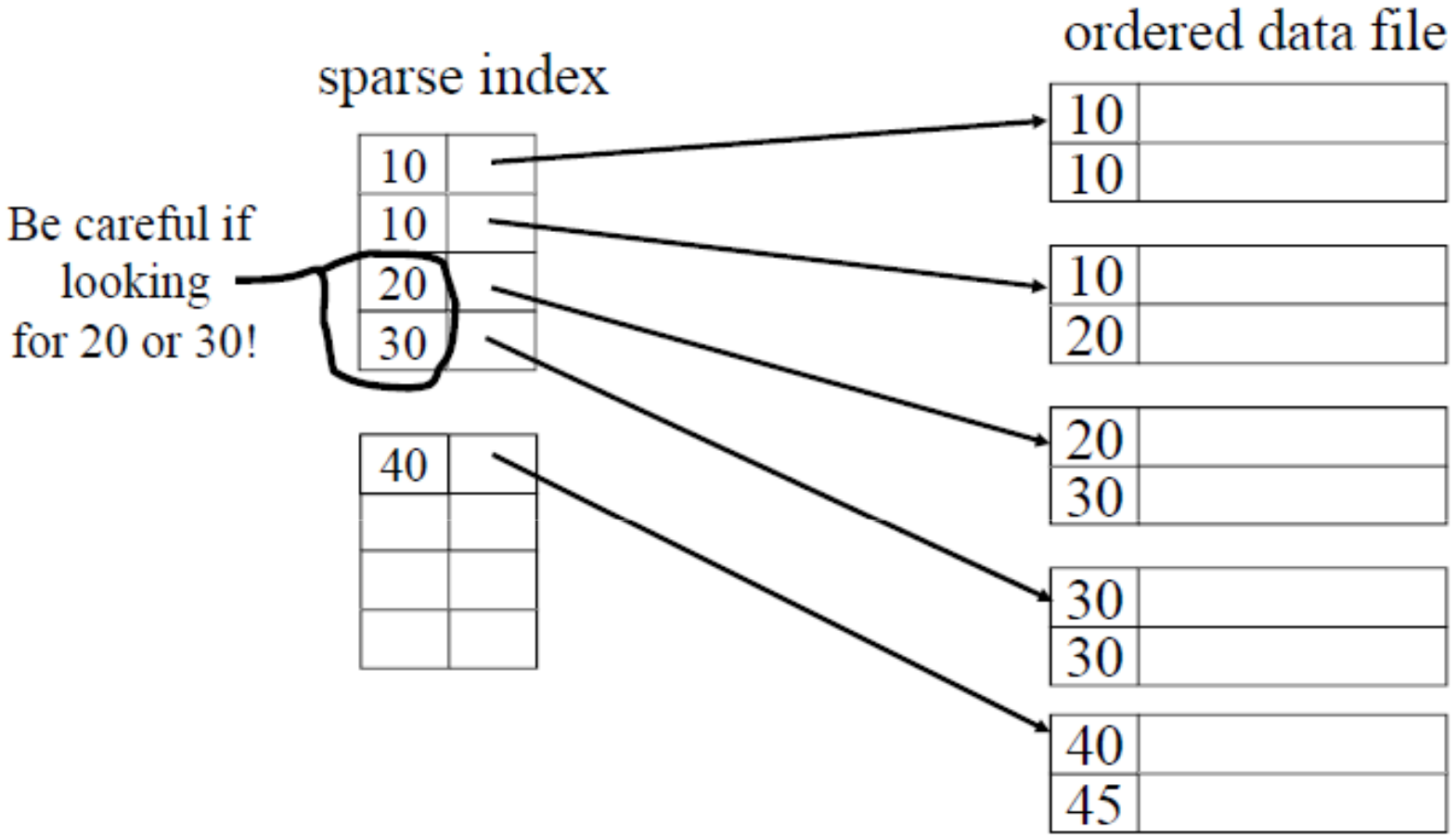


Indexes with Duplicate Search Keys

- What happens if the search key for our index is not unique?
  - The data file contains many records with the same search key.
  - This is possible because we may index a field that is not a primary key of the relation.
- Both sparse and dense indexes still apply:
  - Dense index with entry for every record
  - Sparse index containing one entry per block

- But search strategy changes if we have many records with same search key.

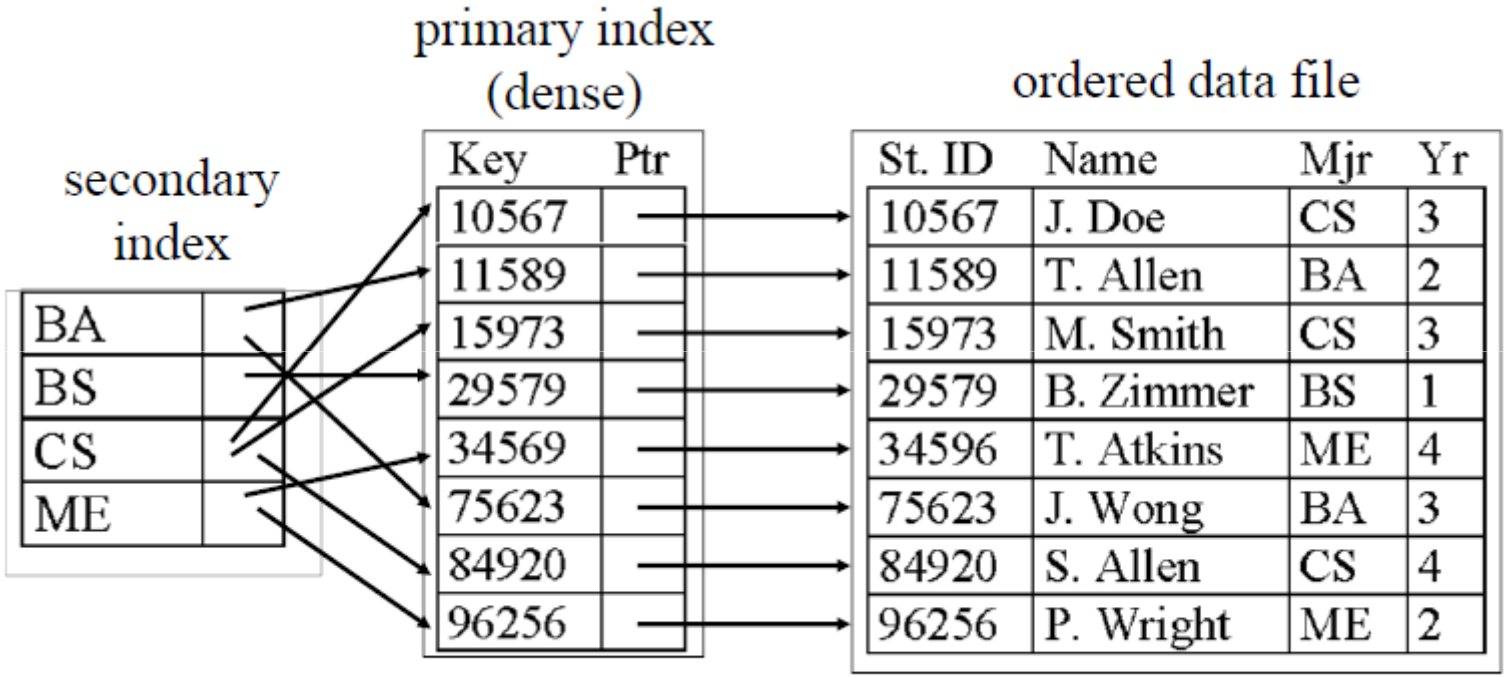

dense index
ordered data file
10
10
10
20
20
30
30
30
40
45
10
10
10
20
20
30
30
30
40
45


ordered data file
sparse index
Be careful if looking for 20 or 30!
10
10
20
30
40
10
10
10
20
20
30
30
30
40
45

Secondary Indexes

- A secondary index is an index whose search key does not determine the ordering of the data file.
  - A data file can have only one primary index but many secondary indexes.
  - Secondary index entries often refer to the primary index instead of the data records directly.

- Advantage - simpler maintenance of secondary index.
  - Secondary index changes only when primary index changes not when the data file changes.
- Disadvantage - less efficient due to indirection.
  - Multiple levels of indirection as must use secondary index, then go to primary index, then access record in data file.


primary index
(dense)
ordered data file
secondary
index
Key
Ptr
St. ID
Name
Mjr
Yr
BA
BS
CS
ME
10567
11589
15973
29579
34569
75623
84920
96256
10567
J. Doe
CS
3
11589
T. Allen
BA
2
15973
M. Smith
CS
3
29579
B. Zimmer
BS
1
34596
T. Atkins
ME
4
75623
J. Wong
BA
3
84920
S. Allen
CS
4
96256
P. Wright
ME
2

Index Maintenance

- As the data file changes, the index must be updated as well.
- The two operations are **insert** and **delete**.
- Maintenance of an index is similar to maintenance of an ordered file. The only difference is the index file is smaller.
- Techniques for managing the data file include:

1. Using overflow blocks
   - Create/delete overflow block for data file
   - No effect on both sparse/dense index (overflow block not indexed).

2. Re-organizing blocks by shifting records
   - Create/delete new sequential block for data file
   - Dense index unaffected, sparse index needs new index entry for block.

3. Adding or deleting new blocks in the file
   - Insert/Delete/Move record
   - Dense index must either insert/delete/update entry.
   - Sparse index may only have to update entry if the smallest key value in the block is changed by the operation.

Hash Indexes

- The goal of hash indexes is to make all operations require only 1 block access.
- Hash-based indexes are best for equality selections
  - Cannot support range searches
- *Hashing* is a technique for mapping key values to locations.
  - requires the definition of a hash function `f(x)`, that takes the key value x and computes `y=f(x)` which is the location of where the key should be stored.
- A *collision* occurs when we attempt to store two different keys in the same location.

  `f(x1) = y and f(x2) = y for two keys x1 <> x2`

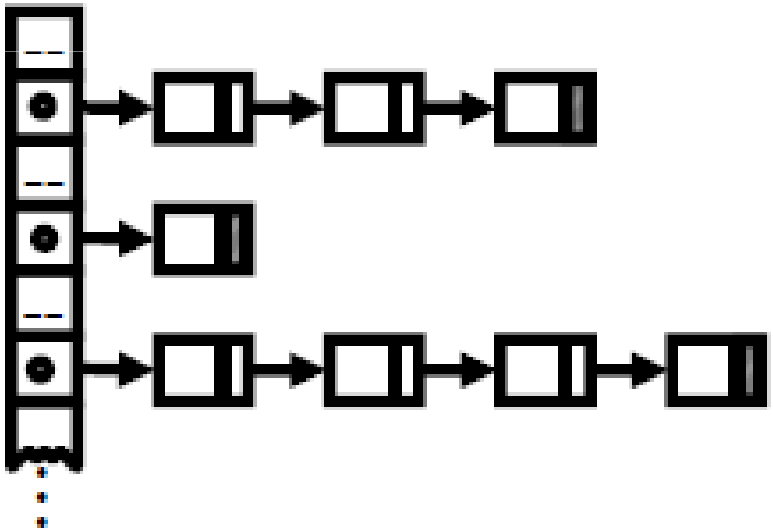
Handling Collisions

- One simple way is to make the hash table really large to minimize the probability of collisions.
  - Not practical in general. However, we do want to make our hash table moderately larger than the # of keys.

- **Open addressing with linear probing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
  - Example
- Insert using linear probing creates the potential that a key may be inserted many locations away from its original hash location.
- What happens if we delete an element that is in its exact hashed location and had a collision with another element?
  - What happens if we search for this other element?

Handling Collisions

- **Separate Chaining**:
  - Separate chaining resolves collisions by having each array location point to a linked list of elements.
  - Algorithms for operations such as insert, delete, and search straightforward.
  - As with open addressing, separate chaining has the potential to degenerate into a linear algorithm if the hash function does not distribute the keys evenly in the array.



Handling Collisions

- **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Hash Limitations and Analysis

- Hashing gives very good performance when the hash function is good and the number of conflicts is low.
- If the # of conflicts is high, then the performance of hashing rapidly degrades. The worst case is O(n).
- Collisions can be reduced by minimizing load factor
  - # of occupied locations/size of hash table.
- On average, inserts, searches, and deletes are O(1)

- **Limitations**:
  - Ordered traversals are difficult without an additional structure and a sort (hashing randomizes locations of records)
  - Partial key searches are difficult as the hash function must use the entire key.

Hash Limitations and Analysis

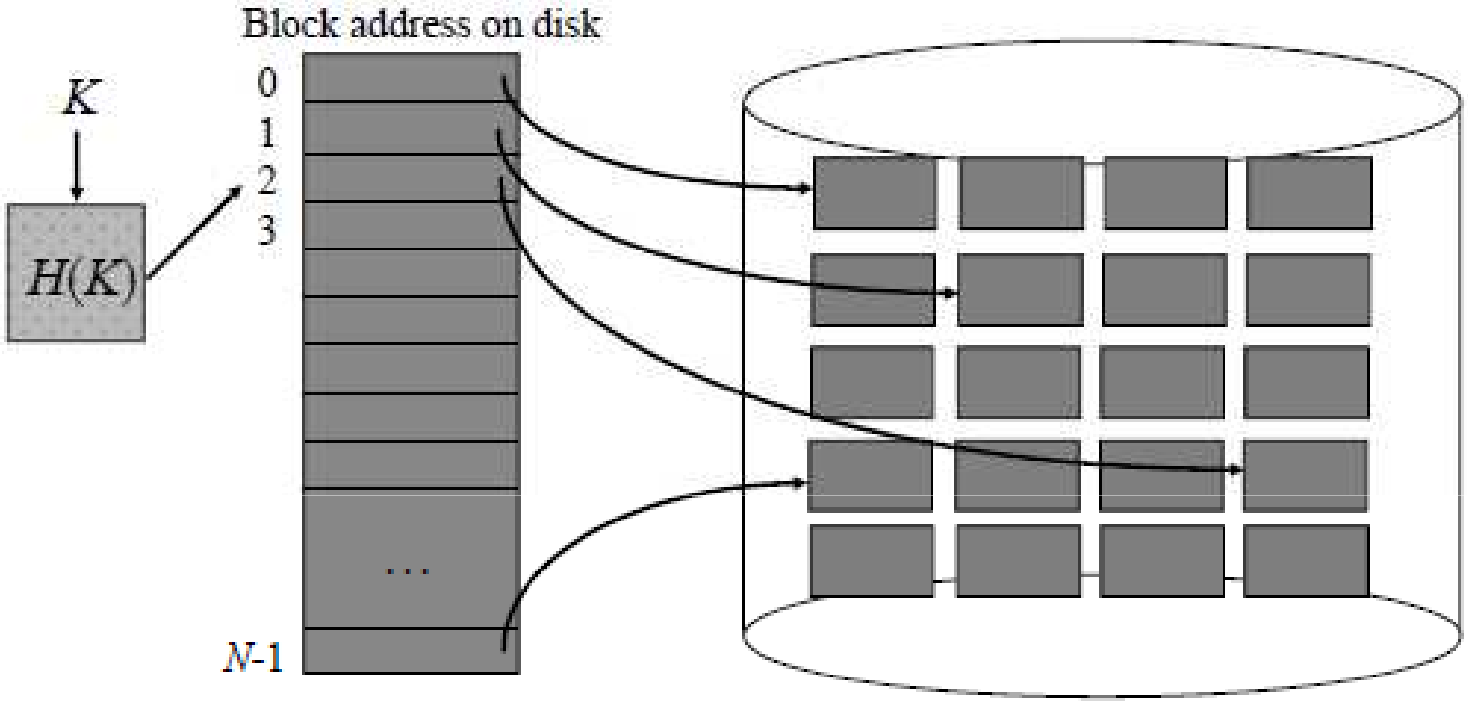
- The *hash field space* is the set of all possible hash field values for records.
  - It is the set of all possible key values that we may use.
- The *hash address space* is the set of all record slots (or storage locations).
  - Size of array in memory or physical locations in a file.

Tradeoff:

- The larger the address space relative to the hash field space, the easier it is to avoid collisions
- BUT the larger the address space relative to the number of records stored, the worse the storage utilization.

External Hashing

- External hashing algorithms allocate records with keys to blocks on disk rather than locations in a memory array.
- Hash file has relative bucket numbers 0 through N-1.
- Map *logical* bucket numbers to *physical* disk block addresses.
- Disk blocks are buckets that hold several data records each.
- The record with hash key value K is stored in bucket i, where i=h(K), and h is the **hashing function**.


Block address on disk
K
H(K)
0
1
2
3
...
N-1

- In **static hashing**, the hash function maps keys to a fixed set of bucket addresses. However, databases grow with time.
  - If initial number of buckets is too small?
  - If file size is made larger to accommodate future needs?

- One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- So we must determine optimal utilization of hash table.
  - Try to keep utilization between 50% and 80%. Hard when data changes.

- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Linear Hashing

- Linear hashing allows a hash file to expand and shrink dynamically.
- A linear hash table starts with 2^d buckets where d is the # of bits used from the hash value to determine bucket membership.
  - Take the **last d** bits of H where d is the current # of bits used.
- The growth of the hash table can either be triggered:
  - Every time there is a bucket overflow.
  - When the load factor of the hash table reaches a given point.

- We will use the load factor method.
  - Since bucket overflows may not always trigger hash table growth, overflow blocks are used.

Load Factor

- The load factor **lf** of the hash table is the number of records stored divided by the number of possible storage locations.
  - The initial number of blocks n is a power of 2.
  - As the table grows, it may not always be a power of 2.
  - The number of storage locations

    s = #blocks X #records/block.
- The initial number of records in the table r is 0 and is increased as records are added.
- Load factor = r / s = r / n * #records/block
- We will expand the hash table when the load factor > 85%.

Linear Hashing Insertion Algorithm

- Insert a record with key **K** by first computing its hash value **H**.
- Take the **last d** bits of **H** where **d** is the **current # of bits** used.
- Find the bucket **m** where **K** would belong using the **d** bits.
- If **m < n**, then bucket exists. Go to that bucket.
  - If the bucket has space, then insert **K**. Otherwise, use an overflow block.
- If **m >= n**, then put **K** in bucket **m - 2^{d-1}**.
- After each insert, check to see if the load factor **lf < threshold**.
- If **lf >= threshold** perform a split:
  - Add new bucket **n**. (Adding bucket **n** may increase the directory size **d**.)
  - Divide the records between the new bucket **n=1b_2…b_d** and bucket **0b_2…b_d**.
  - Note that the bucket split may not be the bucket where the record was added! Update **n** and **d** to reflect the new bucket.

Creating Indexes in SQL

- There are two general ways of creating an index:
- (1) By specifying it in your CREATE TABLE statement:

```
CREATE TABLE test (
  a int,
  b int,
  c varchar(10)
  PRIMARY KEY (a),
  UNIQUE (b),
  INDEX (c)
);
```

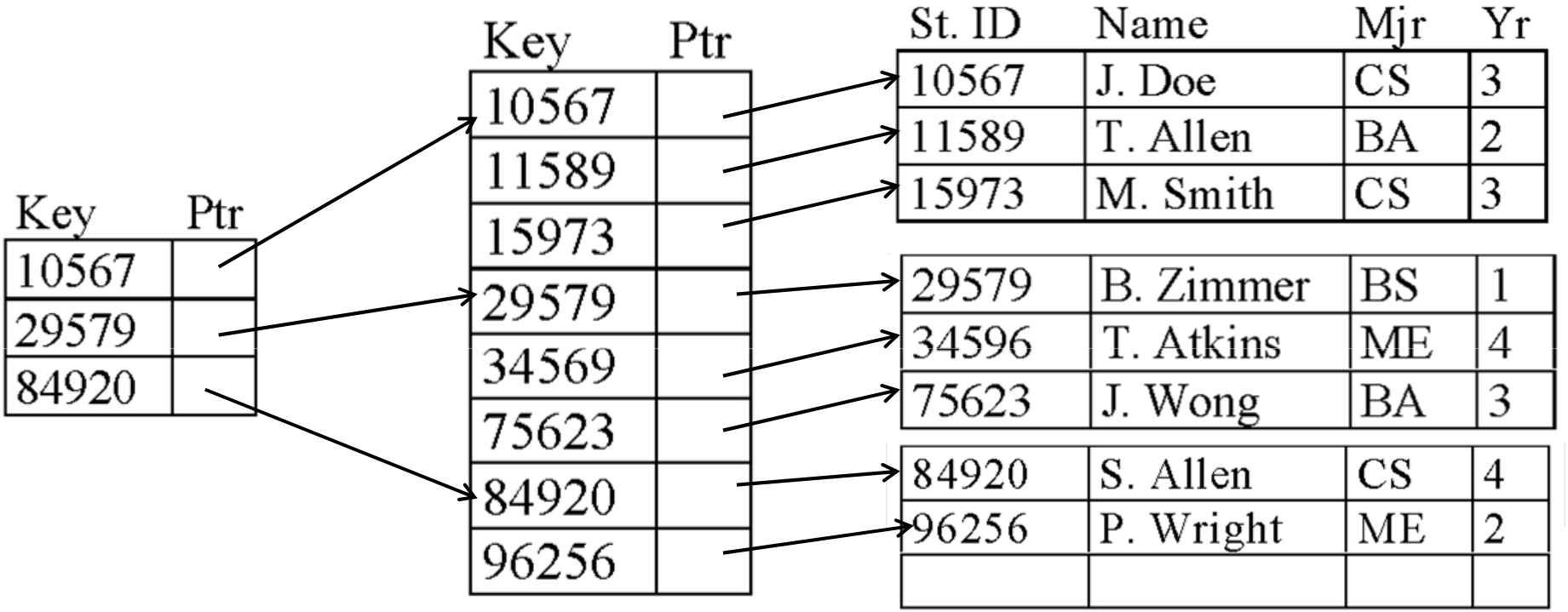
- (2) Using a `CREATE INDEX` command after a table is created:

```
CREATE INDEX myIdxName ON test (a,b);

CREATE UNIQUE INDEX idxStudent ON
  Student(sid)

CREATE INDEX idxMajorYear ON
  student(Major,Year)
```

Creating Indexes in MySQL

- MySQL supports both ways of creating indexes
- By specifying a primary key, an index is automatically created by MySQL. You do not have to create another one!
- The primary key index (and any other type of index) can have more than one attribute.
- MySQL assigns default names to indexes if you do not provide them.
- MySQL supports B+-tree, Hash, and R-tree indexes but support depends on table type.
- Can index only the first few characters of a CHAR/VARCHAR field by using `col_name(length)` syntax. (smaller index size)
- FULLTEXT indexes allow more powerful natural language searching on text fields (but have a performance penalty).

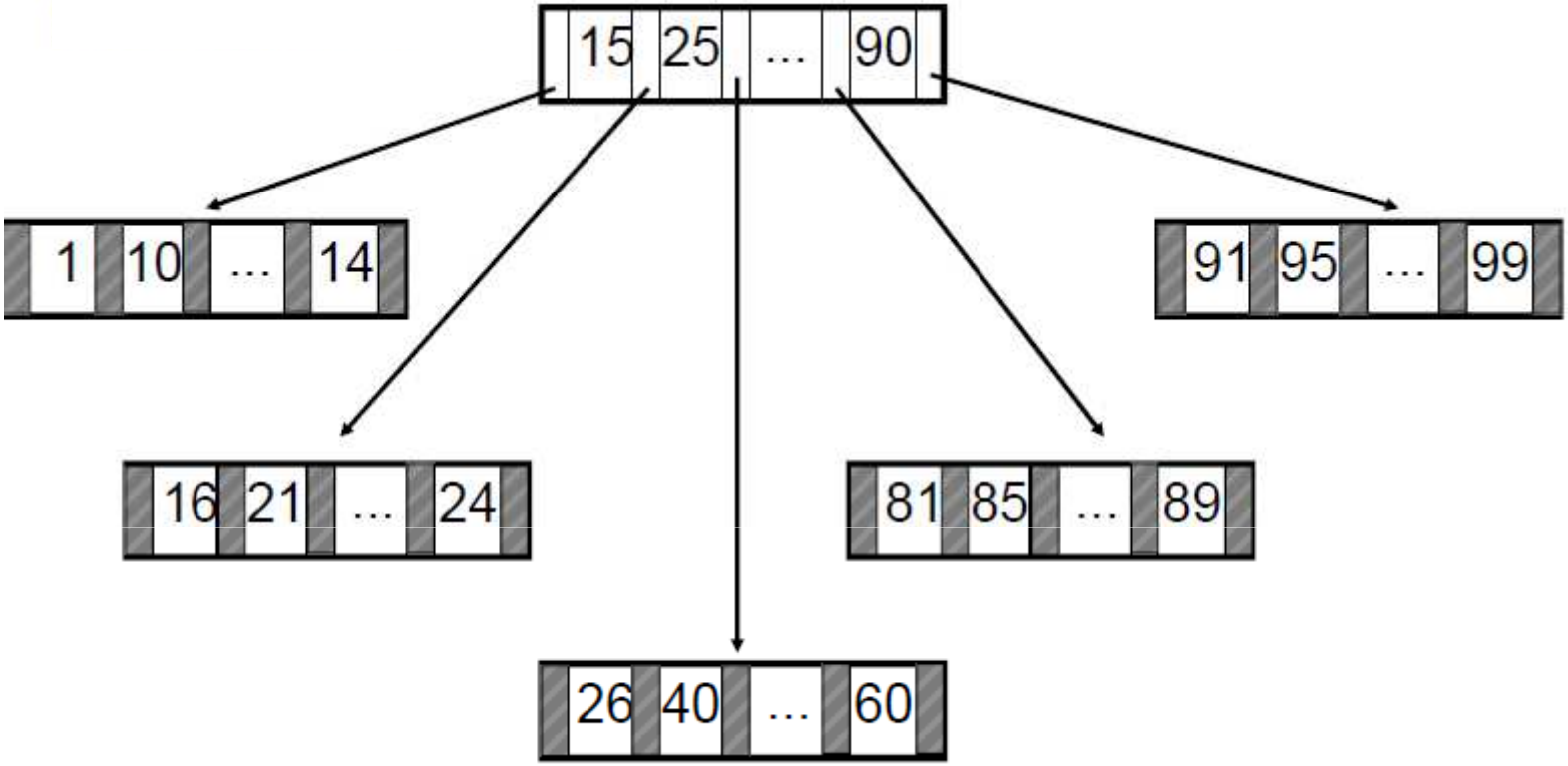
Multi-level index on ordered file



- Recall that multi-level indexes can improve search performance.
- One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.
- B-trees and B+-trees are the most common form of index used in database systems today

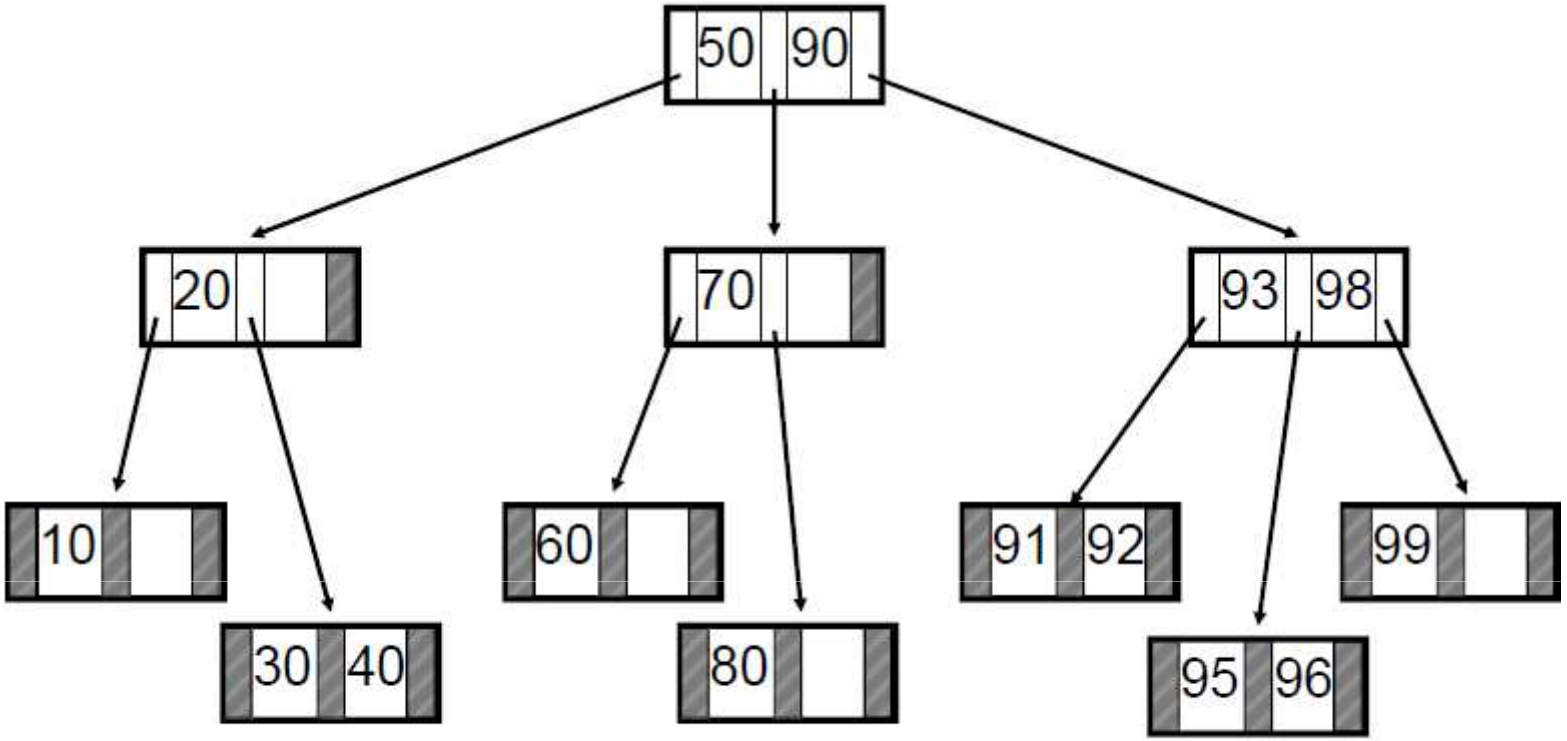
B-tree

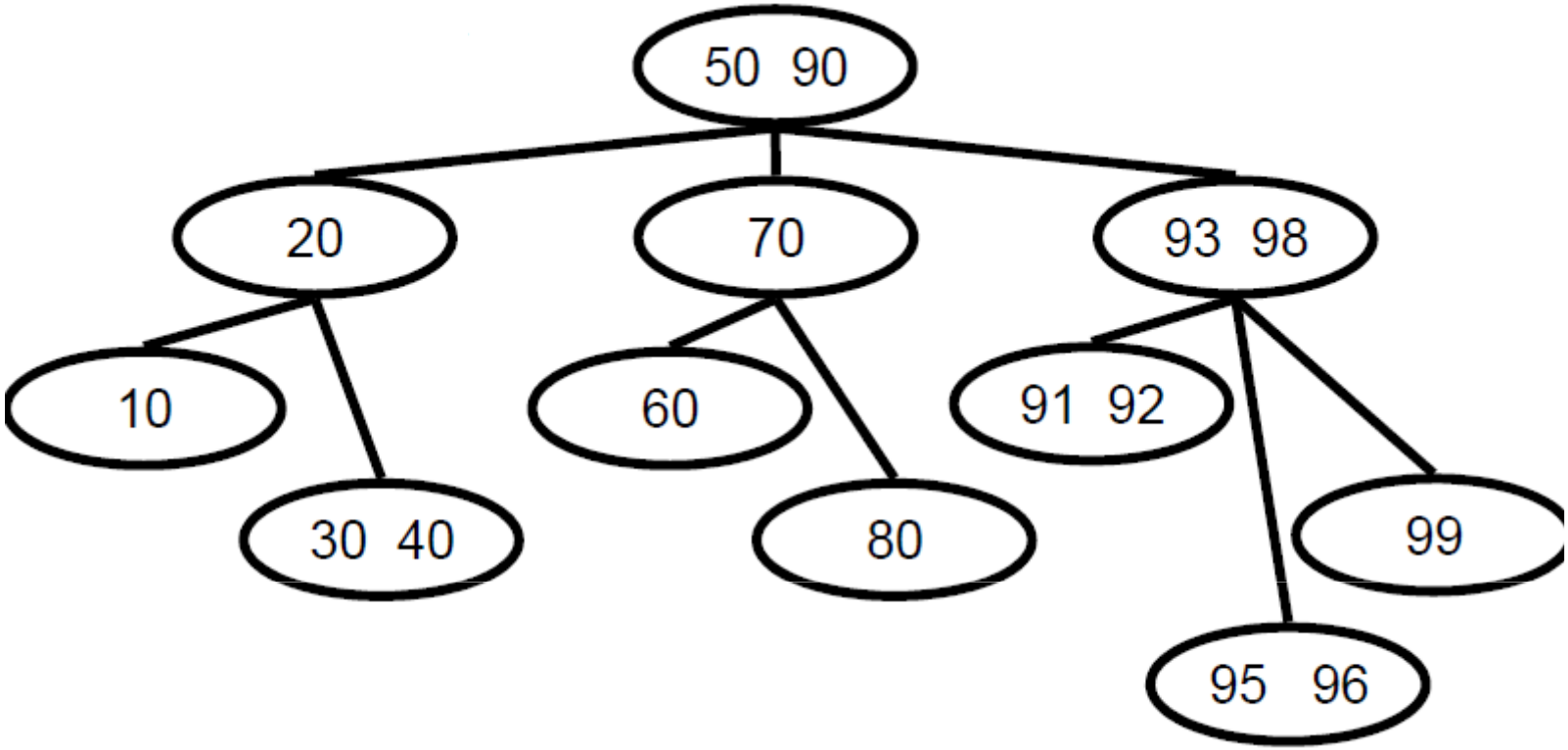
- A B-tree is a search tree where each node has data values >= n and <= 2n, where we chose n for our particular tree.
- Each data value or key in a node is stored in a sorted array.
  - key[0] is the first key, key[1] is the second key,…,key[2n-1] is the $2n^{th}$ key
  - key[0] < key[1] < key[2] < … < key[2n-1]
- There is also an array of pointers to children nodes:
  - child[0], child[1], child[2], …, child[2n]
- Recursive definition: Each subtree pointed to by child[i] is also a B-tree.
- For any key[i]:
  - key[i] > all entries in subtree pointed to by child[i]
  - key[i] <= all entries in subtree pointed to by child[i+1]
- A B-tree is balanced as every leaf has the same depth.
- **# of non-null children = # of keys +1**
- The **order n** is the **minimum # of keys in a node**
  - Other terminology defines this as maximum # of keys in a node


15 25 ... 90
1 10 ... 14
16 21 ... 24
26 40 ... 60
81 85 ... 89
91 95 ... 99

2-3 Trees

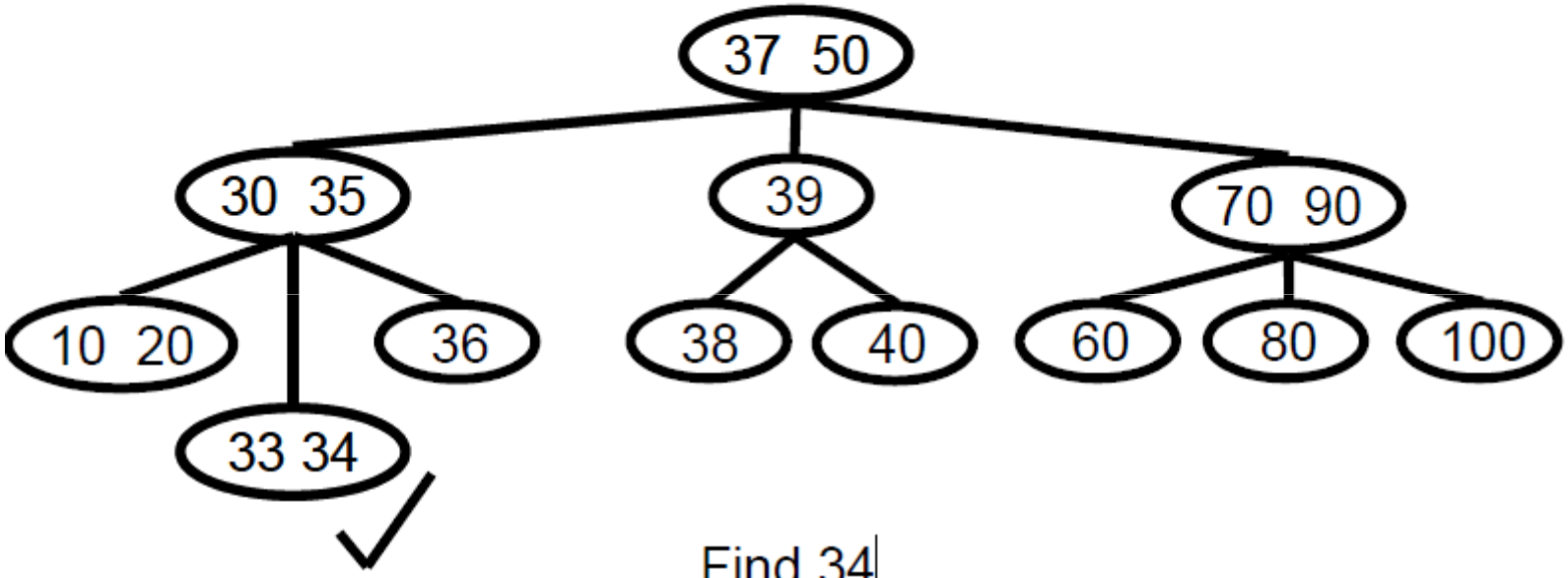
- A 2-3 tree is a B-tree where each node has either 1 or 2 data values and 2 or 3 children pointers.
  - It is a special case of a B-tree.
- A 2-3 tree of height **h** always has at least as many nodes as a full binary tree of height **h**.
  - That is, a 2-3 tree will always have at least $\mathbf{2^h-1}$ nodes.


50 90
20
70
93 98
10
30 40
60
80
91 92
95 96
99

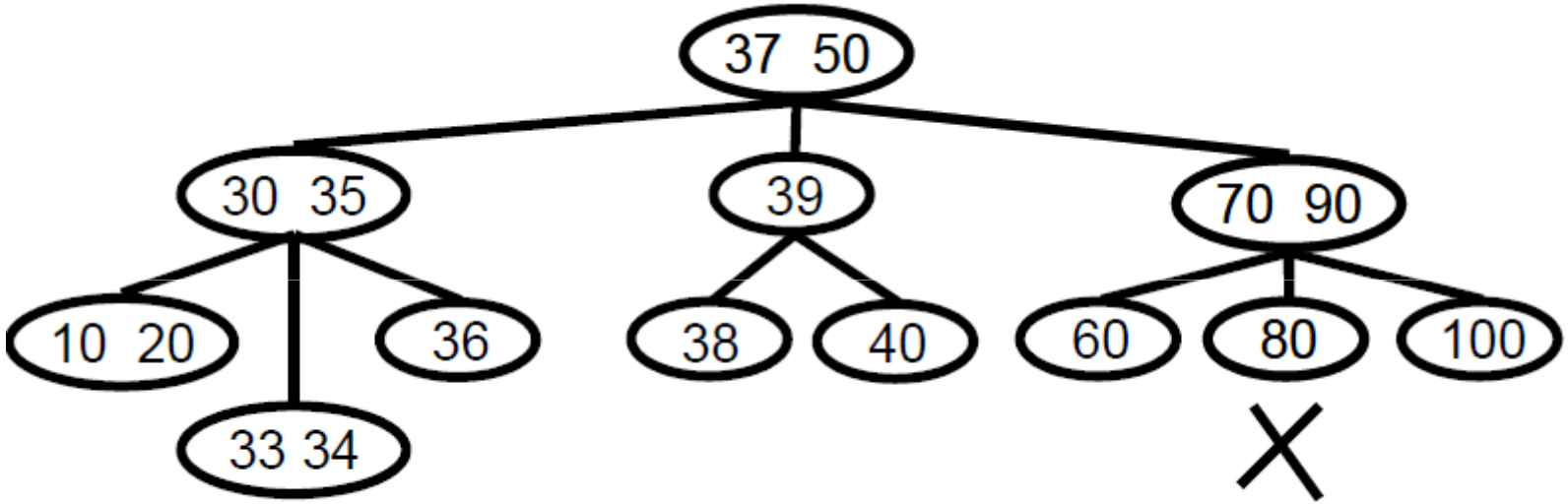

50 90
20
70
93 98
10
60
91 92
30 40
80
99
95 96

Searching a 2-3 Tree

- Start at the root which begins as the `curNode`.
- If `curNode` contains the search key we are done, and have found the search key we were looking for.
- A 2-node contains one key:
  - If search key < key[0], go left (child[0]) otherwise go right (child[1])
- A 3-node contains two key values:
  - If search key < key[0], go left with first child pointer (child[0])
  - else if search key < key[1] go down middle child pointer (child[1])
  - else (search key >= key[1]) go right with last child pointer (child[2])
- If we encounter a `NULL` pointer, then we are done and the search failed.



Find 34



Find 82

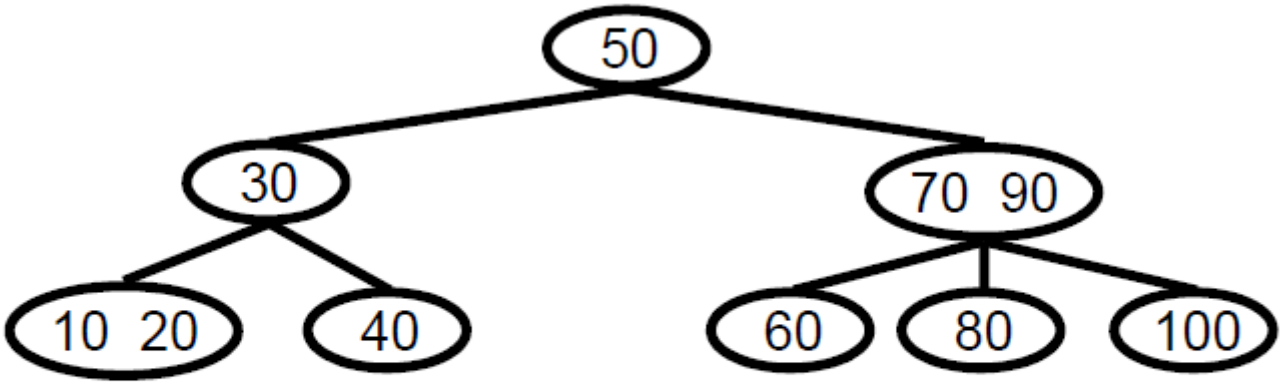
Insertion into a 2-3 Tree

- Find the leaf node where the new key belongs.
- This insertion node will contain either a single key or two keys.
- If the node contains 1 key, insert the new key in the node (in the correct sorted order).
- If the node contains 2 keys:
  - Insert the node in the correct sorted order.
  - The node now contains 3 keys (overflow).
  - Take the middle key and promote it to its parent node. (split node)
  - If the parent node now has more than 3 keys, repeat the procedure by promoting the middle node to its parent node.
- This promotion procedure continues until:
  - Some ancestor has only one node, so overflow does not occur.
  - All ancestors are "full" in which case the current root node is split into two nodes and the tree "grows" by one level.

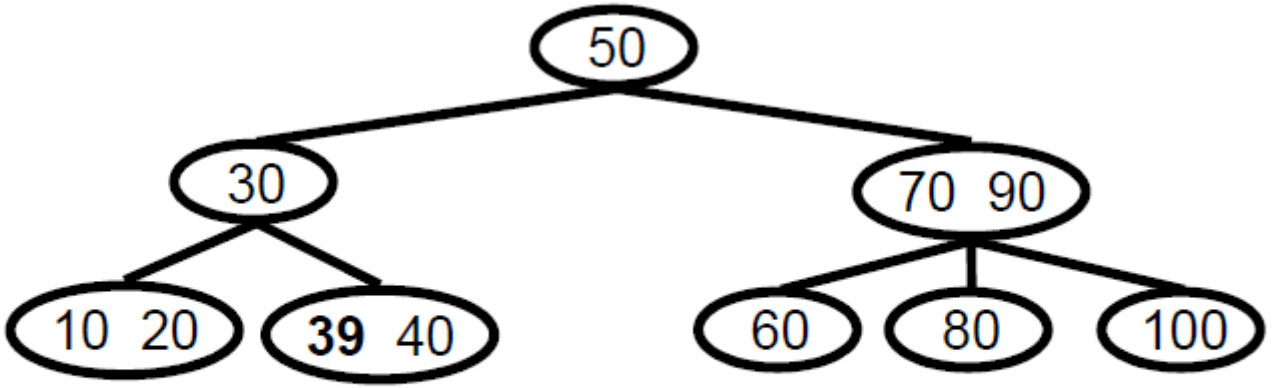
Insertion - Splitting Algorithm

- Given a node with overflow (more than 2 keys in this case), we split the node into two nodes each having a single key.
- The middle value (in this case key[1]) is passed up to the parent of the node.
  - This, of course, requires parent pointers in the 2-3 tree.
- This process continues until we find a node with sufficient room to accommodate the node that is being percolated up.
- If we reach the root and find it has 2 keys, then we split it and create a new root consisting of the "middle" node.

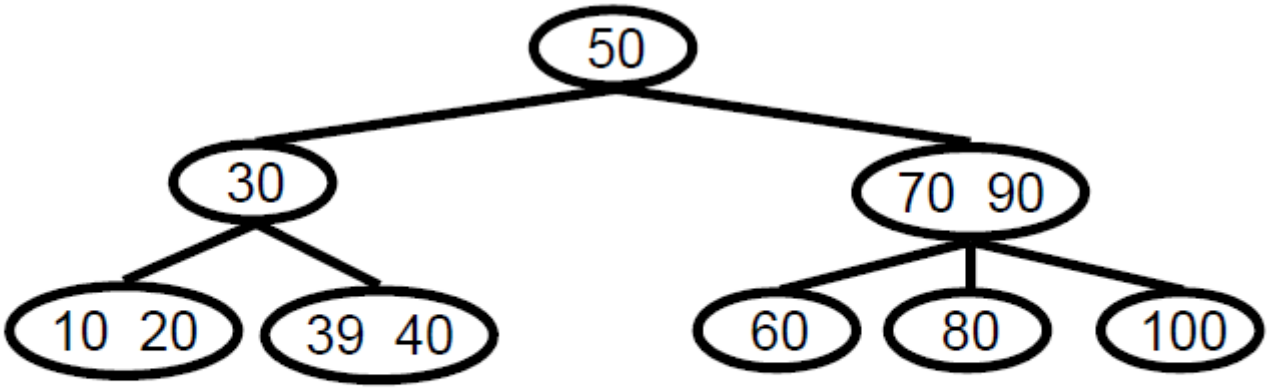
- The splitting process can be done in logarithmic time since we split at most one node per level of the tree and the depth of the tree is logarithmic in the number of nodes in the tree.
  - Thus, 2-3 trees provide an efficient height balanced tree.



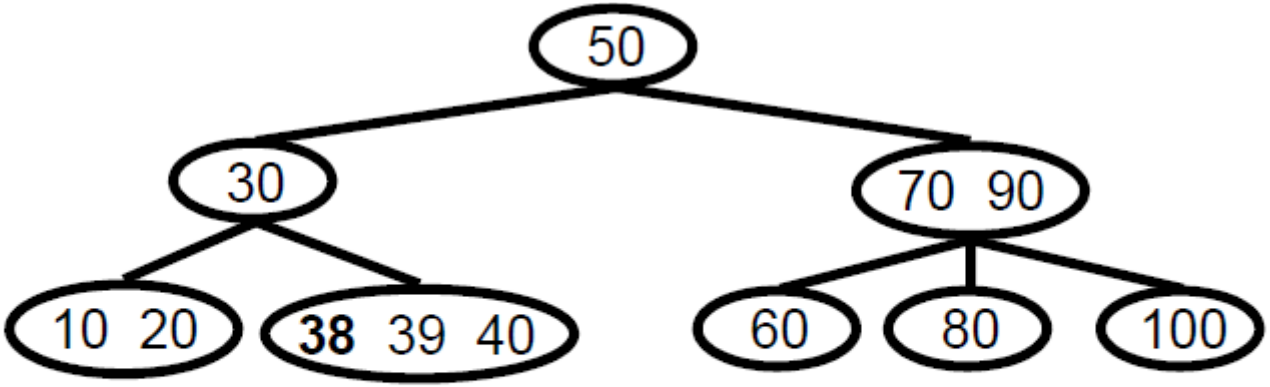
Insert 39



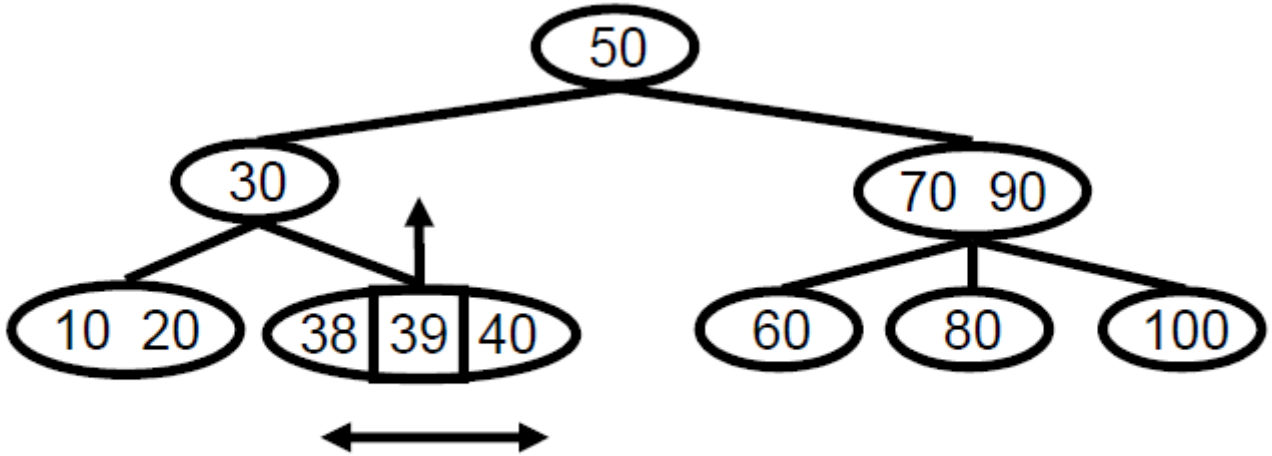
Done!



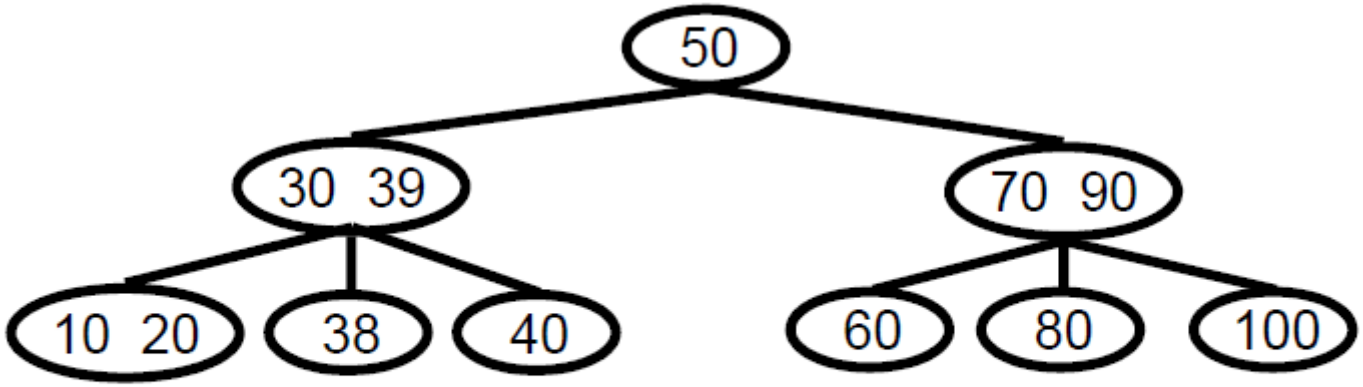
Insert 38



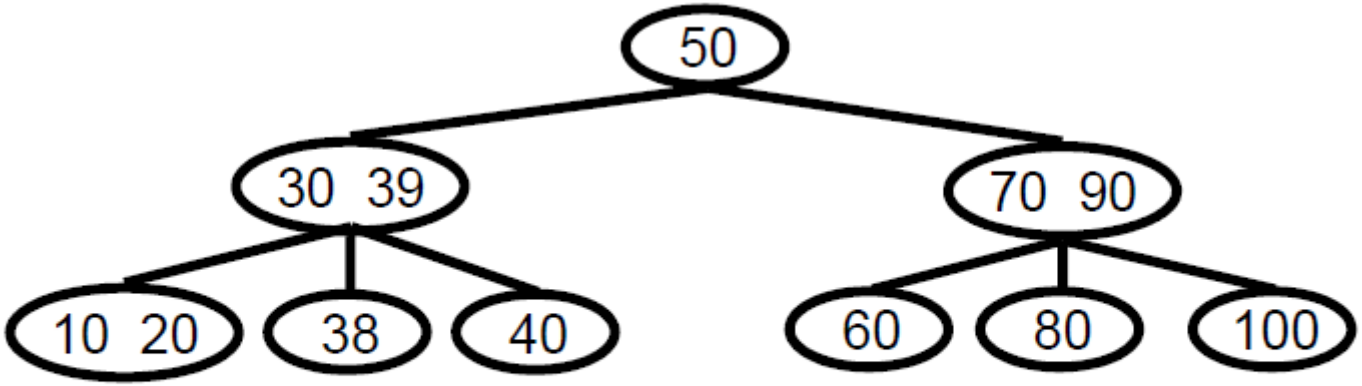
Insert 38



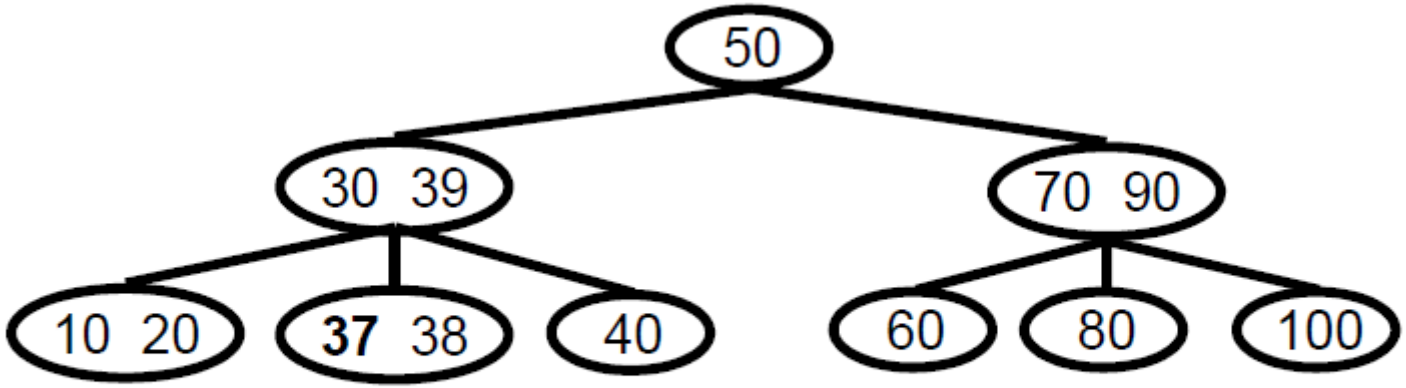
Push up, split apart



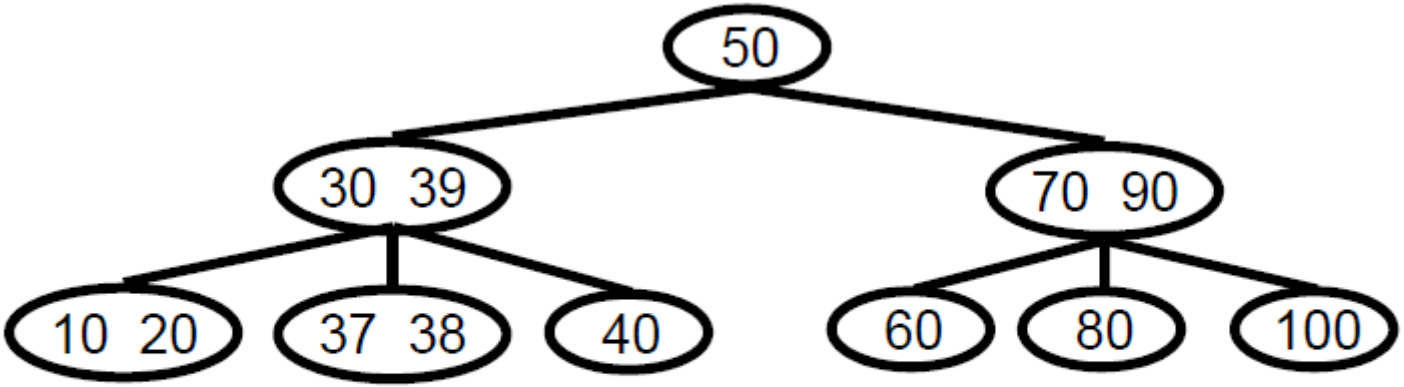
Done!



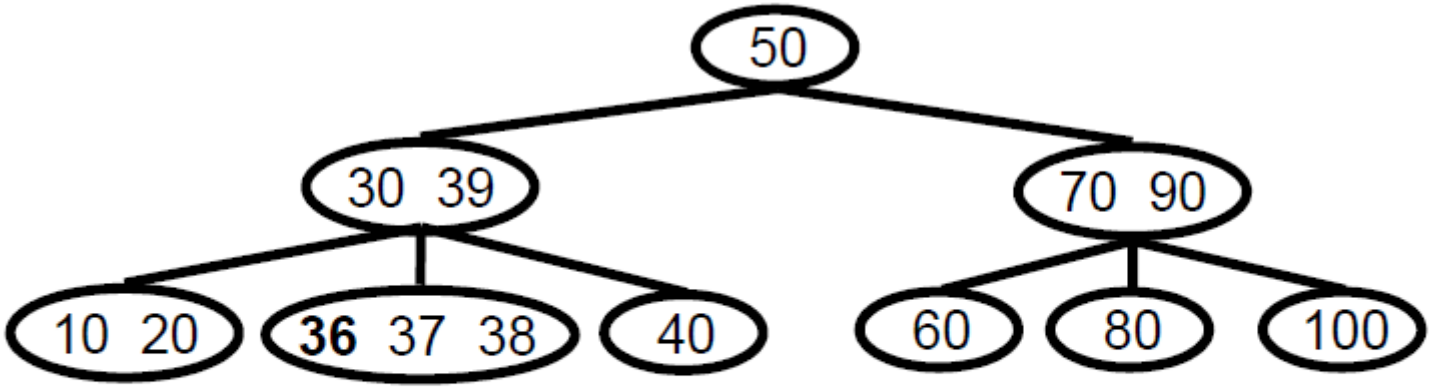
Insert 37



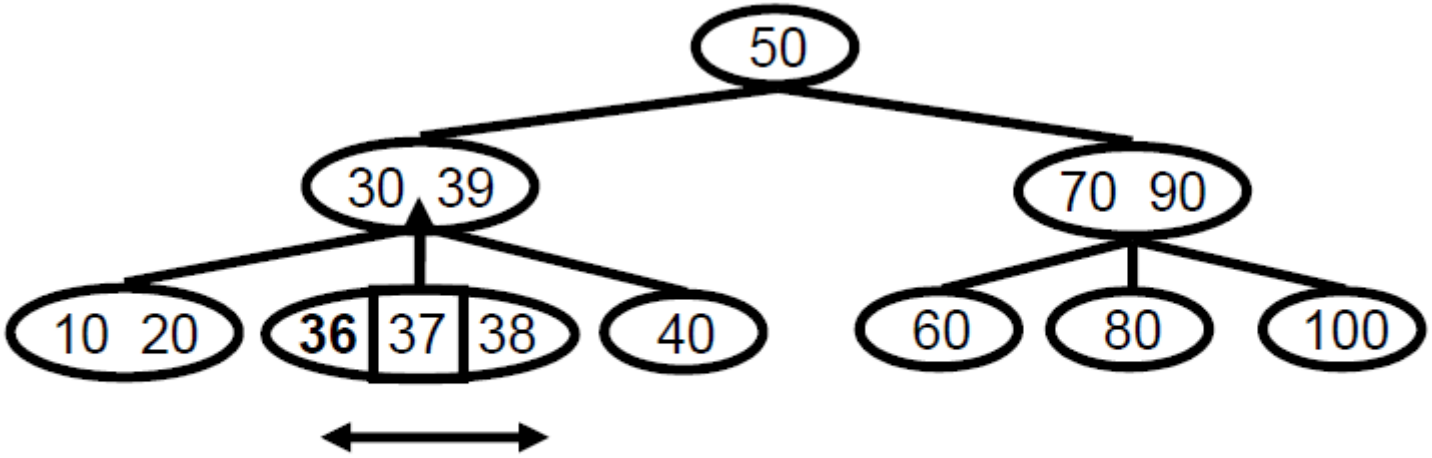
Done!



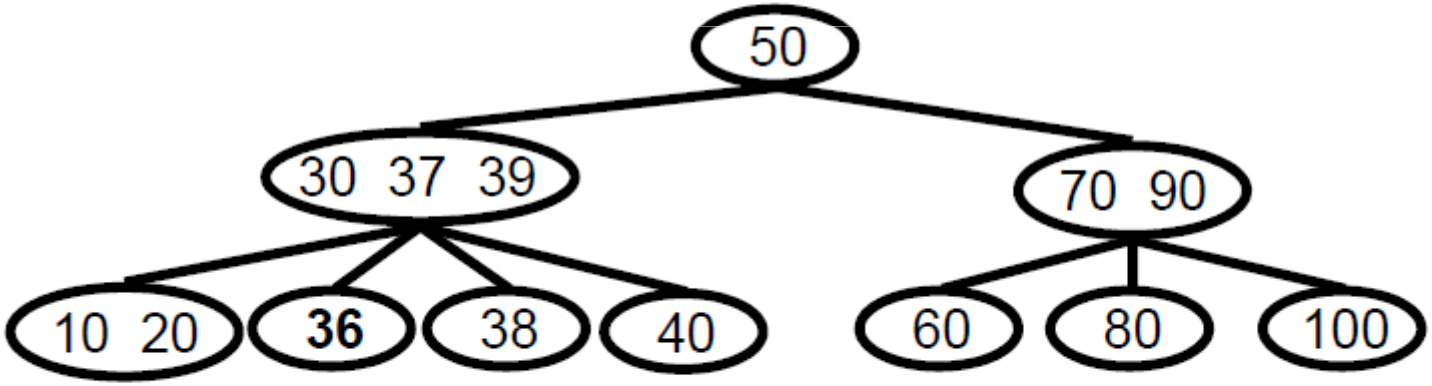
Insert 36



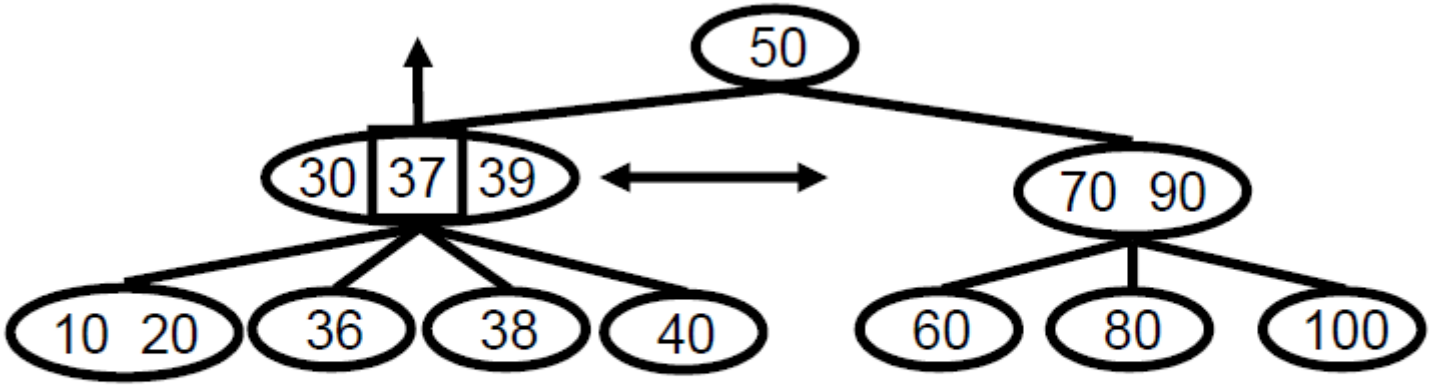
Insert 36



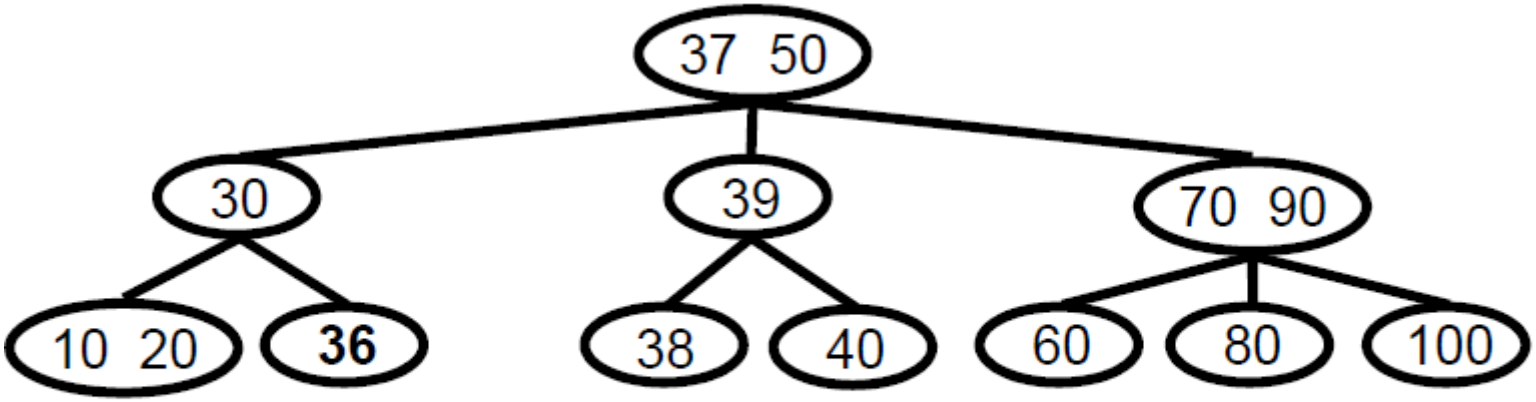
Push up, split apart



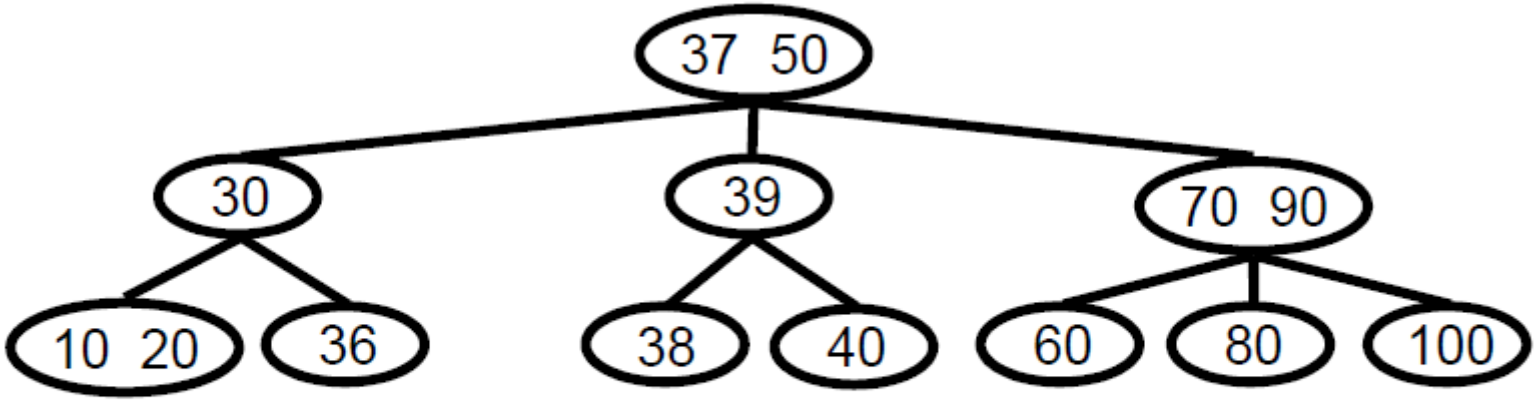
Need to go further up the tree to resolve overcrowding



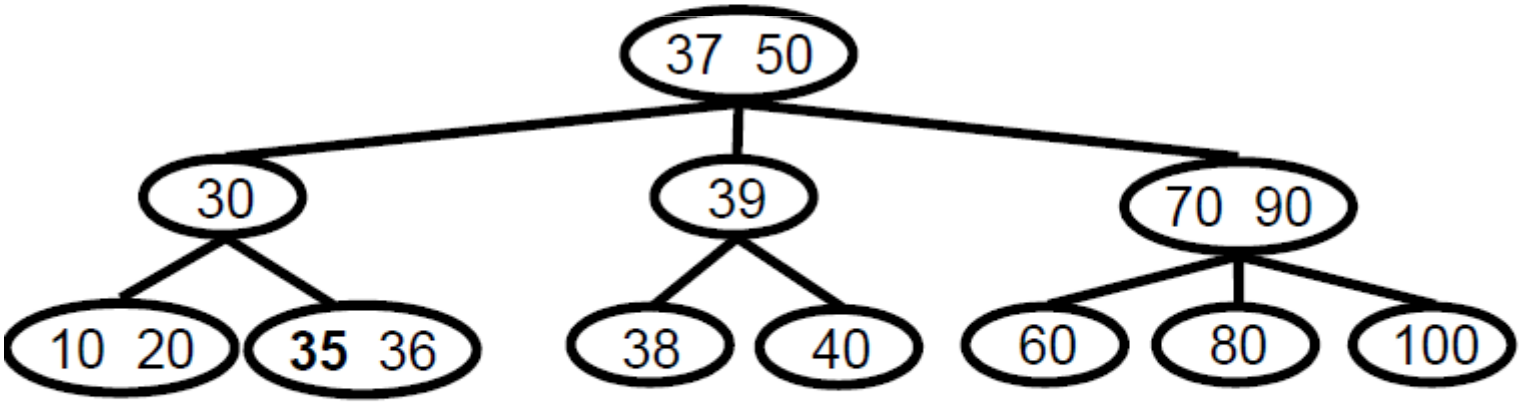
Push up, split apart



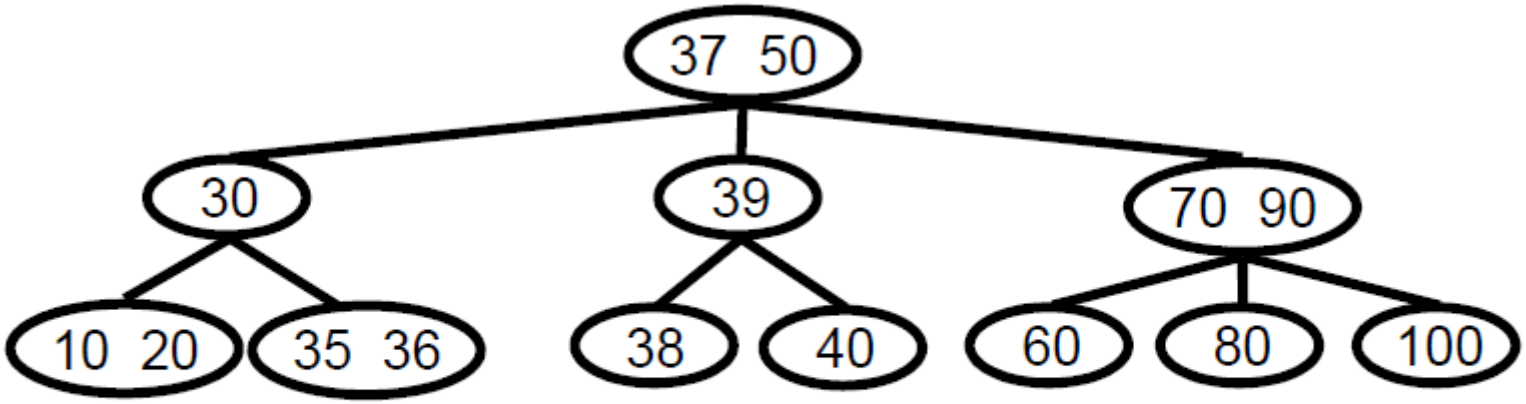
Done!



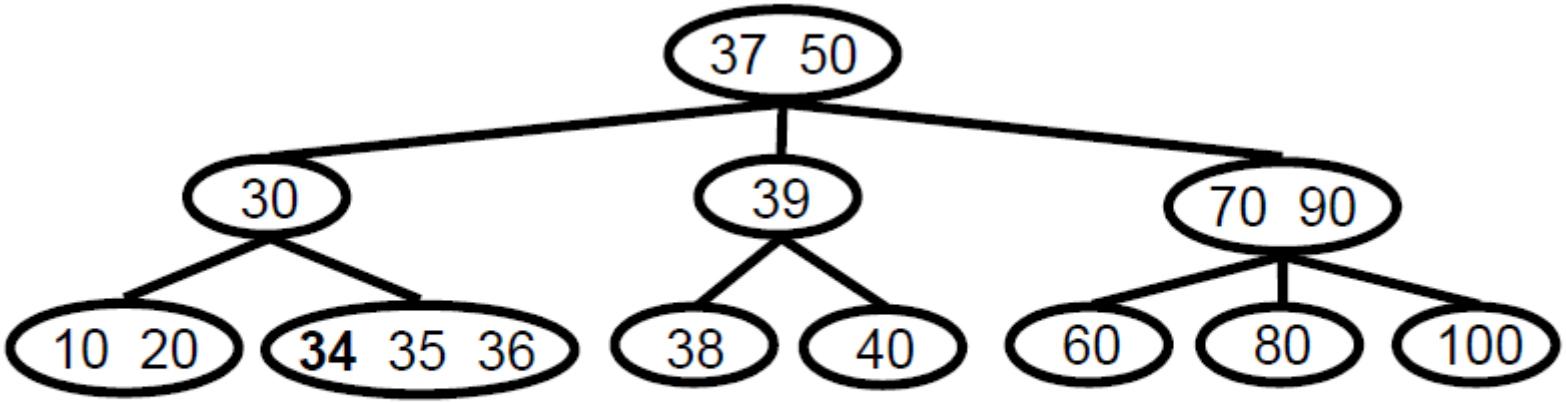
Insert 35



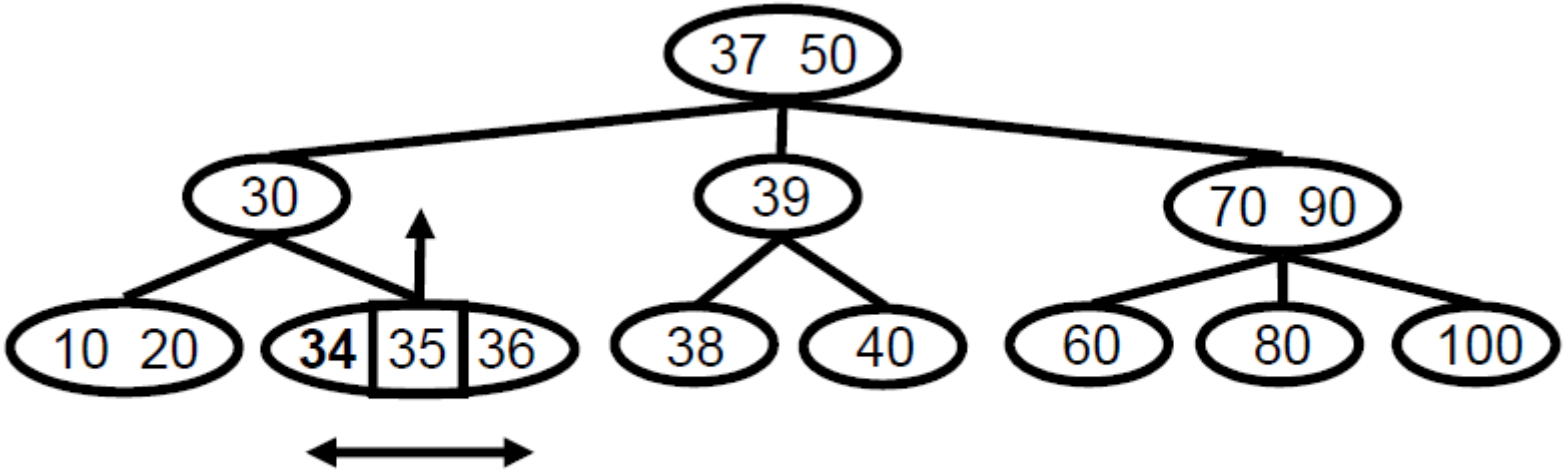
Done!



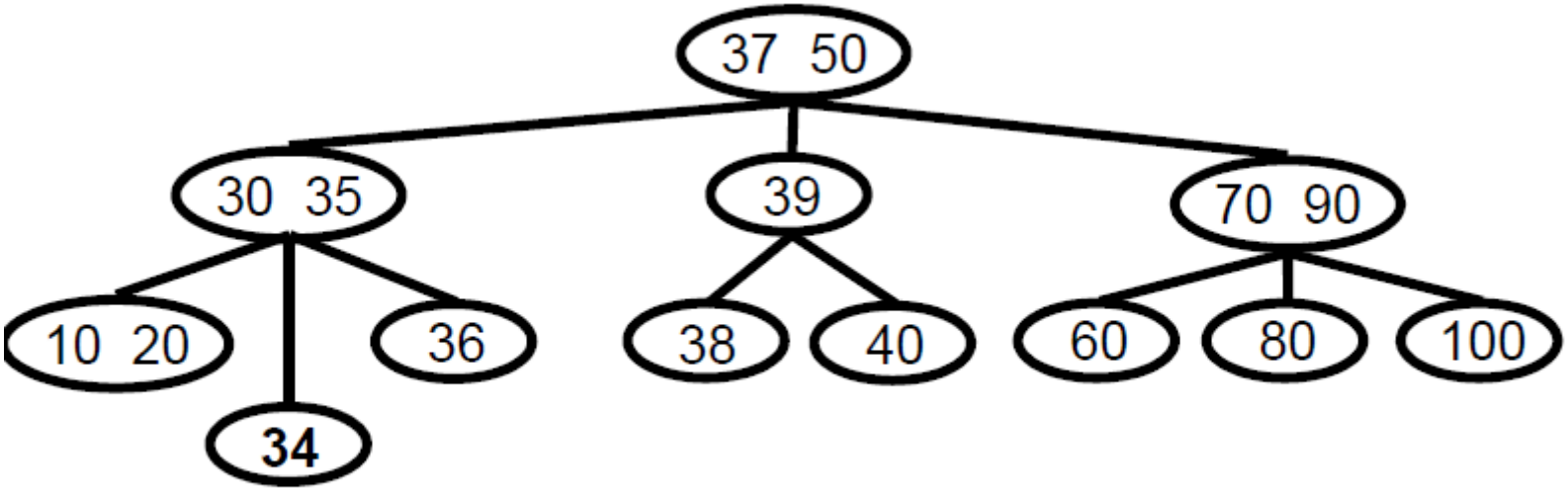
Insert 34



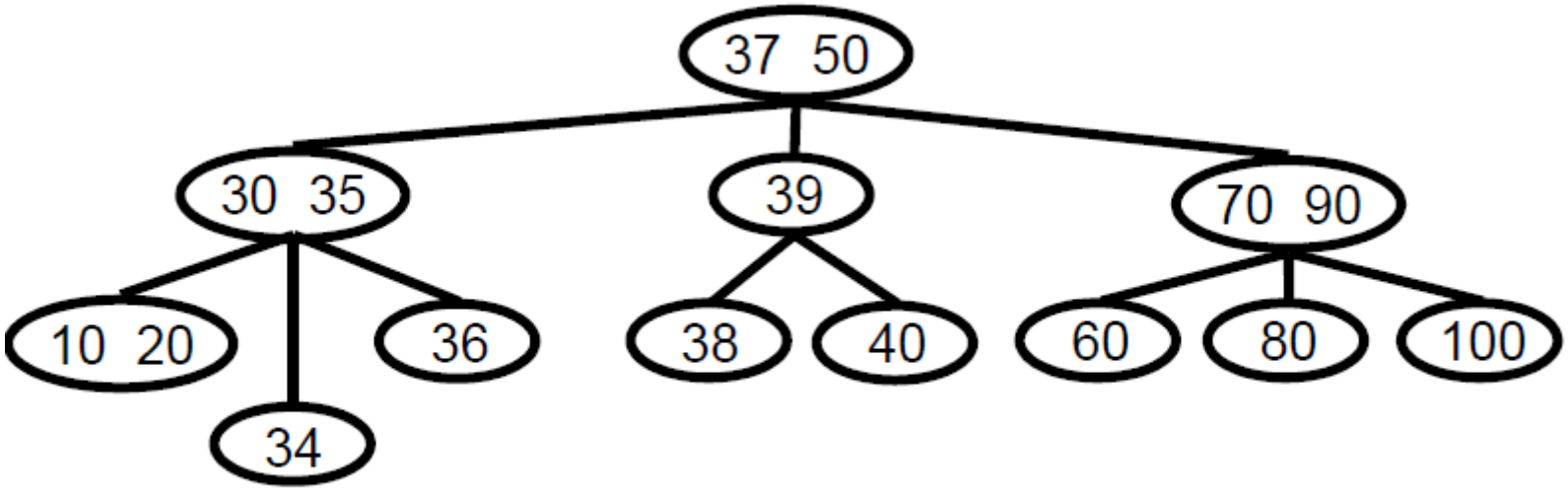
Insert 34



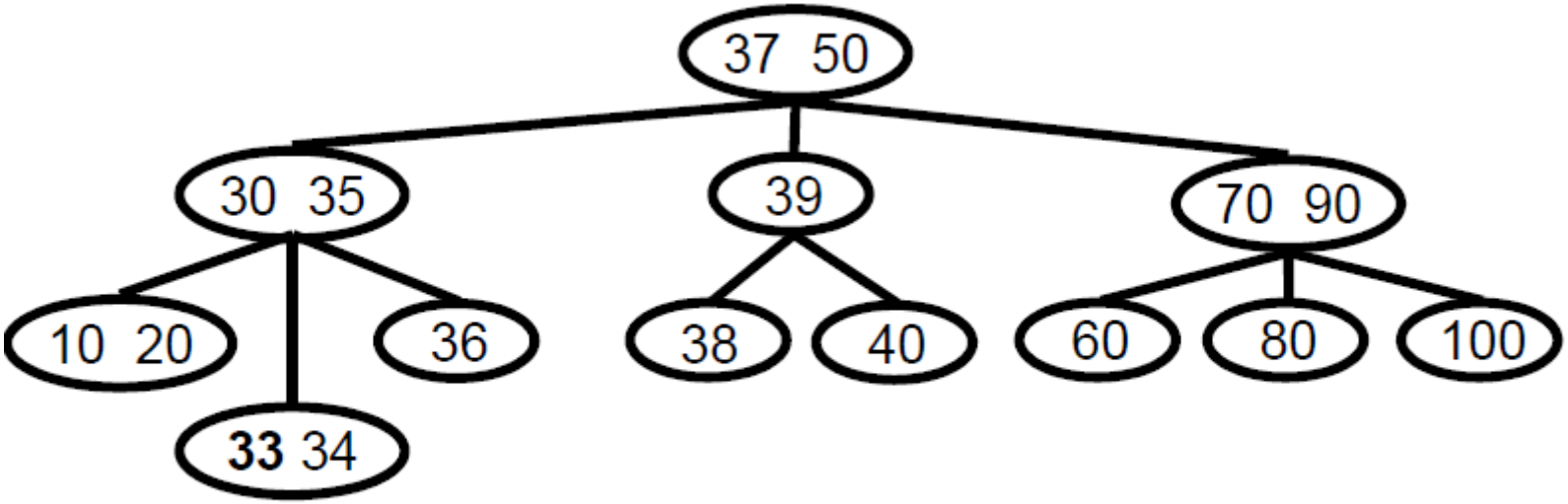
Push up, split apart



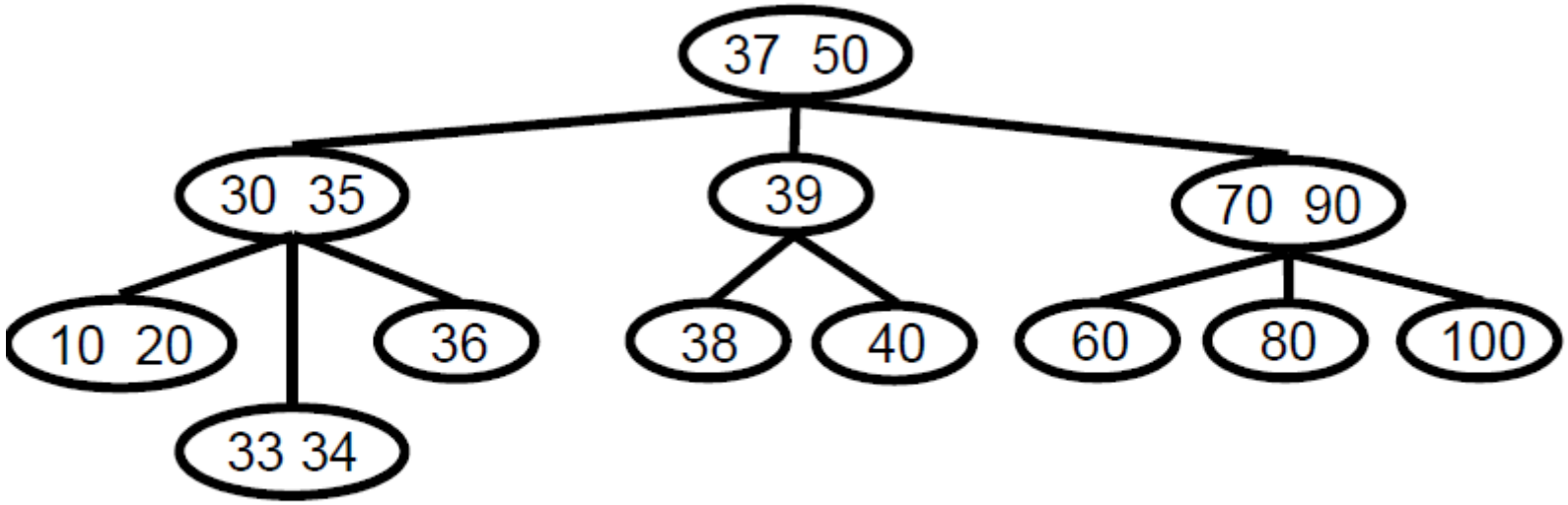
Done!



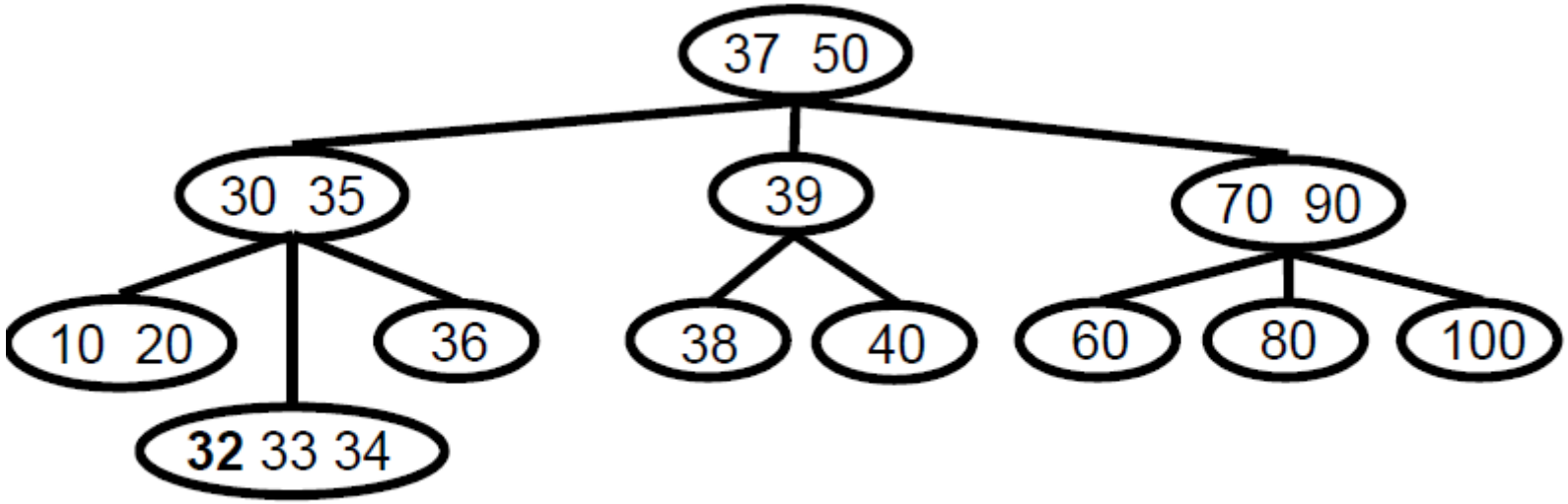
Insert 33



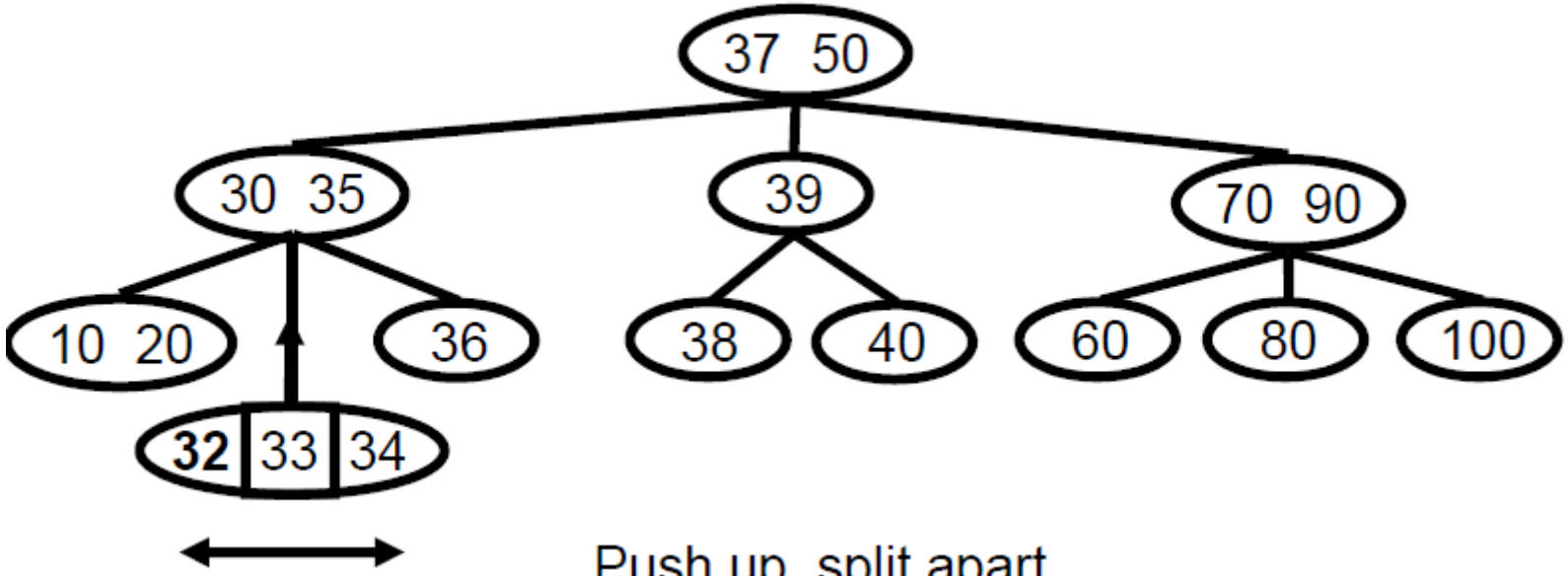
Done!



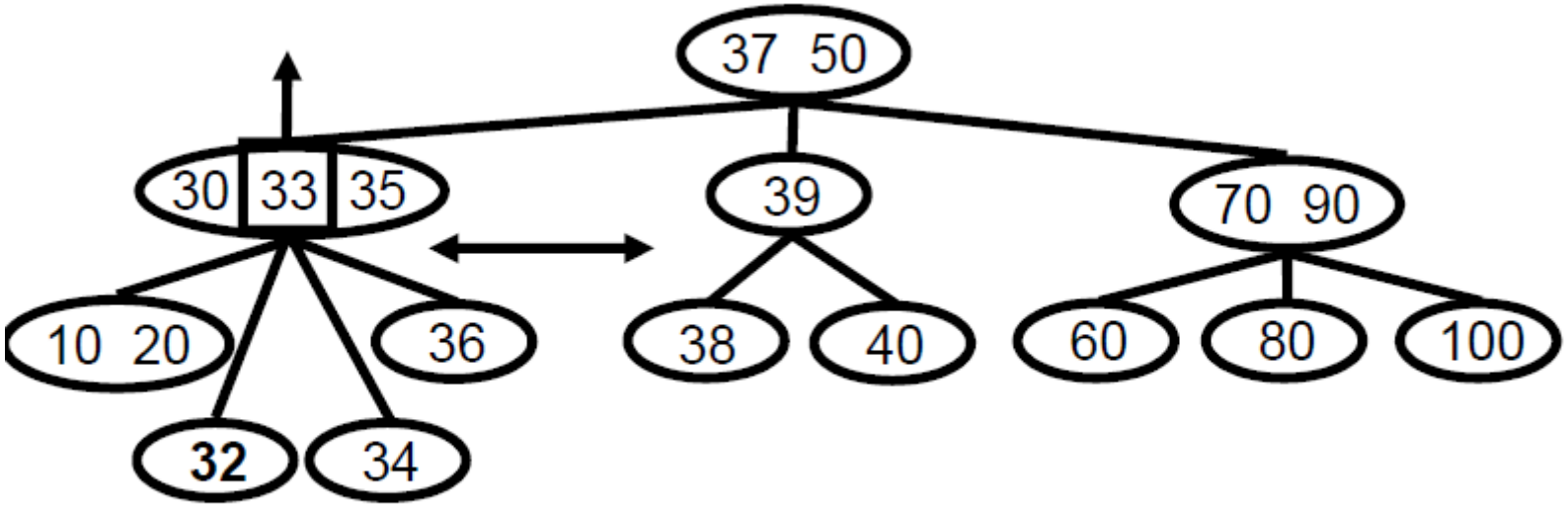
Insert 32



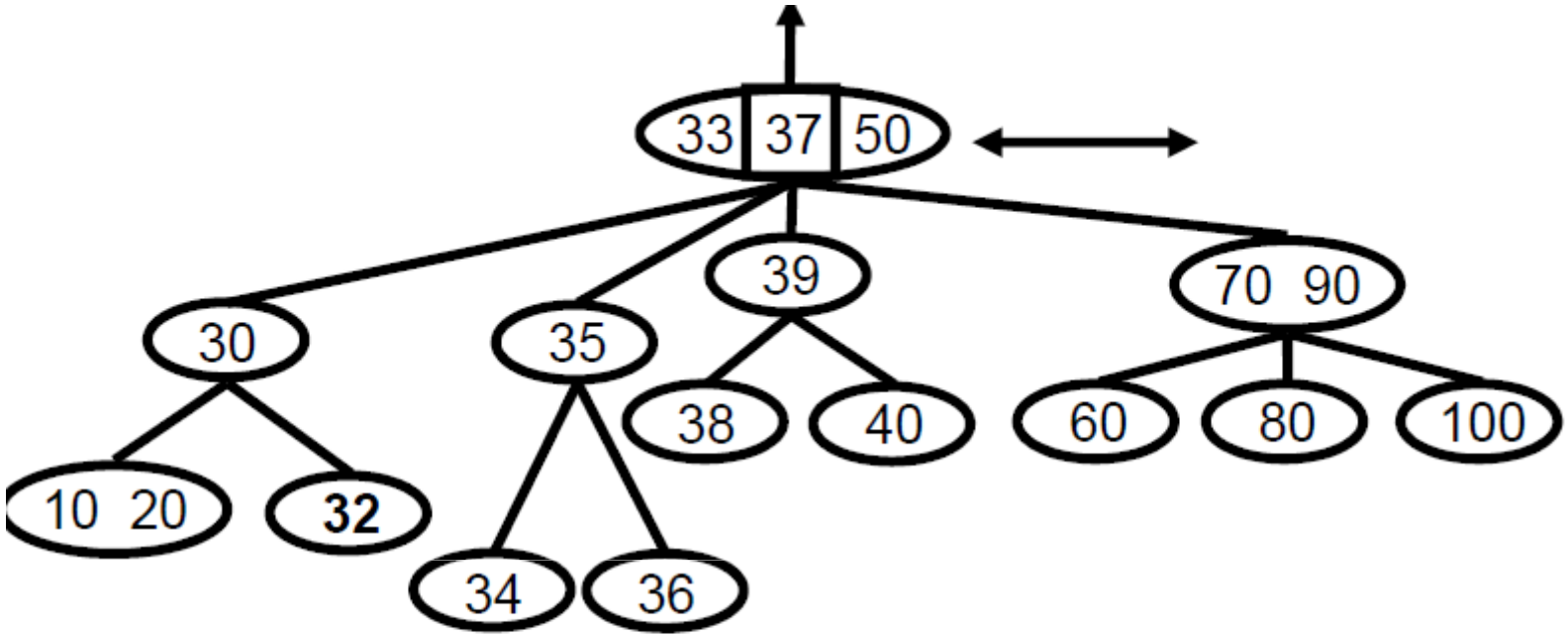
Insert 32



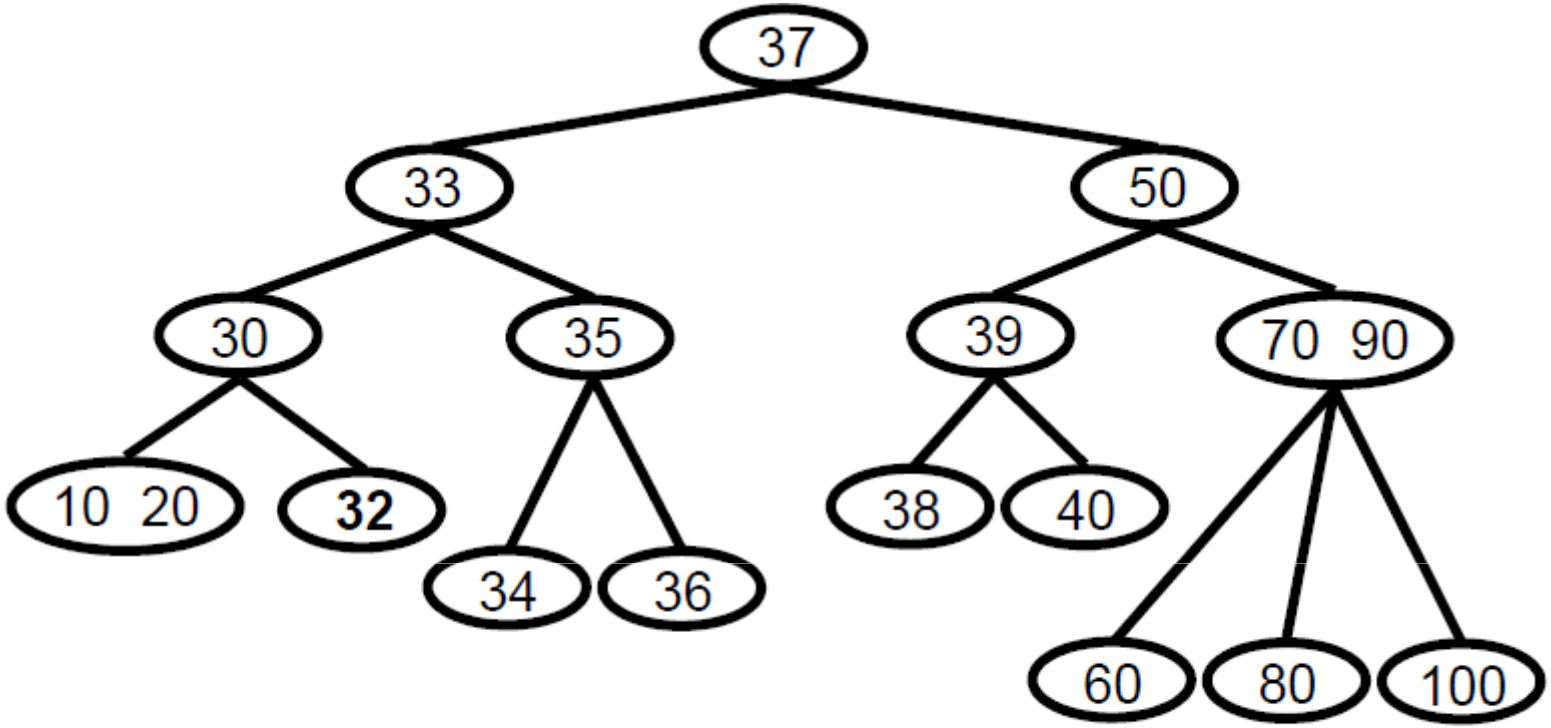
Push up, split apart



Push up, split apart



Push up, split apart



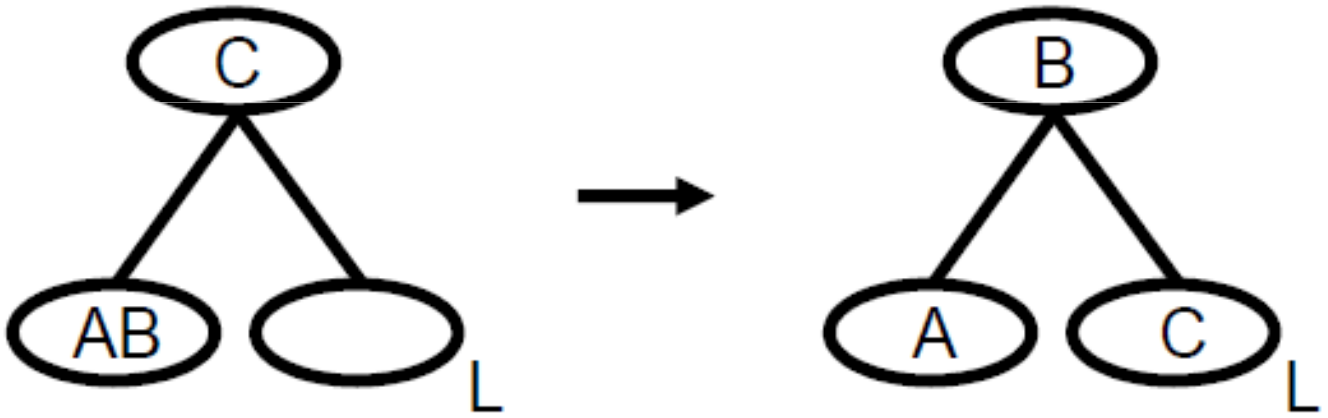
A new level!

Deletion From a 2-3 Tree

- To delete a key K, first locate the node N containing K.
  - If K is not found, then deletion algorithm terminates.
- If N is an interior node, find K's in-order successor and swap it with K. As a result, deletion always begins at a leaf node L.
- If leaf node L contains a value in addition to K, delete K from L, and we're done. (no underflow)
  - For B-trees, underflow occurs if # of keys < minimum.
- If underflow occurs (node has less than required # of keys), we merge it with its neighboring nodes.
  - Check siblings of leaf. If sibling has two values, redistribute them.
  - Otherwise, merge L with an adjacent sibling and bring down a value from L's parent.
  - If L's parent has underflow, recursively apply merge procedure.
  - If underflow occurs to the root, the tree may shrink a level.

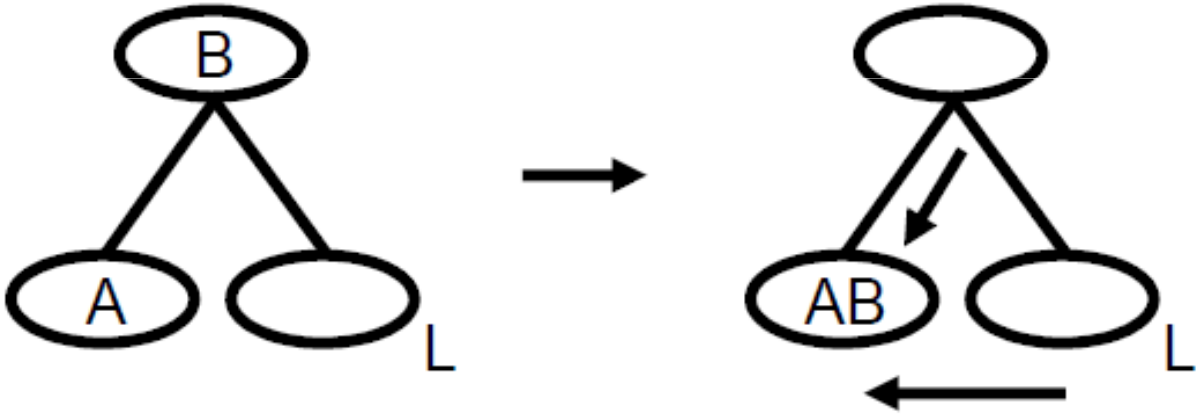
Deletion: Re-distributing values in Leaf Nodes

- If deleting K from L causes L to be empty:
- Check siblings of now empty leaf.
- If sibling has two values, redistribute the values.



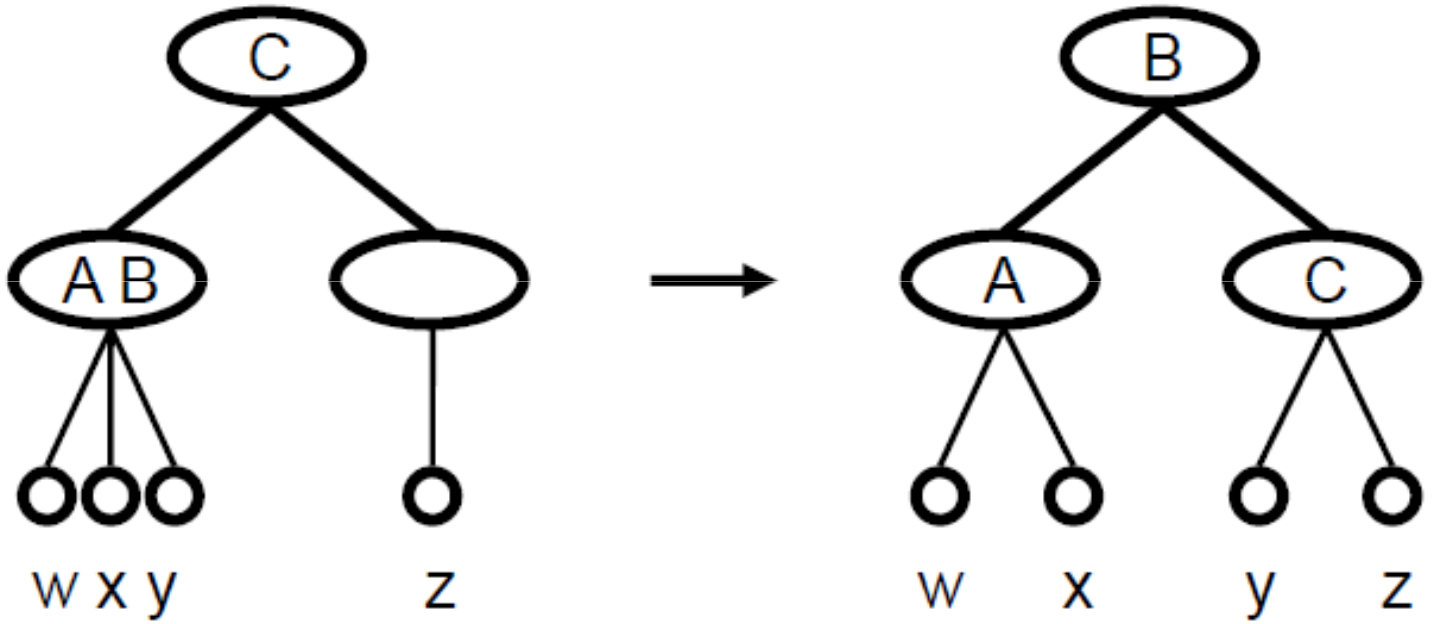
Deletion: Merging Leaf Nodes

- If no sibling node has extra keys to spare, merge L with an adjacent sibling and bring down a value from L's parent.
- The merging of L may cause the parent to be left without a value and only one child. If so, recursively apply deletion procedure to the parent.



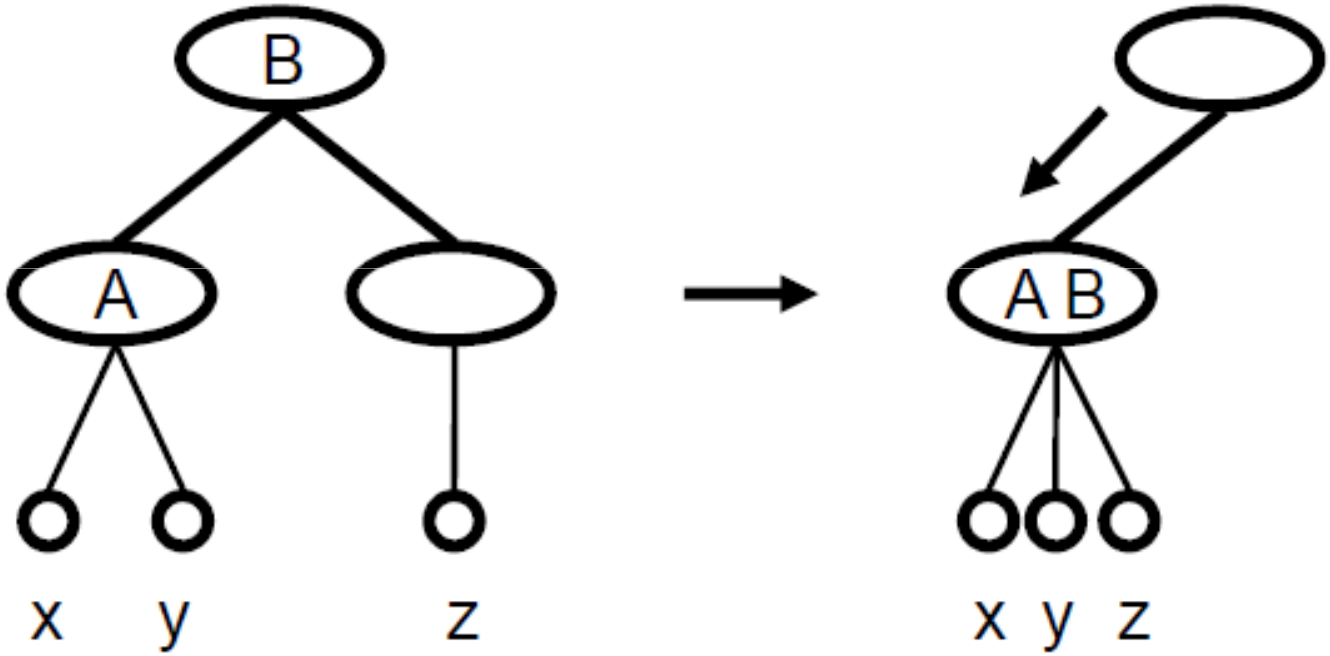
Deletion: Re-distributing values in Interior Nodes

- If the node has a sibling with two values, redistribute the values.



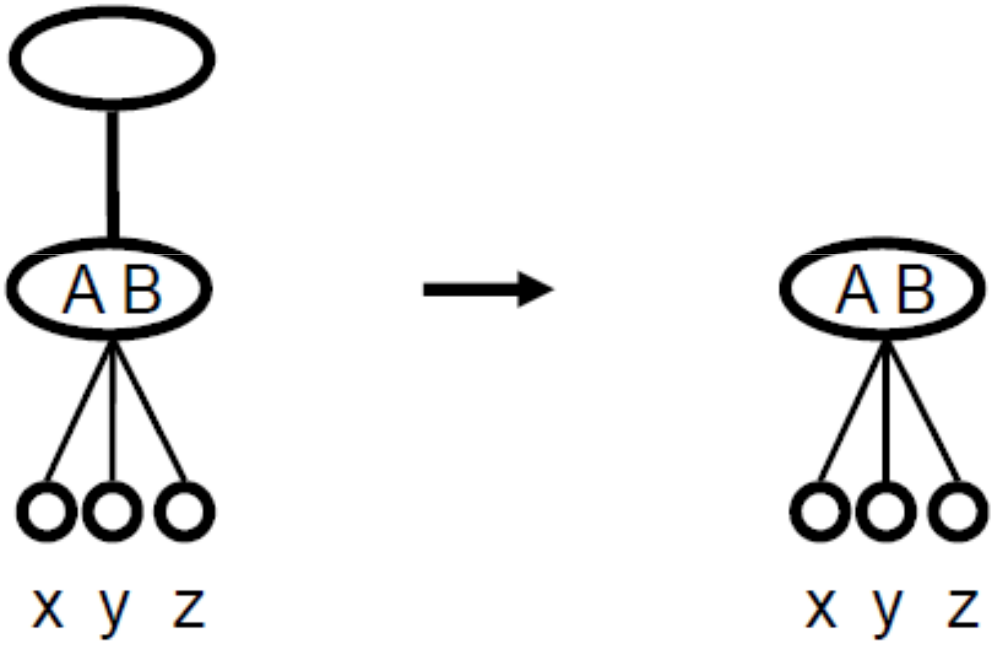
Deletion: Merging Interior Nodes

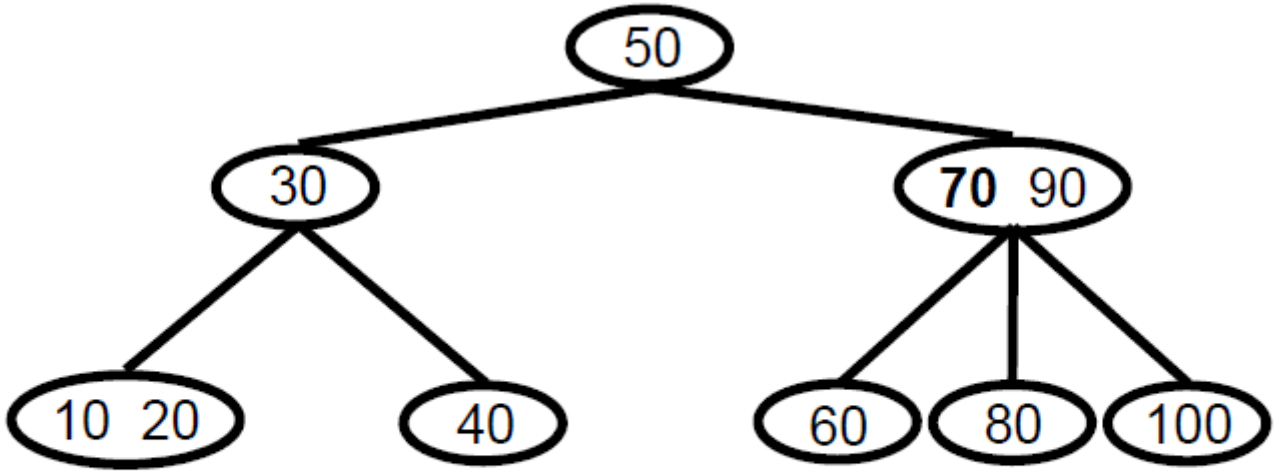
- If the node has no sibling with two values, merge the node with a sibling, and let the sibling adopt the node's child.



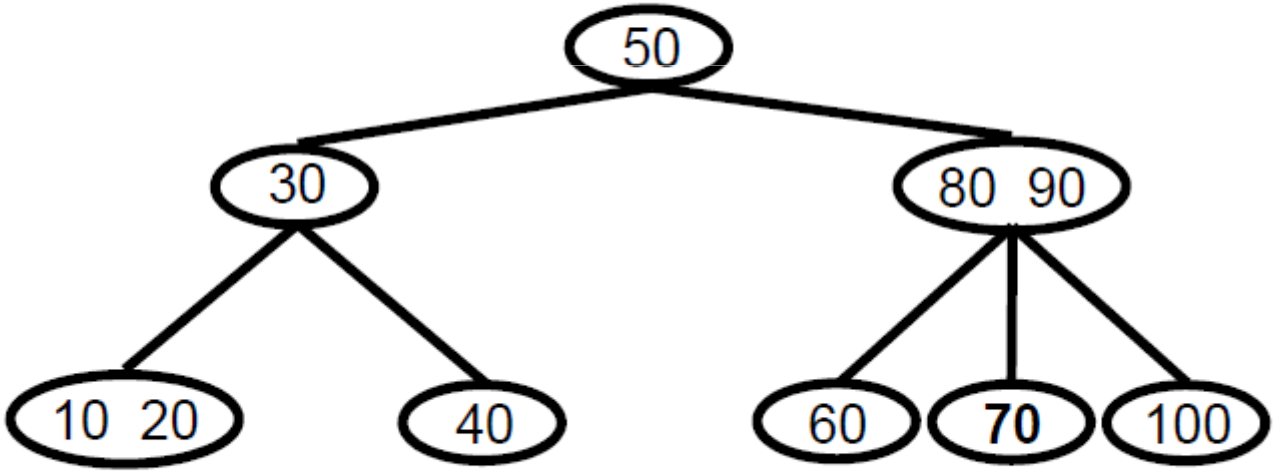
Deletion: Merging on the Root Node

- If the merging continues so that the root of the tree is without a value (and has only one child), delete the root.
- Height will now be h-1.

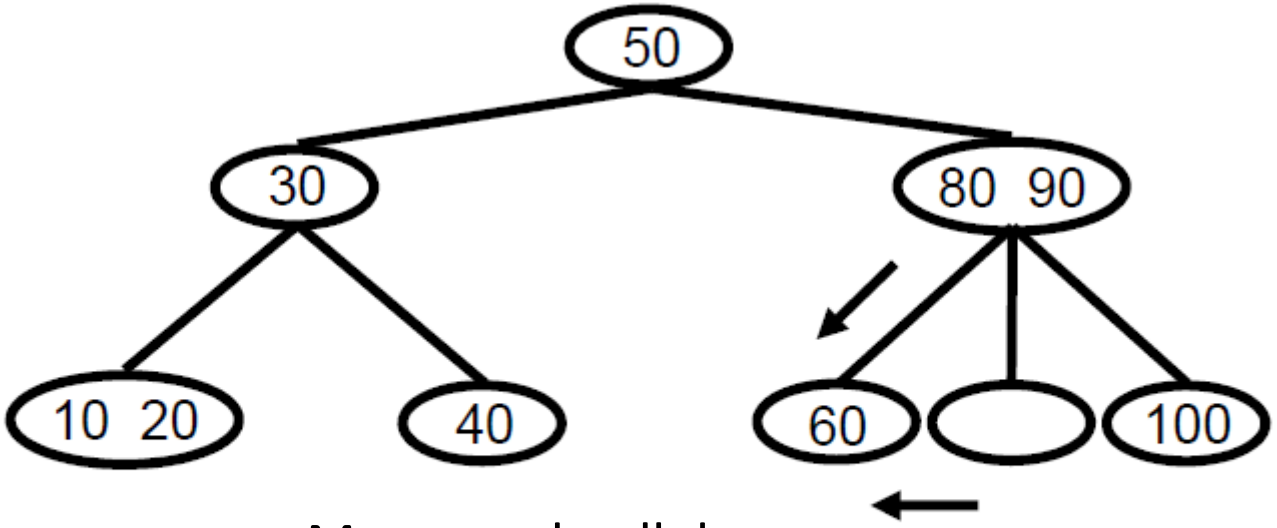




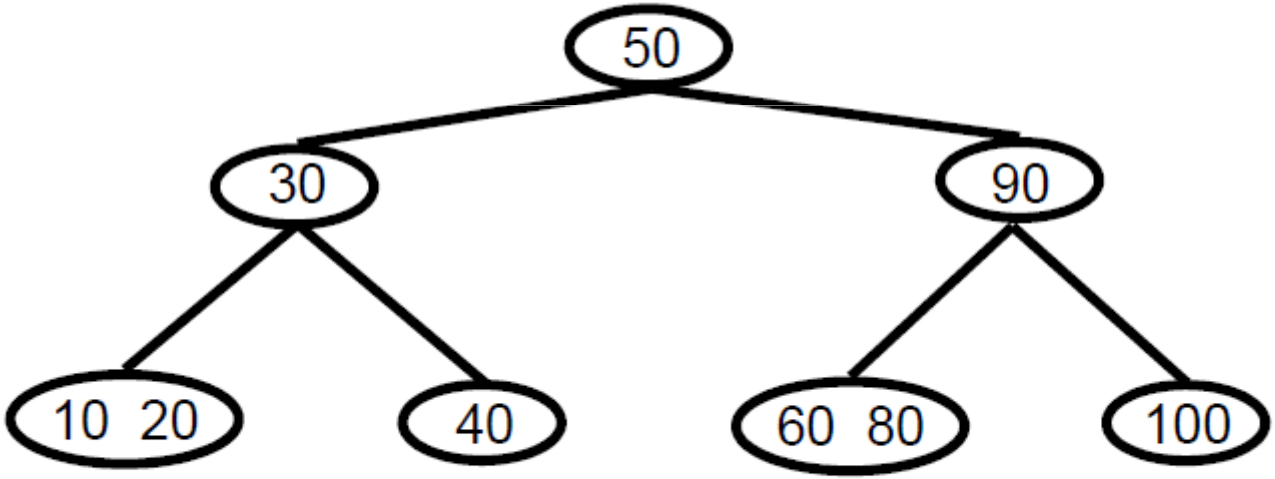
Delete 70



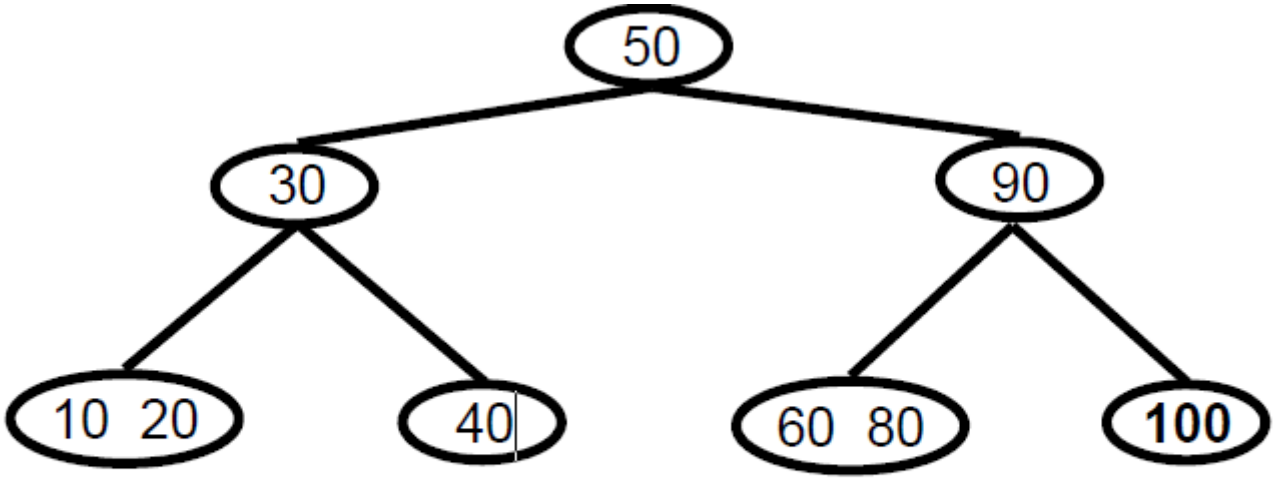
Swap with in-order successor



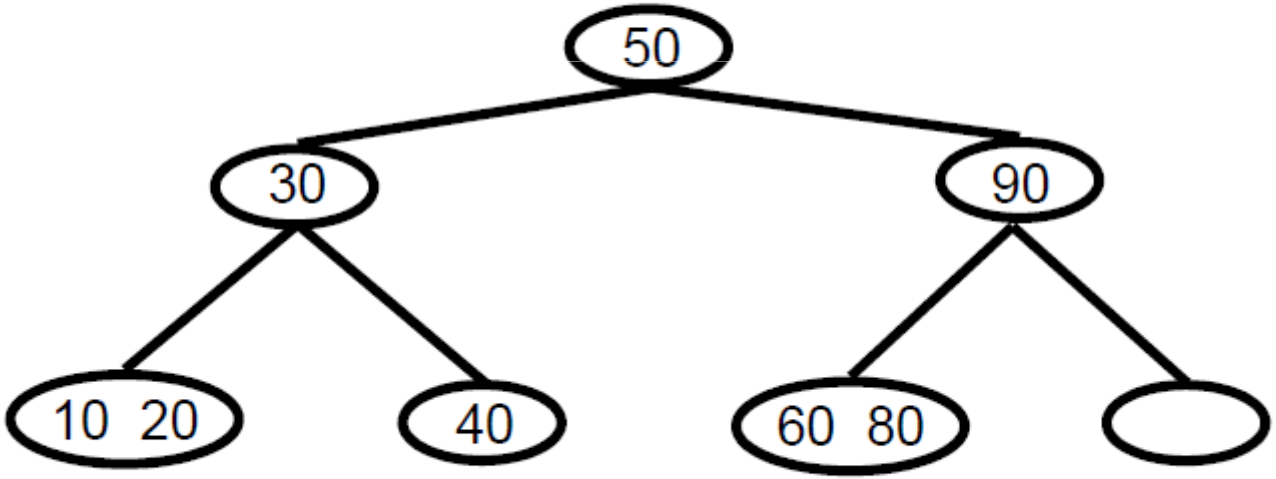
Merge and pull down



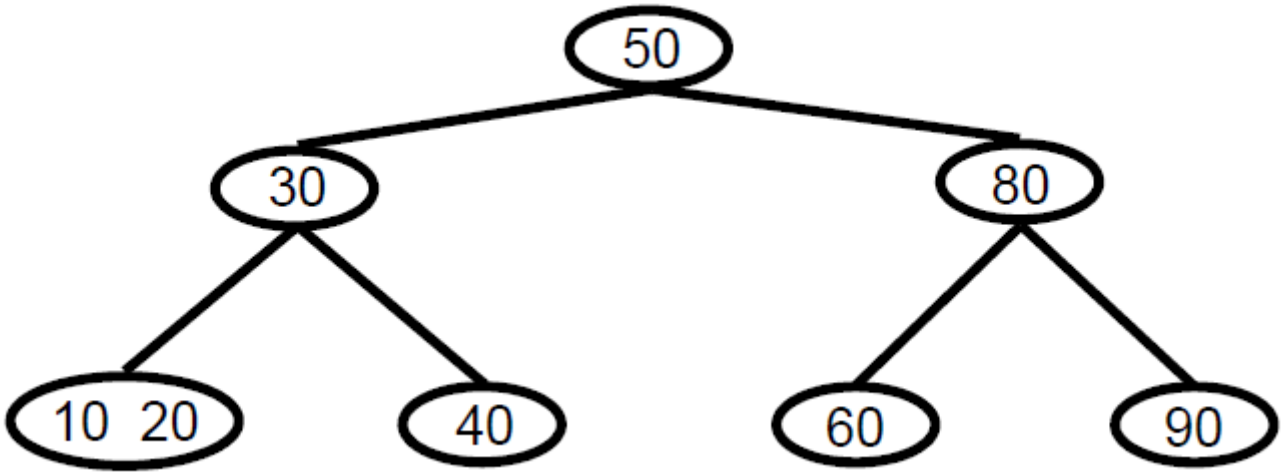
Done!



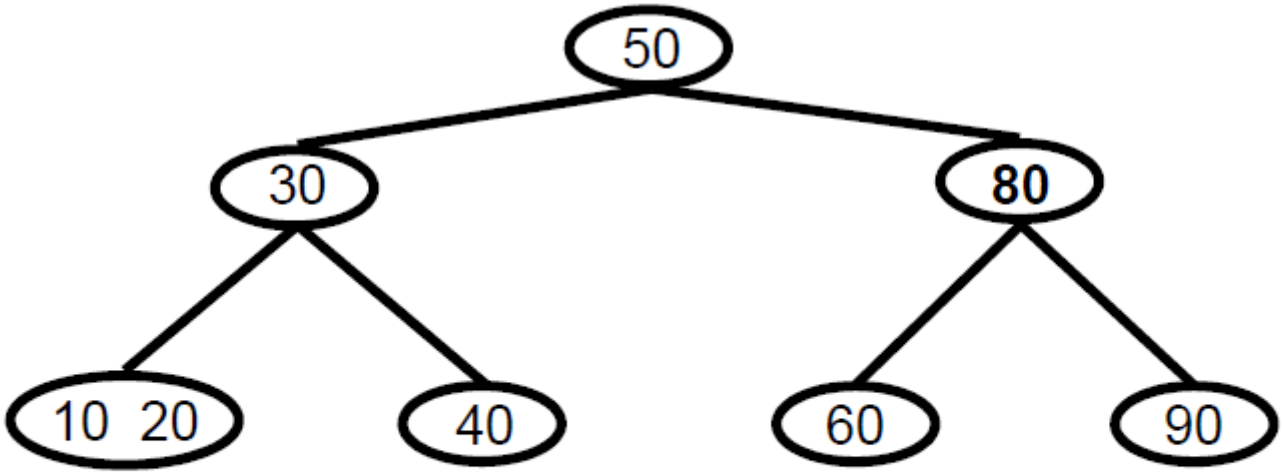
Delete 100



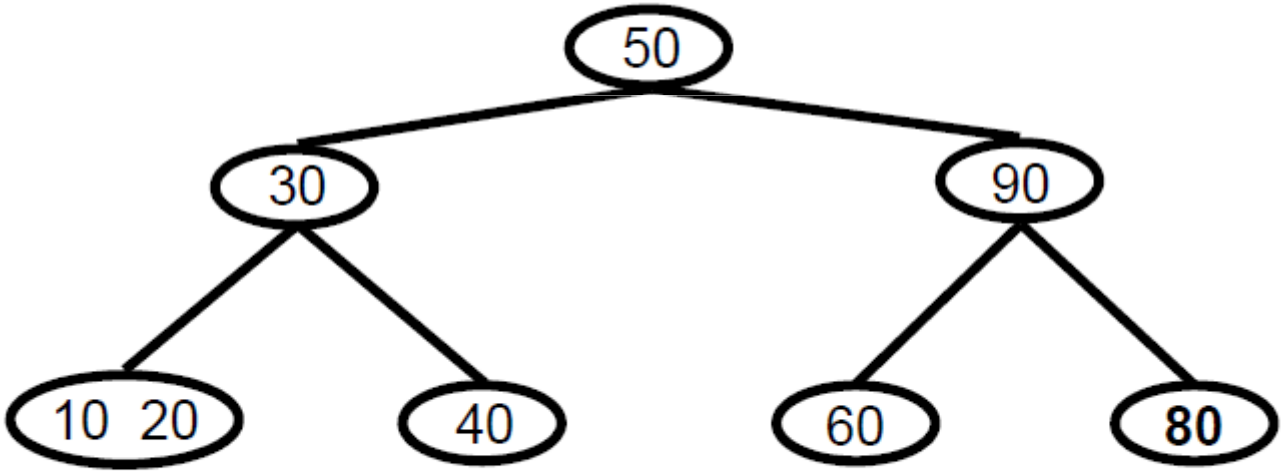
Redistribute



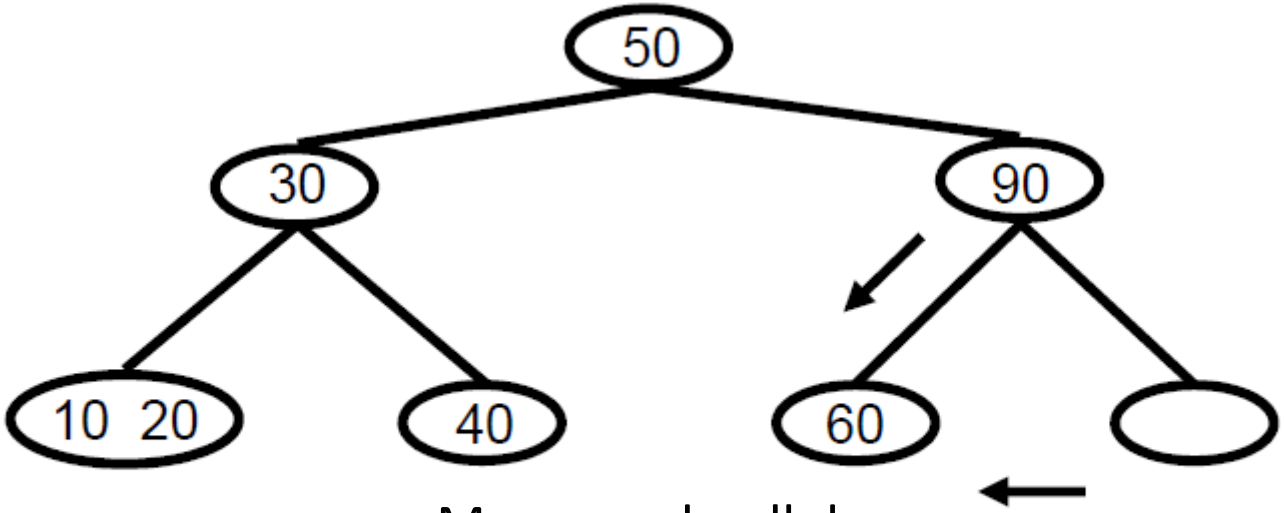
Done!



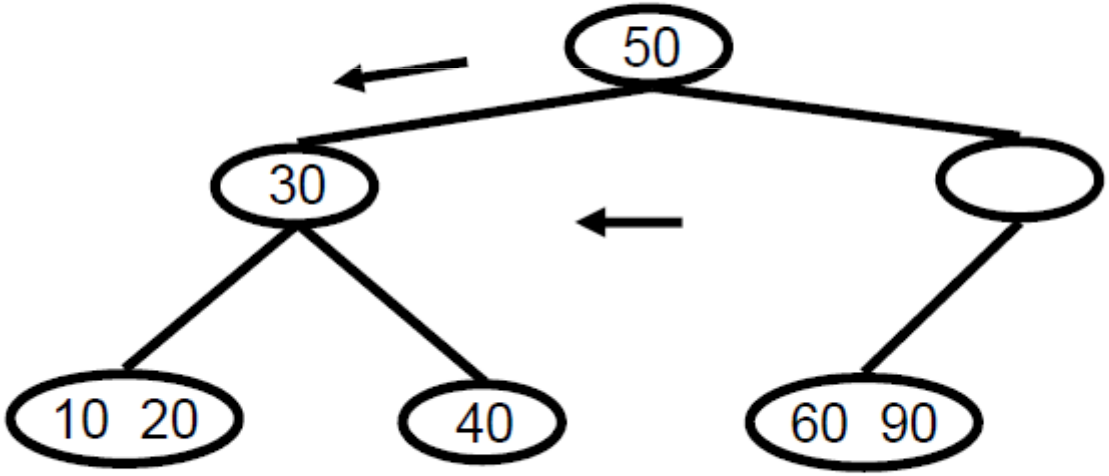
Delete 80



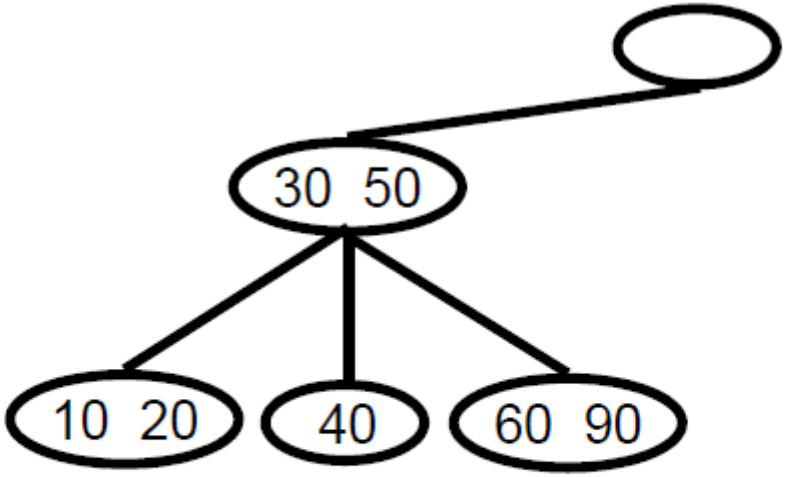
Swap with in-order successor



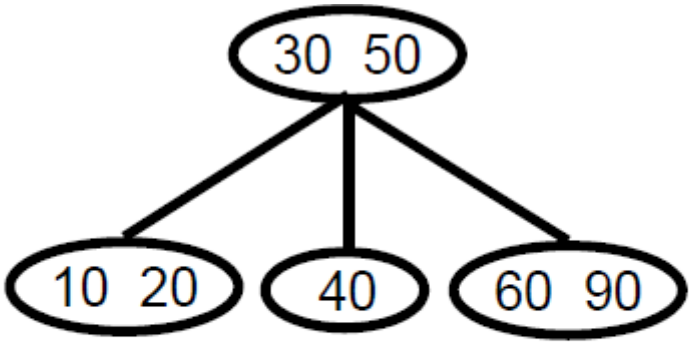
Merge and pull down



Merge and pull down



Merge and pull down



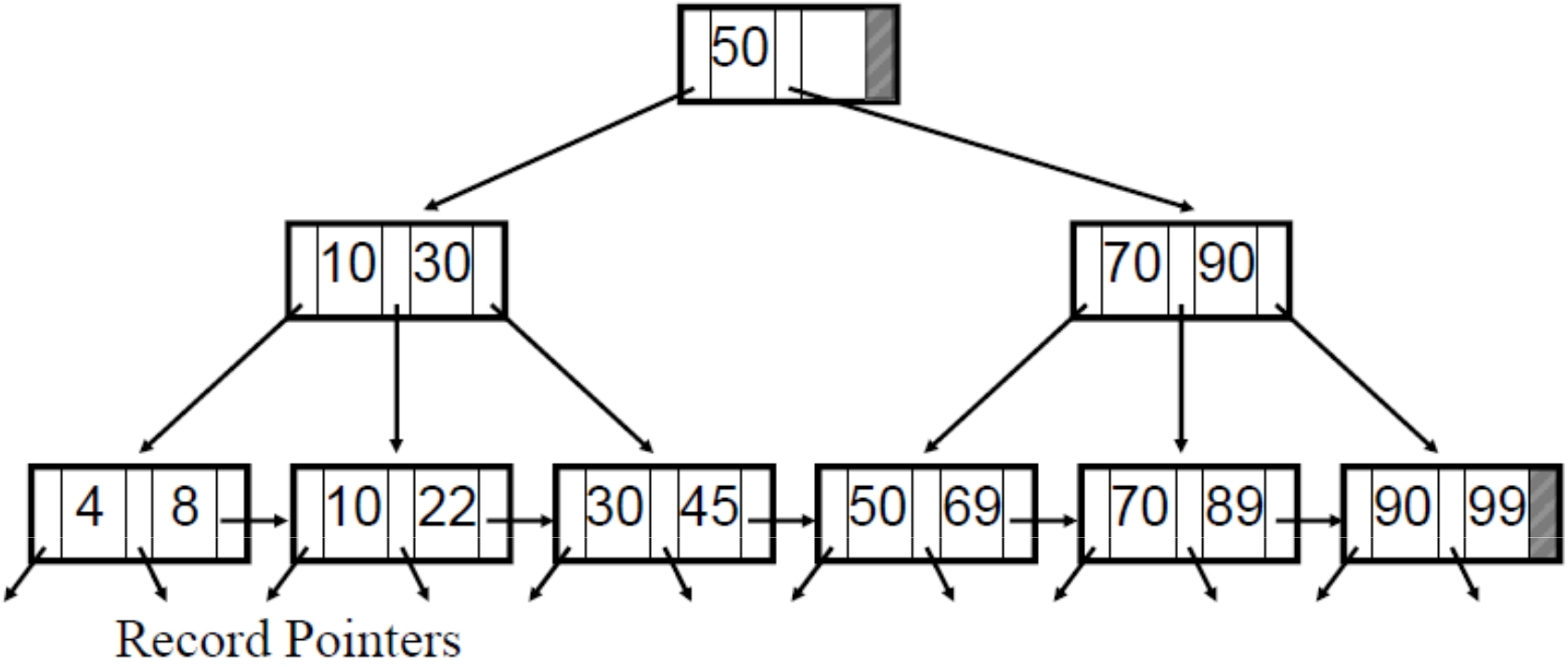
Done

B-trees as External Data Structures

- A regular B-tree can be used as an index:
  - Each node in the B-tree stores not only keys, but also a record pointer for each key to the actual data being stored.
  - To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
- How do we calculate the best B-tree order.
  - Depends on disk block and record size.
  - We want a node to occupy an entire block.
  - Given a block of 4096 bytes, calculate the order of a B-tree if the key size is 4 bytes, and all pointers are 8 bytes.

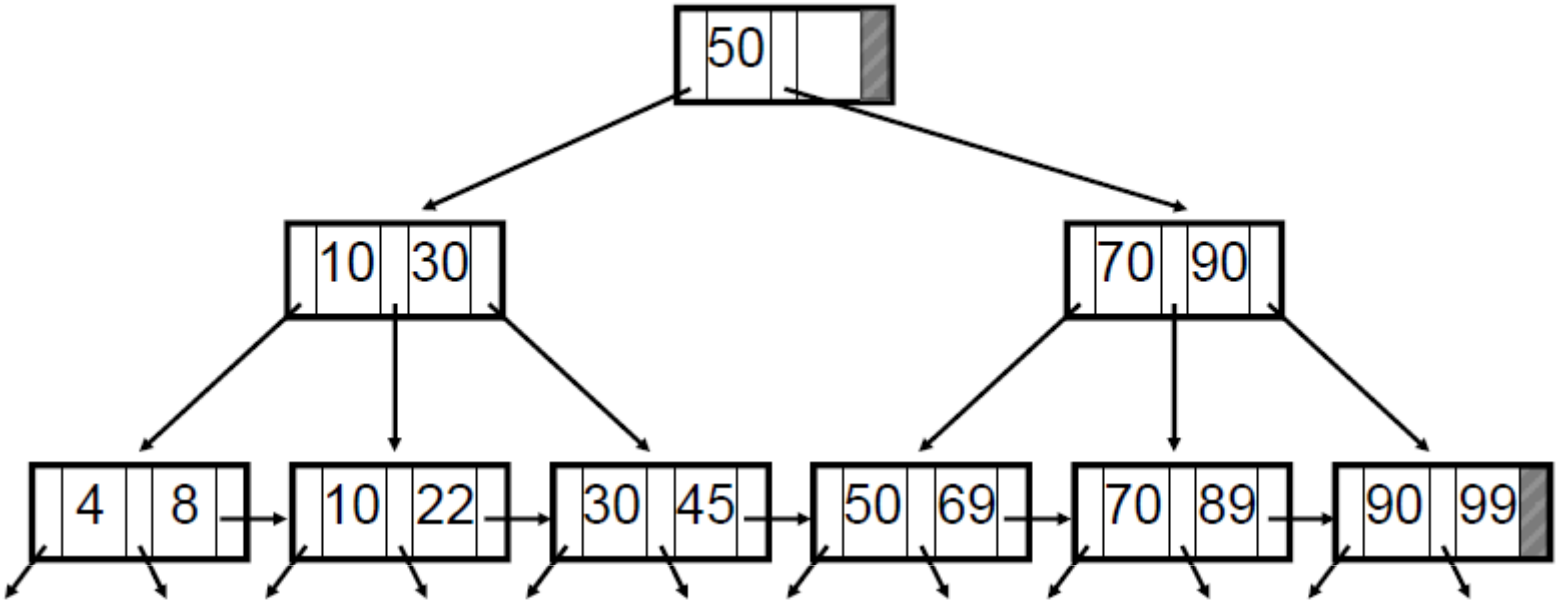
B+-trees

- A B+-tree is a multi-level index structure like a B-tree except that *all data is stored at the leaf nodes* of the resulting tree instead of within the tree itself.
  - Each leaf node contains a pointer to the next leaf node which makes it easy to chain together and maintain the data records in "sequential" order for sequential processing.
- Thus, a B+-tree has two distinct node types:
  - interior nodes - store pointers to other interior nodes or leaf nodes.
  - leaf nodes - store keys and pointers to the data records (or the data records themselves).

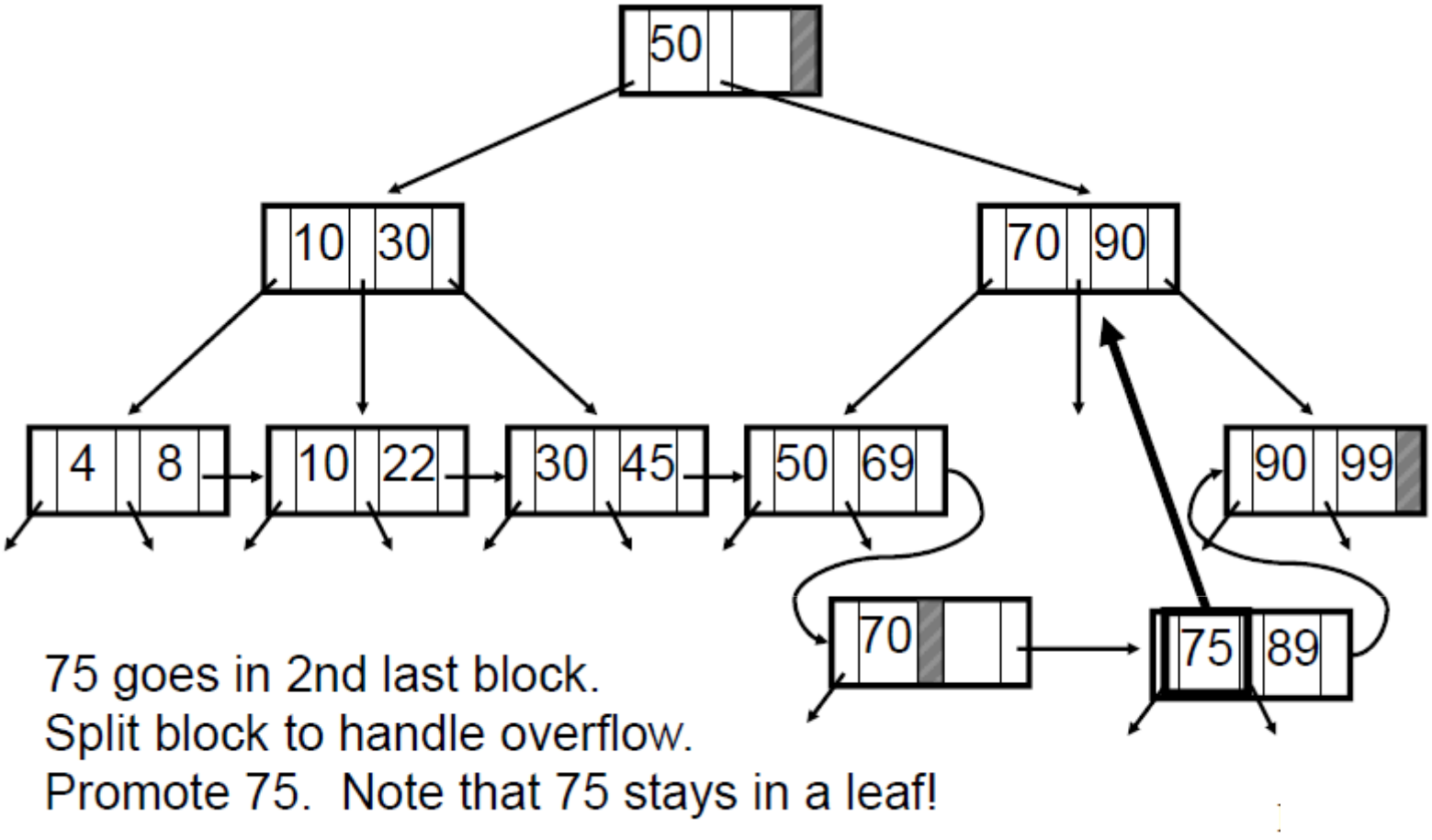

50
10 30
70 90
4 8
10 22
30 45
50 69
70 89
90 99
Record Pointers

Operations on B+-trees

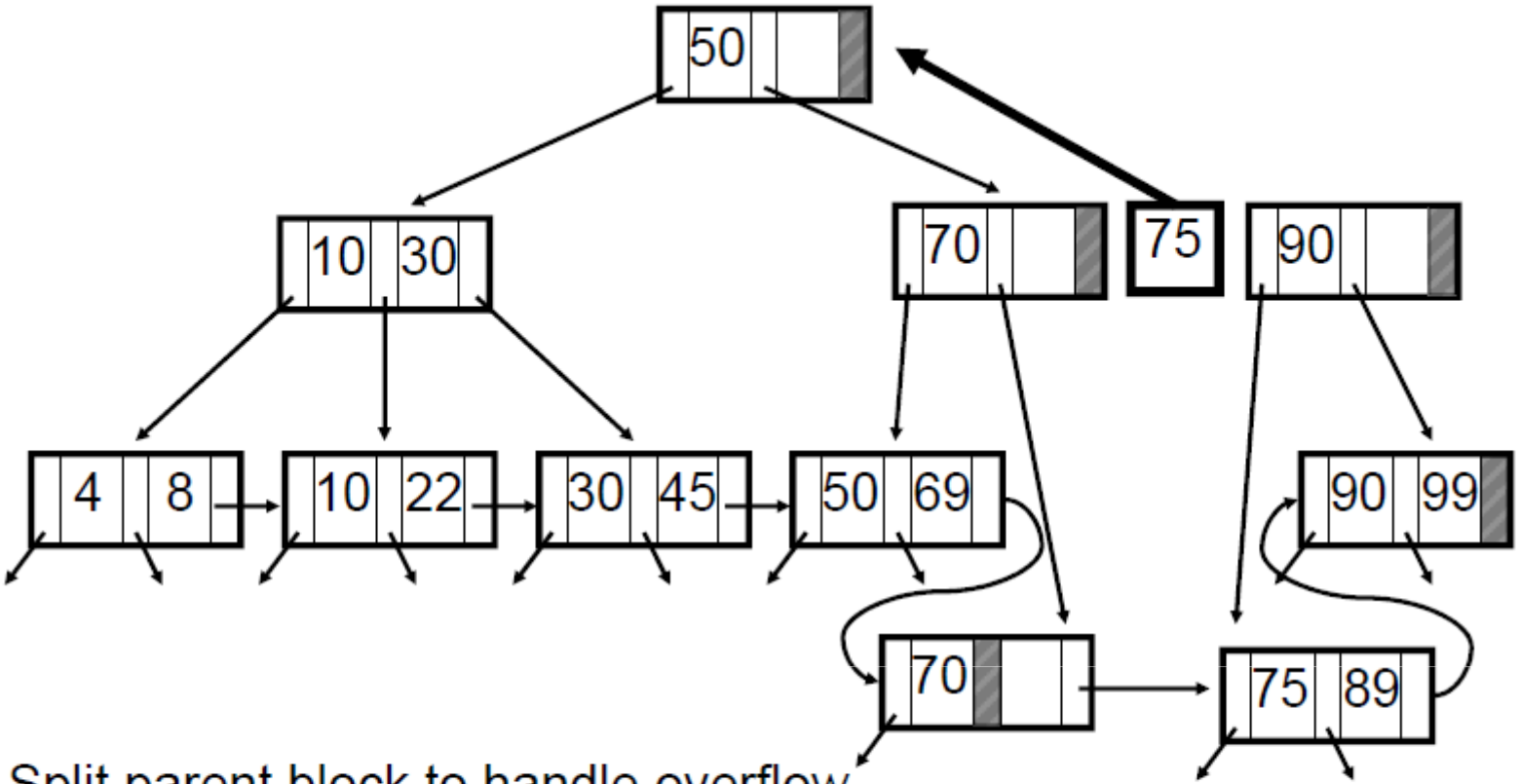
- The general algorithms for inserting and deleting from a B+-tree are similar to B-trees except for one important difference:
  - **All key values stay in leaves.**
- When we must merge nodes for deletion or add nodes during splitting, the key values removed/promoted to the parent nodes from leaves are copies.
  - All non-leaf levels do not store actual data, they are simply a hierarchy of multi-level index to the data.



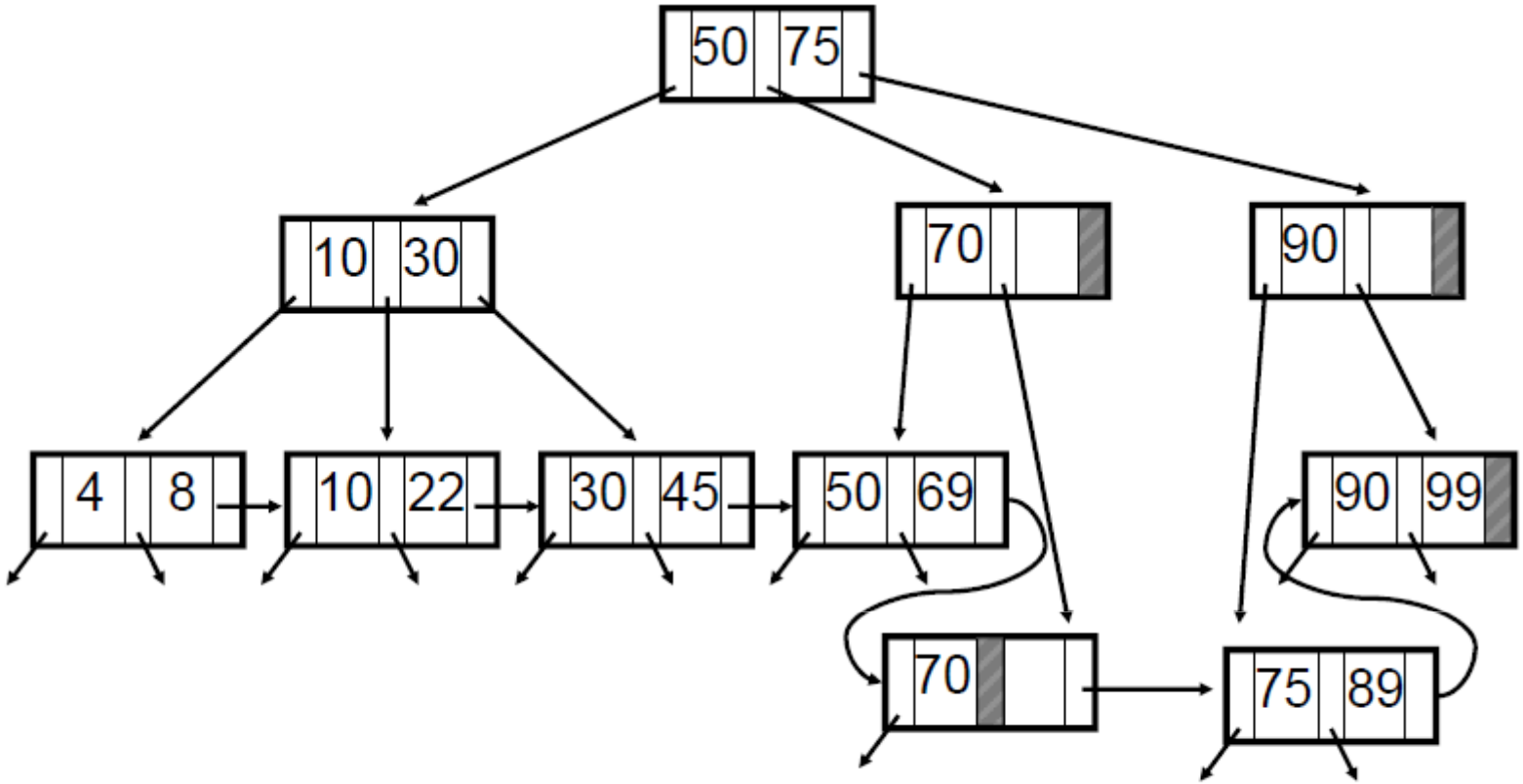
Insert 75



75 goes in 2nd last block.
Split block to handle overflow.
Promote 75. Note that 75 stays in a leaf!



Split parent block to handle overflow.
Promote 75. Note that 75 does not stay!



Insertion done!

- Since the inter-node connections are done by pointers, there is no assumption that in the B+-tree, the "logically" close blocks are "physically" close.
- The B+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches and modifications can be conducted efficiently.
- Example:
  - If a B+-tree node can store 300 key-pointer pairs at maximum, and on average is 69% full, then 208 (207+1) pointers/block.
  - Level 3 B+-tree can index 208^3 records = 8,998,912 records!

- By isolating the data records in the leaves, we also introduce additional implementation complexity because the leaf and interior nodes have different structures.
- This additional complexity is outweighed by the advantages of B+-trees which include:
  - Better sequential access ability.
  - Greater overall storage capacity for a given block size since the interior nodes can hold more pointers which requires less space.
  - Uniform data access times.

B+-trees versus Linear Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value for the key.
  - If range queries are common, B+-trees are preferred.