External Storage

So far …

- … we have been assuming that the data collections we have been manipulating were entirely stored in memory.

BIG Datasets

- … In practice, this is not always a reasonable assumption.
  - What if we were asked to search records of all Canadians for a particular Canadian (search key -> lastname)?
    - How many records?
    - Problem?

Record for a Canadian

```
class Canadian
{
   private:
       string lastName;
       string firstName;
       string middleName;
       string SIN;
   …
};
```

BIG Datasets

- What if we were asked to search records of all Canadians for a particular Canadian (search key -> lastname)?
  - How many records?
  - How much space?
    - 35 million * 20 bytes / string * 100 strings(?) = approx 70GB

- Some large databases, in which records are kept in files stored on external storage such as hard disk, cannot be read entirely into main memory.
  - We refer to such data as **disk-bound data**.

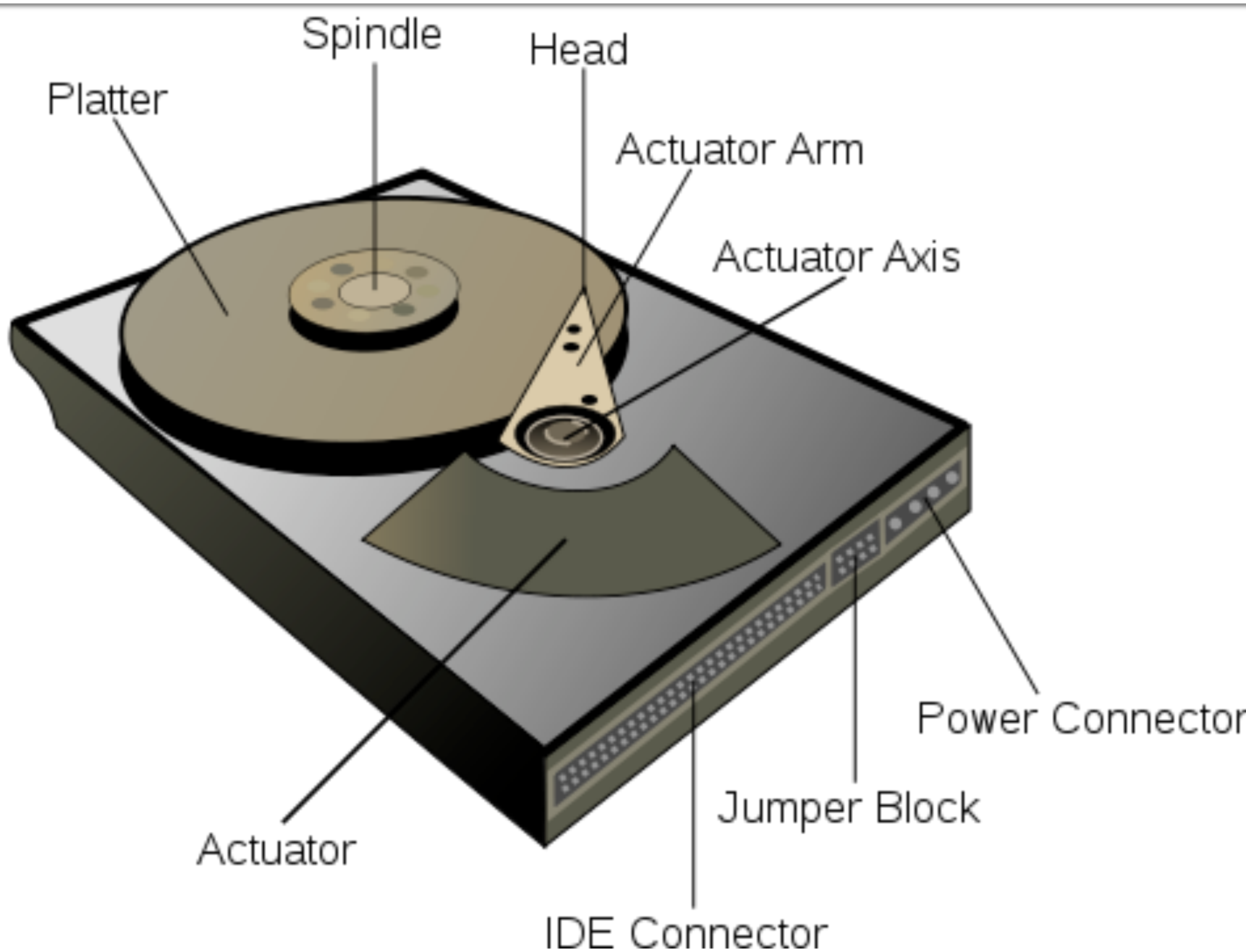
BIG Datasets Stay on Disk

- Hence, big datasets cannot fit in memory
  - Need to keep them on hard disk ("on disk")
  - Just read what we need at one time into memory
- Challenge: memory and disk access are not created equal

Disk-Bound Data

- Time efficiency of search for Canadian?
- Important factors:
  - Accessing data stored in a file kept on the hard disk is extremely slow compared to accessing data in memory -> order of milliseconds (10^{-3})
  - In contrast, accessing data in memory is fast -> order of nanoseconds (10^{-9})
  - Given the million-to-1 ratio of disk access time versus memory access time, to search our 30M records efficiently, we will need to devise a way that minimizes the number of disk accesses performed.

Why is Hard Disk Access Slow?



Your average PC hard drive

7,200 RPM
2.5-Inch Hard Disk Drives

TOSHIBA
Leading Innovation >>>

	MK8054GSY	MK1254GSY	MK1654GSY	MK2554GSY	MK3254GSY
Series Overview					
Drive Capacity	80GB[1]	120GB[1]	160GB[1]	250GB[1]	320GB[1]
Drive Interface			Serial ATA Revision 2.6 / ATA-8		
Number of Platters (disks)	1	1	1	2	2
Number of Data Heads	1	2	2	4	4
Transfer Rate to Host			3 Gb/sec		
Performance					
Track-to-track Seek			1ms		
Average Seek Time			10.5ms (Read), 12ms (Write)		
Rotational Speed			7,200 RPM		
Buffer Size			16MB		

Your not-so-average hard drive

15,000 RPM
3.5-Inch Enterprise Hard Disk Drives

TOSHIBA
Leading Innovation >>>

Series Overview

	MBA3073[2]	MBA3147[2]	MBA3300[2]
Drive Capacity	73.5GB[1]	147GB[1]	300GB[1]
Drive Interface	Dual Port SAS (RC), SCA-2 80Pin (NC), 68Pin Wide (NP), Dual Port FCAL (FC)		
Number of Platters (disks)	1	2	4
Number of Data Heads	2	4	8
RoHS Compliant	Yes		
Transfer Rate to Host	SAS: 3 Gb/sec, SCSI: 320 MB/sec, FCAL: 4 Gb/sec		

Performance

Track-to-track Seek	0.2ms (Read), 0.4ms (Write)
Average Seek Time	3.4ms (Read), 3.9ms (Write)
Rotational Speed	15,000 RPM
Average Latency	2ms
Buffer Size	SCSI: 8MB, SAS/FC: 16MB[3]

Aside

- Solid State Drives (SSD) can be much faster than spinning disks
  - And much more expensive

- However, still large latency compared to RAM

Sanity Check: Is that Slow?

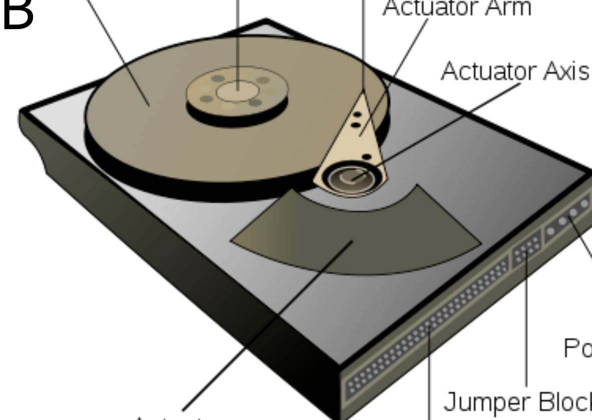
- What do those numbers mean?
  - Search in a red-black tree with 35 million records?
  - log n = 25
- If dataset fits in memory
  - Hundreds of nanoseconds per search
    - Can handle thousands of searches per second
- If dataset doesn't
  - Hundreds of milliseconds per search
    - Can handle only a few searches per second

Disk Access

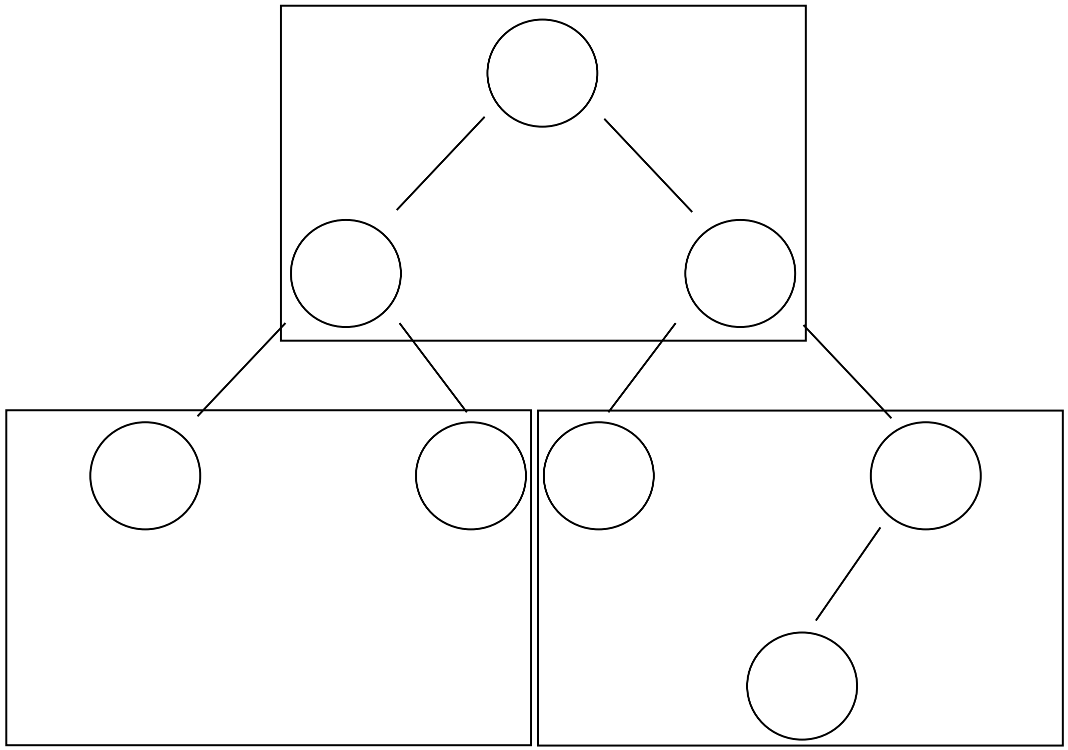
- Most time consuming operation when elements stored in external storage (disk)
  - Compared to 10 milliseconds, compute time is irrelevant
    - How many operations can a CPU do in 10 milliseconds?
    - @3GHz, a lot

Block

- Basic unit written to/read from external storage (disk)
  - If you're going to read don't just read 1 bit
  - Block size varies on each system
    - E.g. could be 8KB, could be 1MB



Example of blocks



Nodes of a binary tree can be located in different blocks on a disk.

File Access

- Random access file
  - Linear data collection (like an array)
- Sequential access file
  - Linear data collection (like a linked list)

Go to external_reading example

Back to our problem

- We have records for ~30M Canadians
  - Assume we can't store them in memory
    - So we keep them on disk
  - Now we want to search for one Canadian
- How should we do it?

Search - Take #1

- We could store our 30M Canadian records in a disk file which we access randomly
  - Assume each block on disk contains only 1 record
- Time efficiency to search for that Canadian?
  - If our records are not sorted: linear search -> O(n)
  - How fast is this in seconds?
    - 30M * milliseconds = seconds

Search - Take #2

- We could store our 30M Canadian records in a disk file which we access randomly.
  - Assume each block on disk contains only 1 record
  - Sort the records within the disk file (A. Aaronson at beginning of file, Z. Zygmund at end)
- Time efficiency to search for that Canadian
  - If our records are sorted: binary search -> $O(\log_2 n)$
  - How fast is this in seconds?
    - log (30M) * milliseconds = hundreds of milliseconds

Better, but still not so good

- Still need to do many disk accesses
  - Array is sorted, so log(n) disk accesses
- Disk accesses are **really** slow
  - Let's try to reduce them even further

Search - Take #3

- Main idea: split data into two files on disk
  - **DATA file**
    - Holds all information about all Canadians (our 70GB of data)
  - **INDEX file**
    - A smaller file that tells me where to find data about each Canadian
      - Remember the seekg command, random access to **DATA file**

Index File

- **INDEX file** should hold entries <*key, file byte*>
  - *key* is name of Canadian (or SIN)
  - *file byte* is offset into **DATA file** of where the record for this Canadian starts

…
<G Mori, 504>
<H Mori, 206>
<G Jensen, 7>
<R Henderson, 1083>
…

INDEX file

t	6	4	2	M	o	r	i
500	501	502	503	504	505	506	507

DATA file

Size of Index File

- Index file will be smaller than data file
- File size will be?
  - # Canadians * key size * file byte size
  - Much smaller than data file if record for each Canadian is large

Organization of Index File

- In order to find data about an individual, need to find his entry in index file
- So what should we do to the index file?
  - Sort it, e.g. into a tree data structure

…
<G Mori, 504>
<H Mori, 206>
<G Jensen, 7>
<R Henderson, 1083>
…

INDEX file

Search – Take #3 Flavour 1

- Let’s assume 30 million * key size * file byte size is not “too big”
  - I.e. it fits in memory
- Build a tree structure to store the contents of the index file in memory
  - Can build it / read it from disk when the program starts
  - Make it a balanced tree (e.g. red-black)

Search - Take #3 – Flavour 1

- Time efficiency to search for a record will be:

  $O(\log_2 n)$ comparisons (worst case)
  (for searching the index tree and finding the desired key, hence block #)

  + 1 disk access to fetch the block, in the data file, that contains the desired record (using block # found above)

- Time efficiency to search for a particular Canadian will be:

  about 25 comparisons + 1 disk access

- Just a few milliseconds

Isn't that Index File Pretty Big?

- Wait a minute, 30 million * key size * file byte size isn't *that* much smaller than the data file!!

- Hmm… we can use a similar trick on the index file

Search - Take #3 – Flavour 2

- If the entire tree stored in the **Index file** cannot be loaded into main memory:
- Each of its nodes, stored in a block, will contain as the "location of this node's left and right subtrees" the block # of the block in the Index file containing the root of the left/right subtree.
  - I.e. instead of a tree in memory with child pointers, a tree in the file with child block #s

Search - Take #3 – Flavour 2

- To perform a search:
  - the block containing the root of the tree is first accessed from the Index file
  - Tree search algorithm is performed on node contained in that block
  - the block # of the next tree node (block in Index file) is determined and the block containing that node is accessed
  - above two steps are repeated until the desired key is found or bottom of tree is reached (i.e., key not found)
  - if key found, the data file block containing the matching record is accessed using the block # of pair

Search - Take #3 – Flavour 2

- Time efficiency to search for a record will be:

  $O(\log_2 n)$ disk accesses (worst case)

  + 1 disk access to fetch the block, in the data file, that contains the desired record

- Time efficiency to search for a particular Canadian will be:

  about 25 disk accesses + 1 disk access

Not so good (again)

- Wait, 25 disk accesses sounds familiar
- That was the case for good old binary search on the data file

- Let's (again) try to do better

Search - Take #4

- How can we improve search performance?
- In order to minimize the number of disk accesses, we need to minimize the number of levels in our search tree, i.e., we need to flatten our tree.
- This can be achieved by increasing the number of records each node of our search tree can deal with.
- A **B Tree** can help …

More Trees (M-way, B)

M-Way Search Tree

- Definition**:** m-way search tree T is a tree of order *m*, in which each node can have at most *m* children
- Binary search trees generalize directly to m-way search trees
- Purpose of m-way search tree: Efficient search (hence retrieval)
- Other names given to m-way search trees are
  - m-ary search trees
  - multiway search trees
  - n-way search trees
  - n-ary search trees

M-Way Search Tree

- Definition**:** An m-way search tree T is an *m*-way tree (a tree of order m) such that:
  - T is either empty or
  - each non-leaf node of T has at most *m* children (subtrees):

    $$T_0, T_1, \ldots, T_{m-1}$$

    and $m - 1$ key values in ascending order:

    $$K_1 < K_2 < \ldots < K_{m-1}$$

  - for every key value V in subtree T_i: (**rules of construction**)

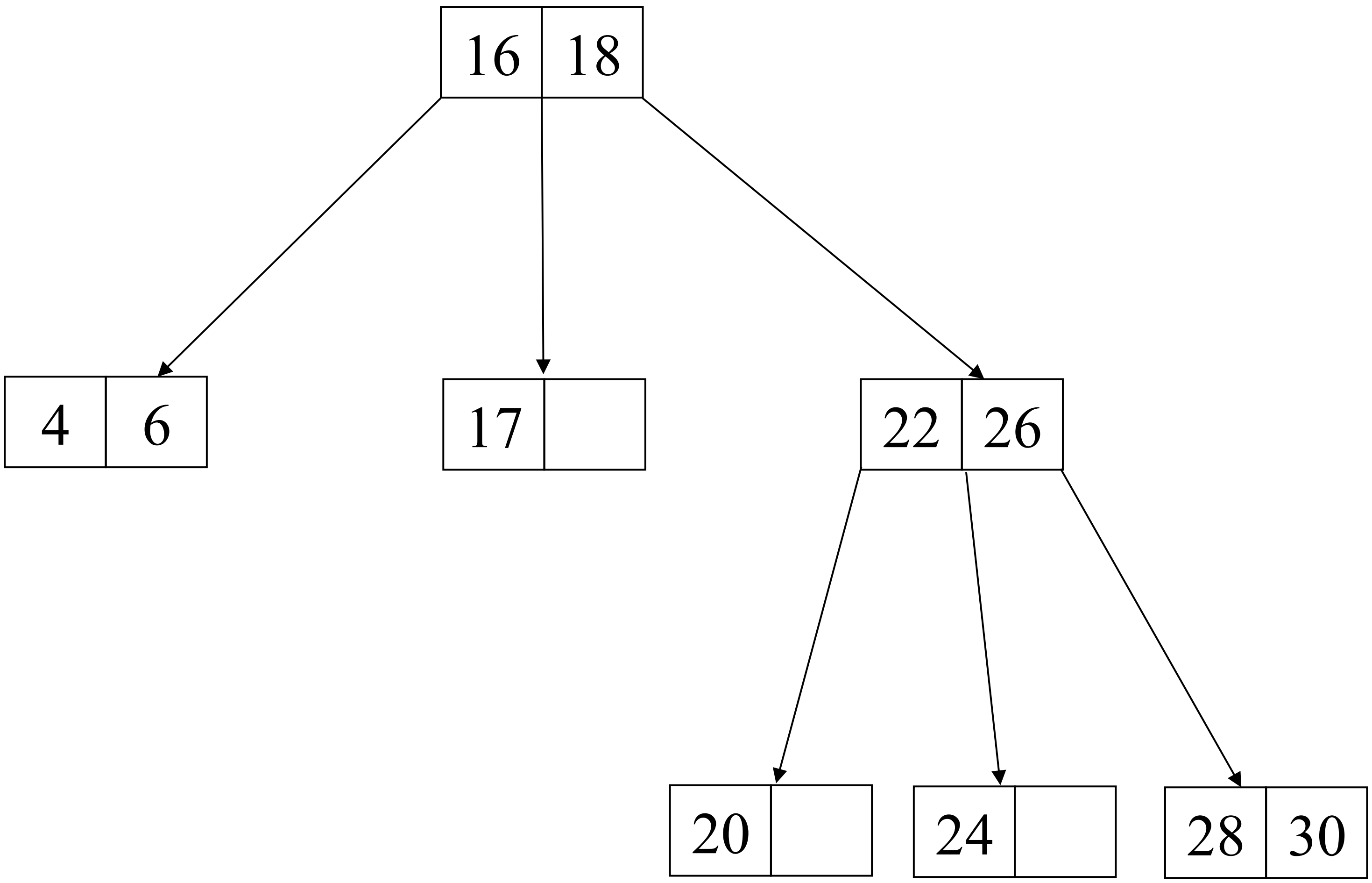
    $V < K_1,\quad i = 0$

    $K_i < V < K_{i+1},\quad 1 <= i <= m-2$

    $V > K_{m-1},\quad i = m-1$

  - every subtree T_i is also an *m*-way search tree

Example: The following is a 3-way search tree:

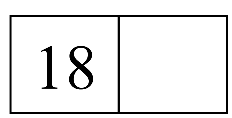


Insertion into m-way Search Tree

- Search for the spot where the new element is to be inserted (using its search key) until you reach an empty subtree
- Insert the new element into the parent of the empty subtree, if there is room in the node.
- Insert the new element into the subtree, if there is no room in its parent.

Insertion into an m-way Search Tree

- Let's construct the m-way search tree shown on the previous slide where m=3
- To do so, we shall insert the following search keys: 18, 16, 6, 22, 26, 4, 28, 24, 20, 30, 17
- Remember: the search keys (and their associated elements) are inserted in ascending sorting order in a node
- Let's begin by inserting 18:
  - since the m-way tree is empty, we create the first node i.e., the root and insert 18



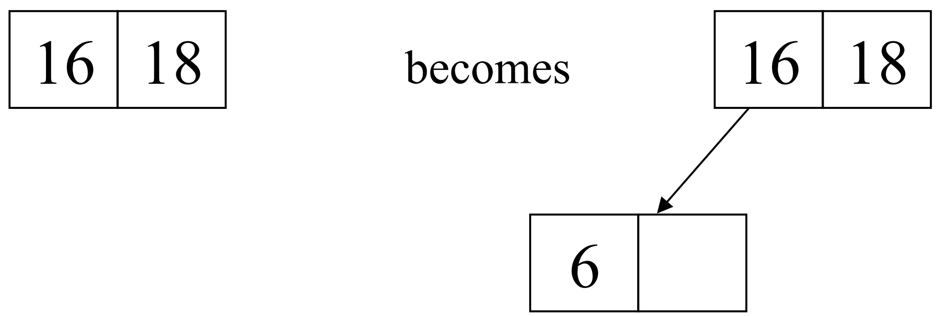
Insertion into an m-way Search Tree

- Let’s insert 16:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the parent of the empty subtree, in the proper sorted order, if there is room in the parent node.

18		becomes	16	18

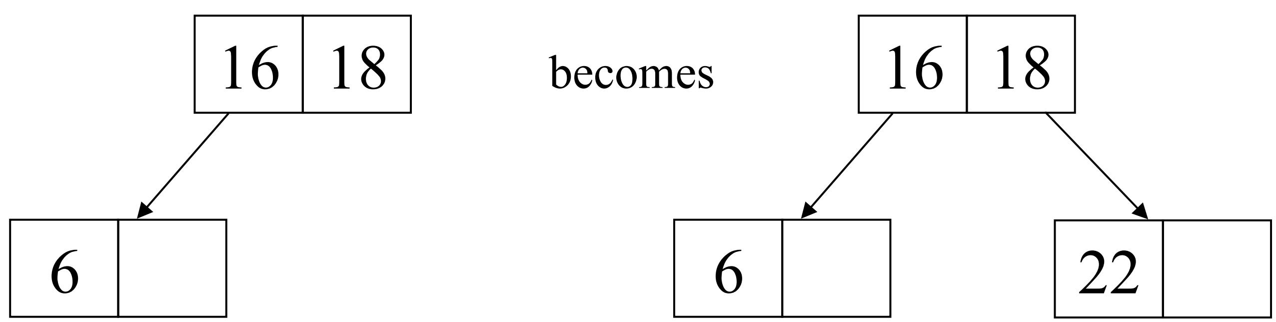
Insertion into an m-way Search Tree

- Let's insert 6:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



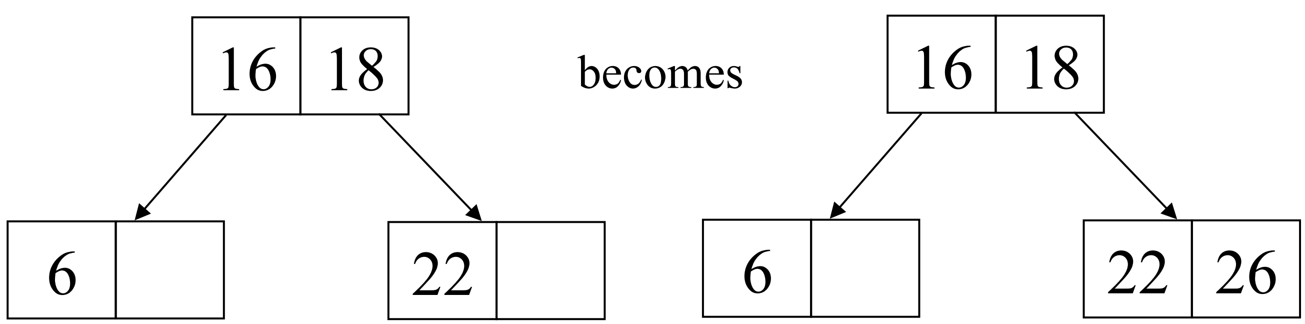
Insertion into an m-way Search Tree

- Let's insert 22:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



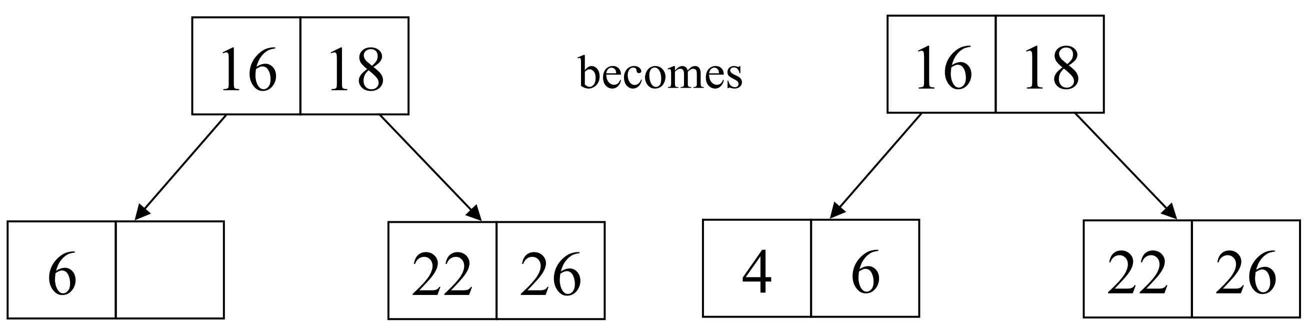
Insertion into an m-way Search Tree

- Let's insert 26:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the parent of the empty subtree, in the proper sorted order, if there is room in the parent node.



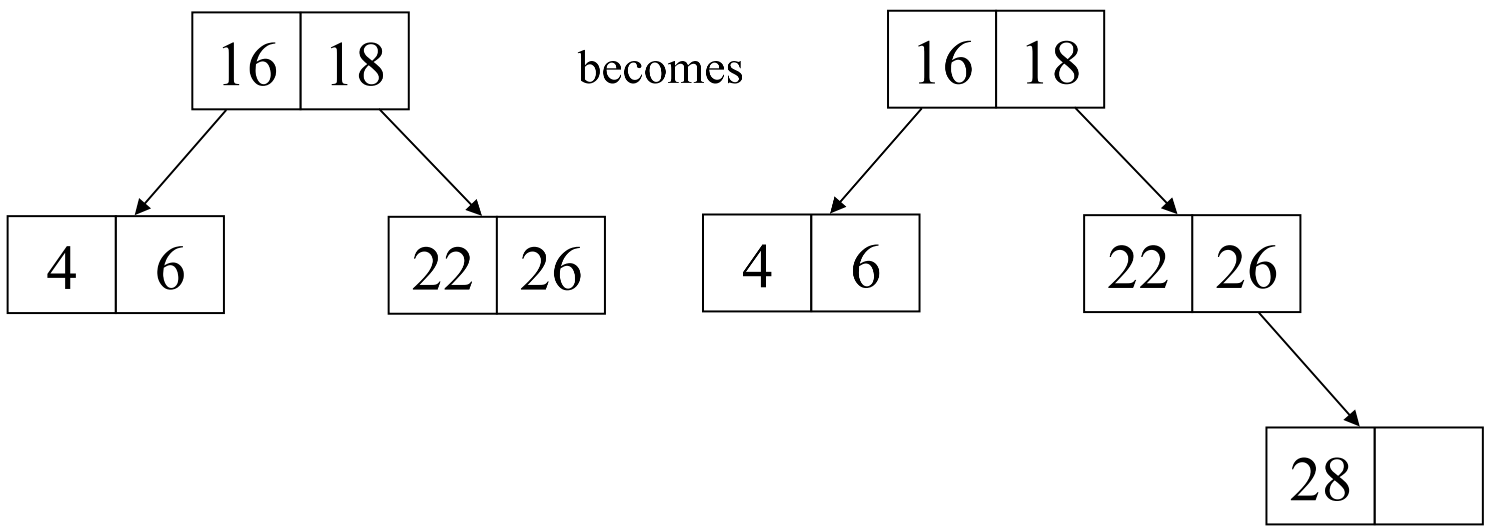
Insertion into an m-way Search Tree

- Let's insert 4:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the parent of the empty subtree, in the proper sorted order, if there is room in the parent node.



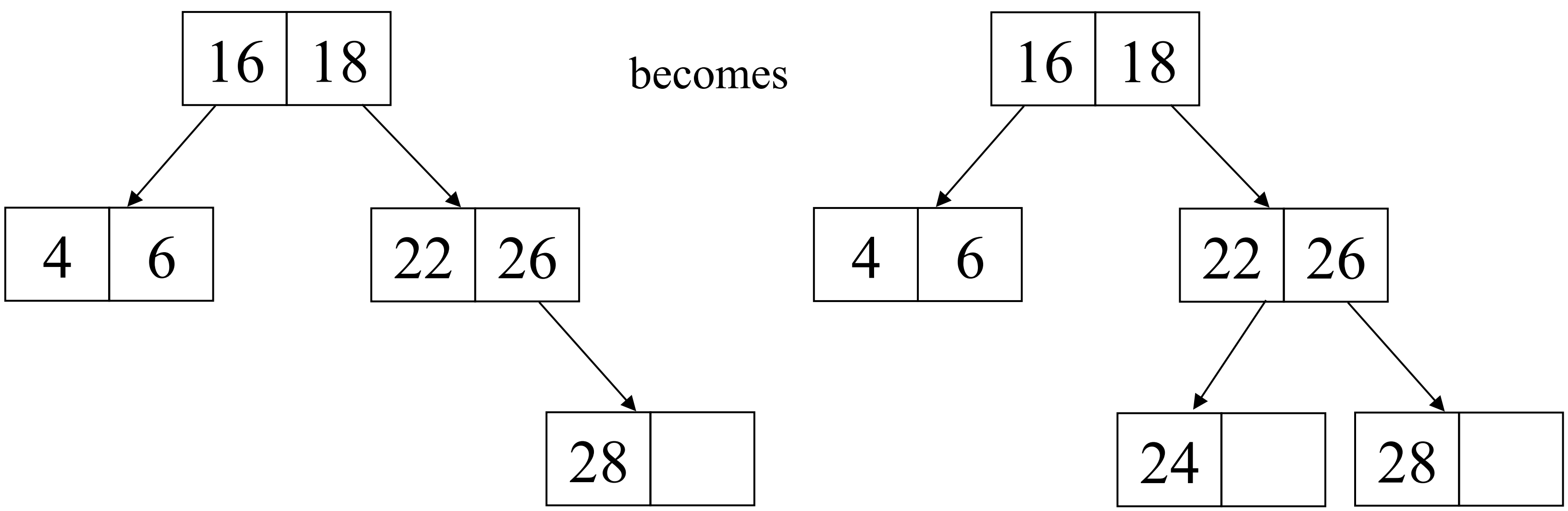
Insertion into an m-way Search Tree

- Let's insert 28:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



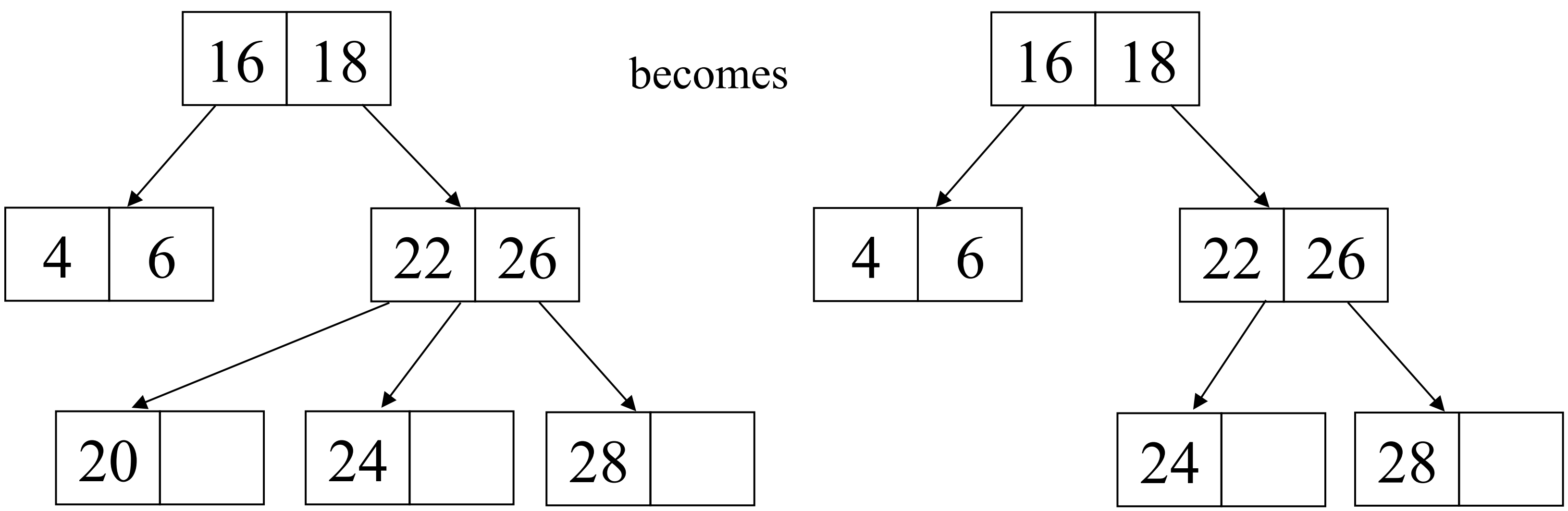
Insertion into an m-way Search Tree

- Let's insert 24:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



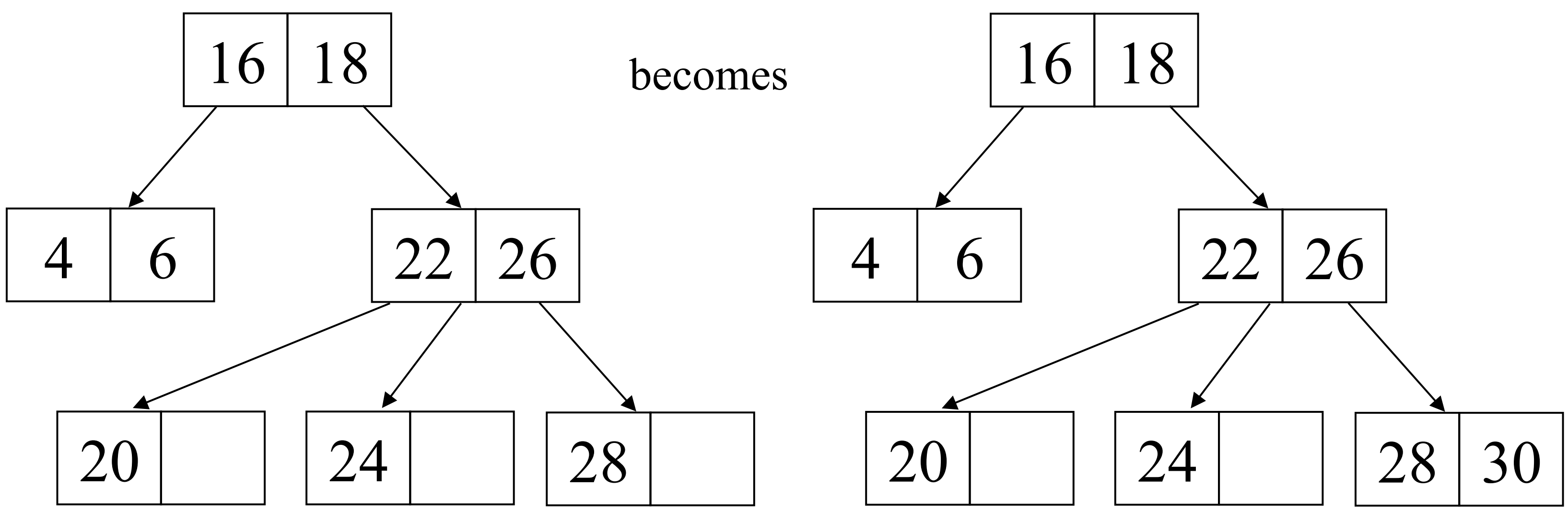
Insertion into an m-way Search Tree

- Let's insert 20:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



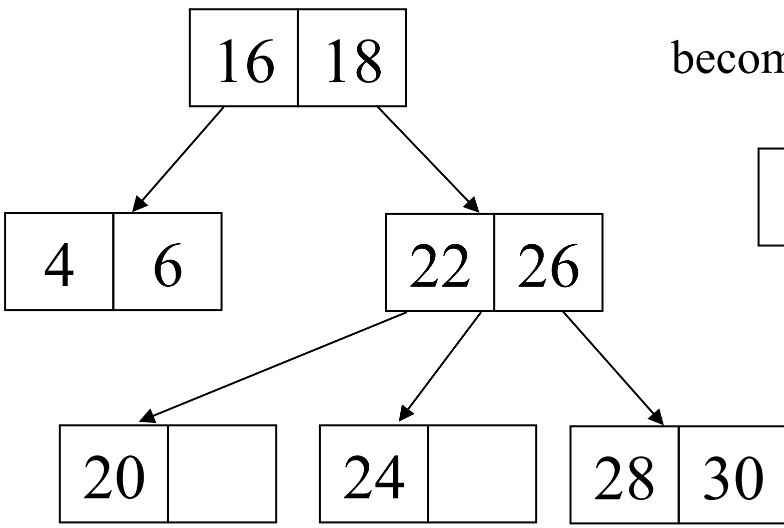
Insertion into an m-way Search Tree

- Let's insert 30:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the parent of the empty subtree, in the proper sorted order, if there is room in the parent node.

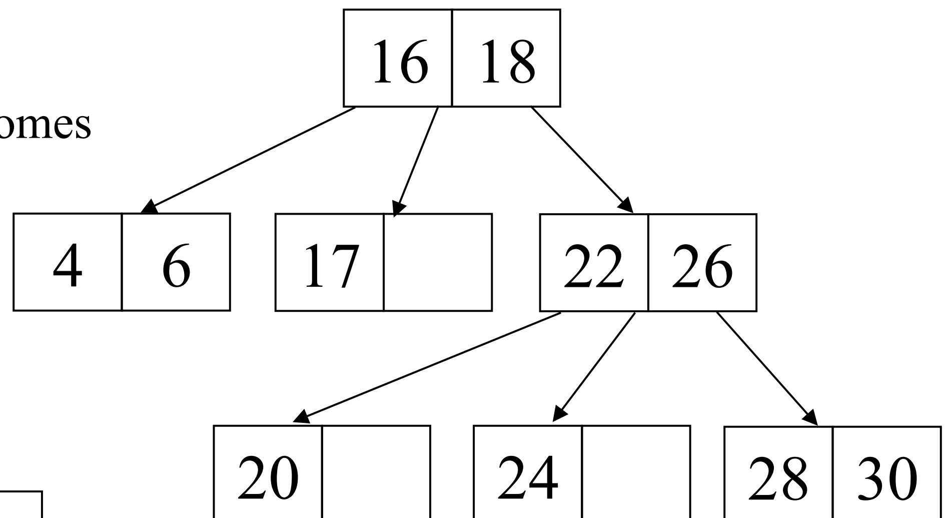


Insertion into an m-way Search Tree

- Let's insert 17:
  - Search for the spot where the new element is to be inserted using its search key until you reach an empty subtree
  - Insert the new element into the empty subtree, if there is no room in its parent node.



becomes



B Trees

B Tree

- Definition**:** A **B Tree** is a data collection that organizes its blocks (**B**) into an *m*-way search tree, and in addition
  - the root of a **B Tree** has at least 2 children (unless it is a leaf node)
  - and its other non-leaf nodes have at least $\lceil m / 2 \rceil$ children.

B Tree

- A **B Tree** is built from the leaves up, rather than from the root down, and so all leaf nodes in a **B Tree** are on the same level.
  - Hence, B Tree is a balanced m-way tree, just as Red-black trees are balanced binary search trees

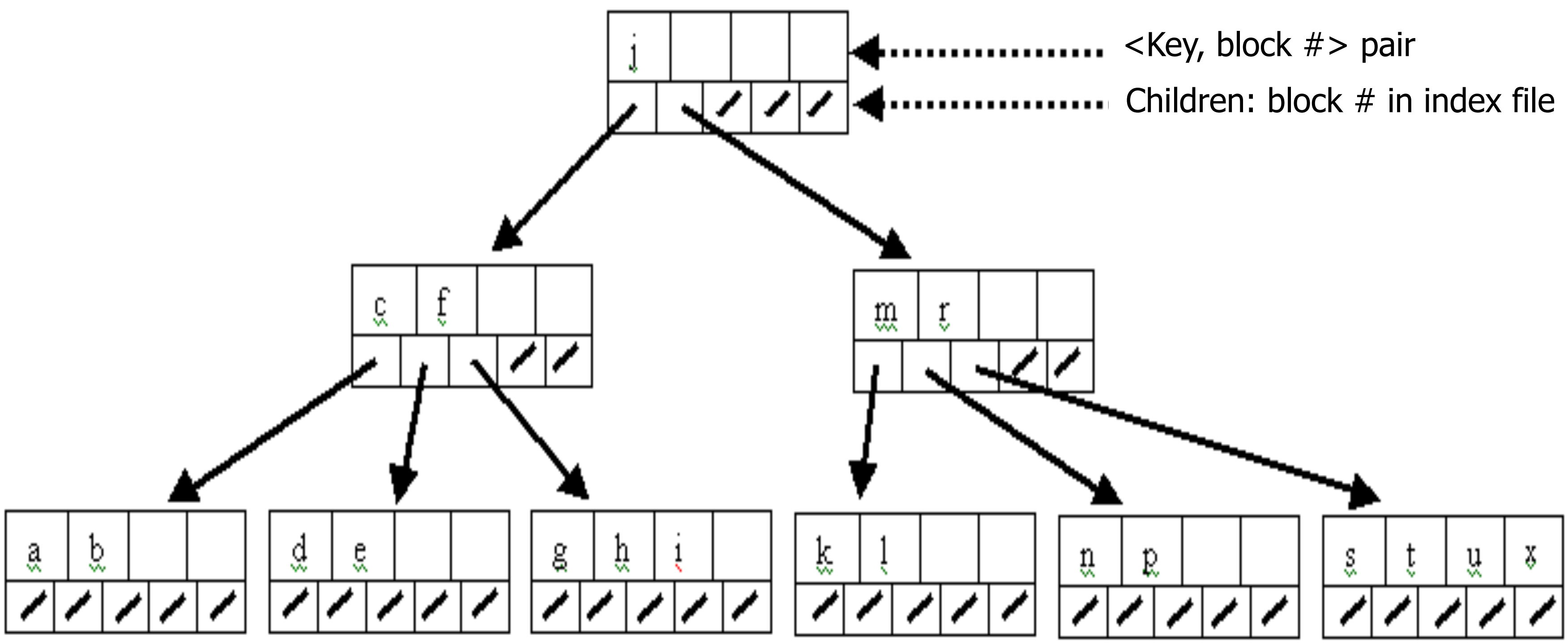
B-Tree Structure

- Each block contains a tree node
- *m-1* <key, data file block #> pairs in a node + index file block # as links to children/subtrees

Example of B Tree

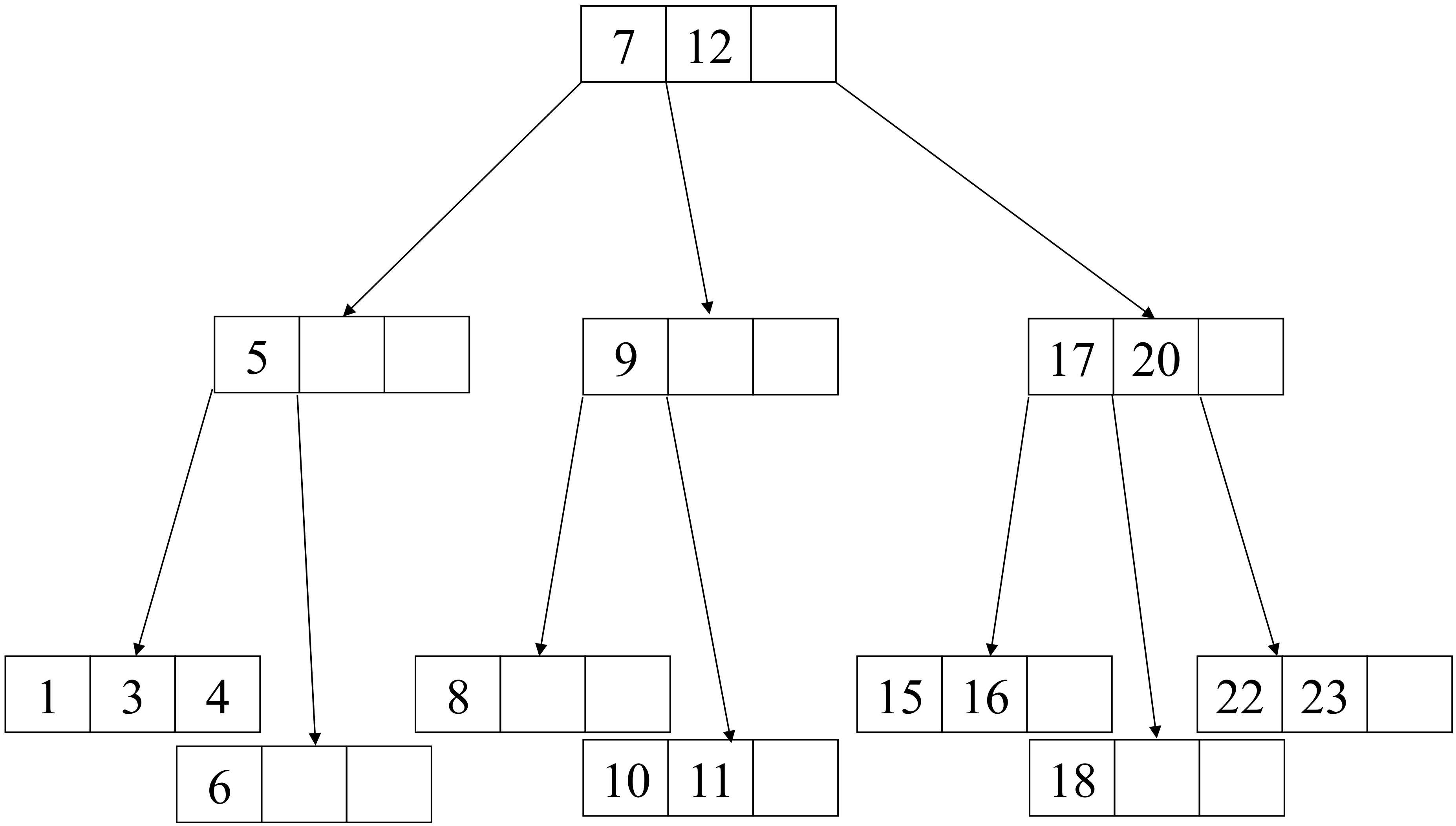
B-Tree of order 5 (m = 5) in which every node (except the root and the leaves) has

- at least ⌈ 5 / 2 ⌉ = 3 children, and
- no more than 5 children



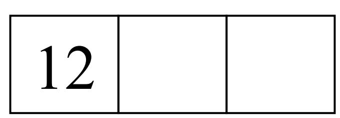
Example: The following is a B Tree with m=4

(such B Trees are called 2-3-4 search trees)



Insertion into a B Tree

- Let's construct the B Tree shown on the previous slide where m=4
  - Actually, that B Tree is an example of a 2-3-4 search tree
- To do so, we shall insert the following search keys: 12, 1, 7, 23, 20, 6, 18, 5, 4, 22, 10, 15, 8, 3 , 9, 17, 11, 16
- Remember: the search keys (and their associated elements) are inserted in ascending sorting order in a node
- Let's begin by inserting 12:
  - since the m-way tree is empty, we create the first node i.e., the root and insert 12



Insertion into a B Tree

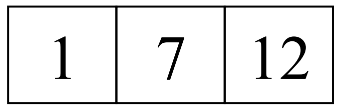
- Insert 1:
  - compare each key found in the root with the key 1 and since 1 < 12, move 12 over, then insert 1

1	12	

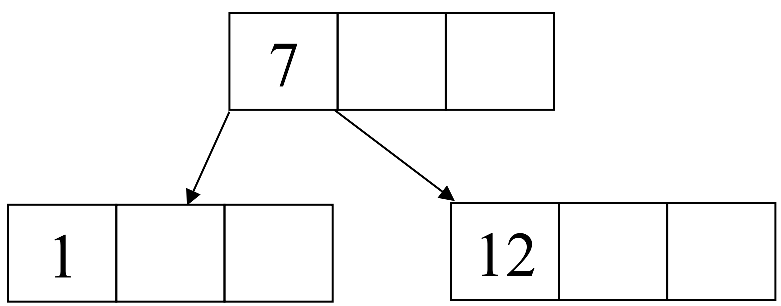
- Insert 7:
  - compare each key found in the root with the key 7 and since 1 < 7 < 12, move 12 over, then insert 7

1	7	12

Insertion into a B Tree

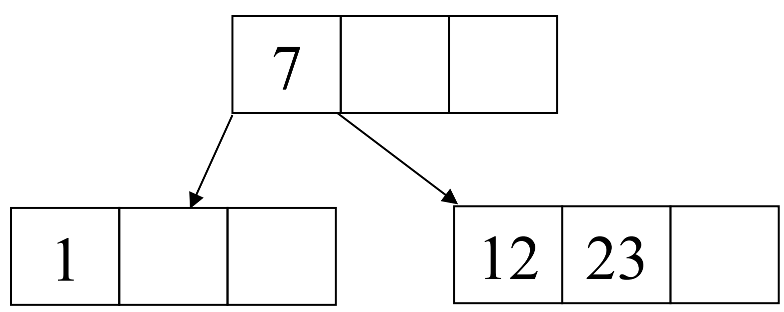
- Insert 23: 

  - starting at the root, right away we encounter a full node so we split it as follows:
    - create a new node (parent) and move the middle key into it
    - create a sibling and move the key > 7 into it
    - link the subtrees to the newly formed parent node



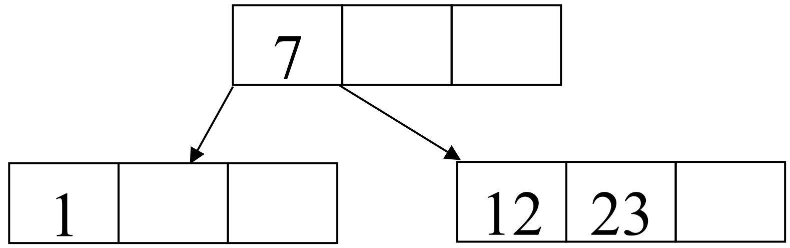
Insertion into a B Tree

- Insert 23 (cont'd):
  - starting at the root, since 7 < 23, 23 is inserted into its right subtree
  - considering the root of its right subtree, since its only key 12 < 23, insert 23 after 12

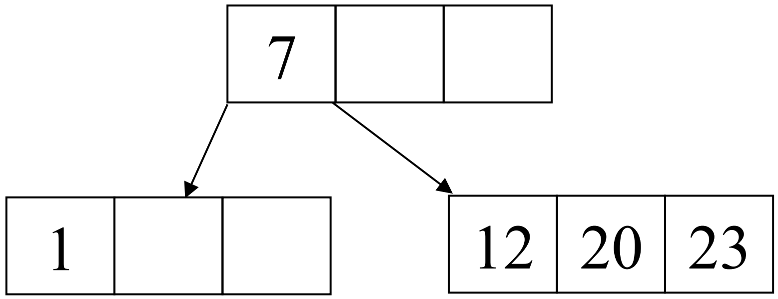


Insertion into a B Tree

- Insert 20:

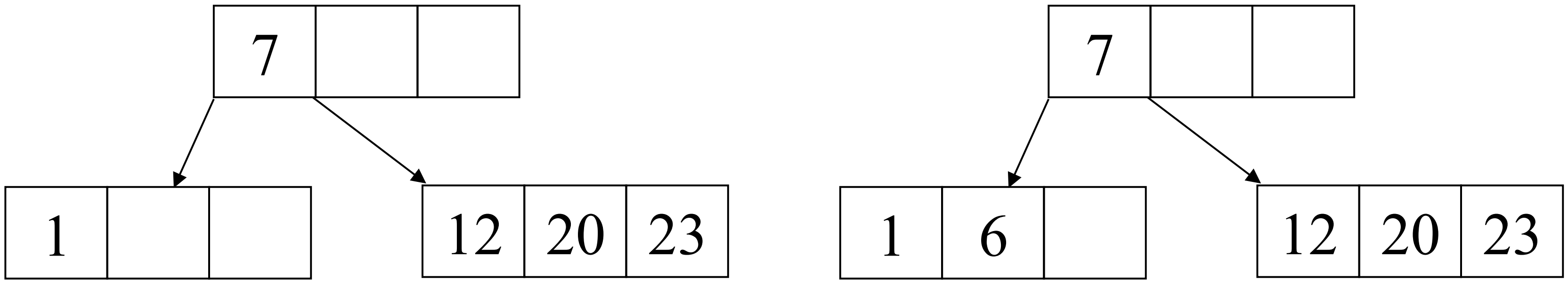


  - starting at the root, since 7 < 20, 20 is inserted into its right subtree
  - moving on to the root of its right subtree, since 12 < 20 < 23, move 23 over, then insert 20



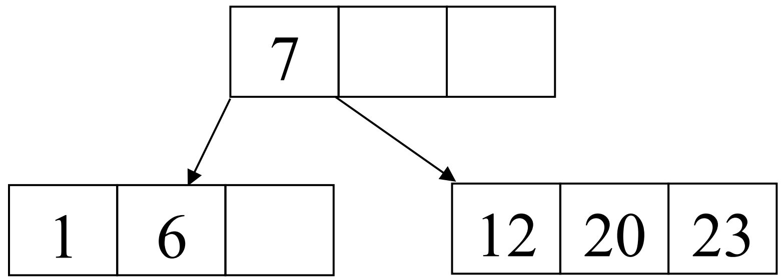
Insertion into a B Tree

- Let's pick up the pace now…
- Insert 6:

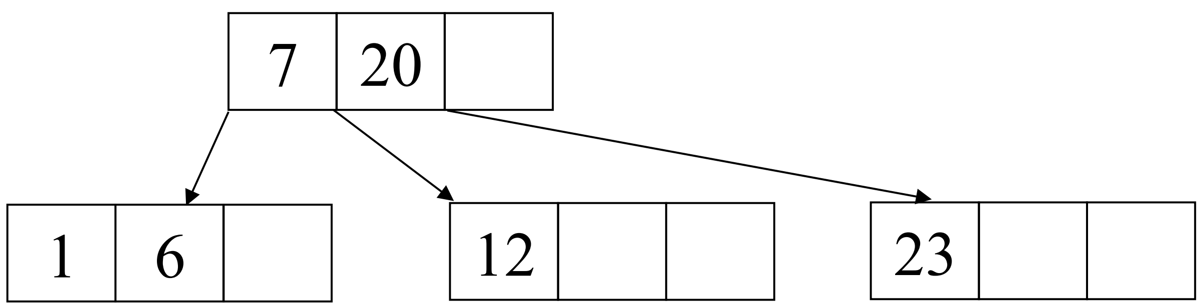


Insertion into a B Tree

- Insert 18:

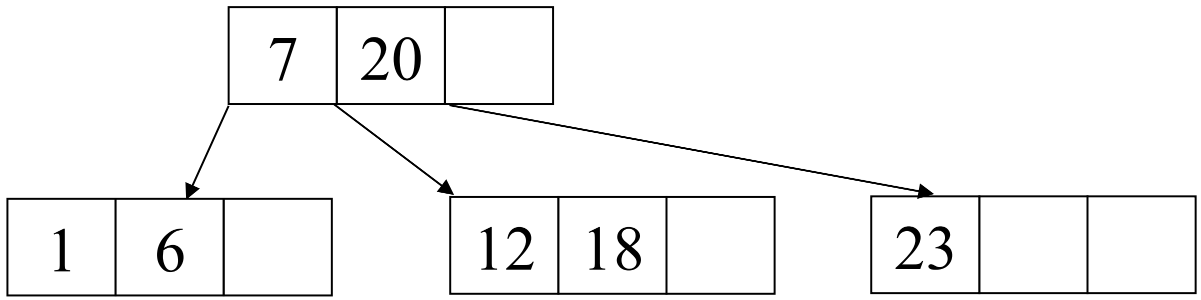


  - on our way to insert 18 we encounter a full node so we split it first:
    - we move its middle key into the parent node
    - we create a sibling and move the key > 20 into it
    - link the newly formed rightmost subtree to the parent node

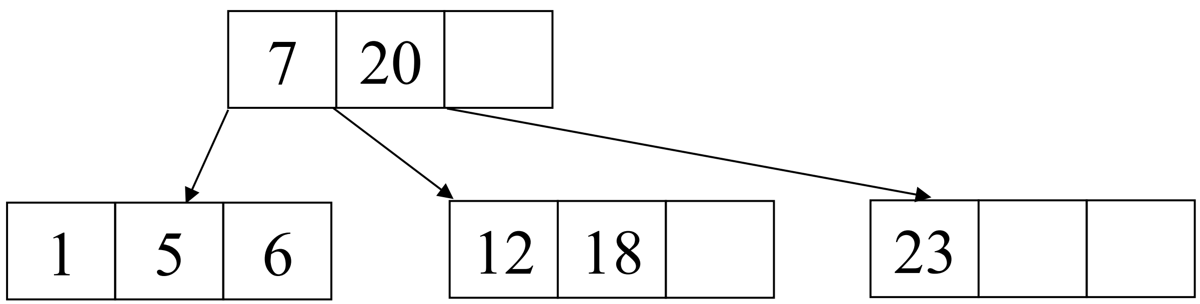


Insertion into a B Tree

- Insert 18 (cont'd):

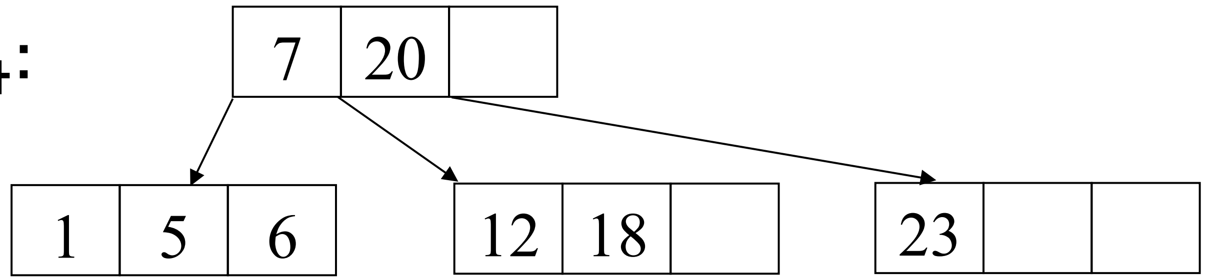


- Insert 5:

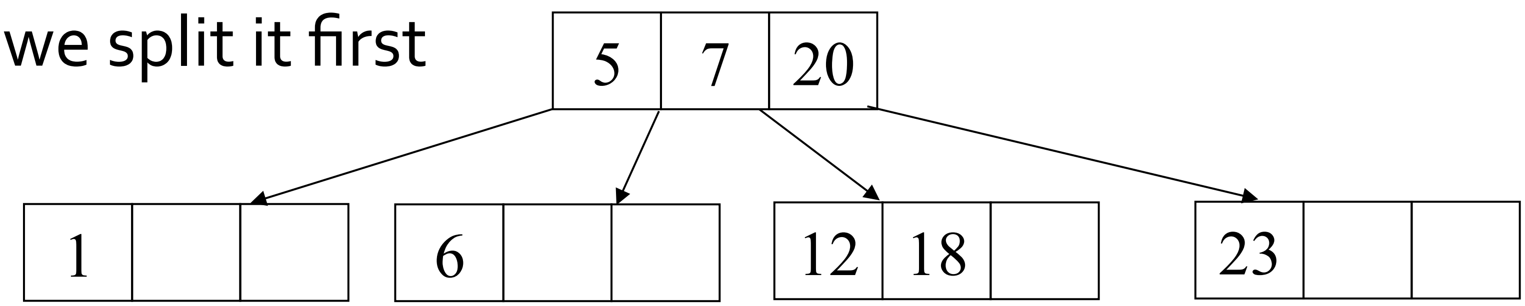


Insertion into a B Tree

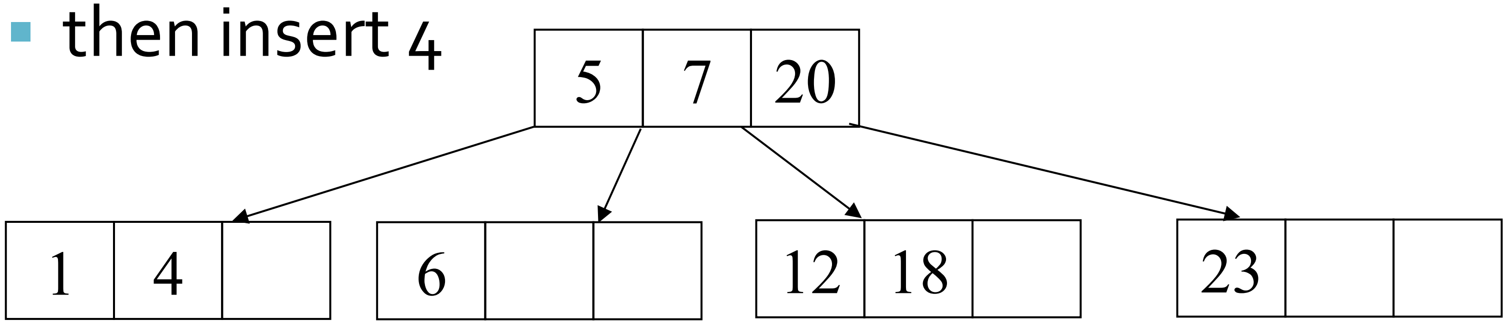
- Insert 4:



  - on our way to insert 4 we encounter a full node, so we split it first

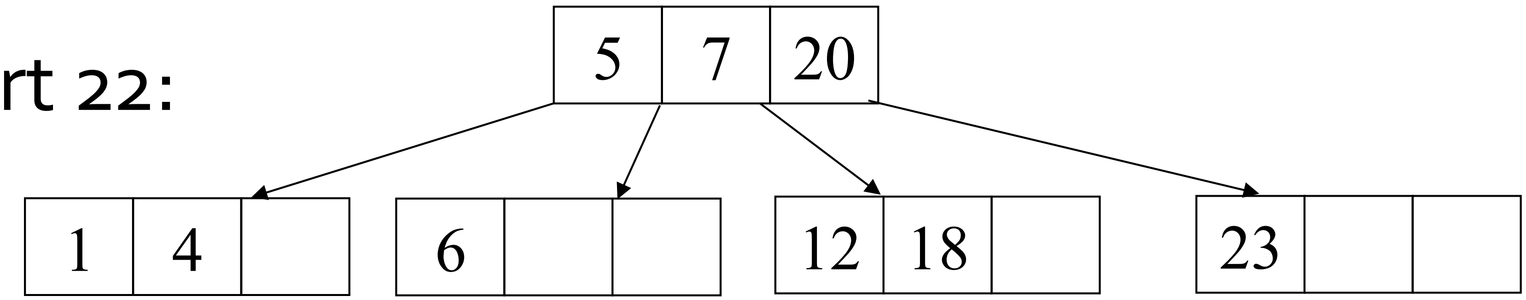


  - then insert 4

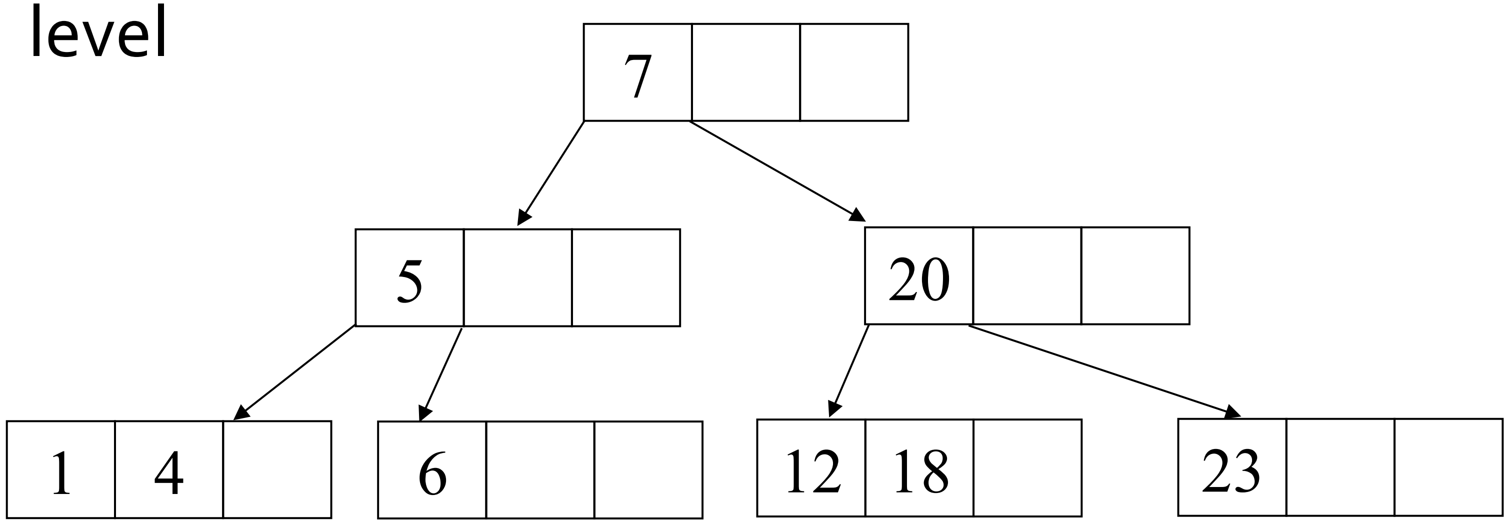


Insertion into a B Tree

- Insert 22:

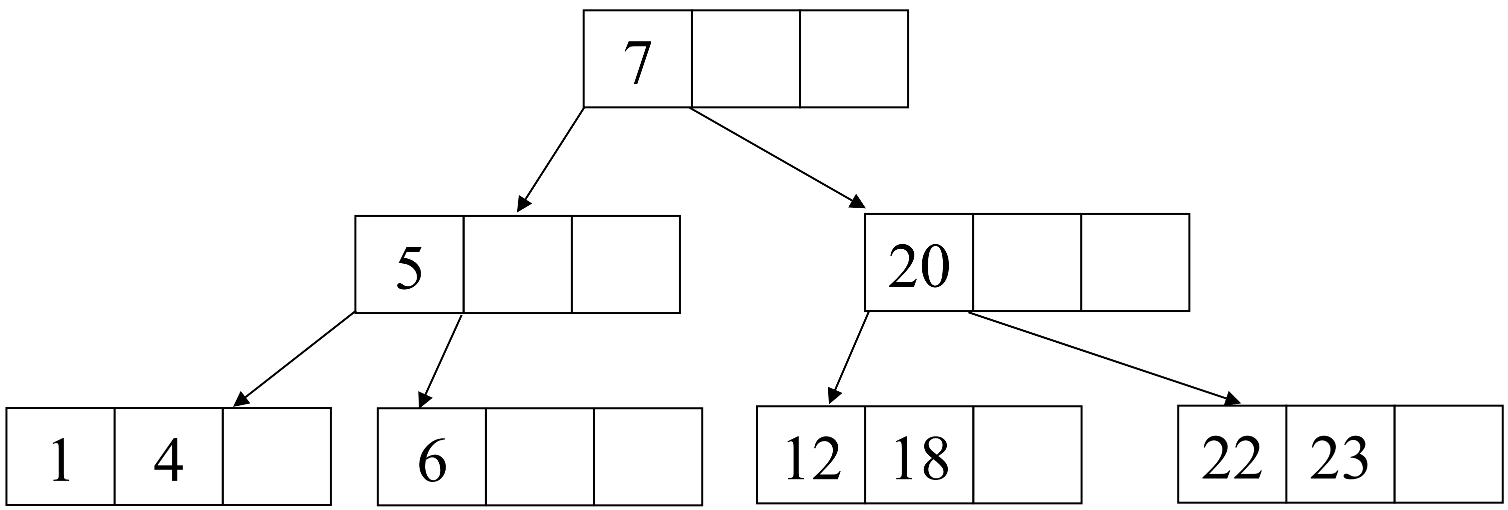


  - on our way to insert 22, right away we encounter a full node so we split it first hence creating another level



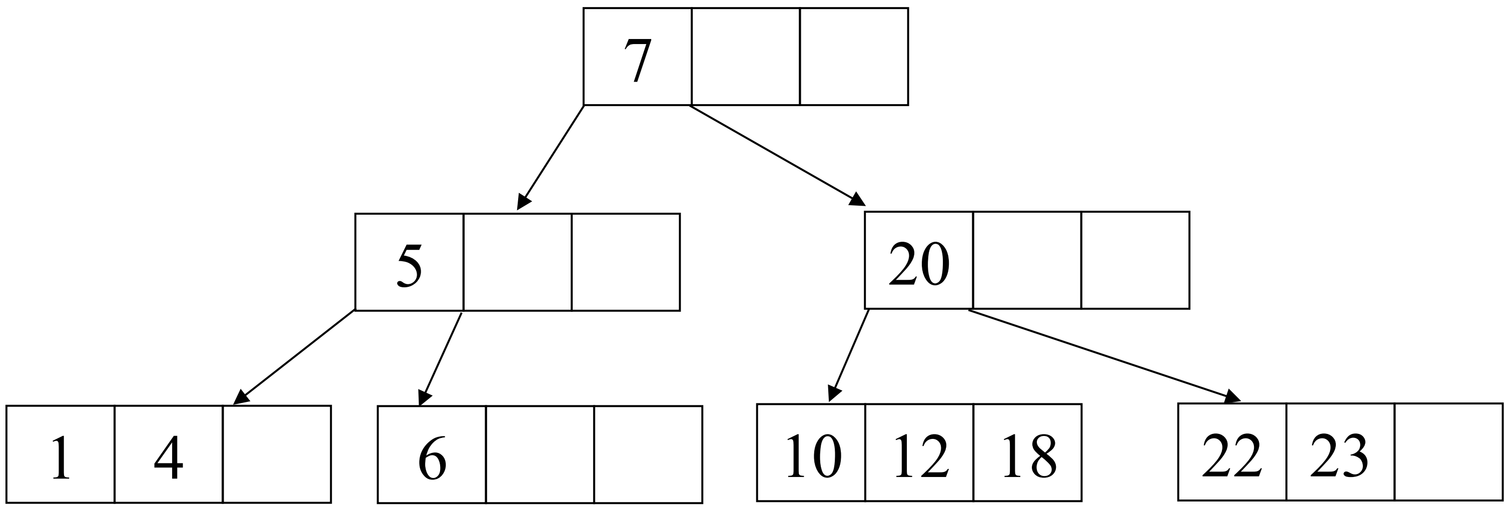
Insertion into a B Tree

- Insert 22 (cont'd):
  - then insert 22:



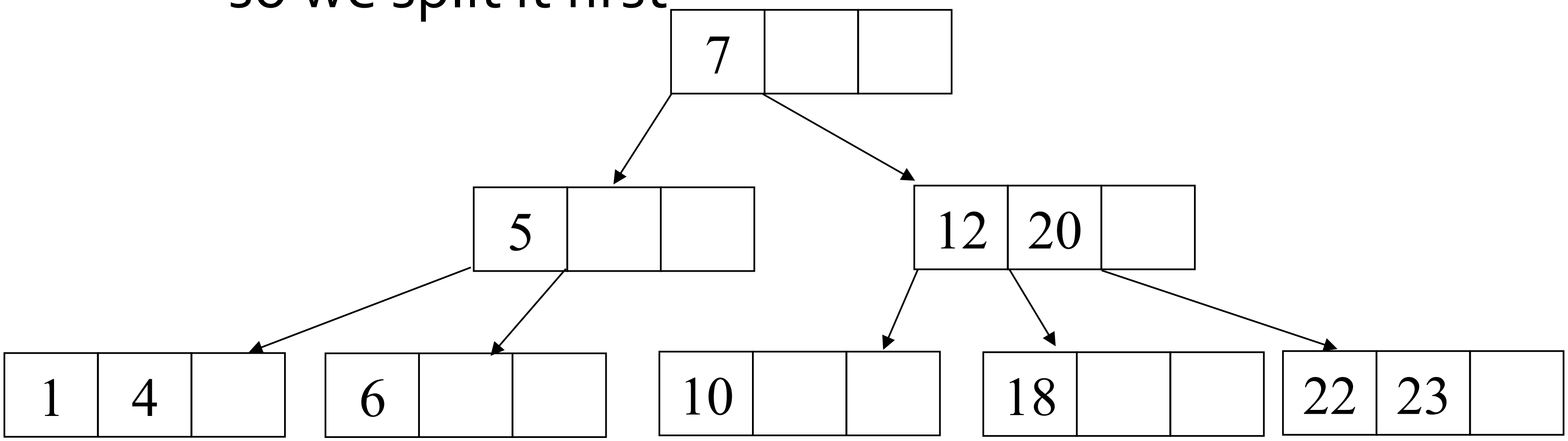
Insertion into a B Tree

- Insert 10:



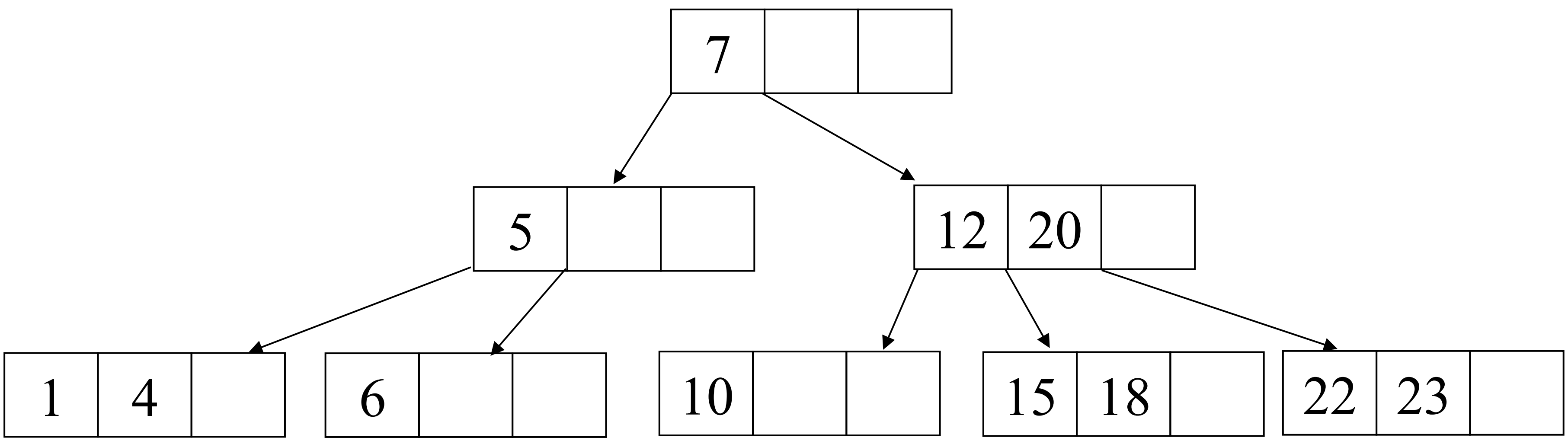
Insertion into a B Tree

- Insert 15:
  - on our way to insert 15, we encounter a full node, so we split it first



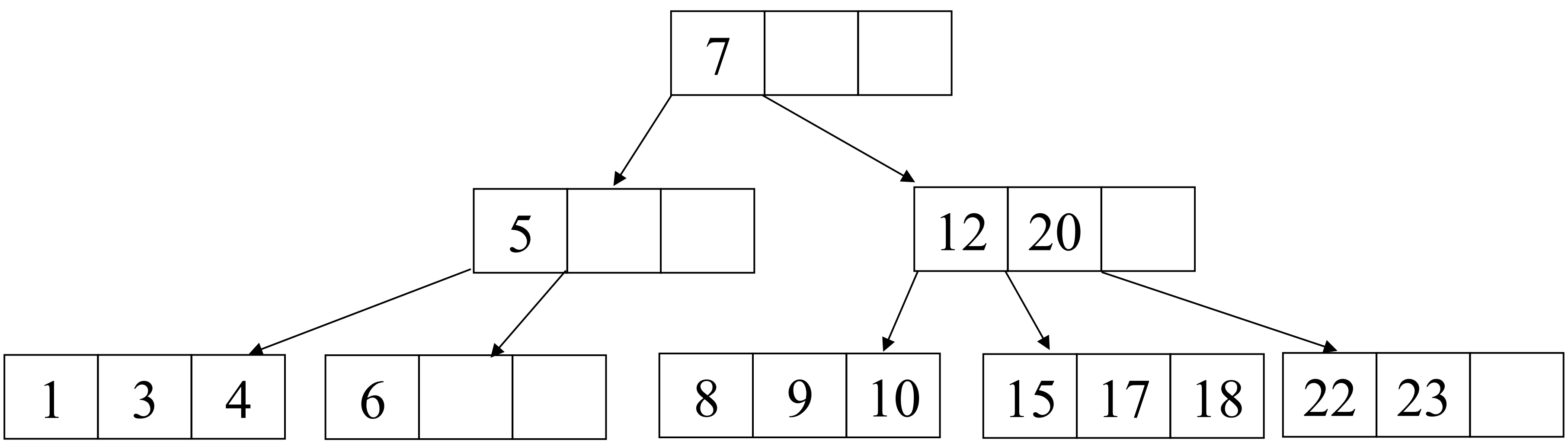
Insertion into a B Tree

- Insert 15:
  - then insert 15:



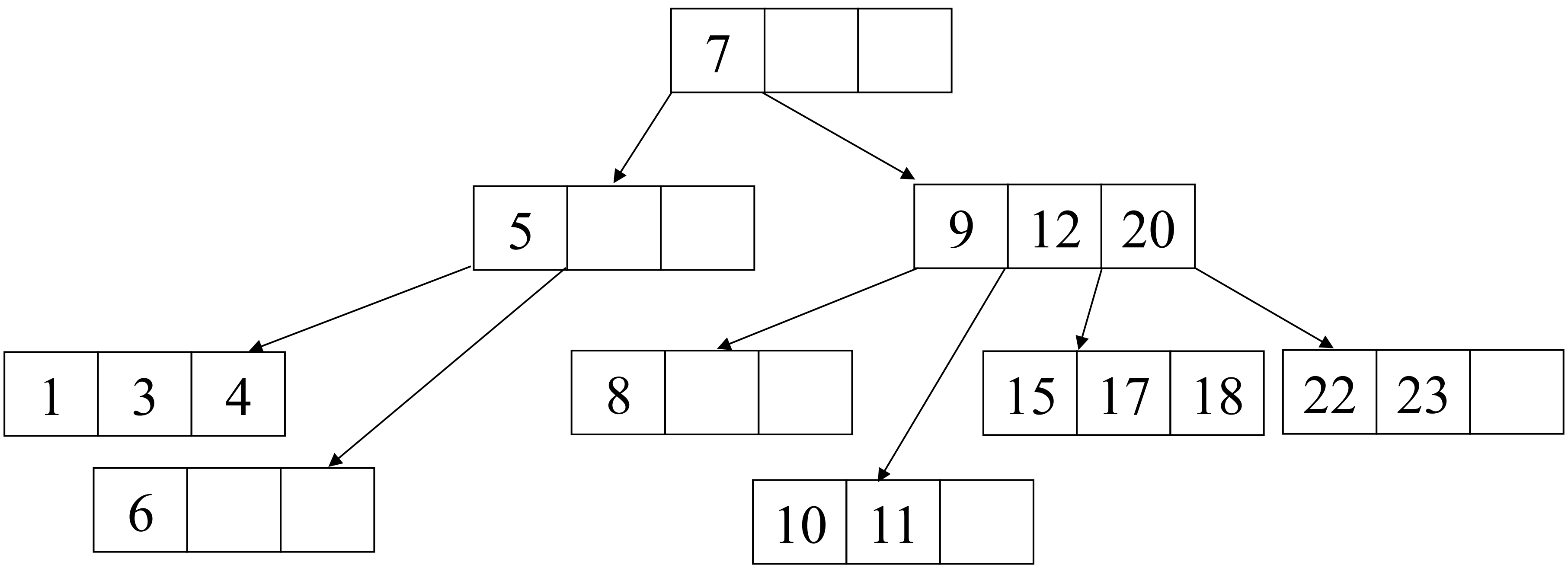
Insertion into a B Tree

- Insert 8, 3, 9 and 17:



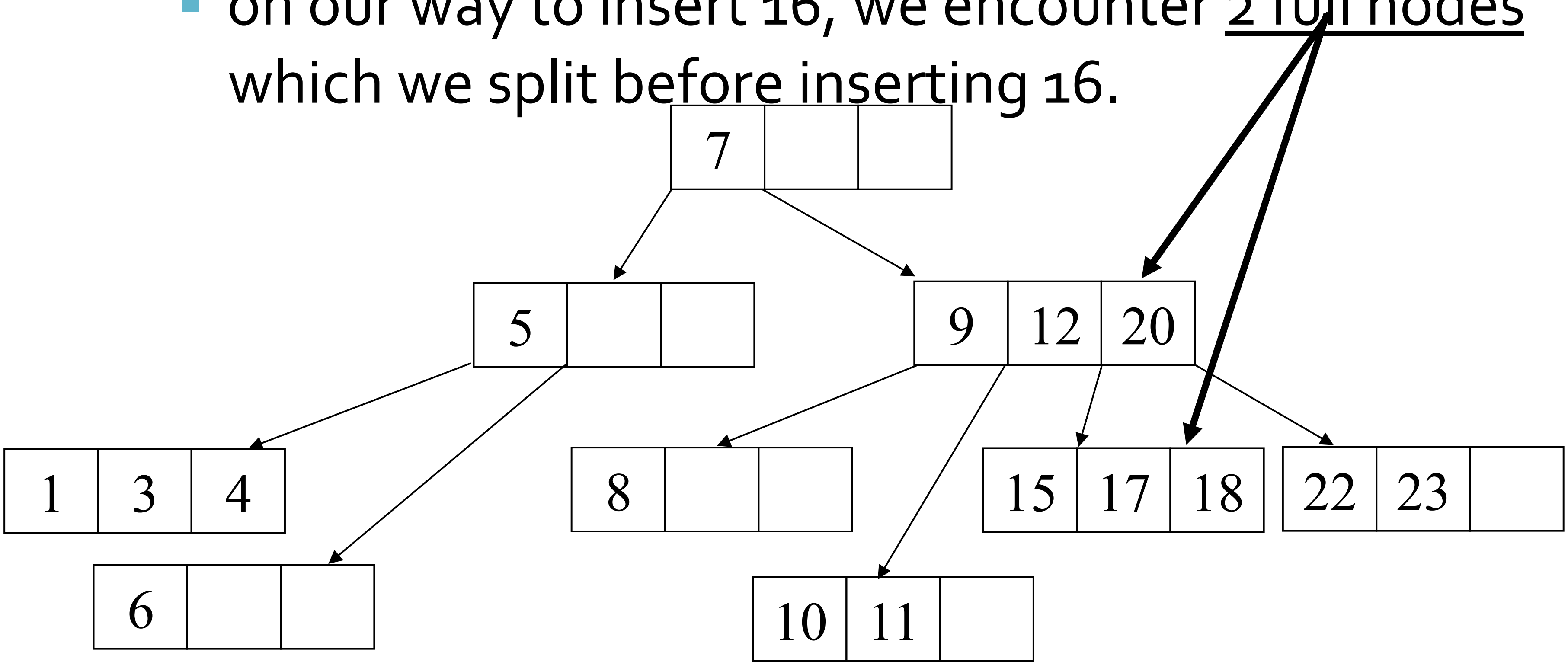
Insertion into a B Tree

- Insert 11:
  - on our way to insert 11, we encounter a full node, so we split it first, then we insert 11



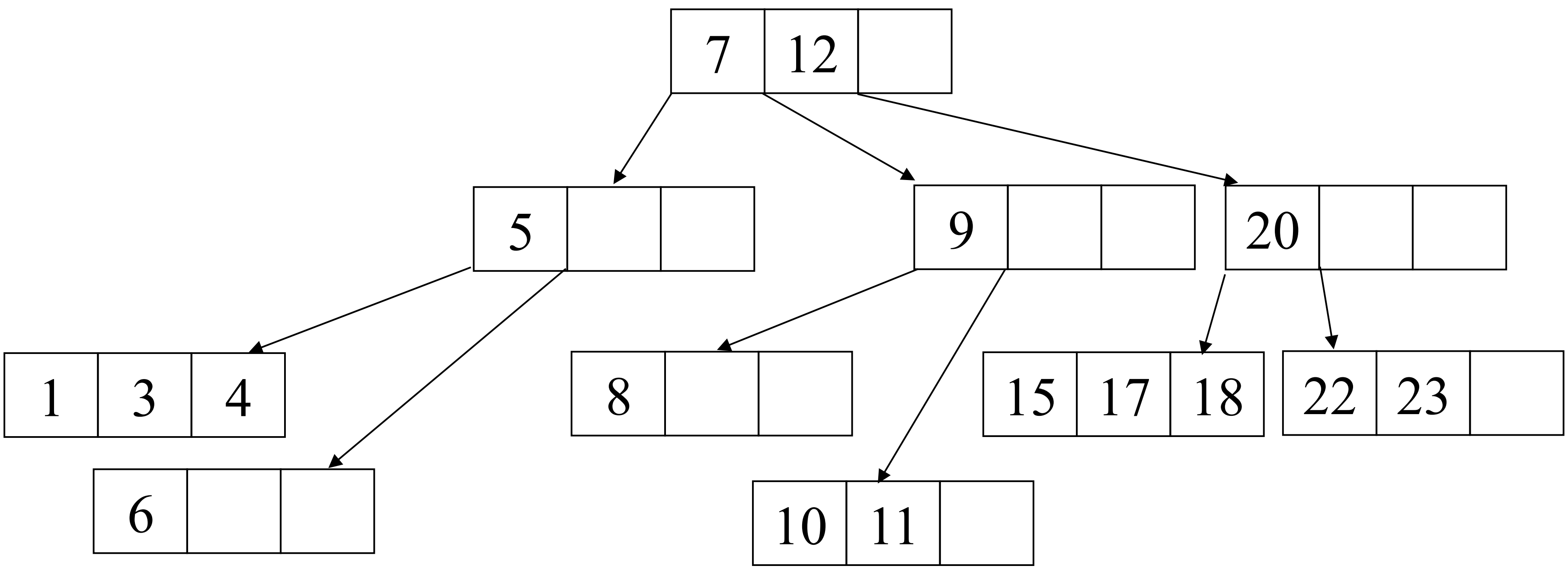
Insertion into a B Tree

- And finally, we insert 16:
  - on our way to insert 16, we encounter 2 full nodes which we split before inserting 16.



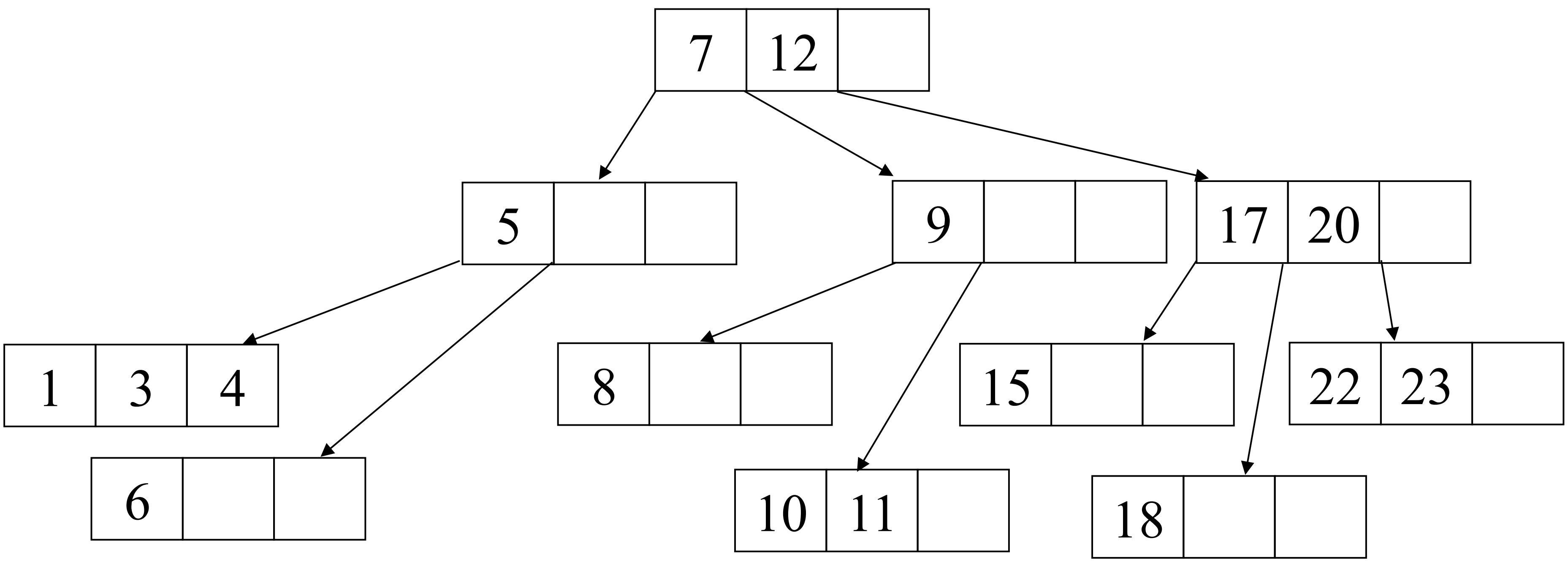
Insertion into a B Tree

- Insert 16 (cont'd):



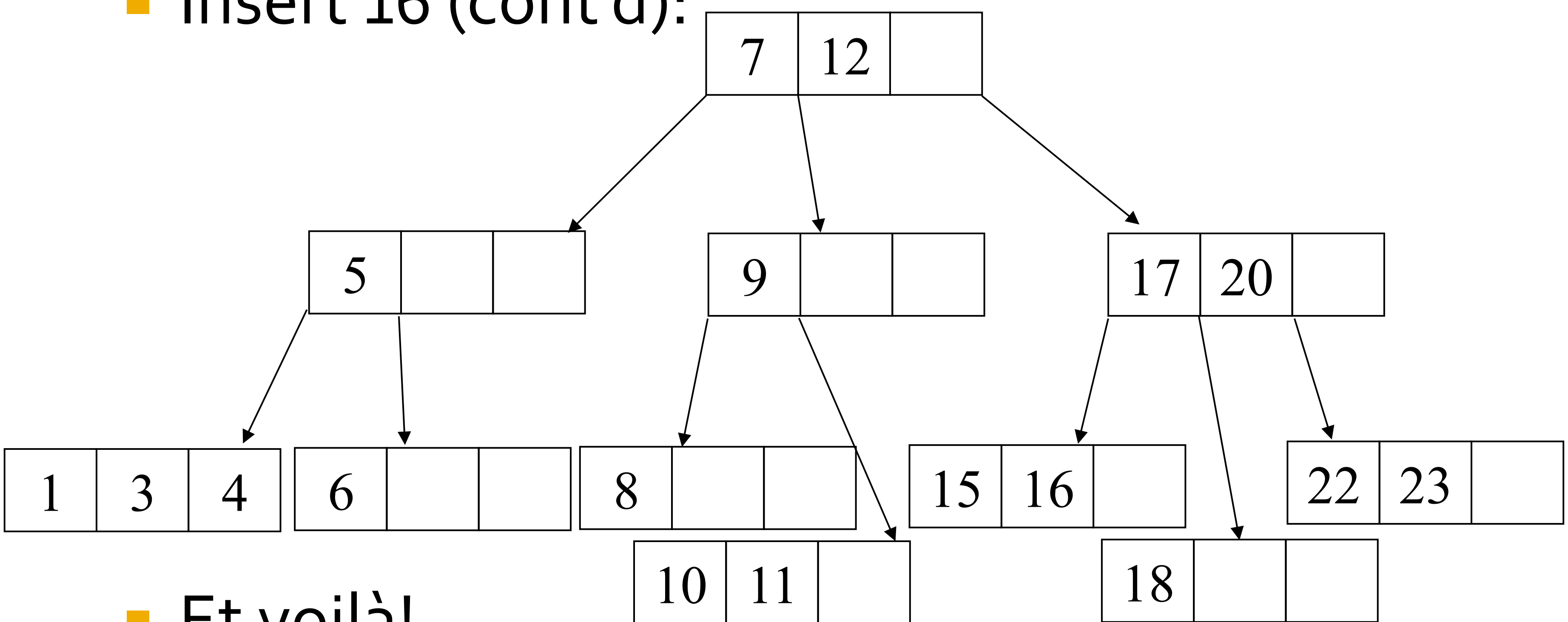
Insertion into a B Tree

- Insert 16 (cont'd):



Insertion into a B Tree

- Insert 16 (cont'd):



- Et voilà!

Phew

- Ok, don't worry, that won't be on the exam
- Summary: another balanced tree
  - But it's not binary, it's an m-way tree
  - Will have far fewer levels in it than a binary tree
  - Has similar balancing properties to red-black
    - Number of levels similar to best case log(n)

B Tree Search Algorithm

- Access block from index file containing the root
- Linearly search for key in accessed block
  - If found -> done!
  - If not found & node (block) is leaf -> not there!
  - Otherwise, determine which index file block # to access next based on rules of construction of m-way search tree
  - Access that block from index file
  - Repeat above step “Linearly search for key in accessed block”
- If found desired key: determine its matching block # and access that block from data file

Search - Take #4 - B Tree

- Assuming the entire Index file (B Tree) cannot be loaded into main memory.
- In analyzing the search time efficiency, we need to know how many levels a B Tree (accommodating 30M records) has.
- Answer:
  - Assuming we are using a B Tree of order 4 to store our 30M keys (and matching block #'s) and that each node of the B Tree is filled (i.e., each node contains 3 key pairs) and that every level of our B Tree is filled, then our B Tree contains:

    $(4^L - 1)$ key pairs, where L is the number of levels.

Search - Take #4 - B Tree

- Hence a data collection containing 30,000,000 data records will have

  $$\frac{\log_2(30{,}000{,}001)}{\log_2 4} \text{ or \_\_\_\_\_\_ levels!}$$

- In this example, we could increase the value of *m*, which would decrease the number of levels in our B Tree, hence further reduce the number of disk accesses performed during a search of our data collection containing 30M Canadians

Advantage of B Trees

- Good for disk-bound data
  - When n is large, m can be set to a large number, which keeps the number of levels low
  - Since the number of disk accesses is proportional to the number of levels in a tree, then small # of levels translates into small number of disk accesses, and hence good time efficiency for search/insert/remove operations
- In practice, commercial databases use specialized versions of these search trees where *m* is of the order of 100

External Sorting

Sorting

- Assume we inserted our 30M Canadian records into a random access disk file.

- How can we sort these records?
  - Let's look at our favourite algorithms
    - QuickSort
    - HeapSort
    - MergeSort

QuickSort

- Find pivot
- Walk data, swapping entries greater than / less than pivot

- Is this going to work well if data are stored on disk?

HeapSort

- Heapify data
  - Call bubbleUp repeatedly
- Remove data from heap

- Is this going to work well if data are stored on disk?

Merge Sort

- The simplest algorithm that can be used to sort disk-bound data, and one that turns out to be quite efficient, is **Merge Sort.**
- Recall the internal Merge Sort algorithm:
  - divide the data collection into two sections of approx. equal size
    - recursively apply the algorithm to sort each of the smaller sections -> sorting is done on adjacent records
    - merge the sorted sections back together

External MergeSort example

- Suppose we're trying to sort 32 million records
- Suppose disk *blocks* hold 1 million records
  - I.e. reading 1 million records is roughly as fast as reading 1
- Suppose we only have enough memory to hold 3 million records in memory at a time

- Let's see how we can MergeSort under these constraints

External Merge Sort Algorithm

- Phase 1:
  - Divide 32 million records into groups of 1 million
  - Read each 1 million into memory in turn
    - For each group i, sort and write back to disk as R_i (sorted)
  - This phase can be done under our constraint
- Phase 2:
  - Merge sorted groups R_1,…,R_32
  - Let's see why this can be done under our constraint

External Merging

- Recall constraint: only 3 million records in memory at a time, blocks are 1 million
- We need to merge up 32 sorted files R_1, …,R_32 each with 1 million records
- First level merge is easy?
  - Merge R_1 and R_2 into R_{1,2}, R_3 and R_4, …
  - Each merge only requires 2 million records
- What about the second level merges?
  - That is, merging R_{1,2} and R_{3,4}

Larger Merges

- Suppose we are merging one sorted 8 million record file with another
- Only need memory for 3 million records!
  - Read 1 million records (a block) from file 1
  - Read 1 million records (a block) from file 2
  - Allocate memory for 1 million records for output
- Start merging
  - Once the output is full, write it to disk
  - Once a file input block is finished, read another

Why is this better?

- QuickSort is O(n log n)
  - But how many disk reads will it require?
  - O(n log n)
- External MergeSort is O(n log n)
  - But how many disk reads will it require?
  - O(n/B log n/B)
    - Where B is the number of records in a block

Summary

Summary

- How to handle big datasets?
  - Big = do not fit in memory
- Disk access is slow
- Minimize number of disk accesses algorithms perform
- Searching
  - Index files and data files
  - Can access index file from disk too if it's too big
- Sorting
  - Use MergeSort

Readings

- Carrano: Ch. 14