Data Structures and Algorithms

Review

- Binary Trees (Binary Search Trees)
  - A hierarchical data structures designed to allow very fast searching.
  - Each node has 0, 1 or 2 children
  - Key functions operating on a binary tree
    - Searching
    - Traversing
    - Insertion
    - Deletion
    - Balancing

Consider Inserting

- Suppose we need to insert a node on the 5^{th} level of a binary tree.
  - First we need to test the root node and choose which branch to take.
  - Then test the second level node…
  - …
  - …then insert the node.
- The algorithms discussed last week, make this process fast, by following pointers.

Secondary Memory vs RAM

- During our discussion, we assumed the data would be stored in primary memory or RAM. Lets consider if the data is too big to store in RAM, so needs to be stored on a hard disk.
- The access time for parts of the memory includes;
  - *seek time* + *rotation time* + *transfer time*
- The seek time is particularly slow as it depends on mechanical movement - diskhead physically moving to the correct position.

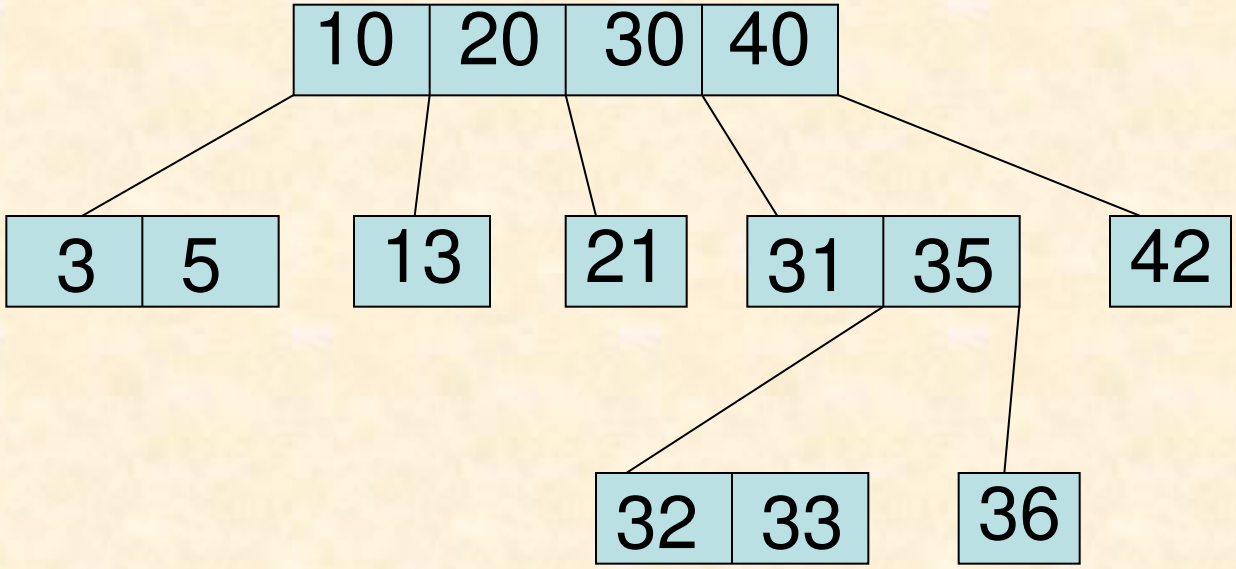
Trees in Secondary Memory

- Constructing a tree for storage in secondary data takes more consideration.
  - Binary Trees may be spread across multiple blocks of disk memory.
- The seek time is in the order of milliseconds, while CPU processes are in the order of microseconds (at least 1,000 times faster).
- Therefore, processing is essentially free (when considering big O).
- Extra time spent on processing could reduce the need for seek time.

Introducing Multiway Trees

- A multiway tree differs from a binary tree in a few key ways;
  - Each node has *m* children
  - Each node has *m-1* keys
  - The keys are in ascending order
  - The keys in the first i children are smaller than the ith key.
  - The keys in the last m-i children are larger than the ith key.

A Multiway Tree



Note this tree suffers from malaise, as it is unbalanced – it takes longer to find 32 than 21.

Multiway trees and Disk Access

- Disk Access costs are expensive, thus if possible data should be arranged to minimise the number of accesses.
- Or to allow more data to be accessed during one disk access.
- A multiway tree allows this.
- A B-Tree is where each node is the size of a ‘block’.
  - The number of keys in each node depends on the size of the keys and the size of the block.
  - Block size can depend on the system.

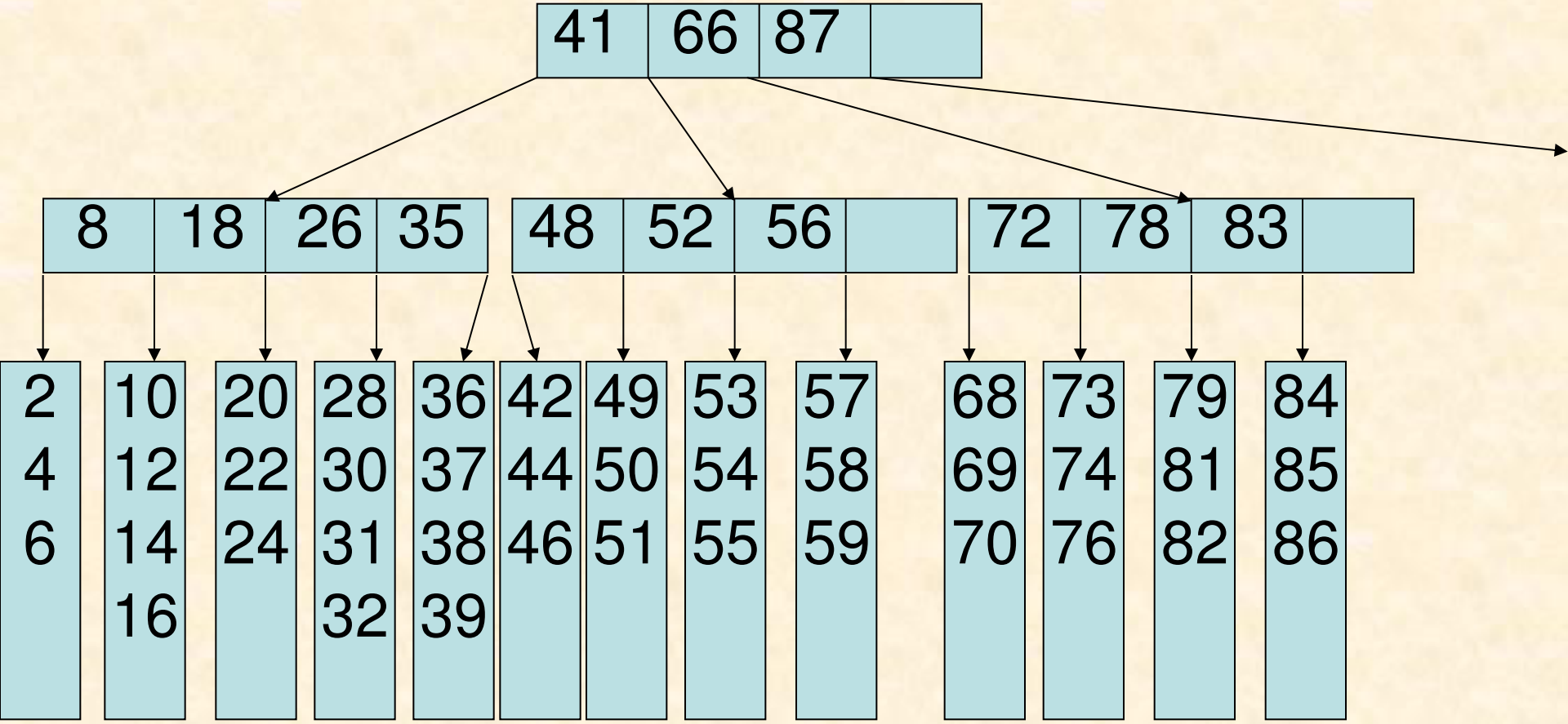
The Family of B-Trees

- Lets look at some types of Multiway trees
  - B Trees
  - B* Trees
  - B+ Trees
  - Prefix B+ Trees
  - Bit Trees

B-Trees

- A B-Tree of order m has the following properties;
  - A root has at least 2 subtrees (unless it's a leaf)
  - Each nonroot and nonleaf node has k-1 keys and k pointers to subtrees where $[m/2] \leq k \leq m$
  - Each leaf node holds k-1 keys where $[m/2] \leq k \leq m$
  - All leaves are on the same level.
- Essentially this means that a B-Tree is at least half full (only when a node fills an entire block does it split into subtrees).

B-Tree of order 5



B-Tree

- Notice that all nodes are at least half full.
- Each node has k pointers, and k-1 keys.
  - 5 pointers and 4 keys
- Finding the right size of k, depends on the size of each key and the size of each block.
- The number of levels depends on the amount of data to be stored.
- Note that the root, and perhaps the first level could be stored in RAM, so less secondary memory access is required.

Implementing a B-Tree

```
template <class T, int M>
class BTreeNode {
public:
    BTreeNode();
    BTreeNode(const T&);
private:
    bool leaf;
    int keyTally;
    T keys[M-1];
    BTreeNode *pointers[M];
    friend BTree<T,M>;
};
```

Searching a B-Tree

- Very similar to searching a binary tree
  - Beginning at the root node, branches are chosen as their values appear either side of the search value.

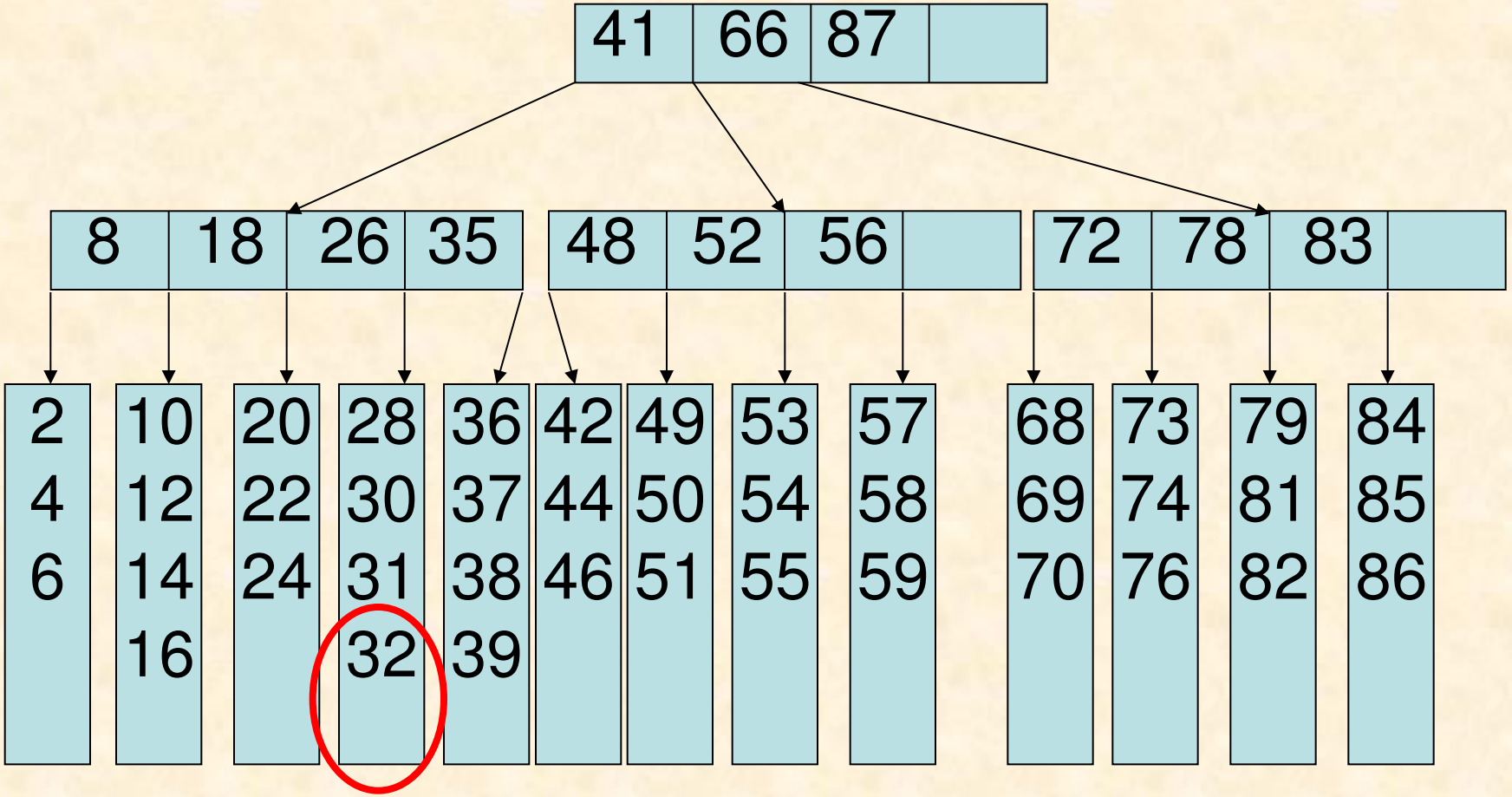
Inserting a node

- When dealing with binary trees, they are built from the top down;
  - i.e. the root node is placed, and then nodes are divided around it.
- When building a b-tree we can build it up from the leaves.
  - i.e. the leaf nodes are positioned and rearranged, and eventually the root nodes are specified.
- This is because b-tree's have the specification that in a b-tree all leaves must be at the same level.
  - Therefore all nodes are inserted as leaves.

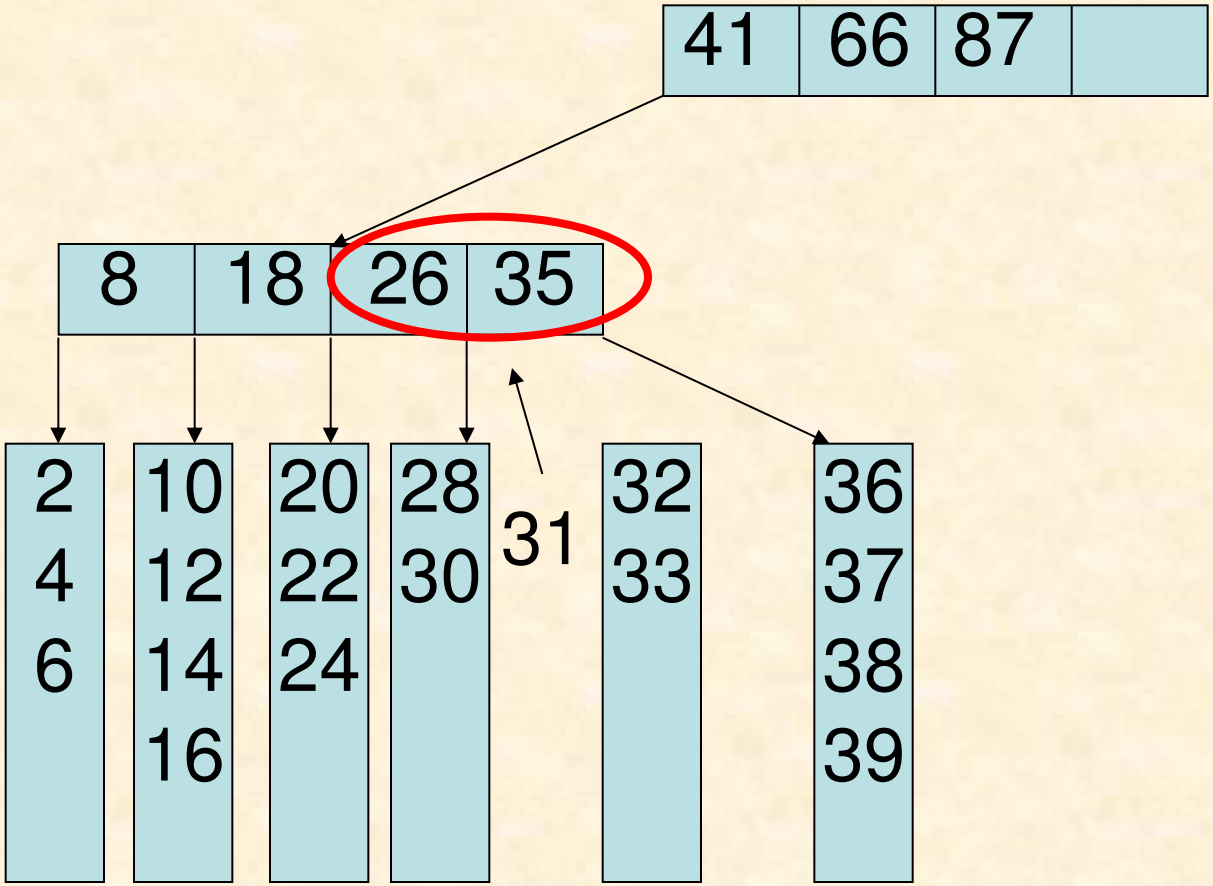
Insertion

- Search the tree to find where the leaf should be placed.
- If there is space insert the node.
  - if there are less than m-1 leaves already there.
- Otherwise the node must be split.
  - Typically the median is chosen – with lower value nodes forming the left branch, and higher values nodes forming the right branch.
  - The median is then added to the parent, which may or may not need to be split.
  - And so on until the root is reached.

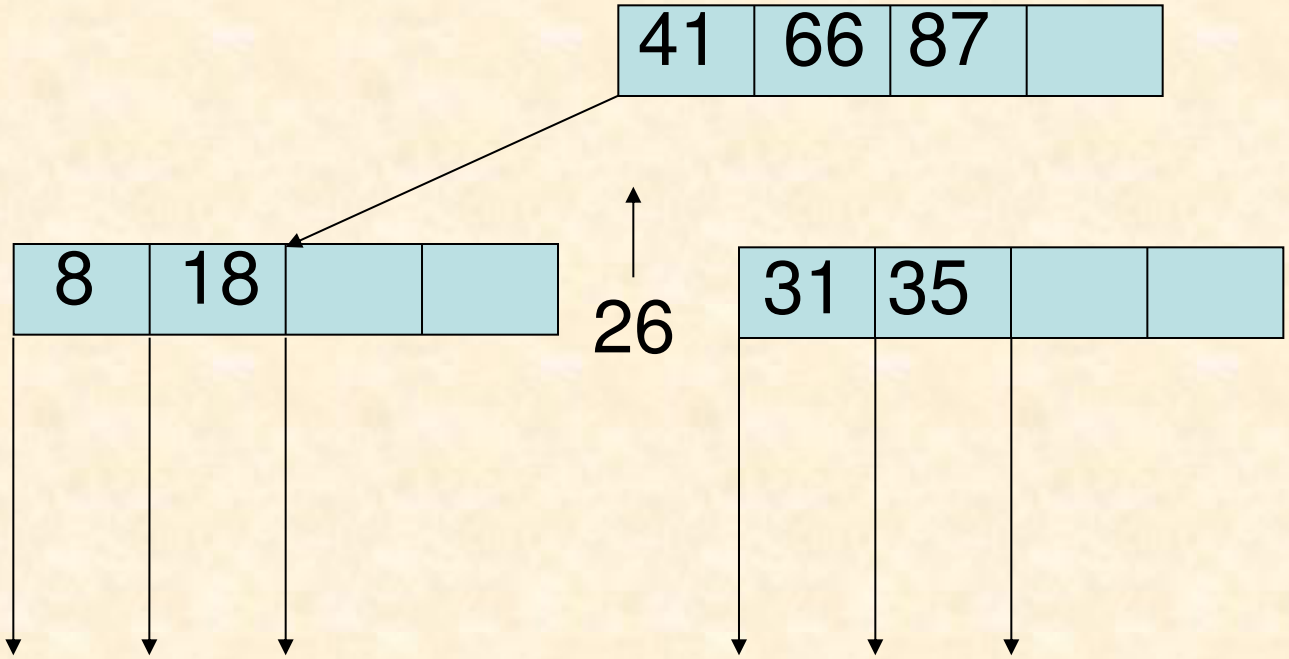
Insert 33


41 66 87
8 18 26 35
48 52 56
72 78 83
2 4 6
10 12 14 16
20 22 24
28 30 31 32
36 37 38 39
42 44 46
49 50 51
53 54 55
57 58 59
68 69 70
73 74 76
79 81 82
84 85 86

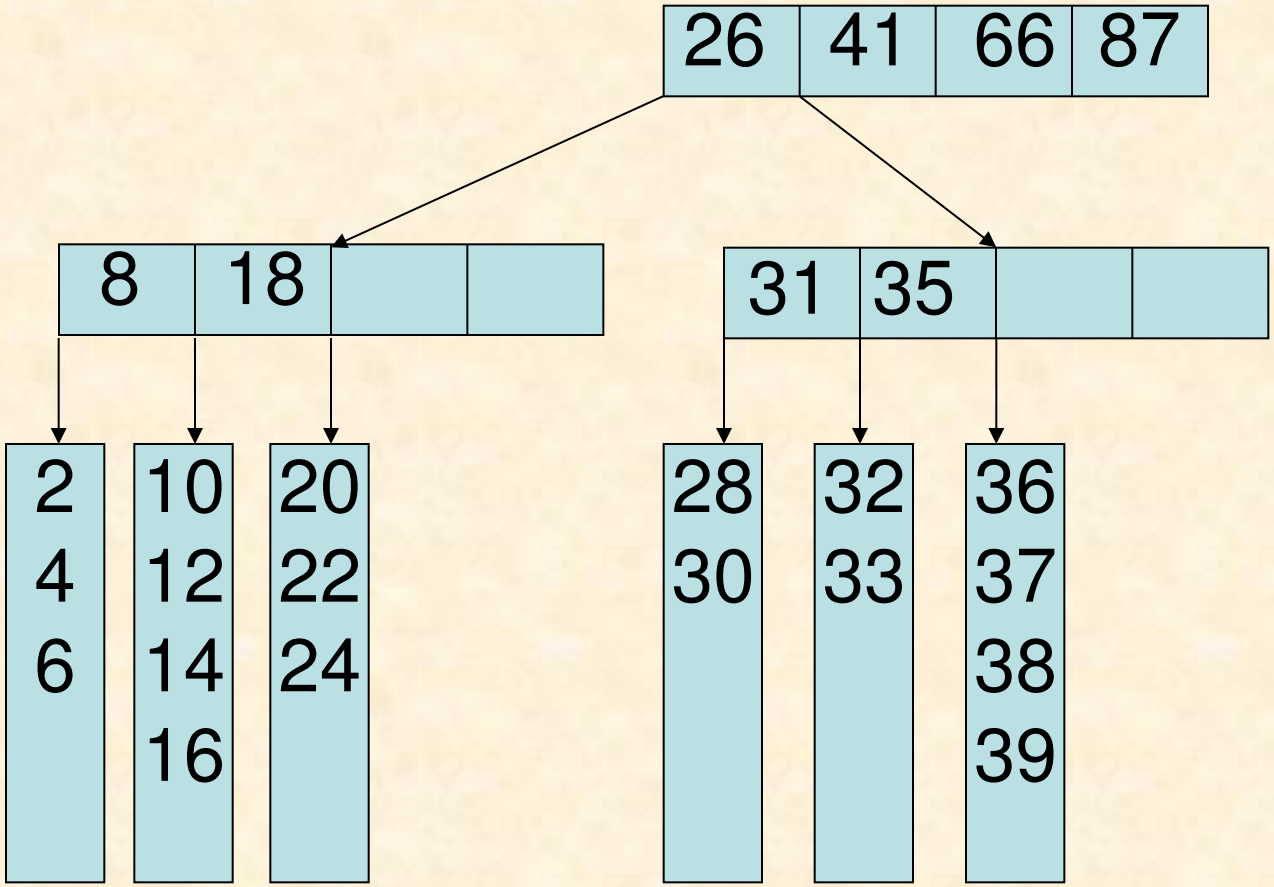
Insert 33 (2)



Insert 33 (3)



Insert 33 (4)



Deleting a Node

- Deleting a node from a b-tree also presents some problems to be addressed.
  - Is the node in a leaf?
    - If so will the leaf still be at least half full?
  - Is the node an internal node?
    - Which new node will become a separator value?

Deleting Leaf Nodes

- Leaf nodes can simply be deleted, which may result in a leaf having too few elements.
  - In this case the tree will need to be rebalanced after node removal.
- The tree can be rebalanced by merging two leaf nodes, choose a sibling leaf node and redistribute the keys.
  - If the left or right node has enough siblings, the median is chosen as the new key and nodes distributed to each leaf.
  - This may lead to a parent node without enough keys, so may iterate towards the root.

Deleting an Internal Node

- If an internal node needs to be deleted;
  - the largest valued node in the left subtree or the smallest valued node in the right subtree become candidates for promotion.
- One of these is chosen, which means deleting a node from either a leaf or an internal node;
  - either case we have now defined.

B*-Trees

- Clearly each node in a B-tree represents a new block of secondary memory – accessing this is expensive.
- To reduce the disk accesses further B*-Trees were proposed.
- The difference between B-Trees and B*-Trees is that B*-Trees must be two thirds full (rather than just half full).

B*-Trees

- B*-Trees delay the splitting nodes by splitting 2 nodes into 3, rather than 1 node into 2.
- Note that B**-Trees are trees which are required to be 75% full.

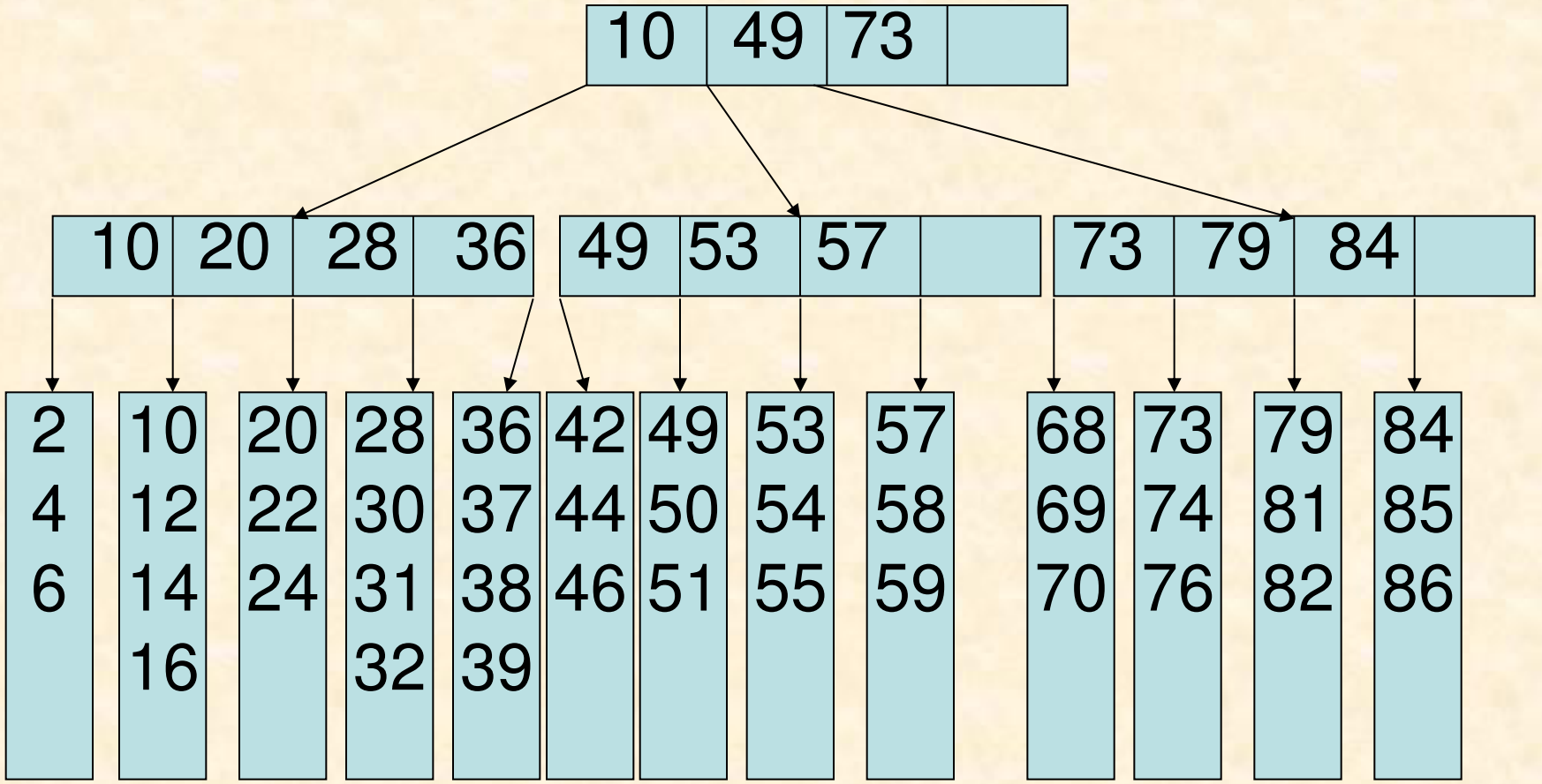
B+Trees

- We have looked at traversal algorithms, which allow us to traverse a binary tree.
  - The in-order algorithm allowed us to start with the lowest value, and then traverse the values in ascending order.
- This is somewhat efficient when transferred to B-Trees.
  - Leaf nodes can all be read from secondary memory in one go.
  - However, when reading non-leaf nodes, we can only read one value per time.

B+Trees

- B+ Trees are variations on a B-Tree, where the internal nodes are simply indexes allowing quicker searching of the tree.
- The values stored in index nodes are repeated in the leaf nodes
  - Essentially all data is stored in leaf nodes, with indexes used to point to the correct leaf.
- In this way only leaf nodes need to be read for in-order traversal.

B+ Tree


10 49 73
10 20 28 36
49 53 57
73 79 84
2 4 6
10 12 14 16
20 22 24
28 30 31 32
36 37 38 39
42 44 46
49 50 51
53 54 55
57 58 59
68 69 70
73 74 76
79 81 82
84 85 86

Prefix B+ Trees

- It is noticeable that if we delete a node from a B+Tree, it isn't always necessary to change the internal node value;
  - For instance, removing the 49 node in the previous slide, doesn't necessitate removal of the 49 index – it is still useful for locating appropriate data.
- Therefore, it is clear that for indexes to be appropriate to guide towards correct data, they needn't actually be the values stored in the leaves.
  - A Prefix B+ Tree stores just a prefix to the data stored in a leaf in the index nodes.
  - For instance 4 or AB.
  - This is similar to the keyword at the top of a dictionary page.

Bit Trees

- One benefit of a Prefix B+ Tree is that an entire data field doesn't need to be stored as the index;
  - consider if the tree contains complicated objects.
- Instead only a small amount of data is stored to direct searches to the leaf.
- A Bit Tree takes this approach to the extreme, by storing the minimum data in an index – a Distinction Bit.
  - A D-Bit is the bit needed to distinguish between two values;
    - K = 01001**0**11
    - N = 01001**1**10