Searching(2)

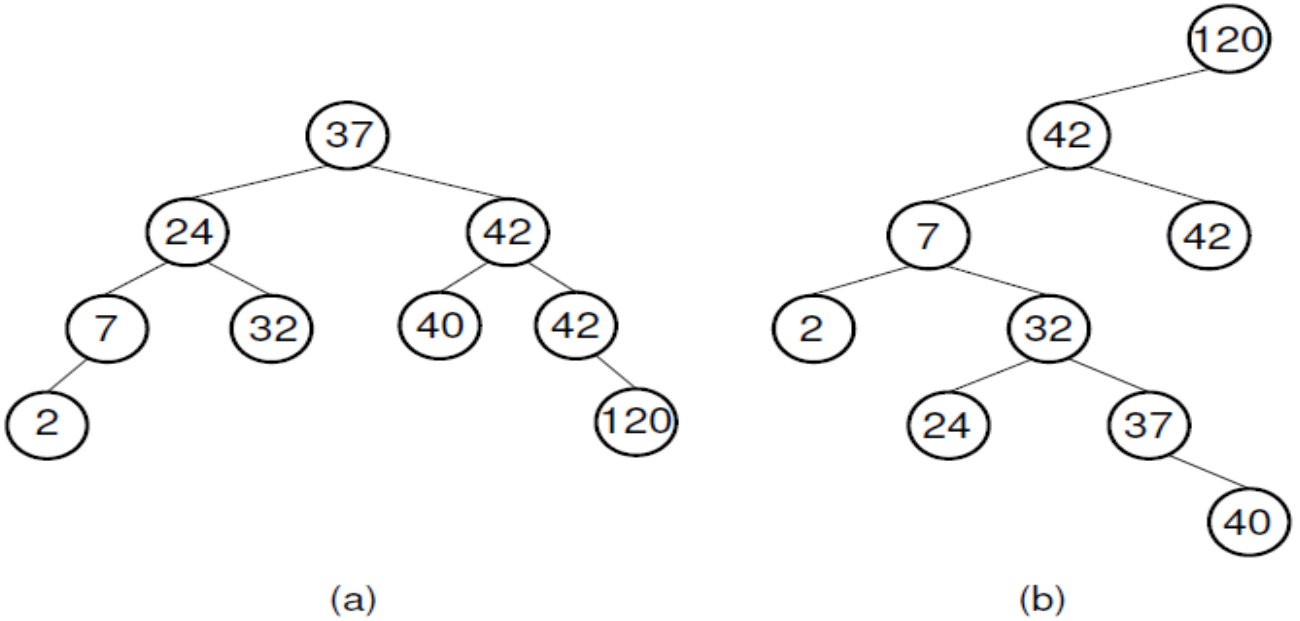
College of Computer Science, CQU

outline

- Binary Search Tree
- B- Tree
- B+ Tree

Binary Search Tree

- Binary Search Trees (BST)



(a) (b)

Figure 5.13 Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.

BST Operations: Search

Searching in the BST

`method search(key)`

- implements the binary search based on comparison of the items in the tree

- the items in the BST must be comparable (e.g integers, string, etc.)

The search starts at the root. It probes down, comparing the values in each node with the target, till it finds the first item equal to the target. Returns this item or `null` if there is none.

Search in BST - Pseudocode

if the tree is empty
    return `NULL`

else if the item in the node equals the target
    return the node value

else if the item in the node is greater than the target return the result of searching the left subtree

else if the item in the node is smaller than the target return the result of searching the right subtree

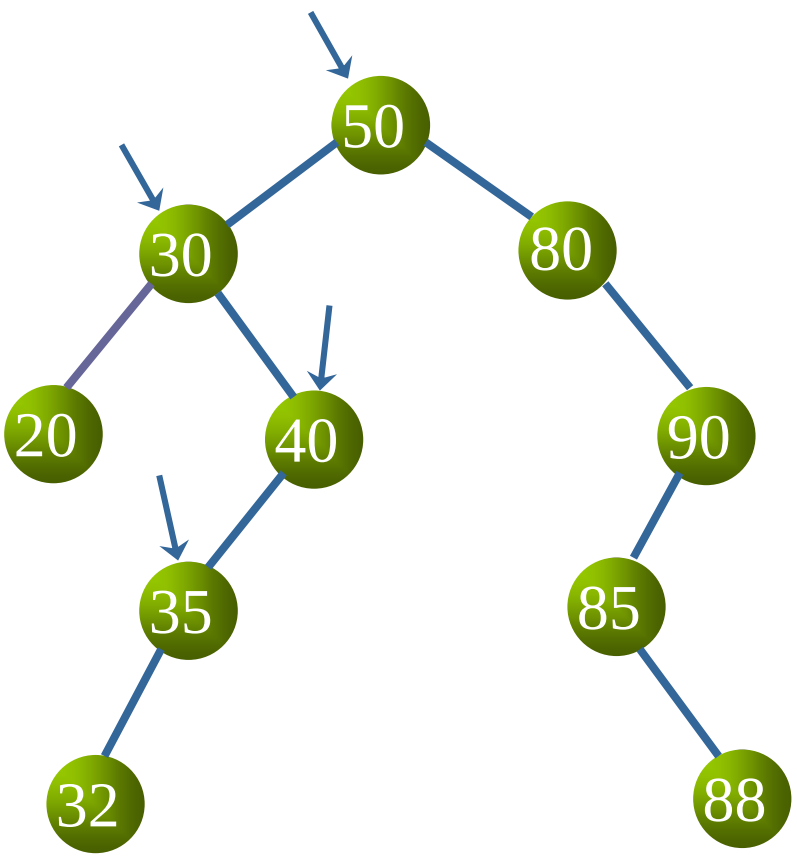
Search in BST - implementation

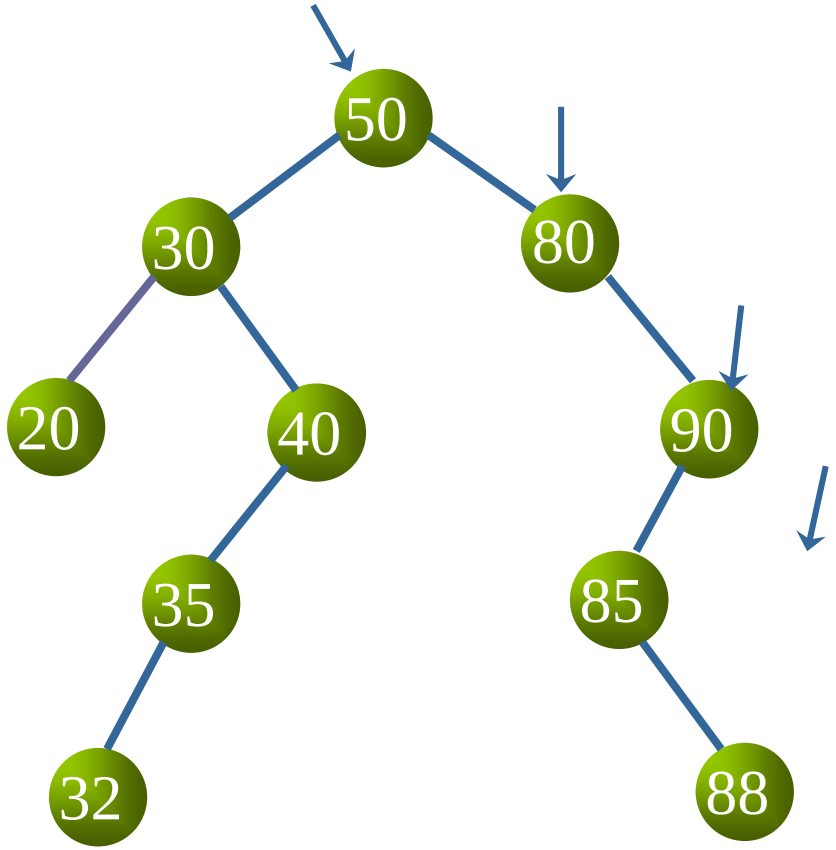
```
// Return Record with key value k, NULL if none exist.
// k: The key value to find. */
// Return some record matching "k".
// Return true if such exists, false otherwise. If
// multiple records match "k", return an arbitrary one.
E find(const Key& k) const { return findhelp(root, k); }

// Re
int s

void
  if
  els
}
};
```

```
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                        const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);   // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

Search in BST - Example

Search for 35， 95





B-Trees

- B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. B-trees are always height balanced, with all leaf nodes at the same level.
2. Update and search operations affect only a few disk blocks. The fewer the number of disk blocks affected, the less disk I/O is required.
3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on searches due to locality of reference.
4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

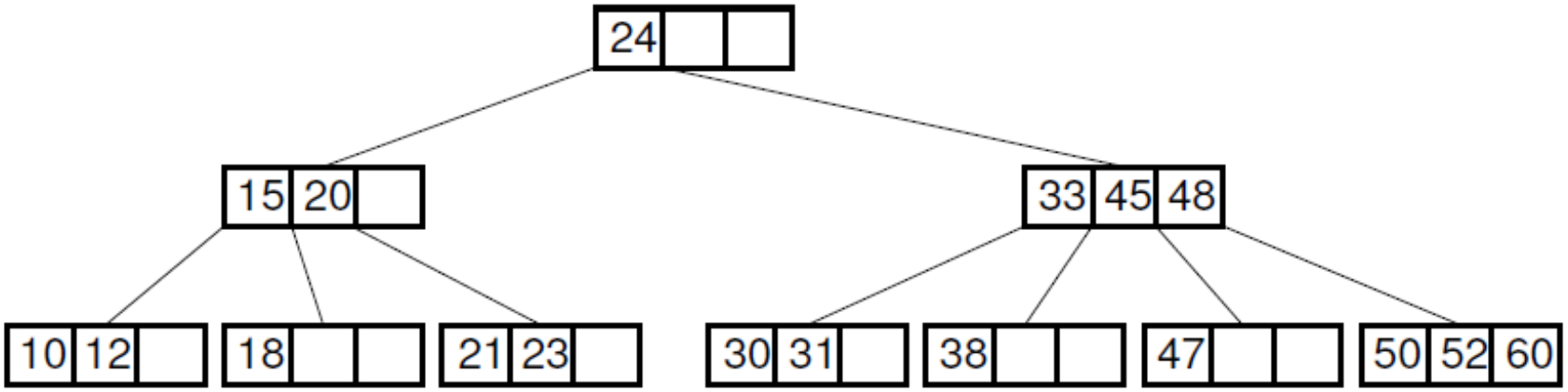
B-Trees

- A B-tree of order m is defined to have the following shape properties:

- The root is either a leaf or has at least two children.
- Each internal node, except for the root, has between 「m/2」 and m children.
- All leaves are at the same level in the tree, so the tree is always height balanced.

B-Trees: Example

A B-tree of order four



Searching a B-Tree

- Search in a B-tree is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

Searching a B-Tree

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

Start at the leftmost key in the node, and go to the right until you go too far.

If it is a leaf node, then you are done, as there is no leaf to inspect

Otherwise, retrieve the child node from the disk, and put it into memory

Search in a B-tree: Example

- Consider a search for the record with key value 47 in the tree of Figure 10.17.
- The root node is examined and the second (right) branch taken.
- After examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B^+-tree

- B^+-tree :
  - a variant of the B-tree
  - most commonly implemented
  - more efficient.
- The B^+-tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.
- The most significant difference between the B^+-tree and the BST or the standard B-tree
  - the B^+-tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search.

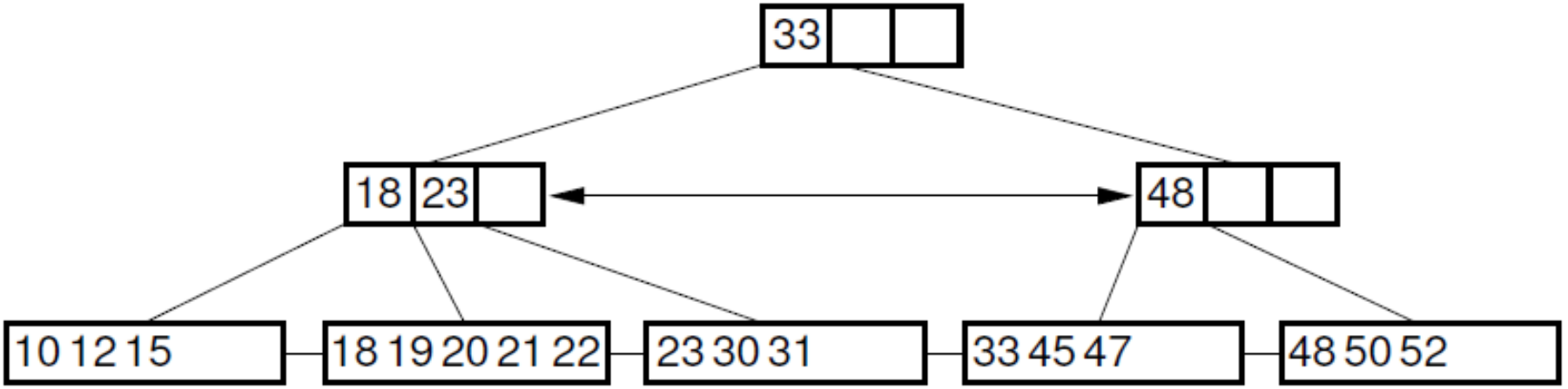
B+-Tree: Properties

- A B+-tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length
  - Two types of nodes: index (internal) nodes and data (leaf) nodes. Each node is one disk page.
  - Each node must have minimum 50% occupancy (except for root). Each node contains d <= m <= 2d entries/pointers.
    - d is the order/branching factor/capacity of the tree
    - The root must have at least 2 children

B+-Trees Example

Example of a B+-tree of order four:

- Internal nodes must store between two and four children.
- For this example, the record size is assumed to be such that leaf nodes store between three and five records.



Search a B$^+$-tree: Example

- To find a record with key value 33 in the B$^+$-tree of Figure 10.18
  1. search begins at the root.
  2. The value 33 stored in the root merely serves as a placeholder, indicating that keys with values greater than or equal to 33 are found in the second subtree.
  3. From the second child of the root, the first branch is taken to reach the leaf node containing the actual record (or a pointer to the actual record) with key value 33.

Queries on B^+-Trees

- Find all records with a search-key value of *k.*

1. Start with the root node
   1. Examine the node for the smallest search-key value > *k.*
   2. If such a value exists, assume it is K_j. *Then follow* P_i *to the child* node
   3. Otherwise $k \geq K_{m-1}$, *where there are m pointers in the node. Then* follow P_m *to the child node.*
2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
3. Eventually reach a leaf node. If for some *i, key* $K_i = k$ follow pointer P_i *to the desired record. Else no record with* search-key value *k exists.*

B+-tree search method: Implementation

```
template <typename Key, typename E>
E BPTree<Key, E>::findhelp(BPNode<Key,E>* rt, const Key k)
                                  const {
  int currec = binaryle(rt->keys(), rt->numrecs(), k);
  if (rt->isLeaf())
    if ((((BPLeaf<Key,E>*)rt)->keys())[currec] == k)
      return ((BPLeaf<Key,E>*)rt)->recs(currec);
    else return NULL;
  else
    return findhelp(((BPInternal<Key,E>*)rt)->
                                  pointers(currec), k);
}
```

Summary of B^+ tree

- Tree-structured indexes are ideal for range searches, also good for equality searches.
- B^+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced;
  - Almost always better than maintaining a sorted file.

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.168-185
  - P.442-445

Thank you for listening!