Binary Search Trees

College of Computer Science, CQU

A Taxonomy of Trees

- **General Trees – any number of children / node**
- **Binary Trees – max 2 children / node**
  - Heaps – parent < (>) children
  - Binary Search Trees

BST：Motivation

- **Binary search For sorted array search**
  - search： $\Theta(\log n)$ ，fast
  - insertion： $\Theta(n)$ on average，slow
    - once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.
- **Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly?**

Binary Search Trees

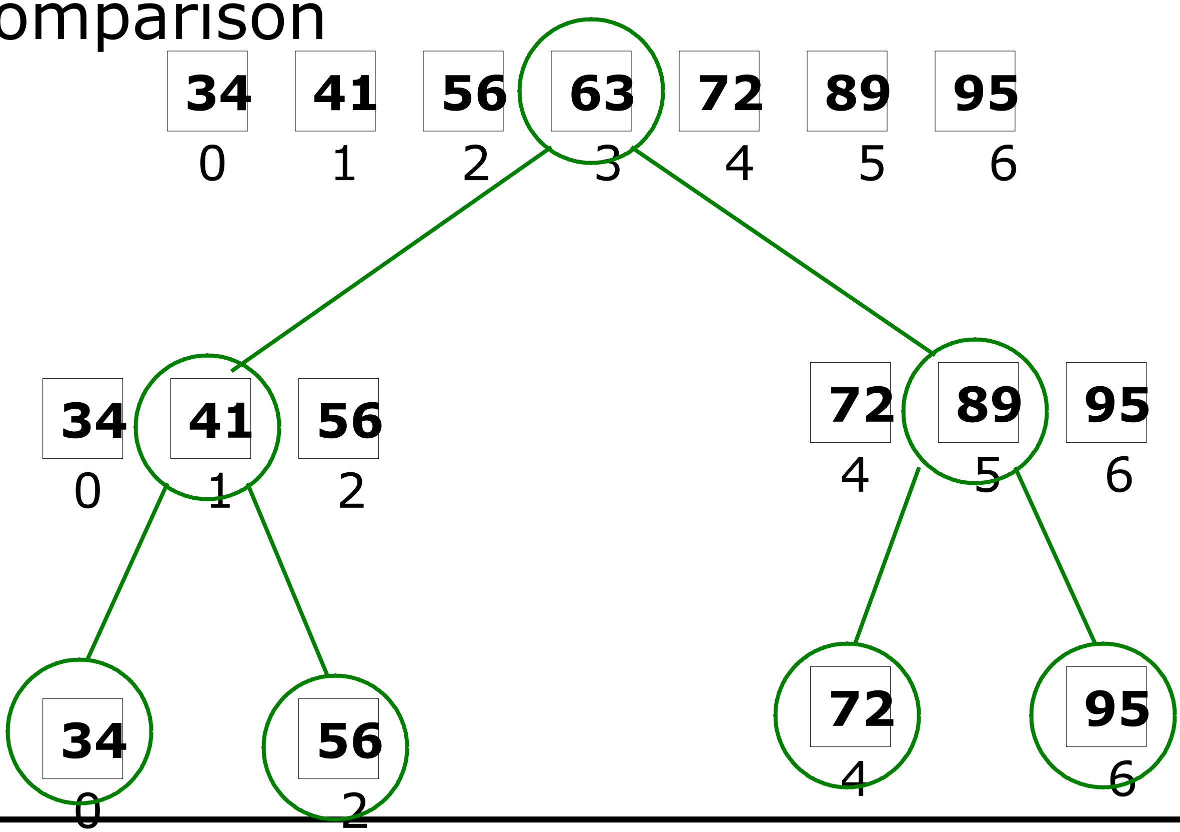
- **Binary search tree (BST)**
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller than(larger than or equal to) the key in the root of subtree.
  - The left and right subtrees are also binary search trees.
- **if the BST nodes are printed using an inorder traversal, the resulting enumeration will be in sorted order from lowest to highest.**

Binary Search Trees

- **Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.**
- **This data organization leads to $\Theta(\log n)$ complexity for searches, insertions and deletions in certain types of the BST (balanced trees).**
  - O(h) in general

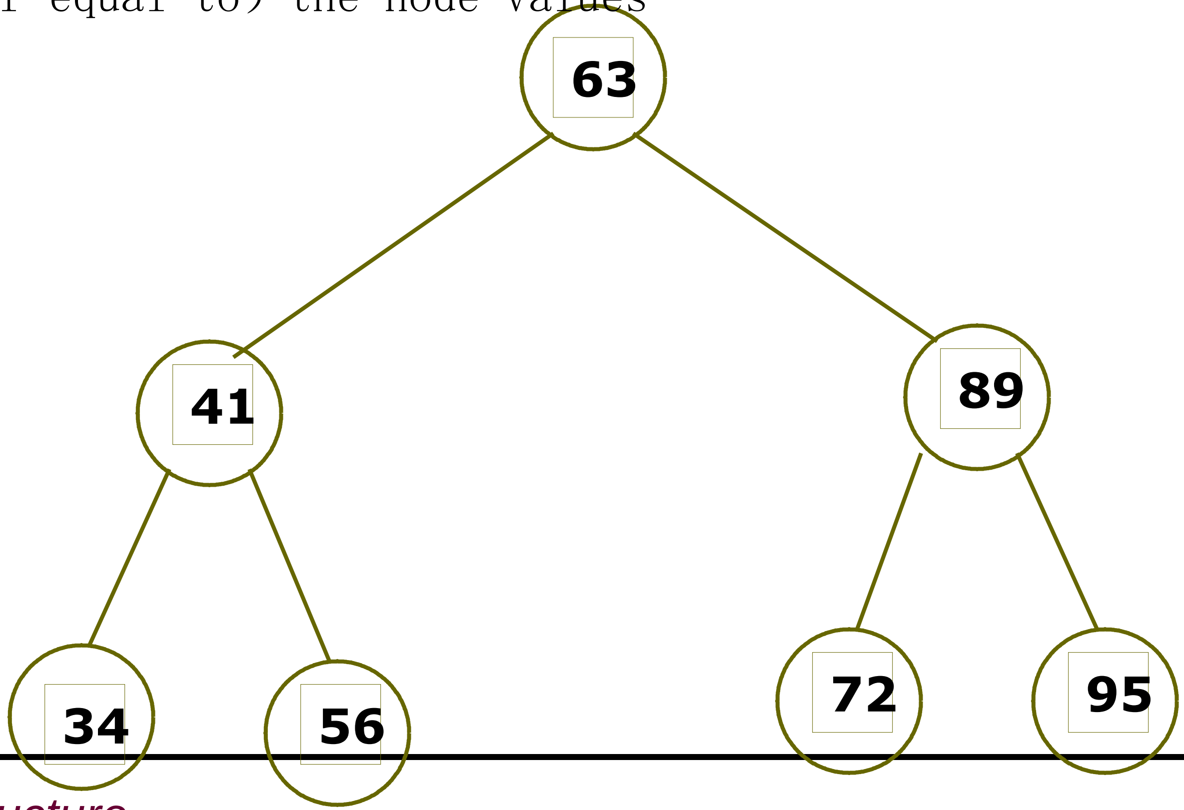
Binary Search Algorithm

Binary Search algorithm of an array of *sorted* items reduces the search space by one half after each comparison

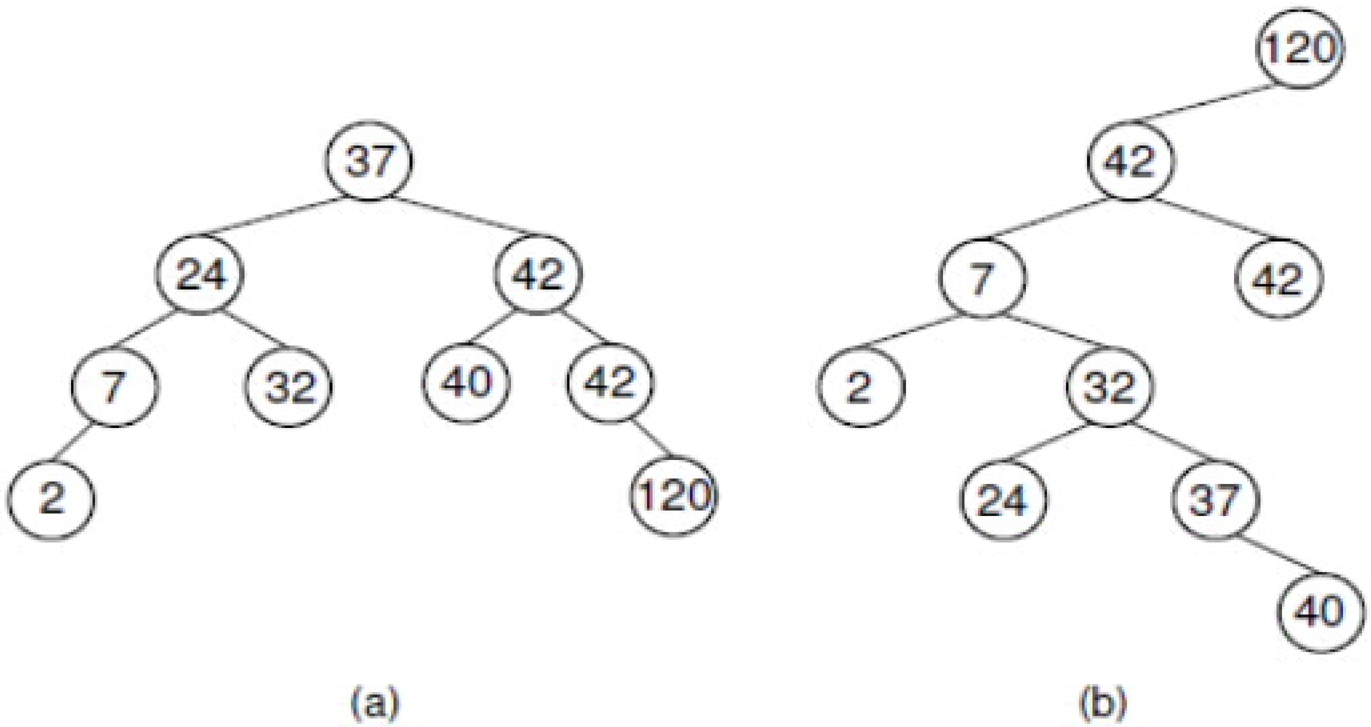


Organization Rule for BST

- the values in all nodes in the left subtree of a node are less than the node value
- the values in all nodes in the right subtree of a node are greater than (or equal to) the node values



BST Example



•The shape of a BST depends on the order in which elements are inserted.

BST：Implementation

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;    // Root of the BST
  int nodecount;          // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                              const Key&, const E&);
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
  E findhelp(BSTNode<Key, E>*, const Key&) const;
  void printhelp(BSTNode<Key, E>*, int) const;

public:
  BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()    // Reinitialize tree
    { clearhelp(root); root = NULL; nodecount = 0; }
```

BST Operations: Insertion

`method insert(key)`

- places a new item near the frontier of the BST while retaining its organization of data:
  - **starting at the root** it probes **down** the tree till it finds a node whose left or right pointer is empty and is a logical place for the new value
  - uses a binary search to locate the insertion point
  - is based on comparisons of the new item and values of nodes in the BST
    - *Elements in nodes must be comparable!*

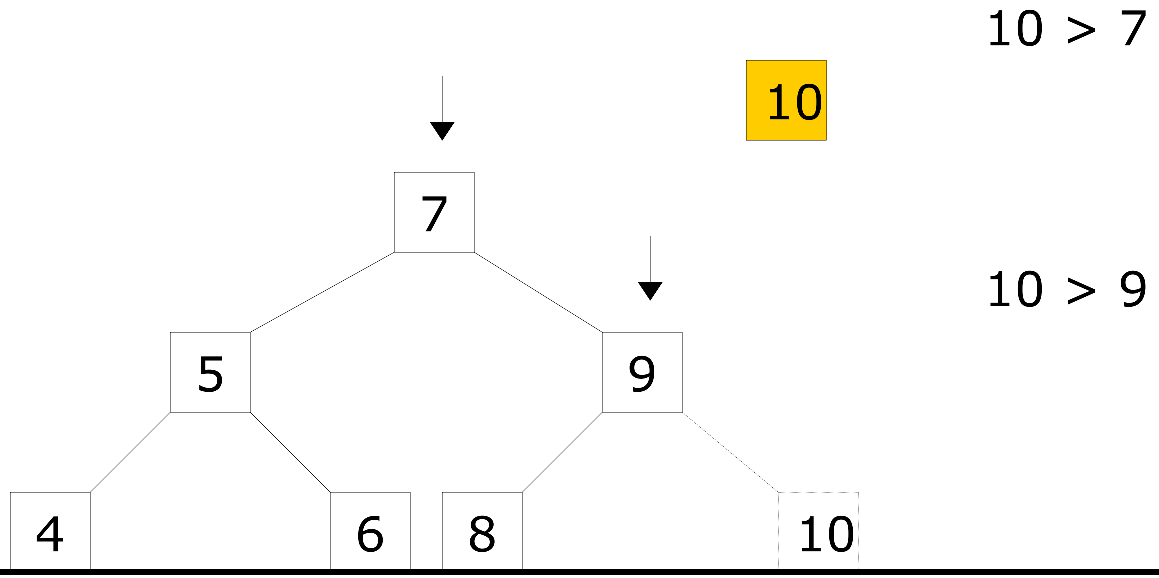
Insertion in BST - Example

Case 1: The Tree is Empty

- Set the root to a new node containing the item

Case 2: The Tree is Not Empty

- Call a recursive helper method to insert the item



Insertion in BST - Pseudocode

if tree is empty
    create a root node with the new key
else
    compare key with the top node
    if **key >= node key**
        compare key with the right subtree:
          if subtree is empty create a leaf node
          else add key in right subtree
    else **key < node key**
        compare key with the left subtree:
          if the subtree is empty create a leaf node
          else add key to the left subtree

BST: Insertion

```
// Insert a record into the tree.
// k Key value of the record.
// e The record to insert.
void insert(const Key& k, const E& e) {
  root = inserthelp(root, k, e);
  nodecount++;
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted
}
```

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); }
```

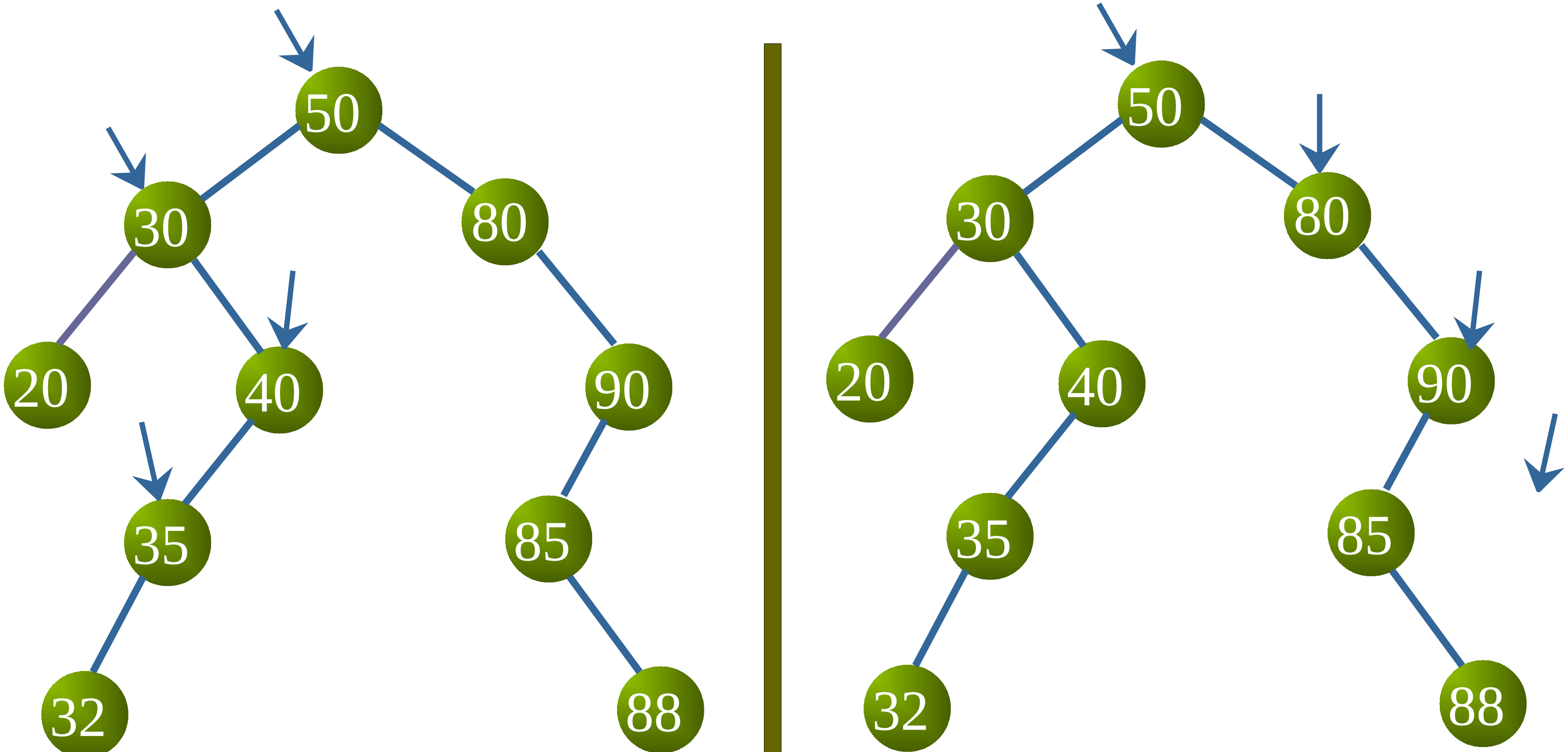
```
// Re
int s

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



BST Operations: Removal

- **removes** a specified item from the BST and **adjusts** the tree
- uses a binary search to locate the target item:
  - **starting at the root** it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

method remove (key)

I if the tree is empty return false

II Attempt to locate the node containing the target using the binary search algorithm

if the target is not found return false

else the target is found, so remove its node:

Case 1: if the node has 2 empty subtrees

replace the link in the parent with null

Case 2: if the node has a left and a right subtree

- replace the node's value with the min value in the right subtree
- delete the min node in the right subtree

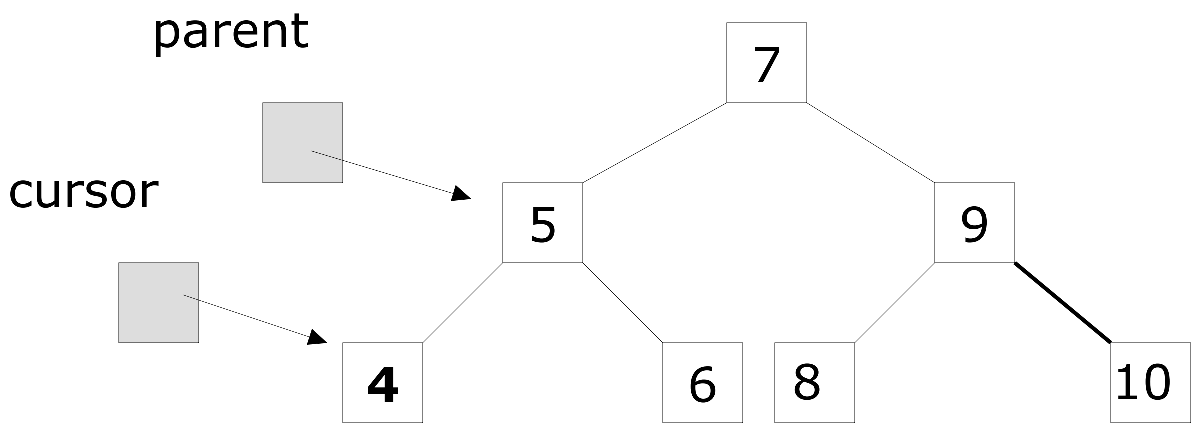
Removal in BST - Pseudocode

Case 3: if the node has no left child
- link the parent of the node to the right (non-empty) subtree

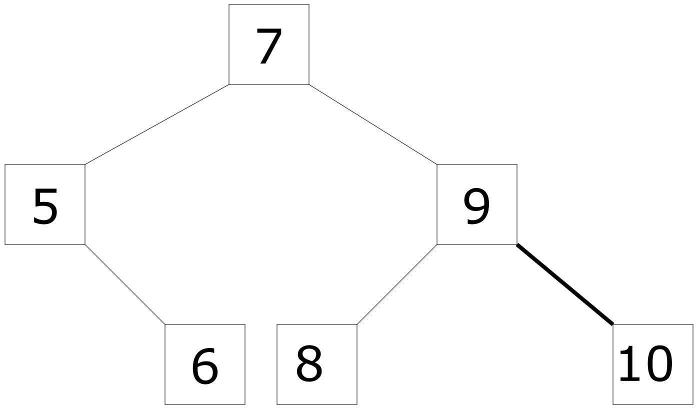
Case 4: if the node has no right child
- link the parent of the target to the left (non-empty) subtree

Removal in BST: Example

Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4
replace the link in the parent with **`null`**

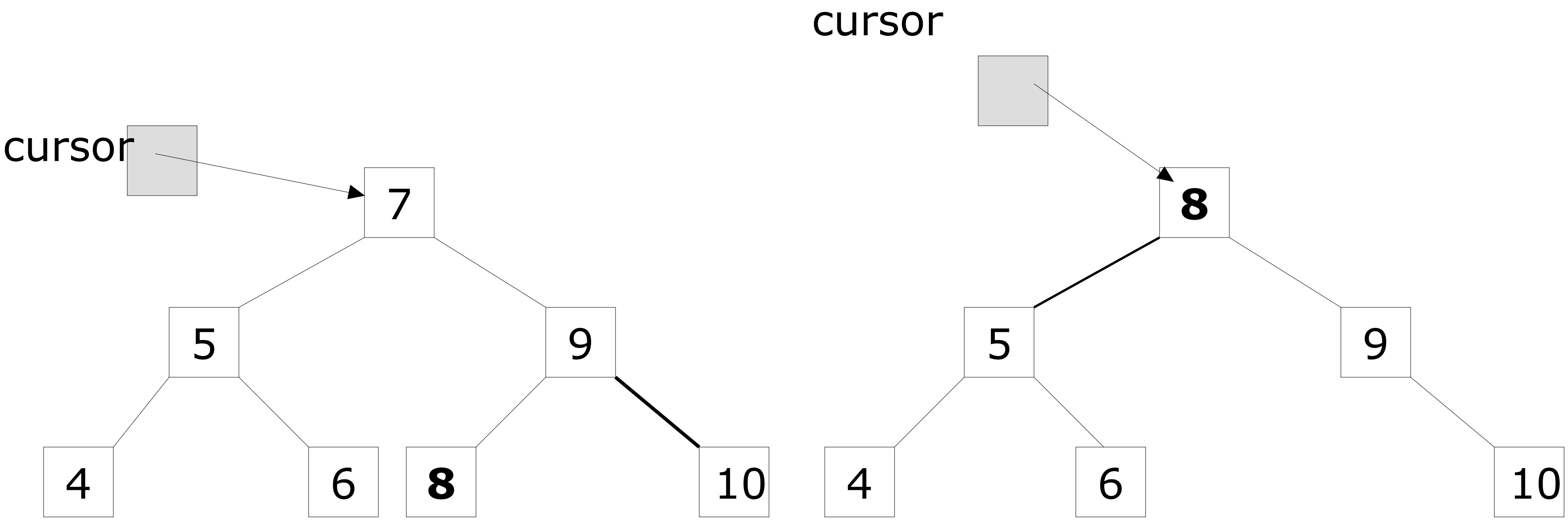


Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

- replace the node's value with the min value in the right subtree
- delete the min node in the right subtree

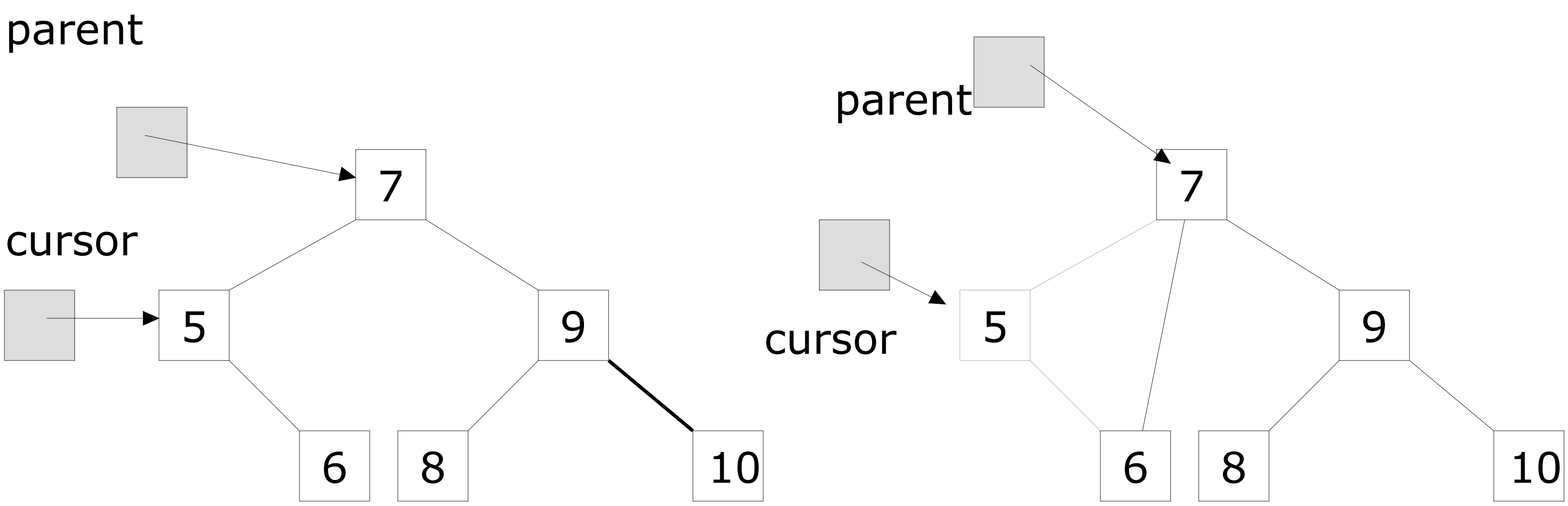
Removing 7



Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

the node has no left child:
link the parent of the node to the right (non-empty) subtree

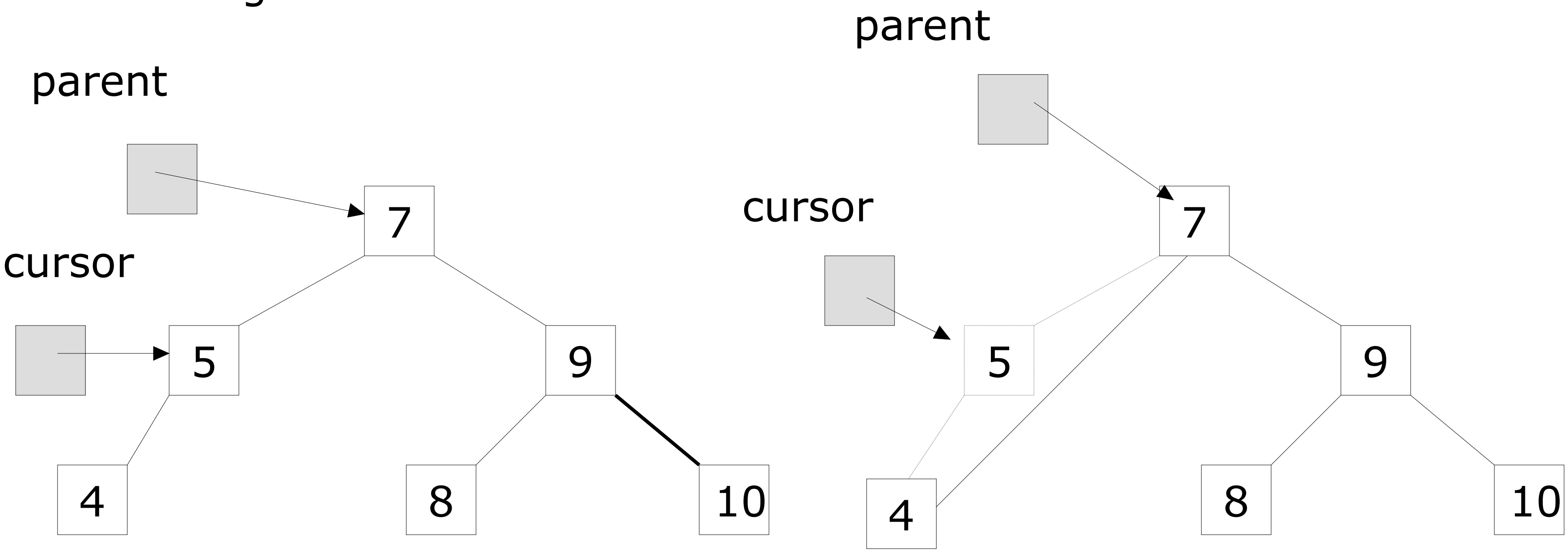


Removal in BST: Example

Case 4: removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5



Removal in BST: implementation

```
// Remove a node with key value k
// Return: The tree with the node remove
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
removehelp(BSTNode<Key, E>* rt, const Ke
  if (rt == NULL) return NULL;    // k i
  else if (k < rt->key())
    rt->setLeft(removehelp(rt->left(), k
  else if (k > rt->key())
    rt->setRight(removehelp(rt->right(),
  else {
    BSTNode<Key, E>* temp =
    if (rt->left() == NULL)
      rt = rt->right();
      delete temp;
    }
    else if (rt->right() ==
      rt = rt->left();
      delete temp;
    }
    else {
      BSTNode<Key, E>* temp
      rt->setElement(temp->
      rt->setKey(temp->key());
      rt->setRight(deletemin(rt->right()));
      delete temp;
    }
  }
  return rt;
}
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
getmin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL)
    return rt;
  else return getmin(rt->left());
}
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();
  else {                        // Continue left
    rt->setLeft(deletemin(rt->left()));
    return rt;
  }
}
```

Analysis of BST Operations

- The complexity of operations **`get`**, **`insert`** and **`remove`** in BST is Θ(h) , where h is the height.
- $\Theta(\log n)$ when the tree is balanced. The updating operations cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become $\Theta(n)$

Balanced Trees

- **BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations.**
- **Solutions :**

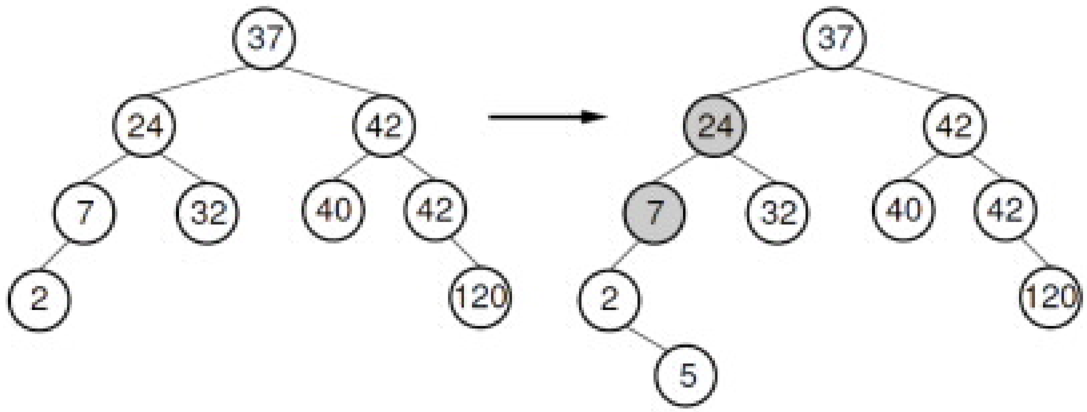
1. **to adopt another search tree structure such as the 2-3 tree or the binary tree.**
2. **to modify the BST access functions in some way to guarantee that the tree performs well.**
   2. requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update

- **If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure, e.g., the AVL tree.**

The AVL tree

- The AVL tree (named for its inventors ***Adelson-Velskii*** and ***Landis***) : a BST with the following additional property:
  - **For every node, the heights of its left and right subtrees differ by at most 1.**
- if a AVL tree contains n nodes, then it has a depth of at most $\Theta(\log n)$. As a result, search for any node will cost $\Theta(\log n)$ and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $\Theta(\log n)$, even in the worst case.
- The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property.
  - **implement the revised update routines in $\Theta(\log n)$ time.**

Insertion in AVL tree: Example



After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

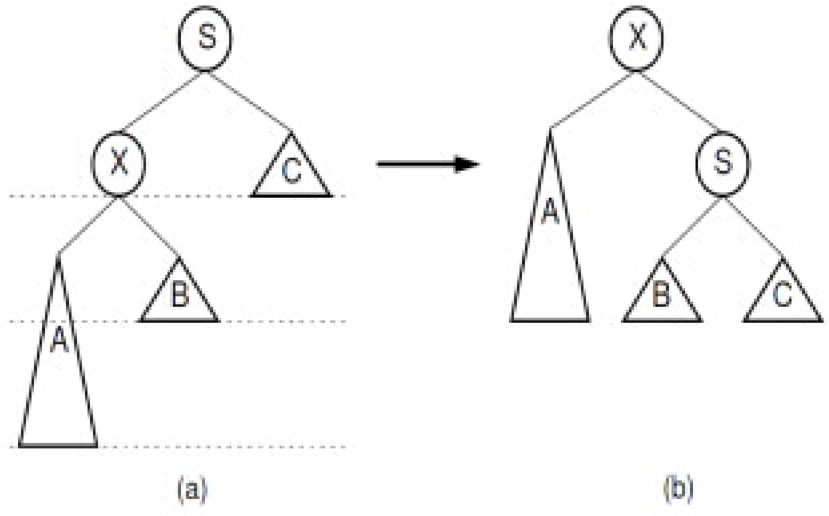
For the bottommost unbalanced node, call it S, there are 4 cases:
1. The extra node is in the left child of the left child of S.
2. The extra node is in the right child of the left child of S.
3. The extra node is in the left child of the right child of S.
4. The extra node is in the right child of the right child of S.

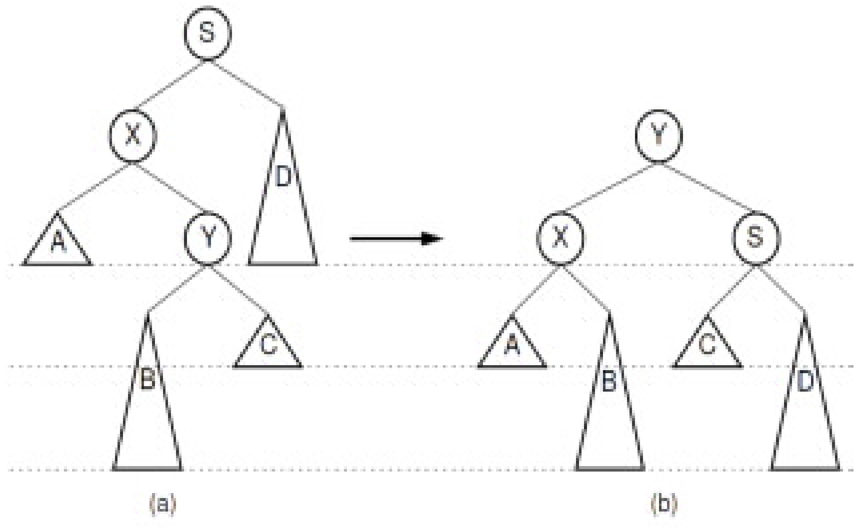
Cases 1 and 4 are symmetrical, as are cases 2 and 3.

Note also that the unbalanced nodes must be on the path from the root to the newly inserted node

How to balance the tree in O(log n) time?

- **using a series of local operations known as rotations**





For case 1 and case 4:
single rotation

For case 2 and case 3:
double rotation

Operations in AVL tree

- **Insertion algorithm:**
  - **begin with a normal BST insert**
  - **Then as the recursion unwinds up the tree, perform the appropriate rotation on any node that is found to be unbalanced.**
- **Deletion is similar**
  - consideration for unbalanced nodes must begin at the level of the deletemin operation.

References

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