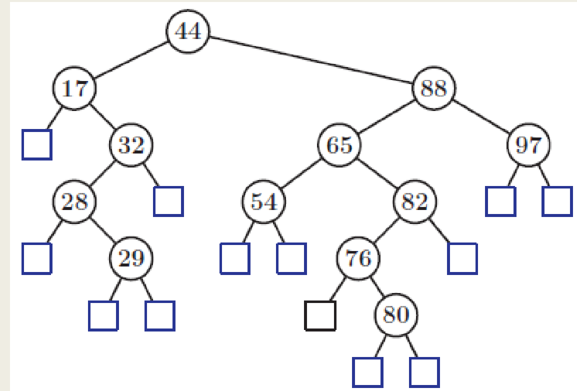
CH10. SEARCH TREES

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

Binary Search Trees: Insertion

- In BST T, each internal node v stores an entry (k, x) such that:
  - *Keys in left subtree of $v \le k$*
  - *Keys in right subtree of $v \ge k$*
- An inorder traversal visits the keys in nondecreasing order.
- Search takes $O(\log n)$ on average
- Insertion
  - *insertAtExternal(v,e): insert e at v (external node), expand v to be internal by having empty external children*



Algorithm TreeSearch(k,v):
    if T.isExternal(v) **then**
        return v
    if $k <$ key(v) **then**
        return TreeSearch$(k, T$.left$(v))$
    else if $k >$ key(v) **then**
        return TreeSearch$(k, T$.right$(v))$
    return v {we know $k =$ key(v)}

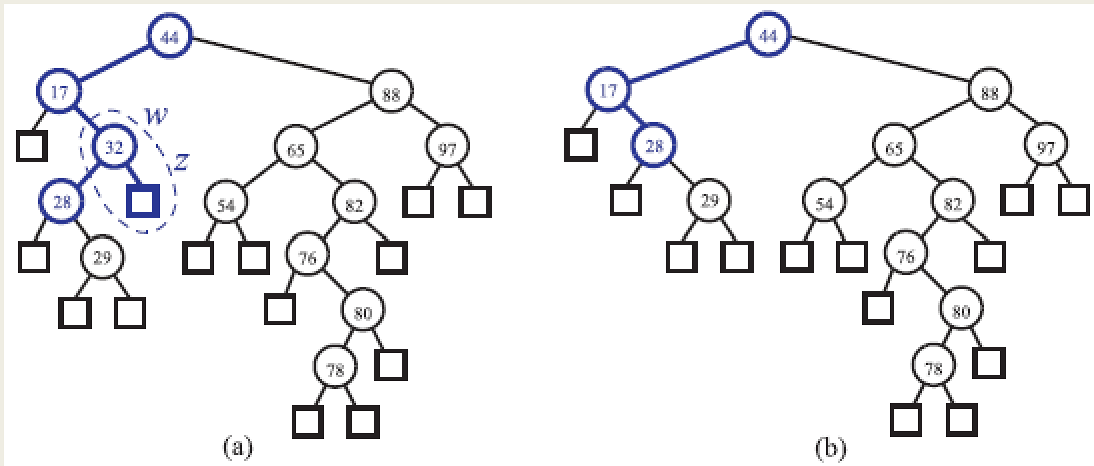
Code Fragment 10.1: Recursive search in a binary search tree.

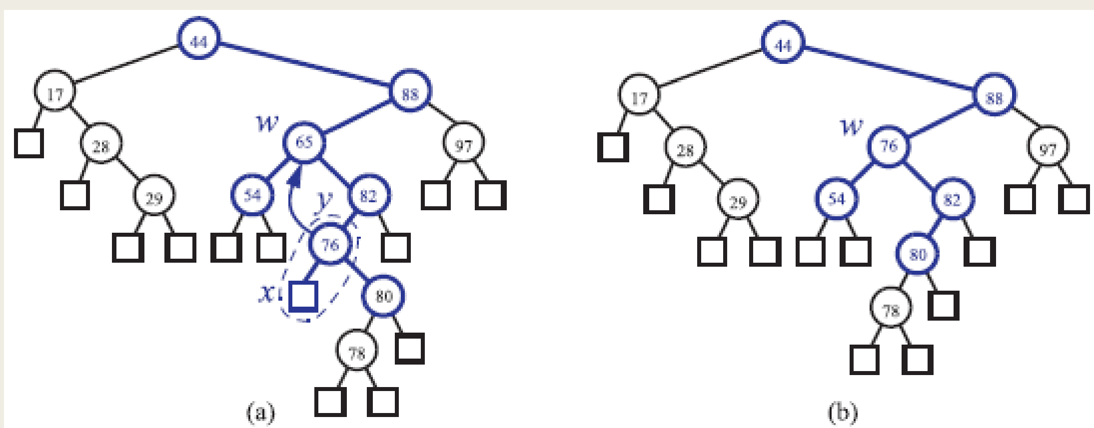
Algorithm TreeInsert(k,x,v):
    Input: A search key k, an associated value, x, and a node v of T
    Output: A new node w in the subtree $T(v)$ that stores the entry (k,x)
    $w \leftarrow$ TreeSearch(k,v)
    if T.isInternal(w) **then**
        return TreeInsert$(k,x,T$.left$(w))$ {going to the right would be correct too}
    T.insertAtExternal$(w,(k,x))$ {this is an appropriate place to put (k,x)}
    return w

Code Fragment 10.2: Recursive algorithm for insertion in a binary search tree.

Binary Search Trees: Removal

- Removal
  - *removeAboveExternal(v): remove v (external node) and its parent, replacing v's parent with v's sibling.*





C++ Implementation of BST

```
template <typename E>
class SearchTree {                                           //
public:                                                      //
  typedef typename E::Key K;                                 //
  typedef typename E::Value V;                               //
  class Iterator;                                            //
public:                                                      //
  SearchTree();                                              //
  int size() const;                                          //
  bool empty() const;                                        //
  Iterator find(const K& k);                                 //
  Iterator insert(const K& k, const V& x);                   //
  void erase(const K& k) throw(NonexistentElement);
  void erase(const Iterator& p);                             //
  Iterator begin();                                          //
  Iterator end();                                            //
protected:                                                   //
  typedef BinaryTree<E> BinaryTree;                          //
  typedef typename BinaryTree::Position TPos;                //
  TPos root() const;                                         //
  TPos finder(const K& k, const TPos& v);                    //
  TPos inserter(const K& k, const V& x);                     //
  TPos eraser(TPos& v);                                      //
  TPos restructure(const TPos& v)                            //
      throw(BoundaryViolation);
private:                                                     //
  BinaryTree T;                                              //
  int n;                                                     //
public:
  // ...insert Iterator class declaration here
};
```

```
class Iterator {
private:
  TPos v;
public:
  Iterator(const TPos& vv) : v(vv) { }
  const E& operator*() const { return *v; }
  E& operator*() { return *v; }
  bool operator==(const Iterator& p) const
    { return v == p.v; }
  Iterator& operator++();
  friend class SearchTree;
};
```

```
Iterator& Iterator::operator++() {
  TPos w = v.right();
  if (w.isInternal()) {
    do { v = w; w = w.left(); }
    while (w.isInternal());
  }
  else {
    w = v.parent();
    while (v == w.right())
      { v = w; w = w.parent(); }
    v = w;
  }
  return *this;
}
```

```
SearchTree() : T(), n(0)
  { T.addRoot(); T.expandExternal(T.root()); }
```

```
TPos root() const
  { return T.root().left(); }
```

```
Iterator begin() {
  TPos v = root();
  while (v.isInternal()) v = v.left();
  return Iterator(v.parent());
}
```

```
Iterator end()
  { return Iterator(T.root()); }
```

C++ Implementation of BST

```
Iterator find(const K& k) {
  TPos v = finder(k, root());
  if (v.isInternal()) return Iterator(v);
  else return end();
}
```

```
TPos finder(const K& k, const TPos& v) {
  if (v.isExternal()) return v;
  if (k < v->key()) return finder(k, v.left());
  else if (v->key() < k) return finder(k, v.right());
  else return v;
}
```

```
Iterator insert(const K& k, const V& x)
  { TPos v = inserter(k, x); return Iterator(v); }
```

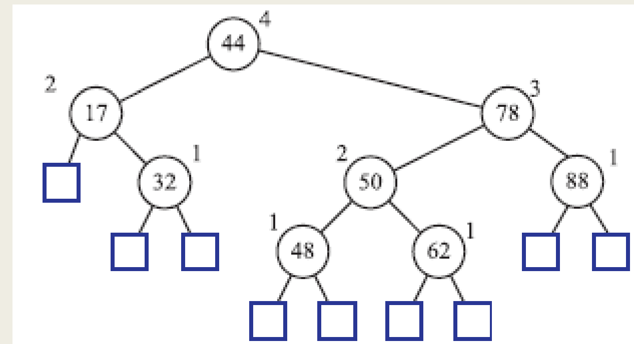
```
TPos inserter(const K& k, const V& x) {
  TPos v = finder(k, root());
  while (v.isInternal())
    v = finder(k, v.right());
  T.expandExternal(v);
  v->setKey(k); v->setValue(x);
  n++;
  return v;
}
```

```
void erase(const K& k) throw(NonexistentElement) {
  TPos v = finder(k, root());                          // sea
  if (v.isExternal())                                  // not
    throw NonexistentElement("Erase of nonexistent");
  eraser(v);                                           // rem
}
```

```
TPos eraser(TPos& v) {
  TPos w;
  if (v.left().isExternal()) w = v.left();             //
  else if (v.right().isExternal()) w = v.right();      //
  else {                                               //
    w = v.right();                                     //
    do { w = w.left(); } while (w.isInternal());       //
    TPos u = w.parent();
    v->setKey(u->key()); v->setValue(u->value());
  }
  n--;                                                 //
  return T.removeAboveExternal(w);                     //
}
```

AVL Trees

- Height-Balance Property
  - *For every internal node v of T, the heights of the children of v differ by at most 1.*
- AVL tree
  - *BST with the height-balance property*
  - *O(log n) even in the worst case*
- Insertion
  - *When unbalanced, restructuring*



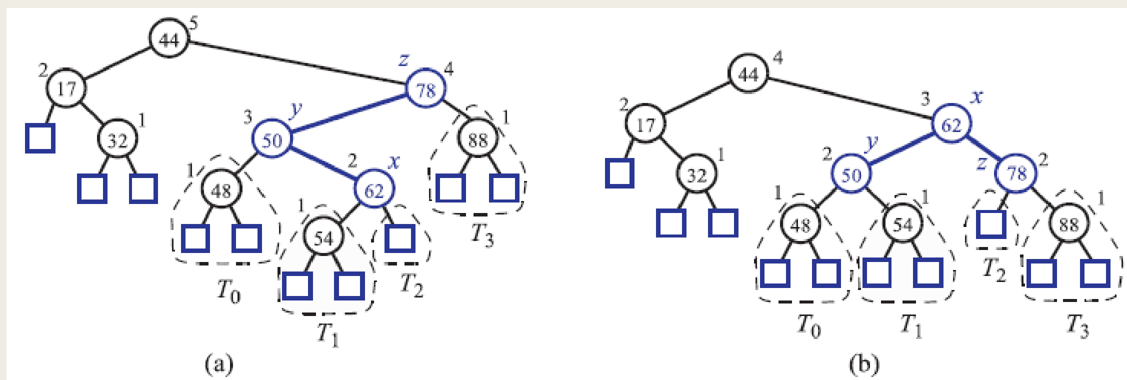


Figure 10.9: An example insertion of an entry with key 54 in the AVL tree of Figure 10.8: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes next to them, and we identify the nodes x, y, and z participating in the trinode restructuring.

AVL Trees: restructuring

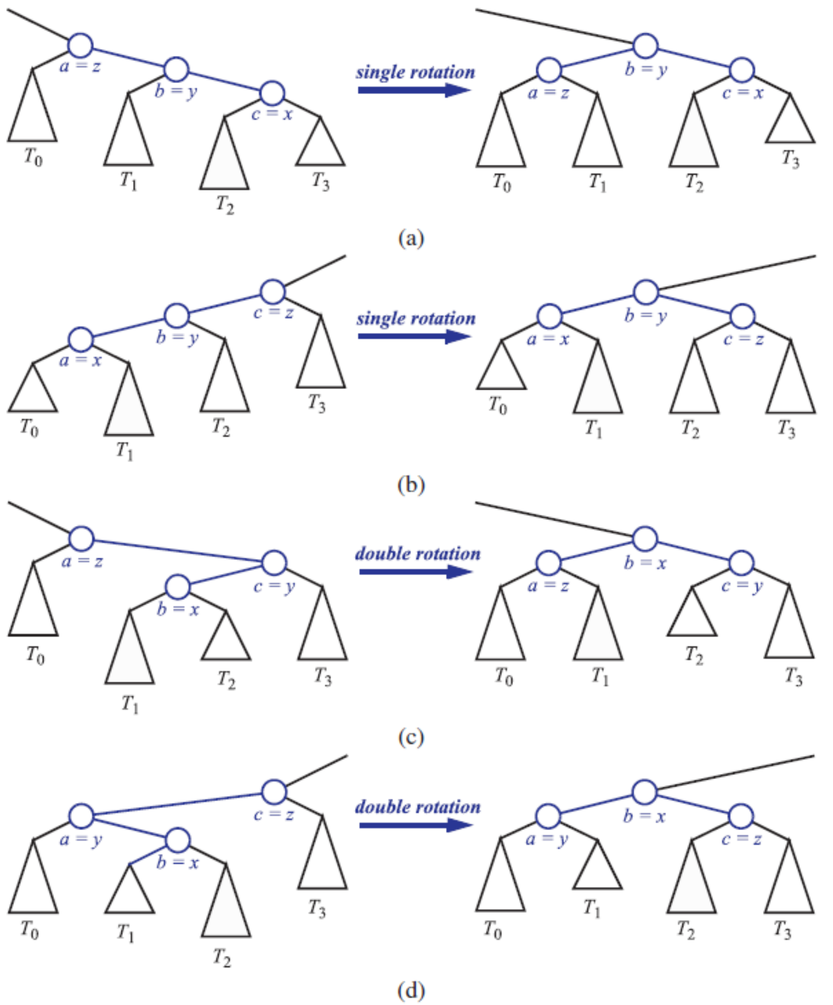


Figure 10.10: Schematic illustration of a trinode restructuring operation (Code Fragment 10.12): (a) and (b) a single rotation; (c) and (d) a double rotation.

Algorithm restructure(x):

Input: A node x of a binary search tree T that has both a parent y and a grand-parent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T_0, T_1, T_2, T_3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Let a be the left child of b and let T_0 and T_1 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T_2 and T_3 be the left and right subtrees of c, respectively.

Code Fragment 10.12: The trinode restructuring operation in a binary search tree.

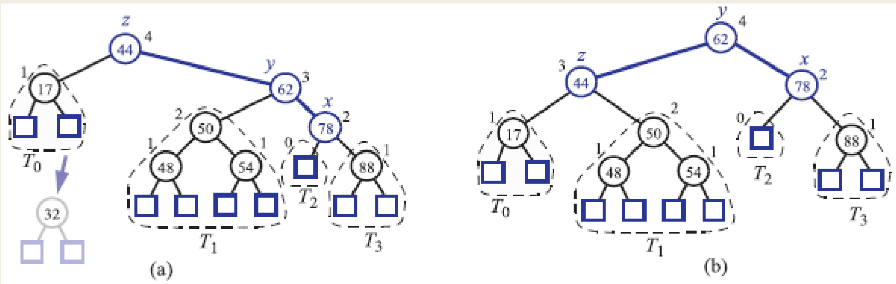


Figure 10.11: Removal of the entry with key 32 from the AVL tree of Figure 10.8: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

C++ Implementation of AVL Trees

```
template <typename E>
class AVLEntry : public E {                          /
private:
  int ht;                                            /
protected:                                           /
  typedef typename E::Key K;                         /
  typedef typename E::Value V;                       /
  int height() const { return ht; }                  /
  void setHeight(int h) { ht = h; }                  /
public:                                              /
  AVLEntry(const K& k = K(), const V& v = V())
    : E(k,v), ht(0) { }
  friend class AVLTree<E>;                           /
};
```

```
bool isBalanced(const TPos& v) const {
  int bal = height(v.left()) − height(v.right());
  return ((−1 <= bal) && (bal <= 1));
}
```

```
TPos tallGrandchild(const TPos& z) const {
  TPos zl = z.left();
  TPos zr = z.right();
  if (height(zl) >= height(zr))
    if (height(zl.left()) >= height(zl.right()))
      return zl.left();
    else
      return zl.right();
  else
    if (height(zr.right()) >= height(zr.left()))
      return zr.right();
    else
      return zr.left();
}
```

```
template <typename E>                                          // a
class AVLTree : public SearchTree< AVLEntry<E> > {
public:                                                        // p
  typedef AVLEntry<E> AVLEntry;                                // a
  typedef typename SearchTree<AVLEntry>::Iterator It
protected:                                                     // l
  typedef typename AVLEntry::Key K;                            // a
  typedef typename AVLEntry::Value V;                          // a
  typedef SearchTree<AVLEntry> ST;                             // a
  typedef typename ST::TPos TPos;                              // a
public:                                                        // p
  AVLTree();                                                   // c
  Iterator insert(const K& k, const V& x);                     // i
  void erase(const K& k) throw(NonexistentElement);            /
  void erase(const Iterator& p);                               // r
protected:                                                     // u
  int height(const TPos& v) const;                             // n
  void setHeight(TPos v);                                      // s
  bool isBalanced(const TPos& v) const;                        // is
  TPos tallGrandchild(const TPos& v) const;                    // g
  void rebalance(const TPos& v);                               // r
};
```

```
int height(const TPos& v) const
  { return (v.isExternal() ? 0 : v−>height()); }
```

```
void setHeight(TPos v) {
  int hl = height(v.left());
  int hr = height(v.right());
  v−>setHeight(1 + std::max(hl, hr));
}
```

```
void rebalance(const TPos& v) {
  TPos z = v;
  while (!(z == ST::root())) {
    z = z.parent();
    setHeight(z);
    if (!isBalanced(z)) {
      TPos x = tallGrandchild(z);
      z = restructure(x);
      setHeight(z.left());
      setHeight(z.right());
      setHeight(z);
    }
  }
}
```

```
Iterator insert(const K& k, const V& x) {
  TPos v = inserter(k, x);
  setHeight(v);
  rebalance(v);
  return Iterator(v);
}
```

```
void erase(const K& k) throw(NonexistentElement) {
  TPos v = finder(k, ST::root());                    // find
  if (Iterator(v) == ST::end())                       // not
    throw NonexistentElement("Erase of nonexistent");
  TPos w = eraser(v);                                 // rem
  rebalance(w);                                       // reb
}
```

Splay Trees

- Splaying is performed (move a specific node to the root by zig-zig, zig-zag, or zig) when
  1. *Searching for k: splay the node*
  2. *Inserting k: splay the new node*
  3. *Deleting k: splay its parent node*
- Splaying Trees
  – *O(log n) on average*
  – *O(n) in the worst case*

(2,4) Trees

- (2,4) tree: a balanced multi-way search tree having two properties.
  - ***Size property:*** *every internal node has 2, 3, or 4 children (1, 2, or 3 keys).*
  - ***Depth property:*** *all the external nodes have the same depth*
- Each internal node maintains an ordered map to store the ordered list of keys with the corresponding values and children nodes.
- Insertion may incur an *overflow*, need to perform *split*

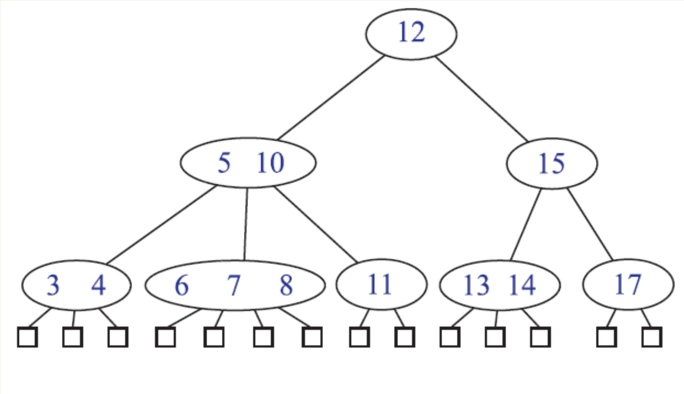


Figure 10.21: A (2,4) tree.

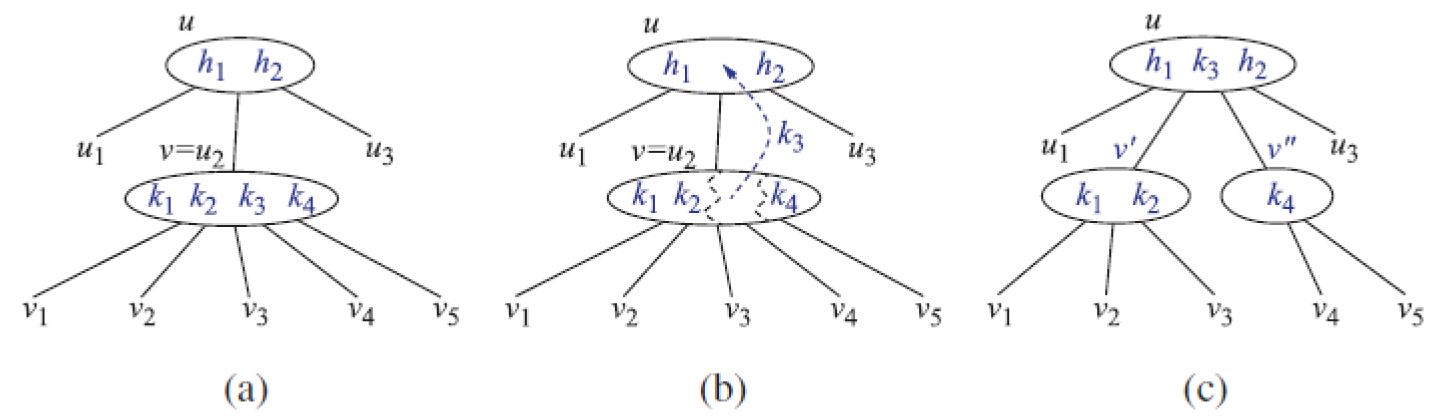
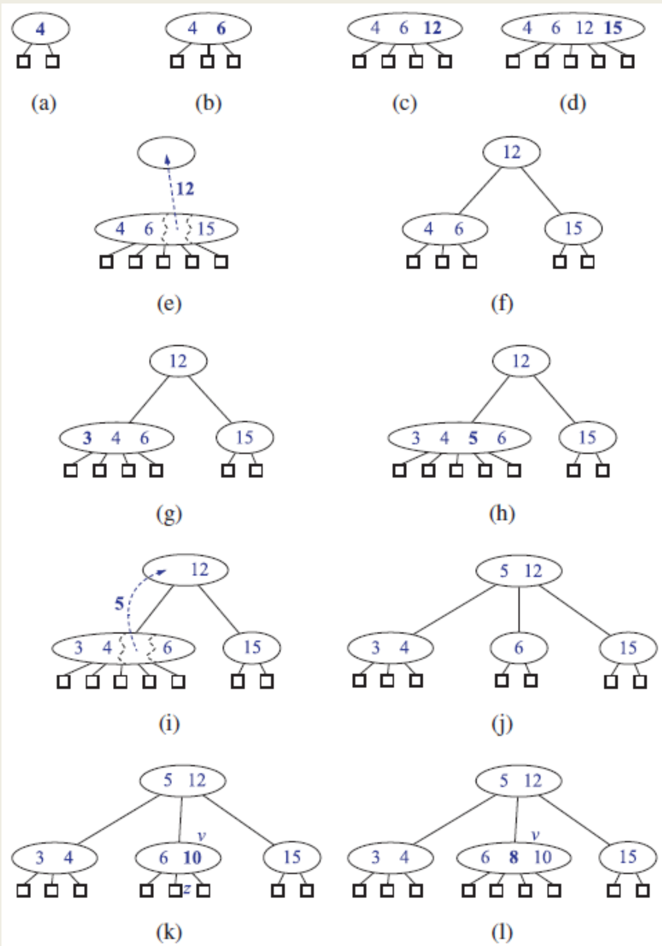


Figure 10.22: A node split: (a) overflow at a 5-node v; (b) the third key of v inserted into the parent u of v; (c) node v replaced with a 3-node v' and a 2-node v''.

(2,4) Trees: Insertion



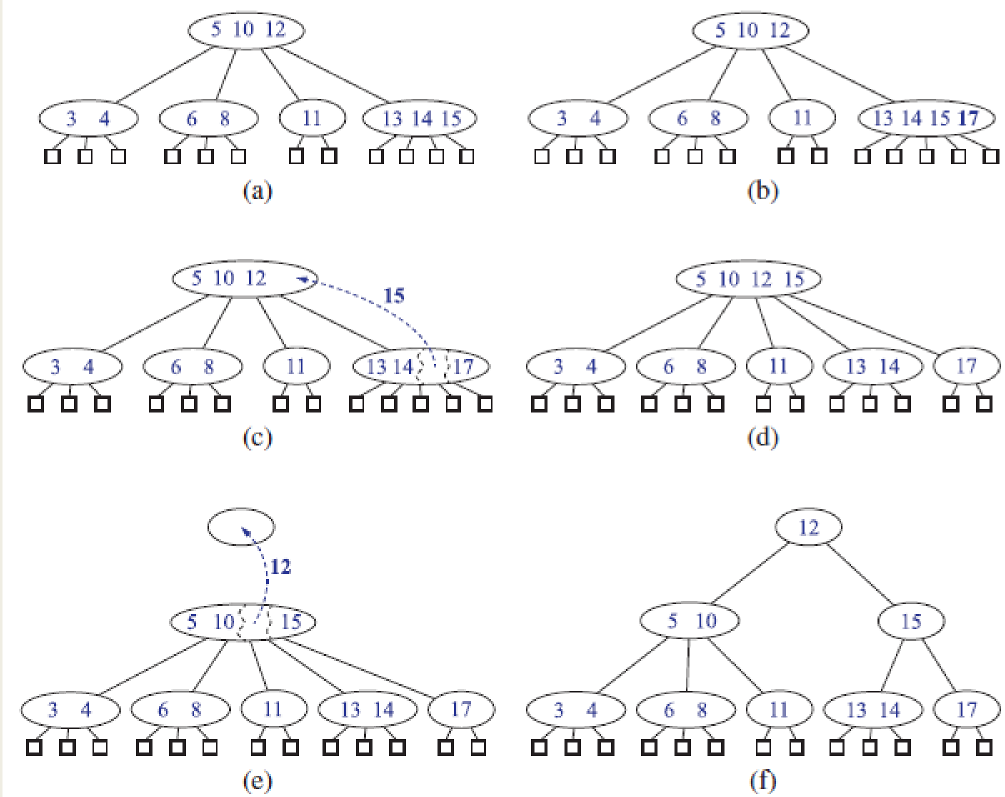
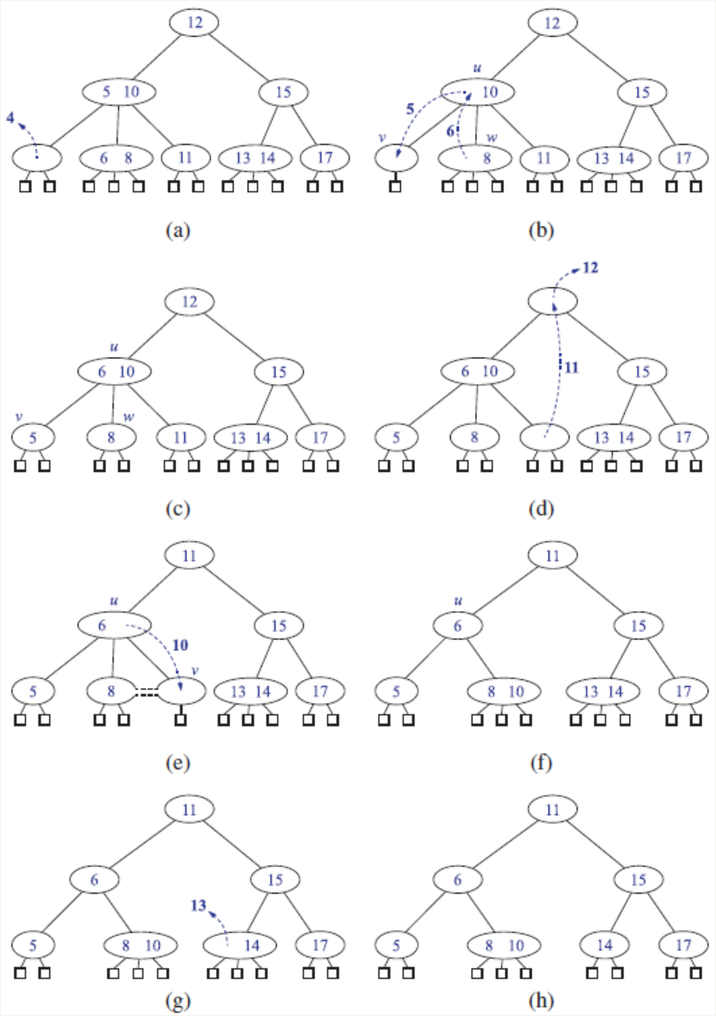


Figure 10.24: An insertion in a (2,4) tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

(2,4) Trees: Removal

- Underflow, Transfer, Fusion



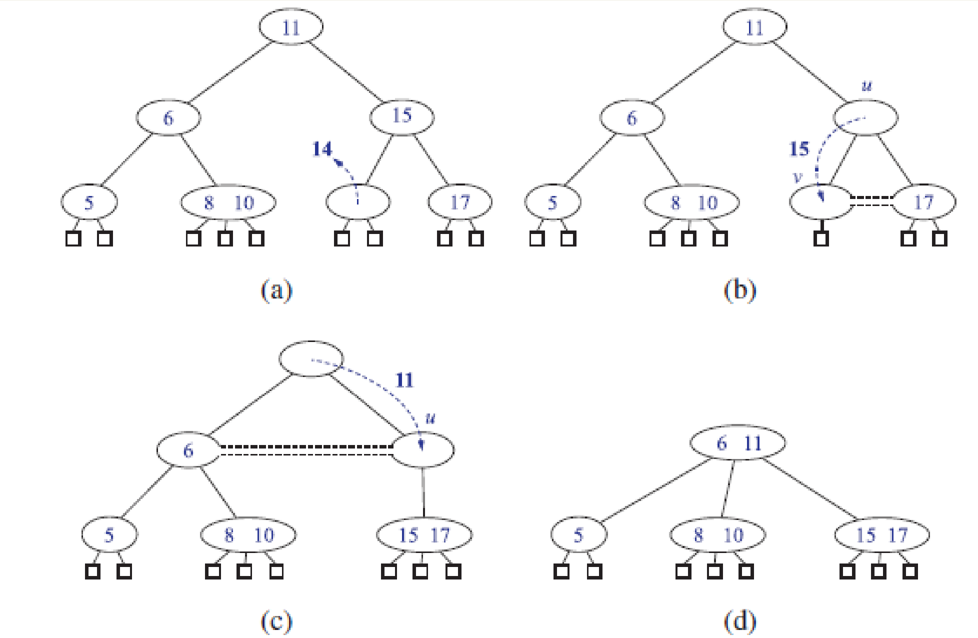
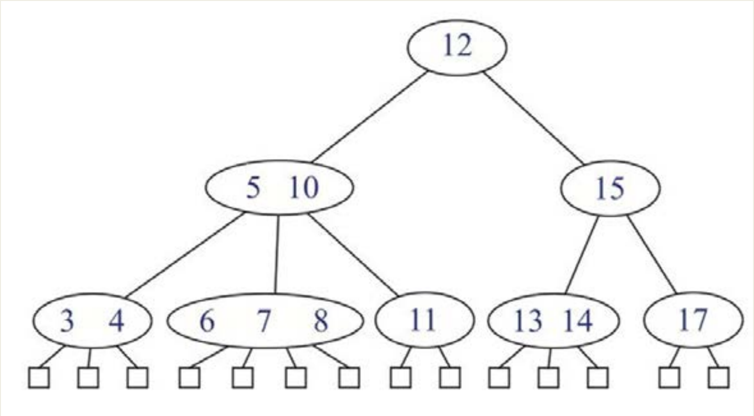


Figure 10.26: A propagating sequence of fusions in a (2,4) tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

Red-Black Trees

- Red-black tree
  - *Root property: The root is black.*
  - *Internal property: The children of a red node are black.*
  - *Depth property: All the external nodes have the same black depth.*



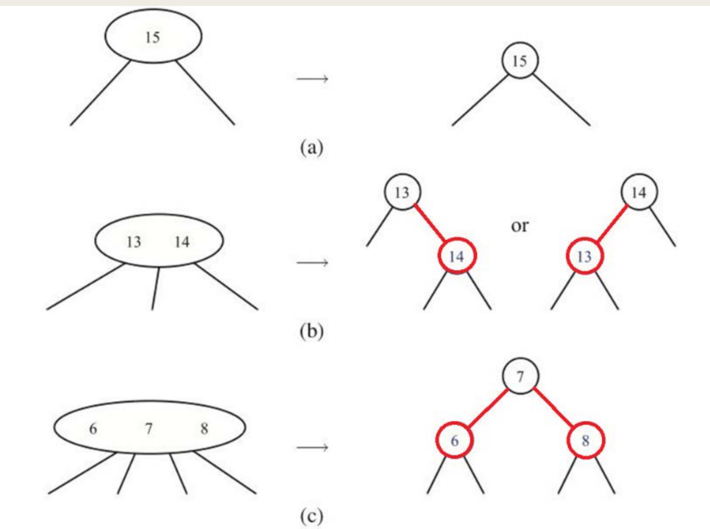
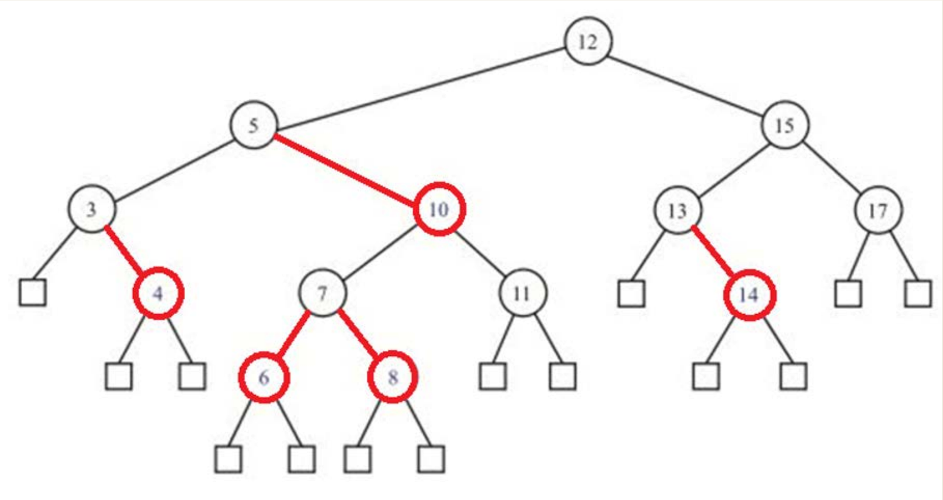


Figure 10.28: Correspondence between a $(2,4)$ tree and a red-black tree: (a) 2-node; (b) 3-node; (c) 4-node.



Red-Black Trees: Insertion

- Insert z into an external node and color z red. What if z's parent is red?
    1. *The sibling w of v is black*
    2. *The sibling w of v is red*

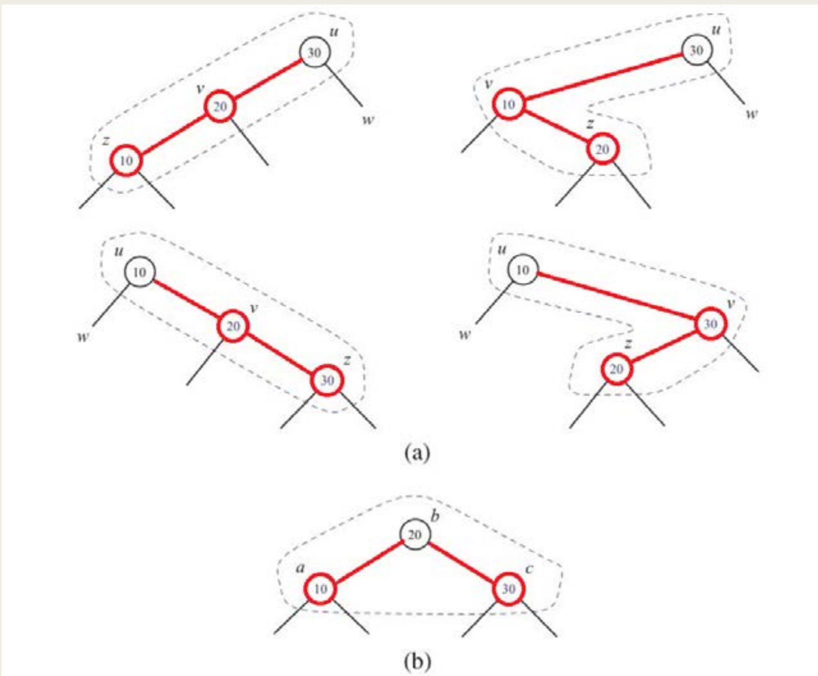


Figure 10.29: Restructuring a red-black tree to remedy a double red: (a) the four configurations for u, v, and z before restructuring; (b) after restructuring.

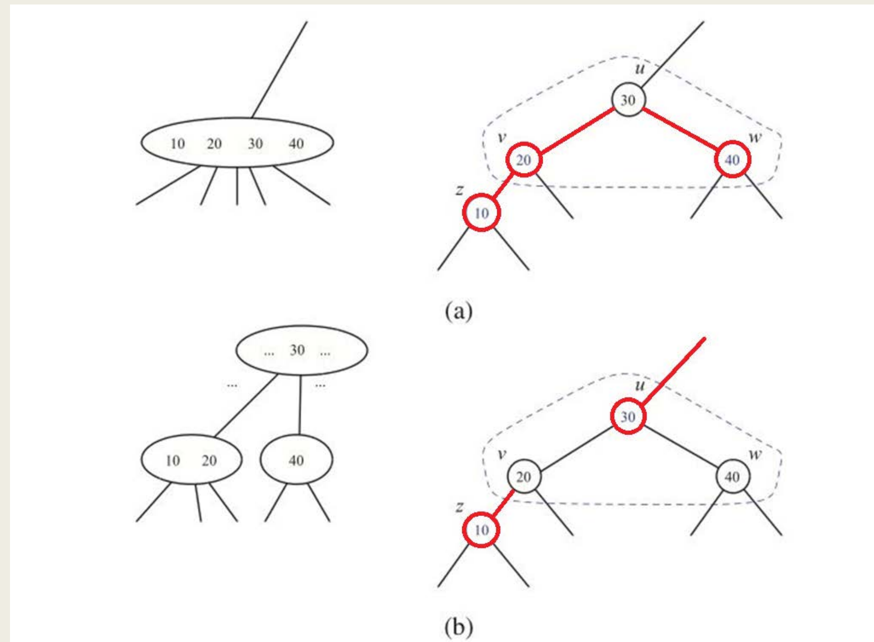


Figure 10.30: Recoloring to remedy the double red problem: (a) before recoloring and the corresponding 5-node in the associated $(2,4)$ tree before the split; (b) after the recoloring (and corresponding nodes in the associated $(2,4)$ tree after the split).

Red-Black Trees: Insertion

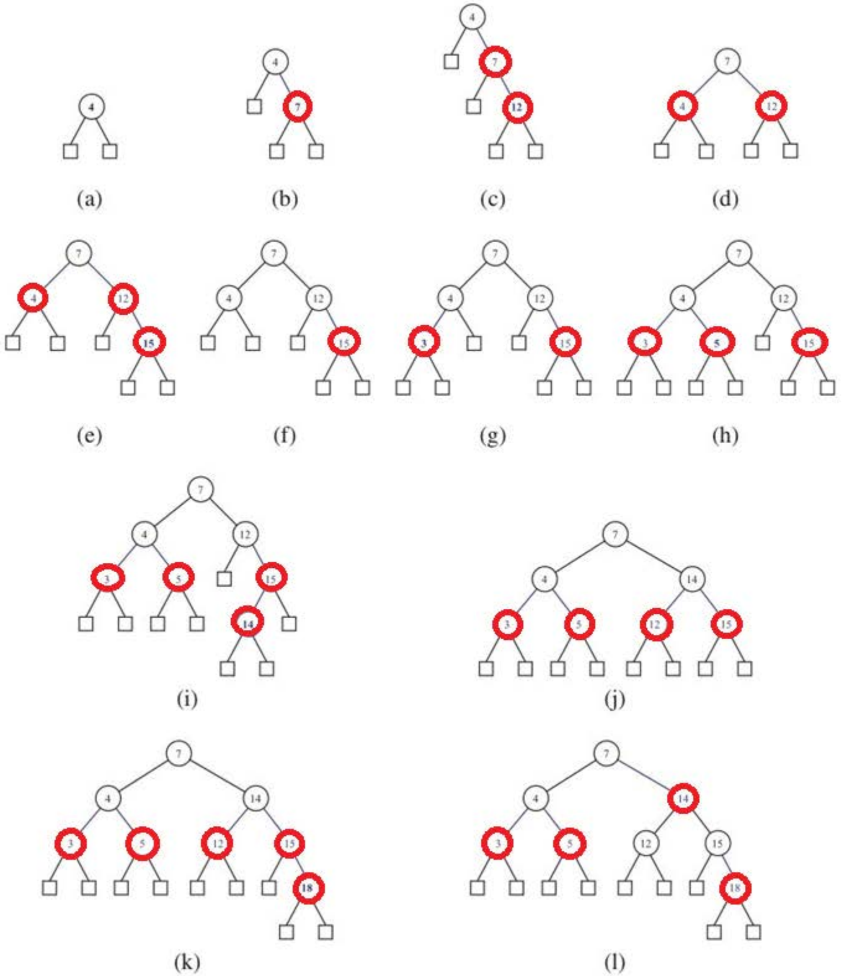


Figure 10.31: A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring. (Continues in Figure 10.32.)

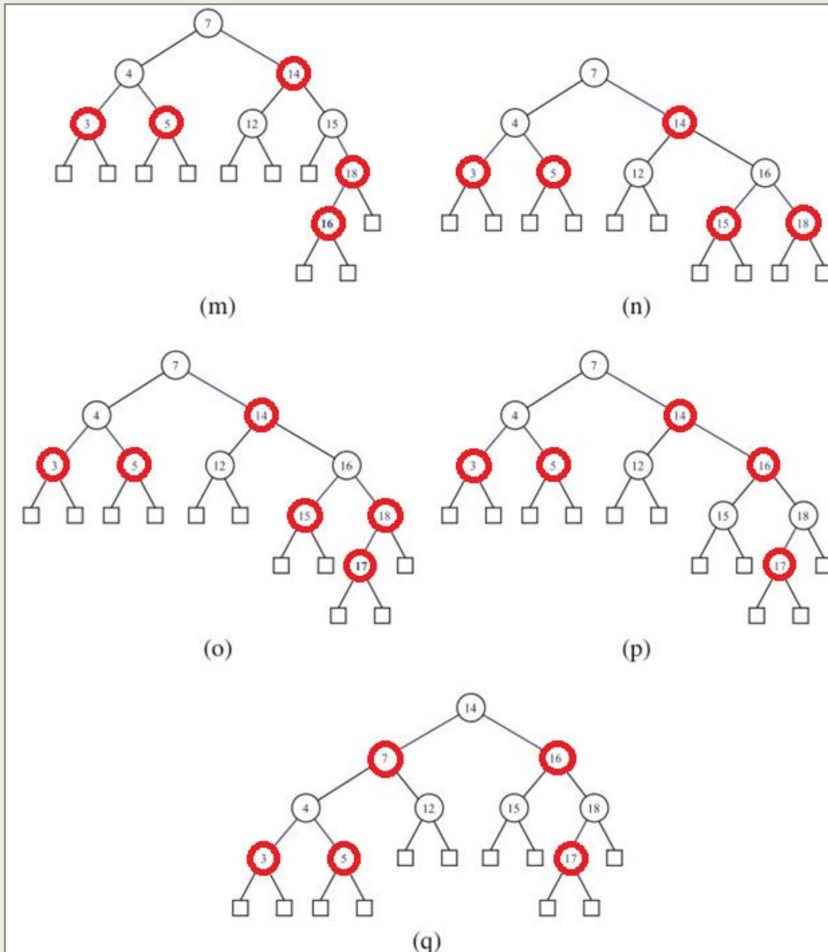


Figure 10.32: A sequence of insertions in a red-black tree: (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring. (Continued from Figure 10.31.)

Red-Black Trees: Removal

- Double black problem: If a black node v is deleted when its child node r is also black, r becomes double-black to preserve the depth property
    1. *The sibling y of r is black and has a red child z*
    2. *The sibling y of r is black and has black children*
    3. *The sibling y of r is red*

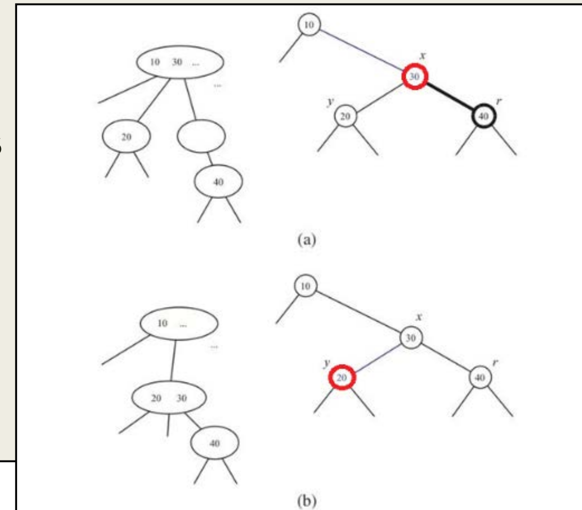


Figure 10.34: Recoloring of a red-black tree that fixes the double black problem: (a) before the recoloring and corresponding nodes in the associated (2,4) tree before the fusion (other similar configurations are possible); (b) after the recoloring and corresponding nodes in the associated (2,4) tree after the fusion.

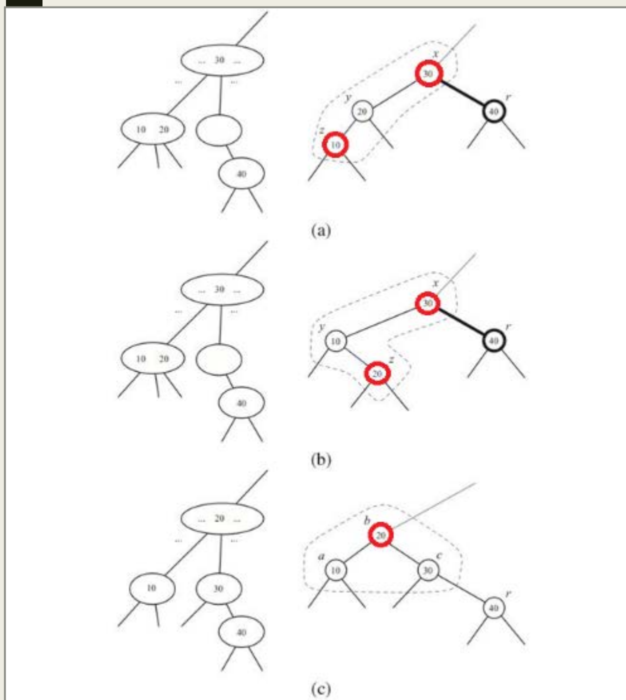


Figure 10.33: Restructuring of a red-black tree to remedy the double black problem: (a) and (b) configurations before the restructuring, where r is a right child and the associated nodes in the corresponding (2,4) tree before the transfer (two other symmetric configurations where r is a left child are possible); (c) configuration after the restructuring and the associated nodes in the corresponding (2,4) tree after the transfer. The grey color for node x in parts (a) and (b) and for node b in part (c) denotes the fact that this node may be colored either red or black.

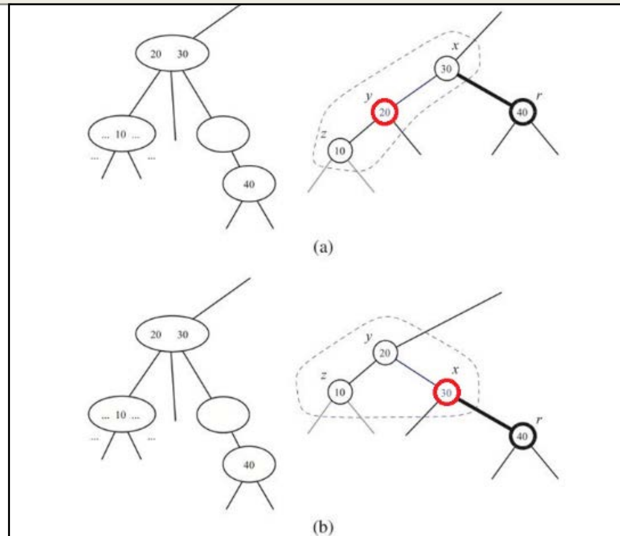


Figure 10.36: Adjustment of a red-black tree in the presence of a double black problem: (a) configuration before the adjustment and corresponding nodes in the associated (2,4) tree (a symmetric configuration is possible); (b) configuration after the adjustment with the same corresponding nodes in the associated (2,4) tree.

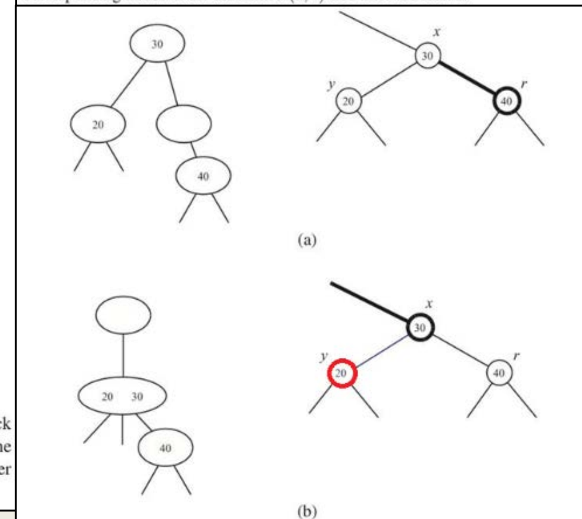


Figure 10.35: Recoloring of a red-black tree that propagates the double black problem: (a) configuration before the recoloring and corresponding nodes in the associated (2,4) tree before the fusion (other similar configurations are possible); (b) configuration after the recoloring and corresponding nodes in the associated (2,4) tree after the fusion.

Red-Black Trees: Removal

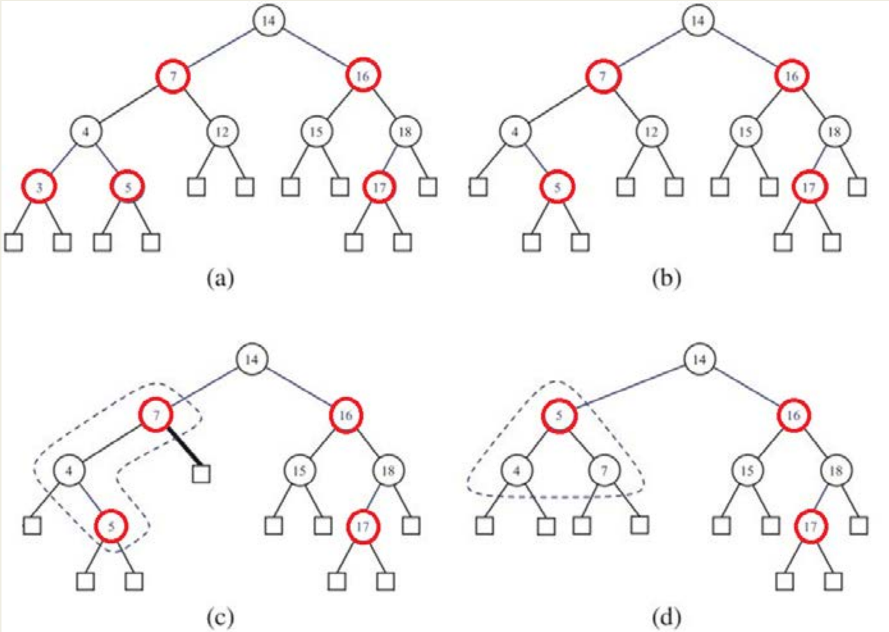


Figure 10.37: Sequence of removals from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a double black (handled by restructuring); (d) after restructuring. (Continues in Figure 10.38.)

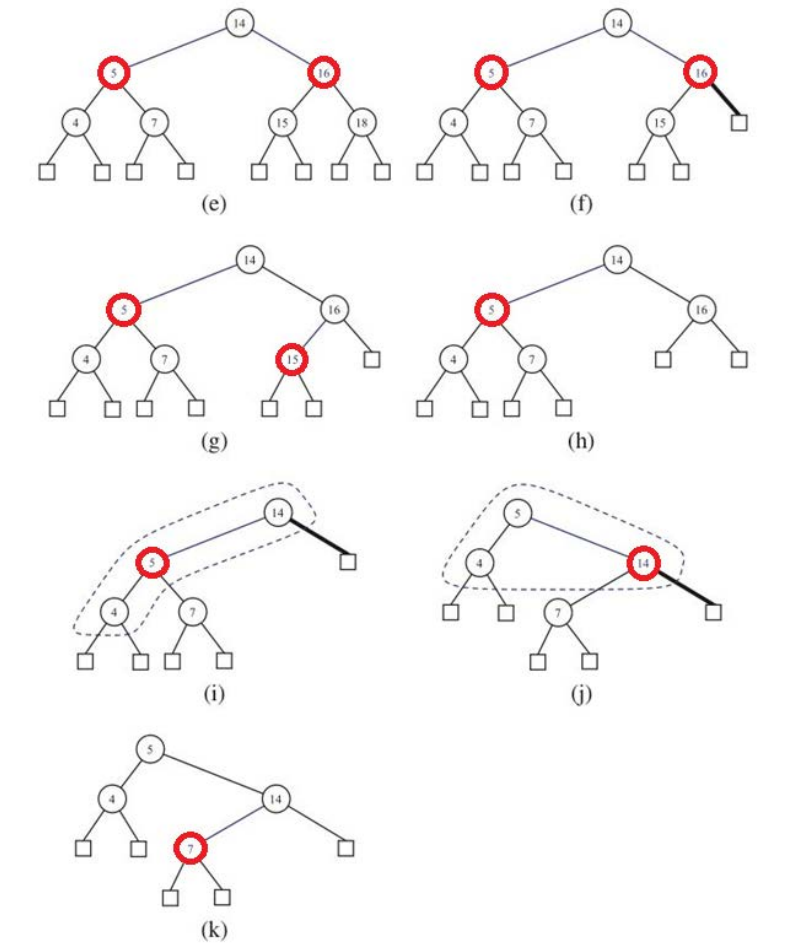


Figure 10.38: Sequence of removals in a red-black tree : (e) removal of 17; (f) removal of 18, causing a double black (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a double black (handled by an adjustment); (j) after the adjustment the double black needs to be handled by a recoloring; (k) after the recoloring. (Continued from Figure 10.37.)