アルゴリズムの設計と解析

教授:　黄　潤和　（W4022）

rhuang@hosei.ac.jp

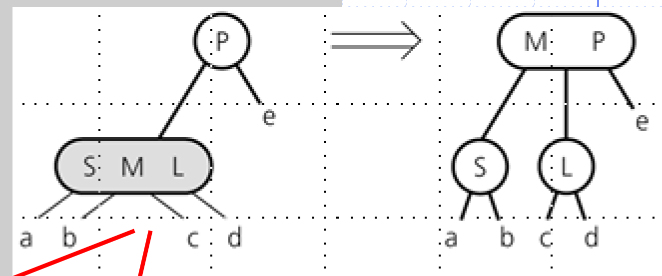
SA:　広野　史明（A4/A8）

fumiaki.hirono.5k@stu.hosei.ac.jp

Contents (L6 – Search trees)

- Searching problems
- AVL tree
- 2-3-4 trees
  - Insertion (review)
  - Deletion

Insertion in 2-3-4 trees

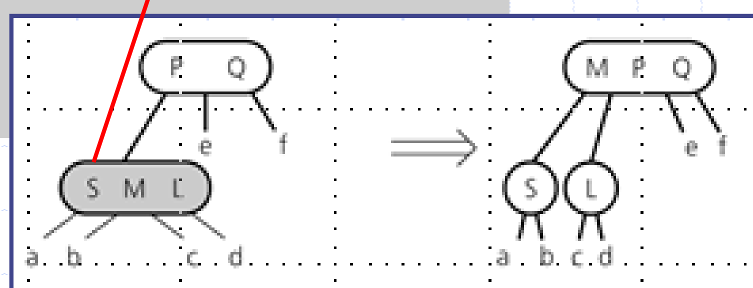


Step 1 Search for the item to be inserted (same as in 2-3 trees).
Step 2 Insert at the leaf level. The following cases are possible:

- The termination node is a 2-node. Then, make it a 3-node, and insert the new item appropriately.
- The termination node is a 3-node. Then, make it a 4-node, and insert the new item appropriately.
- The termination node is a 4 node. Split is, pass the middle to the parent, and insert the new item appropriately.

General rules for inserting new nodes in 2-3-4 trees:
Rule 1: During the search step, every time a 2-node connected to a 4-node is encountered, transform it into a 3-node connected to two 2-nodes.
Rule 2: During the search step, every time a 3-node connected to a 4-node is encountered, transform it into a 4-node connected to two 2-nodes.

Note that two 2-nodes resulting from these transformations have the same number of children as the original 4-node. This is why the split of a 4-node does not affect any nodes below the level where the split occurs.



Insertion in 2-3-4 trees

1. **挿入するノードを見つける**
   - 挿入する値より小さい値は左側、大きい値は右側に位置するような葉ノードを見つける
2. **挿入の準備をする**
   - 2ノードもしくは木全体でアイテムがなければなにもしない
   - 3ノードならば
     - 親ノードのアイテムが2つ以下ならばなにもしない
     - 親ノードのアイテムが3つならば挿入前に親ノードの分裂を行なう
   - 4ノードならば挿入前に分裂を行なう
3. **挿入する**

- **分裂**
  - 根ノードであれば、真ん中の値を新たに根ノードとし、残りの2つの値はそれぞれその子ノードとなる
  - 根ノードでなければ、真ん中の値を親ノードに移動し、残りの2つの値はそれぞれその親ノードの子ノードとなる

2-3-4 tree demo

http://www.cs.unm.edu/~rlpm/499/ttft.html

2-3-4-tree.jar

de-jar jd-gui-1.4.0.jar

http://www.unf.edu/~broggio/cop3540/chap10/Tree234App.java

Compare the Python code with the Java code

Step 2 Insert at the leaf level. The following cases are possible:

- The termination node is a 2-node. Then, make it a 3-node, and insert the new item appropriately.
- The termination node is a 3-node. Then, make it a 4-node, and insert the new item appropriately.
- The termination node is a 4 node. Split is, pass the middle to the parent, and insert the new item appropriately.

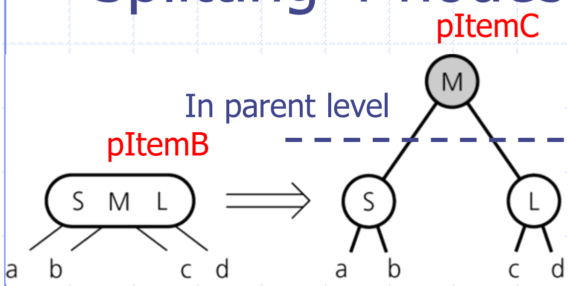
```python
while True:
    if pCurNode.isFull():   #if node full,
        self.split(pCurNode)    #split it
        pCurNode = pCurNode.getParent() #back up
                #search once
        pCurNode = self.getNextChild(pCurNode, dValue)
    #end if(node is full)

    elif pCurNode.isLeaf(): #if node is leaf,
        break   #go insert
    #node is not full, not a leaf; so go to lower level
    else:
        pCurNode = self.getNextChild(pCurNode, dValue)
#end while
```

```java
while(true)
   {
   if( curNode.isFull() )                 // if node full,
      {
      split(curNode);                     // split it
      curNode = curNode.getParent();      // back up
                                          // search once
      curNode = getNextChild(curNode, dValue);
      }  // end if(node is full)

   else if( curNode.isLeaf() )            // if node is leaf,
      break;                              // go insert
   // node is not full, not a leaf; so go to lower level
   else
      curNode = getNextChild(curNode, dValue);
   }  // end while
```

Splitting 4-nodes




```
def split(self, pThisNode):     #split the node
        #assumes node is full

        pItemC = pThisNode.removeItem() #remove items from
        pItemB = pThisNode.removeItem() #this node
        pChild2 = pThisNode.disconnectChild(2)  #remove children
        pChild3 = pThisNode.disconnectChild(3)  #from this node

        pNewRight = Node()      #make new node

        if pThisNode == self._pRoot:    #if this is the root,
                self._pRoot = Node()    #make new root
                pParent = self._pRoot   #root is our parent
                self._pRoot.connectChild(0, pThisNode)  #connect to parent
        else:   #this node not the root
                pParent = pThisNode.getParent() #get parent

        #deal with parent
        itemIndex = pParent.insertItem(pItemB)  #item B to parent
        n = pParent.getNumItems()       #total items?

        j = n-1#move parent's
        while j > itemIndex:    #connections
                pTemp = pParent.disconnectChild(j)      #one child
                pParent.connectChild(j+1, pTemp)        #to the right
                j -= 1
                      #connect newRight to parent
        pParent.connectChild(itemIndex+1, pNewRight)

        #deal with newRight
        pNewRight.insertItem(pItemC)    #item C to newRight
        pNewRight.connectChild(0, pChild2)      #connect to 0 and 1
        pNewRight.connectChild(1, pChild3)      #on newRight
#end split()
```

```
public void split(Node thisNode)     // split the node
   {
   // assumes node is full
   DataItem itemB, itemC;
   Node parent, child2, child3;
   int itemIndex;

   itemC = thisNode.removeItem();    // remove items from
   itemB = thisNode.removeItem();    // this node
   child2 = thisNode.disconnectChild(2); // remove children
   child3 = thisNode.disconnectChild(3); // from this node

   Node newRight = new Node();       // make new node

   if(thisNode==root)                // if this is the root,
      {
      root = new Node();                // make new root
      parent = root;                    // root is our parent
      root.connectChild(0, thisNode);   // connect to parent
      }
   else                              // this node not the root
      parent = thisNode.getParent();    // get parent

   // deal with parent
   itemIndex = parent.insertItem(itemB); // item B to parent
   int n = parent.getNumItems();         // total items?

   for(int j=n-1; j>itemIndex; j--)          // move parent's
      {                                      // connections
      Node temp = parent.disconnectChild(j); // one child
      parent.connectChild(j+1, temp);        // to the right
      }
                                 // connect newRight to parent
   parent.connectChild(itemIndex+1, newRight);

   // deal with newRight
   newRight.insertItem(itemC);       // item C to newRight
   newRight.connectChild(0, child2); // connect to 0 and 1
   newRight.connectChild(1, child3); // on newRight
   }  // end split()
```

2-3-4 Tree implementation in Java

For the complete source code, please see the above file. You can refer to the web link below as well.

http://www.unf.edu/~broggio/cop3540/chap10/Tree234App.java

Objects of this Class Represent Data Items Actually Stored.

```
// tree234.java

import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
{
  public int  dData;          // one data item
//--------------------------------------------------------------
  public DataItem(int dd)    // constructor
    { dData = dd; }

//--------------------------------------------------------------
  public void displayItem()   // display item, format "/27"
    { System.out.print("/"+dData); }

//--------------------------------------------------------------
  }  // end class DataItem
```

This is merely A data item stored at the nodes. In practice, this might be an entire record or object.
Here we are only showing the key.

class Node

```
  private static final int ORDER = 4;
  private int numItems;
  private Node parent;
  private Node childArray[] = new Node[ORDER];
  private DataItem itemArray[] = new DataItem[ORDER-1];
// ------------------------------------------------------------
public void connectChild(int childNum, Node child)  {// connect child to this node
     childArray[childNum] = child;
     if(child != null)
        child.parent = this;
     }
// ------------------------------------------------------------

public Node disconnectChild(int childNum)   {// disconnect child from this node, return it
     Node tempNode = childArray[childNum];
     childArray[childNum] = null;
     return tempNode;
     }
// ------------------------------------------------------------
  public Node getChild(int childNum)
     { return childArray[childNum]; }
// ------------------------------------------------------------
  public Node getParent()
     { return parent; }
// ------------------------------------------------------------
  public boolean isLeaf()
     { return (childArray[0]==null) ? true : false; }
// ------------------------------------------------------------
  public int getNumItems()
    { return numItems; }
```

4

3

Note: two arrays: a **child** array and an **item** array.
Note their size: nodes = 4; item = 3.
The child array is size 4: the links: maximum children.
The second array, itemArray is of size 3 – the maximum number of data items in a node.
Ordered too.

One of three slides of code for class Node

For a given node, note we **store the number of items**
Also, the node's parent is an attribute in the class.

Major work done by findItem(), insertItem() and removeItem() (**next slides)** for a given node.

These are complex routines and NOT to be confused with find() and insert() for the Tree234 class itself.

Recall: references are automatically initialized to null and numbers to 0 when their object is created.
So, Node doesn't need a Constructor.

Class Node (continued)

```
public DataItem getItem(int index)   // get DataItem at index
    { return itemArray[index]; }
// -------------------------------------------------------------
  public boolean isFull()
    { return (numItems==ORDER-1) ? true : false; }
// -------------------------------------------------------------
➔ public int findItem(int  key)       // return index of
    {                                  // item (within node)
    for(int j=0; j<ORDER-1; j++)        // if found,
      {                                // otherwise,
      if(itemArray[j] == null)          // return -1
        break;
      else if(itemArray[j].dData == key)
        return j;
      }
    return -1;
    }  // end findItem
// -------------------------------------------------------------
➔ public DataItem removeItem()  {      // removes largest item
    // assumes node not empty
    DataItem temp = itemArray[numItems-1];  // save item
    itemArray[numItems-1] = null;           // disconnect it
    numItems--;                             // one less item
    return temp;                            // return item
    }
// -------------------------------------------------------------
  public void displayNode()     {      // format "/24/56/74/"
    for(int j=0; j<numItems; j++)
      itemArray[j].displayItem();   // "/56"
    System.out.println("/");        // final "/"
    }
  }  // end class Node
```

Find routine:
Looking for the data within the node where we are located.

Delete Routine
Saves the deleted item.
Sets the location contents to null.
Decrements the number of items at the node.
Returns the deleted data item.

Class Node (continued)

Insert Routine

Increments number of items in node.
Get key of new item.
Now loop.

Go through code and my comments.

Start looking for place to insert the data item. Start on the right and proceed left looking for proper place.

```
// -------------------------------------------------------------
public int insertItem(DataItem newItem)
{
// assumes node is not full
numItems++;                          // will add new item
int newKey = newItem.dData;          // key of new item

for(int j=ORDER-2; j>=0; j--)   // start on right to examine data
  {
  if(itemArray[j] == null)        // if item null, go left one cell.
    continue;
  else
    {                                // if not null, get its key
    int itsKey = itemArray[j].dData;
    if(newKey < itsKey)                    // if it's bigger
      itemArray[j+1] = itemArray[j];       // shift existing key right
    else
      {   // otherwise, insert new item and return index.+1
      itemArray[j+1] = newItem;
      return j+1;
      }
    }  // end else (not null)
  }  // end for                      // shifted all items,
itemArray[0] = newItem;              // insert new item
return 0;
}  // end insertItem()
```

```
class Tree234 {
  private Node root = new Node();    // make root node
  public int find(int key) {
    Node curNode = root;
    int childNumber;
    while(true) {
      if(( childNumber=curNode.findItem(key) ) != -1)
        return childNumber;    // found; recall findItem returns index value
      else if( curNode.isLeaf() )
            return -1;               // can't find it
          else                       // search deeper
            curNode = getNextChild(curNode, key);
    } // end while
  }// end find()
public void insert(int dValue) {  // insert a DataItem
    Node curNode = root;
    DataItem tempItem = new DataItem(dValue);
    while(true)
    {
      if( curNode.isFull() )
      {
        split(curNode);              // call to split node
        curNode = curNode.getParent();   // back up  // search once
        curNode = getNextChild(curNode, dValue);
      } // end if(node is full)
      else if( curNode.isLeaf() )  // if node is leaf, insert data
            break;
          else // node is not full, not a leaf, so go to lower level
            curNode = getNextChild(curNode, dValue);
    } // end while
    curNode.insertItem(tempItem);    // insert new DataItem
```

```
public void split(Node thisNode) {  // split the node    // assumes node is full
  DataItem itemB, itemC;   // When you get here, you know you need to split…
  Node parent, child2, child3;
  int itemIndex;
  itemC = thisNode.removeItem();    // remove items from this node.
  itemB = thisNode.removeItem();    // Note: these are second and third items
  child2 = thisNode.disconnectChild(2); // remove children      These are rightmost
  child3 = thisNode.disconnectChild(3); // from this node        two children
  Node newRight = new Node();       // make new node
  if(thisNode==root)              // if the node we're looking at is the root,
    {
    root = new Node();              // make new root
    parent = root;               // root is our parent
    root.connectChild(0, thisNode);  // connect to parent
    }
  else                      // this node to be split is not the root
    parent = thisNode.getParent();   // get parent
  // deal with parent
  itemIndex = parent.insertItem(itemB); // item B to parent
  int n = parent.getNumItems();        // total items?

  for(int j=n-1; j>itemIndex; j--)
    {                                  // move parent's connections
    Node temp = parent.disconnectChild(j); // one child to the right
    parent.connectChild(j+1, temp);
    }
  parent.connectChild(itemIndex+1, newRight);     // connect newRight to parent
  // deal with newRight
  newRight.insertItem(itemC);      // item C to newRight
  newRight.connectChild(0, child2); // connect to 0 and 1
  newRight.connectChild(1, child3); // on newRight
  } // end split()
```

```
// gets appropriate child of node during search for value
  public Node getNextChild(Node theNode, int theValue)  {
    int j;
    // assumes node is not empty, not full, not a leaf
    int numItems = theNode.getNumItems();
    for(j=0; j<numItems; j++)          // for each item in node
                                                      // are we less?
      if( theValue < theNode.getItem(j).dData )
             return theNode.getChild(j);  // return left child
      // end for                   // we're greater, so
    return theNode.getChild(j);         // return right child
    }// end getNextChild()
// -------------------------------------------------------------
  public void displayTree()
    {    recDisplayTree(root, 0, 0);    }
// -------------------------------------------------------------
  private void recDisplayTree(Node thisNode, int level, int childNumber)   {
    System.out.print("level="+level+"  child="+childNumber+"  ");
    thisNode.displayNode();                // display this node
    // call ourselves for each child of this node
    int numItems = thisNode.getNumItems();
    for(int j=0; j<numItems+1; j++)
      {
      Node nextNode = thisNode.getChild(j);
      if(nextNode != null)
        recDisplayTree(nextNode, level+1, j);
      else
        return;
      }
    }  // end recDisplayTree()
// -------------------------------------------------------------¥
  }  // end class Tree234
```

class Tree234App {

```
public static void main(String[] args) throws IOException {
  long value;
  Tree234 theTree = new Tree234();
  theTree.insert(50);
  theTree.insert(40);
  theTree.insert(60);
  theTree.insert(30);
  theTree.insert(70);
  while(true) {
    System.out.print("Enter first letter of ");
    System.out.print("show, insert, or find: ");
    char choice = getChar();
    switch(choice) {
      case 's':
        theTree.displayTree();
        break;
      case 'i':
        System.out.print("Enter value to insert: ");
        value = getInt();
        theTree.insert(value);
        break;
      case 'f':
        System.out.print("Enter value to find: ");
        value = getInt();
        int found = theTree.find(value);
        if(found != -1)
          System.out.println("Found "+value);
        else
          System.out.println("Could not find "+value);
        break;
      default:
        System.out.print("Invalid entry¥n");
    } // end switch
  } // end while
} // end main()
```

```
//-------------------------------------------------------------
  public static String getString() throws IOException
    {
    InputStreamReader isr = new  InputStreamReader
             (System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
    }
//-------------------------------------------------------------
  public static char getChar() throws IOException
    {
    String s = getString();
    return s.charAt(0);
    }

//-------------------------------------------------------------
  public static int getInt() throws IOException
    {
    String s = getString();
    return Integer.parseInt(s);
    }
//-------------------------------------------------------------
  } // end class Tree234App
```

Analysis of Insertion
挿入の分析

Algorithm *insertItem*(*k*, *o*)

1. We search for key ***k*** to locate the insertion node ***v***

2. We add the new item (***k***, ***o***) at node ***v***

3. **while** *overflow*(*v*)
   if *isRoot*(*v*)
   create a new empty root above ***v***
   v ← ***split***(***v***)

- Let ***T*** be a (2,4) tree with ***n*** items
  n個の値を持つ2-4木、Tで考察。
  - Tree ***T*** has ***O***(log ***n***) height
  - Step 1 takes ***O***(log ***n***) time because we visit ***O***(log ***n***) nodes
  - Step 2 takes ***O***(1) time
  - Step 3 takes ***O***(log ***n***) time because each split takes ***O***(1) time and we perform ***O***(log ***n***) splits
- Thus, an insertion in a (2,4) tree takes ***O***(log ***n***) time

(2,4) tree Deletion Algorithm

(2,4) tree Deletion Algorithm

→**Deleting item *I*: remember**

deletion always begins at a leaf

1. Locate node *n*, which contains item *I*
2. If node *n* is not a leaf → swap *I* with inorder successor

→ deletion always begins at a leaf

3. If leaf node *n* contains another item, just delete item *I* else

From the root down to the leaf
1. find the node in which item *I* is in and then check the following cases
2. if (the node is leaf)
3. delete
4. if (the node is not leaf)
5. swap
6. if (the root and two children are 2-nodes)
7. merge1
8. if (sibling node has 2 or 3 items)
9. right/left rotate
10. if (sibling nodes are 2-nodes)
11. merge2

5. swap:
swap item of internal node with inorder successor

7. merge 1
combine all three elements into the root.

9. right/left rotate:
parent item to the right/left child and the left/right child to its parent

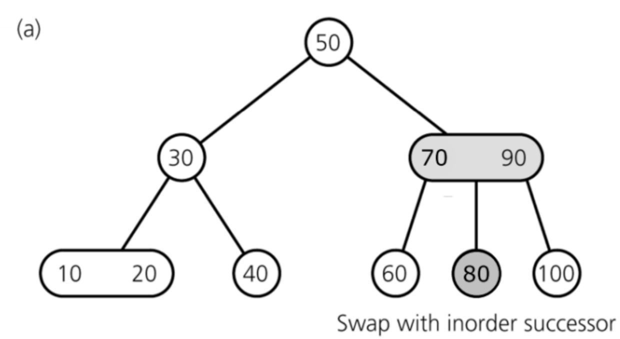
11. merge2
combine all three elements as a child node.

2-3-4 Tree: Deletion

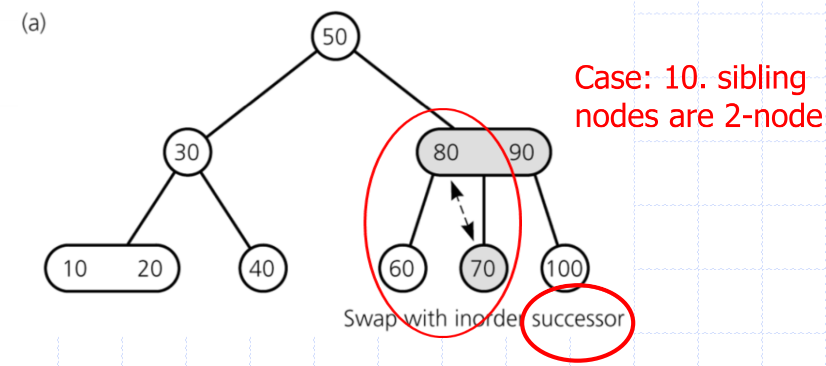
Explain the deletion procedure from examples

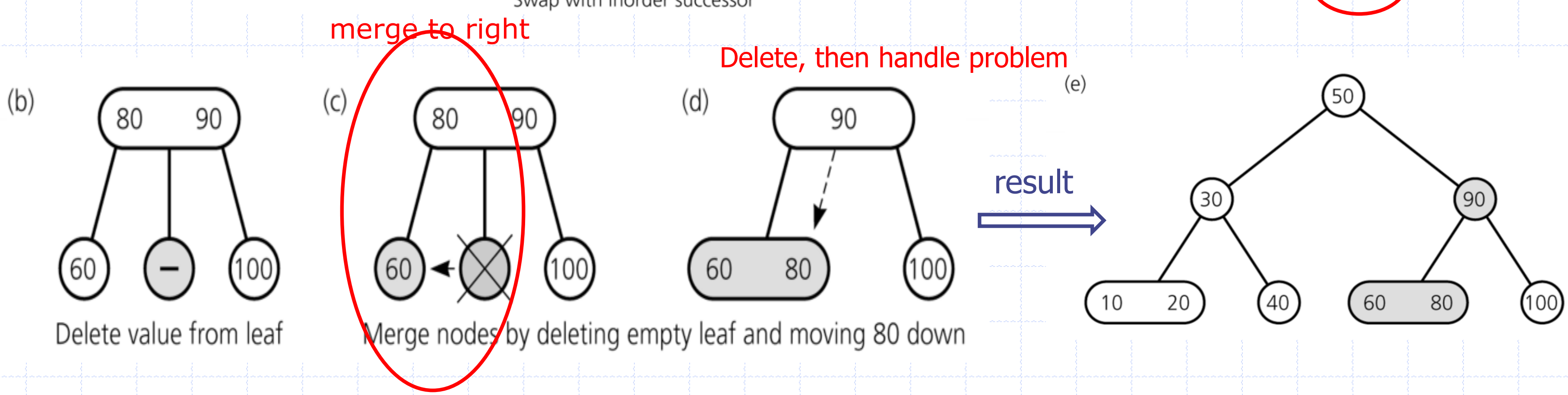
- items are deleted at the leafs
  → swap item of internal node with inorder successor

Delete 70



Deleting 70: swap 70 with inorder successor (80)



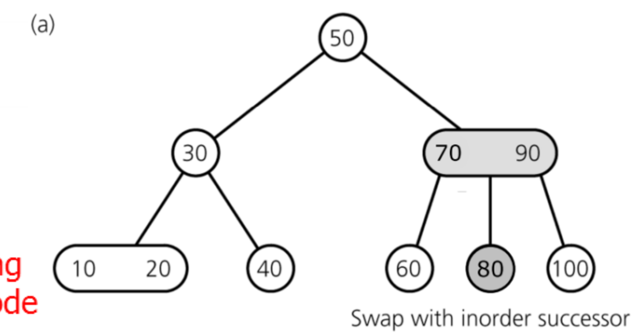


2-3-4 Tree: Deletion

Deletion procedure:

- items are deleted at the leafs
  → swap item of internal node with inorder successor

Delete 70



From the root down to the leaf
1. find the node in which item *I* is in and then check the following cases
2. if (the node is leaf)
3. delete
4. if (the node is not leaf)
5. swap
6. if (the root and two children are 2-nodes)
7. merge1
8. if (sibling node has 2 or 3 items)
9. right/left rotate
10. if (sibling nodes are 2-nodes)
11. merge2

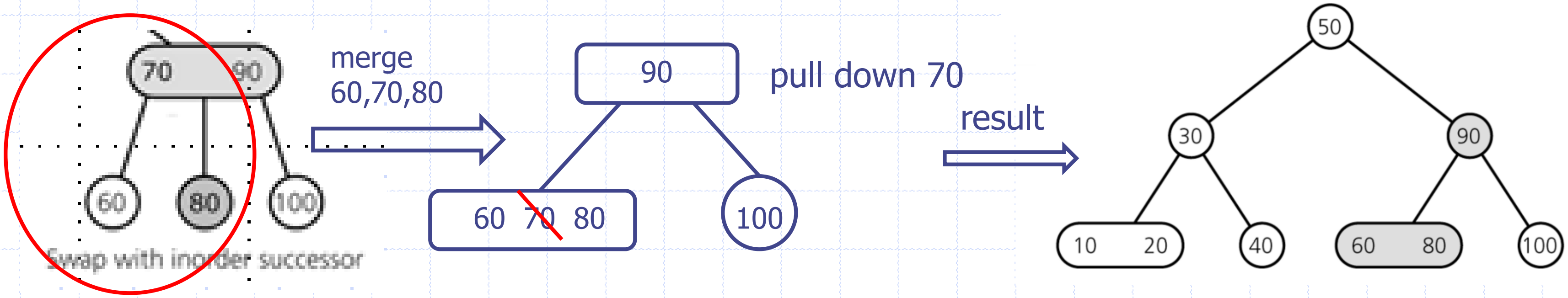
5. swap: swap item of internal node with inorder successor

7. merge 1 combine all three elements into the root.

9. right/left rotate: parent item to the right/left child and the left/right child to its parent

11. merge2 combine all three elements as a child node.

Case: 10. sibling nodes are 2-node



Prevent problem, then delete

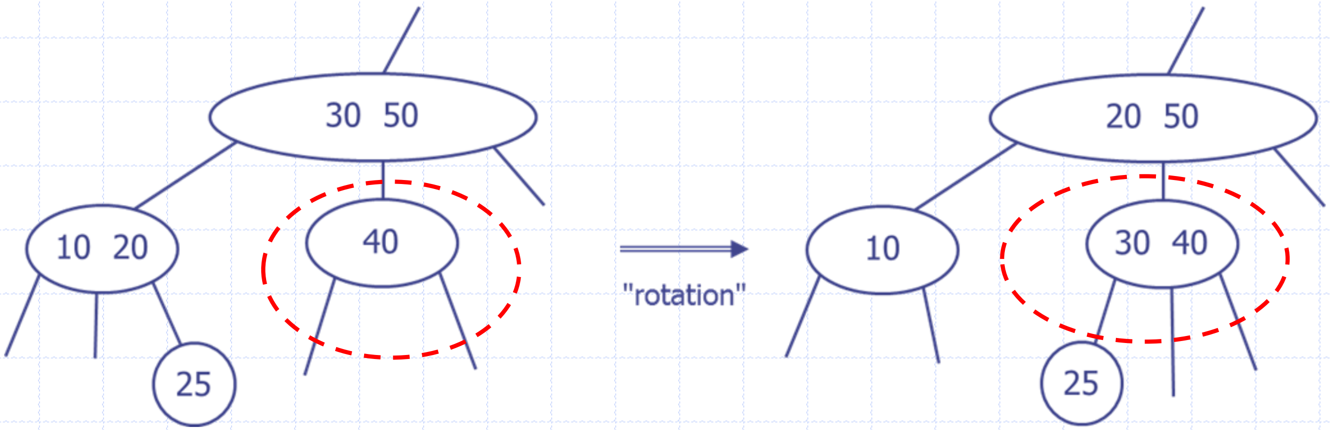
2-3-4 Tree: Deletion

– (underflow problem and solution)

Note: a 2-node leaf creates a problem (1-node, underflow)
Solution: on the way from the root down to the leaf
- turn 2-nodes (except root) into 3-nodes

Case 1: an adjacent sibling has 2 or 3 items

solution → "steal" item from sibling by rotating items and moving subtree



2-3-4 Tree: Deletion

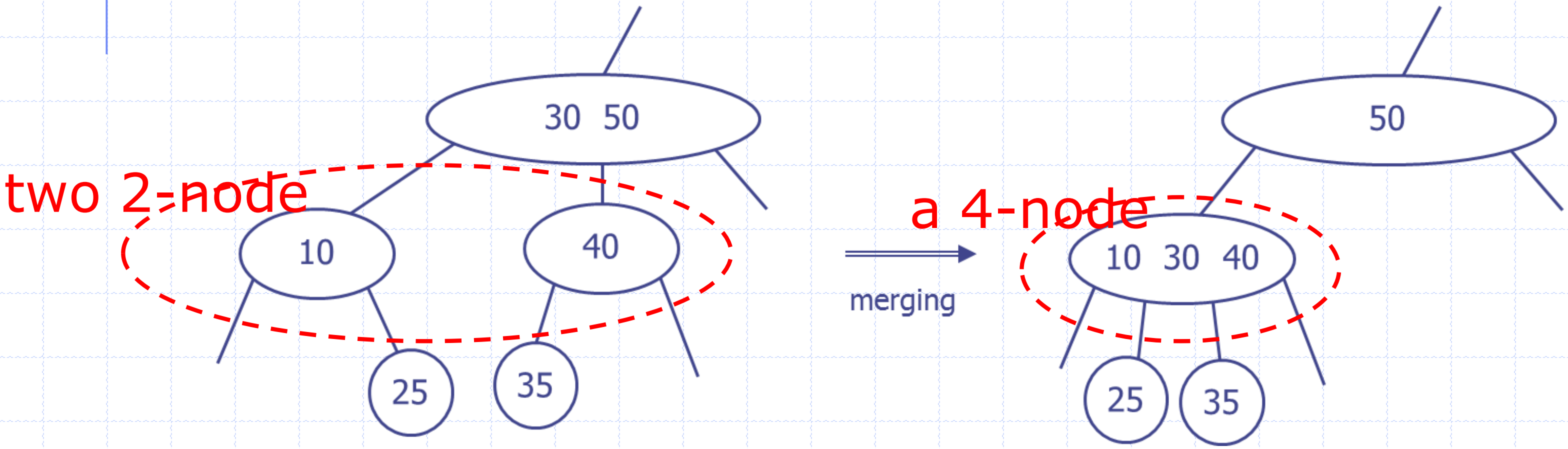
– (underflow problem and solution)

Turning two 2-node into a 4-node …

Case 2: each adjacent sibling has only one item

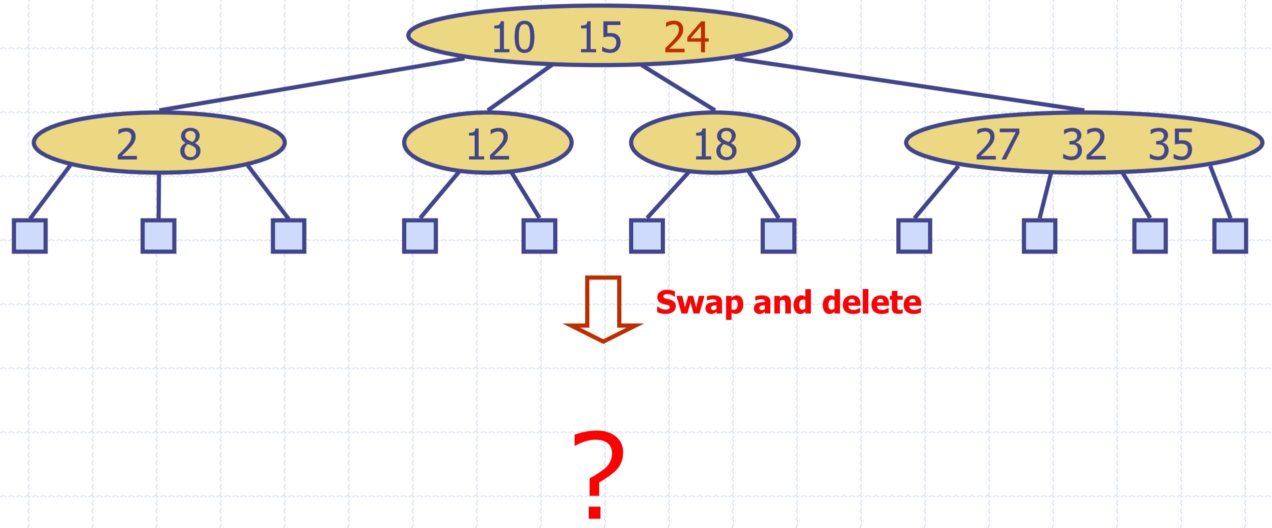
→ "steal" item from parent and merge node with sibling

(note: parent has at least two items, unless it is the root)



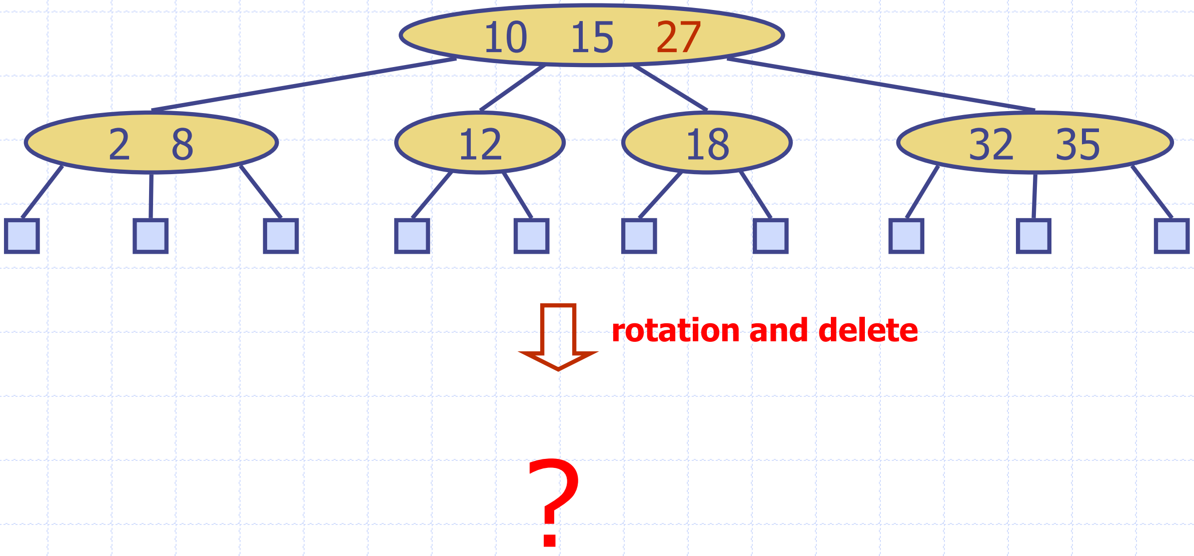
Deletion - more examples

- Example: to delete key 24, we replace it with 27 (inorder successor)



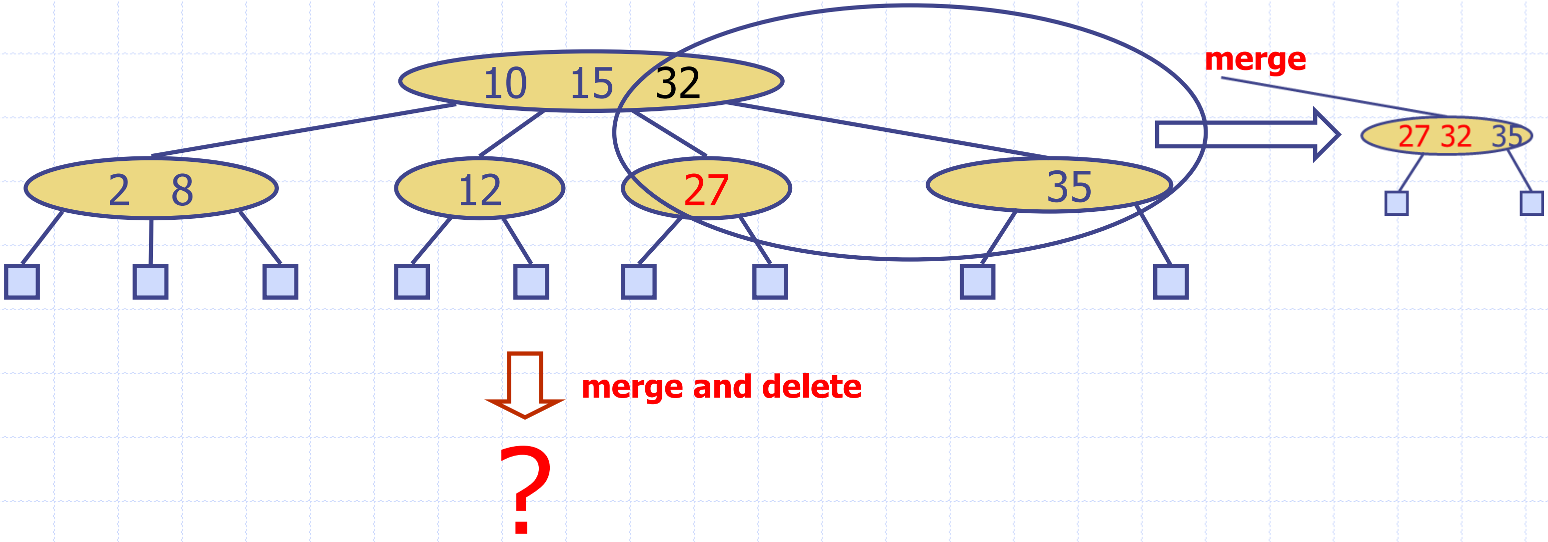
Deletion - more examples

- Example: to delete key 18,



Deletion - more examples

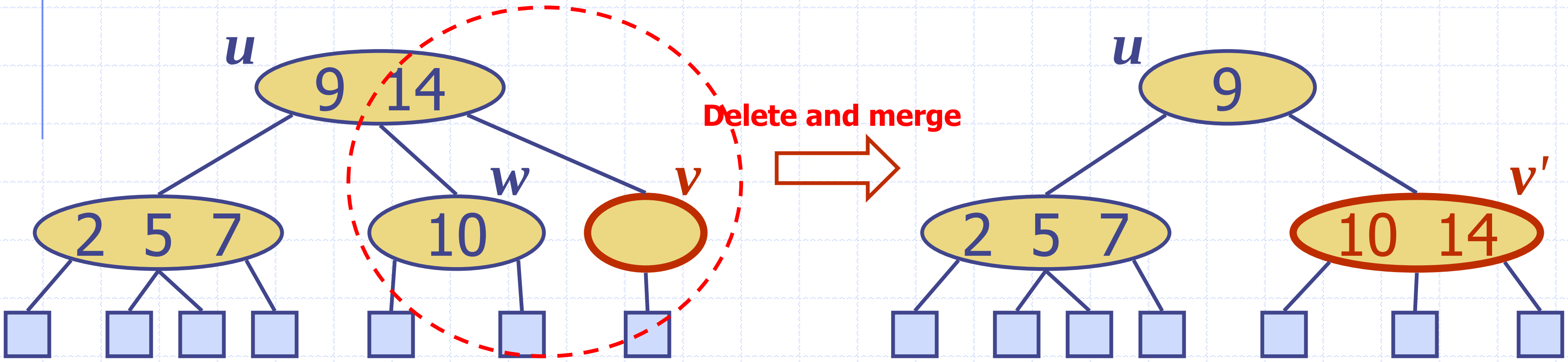
- Example: to delete key 27,



Deletion - more examples

the adjacent siblings of v are 2-nodes

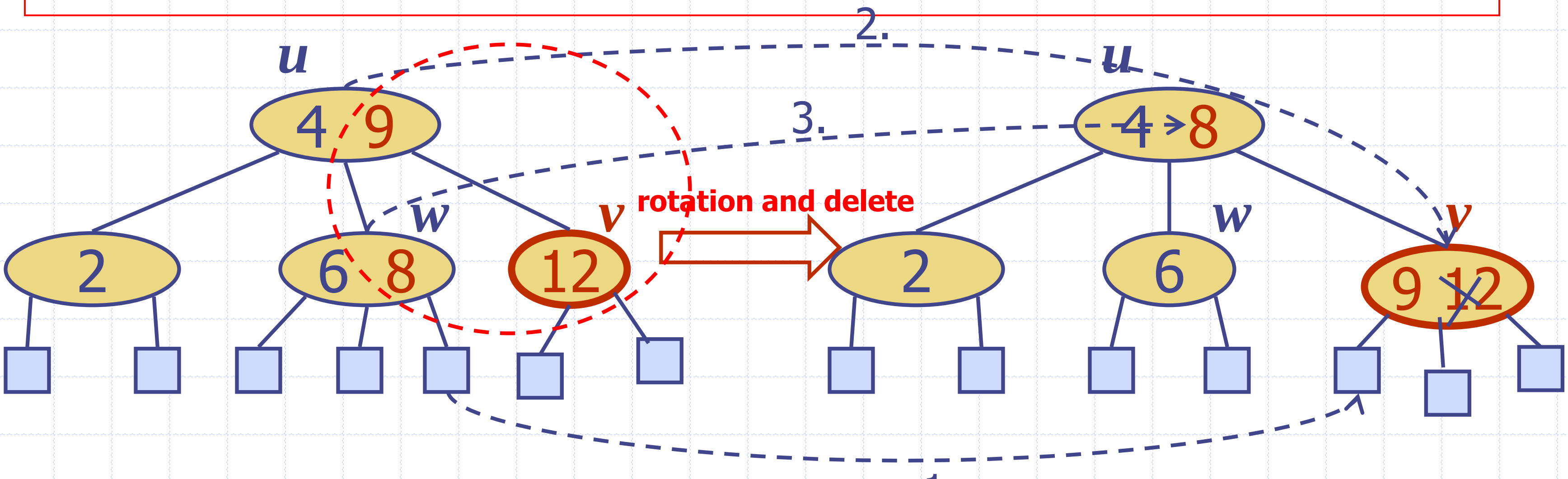
- merge v with an adjacent sibling w and move an item from u to the merged node v'
- After merging, the underflow may propagate to the parent u



Deletion - more example

an adjacent sibling ***w*** of ***v*** is a 3-node or a 4-node

- Transfer operation:
  1. we move a child of ***w*** to ***v***
  2. we move an item from ***u*** to ***v***
  3. we move an item from ***w*** to ***u***
- After a transfer, no underflow occurs

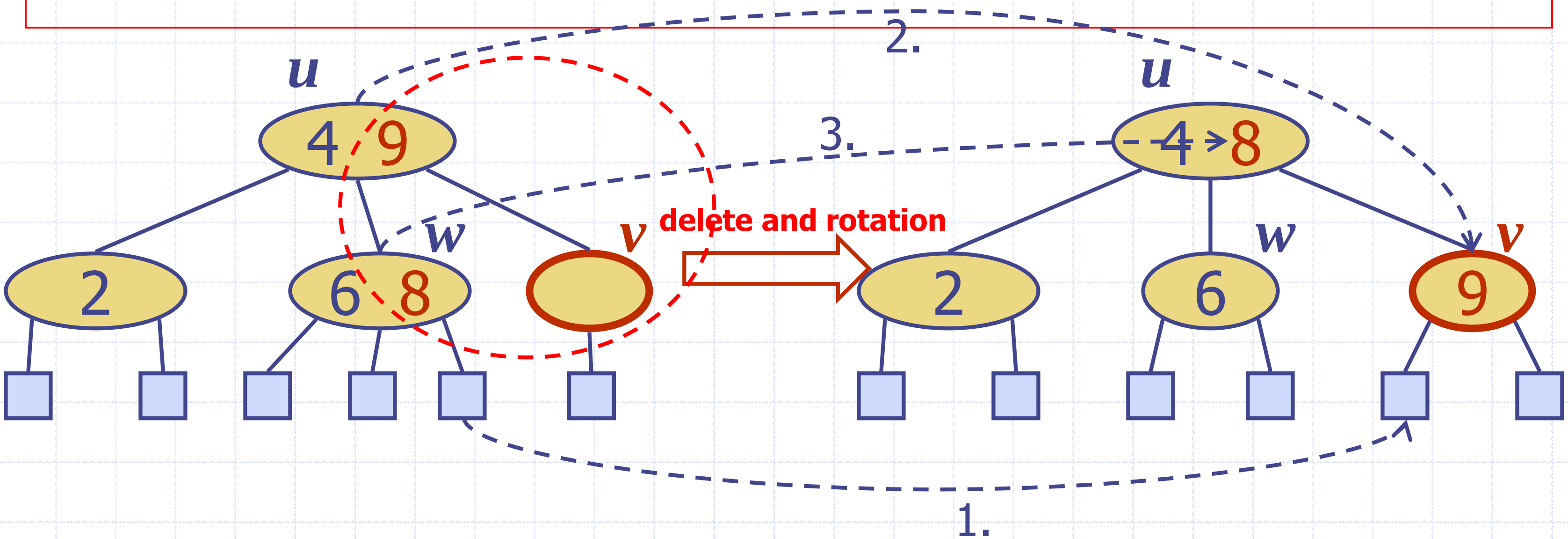


Delete 12

Deletion - more example

an adjacent sibling *w* of *v* is a 3-node or a 4-node

- Transfer operation:
  1. we move a child of *w* to *v*
  2. we move an item from *u* to *v*
  3. we move an item from *w* to *u*
- After a transfer, no underflow occurs



Analysis of Deletion
削除の分析

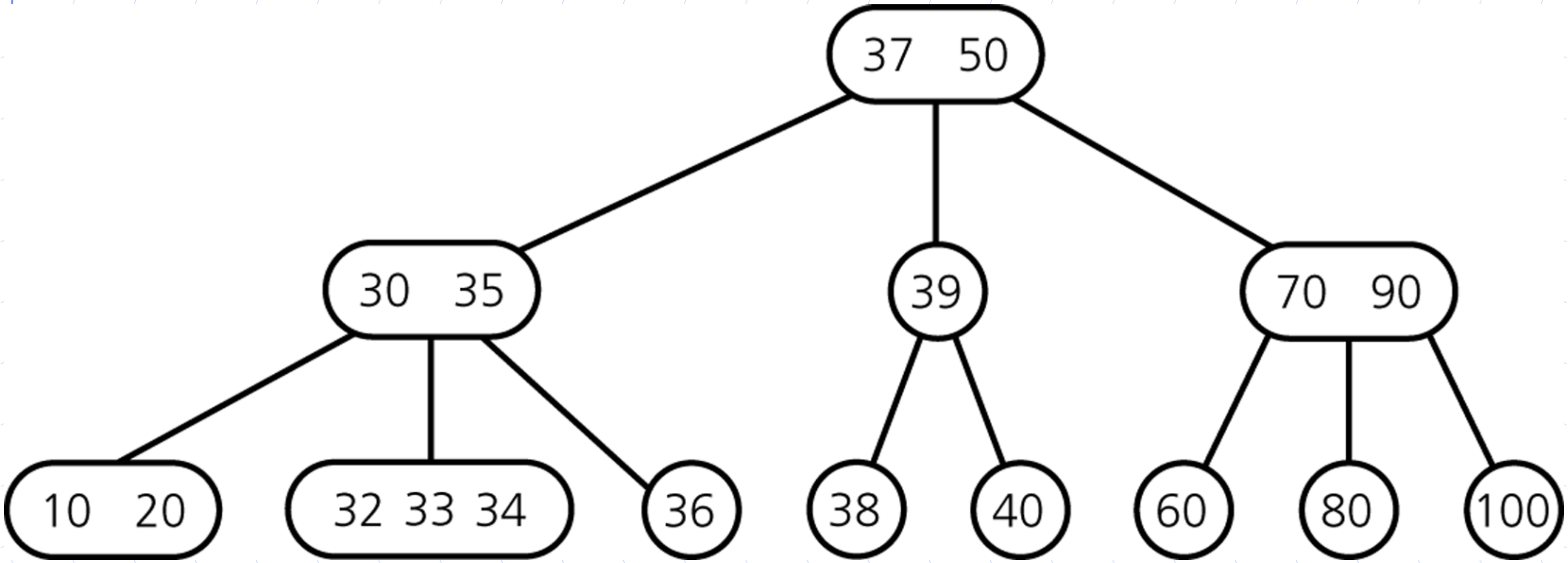
- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
    木Tの高さは$O(\log n)$
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the item
    削除するために$O(\log n)$のノードを訪れる
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
    合体と移動：$O(1)$

Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time
(2,4)木での削除の時間：$O(\log n)$

2-3-4 Tree: Deletion Practice

Work in class

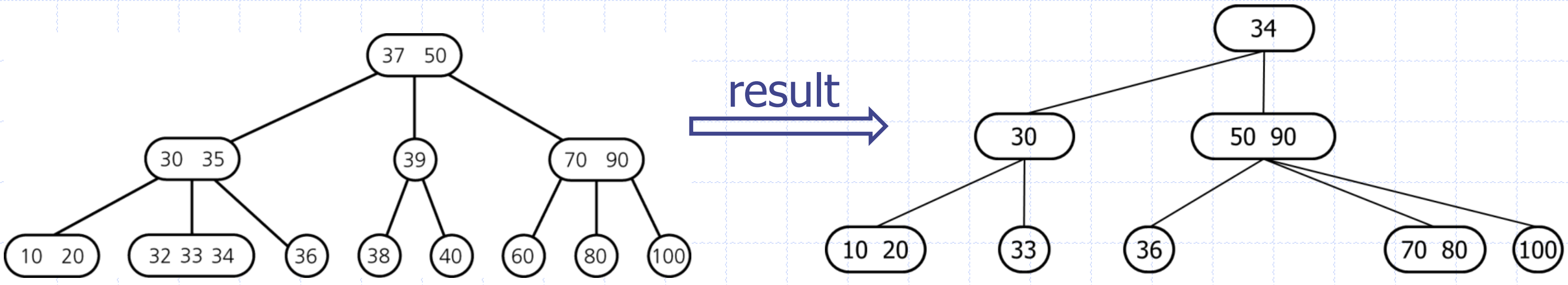
Delete 32, 35, 40, 38, 39, 37, 60



2-3-4 Tree: Deletion Practice

(solution)

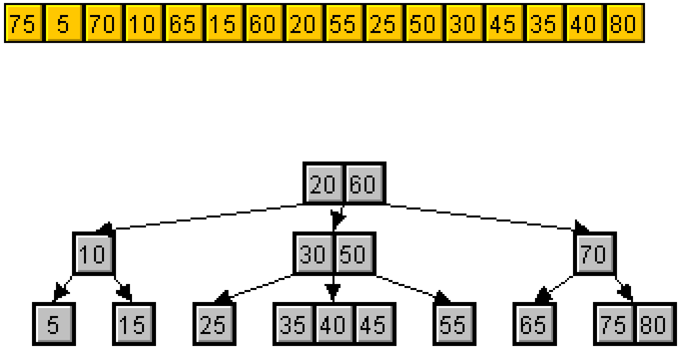
Delete 32, 35, 40, 38, 39, 37, 60



[2-3-4-Tree.jar]()

Work in class

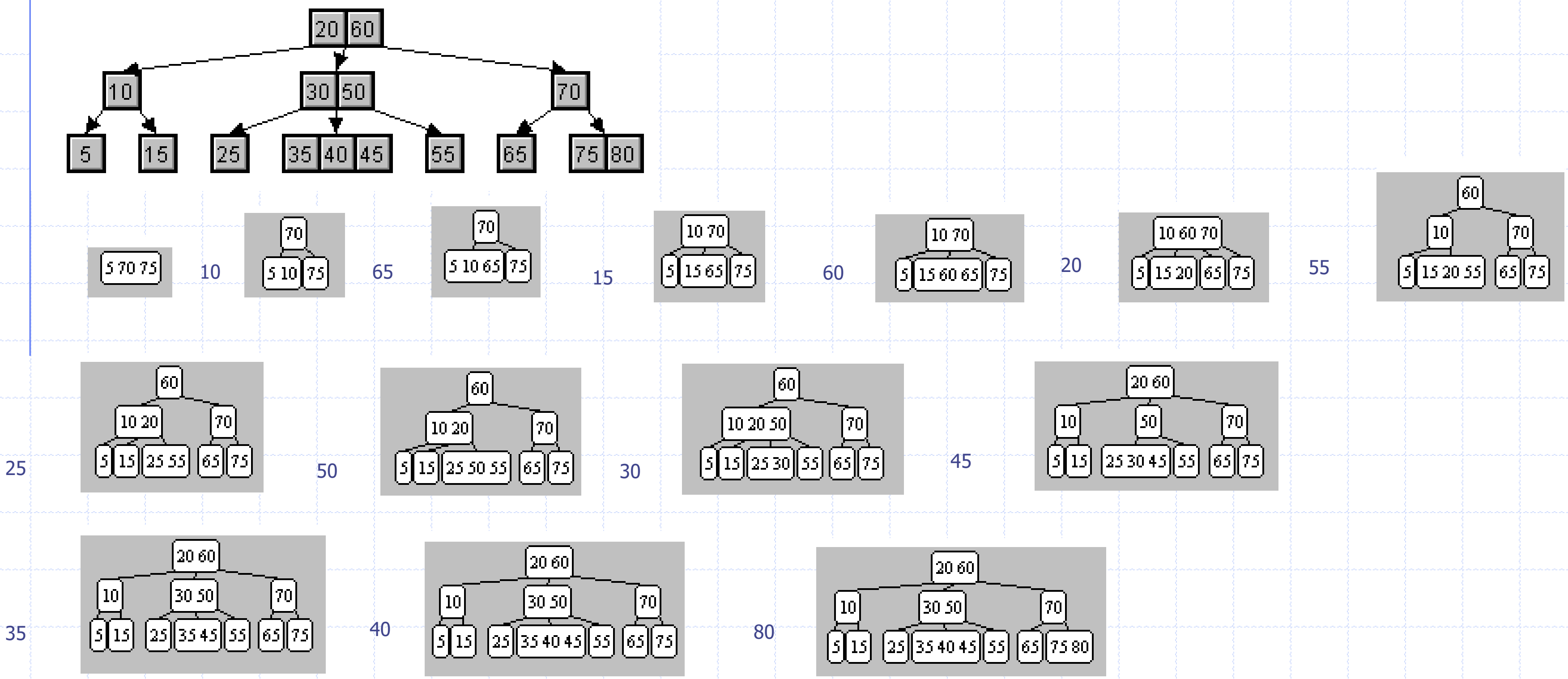
Insertion of 75, 5, 70, 10, 65, 15, 60, 20, 55, 25, 50, 30, 45, 35, 40, 80



解答例:

Insertion of 75, 5, 70, 10, 65, 15, 60, 20, 55, 25, 50, 30, 45, 35, 40, 80

75	5	70	10	65	15	60	20	55	25	50	30	45	35	40	80



例1 : Deletion

60	30	10	20	50	40	70	80	15	90	99

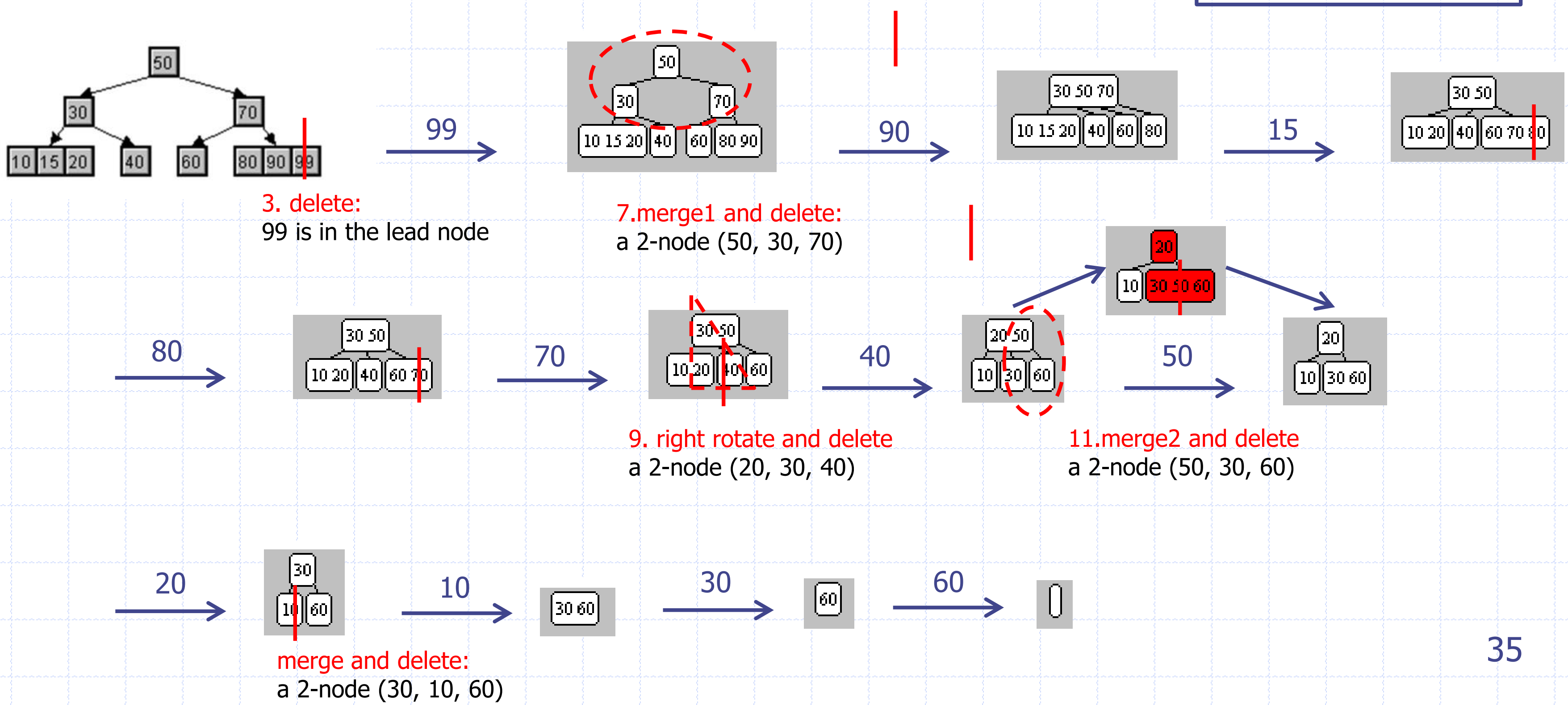
From the root down to the leaf
1. find the node in which item I is in and then check the following cases
2. if (the node is leaf)
3. delete
4. if (the node is not leaf)
5. swap
6. if (the root and two children are 2-nodes)
7. merge1
8. if (sibling node has 2 or 3 items)
9. right/left rotate
10. if (sibling nodes are 2-nodes)
11. merge2

5. swap:
swap item of internal node with inorder successor

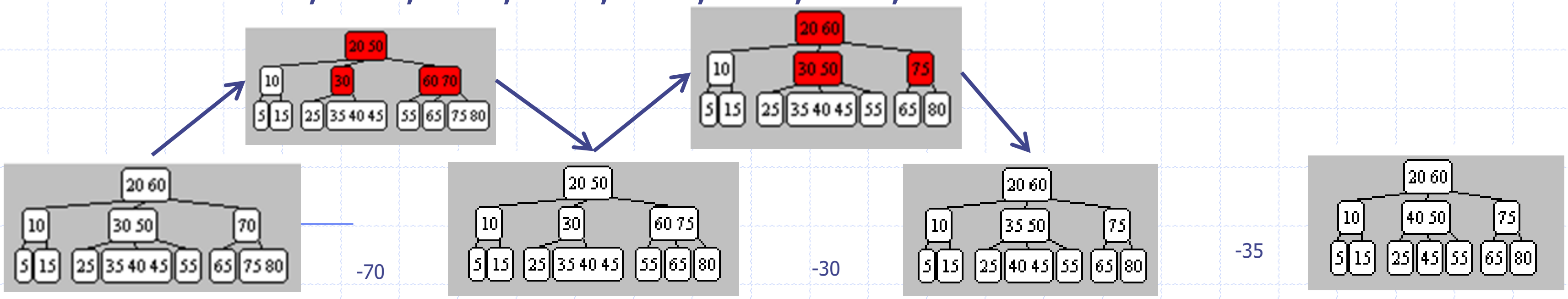
7. merge 1
combine all three elements into the root.

9. right/left rotate:
parent item to the right/left child and the left/right child to its parent

11. merge2
combine all three elements as a child node.



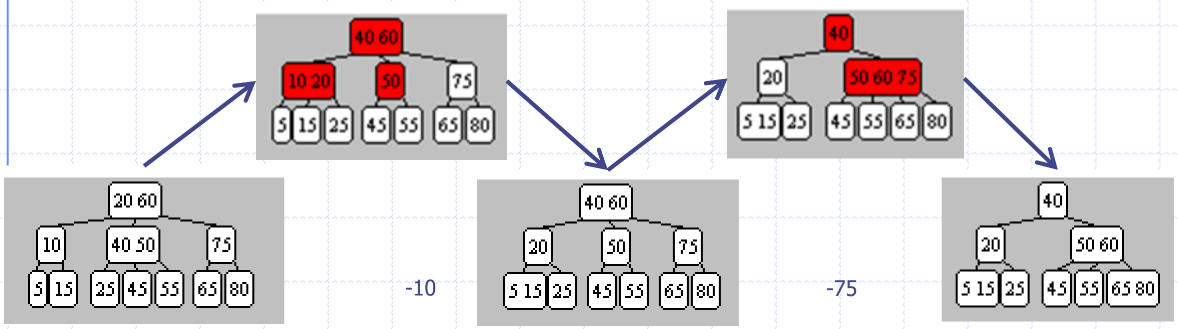
例2:Delete 70, 30, 35, 10, 75, 80, 40,45



(50,60,70) 9.right rotate
(70.75) 5.swap
(70) delete

(30,50,60) 9.left rotate
(30.35) 5.swap
(30) delete

(35.40) 5.swap
(35) delete



(10,20,40) 9.left rotate
(5,10.15) 11.merge2
(10) delete

(60,50,75) 11.merge2
(65,75,80) 11.merge2
(75) delete

From the root down to the leaf

1. find the node in which item *I* is in and then check the following cases
2. if (the node is leaf)
3. delete
4. if (the node is not leaf)
5. swap
6. if (the root and two children are 2-nodes)
7. merge1
8. if (sibling node has 2 or 3 items)
9. right/left rotate
10. if (sibling nodes are 2-nodes)
11. merge2

5. swap: swap item of internal node with inorder successor

7. merge 1: combine all three elements into the root.

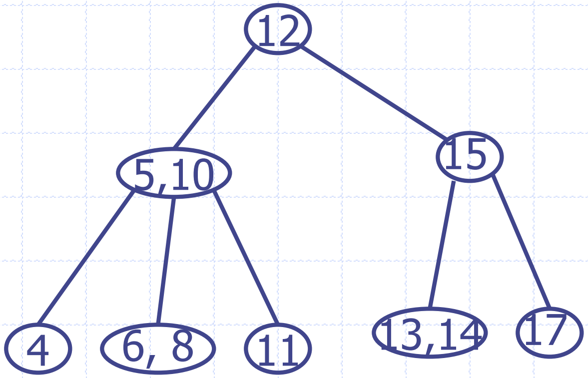
9. right/left rotate: parent item to the right/left child and the left/right child to its parent

11. merge2: combine all three elements as a child node.

Work in class:
delete 80, 40, 45

Exercise 6-1

Consider the following sequence of keys: (4, 12, 13, 14). Remove the items with this set of keys in the order given from the (2,4) tree below. Draw the tree after each removal.



キー配列について考える: (4, 12, 13, 14)。
このキーのセットを図の(2,4)木に削除しなさい。
それぞれの削除後の(2,4)木を描きなさい。

Exercise 6-2 (optional)

Please implement the 2-3-4 Tree deletion algorithm in python.

References

https://www.unf.edu/~broggio/