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 (L5 – Search trees)

- Searching problems
- AVL tree
- 2-3-4 trees（insertion）

Searching Problems

<u>Problem</u>: Given a (multi) set *S* of keys and a search key *K*, find an occurrence of *K* in *S*, if any

- Searching must be considered in the context of:
  - file size (internal vs. external)
  - dynamics of data (static vs. dynamic)
- Like Dictionary: Dictionary operations (dynamic data):
  - find (search)
  - insert
  - delete

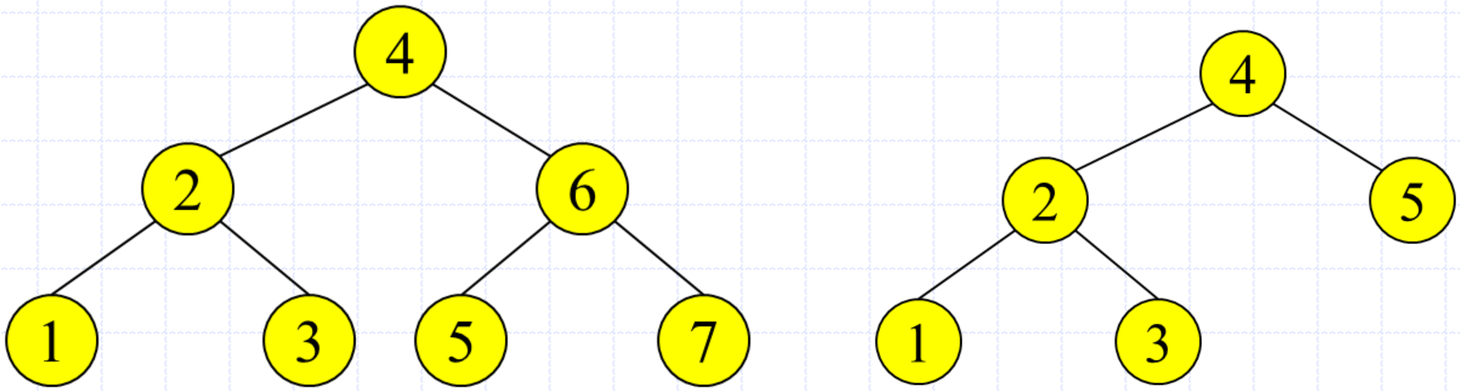
Taxonomy of Searching Algorithms

- List searching
  - sequential search
  - binary search
  - interpolation search
- Tree searching
  - binary search tree (pre-, post-, in- order search)
  - binary balanced trees: AVL trees, red-black trees
  - Multi-way balanced trees: 2-3 trees, 2-3-4 trees, B trees
- Hashing
  - open hashing (separate chaining)
  - closed hashing (open addressing)

AVL tree - Balanced binary search tree

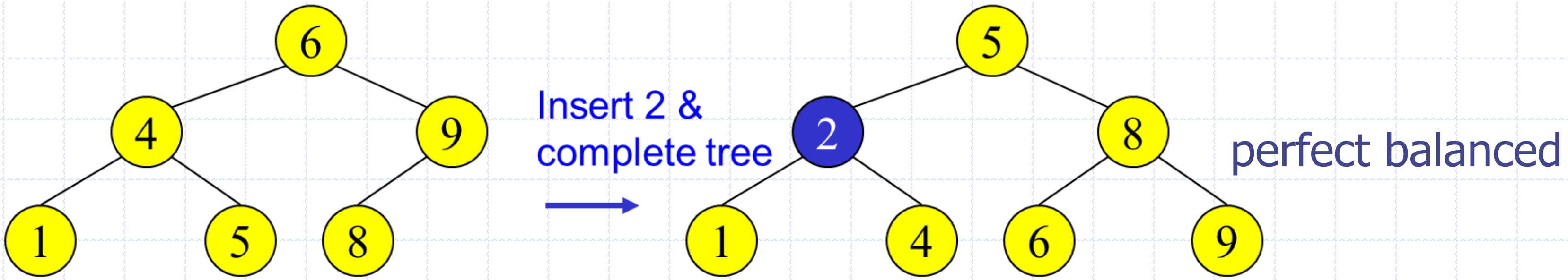
平衡2分探索木

- 特に木構造の一つ。AVL木平衡条件を満たす平衡2分探索木である。左右の部分木の高さの差を多くとも1にする。



balanced?

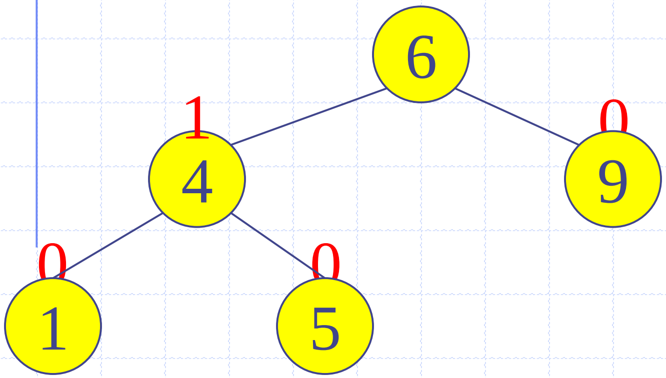
AVL - Good but not Perfect Balance



- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1

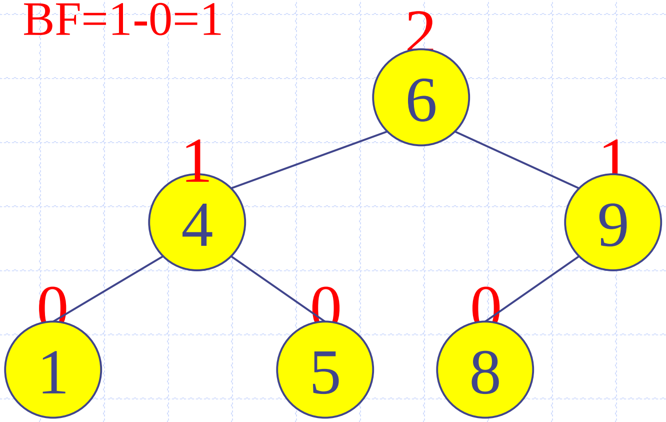
Node Heights

Tree A (AVL)



Tree B (AVL)

height=2   BF=1-0=1



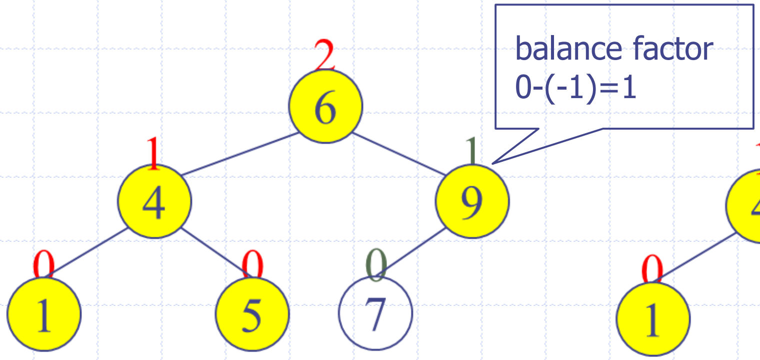
Count from leaf nodes

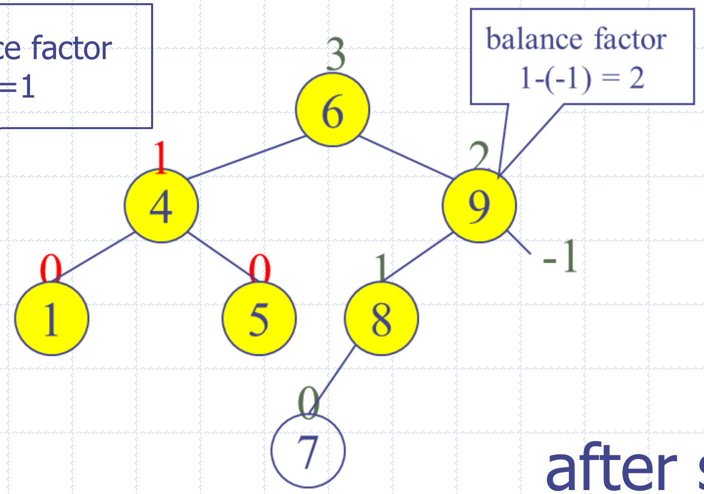
height of node = h

balance factor = $h_{left} - h_{right}$

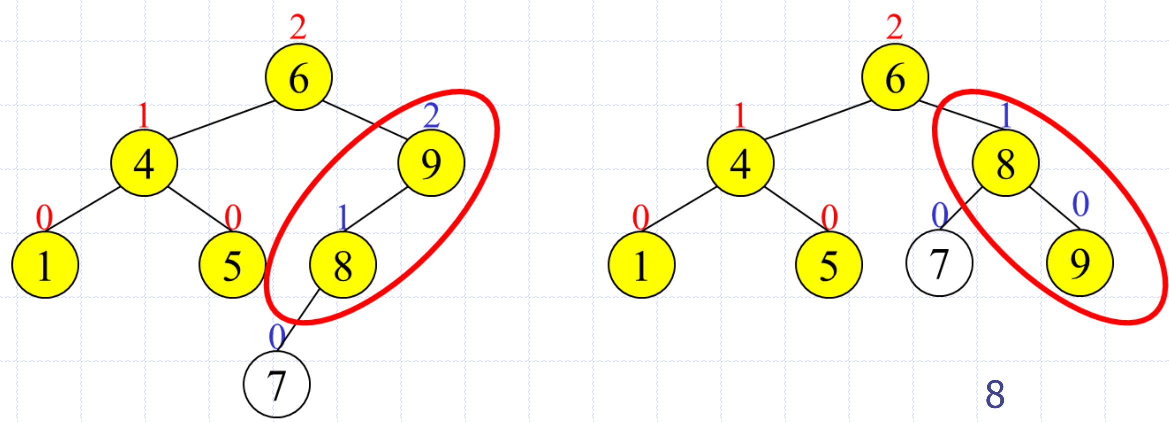
Node Heights after Insert 7

Tree A (AVL) Tree B (not AVL)



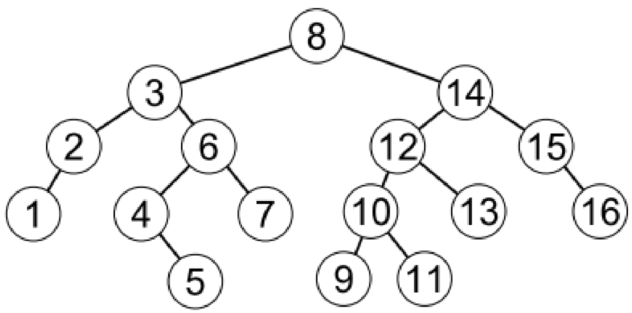


after single rotation

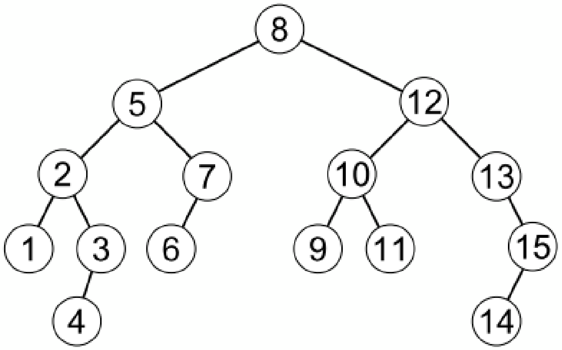


Work in class

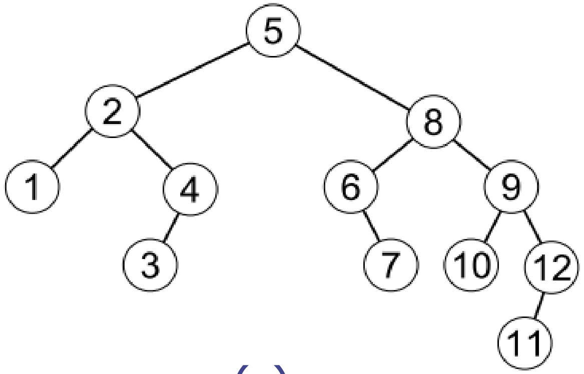
- AVL 木?



(a)



(b)

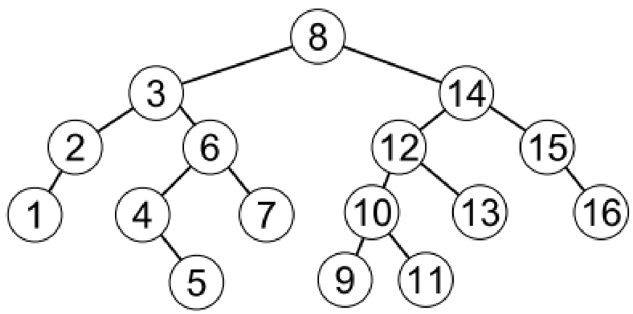


(c)

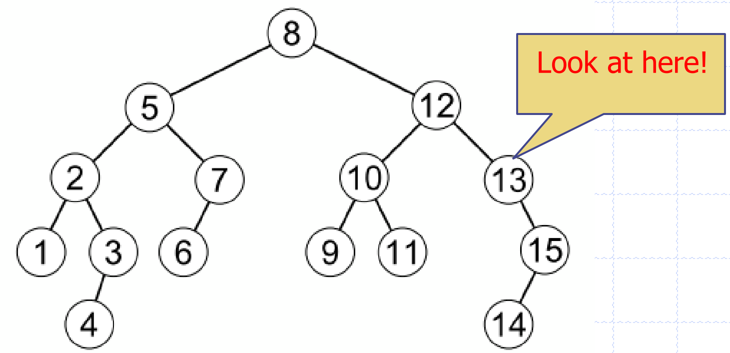
AVL木の部分木もAVL木

Work in class (answer)

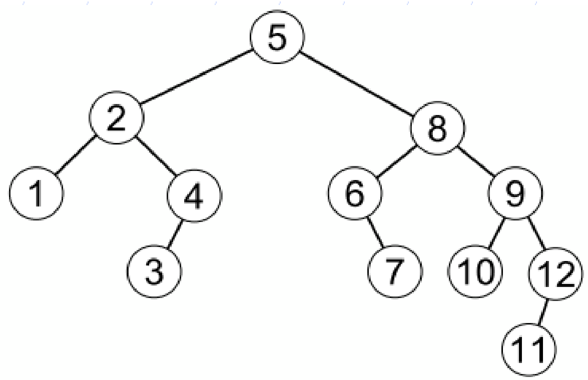
AVL 木?



(a) yes



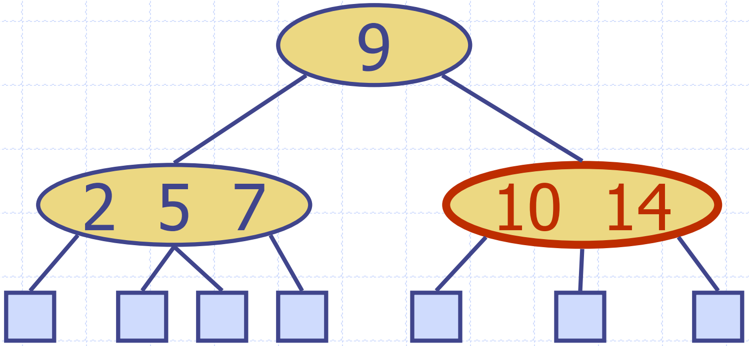
(b) no



(c) yes

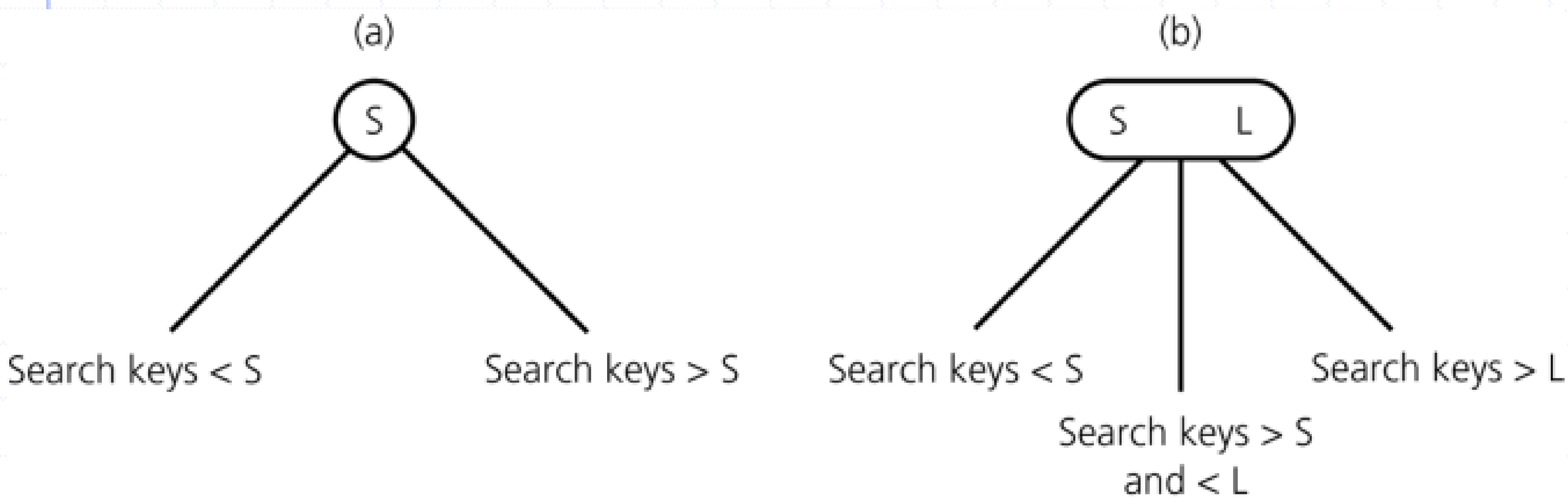
(2,4) Trees

- B木は多分岐の平衡木(バランス木)である。1 ノードから最大 *m* 個の枝が出るとき、これをオーダー(order) *m* のB木という。
- B木の中でも特に、オーダー3のものを2-3木、オーダー4のものを2-3-4木 (2, 4) と呼ぶ。



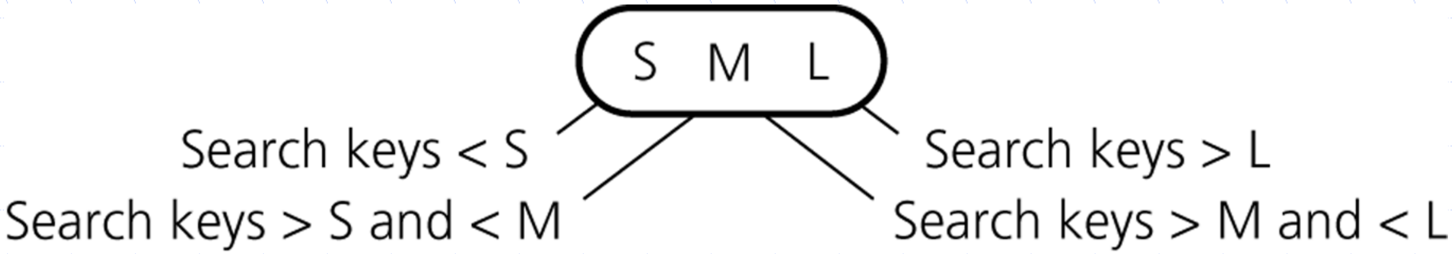
Features of 2-3 Trees

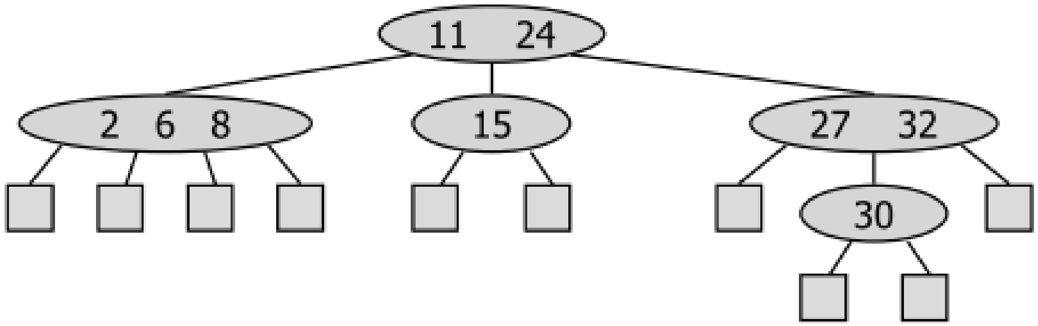
- No 1-nodes
- 2-nodes can have 1 item and 2 children
- 3-nodes can have 2 items and 3 children
- Depending on the number of children, an internal node of a 2-3 tree is called a 2-node or a 3-node



Features of (2,4) Trees

- 4-nodes can have 3 items and 4 children
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

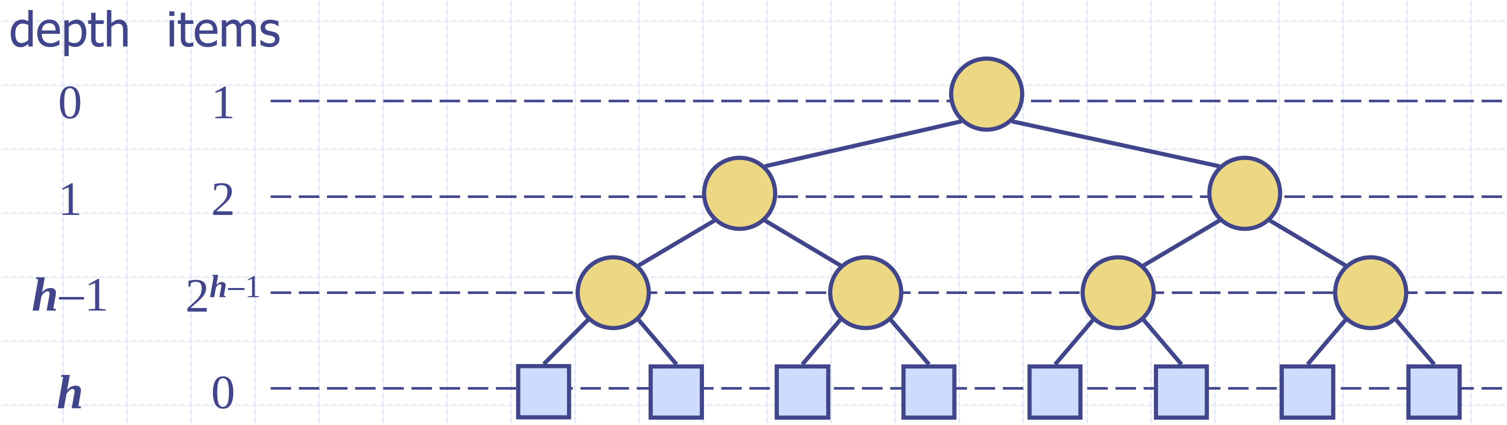




Height of a (2,4) Tree
(2,4)木の高さ

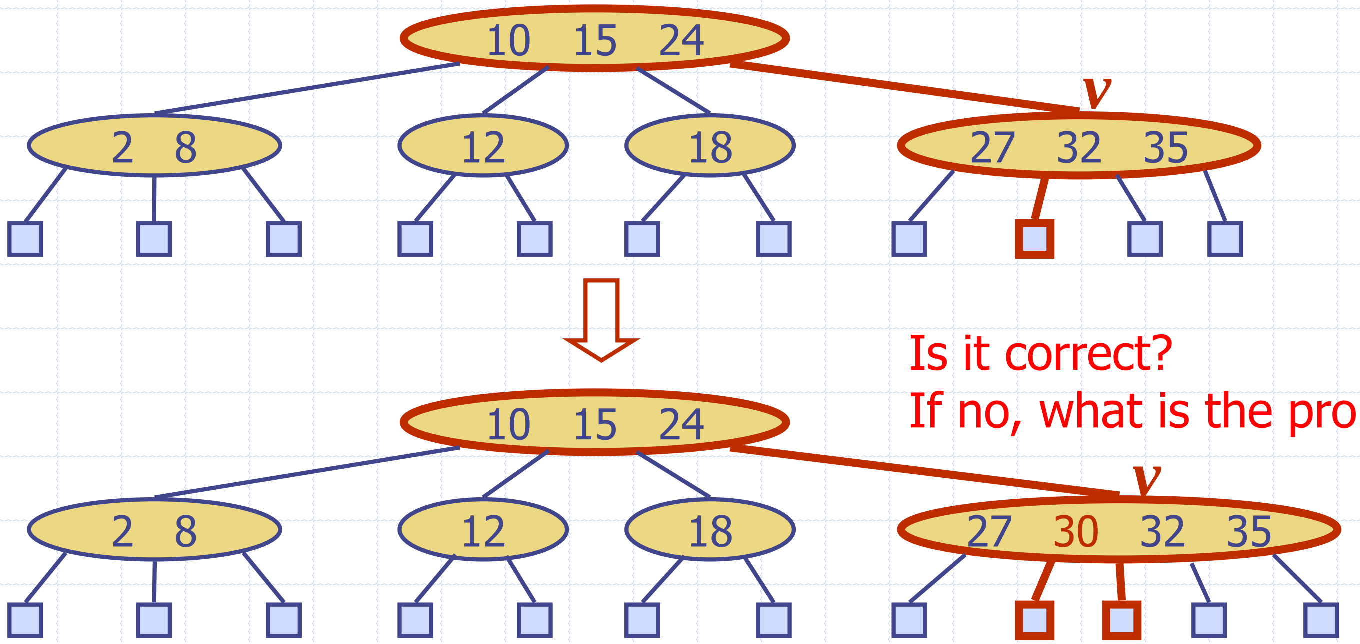
- Theorem: A (2,4) tree storing n items has height $O(\log n)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time

Work in Class
Hint: refer to the following binary tree,
proof this theorem



(2, 4) Tree: Insertion 挿入

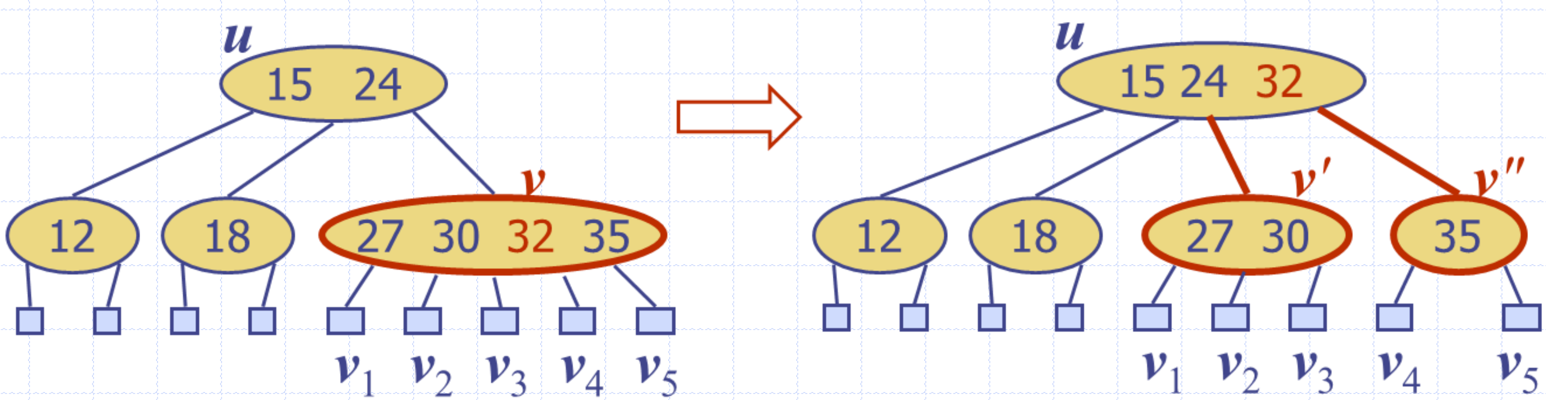
- We insert a new item at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
    ノード数が5になってしまいオーバーフロー
- Example: inserting key 30 causes an overflow

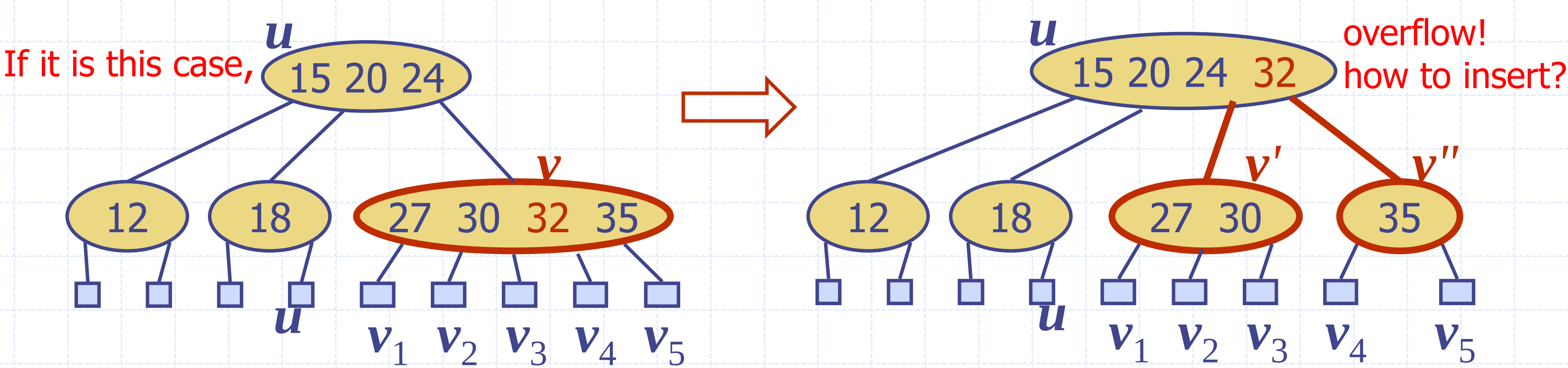


Is it correct?
If no, what is the problem?

Overflow and Split
オーバーフロウと分裂

- We handle an overflow at a 5-node v with a split operation:
  オーバーフローを解決するために分裂を行う
- The overflow may propagate to the parent node u
  親であるノードuによってオーバーフローが広まる

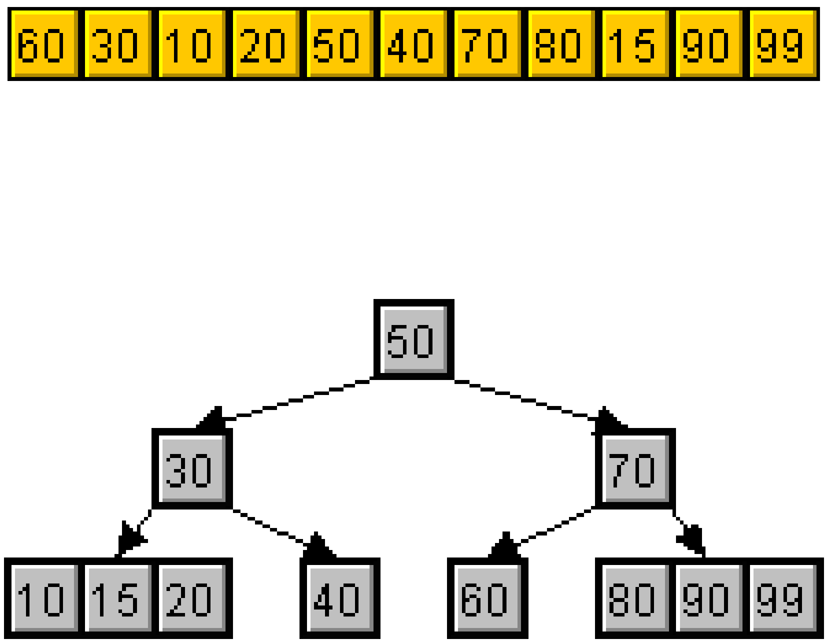




(2,4) Tree: Insertion

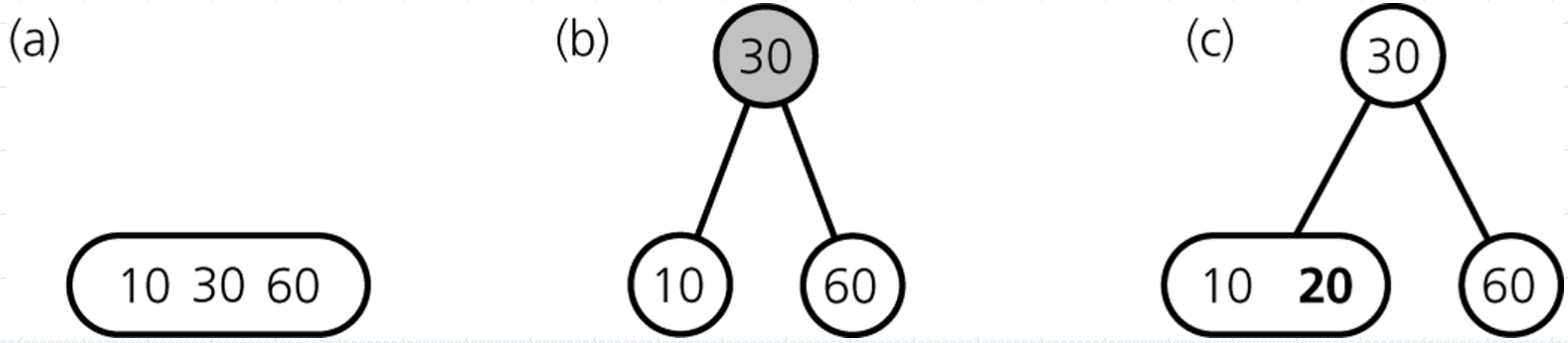
Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 99

2-3-4 tree insert



(2,4) Tree: Insertion

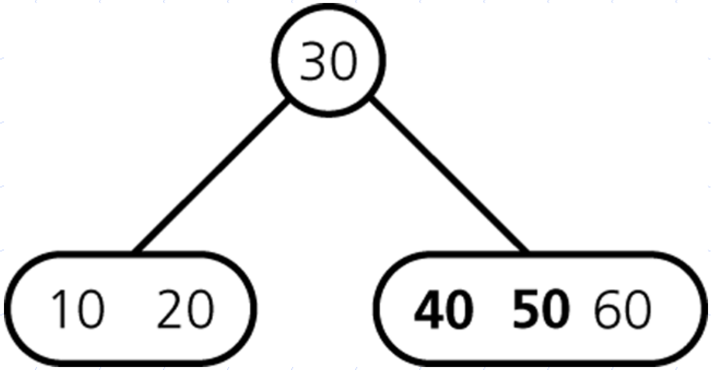
Inserting 60, 30, 10, 20 ...



... 50, 40 ...

(2,4) Tree: Insertion

Inserting 50, 40 …



… 70, …

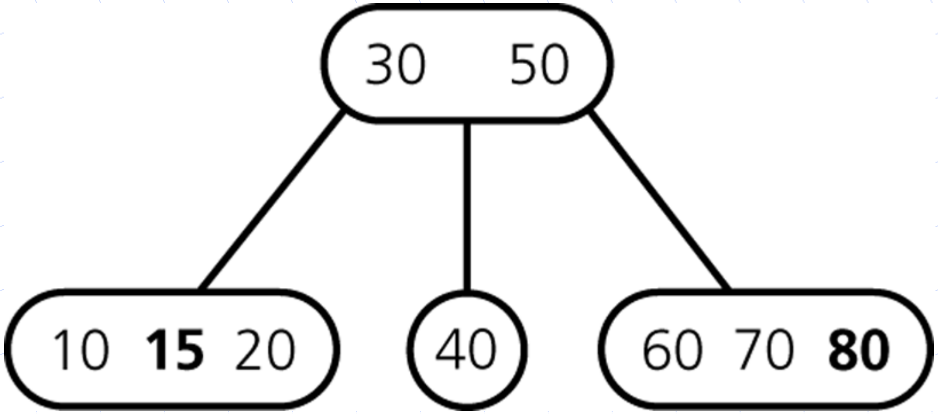
Work in class

Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 99

Insert 70, please draw (2,4) tree

(2,4) Tree: Insertion

Inserting 80, 15 ...



... 90 ...

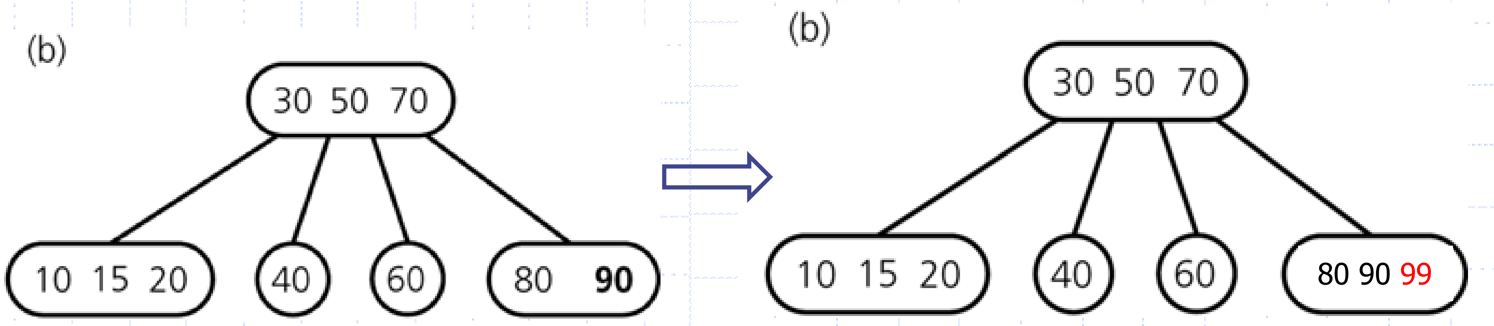
Work in class

Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 99

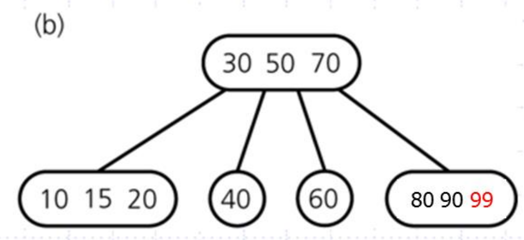
Insert 90, please draw (2,4) tree

(2,4) Tree: Insertion

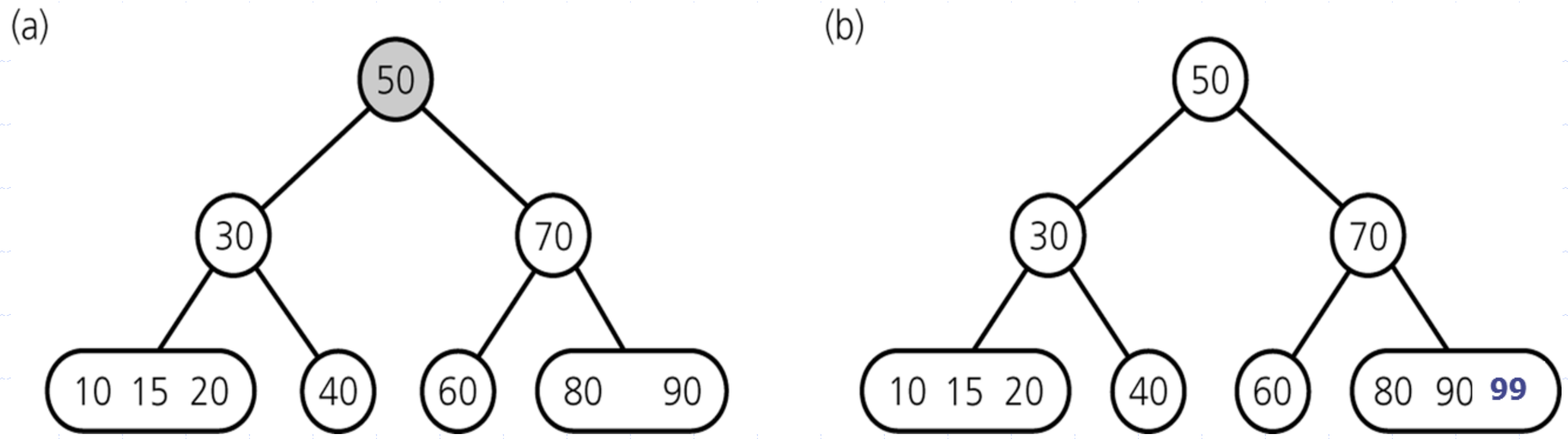
Inserting 99 ...

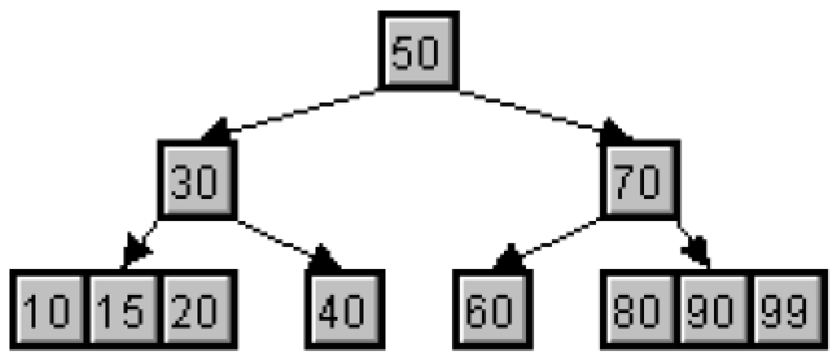


(2,4) Tree: Insertion



Inserting 99 ...

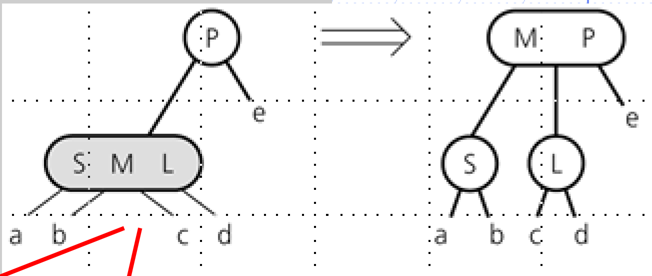




Insertion in 2-3-4 trees

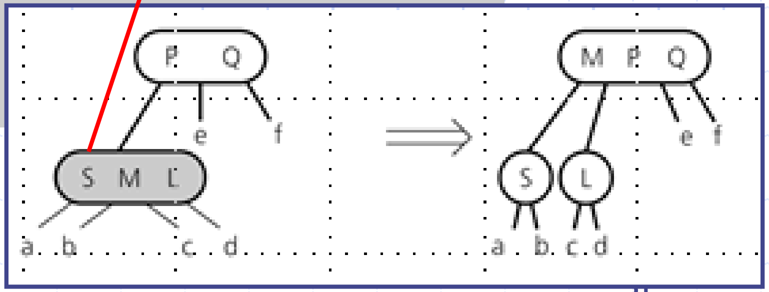
Step 1 Search for the item to be inserted (same as in 2-3 trees).
Step 2 Insert at the leaf level. The following cases are possible:

- 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.



Understand it from the Python code

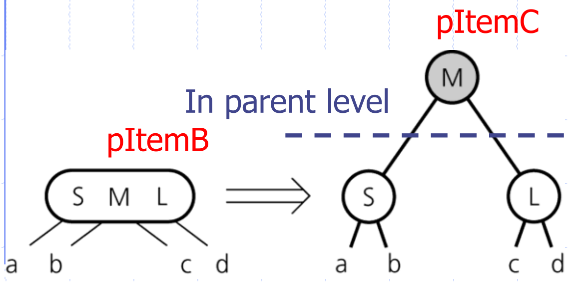
Step 2 Insert at the leaf level. The following cases are possible:

- 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.

```
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
```

(2,4) Tree: Insertion Procedure

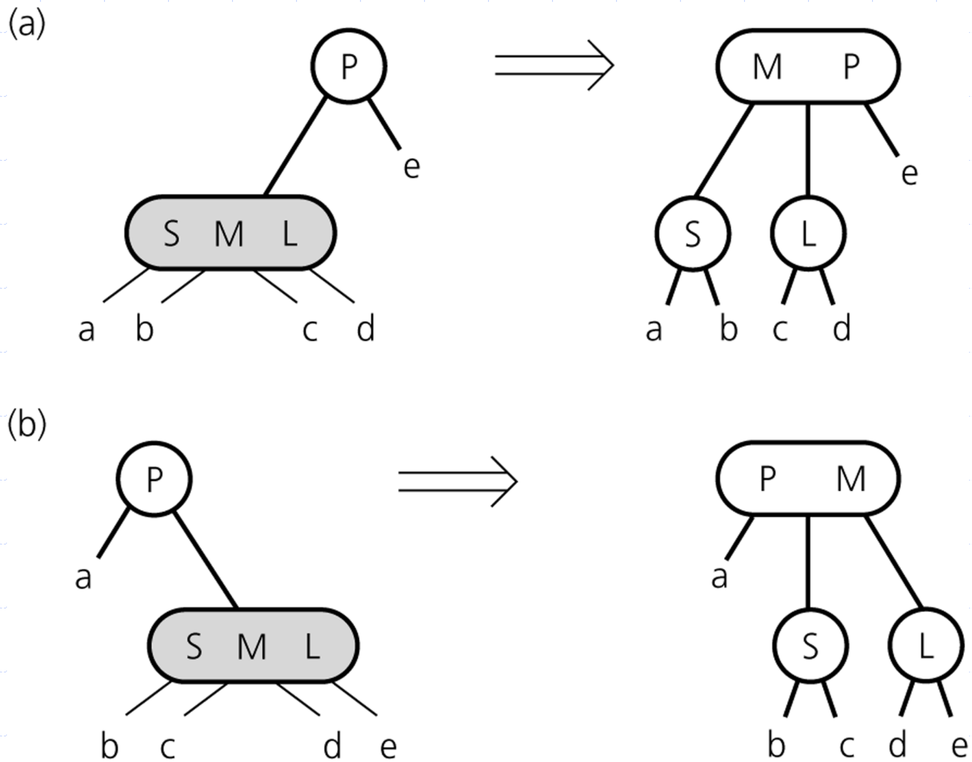
Splitting 4-nodes




```
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()
```

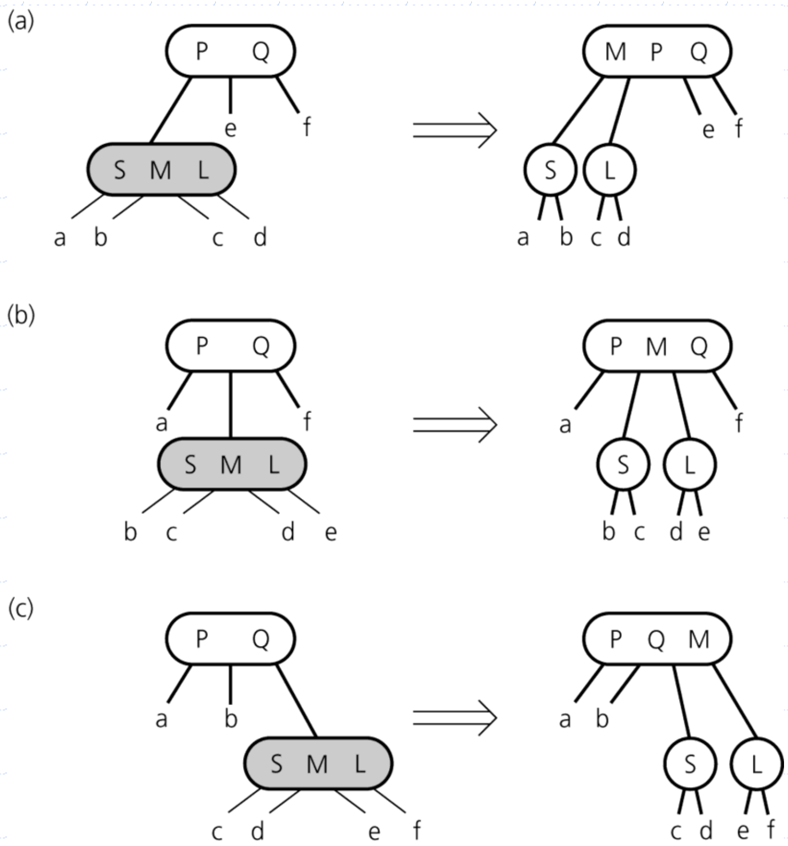
(2,4) Tree: Insertion Procedure

Splitting a 4-node whose parent is a 2-node during insertion



(2,4) Tree: Insertion Procedure

Splitting a 4-node whose parent is a 3-node during insertion



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

2-3-4 Tree

- Insertion process implementation in python

```
class DataItem:

        def __init__(self, dd): #special method to create objects
        #with instances customized to a specific initial state
                self.dData = dd#one piece of data

        def displayItem(self):  #format " /27"
                print '/', self.dData,
#end class DataItem
```

```python
class Node:
        #as private instance variables don't exist in Python,
        #hence using a convention: name prefixed with an underscore, to treat them as non-public part
        _ORDER = 4
        def __init__(self):
                self._numItems = 0
                self._pParent = None
                self._childArray = []   #array of nodes
                self._itemArray = []    #array of data
                for j in xrange(self._ORDER):   #initialize arrays
                        self._childArray.append(None)
                for k in xrange(self._ORDER - 1):
                        self._itemArray.append(None)

                        #connect child to this node
        def connectChild(self, childNum, pChild):
                self._childArray[childNum] = pChild
                if pChild:
                        pChild._pParent = self

                        #disconnect child from this node, return it
        def disconnectChild(self, childNum):
                pTempNode = self._childArray[childNum]
                self._childArray[childNum] = None
                return pTempNode

        def getChild(self, childNum):
                return self._childArray[childNum]

        def getParent(self):
                return self._pParent

        def isLeaf(self):
                return not self._childArray[0]

        def getNumItems(self):
                return self._numItems

        def getItem(self, index):       #get DataItem at index
                return self._itemArray[index]

        def isFull(self):
                return self._numItems==self._ORDER - 1
```

```
def findItem(self, key):         #return index of
        for j in xrange(self._ORDER-1): #item (within node)
                if not self._itemArray[j]:      #if found,
                        break   #otherwise,
                elif self._itemArray[j].dData == key:   #return -1
                        return j
        return -1
#end findItem

def insertItem(self, pNewItem):
        #assumes node is not full
        self._numItems += 1#will add new item
        newKey = pNewItem.dData #key of new item

        for j in reversed(xrange(self._ORDER-1)):       #start on right,        #examine items
                if self._itemArray[j] == None:  #if item null,
                        pass    #go left one cell
                else:   #not null,
                        itsKey = self._itemArray[j].dData       #get its key
                        if newKey < itsKey:     #if it's bigger
                                self._itemArray[j+1] = self._itemArray[j]       #shift it right
                        else:
                                self._itemArray[j+1] = pNewItem #insert new item
                                return j+1#return index to new item
                #end else (not null)
        #end for        #shifted all items,
        self._itemArray[0] = pNewItem   #insert new item
        return 0
#end insertItem()
```

```
    def removeItem(self):    #remove largest item
        #assumes node not empty
        pTemp = self._itemArray[self._numItems-1]       #save item
        self._itemArray[self._numItems-1] = None        #disconnect it
        self._numItems -= 1#one less item
        return pTemp#return item

    def displayNode(self):   #format "/24/56/74"
        for j in xrange(self._numItems):
            self._itemArray[j].displayItem()            #format "/56"
        print '/'        #final "/"

#end class Node
```

```python
class Tree234:
    #as private instance variables don't exist in Python,
    #hence using a convention: name prefixed with an underscore, to treat them as non-public part
    def __init__(self):
        self._pRoot = Node()    #root node

    def find(self, key):
        pCurNode = self._pRoot  #start at root
        while True:
            childNumber=pCurNode.findItem(key)
            if childNumber != -1:
                return childNumber      #found it
            elif pCurNode.isLeaf():
                return -1#can't find it
            else:   #search deeper
                pCurNode = self.getNextChild(pCurNode, key)
        #end while

    def insert(self, dValue):       #insert a DataItem
        pCurNode = self._pRoot
        pTempItem = DataItem(dValue)

        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
        pCurNode.insertItem(pTempItem)  #insert new item
    #end insert()
```

```
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()
```

```
    #gets appropriate child of node during search of value
    def getNextChild(self, pNode, theValue):
            #assumes node is not empty, not full, not a leaf
            numItems = pNode.getNumItems()

            for j in xrange(numItems):      #for each item in node
                    if theValue < pNode.getItem(j).dData:   #are we less?
                            return pNode.getChild(j)        #return left child
            else:   #end for            #we're greater, so
                    return pNode.getChild(j + 1)    #return right child

    def displayTree(self):
            self.recDisplayTree(self._pRoot, 0, 0)

    def recDisplayTree(self, pThisNode, level, childNumber):
            print 'level=', level, 'child=', childNumber,
            pThisNode.displayNode() #display this node

            #call ourselves for each child of this node
            numItems = pThisNode.getNumItems()
            for j in xrange(numItems+1):
                    pNextNode = pThisNode.getChild(j)
                    if pNextNode:
                            self.recDisplayTree(pNextNode, level+1, j)
                    else:
                            return
    #end recDisplayTree()
#end class Tree234
```

```
pTree = Tree234()
pTree.insert(50)
pTree.insert(40)
pTree.insert(60)
pTree.insert(30)
pTree.insert(70)

#as Python doesn't support switch, simulating the same with dictionary and functions
def show():
        pTree.displayTree()

def insert():
        value = int(raw_input('Enter value to insert: '))
        pTree.insert(value)

def find():
        value = int(raw_input('Enter value to find: '))
        found = pTree.find(value)
        if found != -1:
                print 'Found', value
        else:
                print 'Could not find', value

case = { 's' : show,
         'i' : insert,
         'f' : find}
#switch simulation completed

while True:
        print
        choice = raw_input('Enter first letter of show, insert, or find: ')
        if case.get(choice, None):
                case[choice]()
        else:
                print 'Invalid entry'
#end while
del pTree
#end
```

2-3-4 tree demo

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

2-3-4-tree.jar

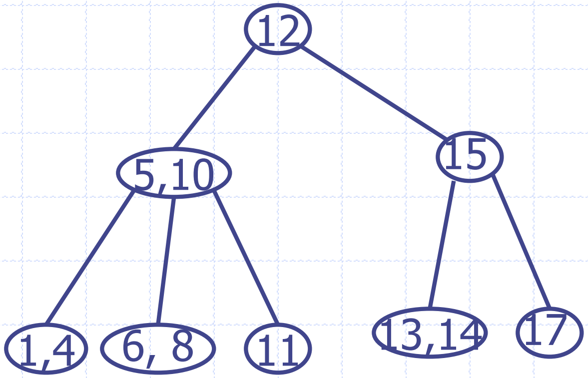
http://stackoverflow.com/questions/15047935/234-tree-python

http://www.clear.rice.edu/comp212/01-fall/lectures/33/

https://tbc-python.fossee.in/convert-notebook/Sams_Teach_Yourself_Data_Structures_and_Algorithms_Analysis_in_24_Hours/chapter20_1.ipynb

Exercise 5-1

Consider the following sequence of keys: (2,3,7,9). Insert the items with this set of keys in the order given into the (2,4) tree below. Draw the tree after each removal.



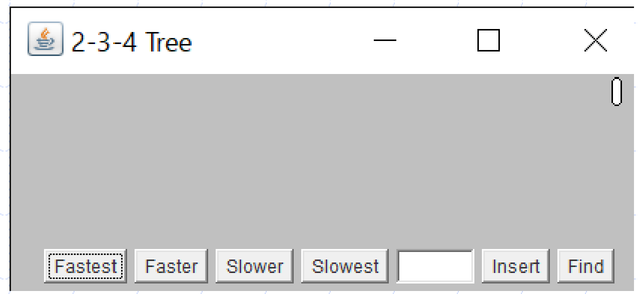
キー配列について考える: (2,3,7,9)。
このキーのセットを図の(2,4)木に挿入しなさい。
それぞれの挿入後の(2,4)木を描きなさい。

Exercise 5-2

3.3.1 Understand 2-3-4 tree and run the example implementation in Python (given in this slide)

3.3.2 (optional) Improve the program so that it have GUI for insertion

```
pTree.insert(50)
pTree.insert(40)
pTree.insert(60)
pTree.insert(30)
pTree.insert(70)
```



and can display 2-3-4 tree.

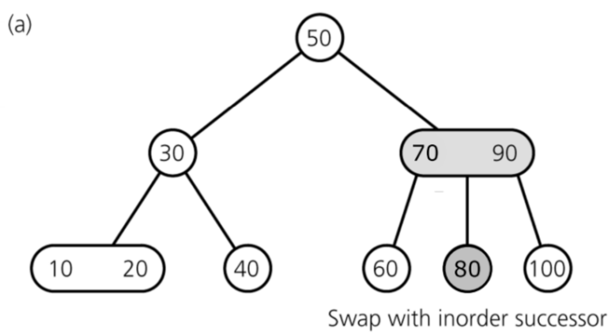
```
Enter first letter of show, insert, or find: s
level= 0 child= 0 / 50 /
level= 1 child= 0 / 30 / 40 /
level= 1 child= 1 / 60 / 70 /
```

2-3-4 Tree: Deletion

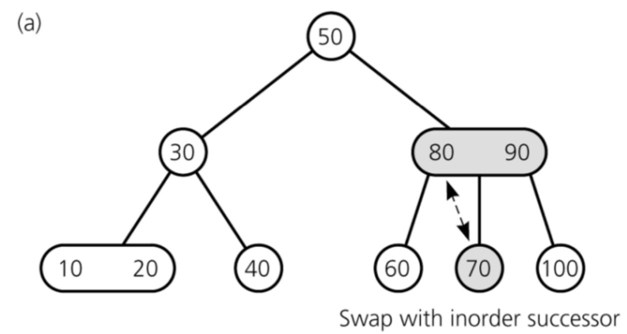
Deletion procedure:

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

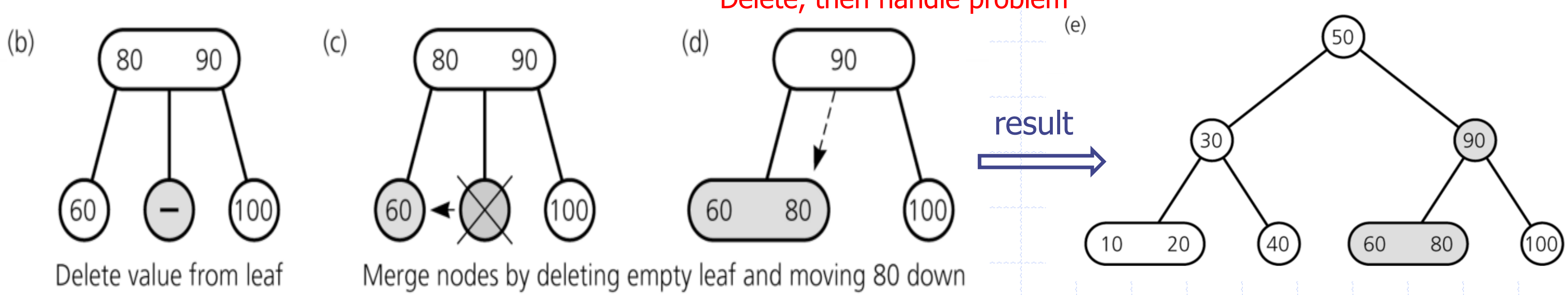
Delete 70



Deleting 70: swap 70 with inorder successor (80)



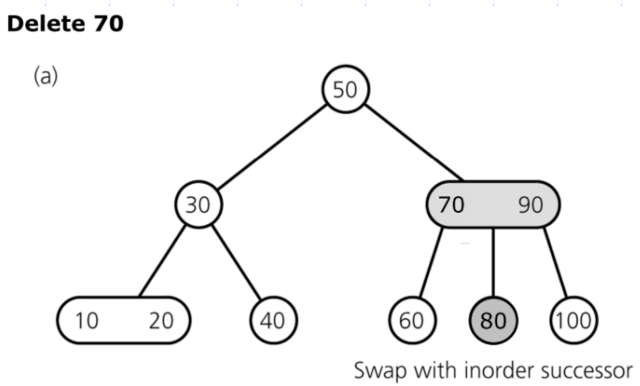
Delete, then handle problem

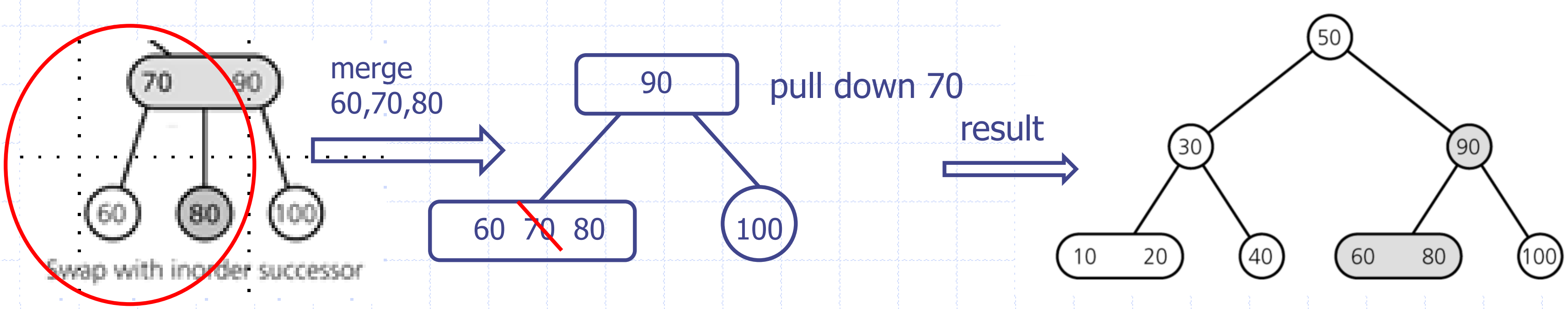


2-3-4 Tree: Deletion

Deletion procedure:

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





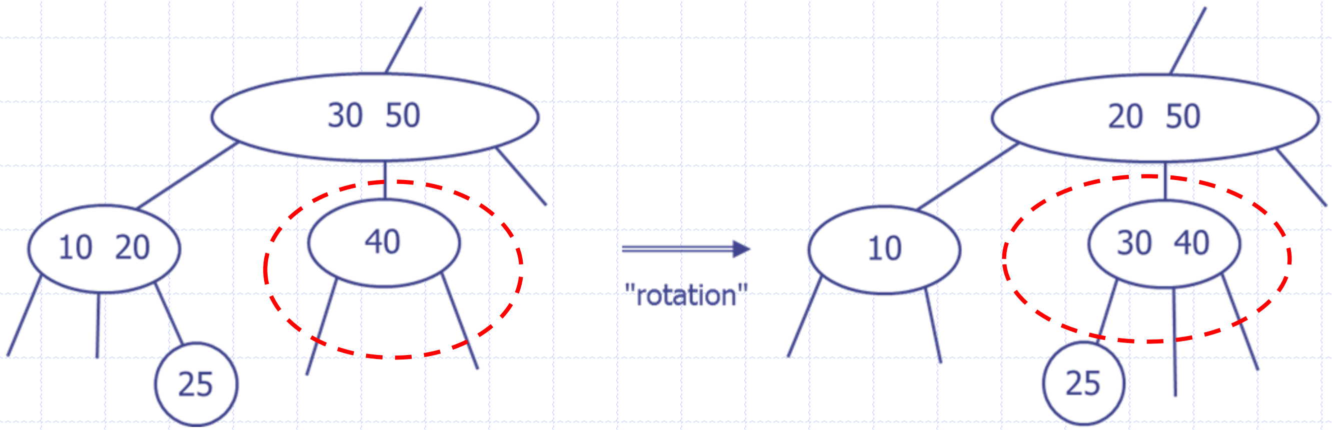
Prevent problem, then delete

2-3-4 Tree: Deletion

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

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



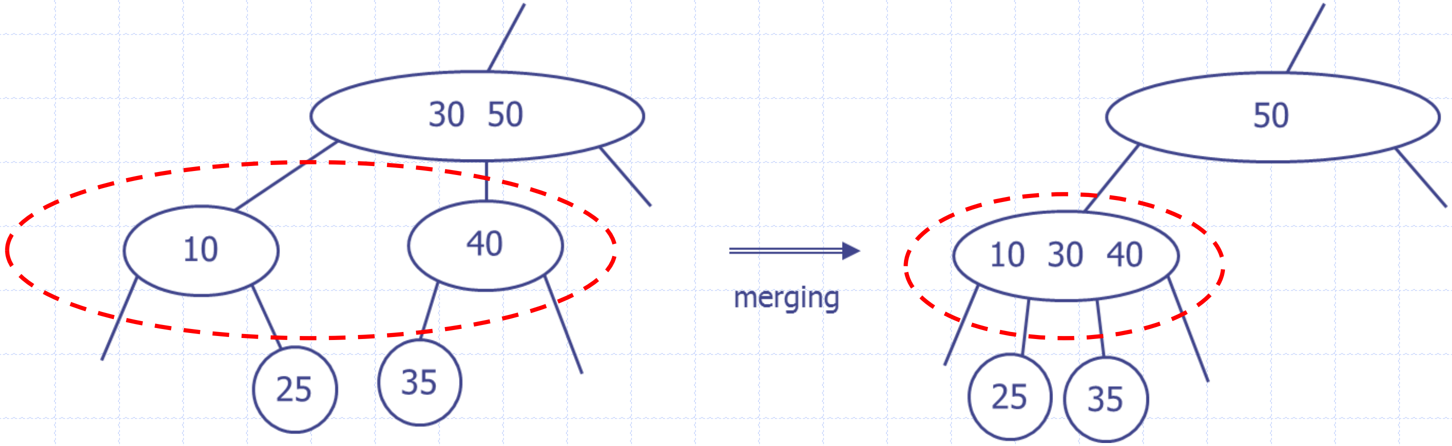
2-3-4 Tree: Deletion

Turning a 2-node into a 4-node ...

Case 2: each adjacent sibling has only one item

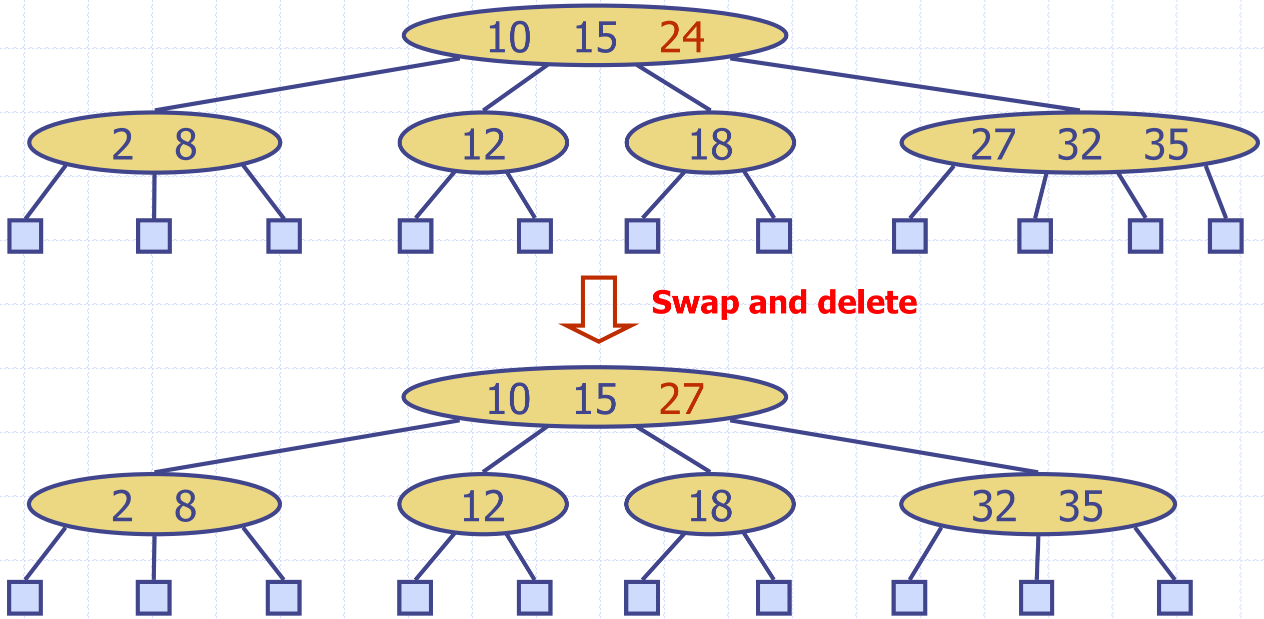
→ "steal" item from parent and merge node with sibling

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



Deletion - more example

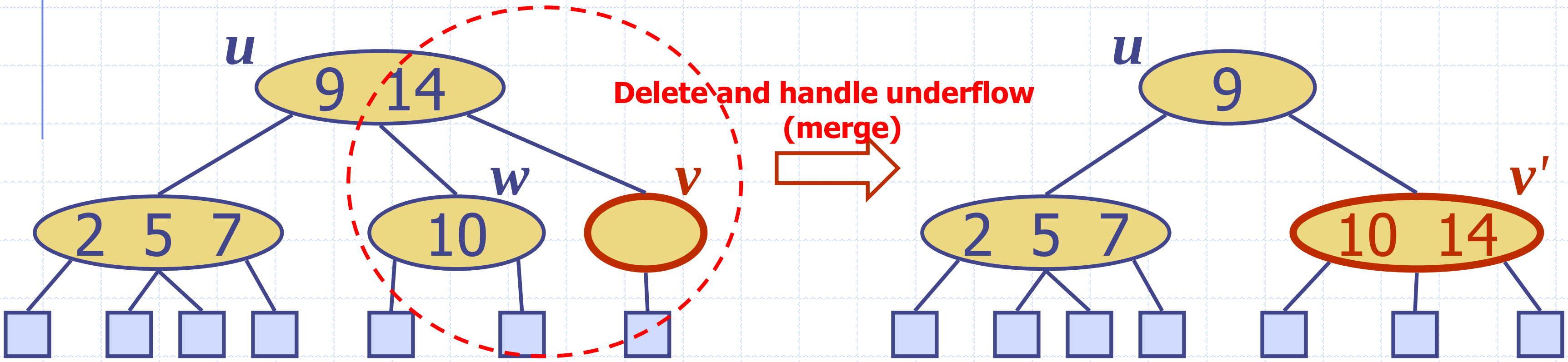
- Example: to delete key 24, we replace it with 27 (inorder successor)



Deletion - more example

the adjacent siblings of *v* are 2-nodes

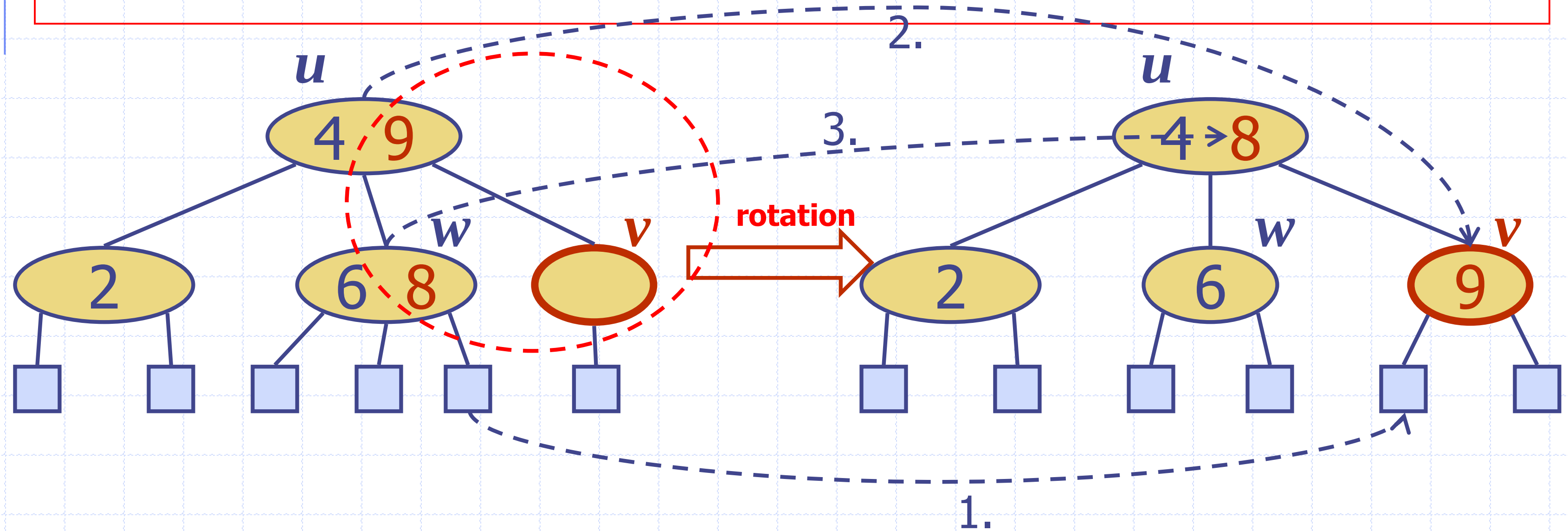
- merge *v* with an adjacent sibling *w* and move an item from *u* to the merged node *v*′
- After merging, the underflow may propagate to the parent *u*



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



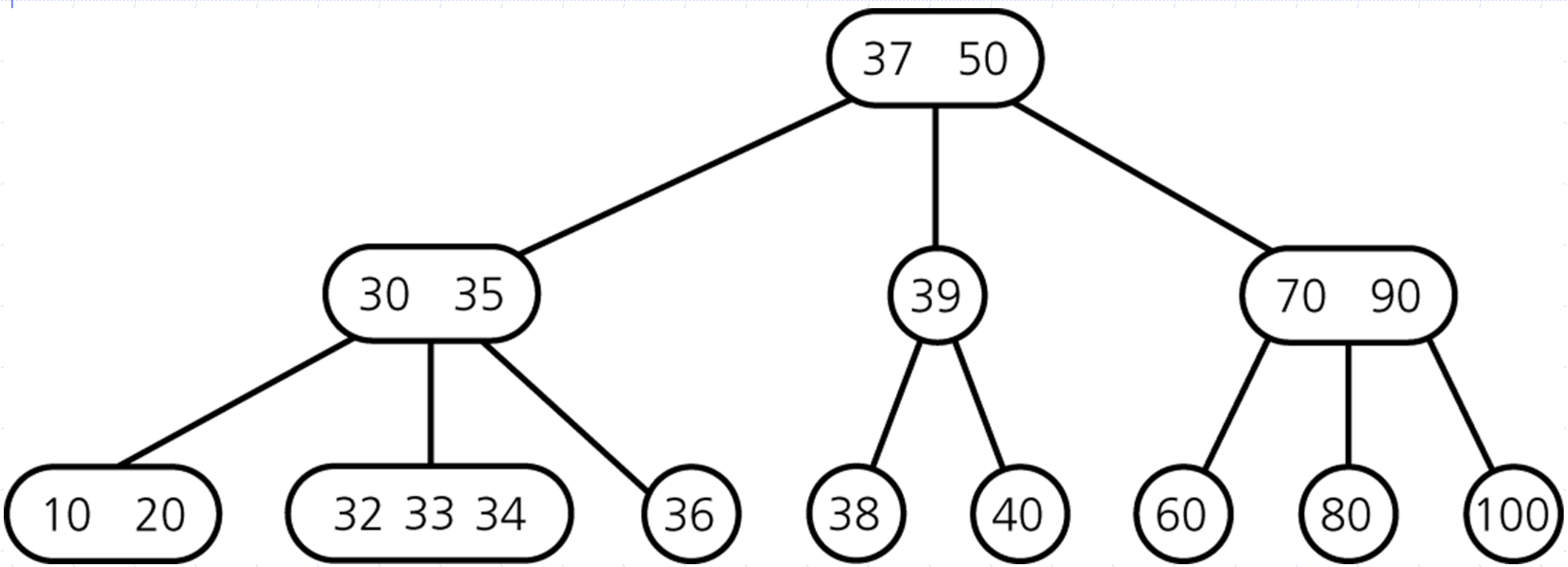
Analysis of Deletion
削除の分析

- 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

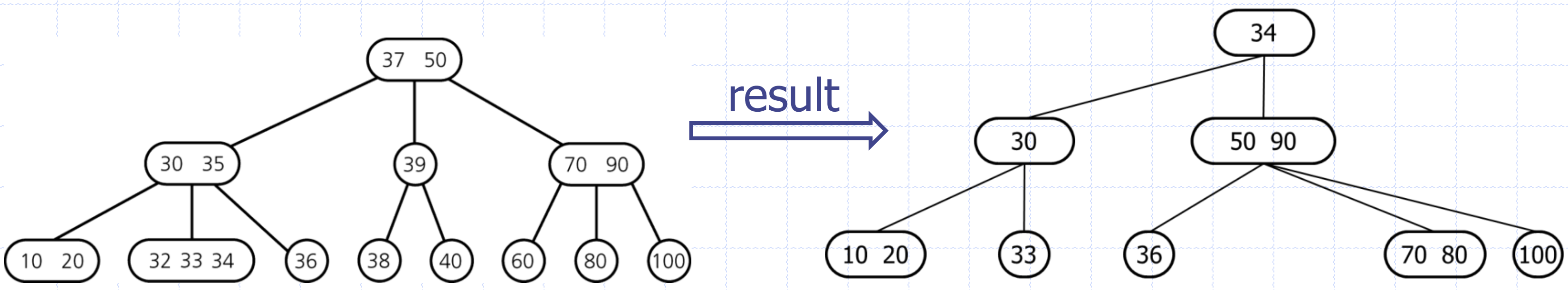
Delete 32, 35, 40, 38, 39, 37, 60



2-3-4 Tree: Deletion Practice

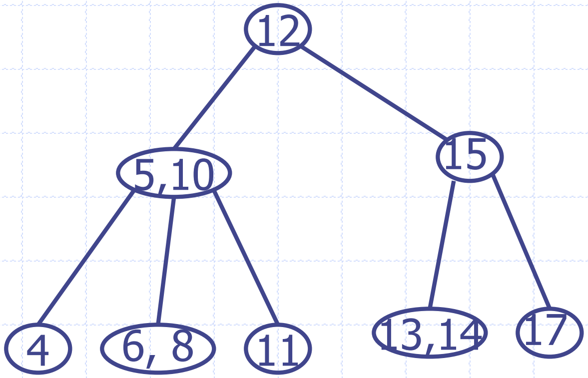
(solution)

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



Exercise 5-3（move to L6）

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。