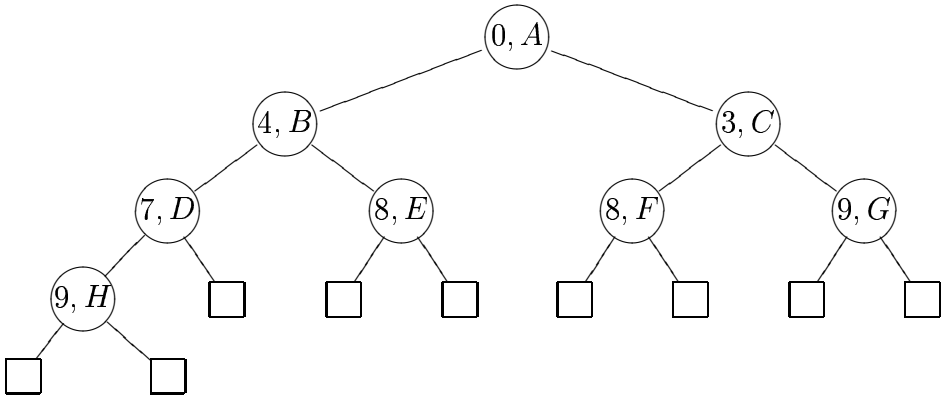
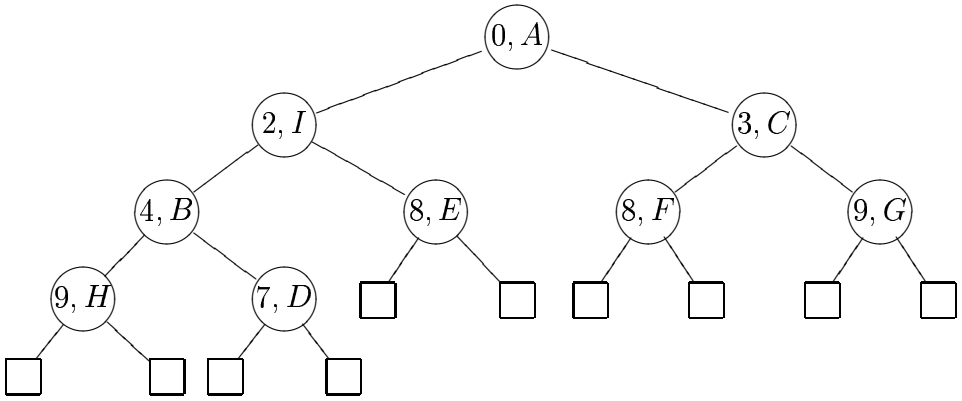
COSC 2011 3.0 Fundamentals of Data Structures
Solution of Final Exam
August 27, 2001; 19:00–22:00

1 Heap (10)

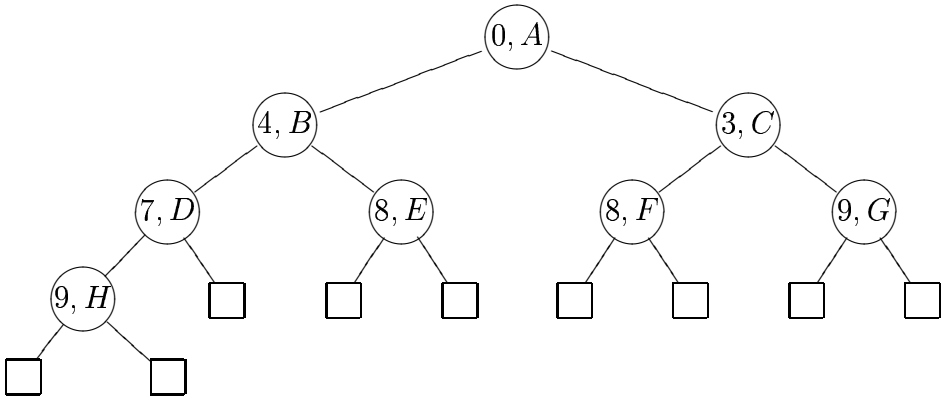
Consider the following heap.



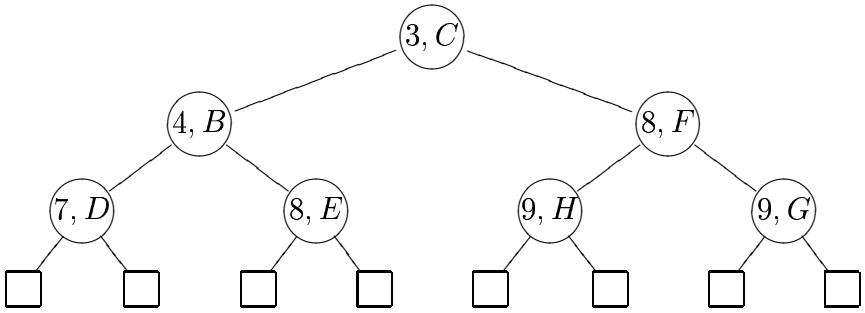
Draw the heap after insertItem$(2, I)$.



Consider the following heap.

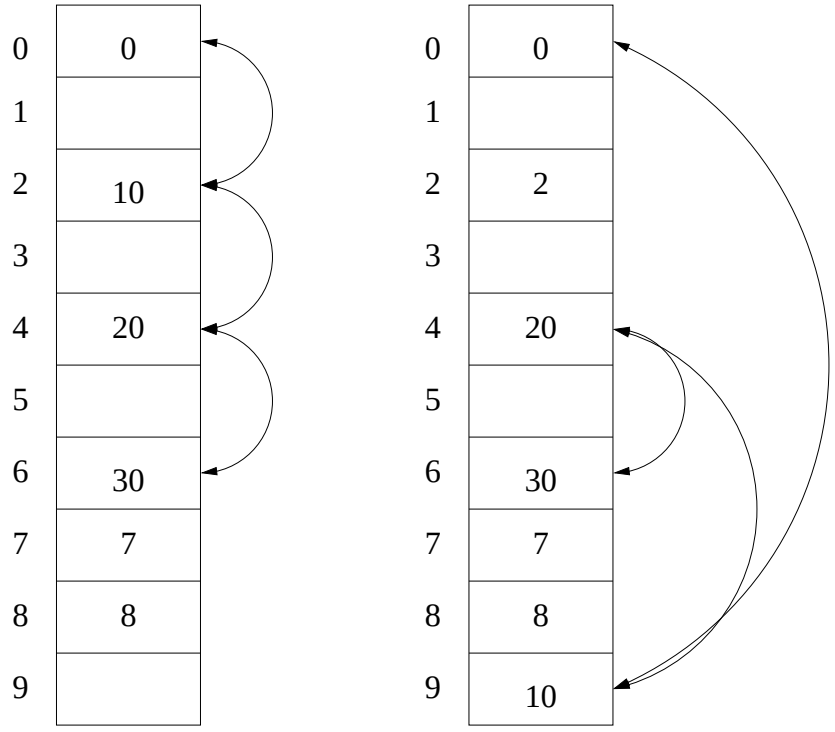


Draw the heap after removeMin*().*



2 Chained hashing with replacement (10)

Consider the hash table on the left.



The hash table has 10 buckets. The arrows denote the chains. Draw the hash table after the insertion of 2 by completing the table on the right. Motivate your answer.

2 goes in bucket 2. 10 needs to be relocated. This is done by following the chain and linear probing. The chain stays the same.

3 Skip list (10)

Consider an empty skip list, in which we want to insert the items $(1, A)$*,* $(2, B)$*,* $(3, C)$ *and* $(4, D)$*, subsequently. Recall that we use coin flips to determine the height of the inserted*

towers: heads (H) means that we add another node to the tower and continue building the tower and tails (T) means that we stop building the tower.

(a) *Which of the following sequences of coin flips is better: H T H T H T H T or H H H T H H H T H H H T H H H T? Motivate your answer.*

H T H T H T H T since the towers are less high and therefore insert, find and remove are more efficient.

(b) *Which of the following sequences of coin flips is better: H T H T H T H T or H T H H T H T H H H T? Motivate your answer.*

H T H H T H T H H H T is better since there are opportunities to skip and hence insert and find will be more efficient (remove may be less efficient as the tower to be removed may be bigger).

4 insertItem (10)

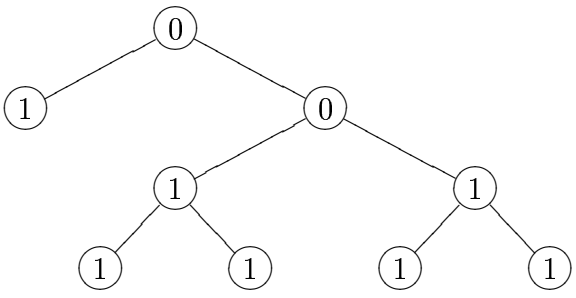
What is the worst-case running time of insertItem *of a dictionary implemented by means of an AVL tree, where the AVL tree is implemented either by means of a linked structure or an array and where the dictionary contains n items? Motivate your answers.*

linked structure: $O(\log(n))$
motivation: height of the AVL is $O(\log(n))$, rotation is $O(1)$

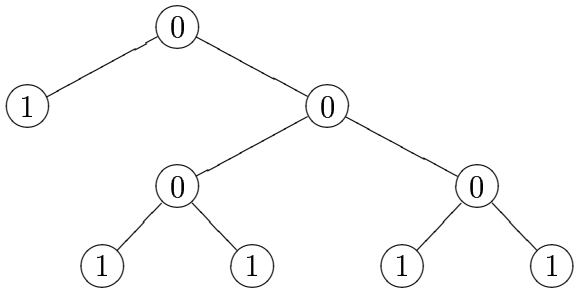
array: $O(n)$
motivation: rotation is $O(n)$ since large parts of the tree may have to be moved in the tree

5 Recursion (15)

Consider a binary tree whose nodes either contain a 0 or a 1. Give a recursive *algorithm which tests if each path from the root of the binary tree to a leaf contains as many 0's as 1's. For example, the binary tree*



has the above described property, but the binary tree



does not. Ideally, your algorithm should run in $O(n)$, where n is the number of nodes of the binary tree. You may either use pseudocode or Java code. You may introduce auxiliary algorithms/methods. Your algorithm should not depend on how the binary tree is implemented.

zerosAndOnes(*tree*):
  input: binary tree, each node of which contains either a 0 or a 1
  output: each path from the root of *tree* to a leaf contains as many 0's as 1's?
return zerosAndOnes(*tree*, root of *tree*, 0)

zerosOnes(*tree*, *node*, *number*):
  input: binary tree, each node of which contains either a 0 or a 1; a node of *tree*; integer
  output: each path from *node* to a leaf contains *number* more 0's than 1's?
if *node* contains a 0 **then**
    $number \leftarrow number - 1$
else
    $number \leftarrow number + 1$
if *node* is a leaf **then**
    return $(number = 0)$
else
    return (zerosAndOnes(*tree*, left child of *node*, *number*) and zerosAndOnes(*tree*, right child of *node*, *number*)

```
/**
 * Tests if each path from the root of the specified binary tree to a leaf
 * contains as many 0's as 1's.
 *
 * @param tree Binary tree, each node of which either contains a 0 or a 1.
 * @return true if each path from the root of the specified binary tree to
 * a leaf contains as many 0's as 1's, false otherwise.
 */
public static boolean zerosAndOnes(BinaryTree tree)
{
    return zerosAndOnes(tree, tree.root(), 0);
}

/**
```

```
 * Tests if each path from the specified node to a leaf contains the specified
 * number more 0's than 1's.
 *
 * @param tree Binary tree, each node of which either contains a 0 or a 1.
 * @param node Node of the binary tree.
 * @param number Integer.
 * @return true if each path from the specified node to a leaf contains the
 * specified number more 0's than 1's.
 */
public static boolean zerosAndOnes(BinaryTree tree, Position node, int number)
{
    if (((Integer) node.element()).intValue() == 0)
    {
        number--;
    }
    else
    {
        number++;
    }
    if (tree.isExternal(node))
    {
        return (number == 0);
    }
    else
    {
        return (zerosAndOnes(tree, tree.leftChild(node), number) &&
                zerosAndOnes(tree, tree.rightChild(node), number));
    }
}
```

6 Path in graph (20)

Consider a connected simple undirected graph whose vertices contain a 0 or a 1. Give an algorithm which tests if there is a path from vertex source to vertex target such that all vertices of the path contain a 0. You may either use pseudocode or Java code. You may introduce auxiliary algorithms/methods. Your algorithm should not depend on how the graph is implemented.

path(*graph*, *source*, *target*):
  input: connected simple undirected graph; vertex of *graph*; vertex of *graph*
  output: there is a path from *source* to *target* such that all the vertices of the path contain a 0?
for each vertex *vertex* of *graph* **do**
    mark *vertex* unvisited
return (*source* contains 0 and zeroSearch(*graph*, *source*, *target*)

zeroSearch(*graph*, *vertex*, *target*):
  input: connected simple undirected graph; vertex of *graph*; vertex of *graph*
  output: there is a path from *vertex* to *target* the vertices of which all contain a 0?
  precondition: vertex *vertex* contains a 0
mark *vertex* visited
found ← (*vertex* = *target*)
for each vertex *adjacent* adjacent to *vertex* **while** not *found* **do**
    if *adjacent* is not visited and *adjacent* contains a 0 **then**
        found ← *found* or zeroSearch(*graph*, *adjacent*, *target*)
return *found*

```
/**
 * Tests if there is a path between the specified vertices of the specified
 * graph such that all the vertices of the path contain a 0.
 *
 * @param graph Connected simple undirected graph.
 * @param source Vertex of the graph.
 * @param target Vertex of the graph.
 * @return true if there exists a path between source and target with all
 * vertices on the path containing 0, false otherwise.
 */
public static boolean path(SimpleGraph graph, Vertex source, Vertex target)
{
    Iterator vertices = graph.vertices();
    while (vertices.hasNext())
    {
        ((DecorableVertex) vertices.next()).unvisit();
    }
    return (((Integer) source.element()).intValue() == 0 &&
            zeroSearch(graph, source, target));
}

/**
 * Tests if there is a path between the specified vertices of the
 * specified graph such that all the vertices of the path contain 0.
 * It assumed that vertex contains a 0.
 *
 * @param graph Connected simple undirected graph.
 * @param vertex Vertex of the graph.
 * @param target Vertex of the graph.
 * @return true if there exists a path between vertex and target with all
 * vertices on the path containing 0, false otherwise.
 */
public static boolean zeroSearch(SimpleGraph graph, Vertex vertex, Vertex target)
{
```

```
    ((DecorableVertex) vertex).visit();
    boolean found = (vertex == target);
    Iterator adjacentVertices = graph.adjacent(vertex);
    while (adjacentVertices.hasNext() && !found)
    {
        DecorableVertex adjacent = (DecorableVertex) adjacentVertices.next();
        if (!adjacent.isVisited() && ((Integer) adjacent.element()).intValue() == 0)
        {
            found = found || zeroSearch(graph, adjacent, target);
        }
    }
    return found;
}
```

7 AVL tree (10)

Consider a binary tree whose internal nodes contain items and whose external nodes are empty. To simplify matters a little, you may assume that the keys are integers. Give an algorithm which tests if the binary tree is an AVL tree. Your algorithm should run in $O(n)$, where n is the number of nodes of the tree. You may either use pseudocode or Java code. You may introduce auxiliary algorithms/methods. Your algorithm should not depend on how the binary tree is implemented.

isAVL(*tree*):
input: binary tree whose internal nodes contain items and whose external nodes are empty
output: is *tree* an AVL tree?
(*isAVL*, *height*, *max*, *min*) ← isAVL(*tree*, root of *tree*)
return *isAVL*

isAVL(*tree*, *node*):
input: binary tree whose internal nodes contain items and whose external nodes are empty; node of *tree*
output: (is subtree rooted at *node* an AVL tree?, height of the subtree rooted at *node*, maximal key of the subtree rooted at *node*, minimal key of the subtree rooted at *node*)
if *node* is a leaf **then**
    return (true, 0, $-\infty$, $+\infty$)
else
    ($isAVL_l$, $height_l$, max_l, min_l) ← isAVL(*tree*, left child of *node*)
    ($isAVL_r$, $height_r$, max_r, min_r) ← isAVL(*tree*, right child of *node*)
    $max \leftarrow \max\{max_l, max_r, \text{key of } node\}$
    $min \leftarrow \min\{min_l, min_r, \text{key of } node\}$
    $height \leftarrow 1 + \max\{height_l, height_r\}$
    $isAVL \leftarrow$ (key of $node \geq max_l$) and (key of $node \leq min_r$) and ($|height_l - height_r| \leq 1$) and $isAVL_l$ and $isAVL_r$
    return (*isAVL*, *height*, *max*, *min*)

```
/**
 * Tests is the specified binary tree is an AVL tree.
 * It is assumed that the internal nodes of the binary
 * tree contain items and that the external nodes of
 * the binary tree are empty.
 *
 * @param tree The binary tree.
 * @return true if the specified binary tree is an AVL
 * tree, false otherwise.
 */
public static void isAVL(BinaryTree tree)
{
    Info info = isAVL(tree, tree.root());
    return info.getIsAVL();
}

/**
 * Returns the following information about the subtree
 * of the specified tree rooted at the specified node:
 * is the subtree an AVL tree?
 * height of the subtree,
 * maximal key of the subtree,
 * minimal key of the subtree.
 *
 * @param The binary tree.
 * @param Node of the bianry tree.
 * @return Information about the subtree.
 */
public Info isAVL(BinaryTree tree, Position node)
{
    if (tree.isExternal(node))
    {
        return new Info(true, 0, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
    else
    {
        Info left = isAVL(tree, tree.leftChild(node));
        Info right = isAVL(tree, tree.rightChild(node));
        int key = ((Integer) ((Item) node.element()).key()).intValue();
        boolean isAVL = left.getIsAVL() &&
                        right.getIsAVL() &&
                        (Math.abs(left.getHeight() - right.getHeight()) <= 1) &&
                        (key >= left.getMax()) &&
                        (key <= right.getMin());
```

```
        return new Info(isAVL, height, max, min);
    }
}

/**
 * Information about a subtree.
 */
public class Info
{
    private boolean isAVL; // is the subtree an AVL tree?
    private int height;    // height of the subtree
    private int max;       // maximal key of the subtree
    private int min;       // minimal key of the subtree

    /**
     * Constructs an Info object with the specified information.
     *
     * @param isAVL Is the subtree an AVL tree?
     * @param height The height of the subtree.
     * @param max The maximal key of the subtree.
     * @param min The minimal key of the subtree.
     */
    public Info(boolean isAVL, int height, int max, int min)
    {
        this.isAVL = isAVL;
        this.height = height;
        this.max = max;
        this.min = min;
    }

    /**
     * Returns the of the subtree.
     *
     * @return The of the subtree.
     */
    public boolean getIsAVL()
    {
        return isAVL;
    }

    /**
     * Returns the height of the subtree.
     *
     * @return The height of the subtree.
```

```
     */
    public int getHeight()
    {
        return height;
    }

    /**
     * Returns the minimal key of the subtree.
     *
     * @return The minimal key of the subtree.
     */
    public int getMax()
    {
        return max;
    }

    /**
     * Returns the minimal key of the subtree.
     *
     * @return The minimal key of the subtree.
     */
    public int getMin()
    {
        return min;
    }
}
```