5. Trees
Part -2

5.6. Balancing a tree

- BSTs where introduced because in theory they give nice fast search time.
- We have seen that depending on how the data arrives the tree can degrade into a linked list
- So what is a good programmer to do.
- Of course, they are to balance the tree

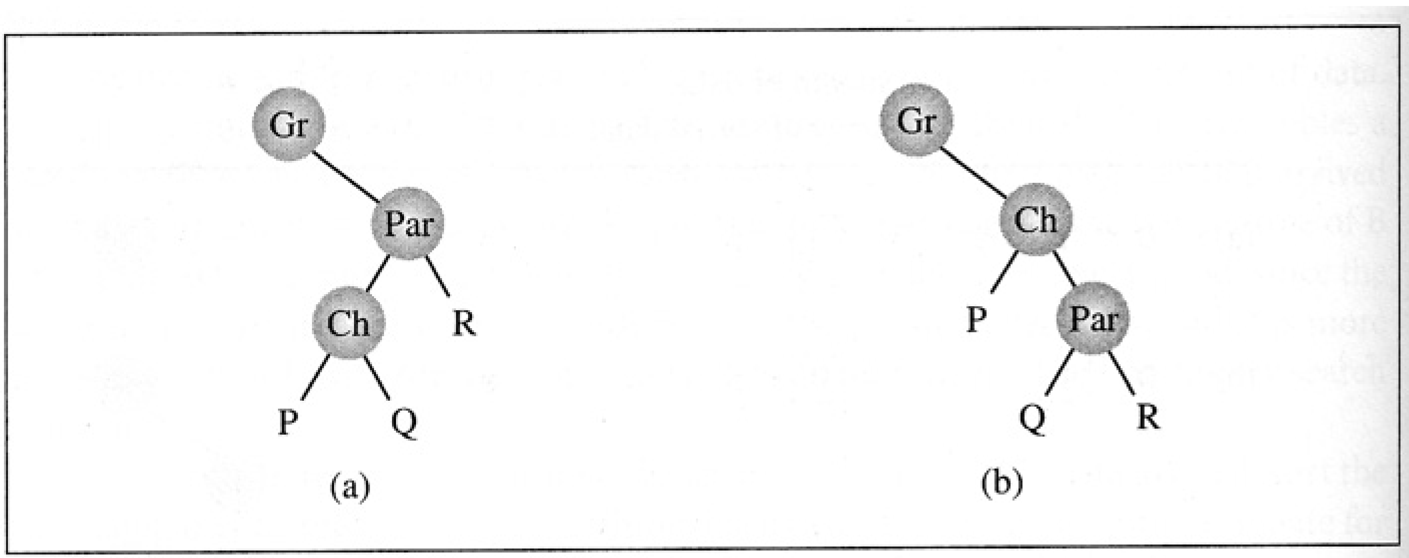
5.6. Balancing a tree -ideas

- One idea would be to get all of the data first, and store it in an array
- Then sort the array and then insert it in a tree
- Of course this does have some drawbacks so we need another idea

5.6.1. DSW Trees

- Named for Colin Day and then for Quentin F. Stout and Bette L. Warren, hence DSW.
- The main idea is a rotation
- rotateRight(Gr, Par, Ch)
  - If Par is not the root of the tree
    - Grandparent Gr of child Ch, becomes Ch's parent by replacing Par;
  - Right subtree of Ch becomes left subtree of Ch's parent Par;
  - Node Ch aquires Par as its right child

Maybe a picture will help



(a) (b)

5.6.1.1. More of the DSW

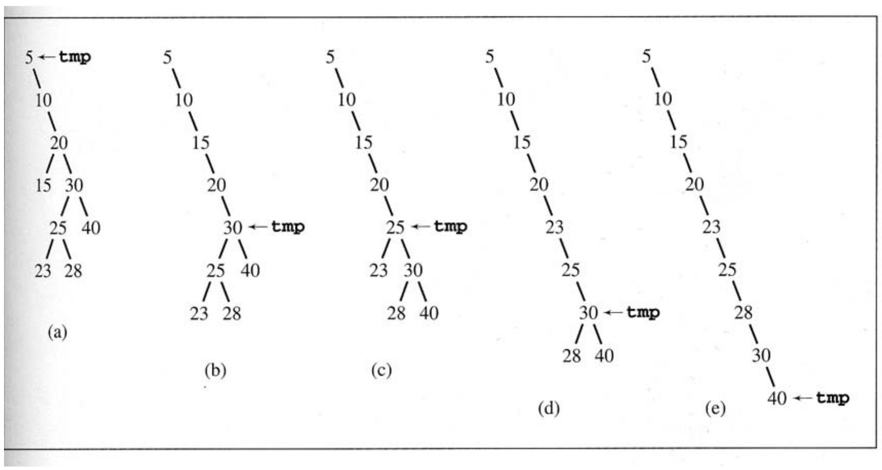
- So the idea is to take a tree and perform some rotations to it to make it balanced.
- First you create a backbone or a vine
- Then you transform the backbone into a nicely balanced tree

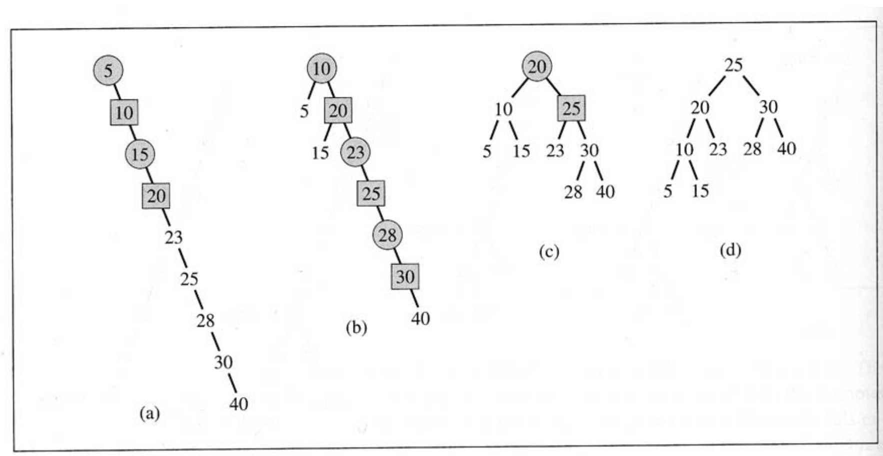
5.6.1.2. Algorithms

- createBackbone(root, n)
  - Tmp = root
  - While (Tmp != 0)
    - If Tmp has a left child
      - Rotate this child about Tmp
      - Set Tmp to the child which just became parent
    - Else set Tmp to its right child

- createPerfectTree(n)
  - $M = 2^{\text{floor}[\lg(n+1)]}-1$;
  - Make n-M rotations starting from the top of the backbone;
  - While (M > 1)
    - M = M/2;
    - Make M rotations starting from the top of the backbone;

Maybe some more pictures





5.6.1.3. Wrap-up

- The DSW algorithm is good if you can take the time to get all the nodes and then create the tree
- What if you want to balance the tree as you go?
- You use an AVL Tree

5.6.2. AVL Trees

- Named after its inventors Adel'son-Vel'skii and Landis, hence AVL
- The heights of any subtree can only differ by at most one.
- Each nodes will indicate balance factors.
- Worst case for an AVL tree is 44% worst then a perfect tree.
- In practice, it is closer to a perfect tree.

5.6.2.1. What does an AVL do?

- Each time the tree structure is changed, the balance factors are checked and if an imbalance is recognized, then the tree is restructured.
- For insertion there are four cases to be concerned with.
- Deletion is a little trickier.

5.6.2.2. AVL Insertion

- Case 1: Insertion into a right subtree of a right child.
  - Requires a left rotation about the child
- Case 2: Insertion into a left subtree of a right child.
  - Requires two rotations
    - First a right rotation about the root of the subtree
    - Second a left rotation about the subtree's parent

Some more pictures

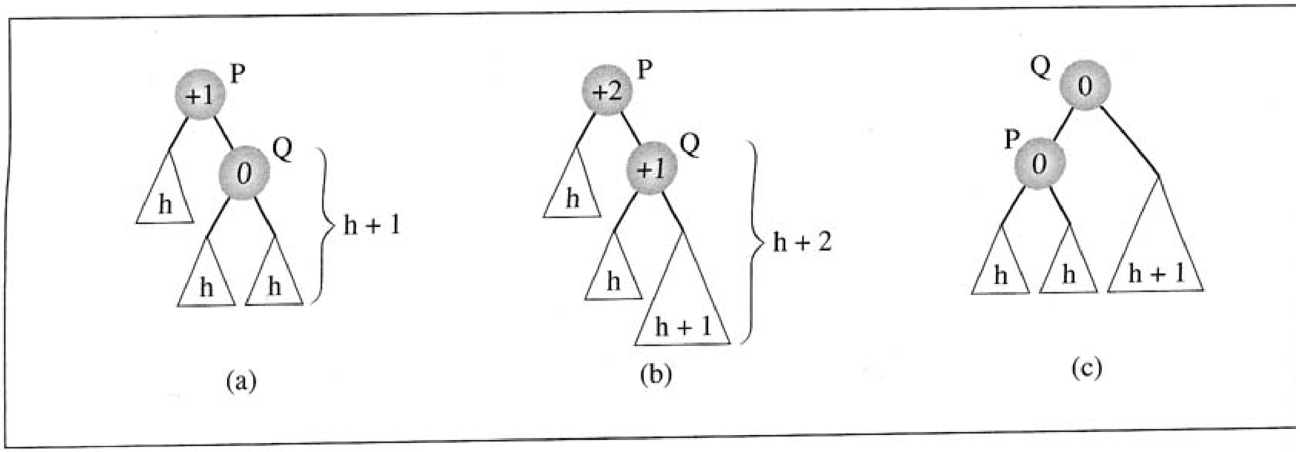
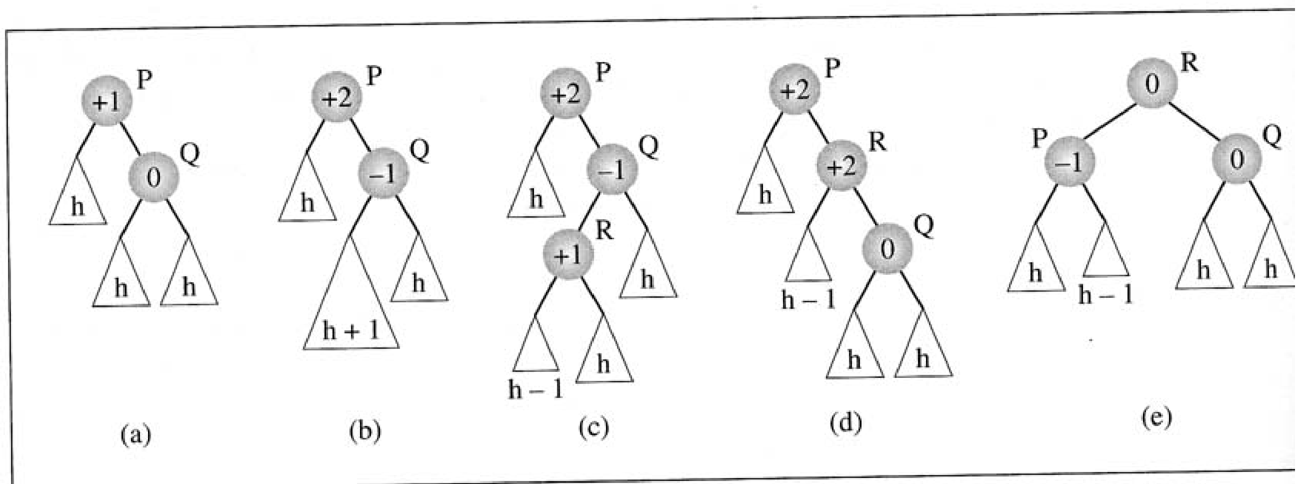


FIGURE 6.42 Balancing a tree after insertion of a node in the left subtree of node Q.



5.6.2.3. Deletion

- Deletion is a bit trickier.
- With insertion after the rotation we were done.
- Not so with deletion.
- We need to continue checking balance factors as we travel up the tree

5.6.2.4. Deletion Specifics

- Go ahead and delete the node just like in a BST.
- There are 4 cases after the deletion:

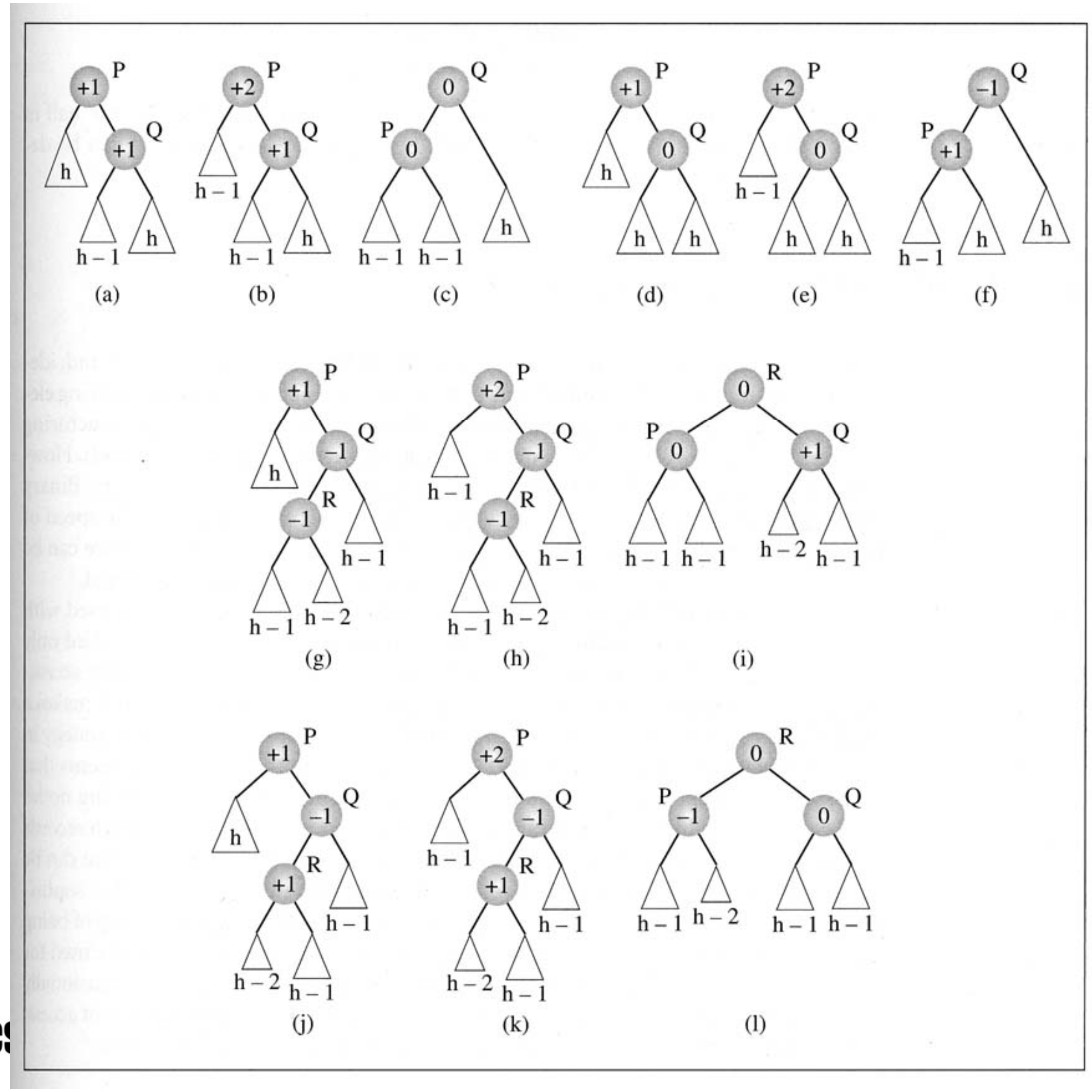
Cases

- Case 1: Deletion from a left subtree from a tree with a right high root and a right high right subtree.
  - Requires one left rotation about the root
- Case 2: Deletion from a left subtree from a tree with a right high root and a balanced right subtree.
  - Requires one left rotation about the root

Cases continued

- Case 3: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a left high left subtree.
  - Requires a right rotation around the right subtree root and then a left rotation about the root
- Case 4: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a right high left subtree
  - Requires a right rotation around the right subtree root and then a left rotation about the root

Definitely some pictures



5.7. Self-adjusting Trees

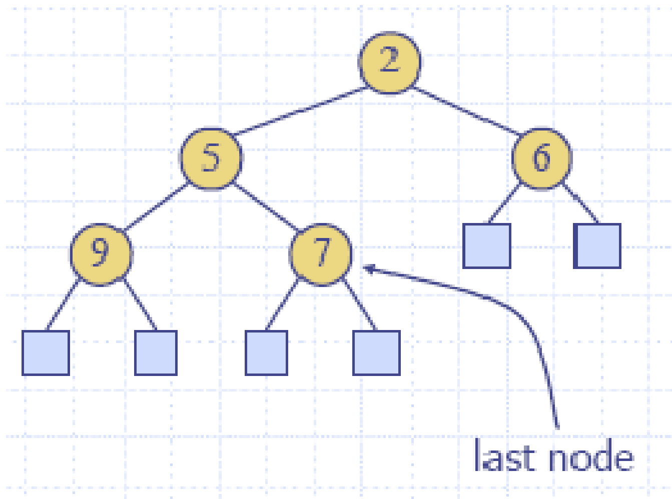
- The previous sections discussed ways to balance the tree after the tree was changed due to an insert or a delete.
- There is another option.
- You can alter the structure of the tree after you access an element
  - Think of this as a self-organizing tree

5.8. Heaps

- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - **Heap-Order**: for every internal node v other than the root,
    $key(v) \geq key(parent(v))$
  - **Complete Binary Tree**: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are $2i$ nodes of depth I
    - at depth $h - 1$, the internal nodes are to the left of the external nodes

5.8. Heaps contd…

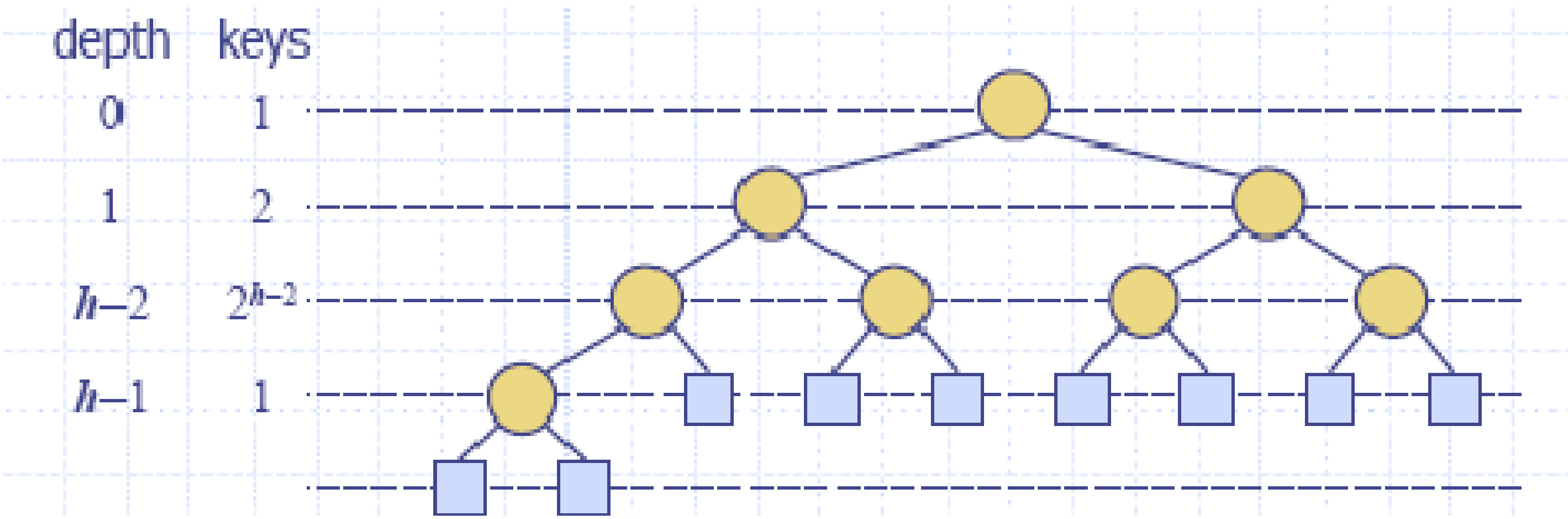
–The last node of a heap is the rightmost internal node of depth ***h*** – 1



5.8.1. Height of a Heap

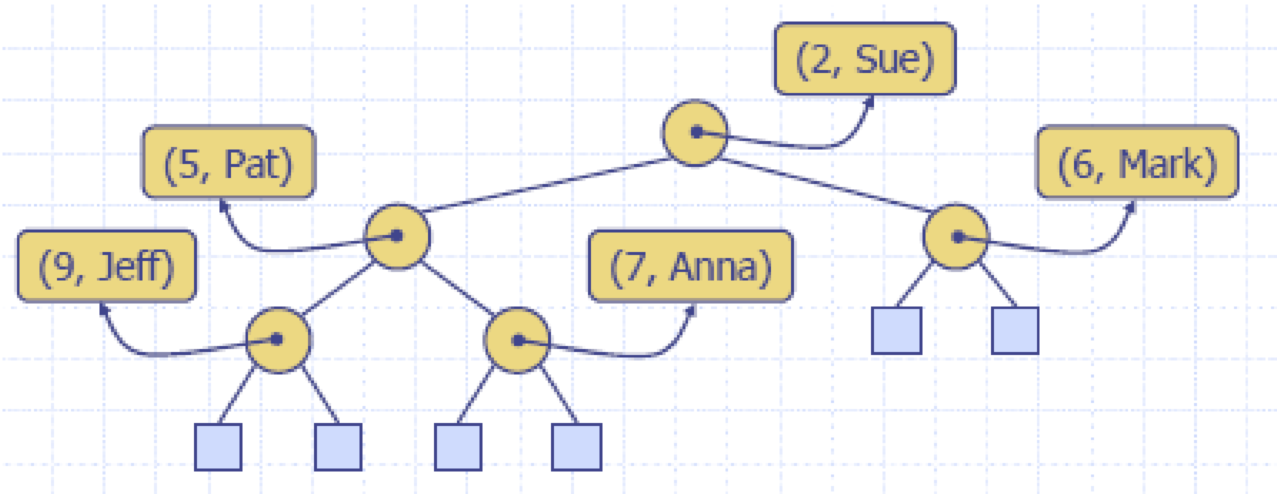
- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are $2i$ keys at depth $i = 0, \ldots, h - 2$ and at least one key at depth $h - 1$, we have $n \geq 1 + 2 + 4 + \ldots + 2h-2 + 1$
  - Thus, $n \geq 2h-1$, i.e., $h \leq \log n + 1$

5.8.1. Height of a Heap contd…



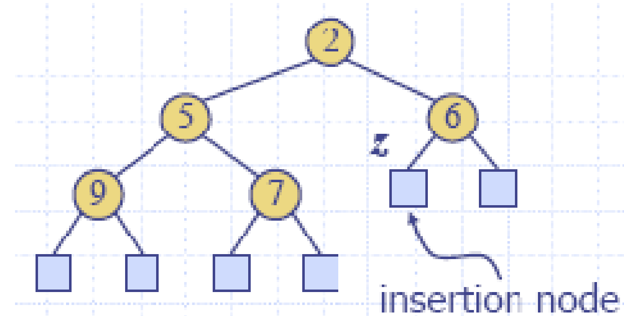
5.8.2. Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures

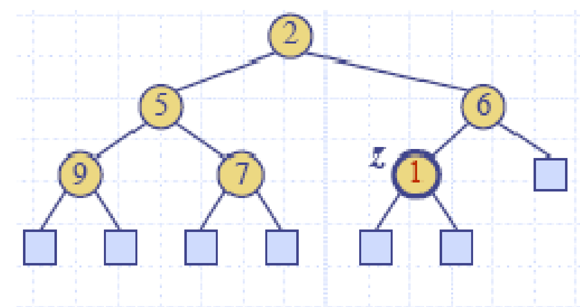


5.8.3. Insertion into a Heap

- Method insertItem of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap



- The insertion algorithm consists of three steps
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z*** and expand ***z*** into an internal node
  - Restore the heap-order property



5.8.4. Removal from a Heap

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Compress ***w*** and its children into a leaf
  - Restore the heap-order property