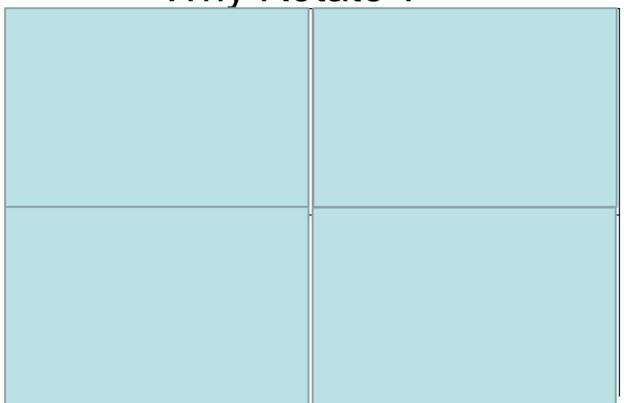
Why Rotate ?

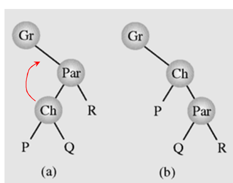


The DSW Algorithm

- Colin **D**ay, Quentin F. **S**tout, Bette L.**W**arren
- The building block for tree transformations in this algorithm is the **rotation**
- There are two types of rotation, left and right, which are symmetrical to one another

The DSW Algorithm (continued)

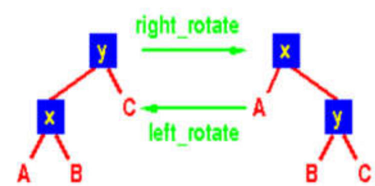
```
rotateRight (Gr, Par, Ch)
    if Par is not the root of the tree          // i.e., if Gr is not null
        grandparent Gr of child Ch becomes Ch's parent;
    right subtree of Ch becomes left subtree of Ch's parent Par;
    node Ch acquires Par as its right child;
```



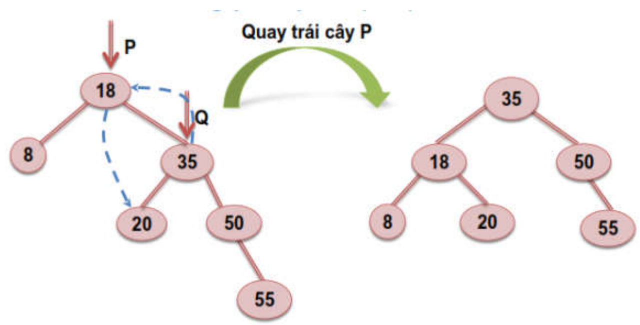

```
Par.left = Ch.right;   //1
Ch.right=Par;          //2
Gr.right = Ch;         //3
```

Figure 6-37 Right rotation of child Ch about parent Par

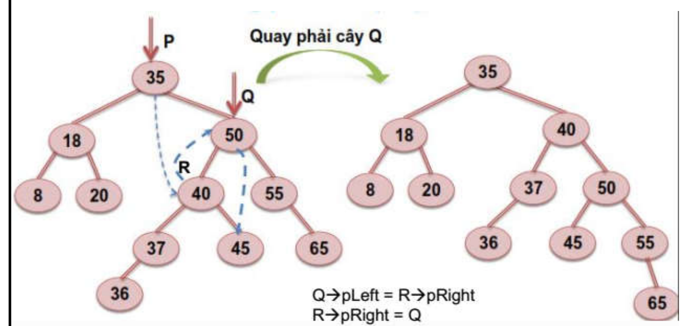
Rotate RIGHT - LEFT



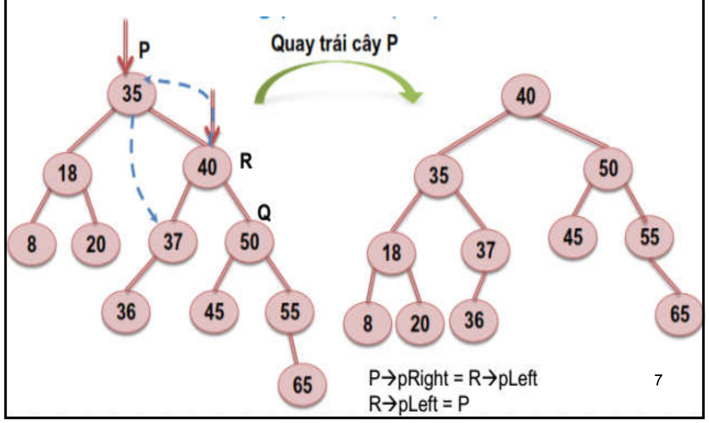
Example



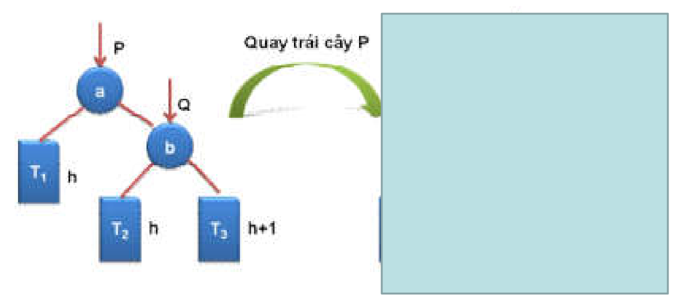
Example



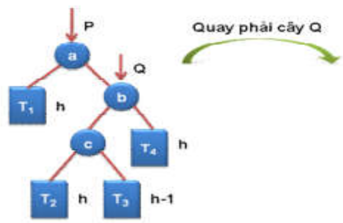
Example



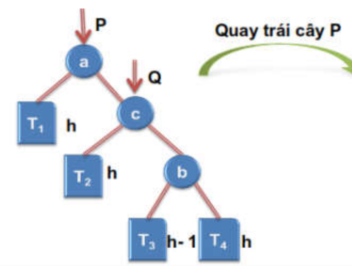
Example



Example



Example



The DSW Algorithm (continued)

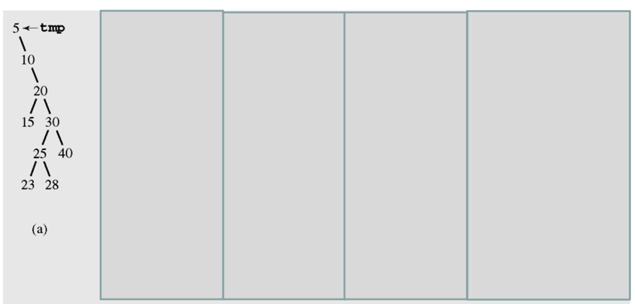


Figure 6-38 Transforming a binary search tree into a backbone (Use the DSW rotateRight algorithm)

Degrading tree
Cây suy biến

The DSW Algorithm (continued)

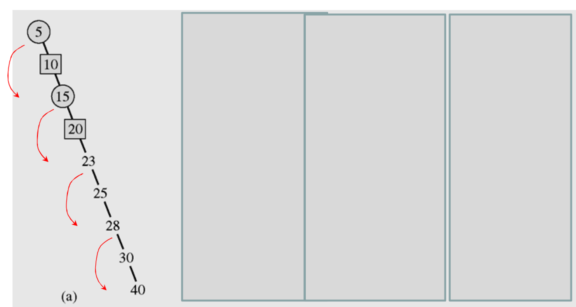


Figure 6-39 Transforming a backbone into a perfectly balanced tree

DAY 4

Giới thiệu

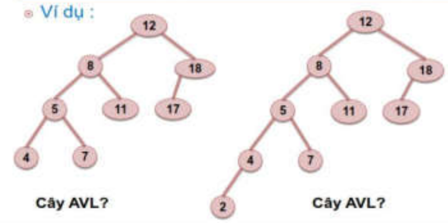
- Do G.M. **A**delsen **V**elskii và E.M. **L**endis đưa ra vào năm 1962, đặt tên là cây AVL.

Định nghĩa

- Cây cân bằng AVL là cây nhị phân tìm kiếm mà tại mỗi đỉnh của cây, độ cao của cây con trái và cây con phải **không chênh lệch quá 1**.

Cây AVL

- Ví dụ :



Xây dựng cây cân bằng

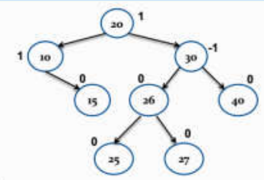
- Việc xây dựng cây cân bằng dựa trên cây nhị phân tìm kiếm, chỉ bổ sung thêm 1 giá trị cho biết sự cân bằng của các cây con như thế nào.
- Cách làm gợi ý:

```
struct NODE {
    Data key;
    NODE *pLeft, *pRight;
    int bal;
};
```

- Trong đó giá trị bal (balance, cân bằng) có thể là: 0: cân bằng; -1: lệch trái; 1: lệch phải

Mô tả cấu trúc cây AVL

Hệ số cân bằng của các nút trong cây AVL

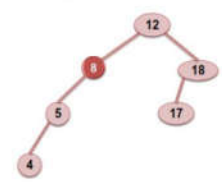


Các trường hợp mất cân bằng

- Mất cân bằng trái-trái (L-L)
- Mất cân bản trái-phải (L-R)
- Mất cân bằng phải-phải (R-R)
- Mất cân bằng phải-trái (R-L)

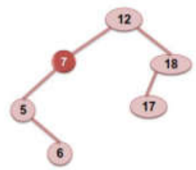
Các trường hợp mất cân bằng

- Mất cân bằng trái-trái (L-L)



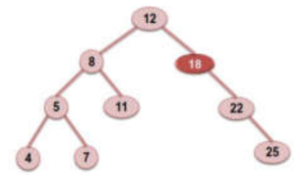
Các trường hợp mất cân bằng

- Mất cân bằng trái-phải (L-R)



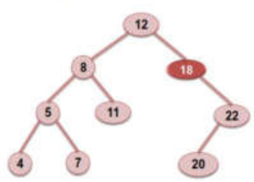
Các trường hợp mất cân bằng

- Mất cân bằng phải-phải (R-R)



Các trường hợp mất cân bằng

- Mất cân bằng phải-trái (R-L)

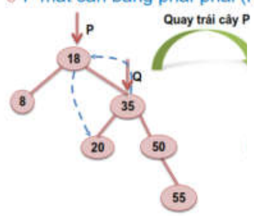


Xử lý mất cân bằng

- Giả sử tại một node cây xảy ra mất cân bằng bên phải (cây con phải chênh lệch với cây con trái hơn một đơn vị):
  - Mất cân bằng phải-phải (RR)
    - Quay trái
  - Mất cân bằng phải-trái (R-L)
    - Quay phải
    - Quay trái

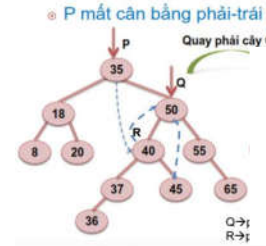
Xử lý mất cân bằng

- P mất cân bằng phải-phải (RR):



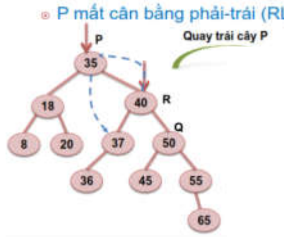
Xử lý mất cân bằng

- P mất cân bằng phải-trái (RL) – Bước 1:



Xử lý mất cân bằng

- P mất cân bằng phải-trái (RL) - Bước 2:

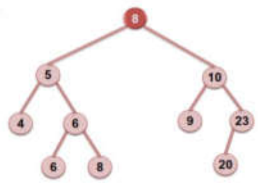


Xử lý mất cân bằng

- Khi một node cây xảy ra mất cân bằng bên trái (cây con trái chênh lệch với cây con phải hơn một đơn vị): (thực hiện đối xứng với trường hợp mất cân bằng bên phải)
  - Mất cân bằng trái-trái (LL)
    - Quay phải
  - Mất cân bằng trái-phải (L-R)
    - Quay trái
    - Quay phải

Thao tác tìm kiếm

- Thực hiện hoàn toàn tương tự cây nhị phân tìm kiếm.



Thao tác thêm phần tử

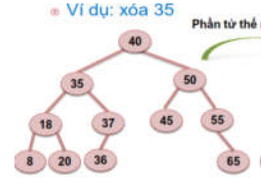
- Thực hiện tương tự với việc thêm phần tử của cây nhị phân tìm kiếm.
- Nếu xảy ra việc mất cân bằng thì xử lý bằng các trường hợp mất cân bằng đã biết.

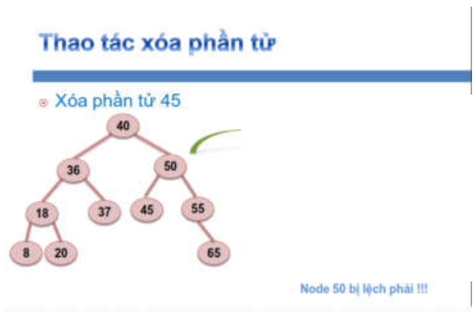
Thao tác xóa phần tử

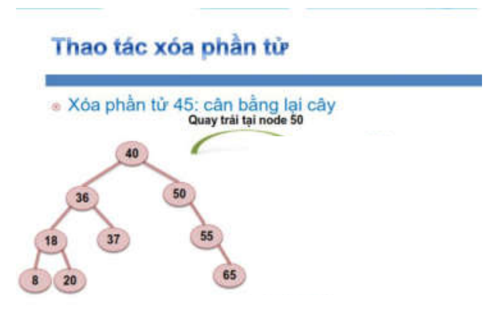
- Thực hiện tương tự cây nhị phân tìm kiếm: xét 3 trường hợp, và tìm phần tử thế mạng nếu cần.
- Sau khi xóa, nếu cây mất cân bằng, thực hiện cân bằng cây.
- Lưu ý: *việc cân bằng sau khi hủy có thể xảy ra dây chuyền.*

Thao tác xóa phần tử

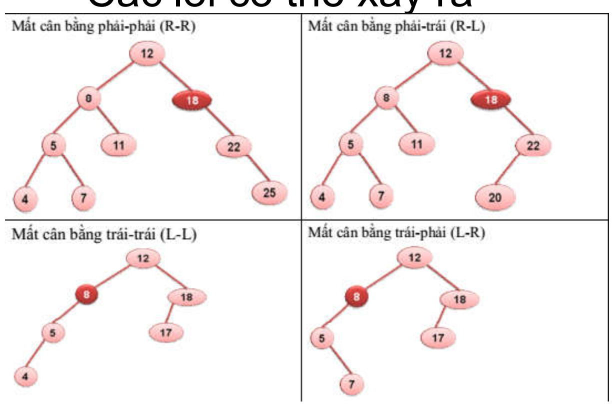
- Ví dụ: xóa 35







Các lỗi có thể xảy ra



Quy tắc

MẤT CÂN BẰNG PHẢI	
Mất cân bằng phải-phải (R-R) - Quay trái tại node bị mất cân bằng	Mất cân bằng phải-trái (R-L) - Quay phải tại node con phải của node bị mất cân bằng - Quay trái tại node bị mất cân bằng
MẤT CÂN BẰNG TRÁI	
Mất cân bằng trái-trái (L-L) - Quay phải tại node bị mất cân bằng	Mất cân bằng trái-phải (L-R) - Quay trái tại node con trái của node bị mất cân bằng - Quay phải tại node bị mất cân bằng

Bài tập

❖ Tạo cây AVL với các khóa lần lượt là: 30, 20, 10. Sau đó thêm lần lượt các khóa: 15, 40, 25, 27, 26, 5, 13, 14 vào cây trên.

❖

AVL Trees

- An **AVL tree** is one in which the height of the left and right subtrees of every node differ by at most one
- A balance factor is the height of the right subtree minus the height of the left subtree.

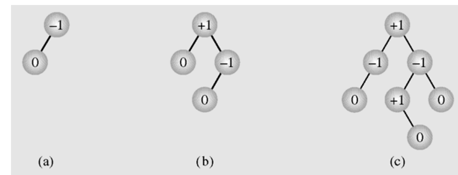


Figure 6-40 Examples of AVL trees

AVL Trees (continued)

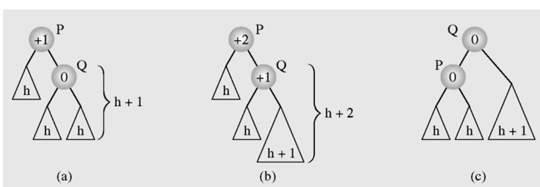


Figure 6-41 Balancing a tree after insertion of a node in the right subtree of node *Q*

AVL Trees (continued)

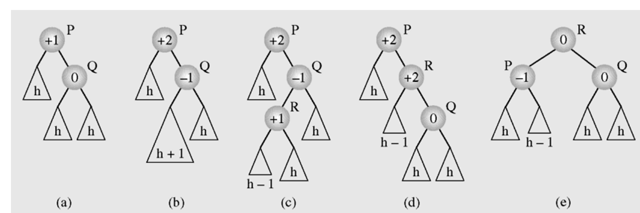


Figure 6-42 Balancing a tree after insertion of a node in the left subtree of node *Q*

AVL Trees (continued)

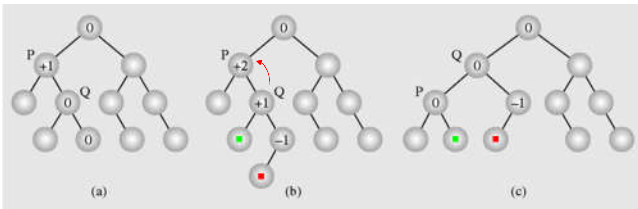


Figure 6-43 An example of inserting a new node (b) in an AVL tree (a), which requires one rotation (c) to restore the height balance

AVL Trees (continued)

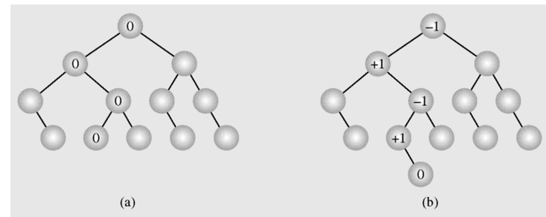


Figure 6-44 In an AVL tree (a) a new node is inserted (b) requiring no height adjustments

AVL Trees (continued)

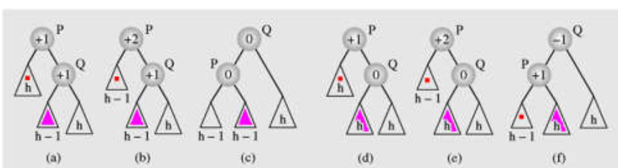


Figure 6-45 Rebalancing an AVL tree after deleting a node

AVL Trees (continued)

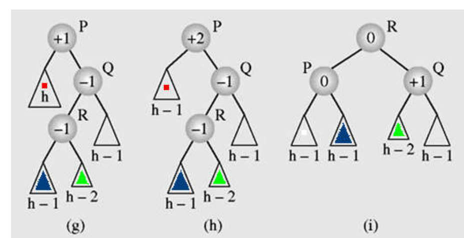


Figure 6-45 Rebalancing an AVL tree after deleting a node (continued)

AVL Trees (continued)

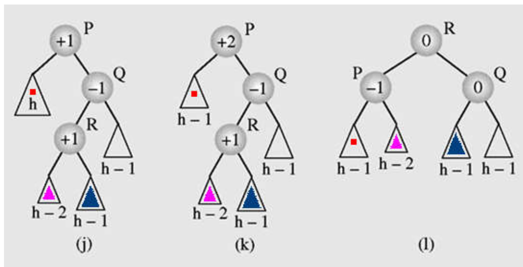


Figure 6-45 Rebalancing an AVL tree after deleting a node (continued)

Self-Adjusting Trees

- The strategy in self-adjusting trees is to restructure trees by moving up the tree with only those elements that are used more often, and creating a **priority tree → These elements will be found faster**

Self-Restructuring Trees

- **Single rotation** – Rotate a child about its parent if an element in a child is accessed, unless it is the root

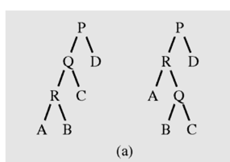


Figure 6-46 Restructuring a tree by using (a) a single rotation or (b) moving to the root when accessing node R

Self-Restructuring Trees (continued)

- **Moving to the root –** Repeat the child–parent rotation until the element being accessed is in the root

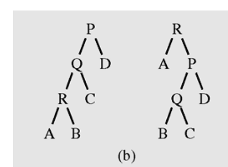


Figure 6-46 Restructuring a tree by using (a) a single rotation or (b) moving to the root when accessing node R (continued)

Self-Restructuring Trees (continued)

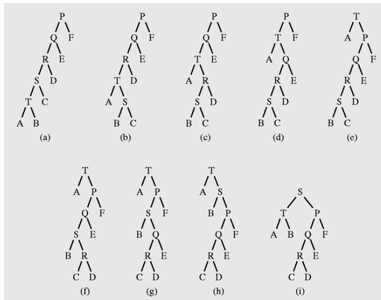


Figure 6-47 (a–e) Moving element *T* to the root and then (e–i) moving element *S* to the root

Splaying

- A modification of the move-to-the-root strategy is called **splaying**
- Splaying applies single rotations in pairs in an order depending on the links between the child, parent, and grandparent
- **Semisplaying** requires only one rotation for a homogeneous splay and continues splaying with the parent of the accessed node, not with the node itself

Splaying (continued)

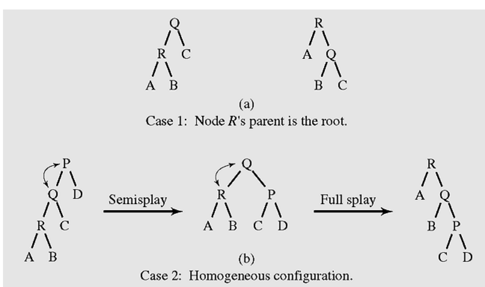


Figure 6-48 Examples of splaying (The node R is accessed)

Splaying (continued)

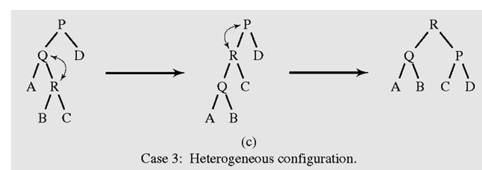


Figure 6-48 Examples of splaying (continued) – The node R is accessed

Splaying (continued)

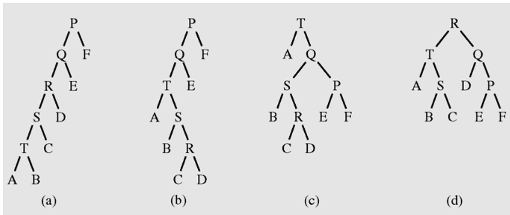


Figure 6-49 Restructuring a tree with splaying (a–c) after accessing *T* and (c–d) then *R*

Splaying (continued)

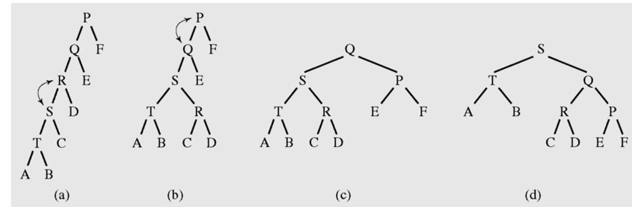


Figure 6-50 (a–c) Accessing *T* and restructuring the tree with semisplaying; (c–d) accessing *T* again

Heaps

- A particular kind of binary tree, called a **heap**, has two properties:
  - The value of each node is greater (max heap)/less (min heap) than or equal to the values stored in each of its children
  - The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions
- These two properties define a **max heap**
- If "greater" in the first property is replaced with "less," then the definition specifies a **min heap**

Heaps (continued)

With the view of Max heaps

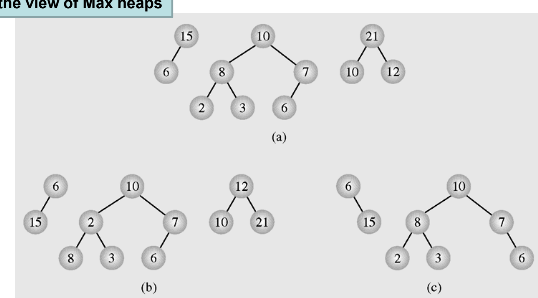


Figure 6-51 Examples of (a) heaps and (b–c) nonheaps

Heaps (continued)

Heap can be implemented by an array, this array is partitioned to groups of same-level elements.

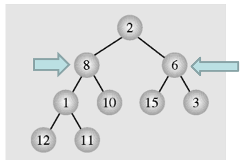


Figure 6-52 The array [2 8 6 1 10 15 3 12 11] seen as a tree
This array It is not a heap because the first property is violated.

Heaps (continued)

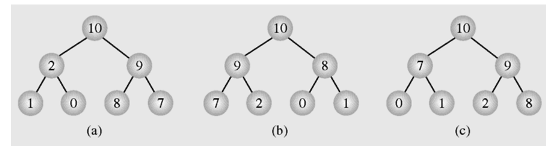


Figure 6-53 Different heaps constructed with the same elements

Heaps as Priority Queues

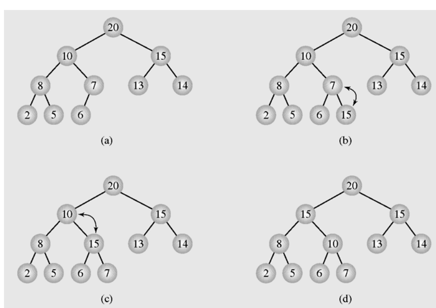


Figure 6-54 Enqueuing an element to a heap

Heaps as Priority Queues (continued)

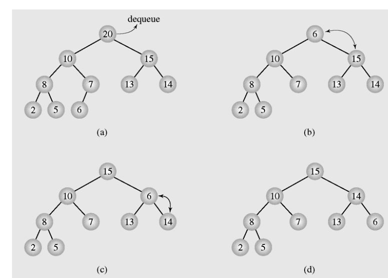


Figure 6-55 Dequeuing an element from a heap

Heaps as Priority Queues (continued)

```
void moveDown(Object[] data, int first, int last) {
    int largest = 2*first + 1;
    while (largest <= last) {
        if (largest < last && // first has two children (at 2*first+1 and
                                                      //    2*first+2)
            ((Comparable)data[largest]).compareTo(data[largest+1]) < 0)
                largest++;
        if (((Comparable)data[first]).compareTo(data[largest]) < 0) {
             swap(data,first,largest);       // if necessary, swap values
             first = largest;               // and move down;
             largest = 2*first + 1;
        }
        else largest = last + 1;// to exit the loop: the heap property
    }                           // isn't violated by data[first]
}
```

Figure 6-56 Implementation of an algorithm to move the root element down a tree

Organizing Arrays as Heaps

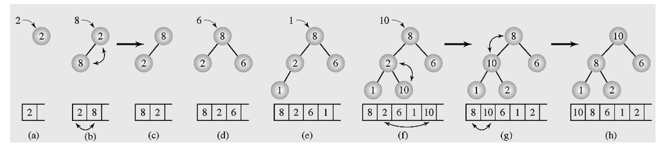


Figure 6-57 Organizing an array as a heap with a top-down method

Organizing Arrays as Heaps

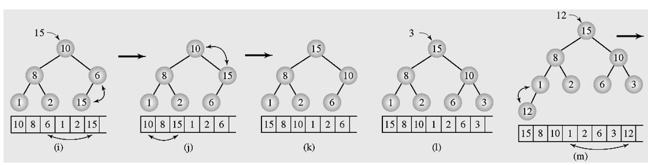


Figure 6-57 Organizing an array as a heap with a top-down method (continued)

Organizing Arrays as Heaps

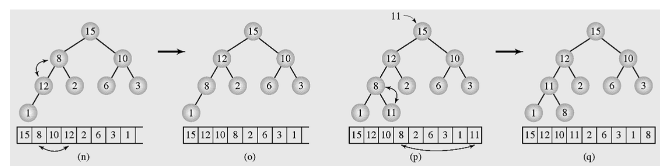


Figure 6-57 Organizing an array as a heap with a top-down method (continued)

Organizing Arrays as Heaps (continued)

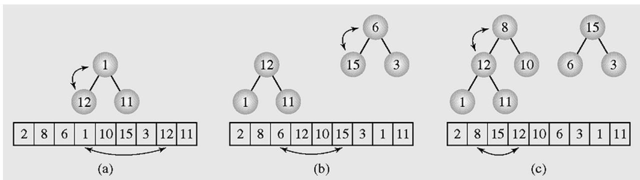


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method

Organizing Arrays as Heaps (continued)

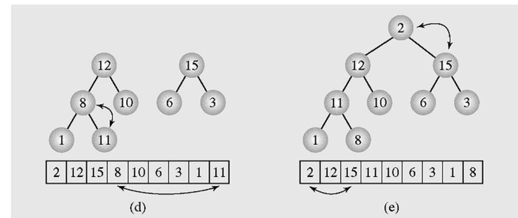


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method (continued)

Organizing Arrays as Heaps (continued)

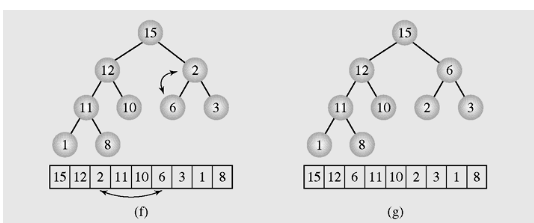


Figure 6-58 Transforming the array [2 8 6 1 10 15 3 12 11] into a heap with a bottom-up method (continued)

Polish Notation and Expression Trees

- **Polish notation (postfix notation)** is a special notation for propositional logic that eliminates all parentheses from formulas
- The compiler rejects everything that is not essential to retrieve the proper meaning of formulas rejecting it as “syntactic sugar”

Polish Notation and Expression Trees (continued)

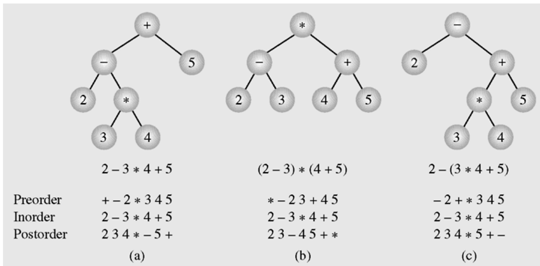
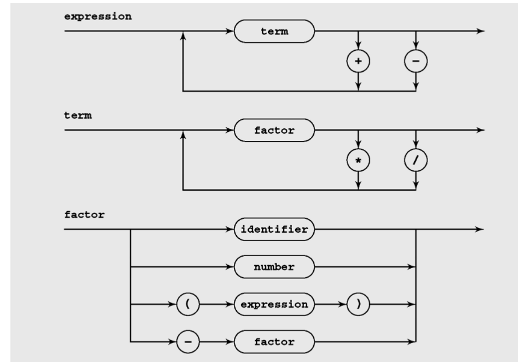


Figure 6-59 Examples of three expression trees and results of their traversals

Diagram



Operations on Expression Trees

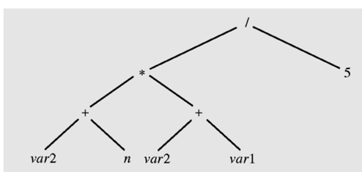


Figure 6-60 An expression tree

Operations on Expression Trees (continued)

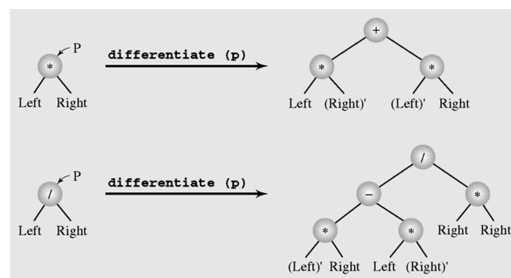


Figure 6-61 Tree transformations for differentiation of multiplication and division

Case Study: Computing Word Frequencies

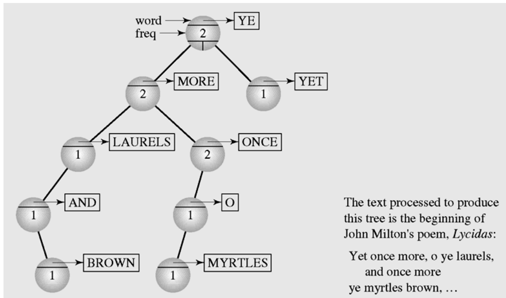


Figure 6-62 Semisplay tree used for computing word frequencies

Case Study: Computing Word Frequencies (continued)

```
/************************  BSTNode.java  *************************
 *           node of a generic binary search tree
 */

public class BSTNode {
    protected Comparable el;
    protected BSTNode left, right;
    public BSTNode() {
        left = right = null;
    }
    public BSTNode(Comparable el) {
        this(el,null,null);
    }
    public BSTNode(Comparable el, BSTNode lt, BSTNode rt) {
        this.el = el; left = lt; right = rt;
    }
}
```

Figure 6-63 Implementation of word frequency computation

Case Study: Computing Word Frequencies (continued)

```
/************************  BST.java  **************************
 *                generic binary search tree
 */

public class BST {
    protected BSTNode root = null;
    public BST() {
    }
    public Comparable search(Comparable el) {
        return search(root,el);
    }
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
    protected Comparable search(BSTNode p, Comparable el) {
        while (p != null)
            if (el.equals(p.el))
                return p.el;
            else if (el.compareTo(p.el) < 0)
                p = p.left;
            else p = p.right;
        return null;
    }
    public void insert(Comparable el) {
        BSTNode p = root, prev = null;
        while (p != null) {  // find a place for inserting new node;
            prev = p;
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
            if (p.el.compareTo(el) < 0)
                 p = p.right;
            else p = p.left;
        }
        if (root == null)    // tree is empty;
             root = new BSTNode(el);
        else if (prev.el.compareTo(el) < 0)
             prev.right = new BSTNode(el);
        else prev.left  = new BSTNode(el);
    }
    protected void visit(BSTNode p) {
        System.out.print(p.el + " ");
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
    }
    public void inorder() {
        inorder(root);
    }
    protected void inorder(BSTNode p) {
        if (p != null) {
             inorder(p.left);
             visit(p);
             inorder(p.right);
        }
    }
    ........................................
}
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
/*************************  SplayTreeNode.java  *********************
 *                   node for generic splaying tree class
 */

public class SplayTreeNode extends BSTNode {
    protected BSTNode parent;
    public SplayTreeNode() {
        left = right = parent = null;
    }
    public SplayTreeNode(Comparable el) {
        this(el,null,null,null);
    }
    public SplayTreeNode(Comparable ob, SplayTreeNode lt,
                         SplayTreeNode rt, SplayTreeNode pr) {
        el = ob; left = lt; right = rt; parent = pr;
    }
}
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
/***************************  SplayTree.java  ************************
 *                     generic splaying tree class
 */

public class SplayTree extends BST {
    public SplayTree() {
        super();
    }
    private void continueRotation(BSTNode gr, BSTNode par,
                                  BSTNode ch, BSTNode desc) {
        if (gr != null) {                   // if par has a grandparent;
             if (gr.right == ((SplayTreeNode)ch).parent)
                  gr.right = ch;
             else gr.left  = ch;
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
        }
        else root = ch;
        if (desc != null)
            ((SplayTreeNode)desc).parent = par;
        ((SplayTreeNode)par).parent = ch;
        ((SplayTreeNode)ch).parent = gr;
    }
    private void rotateR(SplayTreeNode p) {
        p.parent.left = p.right;
        p.right = p.parent;
        continueRotation(((SplayTreeNode)p.parent).parent,
                  p.right,p,p.right.left);
    }
    private void rotateL(SplayTreeNode p) {
        p.parent.right = p.left;
        p.left = p.parent;
        continueRotation(((SplayTreeNode)p.parent).parent,
                  p.left,p,p.left.right);
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
 }
 private void semisplay(SplayTreeNode p) {
     while (p != root) {
         if (((SplayTreeNode)p.parent).parent == null) // if p's
         parent is
              if (p.parent.left == p)              // the root;
                   rotateR(p);
              else rotateL(p);
         else if (p.parent.left == p)     // if p is a left child;
              if (((SplayTreeNode)p.parent).parent.left == p.parent) {
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
                   rotateR((SplayTreeNode)p.parent);
                   p = (SplayTreeNode)p.parent;
              }
              else {
                   rotateR((SplayTreeNode)p); // rotate p and its
                   parent;
                   rotateL((SplayTreeNode)p); // rotate p and its new
                   parent;
              }
         else                           // if p is a right child;
              if (((SplayTreeNode)p.parent).parent.right == p.parent) {
                   rotateL((SplayTreeNode)p.parent);
                   p = (SplayTreeNode)p.parent;
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
              }
              else {
                   rotateL(p);           // rotate p and its parent;
                   rotateR(p);           // rotate p and its new

              }                          // parent;
              if (root == null)          // update the root;
                   root = p;
         }
     }
 }
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
/**********************  WordSplaying.java  **********************/

import java.io.*;

class Word implements Comparable {
    private String word = "";
    public int freq = 1;
    public Word() {
    }
    public Word(String s) {
        word = s;
    }
    public boolean equals(Object el) {
        return word.equals(((Word)el).word);
    }
    public int compareTo(Object el) {
        return word.compareTo(((Word)el).word);
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
    }
    public String toString() {
        return word + ": " + freq + " ";
    }
}

class WordSplay extends SplayTree {
    private int differentWords, // counter of different words in text
                                // file;
                wordCnt;        // counter of all words in the same file;
    public WordSplay() {
        differentWords = wordCnt = 0;
    }
    protected void visit(BSTNode p) {
        differentWords++;
        wordCnt += ((Word)p.el).freq;
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
}
public void run(InputStream fIn, String fileName) {
    int ch = 1;
    Word p;
    try {
        while (ch > -1) {
            while (true)
                if (ch > -1 && !Character.isLetter((char)ch)) // skip
                     ch = fIn.read();                  // nonletters;
                else break;
            if (ch == -1)
                break;
            String s = "";
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
            while (ch > -1 && Character.isLetter((char)ch)) {
                s += Character.toUpperCase((char)ch);
                ch = fIn.read();
            }
            if ((p = (Word)search(new Word(s))) == null)
                 insert(new Word(s));
            else ((Word)p).freq++;
        }
    } catch (IOException io) {
        System.err.println("A problem with input");
    }
    inorder();
    System.out.println("\n\nFile " + fileName
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
                + " contains " + wordCnt + " words among which "
                + differentWords + " are different\n");
    }
}

class WordSplaying {
    static public void main(String args[]) {
        String fileName = "";
        InputStream fIn;
        BufferedReader buffer = new BufferedReader(
                                new InputStreamReader(System.in));
        try {
            if (args.length == 0) {
                 System.out.print("Enter a file name: ");
                 fileName = buffer.readLine();
                 fIn = new FileInputStream(fileName);
```

Figure 6-63 Implementation of word frequency computation (continued)

Case Study: Computing Word Frequencies (continued)

```
            }
            else {
                 fIn = new FileInputStream(args[0]);
                 fileName = args[0];
            }
            (new WordSplay()).run(fIn,fileName);
            fIn.close();
        } catch(IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

Figure 6-63 Implementation of word frequency computation (continued)

Summary

- A tree is a data type that consists of nodes and arcs.
- The root is a node that has no parent; it can have only child nodes.
- Each node has to be reachable from the root through a unique sequence of arcs, called a path.
- An orderly tree is where all elements are stored according to some predetermined criterion of ordering.

Summary (continued)

- A binary tree is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child.
- A decision tree is a binary tree in which all nodes have either zero or two nonempty children.
- Tree traversal is the process of visiting each node in the tree exactly one time.
- Threads are references to the predecessor and successor of the node according to an inorder traversal.

Summary (continued)

- An AVL tree is one in which the height of the left and right subtrees of every node differ by at most one.
- A modification of the move-to-the-root strategy is called splaying.
- Polish notation is a special notation for propositional logic that eliminates all parentheses from formulas.