Trees

- **Definition (recursive):** finite set of elements (nodes) which is either empty or it is partitioned into \( m + 1 \) disjoint subsets \( T_0, T_1, \ldots, T_m \) where \( T_0 \) contains one node (root) and \( T_1, T_2, \ldots, T_m \) are trees.

\[
\begin{align*}
T: & \\
 & T_0 \\
\downarrow & \\
T_1 & T_2 & \ldots & T_m
\end{align*}
\]
Definitions

- **Depth of tree node**: length of the path from the root to the node
  - the root has depth 0
- **Height of tree**: depth of the deepest node
- **Leaf**: all leaves are at level $d$
- **Full tree**: all levels are full
- **Complete tree**: all levels except perhaps level $d$ are full
root (tree) = A
descendents (B) = {F,G} ≡ sons (B)
ancestors (H) = {E} ≡ parent (H)
degree (tree) = 4: max number of sons
degree (E) = 3
level (H) = 2  level (A) = 0
depth (tree) = 3 ≡ max level
Binary Trees

Finite set of nodes which is either empty or it is partitioned into sets $T_0, T_{\text{left}}, T_{\text{right}}$ where $T_0$ is the root and $T_{\text{left}}, T_{\text{right}}$ are binary trees.

A binary tree with $n$ nodes at level $l$ has at most $2n$ nodes at level $l+1$.
Full Binary Tree

- **d**: depth
- A binary tree has exactly $2^l$ nodes at level $l \leq d$
- Total number of nodes: $N = 2^{d+1} - 1$
- $N = 2^0 + 2^1 + \ldots + 2^d = \sum_{j=0}^{d} 2^j = 2^{d+1} - 1$

![Binary Tree Diagram]

all leaf nodes at the same level
Binary Tree Traversal

- **Traversal**: list all nodes in order
  - a) **preorder** (depth first order):
    - root
    - traverse $T_{\text{left}}$ preorder
    - traverse $T_{\text{right}}$ preorder
  - b) **inorder** (symmetric order):
    - traverse $T_{\text{left}}$ inorder
    - root
    - traverse $T_{\text{right}}$ inorder
  - c) **postorder**:
    - traverse $T_{\text{left}}$
    - traverse $T_{\text{right}}$
    - root
preorder:  ABDGCEHIF
inorder:   DGBAHEICF
postorder: GDBHIEFCA
preorder:  ABCEIFJDGHKL
inorder:  EICFJBGDKHLA
postorder: IEJFCGKLHDBA
Traversal of Trees of Degree >= 2

preorder
• $T_0$ (root)
• $T_1$ preorder
• $T_2$ preorder
• ...
• $T_M$ preorder

inorder
• $T_1$ inorder
• $T_0$ (root)
• $T_2$ inorder
• ...
• $T_M$ inorder

postorder
• $T_1$ postorder
• $T_2$ postorder
• ...
• $T_M$ postorder
• $T_0$ (root)
preorder:  ABCDHEFG
inorder:  CBHDEAGF
postorder:  CHDEBGFA
Binary Tree Node ADT

interface BinNode {

    public Object element(); // Return and set the element value
    public Object setElement(Object v);

    public BinNode left(); // Return and set the left child
    public BinNode setLeft(BinNode p);

    public BinNode right(); // Return and set the right child
    public BinNode setRight(BinNode p);

    public boolean isLeaf(); // Return true if this is a leaf node

} // interface BinNode
class BinNodePtr implements BinNode {

private Object element; // Object for this node
private BinNode left; // Pointer to left child
private BinNode right; // Pointer to right child

public BinNodePtr() {left = right = null;} // Constructor 1
public BinNodePtr(Object val) { // Constructor 2
    left = right = null;
    element = val;
}

public BinNodePtr(Object val, BinNode l, BinNode r) // Constructor 3
{ left = l; right = r; element = val; }
}
Binary Tree Node Class (cont.)

// Return and set the element value
public Object element() { return element; }
public Object setElement(Object v) { return element = v; }

// Return and set the left child
public BinNode left() { return left; }
public BinNode setLeft(BinNode p) { return left = p; }

// Return and set the right child
public BinNode right() { return right; }
public BinNode setRight(BinNode p) { return right = p; }

public boolean isLeaf() // Return true if this is a leaf node
{ return (left == null) && (right == null); }

} // class BinNodePtr
void preorder (BinNode tree) {
    if (tree == NULL) return;
    else {
        System.out.print(tree.element() + " ");
        preorder ( tree.left( ) );
        preorder ( tree.right( ) );
    }
}

Binary Tree Traversal
Binary Search Tree (BST)

- Each node stores key $K$
- The nodes of $T_{\text{left}}$ have keys $< K$
- The nodes of $T_{\text{right}}$ have keys $\geq K$
Search in BST

1. Compare with root
2. If \( x = \text{root}(\text{key}) \) => key found !!
3. If \( x < \text{key(root)} \) search the \( T_{\text{left}} \) recursively
4. If \( x \geq \text{key(root)} \) search \( T_{\text{right}} \) recursively
Find Min/Max Key

- Find the maximum key in a BST: follow nodes on \( T_{\text{right}} \) until key is found or NULL
- Find the minimum key in a BST: follow nodes on \( T_{\text{left}} \)
Insert Key in BST

- Search until a leaf is found
  - insert it as left child of leaf if key(leaf) < x
  - insert it as right child of leaf if key(leaf) >= x
Shape of BST

- Depends on insertion order
- For random insertion sequence the BST is more or less balanced
  - e.g., 10 20 5 7 1
- For order insertion sequence the BST becomes list
  - e.g., 1 5 7 10 20
Delete key x from BST

- Three cases:
  a) X is a leaf: simply delete it
  b) X is not a leaf & it has exactly one sub-tree
     - the father node points to the node next to x
     - delete node(x)
  c) X is not a leaf & it has two sub-trees
     - find p: minimum of $T_{right}$ or maximum of $T_{left}$
     - this has at most one sub-tree!!
     - delete it as in (a) or (b)
     - substitute x with p
a) The key is a Leaf

(a) Deleting node with key 15.
b) The key has one Sub-Tree

(b) Deleting node with key 5.

E.G.M. Petrakis
Trees
c) The key has Two Sub-Trees

(c) Deleting node with key 11.
BST Sorting

- Insert keys in BST
- Insertion sequence: 50 30 60 20 35 40 37 65
- Output keys inorder
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$
Implementation of BSTs

- Each node stores the key and pointers to roots of $T_{\text{left}}$, $T_{\text{right}}$
- **Array-based implementation:**
  - known maximum number of nodes
  - fast
  - good for heaps
- **Dynamic memory allocation:**
  - no restriction on the number of nodes
  - slower but general
Dynamic Memory Allocation

- Two C++ classes
  - BinNode: node class (page 12)
    - node operations: left, right, setValue, isleaf, father ...
  - BST: tree class
    - composite operations: find, insert, remove, traverse ...
- Use of “help” operations:
  - easier interface to class
  - encapsulation: implementation becomes private
Elem Interface

Definition of an Object with support for a key field:

```java
interface Elem {  // Interface for generic
    // element type
    public abstract int key(); // Key used for search
    // and ordering
}
```

// interface Elem
BST Class

class BST { // Binary Search Tree implementation

    private BinNode root; // The root of the tree

    public BST() { root = null; } // Initialize root to null
    public void clear() { root = null; }
    public void insert(Elem val)
    { root = inserthelp(root, val); }
    public void remove(int key)
    { root = removehelp(root, key); }
    public Elem find(int key)
    { return findhelp(root, key); }
    public boolean isEmpty() { return root == null; }
}
public void print() { // Print out the BST
    if (root == null)
        System.out.println("The BST is empty.");
    else {
        printhelp(root, 0);
        System.out.println();
    }
}
BST Class (cont.)

```java
private Elem findhelp(BinNode rt, int key) {
    if (rt == null)
        return null;
    Elem it = (Elem)rt.element();
    if (it.key() > key)
        return findhelp(rt.left(), key);
    else if (it.key() == key)
        return it;
    else
        return findhelp(rt.right(), key);
}
```
private BinNode inserthelp(BinNode rt, Elem val) {
    if (rt == null)
        return new BinNodePtr(val);
    Elem it = (Elem)rt.element();
    if (it.key() > val.key())
        rt.setLeft(inserthelp(rt.left(), val));
    else
        rt.setRight(inserthelp(rt.right(), val));
    return rt;
}
BST Class (cont.)

private BinNode deletemin(BinNode rt) {
    if (rt.left() == null)
        return rt.right();
    else {
        rt.setLeft(deletemin(rt.left()));
        return rt;
    }
}

// Binary Search Tree implementation
Array Implementation

- General for trees with degree $d \geq 2$
- Many alternatives:
  1. array with pointers to father nodes
  2. array with pointers to children nodes
  3. array with lists of children nodes
  4. binary tree array (good for heaps)
a) Pointers to Father Nodes

- Easy to find ancestors
- Difficult to find children
- Minimum space
b) Pointers to Children Nodes

- Easy to find children nodes
- Difficult to find ancestor nodes
c) Lists of Children Nodes

- Easy to find children nodes
- Difficult to find ancestor nodes
d) Binary Tree Array

- **parent (r) = (r-1)/2** if 0 < r < n
- **leftchild (r) = 2r+1** if 2r + 1 < n
- **rightchild (r) = 2r+2** if 2r + 2 < n
- **leftsibling (r) = r-1** if r even & 0 < r < n
- **rightsibling (r) = r+1** if r odd & 0 < r+1 < n
Heap

- Complete binary tree (not BST)
- Max heap: every node has a value $\geq$ children
- Min heap: every node has a value $\leq$ children
Heap Operations

- **insert** new element in heap
- **build** heap from \( n \) elements
- **delete** max (min) element from heap
- **shift-down**: move an element from the root to a leaf
- Array implementation of page 36
- In the following assume **max** heap
Insertion

1. Insert last in the array (as leaf) and
2. shift it to its proper position: swap it with parent elements as long as it has greater value

- Complexity $O(\log n)$: at most $d$ shift operations in array ($d$: depth of tree)
Build Head

- Insert $n$ elements in an empty heap
  - complexity: $O(n \log n)$
- Better bottom-up method:
  - Assume $H_1, H_2$ heaps and shift $R$ down to its proper position (depending on which root from $H_1, H_2$ has greater value)
  - The same (recursively) within $H_1, H_2$
Example

- Starts at $d-1$ level: the leafs need not be accessed
- Shift down elements
Complexity

- Up to $n/2^1$ elements at $d$ level $\Rightarrow$ 0 moves/elem
- Up to $n/2^2$ elements at $d-1$ level $\Rightarrow$ 1 moves/elem
  ...
- Up to $n/2^i$ elements at level $i$ $\Rightarrow$ $i-1$ moves/elem

\[
\text{total number of moves } \sum_{i=1}^{d=\log n} (i - 1)n/2^i \in \Theta(n)
\]
Deletion

- The max element is removed
  - swap with last element in array
  - delete last element
  - shift-down root to its proper position
- Complexity: $O(\log n)$
public class Heap { // Heap class
private Elem[] Heap;   // Pointer to the heap array
private int size;      // Maximum size of the heap
private int n;         // Number of elements now in the heap

public Heap(Elem[] h, int num, int max)  // Constructor
{ Heap = h;  n = num;  size = max;  buildheap(); }

public int heapsize() // Return current size of the heap
{ return n; }

public boolean isLeaf(int pos) // TRUE if pos is a leaf position
{ return (pos >= n/2) && (pos < n); }
Class Heap (cont.)

// Return position for left child of pos
public int leftchild(int pos) {
    Assert.notFalse(pos < n/2, "Position has no left child");
    return 2*pos + 1;
}

// Return position for right child of pos
public int rightchild(int pos) {
    Assert.notFalse(pos < (n-1)/2, "Position has no right child");
    return 2*pos + 2;
}
public int parent(int pos) { // Return position for parent
    Assert.notFalse(pos > 0, "Position has no parent");
    return (pos-1)/2;
}

public void buildheap() { // Heapify contents of Heap
    for (int i=n/2-1; i>=0; i--) siftdown(i);
}
// Put element in its correct place
private void siftdown(int pos) {
    Assert.notFalse((pos >= 0) && (pos < n), "Illegal heap position");
    while (!isLeaf(pos)) {
        int j = leftchild(pos);
        if ((j<(n-1)) && (Heap[j].key() > Heap[j+1].key())) {
            j++; // j is now index of child with greater value
        } else { // j is leaf
            if (Heap[pos].key() > Heap[j].key()) return; // Done
            DSutil.swap(Heap, pos, j);
            pos = j; // Move down
        }
    }
}
Class Heap (cont.)

```java
public void insert(Elem val) { // Insert value into heap
    Assert.notFalse(n < size, "Heap is full");
    int curr = n++;
    Heap[curr] = val; // Start at end of heap
    // Now sift up until curr's parent's > curr's key
    while ((curr!=0) && (Heap[curr].key()<Heap[parent(curr)].key()))
    {
        DSutil.swap(Heap, curr, parent(curr));
        curr = parent(curr);
    }
}
```
public Elem removemin() { // Remove minimum value
    Assert.notFalse(n > 0, "Removing from empty heap");
    DSutil.swap(Heap, 0, --n); // Swap minimum with last value
    if (n != 0) { // Not on last element
        siftdown(0); // Put new heap root val in correct place
    }
    return Heap[n];
}
Class Heap (cont.)

// Remove value at specified position
public Elem remove(int pos) {
    Assert.notFalse((pos > 0) && (pos < n), "Illegal heap position");
    DSutil.swap(Heap, pos, --n); // Swap with last value
    if (n != 0) // Not on last element
        siftdown(pos); // Put new heap root val in correct place
    return Heap[n];
}

} // Heap class
Heap Sort

- Insert all elements in new heap array
- At step $i$ delete $\text{max}$ element
  - $\text{delete}$: put max element last in array
- Store this element at position $i+1$
- After $n$ steps the array is sorted!!
- Complexity: $O(n\log n)$ why??
(a) Initial heap

(b) $x[7] = \text{delete}(92)$

(c) $x[6] = \text{delete}(86)$
(d) $x[5] = \text{delete}(57)$

(e) $x[4] = \text{delete}(48)$
(f) \( x[2] = \text{delete}(33) \)

(g) \( x[1] = \text{delete}(25) \)

(\( \zeta \)) \( x[3] = \) delete(37)
Huffman Coding

- **Goal**: improvement in space requirements in exchange of a penalty in running time
- Assigns codes to symbols
  - **code**: binary number
- **Main idea**: the length of a code depends on its frequency
  - shorter codes to more frequent symbols
Huffman Coding (cont.)

- **Input**: sequence of symbols (letters)
  - typically 8 bits/symbol
- **Output**: a binary representation
  - the codes of all symbols are concatenated in the same order they appear in the input
  - a code cannot be prefix of another
  - \( \bar{n} = \frac{\sum_{i=1}^{N} n_i p_i}{\sum_{i=1}^{N} n_i} \) less than 8 bits/symbol on the average
Huffman Coding (cont.)

- The code of each symbol is derived from a binary tree
  - each leaf corresponds to a letter
  - goal: build a tree with the minimum external path weight (epw)
  - epw: sum. of weighted path lengths
  - min epw $\iff$ min $n$
  - a letter with high weight (frequency) should have low depth $\Rightarrow$ short code
  - a letter with low weight may be pushed deeper in the Huffman tree $\Rightarrow$ longer code
Building the Huffman Tree

- The tree is built bottom-up
- Order the letters by ascending frequency
- The first two letters become leaves of the Huffman tree
- Substitute the two letters with one with weight equal to the sum of their weights
- Put this new element back on the ordered list (in its correct place)
- Repeat until only one element (root of Huffman tree) remains in the list
Assigning Codes to Letters

- Beginning at the root assign 0 to left branches and 1 to right branches.
- The Huffman code of a letter is the binary number determined by the path from the root to the leaf of that letter.
- Longer paths correspond to less frequent letters (and the reverse).
- Replace each letter in the input with its code.
Huffman Example

0
Weights: .12 .40 .15 .08 .25
Symbols: a b c d e

1
Weights: .20 .40 .15 .25
Symbols: b c e

2
Weights: .35 .40 .25
Symbols: c b e

3
Weights: .60 .40
Symbols: a b
e
c
Huffman Tree

4
Weights: 1.0
Symbols: a b
e c
d
Huffman Tree

E.G.M. Petrakis
Trees
Huffman Encoding

Huffman Tree

Huffman codes
a: 1111
b: 0
c: 110
d: 1110
e: 10

input: acde => Huffman code: 1111 110 1110 10
Huffman Decoding

Repeat until end of code
- beginning at the root, take right branch for each 1 and left branch for each 0 until we reach letter
- output the letter
Critique

- **Advantages:**
  - compression (40-80%)
  - cheaper transmission
  - some security (encryption)
  - locally optimal code

- **Disadvantages:**
  - CPU cost
  - difficult error correction
  - we need the Huffman tree for the decoding
  - code not globally optimal