Data Structure Building Blocks

These are implementation “building blocks” that are often used to build more-complicated data structures:

- Arrays
- Linked Lists
  - Singly linked
  - Doubly linked
- Binary Trees
- Graphs
  - Adjacency matrix
  - Adjacency list

Arrays

- Declaration/Initialization
  ```java
  String[ ] s = new String[3];
  s[0] = "jan";
  s[1] = "feb";
  s[2] = "mar";
  or
  String[ ] s = new String[3] {"jan","feb","mar"};
  ```

- Iteration
  ```java
  for (int i = 0; i < s.length; i++) {
    // Do something using s[i]
  }
  ```

- Advantages
  - Fast access to each element
  - O(1) time
  - Space efficient

- Disadvantages
  - Hard to insert an element in the middle
  - Size must be known when created

Singly-Linked List

- Declaration
  ```java
  static class Node {
    Object data;
    Node next;
    Node(Node p, Object d, Node n) {
      prev = p;
      data = d;
      next = n;
    }
  }
  ```

- Initialization
  ```java
  Node head = new Node(null, "jan", null);
  head.next = new Node(head, "feb", null);
  head.next.next = new Node(head.next, "mar", null);
  ```

- Iteration
  ```java
  Node node = head;
  while (node != null) {
    // Use node somehow
    node = node.next;
  }
  ```

- Advantages
  - Grows as needed
  - Efficient insertion

- Disadvantages
  - Element access can be expensive
  - Generally, O(n)
  - Uses extra space for pointers
  - Can go forward, but not backward

Doubly-Linked List

- Declaration
  ```java
  static class Node {
    Object data;
    Node next, prev;
    Node(Node p, Object d, Node n) {
      prev = p;
      data = d;
      next = n;
    }
  }
  ```

- Initialization
  ```java
  Node head = new Node(null, "jan", null);
  head.next = new Node(head, "feb", null);
  head.next.next = new Node(head.next, "mar", null);
  ```

- Iteration
  ```java
  Node node = head;
  while (node != null) {
    // Use node somehow
    node = node.next;
  }
  ```

- Advantages
  - Grows as needed
  - Efficient insertion
  - Can move both forward and backward

- Disadvantages
  - Element access can be expensive
  - Uses even more extra space for pointers

What do we mean by “List”?

- Intuitive idea of a list
  - Used when speaking informally
  - Examples: grocery list, cs211 class list, list of possible running mates

- ADT List
  - Includes operations that (should) correspond to our intuitive idea of a list
  - There is only partial agreement on what those operations should be
  - Java includes a List interface (java.util.List) as part of the Java Collections Framework
Terminology for (Rooted) Trees

- Each tree has a distinguished root; there is a unique path from the root to each node (i.e., no loops)
- Each node, except the root, has one parent
- A node can have multiple children
- A node with no children is called a leaf
- The height of a tree is the length of its longest root-to-leaf path
- Ancestor and descendant are based on analogy to family trees

Binary Trees

- Declaration
  ```java
  class Node {
      Object data;
      Node lchild, rchild;
      Node (Node lc, Object d, Node rc) { data = d; lchild = lc; rchild = rc; }
  }
  ```
- Initialization
  ```java
  Node root = new Node(null, "jan", null);
  root.lchild = new Node(null, "feb", null);
  root.rchild = new Node(null, "mar", null);
  ```
- Advantages
  - Grows as needed
  - Efficient access to elements ▲ generally, O(log n)
  - Requires balanced tree ▲ Efficient insertion
- Disadvantages
  - Uses extra space for pointers

Graph Terminology

- A graph G is a pair (V,E) where V is a set of vertices and E is a set of edges
- Each edge is a pair (u,v) of vertices
- In a directed graph (or digraph), edge pairs are ordered (i.e., (u,v) is considered to be a different edge than (v,u))
- In an undirected graph, the edge pairs are unordered
- A path is a sequence of vertices v0, …, vn such that (v_i,v_i+1) is an edge for each i
- A cycle is a path (of length at least one) for which v_0 = v_n
- A weighted graph has a cost for each edge

Implementing Digraphs

- Adjacency Matrix
  ```java
  g[u][v] is true iff there is an edge from u to v
  ```
- Adjacency List
  ```java
  The list for u contains v iff there is an edge from u to v
  ```

Implementing Weighted Digraphs

- Adjacency Matrix
  ```java
  g[u][v] is c iff there is an edge of cost c from u to v
  ```
- Adjacency List
  ```java
  The list for u contains v,c iff there is an edge from u to v that has cost c
  ```
Implementing Undirected Graphs

- Adjacency Matrix
  - $g(u, v)$ is true if there is an edge from $u$ to $v$
  - $G$ uses space $O(v^2)$
  - Can iterate over all edges in time $O(v^2)$
  - Can answer "Is there an edge from $u$ to $v$?" in $O(1)$ time
  - Better for dense (i.e., lots of edges) graphs

- Adjacency List
  - The list for $u$ contains $v$ if there is an edge from $u$ to $v$
  - $G$ uses space $O(e + v)$
  - Can iterate over all edges in time $O(e + v)$
  - Can answer "Is there an edge from $u$ to $v$?" in $O(e_u)$ time
  - Better for sparse (i.e., fewer edges) graphs

Adjacency Matrix or Adjacency List?

- $v$ = number of vertices
- $e$ = number of edges
- $e_u$ = number of edges leaving $u$