Data Structure Building Blocks

CS211
Fall 2000

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

2

Arrays

- Declaration/Initialization

```
String[ ] s = new String[3];
s[0] = “jan”; s[1] = “feb”; s[2] = “mar”;
or
String[ ] s;
s = new String[ ] {“jan”,”feb”,”mar”};
```

- Iteration

```
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

3

Singly-Linked List

- Declaration

```
static class Node {
        Object data;
        Node next;
        Node (Object d, Node n)
                {data = d; next = n;}
        }
```

- Initialization

```
Node head = new Node(“jan”,null);
head.next = new Node(“feb”,null);
head.next.next = new Node(“mar”,null);
```

- Iteration

```
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

4

Doubly-Linked List

- Declaration

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

- Initialization

```
Node head = new Node(null,“jan”,null);
head.next = new Node(head,“feb”,null);
head.next.next = new
    Node(head.next,“mar”,null);
```

- Iteration
  - Same as singly-linked list
- 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

5

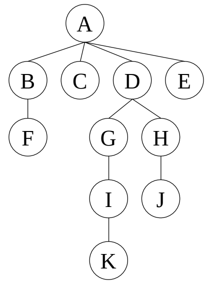
What do we mean by “List”?

- Intuitive idea of a list
  - Used when speaking informally
  - Examples: grocery list, cs211 class list, list of possible running mates
- Implementations of a list
  - Used when speaking of algorithms
  - Examples: array, singly-linked list, doubly linked-list
- 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

6

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 *descendent* are based on analogy to family trees



7

Binary Trees

- Declaration

```
class Node {
    Object data;
    Node lchild,rchild;
    Node (Node lc, Object d, Node rc)
        {data = d; lchild = lc; rchild = rc;}
}
```

- Initialization

```
Node root = new Node(null,"jan",null);
root.lchild = new Node(null,"feb",null);
root.rchild = new Node(null,"mar",null);
```

- Iteration (next slide)

- Advantages
  - Grows as needed
  - Efficient access to elements
    - generally, O(log n)
    - requires *balanced* tree
  - Efficient insertion
- Disadvantages
  - Uses extra space for pointers

8

Binary Tree Iteration: Tree Traversals

- Preorder Traversal

```
static void preorder (Node node) {
    if (node == null) return;
    // Process node
    preorder(node.lchild);
    preorder(node.rchild);
    }
```

- Inorder Traversal

```
static void inorder (Node node) {
    if (node == null) return;
    inorder(node.lchild);
    // Process node
    inorder(node.rchild);
    }
```

- Postorder Traversal

```
static void postorder (Node node) {
        if (node == null) return;
        postorder(node.lchild);
        postorder(node.rchild);
        // Process node
        }
```

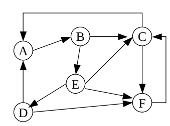
- Level-Order Traversal

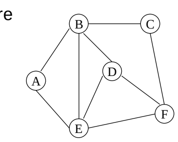
```
static void levelOrder (Node root) {
    Queue q = new Queue( ); q.put(root);
    while (!q.isEmpty( )) {
        Node node = (Node) q.get( );
        // Process node
        if (node.lchild != null) q.put(node.lchild);
        if (node.rchild != null) q.put(node.rchild);
        } }
```

9

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 $v_0,\ldots,v_n$ 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





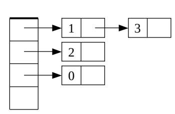
10

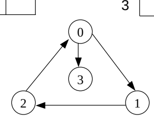
Implementing Digraphs

- Adjacency Matrix
  g[u][v] is true iff there is an edge from u to v

	0	1	2	3
0		T		T
1			T	
2	T			
3				

- Adjacency List
  The list for u contains v iff there is an edge from u to v





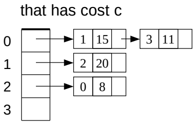
11

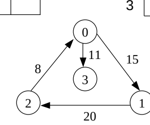
Implementing Weighted Digraphs

- Adjacency Matrix
  g[u][v] is c iff there is an edge of cost c from u to v

	0	1	2	3
0		15		11
1			20	
2	8			
3				

- Adjacency List
  The list for u contains v,c iff there is an edge from u to v that has cost c





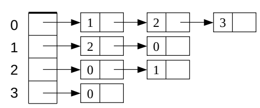
12

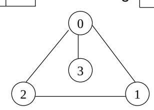
Implementing Undirected Graphs

- **Adjacency Matrix**
  g[u][v] is true iff there is an edge from u to v

	0	1	2	3
0		T	T	T
1	T		T	
2	T	T		
3	T			

- **Adjacency List**
  The list for u contains v iff there is an edge from u to v





Adjacency Matrix or Adjacency List?

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

- Adjacency Matrix
  - 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
  - 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