Chapter 3
Data Structures

Stack in Java

import java.util.*;
public class Stack<E> extends Vector<E> {
  public Stack<E>() {
    public static void main(String[] args) {
      Stack<Integer> st = new Stack<Integer>();
      st.push(44);
      int top = st.pop();
      System.out.println("the last item is " + top);
      if (st.empty())
        System.out.println("the top item is " + st.peek());
      else
        System.out.println("the top item is " + st.peek());
    }
  }
}

Queue in Java

public class Queue<E> extends Collection<E> {
  public Queue<E>() {
    public static void main(String[] args) {
      Queue<Integer> queue = new LinkedList<Integer>();
      for (int i = 10; i >= 0; i--) queue.add(i);
      while (!queue.isEmpty()) {
        System.out.println(queue.remove());
      }
    }
  }
}

All Known Implementing Classes:
AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue,
DelayQueue, LinkedBlockingQueue, LinkedBlockingDeque, LinkedList,
PriorityBlockingQueue, PriorityQueue, SynchronousQueue

Graphs

Graph. G = (V, E)
- V = nodes.
- E = edges between pairs of nodes.
- Captures pairwise relationship between objects:
  - Undirected graph represents symmetric relation
  - Directed graph represents general binary relation
- Graph size parameters: n = |V|, m = |E|.
- Simple: no loops and no multiple edges

Example: Display the digraph with V = (a, b, c, d),
E = ((a, b), (a, d), (b, b), (b, d), (c, a), (c, b), (d, b)).

An edge of the form (b, b) is called a loop.

Relations vs Graph

A relation R on the set A is a subset of A x A.
There is 1-to-1 correspondence between R and (directed) G=(A, R).

Example: Let A = {1, 2, 3, 4}. Which ordered pairs are in the relation R = {(a, b) | a < b}?

R = {(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)}
Relations on a Set

How many different relations can we define on a set $A$ with $n$ elements?

A relation on a set $A$ is a subset of $A \times A$.

How many elements are in $A \times A$?

There are $n^2$ elements in $A \times A$, so how many subsets (relations on $A$) does $A \times A$ have?

The number of subsets that we can form out of a set with $m$ elements is $2^m$. Therefore, $2^{n^2}$ subsets can be formed out of $A \times A$.

Answer: We can define $2^{n^2}$ different relations on $A$. As a result, we have that much graphs on $n$ points.

Properties of Relations

Definition: A relation $R$ on a set $A$ is called reflexive if $(a, a) \in R$ for every element $a \in A$.

The graph that each node has a loop represents a reflexive relation.

Properties of Relations

Definitions:

A relation $R$ on a set $A$ is called symmetric if $(b, a) \in R$ whenever $(a, b) \in R$ for all $a, b \in A$.

A relation $R$ on a set $A$ is called antisymmetric if $a = b$ whenever $(a, b) \in R$ and $(b, a) \in R$.

A relation $R$ on a set $A$ is called asymmetric if $(a, b) \in R$ implies that $(b, a) \not\in R$ for all $a, b \in A$.

Every undirected graph represents a symmetric relation.

What is the relation between "antisymmetric" and "asymmetric"?

$R$ is asymmetric iff $R$ is antisymmetric and has no loops.

Properties of Relations

Definition: A relation $R$ on a set $A$ is called transitive if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for all $a, b, c \in A$.

Whenever there is a path that goes from $a$ to $b$, then there is an edge $(a, b)$ in the graph, then the graph represents a transitive relation.

Are the following relation on $\{1, 2, 3\}$ transitive?

$R = \{(1, 1), (1, 2), (2, 2), (2, 1), (3, 3)\}$

Combining Relations

Definition: Let $R$ be a relation on the set $A$. The powers $R^n$, $n = 1, 2, 3, \ldots$, are defined inductively by

- $R^1 = R$
- $R^{n+1} = R^n \circ R$

In other words: $R^n = R \circ R \circ \ldots \circ R$ (n times the letter $R$)

The relation $R^n = R \cup R^2 \cup R^3 \cup \ldots \cup R^n$, where $n$ is the number of nodes, is called the transitive closure of $R$.

To decide if $(a, b)$ in $R^n$, we need to decide if there is a path from $a$ to $b$ in $G = (A, R)$. 

Combining Relations

Definition: Let $R$ be a relation from a set $A$ to a set $B$ and $S$ a relation from $B$ to a set $C$. The composite of $R$ and $S$ is the relation consisting of ordered pairs $(a, c)$, where $a \in A$, $c \in C$, and for which there exists an element $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$. We denote the composite of $R$ and $S$ by $S \circ R$.

If $A = B = C$, and $S = R$, then $R \circ R$ can be written as $R^2$.

If $R$ is represented by a graph, then $(a, b)$ is in $R^2$ iff there is a path of length 2 from $a$ to $b$. 
Combining Relations

**Theorem:** The relation \( R \) on a set \( A \) is transitive if and only if \( R^n \subseteq R \) for all positive integers \( n \).

Remember the definition of transitivity:

**Definition:** A relation \( R \) on a set \( A \) is called transitive if whenever \((a, b) \in R \) and \((b, c) \in R \), then \((a, c) \in R \) for \( a, b, c \in A \).

The composite of \( R \) with itself contains exactly these pairs \((a, c)\).

Therefore, for a transitive relation \( R \), \( R \circ R \) does not contain any pairs that are not in \( R \), so \( R \circ R \subseteq R \).

Since \( R \circ R \) does not introduce any pairs that are not already in \( R \), it must also be true that \((R \circ R) \circ R \subseteq R \), and so on, so that \( R^n \subseteq R \).

---

Equivalence Relations

**Equivalence relations** are used to relate objects that are similar in some way.

**Definition:** A relation on a set \( A \) is called an equivalence relation if it is reflexive, symmetric, and transitive.

Two elements that are related by an equivalence relation \( R \) are called equivalent.

The best representation of an equivalence relation is Sets.

---

Graph Representation: Adjacency Matrix

- **Adjacency matrix.** \( n \)-by-\( n \) matrix with \( A_{uv} = 1 \) if \((u, v)\) is an edge.
  - Two 1's of each edge for undirected graph.
  - Space proportional to \( n^2 \).
  - Checking if \((u, v)\) is an edge takes \( O(1) \) time.
  - Identifying all edges takes \( O(n^2) \) time.

---

Graph Representation: Adjacency List

- **Adjacency list.** Node indexed array of lists.
  - Two representations of each edge for undirected graphs.
  - Space proportional to \( m + n \).
  - Checking if \((u, v)\) is an edge takes \( O(\text{deg}(u)) \) time.
  - Identifying all edges takes \( O(m + n) \) time.

---

Paths and Connectivity

- **Def.** A path in a graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \) with the property that each consecutive pair \((v_i, v_{i+1})\) is an edge in \( E \).

- **Def.** A path is simple if all nodes are distinct.

- **Def.** An undirected graph is connected if for every pair of nodes \( u \) and \( v \), there is a path from \( u \) to \( v \).

- **Def.** A directed graph is strongly connected if for every pair of nodes \( u \) and \( v \), there is a path from \( u \) to \( v \).

---

Cycles

- **Def.** A cycle is a path \( v_1, v_2, \ldots, v_{k-1}, v_k \) in which \( v_1 = v_k \), \( k > 2 \), and the first \( k-1 \) nodes are all distinct.
**Trees**

**Definition.** An undirected graph is a **tree** if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.
- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.

**Rooted Trees**

**Rooted tree.** A directed graph where its underlying undirected graph is a tree and there is a node $r$ called root and each edge points away from $r$.

**Importance.** Models hierarchical structure.

**Tree Terminology**

- **Node**
  - Internal node
  - Leaf (external node)
- **Branch (link, edge)**
- **Parent/child**
- **Descendant/ancestor**
- **Sibling**
- **Root**

**Tree Terminology (continued)**

- **Degree**
  - Binary tree
- **Level (depth)**
- **Height**
- **Subtree**
- **Ordered/unordered tree**
- **First (least) common ancestor**

**Tree Terminology (continued again)**

- **Full tree** — Every node has zero or as many children as possible
- **Complete tree** — Every level is full, except possibly the last level where every node is pushed to the left
- **Perfect tree** — Full and leaves are on the same level

**Binary Trees Properties**

- **Let:**
  - $M = \# \text{ branches}$
  - $N = \# \text{ nodes}$
  - $L = \# \text{ leaf nodes}$
  - $I = \# \text{ internal nodes}$
  - $H = \text{ tree height}$
  - $M = N - 1$
Binary Trees Properties

• In a perfect binary tree,
  \[N = 2^H - 1, \quad H = \log_2(N + 1) - 1\]
• In a perfect binary tree, \(L = 2^H\)
• If a binary tree has \(L\) leaf nodes and \(N_2\) nodes with 2 children, \(L = N_2 + 1\)
• In other words, there is one more leaf node than nodes with 2 children.

Computing the height of a binary tree

```java
public int height () {
    // Pre: assuming the current node is not null.
    // Post: return the height of the binary tree
    //       rooted by the current node.
    int h1 = h2 = 0;
    if (leftChild != null) h1 = leftChild.height() + 1;
    if (rightChild != null) h2 = rightChild.height() + 1;
    return (h1 < h2)? h2 : h1;
} // height
```

Tree Traversal

• Traversals visit a tree’s nodes in different orders

Preorder

• Visit the node, then recursively visit the children

\[\text{D B A C E}\]

Inorder (Symmetric)

• Recursively visit the left child, then the node, then recursively visit the right child

\[\text{A B C D E}\]

Postorder

• Recursively visit the children, then visit the node

\[\text{A C B E D}\]
Breadth-First

- Visit the nodes on each level of the tree before visiting any nodes on the next level

- DBEAC

Sorted Trees (Search Trees)

- Nodes are arranged so an inorder traversal visits them in sorted order

Binary Search Tree

```java
class BinaryNode {
    KeyType Key;
    BinaryNode LeftChild;
    BinaryNode RightChild;
    BinaryNode parent; // optional
    
    BinaryNode Constructor(KeyType key){
        Key = key; parent = null;
    }
}
```

Binary Search Tree Property

- Binary search tree property:
  - If y is in left subtree of x, then y.Key ≤ x.Key
  - If y is in right subtree of x, then y.Key ≥ x.Key

Traversing a Binary Search Tree

- **Alg**: INORDER-TREE-WALK(x)
  1. if x ≠ NIL
  2. then INORDER-TREE-WALK ( x.LeftChild )
  3. print x.Key 
  4. INORDER-TREE-WALK ( x.RightChild )

- **E.g.**: Output: 2 3 5 5 7 9

- Running time:
  - $\Theta(n)$, where n is the size of the tree rooted at x

Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Root is printed between the values of its left and right subtrees: left, root, right
  - Keys are printed in sorted order
- **Preorder** tree walk:
  - root printed first: root, left, right
- **Postorder** tree walk:
  - root printed last: left, right, root

- Inorder: 2 3 5 5 7 9
  - Preorder: 5 3 2 5 7 9
  - Postorder: 2 5 3 9 7 5
Binary Search Trees

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees: $O(h)$, $h$ is the height of the tree
  - On average: $\Theta(\log_2(n))$
    - The expected height of the tree is $\log_2(n)$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of $n$ nodes

Searching for a Key

- Given a pointer to the root of a tree and a key $k$:
  - Return a node with key $k$ if one exists
  - Otherwise return NIL
- Idea
  - Starting at the root: trace down a path by comparing $k$ with the key of the current node:
    - If the keys are equal: we have found the key
    - If $k < x.Key$ search in the left subtree of $x$
    - If $k > x.Key$ search in the right subtree of $x$

Alg: TREE-SEARCH($x$, $k$)
1. if $x = NIL$ or $k = x.Key$
2. then return $x$
3. if $k < x.Key$
4. then return TREE-SEARCH($x$.LeftChild, $k$ )
5. else return TREE-SEARCH($x$.RightChild, $k$ )

Running Time: $O(h)$, $h$ – the height of the tree

Finding the Minimum in a Binary Search Tree

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered
Alg: TREE-MINIMUM($x$)
1. while $x$.LeftChild $\neq$ NIL
2. do $x \leftarrow x$.LeftChild
3. return $x$

Running time: $O(h)$, $h$ – height of tree
Finding the Maximum in a Binary Search Tree

- **Goal:** find the maximum value in a BST
  - Following right child pointers from the root, until a NIL is encountered

**Alg:** TREE-MAXIMUM(Binary Node: x)
1. while x.RightChild ≠ NIL
   1. x ← x.RightChild
2. return x

- Running time: O(h), h – height of tree

Maximum = 20

---

Successor

**Def:** successor(x) = y, such that y.Key is the smallest key > x.Key

- **E.g.:** successor(15) = 17
  - successor(13) = 15
  - successor(9) = 13

- **Case 1:** x.RightChild is non-empty
  - successor(x) = the minimum in x.RightChild

- **Case 2:** x.RightChild is empty
  - go up the tree until the current node is a left child: successor(x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element and successor(x) = nil.

---

Finding the Successor (with parent link)

**Alg:** TREE-SUCCESSOR(Binary Node: x)

// Pre: x ≠ NIL
// Post: return the successor of node x
1. if x.RightChild = NIL
   1. then return TREE-MINIMUM(x.RightChild)
2. y ← x.parent
3. while y ≠ NIL and x ≠ y.RightChild
4. do x ← y
5. y ← y.parent
6. return y

Running time: O(h), h – height of the tree

---

Predecessor

**Def:** predecessor(x) = y, such that y.Key is the biggest key < x.Key

- **E.g.:** predecessor(15) = 13
  - predecessor(9) = 7
  - predecessor(7) = 6

- **Case 1:** x.LeftChild is non-empty
  - predecessor(x) = the maximum in x.LeftChild

- **Case 2:** x.LeftChild is empty
  - go up the tree until the current node is a right child: predecessor(x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the smallest element and predecessor(x) = nil.

---

Finding the Successor (without parent link)

**Alg:** Find-SUCCESSOR(KeyType: k, Binary Node: root, default)

// return the minimum node x in the tree rooted by "root" where
// x.Key > k; if k >= the max key in the tree, return "default"
1. if [root == NIL] return default;
2. if (root.Key == k)
3. if (root.RightChild == NIL)
4. return TREE-MINIMUM(root.RightChild)
5. else return default
6. else if (root.Key > k)
7. return Find-SUCCESSOR(root.LeftChild, root)
8. else
9. return Find-SUCCESSOR(root.RightChild, default)

Running time: O(h), h – height of the tree

---

Adding Nodes

- If the new value is smaller than the current node's value, move down the left branch
- If the new value is larger than the current node's value, move down the right branch
- When there is no branch, add a new node
**Insertion**

- **Goal:**
  - Insert value \( v \) into a binary search tree
- **Idea:**
  - If \( x.\text{Key} < v \) move to the right child of \( x \),
  - else move to the left child of \( x \)
  - When \( x \) is NIL, we found the correct position
  - If \( v < y.\text{Key} \) insert the new node as \( y \)'s left child
  - else insert it as \( y \)'s right child
  - Beginning at the root, go down the tree and maintain:
    - Node \( x \): traces the downward path (current node)
    - Node \( y \): parent of \( x \) ("trailing pointer")

**Alg:** TREE-INSERT(BinaryNode: root, z)

1. \( y \leftarrow \text{NIL} \)
2. \( x \leftarrow \text{root} \)
3. while \( x \neq \text{NIL} \)
4. \( y \leftarrow x \)
5. if \( z.\text{Key} < x.\text{Key} \)
6. then \( x \leftarrow x.\text{LeftChild} \)
7. else \( x \leftarrow x.\text{RightChild} \)
8. \( z.\text{parent} \leftarrow y \)
9. if \( y == \text{NIL} \)
10. then \( \text{root} \leftarrow z \)  
    Tree was empty
11. else if \( z.\text{Key} < y.\text{Key} \)
12. then \( y.\text{LeftChild} \leftarrow z \)
13. else \( y.\text{RightChild} \leftarrow z \)  
    Running time: \( O(h) \)

**Example:** TREE-INSERT

Insert 13:

**Deletion**

- **Goal:**
  - Delete a given node \( z \) from a binary search tree
- **Idea:**
  - **Case 1:** \( z \) has no children
    - Delete \( z \) by making the parent of \( z \) point to NIL
  - **Case 2:** \( z \) has one child
    - Delete \( z \) by making the parent of \( z \) point to \( z \)'s child, instead of to \( z \)
  - **Case 3:** \( z \) has two children
    - \( z \)'s successor (\( y \)) is the minimum node in \( z \)'s right subtree
    - \( y \) has either no children or one right child (but no left child)
    - Delete \( y \) from the tree (via Case 1 or 2)
    - Replace \( z \)'s key and satellite data with \( y \)'s.
TREE-DELETE(root, z)

1. if z.LeftChild == NIL or z.RightChild == NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
2. if y.LeftChild == NIL or y.RightChild == NIL
   then y ← z
3. else y ← y.RightChild
4. if y.LeftChild == NIL
   then x ← y.LeftChild
5. else x ← y.RightChild
6. if x == NIL
   then root ← y
7. if y.parent == NIL
   then root ← x
8. if y == (y.parent).LeftChild
   then (y.parent).LeftChild ← x
9. else (y.parent).RightChild ← x
10. if y == z then z.Key ← y.Key
11. return y

Binary Search Trees - Summary

- Operations on binary search trees (h: height of tree):
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESSOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)

- These operations are fast if the height of the tree is *small* — otherwise their performance is similar to that of a linked list

**Operations**

<table>
<thead>
<tr>
<th>Operations</th>
<th>Unsorted array</th>
<th>Sorted array</th>
<th>Binary search tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(h)</td>
</tr>
<tr>
<td>PREDECESSOR</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>SUCCESSOR</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(h)</td>
</tr>
<tr>
<td>HICUMUM</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(h)</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>INSERT</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>DELETE</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

- These operations are fast if the height of the tree is *small* — otherwise their performance is similar to that of a linked list