1.204 Lecture 6

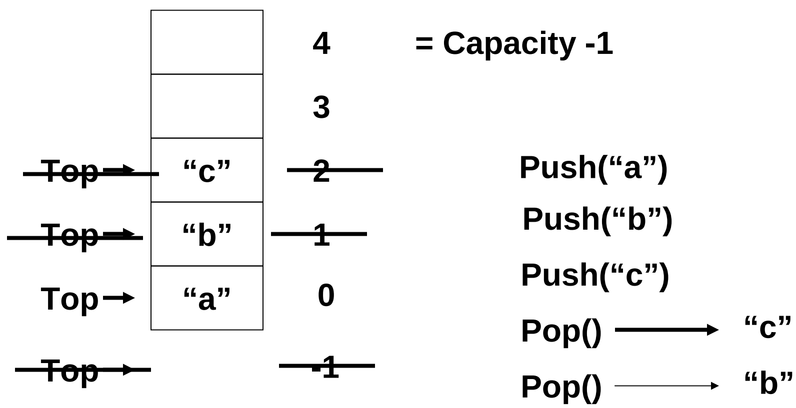
Data structures: stacks, queues, trees, dictionaries

Data structures

- Correct and efficient representation of data and applicable rules
  - Stack: last in, first out discipline
  - Queue: first in, first out discipline
    - Double-ended queue (deque): general line discipline
  - Heap: priority queue discipline
  - Tree:
    - Binary search tree (BST): ordered data, using a key
    - Heaps are represented using binary tree
    - Many other tree variations (B-tree, quadtree, AVL tree…)
  - Set:
    - Disjoint sets of elements, modeled as forest: set of disjoint trees
  - Graph/network:
    - Set of nodes and arcs (with costs)
  - (Arrays are a simple data structure but are not as efficient nor do they ensure correctness)

Stacks

Stack s



Using a Stack

```
public class StackTest {
        public static void main(String args[]) {
                int[] array = { 12, 13, 14, 15, 16, 17 };
                Stack stack = new Stack();
                for (int i : array) {
                        stack.push(i);
                }
                while (!stack.isEmpty()) {
                        int z= (Integer) stack.pop();
                        System.out.println(z);
                }
        }
}
// Output: 17 16 15 14 13 12
```

Stack, 1

```
import java.util.*;

public class Stack {
   public static final int DEFAULT_CAPACITY = 8;
   private Object[] stack;
   private int top = -1;
   private int capacity;

   public Stack(int cap) {
      capacity = cap;
      stack = new Object[capacity];
   }
   public Stack() {
     this( DEFAULT_CAPACITY );
   }
```

Stack, 2

```
  public boolean isEmpty() {
    return (top == -1);
  }

  public void clear() {
    top = -1;
  }
```

Stack, 3

```
public void push(Object o) {
  if (++top == capacity)
    grow();
   stack[top] = o;
}

private void grow() {
  capacity *= 2;
  Object[] oldStack = stack;
  stack = new Object[capacity];
  System.arraycopy(oldStack, 0, stack, 0, top);
}
```

Stack, 4

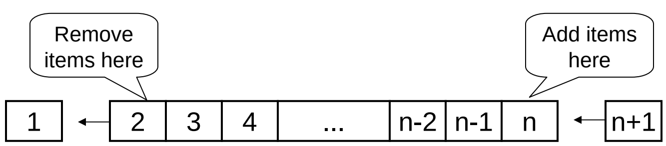
```
public Object pop()
   throws EmptyStackException
 {
      if (isEmpty())
              throw new EmptyStackException();
      else {
              return stack[top--];
      }
 }

// Java has Stack class that will be deprecated soon
// Java suggests using Deque for stack and queue
```

Stack uses and efficiency

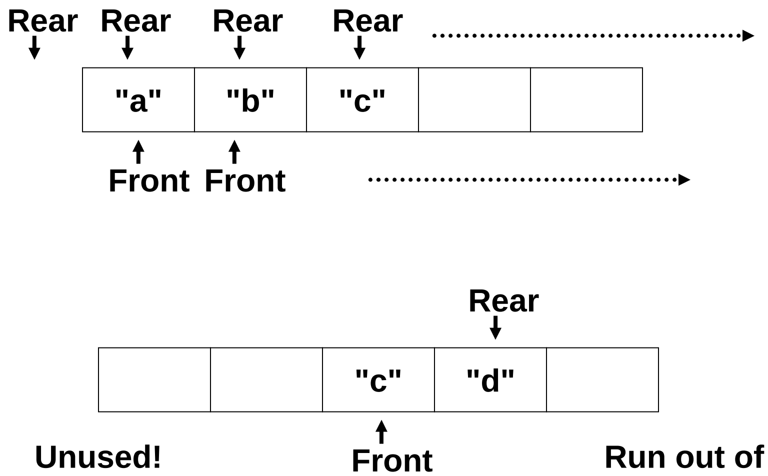
- Applications
  - Keep track of pending operations
    - Tree branches not explored (branch and bound)
    - Divide and conquer splits not completed/combined yet
    - Hierarchical communications networks (e.g., MPLS)
  - Physical stacks of items
  - Expression evaluation (with precedence)
- Efficiency
  - Pop() and push() are both O(1)
    - Size of stack does not affect these methods
  - Space complexity of stack is O(n)

Queues

A *queue* is a data structure to which you add new items at one end and remove old items from the other.



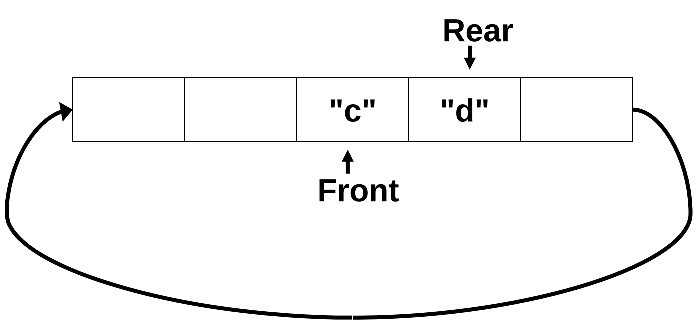
Queue



Unused!

Run out of room!

Queue

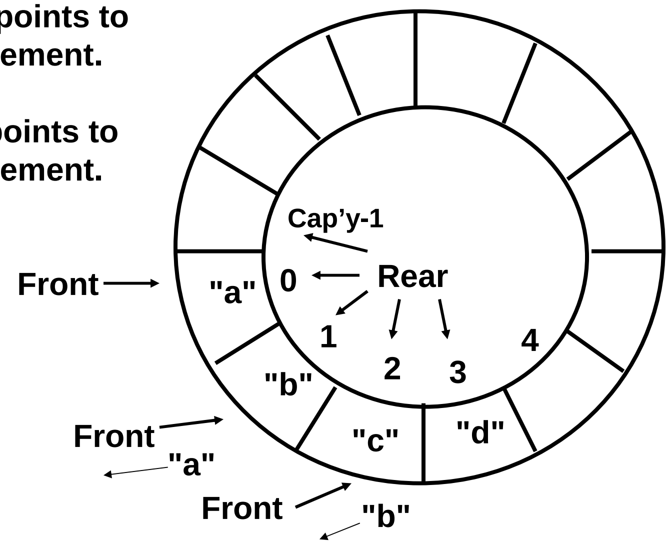


Wrap around!

Ring Queue

Front points to first element.

Rear points to rear element.



Queue

```
public class Queue {
      private Object[] queue;
      private int front;
      private int rear;
      private int capacity;
      private int size = 0;
      static public final int DEFAULT_CAPACITY= 8;
```

Queue Data Members

`queue`: **Holds a reference to the ring array**

`front`: **If `size>0`, holds the index to the next item to be removed from the queue**

`rear`: **If `size>0`, holds the index to the last item that was added to the queue**

`capacity`: **Holds the size of the array referenced by `queue`**

`size`: **Always >=`0`. Holds the number of items on the queue**

Queue Methods

```
public Queue(int cap) {
    capacity = cap;
    front = 0;
    rear = capacity - 1;
    queue= new Object[capacity];
}

public Queue() {
   this( DEFAULT_CAPACITY );
}

public boolean isEmpty() {
    return ( size == 0 );
}

public void clear() {
    size = 0;
    front = 0;
    rear = capacity - 1;
}
```

Queue Methods

```
 public void add(Object o) {
        if ( size == capacity )
            grow();
        rear = ( rear + 1 ) % capacity;
        queue[ rear ] = o;
        size++;
    }

  public Object remove() {
    if ( isEmpty() )
      throw new NoSuchElementException();
    else {
      Object ret = queue[ front ];
      front = (front + 1) % capacity;
      size--;
      return ret;
    }
  }
// See download code for grow() method and for QueueTest class
```

Queue uses and efficiency

- **Queue applications:**
  - **First in, first out lists, streams, data flows**
  - **Buffers in networks and computers**
  - **Physical queues**
  - **Keep track of pending operations**
    - **Tree branches not explored in branch-and-bound, etc.**
  - **Label correcting shortest path algorithms**
    - **Use a ‘candidate list’ queue that allows arbitrary insertions**
- **Queue efficiency:**
  - **add() and remove() are O(1)**
    - **Constant time, regardless of queue size**
  - **Space complexity of stack is O(n)**
    - **Where n is maximum queue size, not number of items processed**

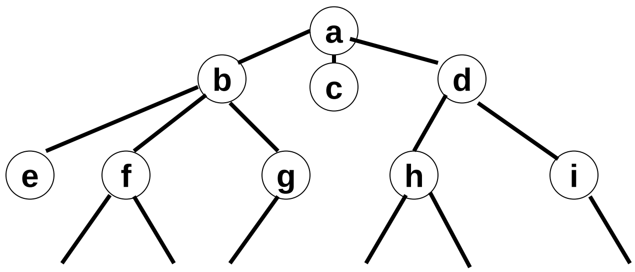
Tree definitions

Level
(distance from root)
0

1

2

...



Root: a
Degree (of node): number of subtrees
b:3, c:0, d:2
Leaf: node of degree 0: e, c
Branch: node of degree >0
Depth: max level in tree

Children: of a are b, c, d
Parent: of g is b
Siblings: children of same parent: b, c, d
Degree of tree: max degree of its nodes(3)
Ancestors: nodes on path to root:
g's ancestors are b and a

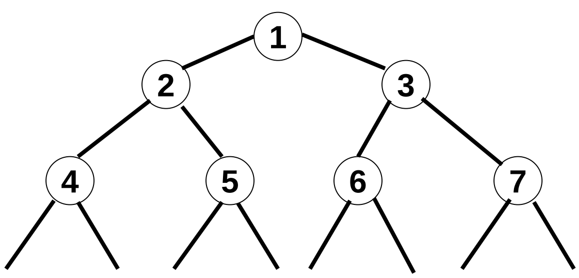
Binary tree definitions

Level
0

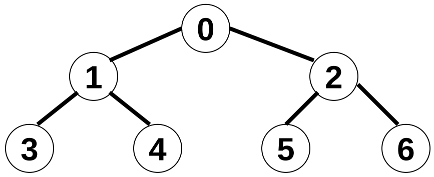
1

2

...



Max nodes on level i= 2^i
Max nodes in tree of depth k= $2^{k+1}-1$
(full tree of depth k)



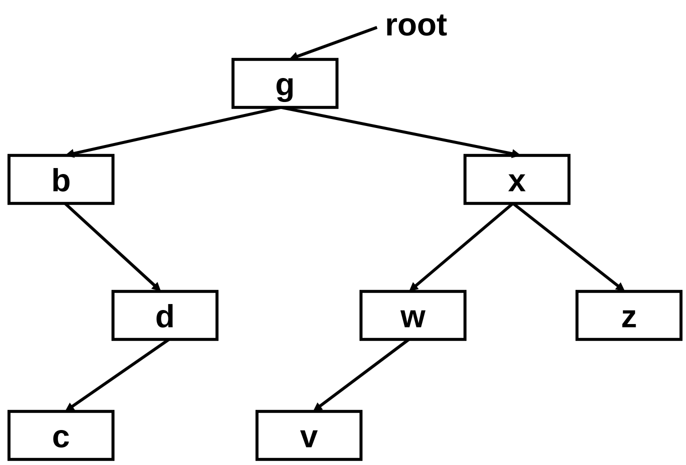
<u>Complete binary tree in array:</u>
Parent[i]= i/2
LeftChild[i]= 2i
RightChild[i]= 2i+1

<u>If root is node 0 (rather than 1):</u>
Parent[i]= (i-1)/2
LeftChild[i]= 2i+1
RightChild[i]= 2i+2

Tree Traversal

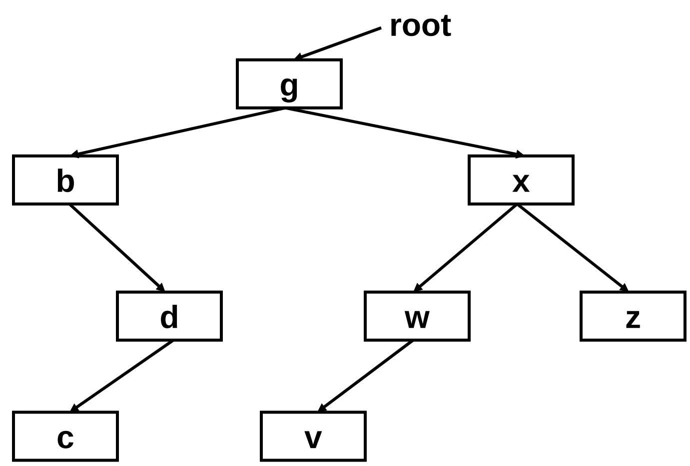
- We call a list of a tree's nodes a traversal if it lists each tree node exactly once.
- The three most commonly used traversal orders are recursively described as:
  - Inorder: traverse left subtree, visit current node, traverse right subtree
  - Postorder: traverse left subtree, traverse right subtree, visit current node
  - Preorder: visit current node, traverse left subtree, traverse right subtree

Tree traversal examples



Inorder: b c d g v w x z

Tree traversal examples



Postorder: c d b v w z x g

Binary Search Trees

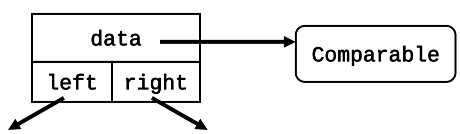
- There are many ways to build binary trees with varying properties:
  - In a heap or priority queue, the largest element is on top. In the rest of the heap, each element is larger than its children
  - In a binary search tree, the left subtree has nodes smaller than or equal to the parent, and the right subtree has nodes bigger than or equal to the parent
    - We saw that performing an *inorder* traversal of such a tree visited each node in order
- We'll build a binary search tree in this lecture and a heap in the next lecture

Writing a Binary Search Tree

- **We'll build a `Tree` class:**
  - **One data member: `root`**
  - **One constructor: `Tree()`**
  - **Methods:**
    - **`insert`: build a tree, node by node**
    - **`inorder` traversal**
    - **`postorder` traversal**
    - **(we omit preorder)**
    - **`find:` whether an object is in the tree**
    - **`print` tree**

Writing a BST, p.2

- **We also build a `Node` nested class inside `Tree`:**
  - **Three data members: `data, left, right`**
    - **`data` is a reference to an `Object`, so our `Node` is general**



  - **Our `data` Objects must implement the `Comparable` interface, which has one method:**

    ```
    int compareTo(Object other)
    ```
  - **`compareTo` returns:**
    - **An int < 0 if (this < other)**
    - **0 if (other equals this)**
    - **An int > 0 if (this > other)**

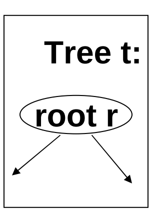
Writing a BST, p.3

- Node class also has a set of methods, all used by corresponding methods in the `Tree` class:
  - `insertNode`
  - `traverseInorder`
  - `traversePostorder`
  - `findNode`
  - `printNode`
- Methods are invoked on root node and then traverse the tree as needed

Tree and Node Classes

```
Tree:
  private Node root;
  public Tree() {root=null;}
  public void inorder() {…}
  public void postorder() {…
  public void insert(o) {…}
  public boolean find(o) {…}
  public void print() {…}
```

Tree methods invoked on Tree object; they call Node methods invoked on the root node object




```
Nod
  publ       rable data;
  publ          t, right;
  publ       o) {data=o;}
  public void traverseInorder() {…}
  public void traversePostorder() {…}
  public void insertNode(n) {…}
  public boolean findNode(o) {…}
  public void printNodes() {…}
```

Tree class

```
public class Tree {
    private Node root;

    public Tree() {
        root= null;    }

    public void inorder() {
        if (root != null)  root.traverseInorder();   }

    public void postorder() {
        if (root != null)  root.traversePostorder(); }

    public void insert(Comparable o) {
        Node t= new Node(o);
        if (root==null)
            root= t;
        else
            root.insertNode(t);    }
```

Tree class, p.2

```
    public boolean find(Comparable o) {
        if (root== null)
            return false;
        else
            return root.findNode(o);
    }

    public void print() {
        if (root != null)
            root.printNodes();
    }
```

Node class: data, constructor

```
private static class Node {
    public Comparable data;
    public Node left;
    public Node right;

    public Node(Comparable o) {
        data= o;
        left= null;
        right= null;
    }
```

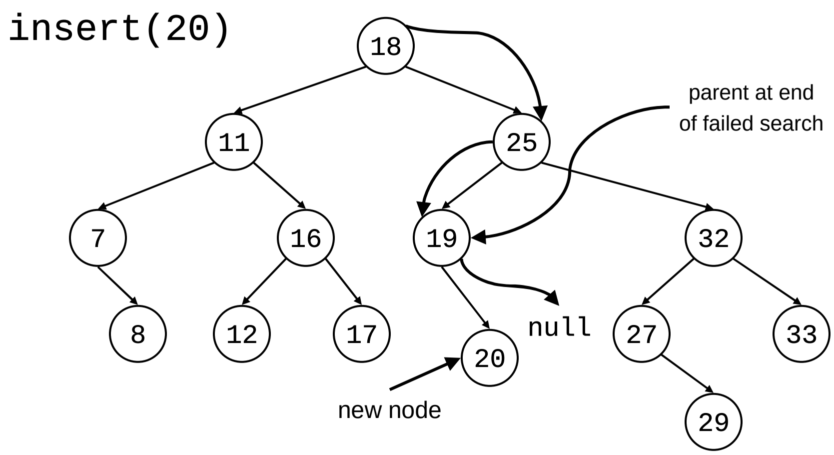
Traversal

```
public void traverseInorder() {
    if (left != null) left.traverseInorder();
    System.out.println(data);
    if (right != null) right.traverseInorder();
}

public void traversePostorder() {
    if (left != null) left.traversePostorder();
    if (right != null) right.traversePostorder();
    System.out.println(data);
}
```

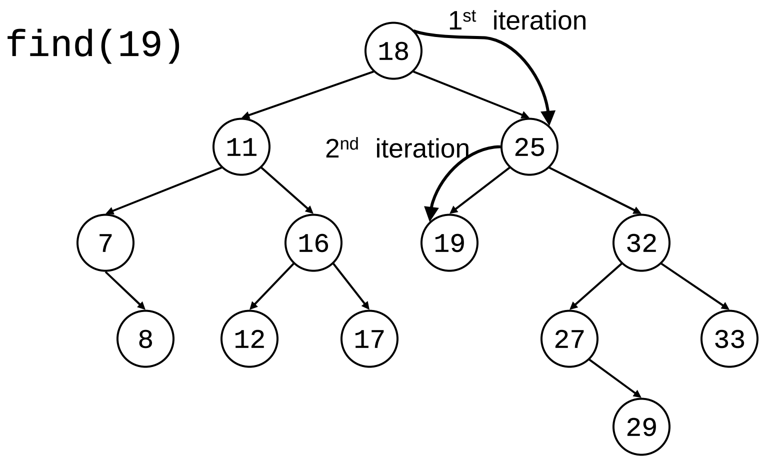
Node class, insertNode

```
public void insertNode(Node n) {
    if (data.compareTo(n.data) > 0) {
        if (left==null)
            left= n;
        else
            left.insertNode(n);
    }
    else {
        if (right == null)
            right= n;
        else
            right.insertNode(n);
    }
}
// No ties allowed
```

`insert()` in Action



find() in Action



Find

```
public boolean findNode(Comparable o) {
            if (data.compareTo(o) > 0) {
                if (left== null)
                    return false;
                else
                    return left.findNode(o);
            }
            else if (data.compareTo(o) < 0) {
                if (right == null)
                    return false;
                else
                    return right.findNode(o);
            }
            else    // Equal
                return true;
}
```

Keys and Values

- **If binary search trees are ordered, then they must be ordered on some *key* possessed by every tree node.**
- **A node might contain nothing but the *key*, but it's often useful to allow each node to contain a *key* and a *value*.**
- **The *key* is used to look up the node. The *value* is extra data contained in the node indexed by the *key*.**

Maps/Dictionaries

- **Such data structures with key/value pairs are usually called *maps* or sometimes *dictionaries***
- **As an example, consider the entries in a phone book as they might be entered in a binary search tree. The subscriber name, last name first, serves as the *key*, and the phone number serves as the *value*.**

Maps

- Implementing tree structures with keys and values is a straightforward extension to what we just did. The Node contains the same members:
  - Data, a reference to a Comparable object with key and value
  - Left
  - Right
- We add or modify methods to set or get the values associated with the keys
  - No change in logic
- Map example on next slides
  - This could be improved by having find() return the Object instead of a boolean whether it was found
  - You'd then have to check if the object is null, etc.
  - These are straightforward changes, but we show the simplest implementation here

Phone class

```
public class Phone implements Comparable {
   private String name;              // Name of person (key)
   private int phone;                // Phone number (value)

   public Phone(String n, int p) {
        name= n;
        phone= p;
   }

   public int compareTo(Object other) {
        Phone o= (Phone) other;
        return this.name.compareTo(o.name);   // String compare
   }

   public String toString() {
        return("Name: "+ name +" phone: "+ phone);
   }
}
// This will be the object pointed to by 'data' in Node
```

MapTest

```
public class MapTest {
   public static void main(String[] args) {
        Tree z= new Tree();
        z.insert(new Phone("Betty", 4411));
        z.insert(new Phone("Quantum", 1531));
        z.insert(new Phone("Thomas", 6651));
        z.insert(new Phone("Darlene", 8343));
        z.insert(new Phone("Alice", 6334));
        z.print();
        System.out.println("Inorder");
        z.inorder();
        System.out.println("Postorder");
        z.postorder();
        System.out.println("Search for phone numbers");
        System.out.println("Find Betty? " +
               z.find(new Phone("Betty", -1)));
        System.out.println("Find Thomas? " +
               z.find(new Phone("Thomas", -1)));
        System.out.println("Find Alan? " +
               z.find(new Phone("Alan", -1)));
    }
}    // TreeTestGeneric and TreeGeneric use Java 1.6 generics
```

Tree uses and efficiency

- **Applications**
  - **Data storage and search**
  - **Optimization: search discrete alternatives (DP, B-and-B)**
  - **Priority queues**
  - **Shortest paths, spanning trees on graphs**
  - **Basis in network simplex**
- **Efficiency**
  - **insert(), delete(), find() are O(lg n) average case**
    - **We don't cover delete()—it's straightforward but tedious**
  - **Degenerate trees are O(n) but can be avoided with care**
    - **Never build a tree in sorted order**
    - **Choose a non-key field to sort input to build the tree**
  - **AVL, red-black trees rebalance to avoid worst case**
- **Most of our trees will keep parent node, not children this term**