Polymorphism

Method *overriding* (same signature in subclass — narrowly tailored)
  Leads to **dynamic method selection**
Method *overloading* (same name but different parameters same class)
  Leads to automatic selection of the suitable method

Casting

Treat an expression as having a different compile-time type

Dynamic Method Selection

At Compile Time

(1) Check syntax correct; variables are properly declared
(2) **If there is a cast:** We have a variable with static type **`S`** being cast to the class **`C`**. Is a **`C`** an **`S`**, or is an **`S`** a **`C`**? Are **`C`** and **`S`** on the same inheritance path? [ERROR if false]
(3) **If there is a method call:**
  (1) We have a variable of static type **`S`** calling method **`M`** taking in parameters of type **`P`**. Does the class **`S`** have a method **`M`** that takes parameters **`P`**?
  (2) If not, does a parent class of **`S`** have a method **`M`** that takes parameters **`P`**?
  (3) If not, does **`S`** have a method **`M`** that takes parameters **`Q`**, where **`Q`** is a superclass of **`P`**? Does the class S have a method **`M(Q)`** that would accept a **`P`** because a **`P`** is a **`Q`**?
  (4) If not, does a parent class of **`S`** have a method **`M`** that takes parameters **`Q`**, where **`Q`** is a superclass of **`P`**? [ERROR if false]
(4) **If there is an assignment:** We are assigning a variable of static type **`S`** to some value on the right side with static type **`V`**. Is a **`V`** an **`S`**? Does **`V`** inherit from **`S`**? [ERROR if false]
(5) The line compiles → proceed to runtime!

At Runtime

(1) Was there a cast on this line? (YES) → (2) (NO) → (3)
(2) We casted a variable of dynamic type **`D`** to the class **`C`**. Is a **`D`** a **`C`**? Does **`D`** inherit from **`C`**? (YES) → (3) (NO) → ERR
(3) Is there a method call on this line? (YES) → (4) (NO) → (7)
(4) We have a variable of dynamic type **`D`** calling a method. Use the method signature saved from compile time. Does the class **`D`** have a method with the exact same signatures, including the parameter types? (YES) → (5) (NO) → (6)
(5) Execute that method in **`D`** (YES) → (7)
(6) Go up the inheritance tree until you find a method that matches the signatures exactly. (YES) → (7)
(7) The line runs!

Inheritance

A class can *implement* an interface
  `SLList<Blorp>` **`implements`** `List61B<Blorp>`
A class can *extend* another class
  `DLList<Blorp>` **`extends`** `SLList <Blorp>`
A child class can refer to its parent using `super`
  This includes constructors, done automatically by default
Inheritance is for "**is-a**" relationships

Asymptotic

$\Theta(\cdot)$ represents precise bound
$O(\cdot)$ represents maximum (upper bound)
$\Omega(\cdot)$ represents minimum (lower bound)

Iterators

```
for (int x : javaset) { // Equivalent to the code below
   System.out.println(x);
}
Iterator<Integer> seer = javaset.iterator();
while (seer.hasNext()) {
    System.out.println(seer.next());
}
```

To implement iterators, add `iterator()` method that returns an `Iterator<T>` and the `Iterator<T>` we return needs to have a `hasNext()` and `next()`

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
public interface Iterable<T> {
    Iterator<T> iterator(); ...
}
private class ArraySetIterator implements Iterator<T> {
    private int wizPos;
    public ArraySetIterator() { wizPos = 0; }
    public boolean hasNext() {
        return wizPos < size;
    }
    public T next() {
        T returnItem = items[wizPos];
        wizPos += 1;
        return returnItem;
    }
}
public Iterator<T> iterator() {
    return new ArraySetIterator();
}
```

Comparator

```
public interface Comparator<T> {
      int compare(T o1, T o2);
}
class Movie implements Comparable<Movie> {
    public int star; public String name; public int year;
    public int compareTo(Movie m) {
        return this.year - m.year;
    }
}
class RatingCompare implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}
```

Disjoint Sets

QuickUnion		**Weighted QuickUnion**	
`constructor`	$\Theta(N)$	`constructor`	$\Theta(N)$
`connect`	$O(N)$	`connect`	$O(\log N)$
`isConnected`	$O(N)$	`isConnected`	$O(\log N)$

1. Locate the n^{th} index
2. Change it to the index of m
3. Change the target's root by -1 to indicate size

WQU: Connect the smaller tree to the root of the larger tree

Mergesort

Create a sorted array from two sorted arrays → $O(n \log(n))$ runtime

Trees

BST: * Every key in the left subtree is less than current node
* Every key in right subtree is greater than current node

	Height/Length	Contains	Add
LinkedList	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$
ArraySet	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$
BST/Bushy	$\Theta(N)$ / $\Theta(\log N)$	$\Theta(N)$ / $\Theta(\log N)$	$\Theta(N)$ / $\Theta(\log N)$
LLRB	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(\log N)$
B-Trees	$\Theta(\log N)$	$\Theta(\log N)$	$\Theta(\log N)$
Hash Table Good Hash	$\frac{\text{N}}{\text{Load Factor}}$	$\Theta(N)$ / $\Theta(Q)$	$\Theta(N)$ / $\Theta(Q)$

Deletion from a BST: (Hibbard deletion)

1. If no children, just delete it
2. If one child, delete node and have child take its place
3. If two children, delete node & pick either the rightmost node of the left subtree or the leftmost node of the right subtree

Set vs. Map

Trees represent a set; to represent a map, add attribute to each node

B-Trees (B for Balanced)

Each leaf node can contain multiple keys, up to a limit L

Leaf nodes with more than L keys will split up

LLRB

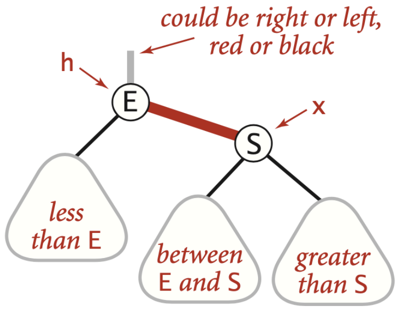
Representing a 2-3 tree as a binary search tree

Two red links not allowed (can't have a 4 node)

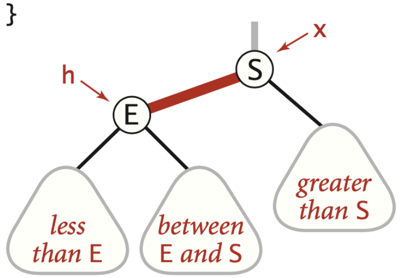
Path from root to each null reference = same num. of black links

Tree Rotations

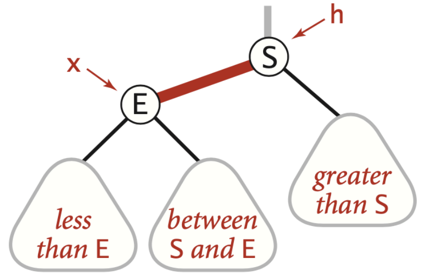
Motivation: Preserve the Left-Leaning Property



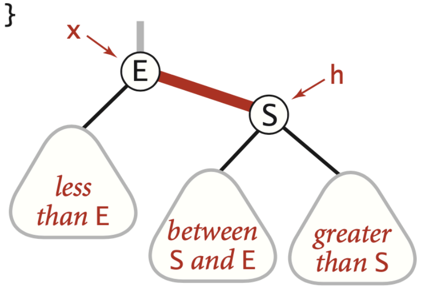

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Left rotate (right link of h)




```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



Right rotate (left link of h)

Like temporarily merging **S** and **E**, then sending middle child down¶

Hashing

Collection of Buckets, index refers to **hashcode** of objects

$\Theta(Q)$ where Q is the length of the longest list (constant if good hash)

$$\text{Load Factor} = \frac{N = \text{Number of Items}}{M = \text{Number of Buckets}}$$

1. Figure out the **hashcode**
2. Use modulo function to convert **hashcode** to index ($n \,\%\, 4$)
3. Insert into the appropriate index

Caveats

1. Don't store objects that can change, or they will get lost
2. Don't override `equals` without also overriding **hashcode**

Applications

TreeSet / TreeMap — Store sorted values

HashSet / HashMap — Storage based on **hashCode**

WeightedQuickUnion — Track Connectedness

Queue — First-in-first out

Stack — Last-in-first-out

Asymptotic Patterns

$1+2+3+4+5+\ldots+N = \Theta(N^2)$ $\qquad 2^{\log_2 N} = N$

$1+2+4+8+16+\ldots+N = \Theta(N)$

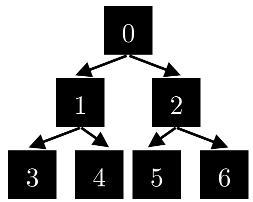
algorithm	order of growth for N sites (worst case) constructor	union	find
quick-find	N	N	1
quick-union	N	*tree height*	*tree height*
weighted quick-union	N	$\lg N$	$\lg N$
weighted quick-union with path compresson	N	*very, very nearly, but not quite 1 (amortized)* (see EXERCISE 1.5.13)	

Heaps / Priority Queue

Tracking and removal of smallest (or largest) item

1. Each node is smaller than its children
2. Tree is bushy and nodes as far left as possible

Store keys in an array — assuming complete trees



1	2	3	4	5	6

`add(x)` | insert **`x`** at the last position, and promote as high as possible

`removeSmallest()` | remove root, pick rightmost number (i.e. 6) to be at root, and demote repeatedly, always taking the best successor

Amortized $\Theta(\log N)$ **`add`** and **`remove`** (Resizing averaged out)

$\Theta(1)$ **`getSmallest`**

Tree Traversals

Depth First Search $0 \to 1 \to 3 \to 4 \to 2 \to 5 \to 6$

Breadth First Search$0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$

Preorder **`doSthAtNode`** → **`traverse(left)`** → **`traverse(right)`**

"visit" every time we pass the *left* of a node

Inorder **`traverse(left)`** → **`doSthAtNode`** → **`traverse(right)`**

"visit" every time we pass the *bottom* of a node

Postorder **`traverse(left)`** → **`traverse(right)`** → **`doSthAtNode`**

"visit" every time we pass the *right* of a node

```
bfs(graphNode s) {
    queue = new Queue<>()
    mark s as visited
    queue.add(s)
    while (!queue.isEmpty()):
        v = queue.pop()
        for each unmarked neighbor of v:
            edgeTo[neighbor] = v;
            marked[neighbor] = true
            queue.add(neighbor) }
```

```
dfs(Graph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w); }
```