Heaps
public void insert(Comparable[] heap, Comparable x) {
    if(size == heap.length - 1) doubleSize();

    int pos = ++size; // Insert a new item to the end of the array
    // Percolate up
    for(; pos > 1 && x.compareTo(heap[pos/2]) < 0; pos = pos/2 )
        heap[pos] = heap[pos/2];

    heap[pos] = x;
}
public void delete(Comparable[] heap, int index)
{
    heap[index] = heap[size]; // override previous element with last element
    heap[size--] = -infinity; // or infinity for min heap
    heapify(index);
}
public void buildHeap(Comparable[] heap, Comparable[] arr){
    for(int i=0; i<arr.length; i++)
        heap[i] = arr[i]; // copy all data into heap
    for(int i = arr.length/2; i >= 0; i++)
        MaxHeapify(heap, i);
}
Hashing
Types of Hashing Functions

1. Linear Probing
   a. $H(k,i) = H(k) + c*i \pmod{n}$

2. Quadratic Probing
   a. $H(k,i) = H(k) + c*i + d*i^2 \pmod{n}$

3. Double Hashing
   a. $H(k) = f(k) + g(k) \pmod{n}$
      i. $f(k)$ is mod $p$ and $g(k)$ is mod $q$ where $(p,q)=1$
Double Hashing

The two composing hashing functions must have coprime moduli.

Easiest to do when at least one number is prime.

How to make a prime?

1. Generate a number (carefully)
2. Check if it’s prime
3. Repeat until prime is found
Finding a Prime

Note that for all primes $p>3$, there exists integer $x$ such that $p = 6x±1$

Generate an $x$ and compute a potential $p$. (Note that this is not always prime)

Use a primality checker on this number. An easy method is Wilson’s theorem.

Theorem: $n$ is prime iff $(n-1)! = -1 \pmod{n}$

Proof: Let $g(x)=x(x-1)...(x-p+1)$ and $h(x)=x^{p-1}-1$. Both have roots $1, 2, ..., p-1$ and are of degree $p-1$, and $g(x)$ has constant term $(p-1)!$ by Binomial Thm. Consider $f(x)=g(x)-h(x)$. Then $f$ has degree $p-2$ but still has $p-1$ roots, which contradicts Lagrange’s Thm. So $\pmod{p}$, $f(x)=0 \iff g(x)-h(x)=0 \iff (p-1)!+1=0 \iff (p-1)!=-1$. ∎
Comparison Based Sorting
Lower Bound on Comparison Based Sorting

Our best algorithms have nlogn time. Is this truly a lower limit? Yes.

Note: For a collection of size n, there are n! possible orderings of the collection.

Note 2: \( \log(n!) \sim \log(n^n) \sim n \log(n) \) by Stirling’s approximation.

(Proof sketch on board.)
Binary Trees
Types of Trees for this Presentation

- Binary Search
- AVL Tree
- Red-Black
- 2-3
Important BST Property - Height

The height of a balanced binary tree is \( \sim \log(n) \). Useful for searching and modifying the tree.

Proof: For a full binary tree, the number of nodes per level \( k \) is \( L_k = 2^{L_{k-1}} \), \( L_0 = 1 \).

Solving the recurrence, \( L_k = 2^k \). So if we have tree of height \( h \) with \( n \) nodes, we see

\[
\begin{align*}
  n &= 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = (1-2^h)/(1-2) \\
  n &= 2^h-1 \\
  n+1 &= 2^h \\
  \log_2(n+1) &= h \sim \log(n) \quad \blacksquare
\end{align*}
\]
AVL Tree

How to keep a tree balanced after every operation? AVL Trees.

For all nodes in the tree, the height of the left and right subtrees can differ by no more than one. This guarantees that the tree is balanced.

This is fairly difficult to implement, as you need to update the height of each subtree after every modification.
Red-Black Tree

AVL Trees are hard to implement, so we can engineer a close-enough solution.

RB Trees allow the height of the smallest and largest subtrees to differ by a factor of 2 at most.

Each node contains a color property, which holds either red or black. This allows for properly balanced trees.
RB Left Rotation

TreeNode<T> leftRotate(TreeNode<T> root, TreeNode<T> x)

//returns a new root; Pre: right child of x is a proper node (with value)
{

TreeNode<T> z = x.getRight();

x.setRight(z.getLeft());

// Set parent reference

if (z.getLeft() != null) //x is not the root
    if (x == x.getParent().getLeft()) //left child
        x.getParent().setLeft(z);
    else
        x.getParent().setRight(z);
else
    z.setLeft(x); //move x down;

z.setParent(x.getParent());

// Set parent reference of x

if (xParent != null) //x is not the root
    if (x == x.getParent().getLeft()) //left child
        x.getParent().setLeft(z);
    else
        x.getParent().setRight(z);
else
    root = z;

x.setParent(z);

return root;
}
TreeNode<
	T> rbInsert(TreeNode<
	T> root, TreeNode<
	T> x)

// returns a new root
{
    root = bstInsert(root, x); // a modification of BST insertItem
    x.setColor(red);
    while (x != root and x.getParent().getColor() == red) {
        y = x.getParent().getParent().getRight(); // uncle of x
        if (y.getColor() == red) {// uncle is red
            x.getParent().setColor(black);
            y.setColor(black);
            x.getParent().getParent().setColor(red);
            x = x.getParent().getParent();
        } else { // uncle is black
            // ................
        }
        } else {
            // parent is left child
            y = x.getParent().getParent().getRight(); // uncle of x
            if (y.getColor() == red) {// uncle is red
                x.getParent().setColor(black);
                x.getParent().getParent().setColor(red);
                y.setColor(black);
            } else { // uncle is black
                // ................
            }
        } else { // parent is right child
            // ............
        }
    } else { // uncle is black
        // ................
    }
    root.setColor(black);
    return root;
}
TreeNode<T> rbInsert(TreeNode<T> root, TreeNode<T> newNode)

// returns a new root
{
    root = bstInsert(root, newNode); // a modification of BST insertItem
    x.setColor(red);
    while (x != root and x.getParent().getColor() == red) {
        if (x.getParent() == x.getParent().getParent().getLeft()) {
            // parent is left
            y = x.getParent().getParent().getRight(); // uncle of x
            if (y.getColor() == red) {// uncle is red
                // ................
            } else { // uncle is black
                if (x == x.getParent().getRight()) {
                    x = x.getParent();
                    root = left_rotate(root, x);
                }
                x.getParent().setColor(black);
                x.getParent().getParent().setColor(red);
                root = right_rotate(root, x.getParent().getParent());
            }
        } else {
            // ... symmetric to if
        }
    }
    root.setColor(black);
    return root;
} // end while

root.setColor(black);
return root;
2-3 Trees

2-3 Trees are a variation of BST, where there are two possible nodes: nodes with a two children and a single data type, and nodes with three children and two data types.

Examples:

Searching

Insertion
Questions?