Binary Tree

Definitions

- a binary tree is **full** if every non-leaf node contains 2 children (i.e. every node has 0 or 2 children)
- a binary tree is **complete** have all levels completely filled except possibly the last level, and the last level has all nodes as left as possible
- a **perfect** Binary tree have all internal nodes filled two children, and all leaves are at the same level. A perfect binary tree is also full.

Height and size

- the maximum height of a binary tree with n nodes is $n-1$
- the minimum height of a binary tree with n nodes is $O(log(n))$
- the minimum number of nodes in a binary tree with height h is $h+1$
- the maximum number of nodes in a binary tree with height h is $2^{h+1}-1$

Binary Search Tree

Now, since we have a **Seach** tree, we need some sort of orderings inside the tree. All the nodes on the left subtree must be smaller than the parent/root node (**recursively**). All the nodes on the right subtree must be larger than the parent/root node.

Methods

- `contains()` / `insert()` / `findMin()` / `findMax()`
  - **Cost Analysis**
    - if we have a **balanced/complete/perfect** binary Search tree, then the complexity is $O(height) = O(log(n))$
    - if we have a **full** binary Search tree, then the complexity (in the worst case) is $O(\frac{n}{2}) = O(n)$
- `remove()`
  - To find that node, it takes the same cost as `contains()` . Then it depends on the height of the node being removed
  - if that node has only **one child**, just shift the child up.
  - if the node has **two children**, replace that node with either have the largest on the left subtree or the smallest on the right subtree. It works nicely because that node will only have **one child**. Then, you replace that node's original position itself with children (and the only children).

In [ ]:
```
/**
 * Note that it returns the root BinaryNode. This is to prevent the case
 * when the tree is empty, so that the root changes. Note its the NODE
 * that changed, hence we need rewindings
 */
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t ){
    // base case: node does not exist, so we REPLACE the null node to a
    // new BinaryNode
    if( t == null )
        return new BinaryNode<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    // looks similar to the contains() method, but it is quite different
    // it actually rewinds the ENTIRE Tree
    if( compareResult < 0 )
        // inserting in the left subtree
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        // inserting in the right subtree
        t.right = insert( x, t.right );
    else
        // Duplicate; do nothing

    // this is necessary, as we need to give the node back
    return t;
}
```

In [ ]:
```
private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t ){
    if( t == null )
        return t;   // Item not found; do nothing

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );
    // now the node is found
    else if( t.left != null && t.right != null ) // Two children
    {
        // replace the VALUE with the minimum of the right sub-tree
        // we did not actually remove anything
        t.element = findMin( t.right ).element;
        // then remove that replaced node from below, which MUST have
        // either one or zero children
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right; // one or zero children
    return t;
}
```

In [ ]:
```
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t ){
    if( t == null )
        return null;
    else if( t.left == null )
        return t;

    // tail recursion, when the very last return is the recursive call
    // in general, tail recursion is easy to rewrite into a while loop
    return findMin( t.left );
}
```

In [ ]:
```
private boolean contains( AnyType x, BinaryNode<AnyType> t ){
    if( t == null )
        return false;

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        return contains( x, t.left );
    else if( compareResult > 0 )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

Expression Tree

- Algorithm
  - everytime when we get an operand, you push the nodes into the stack
  - everytime when we get an operator, you pop TWO of the top nodes in the stack
  - **AFTER YOU POP the two operand nodes, you PUSH the node of the operand back in the stack** (Note that now you constructed a subtree where the opertor will be the parent nodes)
  - finally, if there is *no more operator/operands in the given expression*, you pop the stack and you will get the **ROOT of the tree**

- **Post-Fix** using **Expression Tree**
  - because the tree is constructed using stack, which means first in last out, we need **post-order traversal** to get the expression out in sequence and in **post-fix**

In [ ]:
```
public int evaluate(Node t){
    if(t.left == null && t.right == null){
        return t.operand;
    }
    int leftVal = evaluate(t.left);
    int rightVal = evaluate(t.right);

    // apply method does the corresponding mathematical operation
    // note that this t is at an upper level than the t in the if statement above
    // @returns an integer value after the mathematical expression
    return apply(t.operator,leftValue,rightValue);
}
```

AVL Tree

if we can have a **balanced binary tree**, such that if the height of every left subtree differs no more than 1 with the height of the right subtree, then the worst case operation cost will be $O(2log(n)) \approx O(log(n))$

Properties

- any **AVL Tree** is a **Binary Tree**
- fulfills the **AVL Condition**, or the **Balance Condition**, which says that *for every node, the height of a left subtree cannot differ from the right subtree by more than one*

Checking Algorithm

- We need to have a recursive algorithm that keeps track of the height of the subtree. This could be expensive, but one solution is that we add an additional field in the node, namely the `private int height`

In [ ]:
```
private int checkBalance( AvlNode<AnyType> t ){
    // if that subtree itself does not exist
    if( t == null )
        return -1;

    // recursion
    if( t != null )
    {
        int hl = checkBalance( t.left );
        int hr = checkBalance( t.right );
        if( Math.abs( height( t.left ) - height( t.right ) ) > 1 ||
                height( t.left ) != hl || height( t.right ) != hr )
            System.out.println( "OOPS!!" );
    }

    return height( t );
}
```

Balancing Algorithm

- it can only occur at a subtree with **3 nodes in a row**
- rotations will have the aim of putting the **median** to be the new root of the subtree

- **case 1:** Zig-Zig
  - **Single Rotation** the median/middle node up
    - careful of the secondary rewindings
    - use the fact that it is a **Binary Tree**, left node is always smaller than parent
- **case 2:** Zig-Zag
  - the **median** is not in the middle
  - **Double Rotation**
    - first rotate the **median** up to the middle
    - then **Single Rotation**
      - careful of the secondary rewindings
      - use the fact that it is a **Binary Tree**, left node is always smaller than parent

In [ ]:
```
// Assume t is either balanced or within one of being balanced
private AvlNode<AnyType> balance( AvlNode<AnyType> t ){
    if( t == null )
        return t;

    // first, determine if there is an imbalance or not
    // if so, which way is the imbalance

    // notice that we are using the height method instead of the field height
    // this is because we want to avoid the issue of asking for the height of null object
    if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
        // Zig-Zig. Hence single rotation
        if( height( t.left.left ) >= height( t.left.right ) )
            t = rotateWithLeftChild( t );
        // Zig-Zag. Hence double rotation
        else
            t = doubleWithLeftChild( t );
    else if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
        // Zag-Zag. Hence a single rotation
        if( height( t.right.right ) >= height( t.right.left ) )
            t = rotateWithRightChild( t );
        else
            t = doubleWithRightChild( t );

    // updates the height. Other updates are done inside rotateWithLeftChild methods
    t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

In [ ]:
```
// single rotation of the node itself with the left child
private AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 ){
    AvlNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    // the height of k1 and k2 has changed
    k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = Math.max( height( k1.left ), k2.height ) + 1;
    // the new subtree node
    return k1;
}
```

In [ ]:
```
// the double rotation for Zig-Zag
private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 ){
    // first single rotation of the left child with the left right child
    k3.left = rotateWithRightChild( k3.left );
    // another single rotation of the child and the left child which is rotated
    return rotateWithLeftChild( k3 );
}
```

HashTable

- then when you want to **find()** an object, you apply the **hash** function to that object, and just look up at that position
  - note the problem would be collision, which can happen
- **HashTable** basically does the same thing as a **HashSet**, which inherits the **Set** interface. *Both* are essentially a mathemtical set.
- **HashTable** only insert **key**, which has to be unique (a set), but not **value** (as compared to python **dictionaries** and Java **HashMap** where both are inserted)

Special Java Hash Functions

Note that **after** every hash function, you need to **mod** by `tableSize`

- `int` has hash functions being themselves, $hash(x) = x$
- `Object` has hash function being their memory address
- `String` has hash function being its content

Rehash

Create a new **HashTable** and **rehash** every thing we had into the new table (not just copying, since the MOD factor changed)

- **Rehash** operation of order $O(n) = O(N), N = tableSize$, and notice that during that **rehash** we do not need to check **contains()** since whatever was in the table cannot be duplicates
- also, since we need to keep the **tableSize** to be prime for minimizing collision

Seperate Chaining

- where each index position is a **head node** with a link
- `rehash()` when $\lambda > 1$

- **Cost Analysis**
  - since we have a **LinkedList** implementation, we will have $O(k)$ for `insert()` / `contains()` and `remove()`, where k is the number of elements inside that position's **LinkedList**
  - but, if we have $\lambda \leq 1$, the average cost of `insert()` / `contains()` and `remove()` is $O(1)$

In [ ]:
```
public void insert( AnyType x ){
    // first find the list
    List<AnyType> whichList = theLists[ myhash( x ) ];
    // if that list does not have that element
    if( !whichList.contains( x ) )
    {
        // just add it at the end
        whichList.add( x );

            // Rehash if loadFactor is greater than 1; see Section 5.5
        if( ++currentSize > theLists.length )
            rehash( );
    }
}
```

In [ ]:
```
public boolean contains( AnyType x ){
    // first get that position's linked list
    List<AnyType> whichList = theLists[ myhash( x ) ]; // this myhash function actually also MOD by tableSize
    return whichList.contains( x );
}
```

Probing

In general, we will have $h_i(x) = (hash(x) + f(i))\% \, tableSize$, where $h)i(x)$ is the i^{th} **probing function**, starting with $i = 0$

The same probing will be used for `insert()` , `remove()` , `contains()`

For most probing implementations, we keep $\lambda < 0.5$

- for **Quadratic Probing**, see proof on **Lecture15**
  - you basically start with the idea that:
    - $h_i(x)$ and $h_j(x)$ are distinct (so that, for example. h_7 probe does not wrap around and equals to h_1 probe) if:

$$TS = prime$$
$$\lambda \leq 0.5$$

- **Linear Probing**
  - where we have $f(i) = i$
  - where we would encounter **Primary Clustering**, which means you have nearly every element shifted in the array
- **Quadratic Probing**
  - where we have $f(i) = i^2$
    - this gets us away from **Primary Clustering**, as we spread out more
    - however, there is a constraint: to guarantee that an item is always *insertable*, we need $\lambda < 0.5$, and the tableSize has to be a *prime (>2)*
      - otherwise, it is *possible* that your probing gets you stuck at an infinite cycle

- **Lazy Deletion for** `remove()`
  - this is used quite often in implementations. We cannot directly delete the element in probing, because it breaks the probing scheme
  - so **lazy deletion** works, but need to care that:
    - `contains()` changes slightly as we need to check the **boolean deleted** as well
    - `insert()` encounters the greatest problem. As you still could encounter the problem of *duplicate* in later position while you have an empty spot here
      - as a reult, the only time you can be assured to **overwrite** that spot would be inserting the **same element at that spot** (the most inexpensive operation). So this would cause a *waste a space*

In [ ]:
```
// quadratic probing implementation
private void rehash( ){
    HashEntry<AnyType> [ ] oldArray = array;

        // Create a new double-sized, empty table
    allocateArray( 2 * oldArray.length );
    occupied = 0;
    theSize = 0;

        // Copy table over, REINSERTING EVERYTHING
    for( HashEntry<AnyType> entry : oldArray )
        if( entry != null && entry.isActive ) // skipping the lazy deleted one as well
            insert( entry.element );
}
```

In [ ]:
```
private int findPos( AnyType x )
{
    // actually stores the relative location, for more efficiency
    int offset = 1;
    // the first step
    int currentPos = myhash( x );

    // if either of these two conditions are true, continue moving
    while( array[ currentPos ] != null &&
            !array[ currentPos ].element.equals( x ) )
    {
        currentPos += offset;  // Compute ith probe
        offset += 2; // the DIFFERENCE between the currentPosition will always be 2 less than the next Probe positi
on
        // this is just to wrap around back to the array's beginning
        if( currentPos >= array.length )
            currentPos -= array.length;
    }

    // whereever it stops, it must be either an emptyPosition or you found it
    return currentPos;
}
```

In [ ]:
```
public boolean insert( AnyType x ){
    // Insert x as active
    int currentPos = findPos( x ); // first find that position
    // if it is active (not lazy deleted), means it is duplicate, do nothing
    if( isActive( currentPos ) )
        return false;

    // only when that place is actually empty, you increase occupied
    if( array[ currentPos ] == null )
        ++occupied;
    // now, no matter what case it is (empty or inActive), you make that element ot be active
    array[ currentPos ] = new HashEntry<>( x, true );
    theSize++;

        // Rehash, since it is quadratic probing, rehash when half the size; see Section 5.5
    if( occupied > array.length / 2 )
        rehash( );

    return true;
}
```

In [ ]:
```
// lazy deletion
public boolean remove( AnyType x ){
    int currentPos = findPos( x );
    if( isActive( currentPos ) )
    {
        // Lazy Deletion
        array[ currentPos ].isActive = false;
        theSize--;
        return true;
    }
    else
        return false;
}
```

HashMap

Very similar to **dictionaries** in Python, where there contains **key** and **value** pairs. In a **HashMap**:

- **key** are unique and cannot be duplicates
- **value** can be duplicates
- look up operations only work for **key**, not on **value**
  - therefore, **hash** functions are only applied to **key**

PriorityQueue/ Heap

dequeuing things based on some assigned **priority number**

- note that **PriorityQueue** are not sets. They can all have the same **priority number**.
- here, we just assume that the elements inserted are implementing **Comparable**

BinaryHeap

a **MinHeap**, because we are deleting Min

- one way we could implement this is using a **BinaryTree** (not search tree, because that is a set)
  - we need this tree to be **Complete** (filled from left to right)
    - Note that a **Complete** binary tree is **Balanced**
  - and we need to maintain a way to move *back up the tree*
    - which is done by the array
  - we need to hold the **heap order** condition at every **node**
    - **heap order**: each **node** must be less than or equal to its children
    - this means that the minmum will always be the **root node**

Array Based BinaryHeap

- build using a **level order traversal** of the tree
- item **0** will be left empty intentionally

Properties of this Implementation

- i will be now the index of the node
- $2 \times i$ will be the **left child** of the i^{th} node
- $2 \times i + 1$ will be the **right child** of the i^{th} node
- **size** will always be the index of the last node
  - therefore, you could use it to detect whether if a node is a **leaf node** by looking at if it has a **left child** by doing the calculation above and comparing it with **size**
- $\frac{i}{2}$ will be the parent of a node (in **Java** integer division is floored)
  - the **root node** will always have $\frac{1}{2} = 0$
- there are no gaps inside the array, because this tree is **complete**

Methods

- `buildHeap(Array arr)`
  - one way to implement this with $O(N)$ is to start at the **last non-leaf child**, which would be at $\frac{length}{2}$, and then `percholateDown()`
  - then after this level is completed, we move up a level and `perchoalteDown()`
  - Notice now, everytime we do percholate down, we only swap for *height of the current parent node* times. Therefore, the total number of swaps/operations we did was the $\sum heights = O(N)$

- `deleteMin()`
  - deleting is fast, but we need to maintain the **heap order**
  - this is done by letting the **last element** to be the new root
    - so that this is still **complete**
  - and then we `percholateDown()` all the way
  - this will be $O(log(N))$

- `insert()`
  - first put the element at the **last position** of the array, so that the **complete** condition is fulfilled
  - then we `perchoalteUp()` until the **heap order** is satisfied
  - this will be $O(log(N))$

- `percholateUp()`
  - this is actually not implemented specificially, because it is only used when we use `buildHeap()`
  - this is also simple. Simply swap with **parent** if it is smaller.

- `percholateDown()`
  - if it is smaller than both children, then **break**
  - if it is smaller than one child, then swap with that
  - if it is smaller than both, swap with the smaller child
  - then it continues

In [ ]:
```
private void buildHeap( ){
    // just walk backwards, starting from the last/rightmost interior child and move left
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

In [ ]:
```
public void insert( AnyType x ){
    if( currentSize == array.length - 1 )
        enlargeArray( array.length * 2 + 1 );

    // Percolate up, start with last spot
    int hole = ++currentSize;
    // note position 0 is temporary holder. This way of setting up also stops the loop correctly
    // and hole/2 is the parent of the hole
    for( array[ 0 ] = x; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        // swaping the parent value down, so the HOLE percholates up
        array[ hole ] = array[ hole / 2 ];
    // after the loop, the stopping position of hole is where you insert
    array[ hole ] = x;
}
```

In [ ]:
```
public AnyType deleteMin( ){
    if( isEmpty( ) )
        throw new UnderflowException( );

    AnyType minItem = findMin( );
    // filling the root with the last item
    array[ 1 ] = array[ currentSize-- ];
    // then re-order it using percholate down
    percolateDown( 1 );

    return minItem;
}
```

In [ ]:
```
private void percolateDown( int hole ) // hole is the position where we have the wrong thing
{
    int child; // it will either be the index of the only child, or the smaller of the two
    AnyType tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child ) // hole*2 > currentSize means it is a leaf
                                                   // we also set the hole=child at the end of every loop as well
    {
        // first set the child to be the left children
        child = hole * 2;
        if( child != currentSize && //if child is not the last element
                array[ child + 1 ].compareTo( array[ child ] ) < 0 ) // and if right child is smaller
            child++;
        // now, child is guaranteed to be the smaller of the two children
        if( array[ child ].compareTo( tmp ) < 0 ) //now you either swap or not swap/break out of the loop
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```

Comparison Based Sorting

Usually $O(N^2)$

- **Selection Sort**
- **Insertion Sort**

Both keeps one part of the array being sorted, and the other part unprocessed

SelectionSort

$O(N^2)$ in all cases

- **Algorithm**
  - looks for the smallest element in the unsorted array
  - insert that element to the **last place** of the sorted part
  - continues

InsertionSort

The best case is that we are given an already sorted array.

- in this case, we only check forward, as they are sorted, and no swaps are done
- therefore, we have $O(N)$

The worst case is that we are given a reverse ordered array

- every element needs to swapped to its maximum effect
- so we get $1+2+3+..+N-1=O(N^2)$

- **Algorithm**
  - continously insert the next/first element from the unsorted array into the sorted part
  - **swap** with left if it is smaller
  - continues

In general, **insertion sorts** will be preferred for small amount of data as compared to other $O(Nlog(N))$ recursive algorithms, which in general has quite a large constant factor attached in front

In [ ]:
```
public static <AnyType extends Comparable<? super AnyType>> void insertionSort( AnyType [ ] a ){
    int j;

    for( int p = 1; p < a.length; p++ )
    {
        // storing the element that needs to be compared
        AnyType tmp = a[ p ];
        for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            // if need to swap, move the wrong element to the right
            a[ j ] = a[ j - 1 ];
        // if not, then place the value there
        a[ j ] = tmp;
    }
}
```

Divide and Conquer Sorting

In general these would be $O(Nlog(N))$

- **MergeSort**
- **QuickSort**

MergeSort

- **Algorithm**
  - first, the base case would be that you have an array of size 1, and therefore nothing happens
  - then, invoke **mergeSort()** on the first half of the array **a**, and another on the other half, assuming that the algorithm works

    ```
    int[] a1 = mergeSort(1st half of a);
    int[] a2 = mergeSort(2nd half of a);
    ```

  - finally, we need to merge the two sorted array at *linear time*
    - which is done by having **two pointers** moving at *different speed*

      ```
      return merge(a1,a2);
      ```

In [ ]:
```
private static <AnyType extends Comparable<? super AnyType>> void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
                                                                             int left, int right ){
    // otherwise we have size 1 or 0
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}

private static <AnyType extends Comparable<? super AnyType>> void merge( AnyType [ ] a, AnyType [ ] tmpArray, int l
eftPos,
                                                                         int rightPos, int rightEnd ){
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}
```

QuickSort

Worst case degrades to $O(N^2)$

- this happens when the **pivot** is always the min/max, so that *partitioning gives pretty much only 1 array* instead of 2

Average case it is $O(Nlog(N))$

- **Algorithm**
  - pick a **pivot**
    - if we use **MedianOfThree pre-partition**, then at this stage the smaller will go position 0, largest to $length$, pivot to $length - 1$
  - **partition/order** the array into two parts, one part smaller than the **pivot**, the other larger than the **pivot**
  - recursively deal with the two partitioned parts, until less than 3 elements are met

- **Parition Algorithm**
  - have **two pointers at two ends**
  - increment i until it is at the wrong position
  - increment j until it is at the wrong position
  - if i crossed j, **break**
  - if not, **swap**, and continue the increment/decrement

In [ ]:
```
private static <AnyType extends Comparable<? super AnyType>> AnyType median3( AnyType [ ] a, int left, int right ){
    int center = ( left + right ) / 2;

    // it not only finds the median, but also pre-sort the three elements
    // notice this is the same idea for InsertionSort
    if( a[ center ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, center );
    if( a[ right ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, right );
    if( a[ right ].compareTo( a[ center ] ) < 0 )
        swapReferences( a, center, right );

    // Place pivot at position right - 1
    // remember the figure we drew, so that we have
    // the smallest on the left most
    // the median at right-1
    // the largest at right
    swapReferences( a, center, right - 1 );

    // so that pivot is out-of-place at right-1, whcih is restored later
    return a[ right - 1 ];
}
```

In [ ]:
```
private static <AnyType extends Comparable<? super AnyType>> void quicksort( AnyType [ ] a, int left, int right )
{
    // so that the sub-array we need to deal with is larger than 3
    if( left + CUTOFF <= right )
    {
        AnyType pivot = median3( a, left, right );

        // Begin partitioning
        // though this seems wrong, but in the loop, we did ++i and --j, so its the same
        int i = left, j = right - 1;
        for( ; ; )
        {
            // move i until it is at the wrong position
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            // move j until it is at the wrong position
            while( a[ --j ].compareTo( pivot ) > 0 ) { }

            // if i did not cross j, we swap and continue the loop
            if( i < j )
                swapReferences( a, i, j );
            // otherwise, we break and finish
            else
                break;
        }

        swapReferences( a, i, right - 1 );   // Restore pivot

        // then we just recursively sort the rest
        quicksort( a, left, i - 1 );    // Sort small elements
        quicksort( a, i + 1, right );   // Sort large elements
    }
    else  // Do an insertion sort on the subarray
        insertionSort( a, left, right );
}
```

Graph

A set of edges/links connecting a set of vertices

Definitions and Terminologies

- a **sparse graph** is when the number of **edges** is *upperbounded* by the number of **vertices**
  - so that $|E| = O(|V|)$
- a **dense graph** is when the number of **edges** is *upperbounded* by the number of **vertices** squared
  - so that $|E| = O(|V|^2)$ (the case when a **vertex** has an **edge** connected to *every other* **vertex**, mathematically it will be $(|V| - 1)^2$)
  - so that every **vertex** has one **edge** to any other **vertex**

- a **directed graph**, or **digraph**, is defined as **Graph** with only **directed edges**
- an **undirected graph** is defined to as **Graph** with only **undirected edges**
- we will not be using **hybrid graphs**

- a **path** would be the sequence of nodes/**vertices** that we follow to go from one **vertex** to another
  - a **simple path** is when each **vertex** in the sequence is distinct, in other words, no **cycles**
- the **length** of the path would be the number of **edges** traversed
  - for example, 3 **edges** connecting 4 **vertices** have a length of path of 3

- a **loop** is having an **edge** directly connecting to itself
  - fore example: $V_1 - V_1$
- a **cycle** is when after visting other **vertices** in a **path**, the same **vertex** gets visited again
  - for example: $V_1 - V_3 - V_2 - V_1$

- an **acylic graph** is when the **Graph** does not have any **cycles** represent
  - basically when each **vertex** is linked only once
  - for example, a **Tree** would be **acyclic**
- a **DAG**, or a **directed acyclic graph** is when you satisfy both a **directed graph** and an **acyclic graph**
  - this is quite often encountered so there is an abbreviation given

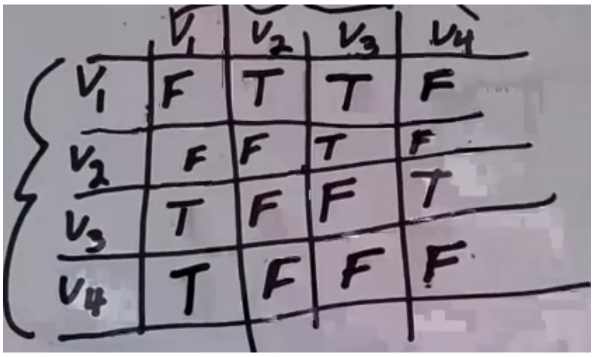
- a **connected graph**, is a **graph** has each **vertex** reachable from *every other* **vertex**
  - this is most easily achieved for an **undirected graph**
- a **strongly connected graph** would be a **connected graph** but also **directed**
- a **weakly connected graph** would be strictly speaking **disconnected** *if we take into account the direction ONLY*
  - however, if we ignore the directions, a **weakly connected graph** would be a **connected graph**

Examples of Graphs

- the **Trees**, especially the **BinarySearch Tree**, is a **directed graph**
  - we can go down from **parent** to **child**, but not going up
- the **BinaryHeap** is an **undirected graph**
  - because we can easily go up and down using the indices in the array

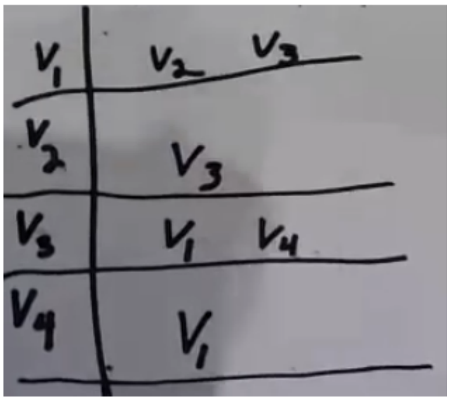
Representations of a Graph

- **Adjacency Matrices**
  - a space complexity of $O(N^2)$
  - we have one side as **source vertex**, the other as **destination vertex**



	V1	V2	V3	V4
V1	F	T	T	F
V2	F	F	T	F
V3	T	F	F	T
V4	T	F	F	F

- **Adjacency List**
- this will have a better spacial complexity $|V| + |E|$
  - therefore, even in the case of **dense graph**, we have $|V| + |V|^2 = O(|V|^2)$
- where each **vertex** has a **LinkedList** storing the **vertices** that it can reach
- using the previous example, it looks like this
  - notice that the number of elements in each **LinkedList** corresponds to the number of **edges**
  - therefore, the sum of the sizes of the **LinkedLists** are $|E|$
  - however, since the leading node is the **vertex** itself, we also have $|V|$



Topological Sort

Having a **directed graph** (**DAG**, actually), and needs a way to **visit all vertices**. *Do not have to follow the sequence on the graph*

Terminolgies and Definitions

- **indegree**
  - the number of **edges** that terminates at a **vertex**
- in fact, this sorting only works for **DAG**s, because
  - if you have a **cyclic graph**, then you cannot find a starting point to a class

- **Algorithm**
  - start at a **vertex with indegree 0**
  - then update all **vertices** it can reach with `indegree--`
  - then visit the **next vertex with indegree 0**
  - continues until all vertices are visited

- **Cost Analysis**
  - If using **Queue**
    - we will have $O(|E| + |V|)$
    - for **dense graph**, it will stil be $O(|V|^2)$, but
    - for **sparse graph**, it takes only $O(|V|)$

In [ ]:
```
void improvedTopSort() throws CycleFoundException{
    Queue<Vertex> q = new Queue<Vertex>();
    int counter = 0;

    // this still takes |V|
    for each Vertex v:
        if(v.indegree == 0){
            q.enqueue(v)
        }

    while(!q.isEmpty()){
        // automatically gets a vertex of 0 indegree
        // reduces the cost to be constant time here
        Vertex v = q.dequeue();
        // this tells us the sequence of visit
        v.topNum = ++counter;

        // this part is still the same, but we add a vertex of indegree of 0 to the queue
        // this takes |E| in total, the same
        for each Vertex w adjacent to v:
            if (--w.indegree == 0){
                q.enqueue(w);
            }
    }

    if (counter != NUM_VERTICIES){
            throw new CycleFoundException();
    }

}
```

Single Source Unweighted Shortes Path

Basically uses a breath-first approach to go through all vertices in the **adjacency list**, kind of similar to **Dijkstra's Algorithm**. Once finished, we also know the shortes path pairs within the same path.

- **Cost Analysis**
  - if using **Queue**
    - $O(|V| + |E|)$ since it is breadth-first
      - so for **Dense Graph**, it will be $O(|V| + |V|^2)$ = $O(|V|^2)$
      - for a **Sparce Graph**, it will be $O(|V| + |V|)$ = $O(|V|)$
- **Algorithm**
  - initialization: marks everything as **unvisited**, and set each **Dv** field to max. Then set the source **Dv** to 0 and **visted**
    - same as **Dijkstra's Algorithm**
  - visit **every vertex** it can reach, and update those **Pv** fields and **Dv** fields incrementally
    - notice, here every distance will be the same
  - then `push()` those **vertices** into a **Queue**
  - `pop()` and continues the loop

- **Data Chart**
  - we will use this for retrieving the shortest path once computation is finished

Vertex	Known/Visted	Dv/Distance	Pv/Previous
V_1	True	1	V_3
V_2	True	2	V_1
V_3	True	0	N/A
V_4	True	2	V_1
V_5	True	3	V_2
V_6	True	1	V_3
V_7	True	3	V_4

In [ ]:
```
void unweighted(Vertex s){
    Queue<Vertex> q = new Queue<Vertex>();

    for each Vertex v{
        v.dist = INFINITY;
    }

    s.dist = 0;
    q.enqueue(s);

    // ends smarter than the case before

    // this will happen |V| times
    while(!q.isEmpty()){
        // this happens constantly
        Vertex v = q.dequeue();

        // this happens exactly |E| times
        for each Vertex w adjacent to v{
            // not enqueuing things that have already visited
            if(w.dist == INFINITY){
                w.dist = v.dist+1;
                w.pv = v;
                // level order traversal
                q.enqueue(w);
            }
        }
    }
}
```

Single-Source Weighed Shortest Path / Dijkstra's Algorithm

In this case for a **weighted graph**, **Greedy Algorithm** actually works, if the **weights** are non-negative

- **Algorithm**
  - same initializatio process as for unweighted graph
  - vistit each reachable **vertices** and update the field
    - update **Dv** only when it is smaller
    - but not update **visted**, yet
  - then we take the **greedy step** of marking the **unvisited vertex with current smallest Dv** to be **visited**
  - continues with that **vertex**

- **Cost Analysis**
  - if we used linear scanning
    - this will have a cost of $O(|V|^2 + |E|)$
  - if we used a **Priority Queue** for finding the **smallest vertex**
    - The total cost is $O(|V|log(|V|) + |E|log(|V|)) = O(|E|log(|V|))$, because $|E|$ is the only "variable"
      - this means in a **sparce graph**, we would have an improvement for $O(|V|log(|V|))$
      - but in a **dense graph**, we would have $O(|V|^2 log(|V|))$
    - this is because:
      - the loop for search gives $O(|V|log(|V|))$, but comes with a cost
      - now, the inner loop is **not** $O(|E|)$, because if we just reassigned the value of a **vertex**, it might disturb the **heap order**. This means we need to call another **percholate up**, which takes $O(log(|V|))$
      - this means the that loop itself takes $O(|E|log(|V|))$

- **Data Chart**
  - the same as the one for **unweighted graph**

Vertex	Known	Dv/Distance	Pv/Previous
V_1	T	0	N/A
V_2	T	2	V_1
V_3	T	3	V_4
V_4	T	1	V_1
V_5	T	3	V_4
V_6	T	6	V_7
V_7	T	5	V_4

In [ ]:
```
void dijkstra(Vertex s){
    for each Vertex v{
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    // so this happens |V| times
    while(there is an unknown vertex){
        // if you find the min by scanning through the list/linear scanning
        // then this is also |V|
        Vertex v = smallest known vertex

        v.known = true;

        // this loop goes IN TOTAL |E| times
        for each Vertex w adjacent to v{
            if(!w.known){
                // the next step/edge's cost
                DistType cvn = cost of edge from v to w;

                // if we did find something smaller
                if(v.dist + cvn < w.dist){
                    // update w
                    // in this version, w.dist = v.dist+cvw works the same

                    // this is useful, because there is actually ANOTHER way of doing this
                    decrease(w.dist to v.dist+cvn);
                    w.pv = v;
                }
            }
        }
    }
}
```

Minimum Spannig Tree (MST)

For an **undirected graph**, produce an **acyclic tree** that is the subset of the graph that spans all of the **Vertices** (Spanning), and it needs to have a minimum sum in terms of the edges included (minimum)

In general, there are two algorithms that we can use. In fact, both are **Greedy Algorithms**

- **Prim's algorithm**
- **Kruskal's algorithm**

Prim's Algorithm

this is pretty much the same as **Dijkstra's Algorithm**, but since we are constructing a tree, the difference is

- **known** means whether if we have **included that vertex into the tree**
- **Dv** means the current smallest distance we currently know to bring that vertex into the graph (**not cumulative**)
- **Pv** vertex that achieves the shortest distance in the **Dv** field

Vertex	Known	Dv	Pv
V_1	T	0	N/A
V_2	F	∞	-
V_3	F	∞	-
V_4	F	∞	-
V_5	F	∞	-
V_6	F	∞	-
V_7	F	∞	-

Kruskal's Algorithm

- instead of looking at **vertices**, it looks at **edges**
- and realize that, if you have N vertices, and you have $N-1$ edges without creating a **cycle**, you must have completed the graph

- **Algorithm**
  - first we need to put **every edge** into a **PriorityQueue**
  - then, we call **deleteMin()** to pick out the current **minimum edge**
  - now, we take the **Greedy Step** so that that connection must be the solution
  - now, before we incorporate the **edge**, we need to check if there is a **cycle**
  - if there is a cycle, **reject the edge** and continue
  - if not, **add the edge** to ouput and continue
  - **breaks** when there is $N-1$ edges in the list

- **Cycle Detection Algorithm**
  - to solve this problem, we need to use **Disjoined Set**
    - if any two or more **vertices** live in the same set, then it means that here exists a **path** that connects them
    - therefore, if we want to further add an **edge** that connects two **vertices** in the same set, then we know we created a **cycle**
  - in general, you have two operatios to use on a disjoined set
    - **find()**, which looks up that **vertex** and see which set it belongs to
      - this will be used to see whether **two vertices are in the same set**
    - **union()**, which takes two set and merge them into one set
      - this **unions two sets** if they are disconencted

- **Cost Analysis**
  - the worst case analysis gives the algorithm performs $O(|E|log(|E|))$
    - in the **dense graph**, we would have $O(|E|log(|E|)) = O(|V|^2log(|V|^2)) = O(2|V|^2log(|V|)) = O(|V|^2log(|V|))$
    - in the **sparce graph**, we would have $O(|V|log(|V|))$
  - So in general, we would have $O(|E|log(|V|))$, which is the same RunTime for the **Dijkstra's Algorithm**, and hence the same time for **Prim's Algorithm**

In [ ]:
```
// we return a list of Edges which we could use to construct the Minimum Spanning Tree
ArrayList<Edge> kruskal(List<Edge> edges, int numVertices){

    DisjoinedSets ds = new DisjoinedSets(numVertices);
    PriorityQueue<Edge> pq = new PriorityQueue<>(edges);
    List<Edge> mst = new ArrayList<>();

    // once we have N-1 edges, we finish
    // in the worst case, this could happen |E| times
    while(mst.size()!= numVertices -1){
        // notice that this is O(log(|E|))
        // and the rest of the code here is actually constant time
        Edge e = pq.deleteMin();

        // since edge has two vertices, e= (u,v), we need get both sets out
        // by using the method ds.find()
        SetType uset = ds.find(e.getu());
        SetType vset = ds.find(e.getv());

        // if uset is equal to vset, then it means there is a cycle as they are in the same set
        if(uset != vset){
            // Accept the edge
            mst.add(e);
            ds.union(uset,vset);
        }
    }

}
```

P vs NP

- **nondeterministic polynomial time**, are the set of problems that can be solved in a linear time if there exists a kind of **Oracle**, which tells you exactly how to solve a problem
  - or, you can say that these are the problems whose solution can be **verified** *in polynomial time*
- **a polynomial problem** means you can definitely solve it in a polynomial time
  - therefore, it also means that you can **verifty** in polynomial time as well

Terminologies

- **NP-complete** is a subset of **NP** problems, that are the hardest problems in **NP**, **such that**
  - if you can solve a **NP-complete** problem, then you have solved every **NP** problem as well
  - therefore, it means that if you solved a **NP-Complete** problem in Linear Time, then you **solved** (not verify) every **NP** problem, and hence the equivalence can be proven **P = NP**
- **NP-hard**, is technically not a **NP problem**, as often we are not sure if we can even **verify** in polynomial time. However, they do resemble a similar for to **NP**, and they are harder than **NP-complete**.

Example NP-Complete Problem

A problem that can be **verified** at polynomial time, but may not solved at **Polynomial Time**

- consider the **Traveling Salesman Problem (TSP)** (this is the **NP-Complete** version)
  - and you have a **Complete Weighted Graph**, namely you have an **undirected edge** from every vertext to every other **vertex**
- the goal is that you need to find a **simple cycle** that visits all the **vertices**
  - a **simple cycle** means that you can only visit each **vertex** exactly once, except for the **starting vertex** which you have to go back to
  - **AND** we need to have the length of the total path satisfying $length < k$, where k is an arbitrary value that I can specify
  - however, notice that this version does not require a minimal cost!

Now, if you try to solve the problem, you will in some sense have to evaluate every possible path, before determining whether a path satisfying the requirement exists or determining the minimum path

However, if I give you an **oracle** that tells you a solution path, then I can **verify** it in polynomial time by simply comparing it with k.

- however, notice that this would **not work** if you have the version that the task is to find the **shortes path/optimal path**, then it might not even by an **NP** at all. This type of problems are known as **NP-hard**