reading 2.1

Above, we said that there are 8 primitive types: byte, short, int, long, float, double, boolean, char. Everything else, including arrays, is not a primitive type but rather a `reference type`.

When we *declare* a variable of any reference type (Walrus, Dog, Planet, array, etc.), Java allocates a box of 64 bits, no matter what type of object.

Java does not write anything into the reserved box when a variable is declared. In other words, there are no default values. As a result, the Java compiler prevents you from using a variable until after the box has been filled with bits using the `=` operator. For this reason, I have avoided showing any bits in the boxes in the figure above.

When we *instantiate* an Object using `new` (e.g. Dog, Walrus, Planet), Java first allocates a box for each instance variable of the class, and fills them with a default value. The constructor then usually (but not always) fills every box with some other value.

However, this problem is easily resolved with the following piece of information: the 64 bit box contains not the data about the walrus, but instead the address of the Walrus in memory.

```
Walrus someWalrus;
someWalrus = new Walrus(1000, 8.3);
```

The first line creates a box of 64 bits. The second line creates a new Walrus, and the address is returned by the `new` operator. These bits are then copied into the `someWalrus` box according to the GRoE.

It turns out our Mystery has a simple solution: When you write `y = x`, you are telling the Java interpreter to copy the bits from x into y. This Golden Rule of Equals (GRoE) is the root of all truth

reading 2.3

A generic `DLList` that can hold any type would look as below:

```
public class DLList<BleepBlorp> {
    private IntNode sentinel;
    private int size;

    public class IntNode {
        public IntNode prev;
        public BleepBlorp item;
        public IntNode next;
        ...
    }
    ...
}
```

Now that we've defined a generic version of the `DLList` class, we must also use a special syntax to instantiate this class. To do so, we put the desired type inside of angle brackets during declaration, and also use empty angle brackets during instantiation. For example:

```
DLList<String> d2 = new DLList<>("hello");
d2.addLast("world");
```

reading 4.1

overload override (diffence)

```
public interface List61B<Item> {
    public void addFirst(Item x);
    public void add Last(Item y);
    public Item getFirst();
    public Item getLast();
    public Item removeLast();
    public Item get(int i);
    public void insert(Item x, int position);
    public int size();
}
```

If we define this method in List61B

```
default public void print() {
    for (int i = 0; i < size(); i += 1) {
        System.out.print(get(i) + " ");
    }
    System.out.println();
}
```

```
public class AList<Item> implements List61B<Item>{...}
```

```
List61B<String> lst = new SLList<String>();
```

In the above declaration and instantiation, lst is of type "List61B". This is called the "static type"

However, the objects themselves have types as well. the object that lst points to is of type SLList. Although this object is intrinsically an SLList (since it was declared as such), it is also a List61B, because of the "is-a" relationship we explored earlier. But, because the object itself was instantiated using the SLList constructor, We call this its "dynamic type".

Aside: the name "dynamic type" is actually quite semantic in its origin! Should lst be reassigned to point to an object of another type, say a AList object, lst's dynamic type would now be AList and not SLList! It's dynamic because it changes based on the type of the object it's currently referring to.

When Java runs a method that is overriden, it searches for the appropriate method signature in it's **dynamic type** and runs it.

reading 4.2

```
public class VengefulSLList<Item> extends SLList<Item> {
    SLList<Item> deletedItems;

    public VengefulSLList() {
        deletedItems = new SLList<Item>();
    }

    @Override
    public Item removeLast() {
        Item x = super.removeLast();
        deletedItems.addLast(x);
        return x;
    }

    /** Prints deleted items. */
    public void printLostItems() {
        deletedItems.print();
    }
}
```

By using the `extends` keyword, subclasses inherit all **members** of the parent class. "Members" includes:

- All instance and static variables
- All methods
- All nested classes

Note that constructors are not inherited, and private members cannot be directly accessed by subclasses.

While constructors are not inherited, Java requires that all constructors **must start with a call to one of its superclass's constructors**.

Thus, we can either explicitly make a call to the superclass's constructor, using the `super` keyword:

```
public VengefulSLList() {
    super();
    deletedItems = new SLList<Item>();
}
```

Or, if we choose not to, Java will automatically make a call to the superclass's *no-argument* constructor for us.

```
public static Dog maxDog(Dog d1, Dog d2) { ... }
```

Casting

```
Poodle frank  = new Poodle("Frank", 5);
Poodle frankJr = new Poodle("Frank Jr.", 15);
```

```
Dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr);
```

reading 4.3

```
public interface Comparable<T> {
     public int compareTo(T obj);
}
```

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

```
public class Dog implements Comparable<Dog> {
    ...
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }
```

```
    private static class NameComparator implements Comparator<Dog> {
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }

    public static Comparator<Dog> getNameComparator() {
        return new NameComparator();
    }
}
```

```
Comparator<Dog> nc = Dog.getNameComparator();
```

reading 4.4

We've seen interfaces that can do a lot of cool things! They allow you to take advantage of interface inheritance and implementation inheritance. As a refresher, these are the qualities of interfaces:

- All methods must be public.
- All variables must be public static final.
- Cannot be instantiated
- All methods are by default abstract unless specified to be `default`
- Can implement more than one interface per class

The `final` keyword is a keyword for variables that prevents the variable from being changed after its first assignment. For example, consider the `Date` class below:

We will now introduce a new class that lies somewhere in between interfaces and concrete classes: the abstract class. Below are the characteristics of abstract classes:

- Methods can be public or private
- Can have any types of variables
- Cannot be instantiated
- Methods are by default concrete unless specified to be `abstract`
- Can only implement one per class

reading 6.3

```
Set<String> s = new HashSet<>();
...
for (String city : s) {
    ...
}
```

The above code translates to:

```
Set<String> s = new HashSet<>();
...
Iterator<String> seer = s.iterator();
while (seer.hasNext()) {
    String city = seer.next();
    ...
}
```

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

```
public class ArraySet<T> implements Iterable<T> {
    ...
private class ArraySetIterator implements Iterator<T>
```

```
public Iterator<T> iterator() {
    return new ArraySetIterator();
}
```

reading 6.4

`equals()` and `==` have different behaviors in Java. `==` Checks if two objects are actually the same object in memory. Remember, pass-by-value! `==` checks if two boxes hold the same thing. For primitives, this means checking if the values are equal. For objects, this means checking if the address/pointer is equal.

reading 5.3 generic methods

```
public static <K,V> V get(Map61B<K,V> map, K key) {
    if map.containsKey(key) {
        return map.get(key);
    }
    return null;
}
```

Here's an example of how to call it:

```
ArrayMap<Integer, String> isMap = new ArrayMap<Integer, String>();
System.out.println(mapHelper.get(isMap, 5));
```

```
public static <K extends Comparable<K>, V> K maxKey(Map61B<K, V> map) {
    List<K> keylist = map.keys();
    K largest = map.get(0);
    for (K k: keylist) {
        if (k.compareTo(largest)) {
            largest = k;
        }
    }
    return largest;
}
```

reading 7.2

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
	Y	Y	N	N
private	Y	N	N	N

reading 8.2

Suppose we have a function R(N) with order of growth f(N).

$R(N) \in \Theta(f(N))$ means that there exists positive constants k_1, k_2 such that:
$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$, for all values of N greater than some N_0 (a very large N).

Similarly, here's the formal definition of Big O: $R(N) \in O(f(N))$ means that there exists positive constants k_2 such that:
$R(N) \leq k_2 \cdot f(N)$ for all values of N greater than some N_0 (a very large N).

reading 8.3 (runtime analysis)

1.Loop
2.Recursion
3.Binary Search O(logN)
4.Merge Sort O(NlogN)

reading 8.4

Answer: This time, the runtime depends on not only the length of the input, but also the array's contents. In the best case, $R(N) \in \Theta(1)$. If the input array contains all of the same element, then no matter how long it is, dup4 will return on the first iteration.

On the other hand, in the worst case, $R(N) \in \Theta(N^2)$. If the array has no duplicates, then dup4 will never return early, and the nested for loop will result in quadratic runtime.

This exercise highlights one limitation of Big Theta. Big Theta expresses the exact order of as a function of the input size. However, if the runtime depends on more than just the *size* of the input, then we must qualify our statements into different cases before using Big Theta. Big O does away with this annoyance. Rather than having to describe both the best and worse case, for the example above, we can simply say that the runtime of dup4 is $O(N^2)$. Sometimes dup4 is faster, but it's at worst quadratic.

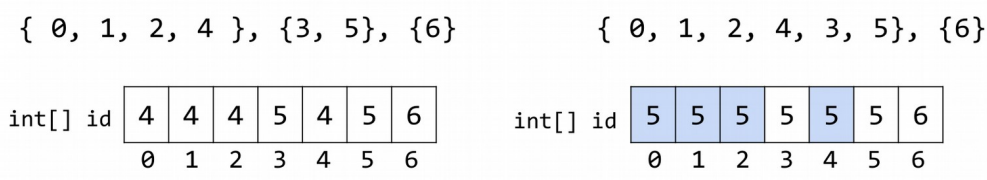
Big O is still a useful idea:

- Allows us to make simple blanket statements, e.g. can just say “binary search is O(log N)” instead of “binary search is Θ(log N) in the worst case”.

Note: Big O is NOT the same as "worst case". But it is often used as such.

lecture14

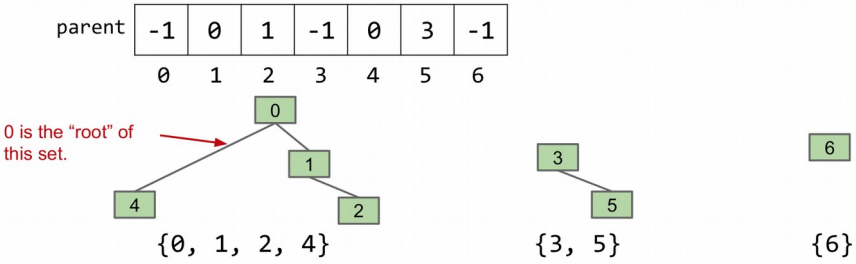
QuickFindDS: just use a array of integers, where ith entry gives set number (a.k.a. “id”) of item i.



QuickUnion: Assign each item a parent (instead of an id). Results in a tree-like shape.

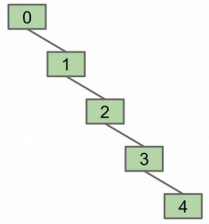
connect(5, 2)

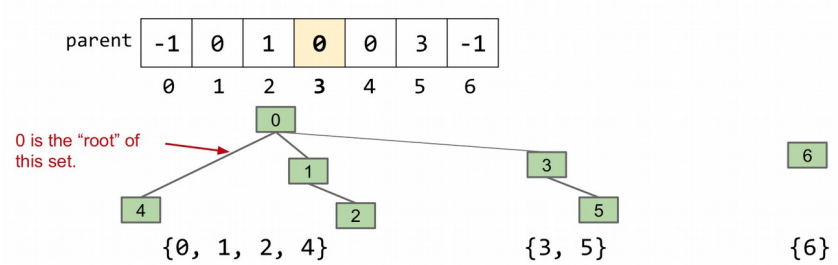
- Find root(5). // returns 3
- Find root(2). // returns 0
- Set root(5)’s value equal to root(2).



The Worst Case:If we always connect the first item’s tree below the second item’s tree, we canend up with a tree of height M after M operations:

connect(4, 3)
connect(3, 2)
connect(2, 1)
connect(1, 0)





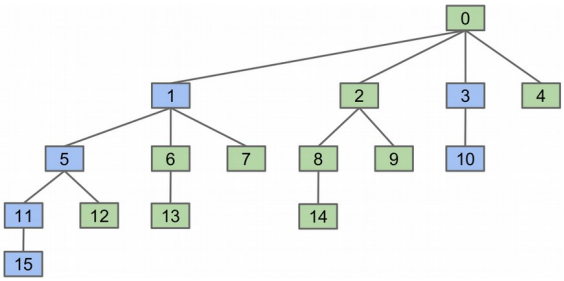
For N items, what’s the worst case runtime…

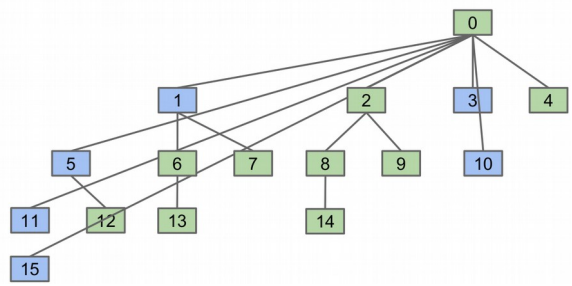
- For connect(p, q)? Θ(N)
- For isConnected(p, q)? Θ(N)

Weighted QuickUnion: 1.Track tree size (number of elements). 2.Always link root of smaller tree to larger tree.

Implementation	Constructor	connect	isConnected
QuickFindDS	Θ(N)	Θ(N)	Θ(1)
QuickUnionDS	Θ(N)	O(N)	O(N)
WeightedQuickUnionDS	Θ(N)	O(log N)	O(log N)

Path Compression: When we do isConnected(15, 10), tie all nodes seen to the root.





reading 10.2

Trees are composed of:

- nodes
- edges that connect those nodes.
  - **Constraint**: there is only one path between any two nodes.

In some trees, we select a **root** node which is a node that has no parents.

A tree also has **leaves**, which are nodes with no children.

- **Binary Trees**: in addition to the above requirements, also hold the binary property constraint. That is, each node has either 0, 1, or 2 children.
- **Binary Search Trees**: in addition to all of the above requirements, also hold the property that For every node X in the tree:
  - Every key in the left subtree is less than X's key.
  - Every key in the right subtree is greater than X's key. **Remember this property!! We will reference it a lot throughout the duration of this module and 61B.

Search

If our tree is relatively "bushy", the find operation will run in $\log(n)$ time because the height of the tree is logn, that's pretty fast!

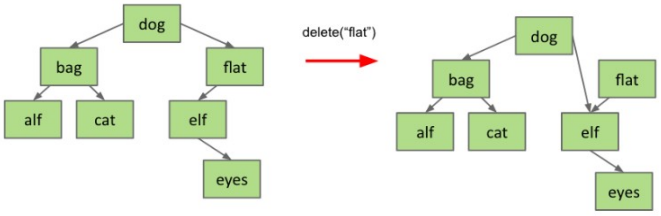
Insert

BSTs have best case height Θ(log N), and worst case height Θ(N).

Delete

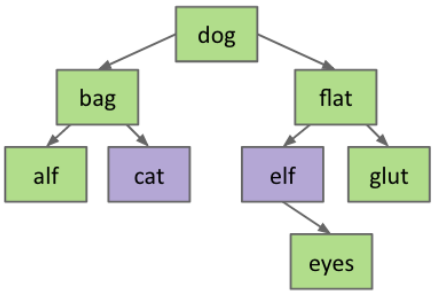
1.No child

2.One Child



3.Two Child

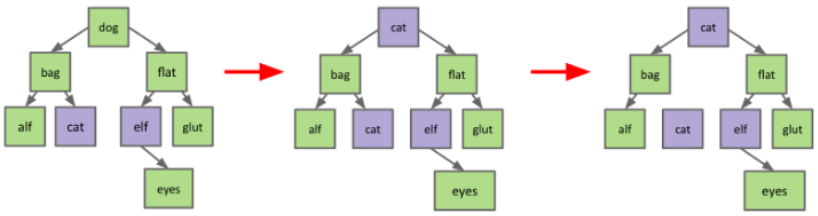
In the below tree, we show which nodes would satisfy these requirements given that we are trying to delete the `dog` node.



To find these nodes, you can just take the right-most node in the left subtree or the left-most node in the right subtree.

Then, we replace the `dog` node with either `cat` or `elf` and then remove the old `cat` or `elf` node.

This is called **Hibbard deletion**, and it gloriously maintains the BST property amidst a deletion.



lecture 17

B-trees / 2-3 trees / 2-3-4 trees

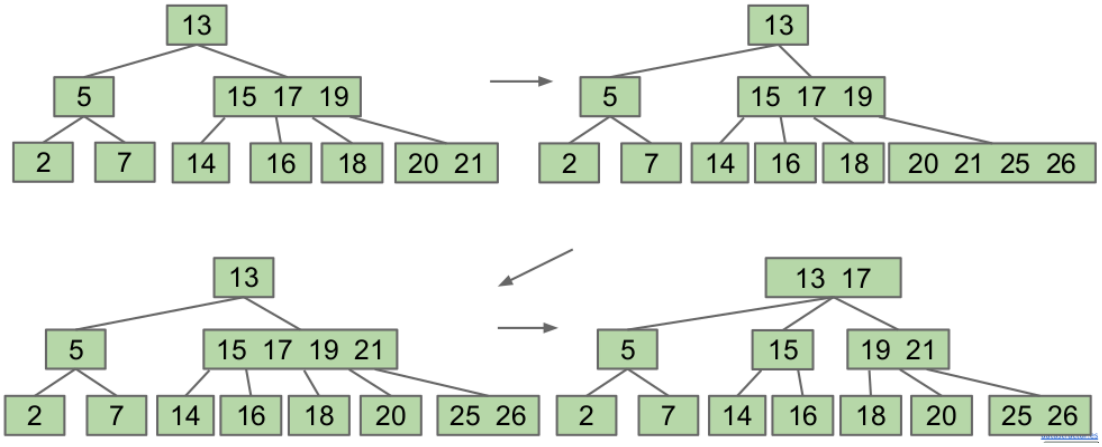
Observation: Splitting-trees have perfect balance.

- If we split the root, every node gets pushed down by exactly one level.
- If we split a leaf node or internal node, the height doesn't change.

All leaves must be the same distance from the source.

A non-leaf node with k items must have exactly k+1 children.

- Suppose we add 25, 26:



Overall height is therefore Θ(log N).

runtime is therefore O(log N)

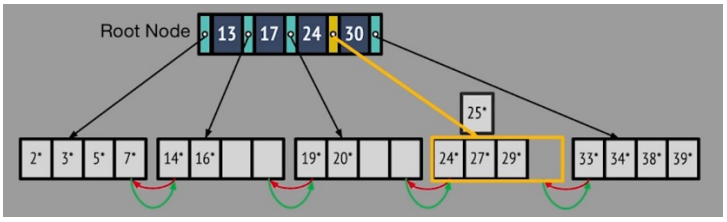
In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in **B+ tree**, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

https://www.youtube.com/watch?v=xeyE8tiVzbw

Each interior node is at least partially full:

- **d <= #entries <= 2d**
- **d: order of the tree (max fan-out = 2d + 1)**

Inserting 25* into a B+ Tree

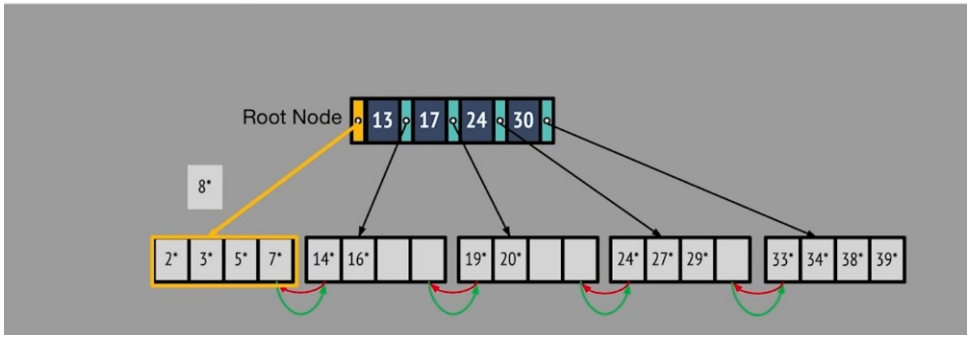


Find the correct leaf

If there is room in the leaf just add the entry

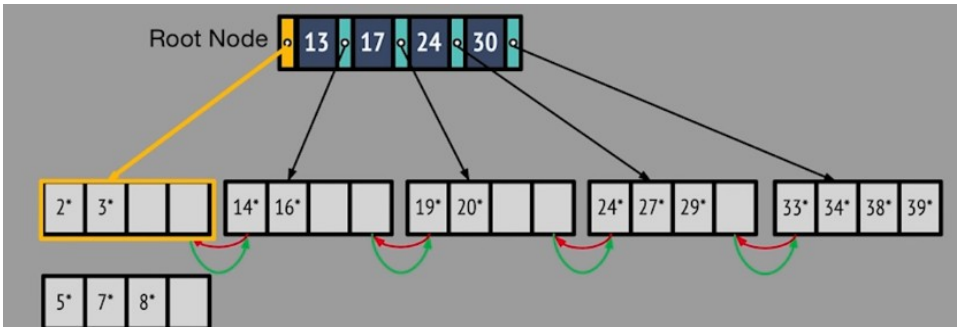
- Sort the leaf page by key

Inserting 8* into a B+ Tree: Insert

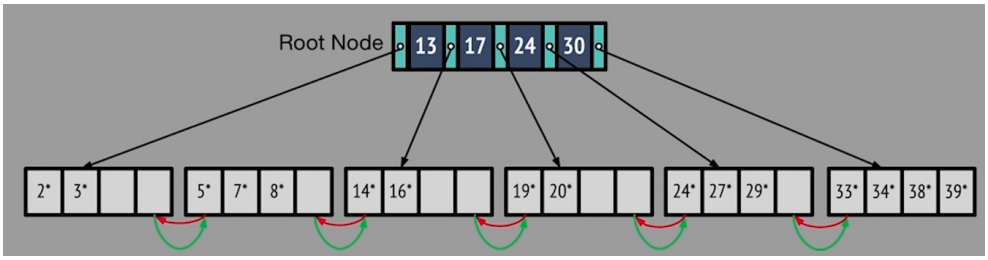


Find the correct leaf

- Split leaf if there is not enough room
- Redistribute entries evenly



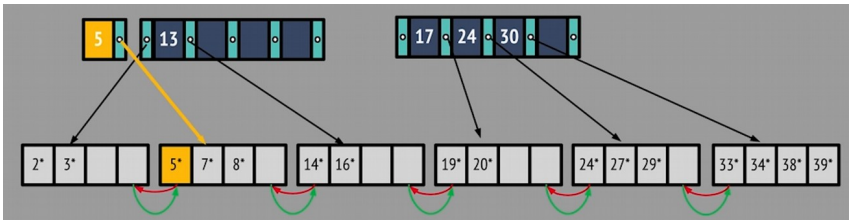
Fix next/prev pointers



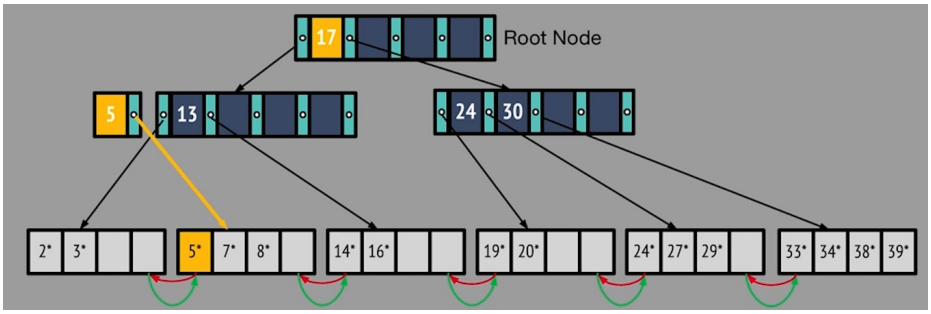
Copy up from leaf the middle key

No room in parent? Recursively split index nodes

- Redistribute the rightmost d keys

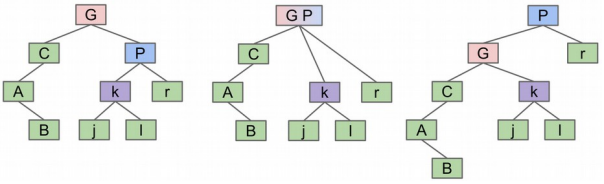


Push up middle key



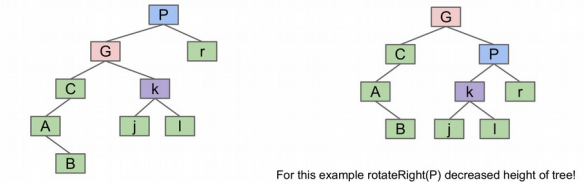
rotateLeft(G): Let x be the right child of G. Make G the **new left child** of x.

- Can think of as temporarily merging G and P, then sending G down and **left**.
- Preserves search tree property. No change to semantics of tree.



rotateRight(P): Let x be the left child of P. Make P the **new right child** of x.

- Can think of as temporarily merging G and P, then sending P down and **right**.
- Note: k was G's right child. Now it is P's left child.



For this example rotateRight(P) decreased height of tree!

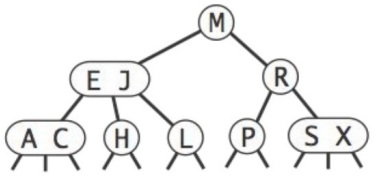
LLRB

A BST with left glue links that represents a 2-3 tree is often called a "Left Leaning Red Black Binary Search Tree" or LLRB.

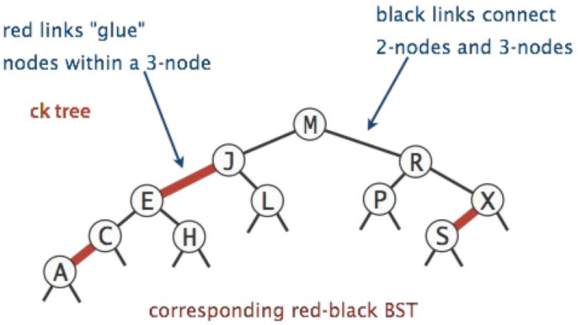
- LLRBs are normal BSTs!
- There is a 1-1 correspondence between an LLRB and an equivalent 2-3 tree.

Some handy LLRB properties:

- No node has two red links [otherwise it'd be analogous to a 4 node, which are disallowed in 2-3 trees].
- Every path from root to a leaf has same number of **black links** [because 2-3 trees have the same number of links to every leaf]. LLRBs are therefore balanced.



2-3 tree



corresponding red-black BST

LLRB Construction

One last important question: Where do LLRBs come from?

- Would not make sense to build a 2-3 tree, then convert. Even more complex.
- Instead, it turns out we implement an LLRB insert as follows:
  - Insert as usual into a BST.
  - Use zero or more rotations to maintain the 1-1 mapping.

Congratulations, you just invented the red-black BST.

- When inserting: Use a red link.
- If there is a *right leaning "3-node"*, we have a **Left Leaning Violation.**
  - Rotate left the appropriate node to fix.
- If there are *two consecutive left links*, we have an **Incorrect 4 Node Violation.**
  - Rotate right the appropriate node to fix.
- If there are any *nodes with two red children*, we have a **Temporary 4 Node.**
  - Color flip the node to emulate the split operation.

LLRB tree has height O(log N).

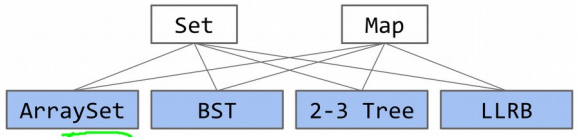
Contains is trivially O(log N).

Insert is O(log N).

- O(log N) to add the new node.
- O(log N) rotation and color flip operations per insert.

lecture 19

We've now seen several implementations of the Set (or Map) ADT.



Worst case runtimes

	contains(x)	add(x)	Notes
ArraySet	Θ(N)	Θ(N)	
BST	Θ(N)	Θ(N)	Random trees are Θ(log N).
2-3 Tree	Θ(log N)	Θ(log N)	Beautiful idea. Very hard to implement.
LLRB	Θ(log N)	Θ(log N)	Maintains bijection with 2-3 tree. Hard to implement.

Uniqueness

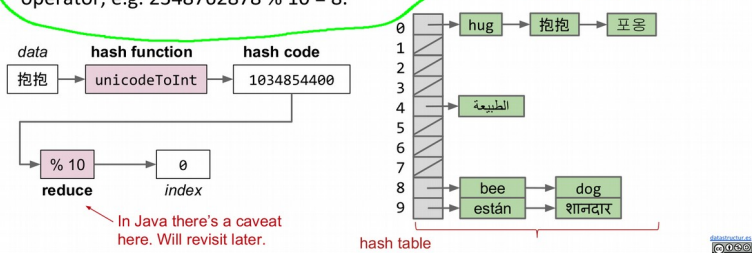
- $cat_{27} = (3 \times 27^2) + (1 \times 27^1) + (20 \times 27^0) = 2234_{10}$
- $bee_{27} = (2 \times 27^2) + (5 \times 27^1) + (5 \times 27^0) = 1598_{10}$

As long as we pick a base ≥ 26, this algorithm is guaranteed to give each lowercase English word a unique number!

The Hash Table

What we've just created here is called a **hash table**.

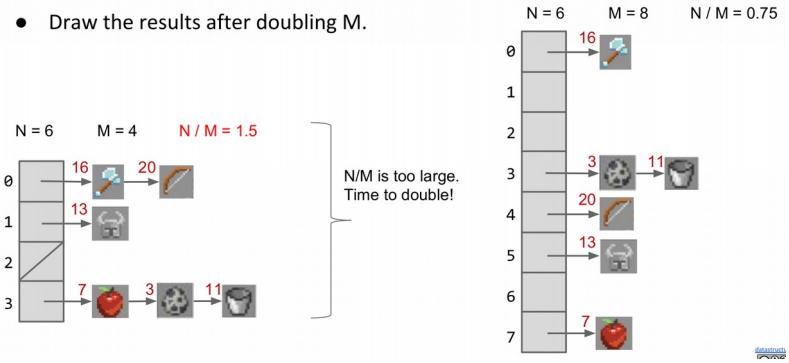
- *Data* is converted by a **hash function** into an integer representation called a **hash code**.
- The **hash code** is then **reduced** to a valid *index*, usually using the modulus operator, e.g. 2348762878 % 10 = 8.



Hash Table Resizing Example

When N/M is ≥ 1.5, then double M.

- Draw the results after doubling M.



Worst case runtimes

	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Hash Table With No Resizing	Θ(N)	Θ(N)
… With Resizing	Θ(1)†	Θ(1)*†

*: Indicates "on average".
†: Assuming items are evenly spread.

Warning #1: Never store objects that can change in a `HashSet` or `HashMap`!

- If an object's variables changes, then its hashCode changes. May result in items getting lost.

Warning #2: Never override `equals` without also overriding `hashCode`.

- Can also lead to items getting lost and generally weird behavior.
- HashMaps and HashSets use equals to determine if an item exists in a particular bucket.
- See HW3 or the study guide problems.

Can think of multiplying data by powers of some base as ensuring that all the data gets scrambled together into a seemingly random integer.

Example hashCode Function

The Java 8 hash code for strings. Two major differences from our hash codes:

- Represents strings as a base 31 number.
  - Why such a small base? Real hash codes don't care about uniqueness.
- Stores (caches) calculated hash code so future `hashCode` calls are faster.

```
@Override
public int hashCode() {
    int h = cachedHashValue;
    if (h == 0 && this.length() > 0) {
        for (int i = 0; i < this.length(); i++) {
            h = 31 * h + this.charAt(i);
        }
        cachedHashValue = h;
    }
    return h;
}
```

Example: Choosing a Base

Java's `hashCode()` function for Strings:

- $h(s) = s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \ldots + s_{n-1}$

Our `asciiToInt` function for Strings:

- $h(s) = s_0 \times 126^{n-1} + s_1 \times 126^{n-2} + \ldots + s_{n-1}$

Which is better?

- Might seem like 126 is better. Ignoring overflow, this ensures a unique numerical representation for all ASCII strings.
- … but overflow is a particularly bad problem for base 126!

Example: Base 126

Major collision problem:

- "geocronite is the best thing on the earth.".hashCode() yields 634199182.
- "flan is the best thing on the earth.".hashCode() yields 634199182.
- "treachery is the best thing on the earth.".hashCode() yields 634199182.
- "Brazil is the best thing on the earth.".hashCode() yields 634199182.

Any string that ends in the same last 32 characters has the same hash code.

- Why? Because of overflow.
- Basic issue is that 126^32 = 126^33 = 126^34 = ... 0.
  - Thus upper characters are all multiplied by zero.
  - See CS61C for more.

2,147,483,647 + 1 +
2,147,483,648

Typical Base

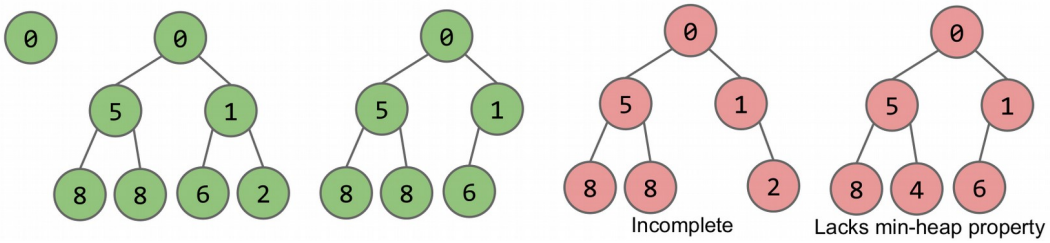
A typical hash code base is a small prime.

- Why prime?
  - Never even: Avoids the overflow issue on previous slide.
  - Lower chance of resulting hashCode having a bad relationship with the number of buckets: See study guide problems and hw3.
- Why small?
  - Lower cost to compute.

lecture 20

Binary min-heap: Binary tree that is ***complete*** and obeys ***min-heap property***.

- Min-heap: Every node is less than or equal to both of its children.
- Complete: Missing items only at the bottom level (if any), all nodes are as far left as possible.



Heaps lend themselves very naturally to implementation of a priority queue.

Given a heap, how do we implement PQ operations?

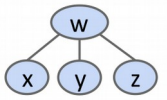
- `getSmallest()` - return the item in the root node.
- `add(x)` - place the new employee in the last position, and promote as high as possible.
- `removeSmallest()` - assassinate the president (of the company), promote the rightmost person in the company to president. Then demote repeatedly, always taking the 'better' successor.

See https://goo.gl/wBKdFQ for an animated demo.

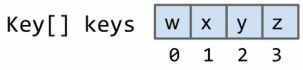
How do we Represent a Tree in Java?

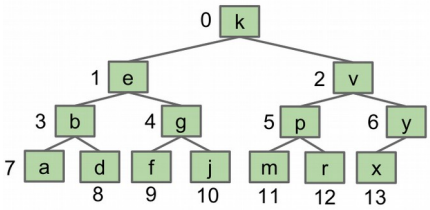
Approach 3: Store keys in an array. Don't store structure anywhere.

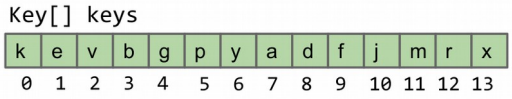
- To interpret array: Simply assume tree is complete.
- Obviously only works for "complete" trees.




```
public class Tree3<Key> {
   Key[] keys;
   ...
```







	Heap
add	Θ(log N)
getSmallest	Θ(1)
removeSmallest	Θ(log N)

Suppose we know that our keys are always strings.

- Can use a special data structure known as a Trie.
- Basic idea: Store each letter of the string as a node in a tree.

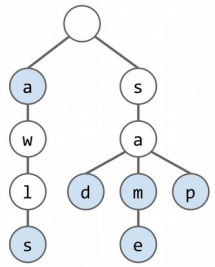
Tries: Search Hits and Misses

Suppose we insert “sam”, “sad”, “sap”, “same”, “a”, and “awls”.

- contains(“sam”): true, blue node
- contains(“sa”): false, white node
- contains(“a”): true, blue node
- contains(“saq”): false, fell off tree

“hit”

“miss”



Two ways to have a search “miss”:

- If the final node is white.
- If we fall off the tree, e.g. contains(“sax”).

When our keys are strings, Tries give us slightly better performance on contains and add.

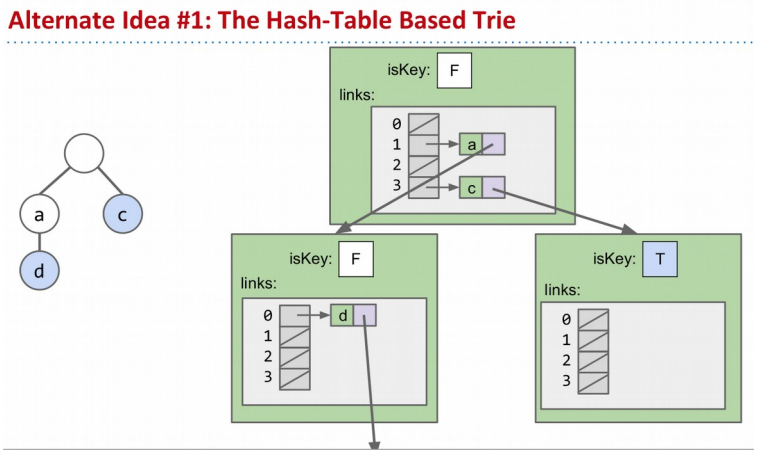
in terms of L, the length of the key:

Add: Θ(L)

Contains: O(L)

Trie Implementation

Alternate Idea #1: The Hash-Table Based Trie



Challenging Warmup Exercise: Collecting Trie Keys

Challenging Exercise: Give an algorithm for collecting all the keys in a Trie.

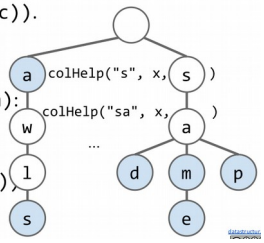
`collect()`:

- Create an empty list of results x.
- For character c in `root.next.keys()`:
  - Call `colHelp(“c”, x, root.next.get(c))`.
- Return x.

`colHelp(String s, List<String> x, Node n)`:

- If `n.isKey`, then `x.add(s)`.
- For character c in `n.next.keys()`:
  - Call `colHelp(s + c, x, n.next.get(c))`.

```
x = ["a", "awls", "sad",
     "sam", "same", "sap"]
```



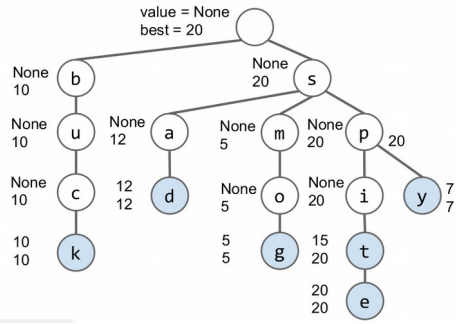
A More Efficient Autocomplete

One way to address this issue:

- Each node stores its own value, as well as the value of its best substring.

Search will consider nodes in order of “best”.

- Consider ‘sp’ before ‘sm’.
- Can stop when top 3 matches are all better than best remaining.



lecture 22 K-d Tree 没看

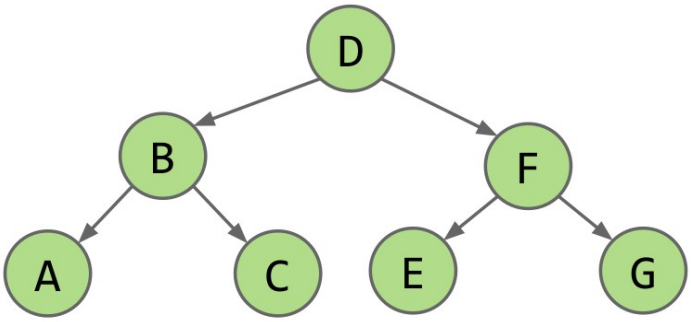
lecture 23

Depth First Traversals

Preorder traversal: “Visit” a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG

Postorder traversal: Traverse left, traverse right, then visit: ACBEGFD



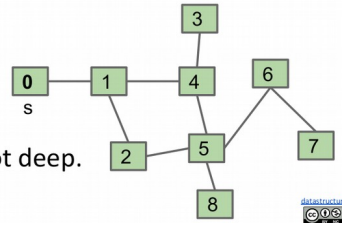
Graph Definition

A simple graph is a graph with:

- Note, all trees are graphs!
- No edges that connect a vertex to itself, i.e. no “loops”.
- No two edges that connect the same vertices, i.e. no “parallel edges”.

So too are there many graph traversals, given some source:

- DFS Preorder: 012543678 (dfs calls).
- DFS Postorder: 347685210 (dfs returns).
- BFS order: Act in order of distance from s.
  - BFS stands for “breadth first search”.
  - Analogous to “level order”. Search is wide, not deep.
  - 0 1 24 53 68 7



visit each vertex at most once.

Marking nodes prevents multiple visits.

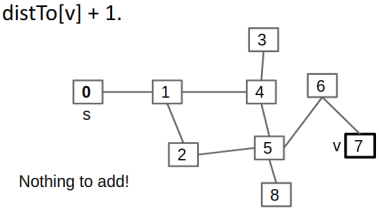
lecture 24

BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- **Initialize the fringe (a queue with a starting vertex s) and mark that vertex.**
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

#	marked	edgeTo	distTo
0	T	-	0
1	T	0	1
2	T	1	2
3	T	4	3
4	T	1	2
5	T	2	3
6	T	5	4
7	T	6	5
8	T	5	4



Nothing to add!

Queue: []

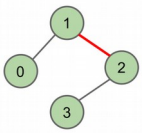
Graph Representations

Graph Representation 1: Adjacency Matrix.

- G.adj(2) would return an iterator where we can call next() up to two times
  - next() returns 1
  - next() returns 3
- Total runtime to iterate over all neighbors of v is Θ(V).
  - Underlying code has to iterate through entire array to handle next() and hasNext() calls.

G.adj(2) returns an iterator that will ultimately provide 1, then 3.

v \ w	0	1	2	3
0	0	1	0	0
1	1	0	**1**	0
2	0	**1**	0	1
3	0	0	1	0



Graph Printing Runtime: http://yellkey.com/allow

What is the order of growth of the running time of the print client from before if the graph uses an **adjacency-matrix** representation, where V is the number of vertices, and E is the total number of edges?

A. Θ(V)
B. Θ(V + E)
C. Θ(V^2)
D. Θ(V*E)

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

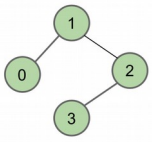
Runtime to iterate over v's neighbors?

- Θ(V).

How many vertices do we consider?

- V times.

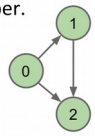
	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

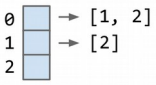


More Graph Representations

Representation 3: Adjacency lists.

- Common approach: Maintain array of lists indexed by vertex number.
- Most popular approach for representing graphs.





V is total number of vertices.

E is total number of edges in the entire graph.

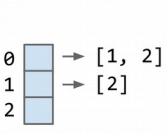
Graph Printing Runtime: http://yellkey.com/effort

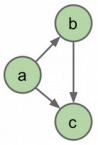
What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

A. Θ(V)
B. Θ(V + E)
C. Θ(V^2)
D. Θ(V*E)

```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

Best case: Θ(V) Worst case: Θ(V^2)





All cases: Θ(V + E)

- Create V iterators.
- Print E times.

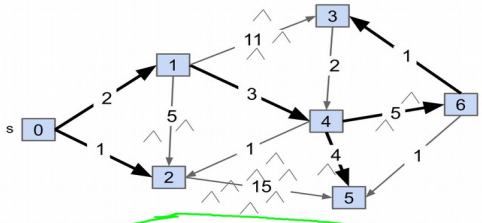
Problem	Problem Description	Solution	Efficiency (adj. list)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V+E) time Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V+E) time Θ(V) space

Problem	Problem Description	Solution	Efficiency (adj. matrix)
s-t paths	Find a path from s to every reachable vertex.	DepthFirstPaths.java Demo	O(V^2) time Θ(V) space
s-t shortest paths	Find a shortest path from s to every reachable vertex.	BreadthFirstPaths.java Demo	O(V^2) time Θ(V) space

lecture 25

Problem: Single Source Shortest Paths

Goal: Find the shortest paths from source vertex s to every other vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	2.0	0→1
2	1.0	0→1
3	11.0	6→3
4	5.0	1→4
5	9.0	4→5
6	10.0	4→6

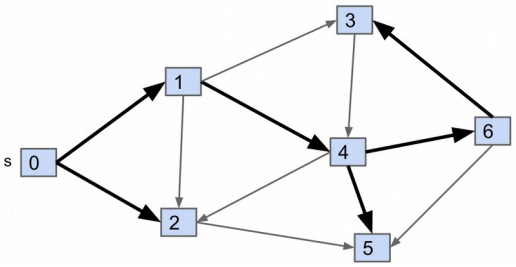
Shortest paths from s=0

Observation: Solution will always be a **tree**.

- Can think of as the union of the shortest paths to all vertices.

Observation: Solution will always be a path with no cycles (assuming non-negative weights).

If G is a connected edge-weighted graph with V vertices and E edges, how many edges are in the **Shortest Paths Tree** (SPT) of G? [assume every vertex is reachable]



V: 7
Number of edges in SPT is 6

Always V-1:

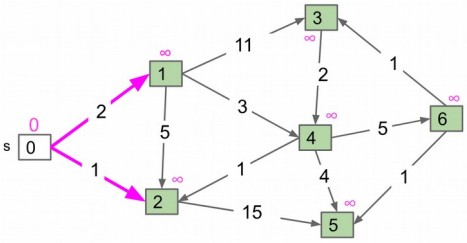
- For each vertex, there is exactly one input edge (except source).

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

#	distTo	edgeTo
0	0	-
1	∞	-
2	∞	-
3	∞	-
4	∞	-
5	∞	-
6	∞	-



Fringe: [(1: ∞), (2: ∞), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]

Dijkstra's is guaranteed to return a correct result if all edges are non-negative.

Dijkstra's Algorithm Pseudocode

Dijkstra's:

- PQ.add(source, 0)
- For other vertices v, PQ.add(v, infinity)
- While PQ is not empty:
  - p = PQ.removeSmallest()
  - Relax all edges from p

Relaxing an edge p → q with weight w:

- If distTo[p] + w < distTo[q]:
  - distTo[q] = distTo[p] + w
  - edgeTo[q] = p
  - PQ.changePriority(q, distTo[q])

Key invariants:

- edgeTo[v] is the best known predecessor of v.
- distTo[v] is the best known total distance from source to v.
- PQ contains all unvisited vertices in order of distTo.

Important properties:

- Always visits vertices in order of total distance from source.
- Relaxation always fails on edges to visited (white) vertices.

Dijkstra's Algorithm Runtime

Priority Queue operation count, assuming binary heap based PQ:

- add: V, each costing O(log V) time.
- removeSmallest: V, each costing O(log V) time.
- changePriority: E, each costing O(log V) time.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V) for a connected graph.

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ removeSmallest	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

Single Source, Single Target:

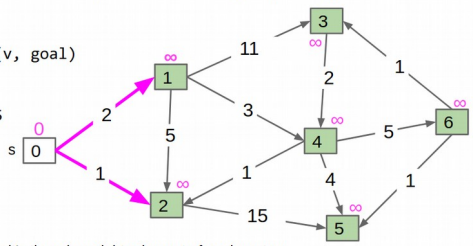
- Dijkstra's is inefficient (searches useless parts of the graph).

A* Demo, with s = 0, goal = 6.

Insert all vertices into fringe PQ, storing vertices in order of d(source, v) + h(v, goal).

Repeat: Remove best vertex v from PQ, and relax all edges pointing from v.

#	distTo	edgeTo	h(v, goal)
0	0	-	1
1	∞	-	3
2	∞	-	15
3	∞	-	1
4	∞	-	5
5	∞	-	∞
6	∞	-	0



h(v, goal) is arbitrary. In this example, it's the min weight edge out of each vertex.

Fringe: [(1: ∞), (2: ∞), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]

Heuristics and Correctness

For our version of A* to give the correct answer, our A* heuristic must be:

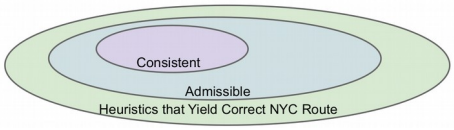
- **Admissible**: h(v, NYC) ≤ true distance from v to NYC.
- **Consistent**: For each neighbor of w:
  - h(v, NYC) ≤ dist(v, w) + h(w, NYC).
  - Where dist(v, w) is the weight of the edge from v to w.

Our heuristic was inadmissible and inconsistent.

- Inadmissible:

Admissibility and consistency are sufficient conditions for certain variants of A*.

- If heuristic is admissible, A* tree search yields the shortest path.
- If heuristic is consistent, A* graph search yields the shortest path.
- These conditions are sufficient, but not necessary.



Our version of A* is called "A* graph search". There's another version called "A* tree search". You'll learn about it in 188.

lecture 26

Spanning Trees

Given an **undirected** graph, a ***spanning tree*** T is a subgraph of G, where T:

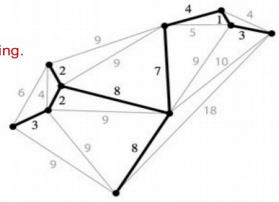
- Is connected.
- Is acyclic.

(These two properties make it a tree.)

- Includes all of the vertices. (This makes it spanning.)

Example:

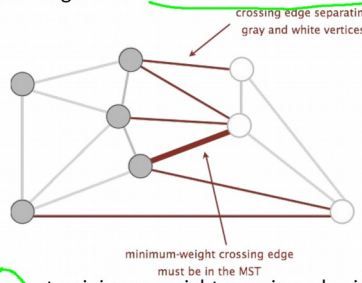
- Spanning tree is the black edges and vertices.



A ***minimum spanning tree*** is a spanning tree of minimum total weight.

A Useful Tool for Finding the MST: Cut Property

- A ***cut*** is an assignment of a graph's nodes to two non-empty sets.
- A ***crossing edge*** is an edge which connects a node from one set to a node from the other set.



Cut property: Given any cut, minimum weight crossing edge is in the MST.

- For rest of today, we'll assume edge weights are unique.

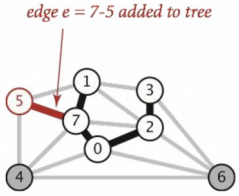
Prim's Algorithm

Start from some arbitrary start node.

- Repeatedly add shortest edge (mark black) that has one node inside the MST under construction.
- Repeat until V-1 edges.

Conceptual Prim's Algorithm Demo (Link)

Why does Prim's work? Special case of generic algorithm.



MST vs. SPT, http://yellkey.com/approach

A shortest paths tree depends on the start vertex:

- Because it tells you how to get from a source to EVERYTHING.

There is no source for a MST.

Nonetheless, the MST sometimes happens to be an SPT for a specific vertex.

Generic MST Finding Algorithm

Start with no edges in the MST.

- Find a cut that has no crossing edges in the MST.
- Add smallest crossing edge to the MST.
- Repeat until V-1 edges.

This should work, but we need some way of finding a cut with no crossing edges!

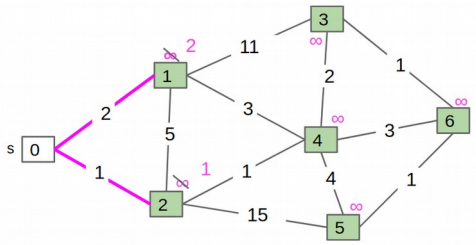
- Random isn't a very good idea.

Prim's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from tree.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

#	distTo	edgeTo
0	0	-
1	2	0
2	1	0
3	∞	-
4	∞	-
5	∞	-
6	∞	-



Fringe: [(2: 1), (1: 2), (3: ∞), (4: ∞), (5: ∞), (6: ∞)]

Prim's Algorithm Runtime

Priority Queue operation count, assuming binary heap based PQ:

- Insertion: V, each costing O(log V) time.
- Delete-min: V, each costing O(log V) time.
- Decrease priority: O(E), each costing O(log V) time.
  - Operation not discussed in lecture, but it was in lab 10.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V).

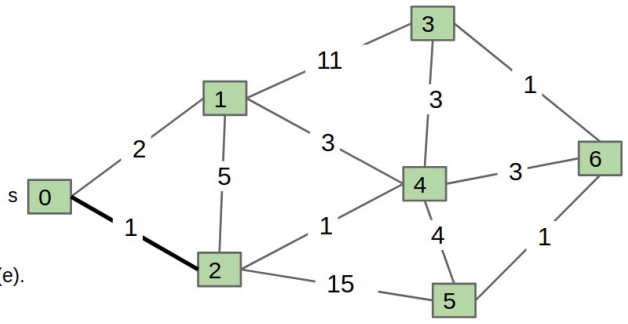
	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ delMin	V	O(log V)	O(V log V)
PQ decreasePriority	O(E)	O(log V)	O(E log V)

Kruskal's Demo

Insert all edges into PQ.

Repeat: Remove smallest weight edge. Add to MST if no cycle created.

```
Fringe: (2-4: 1),
(3-6: 1), (5-6: 1),
(0-1: 2), (4-1: 3),
(3-4: 3), (6-4: 3),
(4-5: 4), (1-2: 5),
(1-3: 11), (2-5: 15)
```



Removed edge: e=0-2

Cycle? No. union(0, 2). add(e).

```
WQU: [0-2]
MST: [0-2]
```

Kruskal's Runtime

Kruskal's algorithm on previous slide is O(E log E).

Fun fact: In HeapSort lecture, we discuss how do this step in O(E) time using "bottom-up heapification".

Operation	Number of Times	Time per Operation	Total Time
Insert	E	O(log E)	O(E log E)
Delete minimum	O(E)	O(log E)	O(E log E)
union	O(V)	O(log* V)	O(V log* V)
isConnected	O(E)	O(log* V)	O(E log*V)

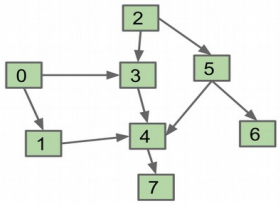
Note: If we use a pre-sorted list of edges (instead of a PQ), then we can simply iterate through the list in O(E) time, so overall runtime is O(E + V log*V + E log* V) = O(E log*V).

Shortest Paths and MST Algorithms Summary

Problem	Algorithm	Runtime (if E > V)	Notes
Shortest Paths	Dijkstra's	O(E log V)	Fails for negative weight edges.
MST	Prim's	O(E log V)	Analogous to Dijkstra's.
MST	Kruskal's	O(E log E)	Uses WQUPC.
MST	Kruskal's with pre-sorted edges	O(E log* V)	Uses WQUPC.

lecture 27

Topological Sort A topological sort only exists if the graph is a directed acyclic graph (DAG)



Start at 2:

- Go as far as you can get (DFS).
  - 2, 3, 4, 7, 5, 6

Suppose we have tasks 0 through 7, where an arrow from v to w indicates that v must happen before w.

- What algorithm do we use to find a valid ordering for these tasks?
- Valid orderings include: [0, 2, 1, 3, 5, 4, 7, 6], [2, 0, 3, 5, 1, 4, 6, 7], …

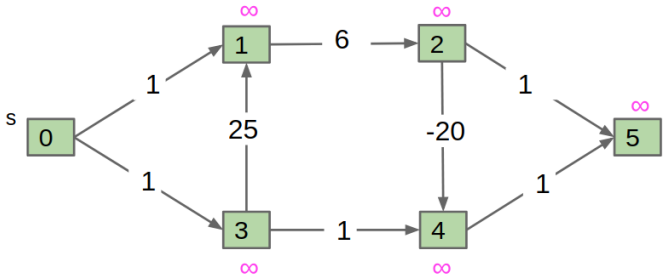
Perform a DFS traversal from every vertex with indegree 0, NOT clearing markings in between traversals.

- Record DFS postorder in a list: [7, 4, 1, 3, 0, 6, 5, 2]
- Topological ordering is given by the reverse of that list (reverse postorder):
  - [2, 5, 6, 0, 3, 1, 4, 7]

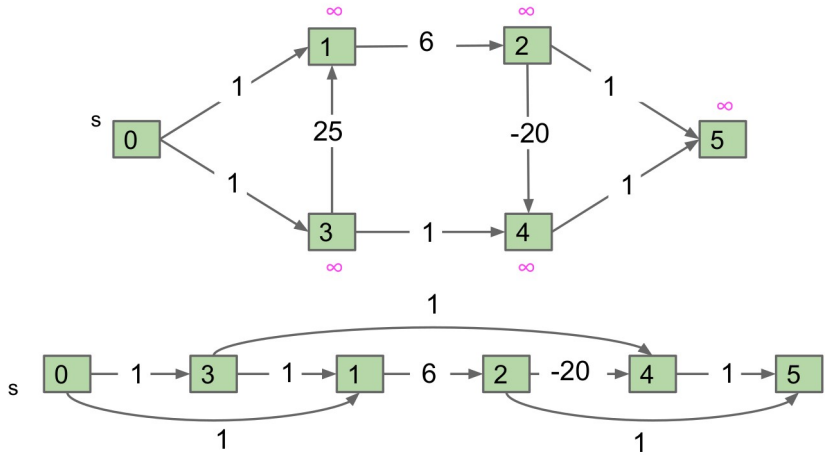
Problem	Problem Description	Solution	Efficiency
topological sort	Find an ordering of vertices that respects edges of our DAG.	Demo Code: Topological.java	O(V+E) time Θ(V) space
DAG shortest paths	Find a shortest paths tree on a DAG.	Demo Code: AcyclicSP.java	O(V+E) time Θ(V) space

Try to come up with an algorithm for shortest paths on a DAG that works even if there are negative edges.

- Hint: You should still use the "relax" operation as a basic building block.



First: We have to find a topological order, e.g. 031245. Runtime is O(V + E).



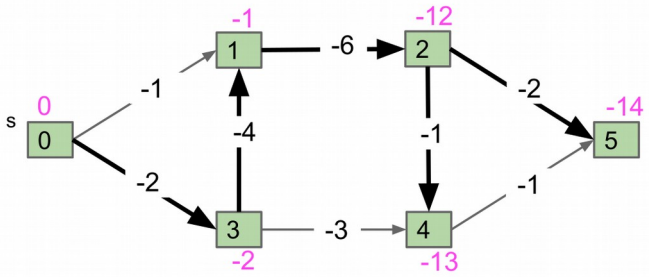
Second: We have to visit all the vertices in topological order, relaxing all edges as we go. Let's see a demo [Link].

- Runtime for step 2 is also O(V + E).

The Longest Paths Problem on DAGs

DAG LPT solution for graph G:

- Form a new copy of the graph G' with signs of all edge weights flipped.
- Run DAGSPT on G' yielding result X.
- Flip signs of all values in X.distTo. X.edgeTo is already correct.



longest paths	Find a longest paths tree on a graph.	No known efficient solution.	O(???) time O(???) space
DAG longest paths	Find a longest paths tree on a DAG.	Flip signs, run DAG SPT, flip signs again.	O(V+E) time Θ(V) space

lecture 29

Selection Sort

We've seen this already.

- Find smallest item.
- Swap this item to the front and 'fix' it.
- Repeat for unfixed items until all items are fixed.

Naive Heapsort: Leveraging a Max-Oriented Heap

Idea: Instead of rescanning entire array looking for minimum, maintain a heap so that getting the minimum is fast!

For reasons that will become clear soon, we'll use a max-oriented heap.

A min heap would work as well, but wouldn't be able to take advantage of the fancy trick in a few slides.

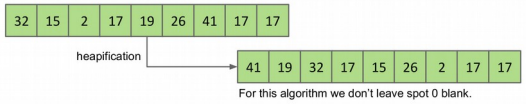
Naive heapsorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

In-place Heapsort

Alternate approach, treat input array as a heap!

- Rather than inserting into a new array of length N + 1, use a process known as "bottom-up heapification" to convert the array into a heap.
  - To bottom-up heapify, just sink nodes in reverse level order.
- Avoids need for extra copy of all data.
- Once heapified, algorithm is almost the same as naive heap sort.



For this algorithm we don't leave spot 0 blank.

In-place heap sort: Demo

Heapsort Runtime Analysis

Use the magic of the heap to sort our data.

- Getting items into the heap O(N log N) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Overall runtime is O(N log N) + Θ(N) + O(N log N) = **O(N log N)**

- Far better that selection sort!

Mergesort

We've seen this one before as well.

Mergesort: Demo

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Time complexity, analysis from previous lecture: Θ(N log N runtime)

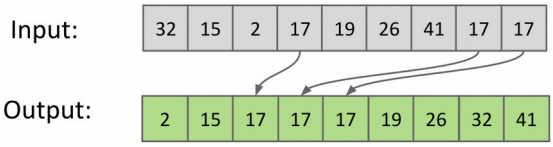
- Space complexity with aux array: Costs Θ(N) memory.

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output: Demo (Link)



In-place Insertion Sort

Two more examples.

7 swaps:

```
P O T A T O
P O T A T O  (0 swaps)
O P T A T O  (1 swap )
O P T A T O  (0 swaps)
A O P T T O  (3 swaps)
A O P T T O  (0 swaps)
A O O P T T  (3 swaps)
```

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E  (0 swaps)
O S R T E X A M P L E  (1 swap )
O R S T E X A M P L E  (1 swap )
O R S T E X A M P L E  (0 swaps)
E O R S T X A M P L E  (4 swaps)
E O R S T X A M P L E  (0 swaps)
A E O R S T X M P L E  (6 swaps)
A E M O R S T X P L E  (5 swaps)
A E M O P R S T X L E  (4 swaps)
A E L M O P R S T X E  (7 swaps)
A E E L M O P R S T X  (8 swaps)
```

Purple: Element that we're moving left (with swaps).
Black: Elements that got swapped with purple.
Grey: Not considered this iteration.

Insertion Sort Runtime

What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

You may recall Ω is not "best case".

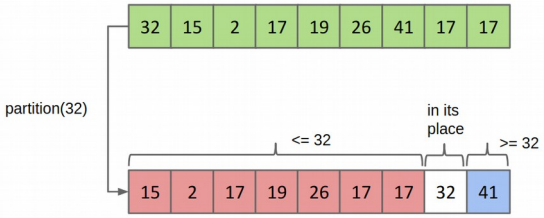
So technnnniically you could also say $\Omega(1)$

lecture 30

Partition Sort, a.k.a. Quicksort

Quicksorting N items: (Demo)

- Partition on leftmost item.
- Quicksort left half.
- Quicksort right half.



Quicksort Performance

Theoretical analysis:

- Best case: Θ(N log N)
- Worst case: Θ(N^2)
- **Randomly chosen array case: Θ(N log N) expected**

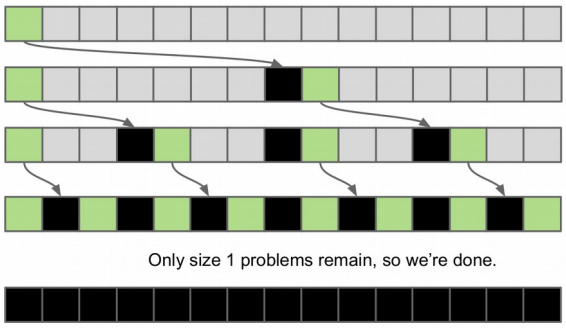
For our pivot/partitioning strategies: Sorted or close to sorted.

With extremely high probability!!

Compare this to Mergesort.

- Best case: Θ(N log N)
- Worst case: Θ(N log N)

Best Case: Pivot Always Lands in the Middle



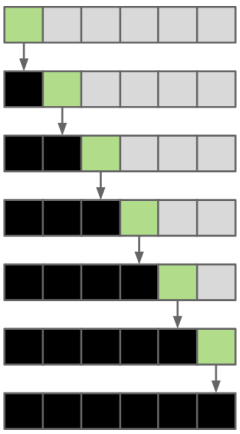
Worst Case: Pivot Always Lands at Beginning of Array

Give an example of an array that would follow the pattern to the right.

- 1 2 3 4 5 6

What is the runtime Θ(·)?

- N^2



Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)
Insertion	Θ(1)	Θ(N^2)	Θ(N) if almost sorted
Mergesort	Θ(N)	Θ(N log N)	
Quicksort	Θ(log N) (call stack)	Θ(N log N) expected	Fastest sort

If pivot always lands somewhere "good", Quicksort is Θ(N log N). However, the very rare Θ(N^2) cases do happen in practice, e.g.

- Bad ordering: Array already in sorted order (or almost sorted order).
- Bad elements: Array with all duplicates.

What can we do to avoid worst case behavior?

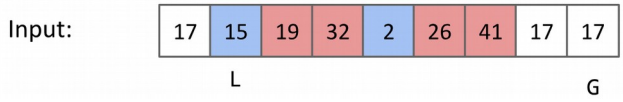
- Always use the median as the pivot -- this works.
- Randomly swap two indices occasionally.
  - Sporadic randomness. Maybe works?
- Shuffle before quicksorting.
  - This definitely works and is a harder core version of the above.
- Partition from the center of the array. Does not work, can still find bad cases.

lecture 32

Hoare Partitioning

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

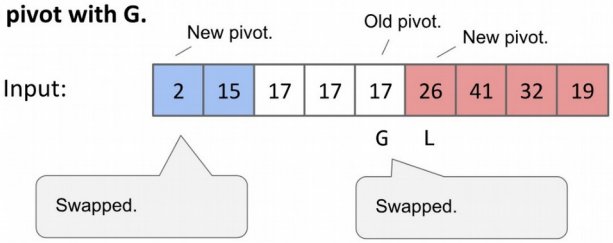


Hoare Partitioning

Partition strategy which will avoid O(n^2) When all elements are the same

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
  - When pointers cross, you are done walking.
- **Swap pivot with G.**



Other Desirable Sorting Properties: Stability

A sort is said to be stable if order of equivalent items is preserved.

sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Bas	3
Jana	3
Jouni	3
Rosella	3
Fikriyya	4
Nikolaj	4

Equivalent items don't 'cross over' when being stably sorted.

Stability

	Memory	# Compares	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)	No
Insertion	Θ(1)	Θ(N^2)	Θ(N) if almost sorted	Yes
Mergesort	Θ(N)	Θ(N log N)		Yes
Quicksort LTHS	Θ(log N)	Θ(N log N) expected	Fastest sort	No

This is due to the cost of tracking recursive calls by the computer, and is also an "expected" amount. The difference between log N and constant memory is trivial.

You can create a stable Quicksort (i.e. the version from the previous lecture). However, unstable partitioning schemes (like Hoare partitioning) tend to be faster. All reasonable partitioning schemes yield Θ(N log N) expected runtime, but with different constants.

We have shown several sorts to require Θ(N log N) worst case time.

- Can we build a better sorting algorithm?

By comparison sort, I mean that it uses e.g. the compareTo method in Java to make decisions.

Let the ultimate comparison sort (TUCS) be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let R(N) be its worst case runtime.

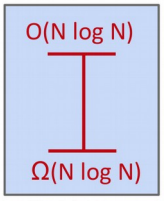
The Sorting Lower Bound (Finally)

Since TUCS is Ω(lg N!) and lg N! is Ω(N log N), we have that **TUCS is Ω(N log N).**

Any comparison based sort requires at least order N log N comparisons in its worst case.

Proof summary:

- Puppy, cat, dog is Ω(lg N!), i.e. requires lg N! comparisons.
- TUCS can solve puppy, cat, dog, and thus takes Ω(lg N!) compares.
- lg(N!) is Ω(N log N)
  - This was because N! is $\Omega(N/2)^{N/2}$



TUCS Worst Case Θ Runtime

Informally: TUCS ≥ puppy, cat, dog ≥ log N! ≥ N log N

lecture 35

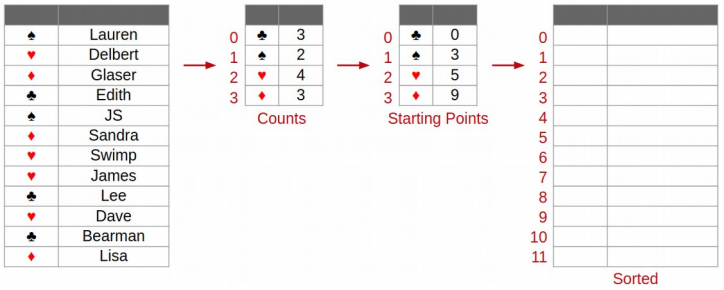
Counting Sort

Alphabet case: Keys belong to a finite ordered alphabet.

- Example: {♣, ♠, ♥, ♦} (in that order)

Example:

- Alphabet: {♣, ♠, ♥, ♦}



Counting Sort Runtime Analysis

Total runtime on N keys with alphabet of size R: Θ(N+R)

- Create an array of size R to store counts: Θ(R)
- Counting number of each item: Θ(N)
- Calculating target positions of each item: Θ(R)
- Creating an array of size N to store ordered data: Θ(N)
- Copying items from original array to ordered array: Do N times:
  - Check target position: Θ(1)
  - Update target position: Θ(1)
- Copying items from ordered array back to original array: Θ(N)

Memory usage: Θ(N+R)

For ordered array. For counts and starting points.

Empirical experiments needed to compare vs. Quicksort on practical inputs.

Bottom line: If N is ≥ R, then we expect reasonable performance.

Radix Sort

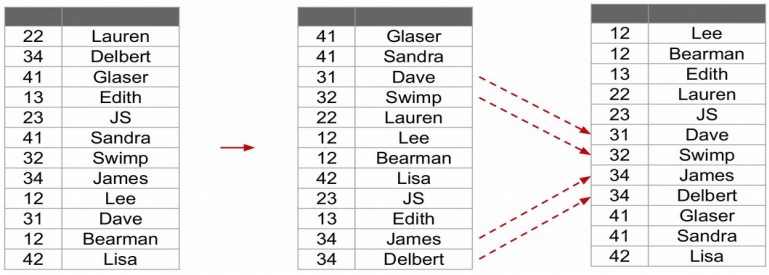
Not all keys belong to finite alphabets, e.g. Strings.

- However, Strings consist of characters from a finite alphabet.

LSD (Least Significant Digit) Radix Sort -- Using Counting Sort

Sort each digit independently from rightmost digit towards left.

- Example: Over {1, 2, 3, 4}

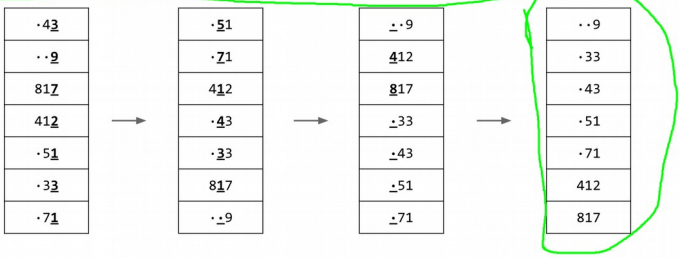


What is the runtime of LSD sort?

- Θ(WN+WR)
- N: Number of items, R: size of alphabet, W: Width of each item in # digits

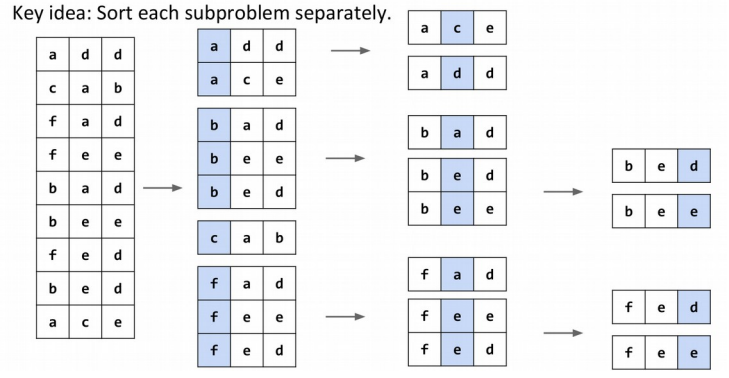
Non-equal Key Lengths

When keys are of different lengths, can treat empty spaces as less than all other characters.



MSD Radix Sort (correct edition)

Key idea: Sort each subproblem separately.



Sorting Runtime Analysis

	Memory	Runtime (worst)	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)*	Bad caching (61C)	No
Insertion	Θ(1)	$\Theta(N^2)$*	Fastest for small N, almost sorted data	Yes
Mergesort	Θ(N)	Θ(N log N)*	Fastest stable sort	Yes
Random Quicksort	Θ(log N)	Θ(N log N)* expected	Fastest compare sort	No
Counting Sort	Θ(N+R)	Θ(N+R)	Alphabet keys only	Yes
LSD Sort	Θ(N+R)	Θ(WN+WR)	Strings of alphabetical keys only	Yes
MSD Sort	Θ(N+WR)	Θ(N+R) (best) Θ(WN+WR) (worst)	Bad caching (61C)	Yes

N: Number of keys. R: Size of alphabet. W: Width of longest key.

*: Assumes constant compareTo time.

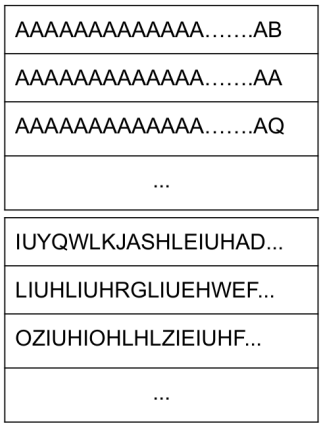
LSD vs. Merge Sort (My Answer)

The facts:

- Treating alphabet size as constant, LSD Sort has runtime Θ(WN).
- Merge Sort is between Θ(N log N) and Θ(WN log N).

Which is better? It depends.

- When might LSD sort be faster?
  - Sufficiently large N.
  - If strings are very similar to each other.
    - Each Merge Sort comparison costs Θ(W) time.
- When might Merge Sort be faster?
  - If strings are highly dissimilar from each other.
    - Each Merge Sort comparison is very fast.



LSD Radix Sort on Integers

Note: There's no reason to stick with base 10!

- Could instead treat as a base 16, base 256, base 65536 number.

Example: 512,312 in base 16 is a 5 digit number:

- $512312_{10} = (7 \times 16^4) + (13 \times 16^3) + (1 \times 16^2) + (3 \times 16^1) + (8 \times 16^0)$

Note this digit is greater than 9! That's OK, because we're in base 16.

Example: 512,312 in base 256 is a 3 digit number:

- $512312_{10} = (7 \times 256^2) + (209 \times 256^1) + (56 \times 256^0)$

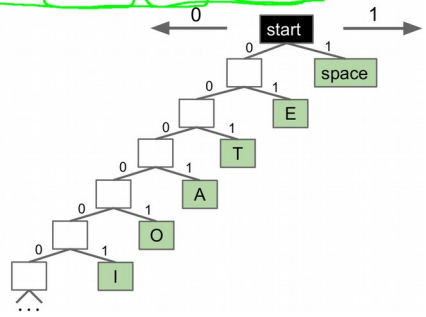
Note these digit are greater than 9! That's OK, because we're in base 256.

lecture 38

Prefix-Free Codes [Example 1]

A prefix-free code is one in which no codeword is a prefix of any other. Example for English:

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	



I ATE: 000001100010010 1

Prefix Free Code Design

Observation: Some prefix-free codes are better for some texts than others.

Better for EEEEAT (8+3+4 = 15 bits).

space	1
E	01
T	001
A	0001
O	00001
I	000001
...	

Much worse for JOSH (25+5+8+10 = 48 bits).

space	111
E	010
T	1101
A	1011
O	1001
I	1000
...	

Worse for EEEEAT (12+4+4 = 20 bits).

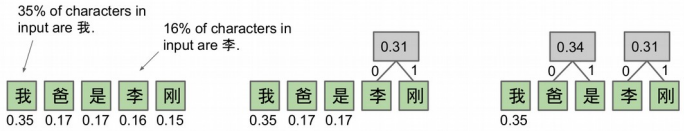
Better for JOSH (7+4+6+6 = 23 bits).

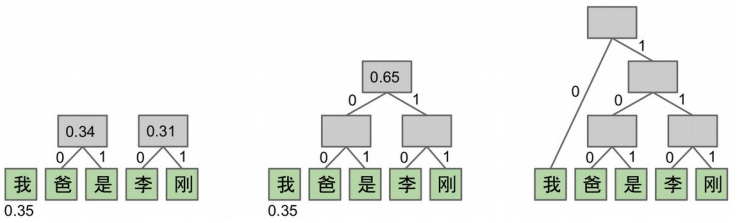
Observation: It'd be useful to have a procedure that calculates the "best" code for a given text.

Code Calculation Approach #2: Huffman Coding

Calculate relative frequencies.

- Assign each symbol to a node with weight = relative frequency.
- Take the two smallest nodes and merge them into a super node with weight equal to sum of weights.
- Repeat until everything is part of a tree.





Huffman Compression

Two possible philosophies for using Huffman Compression:

1. For each input type (English text, Chinese text, images, Java source code, etc.), assemble huge numbers of sample inputs for that category. Use each corpus to create a standard code for English, Chinese, etc.
2. For every possible input file, create a unique code just for that file. Send the code along with the compressed file.

What are some advantages/disadvantages of each idea? Which is better?

- Approach 1 will result in suboptimal encoding.
- Approach 2 requires you to use extra space for the codeword table in the compressed bitstream.

Huffman Coding Summary

Given a file X.txt that we'd like to compress into X.huf:

- Consider each b-bit symbol (e.g. 8-bit chunks, Unicode characters, etc.) of X.txt, counting occurrences of each of the 2^b possibilities, where b is the size of each symbol in bits.
- Use Huffman code construction algorithm to create a decoding trie and encoding map. Store this trie at the beginning of X.huf.
- Use encoding map to write codeword for each symbol of input into X.huf.

To decompress X.huf:

- Read in the decoding trie.
- Repeatedly use the decoding trie's longestPrefixOf operation until all bits in X.hug have been converted back to their uncompressed form.

lecture 39

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
	Y	Y	N	N
private	Y	N	N	N

Implementation	Constructor	connect	isConnected
ListOfSetsDS	Θ(N)	O(N)	O(N)
QuickFindDS	Θ(N)	Θ(N)	Θ(1)
QuickUnionDS	Θ(N)	O(N)	O(N)
WeightedQuickUnionDS	Θ(N)	O(log N)	O(log N)

[{0, 1, 2, 4}, {3, 5}, {6}]

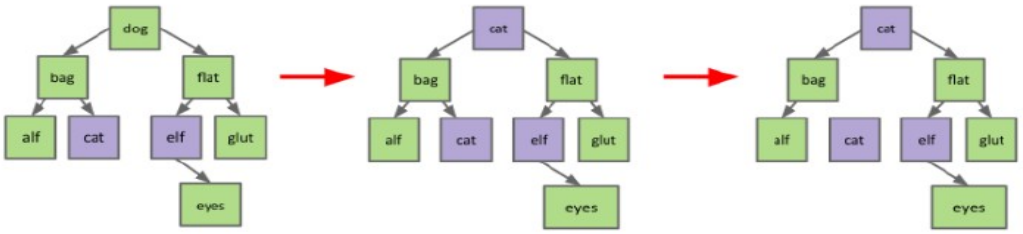
int[] id	4	4	4	5	4	5	6
	0	1	2	3	4	5	6

connect(5, 2) Set root(5)'s value equal to root(2)

Track tree size (**number** of elements).
New rule: Always link root of ***smaller*** tree ***to larger*** tree

Path Compression When we do isConnected(15, 10), tie all nodes seen to the root

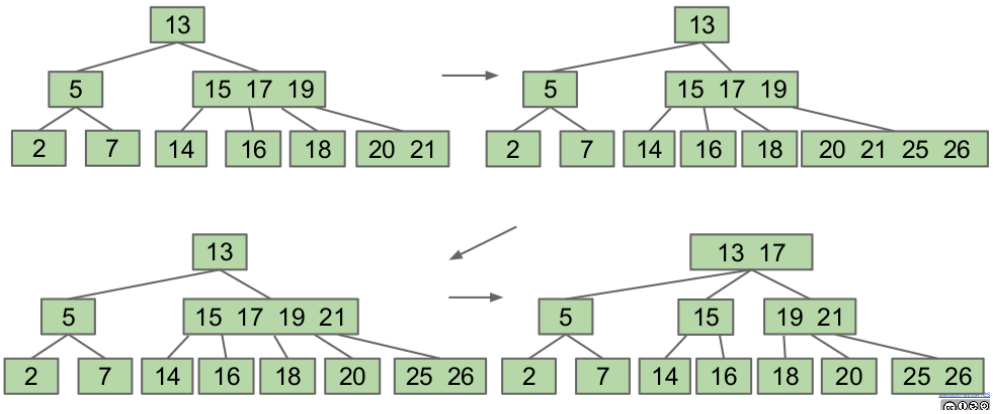
take the right-most node in the left subtree or the left-most node in the right subtree. **Hibbard deletion**



BSTs have best case height Θ(log N), and worst case height Θ(N).

- Suppose we add 25, 26:

All leaves must be the same distance from the source.
A non-leaf node with k items must have exactly k+1 children.



Runtime for contains

Runtime for add

O(L log N).

L is a constant

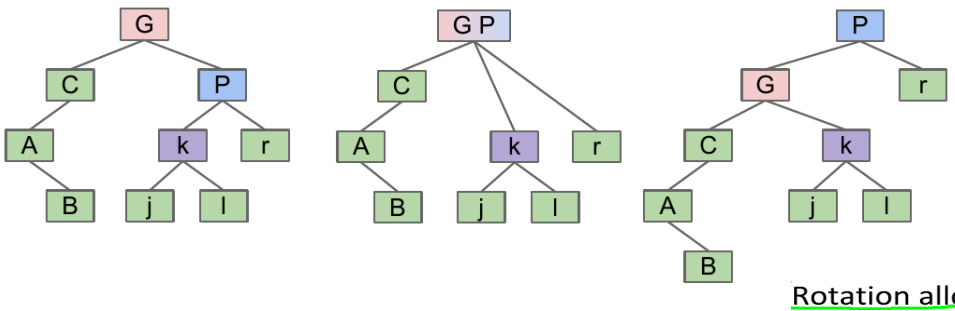
runtime is therefore O(log N)

Height: Between ~$\log_{L+1}(N)$ and ~$\log_2(N)$

Tree Rotation Definition

rotateLeft(G): Let x be the right child of G. Make G the **new left child** of x.

- Can think of as temporarily merging G and P, then sending G down and **left**.
- Preserves search tree property. No change to semantics of tree.



Rotation allows balancing of a BST in O(N) moves

A BST with left glue links that represents a 2-3 tree

maximum height of the corresponding LLRB? H (black) + (H + 1) (red) = 2H + 1.

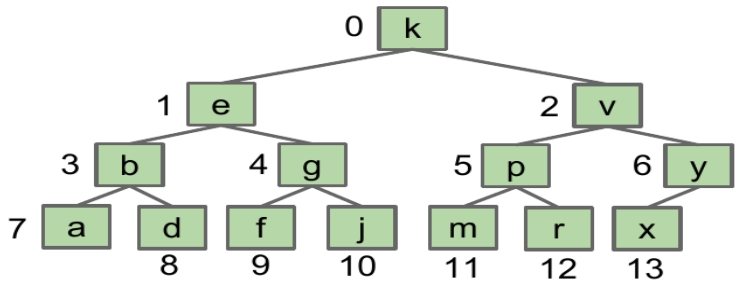
- When inserting: Use a red link.
- If there is a *right leaning "3-node"*, we have a **Left Leaning Violation.**
  - Rotate left the appropriate node to fix.
- If there are *two consecutive left links*, we have an **Incorrect 4 Node Violation.**
  - Rotate right the appropriate node to fix.
- If there are any *nodes with two red children*, we have a **Temporary 4 Node.**
  - Color flip the node to emulate the split operation.

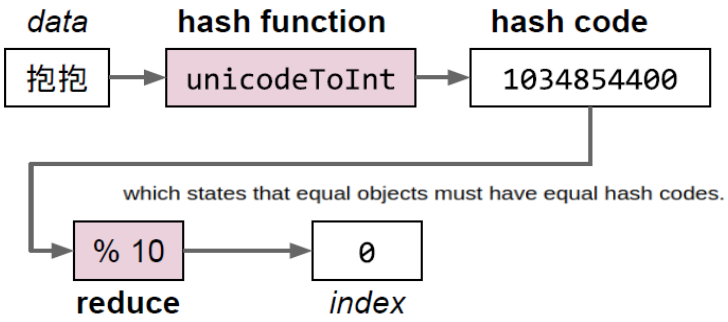
LLRB tree has height O(log N).
Contains is trivially O(log N).
Insert is O(log N).
- O(log N) to add the new node.
- O(log N) rotation and color flip operations per insert.

Given a heap, how do we implement PQ operations?

- `getSmallest()` - return the item in the root node.
- `add(x)` - place the new employee in the last position, and promote as high as possible.
- `removeSmallest()` - assassinate the president (of the company), promote the rightmost person in the company to president. Then demote repeatedly, always taking the 'better' successor.

Min-heap: Every node is less than or equal to both of its children.
Complete: Missing items only at the bottom level (if any), all nodes are as far left as possible.





When N/M is ≥ 1.5, then double M.

N/M is often called the "load factor"

A typical hash code base is a small prime.

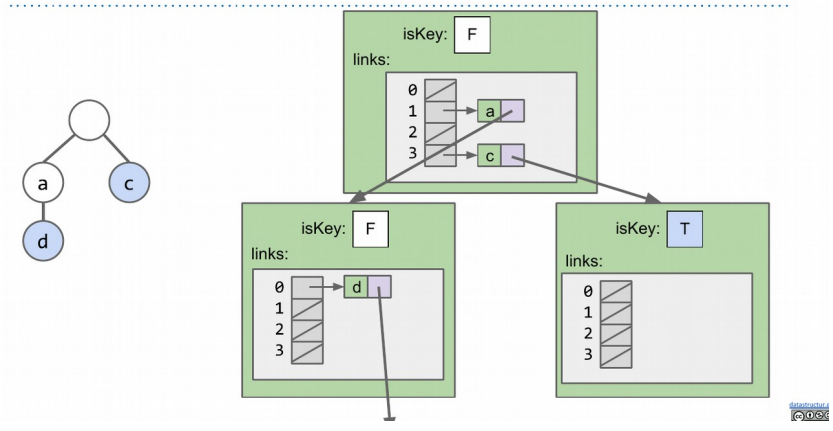
- Why prime?
  - Never even: Avoids the overflow issue on previous slide.
  - Lower chance of resulting hashCode having a bad relationship with the number of buckets: See study guide problems and hw3.

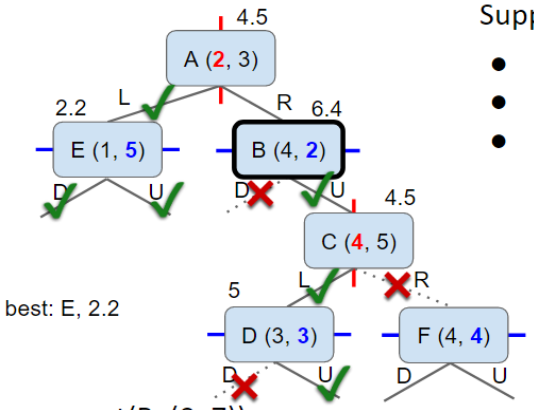
	Ordered Array	Bushy BST	Hash Table	Heap
add	Θ(N)	Θ(log N)	Θ(1)	Θ(log N)
getSmallest	Θ(1)	Θ(log N)	Θ(N)	Θ(1)
removeSmallest	Θ(N)	Θ(log N)	Θ(N)	Θ(log N)

k	e	v	b	g	p	y	a	d	f	j	m	r	x
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
public int parent(int k) {
    return (k - 1) / 2;
}
```

Alternate Idea #1: The Hash-Table Based Trie





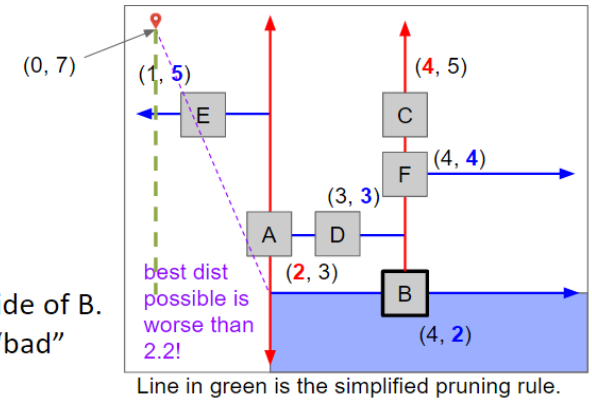
best: E, 2.2

nearest(B, (0, 7))

- ...just finished exploring the good side of B.
- Could something better be on the "bad" side of the line, i.e. B.down? No!

Suppose we have the k-d tree shown.

- We want to find nearest((0, 7)).
- Can visually see the answer is (1, 5).
- Let's do a proper k-d tree traversal.



Line in green is the simplified pruning rule.

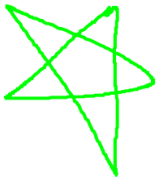
BreadthFirstPaths Demo

Goal: Find shortest path between s and every other vertex.

- **Initialize the fringe (a queue with a starting vertex s) and mark that vertex.**
- Repeat until fringe is empty:
  - Remove vertex v from fringe.
  - For each unmarked neighbor n of v: mark n, add n to fringe, set edgeTo[n] = v, set distTo[n] = distTo[v] + 1.

What is the order of growth of the running time of the print client if the graph uses an ***adjacency-list*** representation, where V is the number of vertices, and E is the total number of edges?

A. Θ(V)
B. Θ(V + E)
C. $\Theta(V^2)$
D. Θ(V*E)

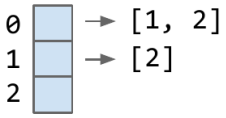


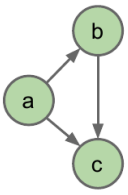
```
for (int v = 0; v < G.V(); v += 1) {
    for (int w : G.adj(v)) {
        System.out.println(v + "-" + w);
    }
}
```

Best case: Θ(V) Worst case: $\Theta(V^2)$

All cases: Θ(V + E)

- Create V iterators.
- Print E times.





idea	addEdge(s, t)	for(w : adj(v))	print()	hasEdge(s, t)	space used
adjacency matrix	Θ(1)	Θ(V)	$\Theta(V^2)$	Θ(1)	$\Theta(V^2)$
list of edges	Θ(1)	Θ(E)	Θ(E)	Θ(E)	Θ(E)
adjacency list	Θ(1)	Θ(1) to Θ(V)	Θ(V+E)	Θ(degree(v))	Θ(E+V)

DFS and BFS runtime with adjacency list: O(V + E)
DFS and BFS runtime with adjacency matrix: $O(V^2)$

Dijkstra's Demo

Insert all vertices into fringe PQ, storing vertices in order of distance from source.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

V: 7
Number of edges in SPT is 6

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V) for a connected graph.

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ removeSmallest	V	O(log V)	O(V log V)
PQ changePriority	E	O(log V)	O(E log V)

Prim's Demo

A ***minimum spanning tree*** is a spanning tree of minimum total weight. V-1 edges

Insert all vertices into fringe PQ, storing vertices in order of distance from tree.

Repeat: Remove (closest) vertex v from PQ, and relax all edges pointing from v.

Overall runtime: O(V*log(V) + V*log(V) + E*logV).

- Assuming E > V, this is just O(E log V).

	# Operations	Cost per operation	Total cost
PQ add	V	O(log V)	O(V log V)
PQ delMin	V	O(log V)	O(V log V)
PQ decreasePriority	O(E)	O(log V)	O(E log V)

Kruskal's Algorithm

Initially mark all edges gray.

- Consider edges in increasing order of weight.
- Add edge to MST (mark black) unless doing so creates a cycle.
- Repeat until V-1 edges.

Kruskal's Runtime

Kruskal's algorithm on previous slide is O(E log E).

Fun fact: In HeapSort lecture, we discuss how do this step in O(E) time using "bottom-up heapification".

Operation	Number of Times	Time per Operation	Total Time
Insert	E	O(log E)	O(E log E)
Delete minimum	O(E)	O(log E)	O(E log E)
union	O(V)	O(log* V)	O(V log* V)
isConnected	O(E)	O(log* V)	O(E log*V)

Note: If we use a pre-sorted list of edges (instead of a PQ), then we can simply iterate through the list in O(E) time, so overall runtime is O(E + V log*V + E log* V) = O(E log*V).

Suppose we have tasks 0 through 7, where an arrow from v to w indicates that v must happen before w. graph is a directed acyclic graph (DAG).

Perform a DFS traversal from every vertex with indegree 0, NOT clearing markings in between traversals.

- Record DFS postorder in a list.
- Topological ordering is given by the reverse of that list (reverse postorder).

Try to come up with an algorithm for shortest paths on a DAG that works even if there are negative edges.

DAG SPT Algorithm

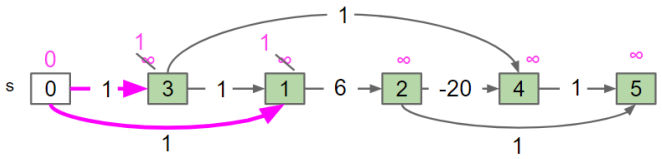
Visit vertices in topological order.

- When we visit a vertex: relax all of its going edges.

DAG LPT solution for graph G:

- Form a new copy of the graph G' with signs of all edge weights flipped.
- Run DAGSPT on G' yielding result X.

#	distTo	edgeTo
0	0	-
1	1	0
2	∞	-
3	1	0
4	∞	-
5	∞	-



Fringe: [3, 1, 2, 4, 5]

Problem	Problem Description	Solution	Efficiency
topological sort	Find an ordering of vertices that respects edges of our DAG.	Demo Code: Topological.java	O(V+E) time Θ(V) space
DAG shortest paths	Find a shortest paths tree on a DAG.	Demo Code: AcyclicSP.java	O(V+E) time Θ(V) space
longest paths	Find a longest paths tree on a graph.	No known efficient solution.	O(???) time O(???) space
DAG longest paths	Find a longest paths tree on a DAG.	Flip signs, run DAG SPT, flip signs again.	O(V+E) time Θ(V) space

Selection Sort

Find smallest item.
Swap this item to the front and 'fix' it.
Repeat for unfixed items until all items are fixed

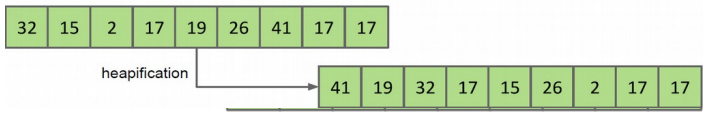
Mergesort

Split items into 2 roughly even pieces.
Mergesort each half (steps not shown, this is a recursive
Merge the two sorted halves to form the final result.

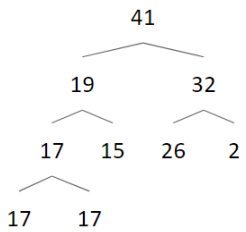
In-place Heap Sort

Heap sorting N items: To bottom-up heapify, just sink nodes in reverse level order.

- **Bottom-up heapify input array (done!).**
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.



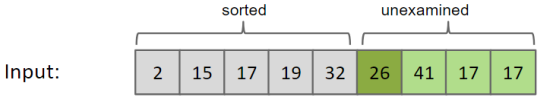
Size: 9



In-place Insertion Sort

Repeat for i = 0 to N - 1:

- **Designate item i as the traveling item.**
- Swap item backwards until traveller is in the right place among all previously examined items.



On arrays with a small number of inversions, insertion sort is extremely fast.

- One exchange per inversion (and number of comparisons is similar). Runtime is Θ(N + K) where K is number of inversions.
- Define an ***almost sorted*** array as one in which number of inversions ≤ cN for some c. Insertion sort is excellent on these arrays.

	Best Case Runtime	Worst Case Runtime	Space
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	Θ(1)
Heapsort (in place)	Θ(N)*	Θ(N log N)	Θ(1)
Mergesort	Θ(N log N)	Θ(N log N)	Θ(N)
Insertion Sort (in place)	Θ(N)	$\Theta(N^2)$	Θ(1)

Is insertion sort stable?

- Yes.
- Equivalent items never move past their equivalent brethren.

Partition Sort, a.k.a. Quicksort

Partition on leftmost item.
Quicksort left half.
Quicksort right half.

If pivot always lands somewhere "good", Quicksort is Θ(N log N). However, the very rare Θ(N^2) cases do happen in practice, e.g.

- Bad ordering: Array already in sorted order (or almost sorted order).
- Bad elements: Array with all duplicates.

Pivot selection: Always use **leftmost**.

QuicksortL3S

Shuffle before starting (to avoid worst case)

Partition algorithm: Make an array copy then do **three** scans for red, white and blue items (white scan trivially finishes in one compare).

Hoare Partitioning

Partition strategy which will avoid O(n^2) When all elements are the same

Create L and G pointers at left and right ends.

- L pointer is a friend to small items, and hates large or equal items.
- G pointer is a friend to large items, and hates small or equal items.
- Walk pointers towards each other, stopping on a hated item.
  - When both pointers have stopped, swap and move pointers by one.
- When pointers cross, you are done.

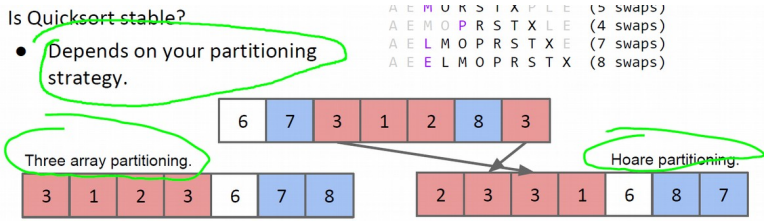
	Memory	# Compares	Notes	Stable?
Quicksort LTHS	Θ(log N)	Θ(N log N) expected	Fastest sort	No

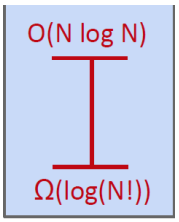
Stability

Equivalent items don't 'cross over' when being stably sorted

Is Quicksort stable?

- Depends on your partitioning strategy.





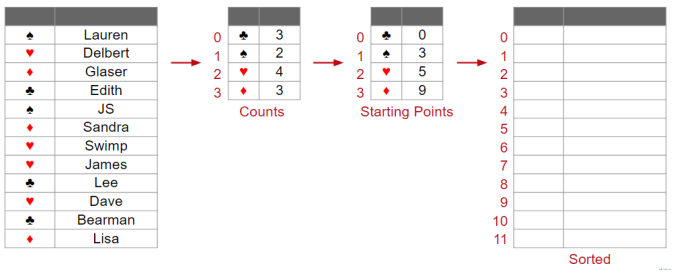
TUCS Worst Case Θ Runtime

Counting Sort

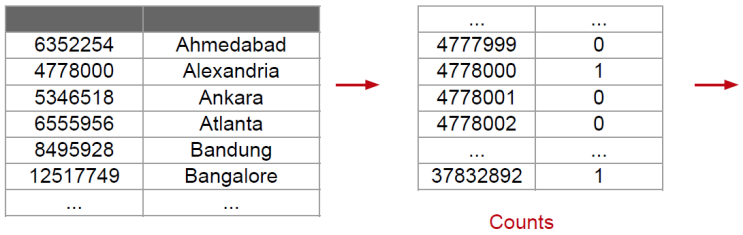
Alphabet case: Keys belong to a finite ordered alphabet.

Example:

- Alphabet: {♣, ♠, ♥, ♦}



Alphabet: {0,1,...,37832892(biggest population)}



Counting Sort	Θ(N+R)	Θ(N+R)	Alphabet keys only	Yes

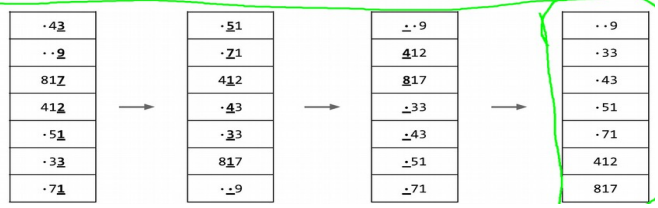
N: Number of keys. R: Size of alphabet.

LSD (Least Significant Digit) Radix Sort -- Using Counting Sort

Sort each digit independently from rightmost digit towards left.

- Example: Over {1, 2, 3, 4}

When keys are of different lengths, can treat empty spaces as less than all other characters.



LSD Sort	Θ(N+R)	Θ(WN+WR)	Strings of alphabetical keys only	Yes

	Memory	Runtime (worst)	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)*	Bad caching (61C)	No
Insertion	Θ(1)	Θ(N^2)*	Fastest for small N, almost sorted data	Yes
Mergesort	Θ(N)	Θ(N log N)*	Fastest stable sort	Yes
Random Quicksort	Θ(log N)	Θ(N log N)* expected	Fastest compare sort	No
Counting Sort	Θ(N+R)	Θ(N+R)	Alphabet keys only	Yes
LSD Sort	Θ(N+R)	Θ(WN+WR)	Strings of alphabetical keys only	Yes
MSD Sort	Θ(N+WR)	Θ(N+R) (best) Θ(WN+WR) (worst)	Bad caching (61C)	Yes

N: Number of keys. R: Size of alphabet. W: Width of longest key.

MSD (Most Significant Digit) Radix Sort

Just like LSD, but sort from leftmost digit towards the right.

Sort each subproblem separately

Best Case.

- We finish in one counting sort pass, looking only at the top digit: Θ(N + R)

Worst Case.

- We have to look at every character, degenerating to LSD sort: Θ(WN + WR)

What is Merge Sort's runtime on strings of length W?

It depends!

- Θ(N log N) if each comparison takes constant time.
  - Example: Strings are all different in top character.
- Θ(WN log N) if each comparison takes Θ(W) time.
  - Example: Strings are all equal.

LSD vs. Merge Sort (My Answer)

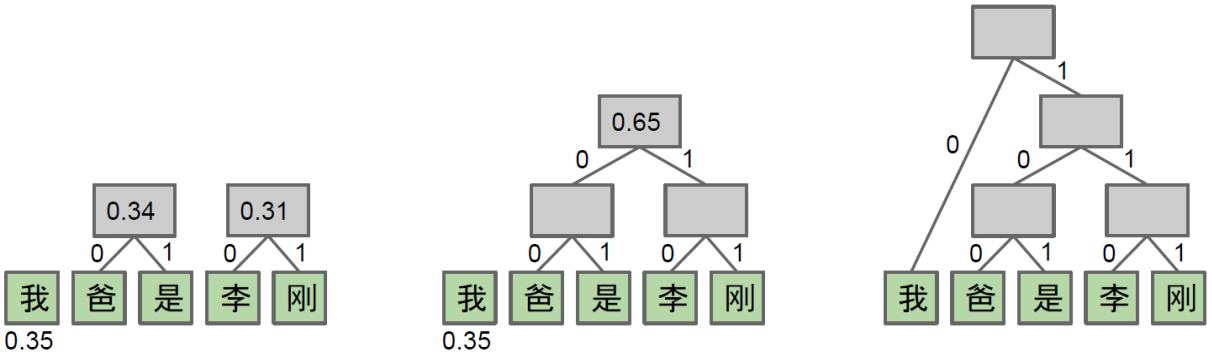
The facts:

- Treating alphabet size as constant, LSD Sort has runtime Θ(WN).
- Merge Sort is between Θ(N log N) and Θ(WN log N).

Which is better? It depends.

- When might LSD sort be faster?
  - Sufficiently large N.
  - If strings are very similar to each other.
    - Each Merge Sort comparison costs Θ(W) time.
- When might Merge Sort be faster?
  - If strings are highly dissimilar from each other.
    - Each Merge Sort comparison is very fast.

AAAAAAAAAAAAA.....AB
AAAAAAAAAAAAA.....AA
AAAAAAAAAAAAA.....AQ
...
IUYQWLKJASHLEIUHAD...
LIUHLIUHRGLIUEHWEF...
OZIUHIOHLHLZIEIUHF...
...




```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
    ...
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }

    private static class NameComparator implements Comparator<Dog> {
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }

    public static Comparator<Dog> getNameComparator() {
        return new NameComparator();
    }
}
```

```
public class ArraySet<T> implements Iterable<T> {
    private T[] items;
    private int size; // the next item to be added will be at position size

    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }
```

```
    private class ArraySetIterator implements Iterator<T> {
        private int wizPos;

        public boolean hasNext() {
            return wizPos < size;
        }

        public T next() {
            T returnItem = items[wizPos];
            wizPos += 1;
            return returnItem;
        }
```