Lec3 Testing

1.

str1.compareTo(str2) 方法会返回：

负数：str1 < str2

0:两者相等

正数: str1 > str2

Lec5 SLLists, Nested Classes, Sentinel Nodes

1.

如果嵌套类不需要访问外部类 `SLList` 的实例方法或变量，你可以将嵌套类声明为 `static`

```
public class SLList {
       public static class IntNode {
            public int item;
            public IntNode next;
            public IntNode(int i, IntNode n) {
                item = i;
                next = n;
            }
       }

       private IntNode first;
...
```

这样可以节省一点内存，因为每个 `IntNode` 不再需要保存访问其外部 `SLList` 的引用。换句话说，如果嵌套类从不使用外部类的实例成员，就可以用 `static`，节省内存。

Lec6 DLLists, Arrays

1.

The basic idea is that right after the name of the class in your class declaration, you use an arbitrary placeholder inside angle brackets: `<>`.

```
public class DLList {
    private IntNode sentinel;
    private int size;

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

to

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

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

Since generics only work with reference types, we cannot put primitives like `int` or `double` inside of angle brackets, e.g. `<int>`. Instead, we use the reference version of the primitive type, which in the case of `int` case is `Integer`, e.g.

```
DLList<Integer> d1 = new DLList<>(5);
d1.insertFront(10);
```

Lec7 ALists, Resizing vs SLists

1.

Java does not allow us to create an array of generic objects due to an obscure issue with the way generics are implemented. That is, we cannot do something like:

```
Glorp[] items = new Glorp[8];
```

Instead, we have to use the awkward syntax shown below:

```
Glorp[] items = (Glorp []) new Object[8];
```

Lec8 Inheritance, Implements

1.

```
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();
}
```

We will add to

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

a relationship-defining word: implements.

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

2.

override的方法是根据dynamic method selection来进行操作的，而重载方法不是这样（overload）

Say there are two methods in the same class

```
public static void peek(List61B<String> list) {
    System.out.println(list.getLast());
}
public static void peek(SLList<String> list) {
    System.out.println(list.getFirst());
}
```

and you run this code

```
SLList<String> SP = new SLList<String>();
List61B<String> LP = SP;
SP.addLast("elk");
SP.addLast("are");
SP.addLast("cool");
peek(SP);
peek(LP);
```

The first call to peek() will use the second peek method that takes in an SLList. The second call to peek() will use the first peek method which takes in a List61B. This is because the only distinction between two overloaded methods is the types of the parameters. When Java checks to see which method to call, it checks the **static type** and calls the method with the parameter of the same type.

3.

How do we differentiate between "interface inheritance" and "implementation inheritance"? Well, you can use this simple distinction:

- Interface inheritance (what): Simply tells what the subclasses should be able to do.
    - EX) all lists should be able to print themselves, how they do it is up to them.
- Implementation inheritance (how): Tells the subclasses how they should behave.
    - EX) Lists should print themselves exactly this way: by getting each element in order and then printing them.

When you are creating these hierarchies, remember that the relationship between a subclass and a superclass should be an "is-a" relationship. AKA Cat should only implement Animal Cat **is an** Animal. You should not be defining them using a "has-a" relationship. Cat **has-a** Claw, but Cat definitely should not be implementing Claw. (suitable to not only interface inheritance but also implementation inheritance

Lec9 Extends, Casting, Higher Order Functions

1.

We'd like to build RotatingSLList that can perform any SLList operation as well as: rotateRight(): Moves back item to the front.

```
public class RotatingSLList<Blorp> extends SLList<Blorp>{
       public void rotateRight() {
              Blorp oldBack = removeLast();
              addFirst(oldBack);
    }
}
```

Because of extends, RotatingSLList inherits all members of SLList:

- All instance and static variables.
- All methods.
- All nested classes.
- Constructors are **not** inherited!

Lec10 Subtype Polymorphism vs HoFs

Comparables

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

Notice that `Comparable<T>` means that it takes a generic type. This will help us avoid having to cast an object to a specific type! Now, we will rewrite the Dog class to implement the Comparable interface, being sure to update the generic type `T` to Dog:

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

Comparators

1.

Natural order - used to refer to the ordering implied in the `compareTo` method of a particular class.

2.

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

```
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();
    }
}
```

什么时候用Comparable:只有一个要比较的时候

什么时候用Comparator：有多个需要比较的时候

3.

在中文中，Comparable通常翻译成可比较接口/可比较性接口，Comparator通常翻译成比较器接口

Lec11 Exceptions, Iterators, Object Methods

Iteration

1.

We check that there are still items left with `seer.hasNext()`, which will return true if there are unseen items remaining, and false if all items have been processed.

Last, `seer.next()` does two things at once. It returns the next element of the list, and here we print it out. It also advances the iterator by one item. In this way, the iterator will only inspect each item once.

2.

在中文中，Iterator通常翻译成迭代器，Iterable通常翻译成可迭代对象/可迭代容器

3.

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

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

```
import java.util.Iterator;

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 ArraySet() {
        items = (T[]) new Object[100];
        size = 0;
    }

    /* Returns true if this map contains a mapping for the specified key.
     */
    public boolean contains(T x) {
        for (int i = 0; i < size; i += 1) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /* Associates the specified value with the specified key in this map.
       Throws an IllegalArgumentException if the key is null. */
    public void add(T x) {
        if (x == null) {
            throw new IllegalArgumentException("can't add null");
        }
        if (contains(x)) {
            return;
        }
```

```
        items[size] = x;
        size += 1;
    }

    /* Returns the number of key-value mappings in this map. */
    public int size() {
        return size;
    }

    /** returns an iterator (a.k.a. seer) into ME */
    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }

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

        public ArraySetIterator() {
            wizPos = 0;
        }

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

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

    public static void main(String[] args) {
        ArraySet<Integer> aset = new ArraySet<>();
        aset.add(5);
        aset.add(23);
        aset.add(42);

        //iteration
        for (int i : aset) {
            System.out.println(i);
        }
    }

}
```

Lec14 Disjoint Sets

Introduction

1.

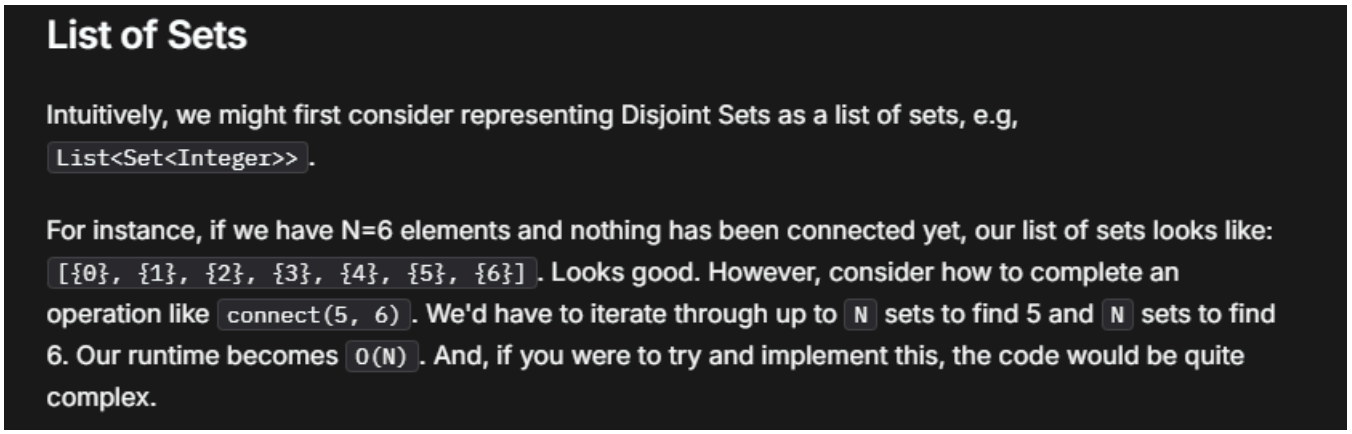
Disjoint-Sets也叫Union-Find

2.

```
public interface DisjointSets {
    /** connects two items P and Q */
    void connect(int p, int q);

    /** checks to see if two items are connected */
    boolean isConnected(int p, int q);
```

Quick Find

List of Sets



Quick Find

1.

```
public class QuickFindDS implements DisjointSets {

    private int[] id;

    /* Θ(N) */
    public QuickFindDS(int N){
        id = new int[N];
        for (int i = 0; i < N; i++){
            id[i] = i;
        }
    }

    /* need to iterate through the array => Θ(N) */
    public void connect(int p, int q){
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++){
            if (id[i] == pid){
                id[i] = qid;
            }
```

```
        }
    }

    /* Θ(1) */
    public boolean isConnected(int p, int q){
        return (id[p] == id[q]);
    }
}
```

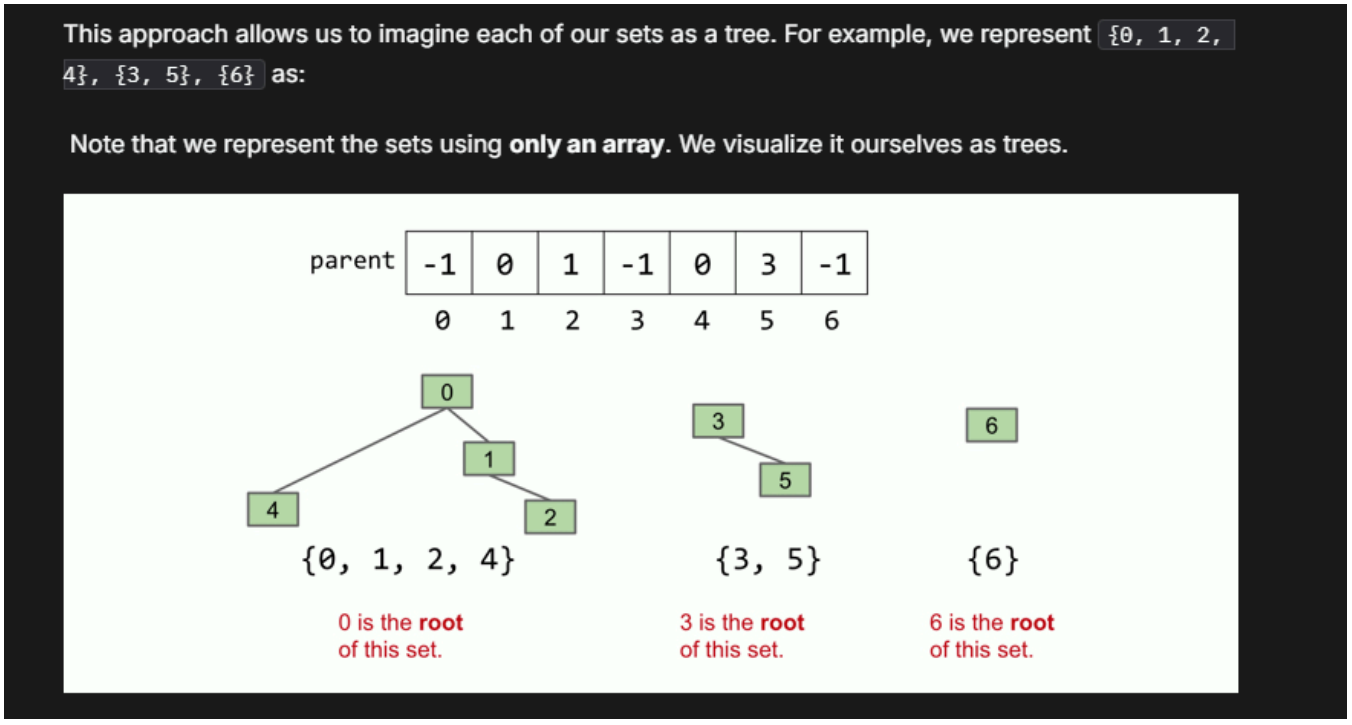
2.

Implementation	Constructor	`connect`	`isConnected`
ListOfSets	Θ(N)[1]	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)

Quick Union

1.

We will still represent our sets with an array. Instead of an id, we assign each item the index of its parent. If an item has no parent, then it is a 'root' and we assign it a negative value.



2.

N = number of elements in our DisjointSets data structure

Implementation	Constructor	`connect`	`isConnected`
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)
QuickUnion	Θ(N)	O(N)	O(N)

```java
public class QuickUnionDS implements DisjointSets {
    private int[] parent;

    public QuickUnionDS(int num) {
        parent = new int[num];
        for (int i = 0; i < num; i++) {
            parent[i] = -1;
        }
    }

    private int find(int p) {
        while (parent[p] >= 0) {
            p = parent[p];
        }
        return p;
    }

    @Override
    public void connect(int p, int q) {
        int i = find(p);
        int j= find(q);
        parent[i] = j;
    }

    @Override
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
}
```

Weighted Quick Union

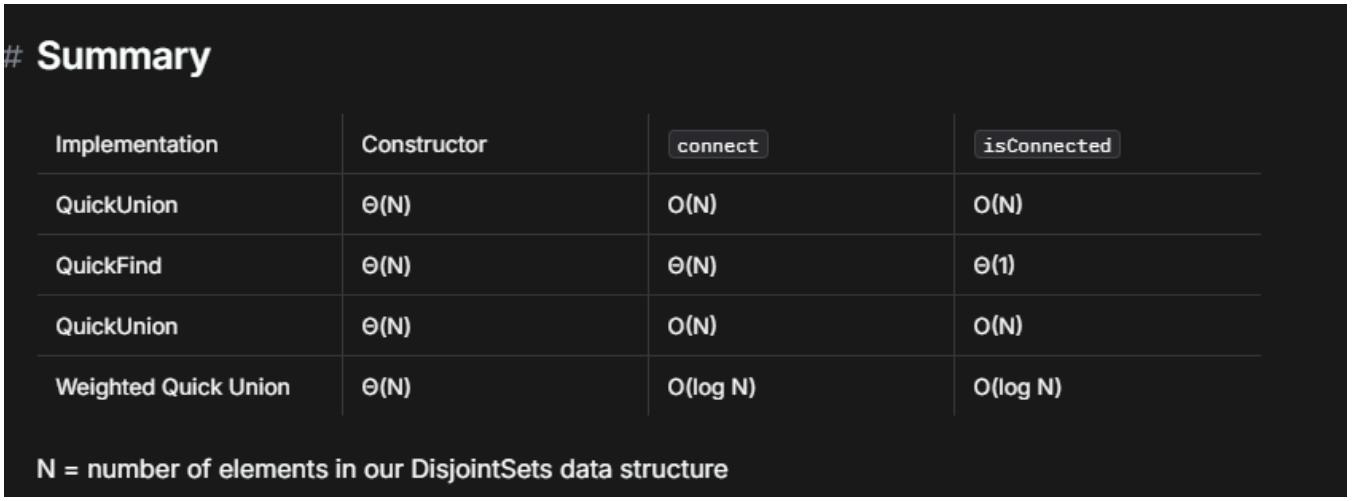
1.

New rule: whenever we call `connect`, we always link the root of the smaller tree to the larger tree.

2.

根据weight来，小的weight排到大的weight上面，root的值是 负号（wight）

3.



Summary

Implementation	Constructor	`connect`	`isConnected`
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)
QuickUnion	Θ(N)	O(N)	O(N)
Weighted Quick Union	Θ(N)	O(log N)	O(log N)

N = number of elements in our DisjointSets data structure

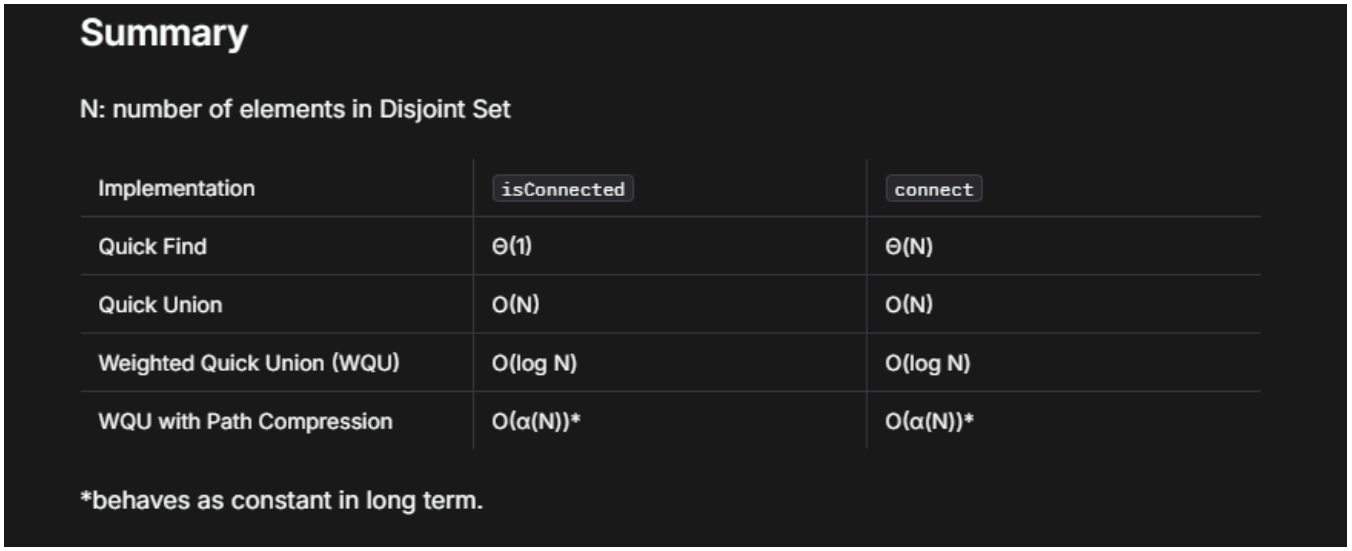
Weighted Quick Union with Path Compression

1.

Connecting all the items along the way to the root will help make our tree shorter with each call to `find`.

Recall that **both** `**connect(x, y)**` **and** `**isConnected(x, y)**` **always call** `**find(x)**` **and** `**find(y)**`**.** Thus, after calling `connect` or `isConnected` enough, essentially all elements will point directly to their root.

2.



Summary

N: number of elements in Disjoint Set

Implementation	`isConnected`	`connect`
Quick Find	Θ(1)	Θ(N)
Quick Union	O(N)	O(N)
Weighted Quick Union (WQU)	O(log N)	O(log N)
WQU with Path Compression	O(α(N))*	O(α(N))*

*behaves as constant in long term.

Lec16 ADTs， Sets, Maps, BSTs

Sets

Sets: an unordered set of unique elements (no repeats)

BST Delete

three categories:

- the node we are trying to delete has no children
- has 1 child

- has 2 children

Deletion: No Children

If the node has no children, it is a leaf, and we can just delete its parent pointer and the node will eventually be swept away by the garbage collector.

Deletion: One Child

If the node only has one child, we know that the child maintains the BST property with the parent of the node because the property is recursive to the right and left subtrees. Therefore, we can just reassign the parent's child pointer to the node's child and the node will eventually be garbage collected.

Deletion: Two Children

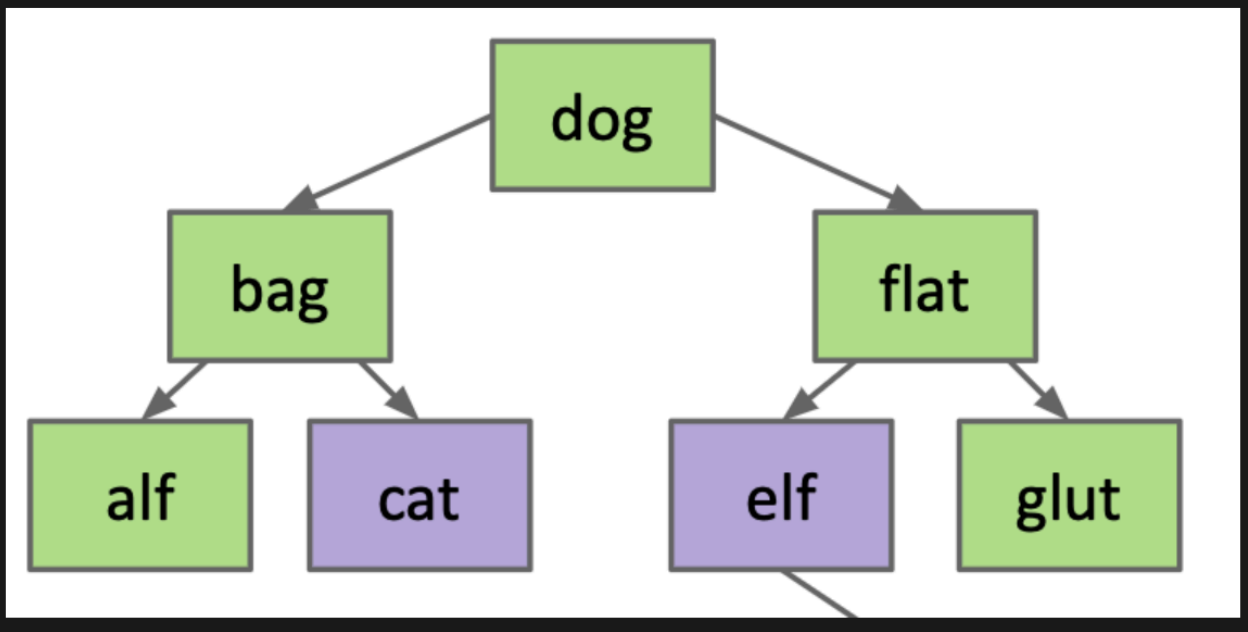
If the node has two children, the process becomes a little more complicated because we can't just assign one of the children to be the new root. This might break the BST property.

Instead, we choose a new node to replace the deleted one.

We know that the new node must:

- be > than everything in left subtree.
- be < than everything right subtree.

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



Possible candidates to replace `dog` after deletion

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.

BSTs as Sets and Maps

We can use a BST to implement the `Set` ADT. If we use a BST, we can decrease the runtime of `contains` to log(n) because of the BST property which enables us to use binary search!

We can also make a binary tree into a map by having each BST node hold `(key,value)` pairs instead of singular values. We will compare each element's key in order to determine where to place it within our tree.

Lec17 B-Trees(2-3，2-3-4 Trees)

B-Trees add

Upon `add` in a B-Tree, we simply append the value to an existing leaf node in the correct location instead of creating a new leaf node. If the node is too full, it splits and pushes a value up.

B-Trees Invariants

Because of the way B-Trees are constructed, they have two invariants:

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

These two invariants guarantee a "bushy" tree with logN height.

Runtime for contains and add

他们都是O(logN)

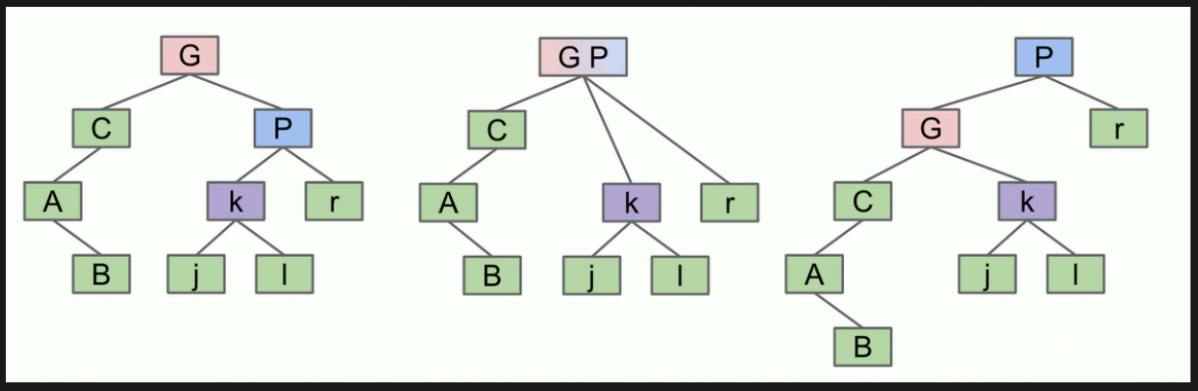
Lec18 Red Black Trees

Wonderfully balanced as they are, B-Trees are really difficult to implement. We need to keep track of the different nodes and the splitting process is pretty complicated. As computer scientists who appreciate clean code and a good challenge, let's find another way to create a balanced tree.

Tree rotation

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

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



rotateLeft(G)

The written description of what happened above is this:

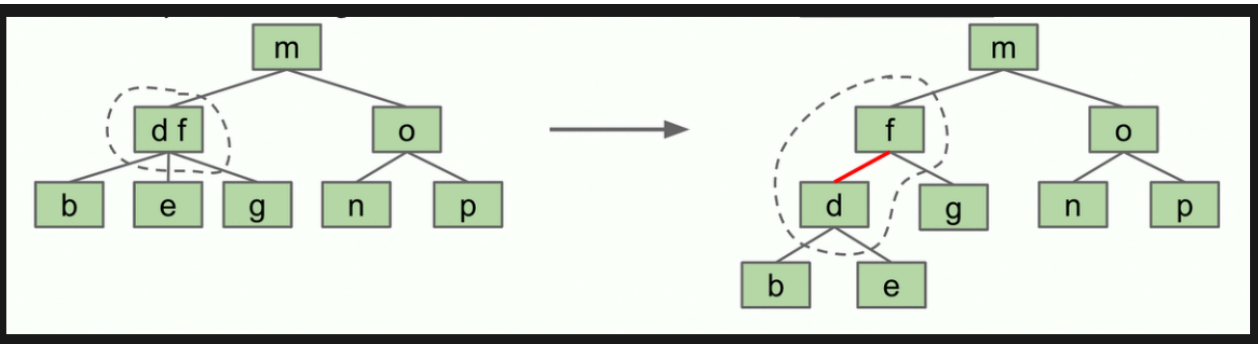
- G's right child, P, merges with G, bringing its children along.
- P then passes its left child to G and G goes down to the left to become P's left child.

You can see that the structure of the tree changes as well as its height. We can also rotate on a non-root node. We just disconnect the node from the parent temporarily, rotate the subtree at the node, then reconnect the new root.

Here are the implementations of `rotateRight` and `rotateLeft:`

```
private Node rotateRight(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    return x;
}

// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    return x;
}
```

Creating LLRB Trees



Left Leaning Red-Black Tree

We show that a link is a glue link by making it red. Normal links are black. Because of this, we call these structures **left-leaning red-black trees (LLRB)**. We will be using left-leaning trees in this course.

One-to-One Correspondence:

Left-Leaning Red-Black trees have a **1-1 correspondence with 2-3 trees.** Every 2-3 tree has a **unique** LLRB red-black tree associated with it.

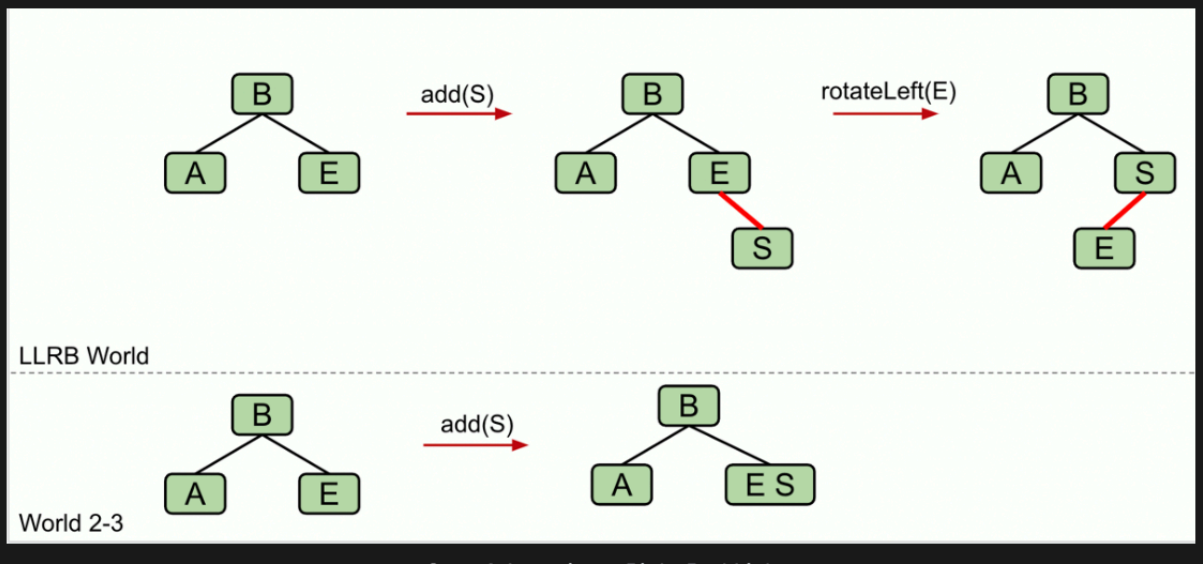
As for 2-3-4 trees, they maintain correspondence with standard Red-Black trees.

Below is a summary of the properties/invariants of LLRB Trees:

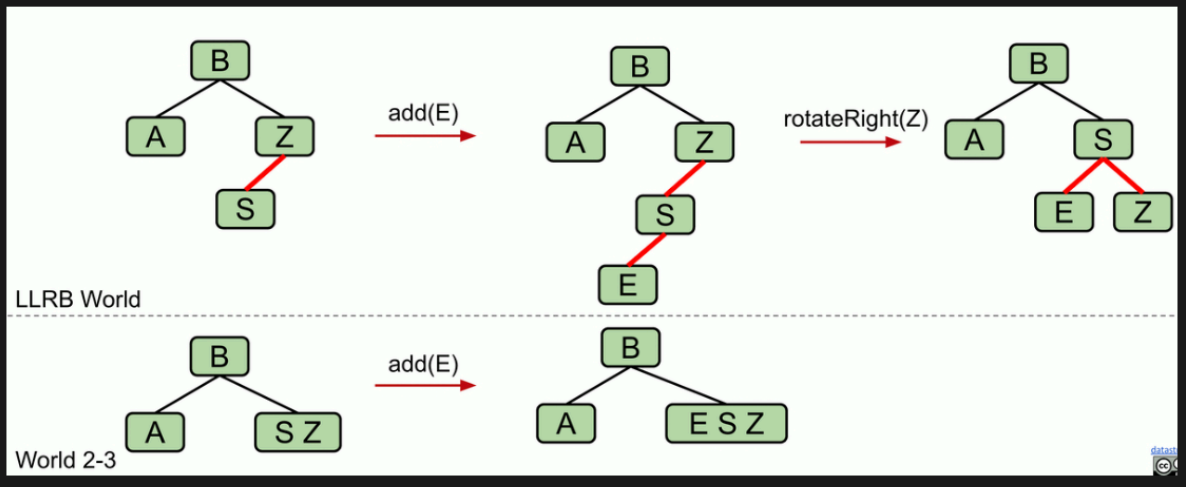
- 1-1 correspondence with 2-3 trees.
- No node has 2 red links.
- There are no red right-links.
- Every path from root to leaf has the same number of black links (because 2-3 trees have the same number of links to every leaf).
- Height is no more than 2x height + 1 of the corresponding 2-3 tree.
- The height of a red-black tree is proportional to the log of the number of entries.

Inserting LLRB Trees

- Case 1: **Insertion results in a right leaning "3-node → Left Leaning Violation**
  - **Task 1: rotateLeft(node)**
  - Recall, we are using *left-leaning* red black trees, which means we can never have a right red link.
  - To address this, we will need to use a **rotation** in order to maintain the LLRB invariant: "There are no right red links".
  - If the left child is *also* a red link, then we will temporarily allow it for purposes that will become clearer in case 2.

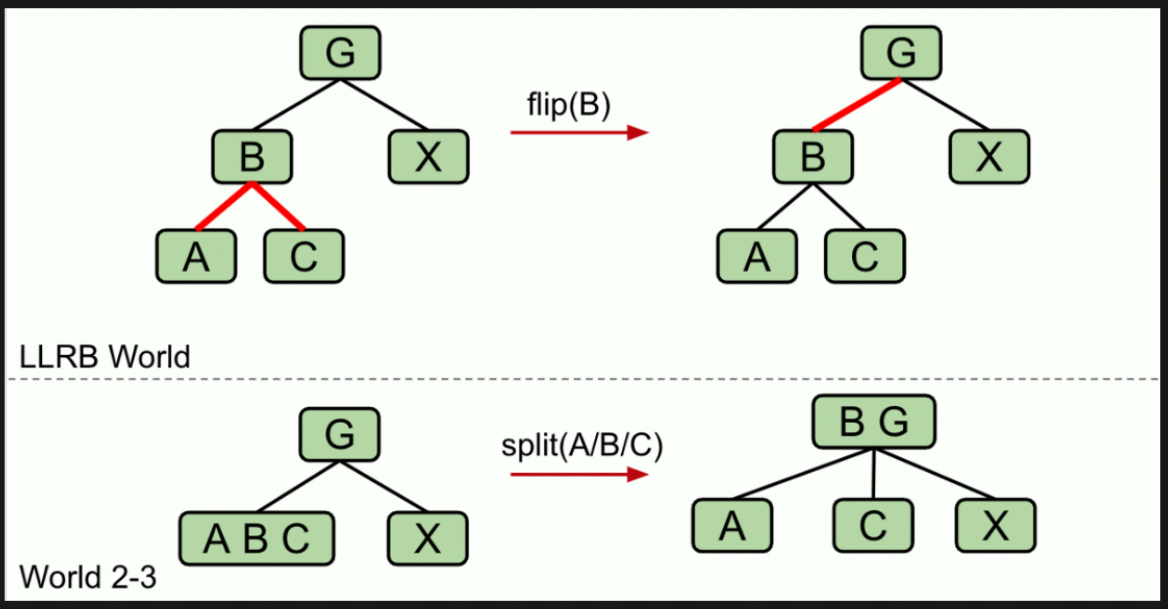


- Case 2: **Double Insertion on the Left → Incorrect 4 Node Violation**
  - **Task 2: rotateRight(node)**
  - If there are 2 left red links, then we have a 4-node which is illegal.
  - To address this, we will first **rotate** to create the same tree seen in case 1 above. Then, in both situations, we will address the temporary issue of having an illegal 4-node in case 3.



Case 2: Double Insertion on the Left

- Case 3: **Node has Two Red Children → Temporary 4 Node**
  - **Task 3: Color Flip**
  - Finally, to address the issue of having illegal 4-node by **flipping the colors** of all edges touching "S" above.
  - This is equivalent to pushing up the middle node in a 2-3 tree.



Case 3: Two Red Links

Note: You may need to go through a series of rotations in order to complete the transformation. It's possible that one operation will result in additional LLRB rule violations, which we can fix with the corresponding operation.

The process is: while the LLRB tree does not satisfy the 1-1 correspondence with a 2-3 tree or breaks the LLRB invariants, perform task 1, 2, or 3 depending on the condition of the tree until you get a legal LLRB.

Summary

LLRBs maintain correspondence with 2-3 trees, Standard Red-Black trees maintain correspondence with 2-3-4 trees.

- Java's TreeMap is a red-black tree that corresponds to 2-3-4 trees.
- 2-3-4 trees allow glue links on either side (see Red-Black Tree).
- More complex implementation, but faster.

Lec19 Hashing

1.

合法 hashCode 的确定性定义：

如果两个对象相等（`equals` 返回 true），它们必须有相同的 `hashCode`，这样哈希表才能正确找到它们。

2.

Bottom line: If your class override equals, you should also override hashCode in a consistent manner.

- If two objects are equal, they must always have the same hash code.

If you don't, everything breaks:

- `Contains` can't find objects (unless it gets lucky).
- `Add` results in duplicates.

Lec20 Heaps and PQs

Priority Queues

1.

```
/** (Min) Priority Queue: Allowing tracking and removal of
  * the smallest item in a priority queue. */
public interface MinPQ<Item> {
    /** Adds the item to the priority queue. */
    public void add(Item x);
    /** Returns the smallest item in the priority queue. */
    public Item getSmallest();
    /** Removes the smallest item from the priority queue. */
    public Item removeSmallest();
    /** Returns the size of the priority queue. */
    public int size();
}
```

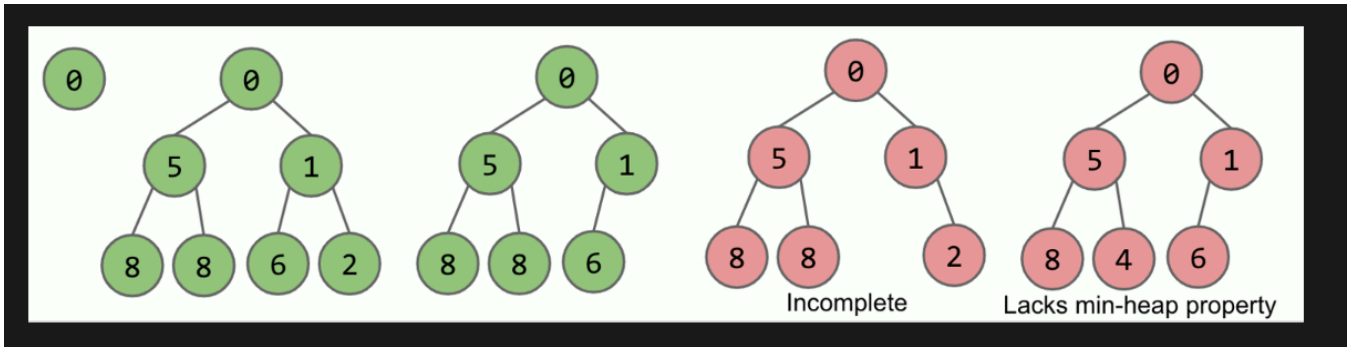
Priority Queue is an Abstract Data Type that optimizes for handling minimum or maximum elements.

Heaps

1.

We will define our binary min-heap as being **complete** and obeying **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.



2.

heap是priority queue这个adt的一个implementation

- `add`: Add to the end of heap temporarily. Swim up the hierarchy to the proper place.

    - Swimming involves swapping nodes if child < parent
- `getSmallest`: Return the root of the heap (This is guaranteed to be the minimum by our *min-heap* property
- `removeSmallest`: Swap the last item in the heap into the root. Sink down the hierarchy to the proper place.
    - Sinking involves swapping nodes if parent > child. Swap with the smallest child to preserve *min-heap* property.

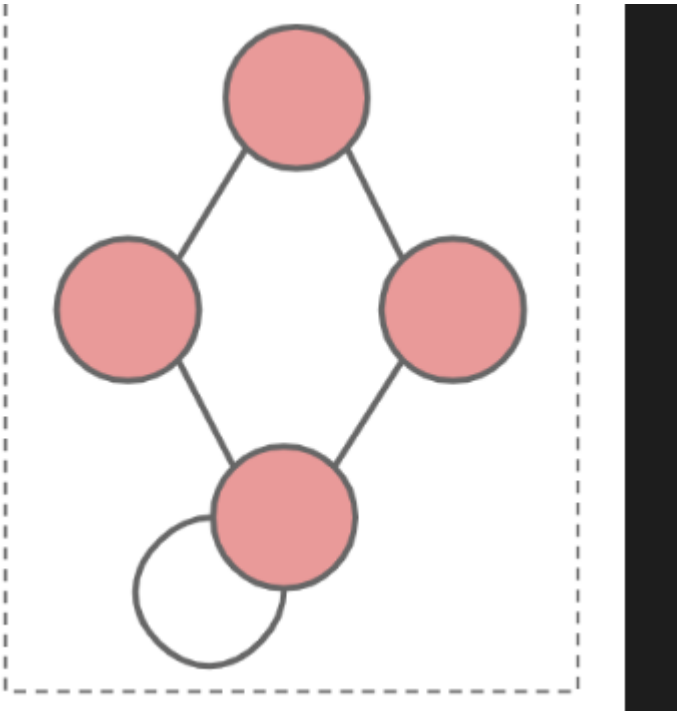
3.

最大堆（max heap）或最小堆（min heap）是一个以数组表示的二叉树，堆是完全二叉树

Lec21 Tree and Graph Traversals

1.simple graph:不允许self-loop，并且两个结点之间最多只有一条path

multigraph graph:两个结点之间有两条path，可以有self-loop,也可以没有self-loop



这个就是self-loop（即指向自身的）

Lec22 Graph Traversals and Implementations

1.

BFS：

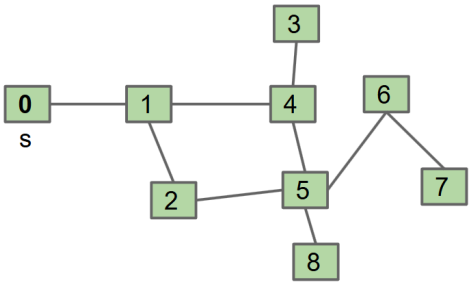
```
Initialize the fringe, an empty queue
    add the starting vertex to the fringe
    mark the starting vertex
    while fringe is not empty:
        remove vertex v from the fringe
        for each neighbor n of vertex v:
            if n is not marked:
                add n to fringe
                mark n
                set edgeTo[n] = v
                set distTo[n] = distTo[v] + 1
```

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	F	-	0
1	F	-	-
2	F	-	-
3	F	-	-
4	F	-	-
5	F	-	-
6	F	-	-
7	F	-	-
8	F	-	-



Queue: []

BFS的fringe是一个队列

2.

DFS

```
Initialize the fringe, an empty stack
    push the starting vertex on the fringe
    while fringe is not empty:
        pop a vertex off the fringe
        if vertex is not marked:
            mark the vertex
            visit vertex
            for each neighbor of vertex:
                if neighbor not marked:
                    push neighbor to fringe
```

DFS的fringe是一个stack

3.

需要注意的是，**DFS 和 BFS 不仅仅是在 fringe（前沿节点结构）上不同**，它们在标记节点的顺序上也不同。

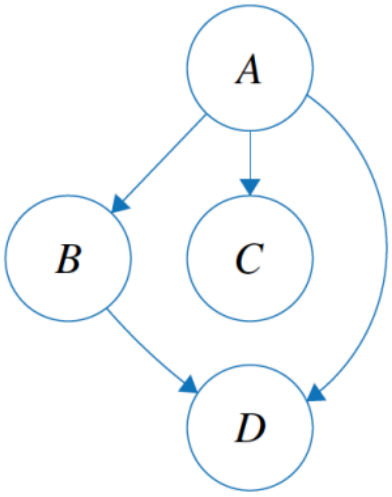
- 对于 **DFS**，我们是在访问节点时才标记它——也就是说，只有当节点从 fringe 中弹出时才会标记它。因此，如果某个节点已经被加入栈但尚未被访问，它可能会出现在栈中多个位置。
- 对于 **BFS**，我们一旦将节点加入 fringe 就立即标记它，这样就不会出现同一个节点在队列中重复出现的情况。

标记顺序很重要，下面给一个例子

(b) Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

First, identify the bug in this implementation. Then, give an example of a graph where this algorithm may not traverse in DFS order.
Hint: When should we be marking vertices?



For the graph above, it's possible to visit in the order $A - B - C - D$ (which is not depth-first) because D won't be put into the fringe after visiting B, since it's already been marked after visiting A. One should only mark nodes when they have actually been visited, but in this buggy implementation, we mistakenly mark them before we visit them, as we're putting them into the fringe.

4.

对于我们的图（Graph）API，我们使用一个常见的约定：将每个唯一节点分配一个整数编号。这可以通过维护一个映射（map）来实现，该映射能够告诉我们每个原始节点标签对应的整数编号。这样，我们就可以将 API 定义为专门处理整数，而不需要引入泛型

5.

DFS/BFS on a graph backed by adjacency lists runs in O(V+E), while on a graph backed by an adjacency matrix runs in O(V^2)

6.

DFS 前序遍历（DFS preorder）：按照 DFS 调用每个顶点的顺序。

DFS 后序遍历（DFS postorder）：按照 DFS 从每个顶点返回的顺序。

Lec23 Shortest Paths

Dijkstra's Algorithm

1.

Dijkstra 算法 接收一个输入顶点 **s**，并输出从 **s** 出发的最短路径树。它是如何工作的？

1. 创建一个 **优先队列**（priority queue）。
2. 将 **s** 加入优先队列，并设置优先级为 **0**；将所有其他顶点加入优先队列，并设置优先级为 ∞（无穷大）。
3. 当优先队列不为空时：
    - 从优先队列中弹出一个顶点。
    - 放松（relax）从该顶点出发的所有边。

2.

Dijkstra's Algorithm适用于所有边都是non-negative

3.

Dijkstra's Algorithm 的伪代码

```
def dijkstras(source):
    PQ.add(source, 0)
    For all other vertices, v, PQ.add(v, infinity)
    while PQ is not empty:
        p = PQ.removeSmallest()
        relax(all edges from p)
```

```
def relax(edge p,q):
   if q is visited (i.e., q is not in PQ):
       return

   if distTo[p] + weight(edge) < distTo[q]:
       distTo[q] = distTo[p] + w
       edgeTo[q] = p
       PQ.changePriority(q, distTo[q])
```

A* Algorithm

1.

源点到该节点的真实距离 + 从该节点到目标节点的估计距离

2.

heuristics need to be good. There are two definitions required for goodness.

1. Admissibility. heuristic(v, target) ≤trueDistance(v, target). (Think about the problem above. The true distance from the neighbor of C to C wasn't infinity, it was much, much smaller. But our heuristic said it was ∞, so we broke this rule.)
2. Consistency. For each neighbor vof w:
    - heuristic(v, target) ≤ dist(v, w) + heuristic(w, target)
    - where dist(v, w) is the weight of the edge from v to w.

Summary

> 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)

Lec24 Minimum Spanning Trees

Prim‘s Algorithm

1.

Prim 算法是一种从图中找到最小生成树（MST）的方法，其步骤如下：

1. 从任意一个节点开始。
2. 重复以下步骤：加入一条**最短的边**，这条边必须有一个节点在当前构建中的 MST 内，另一个节点在 MST 外。
3. 重复直到 MST 中有 V - 1条边为止（其中 V是顶点的数量）。

2.

它的运行方法和dijkstra是差不多的，可以直接看demo，唯一的区别在于，加入候选节点到fringe（priority queue）时，不是基于它们到目标顶点的距离，而是基于它们到当前构建的 MST 的距离。所以它的时间复杂度也是add是O(VlogV),removeSmallest(VlogV),changePriority(ElogV),所以最后的时间复杂度是O(ElogV)

Kruskal's Algorithm

1.

这个算法的步骤如下：

1. 将图中所有的边按权重从小到大排序。
2. 按排序好的顺序一次取出边，将其加入当前构建的最小生成树（MST），**前提是加入它不会形成环。**
3. 重复此过程，直到 MST 中有 V - 1条边（其中 V是顶点的数量）。

2.

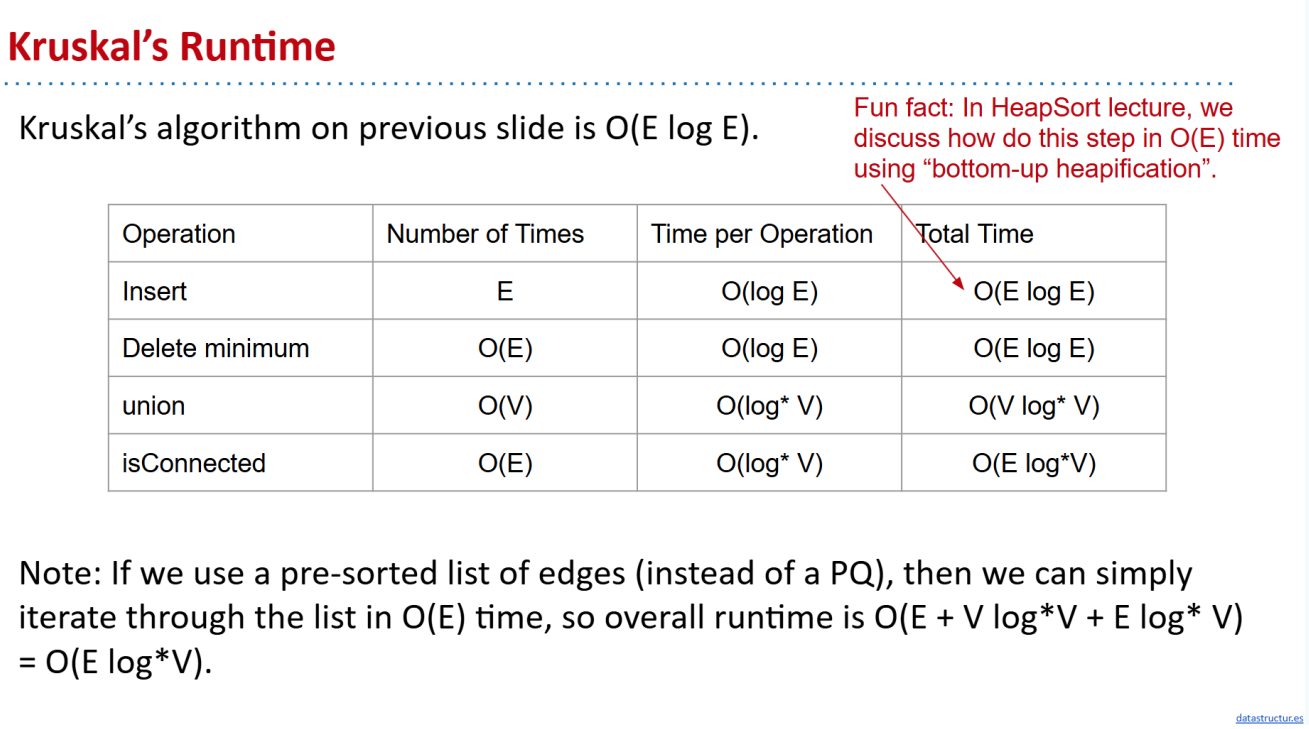
需要注意的是，Kruskal 算法得到的最小生成树（MST）可能**不**会与 Prim 算法得到的完全相同，但**两种算法都会返回一个 MST。**

由于它们得到的都是最小（最优）的生成树，因此它们都会给出有效且最优的结果（它们的总权重是相同的）。

3.

运行方法直接看demo

4.



如果没有pre-sorted list of edges，则是用priority queue来维持边的权重的大小关系的，因此时间复杂度是O(ElogE)+O(ElogE)+O(Vlog*V) + O(Elog星V)，所以时间复杂度是O(ElogE)

Lec25 Range Searching and Multi-Dimensional Data

QuadTree(四叉树)

一个节点底下分为四个，西北，东北，东南，西南

适用于二维空间，因为在二维空间中有四个象限

K-D Tree

将层次化划分的思想推广到二维以上的一种方法是使用 **K-D 树（K-D Tree）**。它的做法是**逐层轮流使用不同的维度来划分。**

在二维情况下，它在第一层像基于 X 的树那样划分；第二层像基于 Y 的树那样划分；第三层再回到基于 X 的树；第四层再是基于 Y 的树，以此类推。

在三维情况下，划分会在三个维度之间每三层循环一次；更高维度的情况也依此类推。K-D 树的一个优势就在于它能够更容易推广到高维空间。不过，无论维度多高，K-D 树始终是一棵**二叉树**，因为在每一层，空间都只会被划分为 “大于” 和 “小于” 两个部分。

Nearest Neighbor using a K-D Tree

为了找到与查询点最近的点（最近邻），我们在 K-D 树中按照下面的步骤进行：

1. 从根节点开始，把该点记为“**目前为止的最佳点**”。计算它到查询点的距离，并将该距离记为“**要打破的分数**（score to beat）”。在上图中，我们从 A 开始，A 到被标记点的距离是 4.5。
2. 该节点将周围空间划分为两个子空间。对于每个子空间，问自己：“这个子空间里是否可能存在一个更接近查询点的点？”可以通过计算查询点到该子空间边界的**最短距离**来回答这个问题（见下图的紫色虚线）。
3. 对于每个被判定为可能包含更好点的子空间，递归地继续上述过程。
4. 最终，记录下来的“目前为止的最佳点”就是最近邻——即距离查询点最近的点。

Lec26 Prefix Operations and Tries

Character Keyed Map

```
public class DataIndexedCharMap<V> {
    private V[] items;
    public DataIndexedCharMap(int R) {
        items = (V[]) new Object[R];
    }
    public void put(char c, V val) {
        items[c] = val;
    }
    public V get(char c) {
        return items[c];
    }
}
```

Trie

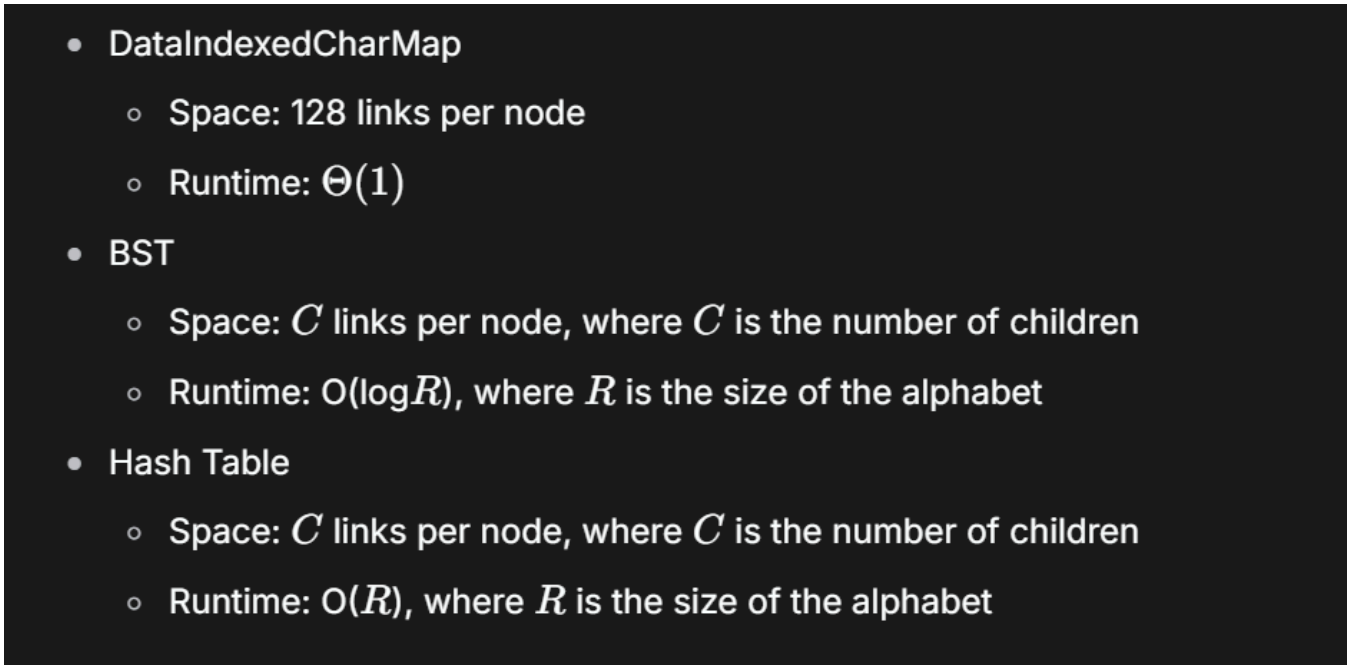
1.

通过它在父节点的 **DataIndexedCharMap** 中的位置来确定字符值(即ch实例变量)

```
public class TrieSet {
   private static final int R = 128; // ASCII
   private Node root;    // Trie 的根节点

   private static class Node {
      // 移除了 'ch' 实例变量
      private boolean isKey;
      private DataIndexedCharMap next;

      private Node(boolean blue, int R) {
         isKey = blue;
         next = new DataIndexedCharMap<Node>(R);
      }
   }
}
```

To address the issue of wasted space, let us explore two possible solutions:

- *Alternate Idea #1*: Hash-Table based Trie. This won't create an array of 128 spots but instead initialize the default value and resize the array only when necessary with the load factor.
- *Alternate Idea #2*: BST based Trie. Again this will only create children pointers when necessary, and we will store the children in the BST. We will have to worry about the runtime for searching in this BST, but this is not a bad approach.



2.

Trie是一种具体的数据结构的实现，它可以用来实现map或set

3.

Know how to insert and search for an item in a Trie. Know that Trie nodes typically do not contain letters, and that instead letters are stored implicitly on edge links. Know that there are many ways of storing these links, and that the fastest but most memory hungry way is with an array of size R. We call such tries R-way tries.

Lec28 Reduction and Decomposition

Topological Sorts and DAGs

1.

Topological Sort: an ordering of a graph's vertices such that for every directed edge ***u***→***v***, ***u*** comes before ***v*** in the ordering.

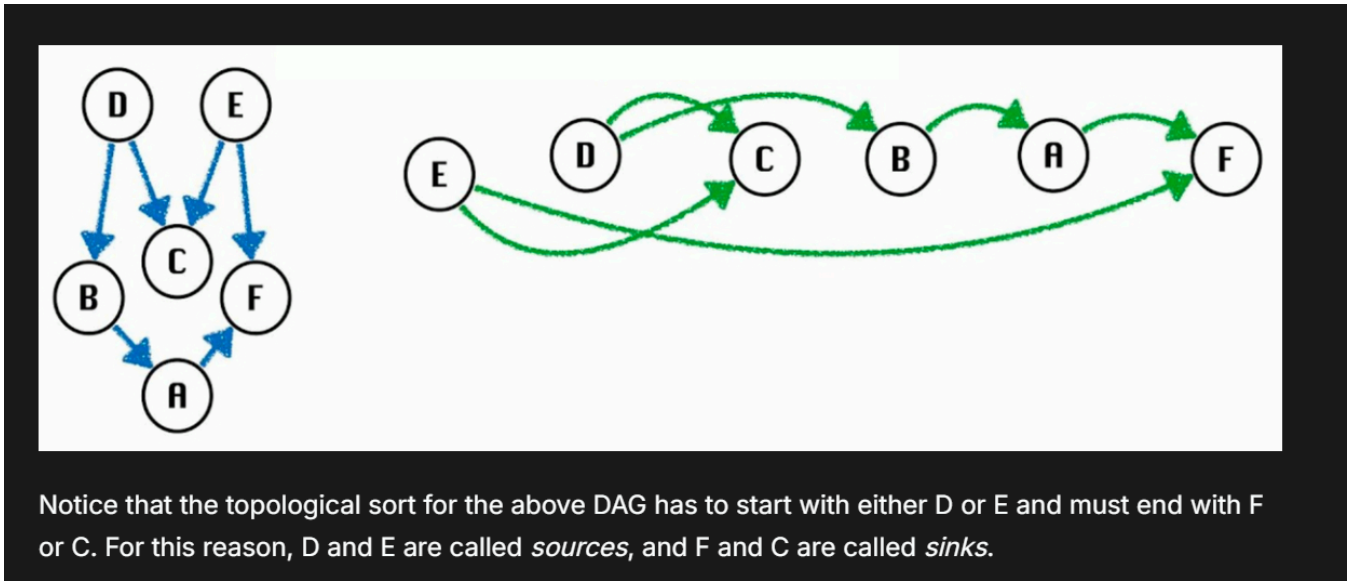
2.

topological sorts only apply to **directed, acyclic (no cycles) graphs** - or **DAG**s.

3.

Topological Sort: an ordering of a **DAG**'s vertices such that for every directed edge ***u***→***v***, ***u*** comes before ***v*** in the ordering.

4.



5.

Topological Sort Algorithm:

- Perform a DFS traversal from every vertex in the graph
- Record DFS postorder along the way.
- Topological ordering is the reverse of the postorder.

所以时间复杂度是O(V+E),其中V是图中顶点的数量，E是边的数量

```
topological(DAG):
    initialize marked array
    initialize postOrder list
    for all vertices in DAG:
        if vertex is not marked:
            dfs(vertex, marked, postOrder)
    return postOrder reversed

dfs(vertex, marked, postOrder):
    marked[vertex] = true
    for neighbor of vertex:
        dfs(neighbor, marked, postOrder)
    postOrder.add(vertex)
```

上述这个代码用的dfs其实是用了stack的，只不过我们之前是用的遍历，这次用的递归

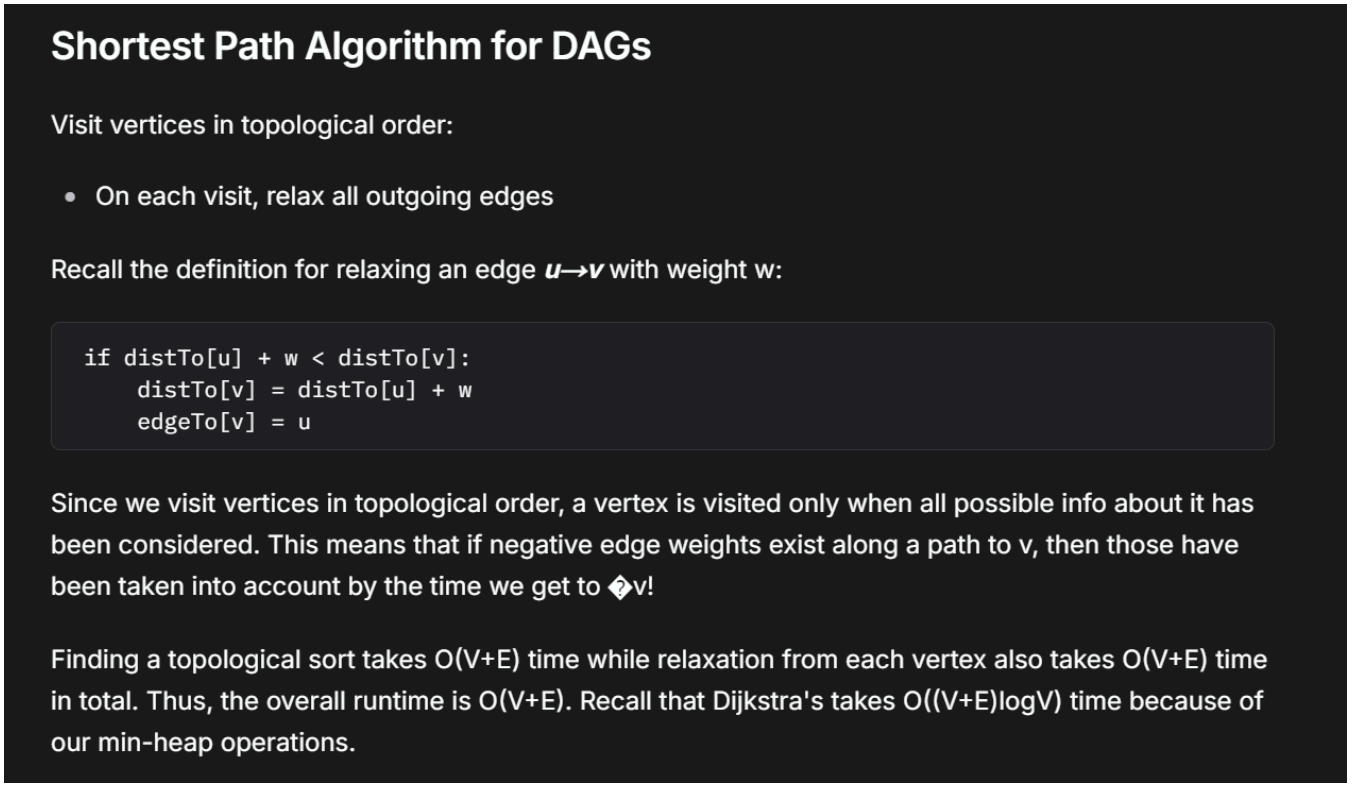
BFS也可以做（但超出了课程的要求范围）

We can find a topological sort of any DAG in **O(V+E)** time using **DFS** (or **BFS**).

Shortest Path on DAGs

1.这个方法不像dijkstra算法一样，它可以处理negative edges，而dijkstra只能处理不是negative edges的图

2.



所以shortest path algorithm for dags的时间复杂度是O(V+E)

Longest Paths on DAGS

第一种方法：

1. Form a new copy of the graph, called G', with all edge weights negated (signs flipped).

2. Run DAG shortest paths on G' yielding result X
3. Flip the signs of all values in X.distTo. X.edgeTo is already correct.

第二种方法

we could modify the DAG shortest path algorithm from the previous section to choose the larger distTo when relaxing an edge.

Reductions and Decomposition

1.

if any subroutine for task Q can be used to solve P, we say P reduces to Q.

例如上面的Longest Paths on DAGs，Since DAG-SPT can be used to solve DAG-LPT, we say that "DAG-LPT reduces to DAG-SPT."

Lec29 Basic Sorts

Selection Sort & Heapsort

Naive Heapsort

1.

to heapsort N items, we can insert all the items into a max heap and create and output array. Then, we repeatedly delete the largest item from the max heap and put the largest item at the end part of the output array.

2.

The overall runtime of this algorithm is Θ(NlogN). There are three main components to this runtime:

- Inserting N items into the heap: O(NlogN).
- Selecting the largest item: Θ(1)
- Removing the largest item: O(logN)

In-place Heapsort

1.

As an alternate approach, we can use the input array itself to form the heap and output array.

Rather than inserting into a new array that represents our heap, we can use a process known as *bottom-up heapification* to convert the input array into a heap. Bottom-up heapification involves moving in reverse level order up the heap, sinking nodes to their appropriate location as you move up.

By using this approach, we avoid the need for an extra copy of the data. Once heapified, we use the naive heapsort approach of popping off the maximum and placing it at the end of our array. In doing so, we maintain an "unsorted" front portion of the array (representing the heap) and a "sorted" back portion of the the array (representing the sorted items so far).

可以看demo

2.

This process overall is still O(NlogN), since bottom-up heapification requires at most N sink-down operations that take at most logN time each.

Note: it is possible to prove that bottom-up heapficiation is bounded by Θ(N) *. However, this proof is out of scope for this class.*

Mergesort

1. Split the items into half.
2. Mergesort each half.
3. Merge the two sorted halves to form the final result.

Mergesort has a runtime of Θ(NlogN)

The auxiliary array used during the merge step requires Θ(N)extra space. Note that in-place mergesort is possible; however it is very complex and the runtime suffers by a significant constant factor, so we will not cover it here.

可以看demo

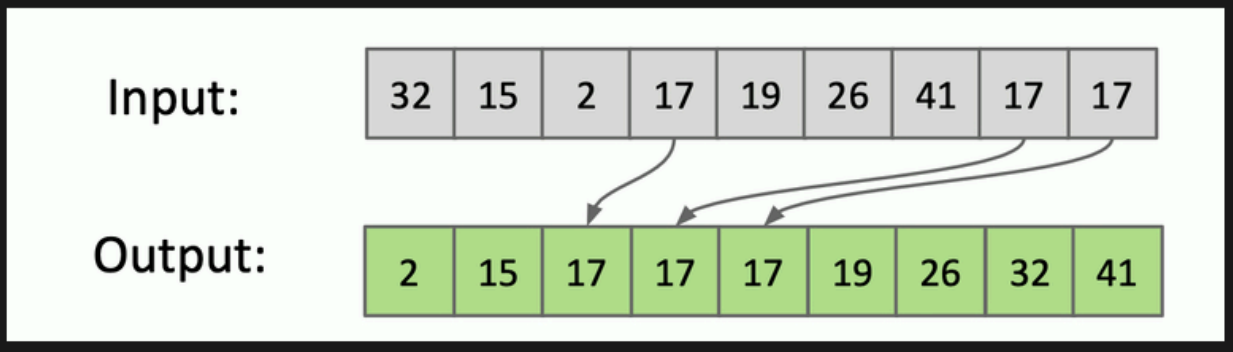
Insertion Sort

Naive Insertion Sort

Naive Insertion Sort

In insertion sort, we start with an empty output sequence. Then, we select an item from the input, inserting into the output array at the correct location.

Naively, we can do this by creating a completely seperate output array and simply putting items from the input there. You can see a demo of this algorithm here ↗.



Naive insertion sort with a seperate output array

For the naive approach, if the output sequence contains k items so far, then an insertion takes $O(k)$ time (shifting every item over in the output array).

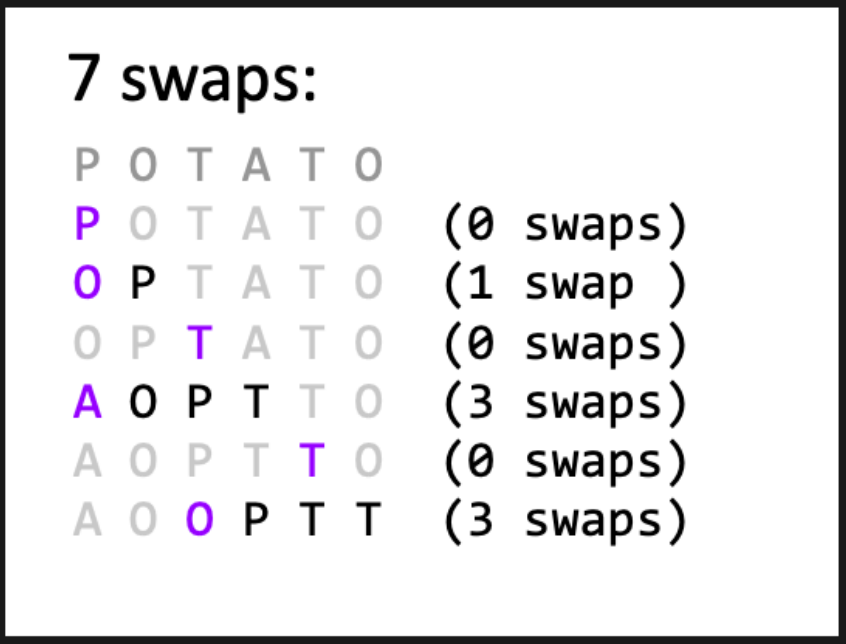
In-place Insertion Sort

In-place Insertion Sort

We can improve the time and space complexity of insertion sort by using in-place swaps instead of building a seperate output array.

You can see a demo of this algorithm here ↗.

Essentially, we move from left to right in the array, selecting an item to be swapped each time. Then, we swap that item as far to the front as it can go. The front of our array gradually becomes sorted until the entire array is sorted.



Insertion sort algorithm. Purple items are the selected item being swapped to the front, and black items are the ones with which the purple items are being swapped.

Insertion Sort Runtime

插入排序的最佳情况运行时间是
$\Theta(N)$ —— 当数组已经有序时，不需要进行交换，我们只需线性扫描整个数组。
插入排序的最坏情况运行时间是
$\Theta(N^2)$ —— 当数组是逆序排列时，每个元素都必须一路交换到最前面。

插入排序的优点
需要注意的是，在已经有序或接近有序的数组上，插入排序所需的工作量非常少。实际上，它执行的交换次数等于数组中的**逆序对**数量。换句话说，对于逆序对数量较少的数组，插入排序可能是最快的排序算法。其运行时间是
$\Theta(N+K)$，其中 K 是数组中的逆序对个数。
如果我们将"接近有序"的数组定义为：逆序对数量满足 $K < cN$（其中 c 是某个常数），那么插入排序可以在**线性时间**内完成。

另一个不太直观但经过实验证明的事实是：在小规模数组上（通常大小为 15 或更少），插入排序非常快。
这一点的理论分析超出了本课程的范围，但总体思路是：分治类算法（例如堆排序和归并排序）在"划分"阶段花费了过多的时间，而插入排序则是直接开始排序。
事实上，Java 的归并排序实现中，当分割后的子数组大小小于 15 时，会直接改用插入排序。

Insertion Sort 每一次交换都会修复恰好一个逆序对

Lec30 Quick Sort

1.

Quicksort Algorithm

To quicksort N items:

1. Partition on the leftmost item as the pivot.
2. Recursively quicksort the left half.
3. Recursively quicksort the right half.

可以看demo

2.

巧合的是，在大多数常见情况下，快速排序在实测中确实是最快的排序算法。

最佳情况（Best Case）

例如，最佳情况是每次选择的枢轴都恰好落在数组中间（换句话说，每次划分都能选到数组的中位数）。

在这种情况下，每一层的工作量大约是 $O(N)$。

- 第一次划分需要检查所有 N 个元素，以决定它们应该放在枢轴的左边还是右边。
- 第二层会重复这一过程，不过是对两个子数组分别进行（每个子数组大约 $N/2$ 个元素），所以这一层的总工作量依然是 $O(N)$。

因此，每一层的工作量都是 $O(N)$。

总的运行时间为 $\Theta(NH)$，其中 H 表示层数。最佳情况时 $H = \Theta(\log N)$，所以快速排序的总运行时间是

$$\Theta(N \log N).$$

最坏情况（Worst Case）

最坏情况发生在每次划分时，枢轴都落在数组的开头。

在这种情况下：

- 每一层仍然要做 $O(N)$ 的工作；
- 但这次层数 $H = N$，因为每个元素都要依次作为枢轴。

因此，最坏情况下快速排序的运行时间是

$$\Theta(N^2).$$

In comparison, Mergesort seems to be a lot better, since Quicksort suffers from a theoretical worst case runtime of Θ(N^2). So how can Quicksort be the fastest sort empirically? Quicksort's advantage empirically comes from the fact that on average, Quicksort is Θ(NlogN)

如何避免快速排序的最坏情况

如上所示，快速排序的性能（无论是增长阶还是常数因子）都强烈依赖于：

- 你如何选择枢轴；
- 你如何围绕枢轴进行划分；
- 以及你是否加入了其他优化手段来加快排序。

在某些条件下，快速排序可能会退化到 $\Theta(N^2)$，这远比 $\Theta(N \log N)$ 差。
这些条件包括：

- **不好的输入顺序**：例如数组已经完全有序（或几乎有序）；
- **不好的元素分布**：例如数组中元素全部相同。

Randomization. Accept (without proof) that Quicksort has on average Θ(NlogN) runtime. Picking a random pivot or shuffling an array before sorting (using an appropriate partitioning strategy) ensures that we're in the average case.

Lec32 More Quick Sort, Sorting Summary

Quicksort Flavors vs. MergeSort

Quicksort Flavors： Philosophies

Philosophy 1 : Randomness

the first method is pick pivots randomly.

the second method is shuffle items before sort.

Philosophy 2 : Smarter Pivot Selection

To the problem is to choose a "good" pivot effectively

the first method is constant time pivot pick:

One of such approach is to pick a few items, and then among them, choose the "best" one as the pivot. This type of approach exemplifies a type of pivot selection procedure that is both *deterministic* and *constant* time.

However, a big drawback of such procedure is that the resulting Quicksort has a family of "dangerous inputs"---inputs that will lead to the worst case runtime or break the algorithm---that an adversary could easily generate.

the second method is Linear time pivot pick

Since we are partioning multiple times, always selecting the median as the pivot seems like a good idea, because each partition will allow us to cut the size of the resulting subarrays by half.

Therefore, to improve upon the idea from the first method, we can calculate the *median* of a given array, which can be done in linear time, then select that as the pivot.

The "exact median QuickSort" algorithm will not have the technical vulnerabilities as the first method approach, and has a runtime of θ(NlogN).(这个exact median Quicksort可以用我们下面提到的Quick Select) However, it is still slower than MergeSort.

Philosophy 3: Introspection

This philosophy simply relies on introspecting the recursive depth of our algorithm, and switch to MergeSort once we hit the depth threshold.

Although this is a reasonable approach, it is not common to use in practice.

Quicksort Falvors vs. MergeSort: Who's the best

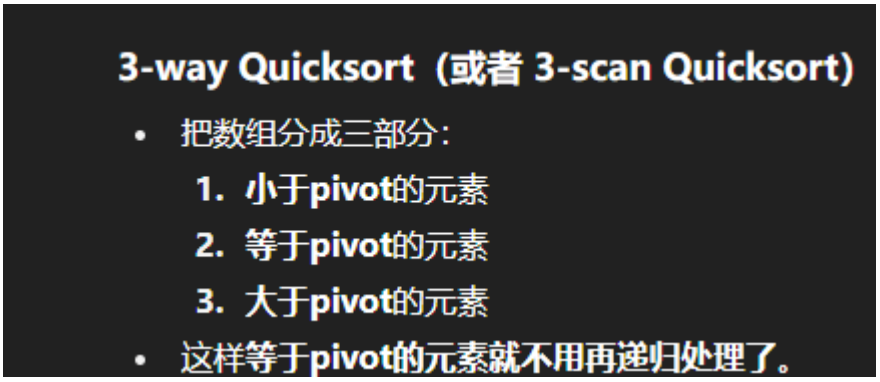
Candidate One: Quicksort L3S

Quicksort L3S is the Quicksort that is introduced last time, with the following properties:

- Always leftmost pivot selection
- Partioning
- Shuffle before starting

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	2.1 seconds
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds

Unfortunately, 0 : 1 with Mergesort in the lead.



Candidate Two:Quicksort LTHS

Quicksort LTHS is very similar to Quicksort L3S, except with a different partioning scheme: **Tony Hoare's In-Place Partioning Scheme**.

Tony Hoare's Partioning

Imagine two pointers, one at each end of the items array, walking towards each other:

- Left pointer loves small items.
- Right pointer loves large items.
- Big idea: Walk towards each other, swapping anything they don’t like; stop when the two pointers cross each other.
    - Left pointer hates larger or equal items
    - Right pointer hates smaller or equal items
- New pivot = Item at right pointer.
- End result is that things on left of pivot (originally the leftmost item) are “small” and things on the right of pivot are “large”.

可以看demo

It's important to also note that:

- Faster schemes have been found since.
- Overall runtime still depends crucially on pivot selection strategy!

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	2.1 seconds
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds

Candidate Three: QuickSort PickTH

Recall from philosophy 2b that the idea of identifying the median and use it as the pivot is inefficient because finding the median itself is a costly process. (Runtime: θ(NlogN)

Turns out, it is possible to find the median in θ(N) time instead, with an algorithm called "PICK".

Will this improved version of Exact Median Quicksort perform better than Mergesort?

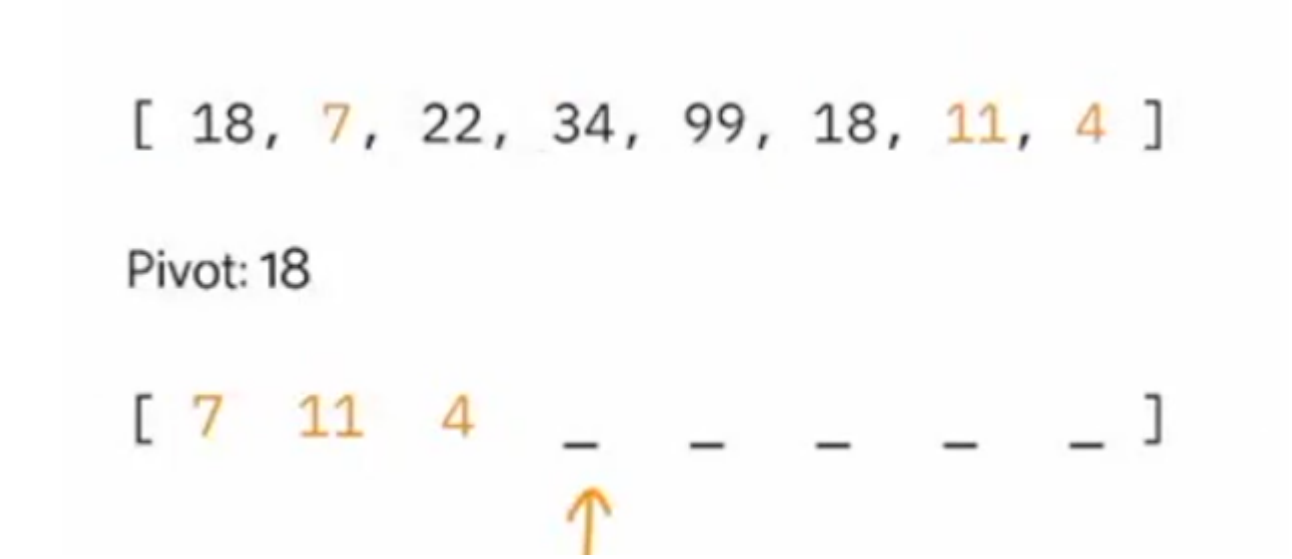
	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	2.1 seconds	Θ(N log N)
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds	$\Theta(N^2)$
Quicksort PickTH	Exact Median	Tony Hoare	Exact Median	10.0 seconds	Θ(N log N)

Sadly, no. And it was terrible! Likely for the following reasons:

- Cost to compute medians is too high.
- Have to live with worst case $\theta(N^2)$ if we want good practical performance.

1 : 2 with Mergesort as the winner.

3-scan partitioning one example：



the 3 scan partitioning process is divided into three:the first is less than the pivot, the second is equal to the pivot,the third is greater than the pivot

3 Way Partitioning or 3 scan partitioning is a simple way of partitioning an array around a pivot. You do three scans of the list, first putting in all elements less than the pivot, then putting in elements equal to the pivot, and finally elements that are greater. This technique is NOT in place.

Hoare Partitioning is a very fast in-place technique for partitioning.

Quick Select

this algorithm is to identify the median

Goal: find the median using partioning.

Key idea: Median of an length n array will be around index n /2.

1. Initialize array with the leftmost item as the pivot.

Initial Array

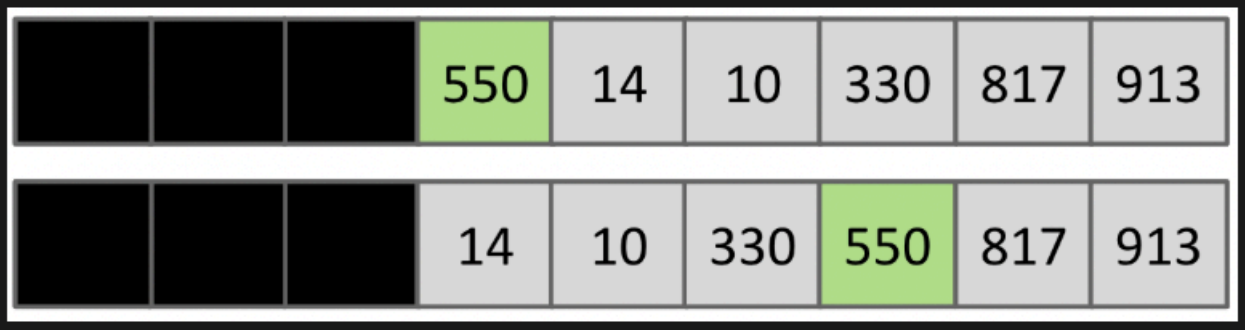
For this example, the index of the median should be 9 / 2 = 4.

2. Partition around pivot.

Pivot lands at index 2

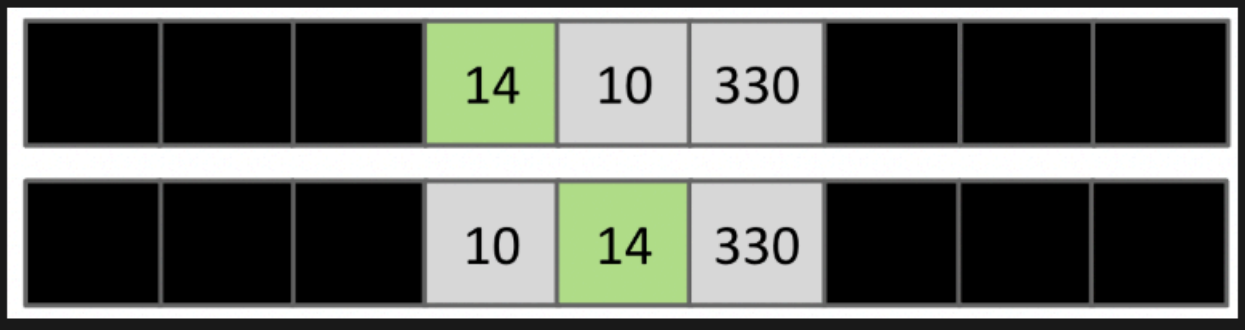
Since the pivot is at index 2, which is not the median. Therefore, need to continue. However, will only partition the **right** subproblem because median can't be to the left! (index n/2 > 2)

3. Partition the subproblem. Repeat the process.



Pivot is at index 6, which means that it is not at the median. Continue.

4. Stop when the pivot is at the median index.



Since pivot is at index 4, we are done.

Runtime

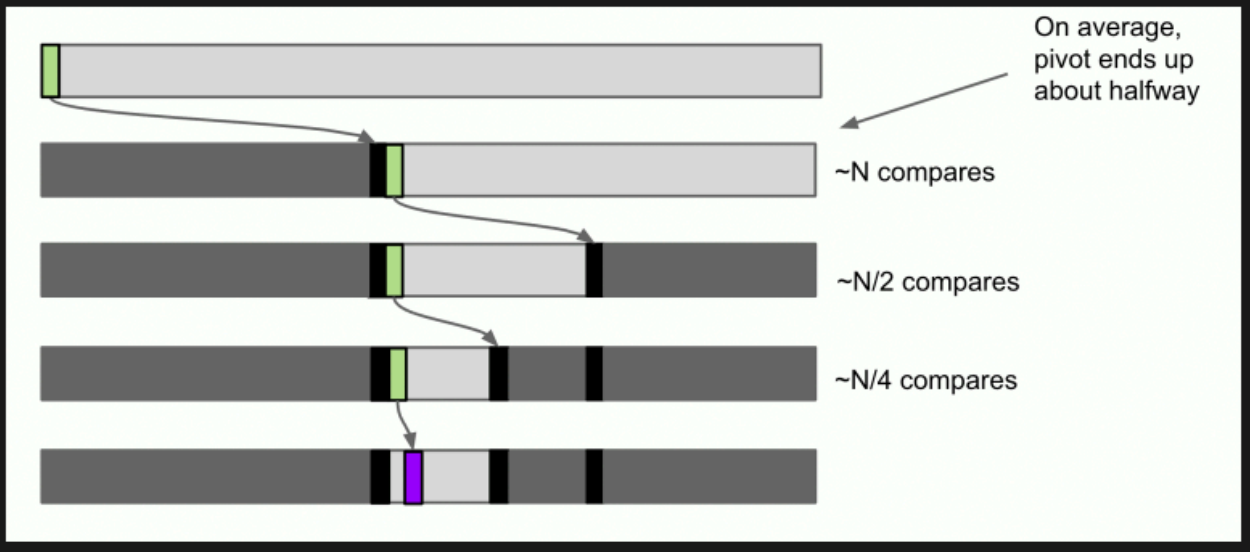
Worst Case Runtime

The worst case is when the array is in sorted order, which yields a runtime of $\theta(N^2)$.

[1 2 3 4 5 6 7 8 9 10 … N]
[1 **2 3 4 5 6 7 8 9 10 … N**]
[1 2 **3 4 5 6 7 8 9 10 … N**]
…
[1 2 3 4 5 … **N/2** … N]

Expected Runtime

The expected runtime for the Quick Select Algorithm is $\theta(N)$. Here's an intuitive diagram that justifies this result:



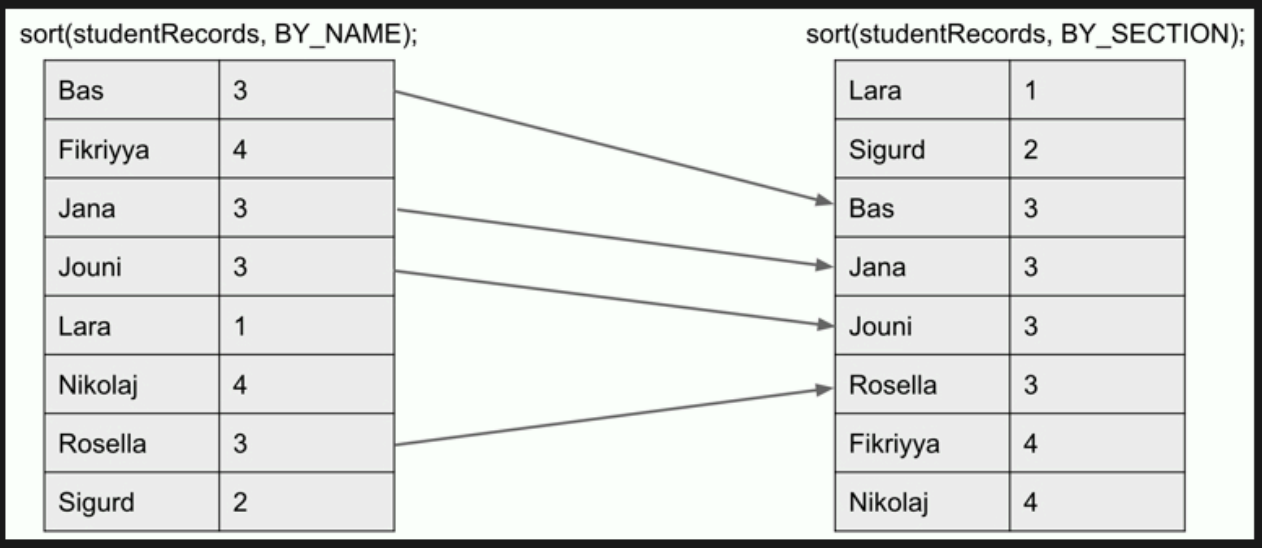
Using one of our favorite sums, we can see the runtime is:

$$N + N/2 + N/4 + ... + 1 = \theta(N)$$

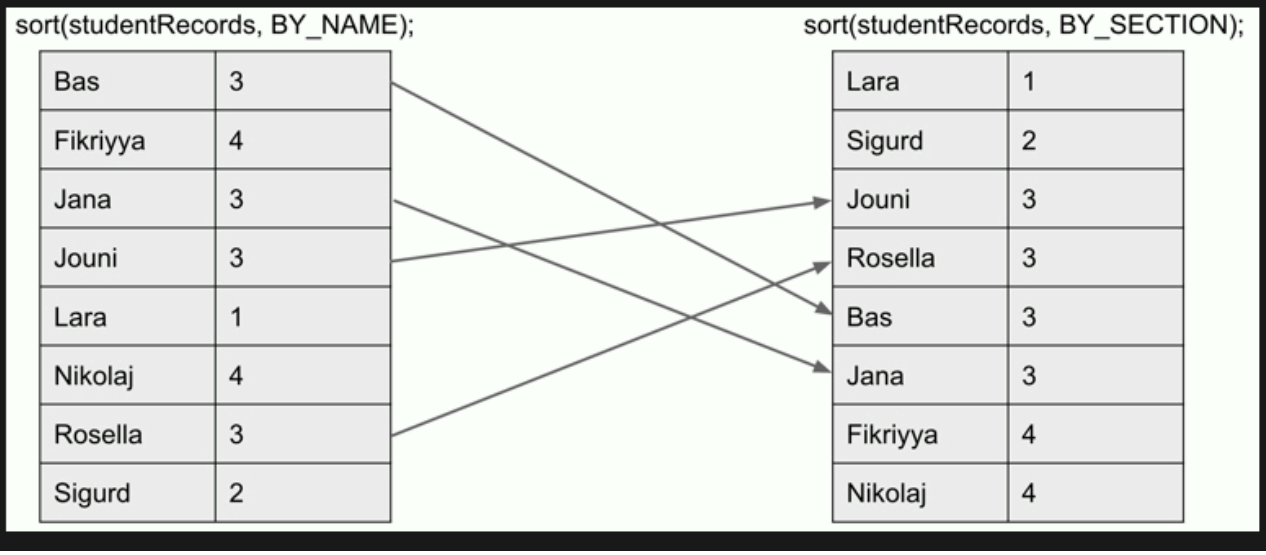
Stability, Adaptiveness, and Optimization

Stability

A sort is stable if the order of equivalent elements is preserved. The following is an example of a stable sort. After sorting by section, notice how Bas, Jana, Jouni, and Rosella are in the same order as before sorting. If we want records sorted by section and then by name within each section, we can sort by name and then by section as below.



The following example is an unstable sort. It can make things really annoying! If we want records sorted by section and then by name within each section, we can't just sort by name and then by section as before. After an unstable sort, the previous ordering is not maintained.



Insertion sort is stable!MergeSort is stable. HeapSort is not stable. QuickSort can be stable depending on its partitioning scheme, but its stability cannot be assumed since many of its popular partitioning schemes, like Hoare, are unstable.(即有的Quicksort是stable，有的是不稳定的)

Optimization

Adaptiveness - A sort that is adaptive exploits the existing order of the array. Examples are InsertionSort, SmoothSort, and TimSort.

Switch to Insertion Sort - When a subproblem reaches size 15 or lower, use insertion sort. It is very very fast for inputs of small sizes.

Exploit restrictions on set of keys - For example, if the number of keys is some constant, we can use this constraint to sort faster by applying 3-way QuickSort.(3-way QuickSort也叫3-scan QuickSort)

Switch from QuickSort - If the recursion goes too deep, switch to a different type of sort.

Exercise

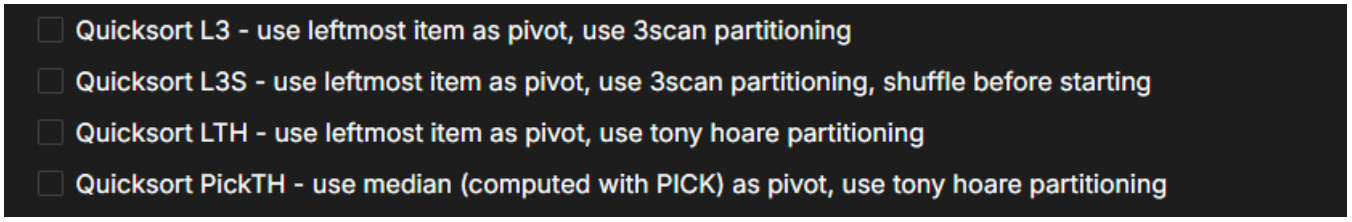
1.

Suppose we have the array `[17, 15, 19, 32, 2, 26, 41, 17, 17]`, and we partition it using 3-scan partitioning using `17` as the pivot. What array do we end up with?

Ans:

`[15, 2, 17, 17, 17, 19, 32, 26, 41]`. First we scan for the smaller elements (15, 2), then the equal elements (17, 17, 17), and finally the larger elements (19, 32, 26, 41) in the order they appear from left to right in the original array.

2.



3.

Why does Java's built-in `Array.sort` method use Quicksort for `int`, `long`, `char`, or other primitive arrays, but Mergesort for all `Object` arrays?

Ans:

This is because primitives don't require stability--an `int` is indistinguishable from any other `int` if they are equal by `.equals()`. However, this is not true for `Object`s, since two different `Object`s at different memory addresses can still be equal, and stability may be desireable when sorting objects.

Lec34 Sorting and Algorithmic Bounds

Sorting Summary

1.

In Java, Arrays.sort(someArray) uses:

- Mergesort (specifically the TimSort variant) if someArray consists of Objects.
- Quicksort if someArray consists of primitives.

`static void`	**`sort(Object[] a)`** Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.

`static void`	**`sort(int[] a)`** Sorts the specified array into ascending numerical order.

Why?

- Quicksort isn't stable, but there's only one way to order them. Wouldn't have multiple types of orders.
- Could sort by other things, say the sum of the digits.
- Order by the number of digits.
- My usual answer: 5 is just 5. There are no different possible 5's.
- When you are using a primitive value, they are the 'same'. A 4 is a 4. Unstable sort has no observable effect.
- There's really only one natural order for numbers, so why not just assume that's the case and sort them that way.
- By contrast, objects can have many properties, e.g. section and name, so equivalent items CAN be differentiated.

2.

	Memory	# Compares	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)	No
Insertion	Θ(1)	Θ(N^2)	Best for almost sorted and N < 15	Yes
Mergesort	Θ(N)	Θ(N log N)	Fastest stable sort	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. However, using unstable partitioning schemes (like Hoare partitioning) and using randomness to avoid bad pivots tend to yield better runtimes.

Math Problems Out of Nowhere

We've shown that $N * logN \in \Theta(log(N!))$.

- In other words, these two functions grow at the same rate asymptotically.

Theoretical Bounds on Sorting

1.

Let's say we place Tom the Cat, Jerry the Mouse, and Spike the Dog in opaque soundproof boxes labeled A, B, and C. We want to figure out which is which using a scale.

Let's say we knew that Box A weighs less than Box B, and that Box B weighs less than Box C. Could we tell which is which?

The answer turns out to be yes!

a < b	b < c		Which is which?
Yes	Yes		a: mouse, b: cat, c: dog (sorted order: abc)

We can find a sorted order by just considering these two inequalities. What if Box A weighs more than Box B, and that Box B weighs more than Box C? Could we still find out which is which?

a < b	b < c		Which is which?
Yes	Yes		a: mouse, b: cat, c: dog (sorted order: abc)
No	No		c: mouse, b: cat, a: dog (sorted order: cba)

The answer turns out to be yes! So far, we have been able to solve this game with just two inequalities! Let's go ahead and try a third case scenario. Could we know which is which if Box A weighs less than Box B, but Box B weighs more than Box C?

a < b	b < c		Which is which?
Yes	Yes		a: mouse, b: cat, c: dog (sorted order: abc)
No	No		c: mouse, b: cat, a: dog (sorted order: cba)
Yes	No		

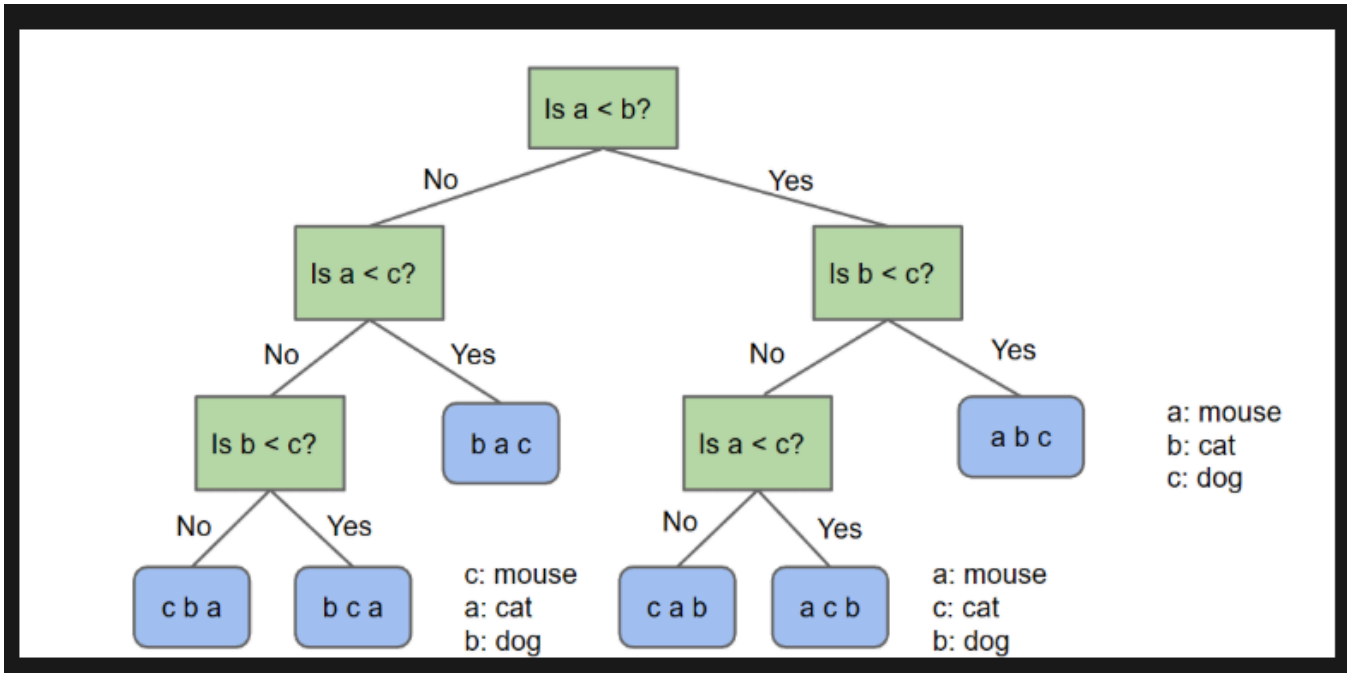
The answer turns out to be no. It turns out to have two possible orderings:

- a: mouse, c: cat, b: dog (sorted order: acb)
- c: mouse, a: cat, b: dog (sorted order: cab)

So while we were on a really great streak of solving the game with only two inequalities, we will need a third to solve all possibilities of the game. If we add the inequality a < c then this problem goes away and becomes solvable.

a < b	b < c	a < c?	Which is which?
Yes	Yes	N/A	a: mouse, b: cat, c: dog (sorted order: abc)
No	No	N/A	c: mouse, b: cat, a: dog (sorted order: cba)
Yes	No	Yes	a: mouse, c: cat, b: dog (sorted order: acb)

Now that we've created this table, we can create a tree to help us solve this sorting game of cat and mouse (and dog).



But how can we make this generalizable for all sorting? We know that each leaf in this tree is a unique possible answer to how boxes should be sorted. So the number of ways the boxes can be sorted should turn out to be the number of leaves in the tree. That then begs the question of how many ways can N boxes be sorted? The answer turns out to be N! ways as there are N! permutations of a given set of elements.

So how many questions do we need to ask in order to know how to sort the boxes? We would need to find the number of levels we must go through to get our answer in the leaf. Since it's a binary tree the minimum amount levels turn out lg(N!) levels to reach a leaf. (Note that lg means (log base 2)).

So, using this game we have found that we must ask at least lg(N!) questions about inequalities to find a proper way to sort it. Thus our lower bound for R(N) is Ω(lg(N!)). 而lg(N!)与 NlgN grow the same asymptote, so it's Ω(NlgN)

因此，**不存在**任何基于比较的排序算法，其最坏情况的增长阶（order of growth）能**优于**

$$\Theta(N \log N)$$

的比较次数。

换句话说：

> $N \log N$ 是**比较排序算法的理论极限下界**，
> 任何想比这更快的算法都必须**不是基于比较的**（例如计数排序、基数排序等）。

Lec35 Radix Sorts

Counting Sort

Counting sort:

- Count number of occurrences of each item.
- Iterate through list, using count array to decide where to put everything.
- Interactive Demo

可以看Demo

Counting Sort Runtime

Counting Sort Runtime Analysis

What is the runtime for counting sort on N keys with alphabet of size R?

- Treat R as a variable, not a constant.

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)

For ordered array. For counts and starting points.

Memory usage: Θ(N+R)

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

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

	Memory	Runtime	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)	No
Insertion	Θ(1)	$\Theta(N^2)$	Small N, almost sorted	Yes
Mergesort	Θ(N)	Θ(N log N)	Fastest stable	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

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

Counting sort is nice, but alphabetic restriction limits usefulness.

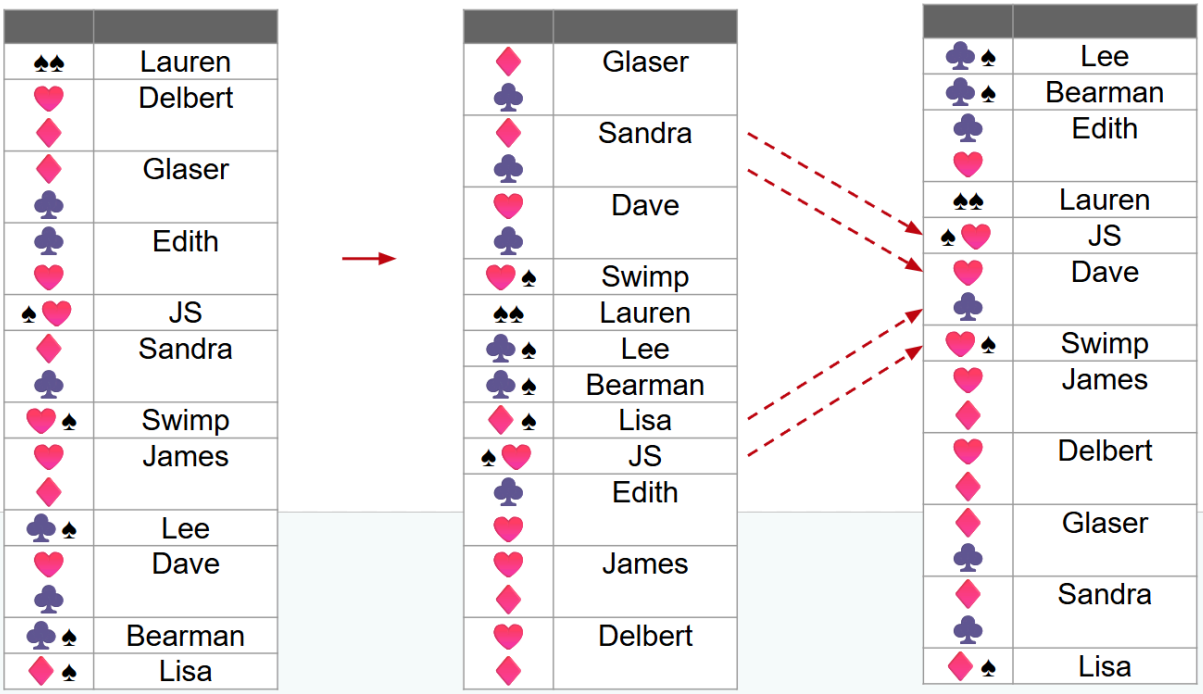
- No obvious way to sort hard-to-count things like Strings.

LSD Radix Sort

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

Sort each digit independently from rightmost digit towards left.

- Example: Over {♣, ♠, ♥, ♦}

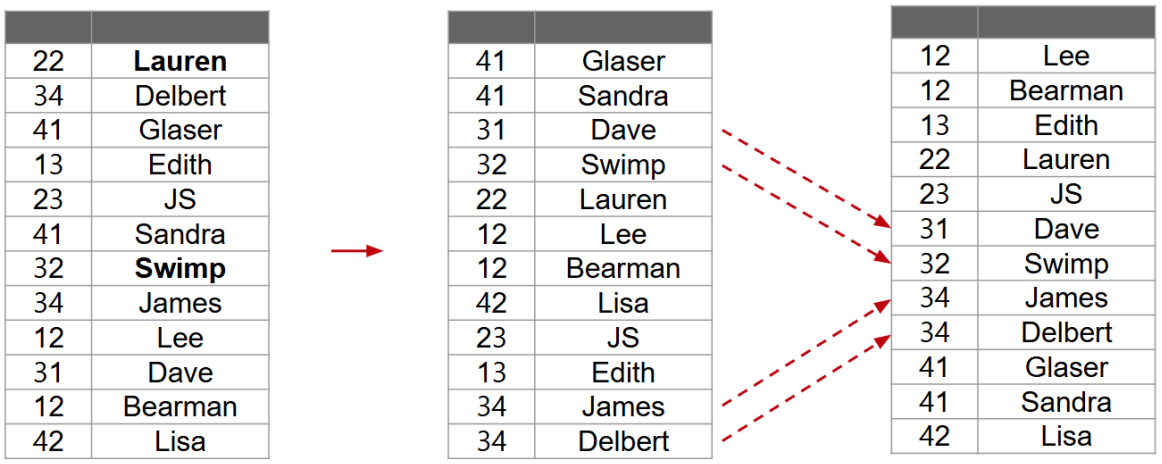


LSD (Least Significant Digit) Radix Sort -- Using Counting Sort

Sort each digit independently from rightmost digit towards left.

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

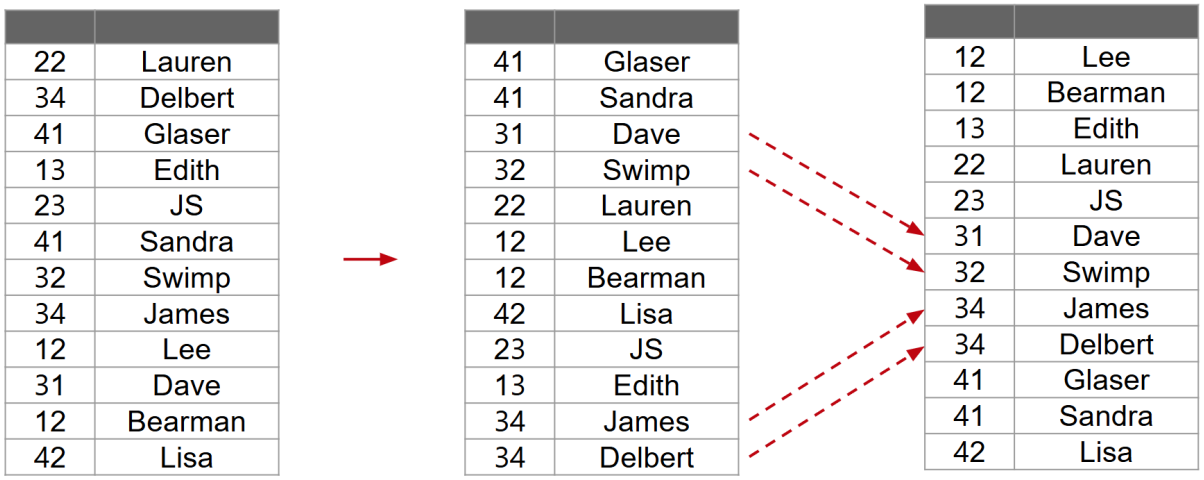
Note: Lauren and Swimp are backwards in middle column, same with Delbert and James, same with Edith and JS



LSD Runtime

What is the runtime of LSD sort?

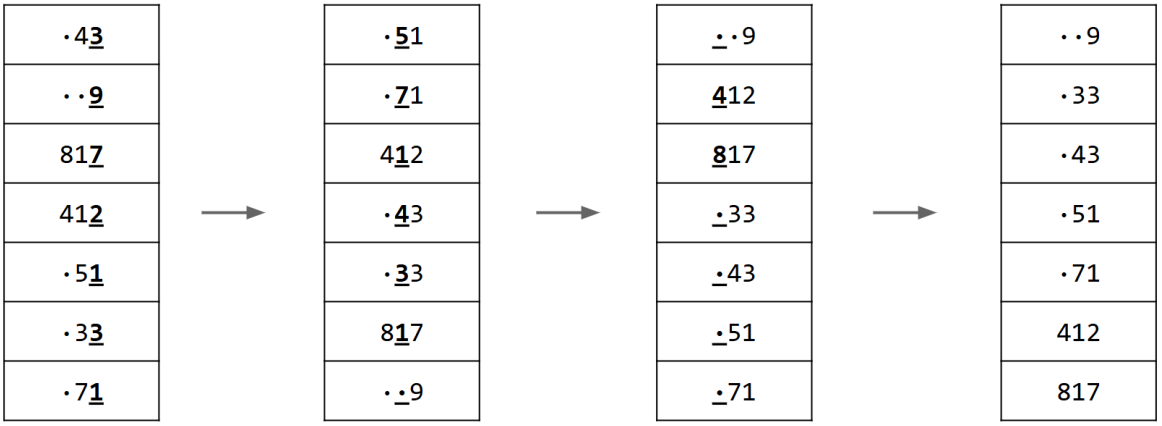
- Θ(WN+WR)
- N: Number of items, R: size of alphabet, W: Width of each item in # digits



即LSD runtime就是W次counting sort

Non-equal Key Lengths

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



Sorting Summary

W passes of counting sort: Θ(WN+WR) runtime.

- Annoying feature: Runtime depends on length of longest key.

	Memory	Runtime	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)*	Bad caching (61C)	No
Insertion	Θ(1)	$\Theta(N^2)$*	Small N, almost sorted	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

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

*: Assumes constant compareTo time.

MSD Radix Sort

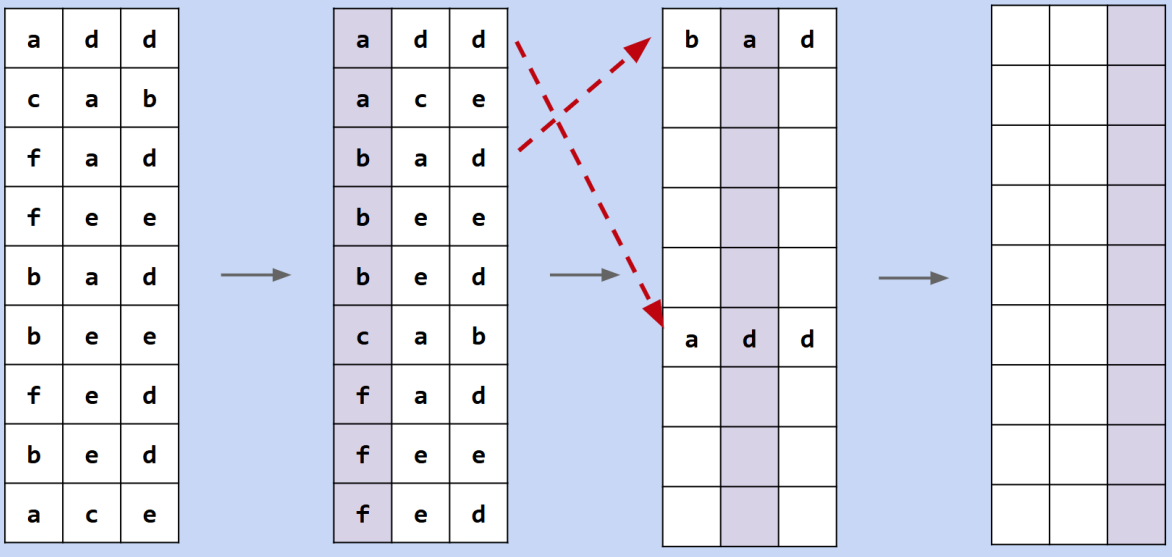
MSD (Most Significant Digit) Radix Sort

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

Pseudopseudohypoparathyroidism
Floccinaucinihilipilification
Antidisestablishmentarianism
Honorificabilitudinitatibus
Pneumonoultramicroscopicsilicovolcanoconiosis

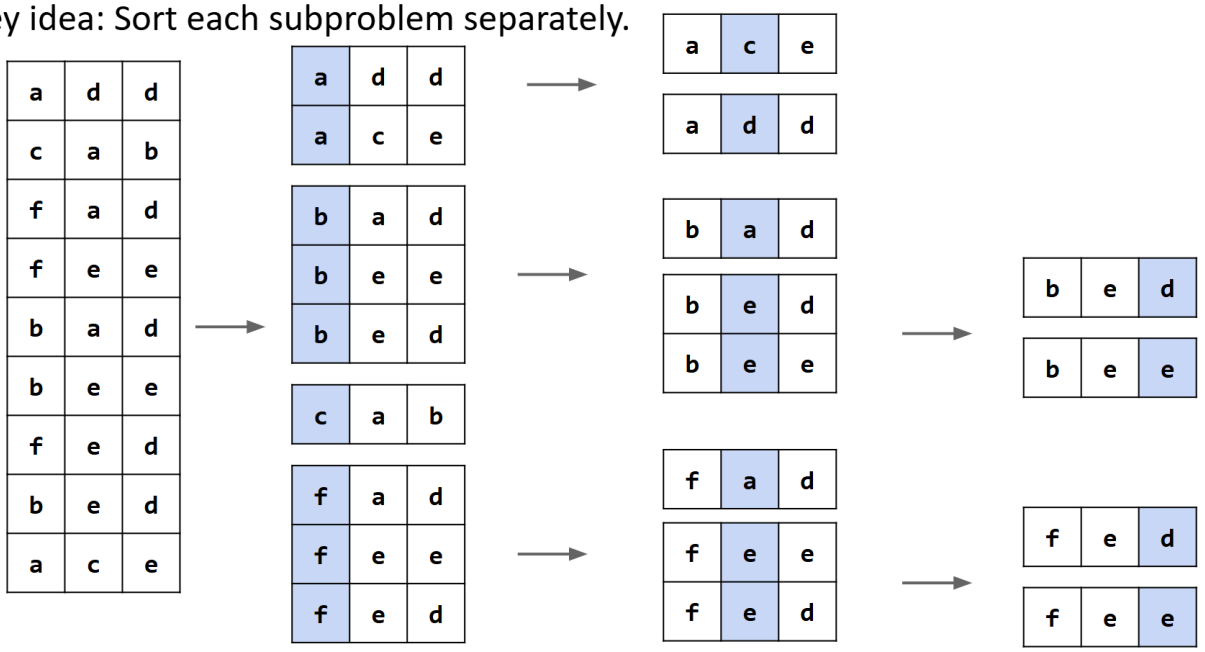
MSD Sort Question

Suppose we sort by topmost digit, then middle digit, then rightmost digit. Will we arrive at the correct result? A. Yes, **B. No.** How do we fix?



MSD Radix Sort (correct edition)

Key idea: Sort each subproblem separately.



Runtime of MSD

What is the Best Case of MSD sort (in terms of N, W, R)?

What is the Worst Case of MSD sort (in terms of N, W, R)?

Runtime of MSD

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)

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.

Lec36 Sorting and Data Structures Conclusion

Intuitive： Radix Sort vs. Comparison Sorting

Merge Sort Runtime

Merge Sort requires Θ(N log N) compares.

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.

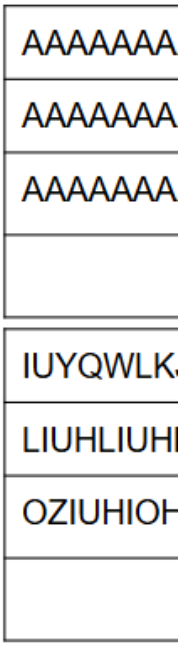
LSD vs. Merge Sort (My Answer)

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.



Just in Time Compiler

Java's Just-In-Time Compiler secretly optimizes your code when it runs.

- The code you write is not necessarily the code that executes!
- As your code runs, the "interpreter" is watching everything that happens.
  - If some segment of code is called many times, the interpreter actually studies and re-implements your code based on what it learned by watching WHILE ITS RUNNING (!!).

JIT Example

The code below creates Linked Lists, 1000 at a time.

- Repeating this 500 times yields an interesting result.

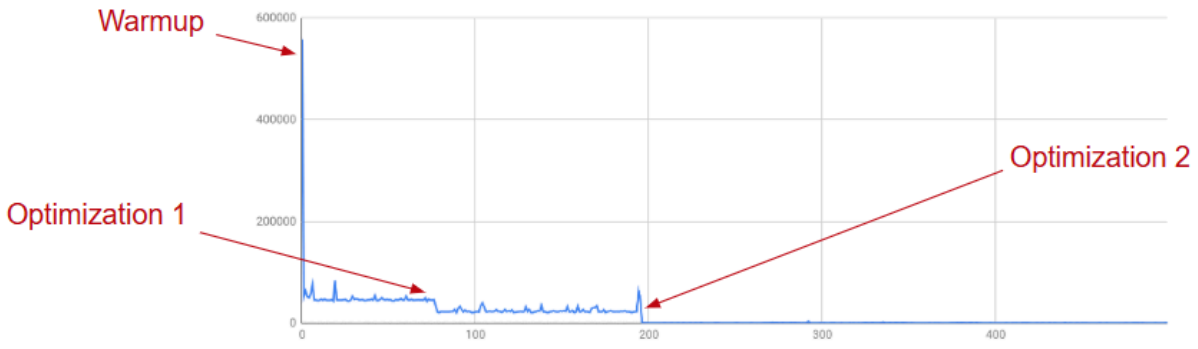
```
public class JITDemo1 {
  static final int NUM_LISTS = 1000;

  public static void main(String[] args) {
    for (int i = 0; i < 500; i += 1) {
      long startTime = System.nanoTime();
      for (int j = 0; j < NUM_LISTS; j += 1) {
        LinkedList<Integer> L = new LinkedList<>();
      }
      long endTime = System.nanoTime();
      System.out.println(i + ": " + endTime - startTime);
    }
  }
}
```

Create 1000 linked lists and print total time it takes.

JIT Example

The code below creates Linked Lists, 1000 at a time.

- Repeating this 500 times yields an interesting result.
- First optimization: Not sure what it does.
- Second optimization: Stops creating linked lists since we're not actually using them.



Bottom Line: Algorithms Can Be Hard to Compare

Comparing algorithms that have the same order of growth is challenging.

- Have to perform computational experiments.
- In modern programming environments, experiments can be tricky due to optimizations like the JIT in Java.

Note: There's always the chance that some small optimization to an algorithm can make it significantly faster.

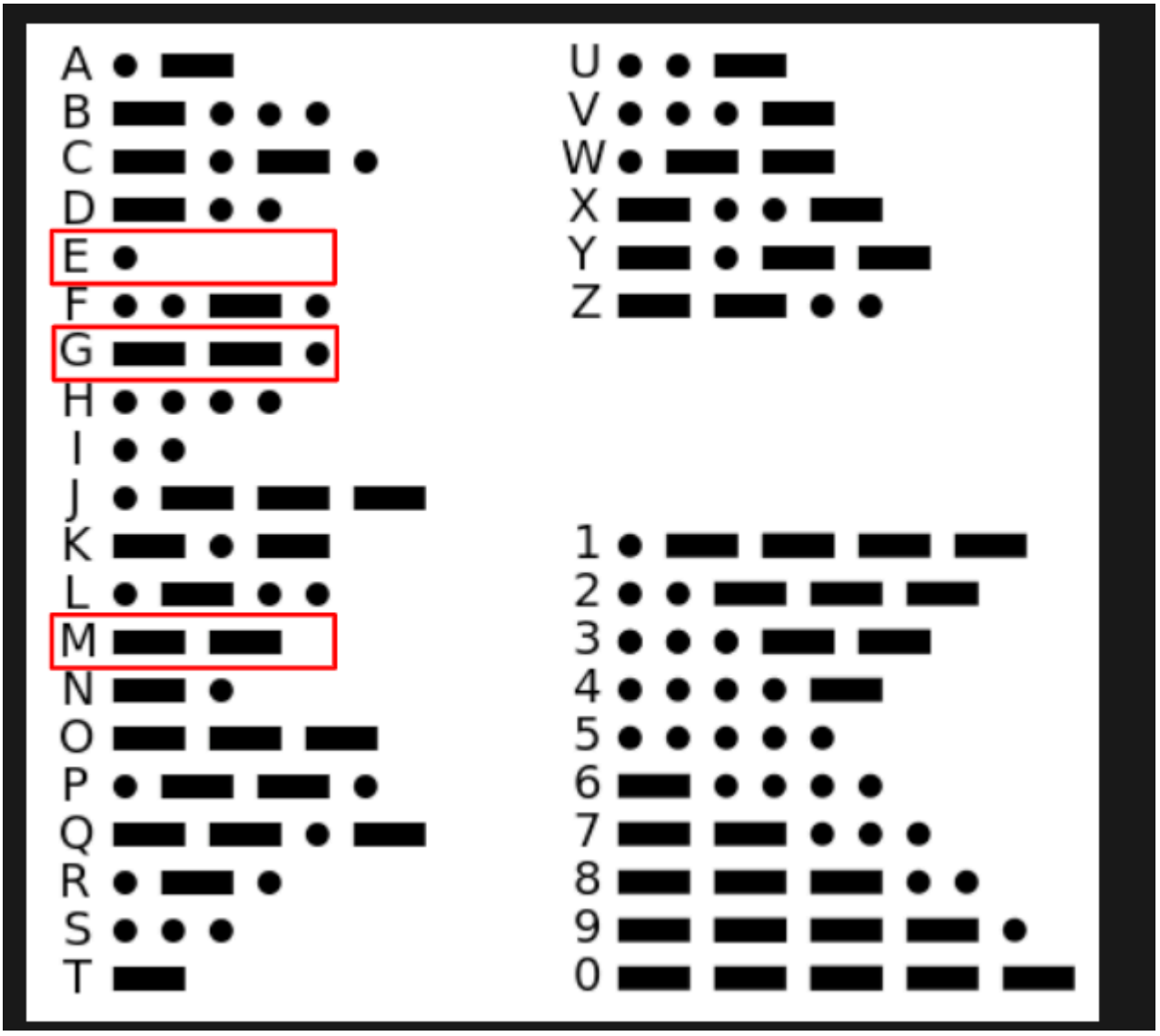
- Example: Change to Quicksort suggested by Vladimir Yaroslavskiy that we mentioned briefly in the quicksort lecture.

Lec38 Compression

Prefix-free Codes

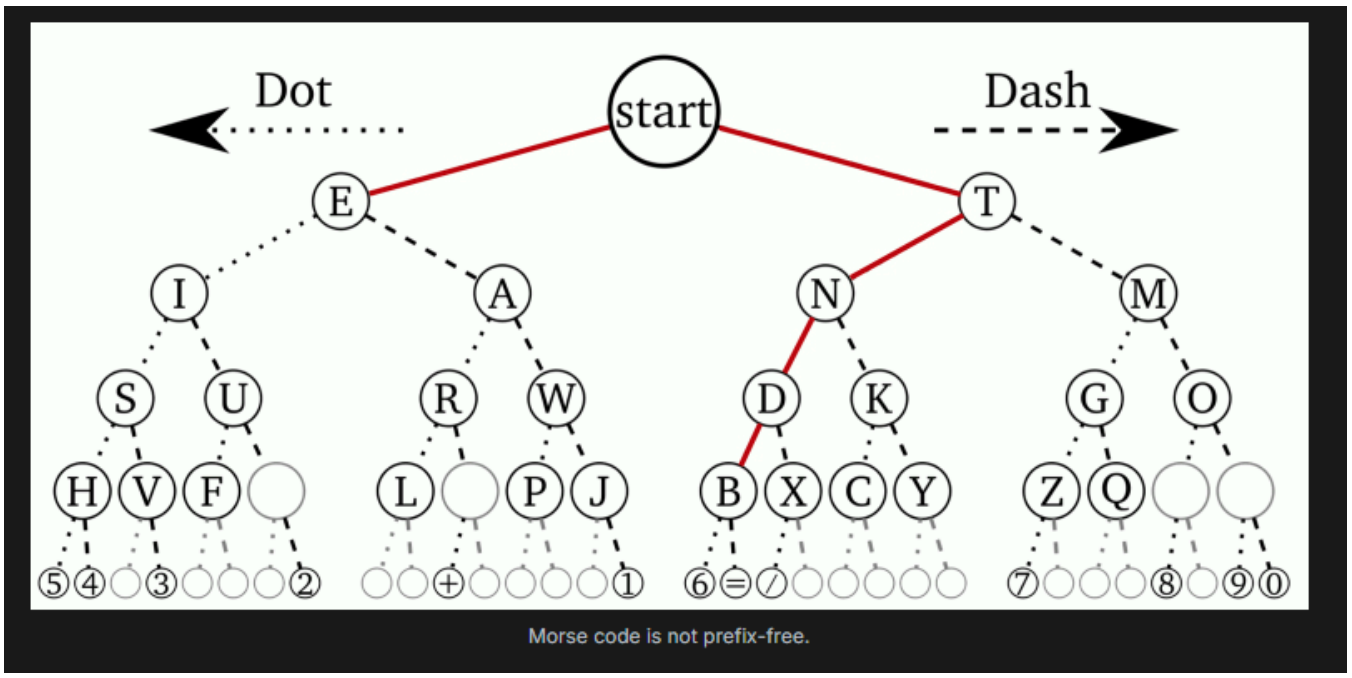
Morse Code

As an introductory example, consider the Morse code alphabet. Looking at the alphabet below, what does the sequence – – • – – • represent? It's ambiguous! The same sequence of symbols can represent either MEME, or GG, depending on what you choose – – • to represent



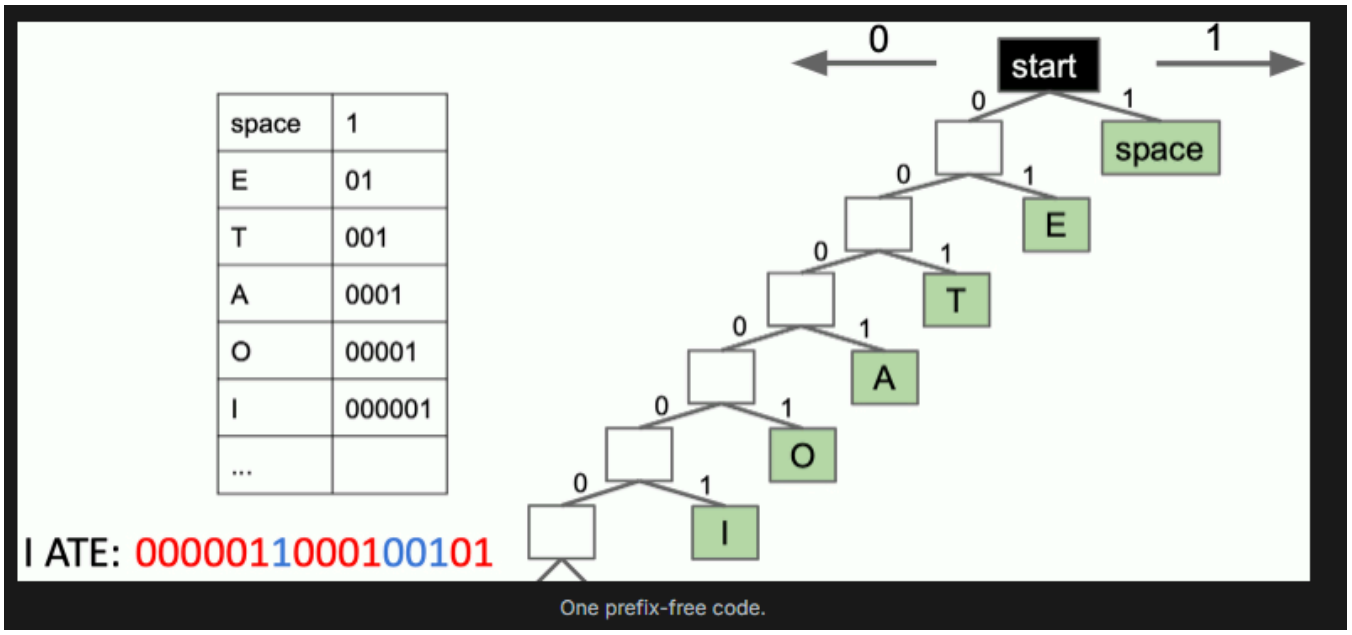
Prefix-free Codes

前缀无歧义码（prefix-free codes）：在这种编码中，没有任何一个代码词是另一个代码词的前缀。我们可以把摩斯电码表示为一棵从根到叶子的**代码词树**，每个叶子节点对应一个符号。 从这棵树中可以看到，**有些符号的编码是其他符号编码的前缀。**



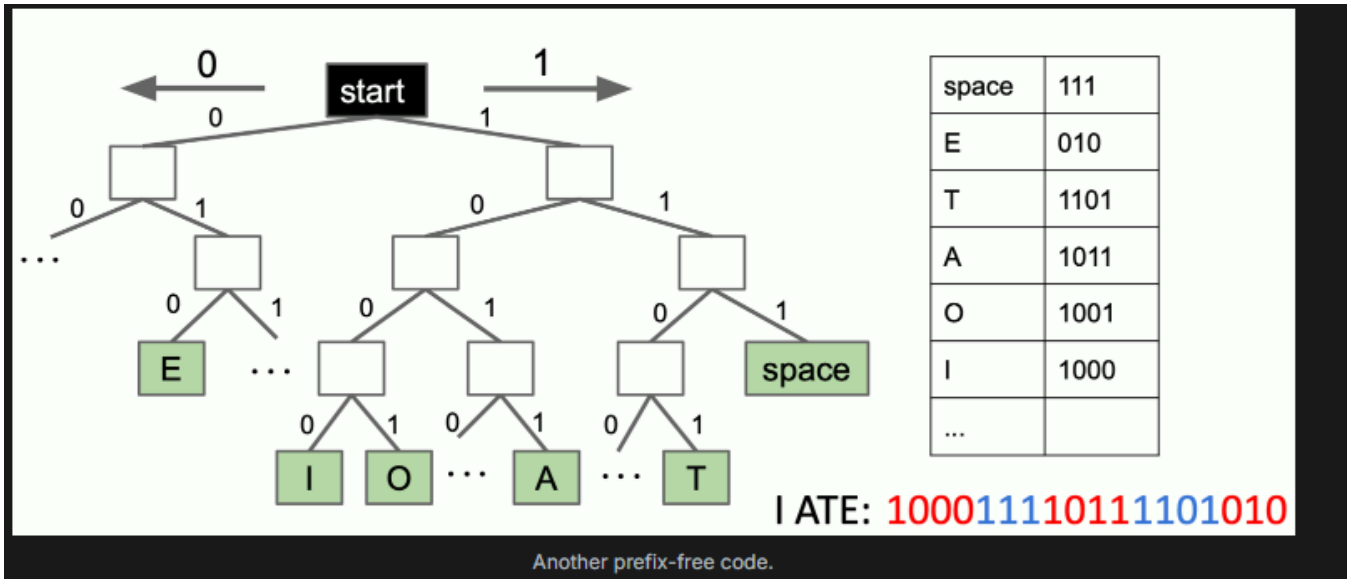
Morse code is not prefix-free.

As an example of an (arbitrary) prefix-free code, consider the following encoding:



One prefix-free code.

The following code is also prefix-free:



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

Shannon-Fano Codes

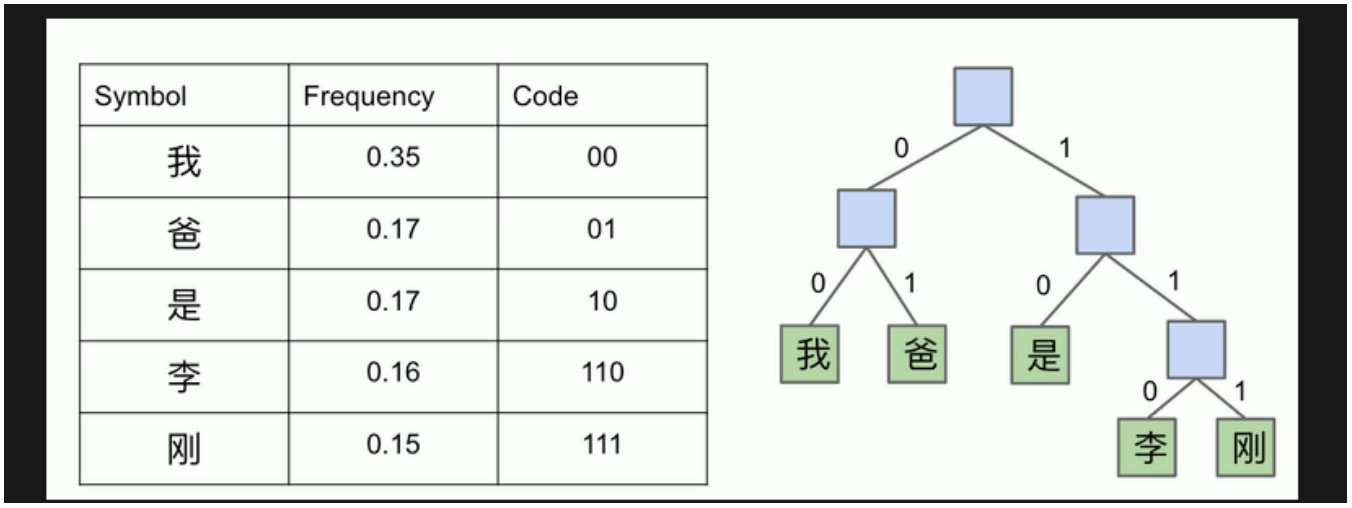
香农-范诺编码（Shannon-Fano codes）是一种**根据符号或字符及其出现概率**来构造**前缀无歧义码（prefix-free codes）*的方法。**

***其核心思想是：**让出现频率较高的字符使用更短的编码**，而*出现频率较低的字符使用更长的编码。**

算法步骤如下：

1. 统计文本中所有字符的**相对出现频率**；
2. 将字符按频率排序后，**划分为左右两部分**，使两部分的总频率尽量接近；
3. 给左半部分的每个字符编码前加上“0”，右半部分的每个字符编码前加上“1”；
4. 对左右两部分**递归重复**以上过程。

最终，你会得到一棵编码树（如下图所示）：
在这棵树中，**出现频率高的字符路径更短**，而**频率低的字符路径更长。**



Symbol	Frequency	Code
我	0.35	00
爸	0.17	01
是	0.17	10
李	0.16	110
刚	0.15	111

However, Shannon-Fano coding is NOT optimal, so it is not used very often.

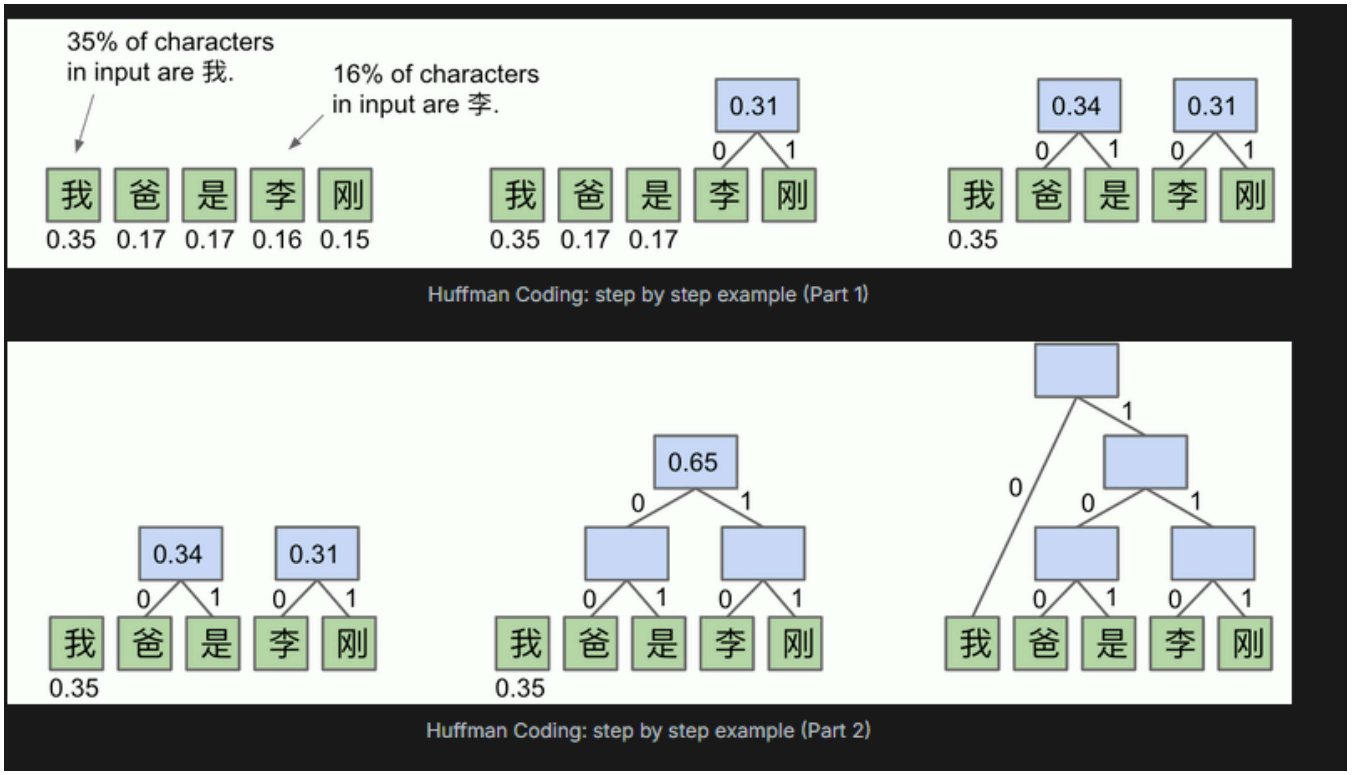
Huffman Coding Conceptuals

1.

哈夫曼编码（Huffman coding）采用了一种**自底向上（bottom-up）**的方法来构造前缀无歧义码（prefix-free codes），这与香农-范诺编码（Shannon-Fano codes）所采用的**自顶向下（top-down）**方法相对。huffman coding 是一种最优的构造prefix-free codes的方法

算法步骤如下：

1. 计算每个符号的**相对频率**；
2. 将每个符号表示为一个**节点**，其**权重（weight）**等于该符号的相对频率；
3. 从所有节点中**取出权重最小的两个节点**，将它们合并成一个新的“超级节点”（super node），其权重等于两者权重之和；
4. **重复**上述过程，直到所有节点都被合并成一棵完整的树。



2. data structures for huffman coding:

Data Structures

Let's now think about the data structures we would use for the encoding and decoding processes of the Huffman Coding process. Recall that encoding will translate symbols to code words and decoding will do the opposite. An example is as follows:

- Encoding translates `I ATE` into `0000011000100101.`
- Decoding translates `0000011000100101` into `I ATE`.

For encoding (bitstream to compressed bitstream), what data structure would we use?

There are two options!

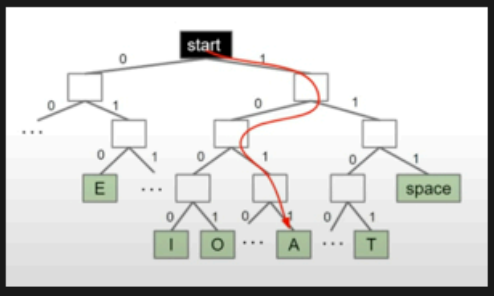
1. **HashMap/TreeMap**: create a map from character to bit sequence, calling the `get()` method to look up each character
2. **Array**: Each index of the array would represent the character, with the bit sequence in that slot of the array. Recall that each character is just an integer. For example, the letter `A` is `65`).

What's the difference? Arrays are faster than maps but might use more memory if indices are unused.

For decoding (compressed bitstream back to bitstream), what data structure would we use?

There is really only one good data structure that can help us find longest prefixes of bit streams.

Trie: Here, we can use a Binary Trie with numbers 0 and 1. When we get the bitstream, the trie allows easy lookup to the longest prefix. Below is an image that demonstrates how the trie could look.



3. 在实际中有两种方法

第一种方法：

对于每种输入类型（英文文本、中文文本、图像等），我们需要收集大量样本输入，用这些样本来构建一个**标准化的编码体系**（例如为英语、中文等分别创建标准编码）。

语料库是指**一组语言片段的集合**，用于作为该语言的代表样本。
下面是一个例子，说明我们希望使用“ENGLISH”语料库来压缩 `mobydick.txt` 文件：

```
$ java HuffmanEncodePh1 ENGLISH mobydick.txt
```

问题：
这样可能会导致**次优编码（Suboptimal encoding）**，
也就是说，我们使用的语料库与实际输入并不完全匹配。
在这个例子中，`mobydick.txt` 的字符频率分布可能与一般的英语文本不同，
因为它可能具有作者特有的语言风格或其他特点。

第二种方法:

对于每一个可能的输入文件，都为它**单独创建一套唯一的编码。**
这样，当别人收到这个压缩文件时，就可以使用我们**随文件一起发送的编码表**来进行解码。

如下例所示，我们在压缩时**没有指定语料库（corpus）**，
而是**附带了一个专门用于该文件的编码信息文件**，以帮助解码过程：

```
$ java HuffmanEncodePh2 mobydick.txt
```

问题：
这种方法需要在压缩的位流（bitstream）中**额外存储编码表（codeword table）**，
因此会占用一些额外空间。
不过，相较于使用通用语料库的方法，这种方式**在实际应用中效果更好**，
因此在现实世界中被**广泛采用。**

实际第二种方法比第一种方法用的更多

4.

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.

用huffman code 压缩得到的是不是.huf

严格来说，**哈夫曼编码本身并没有固定的文件扩展名。**

- `.huf` 是一种**常见约定**，用来表示“这是用哈夫曼编码压缩得到的文件”，但它不是标准或者必须的。
- 实际上，你压缩得到的文件只是一串**比特流**，它的扩展名可以根据你的程序或需求自由命名，比如 `.bin`、`.huf`、`.dat` 等。
- 有些实现为了方便区分，才会用 `.huf` 来提示文件内容是哈夫曼压缩的。

Compression Theory

Compression Ratios(压缩比)

数据压缩的目标是在**尽可能保留信息的前提下，减少数据序列的大小。**
例如，在英文中，字母 **e** 的出现频率比 **z** 高，因此我们希望用**更少的比特**来表示 **e**。

压缩比是衡量**压缩后数据大小与原始数据大小差异**的指标。

哈夫曼编码（Huffman Coding）*是一种压缩技术，它通过*为常见符号分配更短的比特序列来实现更高效的编码。

游程编码（Run-length encoding, RLE）*是另一种压缩方法，它会将*连续重复的字符替换为“该字符 + 出现次数”。

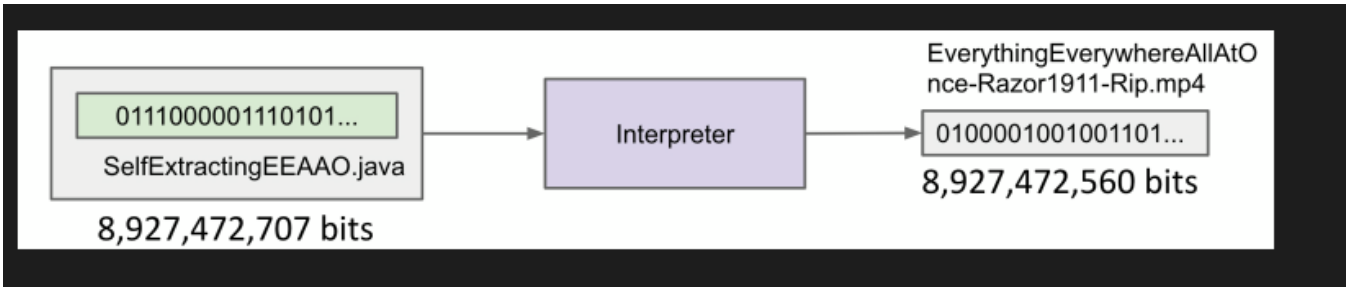
LZW 编码（Lempel–Ziv–Welch）*是一种压缩技术，它会在输入中*查找常见的重复模式，并用**较短的代码**替换这些模式。

大多数压缩技术的总体思想都是：
利用数据序列中存在的冗余性或规律性来减少数据大小。
然而，如果一个序列中**完全没有冗余或规律**，那么压缩就可能**无法实现。**

Self-Extracting Bits

Self-Extracting Bits是一种压缩技术，它将**压缩后的数据位（compressed bits）*和*解压算法（decompression algorithm）即java文件(如果是其他语言就是其他语言的文件）*以及trie对应的文件一起封装成*同一个比特序列。**

Self-Extracting Bits可以用来创建**可执行文件（executable files）**，
这些文件在任何拥有解释器（例如 **Java 解释器**）的系统上都可以**直接运行并自行解压。**



Universial Compression: An Impossible Idea

一：原因

1.

不可能设计一个算法，可以对**任意比特流压缩 50%。**

- 否则你可以反复压缩，最终只剩下 1 位，这显然是不可能的。

2.

Universal Compression: An Impossible Idea

Argument 2: There are far fewer short bitstreams than long ones. Guaranteeing compression even once by 50% is impossible. Proof:

- There are 2^{1000} 1000-bit sequences.
- There are only $1+2+4+...+2^{500} = 2^{501} - 1$ bit streams of length ≤ 500.
- In other words, you have 2^{1000} things and only $2^{501} - 1$ places to put them.
- Of our 1000-bit inputs, only roughly 1 in 2^{499} can be compressed by 50%!

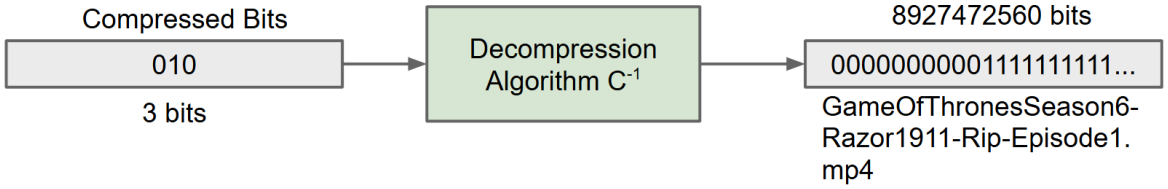
二:

A Sneaky Situation

Universal compression is impossible, but the following example implies that comparing compression algorithms could still be quite difficult.

Suppose we write a special purpose compression algorithm that simply hardcodes small bit sequences into large ones.

- Example, represent GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4 as 010.

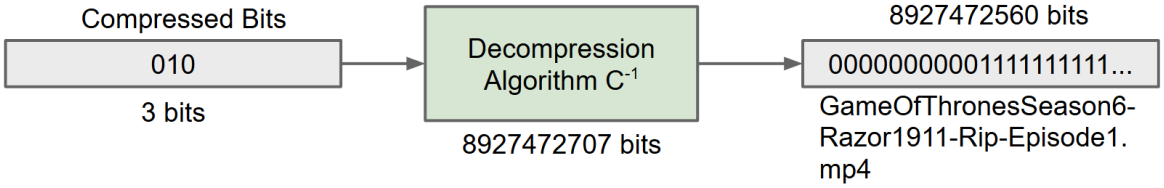


A Sneaky Situation

Suppose we write a special purpose compression algorithm that simply hardcodes small bit sequences into large ones.

- Example, represent GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4 as 010.

To avoid this sort of trickery, we should include the bits needed to encode the decompression algorithm itself.



原文意思

> Universal compression is impossible, but the following example implies that comparing compression algorithms could still be quite difficult.

意思是：**虽然通用压缩是不可能的，但比较不同压缩算法的好坏仍然可能很困难。**

> Suppose we write a special purpose compression algorithm that simply hardcodes small bit sequences into large ones.
> Example, represent GameOfThronesSeason6-Razor1911-Rip-Episode1.mp4 as 010.

这里举了一个"作弊式"的例子：

- 假设你写了一个**专门针对某个文件的压缩算法**，
- 它直接把这个**非常大的文件**硬编码成一个**非常短的比特序列**，比如 010，
- 对于这个特定文件来说，压缩率看起来极好，但这是**作弊式压缩**，只对这个特定文件有效。

> To avoid this sort of trickery, we should include the bits needed to encode the decompression algorithm itself.

这句话的意思是：

- 为了避免这种"只针对某个特定文件作弊"的情况，
- 在评估压缩效果时，我们应该把**用于实现解压算法的比特数**也算进去，
- 也就是说，不仅要看压缩后的比特流多小，还要加上解压程序本身占用的空间。

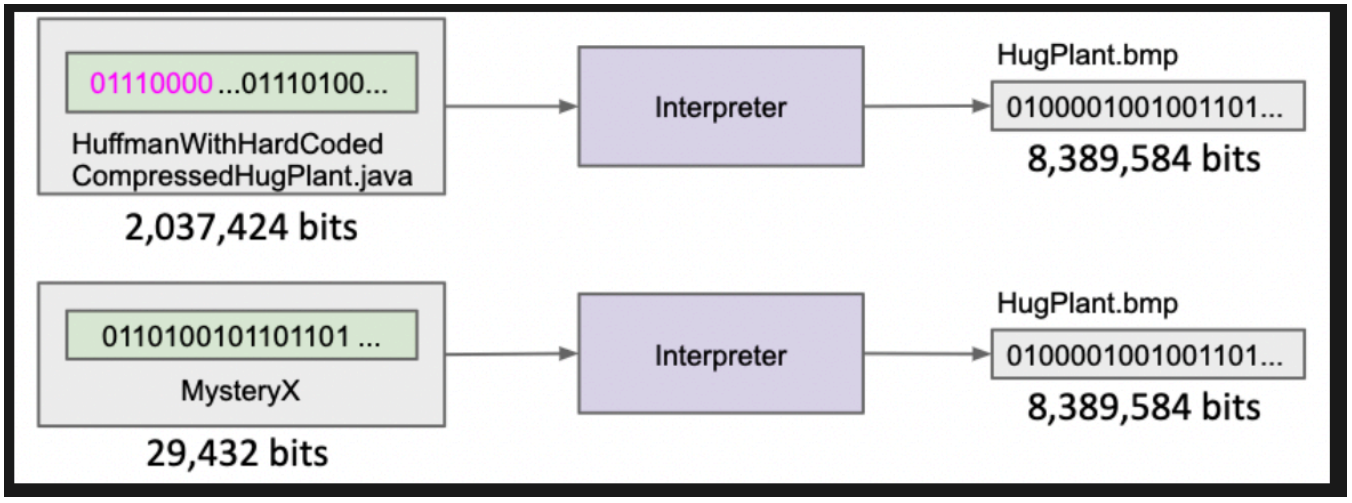
Lec39 Compression, Complexity, and P=NP?

Models of Compression

1.

Recall the `HugPlant` example from the previous chapter. Using Huffman encoding, we can achieve a compression ratio of 25%.

However, using another algorithm we'll call `MysteryX` for now, we can compress `HugPlant.bmp` down to 29,432 bits! This achieves a 0.35% compression ratio.



What is `MysteryX`? Well, it's simply the Java code `HugPlant.java` written to generate the `.bmp` file! Going back to the model of self-extracting bits, we see the power of code and interpreters in compression. This leads us to two interesting questions:

- **comprehensible compression:** given a target bitstream `B`, can we create an algorithm that outputs useful, readable Java code?
- **optimal compression**: given a target bitstream `B`, can we find the *shortest* possible program that outputs this bitstream?