CSCI 136:
Data Structures
and
Advanced Programming

Lecture 33

Graphs, part 4

Instructor: Dan Barowy

Williams

Announcements

Last date for resubmissions: May 19

Can't accept after: *grades due* May 19!

Includes labs 5-9, *all quizzes*.

Lab 7 back today (already graded)

Hashtable activity solution post after class

Outline

Double hashing formula

Shortest paths

Teaching evaluations

Life skill #12:
physical health = mental health

Life skill #13:
be the hero in your own education

How/where are hash codes used?

Hash function

$$h(k) = h_1(k) \ mod \ |T|$$

where k is the key, and $|T|$ is the size of the array T.

$h(k)$ relies on $|T|$. In what class should $h(k)$ be defined?

We typically put $h(k)$ **in the hash table implementation**.

Note that $h_1(k)$ can be defined independently of T.

$h_1(k)$ = `key.hashCode()`

Double hashing

$$h(i, k) = (h_1(k) + i \cdot h_2(k)) \ mod \ |T|$$

where k is the key, i is the ith collision, and $|T|$ is the size of the array T.

Again, $h(i, k)$ should appear in the hash table implementation.

$h_1(k)$ = `key.hashCode()`

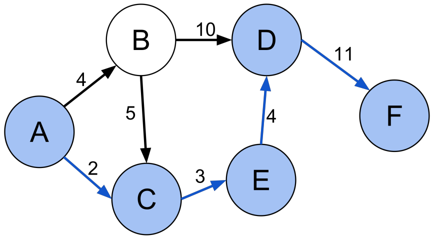
$h_2(k)$ is a second hash function.

$h_2(k)$ = `toSHA1(keyToBytes(key))`

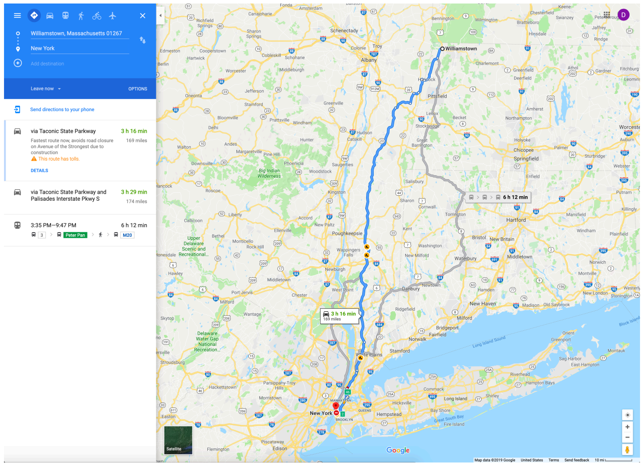
Graphs: shortest paths

Shortest path problem

The **shortest path problem** is the problem of finding a **path between two vertices** in a graph such that **the sum** of the weights of its constituent edges **is minimized**.



Applications



Williamstown, Massachusetts 01267

New York

Add destination

Leave now

OPTIONS

Send directions to your phone

via Taconic State Parkway 3 h 16 min

Fastest route now, avoids road closure on Avenue of the Strongest due to construction 169 miles

⚠ This route has tolls.

DETAILS

via Taconic State Parkway and Palisades Interstate Pkwy S 3 h 29 min

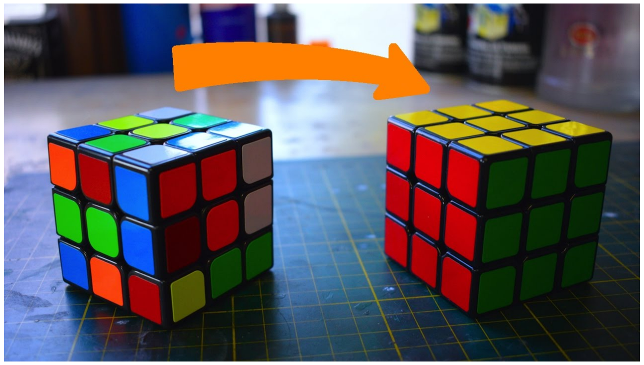
174 miles

3:35 PM—9:47 PM 6 h 12 min

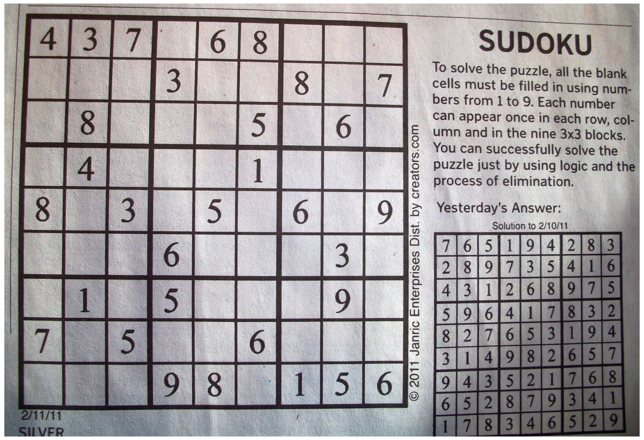
3 > Peter Pan > > M20

Applications

Applications



Applications



4	3	7		6	8			
			3			8		7
	8				5		6	
	4				1			
8		3		5		6		9
			6				3	
	1		5				9	
7		5			6			
			9	8		1	5	6

2/11/11
SILVER

© 2011 Janric Enterprises Dist. by creators.com

SUDOKU

To solve the puzzle, all the blank cells must be filled in using numbers from 1 to 9. Each number can appear once in each row, column and in the nine 3x3 blocks. You can successfully solve the puzzle just by using logic and the process of elimination.

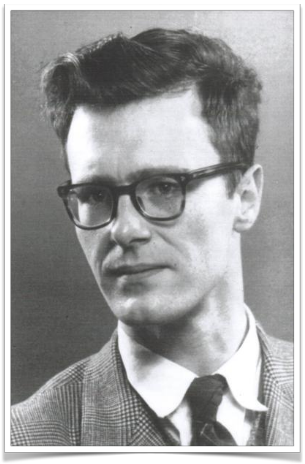
Yesterday's Answer:

Solution to 2/10/11

7	6	5	1	9	4	2	8	3
2	8	9	7	3	5	4	1	6
4	3	1	2	6	8	9	7	5
5	9	6	4	1	7	8	3	2
8	2	7	6	5	3	1	9	4
3	1	4	9	8	2	6	5	7
9	4	3	5	2	1	7	6	8
6	5	2	8	7	9	3	4	1
1	7	8	3	4	6	5	2	9

Applications

Dijkstra's algorithm



- Invented by Edsgar Dijkstra in 1959.
- The original version used a min-priority queue.
- Designed using pencil and paper; algorithm was intended to demonstrate to non-technical people how computers could be useful.

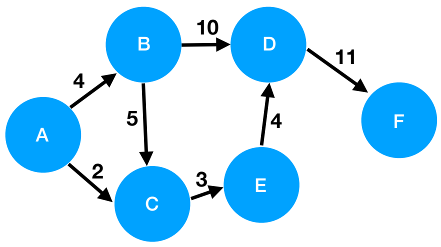
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	∞
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef



Q

{A, B, C, D, E, F}

Looking for path from A to F.

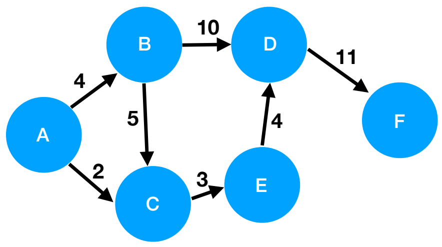
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef



Q

{A, B, C, D, E, F}

Looking for path from A to F.

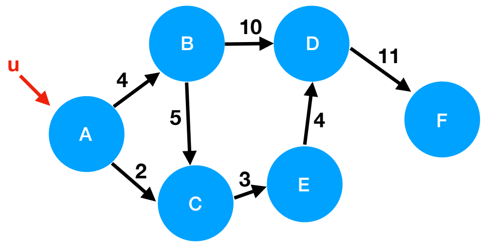
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	∞
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	undef
C	undef
D	undef
E	undef
F	undef
G	undef



Q

{B, C, D, E, F}

Looking for path from A to F.

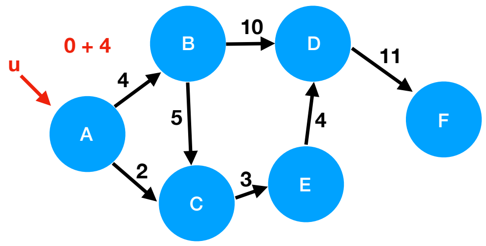
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	∞
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	undef
D	undef
E	undef
F	undef
G	undef



Q

{B, C, D, E, F}

Looking for path from A to F.

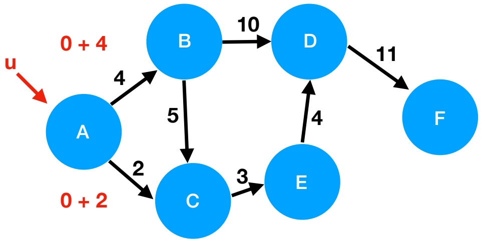
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef



Q

{B, C, D, E, F}

Looking for path from A to F.

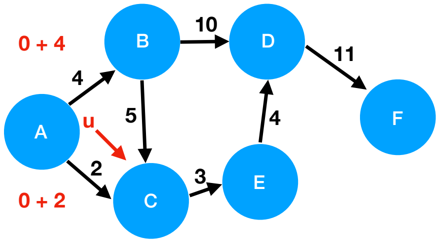
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	∞
E	∞
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	undef
F	undef
G	undef



Q

{B, D, E, F}

Looking for path from A to F.

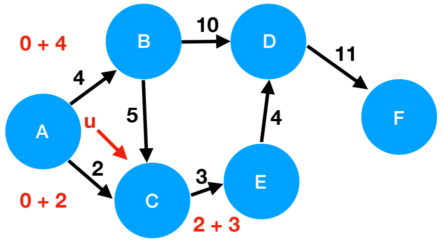
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef



Q

{B, D, E, F}

Looking for path from A to F.

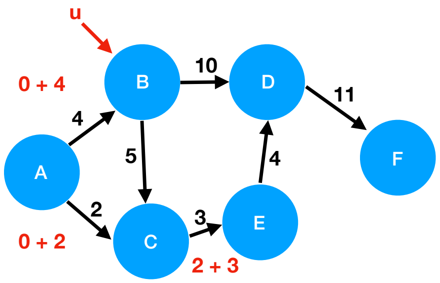
```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

A	0
B	4
C	2
D	∞
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	undef
E	C
F	undef
G	undef



Q

{D, E, F}

Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

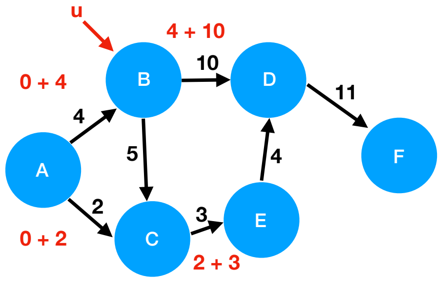
A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

Q

{D, E, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

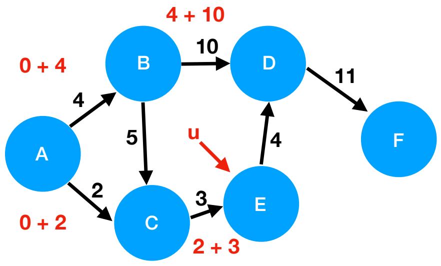
A	0
B	4
C	2
D	14
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	B
E	C
F	undef
G	undef

Q

{D, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

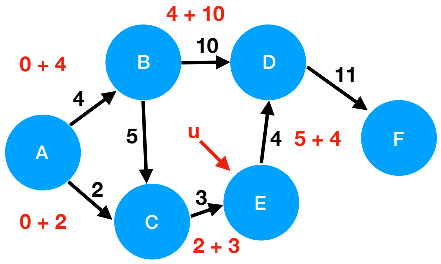
A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

Q

{D, F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

dist

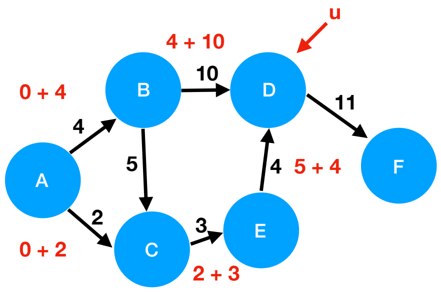
A	0
B	4
C	2
D	9
E	5
F	∞
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	undef
G	undef

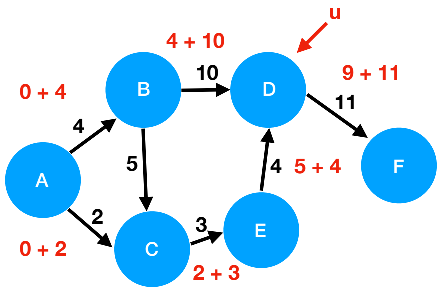
Q

{F}



Looking for path from A to F.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



Looking for path from A to F.

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

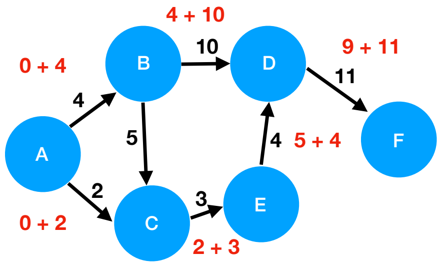
prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{F}

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



Looking for path from A to F.

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

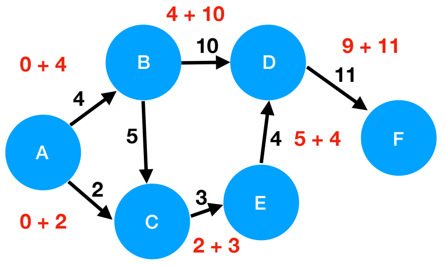
A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{}

Done!

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```



Read backward from F and reverse.

dist

A	0
B	4
C	2
D	9
E	5
F	20
G	∞

prev

A	undef
B	A
C	A
D	E
E	C
F	D
G	undef

Q

{}

Done!

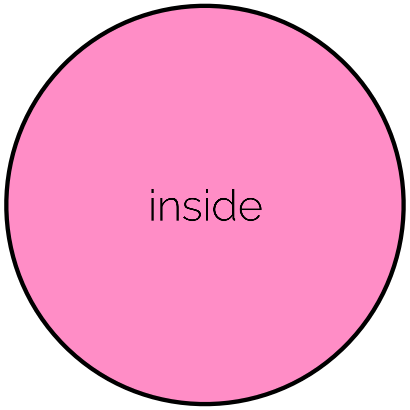
You learned a lot this semester!

(great job!)

Java

Program design

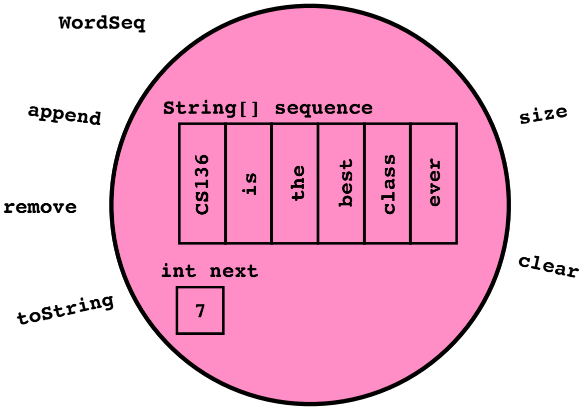
Abstraction



inside

outside

Composition



WordSeq

append

remove

toString

String[] sequence

CS136 | is | the | best | class | ever

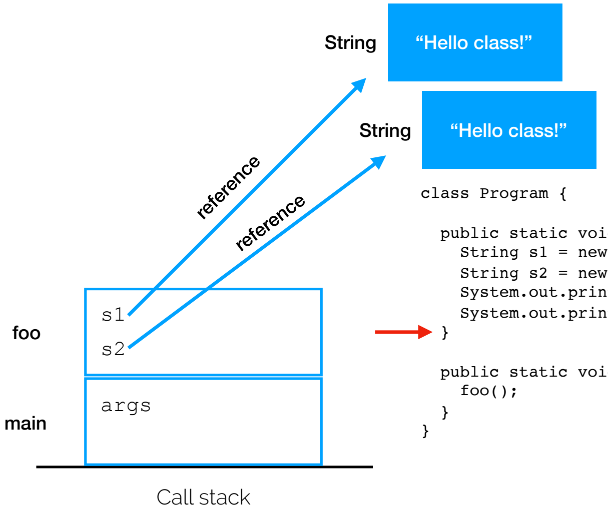
int next

7

size

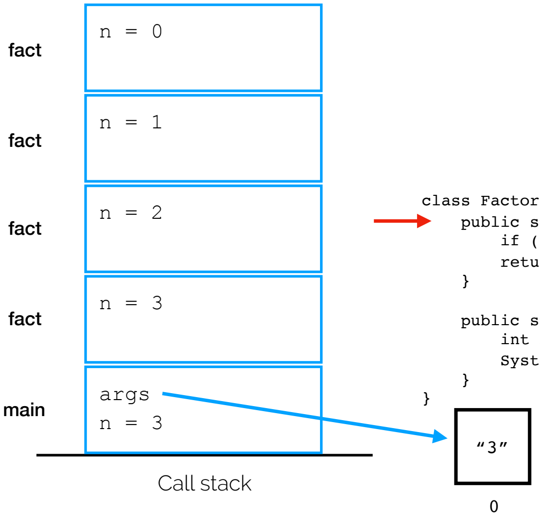
clear

Abstract machine



```
class Program {

  public static void foo() {
    String s1 = new String(“Hello class!”);
    String s2 = new String(“Hello class!”);
    System.out.println(s1 == s2);
    System.out.println(s1.equals(s2));
  }

  public static void main(String[] args) {
    foo();
  }
}
```

Recursion



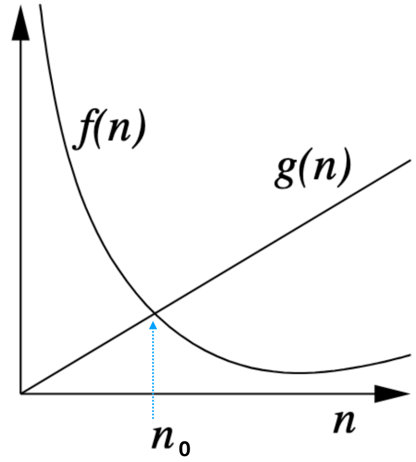
```
class Factorial {
    public static int fact(int n) {
        if (n == 0) { return 1; }
        return n * fact(n – 1);
    }

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        System.out.println(fact(n));
    }
}
```

Formal methods

Induction

Program performance

Big-O analysis



Algorithm design

of copies for doubling expansion:

	1	+	2	+	4	+	…	+ (n/2)
`add()`	up to 2nd elem.		up to 4th elem.		up to 8th elem.			up to nth elem.

Neat theorem: $1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$

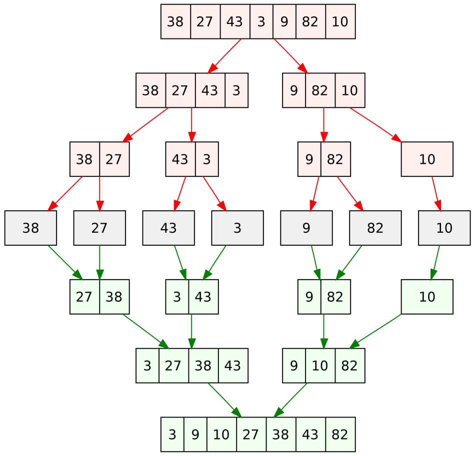
Suppose $n = 2^k$.

Then $1 + \dots + n/2 = 1 + \dots + 2^k/2$

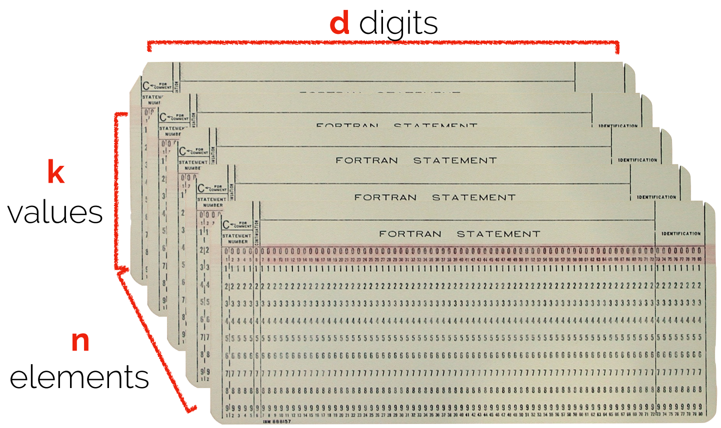
$= 1 + \dots + 2^{k-1} = 2^k - 1 = n - 1$

Doubling expansion costs ≈ **O(n)**

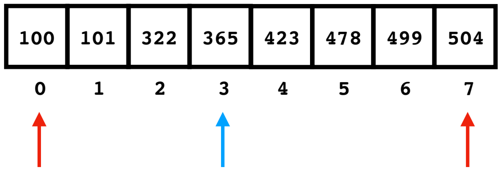
Sorting algorithms



Exotic sorting algorithms



Search algorithms

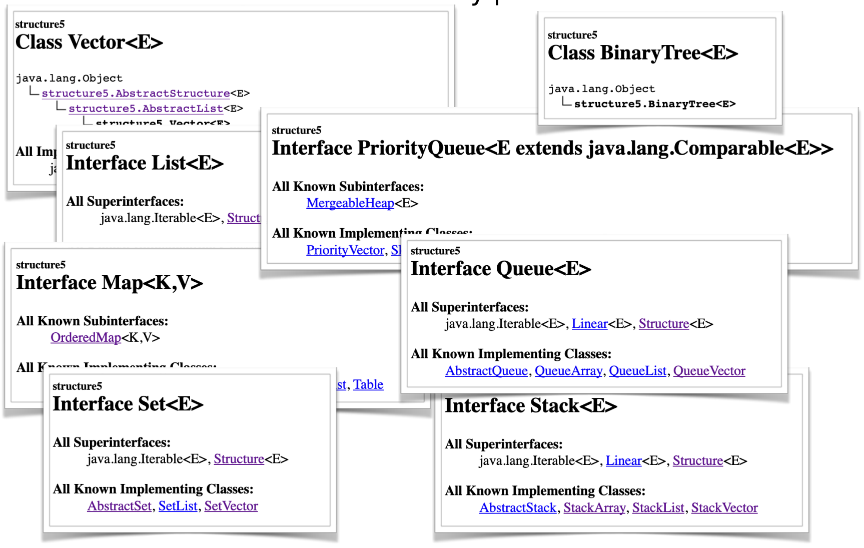


100	101	322	365	423	478	499	504
0	1	2	3	4	5	6	7

322 = 365? **no**

322 < 365? **yes**

Abstract data types (ADTs)



structure5
Class Vector<E>

```
java.lang.Object
  └structure5.AbstractStructure<E>
      └structure5.AbstractList<E>
          └structure5.Vector<E>
```

All Imp
ja

structure5
Class BinaryTree<E>

```
java.lang.Object
  └structure5.BinaryTree<E>
```

structure5
Interface List<E>

All Superinterfaces:
java.lang.Iterable<E>, Struct

structure5
Interface PriorityQueue<E extends java.lang.Comparable<E>>

All Known Subinterfaces:
MergeableHeap<E>

All Known Implementing Classes:
PriorityVector, Sl

structure5
Interface Map<K,V>

All Known Subinterfaces:
OrderedMap<K,V>

All Known Implementing Classes:
st, Table

structure5
Interface Queue<E>

All Superinterfaces:
java.lang.Iterable<E>, Linear<E>, Structure<E>

All Known Implementing Classes:
AbstractQueue, QueueArray, QueueList, QueueVector

structure5
Interface Set<E>

All Superinterfaces:
java.lang.Iterable<E>, Structure<E>

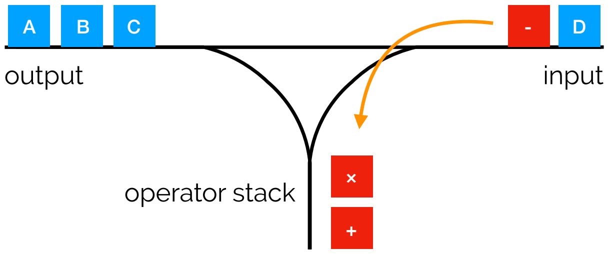
All Known Implementing Classes:
AbstractSet, SetList, SetVector

Interface Stack<E>

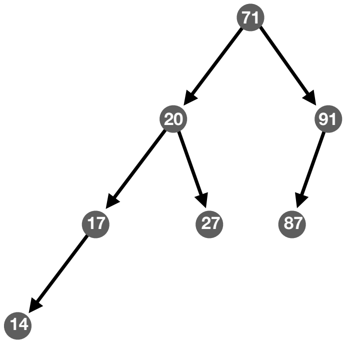
All Superinterfaces:
java.lang.Iterable<E>, Linear<E>, Structure<E>

All Known Implementing Classes:
AbstractStack, StackArray, StackList, StackVector

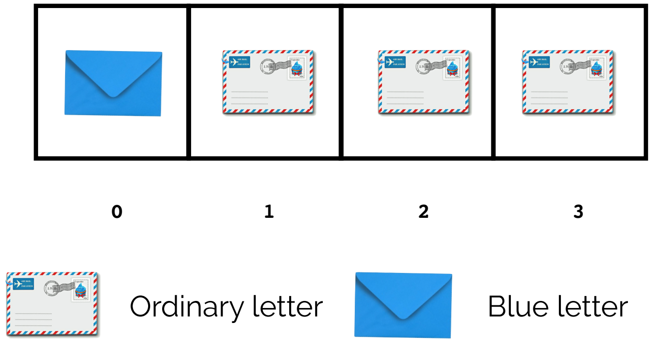
Useful applications of ADTs



Ordering structures



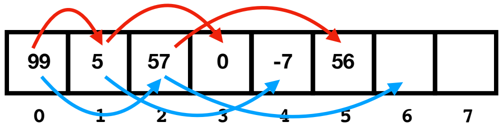
Partially-ordering structures



Number representations

0	1	1	1	1	0	0	1
7	6	5	4	3	2	1	0

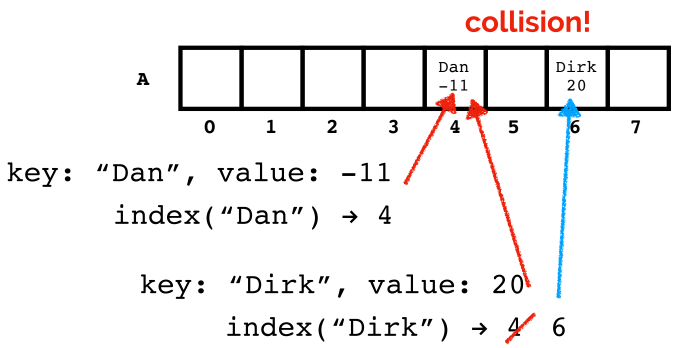
Efficient encoding of structures



left child — right child

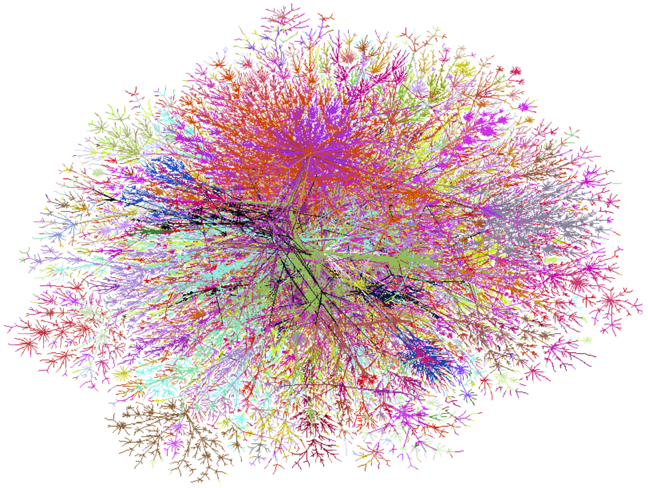
High-performance structures

collision!

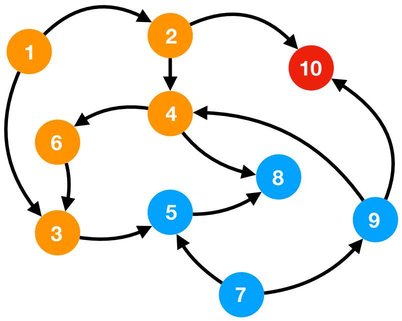


```
key: “Dan”, value: -11
     index(“Dan”) → 4

     key: “Dirk”, value: 20
          index(“Dirk”) → ~~4~~ 6
```

Very general structures: graphs



Graph algorithms



Recap & Next Class

Today we learned:

Double hashing formula

Shortest paths

Next class:

Final exam review

Evaluation Forms

(all of these are anonymous)

We care a lot about what you say in these forms. Please take your time and write thoughtful responses.

I changed a number of parts of this course this semester. Your feedback is very valuable to me, as it will help me decide whether these changes were good.

Purpose of Blue Sheets

> Student comments on the blue sheets [...] are solely for your benefit. They are not made available to department or program chairs, the Dean of the Faculty, or the CAP for evaluation purposes.
>
> —Office of the Provost, Williams College

Purpose of SCS Forms

> “[T]he SCS provides instructors with feedback regarding their courses and teaching. The faculty legislation governing the SCS provides that SCS results are made available to the appropriate department chair, the Dean of the Faculty, and at appropriate times, to members of the Committee on Appointments and Promotions (CAP). The results are considered in matters of faculty reappointment, tenure, and promotion.”
>
> —Office of the Provost, Williams College

Blue sheet prompts:

* What course topic did you enjoy the most?

* What course topic did you least enjoy? Do you think that it was valuable to learn anyway?

* Are there other aspects of the course that you liked or disliked? (E.g., *office hours*, *TAs*, *assignments*, *course structure*, *meeting times*, etc.) Feel free to suggest alternatives.

* Did you look forward to coming to class?