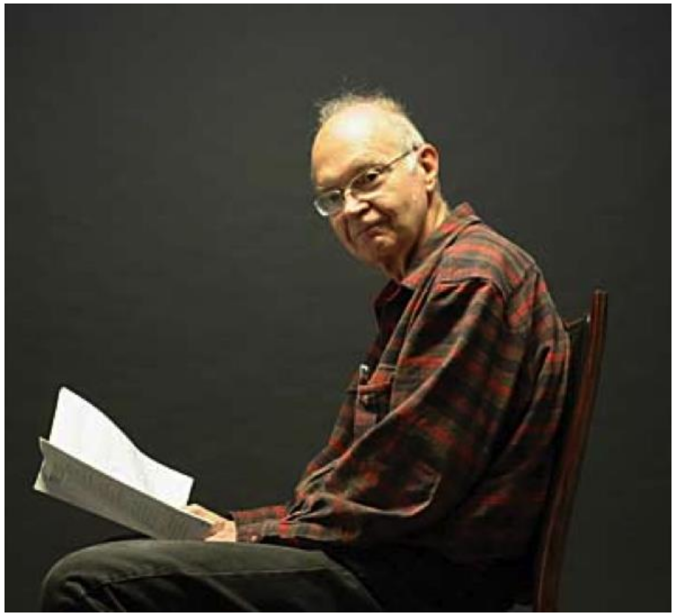
Data Structures and Algorithms

Tony Mullins

Some scientists are like explorers who go out and plant the flag in new territory; others irrigate and fertilize the land and give it laws and structure. (Knuth)



Introduction

- Data Structures + Algorithms = Programs

(Wirth)

- If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?

(Perlis)

Programs

- Supply services to client users

- Formal contract between client and supplier

- User
  - a person,
  - another program,
  - another object

- Programs must be:
  - Correct
  - Robust
  - Deliver results in a timely fashion

Critical Role of Computing Systems

- Hospitals
- Aviation
- Transport Services
- Weather Reporting
- Nuclear Reactors

- Critical Systems must meet deadlines
- Must provide results in optimum time-frames

- How to define shortest time frame possible?
- How to measure and compare performance of programs?

- Algorithmic analysis
  - Used to measure performance
  - Compare different solutions to same problem
- Data Structures

Analysis

- Use benchmarking to compare different programs that solve the same problem
- Develop a mathematical model that can be used to classify programs

Compute the sum of the first N natural numbers

Two separate functions that solve the problem are given.

```
static long sumN(long n){
    long s = n * (n + 1)/2;
    return s;
}

static long sumN1(long n){
    long s = 0;
    for(int j=0; j < n; j++) s = s + (j + 1);
    return s;
}
```

Which solution is the best?

data structure =

- a particular way of storing and organizing data so that it can be used efficiently
- A structure that enforces an ordering between the elements in the collection
- A way to organize data so that it can be managed efficiently
- A structure that optimizes the cost of insertion and retrieval

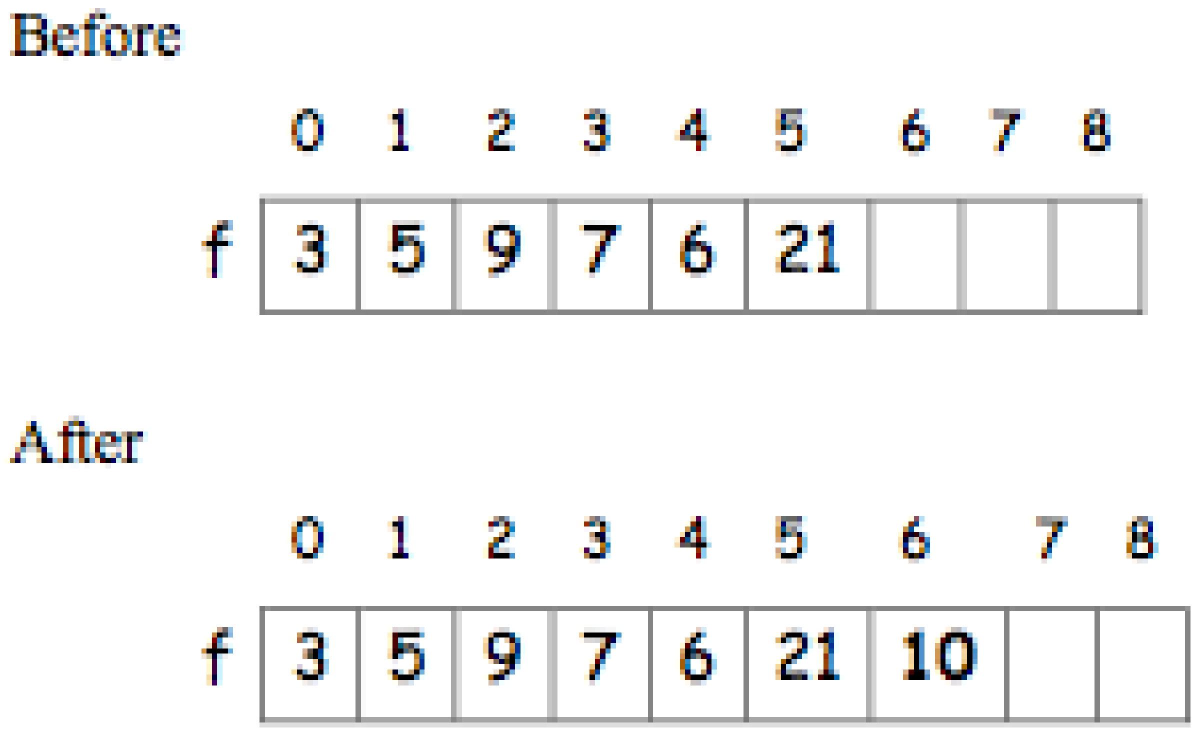
- Simple data structures
  - A class that describes a real world entity
    - Book
    - Car
    - Person
- Complex
  - A collection of things organized in some order

Cost of insertion v Cost of retrieval

- Choice of data structure depends greatly on the data set to be modeled and the performance requirements of the application

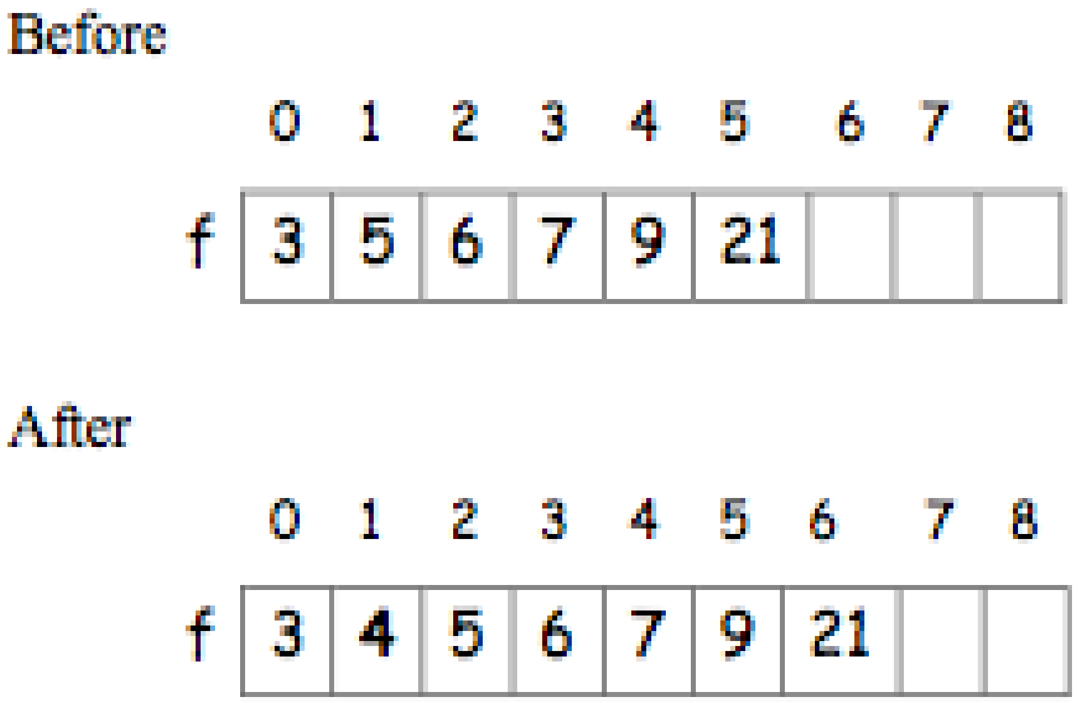
- Manage a collection of integers so as the optimize both cost of insertion and retrieval

No ordering: append 10



- Insert
  - simple assignment
  - fixed cost
- Retrieval
  - Linear search
  - Cost proportional to size of data set

Ordered Sequence: insert 4



Before

	0	1	2	3	4	5	6	7	8
f	3	5	6	7	9	21			

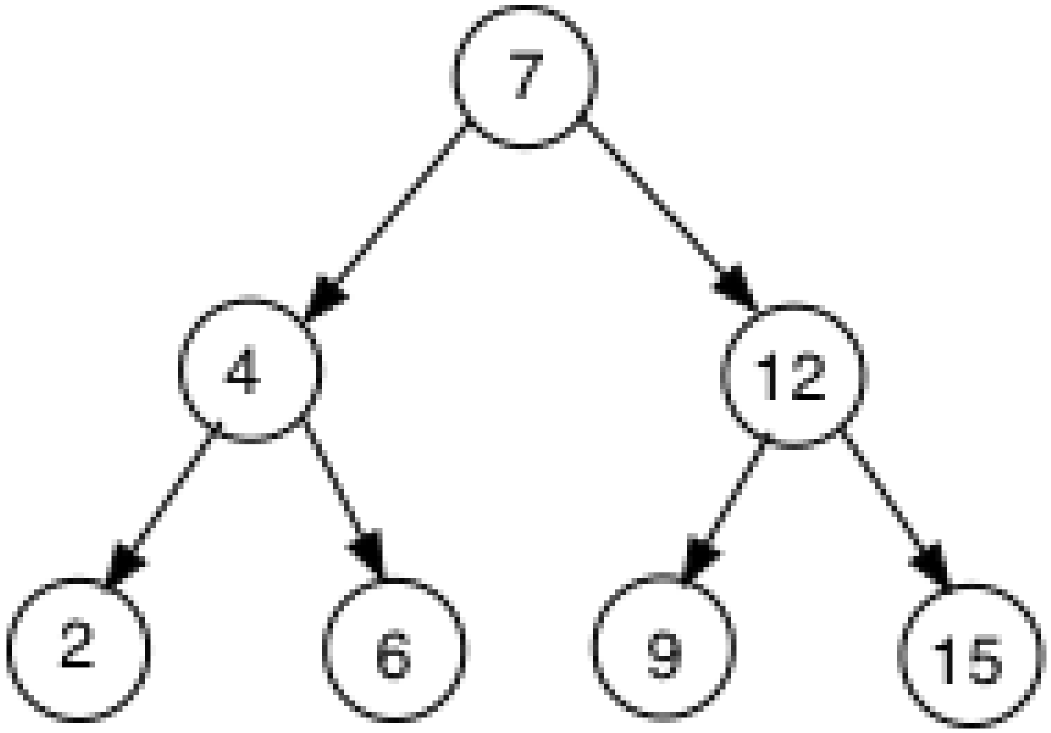
After

	0	1	2	3	4	5	6	7	8
f	3	4	5	6	7	9	21		

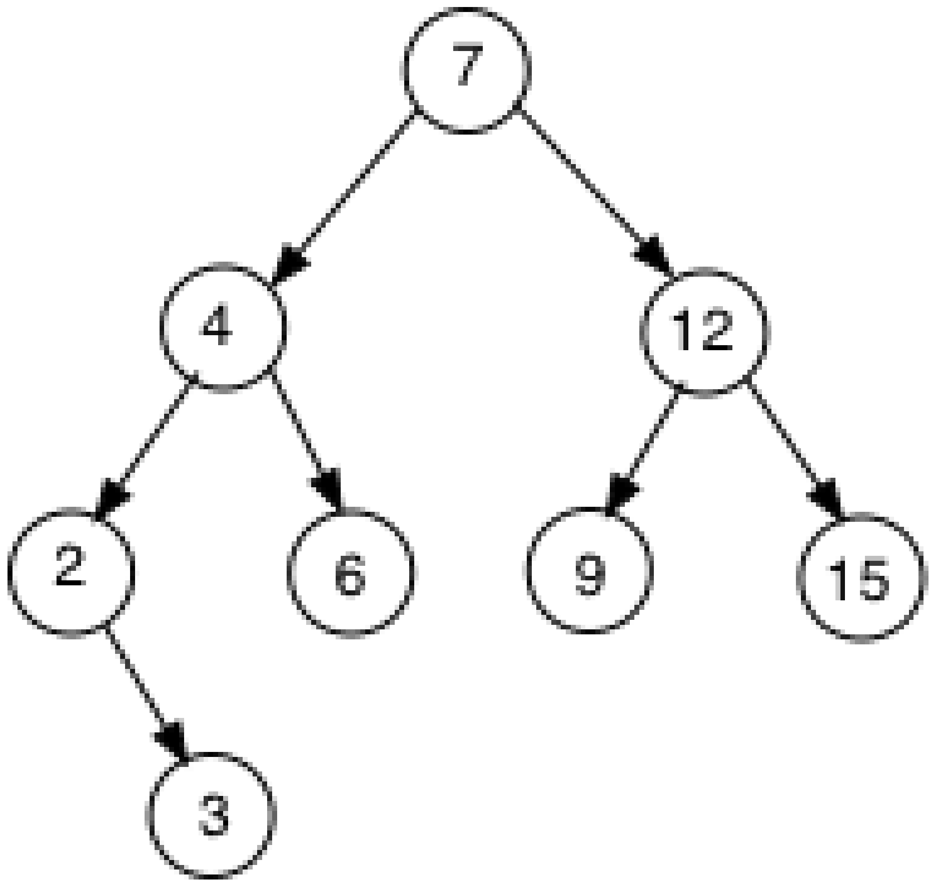
- Insertion
  - Shift elements one position right until correct place for 4 is found
  - Cost proportional to size of sorted sequence

- Retrieval
  - Binary search
  - Cost is $\log_2 N$

Binary Search Tree



Insert 3



Hoare .. made clear that decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined.

(Algorithms + Data Structures = Programs, Wirth)

What Data Structures have you worked with?

- Linear Data Structures
  - String, Array
- Object-oriented Data Structures
  - Set
  - ArrayList
  - Map

Object-oriented Data Structures

- Encapsulation
  - Hides implementation
- Interfaces
  - Provides methods that allow user interact with data
- Genericity
  - Allows type parameterization

Set

- HashSet
  - Unordered set of elements that enforces rule that duplicates not allowed

- TreeSet
  - Ordered set based on the ordering defined by the given type and enforces rule that duplicate values not allowed

Set interface

- Common interface for both types defined
  - add
  - remove
  - contains
  - addAll(union)
  - retainAll(intersection)
  - etc

Generic Type

- Data structure designed so that data type of elements may be defined by the user
- All elements in a set must be the same type or in same family based on inheritance
- Only required to write the class once because type is variable

Set

```
HashSet<Integer> set = new HashSet<Integer>();
 for(int j = 0; j < 10; j++){
        int x = (int)(Math.random()*10);
        set.add(new Integer(x));
}
System.out.println();
System.out.println(set.toString());
```

```
[2, 4, 9, 6, 3, 7, 0]
```

Set

```
TreeSet<Integer> t1 = new TreeSet<Integer>();
while(t1.size() < 10){
        int x = (int)(Math.random()*20);
        t1.add(new Integer(x));
}
System.out.println();
System.out.println(t1);
```

```
[0, 2, 4, 5, 6, 7, 8, 9, 15, 19]
```

Observation

- Collection classes only work if certain methods are implemented by classes contained by them
  - Methods are: equals, hashCode, toString and compareTo
- Do not enforce encapsulation
  - Has consequences for clients
  - May not modify certain attributes in contained objects

- Should write classes so that attributes that define equality and, hence, hashCode are immutable
- Better still: write immutable classes

Writing your own Data Structures

- A primary focus of this course is to learn to write our own data structures.
- Will not use Collection container classes from Java API to do this.
- Do want to deepen our understanding of container classes
- Takes us behind the scenes to look at the foundations
- Study classical data structures
  - Dynamic arrays
  - Stacks and Queues
  - Trees

Basic Structures

- Classes that encapsulate simple data entities
  - Book, person, car, etc
- Arrays or sequences
- Pointers or reference variables that can be used to build dynamic data structures

- Use object-oriented style of programming
- Define named classes that implement a given data structure
- Provide an interface that enforces the underlying semantics of the data model

Example of an IntStack

- Stack manages a collection of data items such that new items are always added to the head of the stack

- Items are always removed from the head position

- Implements last-in-first-out behaviour

```
class IntStack{
    private Integer stack[];
    private int size;
    IntStack(int n){
        stack = new Integer[n];
        size = 0;
     }
```

```
boolean push(Integer x){
    if(size < stack.length){
        stack[size] = x;
        size++;
        return true;
    }
    return false;
}
```

```
boolean pop(){
        if(size > 0){
           size--;
          return true;
        }
        return false;
}
Integer top(){
      if(size > 0) return stack[size-1];
      return null;
}
```

```
  boolean empty(){return size == 0;}
  boolean full(){return size == stack.length;}
}
```

Content

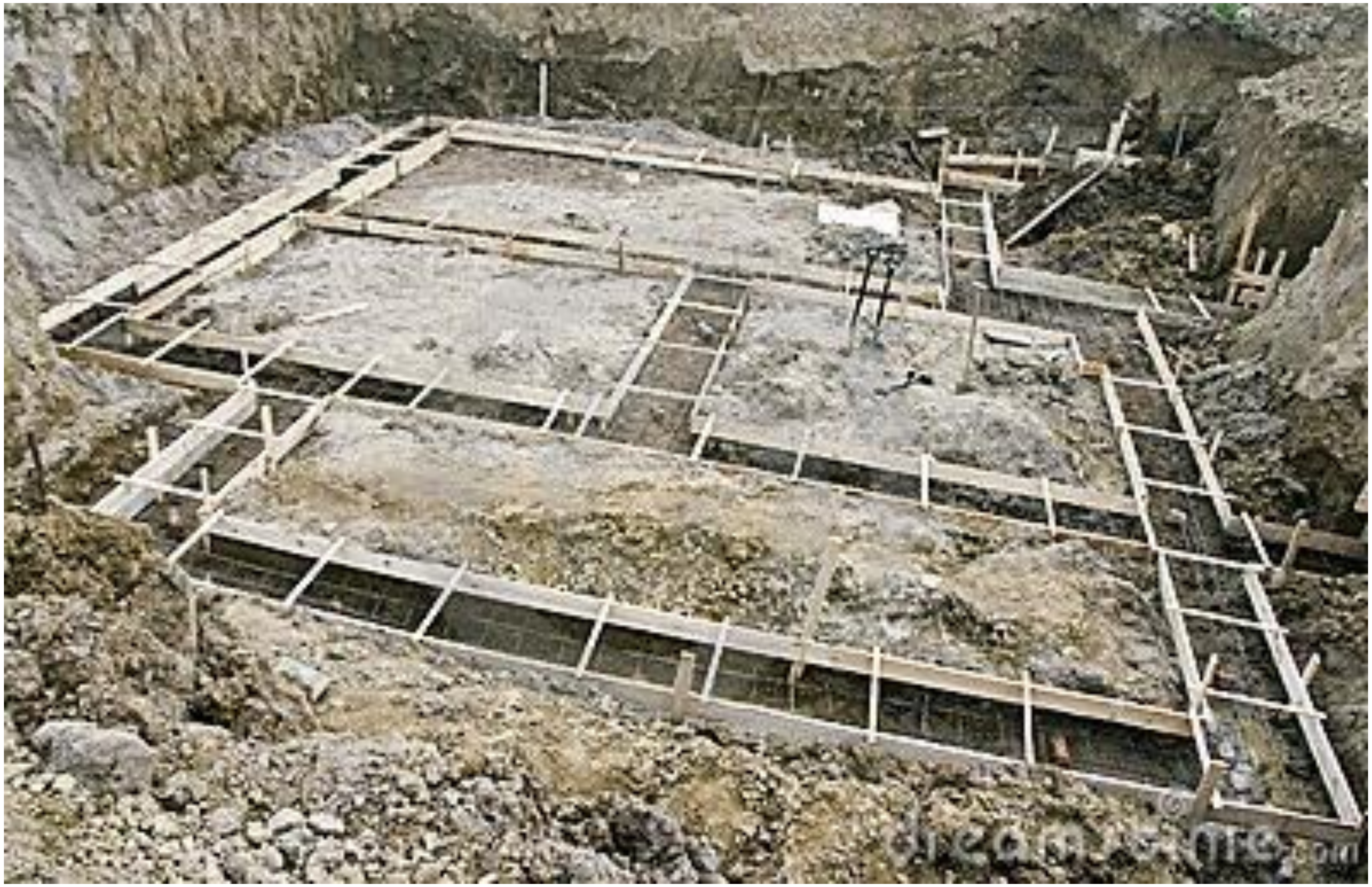
- Recursion
- Analysis of Algorithms
- Fast Sorting
- Dynamic Data Structures
- Hashing
- Binary Search Trees
- AVL Trees
- Maps
- Graphs and Graph Theory
- External Data Structures

Assessment

- Weekly worksheets and Labs
  - 60%
  - Must complete 7 of these during the course
  - Final mark based on best 7
- Written Examination
  - 40%

- Must get minimum of 35% in both components and average of 40% between both to pass

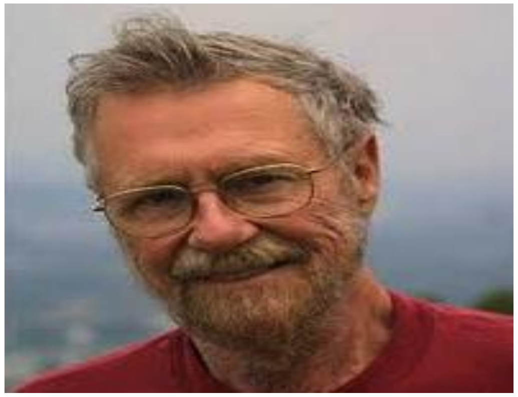
What is this picture?

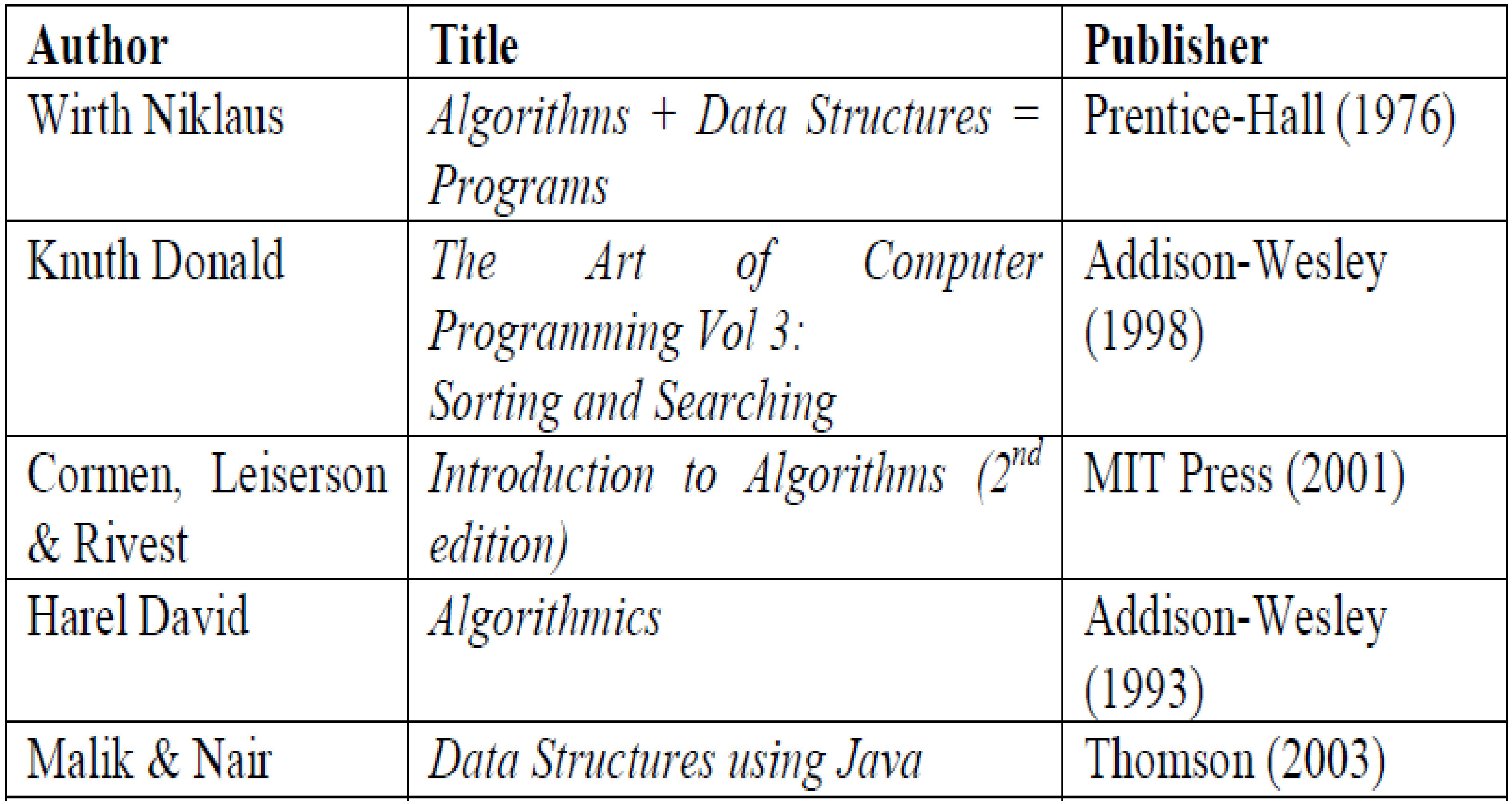


History of Data Structures

- FORTRAN Language designed by Backus(1924 - 2007) in 1954
  - First programming language
- Stack introduced in 1955
- Algol Language(1958-60)
  - Dijkstra proposes stack be used to implement recursive procedures
- Quicksort developed by Hoare in 1960
- Knuth begins Art of Computer Programming in 1963
  - Introduces complexity analysis of algorithms

Knuth(1938-), Dijkstra(1930-2002), Hoare(1934-), Wirth(1934-)





Author	**Title**	**Publisher**
Wirth Niklaus	*Algorithms + Data Structures = Programs*	Prentice-Hall (1976)
Knuth Donald	*The Art of Computer Programming Vol 3: Sorting and Searching*	Addison-Wesley (1998)
Cormen, Leiserson & Rivest	*Introduction to Algorithms (2nd edition)*	MIT Press (2001)
Harel David	*Algorithmics*	Addison-Wesley (1993)
Malik & Nair	*Data Structures using Java*	Thomson (2003)

Wilson Raymond	*An Introduction to Dynamic data Structures*	McGraw-Hill (1988)
Stubbs & Webre	*Data Structures with Abstract Data Types & Pascal*	Brooks Cole (1985)
Collins William	*Data Structures An Object-Oriented Approach*	Addison-Wesley (1992)
Tremblay & Cheston	*Data Structures and Software Development*	Prentice-Hall (2001)
Goodrich, M.T. & Tamassia, R	*Data Structures and Algorithms in Java*	Wiley, 2005

Recursion

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise." The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

Stephen Hawking, A Brief History of Time(1988)

What is recursion?

- Put simply a definition is said to be recursive if it is defined in terms of itself.

- In Mathematics recursion is defined by two properties:
  - a base case
  - a set of rules that reduce the chain of invocations to the base case.

$$n! = \begin{cases} 1, & if\ \ n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

n! = n*(n-1)!

5! = 5*(5-1)! = 5*4!

= 5*4*(4-1)! = 5*4*3!

= 5*4*3*(3-1)! = 5*4*3*2!

= 5*4*3*2*(2-1)! = 5*4*3*2*1!

= 5*4*3*2*1*(1-1)! = 5*4*3*2*1*0!

= 5*4*3*2*1*1

= 120

- Why do we need the base case?
- Force recursion to terminate

n! = n*(n-1)!

4! = 4*3! = 4*3*2!=4*3*2*1!

= 4*3*2*1*0!

= 4*3*2*1*0*-1!...

Calculate 2^n

$2^0 = 1$

$2^n = 2*2^{(n-1)}$

$f(n) = 1$, if $n = 0$,

$2*f(n-1)$, if $n > 0$

$f(3) = 2*f(2) = 2*2*f(1)$

$= 2*2*2*f(0)$

$= 2*2*2*1 = 8$

Peano axioms.

- 1 is a Natural number
- if n is a Natural number , then $n + 1$ is a Natural number

The set of natural numbers is the smallest set satisfying the previous two properties.

- Use the definition to show 3 is a Natural number.

$1 \in N$ (R1)

$\Rightarrow 1+1 \in N$ (R2)

$= 2 \in N$ (addition)

$\Rightarrow 2+1 \in N$ (R2)

$= 3 \in N$ (addition)

How to write an infinite sequence of a’s with a finite piece of text:

$$\begin{aligned} S &= aS \\ &= aaS = aaaS = aaaaS = \ldots \end{aligned}$$

Backus-Naur Form

- System that provides a formal notation to specify the syntax of a formal language such as a programming language.
- The production rules defined in this notation can be used to determine the syntactic correctness of a block of code or a whole program.

- Notation consists of two types of terms called terminal and non-terminal.
- Rules are written in an equational style where non-terminals appear on the left hand side and terminals or non terminals on the right hand side.
- Every non-terminal symbol must appear on the left hand side of one syntactic equation.

Examples

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Use of recursive definition

```
<unSignedInt> ::= <digit> |
                  <unSignedInt> <Digit>
```

Show 256 valid unsigned integer

<unSignedInt> => <unSignedInt> <digit>

=> <unSignedInt><digit> 6

=> <digit>56

=> 256

Signed Integer with no leading zeros

```
<integer> ::= <zero> | <signedInt> |
                         <unsignedInt>

<signedInt> ::= <sign><unSignedInt>

<unsignedInt> ::= <pDigit> | <pDigit> <allDigits>

<alldigits> ::= <digit> | <digit> <allDigits>

<pDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit> ::= <zero> | <pDigit>

<sign> ::= + | -
```

Language with no loops!

Repetition has to be coded with recursive calls on a given function. Each invocation of the function must make progress towards a terminating condition.

$$n! = \begin{cases} 1, & if\ \ n = 0 \\ n * (n-1)!, & \text{if } n > 0 \end{cases}$$

Factorial n

```
static int fac(int n){
    if(n == 0)
        return 1;
    else
        return(n*fac(n-1));
}
```

```
fac(3) = return(3*fac(2))
       = return(3*[return(2*fac(1))])
       = return(3*[return(2*[return(1*fac(0))]])
       = return(3*[return(2*[return(1*1)]])
       = return(3*[return(2*1])
       = return(3*2)
       = 6
```

Calculate 2^n

F(n) = 1, if n = 0,

2*f(n-1), if n > 0

F(3) = 2*f(2) = 2*2*f(1)

= 2*2*2*f(0)

= 2*2*2*1 = 8

```
static int f(int n){
    if(n == 0) return 1;
    else return 2*f(n-1);
}

static int f1(int n){
    int k = 1;
    for(int j = 0; j < n; j++) k = 2*k;
    return k;
}
```

Read list with sentinel

```
static void readList(Scanner in){
    int x = in.nextInt();
    if(x != sentinel){
        // process x
        readList(in);
    }
}
```

Read n items

```
static void readList(int n, Scanner in){
    if(n != 0) {
        int x = in.nextInt();
        // process x
        readList(n-1, in);
    }
}
```

Reverse list

```
static void reverseList(Scanner in){
    int x = in.nextInt();
    if(x != sentinel){
        reverseList(in);
        System.out.printf(“%3d”, x);
    }
}
```

Using run-time stack

Suppose input list: 2,4,5,6,7,-1

The variable x is local to the function and is allocated memory on the run-time stack. Each time reverseList is invoked a new x is given space on the stack. When -1 read the recursion terminates and each of the recursive calls can complete.

This gives output: 7 6 5 4 2

Fibonacci numbers

$$fib(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n \geq 2 \end{cases}$$

```
static int fib(int n){
    if(n == 0)
        return 1;
    else if(n == 1)
        return 1;
    else
        return(fib(n-1) + fib(n-2));
}
```

Compute first n terms of sequence

```
System.out.print("Sequence: ");
for(int j = 0; j < n; j = j + 1){
      t = fib(j);
     System.out.printf(“%4d”, t);
}
```

For n = 10 the output will be:

```
Sequence:    1   1   2   3   5   8  13  21  34  55
```

- Given solution is correct
- But not efficient because each term is generated independently of all other terms

- From the definition and from observation we see that new terms can be generated from existing terms by adding the two previous terms together

- base cases are: fib(0) = 1, fib(1) = 1
- fib(2) = fib(0) + fib(1)

- In general, given two existing Fibonacci terms a, b the next term is a + b.

```
static void fibSequence(int a, int b, int i, int n){
    if(i == 0) System.out.printf("%4d",a);
    else if(i == 1) System.out.printf("%4d",b);
    else{
        System.out.printf("%4d",a+b);
        int temp = a; a = b; b = temp +  b;
    }
    if(i+1 < n) fibSequence(a,b,i+1,n);
}
```

Listing files in a directory

- Write a function that lists the names of all files in a folder including its subfolders.

- Use File class in Java package java.io

Method	**Semantics**
File(String pathname)	Constructor that takes a pathname as argument.
boolean isDirectory()	Returns true if the object is a directory; false otherwise.
String getName()	Return the name of the file or directory.
File[] listFiles()	Returns an array of pathnames denoting the files in the current directory

```
static void listFiles(File dir){
    File files[] = dir.listFiles();
    if(files != null){
        for(File f : files)
            if(f.isDirectory()) //recursive call here
                listFiles(f);
            else //print name of file
                System.out.println(f.getName());
    }
}
```

Recursion over Sequences

Given an int array of N values write recursive functions to:

- compute the sum of the values in the array;
- linearly search the array for a given value x.

```
static int sum(int f[],int n){
    if(n == f.length) return 0;
    else return(f[n]+sum(f,n+1));
}
```

```
int f[] ={2,4,7}
sum(f,0) = return(2 + sum(f,1))
                   =   return(2 + return(4 + sum(f,2)))
                   = return(2 + return(4 +
                                   return(7 + sum(f,3))))
                        = return(2 + return(4
                                      + return(7 + 0)))
                        = return(2 + return(4 + 7))
                        = return(2 + 11)
                        = 13
```

```
static boolean search(int f[],int n,int x){
    if(n == f.length) return false;
    else{
        if(f[n] == x)
            return true;
        else
            return search(f,n+1,x);
    }
}
```

```
public class ArrayTest {
    public static void main(String args[]){
        int k[] = {2,5,6,7,12,34,6,7,8,9,20,21,2,2};
        int t = sum(k,0);
        System.out.println("Sum = "+t);
        System.out.println(search(k,0,12));
        System.out.println(search(k,0,22));
    }
}
```

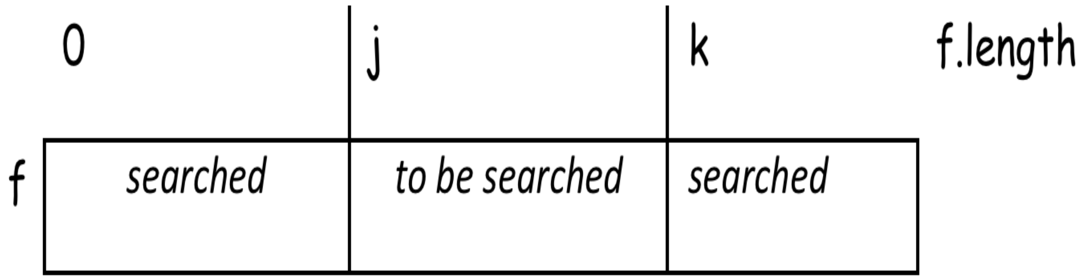
Strategy

Iterative Solution

- Given a sorted array of integer values and some integer x, write a code fragment to determine if x is an element of the array.

strategy is to repeatedly start with the middle element and continuously halve the size of the search space until it reaches a segment of size 1

(Bottenbruch, 1962)


0
j
k
f.length
f
searched
to be searched
searched

```
static boolean binSearch(int f[],int x){
    int j = 0;
    int k = f.length;
    while(j + 1 != k){
        int i = (j + k)/2;
        if( x >= f[i])
            j = i;
        else
            k = i;
    }
    return(f[j]==x);
}
```

Recursive Binary Search

```
static boolean binarySearch(int f[], int lb,
                                    int ub, int x){
    if(lb+1 == ub) return (f[lb] == x);
    else{
        int mid = (lb+ub)/2;
        if(x >= f[mid])
            return binarySearch(f,mid,ub,x);
        else
            return binarySearch(f,lb,mid,x);
    }
}
```

Tail Recursion

```
fac(n : Int) : Int = if(n == 0) 1 else n * fac(n-1)

fac(3) = 3*fac(2)
       = 3*2*fac(1)
       = 3*2*1*fac(0)
       = 3*2*1*1
       = 6
```

Needs stack to hold values until termination reached. For large N can get stack overflow.

Tail Recursion

A *tail recursive* function is a special case of recursion in which the last instruction executed in the method is the recursive call.

This type of function uses what is termed an *accumulator* parameter. The idea is to allow the function to accumulate the result as it recursively invokes itself. In this way, the terminating state *knows* the result and can yield or return it.

Tail Recursion

```
static int fac(int n) {
    if(n == 0) return 1;
    else
        return getFac(1,1,n);
}
static int getFac(int a, int n0, int n){
    if(n0 == n) return a;
    else  return getFac(a*(n0+1),n0+1,n);
}
```

Sum elements in array

```
static int sum(int dt[]){
      if(dt.length == 0) return 0;
      else
         return getSum(0,dt,0);
}
static int getSum(int j, int dt[], int sum){
 if(j == dt.length) return sum;
 else
      return getSum(j+1,dt,sum+dt[j]);
}
```

Create class with static methods

```
class TRF{
    private TRF(){}
    static int fac(int n) {
     if(n == 0) return 1;
     else
       return getFac(1,1,n);
    }
    private static int getFac(int a, int n0, int n){
     if(n0 == n) return a;
     else  return getFac(a*(n0+1),n0+1,n);
    }
     ...
}
```

```
public static void main(String[] args) {
    System.out.println(TRF.fac(3));
    int data[] = {1,2,3,4,5,6,7,8,9};
     System.out.println(TRF.sum(data));
   }
```