Introduction to Data Structure

Bruce Nan

Outline

- Origin & Develop
- What does data structure study?
- Some interesting examples

The Father

- **Donald Ervin Knuth** (born January 10, 1938) is a *computer scientist* and Professor Emeritus of the Art of Computer Programming at Stanford University.
- As author of the seminal multi-volume work *The Art of Computer Programming*, Knuth has been called the "father" of the *analysis of algorithms and data structure*, contributing to the development of, and systematizing formal mathematical techniques for, the rigorous analysis of the computational complexity of algorithms.
- He has received various other awards including the Turing Award, the National Medal of Science, the John von Neumann Medal, and the Kyoto Prize.

Origin & Development

- What is the essential of programming?

  Data Representation: how to store data in computer

  Data Processing: how to process data and solve problem

  Data Structure + Algorithm = Programming

What data structure study?

- How to write program solve problem?

  Practical Problem → Extract variable and relationship → Build problem model → Write program → Solution

- Problem: numerical and non-numerical
  - Numerical problem: formula
  - Non-numerical problem: data structure

- Example 1

  Student manage problem – Table Structure

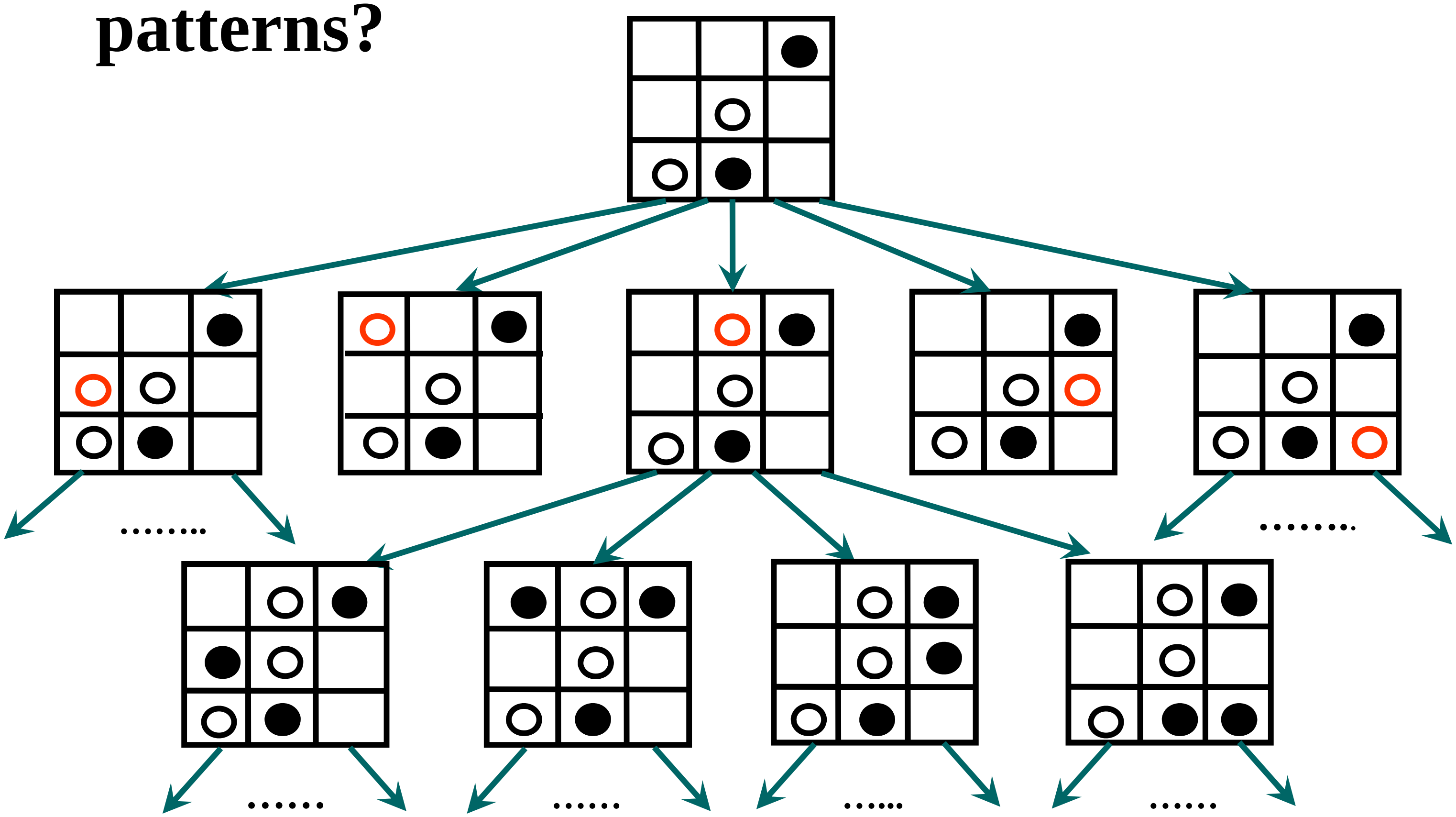
What's other functions? What's the relationship between data?

Number	Name	Gender	Birth
0001	Mike	male	2000/09/02
0002	Monica	female	2001/07/23
0003	Eric	male	2000/12/04

- Example 2

  Chess play problem – Tree Structure

How to realize the play? And how to store all patterns?

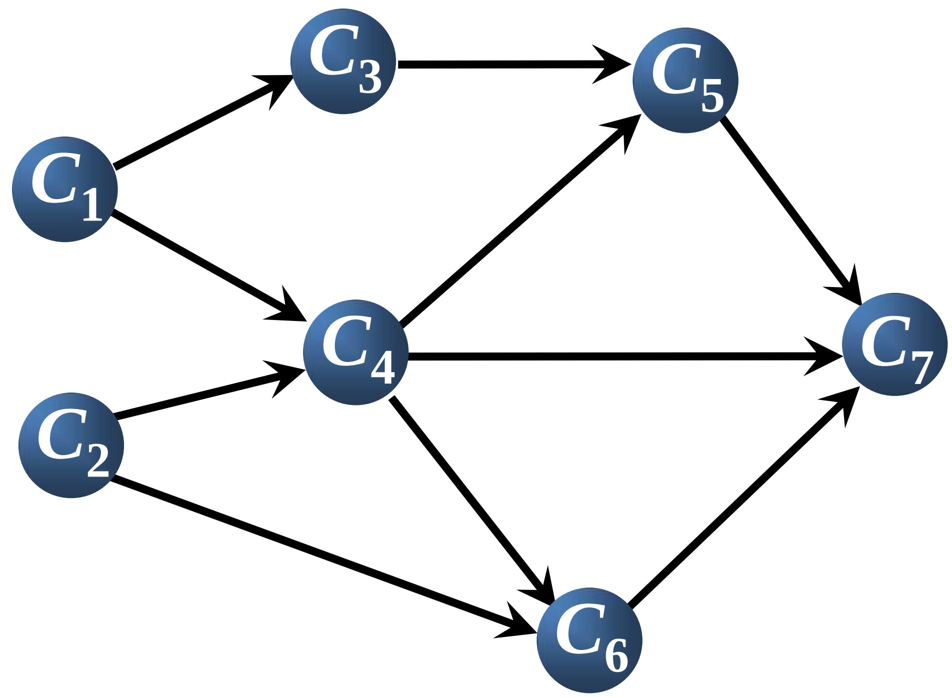


- Example 3

  Course plan problem – Graph Structure

No.	Name	Pre-requisite
C1	Calculus	N/A
C2	Introduction of Computer	N/A
C3	Discrete Mathematics	C1
C4	Program Design	C1, C2
C5	Data Structure	C3, C4
C6	Computer Principle	C2, C4
C7	Database	C4, C5, C6

How to ensure relationship among courses?



- Data Structure is the subject that study on operations and relationships of non-numeric problem.

- With data structure and algorithm, we can build mathematical model for non-numerical problem and write program to get solution.

Algorithms

- What is an algorithm?

•An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

Algorithms

- Properties of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Algorithm Examples

- We will use a pseudocode to specify algorithms, which slightly reminds us of Basic and Pascal.
- Example: an algorithm that finds the maximum element in a finite sequence

- **procedure** max(a_1, a_2, …, a_n: integers)
- max := a_1
- **for** i := 2 **to** n
- **if** max < a_i **then** max := a_i
- {max is the largest element}

Algorithm Examples

- Another example: a linear search algorithm, that is, an algorithm that linearly searches a sequence for a particular element.
- **procedure** linear_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
- i := 1
- **while** ($i \leq n$ and $x \neq a_i$)
- i := i + 1
- **if** $i \leq n$ **then** location := i
- **else** location := 0
- {location is the subscript of the term that equals x, or is zero if x is not found}

Algorithm Examples

- If the terms in a sequence are ordered, a binary search algorithm is more efficient than linear search.

- The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.

Algorithm Examples

binary search for the letter ‘j’

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter ‘j’

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter ‘j’

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter ‘j’

search interval

a c d f g h j l m o p r s u v x z

center element

found !

Algorithm Examples

- **procedure** binary_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
- i := 1 {i is left endpoint of search interval}
- j := n {j is right endpoint of search interval}
- **while** (i < j)
- **begin**
- m := $\lfloor (i + j)/2 \rfloor$
- **if** $x > a_m$ **then** i := m + 1
- **else** j := m
- **end**
- **if** $x = a_i$ **then** location := i
- **else** location := 0
- {location is the subscript of the term that equals x, or is zero if x is not found}

Complexity

- In general, we are not so much interested in the time and space complexity for small inputs.

- For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with n = 10, it is gigantic for $n = 2^{30}$.

Complexity

•For example, let us assume two algorithms A and B that solve the same class of problems.

•The time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.

•For n = 10, A requires 50,000 steps, but B only 3, so B seems to be superior to A.

•For n = 1000, however, A requires 5,000,000 steps, while B requires $2.5 \cdot 10^{41}$ steps.

Complexity

- This means that algorithm B cannot be used for large inputs, while algorithm A is still feasible.

- So what is important is the **growth** of the complexity functions.

- The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

The Growth of Functions

- The growth of functions is usually described using the **big-O notation**.

- **Definition:** Let f and g be functions from the integers or the real numbers to the real numbers.
- We say that f(x) is O(g(x)) if there are constants C and k such that

- $|f(x)| \leq C|g(x)|$

- whenever $x > k$.

The Growth of Functions

- When we analyze the growth of **complexity functions**, f(x) and g(x) are always positive.
- Therefore, we can simplify the big-O requirement to
- $f(x) \leq C \cdot g(x)$ whenever $x > k$.
- If we want to show that f(x) is O(g(x)), we only need to find **one** pair (C, k) (which is never unique).

The Growth of Functions

- The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(x) for large x.
- This boundary is specified by a function g(x) that is usually much **simpler** than f(x).
- We accept the constant C in the requirement
- $f(x) \leq C \cdot g(x)$ whenever $x > k$,
- because **C does not grow with x.**
- We are only interested in large x, so it is OK if $f(x) > C \cdot g(x)$ for $x \leq k$.

The Growth of Functions

Example:

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

For $x > 1$ we have:

$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$

$\Rightarrow x^2 + 2x + 1 \leq 4x^2$

Therefore, for $C = 4$ and $k = 1$:

$f(x) \leq Cx^2$ whenever $x > k$.

$\Rightarrow f(x)$ is $O(x^2)$.

The Growth of Functions

- Question: If $f(x)$ is $O(x^2)$, is it also $O(x^3)$?

- **Yes.** x^3 grows faster than x^2, so x^3 grows also faster than $f(x)$.

- Therefore, we always have to find the **smallest** simple function $g(x)$ for which $f(x)$ is $O(g(x))$.

The Growth of Functions

- "Popular" functions $g(n)$ are
- $n \log n$, 1, 2^n, n^2, $n!$, n, n^3, $\log n$

- Listed from slowest to fastest growth:

- 1
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n
- $n!$

The Growth of Functions

- A problem that can be solved with polynomial worst-case complexity is called **tractable**.

- Problems of higher complexity are called **intractable.**

- Problems that no algorithm can solve are called **unsolvable**.

Complexity Examples

- What does the following algorithm compute?
- **procedure** who_knows($a_1, a_2, \ldots, a_n$: integers)
- $m := 0$
- **for** $i := 1$ to $n-1$
-         **for** $j := i + 1$ to n
-                 **if** $|a_i - a_j| > m$ **then** $m := |a_i - a_j|$
- {m is the maximum difference between any two numbers in the input sequence}
- Comparisons: $n-1 + n-2 + n-3 + \ldots + 1$
-                 $= (n - 1)n/2 = 0.5n^2 - 0.5n$
- Time complexity is $O(n^2)$.

Complexity Examples

Another algorithm solving the same problem:

procedure max_diff(a_1, a_2, …, a_n: integers)
min := a1
max := a1
for i := 2 to n
    if a_i < min **then** min := a_i
    else if a_i > max **then** max := a_i
m := max - min

Comparisons: 2n - 2

Time complexity is O(n).

Time Complexity Analysis

lg N	$\sqrt{N}$	N	$N \lg N$	$N(\lg N)^2$	$N^{3/2}$	N^2
3	3	10	33	110	32	100
7	10	100	664	4414	1000	10000
10	32	1000	9966	99317	31623	1000000
13	100	10000	132877	1765633	1000000	100000000
17	316	100000	1660964	27588016	31622777	10000000000
20	1000	1000000	19931569	397267426	1000000000	1000000000000

seconds

10^2	1.7 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
10^{11}	*never*

operations per second	*problem size 1 million*			*problem size 1 billion*		
	N	$N \lg N$	N^2	N	$N \lg N$	N^2
10^6	seconds	seconds	weeks	hours	hours	never
10^9	instant	instant	hours	seconds	seconds	decades
10^{12}	instant	instant	seconds	instant	instant	weeks

Thank You