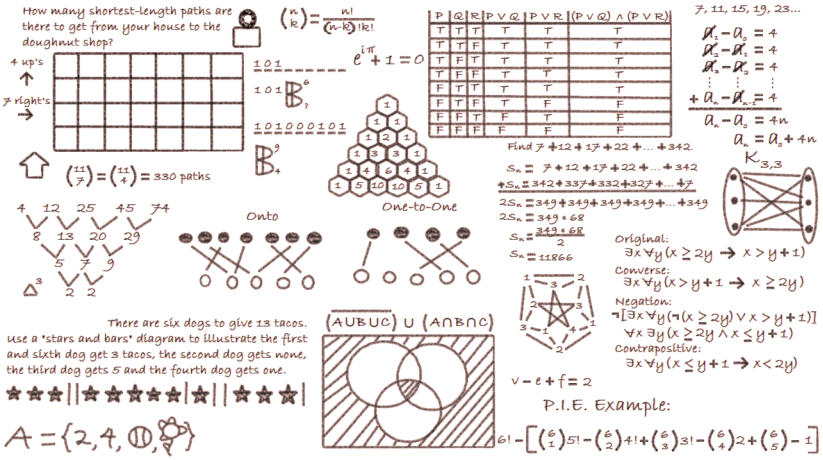
Data Structure And Analysis

(CSC263)

Lecture Notes

Tianquan Di

February 8, 2019



Abstract

Why do we need to learn data structures? This course helps you to explore a new perspective on the algorithm analysis, in the field of Computer Science. CSC263 will also provide you the foundation of technical interviews from your dream tech companies, such as Microsoft, Google, etc. This lecture notes will primarily follow Prof. Sam Toueg's lecture, Prof. David Liu's lecture notes, as well as CLRS. According to Prof. Sam Toueg's description of the course:

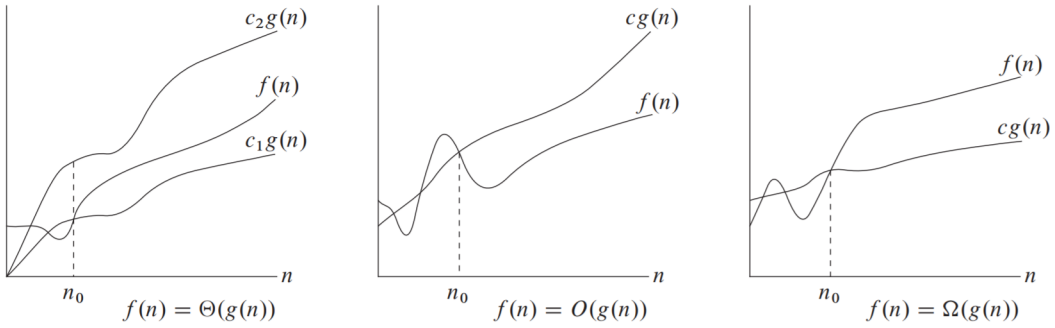
> Data structures are ways of organizing the data involved in computation, suitable for representation in and manipulation by computers. Algorithms are precisely stated, general problem solving methods. Data structures and algorithms are central to computer science. They are also integrally related: neither can be studied fruitfully without knowledge of the other. This course has two goals: First, to learn several important data structures and algorithms, including how to choose and/or modify a data structure to solve various problems; and second, to introduce the basic tools and techniques for the analysis of algorithms and data structures.

Course Website:
http://www.cs.toronto.edu/ sam/teaching/263/
Textbook: Introduction to Algorithms (i.e. CLRS)
This document assumes that you have the knowledge provided from CSC165 and CSC236.

Contents

1 Review of Growth of Functions

1.1 Asymptotic bounds on worst-case time complexity



Let $t(x)$ be the number of steps taken by algorithm $\mathcal{A}$ on the input x.
Let $T(n)$ be the $worst-case$ time complexity of algorithm $\mathcal{A}$. Then:

$$T(n) = \max_{\textbf{all inputs } x \textbf{ of size } n} t(x) = \max\{t(x) : x \text{ is an input of size } n\}$$

To prove that $T(n)$ is $O(g(n))$: we must show that:

$$\exists c > \ 0, \exists n_0 > 0, \forall n > n_0 : T(n) \leq c \cdot g(n)$$

The above statement is equivalent to:
$\Leftrightarrow$ max $\{t(x) : x$ is an input of size $n\} \leq c \cdot g(n)$
$\Leftrightarrow$ For **every** input x of size n, $t(x) \leq c \cdot g(n)$
$\Leftrightarrow$ For **every** input of size n, $\mathcal{A}$ takes ***at most*** $c \cdot g(n)$ steps.

To prove that $T(n)$ is $\Omega(g(n))$: we must show that:

$$\exists c > \ 0, \exists n_0 > 0, \forall n > n_0 : T(n) \geq c \cdot g(n)$$

The above statement is equivalent to:
$\Leftrightarrow$ max $\{t(x) : x$ is an input of size $n\} \geq c \cdot g(n)$
$\Leftrightarrow$ For **some** input x of size n, $t(x) \geq c \cdot g(n)$
$\Leftrightarrow$ For **some** input of size n, $\mathcal{A}$ takes ***at least*** $c \cdot g(n)$ steps.

To prove that $T(n)$ is $\Theta(g(n))$: we must show that:

$$T(n) \text{ is } O(g(n)) \textbf{ and } T(n) \text{ is } \Omega(g(n))$$

For example, the worst-case time complexity for bubble sort is $\Theta(n^2)$. This means that bubble sort is both $O(n^2)$ and $\Omega(n^2)$. Since bubble sort is $O(n^2)$, for **every** input of size n, the algorithm takes ***at most*** $c_1 \cdot n^2$ steps. Since bubble sort is $\Theta(n^2)$, this means that for **some** in put of size n, the algorithm takes ***at least*** $c_2 \cdot n^2$ steps.

2 Priority Queues & Heaps

2.1 Abstract Data Type (ADT) vs. Data Structure

Before we start talking about any else, let us discuss about the definition of **abstract data type** and **data structure**.

> An **abstract data type (ADT)** is a theoretical model of an entity and the set of operations that can be performed on that entity. In other words, it describes an object and its operations.
> A **data structure** is some specific implementation of Abstract Data Type.

2.2 The Priority Queue ADT

Consider the following example, which is the definition for the Priority Quete ADT.

> **The Priority Queue ADT**
> Object: Set S of elements with "keys" (priority) that can be compared
> Operations:
> - **Insert(S, x):** Insert element x into S.
> - **Max(S):** Returns the element with the highest priority (i.e. largest key) in S.
> - **Extract_Max(S, x):** Returns Max(S) and removes it from S.

A classic example of priority queues in practice is a hospital waiting room: more severe injuries and illnesses are generally treated before minor ones, regardless of when the patients arrived at the hospital.[1]

2.3 Heaps

Our goal is to do all the operations above in $\Theta(log(n))$ time. The solution is to use a **data structure** called **Max-Heap**.

Worst Case Time For	Insert	Extract_Max
Unordered Linked List	$\Theta(1)$	$\Theta(n)$
Ordered Linked List	$\Theta(n)$	$\Theta(1)$
Max Heap	$\Theta(log(n))$	$\Theta(log(n))$

Before introducing the max-heap, let us talk about the complete binary tree. Definition of a complete binary tree **complete binary tree (CBT)** is a binary tree such that, for every level L (except the bottom level), the binary tree has 2^L nodes. Also, all the nodes at the bottom level are as far left as possible.

[1] Liu, n.d., 23

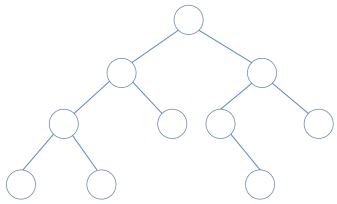


Figure 1: An example of **non-complete** binary tree

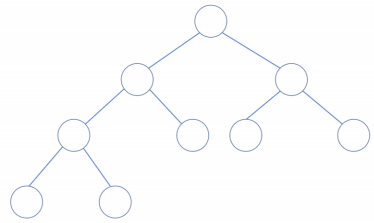


Figure 2: An example of **complete** binary tree

Fact: Height of a CBT with n nodes is $\lfloor log_2(n) \rfloor$

A **max-heap** is a complete binary tree that satisfies **max-heap property**, which is that the value in each internal node is greater than or equal to the values in the children of that node. We typically use an **array** to store a max-heap into the memory.

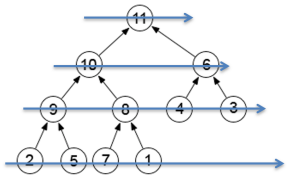


Figure 3: The order of elements in an array implementation of a max-heap.

Notice from the figure that the array is just the **level-order traversal** of the max-heap, from **top to bottom**. Now let us introduce the operations for the max-heap.

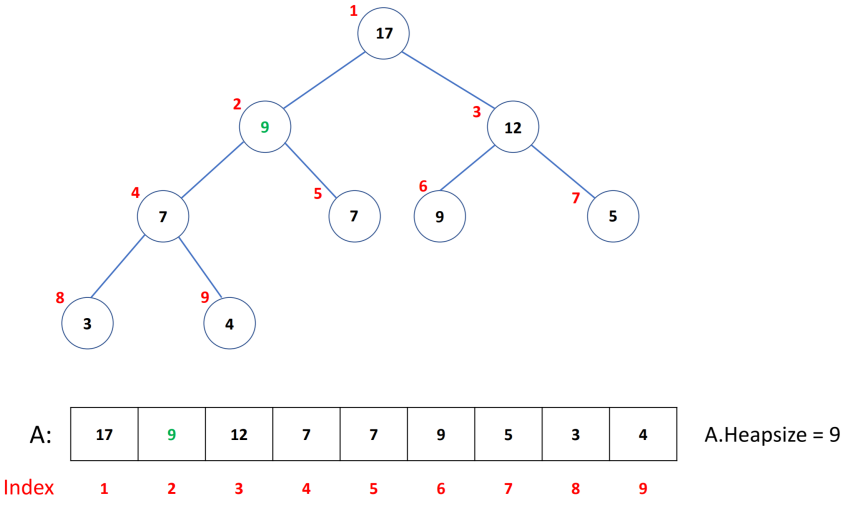


Figure 4: A typical example of array representation of a Max-Heap.

High level idea for operations on Max-Heap

- Maintain the Complete Binary Tree shape.
- Maintain the Max-Heap property.

Insert(S, x): Insert places a key in a new node that is the last node in a level-order-traversal of the heap. The inserted key is then "bubbled" upwards until the heap property is satisfied.

Algorithm 1 The algorithm for insert operation of max-heap.

procedure INSERT(A, x)
  Put x at the bottom left of the tree:
  Increment A.heapsize and and set A[A.heapsize] $= x$
  Bubble x up to the top of the tree:
  while x is not root **AND** priority of $x >$ priority of its parent **do**
    Swap x with its parent

Putting x at the bottom left of the tree **maintains Complete Binary Tree shape**, and bubbling x up to the top of the tree **maintains the Max-Heap property.** Since the max-heap is a CBT with height $\lfloor log_2(n) \rfloor$, bubbling x up the tree takes is $O(log(n))$. Thus, the worst case running time of the algorithm is $O(log(n))$.
Max(S): We simply get the root of the tree. Thus, the Max algorithm is constant runtime.
Extract_Max(S, x):

Algorithm 2 The algorithm for extract-max operation of max-heap.

procedure EXTRACT_MAX(A, x)
  Return the root A[1].
  Remove the returned element from the heap:
    Set A[1] = A[A.heapsize] and decrement A.heapsize
  Drip the element in A[1] down the tree:
    Let x be the element in A[1]
  while x is not a leaf **AND** priority of some child of x >priority of x **do**
    Swap x with the highest-priority of child of x

2.4 Mergeable Priority Queue ADT and Binomial Heaps

Abstract Data Types	Data Structures	Insert	Min	Extract_Min	Union
Priority Queues	Min Heap	Yes	Yes	Yes	**No**
Mergable Priority Queues	Min **Binomial** Heaps	Yes	Yes	Yes	**Yes**

Elements of binomial heaps are stored in a **sequence** of **binomial trees**. (i.e. Binomial Forests)

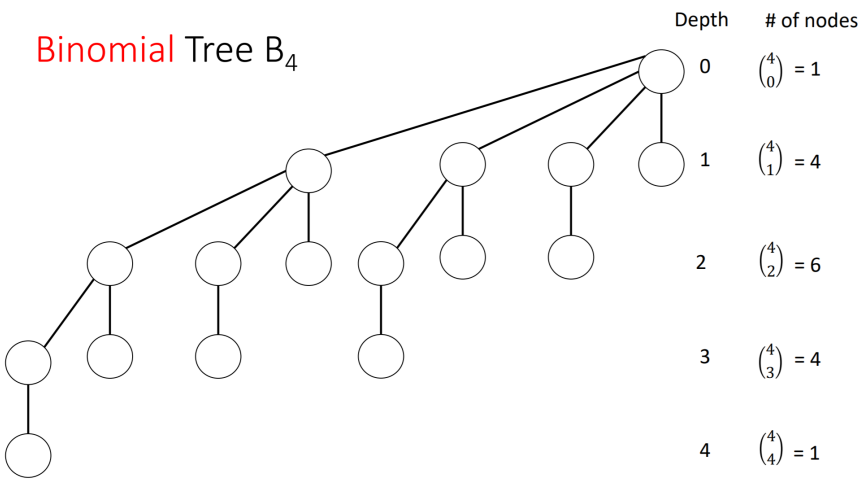


Figure 5: A visualization of binomial tree, B_4.

Properties of Binomial Tree, B_k:

- Height k
- 2^k nodes

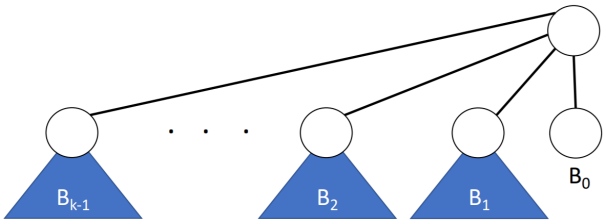


Figure 6: A more general visualization of binomial tree, B_k.

- $\binom{k}{d}$ nodes at depth d

A **Binomial Forest** is a sequence of B_k trees with strictly decreasing k's and a total of n nodes.

Binomial Forest F_n:

- F_n has n nodes.
- $n = < b_t, b_{t-1}, ..., b_0 >_2$. Note that $t = \lfloor log_2(n) \rfloor$
- $F_n = <$ all trees B_i such that bit $b_i = 1 >$

2.5 Implementation of operations on Binomial Heaps

Union(T, Q): $S \leftarrow Union(T, Q)$: Think about binary addition. Additional info about this to be appeared later.

To analyze the worst-case time complexity of Union: Let $|T| \leq n$ and $|Q| \leq n$ (i.e. each contains at most n elements). This implies that each of T, Q have $O(log(n))$ B_k trees. Then Union(T,Q) takes at most $O(log(n))$ key-comparisions. Hence the worst-case time complexity of the algorithm is $O(log(n))$.

Insert(T, x): Simply apply this statement: $S \leftarrow Union(T, x)$

Min(T): Scan the roots of the B_k trees of T and return the smallest key.

Extract_Min(T): The idea of Extract_Min can be expressed as follow.

Algorithm 3 The algorithm for Extract_Min for binomial min heap.

procedure EXTRACT_MIN(T)
// Do Min(T) to locate the smallest element
$U \leftarrow T - B_i$
// Delete root of B_i. By lemma 2, we get a binomial heap S, where:
$S \leftarrow B_i -$ root of B_i
$T \leftarrow Union(U, S)$

3 Dictionary I: AVL Trees

3.1 Abstract Data Type: Dictionary

Object: Set S of (element with) keys

Operations:

- Search(S,x): Returns element with key x, if x is in S; else return "Not Found"
- Insert(S,x): Inserts x into S.
- Delete(S, x): Deletes x from S.

We want all the operations to be done in log time complexity. That is Θ(log n). Possible solution: Binary Search Tree(BST).

3.2 AVL Trees: Intro

Use BST as the data structure for the dictionary ADT can be slow. For example, if we have a balanced BST and we keep inserting node to the rightmost corner. Then the search would take $\Theta(n)$ time.

Balanced BST: We want BST trees with height $\Theta(log\ n)$. So the operations, including search, insert, delete, will have log time complexity. Solution is to use **balanced BSTs**.

AVL Trees: AVL tree is **a type** of self-balancing BST. For AVL Trees, the **balance factor** of each node is always **between -1 and 1.**

Balance Factor(BF) of a node v in AVL Trees:

BF(v) = height(right subtree of v) - height(left subtree of v)

Properties of AVL Trees:

- AVL trees of n nodes has height $\Theta(log\ n)$; height $\leq 1.44\ log_2(n+2)$
- Can do inserts, deletes while maintaining the tree balance in $\Theta(log\ n)$ time.

3.3 AVL Trees: Insert Operation

General Idea:

- Insert x into T as in any BST
- Go up from x to the root:
  - For each node:
    - Adjust the Balance Factor
    - "Rebalance" if $BF > 1$ or $BF < -1$

Specific Implementation:

- Insert x into T as in any BST:
  - x is now a leaf
  - Set BF(x) to 0 (Since it has no children, balance factor is of course 0.)
- Go up **from x to the root** and **for each node v** in this path:
  - Adjust the BF:
    bif x is in right subtree of v: Increment BF(v) if x is in left subtree of v: Decrement BF(v) if BF(v) = 0: Stop
  - Rebalance if necessary:
    - if BF(v) = +2:
      - if BF(v.right) = +1: Do **Left Rotation**, update BFs of rotated nodes, and **stop.**
      - if BF(v.right) = -1: Do **Right-Left Rotation,** update BFs of rotated nodes, and **stop**.
    - if BF(v) = -2:
      - Symmetric to above case.

Exercise: How to implement Delete and Search?

3.4 Augmentation of AVL Trees