# ECE 235: Growth of Functions, Data structures, Thu 9/27/01 (Ch. 3,10)

Pai H. Chou, University of California at Irvine

**This lecture:**

- big $O$, big $\Omega$, and big $\Theta$, how they grow
- fundamental data structures

## 1 Asymptotic Notations

- purpose: express run time as $T(n), n =$ input size
- assumption: $n \in \mathbb{N} = \{0, 1, 2, \ldots\}$ natural numbers

### 1.1 big $O$ = asymptotic upper bound: (page 44)

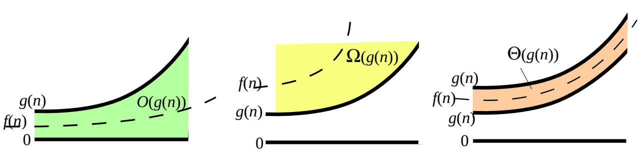
- $f(n) = O(g(n))$ is a *set of functions* that are upper-bounded by $0 \leq f(n) \leq c \cdot g(n)$ for some $n \geq n_0$
- $\Rightarrow f(n)$ grows no faster than $g(n)$
- example: $2n^2 = O(n^3)$, for $c = 1, n_0 \geq 2$
- funny equality, more like a membership: $2n^2 \in O(n^3)$
- $n^2$ and $n^3$ are actually functions, not values.
  $\Rightarrow$ sloppy but convenient

### 1.2 big $\Omega$ = asymptotic lower bound (page 45)

- swap the place between $f(n)$ and $c \cdot g(n)$:
  $0 \leq c \cdot g(n) \leq f(n)$ for some $n \geq n_0$
- example: $\sqrt{n} = \Omega(\lg n)$, for $c = 1, n_0 = 16$

### 1.3 big $\Theta$ = asymptitic tight bounds

- need two constants $c_1$ and $c_2$ to sandwich $f(n)$.
- example: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$ for $c_1 = \frac{1}{4}, c_2 = \frac{1}{2}, n_0 = 8$
- Theorem: ($O$ and $\Omega$) $\Leftrightarrow \Theta$




**Example: insertion sort**

$O(n^2)$ worst case, $\Omega(n)$ best case

$$
\begin{array}{llll}
T(n) = & & c_1 \cdot n & \text{// for loop} \\
& + & c_2 \cdot (n-1) & \text{// } key \leftarrow A \\
& + & \sum_{j=2}^{n} Insert(j) & \text{// } loop\ to\ insert\ j \\
& + & c_8 \cdot (n-1) & \text{// } A[i+1] \leftarrow key
\end{array}
$$

- Best case of Insert($j$) is $\Theta(1)$, Worst is $\Theta(j)$
- Issue: $\Theta(n^2) + \Theta(n)$ is still $\Theta(n^2)$

**Example: MergeSort($A, p, r$)**

where $p, r$ are the lower and upper bound indexes to array $A$.

if $p < r$ then
  $q := \lfloor (p+r)/2 \rfloor$
  MergeSort($A, p, q$), MergeSort($A, q+1, r$)
  Merge($A, p, q, r$)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

$T(n)$ is expressed in terms of $T$ itself! This is called *recurrence*. We want a closed-form solution (no $T$ on the right-hand-side of the equation).

To solve recurrence:

- rewrite as $T(n) = \begin{cases} c & \text{if } n = 1 \\ 2 \cdot T(\frac{n}{2}) + c \cdot n & \text{if } n > 1 \end{cases}$
- $$\begin{array}{lll} T(n) & = 2T(\frac{n}{2}) + cn & \text{Expand} \\ & = 2(2T(\frac{n/2}{2}) + c \cdot \frac{n}{2}) + cn & \\ & = 4T(\frac{n}{4}) + 2c\frac{n}{2} + cn = 4T(\frac{n}{4}) + 2cn & \\ & = 8T(\frac{n}{8}) + 3cn = \ldots & \\ & = (2^i)T(\frac{n}{2^i}) + i \cdot cn & \end{array}$$
- We can divide $n$ by 2 at most $\lg n$ times before $n = 1$. So, $T(n) = (\lg n) \cdot cn + cn = \Theta(n \lg n)$.

Chapter 4 will show how to solve recurrence systematically.

### 1.4 little-$o$, little-$\omega$ not asymptotically tight

$$o(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 : 0 \leq \boxed{f(n) < cg(n)} \forall n \geq n_0\}$$
$$\omega(g(n)) = \{f(n) : \forall c > 0 \exists n_0 > 0 : 0 \leq \boxed{cg(n) < f(n)} \forall n \geq n_0\}$$

*Example*

- $2n^2 = O(n^2)$ is a tight bound $\Rightarrow 2n^2 \neq o(n^2)$.
- $2n = O(n^2)$ is not tight $\Rightarrow 2n = o(n^2)$.
- does not make sense to have little-$\theta$

## 2 Polynomials, log, factorial, fibonacci

### 2.1 Polynomial in $n$ of degree d:

$p(n) = \sum_{i=0}^{d} a_i n_i = \Theta(n^d)$
Polynomially bounded: $f(n) = O(n^k)$ for constant $k$.

### 2.2 logarithm

- useful identity: $a^{\log_b c} = c^{\log_b a}$.
- polylogarithmically bounded if $f(n) = O(\lg^k n)$ for constant $k$
- grows slower than any polynimial: $\lg^b n = o(n^a)$ for any constant $a > 0$.
- "log-star": $\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$
(log of log of log ... of log of $n$) grows very very slowly. Example: $\lg^*(2^{65536}) = 5$.

### 2.3 Factorial:

- Loose bound: $n!$ is faster than $2^n$ but slower than $n^n$
- precisely, $n! = \Theta(n^{n+1/2} e^{-\lg n})$ (Stirling's approximation)
- $\lg(n!) = \Theta(n \lg n)$

### 2.4 Fibonacci

grows exponentially

### 2.5 misc

- $2^{2^n}$ grows faster than $n!$
- $(n+1)!$ grows faster than $n!$
- $n!$ grows faster than $n \cdot 2^n$ faster than $2^n$
- $2^{\lg n} = n$

## 3 Python: Functions

- function definition: **def** *functionName* (*paramList*) **:**
- variables have local scope by default
- control flow: **for**, **while**, **if**
- `range(a, b)` function: returns the list $[a, \ldots\ b-1]$
```
>>> range(1,5)
[1, 2, 3, 4]
```

| *Algorithm from book (p.24)* | *Python code (file `isort.py`)* |
|---|---|
| INSERTION-SORT($A$) | **def** InsertionSort(A)**:** |
| 1 **for** $j \leftarrow 2$ **to** length[$A$] | **for** j **in** range(1, len(A))**:** |
| 2 **do** $key \leftarrow A[j]$ | key = A[j] |
| 3 $i \leftarrow j-1$ | i = j – 1 |
| 4 **while** $i > 0$ and $A[i] > key$ | **while** (i>= 0) **and** (A[i]>key)**:** |
| 5 **do** $A[i+1] \leftarrow A[i]$ | A[i+1] = A[i] |
| 6 $i \leftarrow i-1$ | i = i – 1 |
| 7 $A[i+1] \leftarrow key$ | A[i+1] = key |

- Don't forget the colon **:** for **def**, **for**, and **while** constructs
- Book $A[1 \ldots n] \Rightarrow$ Python $A[0 \ldots n-1]$
Book's $2 \ldots n \Rightarrow$ Python $1 \ldots n-1$.
- line 1: `range(1,len(A))`⇒ [1...len(A)-1]

```
% python
>>> from isort import *      # read file isort.py
>>> x = [2,7,3,8,1]          # create test case
>>> InsertionSort(x)         # call routine
>>> x                        # look at result
[1, 2, 3, 7, 8]
```
Works for other data types, too!

## 4 Data Structures Review

Concepts:

- Stacks: Last-in First-out
- Queue: First-in First-out
- Trees: Root, children

### 4.1 Array implementation

*Stack S: need a* stack pointer *(top)*

| | |
|---|---|
| Push($x$): | `S[++top]` $\leftarrow x$ |
| Pop(): | `return S[top--]` |
| Note: | Actually stacks can grow downward also. |
| Note: | To be safe, check stack underflow/overflow. |

*Queue Q: need 2 pointers (head, tail)*

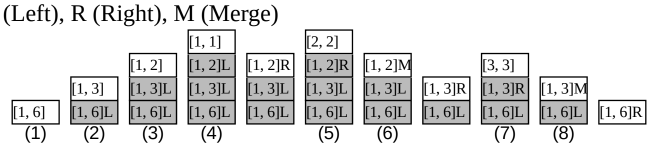
| | |
|---|---|
| Enqueue($x$): | `Q[tail]` $\leftarrow x$ |
| | `tail` $\leftarrow$ `(tail mod len(Q)) + 1` |
| Dequeue(): | `temp` $\leftarrow$ `Q[head]` |
| | `head` $\leftarrow$ `(head mod len(Q)) + 1` |
| | `return temp` |
| Note: | To be safe, check queue underflow/overflow. |

### 4.2 Stack and Calls/Recursion

- Calling routine ⇒ Push local, param, return address
- Return ⇒ Pop and continue

*Example: MergeSort (cf. MergeSort Fig. from last lecture)*

L (Left), R (Right), M (Merge)




### 4.3 Linked List $x$

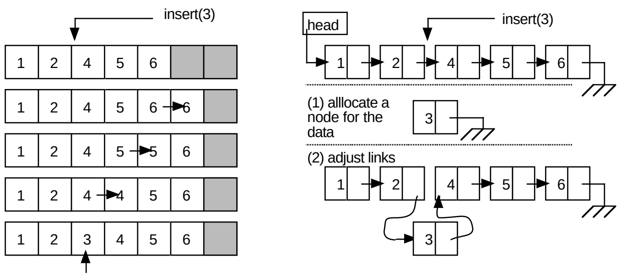
- Singly linked (next) or Doubly linked (next, prev)
- need a *key*
- need a NIL value

*How to implement linked lists?*

- Pre-allocate arrays $prev[1:n], next[1:n], key[1:n]$, and $x$ is an index $\in \{0 \ldots n\}$, where 0 is NIL.
- Records (structures in C, classes in Java): `x.prev`, `x.next`, `x.key`, and `NULL` for NIL.

*Rearrange by adjusting links, not moving data (key)*

- Recall: INSERTIONSORT with array need to copy object to insert.
- Linked list representation: once we know where to insert, adjust link in $O(1)$ time.
- Array version: best case $O(1)$, worst case $O(n)$, average case $O(\frac{n}{2}) = O(n)$.

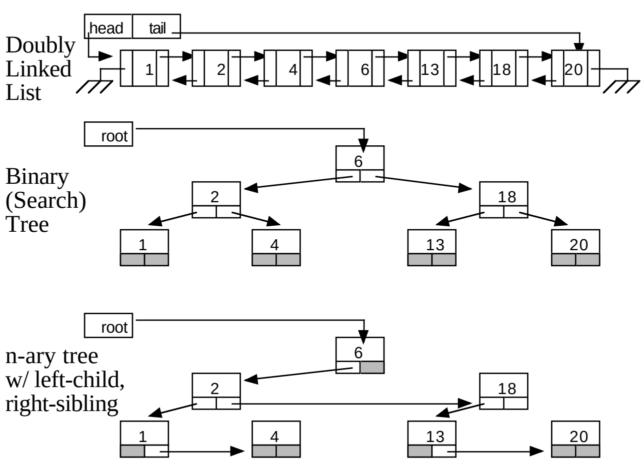



*Search for the item*

- e.g., where to insert?
- list is *unsorted*, worst/average case $O(n)$ time to find it! (compare keys one at a time).
- list is *sorted*:
  - Array: binary search in $O(\lg n)$ times
  - Singly linked list: $O(n)$ avg/worst case, need to do linear search
  - Doubly linked list: does not help! still $O(n)$ – can't jump to the middle of the list, because link is to the head/tail.

### 4.4 Binary search tree

- pointer to root; each node has pointer to left-child, right-child
- optional pointer to parent
- left $\Rightarrow$ smaller, right $\Rightarrow$ bigger key value
- $O(\lg n)$ search, if BALANCED
- Can still be $O(n)$ if not balanced!!
  $\Rightarrow$ Red-black tree (Ch.13) is a way to keep BST balanced
- a *Heap* (Ch.6) implements balanced BST using array; no pointers




### 4.5 rooted trees w/ unbounded branches

Could have arbitrary number of children!

- left-child: head of linked list to children (down one level)
- right-sibling: link to next on same level
- optional link back to parent