SC1007 Data Structures and Algorithms

Analysis of Algorithms

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

Overview

Conduct complexity analysis of algorithms

- Time and space complexities
- Best-case, worst-case and average efficiencies
- Order of Growth
- Asymptotic notations
  - O notation
  - Ω notation (Omega)
  - Θ notation (Theta)
- Efficiency classes

Time and space complexities

- Analyze efficiency of an algorithm in two aspects
  - Time
  - Space

- Time complexity: the amount of time used by an algorithm
- Space complexity: the amount of memory units used by an algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

- Declaration: int x;
- Assignment: x =1;
- Arithmetic operations: +, -, *, /, % etc.
- Logic operations: ==, !=, >, <, &&, ||

These primitive operations take constant time to perform

Basically they are not related to the problem size

changing the input(s) does not affect its computational time

Time Complexity or Time Efficiency

1. Count the number of primitive operations in the algorithm

    i. Repetition Structure: for-loop, while-loop
    ii. Selection Structure: if/else statement, switch-case statement
    iii. Recursive functions

2. Express it in term of problem size

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

```
1: j ← 1
2: factorial ← 1
3: while j ≤ n do
4:     factorial ← factorial * j
5:     j ← j + 1
```

1: c_0

2: c_1

4: c_2

5: c_3

n iterations ➡ $n(c_2+c_3)$

$$f(n) = c_0+c_1+n(c_2+c_3)$$

The function increases linearly with n (problem size)

Time Complexity or Time Efficiency

i. Repetition Structure: for-loop, while-loop

```
1: for j ← 1, m do
2:       for k ← 1, n do
3:             sum ← sum + M[ j ][ k ]
```

c_1

n iterations

$n(c_1)$

m iterations

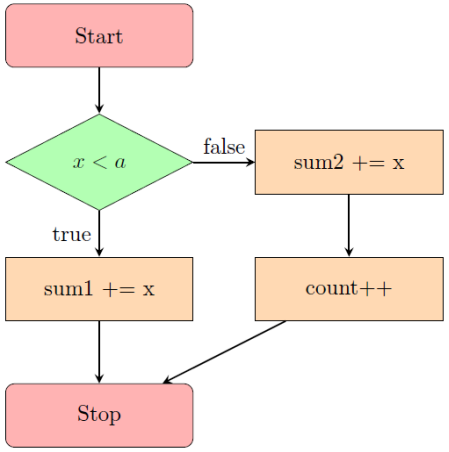
$m(n(c_1))$

The function increases quadratically with n if m==n

*Some constant time operations are ignored here.

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement, switch-case statement




```
1: if(x<a)
2:       sum1 += x;
3: else {
4:       sum2 += x;
5:       count ++;
6:       }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
1: if(x<a)
2:       sum1 += x;
3: else {
4:       sum2 += x;
5:       count ++;
6:       }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. **Best-case analysis** c_1
2. Worst-case analysis
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
1: if(x<a)
2:       sum1 += x;
3: else {
4:       sum2 += x;
5:       count ++;
6:       }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis
2. **Worst-case analysis** c_2
3. Average-case analysis

Time Complexity or Time Efficiency

ii. Selection Structure: if/else statement

```
1: if(x<a)
2:      sum1 += x;
3: else {
4:      sum2 += x;
5:      count ++;
6:      }
```

When x < a, only one primitive operation is executed
When x ≥ a, two primitive operations are executed

How do we analyze the time complexity?

1. Best-case analysis c_1
2. Worst-case analysis c_2
3. Average-case analysis

$$p(x < a)\, c_1 + p(x \geq a) c_2$$
$$= p(x < a)\, c_1 + (1 - p(x < a)) c_2$$

Time Complexity or Time Efficiency

ii. Selection Structure: switch-case statement

```
1: switch(choice){
2:     case 1: compute the summation; break;
3:     case 2: search BST; break;
4:     case 3: print BST; break;
5:     case 4: search for the minimum; break;
6: }
```

$5n$

$6\log_2 n$

$3n$

$4\log_2 n$

Time Complexity

1. Best-case analysis ⋯▸ $C + 4\log_2 n$
2. Worst-case analysis ⋯▸ $C + 5n$
3. Average-case analysis ⋯▸ $C + \sum_{i=1}^{4} p(i)T_i$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of primitive operations in the algorithm
  - Primitive operations in each recursive call
  - Number of recursive calls

```
1 int factorial (int n)
2 {
3     if(n==1) return 1;
4     else return n*factorial(n-1);
5 }
```

c_2

c_1

- *n-1* recursive calls with the cost of c_1.
- The cost of the last call (n==1) is c_2.
- Thus, $c_1(n-1)+c_2$
- It is a linear function

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
1  int count (int array[],int n, int a)
2  {
3      if(n==1)
4            if(array[0]==a)
5               return 1;
6            else return 0;
7      if(array[0]==a)
8           return 1+ count(&array[1], n-1, a);
9      else
10          return count (&array[1], n-1, a);
11 }
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of array[0]==a in the algorithm
  - array[0]==a in each recursive call
  - Number of recursive calls: n-1

```
1  int count (int array[],int n, int a)
2  {
3      if(n==1)
4            if(array[0]==a)
5               return 1;
6            else return 0;
7      if(array[0]==a)
8           return 1+ count(&array[1], n-1, a);
9      else
10          return count (&array[1], n-1, a);
11 }
```

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

$= 1 + 1 + 1 + W_{n-3}$

…

$= 1 + 1 + \ldots + 1 + W_1$

$= (n - 1) + W_1 = n$

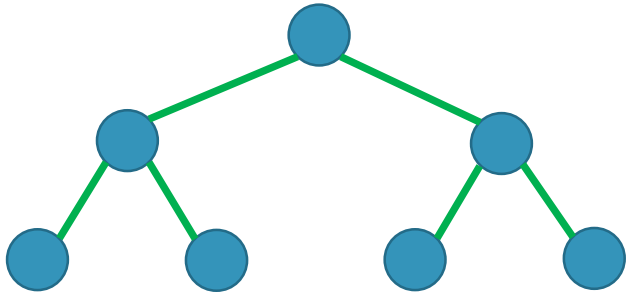
It is known as a **method of backward substitutions**

Time Complexity or Time Efficiency

iii. Recursive functions

- Count the number of multiplication operations in the algorithm

```
1 preorder (simple_t* tree)
2 {
3     if(tree != NULL){
4         tree->item *= 10;
5         preorder (tree->left);
6         preorder (tree->right);
7     }
8 }
```



Geometric Series:

$$
\begin{aligned}
S_n &= a + ar + ar^2 + \ldots + ar^{n-1} \\
rS_n &= ar + ar^2 + \ldots + ar^{n-1} + ar^n \\
(1-r)S_n &= a - ar^n \\
S_n &= \frac{a(1-r^n)}{1-r}
\end{aligned}
$$

$W_0 = 0$

$W_1 = 1$

$W_2 = 1 + W_1 + W_1 = 3$

$W_3 = 1 + W_2 + W_2$

$= 1 + 2\,(1 + W_1 + W_1)$

$= 1 + 2\,(1 + 2)$

$= 1 + 2 + 4 = 7$

$W_{k-1} = 1 + 2 \cdot W_{k-2}$

$= 1 + 2 + 4 + 8 + \ldots + 2^{k-2}$

$W_k = 1 + 2 \cdot W_{k-1} = 1+2+4+8+\ldots+2^{k-1}$

$= \frac{1-2^k}{1-2} = 2^k - 1$

Prove the hypothesis can be done by mathematical induction

It is known as a **method of forward substitutions**

Series

- Geometric Series

$$G_n = \frac{a(1-r^n)}{1-r}$$

- Arithmetic Series

$$A_n = \frac{n}{2}[2a + (n-1)d] = \frac{n}{2}[a_0 + a_{n-1}]$$

- Arithmetico-geometric Series

$$\sum_{t=1}^{k} t2^{t-1} = 2^k(k-1) + 1$$

- Faulhaber's Formula for the sum of the p-th powers of the first n positive integers

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

*Derivation is in note section 0.7.4.1

Cubic Time Complexity

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

```
1      for (i=1; i<=n; i++)
2          M[i] = 0;
3          for (j=i; j>0; j--)
4              for (k=i; k>0; k--)
5                  M[i] += A[j]*B[k];
```

- In each outer loop, both j and k are assigned by value of i.
- Inner loops takes i^2 iterations
- The overall number of iterations is

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2$$

$$= \frac{n(n+1)(2n+1)}{6}$$

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10						
100						
10^4						
10^6						

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013					
10^4	.13					
10^6	13					

Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086				
10^4	.13	.173				
10^6	13	259				

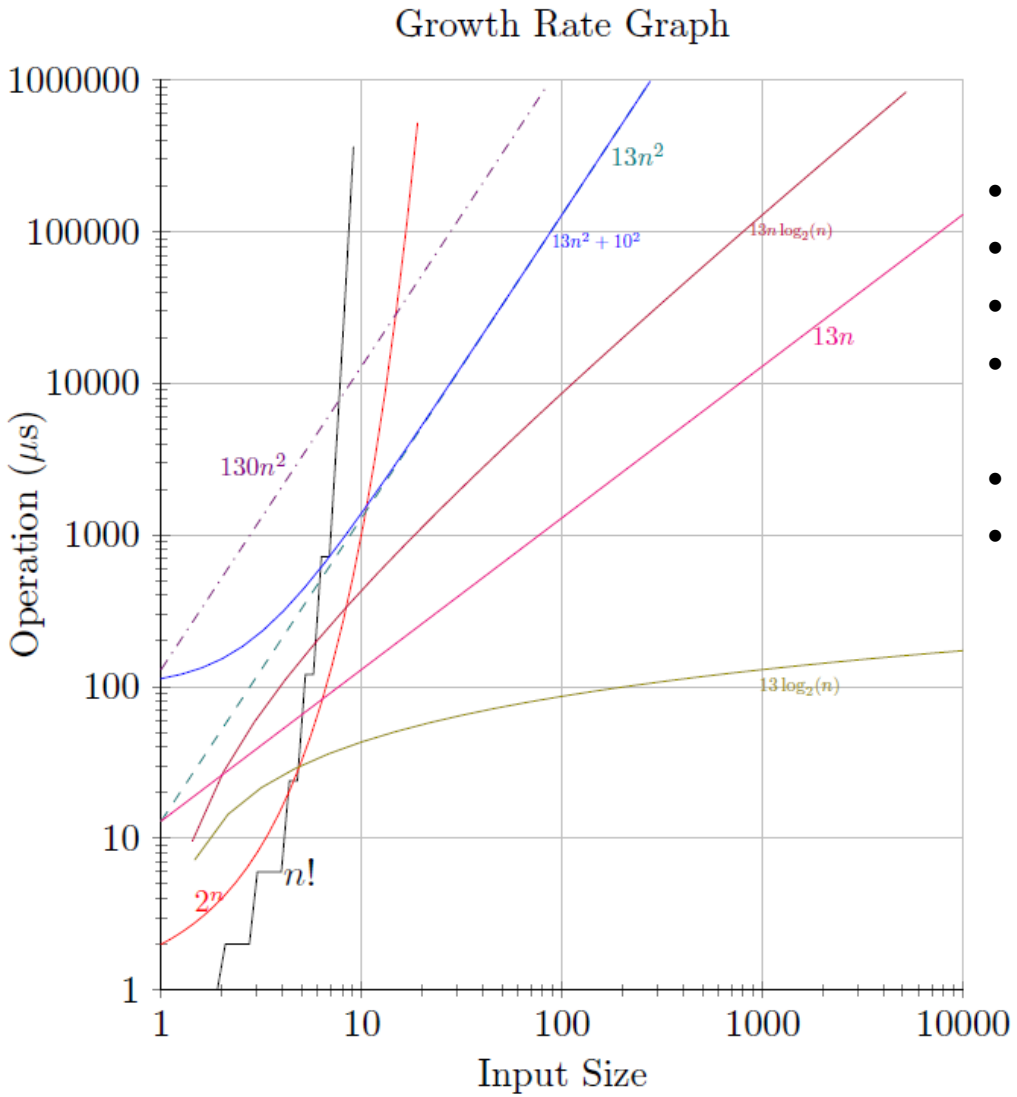
Order of Growth

Algorithm	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Problem size (n)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4x10^{16}$years
10^4	.13	.173	22 mins	3.61hrs	22mins	
10^6	13	259	150 days	1505 days	150days	

Order of Growth

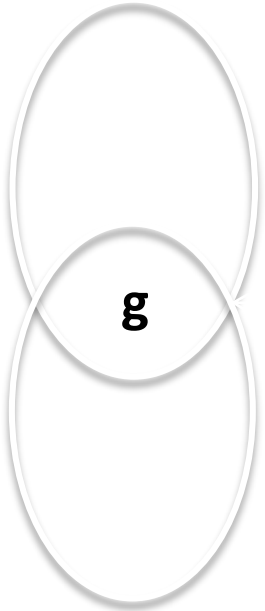


- n! is the fastest growth
- 2^n is the second
- 13n is linear
- $13\log_2 n$ is the slowest

- 10^2 can be ignored when n is large
- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ slightly faster

Asymptotic Notations

- Big-Oh (**O**) , Big-Omega (**Ω**) and Big-Theta (**Θ**) are asymptotic (set) notations used for describing the order of growth of a given function.



$f \in \Omega(g)$ Set of functions that grow at higher or same rate as **g**

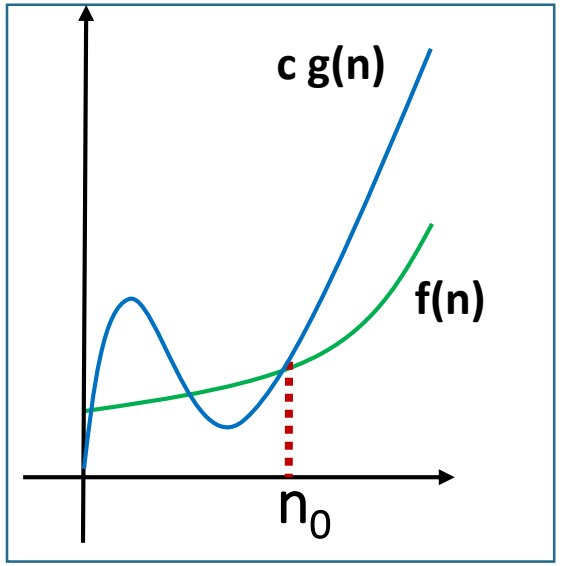
$f \in \Theta(g)$ Set of functions that grow at same rate as **g**

$f \in O(g)$ Set of functions that grow at lower or same rate as **g**

Big-Oh Notation (O)

Definition 3.1 *$\mathcal{O}$-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\mathcal{O}(g(n))$, denoted $f(n) \in \mathcal{O}(g(n))$, if $f(n)$ is **bounded above** by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\mathcal{O}(g(n)) = \{f(n) : \exists \text{positive constants, } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$



f(n) = 4n + 3 and g(n) = n

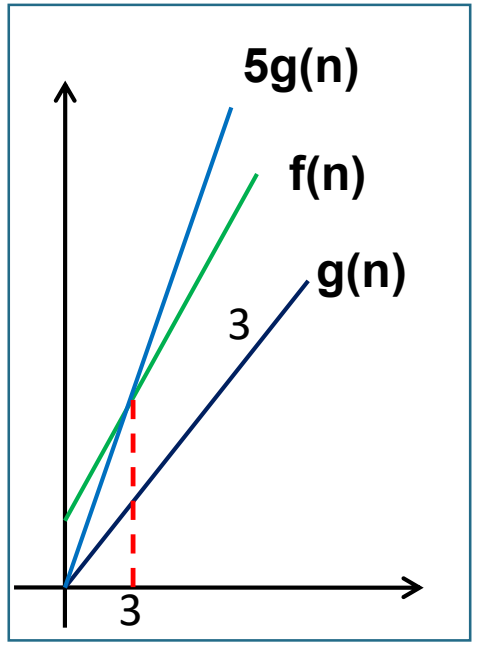
Let c = 5, n0 = 3

$$f(n) = 4n + 3$$

$$4n + 3 \leq 5n \qquad \forall n \geq 3$$

$$f(n) \leq 5g(n) \qquad \forall n \geq 3$$

$$f(n) = O(g(n)) \quad i.e.\, 4n + 3 \in O(n)$$



Big-Oh Notation (O)

Definition 3.1 *$\mathcal{O}$-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\mathcal{O}(g(n))$, denoted $f(n) \in \mathcal{O}(g(n))$, if $f(n)$ is **bounded above** by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\mathcal{O}(g(n)) = \{f(n) : \exists \text{positive constants, } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

f(n) = 4n + 3 and g(n) = n^3

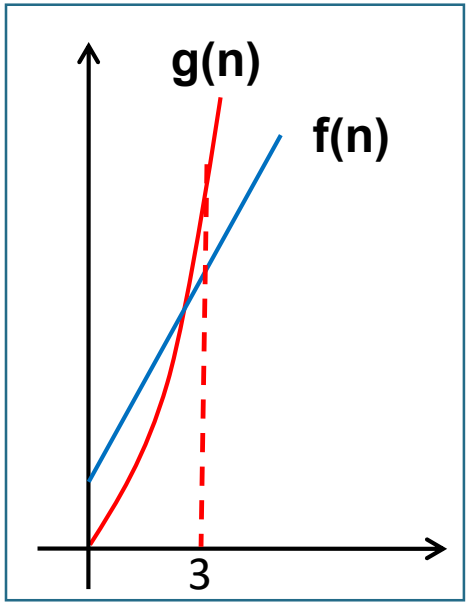
Let c = 1, n0 = 3

$$f(n) = 4n + 3$$

$$4n + 3 \leq n^3 \qquad \forall n \geq 3$$

$$f(n) \leq 5g(n) \quad \forall n \geq 3$$

$$\Rightarrow \quad f(n) = O(g(n)) \qquad i.e. 4n + 3 \in O(n^3)$$



If $f(n) = O(g(n))$, we say

$g(n)$ is asymptotic upper bound of $f(n)$

Big-Oh Notation (O) – Alternative definition

Definition 3.2 *$\mathcal{O}$-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$, then $f(n) \in \mathcal{O}(g(n))$ or $f(n) = \mathcal{O}(g(n))$.*

f(n) = 4n + 3 and g(n) = n

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3n}{n} = 4 < \infty$$

$f(n) = O(g(n))$ $i.e. 4n + 3 \in O(n)$

f(n) = 4n + 3 and g(n) = n^3

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n+3n}{n^3} = 0 < \infty$$

$f(n) = O(g(n))$ $i.e. 4n + 3 \in O(n^3)$

Big-Omega Notation (Ω)

Definition 3.3 *Ω-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\Omega(g(n))$, denoted $f(n) \in \Omega(g(n))$, if $f(n)$ is **bounded below** by some constant multiple of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\Omega(g(n)) = \{f(n) : \exists \text{positive constants, } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0\}$$

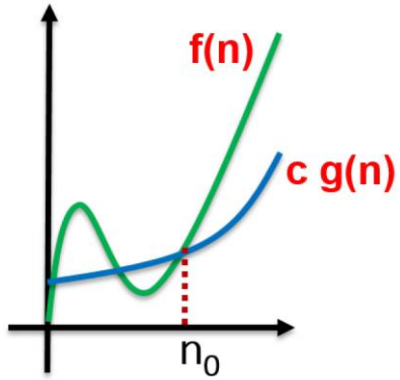
Definition 3.4 *Ω-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, then $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$.*

f(n) = 4n + 3 and g(n) = 5n

Let c=1/5, n_0 =0

f(n) ≥ (1/5)g(n)

4n+3 ≥ (1/5)5n for all n≥0



If $f(n) = \Omega(g(n))$, we say

$g(n)$ is asymptotic lower bound of $f(n)$

Big-Theta Notation (Θ)

Definition 3.5 *Θ-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, $f(n)$ is said to be in $\Theta(g(n))$, denoted $f(n) \in \Theta(g(n))$, if $f(n)$ is* ***bounded both above and below*** *by some constant multiples of $g(n)$ for all large n, i.e., the set of functions can be defined as*

$$\Theta(g(n)) = \{f(n) : \exists \text{positive constants, } c_1, c_2 \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$$

Definition 3.6 *Θ-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$, then $f(n) \in \Theta(g(n))$ or $f(n) = \Theta(g(n))$.*

If $f(n) = \Theta(g(n))$, we say

$g(n)$ is asymptotic tight bound of $f(n)$

Summary of Limit Definition

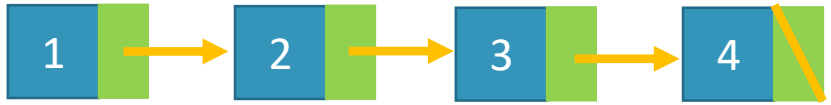
	$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
$\mathbf{0}$	✓		
$\mathbf{0 < c < \infty}$	✓	✓	✓
∞		✓	

Time Complexity of Sequential Search

```
1 pt=head;
2 while (pt->key != a){
3    pt = pt->next;
4    if(pt == NULL) break;
5 }
```

c_1

c_2 (n-1) iterations



Assume that the search key ***a*** is always in the list

1. Best-case analysis: c_1 when **a** is the first item in the list => Θ (1)
2. Worst-case analysis: $c_2 \cdot (n-1) + c_1$ => Θ (n)
3. Average-case analysis
   - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))$$

$$= \frac{1}{n}[nc_1 + c_2\sum_{i=1}^{n}(i-1)]$$

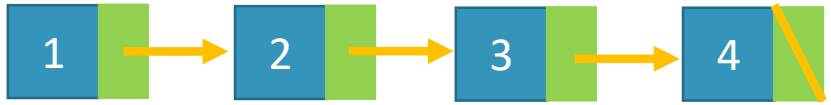
$$= c_1 + \frac{c_2}{n}\cdot\frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2} = \Theta(n)$$

Time Complexity of Sequential Search

```
1 pt=head;
2 while (pt->key != a){
3    pt = pt->next;
4    if(pt == NULL) break;
5 }
```

c_1

c_2 n iterations



3. Average-case analysis
   - Assumed that every item in the list has an equal probability as a search key

$$\frac{1}{n}[c_1 + (c_1 + c_2) + (c_1 + 2c_2) + \cdots + (c_1 + (n-1)c_2)] = \frac{1}{n}\sum_{i=1}^{n}(c_1 + c_2(i-1))$$

$$= \frac{1}{n}[nc_1 + c_2\sum_{i=1}^{n}(i-1)]$$

$$= c_1 + \frac{c_2}{n}\cdot\frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2}$$

If the search key, a, is not in the list, then the time complexity is

$$c_1 + nc_2 = \Theta(n)$$

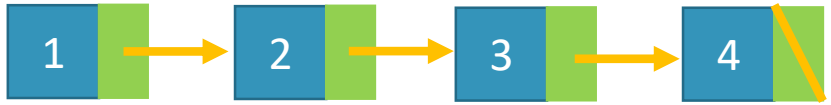
Since the probability of the search key is in the list is unknown, we only can have

$$T(n) = P(a \text{ in the list})(c_1 + \frac{c_2(n-1)}{2}) + (1 - P(a \text{ in the list}))(c_1 + nc_2)$$

Hence, it is a linear function. **Θ (n)**

Time Complexity of Sequential Search

```
1 pt=head;
2 while (pt->key != a){
3    pt = pt->next;
4    if(pt == NULL) break;
5 }
```



- The data is stored in unordered
- To search a key, every element is required to read and compare
- This is a brute-force approach or a näive algorithm
- Its time complexity is **O(n)**

- How can we improve it?

Asymptotic Notation in Equations

When an asymptotic notation appears in an equation, we interpret it as standing for some anonymous function that we do not care to name.

Examples:

- $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
- $T(n) = T(n/2) + \Theta(n)$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

Simplification Rules for Asymptotic Analysis

1. If $f(n) = O(cg(n))$ for any constant c > 0, then $f(n) = O(g(n))$
2. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

   e.g. $f(n) = 2n$, $g(n) = n^2$, $h(n) = n^3$

3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,

   then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

   e.g. $5n + 3\log_2 n = O(n)$

4. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$

   then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$

   e.g. $f_1(n) = 3n^2 = O(n^2)$, $f_2(n) = \log_2 n = O(\log_2 n)$

   Then $3n^2 \log_2 n = O(n^2 \log_2 n)$

Properties of Asymptotic Notation

- Reflexive of O, Ω and Θ

$$f(n) = O\big(f(n)\big)$$
$$f(n) = \Omega(f(n))$$
$$f(n) = \Theta(f(n))$$

- Symmetric of Θ

$$f(n) = \Theta(g(n))$$
$$\Longrightarrow g(n) = \Theta(f(n))$$

- Transitive of O, Ω and Θ

$$f(n) = O\big(g(n)\big) \text{ and } g(n) = O\big(h(n)\big)$$
$$\Longrightarrow f(n) = O(h(n))$$
$$f(n) = \Omega\big(g(n)\big) \text{ and } g(n) = \Omega\big(h(n)\big)$$
$$\Longrightarrow f(n) = \Omega(h(n))$$
$$f(n) = \Theta\big(g(n)\big) \text{ and } g(n) = \Theta\big(h(n)\big)$$
$$\Longrightarrow f(n) = \Theta(h(n))$$

Common Complexity Classes

Order of Growth	Class	Example
1	Constant	Finding midpoint of an array
$\log_2 n$	Logarithmic	Binary Search
n	Linear	Linear Search
$n\log_2 n$	Linearithmic	Merge Sort
n^2	Quadratic	Insertion Sort
n^3	Cubic	Matrix Inversion (Gauss-Jordan Elimination)
2^n	Exponential	The Tower of Hanoi Problem
n!	Factorial	Travelling Salesman Problem

When time complexity of algorithm A grows faster than algorithm B for the same problem, we say A is inferior to B.

Space Complexity

- Determine number of entities in problem (also called problem size)
- Count number of basic units in algorithm

- Basic units
- Things that can be represented in a constant amount of storage space
- E.g. integer, float and character.

Space Complexity

- Space requirements for an array of n integers - $\Theta(n)$
- If a matrix is used to store edge information of a graph,

  i.e. G[x][y] = 1 if there exists an edge from x to y,

  space requirement for a graph with n vertices is $\Theta(n^2)$

Space/time tradeoff principle

- Reduction in time can be achieved by sacrificing space and vice-versa.

CX1107
Data Structures and Algorithms

Advanced Linked Lists, Stack and Queue

Algebraic Expression Conversions

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

So Far …

Dynamic Memory Management

- **`#include <stdlib.h>`**
- **`malloc()`** dynamically memory allocation.
- **`free()`** deallocate memory

Linked List

```
struct _listnode
{
    int item;
    struct _listnode *next;
};
typedef struct _listnode ListNode;
```

Interface Functions

1. **Display:** **`printList()`**
2. **Search:** **`findNode()`**
3. **Insert:** **`insertNode()`**
4. **Delete:** **`removeNode()`**
5. **Size:** **`sizeList()`**

Linked List vs Array

1. **Display: Both are similar**
2. **Search: Array is better**
3. **Insert and Delete: Linked List is more flexible**
4. **Size: Array is better**

Interface Functions

1. Display: printList()
2. Search: findNode()
3. Insert: insertNode()
4. Delete: removeNode()
5. Size: sizeList()

...

```
1  void printList(ListNode *cur){
2      while (cur != NULL){
3          printf("%d\n", cur->item);
4          cur = cur->next;
5      }
6  }
```

```
1  int sizeList(ListNode *head){
2      int count = 0;
3      while (head != NULL){
4        count++;
5        head = head->next;
6      }
7      return count;
8  }
```

```
1  ListNode *findNode(ListNode* cur, int i){
2     if (cur==NULL || i<0)
3        return NULL;
4     while(i>0){
5        cur=cur->next;
6        if (cur==NULL)
7           return NULL;
8        i--;
9     }
10    return cur;
11 }
```

```
1  int insertNode(ListNode **ptrHead, int i, int item){
2      ListNode  *pre, *newNode;
3      if (i == 0){
4          newNode = malloc(sizeof(ListNode));
5          newNode->item = item;
6          newNode->next = *ptrHead;
7          *ptrHead = newNode;
8          return 1;
9      }
10     else if ((pre = findNode(*ptrHead, i-1)) != NULL){
11         newNode = malloc(sizeof(ListNode));
12         newNode->item = item;
13         newNode->next = pre->next;
14         pre->next = newNode;
15         return 1;
16     }
17     return 0;
18 }
```

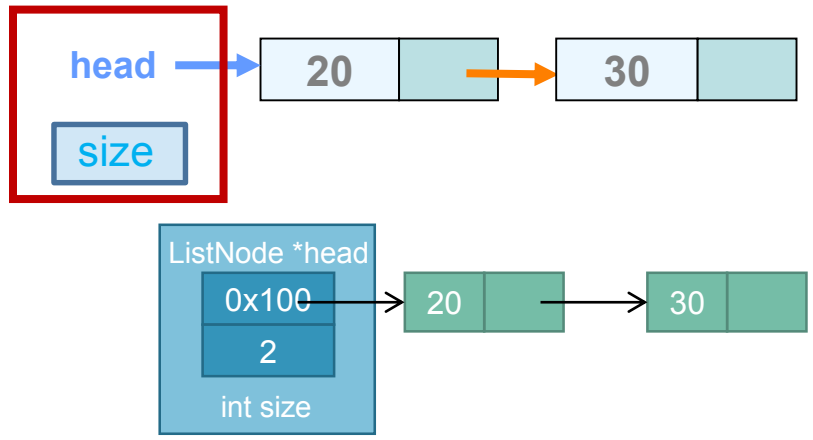
Can we improve our sizeList()?

```
1  int sizeList(ListNode *head){
2      int count = 0;
3      while (head != NULL){
4        count++;
5        head = head->next;
6      }
7      return count;
8  }
```

- Solution:
  - Define another C struct, LinkedList
  - Wrap up all elements that are required to implement the Linked List data structure

```
typedef struct _linkedlist{
    ListNode *head;
    int size;
} LinkedList;
```

```
1  int sizeList(LinkedList ll){
2      return ll.size;
3  }
```



- Remember to change size when adding/removing nodes

Linked list functions using LinkedList struct

- Original function prototypes:
  - void printList(ListNode *head);
  - ListNode *findNode(ListNode *head);
  - int insertNode(ListNode **ptrHead, int i, int item);
  - int removeNode(ListNode **ptrHead, int i);
- New function prototypes:
  - **void printList(LinkedList ll);**
  - **ListNode *findNode(LinkedList ll, int i);**
  - **int insertNode(LinkedList *ll, int index, int item);**
  - **int removeNode(LinkedList *ll, int i);**

```
typedef struct _linkedlist{
   ListNode *head;
   int size;
}LinkedList;
```

```
1  ListNode *findNode(ListNode* cur, int i){
2     if (cur==NULL || i<0)
3        return NULL;
4     while(i>0){
5        cur=cur->next;
6        if (cur==NULL)
7           return NULL;
8        i--;
9     }
10    return cur;
11 }
```

```
1  ListNode *findNode(LinkedList ll, int i){
2      ListNode *temp = ll.head;
3      if (cur==NULL || i < 0|| i >ll.size)
4          return NULL;
5
6      while (i > 0){
7          temp = temp->next;
8          if (temp == NULL)
9              return NULL;
10         i--;
11     }
12     return temp;
13 }
```

Overview

1. **Variations of the Linked List**
   - **Doubly-linked Lists**
   - **Circular Linked Lists**
   - **Circular Doubly-linked Lists**
2. **Stack**
3. **Queue**
4. **Application of Stack**

Advanced Linked List

Variations of the Linked List

- **Doubly-linked Lists**
- **Circular Linked Lists**
- **Circular Doubly-linked Lists**

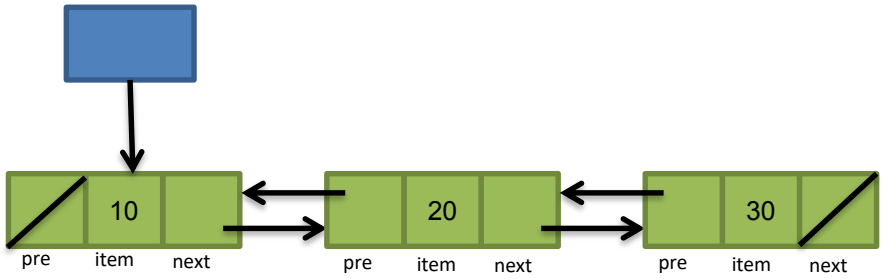
Doubly Linked List

- Singly Linked list: Only one link. Traversal of the list is one way only.

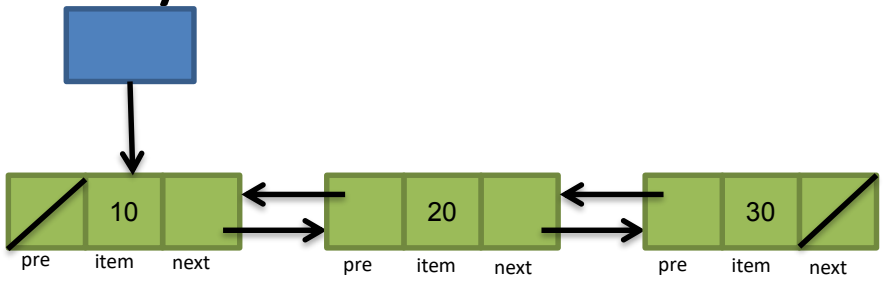
```
struct _listnode
{
    int item;
    struct _listnode *next;
};
typedef struct _listnode ListNode;
```

- Doubly Linked List: two links in each node. It can search forward and backward

```
struct _dbllistnode
{
    int item;
    struct _dbllistnode *pre;
    struct _dbllistnode *next;
};
typedef struct _dbllistnode DblListNode;
```



Doubly Linked List

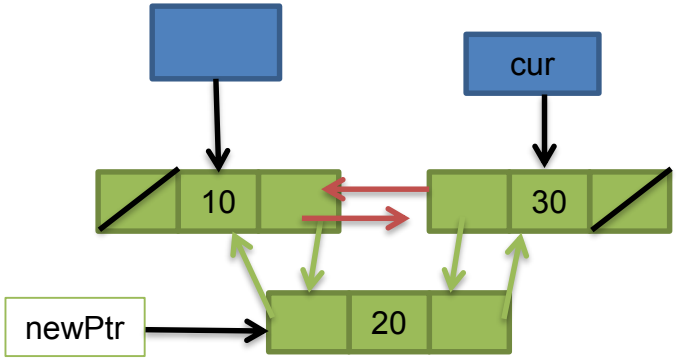


Interface Functions

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`

- Display, Search and Size functions are similar to the Singly Linked List's
- Insert function:

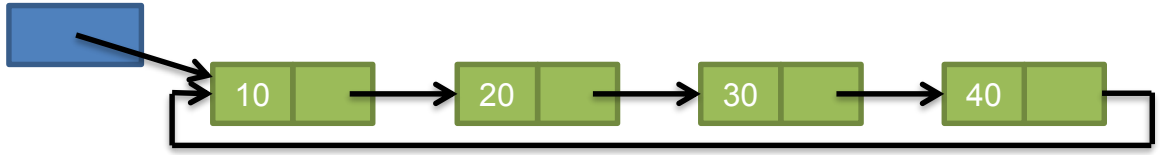
```
newPtr->next = cur;
newPtr->pre = cur->pre;
cur->pre= newPtr;
newPtr->pre->next=newPtr;
```



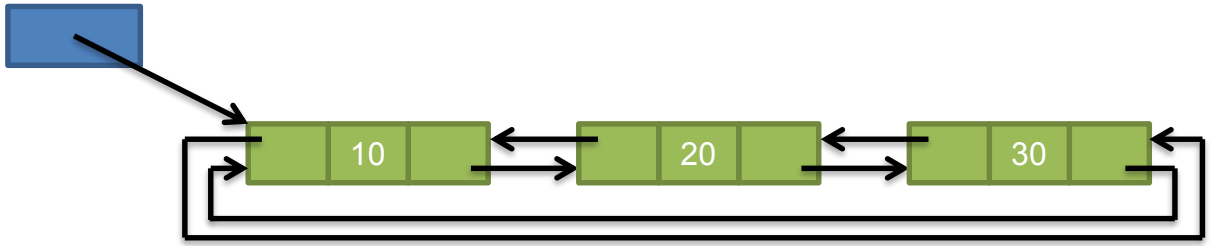
- It is noted that the solution is not unique.
- Delete function will be easier than Singly Linked List's.

Circular linked lists

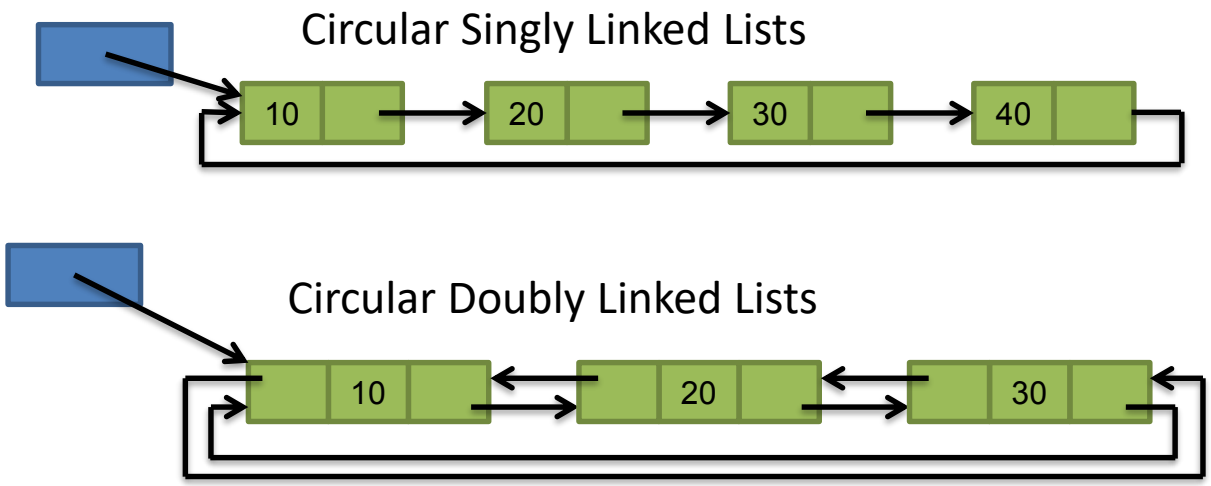
- Circular singly linked lists
  - Last node has next pointer pointing to first node



- Circular doubly linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node



Circular Linked List



Interface Functions

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`

- **Display, Search, Size**: the last node's link is equal to head instead of NULL
- **Insert** and **Delete**: there is no special case at first or last position

Advanced Linked List

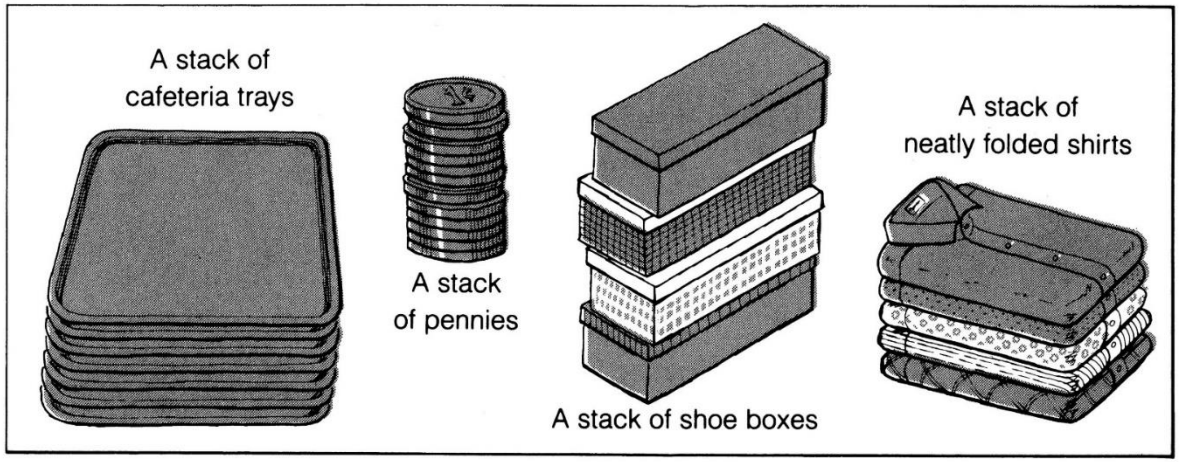
Stack and Queue

What is Stack?

What is Queue?

Stack

- Elements are added to and removed from the top



- A Last-In, First-Out (LIFO) a.k.a First-In, Last-Out (FILO) data structure
- Can be implemented by array or linked list

Queue

- Elements are added only at the tail and removed from the head

- A First-In, First-Out (FIFO) a.k.a Last-In, Last-Out (LILO) data structure
- Can be implemented by array or linked list

Linked List to Stack and Queue

```
struct _listnode
{
     int item;
     struct _listnode *next;
} ListNode;
```

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(),size`

```
typedef struct _linkedlist{
     ListNode *head;
     int size;
} LinkedList;
```

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

Stack

1. ~~Display: `printStack()`~~
2. Retrieve : `peek()`
3. Insert: `push()`
4. Delete: `pop()`
5. Size: `isEmptyStack()`

Queue

1. ~~Display: `printQueue()`~~
2. Retrieve: `getFront()`
3. Insert: `enqueue()`
4. Delete: `dequeue()`
5. Size: `isEmptyQueue()`

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. Size: `isEmptyStack()`

Stack Functions

- Peek(): Inspect the item at the top of the stack without removing it
- Push(): Add an item to the top of the stack
- Pop(): Remove an item from the top of the stack
- IsEmptyStack(): Check if the stack has no more items remaining

- In short, users only can get access to the top of the stack. It is a FILO data structure.

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(), size`

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. Size: `isEmptyStack()`

peek()

- Peek the top of the stack-> return the item on the top
- Here we assume that s.head is not NULL.
- If you would like to validate s.head, then prototype of peek() need to be redefined eg. `int peek(Stack s, int* itemPtr);`

```
int peek(Stack s){
    return s.head->item;
}
```

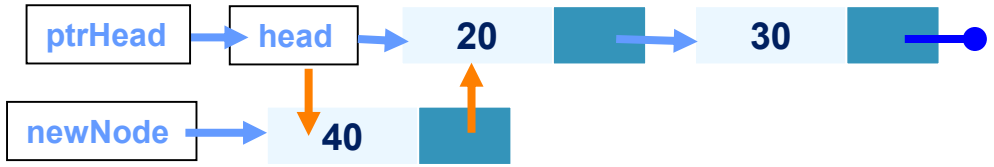
push()

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. **Insert: `push()`**
3. Delete: `pop()`
4. Size: `isEmptyStack()`

```
int insertNode2(LinkedList *ll, int index, int item){
    ListNode  *pre, *newNode;
    if (index == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = ll->head;

        ll->head = newNode;
        ll->size++;
        return 1;
    }
    else if ((pre = findNode2(*ll, index-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        ll->size++;
        return 1;
    }
    return 0;
}
```



- Push a new node onto the stack-> insert a node at index 0

```
void push(Stack *sPtr, int item){

    insertNode2(sPtr, 0, item);

}
```

```
void push(Stack *sPtr, int item){
    StackNode *newNode;
    newNode = malloc(sizeof(StackNode));
    newNode->item = item;
    newNode->next = sPtr->head;

    sPtr->head = newNode;
    sPtr->size++;

}
```

pop()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(), size`

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. **Delete: pop()**
4. Size: `isEmptyStack()`

Note:
return value of removeNode() is SUCCESS (1) or FAILURE (0)

```
int removeNode(LinkedList *ll, int index);
```

- Pop a node from the stack-> Remove a node at index 0 and return SUCCESS (1) or FAILURE (0)
- Here the removal node is freed directly
- Use Peek() to retrieve it first

```
int pop(Stack *sPtr){
return removeNode(sPtr, 0);
}
```

```
int pop(Stack *s){
    if(sPtr==NULL || sPtr->head==NULL){
            return 0;
        }
        else{
            StackNode *temp = sPtr->head;
            sPtr->head = sPtr->head->next;
            free(temp);
            sPtr->size--;
            return 1;
        }
}
```

isEmptyStack()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. **Size**: `sizeList(), size`

Stack

```
typedef ListNode StackNode;
typedef LinkedList Stack;
```

1. Retrieve: `peek()`
2. Insert: `push()`
3. Delete: `pop()`
4. **Size: `isEmptyStack()`**

- Check whether the stack is empty? 1 == empty: 0 == not empty

```
int isEmptyStack(Stack s){
    if (s.size == 0) return 1;
    return 0;
}
```

Linked List to Stack and Queue

```
struct _listnode
{
     int item;
     struct _listnode *next;
} ListNode;
```

```
typedef struct _linkedlist{
     ListNode *head;
     int size;
} LinkedList;
```

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(),size`

```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

Stack

1. ~~Display: `printStack()`~~
2. Retrieve : `peek()`
3. Insert: `push()`
4. Delete: `pop()`
5. Size: `isEmptyStack()`

Queue

1. ~~Display: `printQueue()`~~
2. Retrieve: `getFront()`
3. Insert: `enqueue()`
4. Delete: `dequeue()`
5. Size: `isEmptyQueue()`

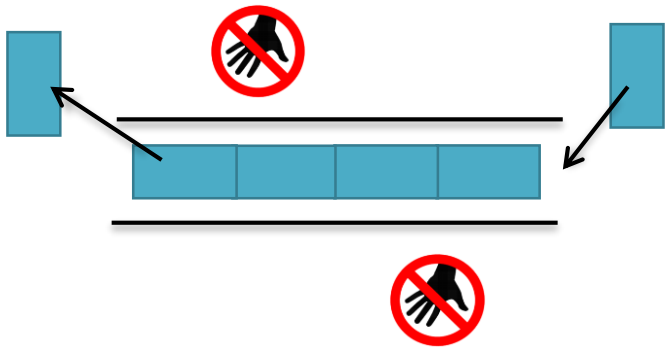
Queue Function

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

- getFront(): Inspect the item at the front of the queue without removing it
- enqueue(): Add an item at the end of the queue
- dequeue(): Remove an item from the top of the queue
- IsEmptyQueue(): Check if the queue has no more items remaining
- In short, users only can add from the back and remove from the front of the linked list. It is a FIFO data structure.
- Due to algorithmic efficiency, ***tail** is introduced



getFront()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(), size`

```
typedef struct _linkedlist{
    ListNode *head;
    int size;
} LinkedList;
```

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
    int size;
    ListNode *head;
    ListNode *tail;
} Queue;
```

1. **Retrieve: `getFront()`**
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

- Inspect the front of the queue-> return the item at the front

```
int getFront(Queue q){
    return q.head->item;
}
```

```
int peek(Stack s){
    return s.head->item;
}
```

enqueue()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList()`,`size`

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

1. Retrieve: `getFront()`
2. **Insert: `enqueue()`**
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

`int insertNode(LinkedList *ll, int index, int value);`

- Put a new node into the Queue-> insert a node at index *size*

```
void enqueue(Queue *qPtr, int item){
  insertNode(qPtr->head, qPtr->size, item);
}
```

```
void push(Stack *s, int item){
  insertNode2(sPtr, 0, item);
}
```

- To make the `insertNode()` more efficient, *tail is introduced
- `enqueue()` needs to be rewritten to let *tail point to the last node
- Queue is empty (Size=0) is a special case

enqueue()

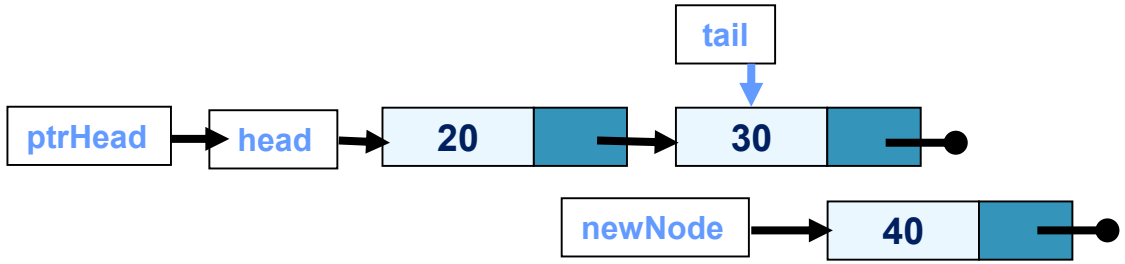
- Put a new node into the Queue-> insert a node at index *size*
- To make the `insertNode()` more efficient, *tail is introduced
- `enqueue()` needs to be rewritten to let *tail point to the last node
- Queue is empty (Size=0) is a special case

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

1. Retrieve: `getFront()`
2. **Insert: enqueue()**
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

```
void enqueue(Queue *qPtr, int item){
  insertNode(qPtr->head, qPtr->size, item);
}
```



Time Complexity	Space Complexity
Θ(n)→ Θ(1)	Θ(n)→ Θ(n)

```
void enqueue(Queue *qPtr, int item){
   QueueNode *newNode;
   newNode = malloc(sizeof(QueueNode));
   newNode->item = item;
   newNode->next = NULL;

   if(isEmptyQueue(*qPtr))
      qPtr->head=newNode;
   else
      qPtr->tail->next = newNode;

   qPtr->tail = newNode;
   qPtr->size++;

}
```

dequeue()

- Remove a new node from the Queue-> remove a node at index 0
- *free() will not let temp ==NULL*
- **tail will point to a free memory which is not NULL*

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

1. Retrieve: `getFront()`
2. **Insert: `enqueue()`**
3. Delete: `dequeue()`
4. Size: `isEmptyQueue()`

```
int dequeue(Queue *qPtr){

    if(qPtr==NULL || qPtr->head==NULL)
            return 0;
    else{
           QueueNode *temp = qPtr->head;
           qPtr->head = qPtr->head->next;
           //Queue is emptied
           if(qPtr->head == NULL)
               qPtr->tail = NULL;

           free(temp);
           qPtr->size--;
           return 1;
        }

}
```

```
int pop(Stack *sPtr){
   if(sPtr==NULL || sPtr->head==NULL)
           return 0;
   else{
           StackNode *temp = sPtr->head;
           sPtr->head = sPtr->head->next;
           free(temp);
           sPtr->size--;
           return 1;
       }
}
```

It is the same as the Stack's pop()

isEmptyQueue()

Linked List

1. Display: `printList()`
2. Search: `findNode()`
3. Insert: `insertNode()`
4. Delete: `removeNode()`
5. Size: `sizeList(), size`

Queue

```
typedef ListNode QueueNode;
typedef struct _queue{
   int size;
   ListNode *head;
   ListNode *tail;
} Queue;
```

1. Retrieve: `getFront()`
2. Insert: `enqueue()`
3. Delete: `dequeue()`
4. **Size: `isEmptyQueue()`**

- Check whether the queue is empty? 1 == empty: 0 == not empty

```
int isEmptyQueue(Queue q){
   if(q.size==0) return 1;
    else return 0;
}
```

```
int isEmptyStack(Stack s){
   if (s.size == 0) return 1;
   return 0;
}
```

Array-based Implementation

- Stacks and queues can be implemented by linked list and array structure.
- Linked list provides more flexibility on its size
- Array allows random access
- But you only can use head or tail in stacks and queues
- Linked list is the better option

Classic Problems

Stack

- Balanced Parentheses Problem (In lab)
- Algebraic expression conversion (infix, prefix and postfix)
- Recursive functions to Iterative functions

Queue

- Palindromes
- Scheduling in multitasking, network, job, mailbox etc.

Algorithm Design Strategies

A general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing

- Brute Force and Exhaustive Search
- Divide-and-Conquer
- Greedy Strategy
- …etc.
- Decrease-and-Conquer
- Transform-and-Conquer
- Iterative Improvement

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f = ?$$

- $+, -, \times, \div$ are known as binary operator
- This expression is an infix expression which the operator is written between its operands.
  - Precedence rules: $\times, \div$ have higher precedence than $+, -$
  - Left-to-right association: Evaluate from left to right
- Without using parentheses, the evaluation is ambiguous
  - $((((a + b) \times c) - d) \times e) \div f$ or
  - $a + (b \times c) - ((d \times e) \div f)$
- Evaluation is tedious by using the infix expression
  - Multiple scanning is required to find the next operation
- How do our calculators work?

The expression is stored as a string "a+b*c"

How does computer interpret the string?

- Precedence rules: $\times$, $\div$ have higher precedence than $+$, $-$
- Left-to-right association: Evaluate from left to right

Learning from simple examples

$$a + b \times c - d \times e \div f$$

How do you know that $b \times c$ is the first operation in the expression?

How about $a + b - c \times d + e$?

Transform-and-Conquer: Algebraic Expressions

$$abc \times +de \times f \div -$$

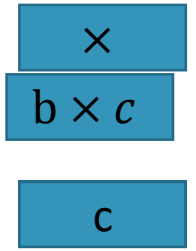
- Use a stack
- When the character is an operand, push it to the stack

- When the character is an operator, '×', pop two operands from the stack
- Evaluate $\mathrm{b} \times c$
- Push the result of $\mathrm{b} \times c$ back to the stack etc.

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$abc \times +de \times f \div -$$



- Use a stack
- When the character is an operand, push it to the stack

- When the character is an operator, '×', pop two operands from the stack
- Evaluate $\mathrm{b} \times c$
- Push the result of $\mathrm{b} \times c$ back to the stack etc.

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$abc \times +de \times f \div -$$

- The expression is known as postfix expression a.k.a reverse Polish notation
- Reduce memory access and improve computational efficiency
- Under this convention, operators appears after its operands

  <operand> <operand> <operator>

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$bc \times$$

- The expression is known as postfix expression a.k.a reverse Polish notation
- Reduce memory access and improve computational efficiency
- Under this convention, operators appears after its operands

  <operand> <operand> <operator>

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$abc \times +$$

- The expression is known as postfix expression a.k.a reverse Polish notation
- Reduce memory access and improve computational efficiency
- Under this convention, operators appears after its operands

  <operand> <operand> <operator>

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$abc \times + \quad de \times f \div$$

- The expression is known as postfix expression a.k.a reverse Polish notation
- Reduce memory access and improve computational efficiency
- Under this convention, operators appears after its operands

  <operand> <operand> <operator>

Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times e \div f$$

$$abc \times + \; de \times f \div -$$

- The expression is known as postfix expression a.k.a reverse Polish notation
- Reduce memory access and improve computational efficiency
- Under this convention, operators appears after its operands

  <operand> <operand> <operator>

Transform-and-Conquer: Algebraic Expressions

Algorithm 1 Infix Expression to Postfix Expression

```
function IN2POST(String infix, String postfix)
    create a Stack S
    for each character c in infix do
        if c is an operand then
            postfix ← c
        else if c = ')' then
            while peek(S) ≠ '(' do
                postfix ← pop(S)
            pop(S)
        else if c = '(' then
            push(c,S)
        else                                   ▷ c is an operator or left parenthesis
            while S ≠ empty && peek(S) != '(' && precedence of peek(S) ≥ precedence of c do
                postfix ← pop(S)
            push(c,S)
    while S is not empty do
        postfix ← pop(S)
```

infix

$$a + b \times c - d \times (e \div f)$$

S

postfix

$$abc \times +def \div \times -$$

Transform-and-Conquer: Algebraic Expressions

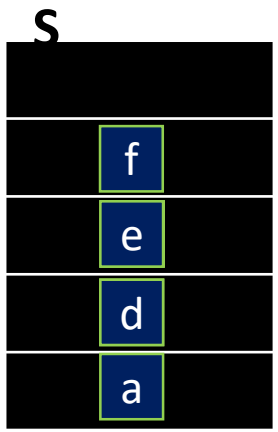
Algorithm 2 Evaluation Postfix Expression

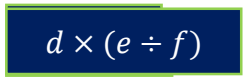
```
function ExePost(String postfix)
    create a Stack S
    for each character c in postfix do
        if c is an operand then
            push(c, S)
        else
            operand1 ← pop(S)
            operand2 ← pop(S)
            result ← Evaluate(operand2, c, operand1)
            push(result, S)
```

$a + b \times c - d \times (e \div f)$

Transform

$abc \times +def \div \times -$





Transform-and-Conquer: Algebraic Expressions

$$a + b \times c - d \times (e \div f)$$

$$- + a \times bc \times d \div ef$$

- The expression is known as prefix expression a.k.a Polish notation
- Under this convention, operators appears before its operands

  <operator> <operand> <operand>

Hint: Its algorithm is similar to the postfix expression's.

Summary

- An algorithm is not simply a computer program
- Stack and Queue are concepts of data structure
  - No new data structure is introduce here
  - We are still using linked lists
- Algorithm Design Strategies
  - Transform-and-Conquer
    - Infix expression to Postfix expression
    - Tree Balancing