SC1007
Data Structures and Algorithms

Introduction

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

yrloke@ntu.edu.sg

N4-02B-69A

Dr. Luu Anh Tuan

anhtuan.luu@ntu.edu.sg

N4-02B-66

Course Schedule

Week	Lecture Topic	Tutorial	Lab	Assignment Deadline
1	Introduction To Data Structure and Algorithm			
2	Linked List (LL) - Linear Search			
3	Analysis of Algorithm	T1	L1 - LL	
4	Stack and Queue (SQ) - Arithmetic Expression			AS1: LL
5	Tree Traversal - Binary Search	T2	L2 - SQ	AS2: SQ
6	AVL, Huffman coding		L3 - Tree 1	AS3: Tree
7	Revision	T3	L4 - Tree 2	AS4: Tree 2
	Recess Week – **Lab Test 1 – 3 March 2022 (Thu)**			
8	Hash Table + Graph Representation			
9	BFS, DFS		L5 - Graph	
10	Backtracking, Permutation	T4	L6 - BFS, DFS	AS5
11	Dynamic Programming	T5	L7 - Backtracking	AS6
12	Bipartite Graph - Matching Problem	T6	L8 - DP	AS7
13	Revision			AS8
14	**Lab2 Test + Quiz – 21 April 2022 (Thu)**			

Learning Outcomes

1. Select appropriate data structures
2. Implement algorithms to solve real world problems using C programming
3. Conduct complexity analysis of algorithms

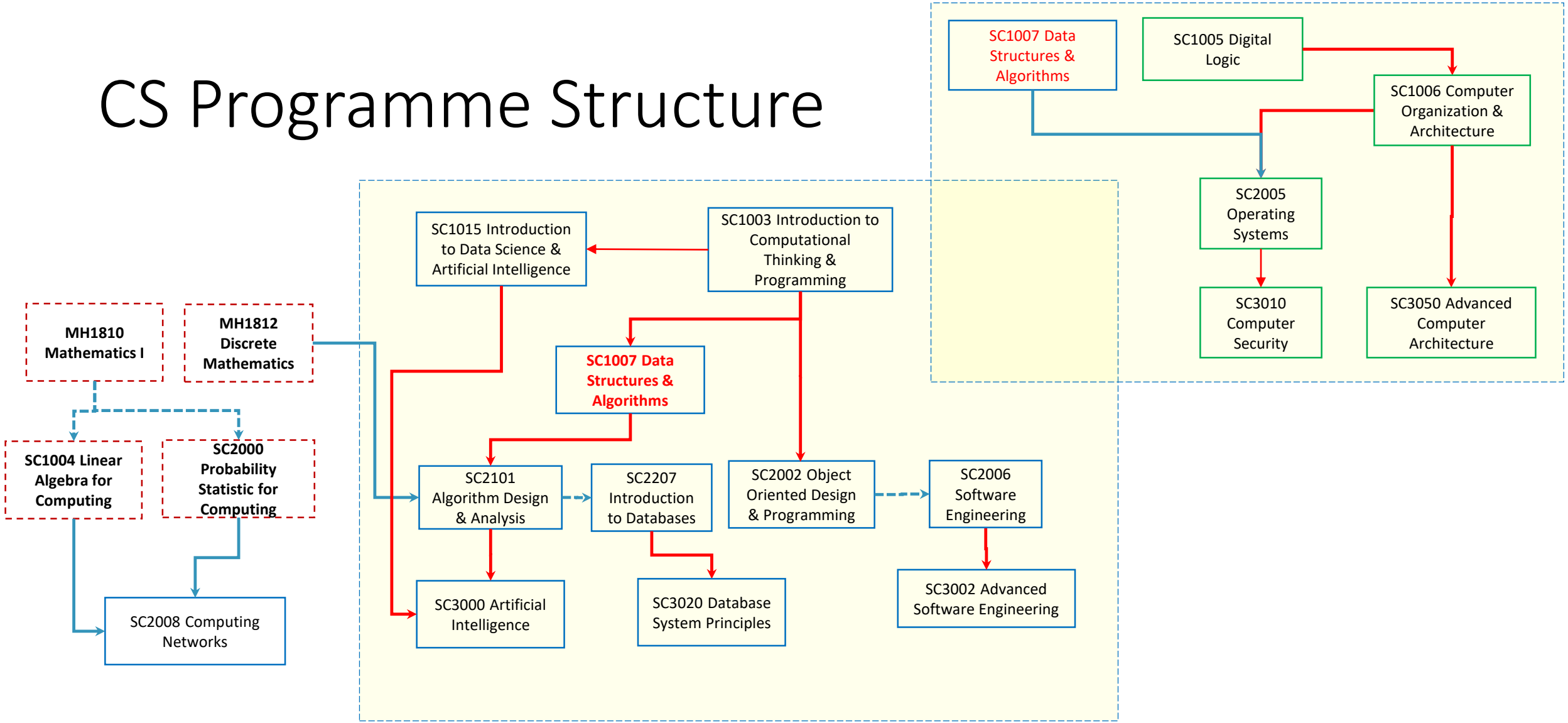
Week	Lecture Topic	Tutorial	Lab	Assignment Deadline
1	Introduction To Data Structure and Algorithm			
2	Linked List (LL) - Linear Search			
3	Analysis of Algorithm	T1	L1 - LL	
4	Stack and Queue (SQ) - Arithmetic Expression			AS1: LL
5	Tree Traversal - Binary Search	T2	L2 - SQ	AS2: SQ
6	AVL, Huffman coding		L3 - Tree 1	AS3: Tree
7	Revision	T3	L4 - Tree 2	AS4: Tree 2
	Recess Week – **Lab Test 1 – 3 March 2022 (Thu)**			
8	Hash Table + Graph Representation			
9	BFS, DFS		L5 - Graph	
10	Backtracking, Permutation	T4	L6 - BFS, DFS	AS5
11	Dynamic Programming	T5	L7 - Backtracking	AS6
12	Bipartite Graph - Matching Problem	T6	L8 - DP	AS7
13	Revision			AS8
14	**Lab2 Test + Quiz – 21 April 2022 (Thu)**			

Assessment Components:

Assessments	Weighting
Assignments	40%
Two Lab Tests	40%
Final Quiz Part 1 and Part 2 concepts	20%

The attendance of tests is compulsory.

CS Programme Structure



Session Objectives

- Lectures focus on introduction to concepts
- Tutorials focus on understanding the concepts, discussion and doubt clarification
- Lab sessions and assignments focus on practice
- Lab tests and quiz are assessments

Overview of SC1007

Data Structures:

- Concepts of pointers and structures (aggregates)
- Introduce some classical data structures
  - Linear: Linked list, stack, queue
  - Non-linear: tree
- Implement these data structures

Algorithms:

- Analysis of Algorithm – time complexity and space complexity
- Introduce to some typical algorithms and their applications
- Introduce to some algorithm design strategies

Implementation:

- C programming

Overview

- What is an algorithm?
- Problem types in computing
- Algorithm design strategies

Algorithm

- Appear in Webster's New World Dictionary after 1957
- It is derived from the name of a Persian Mathematician in the 9th century.
- Euclidean algorithm for finding the greatest common divisor of two numbers – Euclid's Elements (300B.C.)

algorithm

/ˈalgərɪð(ə)m/

noun

noun: **algorithm**; plural noun: **algorithms**

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
"a basic **algorithm for** division"

Origin

late 17th century (denoting the Arabic or decimal notation of numbers): variant (influenced by Greek *arithmos* 'number') of Middle English *algorism*, via Old French from medieval Latin *algorismus* . The Arabic source, *al-Ḵwārizmī* 'the man of Ḵwārizm' (now Khiva), was a name given to the 9th-century mathematician Abū Ja'far Muhammad ibn Mūsa, author of widely translated works on algebra and arithmetic.

Translate algorithm to Choose language

Use over time for: algorithm

Definitions from Oxford Languages Feedback

https://en.wikipedia.org/wiki/Algorithm
Knuth's The Art of Computer Programming

Algorithm

- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Introduction to The Design & Analysis of Algorithms

-Anany Levitin

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Introduction to Algorithms

-T. H. Cormen et. al.

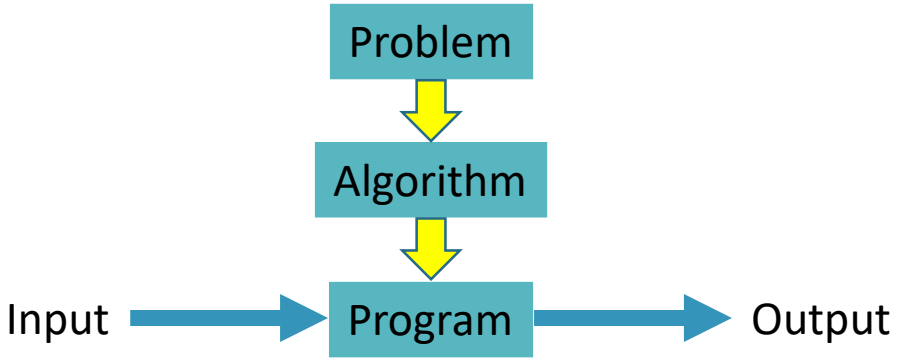
Algorithm

- Correctness:
  - Output results must be correct and consistent for every given input instance
- Precision:
  - A series of well-defined and systematic steps
  - The steps should not contain any ambiguous word like maybe, roughly, about etc.
- Finiteness:
  - Terminates in a finite number of instructions

Algorithm VS Program

- A computer program is an instance, or concrete representation of an algorithm in some programming languages.

- Implementation is the task of turning an algorithm into a computer program.



Example 1: Arithmetic Series

- There are many ways (algorithms) to solve a problem
- Summing up 1 to n

take more memory

Algorithm 1 Summing Arithmetic Sequence

```
1: function Method_One(n)
2: begin
3: sum ← 0
4: for i = 1 to n do
5:     sum ← sum + i
6: end
```

Algorithm 2 Summing Arithmetic

```
1: function Method_Two(n)
2: begin
3: sum ← n ∗ (1 + n)/2
4: end
```

Best way

Algorithm 3 Summing Arithmetic Sequence

```
1: function Method_Three(n)
2: begin
3: if n=1 then
4:     return 1
5: else
6:     return  n+Method_Three(n − 1)
7: end
```

Example 2: Fibonacci Sequence

- 1, 1, 2, 3, 5, 8, …
- The n^{th} term is

best can achieve log n

Which is better algorithm?

$$f(n) = f(n-1) + f(n-2)$$

Algorithm 4 Fibonacci Sequence: A Simple Recursive Function

```
1: function Fibonacci_Recursive(n)
2: begin
3: if n<1 then
4:     return 0
5: if n==1 OR n==2 then
6:     return 1
7: return Fibonacci_Recursive(n − 1)+Fibonacci_Recursive(n − 2)
8: end
```

((1+root(5))/2) ^n =2^n

Is there any better algorithm?

Better

Algorithm 5 Fibonacci Sequence: A Simple Iterative Function

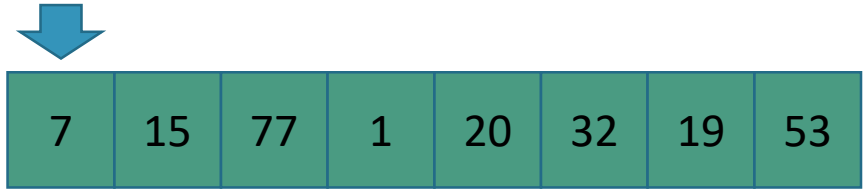
```
1: function Fibonacci_Iterative(n)
2: begin
3: if n<1 then
4:     return 0
5: if n==1 OR n==2 then
6:     return 1
7: F_1 ← 1
8: F_2 ← 1
9: for i = 3 to n do
10:    begin
11:    F_i ← F_{i−2} + F_{i−1}
12:    F_{i−2} ← F_{i−1}
13:    F_{i−1} ← F_i
14:    end
15: return F_n
16: end
```

Problem Types

- Searching
- Graph Problems
- Combinatorial Problems
- Sorting (CZ2101)
- String Processing (CZ2101)
- Geometric Problems
- Numerical Problems

Searching: Find a search key in a given set

20



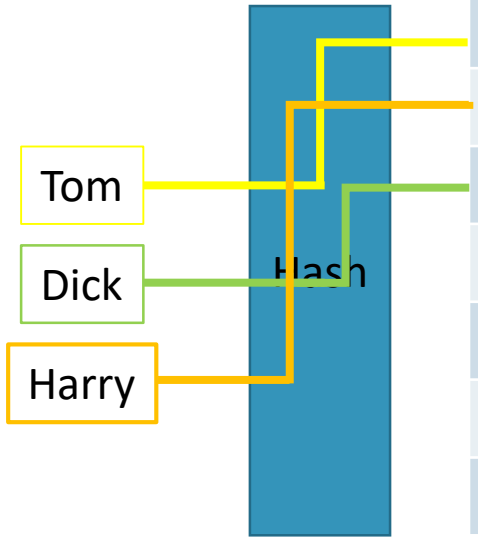
Linear Search/ Sequential Search

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku

Hash Table

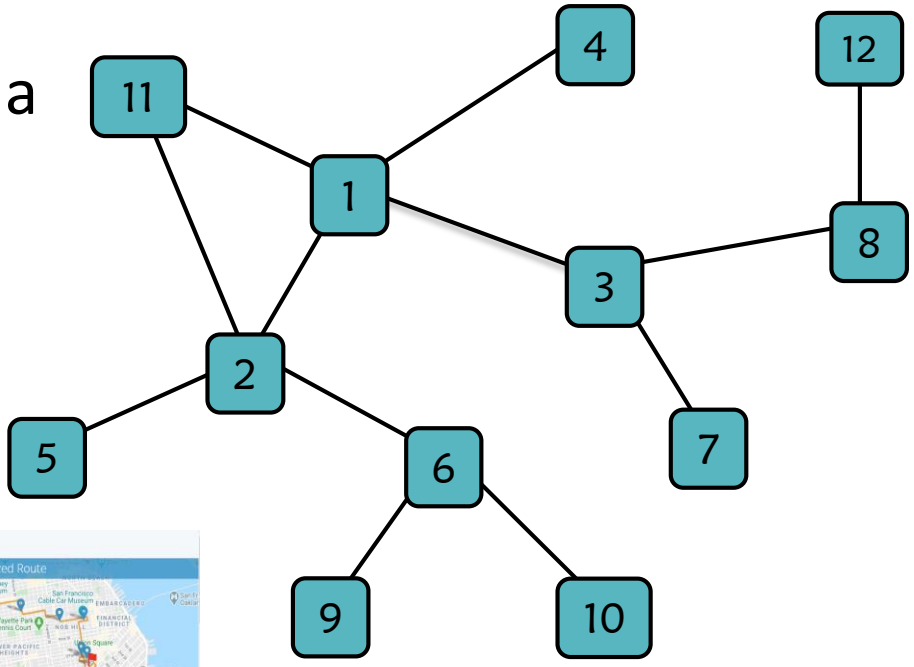
Index	Data
001	Tom, +123456
002	Harry, +369852
003	Dick, +965483
..	…
…	…
..	..
..	..

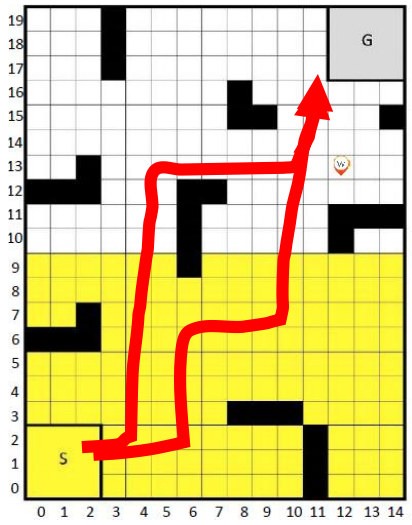


Graph Problems

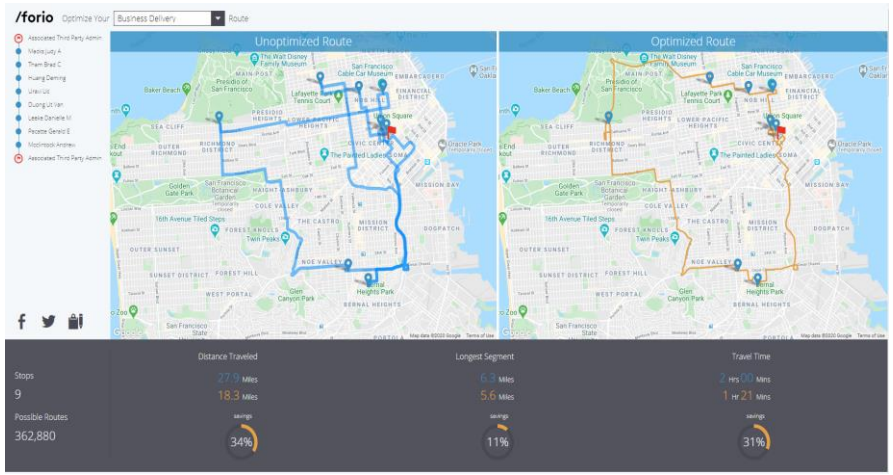
A graph is a mathematical structure consisting of a collection of vertices and edges.

Each edge has one or two vertices associated to it.





Path Finding

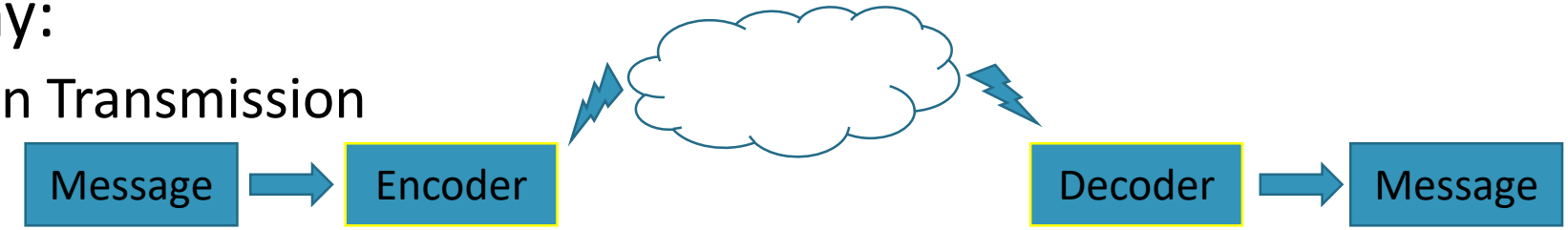


https://forio.com/app/showcase/route-optimizer/

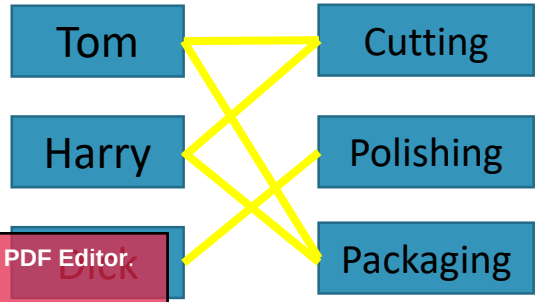
Traveler Salesman Problem

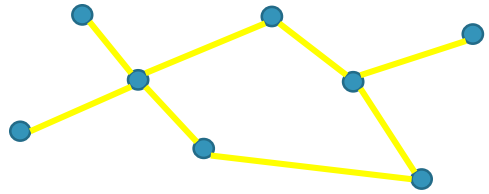
Combinatorial Problems

- The study of arrangements, patterns, designs, assignments schedules, connections and configurations.
- Cryptography:
  - Information Transmission



- Matching and Covering Problem

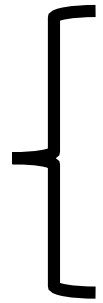




Minimum Vertex Cover Problem

Sorting Problems

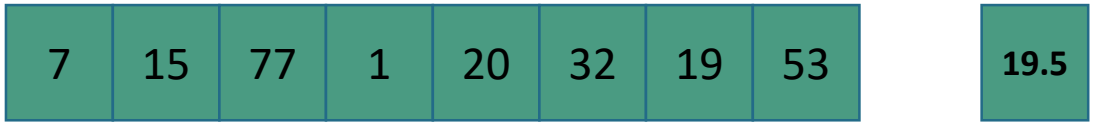
- Rearrange items of a given list in certain order



Numerical Order

Lexicographical Order

- Find the top 5% of students in a class
- Find the median



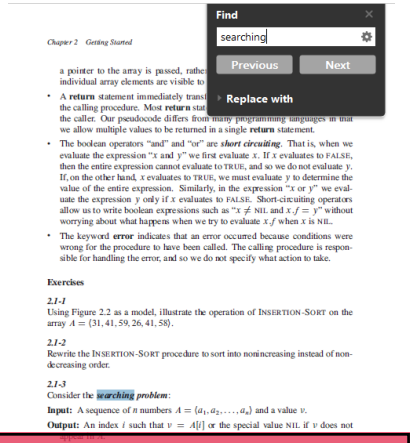
- **Stability:** Stable sorting algorithms sort repeated elements in the same order that they appear in the input.

String Processing

- String matching

PNEUMONOULTRAMICROSCOPICSILICOVOLCANOCONIOSIS

M	I	C	R	O



Text Matching


```
   1 attaaaggtt tataccttcc caggtaacaa accaaccaac tttcgatctc ttgtagatct
  61 gttctctaaa cgaactttaa aatctgtgtg gctgtcactc ggctgcatgc ttagtgcact
 121 cacgcagtat aattaataac taattactgt cgttgacagg acacgagtaa ctcgtctatc
 181 ttctgcaggc tgcttacggt ttcgtccgtg ttgcagccga tcatcagcac atctaggttt
 241 cgtccgggtg tgaccgaaag gtaagatgga gagccttgtc cctggtttca acgagaaaac
 301 acacgtccaa ctcagtttgc ctgttttaca ggttcgcgac gtgctcgtac gtggctttgg
 361 agactccgtg gaggaggtct tatcagaggc acgtcaacat cttaaagatg gcacttgtgg
 421 cttagtagaa gttgaaaaag gcgttttgcc tcaacttgaa cagccctatg tgttcatcaa
 481 acgttcggat gctcgaactg cacctcatgg tcatgttatg gttgagctgg tagcagaact
 541 cgaaggcatt cagtacggtc gtagtggtga gacacttggt gtccttgtcc ctcatgtggg
 601 cgaaatacca gtggcttacc gcaaggttct tcttcgtaag aacggtaata aaggagctgg
 661 tggccatagt tacggcgccg atctaaagtc atttgactta ggcgacgagc ttggcactga
 721 tccttatgaa gattttcaag aaaactggaa cactaaacat agcagtggtg ttacccgtga
 781 actcatgcgt gagcttaacg gaggggcata cactcgctat gtcgataaca acttctgtgg
 841 ccctgatggc taccctcttg agtgcattaa agaccttcta gcacgtgctg gtaaagcttc
 901 atgcactttg tccgaacaac tggactttat tgacactaag aggggtgtat actgctgccg
 961 tgaacatgag catgaaattg cttggtacac ggaacgttct gaaaagagct atgaattgca
1021 gacacctttt gaaattaaat tggcaaagaa atttgacacc ttcaatgggg aatgtccaaa
1081 ttttgtattt cccttaaatt ccataatcaa gactattcaa ccaagggttg aaaagaaaaa
1141 gcttgatggc tttatgggta gaattcgatc tgtctatcca gttgcgtcac caaatgaatg
1201 caaccaaatg tgcctttcaa ctctcatgaa gtgtgatcat tgtggtgaaa cttcatggca
1261 gacgggcgat tttgttaaag ccacttgcga attttgtggc actgagaatt tgactaaaga
1321 aggtgccact acttgtggtt acttacccca aaatgctgtt gttaaaattt attgtccagc
1381 atgtcacaat tcagaagtag gacctgagca tagtcttgcc gaataccata atgaatctgg
1441 cttgaaaacc attcttcgta agggtggtcg cactattgcc tttggaggct gtgtgttctc
1501 ttatgttggt tgccataaca agtgtgccta ttgggttcca cgtgctagcg ctaacatagg
1561 ttgtaaccat acaggtgttg ttggagaagg ttccgaaggt cttaatgaca accttcttga
1621 aatactccaa aaagagaaag tcaacatcaa tattgttggt gactttaaac ttaatgaaga
```

SARS-CoV-2/human/USA/UNC_200265_2020/2020, complete genome

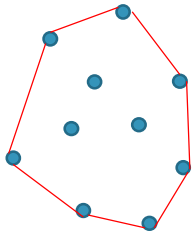
```
   1 aacaaaccaa ccaactttcg atctcttgta gatctgttct ctaaacgaac tttaaaatct
  61 gtgtggctgt cactcggctg catgcttagt gcactcacgc agtataatta ataactaatt
 121 actgtcgttg acaggacacg agtaactcgt ctatcttctg caggctgctt acggtttcgt
 181 ccgtgttgca gccgatcatc agcacatcta ggttttgtcc gggtgtgacc gaaaggtaag
 241 atggagagcc ttgtccctgg tttcaacgag aaaacacacg tccaactcag tttgcctgtt
 301 ttacaggttc gcgacgtgct cgtacgtggc tttggagact ccgtggagga ggtcttatca
 361 gaggcacgtc aacatcttaa agatggcact tgtggcttag tagaagttga aaaaggcgtt
 421 ttgcctcaac ttgaacagcc ctatgtgttc atcaaacgtt cggatgctcg aactgcacct
 481 catggtcatg ttatggttga gctggtagca gaactcgaag gcattcagta cggtcgtagt
 541 ggtgagacac ttggtgtcct tgtccctcat gtgggcgaaa taccagtggc ttaccgcaag
 601 gttcttcttc gtaagaacgg taataaagga gctggtggcc atagttacgg cgccgatcta
 661 aagtcatttg acttaggcga cgagcttggc actgatcctt atgaagattt tcaagaaaac
 721 tggaacacta aacatagcag tggtgttacc cgtgaactca tgcgtgagct taacggaggg
 781 gcatacactc gctatgtcga taacaacttc tgtggccctg atggctaccc tcttgagtgc
 841 attaaagacc ttctagcacg tgctggtaaa gcttcatgca ctttgtccga acaactggac
 901 tttattgaca ctaagagggg tgtatactgc tgccgtgaac atgagcatga aattgcttgg
 961 tacacggaac gttctgaaaa gagctatgaa ttgcagacac cttttgaaat taaattggca
1021 aagaaatttg acatcttcaa tggggaatgt ccaaattttg tatttccctt aaattccata
1081 atcaagacta ttcaaccaag ggttgaaaag aaaaagcttg atggctttat gggtagaatt
1141 cgatctgtct atccagttgc gtcaccaaat gaatgcaacc aaatgtgcct ttcaactctc
1201 atgaagtgtg atcattgtgg tgaaacttca tggcagacgg gcgattttgt taaagccact
1261 tgcgaatttt gtggcactga gaatttgact aaagaaggtg ccactacttg tggttactta
1321 ccccaaaatg ctgttgttaa aatttattgt ccagcatgtc acaattcaga agtaggacct
1381 gagcatagtc ttgccgaata ccataatgaa tctggcttga aaaccattct tcgtaagggt
1441 ggtcgcacta ttgcctttgg aggctgtgtg ttctcttatg ttggttgcca taacaagtgt
1501 gcctattggg ttccacgtgc tagcgctaac ataggttgta accatacagg tgttgttgga
1561 gaaggttccg aaggtcttaa tgacaacctt cttgaaatac tccaaaaaga gaaagtcaac
1621 atcaatattg ttggtgactt taaacttaat gaagagatcg ccattatttt ggcatctttt
```

Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome.

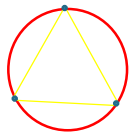
Computational Geometric Problem

- Convex hull problem: Given a set of points, find the smallest convex polyhedron/polygon containing all the points

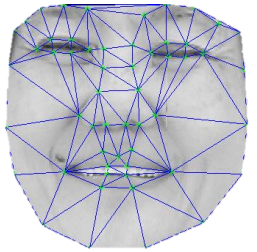
- Delaunay triangulation: for a given set P of discrete points in a plane is a triangulation such that no point in P is inside the circumcircle of any triangle



Convex Hull



circumcircle



Delaunay Triangulation

Numerical Problem and Optimization Problem

- Use numerical approximation for the mathematical analysis
- Widely used for solving problems of engineering and mathematical models
  - Newton's method
  - Gaussian elimination
- Linear programming is an optimization technique for a system of linear constraints and a linear objective function

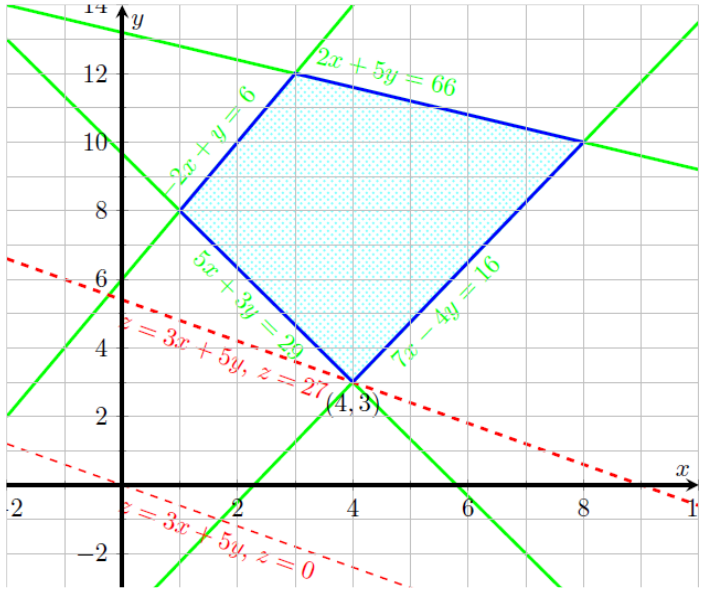
$$\min 3x + 5y$$

$$\text{subject to } 5x + 3y \geq 29$$

$$-2x + y \leq 6$$

$$2x + 5y \leq 66$$

$$7x - 4y \leq 16$$



How do we solve these problems?

How do we solve these problems?

- Select appropriate data structures
  - Arrays
  - Linked Lists
    - Singly linked list, doubly linked list, circular linked list etc.
  - Stack and Queue
  - Trees
  - Table
  - Graphs

- Recursive and non-recursive concepts and their implementation

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

Summary

- An algorithm is not simply a computer program
- Computing Problems
  - Searching
  - Graph Problems etc.
- Algorithm Design Strategies
  - Brute-force
  - Divide-and-Conquer
  - Decrease-and-Conquer
  - Transform-and-Conquer
    - Infix expression to Postfix expression

- Lectures focus on introduction to concepts
- Tutorials focus on understanding the concepts, discussion and doubt clarification
- Lab Sessions and assignments focus on practice and realization
- Lab Tests and quiz are assessments

Next Lecture

- Definition of Pointers and Structures
- Static Data Structure and Dynamic Data Structure
- Computer Memory Layouts
- Memory Allocation
- Memory Deallocation
- Examples and Common Mistakes

- Concepts of Linked Lists

SC1007
Data Structures and Algorithms

Dynamic Data Structure

&

Linked Lists

Dr. Loke Yuan Ren

yrloke@ntu.edu.sg

N4-02B-69A

College of Engineering

School of Computer Science and Engineering

Overview

- Definition of Pointers and Structures
- Static Data Structure and Dynamic Data Structure
- Computer Memory Layouts
- Memory Allocation
- Memory Deallocation
- Examples and Common Mistakes

- Concepts of Linked Lists

C Programming - Quick Recap

Basic C Programming:

1. **Read Input: scanf()**
2. **Write Output: printf()**
3. **Arithmetic: +,-,*,/,%**
4. **Logic: &&, ||, !, ==, !=, >,<,>=,<= etc.**
5. **Control Structure:**
    1. **Sequence Structure**
    2. **Selection Structure: if… else…, switch and break, goto, ? :**
    3. **Repetition Structure: while, do… while, for loop**

1. **Function**
2. **Pointer – * and &**
3. **Array**
4. **Character String**
5. **Structure**
6. **Recursion**

Static Data Structure and Dynamic Data Structure

Static Data Structure: the allocated memory size is fixed at compile time. You are not able to change the size while you are running it.

Built-in Data Types

Scalar variables	char, int, long, float, double
Array	char name[64]; int a[8][8];

Derived Data Type

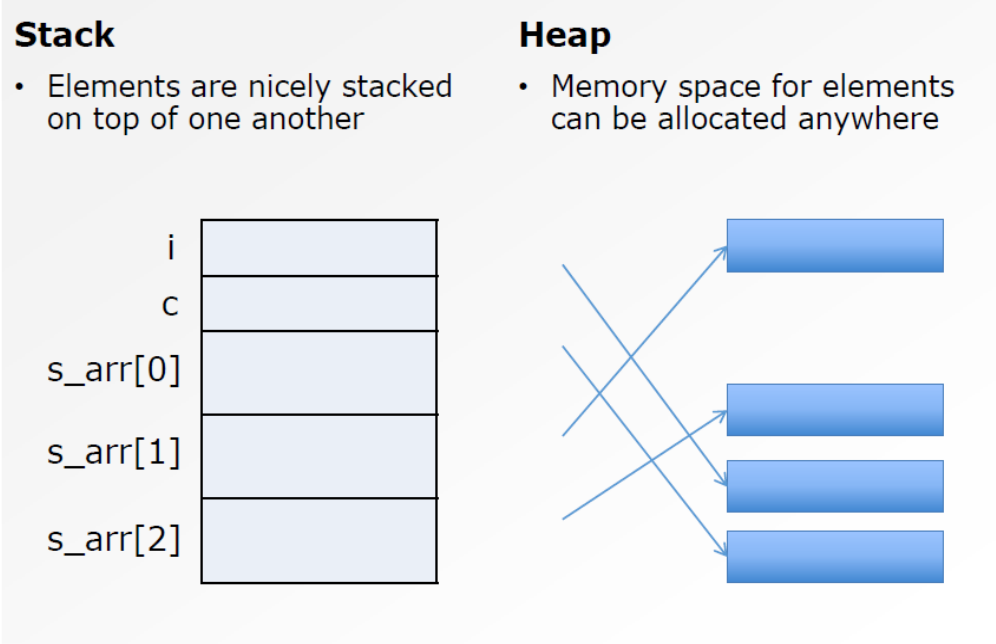
C Structs	**typedef struct node {** **int age;** **float height;** **}student_t;** **student_t s1;** **struct node s2;** struct node { int age; float height; }student;

Dynamic Data Structure: the memory allocation is done during execution time. You have some ways to change the size during the program is running.

Static Data Structure and Dynamic Data Structure

Static Data Structure: the allocated memory size is fixed at compile time. You are not able to change the size while you are running it.

Dynamic Data Structure: the memory allocation is done during execution time. You have some ways to change the size during the program is running.



Memory Allocation

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

Dynamic Memory Allocation and Deallocation

```
#include <stdlib.h>
malloc() and free()
```

- `malloc()` takes an argument of the number of bytes to be allocated and returns a **`void`** pointer `to the allocated memory.`
- `sizeof()` can be used to determine the size of a structure in bytes

```
Eg. int *array = (int *) malloc(sizeof(int)*5);
```

- `free() deallocates memory but freed pointer is not NULL.`

Extra Information

- malloc() does not “clear” the data in the memory.
- If you would like to initialize the elements as zero,
  1. Write a loop to initialize them to zero
  2. Use calloc()

  Eg. int *array = (int *) calloc(5,sizeof(int));
- free() is still required to deallocate memory
- realloc() allows user change the size of the allocated memory.

```c
#include <stdio.h>
#include <stdlib.h>                                     //include library for malloc() and free()

int main(void)
{
      int i;
      double* item;                                     //declare two pointers
      char* string;
      item = (double *) malloc(10*sizeof(double));      //dynamically memory allocation for 10 elements each
      string = (char *) malloc(10*sizeof(char));

      for(i=0;i<10;i++)                                 //Read 10 floating numbers
      {
            scanf("%lf",&item[i]);
      }
      scanf("%*c");                                     //skip the last '\n'

      i=0;                                              //Read 9 character + null character to stop
      char *stringP=string;
      while(i++<9)
            scanf("%c",stringP++);
      *stringP='\0';
      printf("%s\n",string);                            //Print the string

      double* itemP=item;                               //Print the numbers in item
      for(i=0;i<10;i++,itemP++)
            printf("%.2lf ",*itemP);
      printf("\n");

      free(item);                                       //free the allocated memory
      free(string);
      return 0;
}
```

Memory Allocation

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

Memory Allocation

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

Linked List

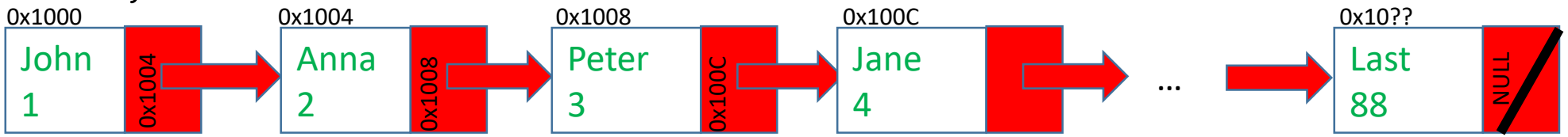
Memory Address	Name	Matric No
0x1000	John	0001
0x1004	Anna	0002
0x1008	Peter	0003
0x100C	Jane	0004
	…	…

- **Structure**: a collection of variables with different types:

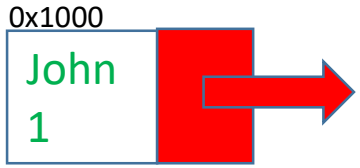
```
struct student{
        char Name[15];
        int matricNo;
}
```

- **Self-referential structure:** a pointer member that points to a structure of the same structure type

```
struct node{
        char Name[15];
        int matricNo;
        struct node *nextPtr;   //link
}
```

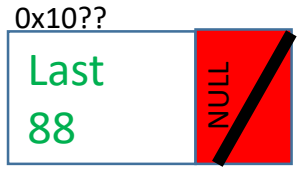


Linked List




```
struct node{
          char Name[15];
          int matricNo;
          struct node *nextPtr;  //link
}
```

1. **Each node contains data and link**
2. **The link contains the address of next node**
3. **If user knows the address of first node, the next node can be found from the link.**
4. **The link of the last node is a NULL pointer**



5. **The example is known as singly-linked list**
   - **There is only ONE link in the node**

Summary

- Static Structure Definition

```
typedef struct node {
        int age;
        float height;
   }student_t;
student_t s1;
struct node s2;
```

- Dynamic Memory Allocation/Deallocation

```
#include <stdlib.h>
item = (double *) malloc(10*sizeof(double));
free(item);
```

- Concepts of Linked Lists

```
struct node{
        char Name[15];
        int matricNo;
        struct node *nextPtr;  //link
    }
```

Overview of Next Lecture

1. **What is the linked list?**
2. **How to create a linked list?**
3. **How to use the linked list?**
4. **Why do you need a linked list?**

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

SC1007
Data Structures and Algorithms

The Linked List and Its Implementation

Dr. Loke Yuan Ren

yrloke@ntu.edu.sg

N4-02B-69A

College of Engineering

School of Computer Science and Engineering

Overview of Today Lecture

1. **What is the linked list?**
2. **How to create a linked list?**
3. **How to use the linked list?**
4. **Why do you need a linked list?**

What is the linked list?

Memory Allocation

3 scenario

1. **Known the data size before compile**
2. **Known the data size at the beginning**
3. **Unknown the data size. The size can be increased or decreased over the time while the program is running**

1. **Static Data Allocation (in stack memory)**
2. **Dynamic Data Allocation (in heap memory)**
3. **Dynamic Data with linked list structure**

Linked List

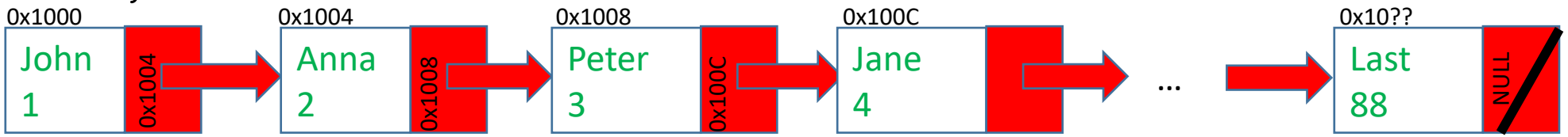
Memory Address	Name	Matric No
0x1000	John	0001
0x1004	Anna	0002
0x1008	Peter	0003
0x100C	Jane	0004
	…	…

- **Structure**: a collection of variables with different types:

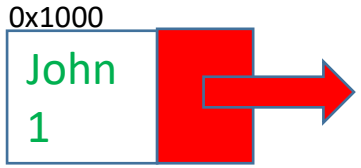
```
struct student{
        char Name[15];
        int matricNo;
}
```

- **Self-referential structure:** a pointer member that points to a structure of the same structure type

```
struct node{
        char Name[15];
        int matricNo;
        struct node *nextPtr;   //link
}
```

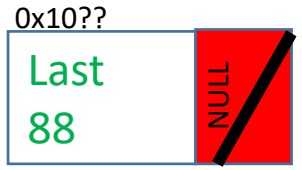


Linked List




```
struct node{
          char Name[15];
          int matricNo;
          struct node *nextPtr;  //link
}
```

1. **Each node contains data and link**
2. **The link contains the address of next node**
3. **If user knows the address of first node, the next node can be found from the link.**
4. **The link of the last node is a NULL pointer**



5. **The example is known as singly-linked list**
   - **There is only ONE link in the node**

How to create a linked list?

Definition and Declaration

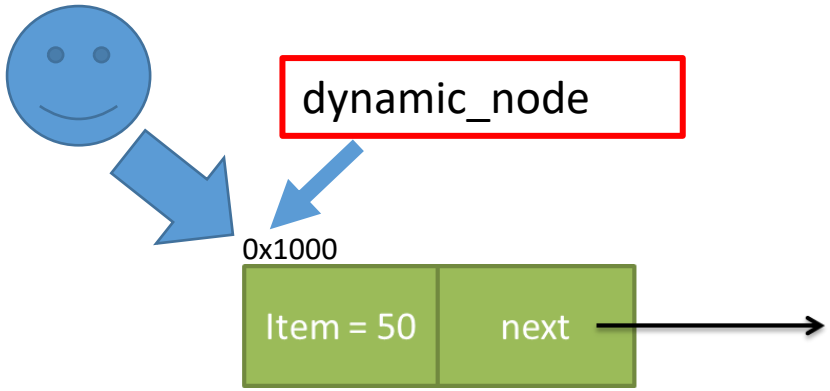
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct _listnode
5  {
6      int item;
7      struct _listnode *next;
8  };
9  typedef struct _listnode ListNode;
10
11 int main(void)
12 {
13 //static node
14     ListNode static_node;
15     static_node.data = 50;
16     static_node.next = NULL;
17
18 //dynamic node
19     ListNode* dynamic_node= (ListNode*) malloc(sizeof(ListNode));
20     dynamic_node->data = 50;
21     dynamic_node->next = NULL;
22     free(dynamic_node);
23
24     return 0;
25 }
```

- Define a self-referential structure, ListNode
- Dynamically allocate a ListNode node
- Free the node
- malloc() does not allocate NULL to the next link
- free() does memory deallocation but not delete
- After dynamic_node is freed, dynamic_node is **NOT** NULL

DEFINE AND CREATE A LINKED LIST

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  struct _listnode
 5  {
 6       int item;
 7       struct _listnode *next;
 8  };
 9  typedef struct _listnode ListNode;
10
11  int main(void)
12  {
13  //static node
14       ListNode static_node;
15       static_node.data = 50;
16       static_node.next = NULL;
17
18  //dynamic node
19       ListNode* dynamic_node=malloc(sizeof(ListNode));
20       dynamic_node->data = 50;
21       dynamic_node->next = NULL;
22
23       ListNode* head = dynamic_node;
24       free(dynamic_node);
25
26       return 0;
27  }
```

- Create a head
  - ListNode* head;
- Multiple ListNode pointers can be created but the node just need to free once in the end



What is head after line 24?

Is There any bug?

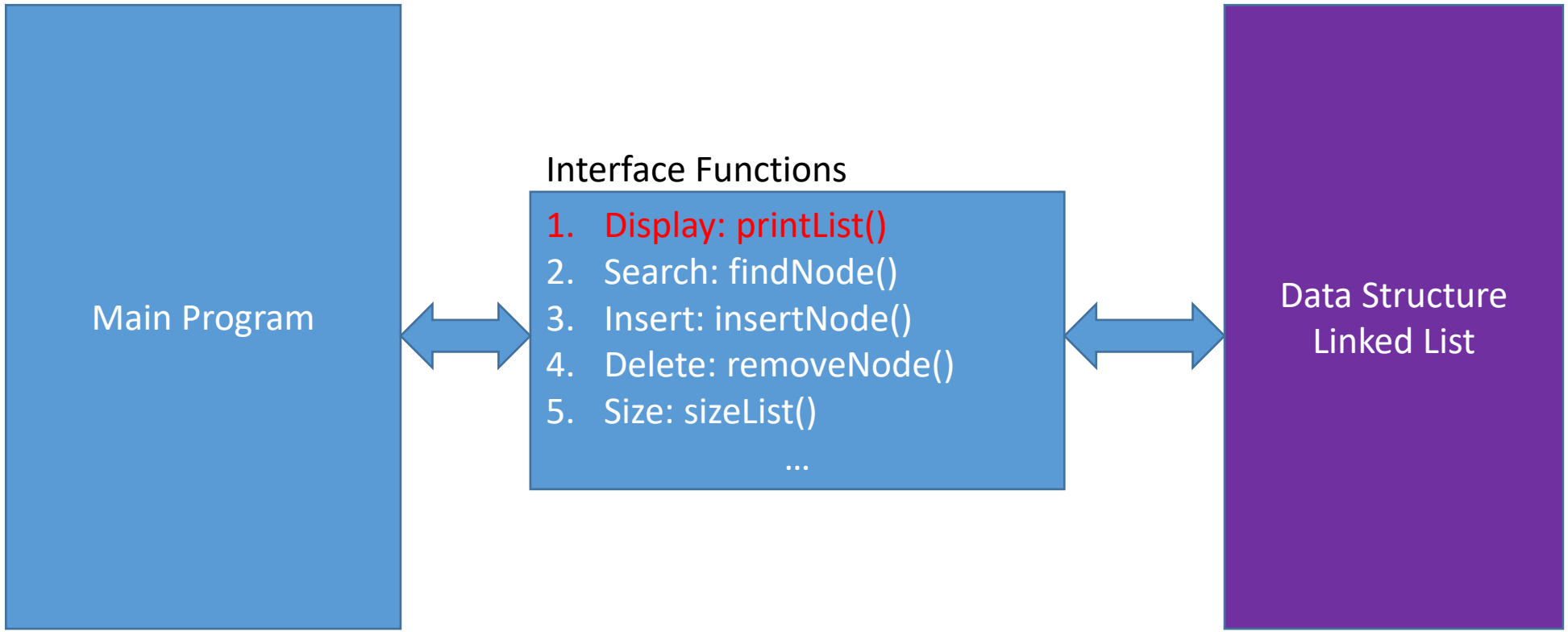
```
1  typedef struct node{
2       int item; struct node *next;
3  } ListNode;
4
5  int main(){
6       ListNode *head = NULL, *temp;
7       int i = 0;
8
9       while (scanf("%d", &i)){
10           if (head == NULL){
11                head = malloc(sizeof(ListNode));
12                temp = head;
13           }
14           else{
15                temp->next = malloc(sizeof(ListNode));
16                temp = temp->next;
17           }
18           temp->item = i;
19      }
20      temp->next = NULL;
21      return 0;
22 }
```

A. Yes

B. No

How to use the linked list?

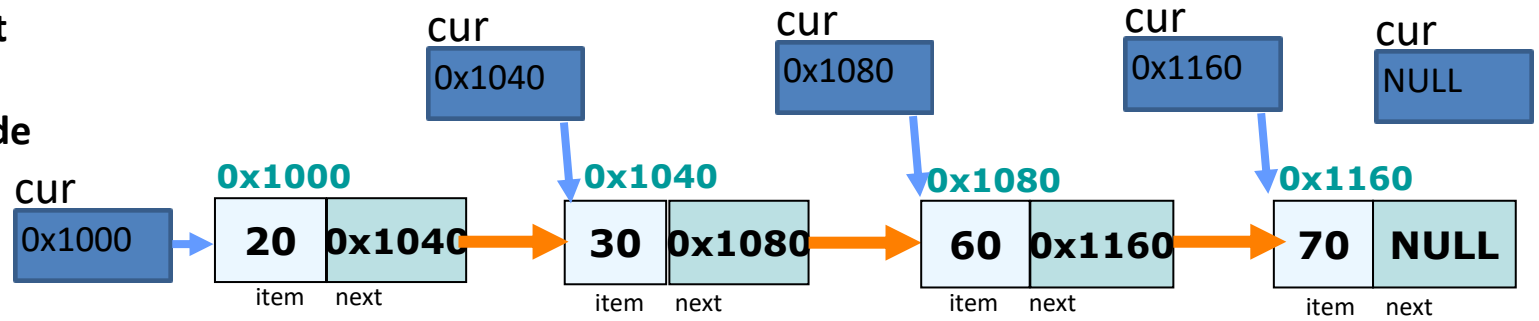
HOW TO USE THE LINKED LISTS?



DISPLAY: printList()

```
void printList(ListNode *cur);
```

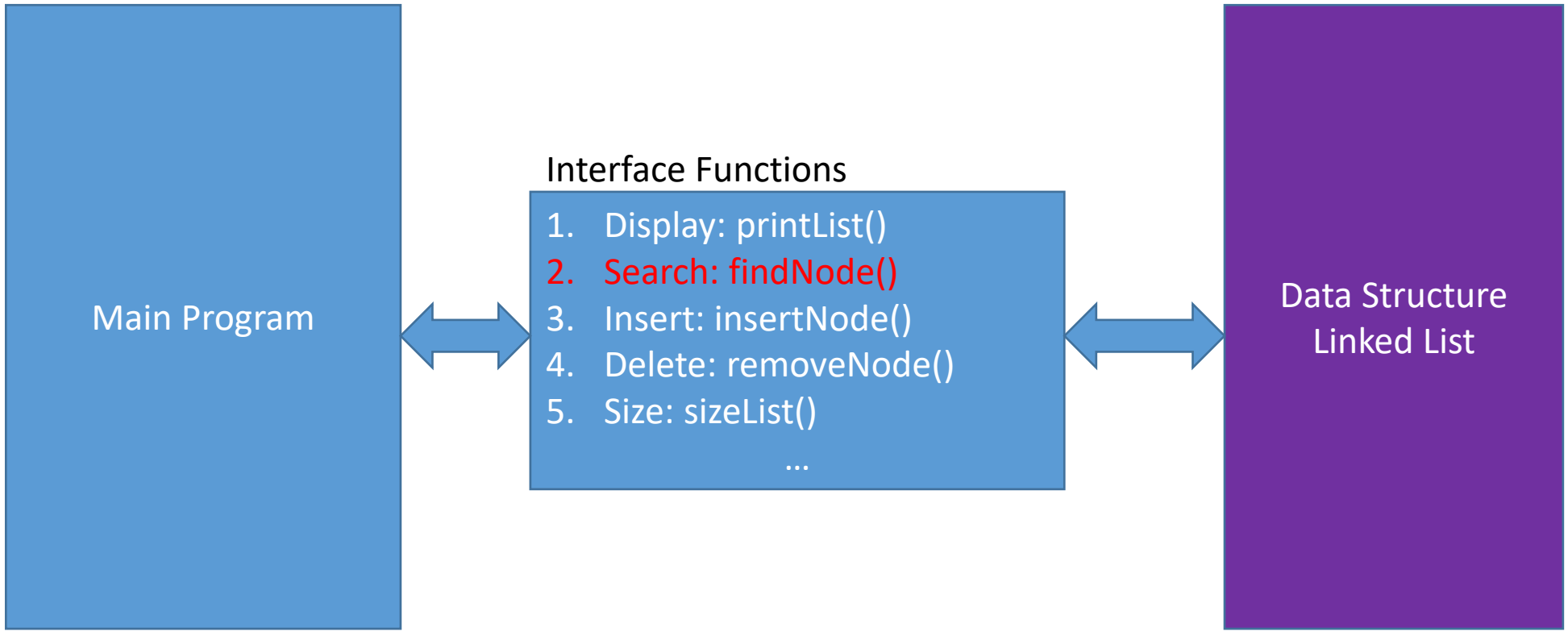
1. **Given the head pointer of the linked list**
2. **Print all items in the linked list**
3. **From first node to the last node**




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

```
20
30
60
70
```

HOW TO USE THE LINKED LISTS?



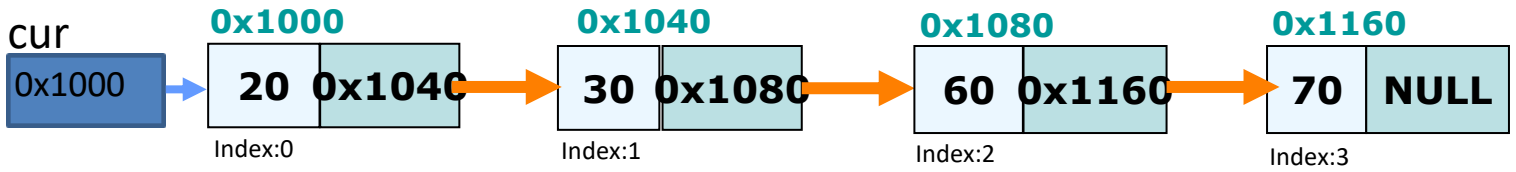
SEARCH: findNode()

```
ListNode* findNode(ListNode *cur, int i);
```

Looking for the *i*th node in the list

1. **Given the head pointer of the linked list and index *i***
2. **Return the pointer to the *i*th node**
3. **NULL will be return if index *i* is out of the range or the linked list is empty**

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



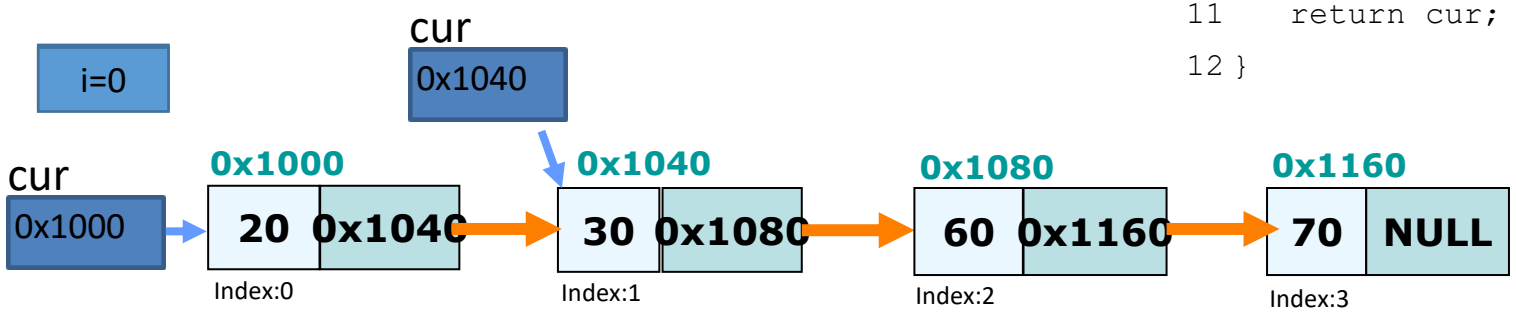
SEARCH: findNode()

```
ListNode* findNode(ListNode *cur, int i);
```

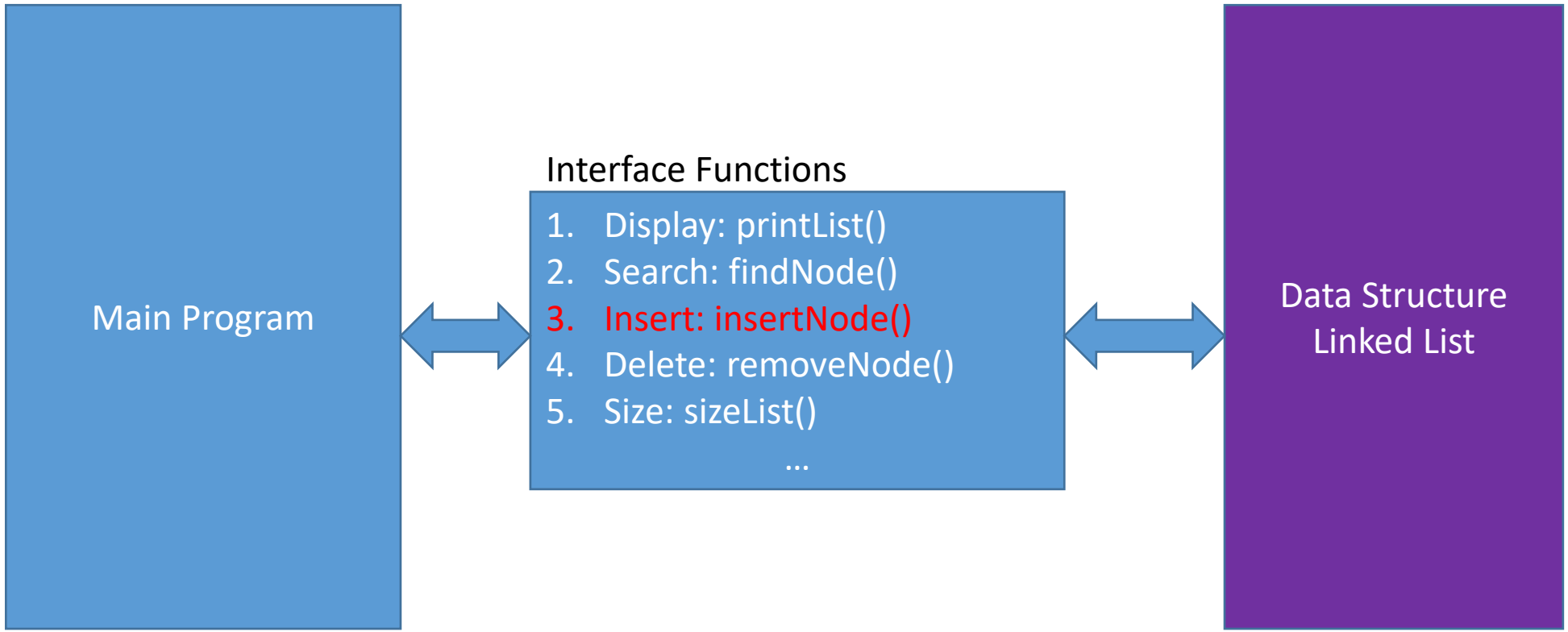
Looking for the *1st* node in the list

1. **Given the head pointer of the linked list and index *i=1***

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



HOW TO USE THE LINKED LISTS?



INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

Add a node in the linked list

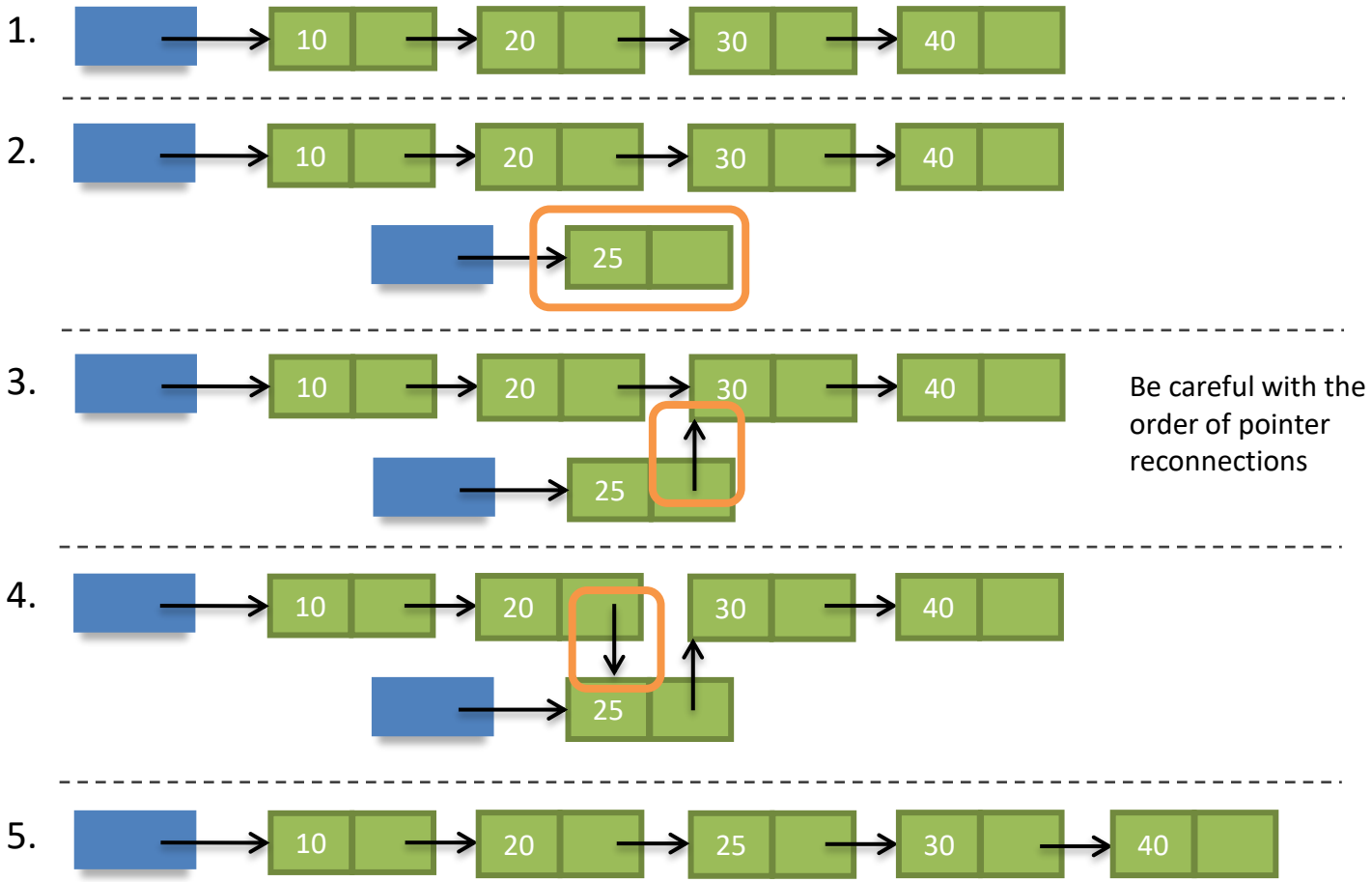
Given

- **the head pointer of the linked list**
- **index *i* where the node to be inserted**
- **the item for the node**

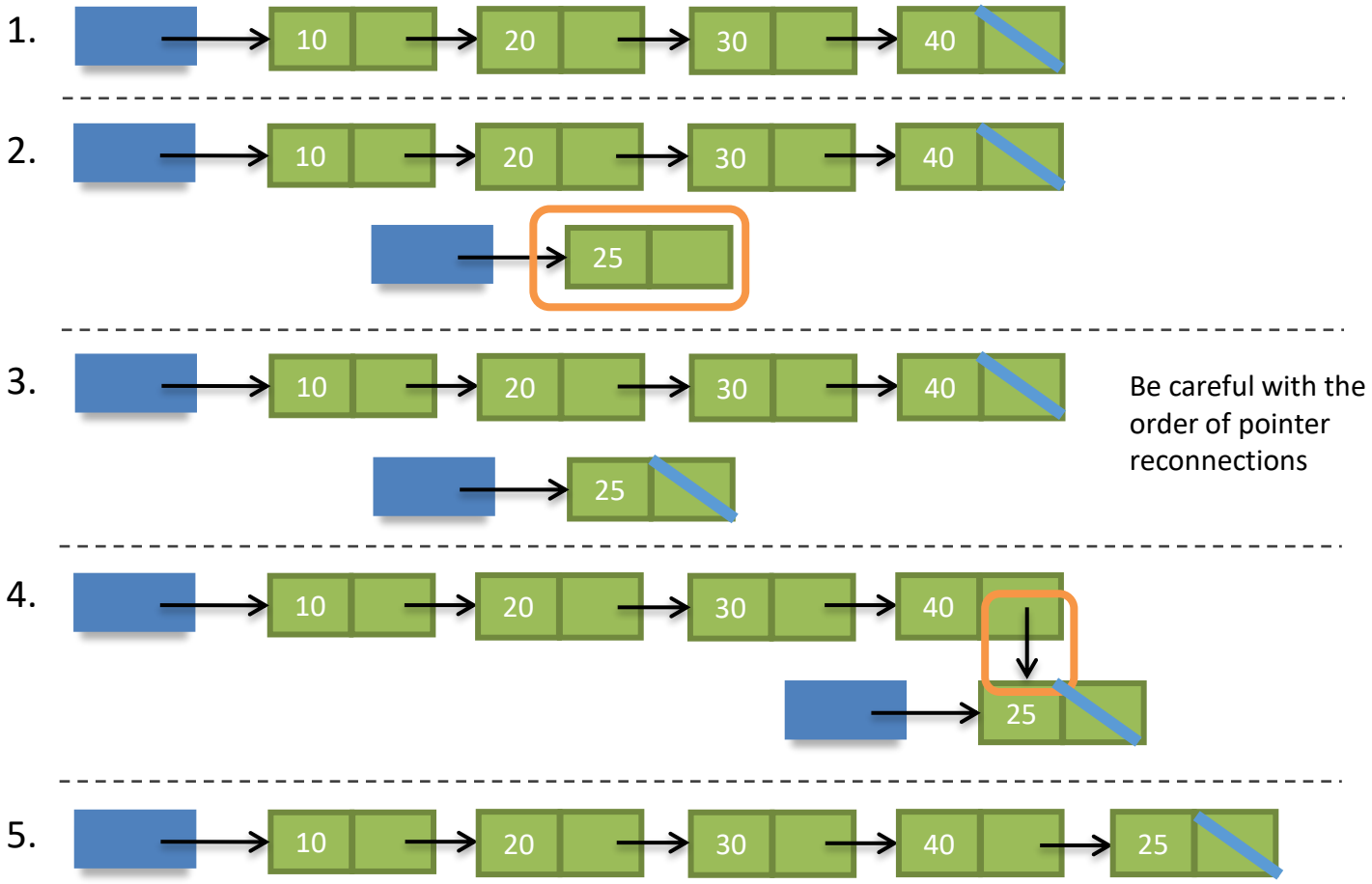
Return SUCCESS (1) or FAILURE (0)

1. **Create a node by the given item**
2. **Insert the node at**
    1. **Front**
    2. **Middle**
    3. **Back**

INSERT A NODE IN MIDDLE



INSERT A NODE IN BACK



INSERT A NODE FRONT

- What is common to both special cases?
  - Empty list

```
head = malloc(sizeof(ListNode))
```

  - Inserting a node at index 0


```
// Save address of the first node
head = malloc(sizeof(ListNode))
head->next = [address of first node]
```

INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

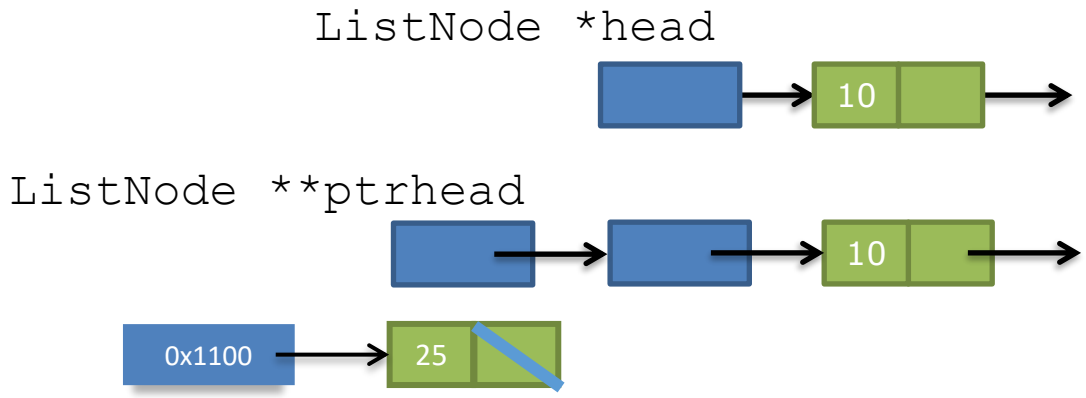
Add a node in the linked list

Given

- **the head pointer of the linked list**
- **index *i* where the node to be inserted**
- **the item for the node**

Return SUCCESS or FAILURE

1. **Create a node by the given item**
2. **Insert the node at**
    1. **Front**
    2. **Middle**
    3. **Back**



Memory Address	Data
0x1000	head=&ListNode 0x1080
0x1060	ptrhead=&head 0x1000
0x1080	item=10 next=0x10c0
0x1100	Item=25 next=NULL

INSERT: insertNode()

```
int insertNode(ListNode **ptrHead, int i, int item);
```

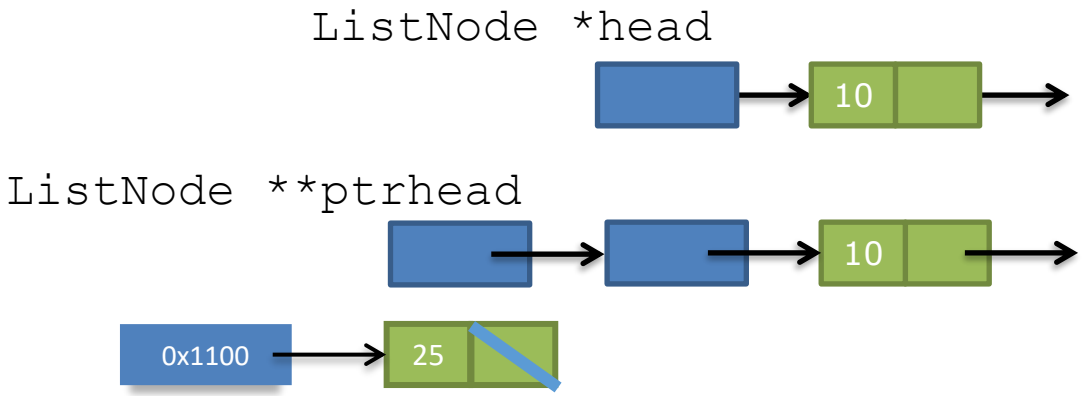
If we only pass head (0x1080) to insertNode(),

we only can access item=10 and next=0x10c0

we cannot modify the content in 0x1000.

When we are back from insertNode() to main(),

0x1000 still remain as 0x1080

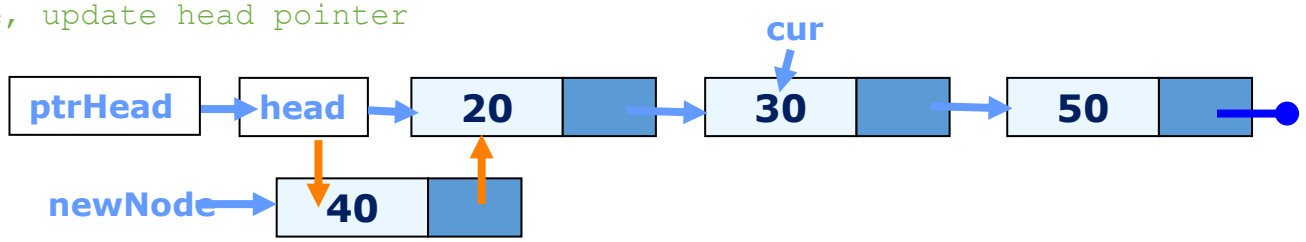


Memory Address	Data
0x1000	head=&ListNode 0x1080
0x1060	ptrhead=&head 0x1000
0x1080	item=10 next=0x10c0
0x1100	item=25 next=NULL

insertNode()

Is there any bug?

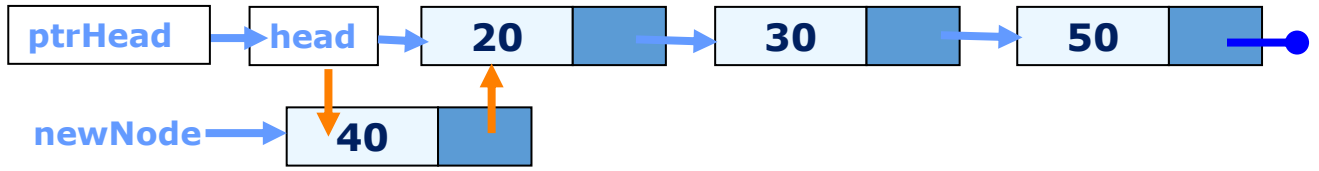
```
1 int insertNode(ListNode **ptrHead, int i, int item){
2     ListNode  *cur, *newNode;
3     // If empty list or inserting first node, update head pointer
4     if (*ptrHead == NULL || i == 0){
5         newNode = malloc(sizeof(ListNode));
6         newNode->item = item;
7         newNode->next = *ptrHead;
8         *ptrHead = newNode;
9         return 1;
10    }
11    // Find the nodes before and at the target position
12    // Create a new node and reconnect the links
13    else if ((cur = findNode(*ptrHead, i-1)) != NULL){
14        newNode = malloc(sizeof(ListNode));
15        newNode->item = item;
16        newNode->next = cur->next;
17        cur->next = newNode;
18        return 1;
19    }
20    return 0;
  }
```



insertNode()

i=0 **item=40**

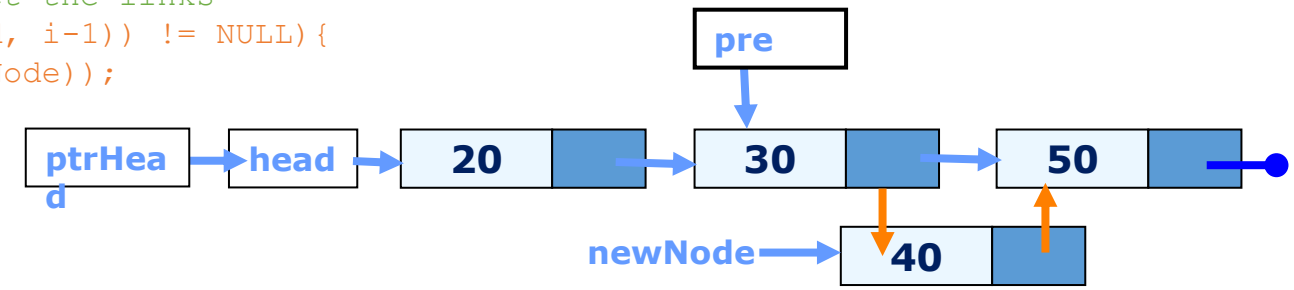
```
1 int insertNode(ListNode **ptrHead, int i, int item){
2     ListNode  *pre, *newNode;
3     // If empty list or inserting first node, update head pointer
4     if (i == 0){
5         newNode = malloc(sizeof(ListNode));
6         newNode->item = item;
7         newNode->next = *ptrHead;
8         *ptrHead = newNode;
9         return 1;
10    }
11    // Find the nodes before and at the target position
12    // Create a new node and reconnect the links
13    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
14        newNode = malloc(sizeof(ListNode));
15        newNode->item = item;
16        newNode->next = pre->next;
17        pre->next = newNode;
18        return 1;
19    }
20    return 0;
  }
```



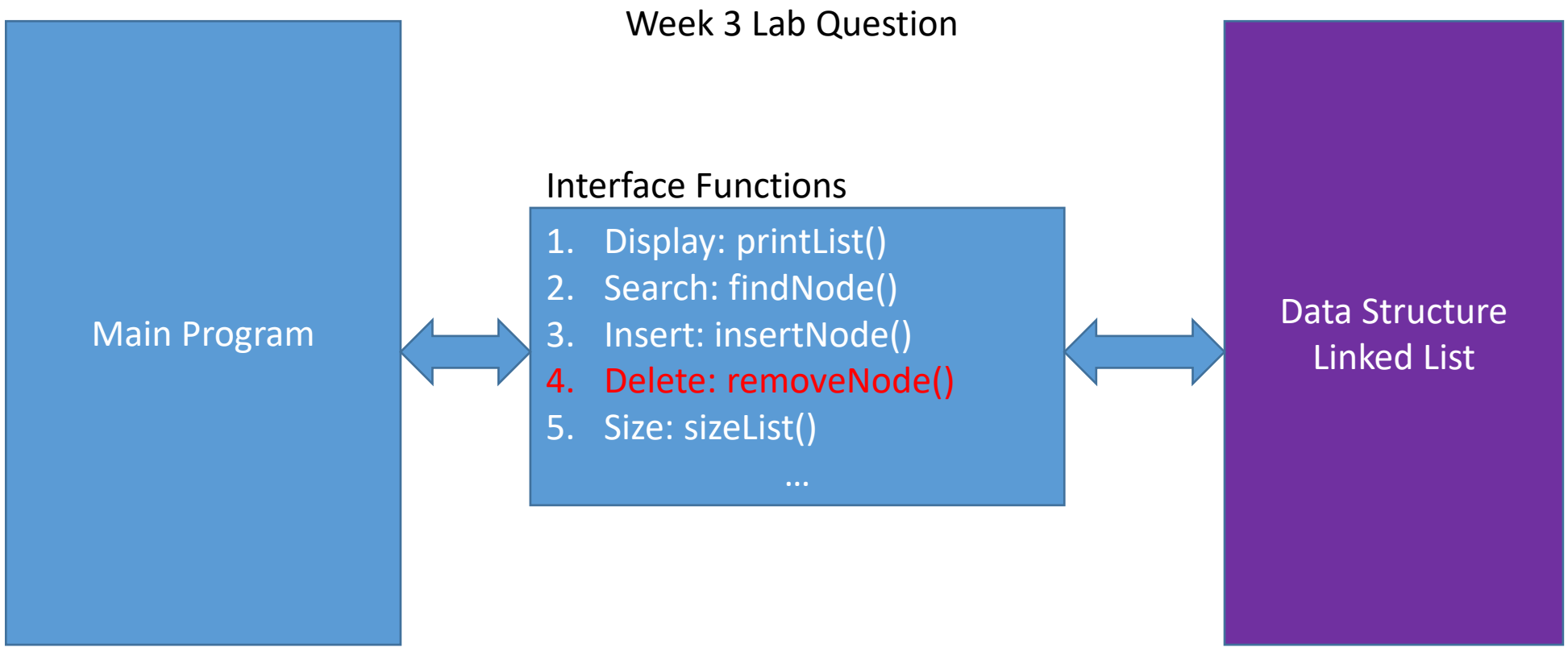
insertNode()

i=2 item=40

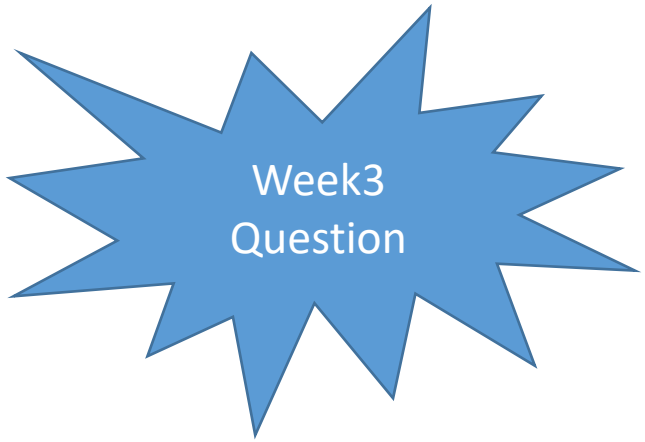
```
 1 int insertNode(ListNode **ptrHead, int i, int item){
 2     ListNode  *pre, *newNode;
 3     // If empty list or inserting first node, update head pointer
 4     if (i == 0){
 5         newNode = malloc(sizeof(ListNode));
 6         newNode->item = item;
 7         newNode->next = *ptrHead;
 8         *ptrHead = newNode;
 9         return 1;
10     }
11     // Find the nodes before and at the target position
12     // Create a new node and reconnect the links
13     else if ((pre = findNode(*ptrHead, i-1)) != NULL){
14         newNode = malloc(sizeof(ListNode));
15         newNode->item = item;
16         newNode->next = pre->next;
17         pre->next = newNode;
18         return 1;
19     }
20     return 0;
   }
```



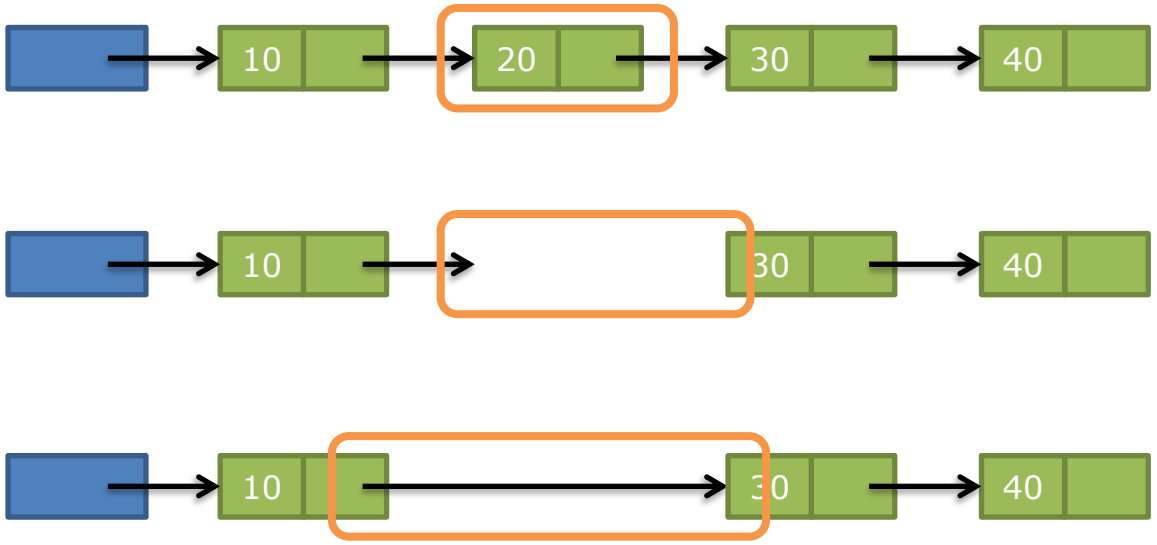
HOW TO USE THE LINKED LIST?



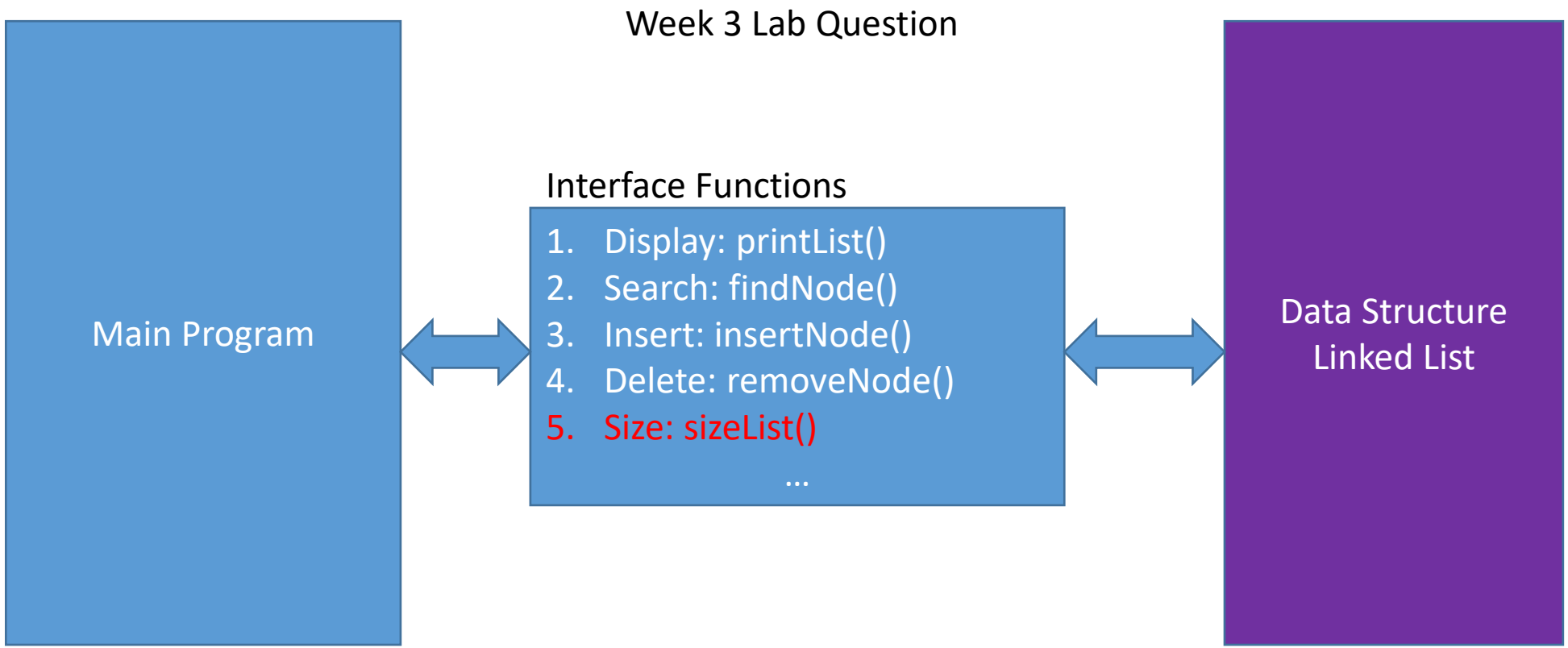
REMOVE A NODE: removeNode()



- Remember to free up any unused memory
- Remove a node at
    1. Front
    2. Middle
    3. Back



HOW TO USE THE LINKED LIST?



SIZE: sizeList()

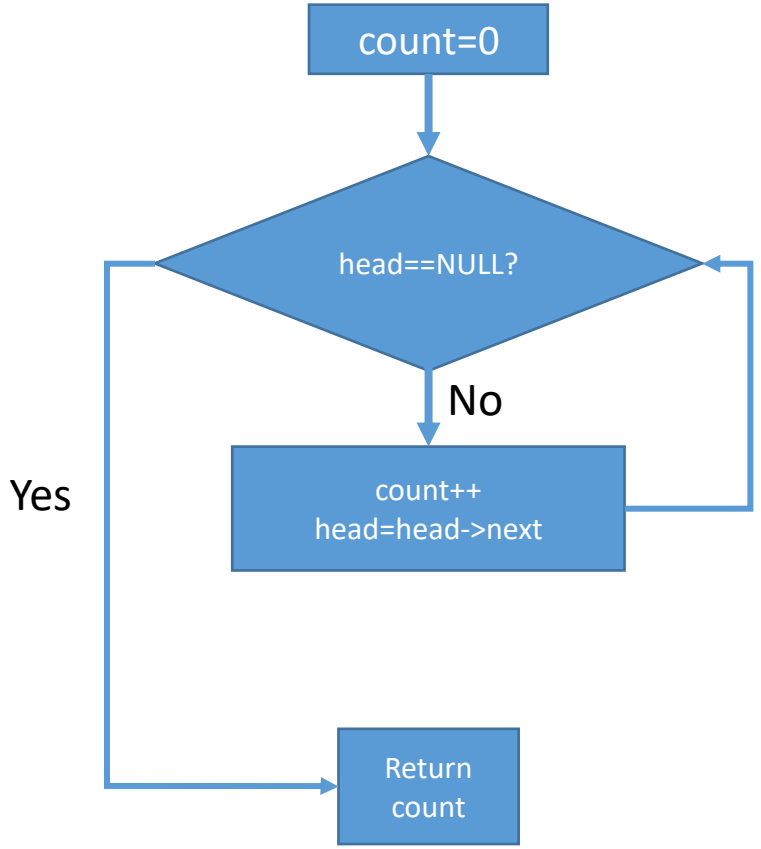
```
int sizeList(ListNode *head);
```

Given

- **the head pointer of the linked list**

Return the number of nodes in the linked list

1. **Declare a counter and initialize it to zero**
2. **Check the pointer whether is NULL or not**
3. **Increase the counter**
4. **Head move to next node**
5. **Repeat step 2**
6. **Return the counter**



SIZE: sizeList()

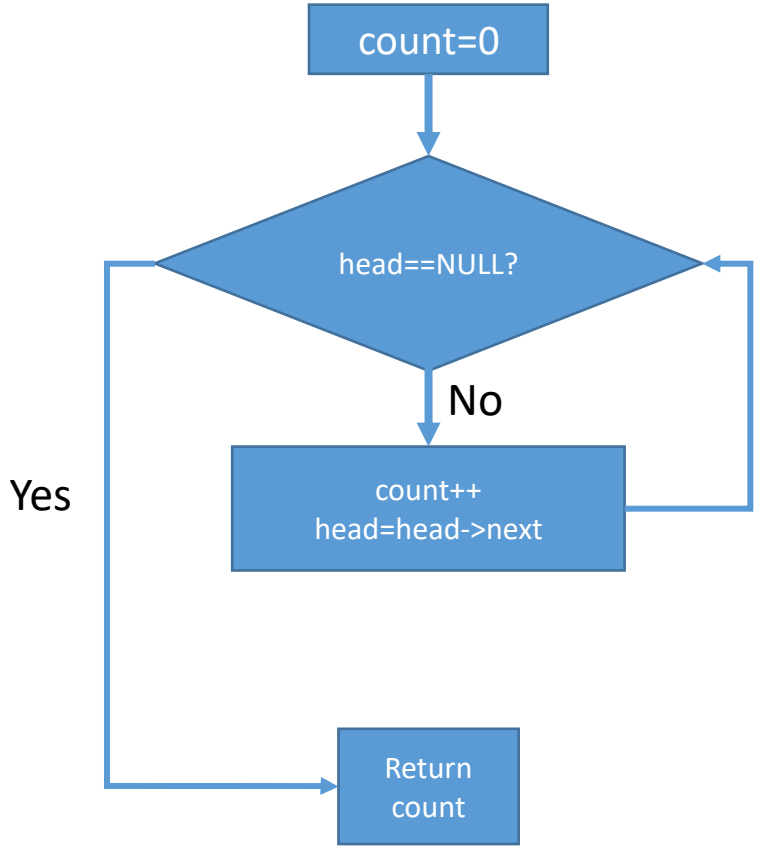
```
int sizeList(ListNode *head);
```

Given

- **the head pointer of the linked list**

Return the number of nodes in the linked list

```
1   int sizeList(ListNode *head){
2
3       int count = 0;
4
5       while (head != NULL){
6           count++;
7           head = head->next;
8       }
9
10      return count;
11  }
```



Why do you need a linked list?

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**

Can we improve our sizeList()?

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 }
```

ARRAYS VS. LINKED LISTS

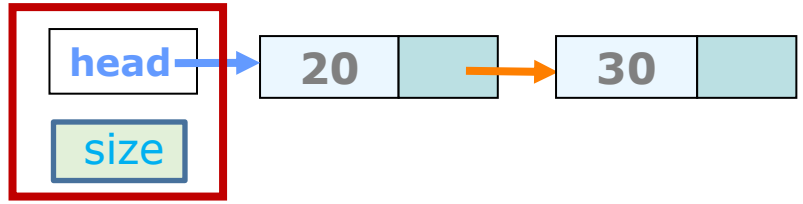
- **Arrays**
  - Efficient random access
  - Difficult to expand, re-arrange
  - When inserting/removing items in the middle or at the front, computation time scales with size of list
  - Generally a better choice when data is immutable
- **Linked lists (dynamic-pointer-based and static-array-based)**
  - “Random access” is hardly implemented
  - cost of storing links, only use internally.
  - Easy to shrink, rearrange and expand (but array-based linked list has a fixed size)
  - Insert/remove operations only require fixed number of operations regardless of list size. no shifting
- Know when to choose an array vs a linked list

CAN WE IMPROVE OUR sizeList()?

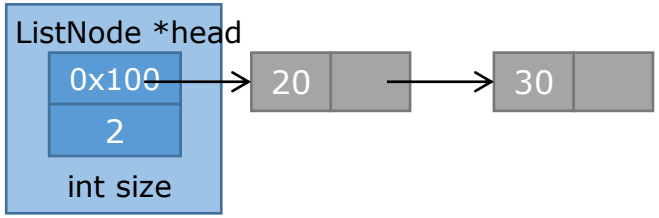
```
1  int sizeList(ListNode *head){
2      int count = 0;
3      while (head != NULL){
4              count++;
5              head = head->next;
6      }
7      return count;
8  }
```

- 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  }
```



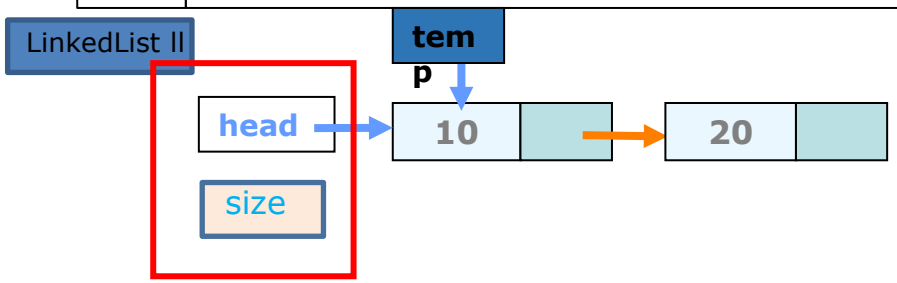
- Remember to change size when adding/removing nodes

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);**

NEW printList()

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




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

```
1  void printList(LinkedList ll){
2      ListNode *temp = ll.head;
3
4      while (temp != NULL){
5          printf("%d\n", temp->item);
6          temp = temp->next;
7      }
8  }
```

NEW findNode()

```
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 }
```

HOMEWORK

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

```
 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  }
```

```
 1  int insertNode(LinkedList *ll, int i, int item){
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18  }
```

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

c_2 + 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

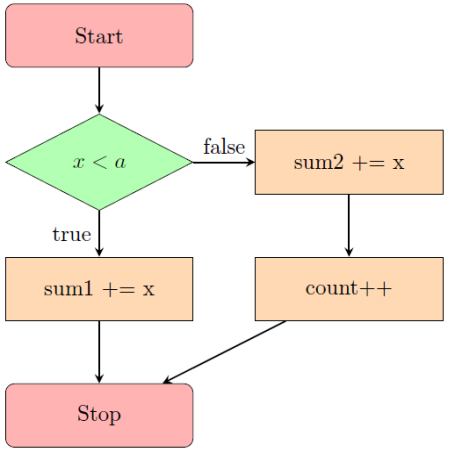
$m(n(c_1))$

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 }
```

$W_1 = 1$

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

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

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

…

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

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

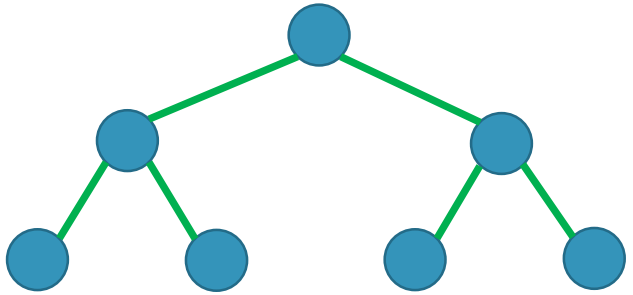
It is known as a **method of backward substitutions**

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

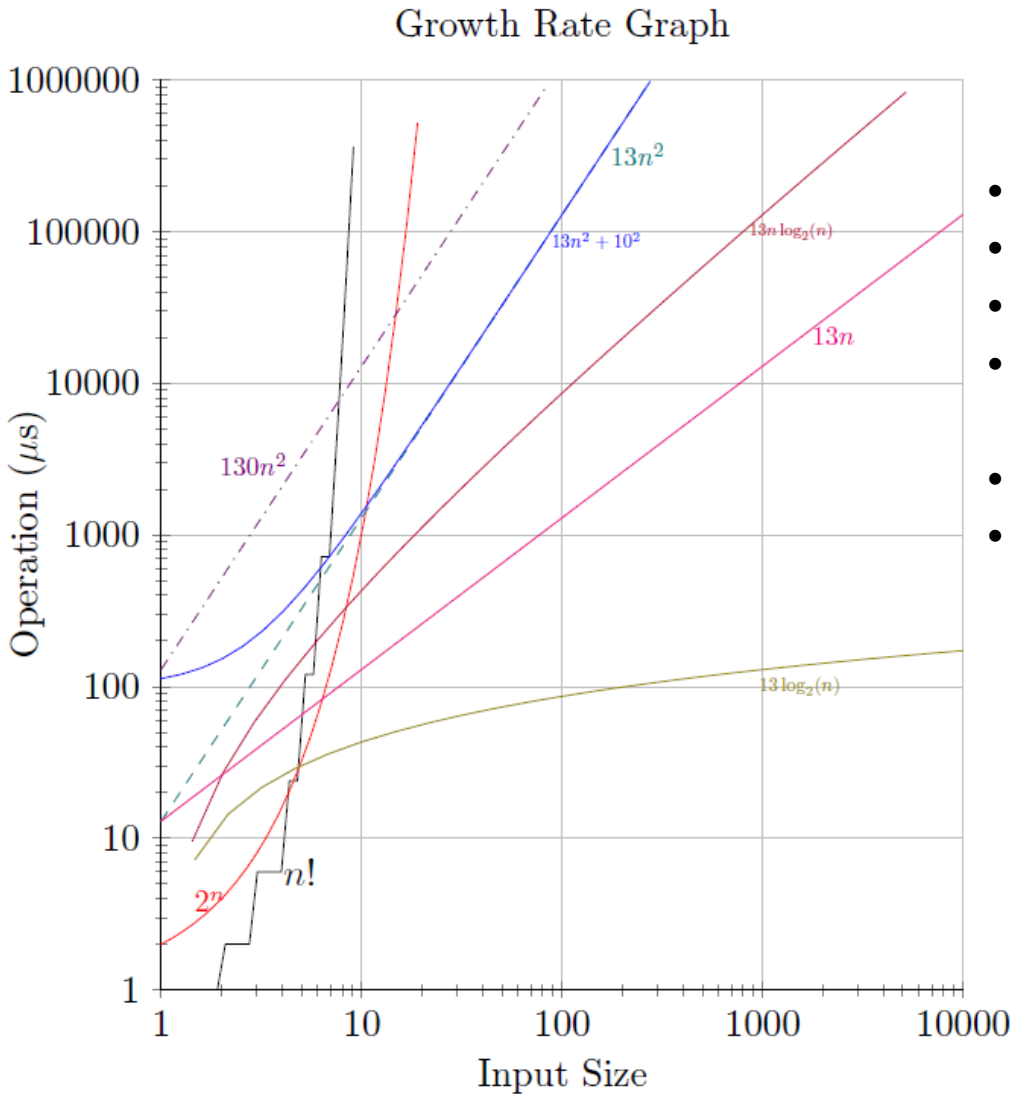
Problem size (n)

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

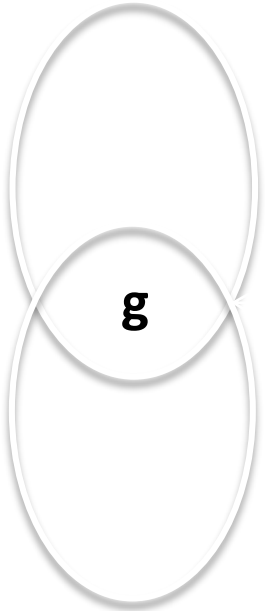


- n! is the fastest growth
- 2^n is the second
- 13n is linear
- $13\log_2 n$ is the slowest

- 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**

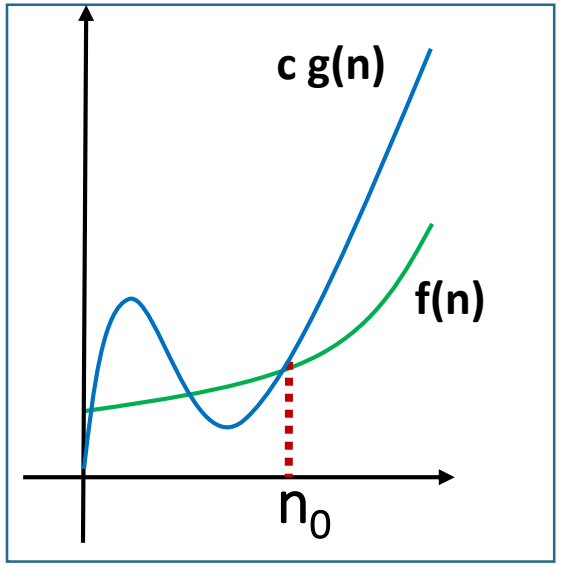
$f \in \Theta(g)$ Set of functions that grow at same rate as **g**

$f \in O(g)$ 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

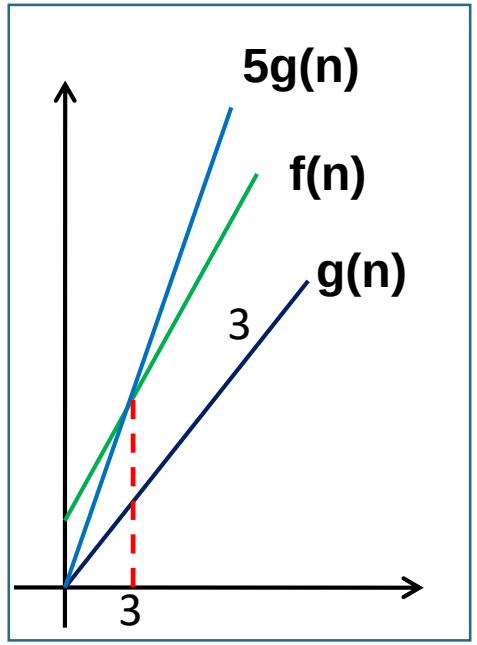
Let c = 5, n0 = 3

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

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

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

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



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

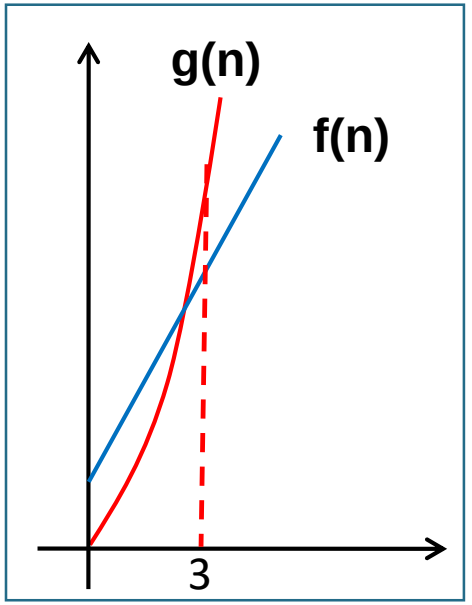
Let c = 1, n0 = 3

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

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

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

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



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

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

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

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

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

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

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*

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

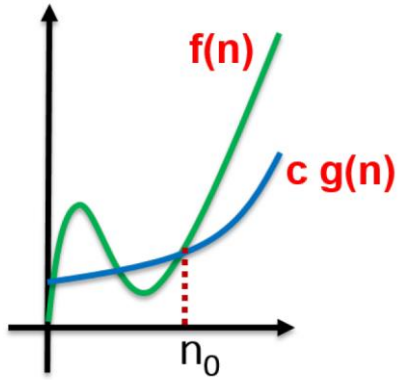
Definition 3.4 *Ω-notation: Let f and g be two functions such that $f(n) : \mathbb{N} \to \mathbb{R}^+$ and $g(n) : \mathbb{N} \to \mathbb{R}^+$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, then $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$.*

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

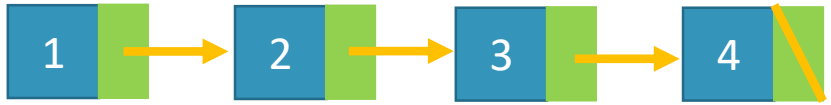
	$f(n) \in O(g(n))$	$f(n) \in \Omega(g(n))$	$f(n) \in \Theta(g(n))$
$\mathbf{0}$	✓		
$\mathbf{0 < c < \infty}$	✓	✓	✓
∞		✓	

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

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

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

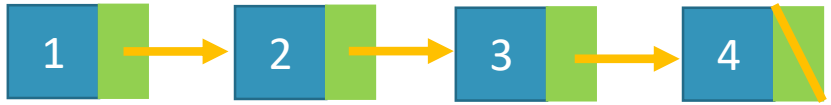
$$= c_1 + \frac{c_2}{n}\cdot\frac{n}{2}(0 + (n-1)) = c_1 + \frac{c_2(n-1)}{2} = \Theta(n)$$

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

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

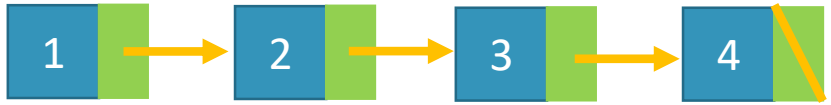
Since the probability of the search key is in the list is unknown, we only can have

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

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 }
```

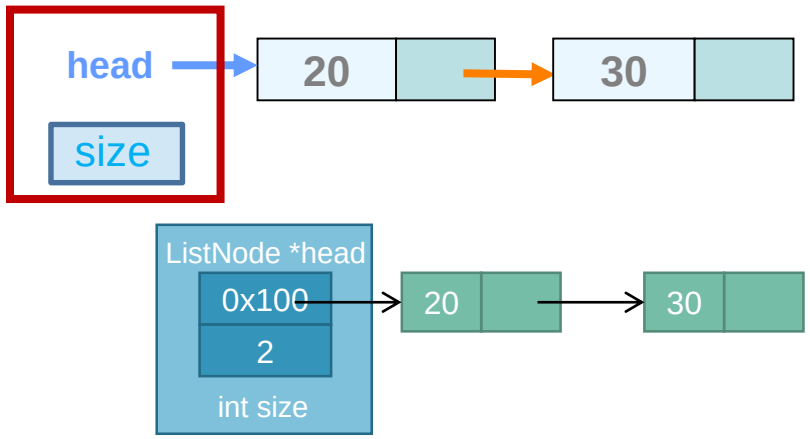
Can we improve our sizeList()?

```
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**

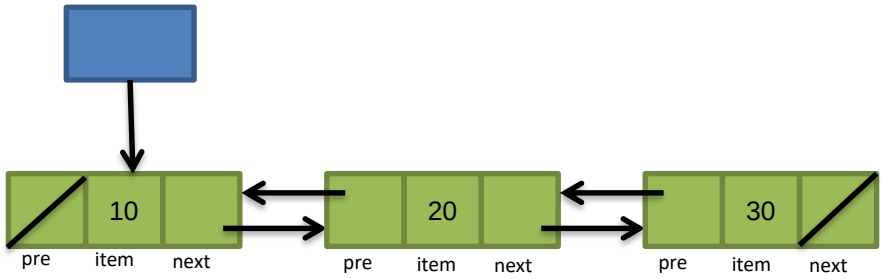
Doubly Linked List

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

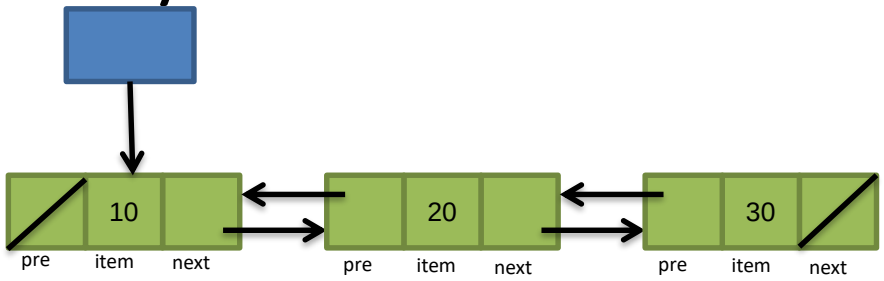
```
struct _listnode
{
    int item;
    struct _listnode *next;
};
typedef struct _listnode ListNode;
```

- 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

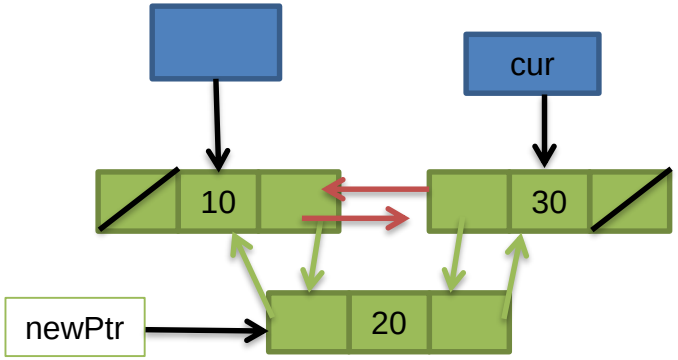


Interface Functions

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:

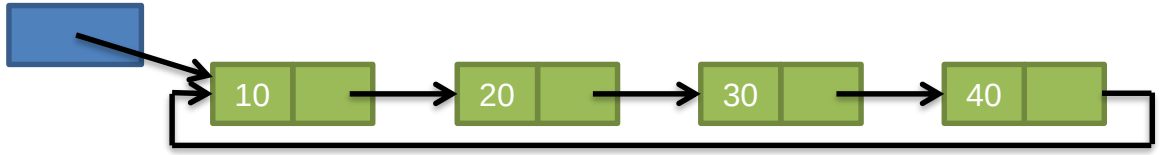
```
newPtr->next = cur;
newPtr->pre = cur->pre;
cur->pre= newPtr;
newPtr->pre->next=newPtr;
```



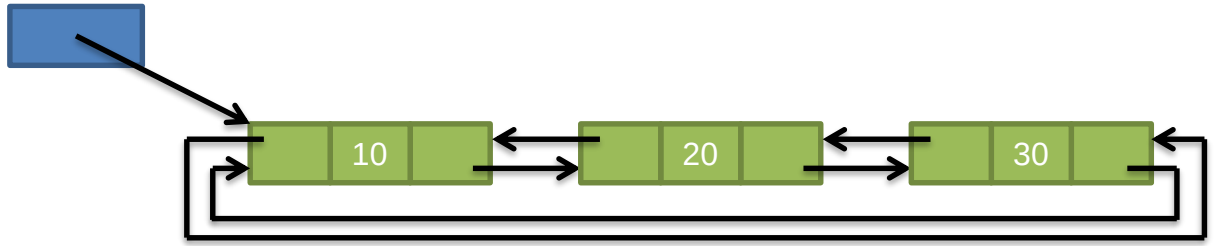
- It is noted that the solution is not unique.
- Delete function will be easier than Singly Linked List's.

Circular linked lists

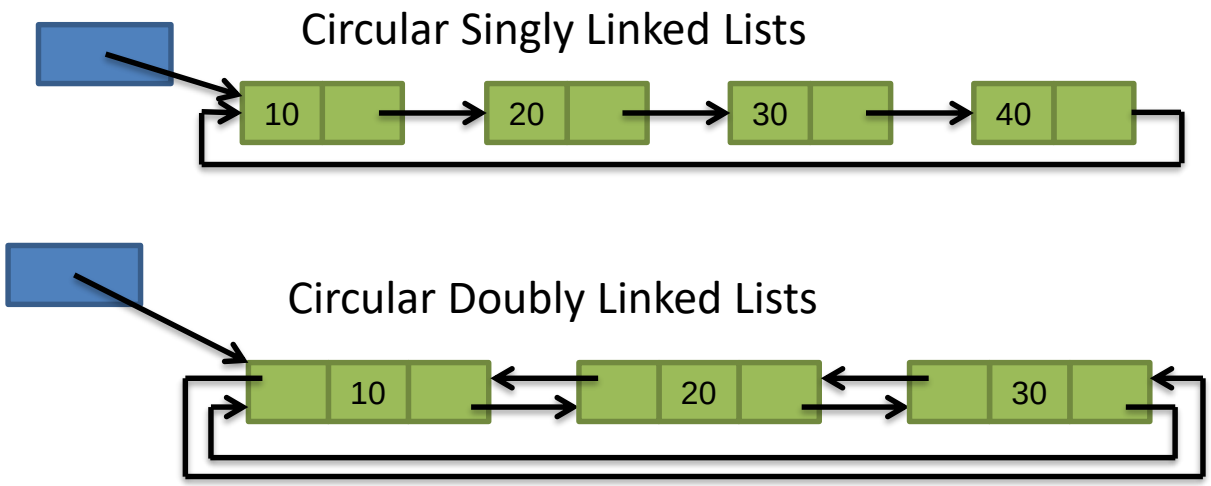
- Circular singly linked lists
  - Last node has next pointer pointing to first node



- 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

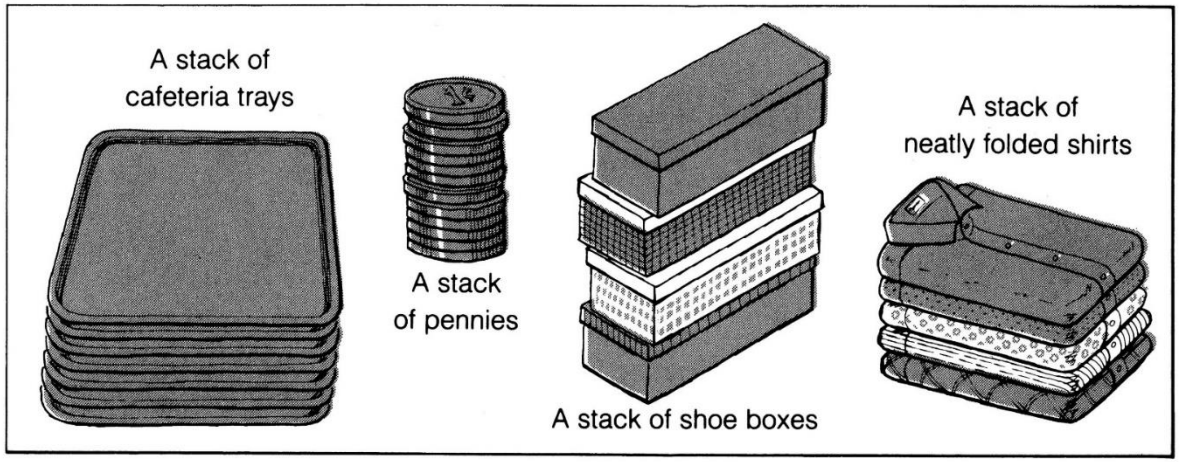
Stack and Queue

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;
```

```
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()`

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);`

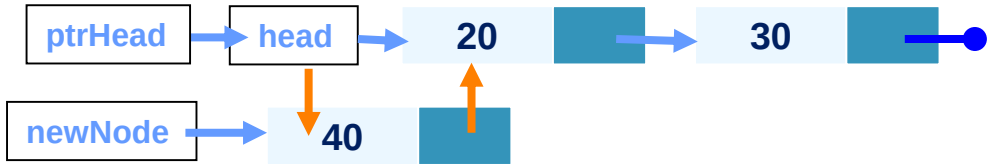
```
int peek(Stack s){
    return s.head->item;
}
```

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()`

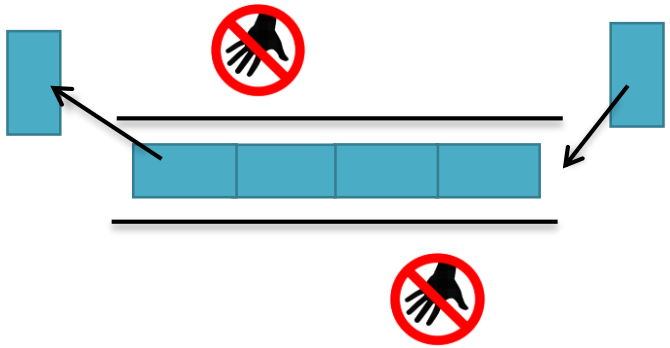
Queue Function

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()

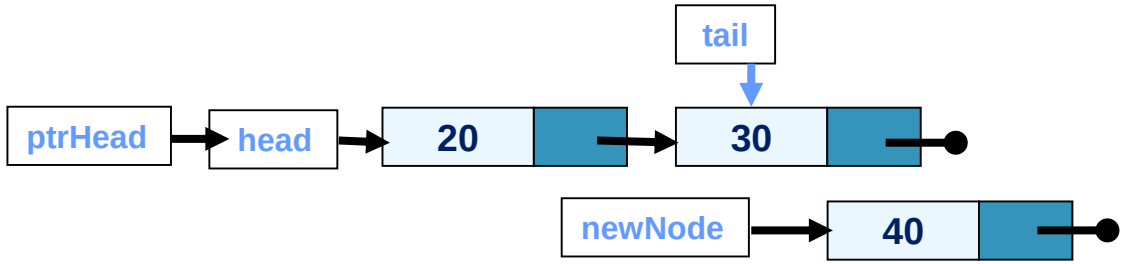
- Put a new node into the Queue-> insert a node at index *size*
- To make the `insertNode()` more efficient, *tail is introduced
- `enqueue()` needs to be rewritten to let *tail point to the last node
- Queue is empty (Size=0) is a special case

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

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

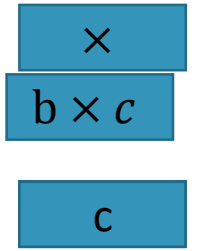
- Use a stack
- When the character is an operand, push it to the stack

- 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

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

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

- 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

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

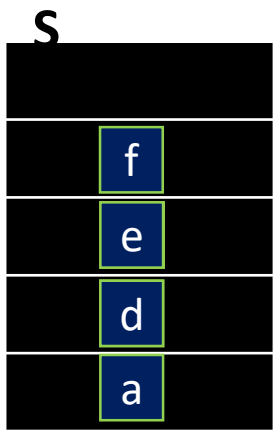
Algorithm 2 Evaluation Postfix Expression

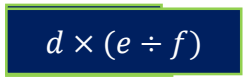
```
function ExePost(String postfix)
    create a Stack S
    for each character c in postfix do
        if c is an operand then
            push(c, S)
        else
            operand1 ← pop(S)
            operand2 ← pop(S)
            result ← Evaluate(operand2, c, operand1)
            push(result,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

CX1107
Data Structures and Algorithms

Trees

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()**
- **free()**

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

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**

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

...

```c
void printList(ListNode *cur){
    while (cur != NULL){
        printf("%d\n", cur->item);
        cur = cur->next;
    }
}
```

```c
int sizeList(ListNode *head){
    int count = 0;
    while (head != NULL){
      count++;
      head = head->next;
    }
    return count;
}
```

```c
ListNode *findNode(ListNode* cur, int i){
   if (cur==NULL || i<0)
      return NULL;
   while(i>0){
      cur=cur->next;
      if (cur==NULL)
         return NULL;
      i--;
   }
   return cur;
}
```

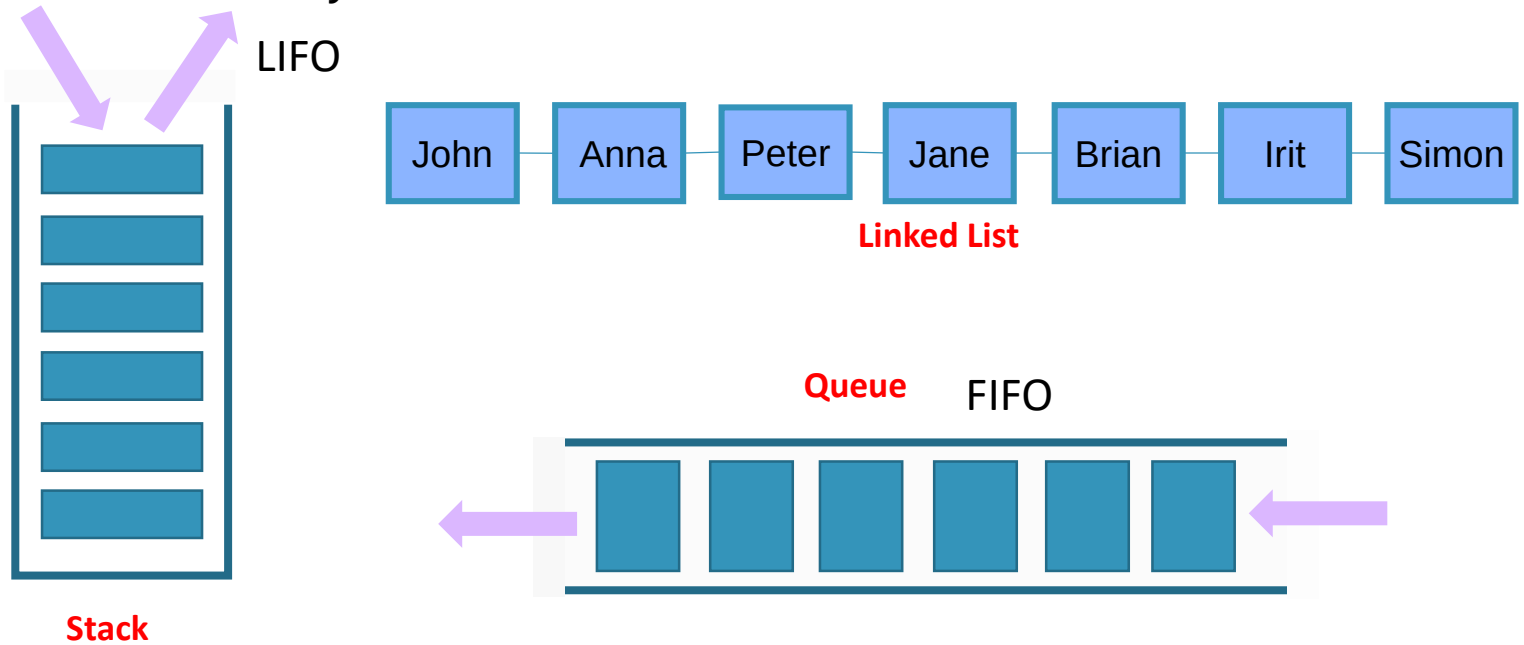
```c
int insertNode(ListNode **ptrHead, int i, int item){
    ListNode  *pre, *newNode;
    if (i == 0){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
        return 1;
    }
    else if ((pre = findNode(*ptrHead, i-1)) != NULL){
        newNode = malloc(sizeof(ListNode));
        newNode->item = item;
        newNode->next = pre->next;
        pre->next = newNode;
        return 1;
    }
    return 0;
}
```

Stacks and Queues

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

1. **Variations of the linked list**
   - **Doubly Linked List**
   - **Circular Linked List**
   - **Circular Doubly Linked List**
2. **Stacks and Queues**

All these dynamic data structures are linear data structures




```
typedef ListNode StackNode;

typedef LinkedList Stack;
```

Stack

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

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

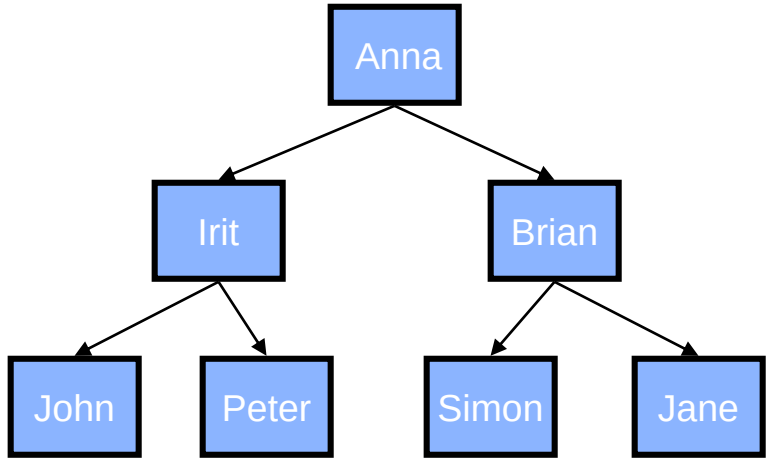
Queue

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

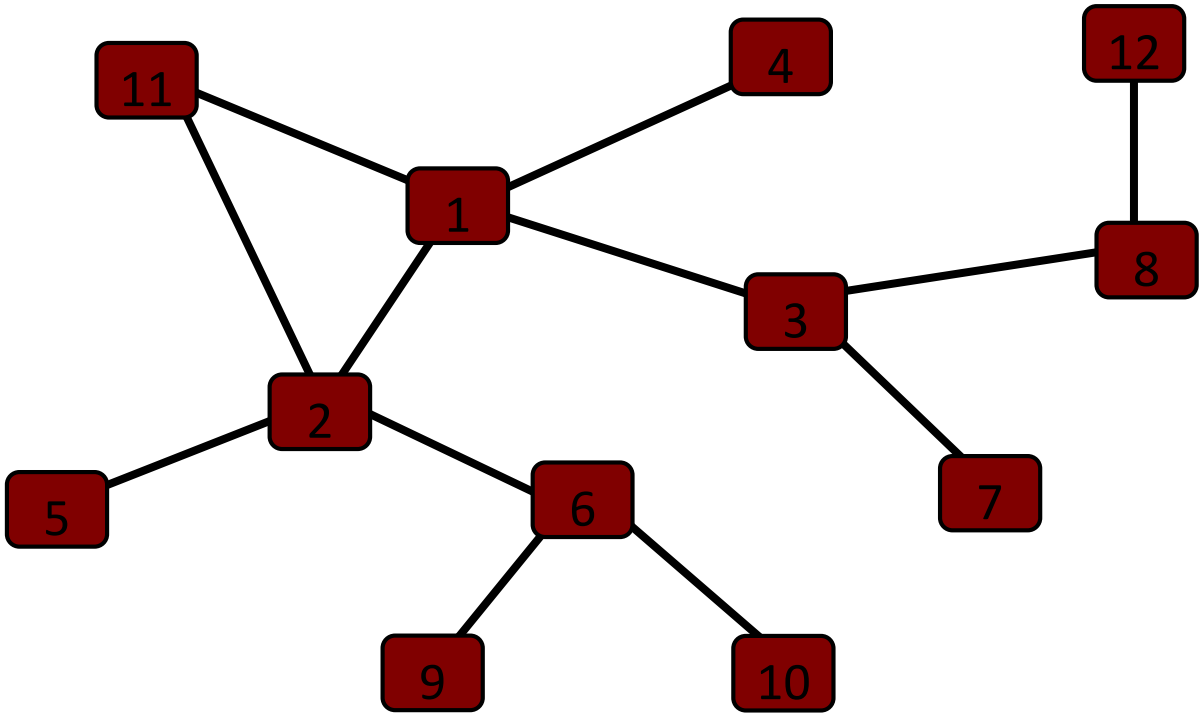
Overview

1. **What are trees?**
2. **Why do you need a tree?**
3. **How to create a tree?**
4. **How to use the tree?**

What are trees?



Tree



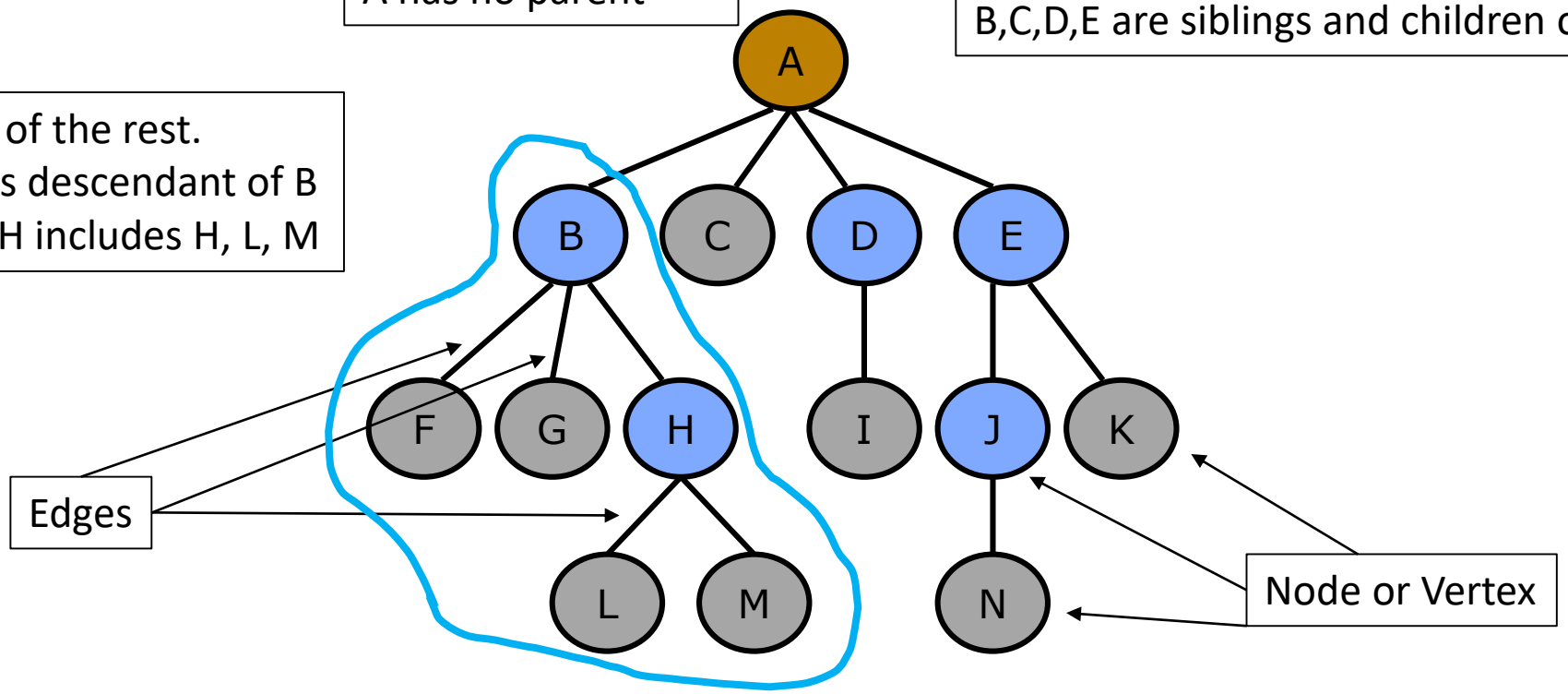
Graph

Terminology In Tree

A is the root of Tree
A has no parent

A is the parent of B,C,D,E
B,C,D,E are siblings and children of A

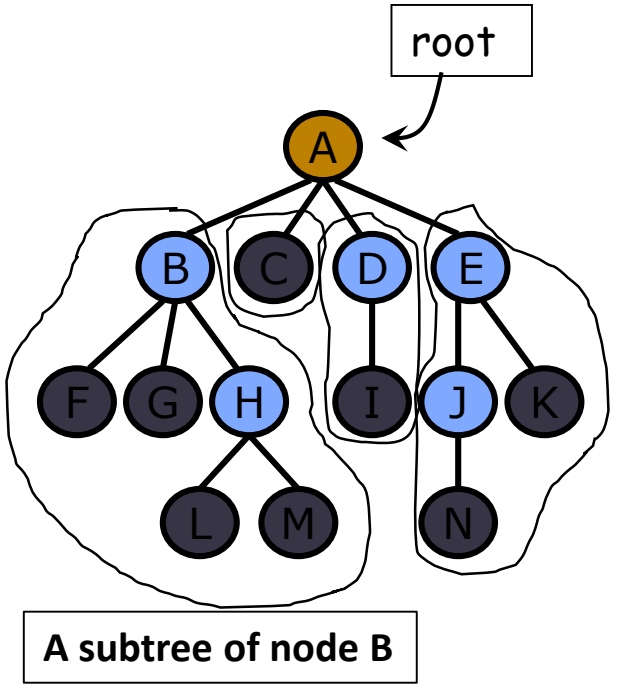
A is ancestor of the rest.
F, G, H, L, M is descendant of B
A subtree of H includes H, L, M



L and M has no children
L and M are leaves

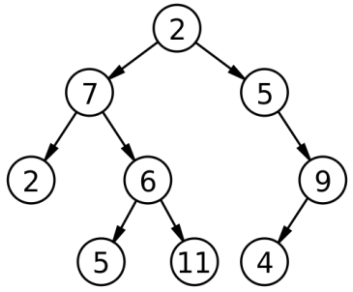
Tree Data Structure

- Similar to family tree concept
- One special node: root
- Each node can has many children
  A has four children: B, C, D, E
- Each node (except the root) has a parent node
- **A is the parent of B, C, D, E**
- Other children of your parent are your siblings
- **B, C, D and E are siblings**
- **Subtree**: Any node in the tree together with all of its descendants for a subtree.

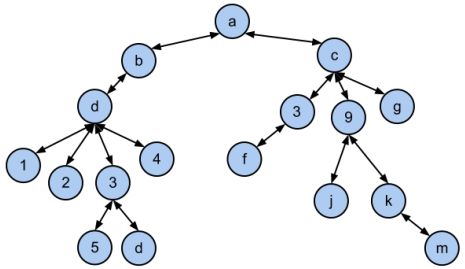


Tree Data Structure

- Tree data structure looks like… a tree: root, branches, leaves
  - Only one root node which has no parent
  - Each node branches out to some number of nodes
    - For binary tree, each node has up to two children (left and right child)
  - Each node has only one “parent” node – the node pointing to it (except the root node)



Binary Tree

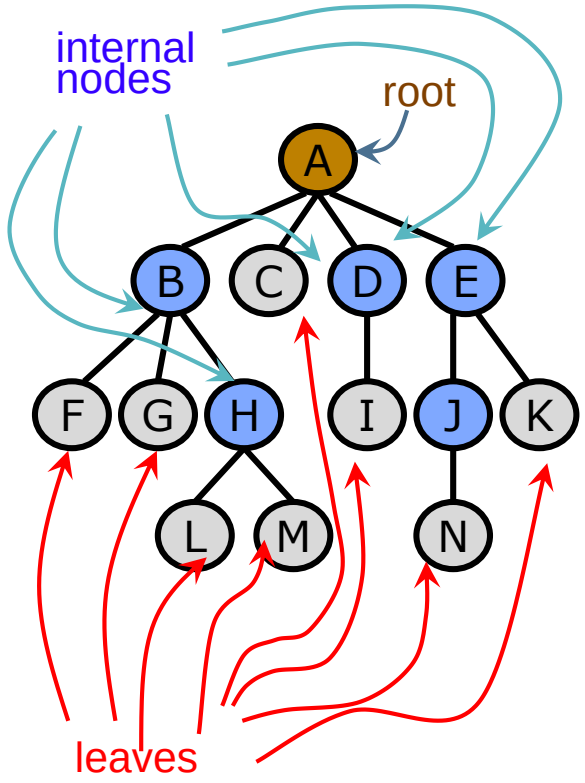


General Tree

- General tree
  - Each node can have links to any number of other nodes

Tree Data Structure

- A tree is composed of nodes
- Types of nodes
  - Root: only one in a tree, has no parent.
  - Internal node(non-leaf): Nodes with children are called internal nodes
  - Leaf (External Node): nodes without children are called leaves



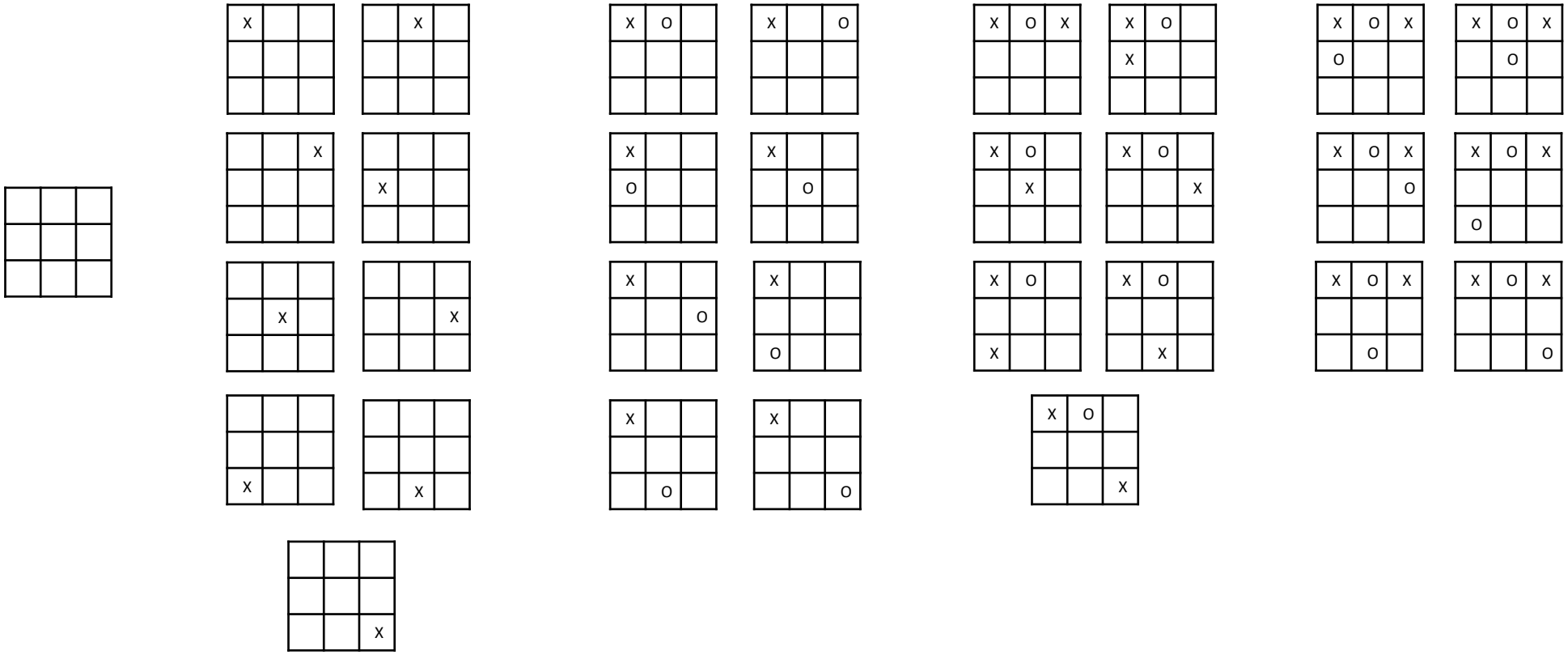
1. What are trees?
2. **Why do you need a tree?**
3. How to create a tree?
4. How to use the tree?

Why Trees?

- Model layouts with hierarchical relationships between items
  - Chain of command in the army
  - Personnel structure in a company
- Optimization problems – Huffman coding (a lossless data compression algorithm. It assigns variable-length codes to input characters based on the usage frequency)
- Permutation, Searching Problems –
  - Eight Queens Problem
  - Gaming eg. Sudoku, Tic-tac-toe

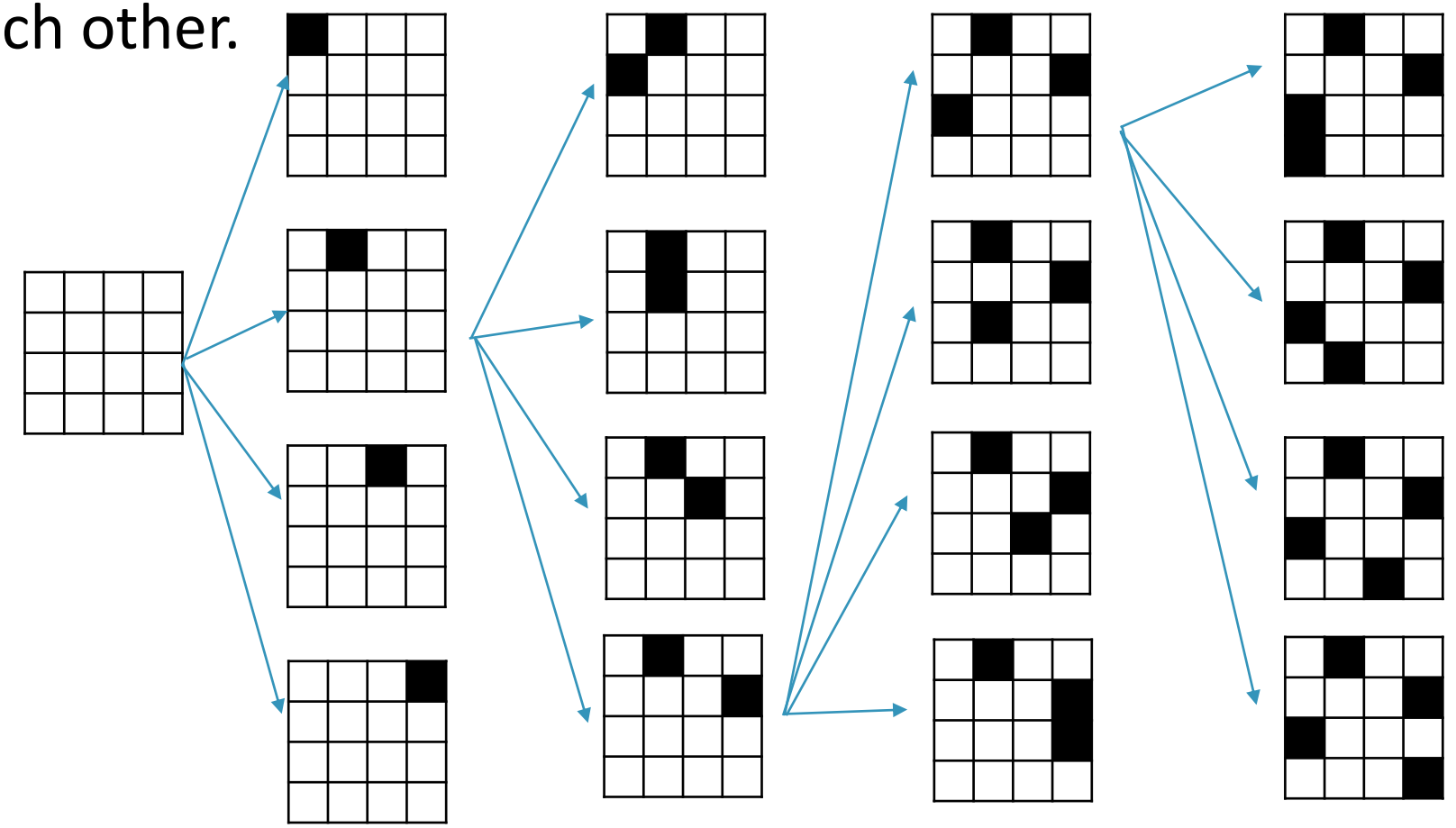
Tic-Tac-Toe

Tic-tac-toe aka noughts and crosses is a paper and pencil game for two players, who take turns marking the spaces in a 3x3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical or diagonal row wins the game

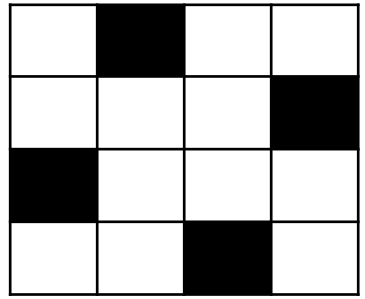


Four Queens Puzzle

Place four queens on a 4x4 chessboard so that no two queens can capture each other.

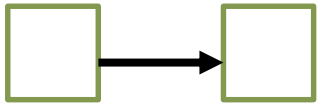


- Each node has 4 children
- The height of the tree is 5

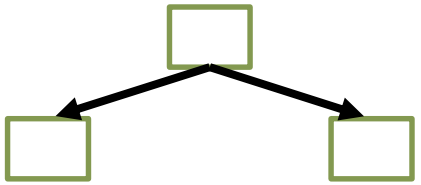


1. What are trees?
2. Why do you need a tree?
3. **How to create a tree?**
4. How to use the tree?

Binary Tree Structure

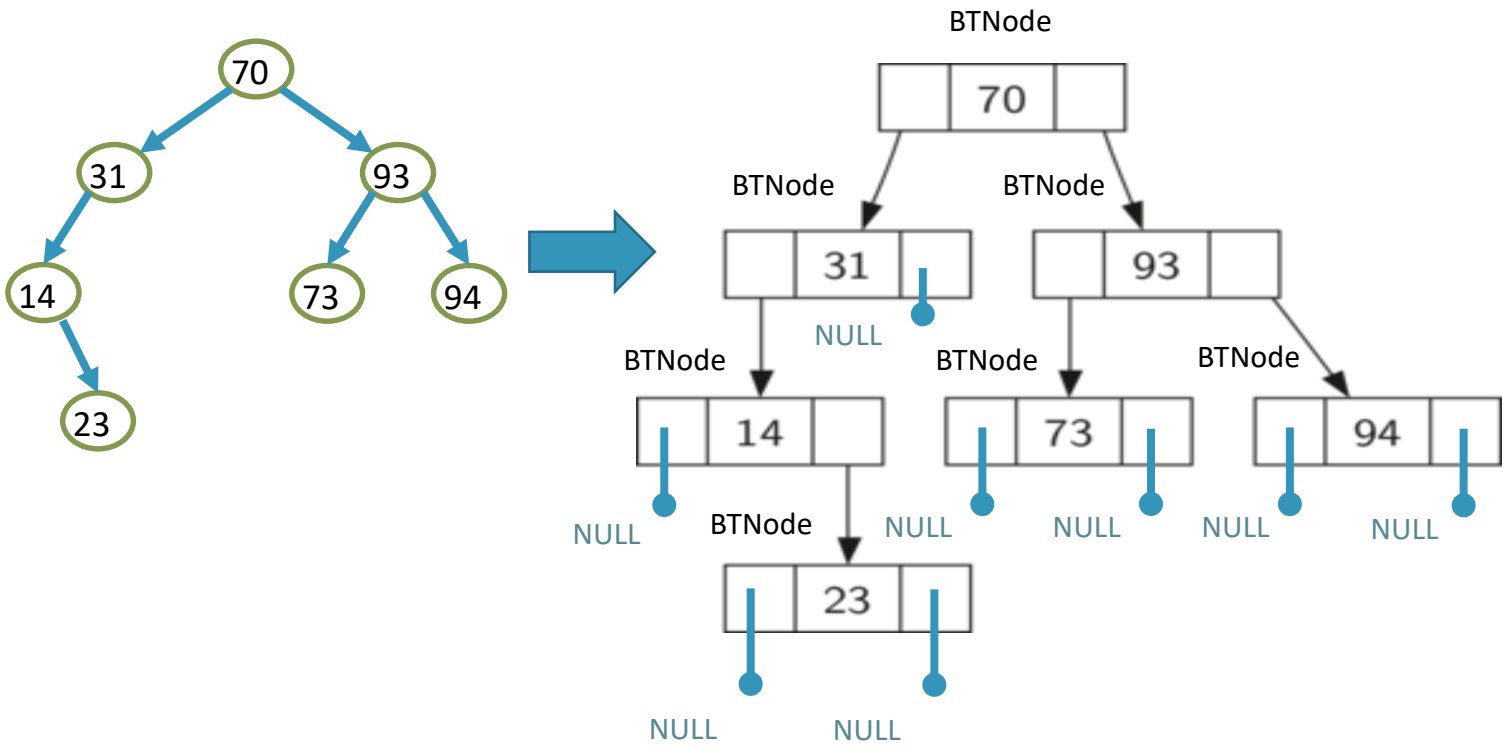


```
typedef struct _listnode{
    int item;
    struct _listnode *next;
}ListNode;
```

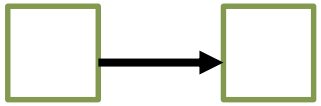


```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

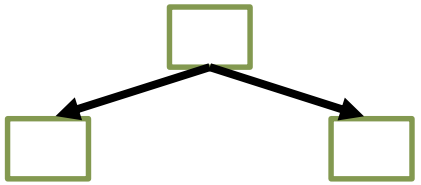
Example Binary Tree



Tree Traversal Problems



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



```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Interface Functions

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

…

Traversal Problem:
How to systematically travel each node in the tree?

1. What are trees?
2. Why do you need a tree?
3. How to create a tree?
4. **How to use the tree?**
    - **Binary Tree Traversal**

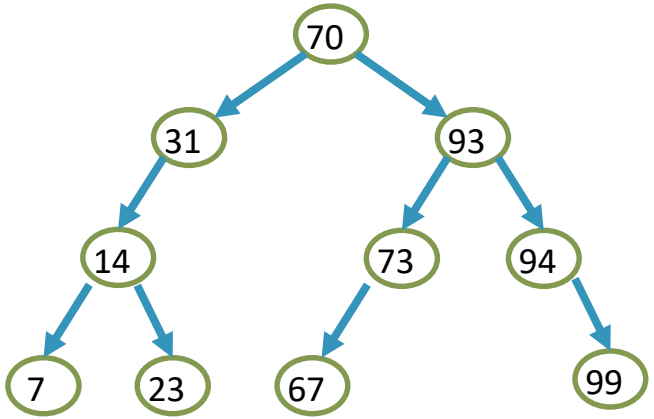
Binary Tree Traversal

```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Given a binary tree,

how do you systematically visit every nodes once only?

- **Print the contents of a tree**
- **Search a node**
- **Find the size of a tree**



- **Insert a node**
- **Remove a node**

Binary Tree Traversal

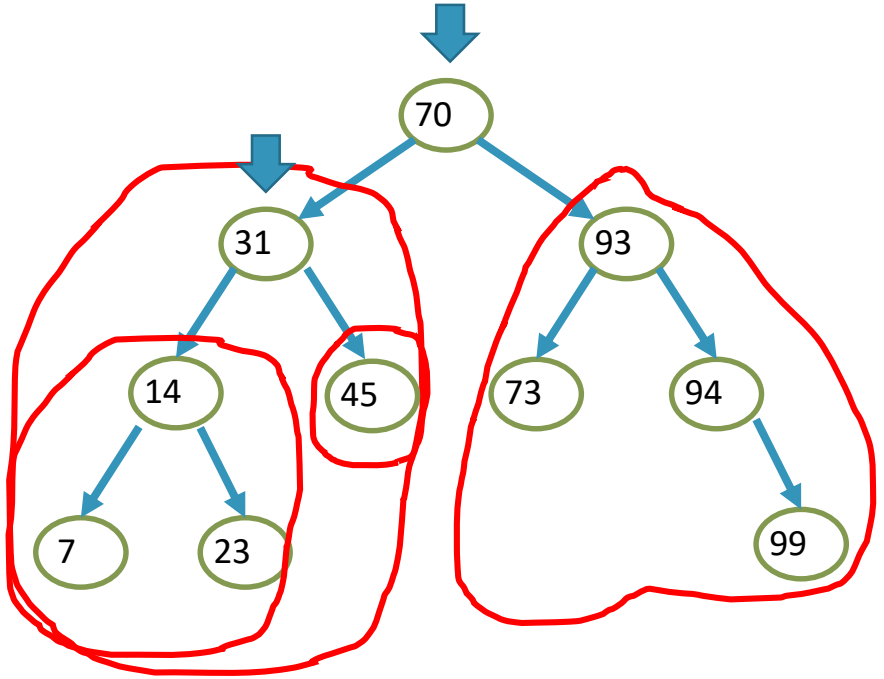
```
typedef struct _btnode{
    int item;
    struct _btnode *left;
    struct _btnode *right;
} BTNode;
```

Traversal Problem:

- **Visit root + right subtree + left subtree**
- **Each subtree repeat the same procedure**

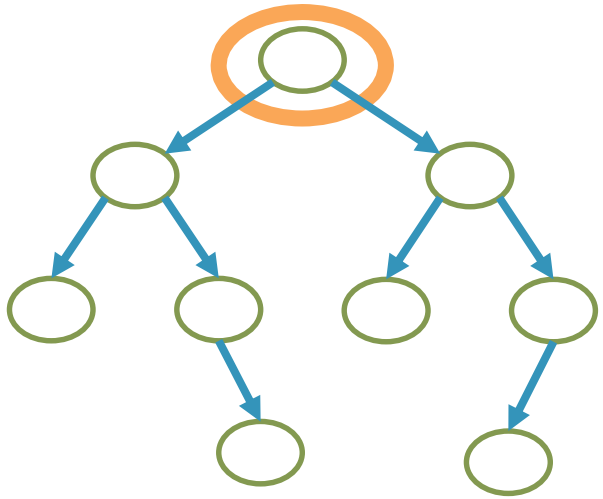
  visit root + right subtree + left subtree
- **Until reach the leave**

  visit the root (leaf only)
- **It is a recursive problem**



Pseudocode of Binary Tree Traversal

```
TreeTraversal(Node N):
    Visit N;
    If (N has left child)
        TreeTraversal(LeftChild);
    If (N has right child)
        TreeTraversal(RightChild);
    Return; // return to parent
```



This traversal approach is known as pre-order depth first traversal.

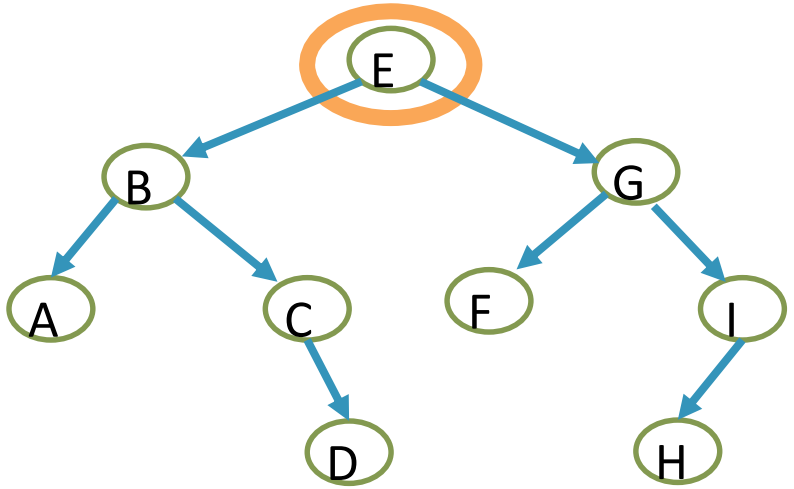
Traversal Approaches on A Binary Tree

- Depth-First Traversal: From the root of a tree, it explores as far as possible. Then it will do the backtracking. There are three traversal orders:
  - Pre-order
  - In-order
  - Post-order

- Breadth-First Traversal: From the root of a tree, it explores each node in level by level.

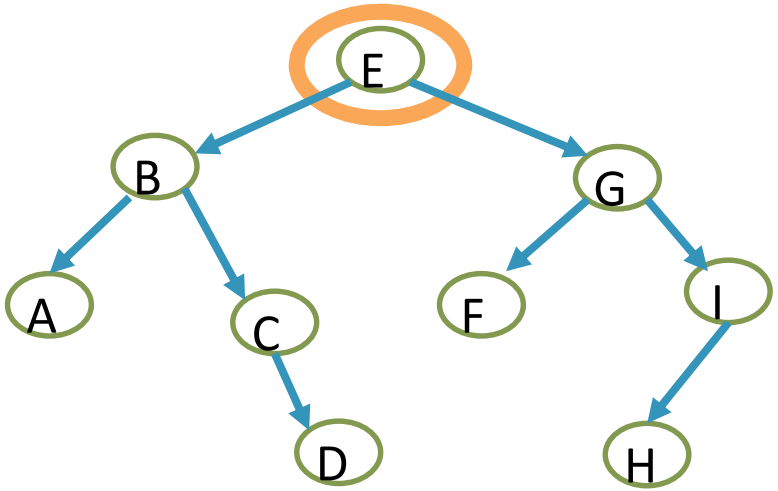
Depth First Traversal: Pre-Order

- **Pre-order**
  - **Process the current node's data**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
- In-order
- Post-order



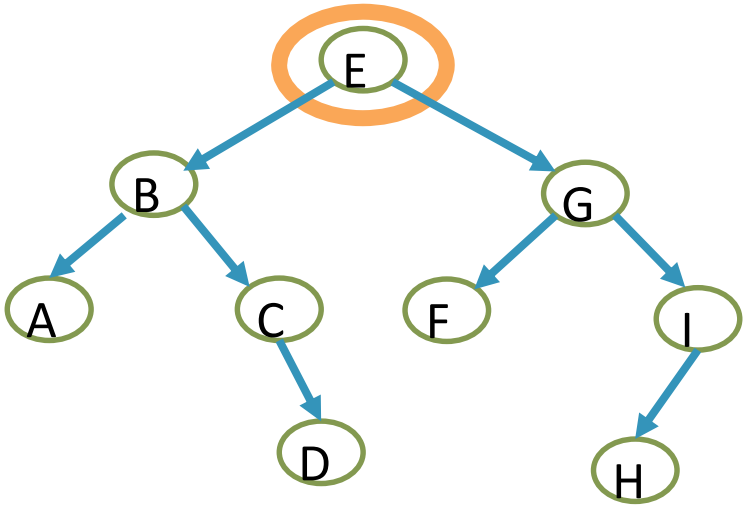
Depth First Traversal: In-Order

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- **In-order**
  - **Visit the left child subtree**
  - **Process the current node's data**
  - **Visit the right child subtree**
- Post-order



Depth First Traversal: Post-Order

- Pre-order
  - Process the current node's data
  - Visit the left child subtree
  - Visit the right child subtree
- In-order
  - Visit the left child subtree
  - Process the current node's data
  - Visit the right child subtree
- **Post-order**
  - **Visit the left child subtree**
  - **Visit the right child subtree**
  - **Process the current node's data**



Traversal Approaches on A Binary Tree

- Depth-First Traversal: From the root of a tree, it explores as far as possible. Then it will do the backtracking. There are three traversal orders:
  - Pre-order
  - In-order
  - Post-order

- **Breadth-First Traversal**: From the root of a tree, it explores each node in **level by level**.

Breadth-First Traversal: Level-by-level

Level-By-Level Traversal:

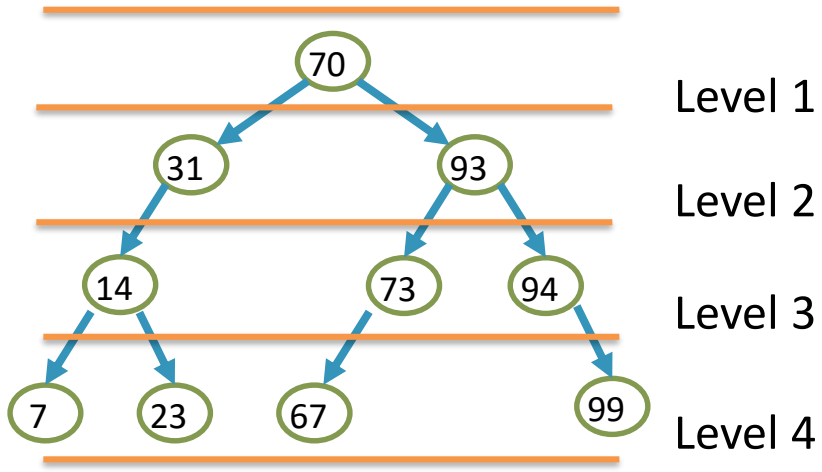
Visit the root (Level 1)

Visit children of the root (Level 2)

Visit grandchildren of the root (Level3) ...

How?

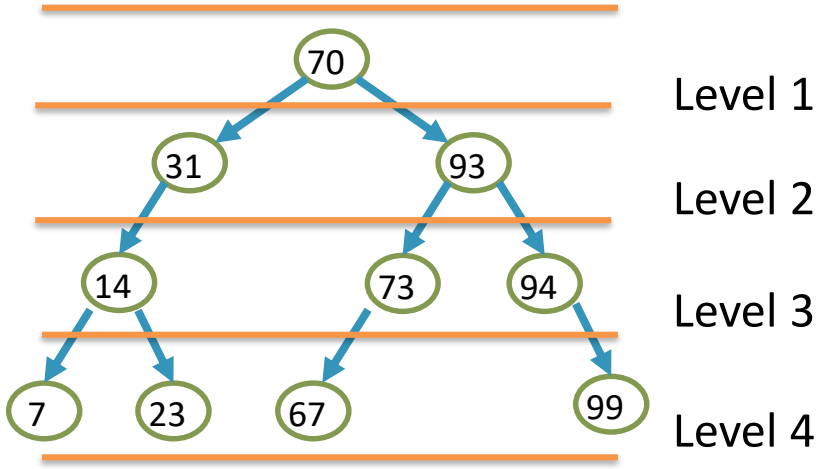
- Visiting the node
- Remember all its children
  - Use a queue (FIFO structure)



Breadth-First Traversal: Level-by-level

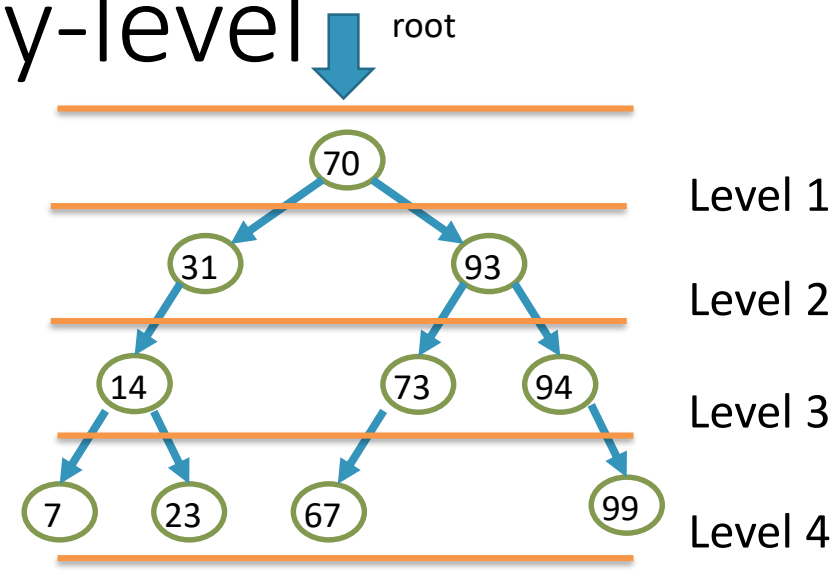
Level-By-Level Traversal:

- Visiting the node
- Remember all its children
  - Use a queue (FIFO structure)



1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

Breadth-First Traversal: Level-by-level



Level-By-Level Traversal:

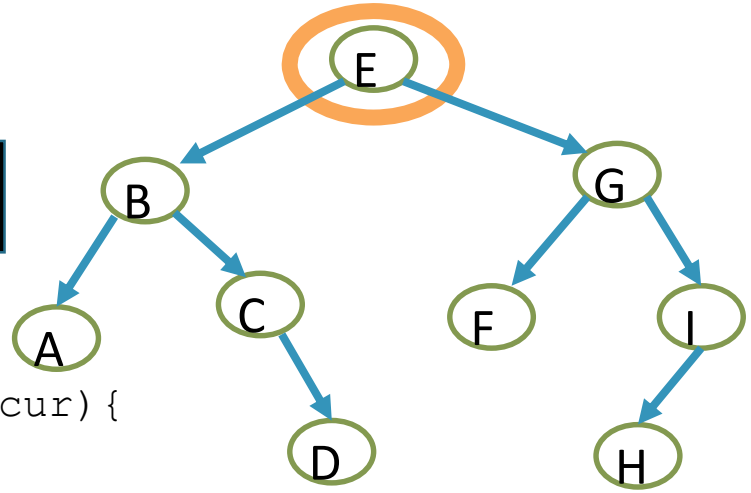
1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

```
void BFT(BTNode *root){
    Queue *q;
    BTNode* node;
    if(root){
       enqueue(q,root); //data type of item in queue is BTNode*
       while(!isEmptyQueue(*q)){
            node = getFront(*q);dequeue(q);
            if(node->left) enqueue(q,node->left);
            if(node->right) enqueue(q,node->right);
       }
    }
}
```

Tree Traversal Pre-order: Print

Output:

```
E B A C D G F I H
```

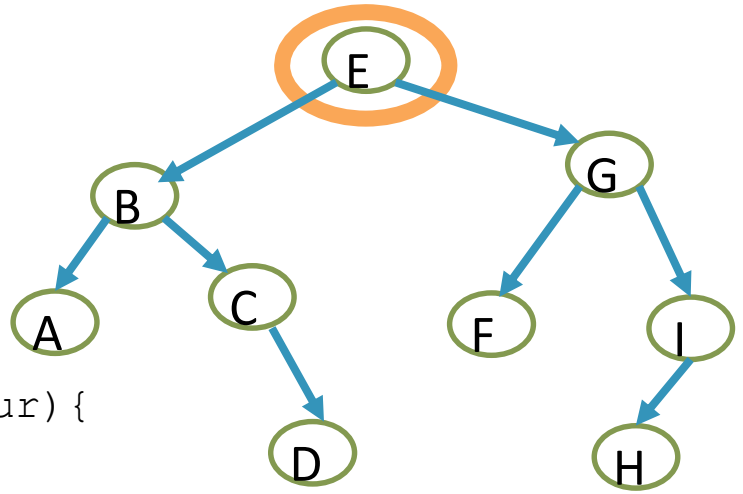



```
void TreeTraversal_pre(BTNode *cur){
    if (cur == NULL)
        return;

    printf("%c  ",cur->item);

    TreeTraversal_pre(cur->left); //Visit the left child node
    TreeTraversal_pre(cur->right);//Visit the right child node
}
```

Tree Traversal In-order: print

Output:

```
A B C D E F G H I
```

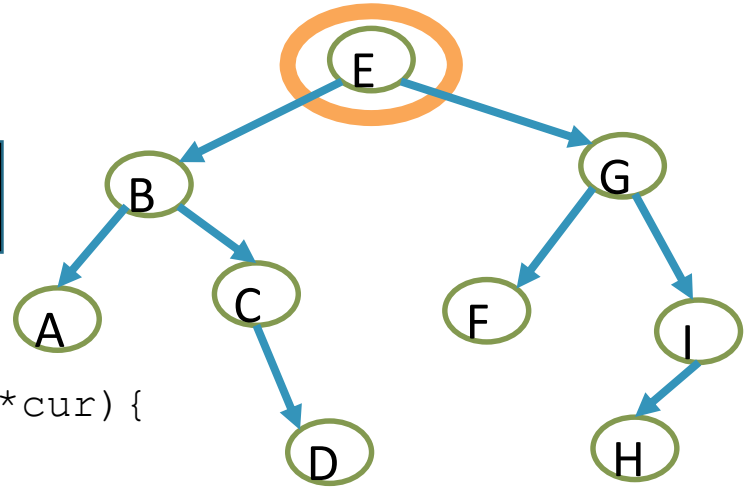



```
void TreeTraversal_in(BTNode *cur){
    if (cur == NULL)
        return;

    TreeTraversal_in(cur->left); //Visit the left child node
    printf("%c  ",cur->item);
    TreeTraversal_in(cur->right);//Visit the right child node
}
```

Tree Traversal Post-order: print

Output:

```
A D C B F H I G E
```




```
void TreeTraversal_post(BTNode *cur){
    if (cur == NULL)
        return;

   TreeTraversal_post(cur->left); //Visit the left child node
   TreeTraversal_post(cur->right);//Visit the right child node
   printf("%c  ",cur->item);
}
```

Pre-Order Traversal

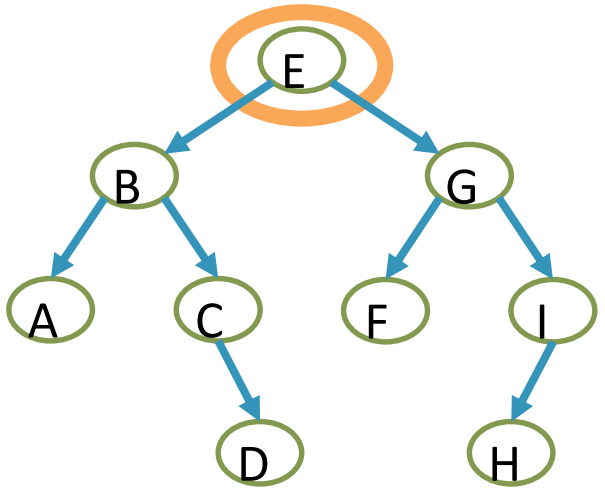
E B A C D G F I H

In-Order Traversal

A B C D E F G H I

Post-Order Traversal

A D C B F H I G E



Count Nodes in a Binary Tree (SIZE)

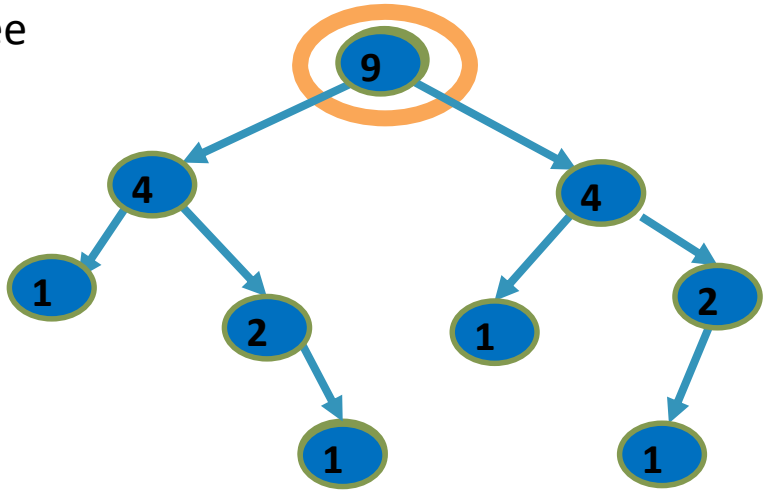
- Recursive definition:
  - Number of nodes in a tree

    = 1

    + number of nodes in left subtree

    + number of nodes in right subtree
- Each node returns the number of nodes in its subtree

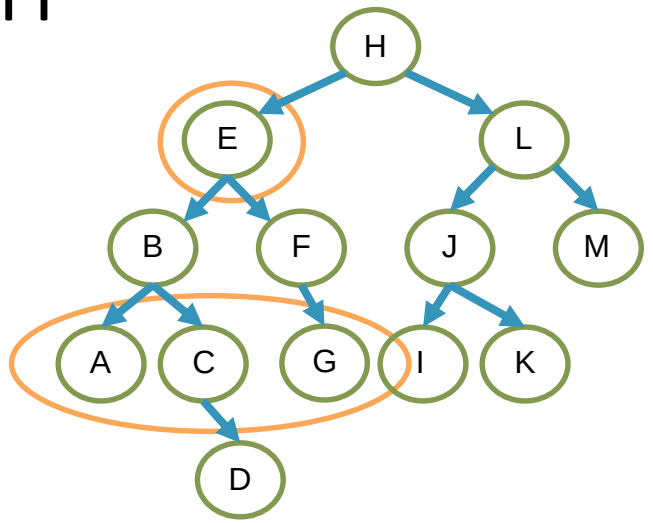
```
int countNode(BTNode *cur){
    if (cur == NULL)
        return 0;

    return (countNode(cur->left)
            + countNode(cur->right)
            + 1);
}
```



Find the k-level Grandchildren

- Given a node X, find all the nodes that are X's grandchildren
- Given node E, we should return grandchild nodes A, C, and G
- What if we want to find k-level grandchildren?
  - **Need a way to keep track of how many levels down we've gone**

```
1. void findgrandchildren(BTNode *cur, int c){
2.      if (cur == NULL) return;
3.      if (c == k){
4.         printf("%d ", cur->item);
5.         return;
6.      }
7.      if (c < k){
8.         findgrandchildren(cur->left, c+1);
9.         findgrandchildren(cur->right, c+1);
10.}
```



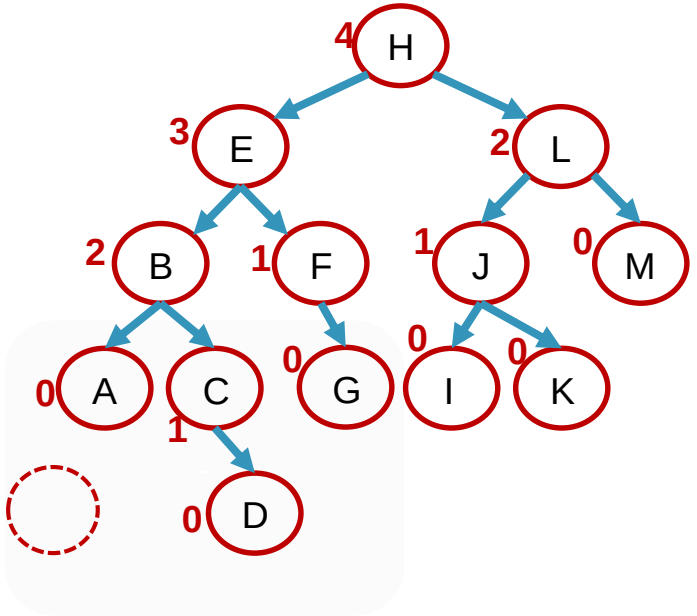
2-level grandchildren

X->left->left
X->left->right
X->right->left
X->right->right

Calculate Height of Every Node

We want each node to report its height

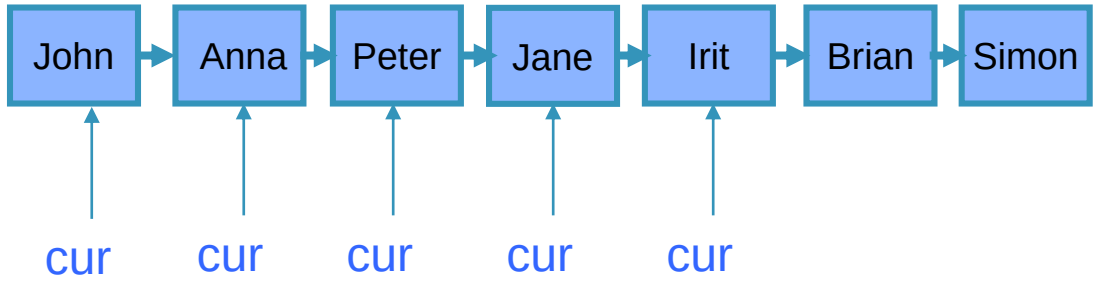
- Leaf node must report 0
- At “null” condition, must report -1




```
int TreeTraversal(BTNode *cur){
    if(cur == NULL)
       return -1;
    int l = TreeTraversal(cur->left);
    int r = TreeTraversal(cur->right);
    int c = max (l, r) + 1;
    return c;
}
```

Sequential Search by a linked list/ array

- Given a linked list of names, how do we check whether a given name(e.g., Irit) is in the list?




```
while (cur!=NULL) {
        if cur->item == "Irit"
            found and stop searching;
        else
            cur = cur->next; }
```

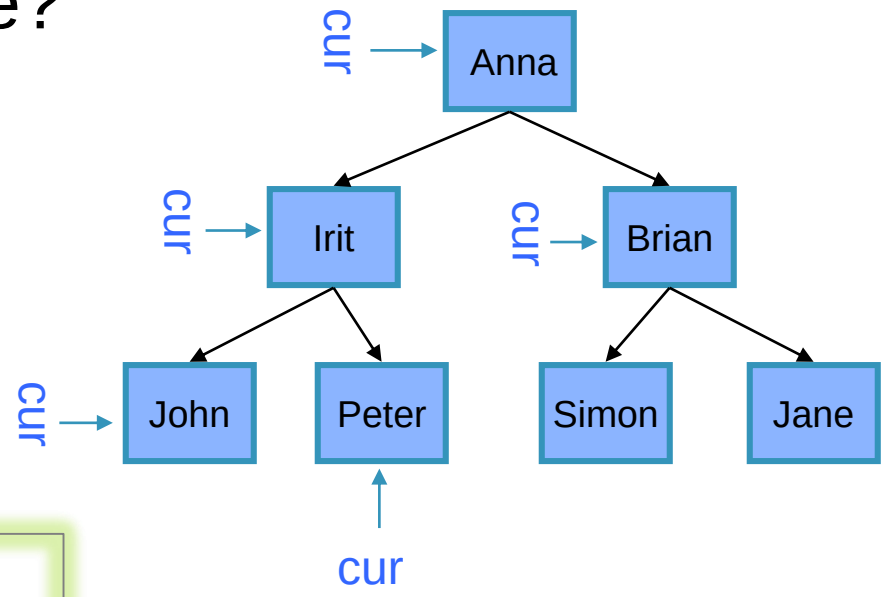
How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

Sequential Search by a binary tree

inefficient

- Given a binary tree of names, how do we check whether a given name(e.g., Brian) is in the tree?

- **Use the TreeTraversal (Pre-order) template, to check every node**




```
TreeTraversal(Node N)
 if N==NULL return;
 if N.item=given_name return;
 TreeTraversal(LeftChild);
 TreeTraversal(RightChild);
 Return;
```

How many nodes are visited during search?
--best case: 1 node (John) => Θ(1)
--worst case: 7 nodes (Simon) => Θ(n)
--avg. case: (1+2+3+…+7)/7=4 nodes => Θ(n)

Summary

- The difference between linked lists and tree structures (linear and non-linear data structures)
- Overview of Tree
- Tree Traversal
  - Depth-First Traversal
    - Pre-order Traversal
    - In-order Traversal
    - Post-order Traversal
  - Breadth-First Traversal: Level-by-level traversal
- Examples

Make sure that you know the difference among them