EE6094
CAD for VLSI Design

Chapter 2
Basic Concept of Data Structures & Algorithms

Spring 2024

Andy, Yu-Guang Chen
Assistant Professor, Department of EE
National Central University
andyygchen@ee.ncu.edu.tw

Outline

- What is Algorithm
- Problem Definition
- Algorithmic Paradigms
- Graphs
- Hierarchical/Multilevel Framework
- Useful Data Structures
- Resources in EDA

Now You Need Algorithms…

- To put devices/interconnects together into VLSI chips
- Fundamental questions: How do you do it smartly?
- Definition of algorithm in a board sense:
  - A step-by-step procedure for solving a problem
- Examples:
  - Cooking a dish
  - Making a phone call
  - Sorting a hand of cards
- Definition of a computational problem:
  - A mathematical object representing a collection of questions that computers might be able to solve
  - A well-defined computational procedure that takes some values as input and produces desired values as output

On Algorithms

- Algorithm: A well-defined procedure for transforming some input to a desired output
- Major concerns:
  - Correctness: Does it halt? Is it correct? Is it stable?
  - Efficiency: Time complexity? Space complexity?
    - Worst case? Average case? (Best case?)
- Better algorithms?
  - How: Faster algorithms? Algorithms with less space requirement?
  - Optimality: Prove that an algorithm is best possible/optimal? Establish a lower bound?
- Applications?
  - Everywhere in computing!

Analysis of Algorithm

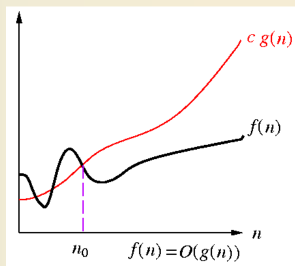
- There can be many different algorithms to solve the same problem
- Need some way to compare 2 algorithms
- Usually the run time is the criteria used
- However, difficult to compare since algorithms may be implemented in different machines, use different languages, etc.
- Also, run time is input-dependent. Which input to use?
- Big-O notation is used

Big-O Notation

- Consider run time for the worst input
  - upper bound on run time
- Express run time as a function input size n
- Interested in the run time for large inputs
- Therefore, interested in the growth rate
- Ignore multiplicative constant
- Ignore lower order terms

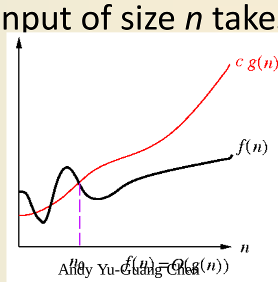
O: Upper Bounding Function

- Def: $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$
  - Examples: $2n^2 + 3n = O(n^2)$, $2n^2 = O(n^3)$, $3n \lg(n) = O(n^2)$
  - Intuition: $f(n)$ "$\le$" $g(n)$ when we ignore constant multiples and small values of n



Big-O Notation

- How to show O (Big-Oh) relationships?
  - $f(n) = O(g(n))$ iff $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = c$ for some $c \ge 0$
- "An algorithm has worst-case running time $O(f(n))$": there is a constant c s.t. for every n big enough, every execution on an input of size n takes at most $cf(n)$ time

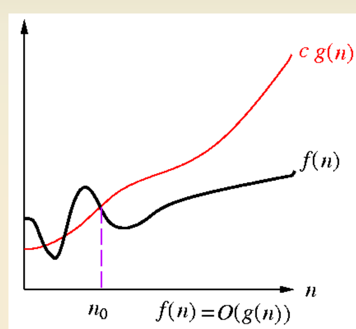


Infinity

- After explaining to a student through various lessons and examples that:
  - $\lim_{x \to 8} (\frac{1}{x-8}) = \infty$
- I tried to check if she really understood that, so I gave her a different example, this was the result:
  - $\lim_{x \to 5} (\frac{1}{x-5}) = 5$

Big-Oh Examples

- Def: $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$



1. $3n^2 + n = O(n^2)$? Yes
2. $3n^2 + n = O(n)$? No
3. $3n^2 + n = O(n^3)$? Yes

   $3n^2 + n \leq cn^2$?
   Take $c = 4$, $n_0 = 1$

- $f(n) = O(g(n))$ implies that $\lim_{n \to \infty} (\frac{f(n)}{g(n)}) = c$ for some $c \geq 0$, if the limit exists

Big-O Notation

- Examples:

  $4n = O(5n)$ [proof: c = 1, n >=1]

  $4n = O(n)$ [proof: c = 4, n >=1]

  $2n^2 = O(n^2)$

  $2n^2 + 3n + 1 = O(n^2)$

  $n^{1.1} + 10000000000n$ is $O(n^{1.1})$

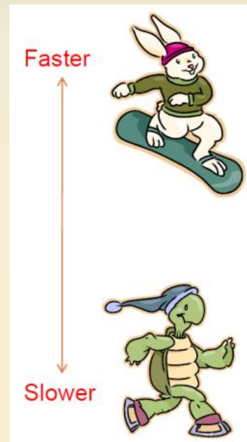
  $n^{1.1} = O(n^2)$

Computational Complexity

- Computational complexity: an abstract measure of the time and space necessary to execute an algorithm as a function of its "input size"
- Scalability with respect to input size is important
  - How does the run time of an algorithm change when the input size doubles?
  - Function of input size *n*
    - Examples: n^2+3n, $2n$, $n\log n$, ...
  - Generally, large input sizes are of interest
    - n > 1,000 or even n > 1,000,000
- Time complexity is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection)
- Space Complexity is expressed in *memory locations* (e.g. bits, bytes, words)

Asymptotic Functions

- Polynomial-time complexity: $O(n^k)$, where n is the input size and k is a constant
- Example polynomial functions:
  - 999: constant
  - $\lg(n)$: logarithmic
  - $\sqrt{n}$: sublinear
  - n: linear
  - $n \lg n$: loglinear
  - n^2: quadratic
  - n^3: cubic
- Example non-polynomial functions
  - 2^n, 3^n: exponential
  - $n!$: factorial



Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

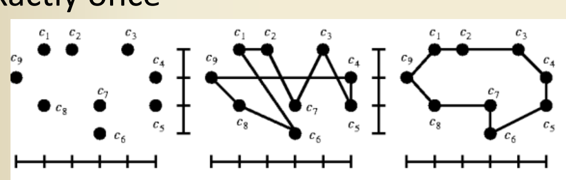
Time	Big-Oh	$n=10$	$n=100$	$n=10^3$	$n=10^6$
500	$O(1)$	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec
$n\log n$	$O(n\log n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$O(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	$O(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^{5} cent.
2^n	$O(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞

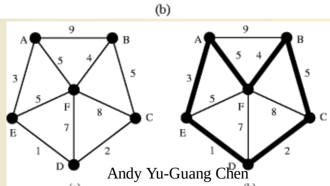
Optimization Problems

- Problem: a general class, e.g., "the shortest-path problem for directed acyclic graphs"
- Instance: a specific case of a problem, e.g., "the shortest-path problem in a specific graph, between two given vertices"
- Optimization problems: those finding a legal configuration such that its cost is minimum (or maximum)
  - MST: Given a graph $G=(V, E)$, find the cost of a minimum spanning tree of G
- An instance $I = (F, c)$ where
  - F is the set of *feasible solutions*, and
  - c is a *cost function*, assigning a cost value to each feasible solution $c : F \rightarrow R$
  - The solution of the optimization problem is the feasible solution with optimal (minimal/maximal) cost
- c.f., Optimal solutions/costs, optimal (exact) algorithms (Attn: optimal $\neq$ exact in the theoretic computer science community)

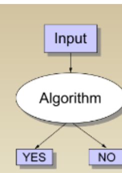
The Traveling Salesman Problem (TSP)

- TSP: Given a set of cities and the distance between each pair of cities, find the distance of a "minimum tour" starts and ends at a given city and visits every city exactly once





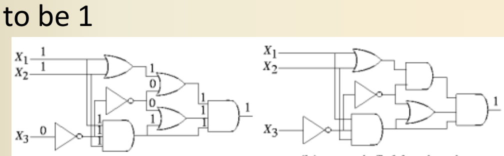
Decision Problem



- Decision problems: problem that can only be answered with "yes" or "no"
  - MST: Given a graph $G=(V, E)$ and a bound K, is there a spanning tree with a cost at most K?
  - TSP: Given a set of cities, the distance between each pair of cities, and a bound B, is there a route that starts and ends at a given city, visits every city exactly once and has total distance at most B?
- A decision problem Π, has instances: $I = (F, c, k)$
  - The set of instances for which the answer is "yes" is given by Y_Π
  - A subtask of a decision problem is *solution checking*: given $f \in F$, checking whether the cost is less than k
- Could apply binary search on decision problems to obtain solutions to optimization problems
- NP-completeness is associated with decision problems

The Circuit-Satisfiability Problem (Circuit-SAT)

- The Circuit-Satisfiability Problem (Circuit-SAT):
  - Instance: A combinational circuit C composed of AND, OR, and NOT gates
  - Question: Is there an assignment of Boolean values to the inputs that make the output of C to be 1?
- A circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit to be 1
  - Circuit (a) is satisfiable since $<x_1, x_2, x_3> = <1, 1, 0>$ makes the output to be 1



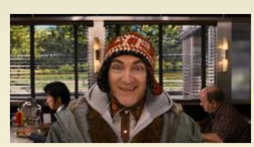
(a) satisfiable circuit (b) unsatisfiable circuit

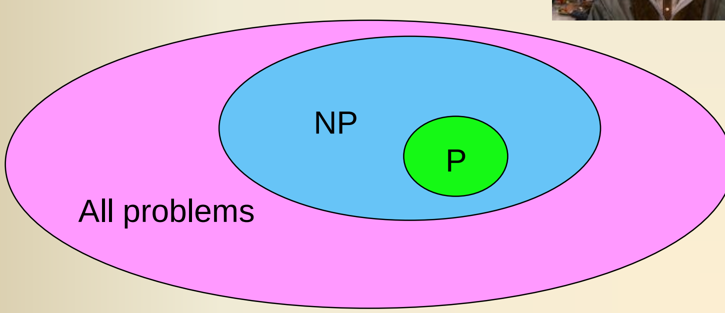
Tractability

- Problems are classified into "easier" and "harder" categories
  - Class P: a polynomial time (in size of input) algorithm is known for the problem (hence, it is a tractable problem)
  - Class NP(non-deterministic polynomial time): a solution is verifiable in polynomial time
  - P ∈ NP. Is P = NP? (Find out and become famous!)
  - Practically, for a problem in NP but not in P: polynomial solution not found yet (probably does not exist)
    - Exact (optimal) solution can be found in exponential time
  - NP-completeness, NP-hardness, etc.
    - Most CAD problems are NP-complete, NP-hard, or worse
    - Be happy with a "reasonably good" solution

NP-class

- Class NP contains those problems that the solution checking can be done in polynomial time
- Address NP problems in 2 stages:
  - make a guess what the solution is
  - check whether the guess is correct





https://read01.com/zh-tw/xABdg7.html#.YiBLsehBxPZ

NP-complete Problems

- A question which is still not answered:

$$P \subset NP \text{ or } P \neq NP$$

- There is a strong belief that $P \neq NP$, due to the existence of NP-complete problems (NPC)
  - All NPC problems have the same degree of difficulty: if one of them could be solved in polynomial time, all of them would have a polynomial time solution.

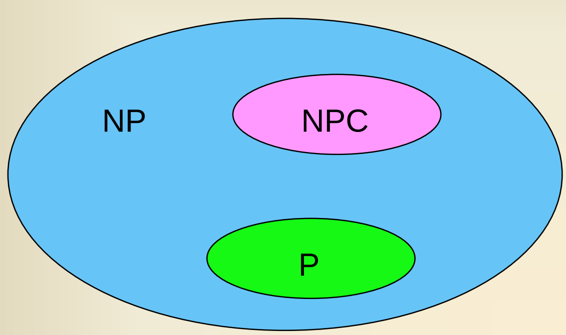
NP-complete Problems

- A problem is NP-complete if and only if
  - It is in NP
  - Some known NP-complete problem can be transformed to it in polynomial time
- Cook's theorem:
  - SATISFIABILITY is NP-complete

Reduction

- Idea: If we can solve problem A, and if problem B can be transformed into an instance of problem A, then we can solve problem B by reducing problem B to problem A and then solve the corresponding problem A.
- Example:
  - Problem A: Sorting
  - Problem B: Given n numbers, find the i-th largest numbers.
  - Polynomial-time Reducible

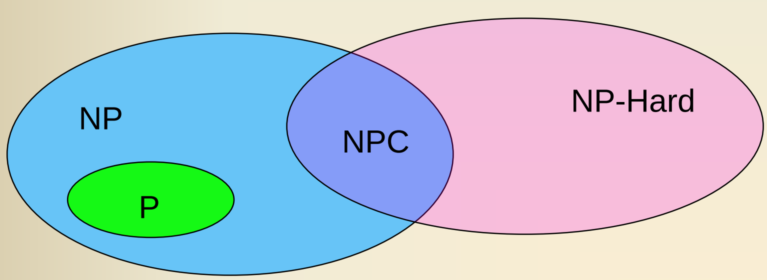
World of NP, Assuming P $\neq$ NP



NP-hard Problems

- Any decision problem (inside or outside of NP) to which we can transform an NP-complete problem to it in polynomial time will have a property that it cannot be solved in polynomial time, unless P = NP
- Such problems are called NP-hard
  - “as hard as the NP-complete problems”

NP-Hard, NP, and NPC



Practical Consequences

- Many problems in CAD for VLSI are NP-complete or NP-hard. Therefore:
  - Exact solutions to such problems can only be found when the problem size is small
  - One should otherwise be satisfied with sub-optimal solutions found by:
    - Approximation algorithms: they can guarantee a solution within e.g. 20% of the optimum
    - Heuristics: nothing can be said a priori about the quality of the solution (experience-based)

Practical Consequences

- Tractable and intractable problems can be very similar:
  - the SHORTEST-PATH problem for undirected graphs is in P
  - the LONGEST-PATH problem for undirected graphs is NP-complete

Brief Summary

- The class NP-complete is a set of problems which we believe there is no polynomial time algorithms
- Therefore, it is a class of hard problems
- NP-hard is another class of problems containing the class NP-complete
- If we know a problem is in NP-complete or NP-hard, there is nearly no hope to solve it efficiently

Algorithmic Paradigms

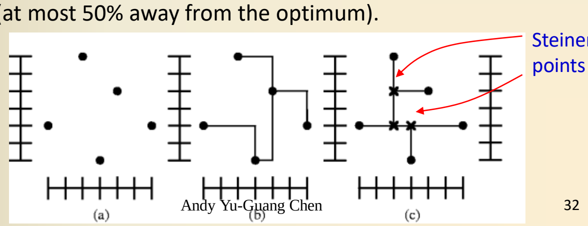
- Exhaustive search: Search the entire solution space
- Branch and bound: Search with pruning
- Greedy: Pick a locally optimal solution at each step
- Dynamic programming: if subproblems are not independent
- Divide-and-conquer (a.k.a. hierarchical): Divide a problem into subproblems (small and similar), solve subproblems, and then combine the solutions of subproblems
- Multilevel: Bottom-up coarsening followed by top-down uncoarsening
- Mathematical programming: Solve an objective function under constraints
- Local search: Move from solution to solution in the search space until a solution deemed optimal is found or a time bound is elapsed
- Probabilistic: Make some choices randomly (or pseudo-randomly)
- Reduction: Transform into a known and optimally solved problem

Algorithm Types

- Algorithms usually used for P problems
  - Exhaustive search
  - Divide-and-conquer (a.k.a. hierarchical)
  - Dynamic programming
  - Greedy
  - Mathematical programming
  - Branch and bound
- Algorithms usually used for NP (but not P) problems (strategy: not seeking "optimal solution", but a "good" one)
  - Approximation
  - Pseudo-polynomial time: polynomial form, but NOT to input size
  - Restriction: restrict the problem to a special case that is in P
  - Exhaustive search/branch and bound
  - Local search: simulated annealing, genetic algorithm, ant colony
  - Heuristics: greedy, … etc.

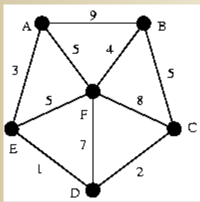
Spanning Tree v.s. Steiner Tree

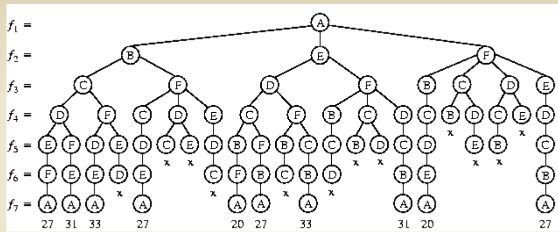
- Manhattan distance: If two points (nodes) are located at coordinates (x_1, y_1) and (x_2, y_2), the Manhattan distance between them is given by $d_{12} = |x_1-x_2| + |y_1-y_2|$.
- Rectilinear spanning tree: a spanning tree that connects its nodes using Manhattan paths (Fig. (b) below).
- Steiner tree: a tree that connects its nodes, and additional points (Steiner points) are permitted to be used for the connections.
- The minimum rectilinear spanning tree problem is in P, while the minimum rectilinear Steiner tree (Fig. (c)) problem is NP-complete.
  - The spanning tree algorithm can be an *approximation* for the Steiner tree problem (at most 50% away from the optimum).



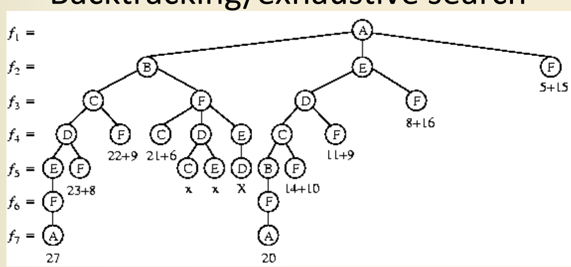
Exhaustive Search v.s. Branch and Bound

- ◆TSP example





Backtracking/exhaustive search



Branch and bound

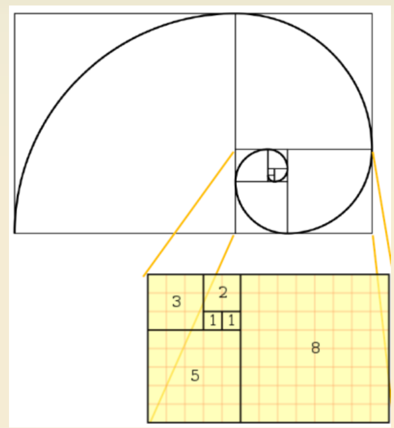
Divide-and-Conquer

- ◆Divide and conquer:
  - ➢ Divide: Recursively break down a problem into two or more sub-problems of the same (or related) type
  - ➢ Conquer: Until these become simple enough to be solved directly
  - ➢ Combine: The solutions to the sub-problems are then combined to give a solution to the original problem
- ◆Correctness: proved by mathematical induction
- ◆Complexity: determined by solving recurrence relations

Example: Fibonacci Sequence

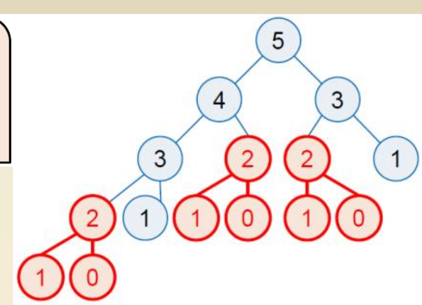
- Recurrence relation: $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$
  - e.g., 0, 1, 1, 2, 3, 5, 8, …

```
fib(n)
1.  if n = 0 return 0
2.  if n = 1 return 1
3.  return fib(n − 1) + fib(n − 2)
```



What's Wrong?

```
fib(n)
1.  if n = 0 return 0
2.  if n = 1 return 1
3.  return fib(n − 1) + fib(n − 2)
```



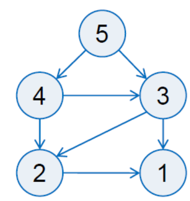
- What if we call fib(5)?
  - fib(5)
  - fib(4) + fib(3)
  - (fib(3) + fib(2)) + (fib(2) + fib(1))
  - ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
  - (((fib(1) + fib(0)) + fib(1))+(fib(1) + fib(0)))+((fib(1) + fib(0))+fib(1))
  - A call tree that calls the function on the same value many different times
    - fib(2) was calculated three times from scratch
    - Impractical for large *n*

Dynamic Programming: Memoization

- Store the values in a table
  - Check the table before a recursive call
  - Top-down!
    - The control flow is almost the same as the original one

fib(*n*)
1. Initialize *f*[0..*n*] with -1 // -1: unfilled
2. *f*[0] = 0; *f*[1] = 1
3. fibonacci(*n*, *f*)

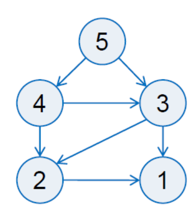
fibonacci(*n*, *f*)
1. **If** *f*[*n*] == -1 **then**
2. *f*[*n*] = fibonacci(*n* − 1, *f*) + fibonacci(*n* − 2, *f*)
3. **return** *f*[*n*] // if *f*[*n*] already exists, directly return



Dynamic Programming: Bottom-up?

- Store the values in a table
  - Bottom-up
    - Compute the values for small problems first
  - Much like induction

fib(*n*)
1. initialize *f*[1..*n*] with -1 // -1: unfilled
2. *f*[0] = 0; *f*[1] = 1
3. **for** *i*=2 **to** *n* **do**
4. *f*[*i*] = *f*[*i*-1]+ *f*[*i*-2]
5. **return** *f*[*n*]



Mathematical Induction

- The first domino falls
- If a domino falls, so will the next domino
- All dominoes will fall!

To see the world in a grain of sand,
And heaven in a wild flower,
Hold infinity in the palm of your hand,
And eternity in an hour.
\- William Blake

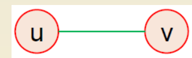
Abstraction

- All about abstraction!
  - Graph: objects and their relationships
  - Vertices: objects
  - Edges: relationships!
- Examples:

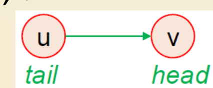
The London Underground Map (a) the 1928 map (b) the 1933 map by H. Beck.

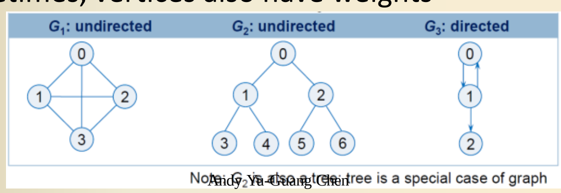
Graphs

- Definition: A graph $G = (V, E)$ consists two sets V and E
  - $V(G)$: a finite nonempty set of vertices (nodes)
  - $E(G)$: a set of edges (links, arcs)



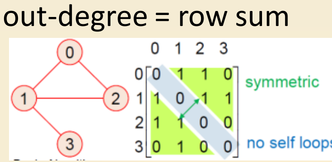
- Undirected graph: undirected edges $(u, v) == (v, u)$
- Directed graph: directed edges $<u, v> \,!=\, <v, u>$
- Weighted graph: weight w: $E \rightarrow R$
  - Sometimes, vertices also have weights

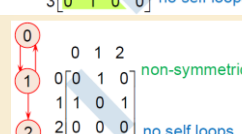




Representation: Adjacency Matrix

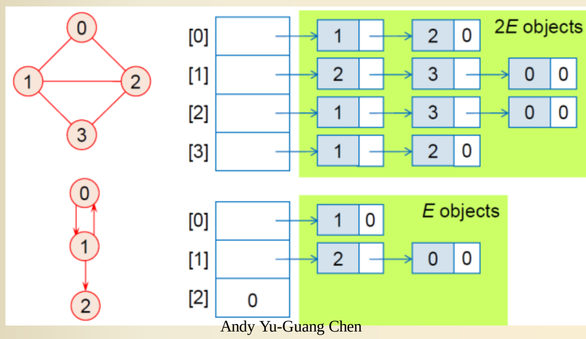
- The adjacency matrix a of G is an $V \times V$ matrix where
  - $a[i][j] = 1$ iff $(u, v) \in E(G)$ (or $<u, v> \in E(G)$)
  - $a[i][j] = 0$, otherwise
- Degree?
  - Undirected: row sum or column sum
  - Directed: in-degree = column sum; out-degree = row sum
- Space: $O(V^2)$
  - Suitable for dense graph
  - How to save space if undirected?
  - What if sparse graph?





Representation: Adjacency List

- The adjacency list is an array of *V* chains, one for each vertex, represents vertices adjacent from it
- Space: $O(V+E)$
  - Good for sparse graph



Edge/Vertex Weights

- Edge weights
  - Usually represent the “cost” of an edge
  - Examples:
    - The delay of a wire in a circuit
    - Distance between two cities
    - Width of a data bus
  - Representation
    - Adjacency matrix: instead of 0/1, keep weight
    - Adjacency list: keep the weight in an additional field of the linked list item
- Vertex weights
  - Usually used to enforce some “capacity” constraint
  - Examples:
    - The size of gates in a circuit
    - The delay of operations in a “data dependency graph”

Graph Traversal

- Purpose: to visit all vertices
- Algorithms
  - Depth-first search
  - Breadth-first search
  - Topological sort
    - Applicable for DAGs
- Example:

From Taipei main station, where can you go by MRT?

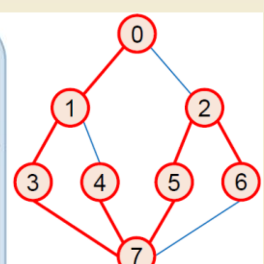
Depth-First-Search (DFS)

- Time complexity:
  - Adjacency list: $O(V+E)$
  - Adjacency matrix: $O(V^2)$

DFS()
1. **foreach** vertex *u* **do**
2. *visited*[*u*] = **false** // initially, all vertices are unvisited
3. *DFSvisit*(*v*) // begin with vertex *v*

Stack-based

DFSVisit(*u*)
// Visit all unvisited vertices reachable from vertex *u*
1. *visited*[*u*] = **true**
2. **foreach** vertex *v* in Adj[*u*] **do**
3. **if** !*visited*[*v*] **then** *DFSVisit*(*v*) // recursive call



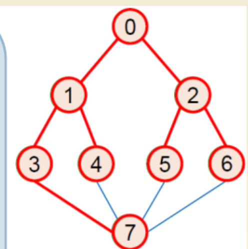
DFS traversal order: 0, 1, 3, 7, 4, 5, 2, 6

Breadth-First-Search (BFS)

- Time complexity:
  - Push each vertex into queue once
  - Adjacency list: $O(V+E)$
  - Adjacency matrix: $O(V^2)$

Queue-based

```
BFS(v)
1.  foreach vertex u do
2.      visited[u] = false // initially, all vertices are unvisited
3.  visited[v] = true; Q = ∅
4.  Enqueue(Q, v) // begin with vertex v
5.  while Q ≠ ∅ do
6.      u = Dequeue(Q)
7.      foreach vertex v in Adj[u] do
8.          if !visited[v] then visited[v] = true
9.              Enqueue(Q, v)
```



BFS traversal order:
0, 1, 2, 3, 4, 5, 6, 7

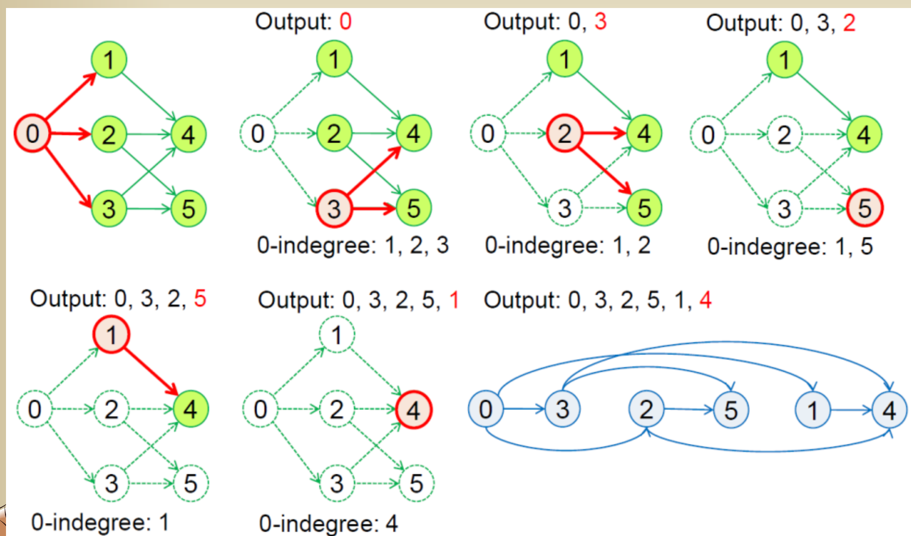
Topological Sort

- Handle precedence constraints
- Time complexity: $O(V+E)$
  - Use a queue/stack to keep "free" vertices
  - 1-4: parse graph once and push free vertices into queue/stack
  - 5-9: pop free vertices from queue/stack one at a time
    - Output it and remove out-going edges
    - Push new free vertices into queue/stack

```
TopologicalSort(G)
1.  compute indegree[v] for each vertex v
2.  Q = ∅; label = 0
3.  foreach vertex v do
4.      if indegree[v] = 0 then Enqueue(Q, v)
5.  while Q ≠ ∅ do
6.      v = Dequeue(Q); label[v] = ++label
7.      foreach vertex u in Adj[v] do
8.          indegree[u] = indegree[u] –1
9.          if indegree[u] = 0 then Enqueue(Q, u)
```

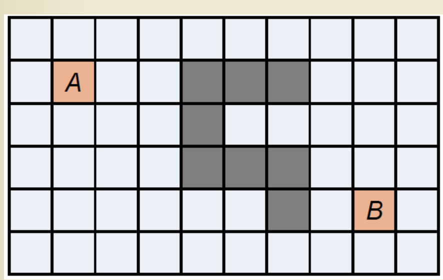
Always can find 0-indegree vertices?

Example: Topological Sorting



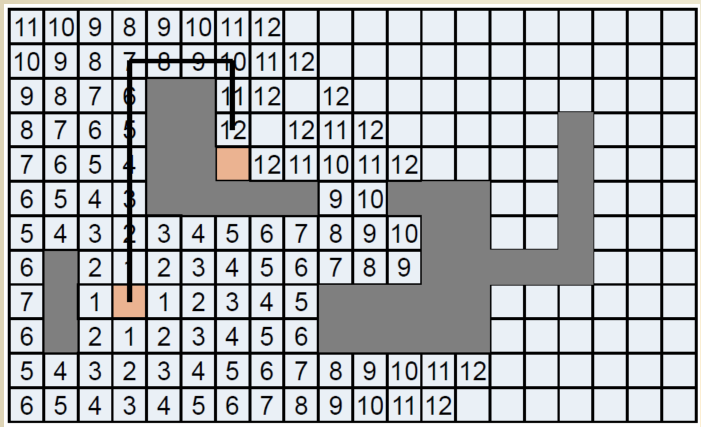
Application: Maze Routing

◆Problem: find a connection from *A* to *B*



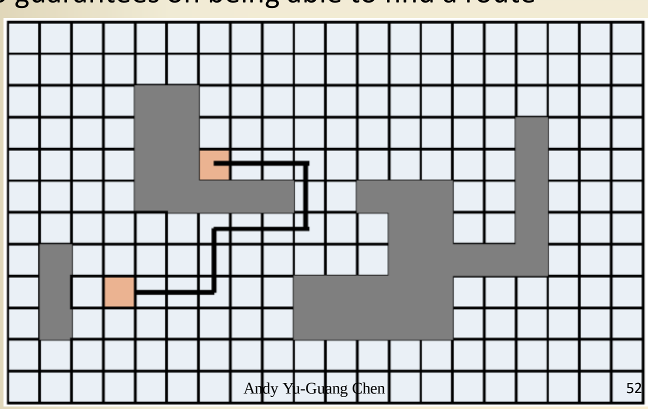
◆Quite useful for two pin nets, rip-up-and-reroute, etc.

BFS: Lee's Algorithm

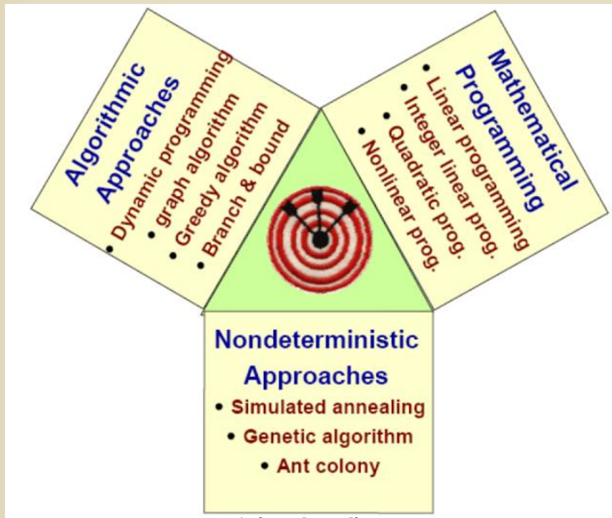


DFS: Line Search Algorithms

- Reduce memory requirement
  - Send out a line and use it to guide the search
  - No guarantees on being able to find a route

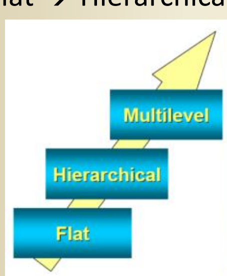


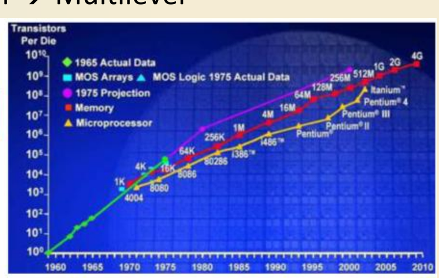
Classifications of Popular EDA Algorithms



Framework Evolution

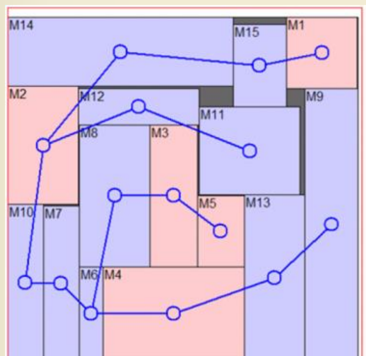
- ◆Billions of transistors may be fabricated in a single chip for nanometer technology
- ◆Need frameworks for very large-scale designs
- ◆Framework evolution for EDA tools:
  - ➢ Flat → Hierarchical → Multilevel





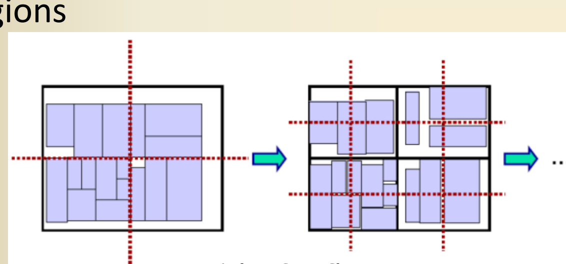
Flat Framework

- Process the circuit components in the whole chip
- Limitation: Good for small-scale designs, but hard to handle larger problems directly



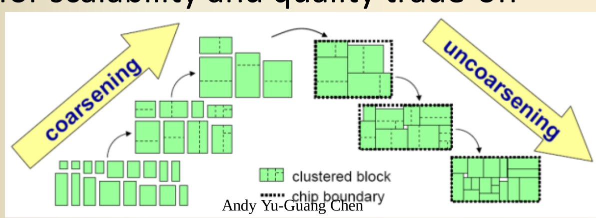
Hierarchical Framework

- The hierarchical approach recursively divides a circuit region into a set of subregions and solve those subproblems independently
- Good for scalability for large-scale design, but lack the global information for the interaction among subregions



Multilevel Framework

- **Bottom-up** coarsening (clustering)
  - Iteratively groups a set of circuit components & collects global information
- **Top-down** uncoarsening (declustering)
  - Iteratively ungroups clustered components & refines the solution
- Good for scalability and quality trade-off



Useful Data Structures

- Struct
- Linked lists
- Stacks
- Queues
- Tress
- Graph
- Standard Template Library (STL)

Structure Definitions

- Structures are aggregate data types—that is, they can be built using elements of several types including other structs.
- Example

```
struct card {
    char *face;
    char *suit;
 };
```

  - `struct` introduces the definition for structure `card`
  - `card` is the structure tag and is used to declare variables of the structure type
  - `card` contains two members of type `char *`
    - These members are `face` and `suit`

Structure Definitions

- Another Example

```
struct employee{
     char firstName[20];
     char lastName[20];
     int age;
     char gender;
     double hourlySalary;
};
```

- Members of the same structure must have unique names, but two different structures may contain members of the same name without conflict.
- Each structure definition must end with a semicolon.

Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables
- Structure variables
  - Defined like other variables:

```
struct card oneCard, deck[ 52 ], *cPtr;
```

  - Can be defined together with a structure definition:

```
struct card {
    char *face;
    char *suit;
} oneCard, deck[ 52 ], *cPtr;
```


Structure Definitions

- Accessing structure members
  - Dot operator (`.`) used with structure variables

```
struct card myCard;
cout << myCard.suit;
```

  - Arrow operator (`->`) used with pointers to structure variables

```
struct card *myCardPtr = &myCard;
cout<< myCardPtr->suit;
```

  - `myCardPtr->suit` is equivalent to `( *myCardPtr ).suit`

Structure Definitions

```cpp
#include <iostream>

using namespace std;

struct card{
    char *face;
    char *suit;
};


int main()
{
    card a;
    card *aptr;

    a.face="Ace";
    a.suit="Spades";

    aptr=&a;

    cout<<a.face<<" of "<<a.suit<<endl;
    cout<<aptr->face<<" of "<<aptr->suit<<endl;
    cout<<(*aptr).face<<" of "<<(*aptr).suit<<endl;

    return 0;
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

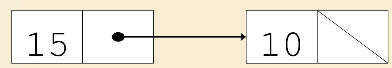

Dynamic Memory Management

- We've studied fixed-size data structures such as one-dimensional arrays and two-dimensional arrays.
- This chapter introduces dynamic data structures that grow and shrink during execution.
- Linked lists are collections of data items logically "lined up in a row"—insertions and removals are made anywhere in a linked list.
- Stacks are important in compilers and operating systems: Insertions and removals are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue and removals are made from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file-system directories and compilation of expressions into machine language.

Self-Referential Structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a `NULL` pointer (**0**)
- Example

```
struct node {
    int data;
    struct node *nextPtr;
}
```



  - `nextPtr`
    - Points to an object of type node
    - Referred to as a link
      - Ties one node to another node

Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes.
- When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future.
- The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system.
- Often, the limits are much smaller, because available memory must be shared among many programs.

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer with a cast
    - If no memory available, returns `NULL`
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```

- `free`
  - Deallocates memory allocated by `malloc`
  - Takes a pointer as an argument
  - `free ( newPtr );`

Dynamic Memory Allocation

- The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type.
- For example, the following statement allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the new `Node`'s address to `newPtr`.

```
// create Node with data 10
Node *newPtr = new Node( 10 );
```

- If no memory is available, `new` throws a `bad_alloc` exception.
- The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`—the memory is returned to the system so that the memory can be reallocated in the future.

Dynamic Memory Allocation

- To free memory dynamically allocated by the preceding `new`, use the statement
  - `delete newPtr;`
- Note that `newPtr` itself is not deleted; rather the space `newPtr` points to is deleted.
- If pointer `newPtr` has the null pointer value `0`, the preceding statement has no effect.

Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links—hence, the term “linked” list.
- A linked list is accessed via a pointer to the list’s first node.
- Each subsequent node is accessed via the link-pointer member stored in the previous node.
- By convention, the link pointer in the last node of a list is set to null (`0`) to mark the end of the list.
- Data is stored in a linked list dynamically—each node is created as necessary.
- A node can contain data of any type, including objects of other classes.

Linked Lists

- Lists of data can be stored in arrays, but linked lists provide several advantages.
- A linked list is appropriate when the number of data elements to be represented at one time is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- The size of a “conventional” C++ array, however, cannot be altered, because the array size is fixed at compile time.
- “Conventional” arrays can become full.
- Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked Lists

- Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
- Existing list elements do not need to be moved.
- Pointers merely need to be updated to point to the correct node.
- Linked-list nodes are not stored contiguously in memory, but logically they appear to be contiguous.

Linked Lists

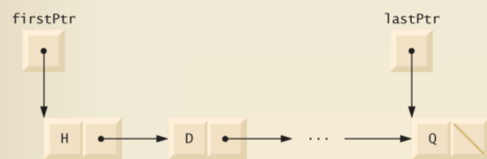


Fig. 20.2 | A graphical representation of a list.

Stacks

- A stack data structure allows nodes to be added to the stack and removed from the stack only at the top.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- One way to implement a stack is as a constrained version of a linked list.
- In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.

Stacks

- The primary member functions used to manipulate a stack are `push` and `pop`.
- Function `push` inserts a new node at the top of the stack.
- Function `pop` removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

Andy Yu-Guang Chen 75

Stacks

- Stacks have many interesting applications.
- For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack.
- If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller.
- Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

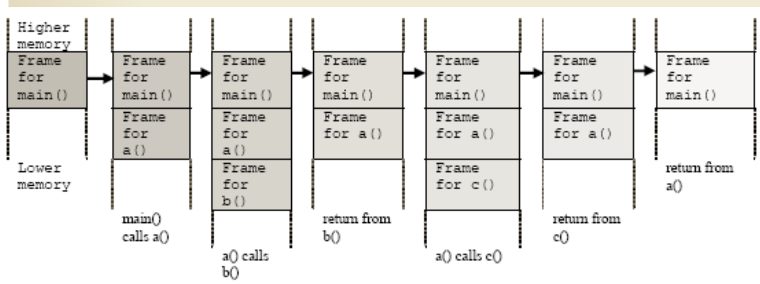
Andy Yu-Guang Chen 76

Stacks

- Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function.
- When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.
- Stacks are used by compilers in the process of evaluating expressions and generating machine-language code.

Stacks

```
#include <stdio.h>

int a();
int b();
int c();

int a()
{
  b();
  c();
  return 0;
}

int b()
{ return 0; }

int c()
{ return 0; }

int main()
{
  a();
  return 0;
}
```



Stacks

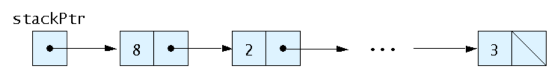


Fig. 12.7 Graphical representation of a stack.

Queues

- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
- For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The insert and remove operations are known as enqueue and dequeue.

Queues

- Queues have many applications in computer systems.
- Computers that have a single processor can service only one user at a time.
- Entries for the other users are placed in a queue.
- Each entry gradually advances to the front of the queue as users receive service.
- The entry at the front of the queue is the next to receive service.

Queues

- Queues are also used to support print spooling.
- For example, a single printer might be shared by all users of a network.
- Many users can send print jobs to the printer, even when the printer is already busy.
- These print jobs are placed in a queue until the printer becomes available.
- A program called a spooler manages the queue to ensure that, as each print job completes, the next print job is sent to the printer.

Queues

- Information packets also wait in queues in computer networks.
- Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet’s final destination.
- The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- A file server in a computer network handles file access requests from many clients throughout the network.
- Servers have a limited capacity to service requests from clients.
- When that capacity is exceeded, client requests wait in queues.

Queues

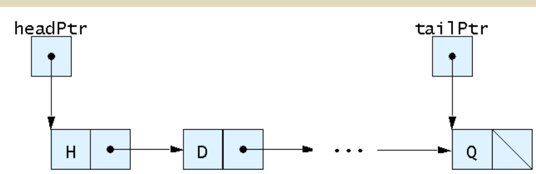
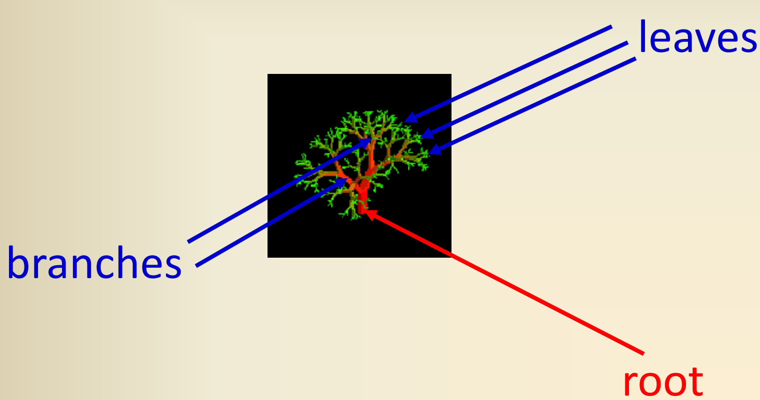


Fig. 12.12 A graphical representation of a queue.

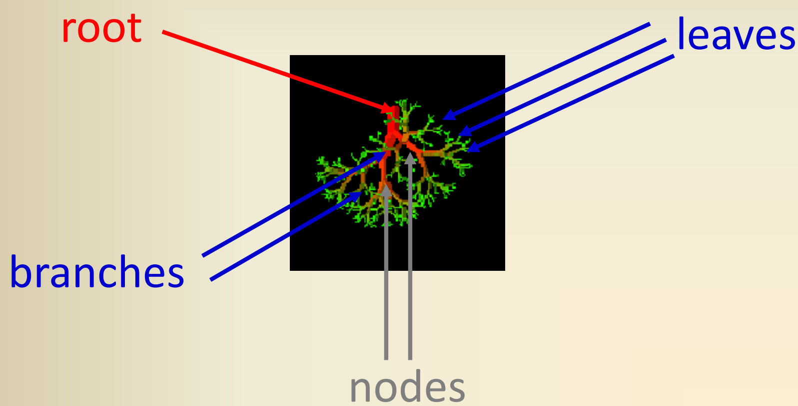
Trees

- Linked lists, stacks and queues are linear data structures.
- A tree is a nonlinear, two-dimensional data structure.
- Tree nodes contain two or more links.

Nature Lover’s View of A Tree

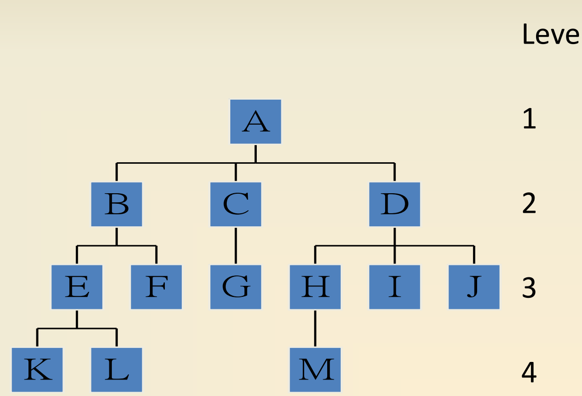


Computer Scientist's View



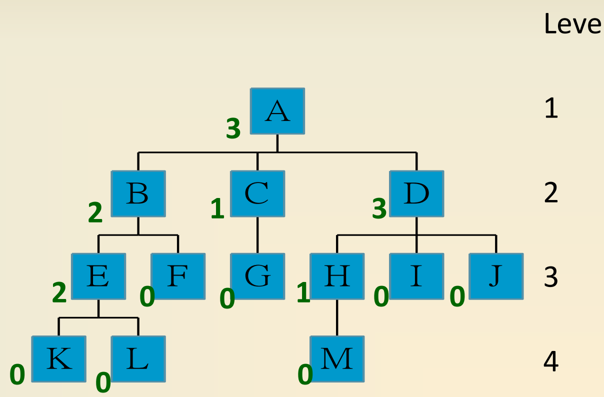
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



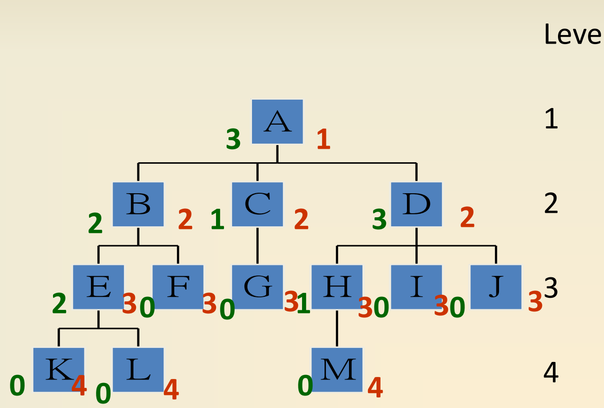
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



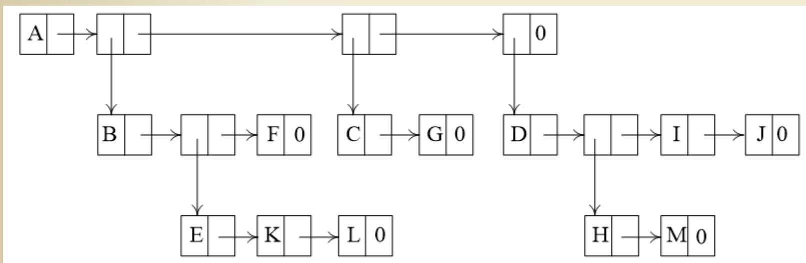
Level and Depth

- Node: 13
- degree of a node
- leaf (terminal)
- nonterminal
- parent
- children
- sibling
- degree of a tree: 3
- ancestor
- level of a node
- height of a tree: 4



Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees



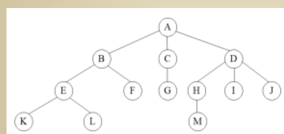
Representation of Trees

- List Representation
  - Possible node structure

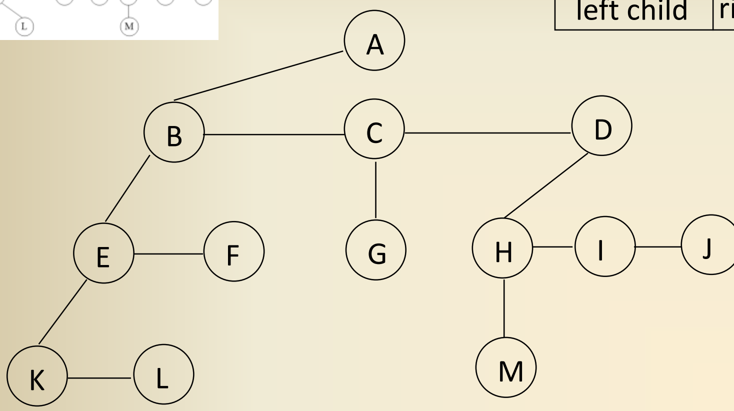
data	link 1	link 2	...	link n

How many link fields are needed in such a representation?

Left Child - Right Sibling



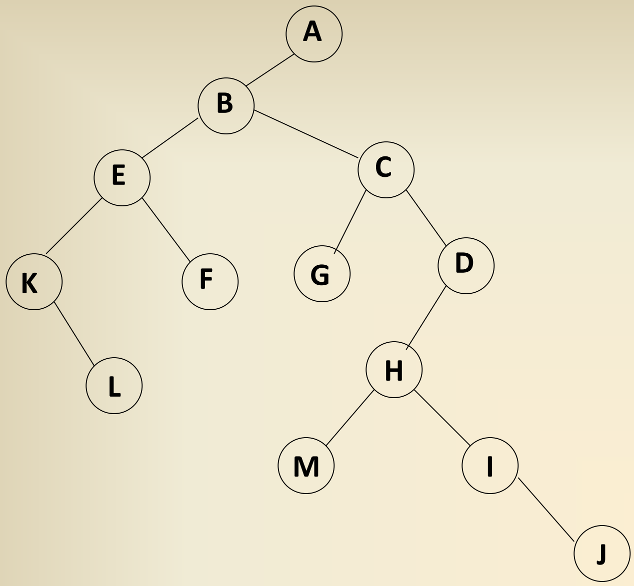
data	
left child	right sibling



Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

Binary Tree



Binary Tree

- This section discusses binary trees (Fig. 20.18)—trees whose nodes all contain two links (none, one or both of which may be null).
- For this discussion, refer to nodes A, B, C and D in Fig. 20.18.
- The root node (node B) is the first node in a tree.
- Each link in the root node refers to a child (nodes A and D).
- The left child (node A) is the root node of the left subtree (which contains only node A), and the right child (node D) is the root node of the right subtree (which contains nodes D and C).
- The children of a given node are called siblings (e.g., nodes A and D are siblings).
- A node with no children is a leaf node (e.g., nodes A and C are leaf nodes).

Binary Tree

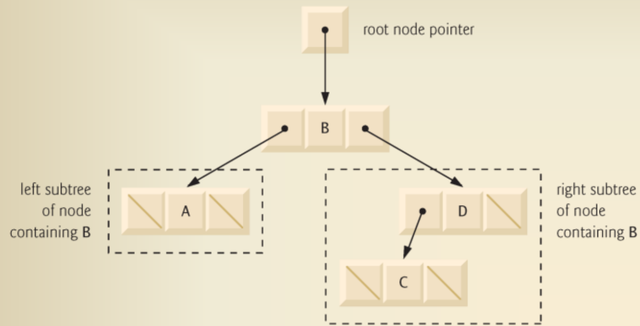
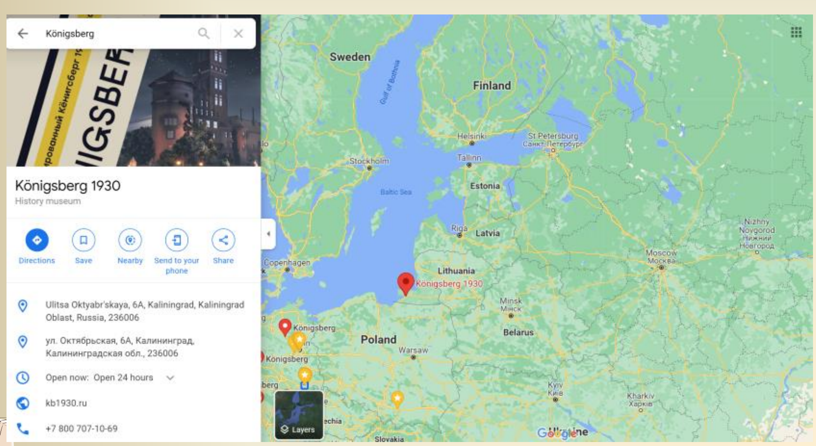


Fig. 20.18 | A graphical representation of a binary tree.

Andy Yu-Guang Chen 97

The Graph ADT

- Introduction
  - A graph problem example: Köenigsberg bridge problem



Andy Yu-Guang Chen 98

The Graph ADT

◆ Introduction

➢ A graph problem example: Köenigsberg bridge problem

The Graph ADT

◆ Introduction

➢ A graph problem example: Köenigsberg bridge problem

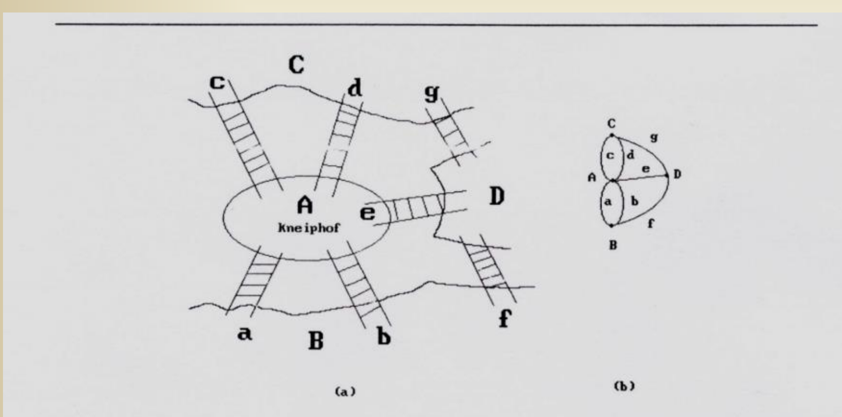


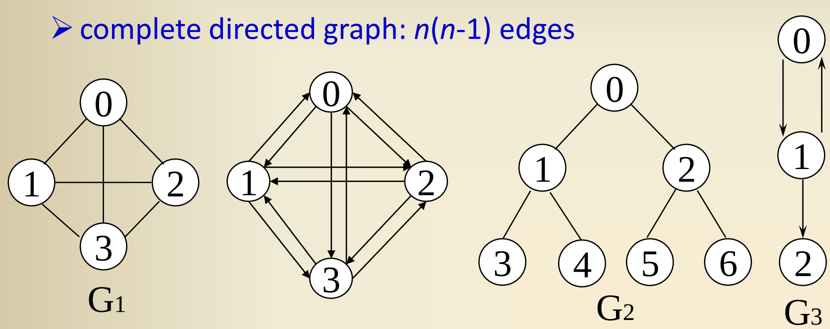
Figure 6.1: The bridges of Koenigsberg

The Graph ADT

- **Definitions**
  - A graph G consists of two sets
    - a finite, nonempty set of vertices $V(G)$
    - a finite, possible empty set of edges $E(G)$
  - $G(V,E)$ represents a graph
  - An undirected graph is one in which the pair of vertices in an edge is unordered, $(v_0, v_1) = (v_1, v_0)$
  - A directed graph is one in which each edge is a directed pair of vertices, $<v_0, v_1> \; != \; <v_1, v_0>$
  - tail $\longrightarrow$ head

The Graph ADT

- **Examples for Graph**
  - complete undirected graph: $n(n-1)/2$ edges
  - complete directed graph: $n(n-1)$ edges



complete graph

incomplete graph

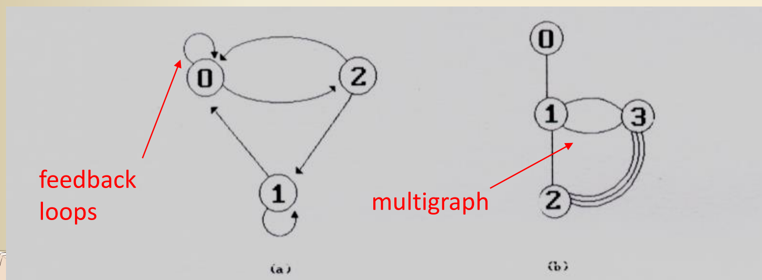
$V(G_1)=\{0,1,2,3\}$ $\quad E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

$V(G_2)=\{0,1,2,3,4,5,6\}$ $\quad E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

$V(G_3)=\{0,1,2\}$ $\quad E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

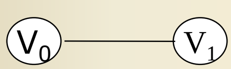
The Graph ADT

- Restrictions on graphs
  - A graph may not have an edge from a vertex, *i*, back to itself. Such edges are known as *self loops*
  - A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data referred to as a multigraph

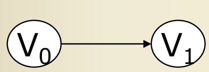


The Graph ADT

- Adjacent and Incident
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are adjacent(相鄰)
  - The edge (v_0, v_1) is incident(附著相鄰) on vertices v_0 and v_1

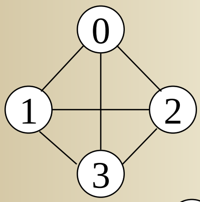


- If <v_0, v_1> is an edge in a directed graph
  - v_0 is adjacent to v_1, and v_1 is adjacent from v_0
  - The edge <v_0, v_1> is incident on v_0 and v_1

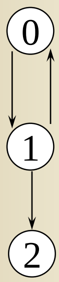


The Graph ADT

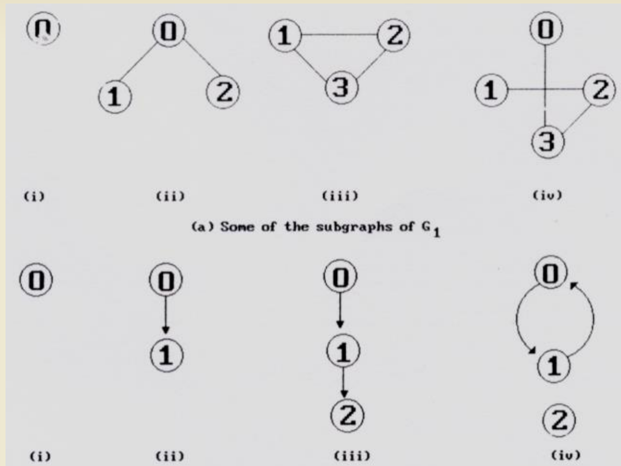
- A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



G1

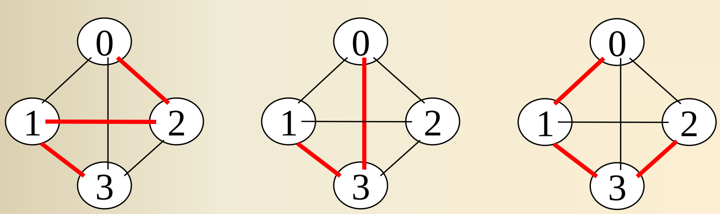


G3



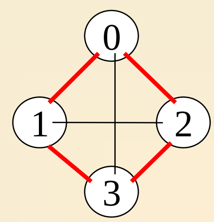
The Graph ADT

- Path
  - A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, $v_p, v_{i_1}, v_{i_2}, ..., v_{i_n}, v_q$, such that $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), ..., (v_{i_n}, v_q)$ are edges in an undirected graph.
    - A path such as (0, 2), (2, 1), (1, 3) is also written as 0, 2, 1, 3
  - The length of a path is the number of edges on it



The Graph ADT

- Simple path and cycle
  - simple path (simple directed path): a path in which all vertices, except possibly the first and the last, are distinct.
  - A cycle is a simple path in which the first and the last vertices are the same.



Standard Template Library (STL)

- We've repeatedly emphasized the importance of software reuse.
- Recognizing that many data structures and algorithms are commonly used, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library.
- The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures.

Introduction to Containers

Standard Library container class	Description
Sequence containers	
`vector`	Rapid insertions and deletions at back. Direct access to any element.
`deque`	Rapid insertions and deletions at front or back. Direct access to any element.
`list`	Doubly linked list, rapid insertion and deletion anywhere.
Associative containers	
`set`	Rapid lookup, no duplicates allowed.
`multiset`	Rapid lookup, duplicates allowed.
`map`	One-to-one mapping, no duplicates allowed, rapid key-based lookup.
`multimap`	One-to-many mapping, duplicates allowed, rapid key-based lookup.

Fig. 21.1 | Standard Library container classes. (Part 1 of 2.)

Introduction to Containers

Standard Library container class	Description
Container adapters	
`stack`	Last-in, first-out (LIFO).
`queue`	First-in, first-out (FIFO).
`priority_queue`	Highest-priority element is always the first element out.

Fig. 21.1 | Standard Library container classes. (Part 2 of 2.)

Introduction to Algorithms

Mutating-sequence algorithms			
copy	partition	replace_copy	stable_partition
copy_backward	random_shuffle	replace_copy_if	swap
fill	remove	replace_if	swap_ranges
fill_n	remove_copy	reverse	transform
generate	remove_copy_if	reverse_copy	unique
generate_n	remove_if	rotate	unique_copy
iter_swap	replace	rotate_copy	

Fig. 21.11 | Mutating-sequence algorithms.

CAD Related Conferences/Journals

- Important Conferences:
  - ACM/IEEE Design Automation Conference (DAC)
  - IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)
  - IEEE Int'l Test Conference (ITC)
  - ACM Int'l Symposium on Physical Design (ISPD)
  - ACM/IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC)
  - ACM/IEEE Design, Automation, and Test in Europe (DATE)
  - IEEE Int'l Conference on Computer Design (ICCD)
  - IEEE Custom Integrated Circuits Conference (CICC)
  - IEEE Int'l Symposium on Circuits and Systems (ISCAS)
  - Others: VLSI Design/CAD Symposium/Taiwan
- Important Journals:
  - IEEE Transactions on Computer-Aided Design (TCAD)
  - ACM Transactions on Design Automation of Electronic Systems (TODAES)
  - IEEE Transactions on VLSI Systems (TVLSI)
  - IEEE Transactions on Computers (TC)
  - IEE Proceedings – Circuits, Devices and Systems
  - IEE Proceedings – Digital Systems
  - INTEGRATION: The VLSI Journal

Q&A

Andy, Yu-Guang Chen
Assistant Professor, Department of EE, NCU
Email: andyygchen@ee.ncu.edu.tw
FB: Yu-Guang Chen
IG: ncu.eda.andy
Google account: andyygchen.ncu