Outcome (2): Introduction to data structures

An Introduction to Data Structure

Terms: Data Types, Data Structures, Abstract Data Type (ADT)

Data Types

- provided by a programming language
- a set of values + a set of operations on these values
- e.g. int (range of possible values & operations such as +,-,*,/)

Abstract Data Types (ADT)

- define what data; what operations required (No need to talk about how the data is stored and how each operation is done.)
- based on applications, but try to be as generic as possible
- capture "requirements"

Some examples

e.g. Student record systems
Data: Student info
Operations:
Add new records:
Assign a new student # and insert the corresponding student record into the system
Search records: search the record by student number
Delete old records: ....

*Can be handled by the same ADT if the ADT is general enough!

e.g. Employee record systems
Data: Employee info
Operations:
Add new records:
Assign a new employee # and insert the corresponding employee record into the system
Search records: search the record by employee id
Delete old records: ....

ADT is at the abstract level, so we have to implement it

Data Structures

- to realize ADT, we use data structures, which are collections of variables, possibly of several different data types, connected in various ways

e.g. Student record systems

Data: Student info

Operations:

Add new records

Search records

Delete old records

Data structure

(1) How data is stored and organized?

Two arrays:

- Stud_num; Stud_other_info;
- Same entry in arrays => same student
- Stored in arbitrary order

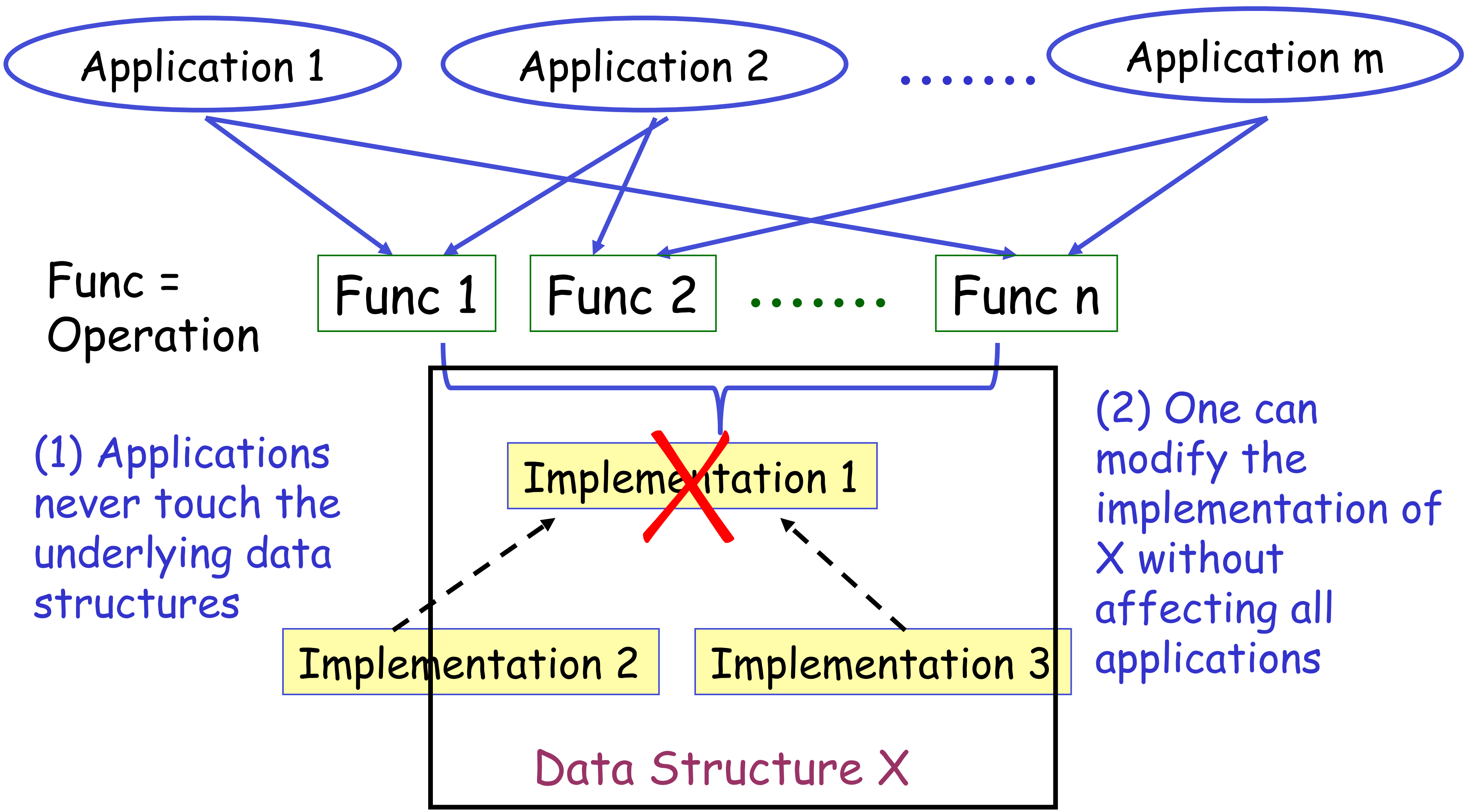
Remark 1:
Applicable to the employee case.

(2) Design algorithm for each operation

```
Search(x) {    // x is the given student number
        for i = 1 to n     // n: total no. of students
            if Stud_num[i] = x
                return Stud_info[i];
        Report "x not found"
}
```

Remark 2:
Change how data is stored => changes algorithms for operations (e.g. store in increasing order of stud #)

Advantage of using ADT and separate it from the implementation:



Two concerns when designing data structures

Example:

Write a program to play chess

Data:

Chess (total #: m); Chess chessboard (nxn)

Operations:

Empty(x,y) – whether the cell (x,y) is not occupied by any chess

Move (c, x, y) – Move chess c to the cell (x,y)

…….

Issue 1: whether operations are supported efficiently

Trade-off?

Issue 2: storage required by the data structure (whether it is space efficient)

2D Array implementation:

e.g. CB[x,y] = -1 if empty, can support operation efficiently: O(1) time

A single array implementation:

Only store occupied cells, store (chess, x, y) in a list

Slower: O(m) time

How about storage? $O(n^2)$ O(m)

[Note: no need to compare bits/bytes at this level, count number of data items and use asymptotic notation]

Basic data structures

List, Set, Stack, Queue, Graph, Tree

(1) List - ADT

Data:

A set of data items arranged in sequence (no special property).

Operations:

Create(L) - Create an empty list L.

Search(L, x) - Return an index to the element x or return -1 if x is not in L.

Insert(L, x, i) – Insert the item x in L at position (i+1).

Delete(L, i) – Delete the item in L at position i (assuming # of elements in L ≥ i).

Data structure for List (one of the implementations)

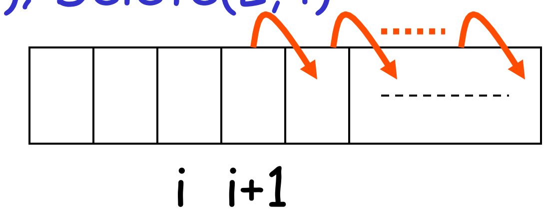
Data

Use an array to store the data items (in arbitrary order).

Use a variable "length" to store the number of items in the list.

Algorithms for each operation

Create(L); Search(L, x); Insert(L, x, i); Delete(L, i)




```
struct List{
   int length;
   element entry[Max_Len];
};
List L;
```

```
void Insert(L, x, i) {
   L.length ++;
   for k = L.length downto i+2 do
         L.entry[k] = L.entry[k-1];
    L.entry[i+1] = x;
}
```

Note: need to check overflow!

```
void Create(L) {
    L.length = 0;
}
```

$\Theta(1)$

```
int Search(L, x) {
    for i = 1 to L.length do
         if (L.entry[i] = x) return i;
    return -1;
}
```

Worst time: $\Theta(n)$; // n = L.length

how about best case, average case?

Running time: $\Theta(n-i)$

Worst time: $\Theta(n)$;

how about best case, average case?

Similarly for Delete(L,i)

Running time: $\Theta(n-i)$

Worst time: $\Theta(n)$;

how about best case, average case?

Variations

(1) Redefine Insert(L, x, i) as Insert(L, x) - insert item x on L (where it is inserted is not important).

```
void Insert(L, x, i) {
    for k = L.length downto i+1 do
        L.entry[k] = L.entry[k-1];
    L.length ++;
    L.entry[i+1] = x;
}
```

$\Theta(n)$

⇨

```
void Insert(L, x) {
    L.length ++;
    L.entry[L.length] = x;
}
```

$\Theta(1)$

(2) Use an array to store the data items (in sorted order).

```
int Search(L, x) {
    for i = 0 to L.length-1 do
        if (L.entry[i] = x) return i;
    return -1;
}
```

$\Theta(n)$

⇨ Binary search

Taught later if you haven't learned it

$\Theta(\log n)$

For "Set", refer to the MIT book.

(2) Stack

Application:

Write a program to parse the followings to check if the parentheses are balanced or not (useful in compilers).

Examples:

(A × ((B + C))	No
A / ((B × (B – C)))	Yes
If (A > 0) and (C = 8))	No

Since the expression (variables etc.) inside parentheses are not important for this simplified problem (note: it is important for the compiler), we can assume the input as:

Examples:

((())

((()))

() ())

Q: any idea to solve the problem?

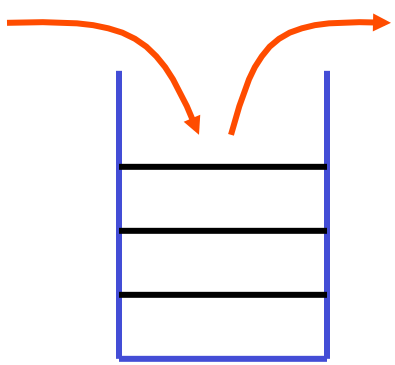
Hint: which "(" should a ")" corresponds to?

(2) Stack –
A set of data items arranged in a sequence [special requirement: element to be deleted from the set is the one most recently inserted (Last-in, first-out or LIFO)]

In general, stack is useful when you need to retrieve stored data in reversed order as it's inserted.

Operations:
Empty(S): return true if the stack S is empty and false otherwise
Top(S): return the element at the top of stack S; (different from the MIT book!)
Push(S, x): insert x to the top of stack S
Pop(S): return and then delete the element at the top of stack S

Insert and delete always done at the beginning of the list (top)



Implementation: Array
- A single array to store items; index points to the top of stack

Operations:
Empty(S)
Top(S)
Push(S, x)
Pop(S)

Example (array implementation):

```
struct stack{
    int index = -1 // empty stack
    element entry[max]
}
stack S;

boolean Empty(S) {
    if (S.index = -1) return true;
    else return false;                    O(1)
}

element Top(S) {
    if (Empty(S))) print("underflow")
    else return S.entry[S.index];         O(1)
}
```

```
void Push(S, x) {
    S.index ++
    S.entry[S.index] = x;
}
```

O(1)

Should check error condition: Overflow

```
element Pop(S) {
    x = Top(S)
    S.index --
    return x
}
```

O(1)

```
boolean Full(S) {
    if (S.index = max) return true
    else return false
}
```

```
void Push(S, x) {
    if Full(S) print "overflow"
    else
        S.index ++
        S.entry[S.index] = x
}
```

Running time = O(1)

In general, stack is useful when you need to retrieve stored data in reversed order as it's inserted.

An example application of stack: balancing the parenthesis

Examples:

( ), (( )), ( ) ( ), (( ), (( ) ( ), ( )) (

Why stack is useful?

Idea: when you see "(", push in stack;
when you see ")", pop out stack
if empty stack, error
if the input is exhausted, but stack not empty, error

```
boolean Par_matching( ){
    stack S;
    while not end of input do{
        read ch
        if (ch = "(") push(S,ch)
        else {
          if Empty(S) return false
            pop(S) }
        }
    if (not (Empty(S))) return false
    return true
}
```

What is the time complexity?

O(n) where n- length of input

Can you handle more than one type of parenthesis, e.g.

({ } [( )]) , ({ (})

```
boolean Par_matching( ){
    stack S;
    while not end of input do{
        read ch
        if (ch = "(") push(S,ch))
        else {
            if Empty(S) return false
            pop(S)}
    if (not (Empty(S))) return false
    return true
}
```

Remark:
Independent of implementation of ADT.

Libraries:

Stack
Queue
Set
List
.....

Function interfaces ⟷ Application Program

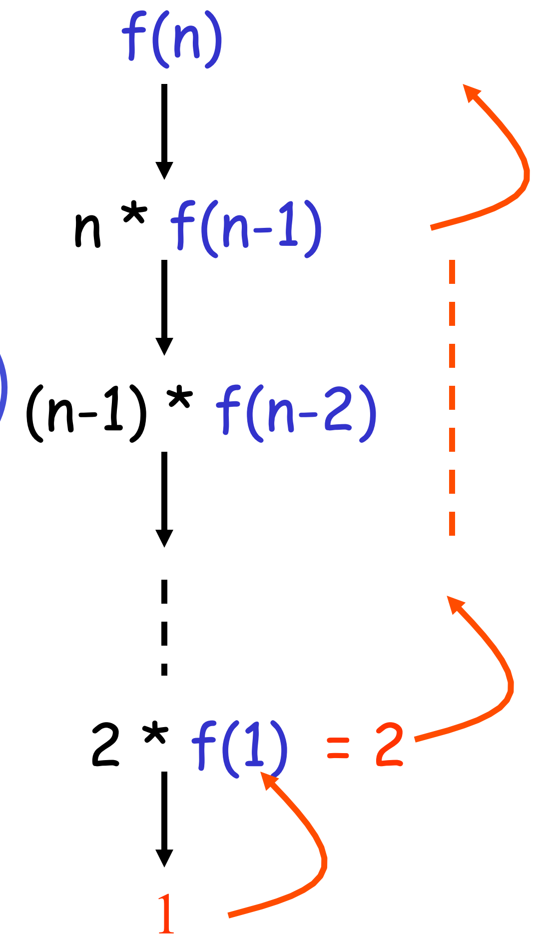
* Implementation independent

Stack and recursion

Example:

```
fact(n){
    if (n = 1) return 1;
    else return n*fact(n-1);
}
```

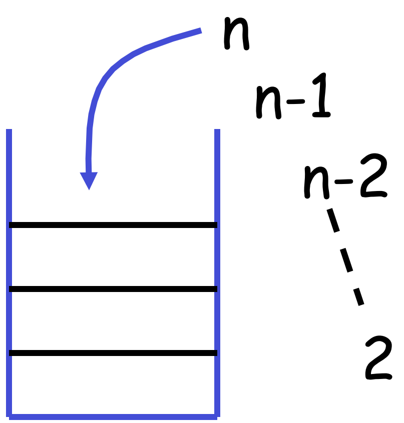
Fit for stack



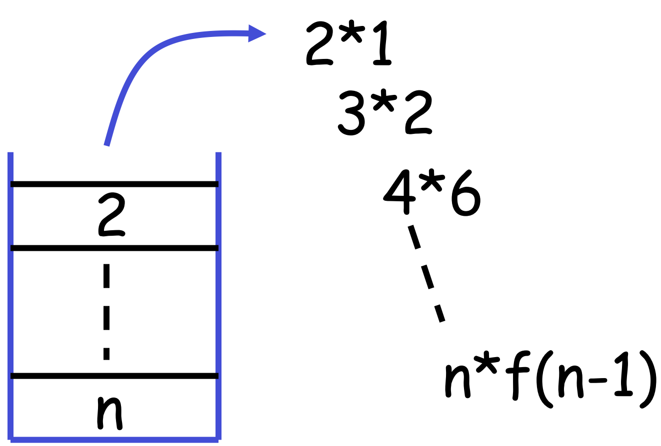
Multiplication is calculated in reverse order as the call is made

Simulate recursion by a stack:

First round



Second round



Rewrite the procedure to eliminate the recursion using a stack:

```
fact(n){
    if (n = 1) return 1;
    for (i = n downto 2)     // simulate the recursive call
        push(S, i);
    ans = 1;
    for (i = 2 to n)         // simulate traceback from
        ans = ans*pop(S);    // recursive call
    return ans;
}
```

In fact, stacks are actually used in the implementation of recursive procedures in the programming languages (an important application).

Conceptual view:

```
Alg(parameters) {
    S1;
    .....
    Sm;
    Alg(a different set of paramters);
    Sm+1;
    ....
    Sn;
}
```

Right before the call is executed, info about local variables, where to return (activation record) is pushed in a stack.

The same function is called using the new set of parameters.

Activation records will be popped in LIFO manner to continue the execution of other statements.

Example:

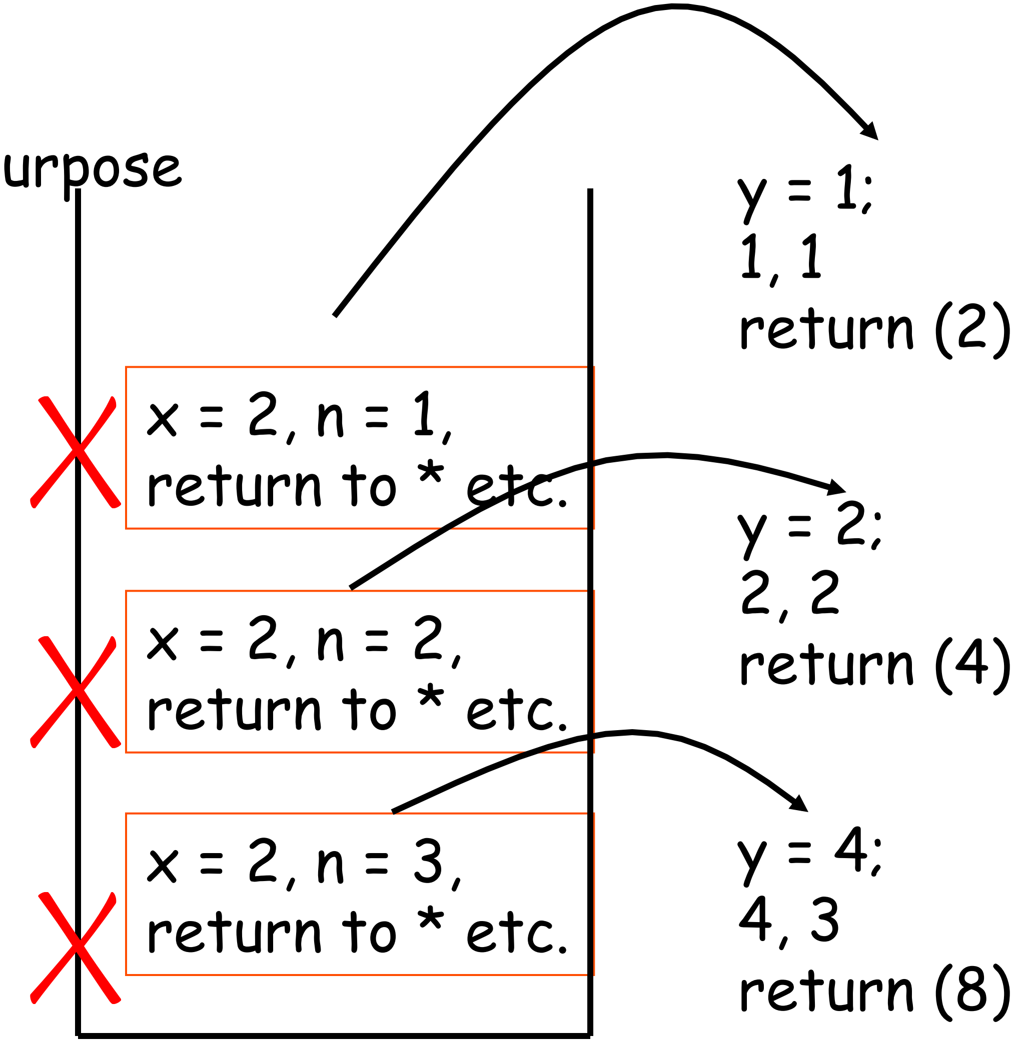
```
Power(x, n) {
  if (n = 0) return (1.0);
  else
     y = Power(x, n-1));  // *
     output (y, n);  // demo purpose
     return (x * y);
}
```

4) Power(2, 0): return 1

3) Power(2, 1): Right at *

2) Power(2, 2): Right at *

1) Power(2, 3): Right at *



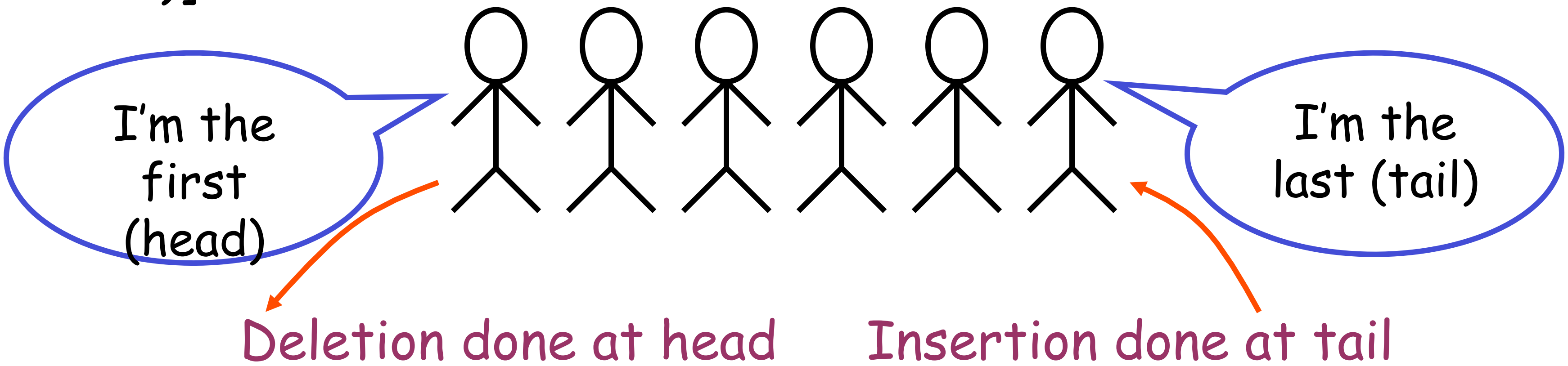
So, for each program (algorithm) that involves recursion, you can always rewrite it to eliminate the recursion by a stack!

Of course, you may be able to come up with an easier non-recursive algorithm without using a stack.

<u>Example:</u>

```
Power(x, n) {
  temp = x;
  for (i = 1 to n-1)
      temp = temp * x;
  return temp;
}
```

(3) Queue – A set of data items arranged in a sequence [Special requirement: element to be deleted from the set is always the one that has been in the set the longest (First-in, first out or FIFO)]



<u>Operations:</u>

Empty(Q): return true if the queue is empty and false otherwise

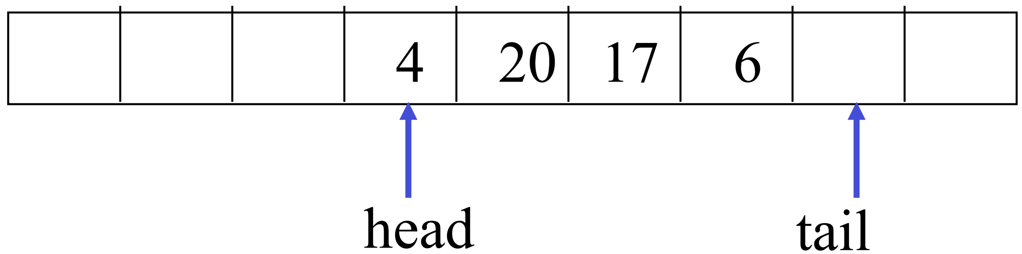
Full(Q): return true if the queue is full and false otherwise

Enqueue(Q, x): insert x to the tail of queue Q

Dequeue(Q): return and then delete the element at the head of Q

Implementation: Array + 2 pointers (both ends of queue)

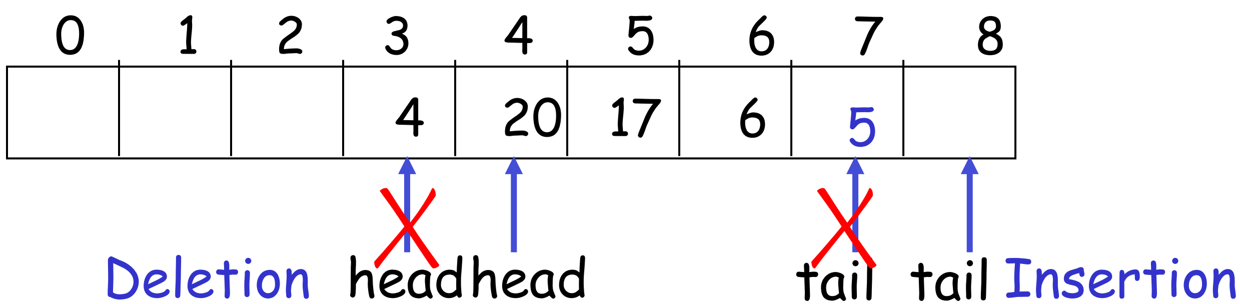
Example: implement a queue using a "circular array"



__One approach__

Two indexes, head always points to the first element; tail points to the next available slot at the end of the queue

one more insertion: element inserted at A[8], tail points to A[0]



When is the queue empty? head = tail So, initially set head = tail = 0

When is the queue full? head = (tail + 1) mod max

// max is the no. of entries in array

Q: can we define tail pointing to the last element? If yes, what will be the empty and full conditions?

Operations:
Empty(Q)
Full(Q)
Enqueue(Q, x)
Dequeue(Q)

```
struct queue{
    int head = tail = 0
    element entry[max]
}
queue Q;
```

```
boolean Empty(Q) {
    if (Q.head = Q.tail) return true
    return false
}
```

$O(1)$

```
boolean Full(Q) {
    if (Q.head = (Q.tail+1)mod max) return true
    return false
}
```

$O(1)$

Operations:
Empty(Q)
Full(Q)
Enqueue(Q, x)
Dequeue(Q)

```
void Enqueue(Q, x) {
    if Full(Q) print "overflow"
    else
      Q.entry[Q.tail] = x;
      Q.tail = (Q.tail +1) mod max
}
```

$O(1)$

```
element Dequeue(Q) {
    if Empty(Q) print "underflow"
    else
       x = Q.entry[Q.head];
       Q.head = (Q.head + 1) mod max
       return x
}
```

$O(1)$

Q: Can you make use of two stacks to implement a queue?

You can do the following:

Initialize a stack

stack S1, stack S2

Use any of the following operations

Empty(S)
Full(S)
Top(S) [S can be S1 or S2]
Push(S, x)
Pop(S)

You cannot create any other data structure

So, all you have is:

```
struct Queue{
    stack S1
    stack S2
}
```

Write the following functions:

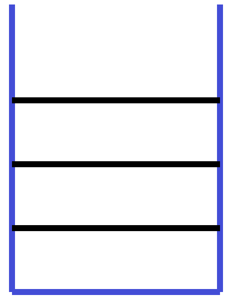
Empty(Q)
Full(Q)
Enqueue(Q, x)
Dequeue(Q)

> Can you do that?
> What is the complexity of each of operation?

One of the approaches:

Top of S1 = Tail of Q

Bottom of S1 = Head of Q



S1

use Stack S1 to represent the queue Q

1) To insert, insert at the top of S1 (easy!)

2) To delete, how we can delete the entry at the bottom?

<u>Solution:</u> make use of Stack S2 as a working buffer.
-pop the elements in S1 and push them to S2 one by one
-delete the top element in S2
-pop the elements in S2 and push them back to S1 one by one

```
struct Queue{
    stack S1
    stack S2
}

Queue Q;
```

```
void Enqueue(Q, x) {
    if Full(Q) print "overflow"
    else
       Push(S1, x);
}
```

O(1)

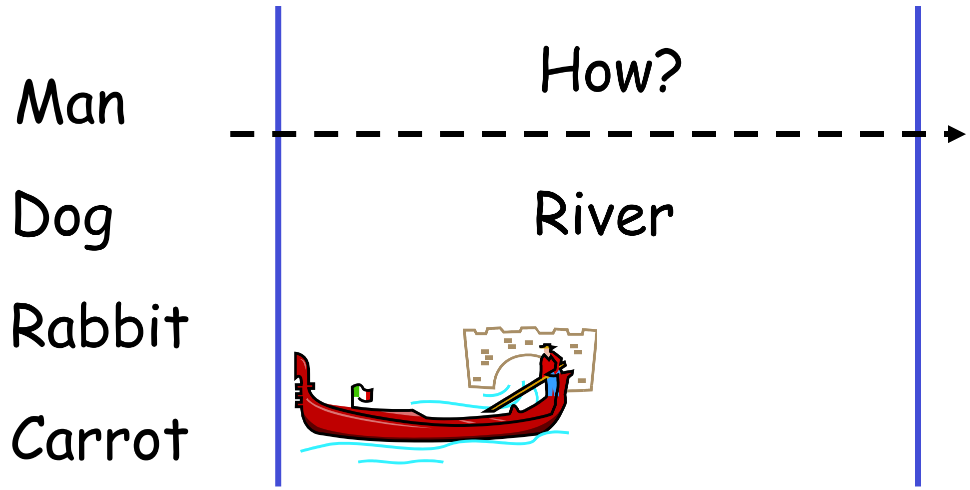
```
element Dequeue(Q) {
    if Empty(Q) print "underflow"
    else
       while not (Empty(S1))
           x = Pop(S1);
           Push(S2, x);
       y = Pop(S2);        // delete the element at head of Q
       while not (Empty(S2))
           x = Pop(S2);
           Push(S1, x);
       return y
}
```

O(n)

Remarks:

1) In the procedures for Queue, we NEVER touch the data structures for Stacks, the advantage is that the implementation of stack can be changed without rewriting the implementation for Queue.
2) Note that it is certainly not a good way to implement Queue, the example only wants to show you sometimes we can use other data structures to implement a "complicated" new data structure.

e.g.

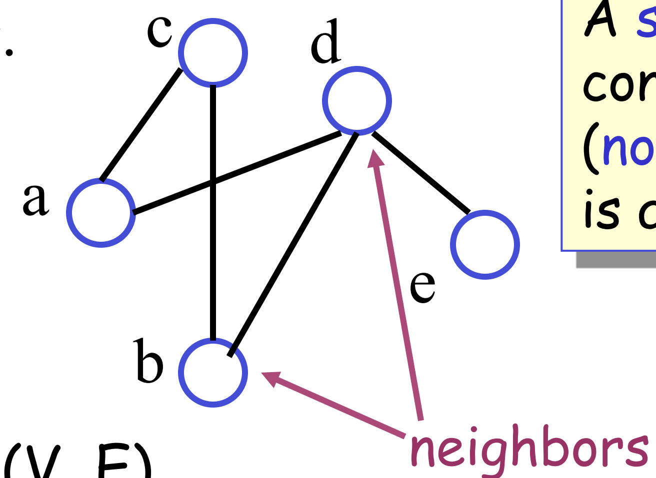


<u>Rules:</u> Only the man knows how to control the boat. The boat can take at most two of them at one time. Without the presence of the man, the dog will eat the rabbit and the rabbit will eat the carrot. How can the man take them over to the other side of the river?

Other applications: DNA assembling, computer networks etc.

Review: (4) Graph

e.g.



G = (V, E)

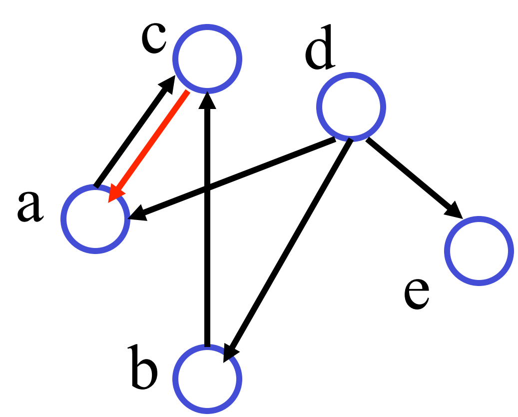
V = {a, b, c, d, e}

E = {(a, c), (a, d), (b, c), (b, d), (d, e)}

A simple undirected graph G = (V, E) consists of a nonempty set of vertices (nodes) V and a set of edges E. Each edge is an unordered pair of distinct vertices.

- undirected: (a, c) and (c, a) refer to the same edge.
- simple: Self loops, e.g. (a, a), are not allowed; and there is at most one edge between two vertices.

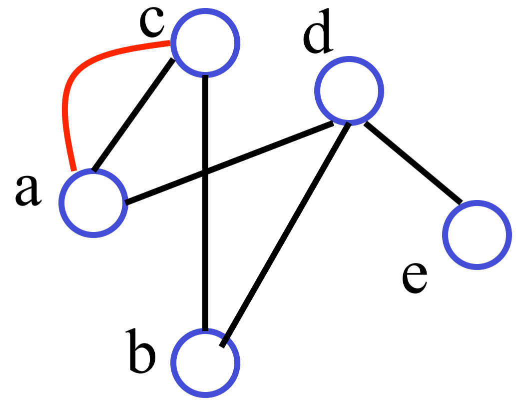
A simple directed graph G = (V, E) consists of a nonempty set of vertices (nodes) V and a set of edges E. Each edge is an ordered pair of distinct vertices.



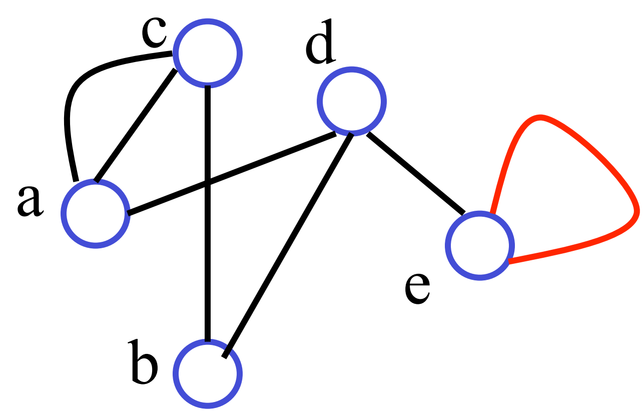
e.g. (a, c) and (c, a) are two different edges; no two edges with the same ordered pair.

Q: what is the degree of an vertex?

Note 1: An undirected graph is called an undirected multigraph if it allows more than one edge between distinct two vertices. (No self loops are allowed).



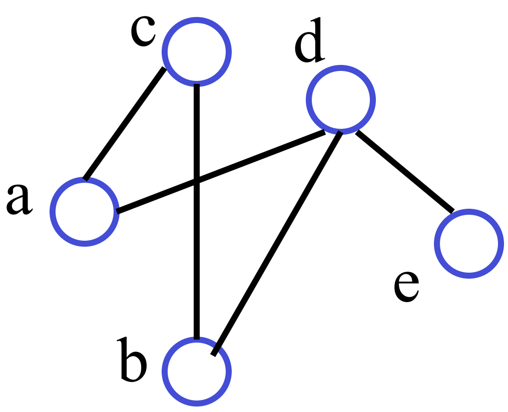
Note 2: An undirected graph is called an undirected pseudograph if it allows more than one edge between distinct two vertices as well as self loops



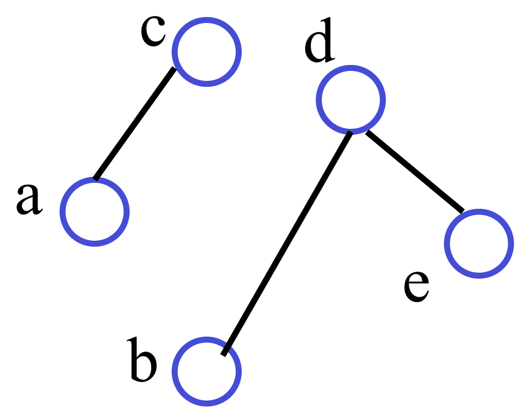
Directed multigraph and directed pseudograph are defined similarly.

What is a connected graph?

For any two vertices u and v in the graph, you can follow the edges from u to v.



Connected



Not connected

Can you state a "path" from a to e?

Note: A lot of problems can be formulated as problems in graph

Which graph to use depends on what problems you want to solve.

e.g. River-crossing problem (man, dog, rabbit, carrot)

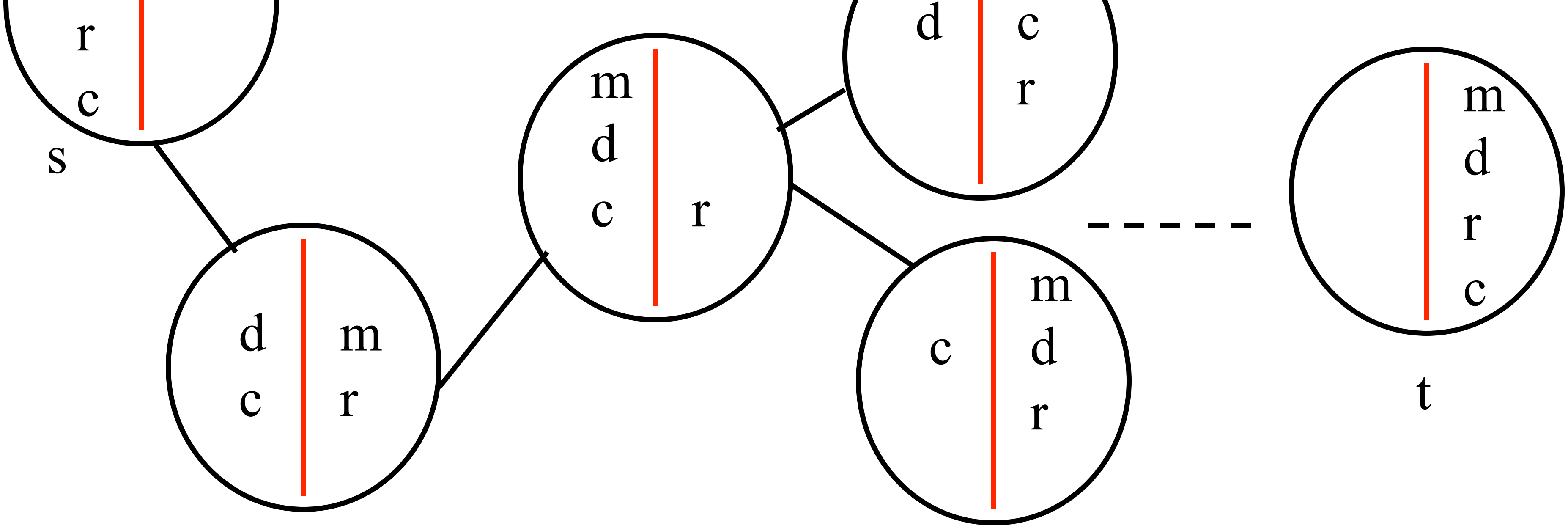
a simple undirected graph

- What does a node represent?
- What does an edge represent?

a possible situation

u — v

u can be changed to v in one step and vice versa



Find a “path” from s to t.

Q: How to represent a graph?

Q: Algorithm to find a path?

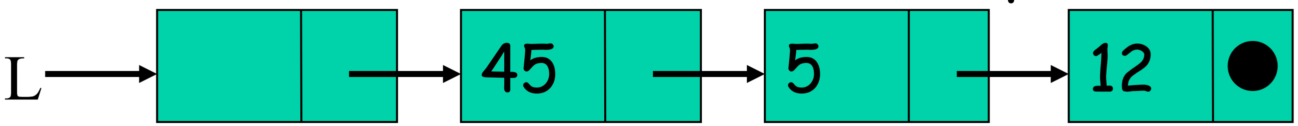
(Linked) list and pointers (a brief overview)

Advantages: The pointer implementation allows us to add/ delete entries from a list without fixing the maximum number of entries in the list at the beginning.

Each node in a list contains (i) the data; (ii) a pointer to the next node

Sentinel (dummy) node

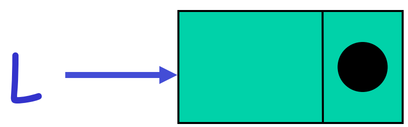
If pointer = nil, end of list



To search a data item, follow L, go through each node one by one.

```
struct node{
    element e;
    node * next;
};
node * L;
```

* means pointer




```
void Create(L) {
    node * t = new node( );
    t.next = nil;
    L = t;
}
```

O(1)

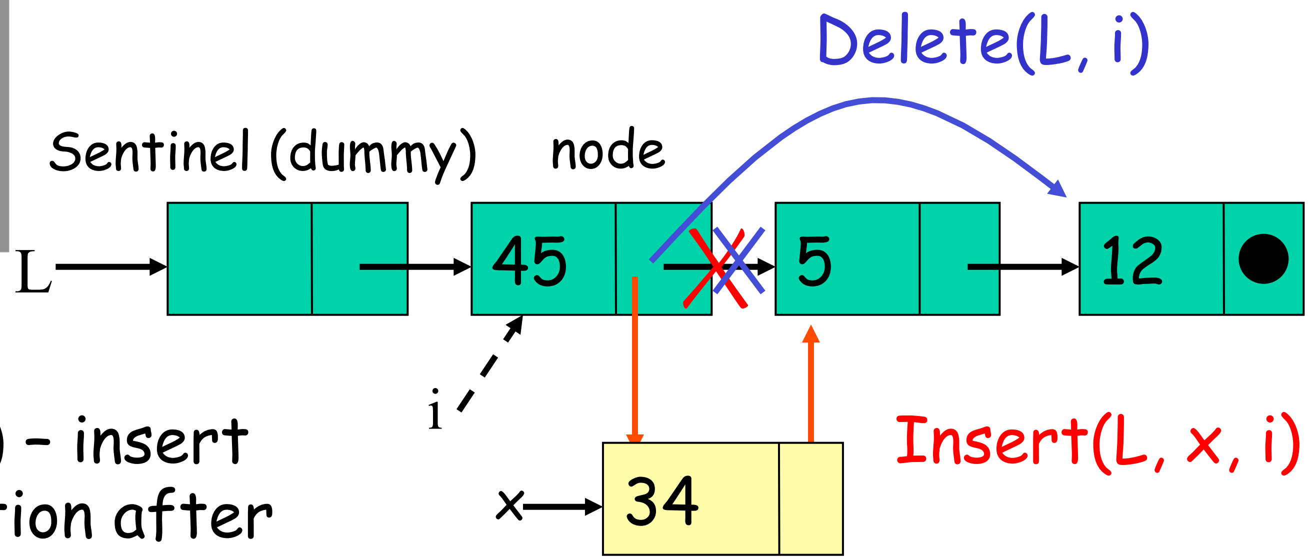
Request memory for a new node

```
node* Search(L, x) {
    node * p = L→next;
    while (p ≠ nil) do
        if (p →e = x →e)
            return p
        p = p →next;
    return nil;
}
```

temp pointer

O(n)

```
struct node{
    element e;
    node * next;
};
node * L;
```



Insert(L, x, i) - insert x in L at location after the node pointed by i (x is also a pointer pointing to the new node to be added).

```
void Insert(L, x, i){
    x →next = i →next;
    i →next = x;
}
```

O(1)

Q: can we swap 2 statements?

Delete(L, i) - Let y be node pointed by i, Delete(L, i) will delete the node after y.

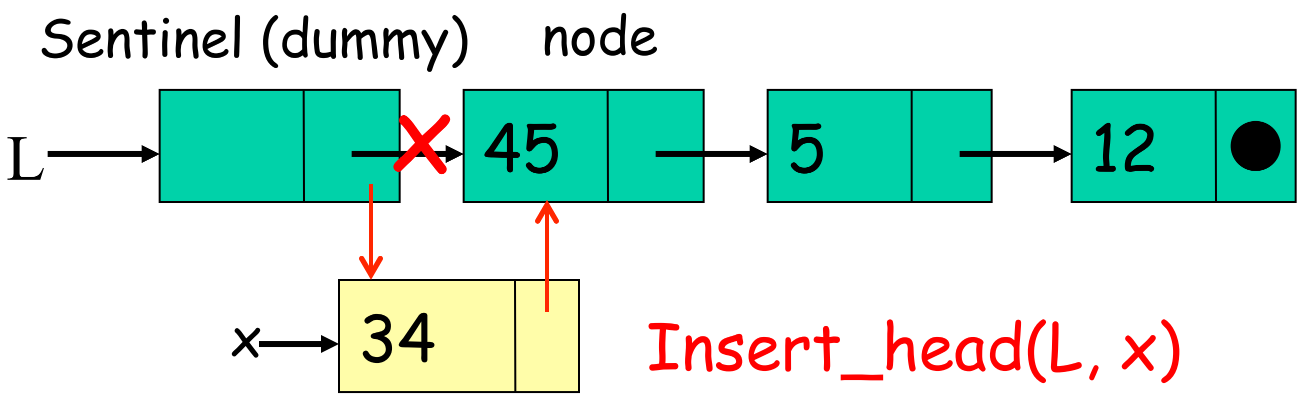
```
void Delete(L, i){
    i →next = i →next →next;
}
```

O(1)

Exercises:

Q: Consider pointer implementation of a linked list, design the algorithm for the following operation and analyze its time complexity.

Insert_head(L, x) – insert the node pointed by x to the head of the list L.




```
void Insert_head(L, x){
    x →next = L →next;
    L →next = x;
}
```

O(1)

Compare the data structure of linked list using array and pointer implementation!

Operation	Array	Pointer
Create(L)	O(1)	
Search(L, x)	O(n)	
Insert(L, x, i)	O(n)	
Delete(L, i)	O(n)	

* For pointer, the definitions of Insert, Delete are not exactly the same as the array implementation in order to achieve O(1) time.

<u>Recall: for array</u>
Insert(L, x, i): Insert x in L at position i+1.
Delete(L, i): Delete the item in L at position i.

Compare the data structure of linked list using array and pointer implementation!

Operation	Array	Pointer
Create(L)	O(1)	O(1)
Search(L, x)	O(n)	O(n)
Insert(L, x, i)	O(n)	O(1)*
Delete(L, i)	O(n)	O(1)*

* For pointer, the definitions of Insert, Delete are not exactly the same as the array implementation in order to achieve O(1) time.

<u>Recall: for array</u>
Insert(L, x, i): Insert x in L at position i.
Delete(L, i): Delete the item in L at position i.

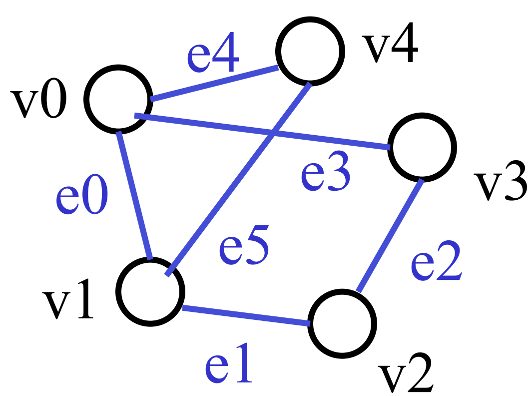
Let us consider a simple connected, undirected graph.

How you can represent a graph?

- Adjacency matrix (2D array)
- Adjacency list (array of linked lists)

$$A = (a_{ij}) \text{ where } a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Example:

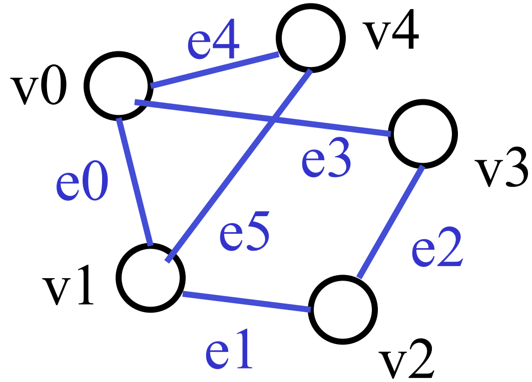


Adjacency matrix

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	0
4	1	1	0	0	0

Q: How about simple directed graphs?

Example:



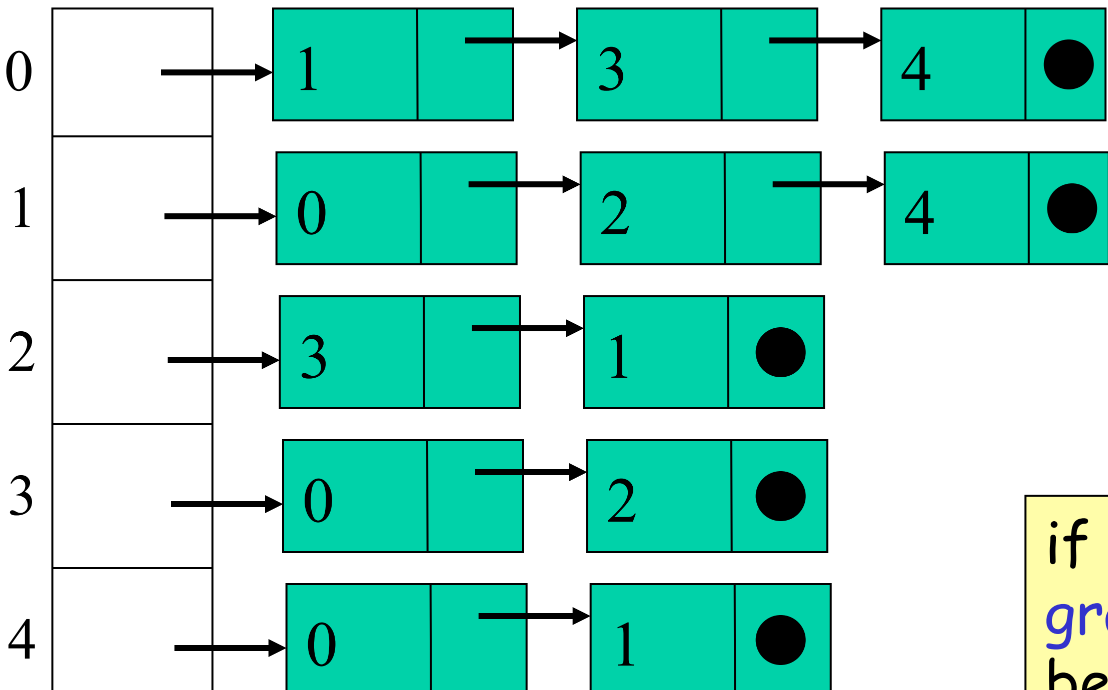
Which representation is better?

Space:
- Adj. Matrix: $O(|V|^2)$
- Adj. List: $O(|V|+|E|)$

Range of |E|?

0 to $|V|\,(|V|-1)/2$

Adjacency List



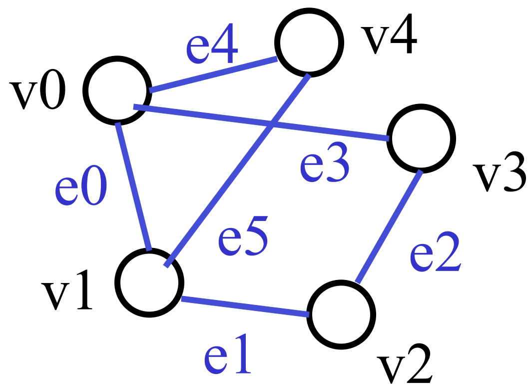
if |E| is large (dense graph), then adj. matrix is better (space efficiency)

if |E| is small (sparse graph), then adj. list is better (space efficiency)

Q: How about simple directed graphs?

One more representation for a graph: Edge list

Example:



Edge	u	v
0	0	1
1	1	2
2	2	3
3	0	3
4	0	4
5	1	4

For each edge, we store the two vertices connected by the edge in a table.

Space complexity: O(|E|)

	Edge List	Adj. Matrix	Adj. List
Space			
Adj. check: $(u, v) \in E$?			
List all adj. vertices of u			
Add an edge (u, v)			
Delete an edge (u, v)			

$m = |E|$, $n = |V|$, d_{max} = maximum degree

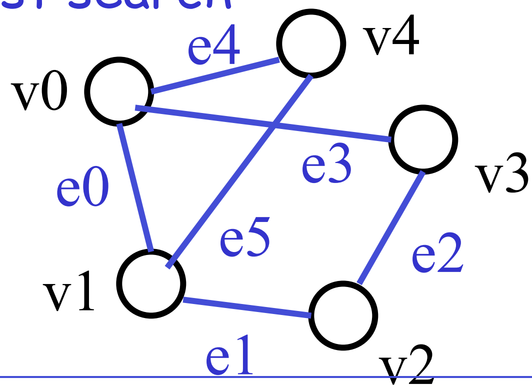
	Edge List	Adj. Matrix	Adj. List
Space	$O(m)$	$O(n^2)$	$O(n+m)$
Adj. check: $(u, v) \in E$?	$O(m)$	$O(1)$	$O(d_{max})$
List all adj. vertices of u	$O(m)$	$O(n)$	$O(d_{max})$
Add an edge (u, v)	$O(1)$	$O(1)$	$O(1)$
Delete an edge (u, v)	$O(m)$	$O(1)$	$O(d_{max})$

$m = |E|$, $n = |V|$, d_{max} = maximum degree

We need some systematic ways to explore a graph (e.g. to find all nodes "reachable" from a given node)

Breadth-first search

Example:



Idea: Start with any vertex, mark it as visited. Visit all its neighbors, then for each visited vertex, repeat the same procedure until all vertices are visited

e.g. v0, v1, v3, v4, v2
e.g. v3, v0, v2, v1, v4

```
BFS(v, n) {// v: starting vertex, n: |V|
   queue Q
   int visited[0..n-1] = {0,...0}  // unvisited
   Enqueue(Q, v)
   visited[v] = 1
   while (not (Empty(Q)) do
      i = Dequeue(Q)
      print(i)
      for each neighbor j of i do
           if (visited[j] = 0) Enqueue(Q,j)
                                visited[j] = 1
}
```

Worst case: O(|V|) time (each vertex in the queue once)

Worst case: O(|V|) neighbors

i	Q
	v0
v0	v1,v3,v4
v1	v3,v4,v2
v3	v4,v2
v4	v2
v2	--

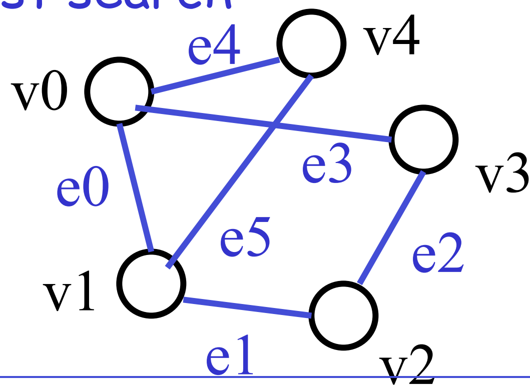
Time complexity (adj. list)?

$O(|V|^2)$ (is it tight?)

We need some systematic ways to explore a graph
(e.g. to find all nodes "reachable" from a given node)

Breadth-first search

Example:



Idea: Start with any vertex, mark it as visited. Visit all its neighbors, then for each visited vertex, repeat the same procedure until all vertices are visited

e.g. v0, v1, v3, v4, v2

e.g. v3, v0, v2, v1, v4

```
BFS(v, n) {// v: starting vertex, n: |V|
   queue Q
   int visited[0..n-1] = {0,…0}  // unvisited
   Enqueue(Q, v)
   visited[v] = 1
   while (not (Empty(Q)) do
        i = Dequeue(Q)
        print(i)
        for each neighbor j of i do
             if (visited[j] = 0) Enqueue(Q,j)
                                  visited[j] = 1
}
```

called at most once per node

each list is scanned at most once

i	Q
	v0
v0	v1,v3,v4
v1	v3,v4,v2
v3	v4,v2
v4	v2
v2	--

Time complexity (adj. list)?

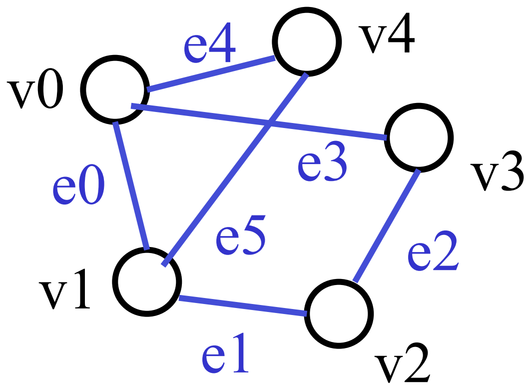
$O(|V|+|E|)$

Worst case: $O(|V|^2)$

How about adj. matrix?

Another searching approach: Depth-first search

Example:



Idea: Start with any vertex v, mark this node as visited, then each unvisited neighbor of v is searched in turn, using the same procedure recursively.

e.g. v0, v1, v2, v3, v4
e.g. v3, v0, v1, v2, v4

Can you write a recursive algorithm for it?
What is the time complexity of your algorithm?
Can you write a non-recursive version for it? [Hint: stack]
How is this non-recursive version compared with BFS(v,n)?

What operations for Graph ADT?
e.g. Connected(G); Neighbors(G, u); Path(u, v, G);
Shortest_path(u, v, G); Has_cycles(G) etc.