Chapter 3

Data Structures

COMP2119

Data types

A programming language usually provides certain *basic data types*. A data type is a set of values and a collection of operations on those values.

- e.g., *integers* with '+', '-', '*', '/', mod....

Some provide additional data types

- e.g., *strings* with "concat", "length", "strcmp". Strings are usually implemented as character arrays, but the system hides all the details. To the users, strings are just another data type with the set of operations that achieve certain specified goals.

Encapsulation

Abstract data type (ADT)

We can extend the concept of *encapsulation* to more complicated data types. An *abstract data type* specifies the *information we want to maintain* together with a *collection of operations* that manage the information.

To represent an ADT we use data structures, which are collections of variables, possibly of several different data types, connected in various ways.

Aggregation

3 common aggregate mechanisms (*in C terminology*): *array*, *record structure*, and *pointer*.

With the basic data types and the aggregate mechanisms, one can construct more complex data structures to represent more complex ADTs.

A plane

As an example, in the wallpaper program, we need an ADT to represent the drawing plane. Intuitively, the drawing plane consists of *n*-by-*n* pixels, each pixel is referred to by a pair of indexes, called co-ordinates. Each pixel is either “on” or “off” (state).

If *P* is a plane, we consider the following operations:

- *Clear*(*P*): set all pixels in *P* to “off”.
- *State*(*P*, *i*, *j*): return the state of the pixel at co-ordinates (*i*,*j*).
- *Plot*(*P*, *i*, *j*): set the pixel at co-ordinates (*i*,*j*) to “on”.

A plane

Information → data structures

As an example, in the wallpaper program, we need an ADT to represent the drawing plane. Intuitively, the drawing plane consists of *n*-by-*n* pixels, each pixel is referred to by a pair of indexes, called co-ordinates. Each pixel is either "on" or "off" (state).

If *P* is a plane, we consider the following operations:

- *Clear*(*P*): set all pixels in *P* to "off".
- *State*(*P*, *i*, *j*): return the state of the pixel at co-ordinates (*i*,*j*).
- *Plot*(*P*, *i*, *j*): set the pixel at co-ordinates (*i*,*j*) to "on".

Operations → algorithms

ADT and data structures

Note that our definition of the ADT plane is *abstract* in the sense that we did not specify how the plane or the operations are implemented in any specific program language.

To realize an ADT in a program, we

- *build data structures* to organize and to represent the information
- *devise algorithms* (and hence programs) to implement the operations

Choosing a data structure

Given an ADT, there are usually a number of ways to implement it. E.g., the plane ADT can be implemented by the following different data structures.

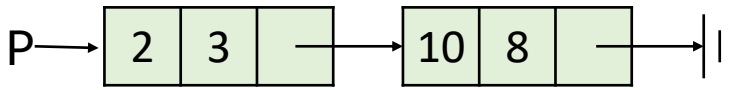
- a 2-dimensional *array* (matrix) of integers.

```
int P[n][n];
```

0	1	0
1	1	0
1	0	0

- a singly *linked list* of nodes, each node stores the co-ordinates of an "on-pixel."

```
typedef struct point* link;
struct point {int x; int y; link next;};
link P;
```



Choosing a data structure

Different data structures result in different implementations of the operations.

e.g., with the matrix representation, the function *State*(*P*,*i*,*j*) can be implemented by the statement:

```
if (P[i][j] == 1) then return (On) else return(Off);
```

which takes constant amount of time.

Choosing a data structure

If we use a singly linked list, we need to search the list, the *worst case* running time is $\Theta(n^2)$. Why?

Space complexity

Another aspect to consider in selecting a data structure is the *storage requirement*.

- The *space complexity* of the matrix representation is $\Theta(n^2)$.
- What is the space complexity of the list representation?

Space complexity

Another aspect to consider in selecting a data structure is the *storage requirement*.

- The *space complexity* of the matrix representation is $\Theta(n^2)$.
- What is the space complexity of the list representation?
  - $\Theta(m)$ where m = number of “on” pixels
  - $O(n^2)$

Choosing a data structure

Our job is to devise data structures to organize the data in such a way that

- the associated operations can be supported efficiently, and
- the structures are space efficient.

Choosing a data structure

Note that the requirements of space and time efficiency could be conflicting.

One has to assess what is more important (time vs. space).

Sets

A set is a collection of elements. An element may have a "*key*" identifying the element. Different data structures are used to handle different types of operations efficiently. Some of those operations include:

- INIT(*S*) -- initialize a set *S* (e.g., to empty set)
- SEARCH(*S*,*k*) -- retrieve the element with key = *k*
- INSERT(*S*,*x*) -- insert element *x* into set *S*
- DELETE(*S*,*x*) -- delete element *x* from *S*
- DELETE_Key(*S*,*k*) -- delete element with key *k* from set *S*

Sets

If we can compare key values …

- MINIMUM(*S*) -- find the element in *S* with the smallest key value.
- MAXIMUM(*S*) -- find the element in *S* with the largest key value.
- SORT(*S*) -- sort the elements according to key values.

If the set is an ordered-set, e.g., each element is given a *position*.

- SUCCESSOR(*S*,*x*) -- find the element in *S* which is ordered next to *x*.
- PREDECESSOR(*S*,*x*) -- find the element in *S* which is ordered in front of *x*.

Sets

We assume that each element has a “key” field.

```
struct element {
    int key;
    .......          /* other info of an element */
}
```

The List ADT

A list is an *ordered* set of elements. E.g., $\{a_1,a_2,a_3,\ldots,a_n\}$ is a list of size n.

For this list:

- a_{i+1} follows (or succeeds) a_i.
- a_{i-1} precedes a_i.
- The *position* of element a_i is i.

The List ADT

Some example operations on a list (*L*):

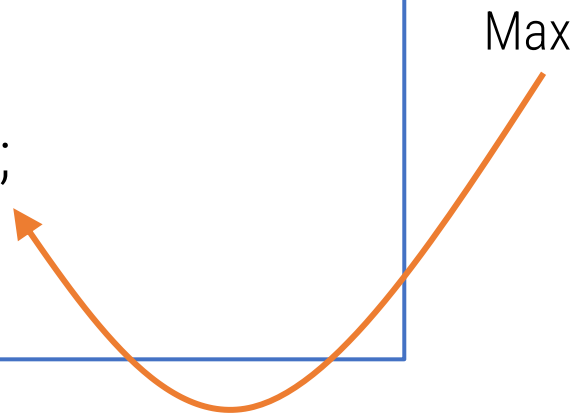
- INIT(*L*),
- SEARCH(*L*,*k*),
- INSERT(*L*,*i*,*x*): insert element *x* at position *i*,
- DELETE(*L*,*x*),
- FIND_i-th(*L*): find the *i*-th element of list *L*.

Implementing a list

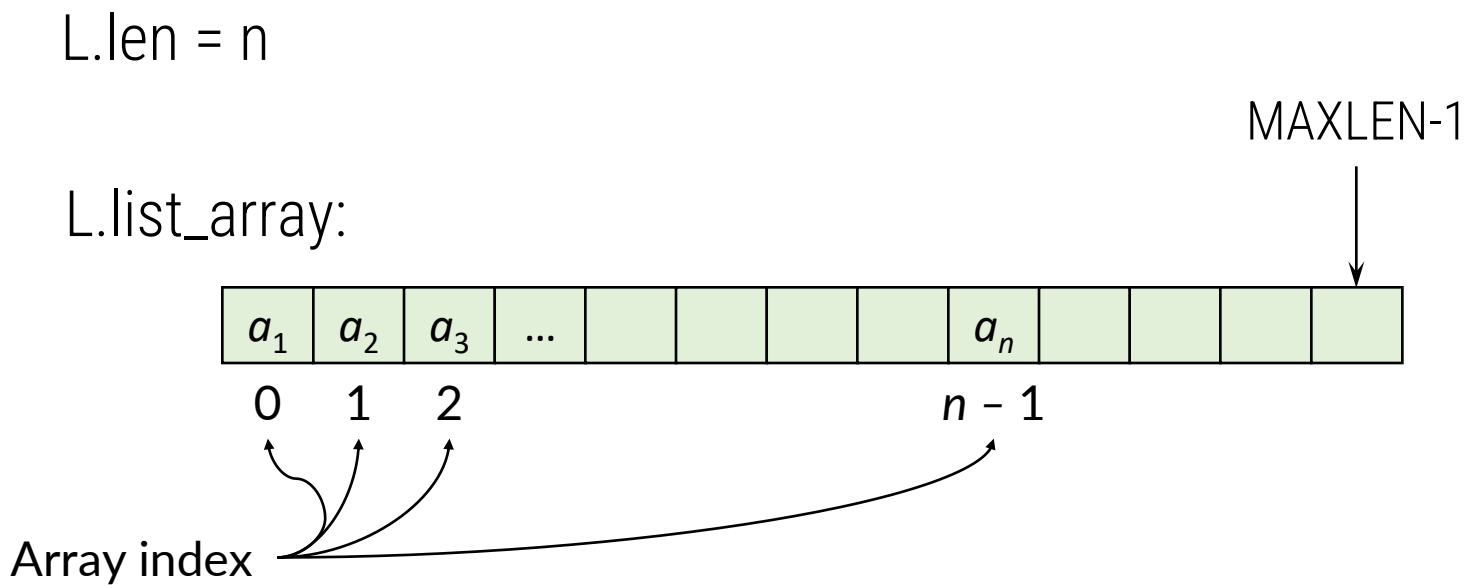
Implementation:

Attempt 1: Simple Array

```
struct list{
   int len;
   element list_array[MAXLEN];
}
list L;
```



Array implementation



Array implementation

INIT takes $\Theta(1)$ time

```
L.len = 0;
```

Array implementation

```
SEARCH(L,k) {
  i = 0;
  while (i <= L.len-1) {
    if (L.list_array[i].key == k)
      break;
    else
      i++;
  }
  if (i > L.len-1)  return ("not found");
  else  return(i);
}
```

Worst case running time of SEARCH is $\Theta(n)$.

What about the average case?

Array implementation

FIND_i-th takes constant time!

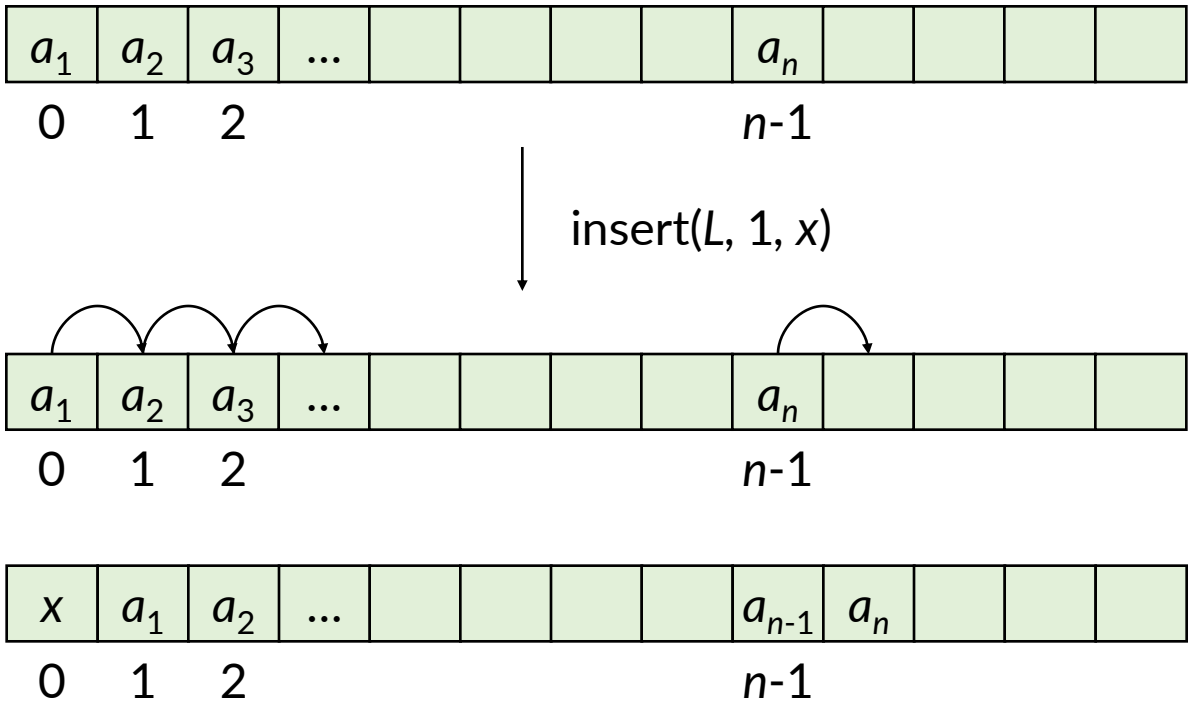
```
return L.list_array[i−1];
```

Array implementation

However, insert and delete are unnecessarily expensive.

- Inserting at the head of the list requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one. The worst case complexity is thus $\Theta(n)$.
- What about the average case?

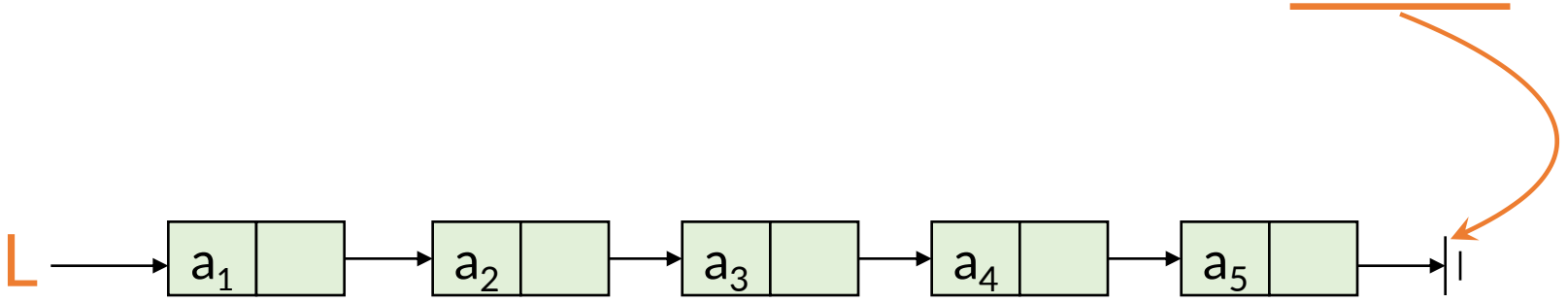
Array implementation



Linked list implementation

Attempt 2: Linked List

A linked list consists of a series of structures (called nodes). Each node contains an element + a link to a successor structure (the "next" pointer). The last node's next pointer equals "NULL"



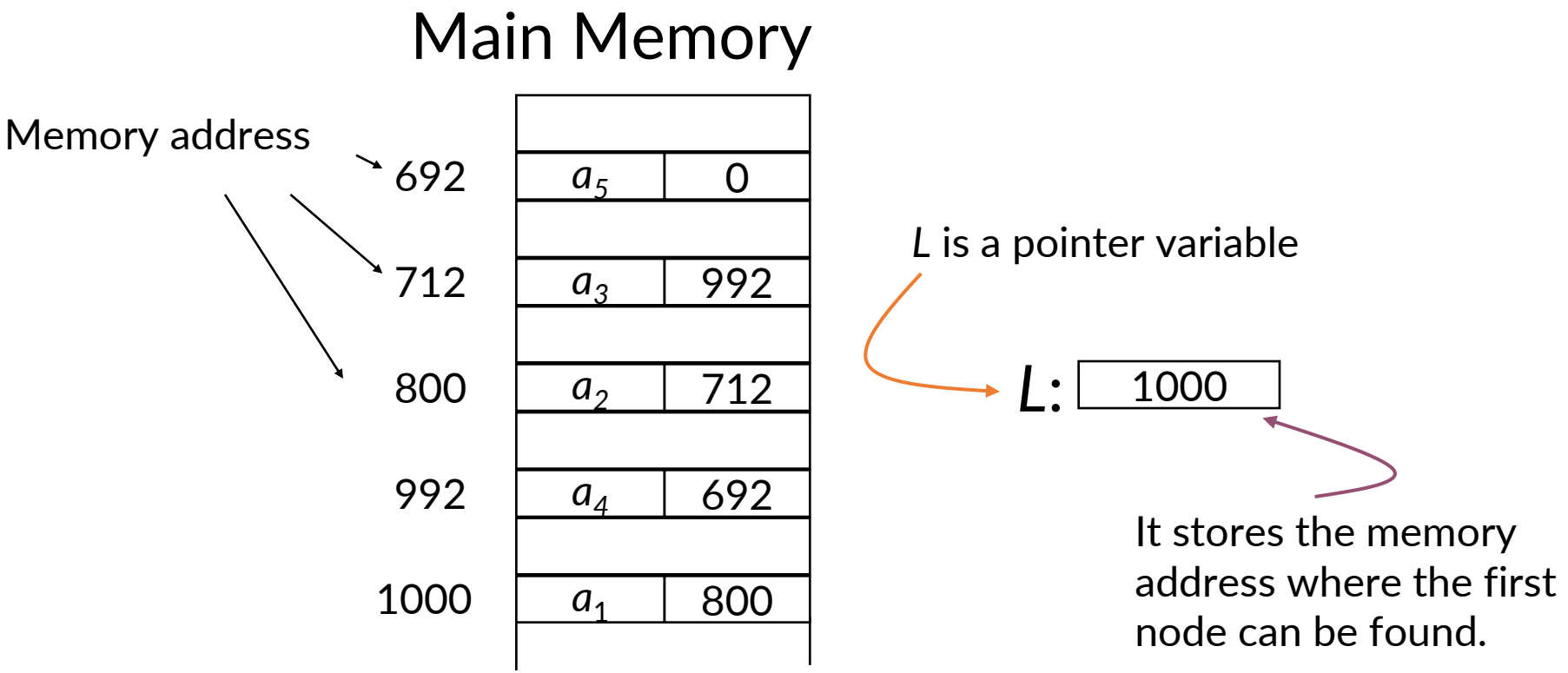
Linked list implementation

A *pointer* is a *variable* that contains the *address* where some other data is stored.

Thus, if *p* = a pointer to a node, then value of *p* = main memory location where a node structure can be found. The key field of that node is accessed by p→key.

A null pointer could be represented by an invalid memory address, e.g., 0.

Linked list implementation

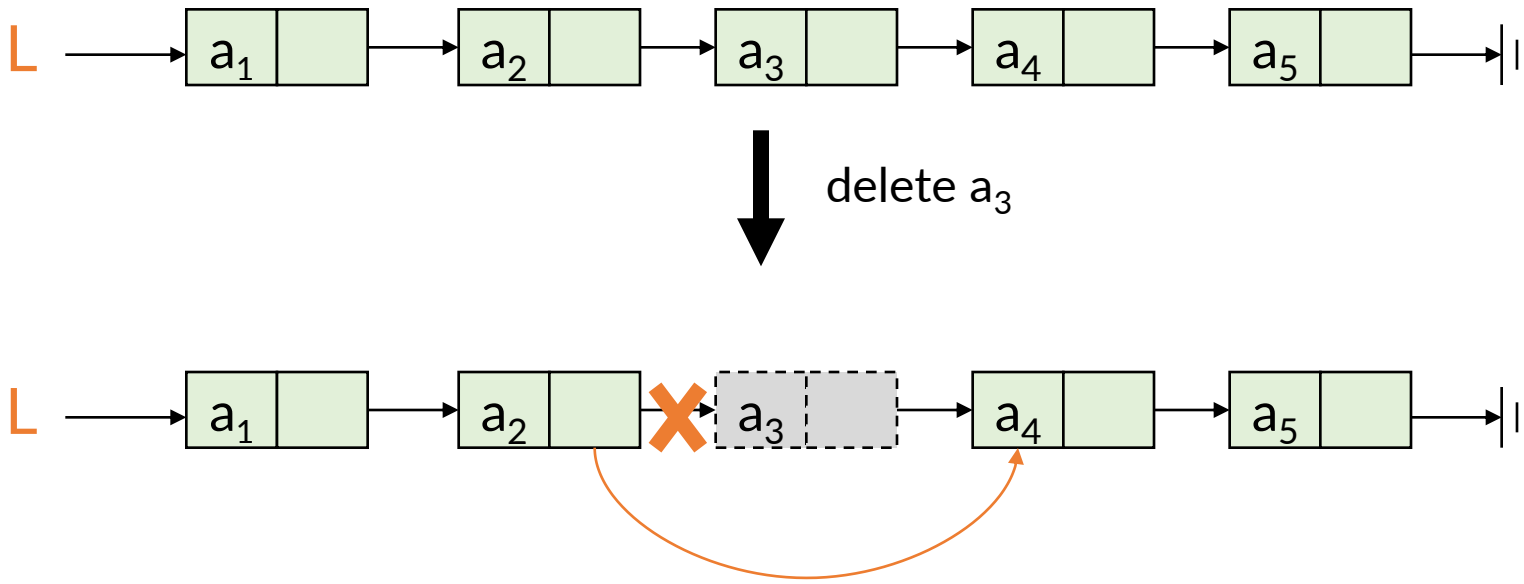


Linked list implementation

SEARCH requires traversing the list through the *next* pointers. Time complexity is thus linear (O(*n*)).

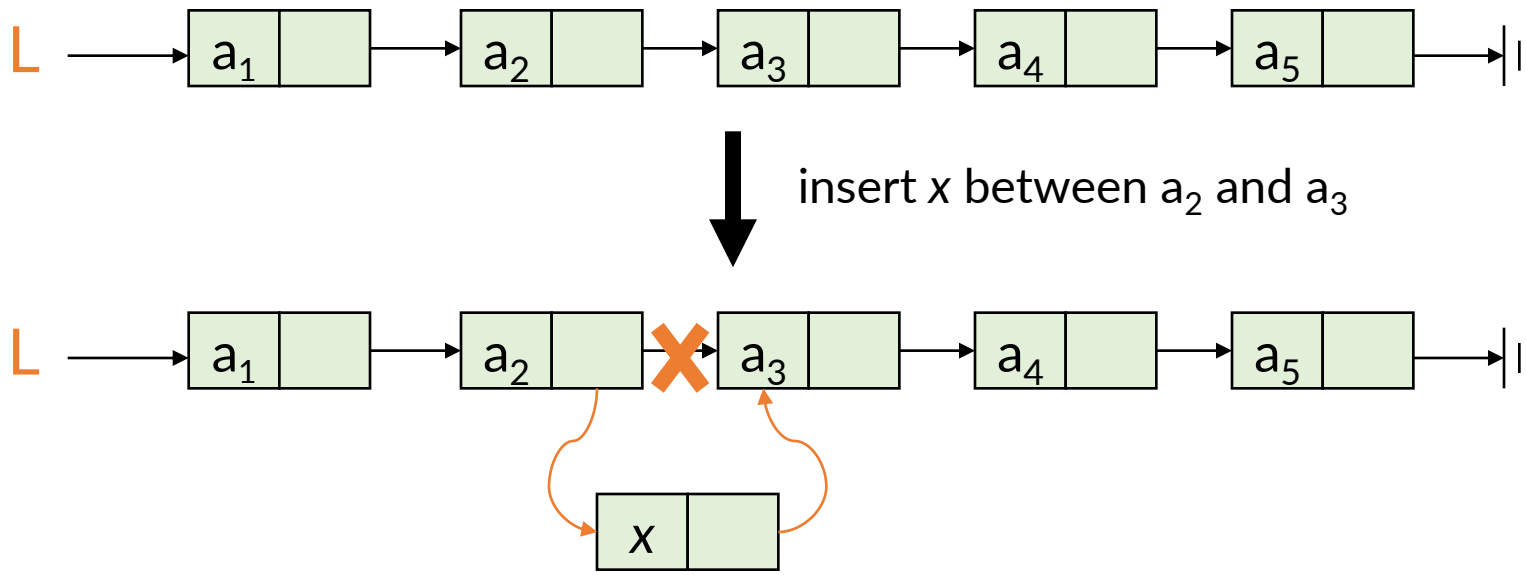
Linked list implementation

DELETE can be done by one pointer change:



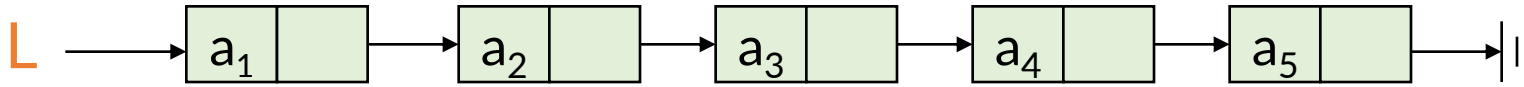
Linked list implementation

INSERT requires obtaining a new node + 2 pointer changes:



Linked list implementation

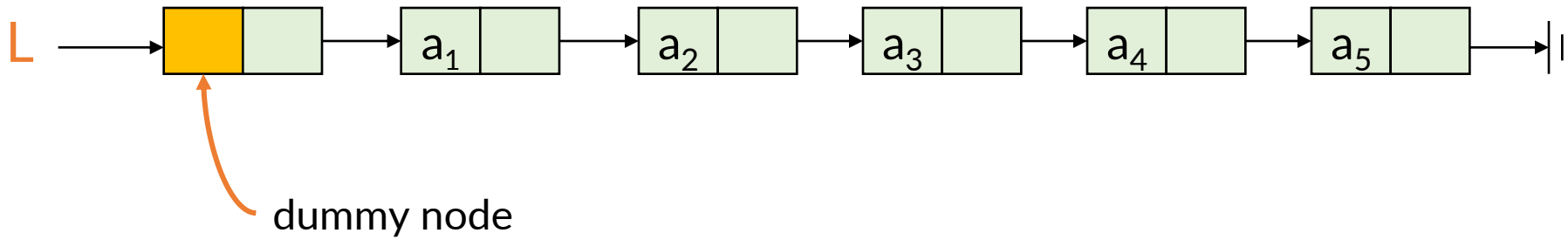
How do we delete a node if it is the 1st node and thus does not have a previous node? Which pointer should be changed?



delete a_1?

Linked list implementation

One simple modification: add a sentinel node (a dummy) at the head of the list.



Linked list implementation

```
typedef struct node * node_ptr;
struct node {
  int key;        /* assume key is of type integer */
  .......         /* some other fields */
  node_ptr next;
};

node_ptr LIST;
```

Linked list implementation

```
INIT(L) {
/* first allocate space for the filler */
  L = new node;

  L → next = NULL;
}
/* an empty list is represented by a lone dummy node */
```

Linked list implementation

To check if a list L is empty:

```
IS_EMPTY(L) {
  if (L → next == NULL)
      return(TRUE);
  else
      return(FALSE);
}
```

Linked list implementation

To search for an element with key *k*

```
SEARCH(L,k) {
   node_ptr p;
   p = L → next;   /* p points to 1st element */
                   /* NULL if none */
   while ((p!=NULL) && (p → key !=k))
      p = p → next;
   return(p);
}
```

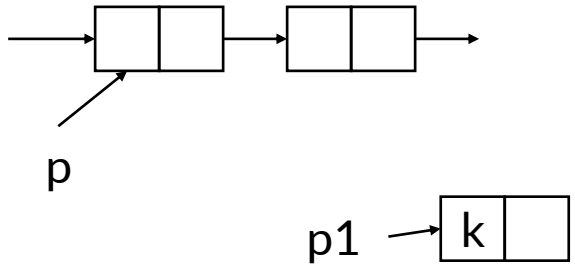
Questions:

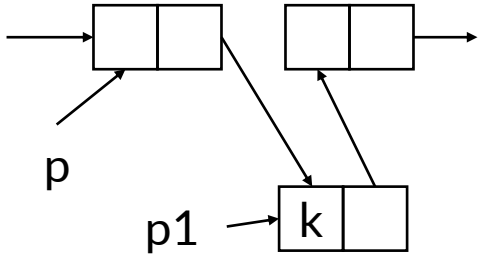
- What does SEARCH return if *k* does not exist in the set?
- What is the complexity of SEARCH?

Linked list implementation

To insert an element (with key = *k*) after node (pointed to by) *p*

```
INSERT(p,k) {
   node_ptr  p1;
   p1 = new node /* allocate a node */
   p1 → key = k;
   p1 → next = p → next;
   p → next = p1;
}
```





Linked list implementation

Q: How to insert an element at the head of the list?

Notes:

- if we know where to insert a node (i.e., we know the pointer *p*), we can achieve insertion in constant time (O(1)).
- if we do not care about where the new element is inserted, we can insert the element to the head of the list. This takes O(1) time.

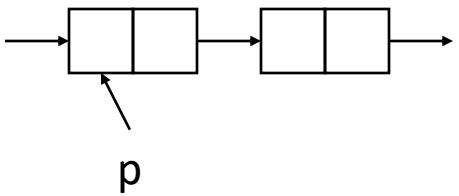
Linked list implementation

If we want to insert the element at the *i*-th position, we have to trace down the list thru the next pointers about *i*-1 times to locate the (*i*-1)st element and insert the new element after it. That takes O(*i*) time.

Linked list implementation

To delete an element following the node pointed to by *p*.

```
DELETE_SUC(p) {
  node_ptr p1;
  if (p → next == NULL) return;
  p1 = p → next;
  p → next = p1 → next;
  delete p1;  /* return the space to
                 the system */
}
```

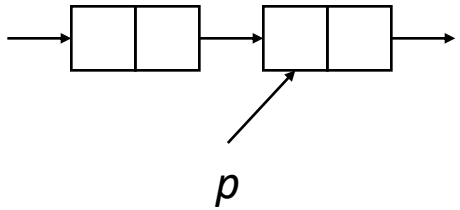


Linked list implementation

What if we want to delete an element pointed to by *p* instead of its successor?

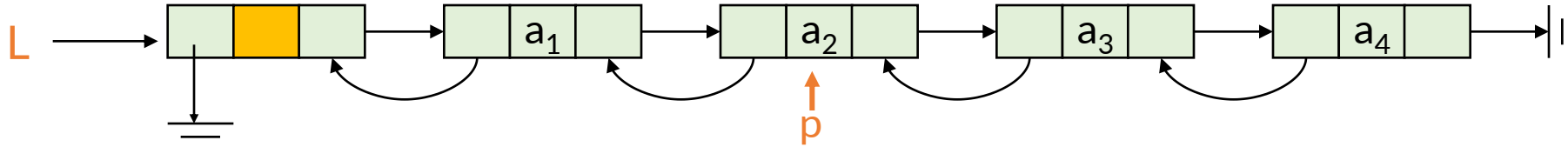
Go thru the list and compare pointers:

```
DELETE(L,p)  {
  node_ptr p1;
  p1 = L;
  while ((p1 → next != NULL) && (p1 → next != p))
       p1= p1 → next;
  DELETE_SUC(p1);}
```



Linked list implementation

DELETE takes O(*n*) time! We can avoid the search by adding a "prev" pointer to the nodes.




```
DELETE(p) {
  p → prev → next = p → next;
  p → next → prev = p → prev;  /* # */
  delete p;
}
```

Linked list implementation

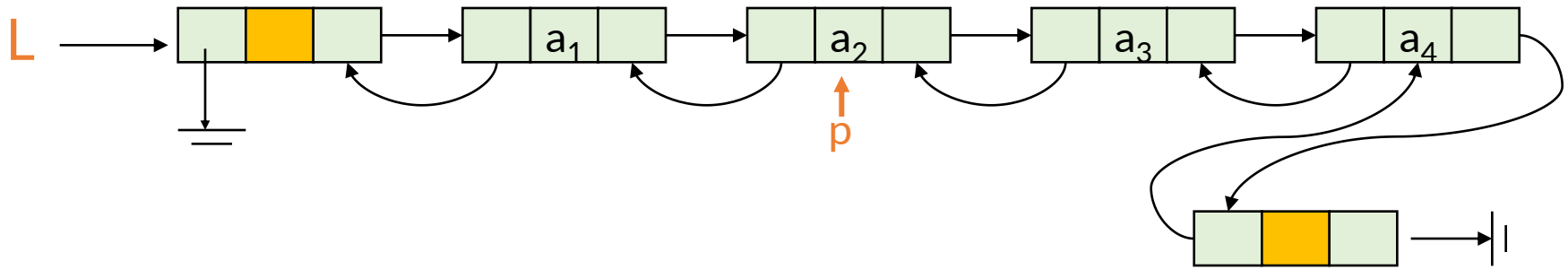
Q: Does it always work?
A: No, considering deleting a_4.

Solution 1: check special case

```
(#)  if (p → next != NULL)
         p → next → prev = p → prev;
```

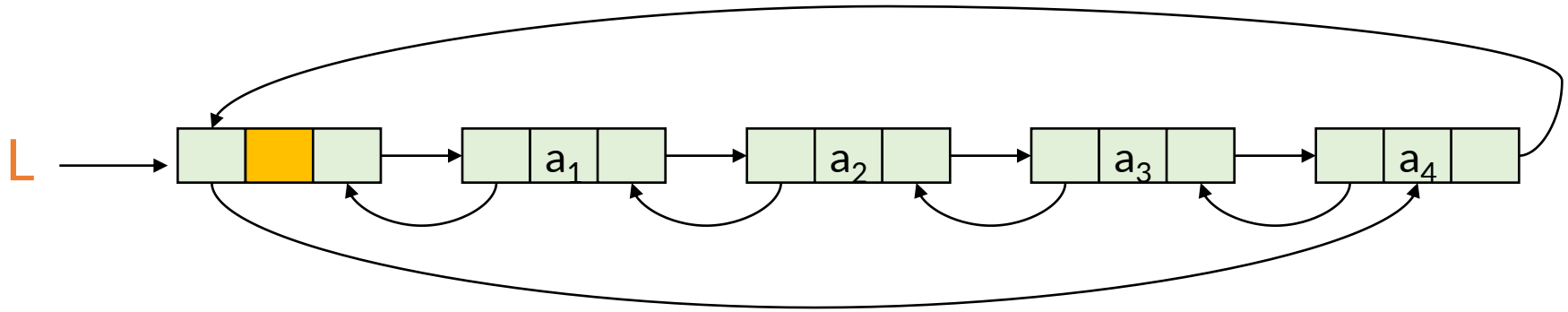
Linked list implementation

Solution 2: add a dummy tail node.



Linked list implementation

Solution 3: circular doubly-linked list



Linked list implementation

If the list is expected to be short (a few elements), the use of dummy node(s) and doubly links could waste space.

If we do not already know the pointer to the element we want to delete (e.g., all we have is the key value), we need to search for the node to be deleted. That takes O(*n*) time.

List implementation

Comparing array and linked list.

- In general, linked lists are not well suited for the "*find the i-th element*" operation that characterizes efficient access in arrays.
- Linked list is more appropriate if we need to rearrange the items very often.
- We don't need to declare the maximum size for the linked list representation.
- Insertion and deletion with linked lists requires dynamic allocation and de-allocation of memory. These are typically expensive operations.

Sorted lists

Although *insert* takes constant time (if we do not care which position in the list to insert a new element), *searching* for a key may require traversing the whole list. This is especially true if the key is not already in the list. If searching is performed very often, we can improve searching efficiency by ordering the elements in the list according to the key values. This is called a *sorted list*.

Searching a sorted list

```
SEARCH_SL(L,k) {
/* assume L is sorted in ascending key value */
/* assume L is a singly linked list with a filler header node */
  node_ptr p;
  p = L → next;   /* p points to 1st element */
                  /* NULL if none */
  while ((p!=NULL) && (p → key < k))
       p = p → next;
  if ((p != NULL) && (p → key == k))
    return(p);
  else
    return(NULL);
}
```

The Stack ADT

A *stack* is a list with the restriction that inserts and deletes can be performed at one end of the list only (called the *top*).

3 main operations:

- push(*x*): insert at top of stack;
- pop(): return and delete the top element;
- top(): return the top element;

Stacks maintain a *LIFO* (Last-In-First-Out) order.

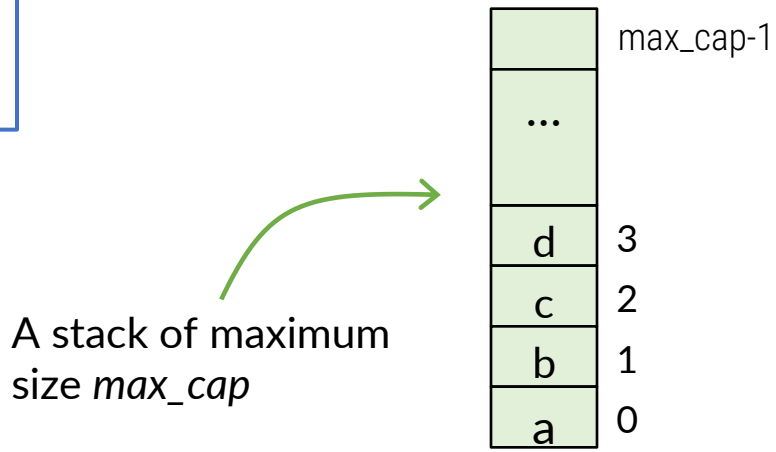
Where do we find stacks?

The Stack ADT

Since a stack is just a special kind of list, we can implement a stack using a singly linked list. However, modification to the stack are done at only one end of the list (via push and pop). An alternative implementation that avoids pointers and dynamic memory allocation is to use an array.

Array implementation of a stack

```
struct Stack {
  int tos;  /* top of stack */
  element  stack_array[max_cap];
}
```



Array implementation of a stack

The disadvantage of using an array is that we need to specify an array size ahead of time.

- if the size is set too large, we waste space;
- if the size is set too small, stack could get overflowed.

If we do not have an idea of how big the stack can get, we use a linked list; otherwise an array implementation is simple and more efficient.

Stack operations

```
INIT(S) {S.tos = -1; }

push(S,x) {
   if (S.tos == max_cap -1)
      return("Stack overflow!");
   S.tos++;
   S.stack_array[S.tos] = x;}

pop(S) {
  if (S.tos == -1)
     return("Stack underflow");
  x = S.stack_array[S.tos];
  S.tos--;   return(x);}
```

The Queue ADT

A queue is another type of list. With a queue, *insert* is done at one end (*tail*) and *delete* is performed at the other end (*head*).

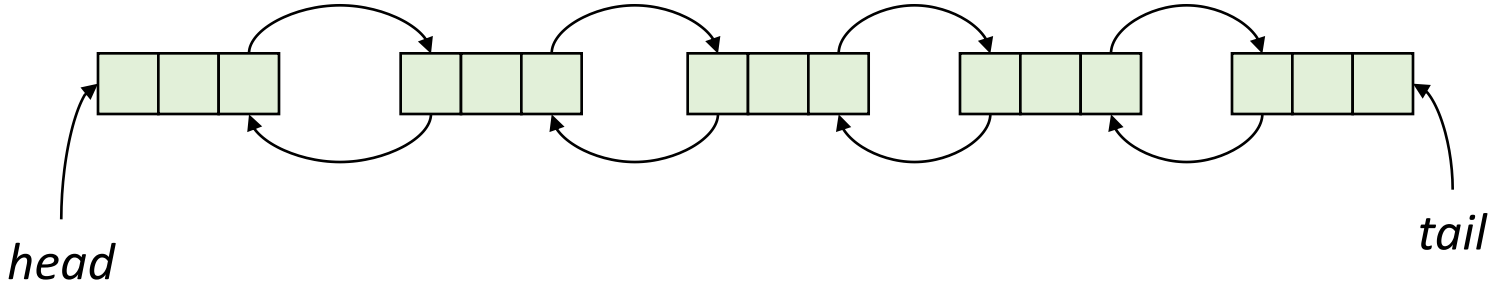
Basic operations:

- enqueue(*Q*,*x*): add element *x* to the tail of the queue;
- dequeue(*Q*): remove and return the element from the head of the queue

A queue thus maintains a *FIFO* (First-In-First-Out) order.

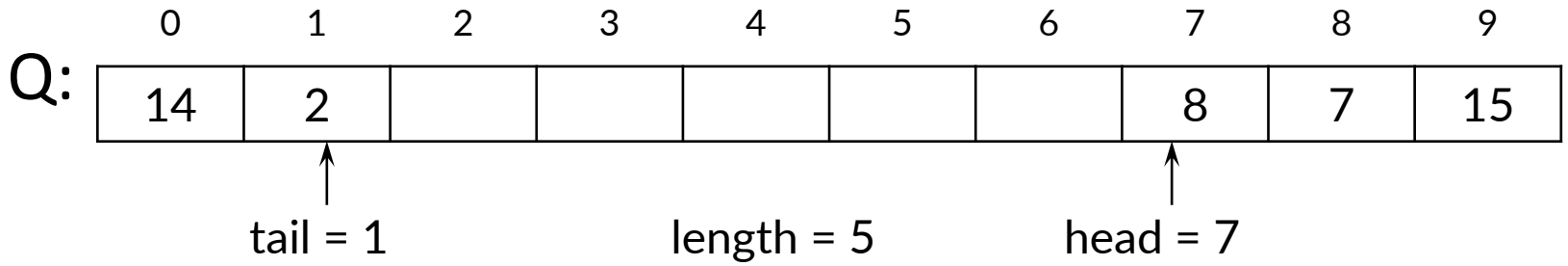
Implementing a queue

A queue can be implemented as a doubly linked list with a pointer to the head and one to the tail.



Array implementation

Or a queue can be implemented by an array



The elements in a queue are stored in positions: head, head+1, …, tail where we "*wrap around*" at the end of the array (i.e., we consider position 0 follows position 9).

We need *two indices* "head" and "tail" + *one count* "length" on the queue length.

Array implementation

```
struct Queue {
   int head;
   int tail;
   int length;
   element queue_array[max_cap]; }


INIT(q) {
   q.head = 1;
   q.tail = q.length = 0;
}
```

Q: what if we do not have q.length?

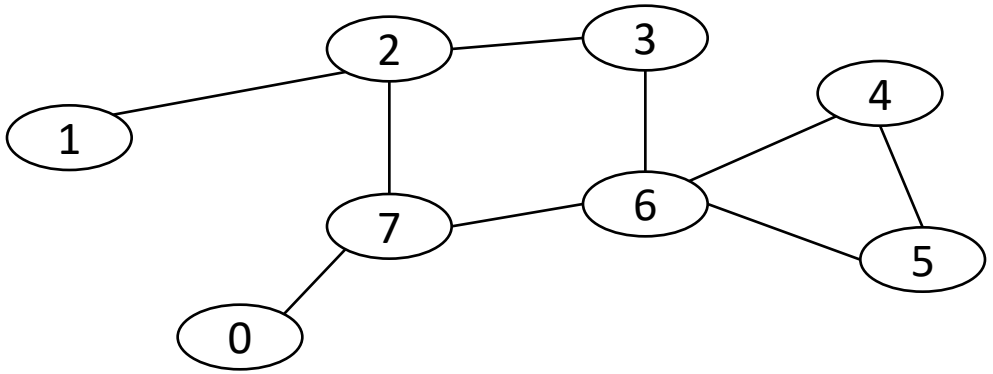
Array implementation

```
enqueue(q,x) {
   if (q.length == max_cap)
      return("Queue overflow");
   tail = (tail + 1) mod max_cap;
   q.queue_array[tail] = x;
   q.length ++;}
```

```
dequeue(q) {
   if (q.length == 0)
      return("Queue underflow");
   x = q.queue_array[head];
   head = (head + 1) mod maxcap;
   q.length --;
   return(x);}
```

The Graph ADT

A graph is a set of *vertices* and a set of connections (*edges*) among the vertices. For example:



The Graph ADT

If a vertex represents certain entity or object, then a graph can be used to represent relationship between objects.

For example,

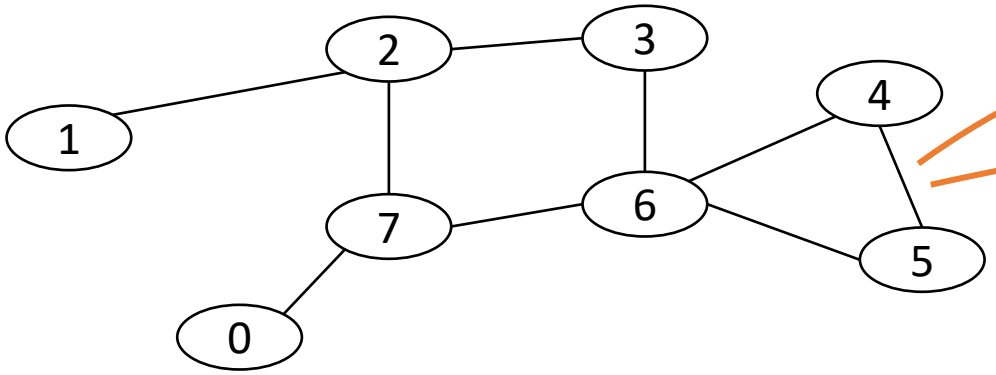
- object: person; edge: two people who know each other
- object: a web page; edge: a hypertext link from one web page to another.

Adjacency matrix

We can represent an (undirected) graph using a 2-dimensional array, called an *adjacency matrix*.

In the matrix, the entry[*i*,*j*] is set to 1 if vertices *i* and *j* are connected; otherwise the entry is set to 0.

Adjacency matrix

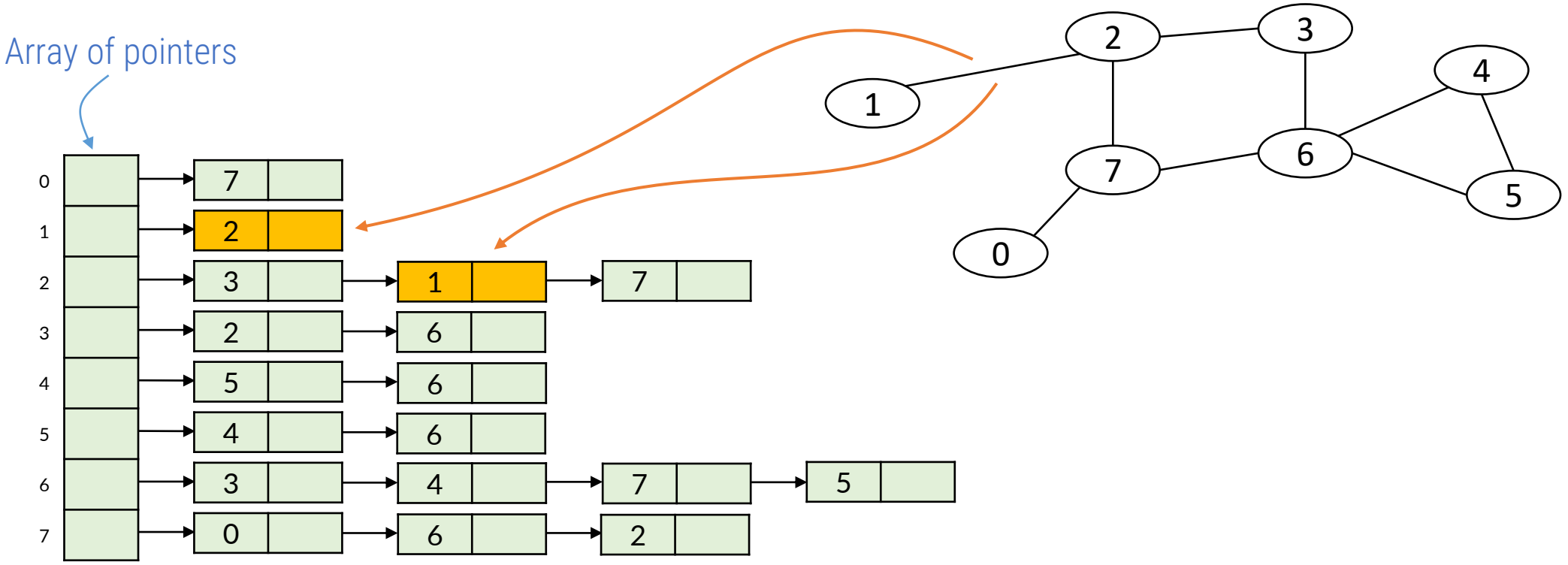


	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0
2	0	1	0	1	0	0	0	1
3	0	0	1	0	0	0	1	0
4	0	0	0	0	0	1	1	0
5	0	0	0	0	1	0	1	0
6	0	0	0	1	1	1	0	1
7	1	0	1	0	0	0	1	0

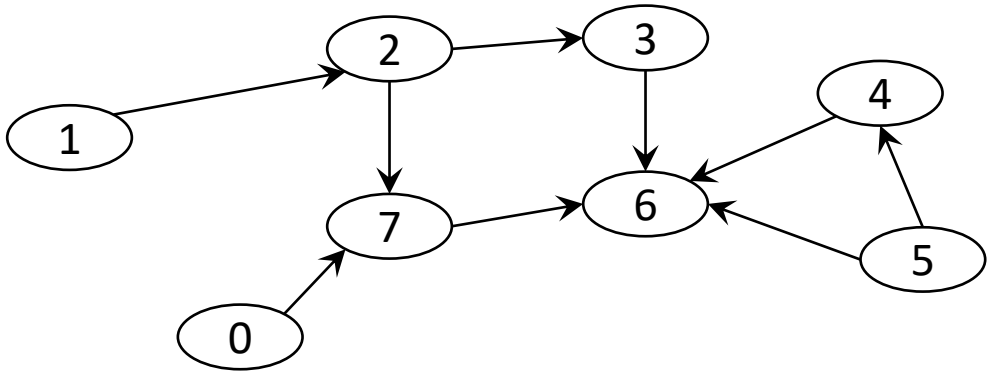
Adjacency lists

Another representation is to use an array of linked lists, called *adjacency lists*. For each vertex *v*, we keep a linked list, which contains one node for each vertex that is connected to *v*.

Adjacency lists

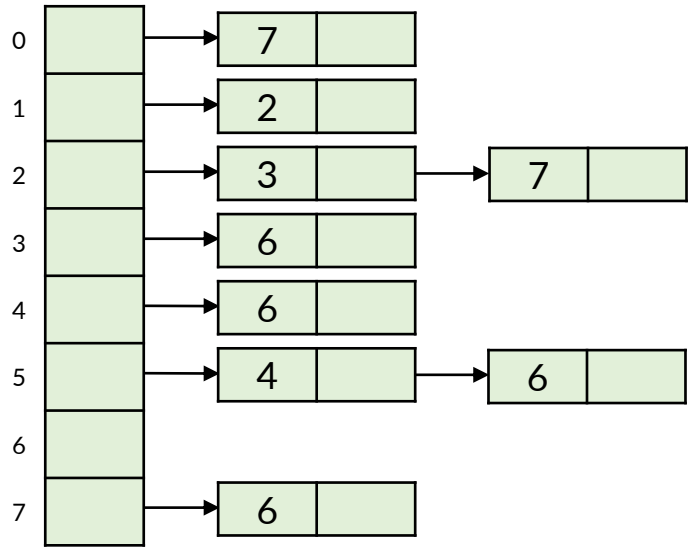


Directed graphs



- A directed graph is one whose edges are directional. Adjacency matrix and adjacency lists structures could be modified to represented directed graphs.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0
2	0	0	0	1	0	0	0	1
3	0	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	0



Space efficiency

Let

- V = number of vertices
- E = number of edges

What are the space requirements of adjacency matrix and adjacency lists?

- adjacency matrix: $O(V^2)$
- adjacency lists: $O(V+E)$

Q: if the graph contains a “large number” of edges, i.e., the graph is “*dense*”, which data structure is more space efficient?

A comparison

In general,

- the matrix representation is more efficient for operations that require determining whether two vertices are connected;
- the adjacency lists representation is more efficient for operations that require processing all the edges in the graph, especially if the graph is sparse.

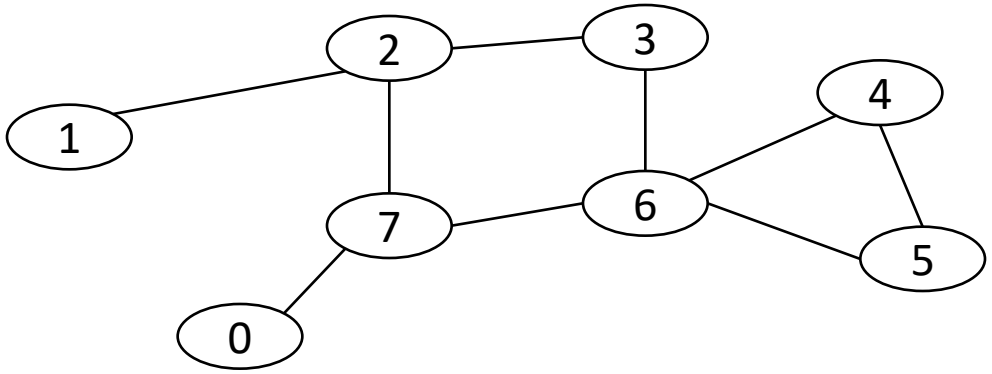
Breadth-first search

Consider the problem of visiting the vertices of a graph in *breadth-first order*. We start with a vertex *k* and visit other vertices such that a vertex that is “close” to *k* is visited before a vertex that is “far” from *k*. A vertex that is not “connected” to *k* should not be visited.

Breadth-first search

If we start at vertex 1 then a valid breadth-first order would be:

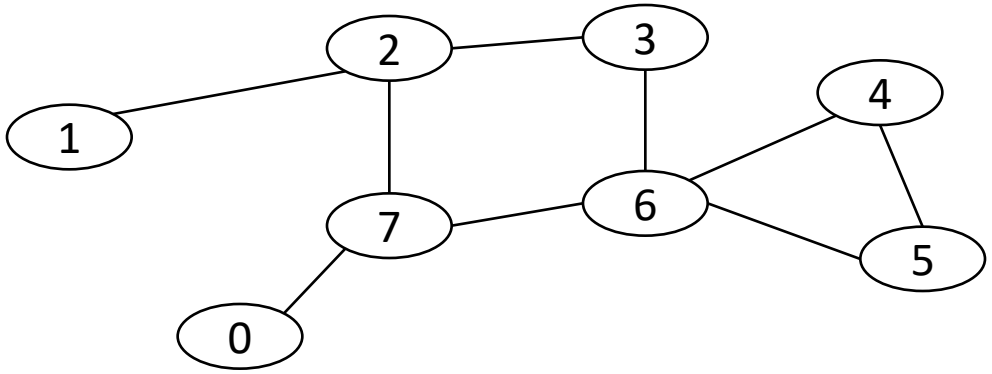
- 1, 2, 3, 7, 6, 0, 5, 4



Breadth-first search

Is the following order a valid breadth-first order if we start at 1?

- 1, 2, 7, 6, 0, 3, 5, 4



Breadth-first search

```
BFS(k,n) {  /* k = starting vertex; n = number of vertices in the graph */
   Queue  Q;    /* declare a queue Q */
   bit visited[n]; /* declare a bit vector visited */
   INIT(Q);
   visited[0..n-1] = all 0's;
   enqueue(Q, k);
   while (!empty(Q)) {
      i = dequeue(Q);
      if (!visited[i]) {
         visited[i] = 1; output(i);
         for each neighbor j of i
              if (!visited[j]) enqueue(Q, j);}}}
```

Breadth-first search

Exercise:

- Try to reason why BFS works.
- How many times enqueue is called? at most? at least?
- How many times dequeue is called?
- The complexity of BFS?
- What if we replace the queue by a stack?

Other basic algorithms on graphs

Depth-first search

Shortest path (Dijkstra's algorithm)

Random walk

Connected components

Network flow