THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

COMP2013
Data Structures and Algorithms

Lecture 6
Elementary Data Structures

Dr. Jieming Shi

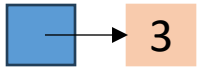
Outline

- **Basics**
- Array
- Matrix
- Queue
- Stack
- Linked List

Value, address, pointer

- Value (e.g., key value)
- Address (address in memory)
- A pointer is a variable of address
  - Point to the object on the address
  - int* p
  - float* p

Address	Value
0x00	01001010



The pointer arrow starts from the *inside* of its box
And points to the object box it refers to.

Dynamic Set

- Manage a set S of n key values
  - Search(S, k)
    - Find if key k exists in S or not, if yes, return a pointer/position to the key in S
  - Insert(S, k)
    - Insert key k into S
  - Delete(S, i)
    - Delete the element pointed by i (at position i) from S
  - Minimum(S)/Maximum(S)
    - Find the minimum/maximum key from S
  - Successor(S, k)
    - Find next larger key in S than k
  - Predecessor(S, k)
    - Find next smaller key in S than k

3	5	4	1	8

Outline

- Basics
- **Array**
- Matrix
- Queue
- Stack
- Linked List

Arrays

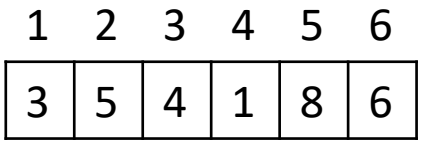
- In an array, each element has the same type and size
- A contiguous sequence of bytes in memory in most programming languages
- If an array with 1-origin indexing starts at address *a*, and each occupies *b* bytes
  - The *i*th element occupies bytes *a+b(i-1)* through *a+bi-1*

Address	Value
0x00	01001010
0x01	10111010
0x02	01011111
0x03	00100100
0x04	01000100
0x05	10100000
0x06	01110100
0x07	01101111
0x08	10111011
...	...
0xFE	11011110
0xFF	10111011

1	2	3	4	5	6
3	5	4	1	8	6

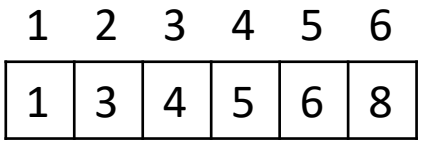
Arrays

- Unsorted array, worst case running time
  - Search O(n)
  - Insert O(n) if the order should be maintained
  - Delete the value at a specific position O(n) if the order should be maintained
  - Minimum and Maximum O(n)
  - Successor and Predecessor O(n)



Arrays

- Sorted array, worst case running time
  - Binary Search O(log n),
  - Insert at the right position O(n)
  - Delete the value at a specific position O(n)
  - Minimum and Maximum O(1)
  - Successor and Predecessor O(log n)
    - Based on binary search



Outline

- Basics
- Array
- **Matrix**
- Queue
- Stack
- Linked List

Matrices

- $M^{m \times n}$: a matrix with m rows and n columns

$$M = \begin{pmatrix} 3 & 5 & 4 \\ 1 & 8 & 6 \end{pmatrix}$$

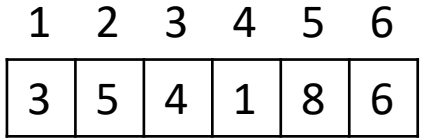
- Store a matrix
  - Row-major order
  - Column-major order

Matrices

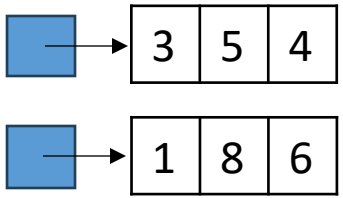
- $M^{m \times n}$: a matrix with m rows and n columns

$$M = \begin{pmatrix} 3 & 5 & 4 \\ 1 & 8 & 6 \end{pmatrix}$$

- Store a matrix
  - Row-major order
    - In a single array



    - With one array per row, and an array of pointers to the row arrays



Matrices

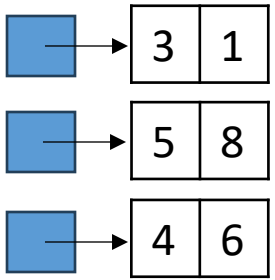
- $M^{m \times n}$: a matrix with m rows and n columns

$$M = \begin{pmatrix} 3 & 5 & 4 \\ 1 & 8 & 6 \end{pmatrix}$$

- Store a matrix
  - Column-major order
    - In a single array

1	2	3	4	5	6
3	1	5	8	4	6

    - With one array per column, and an array of pointers to the column arrays



Matrices

- Block representation
- $$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right)$$

- In a single array
  $\langle 1, 2, 5, 6, 3, 4, 7, 8, 9, 10, 13, 14, 11, 12, 15, 16 \rangle$

Outline

- Basics
- Array
- Matrix
- **Queue**
- Stack
- Linked List

Queue

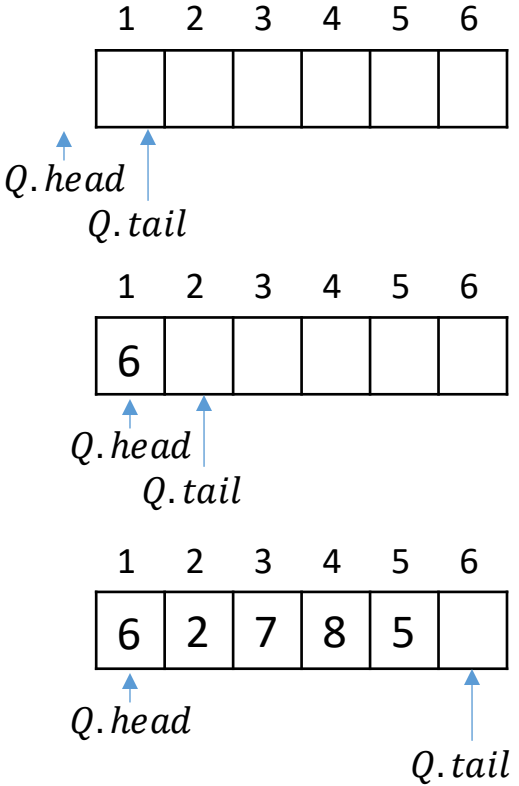
- All elements have the same priority
- A queue has a *head* and a *tail*
- FIFO: first in, first out
  - The element dequeued is the one having firstly joined the queue

Queue

- Operations
  - Enqueue: put an element at the tail; increase queue size by 1
  - Dequeue: get and remove the element at the head, decrease queue size by 1
  - Top: get the element at the head without removing it
- One of the ways to implement a queue is by an array

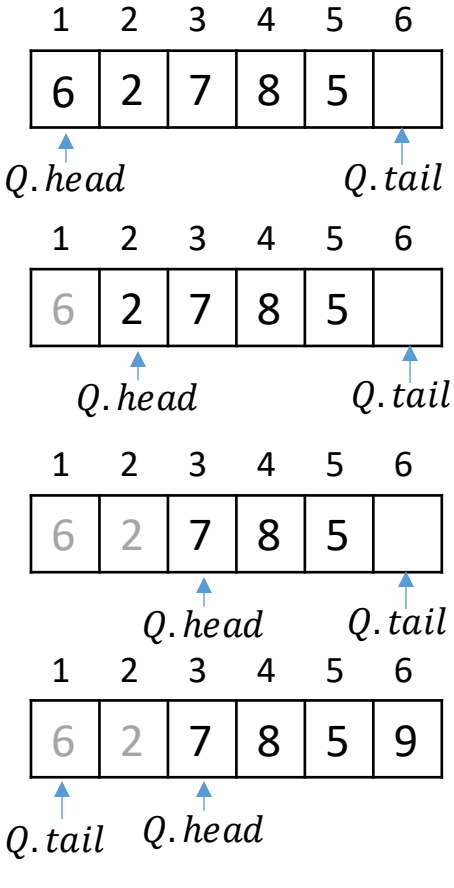
Queue

- A queue *Q* has a *head* and a *tail*
- A queue with *n* (e.g., 6) capacity
- Initially, we have an empty queue
  - *Q.size=0*
- Enqueue 6,
  - *Q.size*=1
- Enqueue 2, 7, 8, 5
  - *Q.size=5*



Queue

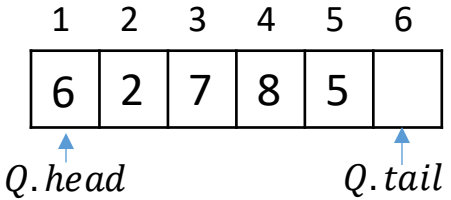
- A queue has a *head* and a *tail*
- A queue with *n* (e.g., 6) capacity
- Dequeue
  - Get and remove head of the queue
  - *Q.size=4*
- Dequeue again
  - Get and remove head of the queue
  - *Q.size=3*
- Enqueue 9
  - *Q.tail+1=7>n=6*
  - *Q.tail* should be **mod** by *n*
- Enqueue 3
  - *Q.size = 5*
  - *Q.tail=2*
  - *Q.head=3*

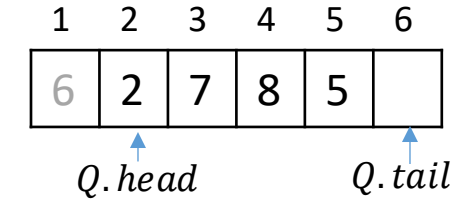


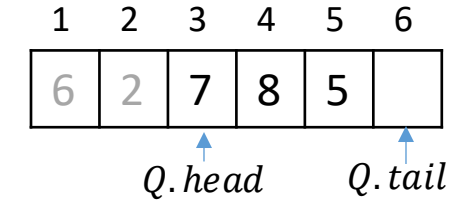
Queue

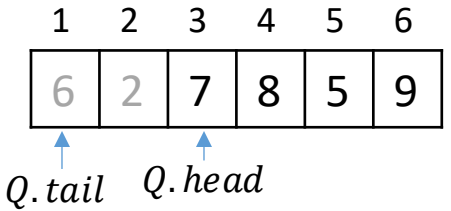
Enqueue(Q, k, n)
1. if Q.size==n
2. error: queue full
3. Q[Q.tail]=k
4. Q.tail=(Q.tail+1) mod n
5. Q.size++

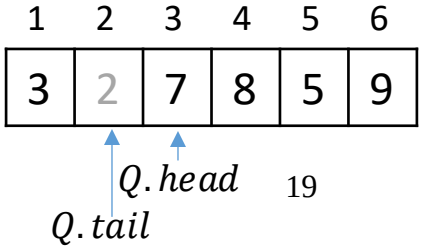
Dequeue(Q)
1. if Q.size==0
2. error: queue empty
3. k=Q[Q.head]
4. Q.head=(Q.head+1) mod n
5. Q.size—
6. return k











Queue

- When a queue is implemented by an array,
  - Enqueue: O(1)
  - Dequeue: O(1)
  - Top: O(1)

Outline

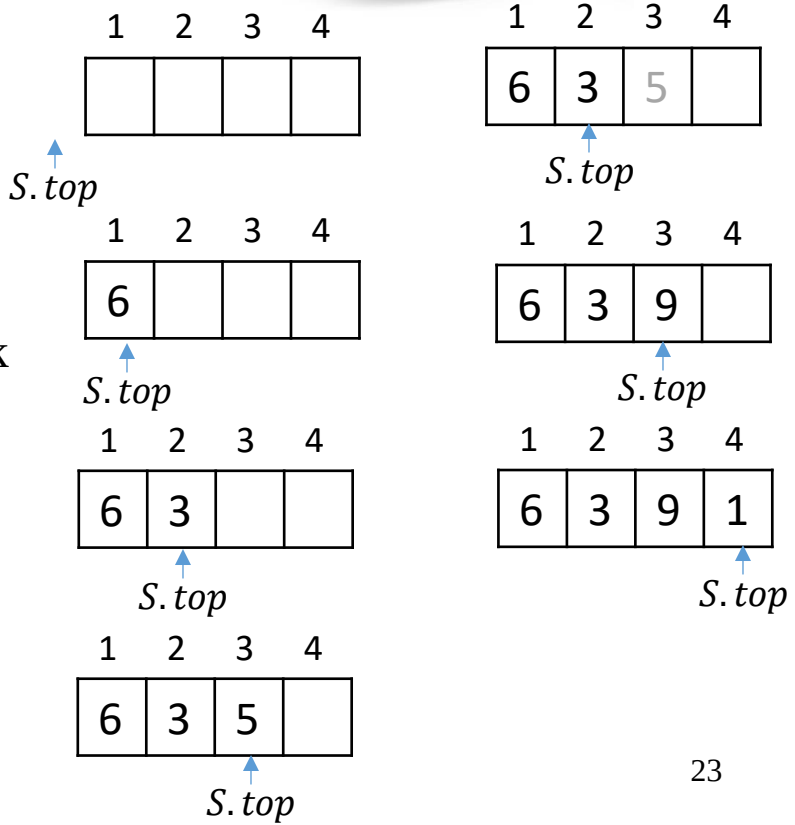
- Basics
- Array
- Matrix
- Queue
- **Stack**
- Linked List

Stack

- A stack of plates with a top plate
- LIFO: Last-in, first out
  - The element deleted from the stack is the one most recently inserted (on the top)
  - First in, last out
- Applications:
  - Syntax parsing (e.g., for XML, HTML)
  - Searching in puzzle problems
  - Function call during program execution
- One of the ways to implement a stack is by an array

Stack

- A stack with capacity n=4 in the array
- $S.top$ points to the top element of the stack
  - Empty stack has $S.top = 0$
- Push(S, k): add k into stack
  - Push (S, 6)
  - Push (S, 3)
  - Push (S, 5)
- Pop(S): get and remove the top from the stack
  - Pop out 5
- Push (S, 9)
- Push (S, 1)
- Push (S, 4) – failed since stack is full
- Top(S)

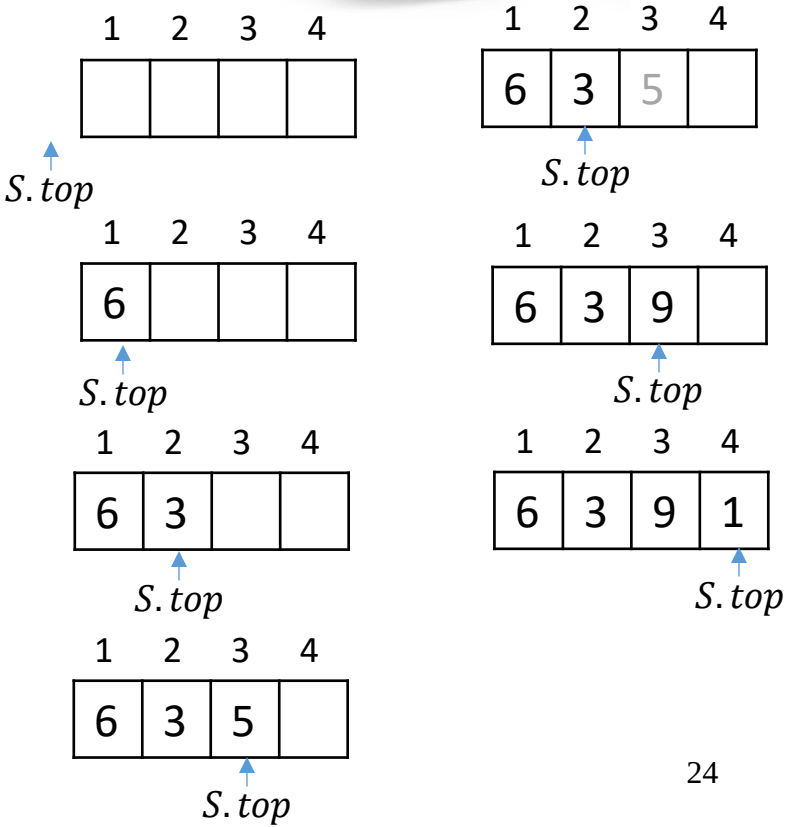


Stack

Empty(S)
1. if S.top==0
2. return True
3. return False

Push(S, k, n)
1. if S.top==n
2. error: stack full
3. S.top++
4. S[S.top]=k

Pop(S)
1. if S.top==0
2. error: stack empty
3. k=S[S.top]
4. S.top--
4. return k



Exercises

- Reverse an array
  - [6,2,7,8] → [8,7,2,6]

Exercises

- Match parentheses
  - Examples:
    - A young elf named Aria (known for her bravery and wisdom) discovered an ancient map [hidden inside an old book {which belonged to her great-grandfather (a renowned explorer)}].
    - Code

```
// The QuickSort function implementation
void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

  - Parentheses:
    - opening [({
    - Closing })]
    - Each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

Exercises

- Match parentheses
  - Are the following parentheses correct? (True/False)
    - [({)}]
    - {([()])}
    - ((((((()))(()))((())))
  - Parentheses:
    - opening [({
    - Closing }])
    - Each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

Exercises

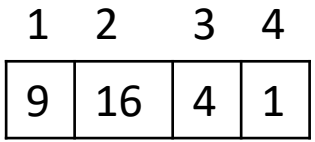
- Match parentheses
  - Are the following parentheses correct? (True/False)
    - [({)}]
    - {([()])}
    - ((((((()))(()))((())))
  - Idea:
    - Read a bracket character
    - If it is an opening bracket, push it into the stack
    - If it is a closing bracket, pop the top element (an opening bracket) from the stack, and
      - If the popped element does not match the closing bracket, return False
      - Otherwise,
    - If all characters have been read and matched AND the stack is empty: TRUE
    - Otherwise, for all the other cases, FALSE

Outline

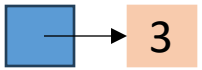
- Basics
- Array
- Matrix
- Queue
- Stack
- **Linked List**

Linked lists

- Array: a contiguous sequence of bytes in memory
  - The linear order is determined by array indices



- Linked list: a linear order of elements but may not be contiguous in memory
  - The linear order is decided by **pointers** (addresses)

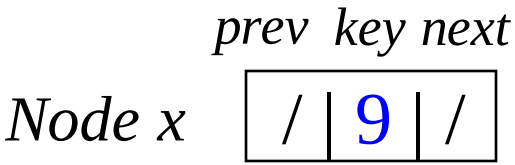


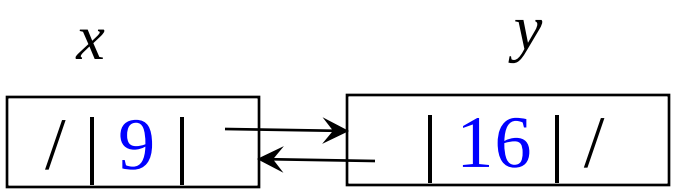
Doubly Linked List

- In a doubly linked list, each node/element *x* stores:
  - *key*: key value of the node
  - *prev*: pointer to the previous node
  - *next*: pointer to the next node

```
class Node {
  int key;
  Node* prev;
  Node* next;
}
```

- A pointer can be set to:
  - The memory address of another node in Node type, OR
  - `NIL` value (shown as '/' in examples)

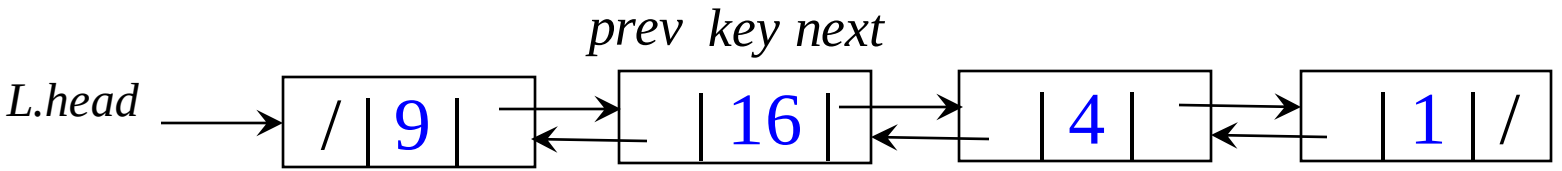




Doubly Linked List *L*

- *L* is a chain of nodes
  - ***head*** is the first node in the linked list
    - *L.head* points to the head
  - First node: *prev* = `NIL`; Last node: *next* = `NIL`
  - Consecutive nodes *x* & *y* should satisfy:

$$x.next = y \quad \text{and} \quad y.prev = x$$

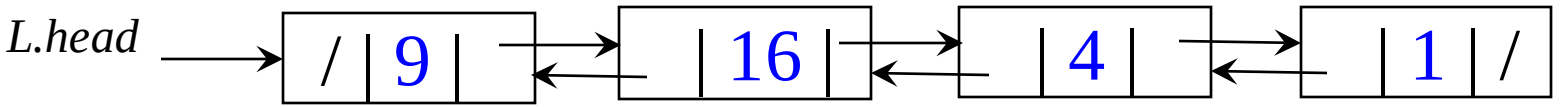


Doubly Linked List *L*

- Search key *k* in doubly linked list *L*

search (*L*, *k*)

1. $x \leftarrow L.head$
2. while $x \neq$ `NIL` and $x.key \neq k$
3. $x \leftarrow x.next$
4. return *x*

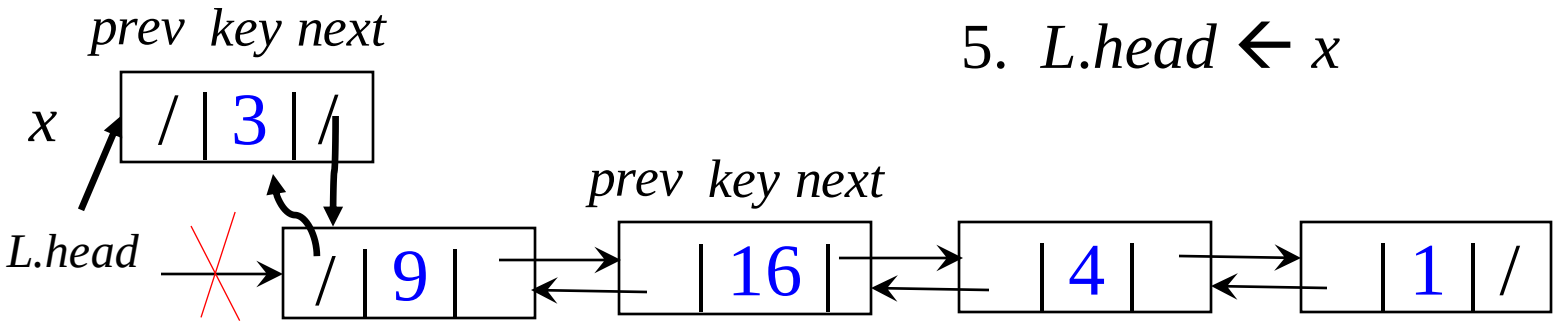


Doubly Linked List *L*

- Insert node *x* with key *k* to the front of the linked list *L*

Prepend-insert (*L*, *x*)

1. *x.prev* ← NIL
2. *x.next* ← *L.head*
3. if *L.head* ≠ NIL
4. *L.head.prev* ← *x*
5. *L.head* ← *x*

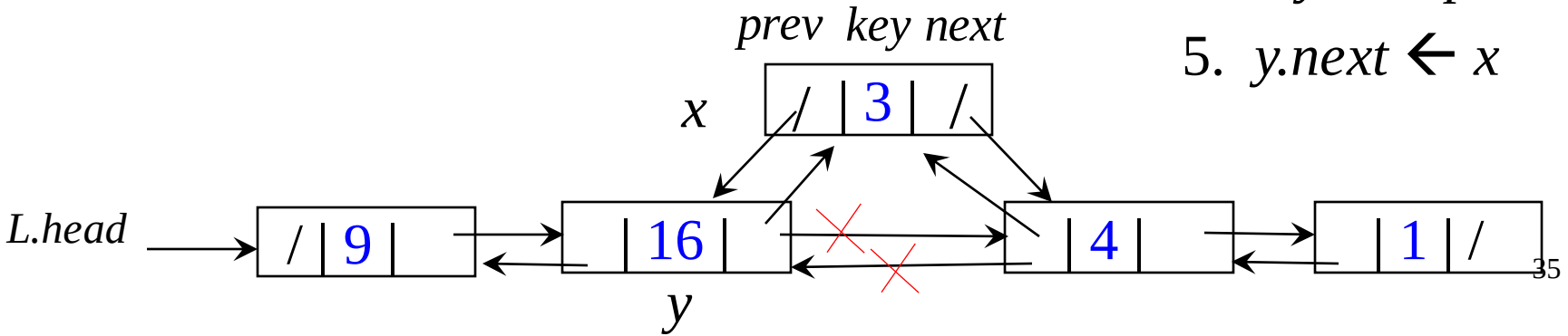


Doubly Linked List *L*

- Insert node *x* after y in the linked list *L*

Insert (*L*, *x*, *y*)

1. *x.prev* ← y
2. *x.next* ← *y.next*
3. if *y.next* ≠ NIL
4. *y.next.prev* ← *x*
5. *y.next* ← *x*

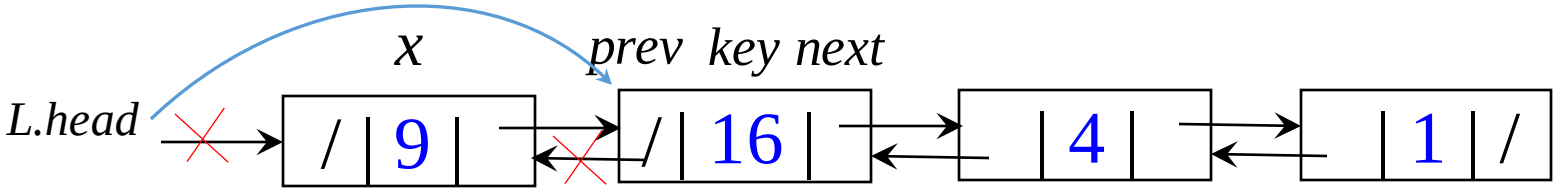


Doubly Linked List *L*

- Delete *node x* from *L*
  - Assume *x* exists in *L*
  - (If we want to remove a node with key *k*, we need to search it first)

Delete (*L*, *x*)

1. if *x.prev* ≠ `NIL`
2. *x.prev.next* ← *x.next*
3. else *L*.head ← *x.next*
4. if *x.next* ≠ `NIL`
5. *x.next.prev* ← *x.prev*

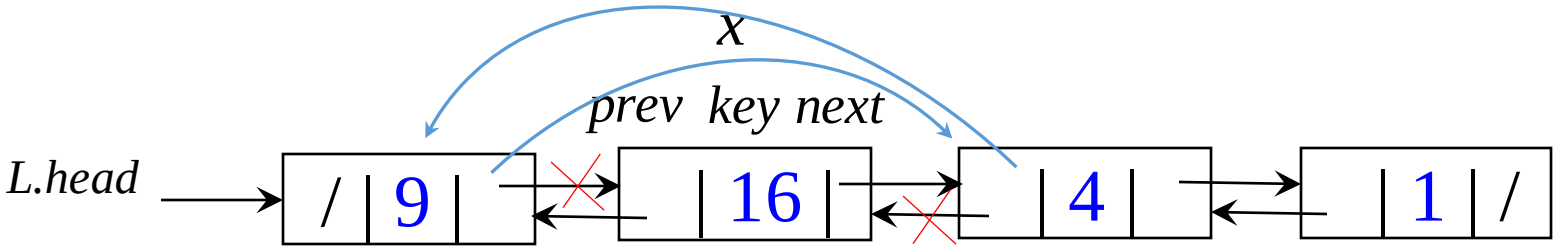


Doubly Linked List *L*

- Delete *node x* from *L*
  - Assume *x* exists in *L*
  - (If we want to remove a node with key *k*, we need to search it first)

Delete (*L*, *x*)

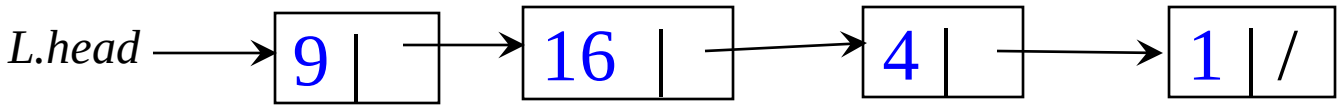
1. if *x.prev* ≠ `NIL`
2. *x.prev.next* ← *x.next*
3. else *L*.head ← *x.next*
4. if *x.next* ≠ `NIL`
5. *x.next.prev* ← *x.prev*



(Singly) Linked List

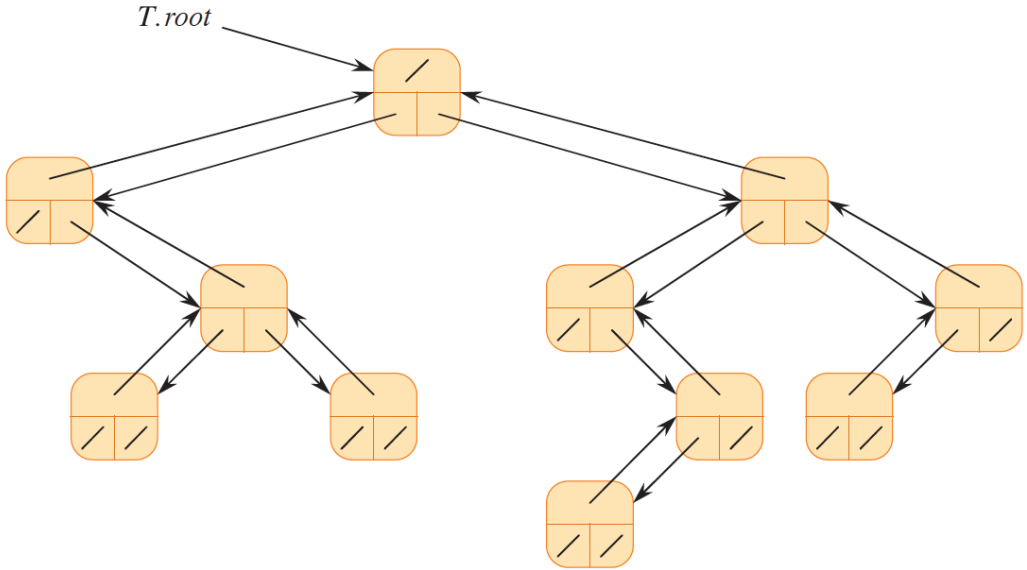
- In a singly linked list, each node stores *key*, *next* only
  - Require less space ☺
  - More complicated algorithm for deletion ☹

```
class Node {
  int key;
  Node* next;
}
```



Representing rooted trees by linked lists

- Binary tree




```
class Node {
   int key;
   Node* parent;
   Node* left;
   Node* right;
}
```

- Thank you!