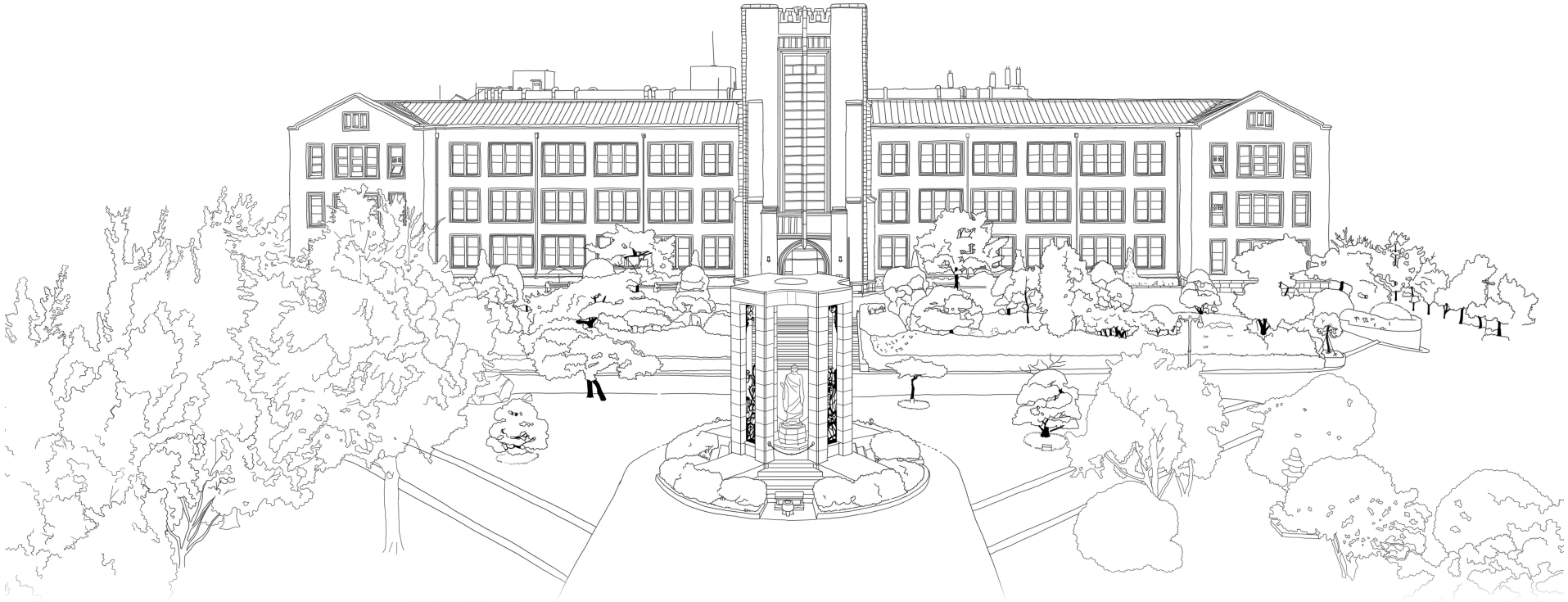


Introduction to Algorithms

DongYeon Kim
Department of Multimedia Engineering
Dongguk University

BigDataLab
Dept. of Multimedia Engineering at Dongguk University

dongguk UNIVERSITY

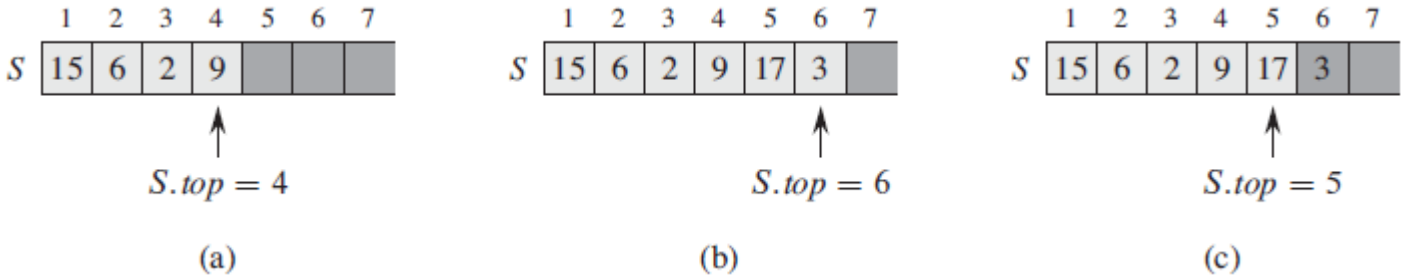
Contents

❖ Chapter.3 Data Structures

❖ Stacks and queues

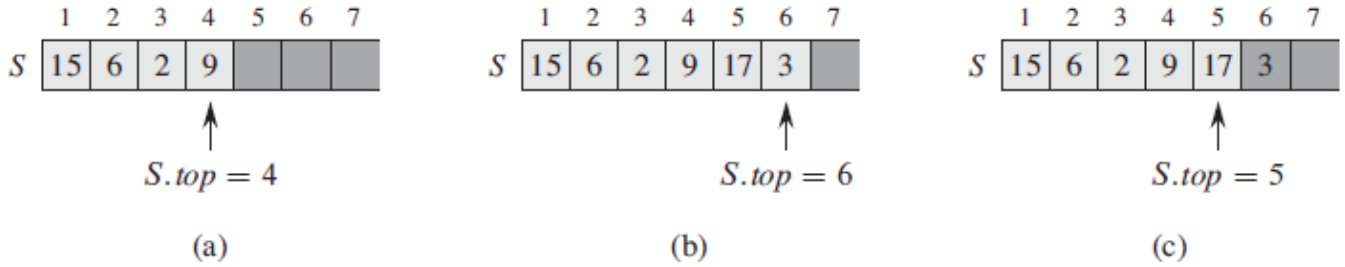
- Introduction
  - ➢ Stack : the element deleted from the set is the one most recently inserted
    - ✓ Last In First Out → ***LIFO***
  - ➢ Queue : the element deleted is the one that has been in the set for the longest time
    - ✓ First In First Out → ***FIFO***

- Stack
  - ➢ The insert operation on a stack is often called ***PUSH***
  - ➢ The delete operation on a stack is called ***POP***
  - ➢ We can implement a stack of $n-1$ elements with an array $S[1..n]$



❖ Stacks and queues

- Stack



➢ Array has an attribute $S.top$

✓ Indexes the most recently inserted element

✓ $S[1 \ldots n] = S[1 \ldots S.top]$

✓ $S[1]$ is the bottom element and $S[S.top]$ is the element at the top

✓ $S.top = 0$ → stack is ***empty*** / $S.top$ exceed n → stack ***overflow*** / pop empty stack → stack ***underflow***

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

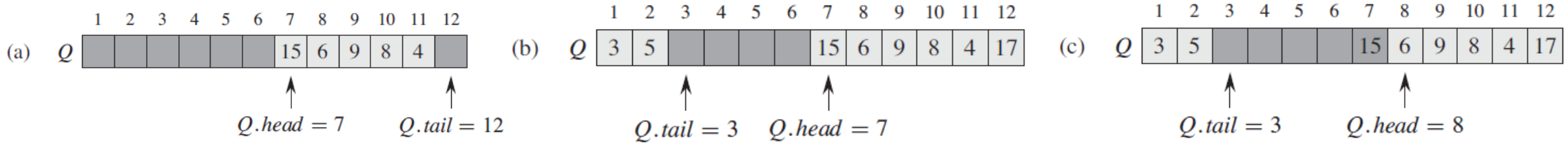
```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

→ each stack operation takes O(1) times

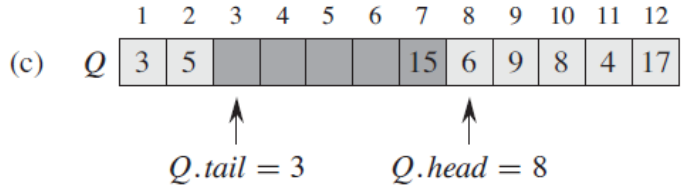
❖ Stacks and queues

- Queue
  - ➢ The insert operation on a queue is often called ***ENQUEUE***
  - ➢ The delete operation on a queue is called ***DEQUEUE***
  - ➢ We can implement a queue of $n-1$ elements with an array $Q[1..n]$
  - ➢ Array has an attribute $Q.\boldsymbol{head}$ and $Q.\boldsymbol{tail}$
    - ✓ $Q.head$: indexes or points its head
    - ✓ $Q.tail$: indexes the next location at which a newly arriving element will be inserted
    - ✓ When an element is enqueued, it takes its place at the tail of the queue $(a \rightarrow b)$
    - ✓ The element dequeued is always the one at the head of the queue $(b \rightarrow c)$



❖ Stacks and queues

- Queue
  - ➢ We can implement queue using **"wrap around"** method
    - ✓ Location n immediately follows location 1 in a circular order



    - ✓ The elements in the queue reside in locations $Q.head, Q.head + 1, .., Q.tail - 1$
  - ➢ When $Q.head = Q.tail$, the queue is empty
    - ✓ If we attempt to dequeue an element from an empty queue the queue ***underflows***
  - ➢ When $Q.head = Q.tail + 1$, the queue is full
    - ✓ If we attempt to enqueue and element, then the queue ***overflows***

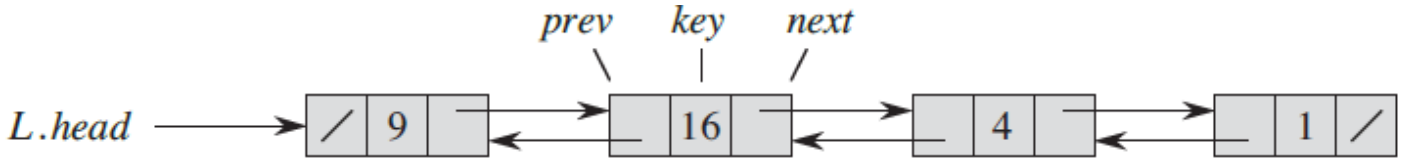
```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

→ each queue operation takes O(1) times

❖ Linked-List

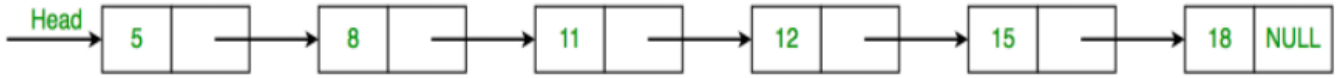
- definition
  - ➢ The ***linked list*** is a data structure in which the objects are arranged in a linear order
  - ➢ The order in a linked list is determined by a pointer in each object



  - ➢ The ***doubly linked list*** L is an object with an attribute $key,\ next,\ prev$
    - ✓ $x.next$ points to its successor
    - ✓ $x.prev$ points to its predecessor
    - ✓ If $x.prev = NIL$ → the element x has no predecessor → the first element (***head***)
    - ✓ If $x.next = NIL$ → the element x has no successor → the last element (***tail***)
    - ✓ $L.head$ points to the first element of the list
    - ✓ If $L.head = NIL$, the list is empty

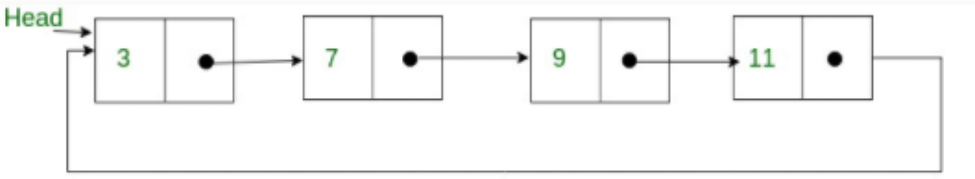
❖ Linked-List

- The form of linked list
  - ➢ The ***singly linked list*** : omit $prev$ pointer in each element
  - ➢ The ***sorted linked list*** : linear order of the list corresponds to the linear order of keys
    - ✓ the minimum element = the head of the list / the maximum element = the tail of the list



  - ➢ The ***unsorted linked list*** : the elements can appear in any order

  - ➢ The ***cirsular linked list***
    - ✓ the $prev$ pointer of the head of the list points to the tail
    - ✓ The $next$ pointer of the tail points to the head



❖ Linked-List

- Searching a linked list

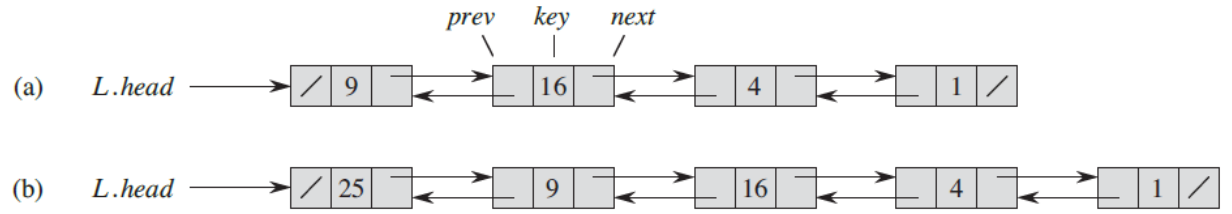
```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

- $line\ 1 \sim 3$: Find the first element with key k in list L by a simple linear search
- $line\ 4$: Return the pointer to this element
- If there is no element in list, return NIL

- In the worst case the $LIST - SEARCH$ procedure takes $\theta(n)$ time
  - ✓ Since it may have to search the entire list

- Inserting into a linked list

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

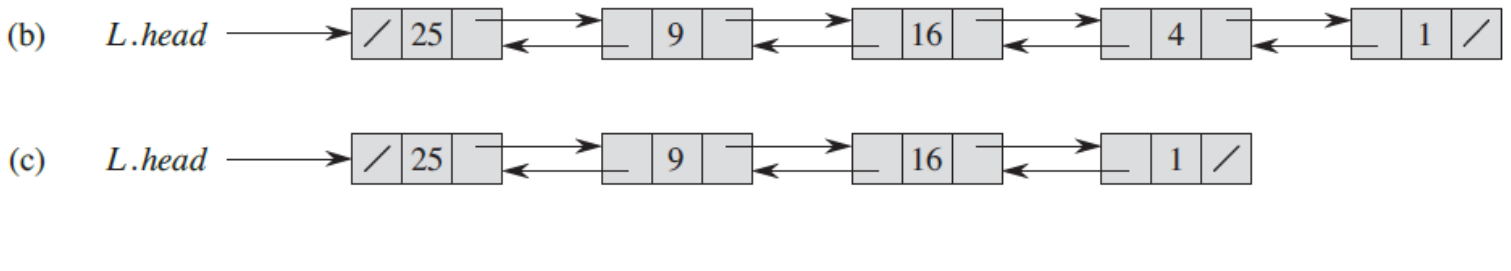


- splices x onto the front of the linked list
- The running time for $LIST - INSERT$ on a list of n elements is $O(1)$

❖ Linked-List

- Deleting from a linked list

```
LIST-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
```



- Removes an element x from a linked list L
- Must be given a pointer to x, and it splices x out of the list by updating pointers
  - ✓ We must first call $LIST-SEARCH$ to retrieve a pointer to the element
- The running time of $LIST-DELETE$ is $O(1)$
  - ✓ Deleting an element with a given key → $\theta(n)$ time is required

❖ Linked-List

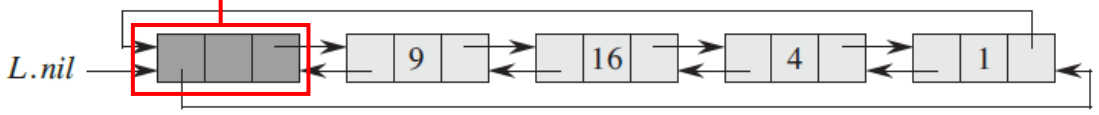
- **Definition of Sentinels**
  - ➢ The code for $LIST - DELETE$ would be simpler if we could ignore the boundary conditions at the head and tail of the list

```
LIST-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
```

→

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

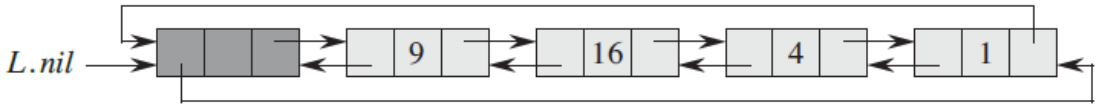
  - ➢ The ***sentinel*** is a dummy object that allows us to simplify boundary conditions
  - ➢ Provide with list L an object $L.nil$ that represents NIL



  - ➢ This change turns a ***regular doubly linked list*** into a ***circular doubly linked list*** with ***sentinel***
    - ✓ The ***sentinel*** $L.nil$ appears between the head and tail

❖ Linked-List

- Definition of Sentinels



➢ The attribute $L.head$ is no longer needed since we can access the head by $L.nil.next$

➢ $L.nil.next$ points to the head, $L.nil.prev$ points to the tail

➢ $next$ attribute of tail, $prev$ attribute of head points to $L.nil$

- ***LIST − SEARCH***

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

→

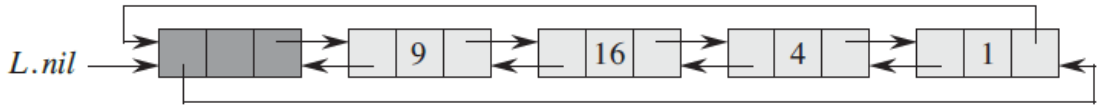
```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

✓ Remains the same as before

✓ The references to NIL and $L.head$ changed

❖ Linked-List

- ***LIST − INSERT***




```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

→

```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

✓ ***sentinel*** rarely reduce the asymptotic time bounds of data structure operations

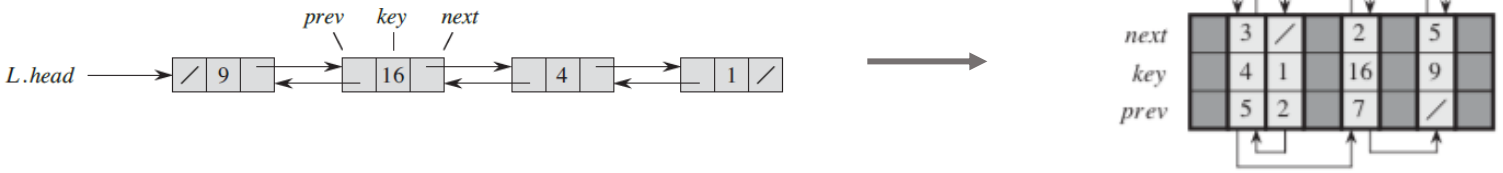
✓ They can reduce constant factors

✓ The gain from using ***sentinel*** within loops is for clarity of code rather than speed

❖ Implementing pointers & objects

- Introduction
  - ➢ In this chapter, introduce two ways of implementing linked data structures without using pointer data type
    - ✓ There are languages that do not provide pointers and objects

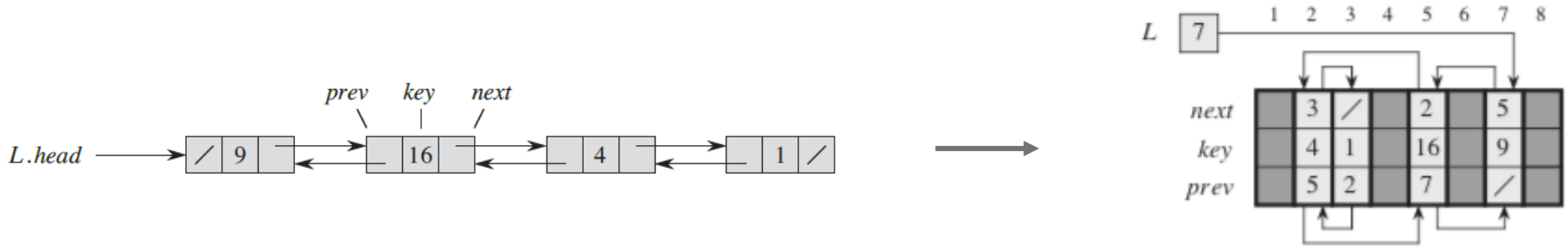
- A multiple array representation of objects



  - ➢ Represent a collection of objects that have the same attributes by using an array for each attribute
  - ➢ key array : holds the values of the keys currently in the dynamic set
  - ➢ $next, prev$ array : represent the pointers

❖ Implementing pointers & objects

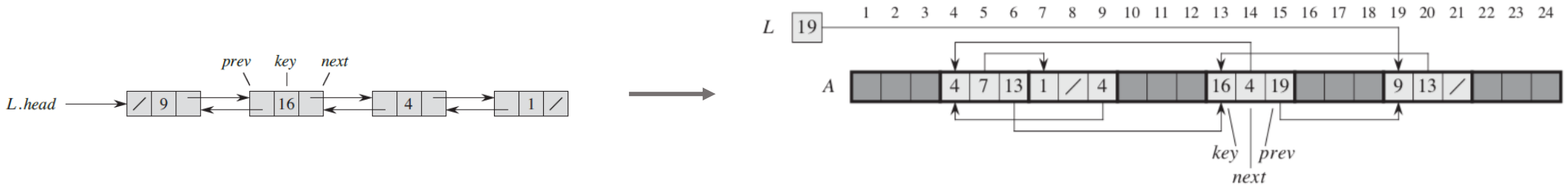
- A multiple array representation of objects



- The array entries $key[x], next[x], prev[x]$ represent an object in the linked list
- $prev$ of $head$ or next of tail → NIL
  - ✓ Represent using integer such as −1
- The variable L represents the index of the head of the list

❖ Implementing pointers & objects

- A single array representation of objects



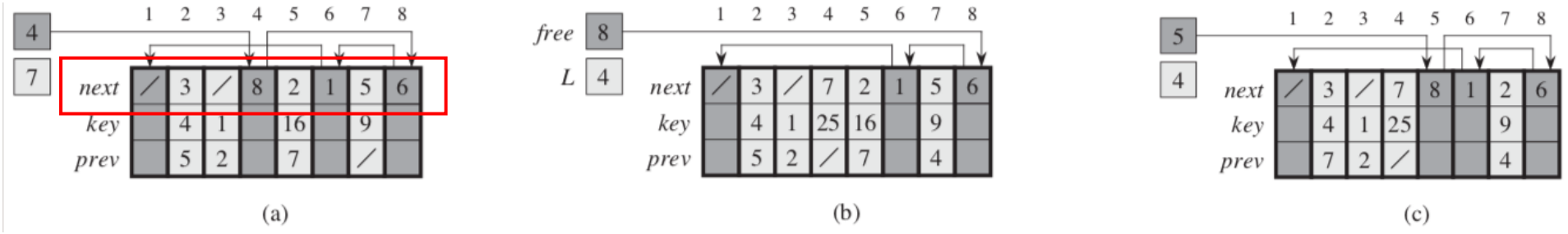
  - ➢ An object occupies a contiguous subarray $A[j..k]$
  - ➢ Each attribute of the object corresponds to an offset in the range from 0 to $k - j$
  - ➢ In this case, the offset corresponding to $key, next, prev$ → 0, 1, 2
    - ✓ $i.prev \rightarrow A[i+2]$ / $i.next \rightarrow A[i+1]$ / $i.key \rightarrow A[i]$

- Allocating and freeing objects
  - ➢ To insert a key into a dynamic set, we must allocate a pointer to a currently unused object in the linked-list representation
  - ➢ It is useful to manage the storage of objects not currently used in the linked list
  - ➢ $Garbage\ collector$ is responsible for determining which objects are unused

    → now explore the problem of allocating and freeing objects using $garbage\ collector$

❖ Implementing pointers & objects

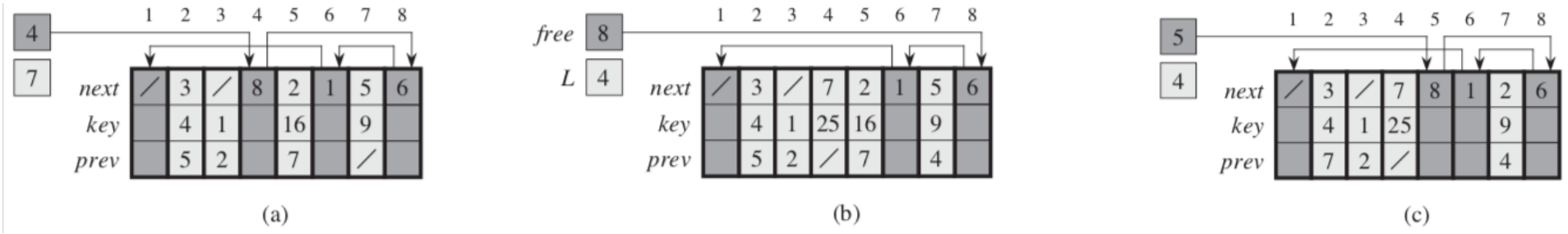
- **Allocating and freeing objects**
  - ➢ Suppose that the arrays in the multiple-array representation have length m
    - ✓ The dynamic set contains $n \leq m$ elements
    - ✓ n objects represent elements currently in dynamic set
    - ✓ $m - n$ objects are free → new elements will be inserted
  - ➢ Keep the free objects in a singly linked list, which we call the free list



  - ✓ Use only next pointer array
  - ✓ The head of the free list is held in the global variable free
  - ✓ The free list acts like a stack

❖ Implementing pointers & objects

- Allocating and freeing objects
  - ➢ $ALLOCATE-OBJECT(\ \ )$ & $FREE-OBJECT(x)$




```
ALLOCATE-OBJECT()
1  if free == NIL
2       error "out of space"
3  else x = free
4       free = x.next
5       return x
```

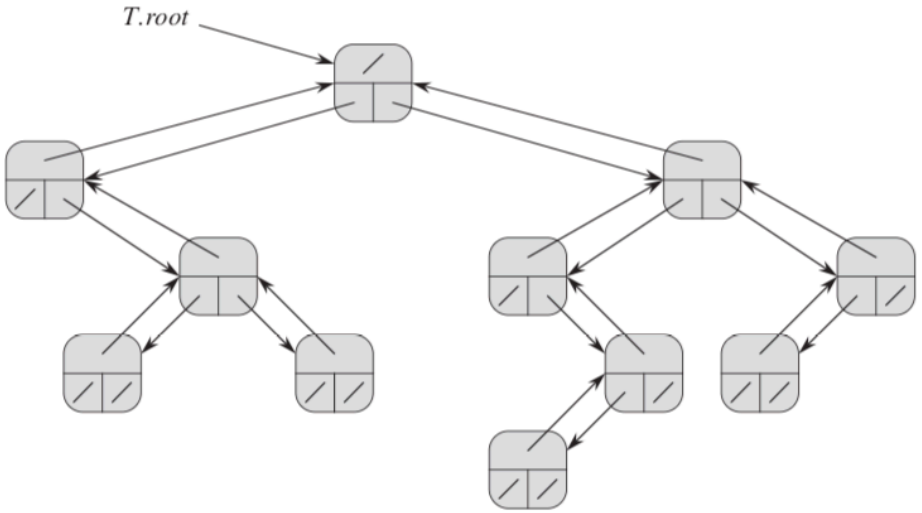
- ➢ $line\ 1\sim 2$: the list is full
- ➢ $line\ 4$: Return the pointer to this element

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

- ➢ $line\ 1\sim 2$: the free list acts like a stack → LIFO

❖ Representing Rooted Trees

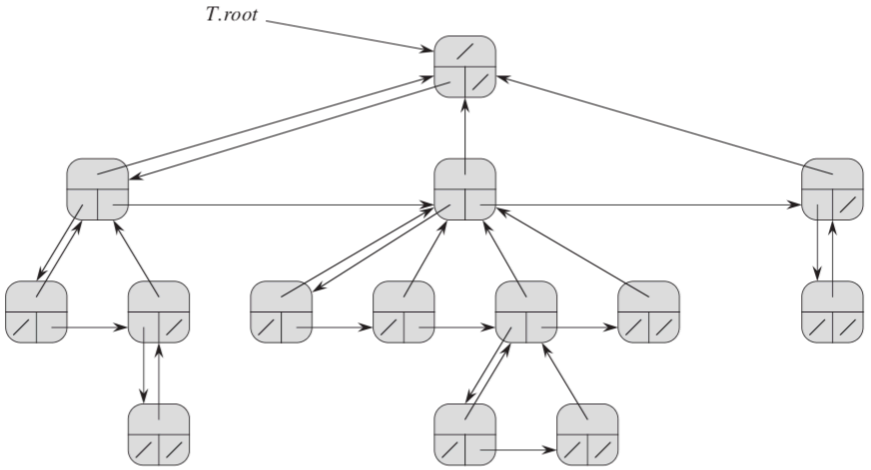
- Introduction
  - ➢ In this section, we look specifically at the problem of representing rooted trees by linked data structure
  - ➢ Represent each node of a tree by an object
    - ✓ Each node contains a key attribute
    - ✓ Pointer attribute are vary according to the type of tree
- Binary tree



- ➢ $x.p$: pointer to parent
- ➢ $x.left$: pointer to left child
- ➢ $x.right$: pointer to right child
- ➢ $x.p = NIL$: x is the root node
- ➢ $x.left = NIL$ or $x.right = NIL$: no left or right child node
- ➢ The root of the entire tree T is pointed to by the attribute $T.root$

❖ Representing Rooted Trees

- Rooted trees with unbounded branching



➢ Extend the method of a binary tree
- ✓ The number of children of each node is at most some constant k
- ✓ Replace the left and right attributes by $child_1$, $child_2$, …, $child_k$
- ✓ It may waste memory space

➢ There is a clever method to represent trees with only two pointers

1. $x.left\text{-}child$ points to the leftmost child of node x, and
2. $x.right\text{-}sibling$ points to the sibling of x immediately to its right.

➢ if $x.left - child = NIL$: node x has no children

➢ If $x.right - sibling = NIL$: node x is the rightmost child of its parents

Thank You!