UMC 201: Data Structures & Algorithms

Lecture 7: Elementary Data Structures

Indian Institute of Science

Recap

- In the last few lectures, we discussed several algorithms to solve the sorting problem.
- These algorithms use elementary data structures such as *arrays* & *linked lists* (remember bucket sort?).

Recap

- In the last few lectures, we discussed several algorithms to solve the sorting problem.
- These algorithms use elementary data structures such as *arrays* & *linked lists* (remember bucket sort?).

- In today's lecture, we'll discuss these and other elementary data structures that are *widely used to store and organize data in order to facilitate access and modifications,* which let us design efficient algorithms.

Arrays and Matrices

Arrays

- We assume that an array is stored as a *contiguous sequence of bytes* in memory.
- If the first element of an array has index s, the array starts at memory address a, and each array element occupies b bytes, then the i-th element occupies bytes $a + b(i - s)$ through $a + b(i - s) + b - 1$.

Arrays

- We assume that an array is stored as a *contiguous sequence of bytes* in memory.
- If the first element of an array has index s, the array starts at memory address a, and each array element occupies b bytes, then the i-th element occupies bytes $a + b(i - s)$ through $a + b(i - s) + b - 1$.
- Assuming that the computer can access all memory locations in the same amount of time (as in the RAM model), it *takes constant time to access any array element*, regardless of the index.

Arrays

- If the elements of a given array might occupy *different numbers of bytes*, then the formula given before fails to apply, since the element size b is not a constant.
- In such cases, the array elements are usually objects of varying sizes and *what actually appears in each array element is a pointer to the object*.

Arrays

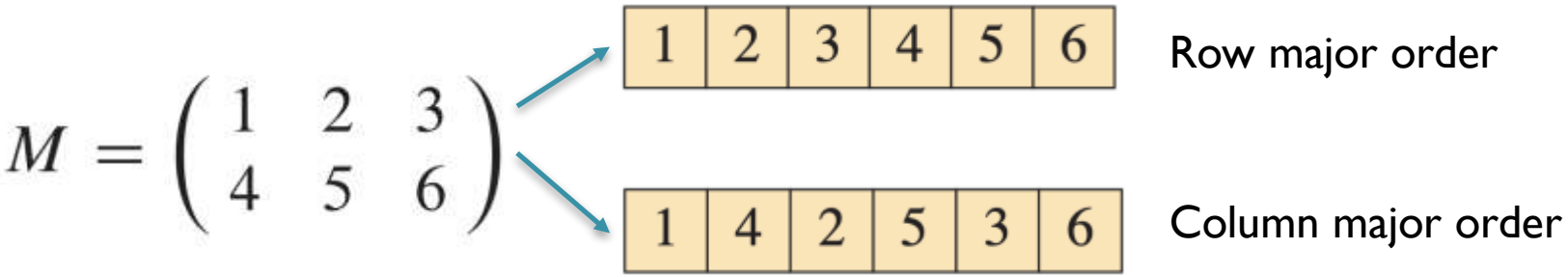
- If the elements of a given array might occupy *different numbers of bytes*, then the formula given before fails to apply, since the element size b is not a constant.
- In such cases, the array elements are usually objects of varying sizes and *what actually appears in each array element is a pointer to the object*.
- The number of bytes occupied by a pointer is typically the *same*, no matter what the pointer references.

Matrices

- We typically represent a matrix by *one or more one-dimensional arrays*. The two most common ways to store a matrix are row-major order and column-major order. Consider an $m \times n$ matrix.
- In row major order, the matrix is stored *row by row in a single array*, and in column-major order, the matrix is stored *column by column in a single array*.

Matrices

- We typically represent a matrix by *one or more one-dimensional arrays*. The two most common ways to store a matrix are row-major order and column-major order. Consider an $m \times n$ matrix.
- In row major order, the matrix is stored *row by row in a single array*, and in column-major order, the matrix is stored *column by column in a single array*.

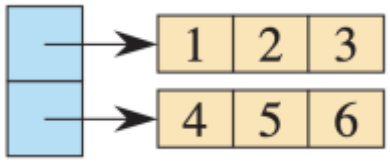


Matrices

- We typically represent a matrix by *one or more one-dimensional arrays*. The two most common ways to store a matrix are row-major order and column-major order. Consider an $m \times n$ matrix.
- In row major order, the matrix is stored *row by row in a single array*, and in column-major order, the matrix is stored *column by column in a single array*.
- If the array is indexed starting at 1, then $M[i, j]$ is at index $n(i-1)+j$ with *row-major order* and $m(j-1)+i$ with *column-major order*.

Matrices

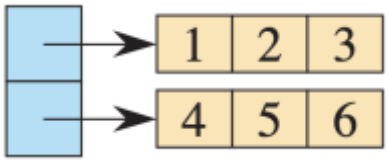
- We can also store a matrix using *multiple arrays*.



- In the *row-major version*, each row is stored in its own array of length n.
- Another array, with m elements, shown in blue, points to the m row arrays.

Matrices

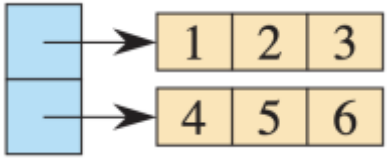
- We can also store a matrix using *multiple arrays*.



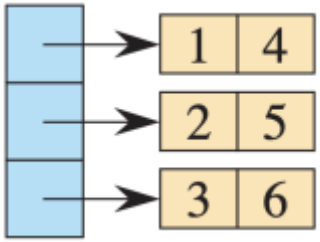
  - In the *row-major version*, each row is stored in its own array of length n.
- Another array, with m elements, shown in blue, points to the m row arrays.
- If we call the blue array A, then $A[i]$ points to the array storing the entries for row i of M, and array element $A[i][j]$ stores matrix element $M[i, j]$.

Matrices

- We can also store a matrix using *multiple arrays*.



- In the *row-major version*, each row is stored in its own array of length n.



- Similarly, in the *column-major version*, each column is stored in its own array of length m.

Stacks and Queues

Stacks & Queues: Delete operation

- Stacks & queues are *dynamic sets* in which the element removed from the set by the DELETE operation is prespecified.
- In a stack, the element deleted from the set is the *one most recently inserted*: the stack implements a last-in, first-out, or LIFO, policy.

Stacks & Queues: Delete operation

- Stacks & queues are *dynamic sets* in which the element removed from the set by the DELETE operation is prespecified.
- In a stack, the element deleted from the set is the *one most recently inserted*: the stack implements a last-in, first-out, or LIFO, policy.
- In a queue, the element deleted is always the *one that has been in the set for the longest:* the queue implements a first-in, first-out, or FIFO, policy.

Stacks & Queues: Delete operation

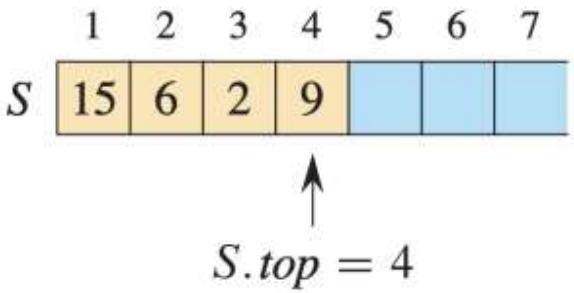
- Stacks & queues are *dynamic sets* in which the element removed from the set by the DELETE operation is prespecified.
- In a stack, the element deleted from the set is the *one most recently inserted*: the stack implements a last-in, first-out, or LIFO, policy.
- In a queue, the element deleted is always the *one that has been in the set for the longest:* the queue implements a first-in, first-out, or FIFO, policy.
- We'll *use arrays* to implement stacks and queues.

Stacks using arrays

- We can implement a stack of at most n elements with an array $S[1:n]$ *with attributes*.
- The stack has *attributes* $S.top$, which indexes the most recently inserted element, and $S.size$, which equals the size n of the array.

Stacks using arrays

- We can implement a stack of at most n elements with an array $S[1:n]$ *with attributes*.
- The stack has *attributes* $S.top$, which indexes the most recently inserted element, and $S.size$, which equals the size n of the array.
- The stack consists of elements $S[1:S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top.



Stack operations

- The INSERT operation on a stack is often called PUSH, and the DELETE operation, which *returns the top element*, is called a POP.

Stack operations

- The INSERT operation on a stack is often called PUSH, and the DELETE operation, which *returns the top element*, is called a POP.
- When $S.top = 0$, the stack is empty. We can test whether the stack is empty with the query operation STACK-EMPTY (pseudocode given in the next slide).

Stack operations

- The INSERT operation on a stack is often called PUSH, and the DELETE operation, which *returns the top element*, is called a POP.
- When $S.top = 0$, the stack is empty. We can test whether the stack is empty with the query operation STACK-EMPTY (pseudocode given in the next slide).
- Upon an attempt to pop an empty stack, the stack underflows, which is an *error*. If $S.top$ exceeds $S.size$, the stack overflows, which is also an *error*.

Stack operations: Pseudocodes

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

```
PUSH(S, x)
1  if S.top == S.size
2        error "overflow"
3  else S.top = S.top + 1
4        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]
```

Stack operations: Time complexity

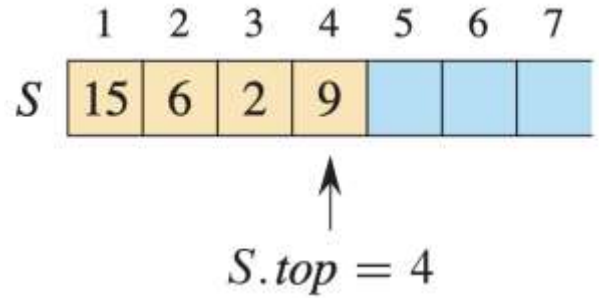
```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  if S.top == S.size
2       error "overflow"
3  else S.top = S.top + 1
4       S[S.top] = x
```

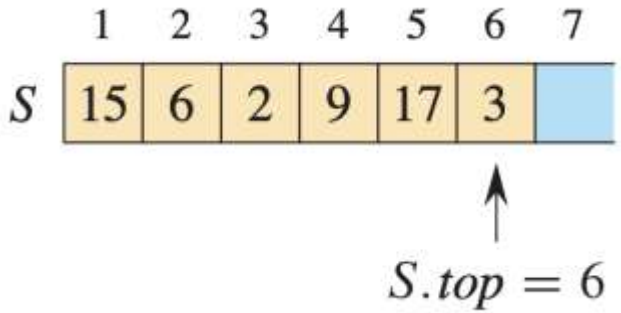
```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

- Each of the three stack operations takes $O(1)$ time.

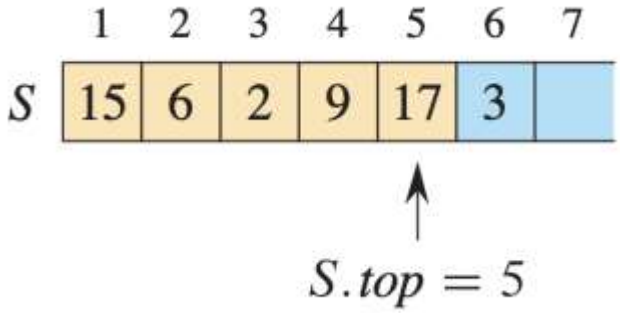
Stack operations: Examples



- Original stack



- After calls PUSH(S, 17) & PUSH(S, 3)



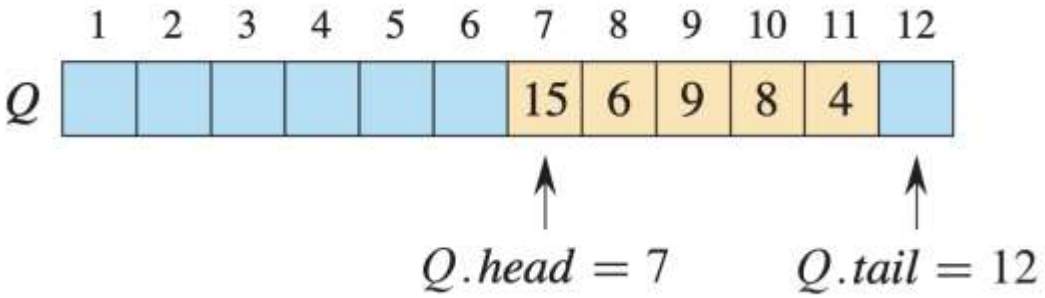
- After one POP call, which *returns the element* 3.

Queues using arrays

- We can implement a queue of at most $n-1$ elements with an array $Q[1:n]$ *with attributes*. A queue has a head and a tail.
- The attribute $Q.size$ equals the size n of the array. The attribute $Q.head$ indexes, or points to, the queue's head. And the attribute $Q.tail$ indexes the *next location at which a new element will be inserted*.

Queues using arrays

- We can implement a queue of at most $n-1$ elements with an array $Q[1:n]$ *with attributes*. A queue has a head and a tail.
- The attribute $Q.size$ equals the size n of the array. The attribute $Q.head$ indexes, or points to, the queue's head. And the attribute $Q.tail$ indexes the *next location at which a new element will be inserted*.



Queues using arrays

- We can implement a queue of at most $n-1$ elements with an array $Q[1:n]$ *with attributes*. A queue has a head and a tail.
- The attribute $Q.size$ equals the size n of the array. The attribute $Q.head$ indexes, or points to, the queue's head. And the attribute $Q.tail$ indexes the *next location at which a new element will be inserted*.
- The elements reside in $Q.head,\ Q.head+1, \ldots, Q.tail-1$, where we "*wrap arround*", i.e., location 1 follows location n in a *circular order*.

Queue operations

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE, which *returns the element at the head*.

Queue operations

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE, which *returns the element at the head*.
- The FIFO property of a queue causes it to operate like a *line of customers waiting for service*. When an element is enqueued, it's *added at the tail of the queue*.

Queue operations

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE, which *returns the element at the head*.
- The FIFO property of a queue causes it to operate like a *line of customers waiting for service*. When an element is enqueued, it's *added at the tail of the queue*.
- When $Q.head = Q.tail$, the queue is *empty*. Initially, $Q.head = Q.tail = 1$. An attempt to dequeue an element from an empty queue causes "underflow".

Queue operations

- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE, which *returns the element at the head*.
- The FIFO property of a queue causes it to operate like a *line of customers waiting for service*. When an element is enqueued, it's *added at the tail of the queue*.
- When $Q.head = Q.tail + 1$ or both $Q.head = 1$ and $Q.tail = Q.size$, the queue is *full*, and an attempt to enqueue causes the queue to "overflow".

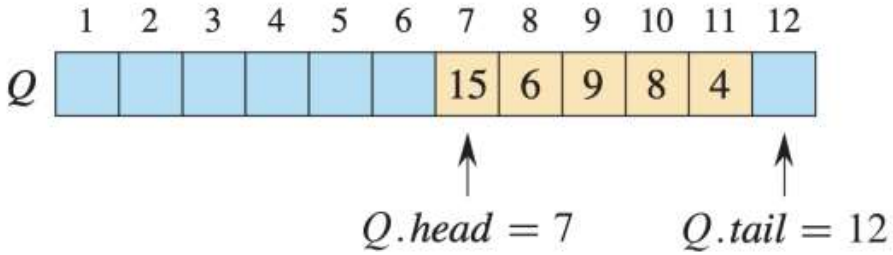
Queue operations: Pseudocodes

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

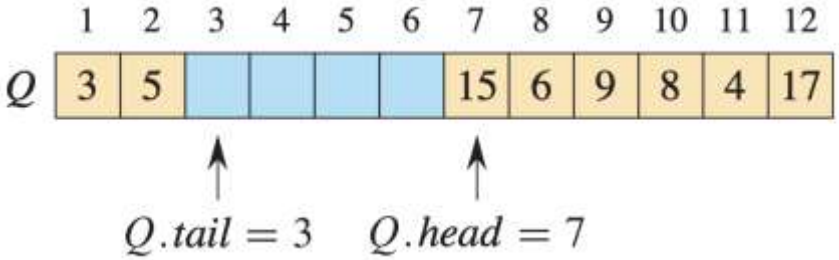
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

- In the above procedures, we have *omitted the error checking* for "underflow" and "overflow".
- Each operation takes $O(1)$ time.

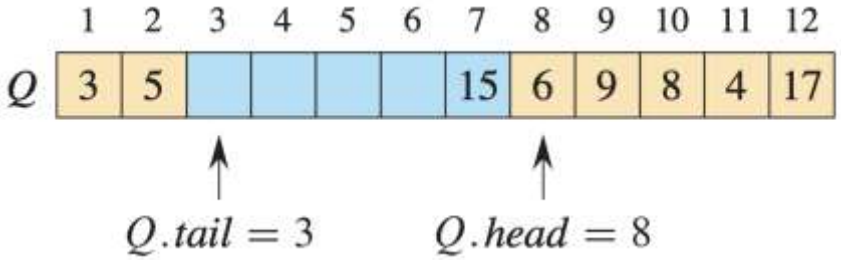
Queue operations: Examples



- Original queue



- After calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5)



- Call DEQUEUE(Q) returns element 15.

Linked lists

Linked lists

- A linked list is a data structure in which the *objects* are arranged in a *linear order*.
- Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by *a pointer in each object*.

Linked lists

- A linked list is a data structure in which the *objects* are arranged in a *linear order*.
- Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by *a pointer in each object*.
- Since the elements of linked lists often contain *keys that can be searched for*, linked lists are sometimes called search lists.
- Linked lists provide a *simple, flexible representation* for dynamic sets.

Doubly linked lists

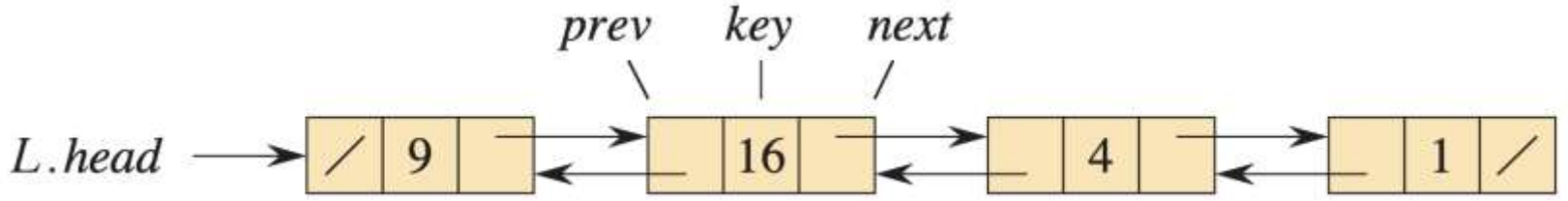
- Each element of a doubly linked list L is an *object* with an attribute key and two *pointer attributes*: next and prev. The object may contain other *satellite data*.
- Given an element x in L, $x.next$ points to its *successor* in L, and $x.prev$ points to its *predecessor*.

Doubly linked lists

- Each element of a doubly linked list L is an *object* with an attribute key and two *pointer attributes*: next and prev. The object may contain other *satellite data*.
- Given an element x in L, $x.next$ points to its *successor* in L, and $x.prev$ points to its *predecessor*.
- If $x.prev = NIL$, the element x is the *head* of L. If $x.next = NIL$, the element x is the *tail* of L.
- An attribute $L.head$ *points to the first element* of the list. If $L.head = NIL$, the list is *empty*.

Doubly linked lists

- Each element of a doubly linked list L is an *object* with an attribute key and two *pointer attributes*: next and prev. The object may contain other *satellite data*.
- Given an element x in L, $x.next$ points to its *successor* in L, and $x.prev$ points to its *predecessor*.
- If $x.prev = NIL$, the element x is the *head* of L. If $x.next = NIL$, the element x is the *tail* of L.



Various kinds of lists

- A list may be either *singly or doubly linked*, it may be *sorted or not*, and it may be *circular or not*.
- In a singly linked list, each element has a $next$ pointer but *not* a $prev$ pointer.

Various kinds of lists

- A list may be either *singly or doubly linked*, it may be *sorted or not*, and it may be *circular or not*.
- In a singly linked list, each element has a $next$ pointer but *not* a $prev$ pointer.
- If a list is sorted, the linear order of the list corresponds to the *linear order of the keys* stored in the elements of the list.

Various kinds of lists

- A list may be either *singly or doubly linked*, it may be *sorted or not*, and it may be *circular or not*.
- In a singly linked list, each element has a $next$ pointer but *not* a $prev$ pointer.
- If a list is sorted, the linear order of the list corresponds to the *linear order of the keys* stored in the elements of the list.
- In a circular list, the $prev$ pointer of the head of the list *points to the tail*, and the $next$ pointer of the tail *points to the head*.

Various kinds of lists

- A list may be either *singly or doubly linked*, it may be *sorted or not*, and it may be *circular or not*.
- In a singly linked list, each element has a $next$ pointer but *not* a $prev$ pointer.
- If a list is sorted, the linear order of the list corresponds to the *linear order of the keys* stored in the elements of the list.
- In this lecture, we will assume that the lists are *unsorted and doubly linked*.

List operation: Search

- The procedure below finds the *first* element with key k in list L and *returns a pointer* to this element.

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

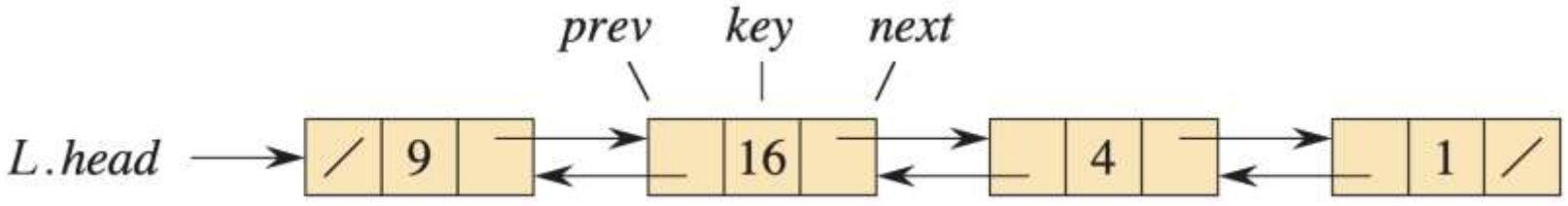
- If *no* object with key k appears, then the procedure returns NIL.

List operation: Search

- The procedure below finds the *first* element with key k in list L and *returns a pointer* to this element.

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

- If *no* object with key k appears, then the procedure returns NIL.



- The call LIST-SEARCH(L, 4) returns a pointer to the *third* element & call LIST-SEARCH(L, 2) returns NIL.

List operation: Search

- The procedure below finds the *first* element with key k in list L and *returns a pointer* to this element.

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

- If *no* object with key k appears, then the procedure returns NIL.

- To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst-case.

List operation: Prepend

- Given an element x whose key attribute has already been set, the procedure below *adds x to the front* of L.

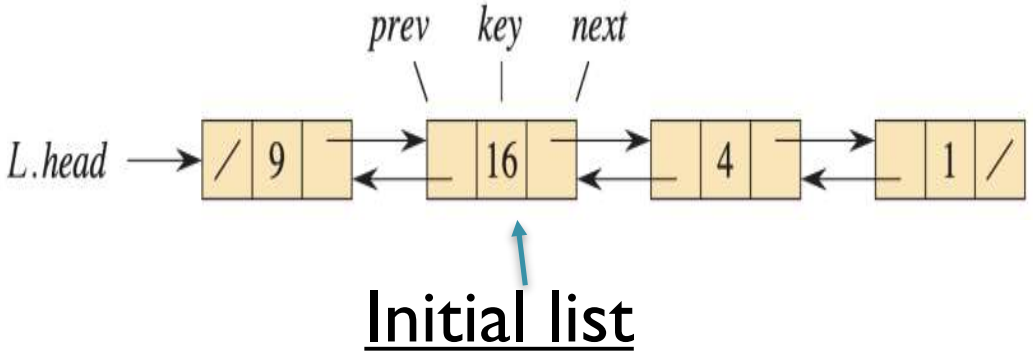
```
LIST-PREPEND(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4       L.head.prev = x
5  L.head = x
```

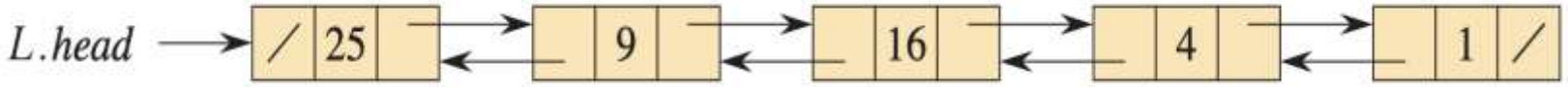
- The running time on a list of n elements is $O(1)$.

List operation: Prepend

- Given an element x whose key attribute has already been set, the procedure below *adds x to the front* of L.

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





- Following the execution of LIST-PREPEND(L, x), where $x.key = 25$.

List operation: Insert

- The following procedure inserts a new element x into the list, *immediately following* y, in $O(1)$ time.

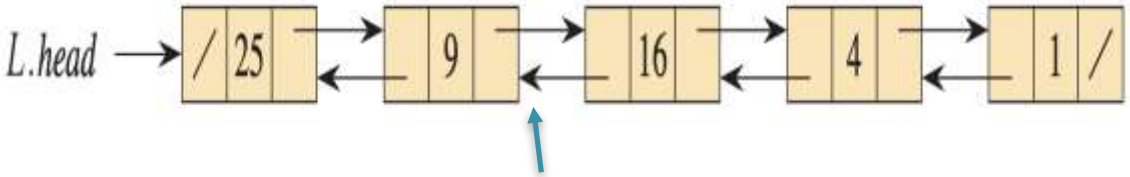
```
LIST-INSERT(x, y)
1   x.next = y.next
2   x.prev = y
3   if y.next ≠ NIL
4         y.next.prev = x
5   y.next = x
```

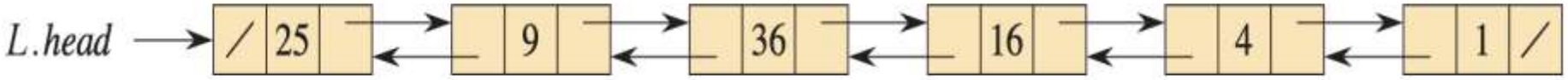
- Note. The procedure *does not need* to be supplied with the list L as a parameter.

List operation: Insert

- The following procedure inserts a new element x into the list, *immediately following* y, in $O(1)$ time.

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





- Following a call to LIST-INSERT(L, x, y), where $x.key = 36$ and y points to the object with key 9.

List operation: Delete

- The following procedure removes an element x from a linked list L in $O(1)$ time.

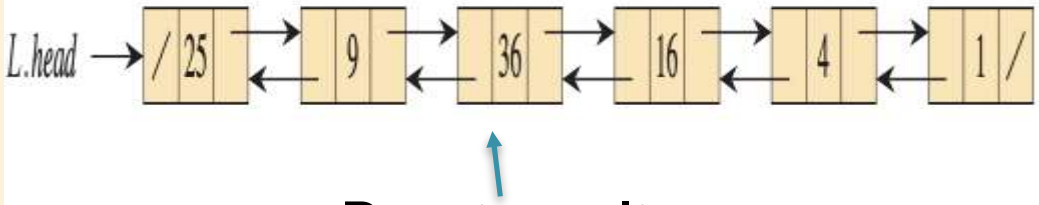
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

- To delete an element with a *given key*, first call LIST-SEARCH to retrieve a pointer to the element.

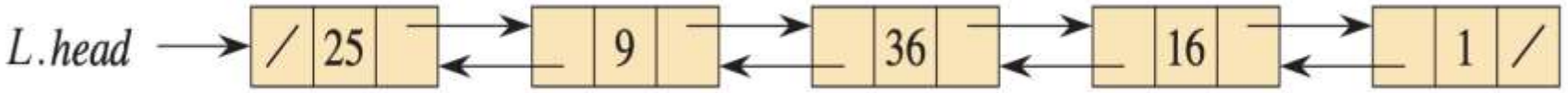
List operation: Delete

- The following procedure removes an element x from a linked list L in $O(1)$ time.

```
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
```



Previous list



- The result of the call LIST-DELETE(L, x), where x points to the object with key 4.

Comparison between lists and arrays

- Insertion and deletion are *faster operations* on doubly linked lists than on arrays.
- In order to *insert a new first element* into an array or *delete the first element* in an array, *maintaining the relative order of all the existing elements*, we need to move each of the other elements by one position.

Comparison between lists and arrays

- Insertion and deletion are *faster operations* on doubly linked lists than on arrays.
- Thus, insertion & deletion take $\Theta(n)$ time in an array, compared with $O(1)$ time for a doubly linked list.

Comparison between lists and arrays

- Insertion and deletion are *faster operations* on doubly linked lists than on arrays.
- Thus, insertion & deletion take $\Theta(n)$ time in an array, compared with $O(1)$ time for a doubly linked list.

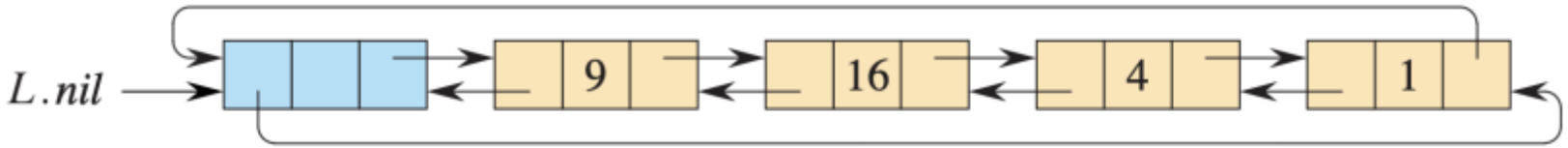
- If, however, we want to find the k-th element in the linear order, it takes just $O(1)$ time in an array regardless of k, but in a linked list, we have to traverse k elements, taking $\Theta(k)$ time.

Linked lists: Sentinels

- A sentinel is a dummy object that allows us to *simply boundary conditions*.
- In a linked list L, the sentinel is an object $L.nil$ that represents NIL but has *all the attributes of the other objects* in the list.

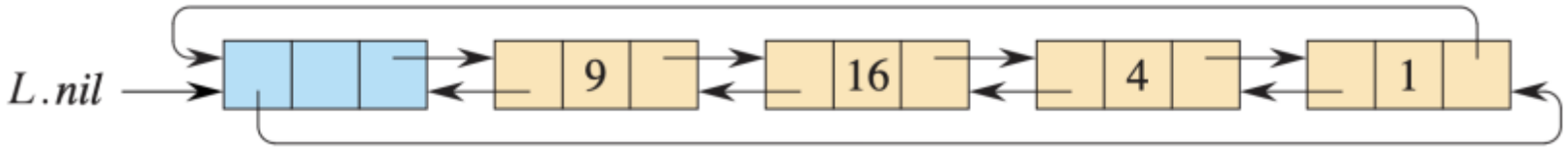
Linked lists: Sentinels

- A sentinel is a dummy object that allows us to *simply boundary conditions*.
- In a linked list L, the sentinel is an object $L.nil$ that represents NIL but has *all the attributes of the other objects* in the list.
- This change turns a regular doubly linked list into a circular, doubly linked list with a sentinel, in which the sentinel $L.nil$ lies between the head and tail.



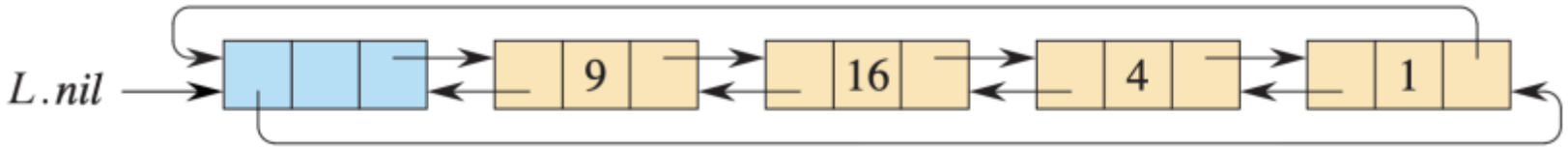
Linked lists: Sentinels

- The attribute $L.nil.next$ points to the *head* of the list, and $L.nil.prev$ points to the *tail*.
- Similarly, both the $next$ attribute of the tail and the $prev$ attribute of the head point to $L.nil$.



Linked lists: Sentinels

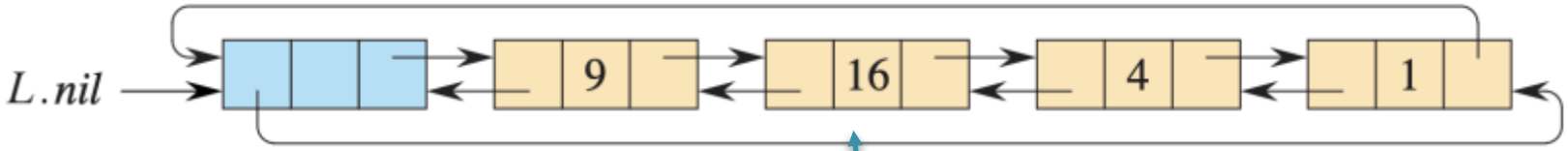
- The attribute $L.nil.next$ points to the *head* of the list, and $L.nil.prev$ points to the *tail*.
- Similarly, both the $next$ attribute of the tail and the $prev$ attribute of the head point to $L.nil$.
- Since $L.nil.next$ points to the head, the attribute $L.head$ is eliminated altogether, with references to it replaced by references to $L.nil.next$.



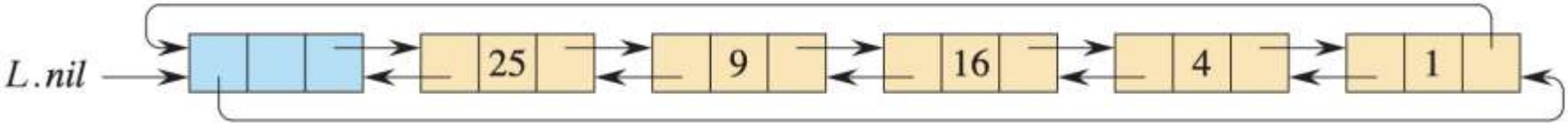
Lists with sentinels: Insert

```
LIST-INSERT′(x, y)
1  x.next = y.next
2  x.prev = y
3  y.next.prev = x
4  y.next = x
```

- *No separate procedure for prepending* is necessary: to insert at the head of the list, let y be $L.nil$.



Previous list

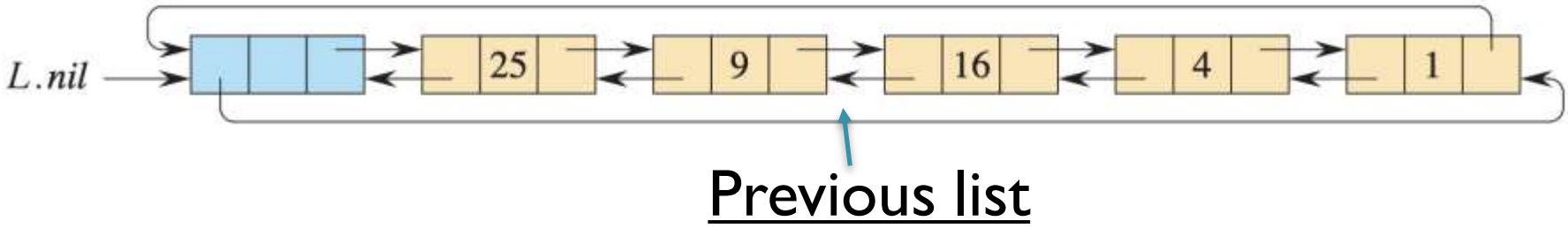


- After executing LIST-INSERT'$(x, L.nil)$ with $x.key = 25$.

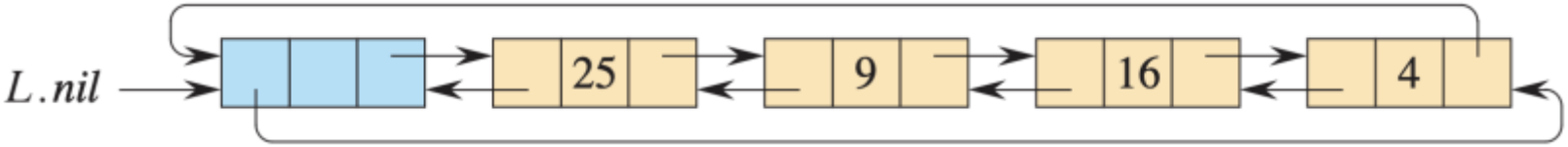
Lists with sentinels: Delete

```
LIST-DELETE′(x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

- Just a two-line procedure!



Previous list



- The list after deleting the object with key 1.

Lists with sentinels: Search

```
LIST-SEARCH′(L, k)
1  L.nil.key = k
2  x = L.nil.next
3  while x.key ≠ k
4        x = x.next
5  if x == L.nil
6        return NIL
7  else return x
```

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

(Old) search in the absence of sentinels

- Searching a list with a sentinel has the same asymptotic running time as without a sentinel, but it is possible to *decrease the constant factor*.

Lists with sentinels: Search

```
LIST-SEARCH'(L, k)
1  L.nil.key = k
2  x = L.nil.next
3  while x.key ≠ k
4        x = x.next
5  if x == L.nil
6        return NIL
7  else return x
```

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

(Old) search in the absence of sentinels

- The test in line 2 of LIST-SEARCH makes *two comparisons*, whereas the test in line 3 of LIST-SEARCH' makes *one comparison*.

Lists with sentinels

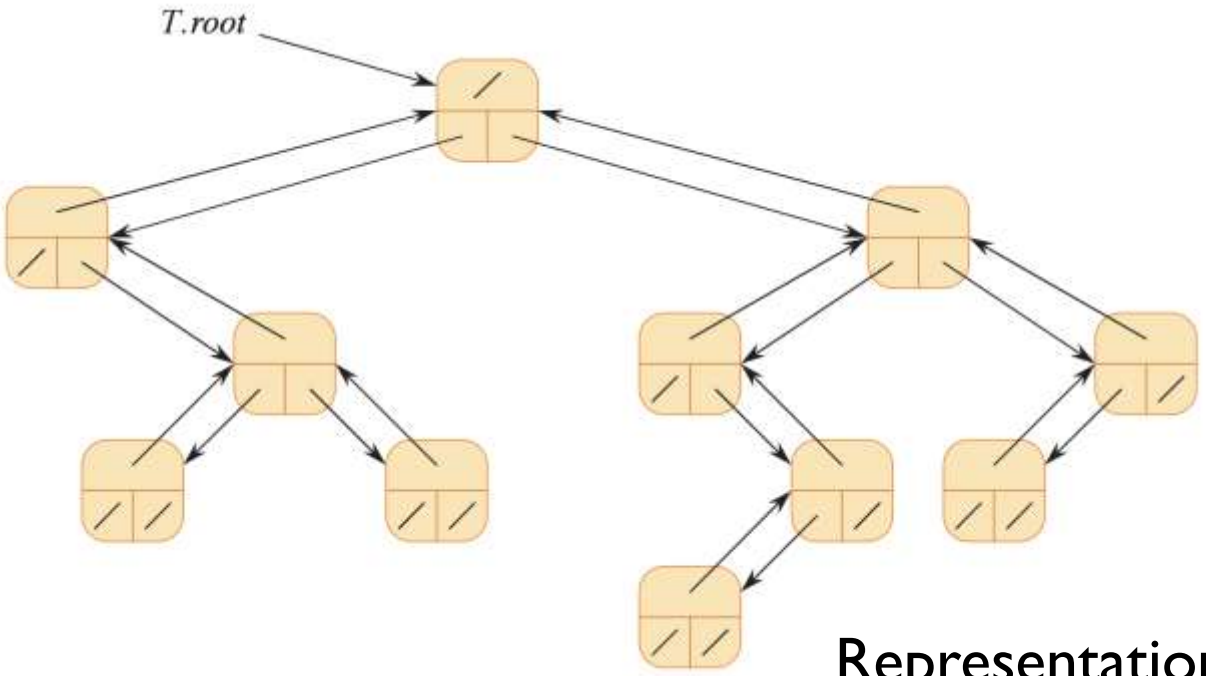
- Sentinels often *simplify code* and, as in searching a linked list, they might *speed up code by a small constant factor*.
- But they *don't* typically improve the asymptotic running time.
- *Use them judiciously*. When there are many small lists, the extra storage used by their sentinels can represent *significant wasted memory*.

Rooted trees

Representing rooted trees

- Linked lists work well for representing linear relationships, but not all relationships are linear. Consider for example rooted trees.
- We represent *each node of a tree by an object*.
- As with linked lists, we assume that each node contains a *key* attribute. The remaining attributes of interest are *pointers to other nodes*, and they vary according to the type of tree.
- A binary tree has two such pointers: left and right.

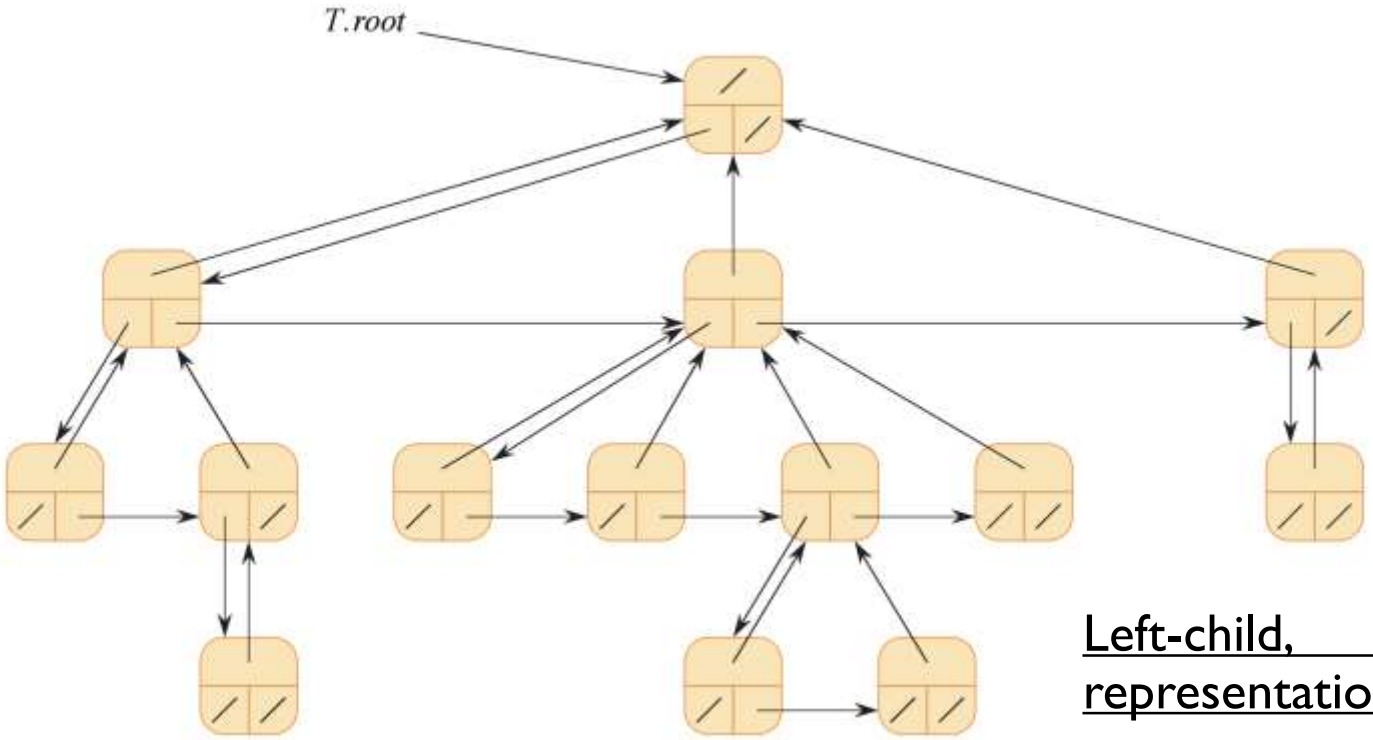
Binary trees



Representation of a binary tree T

- We use the attributes p, $left$, and $right$ to store pointers to the *parent*, *left child*, and *right child* of each node in a binary tree T. If $x.p = NIL$, then x is the *root*. The root of the entire tree T is pointed to by the attribute $T.root$.

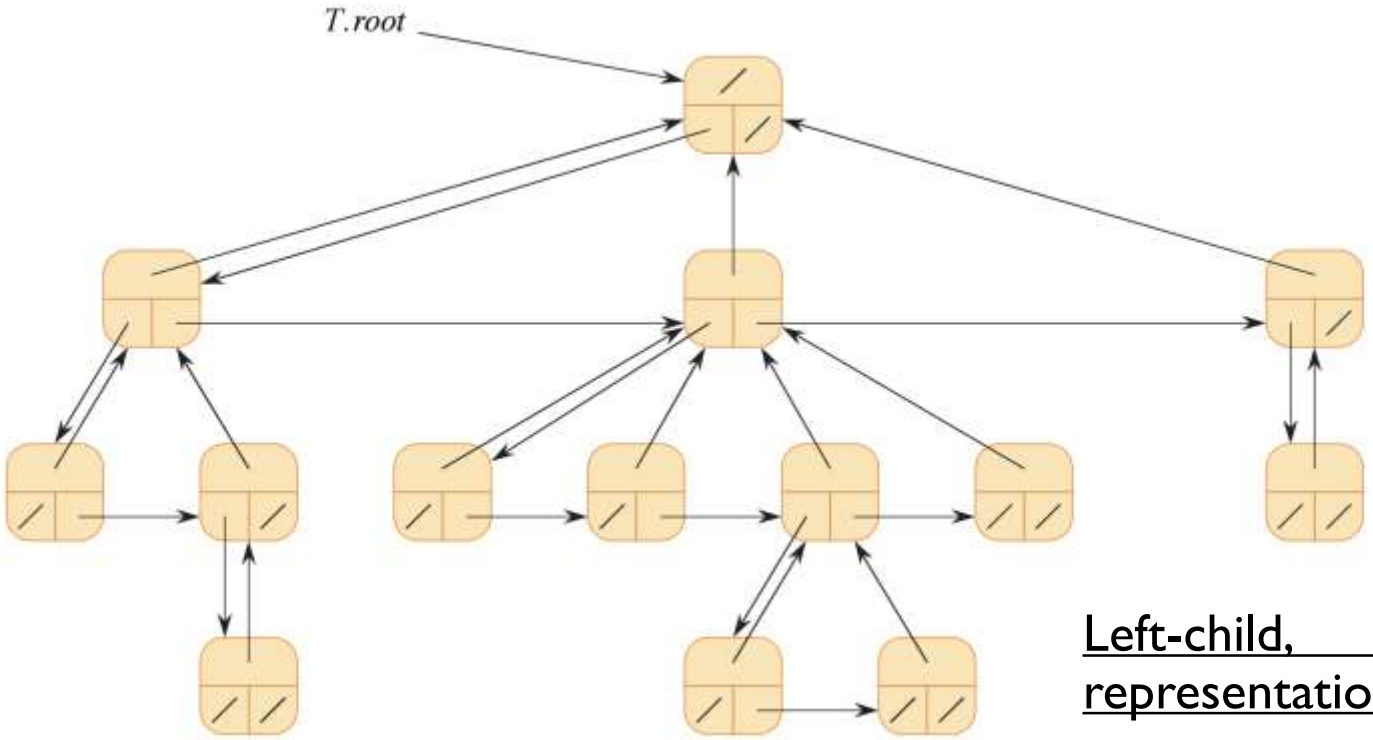
Trees with unbounded no. of children



Left-child, right-sibling representation of a tree T

- In the left-child, right-sibling representation, each node contains a parent pointer p, and two pointers:
  - $x.leftchild$ points to the *leftmost child* of node x, and
  - $x.rightsibling$ points to the sibling of x *immediately to its right*.

Trees with unbounded no. of children



Left-child, right-sibling representation of a tree T

- The representation has the *advantage of using only* $O(n)$ *space* for any n-node rooted tree.

Practice problems

Practice Problem 1

Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

Practice Problem 2

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue, pronounced like "deck") allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

Practice Problem 3

Explain why the dynamic-set operation INSERT on a singly linked list can be implemented in $O(1)$ time, but the worst-case time for DELETE is $\Theta(n)$.

Practice Problem 4

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

Practice Problem 5

Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

Practice Problem 6

Write an $O(n)$-time procedure that prints out all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

Practice Problem 7

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH				
INSERT				
DELETE				
SUCCESSOR				
PREDECESSOR				
MINIMUM				
MAXIMUM				