



Theory of algorithms (3rd lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- Medians and order statistics
 - Selection in expected linear time
- Dynamic sets
 - Operations on dynamic sets
 - Stacks and queues with arrays
 - Linked lists
 - Implementing pointers and objects with arrays
 - Representing rooted trees
- Exercises



Medians and order statistics

The i th **order statistic** of a set of n elements is the i th smallest element.

For example the **minimum** of a set of elements is the first order statistic ($i = 1$), and the **maximum** is the n th order statistic ($i = n$).

A **median**, informally, is the „halfway point” of the set. When n is odd, the median is unique, occurring at $i = (n+1)/2$. When n is even, there are two medians at $i = n/2$ (**lower median**) and $i = n/2+1$ (**upper median**) positions.

MINIMUM(A)

```
1  min =  $A[1]$ 
2  for  $i = 2$  to  $A.length$ 
3      if  $A[i] < min$ 
4           $min = A[i]$ 
5  return min
```

Remark: In the function we assume that the set resides in array A , where $A.length = n$.

Selection in expected linear time

■ The selection problem

Input: A set A of n (distinct) numbers and an integer i , with $1 \leq i \leq n$.

Output: The element $x \in A$ that is larger than exactly $i-1$ other elements of A .

RANDOMIZED-SELECT(A, p, r, i)

```
1  if  $p == r$ 
2      return  $A[p]$ 
3   $q = \text{RANDOMIZED-PARTITION}(A, p, r)$ 
4   $k = q - p + 1$ 
5  if  $i == k$ 
6      return  $A[q]$            // The pivot value is the answer
7  else if  $i < k$ 
8      return RANDOMIZED-SELECT( $A, p, q-1, i$ )
9  else
10     return RANDOMIZED-SELECT( $A, q+1, r, i-k$ )
```

Efficiency: Worst-case running time is $\Theta(n^2)$, but the **expected** running time is $\Theta(n)$.

Remark: There is an algorithm with $O(n)$ running time (it is also recursive and working with medians it does „good” partitioning.)



Exercises

- Give another version of RANDOMIZED-SELECT function with pseudocode that uses iteration instead of recursion?
- What values of i have to be generated in RANDOMIZED-PARTITION to get the worst-case running time to calculate the minimum value of array A ?
 - $A = \langle 2, 7, 5, 6, 1, 4, 3 \rangle$



Dynamic sets

- Dynamic sets
 - Finite sets in computer science that are manipulated by algorithms.
 - The **objects** of dynamic sets have (often unique) **key** and **satellite data**.
- Operations
 - $\text{SEARCH}(S, k)$ A query that, given a set S and a key value k , returns a pointer x to an element in S such that $x.\text{key}=k$, or NIL if no such element belongs to S .
 - $\text{INSERT}(S, x)$ A modifying operation that augments the set S with the element pointed to by x . We usually assume that any attributes in element x needed by the set implementation have already been initialized.
 - $\text{DELETE}(S, x)$ A modifying operation that, given a pointer x to an element in the set S , removes x from S .

Remark: We call a dynamic set that supports these operations a **dictionary**.



Dynamic sets

- Operations (continue)
 - $\text{MINIMUM}(S)$ A query that returns a pointer to the element of S with the smallest key.
 - $\text{MAXIMUM}(S)$ A query that returns a pointer to the element of S with the largest key.
 - $\text{SUCCESSOR}(S, x)$ A query that, given an element x whose key is in the set S , returns a pointer to the next larger element in S , or NIL if x is the maximum element.
 - $\text{PREDECESSOR}(S, x)$ A query that, given an element x whose key is in the set S , returns a pointer to the next smaller element in S , or NIL if x is the minimum element.

Remark: These operations assume that the keys are from a **totally ordered** set. In this kind of set the **trichotomy** is satisfied, that is, exactly one of the following three properties holds with elements a and b : $a < b$, $a = b$, $a > b$.

Stacks

STACK-EMPTY(S)

```
1  if  $S.top == 0$ 
2      return TRUE
3  else
4      return FALSE
```

PUSH(S, x)

```
1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

POP(S)

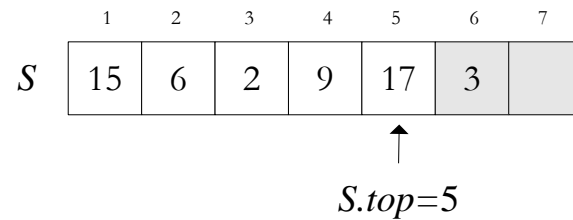
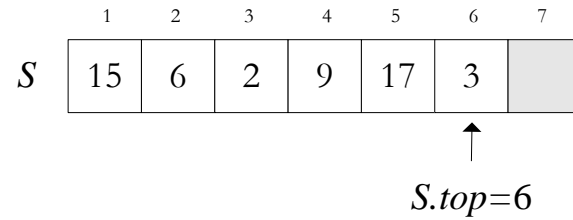
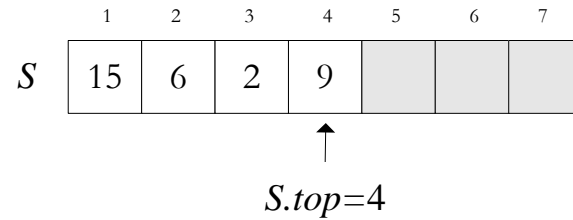
```
1  if STACK-EMPTY( $S$ )
2      error "underflow"
3  else
4       $S.top = S.top - 1$ 
5      return  $S[S.top + 1]$ 
```

Remark: The stack implements a **last-in, first-out**, or **LIFO**, policy.

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



Stacks



An array implementation of a stack

Exercises

- Illustrate the result of each operation in the sequence $\text{PUSH}(S, 4)$, $\text{PUSH}(S, 1)$, $\text{PUSH}(S, 3)$, $\text{POP}(S)$, $\text{PUSH}(S, 8)$, and $\text{POP}(S)$ on an initially empty stack S stored in array $S[1..6]$. What is the value of $S.\text{top}$?



Queues

ENQUEUE(Q, x)

```
1   $Q[Q.tail] = x$ 
2  if  $Q.tail == Q.length$ 
3       $Q.tail = 1$ 
4  else
5       $Q.tail = Q.tail + 1$ 
```

DEQUEUE(Q)

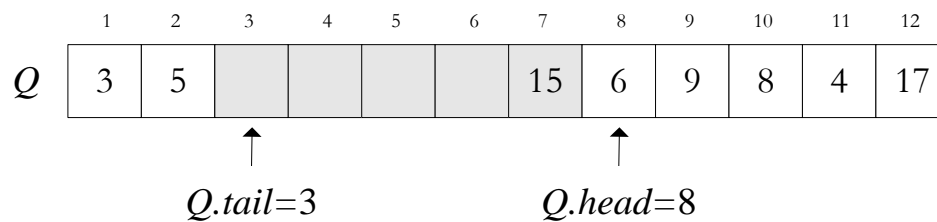
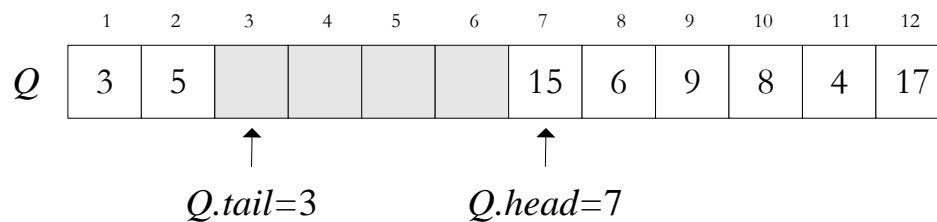
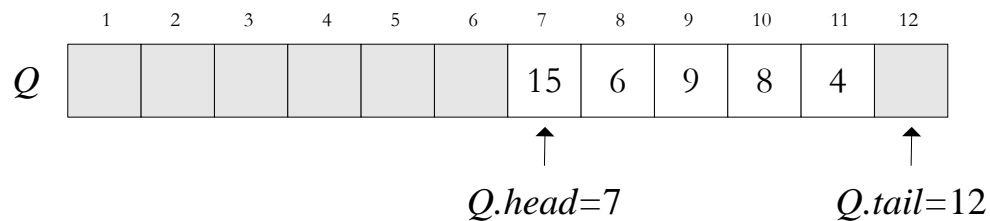
```
1   $x = Q[Q.head]$ 
2  if  $Q.head == Q.length$ 
3       $Q.head = 1$ 
4  else
5       $Q.head = Q.head + 1$ 
6  return  $x$ 
```

Remark: When $Q.head = Q.tail$, the queue is **empty**. Initially they are 1. If we attempt to dequeue an element from an empty queue, the queue **underflows**. When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue **overflows**. The error checking is omitted here.

Efficiency: Both queue operations take $O(1)$ time.



Queues



A queue implemented using an array $Q[1..12]$

Exercises

- Give the result of operations in the sequence $\text{ENQUEUE}(Q, 4)$, $\text{ENQUEUE}(Q, 1)$, $\text{ENQUEUE}(Q, 3)$, $\text{DEQUEUE}(Q)$, $\text{ENQUEUE}(Q, 8)$, and $\text{DEQUEUE}(Q)$ on an initially empty queue Q stored in array $Q[1..6]$. What is the value of $Q.\text{head}$ and $Q.\text{tail}$ if the initial values are $Q.\text{head}=Q.\text{tail}=5$?
- We implement a queue with $Q[1..n]$. Why does it contain at most $n-1$ element instead of n ?



Linked lists

LIST-SEARCH(L, k)

```
1   $x = L.head$   
2  while  $x \neq \text{NIL}$  and  $x.key \neq k$   
3       $x = x.next$   
4  return  $x$ 
```

LIST-INSERT(L, x)

```
1   $x.next = L.head$   
2  if  $L.head \neq \text{NIL}$   
3       $L.head.prev = x$   
4   $L.head = x$   
5   $x.prev = \text{NIL}$ 
```

Efficiency: To search a list of n objects, the LIST-SEARCH function takes $\Theta(n)$ time in the worst case, since it may have to search the entire list. The running time for LIST-INSERT is $O(1)$.

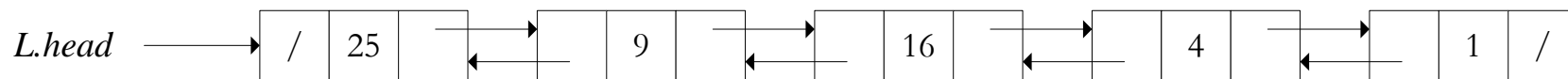
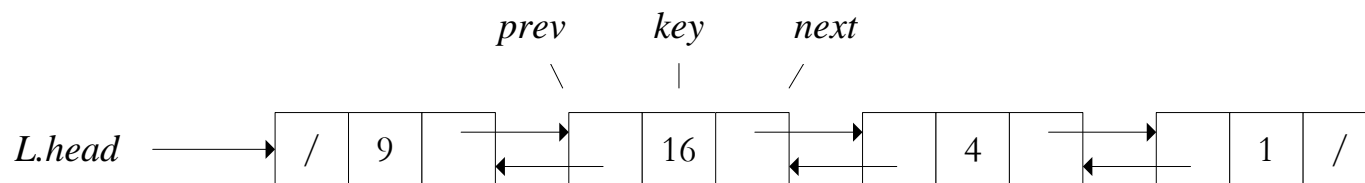
Linked lists

LIST-DELETE(L, x)

```
1  if  $x.prev \neq \text{NIL}$ 
2       $x.prev.next = x.next$ 
3  else
4       $L.head = x.next$ 
5  if  $x.next \neq \text{NIL}$ 
6       $x.next.prev = x.prev$ 
```

Efficiency: LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

Linked lists



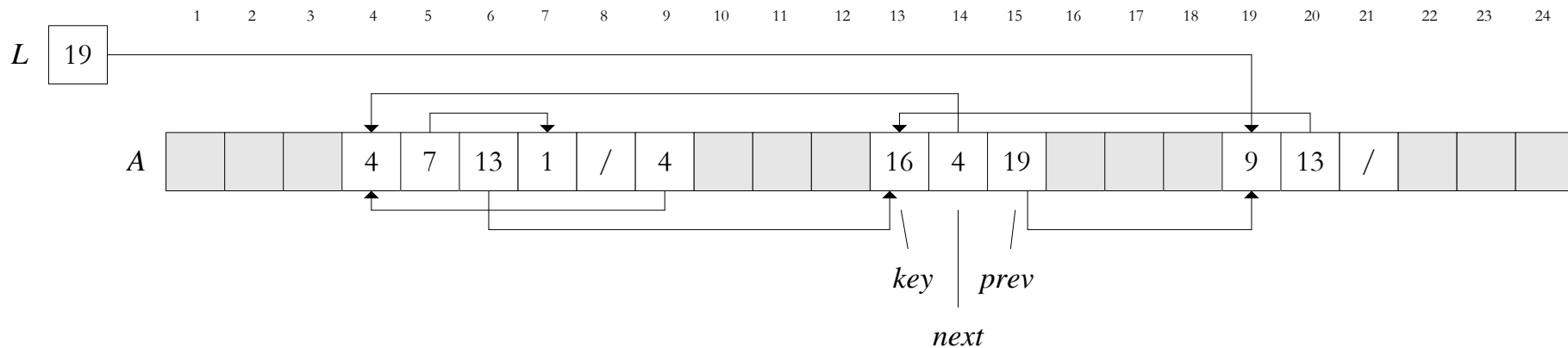
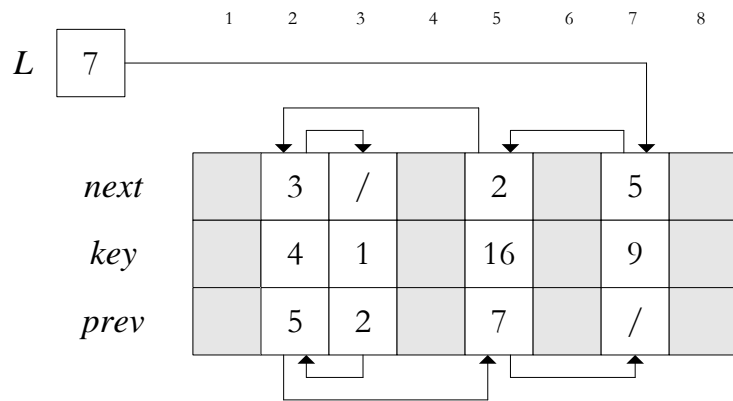
A doubly linked list L representing a dynamic set

Exercises

- 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(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

Implementing pointers and objects



A multiply-array and a single-array representation of a doubly linked list

Exercises

- Give the multiply-array representation of a doubly linked list that contains the keys given in arrays A . Let the size of the arrays be 10, and let the elements be in the odd positions of the arrays.
 - $A = \langle 5, 4, 8, 2, 1 \rangle$
- Give the single-array representation of a doubly linked list that contains the keys given in arrays A . Let the size of the array be 15, and let the elements be in the array continuously from the 1st position.
 - $A = \langle 6, 2, 5, 3 \rangle$



Implementing pointers and objects

ALLOCATE-OBJECT()

```
1  if free == NIL
2      error "out of space"
3  else
4      x = free
5      free = x.next
6      return x
```

FREE-OBJECT(*x*)

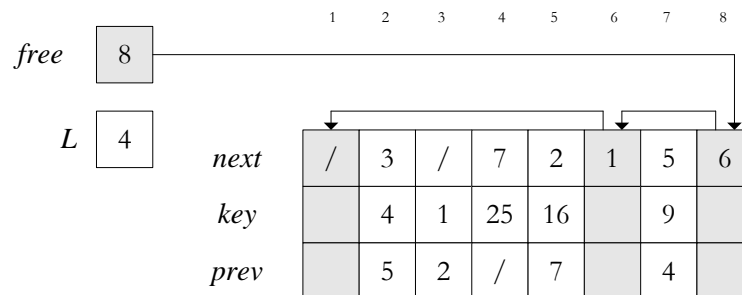
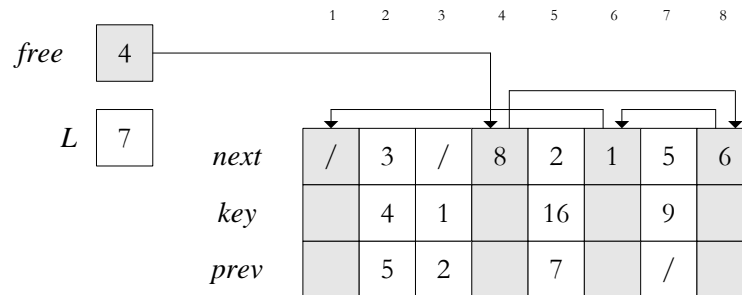
```
1  x.next = free
2  free = x
```

Remark: The free list initially contains all n unallocated objects. Once the free list has been exhausted, running the ALLOCATE-OBJECT signals an error.

Efficiency: Both subroutines run in $O(1)$ time.

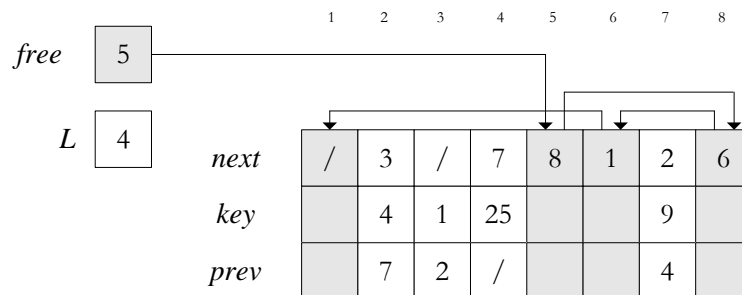


Implementing pointers and objects



The result of calling ALLOCATE-OBJECT() (that returns index 4), setting *key*[4] to 25, and calling LIST-INSERT(*L*, 4).

The new free-list head is object 8, which had been *next*[4] on the free list.



After executing LIST-DELETE(*L*, 5), we call FREE-OBJECT(5).

Object 5 becomes the new free-list head, with object 8 following it on the free list.

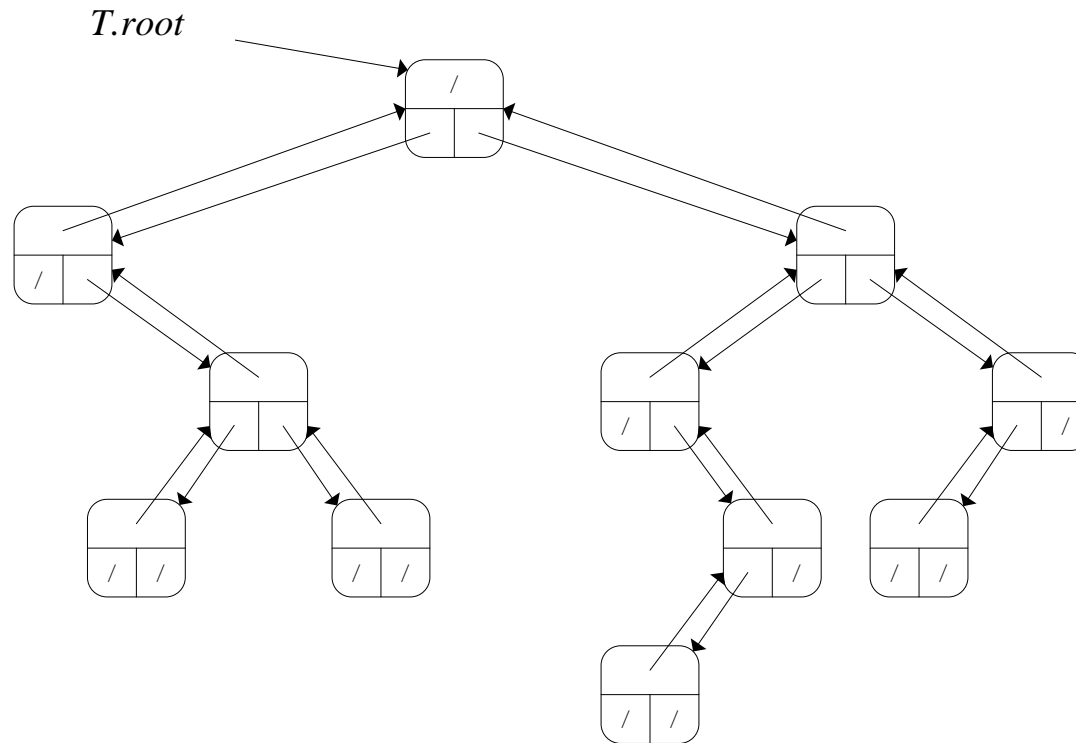
Allocating and freeing objects



Exercises

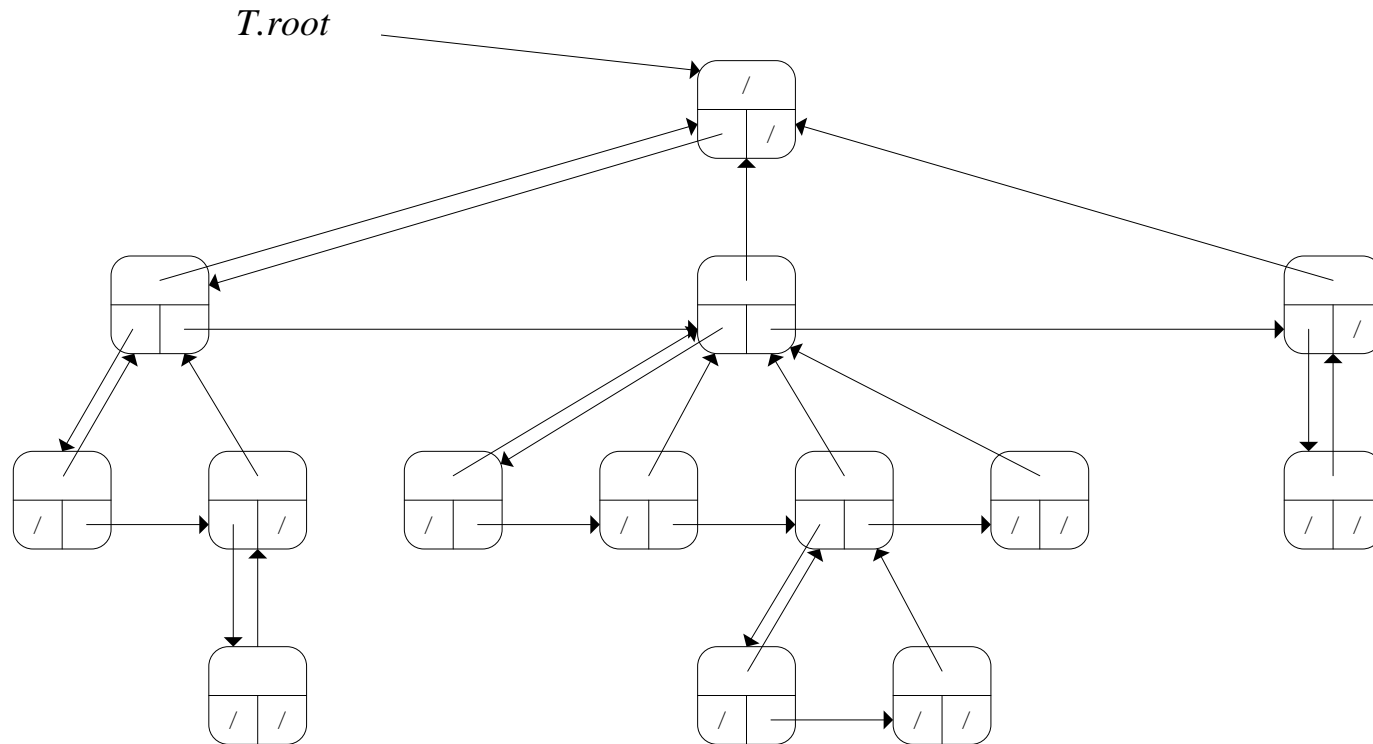
- Insert elements with key 12 and 8 into the previous list and delete the element index 3. Give the result after the operations.

Representing rooted trees



The representation of a binary tree T

Representing rooted trees



The left-child, right-sibling representation of a tree T

Exercises

- Draw the binary tree rooted at index 6 that is represented by the following attributes.

index	key	left	right
1	12	7	3
2	15	NIL	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL