Data Structures and Algorithms: Lecture 10

Barbara Morawska

October 4, 2018

Dynamic sets

dynamic set: can grow, shrink, change over time

Problem

- How to represent a finite dynamic set?
- How to manipulate such a set?

Example: Dictionary

An implementation of a set that supports:

- inserting element
- deleting an element
- test for membership of an element in the set

Operations on dynamic sets

Element has key (k) (value) and pointer (x) (address)

Queries

- $\textsc{Search}(S, k)$
- $\textsc{Minimum}(S)$
- $\textsc{Maximum}(S)$
- $\textsc{Successor}(S, x)$: returns a pointer to the next larger element in S or NIL if x is maximal.
- $\textsc{Predecessor}(S, x)$: returns a pointer to the next smaller element in S or NIL if x is minimal.

Problem: if you have these operations implemented, how to enumerate all elements of S in the increasing order

Modifying operations

- $\textsc{Insert}(S, x)$
- $\textsc{Delete}(S, x)$

Stacks and queues

We use pointers. Later we will see how to implement pointers.

Which element can be deleted?

Stack

- we can delete only the element which was most recently inserted:
- Last-in, first-out
- LIFO policy

Queue

- we can delete only the element which is in the set for longest time
- First-in, first-out
- FIFO policy

Stacks – LIFO policy

- INSERT = PUSH
- DELETE = POP

Implementation with an array

- $S[1..n]$ is an array
- Define an attribute of S, $S.top$, the index for the most recently inserted element
- Stack: $S[1..S.top]$
- $S[1]$ is the bottom of the stack
- If $S.top = 0$, then the stack is empty
- If the empty stack is popped, error: Stack underflows
- If $S.top$ exceeds the length: Stack overflows

Stack operations

Query:

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

Insert:

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

Stack operatins cnt.

Delete:

Procedure POP(S)

```
1 if STACK-EMPTY(S) then
2     return ERROR "STACK UNDERFLOW"
3 else
4     S.top = S.top − 1
5     return S[S.top + 1]
```

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

Queues – FIFO strategy

- INSERT = ENQUEUE
- DELETE = DEQUEUE

Implementation with an array

- Let $Q[1..n]$ be an array
- We define the attributes of Q denoting indexes in Q:
  - $Q.head = Q[1]$
  - $Q.tail = Q[n]$
- If $Q.head = Q.tail$ then Q is empty.
- Initially $Q.head = Q.tail = 1$
- If queue is empty, DEQUEUE causes error: Queue underflow
- If $Q.head = Q.tail + 1$ the queue is full.
- If queue is full, ENGUEUE element causes error: Queue overflow

Queue operations – in $O(1)$ time

Notice: we will not check for underflow or overflow here.

Insert:

Procedure ENQUEUE(Q, x)

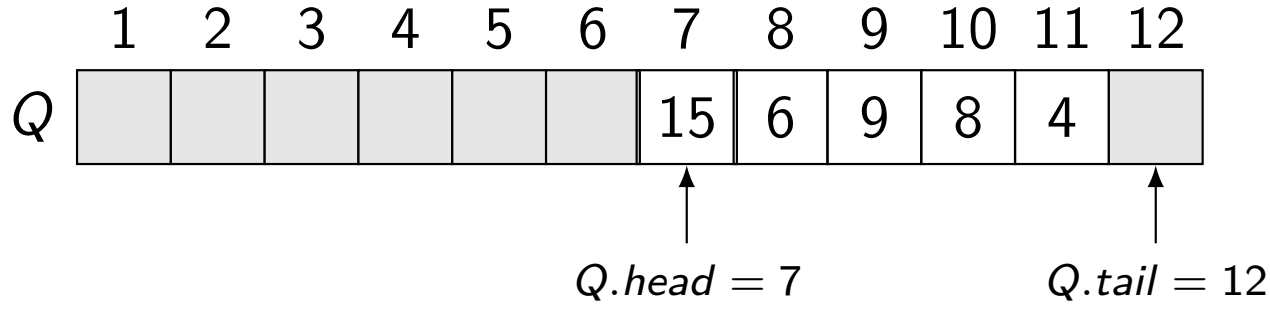
```
1 Q[Q.tail] = x
2 if Q.tail == Q.length then
3       Q.tail = 1
4 else
5       Q.tail = Q.tail + 1
```

Delete:

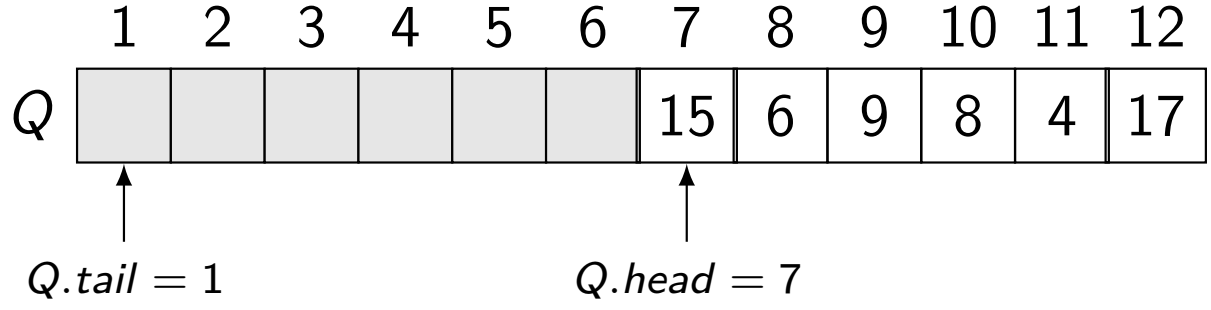
Procedure DEQUEUE(Q)

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

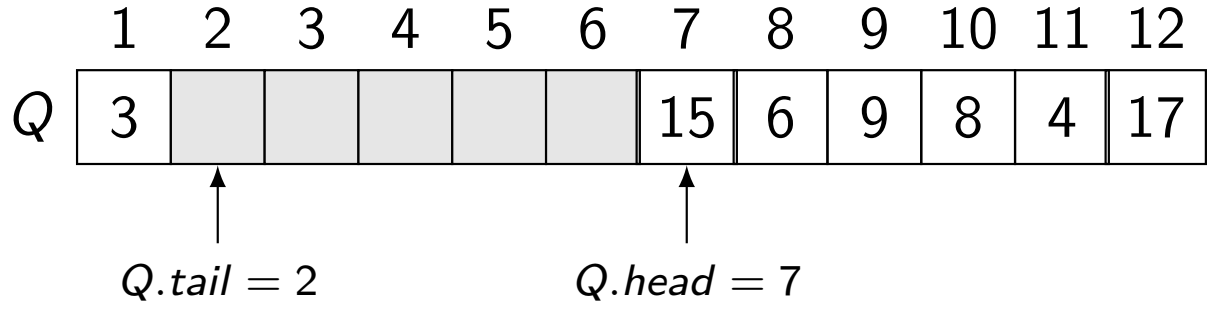
Example



ENQUEUE$(Q, 17)$:

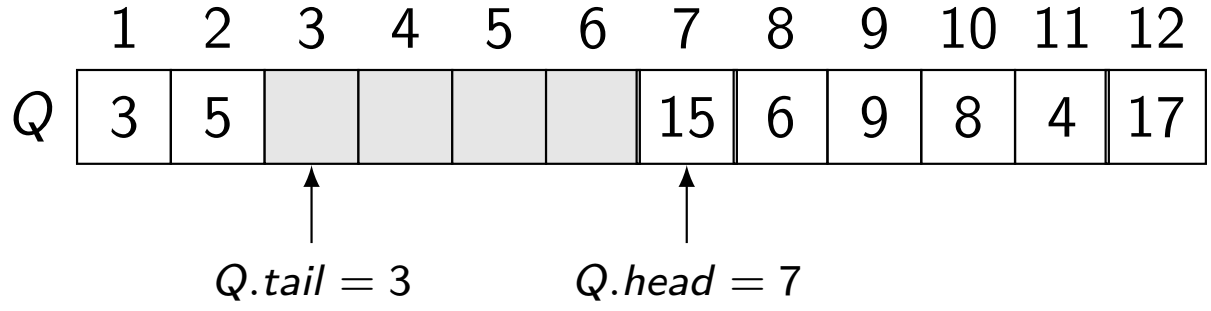


ENQUEUE$(Q, 3)$:

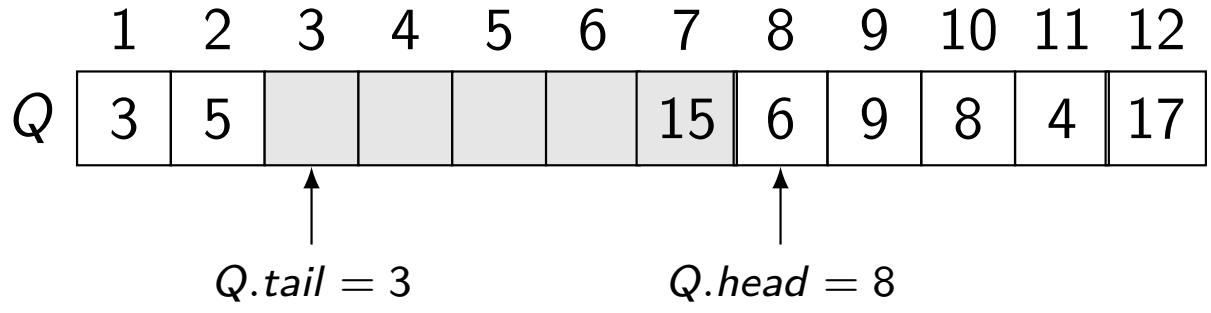


Example cont.

ENQUEUE$(Q, 5)$:



DEQUEUE(Q):



Linked lists

- Array: order of the elements is determined by indexes
- Linked list: order of the elements is determined by pointers

Doubly-linked list

- The list L has one attribute
  $L.head$ ($L.tail$ is just the last element).
- Each object besides *key*, has two attributes (pointers):
  - *next*
  - *prev*
- If $x.next = NIL$, x is the last element.
- If $x.prev = NIL$, x is the first element of the list, *head*.
- If $L.head = NIL$ the list is empty.

Linked lists

Singly-linked list

In the definition of doubly-linked list drop *prev* attribute in each element.

Sorted-linked list

next of an element points to a larger element.
The minimum element is the head of such list and the maximum is the tail.

Unsorted-linked list

The elements may appear in any order.

Circular-linked list (ring of elements)

next of the tail points to the *head* of the list. *prev* of the head points to the *tail* of the list.

Searching a linked list

Assume an unsorted and doubly-linked list L

Procedure LIST-SEARCH(L, k)

```
1 x = L.head
2 while x ≠ NIL and x.key ≠ k do
3     x = x.next
4 return x
```

Running time: $\Theta(n)$ in the worst case.

Inserting into a linked list

We insert new element at the head of the list.

Procedure LIST-INSERT(L, x)

```
1 x.next = L.head
2 if L.head ≠ NIL then
3       L.head.prev = x
4 L.head = x
5 x.prev = NIL
```

Running time: $O(1)$.

Deleting from a linked list

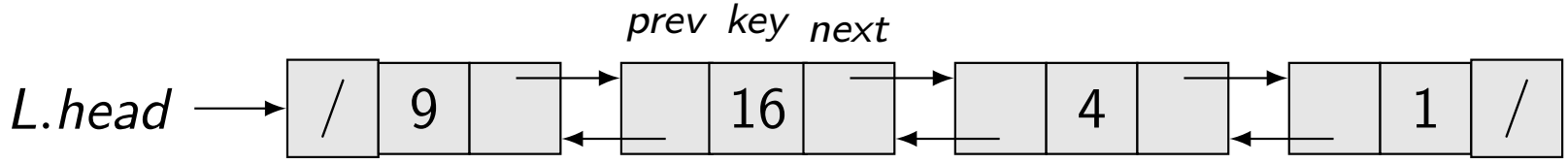
We pass pointer to the element to be deleted. If we pass *key*, we would have to use LIST-SEARCH.

Procedure LIST-DELETE(L, x)

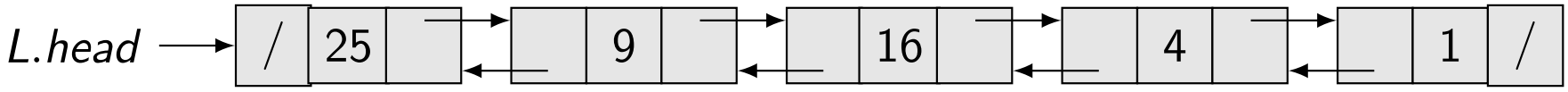
```
1 if x.prev ≠ NIL then
2     x.prev.next = x.next
3 else
4     L.head = x.next
5 if x.next ≠ NIL then
6     x.next.prev = x.prev
```

Running time: $O(1)$.

Example



List-Insert(L, x) where $x.key = 25$:



List-Delete(L, x), where x points to the fourth element with key 4.

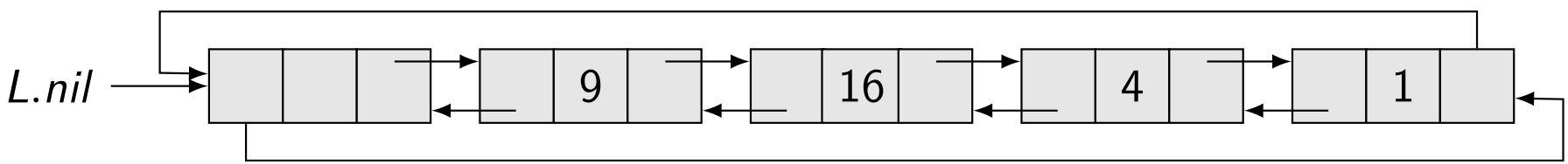
Double-linked list with a sentinel

We can use a sentinel to simplify the code.

Procedure LIST-DELETE'(L, x)

1 $x.prev.next = x.next$
2 $x.next.prev = x.prev$

- A sentinel is a dummy object to simplify boundary conditions of a procedure.
- $L.nil$ – it replaces $L.head$



Circular double-linked list with sentinel

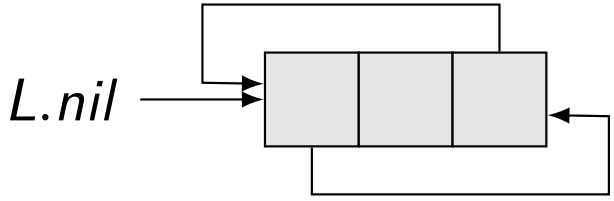
Procedure LIST-SEARCH'(L, k)

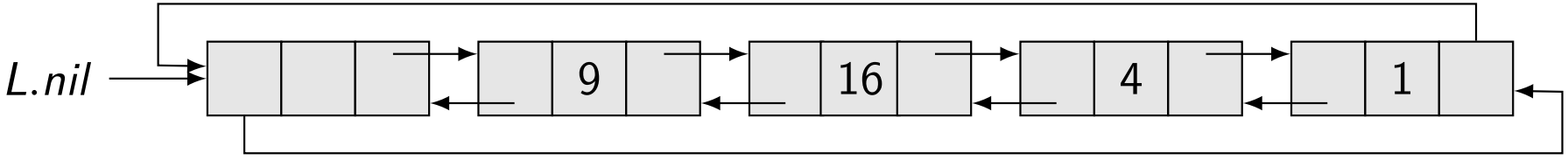
```
1 x = L.nil.next
2 while x ≠ L.nil and x.key ≠ k do
3     x = x.next
4 return x
```

Procedure LIST-INSERT'(L, x)

```
1 x.next = L.nil.next
2 L.next.prev = x
3 L.nil.next = x
4 x.prev = L.nil
```

Example





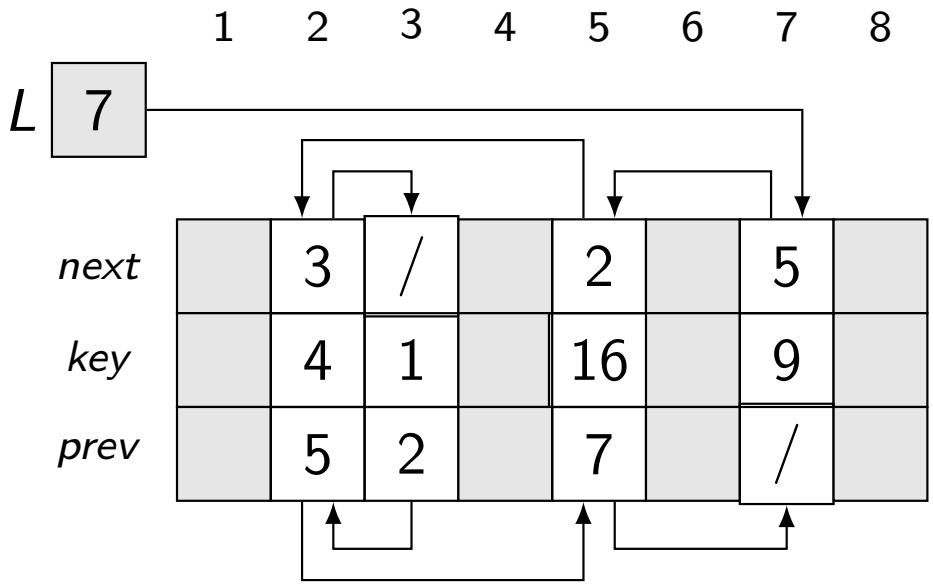
- Run LIST-INSERT'(L, x), where $x.key = 25$.
- Run LIST-DELETE'(L, x), where x is the 4'th element of the list with $x.key = 1$

Implementing pointers and objects

Suppose we cannot use pointers provided by a programming language, we want to implement linked list anyway.

A multiple-array representation of objects

In this approach we have a separate array for each attribute of an object: *prev*, *key*, *next*:



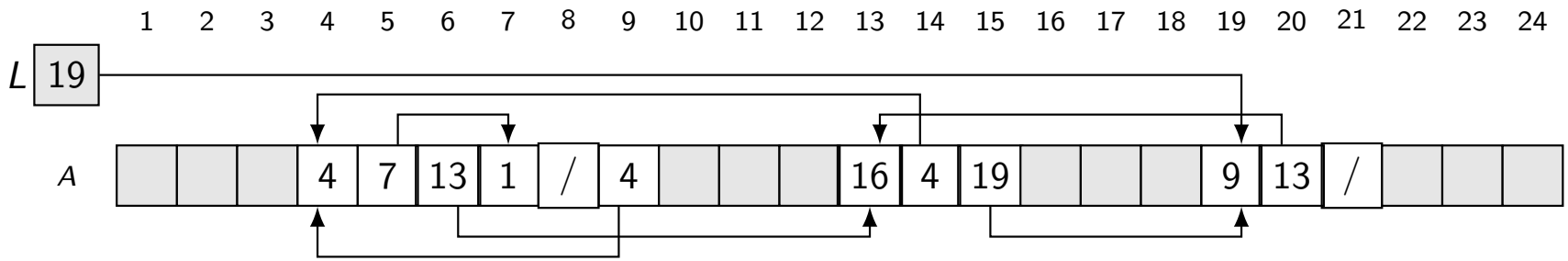
As NIL you could use integer -1 or 0.

Draw this linked list as implemented with pointers.

A single array representation of linked list

Often in object implementation a pointer is an address to the first memory location of the object. The attributes of the object are addressed by an offset to this pointer.

Attributes of an element are addressed by offset to the main location



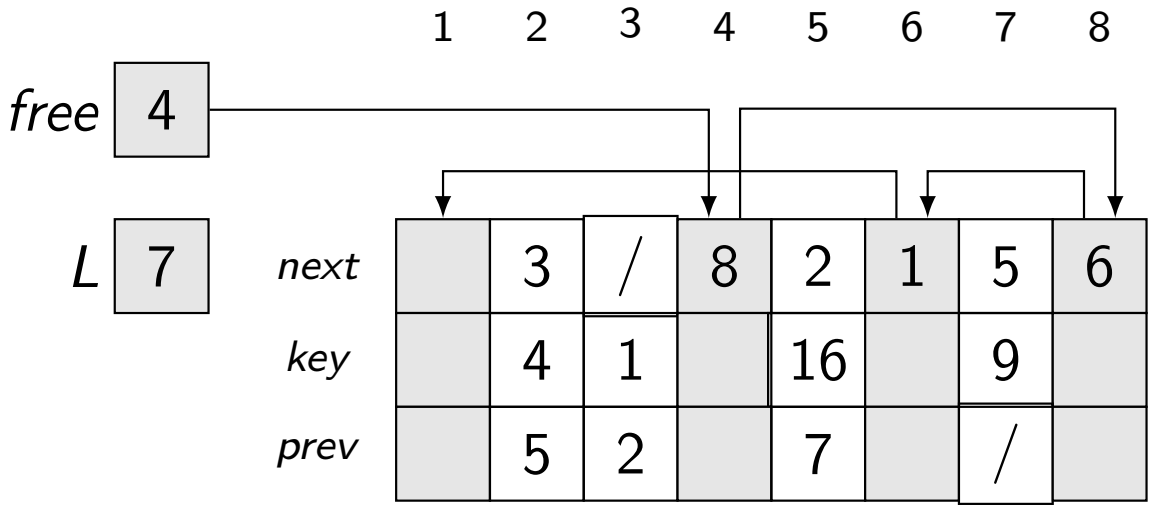
Allocating and freeing objects

- Assume we have a doubly-linked list represented by multiple arrays
- All of them have length m
- The dynamic set contains $n \leq m$ elements.
- $m - n$ objects are free.

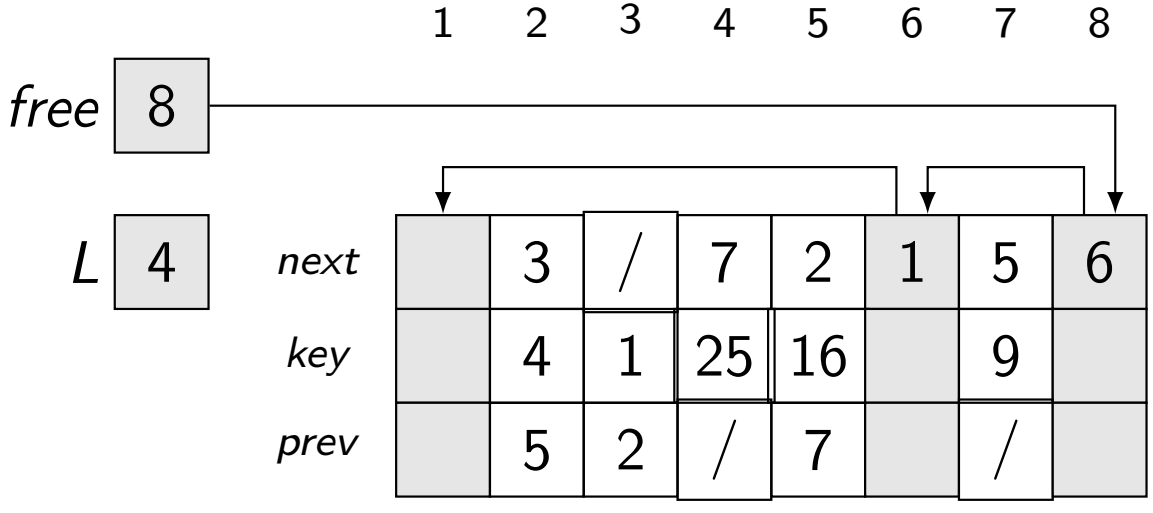
Idea:

- Keep the free objects in a singly-linked list: free list.
- Free list should behave like a stack.
- Two lists can intertwine.

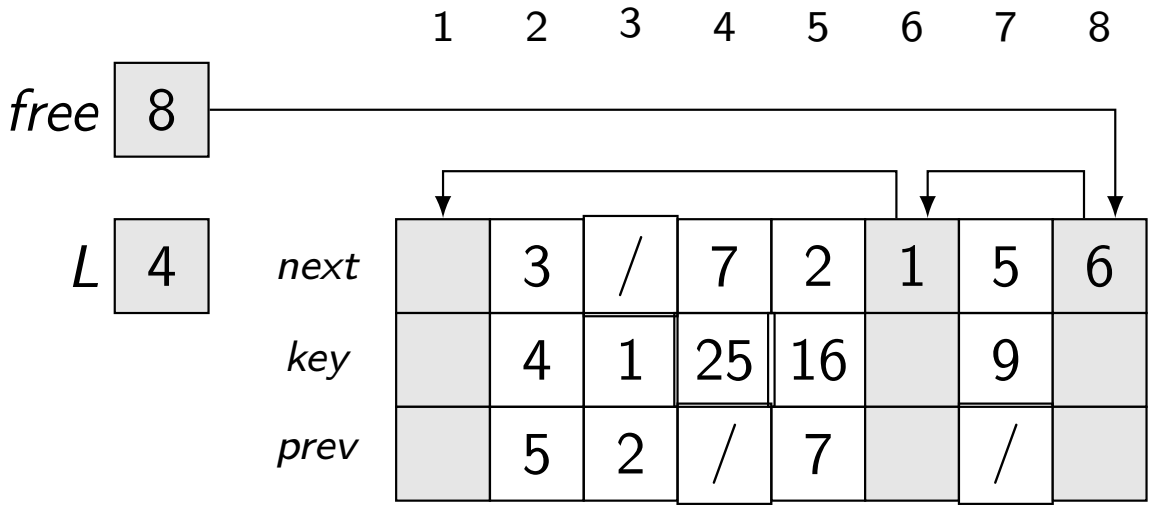
Free list intertwined with a double-linked list



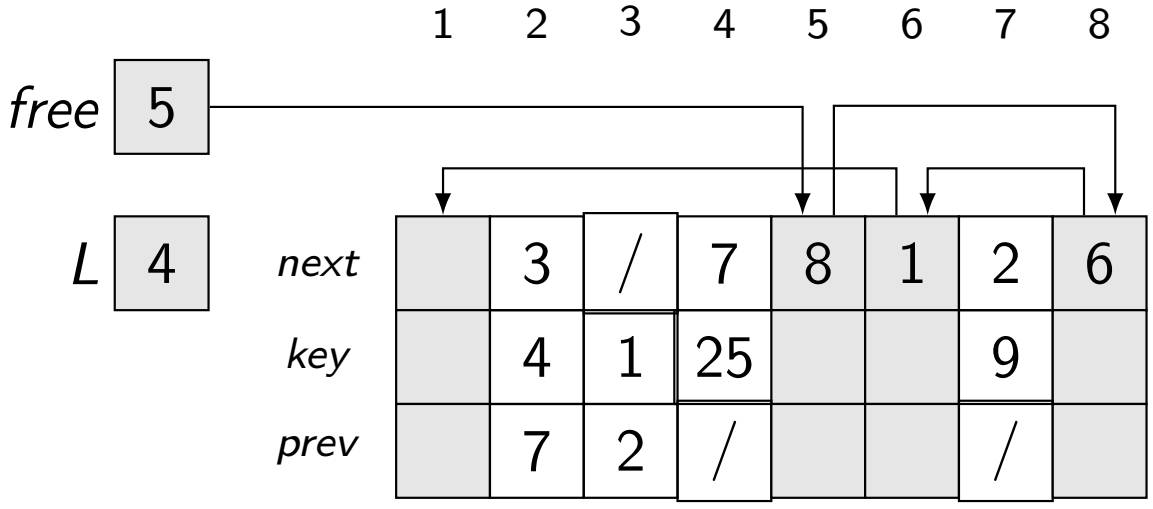
INSERT 25:



Free list intertwined with a double-linked list



DELETE$(L, 5)$



Procedures

Procedure ALLOCATE-OBJECT()

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

Procedure FREE-OBJECT(x)

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

Running time: $O(1)$

Representing rooted trees

- Each node of a tree is an object
- It contains *key* and other attributes

Binary trees

Attributes: $x.p$ (parent), $x.left$ (left child), $x.right$ (right child).

- If $x.p = NIL$, x is the root.
- If x has no left child, $x.left = NIL$
- If tree is T, $T.root$ is a pointer to the root.
- If $T.root = NIL$, the tree T is empty.

Rooted tree with unbounded branching

If we know the maximal number of children for each node, we can define attributes: $x.child\_1, x.child\_2, \ldots, x.child\_k$.
Problem: what if the number of children is unbounded or unknown? What if we need to save space?

Left-child, right-sibling representation

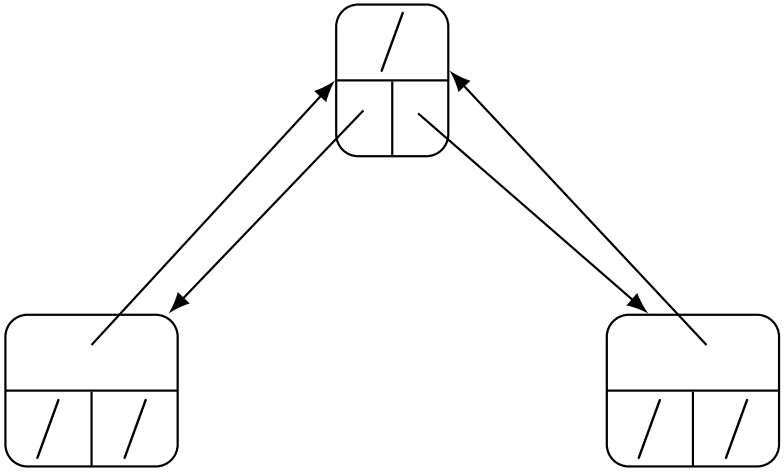
Attributes for each node:

- $x.p$ pointing to the parent
- $x.left\_child$ pointing to the leftmost child
- $x.right\_sibling$ pointing to the right sibling

For the tree T, $T.root$ points to the root.

- If a node x has not children, $x.left\_child = NIL$
- If x is the rightmost child of its parent, $x.right\_sibling = NIL$.

Example: tree representations



Left-child, right sibling representation