Elementary Data Structures

Introduction

- Part III: data structures for dynamic sets
- Set in mathematics
  - {1, 2, 5, 4, 3, 6}
- Set in algorithms
  - Allow repetition in set elements: {1, 2, 5, 4, 3, 6, 4}
  - Dynamic: can grow, shrink, or change over time
  - Set operations: insert, delete, test membership

Elements of A Dynamic Set

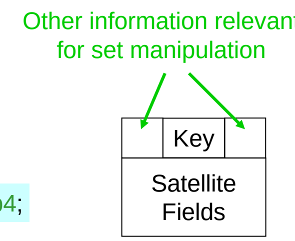
- Each element is represented by an object (or record)
  - An object may consists of many fields
  - Need a pointer to an object to examine and manipulate its fields
  - Key field for identifying objects and for the set manipulation
    - Keys are usually drawn from a totally ordered set
  - Satellite fields: all the fields irrelevant for the set manipulation

```
Type Record patron {
  integer    patron_ID;
  char[20]   name;
  integer    age;
  char[10]   department;
  …
}
```

Key field

Satellite fields

```
Type patron p1, p2, p3, p4;
```

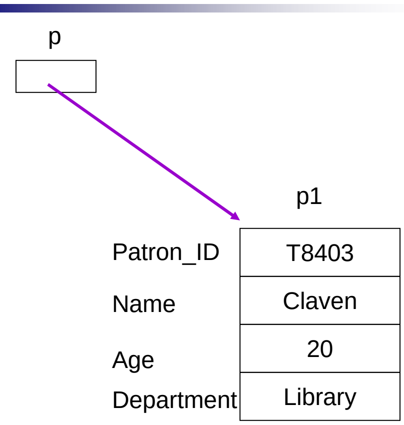


Record(Object) and Pointer

```
Type Record patron {
  integer    patron_ID;
  char[20]   name;
  integer    age;
  char[10]   department;
  …
}
```

```
Type patron p1, p2, p3, p4;
```

```
Type patron  *p;
```



Operations on Dynamic Sets

- Query operations: return information about a set
  - SEARCH(S, k): given a set S and key value k, returns a pointer x to an element in S such that key[x] = k, or NIL if no such element belongs to S
  - MINIMUM(S): returns a pointer to the element of S with the smallest key
  - MAXIMUM(S): returns a pointer to the element of S with the largest key
  - SUCCESSOR(S, x): returns a pointer to the next larger element in S, or NIL if x is the maximum element
  - PREDECESSOR(S, x): returns a pointer to the next smaller element in S, or NIL if x is the minimum element

Operations on Dynamic Sets (Cont.)

- Modifying operations: change a set
  - INSERT(S, x): augments the set S with the element pointed to by x. We usually assume that any fields in element x needed by the set implementation have already initialized.
  - DELETE(S, x): given a pointer x to an element in the set S, removes x from S.

Overview of Part III

- Heap – Chapter 6
- Elementary data structures – Chapter 10
  - Stacks, queues, linked lists, root trees
- Hash tables – Chapter 11
- Binary search trees – Chapter 12
- Red-Black trees – Chapter 13
- Augmenting Data Structures – Chapter 14

Stacks

Introduction

- Stack
  - The element deleted from the set is the one most recently inserted
  - Last-in, First-out (LIFO)
- Stack operations
  - PUSH: Insert
  - DELETE: Delete
  - TOP: return the key value of the most recently inserted element
  - STACK-EMPTY: check if the stack is empty
  - STACK-FULL: check if the stack is full

Represent Stack by Array

- A stack of at most n elements can be implemented by an array S[1..n]
  - top[S]: a pointer to the most recently inserted element
  - A stack consists of elements S[1..top[S]]

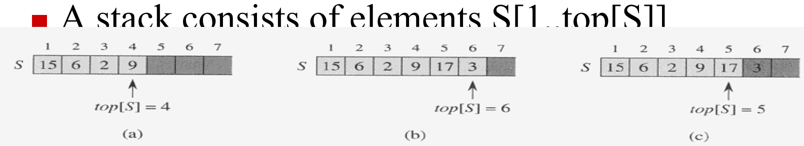


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Stack Operations

STACK-EMPTY(S)
if $top[S] = 0$
  then return TRUE
  else return FALSE

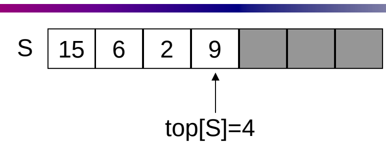
PUSH(S, x)
$top[S] \leftarrow top[S] + 1$
$S[top[S]] \leftarrow x$

POP(S)
if STACK-EMPTY(S)
  then error "underflow"
  else $top[S] \leftarrow top[S] - 1$
    return $S[top[S] + 1]$

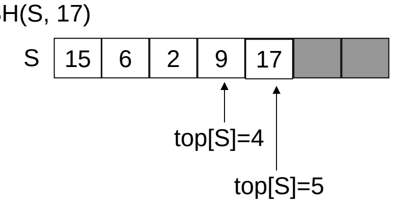
How to implement
TOP(S), STACK-FULL(S) ?

O(1)

Illustration of PUSH



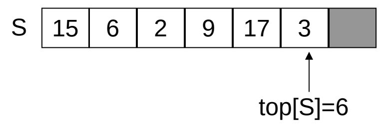
PUSH(S, 17)



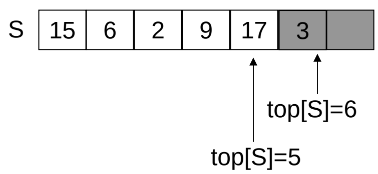
top[S] ← top[S] + 1 (top[S] = 5)

S[top[S]] ← x (S[5] = 17)

Illustration of POP



POP(S)



top[S] ← top[S] - 1 (top[S] = 5)

return S[top[S]+1] (return S[6])

Queues

Introduction

- Queue
  - The element deleted is always the one that has been in the set for the longest time
  - First-in, First-out (FIFO)
- Queue operations
  - ENQUEUE: Insert
  - DEQUEUE: Delete
  - HEAD: return the key value of the element that has been in the set for the longest time
  - TAIL: return the key value of the element that has been in the set for the shortest time
  - QUEUE-EMPTY: check if the queue is empty
  - QUEUE-FULL: check if the queue is full

Represent Queue by Array

- A queue of at most n-1 elements can be implemented by an array Q[1..n]
  - Q.head a pointer to the element that has been in the set for the longest time
  - Q.tail: a pointer to the next location at which a newly arriving element will be inserted into the queue
  - The elements in the queue are in locations Q.head,
  - Q.head + 1, …, Q.tail -1
    - The array is circular
  - Empty queue: Q.head = Q.tail
    - Initially we have Q.head = Q.tail = 1
  - Full queue: Q.head = Q.tail + 1 (in circular sense)

Illustration of A Queue

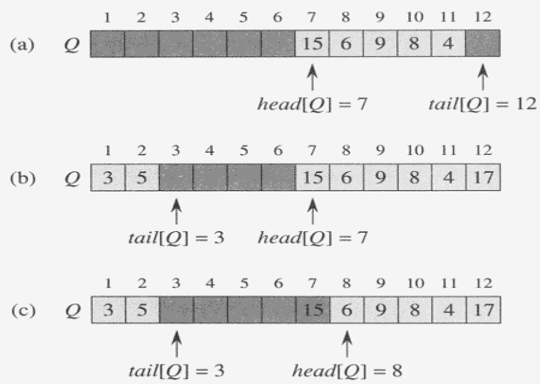


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Queue Operations

```
ENQUEUE(Q, x)
Q[tail[Q]] ← x
if tail[Q] = length[Q]
    then tail[Q] ← 1
    else tail[Q] ← tail[Q] + 1
```

```
DEQUEUE(Q)
x ← Q[head[Q]]
if head[Q] = length[Q]
    then head[Q] ← 1
    else head[Q] ← head[Q] + 1
return x
```

How to implement other queue operations ?

O(1)

Linked Lists

Introduction

- A linked list is a data structure in which the objects are arranged in linear order
  - The order in a linked list is determined by pointers in each object
- Doubly linked list
  - Each element is an object with a *key* field and two other pointer fields: *next* and *prev*, among other satellite fields. Given an element x
    - next[x] points to its successor
      - if x is the last element (called tail), next[x] = NIL
    - prev[x] points to its predecessor
      - if x is the first element (called head), prev[x] = NIL
  - An attribute head[L] points to the first element of the list
    - if head[L] = NIL, the list is empty

Introduction (Cont.)

- Singly linked list: omit the *prev* pointer in each element
- Sorted linked list: the linear order of the list corresponds to the linear order of keys stored in elements of the list
  - The minimum element is the head
  - The maximum element is the tail
- Circular linked list: the *prev* pointer of the head points to the tail, and the *next* pointer of the tail points to the head

Illustration of A Doubly Linked List

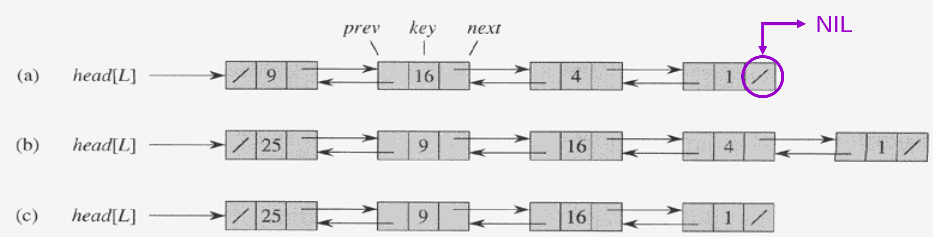


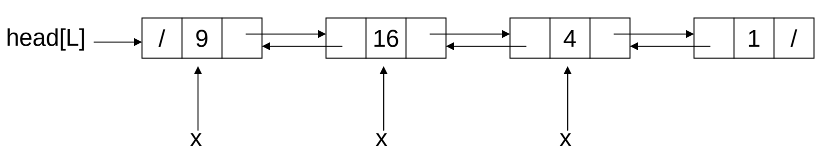
Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The *next* field of the tail and the *prev* field of the head are NIL, indicated by a diagonal slash. The attribute *head*$[L]$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where *key*$[x] = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Searching A Linked List

- LIST-SEARCH(L, k): finds the first element with key k in list L by a simple linear search, returning a pointer to this element

```
LIST-SEARCH(L, k)
  x ← head[L]
  while x ≠ NIL and key[x] ≠ k
        do x ← next[x]
  return x
```

Illustration of LIST-SEARCH



LIST-SEARCH(L, 4)

return x

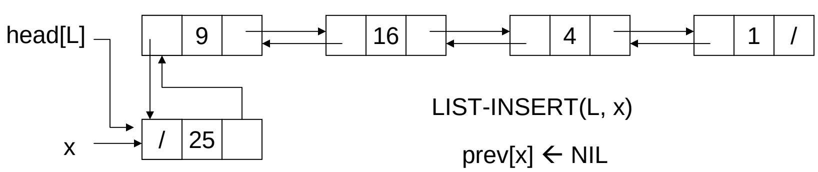
How about LIST-SEARCH(L, 7)?

Inserting Into A Linked List

- LIST-INSERT(L, x): given an element pointed by x, splice x onto the front of the linked list

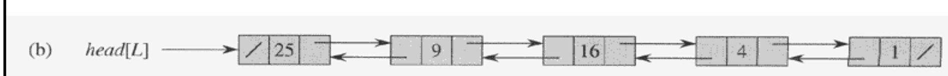
```
LIST-INSERT(L, x)
    next[x] ← head[L]
    if head[L] ≠ NIL
        then prev[head[L]] ← x
    head[L] ← x
    prev[x] ← NIL
```

Illustration of LIST-INSERT



LIST-INSERT(L, x)

prev[x] ← NIL

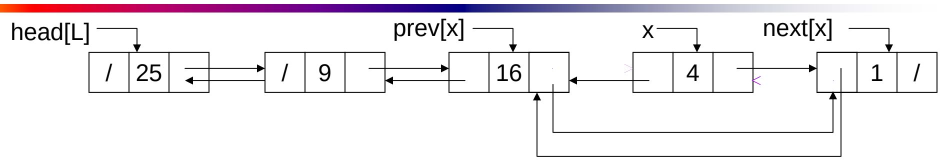


Deleting From A Linked List

- LIST-DELETE(L, x): given an element pointed by x, remove x from the linked list

```
LIST-DELETE(L, x)
if prev[x] ≠ NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x] ≠ NIL
    then prev[next[x]] ← prev[x]
```

Illustration of LIST-DELETE



LIST-DELETE(L, x)

prev[next[x]] ← prev[x]

Need garbage collection for x

Implementing Pointers and Objects

Pointers in Pseudo Language

```
Type Record patron {
  integer     patron_ID;
  char[20]    name;
  integer     age;
  char[10]    department;
  …
}
```

```
Type patron p1, p2, p3, p4;
```

```
Type patron *pointer_to_p1
```

```
Type Record patron_list {
  integer     patron_ID;
  char[20]    name;
  integer     age;
  char[10]    department;
  …
  Type patron_list *prev;
  Type patron_list *next;
}
Type patron_list *head;
```

Some languages, like C and C++, support pointers and objects; but some others not

A Multiple-Array Representation of Objects

- We can represent a collection of objects that have the same fields by using an array for each field.
  - Figure 10.3 (a) and Figure 10.5
    - For a given array index x, key[x], next[x], and prev[x] represent an object in the linked list
    - A pointer x is simply a common index on the the key, next, and prev arrays
    - NIL can be represented by an integer that cannot possibly represent an actual index into the array

A Multiple-Array Representation of Objects Example

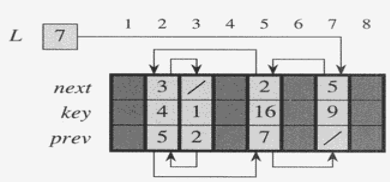
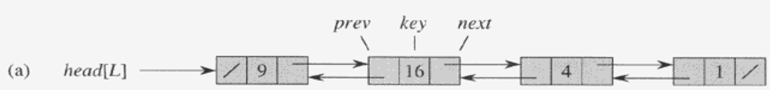
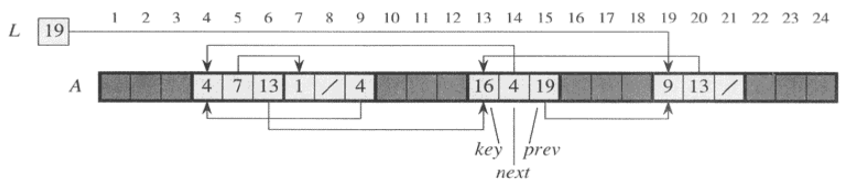


Figure 10.5 The linked list of Figure 10.3(a) represented by the arrays *key*, *next*, and *prev*. Each vertical slice of the arrays represents a single object. Stored pointers correspond to the array indices shown at the top; the arrows show how to interpret them. Lightly shaded object positions contain list elements. The variable *L* keeps the index of the head.



A Single-Array Representation of Objects

- An object occupies a contiguous set of locations in a single array ➔ A[j..k]
  - A pointer is simply the address of the first memory location of the object ➔ A[j]
  - Other memory locations within the object can bed indexed by adding an offset to the pointer ➔ 0 ~ k-j
  - Flexible but more difficult to manage



Allocating and Freeing Objects

- To insert a key into a dynamic set represented by a linked list, 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 representation so that one can be allocated
- Allocate and free homogeneous objects using the example of a doubly linked list represented by multiple arrays
  - The arrays in the multiple-array representation have length m
  - At some moment the dynamic set contains n ≤ m elements
  - The remaining m-n objects are free → can be used to represent elements inserted into the dynamic set in the future

Free List

- A singly linked list to keep the free objects
  - Initially it contains all *n* unallocated objects
- The free list is a stack
  - Allocate an object from the free list ➔ POP
  - De-allocate (free) an object ➔ PUSH
  - The next object allocated the last one freed
- Use only the *next* array to implement the free list
- A variable *free* pointers to the first element in the free list

Free List Example

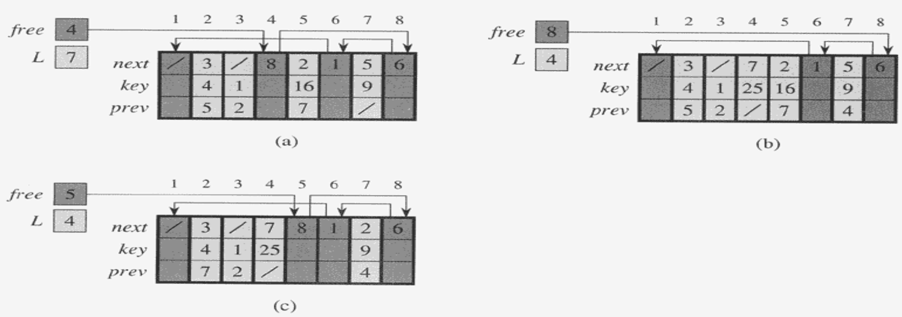


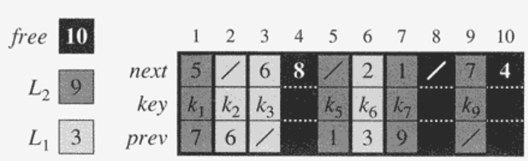
Figure 10.7 The effect of the ALLOCATE-OBJECT and FREE-OBJECT procedures. **(a)** The list of Figure 10.5 (lightly shaded) and a free list (heavily shaded). Arrows show the free-list structure. **(b)** The result of calling ALLOCATE-OBJECT() (which returns index 4), setting *key*[4] to 25, and calling LIST-INSERT(L, 4). The new free-list head is object 8, which had been *next*[4] on the free list. **(c)** 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.

Allocate And Free An Object

```
ALLOCATE-OBJECT()
if free = NIL
    then error "out of space"
    else x ← free
         free ← next[x]
         return x
```

```
FREE-OBJECT(x)
next[x] ← free
free ← x
```

Two Linked Lists L_1 and L_2, and A Free List Intertwined



Representing Rooted Trees

Binary Tree

- Use linked data structures to represent a rooted tree
  - Each node of a tree is represented by an object
  - Each node contains a *key* field and maybe other satellite fields
  - Each node also contains *pointers* to other nodes
- For binary tree…
  - Three pointer fields
    - *p*: pointer to the parent → NIL for root
    - *left*: pointer to the left child → NIL if no left child
    - *right*: pointer to the right child → NIL if no right child

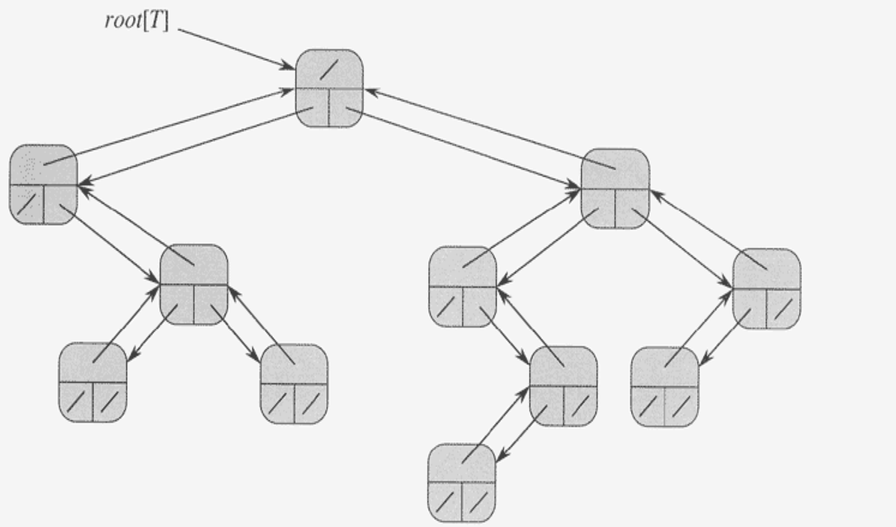
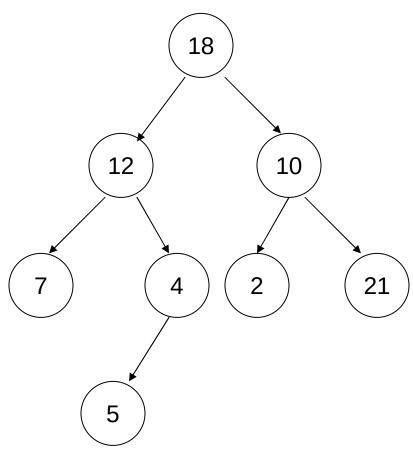


Figure 10.9 The representation of a binary tree T. Each node x has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown.

Draw the Binary Tree Rooted At Index 6

Index	Key	Left	Right
1	12	7	3
2	15	8	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



Rooted Trees With Unbounded Branches

- The representation for binary trees can be extended to a tree in which no. of children of each node is at most *k*
  - left, right ➔ $child_1$, $child_2$, …, $child_k$
- If no. of children of a node can be unbounded, or k is large but most nodes have small numbers of children…
  - Left-child, right sibling representation
    - Three pointer fields
      - *p*: pointer to the parent
      - *left-child*: pointer to the leftmost child
      - *right-sibling*: pointer to the sibling immediately to the right
    - root[T] pointer to the root of the tree
    - O(N) space for any n-node rooted tree

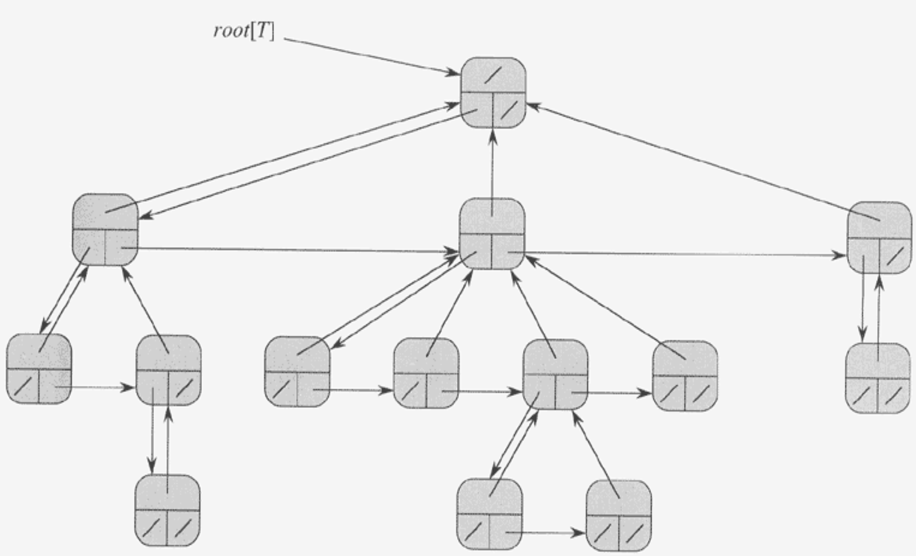


Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has fields $p[x]$ (top), *left-child*$[x]$ (lower left), and *right-sibling*$[x]$ (lower right). Keys are not shown.

Self Study

- Sentinels for linked lists: pp. 206-208