CSCI 470: Linked List, Binary Trees, Binary Search Trees

Vijay Chaudhary

Oct 02, 2023

Department of Electrical Engineering and Computer Science
Howard University

Overview

1. Linked List
2. Binary Trees & Binary Search Trees
3. BST: Querying search
4. BST: Insertion & Deletion

Linked List

Linked List

- 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 List: Example

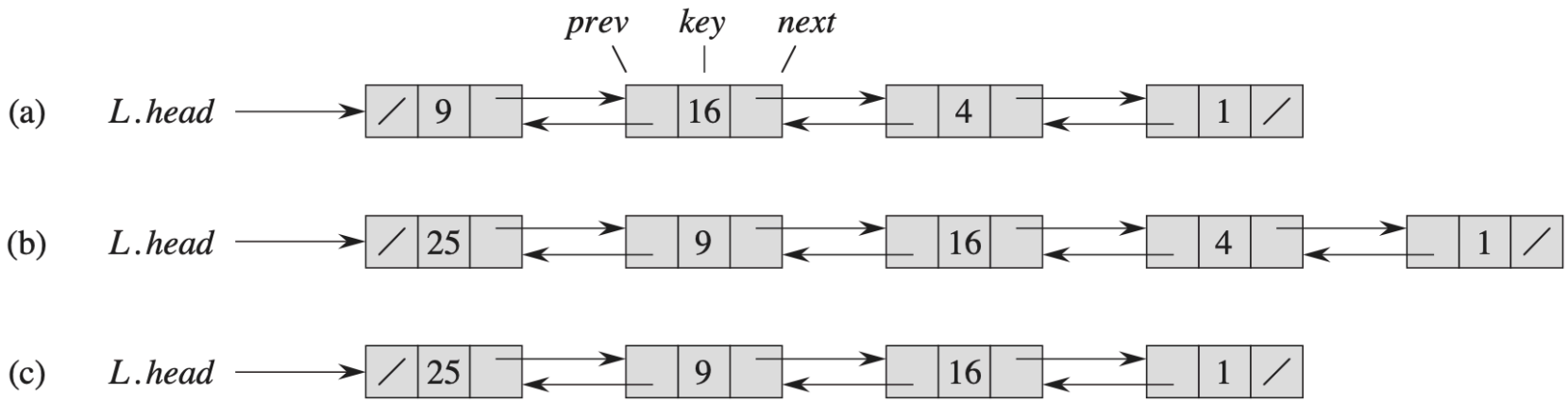

Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL,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

Linked 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
```

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

Linked 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
```

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
```

Sentinel

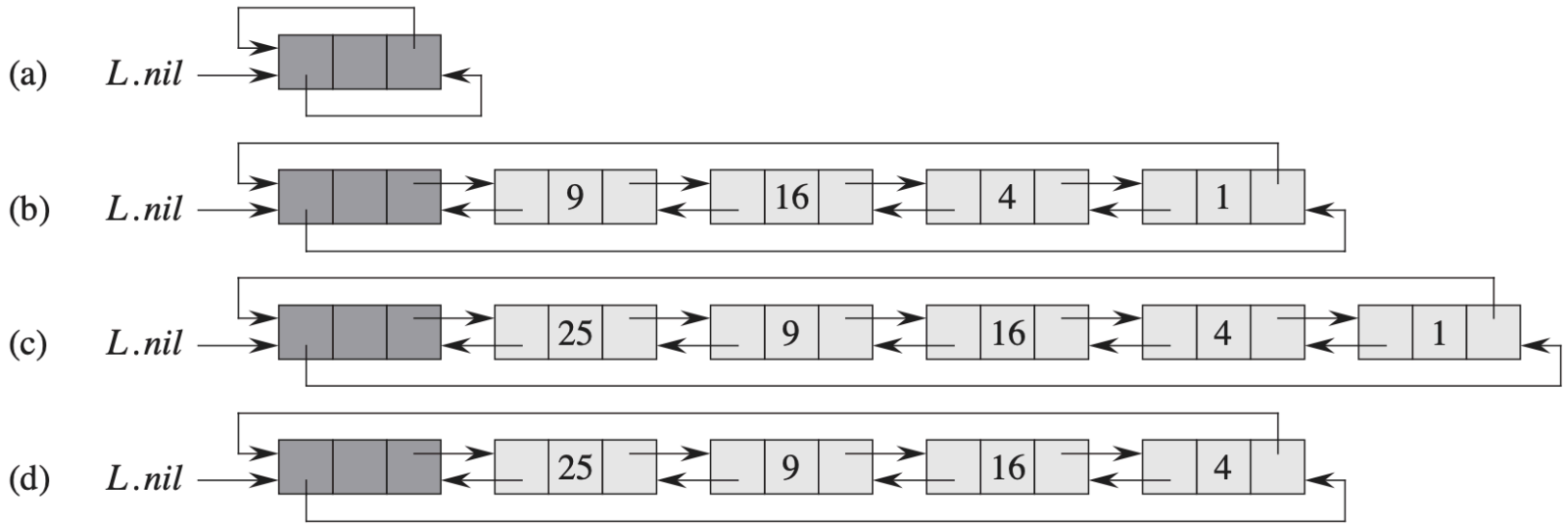

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

Linked List: Sentinel

LIST-DELETE'(L, x)

```
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

LIST-SEARCH'(L, x)

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

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
```

Linked List: Sentinel

- Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors.
- When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.
- We use sentinels only when they truly simplify the code.

Binary Trees & Binary Search Trees

Binary Trees

- attributes: p, *left*, and *right*.
- If $x.p = \text{NIL}$, then x is the root.
- If $x.left = \text{NIL}$, then x has no left child. Similarly, if $x.right = \text{NIL}$, then x has no right child. x will be a leaf, if it has no children.
- If $T.root = \text{NIL}$, then tree is empty.

Binary Tree: Figure

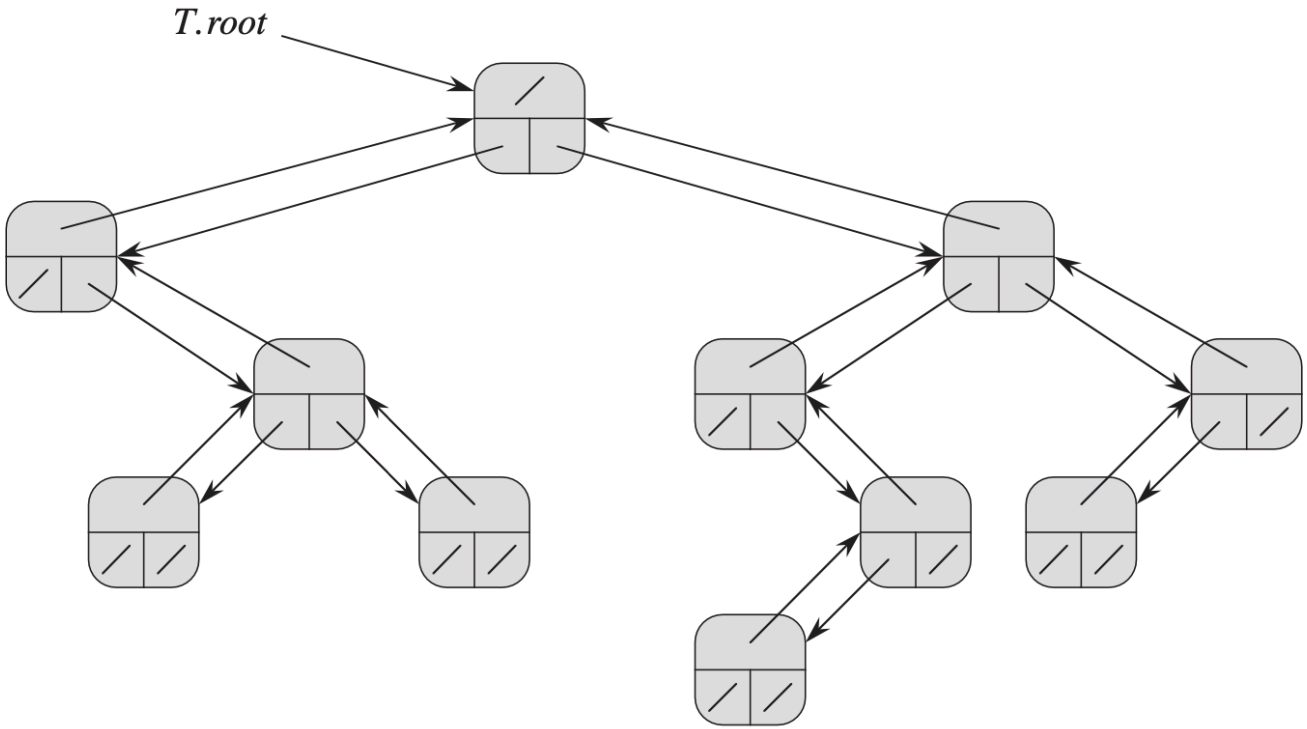

Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

Binary Search Tree

The keys in a binary search tree are always stored in such a way as to satisfy the ***binary-search-tree property***:

> *Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. if y is a node in the right subtree of x, then $y.key \geq x.key$.*

BST: Figure

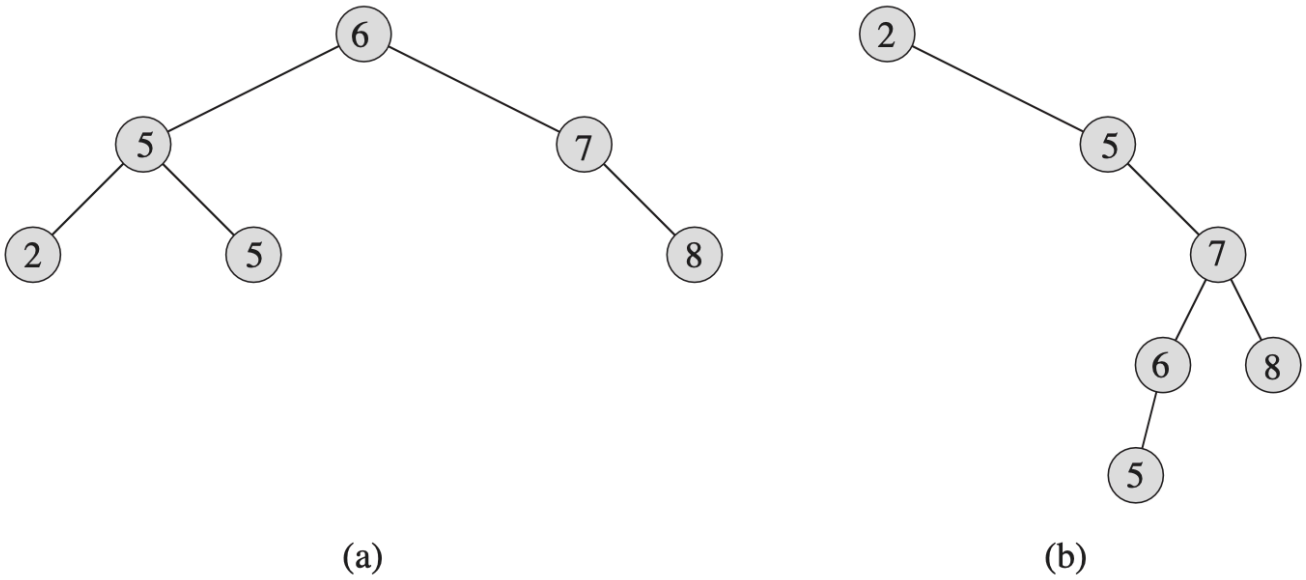

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

BST: Inorder-Tree-Walk

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2        INORDER-TREE-WALK(x.left)
3        print x.key
4        INORDER-TREE-WALK(x.right)
```

BST: Runtime

Theorem: If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

- Since INORDER-TREE-WALK visits all n nodes of the subtree, we have $T(n) = \Omega(n)$.
- It remains to show that $T(n) = O(n)$.

Proof: $T(n) = O(n)$

- In an empty tree, INORDER-TREE-WALK takes a small, constant amount of time to check if the tree is NIL or not, therefore, $T(0) = c$ for some constant $c > 0$.
- For $n > 0$, suppose that INORDER-TREE-WALK is called on a node x whose left subtree has k nodes and whose right subtree has $n - k - 1$ nodes.
- The time to perform INORDER-TREE-WALK(x) is bounded by $T(n) \leq T(k) + T(n - k - 1) + d$ for some constant $d > 0$ that reflects an upper bound on the time to execute the body of INORDER-TREE-WALK(x), exclusive of the time spent in recursive calls.

Proof

We use the substitution method to show that $T(n) = O(n)$ by proving that $T(n) \leq (c + d)n + c$.

For $n = 0$, we have $(c + d).0 + c = c = T(0)$.

For $n > 0$, we have

$$\begin{aligned} T(n) &\leq T(k) + T(n - k - 1) + d \\ &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\ &= (c + d)n + c - (c + d) + c + d \\ &= (c + d)n + c, \end{aligned}$$

which completes the proof.

BST: Querying search

BST: Querying Search

We will go over TREE-SEARCH, MAXIMUM, MINIMUM, and SUCCESSOR. In this section, we shall examine these operations and show how to support each one in time $O(h)$ on any binary search tree of height h.

BST: Searching

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else
6        return TREE-SEARCH(x.right, k)
```

Runtime ?

BST: Searching iteratively

ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL and k ≠ x.key
2        if k < x.key
3              x = x.left
4        else
5              x = x.right
6  return x
```

Runtime ?

BST: minimum and maximum

TREE-MINIMUM(x)

```
1  while x.left ≠ NIL
2        x = x.left
3  return x
```

Runtime ?

TREE-MAXIMUM(x)

```
1  while x.right ≠ NIL
2        x = x.right
3  return x
```

Runtime ?

BST: Successor

TREE-SUCCESSOR(x)

```
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

Runtime ?

BST: Insertion & Deletion

Figure: reference

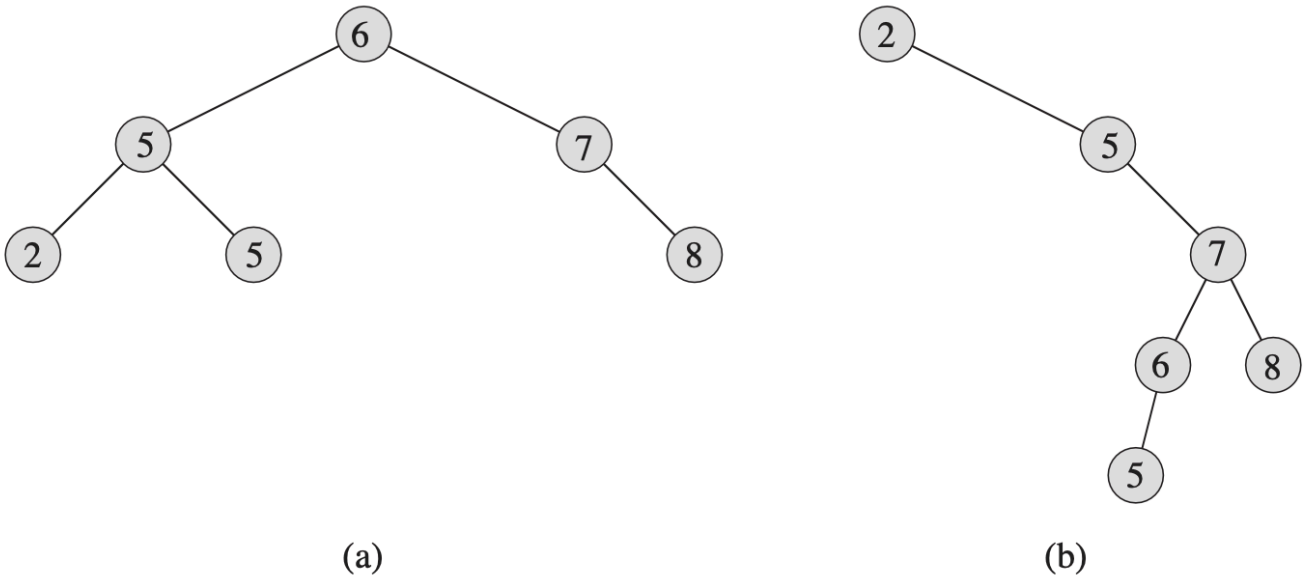

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

Insertion

TREE-INSERT(T, z)

```
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z   // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

Insertion: Figure

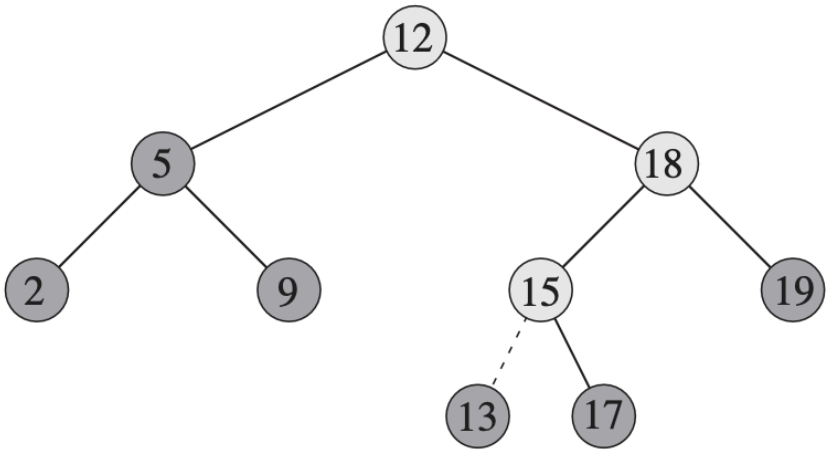

Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.