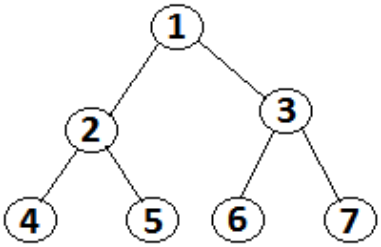
COMP2119 Introduction to Data Structures and Algorithms
Practice Problem Set 3 - Trees & Searching Algorithms

Release Date: Oct 21, 2024

Please do NOT submit the answer of this part.

Chapter 5 - Trees

1. A binary tree with Node 1 as the root is drawn below.



   (a) Answer the following questions related to tree terminology.
      (i) List out all the the children of Node 1.
      (ii) List out all the parents of Node 4.
      (iii) List out all the proper descendants of Node 1.
      (iv) List out all the proper ancestors of Node 5.

   (b) List out the visiting order of vertices by using the following tree traversal algorithms.
      (i) In-order traversal
      (ii) Pre-order traversal
      (iii) Post-order traversal
      (iv) Breath first search
      (v) Depth first search

Solution:

(a) (i) 2, 3
(ii) 2
(iii) 2, 3, 4, 5, 6, 7
(iv) 1, 2

(b) (i) 4, 2, 5, 1, 6, 3, 7
(ii) 1, 2, 4, 5, 3, 6, 7
(iii) 4, 5, 2, 6, 7, 3, 1
(iv) 1, 2, 3, 4, 5, 6, 7
(v) 1, 2, 4, 5, 3, 6, 7

2. (a) Draw the binary tree with the following node traversal order:

   - Pre-order traversal: 1, 2, 4, 5, 7, 3, 6, 8, 9, 10
   - In-order traversal: 4, 2, 7, 5, 1, 3, 9, 8, 10, 6

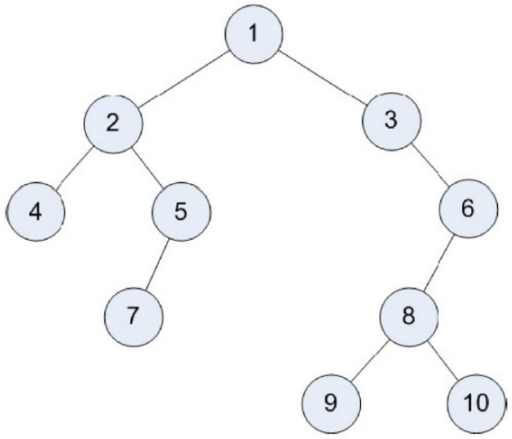
   (b) Suppose that all nodes in a binary tree have distinct values, given its pre-order traversal sequence and in-order traversal sequence, does there exist a different binary tree with the same pre-order traversal sequence and in-order traversal sequence? Explain your answer.

   (c) The following C++ code demonstrates a quadratic time recursive algorithm which constructs a tree from the given preorder and inorder traversal of a tree. Fill in the blanks to complete the code.

```
struct Node {
    int data;
    Node *left;
    Node *right;

    Node(int data) {
        this->data = data;
        left = NULL;
        right = NULL;
    }
};

int preorder[n];
int inorder[n];

// construct tree rooted at preorder[c] from inorder[start:end]
Node *constructTree(int start, int end, int &c) {
    if (1. ____________ ) {
        return NULL;
    }

    int i = start;
    while (i < end && 2. _________________) {
        i = i + 1;
    }
    c = c + 1;

    Node *root = new Node(inorder[i]);
    root->left = 3. _________________ ;
    root->right = 4. _________________ ;

    return root;
}
```

Solution:

(a) The original binary tree:

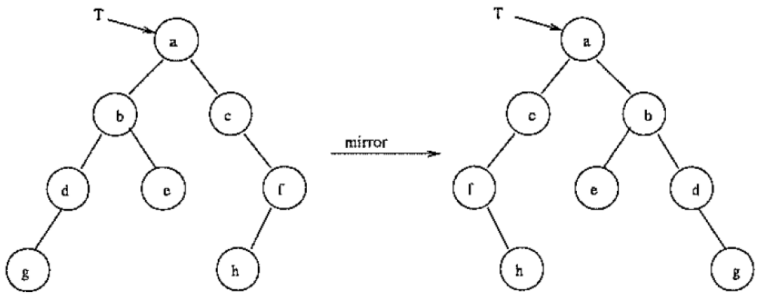


(b) If all values in the binary tree are different, then we can uniquely recover the tree from its pre-order traversal sequence and in-order traversal sequence by recursively identifying the root and the traversal sequences of its sub-trees.

Note that if there are elements of the same value in the tree, then in some cases it is impossible to recover the tree uniquely (e.g. all elements are of the same value).

(c)
```
1. start > end
2. preorder[c] != inorder[i]
3. constructTree(start, i-1, c)
4. constructTree(i+1, end, c)
```

3. A binary tree can be converted to it's mirror image. The figure below shows an example.



Furthermore, the *Node* structure below represents a binary tree node:

```
// Data structure to store a Binary Tree node
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

Given the root pointer T of a binary tree with n nodes, write the pseudocode of the algorithm Mirror(T) that converts a binary tree into its mirror image. Explain how your code works and analyze the time complexity plus extra space required by your program.

Solution:

Pseudocode:

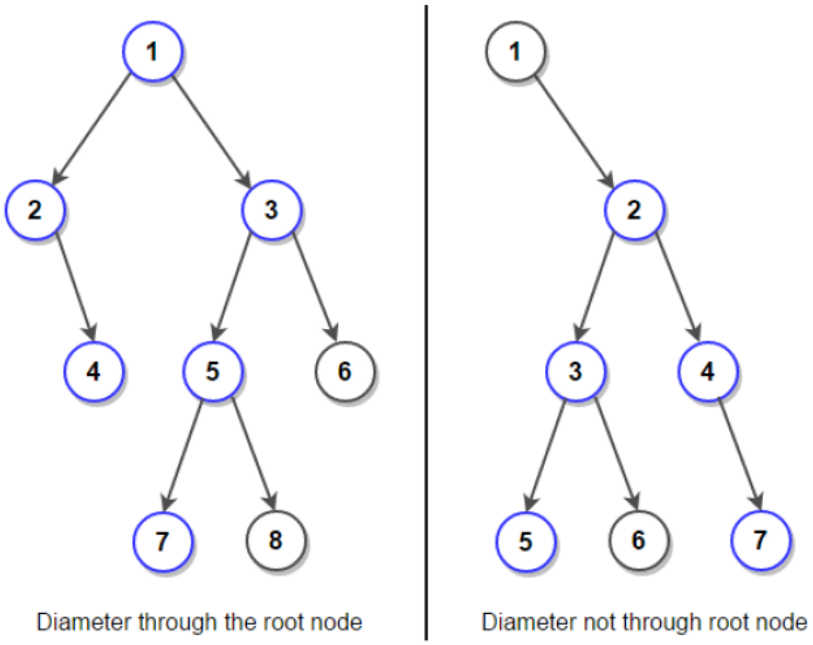
Algorithm 3.1 Produce the Mirror Image of a Binary Tree

```
1:  function MIRROR(T)                          ▷ T: Root node of a binary tree
2:      if (T == NIL) then
3:          return NIL
4:      end if
5:
6:      r = Mirror(T → left)
7:      l = Mirror(T → right)
8:
9:      T → right = r
10:     T → left = l
11:
12:     return T
13: end function
```

Explanation:
The program recursively produce the mirror image of left sub-tree and right sub-tree. Then, it swaps the position of the left and right sub-tree and returns the resultant tree.

Time Complexity & Extra Space:
Since we do constant job at each node, the time complexity is $O(n)$. The program needs O(h) extra space for the call stack, where h is the height of the tree.

4. The diameter of a binary tree is equal to the number of nodes on the longest path between any two leaves of it. For example, the figure below shows two binary trees having diameter 6 and 5 respectively (nodes highlighted in blue color).



Furthermore, the *Node* structure below represents a binary tree node:

```
// Data structure to store a Binary Tree node
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

Given the root pointer T of a binary tree with n nodes, write the pseudocode of the algorithm getDiameter(T) that computes the diameter of a binary tree. Explain how your code works and analyze the time complexity plus extra space required by your program.

Solution:

Pseudocode:

Algorithm 4.1 Get the Diameter of a Binary Tree

```
1:  function GETDIAMETER(T)                          ▷ T: Root node of a binary tree
2:
3:      if (T == NIL) then
4:          return 0, 0
5:      end if
6:
7:      leftHeight, leftDiameter = getDiameter(T → left)
8:      rightHeight, rightDiameter = getDiameter(T → right)
9:
10:     height = Max(leftHeight, rightHeight) + 1
11:     diameter = Max(Max(leftDiameter, rightDiameter), leftHeight + rightHeight + 1))
12:
13:     return height, diameter
14:
15: end function
```

Explanation:
The algorithm first find the diameter and height of the left and right sub-tree of node T recursively. Afterwards, it computes the height of the tree with node T as the root, and see if including the left and right sub-trees can result in a larger diameter. Finally, return the maximum height and diameter found.

Time Complexity & Extra Space:
The time complexity of the solution is $O(n)$, as every node is visited at most once. The program needs $O(h)$ extra space for the call stack, where h is the height of the tree.

Chapter 6 - Search Algorithms

1. Given a list L of n elements and a target element X, for which we want to test if it is present in L.

   (a) What property must L have before binary search can be applied?

   (b) Explain the principle of binary search.

   (c) State the worst-case and average-case complexities (Big-O) of binary search (in terms of number of element comparisons). Prove the worst-case result.

Solution:

(a) L must be sorted before binary search can be applied.

(b) The binary search takes advantage of the ordered nature of an array. Finding an element in an array is degenerated into finding an element in the left part of the array or the right part of the array. The value of the middle element of the array indicates the right part or left part does not contain the searched element.

Given an array with all elements in ascending order, there are three conditions to degenerate the problems:

(1) If the value of the middle element is larger than the value of the searched elements, the right half of the array must not contain the searched elements. The problem is degenerated into searching in the left half of the array.

(2) If the value of the middle element is smaller than the value of the searched elements, the right half of the array must not contain the searched elements. The problem is degenerated into searching in the right half of the array.

(3) If the value of the middle element is equal to the value of the searched elements, the searched element is found.

(c) Worst Case:
The searched element is not in the array. Let $T(n)$ denote the number of element comparisons.

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + 1 \ ; \ T(1) = 1 \\ &= T(\lfloor \frac{n}{4} \rfloor) + 2 \\ &= T(\lfloor \frac{n}{2^i} \rfloor) + i \\ &= T(1) + \lfloor \log n \rfloor \\ &= 1 + \lfloor \log n \rfloor \\ &\to O(\log n) \end{aligned}$$

Average Case:
The time complexity is $O(\log n)$.

2. Given an unsorted array A with n distinct integers, where $n \geq 2$. Write the pseudocode of the algorithm LargestSum(A) that find the largest pair sum in it within $O(n)$ time. You should also explain how your algorithm works and analyze the time complexity of it.

Example:
For array $A_1 = [12, 34, 10, 6, 40]$, the largest pair sum is $34 + 40 = 74$.

Solution:

Pseudocode:

Algorithm 2.1 LargestSum

```
1:  function LargestSum(A)
2:
3:      if (A[0] > A[1]) then                ▷ Initialize first and second largest element
4:          first = A[0]
5:          second = A[1]
6:      else
7:          first = A[1]
8:          second = A[0]
9:      end if
10:
11:     for (i = 2 to n − 1) do              ▷ Traverse remaining array to find the biggest pair sum
12:         if (A[i] > first) then
13:             second = first
14:             first = A[i]
15:         else if (A[i] > second) then
16:             second = A[i]
17:         end if
18:     end for
19:
20:     return (first + second)
21:
22: end function
```

Explanation:
This problem equals to searching the largest and second-largest element in an array. We use linear search to compare each element's value in the array with the candidate largest and second-largest element. Update the candidates if the current element is larger. At last, it returns the sum of the largest and second-largest element found.

Time Complexity:
The time complexity is $O(n)$. The first if-then-else statement uses $O(1)$ time to complete. The for loop iterates the array once in $O(n)$ time, as the statements in each iteration completes in $O(1)$ time. All in all, the overall time complexity is $O(n)$.

3. Consider a sorted array $A[1..n]$ of $n > 1$ distinct positive integers and a target positive integer x, write the pseudocode of the algorithm SLX(A, i, j, x) that returns an index k into the array segment $A[i..j]$ such that $A[k]$ is the smallest element in $A[i..j]$ that is larger than or equal to x. SLX() should return "Null" if no such element is found.

   Your algorithm should run in $O(\log m)$ time, where m is the size of array segment (i.e $m = j - i + 1$).

   Remarks:

   - You may assume that $1 \leq i \leq j \leq n$.
   - You may not assume that x is present in the array segment $A[1..n]$ (i.e. x could be present or absent in the segment).

   Solution:

   The pseudocode of SLX(A, i, j, x) is written as following:

Algorithm 3.1 SLX

```
1:  function SLX(A, i, j, x)
2:
3:      if (x ≤ A[i]) then                          ▷ If x is less than the first element of the array
4:          k = i
5:          return k
6:      else if (x > A[j]) then                     ▷ If x is greater than the last element of the array
7:          return NIL
8:      else
9:          while (i ≤ j) do
10:             middle = ⌊(i + j)/2⌋
11:
12:             if (A[middle] < x) then
13:                 i = middle + 1
14:             else if ((A[middle] ≥ x) and (A[middle − 1] < x)) then
15:                 k = middle
16:                 return k
17:             else
18:                 j = middle − 1
19:             end if
20:         end while
21:     end if
22:
23:     return NIL
24:
25: end function
```

4. Given n distinct integer points on an axis, let a_i denote the ith point, $\forall i$, $a_i \geq 0$, $a_i \leq w$. w is an integer. Now, we want to use k line segments to cover these points.

   Define L to be the length of the longest line segment. Write the pseudocode of an algorithm to return the smallest possible integer value of L. You algorithm should run in $O(n \log w)$ time. You may assume $\{a_n\}$ is already sorted in increasing order.

   Example:
   Consider the case $\{a_n\} = \{1, 2, 4, 6\}$, $k = 2$. 2 is the minimum value of the longest line segment.

   Solution:

   The pseudocode of the algorithm is written as following:

Algorithm 4.1 Find the smallest L

```
1:  function CHECK(A, k, x, n)
2:      covered = 0                          ▷ Record the index of smallest to-be-covered point
3:      count = 0                            ▷ The number of line segments used so far
4:
5:      while (covered < n) do
6:          count = count + 1
7:
8:          for (i = covered to n) do
9:              if (A[i] > A[covered] + x) then
10:                 break
11:             end if
12:         end for
13:
14:         covered = i
15:     end while
16:
17:     return (count ≤ k)
18: end function
19:
20:
21: function FINDMIN(A, k, w, n)             ▷ n is the size of array A
22:     low = 0                              ▷ The searching zone is [low , high)
23:     high = w + 1
24:
25:     while (low < high) do
26:         guess = Floor((low + high) / 2)
27:
28:         if (CHECK(A, k, guess, n) == True) then
29:             high = guess
30:         else
31:             low = guess + 1
32:         end if
33:     end while
34:
35:     return high
36: end function
```

Chapter 7 - Binary Search Trees

1. For each of the following statements, state whether it is True or False. For a False statement, point out the incorrectness.

   (a) In a binary search tree, the left and right sub trees of a node are also binary search trees.

   (b) A node in a binary search tree must have two children.

   (c) Each node of a binary search tree has a parent.

   (d) The in-order traversal of a binary search tree always outputs the data in ascending order.

   (e) The height of a binary search tree with n nodes is always less than $\log n + 1$.

   Solution:

   (a) True

   (b) False. It can have 0 to 2 children.

   (c) False. The root node has no parent.

   (d) True

   (e) False. The binary search tree may not be a full binary tree. Consider inserting 1, 2, 3, 4, 5, 6 into an empty binary search tree. The height of the tree is larger than $\log 6 + 1$.

2. A list of 12 elements are given as following:
   $[5, 4, 8, 1, 9, 7, 6, 2, 12, 11, 10, 3]$

   The following list shows the probability of each element being searched:
   $[0.1, 0.05, 0.05, 0.1, 0.2, 0.05, 0.05, 0.1, 0.05, 0.1, 0.05, 0.1]$

   For example, the element with key value 9 is searched with probability 0.2.

   (a) If you perform linear search over the list, what is the average number of elements inspected for a successful search?

   (b) Draw the binary search tree generated by inserting the elements according to the order they are listed.

   (c) Calculate the average number of nodes inspected for a successful search in this binary search tree.

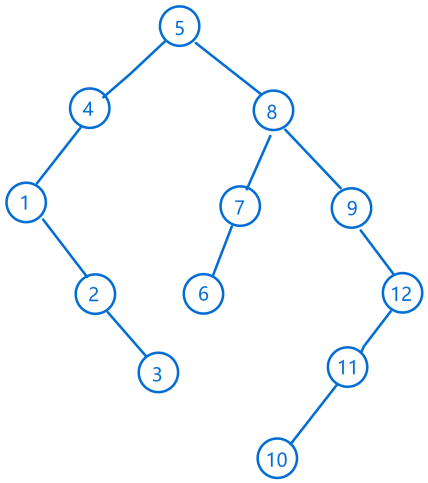
   (d) Draw the binary search tree after deleting the element '8'.

Solution:

(a) The average number of elements inspected is:

$$\begin{aligned} & (0.1)(1) + (0.05)(2) + (0.05)(3) + (0.1)(4) + (0.2)(5) + (0.05)(6) + (0.05)(7) + \\ & (0.1)(8) + (0.05)(9) + (0.1)(10) + (0.05)(11) + (0.1)(12) \\ = \ & 6.4 \end{aligned}$$

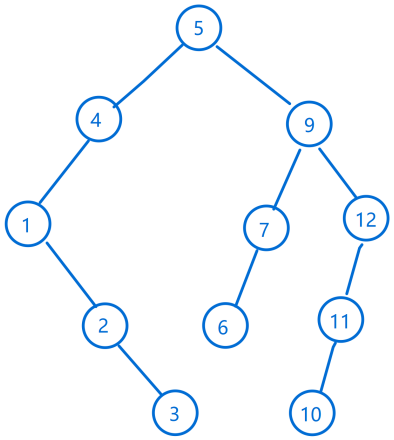
(b) The binary tree is:



(c) The average number of elements inspected is:

$$\begin{aligned} & (0.1)(1) + (0.05)(2) + (0.05)(2) + (0.1)(3) + (0.2)(3) + (0.05)(3) + (0.05)(4) + \\ & (0.1)(4) + (0.05)(4) + (0.1)(5) + (0.05)(6) + (0.1)(5) \\ = \ & 3.45 \end{aligned}$$

(d) The tree is shown as below. After deleting '8', '8' becomes the right child of '5' and '7' becomes the left child of '9'. Other reasonable answers are acceptable (e.g. 8 is replaced by 7).



3. The *Node* structure below represents a binary search tree node:

```
// Data structure to store a Binary Tree node
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

Given the root pointer T of a binary search tree with n nodes, write the pseudocode of the algorithm kthSmallest(T, k) to find the k-th smallest element in it. The algorithm must run in $O(h+k)$ time, where h is the height of the tree and k is an arbitrary constant.

Remarks:
You may assume $1 \leq k \leq$ number of BST's total elements, and k is always valid.

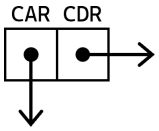
Solution:

Inorder traversal can solve this problem in $O(h + k)$ time.

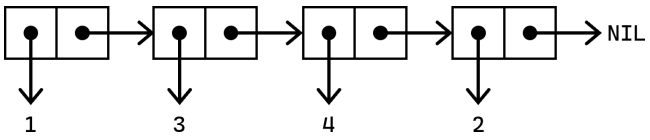
Algorithm 3.1 Find the k-th Smallest Element in a BST

```
1:  function KTHSMALLEST(T, k)                ▷ T: Root node of a BST, k: Value of k
2:
3:      p = T
4:      s = Stack()
5:
6:      while (p != NIL) do                   ▷ Go to far left of the tree
7:          s.push(p)
8:          p = p → left
9:      end while
10:
11:     cnt = 1                               ▷ Record the data ranking of the node
12:     while ((s.empty() == False) and (cnt ≤ k)) do
13:         p = s.top()
14:         s.pop()
15:         cnt = cnt + 1
16:
17:         q = p → right
18:         while (q != NIL) do
19:             s.push(q)
20:             q = q → left
21:         end while
22:     end while
23:
24:     return p → data
25:
26: end function
```

4. A cons cell is a data structure that consists of two slots, `CAR` and `CDR`. Each slot holds a value, or a reference to a cons cell.



Cons cells can be used to build a linked list. Specifically, we use `CAR` to store a value and let `CDR` points to the next node in the linked list. A null value (`NIL`) is used to represent the end of the list. For example, the linked list shown in the figure below has 4 values (1, 3, 4, 2).



Consider a system where all variables can either be an integer or a reference to a cons cell only.

- We can use Algorithm 4.1 to find the cons cell that contains an element e from a list, which is accessed via L, the first cons cell of the linked list
- We can use Algorithm 4.2 to insert an element e at the beginning of list L and return the updated list.

Algorithm 4.1 Linear Search

```
1: function FindElement(L, e)                    ▷ Find position of e in list L
2:     ptr = L
3:
4:     while (ptr ≠ NIL) and (ptr → car ≠ e) do
5:         ptr = ptr → cdr
6:     end while
7:
8:     return ptr
9: end function
```

Algorithm 4.2 Head Insert

```
1: function HeadInsert(L, e)          ▷ Insert e to list L and return the updated list
2:     return cons(e, L)
3: end function
```

The table below shows a list of operations available in such a system. Your algorithms presented for this question must not use any data structure except the cons cell, with only the operations listed in this table.

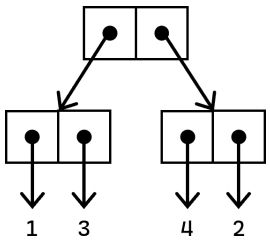
Operation	Description
`cons(`x`,` y`)`	Creates and returns a con cell
$L \rightarrow car$	Access `CAR` of cell L
$L \rightarrow cdr$	Access `CDR` of cell L

(a) Present the pseudocode of an algorithm that, given a list L, reverses the list and returns it. (Note that L might be empty.)

(b) Present the pseudocode of an algorithm that, given a list L of size 2^k, returns the root of a complete binary tree where values are stored in the leaf nodes only.

Internal nodes of the binary tree should be modelled as cons cells. For each such cell, `CAR` points to its left child and `CDR` points to its right child. Leaf nodes are con cells that contain only values.

Example:
The linked list shown in the figure on the last page should be converted to the binary tree shown below:



(c) Explain how you would use cons cells to build a binary search tree that supports the tree operations discussed in Chapter 7 of the notes. Illustrate how a tree node should be implemented (Note: You need not to follow the implementation of part (b)).

Solution:

(a) The algorithm is written below:

Algorithm 4.3 Reverse List

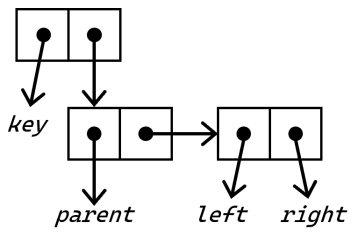
```
1: function ReverseList(L)                          ▷ Reverse a list
2:     ptr = NIL
3:     next = L
4:
5:     while (next ≠ NIL) do
6:         nextnext = next → cdr
7:         next → cdr = ptr
8:         ptr = next
9:         next = nextnext
10:    end while
11:
12:    return ptr
13: end function
```

(b) See the algorithm below for a top-down approach:

Algorithm 4.4 List to Binary Tree

```
1:  function ListToTree(L)                          ▷ Convert a list to a binary tree
2:      if (L == NIL) then
3:          return NIL
4:      else if (L → cdr == NIL) then                ▷ When k = 0 (base case)
5:          return L → car
6:      end if
7:
8:      len_ptr = L
9:      half_ptr = L
10:     prev_ptr = NIL
11:
12:     while (len_ptr ≠ NIL) do        ▷ Move half_ptr once when len_ptr moves twice
13:         len_ptr = len_ptr → cdr
14:         prev_ptr = half_ptr
15:         half_ptr = half_ptr → cdr
16:
17:         if (len_ptr ≠ NIL) then
18:             len_ptr = len_ptr → cdr
19:         end if
20:     end while
21:
22:     prev_ptr → cdr = NIL                                  ▷ Split list
23:     left = ListToTree(L)
24:     right = ListToTree(half_ptr)
25:     prev_ptr → cdr = half_ptr                ▷ Optional: restore changes
26:
27:     return cons(left, right)
28: end function
```

(c) To support the operations, a tree node must contains 4 fields, *key*, *parent*, *left*, and *right*. A possible solution is to keep using `CAR` to store the value, then use `CDR` to store a list of 3 fields by chaining 2 cons cells, as shown in the figure below.



5. In order to support keys of the same values to be inserted into a binary search tree (BST), the tree nodes, as well as the algorithm `Tree_Insert` algorithm in Chapter 7 notes can be modified in the following way:

   - Keep a Boolean flag in each node, initialized to `false`.
   - Right before line 6 of algorithm `Tree_Insert` (`if` $(z \to key < x \to key)$ `...`), check if $z \to key$ equals $x \to key$.
     - If that is the case, flip the flag of x.
     - Then, check the flag of x:
       - ◇ if it's `true`, move to the left child;
       - ◇ otherwise, move to the right child.
     - Skip the remaining of the while loop and continue to the next iteration.

   (a) Suppose the values $\{17, 23, 13, 19, 29, 23, 23, 23, 23\}$ are inserted into an empty BST (in that order) by following the modified algorithm. Draw the resulting BST.

   (b) Design an algorithm in $O(1)$ space that, given the root of a BST T built using the modified algorithm in part (a) and a key x, returns the number of nodes with keys x in the tree. Briefly explain how your algorithm works.

   Remarks:

   - The space needed to maintain the recursive call does not count for the $O(1)$ space requirement.

   (c) Consider another modification as follows:

   - Maintain a linked list in each node, initially empty. (A `NULL` pointer)
   - Right before line 6 of algorithm `Tree_Insert`, check if $z \to key$ equals $x \to key$.
     - If so, insert the key to the front of the linked list kept in the node and return.

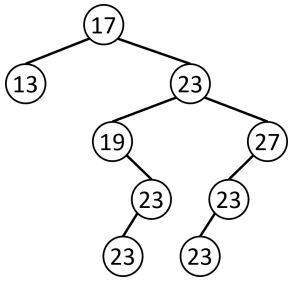
   There are n keys with kn distinct values stored in the BST now (where $0 \leq k \leq 1$). Comment on the space efficiency of the two modifications with respect to k.

   Remarks:

   - You may assume all values and pointers takes 4 bytes each, and a tree node contains 4 fields, key, $parent$, $left$, and $right$.
   - You may assume that the Boolean flag in the modification in part (a) takes no space.

Solution:

(a) Note that the the last part of algorithm `Tree_Insert` also needs to be updated to follow the direction of insertion introduced by the modification.



(b) We can recursively find the count of the two children whenever the key is found as shown in the algorithm below.

Algorithm 5.1 Count Keys

```
1: function TreeCount(T, x)                                   ▷ Count key x in T
2:     if (T == NIL) then
3:         return 0
4:     end if
5:
6:     if (T → key == x) then
7:         return 1 + TreeCount(T → left, x) + TreeCount(T → right, x)
8:     else if (T → key < x → key) then
9:         return TreeCount(T → right, x)
10:    else
11:        return TreeCount(T → left, x)
12:    end if
13: end function
```

(c) For the modified version in part (a), each node contains 4 fields and so it takes 16 bytes per node. There will always be n nodes and so the total amount of memory needed is $16n$ bytes.

For the modified version in part (c), each tree node requires an extra pointer for the linked list, so it takes 20 bytes per tree node. For the linked list, there will be 2 fields and so it takes 8 bytes per list node. There will always be kn tree nodes and $n-kn$ list nodes. Therefore, the total amount of memory needed is $20kn+8(n-kn) = 8n+12kn$ bytes.

The use of linked list improved the space efficiency, but that can only happen if there are many duplicated keys. As otherwise more tree nodes will be needed but the tree nodes in this modification are less space efficient.

Suppose $16n = 8n + 12kn$, then $k = 2/3$. That means when $k < 2/3$, it will be more space efficient to use the modification in part (c).

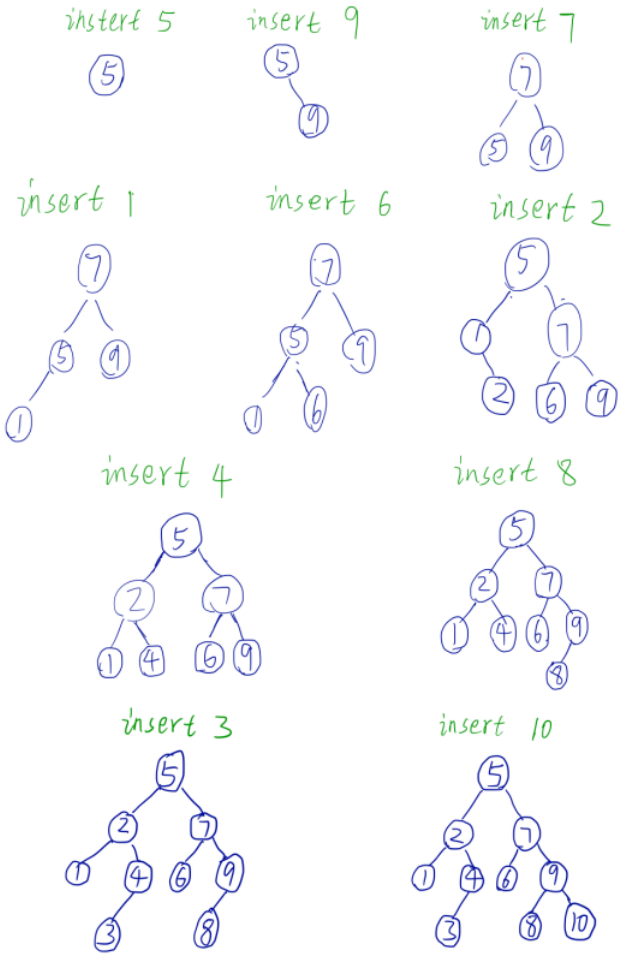
Chapter 8 - Balanced Search Trees

1. (a) 5, 9, 7, 1, 6, 2, 4, 8, 3, 10 (in that order) are inserted into an initially empty AVL tree. Draw the tree after each insertion.

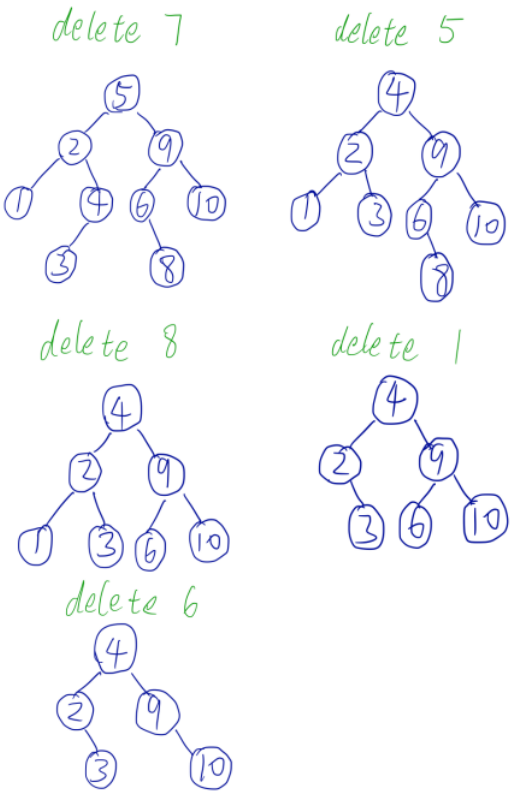
   (b) 7, 5, 8, 1, 6 (in that order) are deleted from the AVL tree constructed in part (a). Draw the tree after each deletion.

Solution:

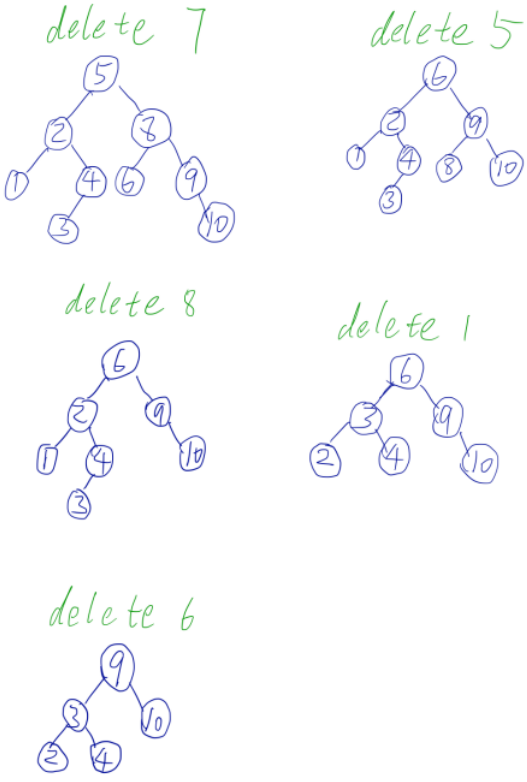
(a) The trees after each iteration are shown below:



(b) Method 1:



Method 2:



2. You have a group of m objects where each object has a distinct price p_i and value v_i ($1 \leq i \leq m$, p_i, v_i are positive integers).

   Design a data structure that takes $O(m)$ space to store the objects, such that it supports the following operation: Given two prices x and y ($x < y$), return the most valuable object (i.e. the one with the highest value) whose price is between x and y ($x < p_i < y$). The operation should take $O(\log m)$ time.

   In your answer, please write clearly on:

   - The construction method, details (e.g., type of data structure, attributes, etc.) and space complexity of the data structure.
   - The pseudocode of the algorithm to find the most valuable object.
   - Verbal explanations on how the algorithm works and it's time complexity.

   Remarks:

   - You may assume that there is at least one object with the price $x < p_i < y$.
   - You can directly use conclusions from lecture notes without proof.

   Solution:

   Data Structure:
   Construct a balanced BST to store the objects, where object prices are used as the key. Suppose the subtree rooted at node n is S_n. Each node n corresponds to an object and stores the following attributes:

   - *price*: the price of the current object;
   - *value*: the value of the current object;
   - *min_price*: the minimum price of objects in S_n;
   - *max_price*: the maximum price of objects in S_n;
   - *max_value*: the maximum value of objects in S_n;
   - $left$: pointer to the left child (or null);
   - *right*: pointer to the right child (or null);

   We can perform any kind of traversal for each subtree to find the minimum price, maximum price, and maximum value of objects.

   As each object takes constant spaces (five values, and optionally pointer to children (at most 2)), the total space to store all objects is $O(m)$.

Pseudocode:
For the input,
root: root of the balanced BST, x: lower bound of price, y: upper bound of price.

Algorithm 2.1 findMostValuableObject

```
1:  function findMostValuableObject(root, x, y)
2:      l_node = root
3:      r_node = root
4:      hv = 0
5:
6:      while (l_node != Null) do
7:          if (x ≤ l_node.price) then
8:              if ((l_node.right != Null) && (l_node.right.max_price < y) &&
9:                  (x < l_node.right.min_price)) then
10:                 hv = Max(hv, l_node.right.max_value)
11:             end if
12:
13:             if ((l_node.price < y) && (x < l_node.price)) then
14:                 hv = Max(hv, l_node.value)
15:             end if
16:
17:             l_node = l_node.left
18:         else
19:             l_node = l_node.right
20:         end if
21:     end while
22:
23:     while (r_node != Null) do
24:         if (y ≥ r_node.price) then
25:             if ((r_node.left != Null) && (r_node.left.max_price < y) &&
26:                 (x < r_node.left.min_price)) then
27:                 hv = Max(hv, r_node.left.max_value)
28:             end if
29:
30:             if ((r_node.price < y) && (x < r_node.price)) then
31:                 hv = Max(hv, r_node.value)
32:             end if
33:
34:             r_node = r_node.right
35:         else
36:             r_node = r_node.left
37:         end if
38:     end while
39:
40:     return hv.
41: end function
```

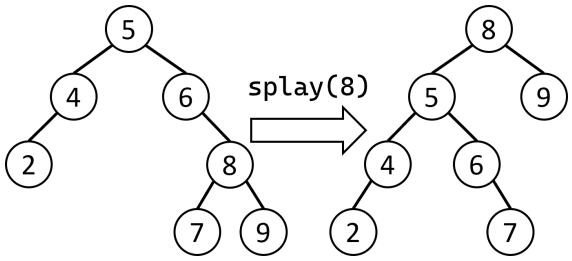
Explanation:
The algorithm works by going down the balanced BST. For the first while loop, the l_node starts from the root and checks the price of the node. If the price is smaller than x, l_node goes to the right. If the price is larger than x, l_node goes to the left. In this case, if its right sibling has all prices between x and y, the maximum value hv will be updated as the maximum value of the subtree rooted at its right sibling. If there is a price in the subtree rooted at its right sibling out of range of x and y, we skip updating the maximum value hv.

For the second while loop, the r_node starts from the root and checks the price of the node. If the price is larger than y, r_node goes to the left. If the price is smaller than y, r_node goes to the right. In this case, if its left sibling has all prices between x and y, the maximum value hv will be updated as the maximum value of the subtree rooted at its left sibling. If there is a price in the subtree rooted at its left sibling out of range of x and y, we skip updating the maximum value hv.

Time Complexity:
As the tree is balanced, the height of the tree is $O(\log m)$. In both while loops, we go down by one level each time, so there are at most $2 \log m$ iterations. In each iteration, there are operations finished in constant time. Therefore, the algorithm finishes in $O(\log m)$ time.

3. Consider a binary search tree (BST) with a number of distinct keys inserted. Some of these keys may be searched much more frequently than the others. In order to improve the average search time, one can consider moving a more popular key to the root of the BST.

   Given a node x in a BST T, let `splay(`x`)` be an operation that transforms T into another BST T' such that (1) T and T'contain the same set of keys and (2) x is the root of T'. The figure below shows an example of the splay operation:



   (a) Design an algorithm for `splay(`x`)`. For each node of a BST, it contains 4 fields: *key*, *parent*, $left$, and *right*. You may use the various rotation operations discussed in class for AVL tree as subroutines. You do not need to maintain the tree as an AVL tree.

   (b) Consider the following sequence of operations on an initially empty BST:

   insert 8,
   insert 1,
   search 1,
   insert 5,
   insert 6,
   search 5,
   search 6,
   insert 10,
   insert 12,
   insert 13,
   search 13,
   search 1,
   search 1

   Suppose splaying is not applied but the tree is balanced as an AVL tree. Count the number of key matching for the search operations.

   (c) Repeat the operations in part (b) on an initially empty BST without balancing the tree, instead, apply splaying after every operation. Count the number of key matching for the search operations.

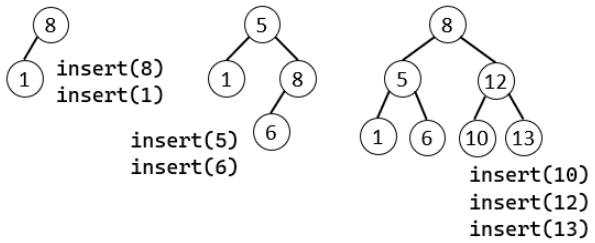
Solution:

(a) The algorithm is written as follows:

Algorithm 3.1 Spary Node x in BST T

```
1: function TreeSplay(T, x)                          ▷ Spray node x in BST T
2:     while (x → parent ≠ NIL) do
3:         y = x → parent
4:         if (y → left == x) then
5:             RIGHT_ROTATE(T, y)
6:         else
7:             LEFT_ROTATE(T, y)
8:         end if
9:     end while
10: end function
```

(b) The insertion process:

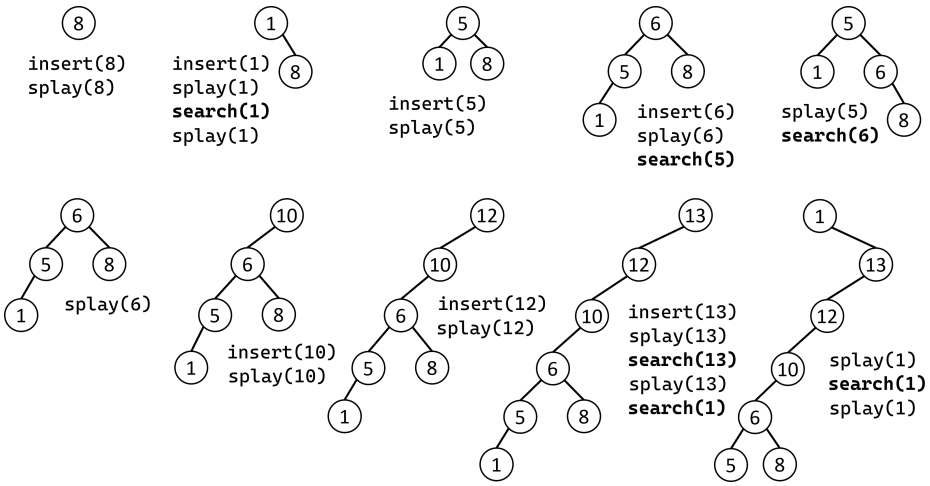


Number of key matching:

insert	1	5	6	13	1	1
match count	2	1	3	3	3	3

Total number of key matching is 15.

(c) The insertion process:



Number of key matching:

insert	1	5	6	13	1	1
match count	1	2	2	1	6	1

Total number of key matching is 13.

Further Practice from the Textbook:
To gain more practice, here are a list of supplementary problems you may attempt from the course textbook **Introduction to Algorithms, 3rd edition**:

Chapter 5 - Trees

1. Basic Concepts:
   Exercise 10.4-1

2. Tree Traversal Algorithms:
   Exercise 10.4-2, 10.4-3, 10.4-4, 10.4-5, 10.4-6

Chapter 6 - Searching

1. Linear Search:
   Exercise 2.2-3

2. Binary Search:
   Exercise 4.5-3

Chapter 7 - Binary Search Trees

1. Basic Concepts:
   Exercise 12.1-1, 12.2-1, 12.2-5, 12.2-6

2. Tree-traversal Algorithms:
   Exercise 12.1-3, 12.1-4, 12.2-2, 12.2-3, 12.3-1, 12.3-2

Chapter 8 - Balanced Trees

1. Implementation of Balanced Trees:
   Exercise 13.3, 13.4