Zoom Poll

W2 Review of Heaps

Which Stages of Heap Sort use the MaxHeapify algorithm (also called Bubble Down)?

A. Going from the unsorted array to the max heap.

B. Going from the max heap to the sorted array.

C. Both

D Neither

Zoom Poll

More Review of Heaps

HeapSort uses MaxHeapify (Bubble down) to sort an array into ascending order. To sort into descending order, what needs to change?

A. Use bubble-up instead of bubble-down to build the heap.

B. Bubble-down the larger values to work with a min heap.

C. Extract from the leaves of the maxheap first.

D. You can only sort into non-descending order.

Dictionary ADT

- Insert (S, x)
- Search (S, k)
- Delete (S, x)

Both the key & the value

assume here that we can directly access the item in the data-structure.

- Today we will examine a number of different data-structures to implement this ADT. Each time we will consider the worst-case complexity of each operation.

For Insert ⇒ it doesn't include the time needed for search

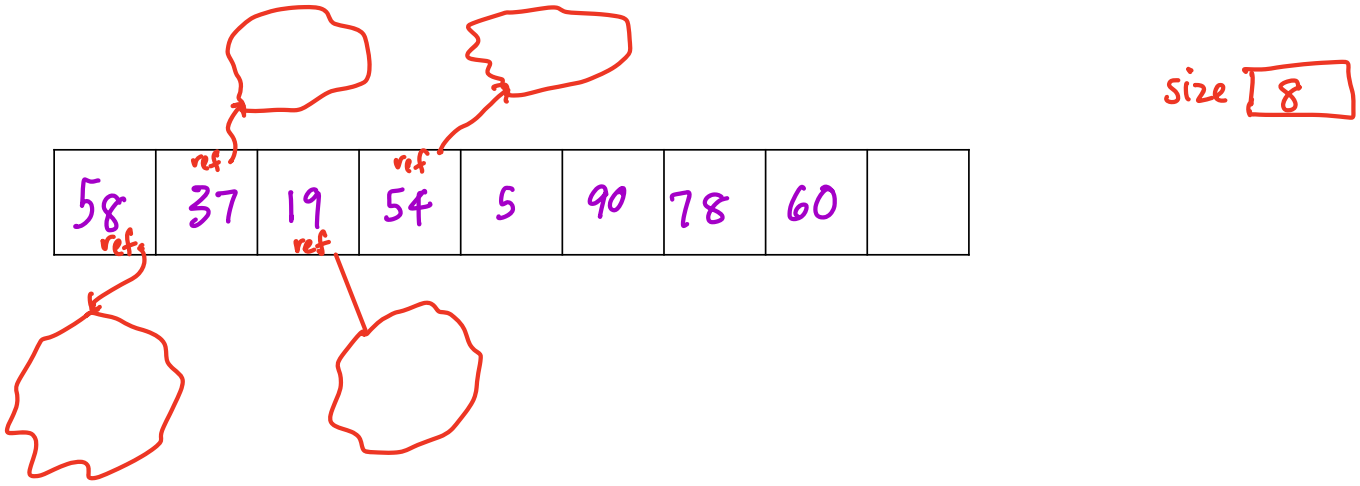
Delete ⇒ it doesn't include the time needed for search

Data Structure	Search	Insert	Delete
Unsorted array	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Sorted array	$\Theta(\log n)$	$+\Theta(n)$	$+\Theta(n)$
Unsorted linked list	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Sorted linked list	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$
Direct-access table	$\Theta(1)$	$+\Theta(1)$	$+\Theta(1)$
Hash table	$\Theta(1)$?	$+\Theta(1)$	$+\Theta(1)$ * BEST CASE
Binary Search Tree	$\Theta(n)$	$+\Theta(1)$	$+\Theta(n)$
Balanced Search Tree	$\Theta(\log n)$	$+\Theta(\log n)$	$+\Theta(\log n)$

Times for Insert and Delete are **in addition to** the time to first Search for the element.

Chalk Talk

Approach 1: An unsorted Array



Data Structure	Search	Insert	Delete
Unsorted array	$\Theta(n)$	$+\Theta(1)$	$+\Theta(1)$

Worksheet

Approaches 2-4

In your breakout groups, discuss the algorithms for the next three approaches in the table (Q1-3) and fill in their runtime. Put any questions here: https://tinyurl.com/breakouts-263

Data Structure	Search	Insert	Delete
Sorted array	$\Theta(\log n)$	$+ n$	$+ n$
Unsorted linked list	n	$+ 1$	$+ 1$
Sorted linked list	n	$+ 1$	$+ 1$

the element value is the index of the element.

PROBLEM → lot of wasted memory

Approach 5: A Direct-access Table

					5				
									19
							37		
				44					
				54				58	
60									

Data Structure	Search	Insert	Delete
Direct-access table	$\theta(1)$	$\theta(1)$	$\theta(1)$

Approach 5: A Direct-access Table

maximum number of items in the dictionary at any one time.

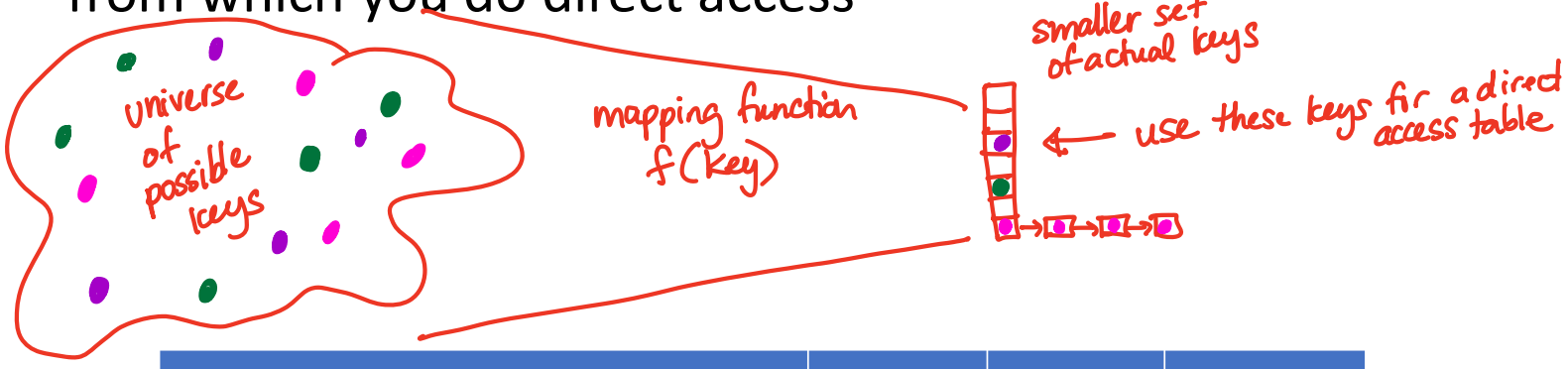
- Everything else we have considered takes O(n) space
- Direct-access tables need one location per possible key

Suppose keys are 32-bit integers

need $\Omega(2^{32})$ space.

Approach 6: Hash Table

- Map all possible keys onto a smaller set of actual keys from which you do direct access



Data Structure	Search	Insert	Delete
Hash table	$\Theta(1)$?	$\Theta(1)$?	$\Theta(1)$?

PROBLEM: 2 keys in actual keys map to the same hash table spot

In best case, $\Theta(1)$ for search, insert, delete
worst case, they all collide

Approach 6: Hash Table

- But wait? Does that actually work? What if two items I want to store in the dictionary have keys that map to the same new key?

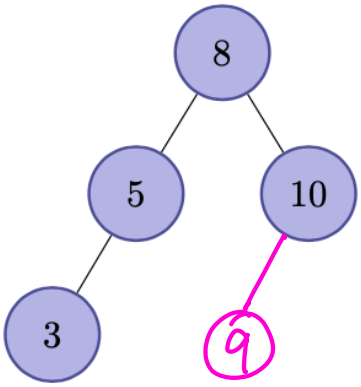
Need strategy for handling collisions

↑ when 2 items have different keys that map to same bucket

- Worst case could be:

Data Structure	Search	Insert	Delete
Hash table → linked list	$\Theta(n)$	$+1$	$+1$

Approach 7: A Binary Search Tree



O(h)
↑
height of tree

Data Structure	Search	Insert	Delete
Binary Search Tree worstcase	O(n)	+ 1	O(n)

finding a successor or predecessor

- worst case height is n.
- best case height is log(n)

Approach 8: A Balanced Binary Search Tree

If we could keep the BST balanced, then we could have $\Theta(\log(n))$ search and $\Theta(\log(n))$ delete

red-black trees, AVL trees, 2-3-4 trees, B trees

Data Structure	Search	Insert	Delete
Balanced Search Tree	$\Theta(\log n)$	$+\Theta(\log(n))$	$+\Theta(\log(n))$

Dictionary using Binary Search Trees

- BST = binary tree with "BST ordering" (fully sorted): for every node,

elements in left subtree $\leq$ element at node $\leq$ elements in right subtree

in a dictionary no duplicate keys $<$ $<$

  - dictionary S stores only S.root (also S.size usually)
  - TreeNode has members
  - .item (element stored in node)
    - .left and .right (children)

- Recursive code for operations.

```
INSERT(S, x):
    # Helper may need to modify root itself: have it return value.
    S.root <- TREE-INSERT(S.root, x)

TREE-INSERT(root, x):
    if root is NIL: # x.key not already in S
        # Found insertion point: create new node with empty children.
        root <- TreeNode(x)
    else if x.key < root.item.key:
        root.left <- TREE-INSERT(root.left, x)
    else if x.key > root.item.key:
        root.right <- TREE-INSERT(root.right, x)
    else:  # x.key == root.item.key
        root.item <- x  # just replace root's item with x
    return root
```

Worksheet

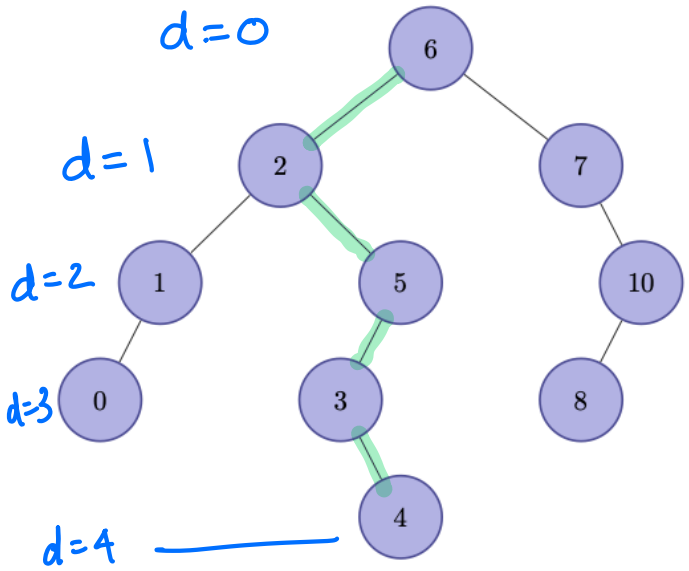
Reviewing Binary Search Tree Insert

Questions 4-6 from the worksheet

Put questions and solutions here:
http://tinyurl.com/breakouts-263

Worksheet Solutions

Question 4

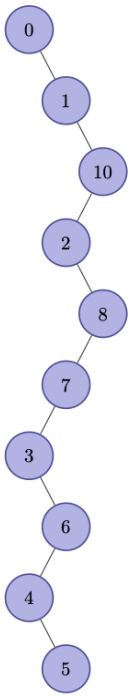


height → longest path from root to leaf. = 4

→ largest depth

Worksheet Solutions

Question 5



height = 9

Worksheet Solutions

Question 6

{0, 1, ……, 8, 9, 10}

[each time we insert, we need to pick the max or min of remaining items.]

```
SEARCH(S, k):
       return TREE-SEARCH(S.root, k)

TREE-SEARCH(root, k):
       # Compare against root.item.key to determine if k belongs in left or
       # right subtree. Return node with node.item.key = k.
       if root is NIL:  # k not in S
           pass
       else if k < root.item.key:
           root <- TREE-SEARCH(root.left, k)
       else if k > root.item.key:
           root <- TREE-SEARCH(root.right, k)
       else:  # x.key == root.item.key
           pass  # root is the node we want
       return root
```

Worksheet

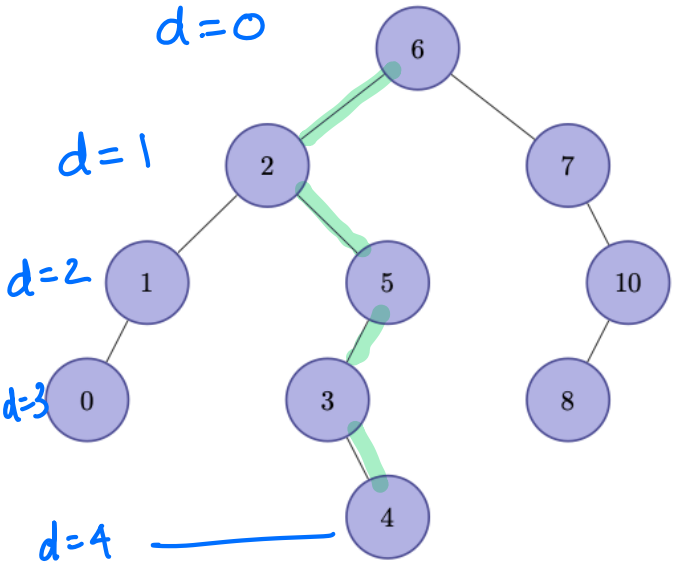
Reviewing Binary Search Tree Search

Questions 7 – 9 from the worksheet on expected number of calls to TREE-SEARCH under different assumptions about the input and different BST.

Put questions and solutions here:
http://tinyurl.com/breakouts-263

Worksheet Solutions

Question 7



$$E[\#\text{ calls to tree search}] = \sum_{\text{nodes}} (\text{prob of node})(\#\text{ of calls node})$$

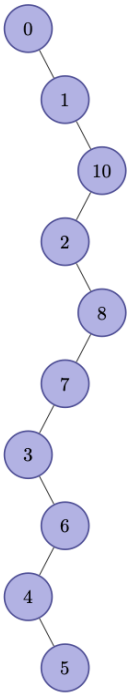
$$= \sum_{\text{depth } d} (\text{prob of node at } d)(d+1)$$

$$= \frac{1}{10}\left[(1)(1)+(2)(2)+(3)(3)+3(4)+1(5)\right]$$

$$= \frac{1}{10}[31] = 3.1$$

Worksheet Solutions

Question 8 & 9

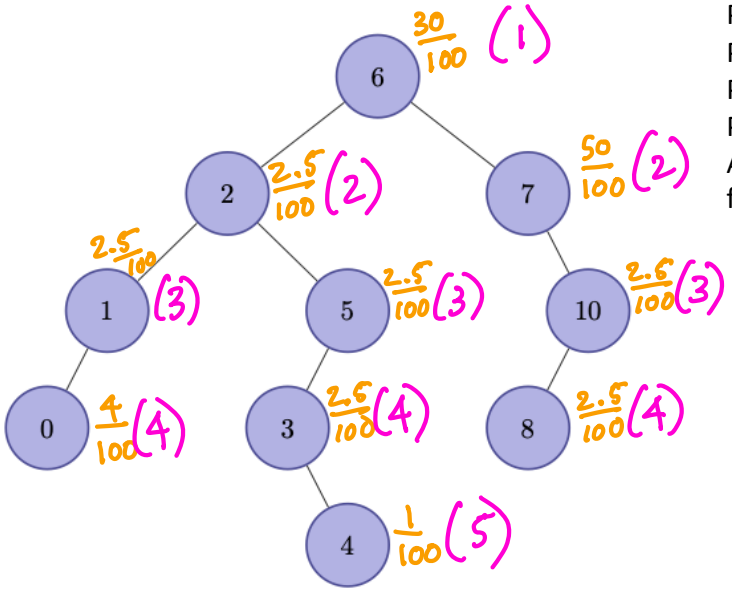


$$E[\text{calls to tree search}] = \frac{1}{10}\sum_{k=1}^{10} k \qquad \frac{10(11)}{2}$$

$$= \frac{10(11)}{10(2)} = 5.5$$

Extra Exercise

What if we weren't picking uniformly?



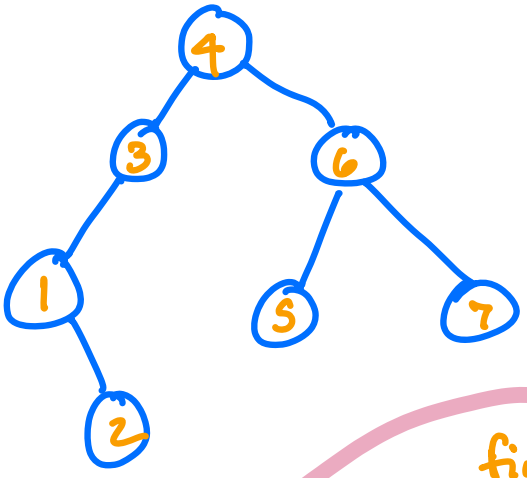
Pr[x=4] = 1/100
Pr[x=7] = ½ = 50/100
Pr[x=0] = 1/25
Pr[x=6] = 3/10
Always searches for key in the tree. Likelihood of searching for any of the remaining values is equal.

$$E[\#\text{ calls to tree search}] = \sum_{\text{all nodes}} (\text{prob of searching for that node})(\#\text{ calls to tree search for that node})$$

DELETE(S, x):

    # Helper may need to modify root itself: have it return value.

    S.root <- TREE-DELETE(S.root, x)



must use successor in 263.

↓

min value in right subtree

Case 0 : node has 0 children

leaf : 2, 5, 7

Case 1 : node has 1 child

1, 3

delete the node and only child takes it's place promote

case 2 : node has 2 children

find successor, put its value in the node we wanted to delete & then delete successor.

```
TREE-DELETE(root, x):
        if root is NIL:  # x.key not in S -- should not happen!
            pass  # nothing to remove
        else if x.key < root.item.key:
            root.left <- TREE-DELETE(root.left, x)
        else if x.key > root.item.key:
            root.right <- TREE-DELETE(root.right, x)
        else:  # x.key == root.item.key so Remove root.item.
            if root.left is NIL:
                root <- root.right  # NIL if both children missing
            else if root.right is NIL:
                root <- root.left
            else:
                # Root has two children: remove element with smallest key in right subtree and
                # move it to root. Assumption: DELETE-MIN returns two values: element removed
                # from right subtree and root of resulting subtree (in case root.right changes).
                root.item, root.right <- DELETE-MIN(root.right)
        return root
```

```
DELETE-MIN(root):
        # Remove element with smallest key in root's subtree;
        # return element removed and root of resulting subtree.
        if root.left is NIL:
            # Root stores item with smallest key; replace with right child.
            return root.item, root.right
        else:
            # Left subtree not empty: root not the smallest.
            item, root.left <- DELETE-MIN(root.left)
            return item, root
```

worst situation:
- linkedlists
- Θ(height of tree)

in the worst tree, height is n

Meta-lessons to Remember

- When we delete a node, we need a strategy to restore the remaining nodes to a valid BST.
- Worst-case running time for delete is:
- Major drawback?

⇒ order of insertion determines shape

worst case Θ(height of tree)

in the worst tree height is n

worst case $\Theta(n)$ → even if we already searched could still need $\Theta(n)$ to find successor

What if we allowed duplicate keys?

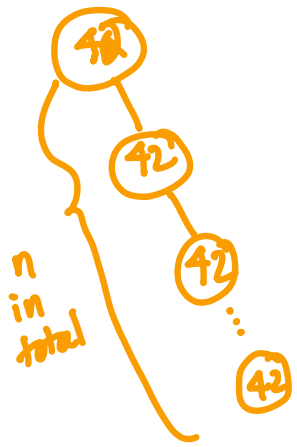
- Our definition of Dictionaries had keys that were distinct, but what if we allowed duplicates?
- We will consider ways to change your algorithm to allow a BST to hold duplicate values

For each strategy:

- change pseudo code
- Analyze total time taken to insert n identical items into an empty tree

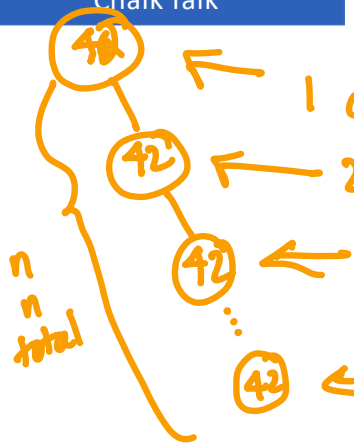
Approach 1: Always go right

```
INSERT(S, x):
      # Helper may need to modify root itself: have it return value.
      S.root <- TREE-INSERT(S.root, x)

   TREE-INSERT(root, x):
      if root is NIL:  # x.key not already in S
          # Found insertion point: create new node with empty children.
          root <- TreeNode(x)
      else if x.key < root.item.key:
          root.left <- TREE-INSERT(root.left, x)
      else: if x.key > root.item.key:
          root.right <- TREE-INSERT(root.right, x)
      else:  # x.key == root.item.key
          root.item <- x  # just replace root's item with x
      return root
```



Chalk Talk

Analysis of Approach 1



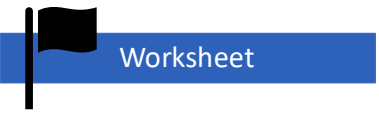
C time

2C time

3C time

nC time

$$\text{total time} = \sum_{k=1}^{n} kC = C\sum_{k=1}^{n} k = C\frac{n(n+1)}{2} = \Theta(n^2)$$



Approaches 2-4

Worksheet Questions 10-12

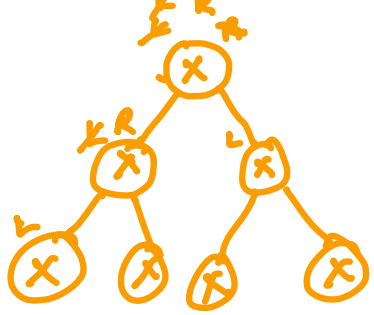
2. Strictly alternate using a flag to keep track

3. Randomly pick a side

4. Keep a chain of values inside the node

Chalk Talk

Analysis of Approach 2



When we insert all identical values, time to insert into a tree is $\Theta(\log m)$ for a tree that currently has m values.

total time $= c(\log 1) + c(\log 2) + c(\log 3) + \cdots + c(\log n)$

$$= c\sum_{i=1}^{n} \log(i) < c\sum_{i=1}^{n} \log(n) = c\, n \log(n) \Rightarrow O(n \log n)$$

$$c\sum_{i=1}^{n} \log(i) > c\sum_{i=\frac{n}{2}}^{n} \log\left(\frac{n}{2}\right) = c\frac{n}{2}\log\left(\frac{n}{2}\right) = \frac{cn}{2}(\log(n) - 1) \Rightarrow \Omega(n \log n)$$

$$\therefore \Theta(n \log n)$$

Final Reminders

Things to take away

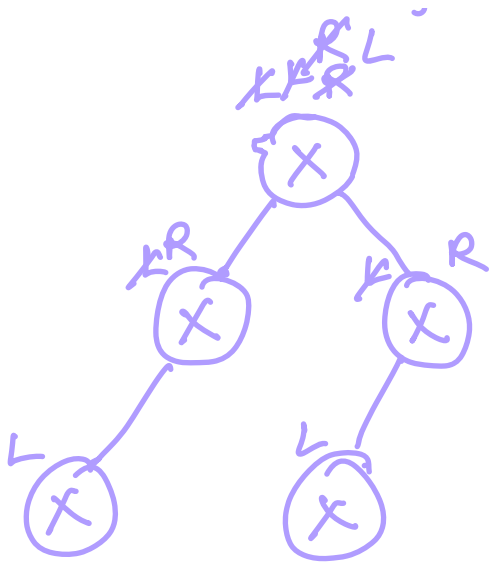
1. Hashing seems promising ← week 6.
2. Combining 2 data structures together seems promising ← week 5
3. Balanced trees could be a big improvement to DS based on trees. ← Wk 4

$\log\left(\frac{n}{2}\right) = \log n - 1$

$= \log n + \log\left(\frac{1}{2}\right)$

$= \log n - 1$

$\log(ab) = \log a + \log b$



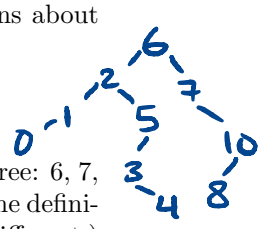
Below is a copy of the table in the lecture slides, as the lecture progresses fill in this copy if you aren't already taking notes on the slides. The time taken for insert should be the time **in addition to** the time to first search for the element. Delete assumes that you already have a direct pointer to the item to be deleted from the data structures.

Approach	Search	Insert	Delete
Unsorted array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Unsorted linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Sorted linked list	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
Direct-access table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$ → best case linked list (unsorted) WC
Binary Search Tree	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Balanced Search Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

1. As a group, discuss how we would use a sorted array to implement the Dictionary ADT. Be specific about the algorithms that you would use for each of the operations and then fill in the table above. Enter the asymptotic time for search in the Google doc at `http://tinyurl.com/breakouts-263`

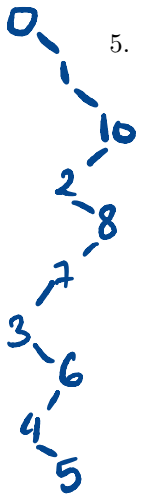
2. Next discuss how we would use an unsorted linked list to implement the Dictionary ADT. Again, discuss how each operation would work and add its worst-case complexity to your table. Enter the name of the slowest operation in the Google Doc.

3. Finally discuss how the operations would change if you kept the linked list in sorted order. Does it change anything in the worst-case? What about the average case? In both this question and the last, we didn't specify if the linked list was singly-linked or doubly-linked. Did your group make any assumptions about this and does it matter? Hint – think specifically about Delete.



4. Draw the binary search tree that results from inserting the following values in order into an empty tree: 6, 7, 10, 2, 5, 3, 1, 4, 8, 0. We will call the resulting tree $T1$. What is the height of $T1$? Make sure to use the definition of height from the CLRS textbook and not the definition from the 148 course notes (which is different.) Enter the height of $T1$ into the breakouts Google spreadsheet at `http://tinyurl.com/breakouts-263`.

$h = 4$



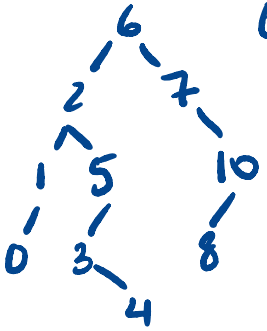
5. Draw the binary search tree that results from inserting the following values in order into an empty tree: 0, 1, 10, 2, 8, 7, 3, 6, 4, 5. We will call the resulting tree $T2$. What is the height of $T2$? Enter the height of $T2$ into the same breakouts Google spreadsheet.

h = 9

6. What must be true about the order of the input so that the resulting tree is the maximum possible height? Discuss this until you are certain that everyone in your group understands the answer.

sorted → ascending or descending

7. Consider calling SEARCH($T1$,x) where x is chosen uniformly from the values in the tree $\{0, 1, ..., 8, 10\}$ Notice that there is no 9 and that there are ten values in total. What is the average number of calls to TREE-SEARCH that are performed? For your convenience, TREE-SEARCH is at the bottom of this handout. Put your answer in `http://tiny-url.com/breakouts-263`.



$$E[\#\text{ of calls}] = \frac{(1\times1)+(2\times2)+(3\times3)+(4\times3)+(5\times1)}{10}$$

$$= \frac{1+4+9+12+5}{10} = \frac{31}{10} = 3.1$$

8. Redo question 7 on $T2$. After you have completed the calculation, put your answers to Q6 and Q7 in the Google doc. Then discuss as a group why these numbers are so different. Could we design a tree $T3$ that could have an even lower number of expected calls to TREE-SEARCH? How?

$$E[\#\text{ of calls}] = \frac{(1)+2+\cdots+10}{10}$$

$$= \frac{\frac{10(11)}{2}}{10} = \frac{55}{10} = 5.5$$

9. Redo this same calculation for the expected number of calls to TREE-SEARCH when searching for x on $T1$ but now assume that x is chosen uniformly from the set $\{0, 1, 2, ..., 9, 10\}$.

5 calls when searching for 9

$$E[\#\text{ of calls}] = \frac{31+(1\times5)}{11} = \frac{36}{11}$$

10. In class we have been discussing strategies for changing the TREE-INSERT code to allow the resulting binary search tree to contain duplicate values. We made changes to the pseudo code so that a duplicate item was always inserted in the right subtree of the duplicated item and we determined that the total running time for inserting n identical keys into an empty tree was $c\frac{n(n-1)}{2}$ which is $\Theta(n^2)$.

    The first strategy for you to discuss as a group is to use a boolean flag to strictly alternate between going right or left each time you encounter a duplicate key. Explore this strategy by doing three things:

    (1) Make changes on the pseudo code to implement this strategy

    (2) Draw the tree that would result from inserting n identical items into an empty tree

    (3) Analyze the total running time for these n insertions.

    Suggestion: Have one group member share their screen in the breakout room and display the original pseudo code. A copy is posted on the lecture slides Quercus page.

11. Repeat the previous question again doing all three parts. This time explore the strategy of randomly choosing the right or left subtree for insertion with equal probabilities for left and right. For the analysis of total running time, discuss both the worst-case and average-case.

12. The third strategy to consider is modifying the node itself so that it stores a chain of elements that share the same key. Spend any remaining breakout time talking about how this would work. Answer all three parts as before. How would this affect Search and Delete?

```
SEARCH(S, k):
        return TREE-SEARCH(S.root, k)

TREE-SEARCH(root, k):
        # Compare against root.item.key to determine if k belongs in left or
        # right subtree. Return node with node.item.key = k.
        if root is NIL: # k not in S
            pass
        else if k < root.item.key:
            root <- TREE-SEARCH(root.left, k)
        else if k > root.item.key:
            root <- TREE-SEARCH(root.right, k)
        else: # x.key == root.item.key
            pass # root is the node we want
        return root
```

SOLUTIONS: Solutions to questions 1-5 were discussed in lectures and can be found by watching the video or looking at the annotated lecture notes that are posted. Solutions to remaining worksheet problems are included here except for question 12 which is left as an optional exercise for the reader.

1. As a group, discuss how we would use a sorted array to implement the Dictionary ADT. Be specific about the algorithms that you would use for each of the operations and then fill in the table above. Enter the asymptotic time for search in the Google doc at `http://tinyurl.com/breakouts-263`

2. Next discuss how we would use an unsorted linked list to implement the Dictionary ADT. Again, discuss how each operation would work and add its worst-case complexity to your table. Enter the name of the slowest operation in the Google Doc.

3. Finally discuss how the operations would change if you kept the linked list in sorted order. Does it change anything in the worst-case? What about the average case? In both this question and the last, we didn't specify if the linked list was singly-linked or doubly-linked. Did your group make any assumptions about this and does it matter? Hint – think specifically about Delete.

4. Draw the binary search tree that results from inserting the following values in order into an empty tree: 6, 7, 10, 2, 5, 3, 1, 4, 8, 0. We will call the resulting tree $T1$. What is the height of $T1$? Make sure to use the definition of height from the CLRS textbook and not the definition from the 148 course notes (which is different.) Enter the height of $T1$ into the breakouts Google spreadsheet at `http://tinyurl.com/breakouts-263`.

5. Draw the binary search tree that results from inserting the following values in order into an empty tree: 0, 1, 10, 2, 8, 7, 3, 6, 4, 5. We will call the resulting tree $T2$. What is the height of $T2$? Enter the height of $T2$ into the same breakouts Google spreadsheet.

6. What must be true about the order of the input so that the resulting tree is the maximum possible height? Discuss this until you are certain that everyone in your group understands the answer.

   SOLUTION: Each input must be either the maximum or minimum of the remaining not-yet-inserted elements.

7. Consider calling SEARCH($T1$,x) where x is chosen uniformly from the values in the tree $\{0, 1, ..., 8, 10\}$ Notice that there is no 9 and that there are ten values in total. What is the average number of calls to TREE-SEARCH that are performed? For your convenience, TREE-SEARCH is at the bottom of this handout. Put your answer in `http://tiny-url.com/breakouts-263`.

   SOLUTION: There is one call to TREE-SEARCH when we search for a 6 because it is at the root. There are two calls to TREE-SEARCH when we search for a 2 or 7. The number of calls to TREE-SEARCH depends on the depth d of the node in the tree. A node at depth d takes $d + 1$ calls.

$$\begin{aligned} E[\text{calls to TREE-SEARCH}] &= \frac{1}{10} \sum_{\text{all d}} (d+1) * \text{number of nodes at depth d}) \\ &= \frac{(1)(1) + (2)(2) + (3)(3) + (3)(4) + (1)(5)}{10} \\ &= \frac{31}{10} \\ &= 3.1 \end{aligned}$$

8. Redo question 7 on $T2$. After you have completed the calculation, put your answers to Q6 and Q7 in the Google doc. Then discuss as a group why these numbers are so different. Could we design a tree $T3$ that could have an even lower number of expected calls to TREE-SEARCH? How?

   SOLUTION: There is one node at each depth.

$$\begin{aligned} E[\text{calls to TREE-SEARCH}] &= \frac{1}{10} \sum_{d=0}^{9} d + 1 \\ &= \frac{1}{10} \sum_{d=1}^{10} d \\ &= \frac{1}{10} \frac{(10)(11)}{2} \\ &= 5.5 \end{aligned}$$

9. Redo this same calculation for the expected number of calls to TREE-SEARCH when searching for x on $T1$ but now assume that x is chosen uniformly from the set $\{0, 1, 2, ..., 9, 10\}$.

   SOLUTION: When we search for a 9 it takes 5 calls to TREE-SEARCH to determine that 9 is not in $T1$.

$$\begin{aligned} E[\text{calls to TREE-SEARCH}] &= \frac{1}{11} \sum_{\text{all nodes}} \text{calls made for that node} \\ &= \frac{(1)(1) + (2)(2) + (3)(3) + (3)(4) + (1)(5) + 1(5)}{11} \\ &= \frac{36}{11} \\ &= 3.27 \end{aligned}$$

10. In class we have been discussing strategies for changing the TREE-INSERT code to allow the resulting binary search tree to contain duplicate values. We made changes to the pseudo code so that a duplicate item was always inserted in the right subtree of the duplicated item and we determined that the total running time for inserting n identical keys into an empty tree was $c\frac{n(n-1)}{2}$ which is $\Theta(n^2)$.

    The first strategy for you to discuss as a group is to use a boolean flag to strictly alternate between going right or left each time you encounter a duplicate key. Explore this strategy by doing three things:

    (1) Make changes on the pseudo code to implement this strategy

    (2) Draw the tree that would result from inserting n identical items into an empty tree

    (3) Analyze the total running time for these n insertions.

    Suggestion: Have one group member share their screen in the breakout room and display the original pseudo code. A copy is posted on the lecture slides Quercus page.

    SOLUTIONS:

```
TreeInsert(root,x):
      if root is NIL:
          root <- TreeNode(x)
      elif x.key < root.key:
          root.left <- TreeInsert(root.left, x)
      elif x.key > root.key:
          root.right <- TreeInsert(root.right, x)
      else:
          if root.goLeft:
              root.left <- TreeInsert(root.left, x)
          else:
              root.right <- TreeInsert(root.right, x)
          root.goLeft <- not root.goLeft
      return root
```

    Runtime: The worst-case running time of one call to TreeInsert on a BST with m keys is Θ(height). When all keys are equal, this is $\Theta(\log m)$: each node has subtrees of roughly equal size (+/-1) because insertion alternates strictly between left and right children.

    To make the calculation simpler, say the running time for each call is exactly $C\log_2(m+1)$, for some constant C (the "+1" is to simplify the case $m = 0$). When inserting n identical keys, the total running time of all n calls to TreeInsert is therefore equal to:

$$C\log_2 1 + ... + C\log_2 n = C\sum_{i=1}^{n}\log_2 i.$$

    We can bound this expression as follows:

$$\begin{aligned} C\sum_{i=1}^{n}\log_2 i &<= C\sum_{i=1}^{n}\log_2 n \\ &= Cn\log_2 n \\ C\sum_{i=1}^{n}\log_2 i &>= C\sum_{i=n/2}^{n}\log_2(n/2) \\ &= C(n/2)\log_2(n/2) \\ &= (C/2)n(\log_2 n - 1) \end{aligned}$$

    Hence, the total is $\Theta(n\log n)$.

11. Repeat the previous question again doing all three parts. This time explore the strategy of randomly choosing the right or left subtree for insertion with equal probabilities for left and right. For the analysis of total running time, discuss both the worst-case and average-case.

    SOLUTIONS:

```
TreeInsert(root,x):
            if root is NIL:
                root <- TreeNode(x)
            elif x.key < root.key:
                root.left <- TreeInsert(root.left, x)
            elif x.key > root.key:
                root.right <- TreeInsert(root.right, x)
            else:
                if random(0,1] <= 0.5:
                    root.left <- TreeInsert(root.left, x)
                else:
                    root.right <- TreeInsert(root.right, x)
            return root
```

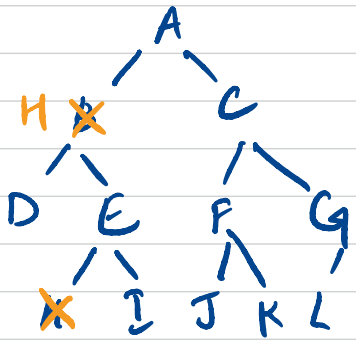
Runtime: The worst case degenerates to the same as the first approach that we discussed in lecture ($\Theta(n^2)$), when all random choices are the same. The expected case becomes the same as question 10, because on average we expect the choices to be roughly equally balanced between left and right at each node.

```
SEARCH(S, k):
        return TREE-SEARCH(S.root, k)

TREE-SEARCH(root, k):
        # Compare against root.item.key to determine if k belongs in left or
        # right subtree. Return node with node.item.key = k.
        if root is NIL: # k not in S
            pass
        else if k < root.item.key:
            root <- TREE-SEARCH(root.left, k)
        else if k > root.item.key:
            root <- TREE-SEARCH(root.right, k)
        else: # x.key == root.item.key
            pass # root is the node we want
        return root
```

```
INSERT(S, x):
    # Helper may need to modify root itself: have it return value.
    S.root <- TREE-INSERT(S.root, x)

TREE-INSERT(root, x):
    if root is NIL: # x.key not already in S
        # Found insertion point: create new node with empty children.
        root <- TreeNode(x)
    else if x.key < root.item.key:
        root.left <- TREE-INSERT(root.left, x)
    else if x.key > root.item.key:
        root.right <- TREE-INSERT(root.right, x)
    else:  # x.key == root.item.key
        root.item <- x  # just replace root's item with x
    return root
```

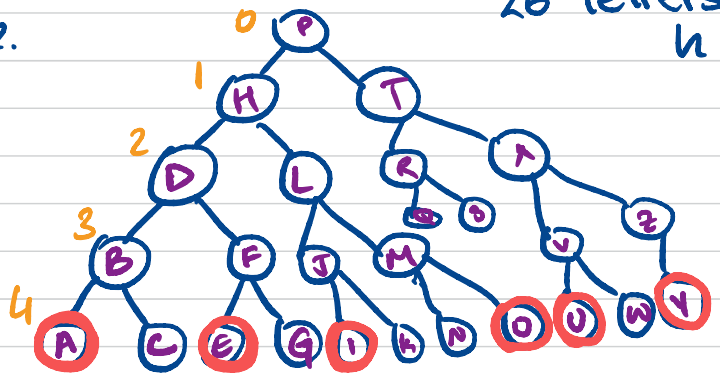
WEEK 3 PREP

1.



Delete(B)
→ successor takes its place
H is successor

2.



26 letters : A → Z $A < B < \cdots < Z$
$h = 5$

$$E[\text{nodes visited}] = \frac{[(1\times1)+(2\times2)+(3\times4)+(4\times8)+5(11)]}{26} = 4$$

Vowel = {A, E, I, O, U, Y} = 6
consonent = 20

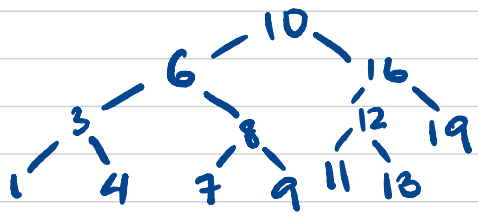
$$E(\text{nodes examined for consonent search}) = \frac{[(1\times1)+(2\times2)+(3\times4)+(4\times8)+(5\times5)]}{20} = 3.7$$

$$E(\#\text{ nodes examined for vowel}) = \frac{(5\times6)}{6} = 5$$

P(Vowel) = 0.4
P(consonent) = 0.6

$$E[\#\text{ nodes searched}] = (0.4\times5)+(0.6\times3.7) = 4.22$$

3.



max no. of calls = 4+1 = 5

NOTES ON AVL TREES

by Vassos Hadzilacos

Binary search trees work well in the average case, but they still have the drawback of linear worst case time complexity for all three operations (SEARCH, INSERT and DELETE).

Definition: A binary tree of height h is *ideally height-balanced* if every node of depth $< h - 1$ has two children.

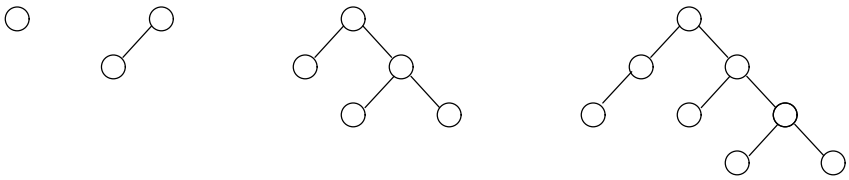
It would be nice if we could keep the binary search tree ideally height-balanced at all times. Then a tree of n nodes would be guaranteed to have height $h = \lfloor \log_2 n \rfloor$, so searches would always take time in $O(\log n)$. But insertions and deletions might destroy the ideally height-balanced property, and a reorganisation (to make the tree ideally height-balanced again, while maintaining the binary search tree property) might take as much as linear time.

AVL (or height-balanced) trees are a happy compromise between arbitrary binary search trees and ideally height-balanced binary search trees. The name "AVL" comes from the names of the two Soviet mathematicians, Adelson-Velski and Landis, who devised them.

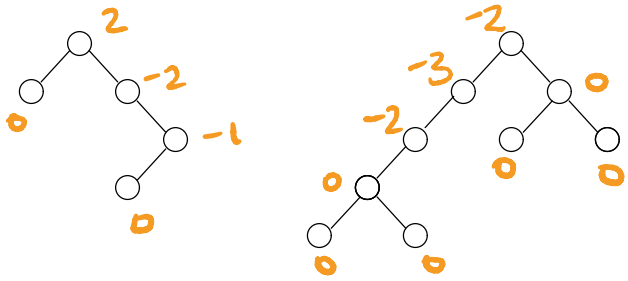
Definition: A binary tree is *height-balanced* if the heights of the left and right subtrees of *every* node differ by at most one. An *AVL tree* is a height-balanced binary search tree.

Note: By convention, the height of an *empty* binary tree (one with 0 nodes) is -1; the height of a tree consisting of a single node is 0.

Examples:



Non-examples:



Good news:

- The worst case height of an AVL tree with n nodes is $1.44 \log_2(n+2)$. Thus, the SEARCH operation can be carried out in $O(\log n)$ time in the worst case.
- Insertions and deletions can also be done in $O(\log n)$ time, while preserving the "AVL-ness" of the tree.
- Empirical studies show that AVL trees work very well on the average case too.

Bad news: The algorithms for insertion and deletion are a bit complex.

Definition: Let h_R and h_L be the heights of the right and left subtrees of a node m in a binary tree respectively. The *balance factor* of m, $BF[m]$, is defined as $BF[m] = h_R - h_L$.

For an AVL tree, the balance factor of any node is -1, 0, or $+1$.

- If $BF[m] = +1$, m is *right heavy*.
- If $BF[m] = -1$, m is *left heavy*.
- If $BF[m] = 0$, m is *balanced*.

In AVL trees we will store $BF[m]$ in each node m. When we draw AVL trees we will put a "+", "−", or "0" next to each node to indicate, respectively, that the node's balance factor is $+1$, -1, or 0.

Next we consider algorithms for the SEARCH, INSERT and DELETE operations in AVL trees.

THE ALGORITHM FOR SEARCH

We simply treat T as an ordinary binary search tree — there is nothing new to say here.

THE ALGORITHM FOR INSERT

To insert a key x into an AVL tree T, let us first insert x in T as in ordinary binary search trees. That is, we trace a path from the root downward, and insert a new node with key x in it in the proper place, so as to preserve the binary search tree property. This may destroy the integrity of our AVL tree in that

- the addition of a new leaf may have destroyed the height-balance of some nodes, and,
- the balance factors of some nodes must be updated to take into account the new leaf.

We will address each of these points in turn.

Rebalancing an AVL tree after Insertion

The height-balance property of a node may have been destroyed as a result of the insertion of the new leaf in two ways:

(1) the new leaf increased the height of the right subtree of a node that was already right heavy (before the insertion); or,

(2) the new leaf increased the height of the left subtree of a node that was already left heavy (before the insertion).

These two cases are illustrated in Figures 1(a) and (b). Note that the insertion of the new leaf can affect the balance factors *only* of its ancestors. To see this, observe that the height of any node that is *not* an ancestor of the new leaf is the same as before the insertion; consequently the heights of the left and right children of such a node are the same as before the insertion. Node m in Figure 1 is assumed to be the *minimum height* ancestor of the new leaf which is no longer height balanced as a result of the insertion.

Since the two cases are symmetric (one is obtained from the other by changing every reference of "right" to "left", and of "+" to "−", and *vice versa*), we shall only consider case (1) in detail. There are two ways in which (1) could arise, illustrated in Figure 2(a) and (b) respectively. The balance factors indicated for A and B are *after* the insertion of the new node.

The subtree shown in Figure 2(a) can be rebalanced by a simple transformation called "single left rotation" on node m. This transformation is illustrated in Figure 3: In 3(a) we copied the subtree of Figure 2(a), and 3(b) shows the result of the single left rotation on that subtree.

Note that this transformation has the following properties.

S.1 It rebalances the subtree rooted at node m (so that subtree becomes height-balanced again).

S.2 It maintains the binary search tree property.

S.3 It can be done in constant time: only a few pointers have to be switched around. As an exercise, write a program that implements this rotation, given a pointer to node m, assuming the standard representation for binary trees.

S.4 It keeps the height of m equal to its height *before* the insertion of the new node, namely, height $h + 2$.

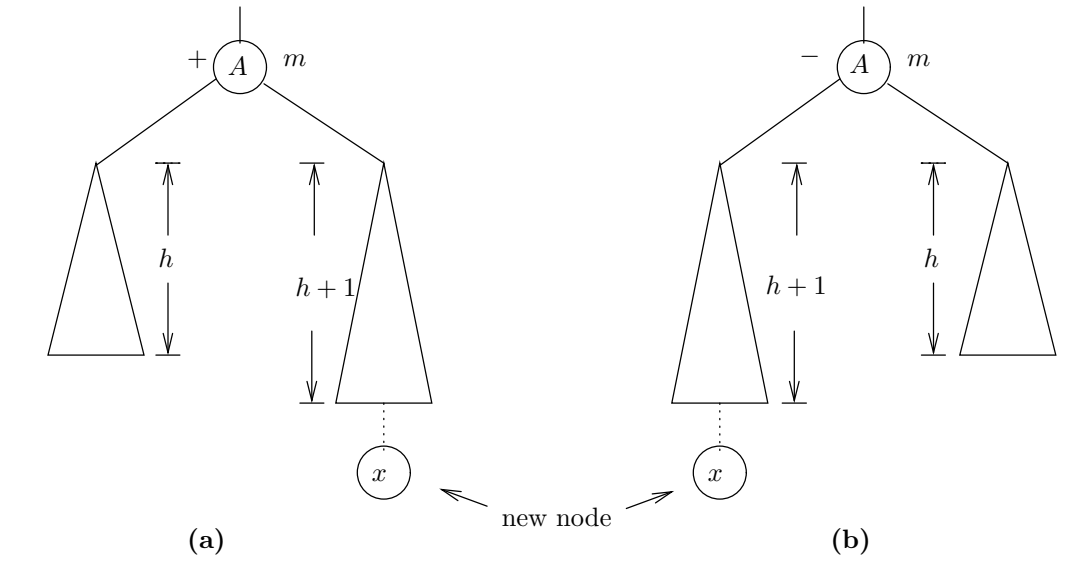


Figure 1

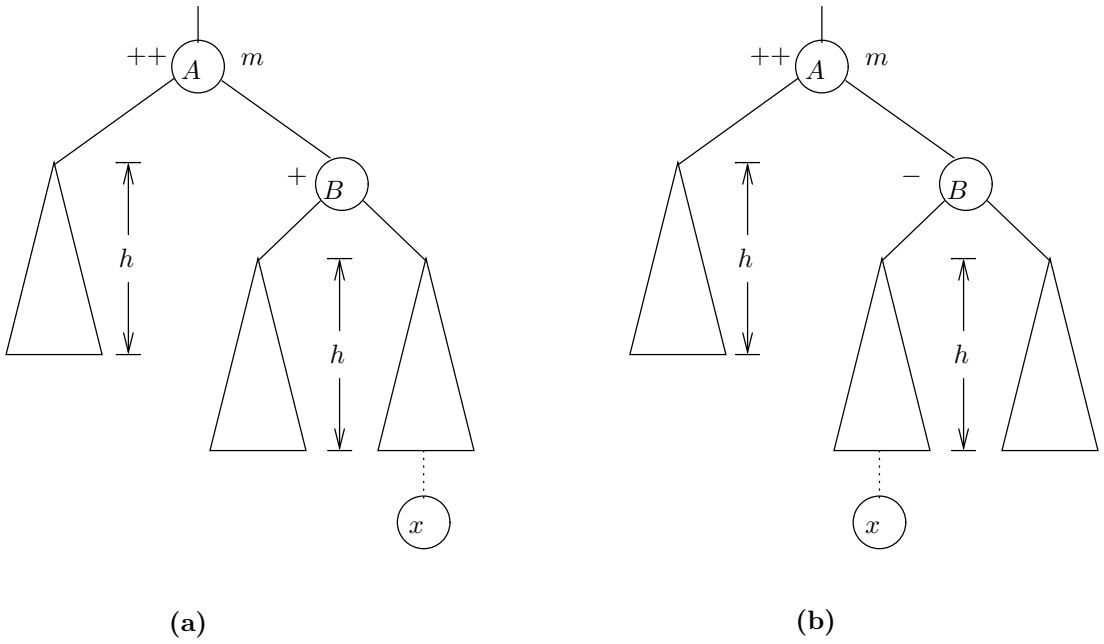
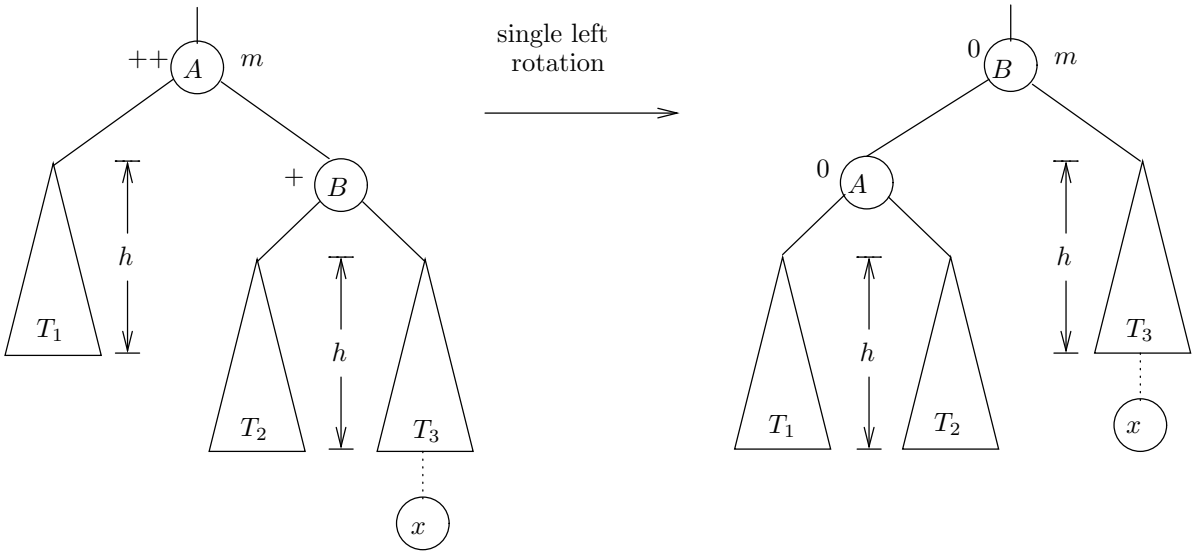


Figure 2

Unfortunately the subtree in Figure 2(b) cannot be rebalanced by a single left rotation. You should check that the subtree resulting from such a transformation is not height-balanced.

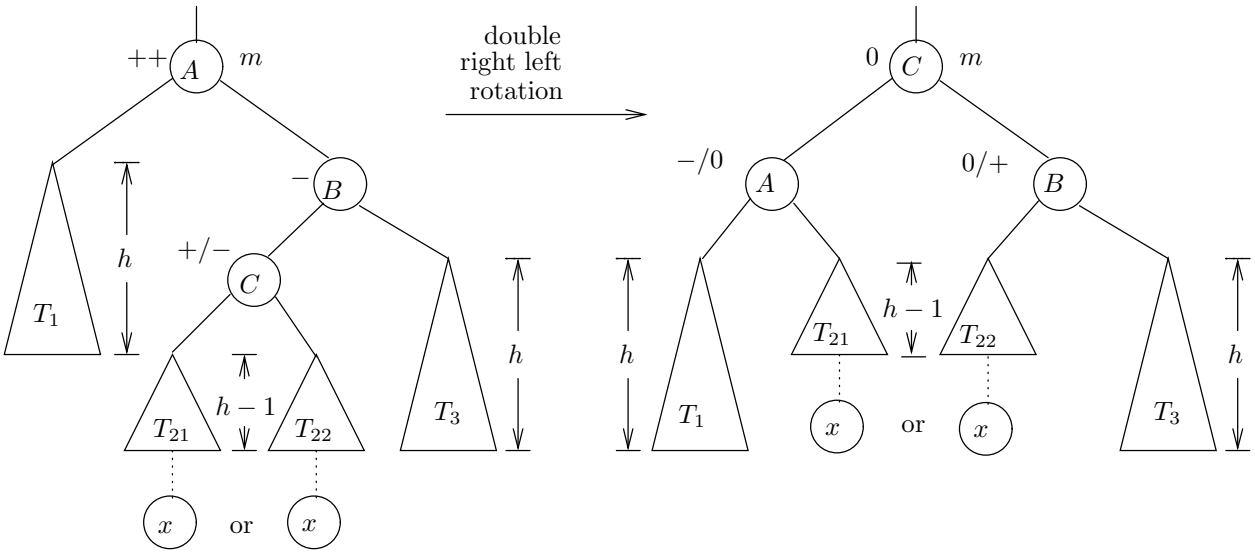
Assume for now that the subtrees of node B in Figure 2(b) are nonempty (i.e., $h \neq -1$). Figure 4(a) shows these subtrees in more detail. This more detailed picture leads to a different way of transforming the subtree into a height-balanced one. This transformation is called a "double right left rotation" and is illustrated in Figure 4(b). The name comes from the fact that this transformation can be obtained if we rotate first B to the right and then, in the resulting subtree, rotate C left. The balance factors labeled as "$*/*$" in Figure 4 depend on whether the new node was actually inserted under T_{22} (the first entry of the label) or under T_{21} (the second entry of the label).

If the subtrees of node B in Figure 2(b) are empty (i.e., $h = -1$) then A has only a right child, B, and B has only a left child, the new node x. The subtree rooted at A is not height-balanced and the situation can be rectified again with a double right-left rotation that makes x the root of the subtree, and A and B its



(a) **(b)**

Figure 3



(a) **(b)**

Figure 4

left and right children, respectively. This case can also be thought of as a degenerate instance of Figures 4(a) and (b), with $C = x$, and subtrees T_1, T_{21}, T_{22} and T_3 all empty.

The double right left rotation has the following properties.

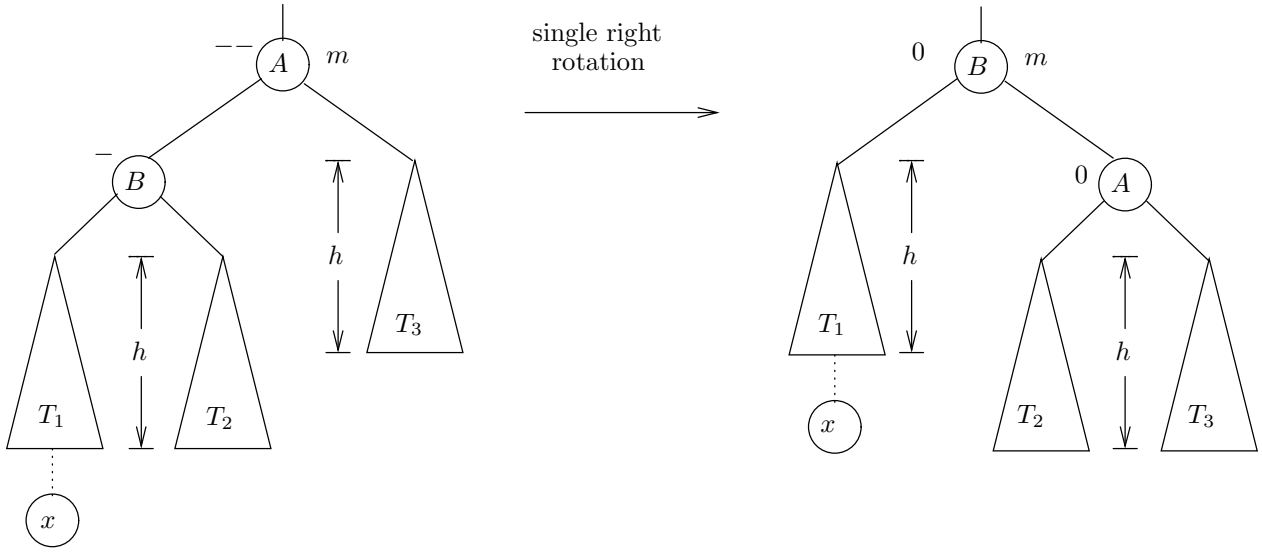
D.1 It rebalances the subtree rooted at m, (so that subtree becomes height-balanced again).

D.2 It maintains the binary search tree property.

D.3 It can be done in constant time: we only have to change a few pointers. As an exercise, write a program that implements this rotation, given a pointer to node m.

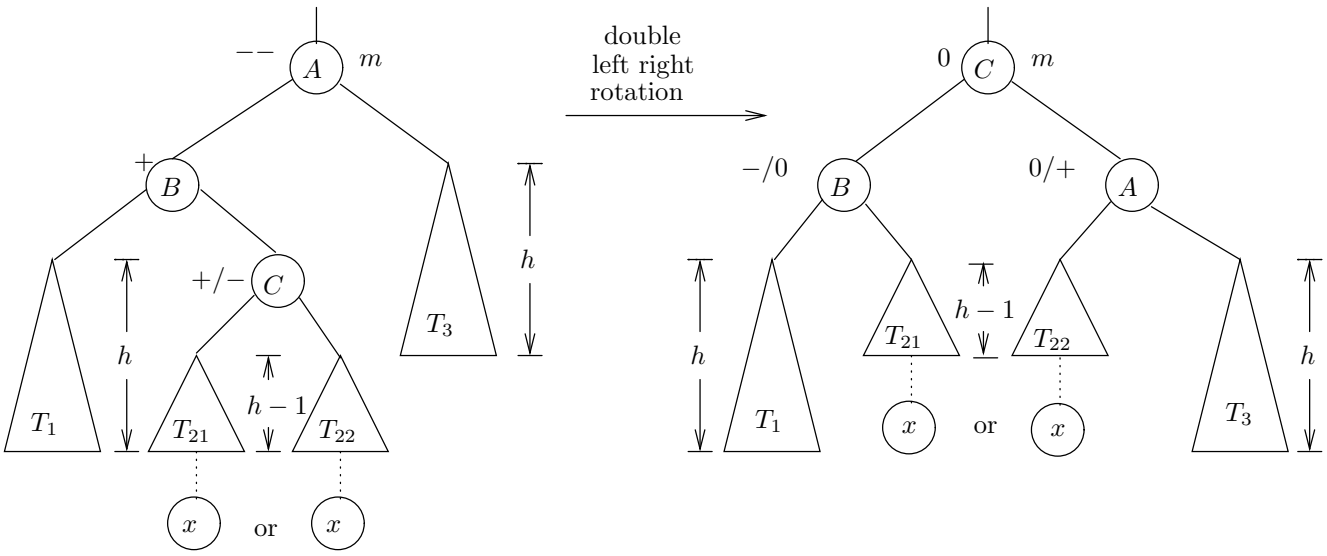
D.4 It keeps the height of m equal to that node's height *before* the insertion of the new node, namely, height $h + 2$.

As we already remarked, the imbalance shown in Figure 1(b) can be fixed in a symmetric way. The two subcases, and the transformations that rebalance the subtrees, called "single right rotation" and "double left right rotation", are illustrated in Figures 5 and 6 respectively. Remarks analogous to S.1–S.4 and D.1–D.4 apply in these transformations as well.



(a) **(b)**

Figure 5



(a) **(b)**

Figure 6

Updating the Balance Factors after Insertion

The balance factors of some nodes may change as a result of inserting a new node. First of all, observe that only the balance factors of the new node's ancestors may need updating: For any other node i, i's left and right subtrees (and, in particular, their heights) have not changed and thus neither has the balance factor of i. But not *all* of the new node's ancestors' balance factors may need updating. Figure 7 illustrates the issue. Insertion of key 8 to the AVL tree in 7(a) results in the AVL tree in 7(b). Note that only the balance of 9, 8's parent, has changed. On the other hand, insertion of key 8 to the AVL tree in 7(c) results in the AVL tree in 7(d), where the balance factors of all of 9's ancestors have changed.

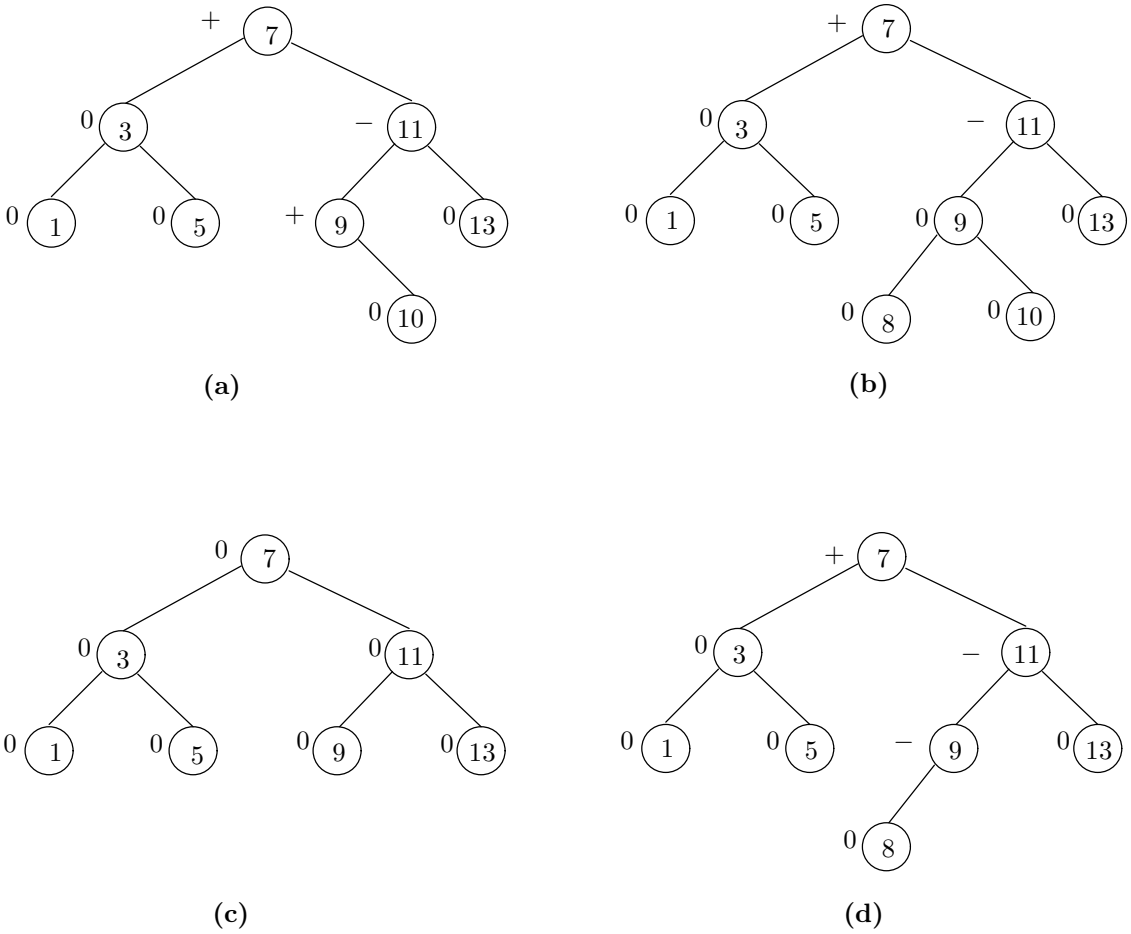


Figure 7

In general, let n be the new node just inserted into the tree and let p be n's parent. Further, let m be the closest ancestor of p that was *not* balanced (that is, that had balance factor $\pm$ 1) before the insertion of n; if no such ancestor of p exists, let m be the root of the tree. (Note that m could be p, if $BF[p] \neq 0$ before the insertion.)

Claim. *Only the balance factors of the nodes between p and m (included) need to be changed as a result of the insertion of n.*

Justification: Consider the (0 or more) nodes that are ancestors of p and proper descendents of m. By choice of m, all these nodes were balanced before the insertion of n. Thus their two subtrees had the same height and the insertion of n has increased the height of one of the subtrees; hence for each such node, its balance factor must be set to -1 or $+1$, depending on whether n was inserted to the left or right subtree,

respectively. Next consider node m. If m is the root and was balanced before the insertion, similar remarks as above apply to m: in this case the insertion of n has the effect of increasing the height of the entire tree. If m was *not* balanced before the insertion, we have two possibilities:

- If m was left heavy and n was inserted to m's right subtree, or if m was right heavy and n was inserted to m's left subtree, the subtree rooted at m becomes balanced as a result of the insertion (so we must set $BF[m] = 0$), but its height does not change. Therefore, neither do the heights of m's ancestors' subtrees; so the balance factors of m's proper ancestors do not change, and we can stop the process of balance factor updating here.
- If, on the other hand, m was right heavy and n was inserted to m's right subtree, or if m was left heavy and n was inserted to m's left subtree, the subtree rooted at m becomes unbalanced (these are the two cases illustrated in Figures 1(a) and 1(b) respectively). We can rebalance the subtree as we discussed previously (by the appropriate type of rotation). After the rebalancing, however, the subtree rooted at m will have the same height as it did before the insertion of n (recall Remarks S.4 and D.4). Thus, as argued before, the balance factors of m's ancestors do not change. Note, however, that when we rotate, the balance factors of the rotated nodes need updating, so we must do that before stopping.† □

The discussion on rebalancing and updating the balance factors after an insertion leads us to the following outline for the AVL tree insertion algorithm.

INSERT(x, T)

1. Trace a path from the root down, as in binary search trees, and insert x into a new leaf at the end of that path (the new leaf must be in the proper position, so as to maintain the binary search tree property).
2. Set the balance factor of the new leaf to 0. Retrace the path from the leaf up towards the root and process each node i encountered as follows:
   (a) If the new node was inserted in i's right subtree, then increase $BF[i]$ by 1 (because i's right subtree got taller); otherwise, decrease $BF[i]$ by 1 (because i's left subtree got taller).
   (b) If $BF[i] = 0$ (so the subtree rooted at i became balanced as a result of the insertion, and its height did not change) then stop.
   (c) If $BF[i] = +2$ and $BF[rchild(i)] = +1$ then do a single left rotation on i, adjust the balance factors of the rotated nodes (A and B in Figure 3(b)), and stop.
   (d) If $BF[i] = +2$ and $BF[rchild(i)] = -1$ then do a double right left rotation on i, adjust the balance factors of the rotated nodes (A, B and C in Figure 4(b)), and stop.
   (e) If $BF[i] = -2$ and $BF[lchild(i)] = -1$ then do a single right rotation on i, adjust the balance factors of the rotated nodes (A and B in Figure 5(b)), and stop.
   (f) If $BF[i] = -2$ and $BF[lchild(i)] = +1$ then do a double left right rotation on i, adjust the balance factors of the rotated nodes (A, B and C in Figure 6(b)), and stop.
   (g) If $i = root$ then stop.

† After a rotation, some of the rotated nodes are no longer ancestors of the inserted node; however, they may still need to have their balance factors updated.

THE ALGORITHM FOR DELETE

To delete a key x from an AVL tree T, we first locate the node n where x is stored. (This can be done by using the algorithm for SEARCH.) If no such node exists, we're done (there's nothing to delete). Otherwise we have three cases (as with ordinary binary search trees).

(1) *n is a leaf:* Then we simply remove it. This may cause the tree to cease being height-balanced. So we may need to rebalance it. We also have to update the balance factors of some nodes. These issues will be dealt with shortly.

(2) *n is a node with only one child:* Let n' be n's only child. Note that n' must be a leaf; otherwise the subtree rooted at n would not have been height-balanced before the deletion. In this case we copy the key stored at n' into n and we remove n' as in case (1) (since, as we just argued, it must be a leaf).

(3) *n has two children:* Then we find the smallest key in n's right subtree which, by the binary search tree property, is the smallest key in T larger than the key stored in n. To find this key, we go to n's right child (which exists), and we follow the longest chain of left child pointers until we get to a node n' that has no left child. We copy the key stored in n' into n and remove n' from the tree, as in (1), if n' does not have a right child either, or as in (2), if n' has only a right child.

To complete the algorithm we must discuss the conditions under which rebalancing is required and how the rebalancing can be performed. Since cases (2) and (3) ultimately reduce to deleting a leaf, case (1) is the only one we need to consider.

Rebalancing an AVL Tree after Deleting a Leaf

The deletion of a leaf n will cause the tree to become unbalanced in one of two cases:

(a) It reduces the height of the right subtree of a left heavy node; or,

(b) It reduces the height of the left subtree of a right heavy node.

These two cases, illustrated in Figure 8, are symmetric (as are the analogous cases in insertion), so we will only consider the first. As an exercise, you should treat the other.

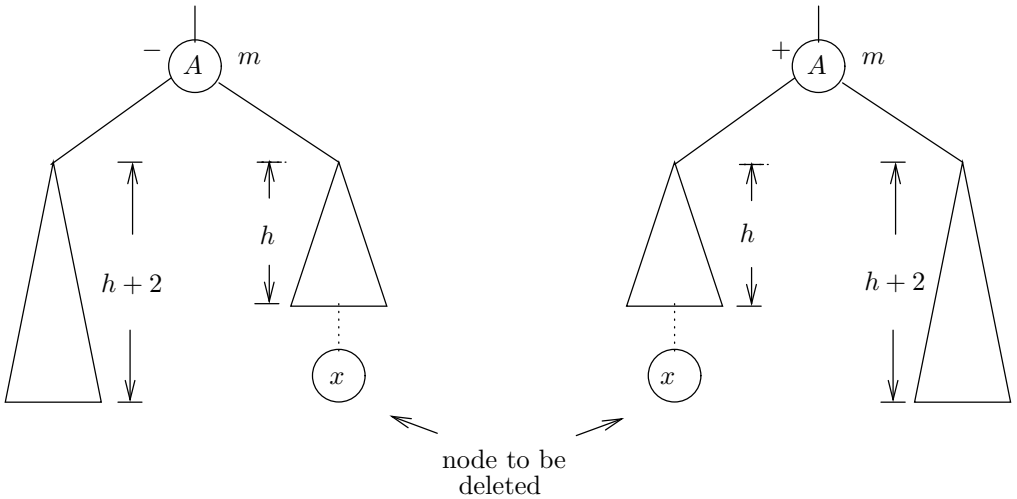


Figure 8

We consider case (a). As with insertion there are two ways this case could arise, shown in Figures 9(a) and 10(a). The subtree in 9(a) can be rebalanced by means of a single right rotation, and the result of this transformation in shown in 9(b). The "0/−" next to B in 9(a) means that this case will arise if the balance

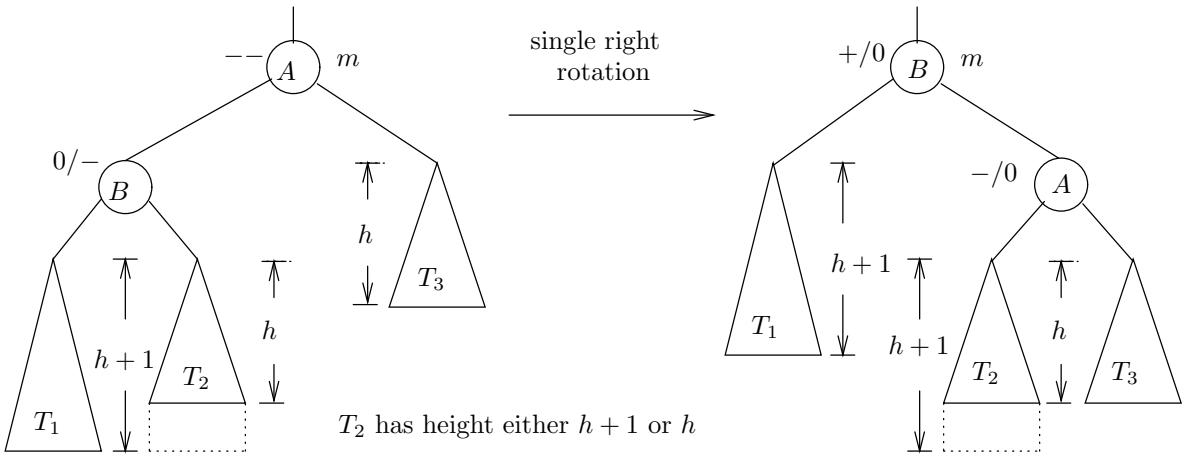


Figure 9

factor of B is 0 or -1 (that is, the height of T_2 is $h+1$ or h). Accordingly, the balance factors of B and A will be $+1$ or 0, and -1 or 0, after the rotation (see 9(b)).

The unbalanced subtree of Figure 10(a) can be rebalanced by means of a double left right rotation, the result of which is shown in 10(b). The balance factors of the nodes that have a label of the form "$*/*/*$" in Figure 10 depend on the heights of T_{21} and T_{22}. Note that *at least one* of these subtrees must have height h (the other could have height $h-1$ or h). The first entry of the label indicates the balance factor in the event T_{21} has height $h-1$ and T_{22} has height h; the second entry of the label indicates the balance factor in the event both trees have height h; and the third entry indicates the balance factor when T_{21} has height h and T_{22} has height $h-1$.

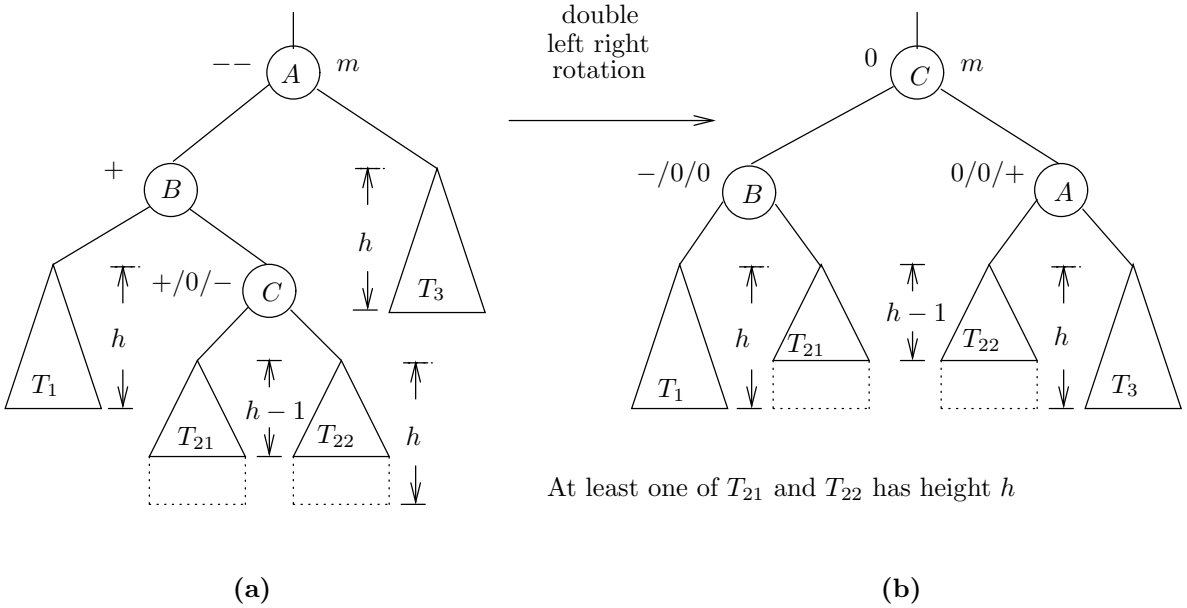


Figure 10

The above two transformations have the following properties.

1. They rebalance the subtree rooted at m (so the subtree becomes height-balanced again).

2. They maintain the binary search tree property.
3. They can be done in constant time by simply manipulating a few pointers. As an exercise, write programs that implement the rotations of Figures 9 and 10, given a pointer to m.
4. *They may decrease the height of the subtree rooted at* m, compared to the height of the subtree before the deletion.

Compare 4 with remarks S.4 and D.4 about rotations to restore balance in insertions. The difference is important: In the insertion algorithm just one rotation always rebalances the subtree, and, by maintaining the height of that subtree, it rebalances the entire tree. In the deletion algorithm the rotation balances the subtree, but since the height is decreased, the balance factor of nodes higher up (closer to the root) may change as a result — so we may have to go on rotating subtrees all the way up to the root in order to rebalance the entire tree. Thus in deletion we may have to do as many as $O(\log n)$ rotations. (That's acceptable though, because each one takes only constant time! We will say more about the complexity of operations shortly.)

Updating the Balance Factors after Deleting a Leaf

We must also address the question of how the deletion of a leaf affects the balance factors of its ancestors (clearly, it doesn't affect the balance factors of other nodes).

Let n be the deleted leaf and let p be its parent. We trace the path from p back to the root and we process each node i we encounter on the way as follows:

- If i was balanced before the deletion (so $BF[i] = 0$) then the left and right subtrees of i had the same height. The removal of n shortened one of them (so i's balance factor must be updated), but the height of the subtree rooted at i after the deletion remains the same as before it. This means that the deletion of n does not affect the balance factors of i's proper ancestors. So, in this case, all we have to do is increase $BF[i]$ by one if n was in i's left subtree (because then the deletion made the right subtree of i taller than the left), or decrease $BF[i]$ by one if n was in i's right subtree (because then the deletion made the left subtree of i taller than the right). After this, we can stop the process of updating balance factors.
- If i was right or left heavy before the deletion ($BF[i] = \pm 1$), we again update $BF[i]$ as above. If this balances node i, the deletion of n shortened one of the two subtrees of i, so we go up the path to consider the next node. Otherwise, the increase or decrease of $BF[i]$ by one causes the subtree rooted at i to become (height) unbalanced ($BF[i]$ becomes ± 2). In this case we need to rebalance the subtree by the appropriate rotation, as discussed previously. If the rotation causes the height of i to decrease compared to its height before the delection (see Remark 4 above), the process of updating balance factors and, possibly, rotating, must continue with i's parent. Otherwise, the rotation leaves the height of i the same as it was before the deletion, and therefore the process stops at i.
- Finally, if the process propagated all the way to the root (i = root) we can stop.

From this discussion you should be able to distill the outline of an algorithm for AVL tree deletion.

WORST CASE TIME COMPLEXITY FOR Search, Insert, Delete

Theorem. (Adelson-Velski and Landis) *The height of an AVL tree with* n *nodes is at most* $1.44 \log_2(n+2)$.

Proof: Let T_h be a height-balanced tree of height h with the *minimum possible number of nodes*, and let n_h be that number of nodes. Since T_h is height-balanced, one of its subtrees must have height $h-1$ and the other height $h-1$ or $h-2$. Since we want T_h to have the minimum number of nodes, we may assume that one of its subtrees is T_{h-1} and the other is T_{h-2}. Thus, the number of nodes in T_h is equal to the number of nodes in T_{h-1} plus the number of nodes in T_{h-2} plus one (for the root); that is,

$$n_h = n_{h-1} + n_{h-2} + 1.$$

Thus $n_0 = 1$, $n_1 = 2$, $n_2 = 4$, $n_3 = 7$, $n_4 = 12$, and so on. Comparing this with the sequence of Fibonacci numbers we see that, in general, $n_h = F_{h+3} - 1$ (where F_h is the h^{th} Fibonacci number).† From the theory of Fibonacci numbers we know that $F_h > (\phi^h/\sqrt{5}) - 1$, where $\phi = (1 + \sqrt{5})/2$‡ (if interested in this and other results on Fibonacci numbers, see Knuth, *The Art of Computer Programming*, Vol. 1 (Fundamental Algorithms), pp. 78–83.)

Thus for the number n of nodes in any AVL tree of height h we must have:

$$n \geq n_h = F_{h+3} - 1 > \left(\frac{\phi^{h+3}}{\sqrt{5}}\right) - 2.$$

Therefore,

$$h < \log_\phi((n+2)\sqrt{5}) - 3,$$

so

$$h < \left(\frac{1}{\log_2 \phi}\right) \cdot (\log_2 \sqrt{5} + \log_2(n+2)) - 3,$$

from which the theorem follows by arithmetic. □

In the worst case, the algorithms for SEARCH, INSERT, and DELETE have to process all nodes in a path from the root to a leaf. The above theorem says that this path must involve at most $O(\log n)$ nodes. Processing a node (be it just comparing the key stored in it to a key we are searching for, updating the balance factor, or performing a rotation on that node) takes constant time. Thus all these algorithms take $O(\log n)$ time in the worst case.

† The i^{th} Fibonacci number is defined inductively as follows: $F_1 = F_2 = 1$, and for $i > 2$, $F_i = F_{i-1} + F_{i-2}$.

‡ ϕ is known as the "golden ratio".