Lecture Notes on
Data Structures

M1522.000900

Seoul National University

Fall 2022

Part III

Trees

Binary Trees

Definition 1

A binary tree is a structure defined on a finite set of nodes such that

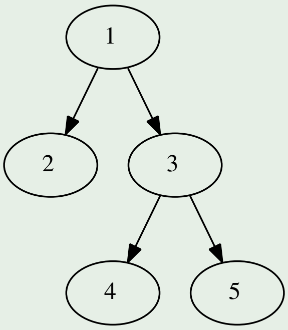
1. either the node set (or the tree) is empty,
2. or there are three disjoint sets of nodes:
    1. a root node,
    2. a binary tree called left subtree, and
    3. a binary tree called right subtree.

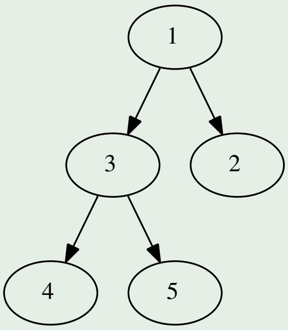
For given binary trees T_1 and T_2, $T_1 = T_2$ if and only if

$$\begin{aligned} root(T_1) &= root(T_2), \\ LeftSubtree(T_1) &= LeftSubtree(T_2), \\ RightSubtree(T_1) &= RightSubtree(T_2). \end{aligned}$$

Example 1

Are these two binary trees identical or different?





Binary Tree ADT

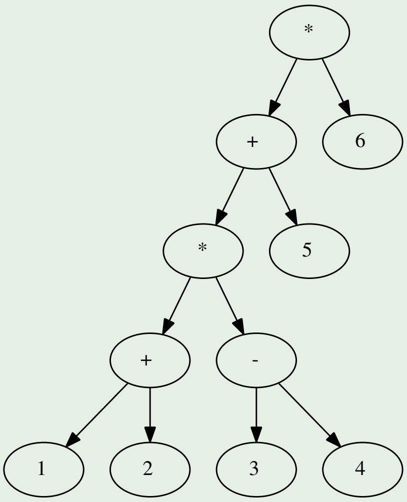
- Data: a finite set of zero or more nodes.
- Operations:
  1. $CreateBT : () \to BT$, (Create an empty binary tree.)
  2. $IsEmpty : BT \to Boolean$,
  3. $LeftSubtree : BT \to BT$,
  4. $RightSubtree : BT \to BT$,
  5. $RootKey : BT \to \{x, EMPTY\}$, (Return EMPTY if BT is empty. Otherwise, return the key in the root node.)
  6. . . .

Expression trees

An expression tree represents an arithmetic expression composed of binary operators. Internal nodes store operators, while leaf nodes store operands.

Example 2

What arithmetic expression does this tree represent?



Tree Traversal

Tree traversal is a process for visiting all the nodes in a tree in some order. There are three well-known tree traversal techniques:

1. Postorder traversal: visit left subtree, then right subtree, then root node.
2. Preorder traversal: visit root node, then left subtree, then right subtree.
3. Inorder traversal: visit left subtree, then root node, then right subtree.

Example 3

Given an expression tree, print its postfix expression.

```
Postorder(T):
   if (T == null) return;
   Postorder(T.left);
   Postorder(T.right);
   print T.key;
```

Example 4

Given an expression tree, print its infix expression.

Sol 1:

```
Inorder(T):
   if (T == null) return;
   Inorder(T.left);
   print T.key;
   Inorder(T.right);
```

Is this correct?

The first solution is not correct, because some necessary parentheses are missing.

Sol 2:

```
Inorder(T):
   if (T == null) return;
   print '(';
   Inorder(T.left);
   print T.key;
   Inorder(T.right);
   print ')';
```

Is this correct?

The second solution produces correct answers, but with too many redundant parentheses. A better solution will be as follows:

Sol 3:

```
Inorder(T):
   if (T == null) return;
   if (T.key == op) print ’(’;
   Inorder(T.left);
   print T.key;
   Inorder(T.right);
   if (T.key == op) print ’)’;
```

Problem 1

When evaluating a postfix expression using a stack, how large can the stack grow?

Lemma 1

For a given expression tree T, let $f(T)$ denote the max size the stack can grow to. Then,

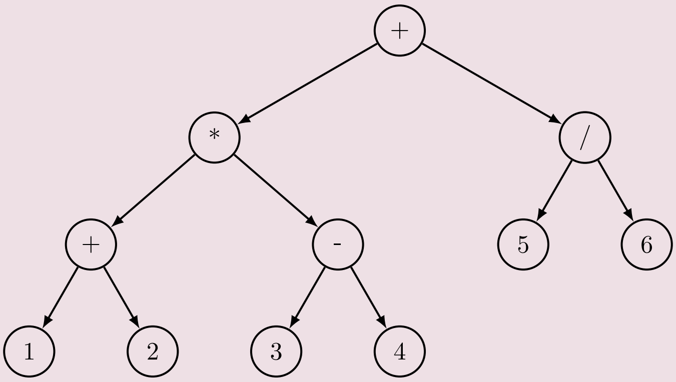
$$f(T) = \begin{cases} 0 & \text{if } T \text{ is empty,} \\ \max\{f(T.Left), 1 + f(T.Right)\} & \text{otherwise.} \end{cases}$$

We can make use of a few observations.

- The root node is an operator and must have two subtrees.
- Both subtrees are evaluated ahead of the root. (See the next slide.)
- $f(T) \geq f(T.Left)$ and $f(T) \geq f(T.Right)$.
- The stack will grow or shrink while evaluating $T.Left$, but it will eventually be reduced to a single operand just before start evaluating $T.Right$.

Postfix expression vs. Expression tree

An arithmetic (infix) expression $(((1+2)*(3-4))+(5/6))$ can be represented by an expression tree below.



The equivalent postfix expression is

$$\underbrace{1\ 2 + 3\ 4\ \text{-}\ *}_{Left\ \ Subtree}\quad \underbrace{5\ 6\ /}_{Right\ \ Subtree}\quad \underbrace{+}_{Root}.$$

Theorem 2

$f(T) \leq depth(T)$. In other words, $f(T) \in \mathcal{O}(depth(T))$.

Proof. Prove it by induction on the depth of T.

Let T_k denote an expression tree of depth k.

- Base: For T_0 (*i.e.*, an empty tree), LHS $= 0$ and RHS $= 0$.
- Induction: Assume that $f(T_k) \leq depth(T_k)$ for $k \geq 0$. From Lemma 1, $f(T_{k+1}) = f(T_{k+1}.Left)$ or $f(T_{k+1}) = 1 + f(T_{k+1}.Right)$. Recall that $depth(T_{k+1}.Left) \leq k$ and $depth(T_{k+1}.Right) \leq k$.
  1. If $f(T_{k+1}) = f(T_{k+1}.Left)$ then
  $$f(T_{k+1}) = f(T_{k+1}.Left) \leq depth(T_{k+1}.Left) \leq k.$$
  2. If $f(T_{k+1}) = 1 + f(T_{k+1}.Right)$ then
  $$f(T_{k+1}) = 1 + f(T_{k+1}.Right) \leq 1 + depth(T_{k+1}.Right) \leq 1 + k.$$

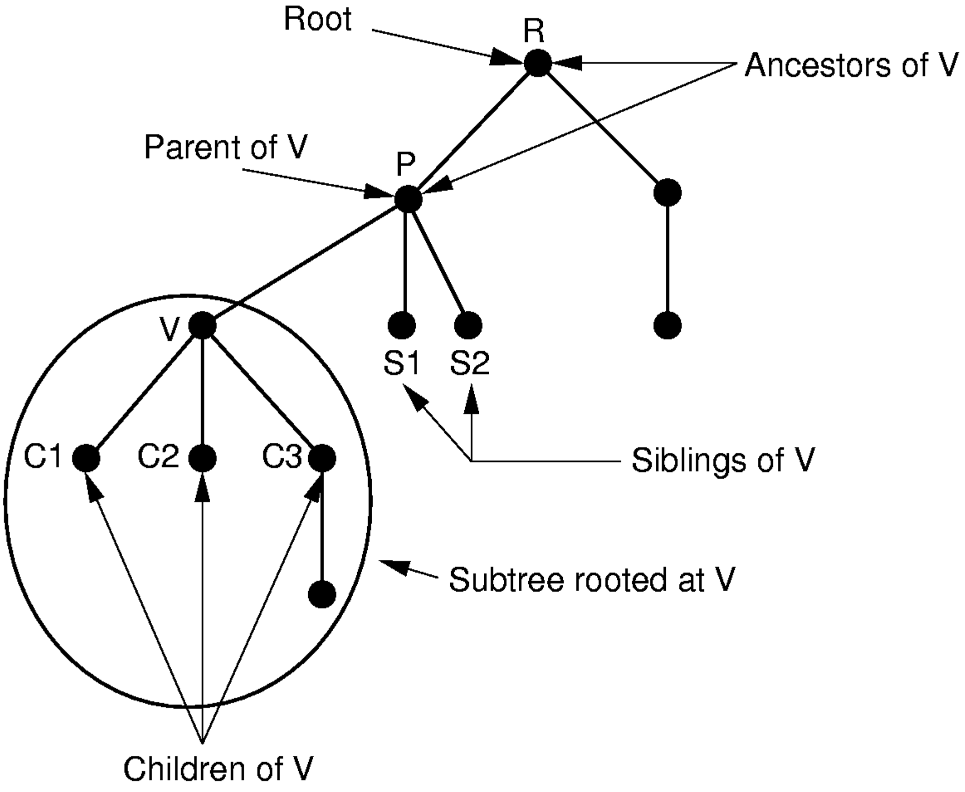
  Therefore, $f(T_{k+1}) \leq k + 1 = depth(T_{k+1})$.

□

Problem 2

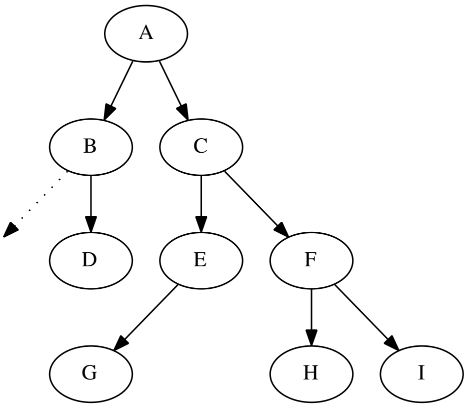
Given a fully-parenthesized infix expression, generate an expression tree that is equivalent to the expression. (HINT: Use stack(s).)

General trees: terms



Binary trees: More terms & definitions

- root node, leaf node, internal node,
- level of a node is the distance from the root. ($level(root) = 0$),
- the height (or depth) of a tree $T = 1 + max_{x \in T}\{level(x)\}$,
- the number of nodes at level $d \leq 2^d$ (proof by induction),
- full binary tree (*e.g.*, Huffman coding tree),
- complete binary tree (*e.g.*, Heap).



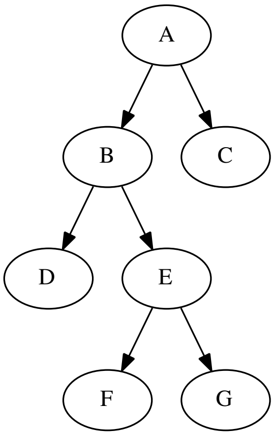
Definition 2 (Full binary tree)

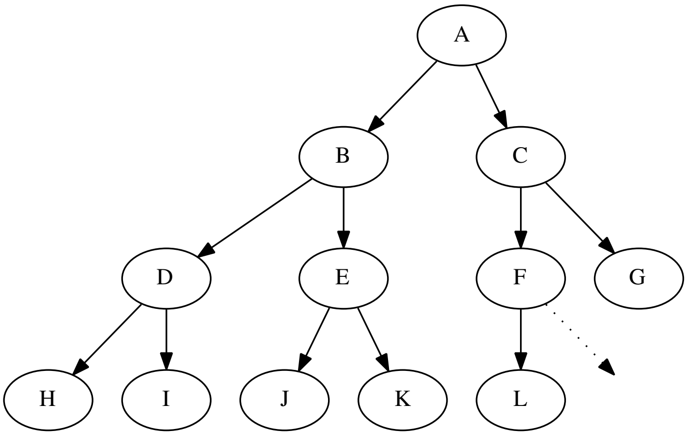
Every node has either zero or two child nodes.

Definition 3 (Complete binary tree)

A binary tree of height d is complete if

1. every level (from 0 to $d-2$) is fully occupied, and
2. the bottom level ($d-1$) is filled from left to right with no gap.





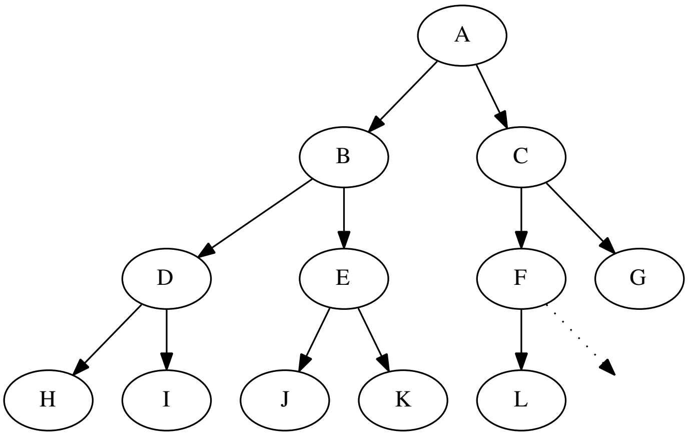
Binary Tree: Array Implementation

A complete binary tree can be stored in an array conveniently.

- Tree nodes are assigned to the array (from the first array element up to the last one) starting from top to bottom and from left to right of the tree.
- Since there is no missing node (or gap) in this particular traversal of a complete binary tree, there is no empty slot in the array either.
- There exists a bijective (*i.e.*, one-to-one and onto) mapping between the tree nodes and the array elements.
- The number of nodes is equal to the number of array elements.

Example 5

Show an array that stores a complete binary tree shown below.



Navigating a complete binary tree

Suppose an array `A[0:N-1]` represents a complete binary tree of N nodes. Then, for a node stored in `A[i]`,

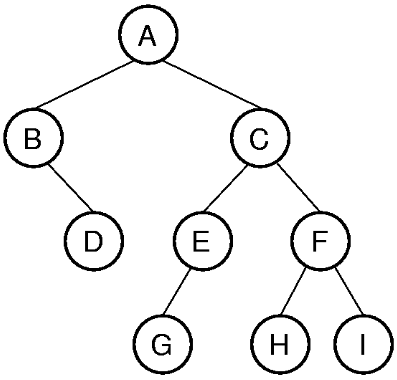
$$\begin{aligned} Parent(A[i]) &= A[(i-1)/2] \\ LeftChild(A[i]) &= A[2i+1] \\ RightChild(A[i]) &= A[2i+2] \end{aligned}$$

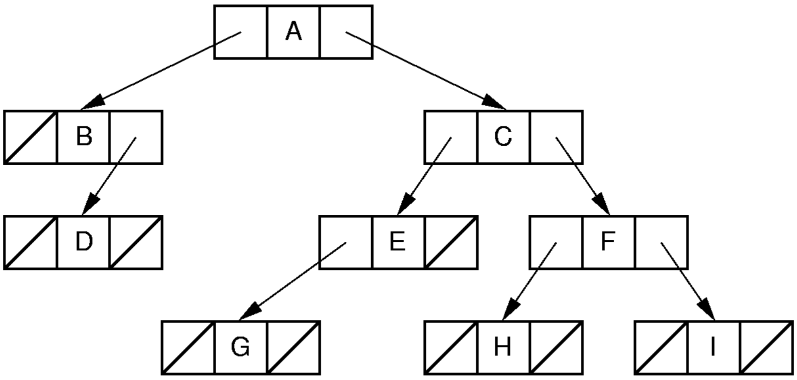
These formulas will be convenient for writing an efficient code for tree traversal. However, the array implementation works only for complete binary trees.

Binary Tree: Pointer-based Implementation

Node structure:

LeftChild	Key	RightChild





Theorem 3 (Full Binary Tree Theorem)

For a non-empty full binary tree T, the number of leaf nodes in T is $1+$ the number of internal nodes in T.

Proof. Proof by induction (on the number of internal nodes).

- Base: For a full BT with no internal node (*i.e.*, a single node tree), LHS $= 1$ and RHS $= 1 + 0$.
- Induction: Assume the statement is true for a full BT T_k with k internal nodes. We need to show it is also true for T_{k+1}.
  Choose an internal node of T_{k+1} that has two leaf nodes as its children. (*i.e.*, The chosen node is at level $height(T_{k+1}) - 2$.)
  Remove the two leaf nodes from T_{k+1} and call the remaining tree T'. Then, T' is still a full BT but with one less, or k, internal nodes. This means that T' has $k+1$ leaf nodes by the assumption. Compare T_{k+1} with T'. How many more internal and leaf nodes does T_{k+1} have than T'?

□

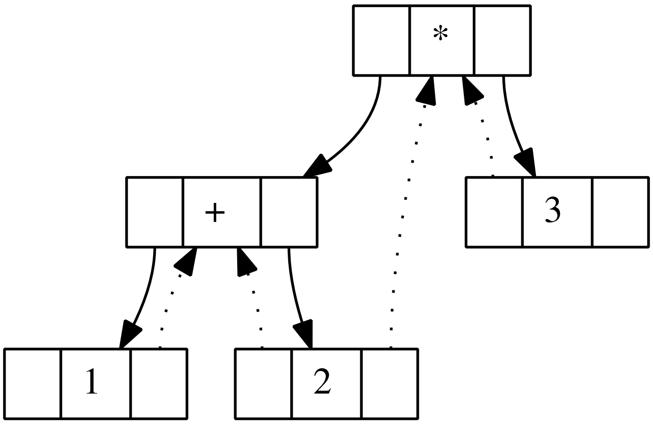
Theorem 4

For any binary tree of n nodes ($n \geq 1$) implemented by a linked structure, the number of null pointers is $n+1$.

Proof. There exist a total of $2n$ pointers in this binary tree. How many of them are in use? □

To improve storage utilization,

1. Use different node structures for different type of data. For example, use two different node structures for operators (with two pointer fields) and operands (without them).
2. Make use of the $n + 1$ null pointers: *Threaded Binary Tree*.



Threaded Binary Trees

To make T a threaded binary tree, for each node $t \in T$,

```
if (t.left == null)
   t.left = inorder-predecessor of t or null;
if (t.right == null)
   t.right = inorder-successor of t or null;
```

Use Boolean flags to distinguish threaded pointers from regular child pointers. The node structure is defined as:

Lthreaded	Left	Key	Right	Rthreaded

The condition `T.Lthreaded` is true indicates that `T.Left` is a threaded link, otherwise a regular child link.

The advantage of threaded binary trees is faster inorder traversal without recursions.

Example 6

Write an algorithm `Insucc(t)` that returns an inorder successor of `t`.

```
Insucc(t):
   if (t.Right == null) return null;                  // Case 1.
   if (t.Rthreaded == true) return t.Right;           // Case 2.
   tmp = t.Right;                                     // Case 3.
   while(tmp.Lthreaded == false) tmp = tmp.Left;
   return tmp;
```

Case 1 Trivial. There is no inorder-successor.

Case 2 `t.Right` is the inorder successor by definition.

Case 3 The node at the leftmost corner of the right subtree of `t` is the inorder successor of `t`.

Example 7

With `Insucc(t)` that returns an inorder successor of `t`, write an algorithm that performs inorder traversal of a threaded binary tree T.

```
Inorder(T):
   tmp = T; // root node
   while(tmp.Left != null) tmp = tmp.Left;
   while(tmp != null) {
      print tmp.Key;
      tmp = tmp.Insucc(tmp);
   }
```

Problem 3

Compare `Inorder(t)` given in Example 7 and its recursive version (for a regular binary tree) with respect to the time and space complexities.

Application: Data Compression

The ASCII code is:

- One of fixed-length coding schemes (8 bits per char).

$$\begin{aligned} A &\longrightarrow 01000001 \\ B &\longrightarrow 01000010 \\ C &\longrightarrow 01000011 \\ &\vdots \end{aligned}$$

- The size of a text with n characters is $8 \times n$ bits.

Data compression by a variable-length coding scheme:

- Assign shorter codes to frequent chars, and longer codes to infrequent chars, hoping that the size of a text will be reduced.
- The size of a text with n characters (with k distinct chars) is

$$n \times \sum_{i=1}^{k} l_i p_i$$

  where l_i is the code length of the i-th letter, and p_i is the probability of the i-th letter being equal to each individual character in the text.
- If f_i is the relative frequency of the i-th letter, then

$$p_i = f_i / (\sum_{j=1}^{k} f_j).$$

Required Property: Immediate decodability

Example 8

Is the following coding scheme unambiguous?

char	code
a	01
b	010
c	1
d	11

For example, can you decode an encoded bit string `01011` using the coding scheme?

No, `01011` can be `aac` or `bd`. The coding scheme is ambiguous, because it is not always possible to decode a bit string into a unique char string.

Definition 4

A bit string x is a prefix of a bit string y if there exists a non-null bit string w such that $xw = y$.

For a coding scheme to be unambiguous, no code is allowed to be a prefix of another code. Such a coding scheme is said to be *immediately decodable*.

Huffman Coding Tree

- A full binary tree each of whose leaf nodes corresponds to a character.
- Provides a variable-length coding scheme called Huffman Code.
- In the formula (given in Slide 31),

$$n \times \sum_{i=1}^{k} l_i p_i$$

  l_i is the level (or the path length) of a leaf node corresponding to the i-th letter, and the term $\sum_{i=1}^{k} l_i p_i$ is called *sum of weighted path lengths* of a Huffman coding tree.
- For a given set of characters and their relative frequencies, the Huffman coding tree is not always unique.

To construct a Huffman tree for *k* characters:

1. Generate *k* (root only) binary trees containing their frequencies; sort them in increasing order of the frequencies.
2. Remove the first two trees; build a new tree T such that

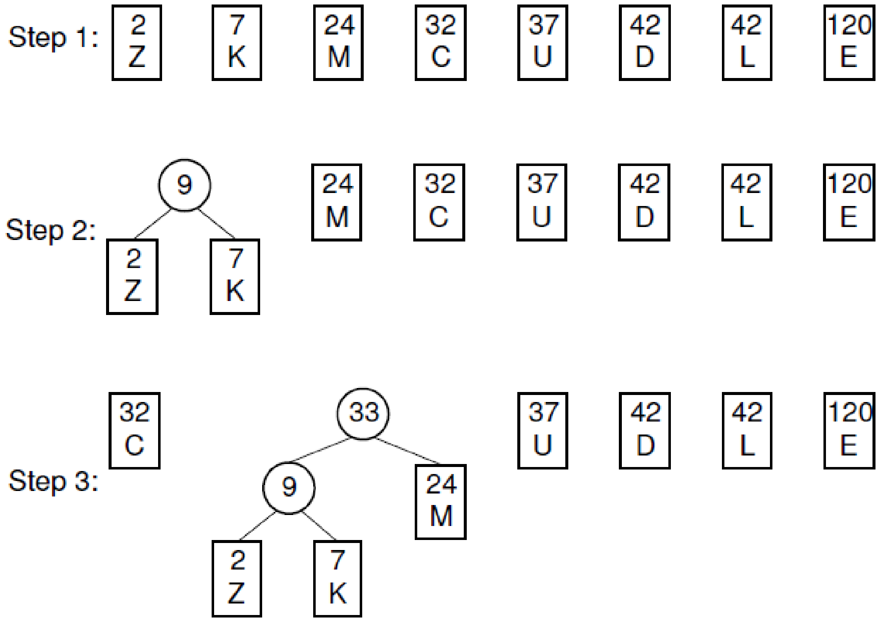
```
T.freq = T1.freq + T2.freq;
T.Left = T1;
T.Right = T2;
```

3. Add the new tree T to the list in a way that the frequency order is preserved.
4. Repeat until only one tree is left.

Example 9

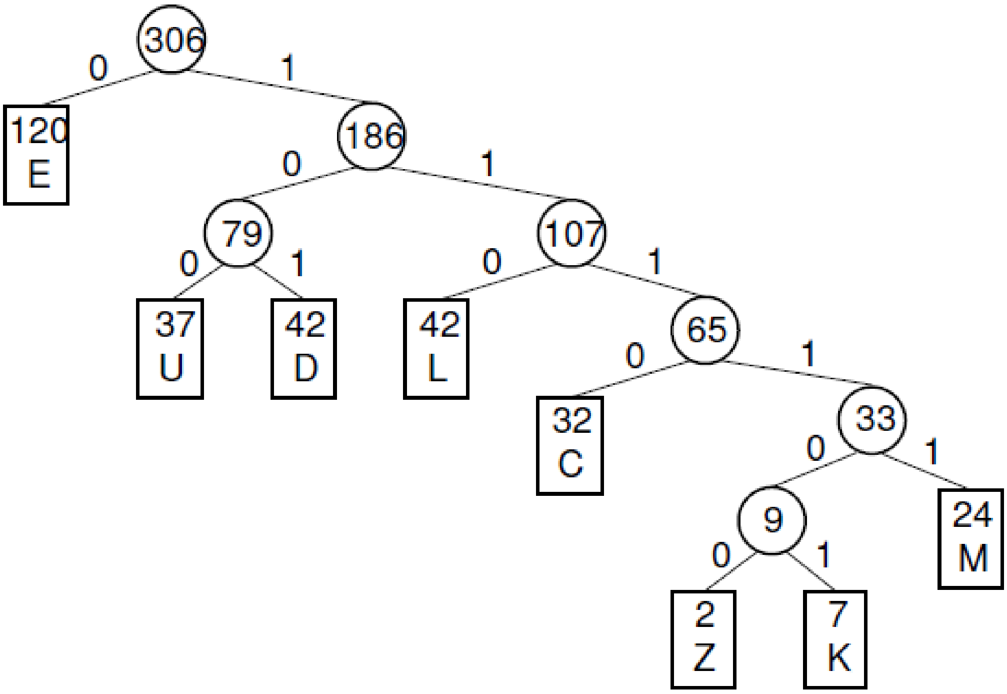
Build a Huffman Tree for the following characters and frequencies.

char	C	D	E	K	L	M	U	Z
freq	32	42	120	7	42	24	37	2



Then, to establish a coding scheme with a Huffman tree,

1. assign 0 to left links and 1 to right links,
2. concatenate all the 0/1 labels on the path from the root node to each leaf node.



Below is a coding table built from the Huffman tree.

char	frequency	code	code length
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

The sum of weighted path lengths is

$$\frac{(4\times 32)+(3\times 42)+(1\times 120)+(6\times 7)+(3\times 42)+(5\times 24)+(3\times 37)+(6\times 2)}{32+42+120+7+42+24+37+2}$$
$$\approx 2.57$$

Example 10

Is the Huffman code for Example 9 ambiguous? What is the expected size of a text with n characters encoded with the Huffman code?

char	frequency	code	code length
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

This coding scheme is not ambiguous because no code is a prefix of another. The expected size of a text with n characters is approximately $2.57 \times n$. In contrast, the size would be $3 \times n$ with a fixed-length coding scheme.

To decode a bit string, use the Huffman tree:

```
T = the root of a Huffman tree;
while (input is not empty) do
   read a bit x;
   if (x==0) then T = T.Left;
            else T = T.Right;
   if (T is a leaf) then { print T.char; T = root; }
end
```

Example 11

Decode a bit string 110001110100 using the Huffman code from Example 10.

The decoded character string is LEECU.

Problem 4

Build a Huffman Tree for the following characters and frequencies. Then create a Huffman coding table and compute the sum of weighted path lengths.

char	a	b	c	d	e	f	g
freq	7	3	2	9	15	2	1

Can you decode a bit string 11110110100 using the Huffman code?

Ambiguity

Problem 5

Does the Huffman code satisfy immediate decodability?

The answer is yes because

- Each code corresponds to a leaf node.
- Any prefix of a code corresponds to a non-leaf node, which cannot be a Huffman code.

Optimality

Problem 6

Is the Huffman code optimal in terms of compression ratio? In other words, does the Huffman code achieve better compression than any other compression algorithm that uses relative frequencies?

Sketch of proof:

1. Any unambiguous variable-length coding table (like the coding table in Example 10) can be transformed into a full binary tree (say T).
2. Compute the sum of external path weights for T.
3. Show by induction on the number of letters that the sum of weighted path lengths of T is always greater than or equal to that of a Huffman coding tree for a given set of letters and frequencies.

Problem 7

What is the running time of the Huffman Tree construction for k chars?

1. Create k trees and sort them ...
2. Remove the first two trees ...
3. Merge them ...
4. Put the merge tree back to the sorted list ...

Application: Search

Compare search techniques that we have discussed so far.

1. Unsorted list (either array or linked list)
    - search: $\mathcal{O}(n)$ by linear search
    - insert: $\mathcal{O}(1)$
2. Sorted list (array)
    - search: $\mathcal{O}(\log n)$ by binary search
    - insert: $\mathcal{O}(n)$ for shifting elements
3. Sorted list (linked list)
    - search: $\mathcal{O}(n)$
    - insert: $\mathcal{O}(n)$

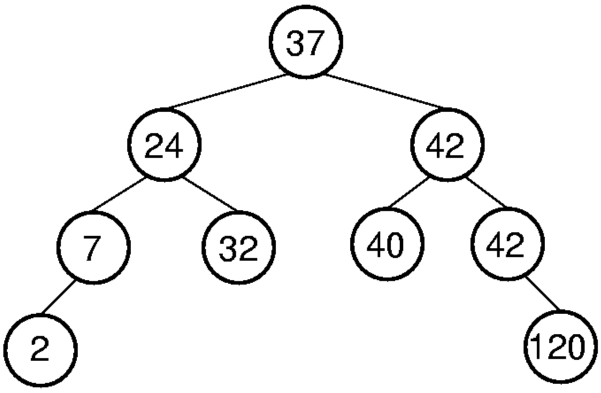
Can we do better than these? Is there any data structure or algorithm that takes $\mathcal{O}(\log n)$ running time for both search and insert?

Binary Search Tree (BST)

Definition 5

A BST is a binary tree that is either empty or that satisfies the following conditions:

1. key of any node in the left subtree $<$ key of the root node,
2. key of any node in the right subtree $\geq$ key of the root node,
3. both the subtrees are BST.



Binary Search Tree ADT

- Data: a finite set of zero or more nodes.
- Operations:
  1. $CreateBST : () \rightarrow BST$, (Create an empty binary search tree.)
  2. $Search : BST, x \rightarrow pointer$,
  3. $Insert : BST, x \rightarrow BST$,
  4. $Remove : BST, x \rightarrow BST$,
  5. $IsEmpty : BST \rightarrow Boolean$,
  6. $\cdots$

Example 12 (treesort)

Write a sort algorithm using a binary search tree.

1. Insert input elements into an initially empty BST.
2. Traverse the BST in inorder.

BST: Search

Algorithm 1

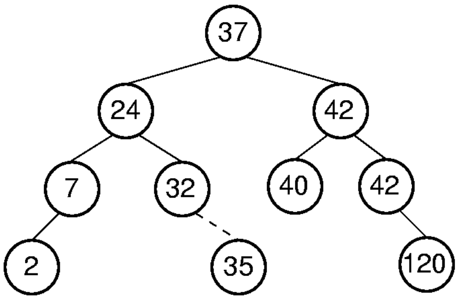
```
Search(T,x)
   if (T == null) return null;
   if (T.key == x) return T;
   if (T.key > x) return Search(T.Left,x);
   else return Search(T.Right,x);
```

The running time of the BST search operation is $\mathcal{O}(h)$, where h is the height of a BST. (Count the number of nodes to visit.)

BST: Insert

Algorithm 2

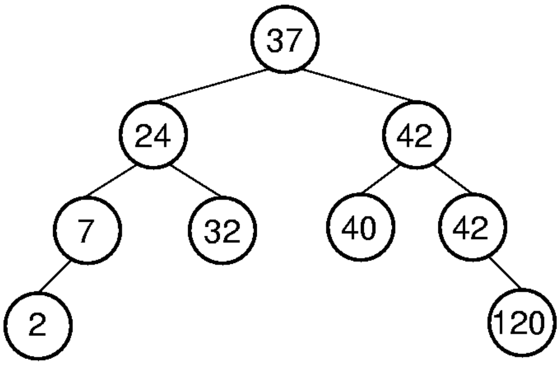
```
Insert(T,x)
    if (T == null) T = new Node(x);
    else if (x < T.key) T.left = Insert(T.Left,x);
    else T.right = Insert(T.Right,x);
    return T;
```

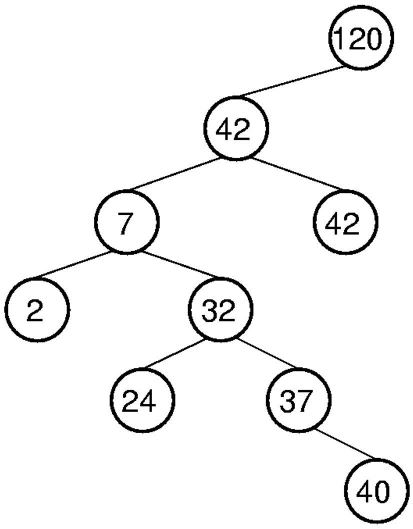


The running time complexity of the BST Insert is the same as that of the BST search.

Remark 1

- Even for the same set of key values, the structure (shape, height) of BST varies depending on the order of insertions.





Can you tell which order the keys were inserted in?

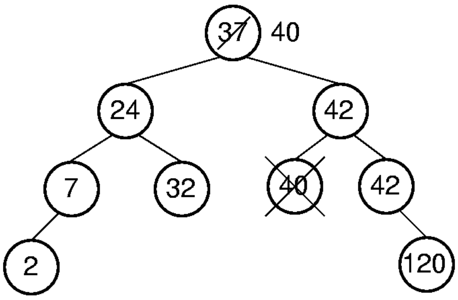
BST: Remove

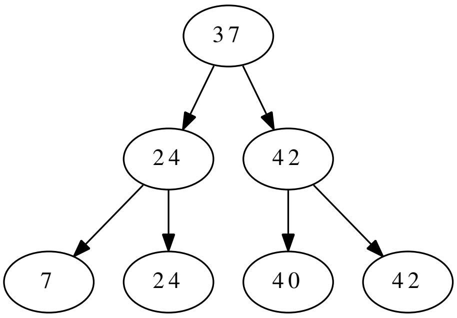
To remove a node p from a BST, if p has two subtrees, replace it by the smallest key in its right subtree.

Remark 2

To remove a node p with two subtrees:

1. If all the keys are distinct, either the smallest key in p's right subtree, or the largest key in p's left subtree can replace the key of the node p.
2. If duplicate keys are allowed, the replacement must be selected from the right subtree. Why?





Algorithm 3 (BST Remove)

```
Remove(T, x)
    if (T == null) return T;
    if (x < T.key) T.Left = Remove(T.Left, x);
    else if (x > T.key) T.Right = Remove(T.Right, x);
    else { // x == T.key
        if (T.Left == null) T = T.Right;
        else if (T.Right == null) T = T.Left;
        else { // T has two subtrees.
            T.key = findMin(T.Right);
            T.Right = RemoveMin(T.Right);
        }
    }
    return T;
```

Algorithm 4 (BST Remove the minimum key)

```
RemoveMin(T)
   if (T == null) return T;                              // Case 1.
   if (T.left == null) return T.Right;                   // Case 2.
   for(tmp=T; tmp.Left != null ;) {                      // Case 3.
      Parent = tmp;
      tmp = tmp.Left;
   }
   Parent.Left = tmp.Right;
   return T;
```

Case 1 Trivial.

Case 2 T has no left subtree. That is, the root node T contains the minimum.

Case 3 Follow the left links until `T.Left` becomes null. Then, make its right subtree a left subtree of the parent of T.

AVL Tree

- The cost of BST operations depends on the shape of the tree.
- For example, the cost of search for a BST with n nodes is $\mathcal{O}(\log n)$ if the BST is a complete binary tree. In fact, a BST like that will provide optimal performance for all the BST operations, because the tree height is minimized.
- Otherwise, the cost of search and insertion is $\mathcal{O}(n)$.
- We can reorganize a BST into a complete binary tree, but, reorganizing a BST is a costly operation.
- Russian mathematicians Adelson-Velskii and E. M. Landis proposed the AVL Tree (1962), which relaxed the *balancing requirement*.

Definition 6

A binary tree T is height-balanced iff either T is empty or

1. T_L and T_R are height-balanced, and
2. $|h_L - h_R| \leq 1$.

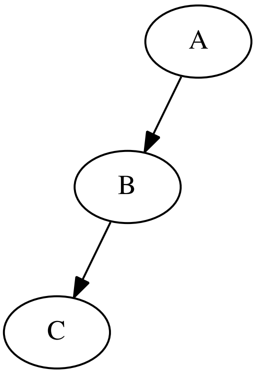
where T_L and T_R are left and right subtrees of T, h_L and h_R are heights of T_L and T_R. ($BF = h_L - h_R$ is known as a balance factor.)

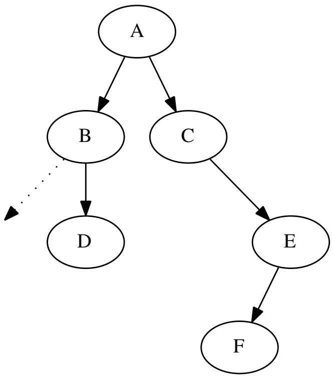
Definition 7

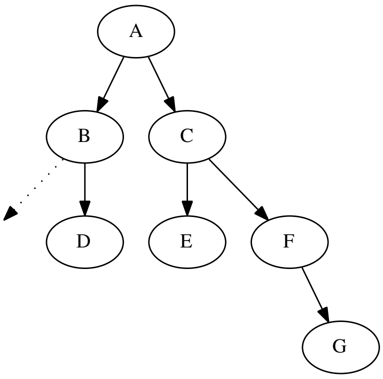
An AVL tree is a height-balanced binary search tree.

Example 13

Which trees are height balanced?

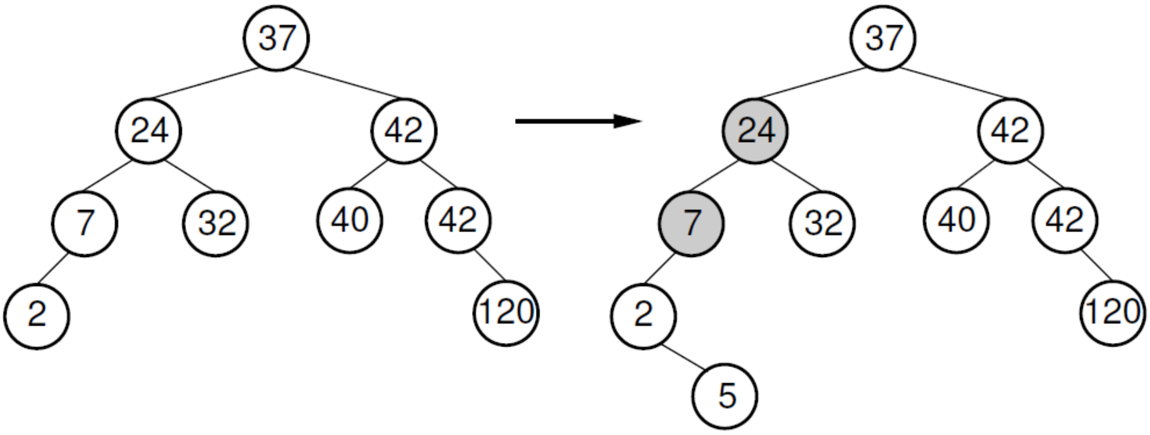






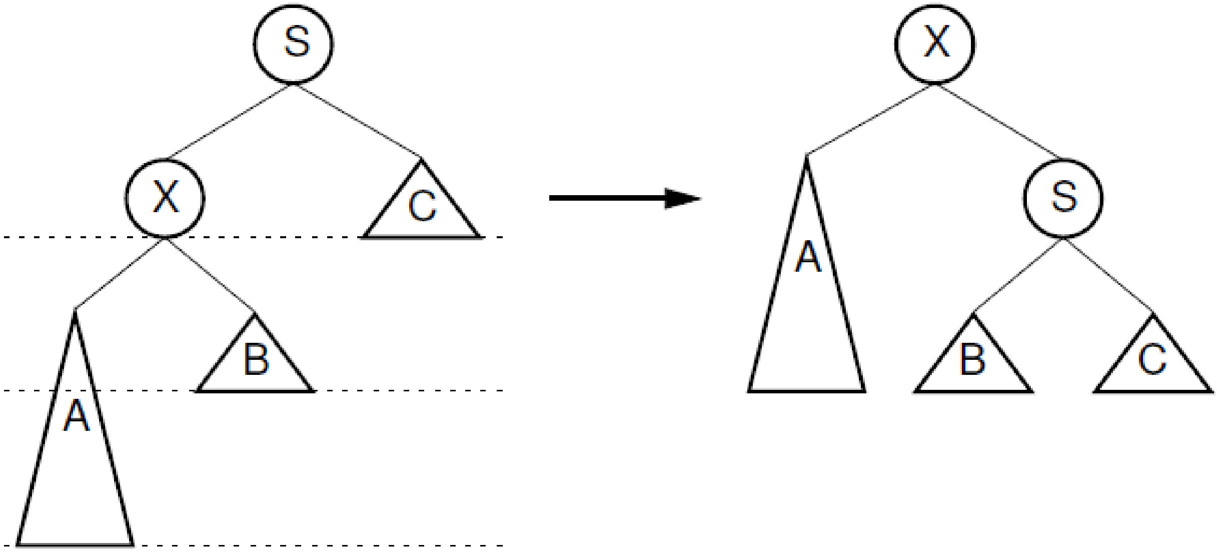
AVL Insertion: Overview

1. A new key is inserted the same way as a regular BST.
2. A new node for the key will inevitably grow some part of the tree, which may or may not cause the tree to become unbalanced.
3. If subtrees containing the new key are unbalanced, re-balance all the subtrees (recursively from small to large) until all of them are balanced.



AVL Insertion: Case 1

An AVL tree was left taller before an insertion. If a new key is added to the left subtree (X) and the left subtree A of X grows taller by one, then S becomes unbalanced. Rebalance it by a single right rotation (centered at the node S).



What does a right rotation do?

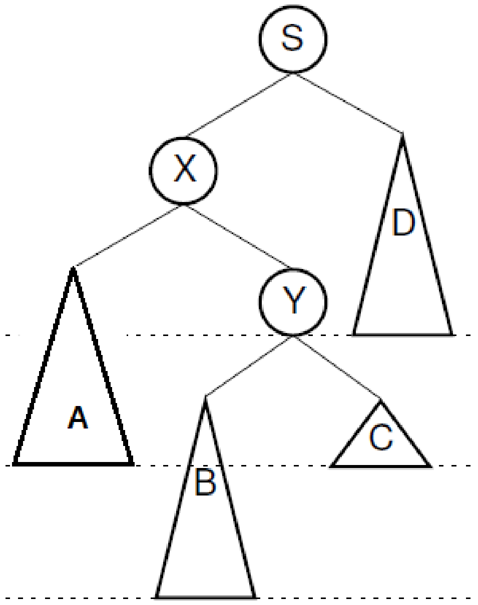
1. Watch out for a subtree labeled with B. The right subtree of X becomes the left subtree of S.
2. Invariant before the rotation: $key(X) \leq key(subtree(B)) < key(S)$. Will this be held after the rotation?

The steps for a right rotation:

```
tmp = S.left;           // tmp = X
S.left = tmp.right;     // S.left = B
tmp.right = S;          // X.right = S
parent(S) = tmp;        // X = new root of this subtree
```

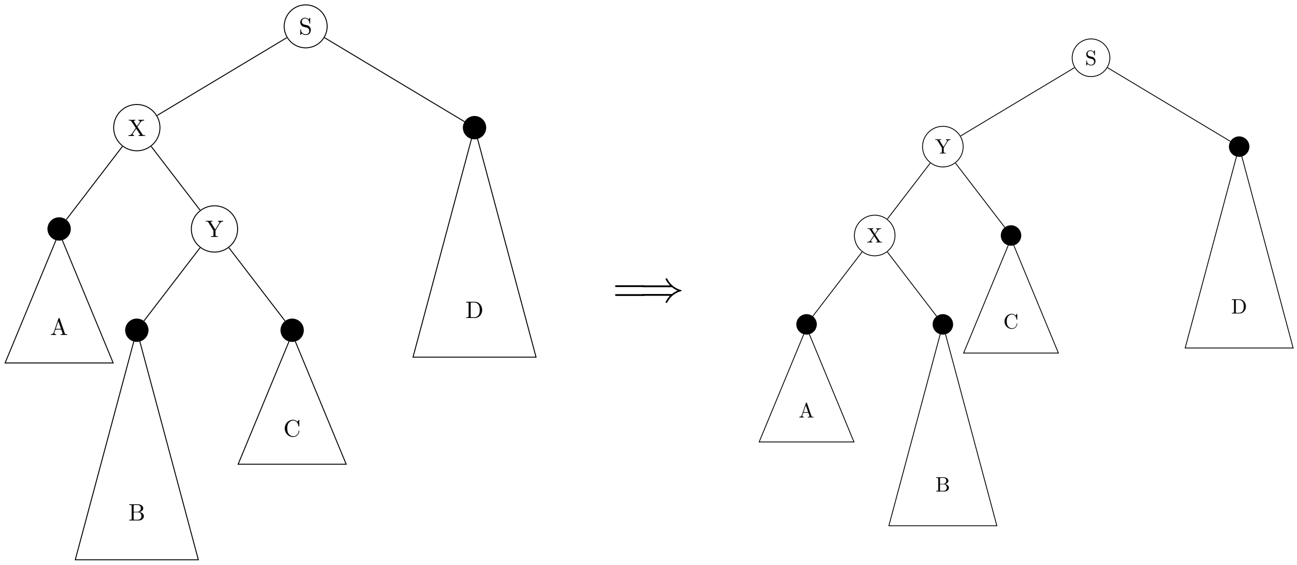
AVL Insertion: Case 2

An AVL tree was left taller before an insertion. Suppose a new key is added to the left subtree (X) and the right subtree Y of X grows taller by one, and S becomes unbalanced. Would a single rotation fix it?

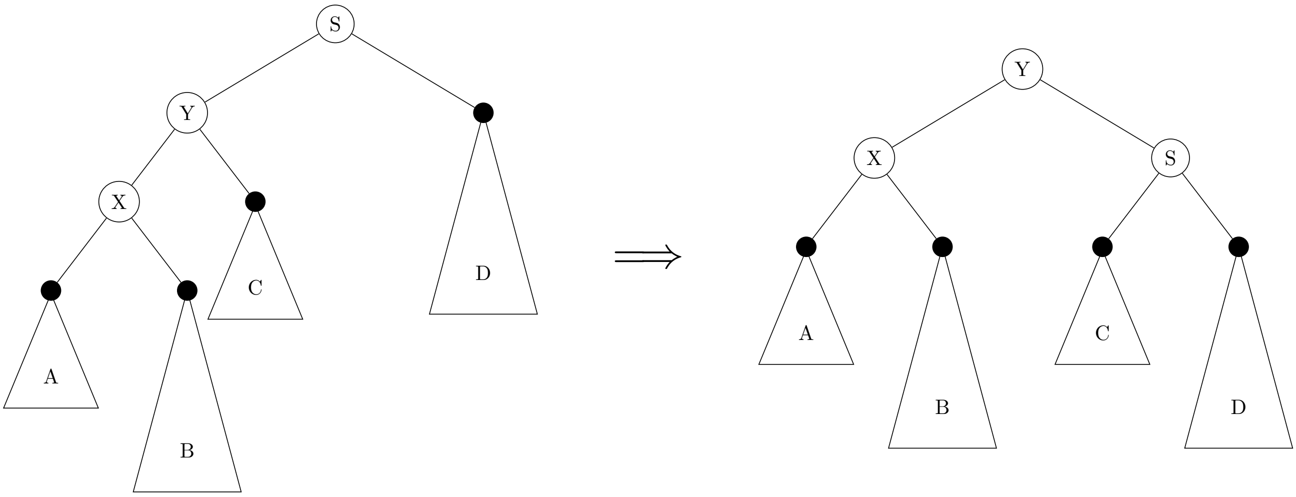


- Y becomes the left child of S, which in turn becomes the right child of X. This will still keep Y two levels deeper than the new root (*i.e.*, X).
- The subtree B will be two levels deeper than the subtree A.
- X will be the one unbalanced.

Fix it by double rotations: (1) a left rotation centered at X, and ...



Then, (2) a right rotation centered at S.



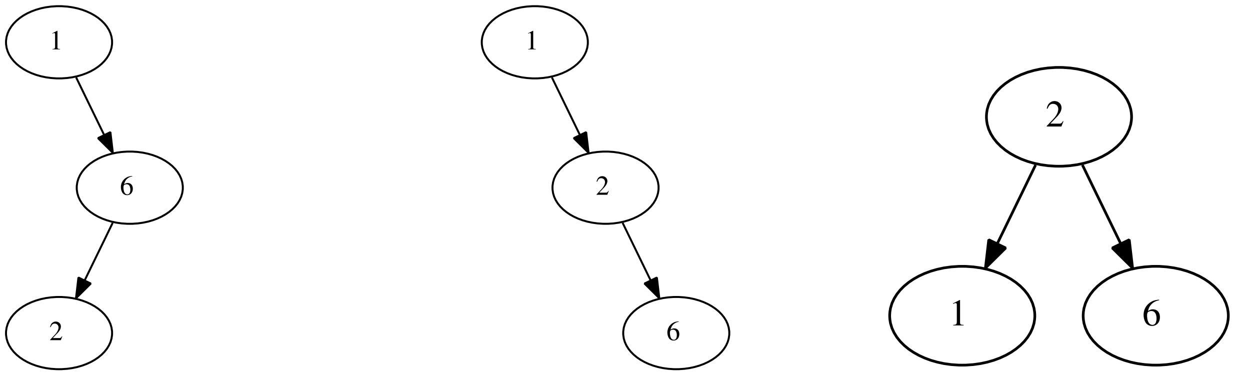
Algorithm 5 (AVL Tree Insert)

```
AvlInsert(T,x) // return a modified tree.
   if (T == null) {
      T = new Node(x); T.height = 1;
   } else if (x >= T.key) {
      T.Right = AvlInsert(T.Right,x);
      if(height(T.Right) - height(T.Left) == 2)
         if(x >= T.Right.key)
            T = RotateLeft(T);                       // Mirror of CASE 1.
         else
            T = RotateRightLeft(T);                  // Mirror of CASE 2.
   } else { /* x < T.key */
      T.Left = AvlInsert(T.Left,x);
      if(height(T.Left) - height(T.Right) == 2)
         if(x < T.Left.key)
            T = RotateRight(T);                                // CASE 1.
         else
            T = RotateLeftRight(T);                            // CASE 2.
   }
   T.height = 1 + max(height(T.Left),height(T.Right));
   return T;
```

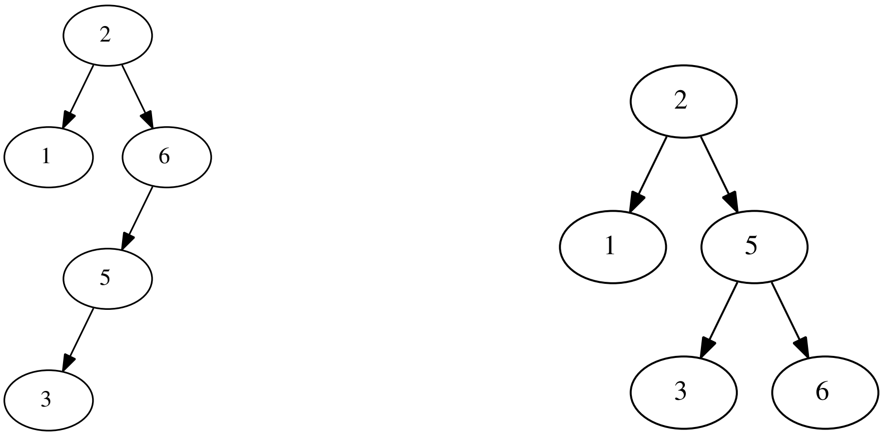
Example 14

Insert 1, 6, 2, 5, 3, 4 into an initially empty AVL tree.

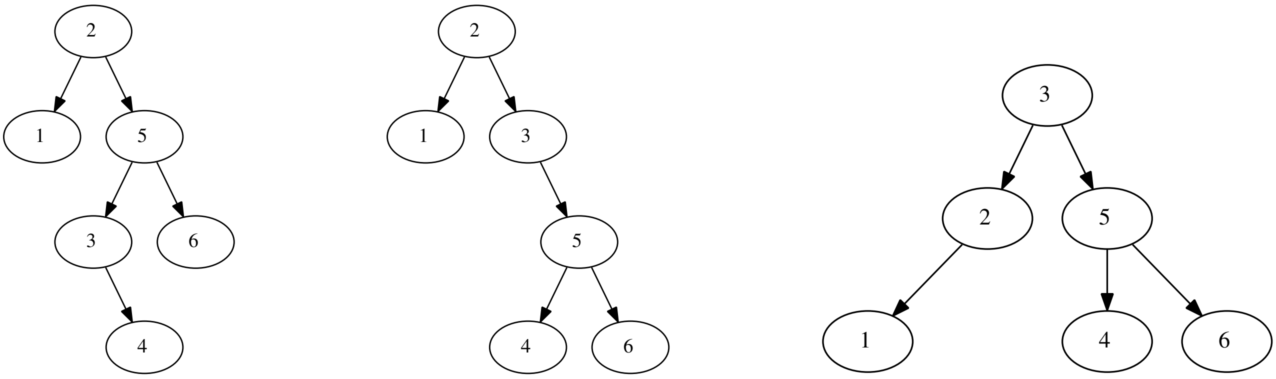
After inserting 1, 6, 2:



After inserting 1, 6, 2, 5, 3:



After inserting 1, 6, 2, 5, 3, 4:



Example 15

Insert 1, 2, 3, 4, 5, 6 into an initially empty AVL tree.

Problem 8

Does AVL Tree allow duplicate keys?

Problem 9

Algorithm 5 checks the balance of nodes from the inserted one all the way back to the root. Is it necessary to apply either a single or a double rotation more than once after a single insertion?

Lemma 5

For the Fibonacci numbers $f_0, f_1, f_2, \ldots, f_k, \ldots$ defined as $f_0 = 1$, $f_1 = 2$, $f_i = f_{i-1} + f_{i-2}$ $(i \geq 2)$, show $f_k \geq \alpha^k$, where $\alpha = (1+\sqrt{5})/2$.

Prove this lemma by induction. Note that $\alpha + 1 = \alpha^2$ (the golden ratio).

- Base: For $k = 0$, $LHS = f_0 = 1$, and $RHS = \alpha^0 = 1$.
  For $k = 1$, $LHS = f_1 = 2$, and $RHS = \alpha^1 = 1.618\ldots$.
- Induction: Assume $f_i \geq \alpha^i$ for $0 \leq i \leq k$, and show $f_{k+1} \geq \alpha^{k+1}$.
  $f_{k+1} = f_k + f_{k-1} \geq \alpha^k + \alpha^{k-1} = \alpha^{k-1}(\alpha + 1) = \alpha^{k-1}\alpha^2 = \alpha^{k+1}$

Theorem 6

The depth of an AVL tree of N nodes is $\mathcal{O}(\log N)$.

Sketch of proof:

1. Let $n(h)$ denote the minimum number of nodes of an AVL tree with height h.
2. Show $n(h)$ grows at least exponentially, that is, $n(h) \in \Omega(c^h)$ for some constant $c > 1$.
3. Show $h \in \mathcal{O}(\log N)$.

Proof. Consider an AVL tree with height h that has the minimum number of nodes. The heights of its two subtrees must be $h-1$ and $h-2$. Otherwise, the number of nodes in the AVL tree cannot be minimal.

$$n(h) = 1 + n(h-1) + n(h-2), n(1) = 1, n(0) = 0.$$

Let $f(h) = n(h) + 1$ (after adding one to the equation). Then, we obtain

$$f(h) = f(h-1) + f(h-2), f(1) = 2, f(0) = 1.$$

From Lemma 5, $f(h) \geq \alpha^h$ where $\alpha = (1+\sqrt{5})/2 > 1$. Therefore,

$$h \leq \log_\alpha f(h) = \log_\alpha (1 + n(h)) \leq \log_\alpha (1 + N).$$

□

Splay Tree

Observations from the AVL trees:

- Single rotation moves a node one level up.
- Double rotation moves a node two levels up.

Key ideas of Splay Tree

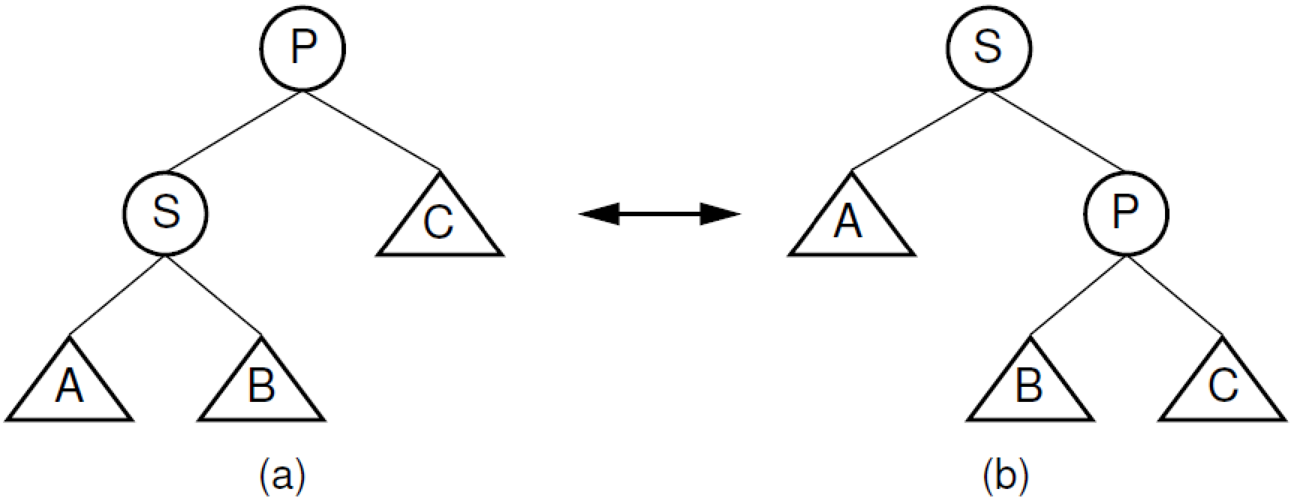
- Attempt to balance by doing similar things (*i.e.*, rotations), but balanced height is not guaranteed.
- Splaying is done in bottom-up fashion until the target node becomes the root node.
- The target node is the one searched for, inserted, or deleted.
  - For a failed search, the target is the node to which the search key would be attached if inserted.
  - For a deletion, the parent node of the deleted one is the target.

Types of rotations

- Three types of rotations: single (zig or zag), zigzag, zigzig.
- Determined by the structural relationship among the target node, its parent and its grand-parent nodes.

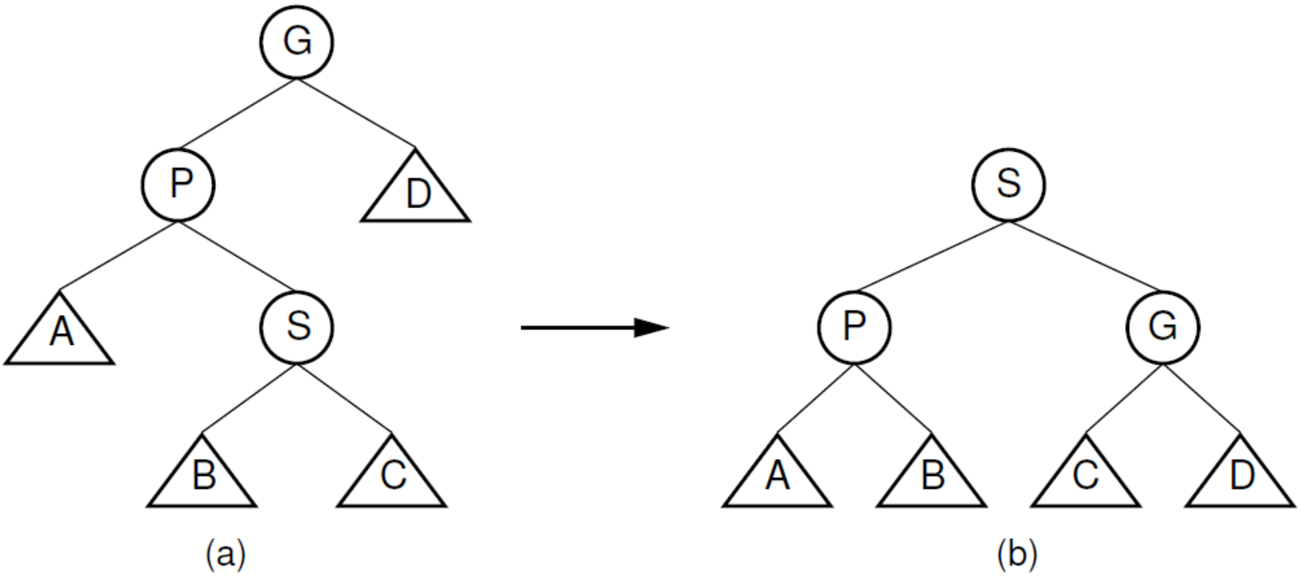
A single rotation if the target node (S) is a child of the root.

- Equivalent to a single rotation in AVL trees.



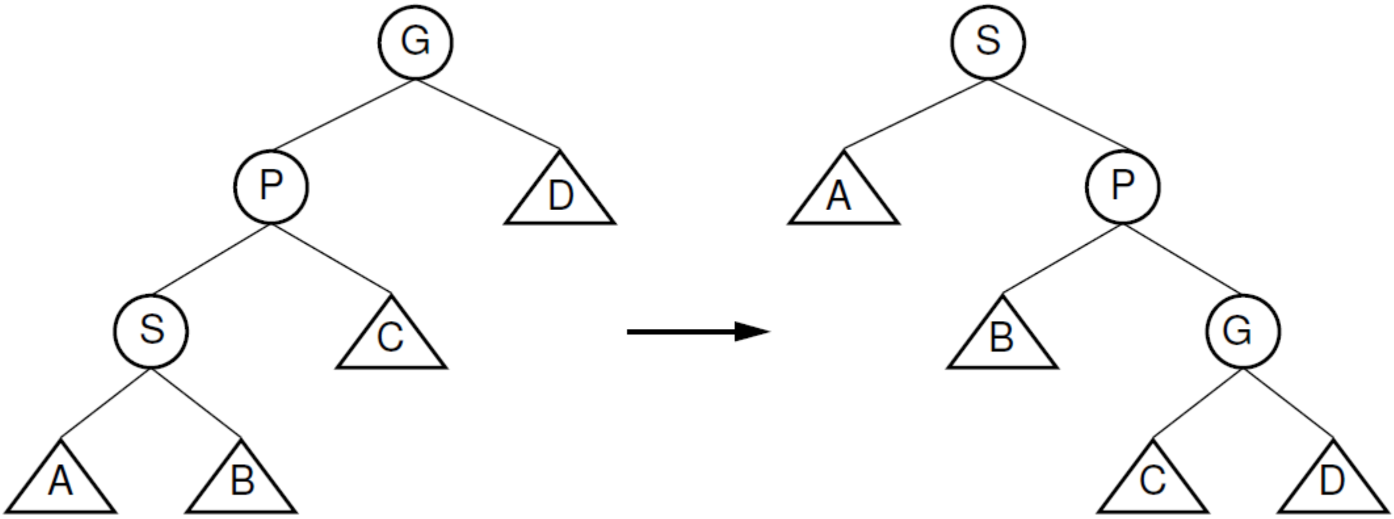
A zigzag rotation if the target (S), its parent and grand-parent nodes are in zigzag formation.

- Equivalent to a double rotation in AVL trees.



A zigzig rotation if the target (S), its parent and grand-parent nodes are in zigzig formation.

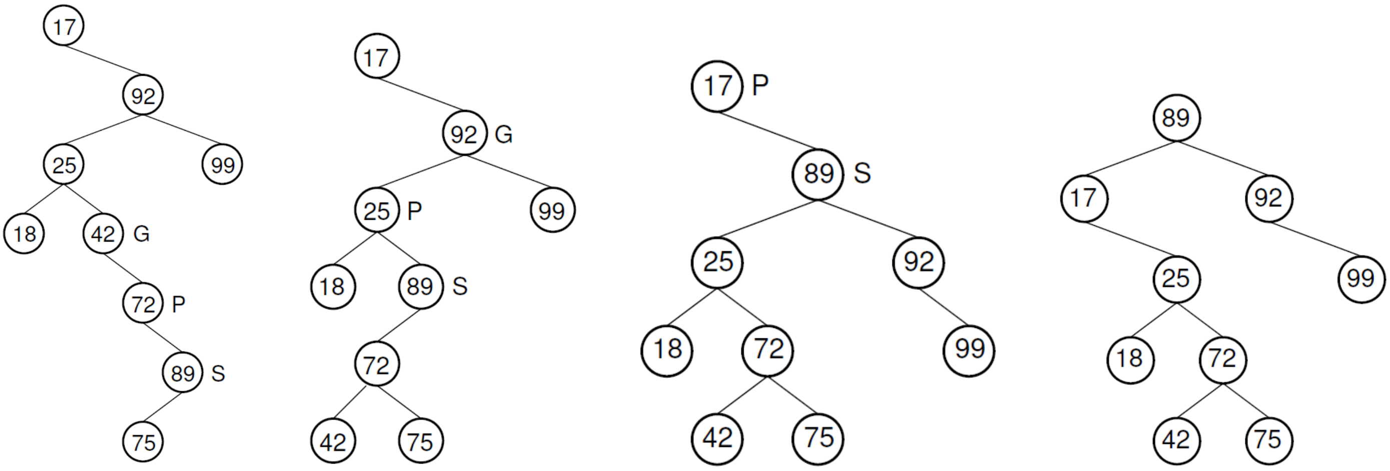
- No equivalent operation in AVL trees.



(a) (b)

Example 16

Splay the BST below after performing a search for key 89.



Analysis of Splay Tree

- Even a search operation may alter the tree structure.
- A single operation (insert, delete, search) takes $\mathcal{O}(n)$ time.
- M consecutive operations take $\mathcal{O}(M \log n)$ time for $M \geq n$. The amortized cost of a single operation is $\mathcal{O}(\log n)$.
- Keep frequently accessed nodes close to the top (or the root node).

Optimality of Binary Search Trees

- Keeping frequently accessed keys close to the top will minimize the average search cost.
- The average number of nodes probed is

  $$\sum_{i=1}^{n} p_i l_i$$

  where $p_i = Prob[k_i \text{ is searched for}]$, $l_i = 1 + level(k_i)$.
- If search frequencies (or probabilities) of keys are known, we can find a BST that has minimum average search time.
- For simplicity, we assume that the probability of a key not being found is zero.

Optimal Binary Search Tree (OBST)

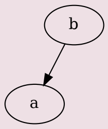
Problem 10

For a given set of keys $k_1, k_2, \ldots, k_n (k_1 < k_2 < \ldots < k_n)$ with search probabilities $p_1, p_2, \ldots, p_n$, find an OBST that minimizes the average search cost $\sum_{i=1}^{n} p_i l_i$.

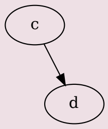
- Assume the set of keys (and their probabilities) remains static. No more insertions nor deletions.
- Let $c(i,j)$ denote the *smallest* average search cost of a BST with keys $k_i, k_{i+1}, \ldots, k_j$. (In other words, $c(i,j)$ is the average search cost of an OBST with keys $k_i, k_{i+1}, \ldots, k_j$.)
  - Note that, in $c(i,j) = \min\{\sum_{m=i}^{j} p_m l_m\}$ for $1 \le i \le j \le n$, the value of l_m (that is, $1 + \mathit{level}(k_m)$) is determined by a (smaller) BST of keys $k_i, k_{i+1}, \ldots, k_j$ rather than a (larger) BST of entire n keys.
- The goal is to find a BST whose average search time is $c(1,n)$.

Example 17

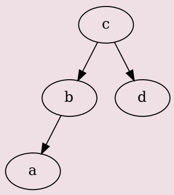
Consider a set of four keys {A, B, C, D} to be searched for with frequencies 1, 2, 4 and 3, respectively. Compute $c(1,2)$, $c(3,4)$ and $c(1,4)$ for the following BSTs.



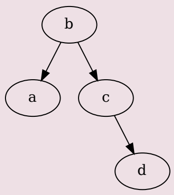
$c(1,2) = 1 \times 2 + 2 \times 1$



$c(3,4) = 4 \times 1 + 3 \times 2$



$$\begin{aligned} Cost_c &= 1 \times 3 + 2 \times 2 + 4 \times 1 + 3 \times 2 \\ &= (1+2+4+3) + c(1,2) + c(4,4) \end{aligned}$$



$$\begin{aligned} Cost_b &= 1 \times 2 + 2 \times 1 + 4 \times 2 + 3 \times 3 \\ &= (1+2+4+3) + c(1,1) + c(3,4) \end{aligned}$$

Suppose k_r is the root of a (not necessarily optimal) BST T with keys $k_1, k_2, \dots, k_n$. Then, the average search cost is

$$\begin{aligned} Cost_{k_r} &= \sum_{i=1}^{r-1} p_i l_i + p_r + \sum_{i=r+1}^{n} p_i l_i \\ &= \sum_{i=1}^{r-1} p_i (1 + l'_i) + p_r + \sum_{i=r+1}^{n} p_i (1 + l'_i) \\ &= \sum_{i=1}^{n} p_i + \sum_{i=1}^{r-1} p_i l'_i + \sum_{i=r+1}^{n} p_i l'_i \\ &\geq \sum_{i=1}^{n} p_i + c(1, r-1) + c(r+1, n) \end{aligned}$$

The level of k_i in the subtrees of T is one less than its level in T. That is, $l'_i = l_i - 1$ is the level of k_i in the subtrees.

Take advantage of the principle of optimality to realize this.

Lemma 7

If T is an OBST for a given set of keys, then so is each of T's subtrees.

We do not know which k_r will make the root of an OBST. However,...

Corollary 8

For a given set of n keys, $k_1 < k_2 < \dots < k_n$, the OBST is one of the n trees, $T_1, T_2, \dots, T_n$, where T_r $(1 \leq r \leq n)$ is a binary search tree whose root contains k_r, whose left subtree is an OBST containing $k_1, \dots, k_{r-1}$, and whose right subtree is an OBST containing $k_{r+1}, \dots, k_n$.

Note that the average search cost of T_r is $\sum_{i=1}^{n} p_i + c(1, r-1) + c(r+1, n)$ because both of its subtrees are optimal. (In Example 17, the BSTs T_c and T_b are candidates of OBST.)

Since one of the n trees, $T_1, T_2, \ldots, T_n$, is an OBST, the *smallest* average search cost, $c(1, n)$, is the smallest among the costs of those n trees.

$$c(1, n) = \sum_{i=1}^{n} p_i + \min_{r=1}^{n}\{c(1, r-1) + c(r+1, n)\}.$$

In general,

$$c(low, high) = \sum_{i=low}^{high} p_i + \min_{r=low}^{high}\{c(low, r-1) + c(r+1, high)\}.$$

If $low > high$, then $c(low, high)$ represents the search cost of an empty BST. Therefore,

$$c(low, high) = 0 \text{ if } low > high.$$

For example,

$$c(3,7) = \sum_{i=3}^{7} p_i + \min\{c(3,2) + c(4,7),\ c(3,3) + c(5,7),$$
$$c(3,4) + c(6,7),\ c(3,5) + c(7,7),\ c(3,6) + c(8,7)\}.$$

$$c(1,3) = \sum_{i=1}^{3} p_i + \min\{c(1,0) + c(2,3),\ c(1,1) + c(3,3),\ c(1,2) + c(4,3)\}.$$

This is a lot of computation. We should find an efficient way.

Use a $(n+2) \times (n+1)$ matrix to compute $c(i,j)$ once and reuse them.

- The keys $(k_1, k_2, \ldots, k_n)$ are in a 1-based array representation so that negative offsets are avoided in the matrix.
- The matrix has $n+2$ rows and $n+1$ columns (both numbered from zero).
    - (*e.g.*) The computation of $c(1,n)$ requires $c(1,0)$ and $c(n+1,n)$.
    - The top row (for $low = 0$) will never be used.
- The elements on the diagonal will be set to p_i, and all elements below the diagonal are set to (or considered) zero by Algorithm 6.

$$c(l,h) = \begin{cases} 0 & \text{if } l > h, \\ p_l \text{ (or } \sum_{i=l}^{h} p_i) & \text{if } l = h. \end{cases}$$

In which order the $c(low, high)$ values should be computed?

$c(low, high)$ depends on $c(low, low-1), \ldots, c(low, high-1)$ (left on the row) and $c(low+1, high), \ldots, c(high+1, high)$ (below on the column).

1. Go by diagonals bottom-up.
2. Or, go by rows bottom-up, left to right on each row.
3. Or, go by columns left to right, bottom-up on each column.

For example, the computation of $c(3,7) = \sum_{i=3}^{7} p_i + \min\{c(3,2) + c(4,7),\ c(3,3) + c(5,7), c(3,4) + c(6,7),\ c(3,5) + c(7,7),\ c(3,6) + c(8,7)\}$ depends on ten elements in the matrix.

						⋮	
. . .	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	. . .
						(4,7) (5,7) (6,7) (7,7) (8,7) ⋮	

Algorithm 6 (OBST)

```
for(low=n+1; low $\geq$ 1 ;low--)                    // bottom-up
   for(high=low-1; high $\leq$ n ;high++) {          // left to right
      if (low > high) {
         c(low,high) = 0;                            // an empty tree
         root(low,high) = 0;
      } else {
         c(low,high) =
               $\sum_{i=low}^{high} p_i + \min_{r=low}^{high}\{c(low, r-1) + c(r+1, high)\}$;
         root(low,high) = $r$ that minimizes c(low,high);
      }
   }
```

Analysis of the OBST algorithm

The memory and time complexities are obviously $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively.

What can we do to make it run faster?

- Find a way to avoid computing $\sum_{i=low}^{high} p_i$ repeatedly.
- Reduce the time complexity to $\mathcal{O}(n^2)$. Apply the *monotonicity property* of OBST.

Theorem 9 (Knuth's monotonicity property)

For any integer pairs (i_1, j_1) and (i_2, j_2) such that $i_1 \leq i_2 \leq j_1 \leq j_2$, the root of sub-OBST $(k_{i_i}, \ldots, k_{j_1})$ is never on the right side of the root of sub-OBST $(k_{i_2}, \ldots, k_{j_2})$. That is, $root(i_1, j_1) \leq root(i_2, j_2)$.

Example 18

Consider a set of four keys {A, B, C, D} to be searched for with frequencies 1, 2, 4 and 3, respectively. Apply the OBST algorithm (Algorithm 6) and show the cost and root tables.

0	0	0	0	0
0	1	4	11	17
	0	2	8	14
		0	4	10
			0	3
				0

`cost[6][5]` table

0	0	0	0	0
0	1	2	3	3
	0	2	3	3
		0	3	3
			0	4
				0

`root[6][5]` table

Row 4: $(low = 4, 3 \leq high \leq 4)$

$$\begin{aligned} c(4,4) &= p_4 = 3. \\ root(4,4) &= 4. \end{aligned}$$

Row 3: $(low = 3, 2 \leq high \leq 4)$

$$\begin{aligned} c(3,3) &= p_3 = 4. \\ root(3,3) &= 3. \\ c(3,4) &= p_3 + p_4 + \min_{3 \leq r \leq 4} \left\{ \begin{array}{c} c(3,2) + c(4,4) \\ c(3,3) + c(5,4) \end{array} \right\} = 3 + 4 + 3 = 10. \\ root(3,4) &= 3. \end{aligned}$$

Row 2: $(low = 2, 1 \leq high \leq 4)$

$$\begin{aligned} c(2,2) &= p_2 = 2. \\ root(2,2) &= 2. \\ c(2,3) &= p_2 + p_3 + \min_{2 \leq r \leq 3} \left\{ \begin{array}{c} c(2,1) + c(3,3) \\ c(2,2) + c(4,3) \end{array} \right\} = 2 + 4 + 2 = 8. \\ root(2,3) &= 3. \\ c(2,4) &= p_2 + p_3 + p_4 + \min_{2 \leq r \leq 4} \left\{ \begin{array}{c} c(2,1) + c(3,4) \\ c(2,2) + c(4,4) \\ c(2,3) + c(5,4) \end{array} \right\} \\ &= 2 + 4 + 3 + 5 = 14. \\ root(2,4) &= 3. \end{aligned}$$

Row 1: ($low = 1, 0 \le high \le 4$)

$$
\begin{aligned}
c(1,1) &= p_1 = 1.\\
root(1,1) &= 1.\\
c(1,2) &= p_1 + p_2 + \min_{1 \le r \le 2} \left\{ \begin{array}{l} c(1,0) + c(2,2) \\ c(1,1) + c(3,2) \end{array} \right\} = 1 + 2 + 1 = 4.\\
root(1,2) &= 2.\\
c(1,3) &= p_1 + p_2 + p_3 + \min_{1 \le r \le 3} \left\{ \begin{array}{l} c(1,0) + c(2,3) \\ c(1,1) + c(3,3) \\ c(1,2) + c(4,3) \end{array} \right\}\\
&= 1 + 2 + 4 + 4 = 11.\\
root(1,3) &= 3.\\
c(1,4) &= p_1 + p_2 + p_3 + p_4 + \min_{1 \le r \le 4} \left\{ \begin{array}{l} c(1,0) + c(2,4) \\ c(1,1) + c(3,4) \\ c(1,2) + c(4,4) \\ c(1,3) + c(5,4) \end{array} \right\}\\
&= 1 + 2 + 4 + 3 + 7 = 17.\\
root(1,4) &= 3.
\end{aligned}
$$

Example 19

Build an OBST for the set of keys given in Example 18.

Since root[1][4]=3, the root of the OBST is k_3 (*i.e.*, C). The left subtree has two keys (k_1, k_2) and the right subtree has one key (k_4).

Since root[1][2]=2, the root of the left subtree is k_2 (*i.e.*, B).

Since root[4][4]=4, the root of the right subtree is k_4 (*i.e.*, D).

Remark 3

Construction of an OBST from the `root` table takes $\mathcal{O}(n)$ time.