Binary Search Trees

Manolis Koubarakis

Search

- The retrieval of a particular piece of information from large volumes of previously stored data is a fundamental operation, called **search** (αναζήτηση).
- Search is an important operation in many applications e.g., bank information systems, airline information systems, Web search etc.

The ADT Symbol Table

- A **symbol table** (πίνακας συμβόλων) is a data structure of **items with keys** (αντικείμενα με κλειδιά) that supports the following operations:
  - **Initialize** a symbol table to be empty.
  - **Count** the number of items in a symbol table.
  - **Insert** a new item.
  - **Search** for an item (or items) having a given key.
  - **Delete** a specified item.
  - **Select** the k-th smallest item in a symbol table.
  - **Sort** the symbol table (visit all items in order of their keys).

- Items can be thought of as pairs **(key, value).**

- If keys are distinct, symbol tables are also called **maps**. If they are not, they are also called **dictionaries** (λεξικά).

Example

- We can use a symbol table to represent **information about students** in a university.
- The key can be their **student number**.
- The item can also contain other information about students (the value for the key): name, address, year of study, courses they have taken etc.
- Keys can be used to find information about a student, update this information etc.
- Since student numbers are distinct are symbol table is a **map**.

A Symbol Table Interface

```
void STinit(int);
int STcount();
void STinsert(Item);
Item STsearch(Key);
void STdelete(Item);
Item STselect(int);
void STsort(void (*visit)(Item));
```

Key-Indexed Search

- Suppose that key values are **distinct small numbers** (e.g., numbers less than M).
- The simplest symbol table implementation in this case is based on storing the items in an **array indexed by the keys**.
- The algorithms implementing the operations of the symbol table interface are straightforward.
- Insertion and search take $O(1)$ time while select takes $O(M)$ time.

Sequential Search

- If the key values are from a large range, one simple approach for a symbol table implementation is to **store the items contiguously in an array, in order according to the keys.**
- The algorithms implementing the operations of the symbol table interface in this case are also straightforward.
- Insertion and search take $O(n)$ time where n is the number of items.
- Select takes $O(1)$ time.
- If we use an **ordered linked list**, insertion, search and select all take $O(n)$ time.

Binary Search

- In the array implementation of sequential search, we can reduce significantly the search time for a large set of items by using a search procedure called **binary search** (δυαδική αναζήτηση) which is based on the **divide-and-conquer paradigm.**

Binary Search

- The problem to be addressed in binary searching is to find the position of a **search key** `K` in an ordered array `A[0:n-1]` of distinct keys **arranged in ascending order**:

  `A[0] < A[1] < … < A[n-1].`
- The algorithm chooses the key in the middle of `A[0:n-1]`, which is located at `A[Middle]`, where `Middle=(0+(n-1))/2`, and compares the search key `K` and `A[Middle]`.
- If `K==A[Middle]`, the search terminates successfully.
- If `K < A[Middle]` then further search is conducted among the keys to the left of `A[Middle]`.
- If `K > A[Middle]` then further search is conducted among the keys to the right of `A[Middle]`.

Iterative Binary Search

```
int BinarySearch(Key K)
{
    int L, R, Midpoint;

    /* Initializations */
    L=0;
    R=n-1;

    /* While the interval L:R is  non-empty, test key K against the middle key */
    while (L<=R){
      Midpoint=(L+R)/2;
      if (K==A[Midpoint]){
        return Midpoint;
      } else if (K > Midpoint) {
        L=Midpoint+1;
      } else {
        R=Midpoint-1;
      }
    }
    /* If the search interval became empty, key K was not found */
    return -1;
}
```

Recursive Binary Search

```
int BinarySearch (Key K, int L, int R)
{
   /* To find the position of the search key K in the subarray
      A[L:R]. Note: To search for K in A[0:n-1], the initial call
      is BinarySearch(K, 0, n-1) */

   int Midpoint;

   Midpoint=(L+R)/2;

   if (L>R){
     return -1;
   } else if (K==A[Midpoint]){
     return Midpoint;
   } else if (K > A[Midpoint]){
     return BinarySearch(K, Midpoint+1, R);
   } else {
     return BinarySearch(K, L, Midpoint-1);
   }
}
```

Complexity

- Let us compute the running time of recursive binary search.
- We call an entry of our array a **candidate** if, at the current stage of the algorithm, we cannot rule out that this entry has key equal to `K`.
- We observe that a constant amount of primitive operations are executed at each recursive call of function `BinarySearch`.
- Hence the running time is proportional to the number of recursive calls performed.
- Moreover, **the number of remaining candidates is reduced by at least half with each recursive call.**

Complexity (cont'd)

- Initially, the number of candidate entries is n. After the first call to `BinarySearch`, it is at most $\frac{n}{2}$. After the second call, it is at most $\frac{n}{4}$ and so on.
- In general, after the i-th call to `BinarySearch`, the number of candidate entries is at most $\frac{n}{2^i}$.
- In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate entries. Hence, the maximum number of recursive calls performed, is the smallest integer m such that $\frac{n}{2^m} < 1$.

Complexity (cont'd)

- Equivalently, $2^m > n$.
- Taking logarithms in base 2, we have $m > \log n$.
- Thus, we have $m = \lfloor \log n \rfloor$+1 which implies that the complexity of recursive `BinarySearch` is $\boldsymbol{O(\log n)}$.
- The complexity of iterative `BinarySearch` is also $\boldsymbol{O(\log n)}$.

Binary Search Trees

- To overcome the problem that insertions are expensive, we use an explicit tree structure as the basis for symbol table implementation.
- **Binary search trees** (δένδρα δυαδικής αναζήτησης) are an excellent data structure for representing sets whose elements are ordered by some linear order.
- A **linear order** (γραμμική διάταξη) $<$ on a set S satisfies two properties:
  - For any $a, b \in S$, exactly one of $a < b, a = b$ or $a > b$ is true.
  - For all $a, b, c \in S$, if $a < b$ and $b < c$ then $a < c$ (transitivity).

- Examples of sets with a natural linear order are integers, floats, characters and strings in C.

Definition

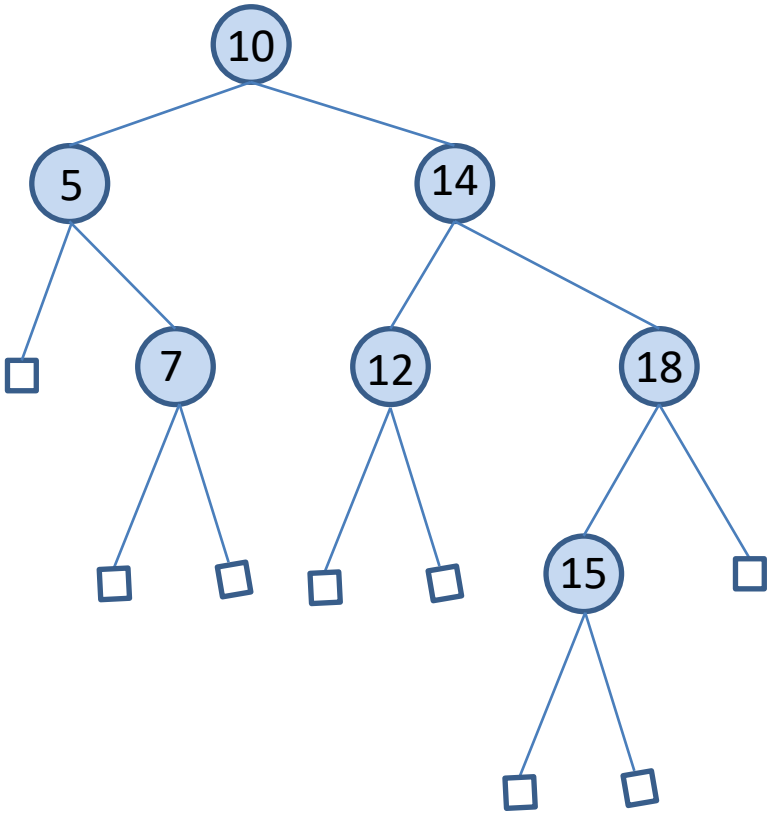
- A **binary search tree (BST)** is a binary tree that has a key associated with each of its internal nodes, and it also has the following property:
  - For each node *N*: keys in the left subtree of *N* $\leq$ key *K* in node *N* $\leq$ keys in the right subtree of *N*

- The above condition is called the **binary search tree property.**

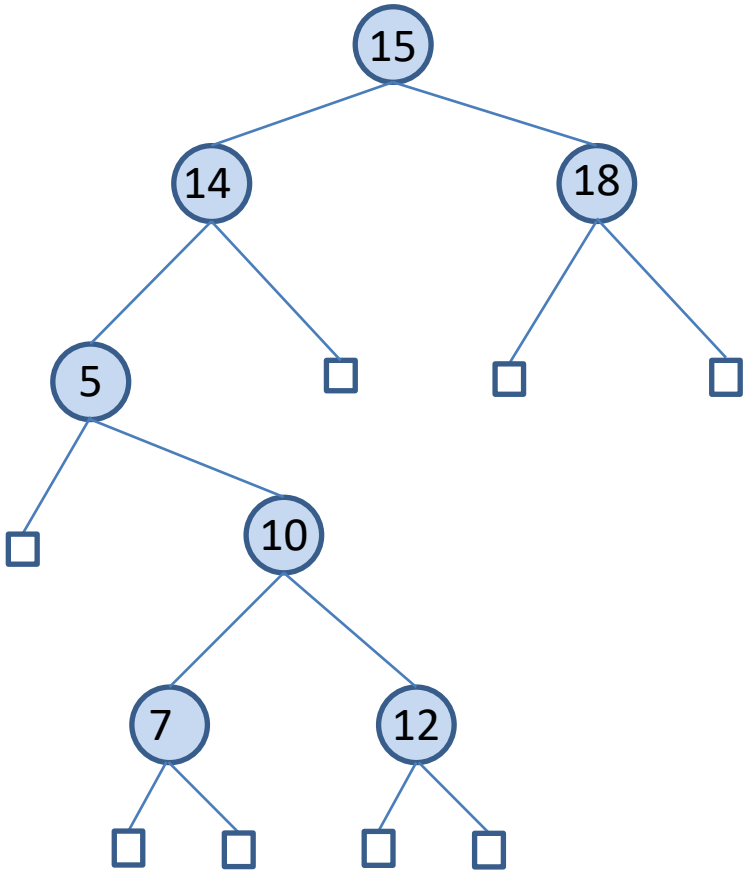
Duplicate Keys

- The previous definition **allows duplicate keys**, this definition of BSTs can be used to implement dictionaries. If we want to use BSTs to implement maps, then we have to change the $\leq$ to $<$ in the above definition.

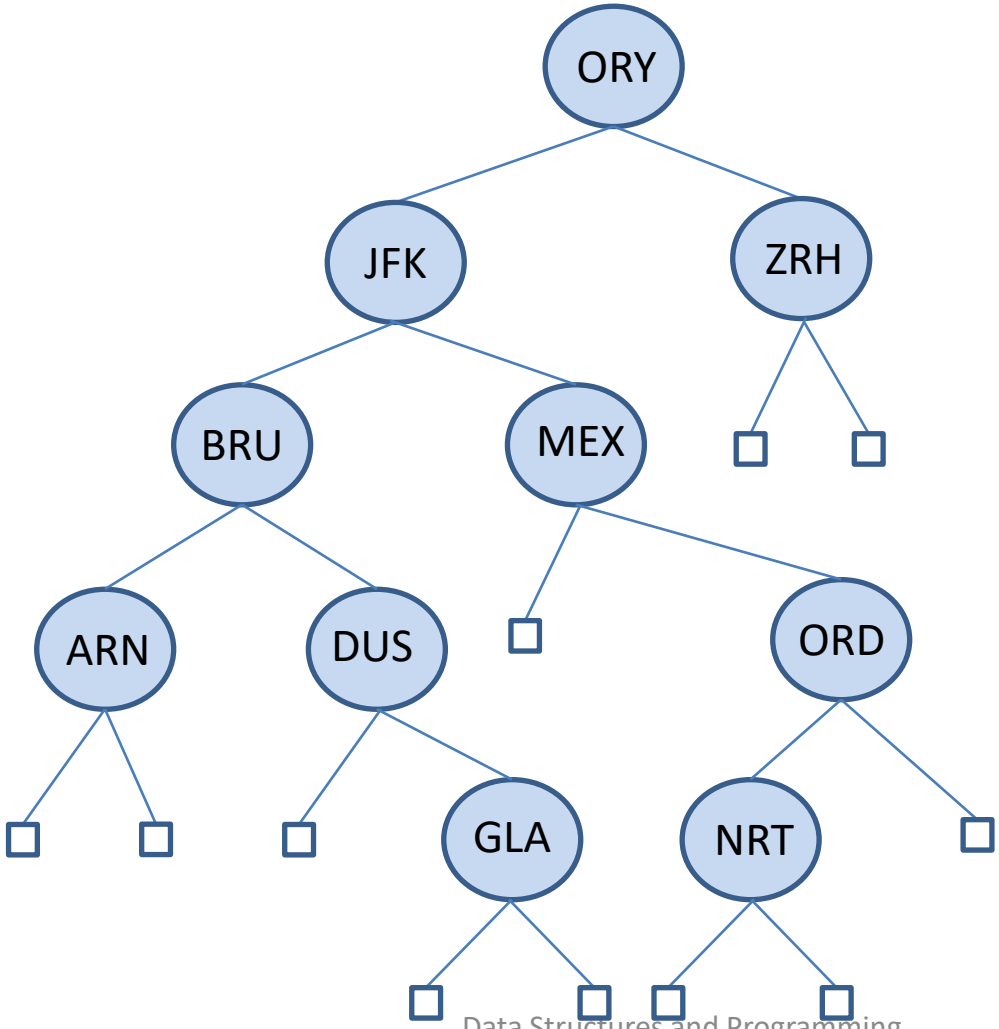
Example



Example (for the same set)



Example



Type Definitions for BSTs

- The following statements define the data structure for a BST:

```
typedef struct STnode* link;
struct STnode { Item item; link l, r; int N; };
```

- Each node in a BST contains an item `item` (with a key), a left link `l`, a right link `r` and an integer `N` which counts how many nodes there are in the tree (or subtree).
- Items, their data types and operations on them can be defined in an appropriate interface file `Item.h`.

The Interface File `Item.h`

```
typedef int Item;
typedef int Key;

#define NULLitem -1 /* NULLitem is a constant */
#define key(A) (A)
#define less(A, B) (key(A) < key(B))
#define eq(A, B) (!less(A, B) && !less(B, A))

Item ITEMrand(void);
int ITEMscan(Item *);
void ITEMshow(Item);
```

Notes

- The previous interface file assumes that items consist just of keys. This assumption can be changed.
- `NULLitem` is a constant to be returned when a BST does not contain a key.
- The function `ITEMrand` returns a random item.
- The function `ITEMscan` reads an item from the standard input.
- The function `ITEMshow` prints an item on the standard output.
- `less` and `eq` are macros that we will be using in our code (they could be defined as functions too).

An Implementation of the Item Interface

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"

int ITEMrand(void)
         { return rand(); }

int ITEMscan(int *x)
         { return scanf("%d", x); }

void ITEMshow(int x)
         { printf("%5d ", x); }
```

The Operations `Stinit` and `STcount`

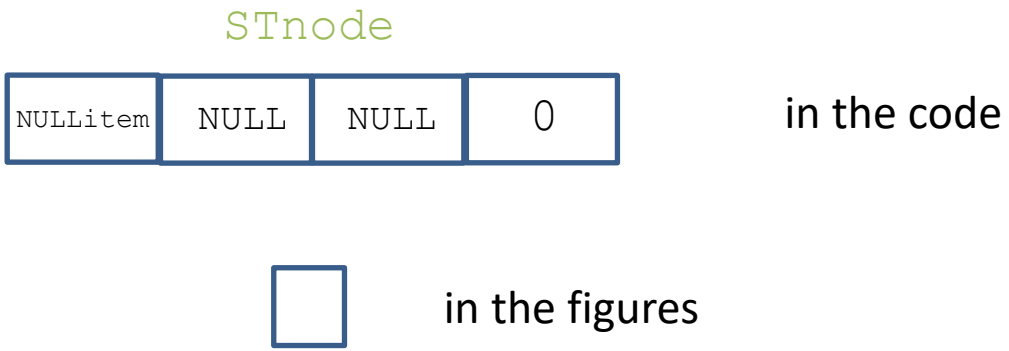
```
static link head, z;

link NEW(Item item, link l, link r, int N)
  { link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r; x->N = N;
    return x;
  }

void STinit()
  { head = (z = NEW(NULLitem, NULL, NULL, 0)); }

int STcount() { return head->N; }
```

Notes

- `head` is a static pointer variable that points to the **root** of the tree.
- `z` is a static pointer variable which points to a **dummy node** representing **empty trees** or **external nodes**.
- `NEW` is a function which creates a new tree node and returns a pointer to it.

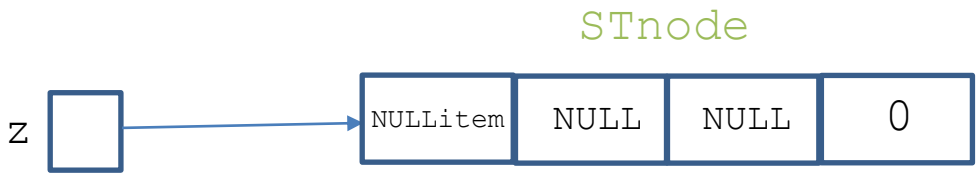
External Nodes

- In our figures external nodes are represented by **small rectangles**.
- In our code **external nodes** are implemented by structures of type `STnode` with elements as shown below.



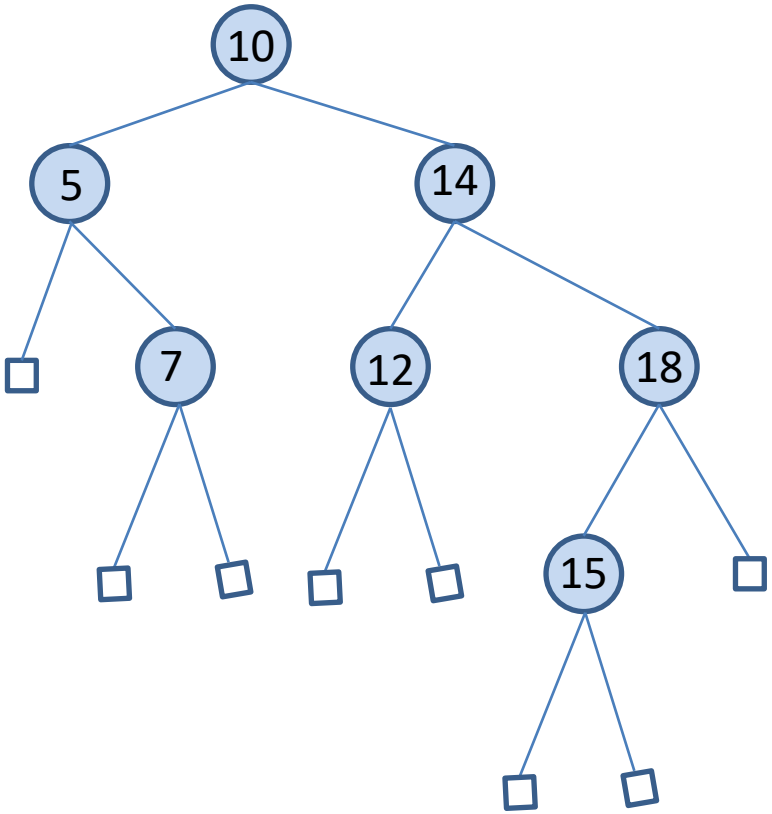
The Pointer Variable z

- There is **only one dummy (external) node** in the implementation, although in our figures we will show many such nodes.

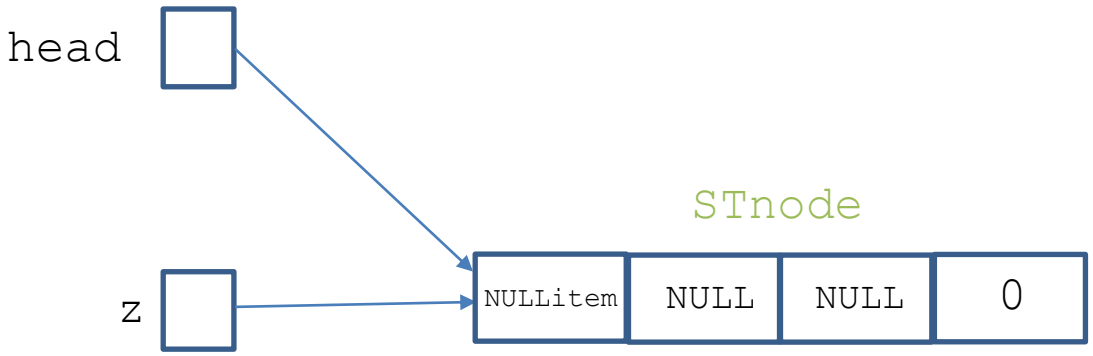


- Pointer variable z points to that node.

Example



The Result of `STinit()`



Important

- If we did not have these dummy nodes to denote external nodes **both our theoretical discussions and our implementations would be more complex.**

Searching for a Key Recursively

- To **search for a key** K in a BST T, we compare K to the key K_r of the root of T.
- If $K == K_r$, the search terminates successfully.
- If $K < K_r$, the search continues recursively in the left subtree of T.
- If $K > K_r$, the search continues recursively in the right subtree of T.
- If T is the empty tree, we have a search miss and return `NULLitem` (which we defined to be -1).

The Function STsearch

```
Item searchR(link h, Key v)
  { Key t = key(h->item);
    if (h == z) return NULLitem;
    if eq(v, t) return h->item;
    if less(v, t) return searchR(h->l, v);
            else return searchR(h->r, v);
  }

Item STsearch(Key v)
  { return searchR(head, v); }
```

Notes

- The function `STsearch` calls a recursive function `searchR` which does the work.
- At each step of `searchR`, we are guaranteed that no parts of the tree other than the **current subtree** can contain items with the search key.
- Just as the size of the interval in binary search shrinks by a little more than half on each iteration, **the current subtree in binary tree search is smaller than the previous (by about half, ideally).**

Inserting a Key

- An essential feature of BSTs is that **insertion is as easy to implement as search.**

Inserting a Key Recursively

- To **insert an item with key** K in a BST T, we compare K to the key K_r of the root of T.
- If $K < K_r$, the algorithm continues recursively in the left subtree of T.
- If $K \geq K_r$, the algorithm continues recursively in the right subtree of T.
- If T is the empty tree, then the item with key K is inserted there.

Inserting a Key Recursively (cont'd)

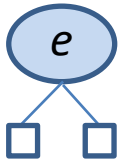
- If we disallow **duplicate keys,** the insertion algorithm is as follows.
- If $K = K_r$, the key is already present in T and the algorithm stops.
- If $K < K_r$, the algorithm continues recursively in the left subtree of T.
- If $K > K_r$, the algorithm continues recursively in the right subtree of T.
- If T is the empty tree, then the item with key K is inserted there.

Example

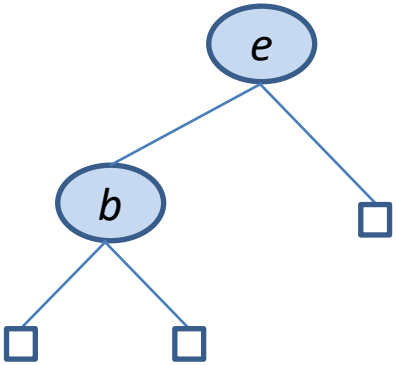
- Let us insert keys *e, b, d, f, a, g, c* into an initially empty tree in the order given.

Initial Empty Tree

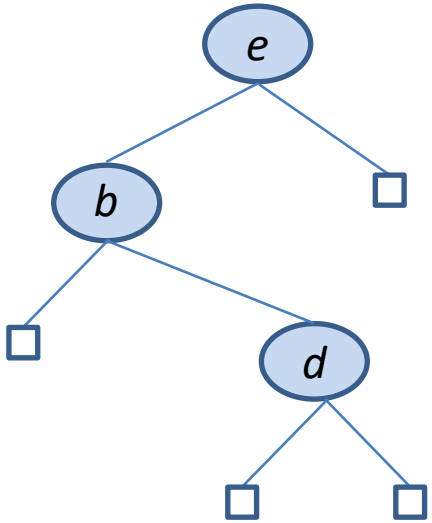
After Inserting *e*



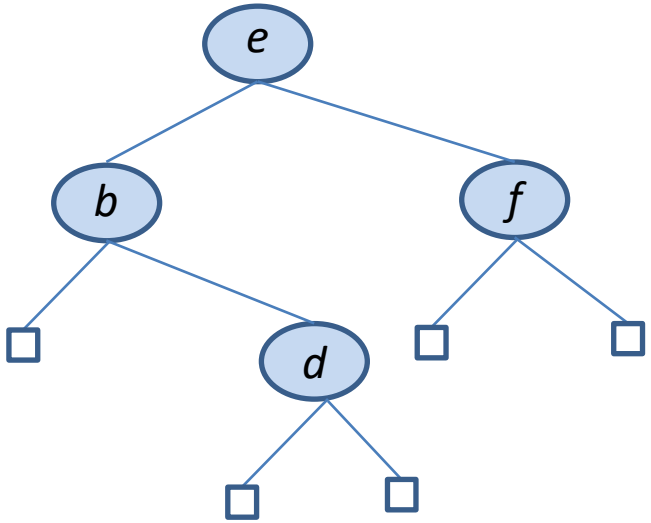
After Inserting *b*



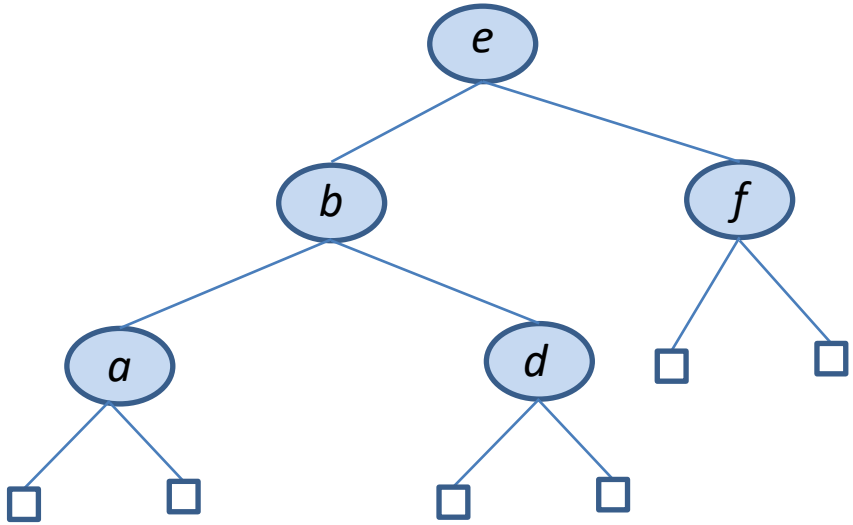
After Inserting *d*



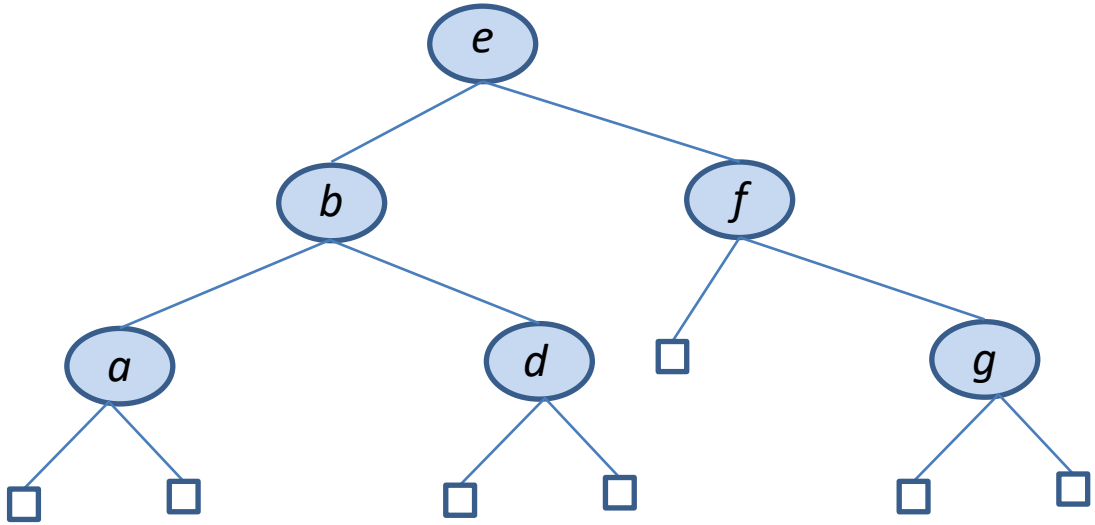
After Inserting *f*



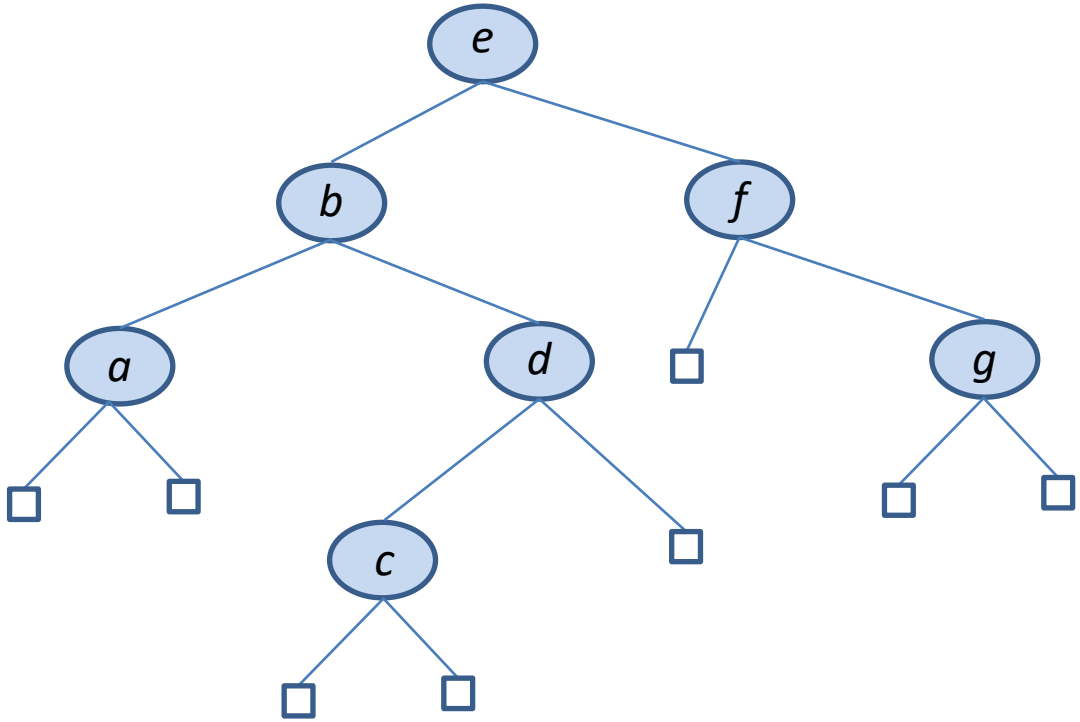
After Inserting *a*



After Inserting *g*



After Inserting *c*

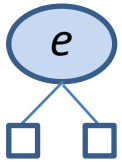


Example

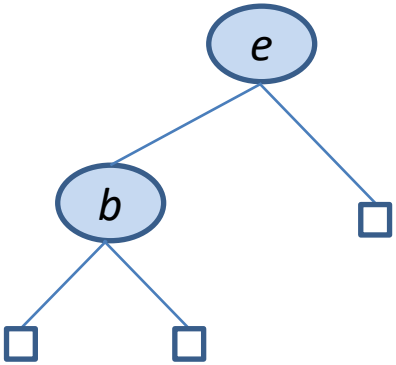
- Let us now insert the keys *e, b, d, e, f, e* in an empty tree.
- Now we have duplicate keys.

Initial Empty Tree

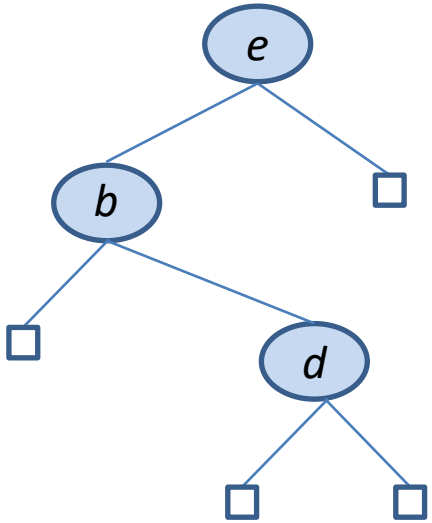
After Inserting *e*



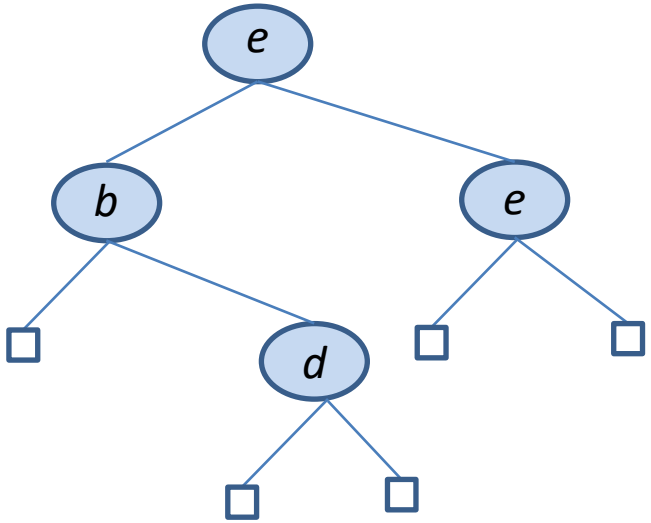
After Inserting *b*



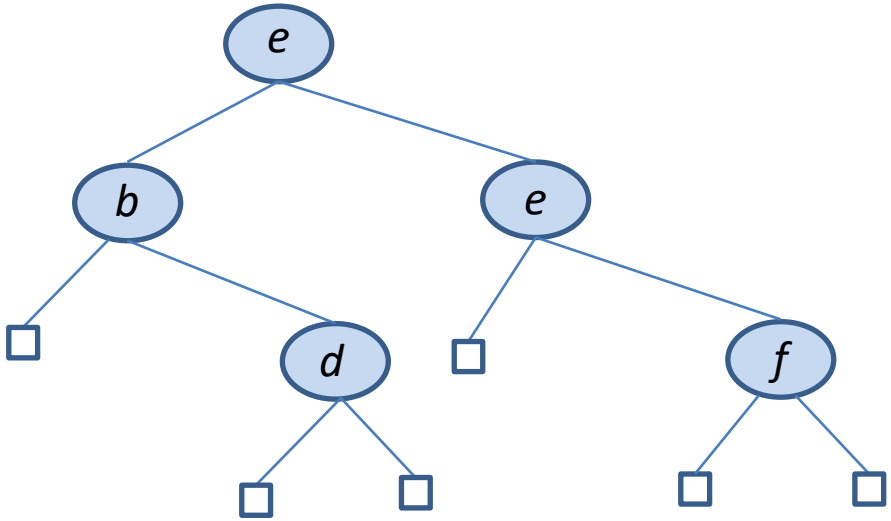
After Inserting *d*



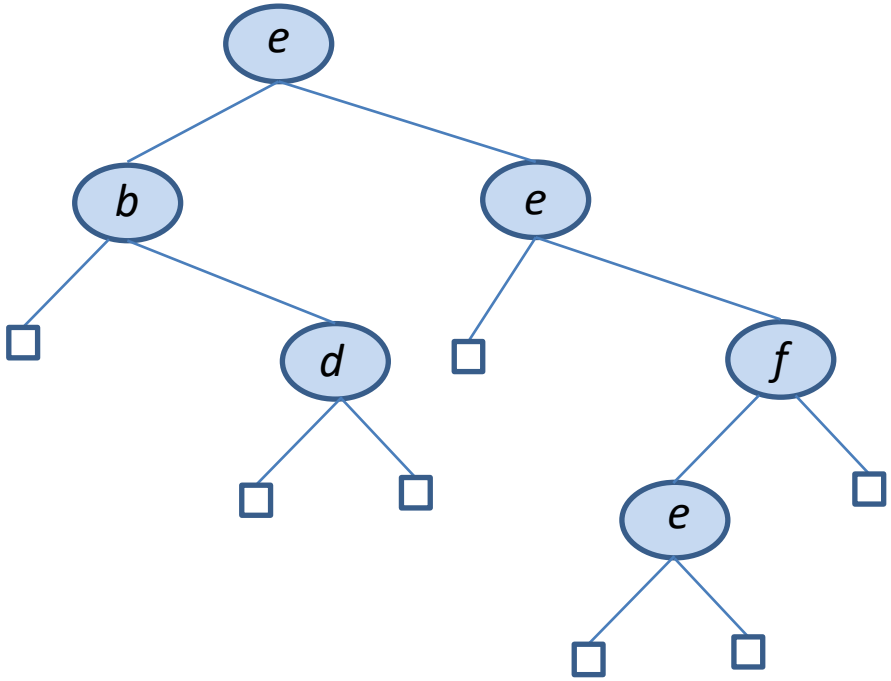
After Inserting *e*



After Inserting *f*



After Inserting *e*



Comments

- We see that in the case of duplicate keys, these keys appear **scattered throughout the BST.**
- However, duplicate keys do all appear on the appropriate search path for the key from the root to the external node, so they can be found by the search algorithm.
- The search algorithm implemented by `STsearch` will return the item corresponding to the **first** duplicate key in this path.

The Function STinsert

```
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if less(v, t)
         h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    (h->N)++; return h;
  }

void STinsert(Item item)
  { head = insertR(head, item); }
```

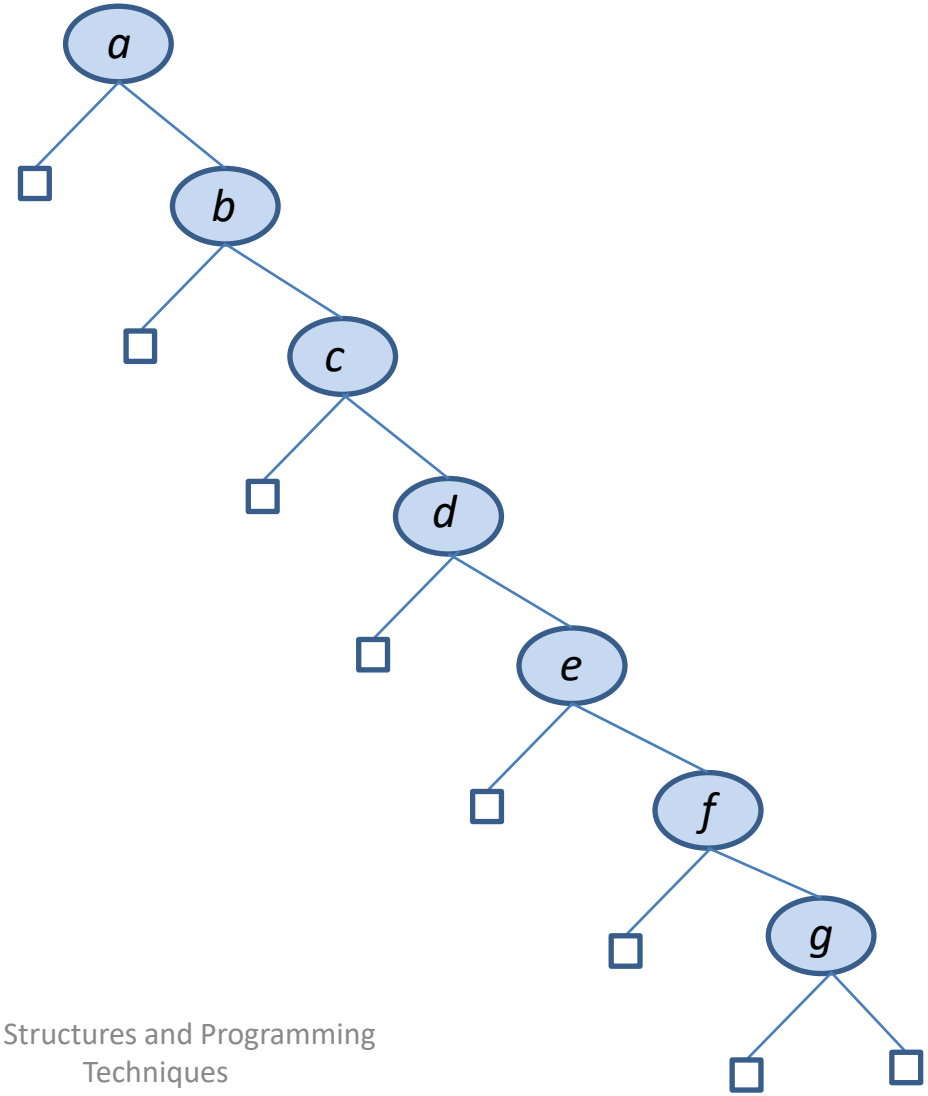
Notes

- The function `STinsert` calls the recursive function `insertR` which does the work.

Inserting in the Natural Order

- Let us now revisit the previous example and insert the same keys in their **natural order** *a, b, c, d, e, f, g*.
- Then the tree constructed is a **chain** (αλυσίδα).

Insert *a, b, c, d, e, f, g*

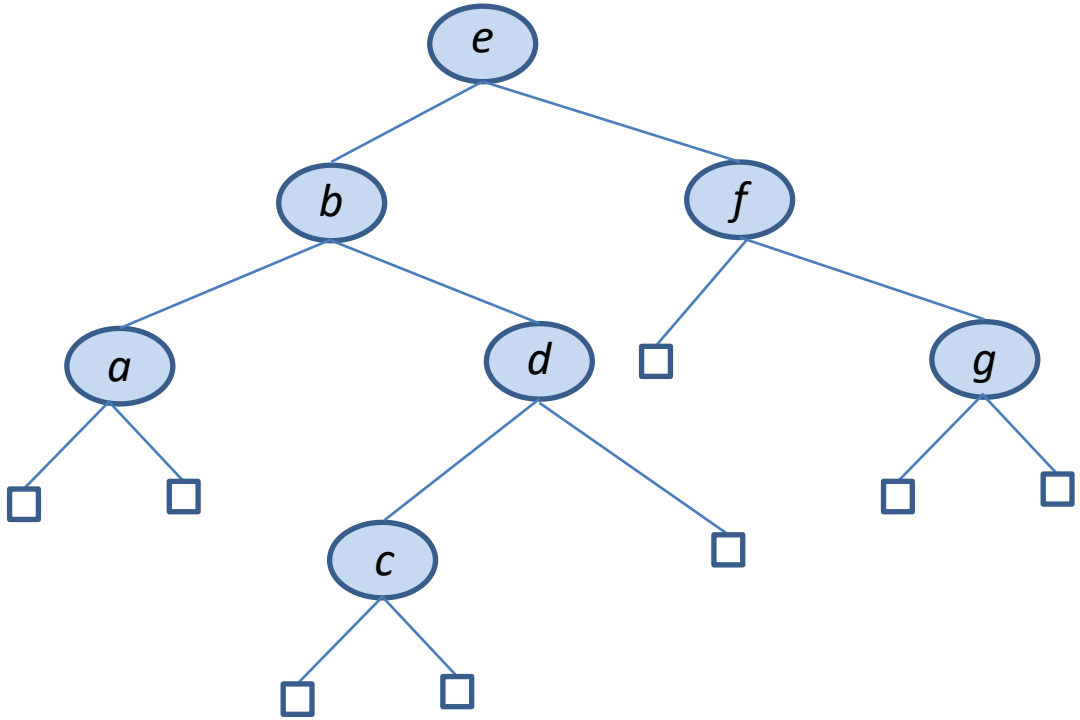


Inserting in the Natural Order (cont'd)

- As we will see below, **chains result in inefficient searching**. So, we should never insert keys in their natural order in a BST.

- Similar things hold if keys are in **reverse order** or if they are nearly ordered.

Example



Let us traverse this BST in **inorder**. What do you notice?

Inorder Traversal of BSTs

- If we traverse a BST using the inorder traversal, then the keys of the nodes come out **sorted** in their natural order.
- This gives rise to a sorting algorithm called **TreeSort**: insert the keys one by one in a BST, then traverse the tree inorder.
- The function `STsort` shown in the next slide implements traverses a tree in inorder.

The Function `STsort`

```
void sortR(link h, void (*visit)(Item))
   {
        if (h == z) return;
        sortR(h->l, visit);
        visit(h->item);
        sortR(h->r, visit);
   }

void STsort(void (*visit)(Item))
   { sortR(head, visit); }
```

Notes

- The second parameter of `sortR` is `visit` which is a **pointer to a function** with return type `void` declared by

```
void (*visit)(Item).
```

- `sortR` can then be called with **the name of a function as second argument**; this is the function that we want to apply to each node of the tree when we visit it.

Example of a Symbol Table Client

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "ST.h"

void main(int argc, char *argv[])
 { int N, maxN = atoi(argv[1]), sw = atoi(argv[2]);
    Key v; Item item;
    STinit(maxN);
    for (N = 0; N < maxN; N++)
      {
        if (sw) v = ITEMrand();
          else if (ITEMscan(&v) == EOF) break;
        if (STsearch(v) != NULLitem) continue;
        key(item) = v;
        printf("Inserting item %d\n", item);
        STinsert(item);
      }
    STsort(ITEMshow); printf("\n");
    printf("%d keys\n", N);
    printf("%d distinct keys\n", STcount());
 }
```

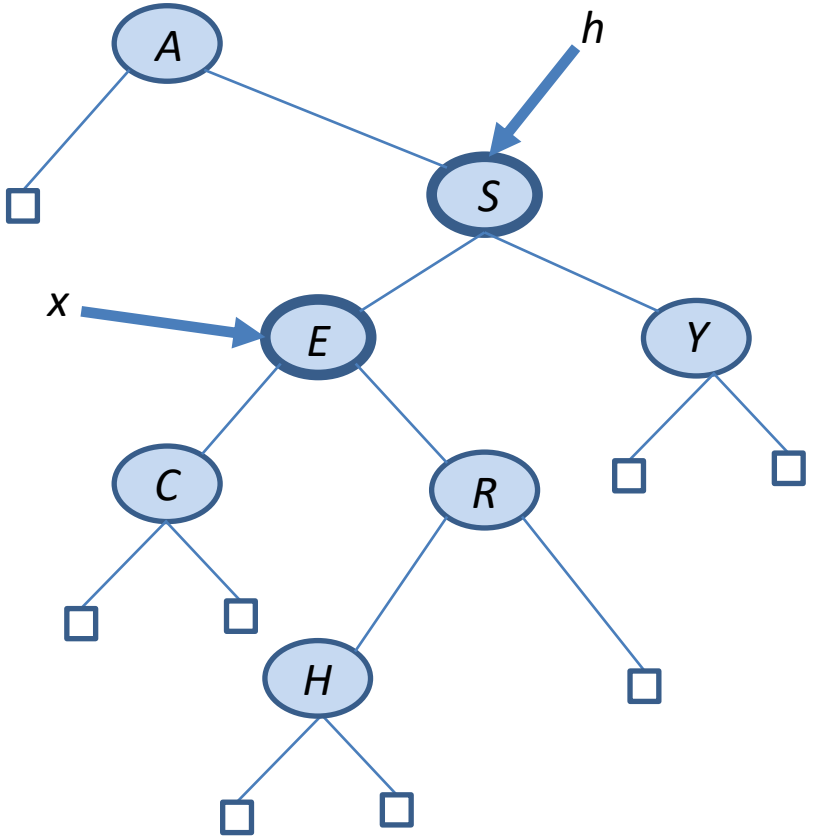
Notes

- The previous client program generates randomly or reads from the standard input at most `MaxN` integers, inserts them in a BST one by one and, then, prints them in sorted order using the inorder traversal of the tree.
- It also prints the number of keys given and the number of distinct keys encountered.

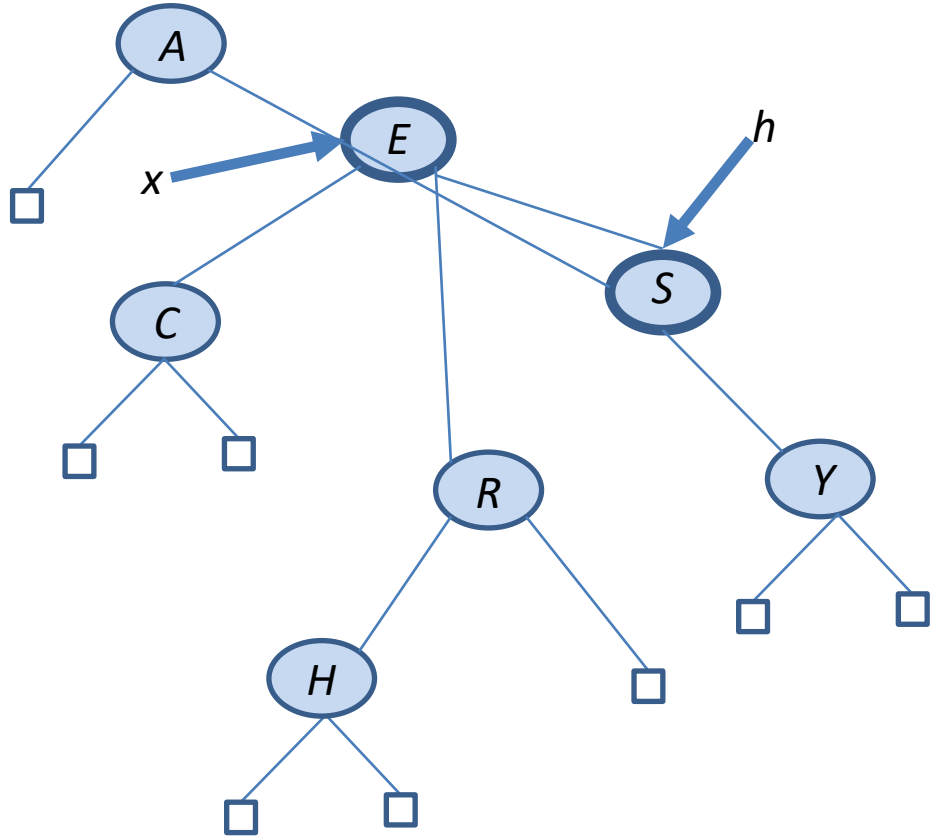
Rotations

- **Rotation (περιστροφή)** is a fundamental operation on trees.
- Rotation allows us to **interchange the role of the root node of the rotation and one of the root's children** in a tree while still **preserving the BST ordering** among the keys in the nodes.
- Rotation is important because it helps us make our trees **balanced** as we will see in forthcoming lectures.
- We will define **right rotation** and **left rotation**.

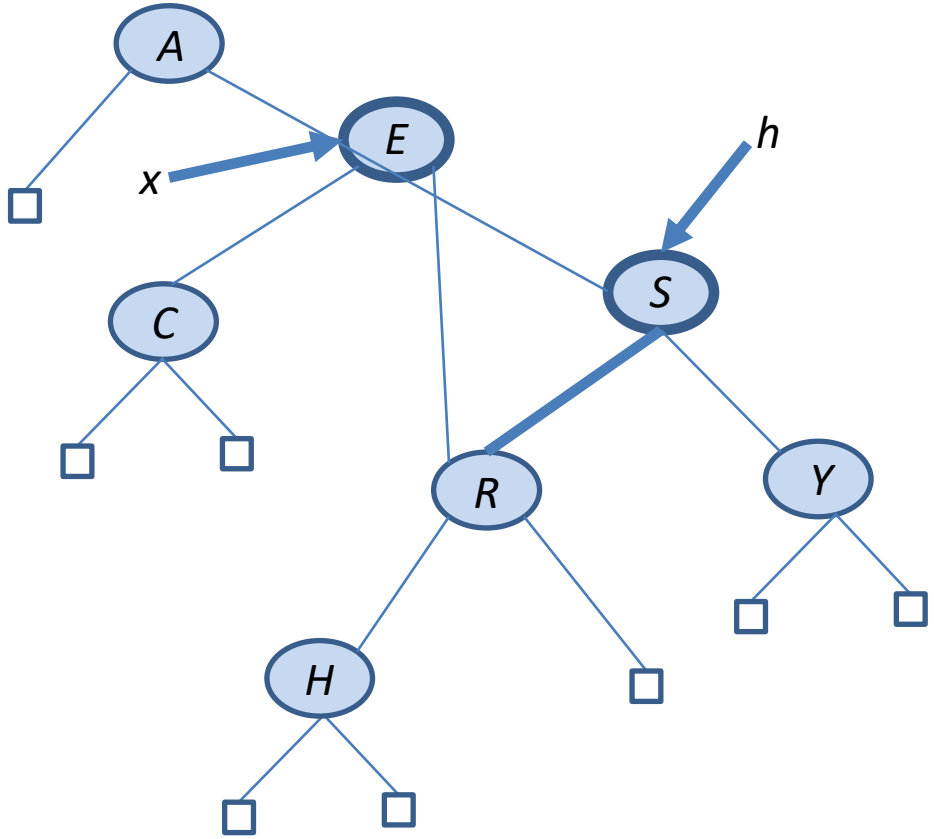
Example: Right Rotation at Node *S*



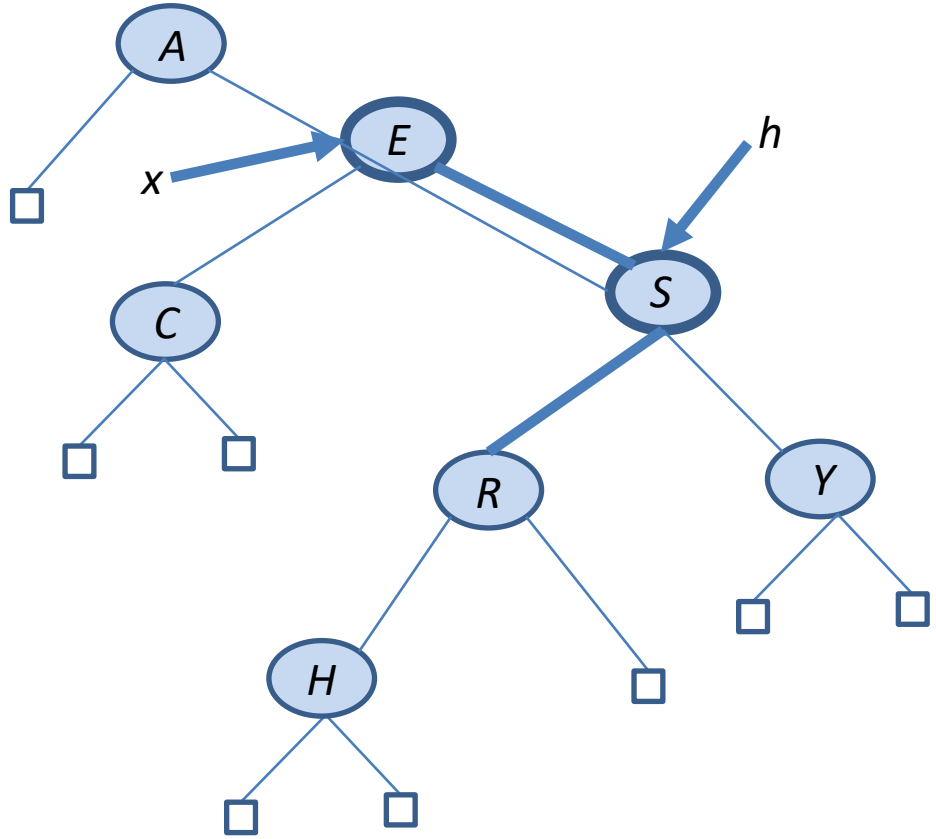
Example: Right Rotation at Node *S*



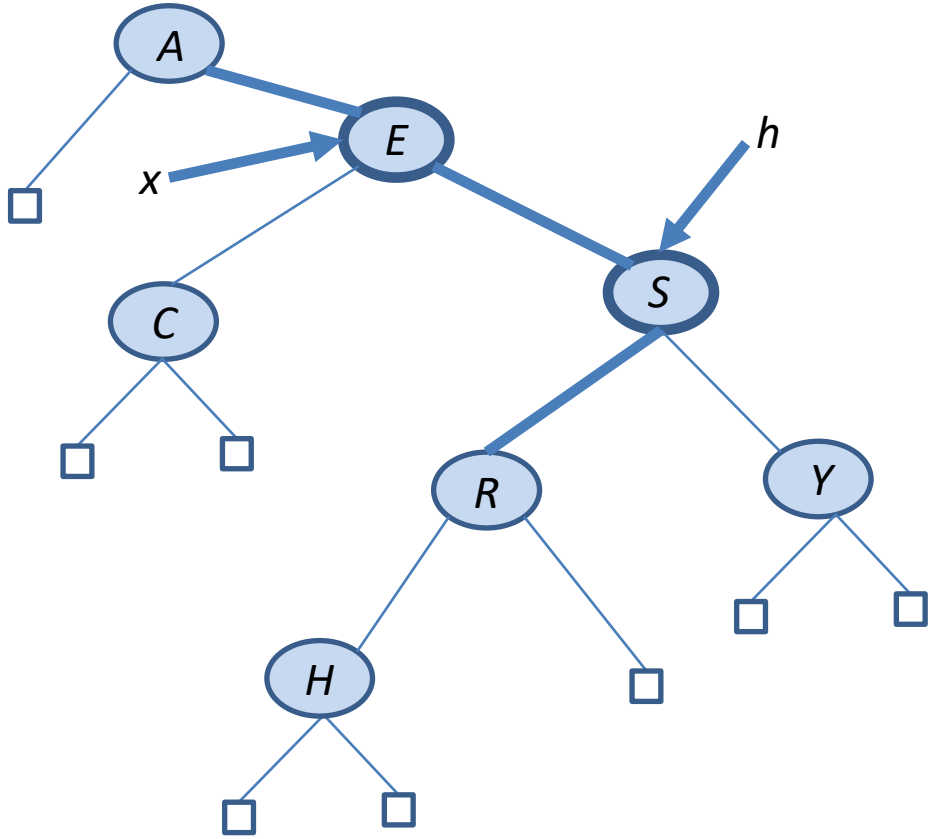
Example: Right Rotation at Node *S*



Example: Right Rotation at Node *S*



Example: Right Rotation at Node *S*



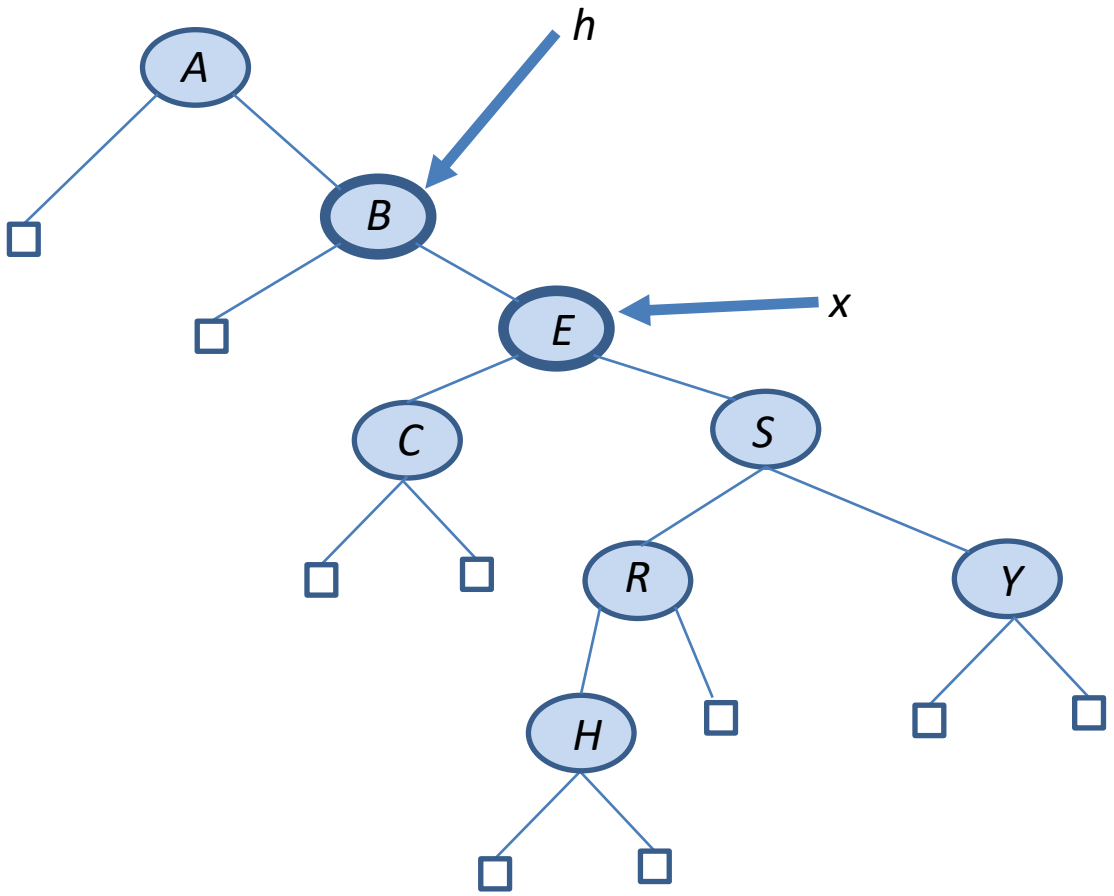
Right Rotation

- A **right rotation** involves the **root** node of the rotation and its **left child**.
- The rotation puts the root on the right, essentially reversing the direction of the left link of the root.
- Before the rotation, the left link of the root points from the root to the left child; after the rotation, it points from the old left child (the new root) to the old root (the right child of the new root).
- The tricky part, which makes the rotation work, is to copy the right link of the left child to be the left link of the old root. This link points to all the nodes with keys **between** the two nodes involved in the rotation.
- Finally, the link to the old root has to be changed to point to the new root.

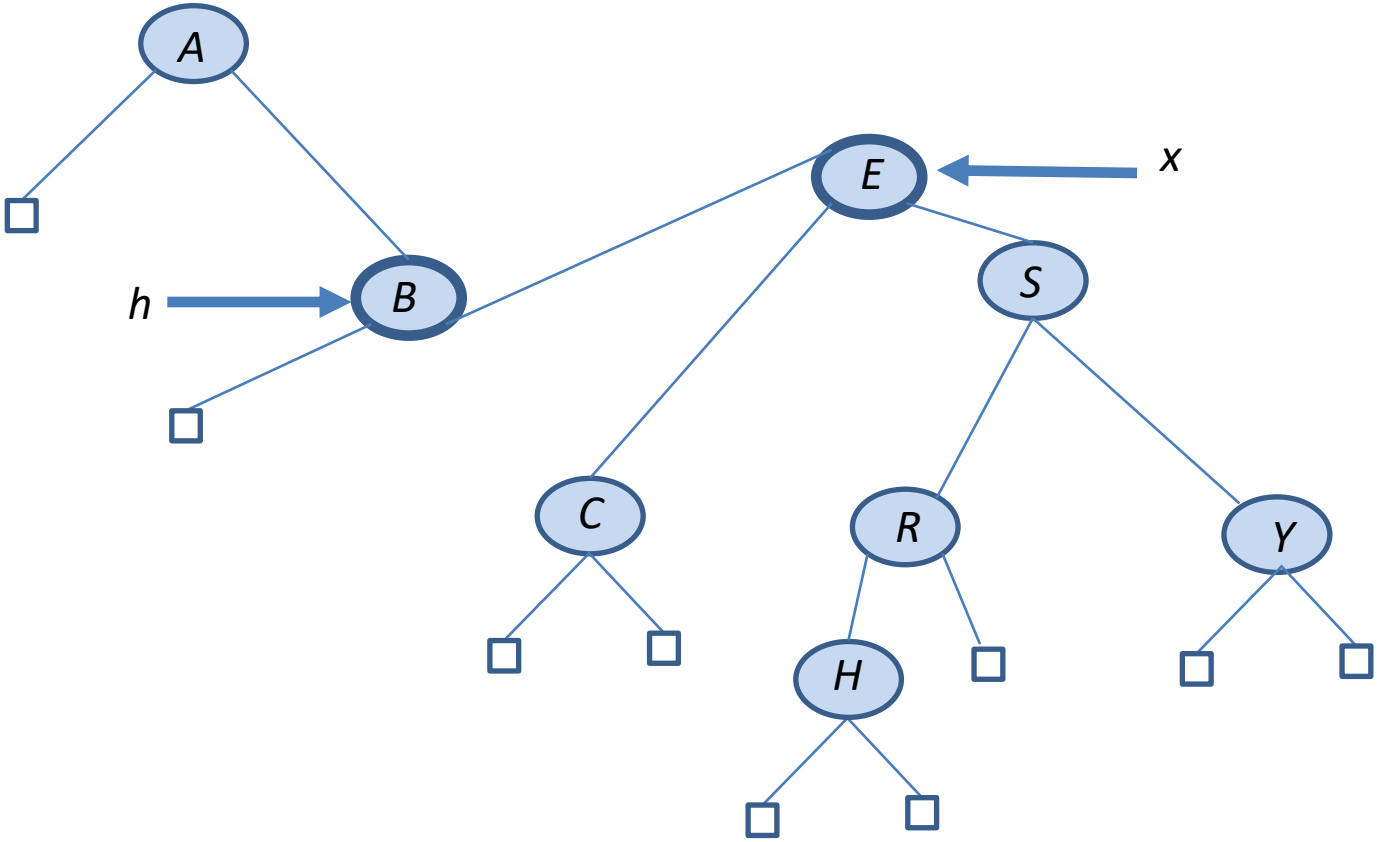
Left Rotation

- A **left rotation** involves the **root** node of the rotation and its **right child.**
- The description of left rotation is identical to the description of the right rotation with “right” and “left” interchanged everywhere.

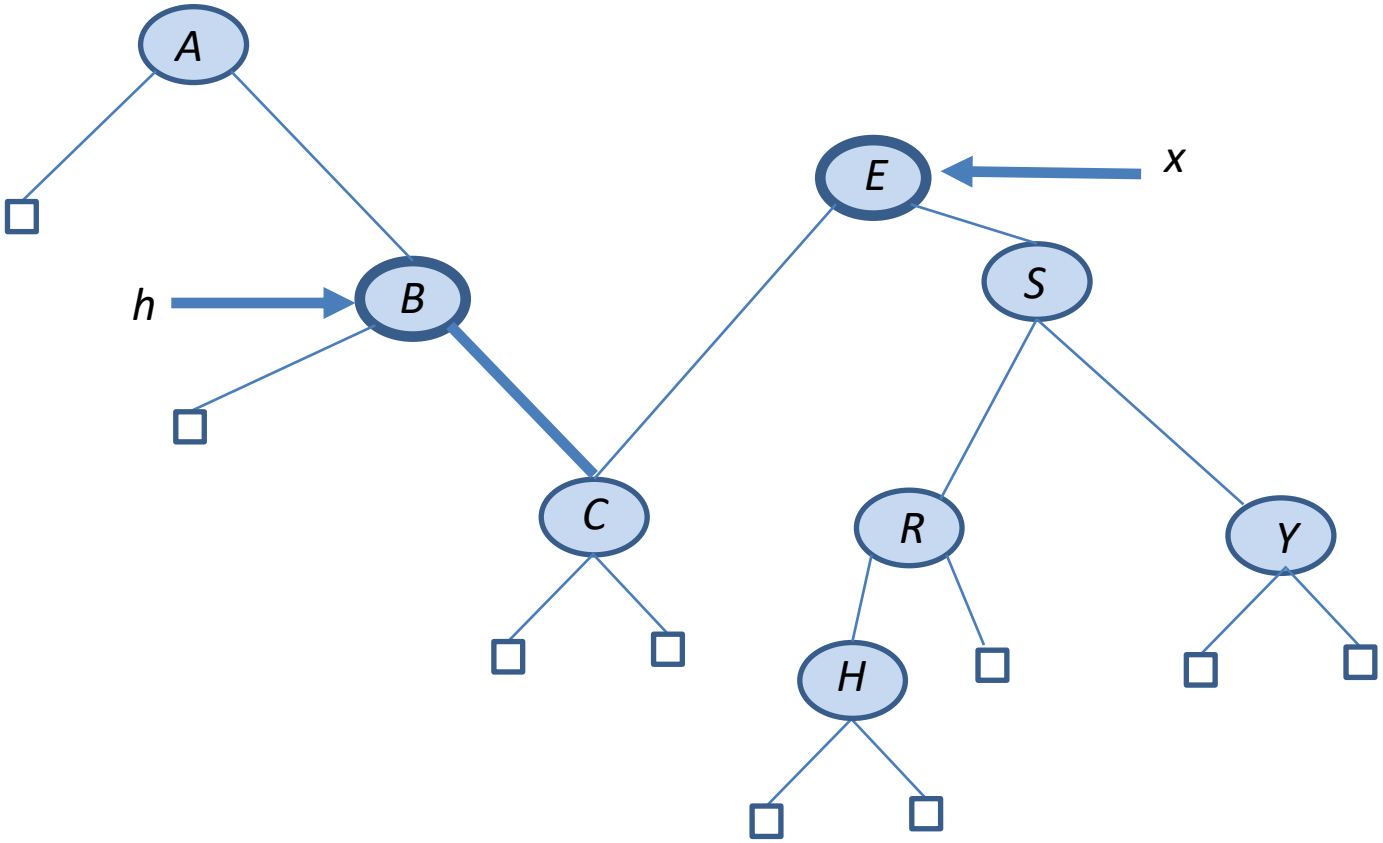
Example: Left Rotation at Node *B*



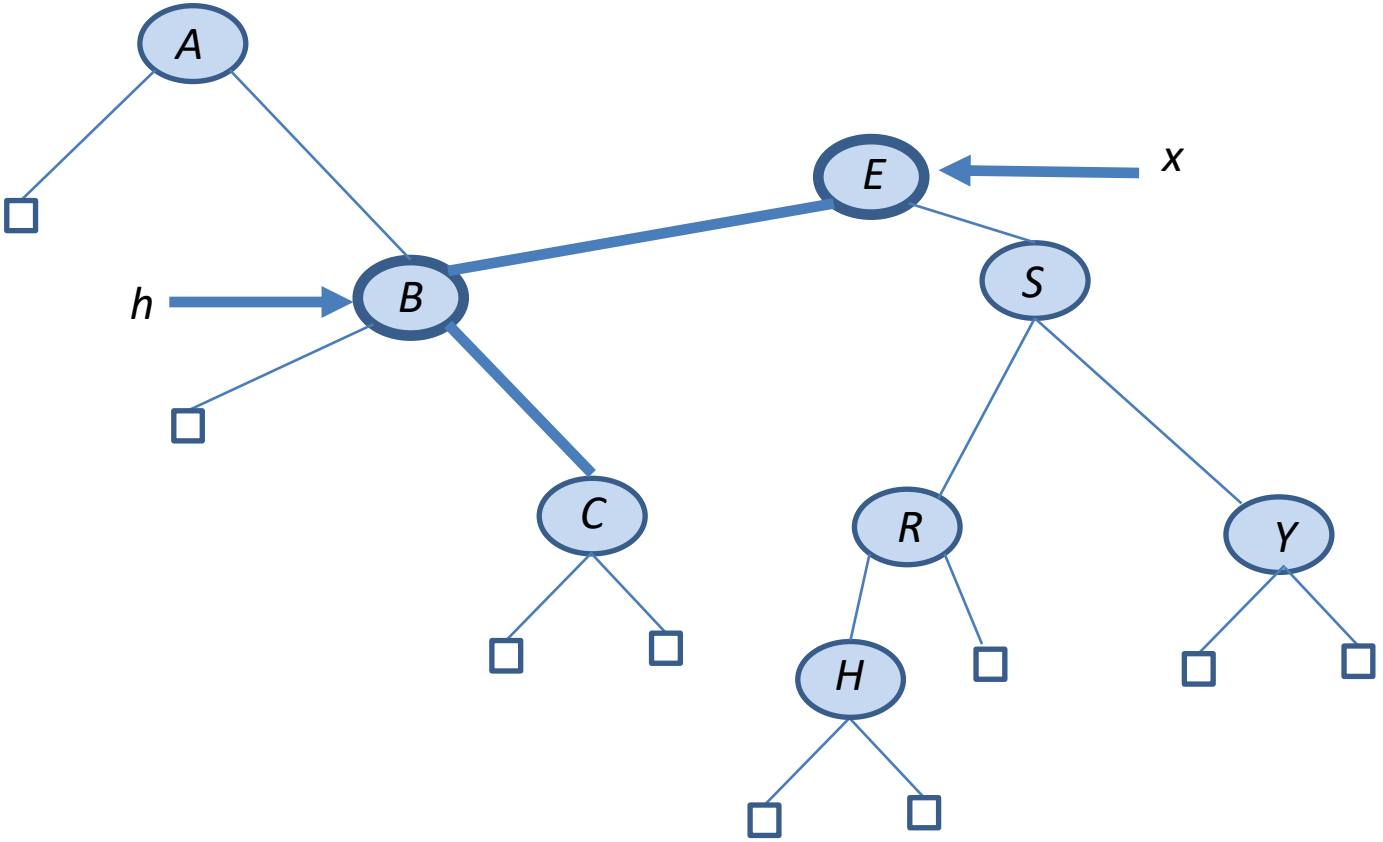
Example: Left Rotation at Node *B*



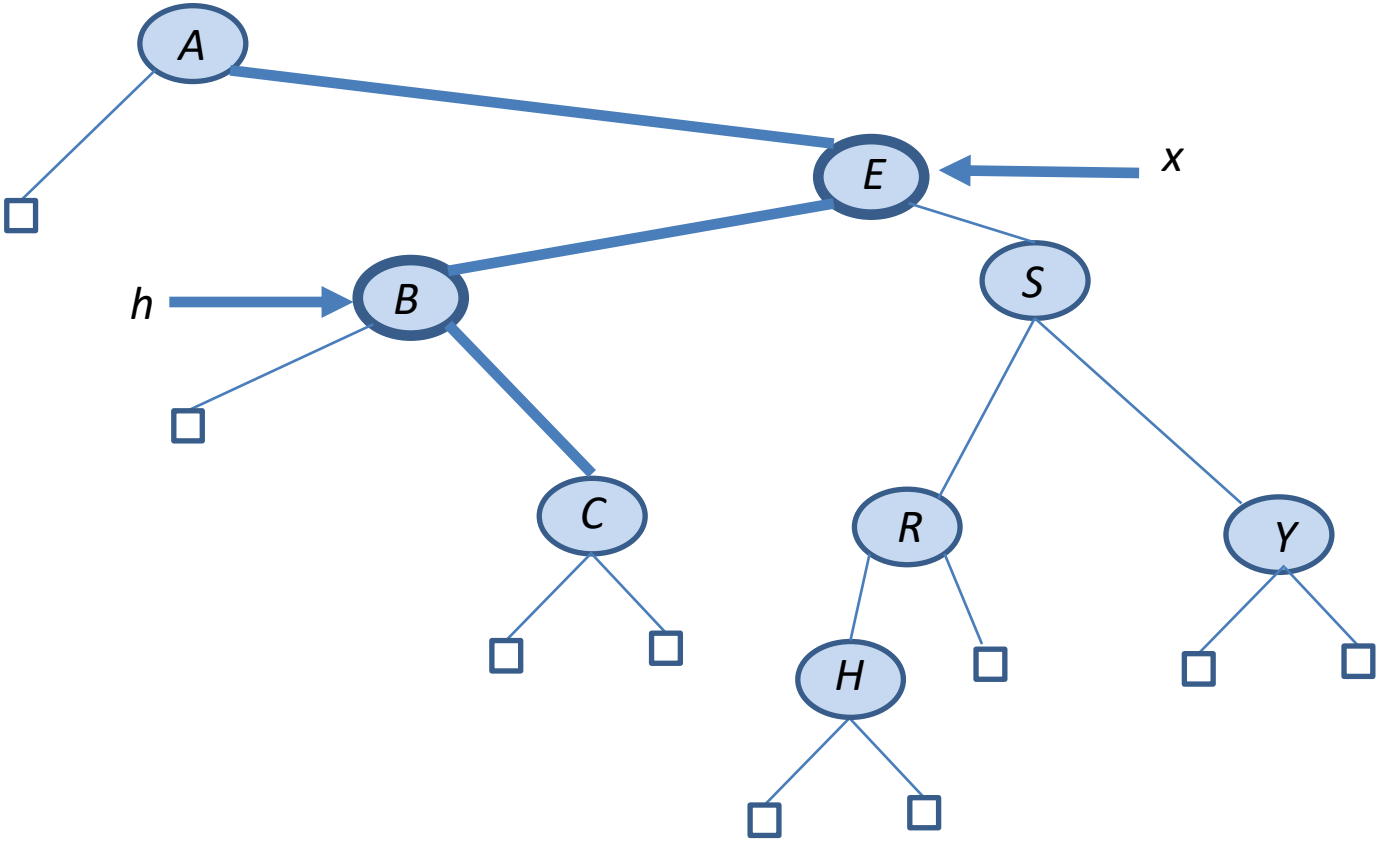
Example: Left Rotation at Node *B*



Example: Left Rotation at Node *B*



Example: Left Rotation at Node *B*



Functions for Rotation

```
link rotR(link h)
 {
    link x = h->l; h->l = x->r;
    x->r = h; return x;
 }

link rotL(link h)
 {
    link x = h->r; h->r = x->l;
    x->l = h; return x;
 }
```

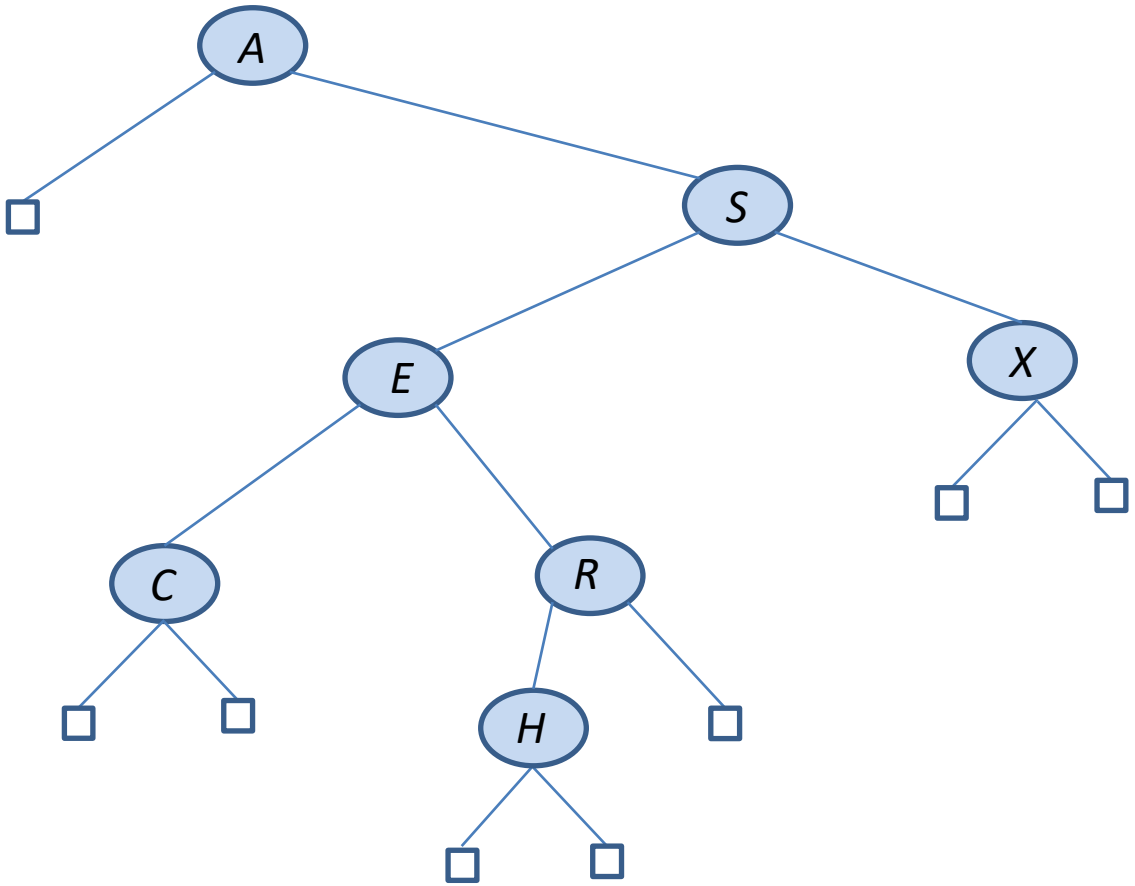
Insertion at the Root

- In the implementation of insertion presented earlier, every new node inserted goes somewhere at the bottom of the tree.
- We can consider an alternative insertion method where we insist that **each new node is inserted at the root.**
- In practice, the advantage of the root insertion method is that recently inserted keys are near the top. This can be useful in some applications.

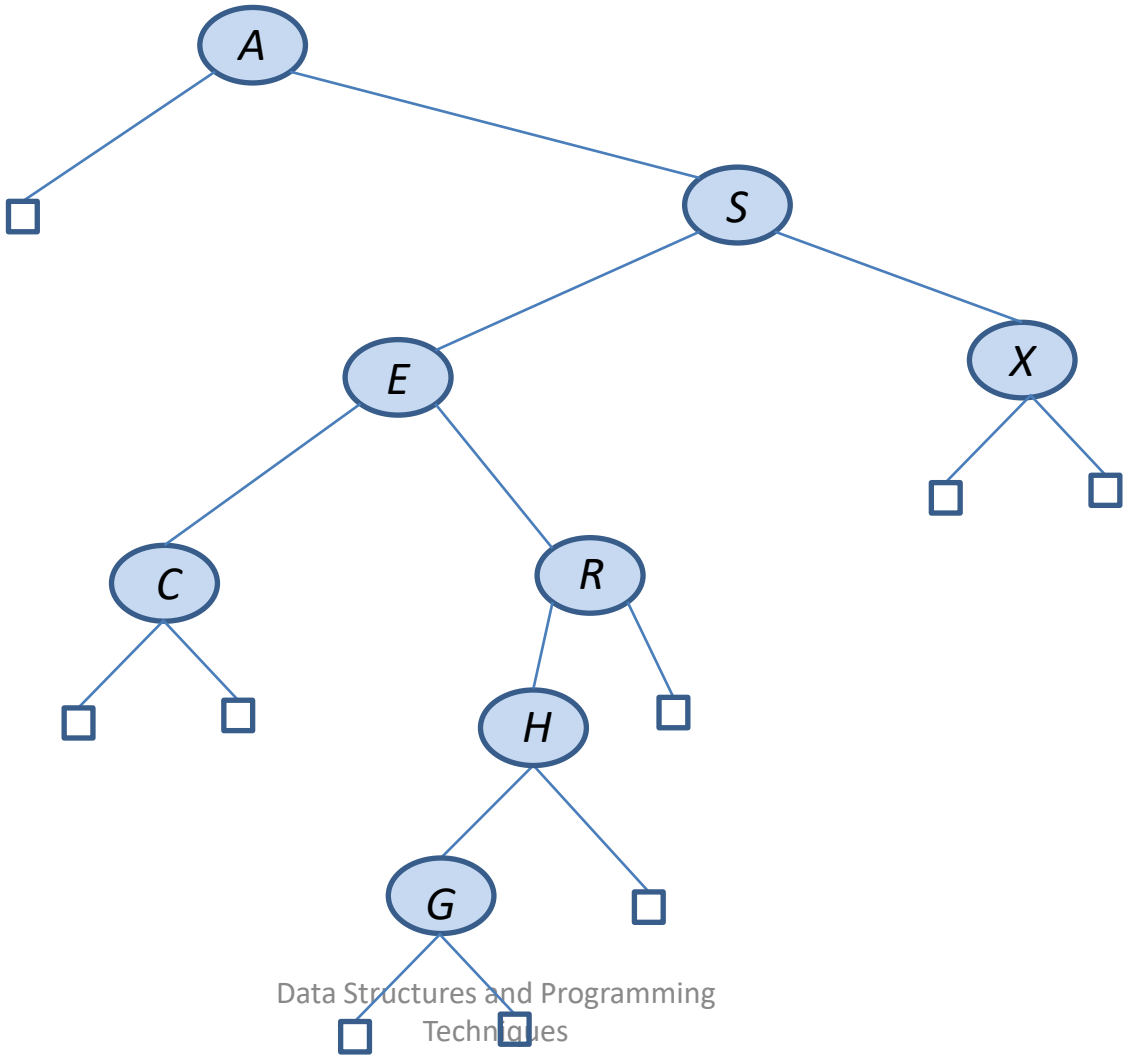
Insertion at the Root (cont'd)

- The rotation operations provide **a straightforward implementation of root insertion:**
  - Recursively insert the new item into the appropriate subtree, leaving it, when the recursion is complete, at the root of that tree.
  - Then, rotate repeatedly to make it the root of the tree.

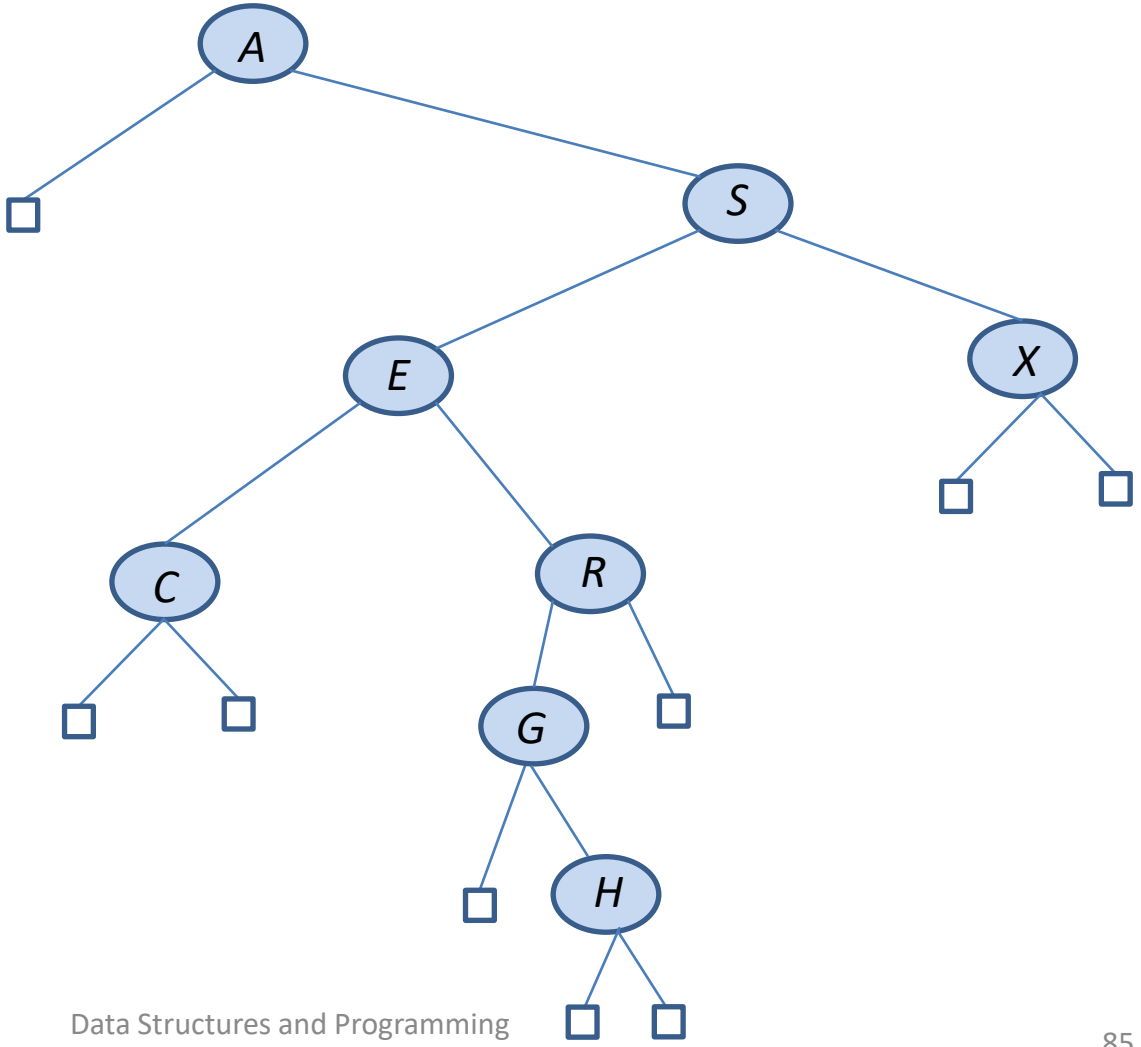
Example: Root Insertion of *G*



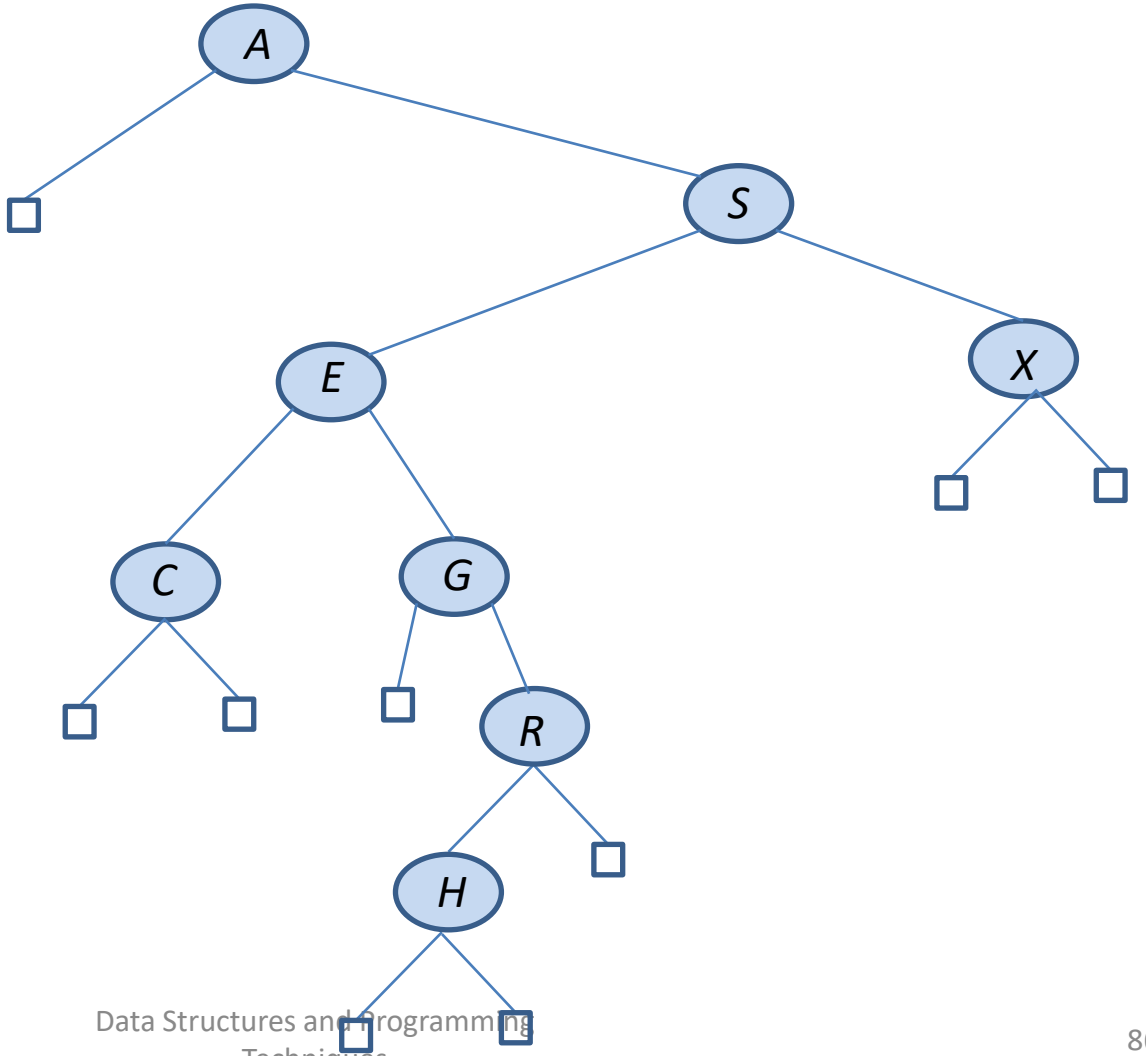
Example: *G* is inserted



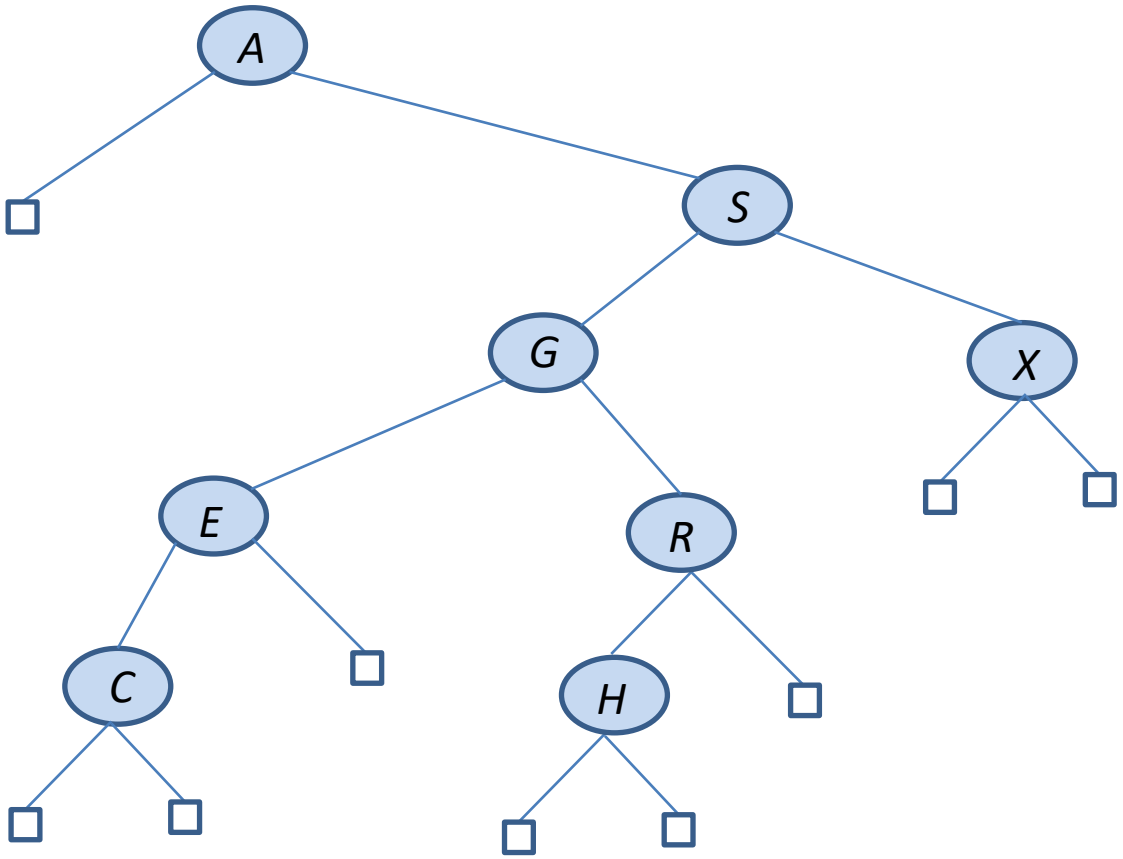
Example: after right rotation at *H*



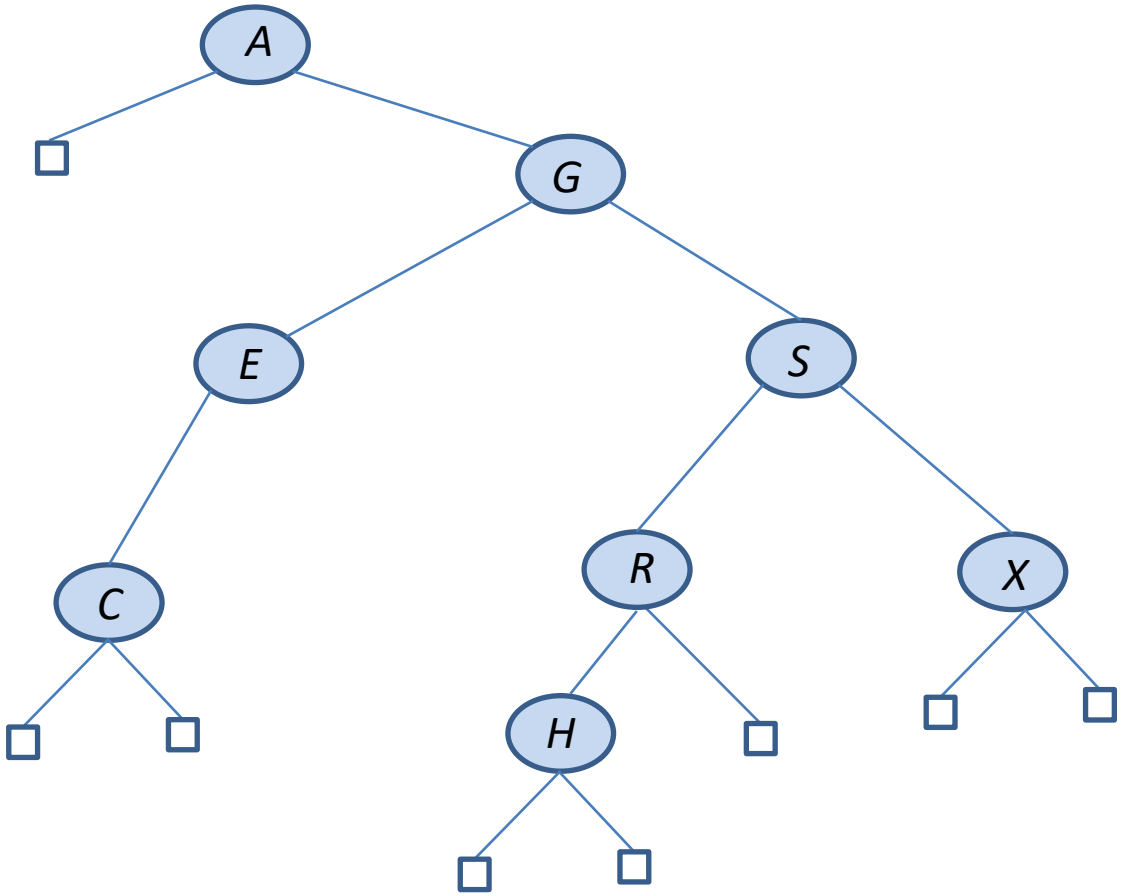
Example: after right rotation at *R*



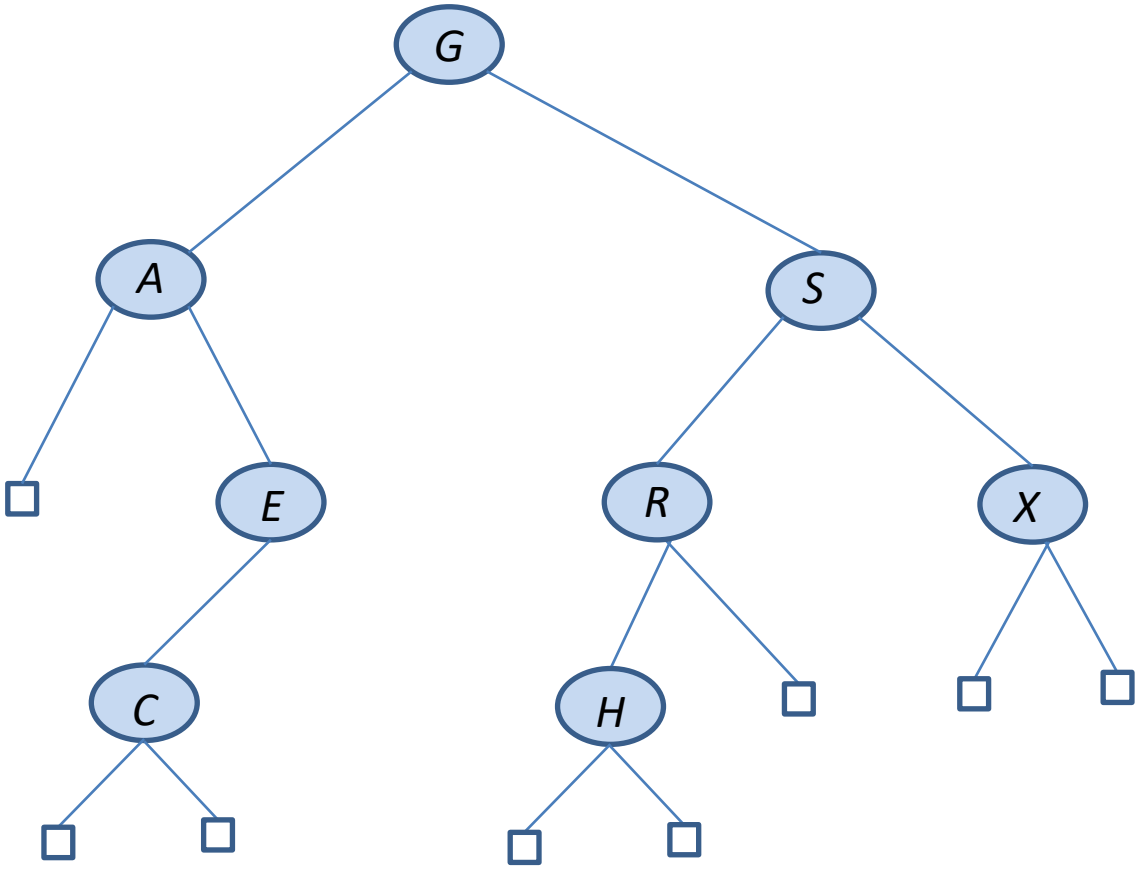
Example: after left rotation at *E*



Example: after right rotation at *S*



Example: after left rotation at *A*



Functions for Root Insertion

```
link insertT(link h, Item item)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (less(v, key(h->item)))
      { h->l = insertT(h->l, item); h = rotR(h); }
    else
      { h->r = insertT(h->r, item); h = rotL(h); }
    return h;
  }

void STinsert(Item item)
  { head = insertT(head, item); }
```

Selecting the k-th Smallest Key

- To implement select, we use a recursive procedure.
- We use **zero-based indexing.** So, when $k = 3$, we will be looking for the item with the fourth smallest key.
- To find the item with the k-the smallest key, we check the number of nodes in the left subtree.
- If there are k nodes there, we return the item at the root.
- If the left subtree has more than k nodes, we recursively look for the item with the k-th smallest key there.
- If none of the above conditions holds then the left subtree has t items with $t < k$, and the item with the k-th smallest key is the item with the $(k - t - 1)$-th smallest key in the right subtree.

Functions for Selection

```
Item selectR(link h, int k)
  {
    int t;
    if (h == z) return NULLitem;
    t = (h->l == z) ? 0 : h->l->N;
    if (t > k) return selectR(h->l, k);
    if (t < k) return selectR(h->r, k-t-1);
    return h->item;
  }

Item STselect(int k)
  { return selectR(head, k); }
```

Partition

- We can change the implementation of the select operation into a **partition** operation (διαμέριση), which rearranges the tree to **put the item with the k-th smallest key at the root.**
- If we recursively put the desired node at the root of one of the subtrees, we can then make it the root of the whole tree with a **rotation**.

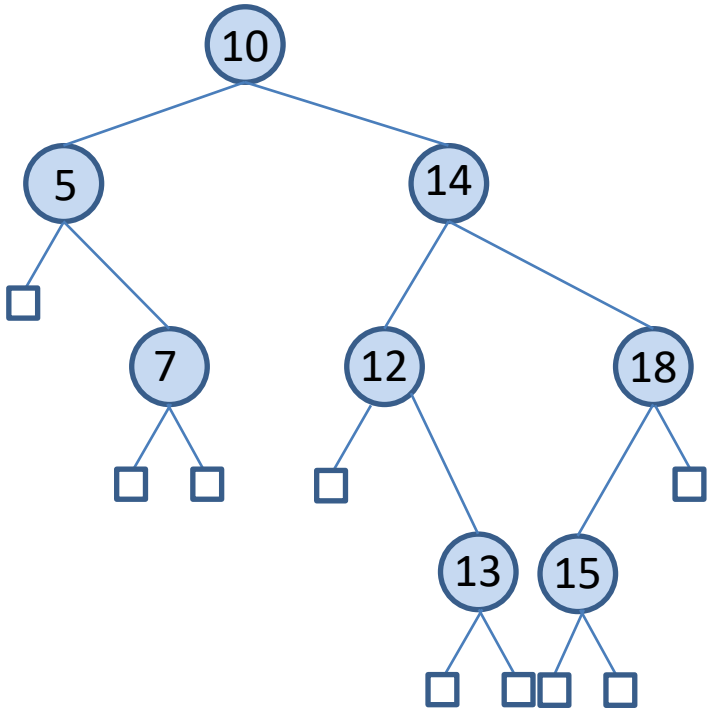
Functions for Partition

```
link partR(link h, int k)
  {
    int t = h->l->N;
    if (t > k )
      { h->l = partR(h->l, k);
        h = rotR(h); }
    if (t < k )
      { h->r = partR(h->r, k-t-1);
        h = rotL(h); }

    return h;
  }
```

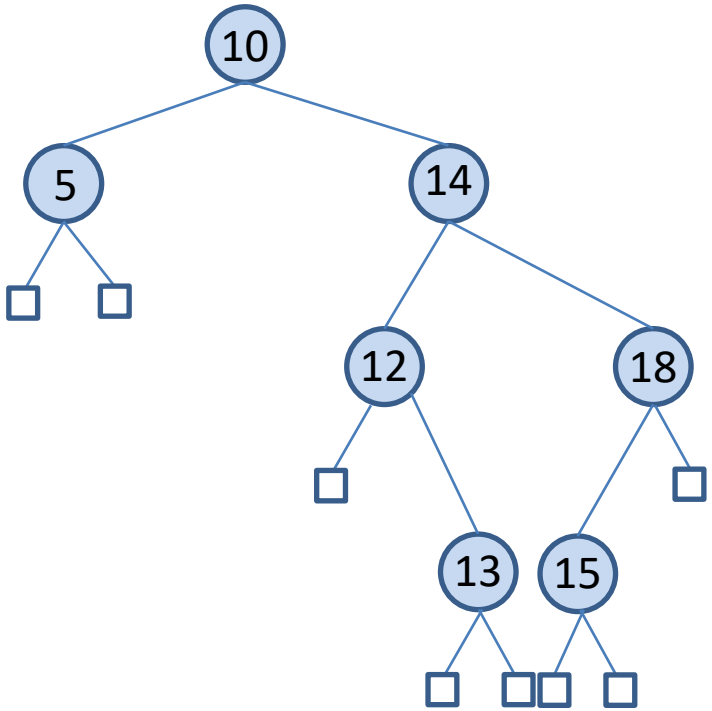
Deletion from a BST

- An algorithm for **deleting a key *K*** from a BST *T* is the following.
- If *K* is in an internal node with only (dummy) external nodes as children, then delete it and replace the link to the deleted node by a link to a new (dummy) external node.

Example: Delete 7



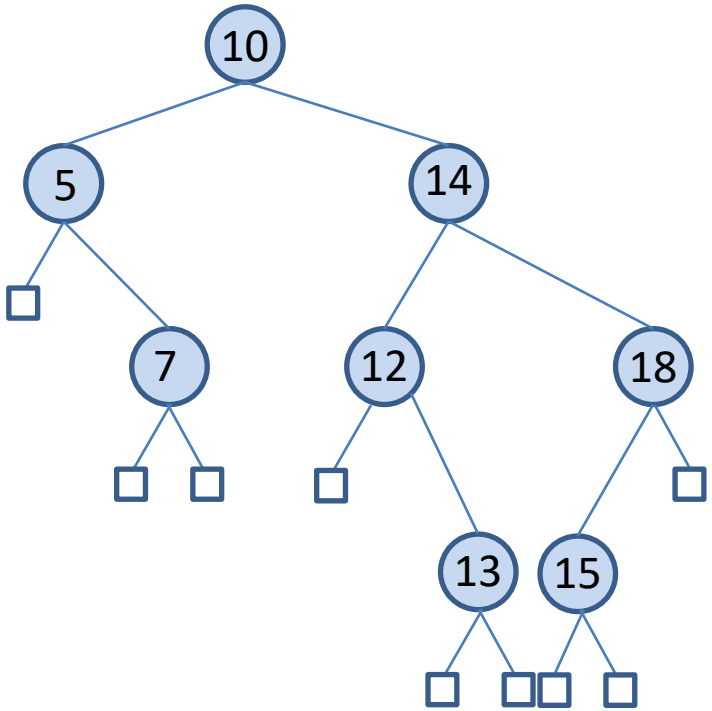
Result



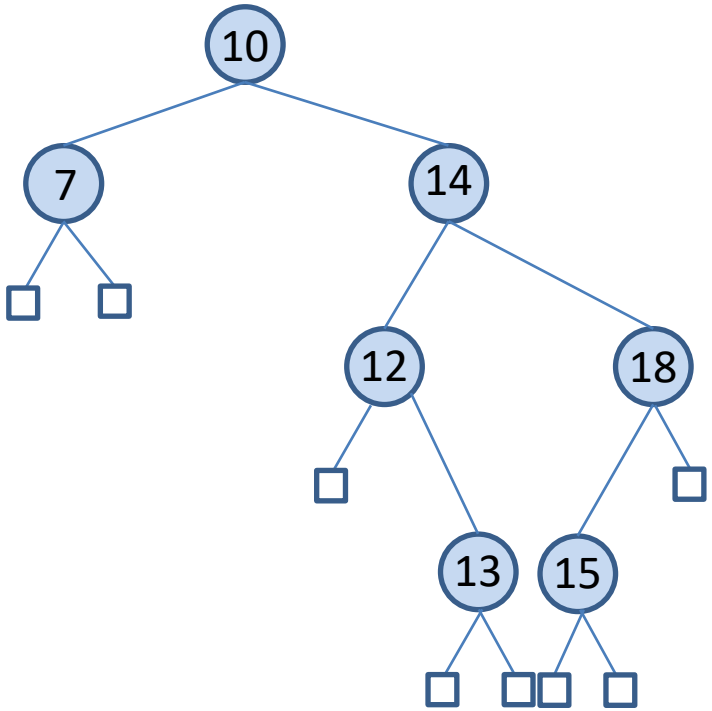
Deletion from a BST (cont'd)

- If *K* is an internal node with a single (dummy) external node as its child, then delete it and adjust the link from its parent to point to its non-empty subtree.

Example: Delete 5



Result



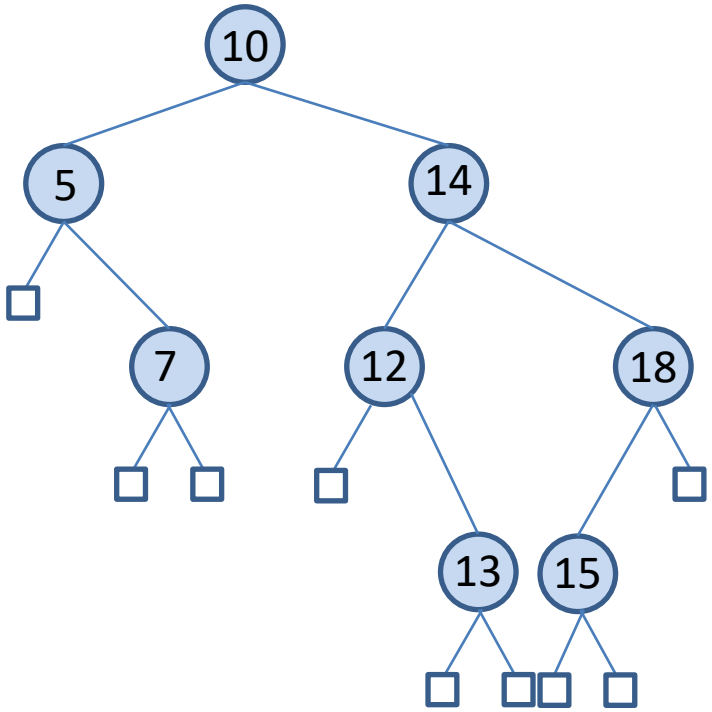
Deletion from a BST (cont'd)

- If *K* is in a node *N* with both a left and a right non-empty subtree then delete it and combine the two subtrees into one tree. But how?

Deletion from a BST (cont'd)

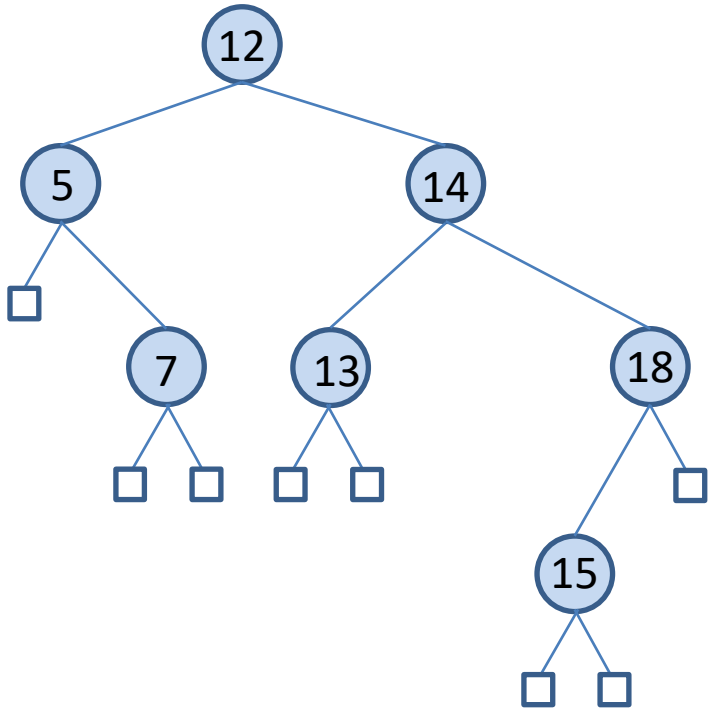
- Various options are available.
- One simple option is to find the **lowest-valued key** *K'* in the descendants of the right child and replace *K* by *K'*.
- This key is in a node *N'* which is the **successor of** *N* under the **inorder traversal** of the tree.
- This node can be found by starting at the right child of *N* and then following left child pointers until we find a node with a left child which is an external node.
- Of course, we also need to remove node *N'* from the tree. This can be done easily since this node has at most one (right) child.

Example: Delete 10



The lowest-valued key among the descendants of 14 is 12. This key will replace 10 in the tree and its current node will be removed.

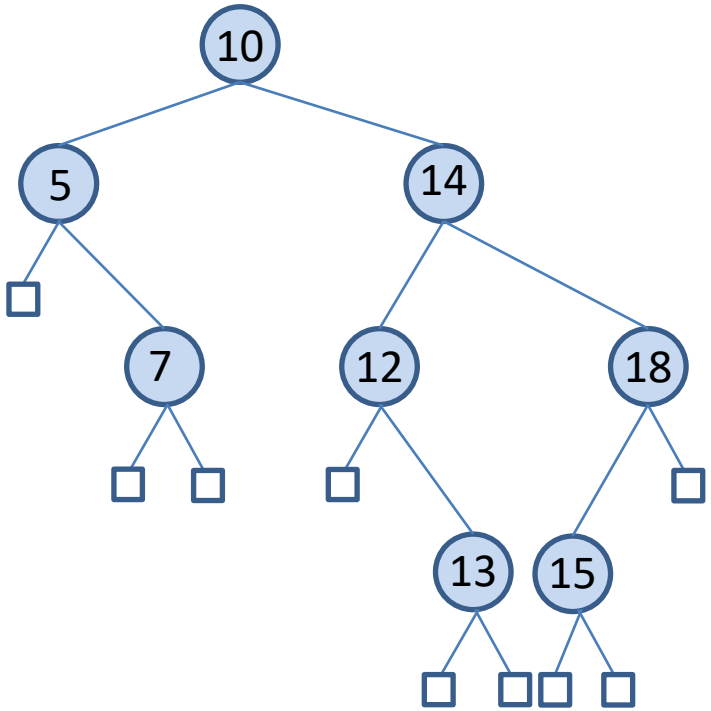
Result



Deletion from a BST (cont'd)

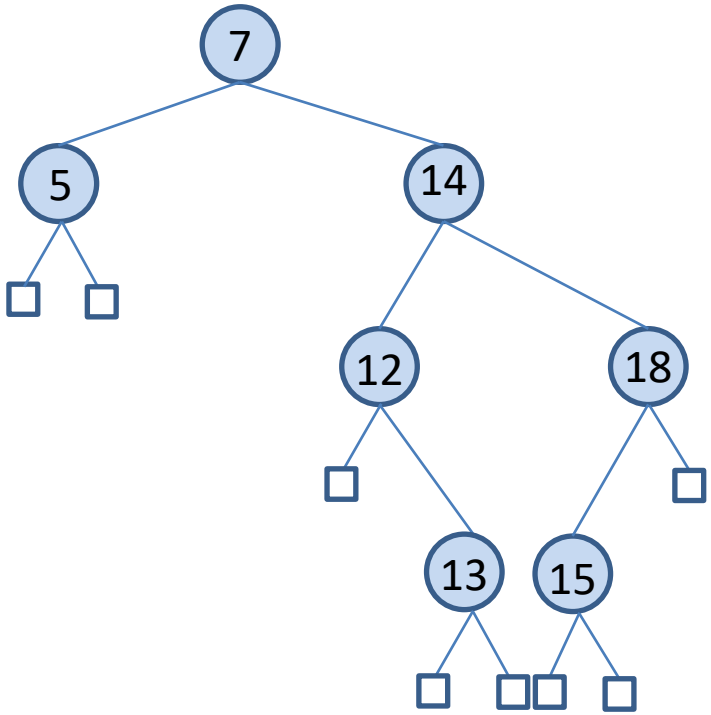
- Notice that the **highest-valued key** among the descendants of the left child would do as well.
- This key is in a node *N''* which is the **predecessor** of *N* under the **inorder traversal** of the tree.

Example: Delete 10



Alternatively, we can replace 10 with the highest-valued key among the descendants of its left child. This is the key 7.

Alternative Result



Functions for Deletion

```
link joinLR(link a, link b)
  {
     if (b == z) return a;
     b = partR(b, 0);
     b->l = a;
     return b;
   }

link deleteR(link h, Key v)
  {
     link x;
     Key t = key(h->item);
     if (h == z) return z;
     if (less(v, t)) h->l = deleteR(h->l, v);
     if (less(t, v)) h->r = deleteR(h->r, v);
     if (eq(v, t)) { x = h; h = joinLR(h->l, h->r); free(x); }
     return h;
   }

void STdelete(Key v)  { head = deleteR(head, v); }
```

Notes

- The function `deleteR` implements the algorithm we discussed in a recursive way.
- The function `joinLR` combines the two subtrees into a new tree by utilizing the partition function `partR` we presented earlier for making the smallest key of the right subtree as the new root.
- `joinLR` searches in the right subtree of the deleted node, finds its smallest element (call `part(b,0)`), makes this element the root of the new tree and attaches to this root, as left child, the left subtree of the deleted node (statements `b->l=a` and `return b`).

The Join Operation (Ένωση)

- Let us now present an algorithm that merges two BSTs into one BST. The algorithm is as follows.
- First, we delete the root of the first BST and insert it into the second BST using root insertion.
- This operation gives us two subtrees with keys known to be smaller than this root, and two subtrees with keys known to be larger than this root.
- We recursively combine the former pair to be the left subtree of the root and the latter to be the right subtree of the root.

Functions for Join

```
link STjoin(link a, link b)
  {
    if (b == z) return a;
    if (a == z) return b;
    b = insertT(b, a->item);
    b->l = STjoin(a->l, b->l);
    b->r = STjoin(a->r, b->r);
    free(a);
    return b;
  }
```

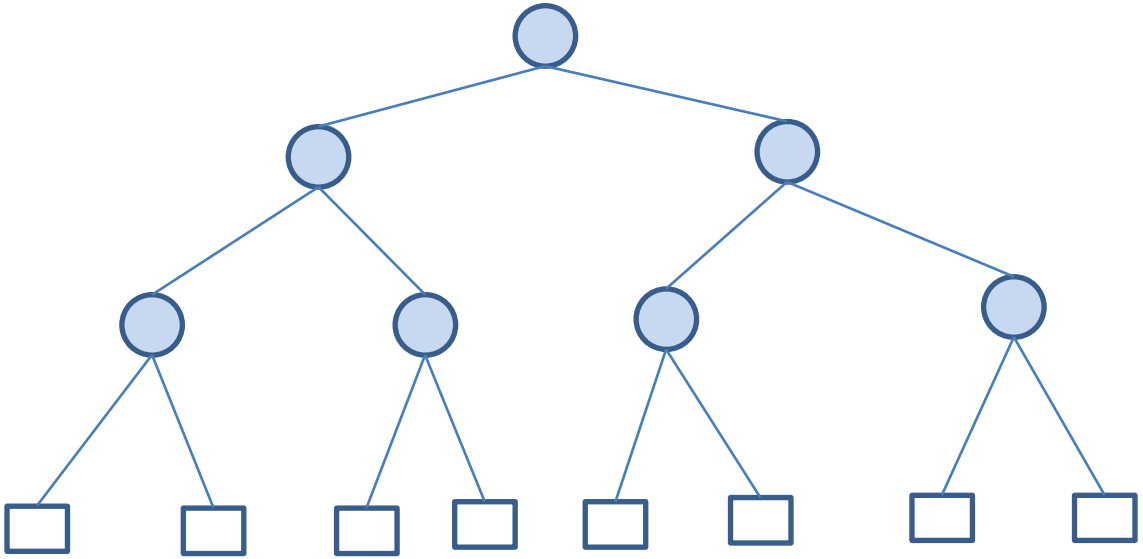
Complexity Analysis

- The algorithm for searching in a BST executes a **constant number** of primitive operations for each recursive call.
- `SearchR` is called on the nodes of a path that starts at the root and goes down one level at a time.
- Thus, **the number of such nodes is bounded by $\boldsymbol{h + 1}$** where h is the height of the tree.

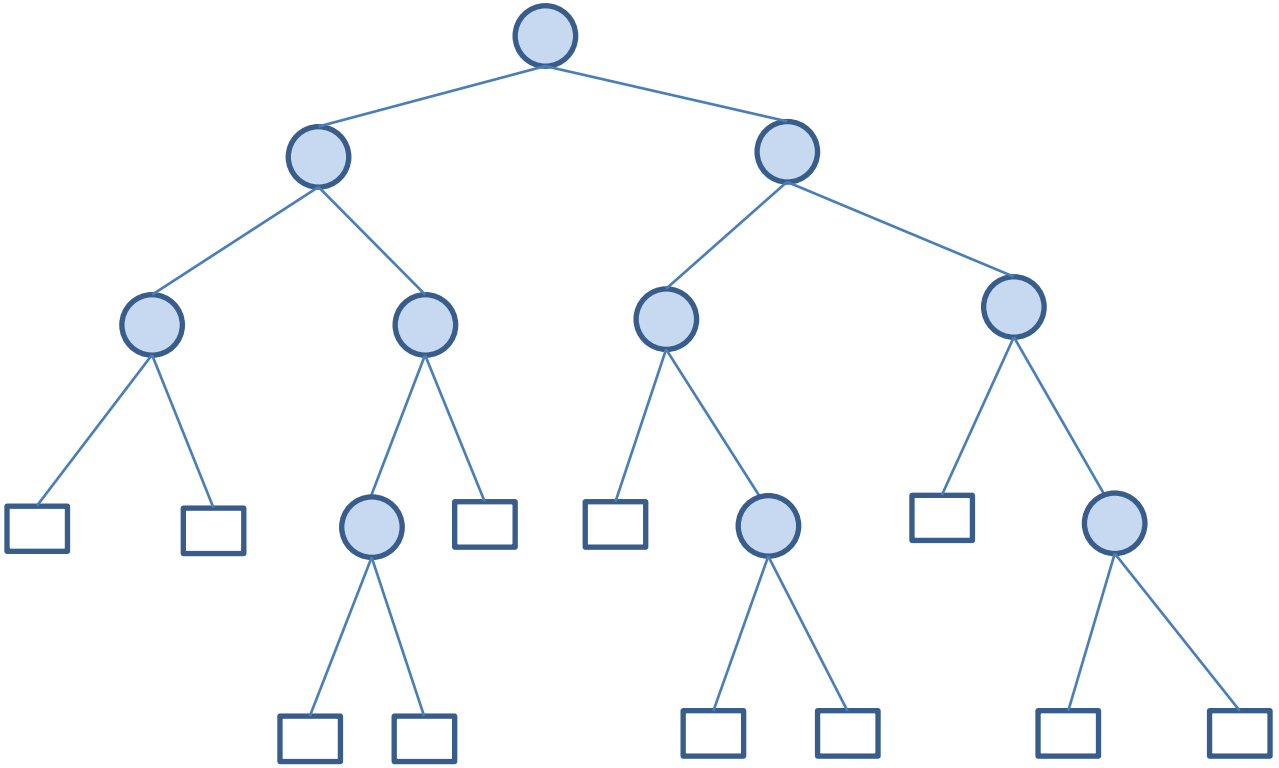
Complexity Analysis (cont'd)

- The **best case** for searching in a BST is the case when **the leaves of the tree are on at most two adjacent levels.** We will call these trees **balanced** (**ισορροπημένα ή ισοζυγισμένα**).
- In this case searching takes $\boldsymbol{O(h) = O(\log n)}$ time where h is the height of the BST and n is the number of nodes.
- Proof?

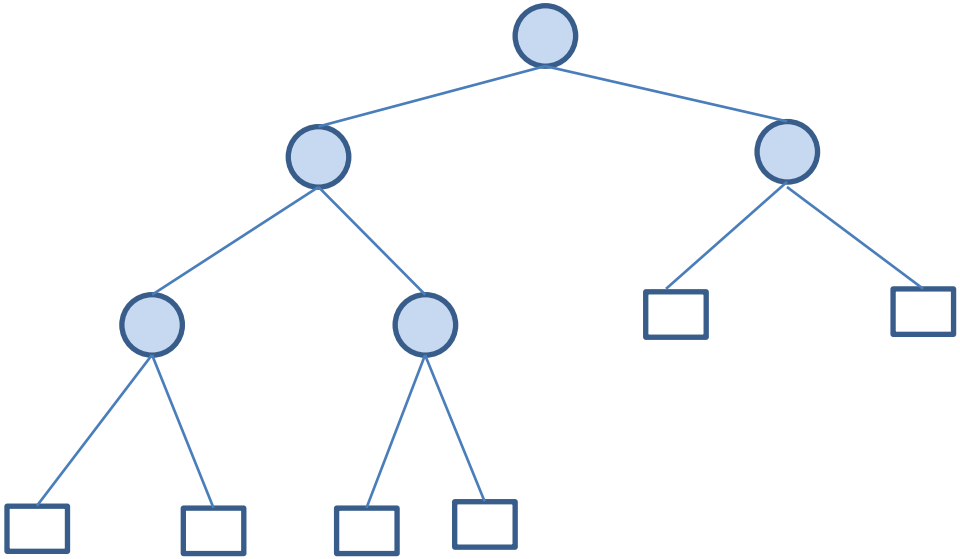
Best Case Examples



Best Case Examples (cont'd)



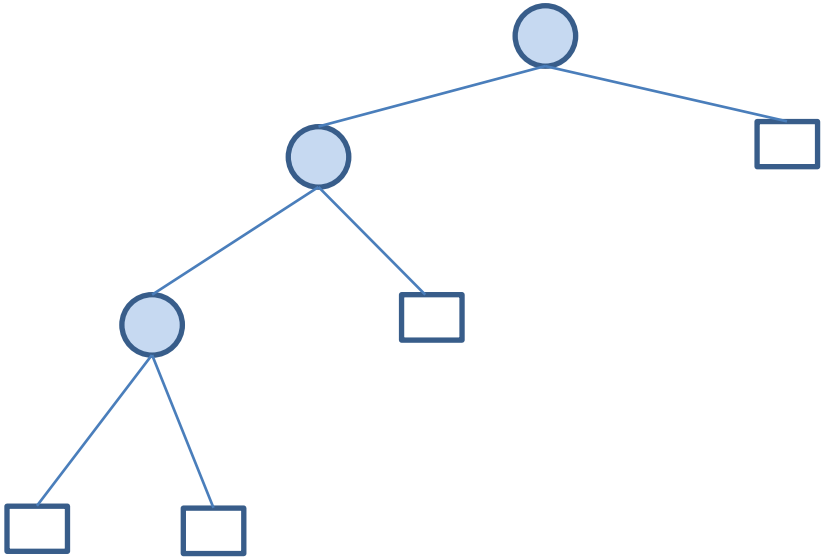
Best Case Examples (cont'd)



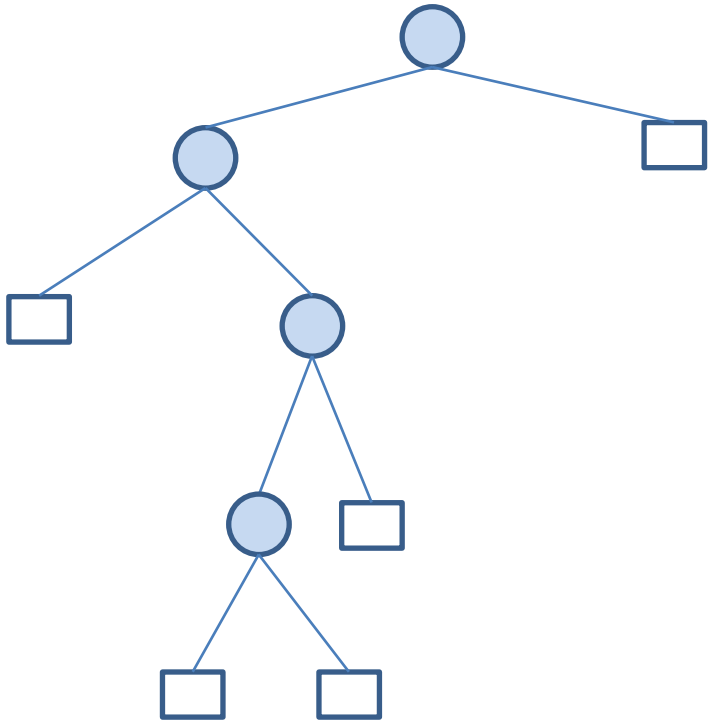
Complexity Analysis (cont'd)

- The **worst case** for searching in a BST is the case when the trees are as **deep and skinny** as possible. These are trees with **exactly one internal node** on each level.
- In this case searching takes $\boldsymbol{O(h) = O(n)}$ time where h is the height of the tree and n is the number of nodes.

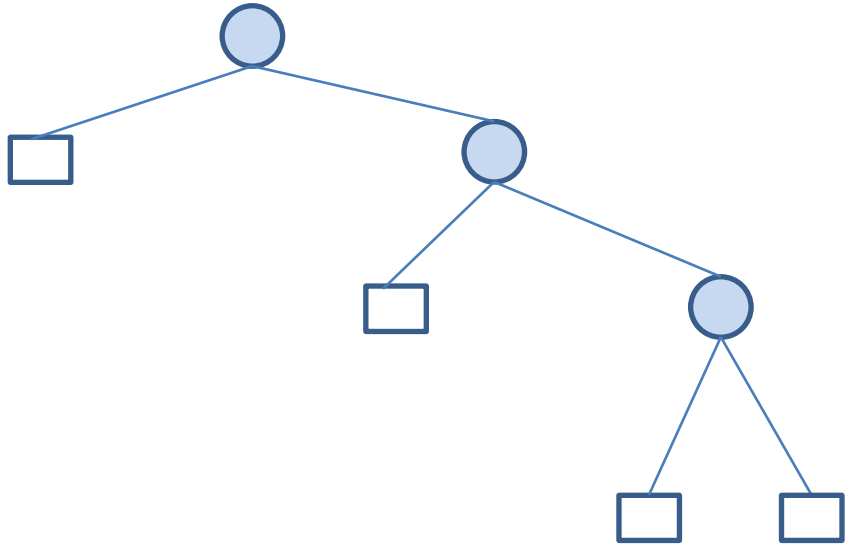
Worst Case Example (Left-linear Tree)



Worst Case Example (Zig-zag)



Worst Case Example (Right-linear)



Complexity Analysis (cont'd)

- The **average case** for searching in a BST is the case when the tree is one from the set of all equally likely binary search trees, and all keys are equally likely to be searched.
- For example, consider **a BST built from n random keys.**
- In this case searching takes $\boldsymbol{O(h) = O(\log n)}$ time where h is the height of the tree and n is the number of nodes.

Complexity Analysis (cont'd)

- The complexity of **insertion** in a binary search tree is the same as the complexity of search given that it makes the same comparisons plus changing a few pointers.
- The complexity of **deletion** is also the same.

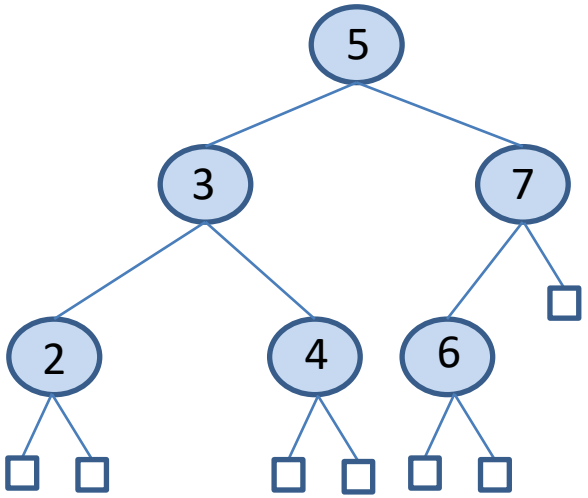
Complexity Analysis (cont'd)

- The complexity of join is $\boldsymbol{O(n)}$ where n is the number of nodes in the tree.
- This follows from the fact that each node can be the root node on a recursive call at most once.

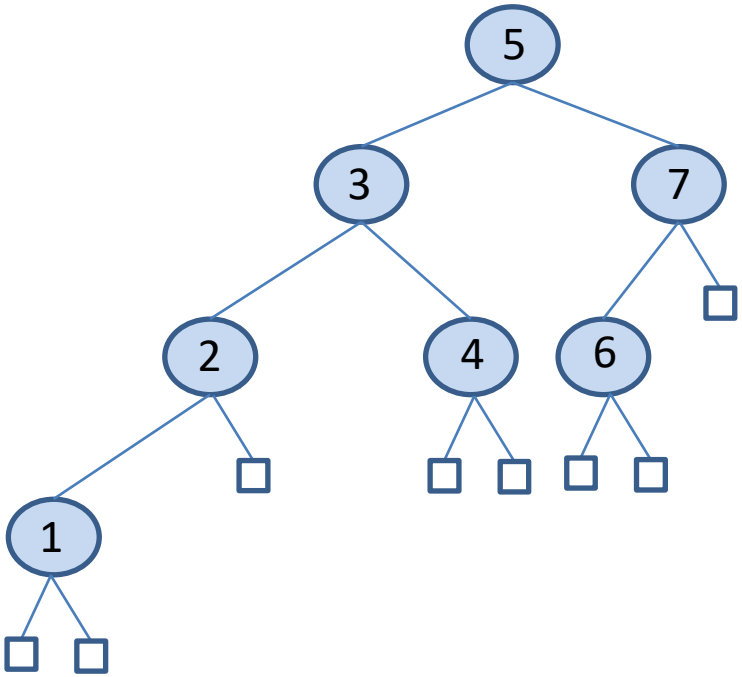
Discussion

- **Balanced BSTs** have very good search times ($O(\log n)$). But if the tree gets out of balance then the performance can degrade to $O(n)$.
- How much does it cost to keep a BST balanced?

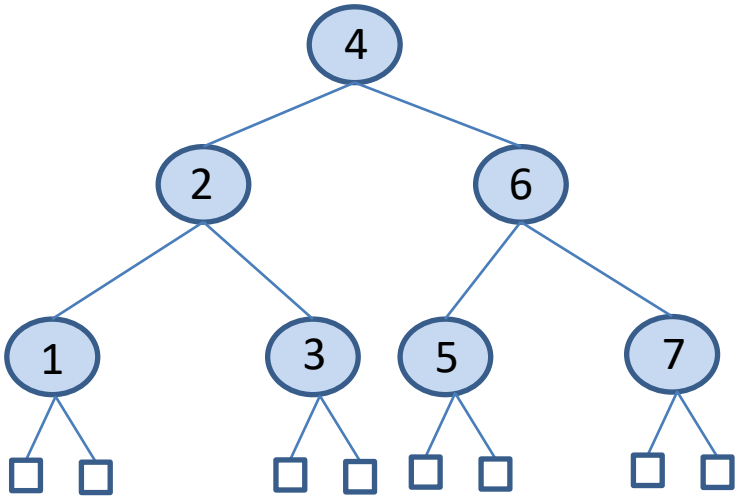
Example



Inserting Key 1



Rebalancing



- Note that every key was moved to a new node, hence rebalancing can take $O(n)$ time.

Question

- Is there a way to achieve $O(\log n)$ search time while also achieving $O(\log n)$ insertion and deletion time in the worst case?
- In the following lectures, we will answer this question positively by introducing special kinds of BSTs that have this property: **AVL trees, 2-4 trees and red-black trees**.

Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
  - Chapter 5. Sections 5.6 and 6.5.
  - Chapter 9. Section 9.7.
- R. Sedgewick. Αλγόριθμοι σε C.
  - Κεφ. 12.