BST: Extension 02

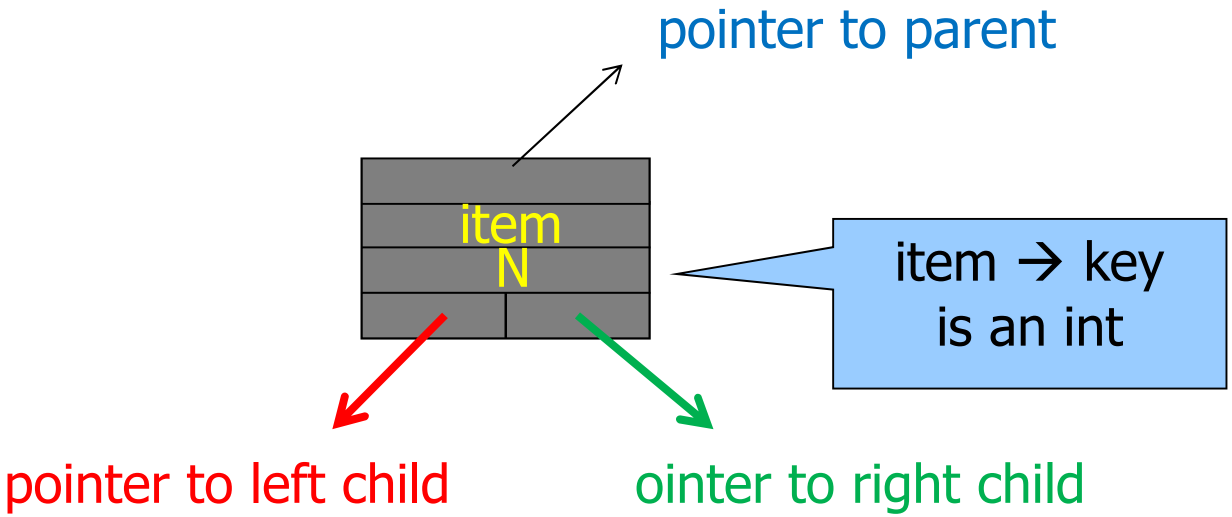
Gianpiero Cabodi and Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

Second BST Extension: Pointers & Counter

- By adding new information to each node it is possible to develop new functions
  - Pointer to the father
  - Number of nodes of the tree rooted at the current node
- This info fields have to be updated (when necessary) by **all** already analysed functions

Binary Search Trees

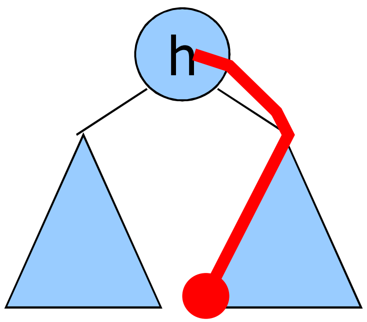
- Node

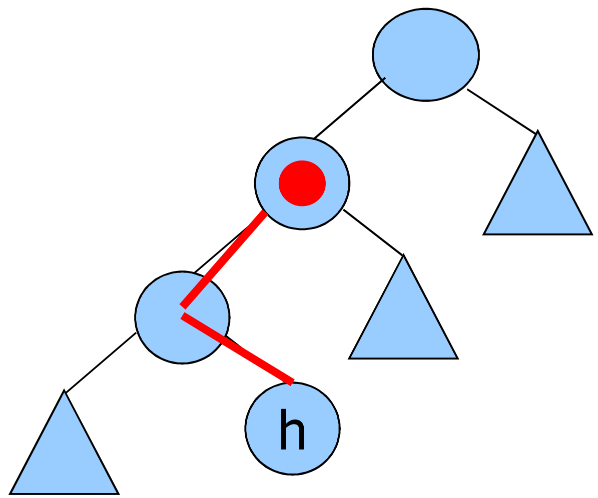



```
typedef struct node *link;
struct node {
  link p;
  Item item;
  int N;
  link l;
  link r;
};
```

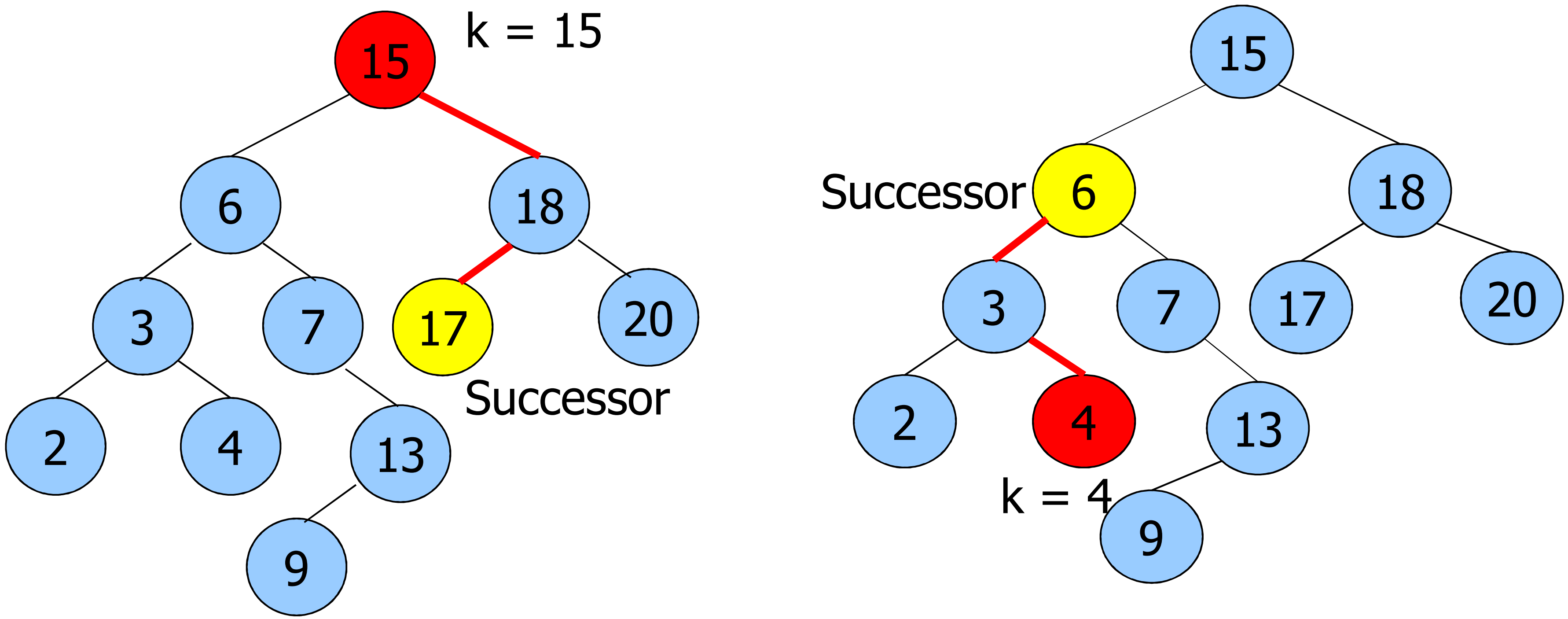
Successor of a node

- Node h with the smallest key larger than the node key
- Two cases
  - $\exists$ Right(h): succ(key(h)) = min(Right(h))
  - $\nexists$ Right(h): succ(key(h)) = first ancestor of h such that the left child is also an ancestor of h





Example

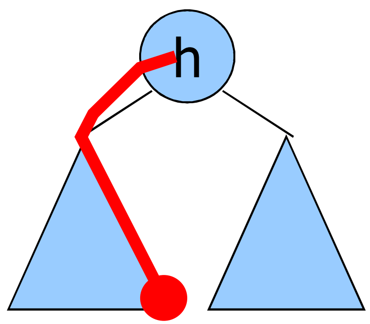

k = 15
15
6
18
3
7
17
20
Successor
2
4
13
9
15
Successor
6
18
3
7
17
20
2
4
13
k = 4
9

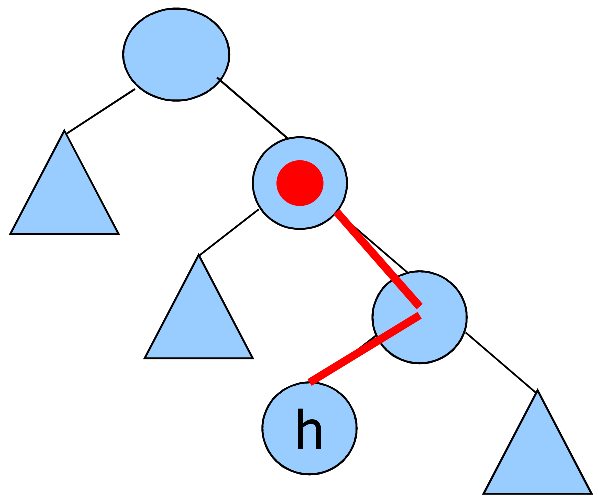
Implementation

```
link search_succ_r (link root, Item item, link z) {
  link p;
  if (root == z) return z;
  if (ITEMless (item, root->item))
    return search_succ_r (root->l, item, z);
  if (ITEMless (root->item, item))
    return search_succ_r (root->r, item, z);
  if (root->r != z) {
    return min_r (root->r, z);
  } else {
    p = root->p;
    while (p != z && root == p->r) {
      root = p; p = p->p;
    }
    return p;
  }
}
```

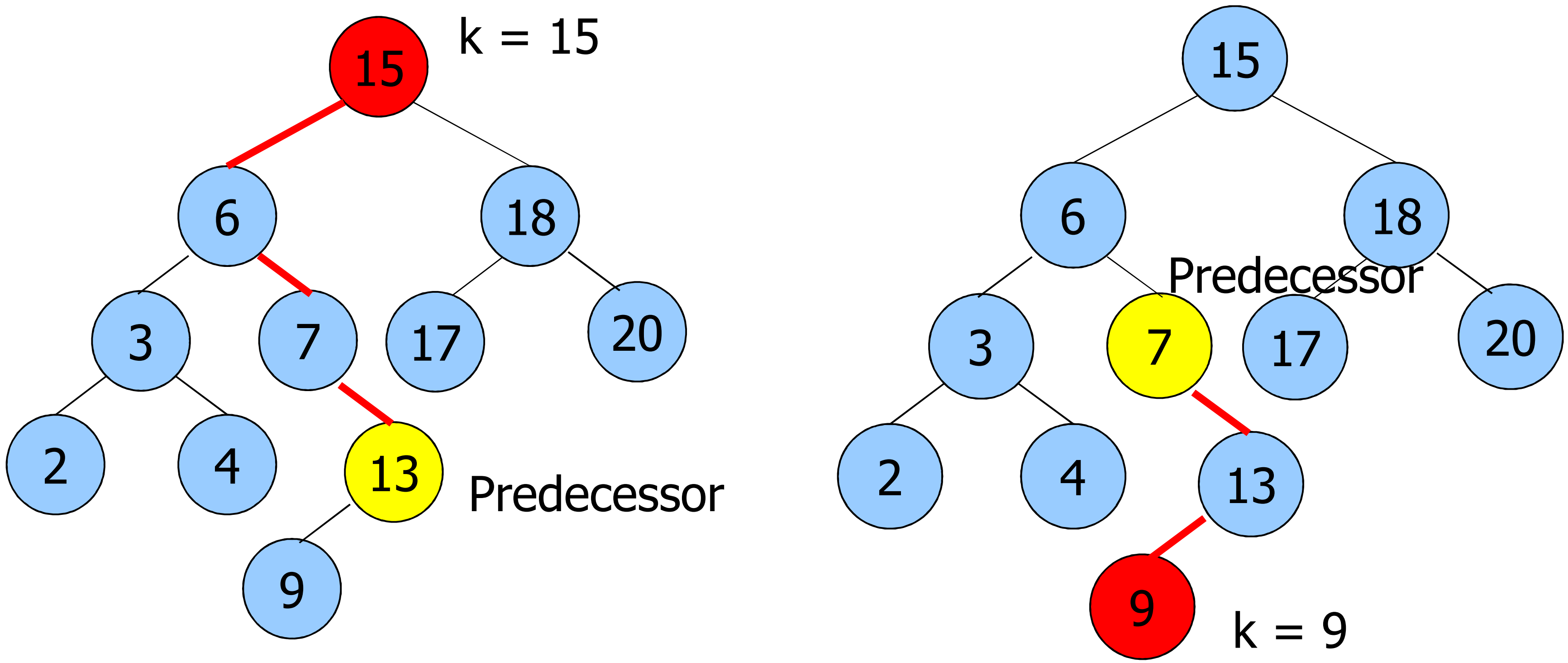
Predecessor of an item

- Node h with the largest item smaller than the item key
- Two cases
  - $\exists$ Left(h): pred(key(h)) = max(Left(h))
  - $\nexists$ Left(h): pred(key(h)) = first ancestor of h such that the right child is also an ancestor of h





Example



k = 15
15
6
18
3
7
17
20
2
4
13
Predecessor
9
15
6
18
Predecessor
3
7
17
20
2
4
13
9
k = 9

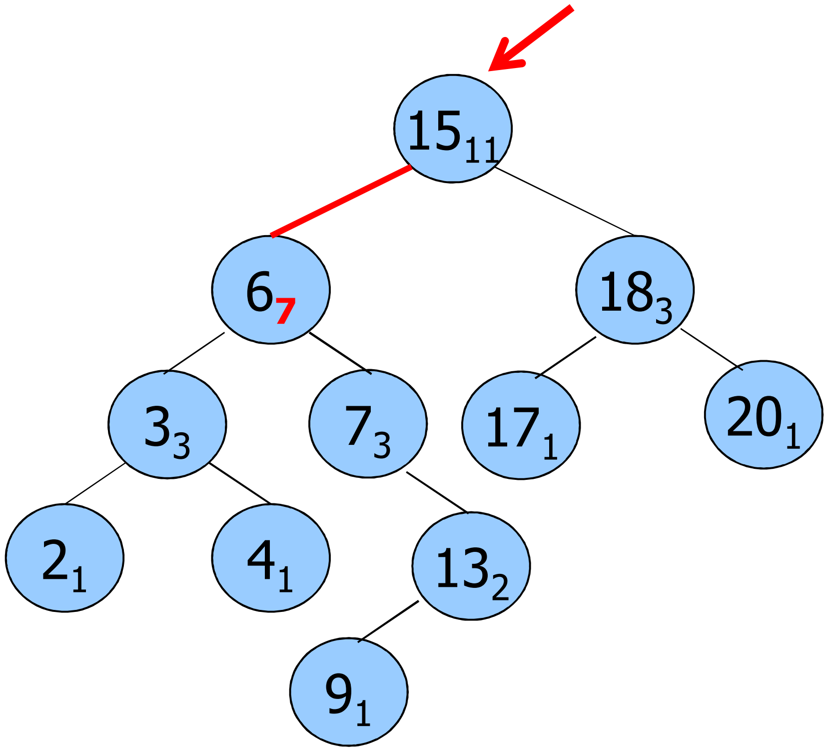
Implementation

```
link search_pred_r (link root, Item item, link z) {
  link p;
  if (root == z) return z;
  if (ITEMless (item, root->item))
    return search_pred_r (root->l, item, z);
  if (ITEMless (root->item, item))
    return search_pred_r (root->r, item, z);
  if (root->r != z) {
    return max_r (root->l, z);
  } else {
    p = root->p;
    while (p != z && root == p->l) {
      root = p; p = p->p;
    }
    return p;
  }
}
```

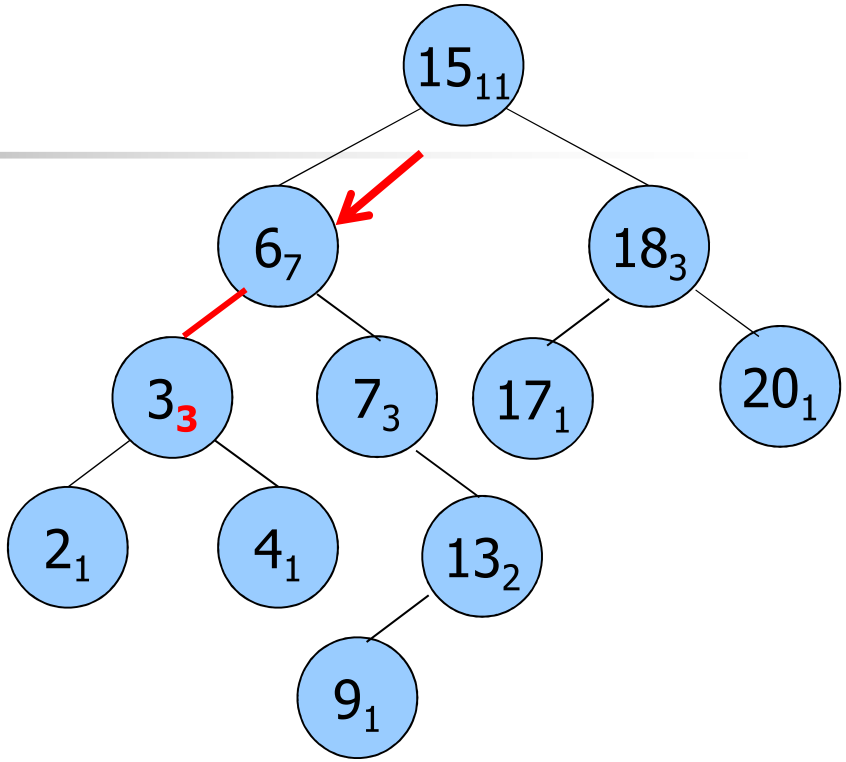
Select

- Select the item with the k-th smallest key (zero based indexing, for example, k=0 means the item with the smallest key)
- t is the number of nodes of the left sub-tree
  - k = t: Return the sub-tree root
  - k < t: Recur into the left sub-tree to look-for the smallest k-th key
  - k > t: Recur on the right sub-tree to look-for the (k-t-1)-th smallest key

Example

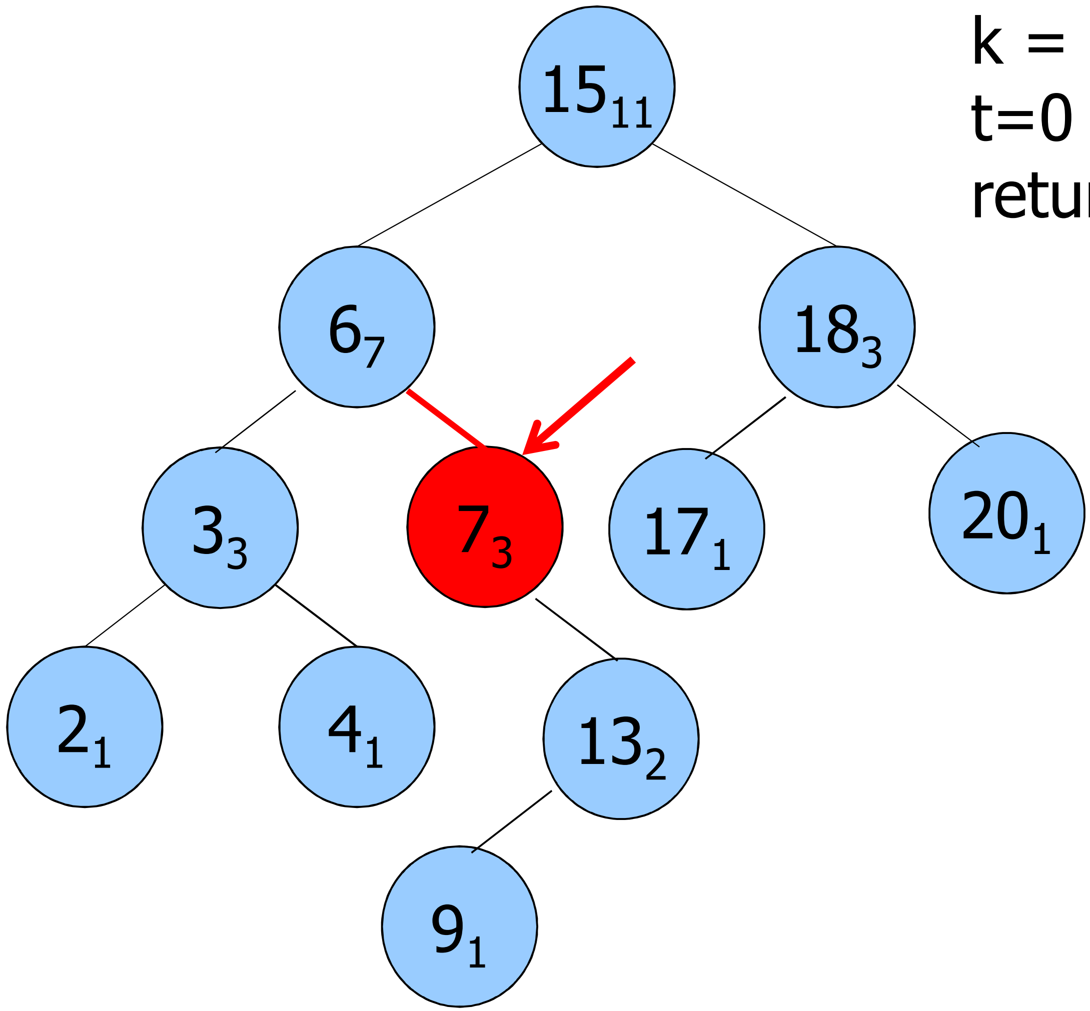


Index = k = 4
Fifth smallest
Key t=7
7>4 left recur



k = 4
t=3
3<4 right recur
Look-for
k=4-3-1=0

Example



k = 0
t=0
return 7

Implementation

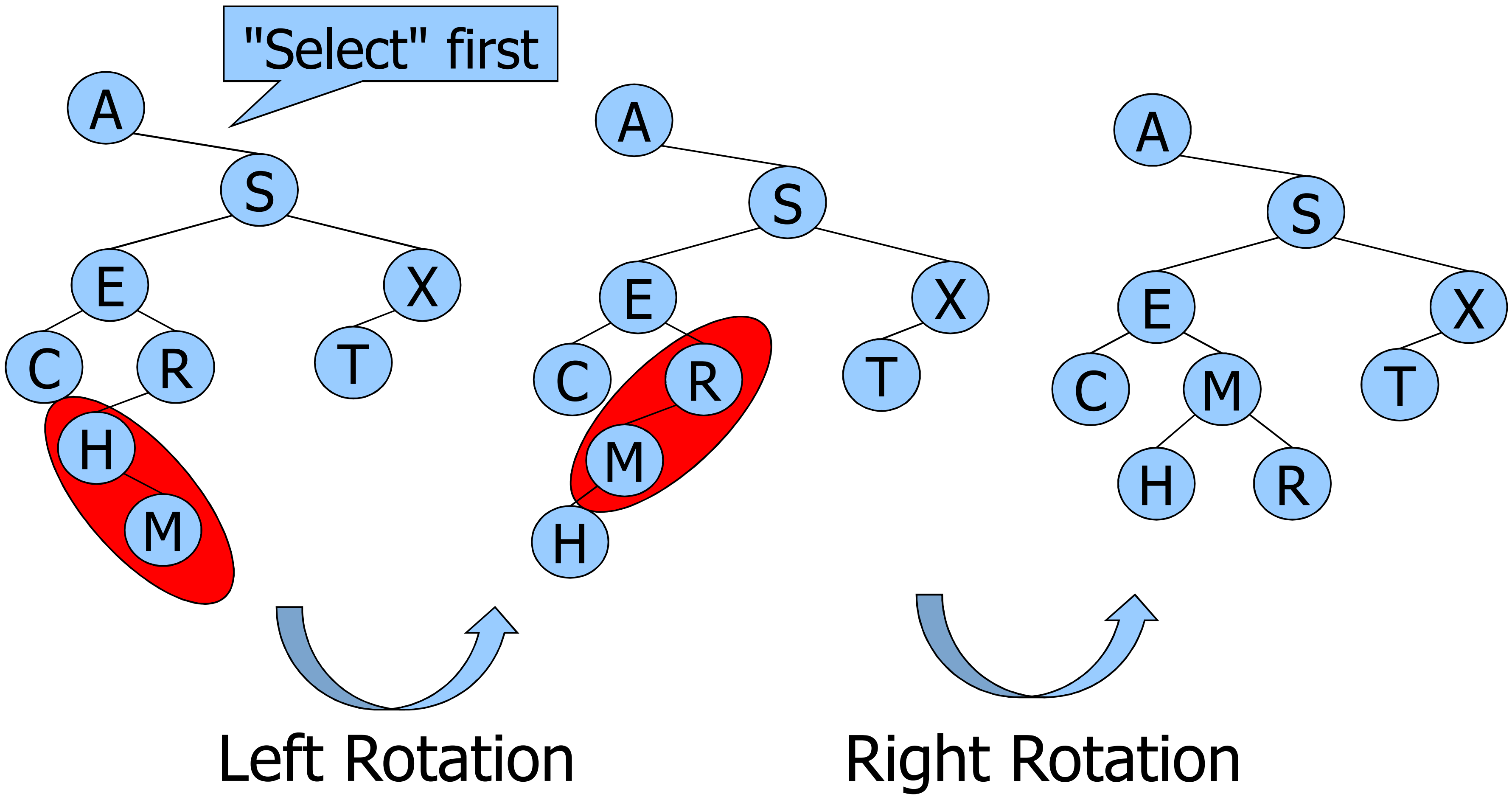
```
link select_r (link root, int k, link z) {
  int t;

  if (root == z)
    return z;

  t = (root->l == z) ? 0 : root->l->N;

  if (k < t)
    return select_r (root->l, k, z);
  if (k > t)
    return select_r (root->r, k-t-1, z);

  return root;
}
```

Partition

- Restructuring the tree, forcing the smallest k-th key into the root
- Consider the sub-tree root node
  - $k < t$: Recur on the left sub-tree, partition with respect to the smallest k-th key, at the end right-rotation
  - $k > t$: Recur on the right sub-tree, partition with respect to the smallest (k-t-1)-th key, at the end left rotation
- Partitioning is often performed around the median key

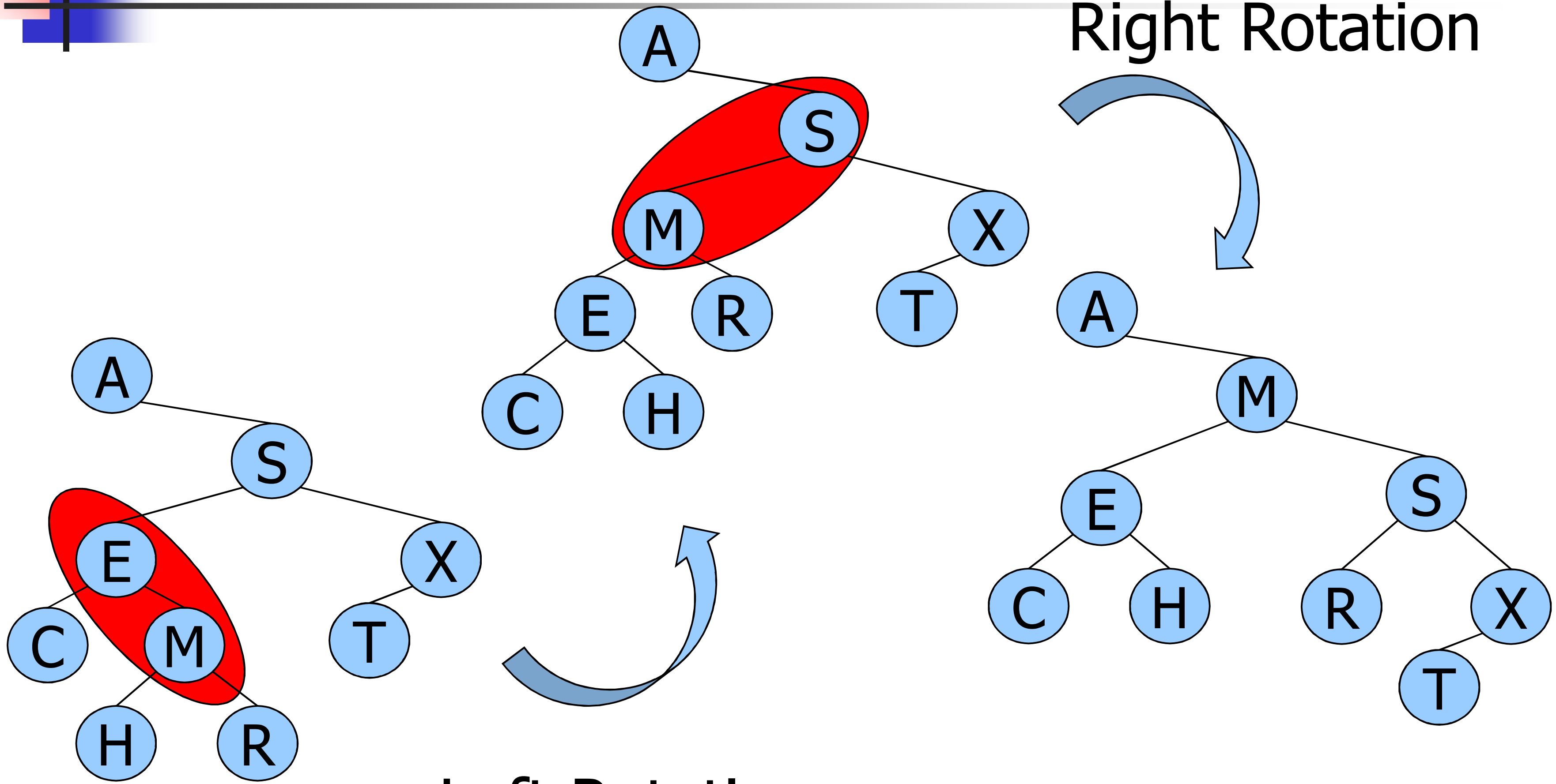
Example

Partition with respect to the 5-th smallest key (M, k=4)



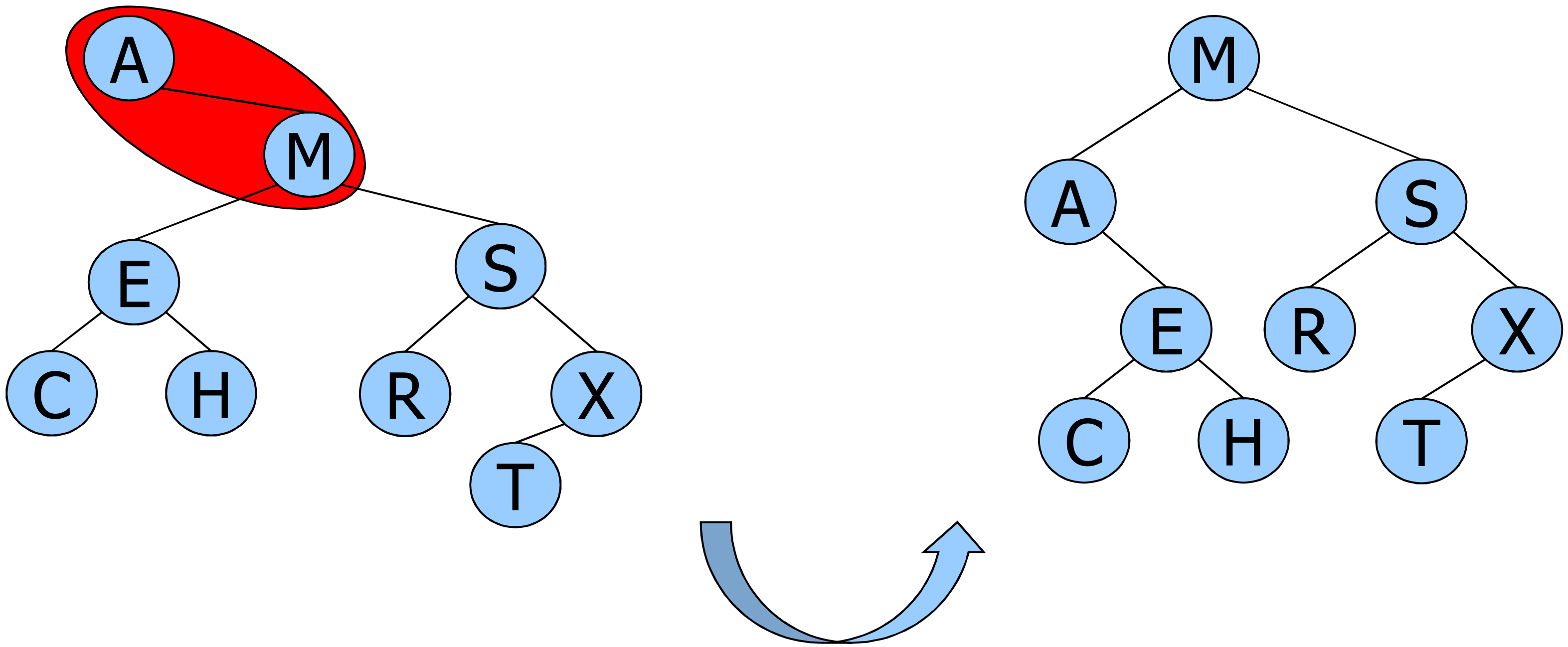
Example

Right Rotation



Left Rotation

Example



Left Rotation

Implementation

```
link part_r (link h, int k) {
  int t  = h->l->N;

  if (k < t) {
    h->l = part_r (h->l, k);
    h = rotR (h);
  }
  if (k > t) {
    h->r = part_r (h->r, k-t-1);
    h = rotL (h);
  }

  return h;
}
```

Delete: Version 2

- To delete from a BST a node with an item with a given key k, it is possible to use the partition funciton togheter with rotations
  - Check whether the node with the item to delete belongs to one sub-tree. If yes, recursive delete such a sub-tree
  - If it is the root, delete the node
  - The new root is the succ or pred of the deleted item
    - Rotate one of them up to the root
  - Combine the two sub-trees into the new root

Example