BSTs: Binary Search Trees

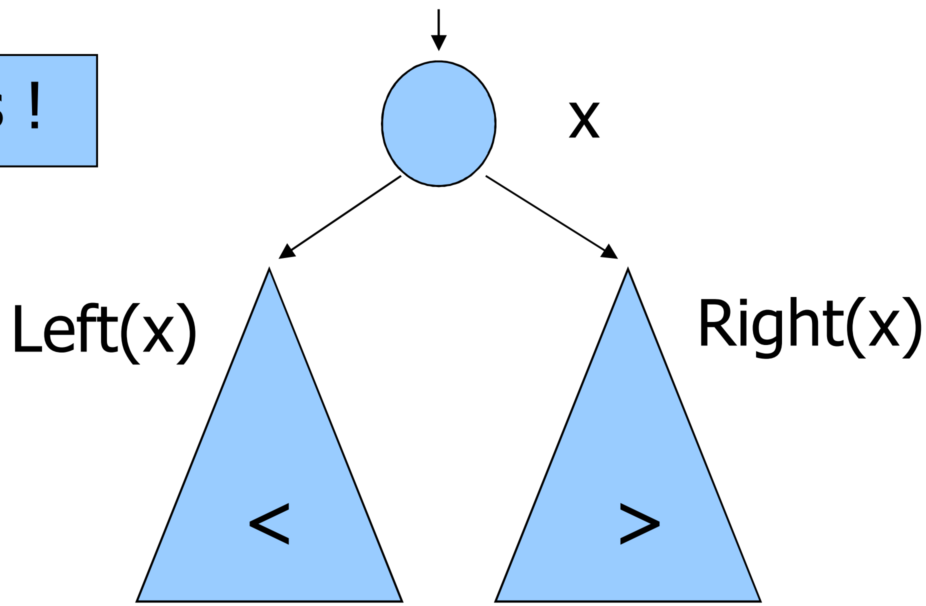
Gianpiero Cabodi and Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino

Binary Search Trees (BSTs)

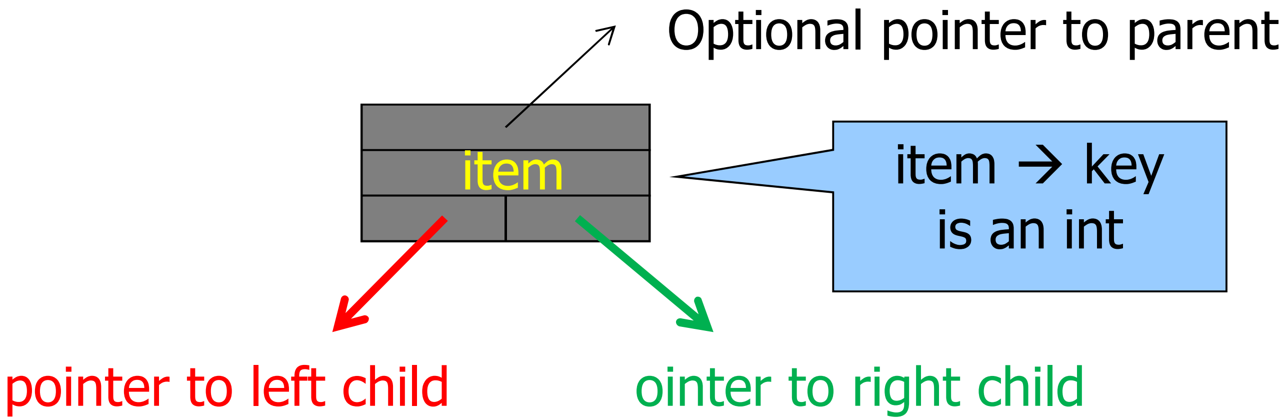
- Binary tree with the following property
- $\forall$node x
  - $\forall$ node $y \in Left(x)$, $key[y] < key[x]$
  - $\forall$ node $y \in Right(x)$, $key[y] > key[x]$

Distinct keys !



Binary Search Trees

- Node



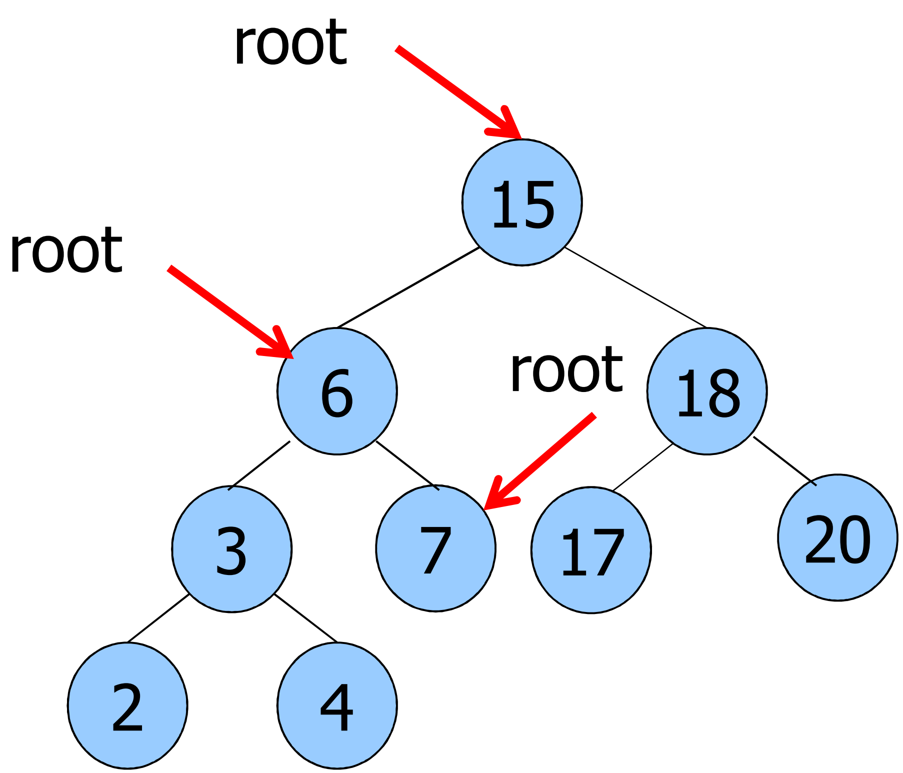

```
typedef struct node *link;
struct node {
  Item item;
  link l;
  link r;
};
```

Search

- Recursive search of a node storing a node with the desired key
  - Visit tre tree from the root
  - Termination: Either the searched key is the one of the current node (search hit) or an empty tree has been reached (search miss)
  - Recursion: from the current node
    - On the left sub-tree if the searched key is smaller than the key of the current node
    - On the right sub-tree otherwise

Example

Search node with item (key) 7



Implementation

```
link search_r (link root, Item item, link z) {
  if (root == z)
    return (z);

  if (ITEMless(item, root->item))
    return search_r (root->l, item, z);

  if (ITEMless(root->item, item))
    return search_r (root->r, item, z);

  return root;
}
```

Minimum and Maximum

- Minimum
  - Follow pointers onto left sub-trees until they exist
- Maximum
  - Follow pointers onto right sub-trees until they exist

Implementation

```
link min_r (link root, link z) {
  if (root == z)
    return (z);
  if (root->l == z)
    return (root);
  return min_r (root->l, z);
}

link max_r (link root, link z) {
  if (root == z)
    return (z);
  if (root->r == z)
    return (root);
  return max_r (root->r, z);
}
```

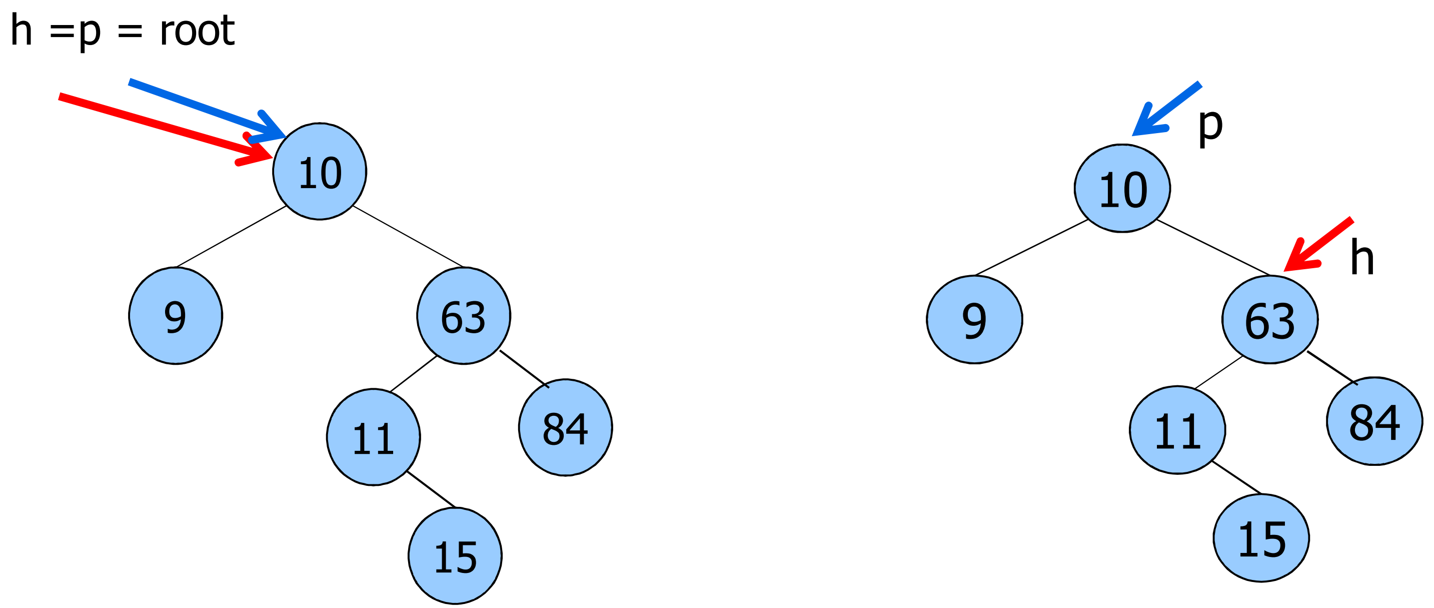
Leaf Insert

Insert into a BST a node storing a new item (mantaining the BST property)

- If the BST is empty, create a new tree
- Recursive insert: Insert into the left or right sub-tree depending on the comparison between the item and the current node key
- Iterative insert: Find the position first, then add the new node

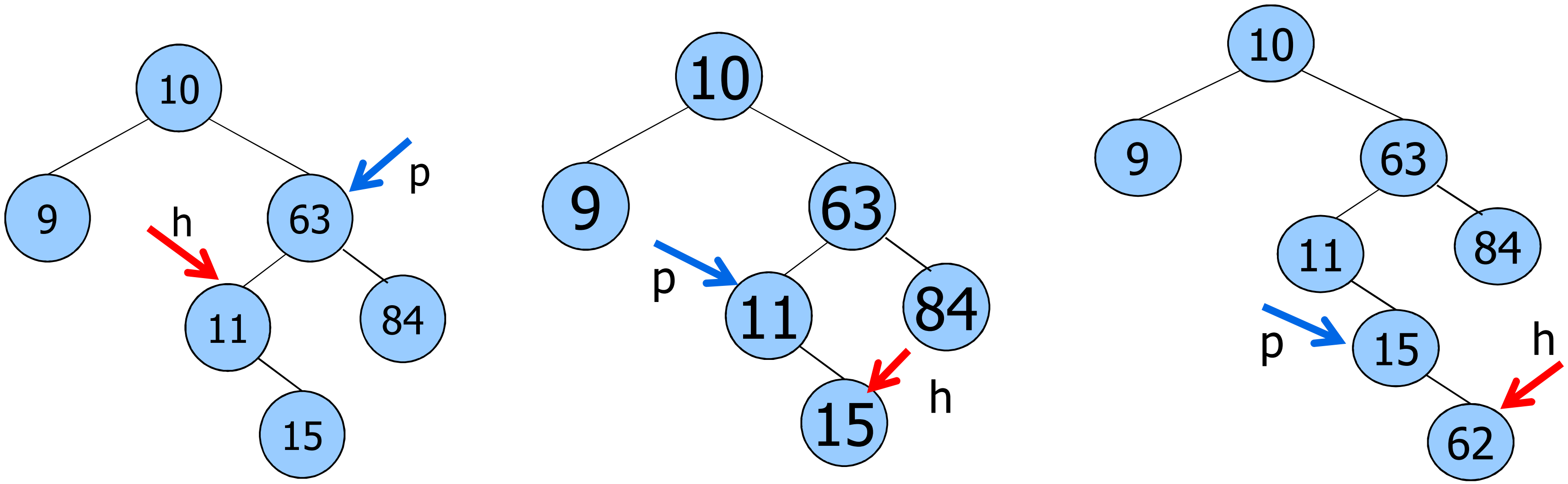
Example

Insert node with item (key) 62



Example

Insert node with item (key) 62



Implementation Recursive

```
link insert_r (link root, Item x, link z) {
  if (root == z)
    return (NEW(x, z, z));

  if (ITEMless(x, root->item))
    root->l = insert_r (root->l, x, z);
  else
    root->r = insert_r (root->r, x, z);

  return root;
}
```

Implementation Iterative

```
link insert_i (link root, Item x, link z) {
  link p = root, h = p;
  if (root == z) {
    return (NEW(x, z, z));
  }
  while (h != z) {
    p = h;
    h = (ITEMless(x, h->item)) ? h->l : h->r;
  }
  h = NEW(x, z, z);
  if (ITEMless(x, p->item))
    p->l = h;
  else
    p->r = h;
  return root;
}
```

Delete

To delete a previously stored node from a BST we have to recursively search the key into the BST

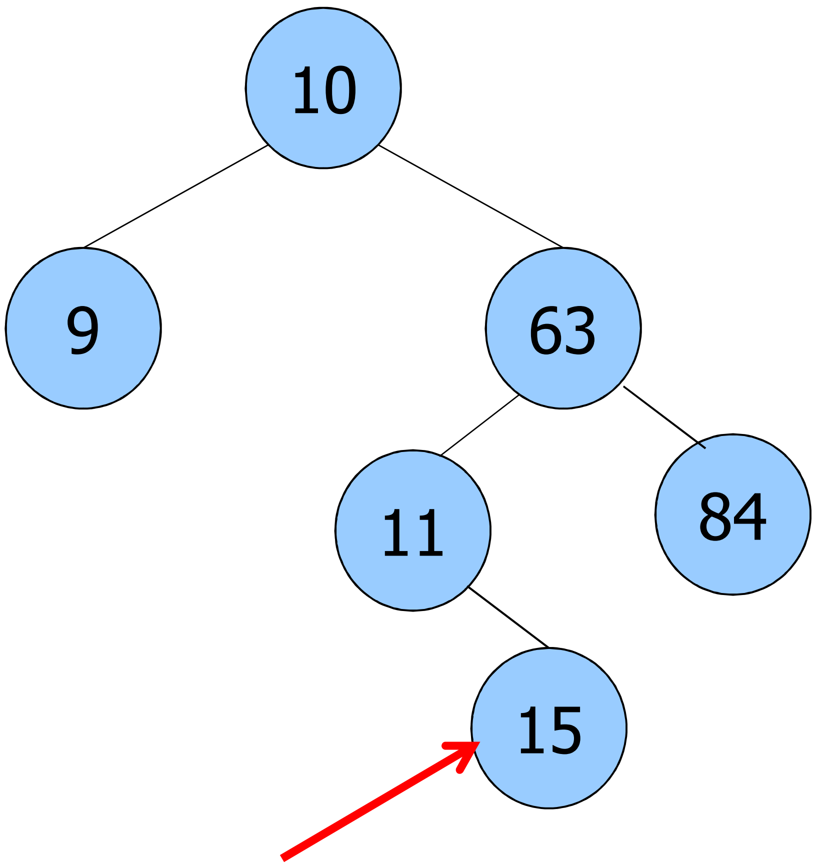
- Delete from the left or right sub-tree depending on the comparison between the item and the current node key
- If the BST is empty, just return doing nothing

Delete

- If the node with the desired key is found, apply one of the following three basic rules
  - If the node has no children, simply remove it
  - If the node has one child, then elevate that child to take the node position in the tree
  - If the node has two children, find the greatest node in its left subtree (or the smallest node in its right subtree) and substitute the node with it

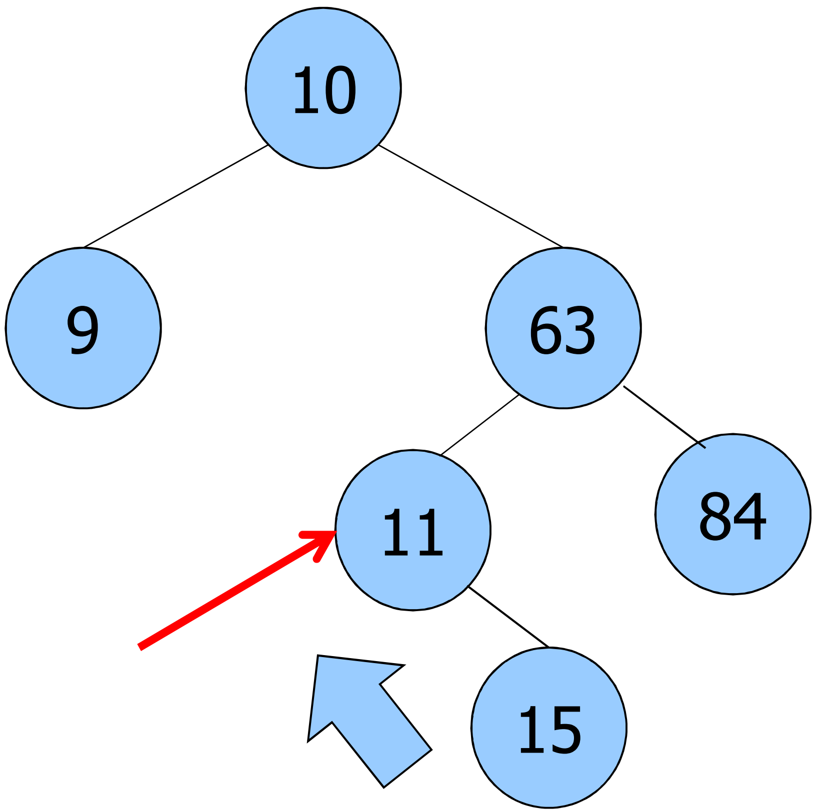
Example: Rule 1

Delete node with item (key) 15



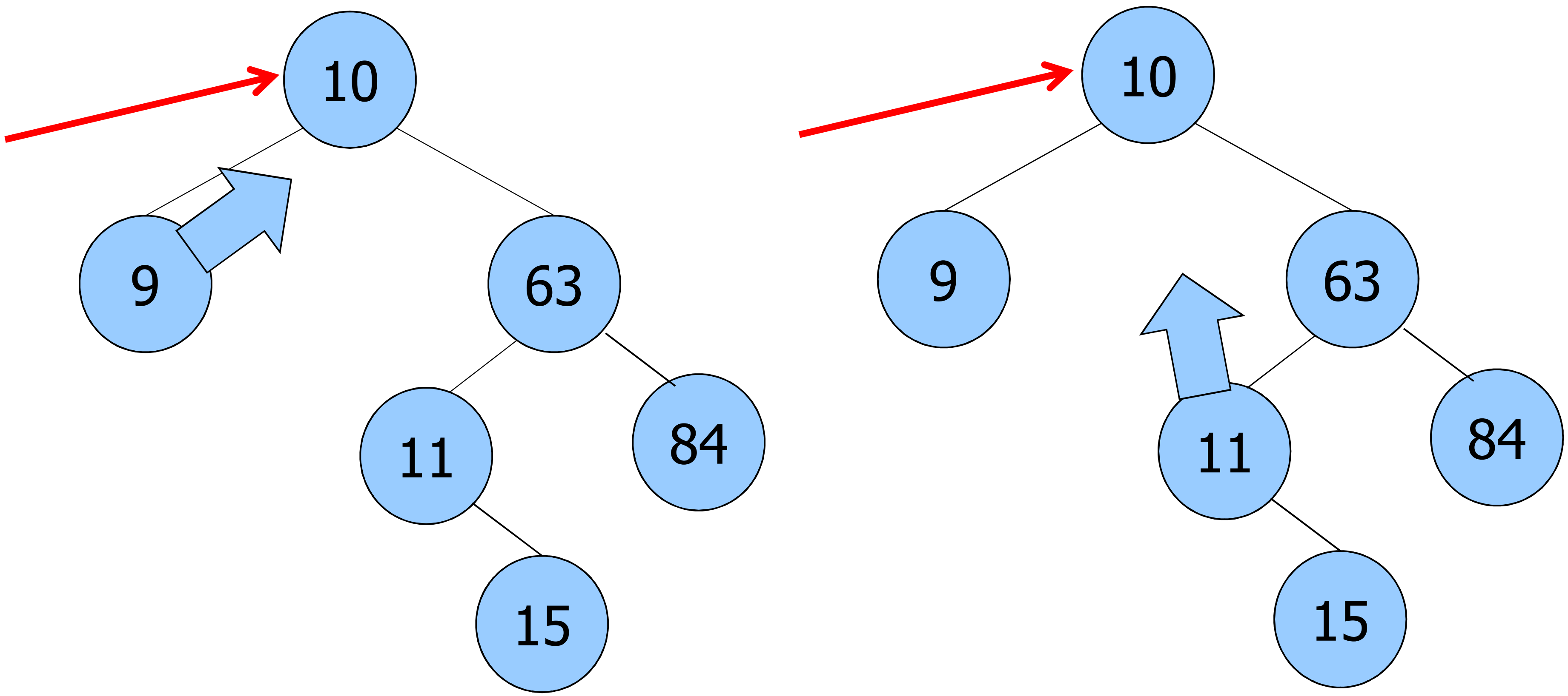
Example: Rule 2

Delete node with item (key) 11



Example: Rule 3 (A and B)

Delete node with item (key) 10



Implementation Recursive

```
link delete_r (link root, Item x, link z) {
  link p;
  Item val;

  if (root == z)
    return (root);
  if (ITEMless (x, root->item)) {
    root->l = delete_r (root->l, x, z);
    return (root);
  }
  if (ITEMless(root->item, x)) {
    root->r = delete_r (root->r, x, z);
    return (root);
  }
```

Implementation

Recursive

```
  p = root;
  if (root->r == z) {
    root = root->l;
    free (p);
    return (root);
  }
  if (root->l == z) {
    root = root->r;
    free (p);
    return (root);
  }
  root->l = max_delete_r (&val, root->l, z);
  root->item = val;
  return (root);
}
```

Node found

Rule 0 or 1 apply
(right child NULL get left one)

Rule 0 or 1 apply
(left child NULL get right one)

Rule 2 apply

Implementation Recursive

```
link max_delete_r (Item *x, link root, link z) {
  link tmp;

  if (root->r == z) {
    *x = root->item;
    tmp = root->l;
    free (root);
    return (tmp);
  }

  root->r = max_delete_r (x, root->r, z);
  return (root);
}
```

Find and delete max value into left child

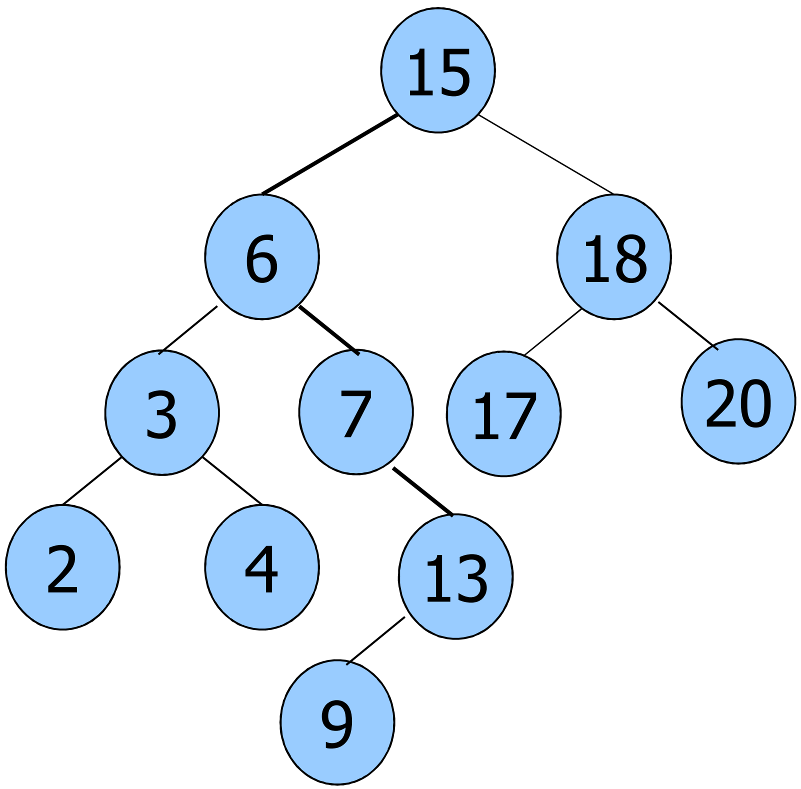
Get key
Free node
Return pointer to left child

Recur until there is a right child

Sort

An in-order visit delivers keys in ascending order

2 3 4 6 7 9 13 15 17 18 20



The (inferior) median key of a set of n element is the element stored in position $\lfloor (n+1)/2 \rfloor$ in the ordered sequence of the element set

Complexity

Operations on BSTs have complexity $T(n) = O(h)$ where h is the height of the tree

- Tree fully balanced with n nodes
  - Height $h = \alpha(\log_2 n)$
- Tree completely unbalanced with n nodes
  - Height $h = \alpha(n)$
- $O(\log n) \leq T(n) \leq O(n)$