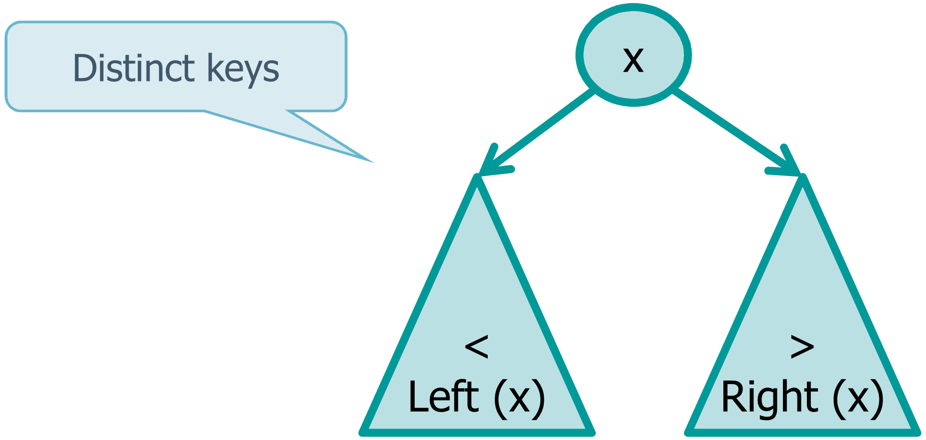
## Trees and BSTs

# BSTs: Binary Search Trees

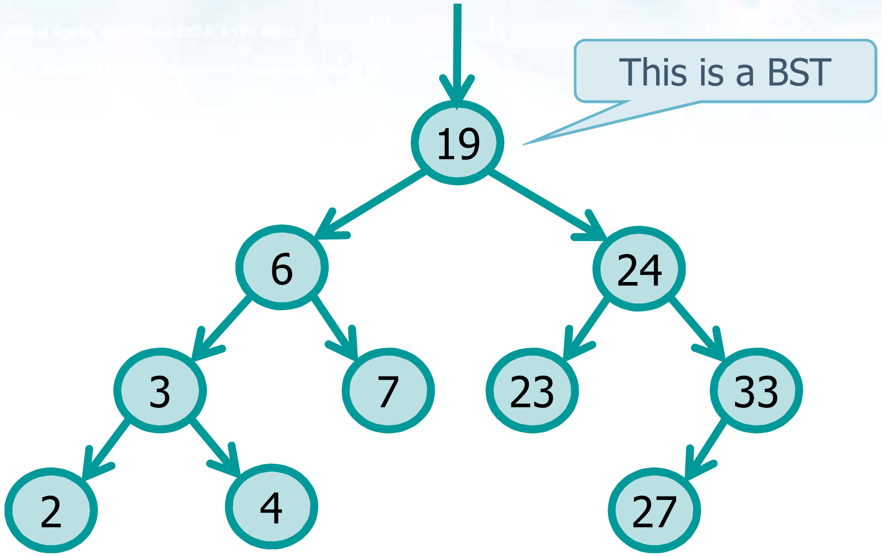
Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

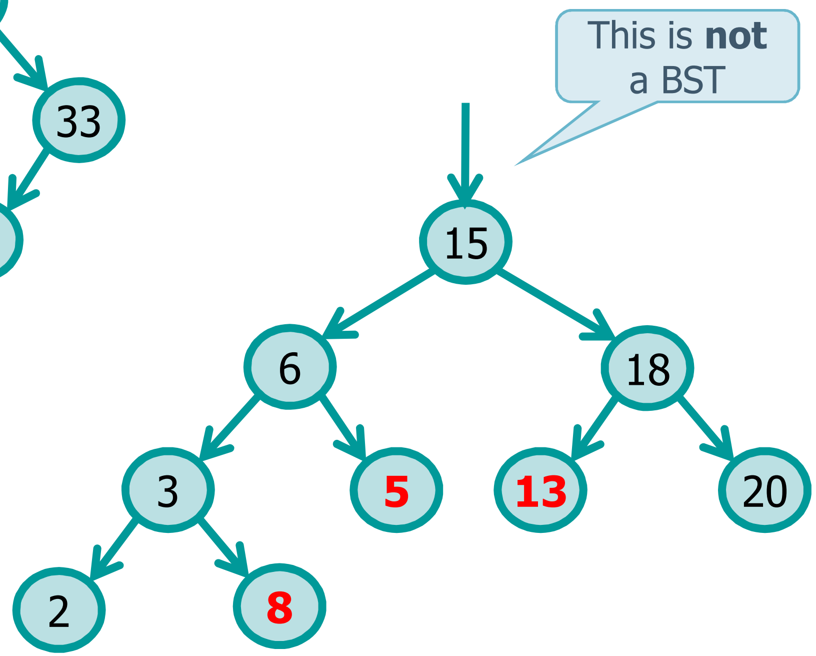
# Binary Search Trees (BSTs)

- Binary tree with the following property
  - ∀ node x
    - ∀ node y∈Left(x), key[y] < key[x]
    - ∀ node y∈Right(x), key[y] > key[x]



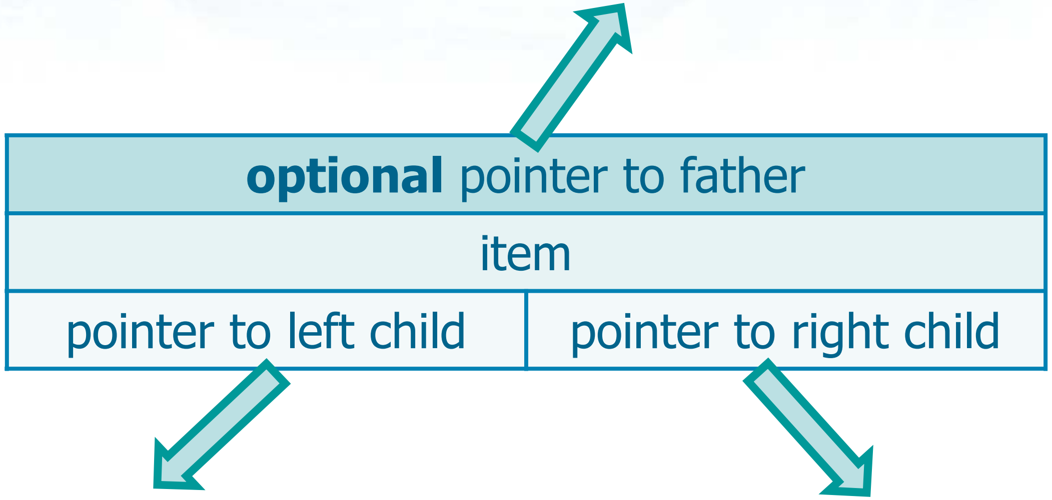
# Examples


This is a BST
19
6
24
3
7
23
33
2
4
27


This is **not** a BST
15
6
18
3
5
13
20
2
8

# Binary Search Trees

item → key
is an integer
(in this section)




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

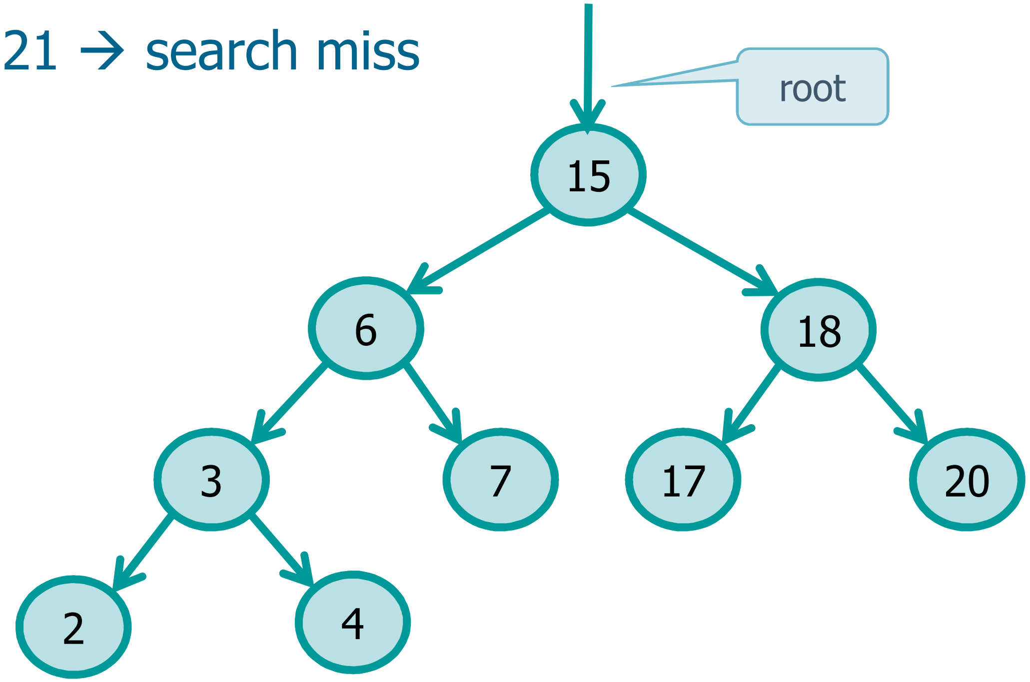
ADT: We use functions to compare keys, etc.

# Search

- Given a BST already formed, how to we search a key in it?
  - Recursive search of a node storing the desired key
    - Visit the tree from the root
    - Terminate the search if
      - Either the searched key is the one of the current node (search hit) or
      - An empty tree (the sentinel node or a NULL pointer) has been reached (search miss)
    - Recur from the current node on
      - The left sub-tree if the searched key is smaller than the key of the current node
      - The right sub-tree otherwise

# Example

- Given the following BST look for
  - key = 7 → search hit
  - key = 20 → search hit
  - key = 21 → search miss



# Recursive implementation

Function **item_less** compares keys

Root node

Searched key

Sentinel or NULL

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

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

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

  return root;
}
```

Sentinel z or NULL

Search miss

Left recursion

Right recursion

Search hit

# Iterative implementation

Function **item_equal** compares keys

Root node

Searched key

Sentinel or NULL

```
link search_i (link root, Item item, link z) {

  while (root != z) {
   if (item_equal (item, root->item))
     return (root);

   if (item_less(item, root->item))
     root = root->l;
   else
     root = root->r;
  }

  return (root);
}
```

Search hit

Move down left

Move down right

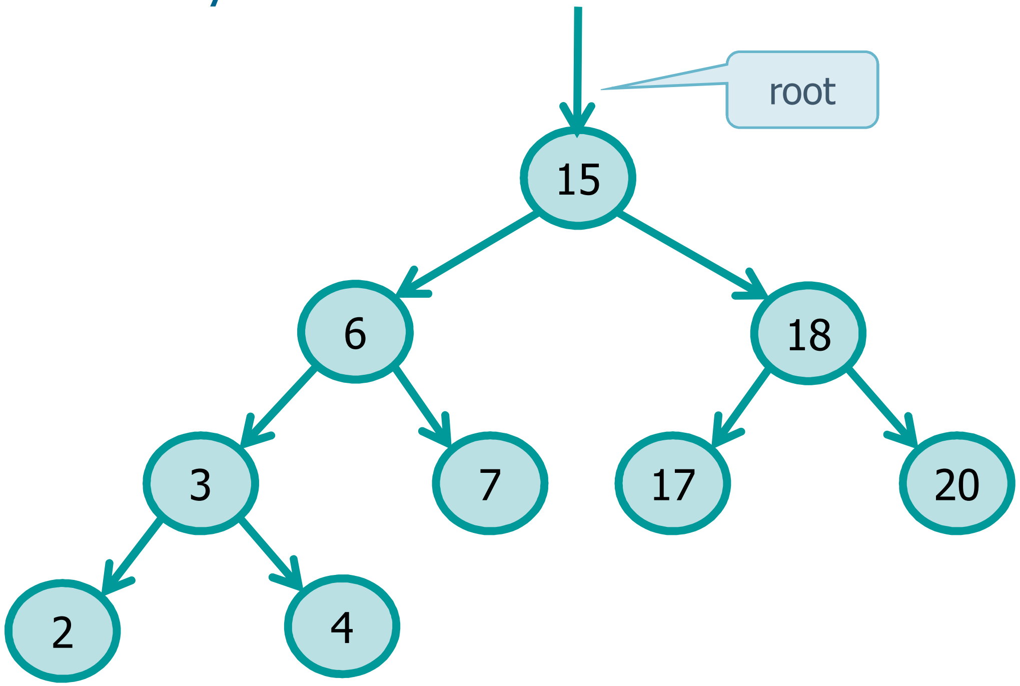
Search miss

# Minimum and Maximum

- Find the minimum key in a given BST
  - If the BST is empty return NULL
  - Follow pointers onto **left** sub-trees until they exist
  - Return last key encountered
- Find the maximum ley in a given BST
  - If the BST is empty return NULL
  - Follow pointers onto **right** sub-trees until they exist
  - Return last key encountered

# Example

- Given the following BST look for
  - Minimun → key = 2
  - Maximum → key = 20



# Recursive 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);
}
```

Empty BST

Termination condition

Left recursion

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

Empty BST

Termination condition

Right recursion

# Iterative implementation

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

Empty BST

Move down

Return result

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

Empty BST

Move down

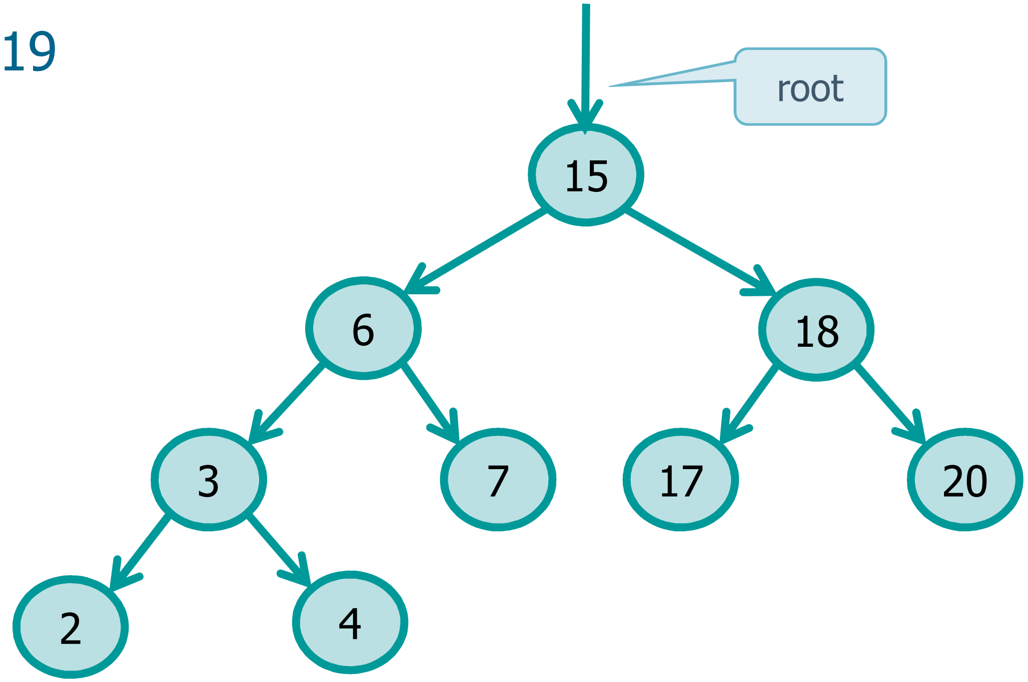
Return result

# Leaf Insert

- Insert into a BST a node storing a new item
- The BST property must be maintained
  - If the BST is empty
    - Create a new tree node with the new key and return its pointer
  - Recursion
    - Insert into the left sub-tree if the item key is less than the current node key
    - Insert into the right sub-tree if the item key is larger than the current node key
- Notice that in all cases the new node in on a BST leaf (terminal node with no children)

# Example

- Given the following BST insert
  - key = 5
  - key = 13
  - key = 19



# Recursive implementation

Function **node_new** creates a new node

BST root

Key

Termination condition: Insert a new node

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

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

  return root;
}
```

Left recursion

Right recursion

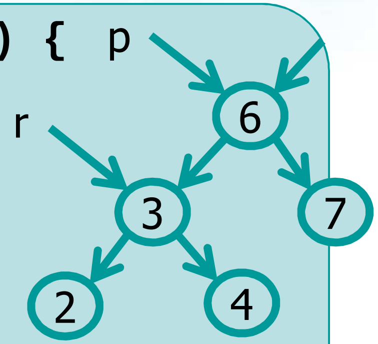
Assign (new) pointer onto parent pointer on the way back

# Iterative implementation

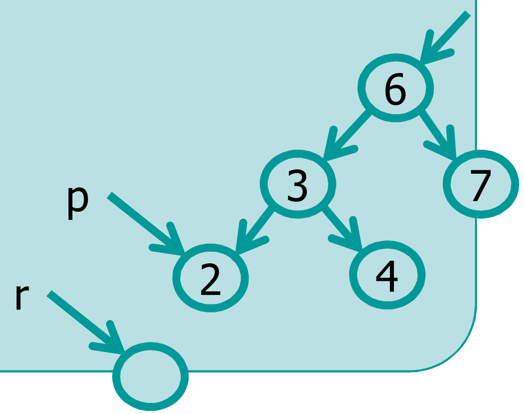
- BST insert can be also be performed using an iterative procedure
  - Find the position first
  - Then add the new node
- As we cannot assign the new pointer on the way back (on recursion) we need two pointers
  - Please remind the ordered list implementation
    - The visit was perfomed either using two pointers or the pointer of a pointer to assign the new pointer to the the pointer of the previous element

# Iterative implementation

```
link insert_i (link root, Item x, link z) {
  link p, r;

  if (root == z) {
    return (node_new(x, z, z));
  }
  r = root;
  p = r;
  while (r != z) {
    p = r;
    r = (item_less(x, r->item)) ? r->l : r->r;
  }
  r = node_new (x, z, z);
  if (item_less (x, p->item))
    p->l = r;
  else
    p->r = r;
  return root;
}
```




Move left or move right




Create link with parent in the right direction

# Node Extract

- Given a BST delete a node with a given key
  - We have to recursively search the key into the BST
  - If we found it
    - Then we must delete it
    - Otherwise the key is not in the BST and we just return
- Search is performed as before and it is followed by the procedure to delete the node

# Node Extract

- To sum up we have to
  - If the BST is empty
    - Return doing nothing
  - If the current node is the one with the desired key, then apply one of the following three basic rules
    - If the node has no children, simply remove it
    - If the node has one child, then move the chile one level higher in the tree to substitute the erased node in the tree with its child
    - If the node has two children, find
      - The greatest node in its left subtree or
      - The smallest node in its right subtree

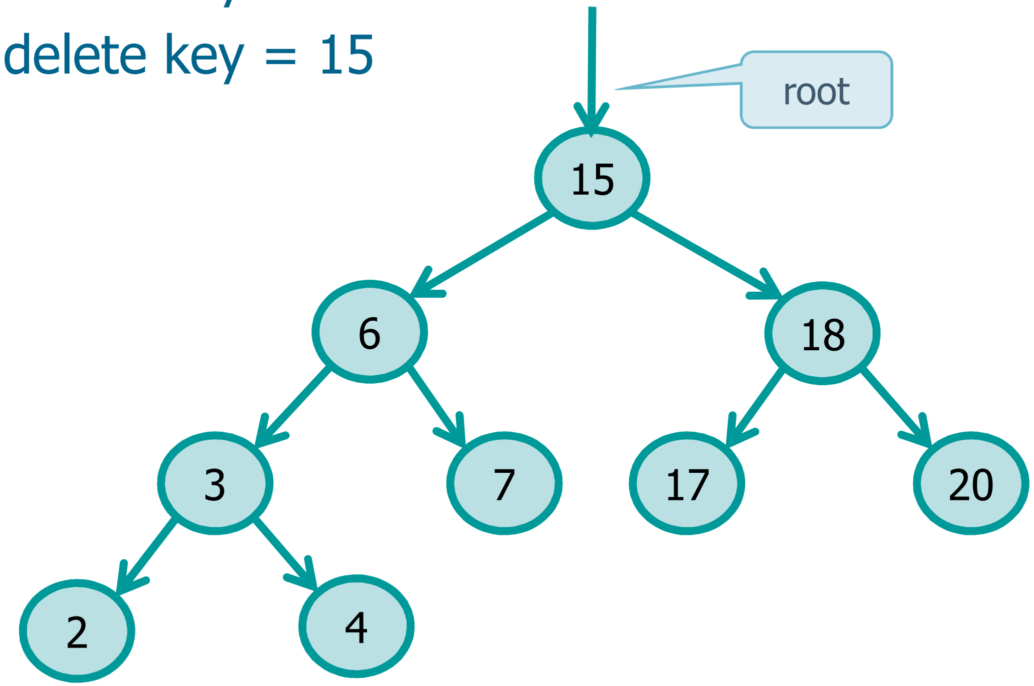
      and substitute the erased node with it

# Node Extract

- If the current node is not the one with the desired key
  - Recur onto the left sub-tree in the key is smaller than the node's key
  - Recur onto the right sub-tree in the key is smaller than the node's key

## Example

- Given the following BST delete key
  - key = 4
  - Then, delete key = 3
  - Then, delete key = 15



# Recursive implementation

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

  if (root == z)
    return (root);

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

Empty BST

Left recursion

Right recursion

# Recursive implementation

```
 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

Right child = NULL
First rule applied

Left child = NULL
First rule applied

Node with 2 children
Second rule applied
(find max into left sub-tree)

# Recursive implementation

Alternative solution: Find and delete minimum value into right sub-tree

Find and delete maximum value into left sub-tree

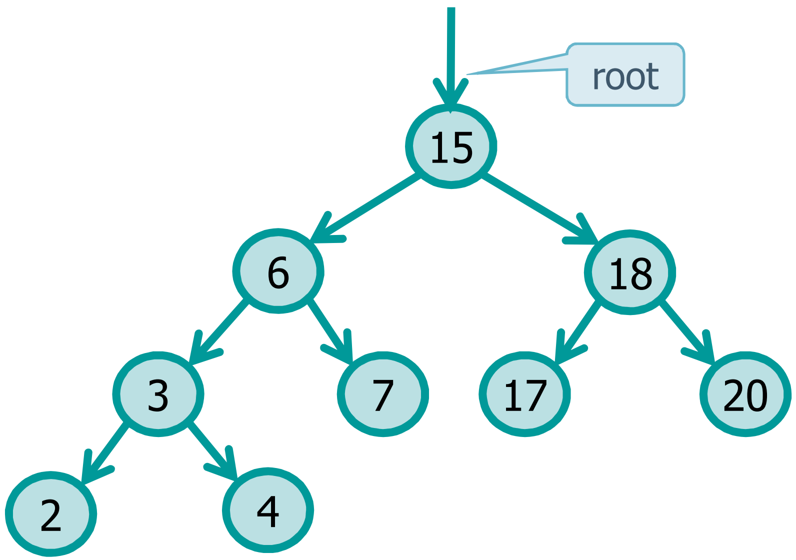
```
link max_delete_r (Item *x, link root, link z) {
  link tmp;

  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);
}
```

Node found: Free node and return pointer to left child

Recur until there is no right child

# Sorting and Median

- Given a BST
  - An in-order visit delivers keys in ascending order
  - Ascending order: 2 3 4 6 7 15 17 18 20



# Sorting and Median

- Given a BST
  - 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

Ascending order

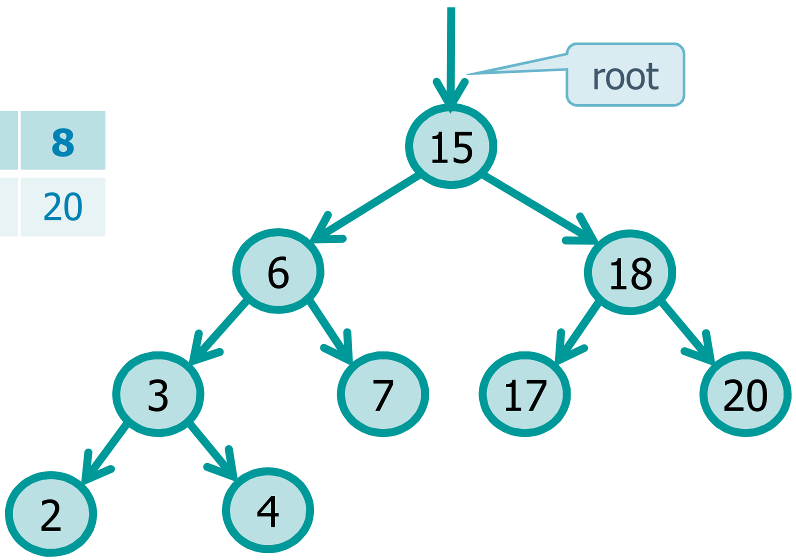
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 6 | **7** | 15 | 17 | 18 | 20 |

$\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{9+1}{2} \rfloor = 5$

→ position 5

→ element of index 4

→ 7 is the median key



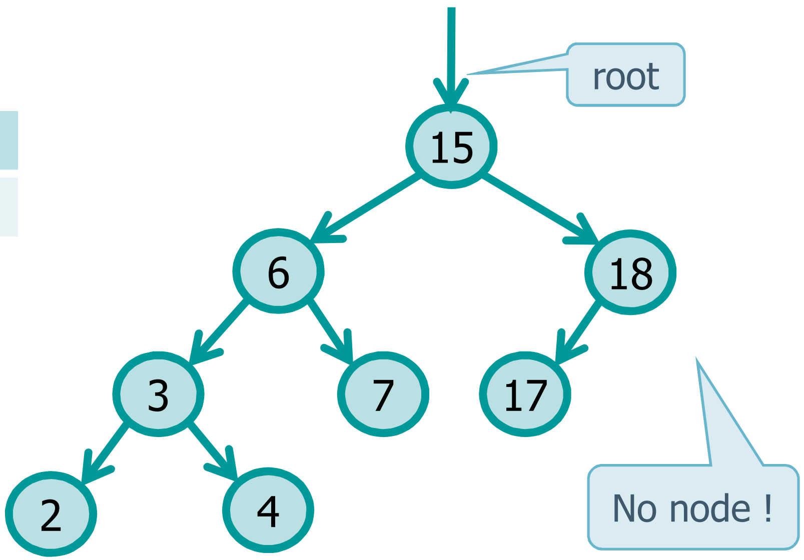
# Sorting and Median

- Given a BST
  - 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

Ascending order

| 0 | 1 | 2 | **3** | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | **6** | 7 | 15 | 17 | 18 |

$\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{8+1}{2} \rfloor = 4$
→ position 4
→ element of index 3
→ 6 is the median key



# Complexity

- Operations on BSTs have complexity
  - $T(n) = O(h)$
    - Where h is the height of the tree
- The height of a tree is equal to
  - Tree fully balanced with n nodes
    - Height $h = \alpha(\log_2 n)$
  - Tree completely unbalanced with n nodes
    - Height $h = \alpha(n)$
  - $O(\log n) \leq T(n) \leq O(n)$

# Exercise

- Given an initially empty BST perform the following insertions (+) and extractions (–)
  - +15 +16 +5 +3 +12 +20 +13 +8 +10 +23 +6 +7 –13 –16 – 5

## Exercise

- Suppose numbers between 1 and 1000 are stored in a BST, and we want to search for the key 363
- Which of the following sequences could be the sequence of nodes examined?
  - 2 252 401 398 330 344 397 363
  - 924 220 911 244 898 258 362 363
  - 925 202 911 240 912 245 363
  - 2 399 387 219 266 382 385 278 363
  - 935 278 347 621 392 358 363

## Exercise

- Suppose numbers between 1 and 1000 are stored in a BST, and we want to search for the key 363
- Which of the following sequences could be the sequence of nodes examined?
  - 2 252 401 398 330 344 397 363 — OK
  - 924 220 911 244 898 258 362 363 — OK
  - 925 202 911 240 912 245 363 — NO
  - 2 399 387 219 266 382 385 278 363 — NO
  - 935 278 347 621 392 358 363 — OK