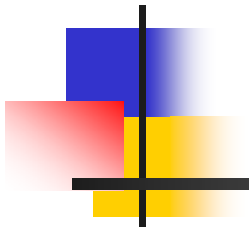
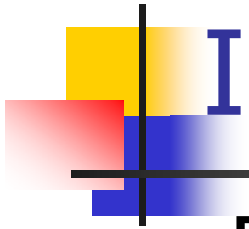


Binary Search Trees

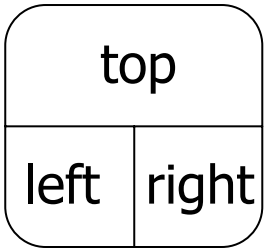
Alexandre David
B2-206

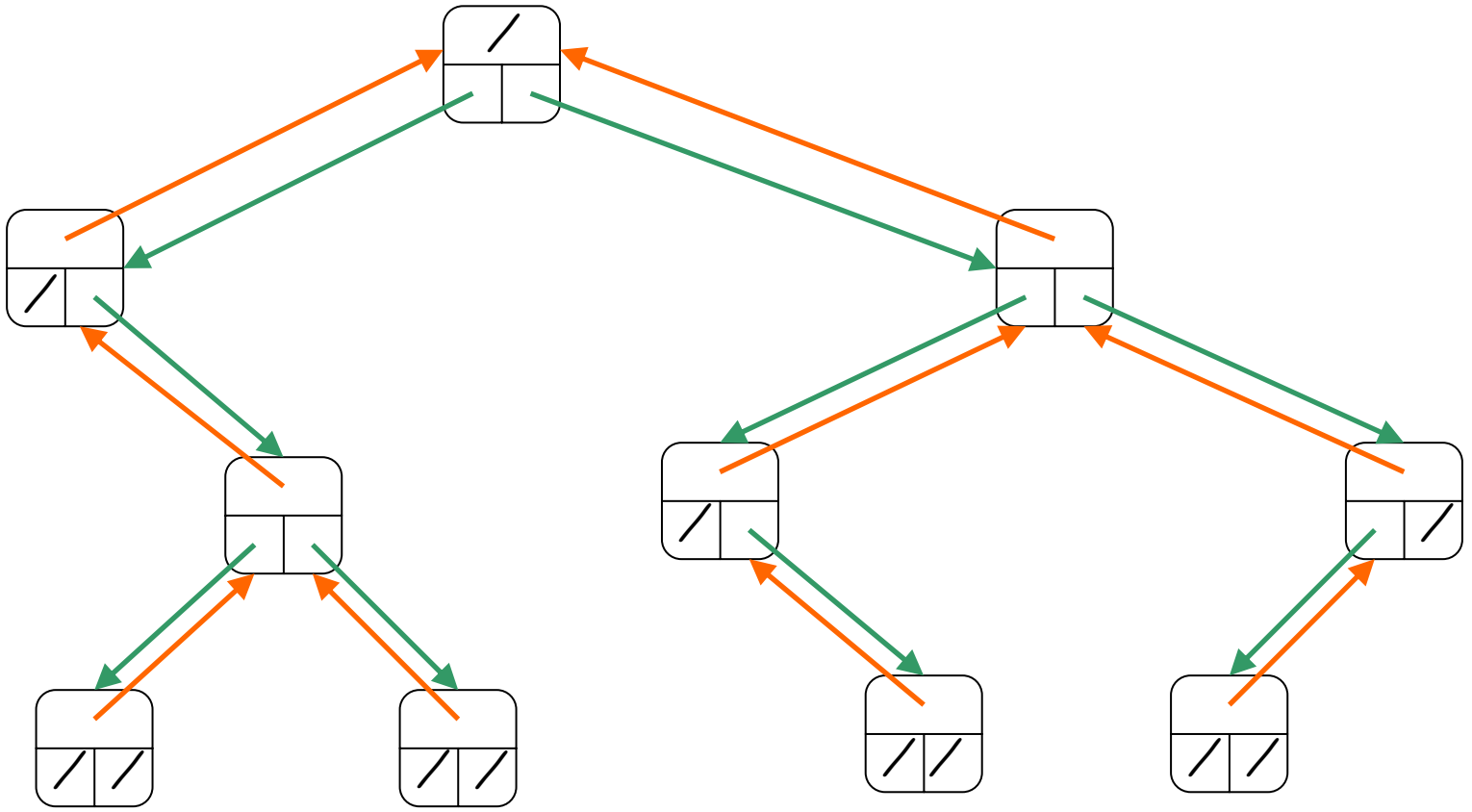


Introduction

- Binary tree: Every node in the tree at most 2 children.
- Support for many operations, e.g., search, min, max, predecessor, successor, insert, delete.
- Suitable as dictionaries and priority queues.
- Important parameter: *height of the tree*.
  - Basic operation proportional to the height.
  - E[height] of randomly built binary tree $\Theta(\lg n)$.

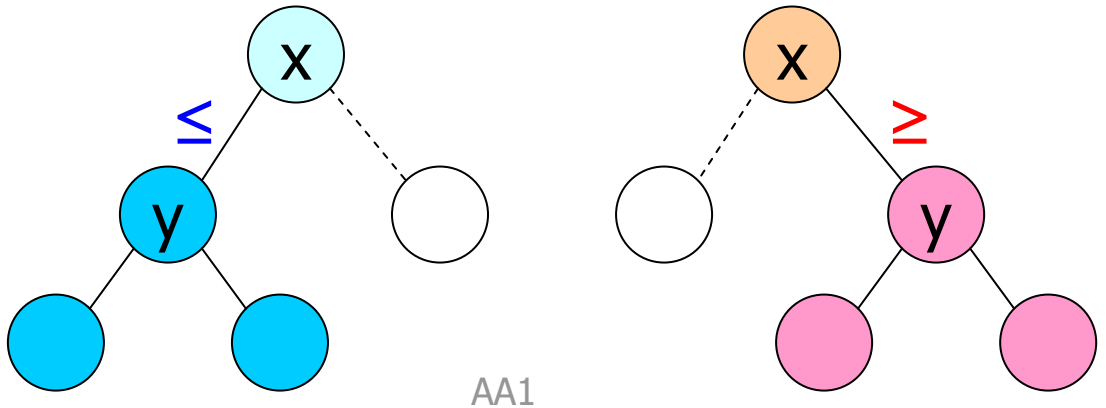
Binary Trees





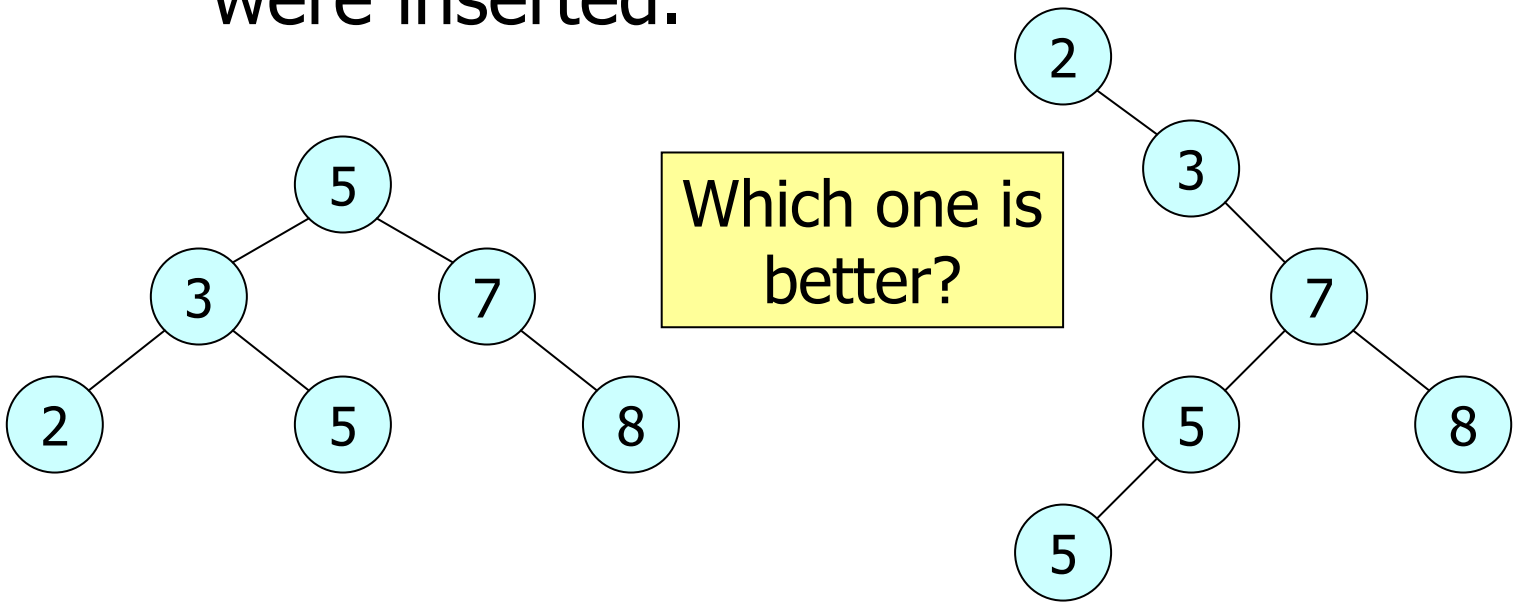
Binary Search Trees

- Binary tree with keys satisfying:
  - For a node *x* and a node *y* in the left sub-tree of *x*, $key(y) \leq key(x)$.
  - For a node *x* and a node *y* in the right sub-tree of *x*, $key(y) \geq key(x)$.
  - Similar to quick-sort.



Binary Search Trees

- Feature: Different binary search trees may represent the same set of values.
  - The tree depends on the order the values were inserted.



In-order Walk

- Print all the element in sorted order.
  - Correctness from the binary search tree properties.
  - Time $\Theta(n)$.

?

Proof: One call per node, n nodes $\Rightarrow$ n calls. $\Theta(1)$ per call $\Rightarrow$ $\Theta(n)$.

```
inorder_tree_walk(x):
if x ≠ NIL then
    inorder_tree_walk(left(x))
    print_key(x)
    inorder_tree_walk(right(x))
fi
```

Print all elements ≤ x.
Print x.
Print all elements ≥ x.

Pre/post-order Walks

```
preorder_tree_walk(x):
if x ≠ NIL then
    print_key(x)
    inorder_tree_walk(left(x))
    inorder_tree_walk(right(x))
fi
```

print before

```
postorder_tree_walk(x):
if x ≠ NIL then
    inorder_tree_walk(left(x))
    inorder_tree_walk(right(x))
    print_key(x)
fi
```

print after

Searching

- Running time is O(height), with E[height]=lg*n*.

```
tree_search(x,k):
while x ≠ NIL and k ≠ key(x) do
    if k < key(x) then
        x = left(x)
    else
        x = right(x)
    fi
done
return x
```

```
tree_min(x):
y = x
while x ≠ NIL do
    y = x
    x = left(x)
done
return y
```

```
tree_max(x):
y = x
while x ≠ NIL do
    y = x
    x = right(x)
done
return y
```

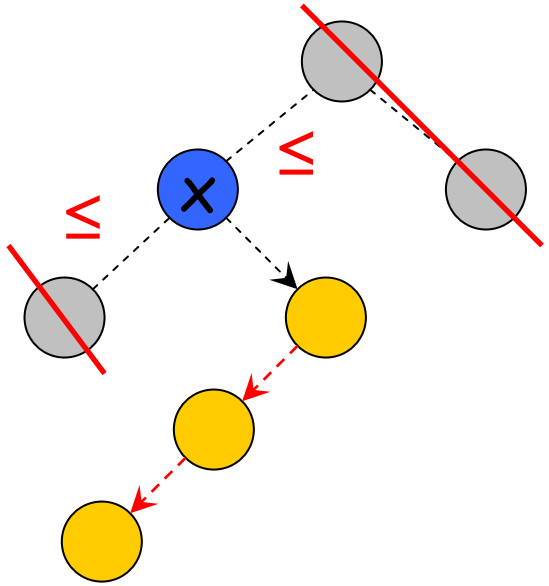
Similar to binary search for sorted arrays.

Why are these correct?

Successors
Sorted Enumeration

successor(x)?

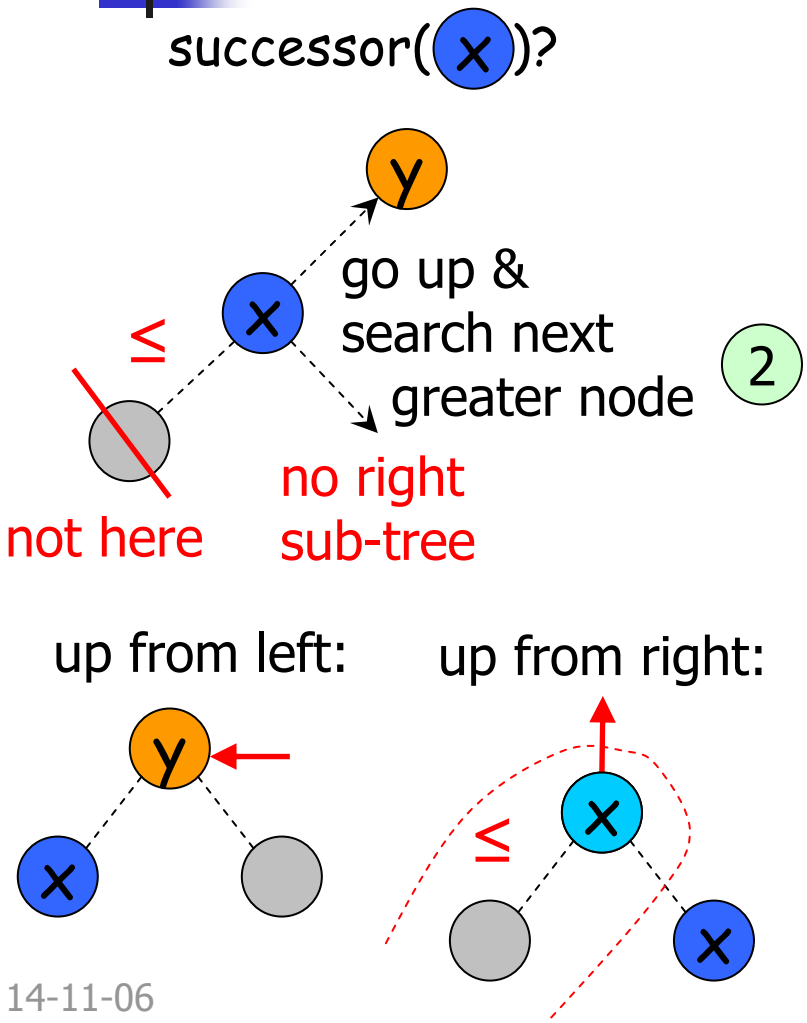
1. There is a right sub-tree, next greater element = min(right(x))



1.
```
tree_successor(x):
if right(x) ≠ NIL then
    return tree_min(right(x))
fi
y = parent(x)
while y ≠ NIL and x == right(y) do
    x = y
    y = parent(y)
done
return y
```

O(height)

Successors
Sorted Enumeration

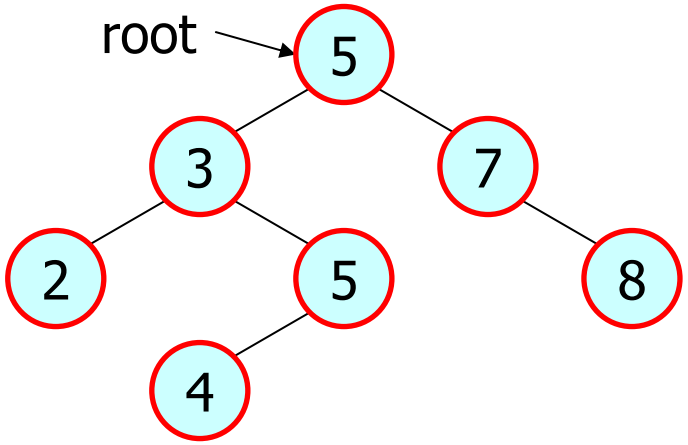



```
tree_successor(x):
if right(x) ≠ NIL then
    return tree_min(right(x))
fi
y = parent(x)
while y ≠ NIL and x == right(y) do
    x = y                          3
    y = parent(y)
done
return y
```

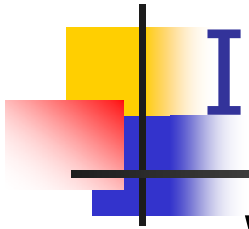
O(height)

Successors
Sorted Enumeration

```
tree_enumerate(root):
x = tree_min(root)
while x ≠ NIL do
    print(x)
    x = tree_successor(x)
done
```




```
tree_successor(x):
if right(x) ≠ NIL then
    return tree_min(right(x))
fi
y = parent(x)
while y ≠ NIL and x == right(y) do
    x = y
    y = parent(y)
done
return y
```



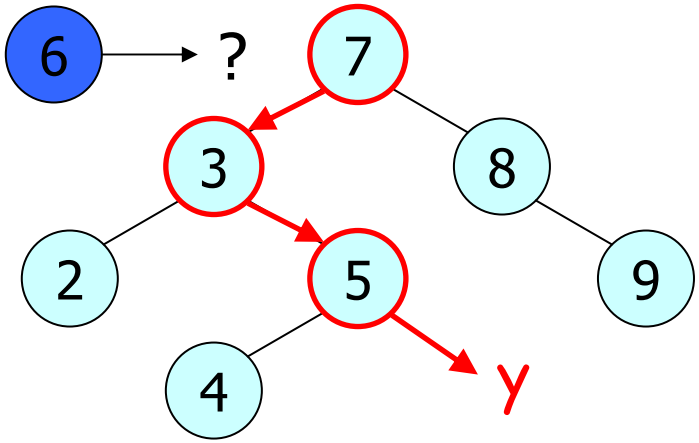
Insertion

- We insert an element $\Rightarrow$ change the tree but keep the *binary search tree property*.
- Idea: Add a leaf and fix the tree.

```
tree_insert(T,z):
y = search_leaf(T,z)
parent(z) = y
fix_child(y,z)
```

Insertion – Find Right Leaf

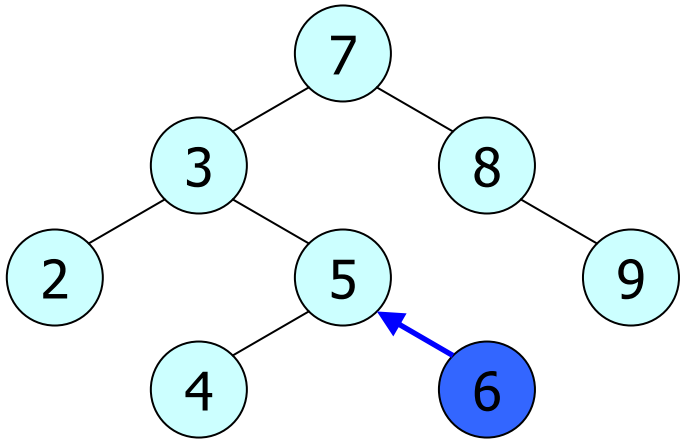
```
tree_insert(T,z):
y = search_leaf(T,z)
parent(z) = y
fix_child(y,z)
```




```
search_leaf(T,z):
y = NIL
x = root(T)
while x ≠ NIL do
    y = x
    if key(z) < key(x) then
        x = left(x)
    else
        x = right(x)
    fi
done
return y
```

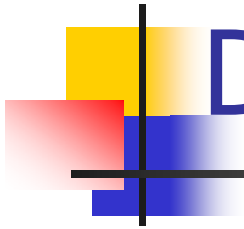
Insertion – Fix The Tree

```
tree_insert(T,z):
y = search_leaf(T,z)
parent(z) = y
fix_child(y,z)
```




```
fix_child(y,z):
if y = NIL then
    root(T) = z
else
    if key(z) < key(y) then
        left(y) = z
    else
        right(y) = z
    fi
fi
```

parent(new node) OK but update left or right child of parent!



Deletion

- Again we change the tree but we must keep the binary search tree property.

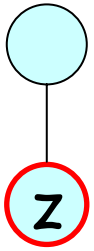
```
tree_delete(T,z):
if is_leaf(z) then
    remove(z)
else if has_1child(z) then
    splice_out(z)
else if has_2children(z) then
    y = tree_successor(T,z)
    remove(y)
    key(z) = key(y) and copy data
fi
```

Book:
Optimized version,
don't panic!

O(height)

(a)

"Optimized" Deletion




```
tree_delete(T,z):
if left(z) == NIL or
   right(z) == NIL then
   y = z
else                          y == z
   y = tree_successor(z)
fi
if left(y) ≠ NIL then
   x = left(y)
else
   x = right(y)               x=NIL
fi
if x ≠ NIL then
   parent(x) = parent(y)
fi
if (parent(y) == NIL then
   root(T) = x                last node?
else
   if y == left(parent(y)) then
      left(parent(y)) = x
   else
      right(parent(y)) = x
   fi                         update "right"
fi                            child of parent(y):
if y ≠ z then                 y is removed
   key(z) = key(y)
fi
return y
```

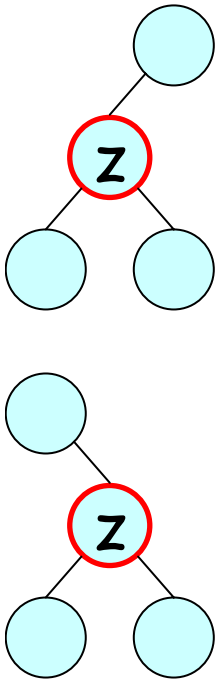
(b)

"Optimized" Deletion


```
tree_delete(T,z):
if left(z) == NIL or
   right(z) == NIL then
   y = z
else                          y == z
   y = tree_successor(z)
fi
if left(y) ≠ NIL then
   x = left(y)
else                          x=child(z)
   x = right(y)
fi
if x ≠ NIL then               may splice-out y
   parent(x) = parent(y)
fi
if (parent(y) == NIL then
   root(T) = x                last node?
else
   if y == left(parent(y)) then
      left(parent(y)) = x
   else
      right(parent(y)) = x
   fi                         update "right"
fi                            child of parent(y):
if y ≠ z then                 y is removed
   key(z) = key(y)
fi
return y
```

(c)

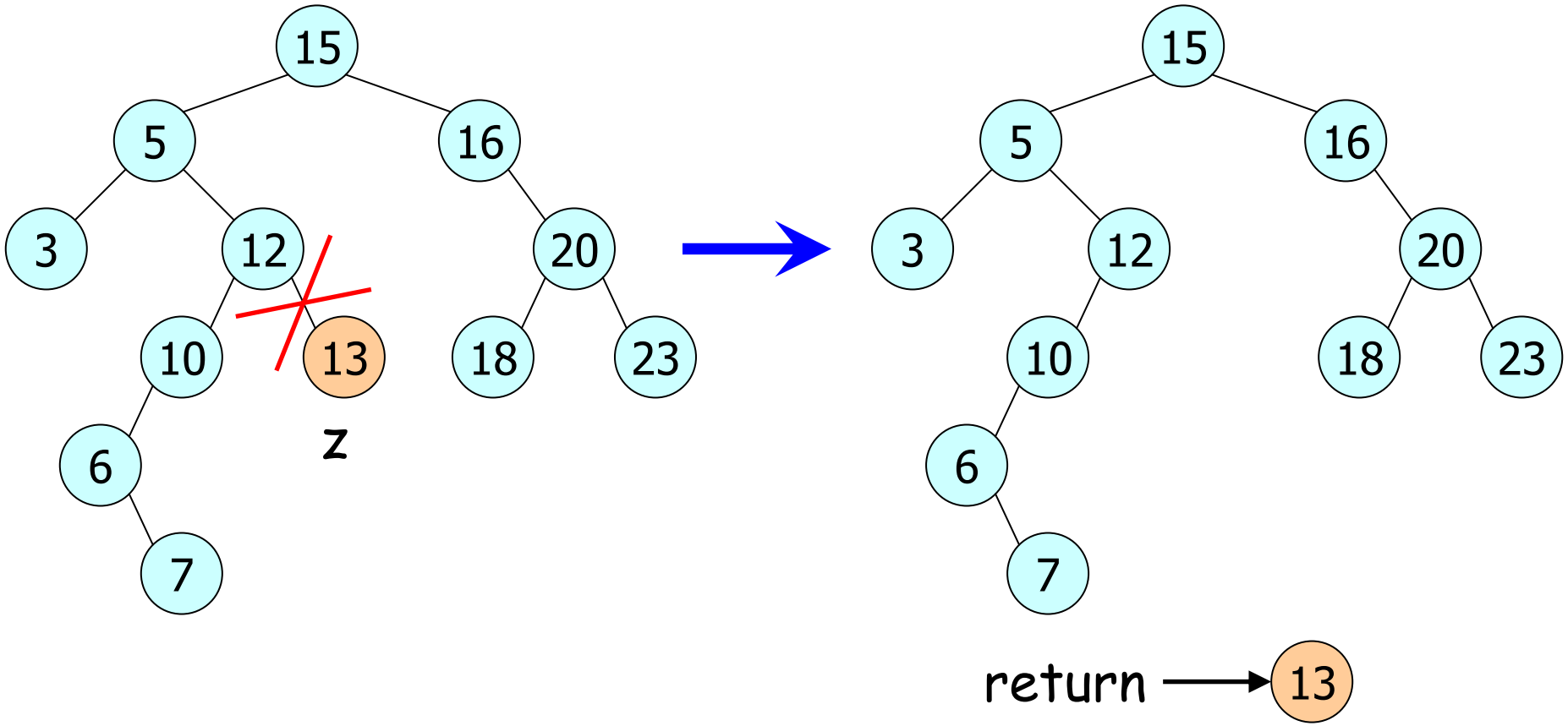
"Optimized" Deletion



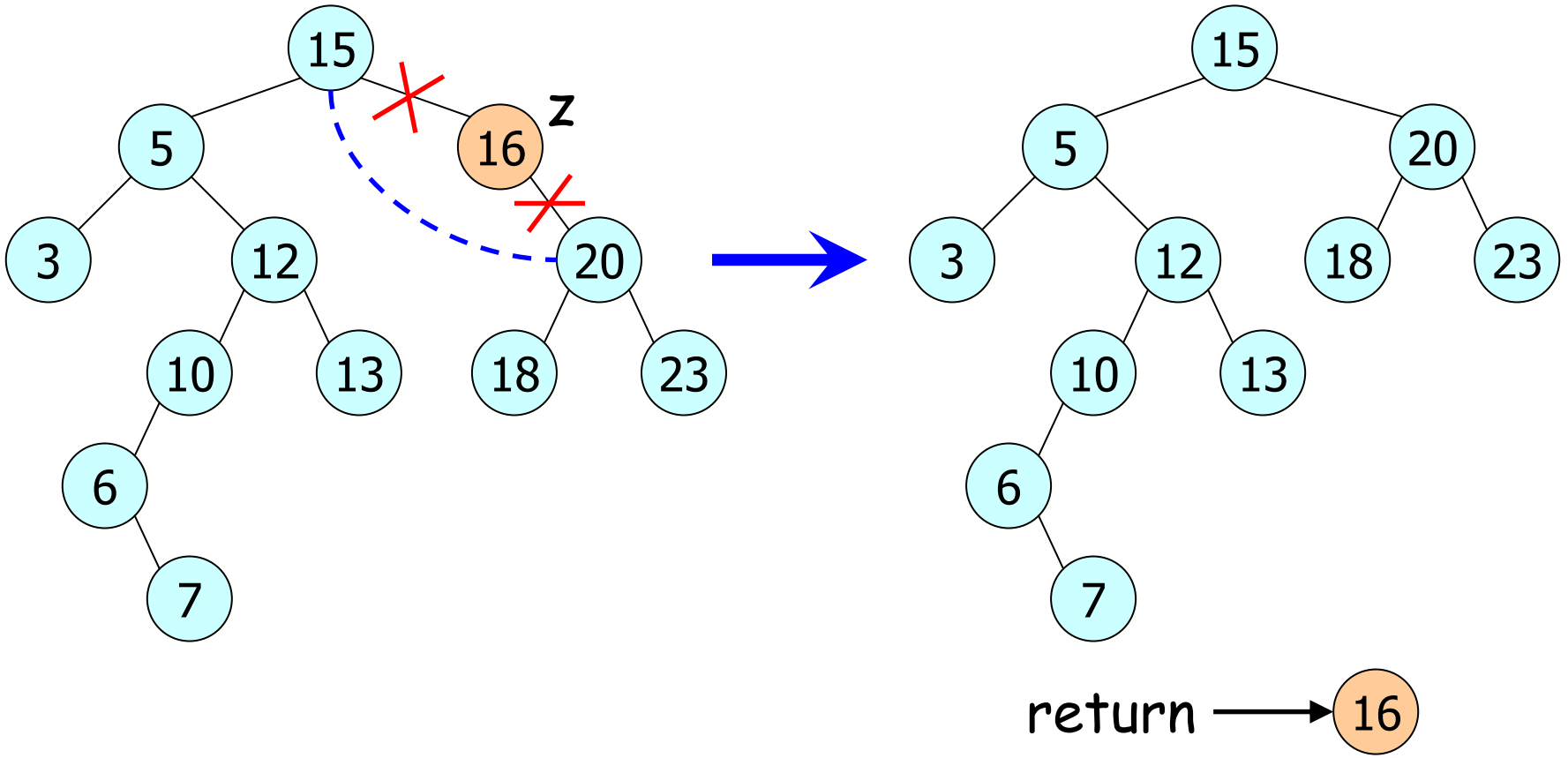

```
tree_delete(T,z):
if left(z) == NIL or
   right(z) == NIL then
   y = z
else
   y = tree_successor(z)
fi
if left(y) ≠ NIL then
   x = left(y)
else
   x = right(y)
fi
```

```
if x ≠ NIL then                    may splice-out y
   parent(x) = parent(y)
fi
if (parent(y) == NIL then
   root(T) = x                     last node?
else
   if y == left(parent(y)) then
      left(parent(y)) = x
   else
      right(parent(y)) = x
   fi
fi                                 update "right"
if y ≠ z then                      child of parent(y):
   key(z) = key(y)                 y is removed
fi
return y
```

Example (a)



Example (b)



Example (c)