Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

CAIDAS WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

# Binary Search Tree

Prof. Dr. Goran Glavaš

16.11.2023

# Content

- Recap & Analysis: Dynamic sets and operations
- Binary Search
- Binary Search Tree

# Search

**Search / find value**

Given a dynamic set **S** and a query value x, `SEARCH` is the problem of finding out whether x ∈ **S**. It is among the most basic/fundamental problems in CS, and one that needs to be solved in almost all more complex problems. Solving it efficiently is thus paramount.

- **Three basic operations** for manipulating the content of **dynamic sets**

1. **INSERT – add new element to the dynamic set**
   - In general, in no particular order
   - Constraints on order or positioning of elements: stacks, queues, heaps, ...
2. **SEARCH –** answer the question „is element X in the set"?
3. **DELETE –** remove an element from the set
   - In general, any element from the set can be removed
   - Constraints on order of element removal – stacks, queues, heaps, ...

# Search

**Search / find value**

Given a dynamic set **S** and a query value x, **SEARCH** is the problem of finding out whether x ∈ **S**. It is among the most basic/fundamental problems in CS, and one that needs to be solved in almost all more complex problems. Solving it efficiently is thus paramount.

- Let's add two more:
  - Finding a **minimal** or **maximal** value in the set (**max/min**)
  - Finding the closest smaller (predecessor) or larger (successor) value for some x (**pred/succ**)

- **Data structures** for **dynamic sets** that we've already examined

**1. (unsorted) array**

**2. (unsorted) linked list**

**3. hash table**

- Some of these basic operations become much easier (**faster** :) for a **sorted array**

# Insert, Delete, Search, Min/Max, Pred/Succ

| | Runtime | | | | |
|---|---|---|---|---|---|
| **Data struct.** | **Search** | **Insert** | **Delete*** ** | **Min/Max** | **Pred/Succ** |
| **Array** | | | | | |
| **Linked List** | | | | | |
| **Hash Table** | | | | | |

- ***`Delete` here refers only to the actual deletion of the element once it is found (via `Search`), and the associated steps for maintaining the corresponding data structure
  - The complexity of the search operation needed to find the element in the data structure is not included
- **Assuming the elements of the array must always be **contiguous** in memory
- *Assuming simple uniform hashing and a fixed $\alpha = n/m$ ratio (i.e., re-hashing with a bigger table (bigger m) if $n$ increases), maintaining constant average length of collision chains

# Insert, Delete, Search, Min/Max, Pred/Succ

| | Runtime | | | | |
|---|---|---|---|---|---|
| Data struct. | Search | Insert | Delete | Min/Max | Pred/Succ* |
| Array | O(n) | O(1) | O(n) | O(n) | O(n) |
| Linked List | O(n) | O(1) | O(1) | O(n) | O(n) |
| Hash Table | O(1) | O(1) | O(1) | not possible | not possible |
| Sorted Array | ? | ? | ? | O(1) | O(1) |

- **What if we had a sorted array?**
  - What would then be the runtime for each of these operations?
  - Assuming that after each operation, the array needs to remain sorted

- ***Pred/Succ** of some given value x is **O(1)** only if we assume that finding x (i.e., its position in the array, `Search`) is not part of the `Pred/Succ` operation
  - I.e., we know the position of x

# Content

- Recap & Analysis: Dynamic sets and operations
- Binary Search
- Binary Search Tree

# Sorted array

- **Reminder**: initial sorting of an unsorted array is done in **O(n log n)**
- Operations in **sorted array** (must remain sorted afterwards):
  - `Insert(A,x)`
    - Put x to the end of the array and re-sort the array? → **O(n log n)**
    - We can do better than that: even if we assume (suboptimal) `Search` as part of `Insert`
    - **Q:** Running time of the algorithm on the right?

```
Insert(A, x):
  A.Length = A.Length + 1
  pos = -1
  for i = 0 to A.Length - 2
     if x < A[i]  # Q: if ≤ instead?
     pos = i
     break # exits from (stops) the loop
  if pos == -1
    A[A.Length - 1] = x
  else
    prev = A[position]
    A[pos] = x
    for j = pos+1 to A.Length-2
       tmp = prev
       prev = A[j]
       A[j] = tmp
```

# Sorted array

- **Reminder**: initial sorting of an unsorted array is done in **O(n log n)**

- Operations in **sorted array** (must remain sorted afterwards):
  - `Delete(A,x)`
    - Deleting any element from the sorted array → the array remains sorted
    - But it becomes discontiguous → fix for that

    - Q: Running time of the algorithm on the right?

```
Delete(A, x):
  position = -1
  for i = 0 to A.Length - 1
    if x == A[i] # Q: if ≤ instead?
    pos = i
    break # exits from (stops) the loop
  if pos >= 0
    next = A[A.Length - 1]
    for j = A.Length - 2 downto pos
       tmp = A[j]
       A[j] = next
       next = tmp
    A.Length = A.Length - 1
```

# Sorted Array: Binary Search

- If the array is **sorted**, we can `Search` for a value in **sublinear** time
  - Think of a divide-and-conquer algorithm that could do that?
  - **Recursion:** core principle the same as in merge sort
  - Divide the array into two and search in the subarrays
    - Sequentially, the second subarray only searched if the element not found in first

```
binary_search(A, x, p, r):
    n = r - p + 1
    if n == 1 # recursion stopping condition
      if A[p]== x
        return p
      else
        return -1
    # Q: why don't we have „else" here?
    if n % 2 == 1 # odd number of elements
      q = p + n//2
    else # even number of elements
      q = p + n/2 - 1

    if x <= A[q]
      return binary_search(A, x, p, q)
    else
      return binary_search(A, x, q+1, r)
```

# Binary Search

- Let's *slightly* modify our binary search
  - We modify the „division" part to be more like the division from quick sort than the division in merge sort
  - Worst case runtime same, but „better constants"
  - The division in this version of the `binary_search` directly „builds" one path of a binary tree top to bottom

```
binary_search(A, x, p, r):
  n = r - p + 1
  if n % 2 == 1 # odd number of elements
    q = p + n//2
  else # even number of elements
    q = p + n/2 - 1

  if A[q] == x
    return q
  else
    if n == 1 # recursion stopping condition
      return -1
    else
      if x < A[q]
        binary_search(A, x, p, q-1)
      else # x > A[q]
        return binary_search(A, x, q+1, r)
```

# Binary Search

- The division in this version of the `binary_search` effectively „operates" on a binary tree, top to bottom

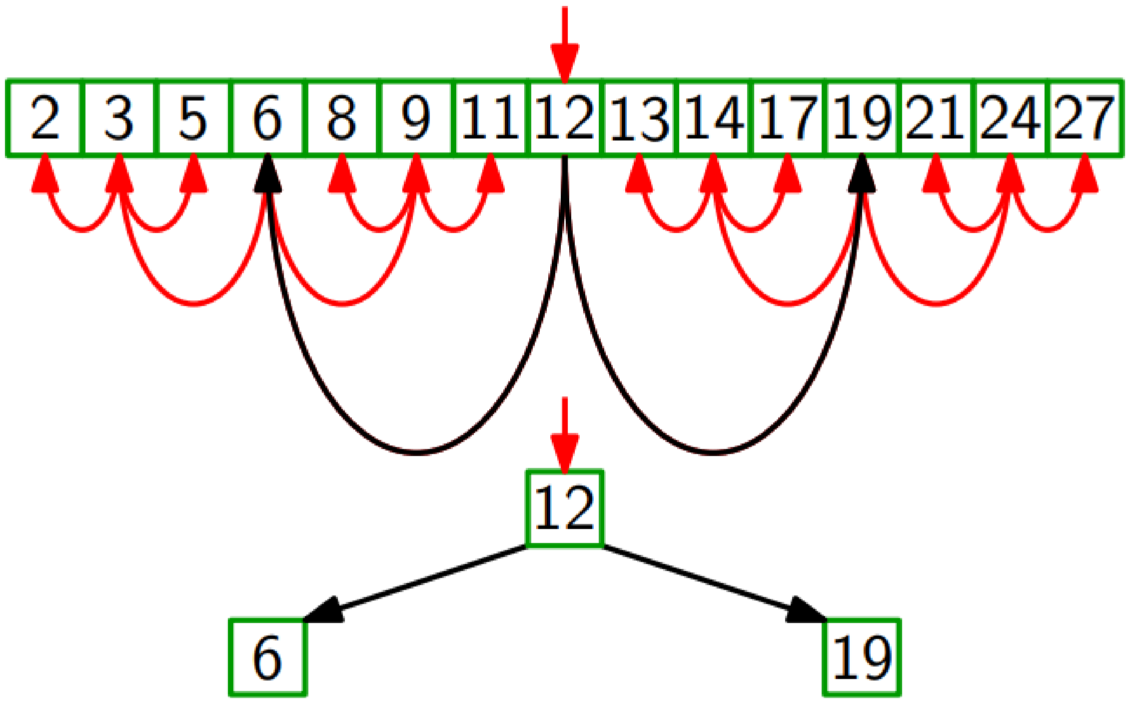


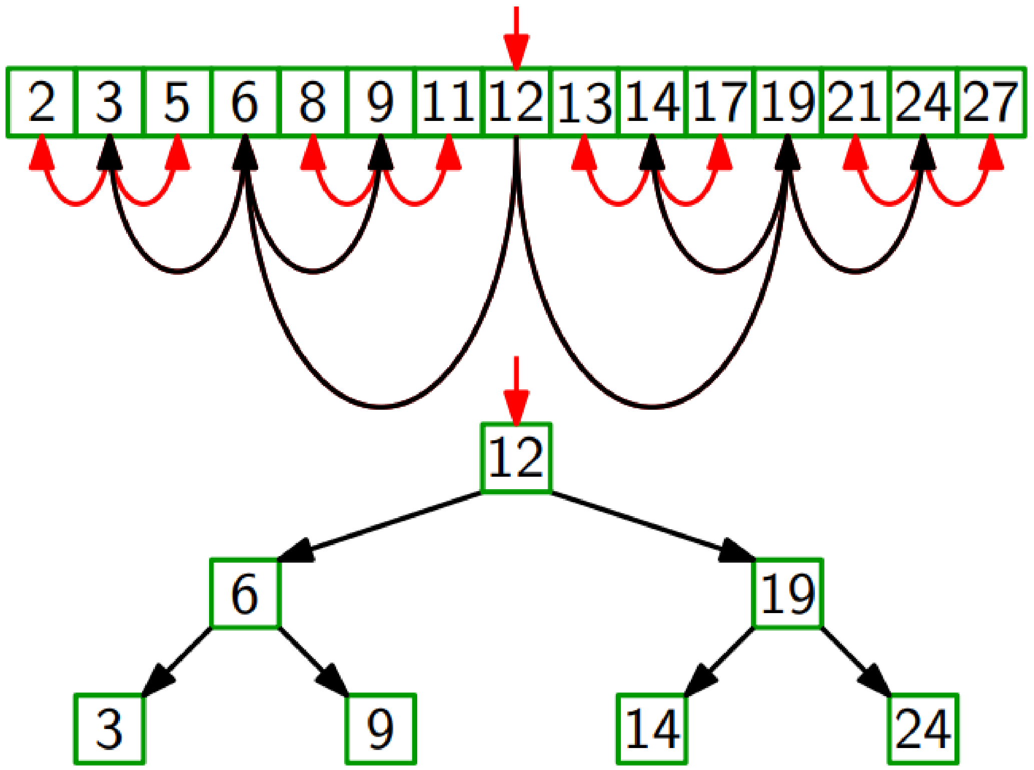

Image taken from ADS slides of Prof. Alexander Wolff

```
binary_search(A, x, p, r):
    n = r - p + 1
    if n % 2 == 1 # odd number of elements
      q = p + n//2
    else # even number of elements
      q = p + n/2 - 1

    if A[q] == x
      return q
    else
      if n == 1 # recursion stopping condition
        return -1
      else
        i = binary_search(A, x, p, q-1)
        if i >= 0
          return i
        else
          return binary_search(A, x, q+1, r)
```

# Binary Search

- The division in this version of the `binary_search` effectively „operates” on a binary tree, top to bottom

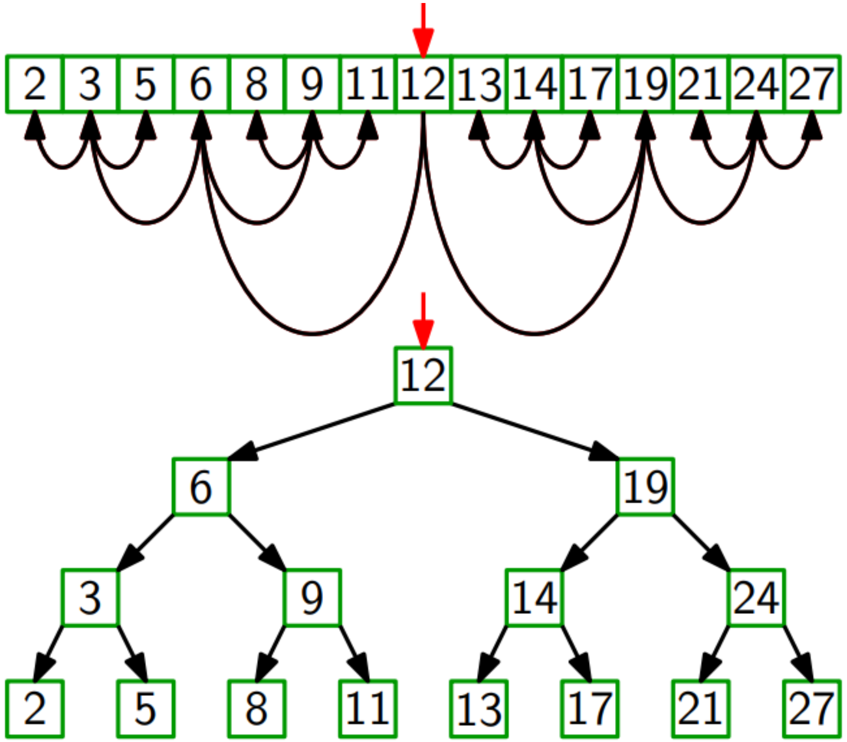



```
binary_search(A, x, p, r):
    n = r - p + 1
    if n % 2 == 1 # odd number of elements
      q = p + n//2
    else # even number of elements
      q = p + n/2 - 1

    if A[q] == x
      return q
    else
      if n == 1 # recursion stopping condition
        return -1
      else
        i = binary_search(A, x, p, q-1)
        if i >= 0
          return i
        else
          return binary_search(A, x, q+1, r)
```

# Binary Search

- The division in this version of the `binary_search` effectively „operates" on a binary tree, top to bottom



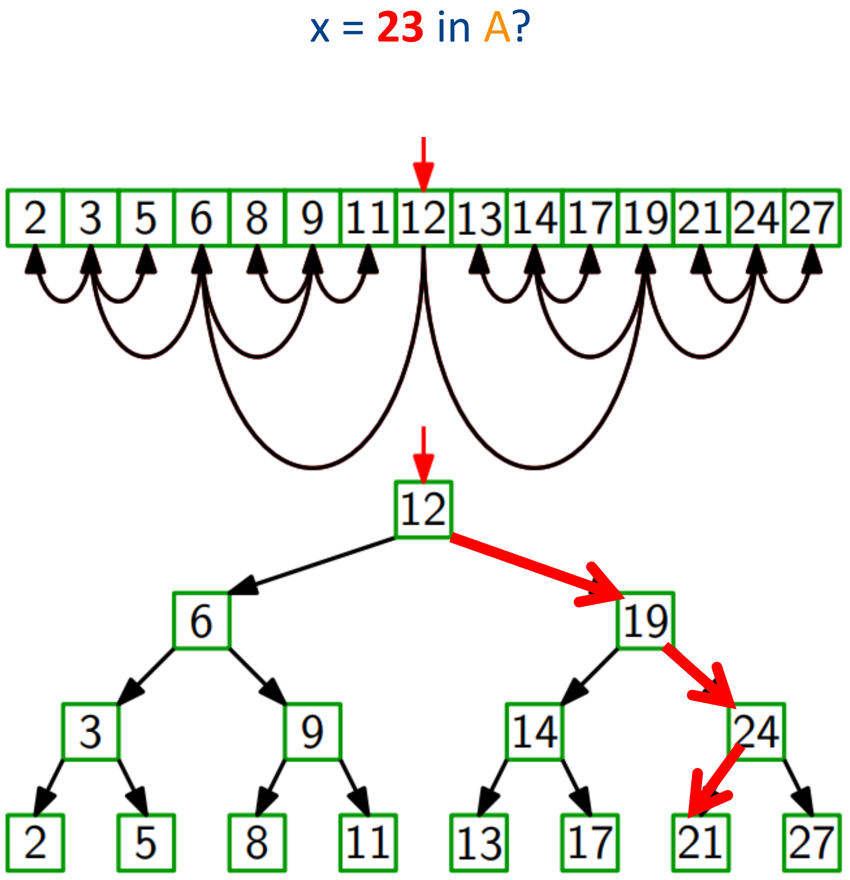

```
binary_search(A, x, p, r):
    n = r - p + 1
    if n % 2 == 1 # odd number of elements
      q = p + n//2
    else # even number of elements
      q = p + n/2 - 1

    if A[q] == x
      return q
    else if n == 1 # recursion stopping condition
      return -1
    else if x < A[q]
      return binary_search(A, x, p, q-1)
    else
      return binary_search(A, x, q+1, r)
```

# Binary Search: Runtime

- **Running time** of `Search` implemented via `binary_search`?
  - Array A with n elements

- **Worst case scenario**: x not in A
  - `binary_search` will proceed towards **one complete path** (root to leaf) of a binary tree with n nodes
  - What is the depth/height of the **balanced** binary tree with n nodes?
  - Worst case runtime of binary search is **O(log n)**

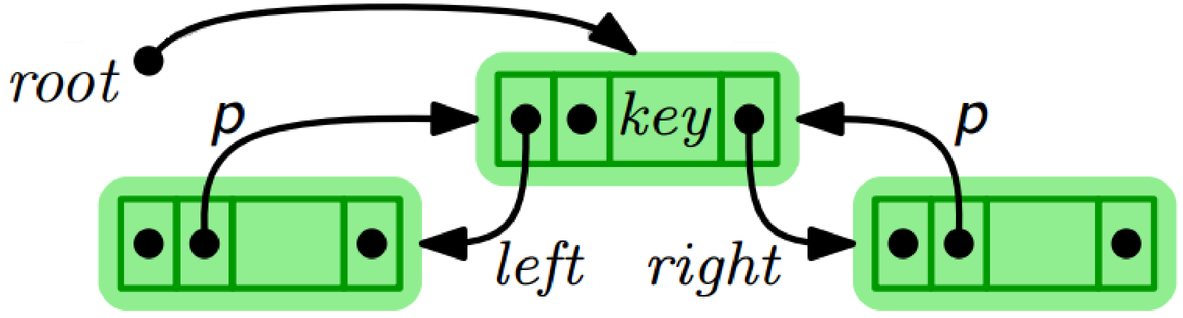


# Content

- Recap & Analysis: Dynamic sets and operations
- Binary Search
- Binary Search Tree

# Binary Tree

- Recap: static arrays not ideal for (very) dynamic sets
  - Frequent memory reallocations are expensive
- Implement flexible **binary tree** without any fixed-allocated memory
  - Something that is for the ADS „binary tree” what **linked list** is for ADS „list”
  - We need „**nodes**” with pointers

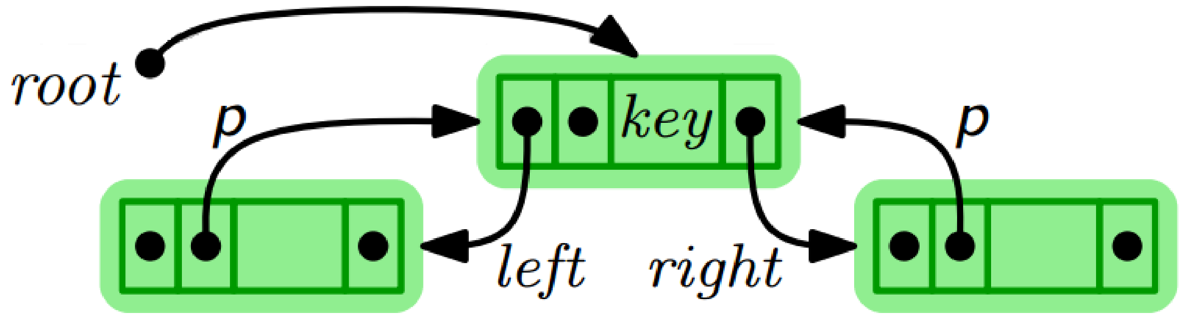


# Binary Tree

Structure/type (or in OOP, Class)

**`Node`**:

- `.key` – search/insert/delete – node's identifier for all operations
- `.data` – arbitrary „satellite" data (<u>not used</u> in any way for tree organization)
- `.parent` – pointer to the parent node
- `.left` – pointer to the left child
- `.right` – pointer to the right node



# Binary Search Tree

- If we want efficient search – like with sorted array and binary search – then we have to maintain the **binary search property** of the tree

**Binary search tree property**

For each non-leaf node x of a binary tree the following **binary search tree property** has to be satisfied: (1) for every node y in the left subtree of x: **y.key ≤ x.key**; **and** (2) for every node y in the right subtree of x: **y.key ≥ x.key**;

- **Q:** How to process tree elements in sorted order (or create a sorted array) in a tree that satisfies the binary search tree property?

# Binary Search Tree: Inorder Walk

- **Q:** How to process tree elements in sorted order (or create a sorted array) in a tree that satisfies the binary search tree property?
- With a recursive **inorder tree walk**
  - **Variant 1:** just prints the keys in sorted order

```
inorder_walk(x)  # x is instance of type „node"
  if x != null # a leaf node would have empty pointers
    inorder_walk(x.left)
    print(x.key)
    inorder_walk(x.right)
```

```
Calling it on the root
 inorder_walk(T.root)
```

- **Q**: What is the **runtime** of `inorder_walk`?

# Binary Tree: Inorder Walk

- **Q:** How to process tree elements in sorted order (or create a sorted array) in a tree that satisfies the binary search tree property?
- With a recursive **inorder tree walk**
  - **Variant 2:** create sorted array from the tree

```
inorder_walk(x, A)  # x is instance of type „node”
  if x != null  # a leaf node would have empty pointers
    inorder_walk(x.left, A)
    A.Length = A.Length + 1
    A[A.Length - 1] = x.key
    inorder_walk(x.right)
```

```
inorder_array(T)
  A.Size = T.Size
  A.Length = 0
  inorder_walk(T.root, A)
  return A
```

# Querying a Binary Search Tree

- Let's revisit our operations

| | Runtime | | | | |
|---|---|---|---|---|---|
| Data struct. | Search | Insert | Delete | Min/Max | Pred/Succ* |
| Array | O(n) | O(1) | O(n) | O(n) | O(n) |
| Linked List | O(n) | O(1) | O(1) | O(n) | O(n) |
| Hash Table | O(1) | O(1) | O(1) | not possible | not possible |
| Sorted Array | O(log n) | O(n) | O(n) | O(1) | O(1) |
| Binary Search Tree | ? | ? | ? | ? | ? |

# Binary Search Tree: **Search** and **Min/Max**

### Search

```
tree_search(x, k)
  if x == null or x.key == k
    return x # if null is returned, not found

  if k < x.key
      return tree_search(x.left, k)
  else
      return tree_search(x.right, k)
```

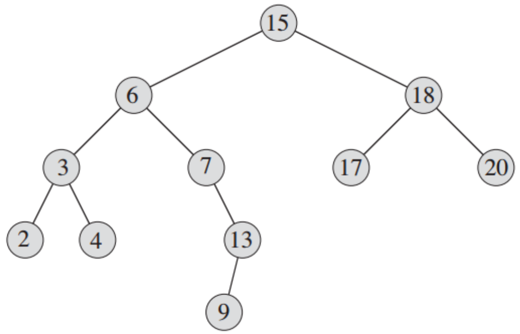
**Q: Runtime of search?**

### Min / Max

```
tree_min(x)
  while x.left != null
    x = x.left
  return x

tree_max(x)
  while x.right != null
    x = x.right
  return x
```

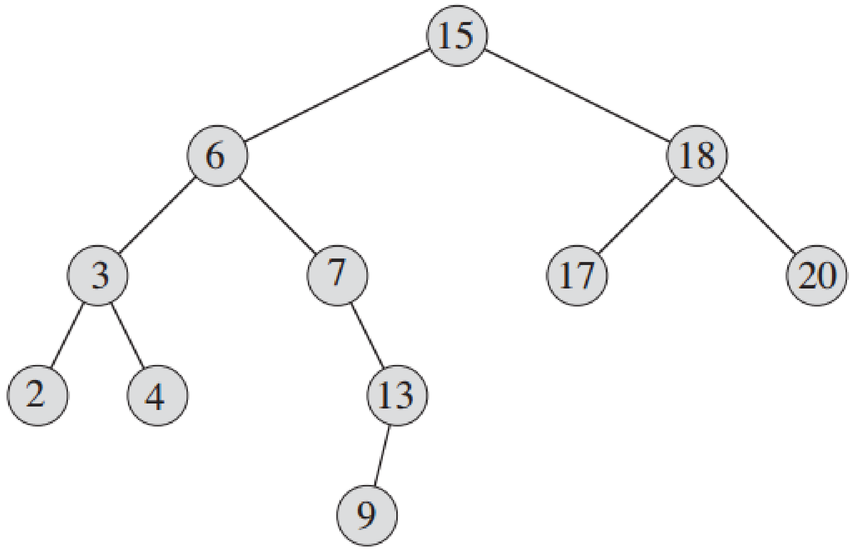
**Q: Runtime of min/max?**



# Binary Search Tree: **Successor**

- **Assumption**: no duplicate values in the tree (i.e., in the dynamic **set**)
- **Successor** of x = smallest y (y.key) in T such that **y.key > x.key**

- Where is the **successor** of x in the tree
  - It is the **minimum of its right subtree**
  - What if `x.right` is `null`?

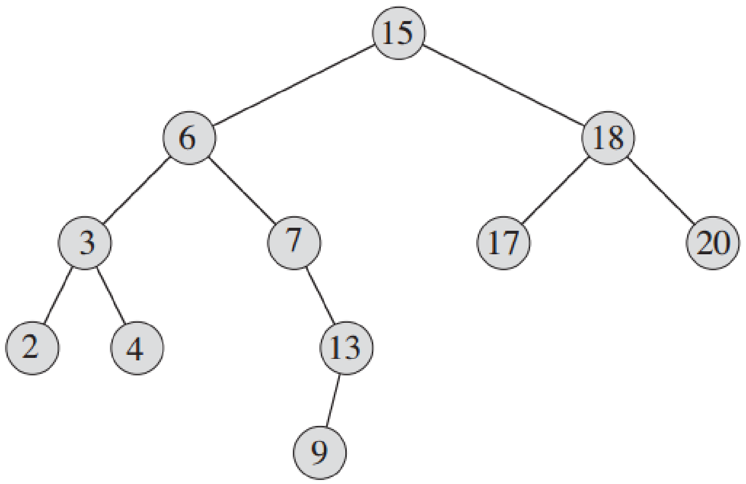


# Binary Search Tree: **Successor**

- Where is the **successor** of x in the tree
  - It is the **minimum of its right subtree**
    - successor(x) if x.key = 6?
  - What if `x.right` is `null`?
    - successor(x) if x.key = 13?

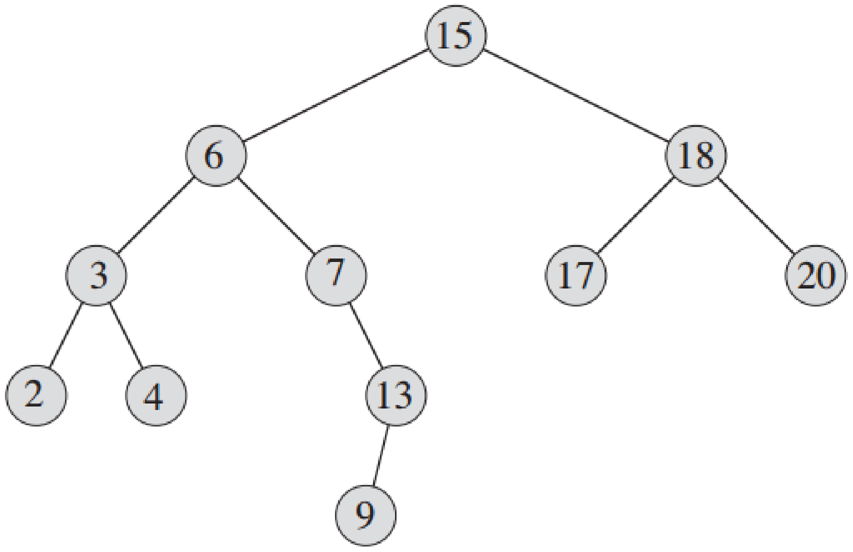
```
successor(x)
  if x.right != null
     return tree_min(x.right)

  par = x.parent
  while par != null and x == par.right
    x = par
    par = x.parent
  return par
```



# Binary Search Tree: **Predecessor**

- **Assumption**: no duplicate values in the tree (i.e., in the dynamic **set**)
- **Predecessor** of x = largest y (y.key) in T such that **y.key < x.key**
- Where is the **predecessor** of x in the tree?
  - It is the **maximum** of its left subtree
  - What if **`x.left`** is **`null`**?
- Write the **pseudocode** for finding the predecessor of x



# Binary Search Tree: **Insertion**

- Insert a new node x with key k into the Tree T; initially:
  - `x.key = k`
  - `x.left = x.right = null`
  - `x.parent = null`
- x needs to be inserted into the **correct place** in the tree
  - After insertion, the tree must **still satisfy** the binary search tree property

```
tree_insert(T, x)
  y = T.root
  par = y.parent # null

  while y != null
    par = y
    if x.key < y.key
      y = y.left
    else
      y = y.right

  if par == null # T was empty
    T.root = x
  else
    x.parent = par
    if par.key < x.key
      par.right = x
    else
      par.left = x
```

# Binary Search Tree: **Deletion**

- We want to delete a node x from a binary search tree T
- Three cases: two simple, one more complex

1. **Node** without children (i.e., leaf node)
   - Simply set it's corresponding parent's pointer (left or right) to **`null`**

2. **Node with one child** (i.e., only one subtree, left or right)
   - „Bypass" the node x to be deleted – set the corresponding parent's pointer (left or right, depending on which child x is) to point to x's only child

# Binary Search Tree: **Deletion**

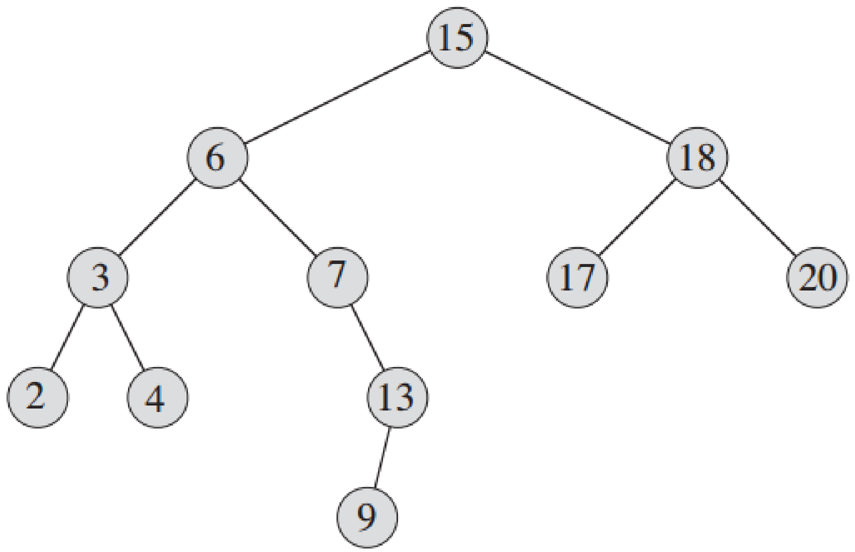
- We want to delete a node x from a binary search tree T
- Three cases: two simple, one more complex

3. **Node with both children** (<u>the trickiest case</u>)
   - Find x's successor y (in x's right subtree) and place y in x's place
   - y surely doesn't have left children (as it's the minimum of the x's right subtree), but it may have a right subtree
   - x's left child (subtree) becomes y's left child (subtree)
   - As for the x's right subtree (y's right subtree after switch) → **two subcases**, depending on whether y was directly the right child of x or not

# Binary Search Tree: **Deletion**

- **Case #1**:
  - `Delete` node with key 4
    - Right pointer of node with key 3 becomes **`null`**
  - `Delete` node with key 9
    - Left pointer of node with key 13 becomes **`null`**

- **Case #2**:
  - `Delete` node with key 13
    - Redirect the right ptr of node with key 7 to point to the node with key 9



# Binary Search Tree: **Deletion**

- **Case #1**:
  - `Delete` node with no children

```
del_case_1(x)
  par = x.parent
  if par.left == x
    par.left = null
  else
    par.right = null
```

- **Case #2**:
  - `Delete` node with one child

```
del_case_2(x)
  # determining which child x has, left or right
  child = null
  if x.left != null
    child = x.left
  else
    child = x.right
  # placing the child of x where x was
  par = x.parent
  if par.left == x
    par.left = child
  else
    par.right = child
```

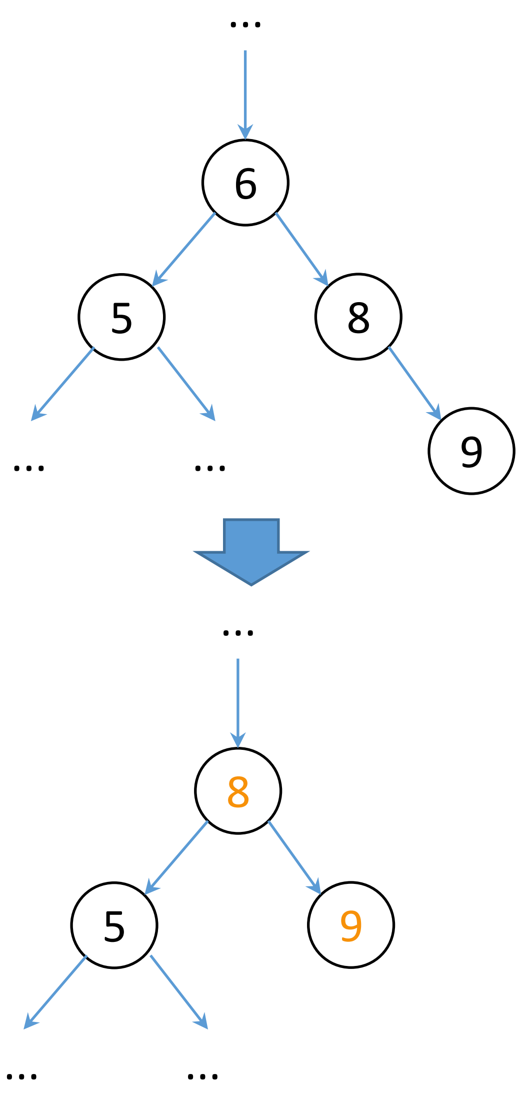
# Binary Search Tree: **Deletion**

- **Case #3**: delete node with two children
  - x – being removed, y – the successor

- **Subcase 3a**: successor is the right child, **y = x.right**
  - y has no left child
  - y may or may not have the right child

  - **Solution**: y replaces x, nothing else

Delete 6
succ = 8
(right child)



# Binary Search Tree: **Deletion**

- **Case #3**: delete node with two children
  - x – being removed, y – the successor

- **Subcase 3a**: successor is the right child, **y = x.right**
  - y has no left child
  - y may or may not have the right child

```
del_case_3a(x)
  par = x.parent
  if par.left == x
    par.left = x.right
  else
    par.right = x.right
```

  - **Solution**: y replaces x, nothing else

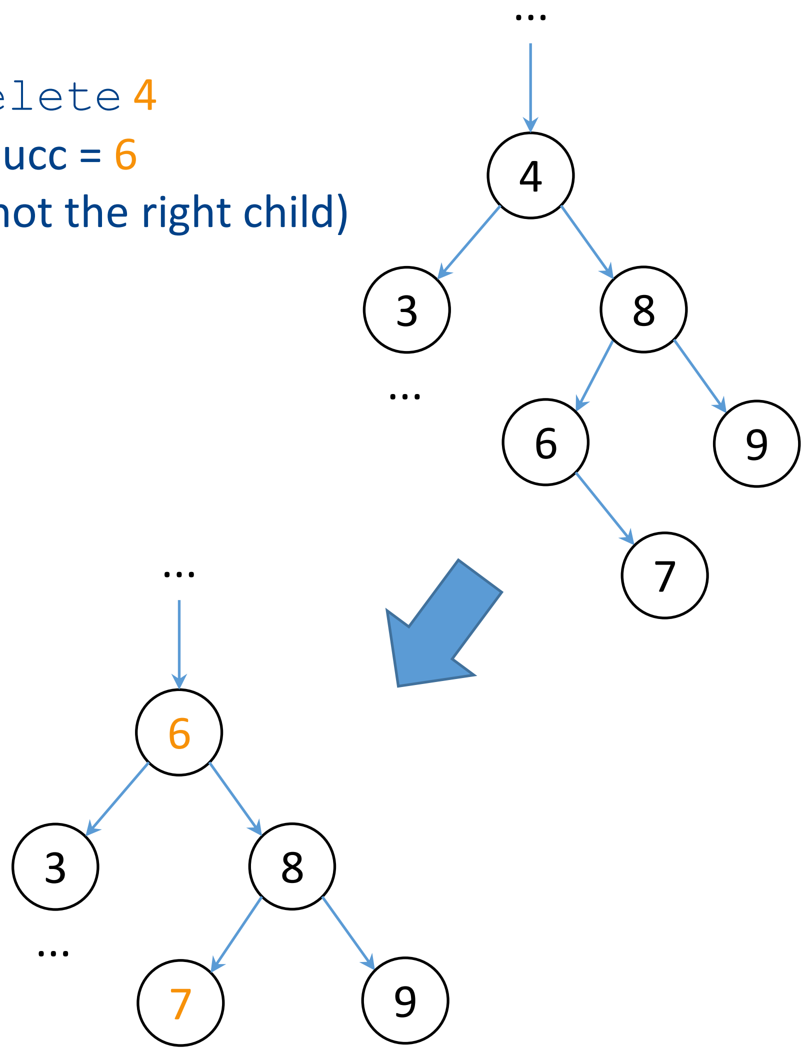
# Binary Search Tree: **Deletion**

- **Case #3**: delete node with two children
  - x – being removed, y – the successor

- **Subcase 3b**: successor is not the right child of x, **y ≠ x.right**
  - Regardless, y has no left child (being a succ of x)
  - y may or may not have the right child

  - **Solution**:
    - We replace y with its own right child
    - Then we replace x with y

```
Delete 4
```
succ = 6
(not the right child)



# Binary Search Tree: **Deletion**

- **Case #3**: delete node with two children
  - x – being removed, y – the successor

- **Subcase 3b**: successor is not the right child of x, **y ≠ x.right**
  - Regardless, y has no left child (being a succ of x)
  - y may or may not have the right child
  - **Solution**:
    - We replace y with its own right child
    - y has only one child → bypass it → **del_case_2(y)**

    - Then we replace x with y

```
del_case_3b(x,y)
   # first bypass y
  del_case_2(y)

   par = x.parent
   y.parent = par
   if par.left == x
      par.left = y
   else
      par.right = y

   y.left = x.left
   y.right = x.right
```

# Binary Search Tree: **Deletion**

- Putting all cases together (modular algorithm design)

```
num_kids(x)
  if x.left != null and x.right != null
    return 2
  elif x.left != null or x.right != null
    return 1
  else
    return 0
```

```
delete(x):
  n = num_kids(x)
  if n == 0
    del_case_1(x)
  elif n == 1
    del_case_2(x)
  else
    y = successor(x)
    if y == x.right
      del_case_3a(x)
    else
      del_case_3b(x, y)
```

# Querying a Binary Search Tree

- Let's revisit our operations

| | Runtime | | | | |
|---|---|---|---|---|---|
| `Data struct.` | `Search` | `Insert` | `Delete` | `Min/Max` | `Pred/Succ*` |
| **Array** | **O(n)** | **O(1)** | **O(n)** | **O(n)** | **O(n)** |
| **Linked List** | **O(n)** | **O(1)** | **O(1)** | **O(n)** | **O(n)** |
| **Hash Table** | **O(1)** | **O(1)** | **O(1)** | not possible | not possible |
| **Sorted Array** | **O(log n)** | **O(n)** | **O(n)** | **O(1)** | **O(1)** |
| **Binary Search Tree** | **O(h) = O(log n)** | **O(h)** | **O(h)*** | **O(h)** | **O(h)** |

***`Delete`** here assumes that we start from the T.root and first have to find the element x in order to delete it (it may not be in the tree at all). **Deletion itself**, if/when x is found, has time complexity **O(1)**

# Dynamic set operations – discussion

- **Q:** The most appropriate ADS for handling dynamic sets?
- Depends for which algorithm and which operations on dynamic sets need to be supported

- **Associate array**:
  - **Best**: if we need only to store values and efficiently retrieve them
  - **Not appropriate**: if we need to capture relations between elements:
    - Compute aggregates (e.g., average, max, min)
    - Find elements „close to” other elements (e.g., successor)
    - Unless we resort to **locality-sensitive hashing** (**LSH**)

# Dynamic set operations – discussion

- **Q:** The most appropriate ADS for handling dynamic sets?
- Depends for which algorithm and which operations on dynamic sets need to be supported

- **Binary search tree**:
  - **Best**: if we need to **actively maintain** the dynamic set and **search in it**
    - We need fast search – faster than with lists/arrays
    - But also support for operations that require capturing relations between elements
      - Which associative arrays cannot capture

# Trees vs. HashMaps: Example

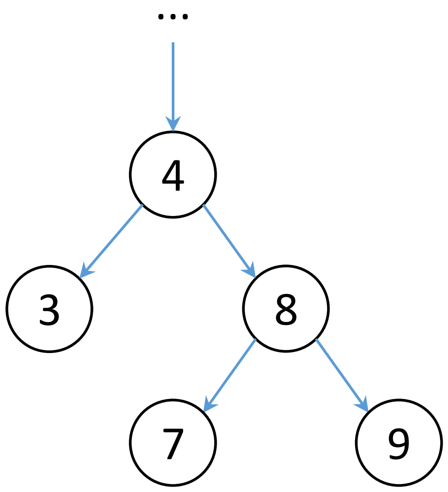
- In Information Retrieval (IR)
  - Work with large text collections
  - Need to store **all words** that appear in any of the documents in the collection
  - **Retrieval**: find documents in which words from the query appear
  - Large document collections: e.g., >10.000 differents words

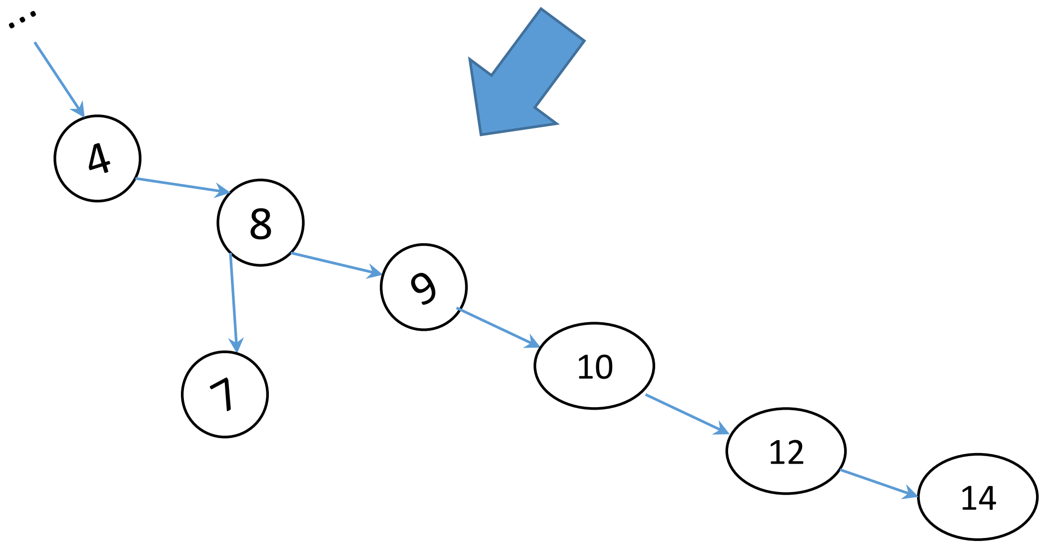
- Adequate data structure?
  - Associative Array (dictionary, hash map): if we expect the words to appear in the same **„form"** in the query as in the documents
  - What if the query has a misspelled word, e.g., *„algoirthm"*?
    - Would like *„algoirthm"* to be stored somewhere close to *„algorithm"* → **tree**

# Binary Search Tree: **Height/Depth**

- The complexity of all operations on the BST is O(h)
- If the BST is **balanced** $h \approx \log_2 n$
- Frequent insertions and deletions can disturb the balance of the tree

```
Insert(10)
Insert(12)
Insert(14)
Delete(3)
```




- **The height/depth drastically increases**
  - Extreme: BST reduced to a **linked list**
  - Search efficiency gains **lost**
  - Need to **re-balance** the tree. **Q:** How?

# Binary Search Tree: **Height/Depth**

- How do we **balance out** an unbalanced binary search tree?

  1. Construct the sorted array from the BST – **O(n)**
  2. Build a new BST from the sorted array (recursively) – **O(n)**

  If n is large, re-balancing this way is expensive and cannot be done frequently

- How to **maintain** balanced BSTs?
  - AVL trees
  - Red-black trees

# Questions?


Awọn ibeere?
Turite klausimų?
ਸਵਾਲ?
Küsimusi?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟