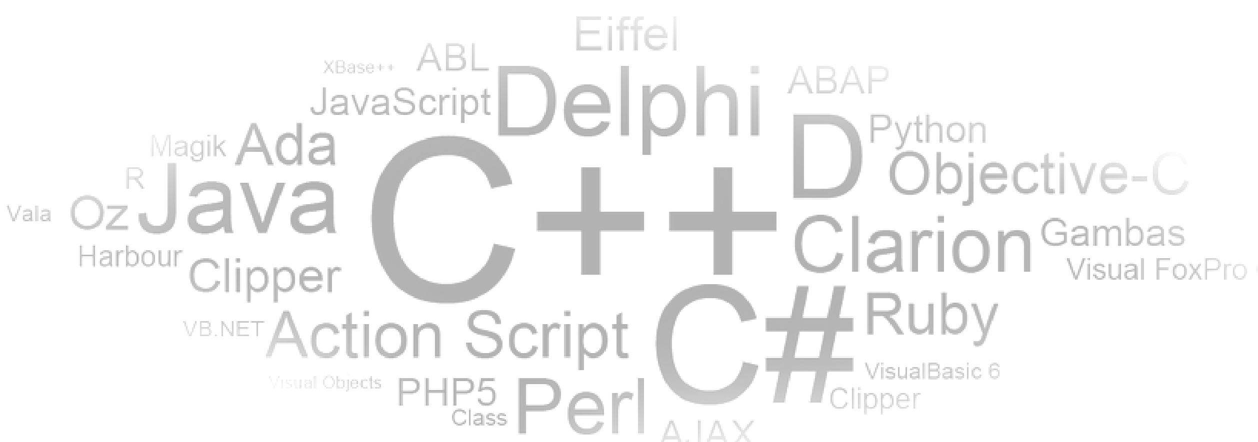
CIS 351-Data Structure-Searching-BST
April 16, 2020

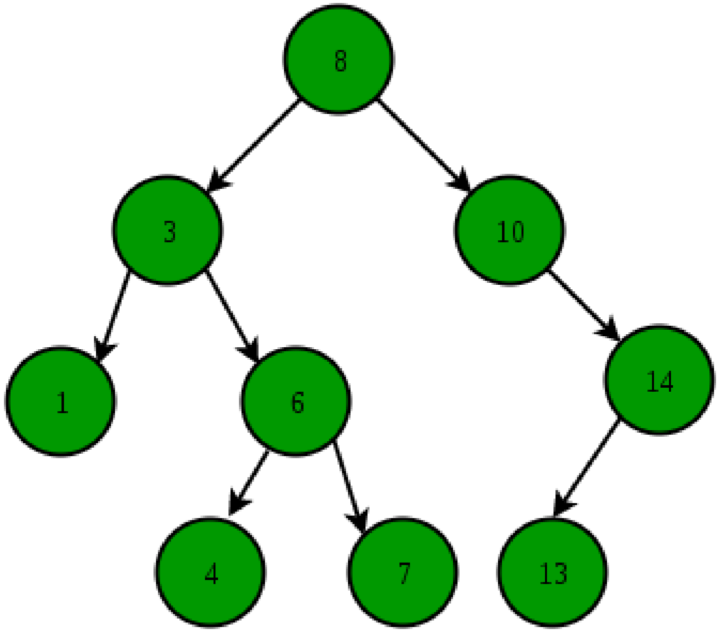
Dr. Farzana Rahman

Syracuse University



What is BST?

- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree Node

- A node in a binary tree is like a node in a linked list, with two node pointer fields:

```
Class Node
{
    int data;
    Node left;
    Node right;
}
```

Creating new Node

- Allocate memory for new node:

```
newNode= new Node();
```

- Initialize the contents of the node:

```
newNode.data = num;
```

- Set the pointers to NULL:

```
newNode.feft = NULL;
newNode.right = NULL;
```

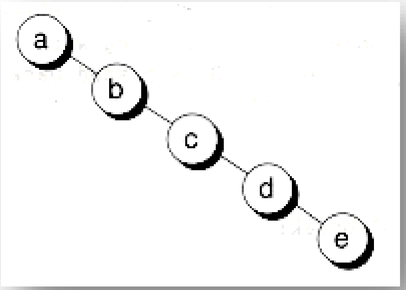
Search

- Search steps:
  - Start search at root node
  - If no match, and search item is smaller than root node, follow `lLink` to left subtree
  - Otherwise, follow `rLink` to right subtree

- Continue these steps until item is found or search ends at an empty subtree

Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - If the tree is degenerate
    - O(n) time
    - Like searching linked list / unsorted array



Degenerate tree

Binary Search Tree Construction

- How to build & maintain binary search trees?
  - Insertion
  - Deletion

- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree

Insert

- After inserting a new item, resulting binary tree must be a binary search tree

- Must find location where new item should be placed
  - Must keep two pointers, current and parent of current, in order to insert

BST Insertion

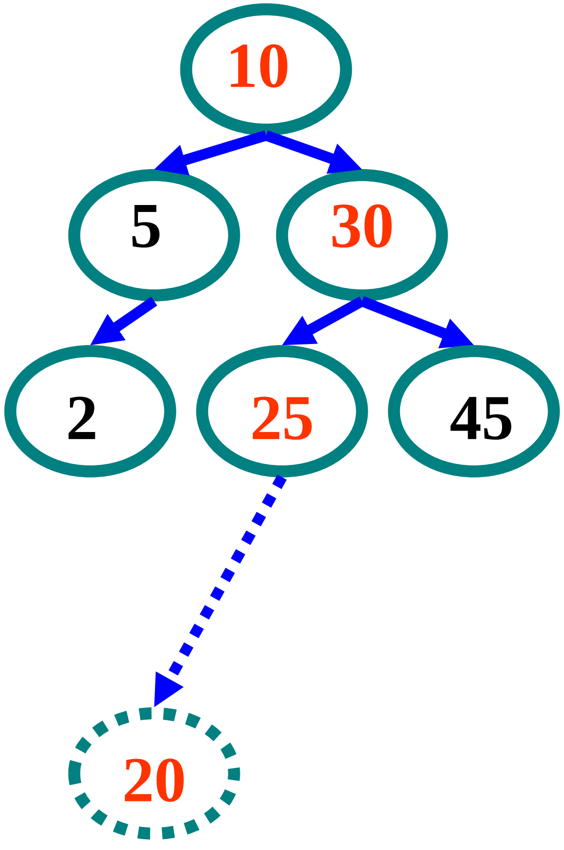
- To insert data all we need to do is follow the branches to an empty subtree and then insert the new node.
- In other words, all inserts take place at a leaf or at a leaflike node – a node that has only one null subtree.

Binary Search Tree – Insertion

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y

Example Insertion

- Insert (20)



20 > 10, right

20 < 30, left

20 < 25, left

Insert 20 on left

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1 if (empty tree)
   1 set root to newNode
   2 return newNode
2 end if
```

```
   Locate null subtree for insertion
3 if (newNode < root)
   1 return addBST (left subtree, newNode)
4 else
   1 return addBST (right subtree, newNode)
5 end if
end addBST
```

Delete (cont'd.)

- The delete operation has four cases:
    1. The node to be deleted is a leaf
    2. The node to be deleted has no left subtree
    3. The node to be deleted has no right subtree
    4. The node to be deleted has nonempty left and right subtrees
- Must find the node containing the item (if any) to be deleted, then delete the node

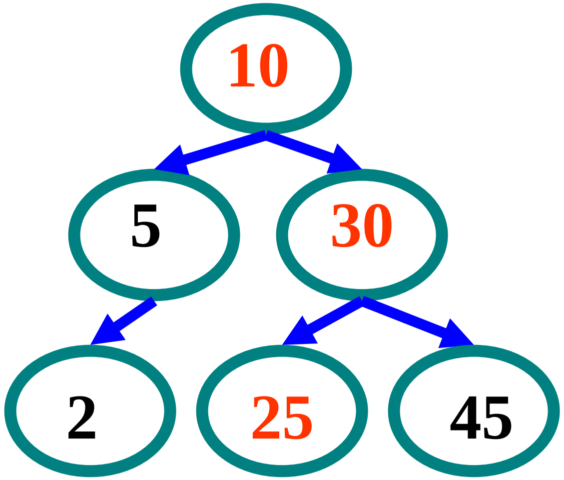
Deletion cases: Leaf Node

- To delete a leaf node, simply change the appropriate child field in the node's parent to point to *null*, instead of to the node.

- The node still exists, but is no longer a part of the tree.

Example Deletion (Leaf)

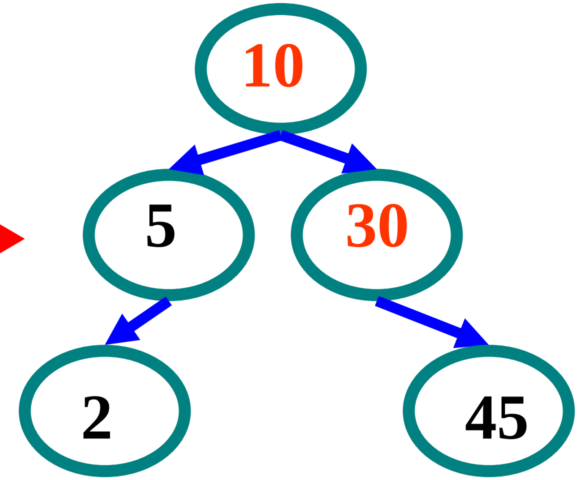
- Delete (25)



10 < 25, right

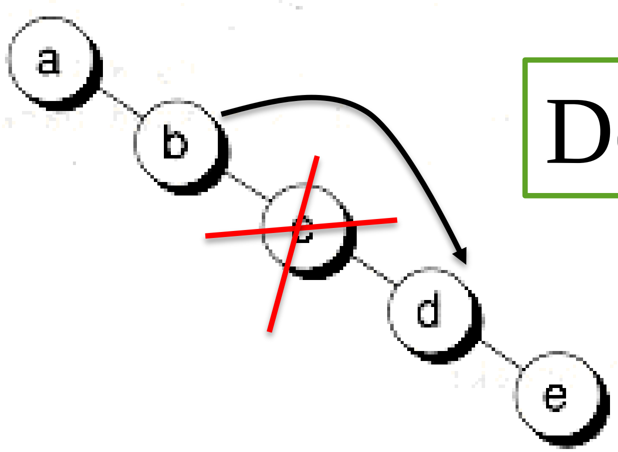
30 > 25, left

25 = 25, delete



Deletion: One Child

- The node to be deleted in this case has only two connections: to its parent and to its only child.
- Connect the child of the node to the node's parent, thus cutting off the connection between the node and its child, and between the node and its parent.



Delete C

Deletion

- The node to be deleted has two subtrees. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.

Deletion from the middle of a tree

- Rather than simply delete the node, we try to maintain the existing structure as much as possible by finding data to take the place of the deleted data. This can be done in one of two ways.

Deletion from the middle of a tree

- We can find the largest node in the deleted node’s left subtree and move its data to replace the deleted node’s data.
- We can find the smallest node on the deleted node’s right subtree and move its data to replace the deleted node’s data.
- Either of these moves preserves the integrity of the binary search tree.

Binary Search Tree – Deletion

- **Algorithm**
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

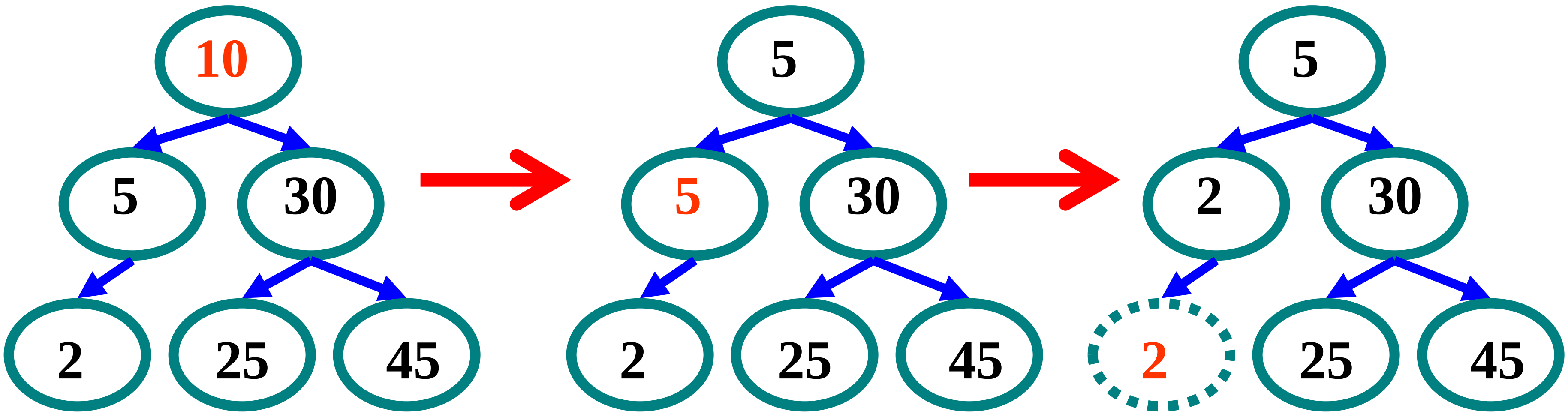
     a) Replace with largest value Y on left subtree

     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree

Example Deletion (Internal Node)

- Delete (10)



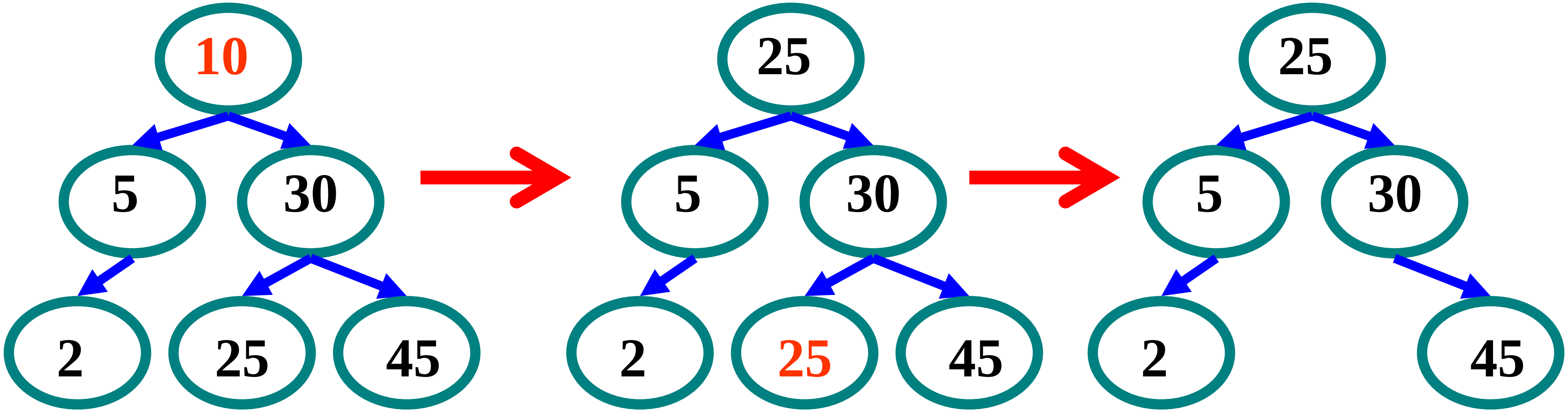
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Sequential search

- **sequential search**: Locates a target value in a list (may not be sorted) by examining each element from start to finish. Also known as *linear* search.
  - How many elements will it need to examine?
  - I

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	2	7	10	30	56	20	68	36	-4	25	42	50	22	92	15	85	103

i

Sequential (linear) search

- **sequential search**: Even if the list is sorted, elements are examined in the way (one after the other).
  - Example: Searching the list below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

i

Sequential (linear) search

- Sequential search code:

```
def sequential_search(my_list, value):
    for i in range(0, len(my_list)):
        if (my_list[i] == value):
            return i
    return -1    # not found
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- Note that -1 is returned if the element is not found.

Sequential (linear) search

- For a list of size N, how many elements will be checked worst case?

- On average how many elements will be checked?

- A list of 1,000,000 elements may require 1,000,000 elements to be examined.

- The number of elements to check grows in proportion to the size of the list, i.e., it grows linearly.

Binary Search

- **Binary search**: a method of searching that takes advantage of sorted data.

- Consider a guessing game:
- Someone thinks of a number between 1 and 100. You must guess the number.
- On each round, you are told whether your number is low, high, or correct.

- Best strategy: use a first guess of 50

  Eliminates half of the numbers immediately

  On each round, half the numbers are eliminated:

    100

    50

    25

    …

Binary search

- **binary search**: Locates a target value in a *sorted* list by successively eliminating half of the list from consideration.
  - How many elements will it need to examine?
  - Example: Searching the list below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

Keep track of indices for a min, mid and max.

- **Search for 42**: Round 1.

list[mid] < 42

eliminate from min to mid (left half)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min

mid

max

- **Search for 42**: Round 2.

list[mid] > 42

eliminate from mid to max (right half of what's left)

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min mid max

- **Search for 42**: Round 3.

list[mid] == 42

found!

index	0	1	2	3	4	5	6	7	8	9	1 0	1 1	1 2	1 3	1 4	1 5	16
value	-4	2	7	1 0	1 5	2 0	2 2	2 5	3 0	3 6	4 2	5 0	5 6	6 8	8 5	9 2	10 3

min mid ma
x

Binary search code

```
# Returns the index of an occurrence of target in a,
# or a negative number if the target is not found.
# Precondition: elements of a are in sorted order
def binary_search(a, target):
    min = 0
    max = len(a) - 1

    while (min <= max):
        mid = (min + max) // 2
        if (a[mid] < target):
            min = mid + 1
        elif (a[mid] > target):
            max = mid - 1
        else:
            return mid   # target found

    return -(min + 1)    # target not found
```

Binary search

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	12	18	25	27	30	36	42	56	68	85	91	92	98	102

What do the following calls return when passed the above list?

```
binary_search(a, 2)
binary_search(a, 68)
binary_search(a, 12)
```