Lecture 10: Searching algorithm (Part. 1)

Algorithm

Jeong-Hun Kim

Table of Contents

Part 1

PRELIMINARIES

Preliminaries

- Record
  - A unit space that contains **all information** about an object
    - e.g., Human record
      - Resident ID, name, address, phone number, etc.

- Field
  - An element in a record that represents **each piece of information**
    - e.g., Resident ID in Human record

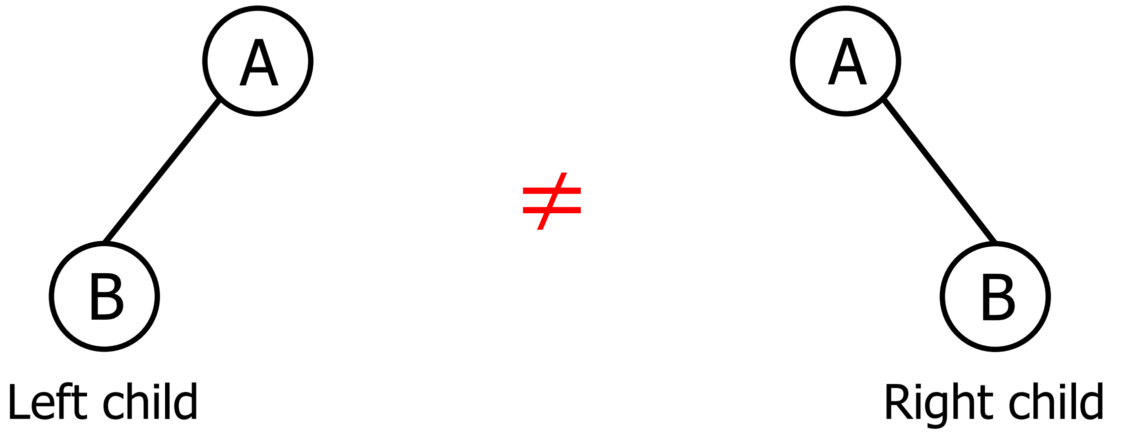
Preliminaries

- (Search) key
  - A field that **uniquely** represents each record to avoid duplication
  - The key may consist of one or more fields
    - e.g., Account ID

- Search tree
  - A tree structure composed of nodes with keys following **specific rules**
  - Allows identification of where a specific record is stored in the tree

Preliminaries

- Binary tree
  - A tree structure where each node has a maximum of **two child nodes**
  - If the positions of the children differ, it becomes a different tree

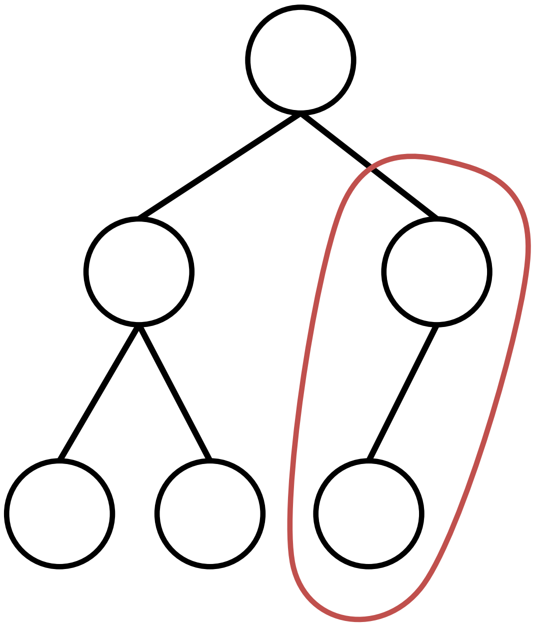


Preliminaries

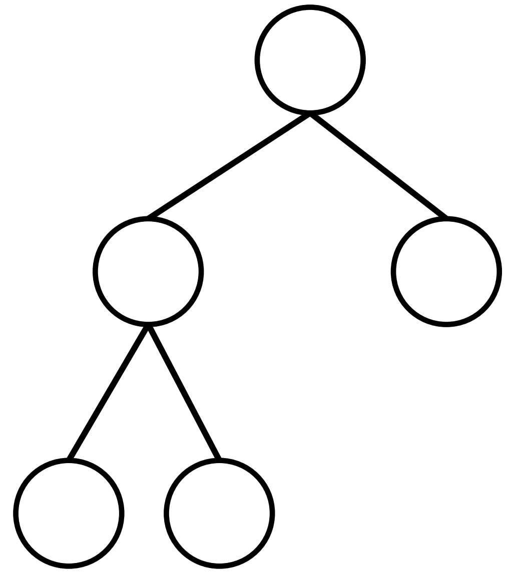
❖ Types of binary tree

1) Full binary tree

- A binary tree where every node has either **0 or 2** child nodes
  - Note that lead nodes are excluded



Binary tree



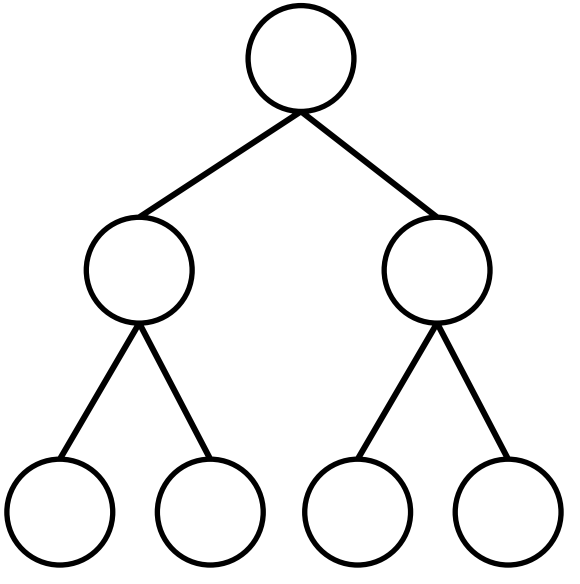
Full binary tree

Preliminaries

❖ Types of binary tree

2) Perfect binary tree

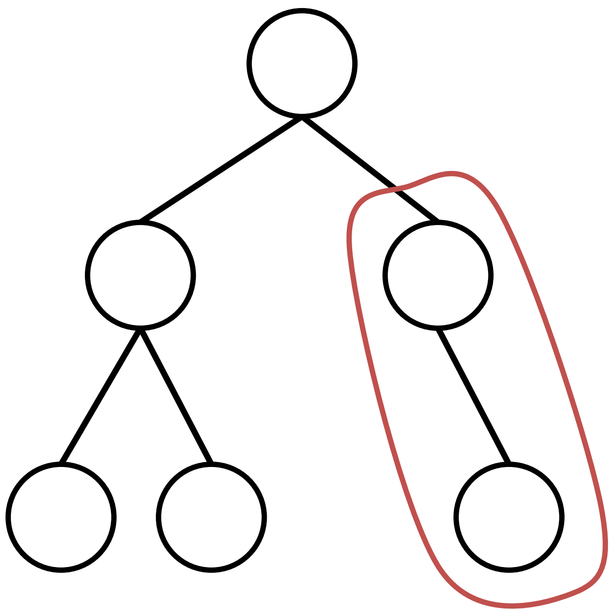
- A binary tree where every node has **2** child nodes
- All leaf nodes are at the **same level**
- Characteristics:
  - If the height is h, the total number of nodes is $2^{(h+1)} - 1$
  - The number of leaf nodes is 2^h



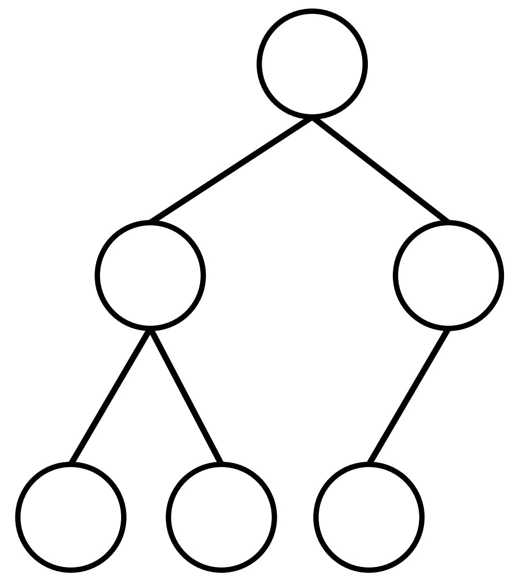
Perfect binary tree

Preliminaries

- Types of binary tree
  3) Complete binary tree
     - All nodes **must be filled** except for the last level
     - All nodes are filled **from left to right**



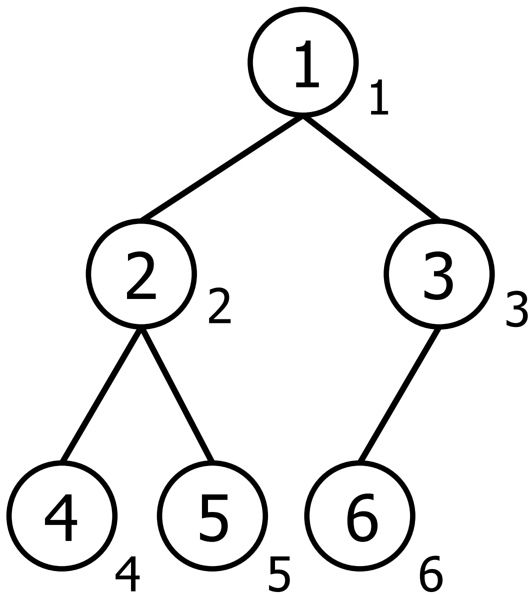
Binary tree



Complete binary tree

Preliminaries

- Characteristics of complete binary tree
  - Can be represented as an 1-d array



Index	0	1	2	3	4	5	6
	-	**1**	**2**	**3**	**4**	**5**	**6**

- Left child of the i-th node: (i * 2)-th node
- Right child of the i-th node: (i * 2 + 1)-th node

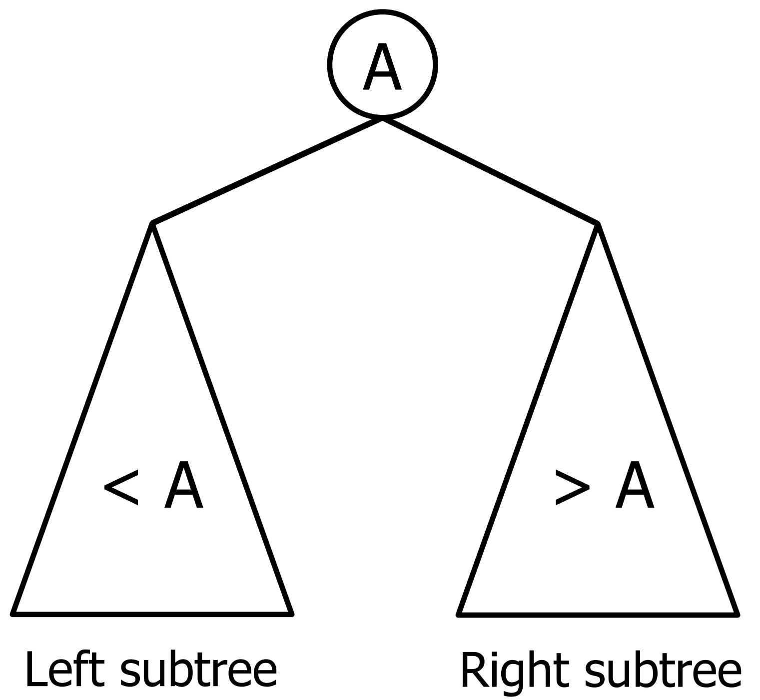
Part 2

BINARY SEARCH TREE

Binary Search Tree

❖ Definition

- A binary tree where each node has a unique key
- The key of any node is **greater than** the key of its **left child** node and **smaller than** the key of its **right child** node



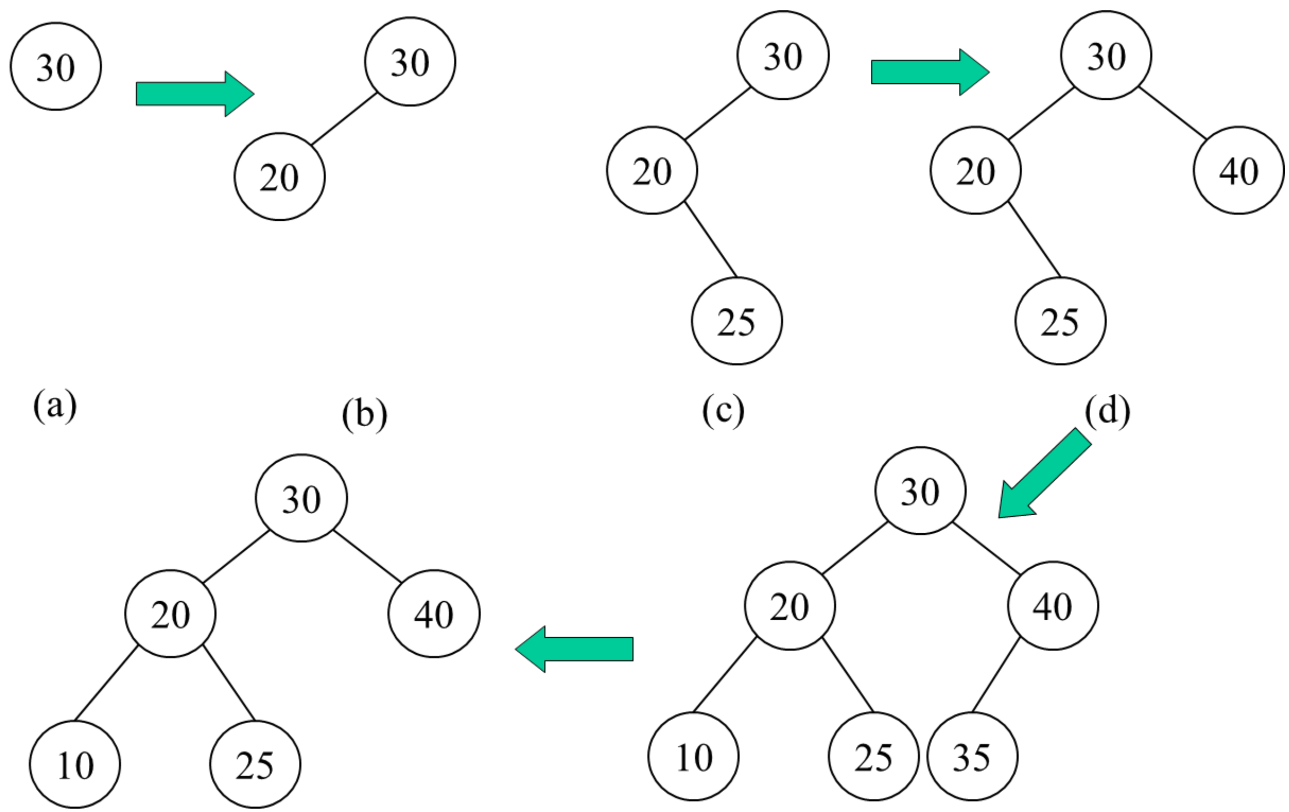
Binary Search Tree

❖ Node insertion in a binary search tree

```
int tree[MAX_SIZE];

void treeInsert(int x, int idx = 1){
   if (idx = NIL)
      tree[idx] = x;
   if (x < tree[idx])
      treeInsert(x, idx * 2);
   else
      treeInsert(x, idx * 2 + 1);
}

※ NIL = nothing
```

Binary Search Tree

- Example of node insertion



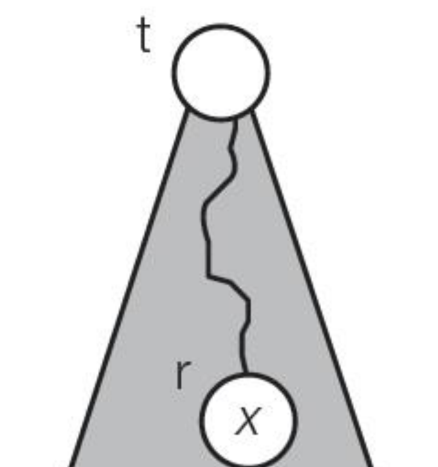
Binary Search Tree

❖ Searching algorithm

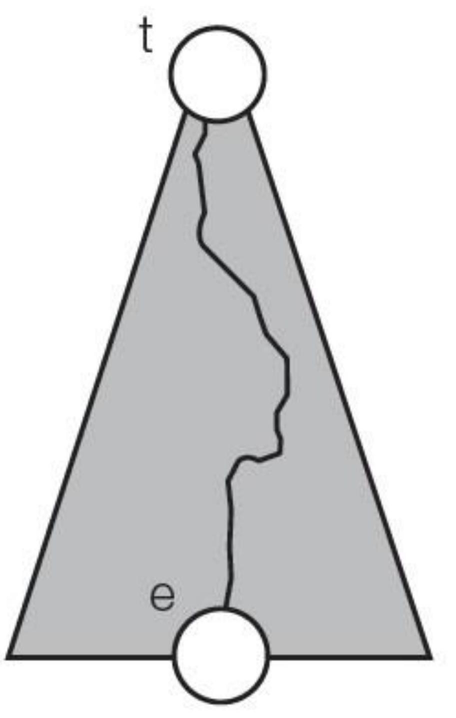
```
int tree[MAX_SIZE];

int treeSearch(int x, int idx = 1){
   if (idx = NIL or tree[idx] = x)
      return idx;
   if (x < tree[idx])
      return treeSearch(x, idx * 2);
   else
      return treeSearch(x, idx * 2 + 1);
}

※ NIL = nothing
```

Binary Search Tree

- Algorithm analysis
  - Successful and unsuccessful searching



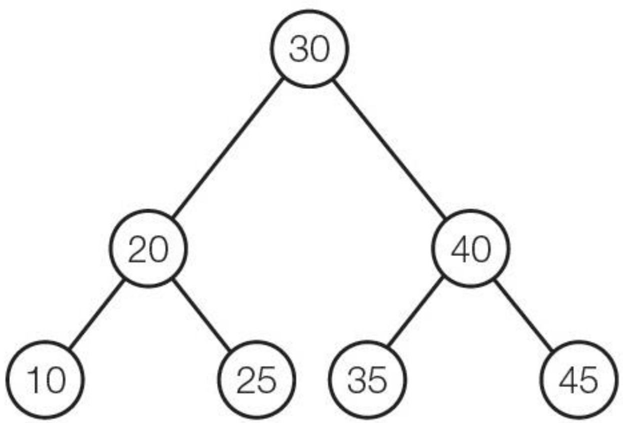
Successful searching



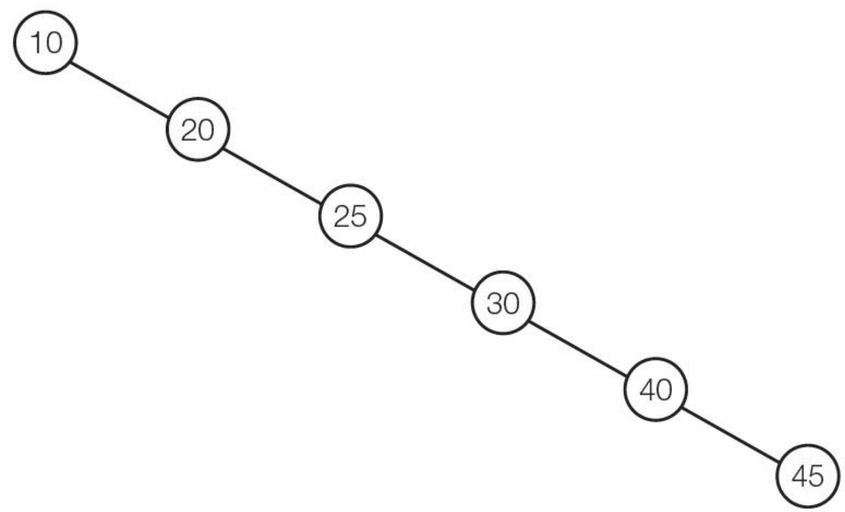
Unsuccessful searching

Binary Search Tree

- Time complexity of insertion and searching in a binary search tree
  - If the tree is balanced (best case): $O(\log n)$
  - If the tree is unbalanced (worst case): $O(n)$
  - Averagely $O(\log n)$



Balanced tree



Unbalanced tree

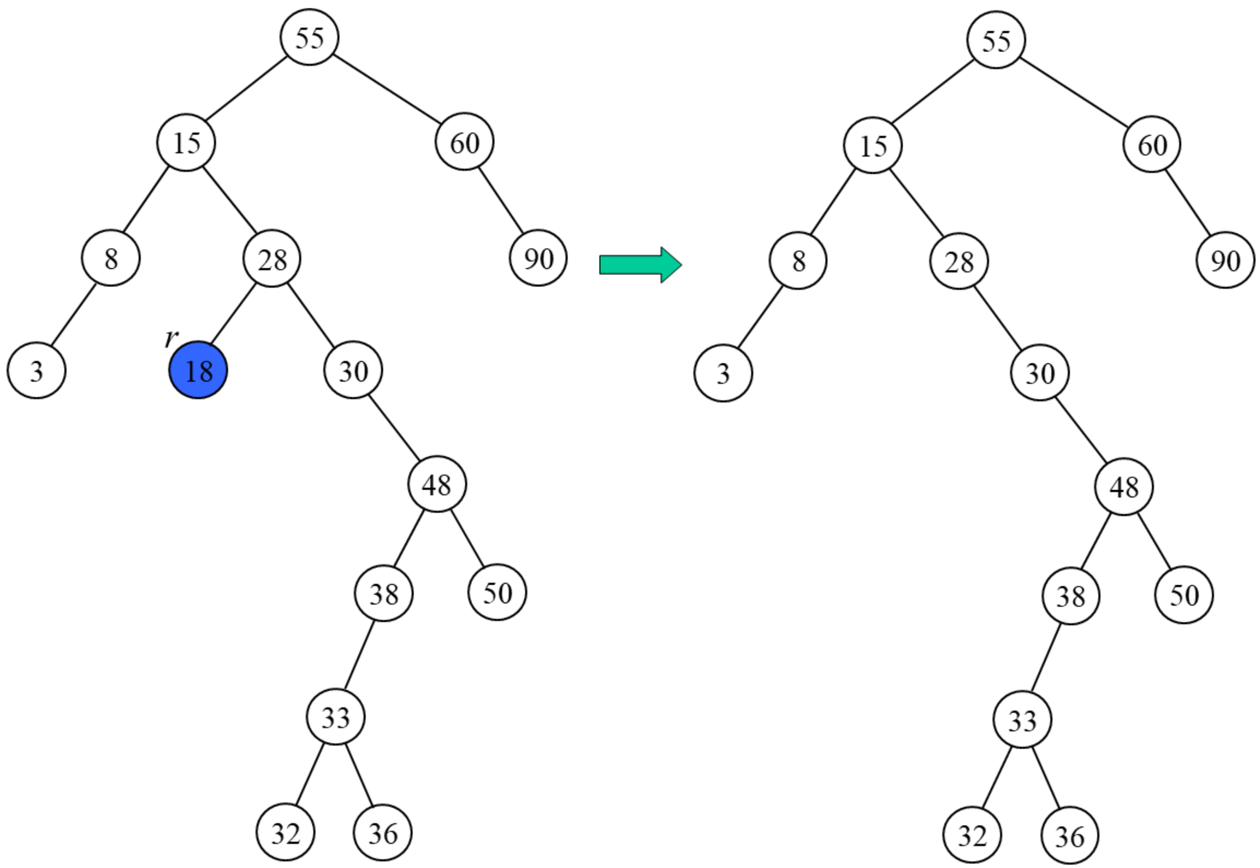
Binary Search Tree

- Node deletion in a binary search tree
  - Handled differently depending on following three cases:
    - Case 1: if the target has 0 child node (= leaf node)
    - Case 2: if the target has 1 child node
    - Case 3: if the target has 2 child nodes

```
int tree[MAX_SIZE];

void treeDelete(int x, int idx = 1){
   int target = treeSearch(x, idx)
   if (tree[target * 2] = NIL and tree[target * 2 + 1] = NIL)
      remove the target node
   else if (tree[target * 2] != NIL and tree[target * 2 + 1] != NIL)
      swap the target node with the minimum node of its right subtree,
      then delete
   else
      directly connect the target node's parent to its child
}
```

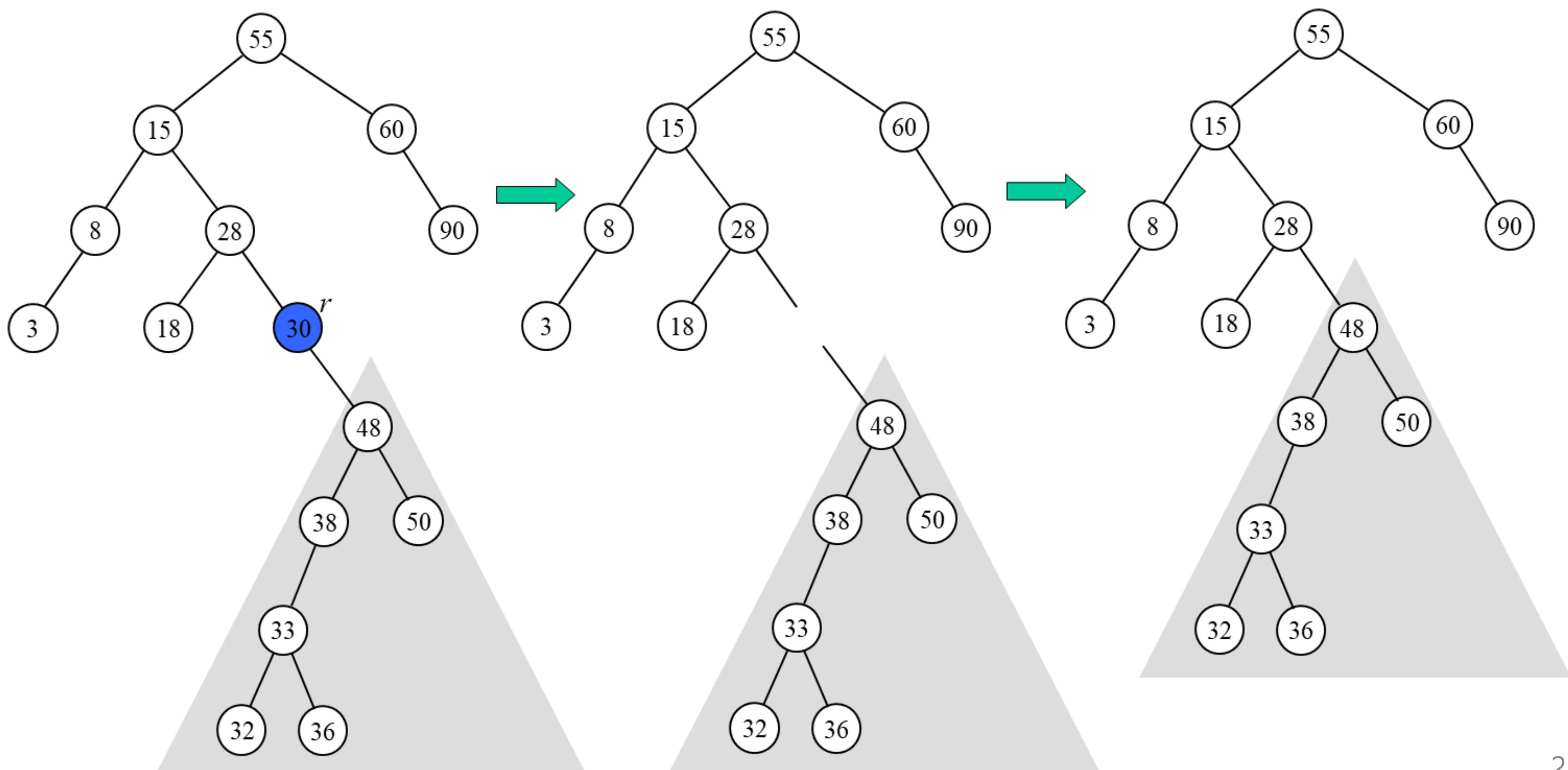
Binary Search Tree

- Node deletion in a binary search tree
  - Handled differently depending on following three cases:
    - Case 1: if the target has 0 child node (= leaf node)



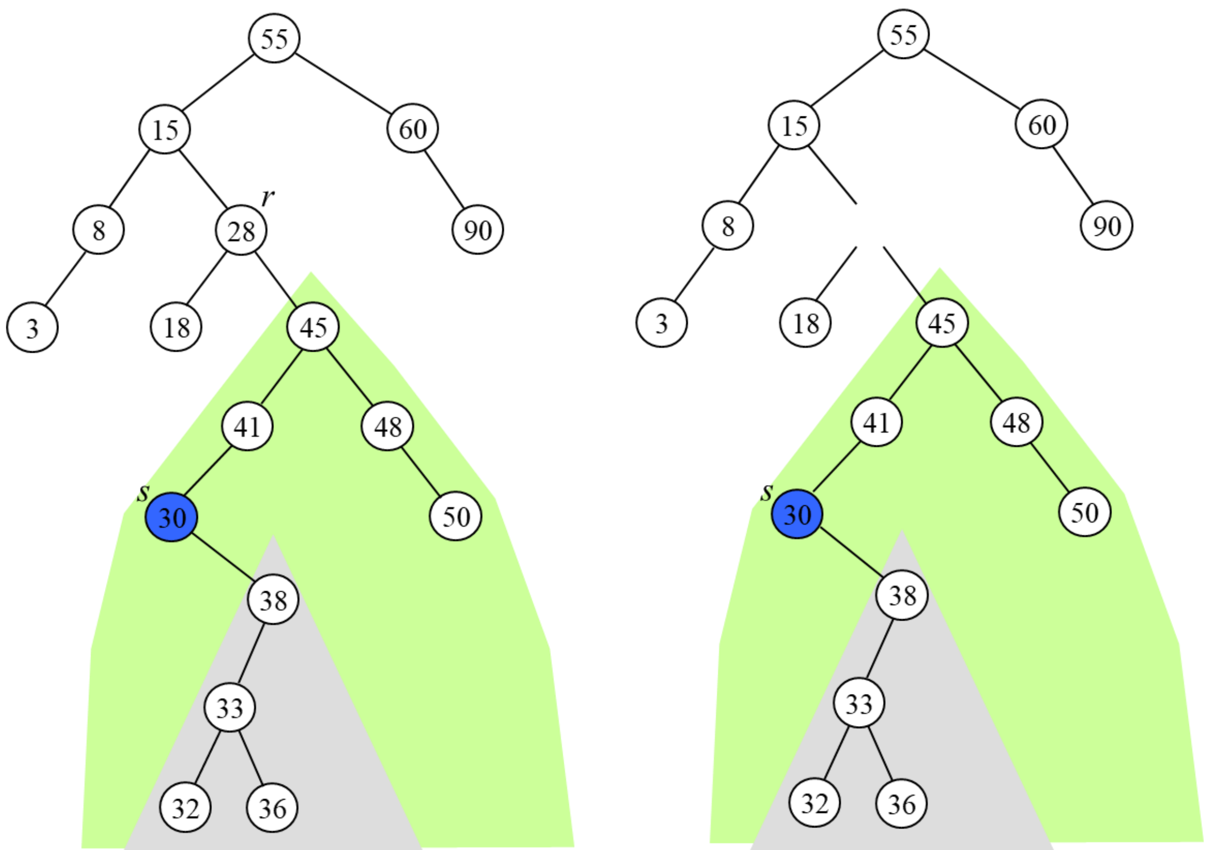
Binary Search Tree

- Node deletion in a binary search tree
  - Handled differently depending on following three cases:
    - Case 2: if the target has 1 child node



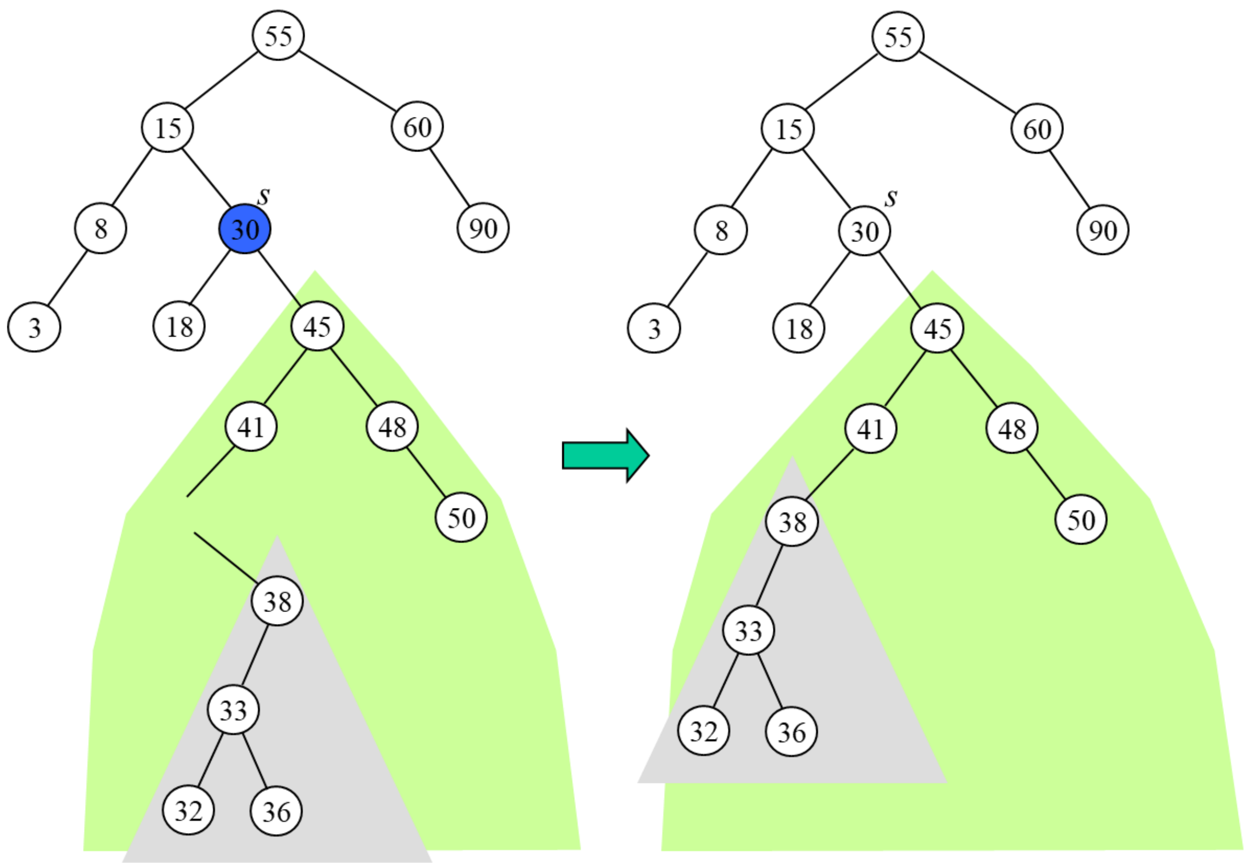
Binary Search Tree

- Node deletion in a binary search tree
  - Handled differently depending on following three cases:
    - Case 3: if the target has 2 child nodes



Binary Search Tree

- Node deletion in a binary search tree
  - Handled differently depending on following three cases:
    - Case 3: if the target has 2 child nodes (cont'd)



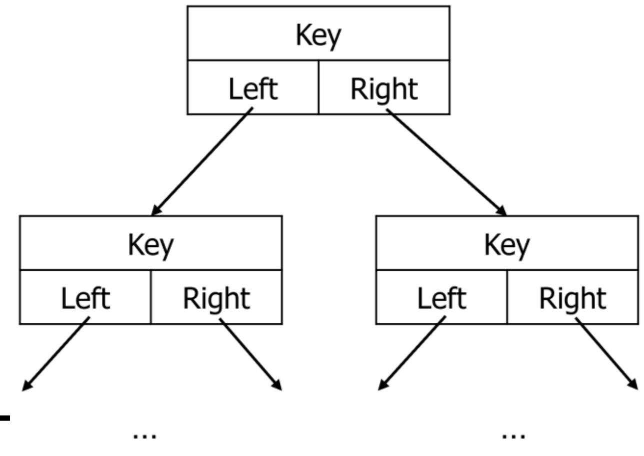
Binary Search Tree

- Node deletion in a binary search tree
  - Issues with 1-d array
    - As the height of the binary tree increases, the space complexity of the array grows exponentially
    - Due to node deletion, elements in the array must be shifted

-	1	2	3	4	5	6	NIL	8	9	10	11	12	13

-	1	2	6	4	5	12	13	8	9	10	11	NIL	NIL

Binary Search Tree



- Node deletion in a binary search tree
  - Linked list-based binary search tree
    - Can mitigate issues with 1-d arrays
    - However, it requires cost for linking

```c
typedef struct Node {
   int key;
   struct Node* left;
   struct Node* right;
} Node;

Node* createNode(int x){
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode -> key = x;
   newNode -> left = NULL;
   newNode -> right = NULL;
   return newNode;
}
```

Binary Search Tree

- Node deletion in a binary search tree
  - Linked list-based binary search tree (cont'd)

```
Node* treeInsert(Node* root, int x){
   if (root = NULL){
      return createNode(x);
   }
   if (x < root -> key)
      root -> left = treeInsert(root -> left, x);
   else if (x > root -> key)
      root -> right = treeInsert(root -> right, x);
   return root;
}
```

Binary Search Tree

❖ Node deletion in a binary search tree

- Linked list-based binary search tree (cont'd)

```
Node* treeSearch(Node* root, int x){
   if (root = NULL and root -> key = x){
      return root;
   }
   if (x < root -> key)
      treeSearch(root -> left, x);
   else
      treeSearch(root -> right, x);
}
```

Binary Search Tree

❖ Node deletion in a binary search tree

- Linked list-based binary search tree (cont'd)

```
Node* treeDelete(Node* root, int x){
  if (root = NULL){
    return root;
  }
  if (x < root -> key)
    root -> left = treeDelete(root -> left, x);
  else if (x > root -> key)
    root -> right = treeDelete(root -> right, x);
  else {
    if (root -> left = NULL and root -> right = NULL){
      free(root);
      return NULL;
    } else if (root -> left = NULL){
      Node* temp = root -> right;
      free(root);
      return temp;
    } else if (root -> right = NULL){
      Node* temp = root -> left;
      free(root);
      return temp;
    } else {
      Node* temp = findMinNode(root -> right);
      root -> key = temp -> key;
      root -> right = treeDelete(root -> right, temp -> key)
    }
  }
  return root;
}
```

```
Node* findMinNode(Node* root){
  while (root -> left != NULL)
    root = root -> left;
  return root;
}
```

Time complexity for node deletion
Balanced: O(log n)
Unbalanced: O(n)

Summary

- Preliminaries
  - Record
  - Field
  - Key
  - Search tree
  - Binary tree

- Binary search tree
  - Searching
  - Insertion
  - Deletion

Questions?

SEE YOU NEXT TIME!