CS 261-020
Data Structures

Lecture 8

Binary Search

Binary Trees

Midterm Review

2/8/24, Thursday

Oregon State University

Odds and Ends

- Assignment 2 due Sunday midnight
- Assignment 1 demo due Friday (2/9)
- Midterm:
  - Tuesday (2/13) during lecture time, same classroom
  - Review today

Lecture Topics:

- Binary Search
- Midterm Review

Binary Search

- Important to be able to search through a collection of element
  - i.e., find the index of an element
  - Determine that element does not exist

- How to do this using linear data structures we've seen?
  - By ***iterating through*** elements one by one until we find the element or the end of the collection (doesn't exist)
  - This is called linear search
  - Runtime Complexity: $O(n)$ where n is number of elements of the collection

- Can we improve this?

Binary Search

- Can you perform a search for 65 in the following array:

3	8	12	14	23	40	48	51	63	65	71	79	83	89	90	100

- What do you notice?
  - The array is sorted
  - Each iteration, eliminate half of the remaining array

- Binary Search: iterate through an ordered (sorted) array, and repeatedly divide the search interval in half
- Run Complexity: O(log n)

Binary Search vs. Linear Search

- Searching in an array of size n = 1,000,000
  - Linear Search: O(n) = 1,000,000 comparisons, on average
  - Binary Search: O(log n) ≈ 20 comparisons, on average

- Searching in an array of size n = 4,000,000,000
  - Linear Search: O(n) = 4,000,000,000 comparisons, on average
  - Binary Search: O(log n) ≈ 32 comparisons, on average

→ Binary Search is a lot faster, especially for large values of n

How does Binary Search work?

- At each iteration:
  - Compare the query value (the value it's searching for) to the value at the midpoint of the array
  - If they matches, break and return (i.e., index)
  - Otherwise …
    - If query value < array's midpoint value, repeat only on the "lower" half of the array
    - If query value > array's midpoint value, repeat only on the "upper" half of the array
  - If the array under consideration has size 0, break and return. The query value does not exist. (i.e., -1 or where it should be inserted to maintain a sorted array)

How does Binary Search work?

- Iteration:

3	8	12	14	23	40	48	51	63	65	71	79	83	89	90	100

```
int binary_search(int q, int* array, int n) {
        int mid, low = 0, high = n - 1;
        while (low <= high) {
                mid = (low + high) / 2;
                if (array[mid] == q)
                        return mid;

                else if (array[mid] < q)
                        low = mid + 1;

                else

                        high = mid - 1;
        }
        return low;
}
```

- low – the first index of the sub-array
- high – the last index of the sub-array
- mid – the index of the midpoint of the sub-array

How does Binary Search work?

- Recursion:

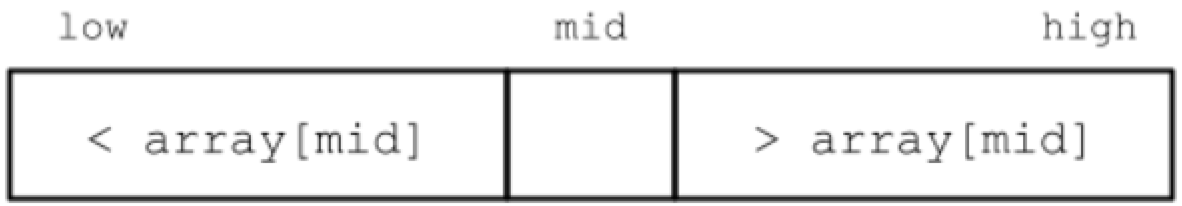
3	8	12	14	23	40	48	51	63	65	71	79	83	89	90	100

```
int binary_search(int q, int* array, int low, int high) {
        while (low <= high) {
                int mid = (low + high) / 2;
                if (array[mid] == q)
                        return mid;

                else if (array[mid] < q)
                        return binary_search(q, array, mid+1, high);

                else

                        return binary_search(q, array, low, mid-1);
        }
        return low;
}
```

- low – the first index of the sub-array
- high – the last index of the sub-array
- mid – the index of the midpoint of the sub-array

Ordered Array

- Note: Binary Search can only work within an ordered (sorted) array
  - The assumption that allows binary search to eliminate half of the array at each iteration



- Make sure the array is sorted before using binary search!
  - Using a sorting algorithm
  - Using binary search

Ordering Array using a sorting algorithm

- Using a sorting algorithm to order an array

- Runtime complexity of the best general-purpose sorting algorithm
  - O (n log n)
  - Best if we limit the number of times to "sort"

- Examples:
  - Look up in a phone book
  - Look up a word in a dictionary

- What if we expected new elements to be inserted frequently?

Ordering Array using Binary Search

- If data is frequently changing (i.e., insertion), run binary search after each insertion to maintain an ordered array
  - Recall: `binary_search()` may return the index where an element should be inserted

- Thus, the cost of each insertion:
  - O(log n) to identify the index to be inserted using binary search
  - O(n) to shift the subsequent elements back one spot
  - Since O(n) dominates O(log n), the cost of each insertion is O(n)

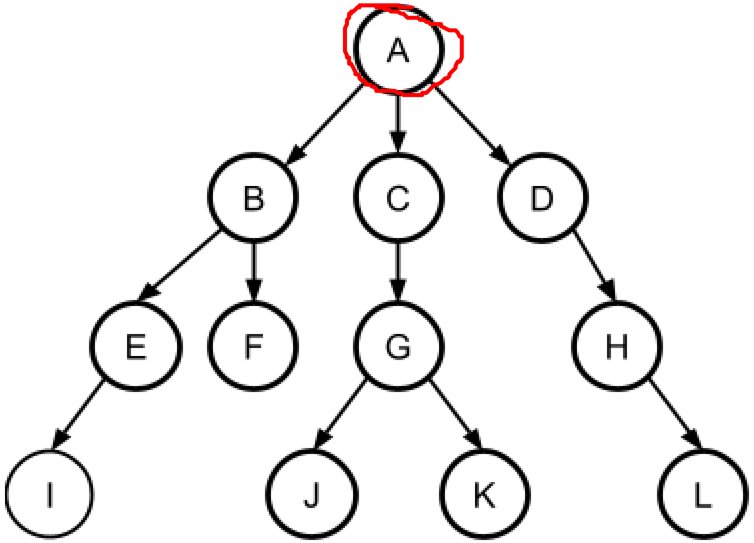
- The total cost of n insertion is $O(n*n) = O(n^2)$

Lecture Topics:

- Binary Trees

Trees

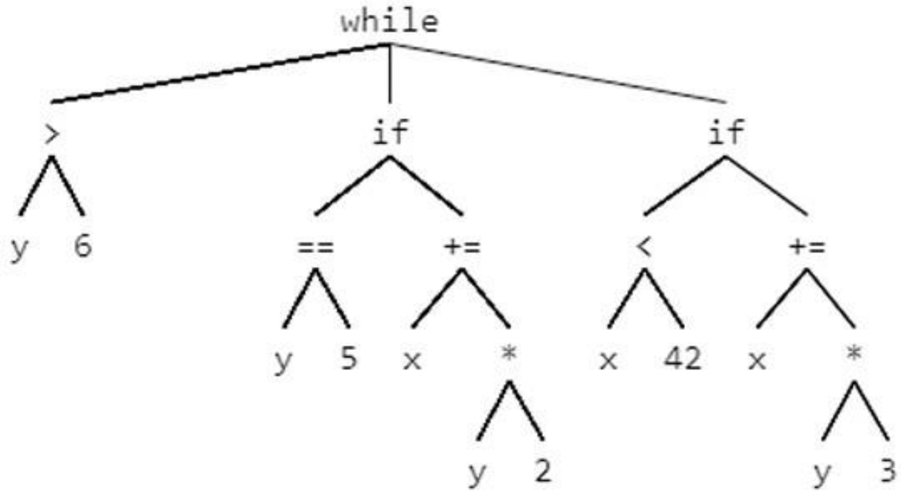
- Tree: non-linear data structure, represents data as a hierarchical structure, encoding the hierarchical relationships between different elements

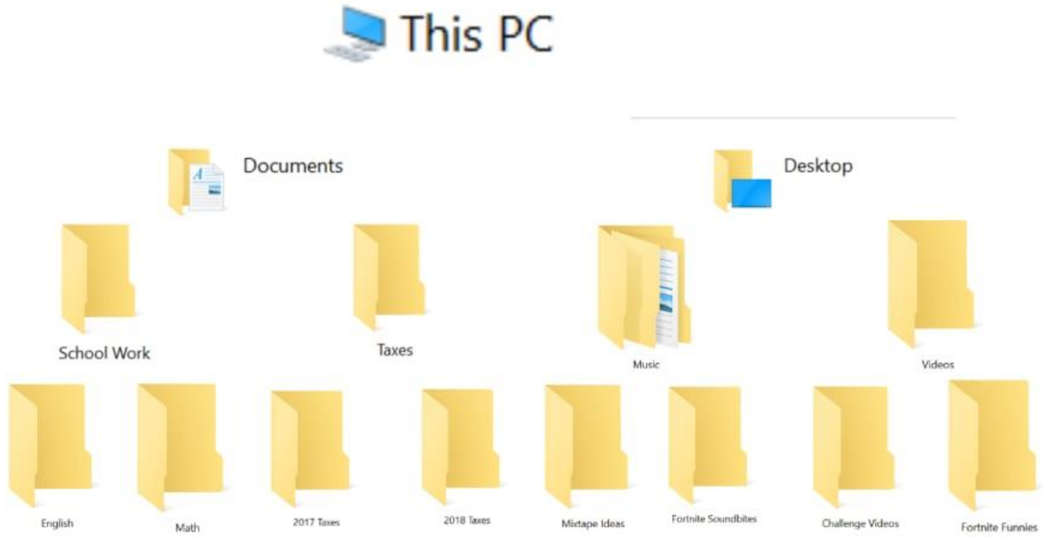


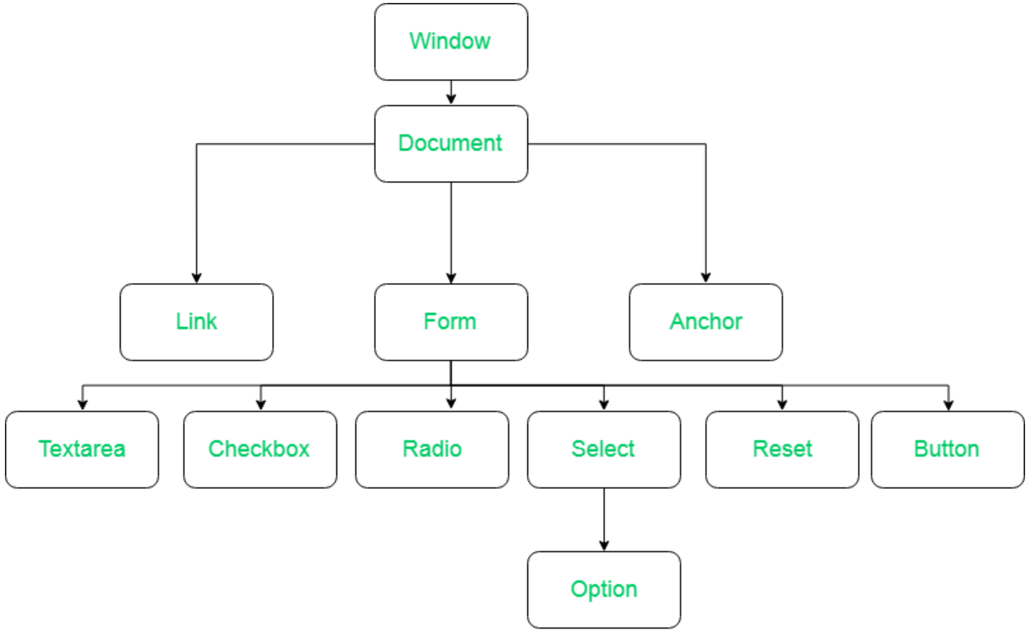
- Node: each individual data element in a tree
  - Contains the data element and points to other nodes
- Edge (arc): an encoded relationship between data elements
  - Represents directed relationships

Tree Examples

- Examples:
  - Computer’s filesystem
  - Object model of a web page
  - Compiler’s abstract syntax tree of a program

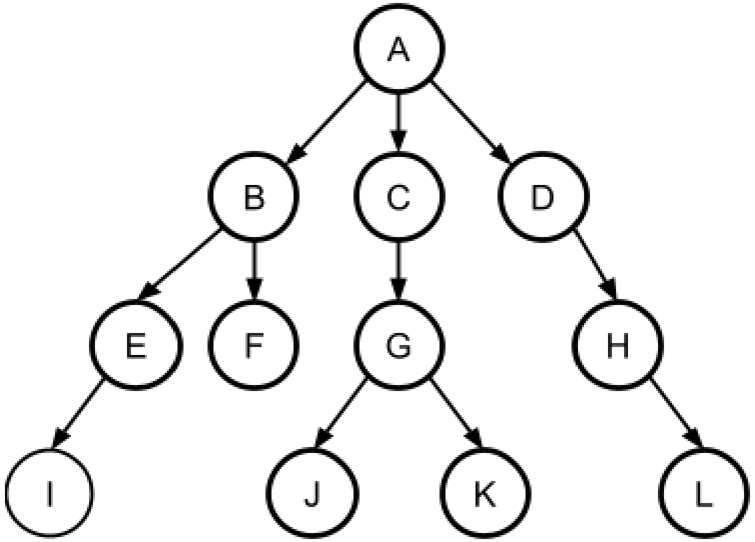






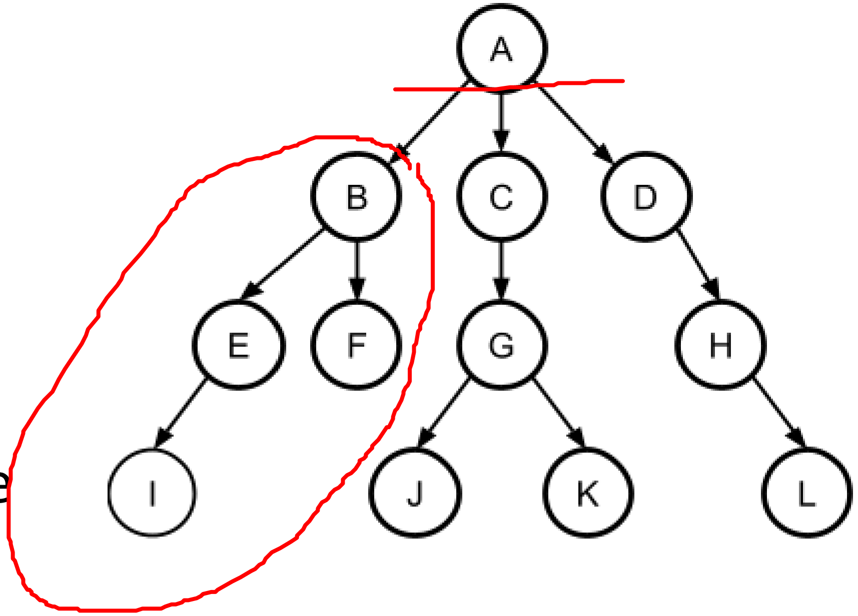
Trees

- ***Parent***: A node P in a tree is called the parent of another node C if P has an edge that points directly to C.
  - A is parent of B, C, D; B is parent of E and F
- ***Child:*** A node C in a tree is called the child of another node P if P is C's parent.
  - B, C, D are children of A; J, K are children of G
- ***Sibling:*** A node S_1 is the sibling of another node S_2 if S_1 and S_2 share the same parent node P
  - B, C, D are siblings; J, K are siblings
- ***Descendant:*** The descendants of a node N are all of N's children, plus its children's children, and so forth.
  - E, F, and I are descendants of node B, and nodes H and L are descendants of node D
- ***Ancestor***: A node A is the ancestor of another node D if D is a descendant of A
  - E, B, and A are ancestors of I, and G, C, and A are ancestors of node K



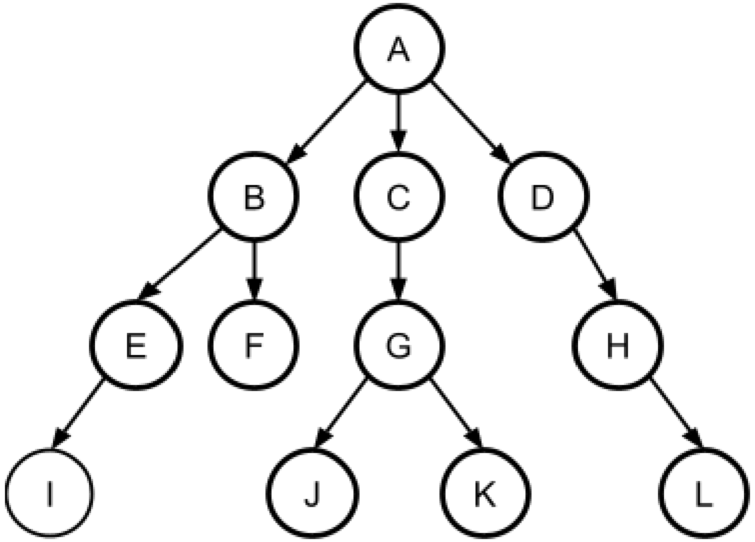
Trees

- ***Root***: Ancestor of all other nodes in the tree. Each tree has exactly one root.
  - node A is the root.
- ***Interior (node)***: A node has at least one child.
  - A, B, C, D, E, G, and H are interior nodes.
- ***Leaf (node):*** A node has no children.
  - F, I, J, K, and L are leaves.
- ***Subtree***: the portion of a tree that consists of a single node *N*, all of *N*'s descendants, and the edges joining these nodes.
  - the subtree rooted at node B contains the nodes B, E, F, and I and the edges joining those nodes.



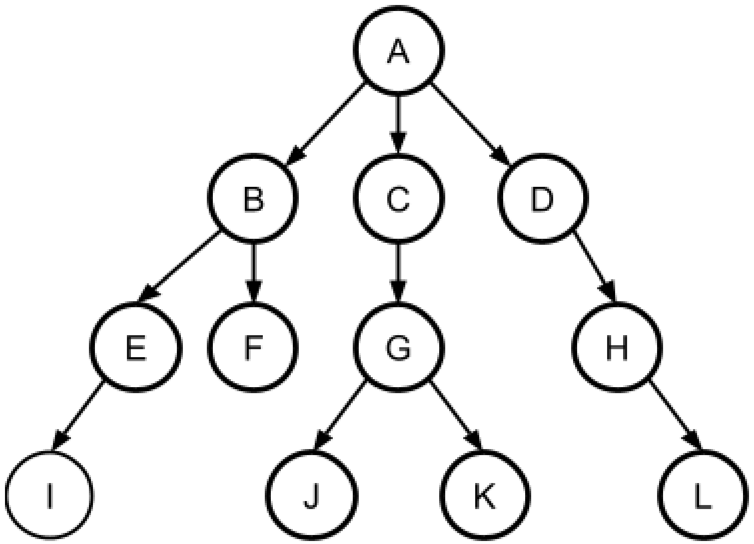
Trees

- ***Path***: the collection of edges in a tree joining a node to one of its descendants.
- ***Path length:*** the number of edges in that path.
  - the path from C to K has length 2, since it contains 2 edges.
- ***Depth:*** The depth of a node *N* in a tree is the length of the path from the root to *N*.
  - the depth of K is 3.
  - The depth of A (root) is 0.
- ***Height:*** The maximum depth of any node in the tree.
  - The tree has height 3



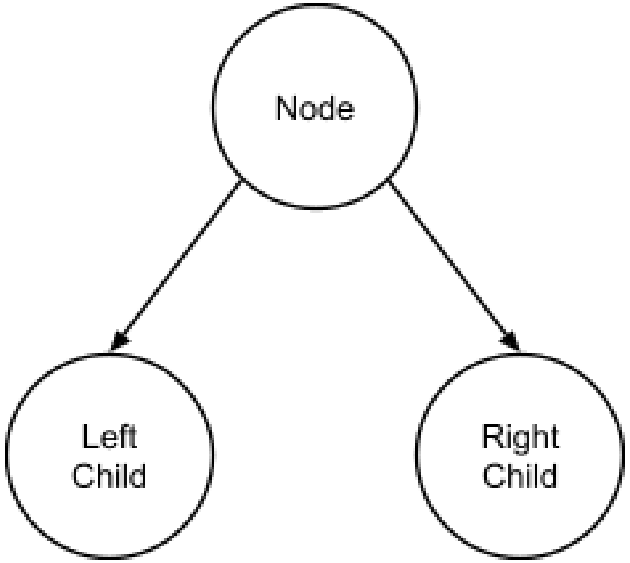
Trees

- Constraints to be counted as a tree:
  - Each node in the structure may have only one parent.
  - The edges of the structure many not form any cycles.
    - there cannot be a path from any node to itself.



Binary Trees

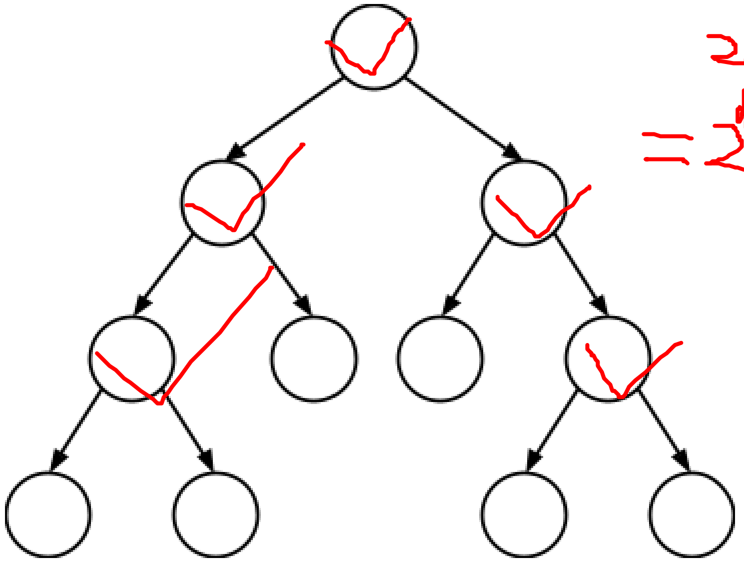
- *Binary Tree*: a tree in which each node can have at most two children (left child and right child).

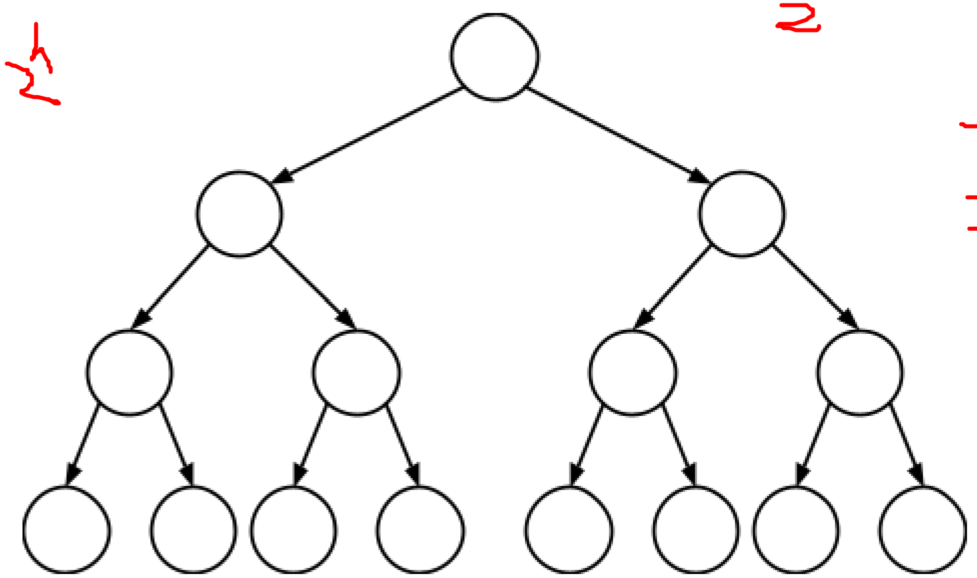


- *Left subtree*: the subtree rooted at that node's left child
- *Right subtree*: the subtree rooted at that node's right child

Binary Trees

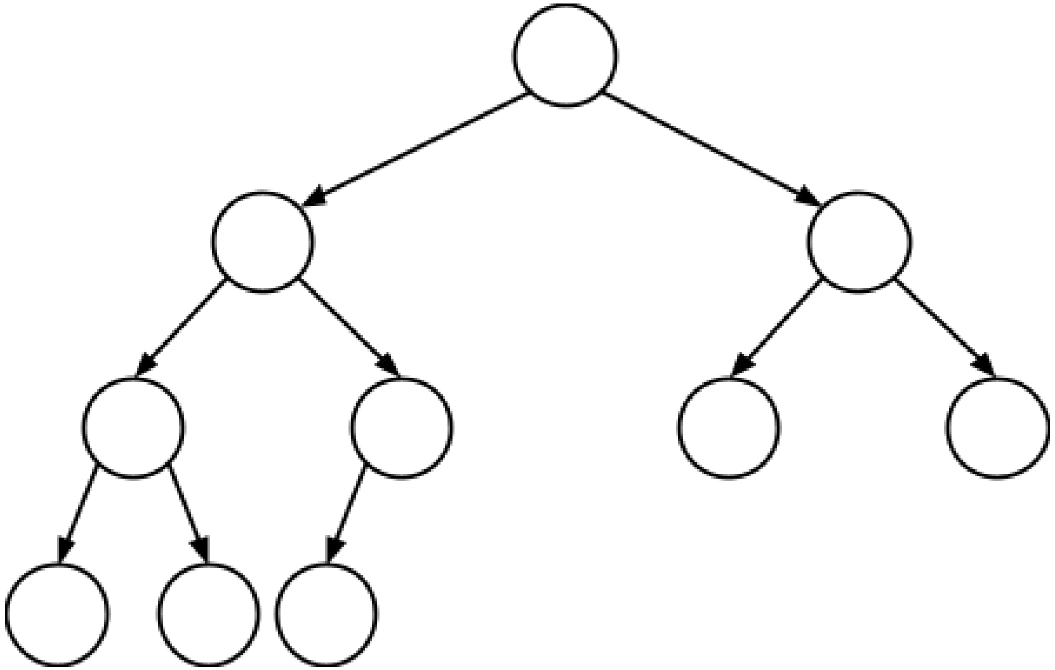
- *Full Binary Tree*: a binary tree that every interior node has exactly two children.
- *Perfect Binary Tree*: a full binary tree where all the leaves are at the same depth.
  - If a perfect binary tree has height h, then
    - It has 2^h leaves
    - It has $2^{h+1} – 1$ total nodes
  - If a perfect binary tree has n nodes, then its height is *approximately* log(n)





Binary Trees

- *Complete Binary Tree*: a binary tree that is perfect except for the deepest level, whose nodes are all as far left as possible



Binary Search Trees

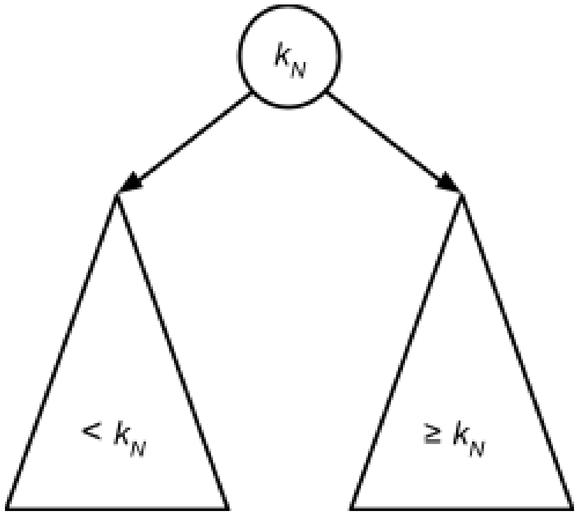
- Recall: each node in a tree represents a data element.

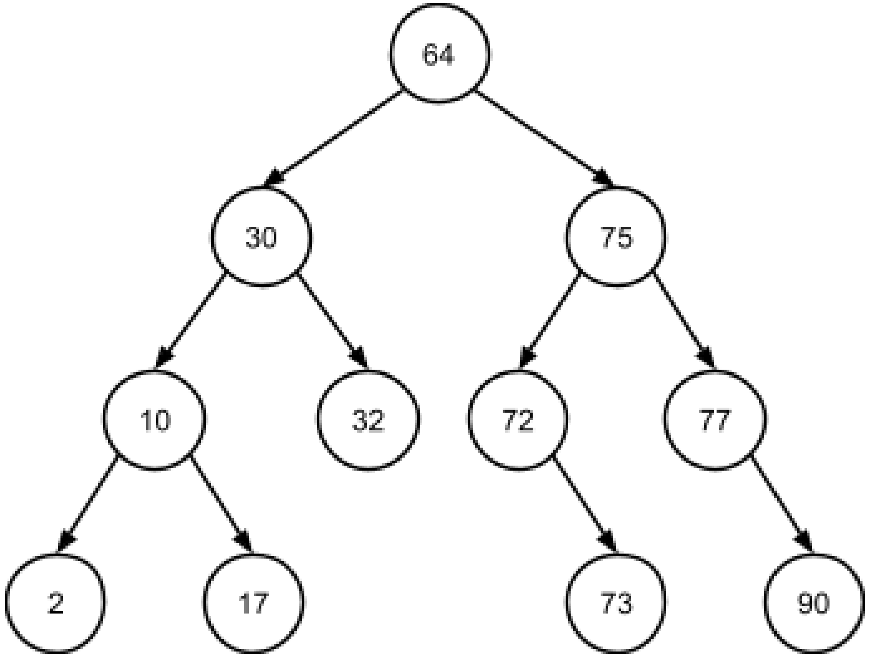
- Represent each data element using a **key** (identifier)
  - The data element may also contain other data, which we can refer to as its **value**

- Assuming these keys can be ordered in relation to others
  - i.e., integer keys can be ordered numerically, string keys can be ordered alphabetically

Binary Search Trees

- A *binary search tree* (**BST**) is a binary tree that:
  - *the key of each node N is **greater than** all the keys in N's left subtree and **less than or equal to** all the keys in N's right subtree*





- **Note: A BST does NOT have to be full, perfect, complete, etc.*

Next Lecture: BST Operations

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

Lecture Topics:

- Midterm Review

Midterm

- 2/13 Tuesday during lecture time (2:00 – 3:20)
- Same classroom
- Close book, close notes
- No calculator allowed
- Question types: multiple choices, T/F, short answer
  - Similar to your quizzes
- Bring pencil/pen, and your photo ID (student ID/driver license/passport)
- Scratch paper will be provided if needed

Midterm

- Topics: Week 1-5 (lecture 1-8):
  - C Basics
    - scanf()/printf()
    - Conditionals and loops
    - Struct
    - Pointers
      - void*
    - Stack vs. heap
    - C strings
    - Function pointers

char*   char[ ]

Midterm

- Topics: Week 1-5 (lecture 1-8):
  - Dynamic Arrays
    - Struct: data, size, capacity
    - Basic operations:
      - get()
      - set()
      - insert()
        - When to resize?
      - remove()
  - Linked List
    - Struct: val, next pointer
    - Basic operations:
      - Insert()
      - Remove()

Midterm

- Topics: Week 1-5 (lecture 1-8):
  - Complexity Analysis
    - Big O
    - Compute Runtime & Space complexity
      - Dominant Components
    - Best, worst, and average cases
    - Dynamic Array insertion
    - Linked list insertion
  - Stack
    - LIFO
    - Basic Operations:
      - Push()
      - Pop()
    - Implement stack using linked list vs. dynamic array
      - Complexity

Midterm

- Topics: Week 1-5 (lecture 1-8):
  - Queue
    - FIFO
    - Basic Operations
      - Enqueue()
      - Dequeue()
    - Implement queue using linked list vs. dynamic array
      - Complexity
      - Circular buffer: logical index vs. physical index
  - Deque
    - Basic operations:
      - Add front
      - Add back
      - Remove front
      - Remove back

Midterm

- Topics: Week 1-5 (lecture 1-8):
  - Deque
    - Implement deque using doubly linked list
      - Sentinels
      - Complexity
  - Encapsulation
  - Iterator
    - next()
    - has_next()
  - Binary Search
    - collection must be sorted
    - Complexity