Assignment 1: Part 1 of 3

Question 1: Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do to find the key 18? Show the steps.

Answer

For the input data set, 2 comparisons are required.

Steps

1. Set starting search position (“`index`”) to zero
2. If there are no elements in the list, proceed to Step 6
3. Extract and compare element in the list at the `index`-th position with the search value (“18”); If values are equal, proceed to Step 7
4. Increment `index` by 1
5. If `index` is greater than number of elements, proceed to Step 6, otherwise proceed to Step 3
6. Show message that value was not found and exit the routine
7. Show message that value was found after `index` + 1 attempts and exit the routine

Code Flow

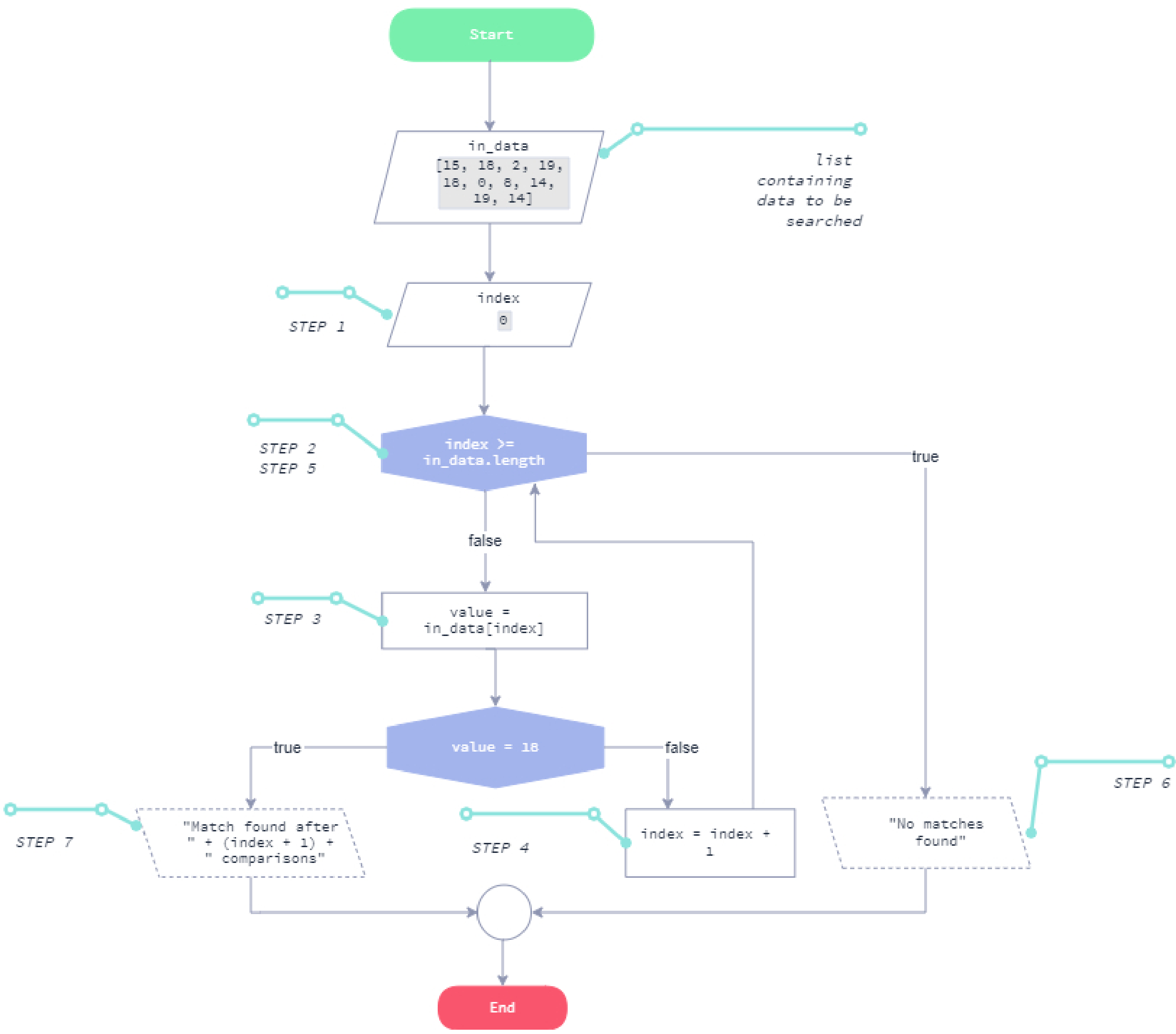


Figure 1 Flow chart showing steps of algorithm

Pseudocode Implementation

```
PROCEDURE Search (in_data: ITEMS, search_value: ITEM)
      DECLARE index and set value to zero

      `` Early termination check
      IF index GTE LENGTH(in_data) THEN
            PRINT No Matches Found
            EXIT PROCEDURE
      END IF


      `` This is the main loop for linear searching
      `` We only break if we have scanned ALL items OR
      `` we found a match

      LOOP WHILE index GTE LENGTH(in_data)

            IF in_data[index] = search_value THEN
                  `` We specify [index + 1] because we set
                  `` index to zero at the start. This is because
                  `` we assume our list implementation
                  `` is zero-based for indexing
                  ``
                  `` Therefore to show the one-based offset, we
                  `` increment the index by one

                  PRINT Match found after [index + 1] comparisons
                  EXIT PRODCEDURE
            END IF

            `` prepare to scan the next element
            INCREMENT index by 1

      END LOOP

      PRINT Value not found in dataset

END PROCEDURE
```

Analysis

- Sequential search requires testing every item in the dataset against a specific search value.
- It is assumed the search value is of the same *data type* as the elements in the dataset. For example, one cannot search for the key “Apples” in a dataset that contains numerical values.
- Complex search keys, such as Object Instances require custom comparers to check an object instance against a dataset’s value.

An example provided in JSON format:

```
in_data: [
      { "Name": "Test", "Age": "30" },
      { "Name": "Fun", "Age": "90" }
      ]

CALL Search (in_data, { "Name": "Jeff" })
```

Inside the `Search` algorithm, the line

```
IF in_data[index] = search_value THEN ...
```

will require a custom comparer associated with the "`=`" operator to return a true or false assessment of comparison. In this case the custom comparer could be designed to check the "`Name`" property only.

Performance

Sequential search has worst-case performance of O(n) since either the last element was a match or the search value is not present. Sequential search has best-case performance of O(1) if the very first element is a match.

Improvements

1. Use an ordered list of items.

   Algorithm stops searching if the value of an element in the dataset is greater than the search value. However, for custom comparers of object instances, it's up to the comparer to determine what is meant by "greater than the search value".

Alternatives

** the following comparisons based on a dataset of n=1 Million items*

Algorithm	Key Summary	Average Performance
Binary Search	• Requires a sorted array • Repeatedly searches key based on value of middle element in split array: values less than are searched in left split, values greater than are searched in right split	**O(log(n))** = 19 comparisons
Interpolation Search	• Requires sorted array with uniform distribution of values • Like binary search, but uses "estimation" to determine middle element	**O(log (log(n)))** = 4 comparisons *However, for non-uniform distribution: = 1000000 comparisons*
Jump Search	• Requires a sorted array • Improvement of Linear Search • Data set divided into "blocks" used for jumps. If value not found in a small block, algorithm "jumps" to the next block and linear search performed within block	**O(sqrt(n))** = 1000 comparisons
Hash Tables	• Search using key's computed hash value • Hash value provides direct index map into dataset	O(1) = 1

Recommendation

For almost all cases involving sorted data, I would recommend **Binary Search** algorithm. If datasets are not sorted, then I would make a secondary recommendation to use hash table lookups provided the clash ratio of computed hashes is extremely low.

Question 2: Suppose you have following list of numbers to sort [19, 1, 9, 7, 3, 10, 13, 15, 8, 12]. Show the partially sorted list after three complete phases of bubble sort.

Answer

Input	**19 ↔ 1**		**9**	**7**	**3**	**10**	**13**	**15**	**8**	**12**
	1	19 ↔ 9		7	3	10	13	15	8	12
		9	19 ↔ 7		3	10	13	15	8	12
			7	19 ↔ 3		10	13	15	8	12
PHASE 1				3	19 ↔ 10		13	15	8	12
					10	19 ↔ 13		15	8	12
						13	19 ↔ 15		8	12
							15	19 ↔ 8		12
								8	19 ↔ 12	
Phase Output	**1**	**9 ↔ 7**		**3**	**10**	**13**	**15**	**8**	**12**	**19**
		7	9 ↔ 3		10	13	15	8	12	19
PHASE 2			3	9	10	13	15 ↔ 8		12	19
							8	15 ↔ 12		19
								12	15	19
Phase Output	**1**	**7 ↔ 3**		**9**	**10**	**13**	**8**	**12**	**15**	**19**
		3	7	9	10	13 ↔ 8		12	15	19
PHASE 3						8	13 ↔ 12		15	19
							12	13	15	19
Result	**1**	**3**	**7**	**9**	**10**	**8**	**12**	**13**	**15**	**19**

Steps (Bubble Sort)

1. If there are no elements or less than two elements in the list, exit routine
2. Set `swap-counter` to zero
3. Set starting position (“`index`”) to zero

4. Swap element values at `index` and `index` + 1 if element value at `index` is greater. If swap occurred, increment `swap-counter`
5. Increment `index` by 1
6. If `index` is greater than the number of elements, proceed to Step 7 otherwise proceed to Step 4
7. If no swaps occurred, exit routine the list is sorted, otherwise proceed to Step 3

Code Flow

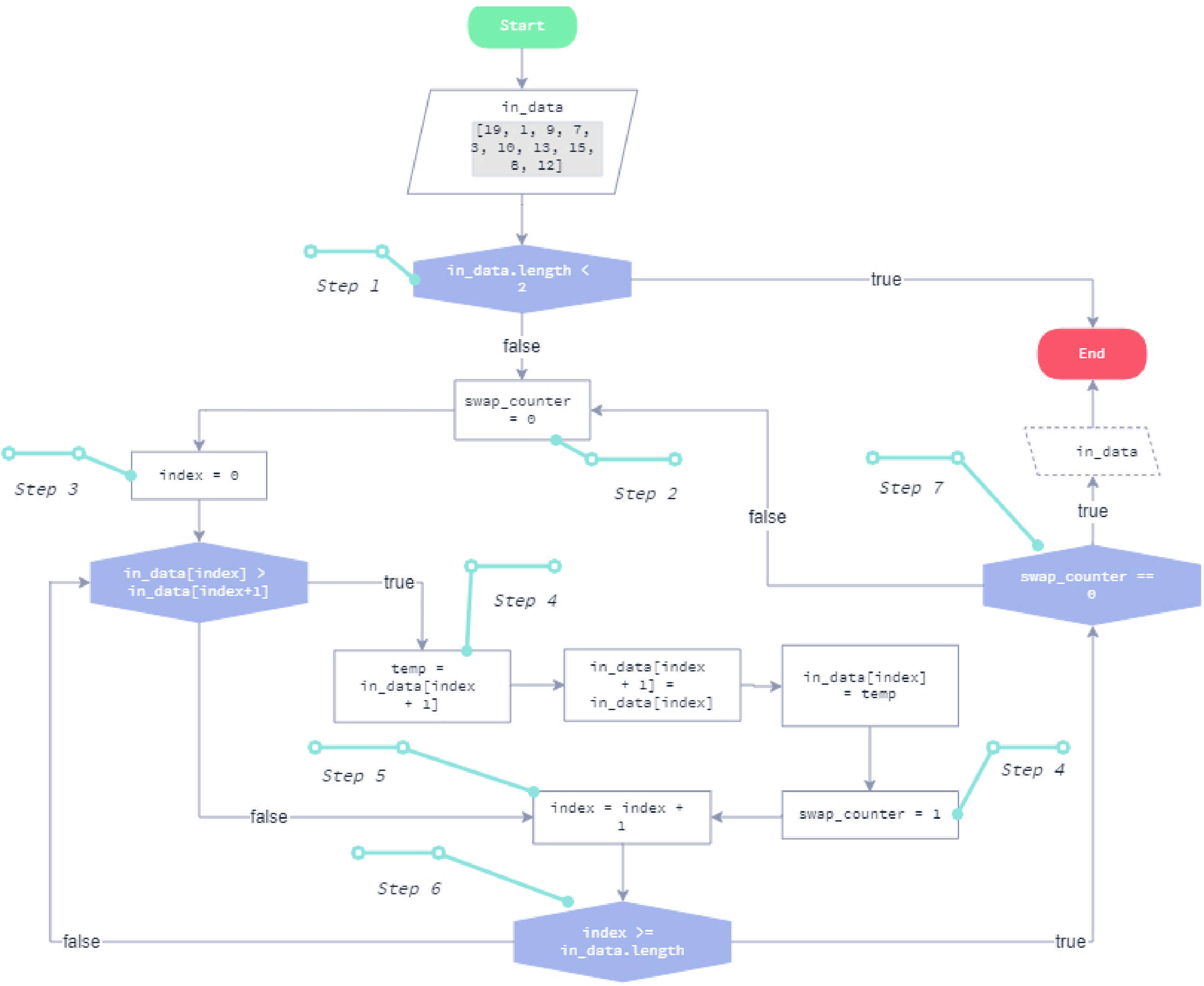


Figure 2 Flow chart for Bubble Sort algorithm

Pseudocode Implementation

```
PROCEDURE BubbleSort (in_data: ITEMS)
      `` Bubble sort requires a PAIR of items to compare against
      `` The minimal length of a dataset is therefore 2 items
      IF LENGTH(in_data) < 2 THEN
            EXIT PROCEDURE
      END IF

      `` This loop repeats until no swaps have occurred
      LOOP
            INITIALIZE swap_counter to zero
            INITIALIZE index to zero

            `` The following loop test every item in the dataset

            `` Test if we need to swap a pair of items based on
            `` their numerical sort order
            LOOP
                  IF in_data[index] > in_data[index+1] THEN
                        SWAP in_data[index] and in_data[index + 1]
                        INCREMENT swap_counter
                  END IF

                  `` continue to next element in the list
                  INCREMENT index
            UNTIL index >= LENGTH(in_data)

            IF swap_counter = 0 THEN
                  PRINT in_data
                  EXIT PROCEDURE
            END IF
      UNTIL swap_counter = 0

END PROCEDURE
```

Analysis

- Bubble sort algorithm gets the job done by repeatedly looping through the dataset to determine which elements ought to be swapped.
- The algorithm requires (n – 1) passes to sort all items.
- The algorithm terminates when no swaps have occurred.
- Bubble sort is simple to understand and implement and is a good introduction to the concept of sorting.

Common Alternatives

Algorithm	Type of Algorithm	Key Summary	Average Performance
Heap Sort		• Divide dataset into “sorted” and “unsorted” lists. • Unsorted list is maintained on a heap using a binary tree structure	O(n * log(n))
Insertion Sort		• Take elements one-by-one and insert into correct location by scanning dataset for insertion point • *Shell Sort is an improved algorithm*	O(n ^ 2)
Merge Sort	Divide-and-Conquer	• Divide dataset into smaller and smaller sets until only two items remain • Repeatedly *merge* these sub lists until a single list remains	O(n * log(n))
Quick Sort	Divide-and-Conquer	• Repeatedly use a “pivot” to partition data into two arrays based on pivot value	O(n * log(n))
Selection Sort		• Divide into sorted and unsorted lists • Repeatedly find lowest value in unsorted list and inserts it into sorted list • *Heap Sort performs better*	O(n ^ 2)

Considerations for Algorithms

- Algorithms differ in their ***space requirements***. Algorithms like Heap Sort do not require extra space to sort as they are deemed in-place.
- Algorithms exhibit ***time complexity*** which is expressed using Big-O notation (as shown in the “Average Performance”) column above.
- The use of ***parallelization*** in algorithm implementation can improve performance but require use of synchronization techniques.
- Sorting elements that are object instances, requires implementation of custom comparers.

Question 3: Given the statement below

```
x = BinaryTree('a')

insert_left(x,'b')

insert_right(x,'c')

insert_right(get_right_child(x),'d')

insert_left(get_right_child(get_right_child(x)),'e')
```

Which of these answers is the correct representation of the tree? Show your working out.

A. ['a', ['b', [], []], ['c', [], ['d', [], []]]]

B. ['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]

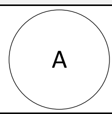
C. ['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]

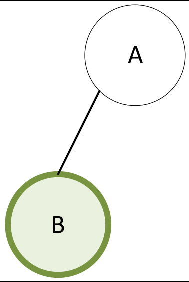
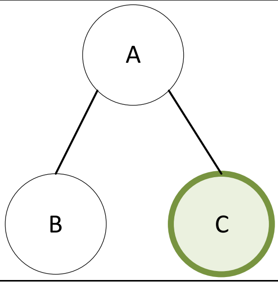
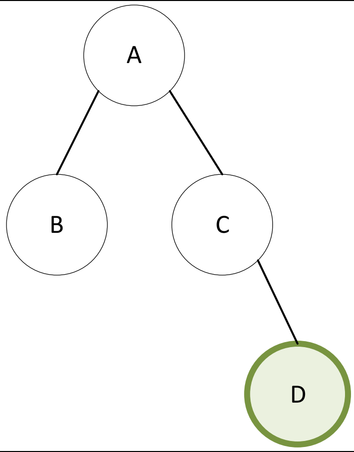
D. ['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

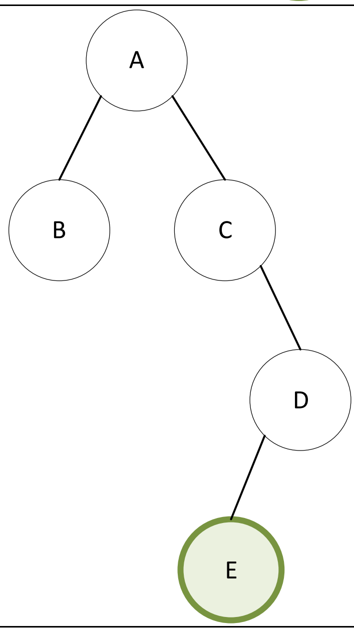
Answer

C is the correct representation of the code

Steps

Code	**Tree**
`x = BinaryTree('a')`	

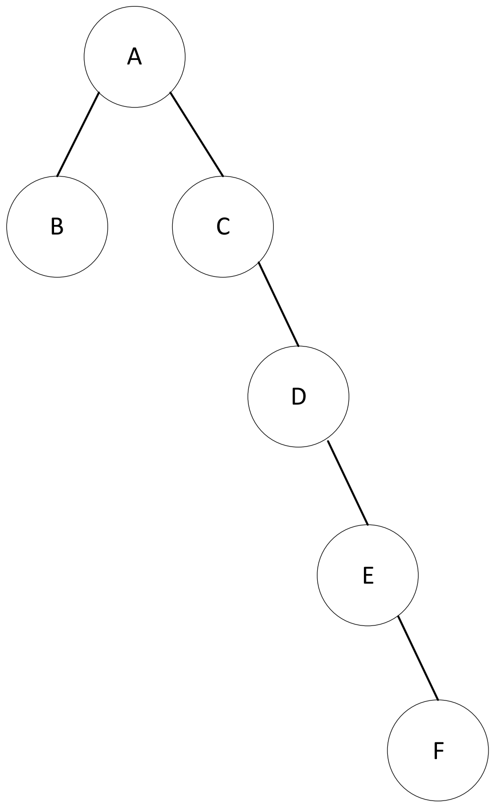
insert_left(x,'b')	
insert_right(x,'c')	
insert_right(get_right_child(x),'d')	

insert_left(
      get_right_child(
            get_right_child(x)),'e')
	

Analysis

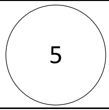
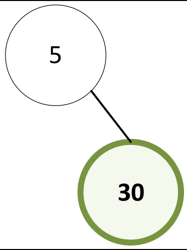
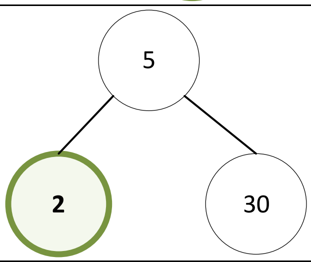
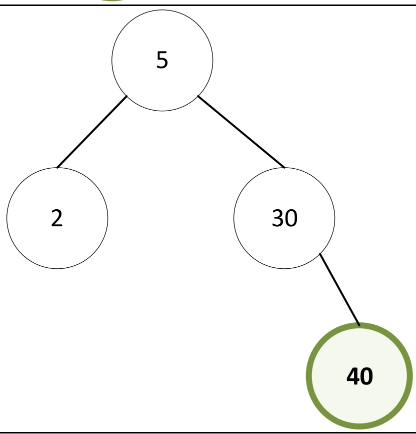
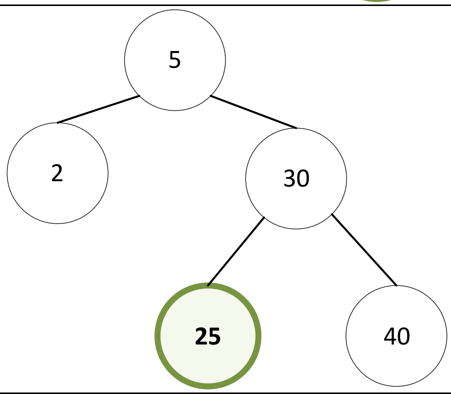
- The first element inserted into a binary tree becomes the “root” node from which all future searches and insertions are made
- Binary trees can be classified as “balanced” or “unbalanced”. A balanced tree has an equal number of child nodes distributed across each parent, while an unbalanced tree has many child nodes positioned in one portion of the tree.

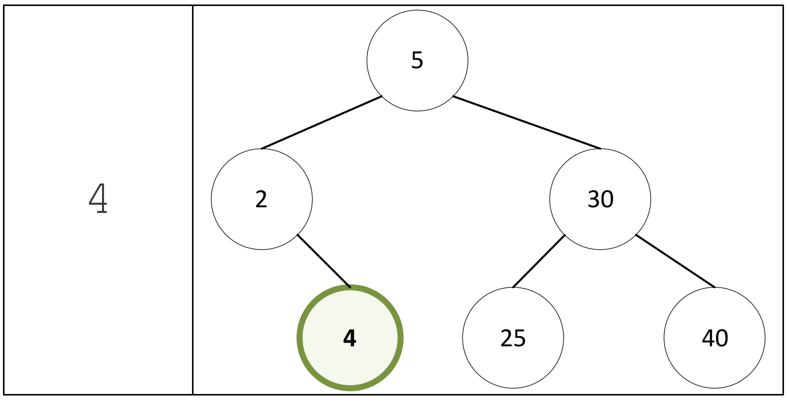
  For example, an unbalanced tree would look like this:



- Unbalanced binary trees degrade search performance and tend toward linear search time. For this reason, it is best practice to ensure binary trees are balanced to maintain optimal search performance.

Question 4: Draw a tree showing a correct binary search tree given that the keys were inserted in the following order 5, 30, 2, 40, 25, 4.

Binary Tree Input	Tree
5	
30	
2	
40	
25	



Assumptions

- Binary tree insertion allocates first input as root node
- Subsequent insertions are based on their *numerical* value compared to the root node.
- If the root node has not a **left** and **right** node, then the insertion continues based on an appropriate child node, which then serves (for context of insertion) as the root node

Considerations

- It is possible to insert non-primitive data types into a binary tree, however, the implementor must provide support for custom *comparers* for the algorithm to determine what is “less than” or “greater than” the root node

Alternatives

Tree	Key Summary
B- Tree	• Nodes are sorted into in-order traversal upon insertion or updates • Can have multiple child nodes • Is also known as a “balanced tree” • Found in DBMS systems
B+ Tree	• All leaves are at the same distance from the root

	• Is also known as a “balanced tree” • Root node can contain multiple children • Used by Microsoft’s NTFS file system