Lecture 1

Data Structures Definitions

- Ways to organize and store data
- Ways to access and manipulate the stored data.

Data structures

- A **data structure** is a systematic way of organizing a collection of data.
- A **static data structure** is one whose capacity is fixed at creation.
- A **dynamic data structure** is one whose capacity is variable, e.g.: a linked list, or binary tree.
- For each **data structure**, we need algorithms for insertion, deletion, searching, etc.
- **Algorithms:**
  - ✓ can be performed by humans or machines
  - ✓ can be expressed in any suitable language
  - ✓ may be as abstract as we like.
- **Programs:**
  - ✓ must be performed by machines
  - ✓ must be expressed in a programming language
  - ✓ must be detailed and specific.

Array

- An array is a collection of items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).
- Why Do You Need an Array in Data Structures?
  - ✓ Let's suppose a class consists of ten students, and the class must publish their results.
  - ✓ If you had declared all ten variables individually, it would be challenging to manipulate and maintain the data.
  - ✓ If more students were to join, it would become more difficult to declare all the variables and keep track of it.
  - ✓ To overcome this problem, arrays came into the picture.

Ordered Array

- The elements of an ordered array are arranged in ascending (or descending) order.

Stacks

- A container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle.
- Only the last (the most recently inserted) object can be removed.

Stack Operations

- Some stack operations:
  - ✓ push - add an item to the top of the stack
  - ✓ pop - remove an item from the top of the stack
  - ✓ peek - retrieves the top item without removing it
  - ✓ empty - returns true if the stack is empty

Queue

- Differs from a stack in that its insertion and removal follow the first-in-first-out (FIFO) principle.
- The element which has been in the queue the longest may be removed.

Queue Operations

- We can define the operations for a queue
- enqueue - add an item to the rear of the queue
- dequeue (or serve) - remove an item from the front of the queue
- empty - returns true if the queue is empty

linked list

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

- Insertion − Adds an element at the beginning of the list.
- Deletion − Deletes an element at the beginning of the list.
- Display − Displays the complete list.
- Search − Searches an element using the given key.
- Delete − Deletes an element using the given key.

Tree

- A collection of objects arranged hierarchically.
- E.g., organization of a corporation, a table of content, dos/Unix file systems, and family tree.
- The notion of parents and children, root, and leaves.

Binary Trees

- Why Use Binary Trees?
  - ✓ You can search a tree quickly, as you can with an ordered array,
  - ✓ you can also insert & delete items quickly, with a linked list.

Click on the word link to see the array code

Lecture 2

Arrays

- You want to store 5 numbers in a computer
  - ✓ Define 5 variables, e.g., num1, num2, ..., num5
- What, if you want to store 1000 numbers?
  - ✓ Defining 1000 variables is a good solution!
  - ✓ Requires much programming effort

Any better solution?

- Yes, some structured data type
  - ✓ The array is one of the most common structured data types.
  - ✓ The idea of an array is to represent many instances in one variable.

- An array in C++ is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using the indices of an array.
- They can be used to store collections of primitive data types such as int, float, double, and char.
- The array is a static data structure that cannot grow or shrink during program execution – its size is fixed.
- There are various ways in which we can declare an array. It can be done by specifying its type and size, initializing it, or both.
  - ✓ Array declaration by specifying the size
  - ✓ Array declaration by initializing elements
  - ✓ Array declaration by specifying the size and initializing elements
- Advantages of an Array in C/C++
  - ✓ Arrays represent multiple data items of the same type using a single name.
  - ✓ Elements can be accessed randomly by using the index number.
  - ✓ Easy access to all the elements.
  - ✓ Traversal through the array becomes easy using a single loop.
  - ✓ Sorting becomes easy as it can be accomplished by writing fewer lines of code.
- Disadvantages of an Array in C/C++
  - ✓ An array is a static structure, which allows a fixed number of elements to be entered that is decided at the time of declaration.
  - ✓ Allocating more memory than the requirement leads to a wastage of memory space and less allocation of memory also leads to a problem.

Unsorted array

Search in an unsorted array (Sequential Search)

- In an unsorted array, a search operation can be performed by linear traversal from the first element to the last element. The time complexity of search in an unsorted array is $\boldsymbol{O}(\boldsymbol{n})$

Click on the word link to see the code

Insert in an unsorted array

- In an unsorted array, the insert operation is faster as compared to a sorted array because we don't have to care about the position at which the element is to be placed. The time complexity of inserting in an unsorted array is $\boldsymbol{O}\ (\mathbf{1})$

Click on the word link to see the code

Delete in an unsorted array

- In the delete operation, the element to be deleted is searched using the linear search, and then the delete operation is performed followed by shifting the elements. The time complexity of delete in an unsorted array is $\boldsymbol{O}(\boldsymbol{n})$

Click on the word link to see the code

Sorted array

Search in the sorted array (Binary Search)

- In a sorted array, the search operation can be performed by using binary search. The Time Complexity of the Search Operation in a sorted array is $\boldsymbol{O}\ (\boldsymbol{Log\ n})$ [Using Binary Search]

Click on the word link to see the code

Insert in a sorted array

- In sorted arrays, we care about the position at which the element is placed, unlike the unsorted array. The Time Complexity of the Insert Operation in a sorted array is $\boldsymbol{O}(\boldsymbol{n})$ [In the worst case all elements may have to be moved]

Click on the word link to see the code

Delete in a sorted array

- In the delete operation, the element to be deleted is searched using binary search, and then the delete operation is performed followed by shifting the elements. The Time Complexity of the Delete Operation in a sorted array is $\boldsymbol{O}\ (\boldsymbol{n})$ [In the worst case all elements may have to be moved]

Click on the word link to see the code

Time Complexity

Operation	*Unsorted Array*	*Sorted Array*
Search	*O (n)*	*O (Log n)*
Insert	*O (1)*	*O (n)*
Delete	*O (n)*	*O (n)*

Lecture 3

Sorting Techniques

Example

- Original list:
  - ✓ 10, 30, 20, 80, 70, 10, 60, 40, 70
- Sorted in non-decreasing order:
  - ✓ 10, 10, 20, 30, 40, 60, 70, 70, 80
- Sorted in non-increasing order:
  - ✓ 80, 70, 70, 60, 40, 30, 20, 10, 10

Issues in Sorting

- Many issues are there in sorting techniques
- How to rearrange a given set of data?
- Which data structures are more suitable to store data before their sorting?
- How fast the sorting can be achieved?
- How sorting can be done in a memory constraint situation?
- How to sort various types of data?

Sorting by Comparison

- The basic operation involved in this type of sorting technique is comparison. A data item is compared with other items in the list of items to find its place in the sorted list.
  - ✓ Insertion:
    - From a given list of items, one item is considered at a time. The item chosen is then inserted into an appropriate position relative to the previously sorted items. The item can be inserted into the same list or a different list.
      - ❖ e.g.: Insertion sort
  - ✓ Selection:
    - First, the smallest (or largest) item is located, and it is separated from the rest; then the next smallest (or next largest) is selected, and so on until all items are separated.
      - ❖ e.g.: Selection sort, Heap sort
  - ✓ Exchange:
    - If two items are found to be out of order, they are interchanged. The process is repeated until no more exchange is required.
      - ❖ e.g.: Bubble sort, Shell Sort, Quick Sort
  - ✓ Enumeration:
    - Two or more input lists are merged into an output list and while merging the items, an input list is chosen following the required sorting order.
      - ❖ e.g.: Merge sort

Sorting by Distribution

- No key comparison takes place
- All items under sorting are distributed over an auxiliary storage space based on the constituent elements in each and then grouped to get the sorted list.
- Distributions of items based on the following choices
  - ✓ Radix - An item is placed in a space decided by the bases (or radix) of its components with which it is composed.
  - ✓ Counting - Items are sorted based on their relative counts.
  - ✓ Hashing - Items are hashed, that is, dispersed into a list based on a hash function.
- Note: This lecture concentrates only on sorting by comparison.

Insertion Sort

- In insertion sort, each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements.
- We divide our array into a sorted and an unsorted array
- Initially, the sorted portion contains only one element: the first element in the array.
- We take the second element in the array and put it into its correct place
- Example:
  - ✓ 99 | 55 4 66 28 31 36 52 38 72
  - ✓ 55 99 | 4 66 28 31 36 52 38 72
  - ✓ 4 55 99 | 66 28 31 36 52 38 72
  - ✓ 4 55 66 99 | 28 31 36 52 38 72
  - ✓ 4 28 55 66 99 | 31 36 52 38 72
  - ✓ 4 28 31 55 66 99 | 36 52 38 72
  - ✓ 4 28 31 36 55 66 99 | 52 38 72
  - ✓ 4 28 31 36 52 55 66 99 | 38 72
  - ✓ 4 28 31 36 38 52 55 66 99 | 72
  - ✓ 4 28 31 36 38 52 55 66 72 99 |

Click on the word link to see the code

Selection Sort

- Selection sort is a sorting algorithm that works as follows:
  - ✓ Find the minimum value in the list
  - ✓ Swap it with the value in the first position
  - ✓ Repeat the steps above for the remainder of the list (starting at the second position)
- Example:
  - ✓ 26 33 43 100 46 88 52 17 53 77
  - ✓ 17 | 33 43 100 46 88 52 26 53 77
  - ✓ 17 26 | 43 100 46 88 52 33 53 77
  - ✓ 17 26 33 | 100 46 88 52 43 53 77
  - ✓ 17 26 33 43 | 46 88 52 100 53 77
  - ✓ 17 26 33 43 46 | 88 52 100 53 77
  - ✓ 17 26 33 43 46 52 | 88 100 53 77
  - ✓ 17 26 33 43 46 52 53 | 100 88 77
  - ✓ 17 26 33 43 46 52 53 77 | 88 100
  - ✓ 17 26 33 43 46 52 53 77 88 | 100

Click on the word link to see the code

Bubble Sort

- The sorting process proceeds in several passes.
  - ✓ In every pass, we go on to compare neighboring pairs and swap them if out of order.
  - ✓ In every pass, the largest of the elements under consideration will bubble to the top (i.e., the right).
- **How do you make the best case with (n-1) comparisons only?**
  - ✓ By maintaining a variable flag, check if there have been any swaps in each pass.
  - ✓ If not, the array is already sorted.

Click on the word link to see the code

Efficient Sorting algorithms

- Two of the most popular sorting algorithms are based on the divide-and-conquer approach.
  - ✓ Quick sort
  - ✓ Merge sort
- The basic concept of the **divide-and-conquer** method:
  - ✓ sort (list)
  - ✓ {
  - ✓   if the list has a length greater than 1
  - ✓   {
  - ✓   Partition the list into lowlist and highlist;
  - ✓   sort (lowlist);
  - ✓   sort (highlist);
  - ✓   combine (lowlist, highlist);
  - ✓   }
  - ✓ }

Quick Sort

How does it work?

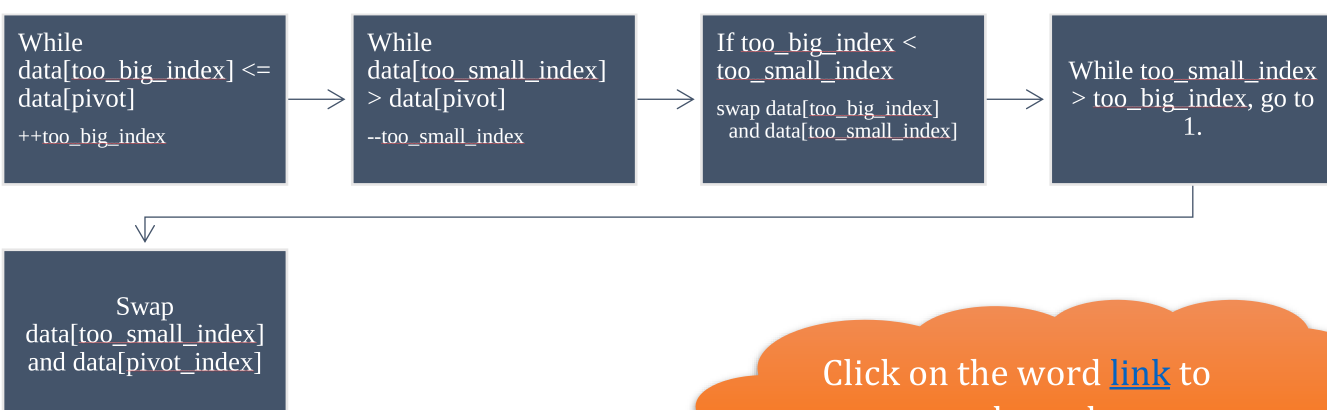
- **At every step, we select a pivot element in the list (usually the first element).**
  - ✓ We put the pivot element in the final position of the sorted list.
  - ✓ All the elements less than or equal to the pivot element are to the left.
  - ✓ All the elements greater than the pivot element is to the right.

Quicksort Algorithm

- **Given an array of n elements (e.g., integers):**
  - ✓ If the array only contains one element, return
  - ✓ Else
    - ▪ pick one element to use as a pivot.
    - ▪ Partition elements into two sub-arrays:
      - ❖ Elements less than or equal to the pivot
      - ❖ Elements greater than the pivot
    - ▪ Quicksort two sub-arrays
    - ▪ Return results

Partitioning Array

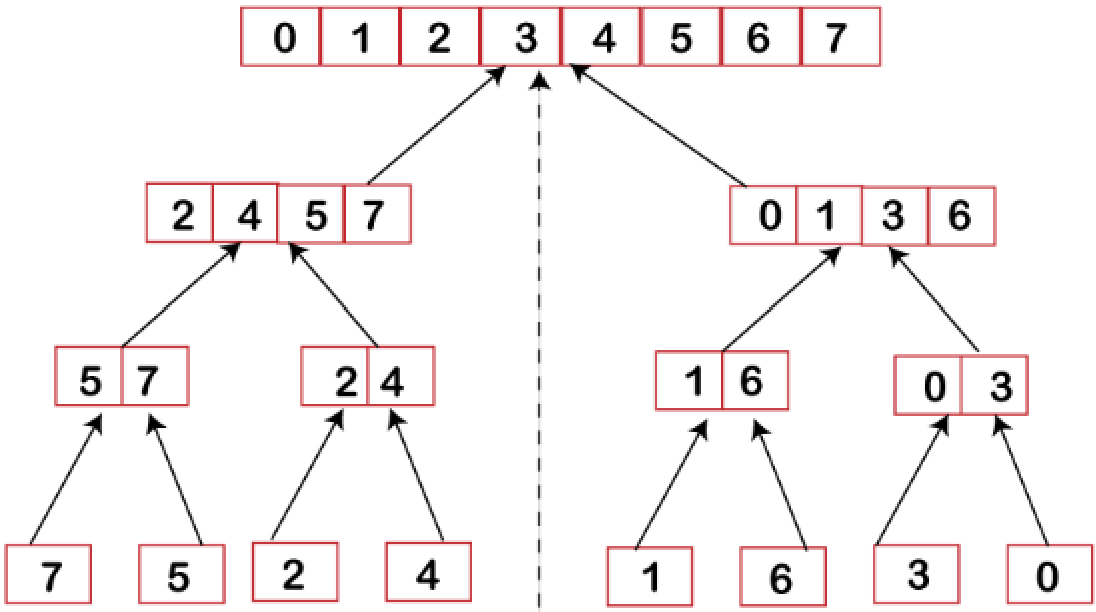
- **Given a pivot, partition the elements of the array such that the resulting array consists of:**
  - ✓ One sub-array that contains elements >= pivot
  - ✓ Another sub-array that contains elements < pivot
- The sub-arrays are stored in the original data array.
- Partitioning loops through, swapping elements below/above the pivot.

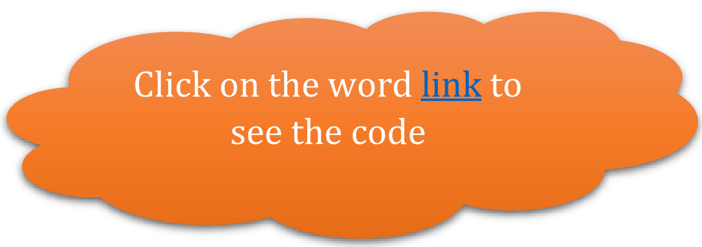


Click on the word link to see the code

Merge Sort

How does it work?





Quick Sort vs. Merge Sort

Both algorithms divide the problem into two sub-problems.

- Merge sort:
  - ✓ two subproblems are of almost equal size always.
- Quick sort:
  - ✓ an equal subdivision is not guaranteed.
- This difference between the two sorting methods appears as the deciding factor of their run time performances.

Lecture 4

Stack-based on the array

- A stack is a linear data structure that stores a group of homogeneous items of elements.
  - ✓ Elements are added to and removed from the top of the stack i.e., The last element to be added is the first to be removed *LIFO* (Last in First Out) or *FILO* (First in Last Out).
- Main Functions
  - ✓ Boolean IsEmpty()
    - Function: Check whether the stack is empty or not.
  - ✓ Boolean IsFull()
    - Function: Check whether the stack is full or not.
  - ✓ Push (newItem)
    - Function: Adds newItem to the top of the stack.
    - Preconditions: The stack has been initialized and is not full.
  - ✓ Pop ()
    - Function: Removes topItem from the stack and returns it in item.
    - Preconditions: The stack has been initialized and is not empty.
  - ✓ Peek () or Top ()
    - Function: Return topItem from the stack.
    - Preconditions: The stack has been initialized and is not empty.

Stack Implementation

- To implement a stack, we need to define a class named Stack and use an array for storing their elements, and define a function for each main operation.

Click on the word link to see the code

Stack-based on the pointer

Linked Implementation of Stacks

- The disadvantage of array (linear) stack representation
  - ✓ A fixed number of elements can be pushed onto a stack
- Solution
  - ✓ Use pointer variables to dynamically allocate, and deallocate memory
  - ✓ Use a linked list to dynamically organize data
- Value of `stackTop`: linked representation
  - ✓ Locates the top element in the stack
    - ▪ Gives address (memory location) of the top element of the stack

Default Constructor

- When the stack object declared
  - ✓ Initializes stack to an empty state
  - ✓ Sets `stackTop` to `NULL`

Empty Stack and Full Stack

- Stack empty if `stackTop` is `NULL`
- Stack never full
  - ✓ Element memory allocated/deallocated dynamically
  - ✓ Function `isFullStack` always returns a false value

Initialize Stack

- Reinitializes stack to an empty state
- Because the stack might contain elements and you are using a linked implementation of a stack
  - ✓ Must deallocate memory occupied by the stack elements, set `stackTop` to `NULL`

Push

- `newElement` added at the beginning of the linked list pointed to by `stackTop`
- Value of pointer `stackTop` updated

Return the Top Element

- Returns information of the node to which `stackTop` pointing

Pop

- Removes the top element of the stack
  - ✓ Node pointed to by `stackTop` removed
  - ✓ Value of pointer `stackTop` update

Copy Stack

- ✓ Makes an identical copy of a stack

Click on the word link to see the code

Lecture 5

Linked List

Declarations for Linked Lists

- A program can keep track of the front node by using a pointer variable such as `head_ptr`.
- Notice that `head_ptr` is not a node -- it is a pointer to a node.
- We represent the empty list by storing `null` in the head pointer.

Inserting a Node at the Front

- `insert_ptr = new node;`
- Place the data in the new node's `data_field`.
- Connect the new node to the front of the list.
- When the function returns, the linked list has a new node at the front.

Click on the word link to see the

Caution

- Always make sure that your linked list functions work correctly with an empty list.

Pseudocode for Inserting Nodes

- Nodes are often inserted at places other than the front of a linked list.
- There is a general pseudocode that you can follow for any insertion function. . .
- The process of adding a new node in the middle of a list can also be incorporated as a separate function. This function is called `list_insert` in the linked list

Pseudocode for Removing Nodes

- Nodes often need to be removed from a linked list.
- As with insertion, there is a technique for removing a node from the front of a list and a technique for removing a node from elsewhere.

Summary

- It is easy to insert a node at the front of a list.
- The linked list toolkit also provides a function for inserting a new node elsewhere
- It is easy to remove a node at the front of a list.
- The linked list toolkit also provides a function for removing a node elsewhere--you should read about this function and the other functions of the toolkit.

Doubly Linked List

- A Doubly linked list is a bidirectional linked list, i.e., you can traverse it from head to tail node or tail to head node.
- Unlike singly linked lists, its node has an extra pointer that points at the previous node.
- Each node consists of three elements: one holds the data, and another two are the next and the previous node's pointers.
- These two pointers help us to go forward or backward from a particular node.

INSERT A NODE

- Case 1: Insertion in an empty list
- Case 2: Insertion at the beginning of a nonempty list
- Case 3: Insertion at the end of a nonempty list
- Case 4: Insertion somewhere in a nonempty list
- Both cases 1 and 2 require us to change the value of the pointer first.
- Cases 3 and 4 are similar. After inserting an item, the count is incremented by 1.

DELETE A NODE

- We first search the list to see whether the item to be deleted is in the list.
- This operation (if the item to be deleted is in the list) requires the adjustment of two pointers in certain nodes.
- Case 1: The list is empty.
- Case 2: The item to be deleted is in the first node of the list, which would require us to change the value of the pointer first.
- Case 3: The item to be deleted is somewhere in the list.
- Case 4: The item to be deleted is not on the list

Click on the word link to see the code

Lecture 6

Queue

- We introduce the `queue` data type.
- Several example applications of queues are given in that chapter.
- This presentation describes `queue` operations and two ways to implement a queue.

The Queue Operations

- A `queue` is like a line of people waiting for a bank teller.
- The `queue` has a `front` and a `rear`.
- New people must enter the `queue` at the `rear`.
- The C++ `queue` class calls this a `push`, although it is usually called an `enqueue` operation.
- When an item is taken from the `queue`, it always comes from the `front`.
- The C++ queue calls this a `pop`, although it is usually called a `dequeue` operation.

The Queue Class

- The C++ standard template library has a queue template class.
- The template parameter is the type of items that can be put in the `queue`.

Overflow vs underflow

- If a program attempts to add an entry to a queue that is already at its capacity, this is, of course, an error.
- This error is called queue overflow.
- If a program attempts to remove an entry from an empty queue, that is another kind of error, called queue underflow.

Array Implementation

- A queue can be implemented with an array, as shown here. For example, this `queue` contains the integers 4 (at the `front`), 8, and 6 (at the `rear`).
- The easiest implementation also keeps track of the number of items in the `queue` and the index of the first element (at the `front` of the `queue`), and the last element (at the `rear`).

A Dequeue Operation

- When an element leaves the `queue`, size is decremented, and first changes, too.

An Enqueue Operation

- When an element enters the `queue`, size is incremented, and the last changes, too.

At the End of the Array

- There is special behavior at the end of the array. For example, suppose we want to add a new element to this `queue`, where the last index is [5]
- The new element goes at the `front` of the array (if that spot isn't already used)

Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Special behavior is needed when the rear reaches the end of the array.

Uses for Queues

- Suppose you want a program to read a word and then write the word.
- One way to accomplish this task is to read the input one letter at a time and place each letter in a `queue`.
- After the word is read, the letters in the `queue` are written out.
- Because a `queue` is a First-In/First-Out data structure, the letters are written in the same order in which they were read.

Recognizing Palindromes

- A palindrome is a string that reads the same forward and backward; that is, the letters are the same whether you read them from right to left or from left to right.
- For example, the one-word string "radar" is a palindrome.
- Able was I ere I saw Elba
- We can do this by using both a `stack` and a `queue`.
- We will read the line of text into both a `stack` and a `queue`
- The program can simply compare the contents of the `stack` and the `queue` character-by-character to see if they would produce the same string of characters.

Summary

- Like `stacks`, `queues` have many applications.
- Items enter a `queue` at the `rear` and leave a `queue` at the `front`.
- Queues can be implemented using an `array` or using a `linked list`.

Click on the word link to see the code

Lecture 7

Tree

Binary Trees

- A binary tree has nodes, like nodes in a linked list structure.
- Data of one sort or another may be stored at each node.
- But it is the connections between the nodes which characterize a binary tree.
- *A binary tree is a finite set of nodes.*
- The set might be empty (no nodes, which is called the empty tree).
- But if the set is not empty, it follows these rules:
  - ✓ There is one special node, called the **root**.
  - ✓ Each node may be associated with up to two other different nodes, called its **left child and its right child.**
  - ✓ If node c is the child of another node p, then we say that "**p is c's, parent**."
  - ✓ Each node, except the root, has exactly one parent; **the root has no parent**.

Subtree

- Any node in a tree also can be viewed as the root of a new, smaller tree.

Left and right subtrees

- For a node in a binary tree, the node begins with its left child, and below is its **left subtree**.
- The nodes begin with their right child and below is its **right subtree**.

Depth of a node.

- Suppose you start at a node n and move upward to its parent.
- We'll call this "one step." Then move up to the parent of the parent—that's a second step.
- Eventually, you will reach the root, and the number of steps taken is called the **depth of the node n.**

Depth of a tree/ height of a tree

- The depth of a tree is the maximum depth of any of its leaves
- the leaf containing 13 has a depth of three, and there is no deeper leaf.
- If a tree has only one node, the root, then its depth is zero (since the depth of the root is zero).
- The empty tree doesn't have any leaves, so we use –1 for its depth.
- If a tree has only one node, the root, then its depth is zero (since the depth of the root is zero).
- The empty tree doesn't have any leaves, so we use –1 for its depth.

Full binary trees

- In a full binary tree, every leaf has the same depth,
- and every non-leaf has two children.

Complete binary trees

- To be a complete tree, every level except the deepest must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- In a complete binary tree, all the depths are full, except perhaps for the deepest. At the deepest depth, the nodes are as far left as possible.
- A complete binary tree is a special kind of binary tree that will be useful to us.

Tree class

- Such a class will have at least two private member variables:
  - ✓ The array itself is one member variable, and
  - ✓ A second member variable keeps track of how much of the array is used.
- The actual links between the nodes are not stored.

Tree links

- Instead, these links exist only via the **formulas** that determine where an item is stored in the array based on the item's position in the tree.

Summary

- Binary trees contain nodes.
- Each node may have a left child and a right child.
- If you start from any node and move upward, you will eventually reach the root.
- Every node except the root has one parent. The root has no parent.
- Complete binary trees require the nodes to fill in each level from left to right before starting the next level.