Data Structures (15B11CI311)

Odd Semester 2020

3rd Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Batches : B1 , B2 , B13

Faculty : Ankita Wadhwa

Lecture 2 – Introduction to Data Structures

Outline

- ✓ Data Structures
- ✓ Discussion of Abstract data type
- ✓ Need of Linear and Non-linear data structures using examples
- ✓ Static and dynamic memory allocation
- ✓ memory allocation for arrays and linked list

Atomic and Composite Data

Atomic Data

- ✓ Consists of a single piece of information.
- ✓ Cannot be further divided into other meaningful components.
- ✓ Example: a four digit car number
- ✓ A students grade

In programming languages we can use primitive (atomic) data types to represent these. Ex: Integer, char, float

Composite Data

- ✓ Consists of multiple pieces of information
- ✓ Can be divided into meaningful sub components.
- ✓ Example: String can be divided into characters
- ✓ Telephone number can be divided into area code and unique phone number.

In programming languages we can use composite data types to represent these. Example: Structures, classes

Data Type

A data type consists of two parts:

- ✓ A set of data (values)
- ✓ A set of operations (that can be performed on the data/values)

Type	Value	Operations
Integer	-2,-1,0,1,2	+, -, *, /
Character	‘a’,’b’, ‘A’,’B’, ‘$’	<,>,+, =

Data Structures

- ✓ It is aggregation of atomic or/and composite data into a set with defined relationships.

  (that relationship can be linear, tree type , graph type)

- ✓ Here, **Structure means a set of rules that define how data holds together.**
- ✓ Thus, a data structure is a scheme/way for organizing data in the memory of a computer.
- ✓ The way in which the data is organized affects the performance of a program for different tasks.
- ✓ Every data structure (organization of data) has its own strengths, and weaknesses.
- ✓ Also, every data structure specially suits to specific problem types depending upon the operations performed and the data organization.

Abstract Data type (ADT)

A type whose behavior is defined using:

- ✓ Declaration of Data
- ✓ Declaration of Operations
- ✓ Encapsulation of Data and Operations

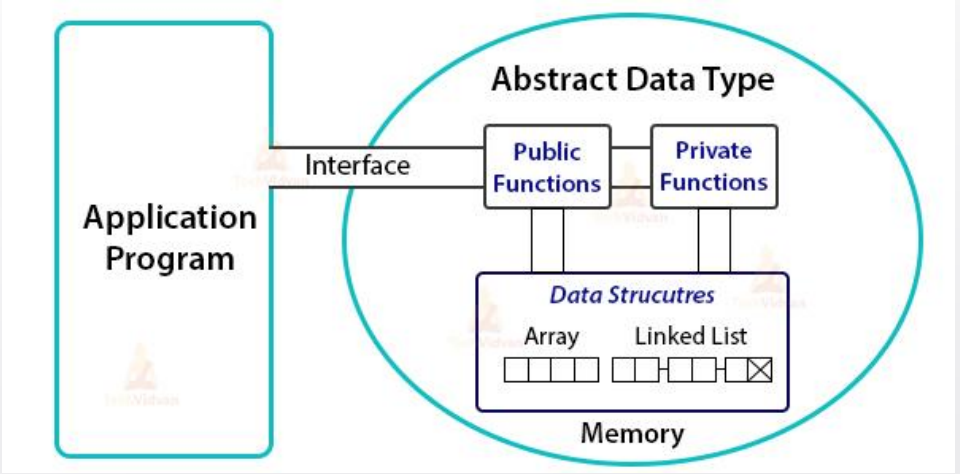


Image source: https://techvidvan.com/tutorials/java-abstract-data-type/

With ADT, users are not bothered about "How the task is done" but rather "what it can do?"

When a programmer creates an ADT, he only mentions the operations that can be performed. (and not how they can be performed).

Data Structures vs ADT

- **A data structure is a concrete implementation of the contract (rules/abstraction) provided by an ADT.**

What that means is:
1. **ADT is abstract** representation, from the user point of view
2. **DS is concrete** representation, *not* from the user point of view

In simple terms:
1. **Stack is ADT**, it only specifies that there should be a push, pop, etc. for the user to use.
2. Stack (like other ADTs like Queue) **is implemented using DS like *array or list (linked list, etc.)***

ADT Examples

- ADT examples: Stack, Queue, List etc.
- ADT operations:

Stack operations –

- isFull(): to check whether stack is full or not
- isEmpty(): to check whether stack is empty or not
- push(x): to push x into the stack
- pop(): to delete one element from top of the stack
- peek(): to get the top most element of the stack
- size(): to get number of elements present into the stack

ADT Examples

Queue operations –

- isFull(): to check whether queue is full or not
- isEmpty(): to check whether queue is empty or not
- insert(x): to add x into the queue at the rear end
- delete(): to delete one element from the front end of the queue
- size(): to get number of elements present into the queue

ADT Examples

List operations –

- insert(x): to insert one element into the list
- remove(x): to remove given element from the list
- size(): to get number of elements present into the list
- get(i): to get element at position i
- replace(x, y):to replace x with y value

Need of Linear and Non-linear data structures using examples

Linear Data Structure

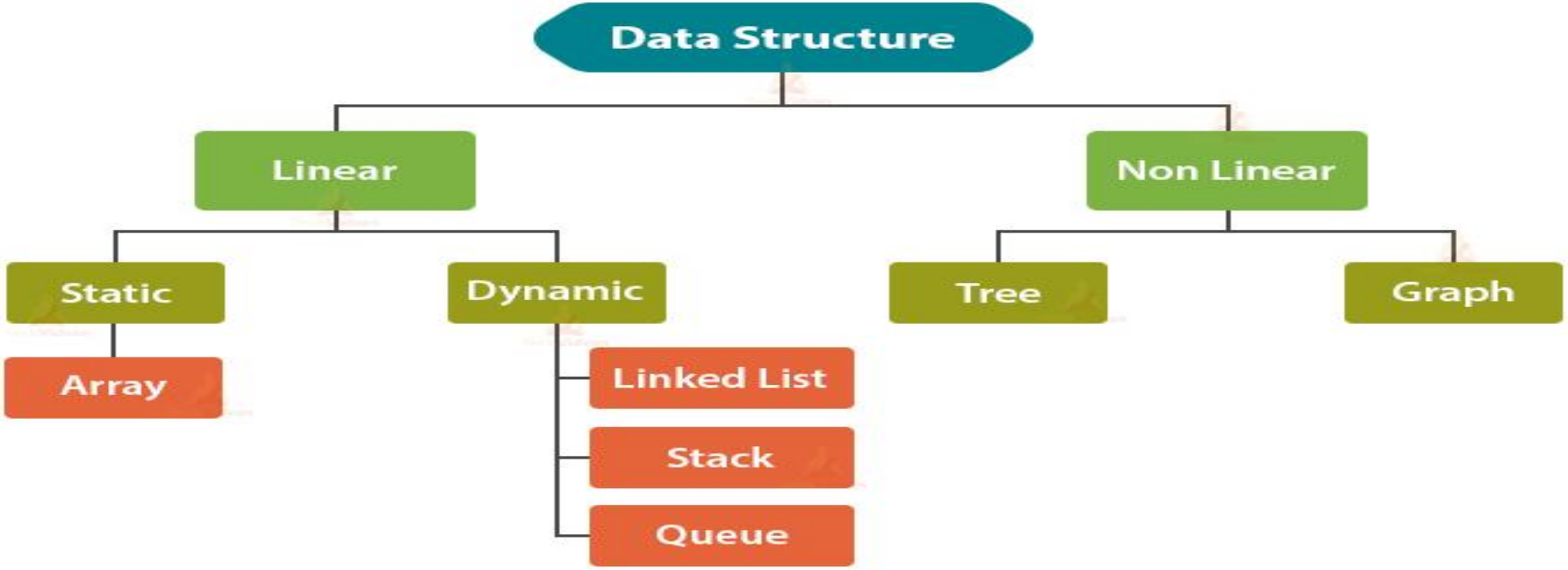
- data elements are arranged Linearly (sequentially)
- The elements are attached to its previous and next adjacent
- Only single level is involved.
- Therefore, all the elements can be traversed in single run only.
- easy to implement because computer memory is arranged in a linear way.
- Retrieval of elements can be costly.
- Examples: array, stack, queue, linked list, etc.

Need of Linear and Non-linear data structures using examples

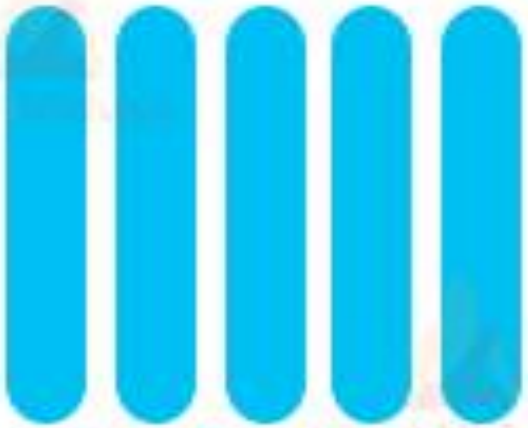
Non-linear Data Structure

- data elements are not arranged linearly.
- single level is not involved.
- All the elements can not be traversed in single run only.
- not easy to implement in comparison to linear DS.
- It utilizes computer memory efficiently in comparison to a linear data structure.
- Retrieval of elements can be efficient.
- Examples: trees and graphs.

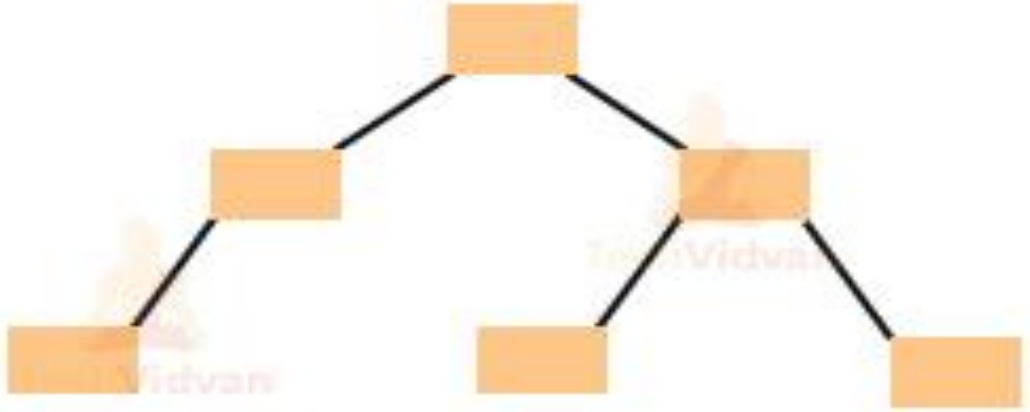
Linear and Non-linear data structures



Source: https://techvidvan.com/tutorials/data-structure-in-java/

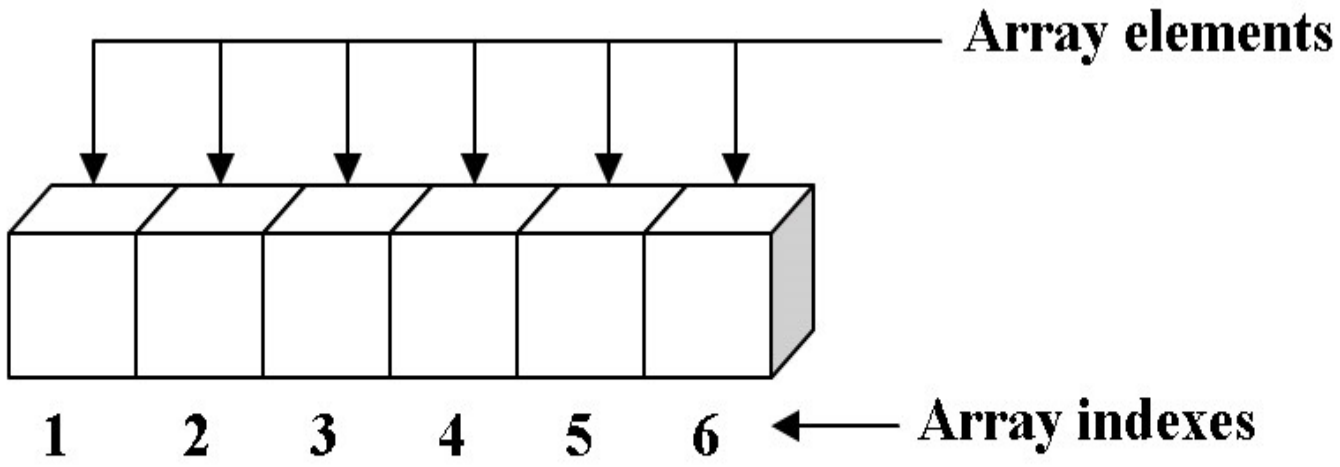


(i) Linear Data Structure



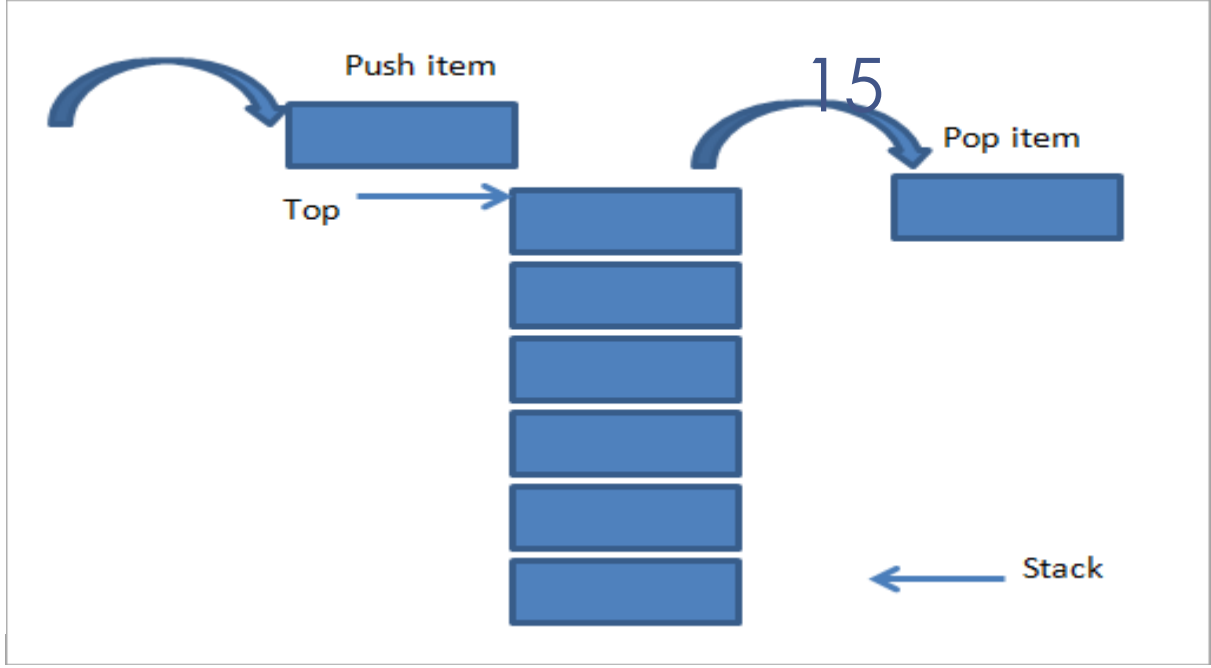
(ii) Non-Linear Data Structure

Source: https://techvidvan.com/tutorials/data-structure-in-java/

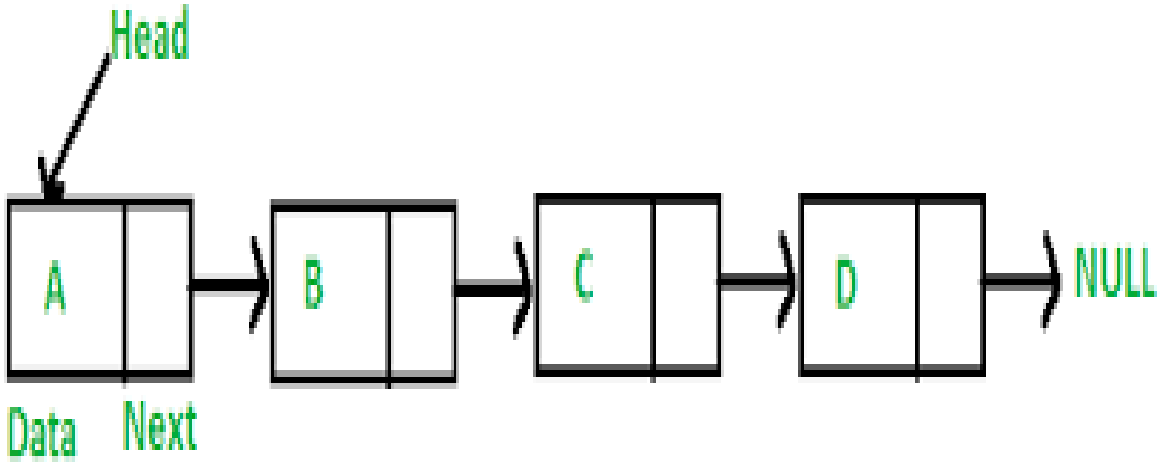


Source: http://123codegenerator.blogspot.com/2010/05/array-data-type.html

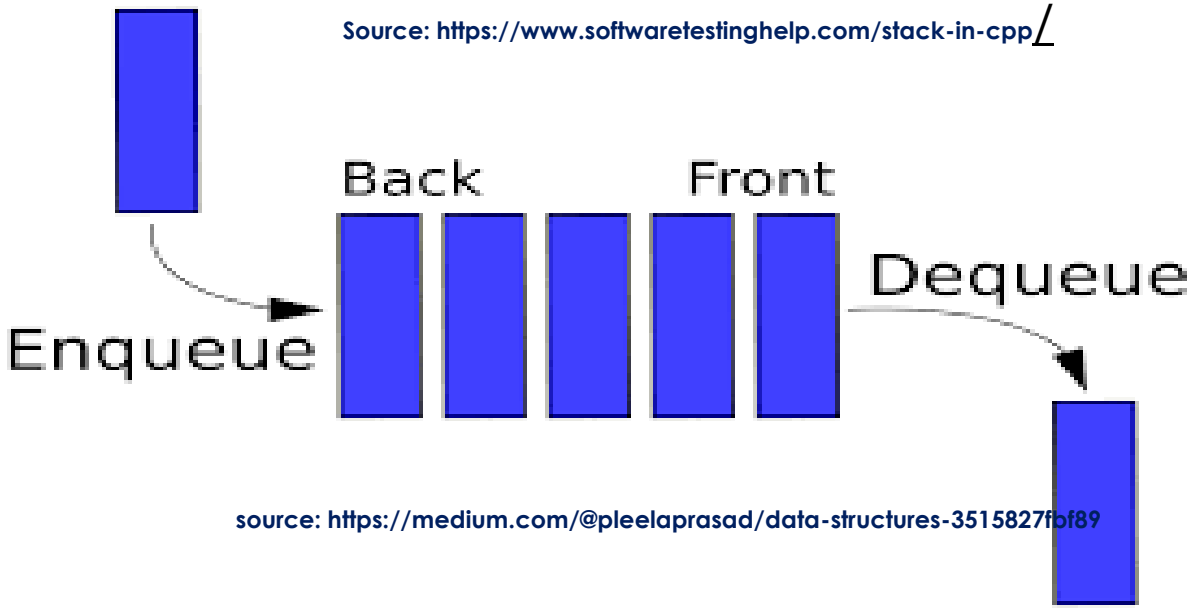
One-dimensional array with six elements



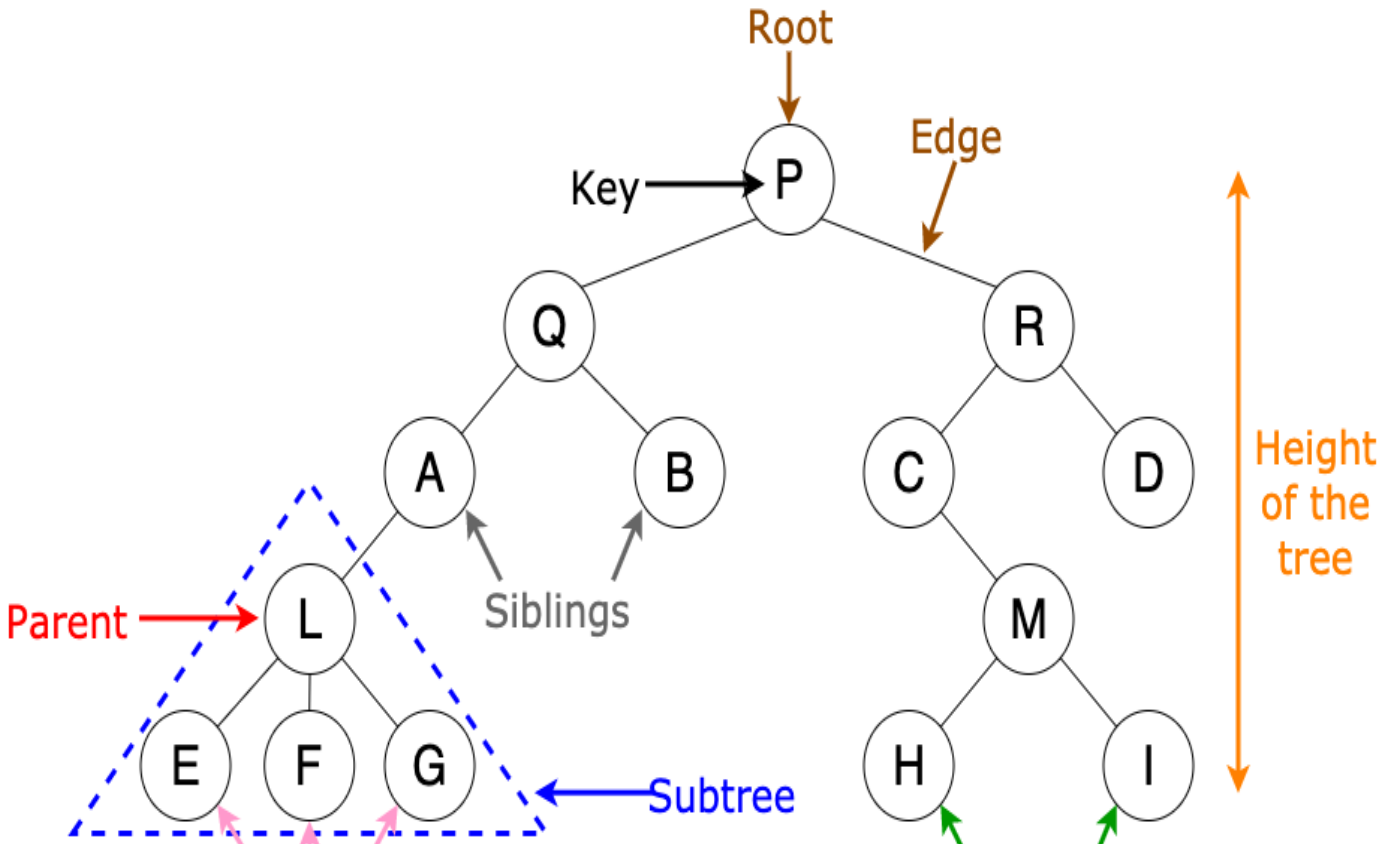
Source: https://www.softwaretestinghelp.com/stack-in-cpp/



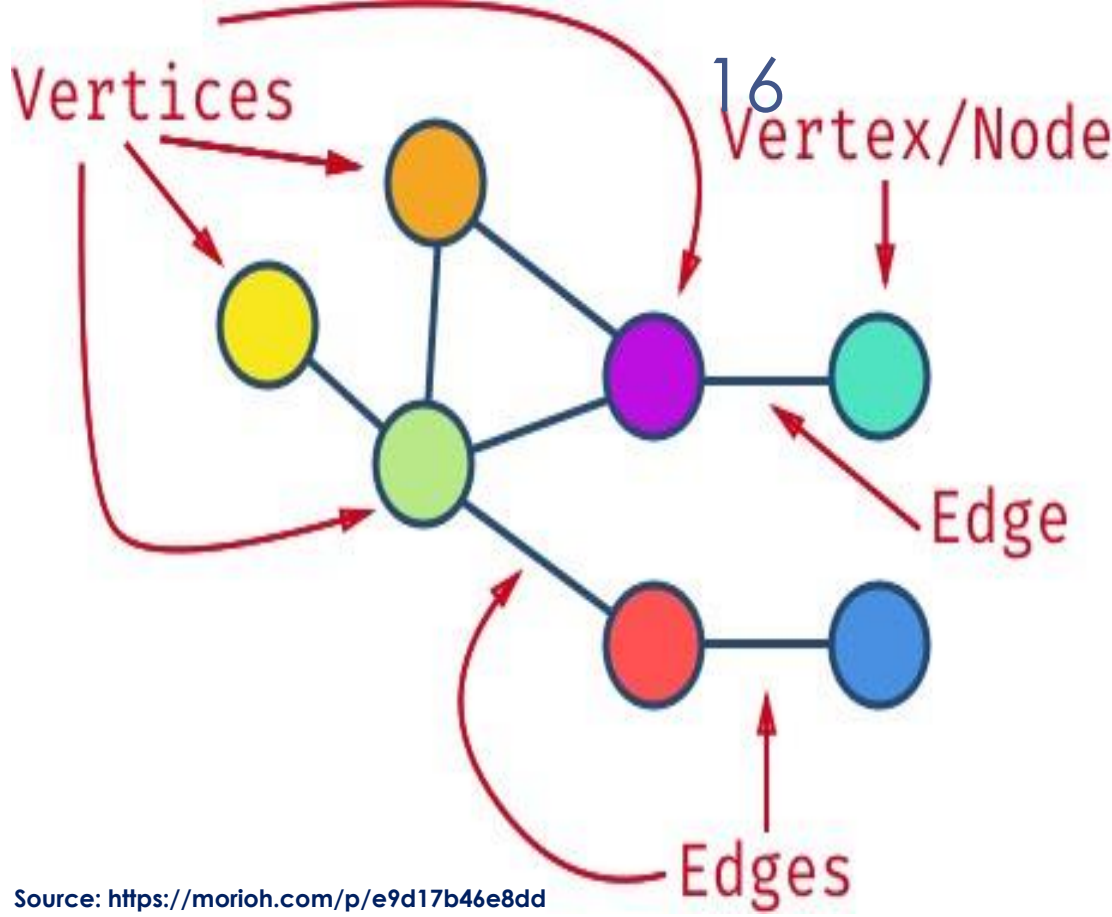
Source: https://www.geeksforgeeks.org/data-structures/linked-list/



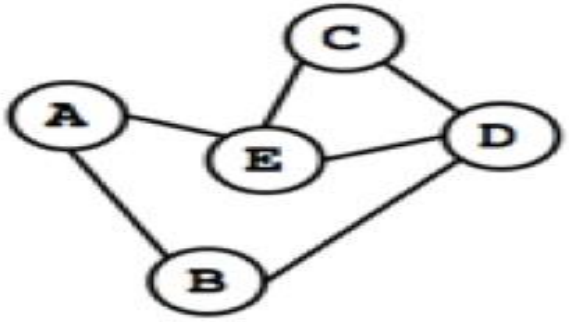
source: https://medium.com/@pleelaprasad/data-structures-3515827fbf89

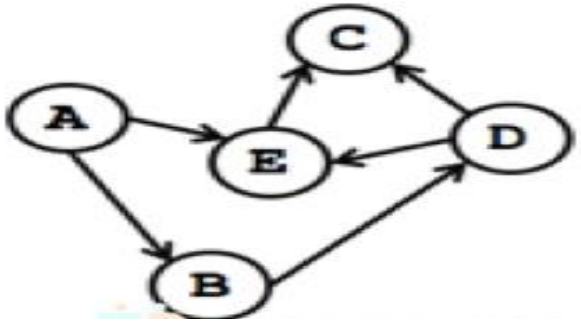


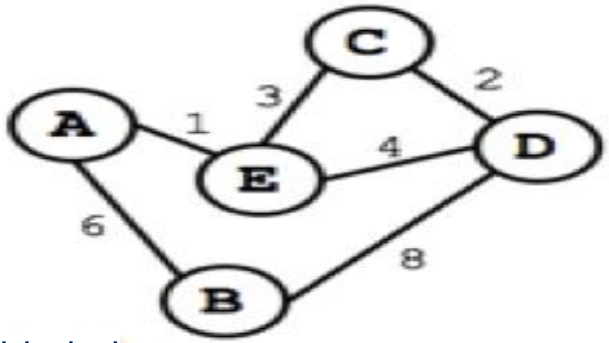
Source: https://towardsdatascience.com/8-useful-tree-data-structures-worth-knowing-8532c7231e8c



Source: https://morioh.com/p/e9d17b46e8dd







Source: https://dzone.com/articles/an-overview-to-working-of-graphs-in-data-structure

(A) Undirected Graph (B) Directed Graph (C) Weighted Graph (D) Self-Loop

Memory Allocation

- ✓ Memory allocation in programming is very important for storing values when you assign them to variables
- ✓ allocates memory for the variables declared by a programmer via the compiler
- ✓ allocation is done either before or at the time of program execution

Ways for memory allocation

- **Compile time allocation or static allocation of memory:**
  - ✓ The memory for named variables is allocated by the compiler.
  - ✓ The memory allocated by the compiler is allocated on the stack.
  - ✓ Exact size and storage must be known at compile time.
  - ✓ for array declaration, the size has to be constant.
- **Runtime allocation or dynamic allocation of memory:**
  - ✓ The memory is allocated at runtime.
  - ✓ The memory space is allocated dynamically within the program run.
  - ✓ Dynamically allocated memory is allocated on the heap.
  - ✓ The exact space or number of the data items does not have to be known by the compiler in advance.
  - ✓ Pointers play a major role for dynamic memory allocation.

Dynamic Memory De-Allocation

- Memory de-allocation is also an important part of this concept
- The "clean-up" of storage space is done for variables or for other data storage.
- It is the job of the programmer to de-allocate dynamically created space.
- For de-allocating dynamic memory, **delete** operator is used.
- In other words, dynamic memory Allocation refers to performing memory management for dynamic memory allocation manually.

Memory Parts in C++ Program

- **stack:**
  - ✓ All variables declared inside any function takes up memory from the stack.
- **heap:**
  - ✓ It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.

The “new” Operator

- To allocate space dynamically, unary operator *new is used* , followed by the type being allocated for memory.

✓new int; //dynamically allocates memory for an integer type

✓new double; // dynamically allocates memory an double type

✓new int[30]; // dynamically allocates memory for integer array

Allocating space using new

- int * ptr; // declares a pointer ptr
- ptr = new int; // dynamically allocate an int for loading the address in ptr
- int *ptr=new int; // can be clubbed together
- Int *ptr=new int(5); // initialized also to 5
- double * i; // declares a pointer i
- i = new double; // dynamically allocate a double and loading the address in I

In the above example,

✓ we have declared a pointer variable 'ptr' to integer and initialized it to null.

✓ Using the "new" operator memory will be allocate to the "ptr" variable.

The Delete Operator

- The memory allocated dynamically using the new operator has to be freed explicitly by the programmer. For this purpose, the "delete" operator is used
- **The general syntax of the delete operator is:**

```
delete pointer_variable;
```

- **So memory allocated to the ptr variable can be freed as:**

```
delete ptr;
```

- This statement frees the memory allocated to the variable "ptr" back to the memory pool.
- delete operator can also be used to free the memory allocated to arrays.

Dynamic memory allocation Programming Example:1

```
#include <iostream>

using namespace std;

 int main()

{

int* i= NULL;

i= new int;

*i= 5;

cout << "Value is : " << *i<< endl;

delete i;

}
```

Dynamic Memory Allocation for Arrays

- To allocate memory for an array of characters, i.e., a string of 50 characters. Using that same syntax, memory can be allocated dynamically
- char* str = NULL; // Pointer initialized with NULL
- str= new char[50]; // Dynamic Allocation will be done

- Here, new operator allocates 50 continuous elements of type characters to the pointer variable str and returns the pointer to the first element of str.

The Delete Operator

- **the memory allocated to the array str above can be freed as follows:**

delete[] str;

- Note the subscript operator used with the delete operator.
- This is because, as we have allocated the array of elements, we need to free all the locations.
- **Instead, if the following statement had executed:**

delete str;

- the above statement will only delete the first element of the array.
- Using subscript "[]" all the memory allocated is to be freed.

Example 2 : dynamic memory allocation

```
#include <iostream>
#include <string>
using namespace std;
  int main()
 {
   int *p=new int;
   int *a= new int(10);

   if(!p)
   {
        cout<<"bad memory allocation"<<endl;
    }
   else
  {
       cout<<"memory allocated successfully"<<endl;
       *p= 5;
       cout<<"*p= "<<*p<<endl;
       cout<<"*a= "<<*a<<endl;
}
```

```
double *arr= NULL;
arr= new double[10];

if(!arr)
 {cout<<"memory not allocated"<<endl;}
else
   {
     for(int i=0;i<5;i++)
       arr[i] = i+1;
       cout<<“Array values : ";
     for(i=0;i<5;i++)
       cout<<arr[i]<<"\t";
   }
delete p;
delete a;
delete[] arr;

return 0;

}
```

Dynamic Memory Allocation for Linked List

Let us take a structure of a linked list node:

```
struct node
{
 int data;
node *next;
 };
node *temp=new  node; // dynamic memory allocation
```

- ✓ Memory is allocated required for a node by the **new** operator.
- ✓ 'temp' points to a node (or space allocated for the node).

Example: Linked List using new operator


```
#include <iostream>
using namespace std;
struct node
{
    int data;
    node *next;
};
class linked_list
{
private:
    node *head,*tail;
public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }

    void add_node(int n)
    {
        node *tmp = new node;
        tmp->data = n;
        tmp->next = NULL;
        if(head == NULL)
                {
                        head = tmp;
                        tail = tmp;
                }
                else
                {
                    tail->next = tmp;
                    tail = tail->next;
                }
        }
};
int main()
{
    linked_list a;
    a.add_node(1);
    a.add_node(2);
    return 0;
}
```

Source: https://www.codesdope.com/blog/article/c-linked-lists-in-c-singly-linked-list/

Program Output

```
memory allocated successfully
*i=  5
*a=  10
Array values:1    2    3    4    5
```

The major differences between static and dynamic memory allocations

Static Memory Allocation	Dynamic Memory Allocation
Memory Allocates variables permanently	Memory Allocates variables only if program unit gets active
Memory Allocation is done before program execution	Memory Allocation is done during program execution
Use stack data structure for implementation	Use heap data structure for implementation
Less efficient	More efficient
Memory not reusable	memory reusable and can be freed when not required

https://www.w3schools.in/difference-in-static-and-dynamic-memory-allocation/

References

- Robert Lafore, Object Oriented Programming in C++, SAMS, 2002
- https://www.softwaretestinghelp.com/new-delete-operators-in-cpp/
- https://www.softwaretestinghelp.com/stack-in-cpp/
- https://www.geeksforgeeks.org/
- https://www.tutorialspoint.com/abstract-data-type-in-data-structures
- Richard F. Gilberg & Behrouz A. Forouzan, Data Structures, A pseudocode approach with C, CENGAGE learning, 2008